Algorithm Analysis

Attribute

- Input
- Output
- Definiteness
- Finiteness
- Effectiveness

Notation

$$T(N) = O(f(N)) : for N \geq n_0, T(N) \leq c \cdot f(N)$$
$$T(N) = \Omega(f(N)) : for N \geq n_0, T(N) \geq c \cdot f(N)$$
$$T(N) = \Theta(f(N)) : T(N) = O(f(N)) \& T(N) = \Omega(f(N))$$
$$T(N) = o(f(N)) : T(N) = O(f(N)) \& T(N) \neq \Theta(f(N))$$

Checking

1. function, like:

$$T(N) = O(N^2), \frac{T(2N)}{T(N)} \approx 4$$

2. limit:

$$T(N) = O(f(N)), \lim_{N \to \infty} \frac{T(N)}{f(N)} \approx Constant$$

List

Data Type

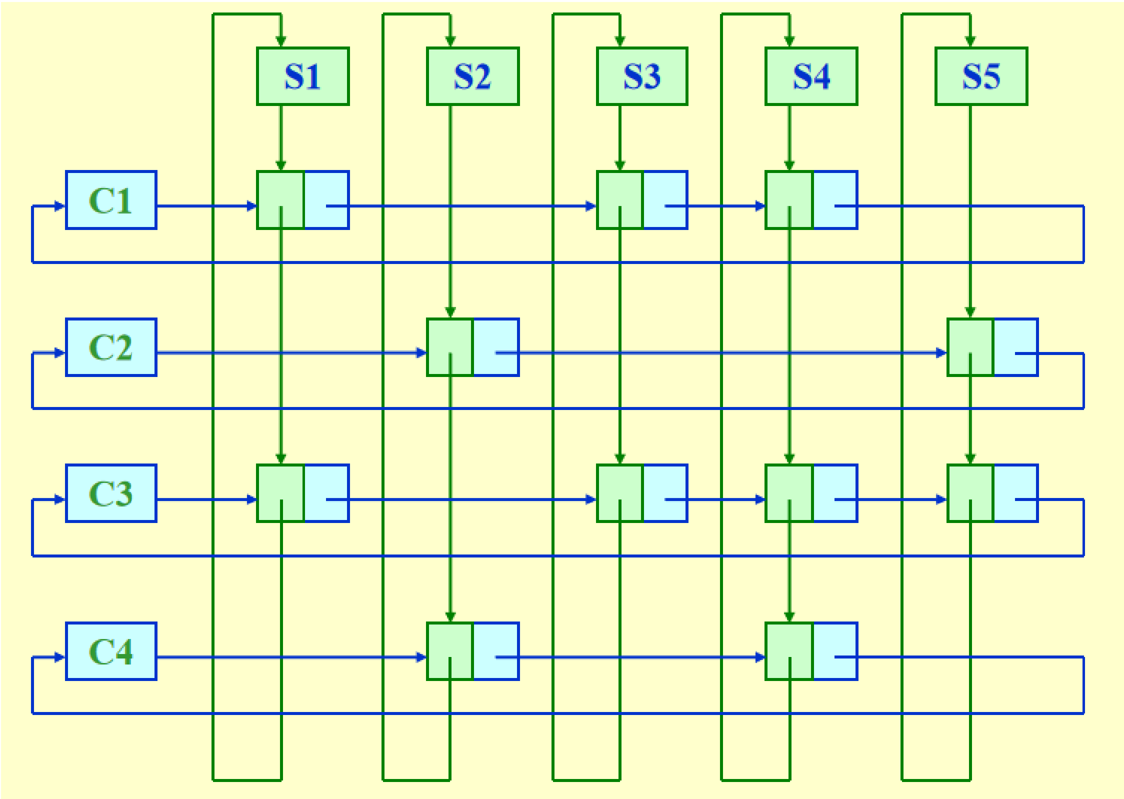
$$\{Objects\} \bigcup \{Operations\}$$

Implementation

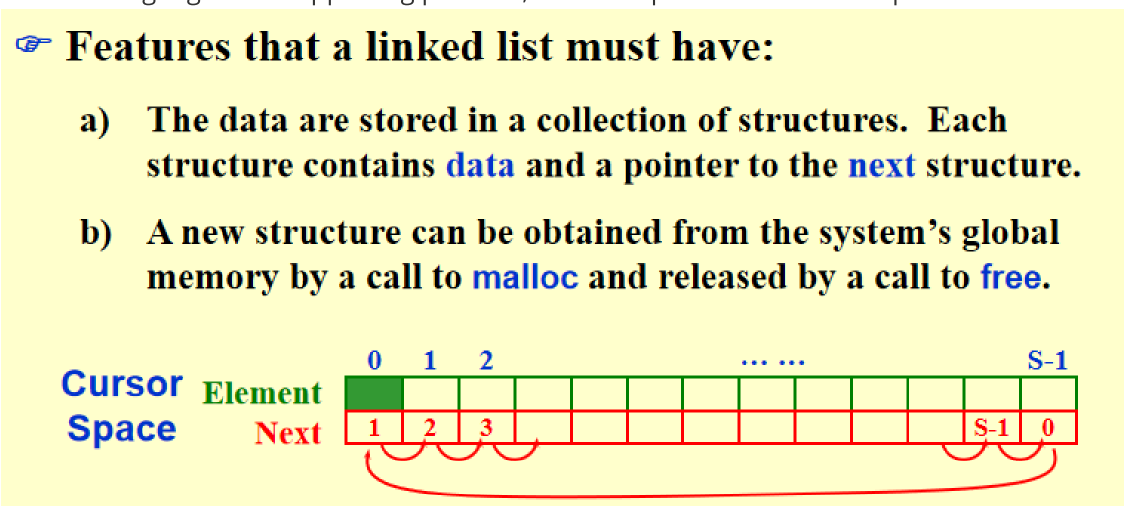
1. Simple array using **sequential mapping**
   - easy for random access
   - **limited size**
   - **inconvenient for insertion and deletion**
2. Linked lists
   - easy to insert and delete(with a dummy head)

- flexible size
- **linear search to find the k^th element**

3. Doubly linked circular list
    - a list that is more convenient for random access
4. Sparse matrix



5. Cursor implementation
   In some languages not supporting pointers, cursor implementation is adopted.



> Usually, the implementation depends on the **desired operations**.

Stack

FILO

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

Infix to Postfix

- Add parentheses if necessary
- If a letter is read, output the letter
- If a operand is read:
  - If the stack is empty, push the operand
  - If the last operand $\geq$ current operand, output the stack and then push the current
  - If a right parenthesis is read, output all the operands between the parentheses
  - If the end is reached, output all the operands
- Observe that when `(` is not in the stack, its precedence is the highest; but when it is in the stack, its precedence is the lowest. Define in-stack precedence and incoming precedence for symbols, and each time use the corresponding precedence for comparison.

Queue

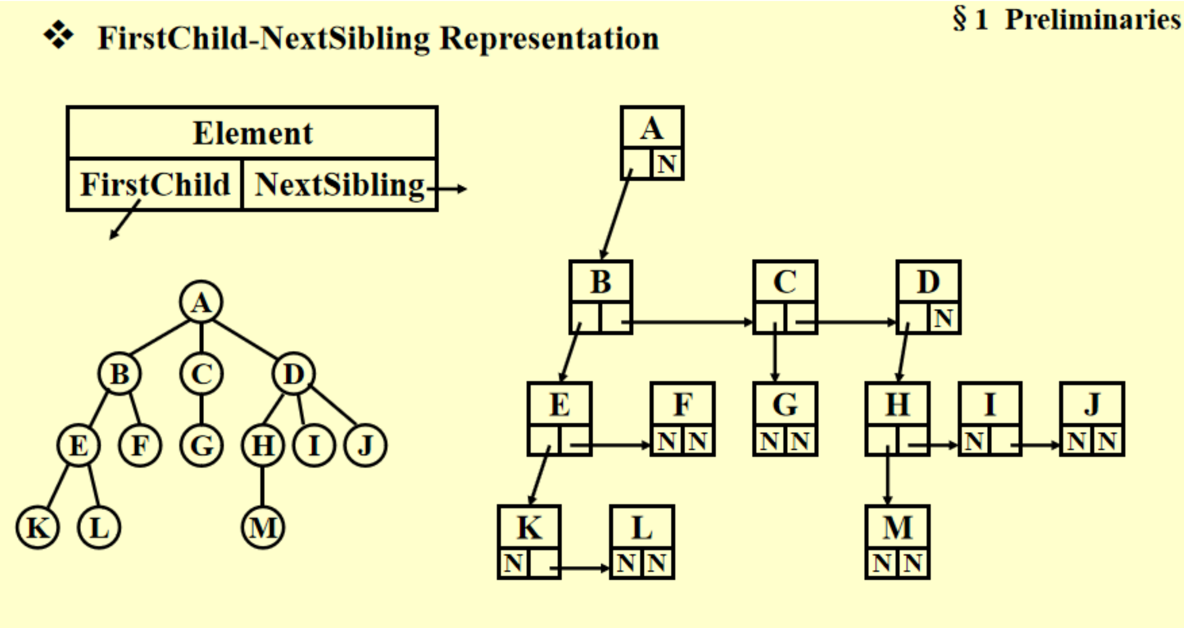
FIFO

1. Linked list
2. Array implementation
   To differentiate FULL and EMPTY:
   - one space is abandoned, that is, EMPTY means font == rear+1 and no elements; FULL means font == rear+2 and N-1 elements
   - add a SIZE field filed to signal the size
   - add a FULL boolean variable

Tree

- degree of a node: **number of subtrees** of the node. For example, degree(A) = 3, degree(F) = 0
- degree of a node: max degree of all the nodes
- depth of n_i: length of the unique path from the root to n_i. Depth(root) = 0.
- height of n_i: length of the longest path from n_i to a leaf. Height(leaf) = 0
- level: level(root)=1
- **Right** and **Left** matters!!

FirstChild-NextSibling Representation



Traversal

Preorder

- Use stack implicitly. (System stack)
- visit the node before a push
- depth-first search

Postorder

- Use stack implicitly. (System stack)
- visit the node after a push

Inorder

- visit the node right after the first push(if left child exists)

Level order

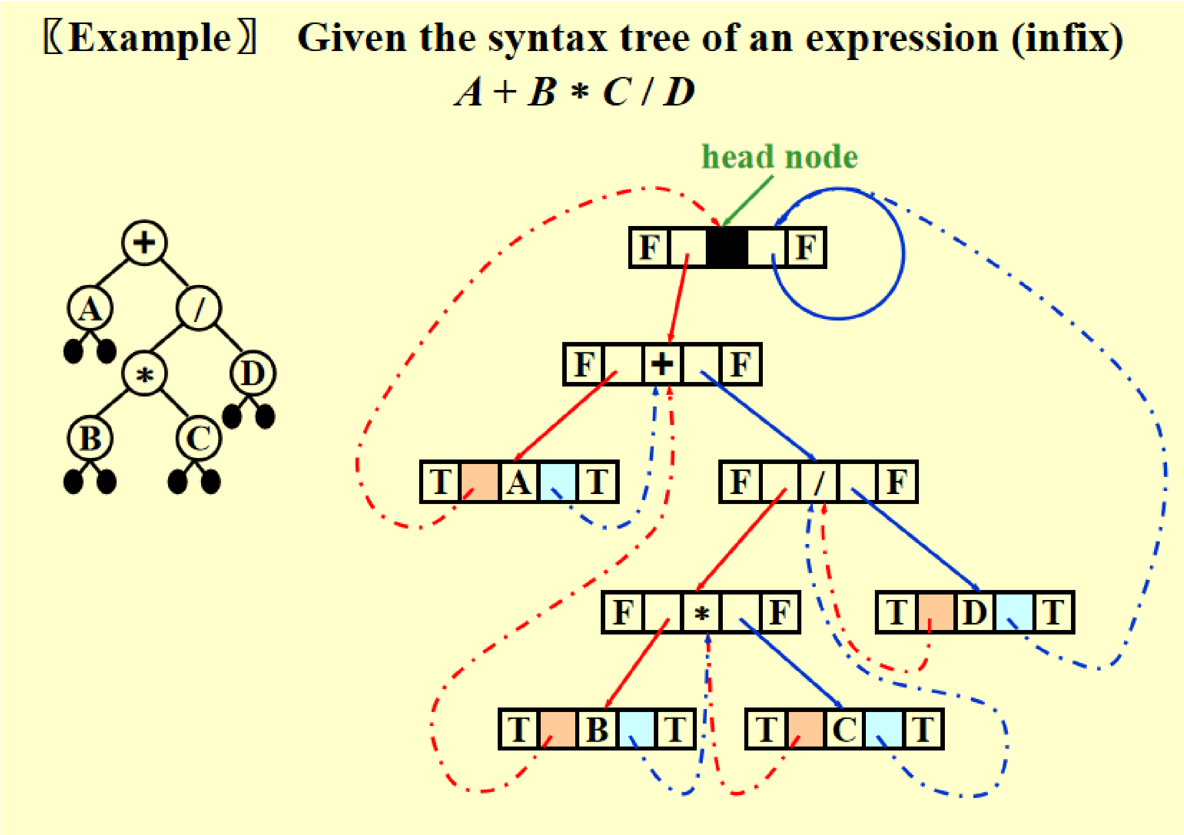
- use a queue instead of a stack
- visit a node and then push its children until the traversal ends
- breadth-first search

Threaded Binary Tree

Rule 1: If Tree->Left is null, replace it with a pointer to the **inorder** predecessor of Tree.

Rule 2: If Tree->Right is null, replace it with a pointer to the **inorder** successor of Tree.

Rule 3: **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.



Binary Search Tree

☑ 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 the number of nodes of degree 2.

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

1. Every node has a key which is an integer, and the keys are **distinct**.
2. The keys in a nonempty left subtree must be **smaller** than the key in the root of the subtree.
3. The keys in a nonempty right subtree must be **larger** than the key in the root of the subtree.
4. The left and right **subtrees** are also binary search trees.
5. From the above, the leftmost element is the smallest; the rightmost element is the largest.

```
Position Find(ElementType X, Tree T)
{
    while(T)
    {
        if(X == T->Element)
        {
            return T;
```

```
        }
        else if(X < T->Element)
        {
            T = T->Left;
        }
        else
        {
            T = T->Right;
        }
    }
    return NULL;
}
```

```
Tree Insert(ElementType X, Tree T)
{
    if(T == NULL)
    {
        T = malloc(sizeof(struct treenode));
        if(T == NULL)
        {
            Error();
        }
        else
        {
            T->Elment = X;
            T->Left = T->Right = NULL;
        }
    }
    else
    {
        if(X < T->Element)
        {
            T->Left = Insert(X, T->Left);
        }
        else if(X > T->Element)
        {
            T->Right = Insert(X, T->Right);
        }
        // else X == T->Element, do nothing
    }
    return T;
}
```

```
SearchTree  Delete( ElementType X, SearchTree T )
{    Position  TmpCell;
      if ( T == NULL )   Error();
      else  if ( X < T->Element )  /* Go left */
        T->Left = Delete( X, T->Left );
      else  if ( X > T->Element )  /* Go right */
        T->Right = Delete( X, T->Right );
      else  /* Found element to be deleted */
         if ( T->Left && T->Right ) {  /* Two children */
```

```
                /* Replace with smallest in right subtree */
                TmpCell = FindMin( T->Right );
                T->Element = TmpCell->Element;
                T->Right = Delete( T->Element, T->Right );
            } /* End if */
            else {  /* One or zero child */
                TmpCell = T;
                if ( T->Left == NULL ) /* Also handles 0 child */
                    T = T->Right;
                else  if ( T->Right == NULL )
                    T = T->Left;
                free( TmpCell );
            }  /* End else 1 or 0 child */
        return  T;
}
```

Heap (Priority Queue)

Perfect Binary Tree:

A tree of height h has $2^{h+1} - 1$ nodes.

Complete Binary Tree:

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

Then, in an array representing the complete binary tree sequentially, for a node with index i:

$$parent(i) = \begin{cases} \lfloor \frac{i}{2} \rfloor & i \neq 1 \\ none & i = 1 \end{cases}$$

$$leftchild(i) = \begin{cases} 2i & 2i \leq n \\ none & 2i > n \end{cases}$$

$$rightchild(i) = \begin{cases} 2i+1 & 2i+1 \leq n \\ none & 2i+1 > n \end{cases}$$

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

Order property

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.

height of the heap: $h = log\lfloor N \rfloor$

```
void  Insert( ElementType  X,  PriorityQueue  H )
{
    int  i;
```

```c
    if ( IsFull( H ) ) {
        Error( "Priority queue is full" );
        return;
    }

    // percolate up
    for ( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
        H->Elements[ i ] = H->Elements[ i / 2 ];

    // faster than swap in the loop
    H->Elements[ i ] = X;
}
```

```c
ElementType  DeleteMin( PriorityQueue  H )
{
    int  i, Child;
    ElementType  MinElement, LastElement;
    if ( IsEmpty( H ) ) {
         Error( "Priority queue is empty" );
         return  H->Elements[ 0 ];
    }
    /* save the min element */
    MinElement = H->Elements[ 1 ];
    /* take last and reset size */
    LastElement = H->Elements[ H->Size-- ];

    for ( i = 1; i * 2 <= H->Size; i = Child ) {  /* Find smaller child */
         Child = i * 2;
         if (Child != H->Size && H->Elements[Child+1] < H->Elements[Child])
           Child++;
         if ( LastElement > H->Elements[ Child ] )   /* Percolate one level */
           H->Elements[ i ] = H->Elements[ Child ];
         else     break;   /* find the proper position */
    }
    H->Elements[ i ] = LastElement;
    return  MinElement;
}
```

```c
Heap *Build(Heap *H, Size N)
{
    // H[1]~H[N] contains the original sequence
    if(N!=1){
        int i=N/2; // the first subroot that needs adjust
        for( ; i>=1; i--){
            PercolateDown(i);
        }
    }
    return H;
}
```

> faster than to build the heap anew

Union and Find

Equivalence Relation:

- reflexive
- symmetric
- transitive

Partial Order:

- reflexive
- anti-symmetric
- transitive

```
void  SetUnion ( DisjSet S, SetType Rt1, SetType Rt2 )
{    S [ Rt2 ] = Rt1 ;     }
```

```
SetType  Find ( ElementType X, DisjSet S )
{   for ( ; S[X] > 0; X = S[X] )   ;
    return  X ;
}
```

Smart Union

Union by size: `S[root]=-size(initialized as -1)`

> Always change the smaller tree

$$height(T) \le \lfloor log_2 N \rfloor + 1$$

N Union and M Find: $O(N + Mlog_2 N)$

Union by height:

> Always change the shallow tree

Path Compression

```
SetType  Find ( ElementType  X, DisjSet  S )
{
    if ( S[ X ] <= 0 )     return  X;
    else     return  S[ X ] = Find( S[ X ], S );
}
```

```
SetType  Find ( ElementType  X, DisjSet  S )
{   ElementType  root,  trail,  lead;
    for ( root = X; S[ root ] > 0; root = S[ root ] )
        ;  /* find the root */
    for ( trail = X; trail != root; trail = lead ) {
       lead = S[ trail ] ;
       S[ trail ] = root ;
    }  /* collapsing */
    return  root ;
}
```

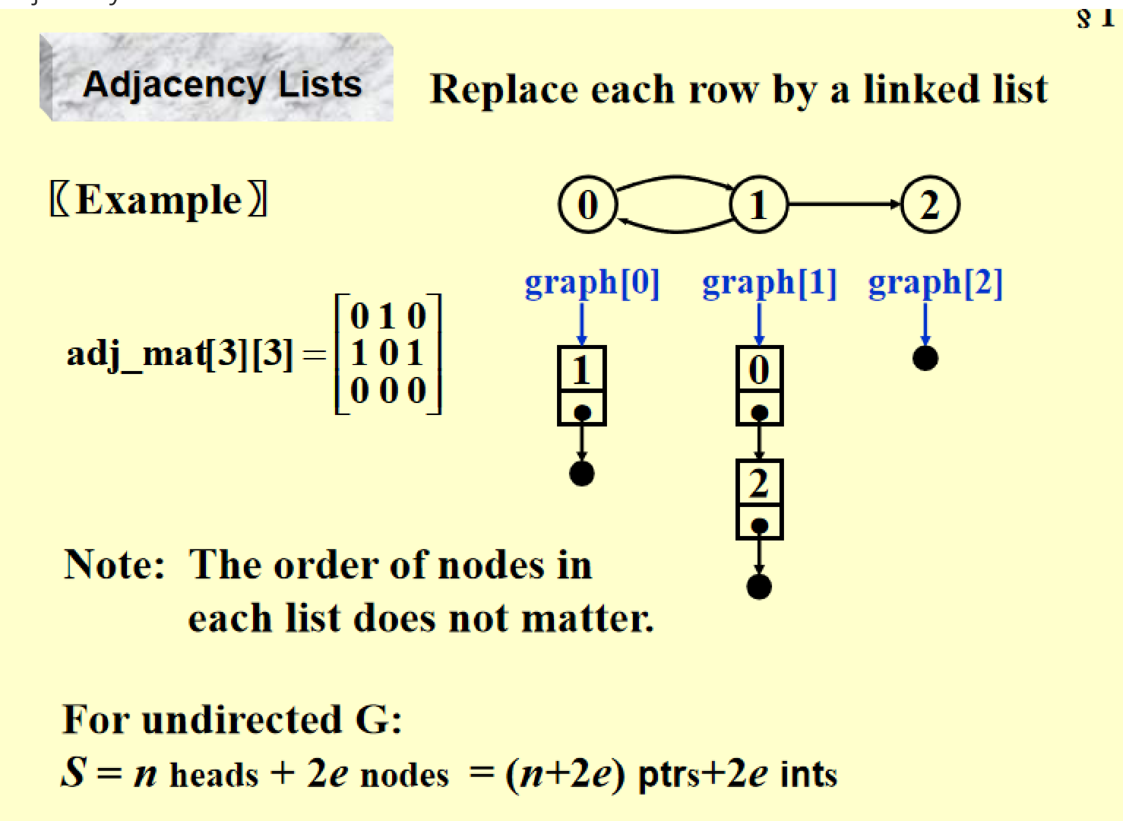
Graph

Simple path ::= $v_{i1}, v_{i2}, .., v_{in}$ are distinct

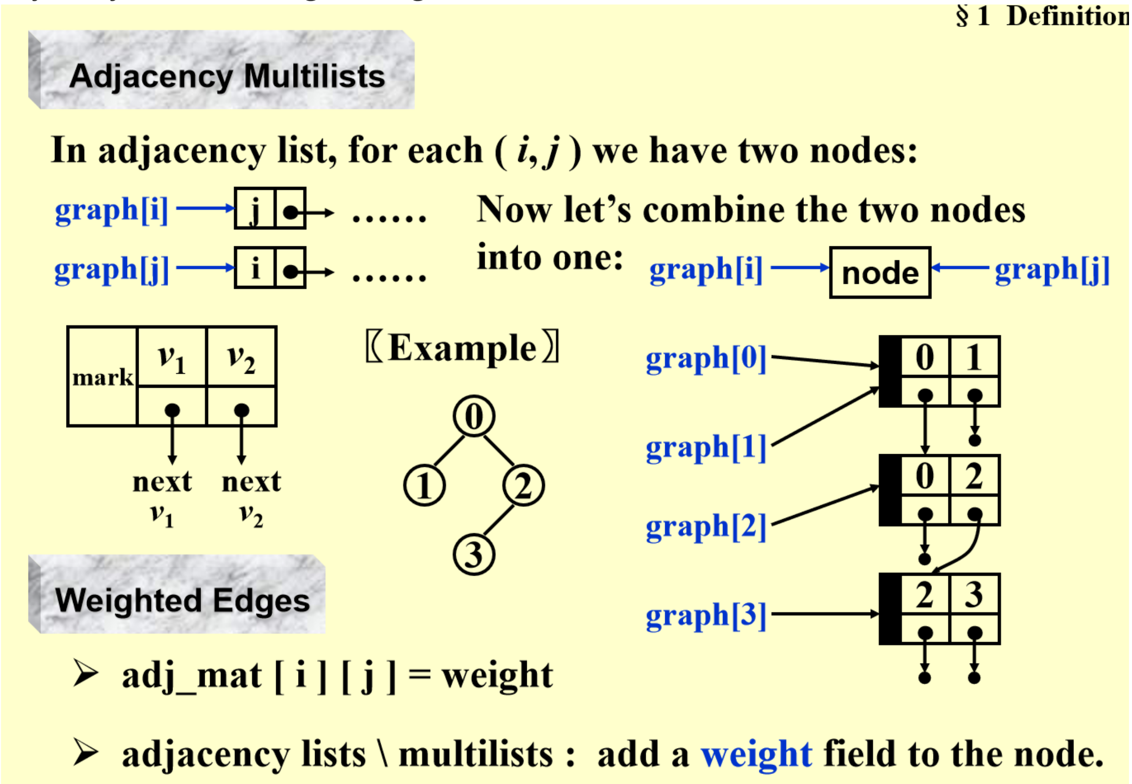
Cycle ::= simple path with $v_p = v_q$

Representation

- Adjacency Matrix
- Adjacency List

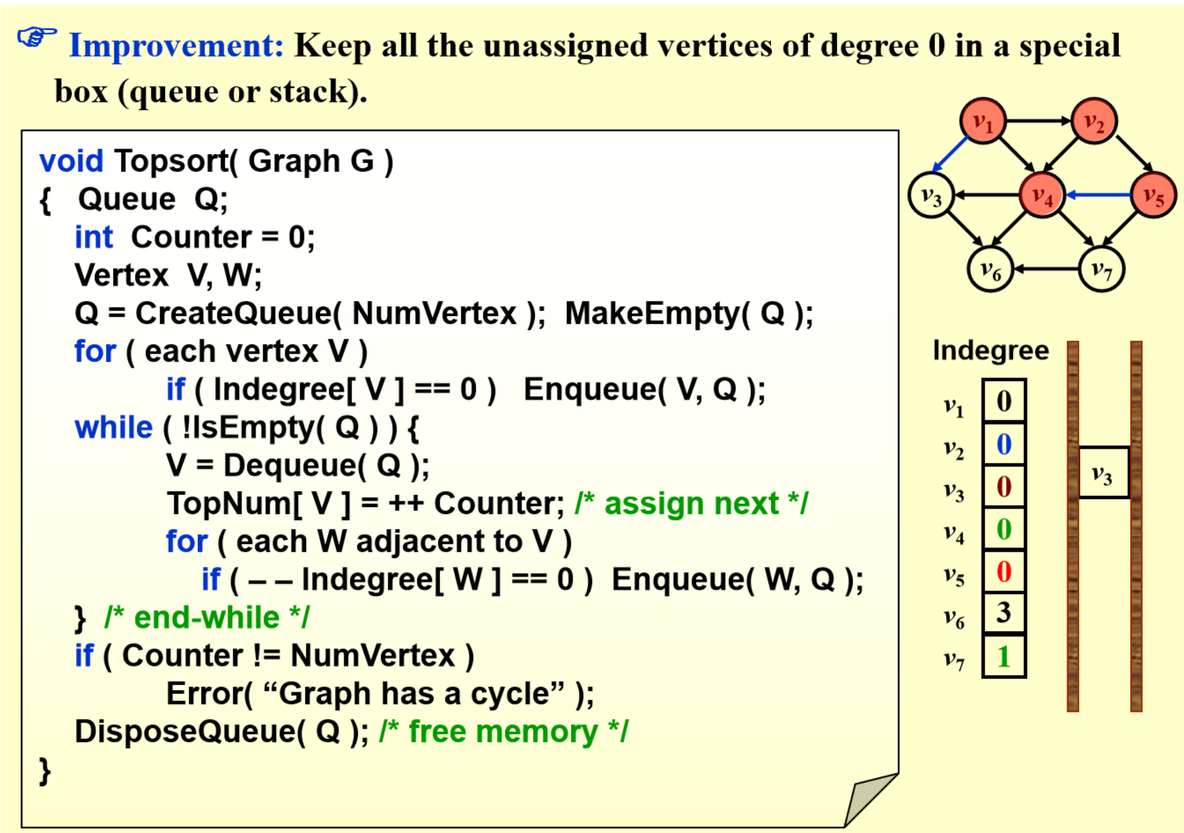


- Adjacency Multilist & Weighted Edge



> When the program needs to mark the edge

Topological Sort



Shortest Path Problem

Unweighted Shortest Path

Breadth-first search

❖ Implementation

Table[i].Dist ::= distance from s to v_i /* initialized to be ∞ except for s */

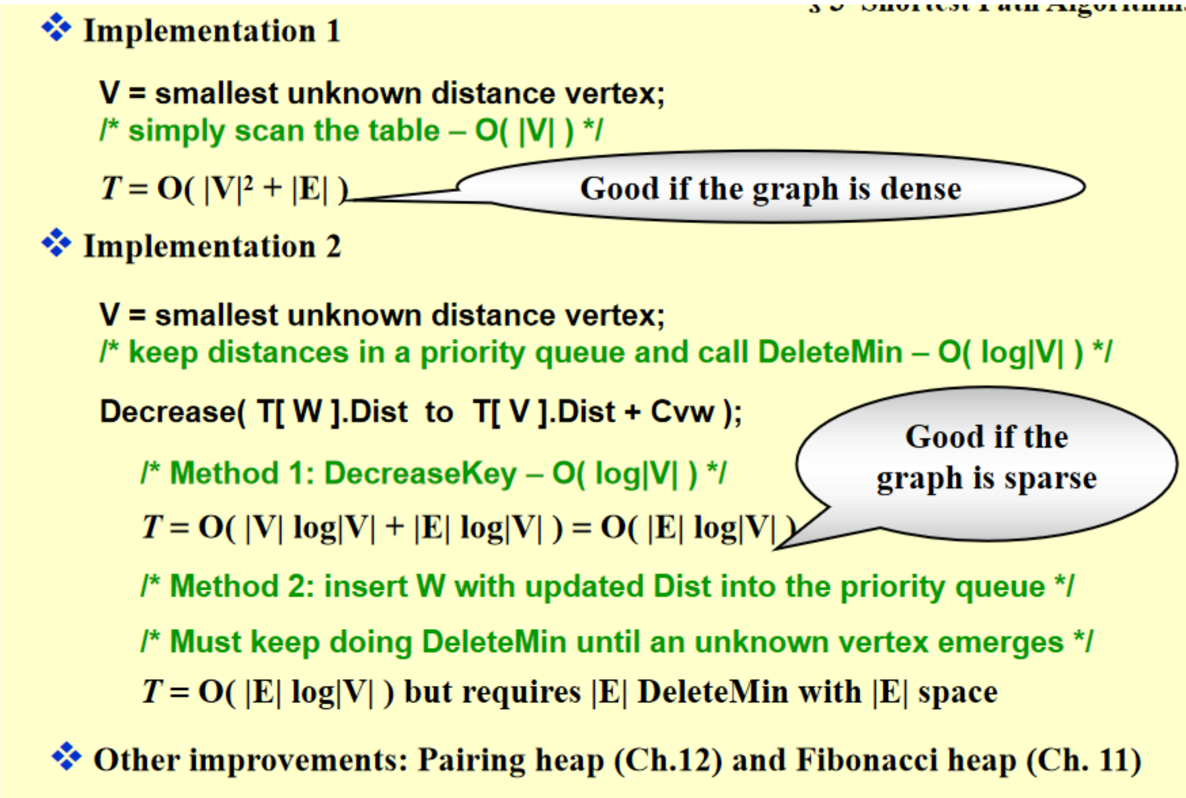
Table[i].Known ::= 1 if v_i is checked; or 0 if not

Table[i].Path ::= for tracking the path /* initialized to be 0 */

```
void Unweighted( Table T )
{   /* T is initialized with the source vertex S given */
    Queue  Q;
    Vertex  V, W;
    Q = CreateQueue (NumVertex );  MakeEmpty( Q );
    Enqueue( S, Q ); /* Enqueue the source vertex */
    while ( !IsEmpty( Q ) ) {
        V = Dequeue( Q );
        T[ V ].Known = true; /* not really necessary */
        for ( each W adjacent to V )
    if ( T[ W ].Dist == Infinity ) {
        T[ W ].Dist = T[ V ].Dist + 1;
        T[ W ].Path = V;
        Enqueue( W, Q );
    } /* end-if Dist == Infinity */
    } /* end-while */
    DisposeQueue( Q ); /* free memory */
}
```

Weighted Shortest Path

```
void Dijkstra( Table T )
{   /* T is initialized by Figure 9.30 on p.303 */
    Vertex  V, W;
    for ( ; ; ) {
        V = smallest unknown distance vertex;
        if ( V == NotAVertex )
    break;
        T[ V ].Known = true;
        for ( each W adjacent to V )
    if ( !T[ W ].Known )
        if ( T[ V ].Dist + Cvw < T[ W ].Dist ) {
            Decrease( T[ W ].Dist to T[ V ].Dist + Cvw );
            T[ W ].Path = V;
        } /* end-if update W */
    } /* end-for( ; ; ) */
}
/* not work for edge with negative cost */
```

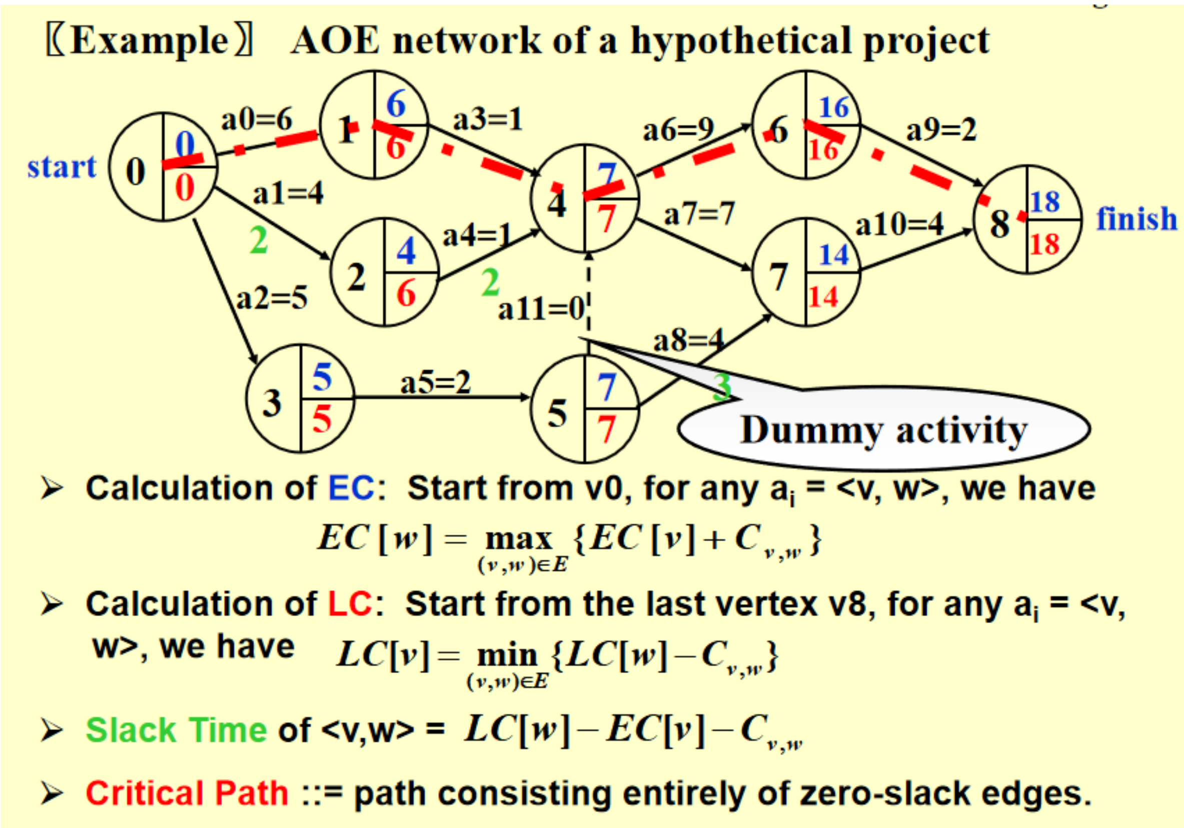


Negative Cost

```
void  WeightedNegative( Table T )
{   /* T is initialized by Figure 9.30 on p.303 */
    Queue  Q;
    Vertex  V, W;
    Q = CreateQueue (NumVertex );  MakeEmpty( Q );
    Enqueue( S, Q ); /* Enqueue the source vertex */
    while ( !IsEmpty( Q ) ) {
        V = Dequeue( Q );
        for ( each W adjacent to V )
    if ( T[ V ].Dist + Cvw < T[ W ].Dist ) {
        T[ W ].Dist = T[ V ].Dist + Cvw;
        T[ W ].Path = V;
        if ( W is not already in Q )
            Enqueue( W, Q );
    } /* end-if update */
    } /* end-while */
    DisposeQueue( Q ); /* free memory */
}
/* negative-cost cycle will cause indefinite loop */
```

> each vertex is visited more than once

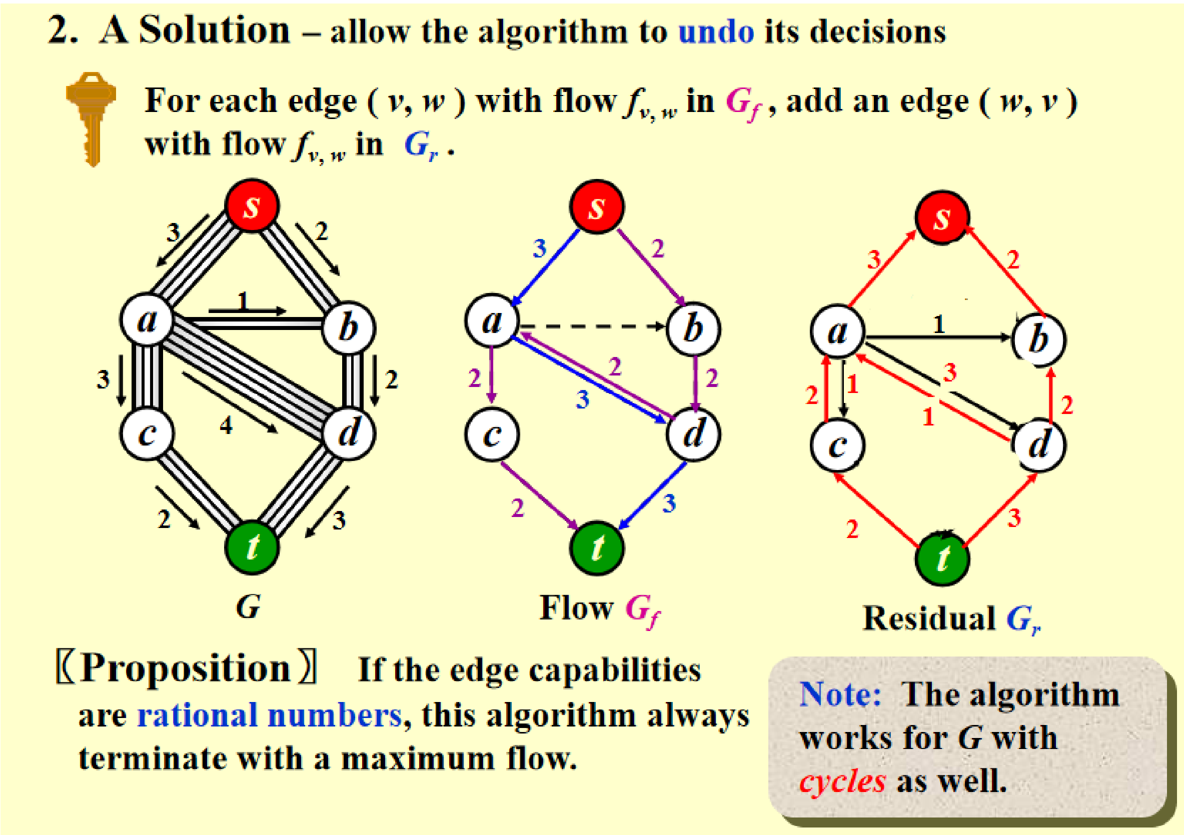
Acyclic Graph



> Those nodes with zero slack are critical, and called critical path. If their values change, the aggregate time changes.

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

Network Flow



> **最大流最小割（Max-flow min-cut）定理**: 在任何网络中，最大流$flow$的值=最小割的$capacity$

https://www.cnblogs.com/ecjtuacm-873284962/p/7113635.html

$$T = O(f \cdot |E|)$$

Minimum Spanning Tree

***Prim*'s Algorithm**:

- Start from a vertex, $S = \{s_0\}$
- for $s_i \in S$, find w such that $min < s_i, w >$, and add w to S

***Kruskal*'s Algorithm**:

```
void Kruskal ( Graph G )
{   T = { } ;
    while  ( T contains less than |V| -1 edges && E is not empty ) {
        choose a least cost edge (v, w) from E ;
        delete (v, w) from E ;
        if  ( (v, w) does not create a cycle in T )
            add (v, w) to T ;
        else
            discard (v, w) ;
    }
    if  ( T contains fewer than |V|-1 edges )
        Error ( “No spanning tree” ) ;
}
```

- Start from an edge, $E = \{e_0\}$
- find $e \notin E$ such that $min < e > \&not\ cycle$, and add e to E

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

DFS

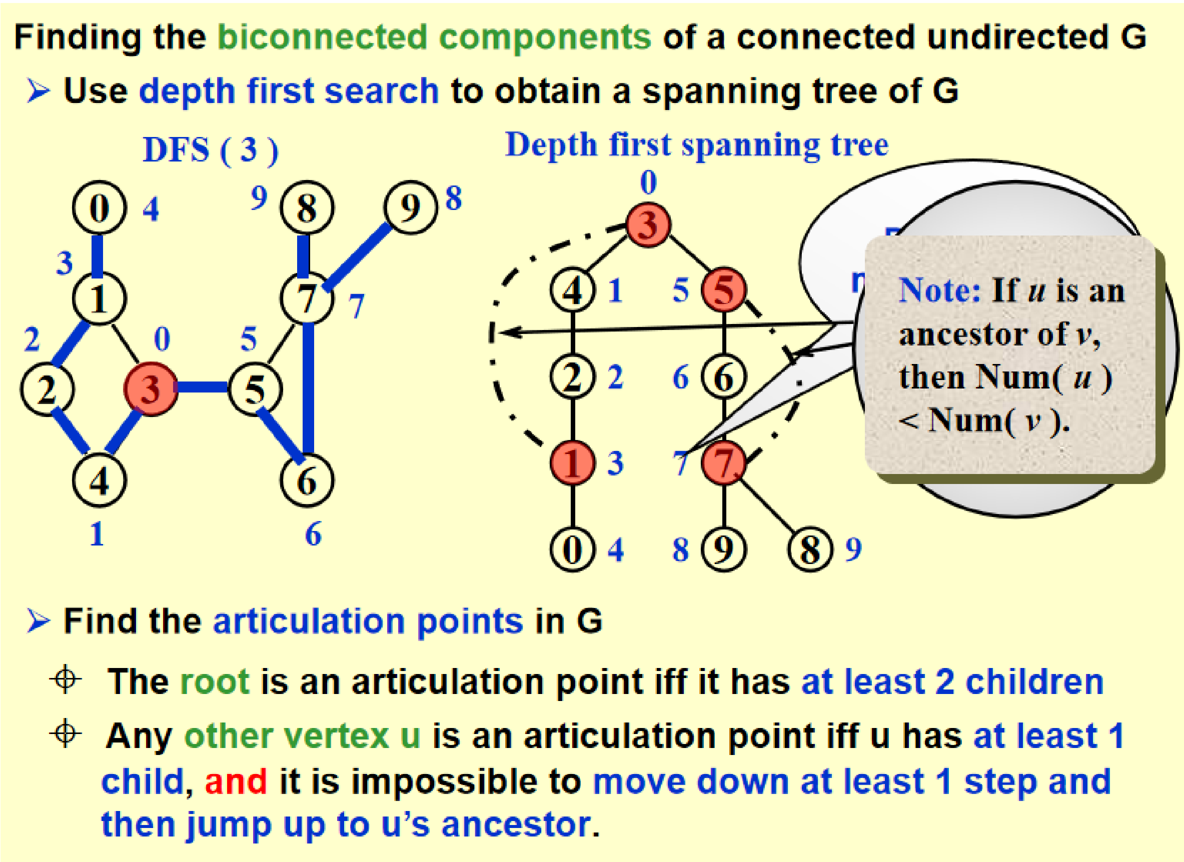
```
void DFS ( Vertex V )  /* this is only a template */
{   visited[ V ] = true;  /* mark this vertex to avoid cycles */
    for ( each W adjacent to V )
        if ( !visited[ W ] )
            DFS( W );
} /* T = O( |E| + |V| ) as long as adjacency lists are used */
```

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

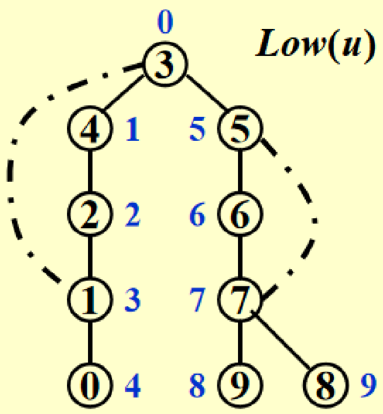
```
void ListComponents ( Graph G )
{   for ( each V in G )
        if ( !visited[ V ] ) {
    DFS( V );
             printf(“\n“);
        }
}
```

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.
- $E(G)$ is partitioned by the biconnected components of G.



> Num: the order produced when a spanning tree is generated. The smaller, the more superior
>
> If u is missed, the superior and the subordinate can not get in touch.



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

vertex	0	1	2	3	4	5	6	7	8	9
Num	4	3	2	0	1	5	6	7	9	8
Low	4	0	0	0	0	5	5	5	9	8

Therefore, u is an articulation point iff

(1) u is the root and has at least 2 children; or

(2) u is not the root, and has at least 1 child such that $Low(\ child\ ) \geq Num(\ u\ )$.

Please read the pseudocodes on p.327 and p.329 for more details.

> Low represents the highest level that u can get in touch, with an exception(back edge)
>
> If $Low(child) \geq Num(u)$, at least one child must get in touch with the superior by u, so u is critical

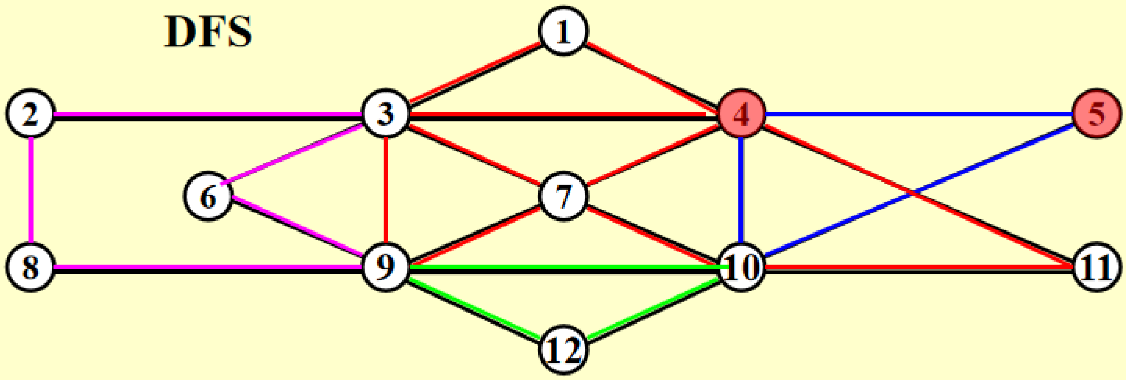
Euler Circuit

Draw each line exactly once without lifting your pen from the paper – *Euler tour*

Draw each line exactly once without lifting your pen from the paper, AND finish at the starting point – *Euler curcuit*

〖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.
- $T = O(|E| + |V|)$

Find a simple cycle in an undirected graph that visits every vertex – *Hamilton cycle*

Sort

Shellsort

§ 4 Shellsort ---- by Donald Shell

〖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* $h_1 < h_2 < \ldots < h_t$ ($h_1 = 1$)

Define an h_k-sort at each phase for $k = t, t - 1, \ldots, 1$

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

> make use of insertion sort

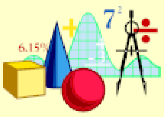
```
void Shellsort( ElementType A[ ], int N )
{
      int  i, j, Increment;
      ElementType  Tmp;
      for ( Increment = N / 2; Increment > 0; Increment /= 2 )
    /*h sequence */
    for ( i = Increment; i < N; i++ ) { /* insertion sort */
          Tmp = A[ i ];
          for ( j = i; j >= Increment; j - = Increment )
        if( Tmp < A[ j - Increment ] )
              A[ j ] = A[ j - Increment ];
        else
              break;
        A[ j ] = Tmp;
    } /* end for-I and for-Increment loops */
}
```

The sequence is critical:

Hibbard's Increment Sequence:

$h_k = 2^k - 1$ ---- consecutive increments have no common factors.

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



Conjectures:

$T_{avg-Hibbard}(N) = O(N^{5/4})$

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).

Sedgewick's best sequence is $\{1, 5, 19, 41, 109, \dots\}$ in which the terms are either of the form $9\times4^i - 9\times2^i + 1$ or $4^i - 3\times2^i + 1$. $T_{avg}(N) = O(N^{7/6})$ and $T_{worst}(N) = O(N^{4/3})$.

Heap Sort

```
void Heapsort( ElementType A[ ], int N )
{  int i;
    for ( i = N / 2; i >= 0; i - - ) /* BuildHeap */
        PercDown( A, i, N );
    for ( i = N - 1; i > 0; i - - ) {
        Swap( &A[ 0 ], &A[ i ] ); /* DeleteMax */
        PercDown( A, 0, i );
}
```

> maintain a max heap
>
> Heapsort data start from position 0.

Mergesort

```
void MSort( ElementType A[ ], ElementType TmpArray[ ],
        int Left, int Right )
{   int  Center;
    if ( Left < Right ) {  /* if there are elements to be sorted */
        Center = ( Left + Right ) / 2;
        MSort( A, TmpArray, Left, Center );     /* T( N / 2 ) */
        MSort( A, TmpArray, Center + 1, Right );    /* T( N / 2 ) */
        Merge( A, TmpArray, Left, Center + 1, Right );  /* O( N ) */
    }
}

void Mergesort( ElementType A[ ], int N )
{   ElementType  *TmpArray;  /* need O(N) extra space */
    TmpArray = malloc( N * sizeof( ElementType ) );
    if ( TmpArray != NULL ) {
        MSort( A, TmpArray, 0, N - 1 );
        free( TmpArray );
    }
    else  FatalError( "No space for tmp array!!!" );
}

void Merge( ElementType A[ ], ElementType TmpArray[ ],
           int Lpos, int Rpos, int RightEnd )
{   int  i, LeftEnd, NumElements, TmpPos;
    LeftEnd = Rpos - 1;
    TmpPos = Lpos;
    NumElements = RightEnd - Lpos + 1;
    while( Lpos <= LeftEnd && Rpos <= RightEnd ) /* main loop */
        if ( A[ Lpos ] <= A[ Rpos ] )
            TmpArray[ TmpPos++ ] = A[ Lpos++ ];
        else
            TmpArray[ TmpPos++ ] = A[ Rpos++ ];
    while( Lpos <= LeftEnd ) /* Copy rest of first half */
        TmpArray[ TmpPos++ ] = A[ Lpos++ ];
    while( Rpos <= RightEnd ) /* Copy rest of second half */
        TmpArray[ TmpPos++ ] = A[ Rpos++ ];
    for( i = 0; i < NumElements; i++, RightEnd - - )
         /* Copy TmpArray back */
        A[ RightEnd ] = TmpArray[ RightEnd ];
}
```

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

$$T = O(NlogN)$$

Quicksort

```
ElementType Median3( ElementType A[ ], int Left, int Right )
{
    int  Center = ( Left + Right ) / 2;
    if ( A[ Left ] > A[ Center ] )
        Swap( &A[ Left ], &A[ Center ] );
    if ( A[ Left ] > A[ Right ] )
        Swap( &A[ Left ], &A[ Right ] );
    if ( A[ Center ] > A[ Right ] )
        Swap( &A[ Center ], &A[ Right ] );
    /* Invariant: A[ Left ] <= A[ Center ] <= A[ Right ] */
    Swap( &A[ Center ], &A[ Right - 1 ] ); /* Hide pivot */
    /* only need to sort A[ Left + 1 ] … A[ Right – 2 ] */
    return  A[ Right - 1 ];  /* Return pivot */
}

void  Qsort( ElementType A[ ], int Left, int Right )
{   int  i,  j;
    ElementType  Pivot;
    if ( Left + Cutoff <= Right ) {  /* if the sequence is not too short */
        Pivot = Median3( A, Left, Right );  /* select pivot */
        i = Left;     j = Right – 1;  /* why not set Left+1 and Right-2? */
        for( ; ; ) {
     while ( A[ + +i ] < Pivot ) { }  /* scan from left */
     while ( A[ – –j ] > Pivot ) { }  /* scan from right */
     if ( i < j )
        Swap( &A[ i ], &A[ j ] );  /* adjust partition */
     else     break;  /* partition done */
        }
        Swap( &A[ i ], &A[ Right - 1 ] ); /* restore pivot */
        Qsort( A, Left, i - 1 );      /* recursively sort left part */
        Qsort( A, i + 1, Right );   /* recursively sort right part */
    }  /* end if - the sequence is long */
    else /* do an insertion sort on the short subarray */
        InsertionSort( A + Left, Right - Left + 1 );
}
```

> `i=Left;j=Right-1;` because of the loop uses `++i` and `--j`. And that is because the sequent swap and sequent recursion.
>
> If equal, make the swap happen to make two balanced arrays.

$$T_{AVG} = O(NlogN)$$
$$T_{WORST} = O(N^2)$$

Sorting Large Structures

§ 8 Sorting Large Structures

Problem: Swapping large structures can be very much expensive.

Solution: Add a pointer field to the structure and swap pointers instead – indirect sorting. Physically rearrange the structures at last if it is really necessary.

〖Example〗 Table Sort

list	[0]	[1]	[2]	[3]	[4]	[5]
key	d	b	f	c	a	e
table	4	1	3	0	5	2

The sorted list is

list [table[0]], list [table[1]], ..., list [table[n–1]]

Note: Every permutation is made up of disjoint cycles.

list	[0]	[1]	[2]	[3]	[4]	[5]
key	a	b	c	d	e	f
table	0	1	2	3	4	5

temp = d
current = 3
next = 3

In the worst case there are $\lfloor N/2 \rfloor$ cycles and requires $\lfloor 3N/2 \rfloor$ record moves.

$T = O(m\ N)$ where m is the size of a structure.

> $table[i]$ records the current position of the element that is supposed to be here
>
> like, $a, b, c, \ldots$ is mapped to $1, 2, 3, \ldots$. e is supposed to mapped to 4 and its **current position** is 5, so $table[4] = 5$

A General Lower Bound for Sorting

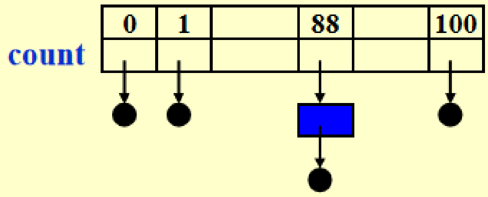
Any algorithm that sorts by comparisons only must have a worst case computing time of $\Omega(NlogN)$.

Bucket Sort and Radix Sort

§ 10 Bucket Sort and Radix Sort

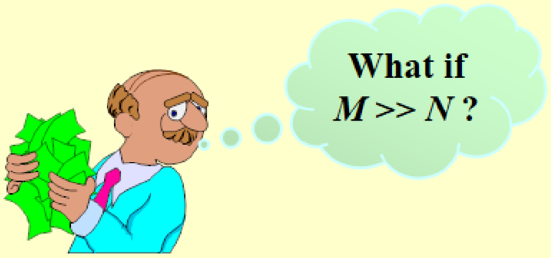
☞ Bucket Sort

〖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?




```
Algorithm
{
   initialize count[ ];
   while (read in a student's record)
      insert to list count[stdnt.grade];
   for (i=0; i<M; i++) {
      if (count[i])
         output list count[i];
   }
}
```

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



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

☞ Radix Sort

What if we sort according to the Most Significant Digit first?

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

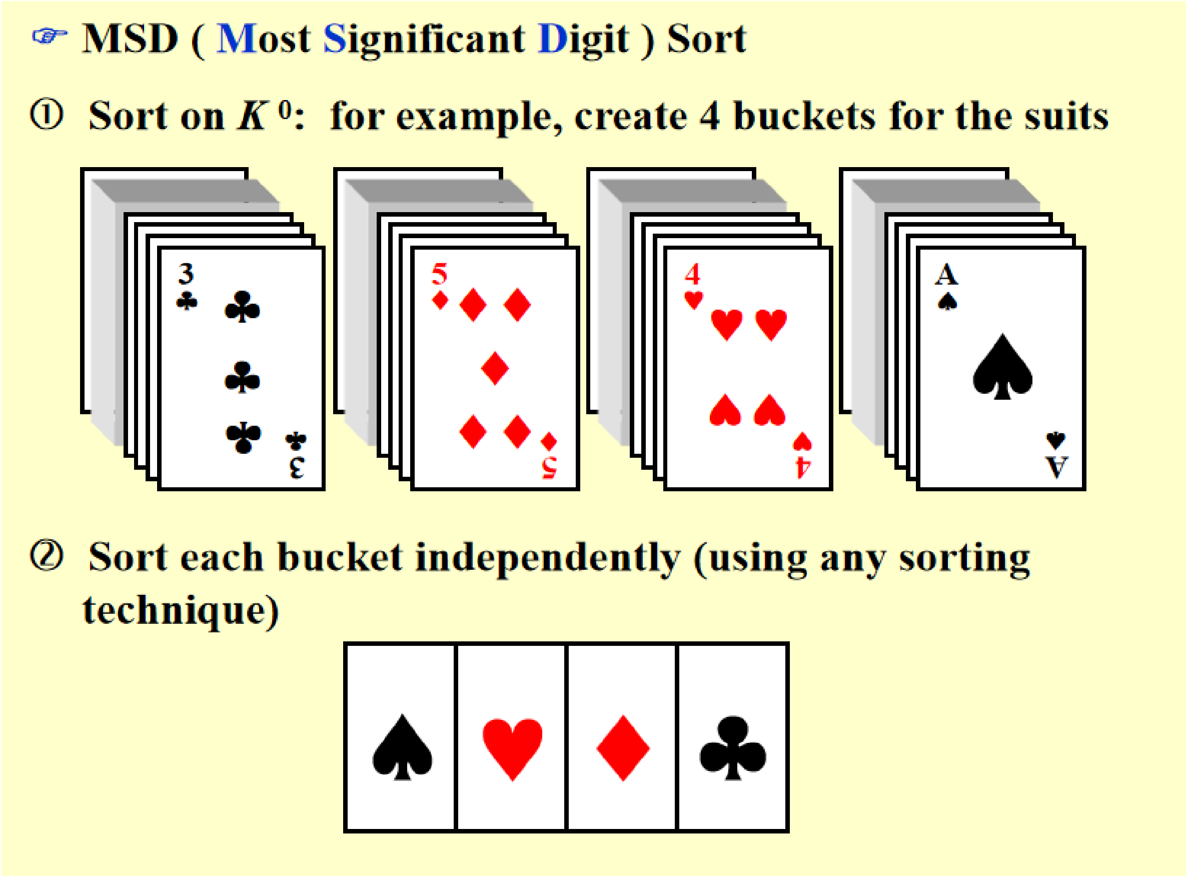
Sort according to the Least Significant Digit first.

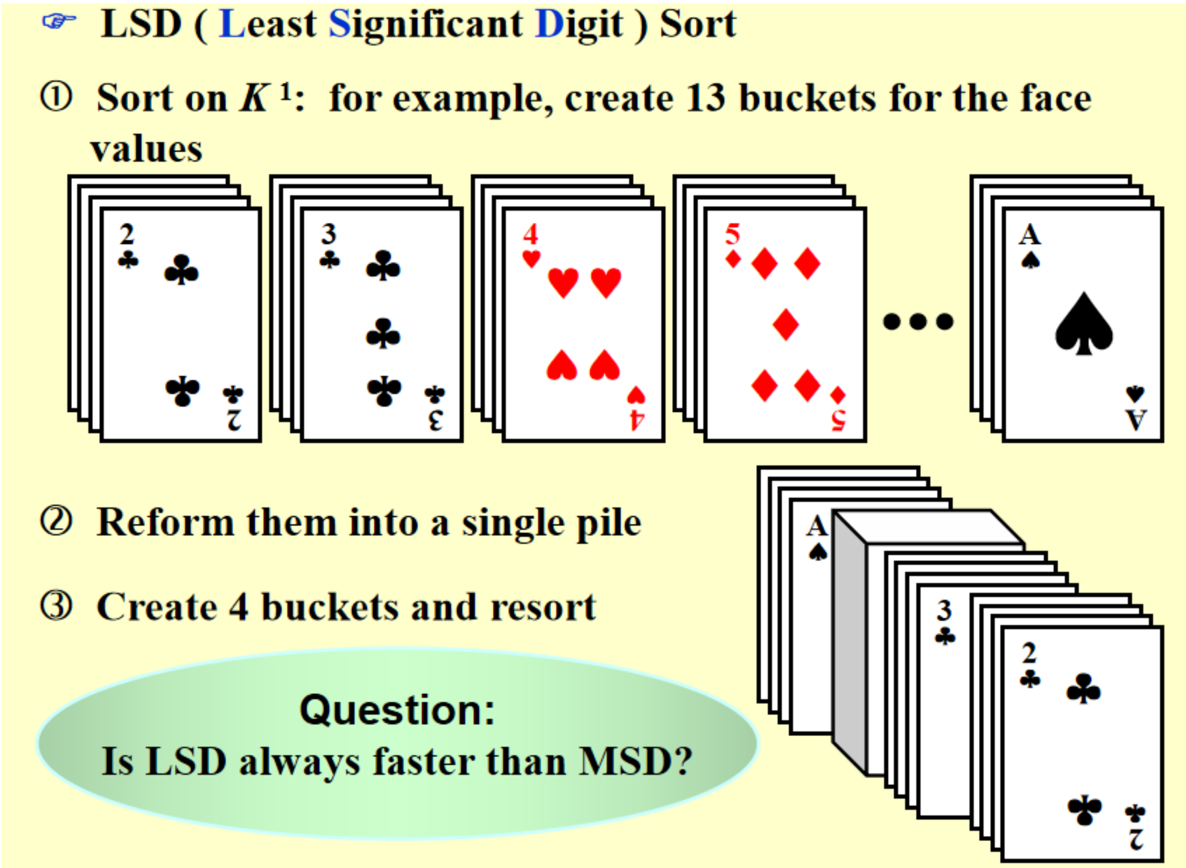
Bucket	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729
Pass 2	0, 1, 8	512, 216	125, 27, 729		343		64			
Pass 3	0, 1, 8, 27, 64	125	216	343		512		729		

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

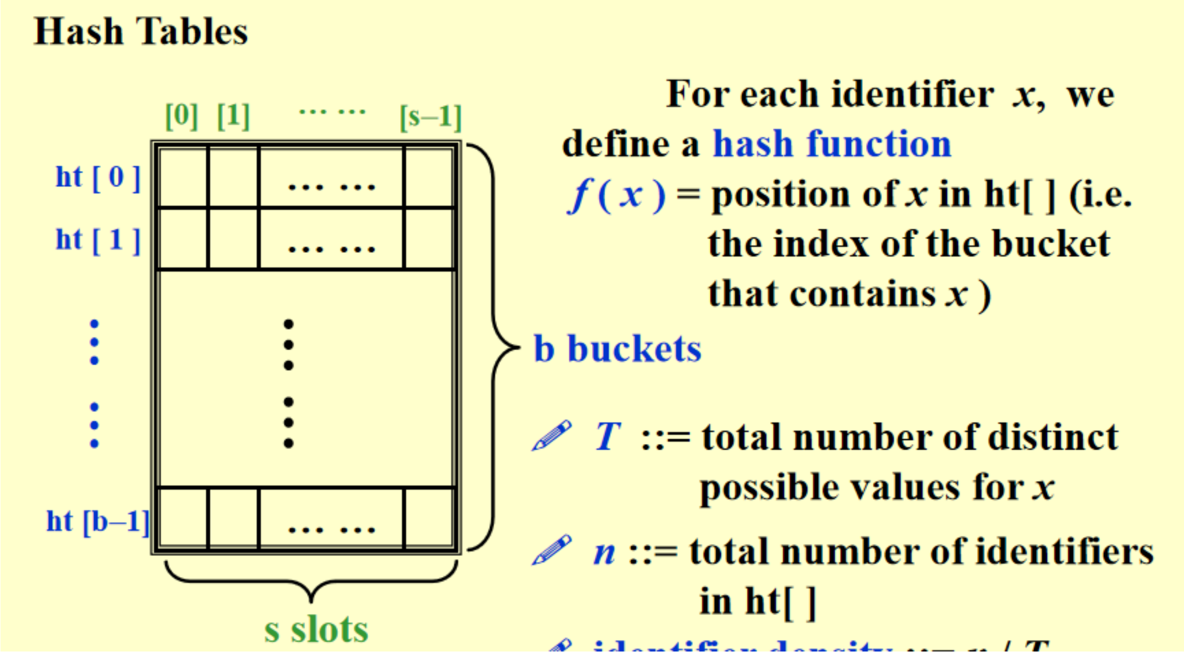
Output: 0, 1, 8, 27, 64, 125, 216, 343, 512, 729

Sample:


MSD (Most Significant Digit) Sort
① Sort on K^0: for example, create 4 buckets for the suits
3
5
4
A
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
3
4
5
A
② Reform them into a single pile
③ Create 4 buckets and resort
Question:
Is LSD always faster than MSD?

Hashing



- identifier density ::= n/T
- loading density λ::= $n/(sb)$

$Tablesize$ **should be a prime number!**

Without overflow:

$$T_{search}=T_{insert}=T_{delete}=O(1)$$

a number

$$f(x)=x\%Tablesize$$

a string

$f(x) = (\sum x[i]) \ \% \ TableSize$; /* if x is a string */

〖Example〗 *TableSize* = 10,007 and string length of $x \le 8$.

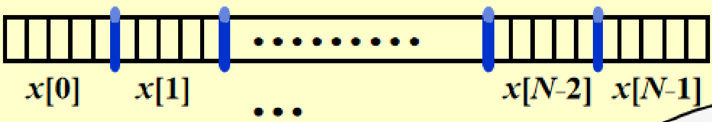
If $x[i] \in [0, 127]$, then $f(x) \in [0, 1016]$ ☹

$f(x) = (x[0] + x[1]*27 + x[2]*27^2) \ \% \ TableSize$;

Total number of combinations = $26^3 = 17{,}576$

☹ Actual number of combinations < 3000

$f(x) = (\sum x[N - i - 1] * 32^i) \ \% \ TableSize$;




```
Index Hash3( const char *x, int TableSize )
{
        unsigned  int  HashVal = 0;
/* 1*/  while( *x != '\0' )
/* 2*/      HashVal = ( HashVal << 5 ) + *x++;
/* 3*/  return HashVal % TableSize;
}
```

Can you make it faster?

Carefully select some characters from x.

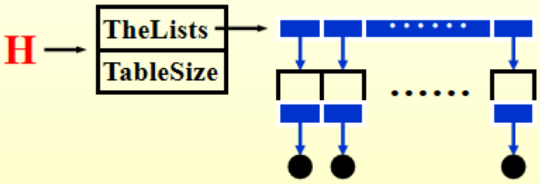
☹ If x is too long (e.g. street address), the early characters will be left-shifted out of place.

For collision

Separate Chaining

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

```c
HashTable  InitializeTable( int TableSize )
{   HashTable  H;
    int  i;
    if ( TableSize < MinTableSize ) {
            Error( "Table size too small" );  return NULL;
    }
    H = malloc( sizeof( struct HashTbl ) );  /* Allocate table */
    if ( H == NULL )    FatalError( "Out of space!!!" );
    H->TableSize = NextPrime( TableSize );  /* Better be prime */
    H->TheLists = malloc( sizeof( List ) * H->TableSize );  /*Array of lists*/
    if ( H->TheLists == NULL )   FatalError( "Out of space!!!" );
    for( i = 0; i < H->TableSize; i++ ) {   /* Allocate list headers */
            H->TheLists[ i ] = malloc( sizeof( struct ListNode ) ); /* Slow! */
            if ( H->TheLists[ i ] == NULL )  FatalError( "Out of space!!!" );
            else    H->TheLists[ i ]->Next = NULL;
    }
    return  H;
}
```




```c
Position  Find ( ElementType Key, HashTable H )
{
    Position P;
    List L;

    L = H->TheLists[ Hash( Key, H->TableSize ) ];

    P = L->Next;
    while( P != NULL && P->Element != Key )  /* Probably need strcmp */
    P = P->Next;
    return P;
}
```

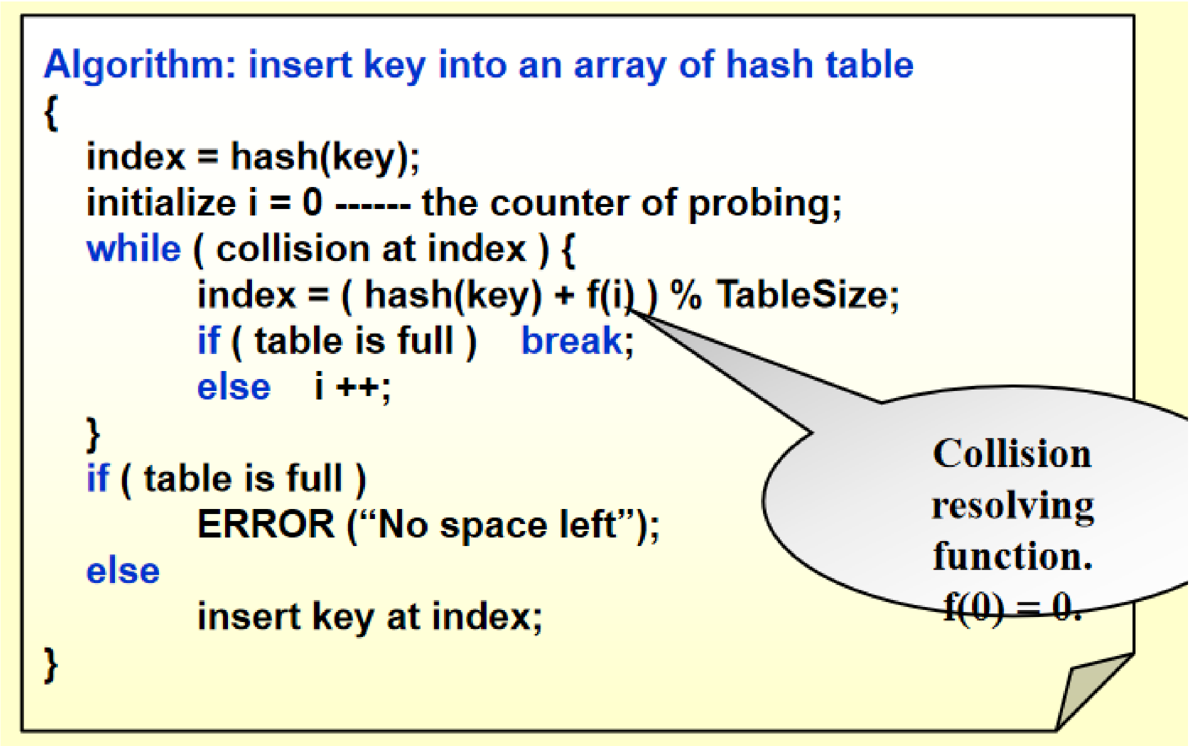
```c
void  Insert ( ElementType Key, HashTable H )
{
    Position   Pos, NewCell;
    List  L;
    Pos = Find( Key, H );
    if ( Pos == NULL ) {   /* Key is not found, then insert */
    NewCell = malloc( sizeof( struct ListNode ) );
    if ( NewCell == NULL )     FatalError( "Out of space!!!" );
    else {
         L = H->TheLists[ Hash( Key, H->TableSize ) ];
         NewCell->Next = L->Next;
         NewCell->Element = Key; /* Probably need strcpy! */
         L->Next = NewCell;
    }
    }
```

```
}
```

> Insertion happens right after the hear node.

Open Addressing

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



Linear Probing

$$f(i) = i$$

Cause primary clustering:

any key that hashes into the cluster will add to the cluster after several attempts to resolve the collision.

Quadratic Probing

$$f(i) = i^2$$

> 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.
>
> 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
>
> For each collision, try from $i = 1$

```
Position  Find ( ElementType Key, HashTable H )
{   Position  CurrentPos;
    int  CollisionNum;
    CollisionNum = 0;
    CurrentPos = Hash( Key, H->TableSize );
    while( H->TheCells[ CurrentPos ].Info != Empty &&
    H->TheCells[ CurrentPos ].Element != Key ) {
        CurrentPos += 2 * ++CollisionNum- 1;
        if ( CurrentPos >= H->TableSize )  CurrentPos - = H->TableSize; //faster
than mod
    }
    return CurrentPos;
}
```

```
void  Insert ( ElementType Key, HashTable H )
{
    Position  Pos;
    Pos = Find( Key, H );
    if ( H->TheCells[ Pos ].Info != Legitimate ) { /* OK to insert here */
    H->TheCells[ Pos ].Info = Legitimate;
    H->TheCells[ Pos ].Element = Key; /* Probably need strcpy */
    }
}
```

> 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

$$f(i) = i * hash_2(x)$$

> $hash_2(x) = R-(x\%R)$with R a prime smaller than $TableSize$, will work well.

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?

1. As soon as the table is **half full**
2. When an insertion fails
3. When the table reaches a certain load factor

The new position of an element is decided by the new hash function!

0	1	2	3
null	21 and 9	null	7

after rehash:

0	1	2	3	4	5	6	7
null	9	null	null	null	21	null	7