

Foundation of Data Structure

by OE.Heart

1 Algorithm Analysis

[Definition] An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria.

1. **Input** : There are zero or more quantities that are externally supplied.
 2. **Output** : At least one quantity is produced.
 3. **Definiteness** : Each instruction is clear and unambiguous.
 4. **Finiteness** : the algorithm terminates after finite number of steps
 5. **Effectiveness** : basic enough to be carried out ; feasible
- A program does not have to be finite. (eg. an operation system)
 - An algorithm can be described by human languages, flow charts, some programming languages, or pseudocode.

[Example] Selection Sort : Sort a set of $n \geq 1$ integers in increasing order

```
1  for (i = 0; i < n; i++){
2      Examine list[i] to list[n-1] and suppose that the smallest
      integer is at list[min];
3      Interchange list[i] and list[min];
4  }
```

1.1 What to Analyze

- Machine and compiler-dependent run times.
- Time and space complexities : machine and compiler independent.
- Assumptions:

- 1. instructions are executed sequentially 顺序执行
- 2. each instruction is simple, and takes exactly one time unit
- 3. integer size is fixed and we have infinite memory

- $T_{avg}(N)$ and $T_{worst}(N)$: the average and worst case time complexities as functions of input size N

[Example] Matrix addition

```

1 void add(int a[][MAX_SIZE],
2         int b[][MAX_SIZE],
3         int c[][MAX_SIZE],
4         int rows, int cols)
5 {
6     int i, j;
7     for (i=0; i<rows; i++)/*rows+1*/
8         for (j=0;j<cols;j++)/*rows(cols+1)*/
9             c[i][j] = a[i][j]+b[i][j];/*rows*cols*/
10 }

```

$$T(rows, cols) = 2rows \times cols + 2rows + 1$$

- 非对称

[Example] Iterative function for summing a list of numbers

```

1 float sum (float list[], int n)
2 { /*add a list of numbers*/
3     float tempsum = 0; /*count = 1*/
4     int i;
5     for (i=0; i<n; i++)
6         /*count++*/
7         tempsum += list[i]; /*count++*/
8     /*count++ for last excutaion of for*/
9     return tempsum; /*count++*/
10 }

```

$$T_{sum}(n) = 2n + 3$$

[Example] Recursive function for summing a list of numbers

```

1 float rsum (float list[], int n)
2 { /*add a list of numbers*/
3     if (n) /*count++*/
4         return rsum(list, n-1) + list[n-1];
5     /*count++*/
6     return 0; /*count++*/
7 }

```

$$T_{rsum}(n) = 2n + 2$$

But it takes more time to compute each step.

1.2 Asymptotic Notation(O , Ω , Θ , o)

- predict the growth ; compare the time complexities of two programs ; asymptotic(渐进的) behavior

[Definition] $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq c \cdot f(N)$ for all $N \geq n_0$. (**upper bound**)

[Definition] $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq c \cdot f(N)$ for all $N \geq n_0$. (**lower bound**)

[Definition] $T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

[Definition] $T(N) = o(p(N))$ if $T(n) = O(p(N))$ and $T(N) \neq \Theta(p(N))$.

- $2N + 3 = O(N) = O(N^{k \geq 1}) = O(2^N) = \dots$ take the **smallest** $f(N)$
- $2^N + N^2 = \Omega(2^N) = \Omega(N^2) = \Omega(N) = \Omega(1) = \dots$ take the **largest** $g(N)$
- Rules of Asymptotic Notation

1. If $T_1(N) = O(f(N))$ and $T_2 = O(g(N))$, then

(1) $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$

(2) $T_1(N) * T_2(N) = O(f(N) * g(N))$

2. 若 $T(N)$ 是一个 k 次多项式, 则 $T(N) = \Theta(N^k)$

3. $\log_k N = O(N)$ for any constant k (**logarithms grow very slowly**)

		Input size n					
Time	Name	1	2	4	8	16	32
1	constant	1	1	1	1	1	1
$\log n$	logarithmic	0	1	2	3	4	5
n	linear	1	2	4	8	16	32
$n \log n$	log linear	0	2	8	24	64	160
n^2	quadratic	1	4	16	64	256	1024
n^3	cubic	1	8	64	512	4096	32768
2^n	exponential	2	4	16	256	65536	4294967296
$n!$	factorial	1	2	24	40326	2092278988000	26313×10^{33}

Time for $f(n)$ instructions on a 10^9 instr/sec computer							
n	$f(n)=n$	$\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μ s	.03 μ s	.1 μ s	1 μ s	10 μ s	10sec	1 μ s
20	.02 μ s	.09 μ s	.4 μ s	8 μ s	160 μ s	2.84hr	1ms
30	.03 μ s	.15 μ s	.9 μ s	27 μ s	810 μ s	6.83d	1sec
40	.04 μ s	.21 μ s	1.6 μ s	64 μ s	2.56ms	121.36d	18.3min
50	.05 μ s	.28 μ s	2.5 μ s	125 μ s	6.25ms	3.1yr	13d
100	.10 μ s	.66 μ s	10 μ s	1ms	100ms	3171yr	4×10^{13} yr
1,000	1.00 μ s	9.96 μ s	1ms	1sec	16.67min	3.17×10^{13} yr	32×10^{283} yr
10,000	10 μ s	130.03 μ s	100ms	16.67min	115.7d	3.17×10^{23} yr	
100,000	100 μ s	1.66ms	10sec	11.57d	3171yr	3.17×10^{33} yr	
1,000,000	1.0ms	19.92ms	16.67min	31.71yr	3.17×10^7 yr	3.17×10^{43} yr	

[Example] Matrix addition

```

1 void add(int a[][MAX_SIZE],
2          int b[][MAX_SIZE],
3          int c[][MAX_SIZE],
4          int rows, int cols)
5 {
6     int i, j;
7     for (i=0; i<rows; i++)
8         for (j=0; j<cols; j++)
9             c[i][j] = a[i][j]+b[i][j];
10 }
```

$$T(\text{rows}, \text{cols}) = \Theta(\text{rows} \cdot \text{cols})$$

General Rules

- **For loops** : The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.
- **Nested for loops** : The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the for loops.
- **Consecutive statements** : These just add (which means that the maximum is the one that counts).
- **If/else** : For the fragment
if (Condition) S1;
else S2;

The running time is never more than the running time of the test plus the larger of the running time of S1 and S2.

1-5 For the following piece of code

(3分)

```
if ( A > B ){  
    for ( i=0; i<N*2; i++ )  
        for ( j=N*N; j>i; j-- )  
            C += A;  
}  
else {  
    for ( i=0; i<N*N/100; i++ )  
        for ( j=N; j>i; j-- )  
            for ( k=0; k<N*3; k++)  
                C += B;  
}
```

the lowest upper bound of the time complexity is $O(N^3)$.

☒ T ☐ F

- **Recursions** :

[Example] Fibonacci number

$$Fib(0) = Fib(1) = 1, Fib(n) = Fib(n-1) + Fib(n-2)$$

```

1  long int Fib (int N) /*T(N)*/
2  {
3      if (N<=1) /*O(1)*/
4          return 1; /*O(1)*/
5      else
6          return Fib(N-1)+Fib(N-2);
7  } /*O(1)*//*T(N-1)*//*T(N-2)*/

```

$$T(N) = T(N-1) + T(N-2) + 2 \geq Fib(N)$$

$$\left(\frac{3}{2}\right)^n \leq Fib(N) \leq \left(\frac{5}{3}\right)^n$$

时间复杂度: $O(2^N)$ $T(N)$ grows **exponentially**

空间复杂度: $O(N)$

1.3 Compare the Algorithms

[Example] 最大子序列和

Algorithm 1

```

1  int MaxSubsequenceSum ( const int A[ ], int N )
2  {
3      int ThisSum, MaxSum, i, j, k;
4      MaxSum = 0; /* initialize the maximum sum */
5      for( i = 0; i < N; i++ ) /* start from A[ i ] */
6          for( j = i; j < N; j++ ) { /* end at A[ j ] */
7              ThisSum = 0;
8              for( k = i; k <= j; k++ )
9                  ThisSum += A[ k ]; /* sum from A[ i ] to A[
10 j ] */
11              if ( ThisSum > MaxSum )
12                  MaxSum = ThisSum; /* update max sum */
13          } /* end for-j and for-i */
14      return MaxSum;
15  }

```

$$T(N) = O(N^3)$$

Algorithm 2

```

1  int MaxSubsequenceSum ( const int A[ ], int N )
2  {
3      int ThisSum, MaxSum, i, j;

```

```

4      MaxSum = 0;    /* initialize the maximum sum */
5      for( i = 0; i < N; i++ ) {    /* start from A[ i ] */
6          ThisSum = 0;
7          for( j = i; j < N; j++ ) {    /* end at A[ j ] */
8              ThisSum += A[ j ];    /* sum from A[ i ] to A[ j ]
          */
9              if ( ThisSum > MaxSum )
10                 MaxSum = ThisSum;    /* update max sum */
11          }    /* end for-j */
12      }    /* end for-i */
13      return MaxSum;
14  }

```

$$T(N) = O(N^2)$$

Algorithm 3 Divide and Conquer 分治法

```

1  static int MaxSubSum(const int A[ ], int Left, int Right)
2  {
3      int MaxLeftSum, MaxRightSum;
4      int MaxLeftBorderSum, MaxRightBorderSum;
5      int LeftBorderSum, RightBorderSum;
6      int Center, i;
7
8      if (Left == Right)
9          if (A[Left] > 0)
10             return A[Left];
11         else
12             return 0;
13
14     Center = (Left + Right) / 2;
15     MaxLeftSum = MaxSubSum(A, Left, Center);
16     MaxRightSum = MaxSubSum(A, Center + 1, Right);
17
18     MaxLeftBorderSum = 0;
19     LeftBorderSum = 0;
20     for (i = Center; i >= Left; i--)
21     {
22         LeftBorderSum += A[i];
23         if (LeftBorderSum > MaxLeftBorderSum)
24             MaxLeftBorderSum = LeftBorderSum;
25     }
26
27     MaxRightBorderSum = 0;
28     RightBorderSum = 0;

```

```

29     for (i = Center+1; i <= Right; i++)
30     {
31         RightBorderSum += A[i];
32         if (RightBorderSum > MaxRightBorderSum)
33             MaxRightBorderSum = RightBorderSum;
34     }
35
36     return Max3(MaxLeftSum, MaxRightSum, MaxLeftBorderSum +
MaxRightBorderSum);
37 }
38
39 int MaxSubsequenceSum(const int A[ ], int N)
40 {
41     return MaxSubSum(A, 0, N - 1);
42 }

```

$$\begin{aligned}
 \therefore T(N) &= 2T\left(\frac{N}{2}\right) + cN \quad T(1) = O(1) \\
 T\left(\frac{N}{2}\right) &= 2T\left(\frac{N}{2^2}\right) + c\frac{N}{2} \\
 &\dots \\
 T(1) &= 2T\left(\frac{N}{2^k}\right) + c\frac{N}{2^{k-1}} \\
 \therefore T(N) &= 2^k T\left(\frac{N}{2^k}\right) + kcN = N \cdot O(1) + cN \log N
 \end{aligned}$$

Algorithm 4 On-line Algorithm 在线算法

```

1  int MaxSubsequenceSum( const int  A[ ],  int  N )
2  {
3      int ThisSum, MaxSum, j;
4      ThisSum = MaxSum = 0;
5      for ( j = 0; j < N; j++ ) {
6          ThisSum += A[ j ];
7          if ( ThisSum > MaxSum )
8              MaxSum = ThisSum;
9          else if ( ThisSum < 0 )
10             ThisSum = 0;
11     } /* end for-j */
12     return MaxSum;
13 }

```

$$T(N) = O(N)$$

- A[] is scanned **once** only. 扫描一次, 无需存储 (处理streaming data)

- 在任意时刻，算法都能对它已经读入的数据给出子序列问题的正确答案(其他算法不具有这个特性)

1.4 Logrithms in the Running Time

- 如果一个算法用常数时间将问题的大小削减为其一部分(通常是1/2)，那么该算法就是 $O(\log N)$ 的

[Example] Binary Search

```
1  int BinarySearch ( const ElementType A[ ], ElementType X, int
    N )
2  {
3      int Low, Mid, High;
4      Low = 0; High = N - 1;
5      while ( Low <= High ) {
6          Mid = ( Low + High ) / 2;
7          if ( A[ Mid ] < X )
8              Low = Mid + 1;
9          else
10             if ( A[ Mid ] > X )
11                 High = Mid - 1;
12             else
13                 return Mid; /* Found */
14     } /* end while */
15     return NotFound; /* NotFound is defined as -1 */
16 }
```

$$T_{worst}(N) = O(\log N)$$

[Example] Euclid's Algorithm

```
1  int Gcd(int M, int N)
2  {
3      int Rem;
4
5      while (N > 0)
6      {
7          Rem = M % N;
8          M = N;
9          N = Rem;
10     }
11     return M;
12 }
```

[Example] Efficient exponentiation

```
1 long int Pow(long int X, int N)
2 {
3     if (N == 0) return 1;
4     if (N == 1) return X;
5     if (IsEven(N)) return Pow(X*X, N/2); /*return Pow(X,
N/2)*Pow(X, N/2) affects the efficiency*/
6     else return Pow(X*X, N/2)*X; /*return Pow(X, N-1)*X is the
same*/
7 }
```

1.5 Checking Your Analysis

Method 1

When $T(N) = O(N)$, check if $T(2N)/T(N) \approx 2$

When $T(N) = O(N^2)$, check if $T(2N)/T(N) \approx 4$

When $T(N) = O(N^3)$, check if $T(2N)/T(N) \approx 8$

Method 2

When $T(N) = O(f(N))$, check if $\lim_{N \rightarrow \infty} \frac{T(N)}{f(N)} \approx C$

2 List, Stacks and Queues

2.1 Abstract Data Type(ADT) 抽象数据类型

[Definition] Data Type = {Objects} and {Operations}

[Definition] An Abstract Data Type(ADT) is a data type that is organized in such a way that the *specification* on the objects and *specification* of the operations on the objects are *separated from the representation* of the objects and the *implementation* on the operations.

2.2 The List ADT

- **Objects** : N items
- **Operations**
 - Finding the length
 - Printing
 - Making an empty
 - Finding
 - Inserting
 - Deleting
 - Finding next
 - Finding previous

Simple Array implementation of Lists

- Sequential mapping 连续存储，访问快
- Find_Kth take $O(1)$ time.
- MaxSize has to be estimated.
- Insertion and Deletion not only take $O(N)$ times, but also involve a lot of data movements which takes time.

1-3 For a sequentially stored linear list of length N , the time complexities for query and insertion are $O(1)$ and $O(N)$, respectively. (3分)

☐ T ☒ F

1-3 答案错误 ① (0分) [创建提问](#)

Query 查询

Linked Lists

- Location of nodes may change on different runs.
- Insertion 先连后断
- Deletion 先连后释放
- 频繁malloc和free系统开销较大
- Finding take $O(N)$ times.

```

1  /*Return true if L is empty*/
2  int IsEmpty(List L)
3  {
4      return L->Next == NULL;
5  }

```

```

1  /*Return true if P is the last position in list L*/
2  /*Parameter L is unused in this implementation*/
3  int IsLast(Position P, List L)
4  {
5      return P->Next == NULL;
6  }

```

```

1  /*Return Position of X in L; NULL if not found*/
2  Position Find(Element X, List L)
3  {
4      Position P;
5
6      P = L->Next;
7      while (P != NULL && P->Element != X) P = P->Next;
8
9      return P;
10 }

```

```

1  /*Delete first occurrence of X from a list*/
2  /*Assume use of a header node*/
3  void Delete(ElementType X, List L)
4  {
5      Position P, TmpCell;
6
7      P = FindPrevious(X, L);
8
9      if (!IsLast(P, L))
10     {
11         TmpCell = P->Next;
12         P->Next = TmpCell->Next;
13         free(TmpCell);
14     }
15 }

```

```

1  /*If x is not found, then Next field of returned*/
2  /*Assumes a header*/
3  Position FindPrevious(ElementType X, List L)
4  {
5      Position P;
6
7      P = L;
8      while (P->Next != NULL && P->Next->Element != X) P =
P->Next;
9
10     return P;
11 }

```

```

1  /*Insert (after legal position P)*/
2  /*Header implementation assumed*/
3  /*Parameter L is unused in this implementation*/
4  void Insert(ElementType X, List L, Position P)
5  {
6      Position TmpCell;
7
8      TmpCell = malloc(sizeof(struct Node));
9      if (TmpCell == NULL) FatalError("Out of space!")
10
11     TmpCell->Element = X;
12     TmpCell->Next = P->Next;
13     P->Next = TmpCell;
14 }

```

```

1  void DeleteList(List L)
2  {
3      Position P, Tmp;
4
5      P = L->Next;
6      L->Next = NULL;
7      while (P != NULL)
8      {
9          Tmp = P->Next;
10         free(P);
11         P = Tmp;
12     }
13 }

```

Doubly Linked Circular Lists

- Finding take $O(\frac{N}{2})$ times.

2-2 If the most commonly used operations are to visit a random position and to insert and delete the last element in a linear list, then which of the following data structures is the most efficient? (2分)

- ☐ A. doubly linked list
- ☐ B. singly linked circular list
- ☒ C. doubly linked circular list with a dummy head node
- ☐ D. sequential list

2-2 答案错误 ⓘ (0 分)  创建提问

The correct answer is D.

Two Applications

1. The Polynomial ADT

- Objects :
- Operations :
 - Finding degree
 - Addition
 - Subtraction
 - Multiplication
 - Differentiation
- [Representation 1]

```
1 typedef struct {
2     int CoeffArray [ MaxDegree + 1 ] ;
3     int HighPower;
4 } *Polynomial ;
```

```

1  /*将多项式初始化为零*/
2  void ZeroPolynomial(Polynomial Poly)
3  {
4      int i;
5      for(i = 0; i <= MaxDegree; i++)
6          Poly->CoeffArray[ i ] = 0;
7      Poly->HighPower = 0;
8  }

```

```

1  /*两个多项式相加*/
2  void AddPolynomial(const Polynomial Poly1, const
Polynomial Poly2, Polynomial PolySum)
3  {
4      int i;
5
6      ZeroPolynomial(PolySum);
7      PolySum->HighPower = Max(Poly1->HighPower, Poly2-
>HighPower);
8
9      for (i = PolySum->HighPower; i >= 0; i--)
10         PolySum->CoeffArray[ i ] = Poly1->CoeffArray[ i ]
+ Poly2->CoeffArray[ i ];
11 }

```

```

1  void MultPolynomial(const Polynomial Poly1, const
Polynomial Poly2, Polynomial PolyProd)
2  {
3      int i, j;
4
5      ZeroPolynomial (PolyProd);
6      PolyProd->HighPower = Poly1->HighPower + Poly2-
>HighPower;
7
8      if(PolyProd->HighPower > MaxDegree)
9          Error("Exceeded array size");
10     else
11         for(i = 0; i <= Poly1->HighPower; i++)
12             for(j = 0; j <= Poly2->HighPower; j++)
13                 PolyProd->CoeffArray[ i + j ] += Poly1-
>CoeffArray[ i ] * Poly2->CoeffArray[ j ];
14 }

```

- [Representation 2]

```

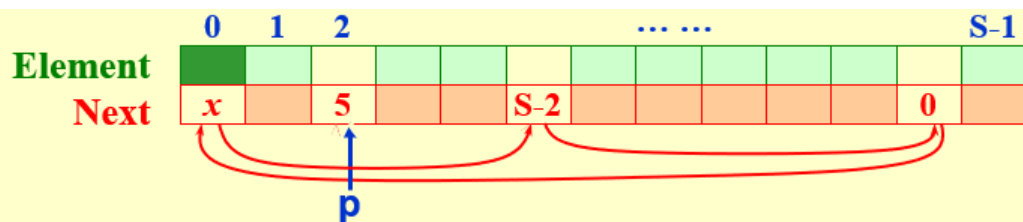
1 typedef struct poly_node *poly_ptr;
2 struct poly_node{
3     int Coefficient; /* assume coefficients are integers
   */
4     int Exponent;
5     poly_ptr Next;
6 };
7 typedef poly_ptr a; /* nodes sorted by exponent */

```

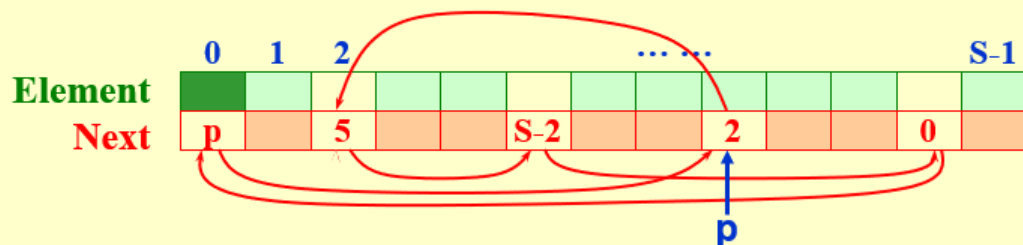
- 只存储非零项

2. Multilists

Cursor Implementation of Linked Lists(no pointer)



malloc: `p = CursorSpace[0].Next ;`
`CursorSpace[0].Next = CursorSpace[p].Next ;`




free(p): `CursorSpace[p].Next = CursorSpace[0].Next ;`
`CursorSpace[0].Next = p ;`


2.3 The Stack ADT


- Last-In-First-Out (LIFO)
- **Objects** : A finite ordered list with zero or more elements.
- **Operations** :
 - IsEmpty
 - CreatStack
 - DisposeStack
 - MakeEmpty
 - Push

- Top
- Pop
- A Pop(or Top) on an empty stack is an error in the stack ADT.
- Push on a full stack is an implementation error but not an ADT error.

Linked List Implementation (with a header node)

 **Push:** ① **TmpCell->Next = S->Next**
 ② **S->Next = TmpCell**

 **Top:** **return S->Next->Element**

 **Pop:** ① **FirstCell = S->Next**
 ② **S->Next = S->Next->Next**
 ③ **free (FirstCell)**

- The calls to malloc and free are expensive. Simply keep another stack as a recycle bin.

```
1 int IsEmpty(Stack S)
2 {
3     return S->Next == NULL;
4 }
```

```
1 Stack CreateStack(void)
2 {
3     Stack S;
4     S = malloc(sizeof(struct Node));
5     if (S == NULL)
6         Fatal Error("Out of space!");
7     S->Next == NULL;
8     MakeEmpty(S);
9     return S;
10 }
11
12 void MakeEmpty(Stack S)
```

```

13 {
14     if (S == NULL)
15         Error("Must use CreateStack first");
16     else
17         while(!IsEmpty(S)) Pop(S);
18 }

```

```

1 void Push(ElementType X, Stack S)
2 {
3     PtrToNode TmpCell;
4     TmpCell = malloc(sizeof(struct Node));
5     if (TmpCell == NULL)
6         Fatal Error("Out of space!") ;
7     else
8     {
9         TmpCell->Element = X;
10        TmpCell->Next = S->Next;
11        S->Next = TmpCell;
12    }
13 }

```

```

1 ElementType Top(Stack S)
2 {
3     if(!IsEmpty(S))
4         return S->Next->Element;
5     Error("Empty stack") ;
6     return 0; /* Return value used to avoid warning*/
7 }

```

```

1 void Pop(Stack s)
2 {
3     PtrToNode FirstCell;
4     if(IsEmpty(S))
5         Error("Empty stack") ;
6     else
7     {
8         FirstCell = S->Next;
9         S->Next = S->Next->Next;
10        free(FirstCell);
11    }
12 }

```

Array Implementation of Stacks

```
1 struct StackRecord {
2     int Capacity;           /* size of stack */
3     int TopOfStack;         /* the top pointer */
4     /* ++ for push, -- for pop, -1 for empty stack */
5     ElementType *Array;     /* array for stack elements */
6 };
```

- The stack model must be well **encapsulated(封装)**. That is, no part of your code, except for the stack routines, can attempt to access the Array or TopOfStack variable.
- Error check must be done before Push or Pop (Top).

```
1 Stack CreateStack(int MaxElements)
2 {
3     Stack S;
4     if(MaxElements < MinStackSize)
5         Error("Stack size is too small") ;
6     S = malloc(sizeof(struct StackRecord));
7     if (S == NULL)
8         Fatal Error("Out of space!!!") ;
9
10    S->Array = malloc(sizeof(ElementType) * MaxElements) ;
11    if(S->Array == NULL)
12        Fatal Error("Out of space!!!");
13    S->Capacity = MaxElements;
14    MakeEmpty(S) ;
15    return S;
16 }
```

```
1 void DisposeStack(Stack S)
2 {
3     if(S != NULL)
4     {
5         free(S->Array);
6         free(S);
7     }
8 }
```

```
1 int IsEmpty(Stack S)
2 {
3     return S->TopOfStack == EmptyTOS;
4 }
```

```
1 void MakeEmpty(Stack S)
2 {
3     S->TopOfStack = EmptyTOS;
4 }
```

```
1 void Push(ElementType X, Stack S)
2 {
3     if (IsFull(S))
4         Error("Full stack");
5     else
6         S->Array[ ++S->TopOfStack ] = X;
7 }
```

```
1 ElementType Top(Stack S)
2 {
3     if(! IsEmpty(S))
4         return S->Array[ S->TopOfStack ];
5     Error("Empty stack") ;
6     return 0; /* Return value used to avoid warning*/
7 }
```

```
1 void Pop(Stack S)
2 {
3     if(IsEmpty(S))
4         Error("Empty stack") ;
5     else
6         S->TopOfStack--;
7 }
```

```
1 ElementType TopAndPop(Stack S)
2 {
3     if(!Is Empty(S))
4         return S->Array[ S->TopOfStack-- ];
5     Error("Empty stack");
6     return 0; /* Return value used to avoid warnin */
7 }
```

Application

1. Balancing Symbols

检查括号是否平衡

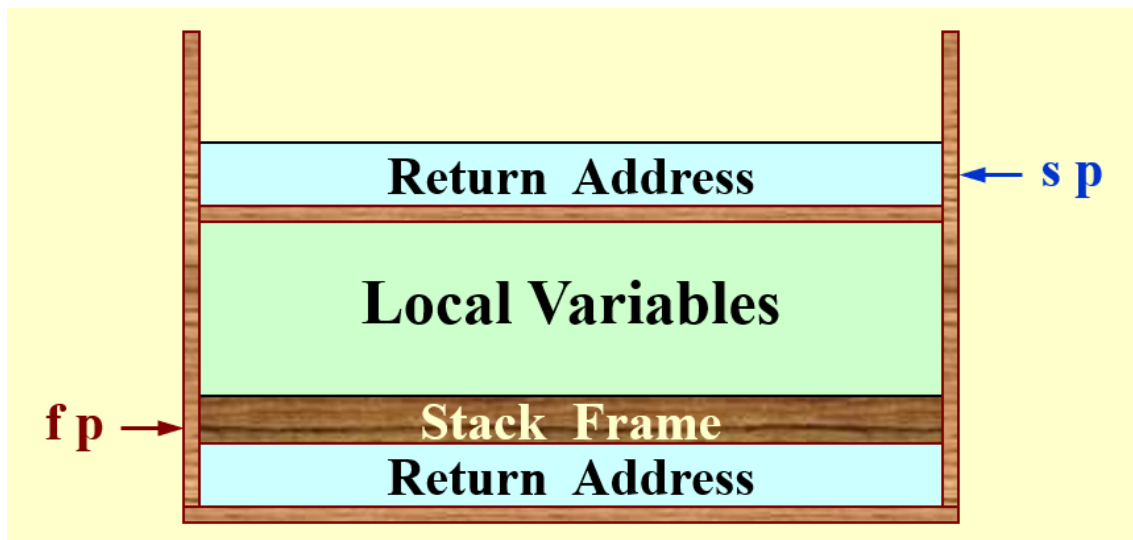
```
1 Algorithm {
2     Make an empty stack S;
3     while (read in a character c) {
4         if (c is an opening symbol)
5             Push(c, S);
6         else if (c is a closing symbol) {
7             if (S is empty) { ERROR; exit; }
8             else { /* stack is okay */
9                 if (Top(S) doesn't match c) { ERROR,
10                    exit; }
11                 else Pop(S);
12             } /* end else-if-closing symbol */
13         } /* end while-loop */
14         if (S is not empty) ERROR;
15     }
```

2. Postfix Evaluation 后缀表达式

3. Infix to Postfix Conversion

- 读到一个操作数时立即把它放到输出中
- 读到一个操作符时从栈中弹出栈元素直到发现优先级更低的元素为止，再将操作符压入栈中
- The order of operands is the **same** in infix and postfix.
- Operators with **higher** precedence appear **before** those with **lower** precedence.
- Never pop a '(' from the stack except when processing a ')'.
- When '(' is not in the stack, its precedence is the highest; but when it is in the stack, its precedence is the lowest.
- Exponentiation associates **right to left**.

4. Function Calls (System Stack)



Note : Recursion can always be **completely removed**. Non recursive programs are generally **faster** than equivalent recursive programs. However, recursive programs are in general much **simpler and easier to understand**.

2.4 The Queue ADT

- First-In-First-Out (FIFO)
- **Objects** : A finite ordered list with zero or more elements.
- **Operations** :
 - IsEmpty
 - CreatQueue
 - DisposeQueue
 - MakeEmpty
 - Enqueue
 - Front
 - Dequeue

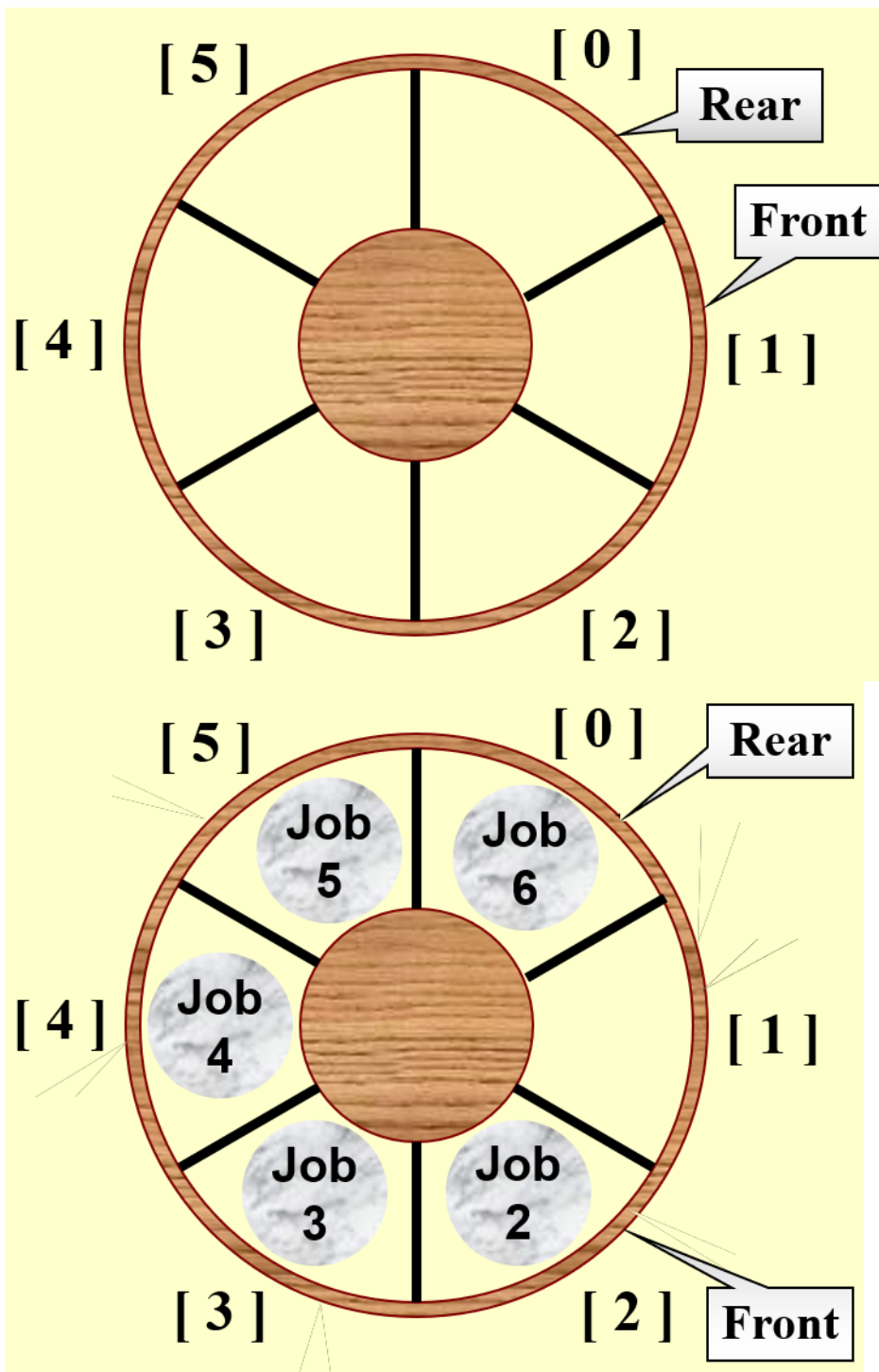
Array Implementation of Queues

```

1 struct QueueRecord {
2     int Capacity ;           /* max size of queue */
3     int Front;               /* the front pointer */
4     int Rear;                /* the rear pointer */
5     int Size;                /* optional - the current size of
6     ElementType *Array;      /* array for queue elements */
7 };

```

Circular Queue :



- The maximum capacity of this queue is 5.

Note : Adding a **Size** field can avoid wasting one empty space to distinguish "full" from "empty".

3 Trees

3.1 Preliminaries

[Definition] A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of (1) a distinguished node r , called the root; (2) and zero or more nonempty (sub)trees, each of whose roots are connected by a directed edge from r .

- Subtrees must not connect together. Therefore every node in the tree is the root of some subtree.
- There are $N-1$ edges in a tree with N nodes

Terminologies

- degree of a node : 结点的子树个数
- degree of a tree : 结点的度的最大值
- parent : 有子树的结点
- children : the roots of the subtrees of a parent
- siblings : children of the same parent
- leaf (terminal node) : a node with degree 0 (no children)
- path from n_1 to n_k : a **unique** sequence of nodes n_1, n_2, \dots, n_k such that n_i is the parent of n_{i+1} for $1 \leq i < k$
- length of path : 路径上边的条数
- depth of n_i : 从根结点到 n_i 结点的路径的长度 ($Depth(root) = 0$)
- height of n_i : 从 n_i 结点到叶结点的最长路径的长度 ($Height(leaf) = 0$)
- height/depth of a tree : 根结点的高度/最深的叶结点的深度
- ancestors of a node : 从此结点到根结点的路径上的所有结点
- descendants of a node : 此结点的子树中的所有结点

List Representation

- The size of each node depends on the number of branches.

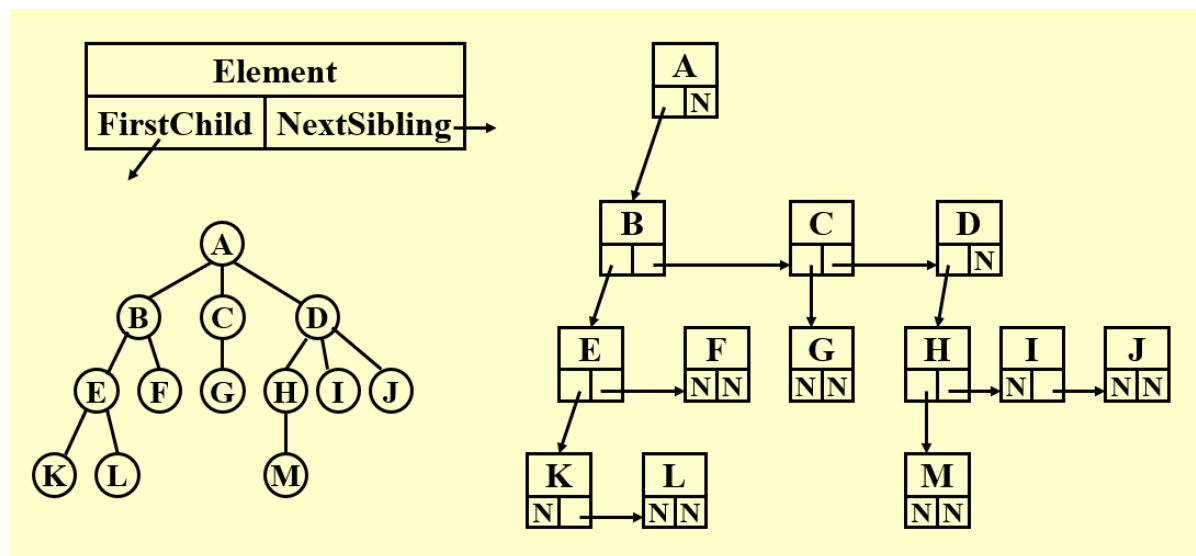
1-1 It is always possible to represent a tree by a one-dimensional integer array. (1分)

☐ T ☒ F

1-1 答案错误 ⓘ (0分)  创建提问

The correct answer is T.

FirstChild-NextSibling Representation



- The representation is **not unique** since the children in a tree can be of any order.

3.2 Binary Trees

[Definition] A binary tree is a tree in which no node can have more than two children.

Tree Traversals (visit each node exactly once)

1. Preorder Traversal

```
1 void preorder( tree_ptr tree )
2 {
3     if( tree )
4     {
5         visit ( tree );
6         for (each child C of tree )
7             preorder ( C );
8     }
9 }
```

2. Postorder Traversal

```

1 void postorder( tree_ptr tree )
2 {
3     if( tree )
4     {
5         for (each child C of tree )
6             postorder ( C );
7         visit ( tree );
8     }
9 }

```

3. Levelorder Traversal

```

1 void levelorder( tree_ptr tree )
2 {
3     enqueue ( tree );
4     while (queue is not empty)
5     {
6         visit ( T = dequeue ( ) );
7         for (each child C of T )
8             enqueue ( C );
9     }
10 }

```

4. Inorder Traversal

```

1 void inorder( tree_ptr tree )
2 {
3     if( tree )
4     {
5         inorder ( tree->Left );
6         visit ( tree->Element );
7         inorder ( tree->Right );
8     }
9 }

```

Iterative Program :

```

1 void iter_inorder( tree_ptr tree )
2 {
3     Stack S = CreateStack( MAX_SIZE );
4     for ( ; ; )
5     {
6         for ( ; tree; tree = tree->Left )
7             Push ( tree, S );

```

```

8      tree = Top ( S );
9      Pop( S );
10     if ( !tree ) break;
11     visit ( tree->Element );
12     tree = tree->Right;
13 }
14 }

```

2-2 If a general tree T is converted into a binary tree BT , then which of the following BT traversals gives the same sequence as that of the post-order traversal of T ? (2分)

- ☐ A. Pre-order traversal
☒ B. In-order traversal
☐ C. Post-order traversal
☐ D. Level-order traversal

2-2 答案正确 (2分) [创建提问](#)

Threaded Binary Trees

- A full binary tree with n nodes has $2n$ links, and $n + 1$ of them are NULL.
- Replace the NULL links by “threads” which will make traversals easier.

Rules :

- If **Tree->Left** is null, replace it with a pointer to the inorder **predecessor(中序前驱)** of Tree.
- If **Tree->Right** is null, replace it with a pointer to the inorder **successor(中序后继)** of Tree.
- There must not be any loose threads. Therefore a threaded binary tree must have a **head node** of which the left child points to the first node.

```

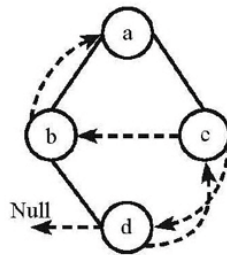
1  typedef struct ThreadedTreeNode *PtrToThreadedNode;
2  typedef struct PtrToThreadedNode ThreadedTree;
3  typedef struct ThreadedTreeNode
4  {
5      int LeftThread;          /* if it is TRUE, then Left */
6      ThreadedTree Left;      /* is a thread, not a child ptr.*/
7      ElementType Element;
8      int RightThread;         /* if it is TRUE, then Right */
9      ThreadedTree Right;     /* is a thread, not a child ptr.*/
10 }

```

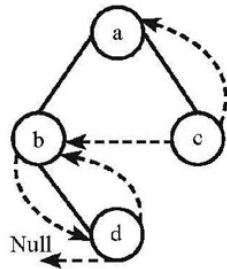
- 线索化的实质就是将二叉链表中的空指针改为指向前驱或后继的线索。由于前驱和后继信息只有在遍历该二叉树时才能得到，所以，线索化的过程就是在遍历的过程中修改空指针的过程。

2-4 Among the following threaded binary trees (the threads are represented by dotted curves), which one is the postorder threaded tree? (2分)

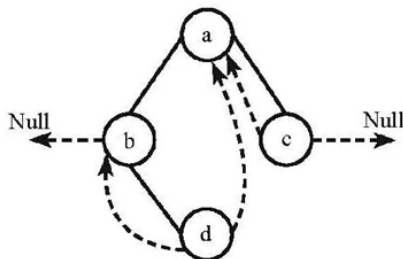
☐ A.



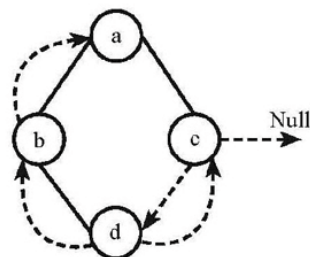
☒ B.



☐ C.



☐ D.



2-4 答案正确 (2分) [创建提问](#)

- In a tree, the order of children does not matter. But in a binary tree, left child and right child are different.

Properties of Binary Trees

- The maximum number of nodes on level i is 2^{i-1} , $i \geq 1$.
- The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.
- For any nonempty binary tree, $n_0 = n_2 + 1$ where n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2.

3.3 Binary Search Trees

[Definition] A binary search tree is a binary tree. It may be empty. If it is not empty, it satisfies the following properties:

- 每个结点有一个互不不同的值
- 若左子树非空，则左子树上所有结点的值均小于根结点的值
- 若右子树非空，则右子树上所有结点的值均大于根结点的值
- 左、右子树也是是一棵二叉查找树

ADT

- **Objects** : A finite ordered list with zero or more elements.
- **Operations** :
 - SearchTree MakeEmpty(SearchTree T)
 - Position Find(ElementType X, SearchTree T)
 - Position FindMin(SearchTree T)
 - Position FindMax(SearchTree T)
 - SearchTree Insert(ElementType X, SearchTree T)
 - SearchTree Delete(ElementType X, SearchTree T)
 - ElementType Retrieve(Position P)

Implementations

1. Find

```
1 Position Find( ElementType X, SearchTree T )
2 {
3     if ( T == NULL )
4         return NULL; /* not found in an empty tree */
5     if ( X < T->Element ) /* if smaller than root */
6         return Find( X, T->Left ); /* search left subtree
7     */
8     else
9         if ( X > T->Element ) /* if larger than root */
10            return Find( X, T->Right ); /* search right
11            subtree */
12        else /* if X == root */
13            return T; /* found */
14 }
```

- $T(N) = S(N) = O(d)$ where d is the depth of X

Iterative program :

```

1 Position Iter_Find( ElementType X, SearchTree T )
2 {
3     while ( T )
4     {
5         if ( X == T->Element )
6             return T; /* found */
7         if ( X < T->Element )
8             T = T->Left; /*move down along left path */
9         else
10            T = T-> Right; /* move down along right path
11    */
12    } /* end while-loop */
13    return NULL; /* not found */
14 }

```

2. FindMin

```

1 Position FindMin( SearchTree T )
2 {
3     if ( T == NULL )
4         return NULL; /* not found in an empty tree */
5     else
6         if ( T->Left == NULL ) return T; /* found left
7    most */
8         else return FindMin( T->Left ); /* keep moving to
9    left */
10 }

```

3. FindMax

```

1 Position FindMax( SearchTree T )
2 {
3     if ( T != NULL )
4         while ( T->Right != NULL )
5             T = T->Right; /* keep moving to find right
6    most */
7     return T; /* return NULL or the right most */
8 }

```

4. Insert

```

1 SearchTree Insert( ElementType X, SearchTree T )
2 {

```

```

3      if ( T == NULL ) /* Create and return a one-node tree
   */
4      {
5          T = malloc( sizeof( struct TreeNode ) );
6          if ( T == NULL )
7              FatalError( "Out of space!!!" );
8          else
9              {
10                 T->Element = X;
11                 T->Left = T->Right = NULL;
12             }
13      } /* End creating a one-node tree */
14      else /* If there is a tree */
15          if ( X < T->Element )
16              T->Left = Insert( X, T->Left );
17          else
18              if ( X > T->Element )
19                  T->Right = Insert( X, T->Right );
20          /* Else X is in the tree already; we'll do nothing
   */
21      return T; /* Do not forget this line!! */
22  }

```

- 内存越界后不会马上报错，在下次free或malloc时会失败
- Handle duplicated keys
- $T(N) = O(d)$

5. Delete

- Delete a leaf node : Reset its parent link to NULL
- Delete a degree 1 node : Replace the node by its single child
- Delete a degree 2 node : 用左子树最大值结点或右子树最小值结点替换

```

1  SearchTree Delete( ElementType X, SearchTree T )
2  {
3      Position TmpCell;
4      if ( T == NULL ) Error( "Element not found" );
5      else if ( X < T->Element ) /* Go left */
6          T->Left = Delete( X, T->Left );
7      else if ( X > T->Element ) /* Go right */
8          T->Right = Delete( X, T->Right );
9      else /* Found element to be deleted */
10         if ( T->Left && T->Right ) { /* Two children */
11             /* Replace with smallest in right subtree */
12             TmpCell = FindMin( T->Right );
13             T->Element = TmpCell->Element;

```

```

14         T->Right = Delete( T->Element, T->Right ); }
    /* End if */
15     else
16     { /* One or zero child */
17         TmpCell = T;
18         if ( T->Left == NULL ) /* Also handles 0 child
    */
19             T = T->Right;
20         else if ( T->Right == NULL )
21             T = T->Left;
22         free( TmpCell );
23     } /* End else 1 or 0 child */
24     return T;
25 }

```

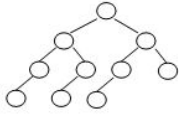
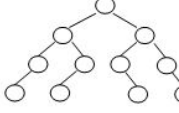
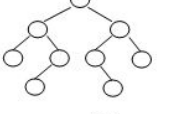

- $T(N) = O(d)$

Note : If there are not many deletions, then **lazy deletion** may be employed: add a flag field to each node, to mark if a node is active or is deleted. Therefore we can delete a node without actually freeing the space of that node. If a deleted key is reinserted, we won't have to call malloc again.

6. Average-Case Analysis

- The average depth over all nodes in a tree is $O(\log N)$ on the assumption that all trees are equally likely.
- 将 n 个元素存入二叉搜索树，树的高度将由插入序列决定

2-5 Among the following binary trees, which one can possibly be the decision tree (the external nodes are excluded) for binary search? (3分)

- ☐ A. 
- ☒ B. 
- ☐ C. 
- ☐ D. 

2-5 答案错误 ① (0分) [创建提问](#)

The correct answer is A.

4 Priority Queues (Heaps)

4.1 ADT Model

- **Objects** : A finite ordered list with zero or more elements.
 - **Operations** :
 - PriorityQueue Initialize(int MaxElements);
 - void Insert(ElementType X, PriorityQueue H);
 - ElementType DeleteMin(PriorityQueue H);
 - ElementType FindMin(PriorityQueue H);
-

4.2 Implementations

Array

- Insertion — add one item at the end $\sim \Theta(1)$
- Deletion — find the largest / smallest key $\sim \Theta(n)$
remove the item and shift array $\sim O(n)$

Linked List

- Insertion — add to the front of the chain $\sim \Theta(1)$
- Deletion — find the largest / smallest key $\sim \Theta(n)$
remove the item $\sim \Theta(1)$
- **Never more deletions than insertions**

Ordered Array

- Insertion — find the proper position $\sim O(\log n)$
shift array and add the item $\sim O(n)$
- Deletion — remove the first / last item $\sim \Theta(1)$

Ordered Linked List

- Insertion — find the proper position $\sim O(n)$
add the item $\sim \Theta(1)$
- Deletion — remove the first / last item $\sim \Theta(1)$

Binary Search Tree

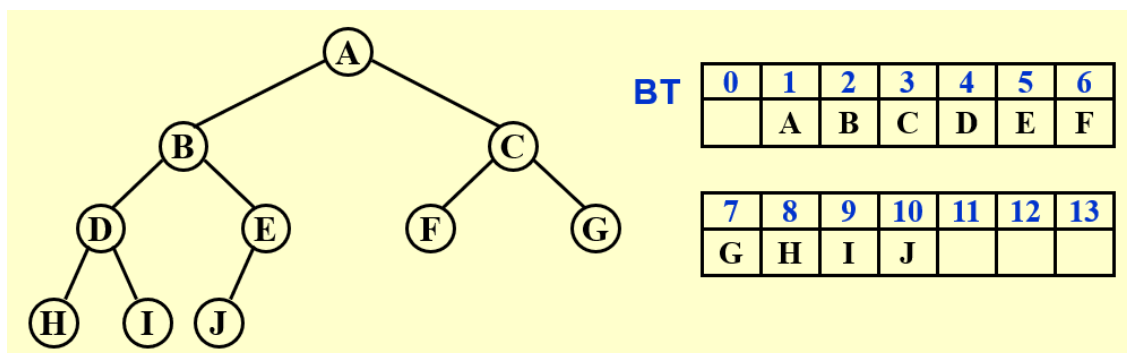
- Both insertion and deletion will take $O(\log N)$ only.
- Only delete the the minimum element, always delete from the left subtrees.
- Keep a balanced tree
- But there are many operations related to AVL tree that we don't really need for a priority queue.

4.3 Binary Heap

Structure Property

[Definition] A binary tree with n nodes and height h is **complete** if its nodes correspond to the nodes numbered from 1 to n in the perfect binary tree of height h .

- A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes.
- $h = \lfloor \log N \rfloor$
- Array Representation : BT[n + 1] (BT[0] is not used)



[Lemma]

1. $index\ of\ parent(i) = \begin{cases} \lfloor i/2 \rfloor & i \neq 1 \\ None & i = 1 \end{cases}$
2. $index\ of\ left_child(i) = \begin{cases} 2i & 2i \leq n \\ None & 2i > n \end{cases}$
3. $index\ of\ right_child(i) = \begin{cases} 2i + 1 & 2i + 1 \leq n \\ None & 2i + 1 > n \end{cases}$

```
1 PriorityQueue Initialize( int MaxElements )
2 {
3     PriorityQueue H;
4     if ( MaxElements < MinPQSize )
5         return Error( "Priority queue size is too small" );
6     H = malloc(sizeof( struct HeapStruct ));
```

```

7     if ( H == NULL )
8         return FatalError( "Out of space!!!" );
9     /* Allocate the array plus one extra for sentinel */
10    H->Elements = malloc(( MaxElements + 1 ) * sizeof(
ElementType ));
11    if ( H->Elements == NULL )
12        return FatalError( "Out of space!!!" );
13    H->Capacity = MaxElements;
14    H->Size = 0;
15    H->Elements[0] = MinData; /* set the sentinel */
16    return H;
17 }

```

Heap Order Property

[Definition] A **min tree** is a tree in which the key value in each node is no larger than the key values in its children (if any). A **min heap** is a **complete** binary tree that is also a min tree.

- We can declare a **max** heap by changing the heap order property.

Basic Heap Operations

1. Insertion

```

1  /*H->Element[ 0 ] is a sentinel that is no larger than the
   minimum element in the heap.*/
2  void Insert( ElementType X, PriorityQueue H )
3  {
4      int i;
5      if ( IsFull( H ))
6      {
7          Error( "Priority queue is full" );
8          return;
9      }
10     for ( i = ++H->Size; H->Elements[ i/2 ] > X; i /= 2 )
11         H->Elements[ i ] = H->Elements[ i/2 ]; /*Percolate
up, faster than swap*/
12     H->Elements[ i ] = X;
13 }

```

$$T(N) = O(\log N)$$

2. DeleteMin

```

1  ElementType DeleteMin( PriorityQueue H )

```

```

2  {
3      int i, Child;
4      ElementType MinElement, LastElement;
5      if ( IsEmpty( H ) )
6      {
7          Error( "Priority queue is empty" );
8          return H->Elements[ 0 ];
9      }
10     MinElement = H->Elements[ 1 ]; /*Save the min
element*/
11     LastElement = H->Elements[ H->Size-- ]; /*Take last
and reset size*/
12     for ( i = 1; i * 2 <= H->Size; i = Child ) /*Find
smaller child*/
13     {
14         Child = i * 2;
15         if (Child != H->Size && H->Elements[Child+1] < H-
>Elements[Child])
16             Child++;
17         if ( LastElement > H->Elements[ Child ] )
18             /*Percolate one level*/
19             H->Elements[ i ] = H->Elements[ Child ];
20         else
21             break; /*Find the proper position*/
22     }
23     H->Elements[ i ] = LastElement;
24     return MinElement;
25 }

```

$$T(N) = O(\log N)$$

Other Heap Operations

- 查找除最小值之外的值需要对整个堆进行线性扫描

1. DecreaseKey — Percolate up
2. IncreaseKey — Percolate down
3. Delete
4. BuildHeap

将N 个关键字以任意顺序放入树中，保持结构特性，再执行下滤

```

1  for (i = N/2; i > 0; i--)
2      PercolateDown(i);

```

$$T(N) = O(N)$$

[Theorem] For the perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h + 1)$.

1-7 For binary heaps with N elements, the *BuildHeap* function (which adjust an array of elements into a heap in linear time) does at most $N - \log(N + 1)$ comparisons between elements.

☒ T ☐ F

1-7 答案错误 (0分) [创建提问](#)

4.4 Applications of Priority Queues

Heap Sort

查找一个序列中第k小的元素

The function is to find the K -th smallest element in a list A of N elements. The function `BuildMaxHeap(H, K)` is to arrange elements $H[1] \dots H[K]$ into a max-heap.

```

1  ElementType FindkthSmallest ( int A[], int N, int K )
2  {  /* it is assumed that K<=N */
3      ElementType *H;
4      int i, next, child;
5
6      H = (ElementType*)malloc((K+1)*sizeof(ElementType));
7      for ( i = 1; i <= K; i++ ) H[i] = A[i-1];
8      BuildMaxHeap(H, K);
9
10     for ( next = K; next < N; next++ ) {
11         H[0] = A[next];
12         if ( H[0] < H[1] ) {
13             for ( i = 1; i*2 <= K; i = child ) {
14                 child = i*2;
15                 if ( child != K && H[child+1] > H[child] )
16                     child++;
17                 if ( H[0] < H[child] )
18                     H[i] = H[child];
19                 else break;
20             }
21             H[i] = H[0];
22         }
23     }
24     return H[1];
25 }
```

4.5 d -Heaps — All nodes have d children

Note :

- DeleteMin will take $d - 1$ comparisons to find the smallest child. Hence the total time complexity would be $O(d \log_d N)$.
- $*2$ or $/2$ is merely a **bit shift**, but $*d$ or $/d$ is not.
- When the priority queue is too large to fit entirely in main memory, a d -heap will become interesting.

2-3 If a d -heap is stored as an array, for an entry located in position i , the parent, the first child and the last child are at: (2分)

- ☐ A. $\lceil (i + d - 2)/d \rceil$, $(i - 2)d + 2$, and $(i - 1)d + 1$
- ☐ B. $\lceil (i + d - 1)/d \rceil$, $(i - 2)d + 1$, and $(i - 1)d$
- ☒ C. $\lfloor (i + d - 2)/d \rfloor$, $(i - 1)d + 2$, and $id + 1$
- ☐ D. $\lfloor (i + d - 1)/d \rfloor$, $(i - 1)d + 1$, and id

2-3 答案正确 (2分)  创建提问

2-11 It is known that a 3-heap is a heap whose nodes have 3 children. Suppose that the level-order traversal sequence of a max-3-heap is {88, 76, 65, 82, 68, 46, 52, 44, 62, 33, 75, 60, 55, 28}. Use the linear algorithm to adjust this max-3-heap into a min-3-heap, and then run DeleteMin. As a result, there are _ nodes whose positions are not moved in the process. (3分)

- ☐ A. 2
- ☐ B. 3
- ☐ C. 4
- ☒ D. 5

评测结果: 答案错误 (0分)

正确答案是4, 注意“in the process”

5 The Disjoint Set

5.1 Equivalence Relations

[Definition] A *relation* R is defined on a set S if for every pair of elements (a, b) , $a, b \in S$, $a R b$ is either true or false. If $a R b$ is true, then we say that a is related to b .

[Definition] A relation, \sim , over a set, S , is said to be an *equivalence relation* over S if it is symmetric, reflexive, and transitive over S .

[Definition] Two members x and y of a set S are said to be in the same *equivalence class* if $x \sim y$.

5.2 The Dynamic Equivalence Problem

- Given an equivalence relation \sim , decide for any a and b if $a \sim b$

```
1 Algorithm: (Union/Find)
2 {
3     /* step 1: read the relations in */
4     Initialize N disjoint sets;
5     while ( read in a ~ b )
6     {
7         if ( !(Find(a) == Find(b)) ) /*Dynamic(on-line)*/
8             Union the two sets;
9     } /* end-while */
10    /* step 2: decide if a ~ b */
11    while ( read in a and b )
12        if ( Find(a) == Find(b) )
13            output( true );
14        else
15            output( false );
16 }
```

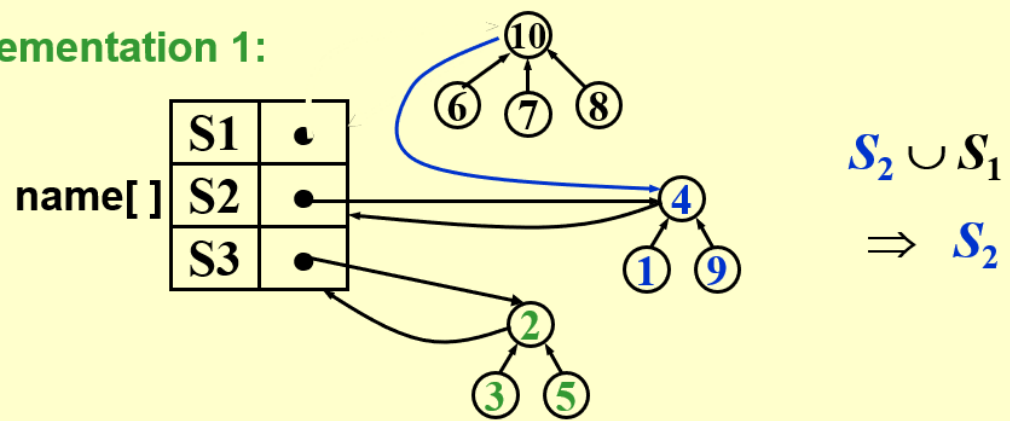
- Elements** of the sets : $1, 2, 3, \dots, N$
 - Sets** : S_1, S_2, \dots and $S_i \cap S_j = \emptyset$ (if $i \neq j$)
 - Operations** :
 - Union(i, j) = Replace S_i and S_j by $S = S_i \cup S_j$
 - Find(i) = Find the set S_k which contains the element i
-

5.3 Basic Data Structure

Union(i, j)

- Make S_i a subtree of S_j , or vice versa, that is to set the parent pointer of one of the roots to the other root.
- Implementation 1** :

Implementation 1:



Implementation 2 :

- The elements are numbered from 1 to N, hence they can be used as indices of an array.
- $S[\text{element}] = \text{the element's parent}$
- Note : $S[\text{root}] = 0$ and set name = root index
- 数组初始化全部为0

```

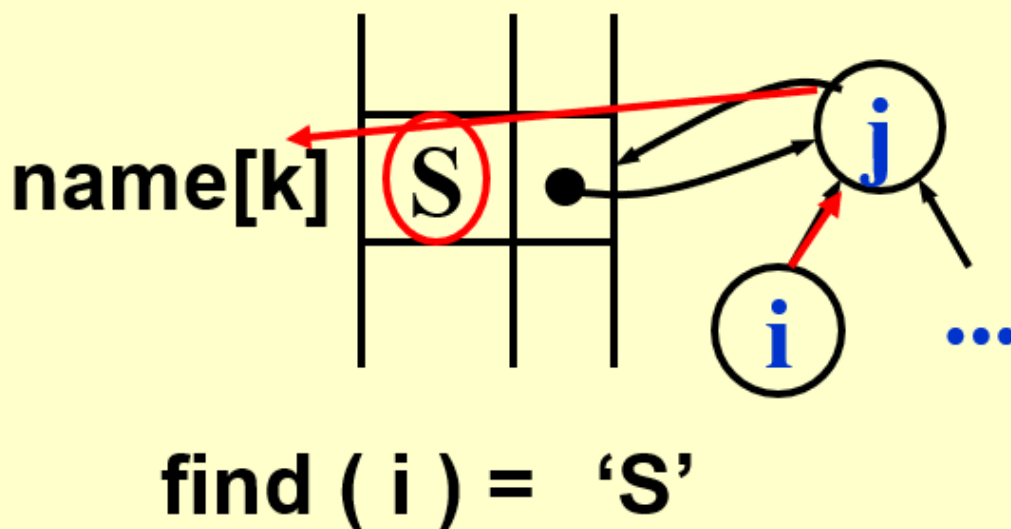
1 void SetUnion(DisjSet S, SetType Rt1, SetType Rt2)
2 {
3     S[Rt2] = Rt1;
4 }

```

Find(i)

Implementation 1 :

Implementation 1:



Implementation 2 :


```

1 SetType Find(ElementType X, DisjSet S)
2 {
3     for ( ; S[X]>0; X=S[X]);
4     return X;
5 }

```

Analysis

- Union and find are always paired. Thus we consider the performance of a sequence of **union-find operations**.

[[Example]] Given $S = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 \}$ and 9 relations: $12 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 12$. We have 3 equivalence classes $\{ 2, 4, 7, 11, 12 \}$, $\{ 1, 3, 5 \}$, and $\{ 6, 8, 9, 10 \}$

```

1 Algorithm using union-find operations:
2 {
3     Initialize Si = { i } for i = 1, ..., 12 ;
4     for ( k = 1; k <= 9; k++ ) /* for each pair i R j */
5     {
6         if ( Find( i ) != Find( j ) )
7             SetUnion( Find( i ), Find( j ) );
8     }
9 }

```

- Worst case : $T(N) = \Theta(N^2)$

5.4 Smart Union Algorithms

Union-by-Size

- Always change the smaller tree
- $S[\text{Root}] = -\text{size}$, initialized to be -1
- **[Lemma]** Let T be a tree created by union-by-size with N nodes, then $\text{height}(T) \leq \lfloor \log_2 N \rfloor + 1$.

Proved by induction. Each element can have its set name changed at most $\log_2 N$ times.

- **Time complexity** of N Union and M Find operations is now $O(N + M \log_2 N)$.

```

1  /* Assumes Root1 and Root2 are roots*/
2  void SetUnion(DisjSet S, SetType Root1, SetType Root2)
3  {
4      if (S[Root1] <= S[Root2])
5      {
6          S[Root1] += S[Root2];
7          S[Root2] = Root1;
8      }
9      else
10     {
11         S[Root2] += S[Root1];
12         S[Root1] = Root2;
13     }
14 }

```

Union-by-Height

- Always change the shallow tree
- 保证所有的树的深度最多是 $O(\log N)$

```

1  /* Assumes Root1 and Root2 are roots*/
2  void SetUnion(DisjSet S, SetType Root1, SetType Root2)
3  {
4      if ( S[Root2] < S[Root1]) /*Root2 is deeper set*/
5          S[Root1] = Root2;      /*Make Root2 new root*/
6      else
7      {
8          if (S[Root1] == S[Root2]) /*Same height*/
9              S[Root1]--;
10         S[Root2] = Root1;
11     }
12 }

```

5.5 Path Compression

- 从X到Root的路径上的每一个结点都使它的父结点变成Root

```

1 SetType Find( ElementType X, DisjSet S )
2 {
3     if ( S[ X ] <= 0 )
4         return X;
5     else
6         return S[ X ] = Find( S[ X ], S );
7 }

```

```

1 SetType Find( ElementType X, DisjSet S )
2 {
3     ElementType root, trail, lead;
4     for ( root = X; S[ root ] > 0; root = S[ root ] ); /*
find the root */
5     for ( trail = X; trail != root; trail = lead )
6     {
7         lead = S[ trail ];
8         S[ trail ] = root;
9     } /* collapsing */
10    return root;
11 }

```

- Note : Not compatible with union-by-height since it changes the heights. Just take “height” as an estimated **rank**.

5.6 Worst Case for Union-by-Rank and Path Compression

[Lemma] Let $T(M, N)$ be the maximum time required to process an intermixed sequence of $M \geq N$ finds and $N - 1$ unions, then $k_1 M \alpha(M, N) \leq T(M, N) \leq k_2 M \alpha(M, N)$ for some positive constants k_1 and k_2 .

- Ackermann's Function

$$A(i, j) = \begin{cases} 2^j & i = 1, j \geq 1 \\ A(i-1, 2) & i \geq 2, j = 1 \\ A(i-1, A(i, j-1)) & i \geq 2, j \geq 2 \end{cases}$$

$$A(2, 4) = 2^{2^{2^2}} = 2^{65536}$$

- $\alpha(M, N) = \min\{i \geq 1 \mid A(i, \lfloor M/N \rfloor) > \log N\} \leq O(\log^* N) \leq 4$

$\log^* N$ (inverse Ackermann function) = number of times the logarithm is applied to N until the result ≤ 1 .

5.7 Conclusion

一共有五种算法，注意看清题设

- No smart union
 - Union-by-size
 - Union-by-height
 - Union-by-size + Path Compression
 - Union-by-height + Path Compression
-

6 Graph Algorithms

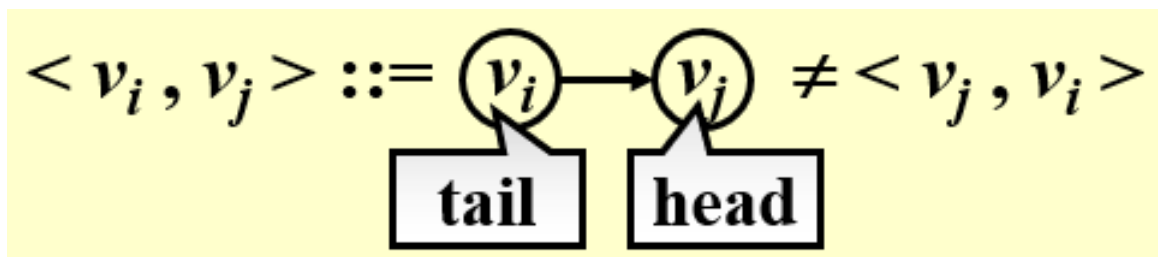
6.1 Definitions

- $G(V, E)$ where G = graph, $V = V(G)$ = finite nonempty set of vertices, and $E = E(G)$ = finite set of edges.

Undirected graph

- $(v_i, v_j) = (v_j, v_i)$ = the same edge.

Directed graph(diagraph)

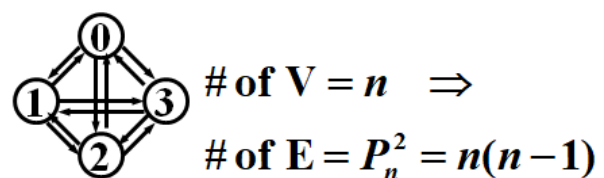
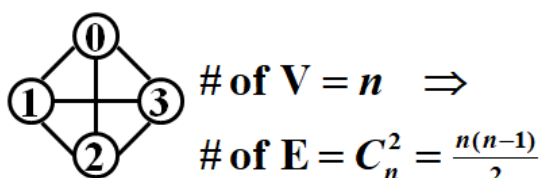


Restrictions

- **Self loop** is illegal.
- **Multigraph** is not considered.

Complete graph

- A graph that has the maximum number of edges.



Adjacent

v_i and v_j are **adjacent** ;
(v_i , v_j) is **incident on** v_i and v_j

v_i is **adjacent to** v_j ; v_j is **adjacent from** v_i ;
< v_i , v_j > is **incident on** v_i and v_j

Subgraph

$$G' \subset G = V(G') \subseteq V(G) \&\& E(G') \subseteq E(G)$$

Path

- Path($\subset G$) from v_p to $v_q = \{v_p, v_{i1}, v_{i2}, \dots, v_{in}, v_q\}$ such that
(v_p, v_{i1}), (v_{i1}, v_{i2}), \dots , (v_{in}, v_q) belong to $E(G)$

Length of a path

- number of edges on the path

Simple path

- $v_{i1}, v_{i2}, \dots, v_{in}$ are distinct.

Cycle

- simple path with $v_p = v_q$

Connected

- v_i and v_j in an undirected G are connected if there is a path from v_i to v_j
(and hence there is also a path from v_j to v_i)
- An undirected graph G is connected if every pair of distinct v_i and v_j are connected

(Connected) Component of an undirected G

- the maximal connected subgraph

Tree

- a graph that is connected and acyclic(非循环的)

DAG

- a directed acyclic graph

Strongly connected directed graph G

- For every pair of v_i and v_j in $V(G)$, there exist directed paths from v_i to v_j and from v_j to v_i .
- If the graph is connected without direction to the edges, then it is said to be weakly connected

Strongly connected component

- the maximal subgraph that is strongly connected

Degree

- number of edges incident to v
- For a directed G , we have **in-degree** and **out-degree**.
- Given G with n vertices and e edges, then

$$e = \left(\sum_{i=0}^{n-1} d_i \right) / 2 \quad \text{where} \quad d_i = \text{degree}(v_i)$$

6.2 Representation of Graphs

Adjacency Matrix

adj_mat[n] [n] is defined for $G(V, E)$ with n vertices, $n \geq 1$:

$$\text{adj_mat}[i][j] = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ or } \langle v_i, v_j \rangle \in E(G) \\ 0 & \text{otherwise} \end{cases}$$

Note : If G is undirected, then $\text{adj_mat}[][]$ is symmetric. Thus we can save space by storing only half of the matrix.

The trick is to store the matrix as a 1-D array:

$$\text{adj_mat} [n(n+1)/2] = \{ a_{11}, a_{21}, a_{22}, \dots, a_{n1}, \dots, a_{nn} \}$$

The index for a_{ij} is $(i * (i - 1) / 2 + j)$.

- This representation wastes space if the graph has a lot of vertices but very few edges.

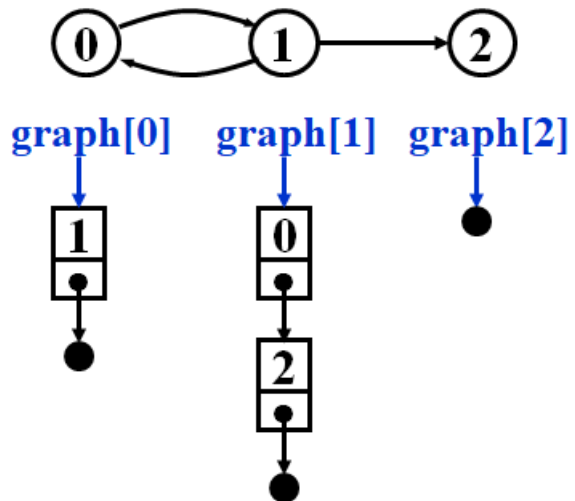
- To find out whether or not G is connected, we'll have to examine all edges. In this case T and S are both $O(n^2)$.

Adjacency Lists

- Replace each row by a linked list

[[Example]]

$$\text{adj_mat}[3][3] = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$



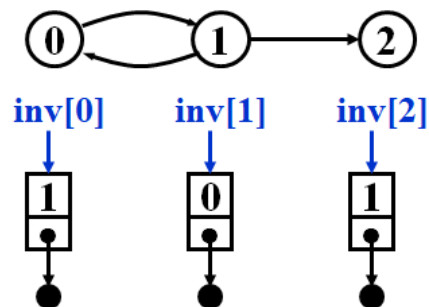
Note : The order of nodes in each list does not matter.

- For undirected G , $S = n \text{ heads} + 2e \text{ nodes} = (n + 2e) \text{ ptrs} + 2e \text{ ints}$
- Degree(i) = number of nodes in $\text{graph}[i]$ (if G is undirected)
- T of examine $E(G) = O(n + e)$

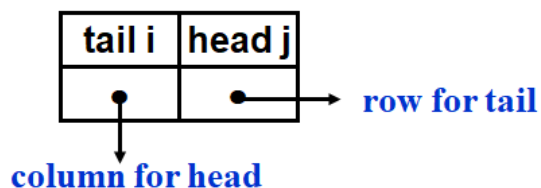
Find in-degree(v) in **directed** graph G

Method 1 Add inverse adjacency lists.

[[Example]]



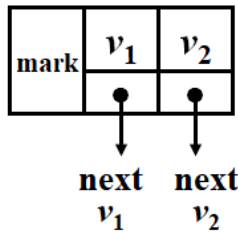
Method 2 Multilist representation for $\text{adj_mat}[i][j]$



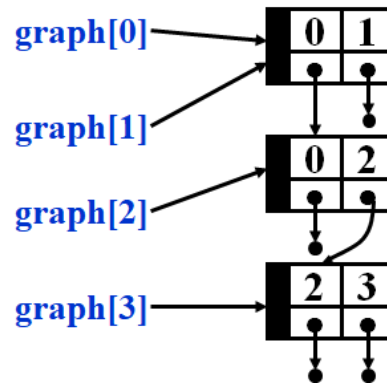
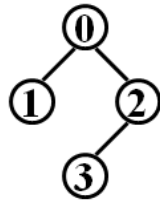
Adjacency Multilists

In adjacency list, for each (i, j) we have two nodes:

$\text{graph}[i] \rightarrow \boxed{j \bullet} \rightarrow \dots\dots$ Now let's combine the two nodes
 $\text{graph}[j] \rightarrow \boxed{i \bullet} \rightarrow \dots\dots$ into one: $\text{graph}[i] \rightarrow \boxed{\text{node}} \leftarrow \text{graph}[j]$



[[Example]]



- Sometimes we need to mark the edge after examine it, and then find the next edge.

Weighted Edges

- $\text{adj_mat}[i][j] = \text{weight}$
- adjacency lists / multilists : add a weight field to the node

6.3 Topological Sort

AOV Network

- digraph G in which $V(G)$ represents activities and $E(G)$ represents precedence relations
- Feasible AOV network must be a directed acyclic graph.
- i is a **predecessor** of j = there is a path from i to j
- i is an **immediate predecessor** of $j = \langle i, j \rangle \in E(G)$. Then j is called a **successor(immediate successor)** of i

Partial order

- a precedence relation which is both **transitive** and **irreflexive**

Note : If the precedence relation is reflexive, then there must be an i such that i is a predecessor of i . That is, i must be done before i is started.

Therefore if a project is **feasible**, it must be **irreflexive**.

[Definition] A *topological order* is a linear ordering of the vertices of a graph such that, for any two vertices, i, j , if i is a predecessor of j in the network then i precedes j in the linear ordering.

Note : The topological orders may **not be unique** for a network.

```
1  /*Test an AOV for feasibility, and generate a topological
   order if possible*/
2  void Topsort( Graph G )
3  {
4      int Counter;
5      Vertex V, W;
6      for ( Counter = 0; Counter < NumVertex; Counter++ )
7      {
8          V = FindNewVertexOfDegreeZero( );
9          if ( V == NotAVertex )
10         {
11             Error ( "Graph has a cycle" );
12             break;
13         }
14         TopNum[ V ] = Counter; /* or output V */
15         for ( each w adjacent to V )
16             Indegree[ w ]--;
17     }
18 }
```

$$T = O(|V|^2)$$

```
1  /*Improvement:Keep all the unassigned vertices of degree 0 in a
   special box (queue or stack)*/
2  void Topsort( Graph G )
3  {
4      Queue Q;
5      int Counter = 0;
6      Vertex V, W;
7      Q = CreateQueue( NumVertex );
8      MakeEmpty( Q );
9      for ( each vertex V )
10         if ( Indegree[ V ] == 0 ) Enqueue( V, Q );
11     while ( !IsEmpty( Q ) )
12     {
13         V = Dequeue( Q );
14         TopNum[ V ] = ++Counter; /* assign next */
15     }
```

```

15         for ( each w adjacent to v )
16             if (--Indegree[ w ] == 0 ) Enqueue( w, Q );
17     } /* end-while */
18     if ( Counter != NumVertex )
19         Error( "Graph has a cycle" );
20     DisposeQueue( Q ); /* free memory */
21 }

```

$$T = O(|V| + |E|)$$

6.4 Shortest Path Algorithms

Given a digraph $G = (V, E)$, and a cost function $c(e)$ for $e \in E(G)$.

The **length** of a path P from **source** to **destination** is $\sum_{e_i \in P} c(e_i)$ (also called **weighted path length**).

Single-Source Shortest-Path Problem

Given as input a weighted graph, $G = (V, E)$, and a distinguished vertex s , find the shortest weighted path from s to every other vertex in G .

Note: If there is no negative-cost cycle, the shortest path from s to s is defined to be zero.

Unweighted Shortest Path

- Breadth-first search 广度优先遍历

Implementation :

- $\text{Table}[i].\text{Dist} ::=$ distance from s to v_i /* initialized to be ∞ except for s */
- $\text{Table}[i].\text{Known} ::=$ 1 if v_i is checked; or 0 if not
- $\text{Table}[i].\text{Path} ::=$ for tracking the path /* initialized to be 0 */

```

1 void Unweighted( Table T )
2 {
3     int CurrDist;
4     Vertex V, W;
5     for ( CurrDist = 0; CurrDist < NumVertex; CurrDist++ )
6     {
7         for ( each vertex v )
8             if ( !T[ v ].Known && T[ v ].Dist == CurrDist )
9             {
10                 T[ v ].Known = true;
11                 for ( each w adjacent to v )

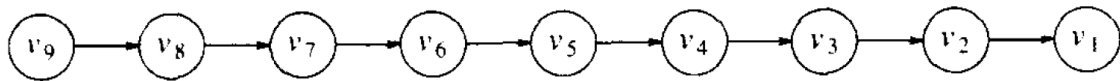
```

```

12         if ( T[ w ].Dist == Infinity )
13         {
14             T[ w ].Dist = CurrDist + 1;
15             T[ w ].Path = v;
16         } /* end-if Dist == Infinity */
17     } /* end-if !Known && Dist == CurrDist */
18 } /* end-for CurrDist */
19 }

```

The worst case :



$$T(N) = O(|V|^2)$$

Improvement :

```

1 void Unweighted( Table T )
2 {
3     /* T is initialized with the source vertex S given */
4     Queue Q;
5     Vertex v, w;
6     Q = CreateQueue( NumVertex );
7     MakeEmpty( Q );
8     Enqueue( S, Q ); /* Enqueue the source vertex */
9     while ( !IsEmpty( Q ) )
10    {
11        v = Dequeue( Q );
12        T[ v ].Known = true; /* not really necessary */
13        for ( each w adjacent to v )
14            if ( T[ w ].Dist == Infinity )
15            {
16                T[ w ].Dist = T[ v ].Dist + 1;
17                T[ w ].Path = v;
18                Enqueue( w, Q );
19            } /* end-if Dist == Infinity */
20    } /* end-while */
21    DisposeQueue( Q ); /* free memory */
22 }

```

$$T = O(|V| + |E|)$$

Weighted Shorted Path

Dijkstra's Algorithm

- Let $S = \{ s \text{ and } v_i \text{'s whose shortest paths have been found} \}$
- For any $u \notin S$, define distance $[u] = \text{minimal length of path } \{ s \rightarrow (v_i \in S) \rightarrow u \}$. If the paths are generated in non-decreasing order, then :
 - the shortest path must go through **only** $v_i \in S$
 - **Greedy Method** : u is chosen so that $\text{distance}[u] = \min\{ w \notin S \mid \text{distance}[w] \}$ (If u is not unique, then we may select any of them)
 - if $\text{distance}[u_1] < \text{distance}[u_2]$ and add u_1 into S , then $\text{distance}[u_2]$ may change. If so, a shorter path from s to u_2 must go through u_1 and $\text{distance}[u_2] = \text{distance}[u_1] + \text{length}(\langle u_1, u_2 \rangle)$.

```
1  typedef int Vertex;
2  struct TableEntry
3  {
4      List Header; /*Adjacency list*/
5      int Known;
6      DistType Dist;
7      Vertex Path;
8  };
9  /*Vertices are numbered from 0*/
10 #define NotAVertex (-1)
11 typedef struct TableEntry Table[ NumVertex ];
```

```
1  void InitTable(Vertex Start, Graph G, Table T)
2  {
3      int i;
4      ReadGraph(G, T); /* Read graph somehow */
5      for(i = 0; i < NumVertex; i++)
6      {
7          T[i].Known = False;
8          T[i].Dist = Infinity;
9          T[i].Path = NotAVertex;
10     }
11     T[ Start ].dist = 0;
12 }
```

```

1  /*Print shortest path to v after Dijkstra has run*/
2  /*Assume that the path exists*/
3  void PrintPath(Vertex v, Table T)
4  {
5      if (T[ v ].Path != NotAVertex)
6      {
7          PrintPath(T[ v ].Path, T);
8          printf(" to" ) ;
9      }
10     printf("%v", v) ; /* %v is pseudocode * /

```

```

1  void Dijkstra( Table T )
2  {
3      Vertex v, w;
4      for ( ; ; )
5      {
6          v = smallest unknown distance vertex;
7          if ( v == NotAVertex ) break;
8          T[ v ].Known = true;
9          for ( each w adjacent to v )
10             if ( !T[ w ].Known )
11                 if ( T[ v ].Dist + Cvw < T[ w ].Dist )
12                     {
13                         Decrease( T[ w ].Dist to T[ v ].Dist +
14 Cvw );
15                         T[ w ].Path = v;
16                     } /* end-if update w */
17             } /* end-for( ; ; ) */
18     }

```

Implementation 1

- Simply scan the table to find the smallest unknown distance vertex.— $O(|V|)$
- Good if the graph is dense

$$T = O(|V|^2 + |E|)$$

Implementation 2

- 堆优化
- Keep distances in a priority queue and call `DeleteMin` to find the smallest unknown distance vertex.— $O(\log |V|)$
- 更新的处理方法

- **Method 1** : `DecreaseKey` — $O(\log |V|)$

$$T = O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$$

- **Method 2** : insert W with updated Dist into the priority queue

Must keep doing `DeleteMin` until an unknown vertex emerges

$$T = O(|E| \log |V|) \text{ but requires } |E| \text{ } \textcode{DeleteMin} \text{ with } |E| \text{ space}$$

- Good if the graph is sparse

Improvements

- Pairing heap
- Fibonacci heap

Graphs with Negative Edge Costs

```

1 void weightedNegative( Table T )
2 {
3     Queue Q;
4     Vertex V, w;
5     Q = CreateQueue (NumVertex );
6     MakeEmpty( Q );
7     Enqueue( S, Q ); /*Enqueue the source vertex*/
8     while ( !IsEmpty( Q ) )
9     {
10         v = Dequeue( Q );
11         for ( each w adjacent to v )
12             if ( T[ v ].Dist + Cvw < T[ w ].Dist )
13             {
14                 T[ w ].Dist = T[ v ].Dist + Cvw;
15                 T[ w ].Path = v;
16                 if ( w is not already in Q )
17                     Enqueue( w, Q );
18             } /*end-if update*/
19     } /*end-while */
20     DisposeQueue( Q ); /*free memory*/
21 }
```

Note : Negative-cost cycle will cause indefinite loop

$$T = O(|V| \times |E|)$$

Acyclic Graphs

- If the graph is acyclic, vertices may be selected in **topological order** since when a vertex is selected, its distance can no longer be lowered without any incoming edges from unknown nodes.
- $T = O(|E| + |V|)$ and no priority queue is needed.

AOE(Activity on Edge) Networks



$EC[j] \setminus LC[j] ::= \text{the earliest \setminus latest completion time for node } v_j$

✂ **CPM (Critical Path Method)**

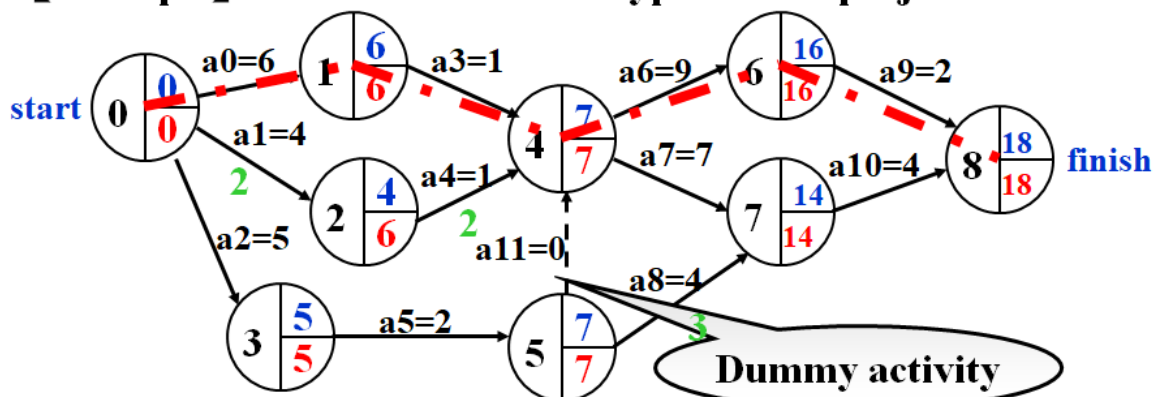
→ Lasting Time
→ Slack Time

Index of vertex

EC Time

LC Time

[[Example]] AOE network of a hypothetical project



➤ Calculation of **EC**: Start from v_0 , for any $a_i = \langle v, w \rangle$, we have

$$EC[w] = \max_{(v,w) \in E} \{EC[v] + C_{v,w}\}$$

➤ Calculation of **LC**: Start from the last vertex v_8 , for any $a_i = \langle v, w \rangle$, we have

$$LC[v] = \min_{(v,w) \in E} \{LC[w] - C_{v,w}\}$$

➤ **Slack Time** of $\langle v, w \rangle = LC[w] - EC[v] - C_{v,w}$

➤ **Critical Path** ::= path consisting entirely of zero-slack edges.

All-Pairs Shortest Path Problem

- For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

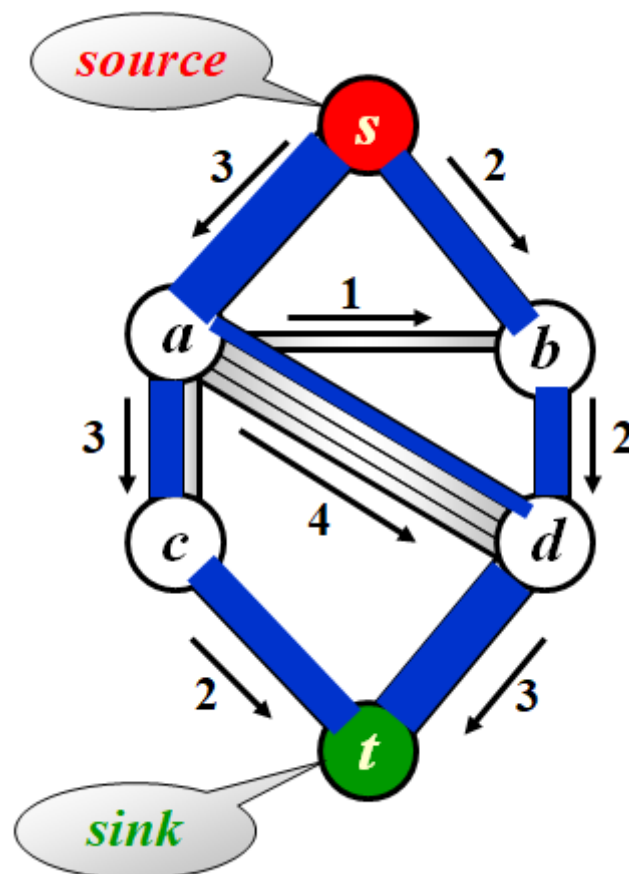
Method 1

- Use **single-source algorithm** for $|V|$ times.
- $T = O(|V|^3)$, works fast on sparse graph.

Method 2

- 动态规划
- $O(|V|^3)$ algorithm given in Chapter 10, works faster on dense graphs.

6.5 Network Flow Problems



- Determine the maximum amount of flow that can pass from s to t .

Note : Total coming in (v) = Total going out (v) where $v \notin \{s, t\}$

A Simple Algorithm

- 流图 G_f 表示算法的任意阶段已经达到的流，开始时 G_f 的所有边都没有流，算法终止时 G_f 包含最大流
- 残余图(residual graph) G_r 表示对于每条边还能添加多少流， G_r 的边叫做残余边(residual edge)

Step 1 : Find any path from s to t in G_r , which is called augmenting path(增长通路).

Step 2 : Take the minimum edge on this path as the amount of flow and add to G_f .

Step 3 : Update G_r and remove the 0 flow edges.

Step 4 : If there is a path from s to t in G_r then go to Step 1, or end the algorithm.

- Step 1中初始选择的路径可能使算法不能找到最优解，贪心算法行不通

A solution

- allow the algorithm to **undo** its decisions
- For each edge (v, w) with flow $f_{v,w}$ in G_f , add an edge (w, v) with flow $f_{v,w}$ in G_r .

Note : The algorithm works for G with **cycles** as well.

[Proposition] If the edge capabilities are *rational numbers*, this algorithm always terminate with a maximum flow.

Analysis

- An augmenting path can be found by an unweighted shortest path algorithm.
- $T = O(f|E|)$ where f is the maximum flow.
- Always choose the augmenting path that allows the largest increase in flow
 - 对Dijkstra算法进行单线(single-line)修改来寻找增长通路
 - cap_{max} 为最大边容量
 - $O(|E| \log cap_{max})$ 条增长通路将足以找到最大流，对于增长通路的每次计算需要 $O(|E| \log |V|)$ 时间

$$\begin{aligned} T &= T_{augmentation} \times T_{find_a_path} \\ &= O(|E| \log cap_{max}) \times O(|E| \log |V|) \\ &= O(|E|^2 \log |V| \log cap_{max}) \end{aligned}$$

- Always choose the augmenting path that has the least number of edges
 - 使用无权最短路算法来寻找增长路径

$$\begin{aligned} T &= T_{augmentation} \times T_{find_a_path} \\ &= O(|E||V|) \times O(|E|) \\ &= O(|E|^2|V|) \end{aligned}$$

Note :

- If every $v \notin \{s, t\}$ has either a single incoming edge of capacity 1 or a single outgoing edge of capacity 1, then time bound is reduced to

$$O(|E||V|^{1/2}).$$

- The **min-cost flow** problem is to find, among all maximum flows, the one flow of minimum cost provided that each edge has a cost per unit of flow.

6.6 Minimum Spanning Tree

[Definition] A *spanning tree* of a graph G is a tree which consists of $V(G)$ and a subset of $E(G)$

Note :

- The minimum spanning tree is a tree since it is acyclic, the number of edges is $|V| - 1$
- It is minimum for the total cost of edges is minimized.
- It is spanning because it covers every vertex.
- A minimum spanning tree exists if G is connected.
- Adding a non-tree edge to a spanning tree, we obtain a cycle.

Greedy Method

Make the best decision for each stage, under the following constraints :

- we must use only edges within the graph
- we must use exactly $|V| - 1$ edges
- we may not use edges that would produce a cycle

1. Prim's Algorithm

- 在算法的任一时刻，都可以看到一个已经添加到树上的顶点集，而其余顶点尚未加到这棵树中
- 算法在每一阶段都可以通过选择边 (u, v) ，使得 (u, v) 的值是所有 u 在树上但 v 不在树上的边的值中的最小者，而找出一个新的顶点并把它添加到这棵树中

2. Kruskal's Algorithm

- 连续地按照最小的权选择边，并且当所选的边不产生环时就把它作为取定的边

```
1 void Kruskal( Graph G )
2 {
3     T = { };
4     while ( T contains less than |V|-1 edges && E is
5         not empty )
6     {
```

```

6         choose a least cost edge (v, w) from E;
        /*DeleteMin*/
7         delete (v, w) from E;
8         if ( (v, w) does not create a cycle in T )
9             add (v, w) to T; /*Union/Find*/
10        else
11            discard (v, w);
12    }
13    if ( T contains fewer than |V|-1 edges )
14        Error( "No spanning tree" );
15 }

```

```

1 void kruskal(Graph G)
2 {
3     int EdgesAccepted;
4     DisjSet S;
5     PriorityQueue H;
6     Vertex U, V;
7     SetType Uset, Vset;
8     Edge E;
9
10    Initialize(S);
11    ReadGraphIntoHeapArray(G, H);
12    BuildHeap(H);
13
14    EdgesAccepted = 0;
15    while(EdgesAccepted < NumVertex-1)
16    {
17        E = DeleteMin(H); /*E = (U,V)*/
18        Uset = Find(U, S);
19        Vset = Find(V, S);
20        if(Uset != Vset)
21        {
22            /*Accept the edge*/
23            EdgesAccepted++;
24            SetUnion(S, Uset, Vset);
25        }
26    }
27 }

```

$$T = O(|E| \log |E|)$$

1-6 Let M be the minimum spanning tree of a weighted graph G. Then the path in M between V1 and V2 must be the shortest path between them in G. (2分)

☐ T ☒ F

评测结果: 答案正确 (2分)

6.7 Applications of Depth-First Search

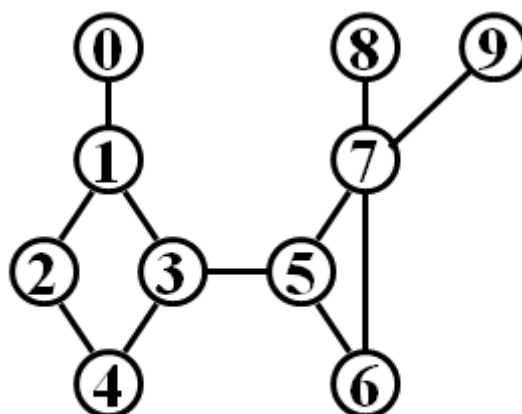
```
1  /*a generalization of preorder traversal*/
2  void DFS(vertex v)
3  {
4      visited[ v ] = true; /*mark this vertex to avoid cycles*/
5      for ( each w adjacent to v )
6          if ( !visited[ w ] ) DFS( w );
7  } /*T = O(|E|+|V|) as long as adjacency lists are used*/
```

Undirected Graphs

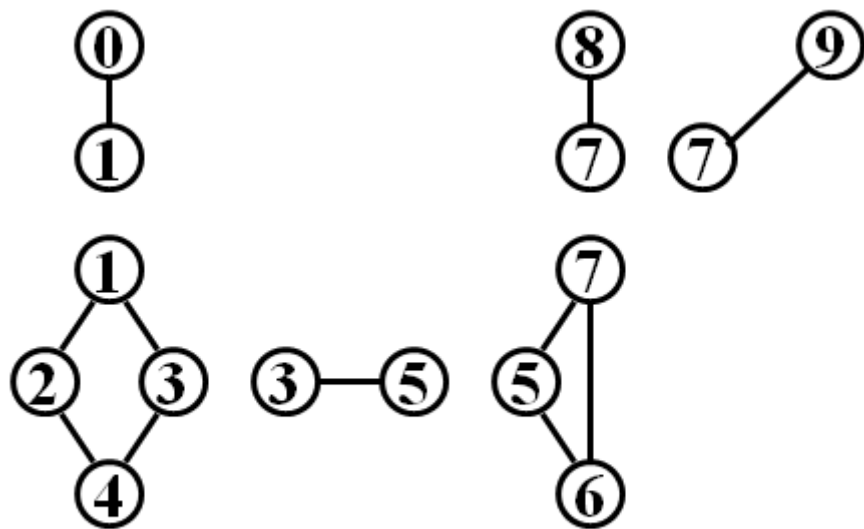
```
1  void ListComponents(Graph G)
2  {
3      for ( each v in G )
4          if ( !visited[ v ] )
5              {
6                  DFS( v );
7                  printf("\n");
8              }
9  }
```

Biconnectivity

- v is an **articulation point** if $G' = DeleteVertex(G, v)$ has **at least 2** connected components.
- G is a **biconnected graph** if G is connected and has no articulation points.
- A **biconnected component** is a maximal biconnected subgraph.



Connected graph

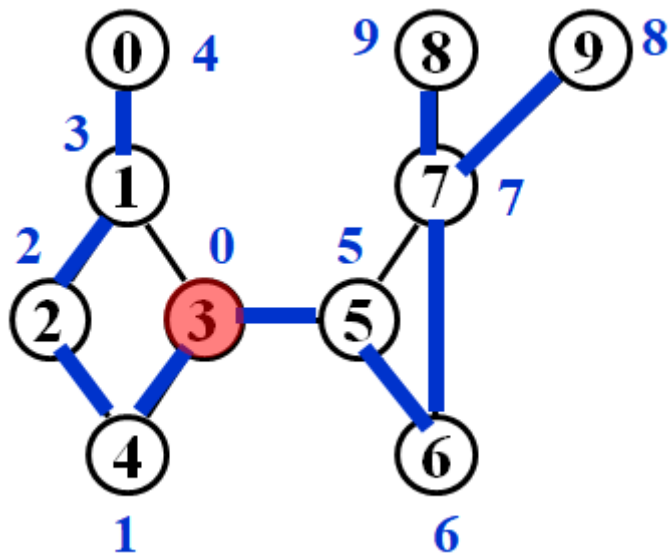


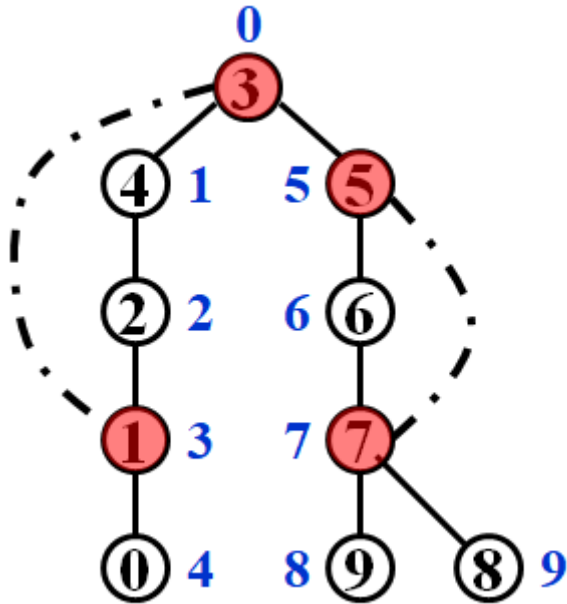
Biconnected components

Note : No edges can be shared by two or more biconnected components.
Hence $E(G)$ is partitioned by the biconnected components of G .

Finding the **biconnected components** of a connected undirected G :

- Use **depth first search** to obtain a spanning tree of G





- Depth first number(Num) 先序编号
- Back edges(背向边) = $(u, v) \notin \text{tree}$ and u is an ancestor of v .

Note : If u is an ancestor of v , then $Num(u) < Num(v)$.

- Find the articulation points in G
 - The root is an articulation point if it has at least 2 children.
 - Any other vertex u is an articulation point if u has at least 1 child, and it is impossible to move down at least 1 step and then jump up to u 's ancestor
- 对于深度优先搜索生成树上的每一个顶点 u , 计算编号最低的顶点, 称之为 $Low(u)$

$$Low(u) = \min\{Num(u), \min\{Low(w) | w \text{ is a child of } u\}, \min\{Num(w) | (u, w) \text{ is a back edge}\}\}$$

```

1  /*Assign Num and compute Parents*/
2  void AssignNum(Vertex v)
3  {
4      Vertex w;
5      Num[v] = Counter++;
6      visited[v] = True;
7      for each w adjacent to v
8          if(!visited[w])
9              {
10                 Parent[w] = v;
11                 AssignNum(w);
12             }
13 }

```

```

1  /*Assign Low; also check for articulation points*/
2  void AssignLow(Vertex V)
3  {
4      Vertex W;
5      Low[V] = Num[V]; /*Rule 1*/
6      for each w adjacent to v
7      {
8          if(Num[W] > Num[V]) /*Forward edge*/
9          {
10             Assignlow(W);
11             if(Low[W] >= Num[V])
12                 printf("%v is an articulation point\n", v);
13             Low[V] = Min(Low[V], Low[W]); /*Rule 3*/
14         }
15         else
16             if (Parent[V] != W) /*Back edge*/
17                 Low[V] = Min(Low[V], Num[W]); /*Rule 2*/
18     }
19 }

```

```

1  void FindArt(Vertex V)
2  {
3      Vertex W;
4      visited[V] = True;
5      Low[V] = Num[V] = Counter++; /*Rule 1*/
6      for each w adjacent to v
7      {
8          if(!Visited[W]) /*Forward edge*/
9          {
10             Parent[W] = V;
11             FindArt(W);
12             if(Low[W] >= Num[V])
13                 printf("%v is an articulation point\n", v);
14             Low[V] = Min(Low[V], Low[W]); /*Rule 3*/
15         }
16         else
17             if(Parent[ V ] != W) /*Back edge*/
18                 Low[V] = Min(Low[V], Num[W]); /*Rule 2*/
19     }
20 }

```

Euler Circuits

[Proposition] An Euler circuit is possible only if the graph is connected and each vertex has an *even* degree.

[Proposition] An Euler tour is possible if there are exactly *two* vertices having odd degree. One must start at one of the odd-degree vertices.

Note:

- The path should be maintained as a linked list.
- For each adjacency list, maintain a pointer to the last edge scanned.
- $T = O(|E| + |V|)$

7 Sorting

7.1 Preliminaries

```
1 void X_Sort (ElementType A[], int N)
```

- N must be a legal integer.
- Assume integer array for the sake of simplicity.
- '>' and '<' operators exist and are the only operations allowed on the input data.
- Consider internal sorting only. The entire sort can be done in main memory.

7.2 Insertion Sort

```
1 void Insertion(ElementType A[], int N)
2 {
3     int j, P;
4     ElementType Tmp;
5
6     for ( P = 1; P < N; P++ )
7     {
8         Tmp = A[ P ]; /*the next coming card*/
9         for ( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )
10             A[ j ] = A[ j - 1 ];
11         /*shift sorted cards to provide a position for the
            new coming card*/
```



```

12 |         A[ j ] = Tmp;  /*place the new card at the proper
    |         position*/
13 |     }/*end for-P-loop*/
14 | }

```

- The worst case : Input $A[]$ is in reverse order

$$T(N) = O(N^2)$$

- The best case : Input $A[]$ is in sorted order

$$T(N) = O(N)$$

7.3 A Lower Bound for Simple Sorting Algorithms

[Definition] An *inversion* in an array of numbers is any ordered pair (i, j) having the property that $i < j$ but $A[i] > A[j]$

- Swapping two adjacent elements that are out of place removes **exactly one** inversion.
- $T(N, I) = O(I + N)$ where I is the number of inversions in the original array.

[Theorem] The average number of inversions in an array of N distinct numbers is $N(N - 1)/4$

[Theorem] Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average

7.4 Shellsort

[[Example]] Sort:

	81	94	11	96	12	35	17	95	28	58	41	75	15
5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

- Define an **increment sequence** $h_1 < h_2 < \dots < h_t (h_1 = 1)$
- Define an h_k -sort at each phase for $k = t, t - 1, \dots, 1$

- 最后一轮就是Insertion Sort

Note : An h_k -sorted file that is then h_{k-1} -sorted remains h_k -sorted.

Shell's Increment Sequence

$$h_t = \lfloor N/2 \rfloor, h_k = \lfloor h_{k+1}/2 \rfloor$$

```

1 void shellsort( ElementType A[ ], int N )
2 {
3     int i, j, Increment;
4     ElementType Tmp;
5     for ( Increment = N / 2; Increment > 0; Increment /= 2 )
6         /*h sequence */
7         for ( i = Increment; i < N; i++ )
8             { /* insertion sort */
9                 Tmp = A[ i ];
10                for ( j = i; j >= Increment; j -= Increment )
11                    if( Tmp < A[ j-Increment ] )
12                        A[ j ] = A[ j-Increment ];
13                    else
14                        break;
15                A[ j ] = Tmp;
16            } /* end for-I and for-Increment loops */
17 }
```

- [Theorem] The worst-case running time of Shellsort, using Shell's increments, is $\Theta(N^2)$.

Hibbard's Increment Sequence

$$h_k = 2^k - 1$$

- [Theorem] The worst-case running time of Shellsort, using Hibbard's increments, is $\Theta(N^{3/2})$.

Conjecture

- $T_{avg-Hibbard}(N) = O(N^{5/4})$
- Sedgewick's best sequence is $\{1, 5, 19, 41, 109, \dots\}$ in which the terms are either of the form $9 \times 4^i - 9 \times 2^i + 1$ or $4^i - 3 \times 2^i + 1$. $T_{avg}(N) = O(N^{7/6})$ and $T_{worst}(N) = O(N^{4/3})$.

Conclusion

- Shellsort is a very simple algorithm, yet with an extremely complex analysis.
- It is good for sorting up to moderately large input (tens of thousands).

7.5 Heapsort

Algorithm1

```
1 void Heapsort( int N )
2 {
3     BuildHeap( H );
4     for ( i = 0; i < N; i++ )
5         TmpH[ i ] = DeleteMin( H );
6     for ( i = 0; i < N; i++ )
7         H[ i ] = TmpH[ i ];
8 }
```

$$T(N) = O(N \log N)$$

- The space requirement is doubled.

Algorithm2

```
1 void Heapsort( ElementType A[ ], int N )
2 {
3     int i;
4     for ( i = N / 2; i >= 0; i-- ) /*BuildHeap*/
5         PercDown( A, i, N );
6     for ( i = N - 1; i > 0; i-- )
7     {
8         Swap( &A[ 0 ], &A[ i ] ); /*DeleteMax*/
9         PercDown( A, 0, i );
10    }
11 }
```

- [Theorem] The average number of comparisons used to heapsort a random permutation of N distinct items is $2N \log N - O(N \log \log N)$.

Note : Although Heapsort gives **the best average time**, in practice it is slower than a version of Shellsort that uses Sedgewick's increment sequence.

7.6 Mergesort

```
1 void MSort( ElementType A[ ], ElementType TmpArray[ ], int
  Left, int Right )
2 {
3     int Center;
4     if ( Left < Right )
5     { /*if there are elements to be sort*/
6         Center = (Left+Right)/2;
7         MSort(A, TmpArray, Left, Center); /*T(N/2)*/
8         MSort(A, TmpArray, Center+1, Right); /*T(N/2)*/
9         Merge(A, TmpArray, Left, Center+1, Right); /*O(N)*/
10    }
11 }
12
13 void Mergesort( ElementType A[ ], int N )
14 {
15     ElementType *TmpArray; /*need O(N) extra space*/
16     TmpArray = malloc(N*sizeof(ElementType));
17     if (TmpArray != NULL)
18     {
19         MSort(A, TmpArray, 0, N-1);
20         free(TmpArray);
21     }
22     else FatalError("No space for tmp array!!!");
23 }
```

- If a TmpArray is declared locally for each call of Merge, then $S(N) = O(N \log N)$.

```
1 /*Lpos = start of left half, Rpos = start of right half*/
2 void Merge( ElementType A[ ], ElementType TmpArray[ ], int
  Lpos, int Rpos, int RightEnd )
3 {
4     int i, LeftEnd, NumElements, TmpPos;
5     LeftEnd = Rpos-1;
6     TmpPos = Lpos;
7     NumElements = RightEnd-Lpos+1;
8     while( Lpos <= LeftEnd && Rpos <= RightEnd ) /*main
  loop*/
9         if ( A[ Lpos ] <= A[ Rpos ] )
10             TmpArray[ TmpPos++ ] = A[ Lpos++ ];
11         else
12             TmpArray[ TmpPos++ ] = A[ Rpos++ ];
```

```

13     while( Lpos <= LeftEnd ) /*Copy rest of first half*/
14         TmpArray[ TmpPos++ ] = A[ Lpos++ ];
15     while( Rpos <= RightEnd ) /*Copy rest of second half*/
16         TmpArray[ TmpPos++ ] = A[ Rpos++ ];
17     for( i = 0; i < NumElements; i++, RightEnd-- )
18         /*Copy TmpArray back*/
19         A[ RightEnd ] = TmpArray[ RightEnd ];
20 }

```

Analysis

$$\begin{aligned}
 T(1) &= O(1) \\
 T(N) &= 2T\left(\frac{N}{2}\right) + O(N) \\
 \frac{T(N)}{N} &= \frac{T\left(\frac{N}{2}\right)}{\frac{N}{2}} + 1 \\
 &\quad \dots \\
 \frac{T\left(\frac{N}{2^{k-1}}\right)}{\frac{N}{2^{k-1}}} &= \frac{T(1)}{1} + 1 \\
 T(N) &= O(N + N \log N)
 \end{aligned}$$

Note : Mergesort requires linear extra memory, and copying an array is slow. It is hardly ever used for internal sorting, but is quite useful for external sorting.

7.7 Quicksort

- the **fastest** known sorting algorithm in practice

Algorithm

```

1 void Quicksort( ElementType A[ ], int N )
2 {
3     if (N < 2) return;
4     pivot = pick any element in A[ ];
5     Partition S = { A[ ] \ pivot } into two disjoint sets:
6     A1 = { a in S | a <= pivot } and A2 = { a in S | a >=
    pivot };
7     A = Quicksort(A1, N1) and { pivot } and Quicksort(A2, N2);
8 }

```

- The pivot is placed at the right place **once and for all**.
- 要研究的问题是如何选取枢纽元和如何划分

Picking the Pivot

A Wrong Way

- Pivot = $A[0]$
- The worst case : $A[]$ is presorted, quicksort will take $O(N^2)$ time to do nothing

A Safe Maneuver

- Pivot = random select from $A[]$
- random number generation is **expensive**

Median-of-Three Partitioning

- Pivot = median(left, center, right)
- Eliminates the bad case for sorted input and actually reduces the running time by about 5%.

Partitioning Strategy

- 当 i 在 j 的左边时, 我们将 i 右移, 移过那些小于枢纽元的元素, 并将 j 左移, 移过那些大于枢纽元的元素
- 当 i 和 j 停止时, i 指向一个大元素而 j 指向一个小元素, 如果 i 在 j 的左边, 那么将这两个元素互换
- 重复该过程直到 i 和 j 彼此交错为止
- 划分的最后一步是将枢纽元与 i 所指向的元素交换
- 如果 i 和 j 遇到等于枢纽元的键值, 就让 i 和 j 都停止, 因为若都不停止 $T(N) = O(N^2)$
- There will be many dummy swaps, but at least the sequence will be partitioned into two equal-sized subsequences.

Small Arrays

- Quicksort is slower than insertion sort for small $N(\leq 20)$.
- Cutoff when N gets small and use other efficient algorithms (such as insertion sort).

Implementation

```

1 void Quicksort( ElementType A[ ], int N )
2 {
3     Qsort( A, 0, N-1 );
4     /*A:the array*/
5     /*0:Left index*/
6     /*N-1:Right index*/
7 }

```

```

1 /* Return median of Left, Center, and Right */
2 /* Order these and hide the pivot */
3 ElementType Median3( ElementType A[ ], int Left, int Right )
4 {
5     int Center = ( Left+Right )/2;
6     if ( A[ Left ] > A[ Center ] )
7         Swap( &A[ Left ], &A[ Center ] );
8     if ( A[ Left ] > A[ Right ] )
9         Swap( &A[ Left ], &A[ Right ] );
10    if ( A[ Center ] > A[ Right ] )
11        Swap( &A[ Center ], &A[ Right ] );
12    /*Invariant: A[ Left ] <= A[ Center ] <= A[ Right ]*/
13    Swap( &A[ Center ], &A[ Right-1 ] ); /*Hide pivot*/
14    /*only need to sort A[ Left+1 ] ... A[ Right-2 ]*/
15    return A[ Right-1 ]; /*Return pivot*/
16 }

```

```

1 void Qsort( ElementType A[ ], int Left, int Right )
2 {
3     int i, j;
4     ElementType Pivot;
5     if ( Left + Cutoff <= Right )
6     { /*if the sequence is not too short*/
7         Pivot = Median3( A, Left, Right ); /*select pivot*/
8         i = Left;
9         j = Right - 1; /*why not set Left+1 and Right-2?*/
10        for( ; ; )
11        {
12            while ( A[ ++i ] < Pivot ) { } /*scan from
left*/
13            while ( A[ --j ] > Pivot ) { } /*scan from
right*/
14            if ( i < j )
15                Swap( &A[ i ], &A[ j ] ); /*adjust
partition*/
16            else break; /*partition done*/

```

```

17     }
18     Swap( &A[ i ], &A[ Right-1 ] ); /*restore pivot */
19     Qsort( A, Left, i-1 );    /*recursively sort left
part*/
20     Qsort( A, i+1, Right );    /*recursively sort right
part*/
21 } /*end if - the sequence is long*/
22 else /*do an insertion sort on the short subarray*/
23     InsertionSort( A+Left, Right-Left+1 );
24 }

```

Note : If set $i = \text{Left} + 1$ and $j = \text{Right} - 2$, there will be an infinite loop if $A[i] = A[j] = \text{pivot}$.

Analysis

$$T(N) = T(i) + T(N - i - 1) + cN$$

- i is the number of the elements in S_1 .
- **The Worst Case**

$$T(N) = T(N - 1) + cN$$

$$T(N - 1) = T(N - 2) + c(N - 1)$$

...

$$T(2) = T(1) + 2c$$

$$T(N) = T(1) + c \sum_{i=2}^N i = O(N^2)$$

- **The Best Case**

$$T(N) = 2T(N/2) + cN$$

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c$$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + c$$

...

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$

$$\frac{T(N)}{N} = \frac{T(1)}{1} + c \log N \quad \frac{T(N)}{N} = \frac{T(1)}{1} + c \log N$$

$$T(N) = cN \log N + N = O(N \log N)$$

- **The Average Case**

- Assume the average value of $T(i)$ for any i is $\frac{1}{N} \left[\sum_{j=0}^{N-1} T(j) \right]$

$$T(N) = \frac{2}{N} \left[\sum_{j=0}^{N-1} T(j) \right] + cN$$

$$NT(N) = 2 \left[\sum_{j=0}^{N-1} T(j) \right] + cN^2$$

$$(N-1)T(N-1) = 2 \left[\sum_{j=0}^{N-2} T(j) \right] + c(N-1)^2$$

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c$$

$$NT(N) = (N+1)T(N-1) + 2cN$$

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1}$$

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N}$$

...

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{N+1} \frac{1}{i}$$

$$T(N) = O(N \log N)$$

Quickselect

- 查找第 K 最大(最小)元

```

1  /*Places the kth smallest element in the kth position*/
2  /*Because arrays start at 0, this will be index k-1*/
3  void Qselect(ElementType A[ ], int k, int Left, int Right)
4  {
5      int i, j;
6      ElementType Pivot;
7
8      if (Left + Cutoff <= Right)
9      {
10         Pivot = Median3(A, Left, Right);
11         i = Left;
12         j = Right-1;
13         for( ; ; )
14         {
15             while(A[ ++i ] < Pivot){ }

```

```

16         while(A[ --j ] > Pivot){ }
17         if(i < j)
18             swap(&A[ i ], &A[ j ]);
19         else
20             break;
21     }
22     swap(&A[ i ], &A[ Right-1 ]); /*Restore pivot*/
23
24     if(k <= i)
25         Qselect(A, k, Left, i-1);
26     else if (k > i+1)
27         Qselect(A, k, i+1, Right);
28 }
29 else /*Do an insertion sort on the subarray*/
30     InsertionSort(A+Left, Right-Left+1);
31 }

```

2-1 During the sorting, processing every element which is not yet at its final position is called a "run". Which of the following cannot be the result after the second run of quicksort?

- ☐ A. 5, 2, 16, 12, 28, 60, 32, 72
- ☐ B. 2, 16, 5, 28, 12, 60, 32, 72
- ☒ C. 2, 12, 16, 5, 28, 32, 72, 60
- ☐ D. 5, 2, 12, 28, 16, 32, 72, 60

2-1 答案错误 (0分) 创建提问

正确答案是D

7.8 Sorting Large Structures

- Swapping large structures can be very much expensive.
- Add a pointer field to the structure and swap pointers instead – **indirect sorting**. Physically rearrange the structures at last if it is really necessary.
- Table Sort

7.9 A General Lower Bound for Sorting

[Theorem] Any algorithm that *sorts by comparisons only* must have a worst case computing time of $\Omega(N \log N)$.

- When sorting N distinct elements, there are $N!$ different possible results.
- Thus any decision tree must have at least $N!$ leaves.
- If the height of the tree is k , then $N! \leq 2^{k-1} \rightarrow k \geq \log(N!) + 1$
- Since $N! \geq (N/2)^{N/2}$ and $\log_2 N! \geq (N/2) \log_2 (N/2) = \Theta(N \log_2 N)$
- Therefore $T(N) = k \geq c \cdot N \log_2 N$

7.10 Bucket Sort

[[Example]] Suppose that we have N students, each has a grade record in the range 0 to 100 (thus there are $M = 101$ possible distinct grades). How to sort them according to their grades in **linear** time?

```

1 Algorithm
2 {
3     initialize count[ ];
4     while(read in a student's record)
5         insert to list count[stdnt.grade];
6     for(int i = 0; i < M; i++)
7     {
8         if(count[i]) output list count[i];
9     }
10 }
```

$$T(N, M) = O(M + N)$$

7.11 Radix Sort

[[Example]] Given $N = 10$ integers in the range 0 to 999 ($M = 1000$)
Is it possible to sort them in **linear** time?

Input: 64, 8, 216, 512, 27, 729, 0, 1, 343, 125

Sort according to the **Least Significant Digit** first.

Bucket	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729
Pass 2	0	512	125		343		64			
	1	216	27							
	8		729							
Pass 3	0	125	216	343		512		729		
	1									
	8									
	27									
	64									

Output: 0, 1, 8, 27, 64, 125, 216, 343, 512, 729

- $T = O(P(N + B))$ where P is the number of passes, N is the number of elements to sort, and B is the number of buckets.

MSD(Most Significant Digit) Sort and LSD(Least Significant Digit) Sort

[[Example]] A deck of cards sorted on 2 keys

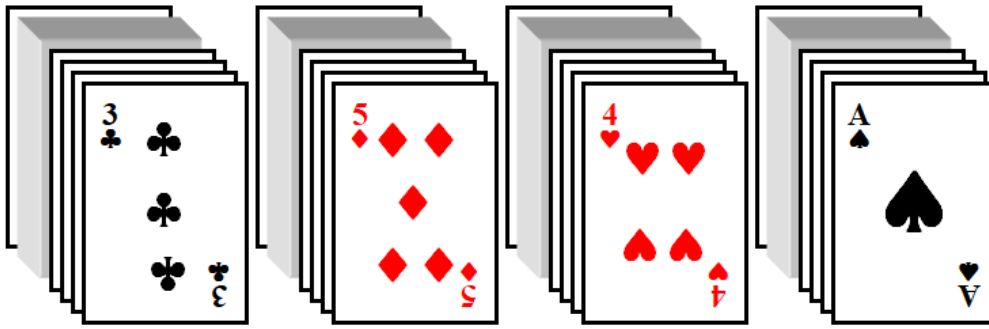
K^0 [Suit] ♣ < ♦ < ♥ < ♠

K^1 [Face value] 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

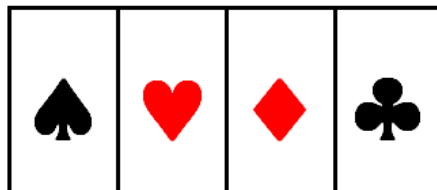
Sorting result: 2♣ ... A♣ 2♦ ... A♦ 2♥ ... A♥ 2♠ ... A♠

☞ MSD (Most Significant Digit) Sort

① Sort on K^0 : for example, create 4 buckets for the suits

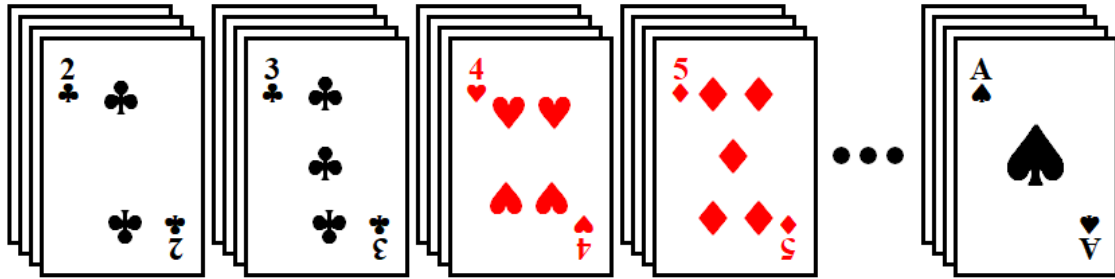


② Sort each bucket independently (using any sorting technique)



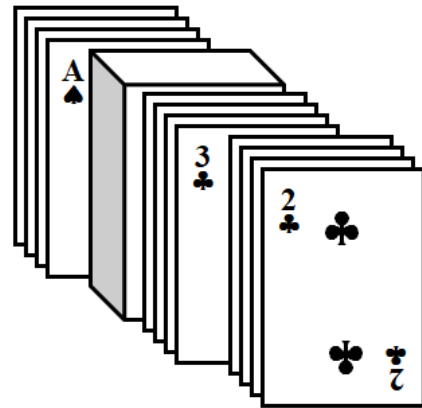
☞ LSD (Least Significant Digit) Sort

① Sort on K^1 : for example, create 13 buckets for the face values



② Reform them into a single pile

③ Create 4 buckets and resort



-
- 稳定的排序算法：冒泡排序、插入排序、归并排序、基数排序
 - 不稳定的排序算法：选择排序、快速排序、希尔排序、堆排序
-

8 Hashing

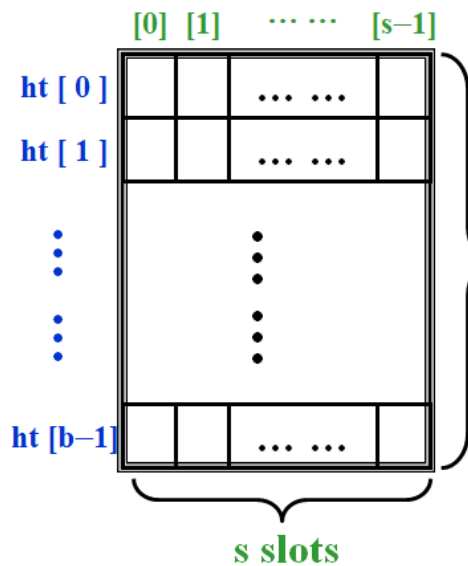
8.1 General Idea

Symbol Table (== Dictionary) ::= { < name, attribute > }

Symbol Table ADT


- **Objects** : A set of name-attribute pairs, where the names are unique
- **Operations** :
 - SymTab Create(TableSize)
 - Boolean IsIn(symtab, name)
 - Attribute Find(symtab, name)
 - SymTab Insert(symtab, name, attr)
 - SymTab Delete(symtab, name)


Hash Tables



For each identifier x , we define a **hash function**
 $f(x) = \text{position of } x \text{ in } ht[]$ (i.e. the index of the bucket that contains x)

b buckets

 $T ::= \text{total number of distinct possible values for } x$

 $n ::= \text{total number of identifiers in } ht[]$

 **identifier density** $::= n / T$

 **loading density** $\lambda ::= n / (s b)$

- A **collision** occurs when we hash two nonidentical identifiers into the same bucket.
- An **overflow** occurs when we hash a new identifier into a full bucket.

8.2 Hash Function

- $f(x)$ must be **easy** to compute and **minimize** the number of **collisions**.
- $f(x)$ should be unbiased. For any x and any i , we have that $\text{Probability}(f(x) = i) = \frac{1}{b}$. Such kind of a hash function is called a **uniform hash function**.

8.3 Separate Chaining

- keep a list of all keys that hash to the same value

```
1 struct ListNode;  
2 typedef struct ListNode *Position;  
3 struct HashTbl;  
4 typedef struct HashTbl *HashTable;  
5 struct ListNode {  
6     ElementType Element;  
7     Position Next;  
8 };  
9 typedef Position List;
```

```

10  /* List *TheList will be an array of lists, allocated later */
11  /* The lists use headers (for simplicity), */
12  /* though this wastes space */
13  struct HashTbl {
14      int TableSize;
15      List *TheLists;
16  };

```

Create an empty table

```

1  HashTable InitializeTable( int TableSize )
2  {
3      HashTable H;
4      int i;
5      if ( TableSize < MinTableSize )
6      {
7          Error( "Table size too small" );
8          return NULL;
9      }
10     H = malloc( sizeof( struct HashTbl ) ); /*Allocate
table*/
11     if ( H == NULL ) FatalError( "Out of space!!!" );
12     H->TableSize = NextPrime( TableSize ); /*Better be
prime*/
13     H->TheLists = malloc( sizeof( List ) * H->TableSize );
/*Array of lists*/
14     if ( H->TheLists == NULL ) FatalError( "Out of space!!!"
);
15     H->TheList = malloc(H->TableSize*sizeof(struct
ListNode));
16     for( i = 0; i < H->TableSize; i++ )
17     { /*Allocate list headers*/
18         //H->TheLists[ i ] = malloc( sizeof( struct ListNode )
); /* slow! */
19         if ( H->TheLists[ i ] == NULL ) FatalError( "Out of
space!!!" );
20         else H->TheLists[ i ]->Next = NULL;
21     }
22     return H;
23 }

```

Find a key from a hash table

```
1 Position Find ( ElementType Key, HashTable H )
2 {
3     Position P;
4     List L;
5     L = H->TheLists[ Hash( Key, H->TableSize ) ];
6     P = L->Next;
7     while( P != NULL && P->Element != Key ) /*Probably need
8     strcmp*/
9         P = P->Next;
10    return P;
11 }
```

Insert a key into a hash table

```
1 void Insert ( ElementType Key, HashTable H )
2 {
3     Position Pos, NewCell;
4     List L;
5     Pos = Find( Key, H );
6     if ( Pos == NULL )
7     { /*Key is not found, then insert*/
8         NewCell = malloc( sizeof( struct ListNode ) );
9         if ( NewCell == NULL ) FatalError( "Out of space!!!"
10     );
11         else
12         {
13             L = H->TheLists[ Hash( Key, H->TableSize ) ];
14             /*Compute again is bad*/
15             NewCell->Next = L->Next;
16             NewCell->Element = Key; /*Probably need strcpy!*/
17             L->Next = NewCell;
18         }
19     }
20 }
```

Note : Make the TableSize about as large as the number of keys expected (i.e. to make the loading density factor $\lambda \approx 1$).

8.4 Open Addressing

- find another empty cell to solve collision(avoiding pointers)

```
1 Algorithm: insert key into an array of hash table
2 {
3     index = hash(key);
4     initialize i = 0 ----- the counter of probing;
5     while (collision at index)
6     {
7         index = (hash(key)+f(i))%TableSize; /*f(i) is
collision resolving function*/
8         if (table is full) break;
9         else i++;
10    }
11    if (table is full) ERROR ("No space left");
12    else insert key at index;
13 }
```

Note : Generally $\lambda < 0.5$.

Linear Probing

- $F(i)$ is a linear function of i , such as $F(i) = i$.
- 逐个探测每个单元(必要时可以绕回)以查找出一个空单元
- 使用线性探测的预期探测次数对于插入和不成功的查找来说大约是 $\frac{1}{2}(1 + \frac{1}{(1-\lambda)^2})$, 对于成功的查找来说是 $\frac{1}{2}(1 + \frac{1}{1-\lambda})$
- Cause **primary clustering** : any key that hashes into the cluster will add to the cluster after several attempts to resolve the collision.

Quadratic Probing

- $F(i)$ is a quadratic function of i , such as $F(i) = i^2$.

[Theorem] If quadratic probing is used, and the table size is *prime*, then a new element can always be inserted if the table is *at least half empty*.

Proof: Just prove that the first $\lfloor \text{TableSize}/2 \rfloor$ alternative locations are all **distinct**. That is, for any $0 < i \neq j \leq \lfloor \text{TableSize}/2 \rfloor$, we have

$$(h(x) + i^2) \% \text{TableSize} \neq (h(x) + j^2) \% \text{TableSize}$$

Suppose: $h(x) + i^2 = h(x) + j^2 \pmod{\text{TableSize}}$

then: $i^2 = j^2 \pmod{\text{TableSize}}$

$$(i+j)(i-j) = 0 \pmod{\text{TableSize}}$$

TableSize is prime \Rightarrow either $(i+j)$ or $(i-j)$ is divisible by **TableSize**

Contradiction !

For any x , it has $\lceil \text{TableSize}/2 \rceil$ distinct locations into which it can go.

If at most $\lfloor \text{TableSize}/2 \rfloor$ positions are taken, then an empty spot can always be found.

Note : If the table size is a prime of the form $4k + 3$, then the quadratic probing $f(i) = \pm i^2$ can probe the entire table.

```

1  HashTable InitializeTable(int TableSize)
2  {
3      HashTable H;
4      int i;
5      if(TableSize < MinTableSize)
6      {
7          Error("Table size too small");
8          return NULL;
9      }
10     /*Allocate table*/
11     H = malloc(sizeof(struct HashTbl));
12     if(H == NULL)
13         Fatal Error("Out of space!!!");
14     H->TableSize = NextPrime(TableSize);
15
16     /*Allocate array of Cells*/
17     H->TheCells = malloc(sizeof(Cell)*H->TableSize);
18     if(H->TheCells == NULL)
19         FatalError("Out of space!!!");
20
21     for(i = 0; i < H->TableSize; i++)
22         H->TheCells[ i ].Info = Empty;
23     return H;
24 }
```

```

1  Position Find(ElementType Key, HashTable H)
2  {
3      Position CurrentPos;
```

```

4     int CollisionNum;
5     CollisionNum = 0;
6     CurrentPos = Hash(Key, H->TableSize);
7     while(H->TheCells[ CurrentPos ].Info != Empty &&
8           H->TheCells[ CurrentPos ].Element != Key)
9     {
10        CurrentPos += 2*++CollisionNum-1;
11        if (CurrentPos >= H->TableSize)
12            CurrentPos -= H->TableSize;    /*Faster than mod*/
13    }
14    return CurrentPos;
15 }

```

```

1 void Insert(ElementType Key, HashTable H)
2 {
3     Position Pos;
4     Pos = Find(Key, H);
5     if (H->TheCells[ Pos ].Info != Legitimate)
6     { /*OK to insert here*/
7         H->TheCells[ Pos ].Info = Legitimate;
8         H->TheCells[ Pos ].Element = Key; /*Probably need
9         strcpy*/
10    }

```

Note :

- Insertion will be seriously slowed down if there are too many **deletions intermixed with insertions**.
- Although primary clustering is solved, **secondary clustering** occurs, that is, keys that hash to the same position will probe the same alternative cells.

Double Hashing

- $f(i) = i * hash_2(x)$
- $hash_2(x) \neq 0$
- make sure that all cells can be probed
- $hash_2(x) = R - (x \% R)$ with R a prime smaller than TableSize, will work well.

Note :

- If double hashing is correctly implemented, simulations imply that the **expected** number of probes is almost the same as for a **random**

collision resolution strategy.

- Quadratic probing does not require the use of a second hash function and is thus likely to be **simpler and faster** in practice.

8.5 Rehashing

- Build another table that is about twice as big.
- Scan down the entire original hash table for non-deleted elements.
- Use a new function to hash those elements into the new table.
- When to rehash
 - As soon as the table is half full
 - When an insertion fails
 - When the table reaches a certain load factor

Note : Usually there should have been $N/2$ insertions before rehash, so $O(N)$ rehash only adds a constant cost to each insertion. However, in an interactive system, the unfortunate user whose insertion caused a rehash could see a slowdown.

```
1  HashTable Rehash(HashTable H)
2  {
3      int i, OldSize;
4      cell *oldCells;
5      oldCells = H->TheCells;
6      OldSize = H->TableSize;
7
8      /*Get a new, empty table*/
9      H = InitializeTable(2*OldSize);
10     /*Scan through old table, reinserting into new*/
11     for(i = 0; i < OldSize; i++)
12         if(OldCells[i].Info == Legitimate)
13             Insert(OldCells[i].Element, H);
14     free(OldCells);
15
16     return H;
17 }
```

