1. Algorithm analysis 算法分析

φ1. What to analyze

Time & space complexity

1. **O(f(N))** : for any n_0<N, T(N)<c f(N) ;
2. Ω(g(N)) : for any n_0<N, T(N)>=c g(N) ;
3. θ(h(N)) : only when T(N) = O(f(N)) = Ω(g(N)) ;
4. o(p(N)) : T(N) = O(p(N)) && T(N) ≠ θ(f(N)) ;

φ2. Compare

EX: 一系列数组，连续数求和的max值：

- 基本算法 : 双循环 : O(N^2) ;
- Divide & conquer : T(N) = 2 T($\frac{N}{2}$) + c N ;
  - T(N) = 2^k O(1) + c k N =N log N ;
- On-line Algorithm :

  -
    ```
    Temp = Sum = 0;
    Temp += A[j];
    if (Temp > Sum) Sum = Temp;
    else if (Temp < 0) Temp = 0;
    ```

2. Lists Stacks and Queues

φ1. Abstract Data Type

Definition: 抽象数据类型

φ2. List

- Simple array : array[i] = item ;
  - Advantage : Find takes O(1) time ;
  - Disadvantage : max size has to be estimated / 不可增删
- Linked list : 链表
  - 优缺点与array相反

φ3. The stack ADT

栈

1. Definition

- A stack is a Last in First out list (LIFO) ;
- insertion & deletion -> top only ;

2. Operations

- Push
- Top
- Pop

```
struct Stack{
    int Capacity ;
    int TopofStack ;
    ElementType *Stack;
}
```

3. Evaluation

- Infix expression : a+b*c-d/e ;
- Prefix expression : -+a*bc/de ;
- Postfix expression : abc*+de/- ;

Infix 转 Postfix :

读到数据output , 遇到符号存入栈 , 若符号优先级高则Push , 优先级低则Pop栈内 .

4. Function Calls

φ4. The Queue ADT

1. Definition

- A Queue is a First in First out list (FIFO) ;
- insertion 队尾 only ; deletion 队头 only ;

2. Operations

- Enqueue
- Dequeue

3. Circular queue

取余，dequeue头部之后，普通的queue不会填充头部，导致空间冗余

因此enqueue的时候用取余算法，填充的是`(rear ++) % N`的位置

但因为为了区分rear和top，实际存储内容个数必须为N-1；

3. Tree

φ1. Preliminaries (预备)

- Definition : A tree is a collection of nodes
- **degree (Node)** : Number of subtrees of the node ;
- siblings : children of the same parent ;
- path for n_1 to n_k : a unique sequence of nodes ;
- depth of n_i : root 到 n_i 的 length(path) ;
- height of n_i : n_i 到 leaf 的 length(path) ;
- length () : edges(边数) for n_i to n_j ;

φ2. Binary tree

1. Definition

each node has at most 2 children ;

2. Tree Traversals (遍历) [1]

Preorder

```
void preorder (tree_ptr T){
    if (T){
        visit(T);
        for (each child C of T)  preorder (C) ;
    }
}
```

Postorder

```
void postorder (tree_ptr T){
    if (T){
        for (each child C of T)  postorder (C) ;
        visit(T);
    }
}
```

Level order

```
void levelorder (tree_ptr T){
    enqueue(T);

    while (!T){
        visit(T = dequeue()) ;
        for (each child C of T)  enqueue (C) ;
    }

}
```

Inorder

```
void inorder (tree_ptr T){
    if (T){
        inorder(T->Left);
        visit(T->Element);
        inorder(T->Right);
    }
}
```

Proporeties

- Edges = Nodes -1 ;
- Max level nodes: 2^{n-1} for n>=1;
- Max tree nodes: 2^n-1 for n >=1;
- For any nonempty tree, **$n_0=n_2+1$**, for n_0=leaf nodes and n_2=nodes of degree 2 (2 children);
    - nodes number = $n_0+n_1+n_2 = n_0*0 + n_1*1 + n_2*2 + 1$

φ3. Binary Search Tree

1. Definition

A Binary Search Tree is a binary tree, and if it is nonempty:

1. each node has one key (integer);
2. **left < root < right;**
3. left and right are also binary search tree

2. ADT

- Objects: A finite ordered list with elements
- Operations

1. Make Empty
2. **Find**
3. Insert
4. Delete
5. Retrieve

3. Implements

Find

Recursive:

```
Position Find (ElementType X, SearchTree T){
    if (! T){
        return NULL;
    }
    if (X < T->Element)  return Find (X,T->Left);
    else {
        if (X > T->Element)  return Find(X,T->Right);
        else return T;
    }
}
```

Time Complexity: **T(N)=O(depth)**;

Nonrecursive:

```
Position Find (ElementType X, SearchTree T){
    while (T){
        if (X == T->Element)  return T;
        if (X < T->Element)  T=T->Left;
        else  T=T->Right;
    }
    return NULL;
}
```

Find Min

```
Position FindMin(SearchTree T){
    if (!N) : return NULL;
    else{
        if (T->Left == NULL)  return T;
        else return FindMin(T->Left);
    }
}
```

Time Complexity: **T(N)=O(depth)**;

Find Max

类似Find Min ;

Insert

插入

```
SearchTree Insert( ElementType X, SearchuTree T){
    /* "if (!T)" is necessary because we insert while !T */
    if (!T){
        T=malloc(sizeof(struct TreeNode));
        if (!T) : FatalError("Out of Space!");
        else{
            T->Element = 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);
        /* If X is in the Tree then we do nothing */
    }
    return T;
}
```

Deletion

删除：情况分类

- Leaf Nodes:
    - Reset its Parent to NULL
- 1 degree Nodes:

    - Replace it by its single child
- 2 degree Nodes:
    - Replace it by the Largest one in its Left subtree or the Smallest one in its Right subtree;
    - Delete the replacing node from the subtree —> Recurtion ?(应该不用，因为这两个用来replace的节点都是leaf)
- Solution for 2 degree Nodes:
    1. Reset left subtree;
    2. Reset right subtree;
- 代码:

```c
SearchTree Delete(ElementType X, SearchTree T){
    Position Temp;
    if (!T) Error("Not Found");
    if (X < T->Element)  T->Left = Delete(X,T->Left);
    else if (X > T->Element){
        T->Right =Delete(X,T->Right);
    }else{
        /* Find the Element */
        if (T->Left && T->Right){
            /* 2 degree */
            Temp=FindMin(T->Right);
            T->Element = Temp->Element; /* replace */
            T->Right = Delete(T->Element, T->Right);
        }else {
            /* 1 or 0 child */
            Temp = T;
            if (!T->Left){
                T=T->Right;
            }else if (!T->Right){
                T=T->Left;
            }
            free(Temp);
        }
    }
    return T;
}
```

Lazy deletion

- Add a flag field to each node, to mark if it’s active or deleted.
- Therefore we delete a node without freeing the space of it.
- If a deleted key is reinserted again, we won’t have to call malloc again.

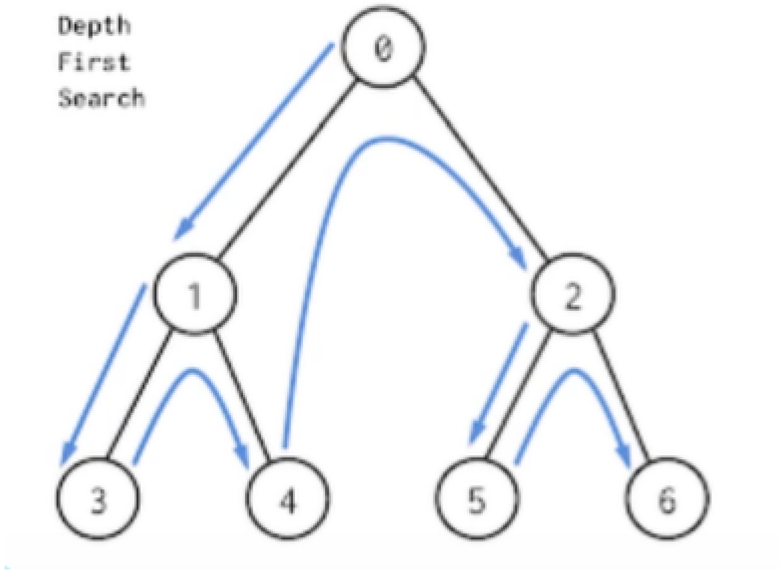
4. Average Case Analysis

5. Decision tree

- 决策树：任意一个非叶子结点的子节点中必至少有一个叶子结点。

𝜑4. DFS & BFS

1. Depth First Search 深度优先算法



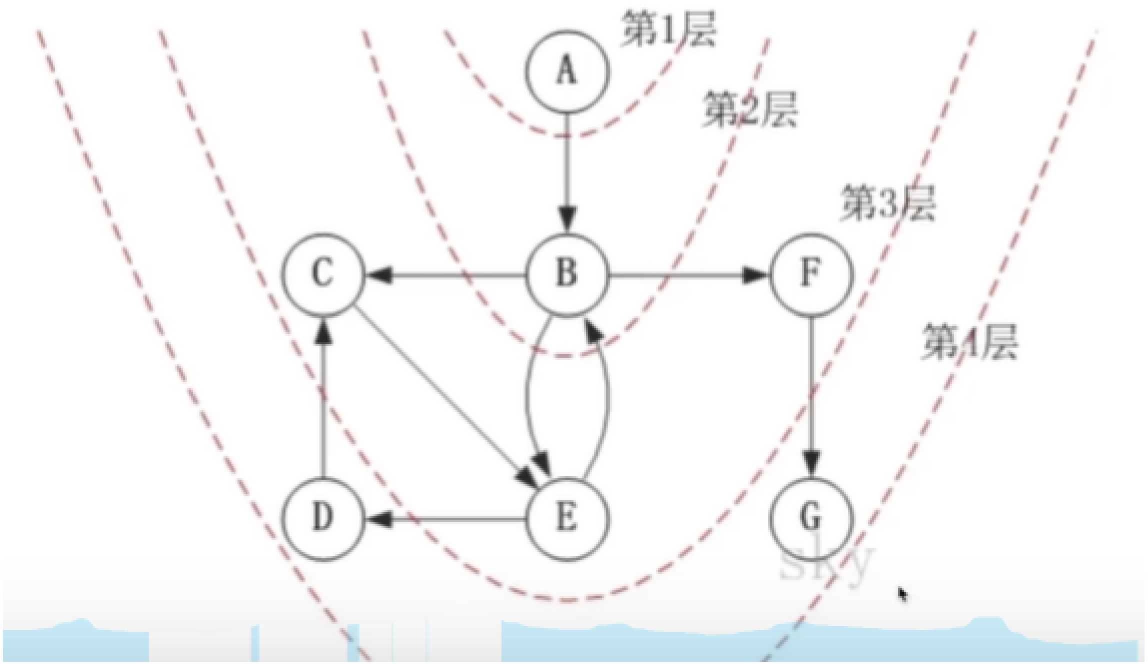
- 假设初始状态所有顶点都没有被访问，做一个初始节点v
- 依次从它各个未被访问的节点出发，遍历，直到图中所有和v相通的节点全部被访问
- 如果还有没被访问的节点，随便设其中一个为v，重复
- T(N)=O(N);

伪代码

```
void DFS_Traversal(int v){
    if (visited[v]) return;  /* 如果访问过了那就不用操作了 */
    visited[v] = true;  /* 标记为访问过了 */
    system.out.print(v+"->");  /* 一个输出 */
    for (each neighbors of Node(v)){  /* 对于图来讲是neighbors，对于二叉树就是children*/
        int Next = neighbors.next();
        DFS_Traveral(Next);
    }
}

void DFS(){
    for (int i=0;i<size;i++){
        if (!visited[i]) DFS_Traveral(i); /* 访问他所有的邻节点 */
    }
}
```

2. Breadth First Search 广度优先算法



- 从图中某一点v出发，先访问v，然后依次访问v的各个没有被访问到的邻节点
- 从这些邻节点出发访问。使得先被访问的节点的邻节点要先于该节点的邻节点的邻节点被访问
- 如果还有没被访问的节点，随便设其中一个为v，重复
- T(N)=O(N);

伪代码

```
void BFS_Traversal(int v){
    struct Queue{};  /* 要建一个队列，相当于levelorder，先进先出 */
    visited[v]=true;
    while(queue.size!=0){
        int cur=queue.first;     /* 队列头 */
        system.out.print(cur);  /* 一种输出 */
        while(neighbors has next){   /* 有未被访问的 */
            int next=neighbors.next();
            if (!visited[next]){
                enqueue(next);    /* 添加到队列 */
                visited[next]=true;   /* 标记为访问过 */
            }
        }
    }
}

```

```
void BFS(){
    for (int i=0;i<size;i++){
        if (!visited[i]) BFS_Traversal(i);
    }
}
```

5. Priority Queue (Heaps)

φ1. ADT Modle

- delete the element with highest priority

φ2. Simple Implementations

- Array
  - Insertion : O(1)
  - Deletion : find O(N) + remove O(N)
- Linked list
  - Insertion : O(1)
  - Deletion : find O(N) + remove O(1)
- Ordered Array
  - Insertion : find O(N) + shift O(N)
  - Deletion : Remove the first and add O(1)
- Ordered linked list
  - find O(N) + shift O(1)
- Binary search tree
  - Insertion : O(1)
  - Deletion :

φ3. Binary Heap (满二叉树)

1. structure priority

height = log N

h^2 <= N <= h^{2+1}-1

- Array repersentation:
  - 故意把a[0]空着，因为他是满二叉树，所以我们可以用int index来判断他的父子节点而不用指针，如下：

- index of parent (i) = i/2 (整除) / null if i==1
- index of left child (i) = 2i / none if 2i > n
- index of right child (i) = 2i+1 / none if 2i+1 > n

PriorityQueue Initialize(int Max)

```c
PriorityQueue Initialize(int Max){
    PQ H;//PQ = PriorityQueue
    if (Max<MinPQsize) return Error;
    H=malloc(sizeof(struct HeapStruct));
    if (!H) return Error;
    /* Allocate the array plus one extra for sentinel */
    H->Element =malloc((Max+1)*sizeof(ElementType));
    if (!H->Element) return Error;
    H->Capacity = Max;
    H->size = 0;
    H->Element[0]=MinData;
    return H;
}
```

> 其实如果数据不多的话根本不用这样建，直接填吧

2. Heap Order Priority

每一层父节点小于子节点就行

因此优先级最高的就是root最顶上

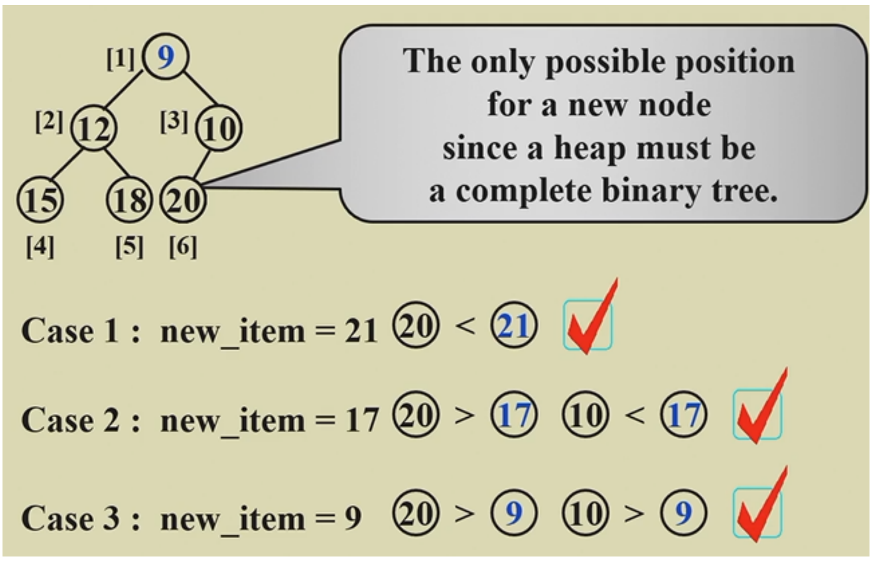
a[0]放一个比所有数都小的

建立Heap的方法不同于插入，他是先把所有数字填进去之后再percolate down

3. Basic Operations

insertion

1. 他必须填进去之后还是一个满二叉树
2. 填进去之后再考虑是否是min heap（**从下向上筛查percolate up**）


[1] 9
[2] 12
[3] 10
15 [4]
18 [5]
20 [6]
The only possible position for a new node since a heap must be a complete binary tree.
Case 1 : new_item = 21 20 < 21
Case 2 : new_item = 17 20 > 17 10 < 17
Case 3 : new_item = 9 20 > 9 10 > 9

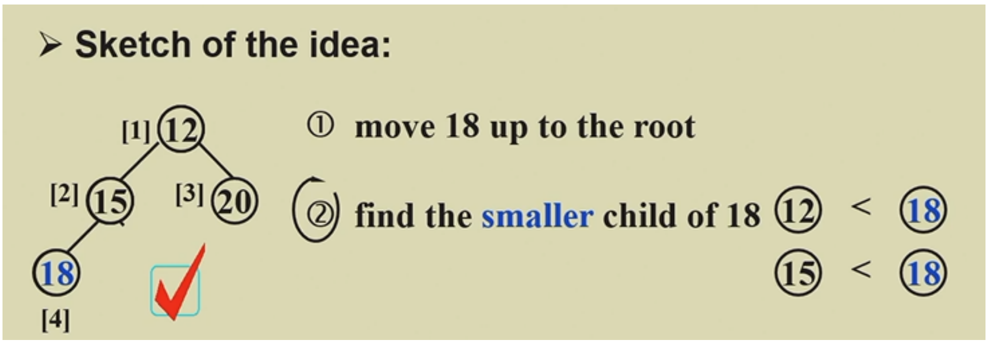
```
void Insert (ElementType X,PQ H){
    int i;
    /* 基础判断 */
    if (isFULL(H)){
        Error;
        return;
    }

    /* percolate up at the bottom */
    for (i=++H->size;H->Element[i/2]>X;i/=2) H->Element[i]=H->Element[i/2];
    H->Element[i]=X;
}
```

T(N)=O(log N)

DeleteMin

1. 删除之后仍然是一个满二叉树并且是min heap
2. 所以真正删除的节点其实是最后一个index的节点
3. 把最后一个数移到root，然后**从上到下percolate down**（替换小的children）



```
ElementType DeleteMin(PQ H){
    /* 感觉也可以写成void，不读，省去Min */
    int i,child;
    /* 检测是否为空，代码略，同上 */
    ElementType Min=H->Element[1],Last=H->Element[H->size--];
    // Last 实际上就是暂时储存最后一个节点的内容，实行一个假的swap，因为
swap要三步用Last暂存就只用一步
    /* percolate down for the root */
    for (i=1;i*2<=H->size;i=child){
        child=2*i;
        /* 找到小的那个child */
        if (child != H->size && H->Element[child]>H->Element[child
+1]) child ++;
        if (Last > H->Element[child]){
            H->Element[i]=H->Element[child];
        }else break;   //find the proper position to store Last
    }
    H->Element[i]=Last;
    return Min;
}
```

T(N)=O(log N)

4. Other Heap Operations

- DecreaseKey(P,Δ,H) 给H中的数P减去Δ
  - percolate up
- Delete
  - 先执行DecreaseKey(P,∞,H)，然后DeleteMin(H)

Biuld Heap(H)

O(N)

- 先建一个满二叉树
- 从倒数第二层开始percolate down

φ4. Applications of PQ

- 排序

找到第k大的数字

当n很大的时候，比如100万里面找前100，那么用heap会很好，因为他不用全排序，只需要O(N)的时间复杂度就可以实现建堆，然后DeleteMin100次

φ5. d-Heaps

all nodes have d children

- DeleteMin take d-1 comparisons to find the smallest child. Hence the time complexity will be O($\log_d$N) ;

- *2 or /2 is merely a bit shift, but *d or /d isn't (so usually we choose d=2^k for integer k)
- when the priority queue is too large to fit entirely in main memory, a d-heap will become interesting
- 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 :
    - $\lfloor (i+d-2)/d \rfloor$, $(i-1)d+2$, and $id+1$

8.Disjoint Set ADT(并查集)

φ1. Equivalence Relations

- Definition : a relation R is defined on a set S if every pair of elements (a,b) , a R b is either true or false .
- Definition : a R b == a ~ b .
- Definition : relation : 对称symmertric , 自反 reflexive, 传递transitive.

φ2. the Dynamic Equivalence Problem

- Example

我们用自然数1-n去定义一堆并查集

```
{
    /* 建并查集 */
    Initialize N joint set;
    while (read a~b){
        if (!(Find(a)==Find(b)))  Union 2 sets;
    }
    /* 新读入一组数a、b，判断是否等价 */
    while (read in a and b){
        if (Find(a)==Find(b)) output true;
        else output false;
    }
}
```

- 与tree最大的不同就是，tree指向children，而并查集指向parent，而parent也是与任何children有同等的关系

φ3. Basic Data Structrue

Union(i,j)

把 j 指向 i 即可，用数组实现

- **S[element] = the element's parent**
- **set S[root] = 0 (or -1) ;**
- S1 Union S2 :
    - S[S2] = S1 ;

Find(i)

从叶指向根

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

SetUnion(Find(i),Find(j)) ;

- Union和Find配套使用

φ4. Smart Union

Union by size

- Always change the smaller tree
- Definition : S[Root] = - size;
    - 一开始都是-1
    - 新加入一个就size--
- Time complexity of N Union and M Find operations is now O(N + Mlog N)

Union by height

- Definition : S[Root] = - height;

φ5. Path Compression 路径压缩

```
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){
    /* 第一遍loop, 找到root */
    for(root=X;S[root]>0;root=S[root]){ }
    /* 第二遍loop，把路径上的全部压缩 */
    for (trail=X;trail!=root;trail=lead){
        lead=S[trail];/* 暂存 */
        S[trail]=root;
    }
    return root;
}
```

> union by
>
> > none
> >
> > size
> >
> > height
> >
> > size with compression
> >
> > rank

φ6. Worst case for

- 结论 : $k_1M\alpha(M,N) <= T(M,N) <= k_2M\alpha(M,N)$
  - Ackermann's Function and α(M,N) ;

$$A(i,j)=\begin{cases}2^j & i=1andj\geq1\\ A(i-2,2) & i\geq2andj=1\\ A(i-1,A(i,j-1)) & i\geq2andj\geq2\end{cases}$$

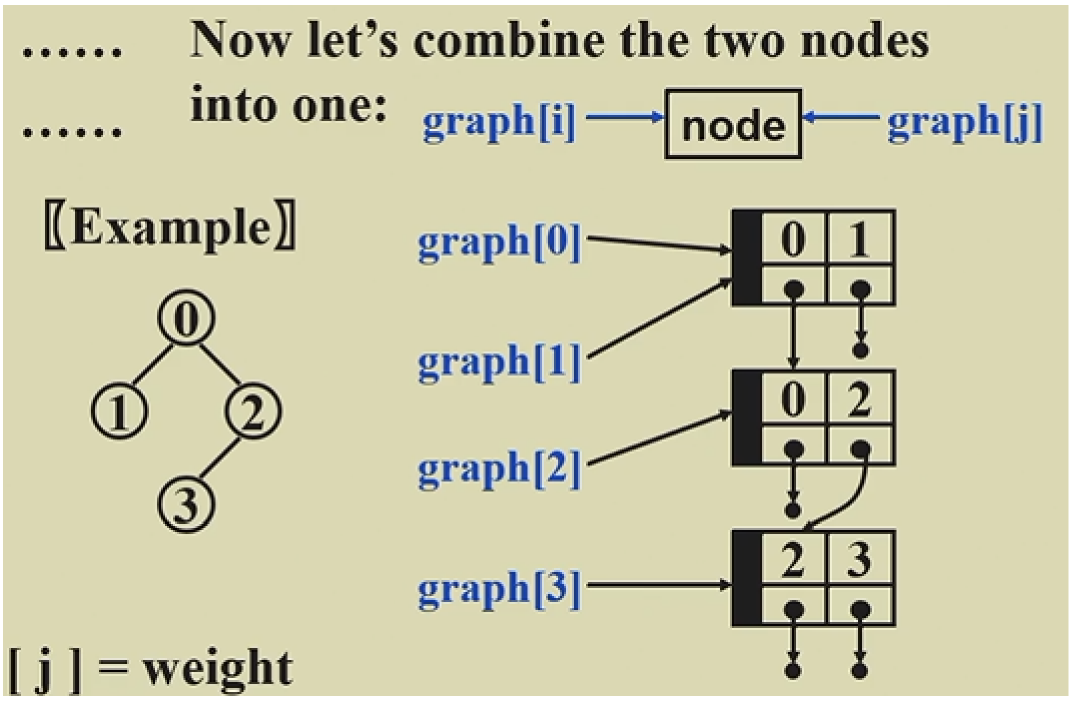
听不懂

9. Graph Algorithm 图论

φ1. Definition

- G(V,E):
  - G := graph
  - V=V(G) := finite nonempty set of vertices(顶点)
  - E=E(G) := finite set of edges
- Undirected graph:
  - (vi,vj)=(vj,vi) is the same edge
- Directed graph:
  - <vi,vj> = vi -> vj = trail -> head
- Restrictions:
  - self loop is illegal (自己指向自己是不被允许的)
  - Multigraph is not considered (多重关系不需要考虑)

- Complete graph:
  - undirected: edges = C_n^2
  - directed: edges = A_n^2
- Path
- Length : edges of the path
- Simple path : distinct path
- Circle : a path where v_p=v_q (起点=终点)
- Connected graph (连通图)
- a DAG = a directed acyclic graph (无环有向图)
- Strongly connected directed graph (component) :
  - for each vi and vj, there are directed paths for both vi to vj and vj to vi
- **Degree**:
  - number of edges incident to vi
  - directed: **in-degree** and out-degree (入度/出度)
  - edges = $\frac{\Sigma\, degree(i)}{2}$
- **Adjacency Matrix:**
  - **adj_mat(i)(j) = 1 if (vi,vj) else = 0**
  - 实现的时候常用一维数组而不是矩阵
  - 有向图满矩阵（$[i][j]$表示i指向j），无向图上三角即可（因此对称）
  - degree(i) = $\Sigma_j\, adj\_mat[i][j]$ (无向) + $\Sigma_j\, adj\_mat[j][i]$ (有向图)
  - in-degree(i) = $\Sigma_j\, adj\_mat[j][i]$（行之和是出度，列之和是入度）
- **Adjacency List:**
  - 用链表存储，若没有出度，则链接到NULL，若有，则链接到下一个节点（包含ElementType 指向元素，Node *Next）
  - For undirected G : S=n heads + 2e nodes.
- **Degree(i):**
  - number of nodes in graph[i] (if undirected) T(E(G))=O(n+e)
  - if directed, we need to find in-degree(v) as well.
- Adjacency Multilists:
  - 把边看成一个节点，一个边被两个节点指，一个边指向两个边节点（含NULL）
- Weighted Edges(权重):
  - **adj_mat(i)(j) = 权重 if (vi,vj) else = 0**



φ2. Topological Sort(拓扑排序)(AOV Network)

eg. 魏学长说的修课问题，需要预修读决定顺序

transitive(传递性) && **irreflexive**(非自反性)

- Definition:
    - a linear order of vertices of a graph for every i and j their order is true
    - 拓扑排序不是真正的排序
    - 结果不唯一（比如有些课哪门先学没关系）

```
void Topsort (Graph G){
    int cnt;
    Vertex V,W;
    for (cnt=0;cnt<NumVertex;cnt++){
        V=FindNewVertexOfDegreeZero();  // 遍历
        /* 把indegree为0的找出来 */
        /* T = number vertex = V（遍历数组一列） */
        if (V == NotAVertex){
            Error("cycle");
            break;
        }
        TopNum[v]=cnt; //输出顺序
        for (each W adjacent to V)  indegree[w]--;
        /* 当indegree为0时，代表所有先决条件已满足，可以输出 */
    }
}
```

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

V:节点　E:边 (边大于节点减一小于节点平方)

- Improvement:

```
void Topsort(Graph G){
    Queue Q;
    int cnt=0;
    Vertex V,W;
    for (each vertex V){
        if (Indegree[V]==0) Enqueue(V,Q);
    }
    while(!IsEmpty(Q)){
        V=Dequeue(Q);
        TopNum[V]=++cnt;    //输出顺序
        for (each W adjacent to V) if(--Indegree[W]=0) 
Enqueue(W,Q);
    }
    if (cnt != NumVertex) Error("cycle");
}
```

有点像中序遍历

T(G)=|V|

φ3. 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 from source to destination is $\Sigma c(e_i)$.

1. Single-Source Shortest-Path Problem

target : to find the shortest path from s to every other vertex in G.

Unweighted Shortest Paths (weight = 1)

- 方法：遍历所有节点，不考虑环
- **BFS**
- Implementation:

```
Table[i].Dist ::= distance from s to vi //initialized as infty
Table[i].Known ::= 1 if vi is checkes else 0
Table[i].Path ::= for tracking the path //initialized as 0
```

- Version 1
    - 完全遍历

```
void Unweighted(Table T){
    int cur;
    Vertex V,W;
    T[S].Known = true; T[S].Dist = 0; //initialize
    for (cur=0;cur<NumVertex;cur++){
        for(each vertex V){
            if (!T[V].Known && T[V].Dist == cur){
                T[V].Known = true;
                for (each W adjacent to V){
                    if (T[W].Dist == infty){
                        T[W].Dist = cur + 1;
                        T[W].Path = V;
                    }
                }
            }
        }
    }
}
```

T = O(|V|^2)

- Version 2

```
void Unweighted(Table T){
    Queue Q;
    Vertex V,W;
    Enqueue(S,Q);
    while( !isEmpty(Q)){
        V = Deque(Q);
        T[V].Known = true; //确认找过
        for (each W adjacent to V){
            if (T[W].Dist == infty){
                T[W].Dist = T[V].Dist + 1;  //V父 W子 路径加1
                T[W].Path = V; //保存来时候的路径
                Enqueue(W,Q);
            }
        }
    }
}
```

T = O(| V |)

Dijkstra’s Algorithm

<u>Let S = {s and vi’s whose shortest paths have been found}</u>

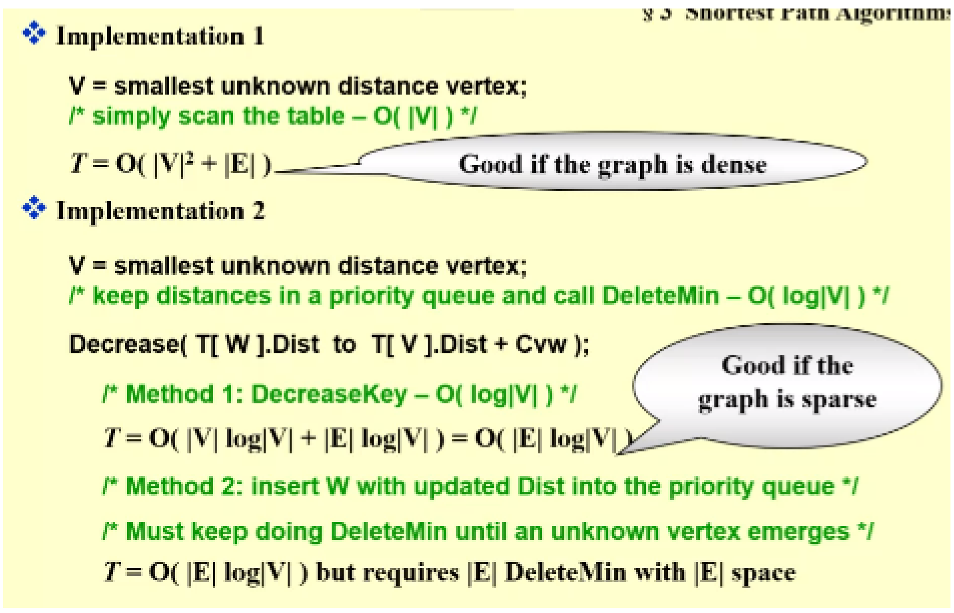
For any u not in S, define [u] = minimal length of path {s -> (vi in S) -> u}.

1. the shortest path must go through ONLY vi in S;
2. u is chosen so that distance[u] = min{w not in S | distance[w]}
3. if distance [u1] < distance[u2] and we add u1 into S ......

Greedy :

```
void Dijk(Table T){
    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;
                }
        }
    }
}
```

T = O(V log V + E log V) = O(E log V);



§ 3 Shortest Path Algorithms

❖ **Implementation 1**

V = smallest unknown distance vertex;
/* simply scan the table – O(|V|) */

$T = O(|V|^2 + |E|)$ Good if the graph is dense

❖ **Implementation 2**

V = smallest unknown distance vertex;
/* keep distances in a priority queue and call DeleteMin – O(log|V|) */

Decrease(T[W].Dist to T[V].Dist + Cvw);

/* Method 1: DecreaseKey – O(log|V|) */

$T = O(|V| \log|V| + |E| \log|V|) = O(|E| \log|V|)$ Good if the graph is sparse

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

/* Must keep doing DeleteMin until an unknown vertex emerges */

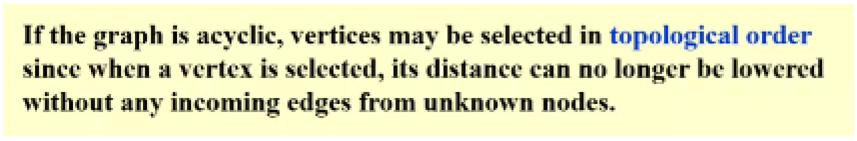
$T = O(|E| \log|V|)$ but requires |E| DeleteMin with |E| space

Graphs with Negative Edge Costs

/* negative-cost cycle will cause indefinite loop */

```
void W(){
    Enqueue(S,Q);
    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 in Q){
                    Enqueue(W,Q);
                }
            }
            }
        }
}
```

Acyclic Graphs(无环)


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.

如果图非周期，顶点按照拓扑顺序选择，顶点的dist就不会改变了（一次过）

T = O(E +V) and no priority queue is needed

EG. AOE network of a hypothetical project

- **Activity on edge**

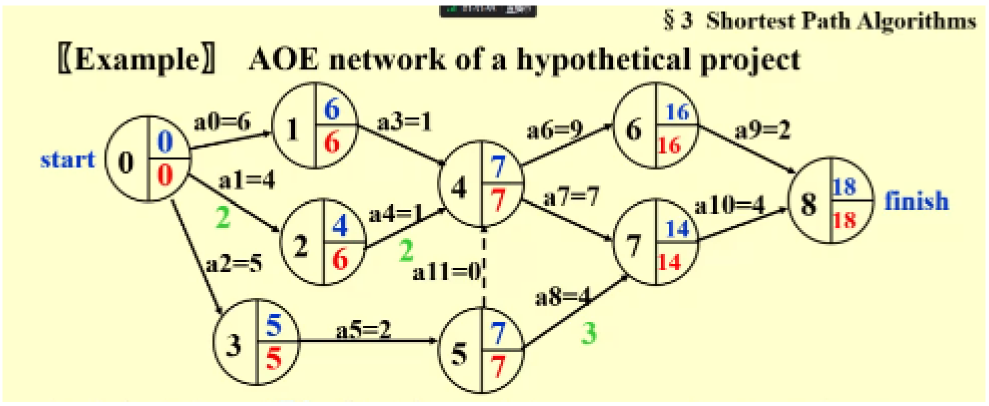
最早完成时间和最迟完成时间

EC[w] = max { EC[v] + Cvw }

LC[v] = min {LC[w] - Cvw }

Slack Time of <v,w> = LC[w] - EC[v] - Cvw

Critical Path = path consisting entirely of 0-slack time edges

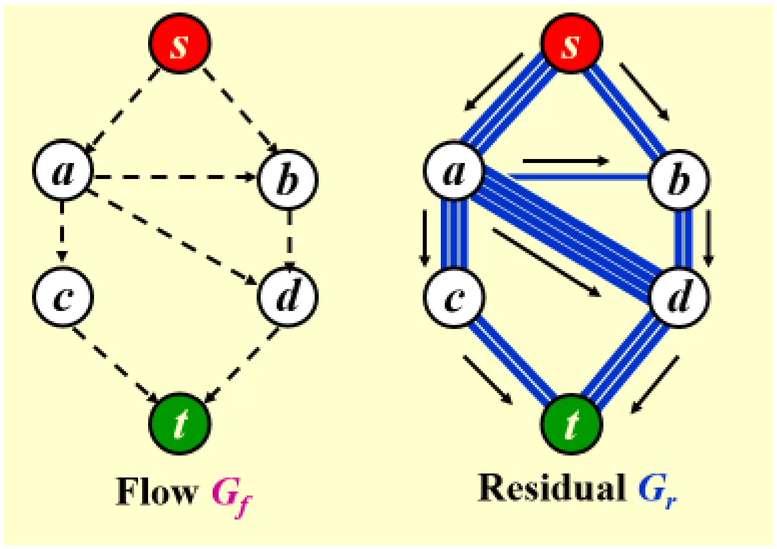


2. All-Pairs Shortest Path Problem

φ4. Network Flow Problems

- 网络流(Source & Sink) : Determine the max amount of flow that can pass from s to t.
- Minimum cut (最小割集)

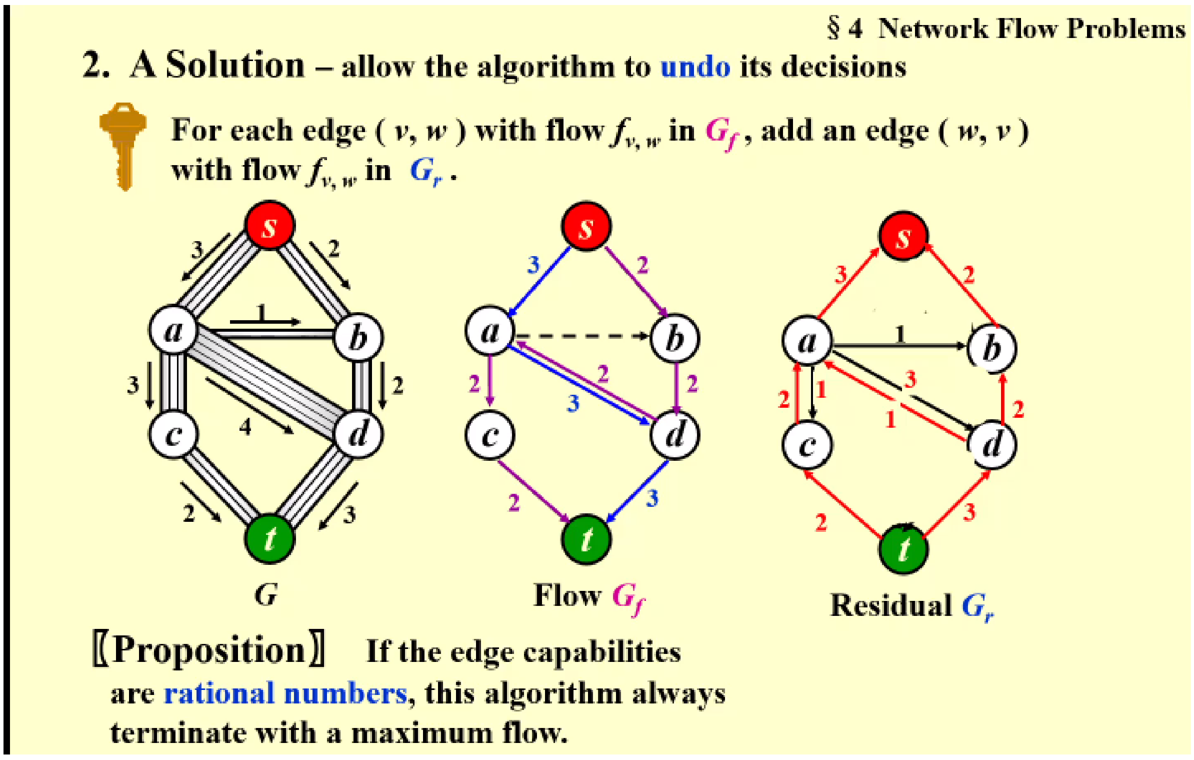
1. A simple Algorithm



1. find any path s->t in G_r (**minimum edge**!!!)
2. take the minimum edge on this path as the amount of flow and add to G_f(给G_f的这条路中加上这条路中最小边权的边的权)
3. Updaate G_r（在G_r图的这条路中减去这个边权） and remove the 0 flow edges.
4. repeat until no path from s to t.

2. A solution - alloew the algorithm to unndo its decisions

给路中所有边减去这条边的权重的同时反向加一条



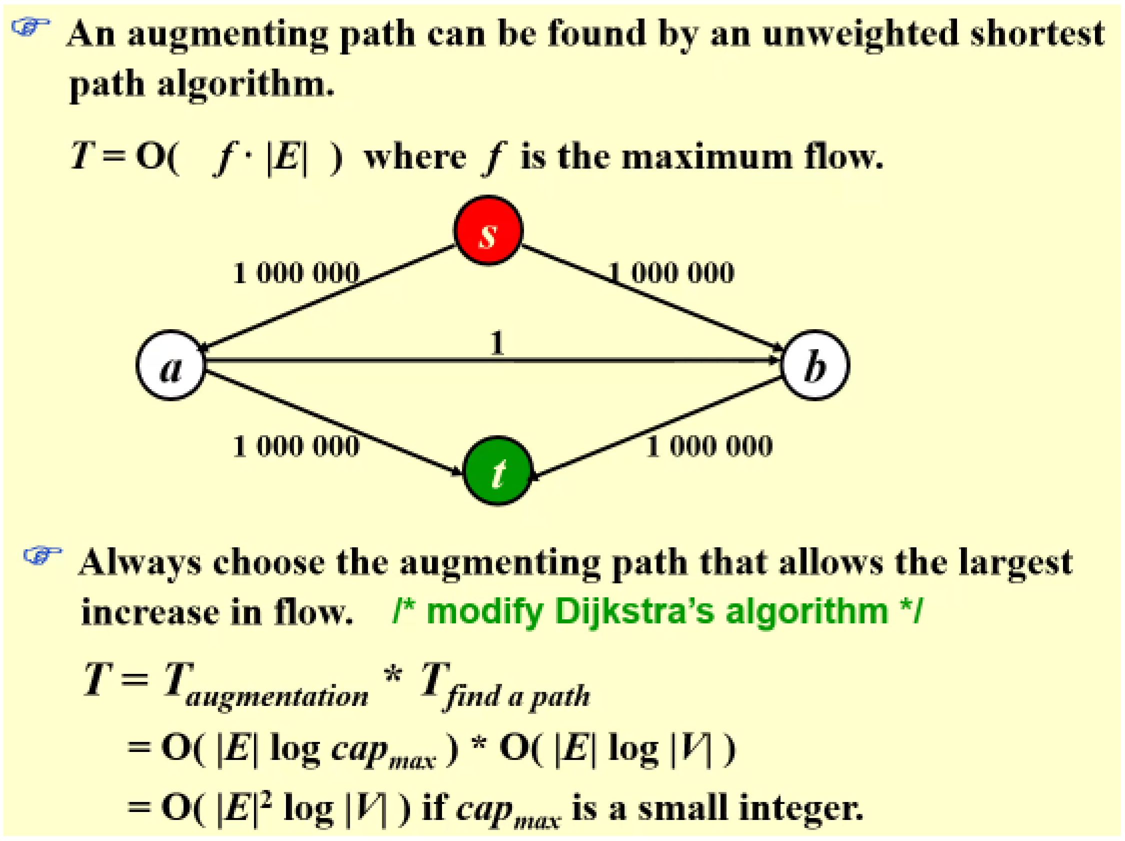
§ 4 Network Flow Problems

2. 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 .

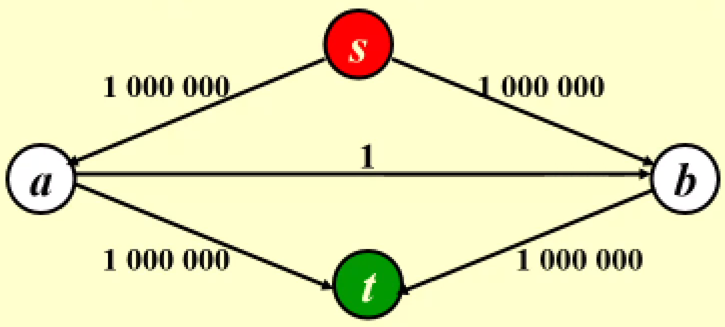
【Proposition】 If the edge capabilities are rational numbers, this algorithm always terminate with a maximum flow.

3.Analysis (if capacities are all integers)



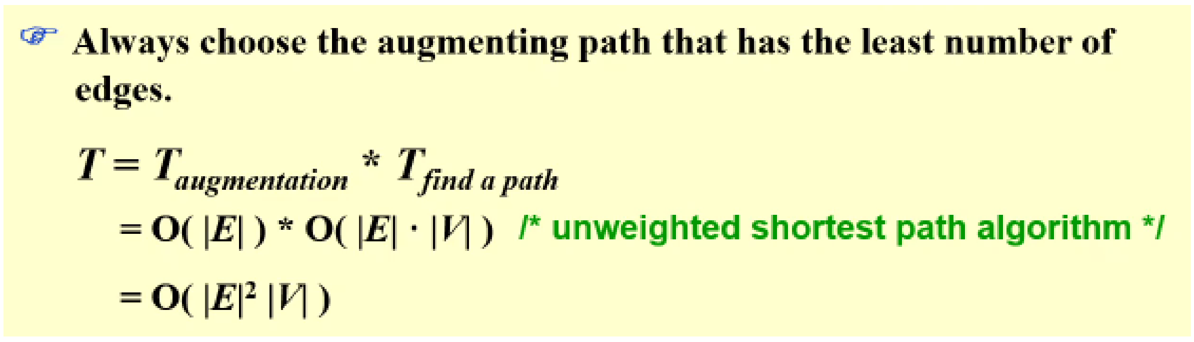
☞ An augmenting path can be found by an unweighted shortest path algorithm.

$T = O(\ f \cdot |E|\ )$ where f is the maximum flow.



☞ Always choose the augmenting path that allows the largest increase in flow. /* modify Dijkstra's algorithm */

$$T = T_{augmentation} * T_{find\ a\ path}$$
$$= O(\ |E| \log cap_{max}\ ) * O(\ |E| \log |V|\ )$$
$$= O(\ |E|^2 \log |V|\ ) \text{ if } cap_{max} \text{ is a small integer.}$$



☞ Always choose the augmenting path that has the least number of edges.

$$T = T_{augmentation} * T_{find\ a\ path}$$
$$= O(\ |E|\ ) * O(\ |E| \cdot |V|\ ) \quad \text{/* unweighted shortest path algorithm */}$$
$$= O(\ |E|^2\ |V|\ )$$

- Note
  - if every v has either a single incoming edge of capacity 1 or a single outgoing edge of capacity 1, the time bound is reduced to $O(|E||V|^{\frac{1}{2}})$.

φ5. Minimum Spanning Tree(最小生成树)

- Definition : a spanning tree of a graph G is a tree which consists of V(G) and a subset of E(G). (带有所有顶点n，边数最小(n-1))
- it's not unique

Greedy Method

1. use only edges within the graph
2. use only exactly |V|-1 edges
3. not use edges that would cause a circle

- **Solution**

1.Prim’s Algorithm - grow a tree(从点集出发)

1. 找一个顶点（任意）
2. 从已经被选过的顶点集合中出发，选与之相连的最小边权的边
3. 要求这条边不构成环（**并查集**）

2.Kruskal’s Algorithm - maintain a forest(从边出发)

```
void Kruskal (Graph G){
    T={};
    while (T contains less than |V|-1 edges && E is not empty){
        choos a least cost edge (v,w) from E;  // DeleteMin (Heap-restore)
        delete (v,w) from E;
        if ((v,w) does not create a cycle in T)
            add (v,w) to T;  // Union & Find
        else
            discard (v,w);
    }
}
```

$T = O(N)?$

φ6. Applications of Depth-First Search

```
void DFS(Vertex V){
    visited[V] = true;
    for (each W adjacent to V){
        if (!visited[W]) DFS(W);
    }
}
```

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

1.Undirected Graphs

- 多个component

```
void DFS_Component(graph G){
    for (each V in graph G){
        if (!visited[V]){
            DFS(V);
            printf("\n");
        }
    }
}
```

2. Biconnectivity

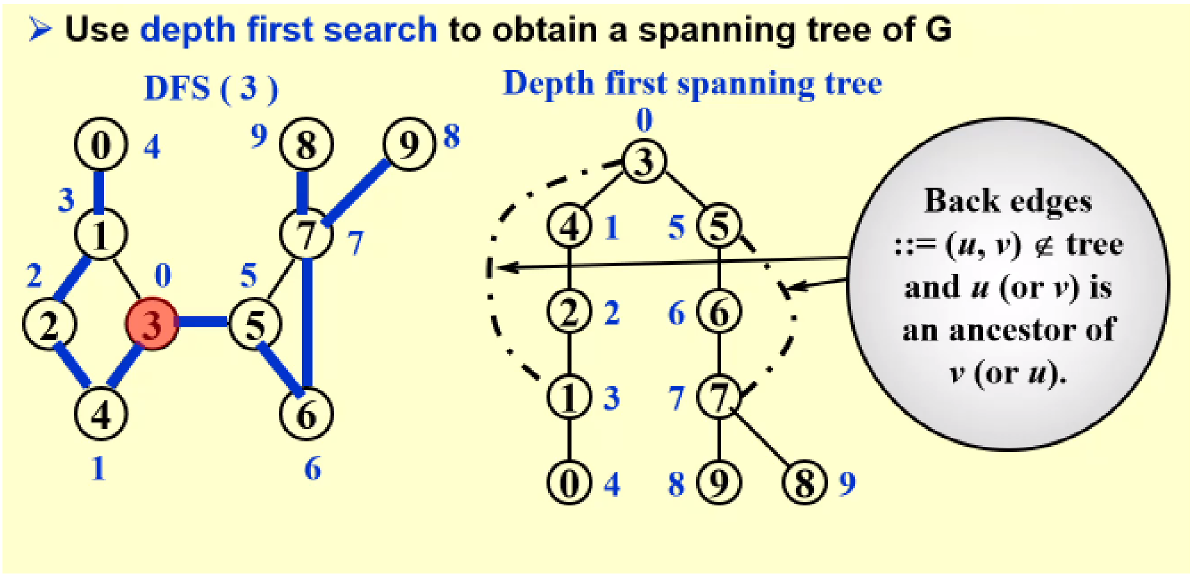
- v is an articulation point if G’ = DeleteVertex(G,v) has ar least 2 connected components.
    - 关键节点
- Biconnected : no articulation points
- 没有边被2个及以上的连通分量共享

- Finding the biconnected components of G

1. Use DFS to obtain a spanning tree of G
2. 给每个节点标上访问次序，做生成树

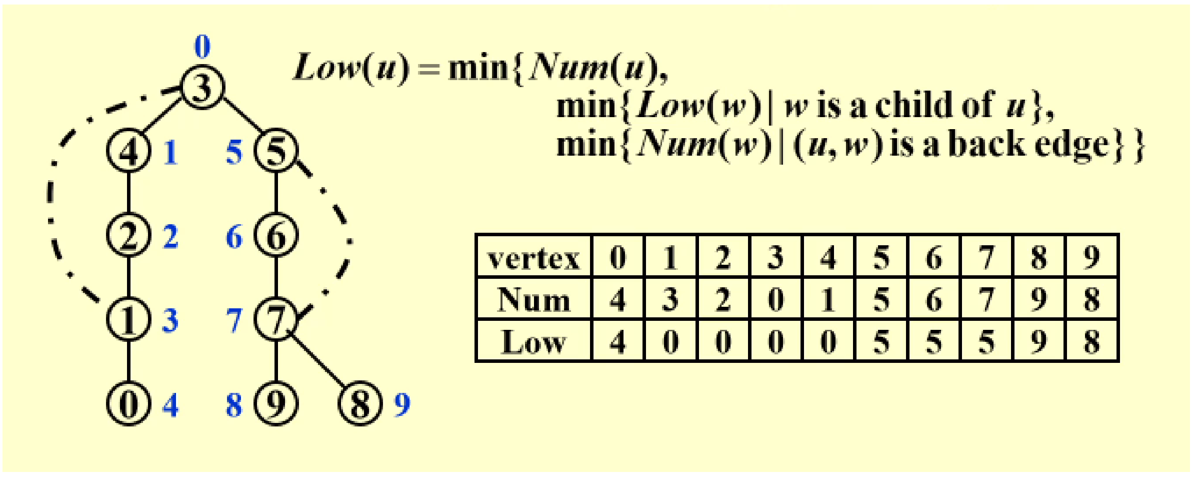
- Find the articulation points

1. The root is an articulation point if it has at least 2 children
2. Any other vertex u is an articulation pointt if it has at least 1 child, and it’s **impossible** to <u>move down at least 1 step</u> and then jump up to u’s ancestor.


➢ Use depth first search to obtain a spanning tree of G

Back edges ::= $(u, v) \notin$ tree and u (or v) is an ancestor of v (or u).

- Low(u) = min{Num(u),min{Loew(u)|w is a child of u},min{Num(w)|(u,w) is a back edge}}


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

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

- Thus, u is an articulation point if

1. u is the roott and has at least 2 children
2. u isn't the root and has at least 1 child such that Low(child) >= Num(u)

3. Euler Circuits(欧拉图)

- Euler tour: Draw each line exactly without lifting your pen from the paper
    - there are exactly 2 odd-degree point and **you must start at one of them**
- Euler curcuit: Draw each line exactly without lifting your pen from the paper AND finishi at the starting point
    - even-degree points only

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

6. Sort

φ1. Preliminaries

一般默认increasing order & integer array

φ2. Insertion Sort

```
void inserton_sort{
    int j,P;
    ElementType Tmp;

    for (P=1;P<N;p++){
        Tmp = A[P];
        for (j=P;j>0&&A[j-1]>Tmp;j--){
            A[j] = A[j-1];
        }
        A[j] = Tmp;
    }
}
```

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

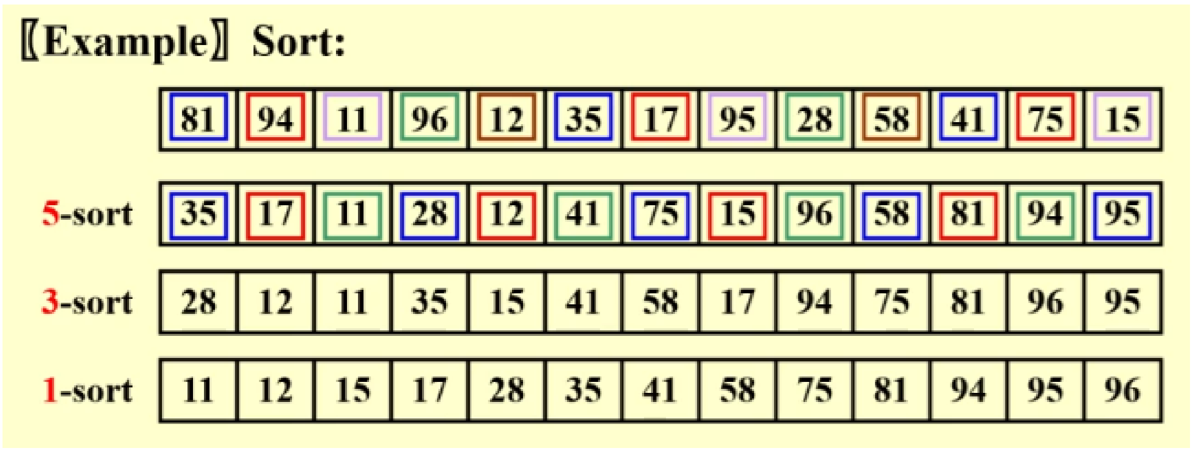
while the best case is $O(N)$

φ3. 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] (逆序对)

- Example : 34 8 64 51 32 21
    - 3+0+3+2+1=9
- The number of swaps is equal to the number of inversions by insertion sort.
- Thus $T(N,I)=O(I+N)$ while I is the number of invertions in the original array.
- The average number of inversions in an array of N distinct numbers is $\frac{N(N-1)}{4}$
- Any algprithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ on average

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

(1-sort is equal to insertion sort)

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

- Shell's increment sequence:
    - $h_t = N/2, h_k = h_{k+1}/2$

```c
void Shellsort(int A[],int N){
    int i,j,In; // In := Increment
    int tmp;
    for (In=N/2;In>0;In/=2){
        for (i=In;i<N;i++){
            tmp = A[i];
            for (j=i;j>=In;j-=In){
                if (tmp<A[j-In]) A[j]=A[j-In];
                else break;
                A[j] = tmp;
            }
        }
    }
}
```

The worst case : $T(N) = \Theta(N^2)$

Hibbard's Increment Sequence

$h_k = 2^k - 1$

The worst case : $T(N) = \Theta(N^{\frac{3}{2}})$

The average case : $T(N) = O(N^{\frac{5}{4}})$

φ5. HeapSort

Build Min-Heap

```
Algorithm 1{
    BuildHeap(H);  // O(N)
    for (i=0;i<N;i++) TmpH[i]=DeleteMin(H); // O(log N)
    for (i=0;i<N;i++) H[i]=TmpH[i];  // O(1)
}
```

$T(N) = O(N \cdot logN)$

Build Max-Heap

```
void Heapsort(int A[],int N){
    int i;
    for (i=N/2;i>0;i--) Percolate_Down(A,i,N);  // Build Heap
    for (i=N-1;i>0;i--){  // zjk这里应该写错了，不是0开始而是1开始
        Swap(&A[0],&A[i]);  // Delete Max
        Percolate_Down(A,0,i);
        /* 相当于把最大数A[0]移到末尾A[i],然后最大数就有序了 */
        /* 之后i--, 每次percolate down不用管已经排好的 */
    }
}
```

average : $2N \cdot logN - O(N \cdot log\ logN)$

Advantage : don't need to build a new array as an storage.

φ6. Mergesort（分而论之）

- Merge 2 sorted lists : use 2 ptr

```
void MSort(int A[],int tmpA[],int l,int r){
    int center;
    if (l<r){  // if there are elements to be sorted
        center = (l+r)/2;  // divide
        MSort(A,tmpA,l,center);  // T(N/2)
        MSort(A,tmpA,center+1,r);  // T(N/2)
        Merge(A,tmpA,l,center+1,r);  // O(N)
    }
}

void Mergesort(int A[],int N){  // 这是真正的函数入口
    int *tmpA;
    tmpA = malloc(N * sizeof(int));  // 因为tmpA之后要一直用，所以开地址，减小空间复杂度
    if(tmpA!=NULL){
        MSort(A,tmpA,0,N-1);
        free(tmpA);
    }else "error";
}
```

- 核心 : Merge

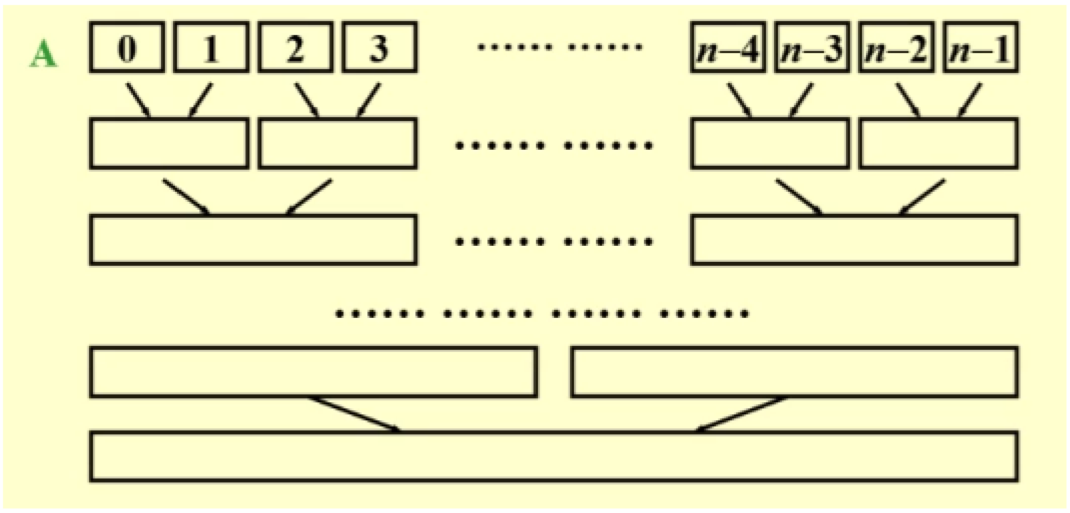
```
/* 其实L,R就是两个数组A,B，这里合起来了而已，用他模拟一个数组的排序，用LeftEnd表示A数组范围，RightEnd表示B数组范围 */
void Merge(int A[],int tmpA[],int L,int R,int RightEnd){
    int i,LeftEnd,Num,tmpPos;
    LeftEnd = R-1;
    tmpPos = L;
    Num = RightEnd-L+1;
    while(L<=LeftEnd && R<=RightEnd){  // 两个数组都没被穷尽
        if (A[L]<A[R]) tmpA[tmpPos++] = A[L++];
        else tmpA[tmpPos++] = A[R++];
    }
    while(L<=LeftEnd) tmpA[tmpPos++]=A[L++];  // B先被穷尽
    while(R<=RightEnd) tmpA[tmpPos++]=A[R++];  // A先被穷尽

    for (i=0;i<Num;i++,RightEnd--) A[RightEnd] = tmpA[RightEnd];
}
```

Analysis

$$T(1) = 1$$
$$T(N) = 2T(N/2) + O(N) = O(N \cdot logN)$$

Iterative version:



Disadvantage

1. **Need linear extra memory**
2. copy an array is slow

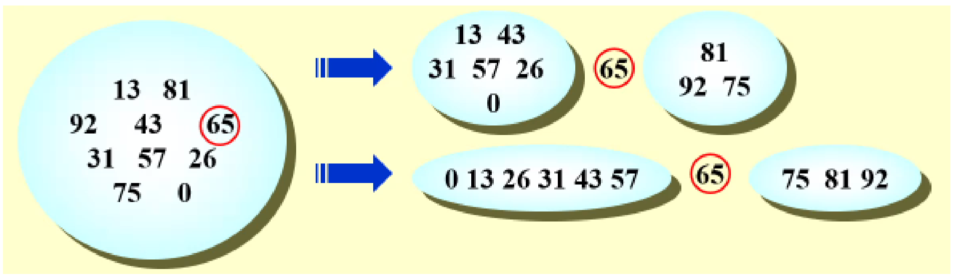
φ7. Quicksort

- The fastest known sorting algorithm in practice

1. Algorithm

- 选择一个质点 : **pivot**

```
void Qsort(int A[],int N){
    if (N<2) return;
    pivot = pick any element in A[];
    Partition S = {A[]\pivot} into 2 disjoint sets:
        A1 = {a in S | a <= pivot} And A2 = {a in S | a >= pivot};
    A=Qsort(A1,N1) union {pivot} union Asort(A2,N2);
}
```

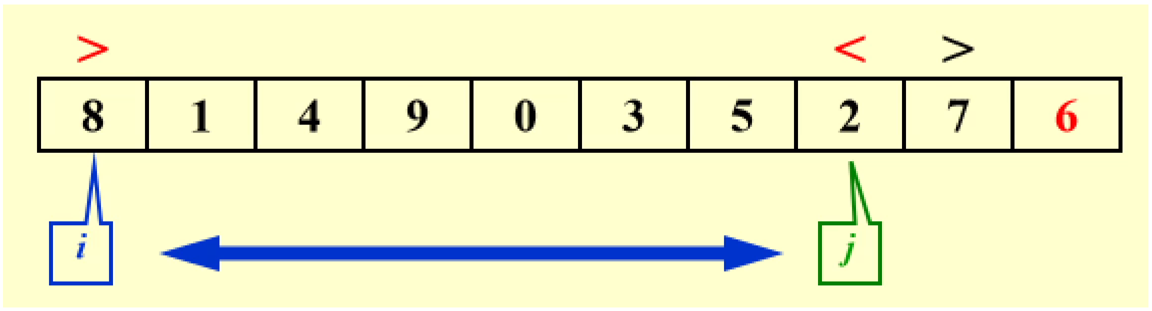


2. Picking the Pivot

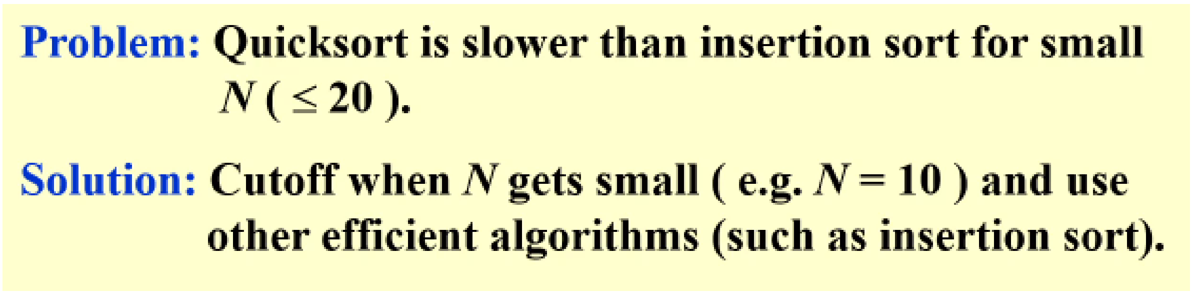
- A wrong way : Pivot = A[0]
  - The worst case : A[] is preesorted -> quicksort will take O(N^2) time to do nothing

- A safe Maneuver : Pivot = random select from A[]
- Median-of-Three Partitioning:
    - **Pivot = median (left, center, right)**

3. Partitioning Strategy



4. Small Arrays



Problem: Quicksort is slower than insertion sort for small N (≤ 20).

Solution: Cutoff when N gets small (e.g. $N = 10$) and use other efficient algorithms (such as insertion sort).

5. Algorithm

```
void QuickSort(int A[],int N){
    Qsort(A,0,N-1);
}

/* return median of l,c,r */
int Median3(int A[],int l, int r){
    int cnt = (l+r)/2;
    if (A[l]>A[c]) Swap(&A[l],&A[c]);
    if (A[l]>A[r]) Swap(&A[l],&A[r]);
    if (A[c]>A[r]) Swap(&A[c],&A[r]);
    /* invariant : A[l]<=A[cnt]<=A[r] */
    Swap(&A[cnt],&A[r-1]); /* Hide pivot to the last */
    /* only need to sort A[l+1]~A[r-2] */
    return A[r-1];  // return pivot
}

void Qsort(int A[],int l,int r){
    int i,j;
    int Pivot;
    if (l+cutoff<=r){  // if it's not too short
        Pivot = Median3(A,l,r);  // select pivot
        i=l;j=r-1;
        for(;;){ // infinity loop
            while (A[++i]<Pivot){}  // scanf from left
            while (A[--j]>Pivot){}
            if (i<j) Swap(&A[i],&A[j]);  // adjust
            else break;  // partition done
        }
        Swap(&A[i],&A[r-1]);  // pivot
        Qsort(A,l,i-1);
        Qsort(A,i+1,r);
        /* don't need to sort A[i] */
    }else InsertionSort( A+l,r-l+1);  // do other sort
}
```

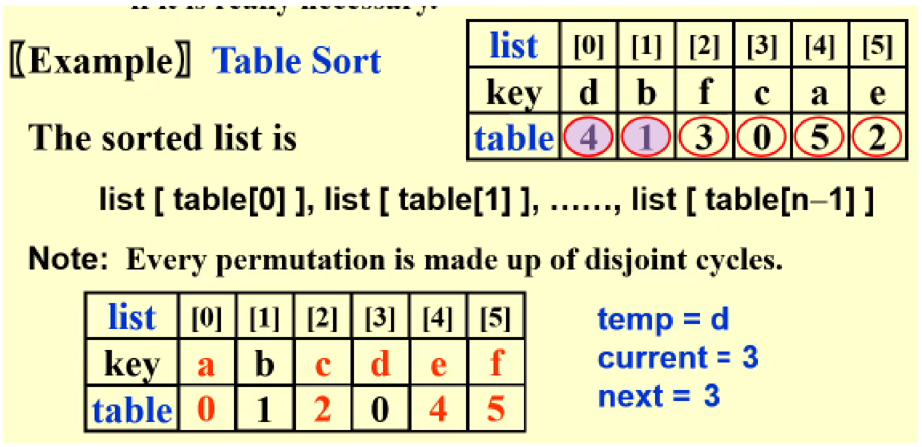
6. Analysis

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

- Whe Worst Case :
  - $i = 0 \ \Rightarrow T(N) = T(N-1) + cN \ \Rightarrow T(N) = O(N^2)$
- The Best Case :
  - $T(N) = T(N/2) + cN = O(N \cdot logN)$
- The Average Case :
  - $T(N) = \frac{2}{N}\left[\Sigma_{j=0}^{N-1} T(j) + cN\right] = O(N \cdot logN)$

φ8. Sorting Large Structures (Table Sort)

- Problem : Swapping large structures can be verry much expensive
- Solution : Add a pointer field to the structure and swap pointers instead -> indirect sorting. (Physically rearange the structures at last if it's 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	0	4	5

temp = d
current = 3
next = 3

φ9. A general Lower Bound for sorting

- **Any** algorithm that sorts by comparisons only must have a **worst case** computing time of $\Omega(N \cdot logN)$
- Proof : there are N! different possible results so the tree's height k>log(N!)+1 ,since N!>(N/2)$^{N/2}$.

φ10. Bucket Sort and Radix Sort

Bucket Sort

- 基于映射的排序

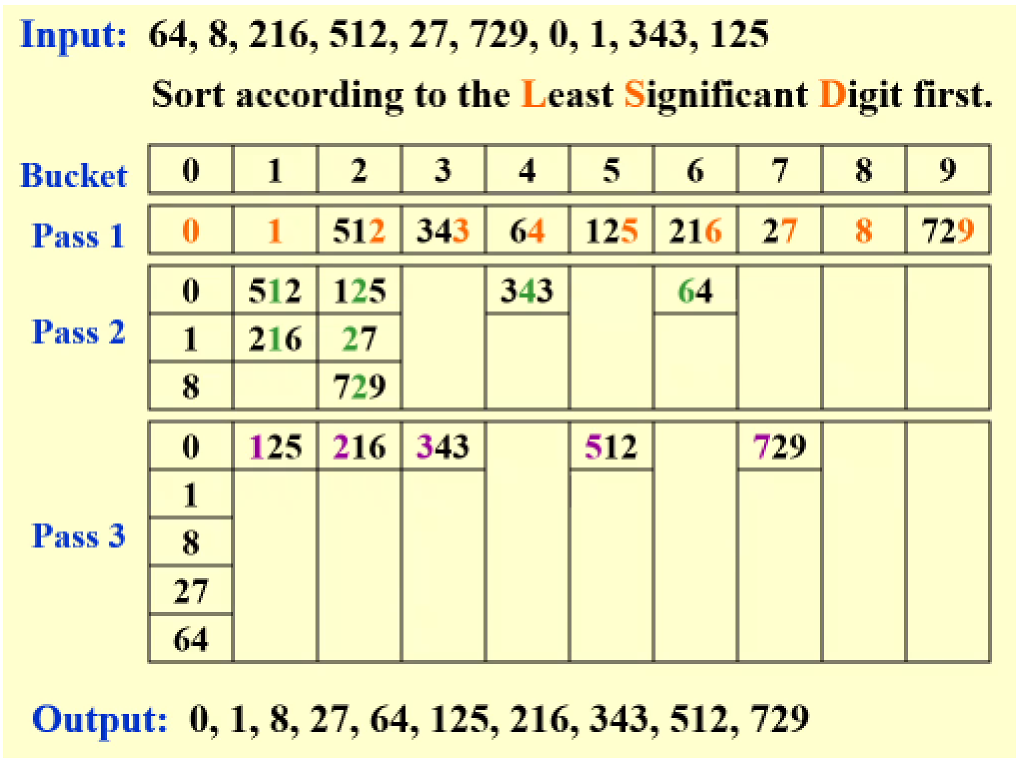
```
Algorithm{
    initialize couunt[];
    while (read in students''grade) insert to list count[stdnt.grade];
    for (i=0;i<M;i++){
        if (count[i]) output list count[i];
    }
}
```

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

N:总数,M:映射范围

Radix Sort

- 基于映射的排序: while M >> N



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

Sort according to the **L**east **S**ignificant **D**igit 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

Suppose the record R_i has r keys

K_i^j : the j=th key of record R_i

K_i^0 : the most significant key of record R_i

K_i^{r-1} : the least ......

MSD Sort

1. Sort on K^0 : for example create 4 buckets
2. Sort each bucket independently

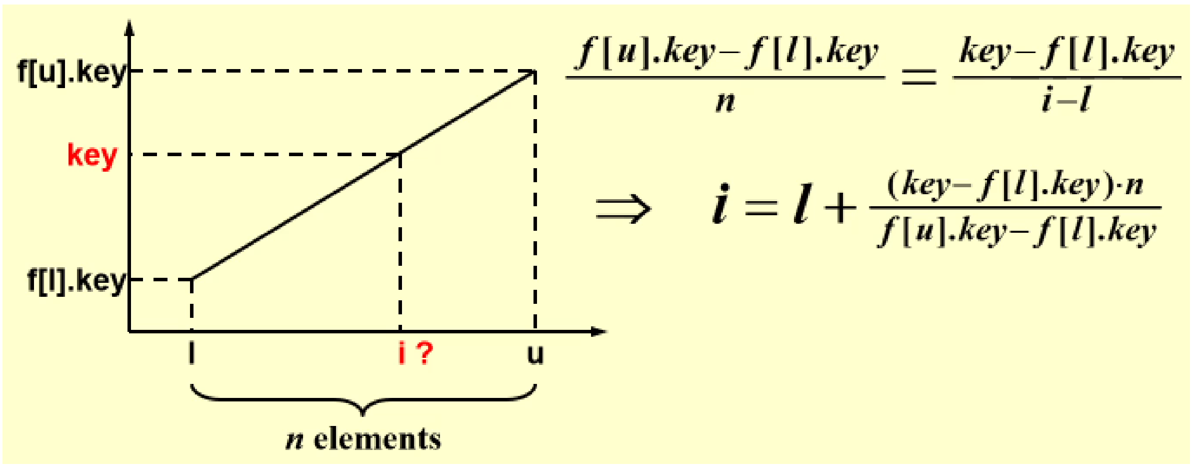
LSD Sort

1. Sort on K^1
2. Reform them into single pile
3. Create 4 buckets and resort

7. HASHING（哈希）

- Searching（基于公式的查找）
- Interpolation Search:
    - Find **key** from a sorted list f[l].key~f[u].key
    - $$\frac{f[u].key - f[l].key}{n} = \frac{key - f[l].key}{i - l}$$

-
  ```
  if (f[i].key < key) l=i;
  else u = i;
  ```



$$\frac{f[u].key - f[l].key}{n} = \frac{key - f[l].key}{i - l}$$

$$\Rightarrow \quad i = l + \frac{(key - f[l].key) \cdot n}{f[u].key - f[l].key}$$

φ1. General Idea

- Symble Table (== Dictionary) ::= {<name,attribute(属性)>}

ADT

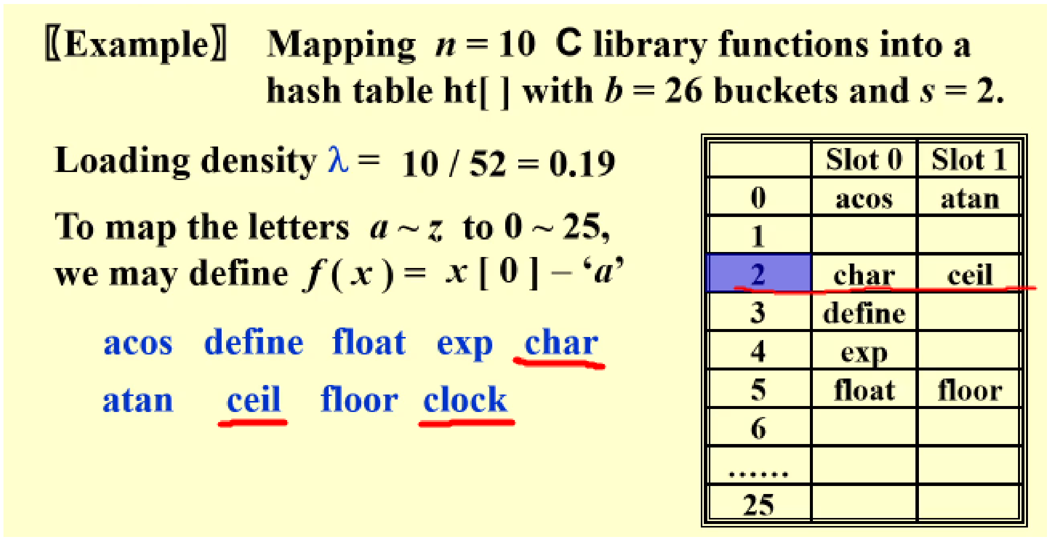
- Objects : A set of name-attribute pairs, where the names are unique
- Operations:
  - Create(TableSize)
  - Boolean IsIn (symtab,name)
  - **Attribute Find (symtab,name)**
  - **Insert**
  - **Delete**

Hash Tables

for each identifier x, we define a hash function

- f(x) = position of x in ht[] (i.e. the index of the bucket that contains x)
- T ::= total number of distinct possible value for x
- n ::= total number of identifiers in ht[] (already done)
- identifier density ::= n/T (到课率)
- loading density ::= n/(s b) (s列slot b行bucket)

- A collision (碰撞，难以避免，需要解决)
  - occurs when we hash 2 nonidentical identifiers into the same bucket, i.e. $f(i_1)=f(i_2)$ when $i_1 \neq i_2$
- An overflow (需要避免)
  - occurs when we hash a new identifier into a full bucket



〖Example〗 Mapping $n = 10$ C library functions into a hash table ht[ ] with $b = 26$ buckets and $s = 2$.

Loading density λ = 10 / 52 = 0.19

To map the letters $a \sim z$ to $0 \sim 25$, we may define $f(x) = x[0] - 'a'$

acos define float exp char
atan ceil floor clock

	Slot 0	Slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
……		
25		

Without overflow:

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

φ2. Hash Function

- Properties of f:
  - f(x) must be easy to compute and minimizes the number of collisions
  - f(x) should be unbiased. That is, for any x and any i, we have that Probability **(f(x)=i)=1/b**. Such kind of a hash function is called a **uniform hash function**
  - (若对于关键字集合中的任一个关键字，经散列函数映象到地址集合中任何一个地址的概率是相等的，则称此类散列函数为均匀散列函数（Uniform Hash function），这就是使关键字经过散列函数得到一个“随机的地址”，从而减少冲突)
  - $f(x) = x\,\%\,TableSize$ if x is an integer
  - $f(x) = (\Sigma x[i]\,\%\,TableSize)$ if x is a string
  - $f(x) = (\Sigma x[N-i-1] * 32^i)\,\%TableSize$

```c
Index Hash3(const char *x,int Tsize){
    unsigned int HashVal = 0;
    while (*x != '\0')
        HashVal = (HashVal << 5) + *x++;
    return HashVal % Tsize;
}
```

φ3. Separate Chaining

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

```c
struct ListNode;
typedef struct ListNode *Position;
struct HashTbl;
typedef struct HashTbl *HashTable;
struct ListNode {
    int Element;
    Position Next;
};
typedef Position List;
struct HashTbl {
    int TableSize;
    List *TheLists;
}
// ListNode = *Position = *List
// HashTbl = *HashTable
```

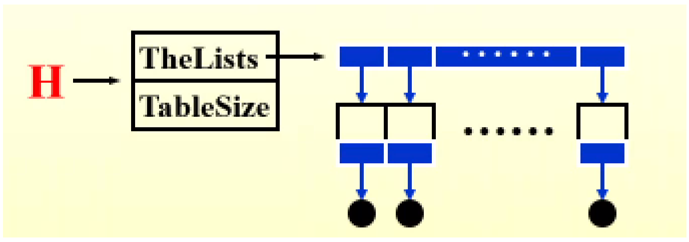
Create

```
HashTable InitializeTable (int Tsize){
    HashTable H;
    int i;
    if (Tsize < MinTableSize) 'error';
    H = malloc(sizeof(struct HashTbl));  // allocate
    if (H == NULL) 'error';

    H->TableSize = NextPrime(Tsize);  // better be prime
    H->TheLists = malloc(sizeof(List)*H->TableSize);  // allocate

    if (H->TheLists == NULL) 'error';

    for (i=0;i<H->TableSize;i++){
        H->TheLists[i] = malloc(sizeof(struct ListNode));  // allocate
        if (H->TheLists[i]==NULL) 'error';
        else H->TheLists[i]->Next = NULL;
    }
}
```



Find

```
Position Find(int Key,HashTable H){
    Position P;
    List L;
    // Position 和 List 应该是一样的吧？
    L = H->TheLists[Hash(Key,H->Tsize)];  // Hash() 就是某个特定的哈希函数

    P=L->Next;
    while(P && P->Element != Key) P=P->Next;
    return P;
}
```

Insert

```c
void Insert(int Key,HashTable H){
    Position Pos,New;
    List L;
    Pos = Find(Key,H);
    if (!Pos){  // Pos == NULL 才需要插入
        New = malloc(sizeof(struct ListNode));
        if (!New) 'error';
        else {
            L=H->TheList[Hash(key,H->Tsize)];
            New->Next = L->Next;
            New->Element = Key;
            L->Next = New;
            // 链表的标准插入
        }
    }
}
```

Tip: Make the TableSize about as large as the number of keys expected (i.e. to make the loading density factor λ≈1).

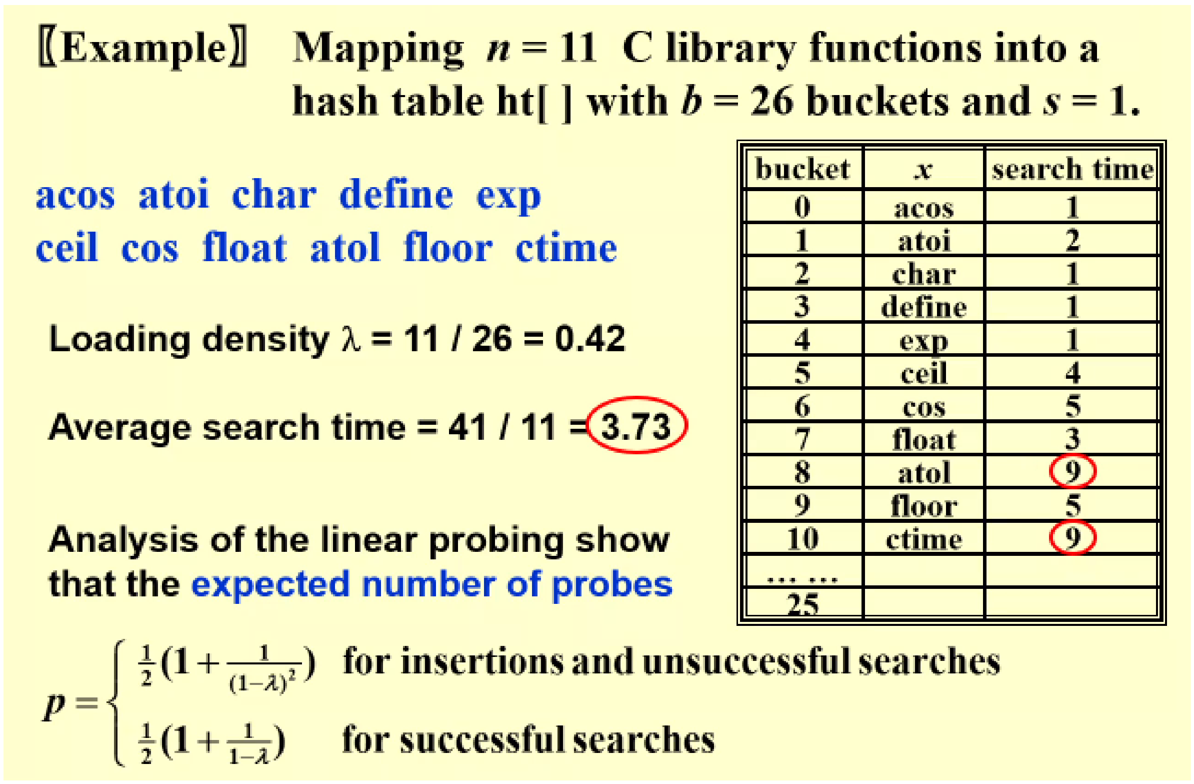
φ4. Open Addressing

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

```
void inset key into an array of hash table{
    index = hash(key);
    initialize i = 0;  // the counter of probling
    while (collision at index){
        index = (hash(key)+f(i)) % Tsize;  // 每次从最佳位往后移f(i)直到有空位
        if (table is full) break;  // overflow
        else i++;
    }
    if (table is full) 'error';
    else insert key at index;
}
```

1. Linear Probing

- $f(i) = i$ a linear function (一次往后移一位直到找到空位)



【Example】 Mapping $n = 11$ C library functions into a hash table ht[ ] with $b = 26$ buckets and $s = 1$.

acos atoi char define exp
ceil cos float atol floor ctime

bucket	x	search time
0	acos	1
1	atoi	2
2	char	1
3	define	1
4	exp	1
5	ceil	4
6	cos	5
7	float	3
8	atol	9
9	floor	5
10	ctime	9
… …		
25		

Loading density $\lambda = 11 / 26 = 0.42$

Average search time = 41 / 11 = 3.73

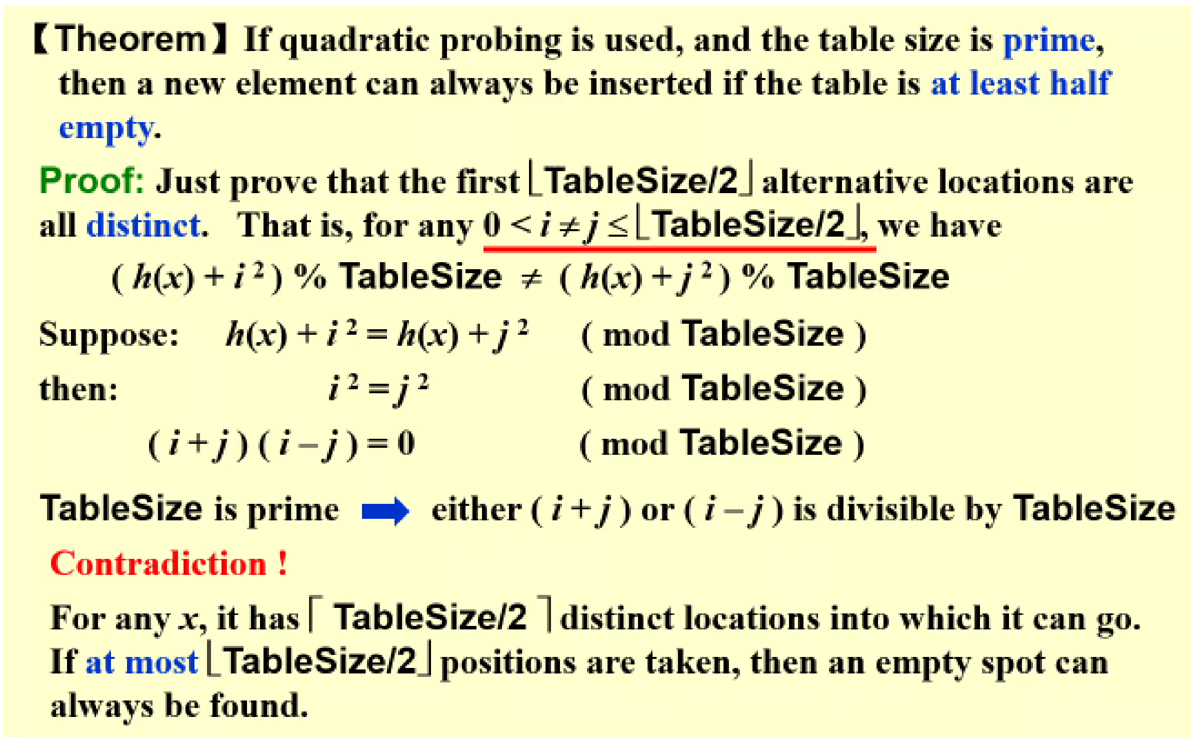
Analysis of the linear probing show that the expected number of probes

$$p = \begin{cases} \frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right) & \text{for insertions and unsuccessful searches} \\ \frac{1}{2}\left(1+\frac{1}{1-\lambda}\right) & \text{for successful searches} \end{cases}$$

- Although average time is small but the worst case time can be large (主聚集，如图中2个9)

2. Quadratic Probing

- $f(i) = i^2$ a quadratic function



【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 ⌊TableSize/2⌋ 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: $h(x) + i^2 = h(x) + j^2 \quad (\text{mod TableSize})$

then: $i^2 = j^2 \quad (\text{mod TableSize})$

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

TableSize is prime ➡ either $(i + j)$ or $(i - j)$ is divisible by TableSize

Contradiction !

For any x, it has ⌈ TableSize/2 ⌉ distinct locations into which it can go. If at most ⌊TableSize/2⌋ 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 $f(i) = +-i^2$ can probe the entire table

Find

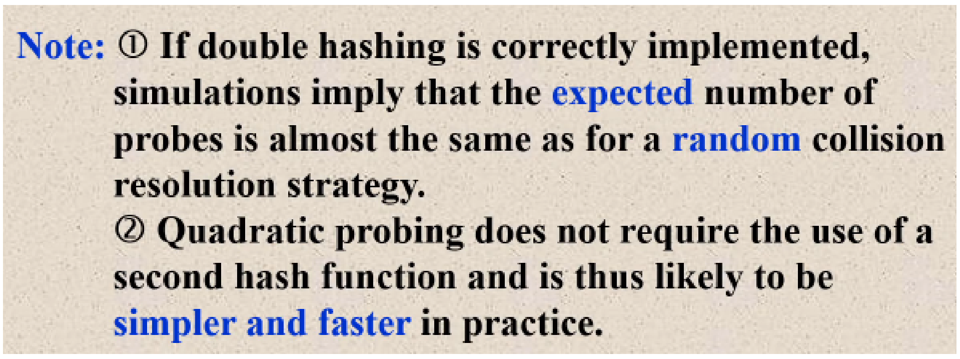
```c
Position Find (ElementType Key, HashTable H){
    Position CurrentPos;  // 这里的Position就是int吗？
    int CollisionNum = 0;
    CurrenPos = Hash(Key,H->Tsize);
    while (H->TheCells[CurrentPos].info != Empty && H->TheCalls[CurrentPos].Element != Key){
        CurrentPos += 2* ++CollisionNum - 1;  // 2i-1是平方的差值，+2i-1比平方快
        if (CurrentPos >= H->TableSize) CurrentPos -= TableSize;  // 相当于mod，减法比mode快
    }
    return CurrentPos;
}
```

Insert

```c
void Insert (int Key,HashTable H){
    Position Pos;
    Pos = Find(Key,H);
    if (H->TheCells[Pos].info != Legitimate){
        H->TheCells[Pos].info = Legitimate;
        H->TheCells[Pos].Element = Key;
    }
}
```

3. Double Hashing

- $f(i) = i * hash_2(x)$
    - hash$_2$ 不恒等于 0
    - eg. $hash_2(x) = R - (x\%R)$ **(R:prime < Tsize)**

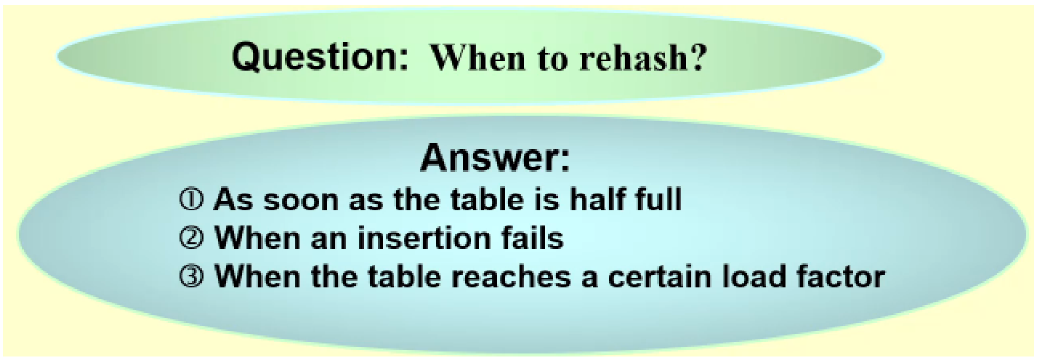


Note: ① If double hashing is correctly implemented, simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy.
② Quadratic probing does not require the use of a second hash function and is thus likely to be simpler and faster in practice.

φ5. Rehashing

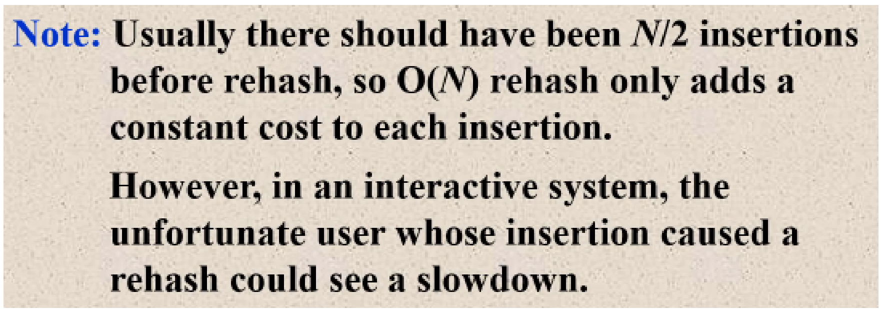
- Build another table (at least) twice as big (and it must be prime)

    - eg. 10 --double--> 10*2=20 --prime--> 23
- scan down the entire original hash table for non-deleted elements
- use a new function to hash those elements into the new table
- T(N)=O(N)



Question: When to rehash?

Answer:
① As soon as the table is half full
② When an insertion fails
③ 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.

均分到每一次，其时间复杂度是线性的

---

1. 其中preorder，postorder和inorder都是recursion 并且时间复杂度均为O（N）； ↩