DATA STRUCTURES AND ALGORITHMS

Textbook:

Fundamentals of Data Structure in C++, Silicon Press, 2006

Instructor:

金远平 ypjin@seu.edu.cn

Teaching assistants:

吴玉林

852955268@qq.com,13255266805

胡孔涛

1821141394@qq.com,15295031250

陈遥

326907403@qq.com,18862182610

刘秀美

1092127125@qq.com,18366155036

Total class hours: 64

Total lab. hours: 16

71153+71154+71Y15 on Wednesday,

71151+71152 on Thursday evening

6:30pm-10:00pm of week 5, 8, 11, 14.

At Room 262，235, Computer Building

Home work:

- Should be handed to teaching assistants on Wednesday of week 4, 6, 8, 10, 12, 14, 16
- All 71153+04015244 to 吴玉林
- All 71154+71Y15 to 胡孔涛
- All 71151 to 陈遥
- All 71152 to 刘秀美

Evaluation:

Course Attendance: 10%,

Exercises and Projects: 20%,

Final Examination (Textbook and Course Notes allowed): 70%

Chapter 1
Basic Concepts

- Software system needs to model objects
- Data structures
- Elements relations
- Data operations
- Implemented level by level
- Study results for middle level: list, set, tree, graph.....

- **Provide the tools and techniques necessary to design and implement large-scale software systems, including:**

• Data abstraction and encapsulation

• Algorithm specification and design

• Performance analysis and measurement

• Recursive programming

1.1 Overview: System Life Cycle

(1) Requirements
specifications, purpose, input, output

(2) Analysis
break the problem into manageable pieces, bottom-up, top-down

(3) Design
data objects, operations on them, abstract data type, algorithm specification and design

(4) Refinement and coding
representations for data object, algorithms for operations , components reuse

(5) Verification and maintenance
testing, error removal, update

1.3 Data Abstraction and Encapsulation

Definition: Data Encapsulation or information Hiding is the concealing of the implementation details of a data object from the outside world.

Definition: Data Abstraction is the separation between the *specification* of a data object and its *implementation*.

Mobile phone example.

Definition: A Data Type is a collection of *objects* and a set of *operations* that act on those objects.

predefined and user-defined:
char, int, arrays, structs, classes.

Definition: An Abstract Data Type (ADT) is a data type with the specification of the objects and the specification of the operations on the objects being separated from the representation of the objects and the implementation of the operations.

Benefits of data abstraction and data encapsulation:

(1) Simplification of software development

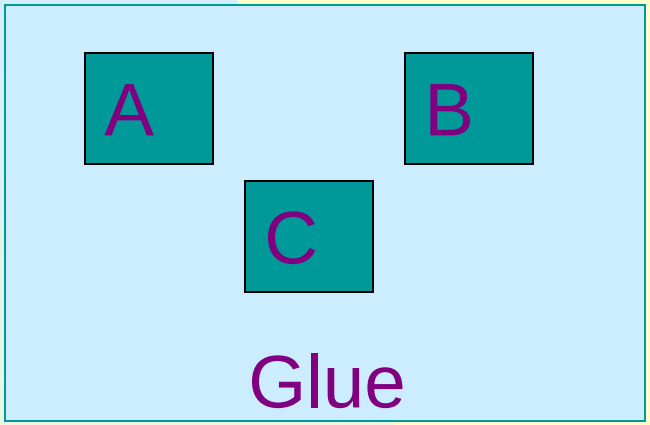
data types A, B, C + Glue

(a) a team of 4 programmers

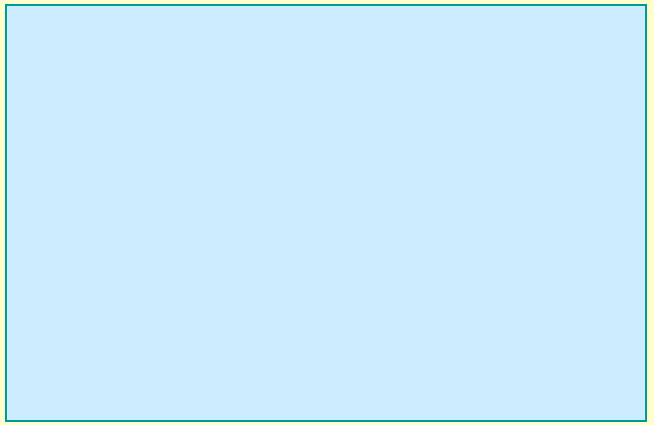
(b) a single programmer

(2) Testing and debugging

Code with data abstraction



Code without data abstraction



Unshaded areas represent code to be searched for bugs.

(3) Reusability

data structures implemented as distinct entities of a software system

(4) Modifications to the representation of a data type

a change in the internal implementation of a data type will not affect the rest of the program as long as its interface does not change.

1.5 Algorithm Specification

1.5.1 Introduction

Definition: An algorithm is finite set of instructions that, if followed, accomplishes a particular task.

Must satisfy the following criteria:

(1) Input Zero or more quantities externally supplied.

(2) Output At least one quantity is produced.

(3) Definiteness Clear and unambiguous.

(4) **Finiteness** Terminates after a finite number of steps.

(5) **Effectiveness** Basic enough, feasible

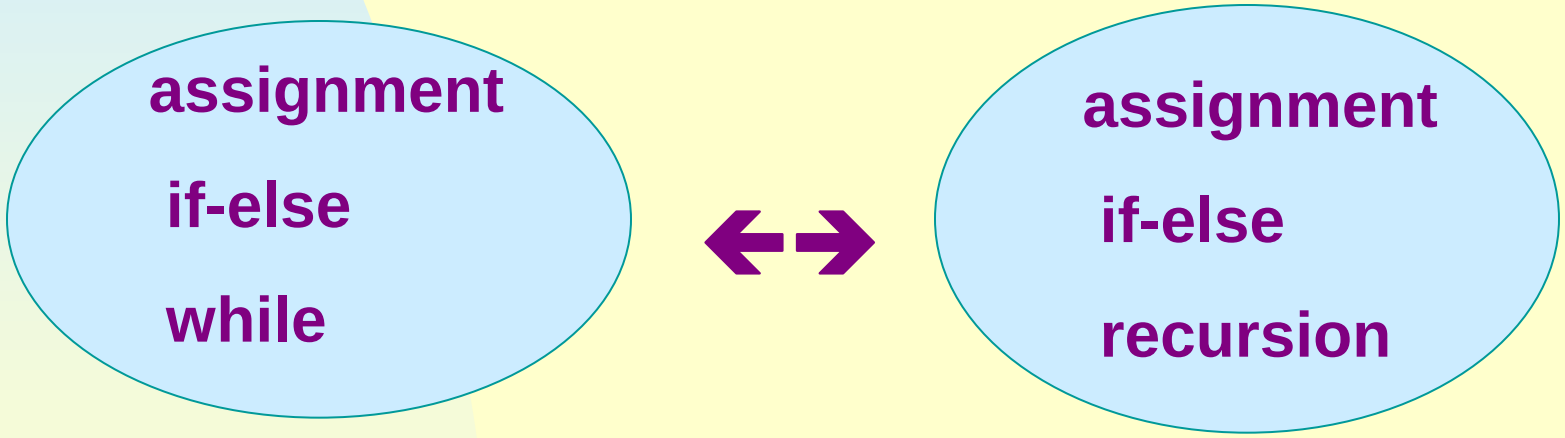
Compare: algorithms and programs

1.5.2 Recursive Algorithms

Direct recursion: functions call themselves.

Indirect recursion: call other functions that again invoke the calling function.

Expressiveness:



Recursion is especially appropriate for recursively defined problems.

Example 1.4 *Recursive binary search*

```
int BinarySearch (int *a, const int x,const int left, const int
 right)
// search the sorted array a[left:right] for x
{
 if (left <=right) {
    int middle=(left+right)/2;
    if (x < a[middle]) return BinarySearch(a,x,left,middle-1);
    else if (x > a[middle])
            return BinarySearch(a,x,middle+1,right);
    return middle; //x==a[middle]
```

```
  }  // end of if
   return –1;  // not found
}
```

To invoke, use: BinarySearch(a,x,0,n-1)

Example 1.5 Permutation Generator

Problem: for a set of $n \geq 1$ elements, print all possible permutations of it.

Solution:

look at (a,b,c), the answer is by printing:

(1) a followed by all permutations of (b,c)

(2) b followed by all permutations of (a,c)

(3) c followed by all permutations of (b,a)

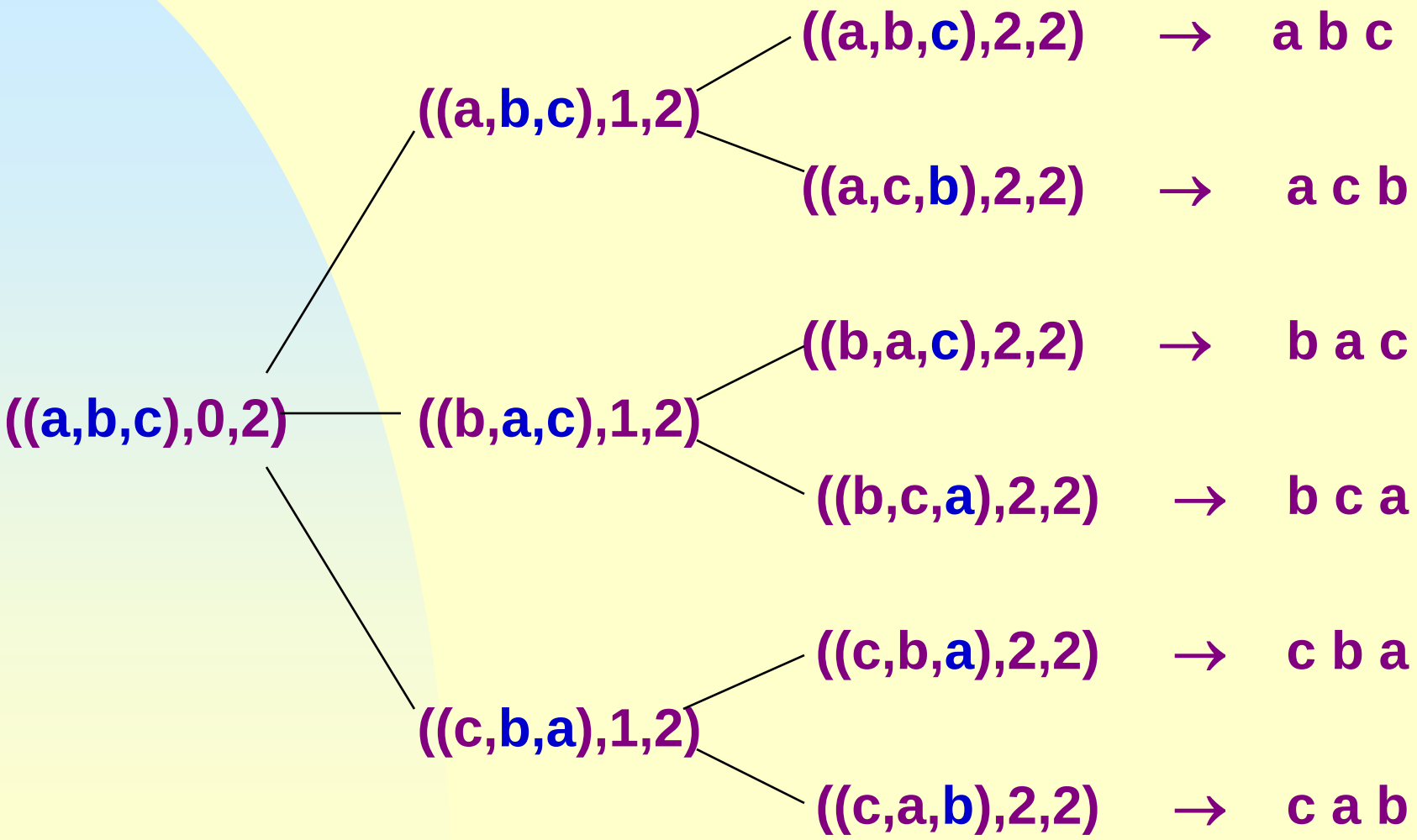
the clue to recursion: we can solve the problem for a set of n elements if we have an algorithm that works on n-1 elements.

```
void Permutations (char *a, const int k,const int m)
// generate all permutations of a[k],…,a[m]
{
   if (k ==m) {     // output permutation
      for (int i=0; i<=m; i++)  cout << a[i] << " ";
      cout << endl;
   }
```

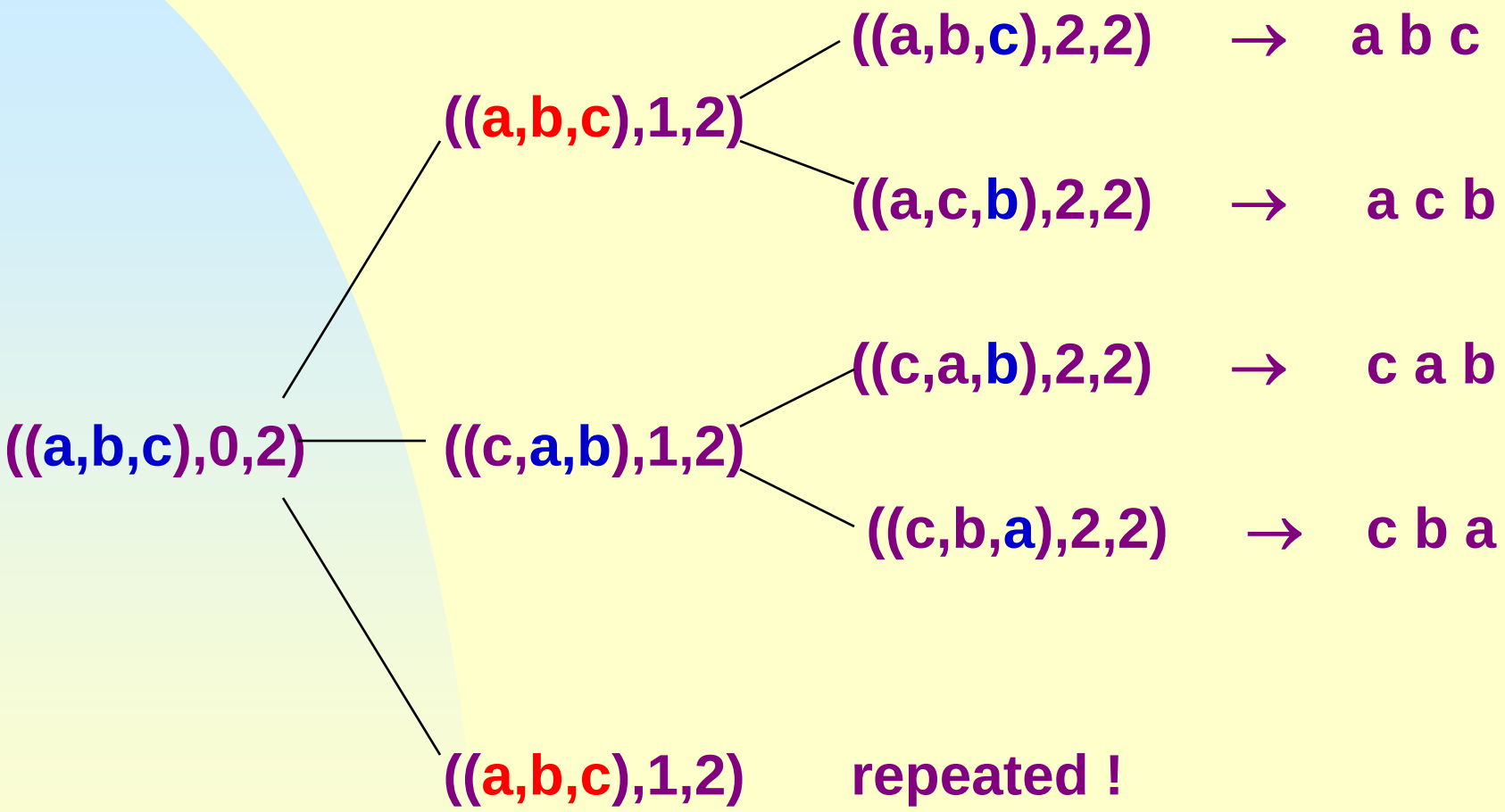
```
    else {    //  a[k:m] has more than one permutation.
                //  generate recursively
        for (i=k; i<=m; i++) {
            swap(a[k], a[i]); // interchange a[k] and a[i]
            Permutations(a,k+1,m); // all permutations of a[k+1:m]
            swap(a[k], a[i]);  // to return to the original configuration
                                  // important
        }
}
```

To invoke: Permutation(a,0,n-1)

Try it with n=3 and a[0:2]=(a,b,c)



If the original configuration were not returned,



and the b x x permutations would be lost.

Exercises: P32-2, P33-14

1.7 Performance Analysis and Measurement

Definition: the Space complexity of a program is the amount of memory it needs to run to completion. The Time complexity of a program is the amount of computer time it needs to run to completion.

(1) Priori estimates --- Performance analysis

(2) Posteriori testing--- Performance measurement

1.7.1 Performance Analysis

1.7.1.1 Space complexity

The space requirement of program P:

$$S(P)=c+S_P(\text{instance characteristics})$$

We concentrate solely on S_P.

Example 1.10

```
float  Rsum (float *a, const int n) //compute Σ_{i=0}^{n-1} a[i] recursively
{
     if (n <=0) return 0;
     else return (Rsum(a,n-1)+a[n-1]);
}
```

The instances are characterized by n, each call requires 4 words (n, a, return value,return address), the depth of recursion is n+1, so

$$S_{rsum}(n) = 4(n+1)$$

1.7.1.2 Time complexity

Run time of a program P:

$$t_P(\text{instance characteristics})$$

A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of instance characteristics.

In P41-43 of the textbook, there is an detailed assignment of step counts to statements in C++.

Our main concern is on how many steps are needed by a program to solve a particular problem instance?

2 ways:

(1) count

(2) table

Example 1.12

```
count=0;
float  Rsum (float *a, const int n)
{
      count++; // for if
     if (n <=0) {
        count++; // for return
        return 0;
       }
      else {
        count++; // for return
        return (Rsum(a,n-1)+a[n-1]);
       }
}
```

$t_{Rsum}(0) = 2,$

$t_{Rsum}(n) = 2 + t_{Rsum}(n-1)$

$= 2+2+ t_{Rsum}(n-2)$

.

.

.

$= 2n+ t_{Rsum}(0)=2n+2$

Example 1.14 Fibonnaci numbers

```
1 void  Fibonnaci (int n)
2 {   // compute the Fibonnaci number Fn
3     if (n <=1) cout << n<< endl; { // F0=0 and F1 =1
4     else { // compute Fn
5         int fn; int fnm2=0; int fnm1=1;
6         for (int i=2; i<=n; i++)
7         {
8             fn=fnm1+fnm2;
9             fnm2=fnm1;
10            fnm1=fn;
11        } //end of for
```

```
12        cout <<fn<<endl;
13    }  //end of else
14 }
```

Let us use a table to count its total steps.

Line	s/e	frequency	total steps
1	0	1	0
2	0	1	0
3	1 (n >1)	1	1
4	0	1	0
5	2	1	2
6	1	n	n
7	0	n-1	0
8	1	n-1	n-1
9	1	n-1	n-1

10	1	n-1	n-1
11	0	n-1	0
12	1	1	1
13	0	1	0
14	0	1	0

So

for n>1, $t_{Fibonnci}(n)=4n+1$,

for n=0 or 1, $t_{Fibonnci}(n)=2$

Sometime, the instance characteristics is related with the content of the input data set.

e.g., *BinarySearch*.

Hence:

- **best-case**
- **worst-case,**
- **average-case.**

1.7.1.3 Asymptotic Notation

Because of the inexactness of what a step stands for, we are mainly concerned with the magnitude of the number of steps.

Definition [O]: f(n)=O(g(n)) iff there exist positive constants c and n_0 such that $f(n) \leq c\, g(n)$ for all n, $n>n_0$.

Example 1.13: $3n+2=O(n)$, $6*2^n+n^2=O(2^n)$,…

Note g(n) is an upper bound.

$n=O(n^2)$, $n= O(2^n)$, …, for f(n)=O(g(n)) to be informative, g(n) should be as small as possible.

In practice, the coefficient of g(n) should be 1. We never say O(3n).

O(1) --- constant

O(n) --- linear

$O(n^2)$ --- quadratic

$O(n^3)$ --- cubic

$O(\log_2 n)$ --- logarithm

$O(2^n)$ --- exponential

Theory 1.2: if $f(n)=a_m n^m+\ldots+ a_1 n+a_0$, then $f(n)=O(n^m)$.

When the complexity of an algorithm is actually, say, O(log n), but we can only show that it is O(n) due to the limitation of our knowledge, it is ok to say so. This is one benefit of O notation as upper bound.

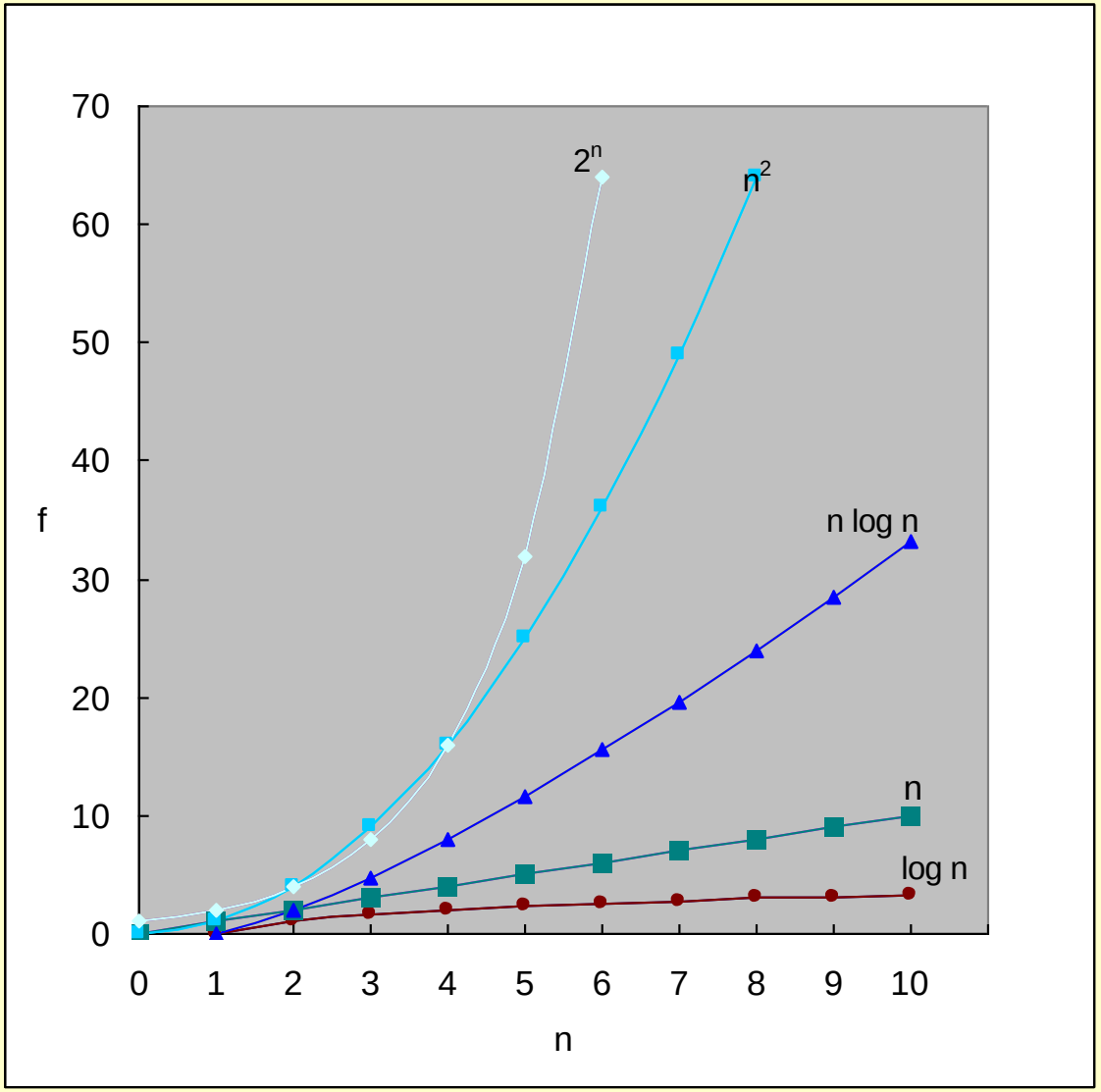
Self-study:

Ω --- low bound

Θ --- equal bound

1.7.1.4 Practical Complexity

The right figure shows how the various functions grow with n.



n	f(n)=n	f(n)=nlog$_2$n	f(n)=n^2	f(n)=n^4	f(n)=n^{10}	f(n)=2^n
10	.01μs	.03 μs	.1 μs	10 μs	10s	1 μs
20	.02 μs	.09 μs	.4 μs	160 μs	2.84h	1 ms
30	.03 μs	.15 μs	.9 μs	810 μs	6.83d	1 s
40	.04 μs	.21 μs	1.6 μs	2.56ms	121d	18m
50	.05 μs	.28 μs	2.5 μs	6.25ms	3.1y	13 d
100	.1 μs	.66 μs	10 μs	100 ms	3171y	4*10^{13}y
10^3	1 μs	9.66 μs	1ms	16.67m		
10^4	10 μs	130 μs	100ms	115.7d		
10^5	100 μs	1.66ms	10s	3171y		

Table 1.8: Times on a 1-billion-steps-per-second computer

1.7.2 Performance Measurement

Performance measurement is concerned with obtaining the actual space and time requirements of a program.

To time a short event it is necessary to repeat it several times and divide the total time for the event by the number of repetitions.

Let us look at the following program:

```
int  SequentialSearch (int *a, const int n, const int x )
{ // Search a[0:n-1].
    int  i;
    for (i=0; i < n && a[i] != x; i++);
    if (i == n) return -1;
    else return i;
}
```

```
void TimeSearch ( )
{
   int a[1000], n[20];
   const long r[20] = {300000, 300000, 200000, 200000,
   100000, 100000, 100000, 80000, 80000, 50000, 50000,
    25000, 15000, 15000, 10000, 7500, 7000, 6000, 5000,
    5000 };

   for ( int j=0; j<1000; j++ ) a[j] = j+1; //initialize a
   for ( j=0; j<10; j++ ) {  //values of n
       n[j] = 10*j;  n[j+10] = 100*( j+1 );
    }

   cout << " n    total    runTime" << endl;
```

```
   for ( j=0; j<20; j++ ) {
      long start, stop;
      time (&start);                                  // start timer
      for ( long b=1; b<=r[j]; b++ )
            int k = seqsearch(a, n[j], 0 );  //unsuccessful search
      time (&stop);                                   // stop timer
      long totalTime = stop - start;
      float runTime =  (float) (totalTime) / (float)(r[j]);
      cout << " " << n[j] << " " << totalTime << " " << runTime
            << endl;
    }
}
```

The results of running *TimeSearch* are as in the next slide.

n	total	runTime	n	total	runTime
0	241	0.0008	100	527	0.0105
10	533	0.0018	200	505	0.0202
20	582	0.0029	300	451	0.0301
30	736	0.0037	400	593	0.0395
40	467	0.0047	500	494	0.0494
50	565	0.0056	600	439	0.0585
60	659	0.0066	700	484	0.0691
70	604	0.0075	800	467	0.0778
80	681	0.0085	900	434	0.0868
90	472	0.0094	1000	484	0.0968

Times in hundredths of a second, the plot of the data can be found in Fig. 1.7.

Issues to be addressed:

(1) Accuracy of the clock

(2) Repetition factor

(3) Suitable test data for worst-case or average performance

(4) Purpose: comparing or predicting?

(5) Fit a curve through points

Exercises:

P72-10

Chapter 2
Arrays

2.2 The Array as an Abstract Data Type

Array:

- **A set of pairs: <index, value>**

(correspondence or mapping)

- **Two operations: retrieve, store**

Now we will use the C++ class to define an ADT.

ADT2.1 GeneralArray

```
class  GeneralArray {
//  a set of pairs <index, value> where for each value of
//  index in IndexSet there is a value of type float. IndexSet is
//  a finite ordered set of one or more dimensions.
public:
    GeneralArray(int j, RangeList list, float initValue =
                                                  defaultValue);
    // The constructor GeneralArray creates a j
    // dimensional array of floats; the range of the kth
    // dimension is given by the kth element of list.
    // For all i∈IndexSet, insert <i, initValue> into the array.
```

```
    float  Retrieve(index i);
    // if (i∈IndexSet) return the float associated with i in the
    //  array;else throw an exception.

    void Store(index i, float x);
    // if (i∈IndexSet) replace the old value associated with i
    // by x;  else throw an exception.
};  //end of GeneralArray
```

Note:

- **Not necessarily implemented using consecutive memory**
- **Index can be coded any way**
- **GeneralArray is more general than C++ array as it is more flexible about the composition of the index set**
- **To be simple, we will hereafter use the C++ array**

2.3 The Polynomial Abstract Data Type

Array can be used to implement other abstract data types. The simplest one might be:

Ordered or linear list.

Example:

(Sun, Mon, Tue, Wed, Thu, Fri, Sat)

(2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A)

( ) *// empty list*

More generally, An ordered list is either empty or $(a_0, a_1,.., a_{n-1})$. // index important

Main operations:

(1) Find the length, n, of the list.

(2) Read the list from left to right (or right to left)

(3) Retrieve the ith element, $0 \le i < n$.

(4) Store a new value into the ith position, $0 \le i < n$.

(5) Insert a new element at position i, $0 \le i < n$, causing elements numbered i, i+1,…n-1 to become numbered i+1, i+2,…n.

(6) Delete the element at position i, $0 \leq i < n$, causing elements numbered i+1, i+2,…n-1 to become numbered i, i+1,…n-2.

How to represent ordered list efficiently?

- **Use array: a_i ←→index i**
- **Sequential mapping, because using conventional array representation, we are storing a_i and a_{i+1} into consecutive location i and i+1.**
- **Random access any element in O(1).**
- **Operations (5) and (6) need data movement.**

Now let us look at a problem requiring ordered list.

Problem: build an ADT for the representation and manipulation of symbolic polynomials.

$A(x)=3x^2+ 2x+4$

$B(x)=x^4+ 10x^3+ 3x^2+1$

Degree: the largest exponent

ADT 2.3 Polynomial

```
class Polynomial {
   //  p(x)=a0x^e0+,…,+ anx^en ; a set of ordered pairs of <ei, ai>,
   //  where ai is a nonzero float coefficient and ei is a
   // non-negative exponent
public:
   Polynomial ( );
   // Construct the polynomial p(x)=0
```

```
    void AddTerm (Exponent e, Coefficient c);
    // add the term <e,c> to *this, so that it can be initialized

    Polynomial Add (Polynomial poly);
    // return the sum of the polynomials *this and poly

    Polynomial Mult (Polynomial poly);
    // return the product of the polynomials *this and poly

    float Eval ( float f);
    // evaluate polynomial *this at f and return the result
}
```

2.3.1 Polynomial Representation

Representation 1

private:

```
int degree;  // degree ≤ MaxDegree
float coef[MaxDegree+1];
```

Let a be $A(x)=a_n x^n + a_{n-1}x^{n-1}+,\ldots,+ a_1 x+ a_0$

a.degree=n;

a.coef[i] = a_{n-i}, $0 \le i \le n$

Simple algorithms for many operations.

Representation 2

When a.degree << MaxDegree, representation 1 is very poor in memory use. To improve, define variable sized data member as:

```
private:
   int degree;
   float *coef;

Polynomial::Polynomial(int d)
 {
   int degree=d;
   coef= new float[degree+1];
 }
```

Representation 3

Representation 2 is still not desirable. For instance, $x^{1000}+1$ makes 999 entries of the coef be zero.

So, we store only the none zero terms:

$$A(x) = b_m x^{em} + b_{m-1} x^{em-1} + \ldots + b_0 x^{e0}$$

Where $b_i \neq 0$, $e_m > e_{m-1} > \ldots > e_0 \geq 0$

```
class Polynomial; // forward declaration
class Term {
friend Polynomial;
private:
    float coef; // coefficient
    int exp;     // exponent
};

class Polynomial {
public:
   ……
private:
   Term *termArray;
   int capacity; // size of termArray
   int terms; // number of nonzero terms
}
```

For $A(x) = 2x^{1000} + 1$

A.termArray looks like:

coef	2	1		
exp	1000	0		

Many zero **--- good**

Few zero **--- not very good, may use twice as much space as in presentation 2.**

2.3.2 Polynomial Addition

Use presentation 3 to obtain C = A +B.

Idea:

Because the exponents are in descending order, we can adds A(x) and B(x) term by term to produce C(x).

The terms of C are entered into its termArray by calling function NewTerm.

If the space in termArray is not enough, its capacity is doubled.

```
1 Polynomial  Polynomial::Add (Polynomial b)
2 {   // return the sum of the polynomials *this and b.
3   Polynomial c;
4   int aPos=0, bPos=0;
5   while (( aPos < terms) && (bPos < b.terms))
6      if (termArray[aPos].exp==b.termArray[bPos].exp) {
7         float t = termArray[aPos].coef + termArray[bPos].coef
8         if ( t ) c.NewTerm (t, termArray[aPos].exp);
9         aPos++;  bPos++;
10     }
11     else if  (termArray[aPos].exp < b.termArray[bPos].exp) {
12  c.NewTerm (b.termArray[bPos].coef, b.termArray[bPos].exp);
13       bPos++;
14     }
```

```
15    else {
16       c.NewTerm (termArray[aPos].coef, termArray[aPos].exp);
17        aPos++;
18     }
19   // add in the remaining terms of *this
20   for ( ; aPos < terms; aPos++ )
21      c.NewTerm(termArray[aPos].coef, termArray[aPos].exp );
22   // add in the remaining terms of b
23   for ( ; bPos < b.terms; bPos++ )
24   c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
25   return c;
26 }
```

```
void Polynomial::NewTerm(const float theCoeff,
                         const int theExp)
{ // add a new term to the end of termArray.
   if (terms == capacity)
   { // double capacity of termArray
      capacity *= 2;
      term *temp = new term[capacity]; // new array
      copy(termArray, termAarry + terms, temp);
      delete [ ] termArray;  // deallocate old memory
      termArray = temp;
   }
  termArray[terms].coef = theCoeff;
  termArray[terms++].exp = theExp;
}
```

Analysis of Add:

Let m, n be the number of nonzero terms in a and b respectively.

- line 3 and 4---O(1)
- in each iteration of the while loop, aPos or bPos or both increase by 1, the number of iterations of this loop $\leq$ m+n-1
- if ignore the time for doubling the capacity, each iteration takes O(1)
- line 20--- O(m), line 23--- O(n)

Asymptotic computing time for Add: O(m+n)

Analysis of doubling capacity:

- the time for doubling is linear in the size of new array
- initially, c.capacity is 1
- suppose when Add terminates, c.capacity is 2^k
- the total time spent over all array doubling is

$$O\left(\sum_{i=1}^{k} 2^i\right) = O(2^{k+1}) = O(2^k)$$

- since c.terms > 2^{k-1} and m + n ≥ c.terms, the total time for array doubling is

$$O(\text{c.terms}) = O(m + n)$$

- so, even consider array doubling, the total run time of Add is O(m + n).
- experiments show that array doubling is responsible for very small fraction of the total run time of **Add**.

Exercises: P93-2,6, P94-9

2.4 Sparse Matrices

2.4.1 Introduction

A general matrix consists of m rows and n columns (m × n)of numbers, as:

	0	1	2
0	-27	3	4
1	6	82	-2
2	109	-64	11
3	12	8	9
4	48	27	47

Fig.2.2(a) 5×3

	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	0	0
2	0	0	0	-6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

Fig. 2.2(b) 6×6

A matrix of m × m is called a square.

A matrix with many zero entries is called sparse.

Representation:

- **A natural way --- a[m][n], access element by a[i][j], easy operations. But for sparse matrix, wasteful of both memory and time.**
- **Alternative way --- store nonzero elements explicitly. 0 as default.**

ADT 2.4 SparseMatrix

```
class SparseMatrix
{ // a set of <row, column, value>, where row, column are
  // non-negative integers and form a unique combination;
  //  value is also an integer.
public:
    SparseMatrix ( int r, int c, int t);
    // creates a r×c SparseMatrix with a capacity of t nonzero
    // terms
    SparseMatrix Transpose ( );
    // return the SparseMatrix obtained by transposing *this
    SparseMatrix  Add ( SparseMatrix b);
    SparseMatrix  Multiply ( SparseMatrix b);
};
```

2.4.2 Sparse Matrix Representation

Use triple <row, col, value>, sorted in ascending order by <row, col>.

We need also the number of rows and the number of columns and the number of nonzero elements. Hence,

```
class  SparseMatrix;
class  MatrixTerm {
friend class SparseMatrix;
private:
    int row, col, value;
};
```

And in class SparseMatrix:

```
private:
    Int rows, cols, terms, capacity;
    MatrixTerm *smArray;
```

Now we can store the matrix of Fig.2.2 (b) as Fig.2.3 (a).

	row	col	value
smArray[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

Fig.2.3 (a)

2.4.3 Transposing a Matrix

Transpose:

If an element is at position [i][j] in the original matrix, then it is at position [j][i] in the transposed matrix.

Fig.2.3(b) shows the transpose of Fig2.3(a).

	row	col	value
smArray[0]	0	0	15
[1]	0	4	91
[2]	1	1	11
[3]	2	1	3
[4]	2	5	28
[5]	3	0	22
[6]	3	2	-6
[7]	5	0	-15

Fig.2.3 (b)

First try:

For (each row i)

take element (i, j, value) and

store it in (j, i, value) of the transpose;

Difficulty: not knowing where to put (j, i, value) until all other elements preceding it have been processed.

Improvement:

For (all elements in column j)

store (i, j, value) of the original matrix as

(j, i, value) of the transpose;

Since the rows are in order, we will locate elements in the correct column order.

```
1 SparseMatrix SparseMatrix::Transpose ( )
2 { // return the transpose of *this
3     SparseMatrix b(cols, rows, terms);
4     if (terms > 0)
5     {   //nonzero matrix
6       int currentB = 0;
```

```
7       for ( int c=0; c<cols; c++ ) // transpose by columns
8         for ( int i=0; i<terms; i++ )
9         // find and move terms in column c
10          if ( smArray[i].col == c )
11          {
12            b.smArray[CurrentB].row = c;
13            b.smArray[CurrentB].col = smArray[i].row;
14            b.smArray[CurrentB++].value= smArray[i].value;
15          }
16    } // end of if (terms > 0)
17   return b;
18 }
```

Time complexity of Transpose:

- **line 7-15 loop--- cols times**
- **line 10 loop--- terms times**
- **other line--- O(1)**

Total time: O(cols* terms)

Additional space: O(1)

Think:

O(cols* terms) is not good. If terms = O(cols* rows) then it becomes O(cols^{2}* rows)---too bad! Since with 2-dimensional representation,

we can get an easy O(cols* rows) algorithm as:

```
for (int j=0;j < columns;j++)
    for (int i=0; i < rows; i++)  B[j][i] = A[i][j];
```

Further improvement:

If we use some more space to store some knowledge about the matrix, we can do much better: doing it in O(cols + terms).

- get the number of elements in each column of *this = the number of elements in each row of b;
- obtain the starting point in b of each of its rows;
- move the elements of *this one by one into their right position in b.

Now the algorithm **FastTranspose.**

```
1 SparseMatrix SparseMatrix::FastTranspose ( )
2 { // return the transpose of *this in O(terms+cols) time.
3    SparseMatrix b(cols, rows, terms);
4    if (terms > 0)
5    { // nonzero matrix
6       int *rowSize = new int[cols];
7       int *rowStart = new int[cols];
8       // compute rowSize[i] = number of terms in row i of b
9       fill(rowSize, rowSize + cols, 0);  // initialze
10    for (i=0; i<terms; i++ )  rowSize[smArray[i].col]++;
```

```
11    // rowStart[i] = starting position of row i in b
12     rowStart[0] = 0;
13    for (i=1;i<cols;i++) rowStart[i]=rowStart[i-1]+rowSize[i-1];
14    for (i=0; i<terms; i++)
15    {    // copy from *this to b
16         int j = rowStart[smArray[i].col];
17         b.smArray[j].row = smArray[i].col;
18         b.smArray[j].col = smArray[i].row;
19         b.smArray[j].value = smArray[i].value;
20         rowStart[smArray[i].col]++;
21     }   // end of for
```

```
22     delete [ ] rowSize;
23     delete [ ] rowStart;
24  }  // end of if
25  return b;
26 }
```

For Fig.2.3(a), after line 13, we get :

	[0]	[1]	[2]	[3]	[4]	[5]
RowSize=	2	1	2	2	0	1
RowStart=	0	2	3	5	7	7

Note the error in P101 of the text book!

Analysis:

3 loops:

- **line 10--- O(terms)**
- **line 13--- O(cols)**
- **line 14 – 21--- O(terms)**

  and line 9--- O(cols), other lines--- O(1)

Total: O(cols+terms)

This is a typical example for trading space for time.

Exercises: P107-1, 2, 4

2.6 The String Abstract data Type

A string $S = s_0, s_1, \ldots, s_{n-1}$,
where $s_i \in$ char, $0 \le i < n$, n is the length.

ADT 2.5 String

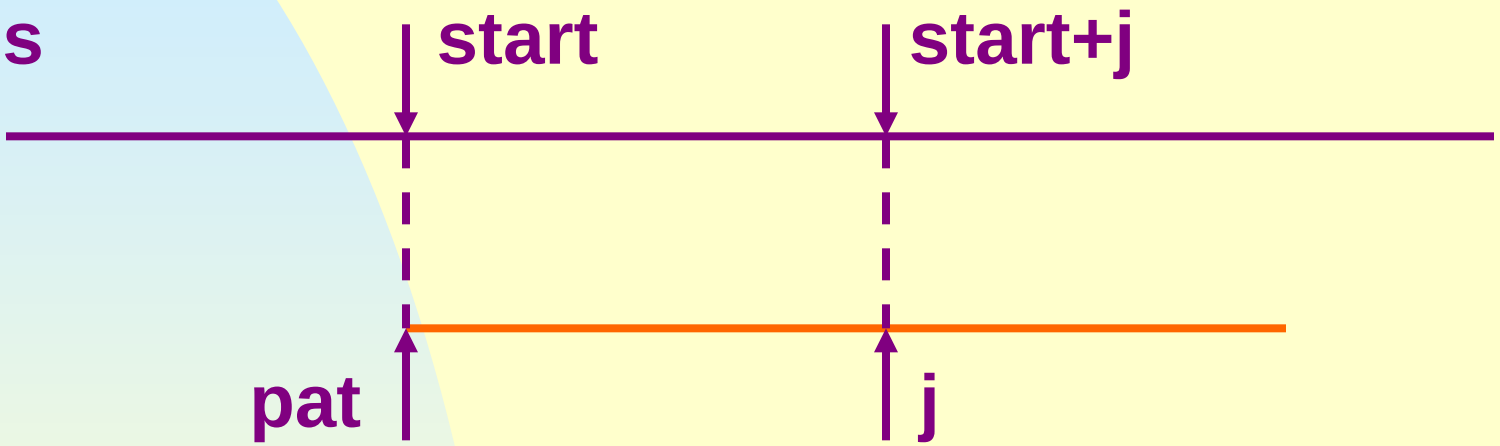
```
class String
{   // a finite set of zero or more characters;
public:
      String (char *init, int m );
      // initialize *this to string init of length m
```

```
    bool operator == (String t );
    // if *this equals t, return true else false.
    bool operator ! (  );
    // if *this is empty return true else false.
    int Length ( );
     // return the number of chars in *this
     String Concat (String t);
     String Substr (int  i, int  j);
     int Find (String pat);
    // return i such that pat matches the substring of *this that
    // begins at position i. Return –1 if pat is either empty or not
    // a substring of *this.
};
```

Assume the String class is represented by:

```
private:
    char* str;
```

2.6.1 String Pattern Matching: A Simple Algorithm



The idea is showed in the function Find .

```
int String::Find ( String pat )
{ // Return –1 if pat does not occur in *this; otherwise
  // return the first position in *this, where  pat begins.
    if (pat.Length( ) == 0) return -1; // pat is empty
    for (int start=0; start<=Length( ) - pat.Length( ); start++)
    { //  check  for match beginning at str[start]
       for (int j=0; j<pat.Length( )&&str[start+j]==pat.str[j];j++)
       if (j== pat.Length( )) return start; // match found
       // no match at position start
     }
     return -1;     // pat does not occur in s
}
```

The complexity of it is O(LengthP * LengthS).
Problem: rescanning.

Even if we check the last character of pat first, the time complexity can't be improved!

2.6.2 String Pattern Matching: The Knuth-Morris-Pratt Algorithm

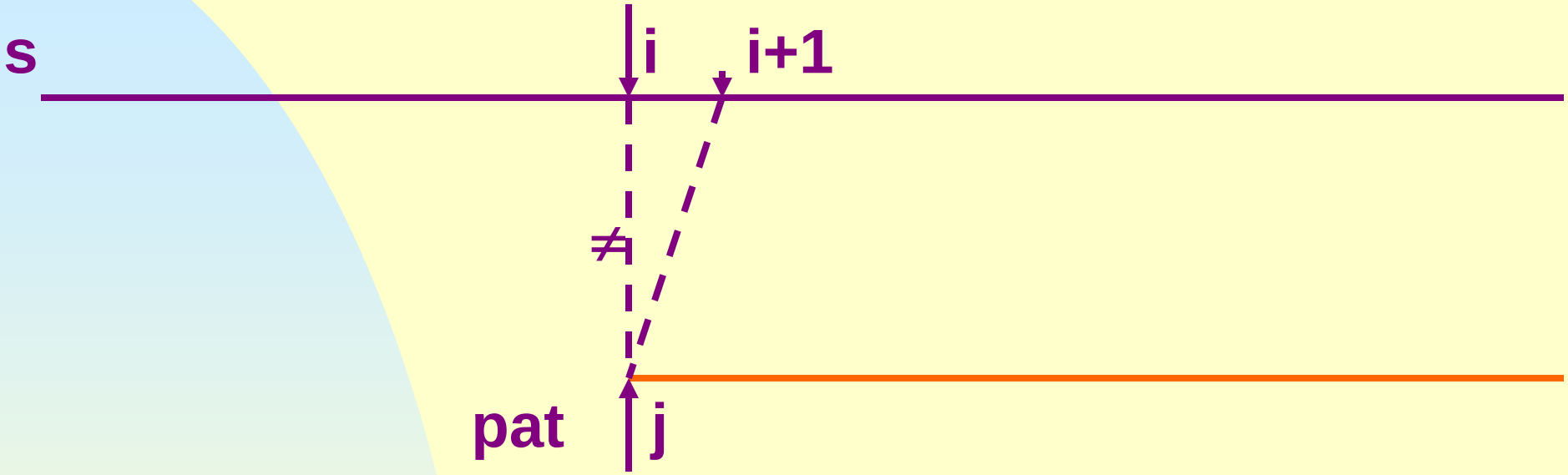
Can we get an algorithm which avoid rescanning the strings and works in O(LengthP + LengthS)?

This is optimal for this problem, as in the worst it is necessary to look at characters in the pattern and string at least once.

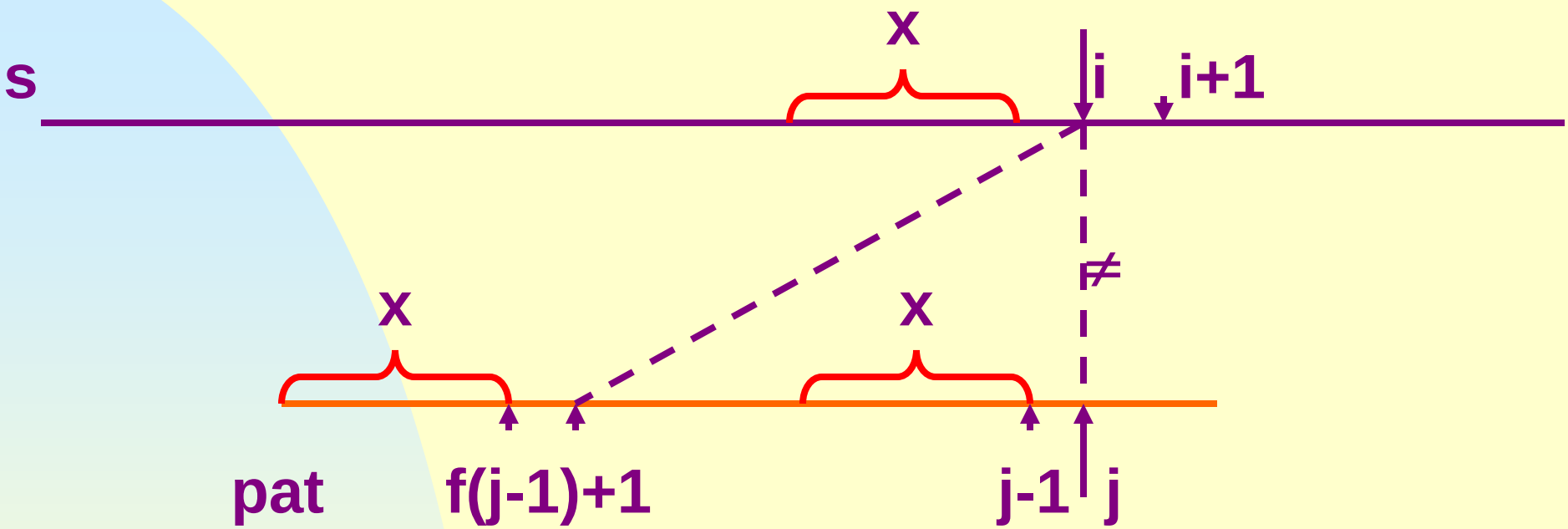
Basic Ideas:

- Rescanning to avoid missing the target --- too conservative. If we can go without rescanning, it is likely to do the job in O(LengthP + LengthS).
- Preprocess the pattern, to get some knowledge of the characters in it and the position in it, so that if a mismatch occurs we can determine where to continue the search and avoid moving backwards in the string.

Now we show details about the idea.

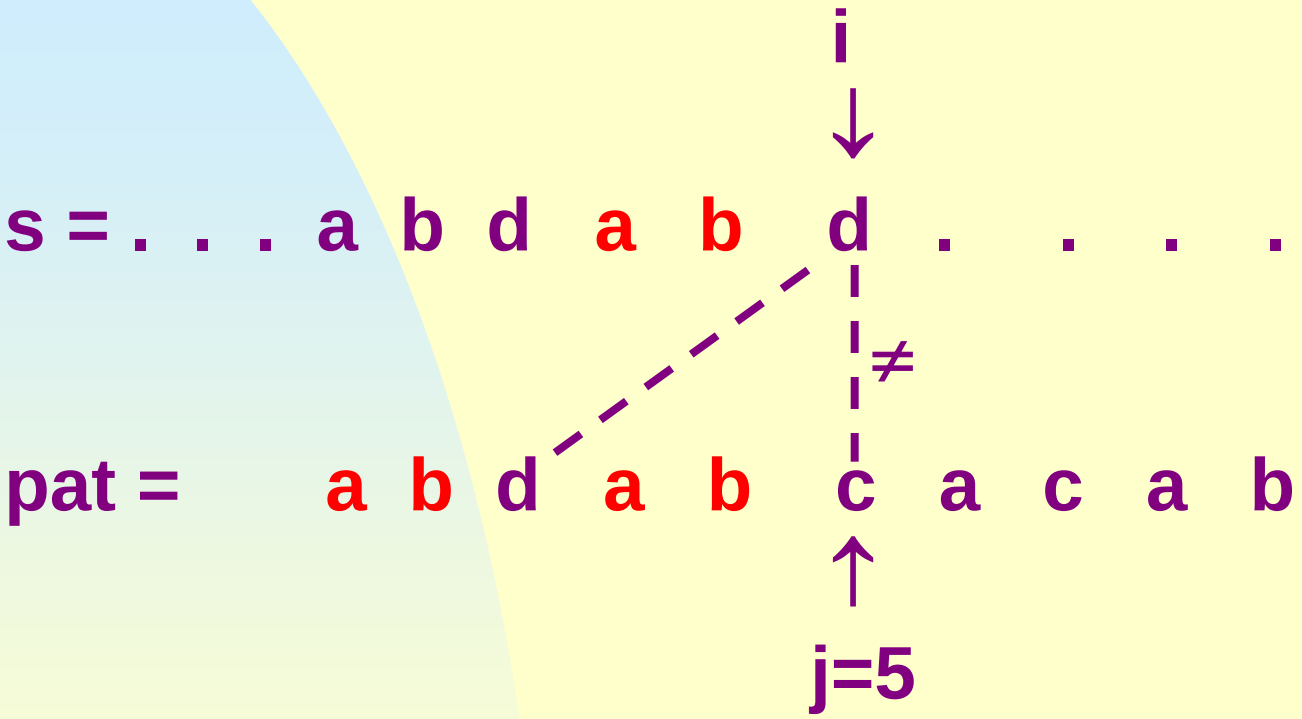


case: j = 0



case: j ≠ 0

An concrete example:



To formalize the above idea:

Definition: if $p=p_0p_1\ldots p_{n-1}$ is a pattern, then its failure function f, is defined as:

$$f(j) = \begin{cases} \text{largest } k < j, \text{ such that } p_0p_1\ldots p_k = p_{j-k}p_{j-k+1}\ldots p_j \\ \qquad\qquad \text{if such } k \geq 0 \text{ exists} \\ -1 \qquad\qquad \text{otherwise} \end{cases}$$

For example, pat = a b c a b c a c a b, we have

j	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

Note:

- largest : no match be missed
- k < j : avoid dead loop

From the definition of f, we have the following rule for pattern matching:

If a partial match is found such that $s_{i-j} \ldots s_{i-1} = p_0 p_1 \ldots p_{j-1}$ and $s_i \neq p_j$ then matching may be resumed by comparing s_i and $p_{f(j-1)+1}$ if $j \neq 0$. If $j=0$, then we may continue by comparing s_{i+1} and p_0.

The failure function is represented by an array of integers **f**, which is a private data member of String.

Now the algorithm **FastFind**.

```
1 int String::FastFind (String pat)
2 { // Determine if pat is a substring of s
3    int PosP = 0,  PosS = 0;
4    int LengthP= pat.Length( ), LengthS= Length( );
5    while ((PosP  < LengthP) && (PosS  < LengthS))
6        if ( pat.str[PosP] == str[PosS] ) { // characters match
7        PosP ++; PosS ++;
8        }
9        else
10         if ( PosP==0)
11            PosS++;
12         else PosP= pat.f [PosP-1] + 1;
13  if ((PosP < LengthP) || LengthP==0)) return –1;
14  else return PosS  - LengthP ;
15 }
```

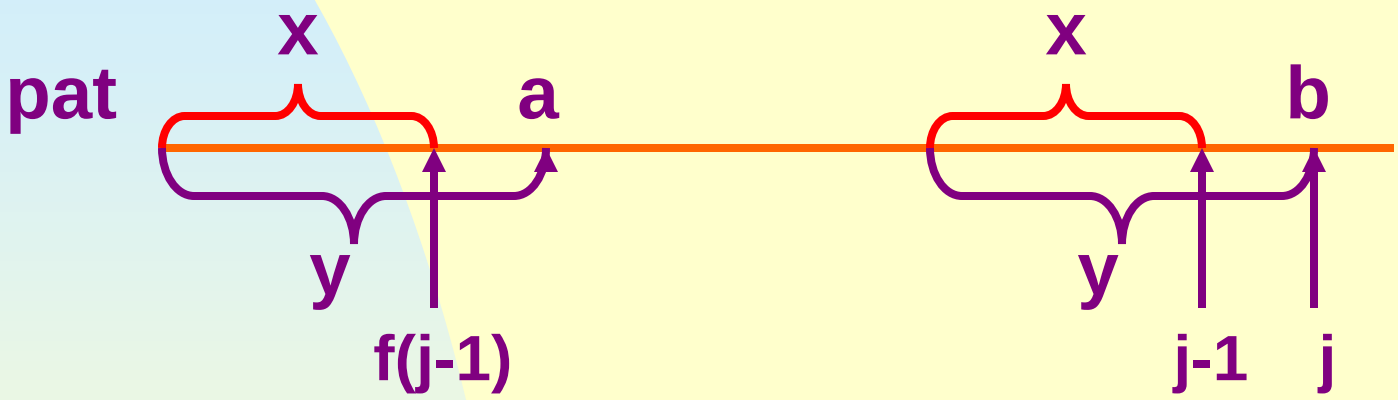
Analysis of FastFind:

- Line 7 and 11 --- at most LengthS times, since PosS is increased but never decreased. So PosP can move right on pat at most LengthS times (line 7).
- Line 12 moves PosP left, it can be done at most LengthS times. Note that f(j-1)+1< j.

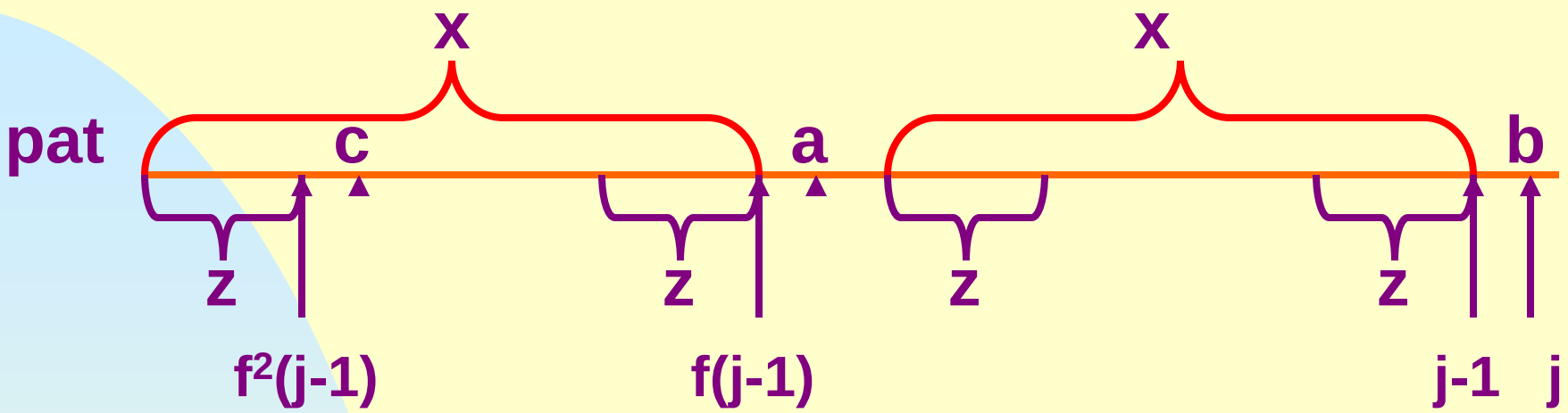
Consequently, the computing time is O(LengthS).

How about the computing of the f for the pattern? By similar idea, we can do it in O(LengthP).

f(0)=-1, now if we have f(j-1), we can compute f(j) from it by the following observation:



If a=b, then f(j)=f(j-1)+1 else



If c=b, $f(j)=f(f(j-1))+1=f^2(j-1)+1$ else ……

In general, we have the following restatement of the failure function:

$$f(j) = \begin{cases} -1 & \text{if } j=0 \\ f^m(j-1)+1 & \text{where m is the least k for which } p_{f^k(j-1)+1} = p_j \\ -1 & \text{if there is no k satisfying the above} \end{cases}$$

Now we get the algorithm to compute f.

```
1 void String::Failurefunction( )
2 { // compute the failure function of the pattern *this.
3    int LengthP= Length( );
4    f [0]= -1;
5    for (int j=1; j< LengthP; j++) // compute  f[j]
6       {
7        int i=f [j-1];
8        while ((*(str+j)!=*(str+i+1)) && (i>=0)) i=f[i]; // try for m
9         if ( *(str+j)==*(str+i+1))
10            f[j]=i+1;
11       else f[j]= -1;
12    }
13 }
```

Analysis of fail:

- In each iteration of the while i decreases (line 8, and f(j)<j)
- i is reset (line 7) to –1 (when the previous iteration went through line 11), or to a value 1 greater than its value on the previous iteration (when through line 10).
- There are only LengthP –1 executions of line 7, the value of i has a total increment of at most LengthP –1.
- i cannot be decremented more than LengthP –1 times, the while is iterated at most LengthP –1 times over the whole algorithm.

Consequently, the computing time is O(LengthP).

Now we can see, when the failure function is not known in advance, pattern matching can be carried out in O(LengthP + LengthS) by first computing the failure function and then using the FastFind.

Exercises: P118-1, P119-7, 9

Experiment 1: P123-8

Chapter 3
Stacks and Queues

3.1 Templates in C++

The mechanism of templates provided by C++ makes classes and functions more reusable.

3.1.1 Templates Functions

A template (or parameterized type) may be viewed as a variable that can be instantiated to any data type.

```
1 template <class T>
2 void  SelectionSort (T *a, int n)
3 //sort a[0] to a[n-1] into nondecreasing order
4 {    for (int  i=0;i<n;i++)
5       {
6          int  j=i;
7          // find the smallest KeyType in a[i] to a[n-1]
8          for (int  k=i +1;k<n;k++)
9              if  (a[k]<a[j]) j=k;
10          swap(a[i], a[j]);
11       }
12 }
```

Function SelectionSort can be used quite easily to sort an array of integers or floats as shown in the following:

```
float farray[100];
int   intarray[250];
…
//assume that the arrays are initialized at this point
sort(farray,100);
sort(intarray,250);
```

Function SelectionSort is instantiated to the type of the array argument supplied to it. e.g., SelectionSort(farray,100) is to sort an array of floats because farray is an array of floats.

For user defined data type T, the operator < should be overloaded in a manner consistent with its usage. e.g., sort is based on the assumption that < is transitive.

Suppose we want to sort an array of Rectangles in non-decreasing order of their areas, operator< should be overloaded with the semantic of area comparing.

The template function ChangeSize1D changes the size of a 1-Dimensional array of type T from oldSize to newSize:

```
template <class T>
void ChangeSize1D(T* a, const int oldSize, const int newSize)
{
   if (newSize < 0) throw “New length must be >= 0”;
   T* temp = new T[newSize];
   int number = min(oldSize, newSize);
   copy(a, a + number, temp);
   delete [] a;
   a = temp;
}
```

3.1.2 Using Templates to Represent Container Classes

A container class---a data structure for containing or storing a number of data objects.

Bag---a data structure into which objects can be inserted and from which objects can be deleted.

A Bag can have multiple occurrences of the same element, but we don't care the position of an element nor do we care which element to delete.

```
template <class T>
class Bag
 {
public:
    Bag (int BagCapacity = 10); // constructor
    ~Bag () ; // destructor

    int Size() const;
    bool isEmpty() const;
    T& Element() const;
    void Push(const T&);
    void Pop();

private:
    T *array;
    int capacity;
    int top;
};
```

```
template <class T>
Bag<T>::Bag(int bagCapacity):capacity(bagCapacity)  {
    if (capacity<1) throw “Capacity must be > 0”;
    array = new T[capacity];
    top = -1;
}

template <class T>
Bag<T>::~Bag()  {delete [ ] array;}

template <class T>
Inline T& Bag<T>::Element() const {
    if (IsEmpty() throw “Bag is empty”;
    return array[0];
}
```

```
template <class T>
void Bag<T>::Push(const T& x)  {
    if (capacity==top+1)
    {
        ChangSize1D(array, capacity, 2*capacity);
        capacity*=2;
     }
     array[++top] = x;
}
```

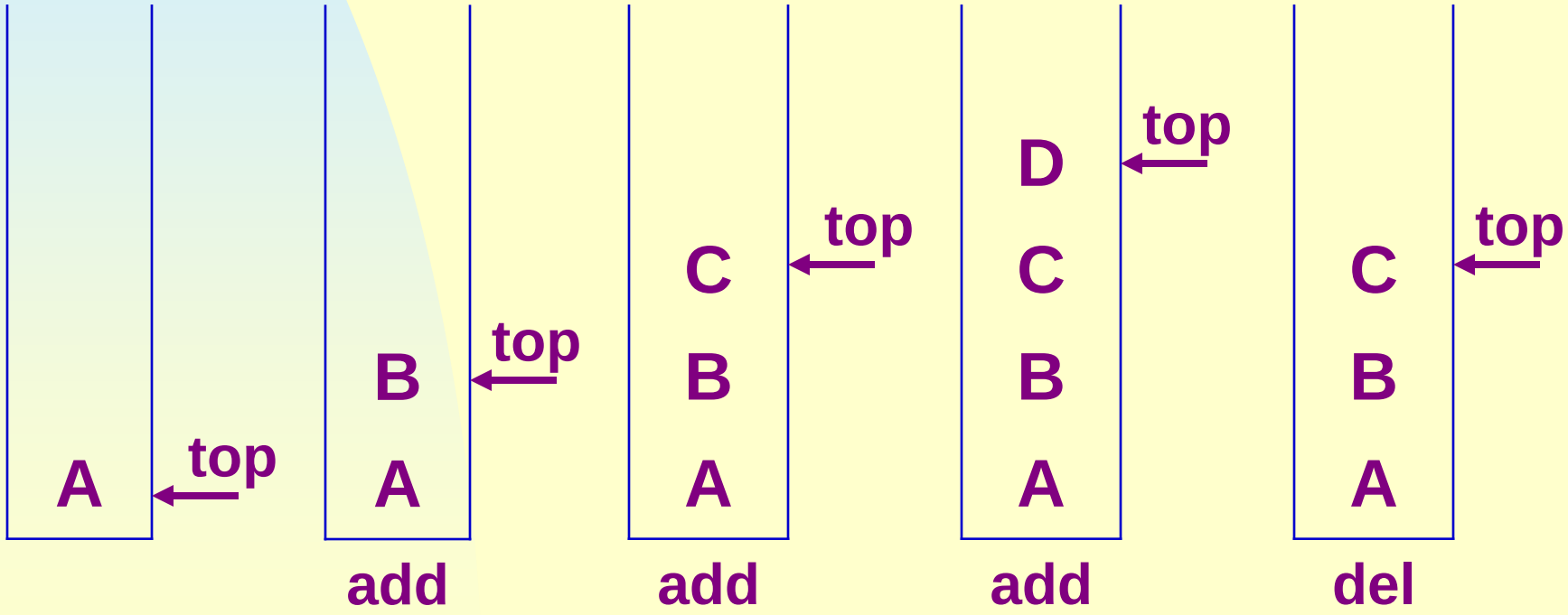
```
template <class T>
void Bag<T>::Pop()  {
    if (isEmpty()) throw “Bag is empty, cannot delete”;
    int  deletePos = top/2; // note we don’t care which to delete
    copy(array+deletePos+1, array+top+1, array+deletePos);
    array[top--].~T(); // destructor for T
}

Bag<int> a;
Bag<Rectangle> r;
```

So a is a Bag of integers and r of Rectangles.

3.2 The Stack Abstract Data type

A Stack---an ordered list in which all insertions and deletions are made at one end called top (LIFO).

Inserting and deleting elements in a stack:



ADT 3.1 Stack

```
template <class T>
class Stack
 { // A finite ordered list with zero or more elements.
 public:
     Stack (int stackCapacity = 10);
    //Creates an empty stack with initial capacity of stackCapacity

     bool IsEmpty() const;
     //If number of elements in the stack is 0, true else false

     T& Top() const;
     // Return the top element of stack

     void Push(const T& item);
     // Insert item into the top of the stack
```

```
    void Pop();
    // Delete the top element of the stack.
};
```

To implement this ADT, we can use an array and a variable top. Initially top is set to –1.

So we have the following data members of Stack:

```
private:
    T* stack;
    int top;
    int capacity;
```

```
template <class T>
Stack<T>::Stack(int stackCapacity): capacity(stackCapacity)
{
    if (capacity < 1) throw “Stack capacity must be > 0”;
    stack = new T[capacity];
    top = -1;
}

template <class T>
Inline bool Stack<T>::IsEmpty() const { return(top == -1);}
```

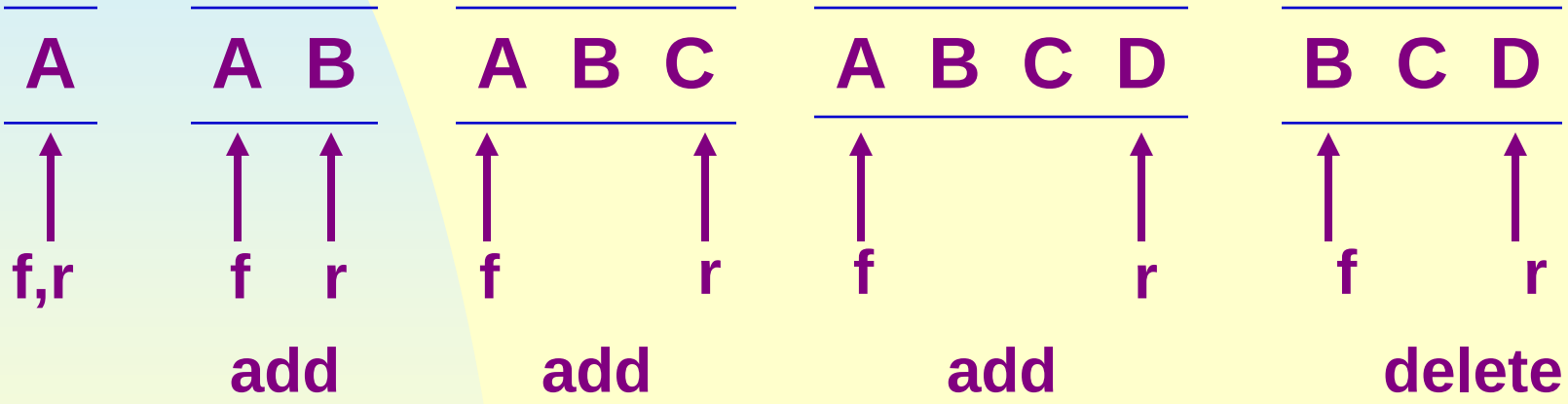
```
template <class T>
inline T& Stack<T>::Top() const
{
    if (IsEmpty()) throw “Stack is Empty”;
    return stack[top];
}

template <class T>
void Stack<T>::Push(const T& x)
{
    if (top == capacity - 1)
    {
       ChangeSize1D(stack, capacity, 2*capacity);
       capacity *= 2;
     }
     stack[++top] = x;
}
```

```
template <class T>
void Stack<T>::Pop()
{ // Delete top element of stack.
    if (IsEmpty()) throw “Stack is empty, cannot delete.”;
    stack[top--].~T();  // destructor for T
}
```

Exercises: P138-1, 2

3.3 The Queue Abstract Data Type

A Queue--- an ordered list in which all insertions take place at one end and all deletions take place at the opposite end (FIFO).



f = queue front r = queue rear

ADT 3.2 Queue

```
template <class T>
class Queue
 { // A finite ordered list with zero or more elements.
 public:
    Queue (int queueCapacity = 10);
    // Creates an empty queue with initial capacity of
    // queueCapacity

    bool IsEmpty() const;

    T& Front() const; //Return the front element of the queue.

    T& Rear() const; //Return the rear element of the queue.

    void Push(const T& item);
     //Insert item at the rear of the queue.
```

```
    void Pop();
    // Delete the front element of the queue.
};
```

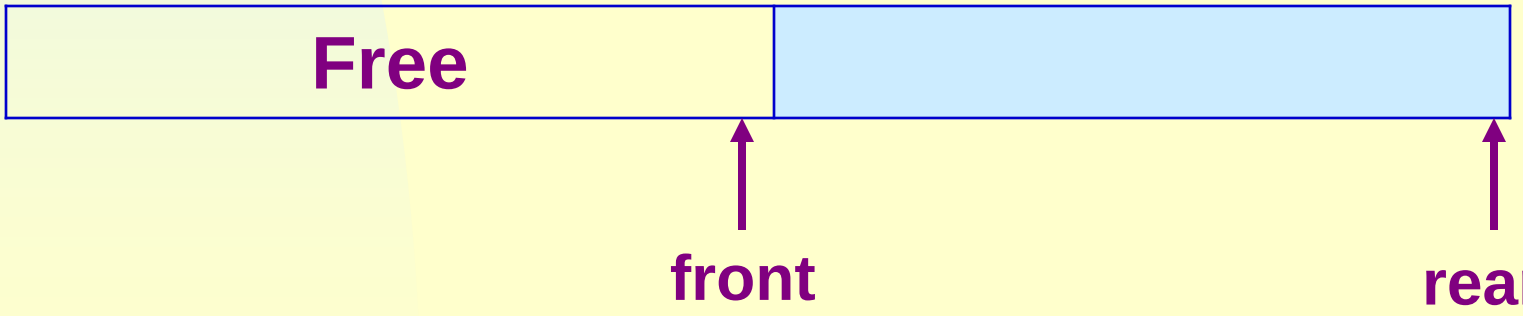
To implement this ADT, we can use an array and two variable front and rear with front being one less than the position of the first element. So we have the following data members of Queue:

```
private:
    T* queue;
    int front,
        rear,
        capacity;
```

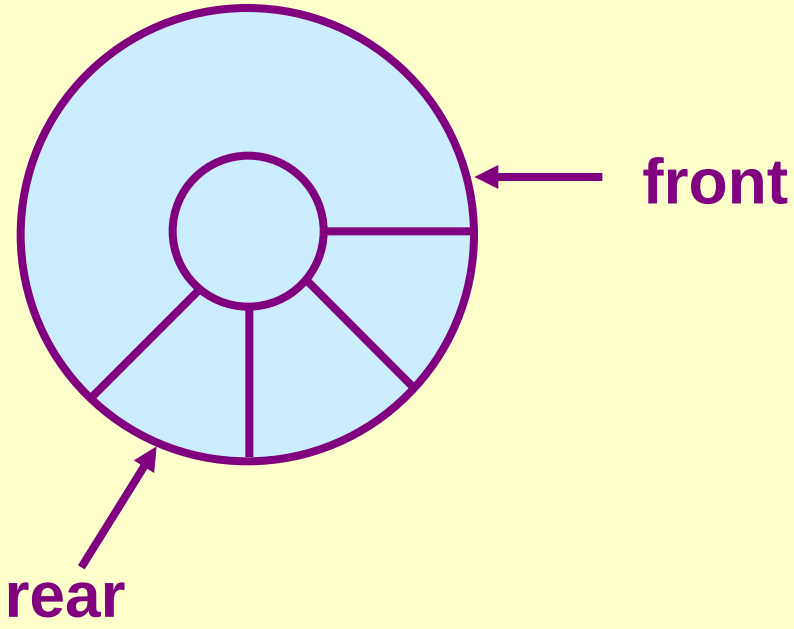
At the beginning, we can set front = rear = -1, and front == rear means that the queue is empty.

As we push and pop the queue, front may be > 0, and rear may be = capacity – 1.

In that case, we cannot add an element to the queue without shifting all elements to the left, as shown below:



Hence, circular representation. To distinguish between queue full and queue empty, we shall increase the capacity of a queue just before it becomes full.

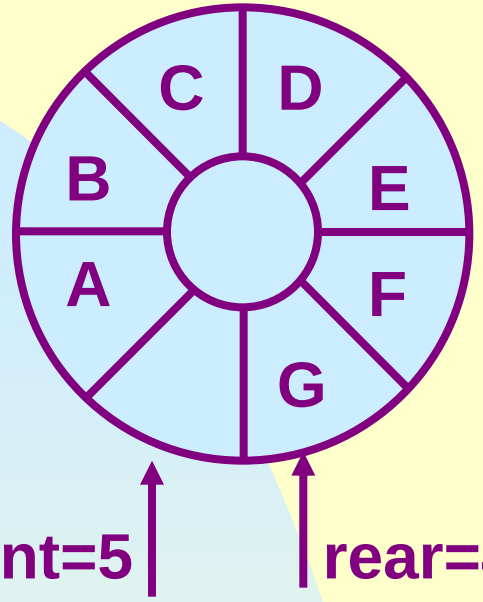


```
template <class T>
Queue<Type>::Queue(int queueCapacity):
                                    capacity(queueCapacity)
{
   if (capacity < 1) throw “Queue capacity must > 0”;
   queue = new T[capacity];
   front = rear = 0;
}
```

```
template <class T>
Inline bool Queue<T>::IsEmpty() const {return front==rear };

template <class T>
inline T& Queue<T>::Front() const
{
   if (IsEmpty()) throw “Queue is empty. No front element”;
   return queue[(front+1)%capacity];
}

template <class T>
inline T& Queue<T>::Rear() const
{
   if (IsEmpty()) throw “Queue is empty. No rear element”;
   return queue[rear];
}
```

```
template <class T>
void Queue<T>::Push(const T& x)
{ // add x at rear of queue
   if ((rear+1)%capacity == front)
   { // queue full, double capacity
     // code to double queue capacity comes here
   }
   rear = (rear+1)%capacity;
   queue[rear] = x;
}
```

We can double the capacity of queue in the way as shown in the next slide:



queue

0	1	2	3	4	5	6	7
C	D	E	F	G		A	B

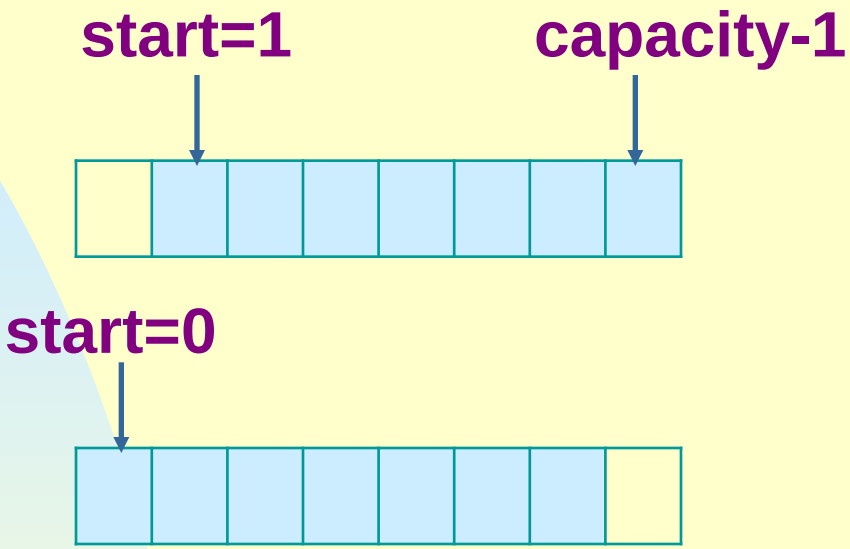
front=5, rear=4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G									

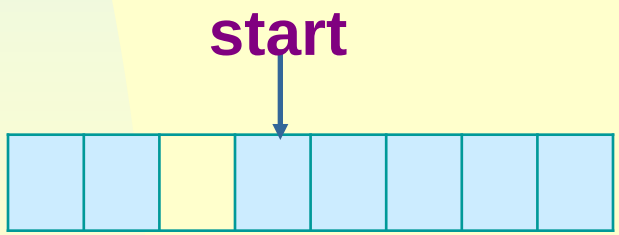
front=15, rear=6

Let start = (front+1)%capacity

Case 1, start < 2:



Case2, start >= 2:



Thus, the previous configuration may be obtained as follows:

(1) Create a new array newQueue of twice the capacity.

(2) Copy the second segment (if any) to positions in newQueue beginning at 0.

(3) Copy the first segment (if any) to positions in newQueue beginning at (capacity-start)%capacity.

The code is in the next slide:

```
// allocate an array with twice the capacity
T*  newQueue = new T[2*capacity];

// copy from queue to newQueue
int start = (front+1)%capacity;
if (start < 2)
    // no wrap around
    copy(queue+start, queue+start+capacity-1, newQueue);
else
{ // queue wraps around
    copy(queue+start, queue+capacity, newQueue);
    copy(queue, queue+rear+1, newQueue+capacity-start);
}

// switch to newQueue
front = 2*capacity-1;  rear =  capacity-2; capacity *= 2;
delete [] queue;
queue = newQueue;
```

```
template <class T>
void Queue<T>::Pop()
{  // Delete front elemnet from queue
   if (IsEmpty()) throw “Queue is empty. Cannot delete”;
   front = (front+1)%capacity;
   queue[front].~T;
}
```

For the circular representation, the worst-case add and delete times (assuming no array resizing is needed) are O(1).

Exercises: P147-1, 3.

3.5 A Mazing Problem

Problem: find a path from the entrance to the exit of a maze.

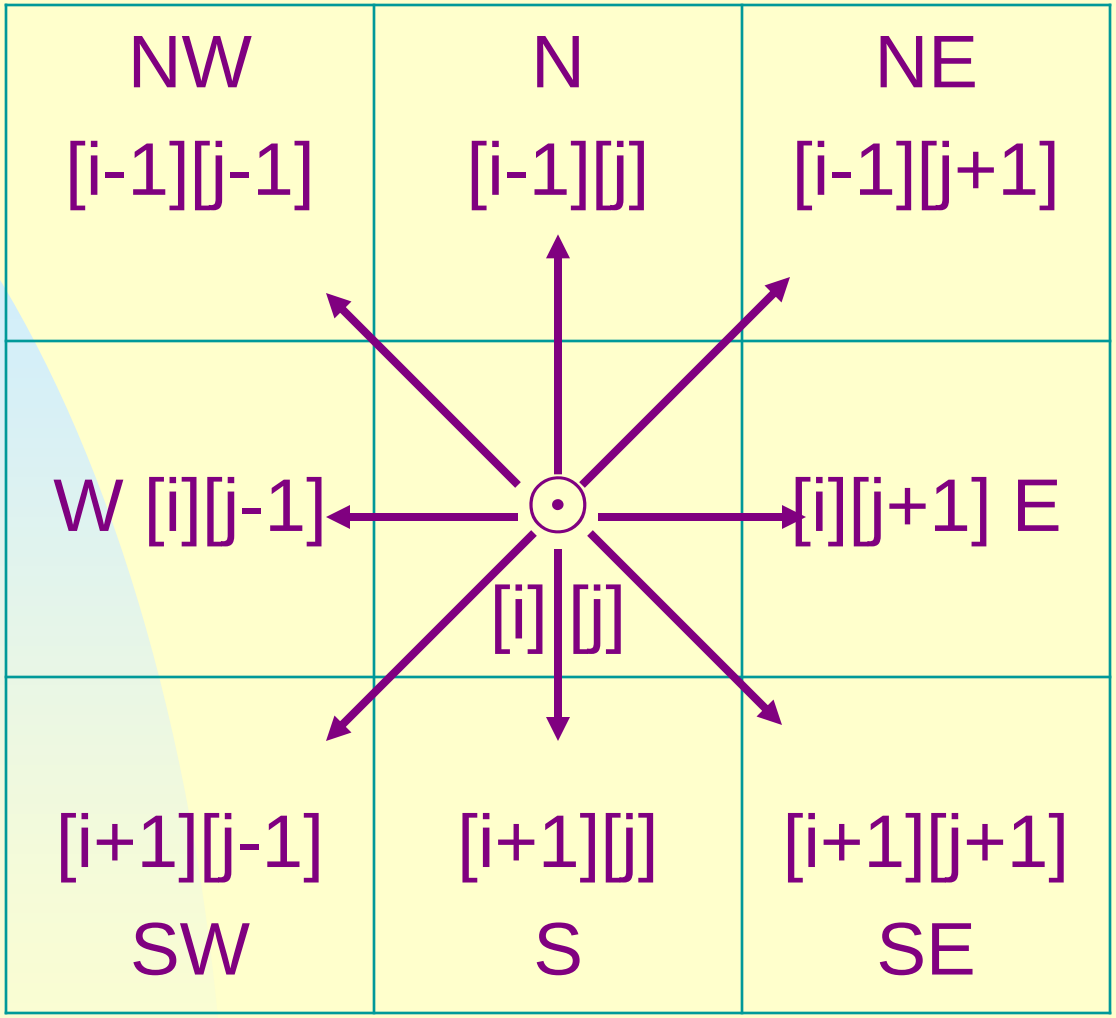
entrance

0	1	0	0	1	1	0	1	1
1	**0**	**0**	1	0	0	1	1	1
0	1	1	**0**	1	1	1	0	1
1	1	0	**0**	1	**0**	**0**	1	0
1	0	0	1	**0**	1	1	**0**	1
0	0	1	1	0	1	**0**	1	1
0	1	0	0	1	1	**0**	**0**	**0**

exit

Representation:

- maze[i][j], $1 \le i \le m$, $1 \le j \le p$.
- 1--- blocked, 0 --- open.
- the entrance: maze[1][1], the exit: maze[m][p].
- current point: [i][j].
- boarder of 1's, so a maze[m+2][p+2].
- 8 possible moves: N, NE, E, SE, S, SW, W and NW.


NW
[i-1][j-1]
N
[i-1][j]
NE
[i-1][j+1]
W [i][j-1]
[i][j]
[i][j+1] E
[i+1][j-1]
SW
[i+1][j]
S
[i+1][j+1]
SE

To predefine the 8 moves:

```
struct  offsets
{
    int a,b;
};
enum directions {N, NE, E, SE, S, SW, W, NW};
offsets move[8];
```

q	move[q].a	move[q].b
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

Table of moves

Thus, from [i][j] to [g][h] in SW direction:

g=i+move[SW].a; h=j+move[SW].b;

The basic idea:

Given current position [i][j] and 8 directions to go, we pick one direction d, get the new position [g][h].

If [g][h] is the goal, success.

If [g][h] is a legal position, save [i][j] and d+1 in a stack in case we take a false path and need to try another direction, and [g][h] becomes the new current position.

Repeat until either success or every possibility is tried.

In order to prevent us from going down the same path twice, use another array mark[m+2][p+2], which is initially 0.

Mark[i][j] is set to 1 once the position is visited.

First pass:

Initialize stack to the maze entrance coordinates and direction east;

while (stack is not empty)

{

(i, j, dir)=coordinates and direction from top of stack;

pop the stack;

while (there are more moves from (i, j))

{

(g, h)= coordinates of next move ;

if ((g==m) **&&** (h==p)) success;

```
        if ((!maze[g][h]) && (!mark[g][h])) // legal and not visited
        {
            mark[g][h]=1;
            dir=next direction to try;
            push (i, j, dir) to stack;
            (i, j, dir) = (g, h, N);
         }
    }
}
cout << “No path in maze.”<< endl;
```

We need a stack of items:

```
struct Items {
        int x, y, dir;
 };
```

Also, to avoid doubling array capacity during stack pushing, we can set the size of stack to m*p.

Now a precise maze algorithm.

```
void path(const int m, const int p)
{ //Output a path (if any) in the maze; maze[0][i] = maze[m+1][i]
  // = maze[j][0] = maze[j][p+1] = 1, 0 ≤ i ≤ p+1, 0 ≤ j ≤ m+1.
     // start at (1,1)
     mark[1][1]=1;
     Stack<Items> stack(m*p);
     Items temp(1, 1, E);
     stack.Push(temp);
     while (!stack.IsEmpty())
     {
           temp= stack.Top();
          Stack.Pop();
          int i=temp.x; int j=temp.y; int d=temp.dir;
```

```
while (d<8)
{
    int g=i+move[d].a; int h=j+move[d].b;
    if ((g==m) && (h==p)) { // reached exit
       // output path
       cout <<stack;
       cout << i<<" "<< j<<" "<<d<< endl; // last two
       cout << m<<" "<< p<< endl;          // points
       return;
     }
```

```
            if ((!maze[g][h]) && (!mark[g][h])) { //new position
                mark[g][h]=1;
                temp.x=i; temp.y=j; temp.dir=d+1;
                stack.Push(temp);
                i=g ; j=h ; d=N;   // move to (g, h)
             }
             else d++;  // try next direction
        }
    }
 cout << "No path in maze."<< endl;
}
```

The operator << is overloaded for both Stack and Items as:

```
template <class T>
ostream& operator<<(ostream& os, Stack<T>& s)
{
    os << "top="<<s.top<< endl;
    for (int i=0;i<=s.top;i++);
        os<<i<<":"<<s.stack[i]<< endl;
    return os;
}
```

We assume << can access the private data member of Stack through the friend declaration.

```
ostream& operator<<(ostream& os,Items& item)
{
    return os<<item.x<<“,”<<item.y<<“,”<<item.dir-1;
    // note item.dir is the next direction to go so the current
    // direction is item.dir-1.
}
```

Since no position is visited twice, the worst case computing time is O(m*p).

Exercises: P157-2, 3

3.6 Evaluation of Expressions

3.6.1 Expressions

A expression is made of operands, operators, and delimiters. For instance,

X = A / B – C + D * E – A * C

- **the order in which the operations are carried out must be uniquely defined.**
- **to fix the order, each operator is assigned a priority.**

- within any pair of parentheses, operators with highest priority will be evaluated first.
- evaluation of operators of the same priority will proceed left to right.
- Innermost parenthesized expression will be evaluated first.

The next slide shows a set of sample priorities from C++.

priority	operator
1	unary minus, !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	==, !=
6	&&
7	||

Figure 3.15

Problem: how to evaluate an expression?

Solution:

1.Translate from infix to post fix;

2.Evaluate the postfix.

3.6.2 Postfix Notation

Infix: operators come in-between operands (unary operators precede their operand).

Postfix: each operator appears after its operands.

e.g.

infix: **A / B – C + D * E – A * C**

postfix: **A B / C – D E * + A C * –**

Every time we compute a value, we store it in the temporary location T_i, $i \geq 1$. Read the postfix left to right to evaluate it:

A B / C – D E * + A C * –

operation	postfix
T_1=A/B	T_1 C – D E * + A C * –
T_2=T_1 - C	T_2 D E * + A C * –
T_3=D*E	T_2 T_3 + A C * –
T_4=T_2+ T_3	T_4 A C * –
T_5=A*C	T_4 T_5–
T_6= T_4–T_5	T_6

T_6 is the result.

Virtues of postfix:

- **no need for parentheses**
- **the priority of the operators is no longer relevant**

These make the evaluation of postfix easy: make a left to right scan, stack operands, and evaluate operators using the correct number of operands from the stack and place the result onto the stack.

```
void  Eval(Expression e)
{ // evaluate the postfix expression e. It is assumed that the
  // last token in e is '#'. A function NextToken is used to get
  // the next token from e. Use stack.
     Stack<Token> stack;  //initialize stack
     for  (Token x = NextToken(e); x!='#'; x=NextToken(e))
        if (x is an operand) stack.Push(x);
        else {  // operator
            remove the correct number of operands for operator x
            from stack; perform the operation x and store the result
            (if any) onto the stack;
         }
}
```

3.6.3 Infix to Postfix

Idea: note the order of the operands in both infix and postfix is the same, we can form the postfix by immediately passing any operands to the output, then store the operators in a stack until just the right time.

e.g.

A*(B+C)*D → ABC+*D*

The algorithm behaves as:

Next token	stack	output
A	empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
*	*	ABC+ *
D	*	ABC+ *D
done	empty	ABC+ *D*

Attention

From the example, we can see the left parenthesis behaves as an operator with high priority when its not in the stack, whereas once it get in, it behaves as one with low priority.

- isp (in-stack priority)
- icp (in-coming priority)
- the isp and icp of all operators in Fig. 3.15 remain unchanged
- isp(‘(‘)=8, icp(‘(‘)=0, isp(‘#’)=8

Hence the rule:

Operators are taken out of stack as long as their isp is numerically less than or equal to the icp of the new operator.

```
void Postfix (Expression e)
{ // output the postfix of the infix expression e. It is assumed
  // that the last token in e is '#'. Also, '#' is used at the bottom
  // of the stack.
    Stack<Token> stack;  //initialize stack
    stack.Push('#');
```

```
  for  (Token x=NextToken(e); x!='#'; x=NextToken(e))
    if (x is an operand) cout<<x;
    else if (x==')')
        { // unstack until '('
          for  (; stackTop()!='('; stack.Pop())
              cout<<stack.Top();
          stack.Pop();  // unstack '('
          }
    else { // x is an operator
         for (; isp(stack.Top()) <= icp(x); stack.Pop())
            cout<<stack.Top();
         stack.Push(x);
     }
  // end of expression, empty the stack
  for (; !stack.IsEmpty()); cout<<stack.Top(), stack.Pop());
  cout << endl;
}
```

Analysis:

- Computing time: one pass across the input with n tokens, O(n).
- The stack will not be deeper than 1 (‘#’) + the number of operators in e.

Exercises: P165-1,2

Chapter 4
Linked Lists

4.1 Singly Linked lists Or Chains

The representation of simple data structure using an array and a sequential mapping has the property:

Successive nodes of the data object are stored at fixed distance apart.

This makes it easy to access an arbitrary node in O(1).

Disadvantage of sequential mapping:

insertion and deletion are expensive, e.g.:

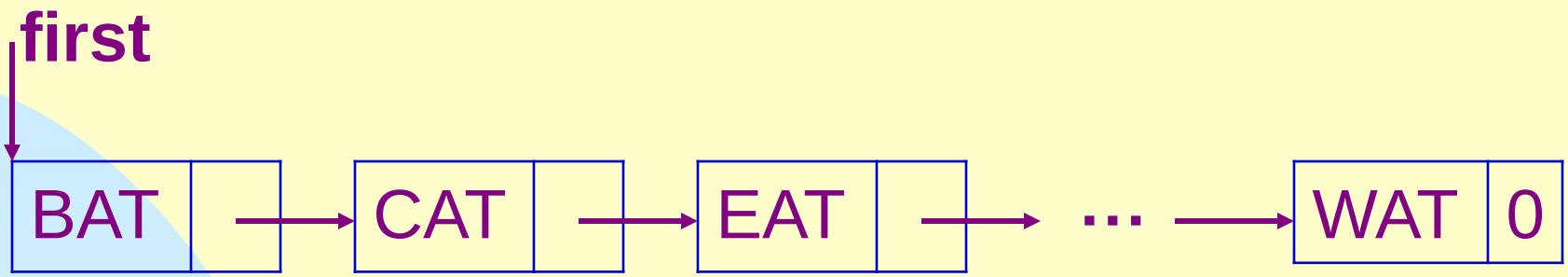
Insert "GAT" into or delete "LAT" from

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, TAT,VAT,WAT)

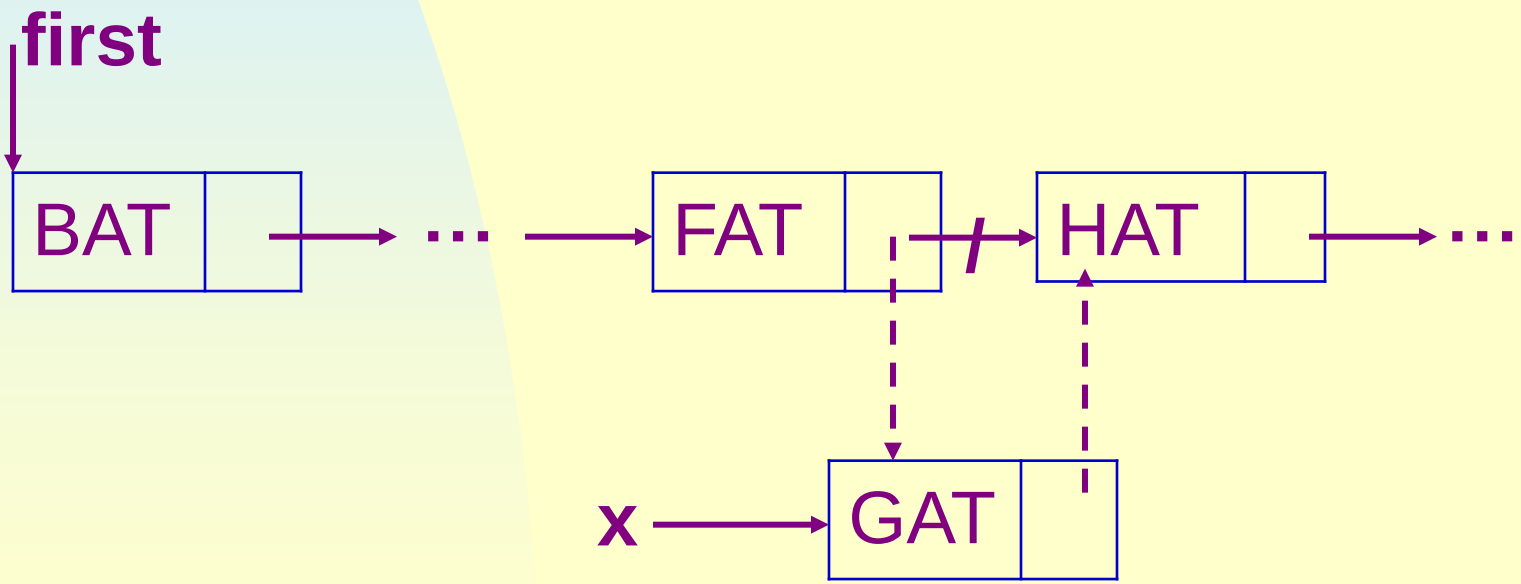
---need data movement.

Solution---linked representation:

- **items of a list may be placed anywhere in the memory.**
- **Associated with each item is a point (link) to the next item.**



In linked list, insertion (deletion) of arbitrary elements is much easier:



The above structures are called singly linked lists or chains in which each node has exactly one point field.

4.2 Representing Chains in C++

Assume a chain node is defined as:

```
class ChainNode {
private:
   int data;
   ChainNode *link;
};

ChainNode *f;
```

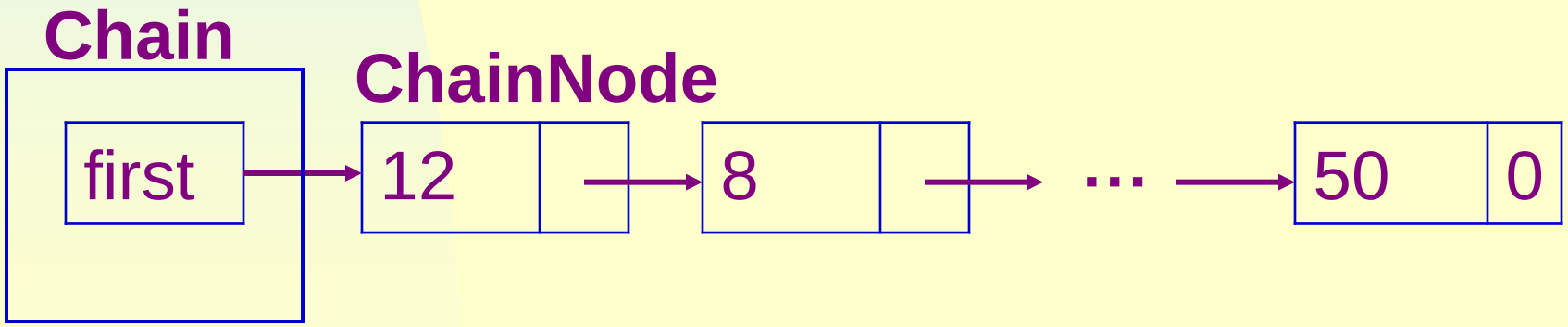
f→data

will cause a compiler error because a private data member cannot be accessed from outside of the object.

Definition: a data object of Type A HAS-A data object of Type B if A conceptually contains B or B is a part of A.

A composite of two classes: ChainNode and Chain.

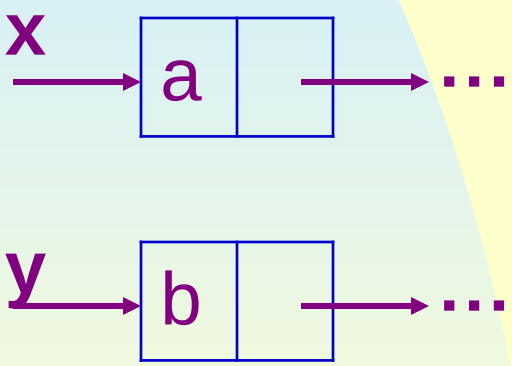
Chain HAS-A ChainNode.



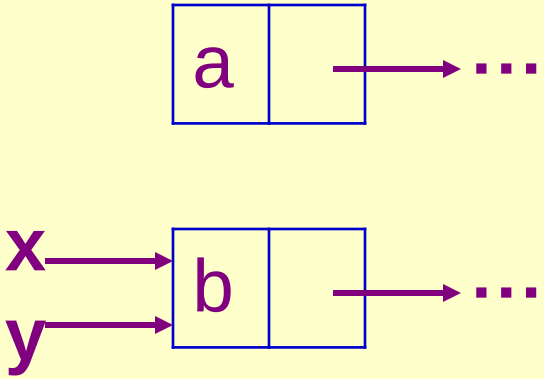
```
class Chain;  // forward declaration
class ChainNode {
friend class Chain; // to make functions of Chain be able to
                    // access private data members of ChainNode
Public:
   ChainNode(int element = 0, ChainNode* next = 0)
       {data = element; link = next;}
private:
   int data;
   ChainNode *link;
};
class Chain {
public:
   // Chain manipulation operations
…
private:
   ChainNode *first;
};
```

Null pointer constant 0 is used to indicate no node.

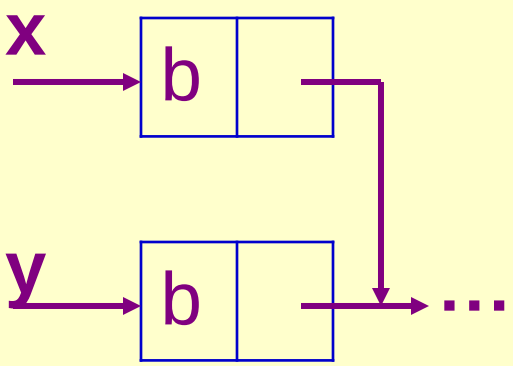
Pointer manipulation in C++:



(a)



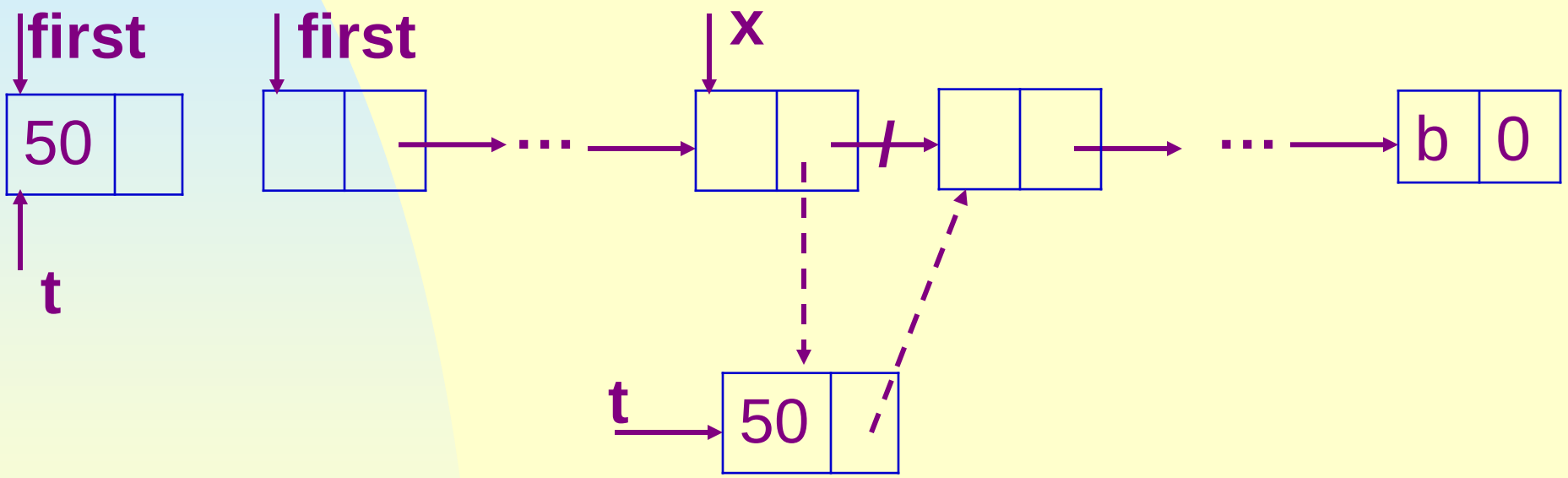
(b) x=y



(c) *x=*y

Chain manipulation:

Example 4.3 insert a node with data field 50 following the node x.



(a) first=0 **(b) First !=0**

```
void Chain::Insert50 (ChainNode *x)
{
    if (first)
        // insert after x
       x→link = new ChainNode(50, x→link);
    else
        // insert into empty chain
        first = new ChainNode(50);
}
```

Exercises: P183-1,2

4.3 The Template Class Chain

We shall enhance the chain class of the previous section to make it more reusable.

4.3.1 Implementing Chains with Templates

```
template <class T> class Chain;  // forward declaration

template <class T>
class ChainNode {
friend class Chain<T>;
public:
    ChainNode(T element, ChainNode* next = 0)
        { data = element; link = next;}
private:
    T data;
    ChainNode *link;
};
```

```
template <class T>
class Chain {
public:
    Chain() { first=0;}; // constructor initializing first to 0
    // Chain manipulation operations
…
private:
    ChainNode<T> *first;
};
```

A empty chain of integers intchain would be defined as:

```
    Chain<int> intchain ;
```

4.3.2 Chain Iterators

A container class is a class that represents a data structure that contains or stores a number of data objects.

An iterator is an object that is used to access the elements of a container class one by one.

Why we need an iterator?

Consider the following operations that might be performed on a container class C, all of whose elements are integers:

(1) Output all integers in C.

(2) Obtain the sum, maximum, minimum, mean, median of all integers in C.

(3) Obtain the integer x from C such that f(x) is maximum.

……

These operations have to be implemented as **member functions** of C to access its private data members.

Consider the container class Chain<T>, there are, however, some drawbacks to this:

(1) All operations of Chain<T> should preferably be independent of the type of object to which T is initialized. However, operations that make sense for one instantiation of T may not for another instantiation.

(2) The number of operations of Chain<T> can become too large.

(3) Even if it is acceptable to add member functions, the user would have to learn how to sequence through the container class.

These suggest that container class be equipped with iterators that provide systematic access to the elements of the object.

User can employ these iterators to implement their own functions depending upon the particular application.

Typically, an iterator is implemented as a nested class of the container class.

A forward Iterator for Chain

A forward Iterator class for Chain may be implemented as in the next slides, and it is required that ChainIterator be a public nested member class of Chain.

```
class ChainIterator {
public:
    // typedefs required by C++ omitted

    // constructor
    ChainIterator(ChainNode<T>* startNode = 0)
        { current = startNode; }

    // dereferencing operators
    T& operator *() const { return current→data;}
    T* operator →() const { return &current→data;}
```

```
// increment
ChainIterator& operator ++() // preincrement
   {current = current→link; return *this;}
ChainIterator& operator ++(int) // postincrement
   {
      ChainIterator old = *this;
      current = current→link;
      return old;
    }
```

```
    // equality testing
    bool operator !=(const ChainIterator right) const
        { return current != right.current; }
    bool operator == (const ChainIterator right) const
        { return current == right.current; }

    private:
        ChainNode<T>* current;
};
```

Additionally, we add the following public member functions to Chain:

```
ChainIterator begin() {return ChainIterator(first);}
ChainIterator end() {return ChainIterator(0);}
```

We may initialize an iterator object yi to the start of a chain of integers y using the statement:

```
Chain<int>::ChainIterator yi = y.begin();
```

And we may sum the elements in y using the statement:

```
sum = accumulate(y.begin(), y.end(), 0);
// note sum does not require access to private members
```

Exercises: P194-3, 4

4.3.3 Chain Operations

Operations provided in a reusable class should be enough but not too many.

Normally, include: constructor, destructor, operator=, operator==, operator>>, operator<<, etc.

A chain class should provide functions to insert and delete elements.

Another useful function is reverse that does an "in-place' reversal of the elements in a chain.

To be efficient, we add a private member last to Chain<T>, which points to the last node in the chain.

InsertBack

```
template <class T>
void Chain<T>::InsertBack(const T& e)
{
   if (first) { // nonempty chain
      last→link = new ChainNode<T>(e);
      last =last→link;
    }
   else first = last= new ChainNode<T>(e);
}
```

The complexity: O(1).

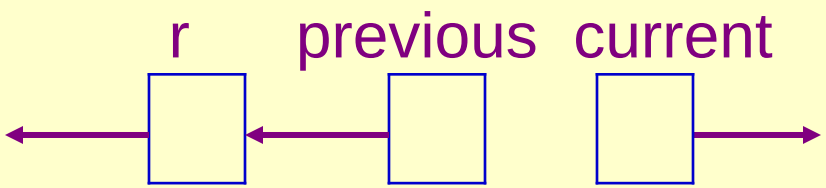
Concatenate

```
template <class T>
void Chain<T>::Concatenate(Chain<T>& b)
{ // b is concatenete to the end of *this
    if (first) { last→link = b.first; last = b.last;}
    else { first = b.first; last = b.last; );}
    b.first = b.last = 0;
}
```

The complexity: O(1).

Reverse

```
template <class T>
void Chain<T>::Reverse()
{ // make (a1,.., an) becomes (an,.., a1).
    ChainNode<T> *current = first, *previous = 0;
    while (current) {
        ChainNode<T> *r = previous; // r trails previous
        previous = current;
        current = current→link;
        previous→link = r;   //
    }
    first = previous;
}
```

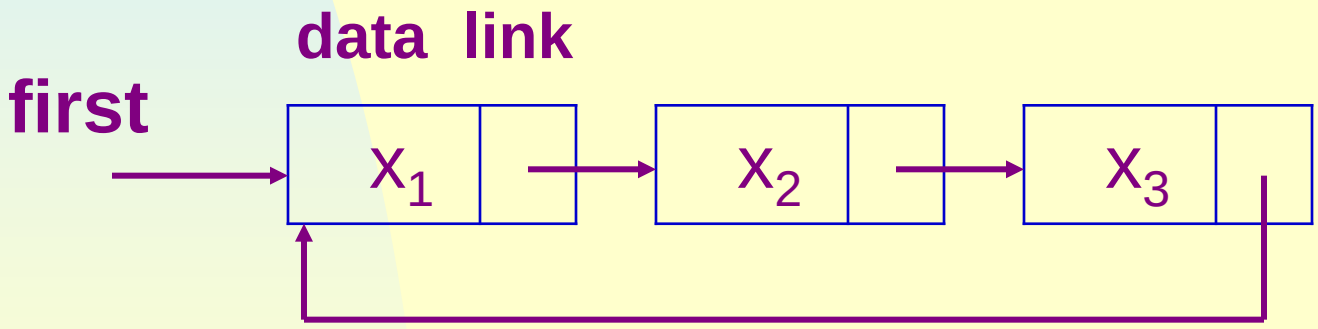


For a chain with $m \geq 1$ nodes, the computing time of Reverse is O(m).

Exercises: P184-6

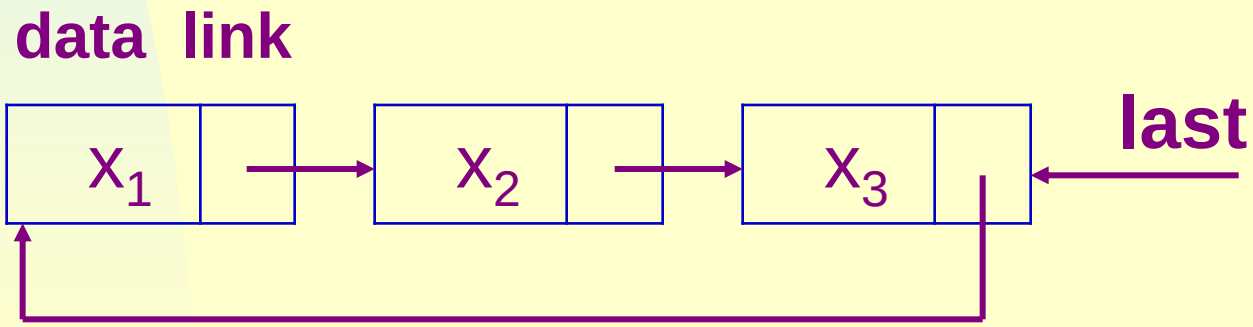
4.4 Circular Lists

A circular list can be obtained by making the last link field point to the first node of a chain.



Consider inserting a new node at the front of the above list. We need to change the link field of the node containing x_3.

It is more convenient if the access pointer points to the **last** rather than the first.



Now we can insert at the front in O(1):

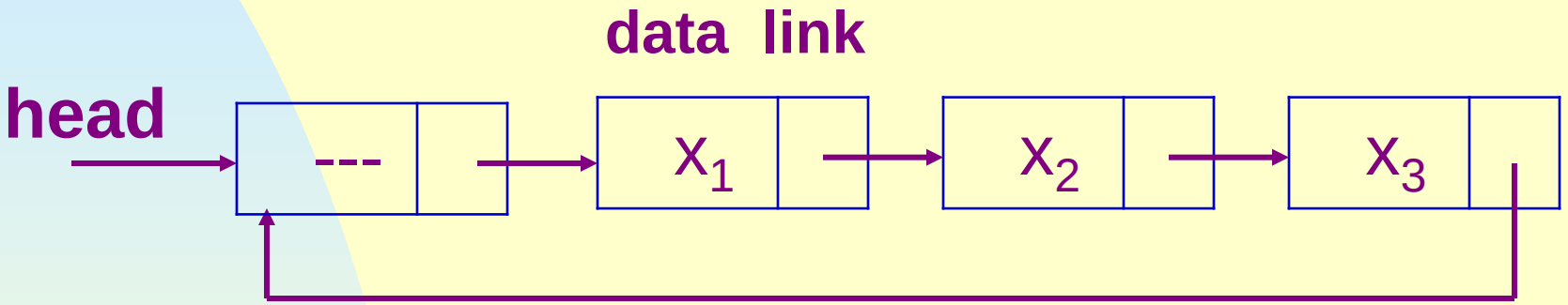
```
template <class T>
void CircularList<T>::InsertFront(const T& e)
{ // insert the element e at the "front" of the circular list *this,
  // where last points to the last node in the list.
    ChainNode<T>* newNode = new ChainNode<T>(e);
    if (last) { // nonempty list
        newNode→link = last→link;
        last→link = newNode;
     }
    else { last = newNode; newNode→link = newNode;}
}
```

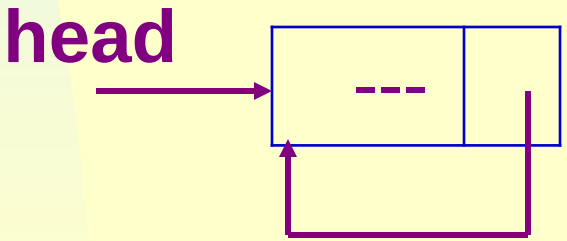
To insert at the back, we only need to add the statement

```
last = newNode;
```

to the if clause of InsertFront, the complexity is still O(1).

To avoid handling empty list as a special case, introduce a dummy head node:.





empty circular list

4.5 Available Space lists

- the time of destructors for chains and circular lists is **linear** in the length of the chain or list.
- it may be reduced to O(1) if we maintain our own chain of free nodes.
- the available space list is pointed by **av**.
- **av** be a static class member of **CircularList<T>** of type **ChainNode<T> ***, initially, **av = 0**.
- only when the **av** list is empty do we need use **new**.

We shall now use CircularList<T>::GetNode instead of using new:

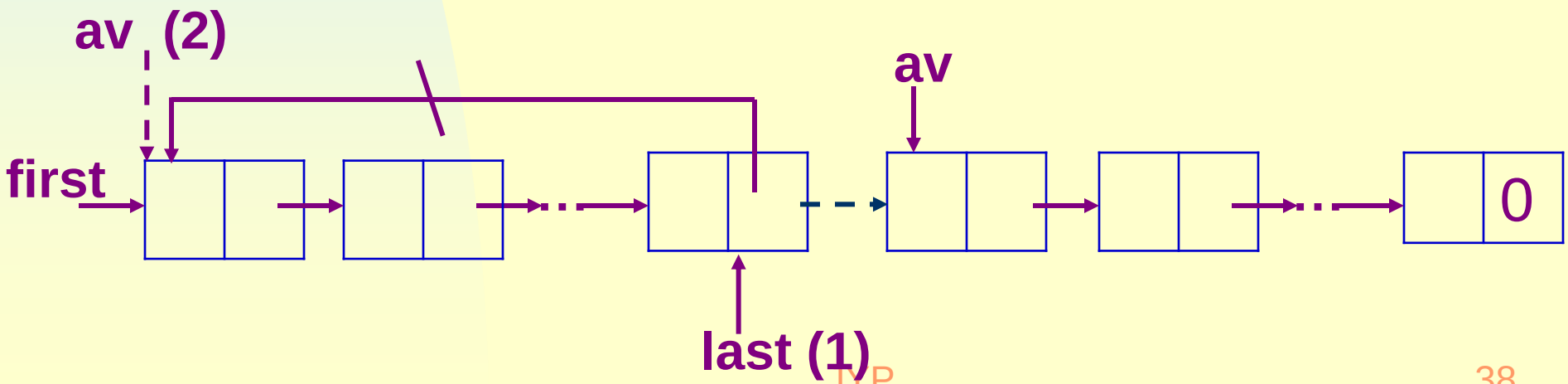
```
template <class T>
ChainNode<T>* CircularList<T>::GetNode( )
{ //provide a node for use
    ChainNode<T> * x;
    if (av) { x = av; av = av→link;}
    else x = new ChainNode<T>;
    return x;
}
```

And we use CircularList<T>::RetNode instead of using delete:

```
template <class T>
 void CircularList<T>::RetNode(ChainNode<T>* x)
{ // free the node pointed to by x
    x→link = av;
    av = x;
    x = 0;
}
```

A circular list may be deleted in O(1):

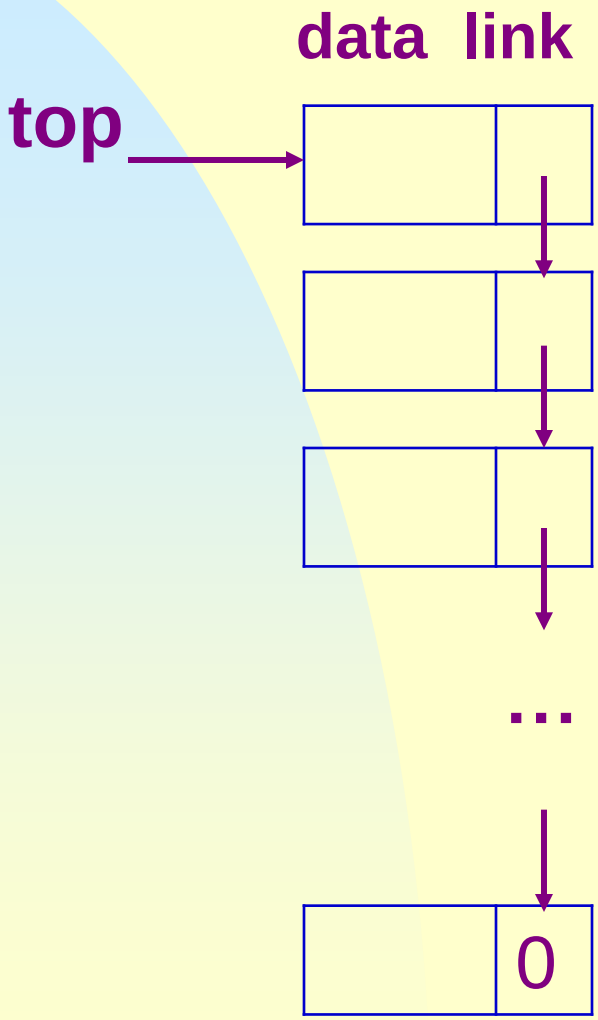
```
template <class T>
void CircularList<T>::~CircularList()
{ // delete the circular list.
   if (last) {  ChainNode <T> * first = last→link;
        last→link = av;   // (1)
        av = first; // (2)
        last = 0;
    }
}
```



A chain may be deleted in O(1) if we know its first and last nodes:

```
template <class T>
Chain<T>::~Chain()
{ // delete the chain
    if (first) {
        last→link = av;
        av = first;
        first = 0;
    }
}
```

4.6 Linked Stacks and Queues



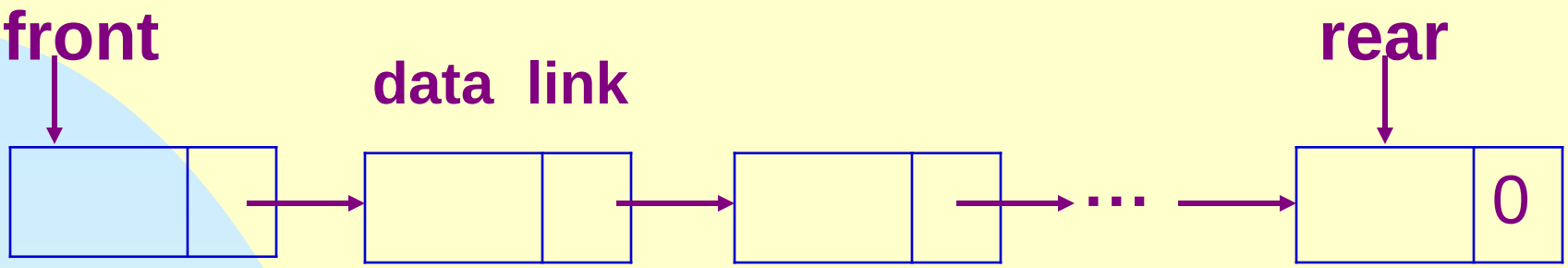
linked stack

Assume the LinkedStack class has been declared as friend of ChainNode<T>.

```
template <class T>
class LinkedStack {
public:
    LinkedStack() { top=0;}; // constructor initializing top to 0
    // LinkedStack manipulation operations
…
private:
    ChainNode<T> *top;
};
```

```
template <class T>
void LinkedStack<T>::Push(const T& e) {
    top = new ChainNode<T>(e, top);
}

template <class T>
void LinkedStack<T>::Pop()
{ // delete top node from the stack.
    if (IsEmpty()) throw "Stack is empty. Cannot delete.";
    ChainNode<T> * delNode = top;
    top = top→link;
    delete delNode;
}
```

The functions IsEmpty and Top are easy to implement, and are omitted.



linked queue

The functions of LinkedQueue are similar to those of LinkedStack, and are left as exercises.

Exercises: P201-2

4.7 Polynomials

4.7.1 Polynomial Representation

Since a polynomial is to be represented by a list, we say Polynomial is IS-IMPLEMENTED-IN-TERMS-OF List.

Definition: a data object of Type A IS-IMPLEMENTED-IN-TERMS-OF a data object of Type B if the Type B object is central to the implementation of Type A object. ---Usually by declaring the Type B object as a data member of the Type A object.

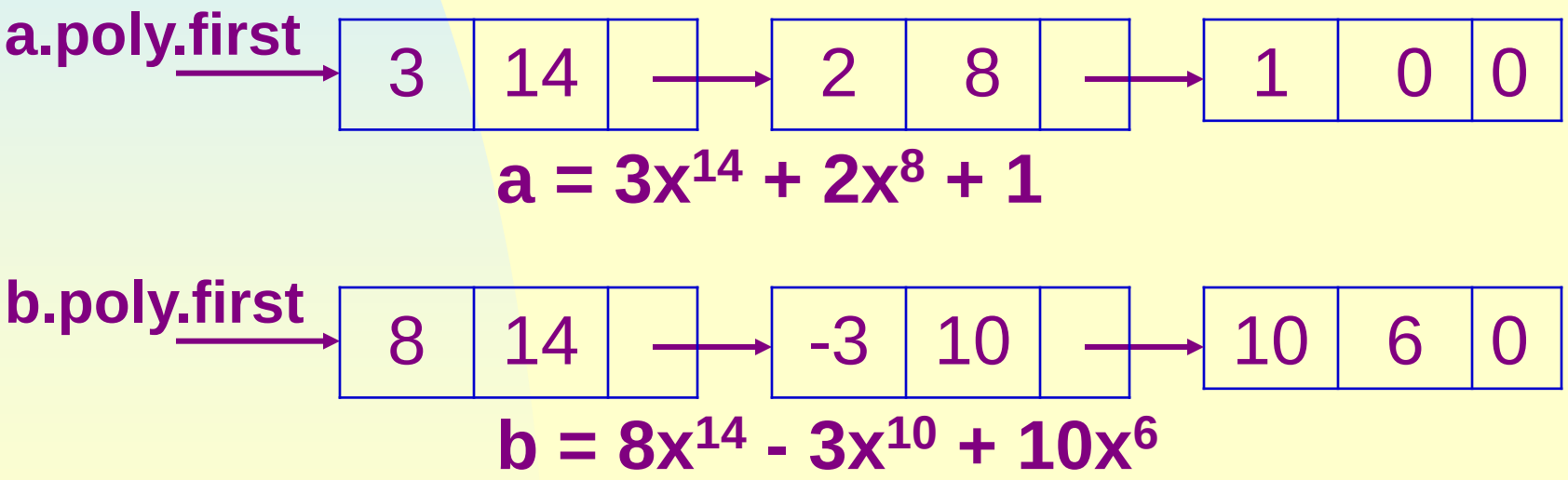
$A(x) = a_m x^{em} + a_{m-1} x^{em-1} + \ldots + a_1 x^{e1}$

Where $a_i \neq 0$, $e_m > e_{m-1} > \ldots > e_1 \geq 0$

- Make the chain poly a data member of Polynomial.
- Each ChainNode will represent a term. The template T is instantiated to struct Term:

```
struct Term
{ // all members of Term are public by default
    int coef;
    int exp;
    Term Set(int c, int e) { coef=c; exp=e; return *this;};
};
```

```
class Polynomial {
public:
    // public functions defined here
private:
    Chain<Term> poly;
};
```



4.7.2 Adding Polynomials

To add two polynomials a and b, use the chain iterators ai and bi to move along the terms of a and b.

```
1 Polynomia Polynomial::operaor+ (const Polynomial& b)
2 { // *this (a) and b are added and the sum returned
3   Term temp;
4   Chain<Term>::ChainIterator ai = poly.begin(),
5                              bi = b.poly.begin();
6   Polynomial c;
```

```
7    while (ai != poly.end() && bi != b.poly.end()) { //not null
8       if (ai→exp == bi→exp) {
9          int sum = ai→coef + bi→coef;
10         if (sum) c.poly.InsertBack(temp.Set(sum, bi→exp));
11         ai++; bi++; // to next term
12      }
13      else if (ai→exp < bi→exp) {
14            c.poly.InsertBack(temp.Set(bi→coef, bi→exp));
15            bi++;  // next term of b
16      }
17      else {
18            c.poly.InsertBack(temp.Set(ai→coef, ai→exp));
19            ai++;  // next term of a
20       }
21   }
```

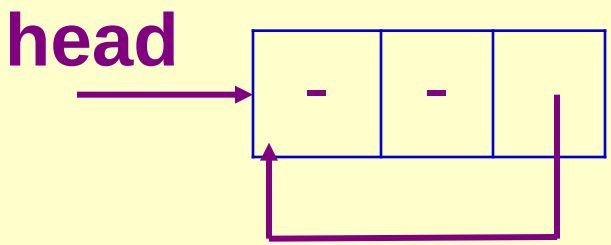
```
22    while (ai != poly.end()) { // copy rest of a
23        c.poly.InsertBack(temp.Set(ai→coef, ai→exp));
24        ai++;
25   }
26    while (bi != b.poly.end()) { // copy rest of b
27        c.poly.InsertBack(temp.Set(bi→coef, bi→exp));
28       bi++;
29    }
30    return c;
31 }
```

Analysis:

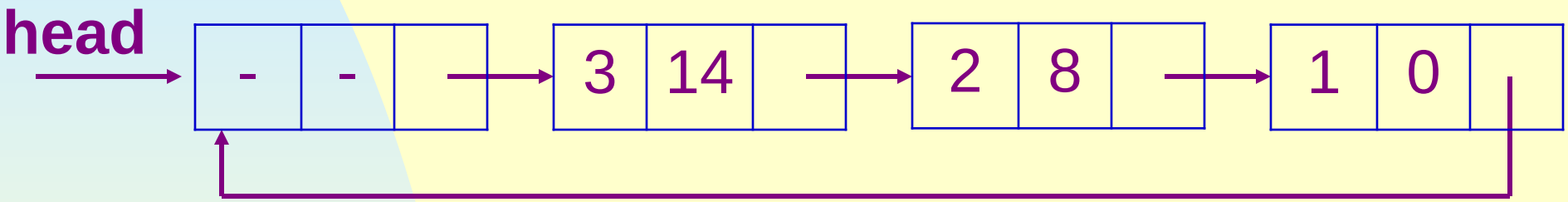
Assume a has m terms, b has n terms. The computing time is O(m+n).

4.7.3 Circular List Representation of Polynomials

Polynomials represented by circular lists with head node are as in the next slide:



(a) Zero polynomial



(b) $3x^{14} + 2x^{8} + 1$

Adding circularly represented polynomials

- The exp of the head node is set to –1 to push the rest of a or b to the result.

- Assume the begin() function for class CircularListWithHead return an iterator with its current points to the node head→link.

```
1 Polynomial Polynomial::operaor+(const Polynomial& b) const
2 { // *this (a) and b are added and the sum returned
3    Term temp;
4    CircularListWithHead<Term>::Iterator ai = poly.begin(),
5                                                bi = b.poly.begin();
6    Polynomial c; //assume constructor sets head→exp = -1
7    while (1) {
8       if (ai→exp == bi→exp) {
9          if (ai→exp == -1) return c;
10         int sum = ai→coef + bi→coef;
11         if (sum) c.poly.InsertBack(temp.Set(sum, ai→exp));
12         ai++; bi++; // to next term
13      }
```

```
14      else if (ai→exp < bi→exp) {
15           c.poly.InsertBack(temp.Set(bi→coef, bi→exp));
16           bi++;  // next term of b
17      }
18      else {
19           c.poly.InsertBack(temp.Set(ai→coef, ai→exp));
20           ai++;  // next term of a
21      }
22   }
23}
```

Experiment: P209-5

4.8 Equivalence Classes

Definition: **A relation ≡ over a set S is an equivalence relation over S iff it is symmetric, reflexive, and transitive over S.**

≡ partitions the set S into equivalence classes. For example, let S={0,1,..,10,11} **and**

0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0

S is partitioned into 3 equivalence classes:
{0,2,4,7,11}; {1,3,5}; {6,8,9,10}

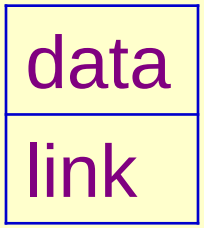
To determine equivalence classes, begin at 0 and find all pairs of (0, j). 0 and j are in the same class. By transitivity, all (j, k) imply k is in the same class. Continue this way until the entire class containing 0 has been found, marked. Then go on for another class…

Major difficulty: find the implied k. Use a strategy similar in the maze problem.

Representation:
m---the number of related pairs.
n---the number of objects.

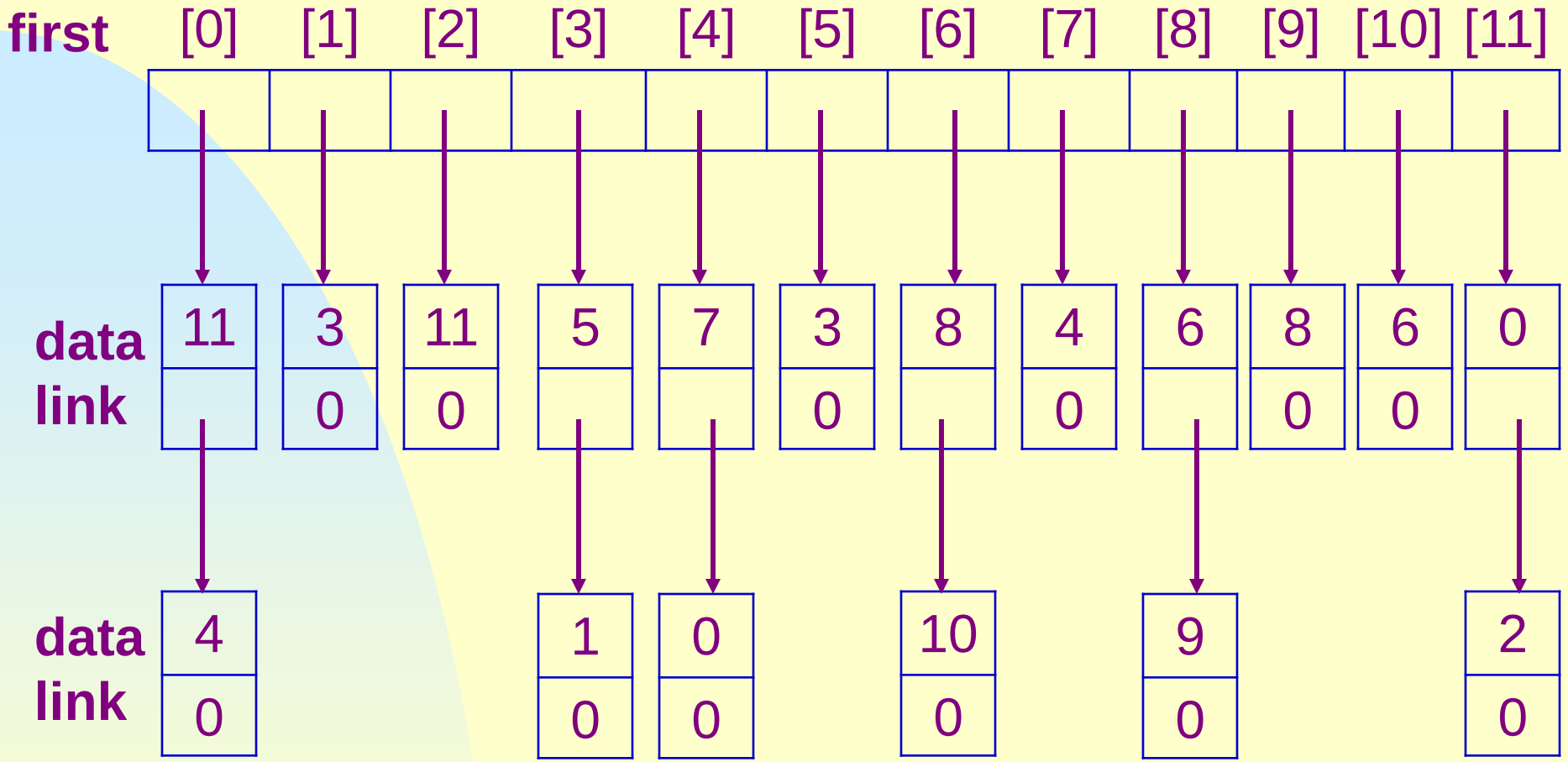
We could use a bool array pairs[n][n], such that pairs[i][j]=TRUE iff i ≡j .

- **To save space use a linked list to represent each row.**



- **To access row i at random, first[n].**
- **A bool array out[n] to mark objects visited.**

Given the above data, the lists will be as in the next slide.


first
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
data
link
11
3
11
5
7
3
8
4
6
8
6
0
0
0
0
0
0
0
data
link
4
1
0
10
9
2
0
0
0
0
0
0

- For each (i, j), 2 nodes used.
- To determine an equivalence class containing i, $0 \leq i < n$, and out[i]==false, each element in the list first[i] is printed.
- To process the remaining lists which , by transitivity, belong in the same class as i, a stack of their nodes is created, using inverse list skill.

Now the code for the algorithm.

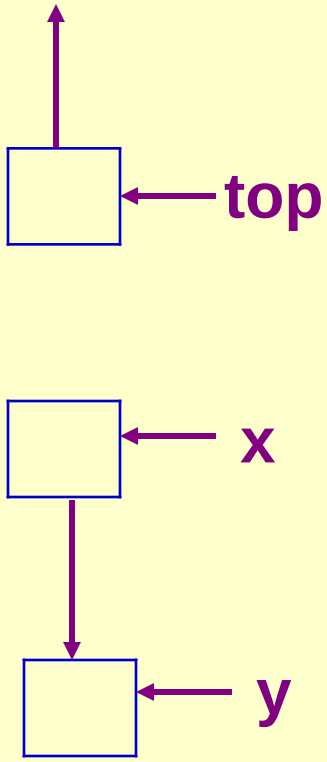
```
class ENode {
friend void equivalence();
Public:
    Enode(int d, Enode* l = 0) //constructor
        {data=d; link=l;}
private:
    int  data;
    ENode *link;
};
```

```
void Equivalence()
{
  // Input the equivalence pairs and output the equivalence
  // classes
   ifstream inFile(“equiv.in”,iso::in); // “equiv.in” is the input file
   if (!inFile) throw ”Cannot open input file.”;
   int i, j, n;
   inFile>>n; // read number of objects
```

```
// initialize first and out
Enode ** first = new Enode*[n];
bool *out = new bool[n];
fill(first, first + n, 0);
fill(out, out + n, false);

// Phase 1:input equivalence pairs
inFile>>i>>j;
while (inFile.good()) { // check end of file
    first[i] = new ENode(j, first[i]);
    first[j] = new Enode(i, first[j]);
    inFile >> i >> j;
}
```

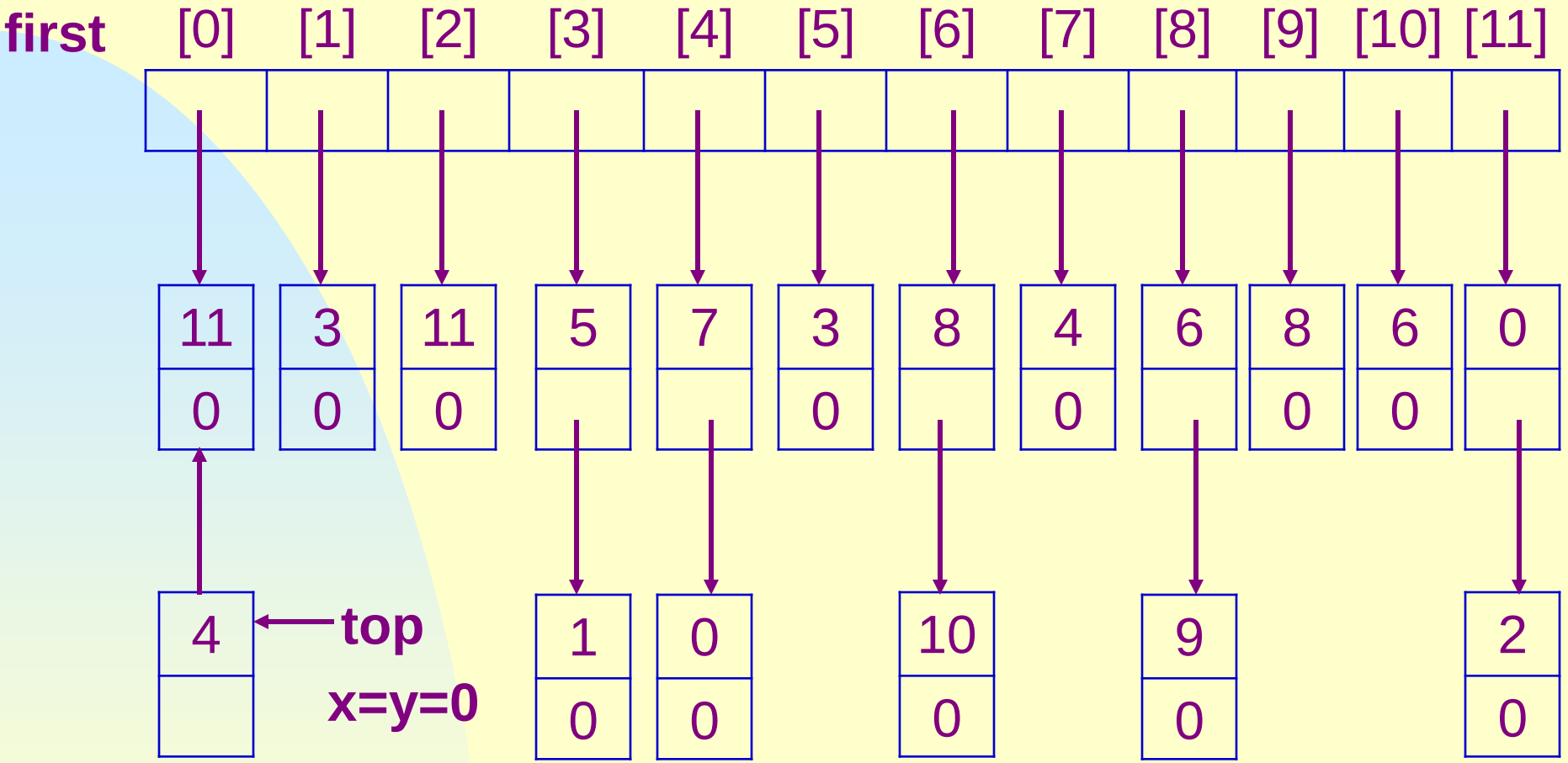
```
// Phase 2: output equivalence classes
for (i=0; i<n; i++)
  if (!out[i]) { // need to be output
     cout<<endl<<"A new class: "<<i;
     out[i]=true;
     ENode *x=first[i]; ENode *top=0; //initialize stack
     while (1) { // find rest of class
```

```
while (x)  {  // process the list
    j=x->data;
   if (!out[j]) {
      cout<<“,”<<j;
      out[j]=true;
      ENode *y=x->link;
      x->link=top;
      top=x;
      x=y;
    }
    else {ENode *y=x->link; delete x; x=y;}
 }   // end of while (x)
```



```
        if (!top) break;
        x=first[top->data];
        ENode *y=top; top=top->link; delete y; //unstack
      }  // end of while (1)
    }  // end of if (!out[i])
    delete [ ] first; delete [ ] out;
} // end of equivalence
```

Running process for the previous data:



A new class: 0, 11, 4 at this moment

Follow the algorithm using the above data, we get:

A new class: 0, 11, 4, 7 (skip over 0, 4, 0), 2 (skip over 11)

A new class: 1, 3, 5 (skip over 1, 3)

A new class: 6, 8, 10 (skip over 6, 6), 9 (skip over 8)

Analysis of Equivalence:

- initialization of first and out ---O(n)
- Phase 1---O(m+n)
- in Phase 2, each node is put onto the stack once, there are only 2m nodes. The for loop is executed n times. ---O(m+n)

Overall time: O(m+n)

Space required is also O(m+n)

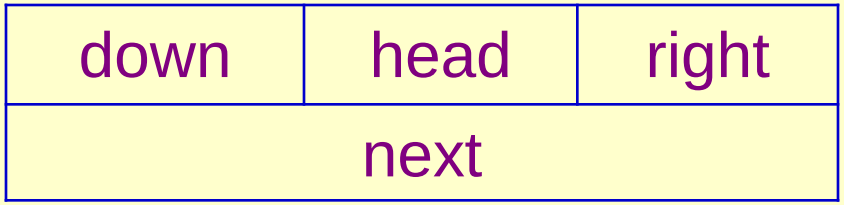
Exercises: P215-1

4.9 Sparse Matrices

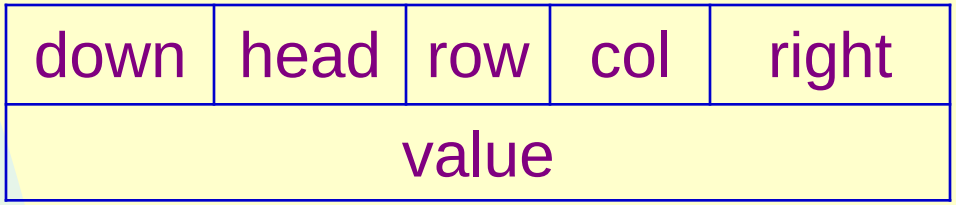
For various number of nonzero terms in sparse matrices, sequential representation is inefficient duo to the data movement involved. To overcome it, linked representation is a better choice.

- **each row , and each column also, is represented as a circularly linked list with head node.**

- **each node has a field head for distinguishing between head node and typical node:**



(a) head node, head=true



(b) typical node, head=false

- each head node is in 3 lists: row (linked by right), column (linked by down) and head (linked by next). The head node for row i is also for column i.

- each typical node is in 2 lists: row (linked by right) and column (linked by down).

- the head node for the head nodes list is a typical node with its row and col storing dimensions, right linked into the head nodes list, value storing the number of nonzero terms.

For an n × m matrix with r nonzero terms, the total nodes needed:

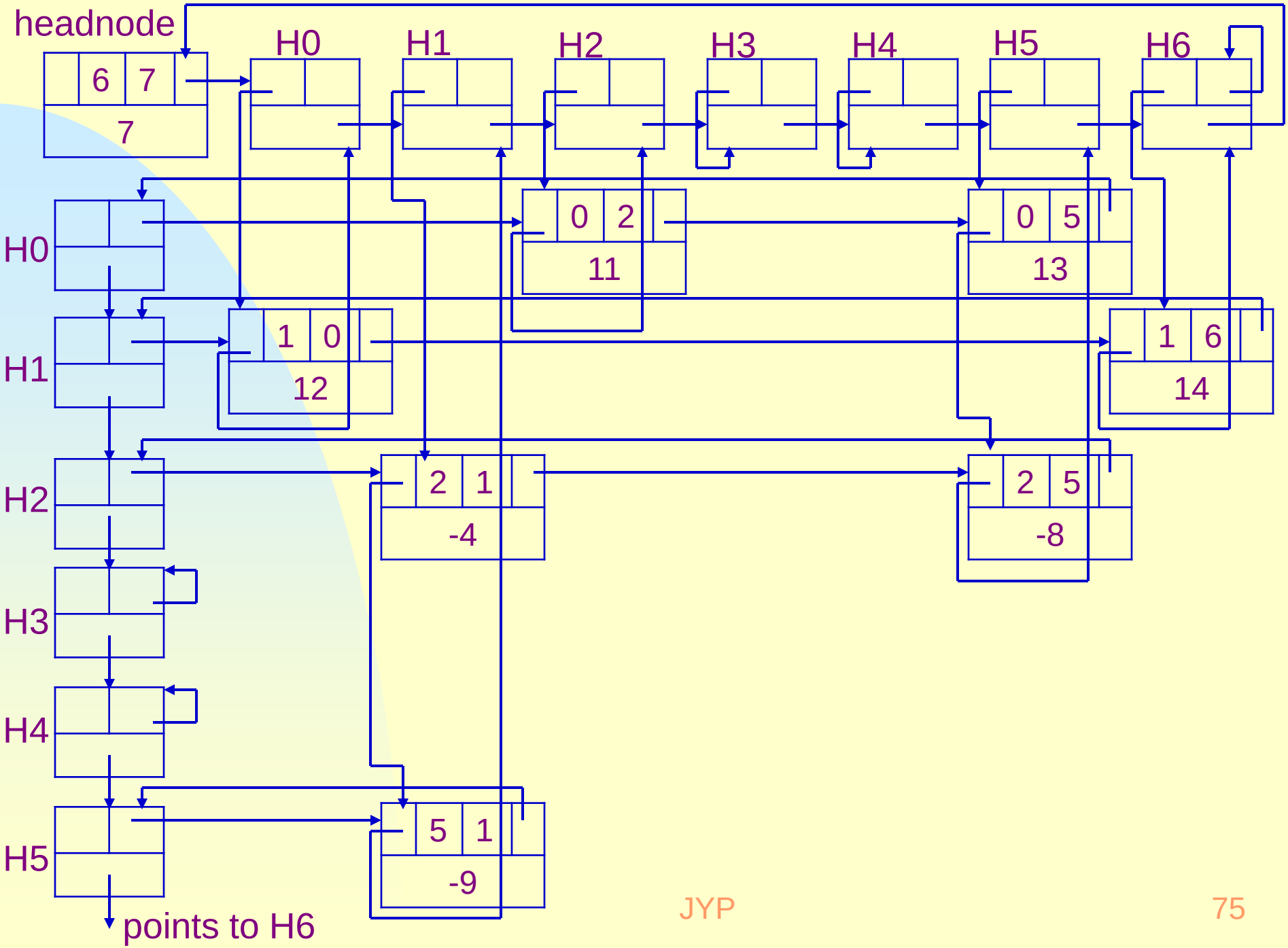
max{m,n} + r +1 (for the head of head)

```
struct Triple { int row, col,value;};
class Matrix;
class MatrixNode {
friend class Matrix;
friend istream& operator>>( istream&, Matrix&);
private:
    MatrixNode *down, *right;
    bool head;
    union {MatrixNode *next; Triple triple;  };
    MatrixNode ( bool, Triple* );   //constructor
};
```

```
MatrixNode::MatrixNode (bool b, Triple * t ) {
    head = b;
    if ( b ) {right=next=this;} // row/column head node,
                                // down will be set in program
    else triple=*t;
}

class Matrix {
friend istream& operator>>(istream&,Matrix& );
public:
    ~Matrix ( );
private:
    MatrixNode *headnode;   //points to the head of head nodes
};
```

0	0	11	0	0	13	0
12	0	0	0	0	0	14
0	-4	0	0	0	-8	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	-9	0	0	0	0	0

The above 6×7 matrix is represented as in the next slide.


headnode
6
7
7
H0
H1
H2
H3
H4
H5
H6
H0
0
2
11
0
5
13
H1
1
0
12
1
6
14
H2
2
1
-4
2
5
-8
H3
H4
H5
5
1
-9
points to H6

4.8.2 Sparse Matrix Input

Input:

n, m, r; i, j, a_{ij}; …; and (i, j, a_{ij}) ordered by (i, j).

Auxiliary array head of size max {n,m}, head[i] points to the head for column i and also row i--- permitting efficient random access.

The next of head node i initially keep track of the last node in column i.

Method:

- **set up all head nodes;**
- **set up each row list, simultaneously build the column lists;**
- **finally close the last row list, close all column lists, set up a head node for the head nodes list and link all head nodes together.**

```
istream& operator>>(istream& is, Matrix& matrix )
// read in a matrix and set up its linked representation
{
    Triple s;
    is >> s.row >> s.col >> s.value; //matrix dimensions
    int p=max(s.row, s.col);

    // set up head node for the head nodes list
    matrix.headnode=new MatrixNode ( false, &s );
    if ( p==0 ) {
        matrix.headnode→right = matrix.headnode;
        return is;
    }   // there is always the head of head nodes list

   // at least one row or column
   MatrixNode **head = new MatrixNode* [p];
```

```
for (int i = 0; i < p; i++) // initialize head nodes
    head[i] = new MatrixNode(true, 0 );
int CurrentRow=0;
MatrixNode *last=head[0]; // last node in current row

for (i = 0; i < s.value; i++ ) { //input triples
   Triple t;
   is >> t.row >> t.col >> t.value;
   if ( t.row > CurrentRow ) { // close current row
       last→right = head[CurrentRow];
       CurrentRow = t.row;
       last = head[CurrentRow];
    }
    last=last→right=new MatrixNode (false, &t); // into row
    head[t.col]→next=head[t.col]→next→down= last;//column
 }
```

```
    last→right=head[CurrentRow]; // close last row
    //close all column lists
    for (i=0; i<s.col; i++) head[i]→next→down=head[i];
    // link the head nodes together
    for (i=0; i< p-1; i++) head[i]→next =head[i+1];
    head[p-1]→next = matrix.headnode; // p≠0
    matrix.headnode→right = head[0];
    delete [ ] head;
    return is;
}
```

Computing time:

- **all head nodes---O(max(n,m))**
- **all triples---O(r)**

Total time: O(n+m+r)

4.8.3 Erasing a Sparse Matrix

Assume av points to the front of the available space list linked through the field right.

Erase a matrix by rows, each row list is circularly linked by the field right, can be done in O(1). Altogether in O(max {n, m})=O(n+m)

```
Matrix::~Matrix ( )
// return all nodes to the av list linked through right,
// av is a static variable of type MatrixNode*
{
   if (! headnode) return; // no nodes to dispose
   MatrixNode *x = headnode→right, *y;
   headnode→right=av; av=headnode;//return the head of head
   while ( x!=headnode ) {
        y = x→right;  x→right = av;  av = y;
        x = x→next;
   }
   headnode = 0;
}
```

Exercises: **P222-1,3**

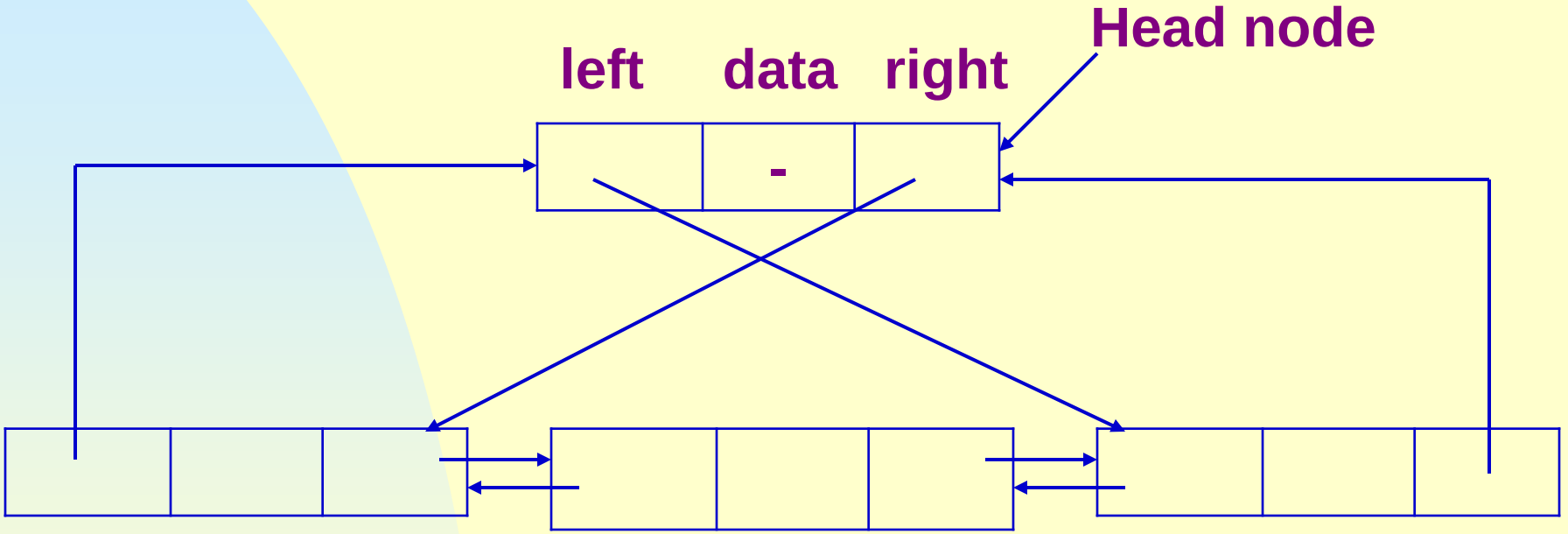
4.10 Doubly Linked Lists

Difficulties with singly linked list:

- **can easily move only in one direction**
- **not easy to delete an arbitrary node---requires knowing the preceding node**

A node in doubly linked list has at least 3 fields: data, left and right, this makes moving in both directions easy.

A doubly linked list may be circular. The following is a doubly linked circular list with head node:



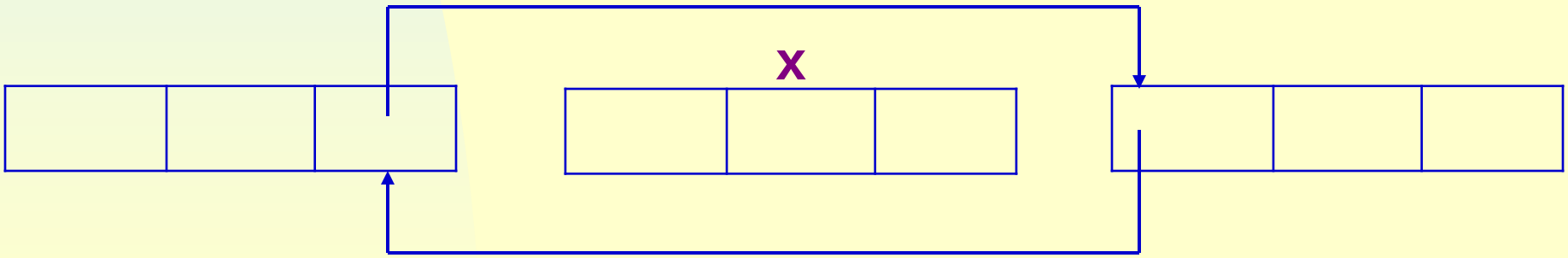
Suppose p points to any node, then

p == p→left→right == p→right→left

```
class DblList;

class DblListNode {
friend class DblList;
private:
    int data;
    DblListNode *left, *right;
};

class DblList {
public:
    // List manipulation operations
    …
private:
    DblListNode *first; // points to head node
};
```

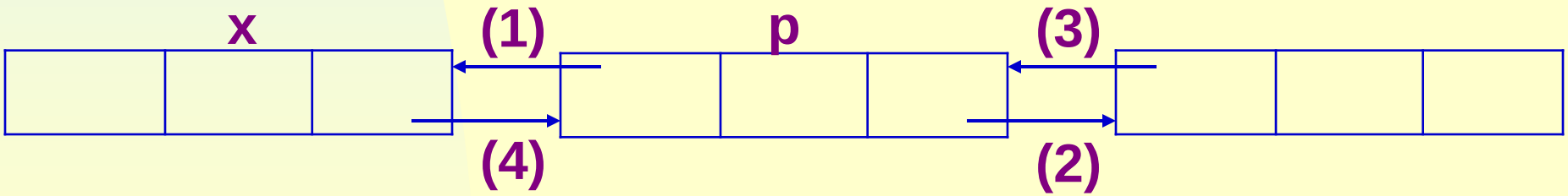
Delete

```
void DblList::Delete(DblListNode *x )
{
   if(x == first) throw "Deletion of head node not permitted";
   else  {
      x→left→right = x→right;
      x→right→left = x→left;
      delete x;
   }
}
```



Insert

```
void DblList::Insert(DblListNode *p, DblListNode *x )
{ // insert node p to the right of node x

        p→left = x;              // (1)
        p→right = x→right;  // (2)
        x→right→left = p;    // (3)
        x→right = p;            // (4)
}
```



Exercises: P225-2

4.11 Generalized Lists

4.11.1 Representation of Generalized Lists

Consider a linear list of $n \geq 0$ elements.

$$A = (\alpha_0, \ldots, \alpha_{n-1})$$

where α_i $(0 \leq i \leq n-1)$ can only be an atom, the only structure property is α_i precedes α_{i+1} $(0 \leq i < n-1)$.

Definition: A generalized list, A, is a finite sequence of $n \geq 0$ elements $\alpha_0, \ldots, \alpha_{n-1}$, where α_i be either an atom or a list. The elements α_i that are not atoms are said to be the sublists of A.

Note the definition is recursive.

The list A is written as $A = (\alpha_0, \ldots, \alpha_{n-1})$

A --- name

n --- length

α_0 --- head

$(\alpha_1, \ldots, \alpha_{n-1})$ --- tail

By convention

list name --- capital letters

atoms --- lowercase letters

Examples:

(1)A = ( )

(2)B = (a, (b, c))

(3)C = (B, B, ( ))

(4)D = (a, D) = (a, (a, (a, …)

2 consequences:

(1)lists may be shared as in (3)

(2)lists may be recursive as in (4)

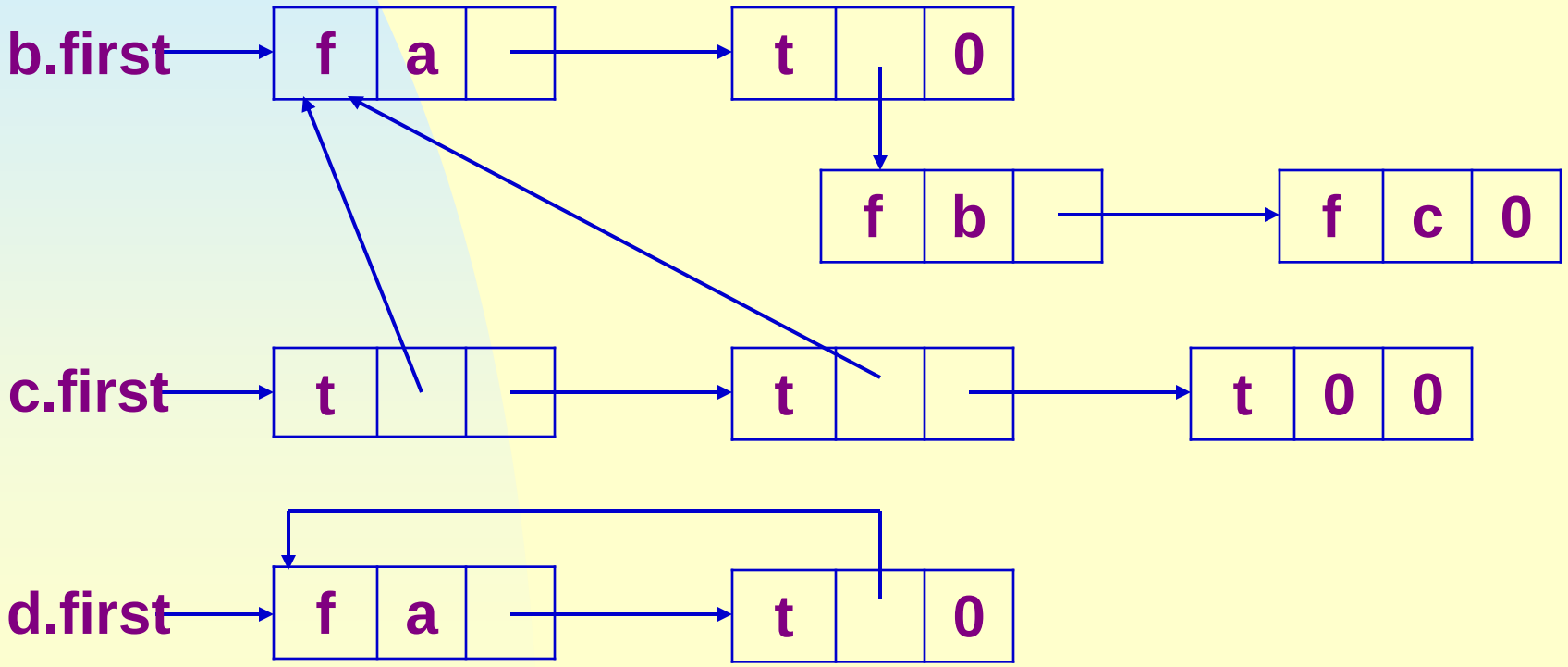
In general, every generalized list can be represented using the node structure:

tag=false/true	data/down	next

```
template <class T> class GenList;
template <class T>
class GenListNode {
friend class GenList;
private:
    GenListNode <T> *next;
    bool tag;
    union { T data;  GenListNode<T> *down;};
};

template <class T>
class GenList {
public:
    // List manipulation operations;
private:
    GenListNode<T> *first;
};
```

For any list A, the next points to the tail of A, the data/down can hold an atom (when tag==false) or a pointer to the list of head(A).

a.first=0 empty list



4.11.2 Recursive Algorithms for Lists

A recursive algorithm consists of two components:

- **workhorse---the recursive function itself.**
- **driver---the function that invokes the recursive function at the top level.**

4.11.2.1 Copying a List

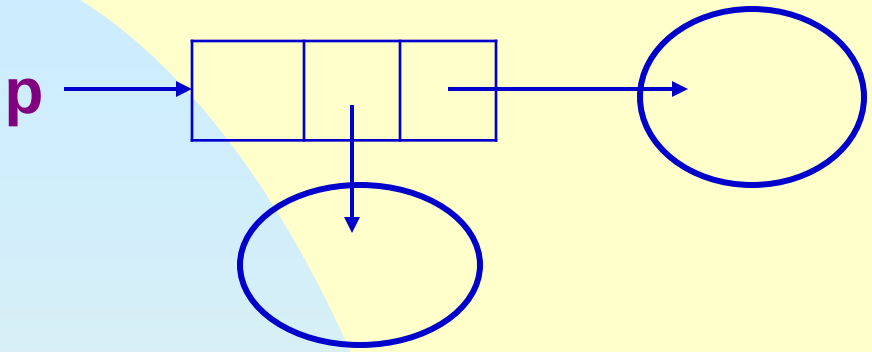
Produces an exact copy of a nonrecursive list l in which no sublists are shared.

Copy

```
template <class T>
void  GenList<T>::Copy(const GenList<T>& l) // Driver
{  first = Copy(l.first);  }
```

```
template <class T>
GenListNode<T> *GenList<T>::Copy(GenListNode<T>* p)
// Workhorse
// copy the non-recursive list with no shared sublists
{
   GenListNode<T> *q=0;
   if (p) {
        q =new GenListNode<T>;
        q→tag = p→tag;
        if   (p→tag) q→down = Copy(p→down);
        else q→data = p→data;
        q→next = Copy(p→next);
    }
    return q;
}
```

The correctness can be verified by induction.

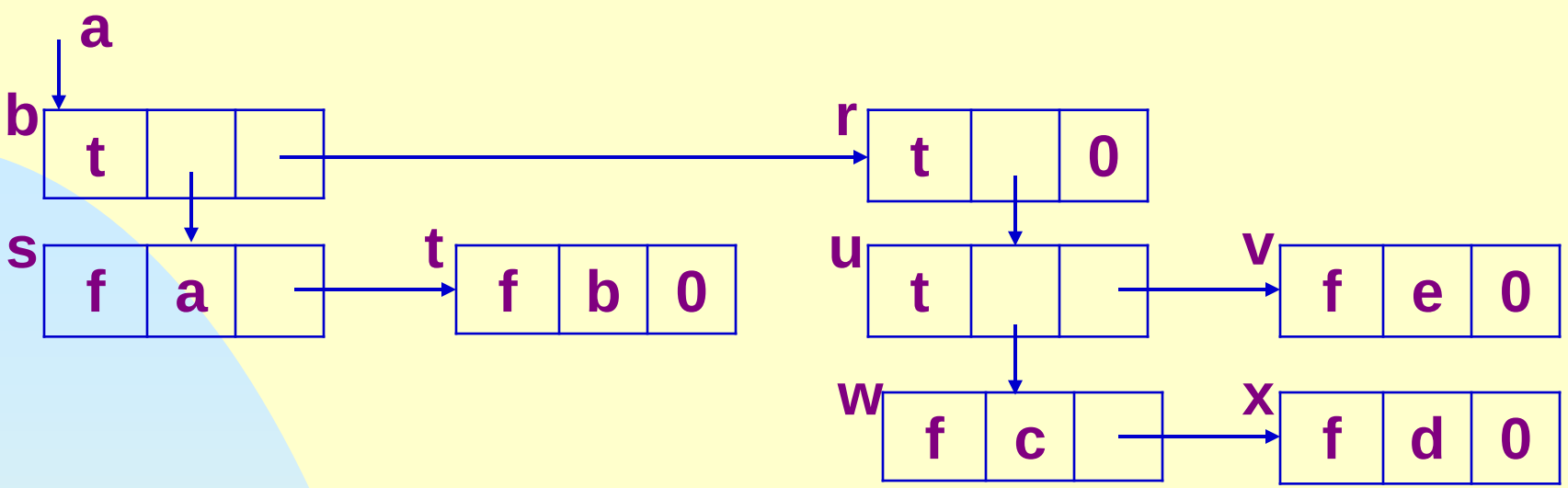


Let us run Copy for A = ((a, b), ((c, d), e)) as in the next slide.

Computing time:

- **nodes with tag==false, visited twice.**
- **nodes with tag==true visited three times.**

If the list has m nodes, the time is O(m).



Level	p	level	p	level	p	level	p
1	b	1	b	5	x	3	u
2	s	2	r	4	w	2	r
3	t	3	u	3	u	3	0
4	0	4	w	4	v	2	r
3	t	5	x	5	0	1	b
2	s	6	0	4	v		

4.10.2.2 List Equality

Two lists are identical if they have the same structure and the same data in corresponding data members.

Equal

```
// Driver
template <class T>
bool GenList<T>::operator==(const GenList<T>& l) const
{ // *this and l are non-recursive lists
  // returns true if the two lists are identical.
  return Equal(first, l.first);
}
```

```
// Workhorse– assume to be a friend of GenListNode
template <class T>
bool Equal(GenListNode<T>* s, GenListNode<T>* t)
{
   if ((!s) && (!t)) return true;
   if ( s && t && (s→tag == t→tag) )
      if (s→tag)
         return Equal(s→down, t→down)
                  && Equal(s→next, t→next);
      else return (s→data == t→data)
                  && Equal(s→next, t→next);
   return false;
}
```

Computing time:
No more than linear when no sublists are shared.
Note if a sublist is shared, it may be visited many times.

Note the algorithm terminates as soon as it discover two lists are not identical.

4.10.2.3 List Depth

$$\text{Depth}(s)=\begin{cases} 0 & \text{if } s \text{ is an atom or empty list} \\ 1+\max\{\text{depth}(x_0),\ldots,\text{depth}(x_{n-1})\} & \text{if } s \text{ is the list } (x_0,\ldots,x_{n-1}),\ n\geq 0 \end{cases}$$

Depth

```
// Driver
template <class T>
int GenList::Depth()
{ // compute the depth of a non-recursive list
   return Depth(first);
}
```

```
// Workhorse
template <class T>
int GenList::Depth(GenListNode<T>* s)
{
    if (!s) return 0;
    GenListNode<T>* current=s; int m=0;
    while (current) {
        if (current→tag) m=max(m,Depth(current→down));
        current=current→next;
     }
    return m+1;
}
```

4.10.3 Reference Counts,Shared and Recursive Lists

Sharing of sublists:

- **saving storage.**
- **consistency.**

To facilitate specification, extend the definition of a list to allow for naming of sublist. E.g.:

A=(a, (b,c)), Z=(b,c) ➔ A=(a, Z(b, c)) and to be consistent, A is written as A(a, Z(b, c)).

Problems with lists sharing:

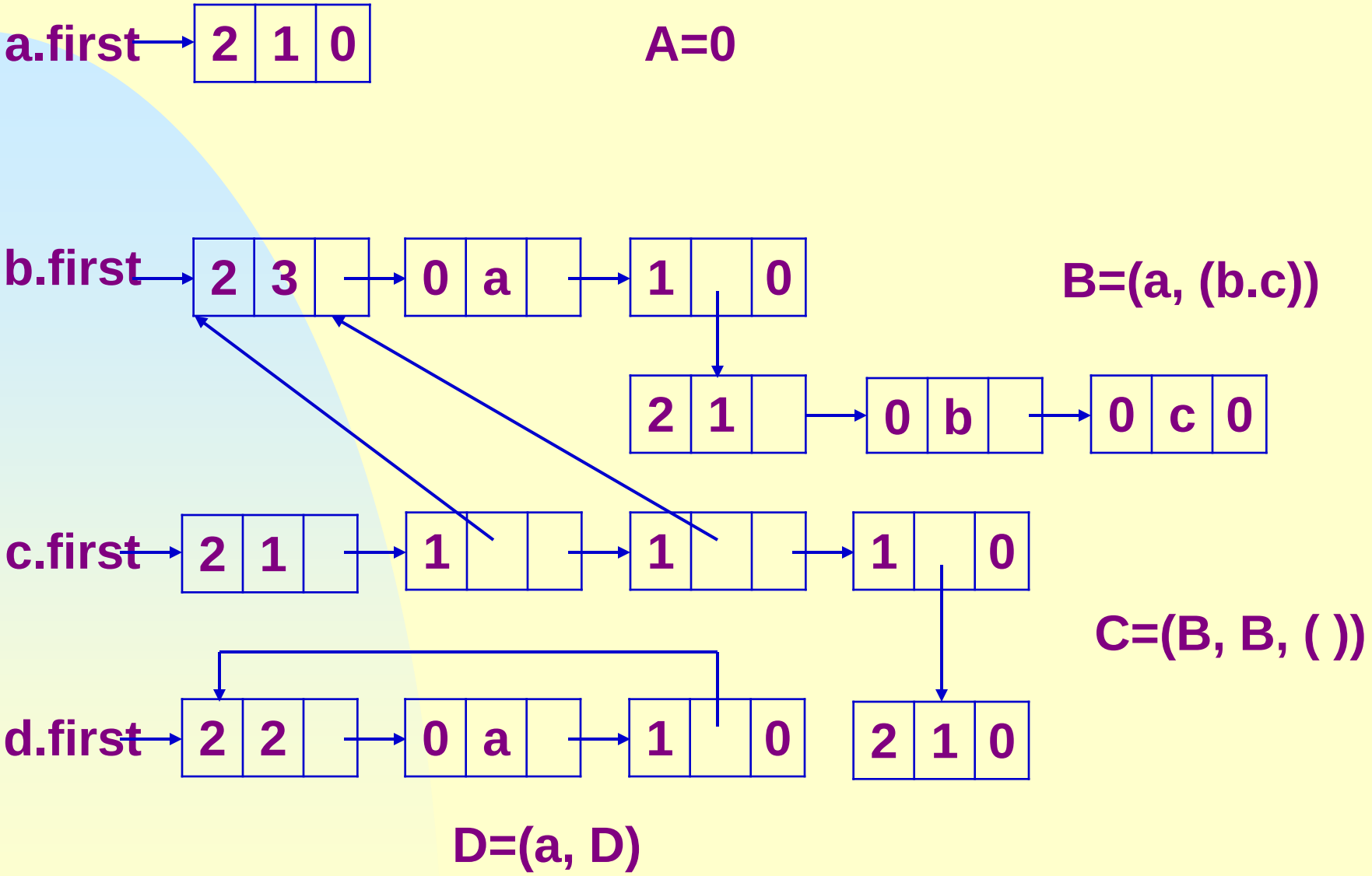
(1) Refer to the Fig. on the slide 93, if the first node of B is deleted, the pointers from C should be updated, but normally we do not know all the pointers from which a particular list is being referenced. --- solution: head node.

(2) How to determine whether or not the lists nodes may be actually freed? --- solution: reference counts, using the head node. The reference count of a list is the number of pointers to it.

We need some change in the class GenListNode:

```
template <class T>
class GenListNode<T> {
friend class GenList<T>;
private:
    GenListNode<T> *next;
    int tag;  // 0 for data, 1 for down, 2 for ref
    union {
        T data;
        GenListNode<T> *down;
        int ref;
    };
};
```

With head node, lists A= ( ), B = (a, (b, c)), C = (B, B, ( )) and D = (a, D) will be represented as in the next slide.


a.first
2 1 0
A=0
b.first
2 3
0 a
1 0
B=(a, (b.c))
2 1
0 b
0 c 0
c.first
2 1
1
1
1 0
C=(B, B, ( ))
d.first
2 2
0 a
1 0
2 1 0
D=(a, D)

Now the ideas of erasing list:
Decrement the reference count by 1, if it becomes 0, then examining the top level nodes, any sublists encountered are erased and finally, the top level nodes are linked into the av list.

~GenList

```
// Driver
template <class T>
GenList::~GenList()
{ // Each head node has a reference count.
    if (first) { Delete(first); first = 0;}
}
```

```
// Workhorse
template <class T>
void GenList<T>::Delete(GenListNode<T>* x)
{
   x→ref--; // decrement reference count of the head node
   if ( !x→ref )
   {
        GenListNode<T> *y=x; // y traverses top level of x.
        while (y→next)  {
            y=y→next;  if (y→tag==1) Delete(y→down);
        }
        y→next = av;
        av = x;
    }
}
```

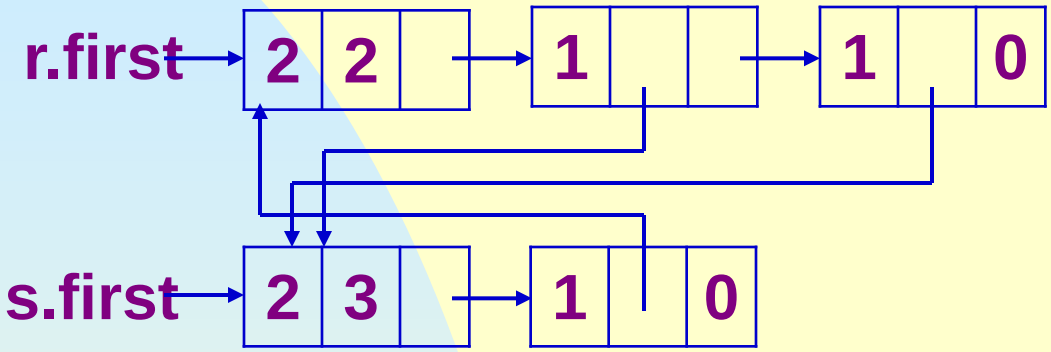
Refer to slide 106, a call to b.~GenList( ) makes:

- ref(b)→2. After that

A call to c.~GenList( ) results in:

- ref(c)→0, ref(b)→1→0.
- all the six nodes of B(a, (b, c)) returned to av list.
- the empty list node of C(B, B, ()) returned to av.
- the top level nodes of c returned to av.

For recursive lists, reference count never becomes 0, even if it is no longer accessible.



After calls to r.~GenList( ) and s.~GenList( ), ref(r) =1, ref(s)=2, but the structure is no longer being used. --- there is no simple way to solve it.

Exercises: P240-6

Chapter 5
Trees

5.1 Introduction

5.1.1 Terminology

Definition: A tree is a finite set of one or more nodes such that

(1) There is a specially designated node called **root**.

(2) The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree. $T_1, \ldots, T_n$ are called **subtrees** of the root.

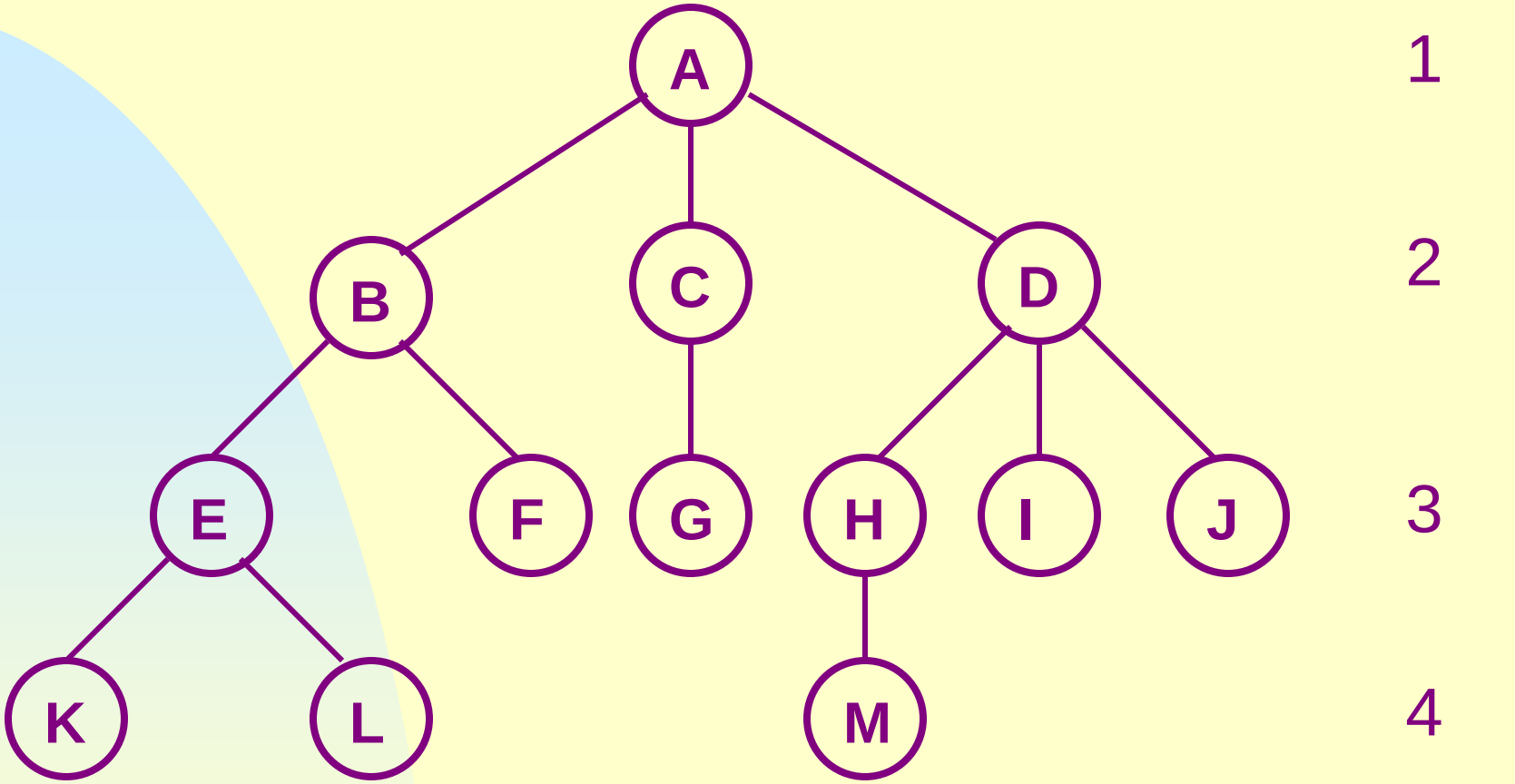


Fig. 5.2 A sample tree

A **node** stands for the item of information plus the branches to other nodes.

The number of subtrees of a node is its **degree**. Nodes with degree 0 are **leaf** or **terminal** nodes, others are **nonterminals**.

The roots of the subtrees of a node X are the **children** of X, X is the **parent** of its children. Children of the same parent are **siblings**.

The **degree of a tree** is the maximum of the degree of the nodes in the tree.

The ancestors of a node are all the nodes along the path from the root to that node.

The level of a node is defined by letting the root be at level 1, if a node is at level l, then its children are at l+1.

The height or depth of a tree is the maximum level of any node in the tree.

5.1.2 Representation of Trees

For a tree of degree k, we could use a tree node that has fields for data and k pointers to the children as below:

Fig.5.4 Possible node structure for a tree of degree k

However, this is very wasteful of space as Lemma5.1 in the next slide shows.

Lemma5.1: If T is a k-ary tree with n nodes, each having a fixed size as in Fig.5.4, then n(k-1)+1 of the nk child fields are 0, $n \geq 1$.

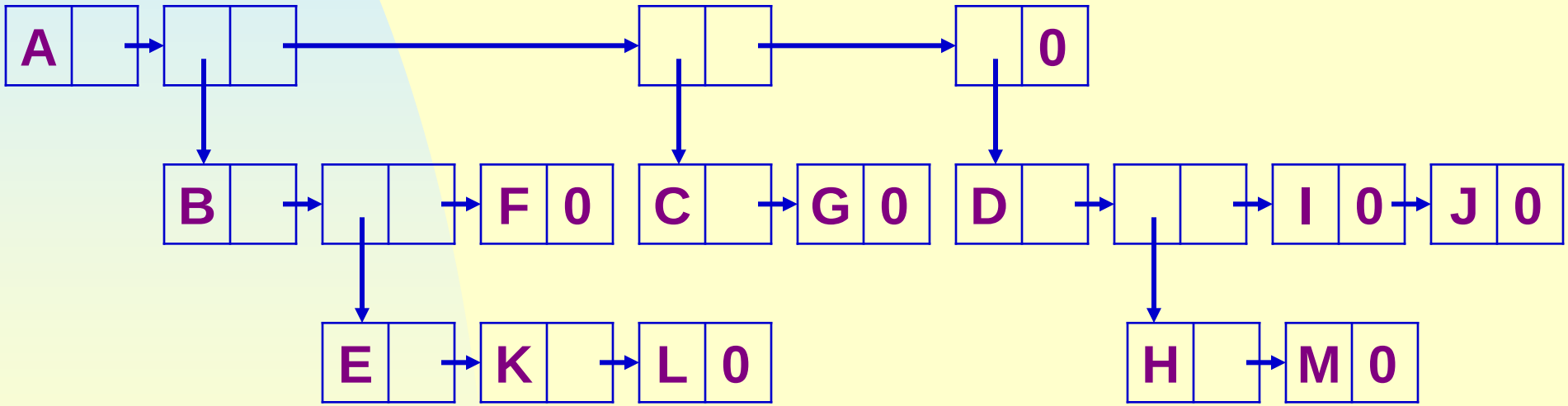
Proof:

- each non-zero child field points to a node
- there is exactly one pointer to each node other than the root
- the number of non-zero child fields in an n node tree is n-1
- the number of zero fields is nk-(n-1)=n(k-1)+1.

The tree of Fig.5.2 could written as a list:

(A(B(E(K, L), F), C(G), D(H(M), I, J)))

Use the general list representation:



In later section, we'll see how to use binary tree to represent forest, and hence trees.

5.2 Binary Trees

5.2.1 The Abstract Data Type

Definition: A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.

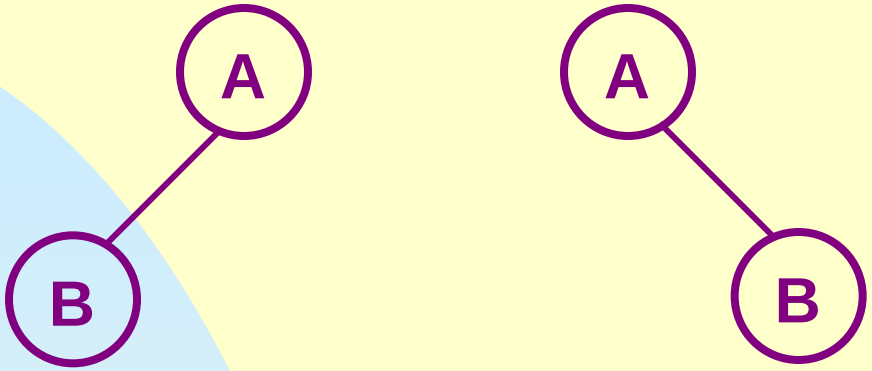
ADT 5.1

```
template <class T>
class BinaryTree
{ // A finite set of nodes either empty or consisting of
  // a root node, left BinaryTree and right BinaryTree.
public:
    BinaryTree ();
    // creates an empty binary tree

    bool IsEmpty ();
    // return true iff the binary tree is empty

    BinaryTree(BinaryTree<T>& bt1, T& item,
                                        BinaryTree<T>& bt2);
    // creates a binary tree whose left subtree is bt1,
    // right subtree is bt2, and root node contain item.

    BinaryTree  LeftSubtree();
    // return the left subtree of *this
```

```
    T RootData();
    // return the data in the root of *this

    BinaryTree  RightSutree();
    // return the right subtree of *this
};
```

Distinction between a binary tree and a tree:

- **no tree has 0 node, but there is an empty binary tree.**
- **a binary tree's root always has two subtrees: the left and the right, although they may be empty.**



So the above two binary trees are different: the first has an empty right subtree while the second has an empty left subtree. But as trees, they are the same.

In a binary tree, the number of non-empty subtrees of a node is its degree.

5.2.2 Properties of Binary Trees

Lemma 5.2 [Maximum number of nodes]:

(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.

(2) The maximum number of nodes in a binary tree of depth k is 2^k-1, $k \geq 1$.

Proof:

Let M_i be the maximum number of nodes on level i

(1) Induction on level i.

i=1, only the root, $M_i = 2^{1-1} = 2^0 = 1$.

Assume $M_{i-1} = 2^{(i-1)-1}$ for i>1.

Since each node has a maximum degree of 2, $M_i = 2 * M_{i-1} = 2*2^{i-2} = 2^{i-1}$.

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=1}^{k} M_i = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1.$$

Lemma 5.3 [Relation between number of leaf nodes and degree-2 nodes]:

For any nonempty binary tree T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof:

Let n_1 be the number of nodes of degree 1 and n the total number of nodes, we have

$$n = n_0 + n_1 + n_2 \qquad (5.1)$$

Each node except for the root has a branch leading into it. If B is the number of branches, then $n = B+1$. And also $B = n_1 + 2n_2$, hence

$$n = n_1 + 2n_2 + 1 \qquad (5.2)$$

(5.1) – (5.2): $0 = n_0 - n_2 - 1$, i.e., $n_0 = n_2 + 1$.

Definition: A full binary tree of depth k is a binary tree of depth k having 2^k-1 nodes, k ≥ 0.

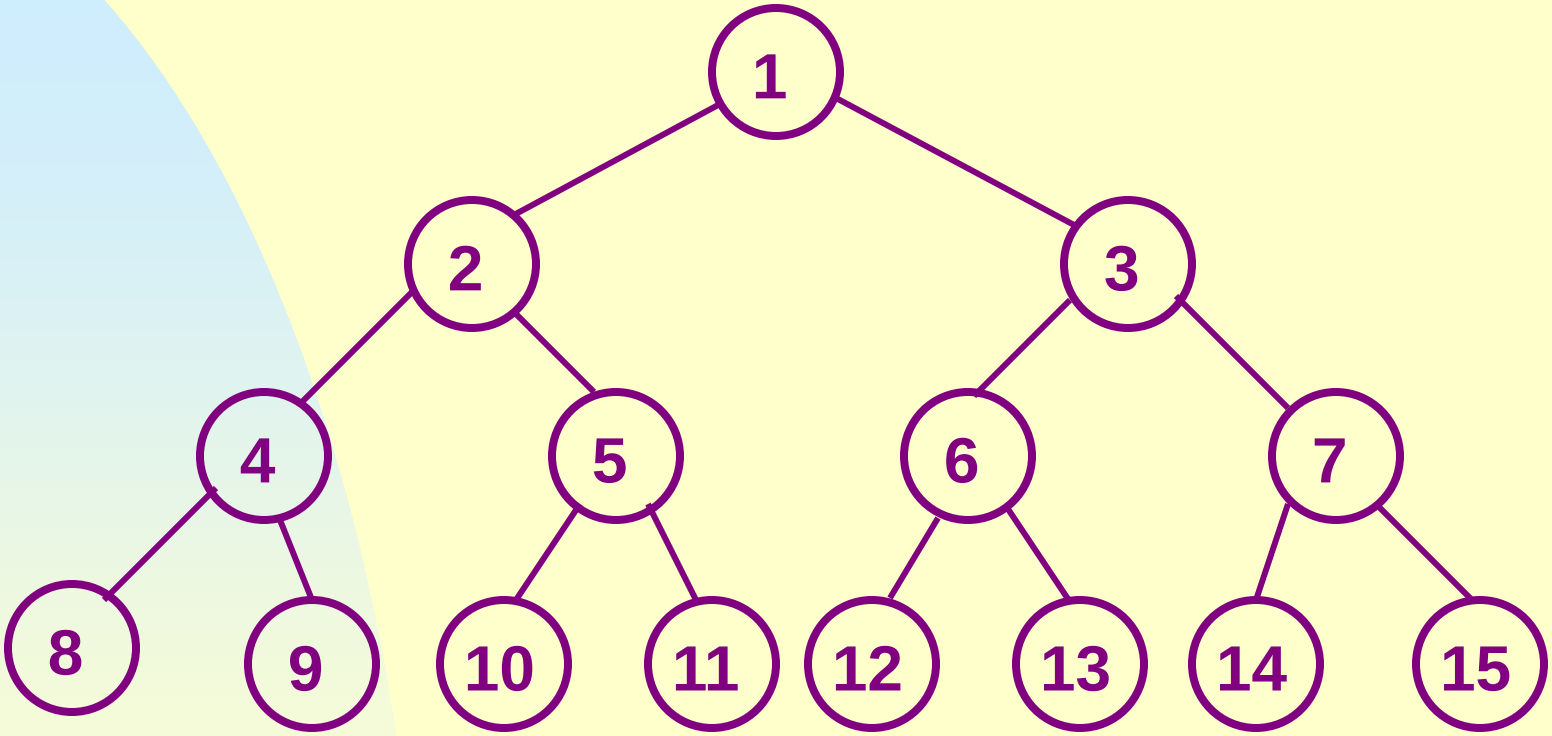


Fig. 5.11 Full binary tree of depth 4

Sequential numbering of the nodes in full binary tree: nodes are numbered 1 to n, from level 1 to level k, and from left to right.

Definition: a binary tree with n nodes and depth k is **complete** iff its nodes corresponding to the nodes numbered from 1 to n in the full binary tree of depth k.

From Lemma5.2, the height of a complete binary tree with n nodes is $\lceil \log_2(n+1) \rceil$.

5.2.3 Binary Trees Representations

5.2.3.1 Array Representation

The nodes may be stored in a one dimensional array tree, with the node sequentially numbered i being stored in tree[i].

Lemma 5.4: if a complete binary tree with n nodes is represented sequentially, then for any nodes with index i, $1 \le i \le n$, we have

(1) parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If i=1, i is the root and has no parent.

(2) LeftChild(i) is at 2i, if $2i \leq n$. If $2i>n$, i has no left child.

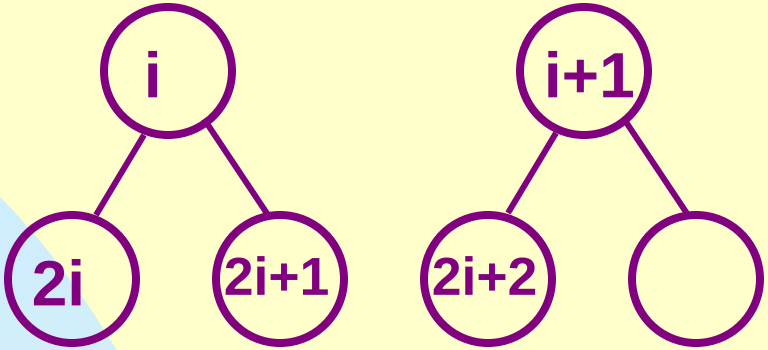
(3) RightChild(i) is at 2i+1, if $2i+1 \leq n$. If $2i+1>n$, i has no right child.

Proof: prove (2). (3) follows from (2) and the nodes numbering from left to right. (1) follows from (2) and (3).

Induction on i.

i=1, clearly the left child is at 2 unless 2>n, in which case i has no left child.

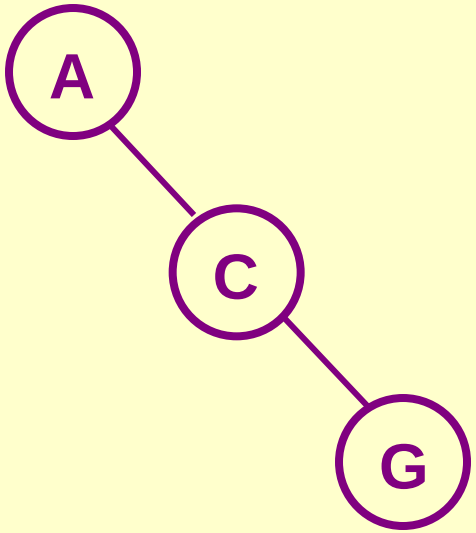
Assume for all j, $1 \leq j \leq i$, LeftChild(j) is at 2j.



The two nodes immediately preceding LeftChild(i+1) are the right and left children of i, hence LeftChild(i+1) is at 2i+2=2(i+1) unless 2(i+1)>n, in which case i+1 has no left child.

The array representation can be used for all binary trees.

- for a complete binary tree---no space wasted.
- for a skewed tree of depth k, in the worst case, it will require 2^k-1 spaces of which only k will be used, as shown below:



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
tree	---	A	---	C	---	---	---	G

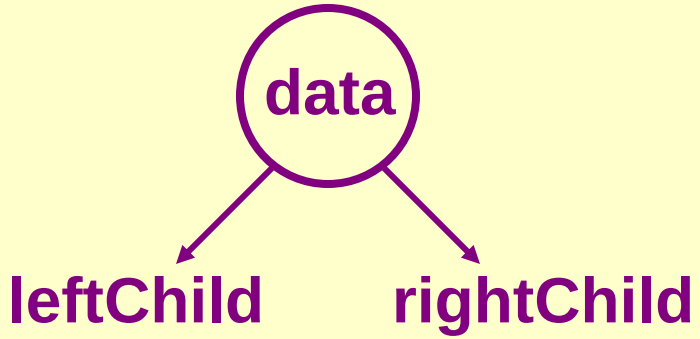
5.2.3.2 Linked Representation

General problem of sequential representation: insertion and deletion cause data movement.

The problem can be easily overcome through linked representation.

```
template <class T> class Tree;
class TreeNode {
friend class Tree<T>;
public:
    TreeNode (T& e, TreeNode<T>* left, TreeNode<T>* right)
      {data=e; leftChild=left; rightChild=right;}
private:
    T data;
    TreeNode<T>* leftChild;
    TreeNode<T>* rightChild;
};
```

leftChild	data	rightChild




```
template <class T>
class Tree {
public:
    // Tree operations
…
private:
    TreeNode<T>* root;
};
```

If necessary, a 4th field, parent, may be included in the node.

5.3 Binary Tree Traversal and Tree Iterators

5.3.1 Introduction

L---moving left, V---visiting the node, R---moving right, and adopt the convention of traversing left before right, we have 3 traversals:

LVR (inorder), LRV (postorder), VLR (preorder).

There is a natural correspondence between these traversals and producing the infix, postfix and prefix of an expression.

We use the following expression tree to illustrate each of these traversals.

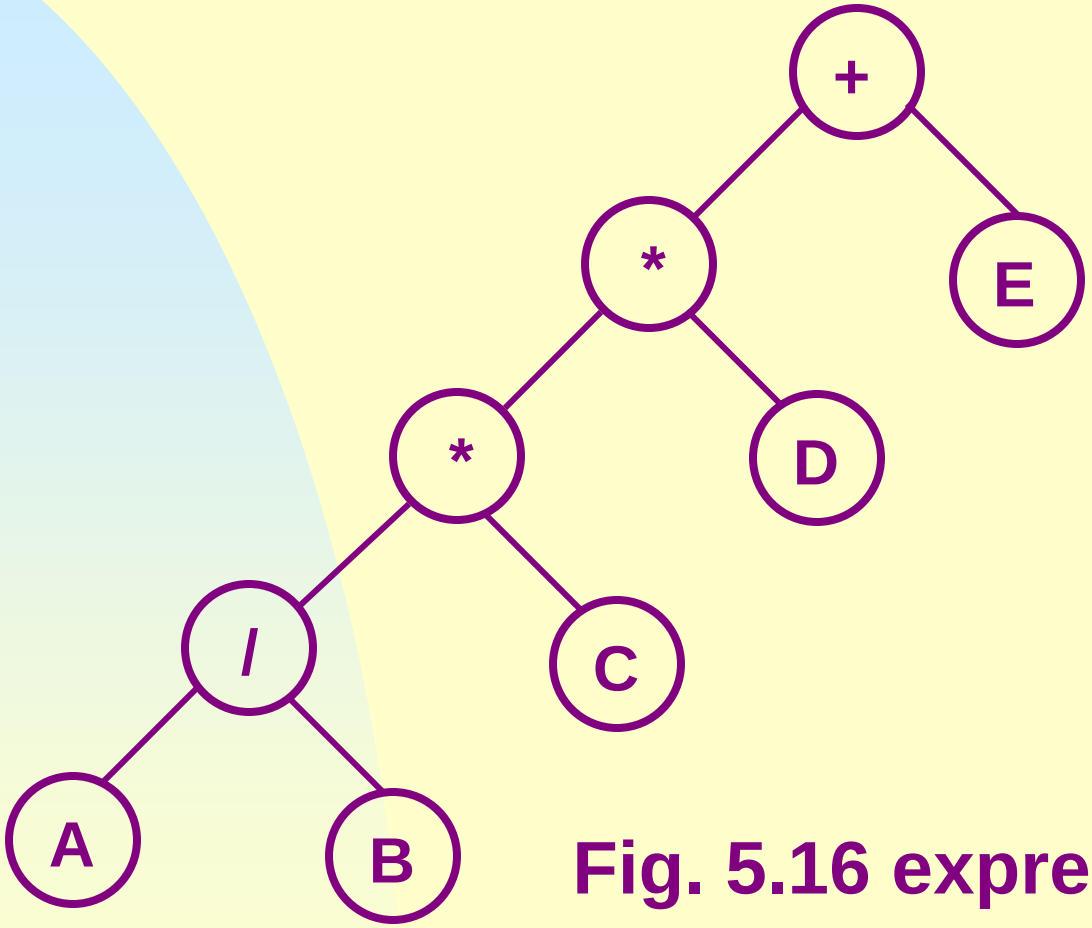


Fig. 5.16 expression tree

5.3.2 Inorder Traversal

```
template <class T>
void Tree<T>::Inorder()
{ // driver as a public member
    Inorder(root);
}

template <class T>
void Tree<T>::Inorder (TreeNode<T>* currentNode)
{ // workhorse as a private member of Tree
    if (CurrentNode) {
        Inorder(currentNode→leftChild);
        Visit(currentNode)
        Inorder(currentNode→rightChild);
     }
}
```

Assume the Visit function has a single line of code:

```
cout <<CurrentNode→data;
```

For the tree of Fig. 5.16, the elements are output as:

A / B * C * D +E (infix)

5.3.3 Preorder Traversal

```
template <class T>
void Tree<T>::Preorder()
{ // Driver.
    Preorder(root);
}
```

```
template <class T>
void Tree<T>::Preorder(TreeNode<T>* currentNode)
{ // workhorse.
     if (currentNode) {
        Visit(currentNode);
        Preorder(currentNode→leftChild);
        Preorder(currentNode→rightChild);
     }
}
```

For the tree of Fig. 5.16, the elements are output as:

+ * * / A B C D E (prefix)

5.3.4 Postorder Traversal

```
template <class T>
void Tree<T>::Postorder()
{ // Driver.
    Postorder(root);
}

template <class T>
void Tree<T>::Postorder (TreeNode<T>* currentNode)
{ // Workhorse.
     if (currentNode) {
        Postorder(currentNode→leftChild);
        Postorder(currentNode→rightChild );
        Visit(currentNode);
     }
}
```

For the tree of Fig. 5.16, the elements are output as:

A B / C * D * E + (postfix)

5.3.5 Iterative Inorder Traversal

To implement a tree traversal by using iterators, we first need to implement a non-recursive tree traversal algorithm.

A direct way to do so is to use a stack.

```
1 template <class T>
2 void Tree<T>::NonrecInorder()
3 { // Nonrecursive inorder traversal using a stack
4   Stack<TreeNode<T>*> s; // declare and initialize a stack
5   TreeNode<T>* currentNode=root;
6   while (1) {
7     while (currentNode) { // move down leftChild
8       s.Push(currentNode); // add to stack
9       currentNode=currentNode→leftChild;
10    }
11    If (s.IsEmpty()) return;
12    currentNode=s.Top();
13    s.Pop(); // delete from stack
14    Visit(currentNode);
15    currentNode=currentNode→rightChild;
16  }
17}
```

The NonrecInorder USES-A template stack.

Definition: A data object of Type A USES-A data object of Type B if a Type A object uses a Type B object to perform a task. Typically, a Type B object is employed in a member function of Type A.

USES-A is similar to IS-IMPLEMENTED-IN-TERMS-OF, but the degree of using the Type B object is less.

Analysis of NonrecInorder:

- n---the number of nodes in the tree.
- every node is placed on the stack once, line 8, 9 and 11 to 15 are executed n times.
- currenetNode will equal 0 once for every 0 link, which is $2n_0 + n_1 = n_0 + n_1 + n_2 + 1 = n + 1$.

The computing time: O(n).

The space required for the stack is equal to the depth of the tree.

Now we use the function NonrecInorder to obtain an inorder iterator for a tree.

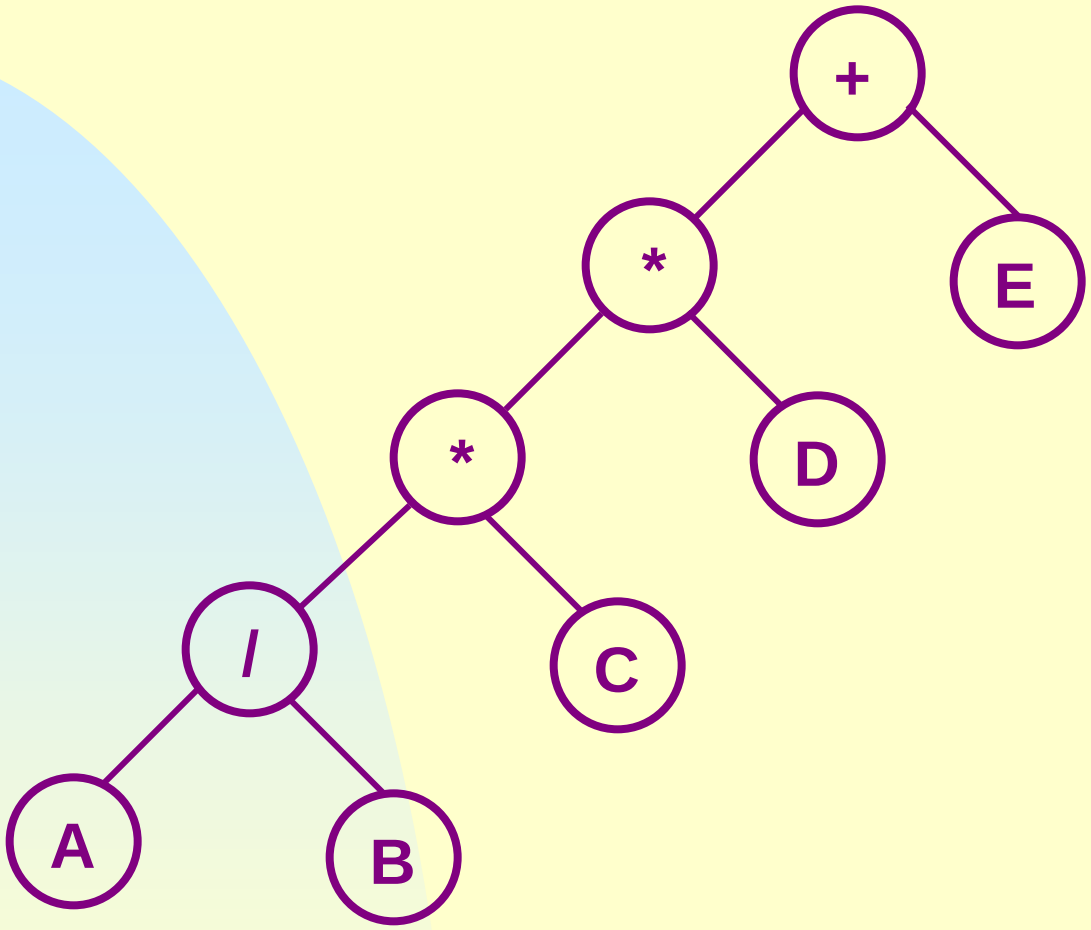
The key observation is that each iteration of the while loop of line 6-16 yields the next element in the inorder traversal of the tree.

```
class InorderIterator { // a public nested member class of Tree
public:
    InorderIterator() {currentNode=root;};
    T* Next( );
private:
    Stack<TreeNode<T>*> s;
    TreeNode<T>* currentNode;
};
```

```
T* InorderIterator::Next()
{
    while (currentNode) {
        s.Push(currentNode);
        currentNode=currentNode→LeftChild;
    }
    if (s.IsEmpty()) return 0;
    currentNode=s.Top();
    s.Pop();
    T& temp=currentNode→data;
    currentNode=currentNode→rightChild;
    return &temp;
}
```

5.3.6 Level-order Traversal

Level-order traversal visits the nodes in the order suggested in the full binary tree nodes numbering.



The level-order traversal of the left tree is:

+ * E * D / C A B

To implement level-order traversal, we use a queue.

```
template <class T>
void Tree<T>::LevelOrder()
{ // traverse the binary tree in level order
    Queue<TreeNode<T>*> q;
    TreeNode<T>* currentNode=root;
    while (currentNode) {
      Vist(currentNode);
      if (currentNode→leftChild)
            q.Push(currentNode→leftChild);
      if (currentNode→rightChild)
            q.Push(currentNode→rightChild);
      if (q.IsEmpty())return;
      currentNode=q.Front();
      q.Pop();
   }
}
```

Exercises: P267-4, 6

Experiment: P267-10

5.4 Additional Binary Tree Operations

5.4.1 Copying Binary Trees

```
template <class T>
Tree<T>::Tree(const Tree<T>& s)  // driver
{ // Copy constructor
    root = Copy( s.root );
}
```

```
template <class T>
TreeNode<T>* Tree<T>::Copy(TreeNode<T>* origNode)
// workhorse
{ // Return a pointer to an exact copy of the binary
  // tree rooted at origNode
    if (!origNode) return 0;
    return new TreeNode<T>(origNode→data,
                           Copy(origNode→leftChild),
                           Copy(origNode →rightChild));
}
```

5.4.2 Testing Equality

```
template <class T>
bool Tree<T>::operator==(const Tree& t) const
{
    return Equal(root, t.root);
}

template <class T>
bool Tree<T>::Equal(TreeNode<T>* a, TreeNode<T>* b)
{// Workhorse-
    if ((!a) && (!b)) return true; // both a and b are 0
    return (a && b          // both a and b are non-0
        && (a→data == b→data)                  //data is the same
        && Equal(a→leftChild, b→leftChild)          //left equal
        && Equal(a→rightChild, b→rightChild));      //right equal
}
```

Exercises: P272-1, P273-4

5.5 Threaded Binary Trees

5.5.1 Threads

Note that in the linked representation of binary tree, there are n +1 0 links, which is more than actual pointers.

A clever way to make use of these 0 links is to replace them by pointers, called threads, to other nodes in the tree.

The threads are constructed using the following rules:

(1) A 0 rightChild field at node p is replaced by a pointer to the inorder successor of p.

(2) A 0 leftChild field at node p is replaced by a pointer to the inorder predecessor of p.

The following is a threaded tree, in which node E has a predecessor thread pointing to B and a successor thread to A.

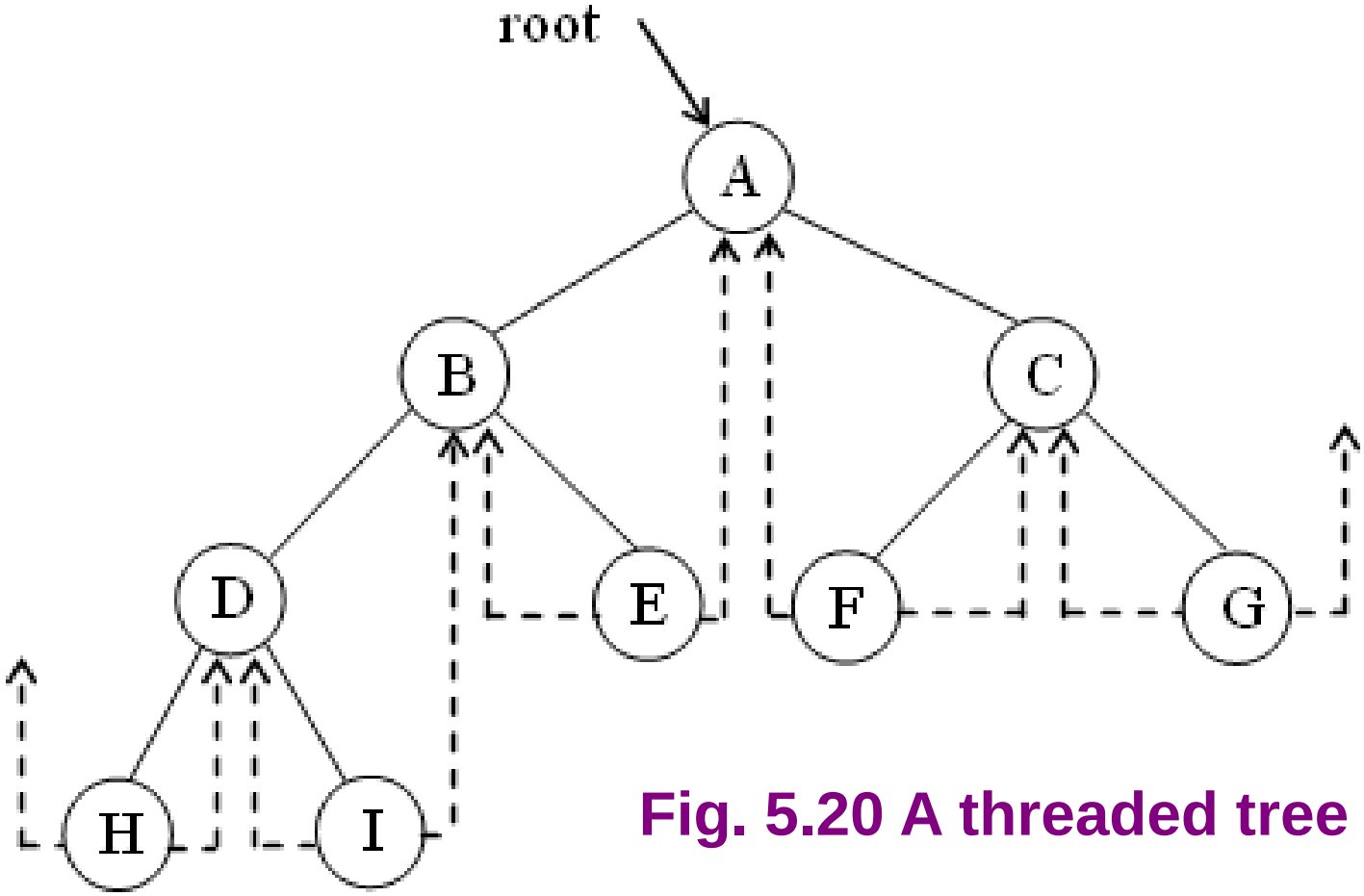


Fig. 5.20 A threaded tree

To distinguish between threads and normal pointers, add two bool fields:

- **leftThread**
- **rifgtThread**

If t→leftThread == true, then t→leftChild contains a thread, otherwise a pointer to left child. Similar for t→rightThread.

```
template <class T>
class ThreadedNode {
friend class ThreadedTree;
private:
    bool  leftThread;
    ThreadedNode<T> * leftChild;
    T data;
    ThreadedNode<T> * rightChild;
    bool  rightThread;
};
```

```
template <class T>
class ThreadedTree {
public:
    // Tree operations
…
private:
    ThreadedNode<T> *root;
};
```

Let ThreadedInorderIterator be a nested class of ThreadedTree:

```
class ThreadedInorderIterator {
public:
    T* Next();
    ThreadedInorderIterator()
        { currentNode = root; }
private:
    ThreadedNode<T>* currentNode;
};
```

To make the left thread of the first node in inorder and the right thread of the last node in inorder un-dangle, we assume a head node for all threaded binary tree, let the two threads point to the head.

The original tree is the left subtree of the head, and the rightChild of head points to the head itself.

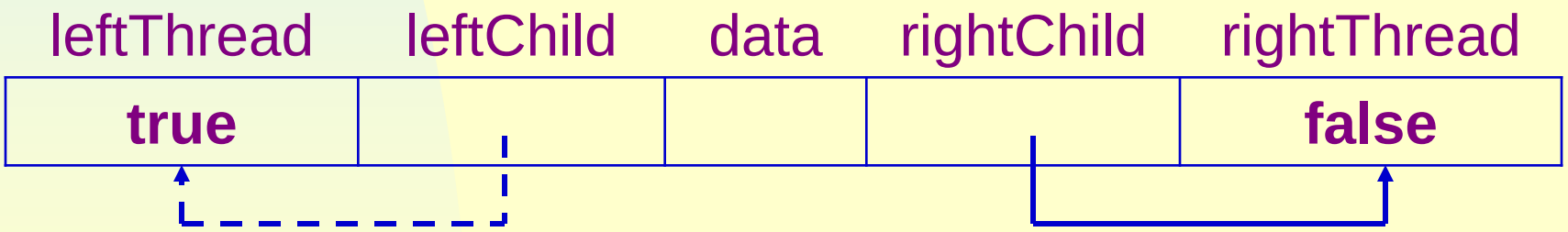


Fig.5.21 An empty threaded binary tree

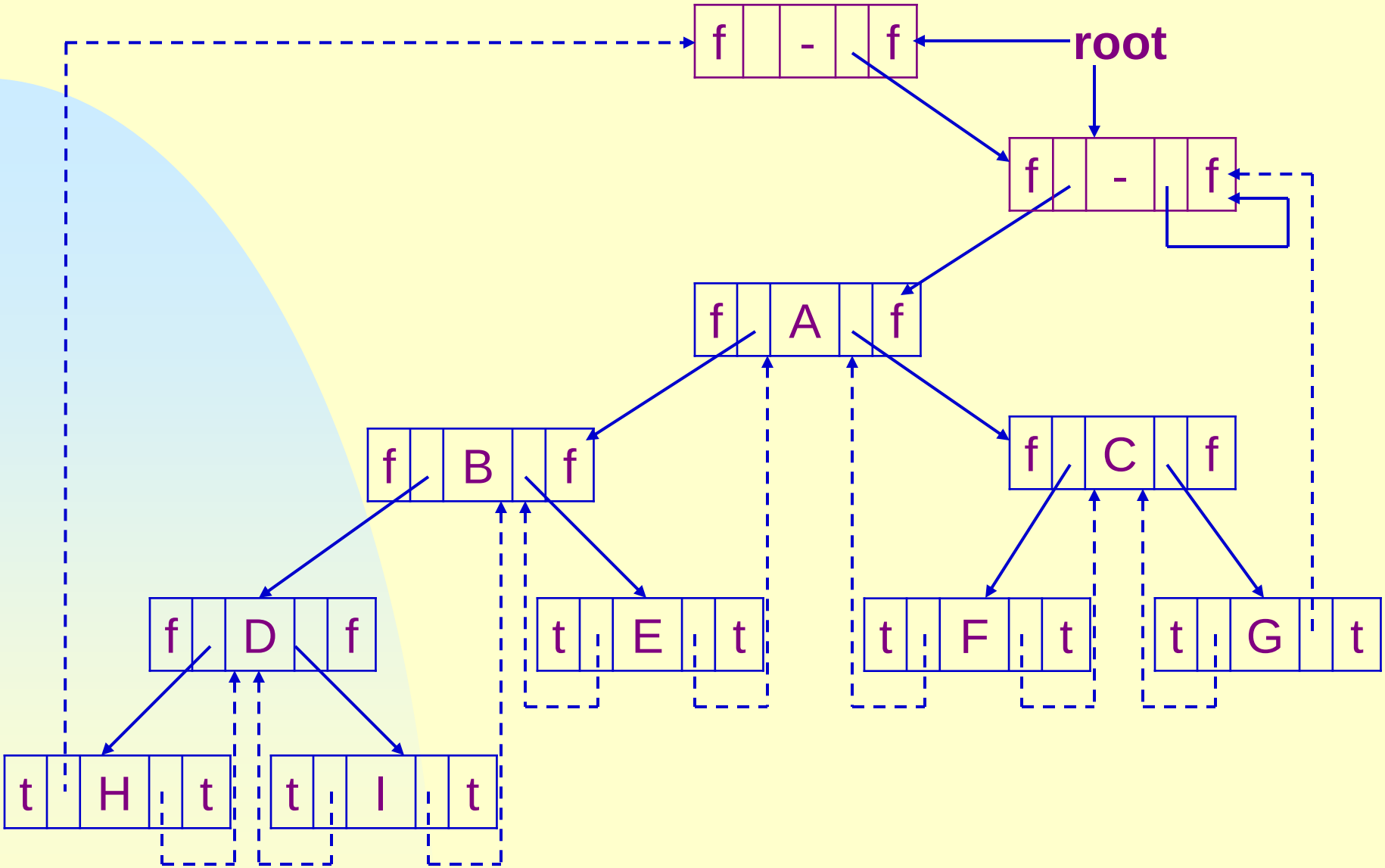


Fig.5.22 memory representation of threaded tree

we can see:

(1) The inorder successor of the head node is the first node in inorder;

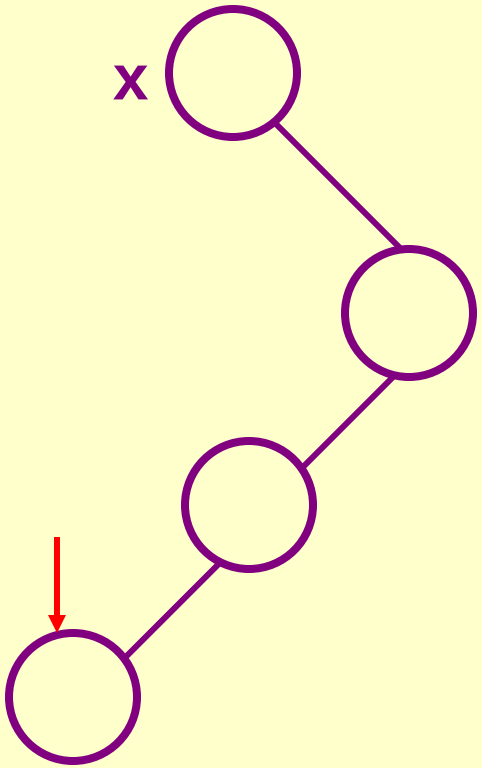
(2) The inorder successor of the last node in inorder is the head node.

5.5.2 Inorder Traversal of a Threaded Binary Tree

Observe:

(1) If x→rightThread==true, the inorder successor of x is x→rightChild;

(2) If x→rightThread==false, the inorder successor of x is obtained by following a path of **leftChild** from the right child of x until a node with leftThread==true is reached.



Thus we have:

```
T* ThreadedInorderIterator::Next()
{ // Return the inorder successor of currentNode in a threaded
  //  binary tree
    ThreadedNode<T>* temp=currentNode→rightChild;
    if (! currentNode→rightThread)
        while (!temp→leftThread) temp=temp→leftChild;
    currentNode=temp;
    if (currentNode==root) return 0; //no next
    else return &currentNode→data;
}
```

Note that when currentNode == root, Next() return the 1st node of inorder, thus we can use the following function to do an inorder travesal of a threaded binary tree:

```
template <class T>
void ThreadedTree<T>::Inorder()
{
   ThreadedInorderiterator ti;
   for (T* p = ti.Next(); p; p = ti.Next())
   Visit(*p);
}
```

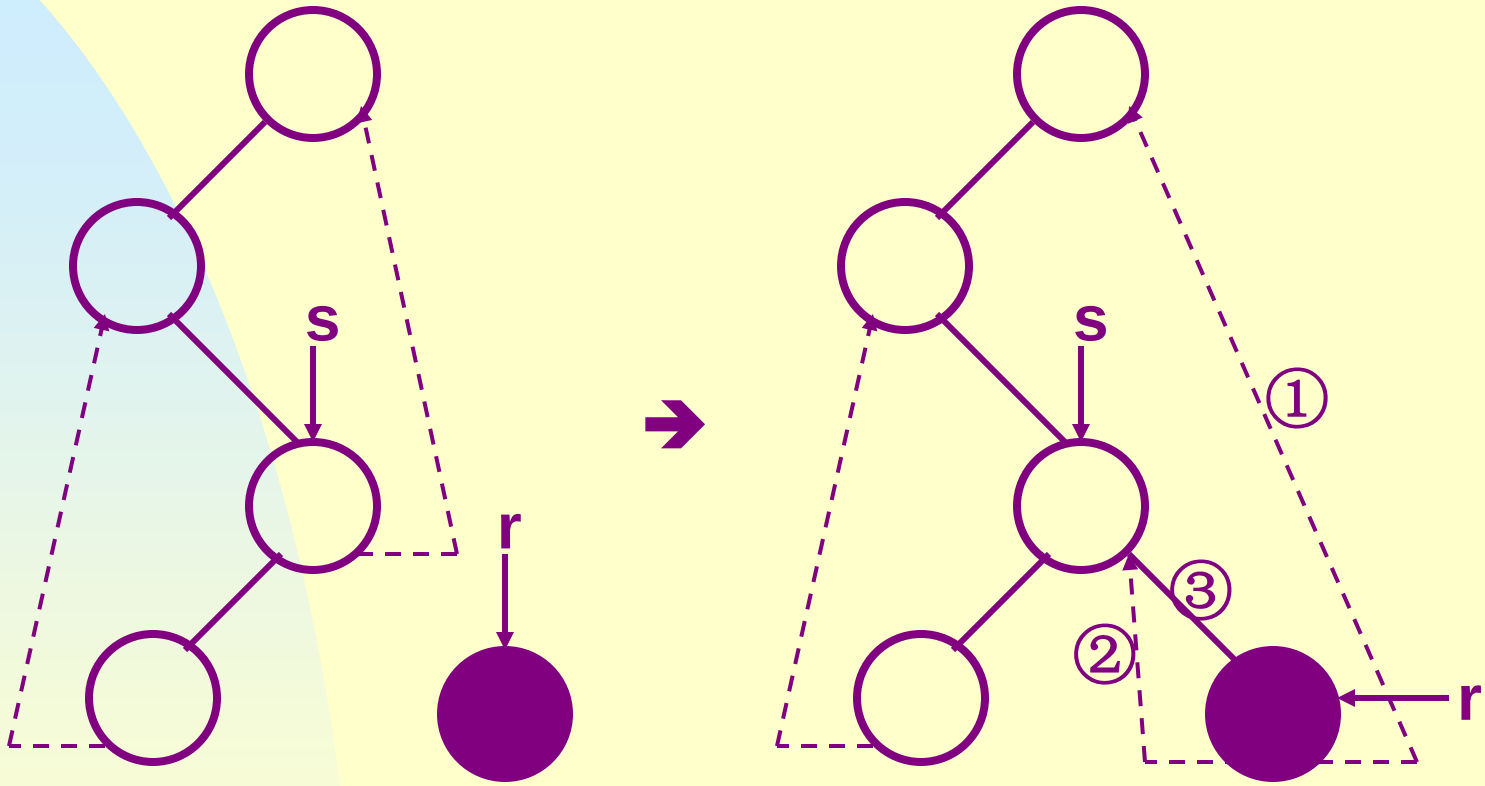
Since each node is visited at most 2 times, the computing time is readily seen to be O(n) for n nodes tree, and the traversal has been done without using a stack.

5.5.3 Inserting a Node into a Threaded Binary Tree

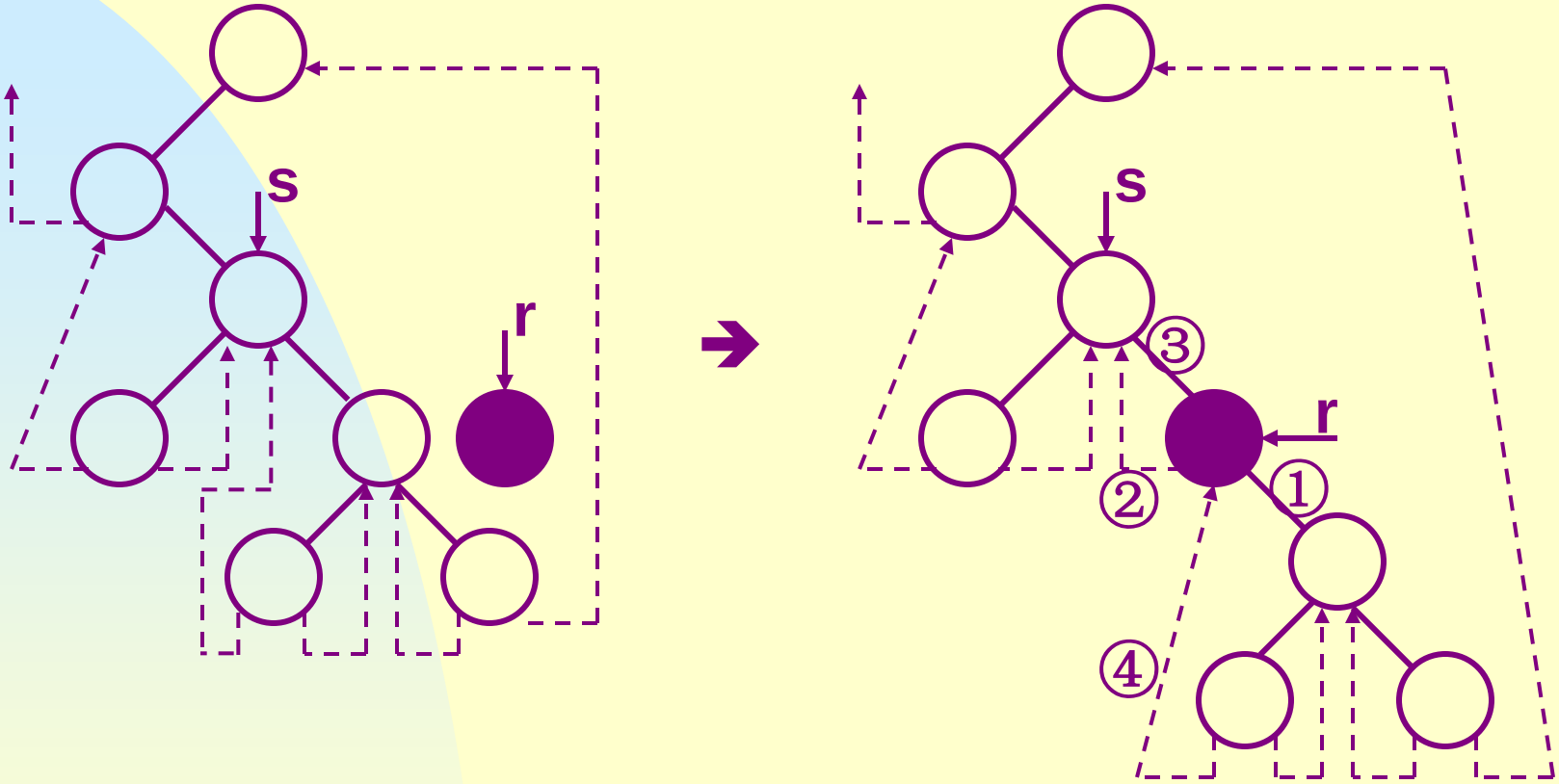
Insertion into a threaded tree provides the function for growing threaded tree.

We shall study only the case of inserting r as the right child of s. The left child case is similar.

(1) If s→rightThread==true, as:



(2) If s→rightThread==false, as:



In both (1) and (2), actions ①,②, ③ are the same, ④ is special for (2).

```
template <class T>
void  ThreadedTree<T>::InsertRight(ThreadedNode<T>* s,
                                   ThreadedNode<T>* r)
{ // insert r as the right child of s
    r→rightChild=s→rightChild;          // ①
    r→rightThread=s→rightThread;        // ① note s!=t.root,
    r→leftChild=s;                      // ②
    r→leftThread=true;                  // ②
    s→rightChild=r;                     // ③
    s→rightThread=false;                // ③
    if (! r→rightThread) {     // case (2)
       ThreadedNode<T>* temp=InorderSucc(r);  // ④
       temp→leftChild=r;                      // ④
    }
}
```

Exercises: P277-1, P278-4

5.6 Heaps

5.6.1 Priority Queues

In a priority queue, the element to be deleted is the one with highest (or lowest) priority.

Assume type T is defined so that operators <, >, etc. compare element priorities.

ADT 5.2 MaxPQ

```
template <class T>
class MaxPQ {
public:
    virtual ~MaxPQ { }
      // virtual destructor
    virtual bool IsEmpty() const = 0;
      // return true iff the priority queue is empty
    virtual const T& Top() const = 0;
      // return reference to the max element
    virtual void Push(const T&) = 0;
      // add an element to the priority queue
    virtual void Pop() = 0;
      // delete the max element
};
```

The simplest way to represent a priority queue is as an unordered linear list:

- **IsEmpty and Push in O(1)**
- **Top and Pop in O(n).**

By using a max heap:

- **IsEmpty and Top in O(1)**
- **Push and Pop in O(log n).**

5.6.2 Definition of a Max Heap

Definition:

A max (min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any).

A max (min) heap is a complete binary tree that is also a max (min) tree.

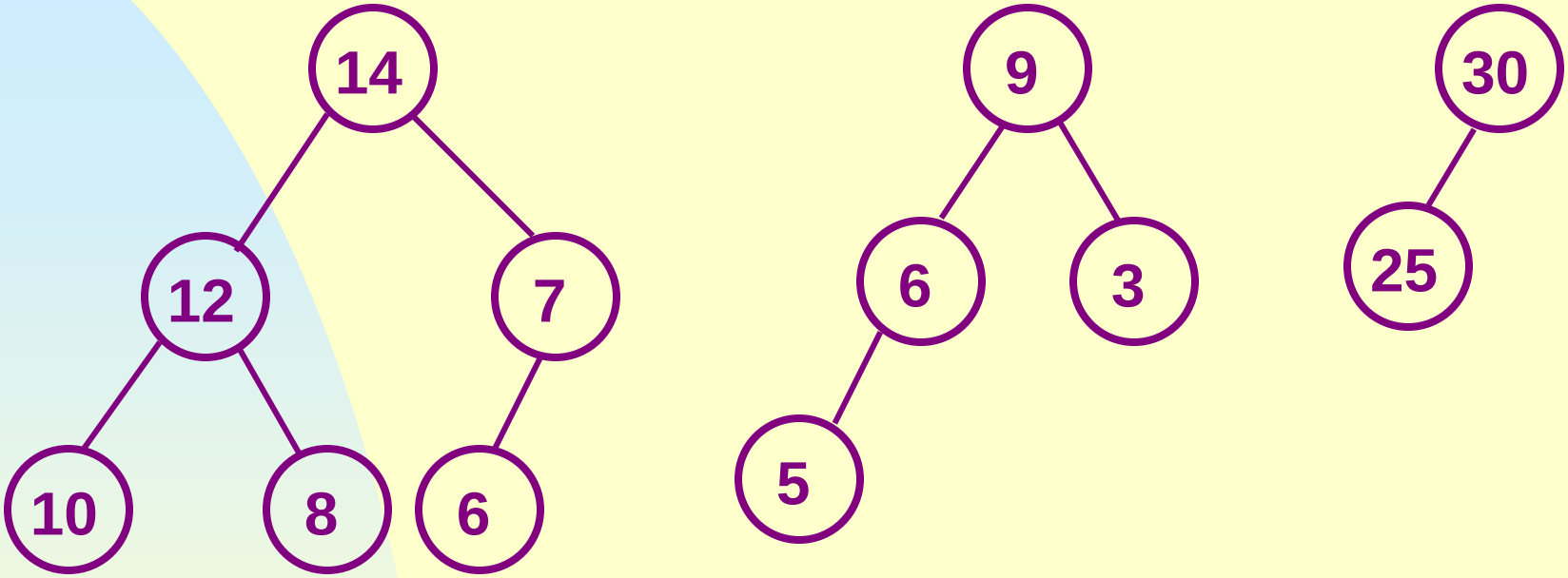


Fig. 5.24 Max heaps

Since max heap is a complete binary tree, we use array heap to represent it.

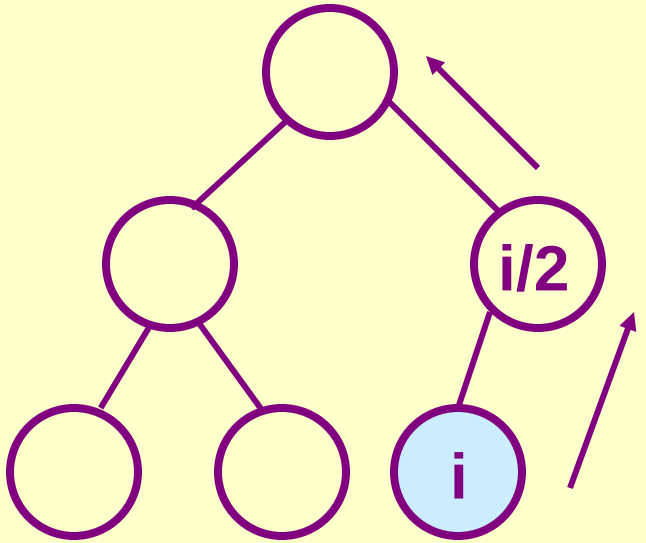
Thus we have the template MaxHeap class, which derives from MaxPQ<T>, as in the next slide:

```
template <class T>
class MaxHeap: public MaxPQ <T>
{
public:
    MaxHeap (int theCapacity=10);
    bool IsEmpty () { return heapSize==0;}
    const T& Top() const;
    void Push(const T&);
    void Pop();
private:
   T* heap;              // element array
   int heapSize;         // number of elements in heap
   int capacity;         // size of the array heap
};
```

```
template <class T>
MaxHeap<T>::MaxHeap (int theCapacity=10)
{ //constructor
    if (theCapacity < 1) throw “Capacity must be >= 1”;
    capacity = theCapacity;
    heapSize = 0;
    heap = new T[capacity+1]; //heap[0] not used
}

template <class T>
Inline T& MaxHeap<T>::Top()
{
    if (IsEmpty()) throw “The heap is empty”;
    return heap[1];
}
```

5.6.3 Insertion into a Max Heap

We use a **bubbling up** process to insert an element, begin at the node i=heapSize+1, and compare its key with that of its parent. If its key is larger, put its parent into node i , all the way up until it is no larger or reach the root.



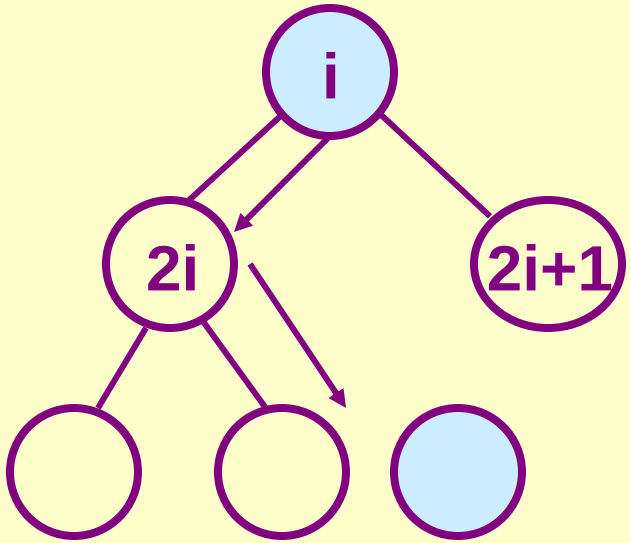
```
template <class T>
void MaxHeap<Type>::Push(const T& e)
{ // insert e into the max heap
    if (heapSize == capacity)  { // double the capacity
        ChangeSize1D(heap, capacity+1, 2*capacity+1);
        capacity *= 2;
    }
    int currentNode = ++heapSize;
    while (currentNode != 1 && heap[currentNode/2] < e)
    {  // bubble up
        heap[currentNode] = heap[currentNode/2];
        currentNode /=2;
    }
    heap[currentNode] = e;
}
```

Analysis of Push:

A complete binary tree with n nodes has a height $\lceil \log_2(n+1) \rceil$, the while loop is iterated O(log n) times, each takes O(1) time, so the complexity is O(log n).

5.6.4 Deletion from a Max heap

The max element is to be deleted from node 1, we assume the element in node heapSize to be in node 1, and heapSize--. Then we compare its key with that of its larger child. All the way down until it is no smaller or reach the leaf---trickle down.



```
template <class T>
void MaxHeap<T>::Pop()
{ // delete the max element.
   if (IsEmpty()) throw “Heap is empty. Cannot delete.”;
   heap[1].~T(); // delete the max

   // remove the last element from heap
   T lastE = heap[heapSize--];

   // trickle down
   int currentNode = 1;  // root
   int child = 2;      // left child of currentNode
```

```
while (child <= heapSize)
{
   // set child to the larger child of currentNode
   if (child<heapSize && heap[child]<heap[child+1]) child++;

   // can we put lastE in currentNode?
   if (lastE>=heap[child]) break;  // yes

   // no
   heap[currentNode]=heap[child]; // move child up
   currentNode=child; child*=2; // move down a level
 }
heap[currentNode]=lastE;
}
```

Analysis of Pop:

The height of a heap with n nodes is $\lceil \log_2(n+1) \rceil$, the while loop is iterated O(log n) times, each takes O(1) time, so the complexity is O(log n).

Exercises: P287-2, 3

5.7 Binary Search Trees

5.7.1 Definition

When arbitrary elements are to be searched or deleted, heap is not suitable.

A **dictionary** is a collection of pairs, each pair has a key and an associated element.

Although natural dictionaries may have several pairs with the same key, we assume here that **no two pairs have the same key.**

The specification of a dictionary is given as **ADT 5.3.**

ADT 5.3

```
template <class K, class E>
class Dictionary {
public:
   virtual bool IsEmpty () const = 0;
      // return true iff the dictionary is empty
   virtual pair<K,E>* Get(const K&) const = 0;
      // return pointer to the pair with specified key;
      // return 0 if no such pair
   virtual void Insert(const pair<K,E>&) = 0;
      // insert the given pair; if key is a duplicate
      // update associated element
   virtual void Delete(const K&) = 0;
      // delete pair with specified key
};
```

The pair can be defined as:

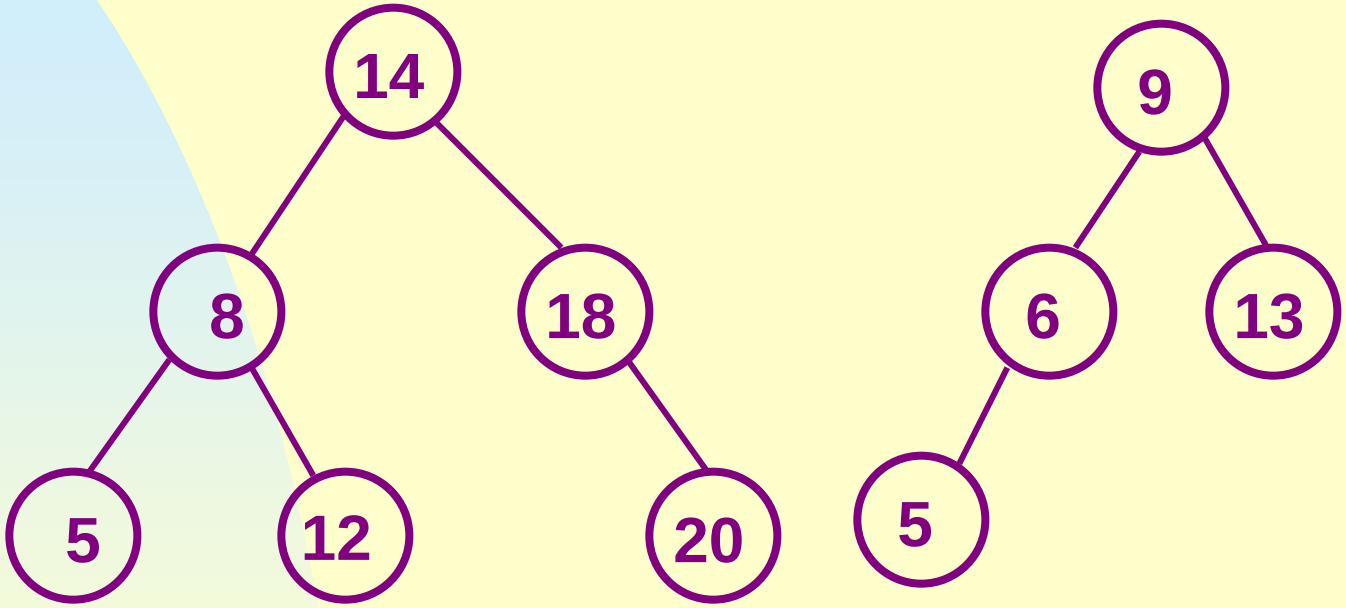
```
template <class K, class E>
struct pair
{
    K first;
    E second;
};
```

A binary search tree has a better performance when the functions to be performed are search, insert and delete.

Definition: A binary search tree, if not empty, satisfies the following properties:

(1) The root has a key.
(2) The keys (if any) in the left subtree are smaller than that in the root.
(3) The keys (if any) in the right subtree are larger than that in the root.
(4) The left and right subtrees are also binary search trees.

Note these properties imply that the keys must be distinct.



Binary search trees

5.7.2 Searching a Binary Search Tree

According to its properties, it is easy to search a binary search tree. Suppose search for an element with key k:

If k==the key in root, success;

If x<the key in root, search the left subtree;

If x>the key in root, search the right subtree.

Assume class BST **derives from the class** Tree<pair<K,E>>**.**

```
template <class K, class E> // Driver
pair<K,E>* BST<K,E>::Get(const K& k)
{ // Search *this for a pair with key k.
    return Get(root, k);
}

template <class K, class E> // Workhorse
pair<K,E>* BST<K,E>::Get(treeNode<pair<K,E>>* p,
                                                const K& k)
{
    if (!p) return 0;
    if (k < p→data.first) return Get(p→leftChild, k);
    if (k > p→data.first) return Get(p→rightChild, k);
    return &p→data;
}
```

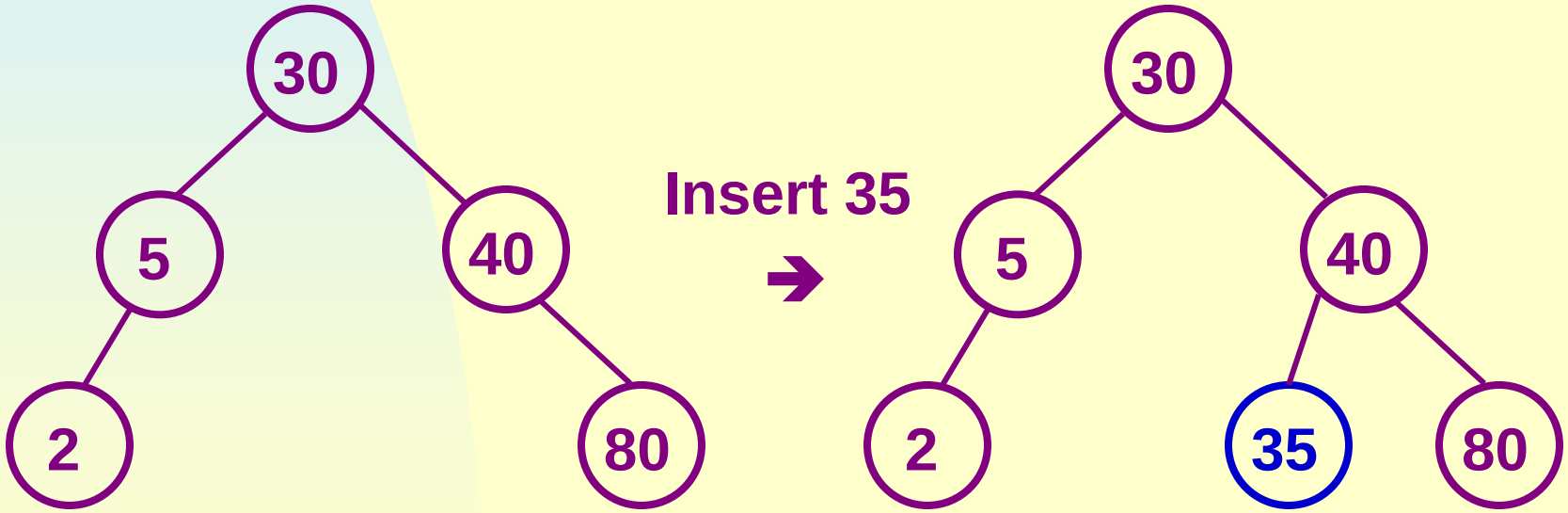
The recursive version can be easily changed into an iterative one as in the following:

```
template <class K, class E>  // Iterative version
pair<K,E>* BST<K,E>::Get(const K& k)
{
    TreeNode<pair<K,E>>* currentNode = root;
    while (currentNode)
      if (k < currentNode→data.first)
          currentNode = currentNode→leftChild;
      else if (k > currentNode→data.first)
          currentNode = currentNode→rightChild;
      else return &currentNode→data;

    // no matching pairs
    return 0;
}
```

As can be seen, a binary search tree of height h can be search by key in O(h) time.

5.7.3 Insertion into a Binary Search Tree

To insert a new element, search is carried out, if unsuccessful, then the element is inserted at the point the search terminated.



When the dictionary already contains a pair with key k, we simply update the element associated with this key to e.

```
template <class K, class E>
void BST<K,E>::Insert(const pair<K,E>& thePair)
{ // Insert thePair into the binary search tree
    // search for thePair.first, pp is parent of p
    TreeNode<pair<K,E>> *p=root, *pp=0;
    while (p) {
      pp=p;
      if (thePair.first < p→data.first) p=p→leftChild;
      else if (thePair.first > p→data.first) p=p→rightChild;
      else // duplicate, update associated element
          {p→data.second=thePair.second;return;}
    }
```

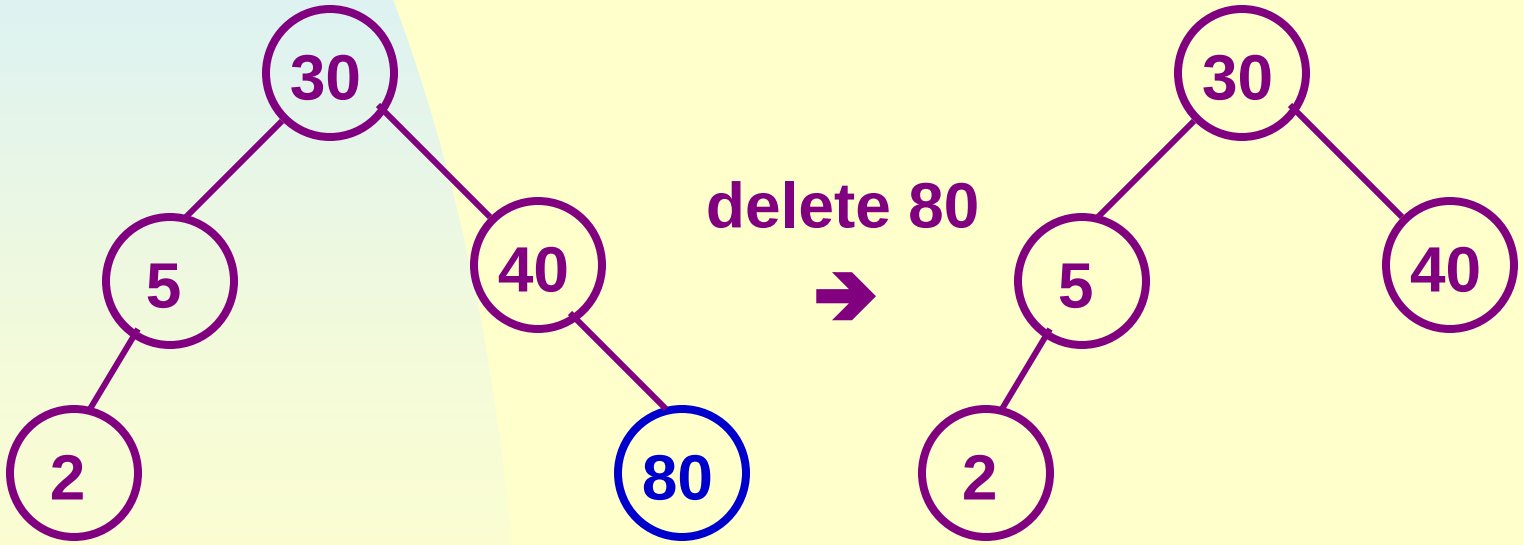
```
    // perform insertion
    p=new TreeNode<pair<K,E>>(thePair,0,0);
    if (root) // tree not empty
       if (thePair.first < pp→data.first) pp→leftChild=p;
          else pp→rightChild=p;
    else root=p;
}
```

The time is O(h).

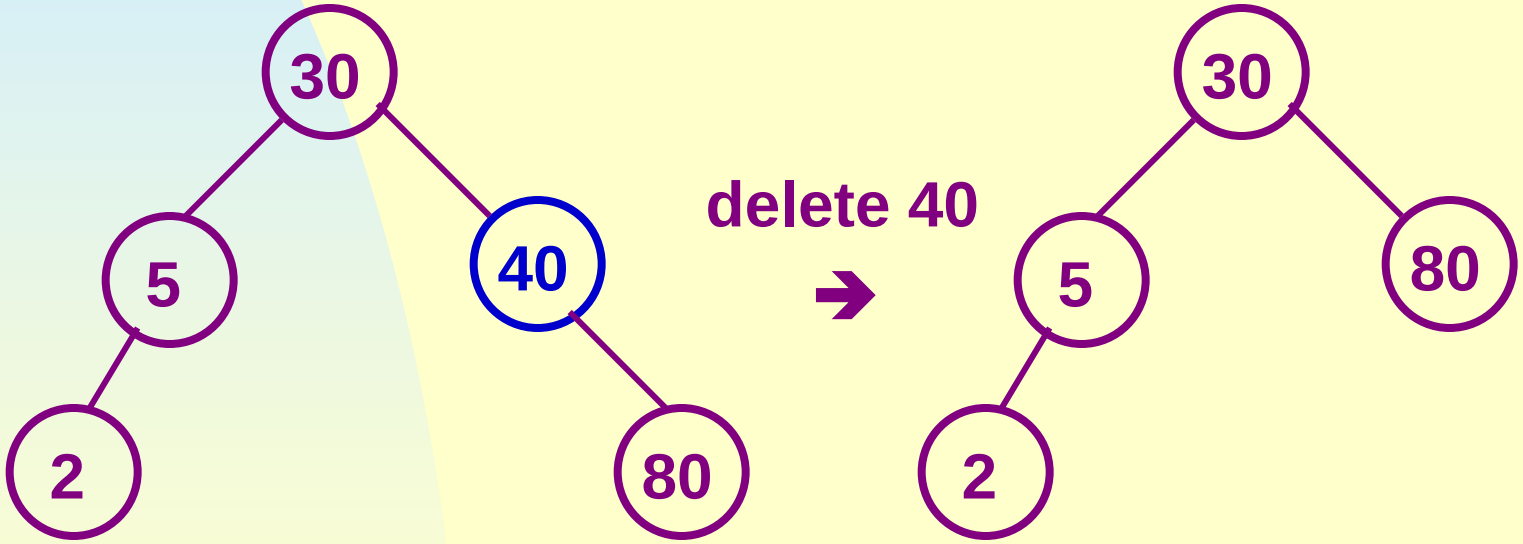
5.7.4 Deletion from a Binary Search Tree

3 cases:

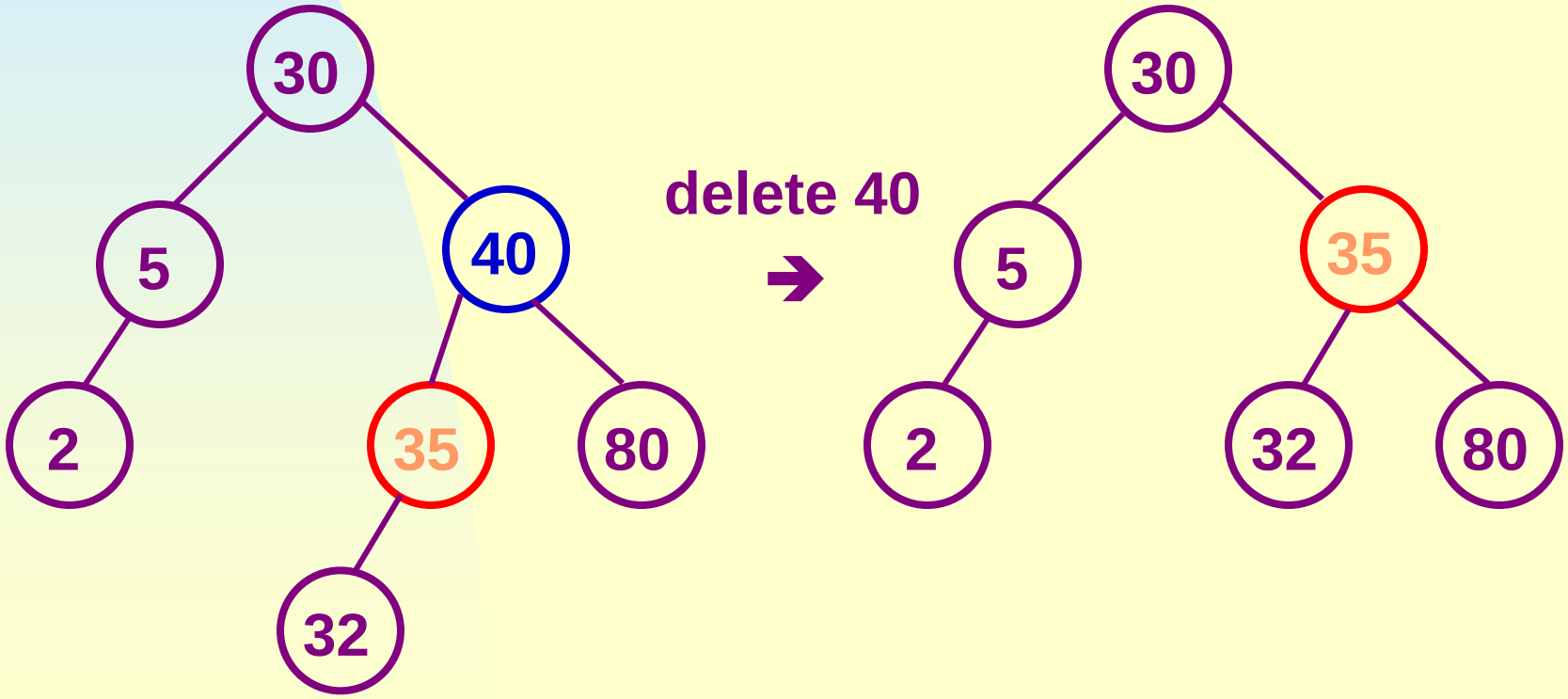
(1) deletion of a leaf: easy, set the corresponding child field of its parent to 0 and dispose the node.



(2) deletion of a nonleaf node with one child: easy, the child of the node takes the place of it and the node disposed.



(3) deletion of a nonleaf node with two children: the element is replaced by either the largest in its leftChild or the smallest in its rightChild. Then delete that node, which has at most one child, in the subtree.



Exercises: P296-1,2

5.8 Selection Trees

5.8.1 Introduction

To merge k ordered sequences, called **runs**, we need to find the smallest from k possibilities, output it, and replace it with the next record in the corresponding run, then find the next smallest, and so on.

The most direct way is to make k-1 comparisons. For k>2, if we make use of the knowledge obtained from the last comparisons, the number of comparisons to find the next can be reduced.

Selection trees including winner tree and loser tree are suitable for this.

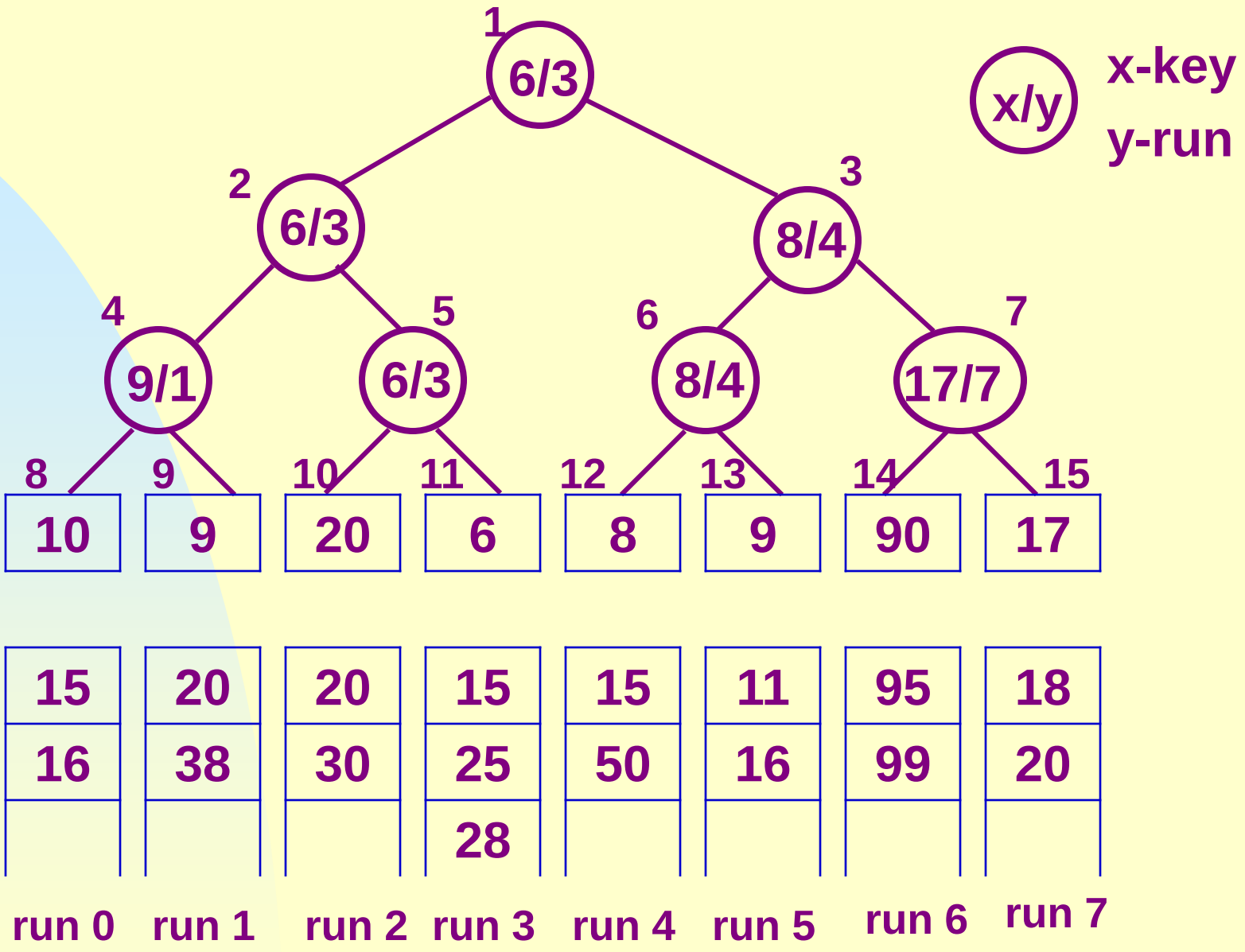
5.8.2 Winner trees

A winner tree is a complete binary tree in which each node represents the smaller (winner) of its children. The root represents the smallest.

The construction of a winner tree may be compared to the playing of a tournament.

Leaf node---the first record in the corresponding run.

Nonleaf node---contains an index to the winner of a tournament.

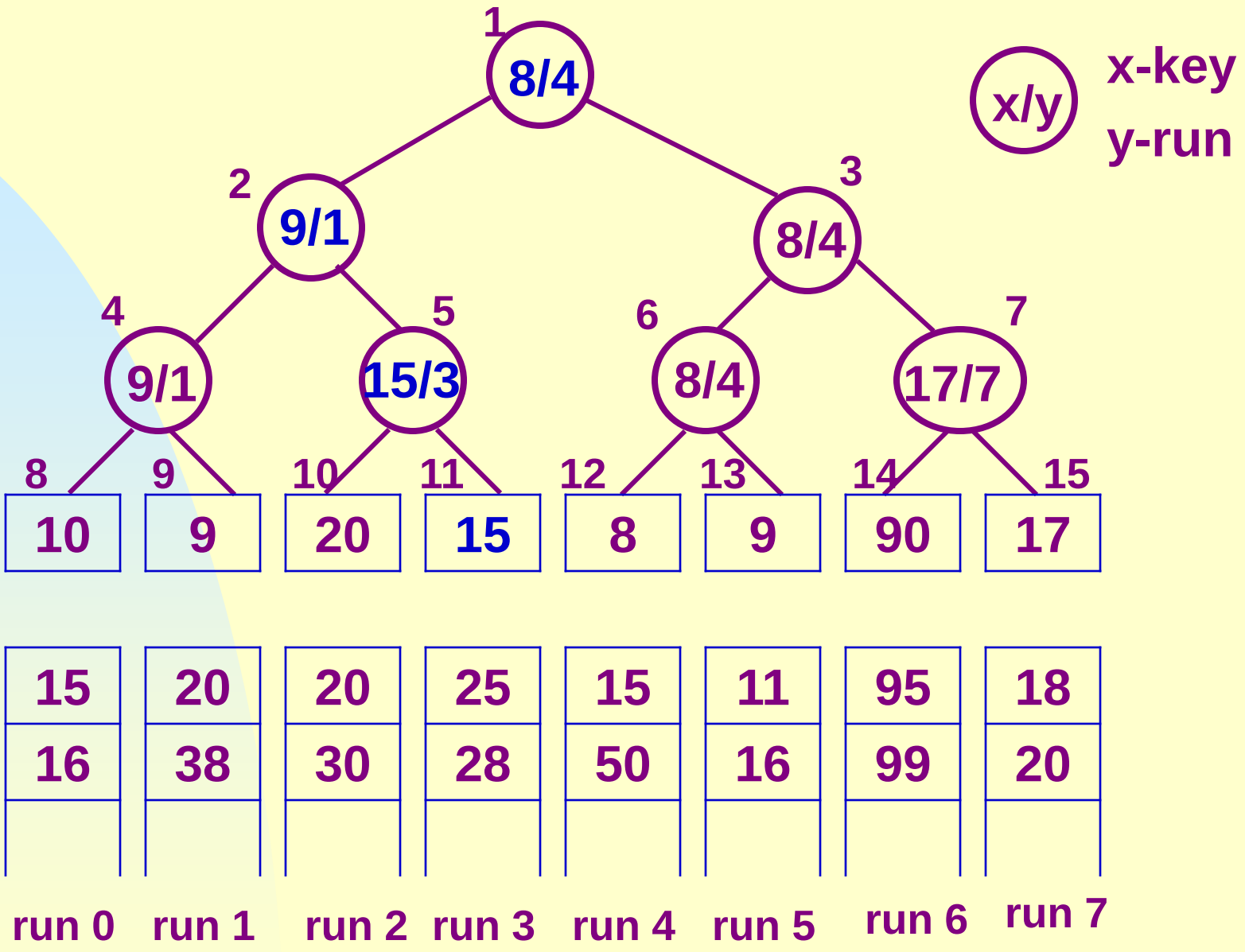

1
6/3
2
6/3
3
8/4
4
9/1
5
6/3
6
8/4
7
17/7
8
9
10
11
12
13
14
15
10
9
20
6
8
9
90
17
15
16
20
38
20
30
20
15
25
28
15
50
11
16
95
99
18
20
run 0
run 1
run 2
run 3
run 4
run 5
run 6
run 7
x/y
x-key
y-run

Actually, leaf nodes can be record buffer[0] to buffer[k-1]. The node number of Buffer[i] in the tree is k+i.

Now record pointed by the root (key==6) is output, buffer[3] is empty, the next record from run 3 (key ==15) is input to buffer[3].

To reconstruct the tree, tournament has to be played along the path from node 11 to the root.

As in the next slide:


1
8/4
2
9/1
3
8/4
4
9/1
5
15/3
6
8/4
7
17/7
8
9
10
11
12
13
14
15
10
9
20
15
8
9
90
17
x/y
x-key
y-run
15
16
20
38
20
30
25
28
15
50
11
16
95
99
18
20
run 0
run 1
run 2
run 3
run 4
run 5
run 6
run 7

The tournament is played between sibling nodes and the result put into the parent node. Lemma 5.4 may be used to compute the address of sibling and parent nodes efficiently.

Time of reconstruction: $O(\log_2 k)$

Time of setting up the tree: $O(k)$

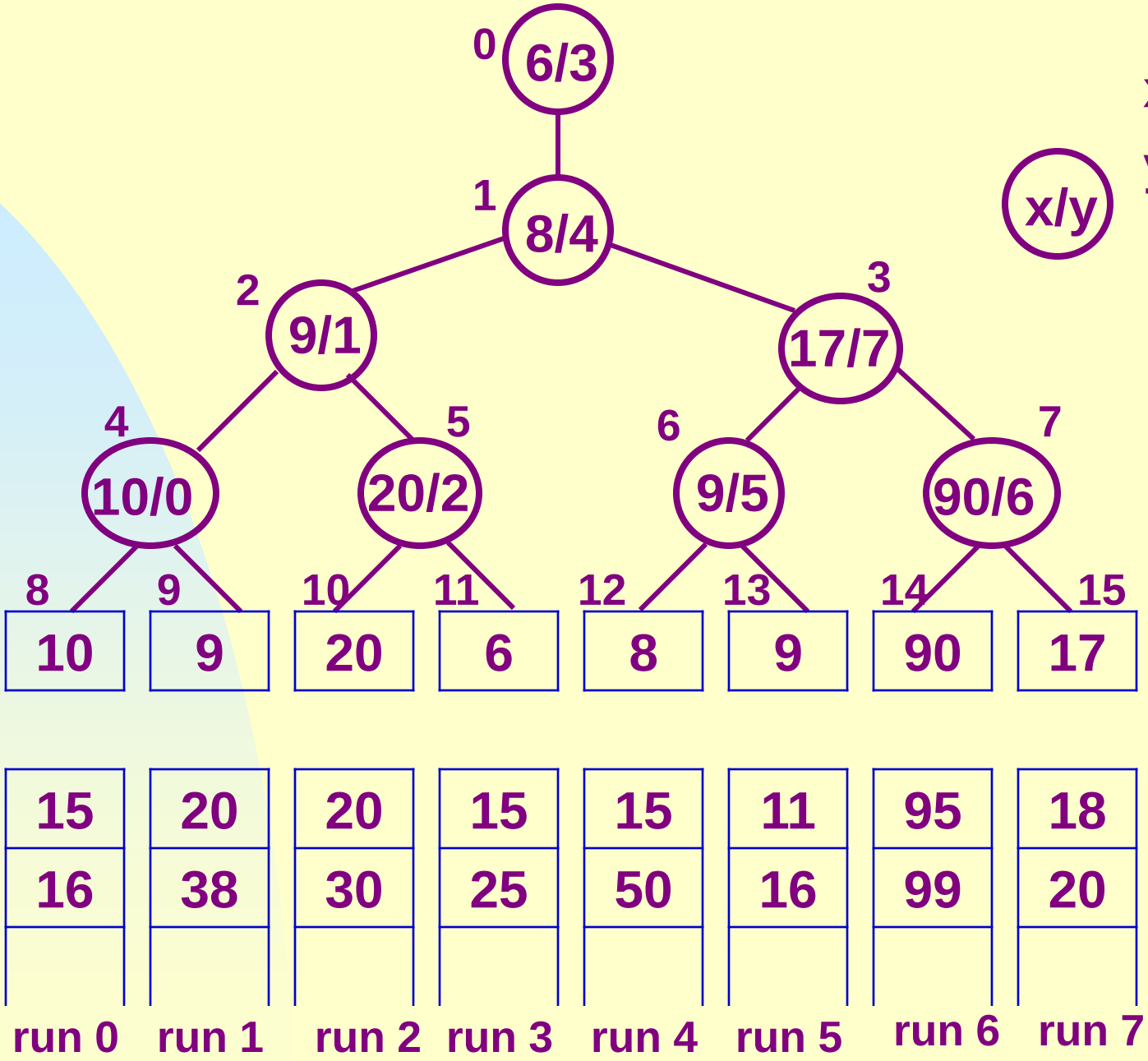
5.8.3 Loser Trees

To simplify the restructuring we can use a loser tree.

A selection tree in which each nonleaf node retains a pointer to the loser is called a loser tree.

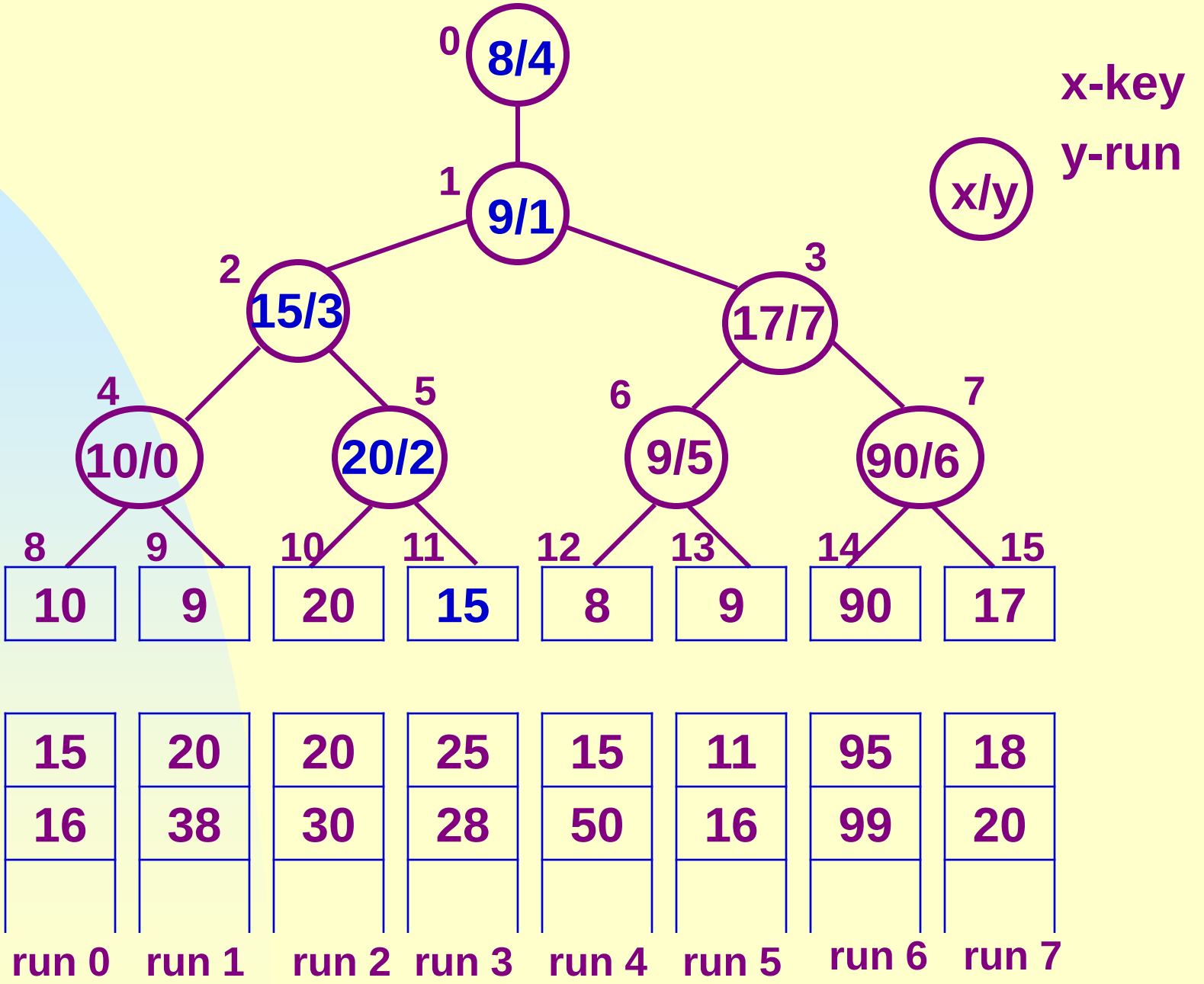
Actually, the loser in node p is the loser of the winners of the children of p.

The next slide shows a loser tree corresponding the previous winner tree.


0
6/3
1
8/4
2
9/1
3
17/7
4
10/0
5
20/2
6
9/5
7
90/6
8
9
10
11
12
13
14
15
10
9
20
6
8
9
90
17
15
16
20
38
20
30
15
25
15
50
11
16
95
99
18
20
run 0
run 1
run 2
run 3
run 4
run 5
run 6
run 7
x-key
y-run
x/y

Node 0 represents the overall winner.

After outputting record[3] (node 11, key 6), the tree is restructured by reading next record from run 3 and playing tournament along the path from node 11 to node 1, as in the next slide:


0
8/4
1
9/1
2
15/3
3
17/7
4
10/0
5
20/2
6
9/5
7
90/6
8
9
10
11
12
13
14
15
10
9
20
15
8
9
90
17
15
16
20
38
20
30
25
28
15
50
11
16
95
99
18
20
run 0
run 1
run 2
run 3
run 4
run 5
run 6
run 7
x-key
y-run
x/y

The records with which these tournaments are to be played are readily available from the parent nodes.

Exercises: P301-1,4

5.9 Forests

Definition: A forest is a set of n≥0 disjoint trees.

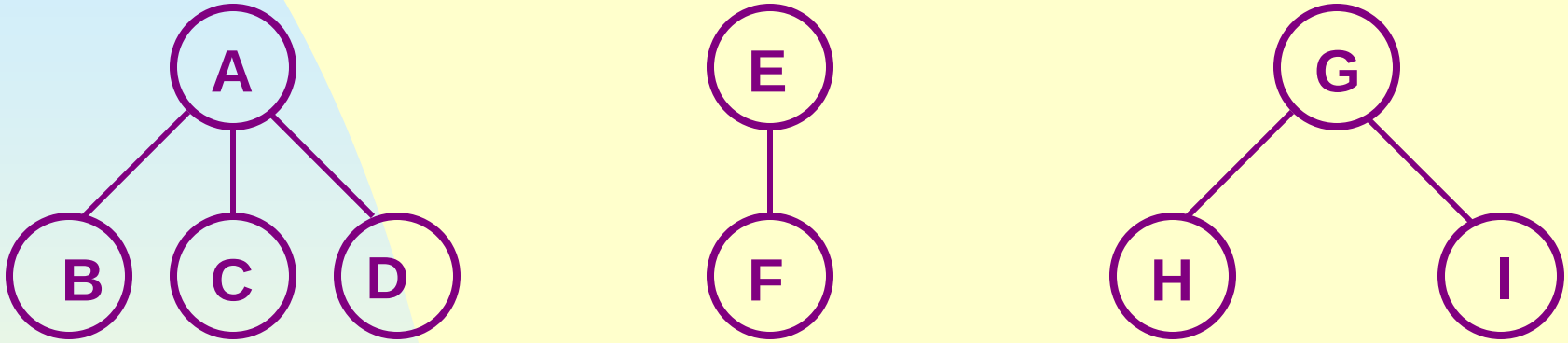


Fig. 5.34: Three-tree forest

5.9.1 Transforming a Forest into a Binary Tree

Definition: If $T_1,\ldots, T_n$ is a forest of trees, then the binary tree corresponding to it , denoted by $B(T_1,\ldots, T_n)$,

(1) is empty if n=0

(2) has root equal to root(T_1); has left subtree equal to $B(T_{11},\ldots, T_{1m})$, where $T_{11},\ldots, T_{1m}$ are the subtrees of root(T_1); and has right subtree $B(T_2,\ldots, T_n)$.

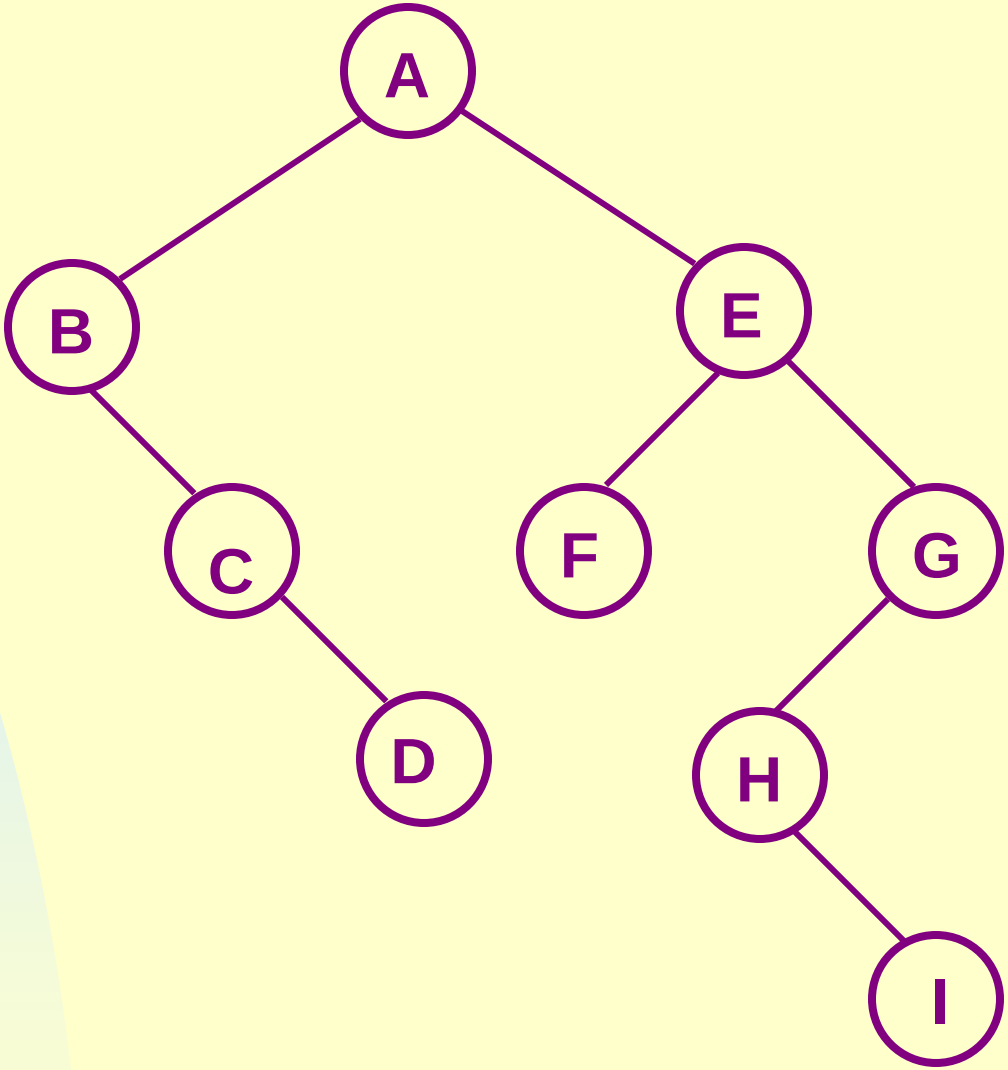


Fig. 5.35: Binary tree representation of forest of Fig.5.34

5.9.2 Forest Traversals

Let T be the corresponding binary tree of a forest F.

Visiting the nodes of F in forest preorder is defined as:

(1) If F is empty then return.

(2) Visit the root of the first tree of F.

(3) Traverse the subtrees of the first tree in forest preorder.

(4) Traverse the remaining trees of F in forest preorder.

Visiting the nodes of F in forest inorder is defined as:

(1) If F is empty then return.

(2) Traverse the subtrees of the first tree in forest inorder.

(3) Visit the root of the first tree of F.

(4) Traverse the remaining trees of F in forest inorder.

Visiting the nodes of F in forest postorder is defined as:

(1) If F is empty then return.

(2) Traverse the subtrees of the first tree in forest postorder.

(3) Traverse the remaining trees of F in forest postorder.

(4) Visit the root of the first tree of F.

In level-order traversal of F, nodes are visited by level, beginning with the roots of each trees in F. Within each level, from left to right.

Exercises: P304-3.

5.10 Set Presentation

5.10.1 Introduction

Assume:

- **Elements of the sets are the numbers 0, 1, 2, …, n-1 (might be thought as indices).**
- **For any two sets S_i, S_j, $i \neq j$, $S_i \cap S_j = \varnothing$.**

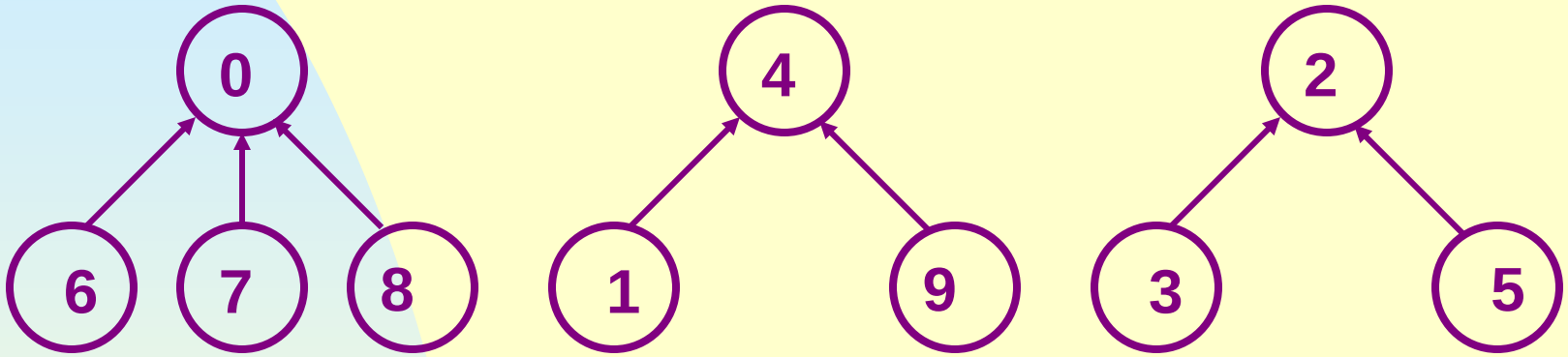
Operations:

(1) Disjoint set union $S_i \cup S_j$.

(2) Find(i)---find the set containing i.

The sets can be represented by trees:

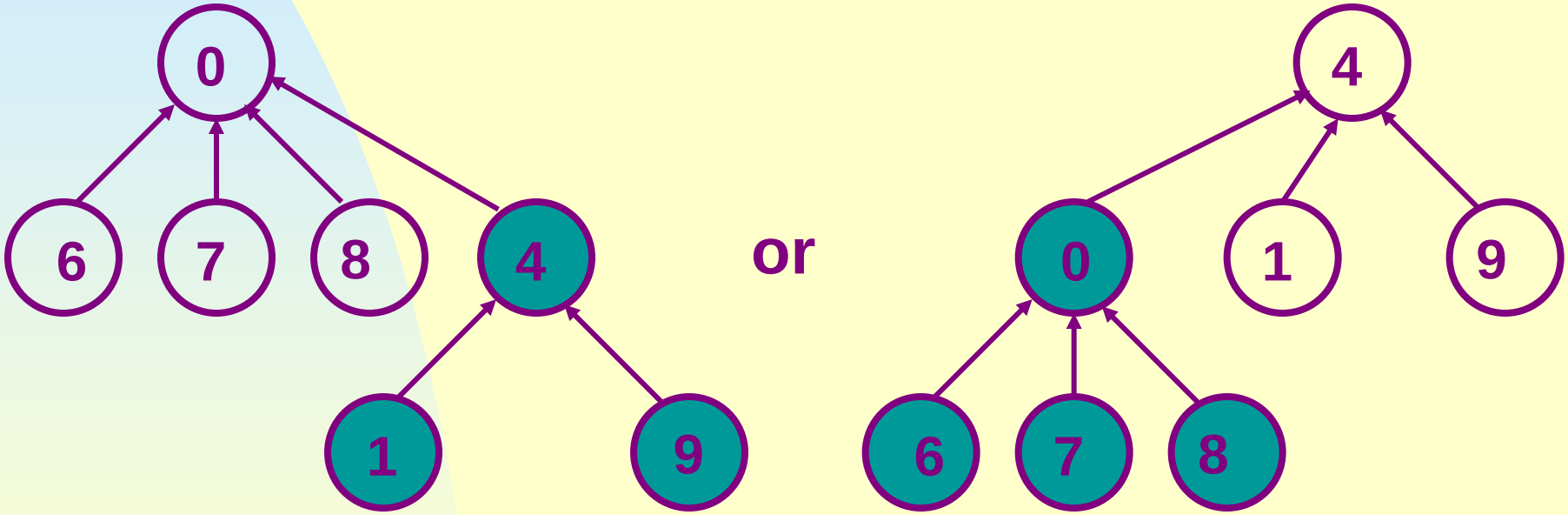
S_1={0,6,7,8} S_2={1,4,9} S_3={2,3,5}



The nodes are linked from the children to the parent.

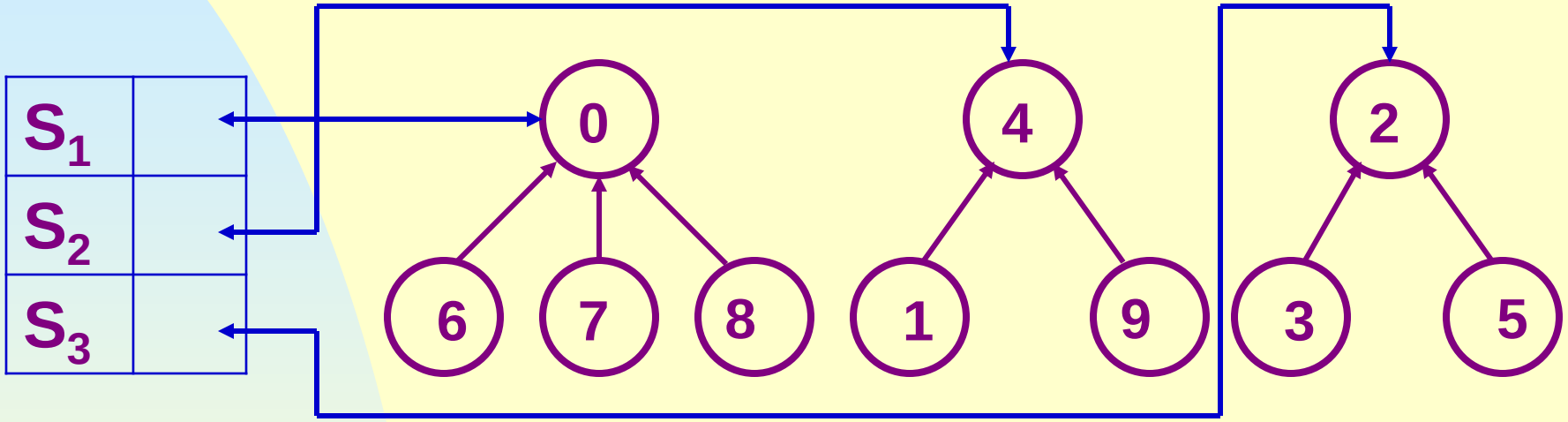
5.10.2 Union and Find Operations

To do $S_1 \cup S_2$, simply make one of the trees a subtree of the other:



$S_1 \cup S_2$

To find the root of a set from its name, we can keep a pointer to the root with each set name.



Here we ignore the actual set names, just identify sets by their roots. The transition to set names is easy.

Since the set elements are numbered 0,1,…,n-1, we represent the tree nodes using an array parent[n]**. The** parent[i] **contains the parent pointer of node i (element i as well).**

The root node has a parent of –1.

The following is the array representation of S_1, S_2, and S_3:

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

```
class Sets {
public:
    // Set operations
    …
private:
    int  *parent;
    int  n; // number of set elements
};

Sets::Sets (int numberOfElements)
{
    if (numberOfElements < 2) throw “Must have at least 2
elements.”;
    n=numberOfElements;
    parent=new int[n];
    fill(parent, parent+n, -1);
}
```

Now we have simple algorithm for union and find.

```
void Sets::SimpleUnion (int i, int j)
{ // Replace the disjoint sets with roots i and j, i!=j with their
  // union
    parent[i] = j;
}

int Sets::SimpleFind (int i)
{ //find the root of the tree containing element i.
    while (parent[i]>=0) i=parent[i];
    return i;
}
```

Analysis of SimpleUnion and SimpleFind:

Given $S_i=\{i\}$, **0 ≤ i < n, do**

Union(0,1), Union(1,2), …, Union(n-2,n-1), Find(0), Find(1),…, Find(n-1) **get**

n-1 unions take O(n)

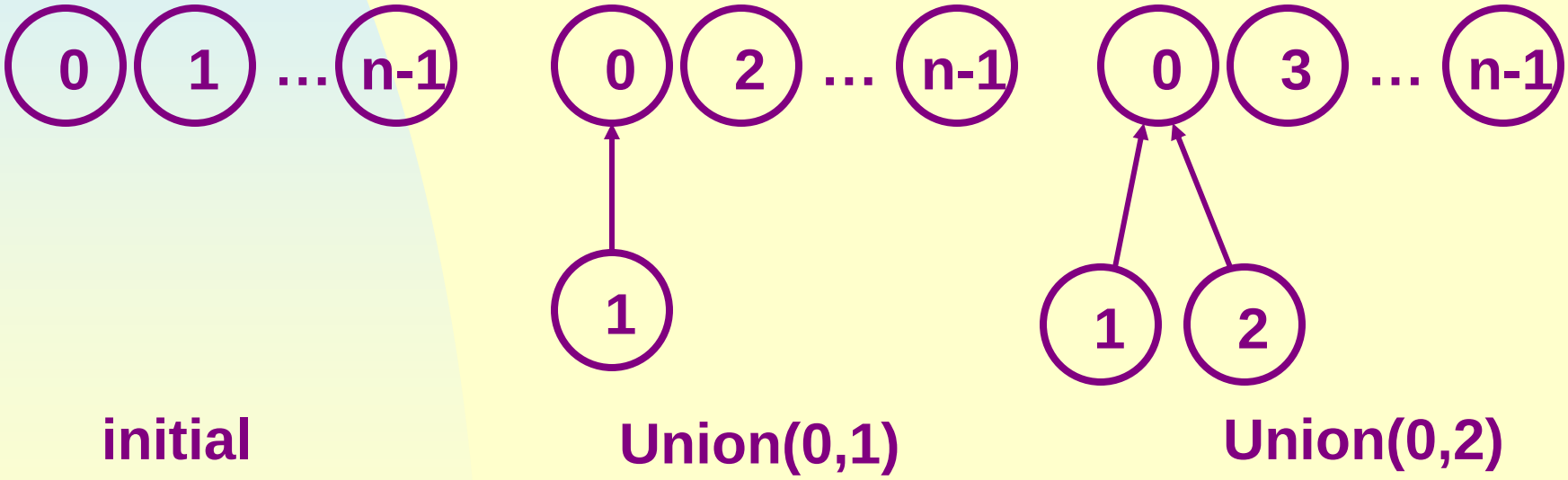
n finds take O($\sum_{i=1}^{n} i$)=O(n^2).

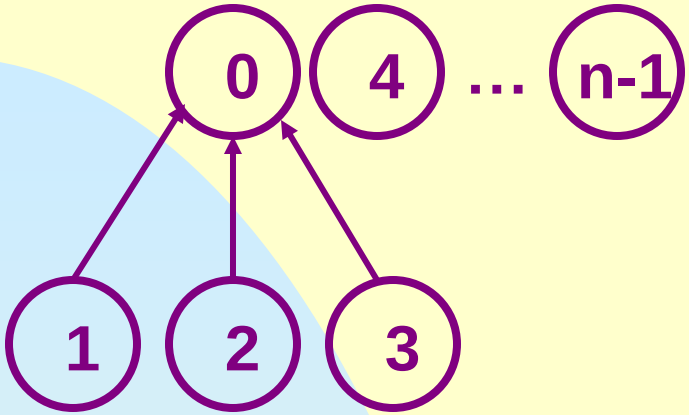
Performance not very good:

to improve by avoiding the creation of degenerate trees .

Definition [Weighting rule for Union(i,j)]: If the number of nodes in the tree with root i is less than that in the tree with root j, then make j the parent of i; otherwise make i the parent of j.

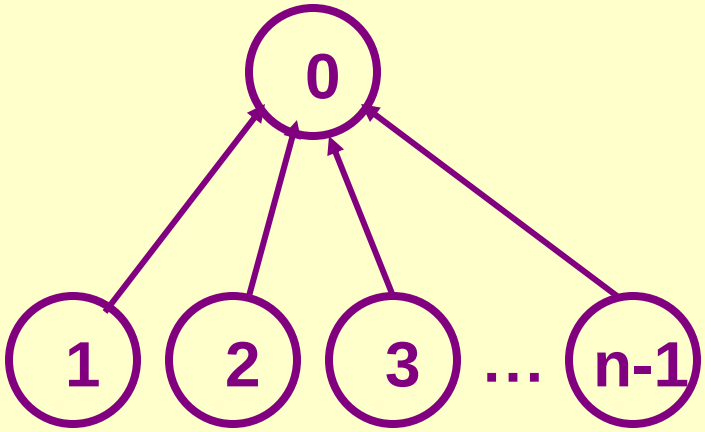
For instance:





Union(0,3)

… …



Union(0,n-1)

We need a count field to know how many nodes in each tree, and it can be maintained in the parent field of the root as a negative number.

```
void Sets::WeightedUnion (int i, int j)
{ // Union sets with roots i and j, i≠j, using the weighting rule
  // parent[i] = - count[i] and parent[j] = - count[j]
    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j]) {   // i has fewer nodes
        parent[i] = j;
        parent[j] = temp;
     }
    else {    // j has fewer nodes or
                // i and j have the same number of nodes
        parent[j] = i;
        parent[i] = temp;
    }
}
```

Analysis of WeightedUnion and SimpleFind:

The time of WeightedUnion is O(1).

The maximum time to perform a find is determined by:

Lemma 5.5: Assume we start with a forest of one node trees. Let T be a tree with m nodes created as a result of a sequence of weighted unions. The height of T is no more than $\lfloor \log_2 m \rfloor + 1$.

Proof by induction:

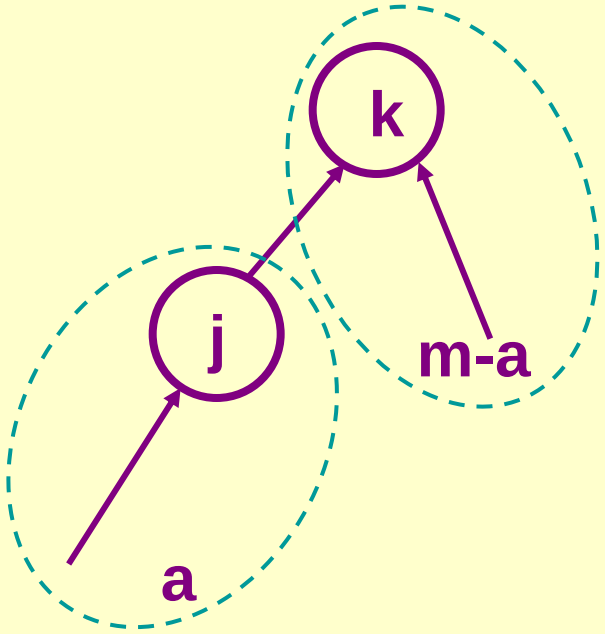
m = 1, it is true.

Assume it is true for all trees with i ≤ m-1 nodes.

For i = m, let T be a tree with m nodes created by WeightedUnion.

Consider the last union performed, Union(k, j).

Let a be the number of nodes in tree j and m-a that in tree k. without loss of generality, assume $1 \le a \le m/2$. Then the height of T is either the same as that of k or is 1 + that of j.



$m-a \ge m/2 \ge a$

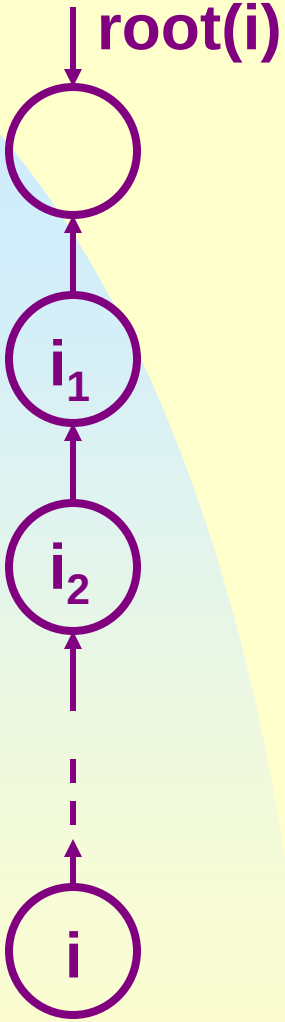
If the former is the case, the height of $T \le \lfloor \log_2 (m-a) \rfloor + 1 \le \lfloor \log_2 m \rfloor + 1$.

If the latter is the case, the height of $T \le \lfloor \log_2 a \rfloor + 2 \le \lfloor \log_2 m/2 \rfloor + 2 \le \lfloor \log_2 m \rfloor + 1$.

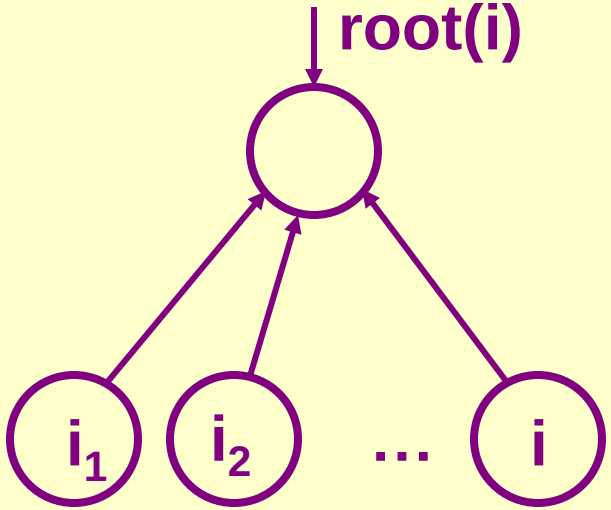
The time to process a find is at most O(log n) in a tree of n nodes. If an intermixed sequence of u-1 union and f find is to be done, the worst case time is O(u + f log u).

Further improvement in the find algorithm.

Definition [Collapsing rule] : If j is a node on the path from i to its root and parent[i] ≠ root(i), then set parent[j] to root(i).



➔



$j=i,\ldots, i_2, i_1$

```
int Sets::CollapsingFind (int i )
{ // Find the root of tree containing element i. Use the
  // collapsing rule to collapse all nodes from i to the root.
    for (int r = i; parent[r] >= 0; r = parent[r]); // find the root
    while ( i != r )  {
        int s = parent[i];
        parent[i] = r;
        i = s;
    }
    return r;
}
```

The worst-case complexity of processing a sequence of unions and finds using WeightedUnion and CollapsingFind is stated in Lemma5.6.

First, a very slow growing function

$$\alpha(p,q)=\min \{z\geq 1 \mid A(z, \lfloor p/q \rfloor)>\log_2 q\},\ p \geq q \geq 1$$

And Ackermann's function A(i,j) is defined as:

$$A(1, j) = 2^j \qquad \text{for } j \geq 1$$

$$A(i, 1) = A(i-1, 2) \qquad \text{for } i \geq 2$$

$$A(i, j) = A(i-1, A(i, j-1)) \qquad \text{for } i, j \geq 2$$

$A(1, 2) = 2^2$

$A(3,1) = A(2, 2) = A(1, A(2, 1)) = A(1, A(1, 2))$

$= 2^{2^2} = 16$

$A(4,1) = A(3, 2) = A(2, A(3, 1)) = A(2, 16)$

$= A(1, A(2, 15)) = 2^{A(2, 15)}$

$= 2^{2^{A(2, 14)}} = \ldots = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (16 twos) (very big !)

So for any practical p and q, $\alpha(p,q) \leq 4$.

Lemma 5.6 [Tarjan and Van Leeuwen]: Assume we start with a forest of trees, each having one node. Let T(f, u) be the maximum time required to process any intermixed sequence of f finds and u unions. Assume $u \geq n/2$, then

$$k_1(n+f\,\alpha(f+n,n)) \leq T(f, u) \leq k_2\,(n+f\,\alpha(f+n,n))$$

For some positive constants k_1 and k_2.

Note T(f, u) is not linear even though $\alpha(f+n,n)$ grows very slow.

The space requirements are one node for each element.

5.10.3 Application to Equivalent Classes

equivalence classes $\Leftrightarrow$ disjoint sets

Initially, parent[i] = -1, $0 \le i \le n-1$.

To process $i \equiv j$,

Let x = find(i), y = find(j) --- 2 finds

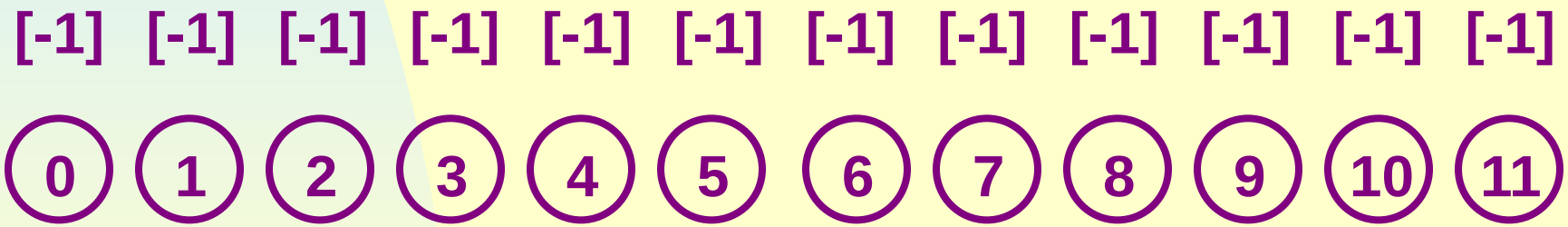
If $x \neq y$ then union(x, y) --- at most 1 union

Thus if we have n elements and m equivalence pairs, we needs 2m finds and min {n-1, m} unions. The total time is $O(n+2m\ \alpha(n+2m,n))$.

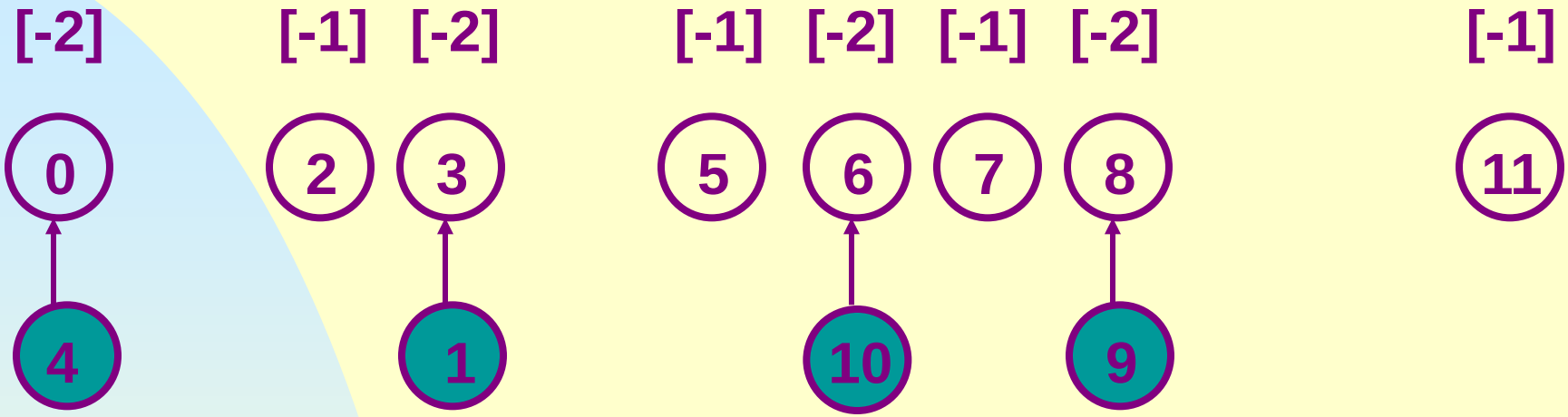
Example:

n = 12, process equivalence pairs:

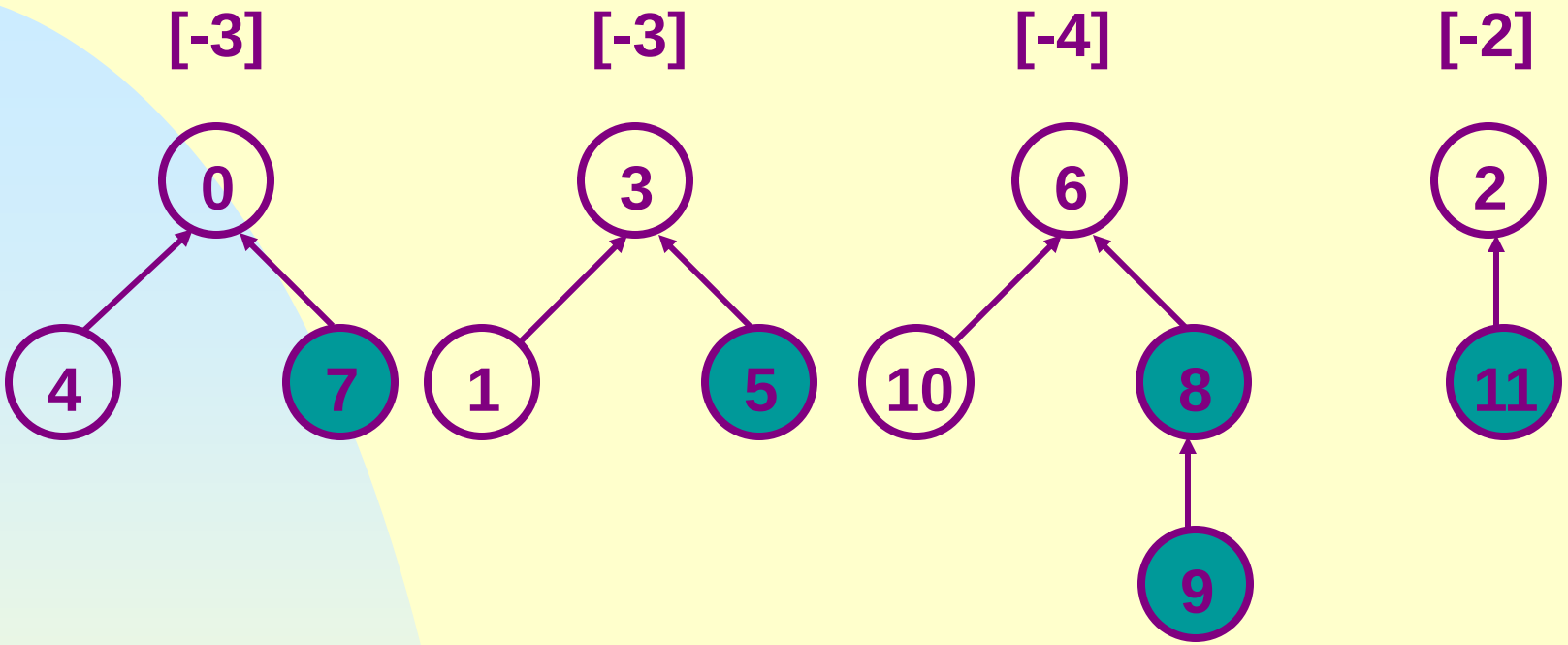
0 ≡ 4, 3 ≡ 1, 6 ≡10, 8 ≡ 9, 7≡ 4, 6 ≡ 8, 3 ≡ 5, 2 ≡ 11, 11 ≡ 0



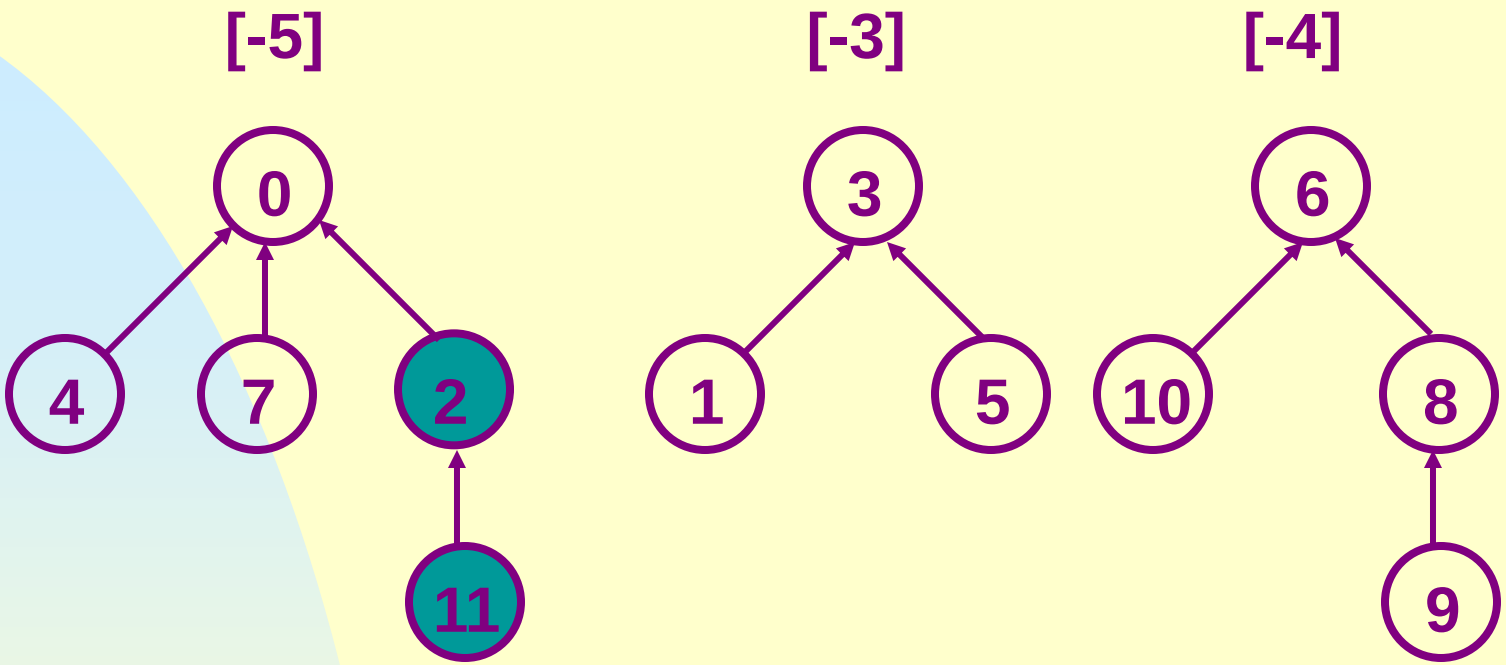
(a) Initial trees



(b) After processing 0 ≡ 4, 3 ≡ 1, 6 ≡10, and 8 ≡ 9



(c) After processing 7 ≡ 4, 6 ≡ 8, 3 ≡ 5, and 2 ≡ 11



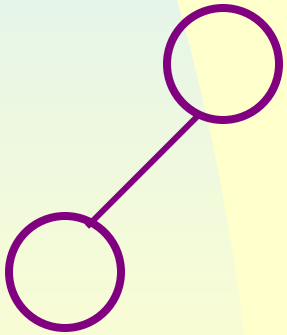
(d) After processing 11 ≡ 0

Exercises: P316-3

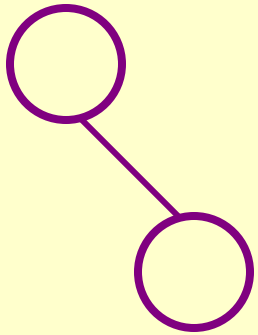
5.12 Counting Binary Trees

If n=0 or 1, there is only one binary tree .

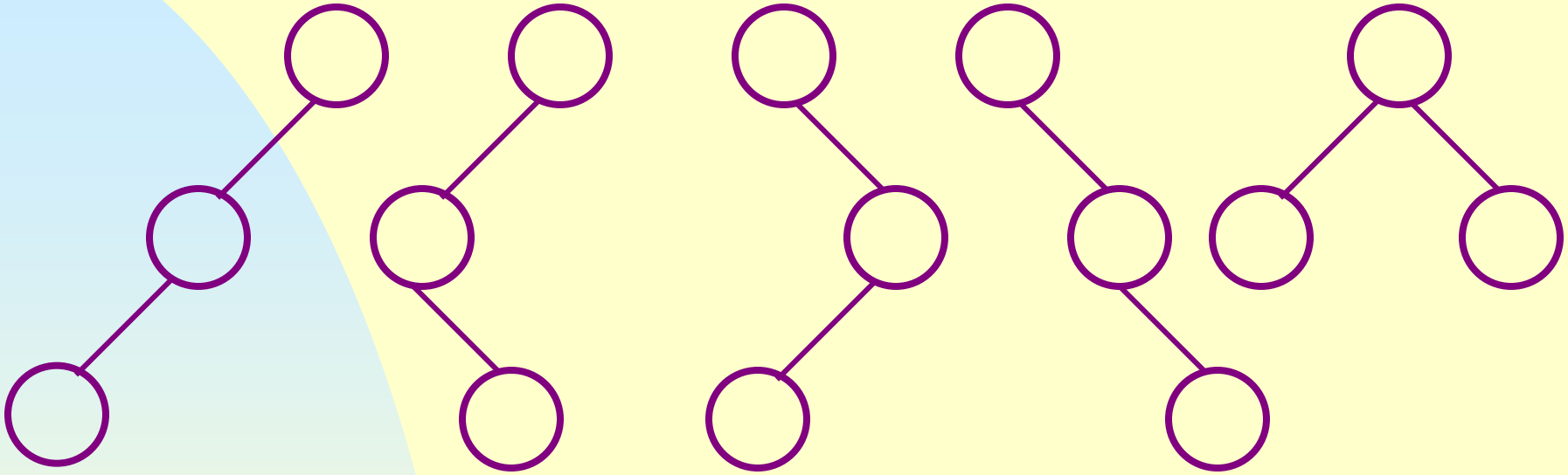
If n=2, there are two:



and



If n=3, there are five:



How many distinct binary trees are there with n nodes? Before deriving a solution, let's examine another equivalent problem.

Given the preorder sequence of node identifiers:

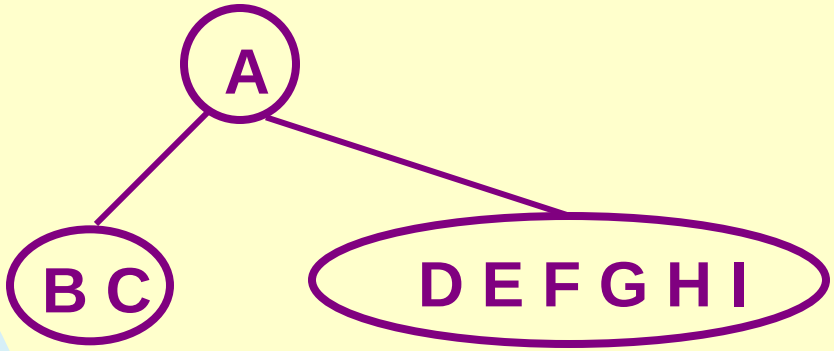
A B C D E F G H I

and the inorder sequence:

B C A E D G H F I

we can construct a unique binary tree by the following observation:

The 1st letter in the preorder, A, must be the root and by the definition of inorder, all the nodes preceding A must be in the left subtree and the remaining nodes in the right.

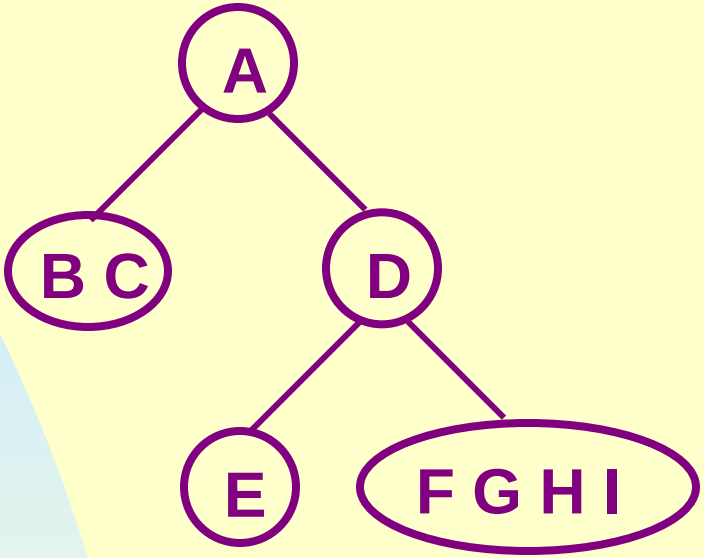


The subtrees, say the right, now has its:

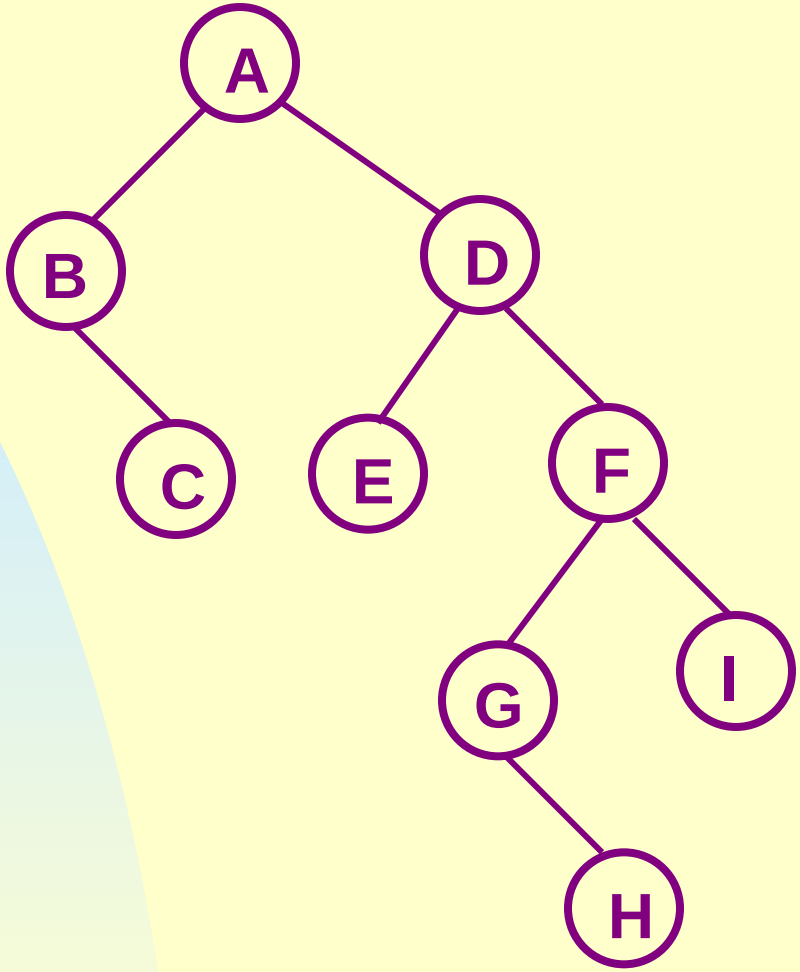
preorder: D E F G H I

inorder: E D G H F I

Using the above observation recursively, we have:



Continuing in this way, we finally get:


A
B
D
C
E
F
G
I
H

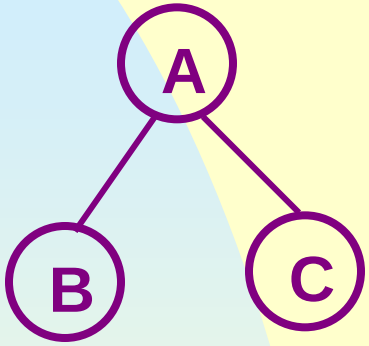
In general, we can develop an algorithm to construct a binary tree through its preorder/ inorder sequences.

In fact, we can prove by induction on n:

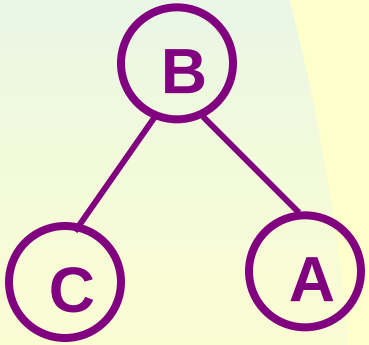
(1)Every pair of preorder/inorder sequences defines a unique binary tree.

(2)Every binary tree has a unique pair of preorder/inorder sequences if either its preorder or its inorder is fixed.

Note the if condition is essential, as otherwise, a binary tree may have many preorder/inorder sequences, as:



=> A B C / B A C



=> B C A / C B A

To be convenient, let the nodes of an n nodes binary tree be numbered 1,2,…,n.

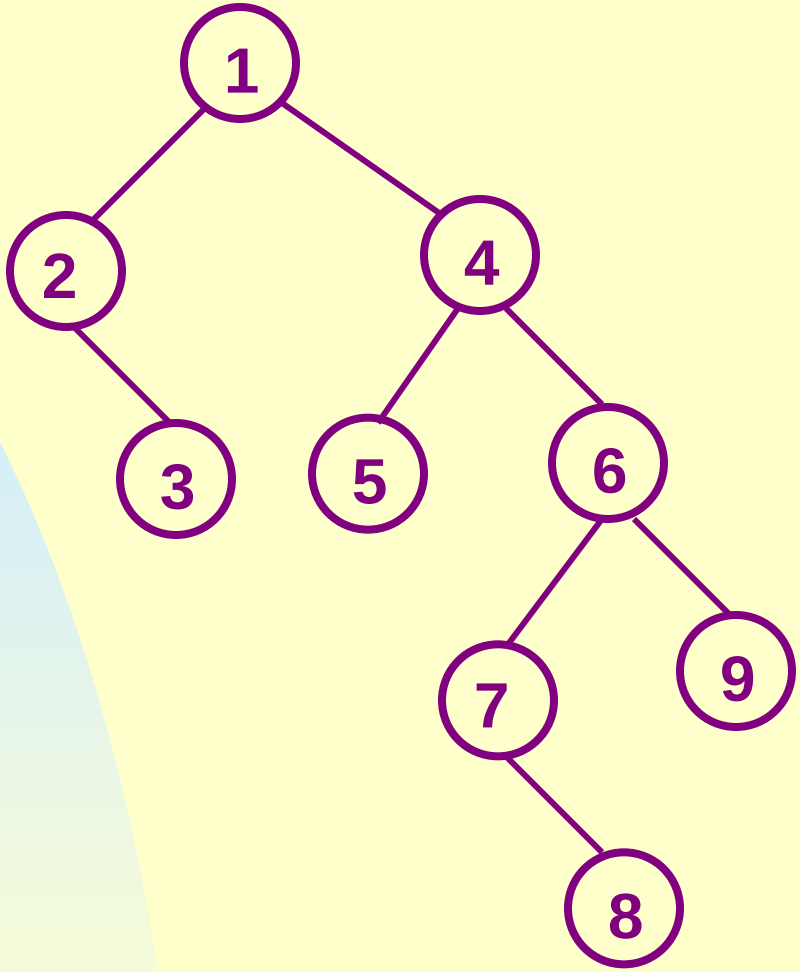
The **inorder permutation** defined by such a binary tree is the order in which its nodes are visited during an inorder traversal.

A **preorder permutation** is similarly defined.

For example, for the binary tree in the next slide,

Its preorder permutation: 1 2 3 4 5 6 7 8 9

Its inorder permutation: 2 3 1 5 4 7 8 6 9


1
2
4
3
5
6
7
9
8

If the nodes of a binary tree are numbered such that its preorder permutation is 1,2,…,n, then distinct binary trees define distinct inorder permutations.

The number of distinct binary trees = the number of distinct inorder permutations obtainable from binary trees having preorder permutation 1,2,…,n.

It can also be shown that:

The number of distinct permutation obtainable by passing 1,2,…,n through a stack and deleting in all possible ways = the number of distinct inorder permutations obtainable from binary trees having preorder permutation 1,2,…,n.

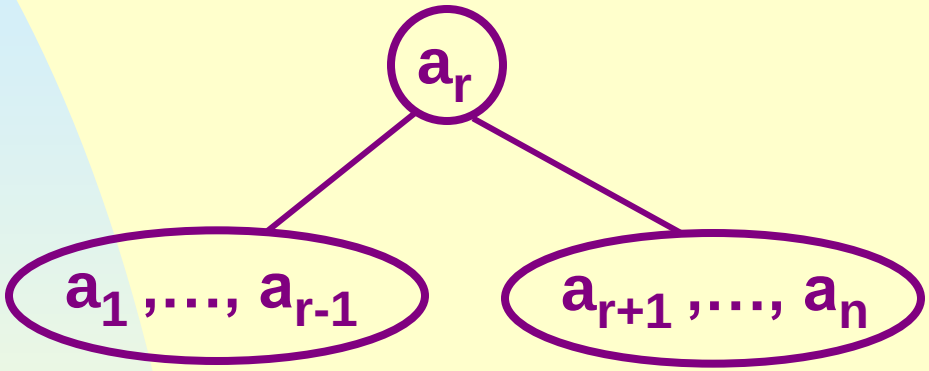
i.e. with preorder permutation i,i+1,…,j, given an inorder permutation we can get an equivalent stack permutation and vice versa.

Proof by induction on k=j-i+1.

k=1,clearly true.

Assume for k<n, it is also true.

For k=n, let $a_1, a_2, \ldots, a_r, a_{r+1}, \ldots, a_n$ be the inorder permutation, and a_r be the root of the tree, i.e.



let $b_1, \ldots, b_{r-1}$ be the preorder permutation of subtree $(a_1, \ldots, a_{r-1})$ and $c_1, \ldots, c_{n-r}$ be the preorder permutation of subtree $(a_{r+1}, \ldots, a_n)$.

Then

$a_r=i$, $(1 \le r \le n)$

$b_1,\ldots, b_{r-1}=i+1, i+2, \ldots, i+r-1,$

$c_1,\ldots, c_{n-r}= i+r, i+r+1, \ldots, j.$

Now with i,i+1,…,j, we can get the inorder permutation through a stack.

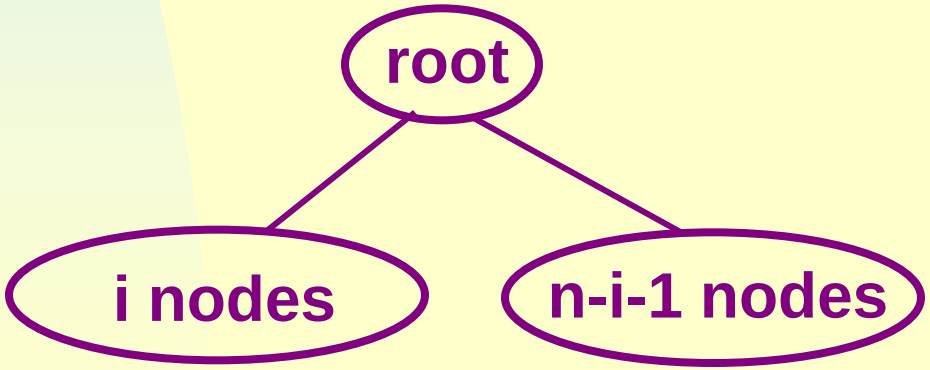
First, put $a_r=i$ into the stack, then by induction hypothesis, with $b_1,\ldots, b_{r-1}=i+1, i+2, \ldots, i+r-1$, we can get $a_1, a_2,\ldots, a_{r-1}$. Pop i, we get $a_1, a_2,\ldots,a_{r-1}, a_r$. Finally, with $c_1,\ldots, c_{n-r}= i+r, i+r+1, \ldots, j$, we can get $a_{r+1},\ldots, a_n$, following $a_1, a_2,\ldots,a_{r-1}, a_r$, we get

$a_1, a_2,\ldots, a_r, a_{r+1},\ldots, a_n$.

Similarly, we can prove that given a stack permutation, we can get a equivalent inorder permutation.

Now let b_n be the number of distinct binary trees with n nodes, then b_n is the sum of all the possible binary trees formed in the following way:

A root and two subtrees with i nodes and n-i-1 nodes, for $0 \leq i \leq n-1$.



For each i, we have $b_i.b_{n-i-1}$ distinct trees, so

$$b_n = \sum_{i=0}^{n-1} b_i\, b_{n-i-1}\ , \quad n \geq 1, \text{ and } b_0 = 1 \qquad (5.3)$$

To obtain b_n in term of n, we must solve the recurrence of Eq. (5.3).

Let $B(x) = \sum_{i \geq 0} b_i\, x^i$ (5.4)

Which is the generating function for the number of binary trees.

Since

$$B(x)=1+\sum_{i\geq 1} b_i x^i$$

$$=1+\sum_{i\geq 1}\sum_{j=0}^{i-1} b_j b_{i-j-1} x^i$$

$$=1+x[\sum_{i\geq 1}\sum_{j=0}^{i-1} b_j b_{i-j-1} x^{i-1}]$$

$$=1+x[\sum_{i\geq 0}\sum_{j=0}^{i} b_j b_{i-j} x^i]$$

$$=1+x[\sum_{k,l\geq 0} b_k b_l x^{k+l}]$$

$$=1+xB^2(x)$$

$xB^2(x) = B(x) - 1$

To solve this quadratic and note $B(0) = b_0 = 1$, we get

$$B(x) = \frac{1 - \sqrt{1 - 4x}}{2x}$$

Using binomial theorem to expand $(1-4x)^{1/2}$ to obtain

$$B(x) = \frac{1}{2x}\left(1 - \sum_{n \geq 0} \binom{1/2}{n} (-4x)^n\right)$$

$$= \frac{1}{2x}\left(1 - \binom{1/2}{0}(-4x)^0 - \sum_{n \geq 1} \binom{1/2}{n}(-4x)^n\right)$$

Let n=m+1, we have

$$B(x)= \frac{1}{2x}\left(-\sum_{m\geq 0}\binom{1/2}{m+1}(-1)^{m+1}.(2^2)^{m+1}.x^{m+1}\right)$$

$$= \frac{1}{2x}\left(\sum_{m\geq 0}\binom{1/2}{m+1}(-1)^{m}.2^{2m+2}.x^{m+1}\right)$$

$$=\sum_{m\geq 0}\binom{1/2}{m+1}(-1)^{m}.2^{2m+1}.x^{m} \qquad (5.5)$$

Comparing Eqs.(5.4) and (5.5), we see that b_n , which is the coefficient of x^n in B(x), is

$$b_n = \binom{1/2}{n+1} (-1)^n 2^{2n+1}$$

$$= \frac{\frac{1}{2}(\frac{1}{2}-1)\ldots(\frac{1}{2}-n)}{(n+1)!} (-1)^n 2^{2n+1}$$

$$= \frac{\frac{1}{2}.\frac{1}{2}.\frac{3}{2}\ldots.\frac{2n-1}{2}.2^{n+1}}{(n+1)!} \; 2^n$$

$$= \frac{1.3.5\ldots(2n-3)(2n-1).2^n}{(n+1)!} \cdot \frac{n!}{n!}$$

$$= \frac{1.3.5...(2n-3)(2n-1).2.4.6...(2n-2)(2n)}{(n+1)!n!}$$

$$= \frac{(2n)!}{(2n-n)!n!} \cdot \frac{1}{n+1}$$

$$= \frac{1}{n+1}\binom{2n}{n}$$

$$= O(4^n/n^{3/2})$$

Exercises P323-4,5

Chapter 6
Graphs

6.1 The Graph Abstract Data Type

6.1.1 introduction

Graphs have been widely used in:

- **analysis of electrical circuits**
- **finding shortest routes**
- **project planning**

- identification of chemical compounds
- statistical mechanics
- genetics
- cybernetics
- linguistics
- social science
- …

6.1.2 Definitions

- graph G = (V, E)
- vertices V(G) ≠ ∅
- edges E(G)
- undirected graph: (u, v) = (v, u)
- directed graph: <u, v>, u---tail, v---head,

<u,v> ≠ <v,u>

Three graphs:

G_1:

$V(G_1)=\{0,1,2,3\}$

$E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

G_2:

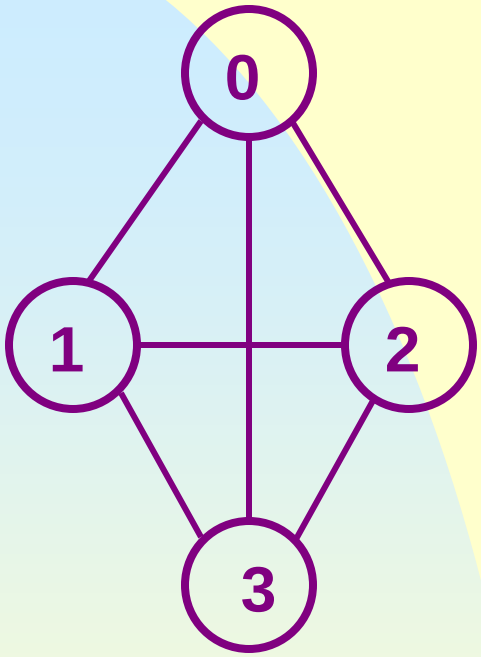
$V(G_2)=\{0,1,2,3,4,5,6\}$

$E(G_2)=\{(0,1),(0,2),,(1,3),(1,4),(2,5),(2,6)\}$

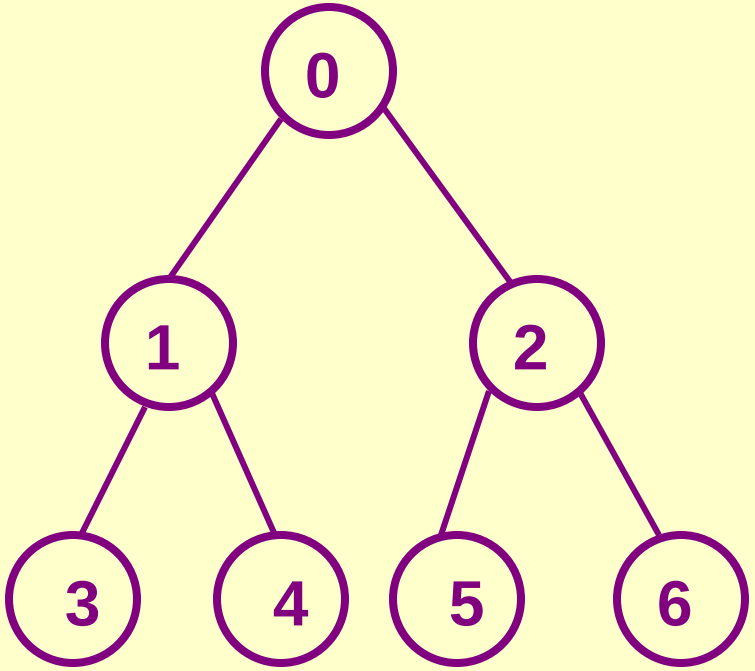
G_3:

$V(G_3) = \{0,1,2\}$

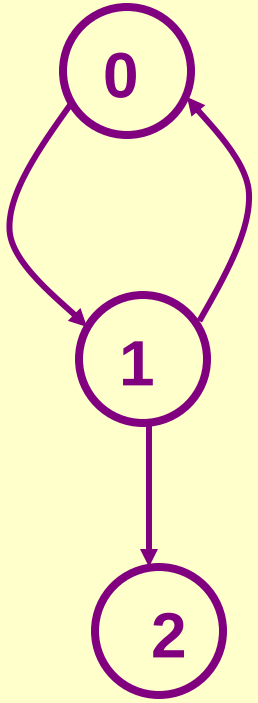
$E(G_3) = \{<0,1>,<1,0>,<1,2>\}$ (directed)



(a) G_1



(b) G_2



(c) G_3

Restrictions:

(1) (v, v) or <v, v> is not legal, such edges are known as self edges.

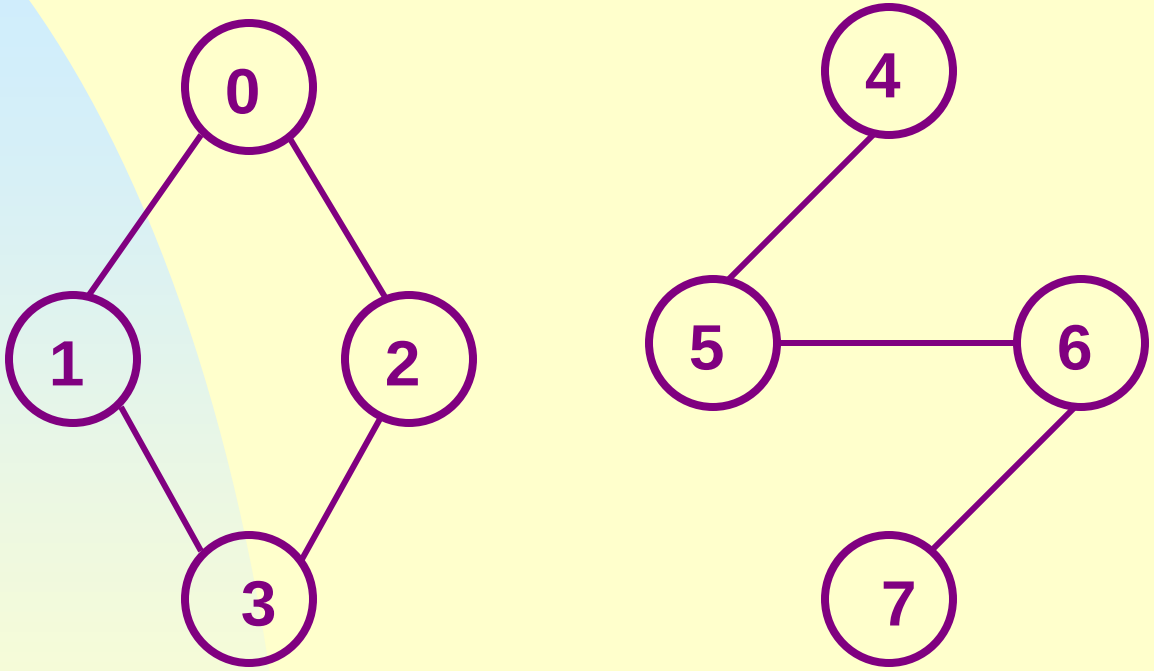
(2) multiple occurrences of the same edges are not allowed. If allowed, we get a multigraph.

- **The maximum number of edges in any n-vertex, undirected graph is n(n-1)/2, and in directed graph is n(n-1).**
- **An n-vertex undirected graph with n(n-1)/2 edges is said to be complete.**

- If (u, v) $\in$E(G), we say u and v are **adjacent** and edge (u, v) is **incident** on vertices u and v. If <u, v> is a directed edge, then vertex u is **adjacent to** v, and v is **adjacent from** u, <u, v> is **incident to** u and v.
- A **subgraph** of G is a graph G` such that V(G`) $\subseteq$ V(G) and E(G`) $\subseteq$ E(G).
- A **path** from u to v in G is a sequence of vertices u, i_1, i_2,…, i_k, v such that (u, i_1), (i_1, i_2),…,(i_k, v) are edges in E(G). If G` is directed, then <u, $i_1$>, <$i_1$, $i_2$>,…,<$i_k$, v> are edges in E(G`).
- The **length** of a path is the number of edges on it.

- A **simple path** is a path in which all vertices except possibly the first and last are distinct.
- A **cycle** is a simple path in which the first and last vertices are the same.
- For directed graph, we have directed paths and cycles.
- In an undirected G, u and v are **connected** iff there is a path in G from u to v (also from v to u).
- An undirected G is **connected** iff for every pair of distinct u and v in V(G), there is a path from u to v.

• **A connected component is a maximal connected subgraph.**



G_4 A graph with two connected components

- A **tree** is a connected acyclic graph.
- A directed G is **strongly connected** iff for every pair of distinct u and v in V(G), there is a directed path from u to v and also from v to u.
- A **strongly connected component** is a maximal subgraph that is strongly connected.
- The **degree** of a vertex is the number of edges incident to it.

- For directed G, the **in-degree** of $v \in V(G)$ is the number of edges for which v is the head. The **out-degree** is the number of edges for which v is the tail.
- If d_i is the degree of vertex i in G with n vertices and e edges, then

$$e = \left(\sum_{i=0}^{n-1} di\right)/2$$

- We'll refer to a directed graph as **digraph**, and a undirected graph as **graph**.

ADT 6.1 Graph

```
class Graph
{ // A non empty set of vertices and a set of undirected
  // edges, where each edge is a pair of vertices.
public:
    virtual ~Graph(){ };
       // virtual destructor
    bool IsEmpty() const {return n==0;};
       // return true iff graph has no vertices
    int NumberOfVertices() const {return n;};
       // return the number of vertices in the graph
    int NumberofEdges() const {return e;};
       // return number of edges in the graph
    virtual int Degree(int u) const =0;
       // return number of edges incident to vertex u
```

```
    virtual bool ExistsEdge(int u, int v) const =0;
      // return true iff graph has edge (u, v)
    virtual void InsertVertex (int v) =0;
      // insert vertex v into graph, v has no incident edges
    virtual void InsertEdge (int u, int v) =0;
      // insert edge (u, v) into graph
    virtual void DeleteVertex (int v);
      // delete v and all edges incident to it
    virtual void DeleteEdge (int u, int v) =0;
      // delete edge (u, v) from the graph
private:
     int n;    // number of vertices
     int e;    // number of edges
};
```

6.1.3 Graph Representations

Three most commonly used representations:

(1) Adjacency matrices

(2) Adjacency lists

(3) Adjacency multilists

The actual choice depends on application.

6.1.3.1 Adjacency Matrix

The adjacency matrix of G is a n×n array, say a, such that:

$$a[i,j] = \begin{cases} 1 & \text{iff } (i, j) \in E(G) \text{ (or } \langle i, j\rangle \in E(G) \text{)} \\ 0 & \text{otherwise} \end{cases}$$

The next slide show adjacency matrices of G_1 , G_3 and G_4.

	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

(a) G_1

	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

(b) G_3

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	1	0	1
7	0	0	0	0	0	0	1	0

G_4

For an graph, a is symmetric, and

$$d_i = \sum_{j=0}^{n-1} a[i][j]$$

For a digraph, a may not be symmetric, and

$$\text{out-}d_i = \sum_{j=0}^{n-1} a[i][j]$$

$$\text{in-}d_j = \sum_{i=0}^{n-1} a[i][j]$$

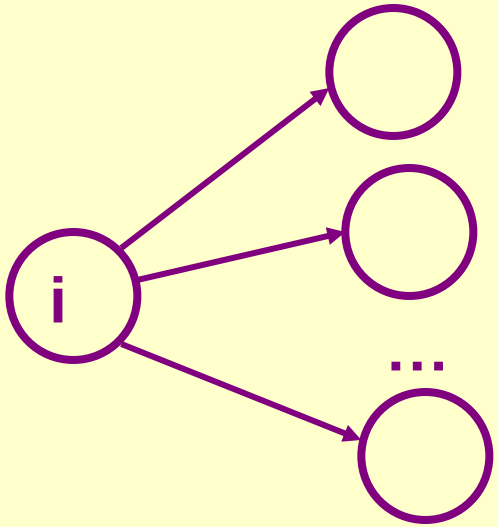
Problem: how many edges are there in G?

Using a, we need $O(n^2)$. When $e << n^2/2$, and if we explicitly represent only edges in G, then we can solve the above problem in $O(e + n)$.

This lead to:

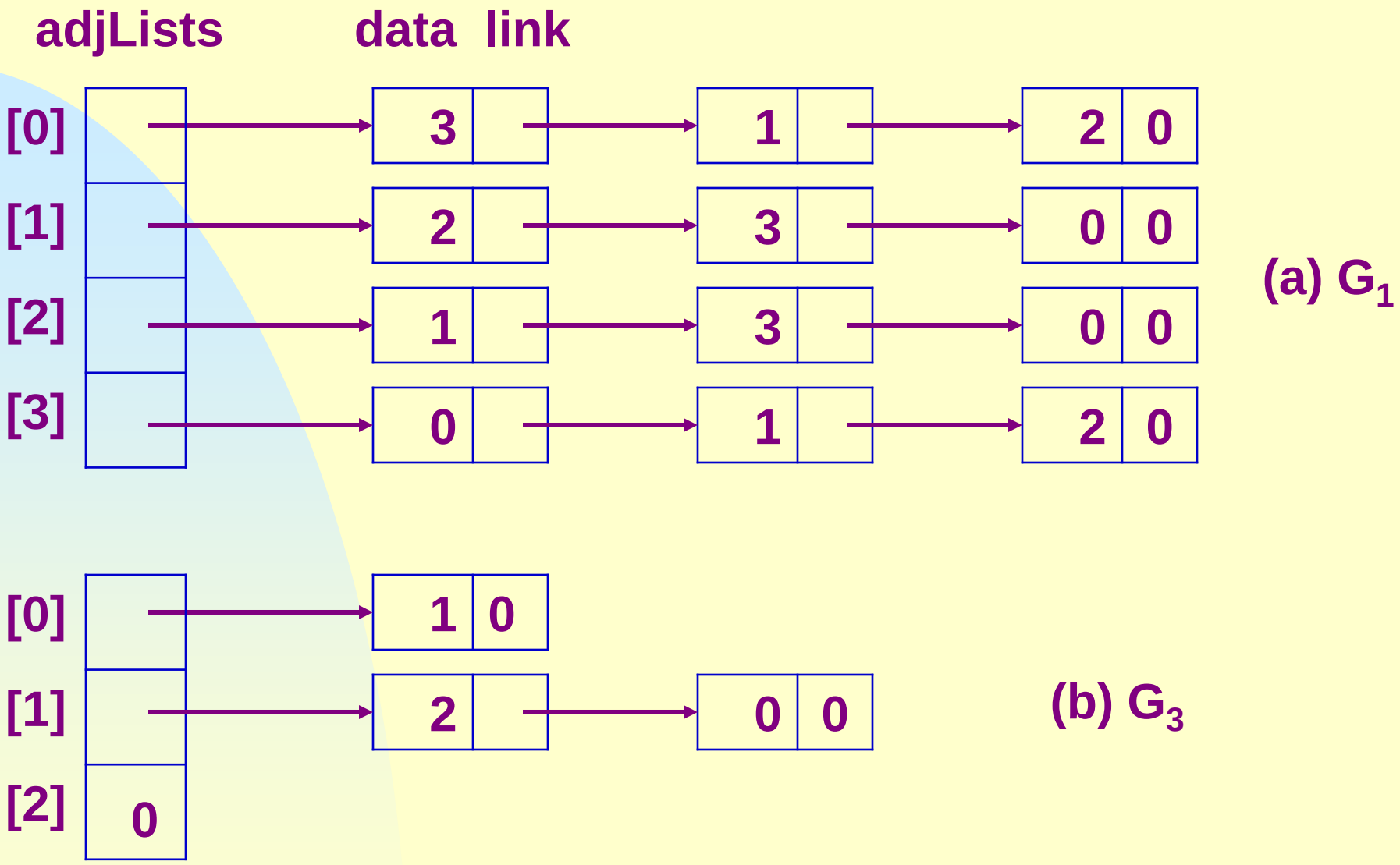
6.1.3.2 Adjacency Lists

- The n rows of the adjacency matrix are represented as n chains.
- The nodes in chain i represent the vertices adjacent from i.

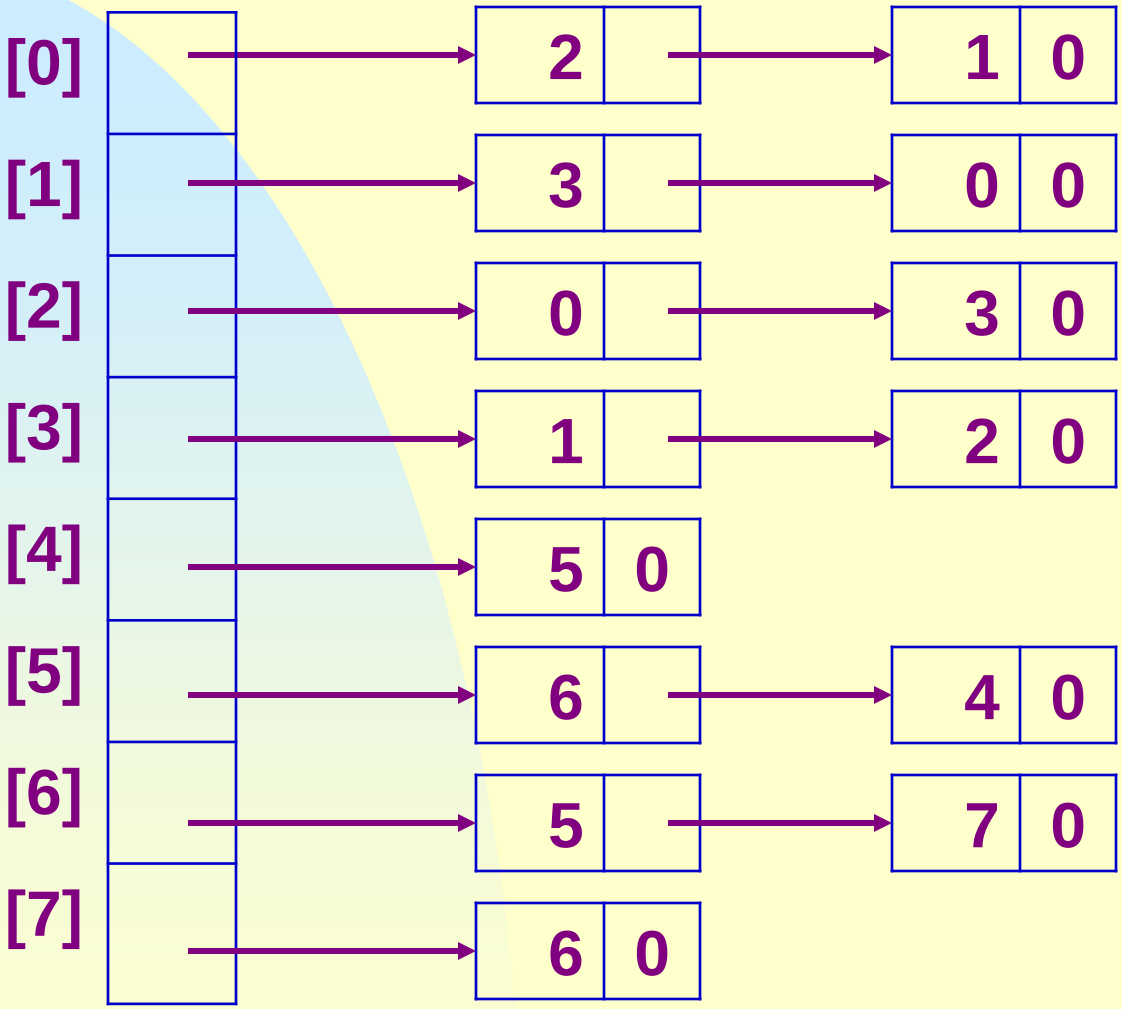


- The **data** field of a chain node store the index of an adjacent vertex.
- The vertices in each chain are not required to be ordered.
- An array **adjLists** is used for accessing any chain in O(1).

```
class LinkedGraph
{
public:
   LinkedGraph (const int vertices): e(0)
   {
      if (vertices < 1) throw "Number of vertices must be > 0";
      n = vertices;
      adjLists = new Chain<int>[n];
      Chain<int> c; // set c.first to 0
      fill(adjLists, adjLists+n,c);
   };
private:
   Chain<int>* adjLists;
   int n;
   int e;
};
```


adjLists
data link
[0]
[1]
[2]
[3]
3
1
2 0
2
3
0 0
1
3
0 0
0
1
2 0
(a) G1
[0]
[1]
[2]
1 0
2
0 0
0
(b) G3

adjLists



(c) G_4

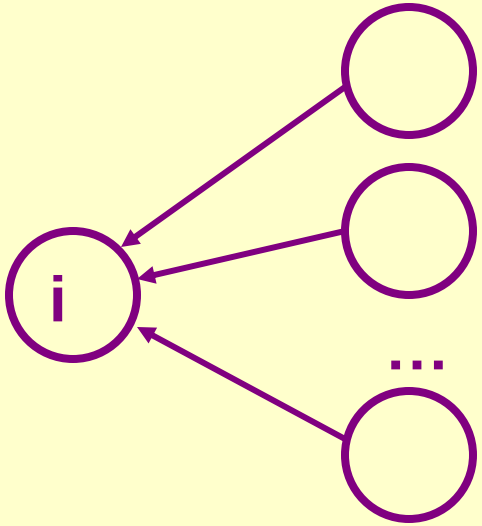
For an undirected graph with n vertices and e edges, this representation requires an array of size n and 2e chain nodes.

Now it is possible to determine the total number of edges in G in O(e+n).

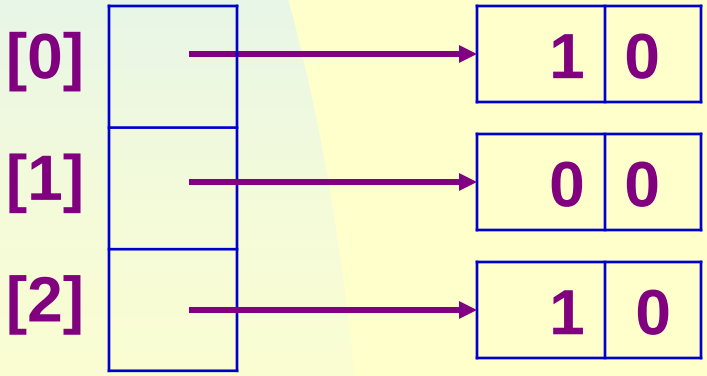
In case of digraph, the in-degree of a vertex is a little more complex to determine --- use a vector c[n], when traverse from i to j, c[j]++.

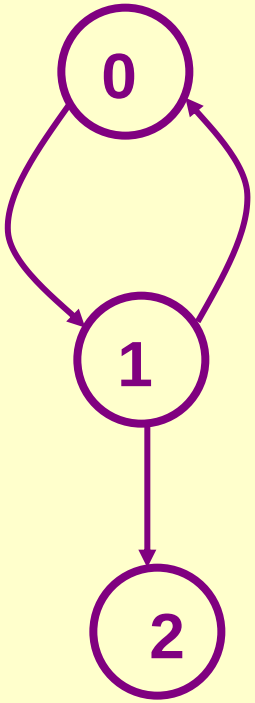
Inverse adjacency lists:

- one list for each vertex
- the nodes in list i represent the vertices adjacent to vertex i.



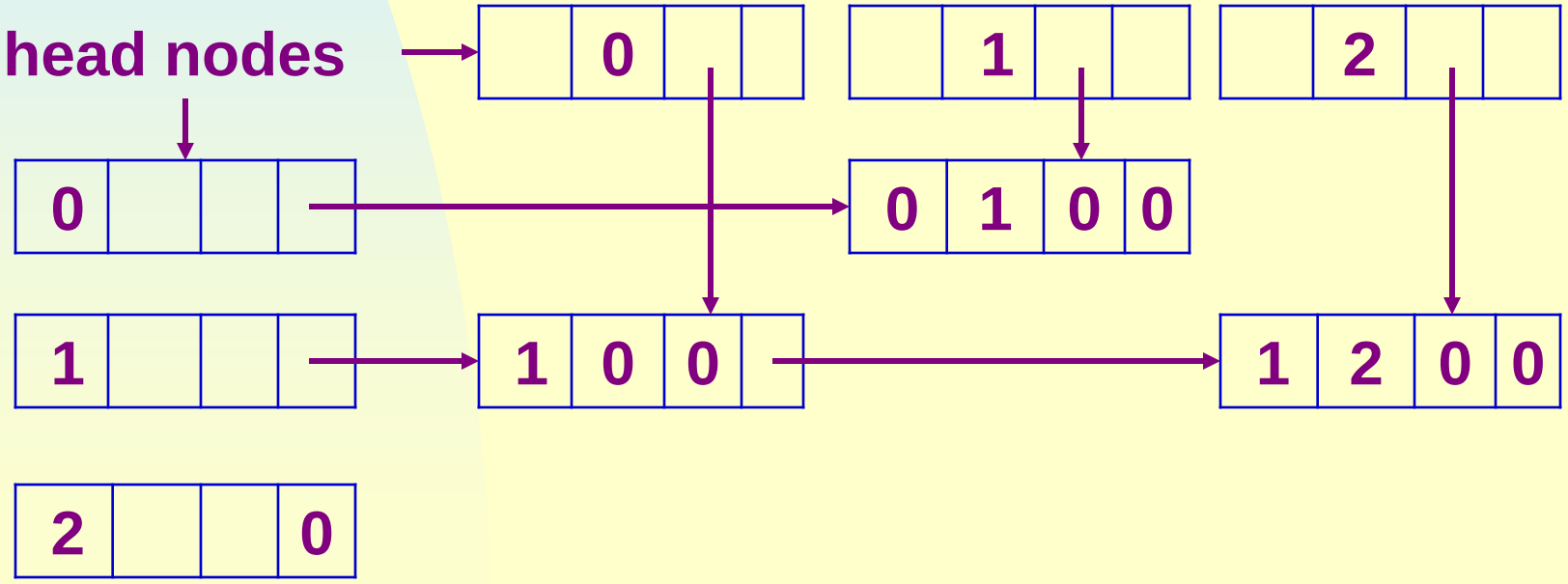
The following is the inverse adjacency lists for G_3 :





Combining nodes for adjacency lists and inverse adjacency lists together forms an orthogonal list node:

Here is the orthogonal representation for G_3:



6.1.3.3 Adjacency Multilists

In the adjacency lists of an undirected graph, each (u, v) is represented by 2 entries.

In some situations, it is necessary to be able to determine the second entry for a particular edge and mark that edge as used.

So we need **multilists**: for each edge there is exactly one node, but it is in two lists.

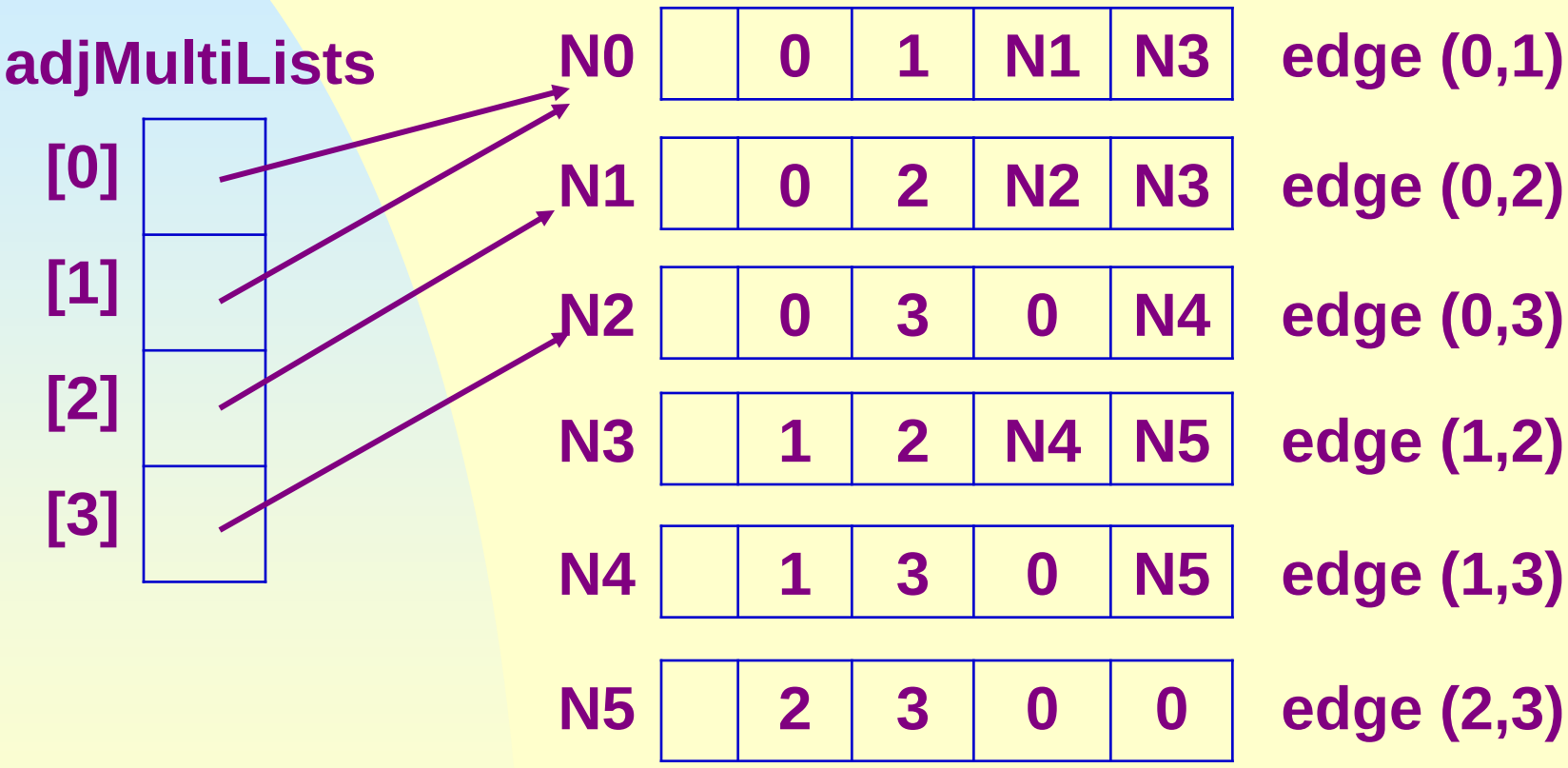
m	vertex1	vertex2	path1	path2

Note an edge can be marked from either path1 or path2.

```
class MGraph;
class MGraphEdge {
friend MGraph;
private:
    bool m;
    int vertex1, vertex2;
    MGraphEdge *path1, *path2;
};
typedef MGraphEdge *EdgePtr ;
class MGraph {
public:
     MGraph(const int);
private:
    EdgePtr *adjMultiLists;
    int n;
    int e;
};
```

```
MGraph::MGraph(const int vertices) : e(0)
{
    if (vertices < 1) throw “Number of vertices must be > 0”;
    n = vertices;
    adjMultiLists = new EdgePtr[n];
    fill(adjMultiLists, adjMultiLists+n,0);
}
```

Here is the adjacency multilists for G_1:



If p points to an MGraphEdge representing (u, v), and given u, to get v we need the following test:

```
if (p→vertex1 == u) v = p→vertex2;else v = p→vertex1;
```

And we can insert an edge in O(1):

```
void MGraph::InsertEdge(int u, int v) {
  MGraphEdge *p = new MGraphEdge;
  p→m = false; p→vertex1 = u; p→vertex2 = v;
  p→path1 = adjMultiLists[u]; p→path2 = adjMultiLists[v];
  adjMultiLists[u] = adhMultiLists[v] = p;
}
```

6.1.3.4 weighted Edges

Edges may have weight.

- **In the case of adjacency matrix, A[i][j] may keep this information.**
- **In the case of adjacency lists, we need a weight field in the list node.**
- **A graph with weighted edges is called a network.**

Exercises: P340-5, 9

6.2 Elementary Graph Operations

Given G = (V, E), and v in V(G), we wish to visit all vertices in G that are reachable from v.

In the following methods, we assume the graphs are undirected, although they work on the directed as well.

6.2.1 Depth-First Search

Idea:

Visit the start v first, next an unvisited w adjacent to v is selected and a depth-first search from w initiated. When u is reached such that all its adjacent vertices has been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w. The search terminates when no unvisited vertex can be reached from any of the visited vertices.

```
virtual void Graph::DFS()   // Driver
{
    visited=new bool[n];
     // visited is declared as a bool* data member of Graph
    fill(visited, visted + n, false);
    DFS (0);      // start search at vertex 0.
    delete [ ] visited;
}

virtual void Graph::DFS ( const int v) // workhorse
{//visit all previously unvisited vertices reachable from v
   visited[v] = ture;
   for (each vertex w adjacent to v)  // actual code uses an
                                      // iterator
      if ( !visited[w] ) DFS ( w);
}
```

Example 6.1

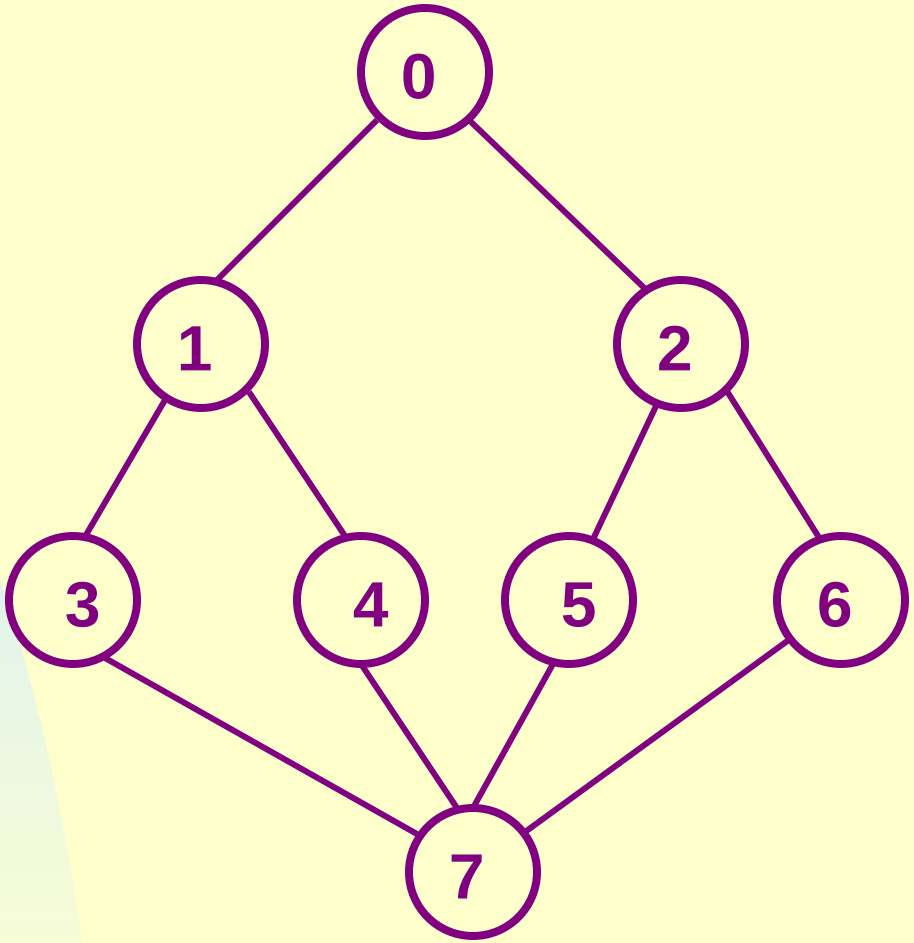


Fig. 6.17 (a) Graph G

adjLists

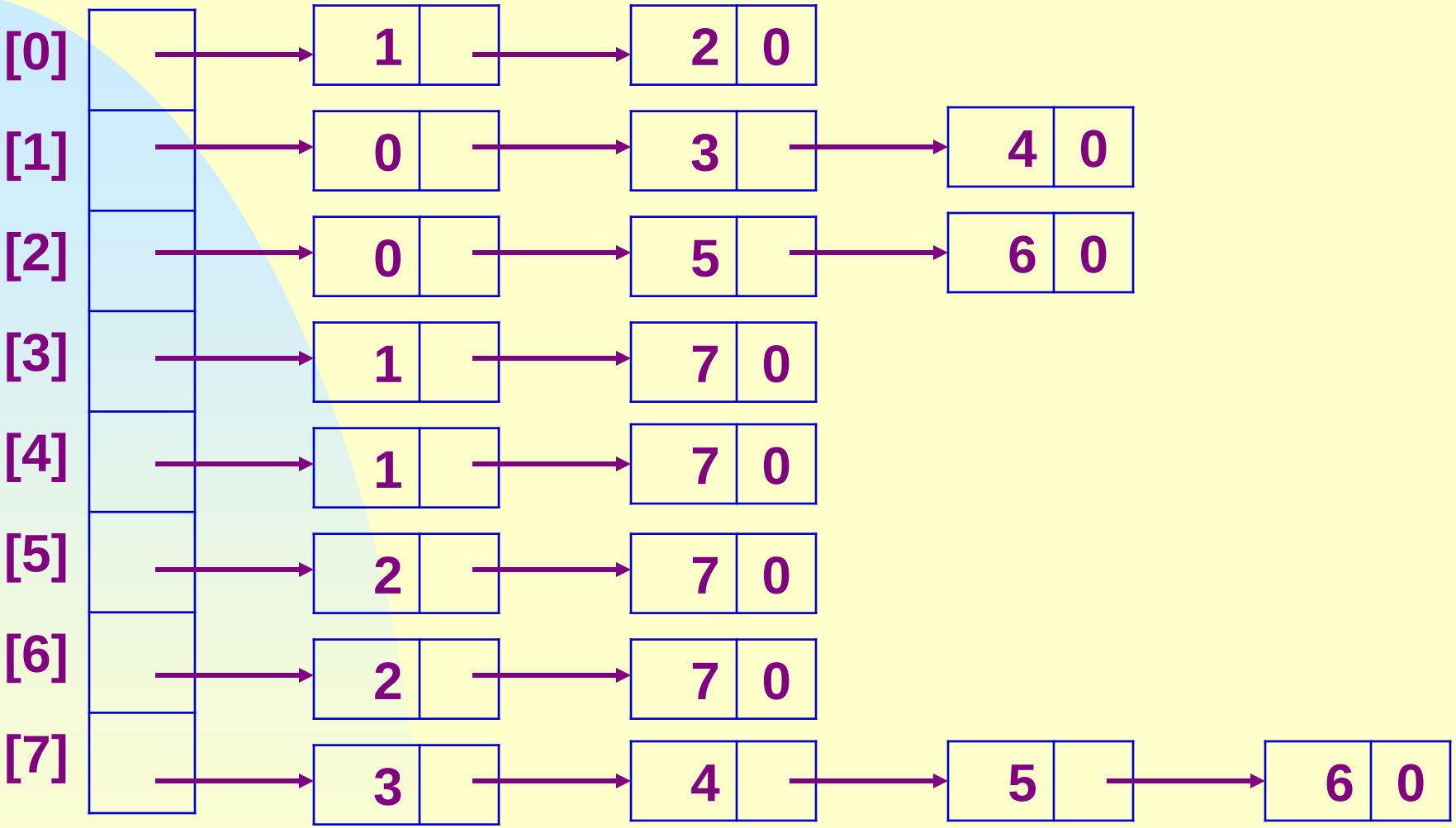


Fig 6.17 (b) Adjacency lists of G

Started from vertex 0, the vertices in G are visited in the order: 0, 1, (0), 3, (1), 7, (3), 4, (1), (7), (back to 7), 5, 2, (0), (5), 6, (2), (7), (back to 2), (back to 5), (7), (back to 7), (6), (back to 3), (back to 1), (4), (back to 0), (2).

Analysis of DFS:

Initiating visited needs O(n). When G is represented by its adjacency lists, DFS examines each node at most once, so the time is O(e+n).

If adjacency matrix is used, the time to determine all vertices adjacent to v is O(n), there are n vertices, the total time is $O(n^2)$.

6.2.2 Breadth-First Search

In a breadth-first search, we begin from the start vertex v. Next, all unvisited vertices adjacent to v are visited. Unvisited vertices adjacent to these newly visited vertices are then visited, and so on.

```
virtual void Graph::BFS (int v)
{ // breadth-first search: begin at v, use a queue.
    visited = new bool[n];
    fill(visited, visited + n, false);
    visited[v] = true;
    Queue<int> q;
    q.Push(v);
```

```
    while ( !q.IsEmpty()) {
        v = q.Front();
        q.Pop();
        for (all vertices w adjacent to v)  // actual code uses an
                                            //  iterator
            if (!visited[w]) {
                q.Push(w);
                visited[w] = true;
            }
     }    // end of while
    delete [ ] visited;
}
```

Example 6.2:

Started from vertex 0, the vertices in G of Fig. 6.17 are visited by breadth-first search in the order:

0, 1, 2, 3, 4, 5, 6, 7.

Analysis of BFS:

Each visited vertex enters the queue exactly once, the while loop is iterated at most n times.

If adjacency matrix is used, the loop takes O(n) for each node visited, the total time is $O(n^2)$.

If adjacency lists are used, the loop has a total cost of $d_0+,\ldots,+ d_{n-1} = O(e)$, the total time is O(n+e).

6.2.3 Connected Components

To obtain all the connected components of a undirected graph, we can make repeated calls to either DFS(v) or BFS(v) for unvisited v.

This leads to function **Components**.

Function **OutputNewComponent** output all vertices visited in the most recent invocation of DFS, together with all edges incident on them.

```
virtual void Graph::Components()
{ // Determine the connected components of the graph.
    visited = new bool[n];
    fill(visited, visited+n, false);
    for (int i=0; i<n; i++)
       if (!visited[i]) {
           DFS (i); // find a component
           OutputNewComponent();
       }
    delete [ ] visited;
}
```

Analysis of Components:

If adjacency lists are used, the total time taken by DFS is O(e+n). The output can be completed in O(e+n) if DFS keeps a list of all newly visited vertices. The for loops take o(n). The total time is O(e+n).

If adjacency matrix is used, the total time is $O(n^2)$.

6.2.4 Spanning Trees

If G is connected, in DFS or BFS, all vertices are visited, the edges of G are partitioned into 2 sets:

- **T --- Tree edges**
- **N --- Nontree edges**

T may be obtained by inserting "$T = T \cup \{(v, w)\}$" in the if clauses of DFS or BFS.

Any tree consisting solely of edges in G and including all vertices in G is called a spanning tree.

- **Depth-first spanning tree --- the spanning tree resulting from a depth-first search**
- **Breadth-first spanning tree --- the spanning tree resulting from a breadth-first search**

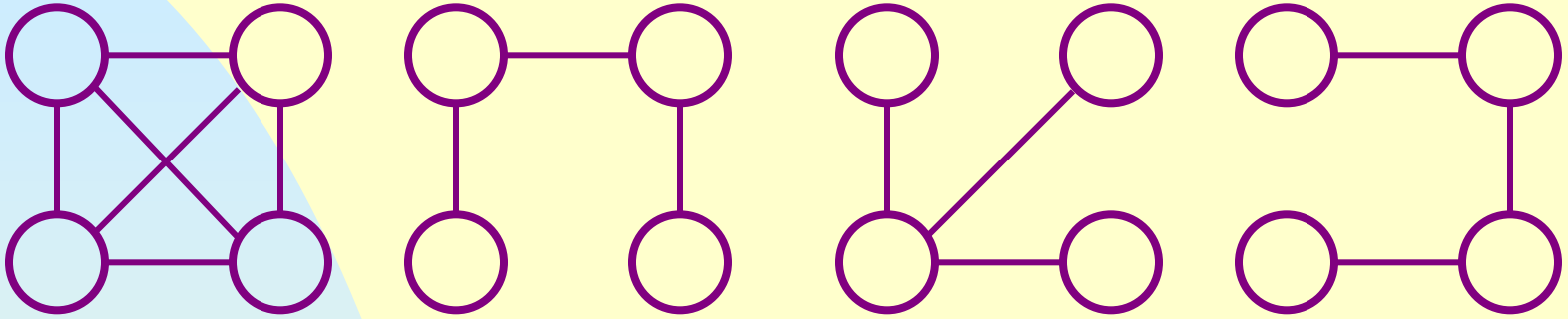


Fig. 6.18 A complete graph and three of its spanning trees

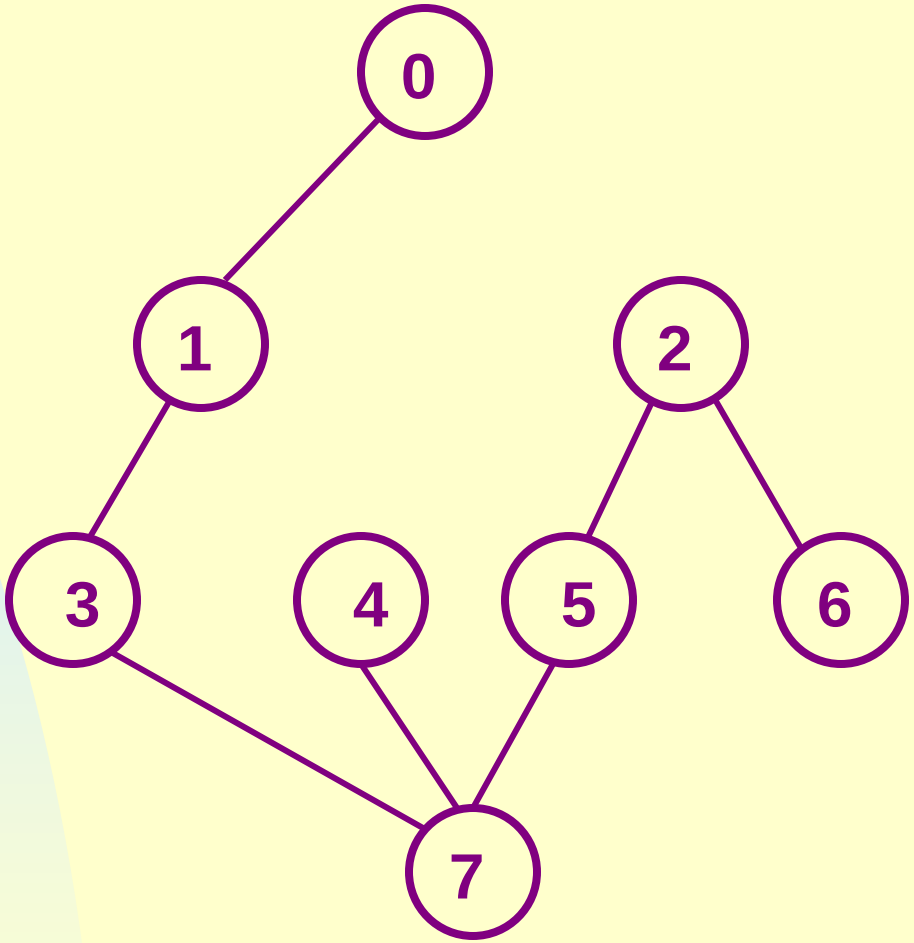


Fig. 6.19 (a) DFS(0) spanning tree for graph of Fig.6.17

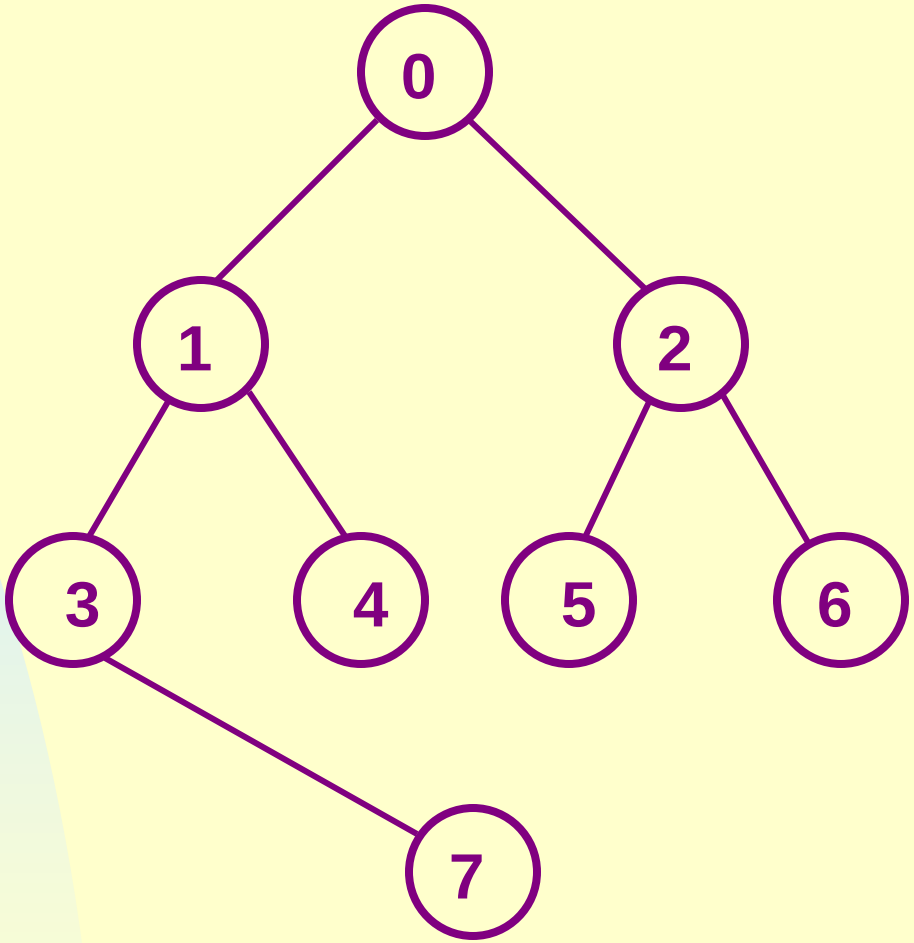


Fig. 6.19 (b) BFS(0) spanning tree for graph of Fig.6.17

If a nontree edge (v, w) is put into any spanning tree T, then a cycle is formed.

Example applications of spanning tree:

(1) Obtain an independent set of circuit equations for an electrical network by introducing nontree edges into the spanning tree one at a time.

(2) Connecting n cities with n-1 communication links.

In practical situation, edges will have weights (or cost) assigned to them. The cost of a spanning tree is the sum of the costs of the edges in that tree.

Exercises: P352-3, 5, 6

6.2.5 Biconnected Components

Assume that G is an undirected, connected graph.

Definition: A vertex v of G is an articulation point iff the deletion of v, together with the the deletion of all edges incident to v, leaves behind a graph that has at least two connected components.

Vertices 1 and 7 are the articulation points of the connected graph of Fig. 6.20(a).

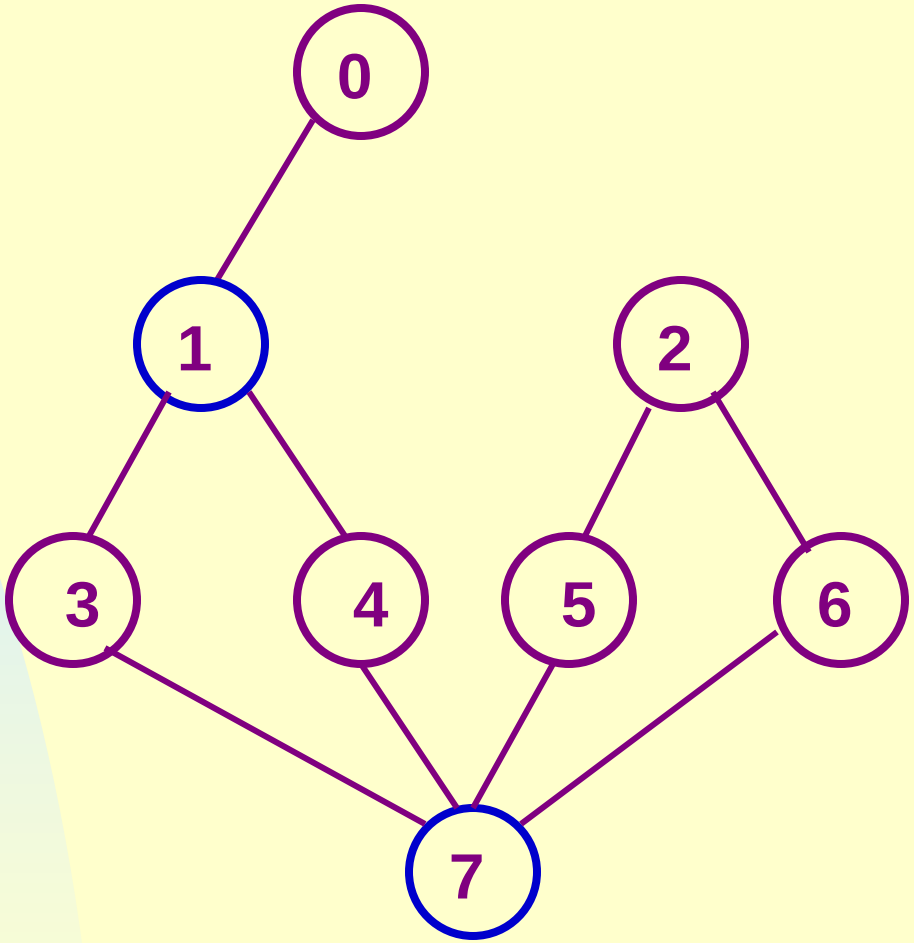


Fig. 6.20 (a) A connected graph

Definition: A biconnected graph is a connected graph that has no articulation points.

The graph of Fig. 6.20(a) is not biconnected.

Articulation points are undesirable in communication network.

Definition: A biconnected component of a connected graph G is a maximal biconnected subgraph H of G.

The graph of Fig. 6.20(a) contains three biconnected components, as shown in Fig. 6.20(b).

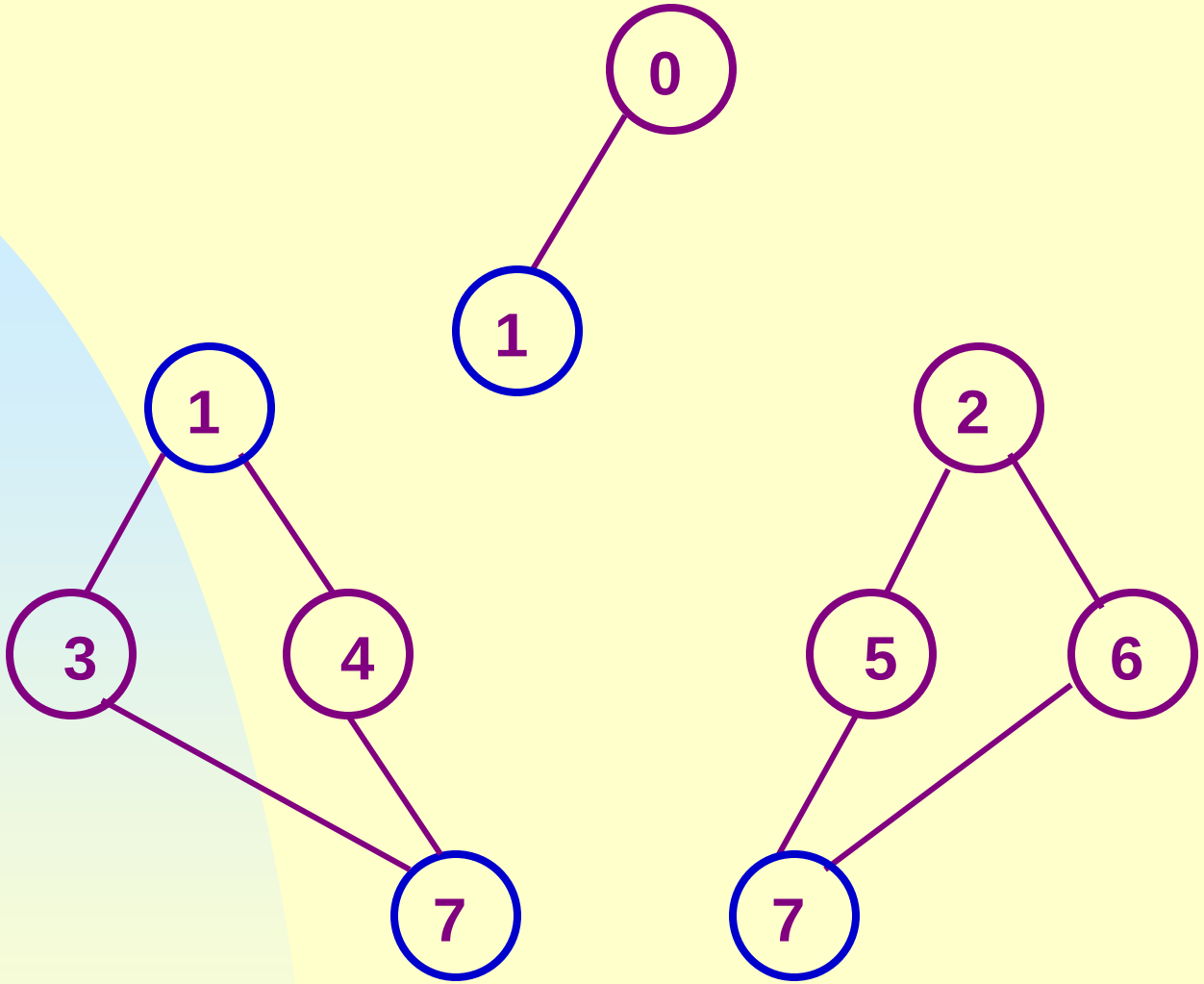


Fig. 6.20 (b) its biconnected components

Of the same graph G:

Two biconnected components can have at most one vertex in common.

No edge can be in two or more biconnected components, hence the biconnected components partition E(G).

To find the biconnected components of G, we can use any depth-first tree of it. For the graph of Fig. 6.20(a), a depth-first tree with root 0 is shown in Fig. 6.21 with the nontree edges shown by broken lines.

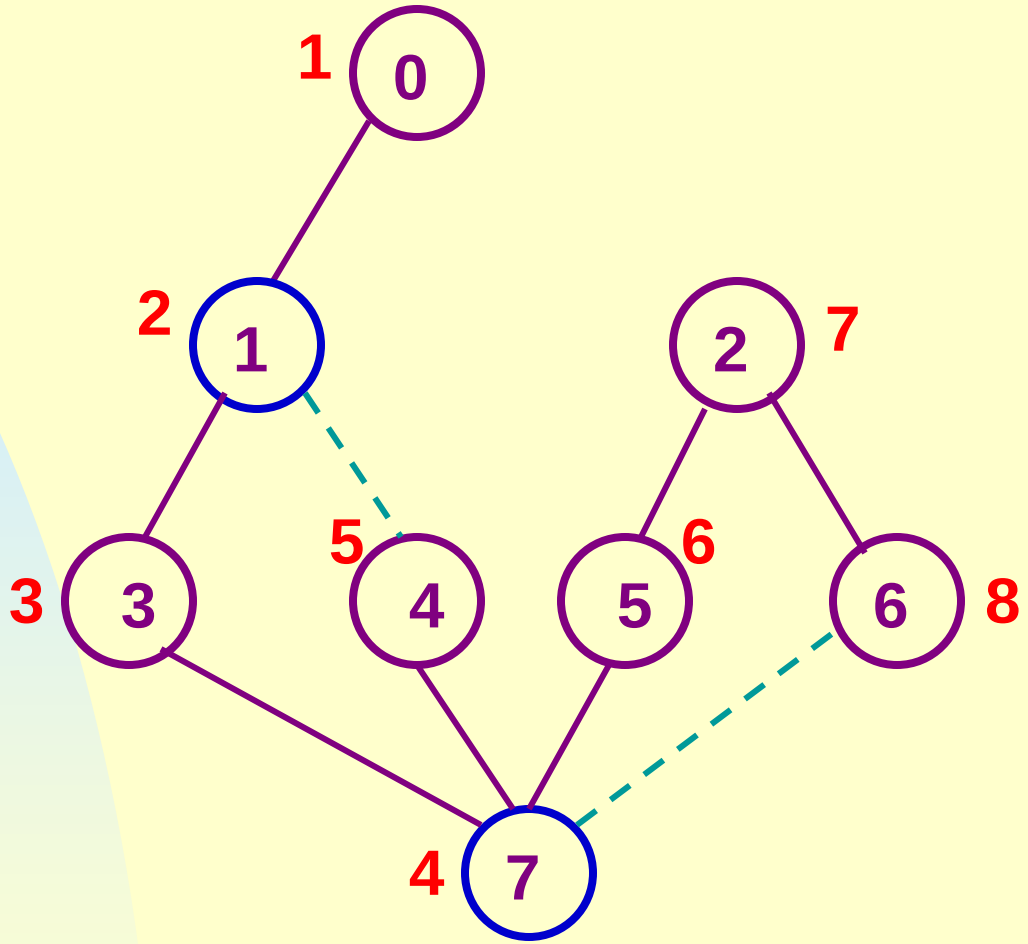


Fig. 6.21 A depth-first spanning tree of Fig. 6.19(a)

The numbers outside the vertices gives the sequence in which the vertices are visited during the depth-first search.

The number is called the **depth-first number**, dfn, of the vertex. For example, dfn(1)=2, dnf(7)=4, dfn(6)=8.

Note that if u is an ancestor of v in the depth-first tree , then dfn(u)<dfn(v).

A nontree edge (u, v) is a **back edge** with respect to a spanning tree T iff either u is an ancestor of v or v is an ancestor of u. (1, 4) and (6, 7) are back edges.

A nontree edge that is not a back edge is called a **cross edge**. No graph can have cross edges with respect to any its depth-first spanning trees.

So the root of the depth-first spanning tree is an articulation point iff it has at least two children.

Further, any vertex u is an articulation point iff it has **at least one** child, w, such that it is not possible to reach an ancestor of u using a path composed solely of w, descendants of w, and a single back edge.

To describe these, we define a value low for each $w \in V(G)$, as:

$$
\begin{aligned}
low(w) = \min \{ & \ dfn(w), \\
& \min \{ low(x) \mid x \text{ is a child of } w \}, \\
& \min \{ dfn(x) \mid (w, x) \text{ is a back edge} \} \\
& \}
\end{aligned}
$$

Thus u is an articulation point iff u is either the root of the spanning tree and has two or more children or u is not the root and u has a child w such that $low(w) \geq dfn(u)$.

vertex	0	1	2	3	4	5	6	7
dfn	1	2	7	3	5	6	8	4
low	1	2	4	2	2	4	4	2

Fig.6.22 dfn and low values for the spanning tree of Fig.6.21

Now we are ready to modify DFS to compute dfn and low as in the function DfnLow:

```
void Graph::DfnLow (const int x )   // begin DFS at vertex x
{
   num = 1;              // num is an int data member of Graph
   dfn = new int[n];         // dfn is declared as int * in Graph
   low = new int[n];        // low is declared as int * in Graph
   fill(dfn, dfn+n,0);
   fill(low, low+n,0);
   DfnLow ( x, -1 );  // x as the root, its parent is dummy -1
   delete [ ] dfn;
   delete [ ] low;
}
```

```
void Graph::DfnLow ( int u,  int v )
// compute dfn and low while performing a depth-first search
// beginning at u. v is the parent of u in the resulting spanning
// tree.
{
    dfn[u] = low[u] = num++;
    for ( each vertex w adjacent from u)
    {
        if ( dfn[w] == 0 ) { // w is an unvisited vertex
            DfnLow ( w, u );
            low[u] = min( low[u], low[w]);
        }
       else if ( w != v )  low[u] = min( low[u], dfn[w]);
                // back edge. note (v, u) is not a back edge.
}
```

Note that following the return from DfnLow(w, u),

- **low[w] has been computed.**
- **If low[w] ≥ dfn[u], then u is an articulation point and a new biconnected component has been identified.**
- **By using a stack to save edges when they are first encountered, we can output all edges in a biconnected component.**

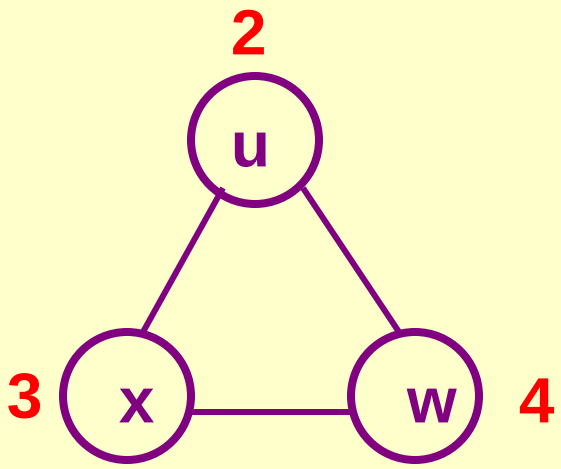
```
void Graph::Biconnected ( )
{
    num = 1;
    dfn = new int[n];
    low = new int[n];
    fill(dfn, dfn+n,0);
    fill(low, low+n,0);
    Biconnected ( 0, -1 );  // start at vertex 0
    delete [ ] dfn;
    delete [ ] low;
}
```

```
void Graph::Biconnected ( int u,  int v )
// compute dfn and low, and output the edges of the graph by
// its biconnected components. v is the parent of u in the
// resulting spanning tree. S is an initially empty stack declared
// as a data member of Graph. The function works only for n>1.
// when n==1, no edges, but the only vertex should be a
// biconnected component.
 {
    dfn[u] = low[u] = num++;
    for ( each vertex w adjacent from u ) {
       if (v != w && dfn[w] < dfn[u])
            // w is not a direct parent of u, see comments later
                          add (u,w) to stack S;
```

```
if ( dfn[w] == 0 ) {  // w is an unvisited vertex
      Biconnected (w, u);
      low[u] = min( low[u], low[w]);
      if ( low[w] >= dfn[u] ) {
          cout<<“New Biconnected Component: ” <<endl;
          do {
              delete an edge from stack S;
              let this edge be (x, y);
              cout << x << "," << y <<endl;
           } while ( (x, y) and (u, w) are not the same edge );
      }
   }
   else if ( w != v ) low[u] = min( low[u], dfn[w]); //back edge
 }
}
```

Comment:

If (dfn[w] > dfn[u]) the edge (u, w) must have been added to the stack S as a back edge, as shown below:



When at w, (w, u) has been added to the stack. When back to u, (u, w) should not be added to the stack.

The time complexity of Biconnected is O(n+e).

Exercises: P393-10

6.3 Minimum-Cost Spanning Trees

The cost of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in the spanning tree.

A minimum-cost spanning tree is a spanning tree of least cost.

A design strategy called **greedy method** can be used to obtain a minimum-cost spanning tree.

In the greedy method,

- construct an optimal solution in stages;
- at each stage, make a decision (using some criterion) that appears to be the best (local optimum);
- since the decision can't be changed later, make sure it will result in a feasible solution, i.e., satisfying the constraints.

At the end, if the local optimum is equal to the global optimum, then the algorithm is correct; otherwise, it has produced a suboptimal solution.

Fortunately, in the case of constructing minimum-cost spanning tree, the method is correct.

To construct minimum-cost spanning tree, we use a least-cost criterion and the constraints are:

(1) must use only edges within G.

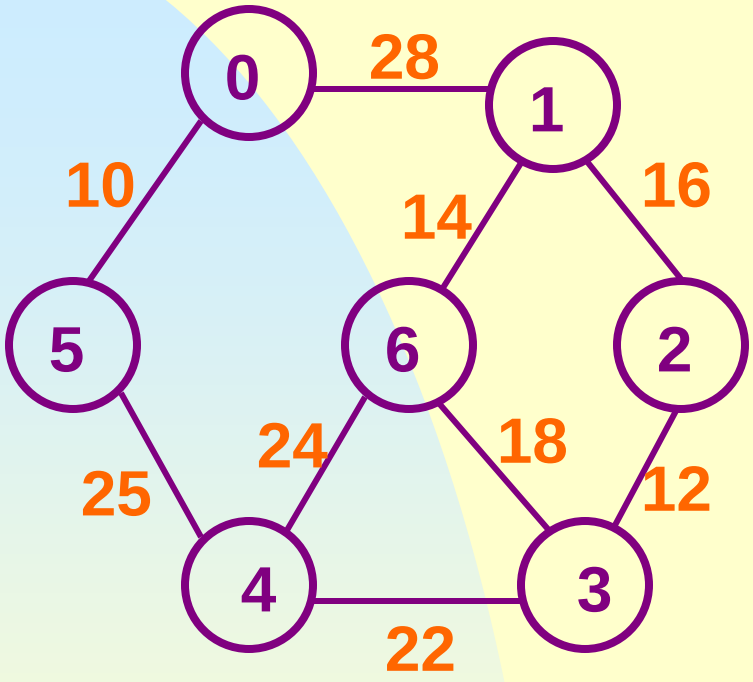
(2) must use exactly n-1 edges.

(3) may not use edges that produce a cycle.

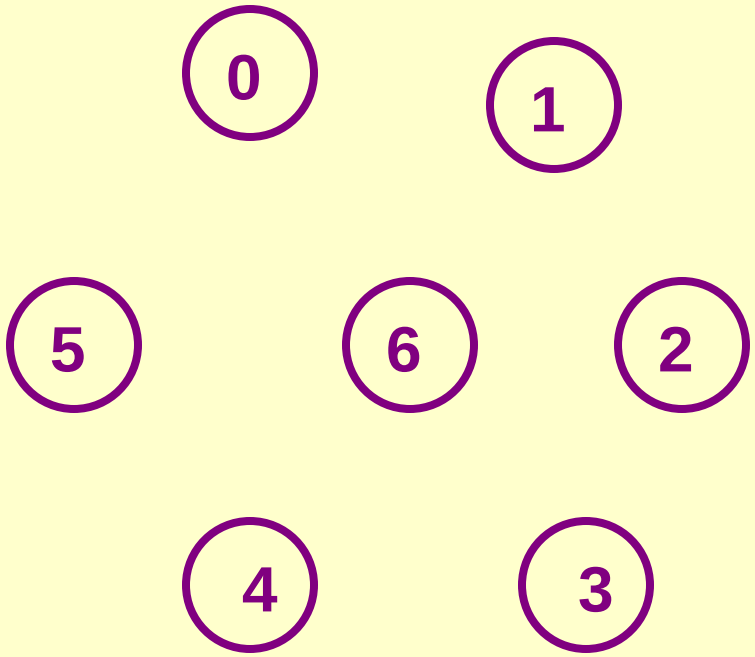
6.3.1 Kruskal Algorithm

- The minimum-cost spanning tree T is built by adding edges to T one at a time.
- Edges are selected for inclusion in T in non-decreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges already in T.
- Exactly n-1 edges will be selected into T.

Example 6.4:

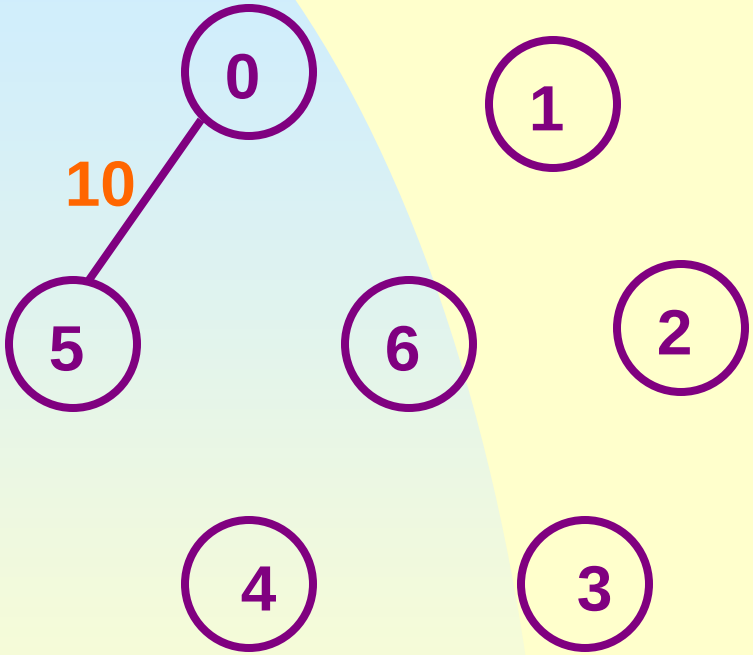


(a) G

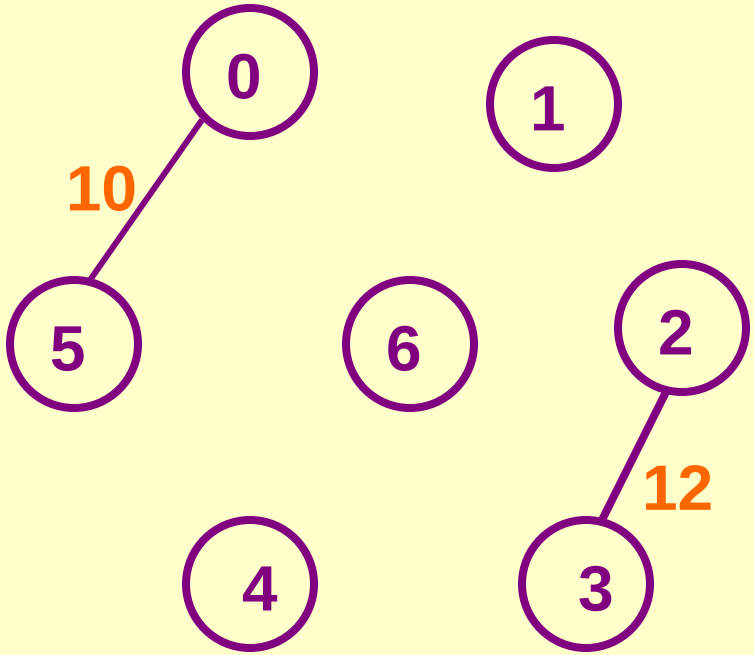


(b)

The cost of remaining edges: 10, 12, 14, 16, 18, 22, 24, 25, 28

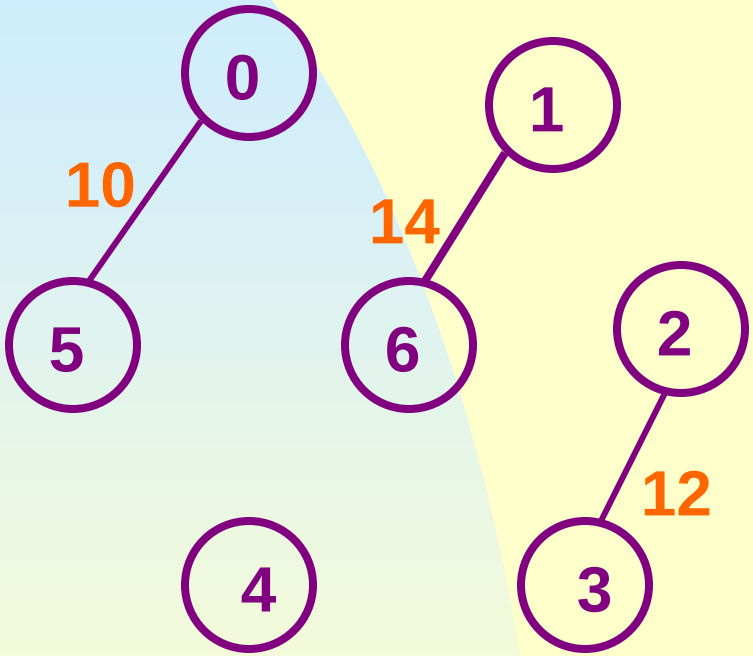


(c)

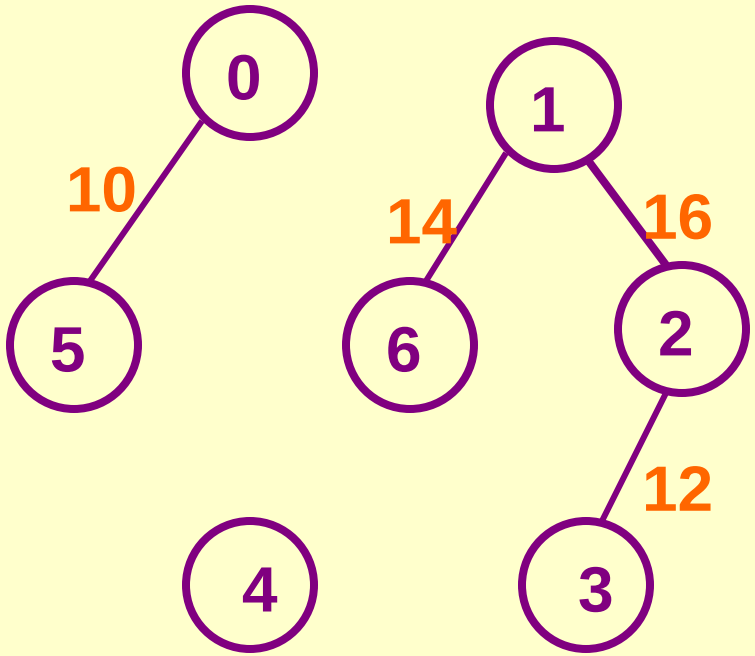


(d)

The cost of remaining edges: 14, 16, 18, 22, 24, 25, 28

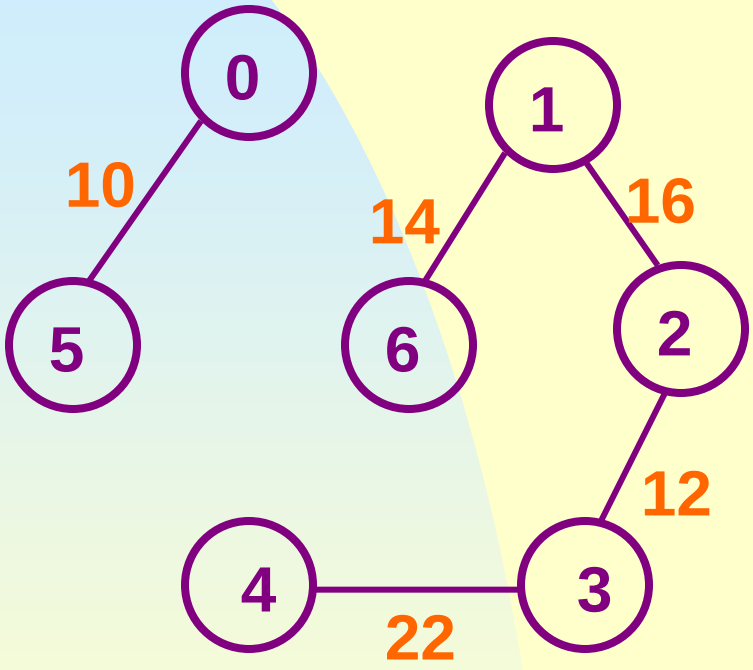


(e)

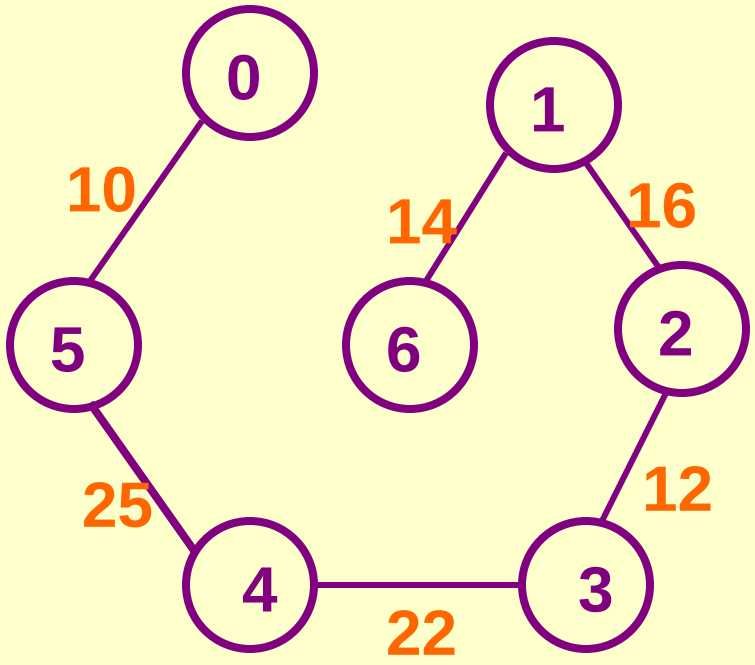


(f)

Remaining edges: 18—(3,6), 22, 24—(4,6), 25, 28



(g) (3,6) discarded



(h) (4, 6) discarded

Given G = (V, E), we have the algorithm:

```
1  T = ∅;
2  while ((T contains less than n-1 edges) && (E not Empty)) {
3      choose an edge (v, w) from E of lowest cost;
4      delete (v, w) from E;
5      if ((v, w) does not create a cycle in T) add (v, w) to T;
6      else discard (v, w);
7  }
8  if (T contains fewer than n-1 edges)
                          cout<<"no spanning tree"<<endl;
```

Analysis:

- **To perform lines 3 and 4, E can be organized as a min heap, so the next edge can be chosen and deleted in O(log e). Initialization of the heap takes O(e) (ref. 7.6).**
- **To perform line 5, vertices in T can be placed into a set using Union-Find. The total cost is O(e. α(e)).**

The total cost: O(e log e).

Correctness:

Theorem 6.1: Let G be any undirected, connected graph, Kruskal's algorithm generates a minimum-cost spanning tree.

The proof is left as a self-study exercise.

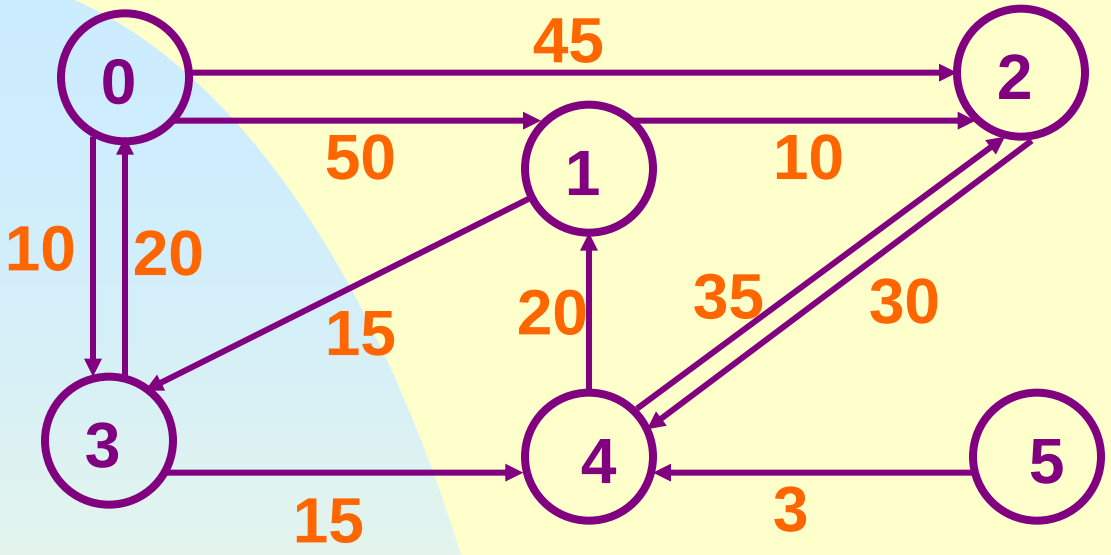
Exercises: P359-1

6.4 Shortest Paths

6.4.1 Single Source/All Destinations: Nonnegative Edge Cost

Problem: given a digraph G=(V, E), a length function length(i, j)≥0 for <i, j>∈E(G), and a source vertex v, to determine the shortest path from v to all remaining vertices of G.

Example:



(a) G

	path	Length
1)	0, 3	10
2)	0, 3, 4	25
3)	0, 3, 4, 1	45
4)	0, 2	45

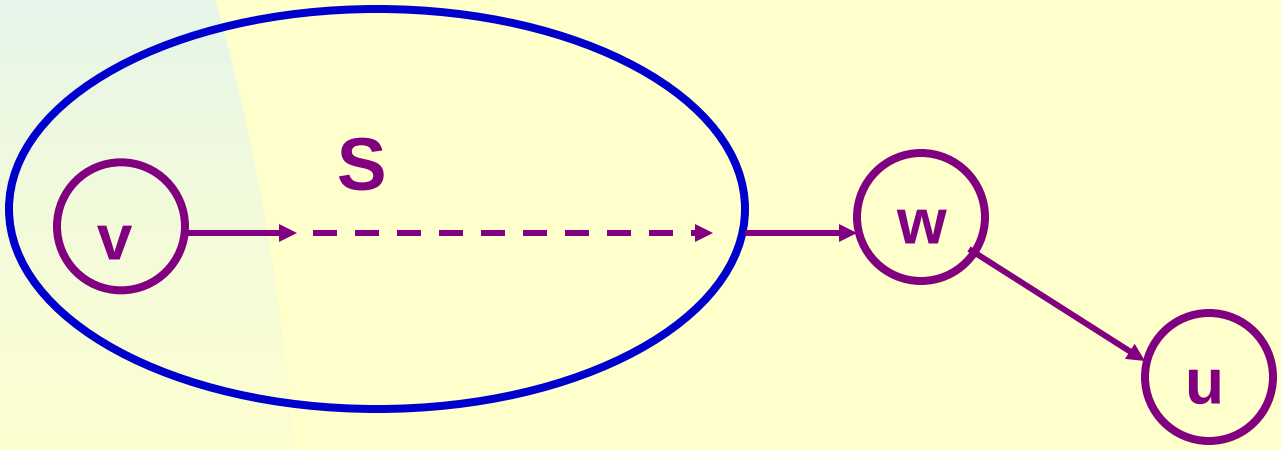
(b) Shortest paths from 0

Let S --- the set of vertices (including v) to which the shortest paths have been found.

For w $\notin$ S, dist[w] --- the length of the shortest path starting from v, going through only the vertices in S, and ending at w.

Assume paths are generated in non-decreasing order of length, observe:

(1) If the next shortest path is to u, then it begins at v, ends at u, and goes through only vertices in S. To prove, assume w on this path is not in S, then the v to u path contains a path from v to w as shown below:

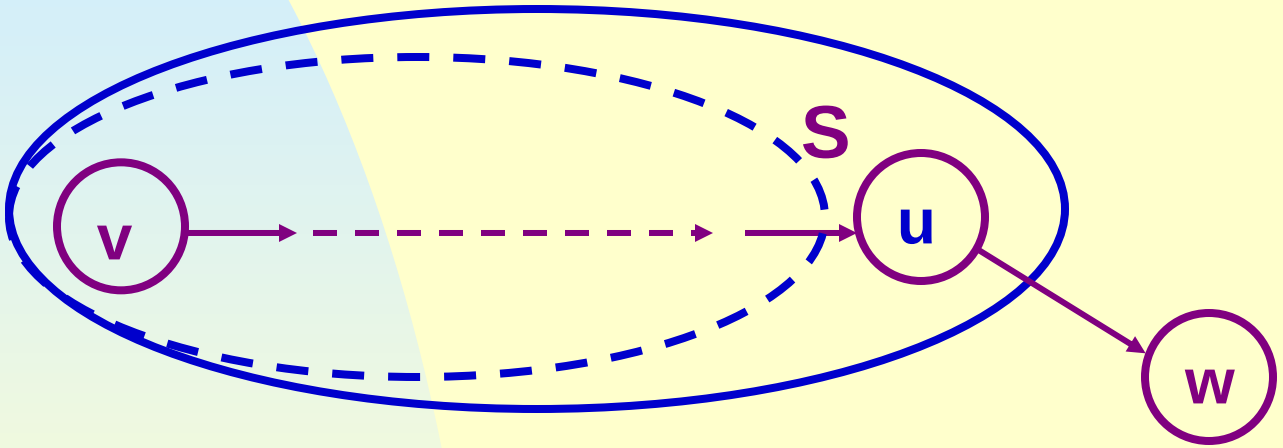


As length(w, u)≥0, length(v---w) must be less than length(v---u), otherwise we don't need to go through w. By assumption, the shorter path(v---w) has been generated already. Hence there is no intermediate vertex that is not in S.

(2) The destination of the next path generated must be u, such that

$$dist[u] = \min_{v \notin S}\{ dist[v]\}$$

(3) Having selected a vertex u as in (2), u becomes a member of S. Now dist[w], w $\notin$S, may change. If it does, it must be due to a shorter path from v to u and then to w.



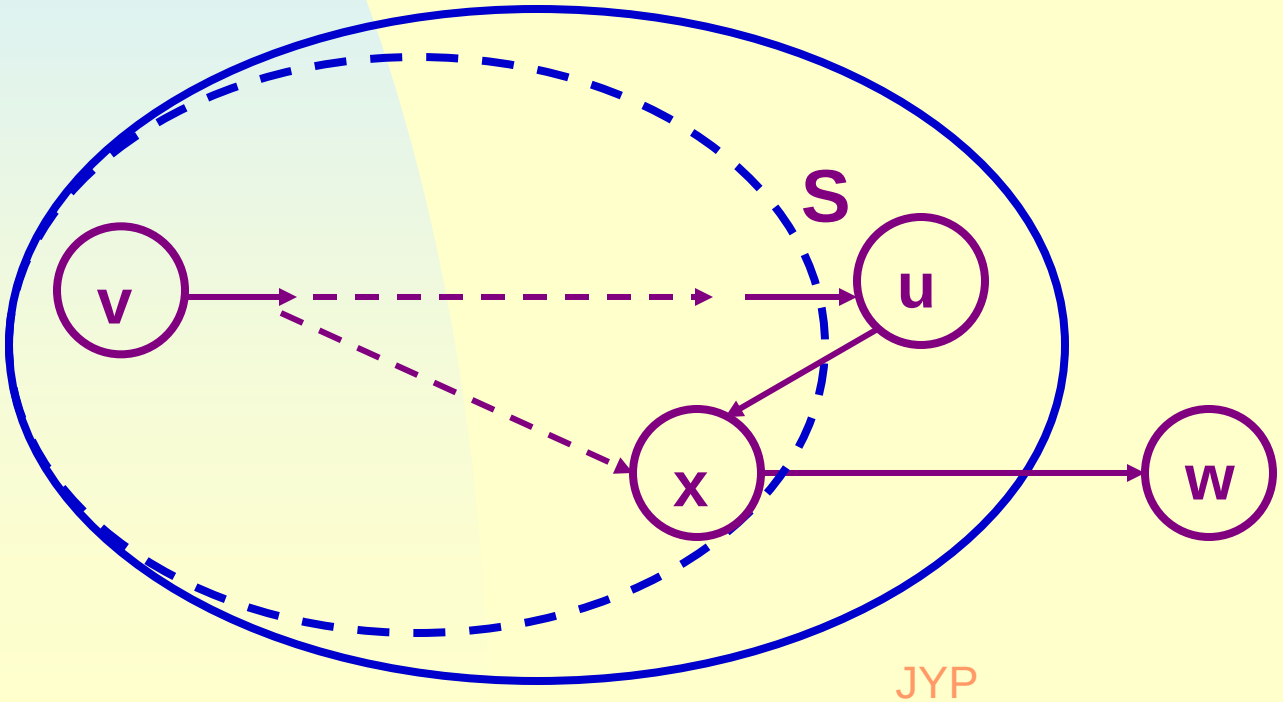
The v to u path must be the shortest v to u path, and the u to w path is just the edge <u, w>. So if dist[w] is to decrease, it is because the current dist[w]>dist[u]+length(u, w).

Note the u to w path with any intermediate vertex x∈S cannot affect dist[w]. Because:

① length(v-u-x)>dist[x]

② length(v-u-x-w)>length(v-x-w)≥dist[w]

As shown below:



Assume:

- **n vertices numbered 0, 1, …, n-1.**
- **s[i]=false if i $\notin$S, s[i]=true if i $\in$S.**
- **length[i][j]. If <i, j> $\notin$E(G) and i$\neq$j, set length[i][j] to some large number.**

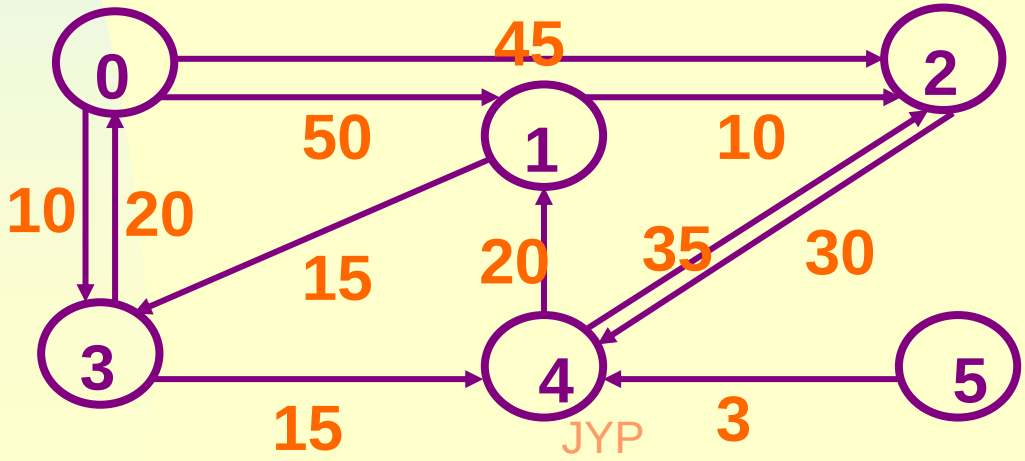
```
class MatrixWDigraph {
private:
    double length[NMAX][NMAX];  // NMAX is a constant
    double *dist;
    bool *s;
public:
    void ShortestPath(const int, const int);
    int choose(const int);
};
```

And we assume the constructor will apply space for array dist and s.

```
1 void MatrixDigraph::ShortestPath(const int n,  const int v)
2 {//dist[j],0≤j<n, is set to the length of the shortest path(v-j)
3 //in a digraph G with n vertices and edge lengths in length[i][j]
4     for (int i=0; i<n; i++) { s[i]=false; dist[i]=length[v][i];}
5     s[v]=true;
6     dist[v]=0;
7     for (i=0; i<n-2; i++) { //determine n-1 paths from v
8        int u=choose(n); //choose returns a value u such that
9                         //dist[u]=minimum dist[w], where s[w]=false
10     s[u]= true;
11     for ( int w=0; w<n; w++)
12       if (!s[w] && dist[u]+length[u][w]<dist[w])
13          dist[w]=dist[u]+length[u][w];
14   }
15}
```

The running status of ShortestPath for the example:

vertex	dist					
	0	1	2	3	4	5
0	0	50	45	10	+∞	+∞
3	0	50	45	10	25	+∞
4	0	45	45	10	25	+∞
1	0	45	45	10	25	+∞
2	0	45	45	10	25	+∞

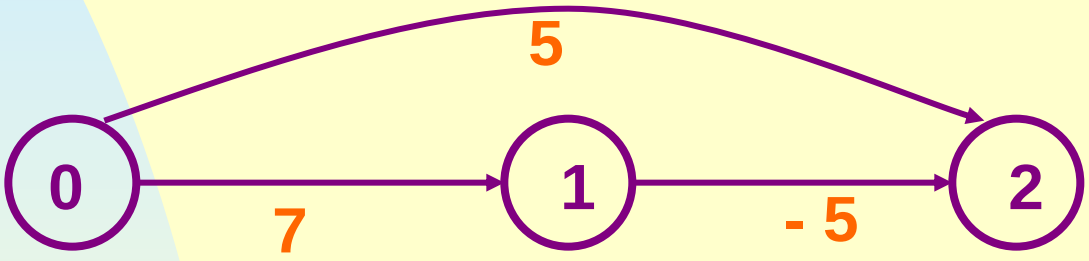


Analysis of ShortestPath:

The time for loop of line 4 is O(n). The for loop of line 7 is executed n-2 times, each requires O(n) at line 8 and at line 11 to 13. So the total time is (n^2).

Even if adjacency lists are used, the overall time for line 11 to 13 can be brought down to O(e), but for line 8 remains $O(n^2)$.

Note the algorithm does not work when some edges may have negative length, as shown bellow:



6.4.3 All-Pairs Shortest Paths

Problem: find the shortest paths between all pairs of vertices u and v, $u \neq v$.

One solution:

For each vertex in G, takes it as the source, apply ShortestPath, all n times, in $O(n^3)$.

Using the dynamic programming approach, we can obtain a conceptually simpler algorithm that has complexity of $O(n^3)$ and works even when G has edges with negative length so long as G has no cycles with negative length.

$A^k[i][j]$ --- the length of shortest path from i to j going through no intermediate vertex of index greater than k.

$A^{n-1}[i][j]$ --- the length of the shortest i--j path in G.

$A^{-1}[i][j]$ --- length[i][j].

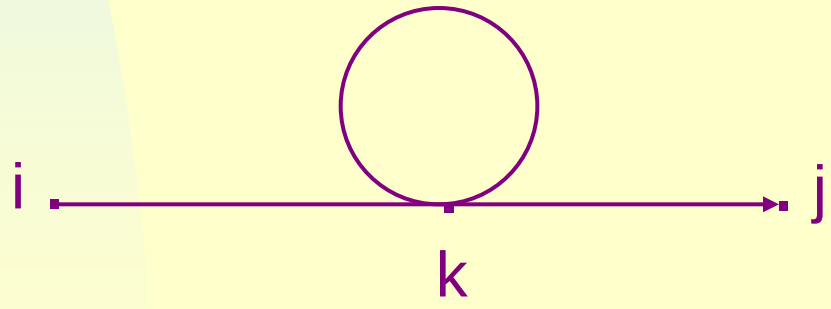
Now the basic idea of the algorithm:

Successively generate A^{-1}, A^0, A^1,…, A^{n-1}. Given A^{k-1}, we can get A^k by realizing for every pair of i and j, either of the following 2 cases applies:

(1) The shortest path from i to j going through no vertex with index greater than k does not go through k, so its length is $A^{k-1}[i][j]$.

(2) The shortest path does go through k. The path consists of a subpath from i to k and another one from k to j. These must be the shortest paths from i to k and from k to j going through no vertex with index greater than k-1, so their lengths are $A^{k-1}[i][k]$ and $A^{k-1}[k][j]$.

Note the above is true only if G has no negative cycle containing k.



From (1) and (2), we have:

$$A^k[i][j]=\min\{A^{k-1}[i][j], A^{k-1}[i][k]+ A^{k-1}[k][j]\}, k\geq 0.$$

Assume:

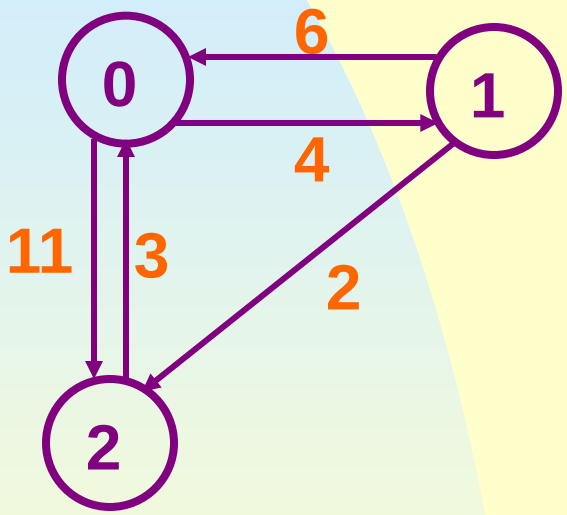
- G is represented by length-adjacency matrix as for ShortestPath.
- double a[NMAX][NMAX] is data member of MatrixDigraph.

```
1 void MatrixDigraph::AllLengths (const int n)
2 { //length[n][n] is the adjacency matrix of G with n vertices.
3   //a[i][j] is the length of shortest path from i to j
4      for (int i=0; i<n; i++)
5         for (int j=0; j<n; j++)
6               a[i][j] = length[i][j]; // copy length into a
7      for ( int k=0; k<n; k++) // for a path with highest index k
8         for (i=0; i<n; i++)   // for all pairs
9            for (j=0; j<n; j++)
10              if (a[i][k]+a[k][j]<a[i][j]) a[i][j]=a[i][k]+a[k][j];
11}
```

The computation is done in place using a, because $A^k[i][k] = A^{k-1}[i][k]$ and $A^k[k][j] = A^{k-1}[k][j]$.

The time is obviously $O(n^3)$.

Example 6.7:



A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

Exercises: P372-1, P373-2, 5, P375-17

6.5 Activity Networks

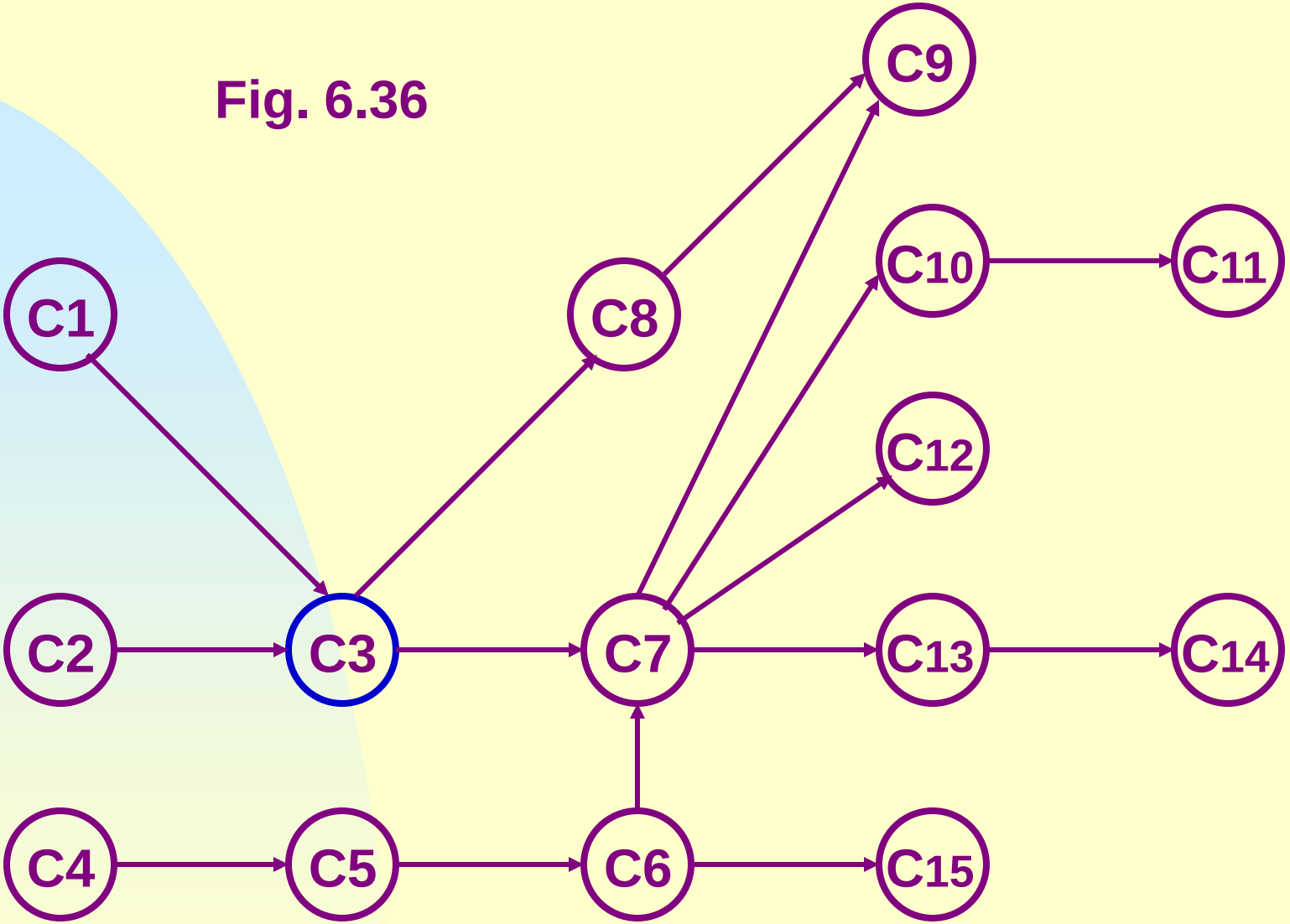
6.5.1 Activity-on-Vertex (AOV) Networks

Definition: A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between tasks is an activity-on-vertex network or AOV network.

In the following, we'll see the AOV network corresponding to the courses of the next slide.

Course-No.	Course-Name	Prerequisites
C1	Programming Ⅰ	None
C2	Discrete Mathematics	none
C3	Data Structures	C1, C2
C4	Calculus Ⅰ	none
C5	Calculus Ⅱ	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating System	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithm	C13
C15	Numerical Analysis	C5

Fig. 6.36



Definition: Vertex i in an AOV network G is a predecessor of j iff there is a directed path from i to j. If <i, j> is an edge in G then i is an immediate predecessor of j and j immediate successor of i.

Definition: A precedence relation that is both transitive and irreflexive is a partial order.

A directed graph with no cycle is an acyclic graph.

Given a AOV network G, we are concerned with determining whether or not it is irreflexive, i.e., acyclic.

And we need to generate the topological order of vertices in G.

Definition: A topological order is a linear ordering of vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network, then i precedes j in the linear ordering.

Two possible topological orders of Fig. 6.36 are:

C1,C2,C4,C5,C3,C6,C8,C7,C10,C13,C12,C14,C15,C11,C9

and

C4,C5,C2,C1,C6,C3,C8,C15,C7,C9,C10,C11,C12,C13,C14

Topological sorting algorithm

Idea: list a vertex with no predecessor, then delete this vertex together with all edges leading out of it from the network. Repeat the above until all vertices have been listed or all remaining vertices have predecessors.

Formally,

```
1  Input the AOV network, let n be the number of vertices;
2  for (int i=0; i<n; i++) // output the vertices
3 {
4    if (every vertex has a predecessor) return;
5       // network has a cycle and is infeasible.
6    pick a vertex v that has no predecessors;
7    cout << v;
8    delete v and all edges leading out of v from the network;
9 }
```

Example:

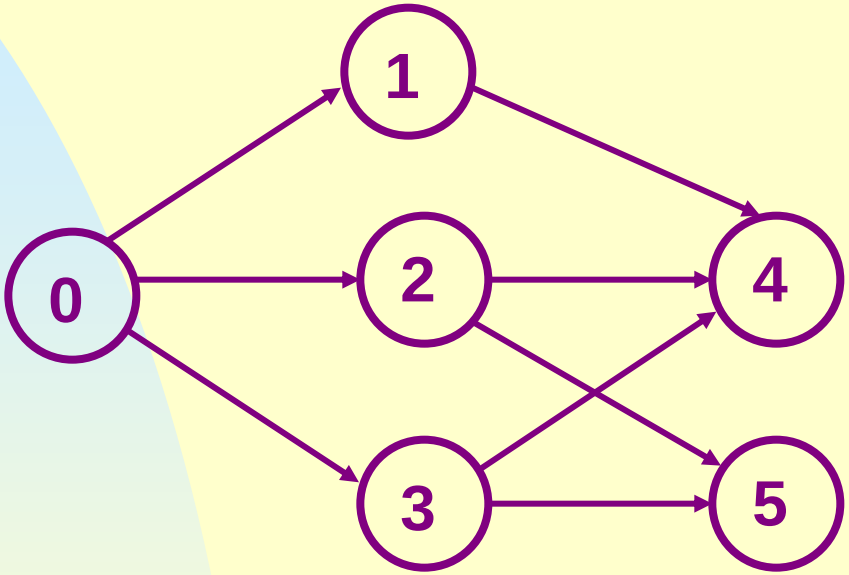
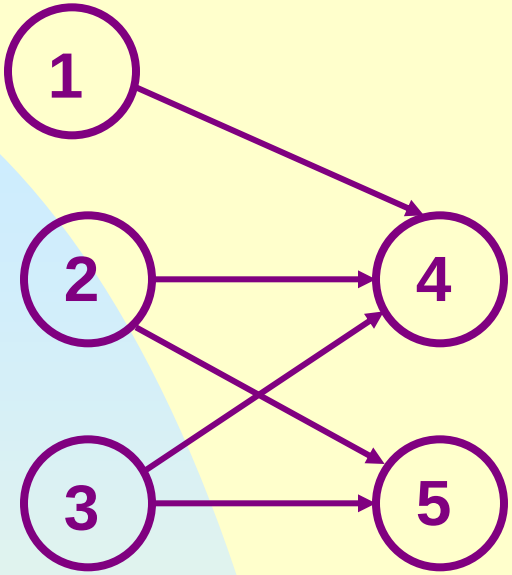
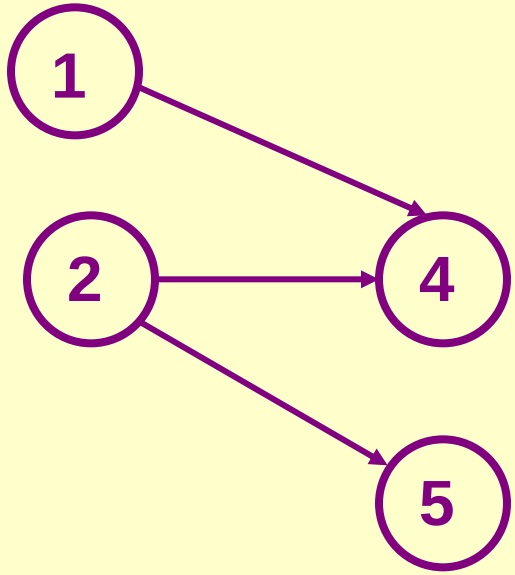


Fig. 6.37

(a) Initial

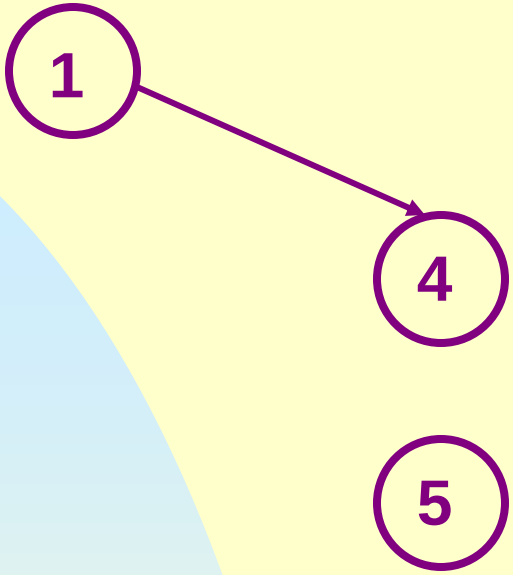


(b) Vertex 0 deleted

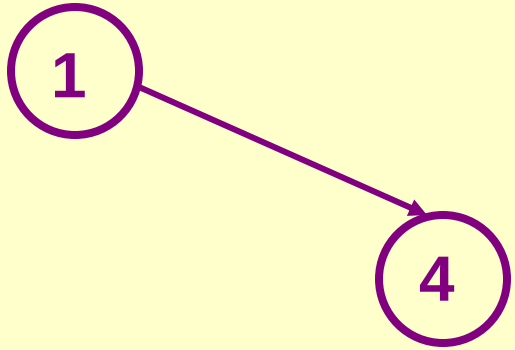


(c) Vertex 3 deleted

Topological order generated: 0, 3,



(d) Vertex 2 deleted



(e) Vertex 5 deleted

(f) Vertex 1 deleted

Topological order generated: 0, 3, 2, 5, 1, 4

The data representation depends on:

(1) Whether a vertex has a predecessor?--- maintain a count of the number of immediate predecessors of each vertex.

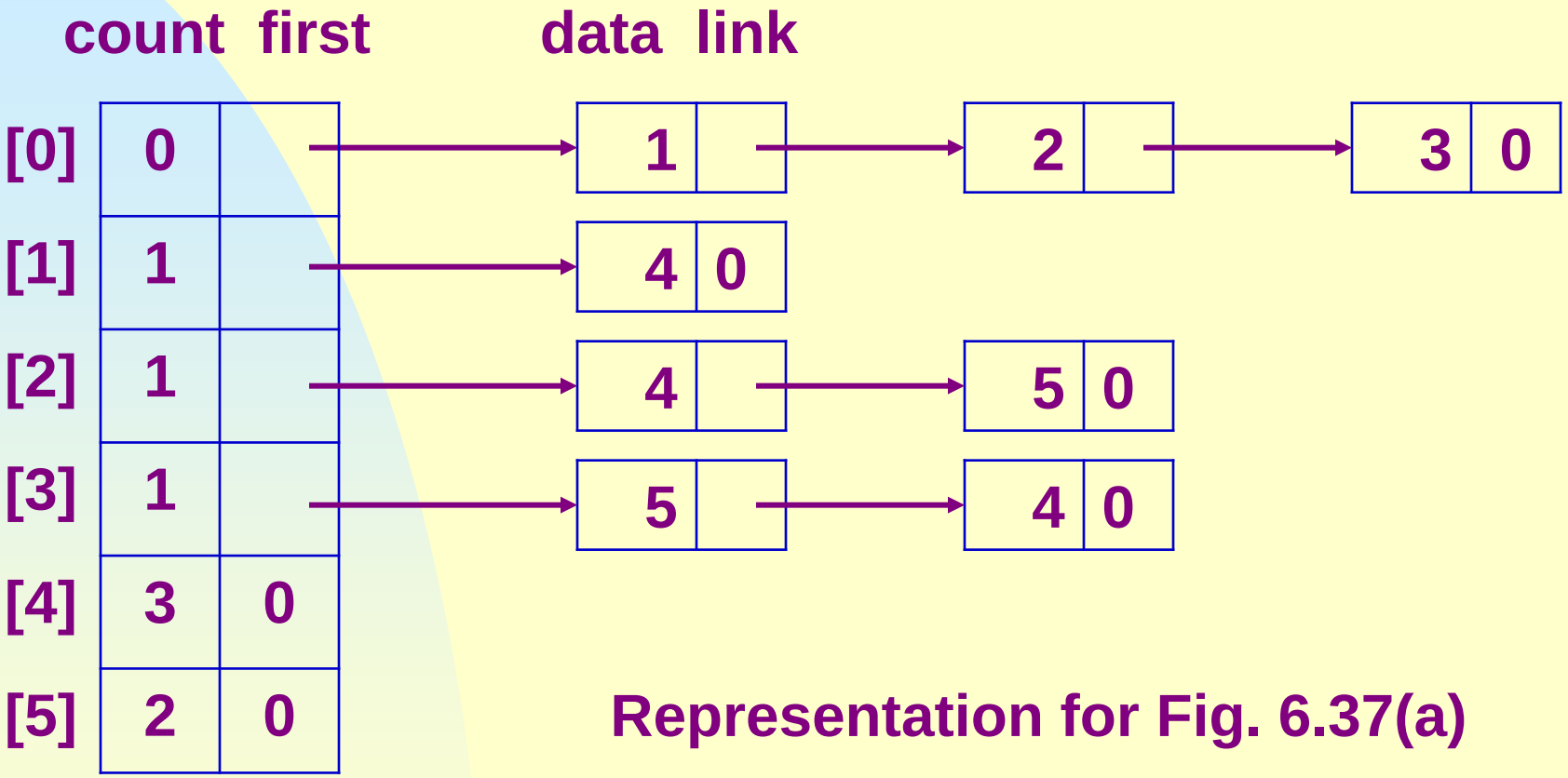
(2) How to delete a vertex together with all its incident edges?--- adjacency lists, and decrease the count of all vertices on its adjacency list.

Whenever the count of a vertex drops to 0, that vertex can be placed onto a list of vertices with 0 count.

Hence, we use adjacency lists representation, and assume:

- int *count is defined as a data member, and its space is allocated in the constructor.
- count[i], 0≤i<n, has been initialized to the in-degree of vertex i. When <i, j> is input, the count of j is increased by 1.
- the list of vertices with 0 count is maintained as a custom stack linked through the count field since it is of no use after the count has become 0.

- **int *t is defined as a data member, and its space is allocated in the constructor. Vertices are stored in t in topological order for future use.**



Representation for Fig. 6.37(a)

```
1 void LinkedGraph::TopologicalOrder()
2 { //The n vertices of a network are listed in topological order
3   int top = -1, pos = 0;
4   for (int i=0; i<n; i++) //create a linked stack of vertices with
5     if (count[i]==0) { count[i]=top;  top=i;} //no predecessors
6   for (i=0; i<n; i++)
7     if (top==-1) throw “network has a cycle.";
8     int j=top;  top=count[top]; //unstack a vertex
9     t[pos++] = j;  // store vertex j in topological order
10    Chain<int>::ChainIterator ji=adjLists[j].begin();
11    while (ji != adjLists[j].end()) { // decrease the count of
12        count[*ji]--;                 // the successor vertices of j
13        if (count[*ji]==0) {count[*ji]=top; top=*ji;} //add to stack
14        ji++; // next successor
15    }
16}
```

Analysis of TopologicalOrder:

Very efficient because of a judicious choice of data structure.

- Lines 4-5 loop: O(n)
- Lines 6-10 totally O(n)
- The while loop of 11-15 takes $O(d_i)$ for each vertex i, where d_i is the out-degree of i. The total time for this part is $O((\sum_{i=0}^{n-1} d_i)+n)=O(e+n)$.

The total time: O(e+n).

6.5.2 Activity-on-Edge (AOE) Networks

The activity-on-edge (AOE) network:

- **directed edges --- tasks to be performed**
- **vertices --- events, signaling the completion of certain activities.**
- **activities represented by edges leaving a vertex cannot be started until the event at that vertex has occurred.**
- **an event occurs only when all activities entering it have been completed.**

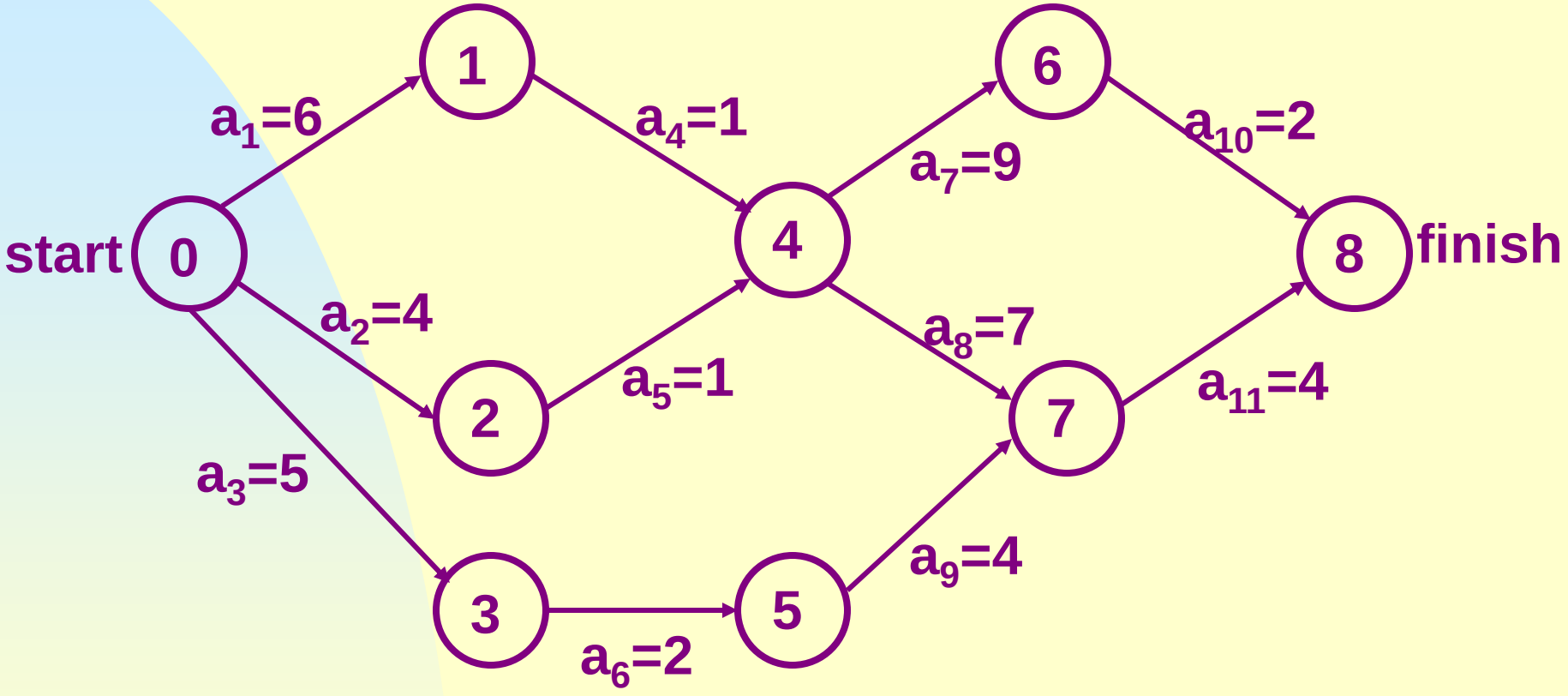

start
0
1
2
3
4
5
6
7
8
finish
a_1=6
a_2=4
a_3=5
a_4=1
a_5=1
a_6=2
a_7=9
a_8=7
a_9=4
a_{10}=2
a_{11}=4

Fig. 6.39

- Since activities in an AOE network can be carried out in parallel, the minimum time to complete the project is the length of the longest path from the start to the finish.
- A path of longest length is a critical path. e.g., the paths 0, 1, 4, 6, 8 (length 18) and 0, 1, 4, 7, 8.

- **e(i) --- the earliest start time for activity a_i.**
- **l(i) --- the latest start time for activity a_i without increasing the project duration.**
- **If e(i) = l(i), a_i is called a critical activity. l(i) - e(i) is a measure of the criticality of a_i .**

- **Speed up a critical activity will not necessarily result in a reduced project length unless it is on all critical paths. e.g., speed up a_{11} will not reduce the project length, but a_1 will.**
- **Once we get e(i) and l(i), critical activities can be easily identified.**

6.5.2.1 Calculation of Early Activity Times

How to obtain e(i) and l(i)? It is easy to first get:

ee[j] --- the earliest time for event j.

le[j] --- the latest time for event j.

If a_i is edge <k, l>, then

e(i) = ee[k] and

l(i) = le[l] – duration of activity a_i } **(6.1)**

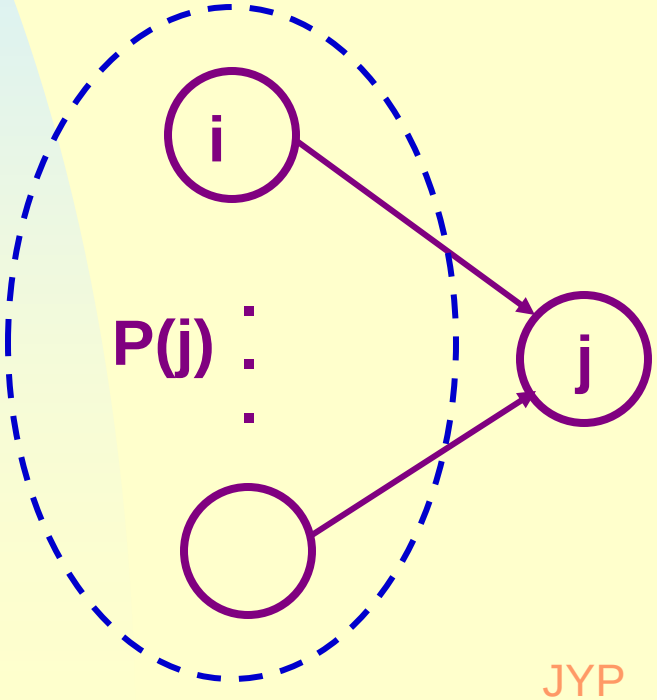
ee[j] and le[j] are computed in 2 stages: a forward stage and a backward stage.

The forward stage:

$ee[0]=0$ (suppose 0 is the start)

$$ee[j]=\max_{i \in P(j)}\{ee[i]+\text{duration of } <i, j>\} \qquad (6.2)$$

where P(j) is the set of all vertices adjacent to j.



To carry the information of duration, we use the structure:

```
struct Pair
{
   int  vertex;
   int  dur;       //activity duration
};

class LInkedGraph {
private:
   Chain<Pair> *adjLists;
   int *count, *t, *ee, *le;
   int n;
```

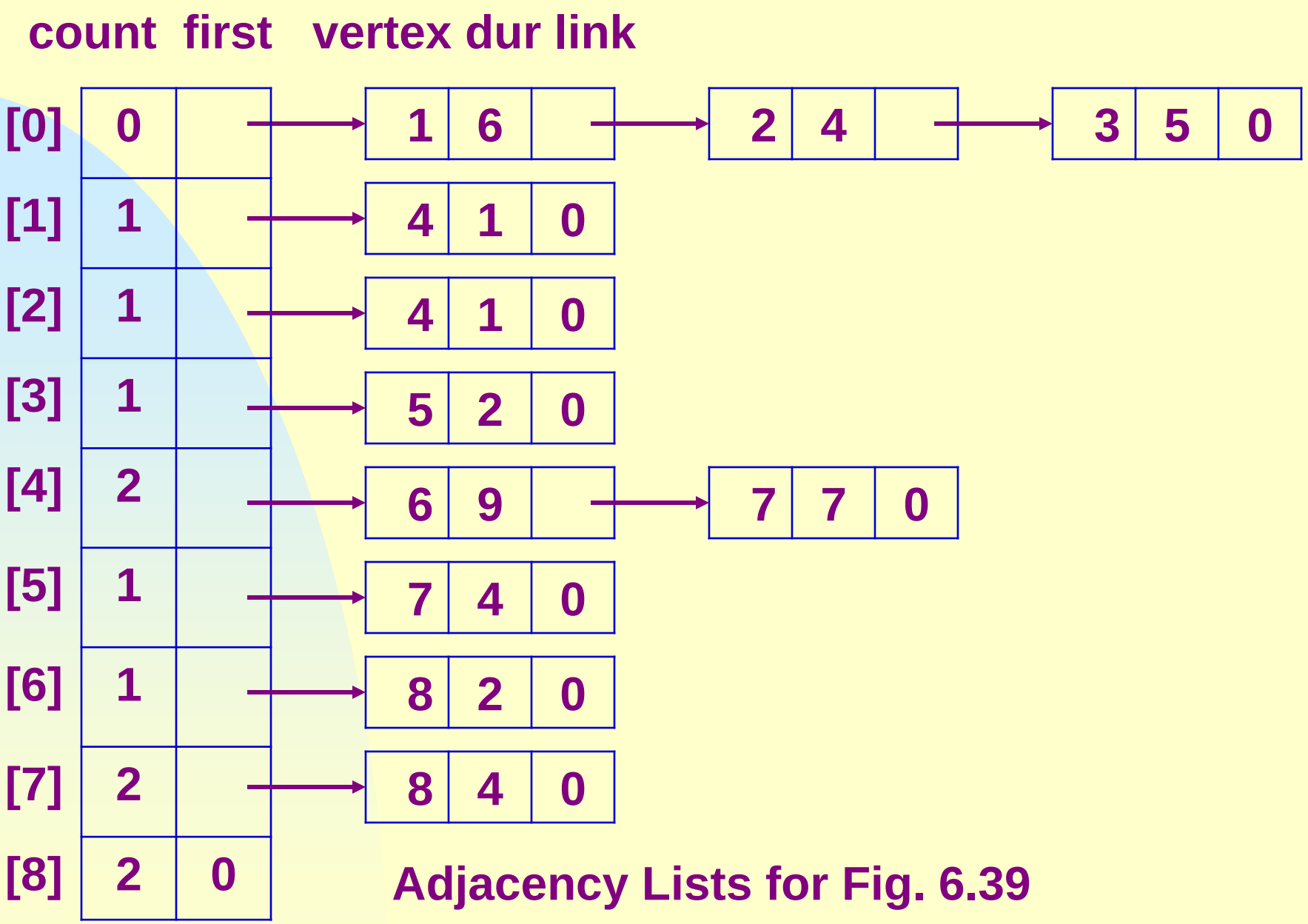
```
public:
   LinkedGraph (const int vertices) : {
      if (vertices < 1) throw "Number of vertices must be > 0";
      n = vertices;
      adjLists = new Chain<Pair>[n];
      count = new int[n]; t = new int[n];
      ee = new int[n]; le = new int[n];
   };
  void TopologicalOrder();
  void EarliestEventTime();
  void LatestEventTime();
  void CriticalActivities();
};
```

If we initialize array ee to 0, and compute in topological order, then the early start times of all predecessors of j would have been computed prior to the computation of ee[j].

Thus we have:

```
void LinkedGraph::EarliestEventTime()
{ // assume a topological order has already been in t,
  // compute ee[j] according to t
   fill(ee, ee+n, 0);   // initialize ee
   for (i=0; i<n-1; i++) {
      int j=t[i];
      Chain<Pair>::ChainIterator ji=adjLists[j].begin();
      while (ji!=adjLists[j].end()) {
         int k=ji→vertex; //k is successor of j
         if (ee[k]<ee[j]+ji→dur) ee[k]=ee[j]+ji→dur;
         ji++;
      }
}
```

To illustrate, let's try it out on the network of Fig. 6.39.



Adjacency Lists for Fig. 6.39

ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
initial	0	0	0	0	0	0	0	0	0
do 0	0	6	4	5	0	0	0	0	0
do 3	0	6	4	5	0	7	0	0	0
do 5	0	6	4	5	0	7	0	11	0
do 2	0	6	4	5	5	7	0	11	0
do 1	0	6	4	5	7	7	0	11	0
do 4	0	6	4	5	7	7	16	14	0
do 7	0	6	4	5	7	7	16	14	18
do 6	0	6	4	5	7	7	16	14	18

Fig. 6.40

The computing time is obviously O(n+e).

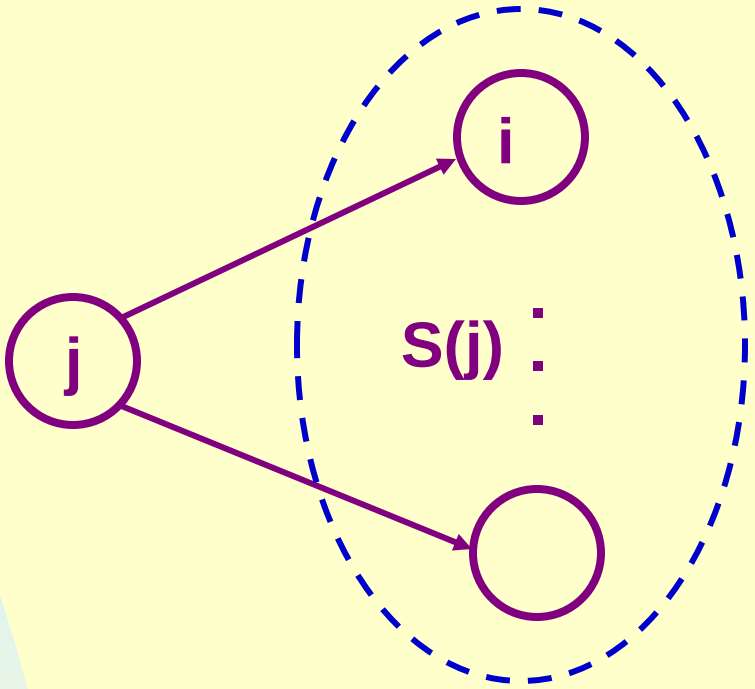
6.5.2.2 Calculation of Late Activity Times

The backward stage:

$$le[n-1]=ee[n-1] \quad \text{(suppose n-1 is the finish)}$$

$$le[j]= \min_{i \in S(j)} \{le[i]-\text{duration of } <j, i>\} \qquad (6.3)$$

where S(j) is the set of all vertices adjacent from j.



If the forward stage has already been done, and a topological order get, then le[i], 0≤i<n, can be computed directly, using (6.3), by performing the computations in the reverse topological order.

Hence we have:

```
void LinkedGraph::LatestEventTime()
{ // assume a topological order has already been in t, ee has
  // been computed, compute le[j] in the reverse order of t
   fill(le, le+n, ee[n-1]); // initialize le
   for (i=n-2; i>=0; i--) {
      int j=t[i];
      Chain<Pair>::ChainIterator ji=adjLists[j].begin();
      while (ji!=adjLists[j].end()) {
         int k=ji→vertex; //k is successor of j
         if (le[k]-ji→dur<le[j]) le[j]=le[k]-ji→dur;
         ji++;
      }
}
```

Example:

given the topological order of Fig. 6.39:

0, 3, 5, 2, 1, 4, 7, 6, 8

we may compute le[i] for i=8, 6, 7, 4, 1, 2, 5, 3, 0.

le[8] = ee[8] = 18

le[6] = min {le[8] – 2} = 16

le[7] = min {le[8] – 4} = 14

le[4] = min {le[6] – 9, le[7] – 7} = 7

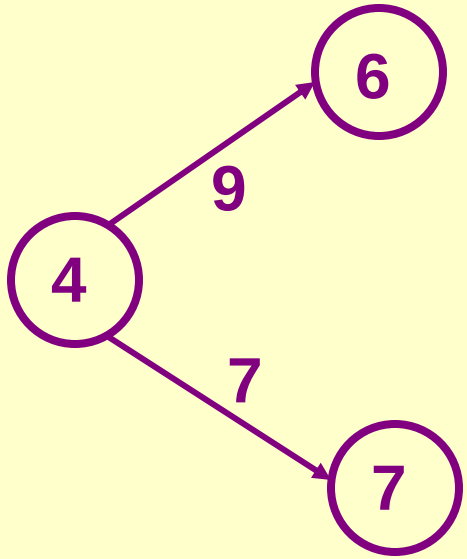
le[1] = min {le[4] – 1} = 6

le[2] = min {le[4] – 1} = 6

le[5] = min {le[7] – 4} = 10

le[3] = min {le[5] – 2} = 8

le[0] = min {le[1]–6, le[2]–4, le[3]–6} = 0



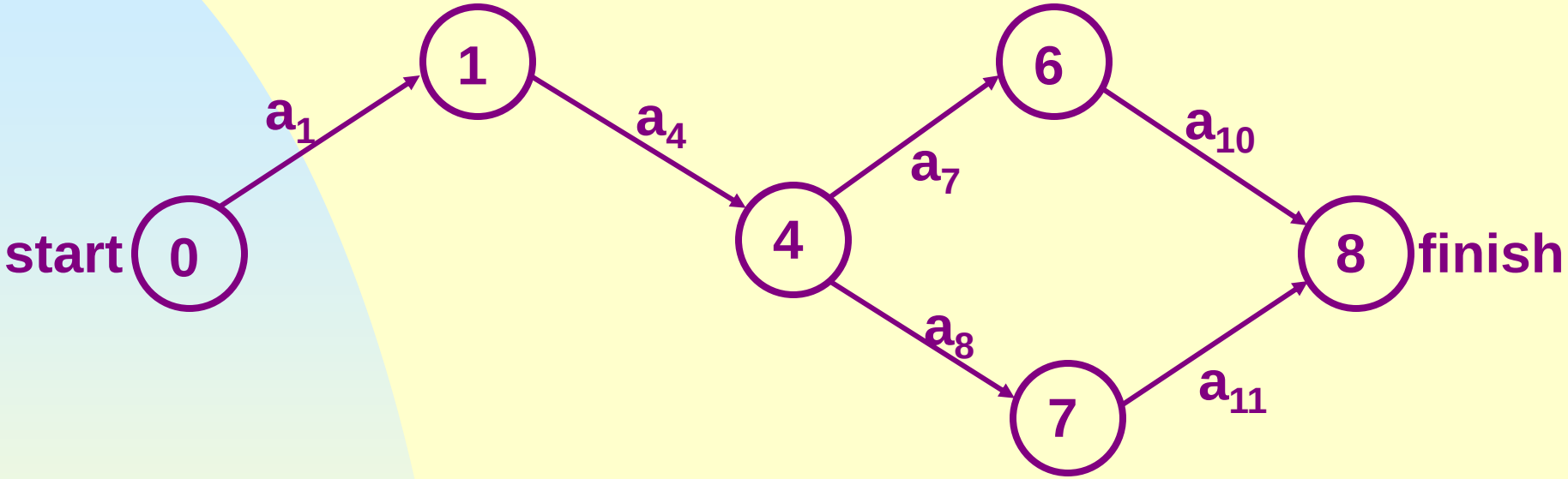
Now we can use (6.1) to output critical activities:

```
void LinkedGraph::CriticalActivities()
{ // compute e[i] and l[i], output critical activities
   int i=1; // the numbering counter for activities
   int u, v, e, l; // e, l are the earliest, latest start time of <u, v>
   for (u=0; u<n; u++) {  // scan the adjacency lists.
      Chain<Pair>::ChainIterator ui=adjLists[u].begin();
      while (ui!=adjLists[u].end()) {
         int v=ui→vertex; // <u, v> is an edge numbered i
         e=ee[u]; l=le[v]-ui→dur;
         if (l==e) cout <<"a"<<i<<"<"<<u<<","<<v<<">"
                        <<"is a critical activity"<<endl;
         ui++; i++;
      }
   }
}
```

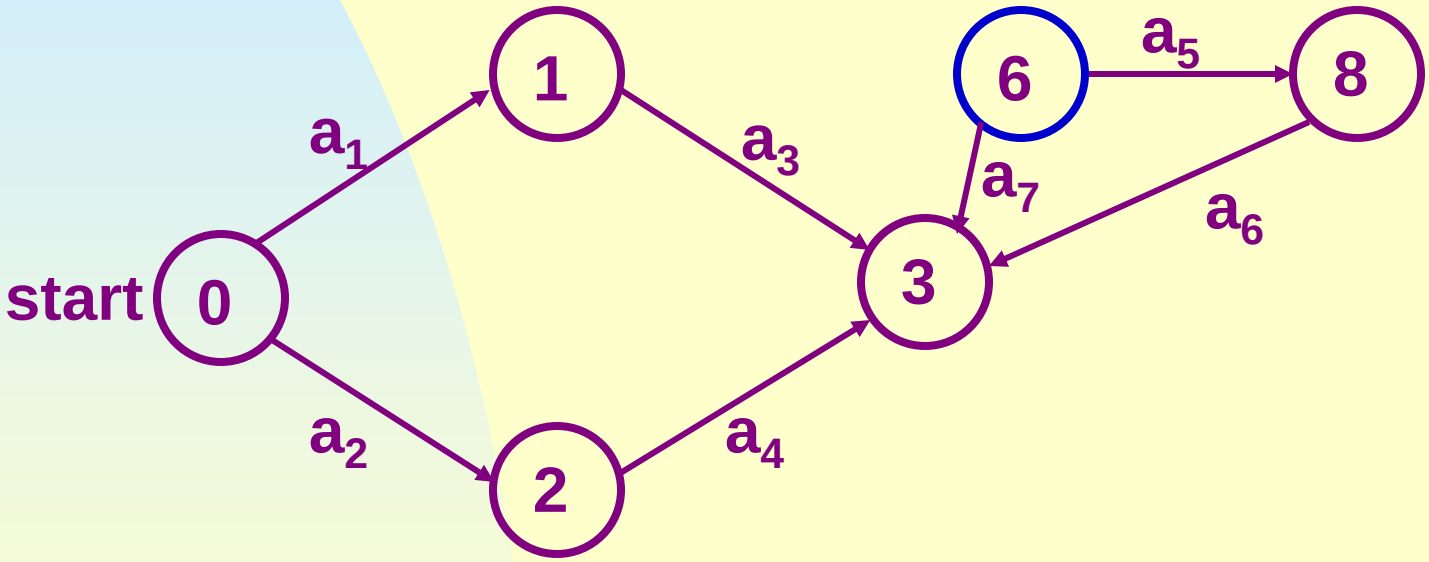
For Fig. 6.39, the e(i) and l(i) are:

activity	e	l	l-e	l-e=0
a_1 <0, 1>	0	0	0	Y
a_2 <0, 2>	0	2	2	N
a_3 <0, 3>	0	3	3	N
a_4 <1, 4>	6	6	0	Y
a_5 <2, 4>	4	6	2	N
a_6 <3, 5>	5	8	3	N
a_7 <4, 6>	7	7	0	Y
a_8 <4, 7>	7	7	0	Y
a_9 <5, 7>	7	10	3	N
a_{10} <6, 8>	16	16	0	Y
a_{11} <7, 8>	14	14	0	Y

The critical activities: a_1, a_4, a_7, a_8, a_{10}, and a_{11}. Deleting non-critical ones we get:



Suppose vertex 0 is the start, since all activity times are assumed >0, if finally ee[i]==0 (i≠0) then we can determine that vertex i is not reachable from the start.



Exercises: P389-2, p390-5

Chapter 7
Sorting

7.1 Motivation

In data structures, the order among data elements is a important relation and sorting becomes the most frequent computing task.

list --- a collection of records, each record having one or more fields.

key --- the fields used to distinguish among records.

Two important uses of sorting:

(1) Binary search a sorted list with n records in O(log n), much better than sequential search an unsorted list in O(n).

(2) Comparing two lists of n and m records containing data that are essentially the same but from different sources. If sorted we can do it in O(n+m), otherwise we need O(nm) time.

Actually there are much more applications of sorting, e.g., query optimization, job scheduling, etc..

Consequently, sorting problem has been extensively studied, and there are many methods proposed.

We shall study several typical methods, indicating when one is superior to others.

Now let us formally state the sorting problem.

- a list of records ($R_1, R_2, \ldots, R_n$)
- each R_i has key value K_i
- assume an ordering relation (<) on the keys, so that for any 2 key values x and y, x=y or x<y or x>y. < is transitive.

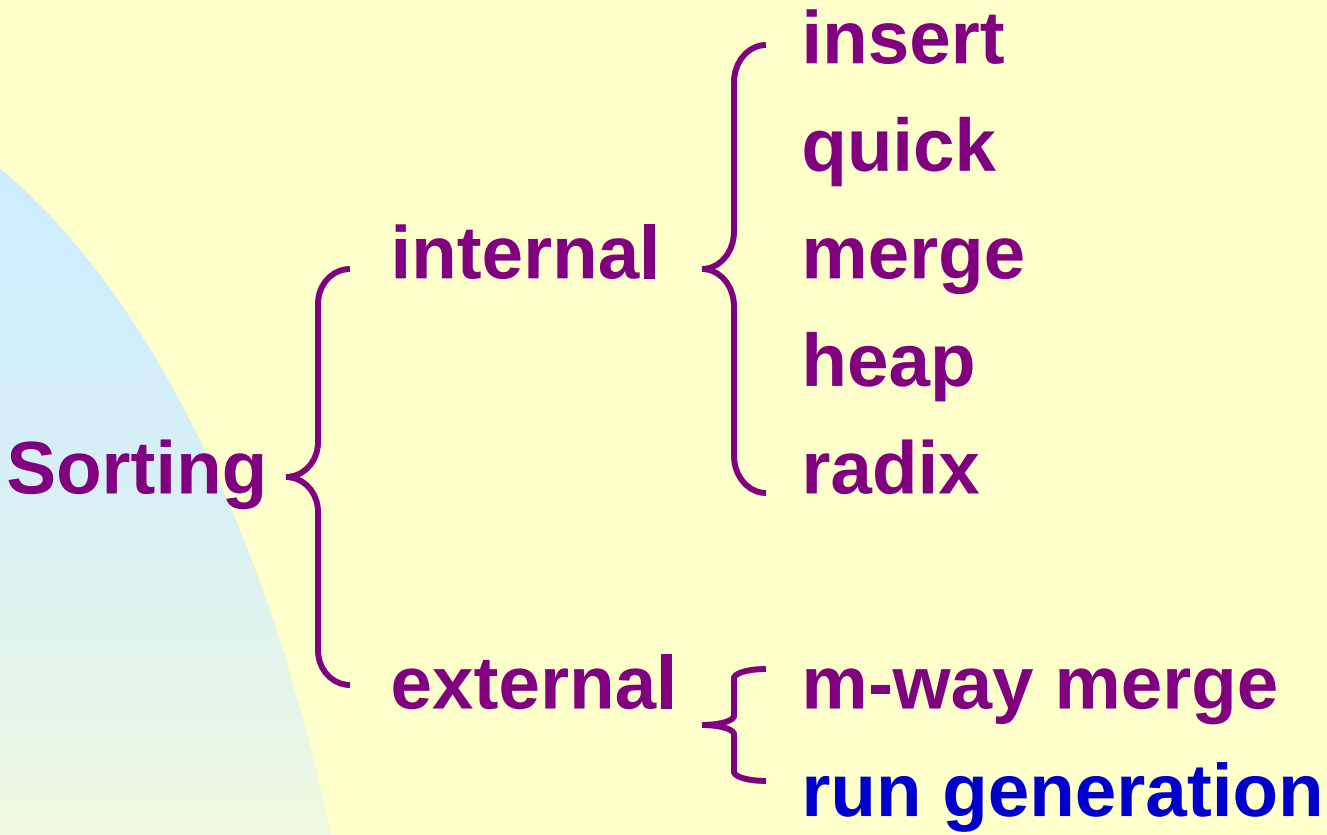
The sorting problem is that of finding a permutation, σ, such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$, $1 \leq i \leq n-1$. The desired ordering is ($R_{\sigma(1)}, R_{\sigma(2)}, \ldots, R_{\sigma(n)}$).

Let σ_s be the permutation with the following properties:

(1) $K_{\sigma s(i)} \leq K_{\sigma s(i+1)}$, $1 \leq i \leq n-1$.

(2) If $i<j$ and $K_i = K_j$ in the input list, then R_i precedes R_j in the sorted list.

A sorting method that generates the permutation σ_s is **stable.**

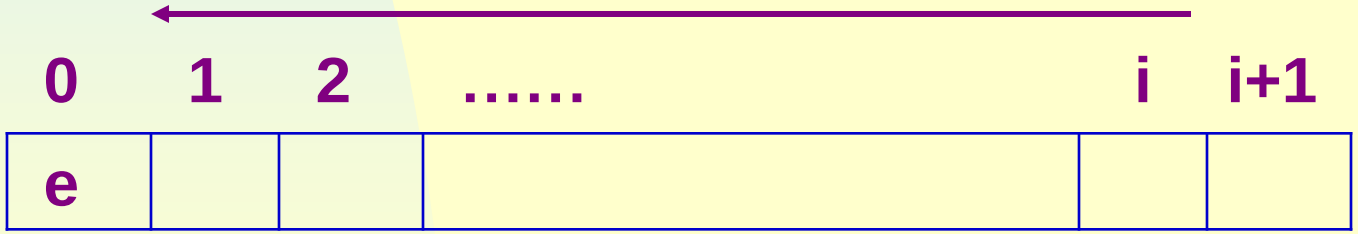


Throughout, we assume that relational operators have been overloaded so that record comparison is done by comparing their keys.

7.2 Insert Sort

Basic step: insert e into a sorted sequence of i records in such a way that the resulting sequence of size i+1 is also ordered.

Uses a[0] to simplify the while loop test.



```
template<class T>
viod Insert(const T& e, T *a, int i)
{ //insert e into the ordered sequence a[1:i] such that the
  // resulting sequence a[1:i+1] is also ordered. The array a
  // must have space for at least i+2 elements.
     a[0]=e;
     while (e < a[i])   //stable
     {
        a[i+1]=a[i];
        i--;    // a[i+1] is always ready for storing element
      }
      a[i+1]=e;
}
```

Insertion sort:

Begin with the ordered sequence a[1], then successively insert a[2], a[3], …, a[n] into the sequence.

```
template<class T>
void InsertionSort (Element *a, const int n)
{ //sort a[1:n] into nondescreasing order.
   for (int j=2; j<=n; j++) {
      T temp = a[j];  // necessary, because a[j] may change in
                      // Insertion
      Insert (temp, a, j-1);
   }
}
```

Analysis of insert sort:

(1) The worst case

Insert(e, a, i) makes i+1 comparisons before making insertion --- O(i).

InsertSort invokes Insert for i=j-1=1, 2,…,n-1, so the overall time is

$$O\left(\sum_{i=1}^{n-1}(i+1)\right)=O(n^2).$$

(2) Estimate of the actual computing time

R_i is **left out of order** (LOO) iff $R_i < \max_{1 \le j < i} \{ R_j \}$.

For every record, at least O(1) is needed. If k is the number of LOO records, the computing time is O(n+kn).

It can also be shown that the average time is $O(n^2)$.

When k<<n, this method is very desirable. And for $n \le 30$, it is the fastest.

Example 7.1: n=5, key sequence is 5, 4, 3, 2, 1

j	[1]	[2]	[3]	[4]	[5]
-	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

Example 7.2: n=5, key sequence is 2, 3, 4, 5, 1

j	[1]	[2]	[3]	[4]	[5]
-	**2**	3	4	5	1
2	**2**	**3**	4	5	1
3	**2**	**3**	**4**	5	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

InsertSort is stable.

Variations:

1 Binary Insert Sort.

2 Linked Insert Sort.

Exercises: P401-1, 3

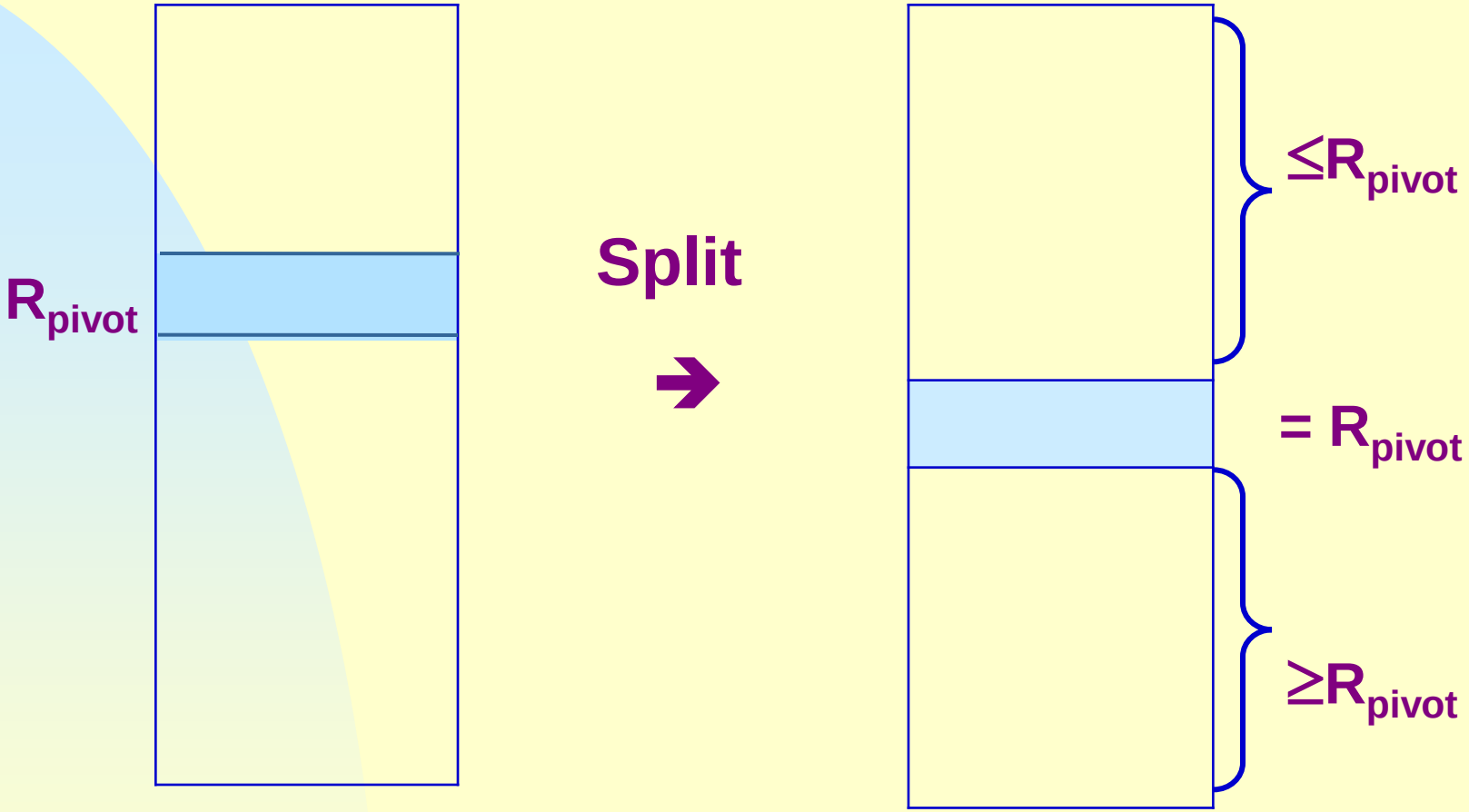
7.3 Quick Sort

The quick-sort has the best average behavior among the sorting methods we shall be studying.

Idea: given a list (R_{left}, R_{left+1}, …, R_{right}), select a pivot record from among R_i (left ≤ i ≤ right) , put the pivot in the correct spot s(pivot) such that after the positioning,

$R_j \le R_{s(pivot)}$ **for j<s(pivot)**

$R_j \ge R_{s(pivot)}$ **for j>s(pivot)**


R_{pivot}
Split
$\leq R_{pivot}$
$= R_{pivot}$
$\geq R_{pivot}$

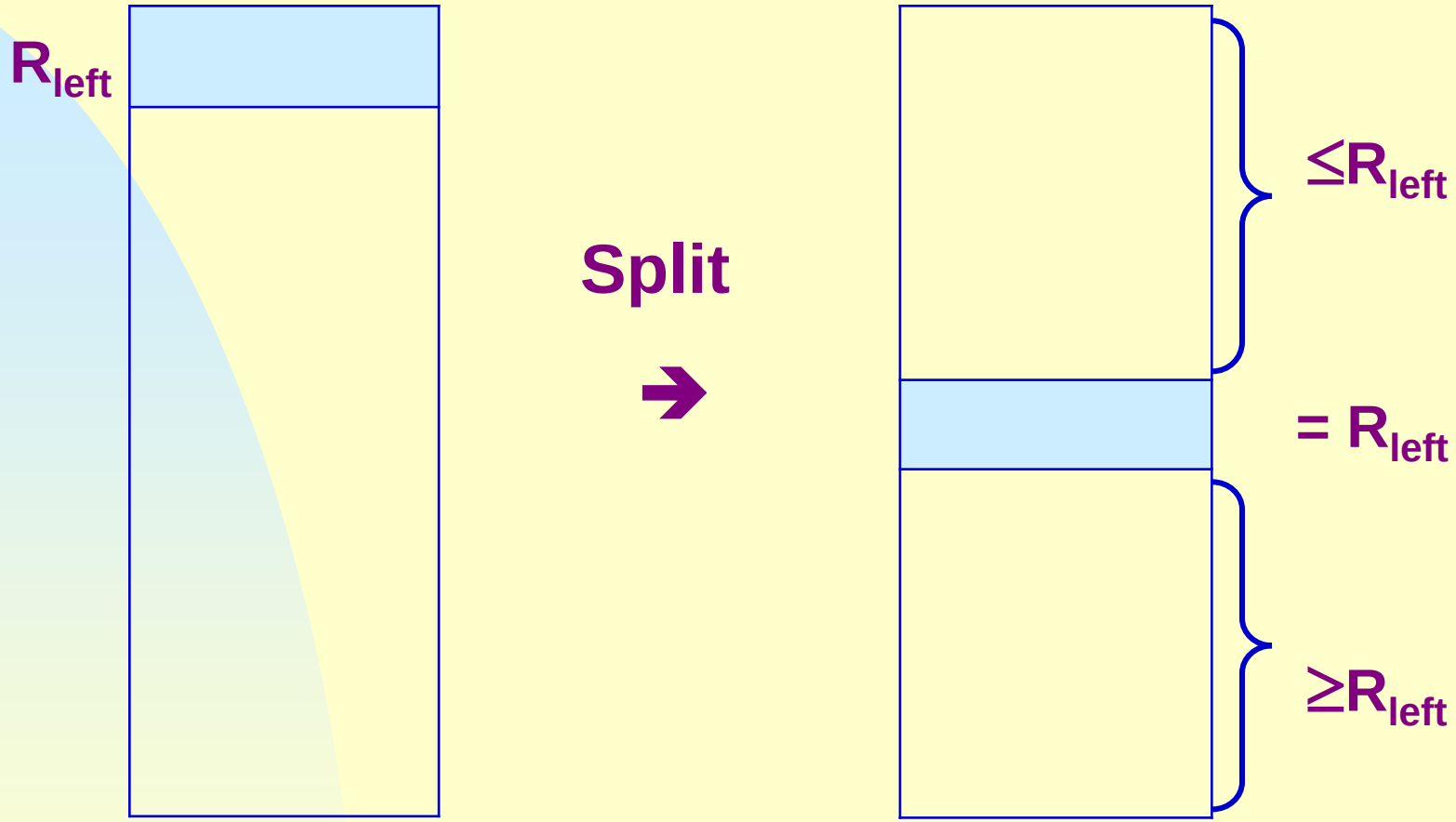
The original list is partitioned into 2 sublists:

$(R_{left}, \ldots, R_{s(pivot)-1})$ and

$(R_{s(pivot)+1}, \ldots, R_{right})$

and they may be sorted independently.

For simplicity, we just choose R[left] as the pivot.


R_{left}
Split
$\leq R_{left}$
$= R_{left}$
$\geq R_{left}$

```
template <class T>
void QuickSort (T *a, const int left, const int right)
{ // Sort a[left:right] into non-decreasing order. a[left] is
  // arbitrarily chosen as the pivot. Assume a[left]≤a[right+1].
    if (left < right) {
      int i=left, j=right+1, pivot=a[left];
      do {
          do i++; while (a[i]<pivot);
          do j --;  while (a[j]>pivot);
          if (i<j) swap(a[i], a[j]);
      }  while (i<j);
      swap(a[left], a[j]);
      QuickSort(a, left, j-1);
      QuickSort(a, j+1, right);
   }
}
```

To sort a[1:n], invoke QuickSort(a, 1, n).

Example 7.3:

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	left	right
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

Analysis of QuichSort:

(1) The worst case

2 sublists with size n-1 and 0.

$T_{worst}(n) \le cn + T_{worst}(n-1)$

$\le cn + c(n-1) + T_{worst}(n-2)$

…

$\le c \sum_{i=1}^{n} i + T_{worst}(0) = cn(n+1)/2 + d = O(n^2)$.

(2) The optimal case

2 sublists with equal size, roughly n/2.

$T_{opt}(n) \le cn + 2T_{opt}(n/2)$

$\le cn + 2(cn/2+2T_{opt}(n/4)$

$\le 2cn + 4T_{opt}(n/4)$

…

$\le cn\log_2 n + nT_{opt}(1) = O(n\log n).$

(3) The average time

Lemma 7.1: $T_{avg}(n) \le k*n\log_e n$ for $n \ge 2$.

Proof:

In the call to QuickSort(a, 1, n), the pivot gets place at position j, we need to sort 2 sublists of size j-1 and n-j. j may take on any of the 1 to n with equal probability, so

$$T_{avg}(n) \le cn + \frac{1}{n}\sum_{j=1}^{n}(T_{avg}(j-1) + T_{avg}(n-j))$$

$$= cn + \frac{2}{n}\sum_{j=0}^{n-1} T_{avg}(j), n \ge 2 ---(7.1)$$

Let $T_{avg}(0) \le b$ and $T_{avg}(1) \le b$ for some constant b,

$k=2(b+c)$,

we show $T_{avg}(n) \le knlog_e n$ for $n \ge 2$.

Induction on n.

n=2, from (7.1) $T_{avg}(2) \le 2c+2b \le k*2log_e 2$.

Assume $T_{avg}(n) \le k*nlog_e n$ for $2 \le n < m$.

Then

$$T_{avg}(m) \leq cm + \frac{2}{m} \sum_{j=0}^{m-1} T_{avg}(j)$$

$$= cm + \frac{4b}{m} + \frac{2}{m} \sum_{j=2}^{m-1} T_{avg}(j)$$

$$\leq cm + \frac{4b}{m} + \frac{2k}{m} \sum_{j=2}^{m-1} j \log_e j - - - (7.2)$$

Since $j\log_e j$ is an increasing function of j, Eq. (7.2) yields

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2k}{m}\int_2^m x\log_e x\,dx$$

$$= cm + \frac{4b}{m} + \frac{2k}{m}\left[\frac{m^2\log_e m}{2} - \frac{m^2}{4}\right]$$

$$= cm + \frac{4b}{m} + km\log_e m - \frac{km}{2}$$

$$= km\log_e m + \left[cm + \frac{4b}{m} - \frac{km}{2}\right]$$

For m≥2

$$cm + \frac{4b}{m} - \frac{km}{2} = cm + \frac{4b}{m} - \frac{2(b+c)m}{2}$$

$$= \frac{4b}{m} - bm \le 0$$

Therefore For m≥2, $T_{avg}(m) \le km\log_e m$.

Stack space requirement:

- **best case: split evenly --- O(log n)**
- **worst case: split into a left sublist of size n-1 and a right of 0 --- O(n)**
- **if we use only one recursion and other parts are replaced by iteration, and sort the smaller sublist first, let S(n) be the space for list of size n, then**
  - **S(1)=c (c is a constant)**
  - **$S(n) \leq c+S(n/2)=c \log n+S(1)=O(\log n)$**

- **better choice of the pivot:**

  pivot=median $\{R_{left}$, $K_{(left+right)/2}$, $R_{right}\}$

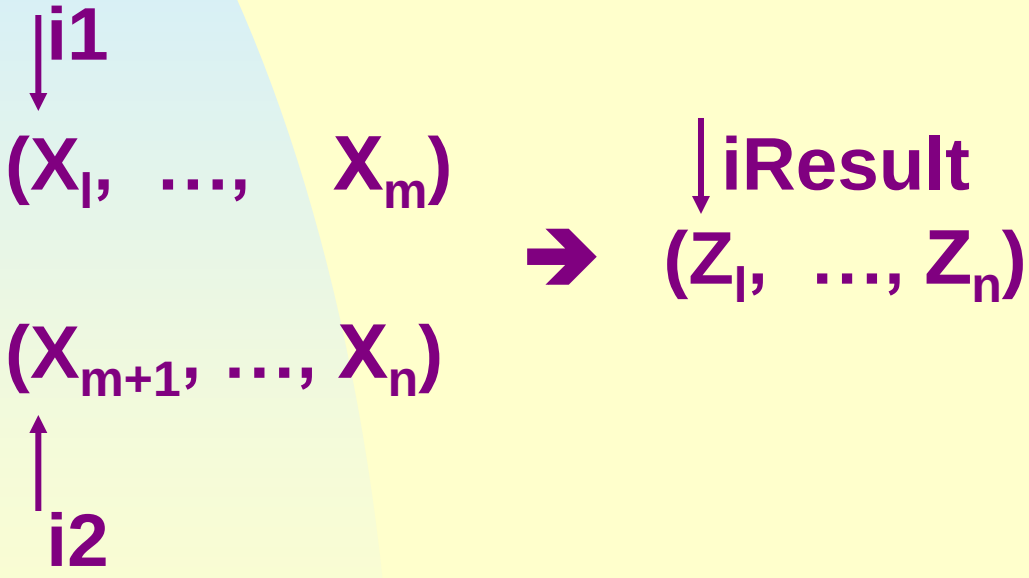
Note also that quick sort is unstable.

Exercises: P405-1, 2, 5

7.5 Merge Sort

7.5.1 Merging

Merge 2 sorted lists to get a single sorted list.



```
template <class T>
void Merge (T *initList, T *mergedList,
                                  const int l, const int m, const int n)
{ // two sorted lists initList[l:m] and initList[m+1:n] are
  // merged to obtain the sorted list mergedList[l:n].
    for (int i1=l, iResult=l, i2=m+1; i1<=m && i2<=n; iResult++)
      if (initList[i1] <= initList[i2])
          mergedList[iResult]=initList[i1++];  //stable
      else
          mergedList[iResult]=initList[i2++];
    // copy remaining records , if any, of the first list
    copy(initList+i1, initList+m+1, mergedList+iResult);
    // copy remaining records , if any, of the second list
    copy(initList+i2, initList+n+1, mergedList+iResult);
}
```

Analysis of Merge:

At each iteration of the for loop, either i1++ or i2++, the for loop is iterated at most n-l+1 times.

At most n-l+1 records are copied.

The total time: O(n-l+1).

If each record has a size s, then the total time is O(s(n-l+1)).

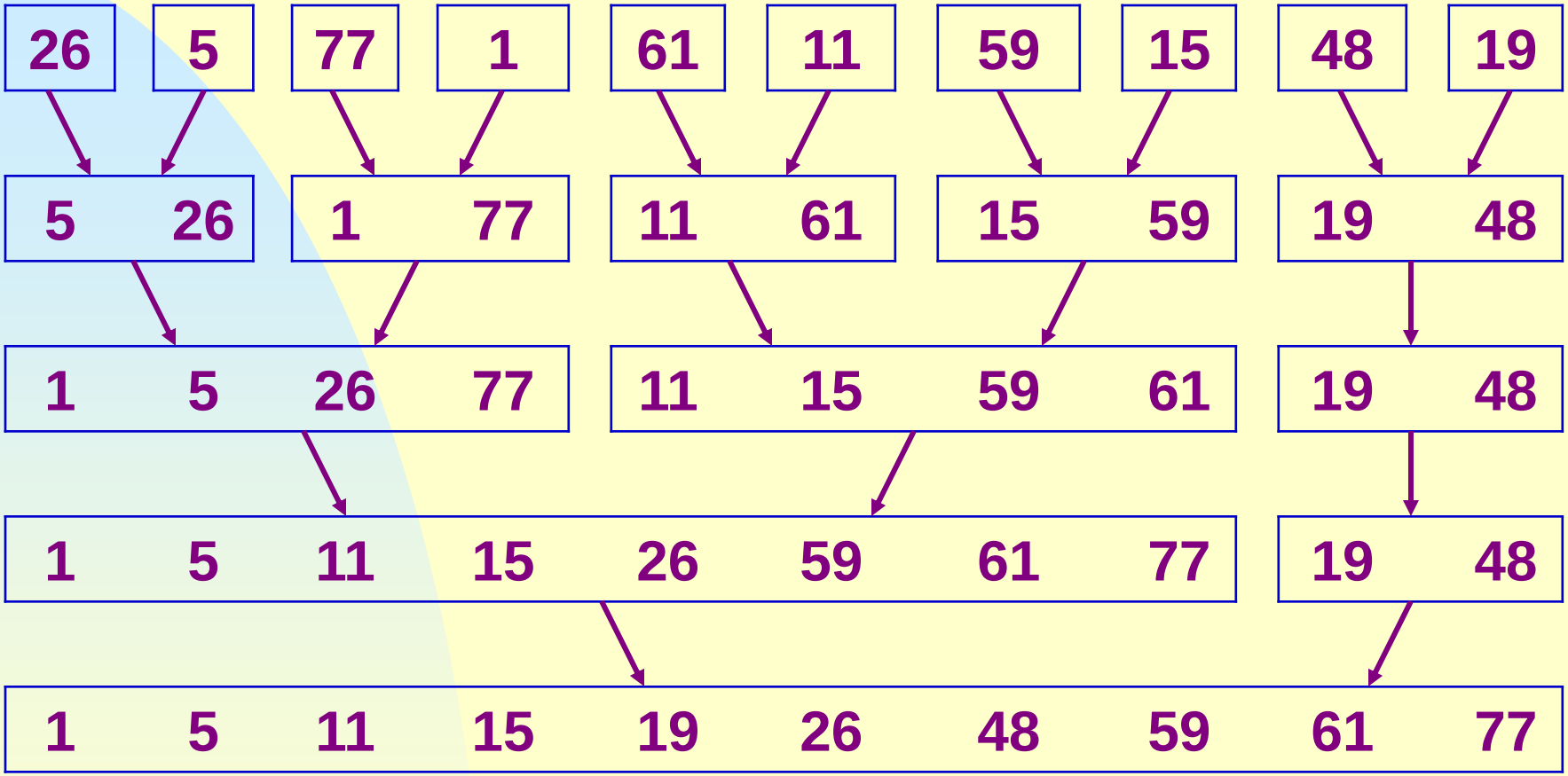
When s>1, use linked lists, only need n-l+1 links, the merge time becomes O(n-l+1) and is independent of s.

7.5.2 Iterative Merge Sort

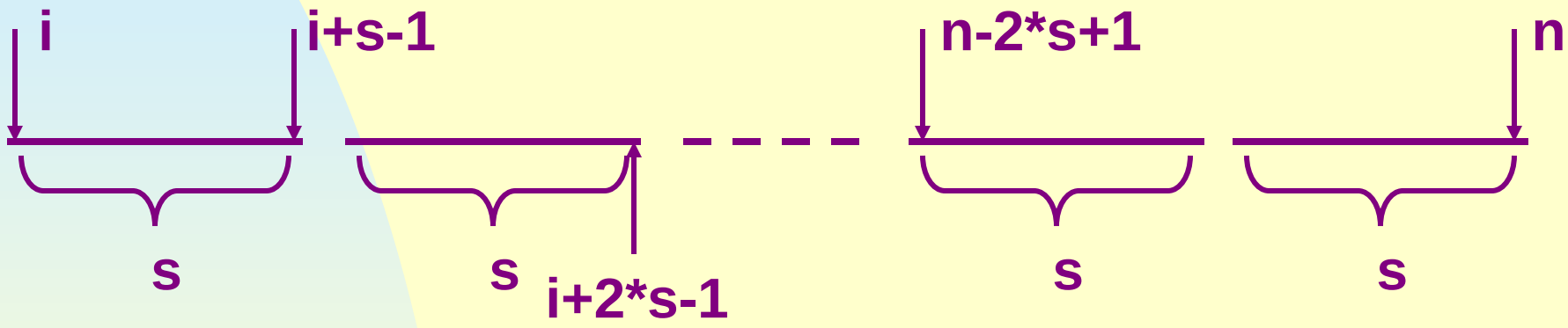
Basic idea:

At the beginning, interpret the input as n sorted sublists, each of size 1. These lists are merged by pairs to obtain n/2 lists, each of length 2 (if n is odd, then one list is of length 1). These n/2 lists are then merged by pairs, and so on until only one list is get.

As shown in the following:


26 5 77 1 61 11 59 15 48 19
5 26 1 77 11 61 15 59 19 48
1 5 26 77 11 15 59 61 19 48
1 5 11 15 26 59 61 77 19 48
1 5 11 15 19 26 48 59 61 77

A merge sort consists of several passes, it is convenient to write a function MergePass for this.



```
template <class T>
void MergePass (T *initList, T *resultList,
                                        const int n, const int s)
{ // adjacent pairs of sublists of size s are merged from initList
  // to resultList.
    for (int i=1; i<=n-2*s+1; //are there 2*s elements?
                                        i+=2*s)
        Merge(initList, resultList, i, i+s-1, i+2*s-1);
    //merge remaining list of length<2*s
    if ((i+s-1)<n) Merge (initList, resultList, i, i+s-1,n);
    else copy(initList+i, initList+n+1, resultList+i);
}
```

Now the sort can be done by repeatedly invoking MergePass.

```
template <class T>
void MergeSort (T *a, const int n)
{ // Sort a[1:n] into nondereasing order.
    T *tempList=new T[n+1];
    //l is the length of the sublist currently being merged
    for (int l=1; l<n; l*=2) {
        MergePass(a, tempList, n, l);
        l*=2;
        MergePass ( tempList, a, n, l);// last pass may just copy
    }
    delete [ ] tempList;
}
```

Analysis of MergeSort:

Makes several passes. After the 1st pass, the result sublists are of size $2=2^1$. After the ith pass, the size is 2^i. Consequently, a total of $\lceil \log_2 n \rceil$ passes are made, each takes o(n), the total time is O(n log n).

It is easy to verify that MergeSort is **stable**.

Exercises: P412-1

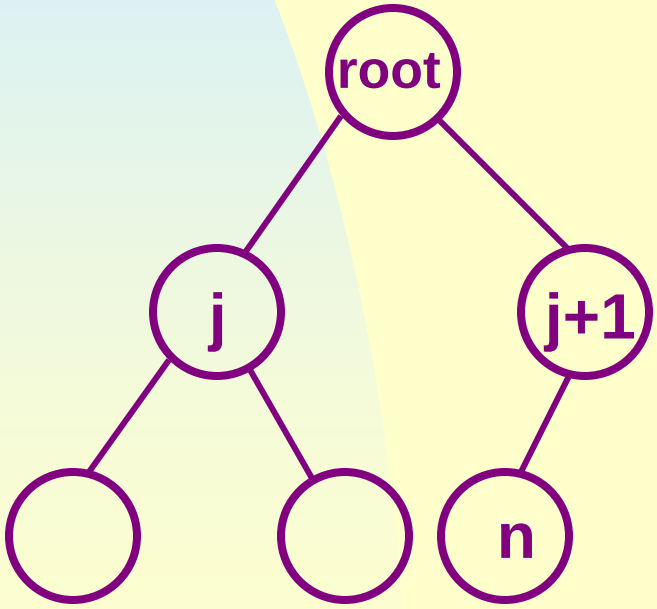
7.6 Heap Sort

Merge sort has a computing time of O(n log n), both in worst and average case, but it requires O(n) additional storage.

Heap sort requires o(1) additional space, and also has as its worst and average computing time O(n log n).

In heap sort, we utilize the max-heap structure in chapter 5 .

To create and recreate a heap efficiently, we need a function Adjust, which starts with a binary tree whose left and right subtrees are max heaps and adjusts the entire binary tree into a max heap.



j/2 is always ready for storing record

```
template <class T>
void adjust (T *a, const int root, const int n)
{ // No node index is > n
    T e=a[root];
    // find proper place for e
    for (int j=2*root; j<=n; j*=2)  {
       if (j<n && a[j]<a[j+1]) j++;  // j is the lager child of its parent
       if (e>=a[j]) break;
       a[j/2]=a[j];  //move jth record up the tree
    }
    a[j/2]=e;
}
```

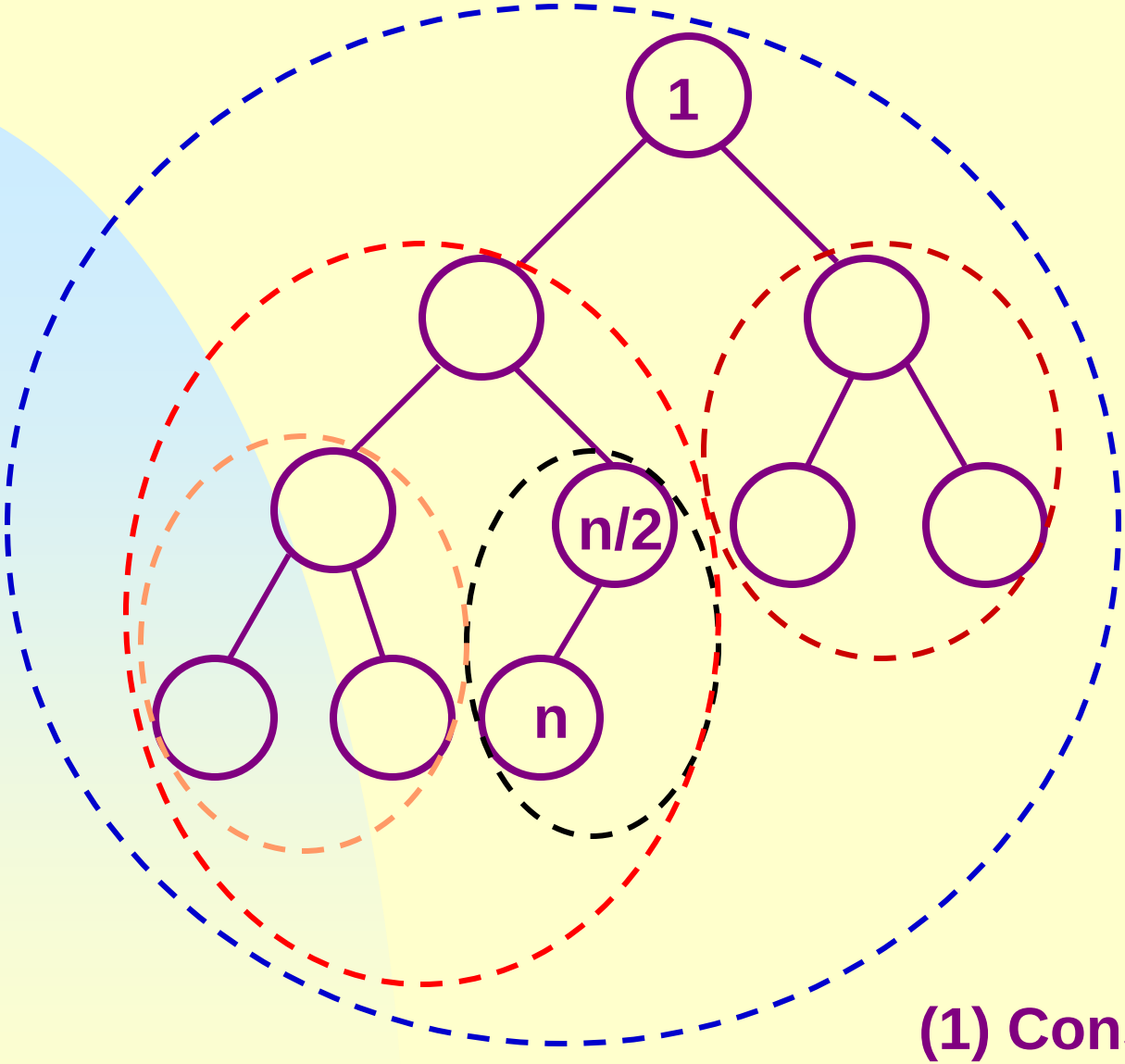
Analysis of adjust:

If the depth of tree is d, the for loop is executed at most d times. Hence, the computing time is O(d).

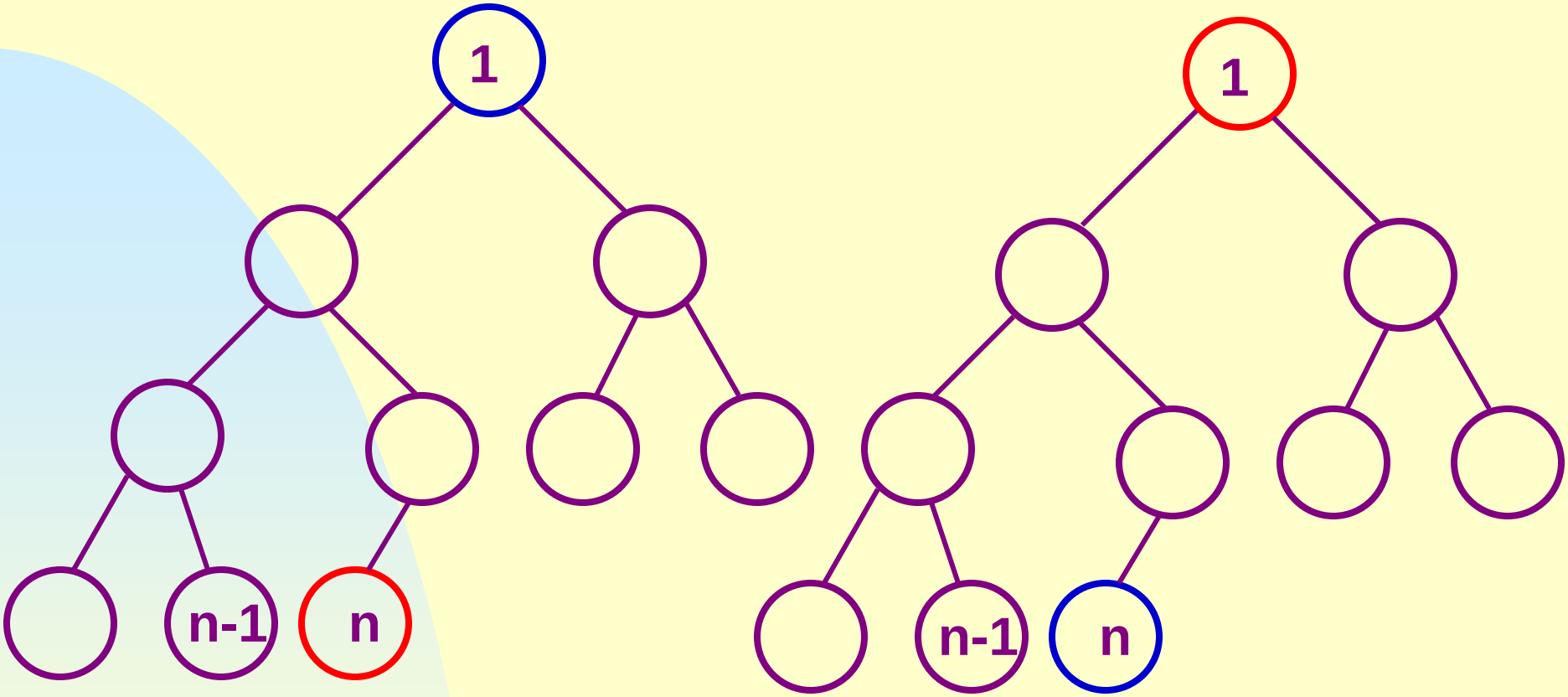
To sort the list

(1) Create a max heap by using Adjust.

(2) Make n-1 passes, in each pass, swap the first and last records in the heap, and decrement the heap size and readjust it.



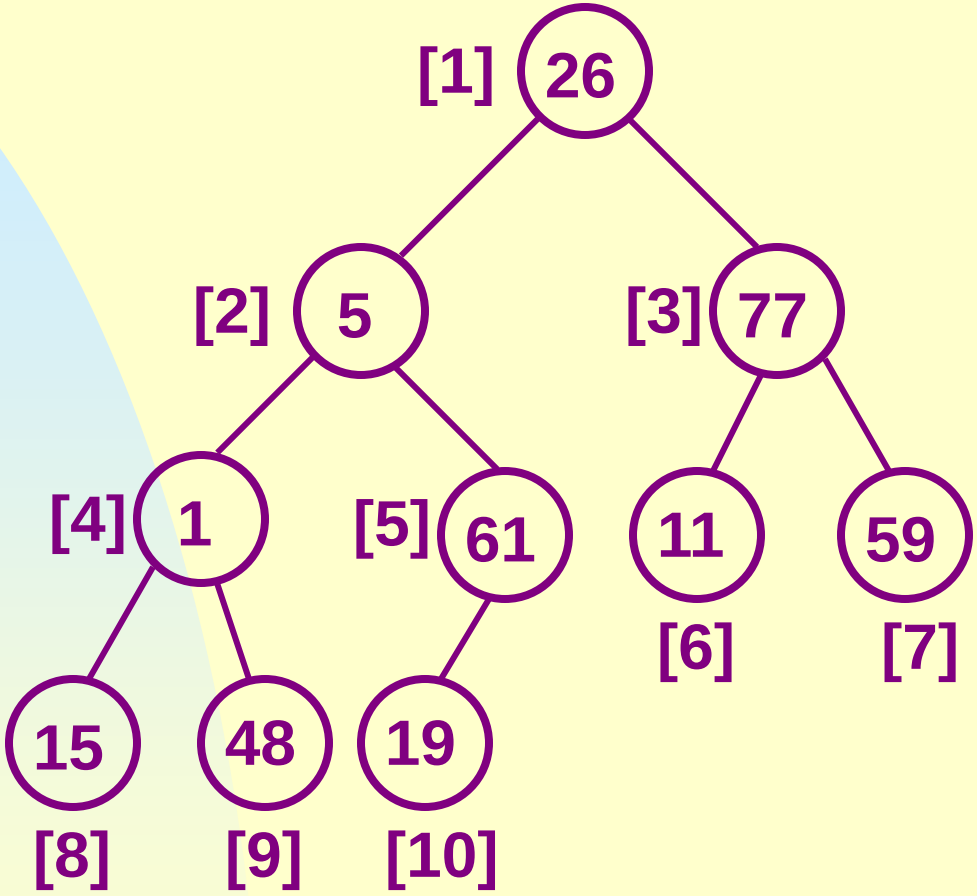
(1) Construct a heap



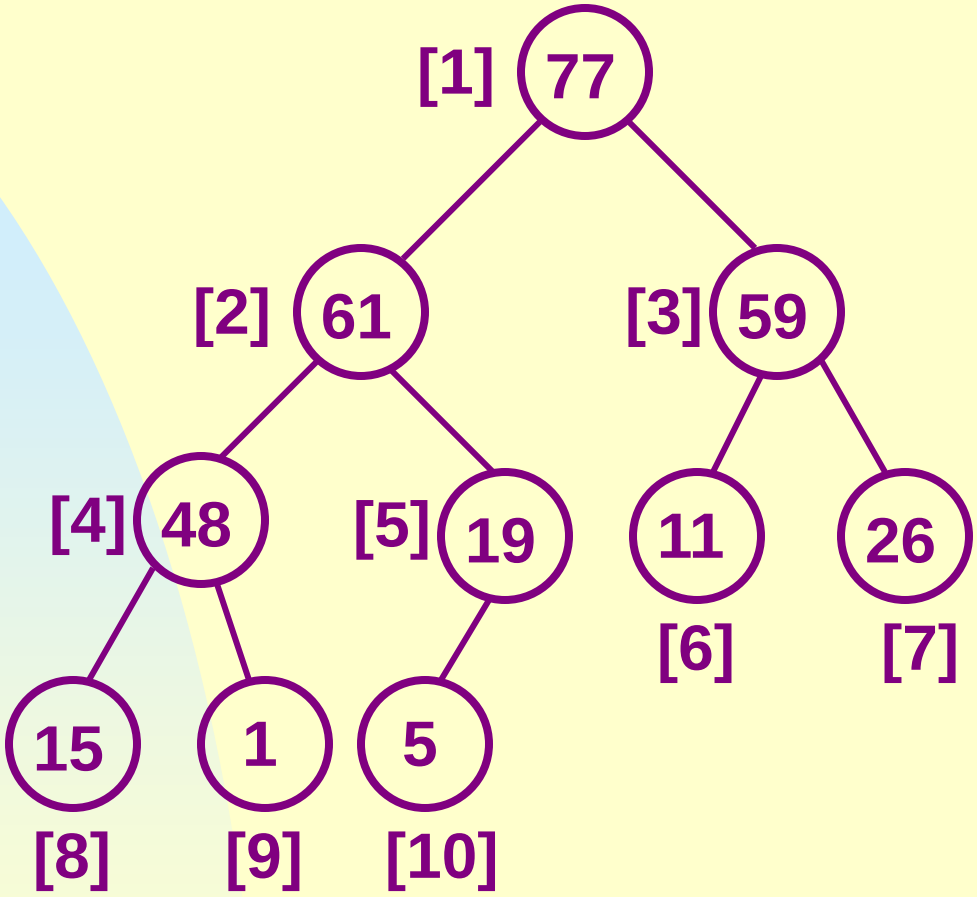
(2) Swap and readjust

```
template <class T>
void HeapSort (T *a, const int n)
{ // Sort a[1:n] into nondeceasing order.
   for (int i=n/2; i>=1; i--)   // convert list into a heap
      Adjust(a, i, n);
   for (i=n-1; i>=1; i--)        // sort
   {
      swap(a[1], a[i+1];    // swap first and last of current heap
      Adjust(a, 1, i);         // recreate heap
   }
}
```

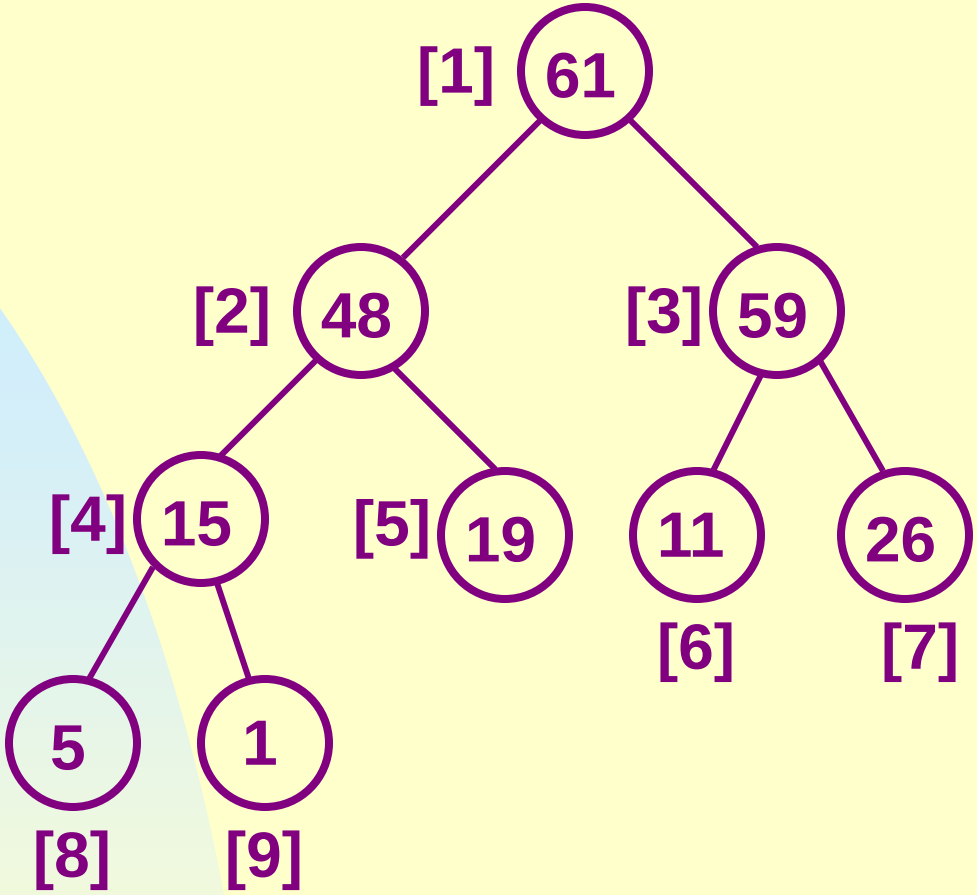
Example 7.7:



(a) Input list

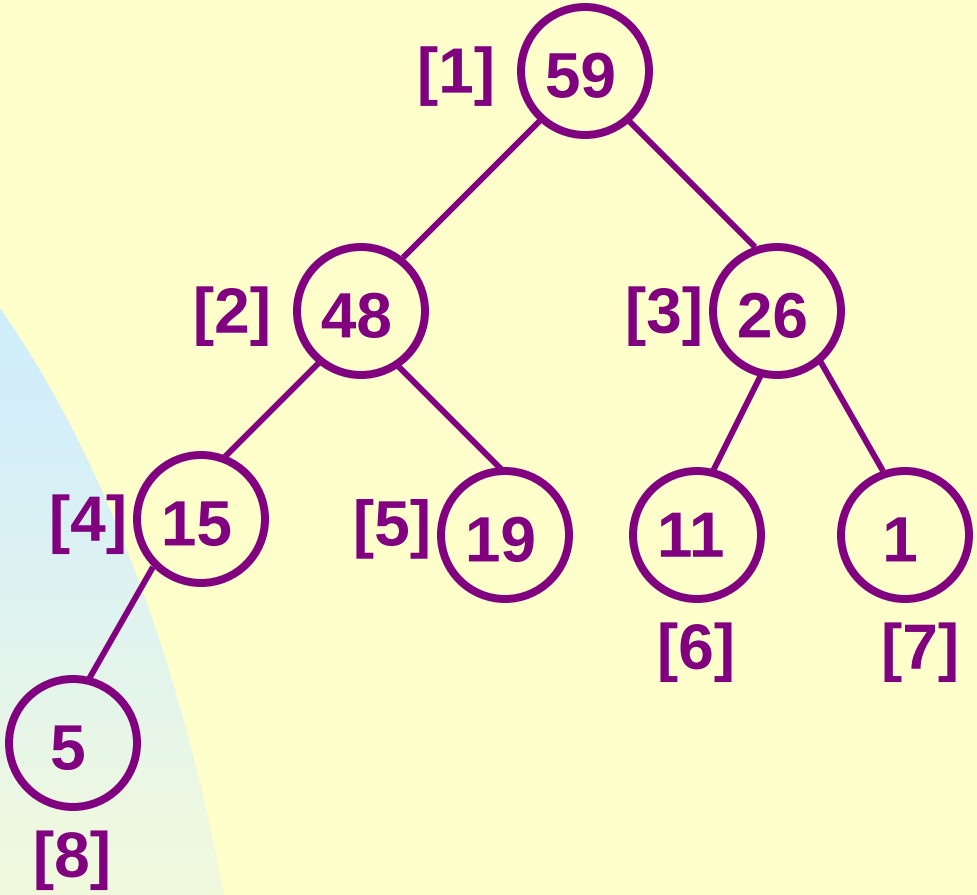


(b) Initial heap



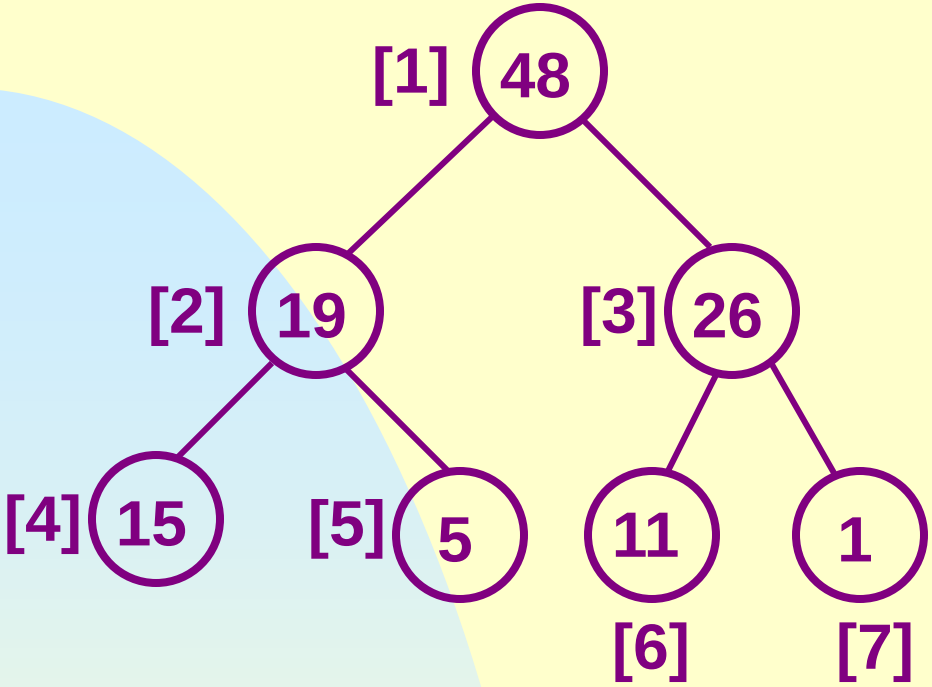
(c) Heap size=9

Sorted=[77]



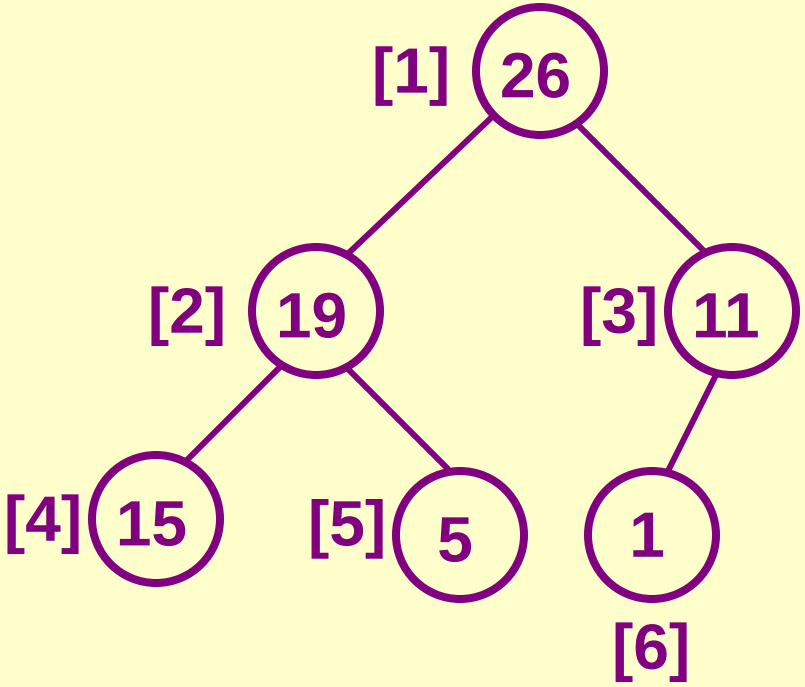
(d) Heap size=8

Sorted=[61, 77]



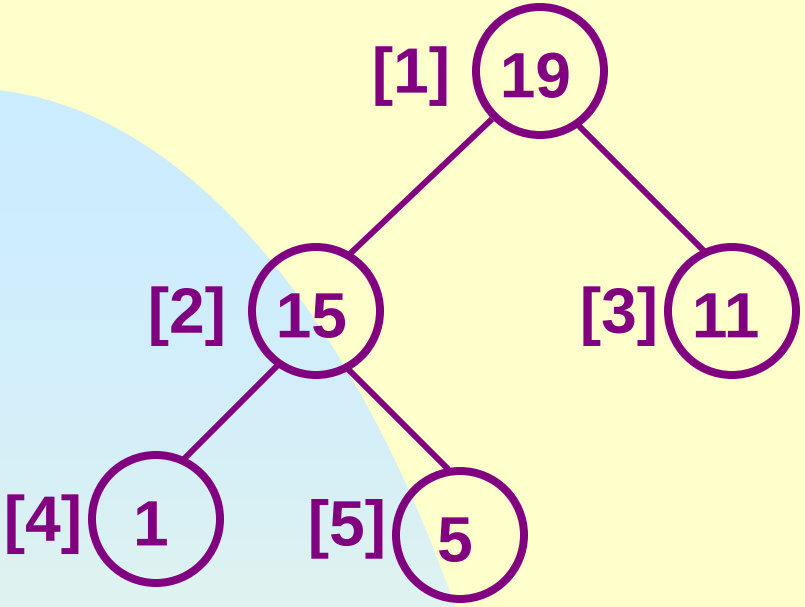
(e) Heap size=7

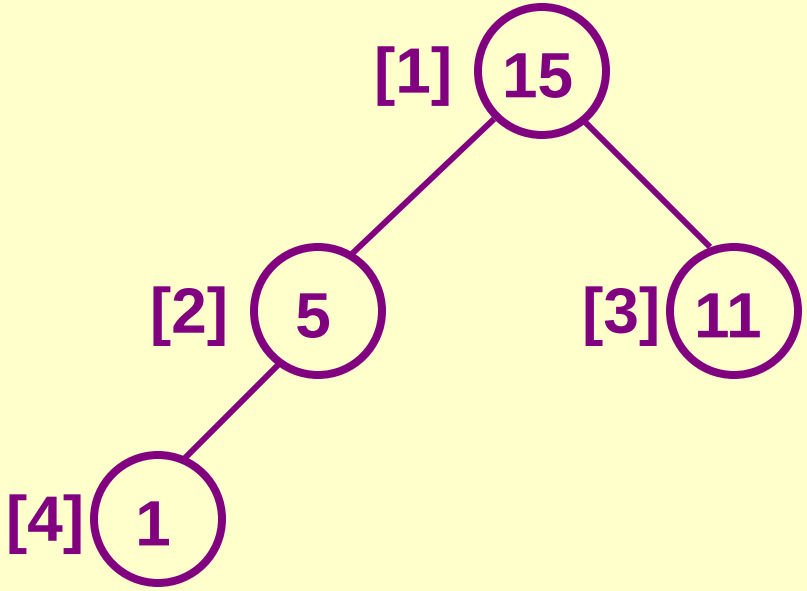
Sorted=[59, 61, 77]



(f) Heap size=6

Sorted=[48, 59, 61, 77]



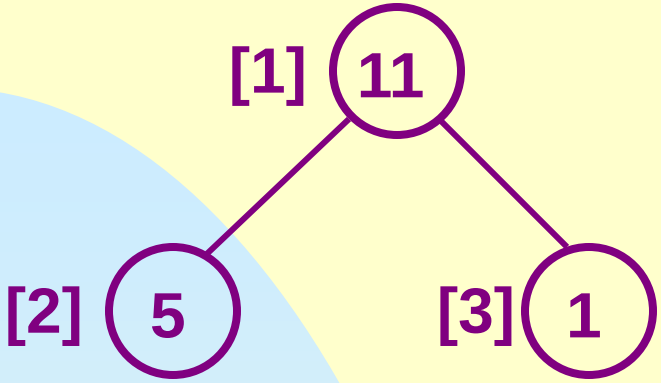


(g) Heap size=5

[26, 48, 59, 61, 77]

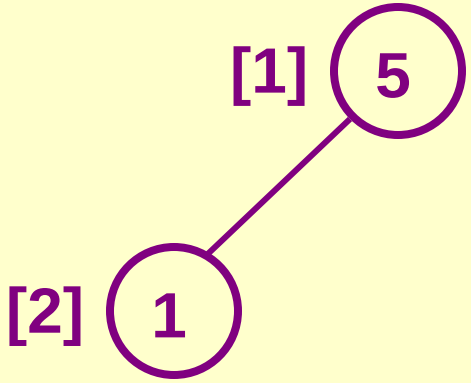
(h) Heap size=4

[19, 26, 48, 59, 61, 77]



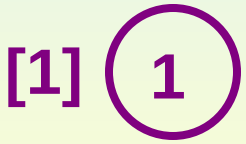
(i) Heap size=3

[15, 19, 26, 48, 59, 61, 77]



(j) Heap size=2

[11, 15, 19, 26, 48, 59, 61, 77]



(j) Heap size=1

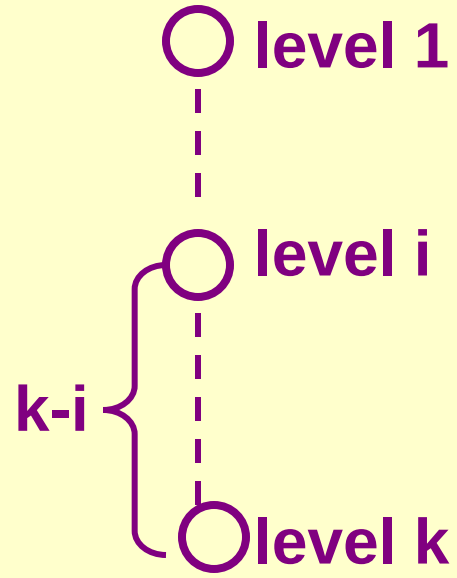
[5, 11, 15, 19, 26, 48, 59, 61, 77]

Analysis of HeapSort:

- suppose $2^{k-1} \le n < 2^k$, the tree has k levels.
- the number of nodes on level i $\le 2^{i-1}$.
- in the first loop, Adjust is called once for each node that has a child, hence the time is no more than

$$\sum_{1 \le i \le k-1} 2^{i-1}(k-i) =$$

$$\sum_{1 \le i \le k-1} 2^{k-i-1} i \le n \sum_{1 \le i \le k-1} i/2^i < 2n = O(n)$$



- in the next loop, n-1 applications of Adjust are made with maximum depth $k = \lceil \log_2(n+1) \rceil$.

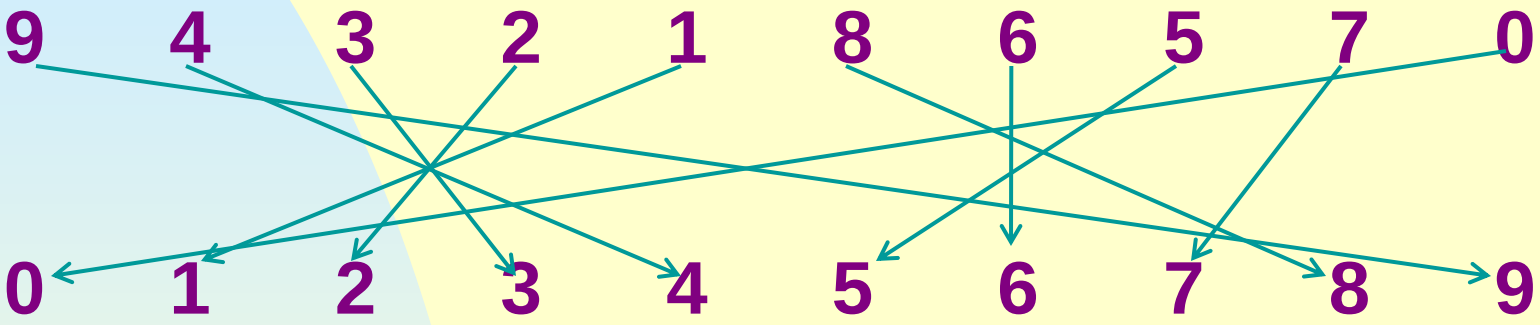
The total time: O(n log n).

Additional space: O(1).

Exercises: P416-1, 2

7.7 Sorting on Several Keys

Observe :



This is the basic idea of bin sort, to make effective use of it, we can interpret a key as several sub-keys.

Problem : to sort records on keys K^1, K^2, …, K^d (K^1 is the most significant key and K^d the least).

A list of records R_1, R_2, …, R_n is sorted with respect to the keys K^1, K^2, …, K^d iff

For every pair of i and j, i<j and

$$(K_i^1,\ldots,K_i^d) \le (K_j^1,\ldots,K_j^d)$$

For example: sort a deck of cards may be regarded as a sort on 2 keys:

K^1 [Suits]:

♣ < ♦ < ♥ < ♠

K^2 [Face values]:

2<3<4<5<6<7<8<9<10<J<Q<K<A

A sorted deck of cards has the following ordering:

2♣,…, A♣,…, 2♠,…, A♠

LSD (least significant digit first) sort :

- sort the cards first into 13 piles corresponding to their face values. Place K's on top of A's, … , 2's on top of 3's.
- then sort on the suit using a **stable** method to obtain 4 piles, combine the piles to obtain the sorted cards.

To sort on each key K^i, use bin sort, i. e., if $0 \leq K^i < r$, r bins are set up, one for each value of K^i, and records are placed into their corresponding bins. For n records, the time is O(n+r)

For one logical key, we can interpret it as being composed of several keys, e.g., if $0 \leq K \leq 999$, K can be considered as $(K^1 K^2 K^3)$, $0 \leq K^i < 10$, i=1,2,3, this is also called Radix-10 sort.

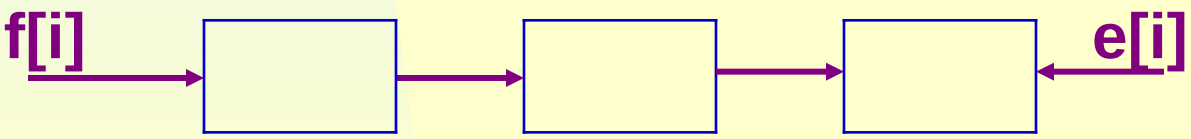
In general, for a Radix-r sort, r bins are needed.

Assume:

(1) records R_1, R_2, …, R_n and their keys are decomposed using a radix of r, each key have d digits in the range of [0, r-1].

(2) records have a link field.

(3) for each bin i, $0 \leq i < r$



```
template <class T>
int RadixSort(T*a, int*link, const int d, const int r, const int n)
{ // Sort a[1:n] using a d-digit radix-r sort. digit(e, i, r) returns
  // the ith radix-r digit (from the left, the first is the 0th digit)
  // of e's key. Each digit is in the range of [0, r).
    int e[r],  f[r];  // queue end and front pointers

    // create initial chain of records starting at first
    if (n==0) return 0;  int first=1;
    for (int i=1; i<n; i++) link[i]=i+1; // linked into a chain
    link[n]=0;

    for (i=d-1; i>=0; i--)
    { // sort on digit i
       fill(f, f+r, 0); // initialize bins to empty queues
       for (int current=first; current; current=link[current])
          {   // put records into queues
```

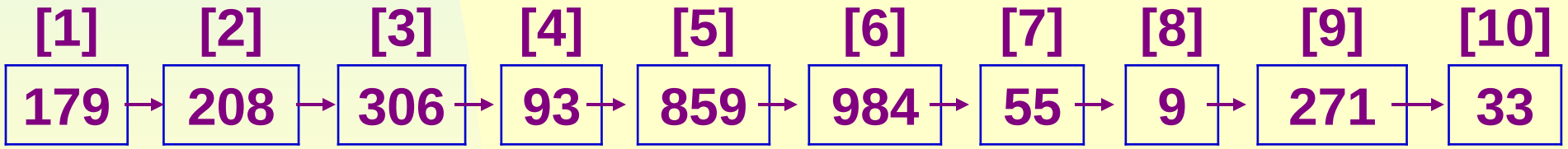
```
            int k=digit(a[current], i, r);
            if (f[k]==0) f[k]=current;
            else link[e[k]]=current;
            e[k]=current;
        }
        for (int j=0; !f[j] ; j++); // find first nonempty queue
        first=f[j];  int last=e[j];
        for (int k=j+1; k<r; k++) // concatenate remaining queues
           if (f[k] ) {
              link[last]=f[k];  last=e[k];
           }
        link[last]=0;
    }
    return first;
}
```

Analysis of RadixSort:

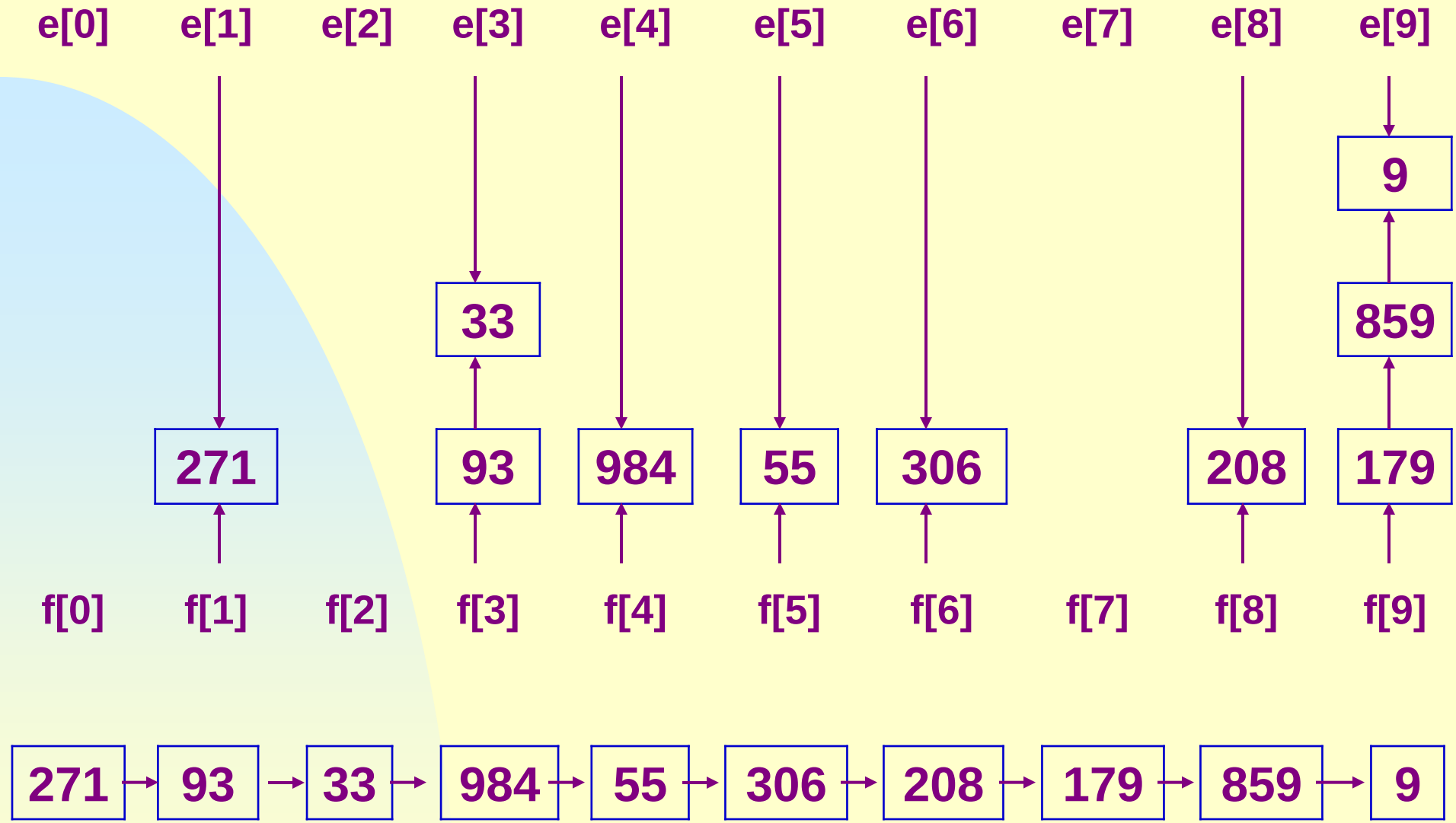
d passes over the data, each taking O(n+r). Hence the total time is O(d(n+r)).

d will depend on r and the largest key.

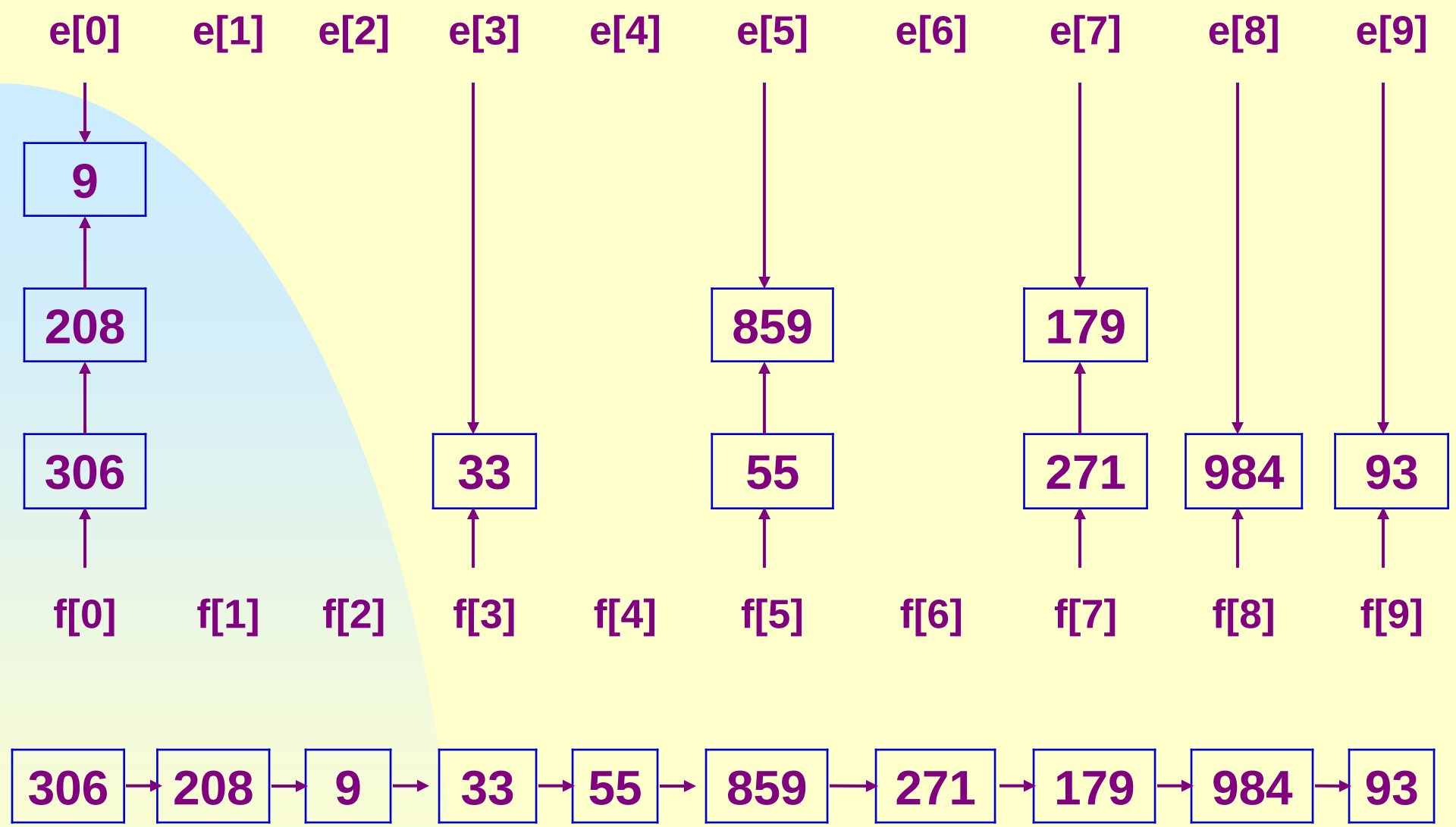
Example 7.8: sort 10 numbers in the range [0,999], r=10, hence d=3.



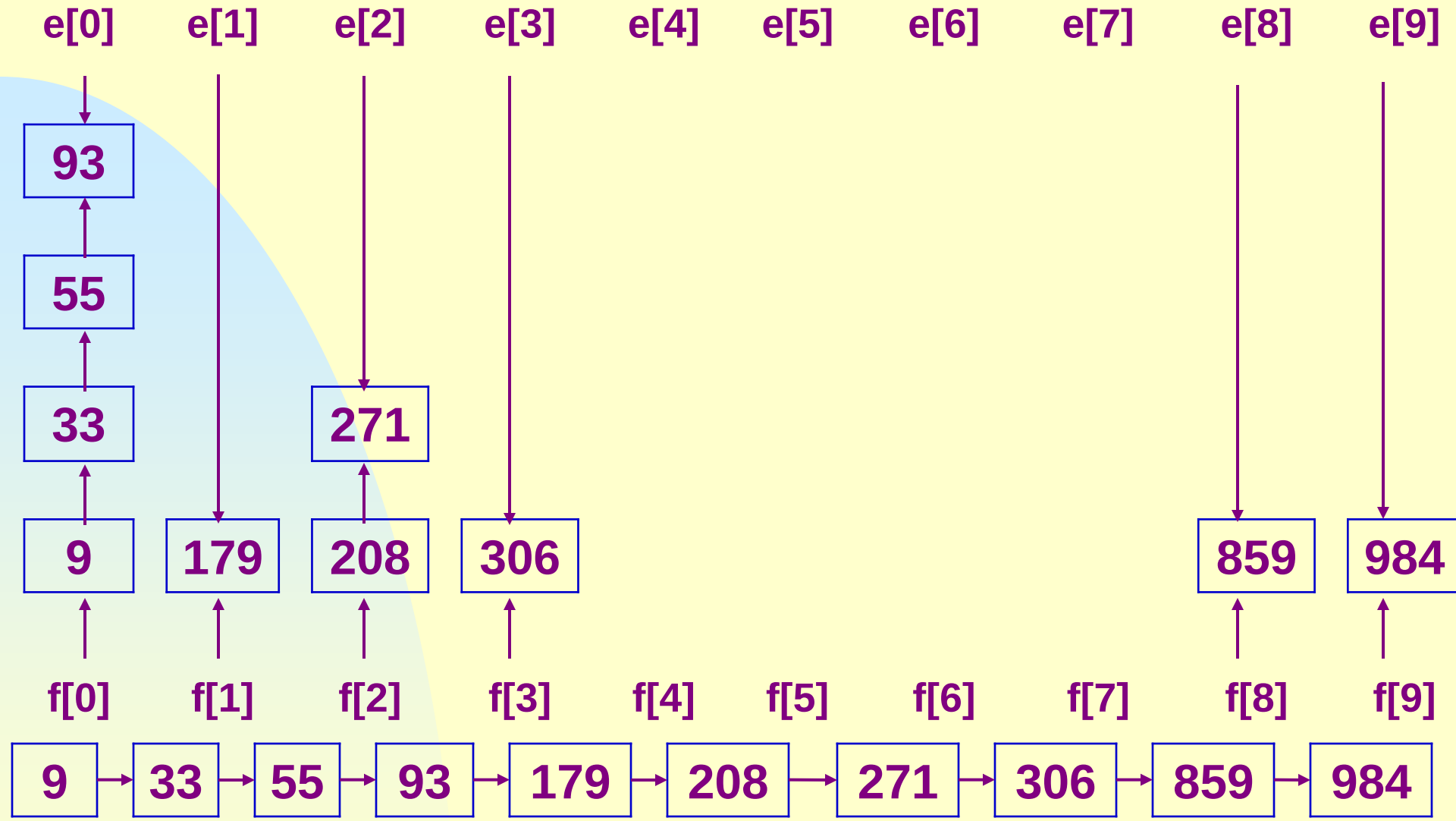
(a) Initial input


e[0]
e[1]
e[2]
e[3]
e[4]
e[5]
e[6]
e[7]
e[8]
e[9]
9
33
859
271
93
984
55
306
208
179
f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]
f[8]
f[9]
271
93
33
984
55
306
208
179
859
9

(b) First pass



(c) second pass


e[0]
e[1]
e[2]
e[3]
e[4]
e[5]
e[6]
e[7]
e[8]
e[9]
93
55
33
271
9
179
208
306
859
984
f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]
f[8]
f[9]
9
33
55
93
179
208
271
306
859
984

(d) third pass

Exercises: P422-1, 3, 5

7.9 Summary of Internal Sorting

The behavior of Radix Sort depends on the size of keys and the choice of r.

The following summarizes the other 4 methods we have studied:

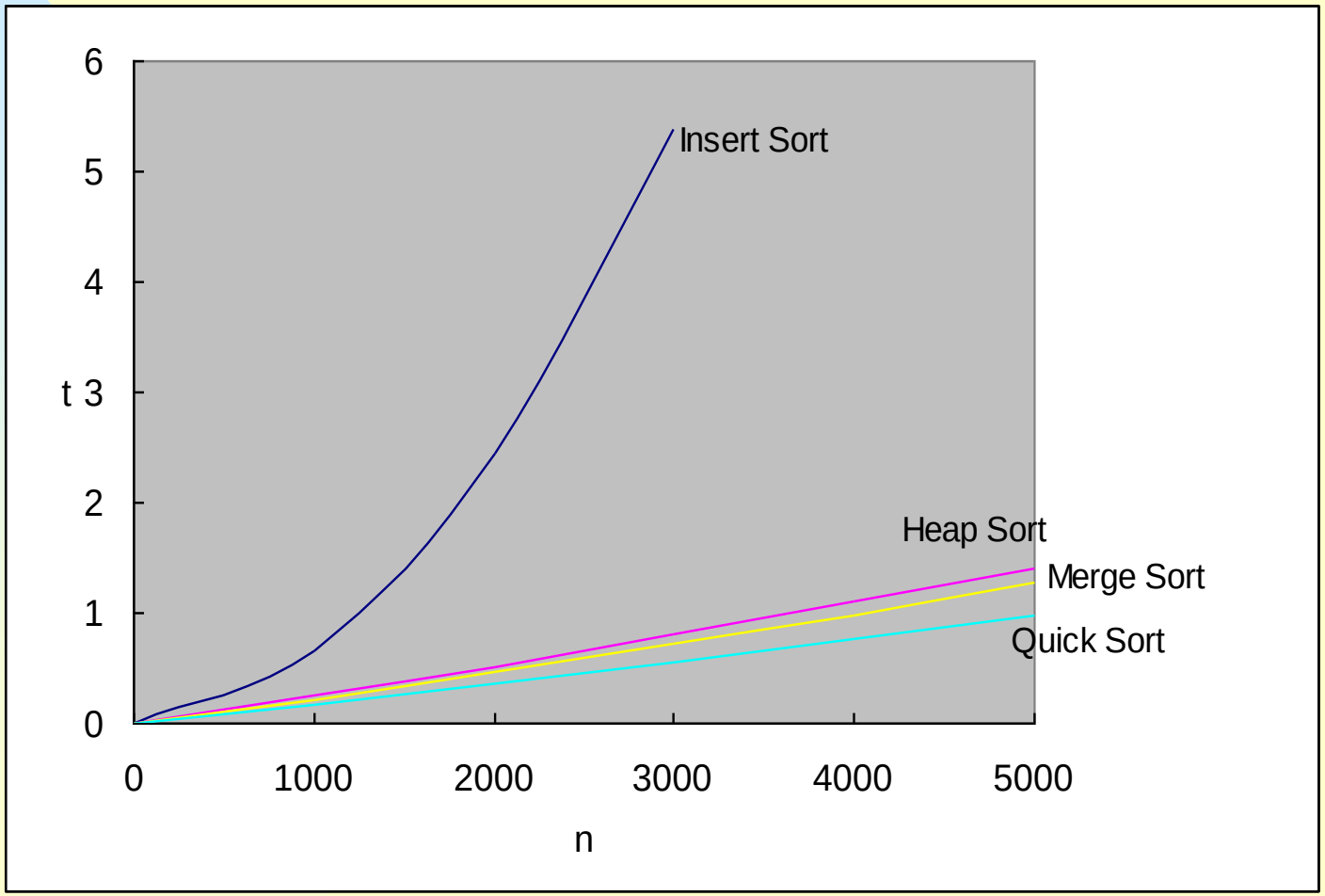
Method	Worst	Average	Working Storage
Insertion Sort	n^2	n^2	O(1)
Heap Sort	n log n	n log n	O(1)
Merge Sort	n log n	n log n	O(n)
Quick Sort	n^2	n log n	O(n) or O(log n)

The next slide gives the average runtimes for the 4 methods on a 1.7G Intel Pentium 4 PC with 512M RAM and Microsoft Visual.NET2003.

For each n at lest 100 randomly generated integer instances were run.

n	Insert	Heap	Merge	Quick
0	0.000	0.000	0.000	0.000
50	0.004	0.009	0.008	0.006
100	0.011	0.019	0.017	0.013
200	0.033	0.042	0.037	0.029
300	0.067	0.066	0.057	0.045
400	0.117	0.090	0.079	0.061
500	0.179	0.116	0.100	0.079
1000	0.662	0.245	0.213	0.169
2000	2.439	0.519	0.459	0.358
3000	5.390	0.809	0.721	0.560
4000	9.530	1.105	0.972	0.761
5000	15.935	1.410	1.271	0.970

The following is a plot of average times (milliseconds) for sort methods:



For average behavior, we can see:

- **Quick Sort outperforms the other sort methods for suitably large n.**
- **the break-even point between Insertion and Quick Sort is near 100, let it be nBreak.**
- **when n < nBreak, Insert Sort is the best, and when n > nBreak, Quick Sort is the best.**
- **improve Quick Sort by sorting sublists of less than nBreak records using Insertion Sort.**

Experiments: P435-4

7.10 External Sorting

7.10.1 introduction

- the lists are too large to be entirely contained in the internal memory, making an internal sort impossible.
- the lists to be sorted resides on a disk.
- block: the unit of data that is read from or written to a disk at one time.
- disk --- provide direct access to block, its I/O time is determined by:

(1) seek time

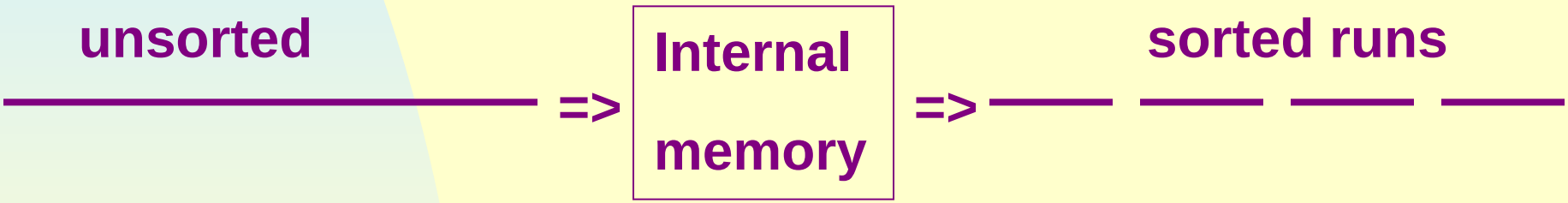
(2) latency time

(3) transmission time

Note I/O rate << CPU rate.

The most popular method for external sorting is merge sort, which consists of 2 phases:

(1) Segments of the input list are sorted in internal memory, these sorted segments, known as **runs**, are written onto disk as they are generated.



(2) The runs are merged together until only one run left.

Because Merge requires only the leading records of runs being merged to be present in memory at one time, it is possible to merge large runs together.

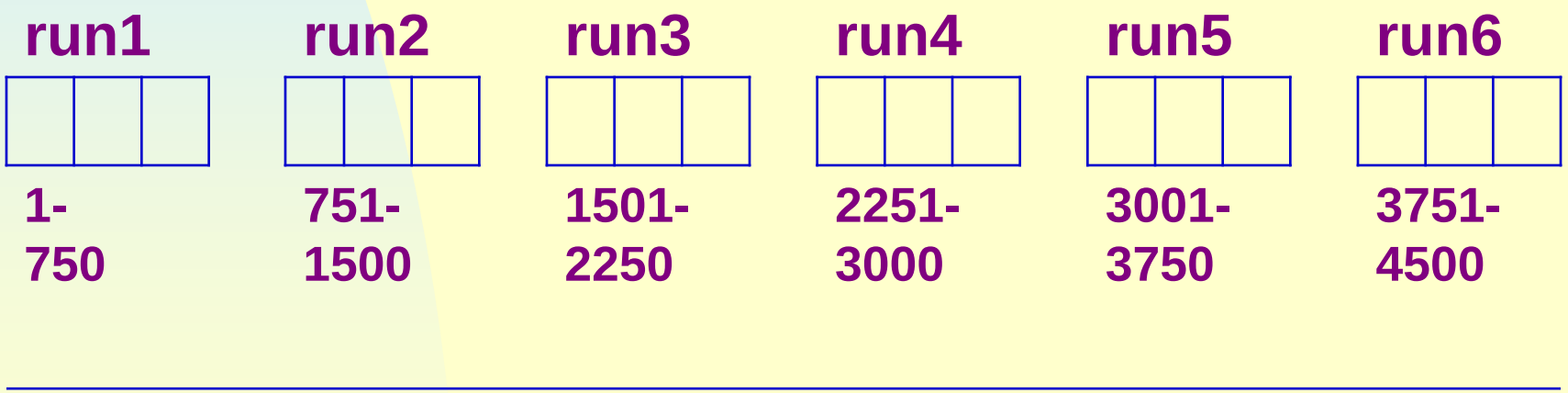
Example 7.12:

- **a list of 4500 records to be sorted**
- **the internal memory is capable of sorting at most 750 records**

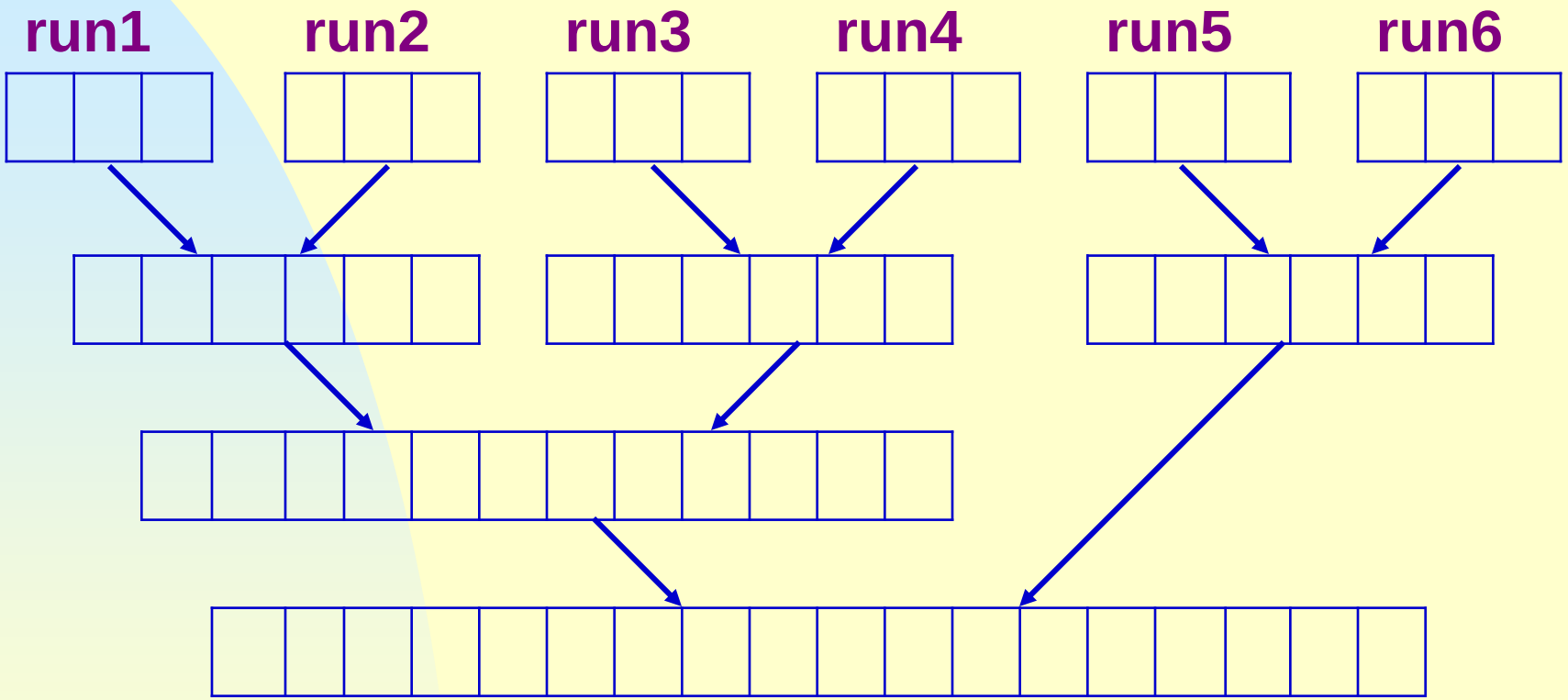
- **block length --- 250 records.**

Now we can sort the list as the following:

(1) Internally sort 3 blocks at a time to obtain 6 runs R_1 to R_6 .



(2) Merge the 6 runs.



Let

t_s = maximum seek time

t_l = maximum latency time

t_{rw} = time to read or write one block of 250 records

$t_{IO} = t_s + t_l + t_{rw}$

t_{IS} = time to internally sort 750 records

$n\ t_m$ = time to merge n records from input buffers to output buffer

The computing times are as:

operation	time
(1) read 18 blocks, internally sort, write 18 blocks	$36\ t_{IO} + 6t_{IS}$
(2) merge runs 1 to 6 in pairs	$36\ t_{IO} + 4500t_{m}$
(3) merge 2 runs of 1500 records each	$24\ t_{IO} + 3000t_{m}$
(4) merge 1 run of 3000 records with 1run of 1500	$36\ t_{IO} + 4500t_{m}$
total time	$132\ t_{IO} + 12000t_{m} + 6t_{IS}$

The computing time depends chiefly on the number of passes over the data.

We are mainly concerned with:

(1) Reduction of the number of passes by using a high-order merge.

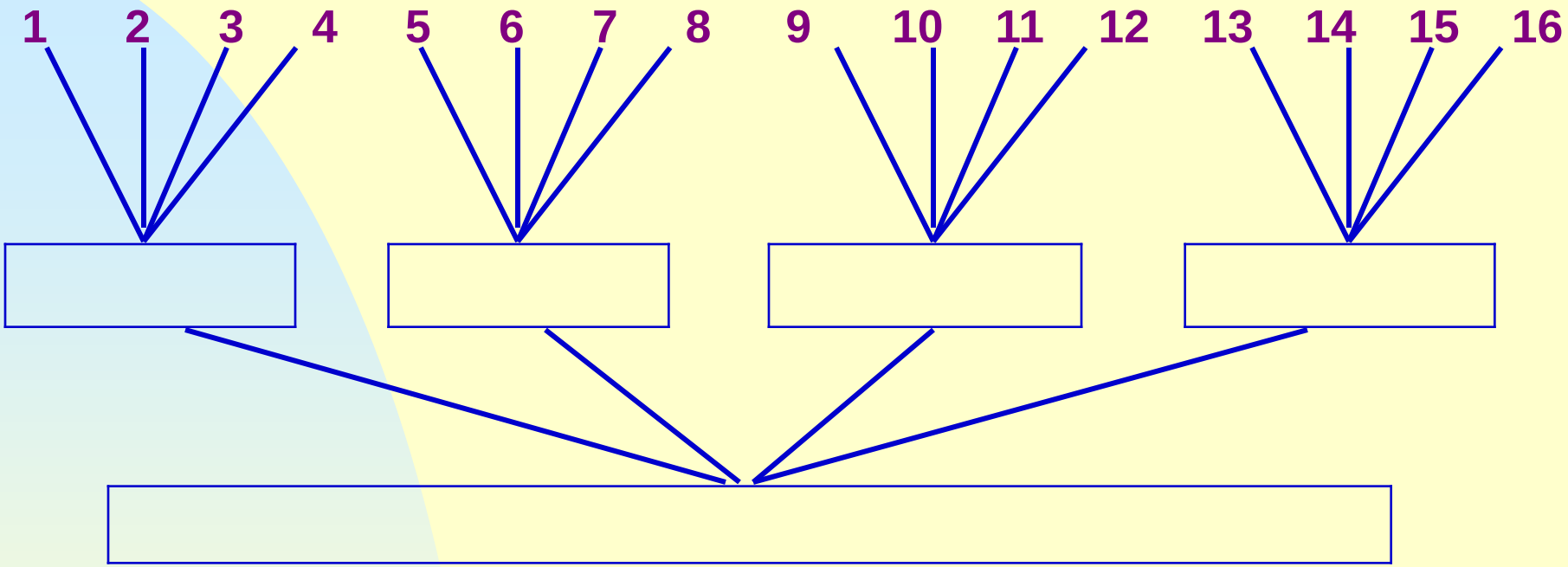
(2) An appropriate buffer-handling scheme to provide for parallel input, output and merging.

(3) Generating fewer runs (or equivalently longer) under the memory limitation.

7.10.2 k-way Merging

For 2-way merge with m runs, a total of $\lceil \log_2 m \rceil$ passes (excluding the one for generating the runs) are needed.

In general, a k-way merge on m runs requires at most $\lceil \log_k m \rceil$ passes over the data. Thus, I/O time may be reduced. As shown in the following tree:


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

For large k (say, $k \geq 6$), to select the smallest from the k possibilities, we can use a loser tree with k leaves.

The total time needed per level of the merge tree is $O(n \log_2 k)$.

The merge tree has $O(\log_k m)$ levels, the total internal merge time is

$$O(n \log_2 k \log_k m) = O(n \log_2 m)$$

which is independent of k!

Comments on high order merge:

(1) save I/O

(2) no big loss of internal processing speed.

(3) the decrease in I/O < indicated by reduction to $\log_k m$ passes, because:

- to do k-way merge we need at least k input buffers and 1 output buffer (or the best totally 2k+2).
- given fixed internal memory,

k increases => smaller buffer size => more blocks => increased seek and latency time.

- hence, beyond a certain k, I/O time increases despite the decrease in the number of passes.

7.10.4 Run Generation

Using a tree of loser, we can generate runs that are , on average, twice as long as obtainable by conventional internal sorting methods.

Background: given k record buffers

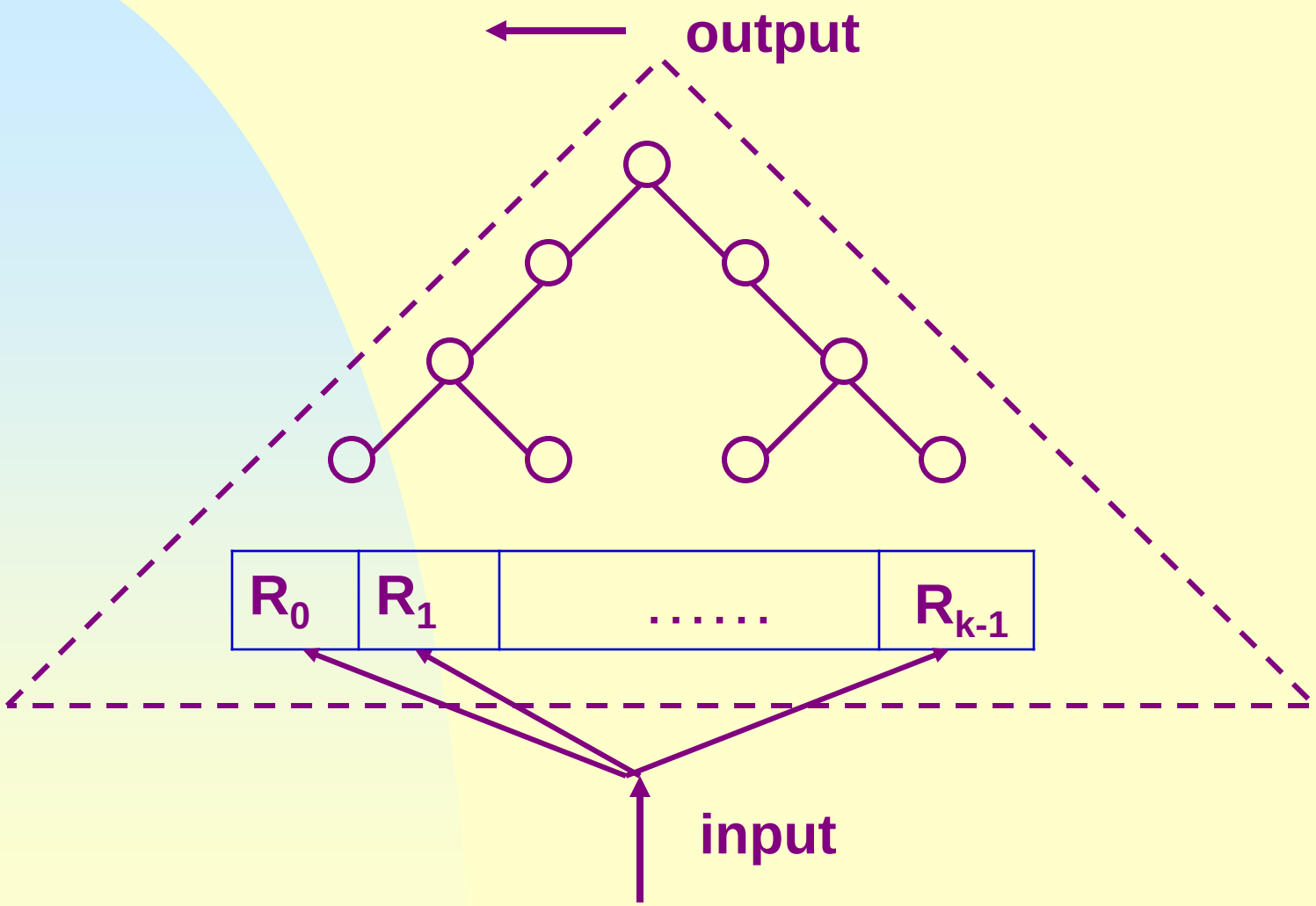
- **conventional method --- static**

Input list

R_0	R_1	.....	R_{k-1}

after finishing sorting on (R_0,…, R_{k-1}), generate k record long runs.

- **tree of loser method --- dynamic**



After one winner is selected, output it, and get a new record from the input list into the winner's buffer →longer runs, on average, 2k long.

Ideas:

- k record buffers r[i], $0 \le i < k$, as leaf nodes, node number=k+i
- k-1 non-leaf nodes numbered 1 to k-1, each non-leaf node i has only one field l[i], $1 \le i < k$, represents the loser of the tournament. l[i] contains the index of the loser's buffer.

- each r[i] has a run number field rn[i] associated with it to determine whether or not r[i] can be output as part of the run currently generated.
- rmax --- the max of the run number of the real records currently in memory.
- if input is empty, create a virtual record with rn=rmax+1, to push real records output before it.

Variable specification:

- r[i], $0 \le i < k$ --- the k records in the tree of loser

- l[i], 1≤i<k --- loser of the tournament played at node i
- l[0], q --- winner of the tournament
- rn[i], 0≤i<k --- the run number to which r[i] belongs
- rc --- current run number
- rq --- run number of r[q]
- rmax --- number of runs that will be generated
- lastRec --- last record output

```
template <class T>
1 void Runs (T *r )
2 {
3    r = new T[k];
4    int *rn=new int[k], *l=new int[k];
5    for (int i = 0; i < k; i++ )  {  //input records
6       ReadRecord(r[i]);  rn[i] = 1;
7    }
8    InitializeLoserTree();
9    int q = l[0];   // tournament winner
10   int rq=1, rc=1, rmax=1; T LastRec;
```

```
11  while (1) {          // output runs
12      if (rq != rc) {    // end of run
13          output end of run marker;
14          if (rq > rmax) break;  //meet virtual record, to line 35
15          else rc = rq;
16      }
17      WriteRecord(r[q]);  LastRec=r[q];
18      //input new record into tree
19      if (end of input) rn[q]=rmax+1; //generate virtual record
20      else {
21          ReadRecord(r[q]);
22          if (r[q]<LastRec)   //new record belongs to next run
23              rn[q] = rmax = rq+1;
24          else rn[q] = rc;
25      }
```

```
26      rq=rn[q];
27      // reconstruct the tree of losers
28     for (int t=(k+q)/2;t; t/=2) // t is initialized to be parent of q
29       if (rn[l[t]]<rq || rn[l[t]]==rq && r[l[t]]<r[q])
30       {  // t is the winner
31          swap(q, l[t]);
32          rq = rn[q];
33        }
34  }  // end of  while
35  delete [ ] r; delete [ ] rn;  delete [ ] l;
36}
```

Analysis of runs:

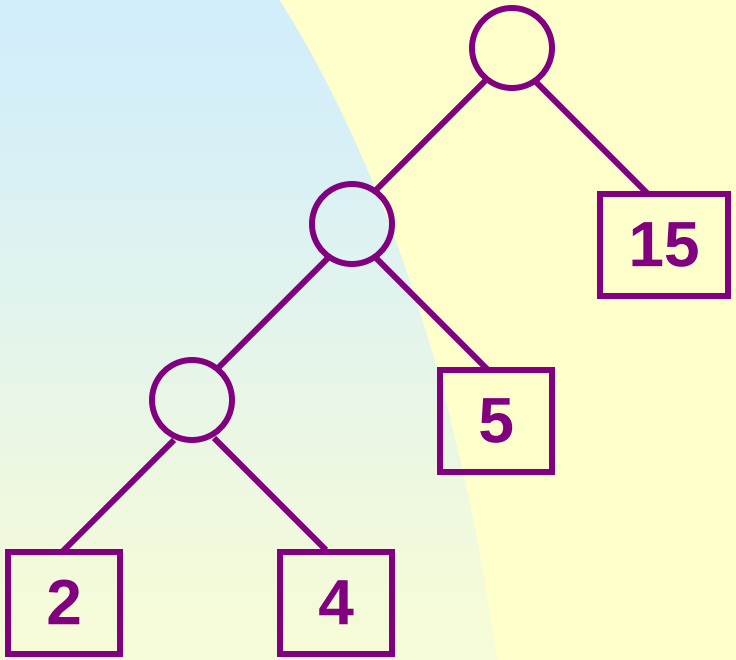
When the input list is sorted, only one run is generated. On average, the run size is 2k. The time required to generate all runs for an n record list is O(n log k).

7.10.5 Optimal Merging of Runs

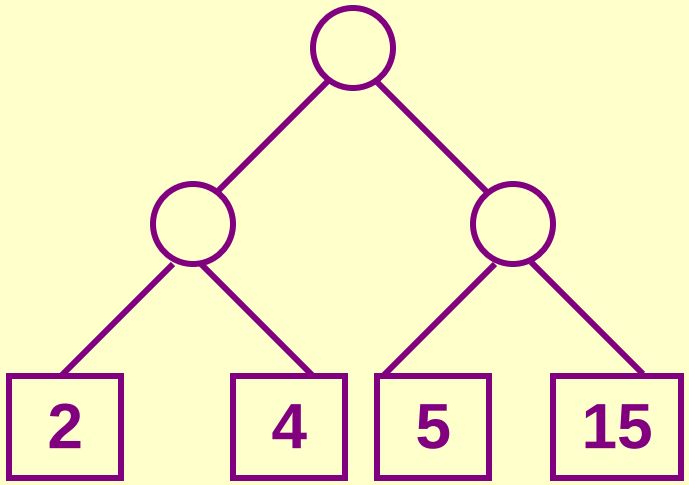
When runs are of different size, the merging strategy of making complete passes over all runs does not yield minimum run times.

For example, suppose we have 4 runs of length 2, 4, 5, and 15, respectively.

2 merge trees for merging the 4 runs:



(a)



(b)

- **internal nodes --- the circular nodes.**
- **external nodes --- the square nodes.**

Given n runs, there are n external nodes in the merge tree. Let the external nodes numbered 1 to n, q_i be the size of the run node i represents, d_i be the distance from the root to the external node i, $1 \le i \le n$. Then the total merge time is

$$E_w = \sum_{1 \le i \le n} q_i d_i$$

The E_w is called the weighted external path length.

The E_w

- of (a) is $2*3 + 4*3 + 5*2 + 15*1 = 43$
- of (b) is $2*2 + 4*2 + 5*2 + 15*2 = 52$

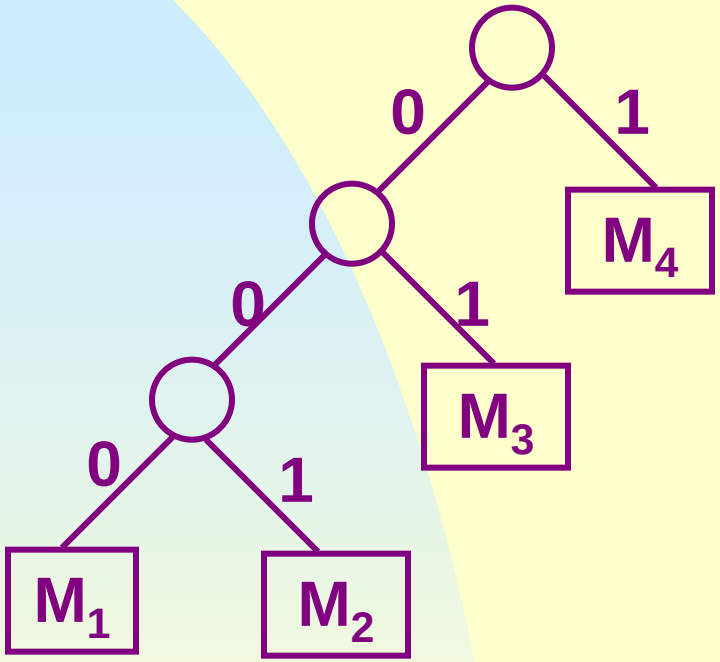
The cost of a k-way merge n runs of length q_i, $1 \le i \le n$, is **minimized** by using a merge tree of degree k that has minimum E_w.

We shall consider k=2 only. The result is easily generalized to the case k>2.

Another application for binary tree with minimum Ew is to obtain an optimal set of codes for messages $M_1,\ldots,M_{n+1}$.

Each code is a binary string that will be used for transmission of the corresponding message.

These codes are called Huffman codes.



q_i--- relative frequency for M_i.

d_i--- distance from the root to the external node for M_i.

The cost of decoding is

$$\sum_{1 \le i \le n+1} q_i d_i$$

Basic ideas of Huffman's solution :

- begin with a min heap of n single-node trees, which is an external node with the weight being one of the q_i's.
- repeatedly extract 2 minimum-weight **a** and **b** from the min heap, combine them into a single binary tree **c**, and insert c into the min heap.
- after n-1 rounds, the min heap will be left with a single binary tree, which is the wanted.

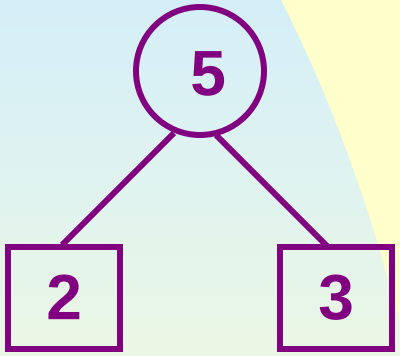
Assume:

- **the function Huffman is a friend of TreeNode.**
- **use the data field of a TreeNode to store the weight of the binary tree rooted at that node.**

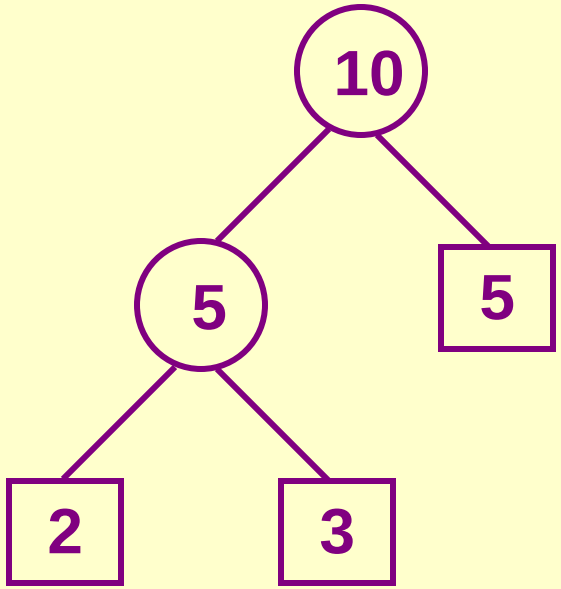
```
template <class T>
void Huffman(MinHeap<TreeNode<T> *> heap, int n)
{ // heap is initially a min heap of n single-node binary trees.
    for (int i=0; i<n-1; i++) {//loop n-1 times to combine n nodes
      TreeNode<T> *a=heap.Top();
      heap.Pop();
      TreeNode<T> *b=heap.Top();
       heap.Pop();
      TreeNode<T> *c=new TreeNode(a→data+b→data,
                                  a, b);  // refer section 5.2.3.2
       heap.Push(c);
    }
}
```

Example 7.15:

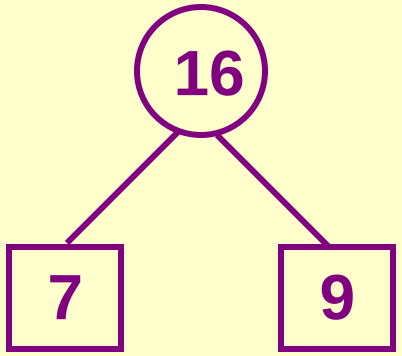
Suppose we have the weights q_1=2, q_2=3, q_3=5, q_4=7, q_5=9, and q_6=13, then the sequence of trees we get is as the following:



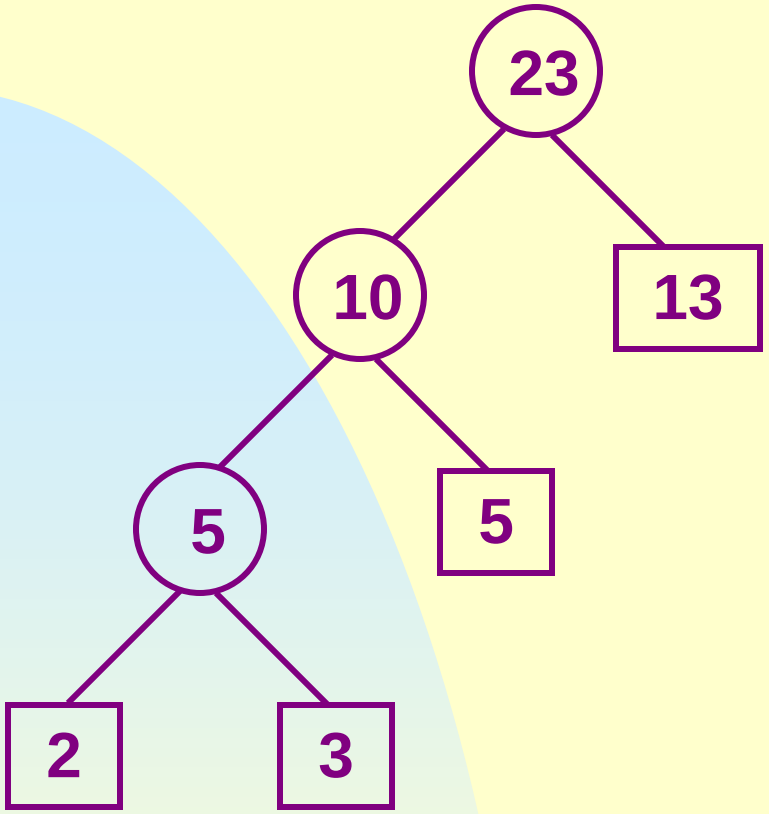
(a) q_1+q_2



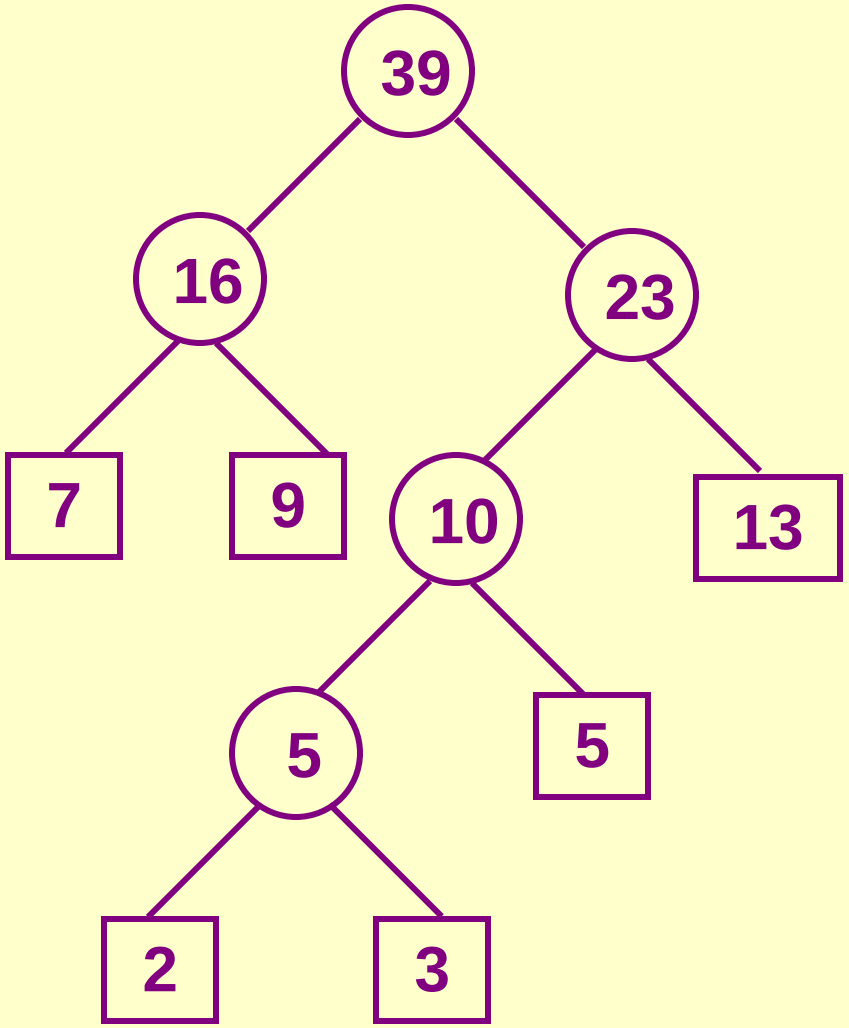
(b) (a)+q_3



(c) q_4+q_5



(d) (b)+q_6



(e) (c)+(d)

Analysis of huffman:

- the main loop --- n-1 times
- each call to Top is O(1), to Pop and Push is O(log n)

The total time: O(n log n).

Exercises: P457-2

Chapter 8
Hashing

8.1 Introduction

Let look back to the ADT Dictionary, which is used in many applications:

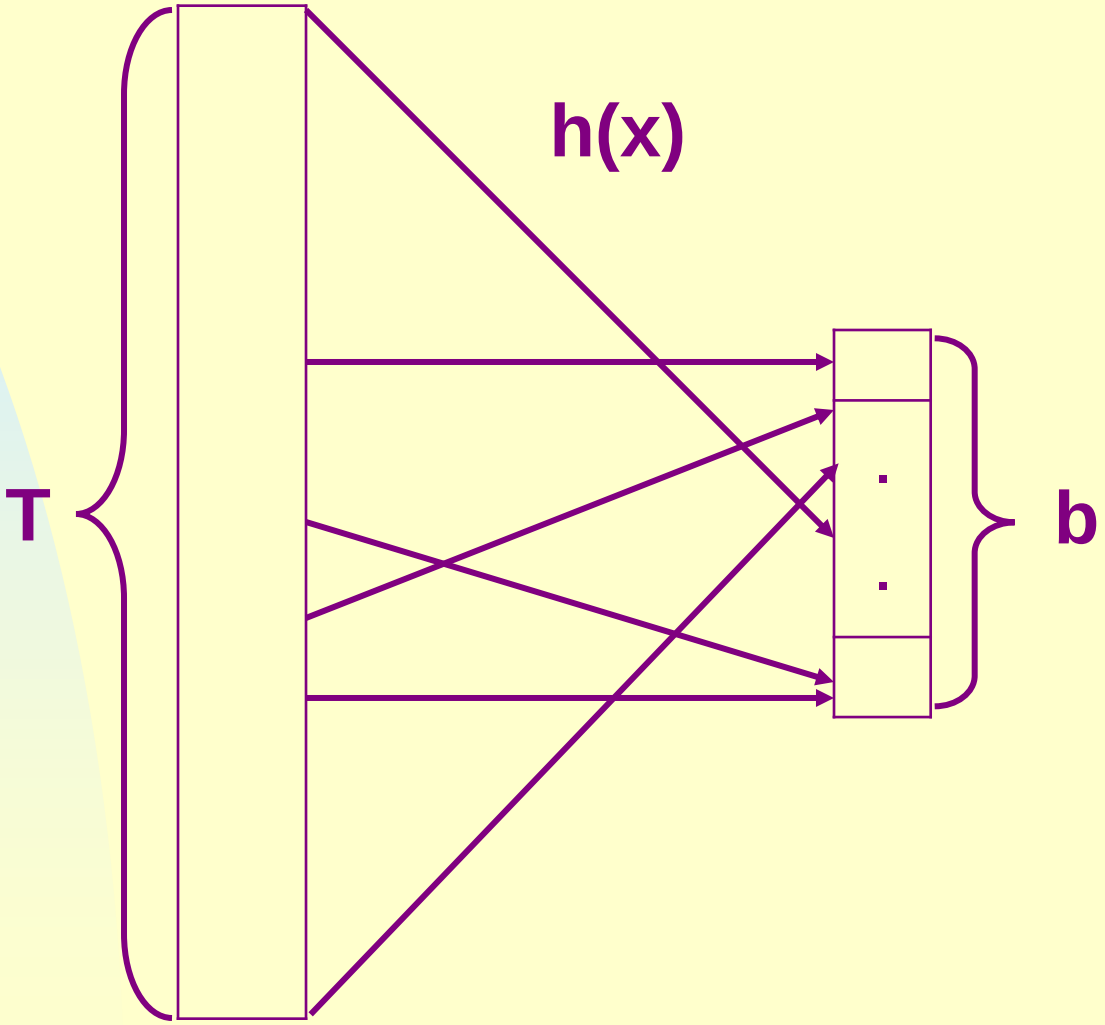
spelling checker, the thesaurus, the index for a database, the symbol tables generated by loaders, assemblers, and compilers, …

ADT 5.3

```
template <class K, class E>
class Dictionary {
public:
   virtual bool IsEmpty () const = 0;
      // return true iff the dictionary is empty
   virtual pair<K,E>* Get(const K&) const = 0;
      // return pointer to the pair with specified key;
      // return 0 if no such pair
   virtual void Insert(const pair<K,E>&) = 0;
      // insert the given pair; if key is a duplicate
      // update associated element
   virtual void Delete(const K&) = 0;
      // delete pair with specified key
};
```

- **The 3 basic operations on Dictionary are Get, Insert and Delete.**
- **In key comparison based search tree methods, these operations take O(h) time, where h is the height of the tree.**
- **If we directly use the key as an address, then we can perform these operations in O(1).**
- **However, the space for key definition is too big, we need to map the big space to a space of practical size.**

This leads to the technique of hashing, in which a hash function is used to map a big space to a small one:



8.2 Static Hashing

8.2.1 Hash Tables

In static hashing the dictionary pairs are stored in a table, ht, called hash table.

- **ht is partitioned into b buckets: ht[0:b-1], and maintained in sequential memory.**
- **each bucket holds s slots, each slot holds one pair, usually, s = 1.**
- **the address of a pair with key k is determined by a hash function h, h(k) is the hash or home address of k, h(k)∈{0, 1,…,b-1}.**

T --- the total number of possible keys.

n --- the number of pairs in the hash table.

Definition:

The key density of a hash table is the ratio n/T.

The loading density (or factor) of a hash table is $\alpha=n/(sb)$.

Usually, n<<T, and b<<T.

- 2 keys k_1 and k_2 are said to be **synonyms** with respect to h if $h(k_1) = h(k_2)$.
- a **collision** occurs when the home bucket for the new pair is not empty.
- an **overflow** occurs when a new pair is hashed into a full bucket.
- when s=1, collisions and overflows occur simultaneously.

Example 8.1:

Let b=26, s=2, n=10, hence α = 10/52 = 0.19

Assume the internal representation for A to Z corresponds to 0 to 25, then

h(k) = the first character of k

Keys: GA, D, A, G, L, A2, A1, A3, A4, and E.

The following shows the keys GA, D, A, G, L and A2 entered into the hash table.

ht	Slot 1	Slot 2
0	A	A2
1		
2		
3	D	
4		
5		
6	GA	G
.	.	.
.	.	.
25		

Next, A1 hashes into ht[0] --- overflow.

If no overflow, the time required to insert, delete or search using hash depends only on the time for computing the hash function and searching one bucket .

--- independent of n.

Since T>>b, it is impossible to avoid overflow, a mechanism to handle overflows is necessary.

8.2.2 Hash Functions

A hash function maps a key into a bucket address in the hash table.

It should be easy to compute and minimize the number of collisions.

If k is a key chosen at random from the key space, then we want the probability that h(k)=i to be 1/b for all buckets i.

A hash function satisfying this property is a uniform hash function.

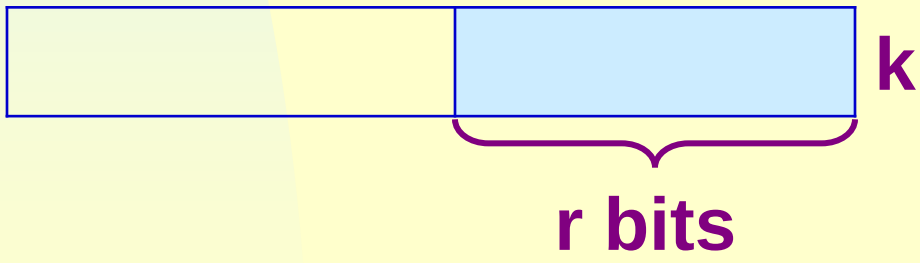
8.2.2.1 Division

h(k) = k % D b = D

The choice of D is critical.

For instance:

(1) If $D=2^r$, then h(k) depends only on the last r bits of k:



(2) If D is divisible by 2, then odd keys are mapped to odd buckets, even keys to even buckets.

In practice, it is sufficient to choose D such that it has no prime divisors less than 20.

8.2.2.2 Mid-Square

h(k) is computed by using an appropriate number of bits from the middle of k^2 to obtain the bucket address.

If r bits used, $b= 2^r$.

8.2.2.3 Folding

k is partitioned into several parts, all but the last being of the same length. These partitions are then added together to obtain the hash address for k.

Example 8.2:

k=12320324111220 is partitioned into parts that are 3 decimal digits long.

P_1=123, P_2=203, P_3=241, P_4=112, P_5=20.

- shift folding

  h(k)=123+203+241+112+20=699

- folding at the boundaries

  h(x)=123+302+241+211+20=897

8.2.2.4 Digit Analysis

Useful in the case of a static file.

- **each k is interpreted as a number using some radix r.**
- **the digits of each k are examined, digits having the most skewed distribution are deleted --- until the number of digits left is small enough to give an address.**

For example, consider students numbers:

7	1	1	1	4	1	0	1
7	1	1	1	4	1	0	2
7	1	1	1	4	1	0	3
					.		
					.		
7	1	1	1	4	4	2	7
7	1	1	1	4	4	2	8
7	1	1	1	4	4	2	9

Let b=1000, we use the last 3 digits.

8.2.4 Overflow Handling

8.2.4.1 Open Addressing

Assume:

- **the hash table, ht, is an array, and stores pointers to dictionary pairs.**
- **for simplicity, s=1.**

The following are the class definitions:

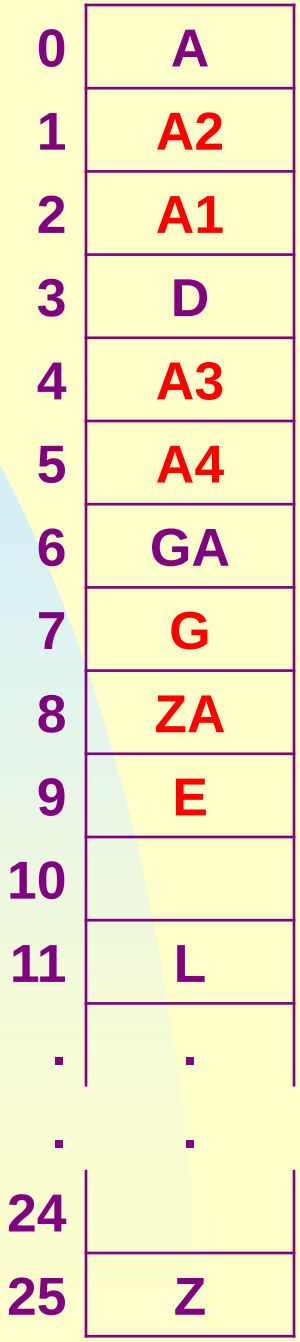
```
template<class K, class E>
typedef pair<K, E> * pairPtr;
class LinearProbing {
public:
    LinearProbing (int size) {
        if (size <3) throw “the size must be >= 3.”;
        b=size;
        ht=new pairPtr[b];
        fill(ht, ht+b,0);  // initialize the hash table to empty.
    };
private:
    int b;
    pairPtr *ht;
};
```

When overflow occurs, we need to find another bucket for the new identifier, the simplest solution is to find the closest bucket.

This is called linear probing or linear open addressing.

Example 8.6:

- b=26, ht is used circularly.
- keys: GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E.
- h(x)=first character of x, e.g., h(GA)=6.
- Initially, all entries in ht are null.



After entering

GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E,

we get the table shown left.

When linear open addressing is used to handle overflows, a hash table search for key k proceeds as follows:

(1) Compute h(k)

(2) Examine ht[h(k)], ht[h(k)+1%b], …, ht[h(k)+j%b] until one of the following happens:

(a) ht[h(k)+j%b] has a pair whose key is k---found.

(b) ht[h(x)+j%b] is empty---not in.

(c) return to the starting position h(k)---the table is full and k is not in.

```
template <class K, class E>
pair<K, E>* LinearProbing<K, E>::Get(const K& k)
{ // search the linear probing hash table ht (s=1) for k.
  // If found return a pointer to the pair, else return 0.
    int i=h(k);  // home bucket
    int j;
    for (j=i; ht[j] && ht[j]→first !=k;) {
       j = (j+1) % b;    // treat the table as circular
       if ( j == i ) return 0;  // back to the start point
    }
    if (ht[j]→first==k) return ht[j];
    return 0;
}
```

Analysis shows that the expected average number of key comparisons to look up a key is approximately

$$(2- \alpha) / (2- 2\alpha)$$

where α is the loading density.

In Example 8.6, $\alpha = 12 / 26 = 0.47$ and the average key comparisons is 1.5.

Although the average number of comparisons is small, the worst case can be quite large.

Some other open addressing methods include:

- **quadratic probing**
- **rehashing**
- **random probing**

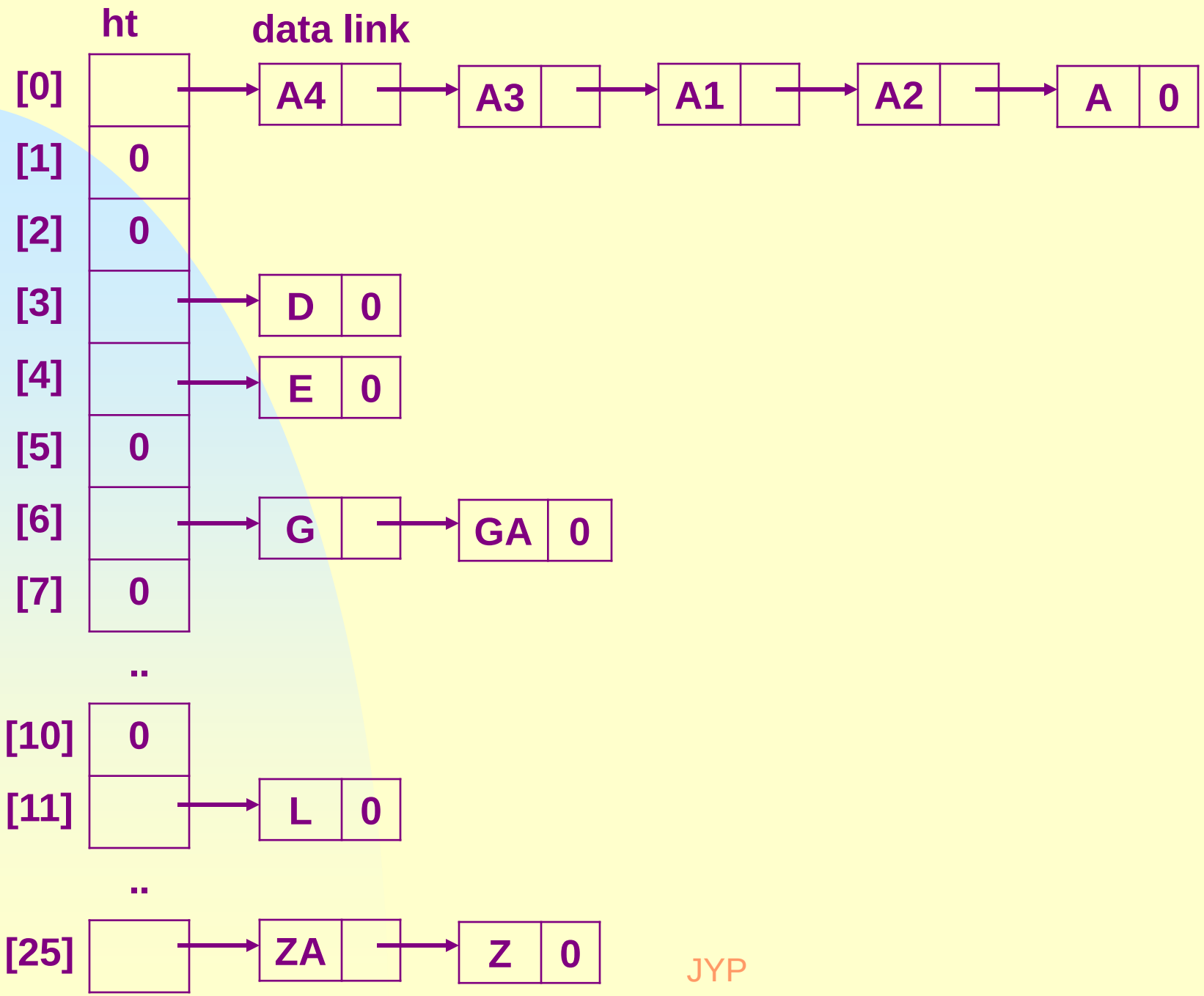
8.2.4.2 Chaining

Linear probing performs poorly because the search for a key involves comparison with keys having different hash values, making a local collision a global one .

Many of the comparisons can be saved if we maintain lists of keys, one list per bucket, each list containing all the synonyms for that bucket.

The lists are most frequently maintained as chains.

When chaining is used for the data of Example 8.6, we obtain the hash chains as in the next slide.


ht
data link
[0]
A4
A3
A1
A2
A
0
[1]
0
[2]
0
[3]
D
0
[4]
E
0
[5]
0
[6]
G
GA
0
[7]
0
..
[10]
0
[11]
L
0
..
[25]
ZA
Z
0

```
template <class K, class E>
typedef ChianNode<pair<K, E>>* ChainNodePtr;
class Chaining {   // assume Chaining is friend of ChainNode
public:
    Chaining(int size) {
        if (size <3) throw “the size must be >= 3.”;
        b=size;
        ht=new ChainNodePtr[b];
        fill(ht, ht+b,0);  // initialize the hash table to empty.
    };
private:
    int b;
    ChainNodePtr *ht;
};
```

```
template <class K, class E>
Pair<K, E>* Chaining<K, E>::Get(const K& k)
{ // Search the chained hash table ht for k. If a pair with key k
  // is found return a pointer to this pair, else return 0.
    int i = h(k);   // home bucket
    // search the chain starting at ht[i]
    for (ChainNode<pair<K, E>>* current=ht[i]; current;
                                          current=current→link)
       if (current→data.first==k) return &current→data;
    return 0;
}
```

The expected average number of key comparisons of a successful search is

$$\approx 1 + \alpha / 2$$

where α is the loading density.

The table in the next slide presents the results of empirical study. The values in each column give the average number of key comparisons.

$\alpha = n/b$	0.50		0.75		0.90		0.95	
Hash function	Chain	Open	Chain	Open	Chain	Open	Chain	Open
mid square	1.26	1.73	1.40	9.75	1.45	37.14	1.47	37.53
division	1.19	4.52	1.31	7.20	1.38	22.42	1.41	25.79
shift fold	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
bound fold	1.39	22.97	1.57	48.70	1.55	69.63	1.51	97.56
digit analysis	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
theoretical	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

As expected, Chaining outperforms Linear open addressing for overflow handling and Division is superior to other hash functions.

Exercises: P475-3, 6

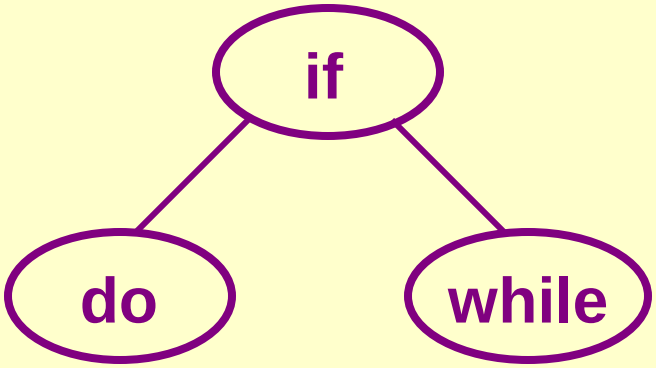
Chapter 10
Search Structures

10.1 Optimal Binary Search Trees

In this section, we consider the construction of binary search trees for a static set of identifiers. That is, we make neither additions nor deletions from the set. Only searches are performed.

Given n identifiers, we can construct a binary search tree equivalent to a **complete** binary search tree for them. Using the function *search* (Program 5.21), the worst search time is $\log_2 n$. If the identifiers are to be searched with equal probability, it is optimal.

For instance, the complete binary search tree for the file (do, if, while) is:

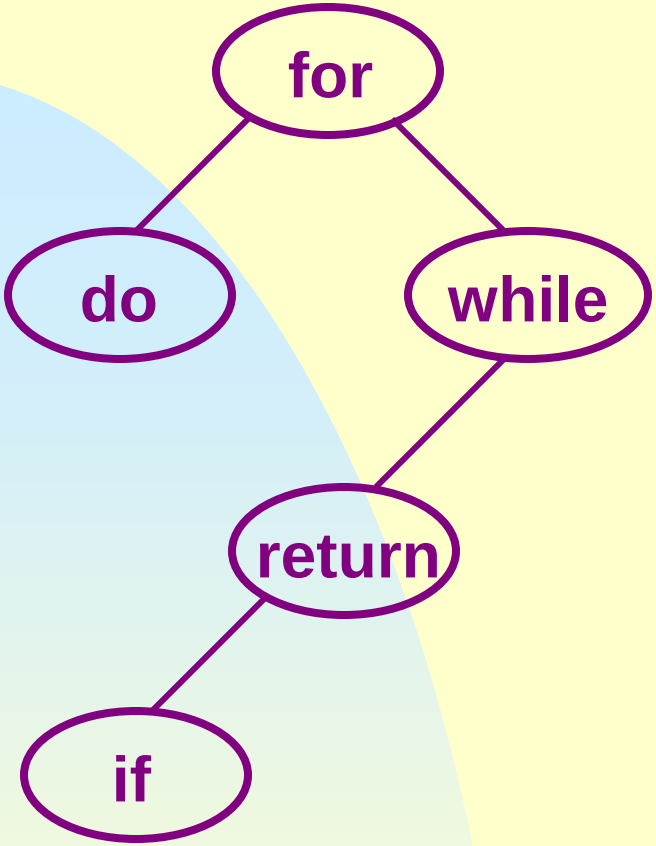


But it may not be the optimal binary search tree to use when different identifiers are to be searched with different probabilities.

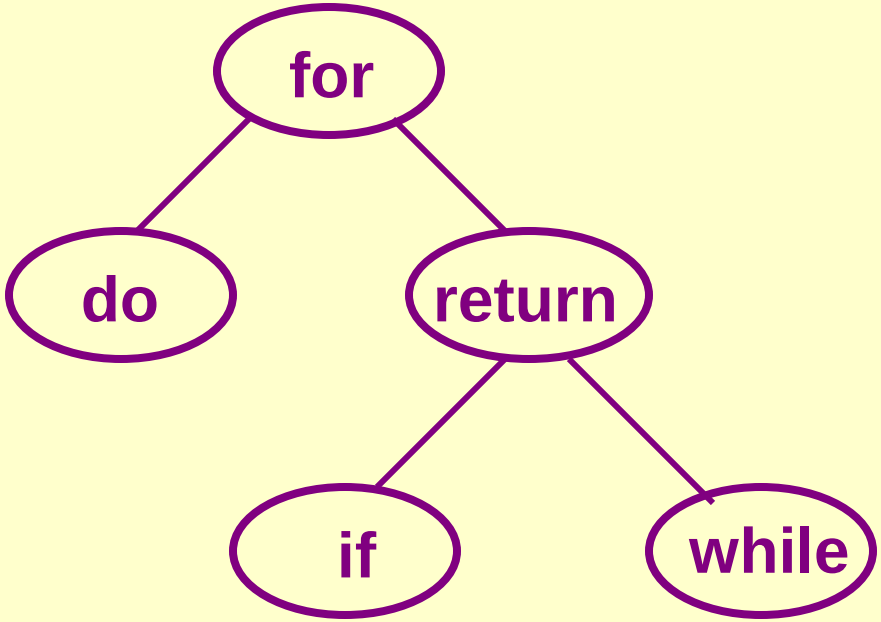
To find an optimal binary search tree for a given static file, we must first decide on a cost measure for search trees.

In the binary search tree *search algorithm*, an identifier at level k needs k iterations to find. Thus it is reasonable to use the level number of a node as its cost.

Consider 2 search trees of Fig. 10.2.



(a)



(b)

Fig. 10.2 Two binary search trees

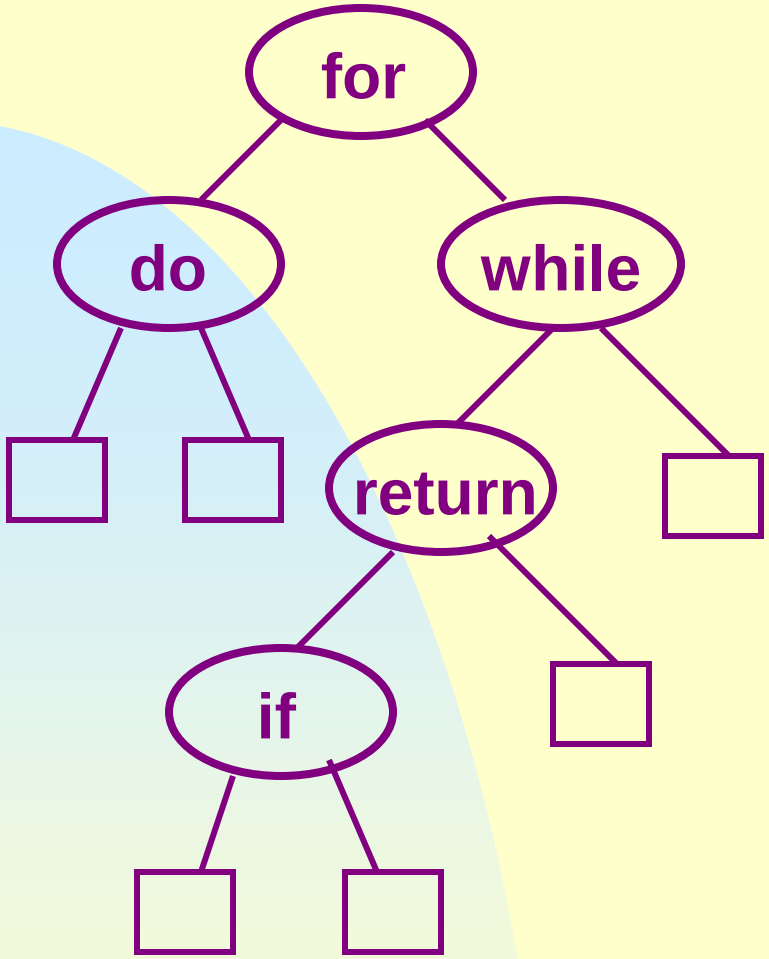
- **worst case: tree(a) requires 4 comparisons, tree(b) requires 3 comparisons, (b) is better.**
- **average case:**
  - **Tree(a): 1 comparison---for, 2---do, 2---while, 3---return, 4---if. With equal probability, the average number of comparisons for a successful search is (1+2+2+3+4)/5=2.4 .**
  - **Tree(b): (1+2+2+3+3)/5=2.2**

(b) is better too.

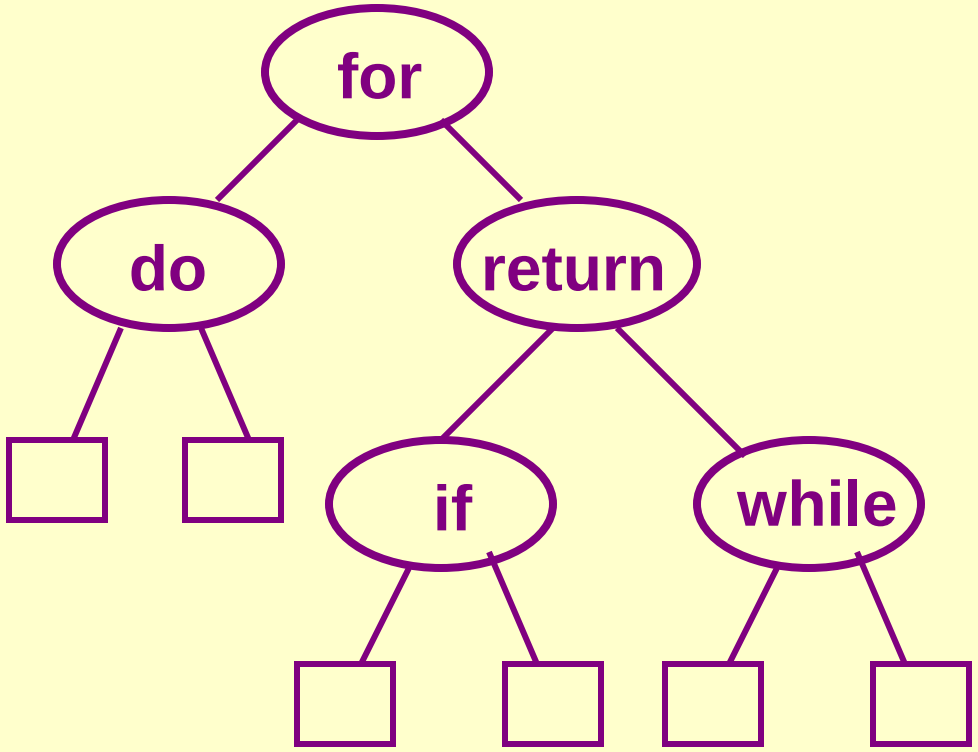
To reflect the cost of unsuccessful search, it is useful to add a special "square" node at every null link.

Doing this to the trees of Fig. 10.2 yields the trees of Fig. 10.3.

Binary tree with n nodes has n+1 null links and therefore will have n+1 square nodes.



(a)



(b)

Fig. 10.3 Extended binary trees corresponding to search trees of Fig. 10.2

- **external (failure) nodes---square nodes.**
- **internal nodes---original nodes.**
- **extended binary tree---a binary tree with external nodes added.**
- **external path length, E, of a binary tree---the sum over all external nodes of the lengths of the paths from the root to those nodes.**
- **internal path length, I, of a binary tree---the sum over all internal nodes of the lengths of the paths from the root to those nodes.**

For Fig. 10.3(a):

I=0+1+1+2+3=7, E=2+2+4+4+3+2=17

In general, E=I+2n.

Hence, binary trees with the maximum E also have maximum I.

Over all binary trees with n internal nodes:

- **worst case: when the tree has a depth of n.**

$$I = \sum_{i=0}^{n-1} i = n(n-1)/2$$

this is the maximum I.

• optimal case: as many internal nodes as close to the root as possible---complete binary tree.

I=0+2*1+4*2+8*3+……

If we number the nodes in a complete binary tree as before, the distance of node i from the root is

$$\lfloor \log_2 i \rfloor$$

so

$$I = \sum_{1 \le i \le n} \lfloor \log_2 i \rfloor = O(n \log_2 n)$$

Now return to the problem of finding an optimal binary search tree for a set of identifiers to be searched with different probabilities.

If the binary search tree contains the identifiers $a_1, a_2, \ldots\ldots, a_n$ with $a_1 < a_2 < \ldots\ldots < a_n$, and the probability of searching for each a_i is p_i, then the total cost of any binary search tree is

$$\sum_{1 \le i \le n} p_i \,.\, \text{level}(a_i)$$

when only **successful** searches are made.

The cost of **unsuccessful** searches should also be included.

Unsuccessful searches terminate with algorithm *search* return a 0 pointer. Every node with a null subtree defines a point at which such a termination can take place. Replace every null subtree by a failure node.

Identifiers not in the binary search tree may be partitioned into n+1 classes, E_i, $0 \le i \le n$.

$E_0 = \{x \mid x < a_1\}$

$E_i = \{x \mid a_i < x < a_{i+1}\}, \quad 1 \le i < n.$

$E_n = \{x \mid x > a_n\}$

For all x in E_i, $0 \le i \le n$, the search terminates at the same failure node i, and it terminates at different failure nodes for x in different classes.

If q_i is the probability for E_i, the cost of the failure nodes is

$$\sum_{0 \le i \le n} q_i.(\text{level(failure node i)}-1)$$

Therefore the total cost is

$$\sum_{1 \le i \le n} p_i.\text{level}(a_i) + \sum_{0 \le i \le n} q_i.(\text{level(failure node i)}-1) \quad (10.1)$$

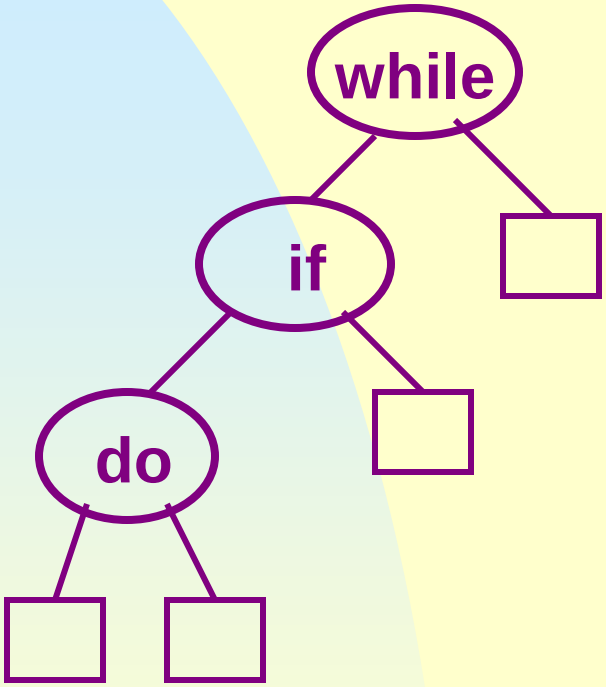
An optimal binary search tree for the identifier set $a_1, a_2, \ldots\ldots, a_n$ is the one that minimizes Eq. (10.1) over all possible binary search trees for this identifier set.

Note

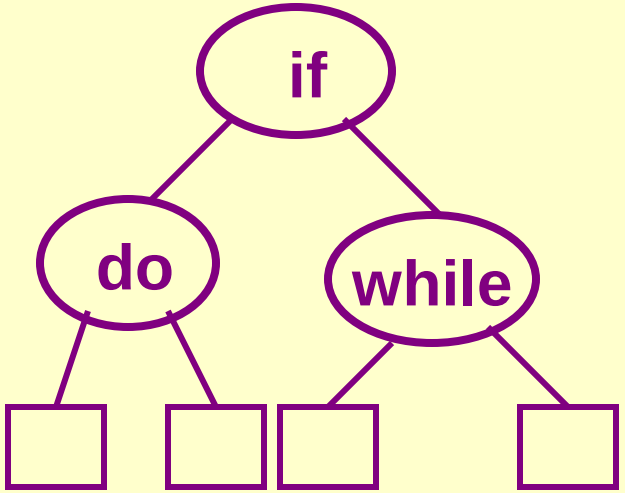
$$\sum_{1\le i\le n} p_i + \sum_{0\le i\le n} q_i = 1$$

Example 10.1:

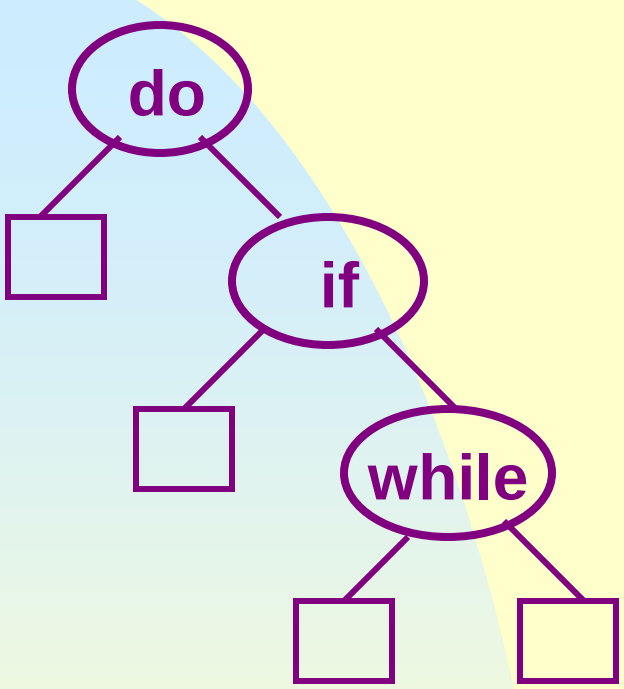
(a_1, a_2, a_3)=(do, if, while). The possible binary search tree for it are as follows:



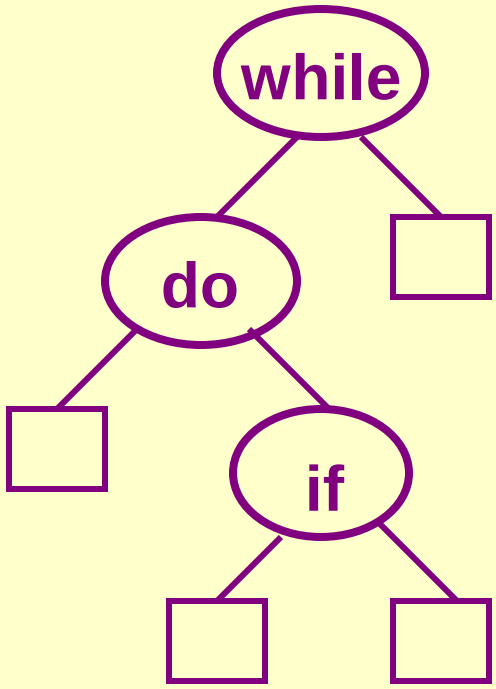
(a)



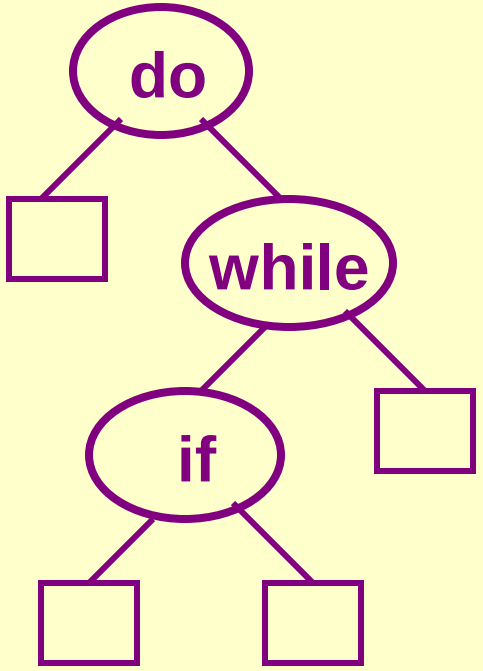
(b)



(c)



(d)



(e)

- with equal probabilities, $p_i=q_j$ for all i and j.

  cost(tree a)=15/7; cost(tree b)=13/7

  cost(tree c)=15/7; cost(tree d)=15/7

  cost(tree e)=15/7

  As expected, tree b is optimal.

- with $p_1=0.5$, $p_2=0.1$, $p_3=0.05$, $q_0=0.15$, $q_1=0.1$, $q_2=0.05$, $q_3=0.05$

  cost(tree a)=2.65; cost(tree b)=1.9

  cost(tree c)=1.5; cost(tree d)=2.15

  cost(tree e)=1.6

For instance,

cost(tree d) =0.05+2*0.5+3*0.1

+2*0.15+3*0.1+3*0.05+0.05

=2.15 (note the error in P557)

tree c is optimal.

How to determine the optimal tree?

One way: explicitly generate all possible binary search trees, compute the cost of each tree in O(n), then determine the optimal---need

$$O\left(\frac{n}{n+1}C_n^{2n}\right)=O\left(C_n^{2n}\right)$$ --- no good.

Instead, we can get an efficient algorithm by making use of the properties of optimal binary search trees(b.s.t.).

Let $a_1<a_2<,\ldots\ldots,<a_n$ be n identifiers in a b.s.t., denote:

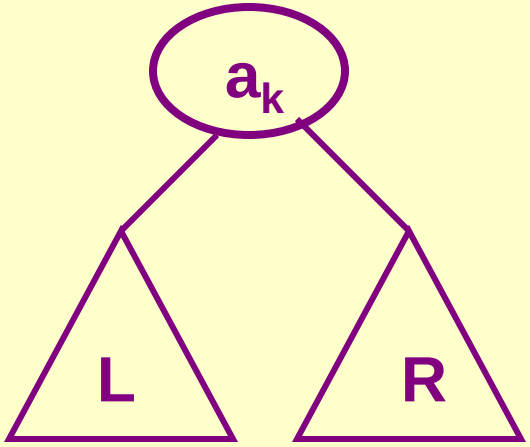
- T_{ij}---an optimal b.s.t. for $a_{i+1},\ldots\ldots, a_j$, $i<j$. T_{ii} is an empty tree, $0\le i\le n$. T_{ij} $(i>j)$ is not defined.
- c_{ij}---the cost of T_{ij}. $c_{ii}=0$.
- r_{ij}---the root of T_{ij}.
- w_{ij}---the weight of T_{ij}. $w_{ij}= q_i+ \sum_{k=i+1}^{j} (q_k+ p_k)$.

- $r_{ii}=0$, $w_{ii}=q_i$, $0 \le i \le n$.
- The optimal b.s.t. for $a_1, \ldots\ldots, a_n$ is T_{0n}. Its cost is c_{0n}, root r_{0n}.

If T_{ij} is an optimal b.s.t. for $a_{i+1}, \ldots\ldots, a_j$ and $r_{ij}=k$, $i<k \le j$, then T_{ij} has 2 subtrees L and R.

L contains the identifiers $a_{i+1}, \ldots\ldots, a_{k-1}$.

R contains the identifiers $a_{k+1}, \ldots\ldots, a_j$.



The cost c_{ij} of T_{ij} is

$$c_{ij}=p_k+cost(L)+cost(R)+w_{i,k-1}+w_{kj} \qquad (10.2)$$

From (10.2), if c_{ij} is to be minimal, then cost(L)=$c_{i,k-1}$ and cost(R)=c_{kj} , as otherwise we could replace either L or R by a subtree with a lower cost, thus generating a b.s.t. for a_{i+1},……, a_j with a lower cost than c_{ij}. This violates the assumption that T_{ij} is optimal.

Hence, E.q.(10.2) becomes

$$c_{ij}=p_k+w_{i,k-1}+w_{kj}+c_{i,k-1}+c_{kj}$$

$$=w_{ij}+c_{i,k-1}+c_{kj} \qquad (10.3)$$

Since T_{ij} is optimal, from (10.3) we know $r_{ij}=k$ is such that

$$w_{ij}+c_{i,k-1}+c_{kj} = \min_{i<l\le j}\{w_{ij}+c_{i,l-1}+c_{lj}\}$$

or

$$c_{i,k-1}+c_{kj} = \min_{i<l\le j}\{c_{i,l-1}+c_{lj}\} \qquad (10.4)$$

Note also

$$w_{ij}=w_{i,j-1}+p_j+q_j .$$

(10.4) gives us a means of obtaining r_{0n}, T_{0n} and c_{0n}, starting from $r_{ii}=0$, $T_{ii}=\varnothing$ and $c_{ii}=0$.

Example 10.2:

Let $n=4$, $(a_1, a_2, a_3, a_4)=$(do, if, return,while).

Let $(p_1, p_2, p_3, p_4)=(3,3,1,1)$

and $(q_0, q_1, q_2, q_3, q_4)=(2,3,1,1,1)$.

The p's and q's are as integer for convenience.

Initially, $w_{ii} = q_i$, $r_{ii}=0$, $c_{ii}=0$, $0 \le i \le 4$.

Using (10.3) and (10.4), we get

$w_{01} = p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 8$

$c_{01} = w_{01} + \min \{c_{00} + c_{11}\} = 8$

$r_{01} = 1$

$w_{12} = p_2 + w_{11} + w_{22} = p_2 + q_2 + w_{11} = 7$

$c_{12} = w_{12} + \min \{c_{11} + c_{22}\} = 7$

$r_{12} = 2$

$w_{23} = p_3 + w_{22} + w_{33} = p_3 + q_3 + w_{22} = 3$

$c_{23} = w_{23} + \min \{c_{22} + c_{33}\} = 3$

$r_{23} = 3$

$w_{34} = p_4 + w_{33} + w_{44} = p_4 + q_4 + w_{33} = 3$

$c_{34} = w_{34} + \min \{c_{33} + c_{44}\} = 3$

$r_{34} = 4$

Knowing $w_{i,i+1}$, $c_{i,i+1}$, we can use EQ.(10.3) and (10.4) to compute $w_{i,i+2}$, $c_{i,i+2}$, $r_{i,i+2}$:

$w_{02} = p_2 + q_2 + w_{01} = 3 + 1 + 8 = 12$

$c_{02} = w_{02} + \min \{c_{00} + c_{12} , c_{01} + c_{22}\} = 12+7=19$

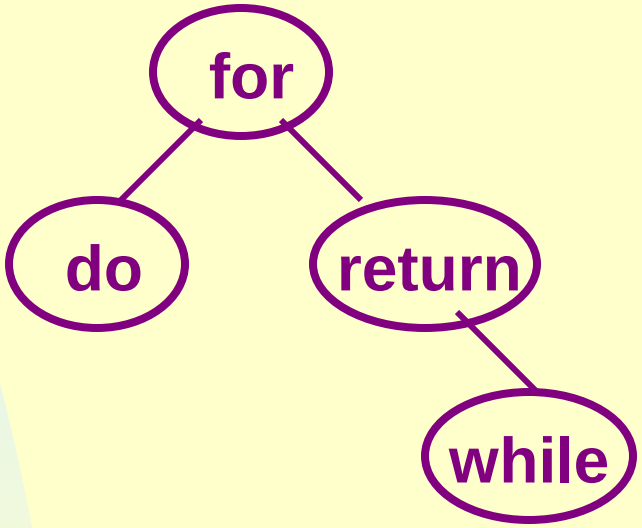
$r_{02} = 1$

…

until w_{04}, c_{04} and r_{04} are obtained. The next slide show the results of the computation.

j-i	i = 0	1	2	3	4
0	$w_{00}=2$ $c_{00}=0$ $r_{00}=0$	$w_{11}=3$ $c_{11}=0$ $r_{11}=0$	$w_{22}=1$ $c_{22}=0$ $r_{22}=0$	$w_{33}=1$ $c_{33}=0$ $r_{33}=0$	$w_{44}=1$ $c_{44}=0$ $r_{44}=0$
1	$w_{01}=8$ $c_{01}=8$ $r_{01}=1$	$w_{12}=7$ $c_{12}=7$ $r_{12}=2$	$w_{23}=3$ $c_{23}=3$ $r_{23}=3$	$w_{34}=3$ $c_{34}=3$ $r_{34}=4$	
2	$w_{02}=12$ $c_{02}=19$ $r_{02}=1$	$w_{13}=9$ $c_{13}=12$ $r_{13}=2$	$w_{24}=5$ $c_{24}=8$ $r_{24}=3$		
3	$w_{03}=14$ $c_{03}=25$ $r_{03}=2$	$w_{14}=11$ $c_{14}=19$ $r_{14}=2$			
4	$w_{04}=16$ $c_{04}=32$ $r_{04}=2$				

With the data in the table, it is possible to construct T_{04} as shown below:



Optimal b.s.t. for Example 10.2

In general, we need to compute c_{ij} for j-i=1,2,…,n.

And for each j-i=m,

there are n-m+1 c_{ij} 's (c_{0m},…, $c_{n-m,n}$) to compute.

The computation of each c_{ij} need to find the minimum from m quantities (10.4)---need O(m).

The total time for all c_{ij}'s with j-i=m is O((n-m)m).

The total time is

$$\sum_{1\le m\le n}((n-m)m)=O(n^3)$$

In fact, Knuth proved that the optimal l in (10.4) may be found by limiting the search to the range $r_{i,j-1} \leq l \leq r_{i+1,j}$. In this case, the computing time becomes $O(n^2)$. Algorithm obst uses this result.

Class BST of Chapter 5 is augmented to include obst as a private member function and to have the private data members r, c, and w.

Now the algorithm obst.

```
template <class KeyType>
BST<KeyType>::Obst(int *p,int *q,Element<KeyType>*a,int n)
//given a1< a2<… <an, and pj, 1≤j≤n, and qi, 0≤i≤n, compute cij
//for Tij for ai+1,…, aj. Also compute rij and wij.
{
   for (int i=0; i<n; i++) {
      w[i][i]=q[i]; r[i][i]=c[i][i]=0;     //initialize
      w[i][i+1]=q[i]+q[i+1]+p[i+1]; //optimal trees with one node
      r[i][i+1]=i+1;
      c[i][i+1]=w[i][i+1];
    }
   w[n][n]=q[n]; r[n][n]=c[n][n]=0;
```

```
    for (int m=2; m<=n; m++)  //find optimal b.s.t. with m nodes
        for (i=0; i<=n-m; i++)  {
            int j=i+m;
            w[i][j]=w[i][j-1]+p[j]+q[j];
            int k=KnuthMin(i,j);
            //KnuthMin return k in the range [r[i,j-1], r[i+1, j]]
            //minimizing c[i,k-1]+c[k,j]
            c[i][j]=w[i][j]+c[i][k-1]+c[k][j]; //Eq. (10.3)
            r[i][j]=k;
        }
}   // end of Obst
```

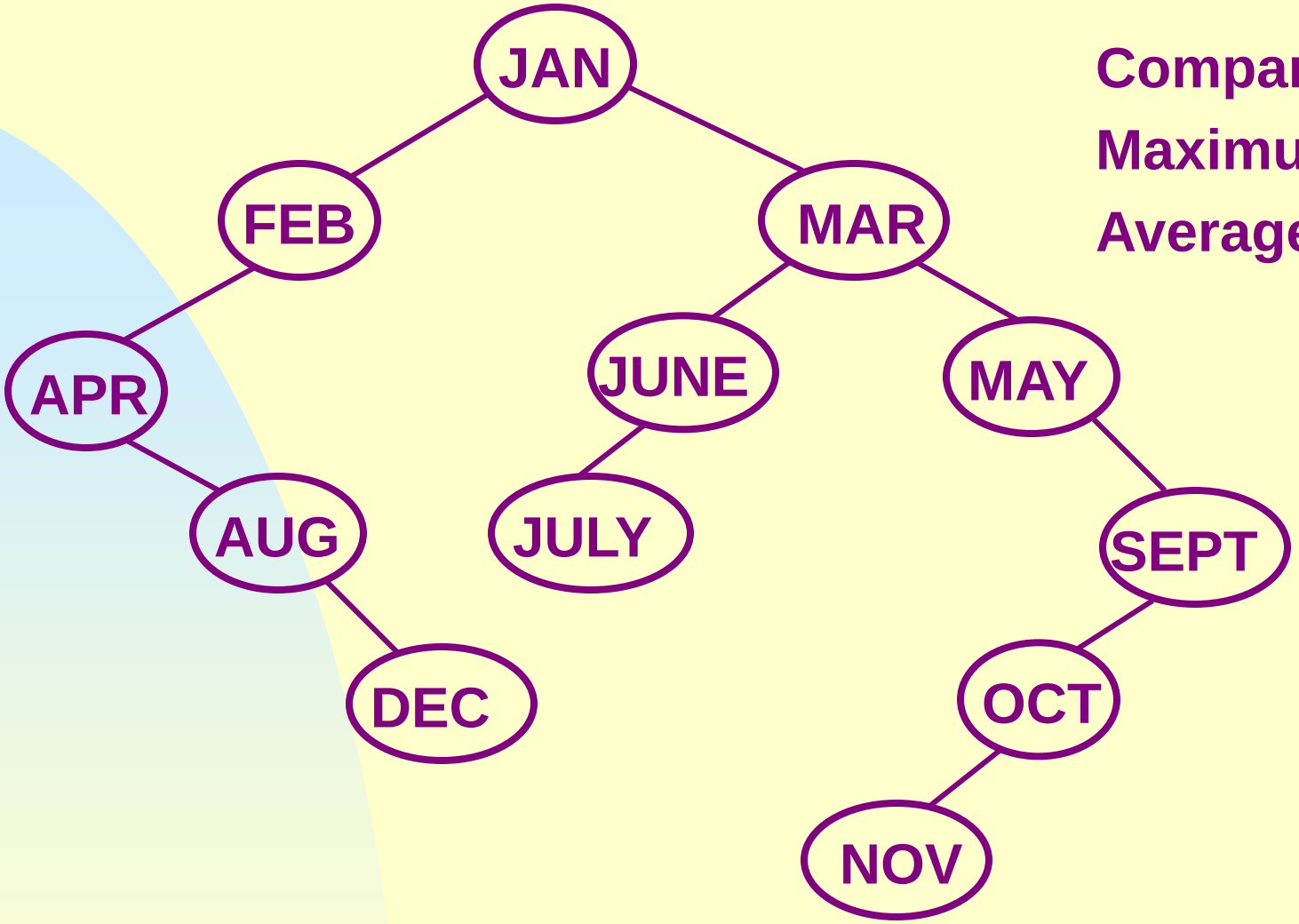
The actual T_{0n} may be constructed from r_{ij} in O(n).

Exercises: P562-3

10.2 AVL Trees

Dynamic collections of elements may also be maintained as binary search trees.

Fig. 10.8 shows the binary search tree obtained by entering the months JAN to DEC into an initially empty binary search tree by using function Insert (Program 5.21).



Comparisons:

Maximum: 6

Average: 3.5

Fig. 10.8

Fig. 10.9 shows the binary search tree obtained by entering JULY, FEB, MAY, AUG, DEC, MAR, OCT, APR, JAN, JUNE, SEPT, NOV into an initially empty binary search tree.

Comparisons:

Maximum: 4

Average: 3.1

Balanced

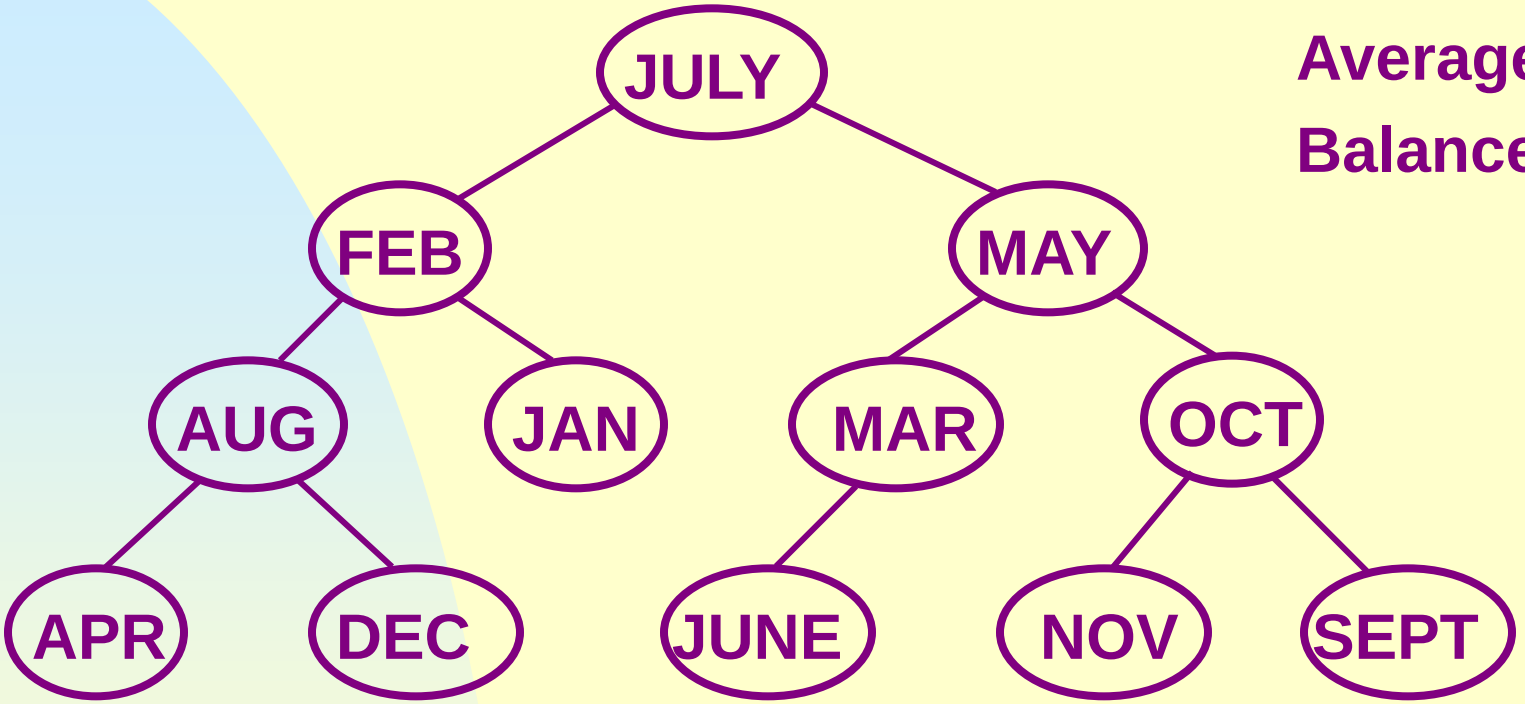
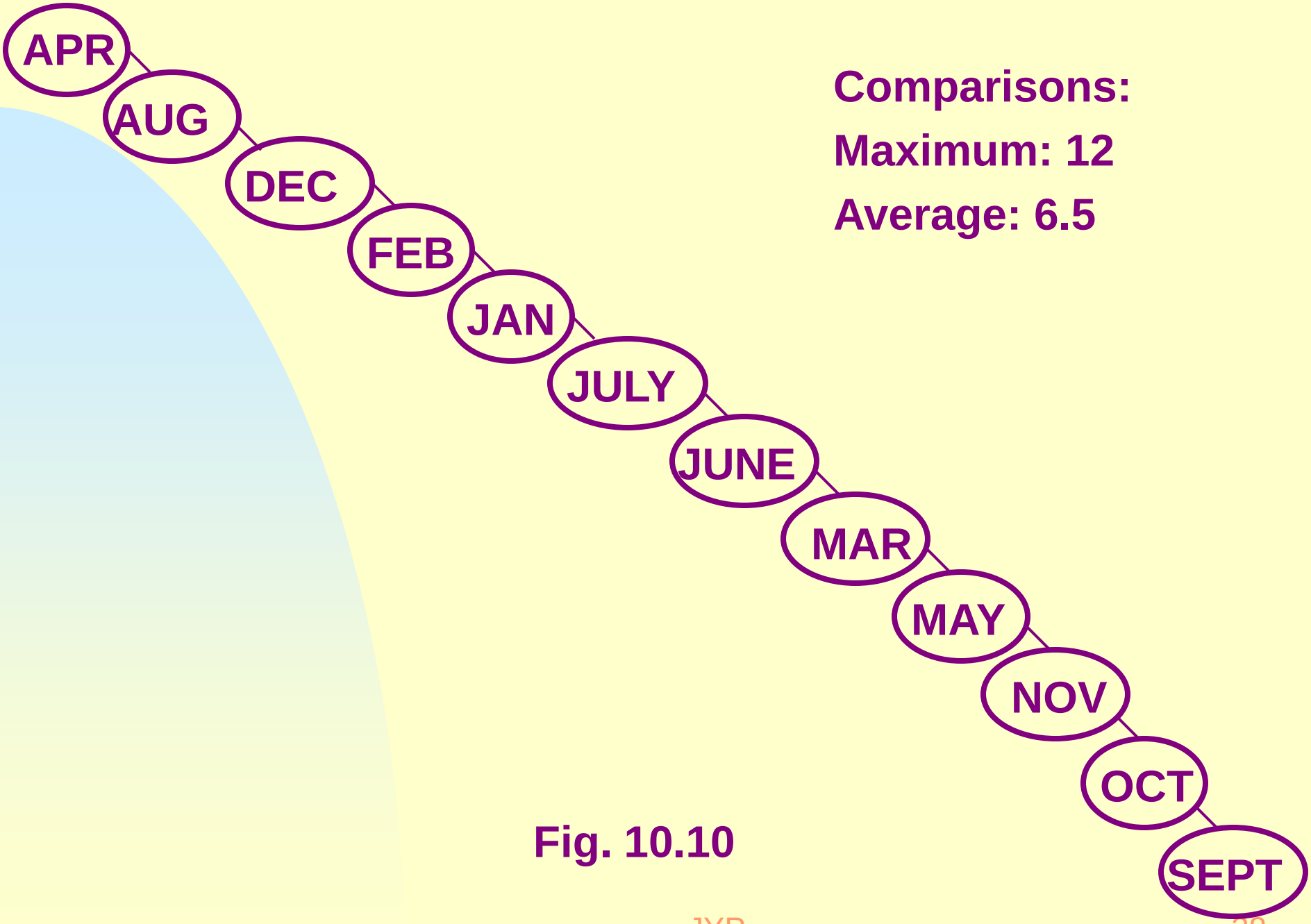


Fig. 10.9

Fig. 10.10 shows the binary search tree obtained by entering the months in lexicographic order into an initially empty binary search tree.



Comparisons:

Maximum: 12

Average: 6.5

Fig. 10.10

With equal search probabilities for keys, both the maximum and average search time will be minimized if the binary search tree is maintained as a complete binary tree at all times.

However, since the dynamic situation, it is difficult to achieve this without making the time required to insert a key very high.

It is, however, possible to keep the tree balanced to ensure both average and worst-case retrieval time of O(log n) for a tree with n nodes.

AVL tree (introduced by Adelson-Velskii and Landis) is a binary search tree that is balanced with respect to the heights of subtrees.

Definition: an empty tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees respectively, then T is height-balanced iff

(1) T_L and T_R are height-balanced and

(2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R respectively.

Fig. 10.8 is not height-balanced.

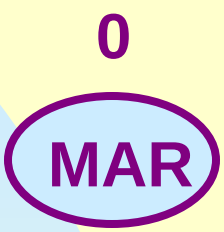
Fig. 10.9 is height-balanced.

Fig. 10.10 is not height-balanced.

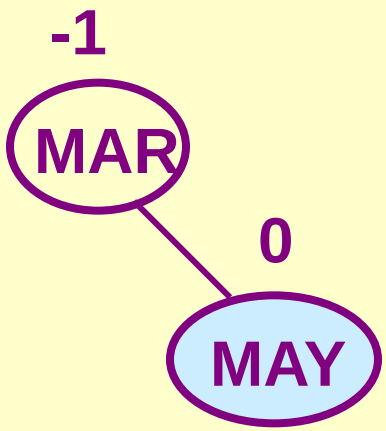
Definition: The balance factor, BF(T), of a node T in a binary tree is defined to be $h_L - h_R$. For any node T in an AVL tree BF(T)=-1, 0, or 1.

To illustrate the processes involved in maintaining an AVL tree, let's insert MAR, MAY, NOV, AUG, APR, JAN, DEC, JULY and FEB into an empty tree.

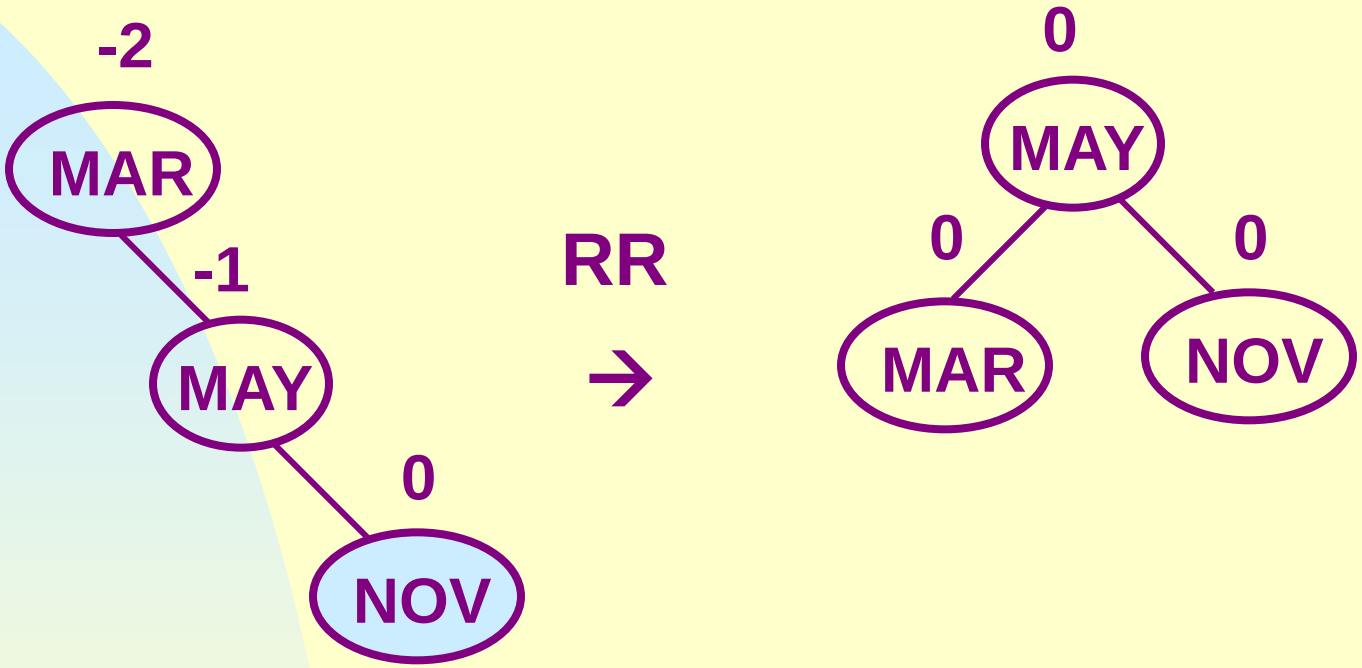
The following show the tree as it grows and restructuring involved in keeping it balanced. The numbers above each node represent the balance factor of that node.



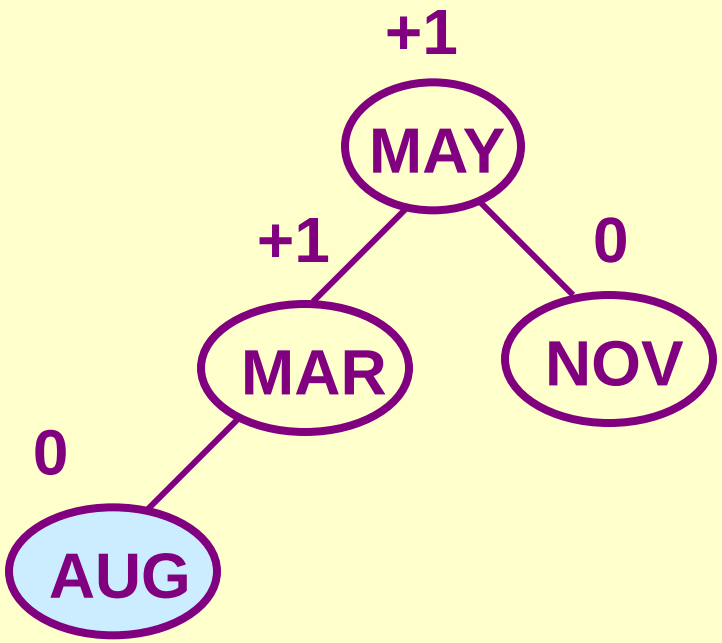
(a) Insert MAR



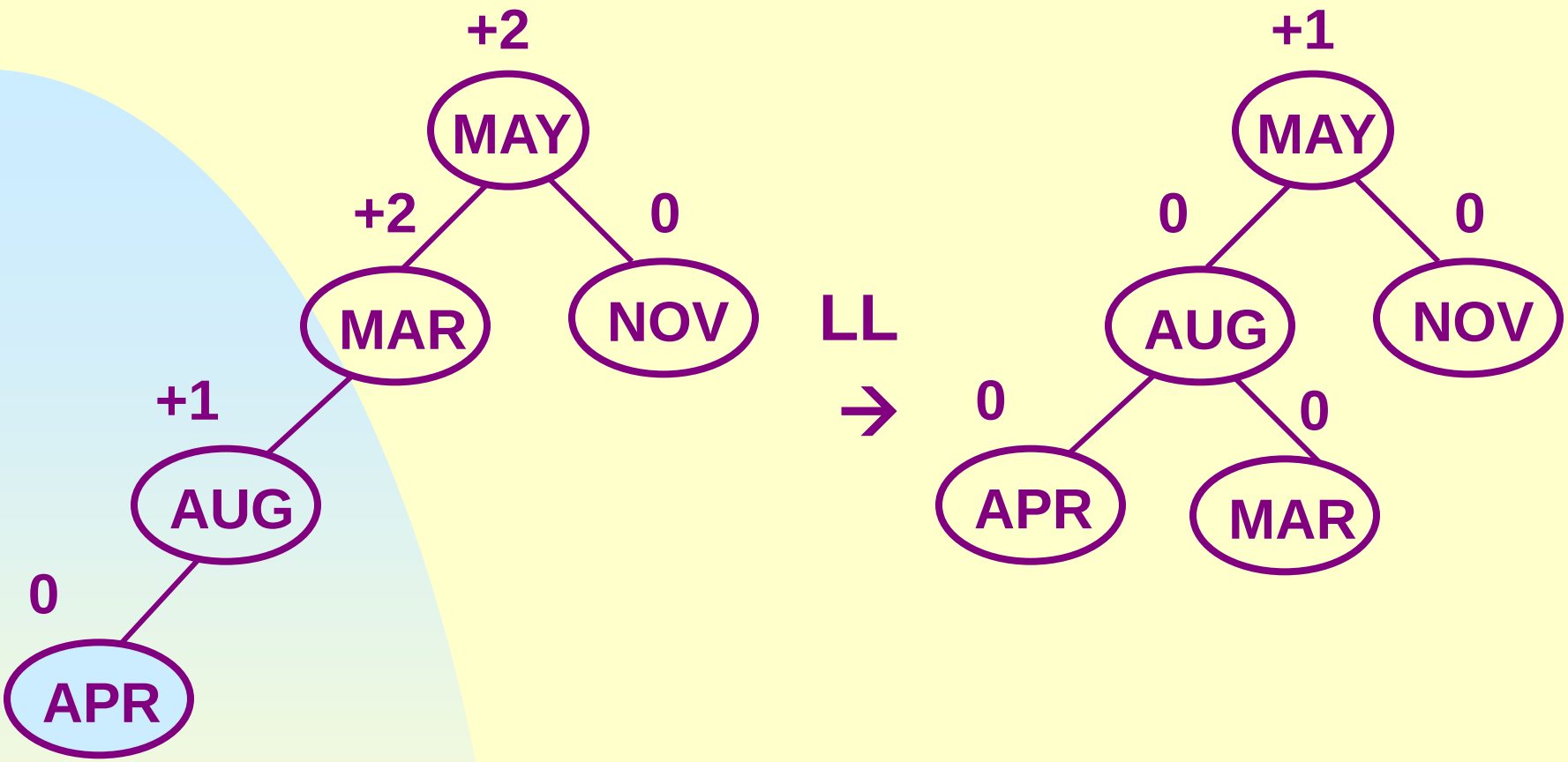
(b) Insert MAY



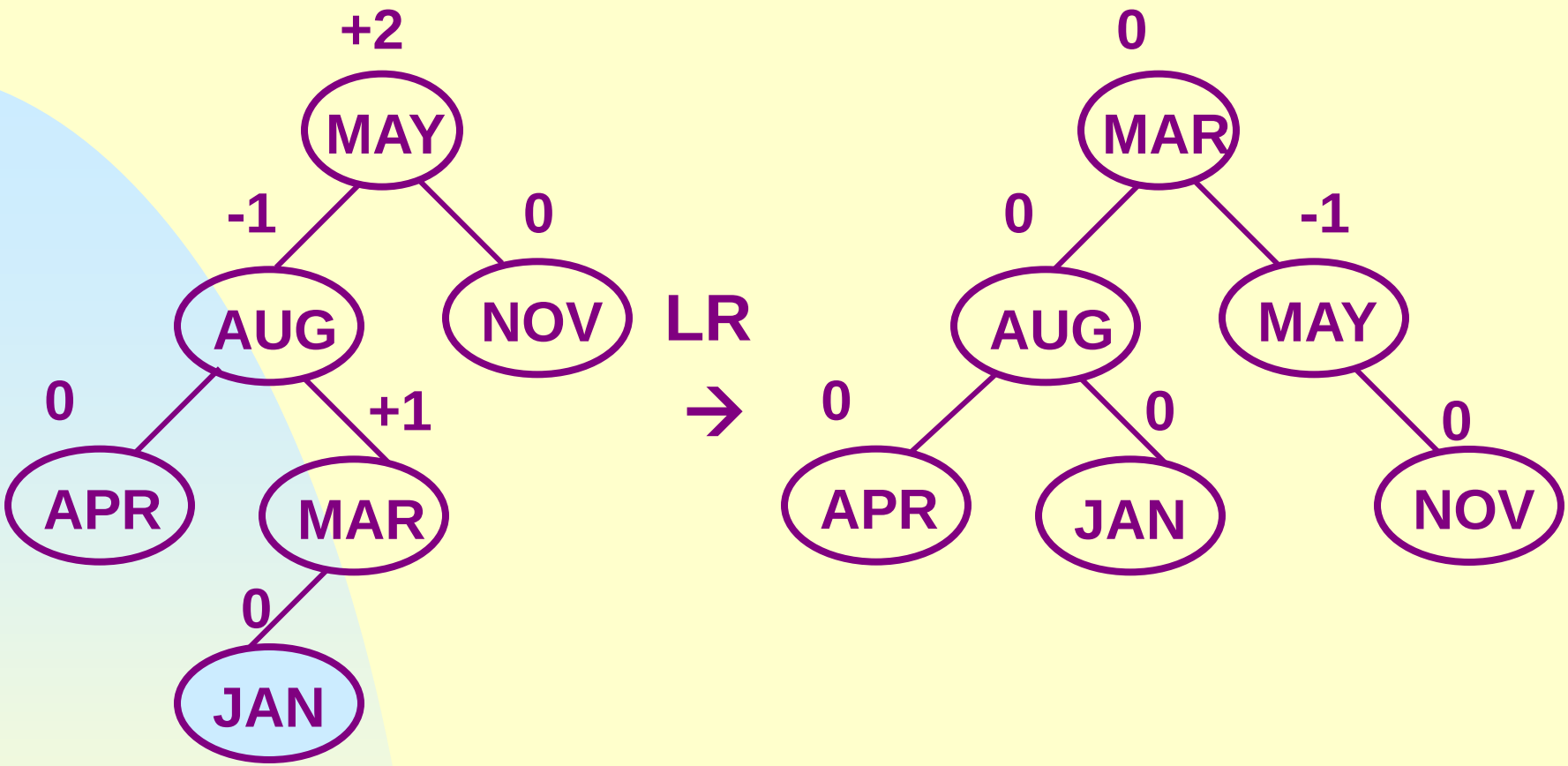
(c) Insert NOV



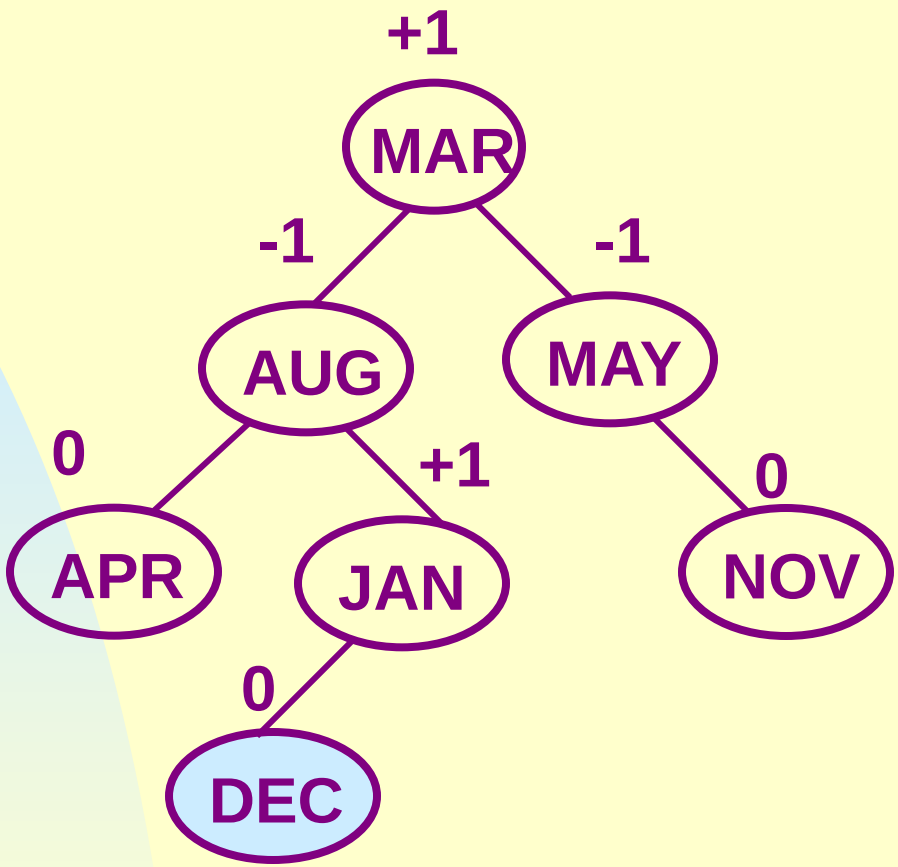
(d) Insert AUG



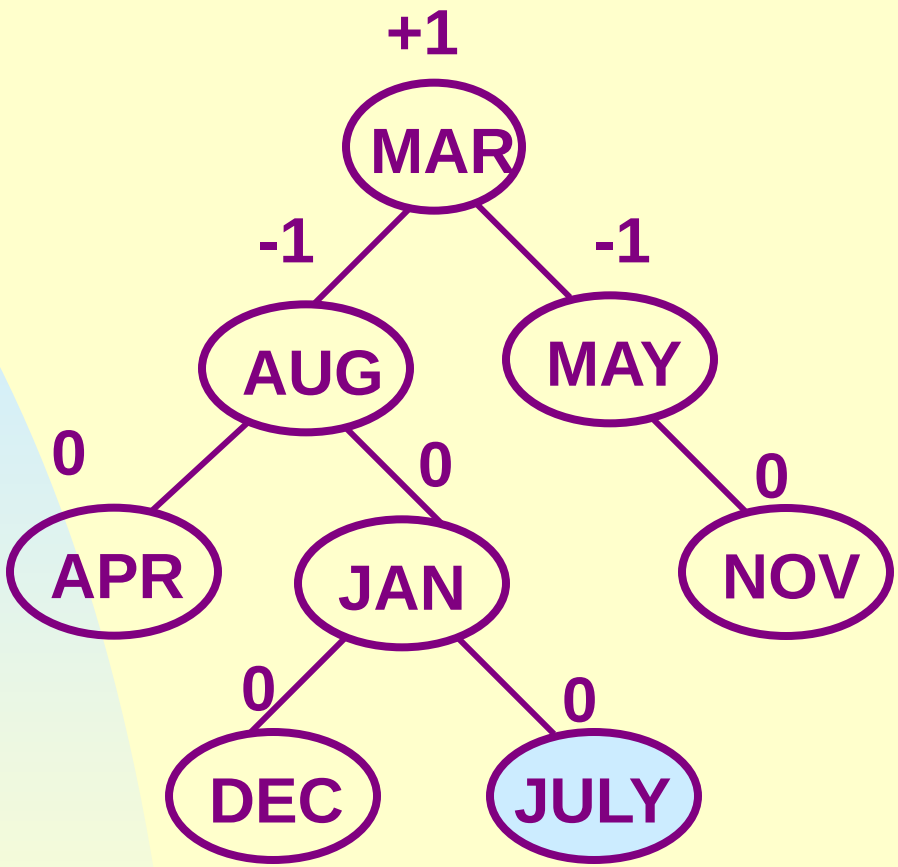
(e) Insert APR


+2
MAY
-1
0
AUG
NOV
0
+1
APR
MAR
0
JAN
LR
0
MAR
0
-1
AUG
MAY
0
0
0
APR
JAN
NOV

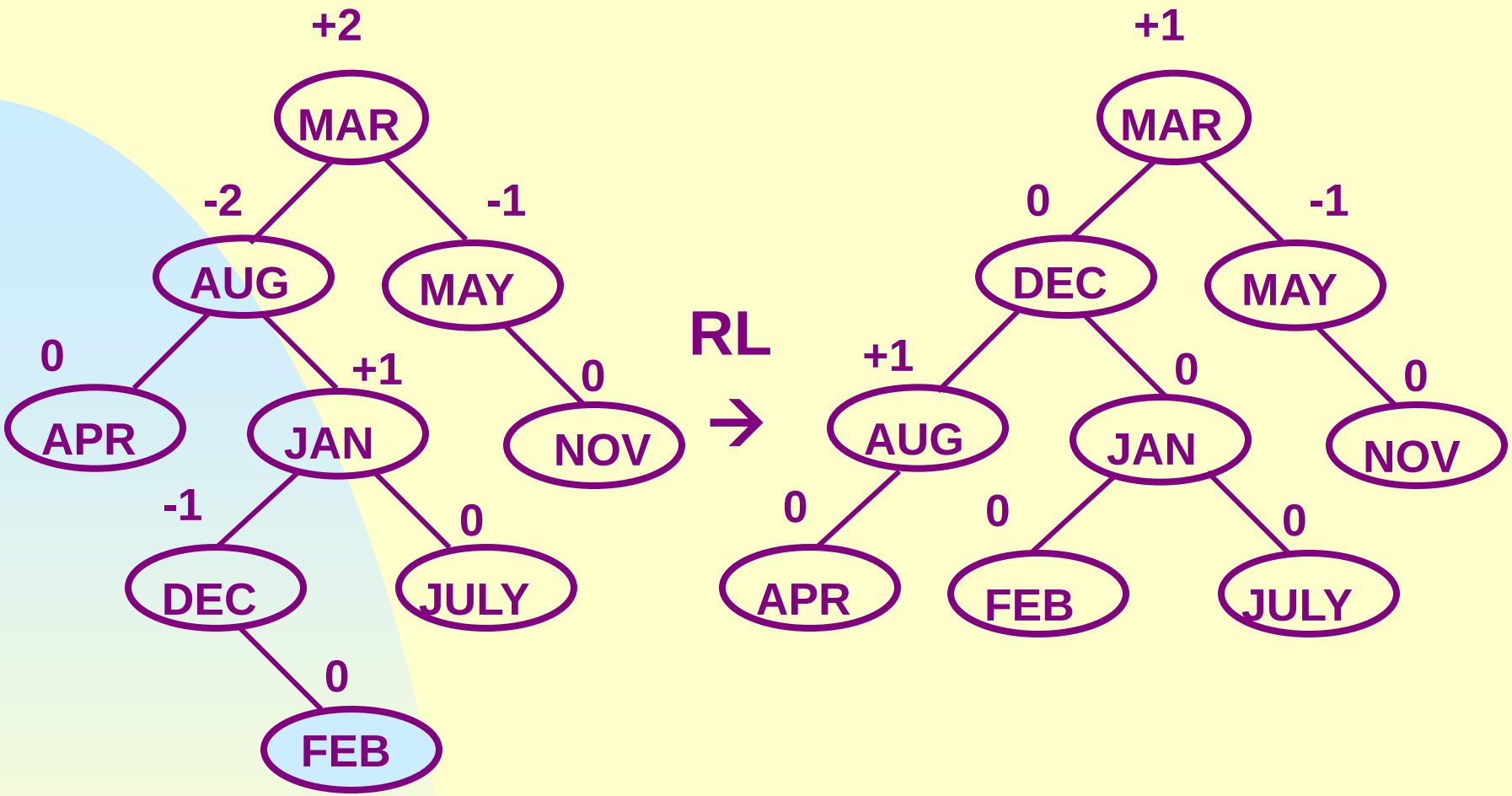
(f) Insert JAN



(g) Insert DEC



(h) Insert JULY


+2
MAR
-2
-1
AUG
MAY
0
+1
0
APR
JAN
NOV
-1
0
DEC
JULY
0
FEB
RL
+1
MAR
0
-1
DEC
MAY
+1
0
0
AUG
JAN
NOV
0
0
0
APR
FEB
JULY

(i) Insert FEB

In the proceeding example we saw that the addition of a node to a balanced binary search tree could unbalance it.

The rebalancing uses essentially 4 kinds of rotations:

LL, RR, LR, and RL.

LL and RR are symmetric, as are LR and RL.

The rotations are characterized by the nearest ancestor, A, whose BF becomes ± 2, of the inserted node Y.

LL: Y is inserted in the left subtree of the left subtree of A.

LR: Y is inserted in the right subtree of the left subtree of A.

RR: Y is inserted in the right subtree of the right subtree of A.

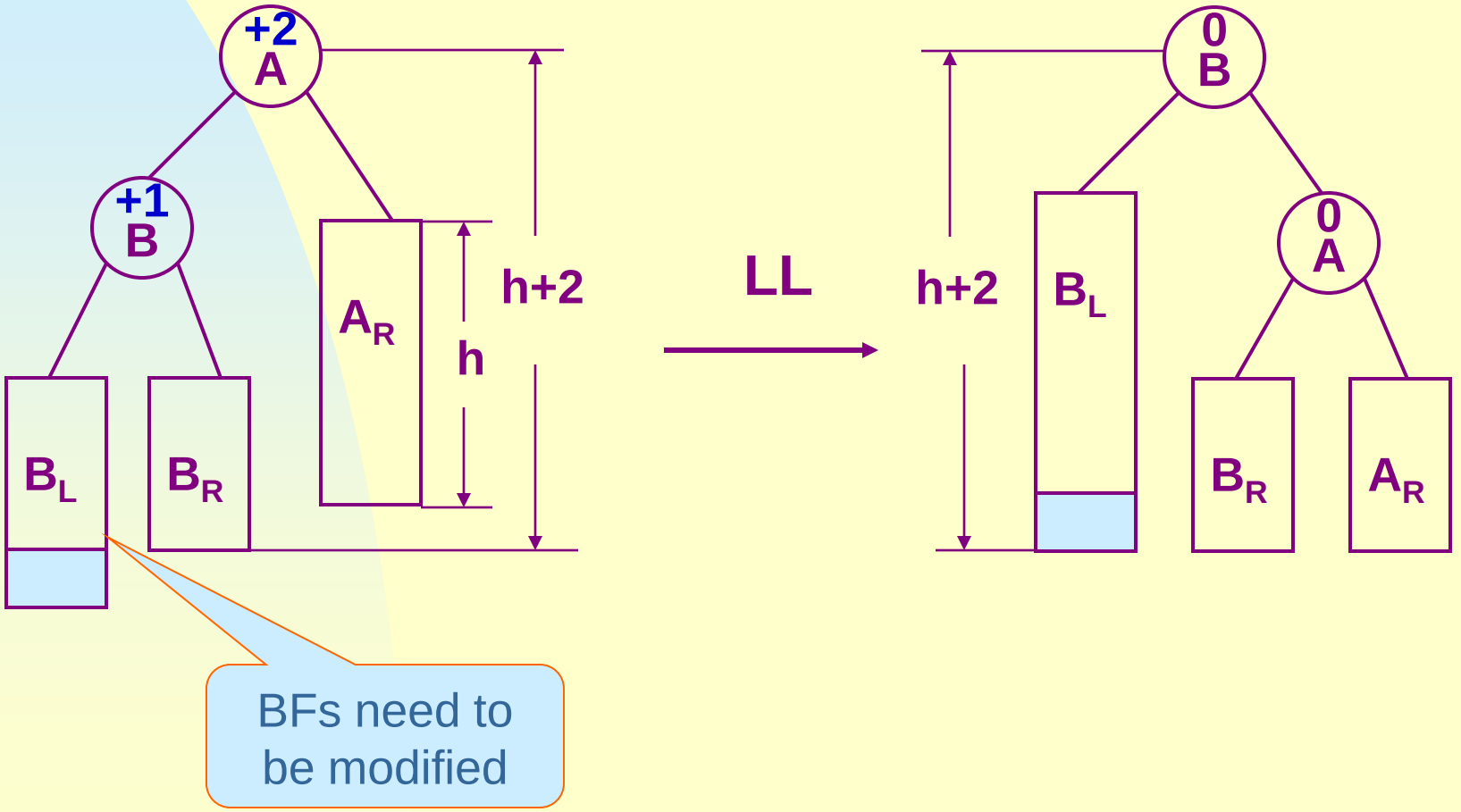
RL: Y is inserted in the left subtree of the right subtree of A.

As shown in the following:

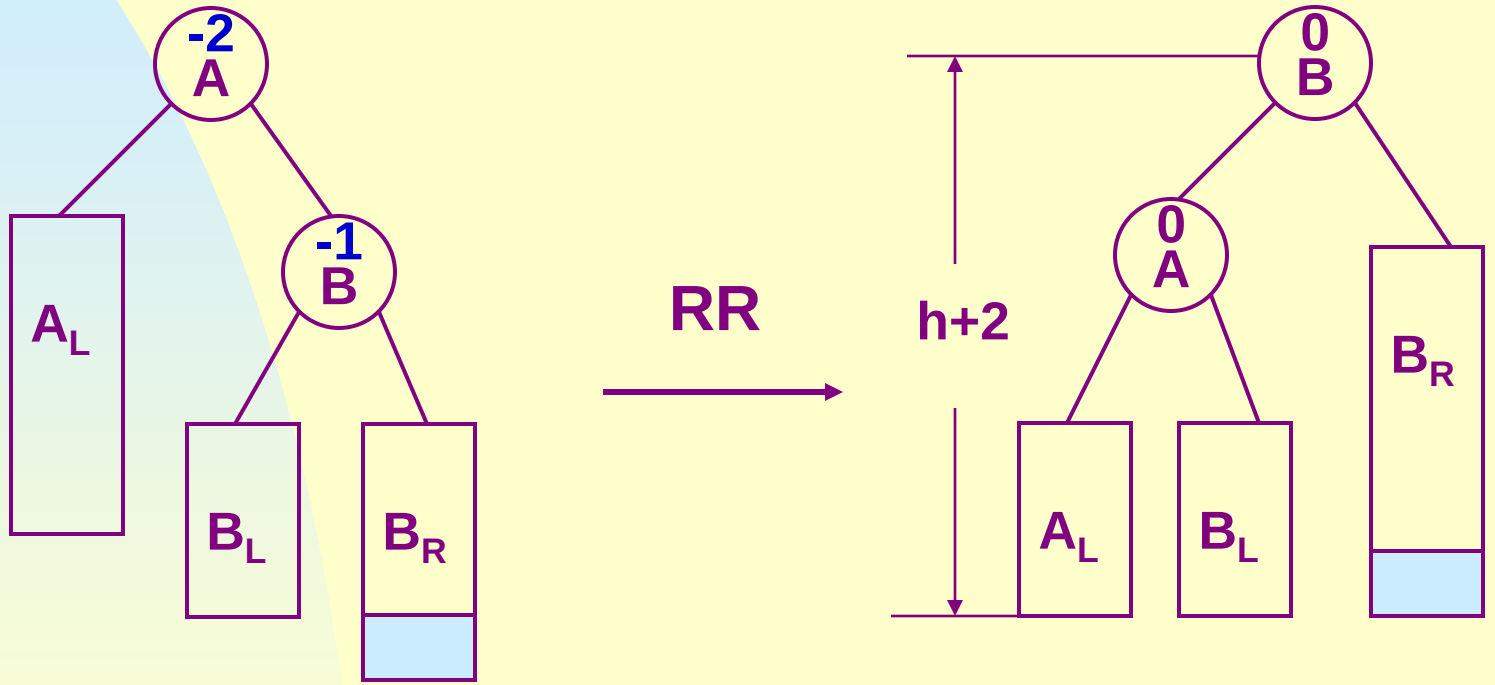
Unbalanced following insertion

rotation type

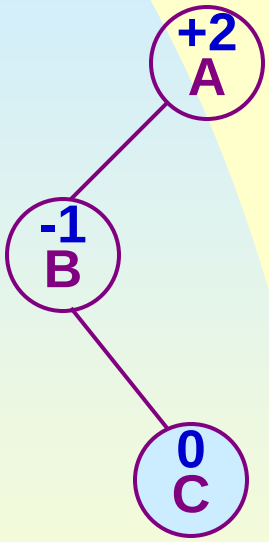
Rebalanced



Unbalanced following insertion | rotation type | Rebalanced



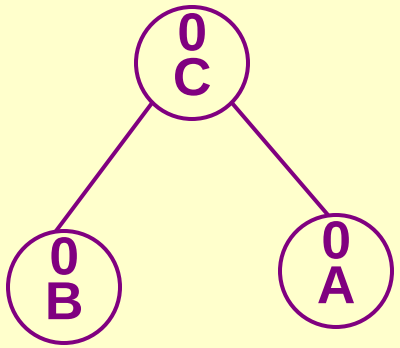
Unbalanced following insertion



rotation type

LR(a)

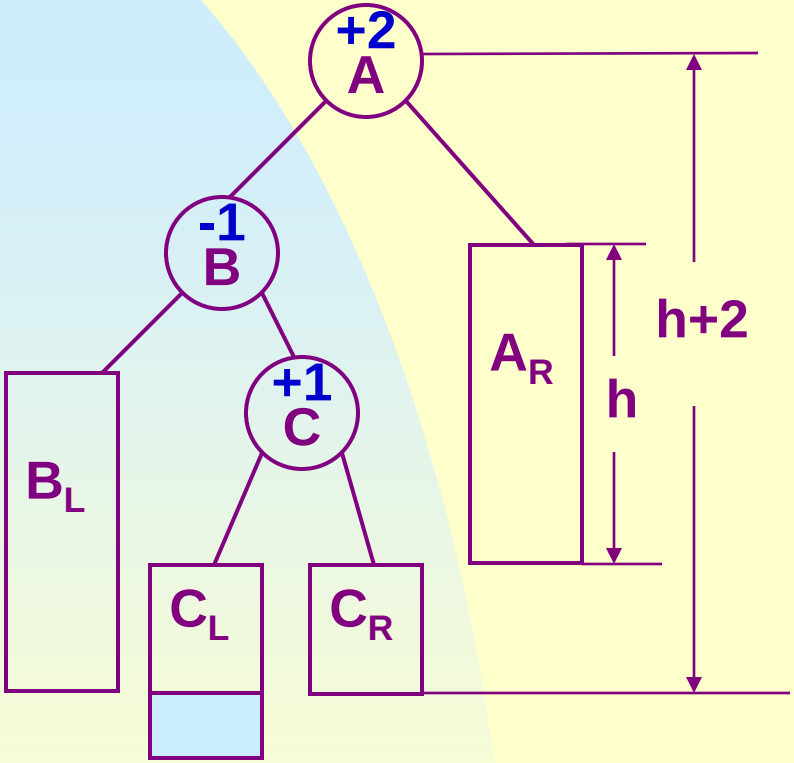
Rebalanced



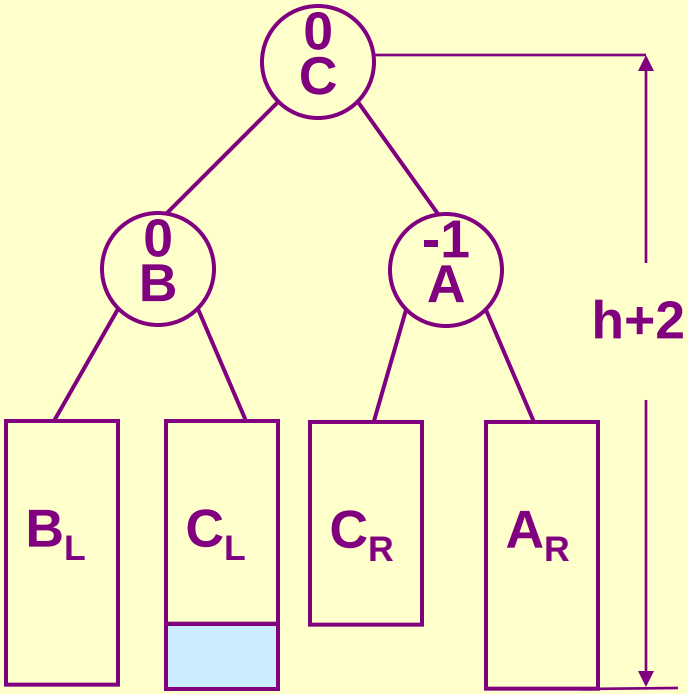
Unbalanced following insertion

rotation type

Rebalanced



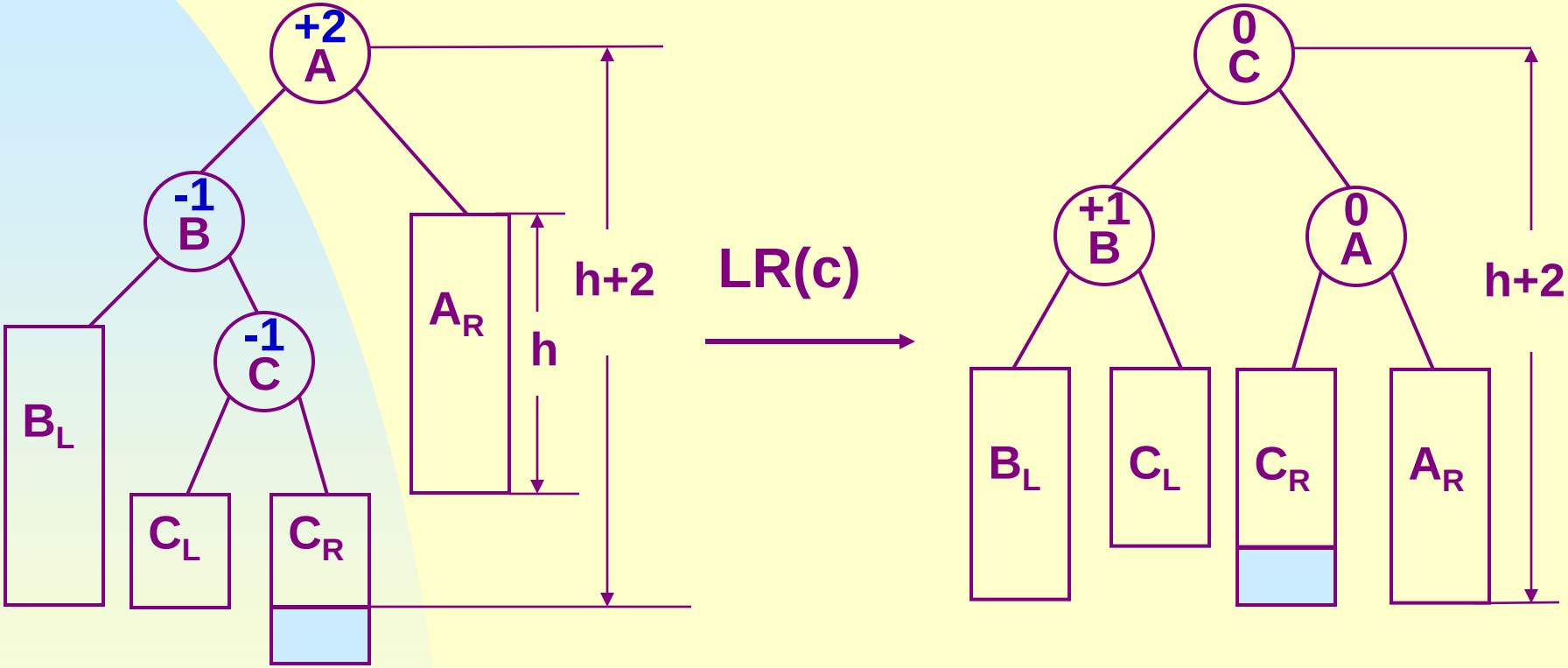
LR(b)



Unbalanced following insertion

rotation type

Rebalanced



LL, RR, LR, and RL are the only 4 cases possible for rebalancing.

LL and RR --- single rotations,

LR and RL --- double rotations.

For example, LR can be viewed as an RR followed by an LL rotation.

Note that the height of the subtree involved in the rotation is the same after rebalancing as it was before the insertion.

This means that once the rebalancing has been done on the subtree in question, examining the remaining tree is unnecessary.

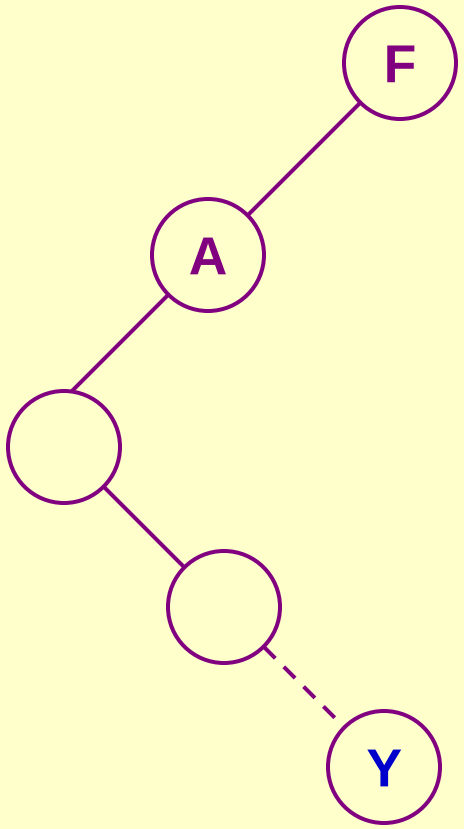
The only nodes whose BF can change are those in the subtree that is rotated.

To locate A, note it is the nearest ancestor of Y, whose BF becomes ±2, and it is also the nearest ancestor with BF= ±1 before the insertion.

Therefore, before the insertion, the BF's of all nodes on the path from A to the new insertion point must have been 0.

Thus, A is the nearest ancestor of the new node having a BF=±1 before insertion.

To complete the rotation, the parent of A, F is also needed.


F
A
Y

Whether or not the restructuring is needed, the BF's of several nodes change.

Let A be the nearest ancestor of the new node with BF=±1 before the insertion.

If no such an A, let A be the root.

The BF's of nodes from A to the parent of the new node will change to ±1.

AVL::Insert gives the details:

```
template <class K, class E> class AVL;

template <class K, class E>
class AvlNode {
friend class AVL<K, E>;
public:
    AvlNode(const K& k, const E& e)
    {key=k; element=e; bf=0; leftChild=rightChild=0;}
private:
    K key;
    E element
    int bf;
    AvlNode<K, E> *leftChild, *rightChild;
};
```

```
template <class K, class E>
class AVL {
public:
    AVL(): root(0) { };
    E& Search(const K&) const;
    void Insert(const  K&, const E&);
    void Delete(const  K&);
private:
    AvlNode<K, E> *root;
};
```

```
template <class K, class E>
void AVL<K, E>::Insert(const K& k, const E& e)
{
   if (!root) {   // empty tree
      root=new AvlNode<K, E>(k, e);
      return;
   }
  // phase 1: Locate insertion point for e.
  AvlNode<K, E> *a=root, // most recent node with BF±1
         *pa=0,       // parent of a
         *p=root, // p move through the tree
         *pp=0;   // parent of p
```

```
while (p)  {  // search for insertion point for x
   if (p→bf) {a=p; pa=pp;}
   if (k<p→key) {pp=p; p=p→leftChild;}
   else if (k>p→key) {pp=p; p=p→rightChild;}
          else {p→element=e; return;} // k already in the tree
 } // end of while

// phase 2: Insert and rebalance. k is not in the tree and
// may be inserted as the appropriate child of pp.
AvlNode<K, E> *y=new AvlNode<K, E>(k, e);
if (k<pp→key) pp→leftChild=y;      // as left child
else pp→rightChild=y;                  // as right child
```

```
// Adjust BF's of nodes on path from a to pp. d=+1 implies k
// is inserted in the left subtree of a and d=-1 in the right.
// The BF of a will be changed later.
int d;
AvlNode<K, E> *b,    // child of a
              *c;    // child of b
if (k>a→key) { b=p=a→rightChild; d=-1;}
else { b=p=a→leftChild; d=1;}
while (p!=y)
   if (k>p→key) { // height of right increases by 1
      p→bf= -1; p=p→rightChild;
   }
   else {  // height of left increases by 1
      p→bf= 1; p=p→leftChild;
   }
```

```
// Is tree unbalanced?
if (!(a→bf) || !(a→bf +d)) {     // tree still balanced
    a→bf +=d; return;
}
//tree unbalanced, determine rotation type
if (d==1)  {  // left imbalance
   if (b→bf==1)  {  // type LL
      a→leftChild=b→rightChild;
      b→rightChild=a; a→bf=0; b→bf=0;
   }
   else  {    // type LR
       c=b→rightChild;
       b→rightChild=c→leftChild;
       a→leftChild=c→rightChild;
       c→leftChild=b;
```

```
        c→rightChild=a;
        switch (c→bf) {
            case 1:  // LR(b)
                    a→bf=-1; b→bf=0;
                    break;
            case -1: // LR(c)
                    b→bf=1; a→bf=0;
                    break;
            case 0:  // LR(a)
                    b→bf=0; a→bf=0;
                    break;
        }
        c→bf=0; b=c; // b is the new root
    }  // end of LR
```

```
    } // end of left imbalance
    else {    // right imbalance: symmetric to left imbalance
    }
    // Subtree with root b has been rebalanced.
    if (!pa) root=b;  // A has no parent and a is the root
    else if (a==pa→leftChild) pa→leftChild=b;
            else pa→rightChild=b;
    return;
} // end of AVL::Insert
```

Computing time:

If h is the height of the tree before insertion, the time to insert a new key is O(h).

In case of AVL tree, h can be at most O(log n), so the insertion time is O(log n).

To prove h=O(log n), let N_h be the minimum number of nodes in an AVL tree of height h, the heights of its subtrees are h-1 and h-2, and both are AVL trees. Hence

$N_h = N_{h-1} + N_{h-2} + 1$ (the root) and

$N_0 = 0$, $N_1 = 1$, $N_2 = 2$.

By induction on h, we can show

$N_h = F_{h+2} - 1$ **for** $h \geq 0$.

As the Fibonacci number

$F_{h+2} \geq \phi^h$, $\quad \phi = (1+\sqrt{5})/2$

If n nodes in the tree, its height h is at most

$\log_\phi (n+1) = O(\log n)$

The exercises show that it is possible to find and delete a node with key k from an AVL tree in O(log n).

Exercises: P578-3, 5, 9

The next slide compares the worst-case times of certain operations on sorted sequential lists, sorted linked lists, and AVL trees.

Operation	Sequential list	Linked list	AVL tree
Search for k	O(log n)	O(n)	O(log n)
Search for jth item	O(1)	O(j)	O(log n)
Delete k	O(n)	O(1)[1]	O(log n)
Delete jth item	O(n-j)	O(j)	O(log n)
Insert	O(n)	O(1)[2]	O(log n)
Output in order	O(n)	O(n)	O(n)

1. Doubly linked list and position of k known
2. Position for insertion known

Chapter 11
Multiway Search Trees

11.1 m-Way Search trees

11.1.1 Definition of m-Way Search Trees

If AVL tree were used to represent a very large collection of elements, it would be on disk. To search among n keys requires $h=1.44 \log_2 (n+1)$ disk accesses in the worst case.

If $n=10^6$, $1.44\log_2 (n+1)\approx 28$ --- too bad!

Recall that the block of a disk access (I/O) is much larger than the node of a binary tree. If we use AVL tree for index, accessing a node is actually accessing a block of which most part are useless, as shown below:

an AVL node in a disk block

To break the $\log_2(n+1)$ barrier on tree height resulting from the use of binary search trees, we must use search trees whose degree is more than 2.

In practice, we use the **largest degree** for which the tree node fits into a **block**.

Definition: An m-way search tree, either is empty or satisfies the following properties:

(1) The root has at most m subtrees and has the following structure:

$$n, A_0, (E_1, A_1), (E_2, A_2), \ldots, (E_n, A_n)$$

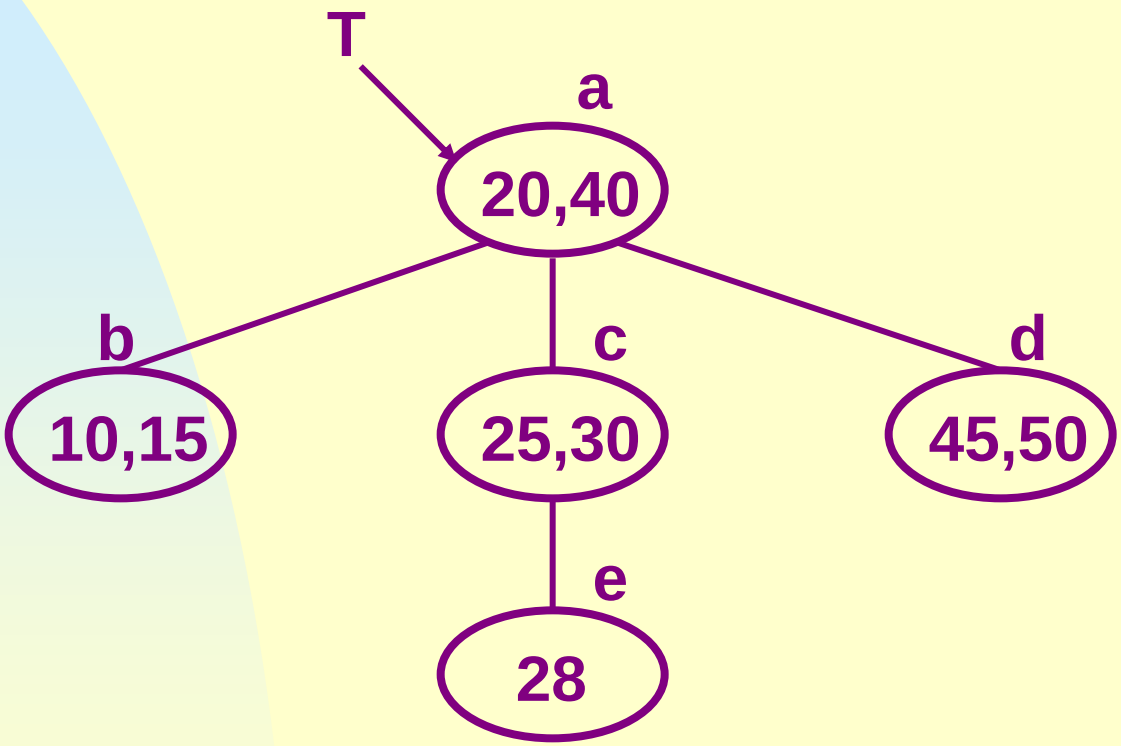
where the A_i , $0 \le i \le n < m$, are pointers to subtrees, and E_i, $1 \le i \le n < m$, are elements. Each E_i has a key $E_i.K$.

(2) $E_i.K < E_{i+1}.K$, $1 \le i < n$.

(3) Let $E_0.K = -\infty$ and $E_{n+1}.K = \infty$. All key values in subtree A_i are less than $E_{i+1}.K$ and greater than $E_i.K$, $0 \le i \le n$.

(4) The subtrees A_i , $0 \le i \le n$, are also m-way search trees.

The following is a 3-way search tree:



In a tree of degree m and height h, the maximum number of nodes is

$$\sum_{0 \le i \le h-1} m^i = (m^h - 1)/(m-1)$$

Each node has at most m-1 keys, the maximum number of keys is m^h-1.

- **for a binary tree with h=3, it is 7.**
- **for a 200-way tree with h=3, it is $8*10^6$-1.**

To achieve a performance close to that of the best m-way search tree for a given number of elements n, the search tree must be balanced.

11.1.2 Searching an m-way Search Tree

The searching is easy, and the next slide gives a high level description of the algorithm to search an m-way search tree.

// Search an m-way search tree for an element with key x.
// Return the element if found, else return **NULL**.
E_0.K = -MAXKEY**;**
for (*p=root**;** p**;** p= A_i)
{
    Let node p have the format n, A_0, (E_1, A_1), …, (E_n, A_n)**;**
    E_{n+1}.K = MAXKEY**;**
    Determine i such that E_i.K <= x < E_{i+1}.K**;**
    if (x==E_i.K) **return** E_i**;**
}
// x is not in the tree
return NULL;

Exercises: P609-3

11.2 B-Trees

11.2.1 Definition and Properties

A particular balanced m-way search tree is B-tree.

To define it, an external (or failure) node is added wherever we otherwise have a NULL pointer.

An external node represents a node that can be reached during a search only if the element being searched for is not in the tree.

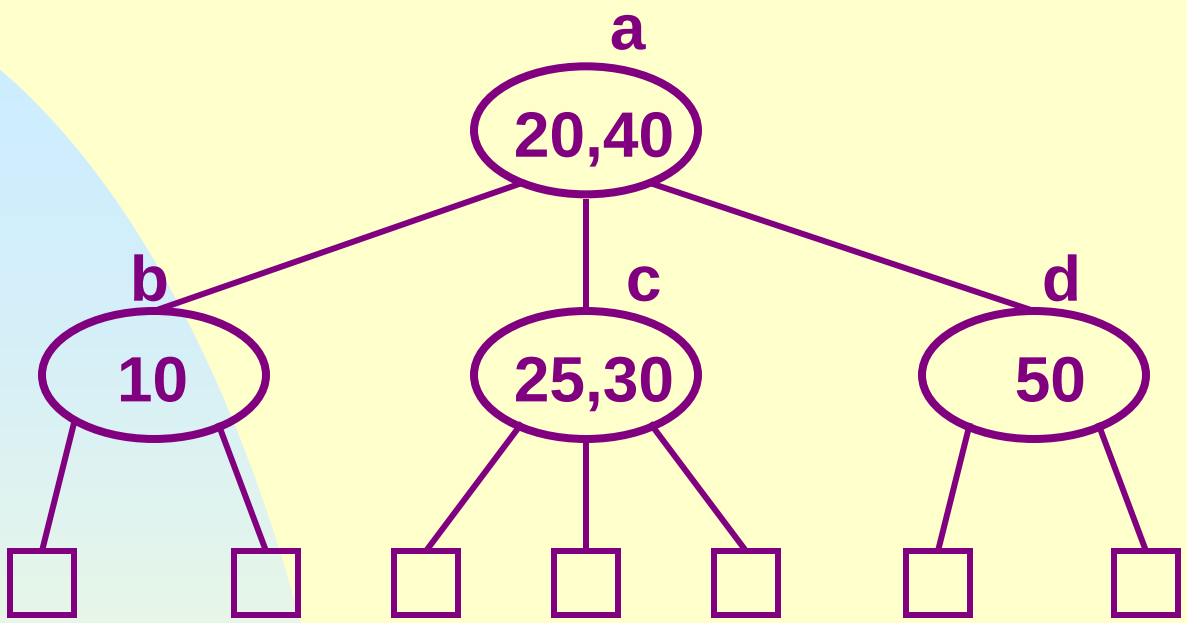
Definition: A B-tree of order m is an m-way search tree that is empty or satisfies the following properties:

(1) The root has at least two children.

(2) All nodes other than the root node and external nodes have at least ⌈m/2⌉ children.

(3) All external nodes are at the same level.

When m=3, all internal nodes have a degree of either 2 or 3, and a B-tree of order 3 is known as an 2-3 tree.



A B-tree of order 3

B-trees of order 2 are full binary trees.

For any $n \geq 0$ and $m > 2$, there is a B-tree of order m that contains n elements.

11.2.2 Number of Elements in a B-tree

Let t be a B-tree of order m in which all external nodes are at level l+1, and let N the number of keys in t. Then

(1) $N \leq m^l - 1$ (upper bound).

(2) $N \geq 2\lceil m/2 \rceil^{l-1} - 1$ (lower bound).

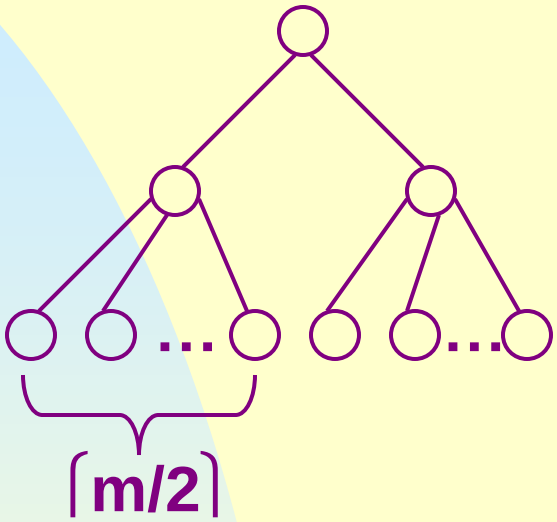
To show this, observe:

If l>1, there are at least 2 nodes at level 2,

there are at least $2\lceil m/2 \rceil$ nodes at level 3,

…

there are at least $2\lceil m/2 \rceil^{l-2}$ nodes at level l.



Level 1

2

3

If the key values in the tree are $K_1, K_2, \ldots, K_N$, $K_i<K_{i+1}$, $1 \le i < N$, then the number of external nodes is N+1 because failures occur for $K_i<x<K_{i+1}$, $0 \le i \le N$,

where $K_0 = -\infty$ and $K_{N+1} = +\infty$.

Therefore,

N+1= number of failure nodes in t

= number of nodes at level l+1

$\geq 2^{\lceil m/2 \rceil^{l-1}}$

$N \geq 2^{\lceil m/2 \rceil^{l-1}} - 1$.

$l \leq \log_{\lceil m/2 \rceil}((N+1)/2)+1$.

In worst case to search the B-tree need l accesses.

Assume m=200, and note l is integer:

(1) For $N \leq 2*10^6-2$,

$l \leq \lfloor \log_{100}(10^6-1/2) \rfloor +1=3$.

(2) For $N \leq 2*10^8-2$,

$l \leq \lfloor \log_{100}(10^8-1/2) \rfloor +1=4$.

11.2.3 Insertion into a B-Tree

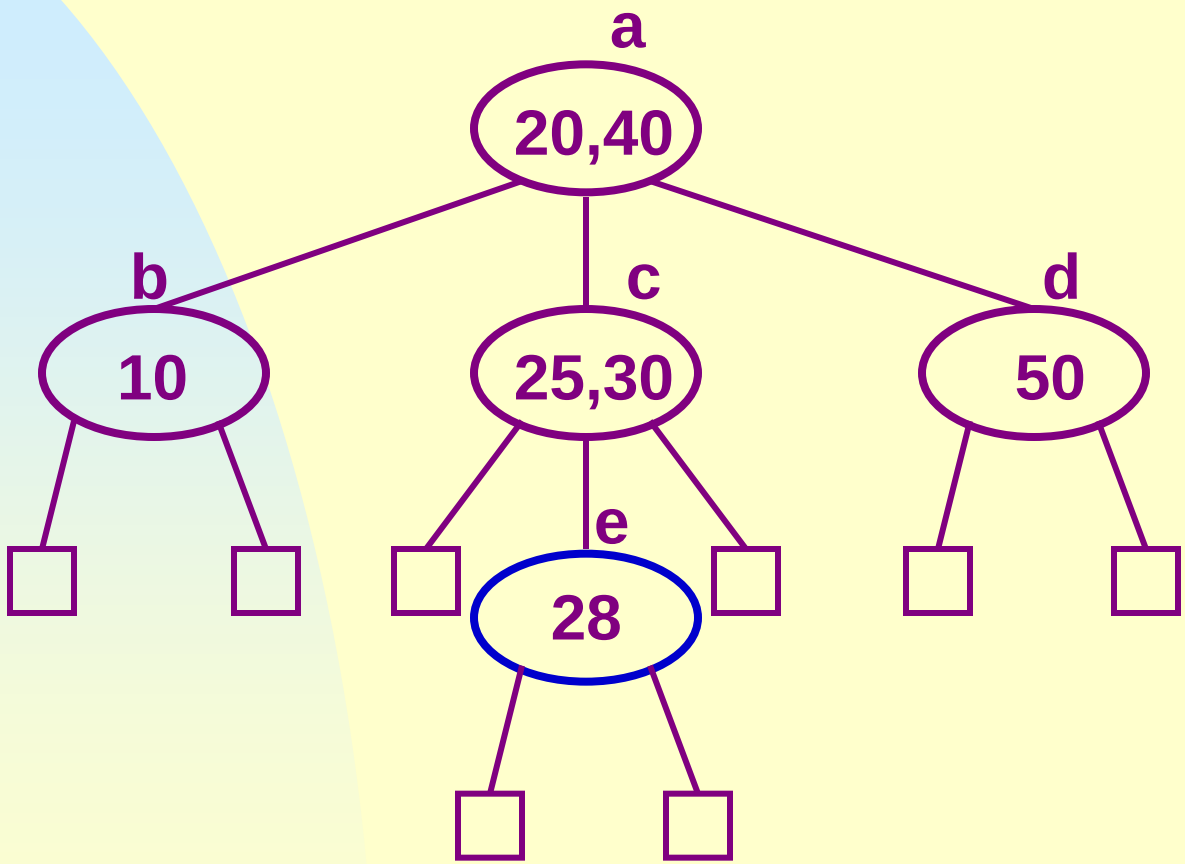
Note that we can't insert a new node in the position of an external node because:

1 One key might be too few;

2 External nodes must be at the same level.

As shown in the next slide.

We can’t insert 28 into a B-Tree like this:



Basic ideas of the insertion algorithm for B-tree of order m:

(1) Perform a search to determine the leaf node, p, into which the new key is to be inserted.

(2) If the insertion results in p having m keys, the node p is split. Otherwise, the new p is written to the disk and the insertion is complete.

To split the node, assume that following the insertion, p has the format

$m, A_0, (E_1, A_1), \ldots, (E_m, A_m)$ and $E_i.K<E_{i+1}.K$, $1 \le i < m$.

The node is split into 2 nodes, p and q, with the following formats:

node p: $\lceil m/2 \rceil - 1, A_0, (E_1, A_1), \ldots, (E_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$ (11.5)

node q: $m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, (E_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}), \ldots, (E_m, A_m)$

And the tuple $(E_{\lceil m/2 \rceil}, q)$ is to be inserted into the parent of p.

Inserting into the parent may require us to split the parent, and the splitting process can propagate all the way up to the root.

When the root split, a new root with a single key is created, and the height of the B-tree increases by one.

Note that when m=2, $\lceil m/2 \rceil - 1 = 0$, this means the above method does not work for m=2.

The next slide gives a high-level description of the insertion algorithm for a disk resident B-tree.

// Insert element x into a disk resident B-tree.

Search the B-tree for an element E with key x.K**;**

if such an E is found, replace E with x and **return;**

else let p be the leaf into which x is to be inserted**;**

q = **NULL;**

for (e=x**;** p**;** p=p→parent()) // the parent of root is **NULL**

{ // (e, q) is to be inserted into p

  Insert (e, q) into appropriate position in node p**;**

  Let the resulting node have the form:

  $n, A_0, (E_1, A_1), \ldots, (E_n, A_n)$**;**

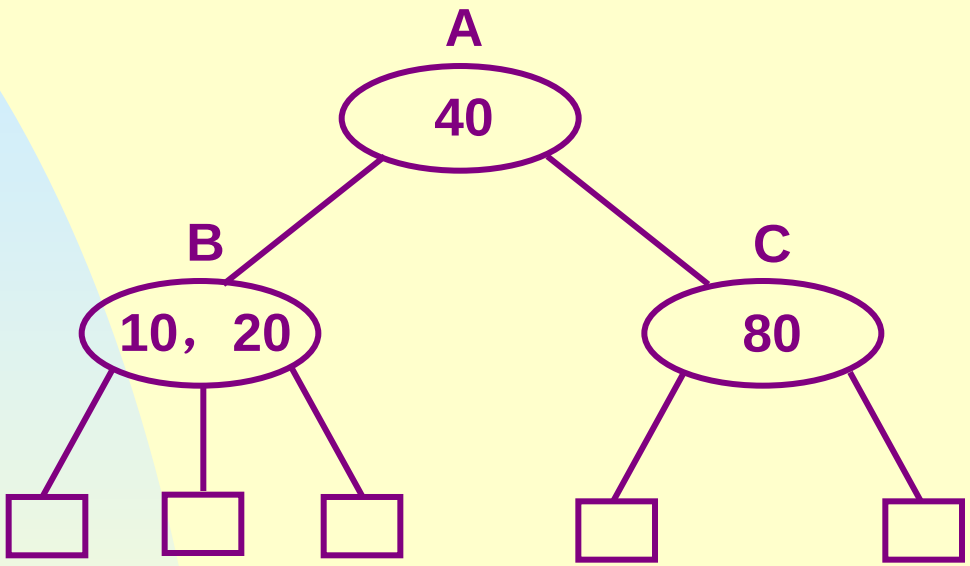
  if (n<=m-1) **{** // the resulting node is not too big

    write node p to disk**; return;**

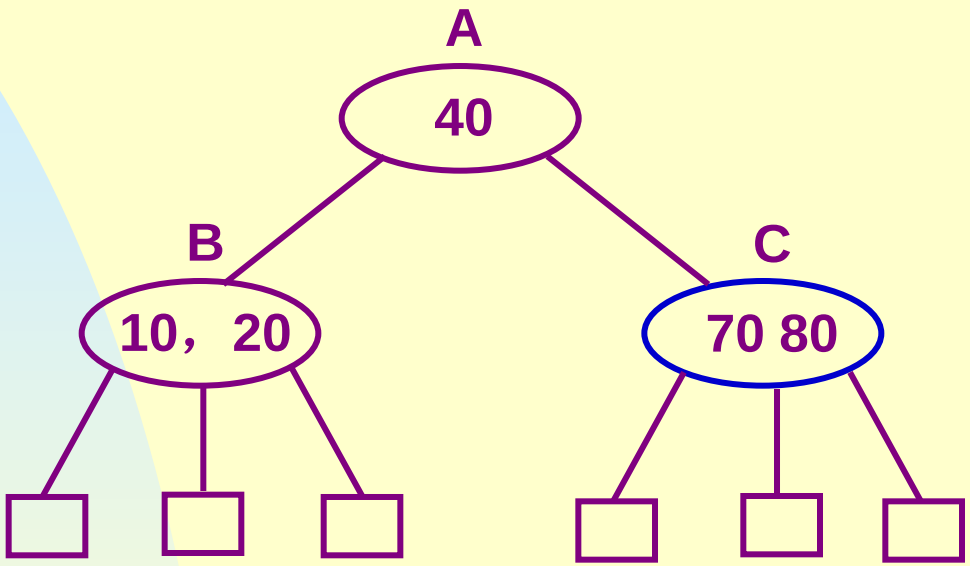
  }

```
    //node p has to be split
    Let node p and q be defined as in (11.5);
    e=E⌈m/2⌉;
    write nodes p and q to the disk;
}
// a new root is to be created
Create a new node r with format:  1, root, (e, q);
// when the tree is empty, root=p=0
root=r;
write  root to disk;
```

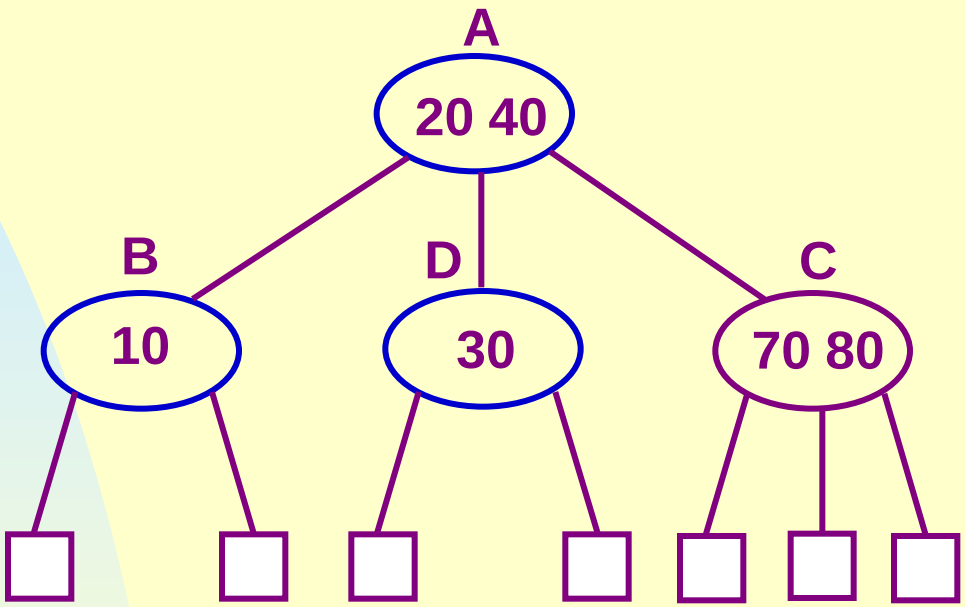
Example 11.1:



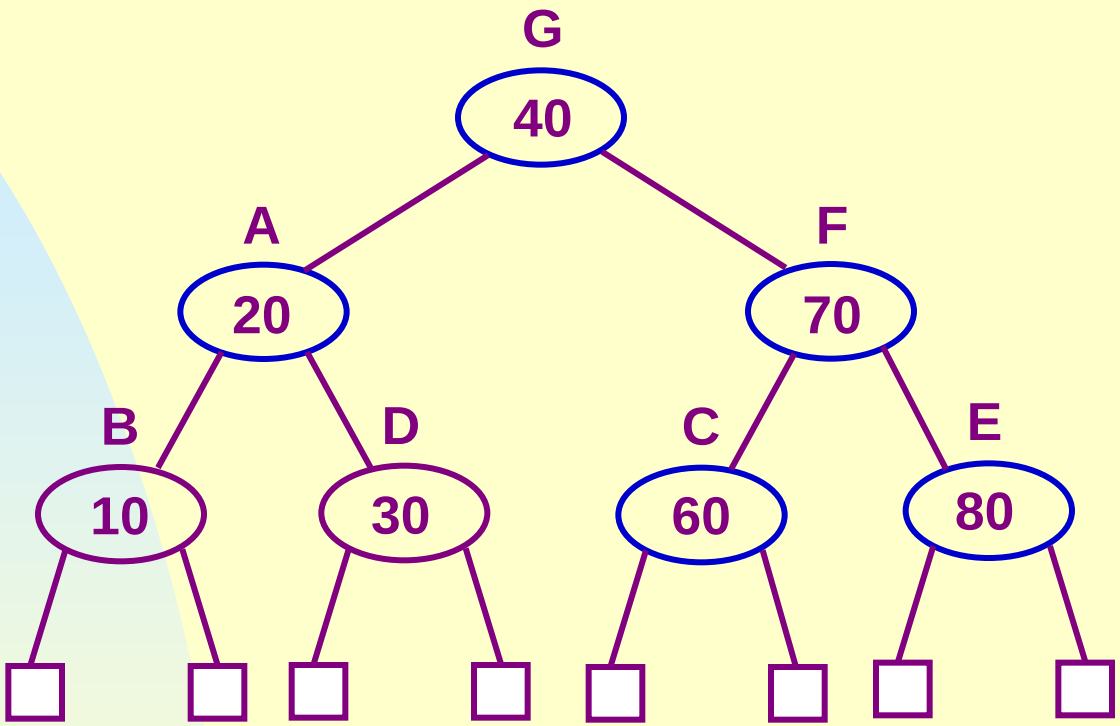
An initial 2-3 tree



70 inserted



30 inserted



60 inserted

Analysis of B-tree Insertion:

Let h be the height of the B-tree, then h disk accesses for the top-down search.

In the worst, all h of the accessed nodes may split during the bottom-up splitting pass. When a non-root node split, 2 nodes are written out. When the root split, 3 nodes are written out.

Assume that the h nodes read in during the top-down pass can be saved in memory so that they are not to be retrieved from disk during the bottom-up pass.

The total disk accesses is at most

$h+2(h-1)+3=3h+1$

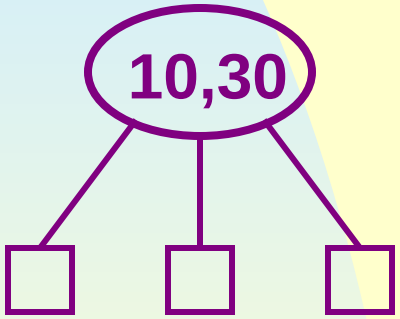
The average number of disk accesses is, however, approximately $h+1$ for lager m.

To show this, suppose we start with an empty B-tree and insert N elements into it.

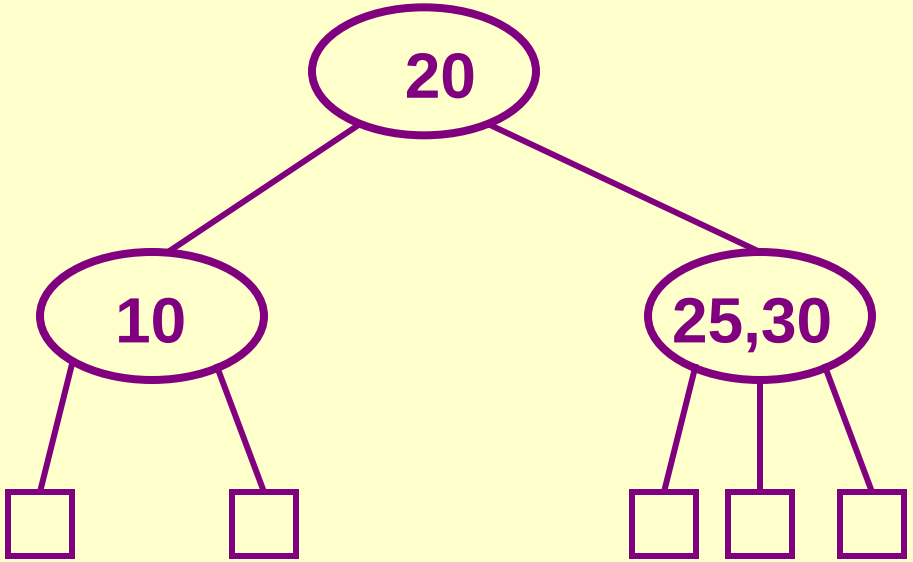
Let p be the number of internal nodes in the final B-tree with N elements, then the total number of nodes split is at most p-2. Because

- Each time a node splits, at least one additional node is created.
- The splitting of the root node create two additional nodes and the root is the first to split.
- The first node created from no splitting.

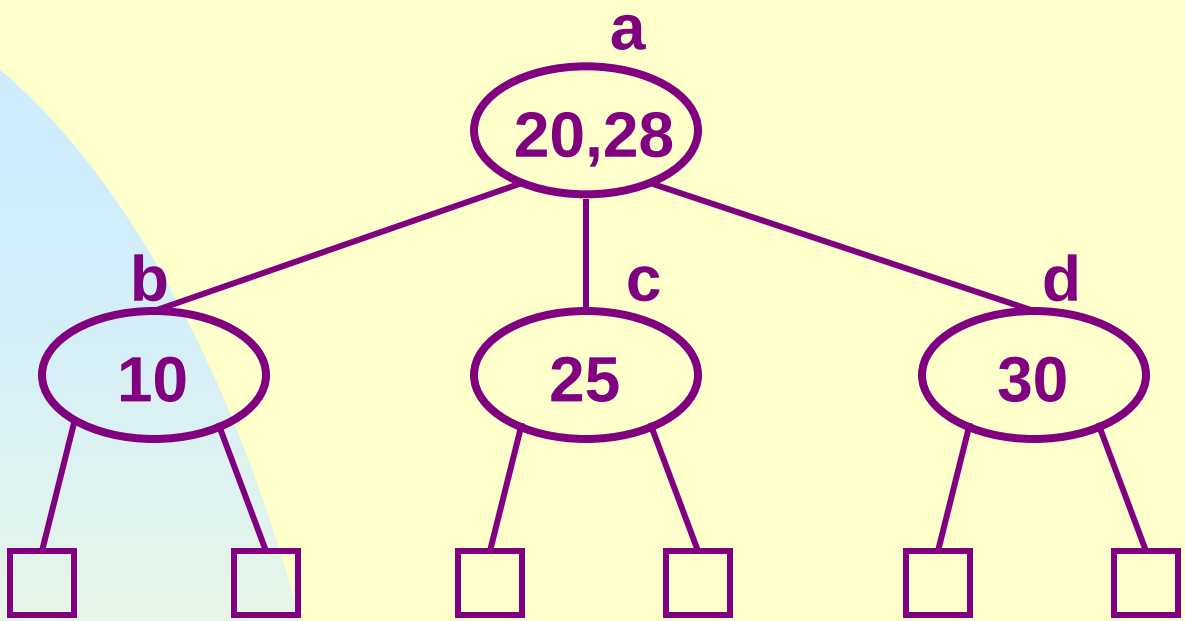
As shown in the following:



(a) p=1,s=0



(b) p=3,s=1



(c) p=4, s=2

A B-tree of order m with p nodes has at least

$$1 + (\lceil m/2 \rceil - 1)(p-1) \quad \text{elements.}$$

(root) (other nodes)

The average number of splitting s_{avg}

s_{avg} = (total number of splits)/N

$\leq (p-2)/\{1+ (\lceil m/2 \rceil -1)(p-1)\}$

$< 1/(\lceil m/2 \rceil -1)$

For m=200, $s_{avg} < 1/99$.

The number of disk accesses in an insertion is h+2s+1, where s is the number of nodes split.

So the average disk accesses is

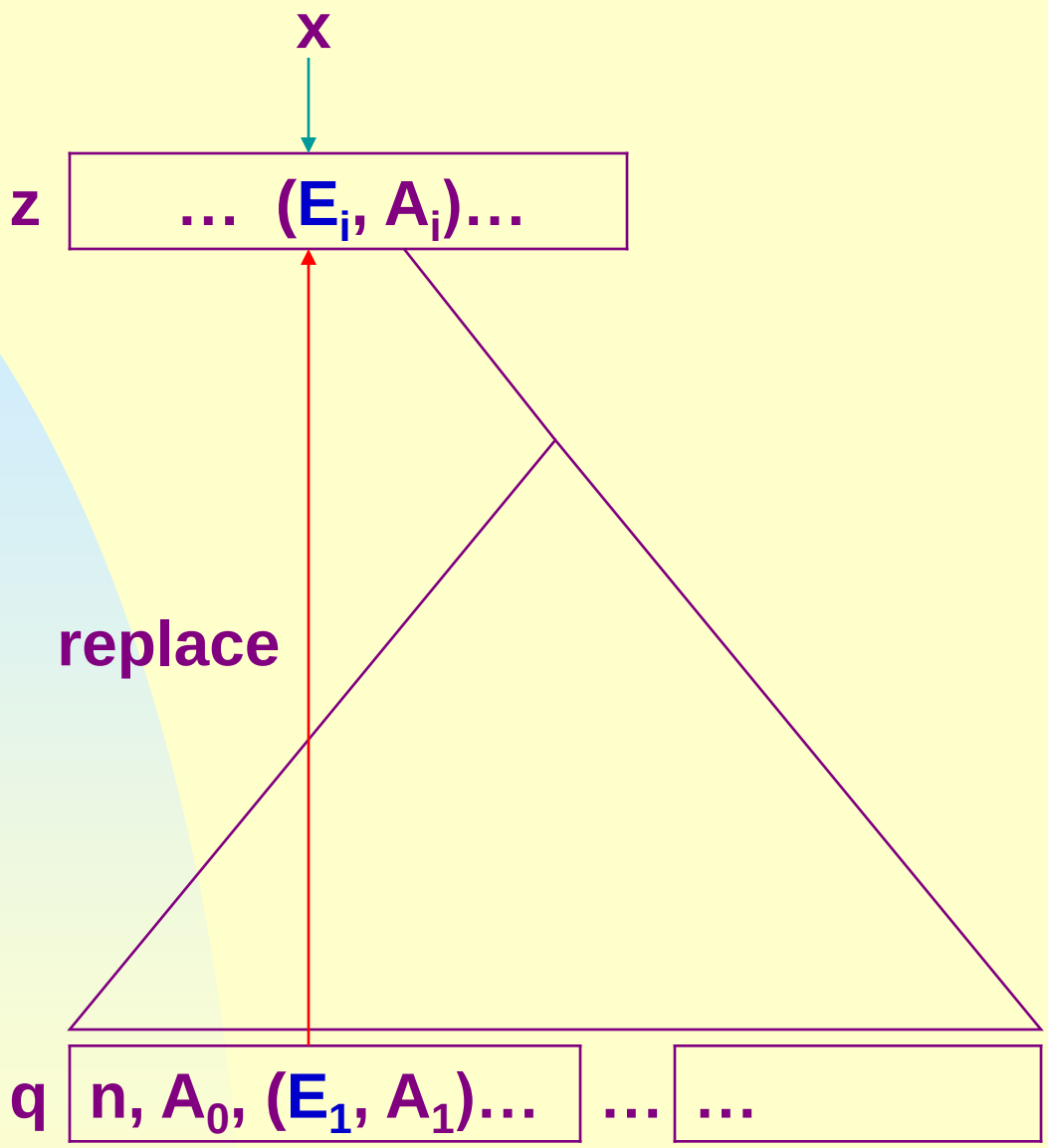
$$h+2\,s_{avg}+1 < h+2/99+1 \approx h+1.$$

11.2.4 Deletion from a B-Tree

Suppose the key of the element to be deleted is x.

First, search for x. If x is in a nonleaf node z and x = E_i.K, then the corresponding element may be replaced by either the element with smallest key in the subtree A_i or the element with largest key in the subtree A_{i-1}. Both are in leaf nodes.

The deletion from a nonleaf node is thus transformed into a deletion from a leaf, as shown in the next slide.


x
z
… (Ei, Ai)…
replace
q
n, A0, (E1, A1)…
…
…

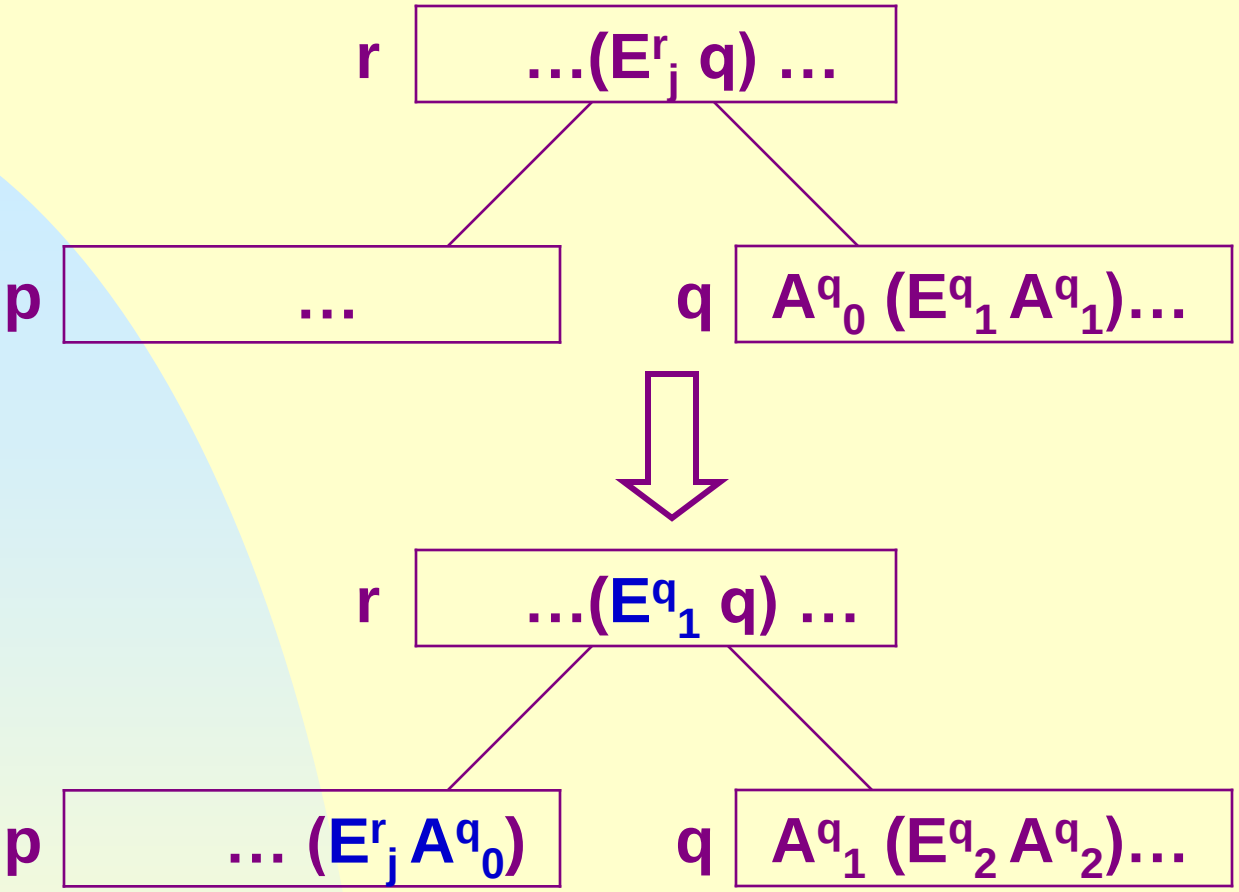
4 cases of deleting from a leaf node p:

(1) p is the root, if p is left with at least 1 key, p is written to disk, done. Otherwise the B-tree is empty following the deletion.

In the remaining cases, p is not the root.

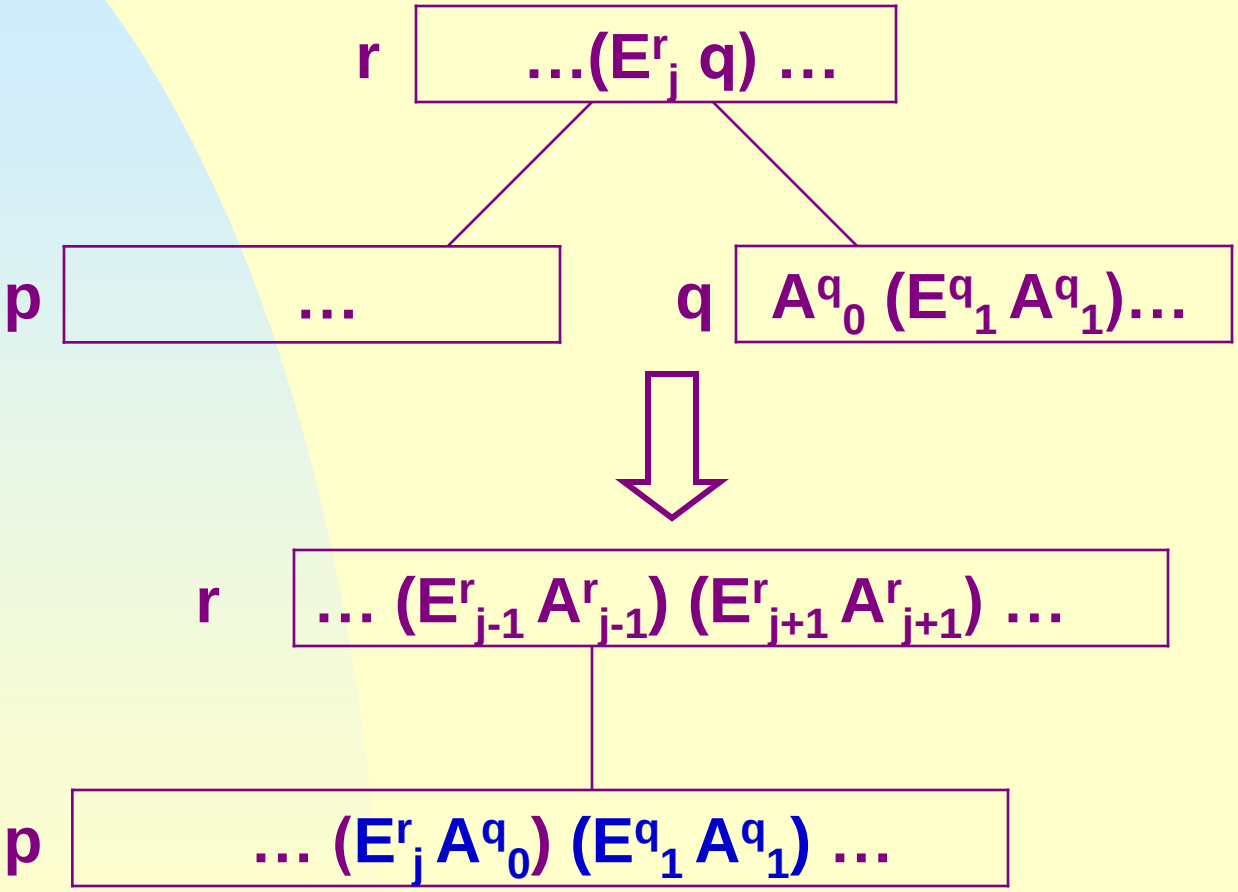
(2) Following the deletion, p has at least $\lceil m/2 \rceil - 1$ keys. The modified p is written to disk, done.

(3) p has $\lceil m/2 \rceil - 2$ keys, and its nearest sibling, q, has at least $\lceil m/2 \rceil$ keys. A rotation is performed as shown in the next slide (suppose q be the nearest right sibling of p, r be parent of p and q).



the changed p, q and r are written to disk, done.

(4) p has ⌈m/2⌉-2 keys, and q has ⌈m/2⌉-1 keys. A combination is performed as shown below:



Now p has (⌈m/2⌉-2)+ (⌈m/2⌉-1)+1=2⌈m/2⌉-2 ≤ m-1 keys. p is written to disk.

The number of keys in the parent, r, has been reduced by 1.

If r does not become deficient (i.e., it has at least 1 element if it is the root and ⌈m/2⌉-1 elements if it is not the root), the changed r is written to disk, done.

When r becomes deficient, if it is the root, it is discarded. Otherwise r has ⌈m/2⌉-2 keys, we can first attempt a rotation with one of its

nearest siblings. If this is not possible, a combine is done. This process of combining can continue up the B-tree until the children of the root are combined.

A high-level description of the deletion algorithm is given in the next slide:

// Delete element with key x from a B-tree of order m
Search the B-tree for node p containing the element with key x;
if (there is no such node p) **return;** // no element to delete
Let p be of the form: n, A_0, (E_1, A_1), …, (E_n, A_n) and E_i.K=x;
if (p is not a leaf) **{**
    Replace E_i with the smallest key element in subtree A_i;
    write the altered p to disk;
    Let p be the leaf of A_i from which the smallest was taken;
    Let p be of the form: n, A_0, (E_1, A_1), …, (E_n, A_n);
    i = 1;
}

// the following deletes E_i from leaf node p

Delete (E_i, A_i) from p; n--;
while (n<⌈m/2⌉-1) **&&** (p!=root)
  if (p has a nearest right sibling q) **{**
    Let q: n_q, A^q_0, $(E^q_1 A^q_1)$,…, $(E^q_{nq} A^q_{nq})$;
    Let r: n_r, A^r_0, $(E^r_1 A^r_1)$,…, $(E^r_{nr} A^r_{nr})$ be parent of p and q;
    Let A^r_j =q and A^r_{j-1}=p;
    if (n_q >= ⌈m/2⌉) **{** // rotation
      $(E_{n+1}, A_{n+1})= (E^r_j , A^q_0)$; n=n+1; // update p
      $E^r_j = E^q_1$; // update r
      $(n_q, A^q_0, (E^q_1 A^q_1),\ldots)=(n_q\text{-}1, A^q_1, (E^q_2 A^q_2),\ldots)$; //update q
      write p, q, r to disk; **return;**
    }
    // combine nodes p, E^r_j, and q
    s=2*⌈m/2⌉-2;

write s, A_0, (E_1 A_1),…,(E_n A_n), (E^r_j A^q_0), (E^q_1 A^q_1),
…,(E^q_{nq} A^q_{nq}) to disk as node p**;**

// update for next iteration

free(q)**;**

(n, A_0,…)=(n_r-1, A^r_0,…,(E^r_{j-1} A^r_{j-1}), (E^r_{j+1} A^r_{j+1})…)**;**

p=r**;**

} // end of **if** p has a nearest right sibling

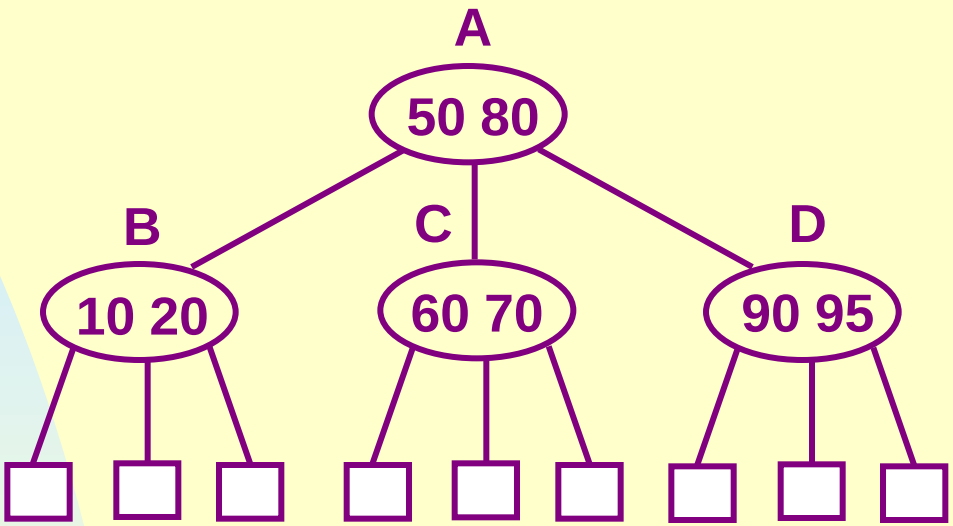
else { // p must have a left sibling, this is symmetric to the
// case where p has a right sibling

} // end of **if-else** and **while**

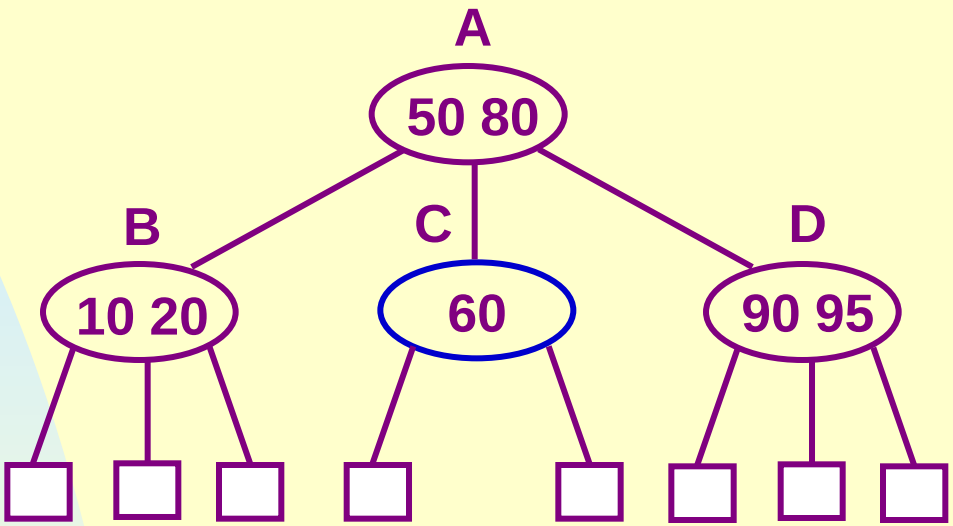
if (n) **write** p: (n, A_0, (E_1, A_1), …, (E_n, A_n))**;**

else { root= A_0**; free**(p)**;}** // change root, always combine to left

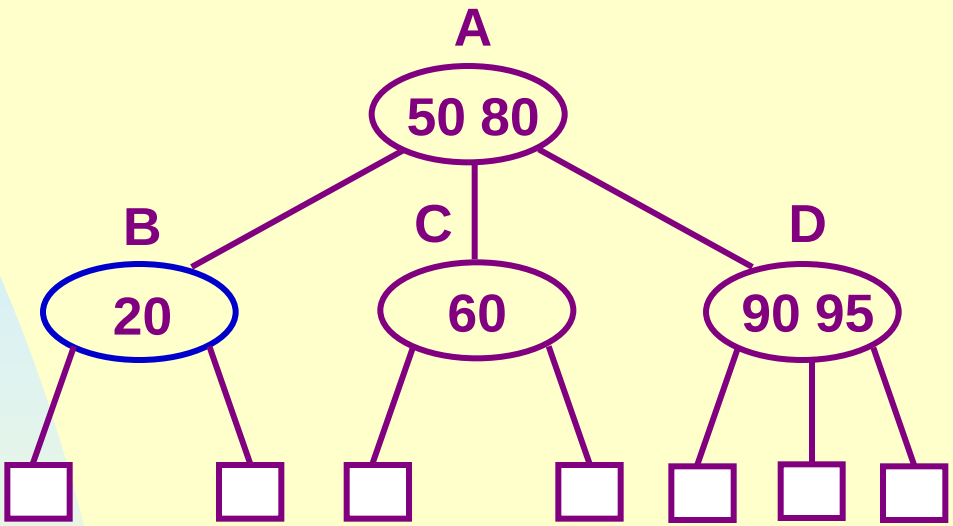
Example 11.2:



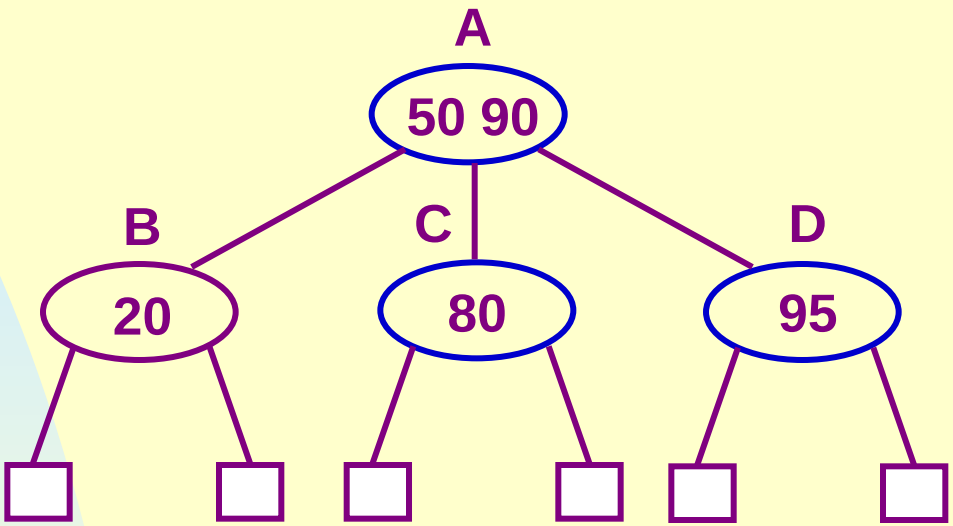
An initial 2-3 tree



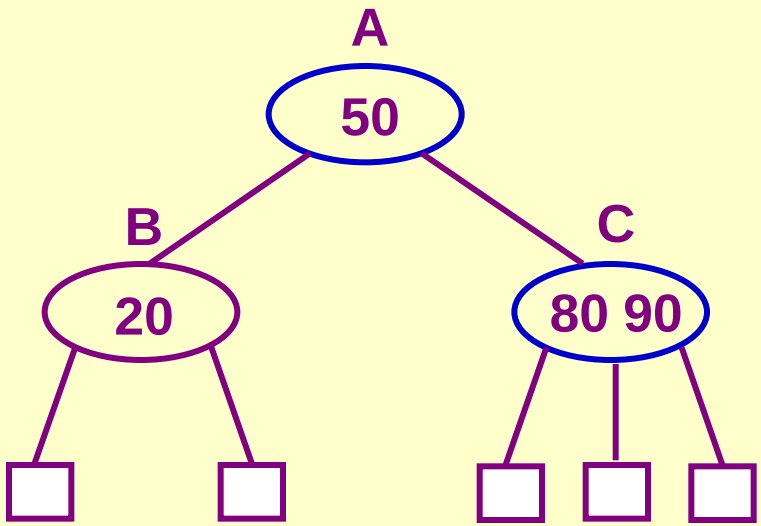
70 deleted



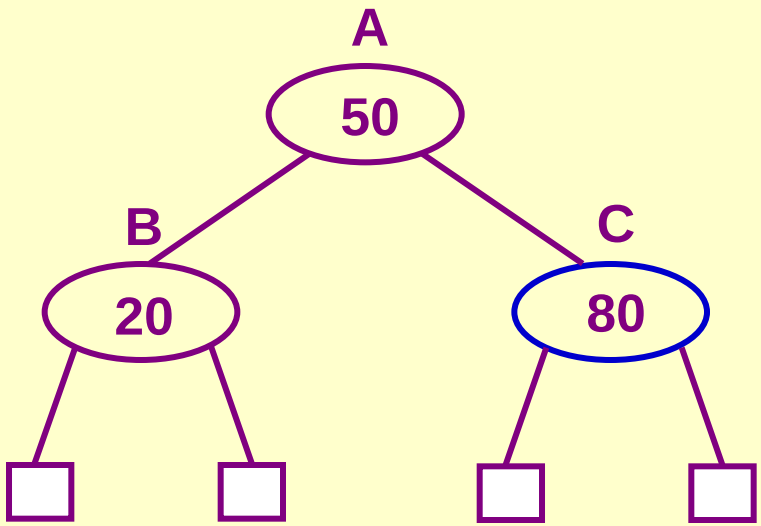
10 deleted



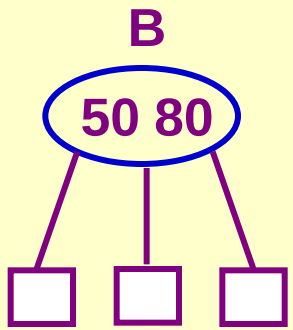
60 deleted



95 deleted



90 deleted



20 deleted

Analysis of B-tree Deletion:

h+1 disk accesses for finding the node from which the key is to be deleted and transforming the deletion to a one from a leaf.

In the worst, a combine takes place at each of the last h-2 nodes on the root-to-leaf path, and a rotation takes place at the 2^{nd} node on this path. The combines need: h-2 disk accesses for sibling and h-2 for writing out. The rotation needs 1 for sibling and 3 for writing out.

Total number of disk accesses is 3h+1.

Exercises: P623-2, 4

The end of the course

Thank you for your cooperation!