Data Structures

Chapter 1: Basic Concepts

Bit and Data Type

- bit: basic unit of information.
- data type: interpretation of a bit pattern.

e.g. **0100 0001** integer 65
ASCII code 'A'
BCD 41
(binary coded decimal)

1100 0001 unsigned integer 193
1's complement **
2's complement **

1's and 2's Complements

- range of 1's complement in 8 bits
  $-(2^7-1) \sim 2^7-1$
  $-127 \sim 127$
- range of 2's complement in 8 bits
  01111111 = 127
  10000001 = **
  10000000 = -128
  $-2^7 \sim 2^7-1$
  $-128 \sim 127$

Data Type in Programming

- data type:
  - a collection of values (objects) and a set of operations on those values.

e.g.
```
int x, y; // x, y, a, b are identifiers
float a, b;
x = x+y; // integer addition
a = a+b; // floating-point addition
```

Are the two additions same ?

Abstract Data Type

- Native data type
  - int (not realistic integer), float (real number), char
  - hardware implementation
- Abstract data type (ADT)
  - Specifications of objects and operations are separated from the object representation and the operation implementation
  - defined by existing data types
  - internal object representation and operation implementation are hidden.
  - by software implementation
  - examples: stack, queue, set, list, ...

ADT of NaturalNumber

ADT *NaturalNumber* is

objects: An ordered subrange of the integers starting at zero and ending at the maximum integer (MAXINT) on the computer

functions: for all *x*, *y* ∈*NaturalNumber*, TRUE, FALSE ∈ Boolean and where +, –, <, ==, and = are the usual integer operations

```
Nat_Num Zero()  ::= 0
Boolean IsZero(x) ::=if (x == 0) return true
                     else return false
Equal(x,y) :: ==     if (x == y) return true
                     else return false
Successor :: ==      …
Add(x, y) :: ==      if (x+y <= MAXINT) Add = x + y
                     else Add = MAXINT
Subtract(x,y) :: ==  …
```

end *NaturalNumber*

Iterative Algorithm for n Factorial

- Iterative definition of n factorial:

  n! = 1 if n = 0

  n! = n*(n-1)*(n-2)*…*1 if n>0

- Iterative C code:

```
int f = 1;
for (int x = n; x > 0; x--)
   f *= x;
return f;
```


Recursion for n Factorial

- recursive definition of n nactorial :

```
n! = 1   if n = 0
n! = n * (n-1)!  if n > 0
```

```
int fact(int n)
{
   if ( n == 0)  //boundary condition
      return (1);
   else                           **

}
```

An Example for Binary Search

- sorted sequence : (search 9)

	1	4	5	7	9	10	12	15
step 1				↑				
step 2						↑		
step 3					↑			

- used only if the table is sorted and stored in an array.
- Time complexity: O(logn)

Algorithm for Binary Search

- Algorithm (pseudo code):

```
while (there are more elements) {
  middle = (left + right) / 2;
  if (searchnum < list[middle])
     right = middle – 1;
  else if (searchnum > list[middle])
     left = middle + 1;
  else return middle;
}
Not found;
```


Iterative C Code for Binary Search

```
int BinarySearch (int *a, int x, const int n)
// Search a[0], ..., a[n-1] for x
{
  int left = 0,  right = n - 1;
  while (left <= right;) {
    int middle = (left + right)/2;
    if (x < a[middle]) right = middle - 1;
    else if (x > a[middle]) left = middle + 1;
    else return middle;
  } // end of while
  return -1; // not found
}
```


Recursion for Binary Search

```
int BinarySearch (int *a, int x, const int left,
  const int right)
//Search a[left], ..., a[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);
    else return middle;
  } // end of if
  return -1;// not found
}
```

Recursive Permutation Generator

- Example: Print out all possible permutations of $\{a, b, c, d\}$
  - We can construct the set of permutations:
    - *a* followed by all permutations of (*b*, *c*, *d*)
    - *b* followed by all permutations of (*a*, *c*, *d*)
    - *c* followed by all permutations of (*b*, *a*, *d*)
    - *d* followed by all permutations of (*b*, *c*, *a*)
  - Summary
    - *w* followed by all permutations of (*x*, *y*, *z*)


```
#include <iostream>
void Permutations (char *a, const int k, const int m)
//Generate all the 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], ..., a[m] has more than one permutation
      for (int i = k; i <= m; i++)
      {
         swap(a[k], a[i]); // exchange
         Permutations(a, k+1, m);
         swap(a[k], a[i]);
      }
    } // end of else
} // end of Permutations
```


Permutation – main() of perm.cpp

```
int main()
{
   char b[10];
   b[0] = 'a'; b[1] = 'b'; b[2] = 'c'; b[3] = 'd';
   b[4] = 'e'; b[5] = 'f'; b[6] = 'g';

   Permutations(b,0,2);
   cout << endl
}
```

Output: **

The Criteria to Judge a Program

- Does it do what we want it to do?
- Does it work correctly according to the original specification of the task?
- Is there documentation that describes how to use it and how it works?
- Do we effectively use functions to create logical units?
- Is the code readable?
- Do we effectively use primary and secondary storage?
- Is running time acceptable?

Fibonacci Sequence (1)

- 0,1,1,2,3,5,8,13,21,34,...
- Leonardo Fibonacci (1170 -1250)

  用來計算兔子的數量

  每對每個月可以生產一對

  兔子出生後, 隔一個月才會生產, 且永不死亡

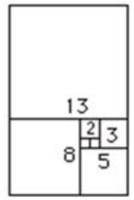
  生產　0 1 1 2 3 ...

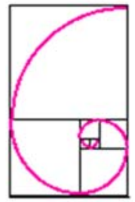
  總數 1 1 2 3 5 8 ...

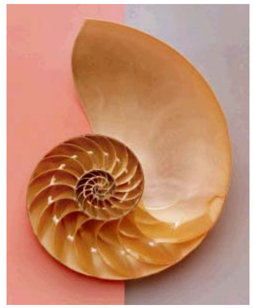
https://r-knott.surrey.ac.uk/Fibonacci/fibnat.html

Fibonacci Sequence (2)

- 0,1,1,2,3,5,8,13,21,34,...





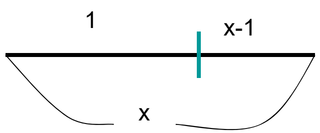


Fibonacci Sequence and Golden Number

- 0,1,1,2,3,5,8,13,21,34,...

$$
\begin{cases}
f_n = 0 & \text{if } n = 0 \\
f_n = 1 & \text{if } n = 1 \\
f_n = f_{n-1} + f_{n-2} & \text{if } n \geq 2
\end{cases}
$$

$$\lim_{n\to\infty} \frac{f_n}{f_{n-1}} = \frac{1+\sqrt{5}}{2} = \text{Golden number}$$



$$\frac{x}{1} = \frac{1}{x-1}$$

$$x^2 - x - 1 = 0$$

$$x = \frac{1+\sqrt{5}}{2}$$

Iterative Code for Fibonacci Sequence

$$
\boxed{
\begin{array}{l}
f_n = 0 \quad \text{if } n = 0 \\
f_n = 1 \quad \text{if } n = 1 \\
f_n = f_{n-1} + f_{n-2} \text{ if } n \geq 2
\end{array}
}
$$

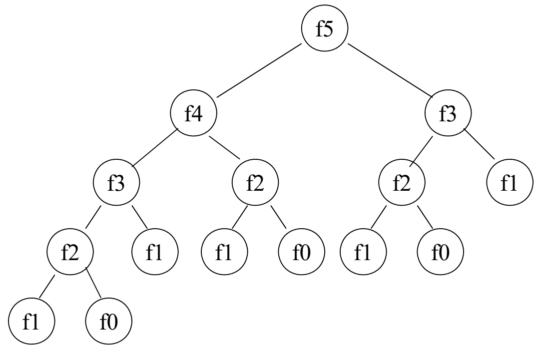
```c
int fib(int n)
{
  int i, x, logib, hifib;

  if (n <= 1)
     return(n);
   lofib = 0;
   hifib = 1;
   for (i = 2; i <= n; i++){
     x = lofib;       /* hifib,  lofib */
     lofib = hifib;
     hifib = x + lofib; /* hifib = lofib + x */
   } /* end for */
   return(hifib);
}
```

Recursion for Fibonacci Sequence

```
int fib(int n)
{
  int x, y;

  if (n <= 1)
    return(n);          **

}
```

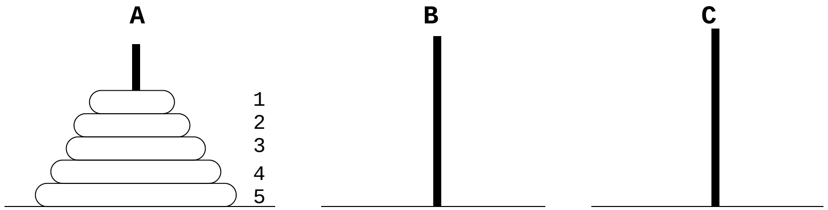
$f_n = 0$ if $n = 0$
$f_n = 1$ if $n = 1$
$f_n = f_{n-1} + f_{n-2}$ if $n \geq 2$

Tree of Recursive Computation of Fibonacci Sequence



- Much computation is duplicated.
- The iterative algorithm for generating Fibonacci sequence is better.

The Towers of Hanoi Problem



The initial setup of the Towers of Hanoi.

- Disks are of different diameters
- A larger disk must be put below a smaller disk
- Object: to move the disks, one each time, from peg A to peg C, using peg B as auxiliary.

Strategy for Moving Disks

- how to move 3 disks?

**

- how to move n disks?

**

Recursive Program for the Tower of Hanoi Problem

```
#include <stdio.h>
void towers(int, char, char, char);

void main()
{
  int n;
  scanf("%d", &n);
  towers(n, 'A', 'B', 'C');
} /* end of main */
```



```
void towers(int n, char A, char B, char C)
{
  if ( n == 1){
    // If only one disk, make the move and return.
    printf("\n%s%c%s%c", "move disk 1 from peg ",
           A, " to peg ", C);
    return;
  }
 /*Move top n-1 disks from A to B, using C as auxiliary*/
  towers(n-1, A, C, B);
 /* move remaining disk from A to C */
  printf("\n%s%d%s%c%s%c", "move disk ", n,
       " from peg ", A, " to peg ", C);
 /* Move n-1 disk from B to C, using A as auxiliary */
  towers(n-1, B, A, C);
} /* end towers */
```


Number of Disk Movements

T(n) : # of movements with n disks

We know $T(1) = 1$ -- boundary condition

$T(2) = 3$

$T(3) = 7$

$$
\begin{aligned}
T(n) &= T(n-1) + 1 + T(n-1) \\
&= 2T(n-1) + 1 \\
&= 2(2T(n-2) + 1) + 1 \\
&= 4T(n-2) + 2 + 1 \\
&= 8T(n-3) + 4 + 2 + 1 \\
\\
&= 2^{n-1}\, T(n-(n-1)) + 2^{n-2} + 2^{n-3} + \ldots + 1 \\
&= 2^{n-1}\, T(1) + 2^{n-2} + 2^{n-3} + \ldots + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
&= 2^n - 1
\end{aligned}
$$

Asymptotic O Notation

$f(n)$ is $O(g(n))$ if there exist positive integers a and b such that $f(n) \leqq a \cdot g(n)$ for all $n \geqq b$

e.g. $4n^2 + 100n = O(n^2)$

$\because n \geqq 100,\ 4n^2+100n \leqq 5n^2$

$4n^2 + 100n = O(n^3)$

$\because n \geqq 10,\ 4n^2+100n \leqq 2n^3$

$f(n)= c_1n^k + c_2n^{k-1} +\ldots+ c_kn + c_{k+1}$

$= O(n^{k+j})$, for any $j \geqq 0$

$f(n)= c = O(1)$, c is a constant

$\log_m n = \log_m k \cdot \log_k n$, for some constants m and k

$\log_m n = O(\log_k n) = O(\log n)$

Values of Functions

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Time Complexity and Space Compexity

- Time Complexity: The amount of computation time required by a program
- Polynomial order: $O(n^k)$, for some constant k.
- Exponential order: $O(d^n)$, for some d >1.
- NP-complete(intractable) problem:
  require exponential time algorithms today.
- Best sorting algorithm with comparisons: O(nlogn)
- Space complexity: The amount of memory required by a program

Selection Sort (1)

e.g. 由小而大 sort (nondecreasing order)

5 9 2 8 6 pass 1
2 9 5 8 6 pass 2
2 5 9 8 6 pass 3
2 5 6 8 9 pass 4
2 5 6 8 9

- 方法: 每次均從剩餘未 sort 部份之資料, 找出最大者(或最小者), 然後對調至其位置
- **Number of comparisons (比較次數):**

$(n-1)+(n-2)+\ldots+1 = \frac{n(n-1)}{2} = O(n^2)$

Time complexity: $O(n^2)$

Selection Sort (2)

- Sort a collection of n integers.
  - From those integers that are currently unsorted, find the smallest and place it next in the sorted list.
- The algorithm (pseudo code):

```
for (int i = 0; i < n ; i++)
{
   /* Examine a[i] to a[n-1] and suppose
      the smallest integer is at a[j]; */
   //Interchange a[i] and a[j];
}
```

C Code for Selection Sort

```
void sort (int *a, int n)
{ // sort the n integers a[0]~a[n-1] into nondecreasing order
    for ( int i = 0; i < n; i++)
    {
        int j = i;  // find the smallest in a[i] to a[n-1]
        for (int k = i+1; k < n; k++)
            if (a[k] < a[j]) j = k;
        // swap(a[i], a[j]);
        int temp = a[i]; a[i] = a[j]; a[j] = temp;
    }
}
```


Time Performance in C Code (1)

```
#include <time.h>
…
time_t start, stop;
…
start = time(NULL); /* # of sec since
  00:00 hours, Jan 1, 1970 UTC (current
  unix timestamp) */
… //the program to be evaluated
stop = time(NULL);
duration = ((double) difftime(stop,
  start);
```


Time Performance in C Code (2)

```
#include <time.h>
…
clock_t start, stop;
…
start = clock(); /* # of clock
  ticks since the program begins */
… //the program to be evaluated
stop = clock();
duration = ((double) (stop -
  start) ) / CLOCKS_PER_SEC;
```

Data Structures

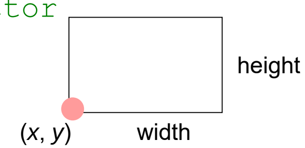
Chapter 2: Arrays

Abstract Data Type and C++ Class

- Four components in a C++ Class
  - class name
  - data members: the data that makes up the class
  - member functions: the set of operations that may be applied to the objects of class
  - levels of program access:
    - private: accessible within the same class
    - protected: accessible within the same class and derived classes
    - public: accessible anywhere

Rectangle.h –Definition of Rectangle Class

```
// in the header file Rectangle.h
#ifndef RECTANGLE_H /* preprocessor
             directive, "if not defined" */
#define RECTANGLE_H
class Rectangle //  class name: Rectangle
{
  public:
     // 4 member functions
     Rectangle();    // constructor
     ~Rectangle();  // destructor
     int GetHeight();
     int GetWidth();
  private:
     // 4 data members
     int xLow, yLow, height, width;
     // (xLow, yLow) is the bottom left corner
};
#endif
```



Rectangle.cpp – Operation Implementation

```
// in the source file Rectangle.cpp
#include <Rectangle.h>

int Rectangle::GetHeight() {return height;}
int Rectangle::GetWidth() {return width;}
```

- ADT: Abstract Data Type
  - Rectangle.h: Object representation (or Class Definition)
  - Rectangle.cpp: Operation implementation

Constructor(1)

- Constructor (建構子): the name of the function is identical to its class name, to initialize the data members

```
Rectangle::Rectangle(int x, int y, int h,
      int w)
{
   xLow = x;   yLow = y;
   height = h;   width = w;
}
int main()
{
 Rectangle r(1,3,6,6); // call constructor
 Rectangle *s=new Rectangle(0,0,3,4);
 return 0;
}
```


Constructor(2)

- Another way of Constructor initialization:

```
Rectangl::Rectangle(int x = 0, int y = 0,
int h = 0, int w = 0)
 : xLow(x), yLow(y), height(h), width(w)
{ }

int main()
{
    Rectangle t; // all set to 0
    return 0;
}
```


Operator Overloading

```
int Rectangle::operator==(const Rectangle& s)
      // overload "=="
  {
     if (this == &s) return 1;
     else if ( (xLow == s.xLow) && (yLow == s.yLow) &&
          (height == s.height) && (width == s.width) )
        return 1;
     return 0;
   }


ostream& Rectangle:: operator<<(ostream& os,
     Rectangle& r)  // overload "<<"
{
   os << "Position is: " << r.xLow << " ";
   os << r.yLow << endl;
   os << "Height is: " << r.height << endl;
   os << "Width is: " << r.width << endl;
   return os;
}
```


Complete Program for Rectangle

```
#include <iostream>
using namespace std;

class Rectangle
{
  friend ostream& operator<< (ostream&, Rectangle&);
  public:
      int Rectangle::operator==(const Rectangle&)
      Rectangle();      // constructor
      ~Rectangle();     // destructor
      int GetHeight();  int GetWidth();
  private:
      int xLow, yLow, height, width;
};
 ...
int main()
{
   Rectangle r(1,3,6,6),t(1,3,6,6); // call constructor
   if (r == t) cout << 1 << endl; // overloaded "=="
   cout << r; // overloaded "<<"
   return 0;
}
```

Array as ADT

- Array Objects:
  - A set of pairs <index, value> where for each value of index (索引) there is a corresponding value.
  - Index set is a finite ordered set of one or more dimensions. For example, {0,…,n–1} for one dimension, {(0,0), (0,1),(0,2),(1,0),(1,1),…,(3,2)…} for two dimensions.

```
class GeneralArray{
 public:
  GeneralArray(int j, RangeList list, float initValue
  = defaultValue);
  // create a linear (1D) array of size j
  float Retrieve(index i);
  // return the value of index i
  void Store(index i, float x);
  // change the value of index i to x
}
```


1-dimentional Array in C/C++

```
// array declaration (陣列宣告)
int a[10]; // no intialization
float f[7];

int a[4]={10,11};
  // a[0]=10, a[1]=11, a[2]=0, a[3]=0

int a[]={10,11,12}; // array size:3
  // a[0]=10, a[1]=11, a[2]=12
bool b[] = {true, false, true};
char c[] = {'B', 'C', '\0', '\n'};
char d[] = "BED";  // 4 bytes
```


Programming skills

```
int a[100];
for (i = 0; i < 100; a[i++] = 0);
```

Better:Constants defined by identifiers

```
#define NUMELTS 100
int a[NUMELTS];
for (i = 0; i < NUMELTS; a[i++] = 0);
```


Representation of Arrays

- Representation of one dimensional array

α	α+1	α+2	α+3	α+4	α+5	α+6	α+7	α+8	α+9	α+10	α+11
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]

- Multidimensional arrays can be implemented by one dimensional array via either row major order or column major order.

Two Dimensional (2D) Arrays

2-dimensional array:

int [ ][ ]a = new int[3][4]; // in C++

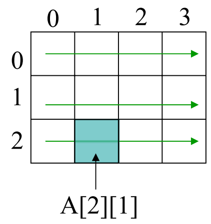
int a[3][4]; // in C/C++

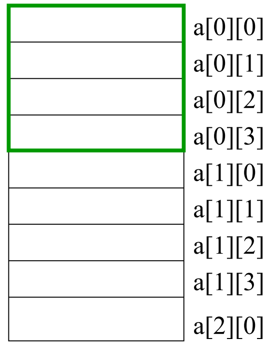
It can be shown as a table:

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Row-major Order (from 2D to 1D)

- int a[3][4]; // row-major order
- logical structure physical structure





Mapping function:
Address of a[*i*][*j*]= **
where one integer occupies 4 bytes.

Memory Mapping of Row-major Order

- α: the start address of array *a*
  - refer to *a*[0], *a*[0][0], *a*[0][0][0], etc.
- β: the memory size for storing each element
  - char: 1 byte, β=1
  - int: 4 bytes, β=4
- 2D to 1D
  - *int (char)* $a[u_1][u_2]$
  - $a[i][j] = \alpha + (i \times u_2 + j) \times \beta$
- 3D to 1D
  - *int (char)* $a[u_1][u_2][u_3]$
  - $a[i][j][k] = \alpha + (i \times u_2 \times u_3 + j \times u_3 + k) \times \beta$

Column-Major Order

```
a b c d
e f g h
i j k l
```

- Convert 2D into 1D by collecting elements first by columns, from left to right..
- Within a column, elements are collected from top to bottom.
- We get *y* = {a, e, i, b, f, j, c, g, k, d, h, l}

Memory Mapping of Column-major Order

- α: the start address of array *a*
  - refer to *a*[0], *a*[0][0], *a*[0][0][0], etc.
- β: the memory size for storing each element
  - char: 1 byte, β=1
  - int: 4 bytes, β=4
- 2D to 1D
  - *int (char)* $a[u_1][u_2]$
  - $a[i][j] = \alpha + (i + j \times u_1) \times \beta$
- 3D to 1D
  - *int (char)* $a[u_1][u_2][u_3]$
  - $a[i][j][k] = \alpha + (i + j \times u_1 + k \times u_1 \times u_2) \times \beta$

Sets in Pascal

var x, x1, x2: set of 1..10;
x = [1, 2, 3, 5, 9, 10]

A set is represented by a bit string.
e.g. 1110100011 1: in the set
0: not in the set

e.g. x1 = [2, 4, 6, 8] 0101010100
x2 = [1, 2, 5, 9] 1100100010

- **union** 聯集:
  x1 ∪ x2 = **
  x1 + x2 = [1, 2, 4, 5, 6, 8, 9] **

x1 = [2, 4, 6, 8]	0101010100
x2 = [1, 2, 5, 9]	1100100010

Intersection 交集:
x1 ∩ x2 = **
x1 * x2 = [2] **

difference 差集:
x1 – x2 = **
x1 – x2 = [4, 6, 8] **

contain 包含:
x1 ⊆ x2 ⇔
x1 ⊇ x2 ⇔ **
a in x1 ⇔

- The size of a set is limited.

Polynomial (多項式)

$f(x) = x^8 - 10x^5 + 3x^3 + 1.5$

- 4 terms: x^8, $-10x^5$, $3x^3$, 1.5
- Coefficients(係數): 1, – 10, 3, 1.5
- Nonnegative integer exponents(指數, 冪): 8, 5, 3, 0

Polynomial Addition

$f(x) = x^8 - 10x^5 + 3x^3 + 1.5$

$g(x) = 3x^3 + 2x - 4$

$$
\begin{aligned}
f(x) &= x^8 - 10x^5 + 3x^3 \qquad + 1.5 \\
g(x) &= \qquad\qquad\quad 3x^3 + 2x - 4 \\
f(x) + g(x) &= x^8 - 10x^5 + 6x^3 + 2x - 2.5
\end{aligned}
$$

- $a(x) + b(x) = \Sigma(a_i + b_i)x^i$

Polynomial Representation (1)

```
#define MaxDegree 100
class Polynomial{
  private:
     int degree;     // degree ≤ MaxDegree
     float coef [MaxDegree + 1];
};
```

$f(x) = x^8 - 10x^5 + 3x^3 + 1.5$

0	1	2	3	4	5	6	7	8	9	10	11	12	…
1.5	0	0	3	0	-10	0	0	1	0	0	0	0	0...

Polynomial Representation (2)

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

$f(x) = x^8 - 10x^5 + 3x^3 + 1.5$

0	1	2	3	4	5	6	7	8	9	[illegible]	11	[illegible]	…
1.5	0	0	3	0	-10	0	0	1	0	0	[illegible]	[illegible]	0...

Polynomial Representation (3)

- Add the following two polynomials:

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

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

\# of elements

	0	1	2
a_coef	2	2	1
a_exp		1000	0

	0	1	2	3	4
b_coef	4	1	10	3	1
b_exp		4	3	2	0

	0	1	2	3	4	5
c_coef	5	2	1	10	3	2
c_exp		1000	4	3	2	0

Sparse Matrices (稀疏矩陣)

- Most elements of the matrix are of zero

0	0	0	0	0	0	Row 0
0	0	0	3	0	4	Row 1
0	0	0	5	7	0	Row 2
0	0	0	0	0	0	Row 3
0	0	2	6	0	0	Row 4

Column 4

5 × 6 matrix (5 by 6)

5 rows

6 columns

30 elements

6 nonzero elements

Matrices

$$\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix}$$

Dense matrix 5×3

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

Sparse matrix

Matrix Transpose in a 2-D Array

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}_{4\times 3} \longrightarrow B = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}_{3\times 4}$$

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

Time Complexity: O(rows × columns)

Sparse Matrix Representation

- Use triples (三元組): (row, column, value)
  - (1,3,3), (1, 5, 4), (2,3,5), (2,4,7), (4,2,2), (4,3,6)
- To perform matrix operations, such as transpose, we have to know the number of rows, number of columns, and number of nonzero elements.

0	0	0	0	0	0
0	0	0	3	0	4
0	0	0	5	7	0
0	0	0	0	0	0
0	0	2	6	0	0

Linear List Representation of Matrix

0	0	0	0	0	0
0	0	0	3	0	4
0	0	0	5	7	0
0	0	0	0	0	0
0	0	2	6	0	0

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

Linear List for a Matrix

- Class SparseMatrix
  - Array smArray of triples of type MatrixTerm
    - int row, col, value   // for a triple (row, col, value)
  - int rows,   // number of rows in this matrix
    cols,   // number of columns in this matrix
    terms, // number of nonzero elements in this matrix
    capacity; // size of smArray
- Size of smArray generally not predictable at time of initialization
  - Start with some default capacity/size (say 10)
  - Increase capacity as needed

C++ Class for Sparse Matrix

```
class SparseMatrix; // forward declaration
class MatrixTerm {
   friend class SparseMatrix
   private:
       int row, col, value;
};
class SparseMatrix {
. . .
private:
   int Rows, Cols, Terms;
   MatrixTerm smArray[MaxTerms];
}
```


Matrix Transpose

- Intuitive way:

  for (each row i)

    take element (i, j, value) and store it in (j, i, value) of the transpose

row	1	1	2	2	4	4		3	5	3	4	2	3
column	3	5	3	4	2	3	→	1	1	2	2	4	4
value	3	4	5	7	2	6		3	4	5	7	2	6

- More efficient way:

  for (all elements in column j)

    place element (i, j, value) in position (j, i, value)

Efficient Way for Matrix Transpose

M

0 0 0 0 0 0
0 0 0 3 0 4
0 0 0 5 7 0
0 0 0 0 0 0
0 0 2 6 0 0

→ M^T

0 0 0 0 0
0 0 0 0 0
0 0 0 0 2
0 3 5 0 6
0 0 7 0 0
0 4 0 0 0

							→						
row	1	1	2	2	4	4		2	3	3	3	4	5
column	3	5	3	4	2	3	→	4	1	2	4	2	1
value	3	4	5	7	2	6		2	3	5	6	7	4

Transposing a Matrix

```
SparseMatrix SparseMatrix::Transpose()
// return the transpose of a (*this)
{
   SparseMatrix b;
   b.Rows = Cols;    // rows in b = columns in a
   b.Cols = Rows;    // columns in b = rows in a
   b.Terms = Terms;  // terms in b = terms in a
   if (Terms > 0)    // nonzero matrix
   {
        int CurrentB = 0;
        for (int c = 0; c < Cols; c++) // transpose by columns
            for (int i = 0; i < Terms; i++)
              // find elements in column c
               if (smArray[i].col == c) {
                  b.smArray[CurrentB].row = c;
                  b.smArray[CurrentB].col = smArray[i].row;
                  b.smArray[CurrentB].value = smArray[i].value;
                  CurrentB++;
               }
   }  // end of if (Terms > 0)
    return b;
}  // end of transpose
```


Time Complexity

- **Time Complexity:**

  O(terms × columns)

  =O(rows × columns × columns)
- A better transpose function
  - It first computes the number of nonzero elements in each columns of matrix A before transposing to matrix B. Then it determines the starting address of each row for matrix B. Finally, it moves each nonzero element from A to B.

Faster Matrix Transpose

0 0 0 0 0 0
0 0 0 3 0 4
0 0 0 5 7 0
0 0 0 0 0 0
0 0 2 6 0 0

→

0 0 0 0 0
0 0 0 0 0
0 0 0 0 2
0 3 5 0 6
0 0 7 0 0
0 4 0 0 0

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

Step 1: #nonzero in row i of transpose

= #nonzero in column i of original matrix

= [0, 0, 1, 3, 1, 1]

Step 2: Start of row i of transpose

= size sum of rows 0,1,2, i-1 of transpose

= [0, 0, 0, 1, 4, 5]

Step 3: Move elements, left to right, from original list to transpose list

Faster Matrix Transpose

$$\begin{matrix}0&0&0&0&0&0\\0&0&0&3&0&4\\0&0&0&5&7&0\\0&0&0&0&0&0\\0&0&2&6&0&0\end{matrix} \longrightarrow \begin{matrix}0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&2\\0&3&5&0&6\\0&0&7&0&0\\0&4&0&0&0\end{matrix}$$

rowStart =
[0, 0, 0, 1, 4, 5]

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

```
- 3 - - - -      - 3 - - - 5      - 3 3 - - 5
- 1 - - - -      - 1 - - - 1      - 1 2 - - 1
- 3 - - - -      - 3 - - - 4      - 3 5 - - 4
```

rowStart =
[0, 0, 0, 2, 4, 5]

rowStart =
[0, 0, 0, 2, 4, 6]

rowStart =
[0, 0, 0, 3, 4, 6]

Time Complexity

Step 1: #nonzero in each row
O(terms)
Step2: Start address of each row
O(columns)
Step 3: Data movement
O(terms)

- Total time complexity:
O(terms + columns)= O(rows ×columns)

Matrix Multiplication

- Given A and B, where A is $m \times n$ and B is $n \times p$, the product matrix D = A × B has dimension $m \times p$.
- Let d_{ij}=D[i][j]

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

where $0 \le i \le m-1$ and $0 \le j \le p-1$.

Matrix Multiplication Example (1)

$$A_{5\times 6}= \begin{matrix}0&0&0&0&0&0\\0&0&0&3&0&4\\0&0&0&5&7&0\\0&0&0&0&0&0\\0&0&2&6&0&0\end{matrix} \qquad B_{6\times 3}= \begin{matrix}0&0&0\\0&2&0\\0&0&0\\0&3&0\\0&4&7\\0&1&0\end{matrix}$$

$$D_{5\times 3}= A \times B = \begin{matrix}0&0&0\\0&13&0\\0&43&49\\0&0&0\\0&18&0\end{matrix}$$

Matrix Multiplication Example (2)

$A_{5\times6}$ =

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

$B_{6\times3}$=

row	1	3	4	4	5
column	1	1	1	2	1
value	2	3	4	7	1

$D_{5\times3} = A \times B =$

```
row      1    2    2    4
column   1    1    2    1
value   13   43   49   18
```


String in C/C++

```
char c[] = {'B', 'C', 'D', '\0'};
char d[] = "BED";   // 4 bytes
```

- In C/C++, a string is represented as a character array

0	1	2	3	4
B	E	D	\0	

- Internal representation

0	1	2	3	4
66	69	68	0	

String in C

- General string operations include comparison(比較), string concatenation(連接), copy, insertion, string matching(匹配,比對), printing
- Function in C: #include <string.h>
  - strcat, strncat
  - strcmp, strncmp
  - strcpy, strncpy, strlen
  - strchr, strtok, strstr, …
- #include <cstring> // in C++ for string.h

String Matching Problem

- Given a text string *T* of length *n* and a pattern string *P* of length *m*, the exact string matching problem is to find all occurrences of *P* in *T*.
- Example: T="AGCTTGCTA" P="GCT"
- Applications:
  - Searching keywords in a file (Text Editor)
  - Searching engines (Google)
  - Database searching (GenBank)

KMP Algorithm for String Matching

- KMP algorithm
  - Proposed by Knuth, Morris and Pratt
  - Time complexity: O(m+n)

- More string matching algorithms (with source codes):
  http://www-igm.univ-mlv.fr/~lecroq/string/

Data Structures

Chapter 3: Stacks And Queues

Templates in C++

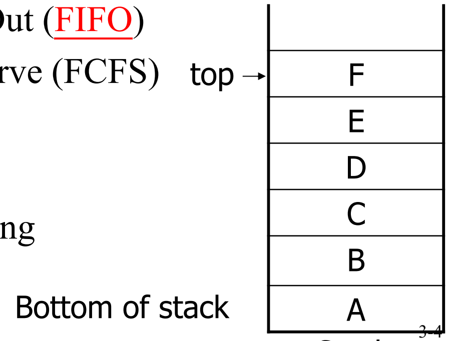
- Template is a mechanism provided by C++ to make classes and functions more reusable.
- A template can be viewed as a variable that can be instantiated (示例) to any data type, irrespective (不論) of whether this data type is a fundamental C++ type or a user-defined type.
- Template function and template class

Selection Sort Template Function

```
template <class T>
void SelectionSort(T *a, const int n)
{ // sort a[0] to a[n-1] into nondecreasing order
   for ( int i = 0; i < n; i++)
   {
       int j = i;
       // find the smallest in a[i] to a[n-1]
       for (int k = i+1; k < n; k++)
          if (a[k] < a[j]) j = k;
       swap(a[i], a[j]);
   }
}

float farray[100];
int intarray[200];
// ...
// assume that the arrays are initialized at this point
SelectionSort(farray, 100);
SelectionSsort(intarray, 200);
```

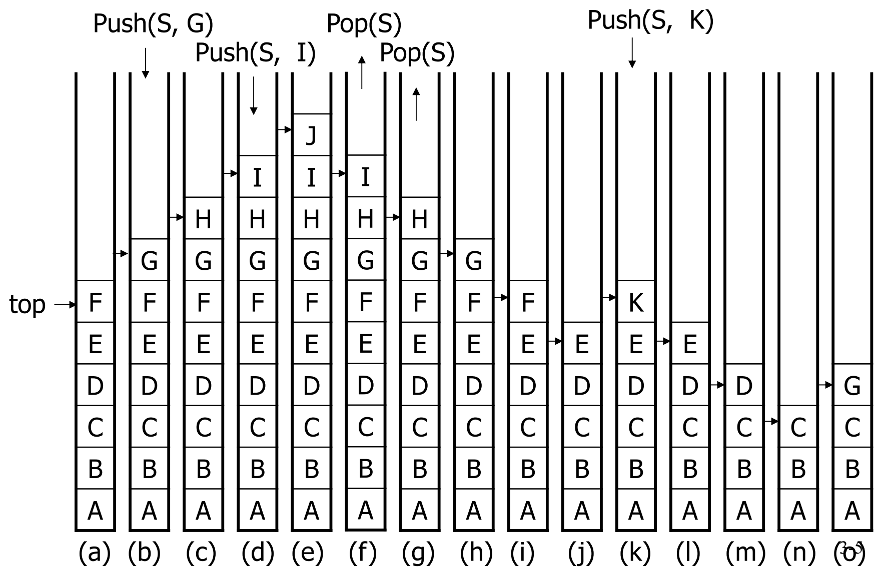

Stack and Queue

- **Stack**: Last-In-First-Out (LIFO)
  - Last-Come-First-Serve (LCFS)
  - only one end
- **Queue**: First-In-First-Out (FIFO)
  - First-Come-First-Serve (FCFS)
  - 2 ends
  - one for entering
  - the other for leaving



Stack

A Motion Picture of a Stack



Operations of a Stack

- IsEmpty( ): check whether the stack is empty or not.
- Top ( ): get the top element of the stack.
- Push(*i*): add item *i* onto the top of the stack.
- Pop( ): remove the top of the stack.

Parentheses Check

- Check whether the parentheses are nested correctly.

7 - ((X * ((X + Y) / (J – 3)) + Y) / (4 - 5))

**

((A + B)

) A + B (

檢查方法:

**

Checking Various Parentheses

- Check whether one left parenthesis matches the corresponding right parenthesis

{x + (y - [a+b]) * c – [(d+e)]} / (h-j)

**

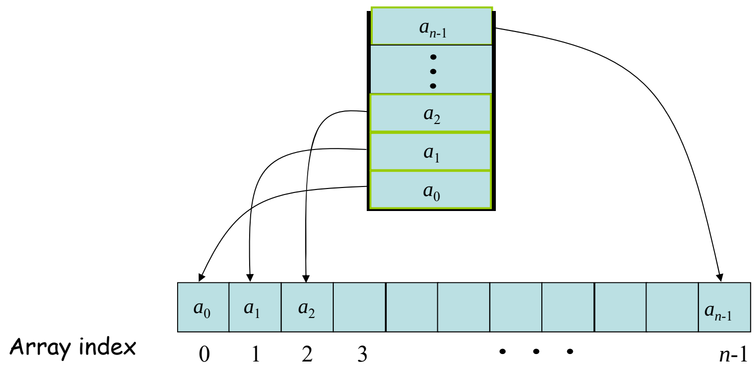
檢查方法:

**

Abstract Data Type Stack

```
template <class T>
class Stack
{ // A finite ordered list with zero or more elements.
  public:
   Stack (int stackCapacity = 10);
   // Create an empty stack whose initial capacity is
  stackCapacity
   bool IsEmpty( ) const;
   // If number of elements in the stack is 0, return true(1) else
  return false(0)
   T& Top( ) const;
    // Return top element of stack.
   void Push(const T& item); // (推入)
    // Insert item into the top of the stack.
   void Pop( ); // (彈出)
   // Delete the top element of the stack.
};
```


Implementation of Stack by Array



Array Implementation of Stack

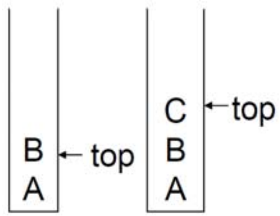
```
private:
    T *stack; // array for stack
    int top;  //  array index of top element
    int capacity; //  size of stack

template <class T>
Stack<T>::Stack (int stackCapacity) :
capacity (stackCapacity)
//  capacity=stackCapacity
{
    stack = new T[capacity];
    top = -1;
}
```


Array Implementation of Stack

```
template <class T>
inline bool Stack<T>::IsEmpty( ) const
{   // inline建議編譯器直接將程式代入呼叫點
 // const：函數內不能修改成員變數之值，僅能參考使用
    return top == -1;
}
template <class T>
inline T& Stack<T>::Top ( ) const
// get the top element
{   if (IsEmpty( )) throw "Stack is empty";
// throw：進行例外(exception)處理，一般會離開程式
    return stack[top];
}
```

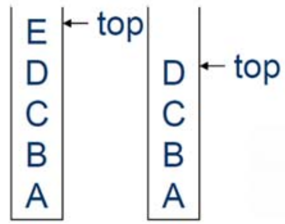
Array Implementation of Stack

```
template <class Type>
void Stack<T>::Push (const T& x)
// push (add) x to the stack
{
    if (top == capacity - 1)
        throw "Stack Overflows";
    stack[++top] = x;
}
```



Array Implementation of Stack

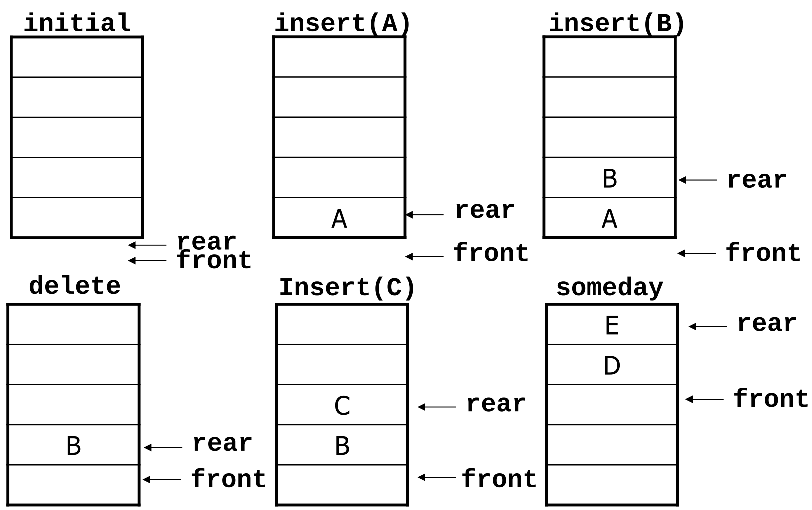
```
template <class T>
void Stack<T>::Pop( )
// Delete top element from the stack
{
   if (IsEmpty())
      throw "Stack is empty. Cannot delete."
   stack[top--].~T( ); // destructor for T
}
```



Queue

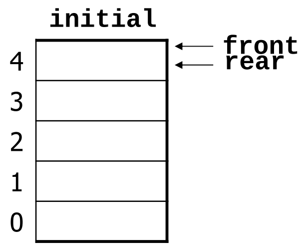
- First-in first-out (FIFO)
- First come first serve (FCFS)
- 2 ends: Data are inserted to one end (rear) and deleted from the other end (front).

Linear Queue

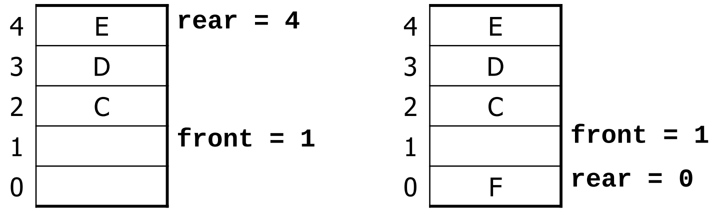


- insert(F): overflow, but there are still some empty locations in the lower part.

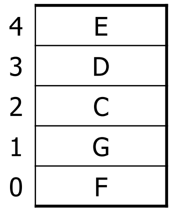
Circular Queue



- Next of the last: 0
- **empty queue 之條件:**
  front = = rear



Full Circular Queue



In this situation, we cannot distinct a full or empty queue.

front = rear = 1

- In a circular queue with n locations, it is full if n-1 locations are used.
- If all of n location are used, we cannot distinct a full queue or an empty queue.

Abstract Data Type Queue

```
template <class T>
class queue
{ // A finite ordered list with zero or more elements
  public:
    queue (int queueCapacity = 10);
  // Create an empty queue whose initial capacity is queueCapacity
    bool IsEmpty() const;
    // Check whether the queue is empty or not
    T& Front() const; // (前面)
    // Return the front element of the queue
    T& Rear() const; // (後面)
    // Return the rea element of the queue
    void Push(const T& item);
    // Insert item at the rear
    void Pop();
    // Delete the front element
};
```


Array Implementation of Queue

```
private:
   T *queue;          // array for queue
   int front;         // front pointer
   int rear;          // rear pointer
   int capacity;      // size of queue

template <class T>
Queue<T>::Queue (int queueCapacity) :
capacity(queueCapacity)
// capacity=queueCapacity
{   queue = new T[capacity];
    front = rear = capacity -1;
                // 0 is also OK
}
```

Array Implementation of Queue

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

template <class T>
inline T& Queue<T>::front()
{
    if (IsEmpty())
        throw "Queue is empty."
    return queue[(front+1)%capacity];
}
```


Array Implementation of Queue

```
template <class T>
inline T& Queue<T>::rear()
{   if (IsEmpty()) throw "Queue is empty."
    return queue[rear];
}
template <class T>
void Queue<T>::Push (const T& x)
// add x to at rear
{   if ((rear + 1) % capacity == front)
        throw "Queue is full."
    rear = (rear + 1) % capacity;
    queue[rear] = x;
}
```


Array Implementation of Queue

```
template <class T>
void Queue<T>::Pop()
// delete the front element
{
    if (IsEmpty())
        throw "Queue is empty."
     front = (front + 1) % capacity;
    queue[front].~T();  // destructor for T
}
```


Class Inheritance in C++

```
class Bag {
public:
   Bag (int bagCapacity = 10);// constructor
   virtual ~Bag();                 // destructor
   virtual int Size() const; // return #elements in the bag
   virtual bool IsEmpty() const; // empty or not
   virtual int Element() const; // return an element
   virtual void Push(const int); // insert an element
   virtual void Pop();     //delete an element
   /* 被繼承時，若子類別的有同名函數，欲執行何者視當時指標
  的實際指向而定 */
   protected:
      int *array;       // array used for storing Bag
      int capacity;     // size of array
      int top;          // array index of top element
};
```

Class Inheritance in C++

```
class Stack : public Bag
// Stack inherits Bag, Stack 繼承 bag
{
public:
    Stack(int stackCapacity = 10);// constructor
    ~Stack();                      // destructor
    int Top() const;
    void Pop();
};

Stack::Stack(int stackCapacity) :
  Bag(stackCapacity) { }
// Constructor for Stack calls constructor for Bag
```

private	: class 內部可用
protected	: 繼承者內部亦可用
public	: 全部可用

Class Inheritance in C++

```
Stack::~Stack() { }
/* Destructor for Bag is automatically called when Stack is destroyed.
  This ensures that array is deleted */
int Stack::Top() const
{
  if (IsEmpty()) throw "Stack is empty.";
  return array[top];
}

void Stack::Pop()
{
  if (IsEmpty())
     throw "Stack is empty. Cannot delete.";
  top--;
}
```


Class Inheritance in C++

```
Bag b(3);  // uses Bag constructor to create array of capacity 3
Stack s(3);// uses Stack constructor to create array of capacity 3

b.Push(1); b.Push(2); b.Push(3);
// use Bag::Push

s.Push(1); s.Push(2); s.Push(3);
// Stack::Push not defined, so use Bag::Push.

b.Pop(x);        // uses Bag::Pop, which calls Bag::IsEmpty
s.Pop(x);
/* uses Stack::Pop, which calls Bag::IsEmtpy because IsEmpty has not
   been redefined in Stack. */
```


Inheritance in C++

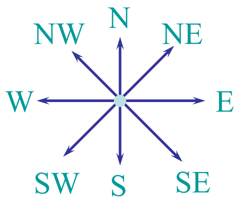
- A derived class (衍生類別, Stack) inherits all the non-private members (data and functions) of the base class (基本類別, Bag).
- Inherited public and protected members from public inheritance have the same level of access in the derived class as they did in the base class.
- The derived class can reuse the implementation of a function in the base class, except constructor and destructor.
- The derived class can implement its own function (override the implementation).

The Mazing Problem (迷宮問題)

Entrance

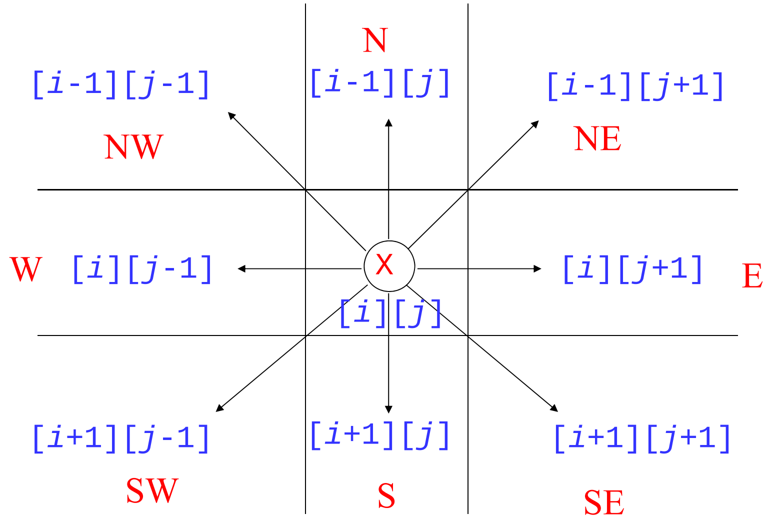
0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

Exit



0:通路 1:障礙物 八方向

Allowable Moves



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
2	1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
3	0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
4	1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
5	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
6	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
7	0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
8	0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
9	1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
10	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
11	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

**

Stack

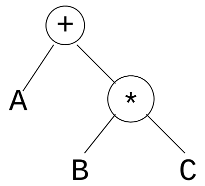
row	col	dir	
			9
			8
			7
			6
			5
			4
1	4	E	3
1	3	E	2
2	2	NE	1
1	1	SE	0

Infix, Postfix, Prefix Expressions

- infix A+B /* A, B: operands, +: operator */
- postfix AB+
  (reverse Polish notation)
- prefix +AB
  (Polish notation)
- Conversion from infix to postfix:

```
e.g.  A + (B*C)     infix     (inorder)
      A + (BC*)
      A(BC*)+
      ABC * +       postfix   (postorder)
```

Expression Tree (1)




```
infix  : A + (B*C)
prefix : + A * B C
postfix: A B C * +
```

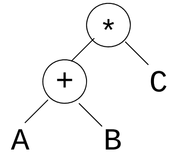
inorder traversal:
1. left subtree
2. root
3. right subtree

preorder traversal:
1. root
2. left subtree
3. right subtree

postorder traversal:
1. left subtree
2. right subtree
3. root

Expression Tree (2)

```
e.g. (A+B) * C    infix
     (AB+) * C
     (AB+)C *
     AB+C *       postfix
```

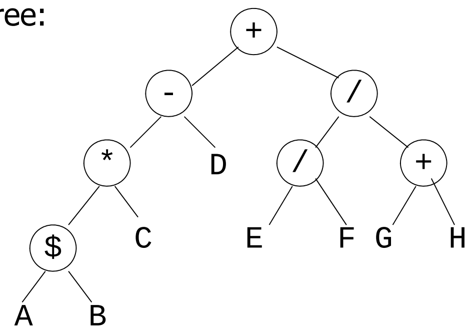



```
infix  : (A+B) * C
prefix : * + A B C
postfix: A B + C *
```


Expression Tree (3)

```
e.g.  infix   A$B*C-D+E/F/(G+H)
                   /* $: exponentation, 次方 */
```

expression tree:



postfix :

prefix :

**

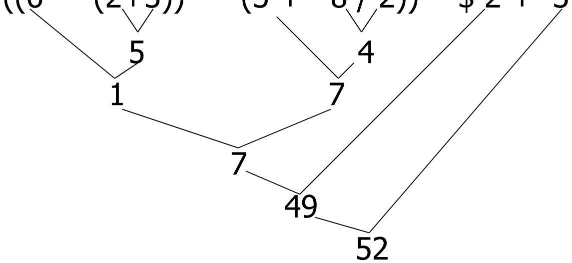
Evaluating a Postfix Expression

- precedence: (,) > $ > *, / > +, -
- left associative: A+B+C = (A+B)+C
- right associative: ABC = A$(B$C)

```
e.g. 6 2 3 + - 3 8 2 / + * 2 $ 3 +
```

infix form:

((6 - (2+3)) * (3 + 8 / 2)) $ 2 + 3 = 52



Evaluation with a Stack

symb	opnd1	opnd2	value	stack
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
$	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Postfix Evaluation with a Stack

Algorithm (演算法)：

**

How to Verify a Postfix Expression?

方法一：

**

方法二：

Pseudo Code for Postfix Evaluation

```
void Eval(Expression e)
{/* A token in e is either an operator,
operand, or '#'. The last token is '#'.
NextToken( ) gets the next token from e. */
   Stack<Token>stack; // initialize stack
   for (Token x = NextToken(e); x!= '#';
          x=NextToken(e))
     if (x is an operand)
        stack.Push(x)  // add to stack
     else {// operator
        remove correct number of operands
           for operator x from stack;
        perform the operation x and store
           result onto the stack; } }
```

Conversion from Infix to Postfix

e.g. A+B*C ⟶ ABC*+

	symb	postfix string	stack
1	A	A	
2	+	A	+
3	B	AB	+
4	*	AB	+ *
5	C	ABC	+ *
6		ABC *	+
7		ABC * +	

e.g. A*B+C ⟶ AB*C+

e.g. ((A-(B+C))*D)$(E+F) ⟶ ABC+-D*EF+$

symb	postfix string	stack
(		(
(		((
A	A	((
-	A	((-
(	A	((-(
B	AB	((-(
+	AB	((-(+
C	ABC	((-(+
)	ABC+	((-
)	ABC+-	(
*	ABC+-	(*
D	ABC+-D	(*
)	ABC+-D*	
$	ABC+-D*	$
(	ABC+-D*	$(
E	ABC+-D*E	$(
+	ABC+-D*E	$(+
F	ABC+-D*EF	$(+
)	ABC+-D*F+	$
	ABC+-D*EF+$	

Algorithm for Conversion

1)遇 operand, 直接 output

2)遇 operator
   (a) 若此 operator 之 precedence 比 top of stack 高 ==> 此 operator 放入 stack.
   (b) 否則, 將所有比此 operator 之precedence 還高之 operator 全 pop 並 output, 再將比 operator 放入 stack.

3) **

4)

5)

Conversion from Infix to Postfix(1)

```
void Postfix(Expression e)
{/* Output postfix form of infix
expression e. NextToken is as in function
Eval. 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 (;stack.Top( ) != ‘(';
              stack.Pop( ))
             cout << stack.Top( );
         stack.Pop( ); // unstack ‘(' }
```

Conversion from Infix to Postfix(2)

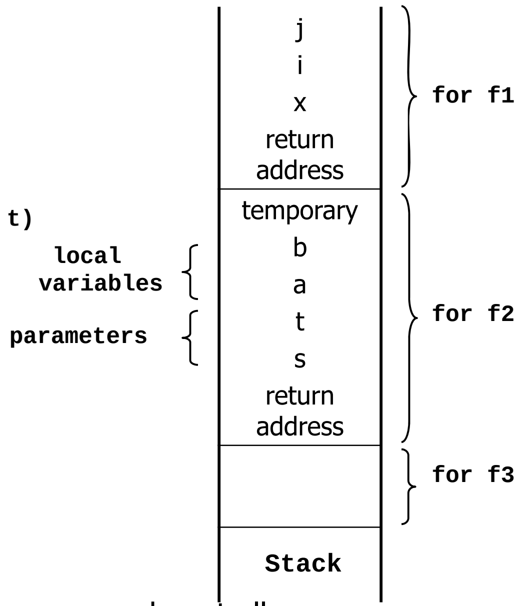
```
        else { // x is an operator
           for (; isp(stack.Top( )) <= icp(x);
                 stack.Pop( ))
               // isp: in-stack priority
               // icp: in-coming priority
               cout << stack.Top( );
            stack.Push(x);  }
      // end of expression; empty the stack
     for (; !stack.IsEmpty( ); cout <<
           stack.Top( ), stack.Pop( ));
        cout << endl;
}
```

註：愈優先(precedence高)，isp或icp之值愈小。
設定 isp('(')=8, icp('(')=0, isp('#')=8

Storage Allocation for a C Compiler

- dynamic allocation:
  - storage for local variables, parameters are allocated when a function is called.
- A function call is maintained by using a stack


```
e.g.
  int f1(int x)
  {
    int i, j;
       :
  }
  int f2(float s,float t)
  {
    char a, b;
       :
    f1(4)
       :
  }
  int f3()
  {
       :
    f2(2.4, 7.5);
       :
  }
```



- Some programming languages do not allow recursive programs, e.g. FORTRAN, COBAL.

Data Structures

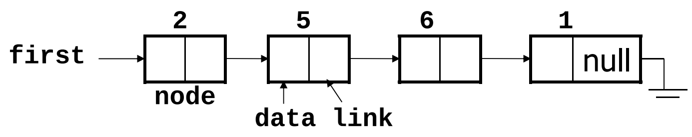
Chapter 4: Linked Lists

Singly Linked List

0	1	2	3	4	5	6	7	8	
	D	A			B	C			data
	-1	5			6	1			link(pointer, address)

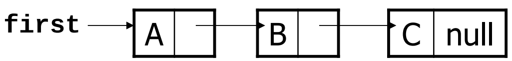
以-1代表結尾 (null pointer)

first = 2 header, external pointer

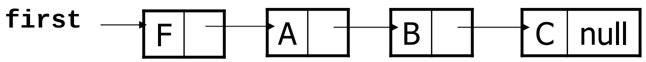


**

Operation on the First Node

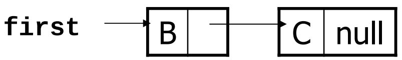


(1) adding 'F' to the front of the list



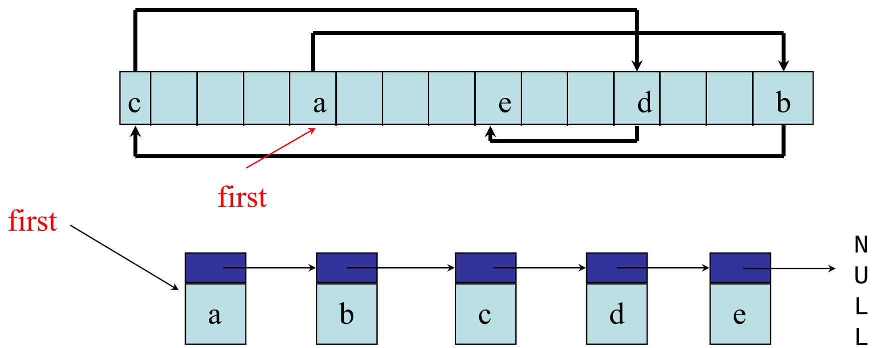

```
p = getnode();   // create a new node
data(p) = 'F';
link(p) = first;
first = p;
```

(2) removing the first node of the list



**

Normal Way to Draw a Linked List

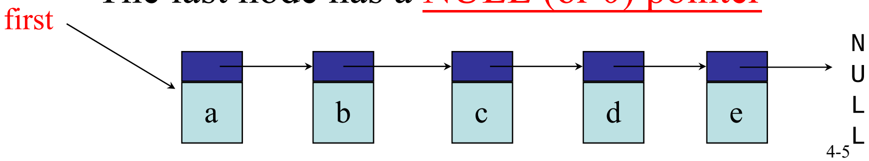


link field (鏈結欄位) of a node, points to next node

data field (資料欄位) of a node

Chain/Singly Linked List

- A chain (鏈), also called a singly linked list (單一鏈結串列), is a linked list in which each node (節點) represents one element.
- Each node has exactly one link or pointer from one element to the next.
- The last node has a NULL (or 0) pointer



Node Representation

```
// in C++
class ChainNode
{
   char data;
   ChainNode *link;
}
```

```
// in C
struct ChainNode
{
   char data;
   struct ChainNode *link;
}
```

Linked List Represented by C++

```
class ChainNode {
friend class Chain;
public:
  ChainNode (int element = 0, ChainNode
*next =0)
// 0: default value for element and next
  {  data = element;
     link = next;
  }
private:
  int data;
  ChainNode *link;
};
```

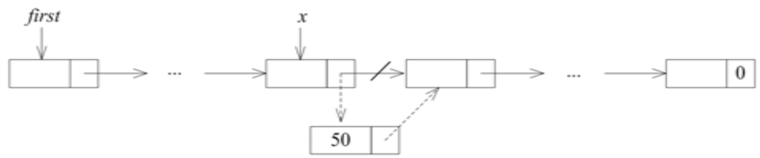

Construction of a 2-Node List

```
void Chain::Create2( )
{
// create and set fields of second node
  ChainNode* second = new ChainNode(20,0)
// create and set fields of first node
  first = new ChainNode(10,second);
// "first" is a data member of class Chain
}
```

**

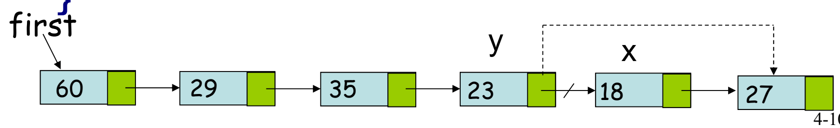
Inserting a Node after Node x

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



Deleting Node x

```
void Chain::Delete(ChainNode* x,
   ChainNode *y)
// x is after y
{
  if(x = = first)
    first = first->link;
  else
    $\boxed{\phantom{xxxxxxxxxxxxxxxxxx}}$          **
  delete x;
}
```



Template Class

- A chain (singly linked list) is a container class (容器類別), and is therefore a good candidate for implementation with templates.
- Member functions of a linked list should be general so that they can be applied to all types of objects.
- The template mechanism can be used to make a container class more reusable.

Template Definition of Class Chain

```
template < class T > class Chain;
   // forward declaration
template < class T > class ChainNode {
friend class Chain <T>;
private:
  T data;
  ChainNode<T>* link;
};
template <class T> class Chain {
public:
  Chain( ){first = 0;}//constructor, set 0
  // Chain manipulation operation
private:  ChainNode<T> * first, *last; }
```

Inserting at the Tail of a List

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


Concatenation of Two Lists

```
template < class T >
void Chain <T>::Concatenate(Chain<T>& b)
{ // b is concatenated to the end of *this.
// this = (a1, …, am),b = (b1, …, bn),
// new chain this = (a1, …, am, b1, …,bn)
  if ( b.first == 0 ) return // empty b
  if ( first ) { // nonempty a
    last->link = b.first;
    last = b.last;
  }
  else { first = b.first; last = b.last; }
  b.first = b.last = 0;
}
```

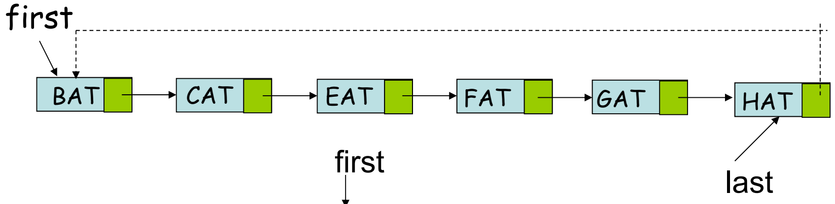

Reversing a List

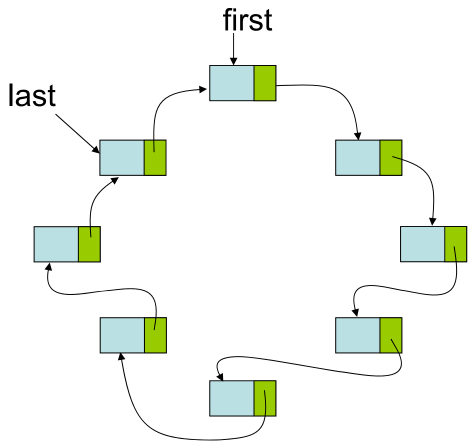
```
template <class T>
void Chain<T>::Reverse( )
{// (a1,…,an) becomes (an,…,a1)
  ChainNode<T> *current = first,
          *previous = 0; // before current
  while (current) {
    ChainNode<T> *r = previous;
    previous = current;
    current = current->link;
             // moves to next node
    previous->link = r; // reverse the link
  }
  first = previous; }
```


When Not to Reuse a Class

- Reusing a class is sometimes less efficient than direct implementation.
- If the operations required by the application are complex and specialized, and therefore not offered by the class, then reusing becomes difficult.

Circular List

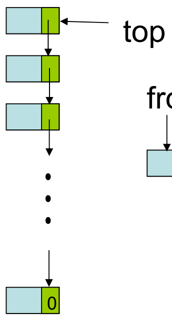




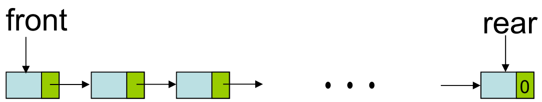
Inserting at the Front of a Circular List

```
template <class T>
void CircularList <T>::InsertFront(const T&
e) // Insert e at the front of *this
{  //  “last” points to the last node
  ChainNode <T> *newNode = new ChainNode
<T>(e);
  if (last) { // nonempty list
    newNode->link = last->link;
    last->link = newNode;   }
  else { // empty list
    last = newNode;
    newNode->link = newNode; }
}
```


Linked Stacks and Queues



Linked Stack



Linked Queue

Push and Pop of a Linked Stack

```
template <class T>
void LinkedStack <T>::Push(const T& e)
{  // Adding an element to a linked stack
     top = new ChainNode <T>(e, top);
}
template <class T> void LinkStack <T>::Pop(
) {  // Delete top node from the stack
  if (IsEmpty( ))
    throw “Stack is empty. Cannot delete.”;
  ChainNode <T> *delNode = top;
  top = top->link;  // remove top node
  delete delNode;   // free the node
}
```

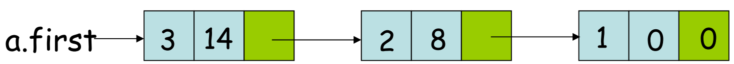
Push (Inserting Rear) of a Linked Queue

```
template <class T>
void LinkedQueue <T>:: Push(const T& e)
{  // Insertion at the rear of queue
  if (IsEmpty( )) // empty queue
    front = rear = new ChainNode(e,0);
  else // attach node and update rear
    rear = rear->link = new ChainNode(e,0);
}
```

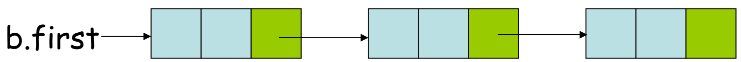

Pop (Deleting Front) of a Linked Queue

```
template <class T>
void LinkedQueue <T>:: Pop()
{// Delete first (front) element in queue
  if (IsEmpty())
    throw “Queue is empty. Cannot delete.”;
  ChainNode<T> *delNode = front;
  front = front->link;// remove first node
  // should be corrected by adding
  // if (front == 0) rear = 0;
  delete delNode;     // free the node
}
```


Revisit Polynomials



$$a = 3x^{14} + 2x^{8} + 1$$



**

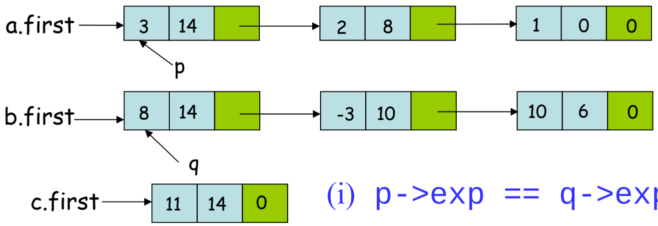
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

Definition of Class Polynomial

```
struct Term
{// All members in “struct” are public
  int coef;  // coefficient
  int exp;   // exponent
  Term Set(int c,int e) {coef = c; exp = e;
return *this;};
};
class Polynomial {
  public: // public function defined here
  private:
    Chain<Term> poly;
};
```

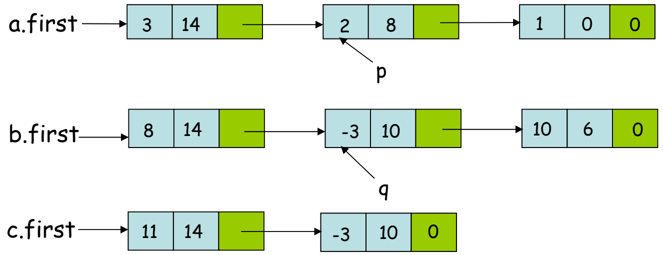
Addition of Two Polynomials (1)

- It is an easy way to represent a polynomial by a linked list.
- Example of adding two polynomials a and b



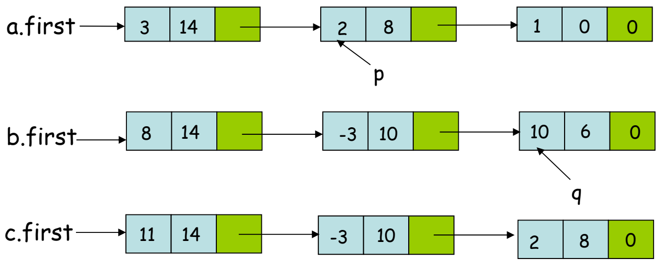
(i) `p->exp == q->exp`

Addition of Two Polynomials (2)



(ii) `p->exp < q->exp`

Addition of Two Polynomials (3)



(iii) `p->exp > q->exp`

Properties of Relations

- For any polygon (多邊形) x, $x \equiv x$. Thus, $\equiv$ is reflexive (反身的，自反的).
- For any two polygons x and y, if $x \equiv y$, then $y \equiv x$. Thus, the relation $\equiv$ is symmetric (對稱的).
- For any three polygons x, y, and z, if $x \equiv y$ and $y \equiv z$, then $x \equiv z$. The relation $\equiv$ is transitive (遞移的).

Equivalence Class (等價類)

- Definition: A relation ≡ over a set S, is said to be an equivalence relation over S *iff* it is symmetric, reflexive, and transitive over *S*.
- Example: Suppose, there are 12 polygons 0 ≡ 4, 3 ≡ 1, 6 ≡ 10, 8 ≡ 9, 7 ≡ 4, 6 ≡ 8, 3 ≡ 5, 2 ≡ 11, and 11 ≡ 0. Then they can be partitioned into three equivalence classes:

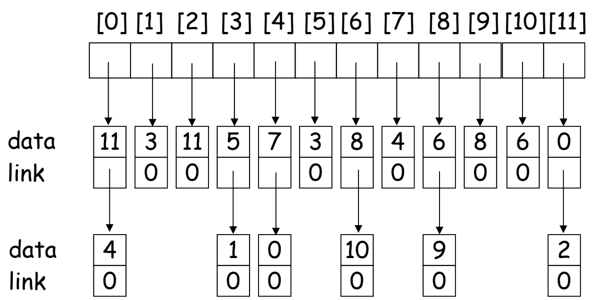
  {0, 2, 4, 7, 11}; {1, 3, 5}; {6, 8, 9, 10}

Pseudo Code for Equivalence Algorithm

```
void Equivalence( ) {
  read n ; // read in number of objects
  initialize first[0:n-1] to 0 and out[0:n-1] to false ;
  while more pairs // input pairs    {
    read the next pair (i, j) ;
    put j on the chain first[i] ;
    put i on the chain first[j] ;     } // example in next page
  for (i = 0; i < n; i++)
    if (!out[i]) {
      out[i] = true ;
      output the equivalence class that contains object i ;
    }
}
```


Linked List Representation

Input: $0 \equiv 4,\ 3 \equiv 1,\ 6 \equiv 10,\ 8 \equiv 9,\ 7 \equiv 4,$
$6 \equiv 8,\ 3 \equiv 5,\ 2 \equiv 11,\ 11 \equiv 0$



**

Summary of Equivalence Algorithm

- Two phases to determine equivalence class
  - Phase 1: Equivalence pairs (*i*, *j*) are read in and adjacency (linked) list of each object is built.
  - Phase 2: Trace (output) the equivalence class containing object *i* with stack (depth-first search). Next find another object not yet output, and repeat.
- Time complexity: $\Theta(m+n)$
  - *n*: # of objects
  - *m*: # of pairs (relations)

Node Structure for Sparse Matrix

next	
down	right

Header node

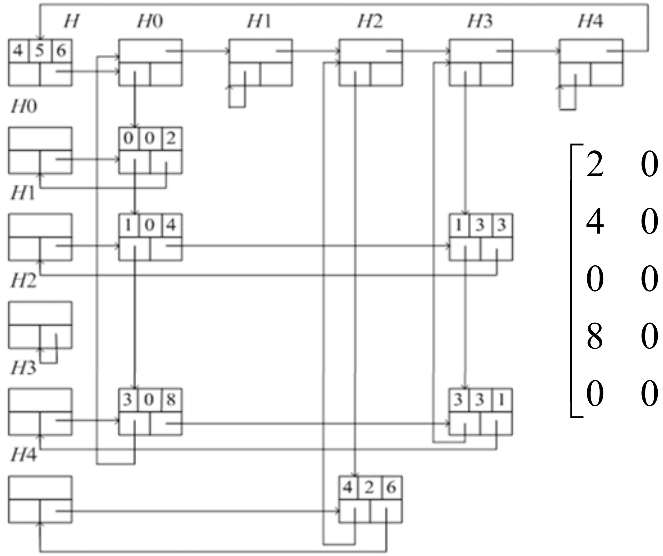
row	col	value
down	right	

Element node

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

A 5×4 sparse matrix

Linked List for Sparse Matrix



$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

Class Definition of Sparse Matrix (1)

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


Class Definition of Sparse Matrix (2)

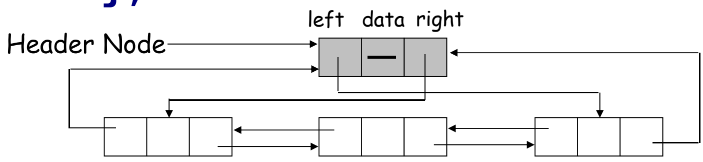
```
MatrixNode::MatrixNode(bool b, Triple *t)
                     // constructor
{   head = b;
    if (b) {right = down = this;}
                   // row/column header node
    else triple = *t; /* element node or header
node for list of header nodes */
}

class Matrix{
  friend istream& operator>>(istream&, Matrix&);
  public:
    ~Matrix(); // destructor
  private:      MatrixNode *headnode;
};
```

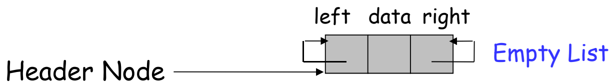
Doubly Linked List

- In a singly linked list, if we want to delete a node *ptr*, we have to know the preceding (前面的) node of *ptr*. Then we have to start from the beginning of the list and to search until the node whose next (link) is *ptr* is found.
- To efficiently delete a node, we need to know its preceding node. Therefore, a doubly linked list is useful.
- A node in a doubly linked list has at least three fields: left, data, right.
- A header node may be used in a doubly linked list.

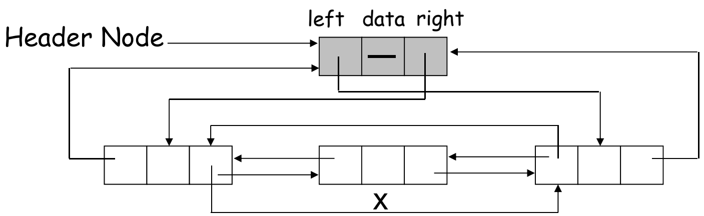
Doubly Linked List

```
class DblListNode {
   int data;
   DblListNode *left, *right;
   };
```



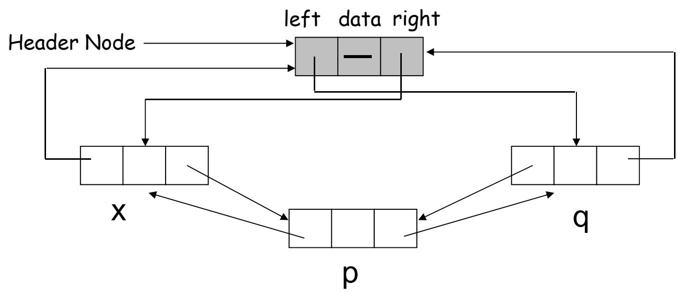


Deleting Node x in a Doubly Linked Circular List




```
x -> left -> right = x -> right;
x -> right -> left = x -> left;
```


Inserting Node p to the Right of Node x in a Doubly Linked Circular List




```
q = x -> right;
p -> left = x;
p -> right = q;
q -> left = p;
x -> right = p;
```

Generalized Lists

- A generalized list, A, is a finite sequence of $n \geq 0$ elements, $a_0, a_1, a_2, \ldots, a_{n-1}$, where a_i, is either an atom or a list. The elements $a_i, 0 \leq i \leq n-1$, that are not atoms are said to be the sublists of A.
- A list A is written as $A = (a_0, \ldots, a_{n-1})$, and the length of the list is n.
- A list name is represented by a capital letter and an atom is represented by a lowercase letter.
- a_0 is the head of list A and the rest $(a_1, a_2, \ldots, a_{n-1})$ is the tail of list A.

Examples of Generalized Lists

- $A = (\ )$: the null, or empty, list; its length is zero.
- $B = (a, (b, c))$: a list of length two; its first element is the atom a, and its second element is the linear list (b, c).
- $C = (B, B, (\ ))$: A list of length three whose first two elements are the list B, and the third element is the null list.
- $D = (a, D)$: is a recursive list of length two; D corresponds to the infinite list $D = (a, (a, (a, \ldots)))$.
- head(B) = 'a' and tail(B) = (b, c), head(tail(C))=B and tail(tail(C)) = ( ).
- Lists may be shared by other lists.
- Lists may be recursive.

General Polynomial

$$p(x,y,z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$$

- $P(x, y, z)$=

$$((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 + 6x^3)y^4 + 2y)z$$

- Rewritten as $Cz^2 + Dz$, where C and D are polynomials.
- Again, in C, it is of the form $Ey^3 + Fy^2$, where E and F are polynomials.
- In general, every polynomial consists of a variable plus coefficient-exponent pairs. Each coefficient may be a constant or a polynomial.

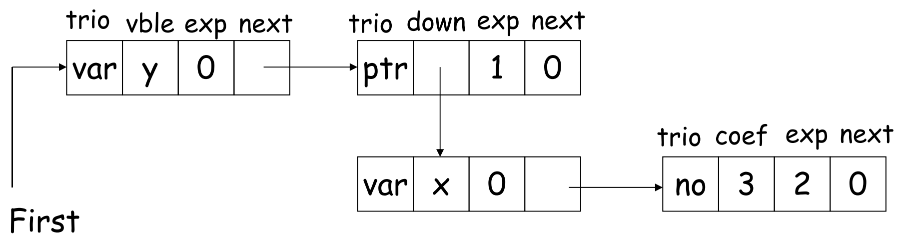
PolyNode Class in C++

```
enum Triple{ var, ptr, no };
class PolyNode
{
  PolyNode *next;  // link
  int exp;
  Triple trio; //explanation in next page
  union {
     char vble;
     PolyNode *down; // link
     int coef;
  };
};
```

3 Types of Nodes in PolyNode

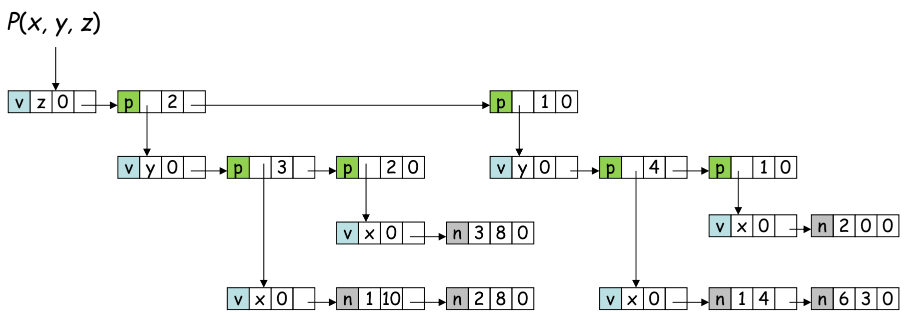
- *trio == var*
  - The node is a header node.
  - *vble* indicates the name of the variable
  - *exp* is set to 0.
- *trio == ptr*
  - Coefficient is a list pointed by *down*.
  - *exp* is the exponent of the variable on which that list is based on.
- *trio == no*
  - Coefficient is an integer, stored in *coef*.
  - *exp* is the exponent of the variable on which that list is based on.

Representation of $3x^2y$



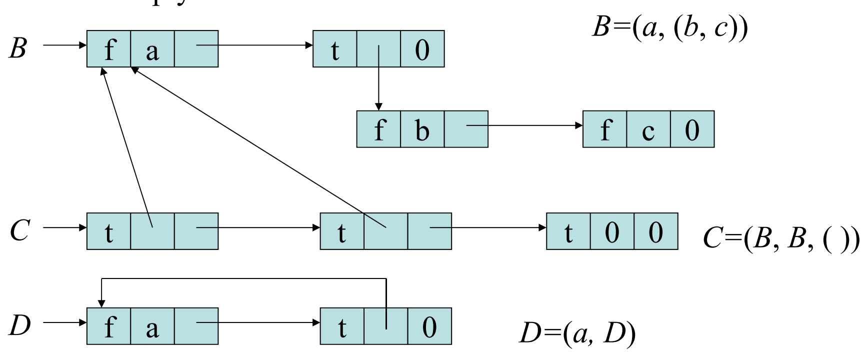
Representation of $P(x, y, z)$

$$((x^{10}+2x^8)y^3+3x^8y^2)z^2+((x^4+6x^3)y^4+2y)z$$



Representations of Generalized Lists

$A = 0$ Empty list

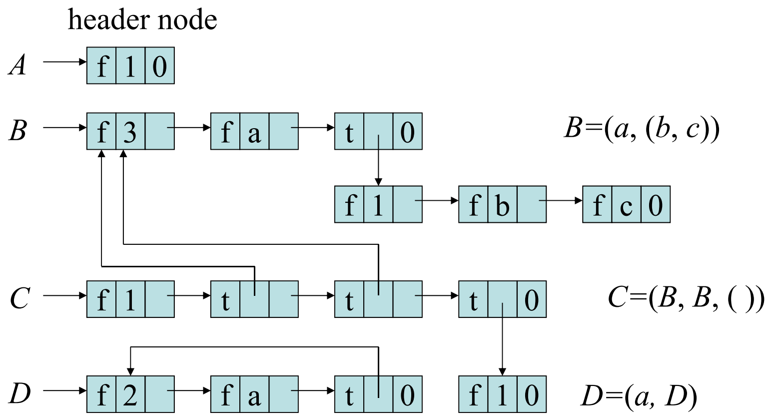


**

Reference Counts in Shared Lists

- Lists may be shared by other lists for storage saving.
- Add a header node to store the reference count, which is the number of pointers to it.
- The reference count can be dynamically updated. The list can be deleted only when the reference count is 0.

Reference Counts in Shared Lists



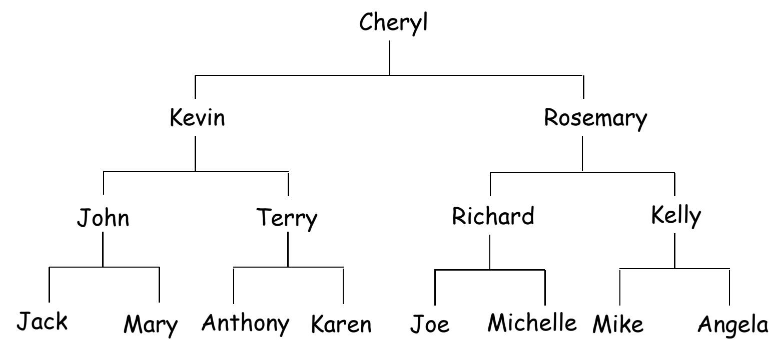
reference count in header node: # of pointers to the list

Data Structures

Chapter 5: Trees

Pedigree Genealogical Chart

血統、家譜　　宗譜的、家系的

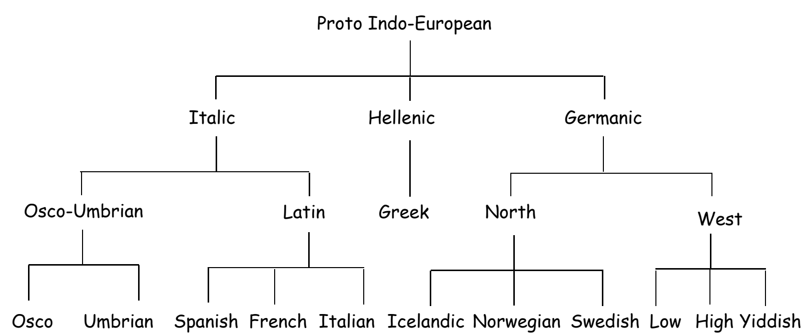


Ancestors of Cheryl: A **binary tree**

John and Terry are parents of Kevin.

Lineal Genealogical Chart

直系的、世襲的　宗譜的、家系的

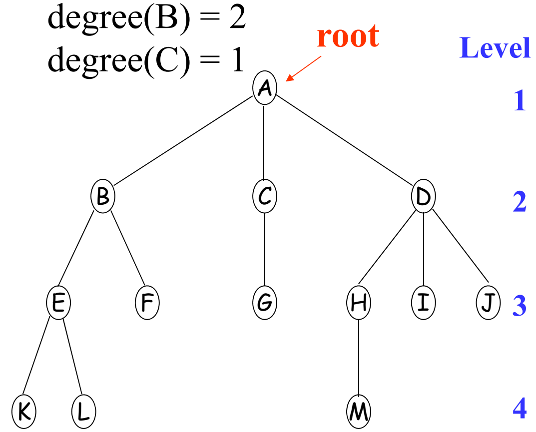


Modern European languages

Latin produces Spanish, French and Italian.

A Tree

degree(A) = 3
degree(B) = 2
degree(C) = 1



height = depth = 4
node: A, B, C, …M
parent of E: B
children of B: E, F
ancestors of E: A, B
descendants of B: E, F, K, L
leaf nodes (terminal nodes): K, L, F, G, M, I, J

nonleaf nodes (nonterminal nodes):

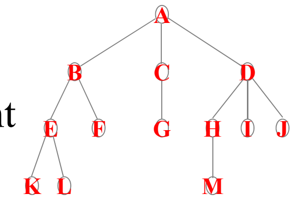
**

Trees

- Definition: A tree is a finite set of one or more nodes such that:
  - There is a specially designated node called the root.
  - The remaining nodes are partitioned into $n \geq 0$ disjoint (無交集的) sets $T_1, \ldots, T_n$, each of which is a subtree.

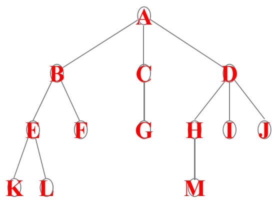
Tree Terminologies (1)

- degree (分支度) of a node:
  number of subtrees of the node.
- degree of a tree:
  maximum degree of the nodes in the tree.
- leaf (terminal) node:
  a node with degree zero
- Siblings (brothers):
  the nodes with the same parent



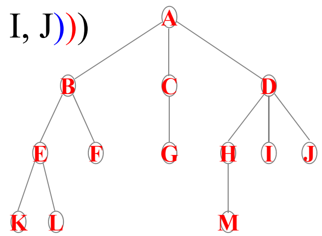
Tree Terminologies (2)

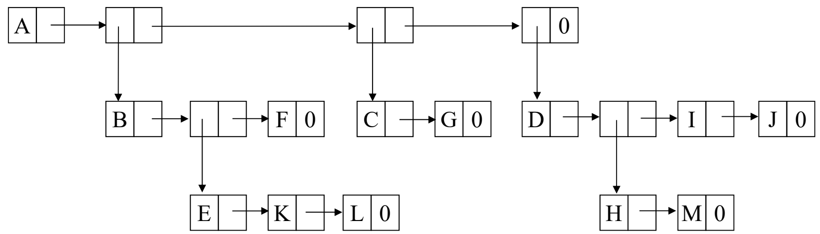
- ancestors of a node:
  all the nodes along the path from the root to the node.
- descendants of a node:
  all the nodes of its subtrees.
- level of a node:
  the level of the node's parent plus one.
  Here, the level of the root is 1.
- height (depth) of a tree:
  the maximum level of the nodes in the tree.



List Representation of a Tree

The tree is represented as a list:
(A (B (E (K, L), F), C(G), D(H (M), I, J)))

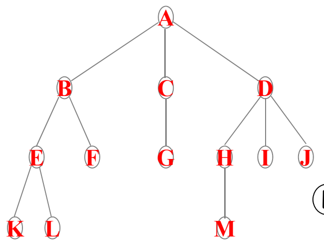


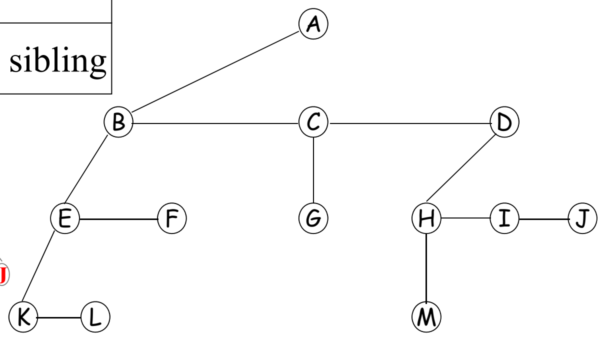


Representation of Trees

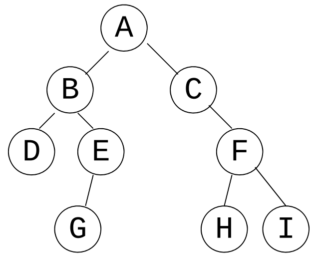
- Left child-right sibling tree
  - two links (or pointers): left child and right sibling

data	
left child	right sibling

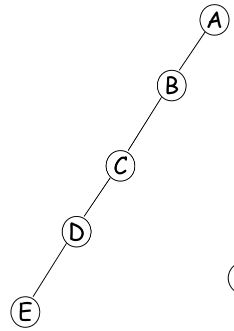




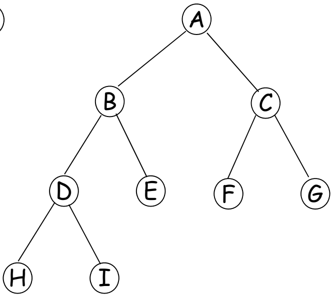
Binary Trees



Binary tree



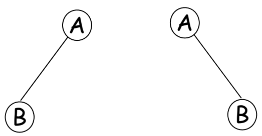
Skewed binary tree
歪斜二元樹



Complete binary tree
完整二元樹

Binary Tree 二元樹

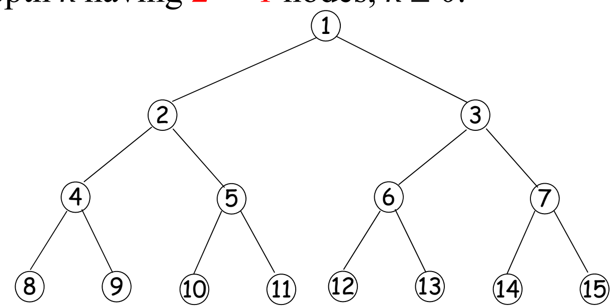
- A binary tree:
  - a finite set of nodes that is either empty, or consists of a root and two disjoint binary trees called the left subtree and the right subtree.
- In a binary tree, we distinguish between the order of the children; in a tree we do not.



Two different binary trees

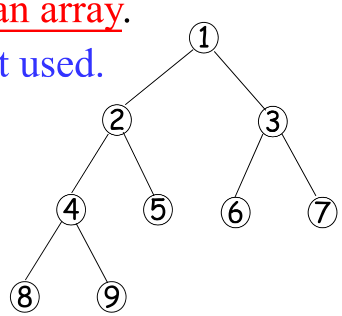
Full Binary Tree

- The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.
- The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.
- A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.



Complete Binary Tree

- A complete binary tree with n nodes and depth k is that its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.
- It can be represented by an array.
- Root is at a[1]. a[0] is not used.



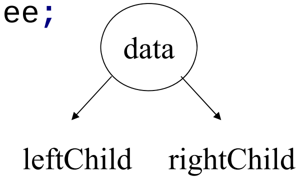
parent(i)= $\left\lfloor \frac{i}{2} \right\rfloor$

** left_child(i)=

right_child(i)=

Linked Representation of Binary Trees

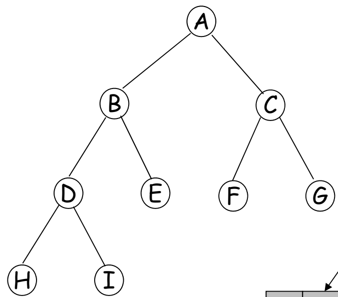
```
template <class T> class Tree;
   //forward declaration
template <class T>
class TreeNode {
friend class Tree <T>;
private:
  T data;
  TreeNode <T> *leftChild;
  TreeNode <T> *rightChild;
};
```

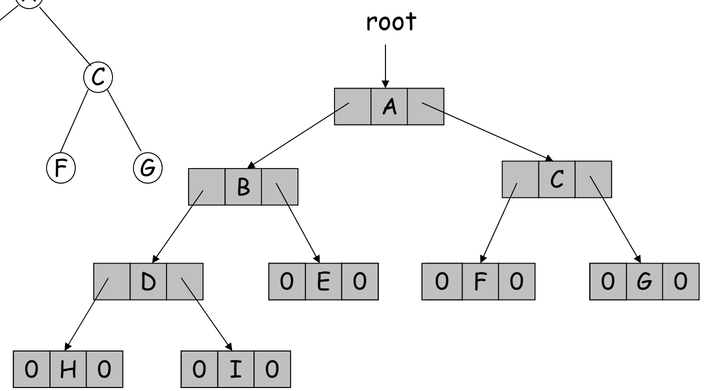


Linked Representation of Binary Trees

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


Linked Representation of a Binary Tree





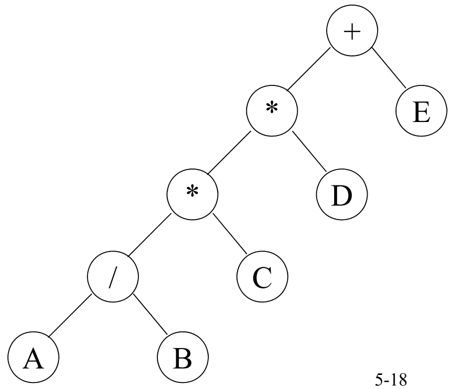
Binary Tree Traversal

- Preorder traversal:
  1. root
  2. left subtree
  3. right subtree
- Inorder traversal:
  1. left subtree
  2. root
  3. right subtree
- Postorder traversal:
  1. left subtree
  2. right subtree
  3. root

Arithmetic Expression Trees

- Preorder Traversal:
  => Prefix expression
- Inorder Traversal:
  => Infix expression
- Postorder Traversal:
  => Postfix expression

**



Preorder Traversal

```
template <class T>
void Tree <T>::Preorder()
{//Driver calls workhorse for traversal of entire tree
  Preorder(root);
}

template <class T>
void Tree <T>::Preorder (TreeNode <T>
  *currentNode)
{//Workhouse traverses the subtree rooted at currentNode
  If (currentNode){
    Visit(currentNode); //visit root
    Preorder(currentNode->leftChild);
    Preorder(currentNode->rightChild);
  }
}
```


Inorder Traversal

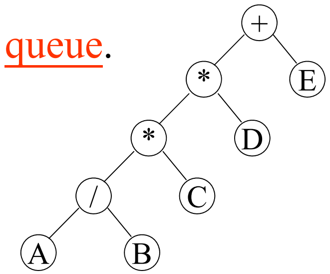
```
template <class T>
void Tree <T>::Inorder()
{//Driver calls workhorse for traversal of entire tree
  Inorder(root);
}

template <class T>
void Tree <T>::Inorder (TreeNode <T>
  *currentNode)
{//Workhouse traverses the subtree rooted at currentNode
  If (currentNode){
    Inorder(currentNode->leftChild);
    Visit(currentNode); //visit root
    Inorder(currentNode->rightChild);
  }
}
```

Postorder Traversal

```
template <class T>
void Tree <T>::Postorder()
{//Driver calls workhorse for traversal of entire tree
  Postorder(root);
}

template <class T>
void Tree <T>::Postorder (TreeNode <T>
  *currentNode)
{//Workhouse traverses the subtree rooted at currentNode
  If (currentNode){
    Postorder(currentNode->leftChild);
    Postorder(currentNode->rightChild);
    Visit(currentNode);  //visit root
  }
}
```


Level-Order Traversal

- Preorder, inorder and postorder traversals all require a stack.
- Level-order traversal uses a queue.
- Level-order traversal:
  1. root
  2. left child
  3. right child.



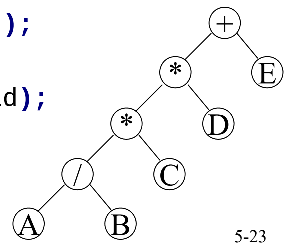
- After all nodes on a level have been visited, we can move down.

Level-order traversal: **

Level-Order Traversal of a Binary Tree

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

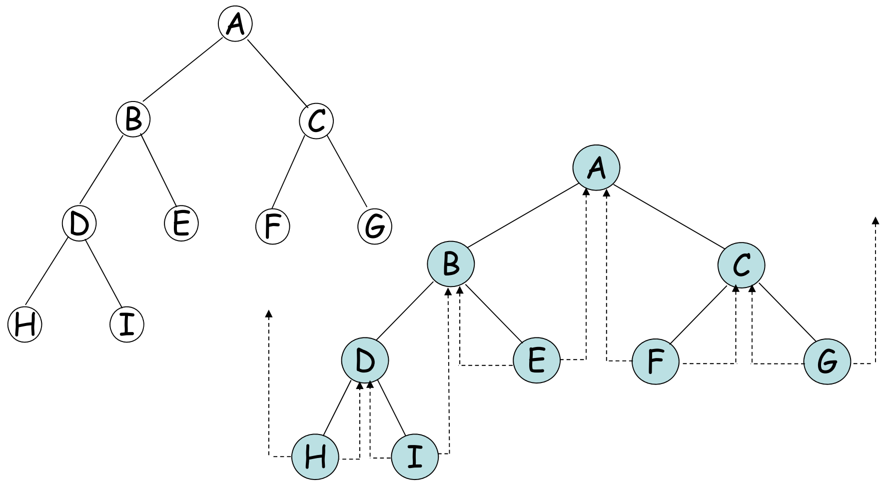
+*E*D/CAB



Threaded Binary Tree 引線二元樹

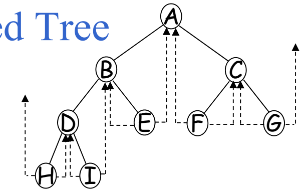
- Threading Rules
  - A 0 *rightChild* field at node *p* is replaced by a pointer to the node that would be visited after *p* when traversing the tree in inorder. That is, it is replaced by the inorder successor of *p*.
  - A 0 *leftChild* link at node *p* is replaced by a pointer to the node that immediately precedes node *p* in inorder (i.e., it is replaced by the inorder predecessor of *p*).

Threaded Binary Tree



Inorder sequence: HDIBEAFCG

Memory Representation of Threaded Tree



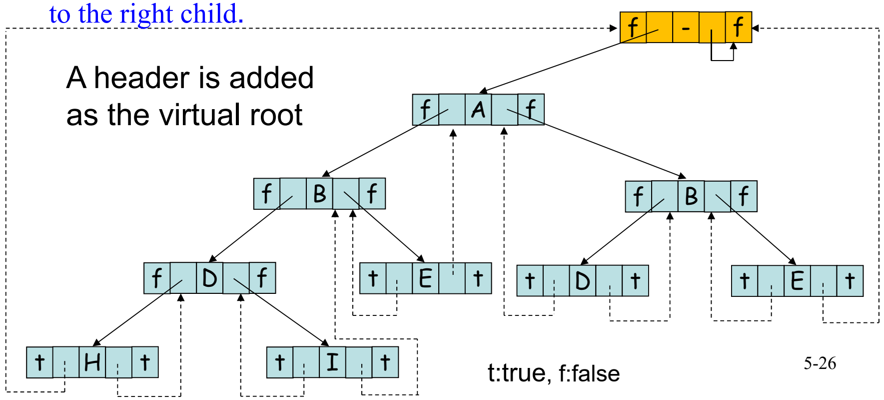
`x->rightThread == TRUE`

=> `x->rightChild` is a **thread**
(pointer to **inorder successor**)

`x->rightThread == FALSE`

=> `x->rightChild` is a **pointer** to the right child.

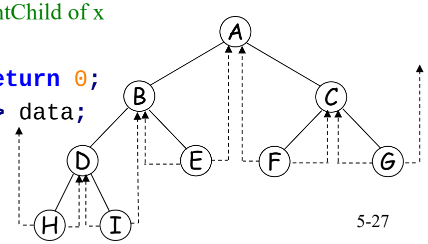
Inorder sequence: H D I B E A F C G



Inorder Successor in Threaded Trees

- By using the threads, we can perform an inorder traversal without making use of a stack.

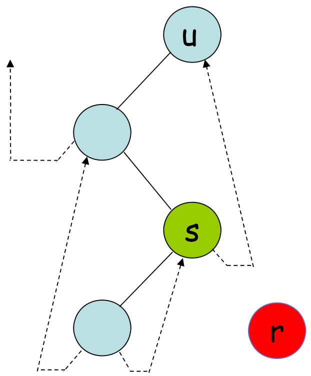
```
T* ThreadedInorderIterator::Next()
{//Return the inorder successor of currentNode=x
   ThreadedNode <T> *temp = currentNode -> rightChild;
   if (!currentNode->rightThread) //real rightChild
       while (!temp->leftThread) temp = temp -> leftChild;
   //a path of leftChild starting from rightChild of x
   currentNode = temp;
   if (currentNode == root) return 0;
   else return &currentNode -> data;
}
```



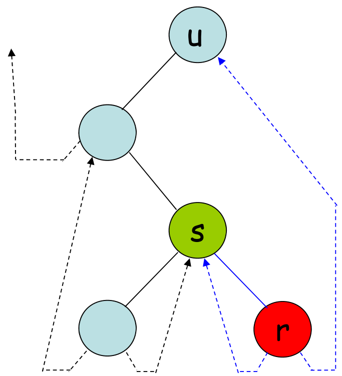
Inorder sequence:
H D I B E A F C G

Insertion of *r* as the Right Child of *s* in a Threaded Binary Tree (1)

Case 1: The right subtree of *s* is empty.



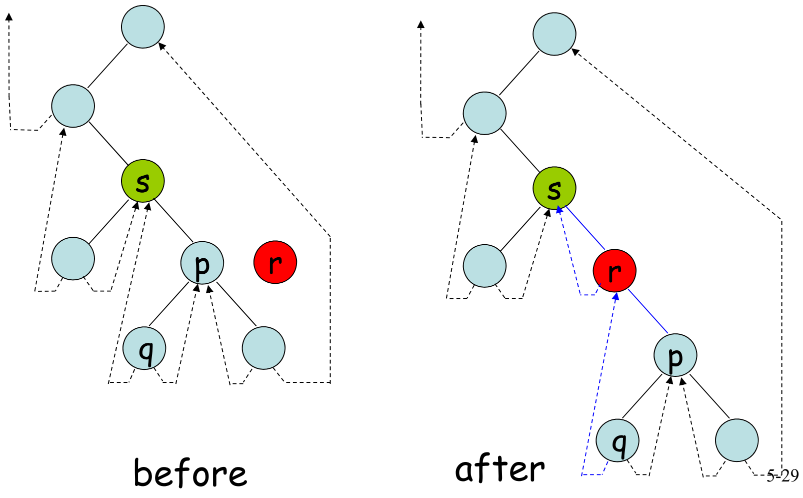
before



after

Insertion of *r* as the Right Child of *s* in a Threaded Binary Tree (2)

Case 2: The right subtree of *s* is not empty.

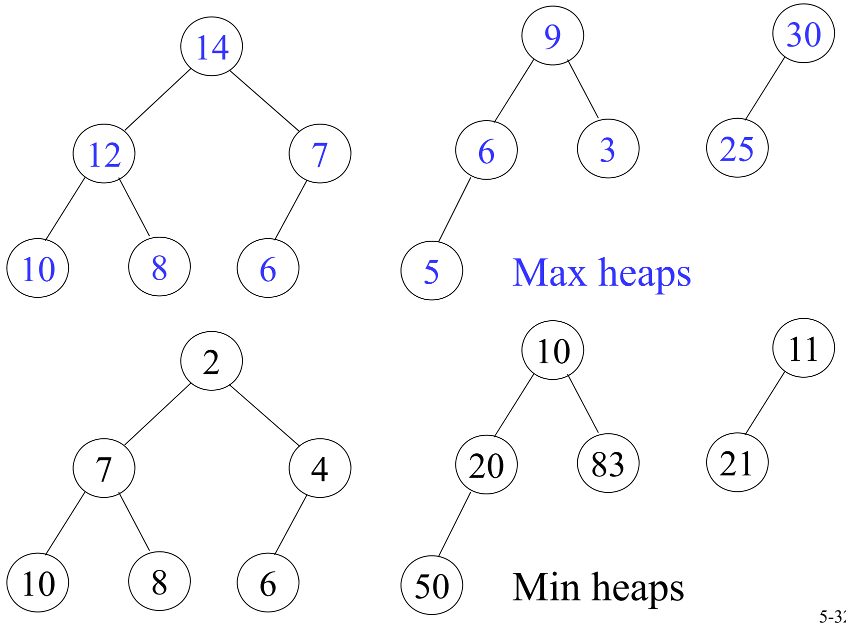


Inserting *r* as the Right Child of *s*

```
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;
  r -> rightThread = s -> rightThread;
  r -> leftChild = s;
  r -> leftThread = True; // leftChild  is a thread
  s -> rightChild = r;
  s -> rightThread = false;
  if (! r -> rightThread) {// rightChild is not a thread
    ThreadedNode <T> *q = InorderSucc (r);
       // return the inorder successor of r
    q -> leftChild = r;
  }
}
```


Priority Queue

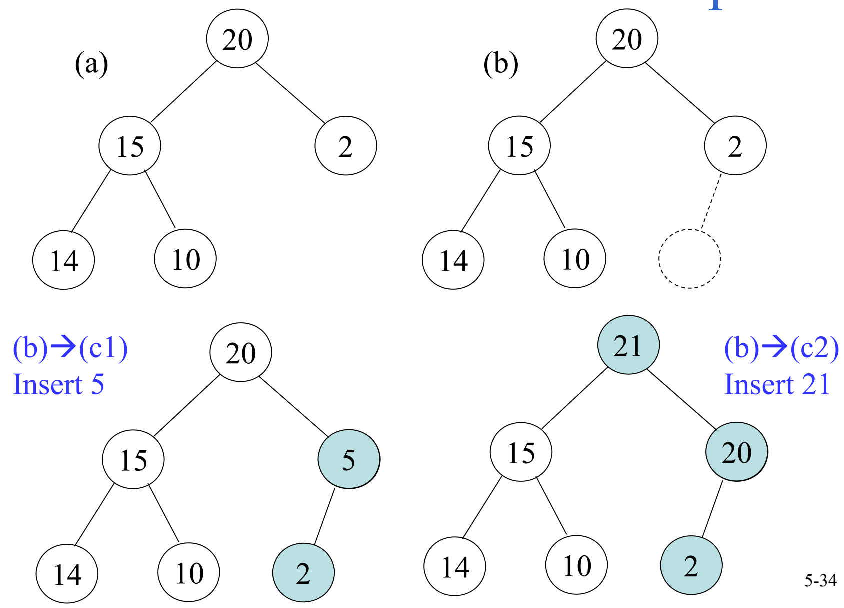
- Maximum (minimum) finding: In a priority queue, the element to be deleted is the one with highest (or lowest) priority. It is easy to get the maximum (minimum).
- Insertion: An element with arbitrary priority can be inserted into the queue according to its priority.
- max (min) priority queue: a data structure supports the above two operations.



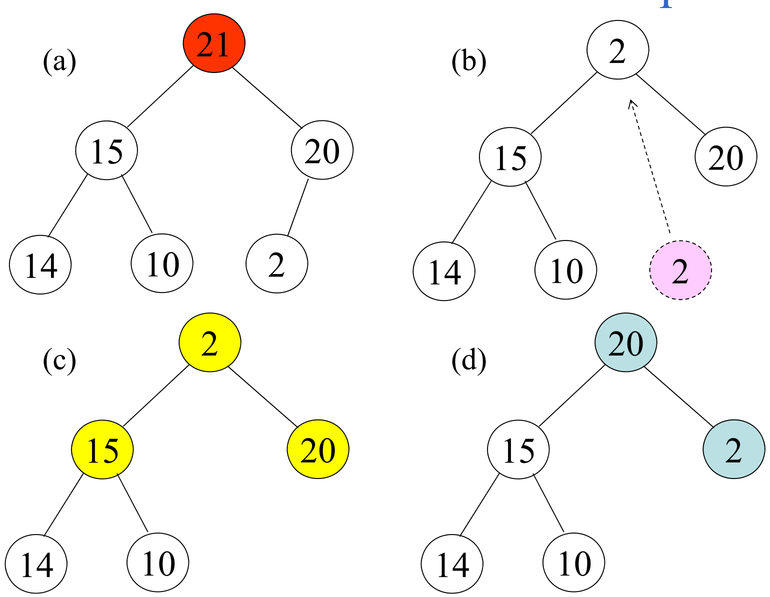
Max (Min) Heaps

- A <u>max (min) heap</u> is a <u>complete binary tree</u> in which the key value in each node is no smaller (larger) than the key values in its children (if any).
- <u>Heaps</u> are frequently used to implement priority queues.
- <u>Time complexity</u> of a max (min) heap with *n* nodes:
  - Max (min) finding: O(1)
  - Insertion: O(log *n*)
  - Deletion: O(log *n*)

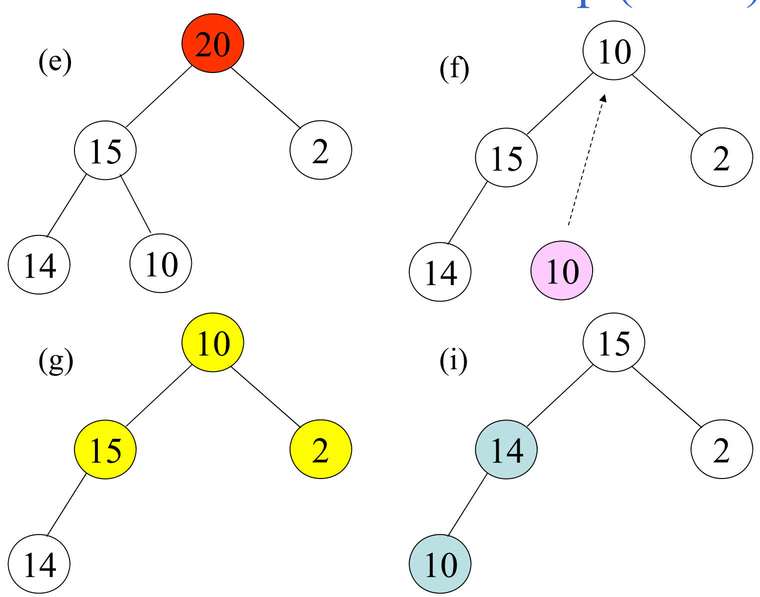
Insertion into a Max Heap



Deletion from a Max Heap



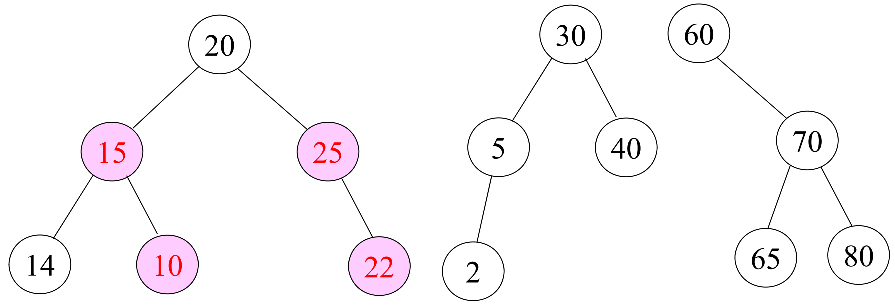
Deletion from a Max Heap (Cont.)



Binary Search Trees

- A binary search tree is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:
  - Every element has a key and no two elements have the same key (i.e., the keys are distinct)
  - The keys (if any) in the left subtree are smaller than the key in the root.
  - The keys (if any) in the right subtree are larger than the key in the root.
  - The left and right subtrees are also binary search trees.

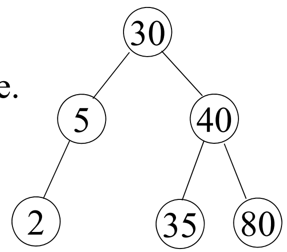
Binary Trees



Not binary search tree   Binary search trees

Searching a Binary Search Tree

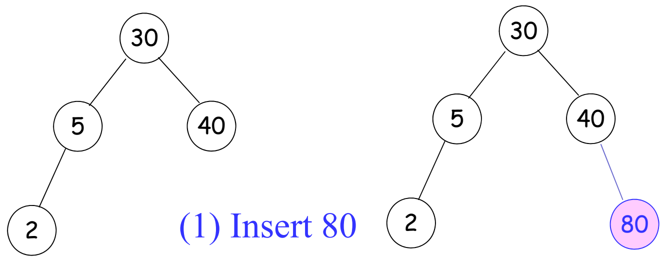
- Search for an element with key value *k*.
- If the root is 0, then this is an empty tree, and the search is unsuccessful.
- Otherwise, compare *k* with the key in the root.
  - ➢ *k==root*: successful search.
  - ➢ *k < root*: search the left subtree.
  - ➢ *k > root*: search the right subtree.

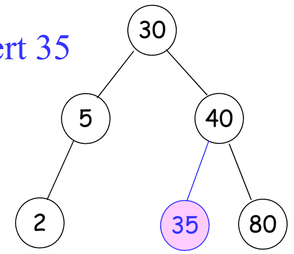


Searching a Binary Search Tree

```
template <class K, class E> // driver
pair<K, E>* BST<K, E> :: Get(const K& k)
{// If found, return a pointer; otherwise, return 0.
 // "pair": A class couples together two fields, first (K) and second (E).
 // K: key, E: node element
  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) { // first field of pair, K
    return Get(p->leftChild, k);
  if (k > p->data.first) //data has two fields, first, second
    return Get(p->rightChild, k);
  return &p->data;
}
```

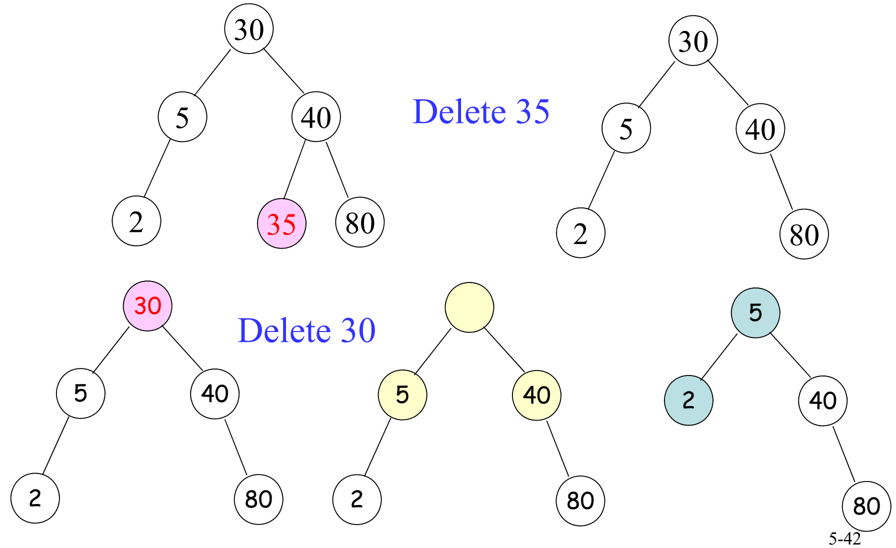
Insertion into a Binary Search Tree





Time complexity: O(h)
h: tree height

Deletion from a Binary Search Tree



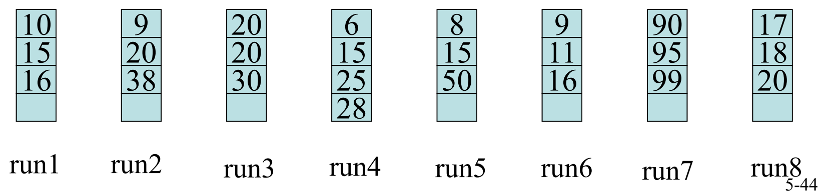
Deletion from a Binary Search Tree

- Deletion of a node x:
  - x is a leaf node: delete x directly.
  - x has one child: move up the position of child to x.
  - x has two children: replace x with either inorder successor (smallest in the right subtree, no left child) or inorder predecessor (largest in the left subtree, no right child).
- Time complexity: O(h), h:tree height

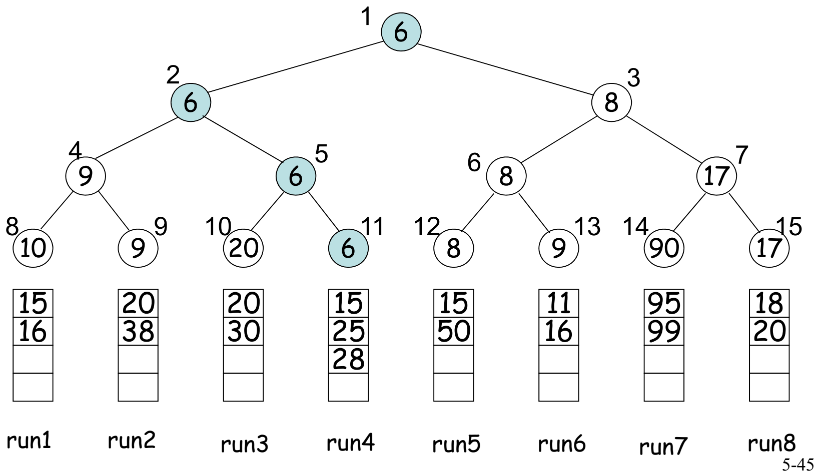
Selection Trees

- Goal: merge (合併) k ordered sequences (called runs) in nondecreasing into a single ordered sequence.
- Straightforward method: perform $k-1$ comparisons each time to select the smallest one among the first number of each of the k ordered sequences.
- Better method: winner tree

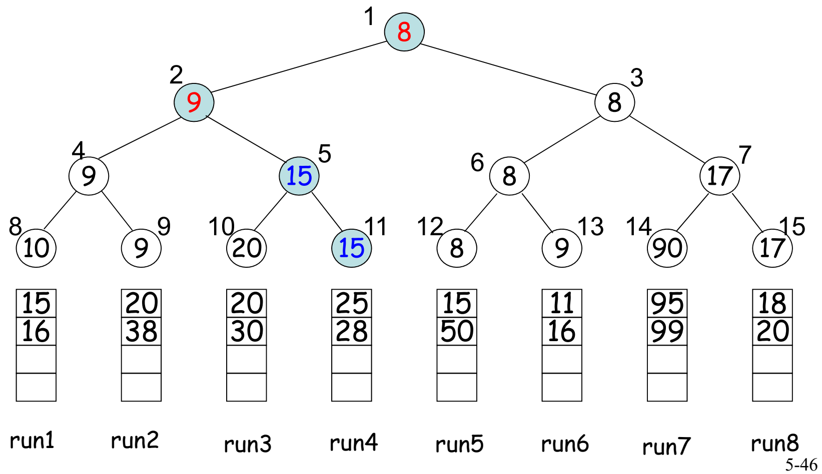
$k = 8$



Winner Tree for $k = 8$



Winner Tree for $k = 8$

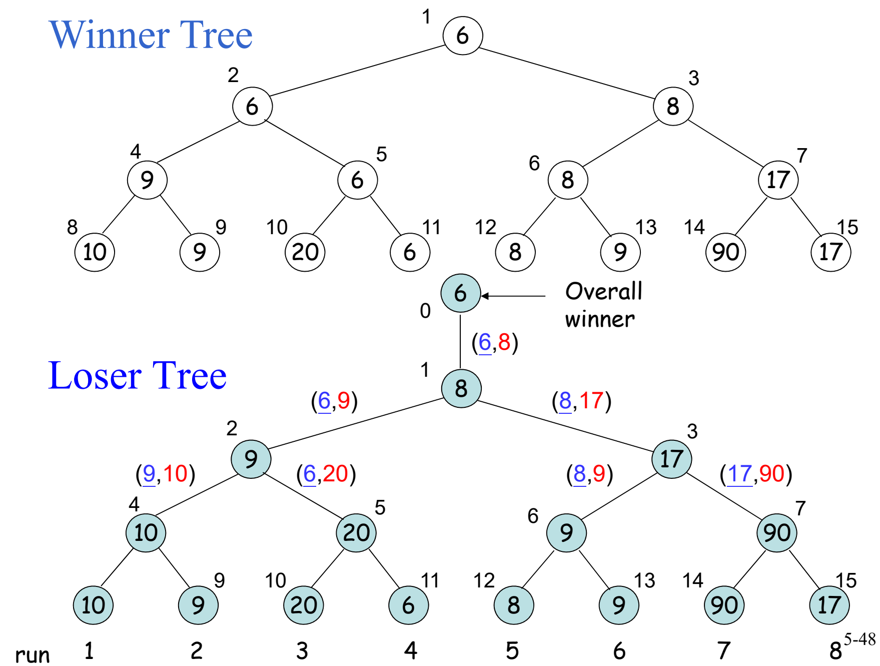


Winner Tree

- A winner tree is a complete binary tree in which each node represents the smaller of its two children. Thus, the root represents the smallest.
- Each leaf node represents the first record in the corresponding run.
- Each nonleaf node represents the winner of a tournament.
- Number of levels: $\lceil \log_2(k+1) \rceil$
- Total time for merging k runs: $O(n \log_2 k)$

Winner Tree

Loser Tree

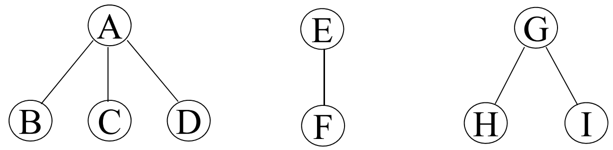


Loser Tree

- Loser tree: A selection tree in which each nonleaf node retains a pointer to the loser.
- Each leaf node represents the first record of each run.
- An additional node, node 0, has been added to represent the overall winner of the tournament.

Forests

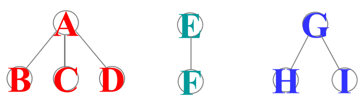
- Forest: a set of $n \geq 0$ disjoint trees.
- If we remove the root of a tree, we obtain a forest.
  - E.g., removing the root of a binary tree produces a forest of two trees.



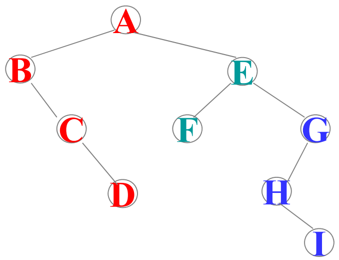
Representing a Forest with a Binary Tree

- leftChild=first child
- rightChild=next sibling

A forest of 3 trees



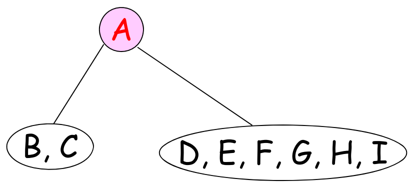
A forest is represented by a binary tree.



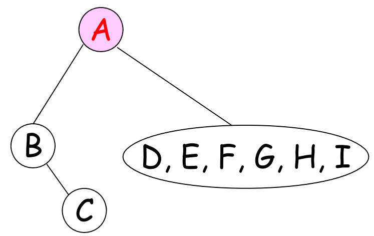
Constructing a Binary Tree from Its Inorder Sequence and Preorder Sequence

Inorder sequence: BCAEDGHFI
Preorder sequence: ABCDEFGHI

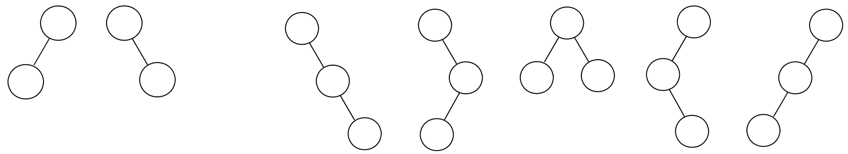
Step 1: Find the root



Step 2: Do recursively



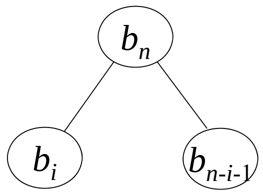
Number of Distinct Binary Trees



$n = 2$, $b_2=2$ $\qquad$ $n = 3$, $b_3=5$

- Number of distinct binary trees with n nodes:

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1}, \quad n \ge 1, \text{ and } b_0 = 1, b_1 = 1$$



Generating function, let $B(x) = \sum_{i \ge 0} b_i x^i$

$$b_n = b_0 b_{n-1} + b_1 b_{n-2} + \ldots + b_{n-2} b_1 + b_{n-1} b_0$$

$$B(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3 + \ldots$$

$$B^2(x) = b_0^{\,2} + 2b_0 b_1 x + (2b_0 b_2 + b_1^{\,2})x^2 + (2b_0 b_3 + 2b_1 b_2)x^3 + \ldots$$

$$= b_1 + b_2 x + b_3 x^2 + b_4 x^3 + \ldots.$$

With $b_0 = b_1 = 1$, we get the identity

$$xB^2(x) = B(x) - 1$$

Solving quadratics, we get

$$B(x) = \frac{1 - \sqrt{1-4x}}{2x}$$

Note: $B(x) = \dfrac{1 + \sqrt{1-4x}}{2x}$ is infeasible, becasue

$2xB(x) \ne 1 + \sqrt{1-4x}$ when $x = 0$, with $B(0) = b_0 = 1$

The binomial theorem:

$$(x+y)^k = \binom{k}{0} x^k y^0 + \binom{k}{1} x^{k-1} y^1 + \binom{k}{2} x^{k-2} y^2 + \ldots + \binom{k}{k} x^0 y^k$$

Expanding $(1-4x)^{1/2}$ with the binomial theorem, we get

$$B(x) = \frac{1}{2x}\left(1 - \sum_{n \ge 0} \binom{1/2}{n} (-4x)^n\right)$$

$$= \frac{1}{2x}\left(1 - \left(1 + \binom{1/2}{1}(-4x)^1 + \binom{1/2}{2}(-4x)^2 + \binom{1/2}{3}(-4x)^3 + \ldots\right)\right)$$

$$= \binom{1/2}{1}(2^1) + \binom{1/2}{2}(-2^3 x) + \binom{1/2}{3}(2^5 x^2) + \binom{1/2}{4}(-2^7 x^3) + \ldots$$

$$= \sum_{m \ge 0} \binom{1/2}{m+1} (-1)^m 2^{2m+1} x^m$$

The coefficient of x^n:

$$b_n = \binom{1/2}{n+1} (-1)^n 2^{2n+1}$$

$$= \frac{\frac{1}{2}(\frac{1}{2}-1)(\frac{1}{2}-2)(\frac{1}{2}-3)\ldots(\frac{1}{2}-n+1)(\frac{1}{2}-n)}{1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \cdot (n+1)} (-1)^n 2^{2n+1}$$

$$= \frac{(2-1)(4-1)(6-1)\ldots(2(n-1)-1)(2n-1)}{1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \cdot (n+1)} \cdot 2^n \cdot \frac{n!}{n!}$$

$$= \frac{1 \cdot 3 \cdot 5 \cdot \ldots \cdot (2n-3)(2n-1)}{1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \cdot (n+1)} \cdot \frac{2 \cdot 4 \cdot 6 \cdot \ldots \cdot 2n}{n!}$$

$$= \frac{(2n)!}{(n+1)(n!)(n!)} = \frac{1}{n+1}\binom{2n}{n}$$

Number of Distinct Binary Trees

- b_n are Catalan numbers:1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, …
- Number of distinct binary trees with n nodes= b_n
  - b_0=1, b_1=1, b_2=2, b_3=5, b_4=14, b_5=42, …
- http://en.wikipedia.org/wiki/Catalan_number

Data Structures

Chapter 7: Sorting

Sorting (1)

- List: a collection of records
  - Each record has one or more fields.
  - Key: used to distinguish among the records.

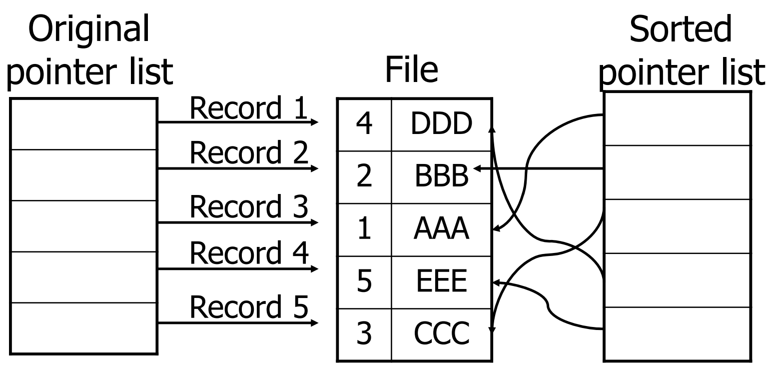
	Key	Other fields
Record 1	4	DDD
Record 2	2	BBB
Record 3	1	AAA
Record 4	5	EEE
Record 5	3	CCC

original list

Key	Other fields
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

sorted list

Sorting (2)



- It is easier to search a particular element after sorting. (e.g. binary search)
- best sorting algorithm with comparisons: O(nlogn)

Sequential Search

- Applied to an array or a linked list.
- Data are not sorted.
- Example: 9 5 6 8 7 2
  - search 6: successful search
  - search 4: unsuccessful search
- # of comparisons for a successful search on record key i is i.
- Time complexity
  - successful search: $(n+1)/2$ comparisons = $O(n)$
  - unsuccessful search: n comparisons = $O(n)$

Code for Sequential Search

```
template <class E, class K>
int SeqSearch (E *a, const int n, const K& k)
{// K: key,   E: node element
 //Search a[1:n] from left to right. Return least i such that
 // the key of a[i] equals k. If there is no i, return 0.
  int i;
  for (i = 1 ; i <= n && a[i] != k ; i++ );
    if (i > n) return 0;
  return i;
}
```


Motivation of Sorting

- A binary search needs O(log n) time to search a key in a sorted list with n records.
- Verification problem: To check if two lists are equal.
  - 6 3 7 9 5
  - 7 6 5 3 9
- Sequential searching method: O(mn) time, where m and n are the lengths of the two lists.
- Compare after sort: O(max{$n$log$n$, $m$log$m$})
  - After sort: 3 5 6 7 9 and 3 5 6 7 9
  - Then, compare one by one.

Categories of Sorting Methods

- stable sorting : the records with the same key have the same relative order as they have before sorting.
  - Example: Before sorting 6 3 7_a 9 5 7_b
  - After sorting 3 5 6 7_a 7_b 9
- internal sorting: All data are stored in main memory (more than 20 algorithms).
- external sorting: Some of data are stored in auxiliary storage.

Insertion Sort

e.g. (由小而大 sort， nondecreasing order)

5	9	2	8	6	pass 1
5	9	2	8	6	pass 2
5	2	9	8	6	
2	5	9	8	6	pass 3
2	5	8	9	6	
2	5	8	9	6	pass 4
2	5	8	6	9	
2	5	6	8	9	
2	5	6	8	9	

Insertion Sort

- 方法: 每次處理一個新的資料時，由右而左insert至其適當的位置才停止。
- 需要 n-1 個 pass
- best case: 未 sort 前，已按順序排好。每個pass僅需一次比較, 共需 (n-1) 次比較
- worst case: 未 sort 前, 按相反順序排好。比較次數為：

$$1+2+3+\cdots+(n-1)=\frac{n(n-1)}{2}=O(n^2)$$

- Time complexity: $O(n^2)$

Insertion into a Sorted List

```
template <class T>
void Insert(const T& e, T* a, int i)
// Insert e into the nondecreasing sequence a[1], …,
  a[i] such that the resulting sequence is also
  ordered. Array a must have space allocated for at
  least i+2 elements
{
   a[0] = e; // Avoid a test for end of list (i<1)
   while (e < a[i])
   {
      a[i+1] = a[i];
      i--;
   }
   a[i+1] = e;
}
```


Insertion Sort

```
template <class T>
void InsertionSort(T* a, const int n)
// Sort a[1:n] into nondecreasing order
{
   for (int j = 2; j <= n; j++)
   {
      T temp = a[j];
      Insert(temp, a, j-1);
   }
}
```


Quick Sort

- Input: 26, 5, 37, 1, 61, 11, 59, 15, 48, 19

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
[26	5	19	1	61	11	59	15	48	37]	1	10
[26	5	19	1	15	11	59	61	48	37]	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		

Quick Sort

- Quick sort 方法： 以每組的第一個資料為基準(pivot)，把比它小的資料放在左邊，比它大的資料放在右邊，之後以pivot中心，將這組資料分成兩部份。然後，兩部分資料各自recursively執行相同方法。

- 平均而言，Quick sort 有很好效能。

Code for Quick Sort

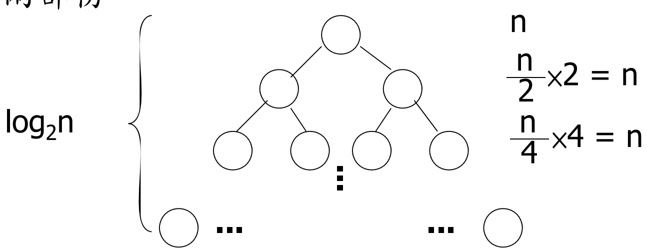
```
void QuickSort(Element* a, const int left, const int right)
// Sort a[left:right] into nondecreasing order.
// Key pivot = a[left].
// i and j are used to partition the subarray so that
// at any time a[m]<= pivot, m < i, and a[m]>= pivot, m > j.
// It is assumed that a[left] <= a[right+1].
{
   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);
   }
}
```


Time Complexity of Quick Sort

- Worst case time complexity: 每次的基準恰為最大，或最小。所需比較次數：

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}=\mathrm{O}(n^2)$$

- Best case time complexity：O(nlogn)
  – 每次分割(partition)時, 都分成大約相同數量的兩部份 。



Mathematical Analysis of Best Case

- $T(n)$: Time required for sorting n data elements.

$T(1) = b$, for some constant b

$T(n) \le cn + 2T(n/2)$, for some constant c

$$\begin{aligned}
&\le cn + 2(cn/2 + 2T(n/4))\\
&\le 2cn + 4T(n/4)\\
&\ \ \vdots\\
&\le cn \log_2 n + T(1)\\
&= \mathrm{O}(n \log n)
\end{aligned}$$

Variations of Quick Sort

- Quick sort using a median of three:
  - Pick the median of the first, middle, and last keys in the current sublist as the pivot. Thus, pivot = median$\{K_l, K_{(l+n)/2}, K_n\}$.
- Use the selection algorithm to get the real median element.
  - Time complexity of the selection algorithm: O(n).

Two-way Merge

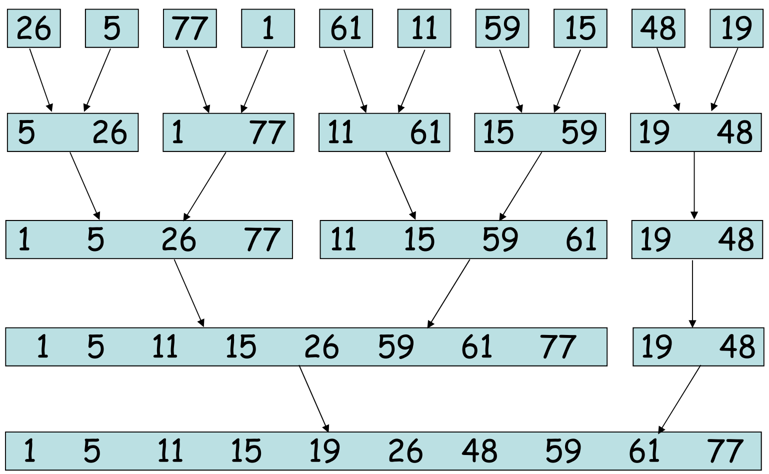
- Merge two sorted sequences into a single one.

```
[1  5  26  77][11  15  59  61]
           ⇩ merge
[1  5  11  25  26  59  61  77]
```

- 設兩個 sorted lists 長度各為 m, n

  Time complexity: O(m+n)

Iterative Merge Sort



Iterative Merge Sort

```
template <class T>
void MergeSort(T* a, const int n)
// Sort a[1:n] into nondecreasing order.
{
   T* tempList = new T[n+1];
   // l: 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);
      //interchange role of a and tempList
   }
   delete[ ] tempList; // free an array
}
```

Code for Merge Pass

```
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.   n: # of records in initList.
  for (int i = 1; // i is first position in first of the sublists being merged
    i <= n-2*s+1; // enough elements for two sublists of length s?
    i+ = 2*s)
      Merge(initList, resultList, i, i+s-1, i+2*s-1);
       // merge [i:i+s-1] and [i+s:i+2*s-1]
   // merge remaining list of size < 2 * s
   if ((i + s -1) < n )
     Merge(initList, resultList, i, i + s -1, n);
   else copy(initList + i, initList + n + 1,
resultList + i);
}
```

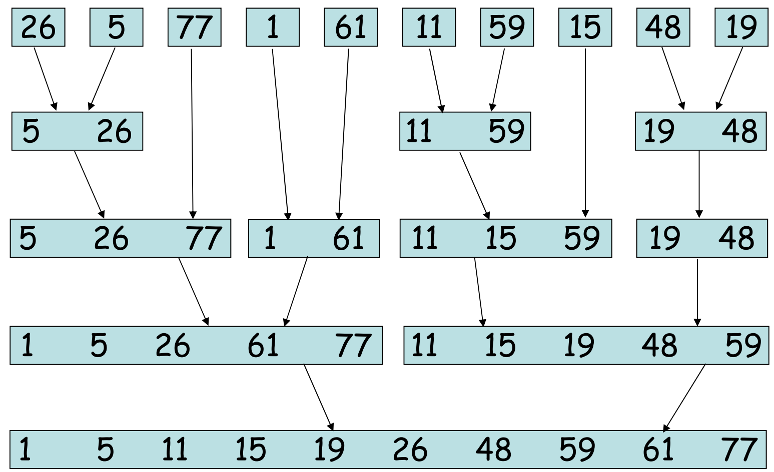

Analysis of Merge Sort

- Merge sort is a stable sorting method.
- Time complexity: O(n log n)
  - $\lceil \log_2 n \rceil$ passes are needed.
  - Each pass takes O(n) time.

Recursive Merge Sort

- Divides the list to be sorted into two roughly equal parts:
  - left sublist [left : (left+right)/2]
  - right sublist [(left+right)/2 +1 : right]
- These two sublists are sorted recursively.
- Then, the two sorted sublists are merged.
- To eliminate the record copying, we associate an integer pointer (instead of real link) with each record.

Recursive Merge Sort



Code for Recursive Merge Sort

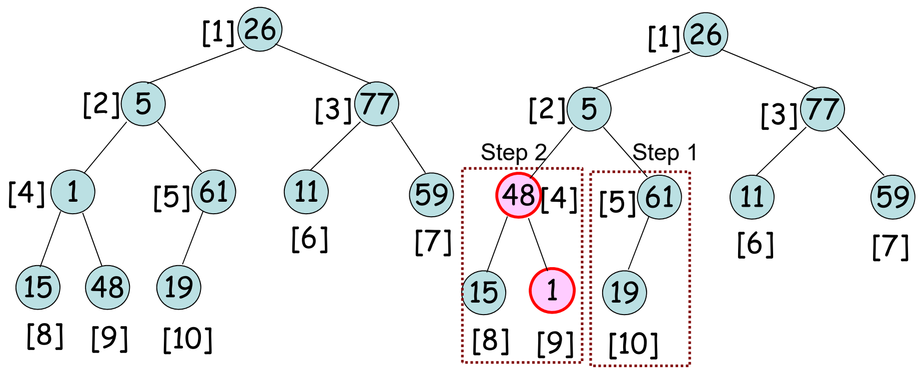
```
template <class T>
int rMergeSort(T* a, int* link, const int
left, const int right)
{// a[left:right] is to be sorted. link[i]
is initially 0 for all i.
 // rMergSort returns the index of the first
element in the sorted chain.
  if (left >= right) return left;
  int mid = (left + right) /2;
  return ListMerge(a, link,
  rMergeSort(a, link, left, mid),
        // sort left half
  rMergeSort(a, link, mid + 1, right));
      // sort right half
}
```


Merging Sorted Chains

```
tamplate <class T>
int ListMerge(T* a, int* link, const int start1, const int
start2)
{// The Sorted chains beginning at start1 and start2, respectively, are merged.
 // link[0] is used as a temporary header. Return start of merged chain.
  int iResult = 0; // last record of result chain
  for (int i1 = start1, i2 =start2; i1 && i2; )
    if (a[i1] <= a[i2]) {
      link[iResult] = i1;
      iResult = i1; i1 = link[i1];  }
    else {
      link[iResult] = i2;
      iResult = i2; i2 =link[i2];  }
  // attach remaining records to result chain
  if (i1 = = 0) link[iResult] = i2;
  else link[iResult] = i1;
  return link[0];
}
```

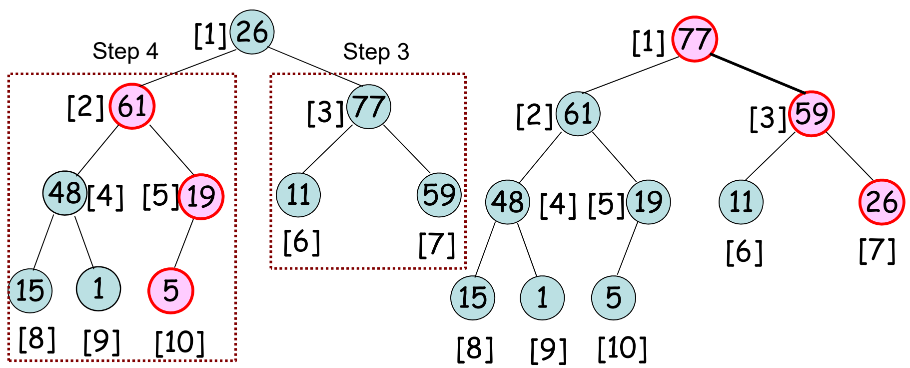

Heap Sort (1)

Phase 1: Construct a max heap.



(a) Input array (b) After Adjusting 61, 1

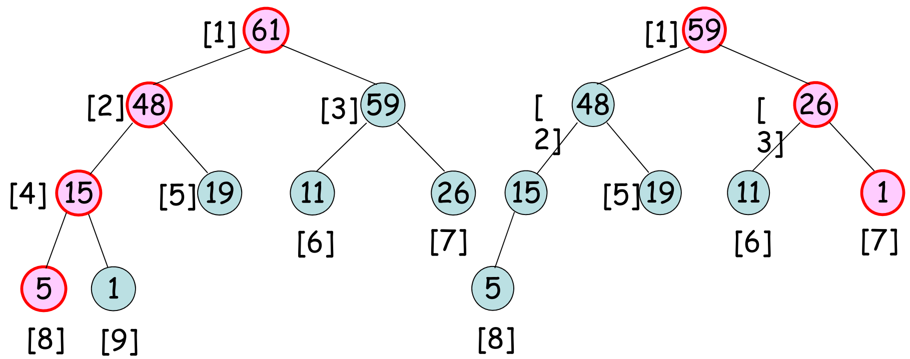
Heap Sort (2)



(c) After Adjusting 77, 5 (d) Max heap after constructing

Heap Sort (3)

Phase 2: Output and adjust the heap.
Time complexity: O(*n*log*n*)



Heap size = 9
a[10] = 77

Heap size = 8
a[9]=61, a[10]=77

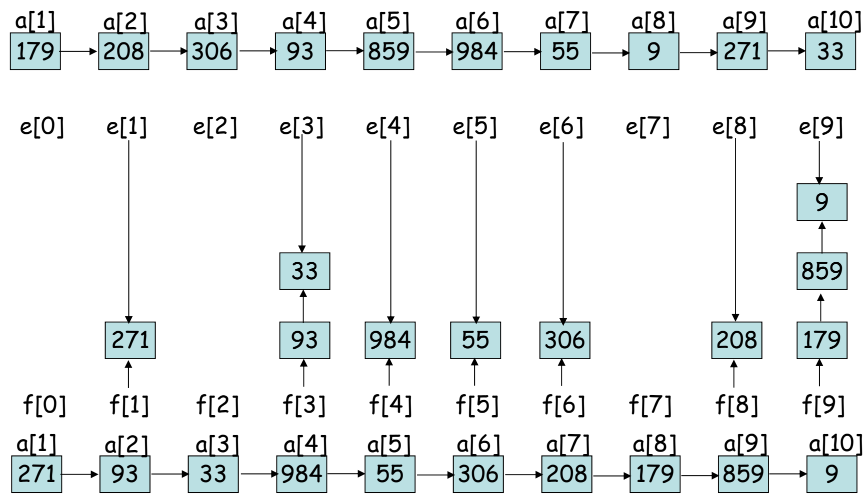
Adjusting a Max Heap

```
template <class T>
void Adjust(T *a, const int root, const int n)
{// Adjust binary tree with root root to satisfy heap property. The left and right
 // subtrees of root already satisfy the heap property. 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 max child of its parent
    if (e >= a[j]) break; // e may be inserted as parent of j
    a[j / 2] = a[j]; // move jth record up the tree
  }
  a[j / 2] = e;
}
```

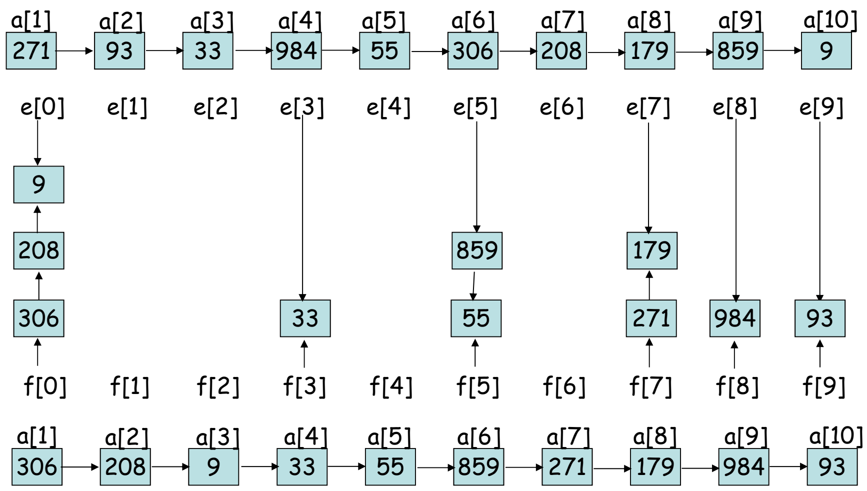

Heap Sort

```
template <class T>
void HeapSort(T *a, const int n)
{// Sort a[1:n] into nondecreasing order
  for (int i = n/2; i >= 1; i--) // heapify
    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);  // heapify
  }
}
```

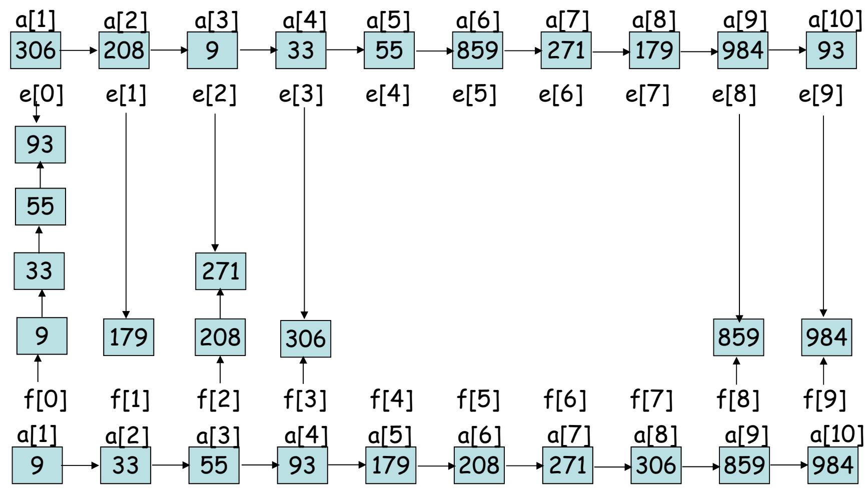

Radix Sort基數排序: Pass 1 (nondecreasing)



Radix Sort: Pass 2



Radix Sort: Pass 3



Radix Sort

- 方法：least significant digit first (LSD)

  (1)每個資料不與其它資料比較，只看自己放在何處

  (2)pass 1 ：從個位數開始處理。若是個位數為 1，則放在 bucket 1，以此類推…

  (3)pass 2 ：處理十位數，pass 3：處理百位數..
- 好處：若以array處理，速度快
- Time complexity: $O((n+r)\log_r k)$
  - k: input data 之最大數
  - r: 以 r 為基數(radix)， $\log_r k$: 位數之長度
- 缺點：若以array處理需要較多記憶體。使用linked list，可減少所需記憶體，但會增加時間

List Sort

- All sorting methods require excessive data movement.
- The physical data movement tends to slow down the sorting process.
- When sorting lists with large records, we have to modify the sorting methods to minimize the data movement.
- Methods such as insertion sort or merge sort can be modified to work with a linked list rather than a sequential list. Instead of physically moving the record, we change its additional link field to reflect the change in the position of the record in the list.
- After sorting, we may physically rearrange the records in place.

Rearranging Sorted Linked List (1)

Sorted linked list, first=4

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
linka	9	6	0	2	3	8	5	10	7	1

Add backward links to become a doubly linked list, first=4

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
linka	9	6	0	2	3	8	5	10	7	1
linkb	10	4	5	0	7	2	9	6	1	8

Rearranging Sorted Linked List (2)

R_1 is in place. first=2

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	77	26	61	11	59	15	48	19
linka	2	6	0	9	3	8	5	10	7	4
linkb	0	4	5	10	7	2	9	6	4	8

R_1, R_2 are in place. first=6

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	77	26	61	11	59	15	48	19
linka	2	6	0	9	3	8	5	10	7	4
linkb	0	4	5	10	7	2	9	6	4	8

Rearranging Sorted Linked List (3)

R_1, R_2, R_3 are in place. first=8

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	11	26	61	77	59	15	48	19
linka	2	6	8	9	6	0	5	10	7	4
linkb	0	4	2	10	7	5	9	3	4	8

R_1, R_2, R_3, R_4 are in place. first=10

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	11	15	61	77	59	26	48	19
linkb	2	6	8	10	6	0	5	9	7	8
linkb	0	4	2	3	7	5	9	10	8	8

Rearranging Records Using A Doubly Linked List

```
template <class T>
void List1(T* a, int *lnka, const int n, int first)
{
  int *linkb = new int[n]; // backward links
  int prev = 0;
  for (int current = first; current; current = linka[current])
  { // convert chain into a doubly linked list
    linkb[current] = prev;
    prev = current;
  }
  for (int i = 1; i < n; i++)
  {// move a[first] to position i
    if (first != i) {
      if (linka[i]) linkb[linka[i]] = first;
      linka[linkb[i]] = first;
      swap(a[first], a[i]);
      swap(linka[first], linka[i]);
      swap(linkb[first], linkb[i]);
    }
    first = linka[i];
  }
}
```

Table Sort

- The list-sort technique is not well suited for quick sort and heap sort.
- One can maintain an auxiliary table, *t*, with one entry per record, an indirect reference to the record.
- Initially, *t*[*i*] = *i*. When a swap are required, only the table entries are exchanged.
- After sorting, the list a[t[1]], a[t[2]], a[t[3]]…are sorted.

- Table sort is suitable for all sorting methods.

Permutation Cycle

- After sorting (nondecreasing):

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
key	35	14	12	42	26	50	31	18
t	3	2	8	5	7	1	4	6

- Permutation [3 2 8 5 7 1 4 6]
- Every permutation is made up of disjoint permutation cycles:
  - (1, 3, 8, 6) nontrivial cycle
    - R1 now is in position 3, R3 in position 8, R8 in position 6, R6 in position 1.
  - (4, 5, 7) nontrivial cycle
  - (2) trivial cycle

Table Sort Example

Initial configuration

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
key	35	14	12	42	26	50	31	18
t	3	2	8	5	7	1	4	6

1 2 3 4 5

after rearrangement of first cycle

key	12	14	18	42	26	35	31	50
t	1	2	3	5	7	6	4	8

after rearrangement of second cycle

key	12	14	18	26	31	35	42	50
t	1	2	3	4	5	6	7	8

Code for Table Sort

```
template <class T>
void Table(T* a, const int n, int *t)
{
   for (int i = 1; i < n; i++) {
      if (t[i] != i) {// nontrivial cycle starting at i
         T p = a[i];
         int j = i;
         do {
            int k = t[j]; a[j] = a[k]; t[j] = j;
            j = k;
         } while (t[j] != i)
         a[j] = p;  // j is the position for record p
         t[j] = j;
      }
   }
}
```

Summary of Internal Sorting

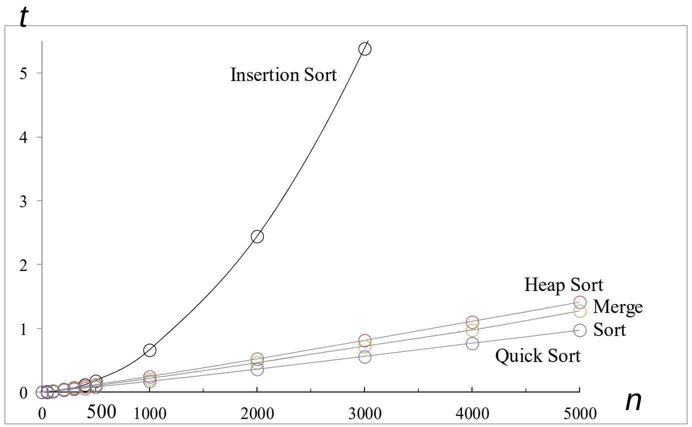
- No one method is best under all circumstances.
  - Insertion sort is good when the list is already partially ordered. And it is the best for small n.
  - Merge sort has the best worst-case behavior but needs more storage than heap sort.
  - Quick sort has the best average behavior, but its worst-case behavior is $O(n^2)$.
  - The behavior of radix sort depends on the size of the keys and the choice of r.

Complexity Comparison of Sort Methods

Method	Worst	Average
Insertion Sort	n^2	n^2
Heap Sort	$n \log n$	$n \log n$
Merge Sort	$n \log n$	$n \log n$
Quick Sort	n^2	$n \log n$
Radix Sort	$(n+r)\log_r k$	$(n+r)\log_r k$

k: input data 之最大數　r: 以 r 為基數(radix)

Average Execution Time



Average execution time, n = # of elements, t=milliseconds

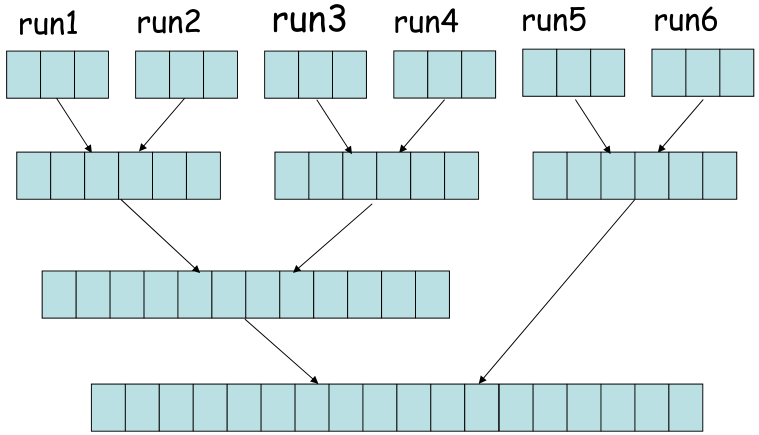
External Sorting

- The lists to be sorted are too large to be contained totally in the internal memory. So internal sorting is impossible.
- The list (or file) to be sorted resides on a disk.
- Block: unit of data read from or written to a disk at one time. A block generally consists of several records.
- read/write time of disks:
  - seek time 搜尋時間：把讀寫頭移到正確磁軌 (track, cylinder)
  - latency time 延遲時間：把正確的磁區(sector)轉到讀寫頭下
  - transmission time 傳輸時間：把資料區塊傳入/讀出磁碟

Merge Sort as External Sorting

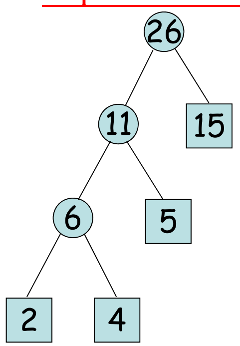
- The most popular method for sorting on external storage devices is merge sort.
- Phase 1: Obtain sorted runs (segments) by internal sorting methods, such as heap sort, merge sort, quick sort or radix sort. These sorted runs are stored in external storage.
- Phase 2: Merge the sorted runs into one run with the merge sort method.

Merging the Sorted Runs

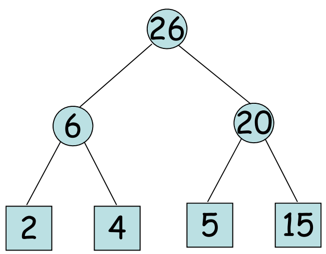


Optimal Merging of Runs

- In the external merge sort, the sorted runs may have different lengths. If shorter runs are merged first, the required time is reduced.



weighted external path length
= 2*3 + 4*3 + 5*2 + 15*1
= 43



weighted external path length
= 2*2 + 4*2 + 5*2 + 15*2
= 52

Huffman Algorithm

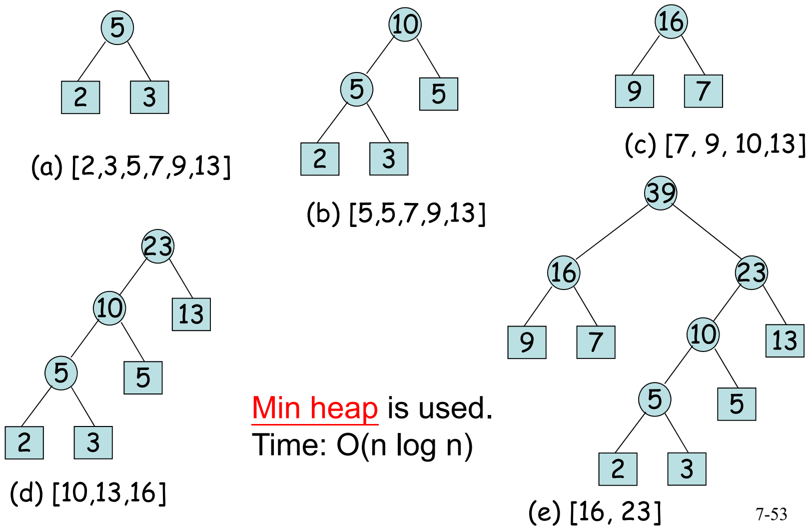
- External path length: sum of the distances of all external nodes from the root.
- Weighted external path length:

$$\sum_{1 \le i \le n+1} q_i d_i, \text{ where } d_i \text{ is the distance from root to node } i$$

$$q_i \text{ is the weight of node } i.$$

- Huffman algorithm: to solve the problem of finding a binary tree with minimum weighted external path length.
- Huffman tree:
  - Solve the 2-way merging problem
  - Generate Huffman codes for data compression

Construction of Huffman Tree



Min heap is used.
Time: O(n log n)

Huffman Code (1)

- Each symbol is encoded by 2 bits (fixed length)

symbol	code
A	00
B	01
C	10
D	11

- Message A B A C C D A would be encoded by 14 bits:

  00 01 00 10 10 11 00

Huffman Code (2)

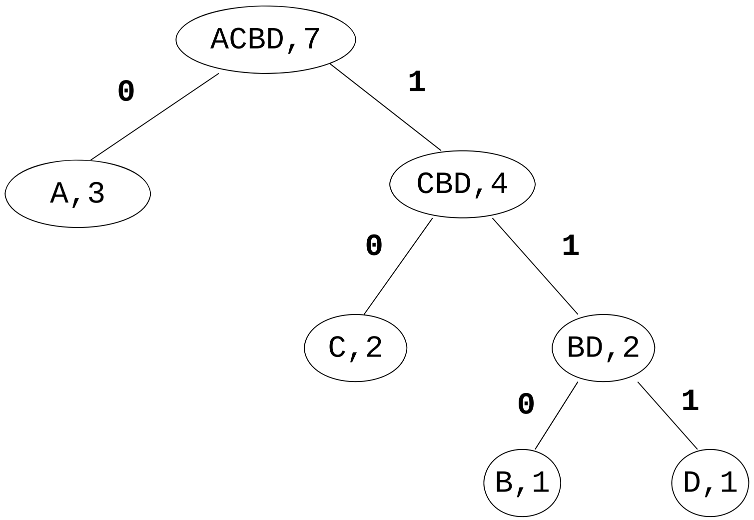
- Huffman codes (variable-length codes)

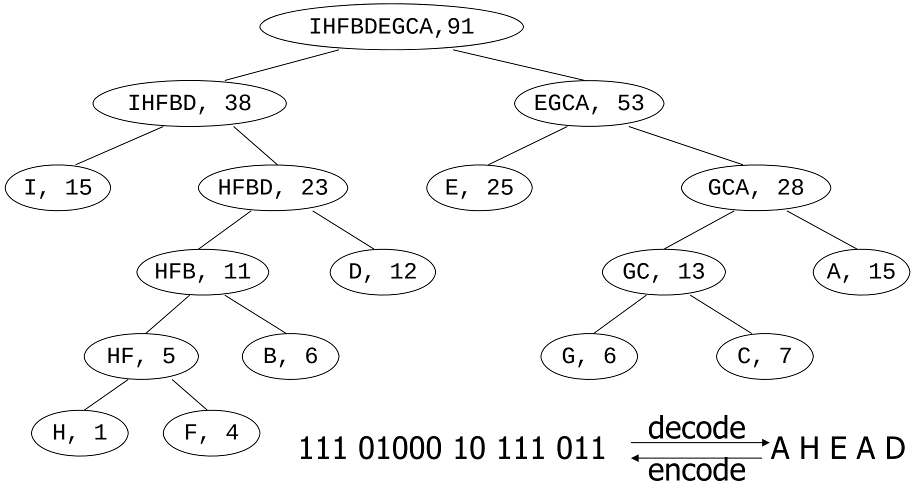
symbol	code
A	0
B	110
C	10
D	111

- Message A B A C C D A would be encoded by 13 bits:

  0 110 0 10 10 111 0
- A frequently used symbol is encoded by a short bit string.

Huffman Tree





Sym	Freq	Code	Sym	Freq	Code	Sym	Freq	Code
A	**15**	**111**	**D**	**12**	**011**	**G**	**6**	**1100**
B	**6**	**0101**	**E**	**25**	**10**	**H**	**1**	**01000**
C	**7**	**1101**	**F**	**4**	**01001**	**I**	**15**	**00**

Data Structures

Chapter 8: Hashing

Performance Comparison of Arrays and Trees

	Array		Tree	
	Sorted	Not Sorted	Unbalanced	Balanced (Chap. 10)
Insertion	O(n)	O(1)	O(h)	O(log n)
Searching	O(log n)	O(n)	O(h)	O(log n)
Deletion	O(n)	O(1)	O(h)	O(log n)

h: tree height

- Is it possible to perform these operations in O(1) ?

Static Hashing 雜湊

- hash function:
  to transform a key into a table index
- Example:
  - key values: 18 23 33 13 24 10
  - hash function:
    h(k) = k mod 10
- hash collision:
  Two records (keys) attempt to insert into the same bucket (position).

0	**10**
1	
2	
3	**23**
4	**33**
5	**13**
6	**24**
7	
8	**18**
9	

Hash Table 雜湊表

- In static hashing, identifiers/keys are stored in a hash table.
- Slot: space for storing data
- Bucket: Each bucket may consist of s slots to hold
- synonym (同義字)
  - k_1 and k_2 are synonyms if $h(k_1) = h(k_2)$
- Hash collision: home bucket for new data is not empty

	slot 1	slot 2
bucket 0	A	A2
bucket 1		
bucket 2		
bucket 3	D	
	…	
	GA	Gg
bucket 25		

Hash Functions

- A good hash function is easy to compute and minimizes # of collisions.
- Uniform hash function
  - Probability $p(h(x) = i)$ is $1/b$, where b is # of buckets
- Four popular hash functions:
  - Division 除法
  - Mid-square 平方取中間位元
  - Folding 折疊
  - Digit Analysis 數位分析

Division

- $h(k) = k \% D$ remainder (餘數)
- Since the bucket address is from 0 to b–1 if there are b buckets, usually $D=b$.
- If D is even, it is not good.
  - D = 12, 20%12 = 8, 30%12 = 4, 28%14 = 0
- If D is multiple (倍數) of 5, it is not good.
  - D = 15, 20%15 = 5, 30%15 = 0, 35%15 = 5
- Prime numbers (質數) are good choice for D .
- For most practical dictionaries, D is a good cadidate if it has no prime factor smaller than 20.

Mid-square

- Square the key and then use an appropriate number of bits from the middle of the square.
- Example
  - Key k= 113586, b =10000, where 9999 is the largest bucket address.
  - Square the key, and then extract the middle 4 digits:

1	2	9	0	1	7	7	9	3	9	6

$h(k) = 1779$

Folding

- The key k is partitioned into several parts, all of the same length. These partitions are then added together to obtain the hash address for k.

P1			P2			P3			P4			P5	
1	2	3	2	0	3	2	4	1	1	1	2	2	0

- Two methods:
  - Shift folding
  - Folding at the boundaries

Shift folding:

P1	1 2 3 →
P2	2 0 3 →
P3	2 4 1 →
P4	1 1 2 →
P5	2 0 →
	6 9 9

Folding at the boundaries:

P1	1 2 3 →
P2	3 0 2 ←
P3	2 4 1 →
P4	2 1 1 ←
P5	2 0 →
	8 9 7

Overflow Handling 溢位處理

- An overflow occurs when the home bucket for a new pair (key, element) is full.
- Two methods for handling overflows:
- Open addressing: Search the hash table in some systematic fashion for a bucket that is not full.
  - Linear probing 線性探測 (linear open addressing)
  - Quadratic probing 二次探測
  - Rehashing 再雜湊
- Chaining: Each bucket keeps a linked list of all pairs to the same bucket address.
  - Linked list

Linear Probing

- Also called linear open addressing
- Search the next available bucket one by one.
- $(h(k) + j)$ % D, j=0,1,2,3…
- Example
  - D (divisor) = b (number of buckets) = 17
  - Bucket address = key % 17
  - Insert pairs whose keys are

```
Key: 6, 12, 34, 29, 28, 11, 23, 7, 0, 33,
h(x):6, 12,  0, 12, 11, 11,  6, 7, 0, 16,
```

0				4				8				12				16
34	0					6	23	7			28	12	29	11		33

Quadratic Probing

- $h(x)$, $(h(x)+i^2)\%b$, and $(h(x)-i^2)\%b$, …
  i=1,2,3, …, $(b-1)/2$
- Every bucket will be examined when b is a prime number of the form $4j+3$.
  - For example, $b=4j+3$, b = 3, 7, 11, …,43, 59, …
- Hash function:
  - $h(x)$
  - $(h(x)+1^2)$ % b
  - $(h(x)-1^2)$ % b
  - $(h(x)+2^2)$ % b
  - $(h(x)-2^2)$ % b
  - …
  - $(h(x) \pm ((b-1)/2)^2)$ % b

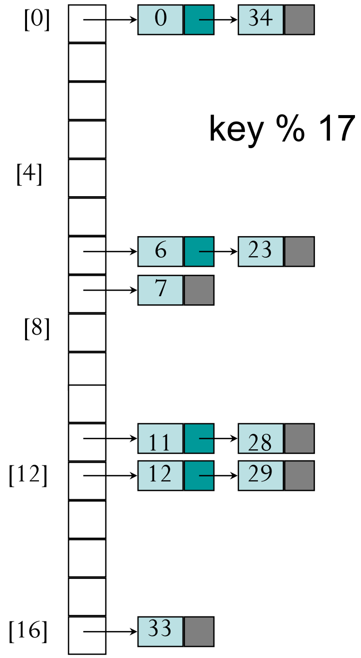
Example: b=7, $(b-1)/2=3$
i=1,2,3
Examine: 0,1,-1,4,-4,9,-9 (%7)
Equivalent to 0,1,6,4,3,2,5

Rehashing

- If the overflow occurs at $h_i(x)$, try $h_{i+1}(x)$.
- Use a series of hash function h_1, h_2, …, h_m to find an empty bucket.
- Example
  - $h_1(x) = x$ % 11
  - $h_2(x) = x$ % 251
  - $h_3(x) = x^2$ % 251
  - $h_4(x) = x$ % 1019
  - …

Chaining

- Disadvantage of linear probing
  - Comparison of identifiers with different hash values.
- Use linked lists to connect the identifiers with the same hash value and to increase the capacity of a bucket.



Dynamic Hashing

- Also called extendable hashing.
- Limitations of static hashing
  - When the table is to be full, overflows increase. As overflows increase, the overall performance decreases.
  - We cannot just copy entries from smaller into a corresponding buckets of a bigger table.
- Allow the size of dictionary to grow and shrink.
  - The size of hash table can be changed dynamically.
    - Hash function: $h(\ ) \rightarrow h'(\ )$
    - Size of hash table: $m \rightarrow m'$

Dynamic Hashing Using Directories

- The hash function $h(k)$ transforms k into 6-bit binary integer.

k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101
B4	101 100
B5	101 101

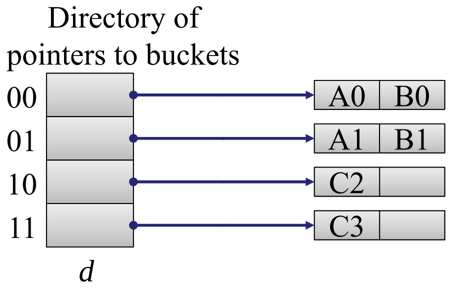
Dynamic Hashing Using Directories

- A directory, d, of pointers to buckets is used.
- Directory size $d = 2^t$, where t is the number of bits used to identify all $h(x)$.
  - Initially, $t = 2$. Thus, $d = 2^2 = 4$.
- $h(k, t)$ is defined as the t least significant bits (LSB) in $h(k)$, where t is also called the directory depth.
- Example:
  - $h(\text{C5}) = 110\ 101$
  - $h(\text{C5}, 2) = 01$
  - $h(\text{C5}, 3) = 101$

Extending the Directory

- Consider the following keys have been already stored. t=2 (directory depth=2) differentiates all input keys.

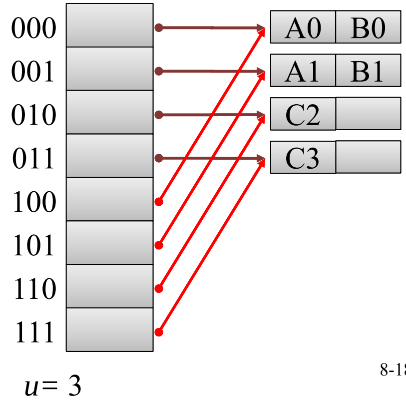
k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C2	110 010
C3	110 011
C5	**110 101**



Insertion of C5 (=110 101)

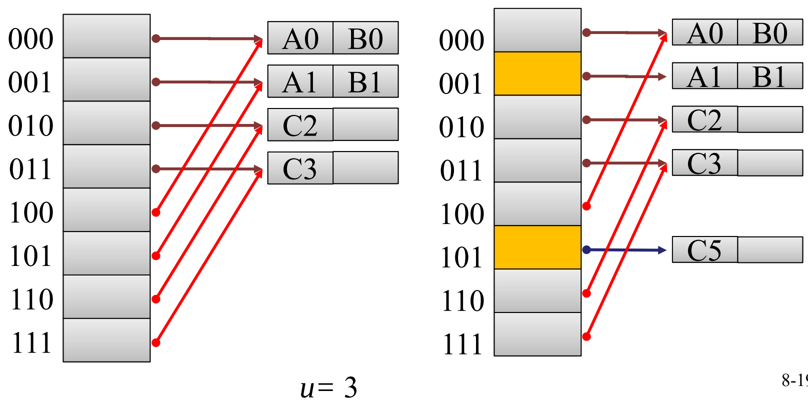
- t = 2 and h(C5, 2) = 01.
- A1 and B1 have been at d[01]. Bucket overflows.
- Decide u, $h(k,u)$ is not the same for A1, B1, C5. Then u=3.
- Extend d to 2^u. Duplicate the pointers to the new half.

k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C2	110 010
C3	110 011
C5	**110 101**



Insertion of C5 (=110 101)

- Split the overflowed bucket using $h(k,3)$. Rehash A1, B1, C5. A new bucket is created for C5.

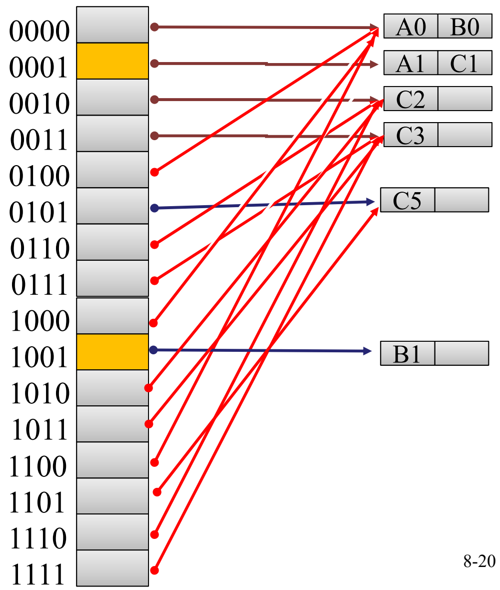


Insertion of C1 (=110 001)

k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C2	110 010
C3	110 011
C5	110 101
C1	**110 001**

$h(k,u)$ is not the same for A1, B1, C1: u=4.

Rehash A1, B1, C1 using $h(k,4)$



Advantages for Dynamic Hashing Using Directories

- Only double the directory rather than the whole hash table used in static hashing.
- Only rehash the entries in the buckets that overflows.

Directoryless Dynamic Hashing (1)

(a) $r = 2$, $q = 0$

00	B4 A0
01	A1 B5
10	C2 -
11	C3 -

Active: $0 \sim 2^r+q-1$
($0 \sim 3$)

(b) Insert C5(=110 101), $r = 2$, $q = 1$

000	A0 -	
001	A1 B5	overflow bucket: C5
010	C2 -	
011	C3 -	
100	B4 -	new active bucket

1. 01 overflows.
2. Activate 2^r+q
   (Activate 100)
3. Increment q
4. Rehash B4, A0 by $h(k,3)$
   (000, 100: $h(k,3)$)
   (001~011: $h(k,2)$)
5. Insert C5 in overflow area.

Directoryless Dynamic Hashing (2)

(c) Insert C1(=110 001), $r = 2$, $q = 2$

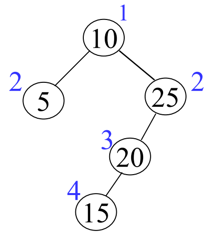
000	A0 -	
001	A1 C1	
010	C2 -	
011	C3 -	
100	B4 -	
101	B5 C5	new active bucket

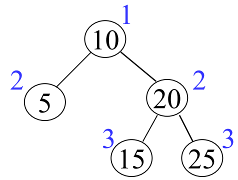
1. 001 overflows.
2. Activate 2^r+q
   (Activate 101)
3. Increment q
4. Rehash A1, B5, C5, C1 by $h(k,3)$
   (000~001, 100~101: $h(k,3)$)
   (010~011: $h(k,2)$)

Data Structures

Chapter 10: Efficient Binary Search Trees

Two Binary Search Trees (BST)

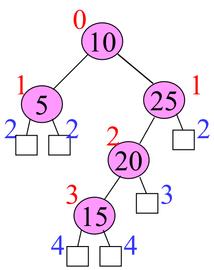




- For each identifier with equal searching probability , average # of comparisons for a successful search:
  Left: (1+2+2+3+4)/5 = 2.4
  Right: (1+2+2+3+3)/5 = 2.2
- prob(5, 10, 15, 20, 25) = (0.3, 0.3, 0.05, 0.05, 0.3)
  Left :0.3×(2+1+2)+0.05×(4+3) = 1.85
  Right: 0.3×(2+1+3)+0.05×(3+2) = 2.05

Extended Binary Trees

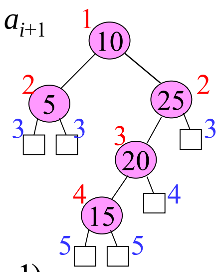
- Add external nodes to the original binary search tree
  - Take external nodes as failure nodes .
- External / internal path length
  - Internal path length, I, is:
    $I = 0 + 1 + 1 + 2 + 3 = 7$
  - External path length, E, is :
    $E = 2 + 2 + 4 + 4 + 3 + 2 = 17$
- $E = I + 2n$, where n is # of internal nodes.



Search Cost in a BST

- In the binary search tree(BST):
  - Identifiers $a_1, a_2, \ldots, a_n$ with $a_1 < a_2 < \ldots < a_n$
  - p_i :probability of successful search for a_i
  - q_i : probability of unsuccessful search $a_i < x < a_{i+1}$

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$



- Total cost

$$\sum_{i=1}^{n} p_i \cdot level(a_i) + \sum_{i=0}^{n} q_i \cdot (level(\text{failure node } i) - 1)$$

- An optimal binary search tree for $a_1, \ldots, a_n$ is the one that minimizes the total cost.

Algorithm for Constructing Optimal BST (1)

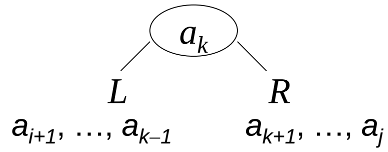
- Solved by dynamic programming.
- T_{ij} : an optimal binary search tree for $a_{i+1}, \ldots, a_j$, $i < j$.
  - T_{ii} is an empty tree for $0 \le i \le n$.
- c_{ij} : cost of T_{ij}, where c_{ii}=0.
- r_{ij} : root of T_{ij}
- w_{ij} : weight of T_{ij} , $w_{ij} = q_i + \sum_{k=i+1}^{j}(q_k + p_k)$
- T_{0n} is an optimal binary search for $a_1, \ldots, a_n$. cost: c_{0n} weight: w_{0n} root: r_{0n}

Algorithm for Constructing Optimal BST (2)

- Suppose a_k is the root of T_{ij} ($r_{ij} = k$).
- T has two subtrees L and R.
  - L: left subtree with $a_{i+1}, \ldots, a_{k-1}$
  - R: right subtree with $a_{k+1}, \ldots, a_j$

$$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R)$$
$$= p_k + c_{i,k-1} + c_{kj} + w_{i,k-1} + w_{kj}$$
$$= w_{ij} + c_{i,k-1} + c_{kj} \qquad (w_{ij} = p_k + w_{i,k-1} + w_{kj})$$
$$= w_{ij} + \min_{i<l\le j}\{c_{i,l-1} + c_{lj}\}$$

- Time complexity: $O(n^3)$



Example for Constructing Optimal BST (1)

- $n = 4$, $(a_1, a_2, a_3, a_4) = (\mathbf{10}, \mathbf{15}, \mathbf{20}, \mathbf{25})$.
  $16\times(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$
  $16\times(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$.
- Initially $w_{ii} = q_i$, $c_{ii} = 0$, and $r_{ii} = 0$, $0 \le i \le 4$

$w_{01} = p_1 + w_{00} + w_{11} = p_1 + q_0 + q_1 = 8$
$c_{01} = w_{01} + \min\{c_{00} + c_{11}\} = 8$, $r_{01} = 1$ // root=1

$w_{12} = p_2 + w_{11} + w_{22} = p_2 + q_1 + q_2 = 7$
$c_{12} = w_{12} + \min\{c_{11} + c_{22}\} = 7$, $r_{12} = 2$ // root=2

$w_{14} = 11$
$c_{14} = w_{14} + \min\{c_{11} + c_{24}, c_{12} + c_{34}, c_{13} + c_{44}\}$ // root=2, 3, 4
$= 11 + c_{11} + c_{24} = 19$, $r_{14} = 2$

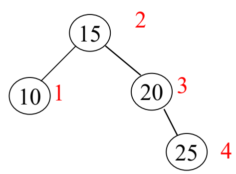
$w_{04} = 16$
$c_{04} = w_{04} + \min\{c_{00} + c_{14}, c_{01} + c_{24}, c_{02} + c_{34}, c_{03} + c_{44}\}$ // root=1,2, 3, 4
$= 16 + c_{01} + c_{24} = 32$, $r_{04} = 2$

Example for Constructing Optimal BST (2)

- $w_{ii} = q_i$
- $w_{ij} = p_k + w_{i,k-1} + w_{kj}$
- $c_{ij} = w_{ij} + \min_{i<l\le j}\{c_{i,l-1} + c_{lj}\}$
- $c_{ii} = 0$
- $r_{ii} = 0$
- $r_{ij} = l$

$(a_1, a_2, a_3, a_4) = (\mathbf{10}, \mathbf{15}, \mathbf{20}, \mathbf{25})$
$(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$
$(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$

	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				



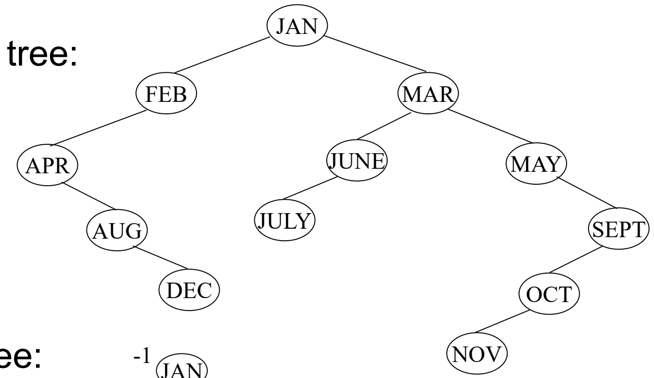
The optimal binary search tree

Computation is carried out row-wise from row 0 to row 4.

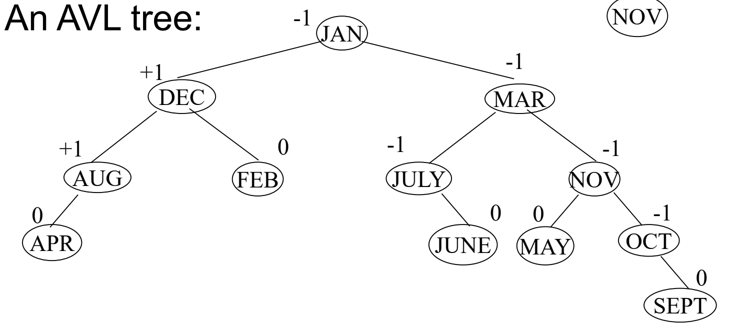
AVL Trees

- Height balanced binary search trees.
- Proposed by G. Adelson-Velsky and E. M. Landis
- Balance factor of each node v
  - $BF(v) = h_L - h_R = -1, 0,$ or 1
  - h_L: height of left subtree
  - h_R: height of right subtree
- We can insert an element into the tree, or delete an element from it, in $O(\log n)$ time.
- At most one single rotation or double rotation is needed when an insertion is performed.

Not an AVL tree:



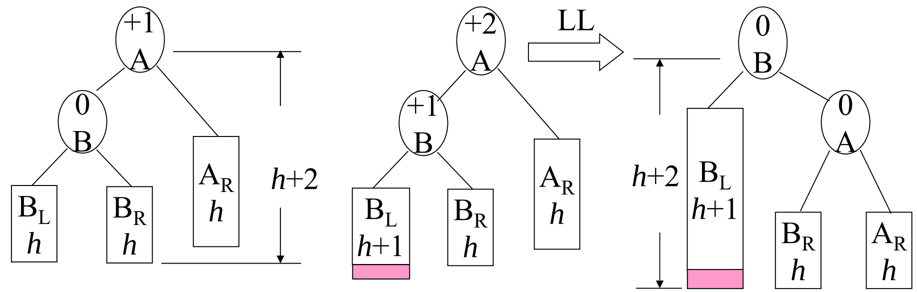
An AVL tree:



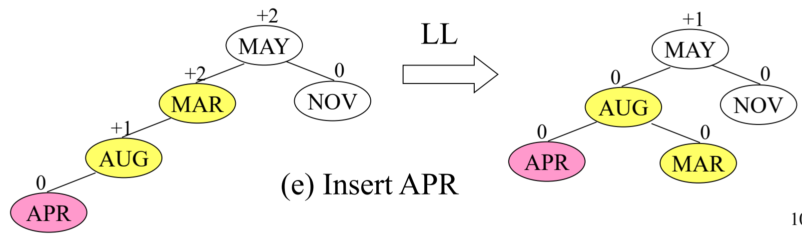
Four Kinds of Rotations in an AVL Tree

- 4 rotations for rebalancing: *LL*, *RR*, *LR*, and *RL*
- These rotations are characterized by the nearest ancestor *A* of the inserted node *Y* whose balance factor becomes ±2.
  - *LL*: insert new node *Y* in the left subtree of the left subtree of *A*.
  - *RR*: insert *Y* in the right subtree of the right subtree of *A*
  - *LR*: insert *Y* in the right subtree of the left subtree of *A*
  - *RL*: insert *Y* in the left subtree of the right subtree of *A*
- *LL* and *RR* are symmetric, called single rotations.
- *LR* and *RL* are symmetric, called double rotations.

LL Rebalancing Rotation

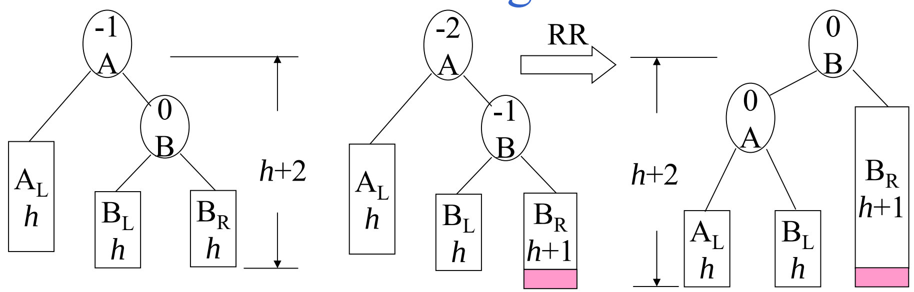


height of B_L increases to $h+1$, right rotation

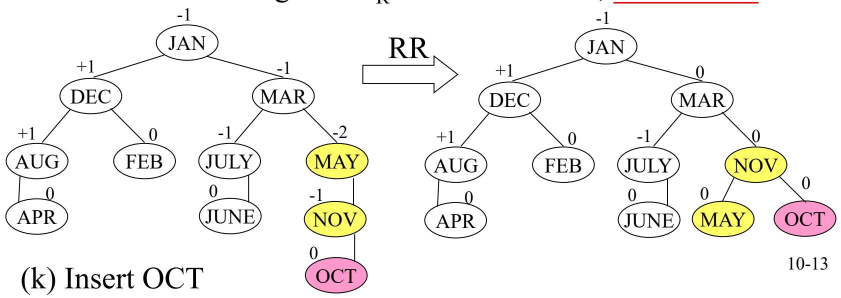


(e) Insert APR

RR Rebalancing Rotation

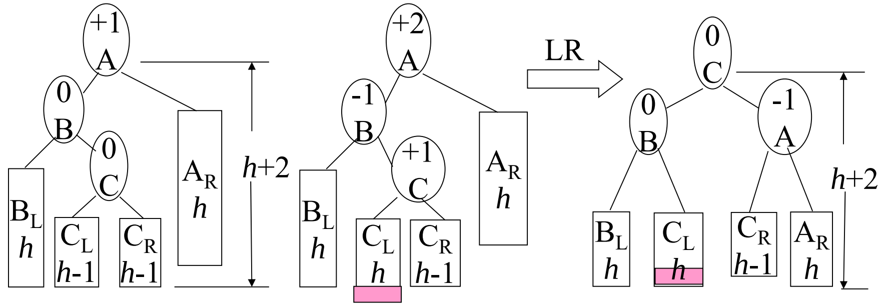


height of B_R increases to $h+1$, left rotation



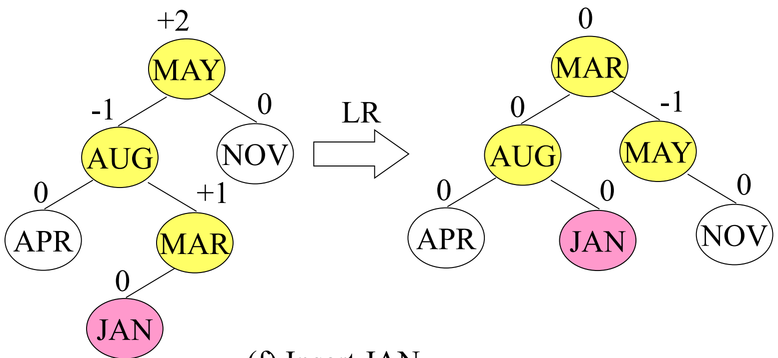
(k) Insert OCT

LR Rebalancing Rotation



- *LR* needs double rotations:
  1. Perform left rotation on the tree rooted at *B*.
  2. Perform right rotation on the tree rooted at *A*.

Example of LR



(f) Insert JAN

Complexity Comparison of Various Structures

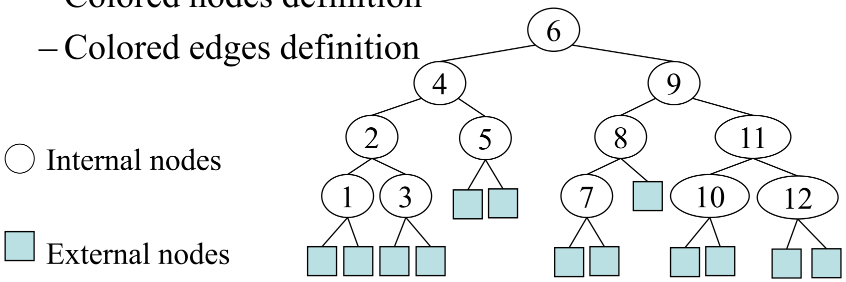
Operation	Sequential List (Sorted Array)	Linked List	AVL Tree
Search for x	$O(\log n)$	$O(n)$	$O(\log n)$
Search for kth item	$O(1)$	$O(k)$	$O(\log n)$
Delete x	$O(n)$	$O(1)$[1]	$O(\log n)$
Delete kth item	$O(n - k)$	$O(k)$	$O(\log n)$
Insert x	$O(n)$	$O(1)$[2]	$O(\log n)$
Output in order	$O(n)$	$O(n)$	$O(n)$

[1]Doubly linked list and position of x known.

[2]Position for insertion known

Red-Black Trees

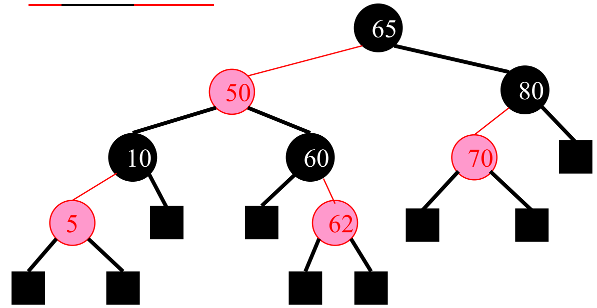
- A red-black tree is an extended binary search tree.
- Each node/pointer (edge) is colored by **red** or **black.**
  - Colored nodes definition
  - Colored edges definition



Extended binary search tree

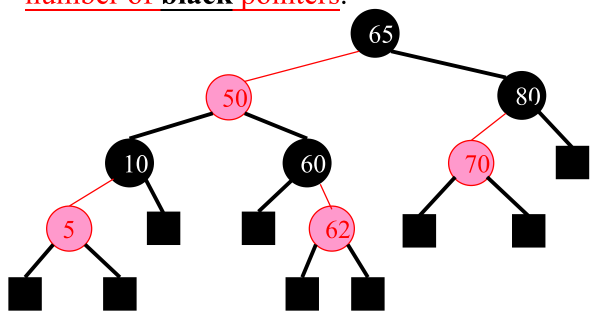
Colored Node Definition

- Colored node definition
  - **RB1**: The root and all external nodes are **black**.
  - **RB2**: No root-to-external-node path has two consecutive red nodes.
  - **RB3**: All root-to-external-node paths have the same number of **black** nodes.



Colored Pointer (Edge) Definition

- Colored pointer (edge) definition
  - **RB1’**: Pointer to an external node is **black**.
  - **RB2’**: No root-to-external-node path has two consecutive red pointers (edges).
  - **RB3’**: All root-to-external-node paths have the same number of **black** pointers.



Length and Rank in a Red-Black Tree

- Let the length of a root-to-external-node path be the number of pointers (edges) on the path.
- Let the rank of a node be the number of black pointers (edges) on any path from the node to any external node in its subtree.
- **Lemma**: P, Q: two root-to-external-node paths

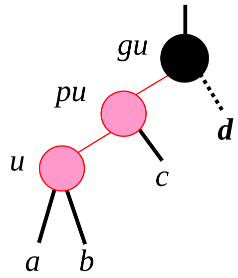
  $length(\text{P}) \leq 2 * length\ (\text{Q})$
- **Proof**: Let the rank of the root be r.
  - From **RB2’**, each red pointer is followed by a black pointer.
  - Therefore, each root-to-external-node path has between r and $2r$ pointers.

Properties of a Red-Black Tree

- **Lemma**: Let h be the height of a red-black tree (excluding the external nodes), let n be the number of internal nodes, and let r be the rank of the root.
  - (a) $h \le 2r$
  - (b) $n \ge 2^r - 1$
  - (c) $h \le 2 \log_2(n+1)$
- **Proof**:
  - (a) is correct by previous lemma.
  - From (b), we have $r \le \log_2(n+1)$. This inequality together with (a) yields (c).
- Height of a red-black tree $\le 2 \log_2(n+1)$, searching, insertion, and deletion needs $O(\log n)$ time.

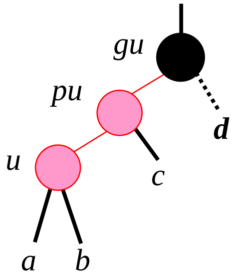
Inserting into a Red-Black Tree

- A new element u is first inserted as the ordinary binary search tree.
- Assign the new node to red.
- The new node may or may not violate **RB2** (imbalance).
  - One root-to-external-node path may have two consecutive red nodes.
  - It can be handled by changing colors or a rotation.



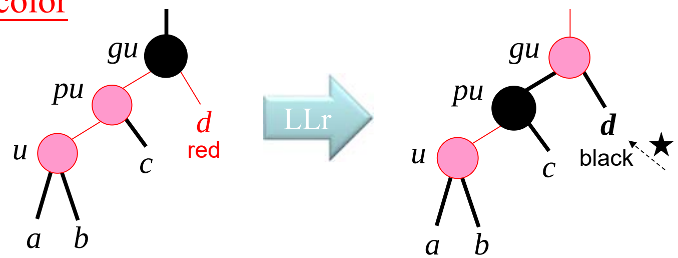
Two Consecutive Red Nodes

- u: new node, red
- pu: parent of u, red
- gu :grandparent of u, black
- **LLb, LLr**
  - Left child, then left child
  - **LLb**: the other child of gu, d, is black.
  - **LLr**: the other child of gu, d, is red.
- **LRb, LRr**:
  - Left child, then right child, black or red
- **RRb, RRr**:
  - Right child, then right child, black or red
- **RLb, RLr**:
  - Right child, then left child, black or red



Color Change of LLr, LRr, RRr, RLr

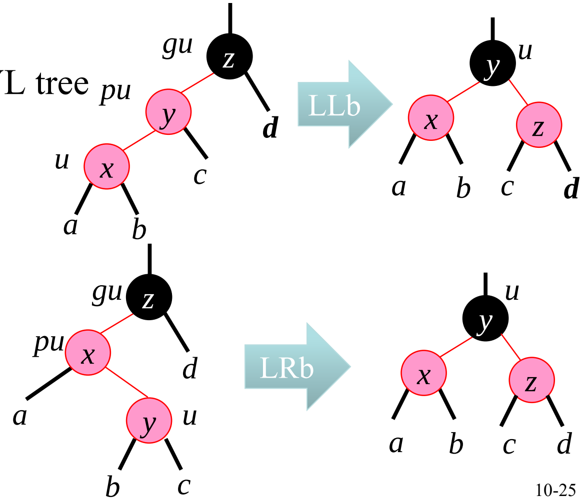
- Change color



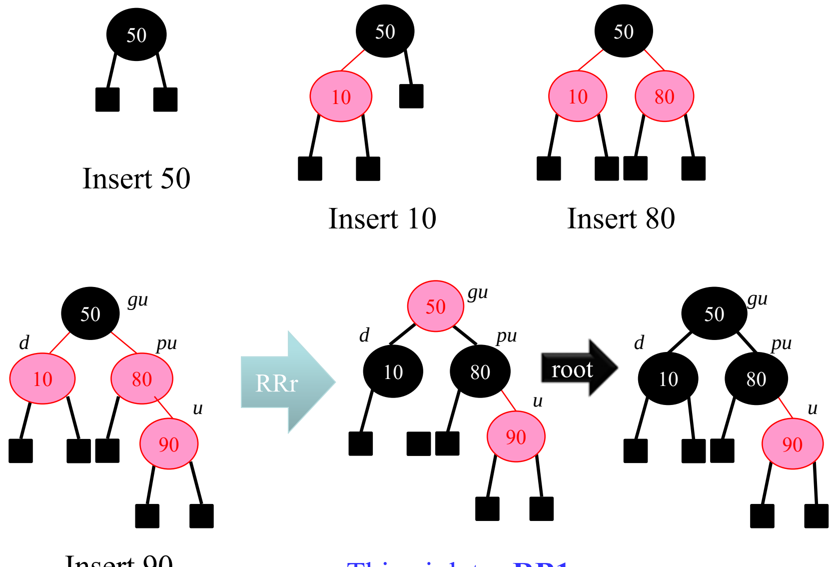
- Move u, pu, and gu up two levels.
  - gu becomes new u
- Continue rebalancing if necessary.
  - If **RB2** is satisfied, stop propagation.
  - If gu is the root, force gu to be black (The number of black nodes on all root-to-external-node paths increases by 1.)
  - Otherwise, continue color change or rotation.

Rotation and Color Change of LLb, LRb, RRb, RLb

- Same as the rotation schemes taken for an AVL tree.
- LLb rotation:
  LL rotation of AVL tree
- LRb rotation:
  LR rotation of AVL tree
- RRb is symmetric to LLb
- RLb is symmetric to LRb.

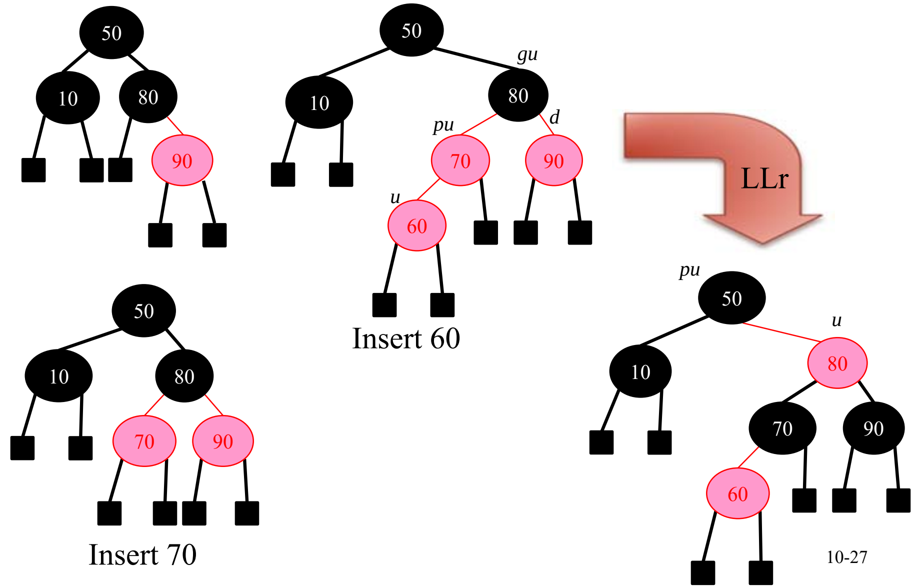


Inserting **50**, **10**, **80**, **90**, 70, 60, 65, 62

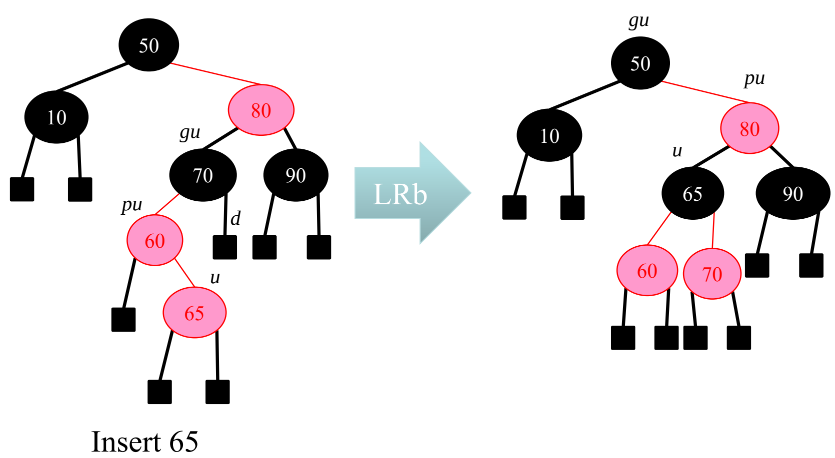


This violates **RB1**

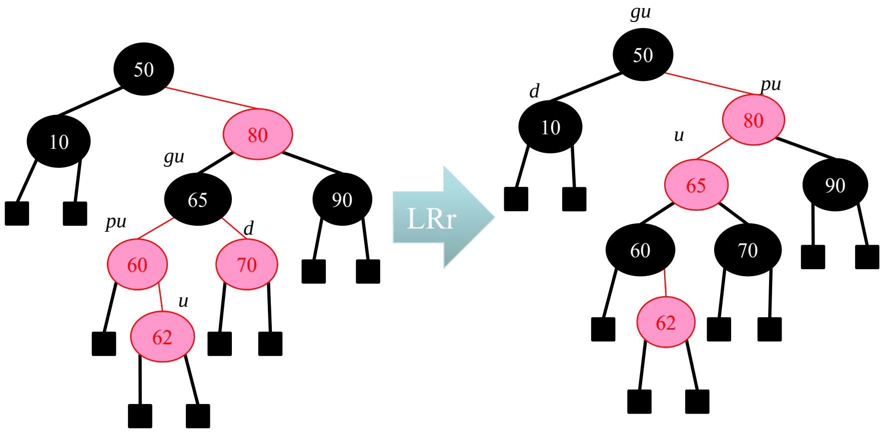
Inserting 50, 10, 80, 90, **70**, **60**, 65, 62



Inserting 50, 10, 80, 90, 70, 60, **65**, 62



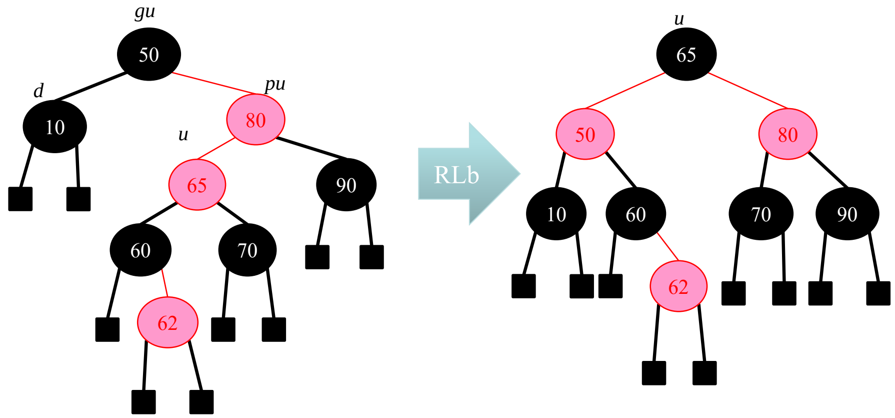
Inserting 50, 10, 80, 90, 70, 60, 65, **62**



Insert 62

Insert 62 …

Further Process for Inserting **62**



Splay Trees 斜張樹/伸展樹

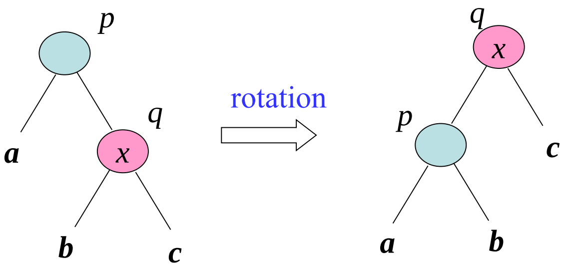
- A splay tree is a binary search tree.
- In an AVL tree, we have to store the balance factor. In a red-black tree, we have store the red/black color.
- In a splay tree, there is no balanced information.
- The operation for searching, insertion or deletion needs O(log *n*) amortized time. (worst case O(*n*).)
- Two variants
  - Bottom-up splay tree
  - Top-down splay tree

Bottom-Up Splay Trees

- Searching, insertion and deletion are performed as in an unbalanced binary search tree, then followed by a splay operation (a sequence of rotations).
- The start node *x* for the splay:
  - The searched, inserted node
  - The parent of the deleted node.
- After the splay operation completes, the splay node *x* becomes the tree root.

The Splay Operation

- *q*: the start node for the splay
- *p*: parent node of *q*
- *gp*: grandparent of *q*
- (1) If *q*=0 or *q*=root, then stop.
- (2) If there is no *gp*, then perform a rotation.



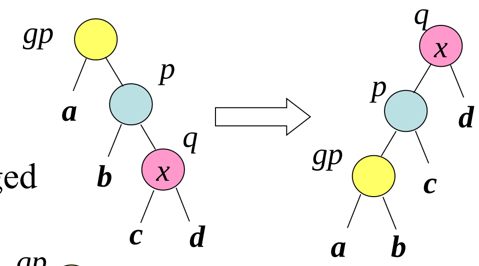
a, ***b***, and ***c*** are substrees

Rotations in the Splay Operation

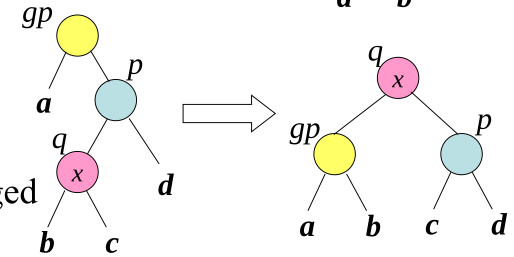
- (3) If *q* has a parent *p* and a grandparent *gp*, then a rotation is performed:
  - LL: left child, left child
  - RR: right child, right child
  - LR; left child, right child
  - RL: right child, left child
- Move up 2 levels at a time.
- The splay is repeated at the new location of *q*, until *q* becomes the root.
- LL and RR are symmetric. LR and RL are symmetric.

RR and RL Rotations

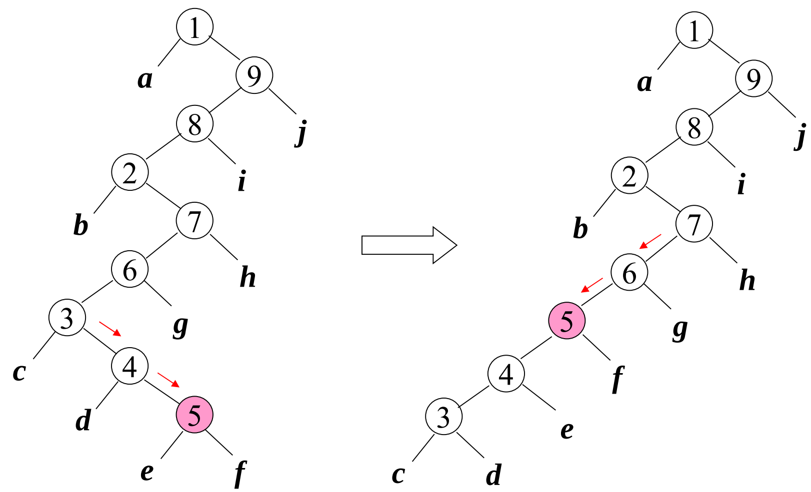
- RR rotation
  - Keep inorder sequence unchanged



- RL rotation
  - Keep inorder sequence unchanged

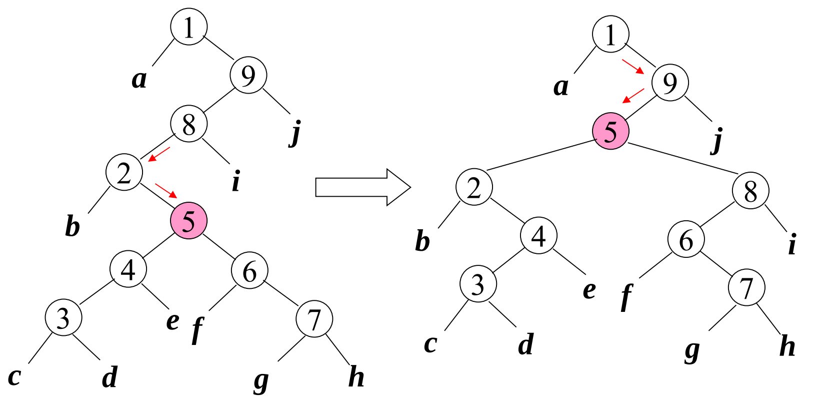


Example for the Splay Operation (1)



(a) Initial search tree, RR (b) After RR rotation, LL

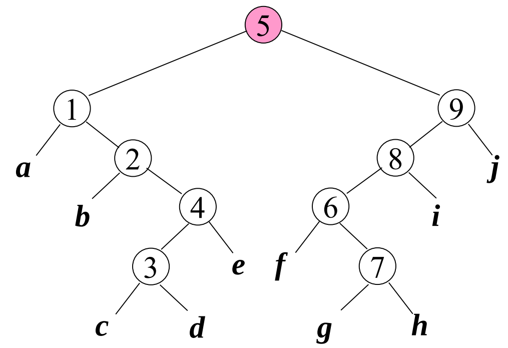
Example for the Splay Operation (2)



(c) After LL rotation, LR

(d) After LR rotation, RL

Example for the Splay Operation (3)



(e) After RL rotation

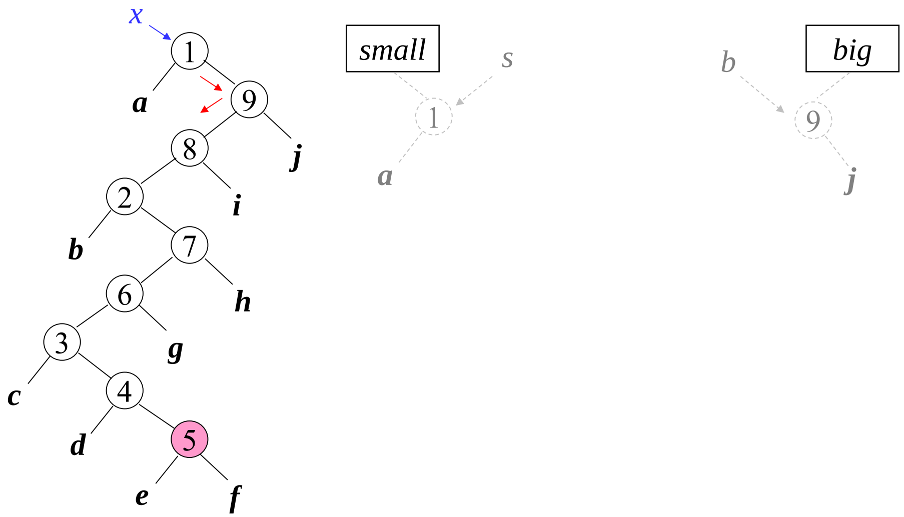
Top-Down Splay Trees (1)

- The splay node *x* (same as bottom-up splay tree):
  - The searched, inserted node
  - The parent of the deleted node.
- Following the path from the root to the splay node, partition the binary search tree into 3 components:
  - small binary search tree (smaller than *x*)
  - big binary search tree (bigger than *x*)
  - the splay node *x*

Top-Down Splay Trees (2)

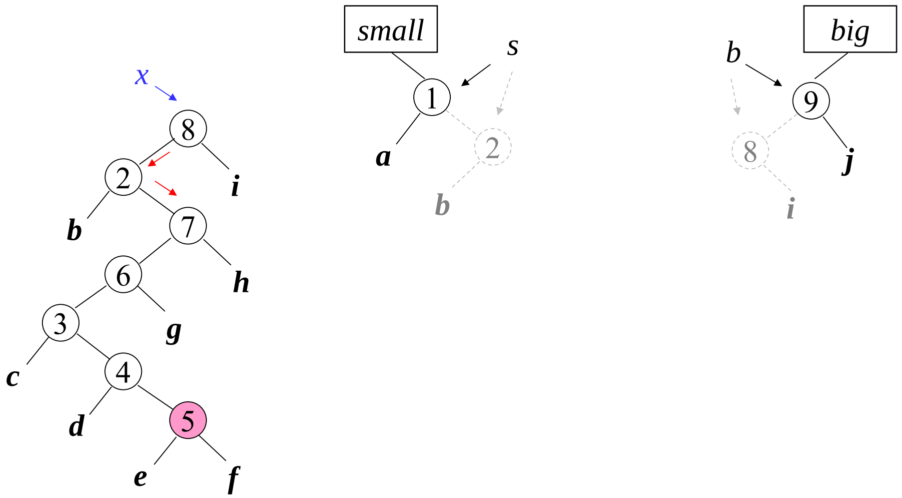
- Move down 2 levels at a time, except (possibly) that one level is moved down when the splay node is reached.
- A rotation is done whenever an LL or RR move is performed.
- When the splay node is reached, the small tree and the big tree are combined into a new binary search tree rooted at the splay node *x*.

An Example for Top-Down Splay Tree (1/7)



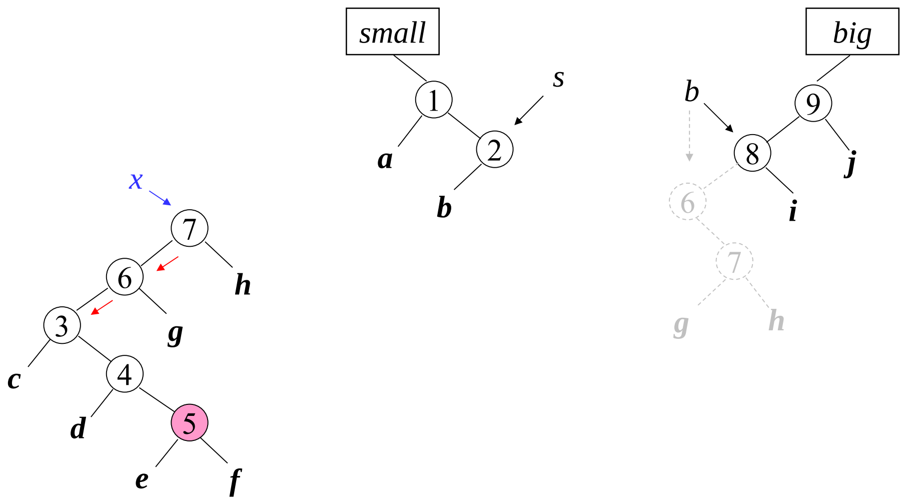
Initial search tree, RL (right subtree, then left subtree)

An Example for Top-Down Splay Tree (2/7)



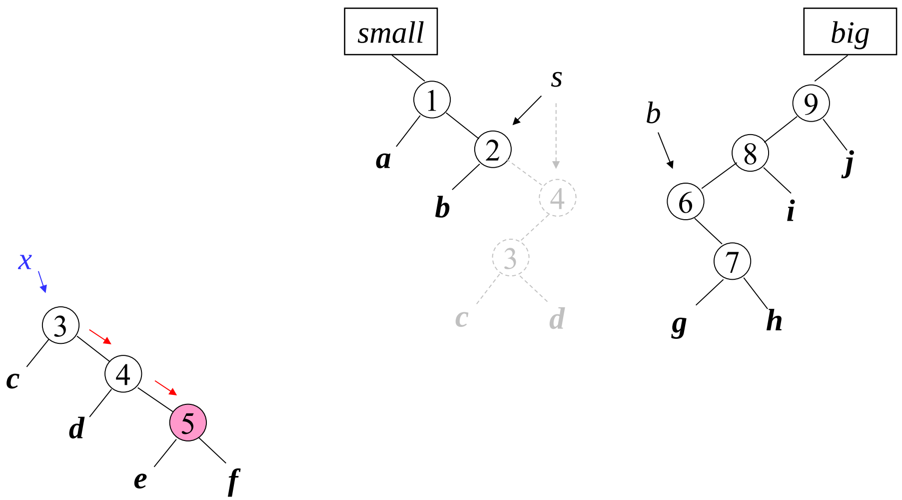
After RL transformation, LR (left, then right)

An Example for Top-Down Splay Tree (3/7)



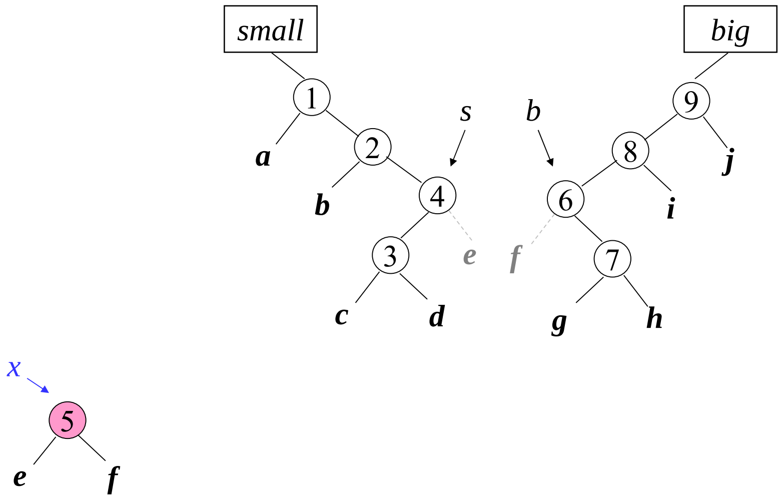
After LR transformation, LL

An Example for Top-Down Splay Tree (4/7)



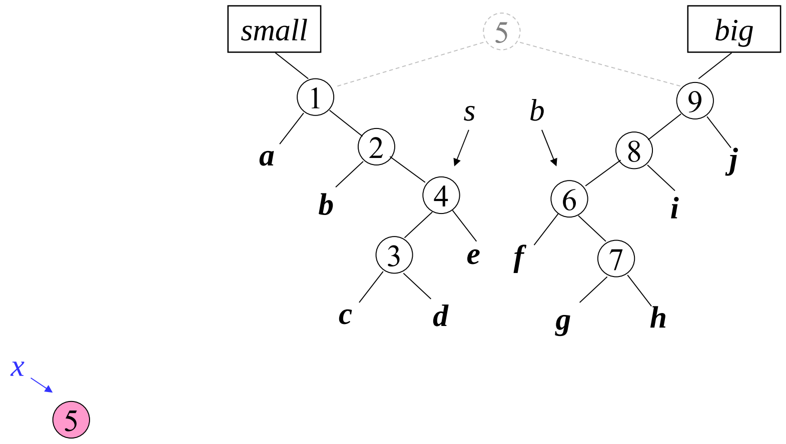
After LL transformation (a rotation), RR

An Example for Top-Down Splay Tree (5/7)



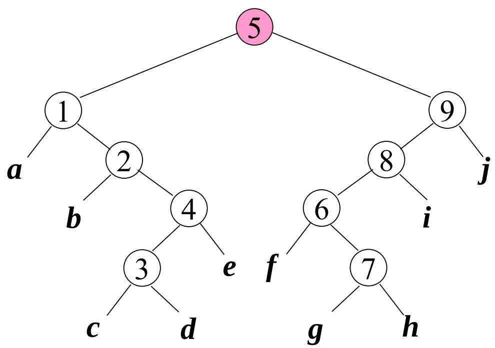
After RR transformation (a rotation), splay node is reached

An Example for Top-Down Splay Tree (6/7)



Splay node

An Example for Top-Down Splay Tree (7/7)



Final new search tree

- Bottom-Up *v.s.* Top-Down
  - Top-down splay trees are faster than bottom-up splay trees by experiments.

Data Structures

Chapter 11: Multiway Search Trees

m-way Search Trees

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 structures:

$$n, A_0, (K_1, A_1), (K_2, A_2), \ldots, (K_n, A_n)$$

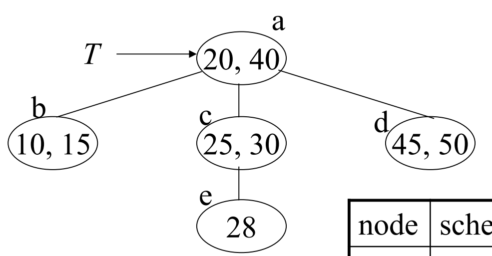
where each A_i, $0 \le i \le n \le m$, is a pointer to a subtree, and each K_i, $1 \le i \le n \le m$, is a key value.

(2) $K_i < K_{i+1}$, $1 \le i \le n\text{-}1$

(3) Let $K_0 = -\infty$ and $K_{n+1} = \infty$. All key values in the subtree A_i are less than K_{i+1} and greater than K_i, $0 \le i \le n$

(4) The subtrees A_i, $0 \le i \le n$, are also m-way search trees.

A Three-way Search Tree



node	schematic format
a	2, b, (20, c), (40, d)
b	2, 0, (10, 0), (15, 0)
c	2, 0, (25, e), (30, 0)
d	2, 0, (45, 0), (50, 0)
e	1, 0, (28, 0)

Searching an m-way Search Tree

- Suppose to search an m-way search tree T for the key value x. By searching the keys of the root, we determine i such that $K_i \le x < K_{i+1}$.
  - If $x = K_i$, the search is complete.
  - If $x \ne K_i$, x must be in a subtree A_i if x is in T.
  - We proceed to search x in subtree A_i and continue the search until we find x or determine that x is not in T.
- Maximum number of nodes in a tree of degree m and height h: $\sum_{0 \le i \le h-1} m^i = (m^h - 1)/(m-1)$
- Maximum number of keys: $m^h - 1$.

B-trees (1)

Definition: A B-tree of order m is an m-way search tree that either is empty or satisfies the following properties:

(1) $2 \le$ # children of root $\le m$

(2) All nodes other than the root node and external nodes: $\lceil m/2 \rceil \le$ # children $\le m$

(3) All external nodes are at the same level.

- 2-3 tree is a B-tree of order 3 and 2-3-4 tree is a B-tree of order 4.
- All B-trees of order 2 are full binary trees.

B-trees (2)

- N: # of keys in a B-tree of order m and height h

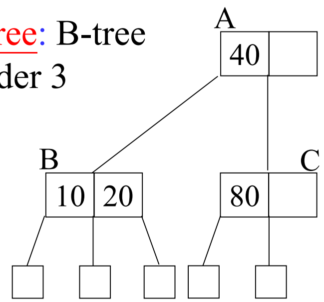
$$2\lceil m/2 \rceil^{h-1} - 1 \le N \le m^h - 1, \quad h \ge 1$$

$$h \le 1 + \log_{\lceil m/2 \rceil}[(N+1)/2]$$

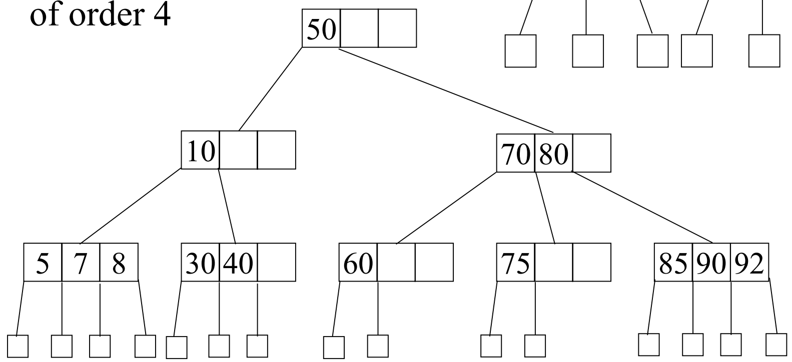
  – Reason: Root has at least 2 children, each other node has at least $\lceil m/2 \rceil$ children.

- For example, a B-tree of order m=200 with # of keys $N \le 2\times10^6 - 2$ will have $h \le 3$.

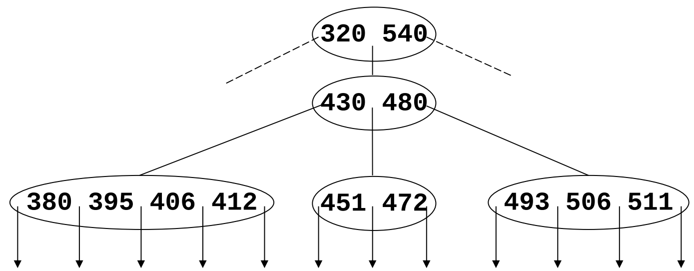
2-3 tree: B-tree of order 3



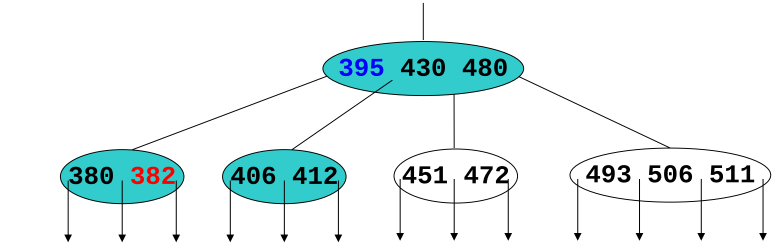
2-3-4 tree: B-tree of order 4



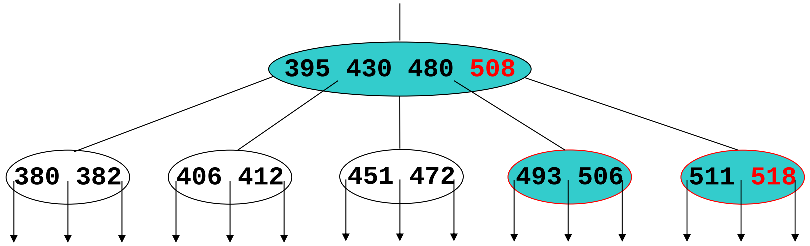
Insertion into a B-tree of Order 5



(a) Initial portion of a B-tree

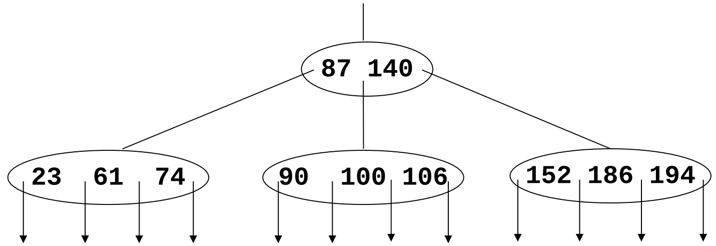


(b) After inserting 382 (Split the full node)

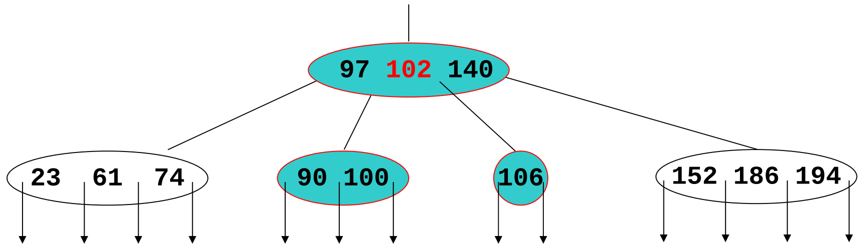


(c) After inserting 518 and 508

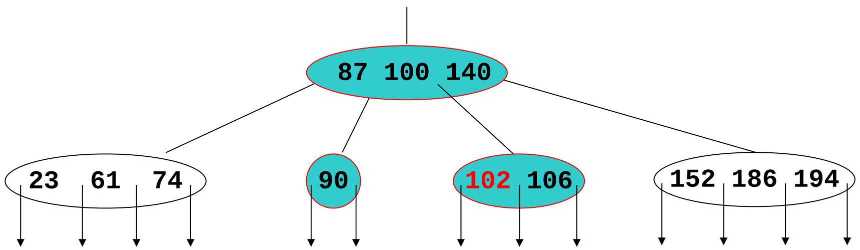
Insertion into a B-tree of Order 4 (2-3-4 Tree)



(a) An initial B-tree



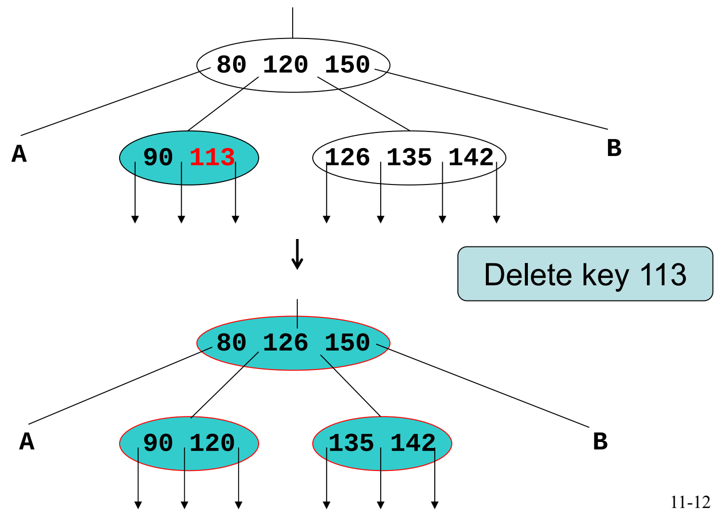
(b) Inserting 102 with a left bias



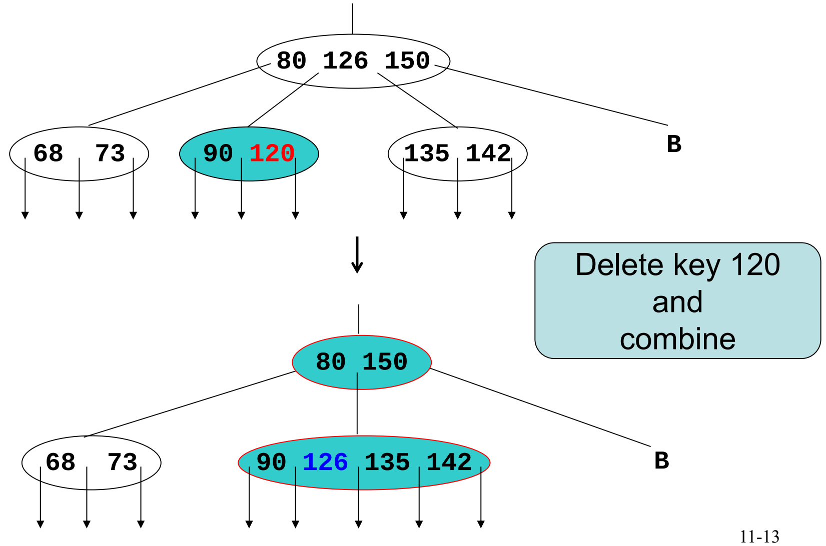
(c) Inserting 102 with a right bias

Deletion from a B-tree of Order 5

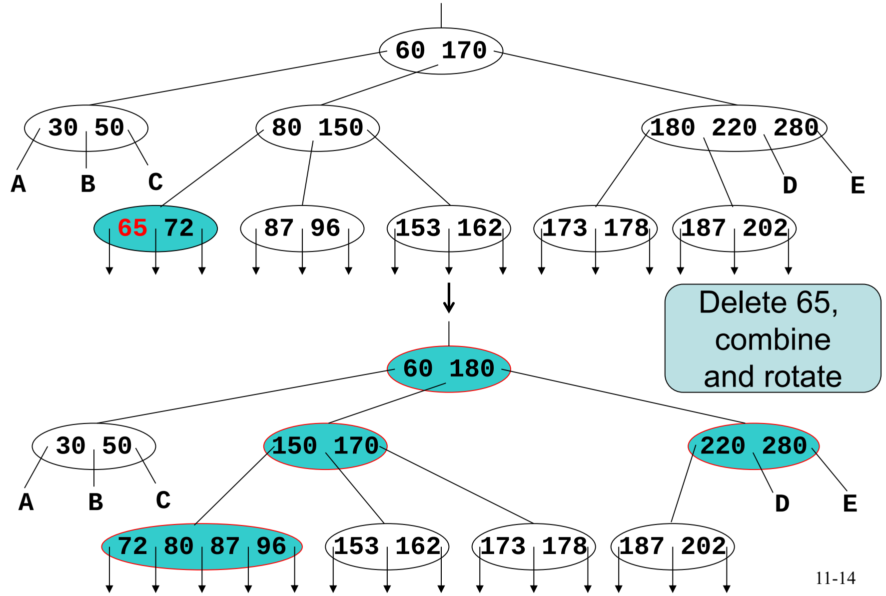
(i) Rotation: Shift a key from its parent and its sibling.



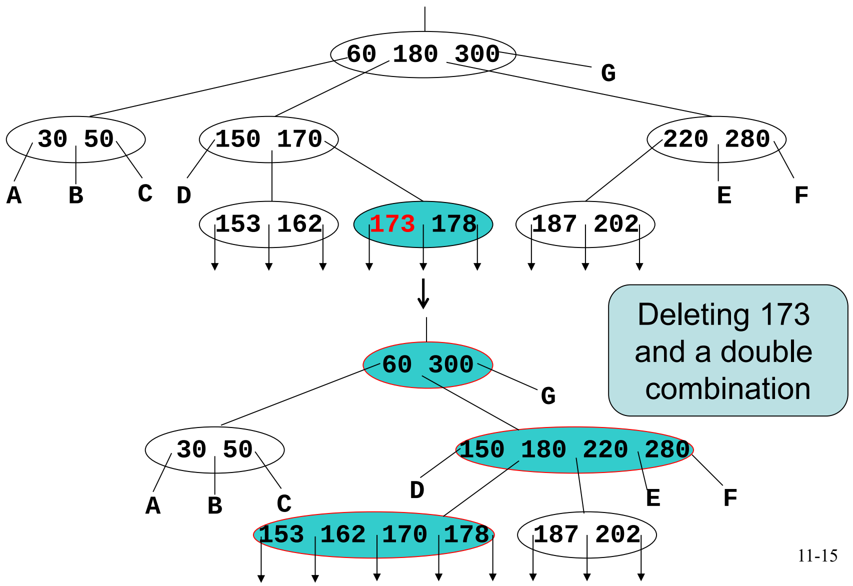
(ii) Combination: Take a key from its parent and combine with its sibling.



(iii) Combine, then rotate for its parent



(iv) Combine, then combine for its parent.



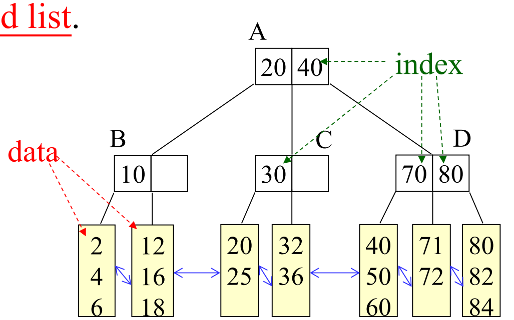
Deletion in a B-tree

- The combination process may be done up to the root.
- If the root has more than one key,
  - Done
- If the root has only one key
  - remove the root
  - The height of the tree decreases by 1.

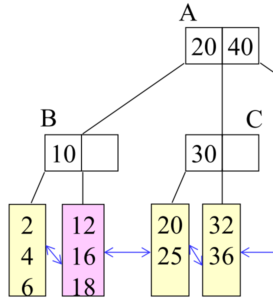
- Insertion, deletion or searching in a B-tree requires O(log$N$) time, where N denotes the number of elements in the B-tree.

B$^+$-trees

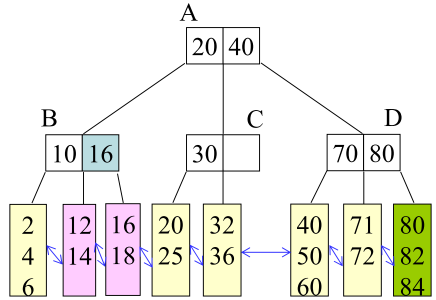
- Index node: internal node, storing keys (not elements) and pointers
- Data node: external node, storing elements (with their keys)
- Data nodes are linked together to form a doubly linked list.



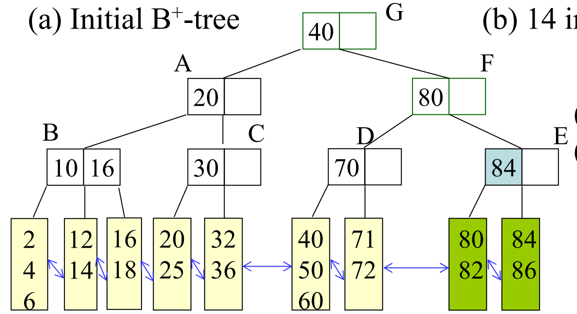
Insertion into the B$^+$-tree



(a) Initial B$^+$-tree

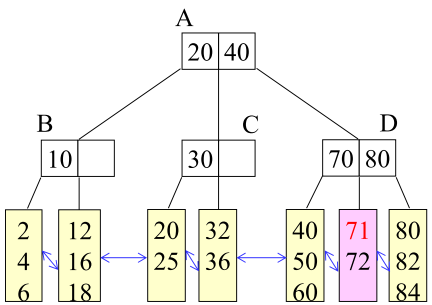


(b) 14 inserted (split)

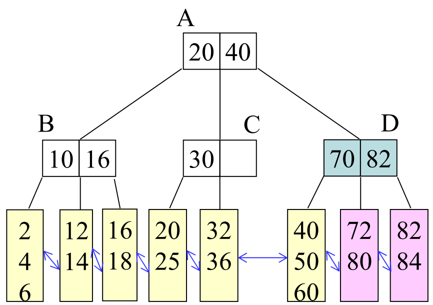


(c) 86 inserted (split)

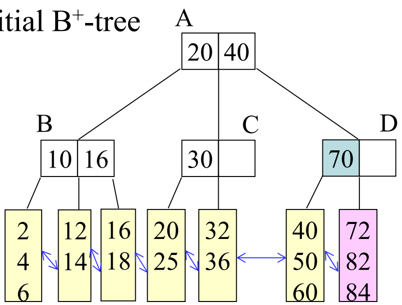
Deletion from a B$^+$-tree



(a) Initial B$^+$-tree

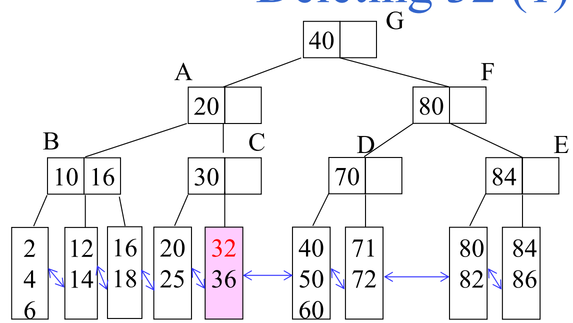


(b) 71 deleted (borrow)

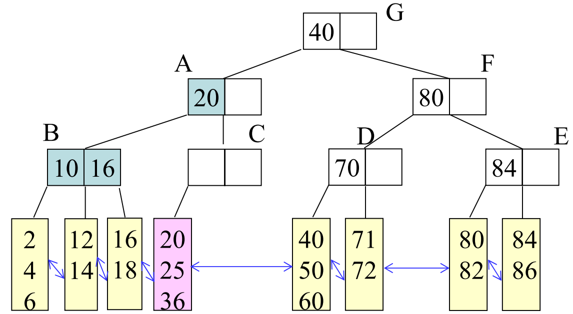


(c) 80 deleted (combine)

Deleting 32 (1)

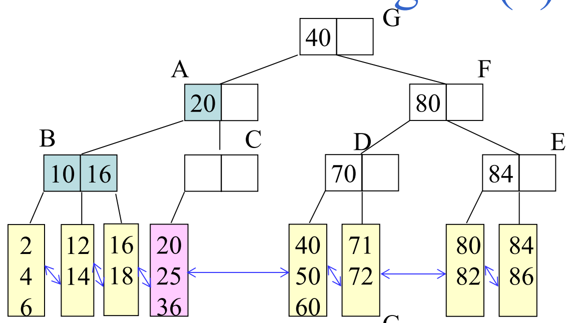


(a) Initial B$^+$-tree

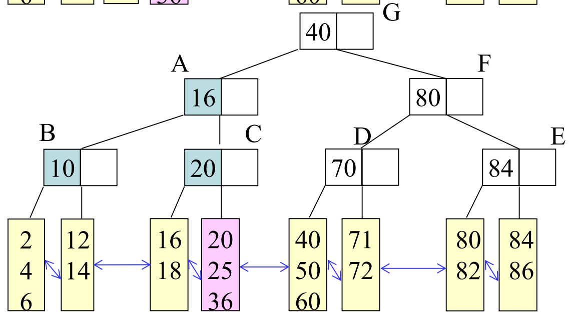


(b) C is deficient

Deleting 32 (2)

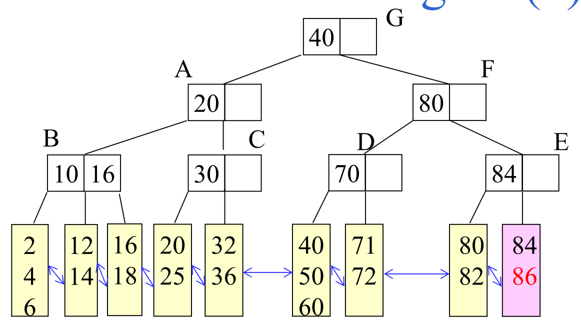


(b) C is deficient

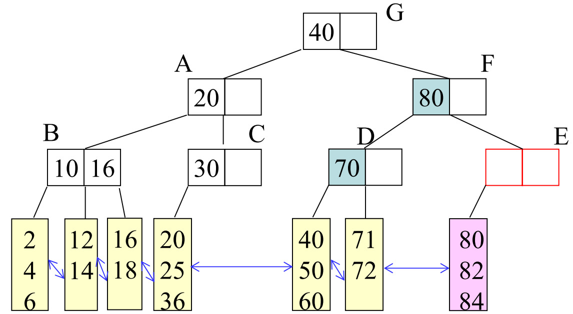


(c) After borrowing from B

Deleting 86 (1)

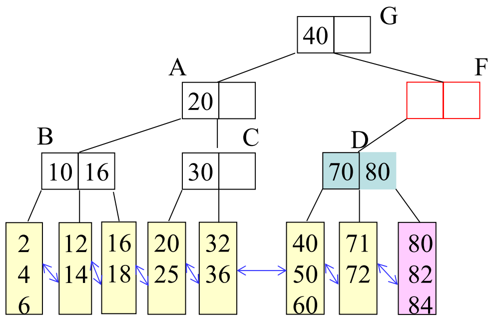


(a) Initial B+-tree

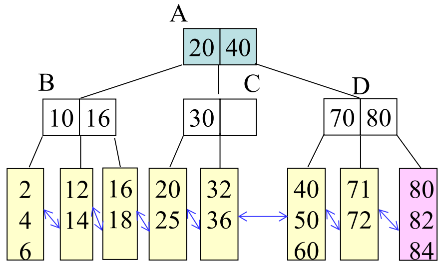


(b) E becomes deficient (combine)

Deleting 86 (2)



(c) F becomes deficient (combine)



(d) After combining A,G,F