## Common Data Structure Operations

| Data Structure | Time Complexity | | | | | | | | Space Complexi |
|---|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | | Worst |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |
| Array | Θ(1) | Θ(n) | Θ(n) | Θ(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| Stack | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Queue | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Singly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Doubly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Skip List | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| Hash Table | N/A | Θ(1) | Θ(1) | Θ(1) | N/A | O(n) | O(n) | O(n) | O(n) |
| Binary Search Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| Cartesian Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(n) | O(n) | O(n) | O(n) |
| B-Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Red-Black Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Splay Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| AVL Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| KD Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |

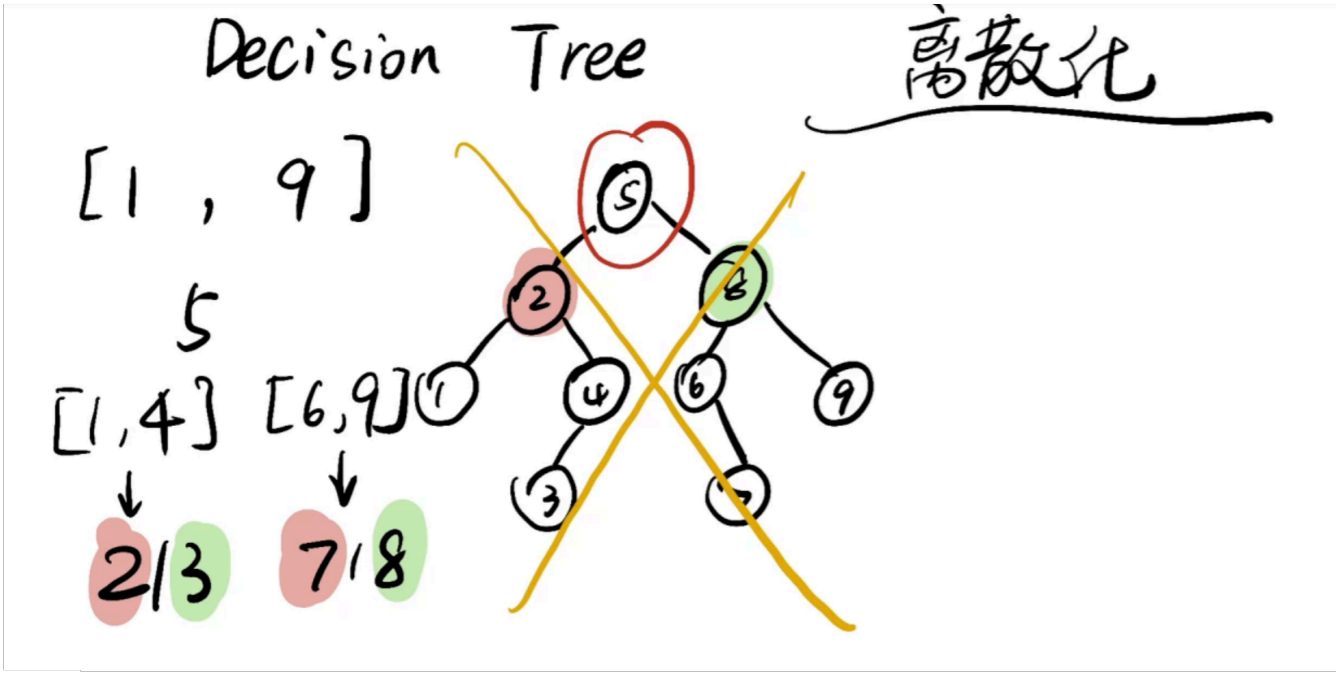


# Sequential List/Linked List

# Threaded Binary Tree线索二叉树

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者后继的线索。

# Decision Tree二分搜索决策树




该决策树每次选择中间的数进行判断，那么中间数有两种取整方法，向上取整或者向下取整。例如[1,4],[6,9]取中间值可以是2/3，7/8。则如果是decision tree的话，要么同时取2/7，要么同时取3/8.所以上图不是

## 完全二叉树

指的是每一个节点对应的位置都比较好，空位都在最后一排。完全二叉树的特殊形式就是满二叉树，即每一个节点都挂满了子树。

## 平衡二叉树AVL

它具有如下几个性质：

1. 可以是空树。
2. 假如不是空树，任何一个结点的左子树与右子树都是平衡二叉树，并且高度之差的绝对值不超过 1。

平衡之意，如天平，即两边的分量大约相同。

下图不是AVL是因为节点60下方没有tree，此时左右子树高度差为2.

平衡因子：左子树高度-右子树高度

## Binary Search Tree二分查找树

1. find  T(N)=O(d)

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

2. insert  T(N)=O(d)

```
SearchTree Insert(SearchTree T)
{
  if(T==NULL) T=malloc(sizeof(struct TreeNode));
  else
  {
    if(X<T->Element)  T->Left=Insert(X,T->Left);
    else if(X>T->Element) T->Right=Insert(X,T->Right);
  }
  return T;
}
```

3. delete **T(N)=O(d)**
   - Delete a leaf node: reset its parent link to NULL
   - Delete a degree 1 node: replace the node by its single node
   - Delete a degree 2 nodes: 用左子树的最大值或右子树的最小值节点替换

```
SearchTree Delete(ElementType X,SearchTree T)
{
  int TmpCell;
  if(T==NULL) Error("Element Not Found");
  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)
    {
      TmpCell=FindMin(T->Right);
      T->Element=TmpCell;
      T->Right=Delete(T->Element,T->Right);
    }
    else
    {
      TmpCell=T;
      if(T->Left==NULL)
```

```
            T=T->Right;
        else if(T->Right==NULL)
            T=T->Left;
        free(TmpCell);
    }
  return T;
}
```

## Heap堆

建堆O(N) 不是$O(NlogN)$,$O(NlogN)$是最坏情况。

建堆可以通过随机，然后percolate

1. insert　T(N)=O(logN)

```
void Insert(ElementType X,PriorityQueue H)
{
    int i;
    if(IsFull(H))
    {
        printf("Priority queue is full!");
        return;
    }
  //从最后一个位置插入，然后percolate up
    for(i=++H->Size;H->Elements[i/2]>X;i/=2)
        H->Elements[i]=H->Elements[i/2];
    H->Elements[i]=X;
}
```

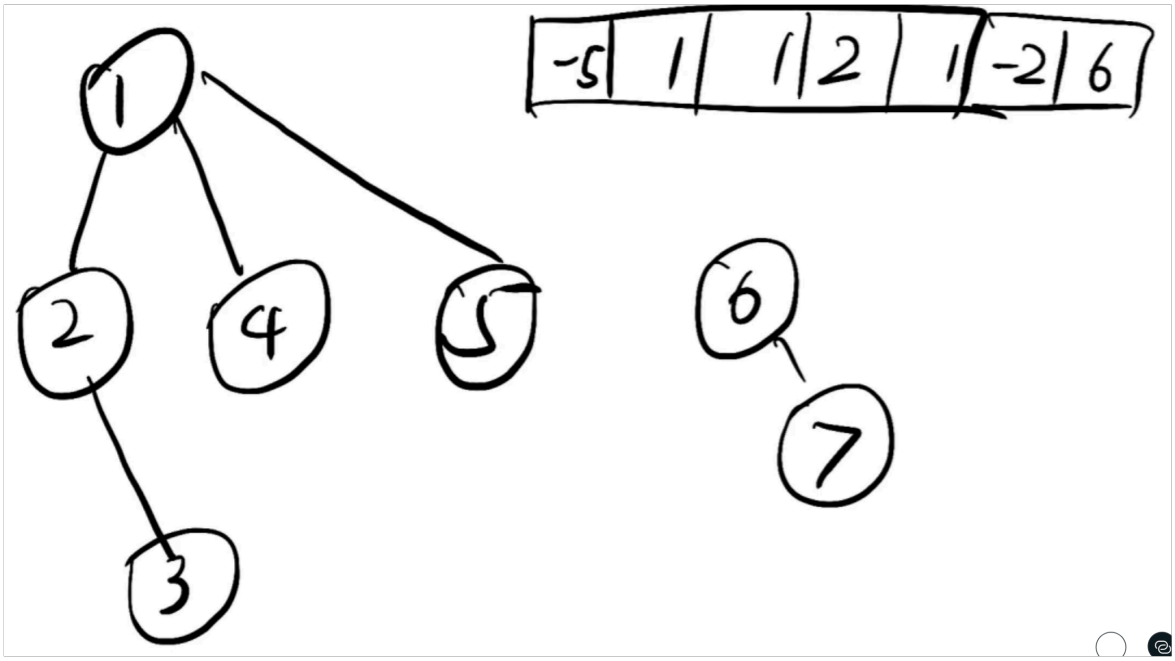
2. DeleteMin　T(N)=O(logN)

```
ElementType DeleteMin(PriorityQueue H)
{
    int i;
    ElementType MinElement, LastElement;
    if(IsEmpty(H))
    {
        printf("Priority queue is empty!");
        return;
    }
    MinElement=H->Elements[0];//堆顶是最小的
    LastElement=H->Element[H->Size--];
    for(i=1;i*2<=H->Size;i=Child)
    {
        child=i*2;
        if(child!=H->Size&&H->Elements[child]>H->Elements[child+1])//右孩子不为空且比左边小
            child++;
        if(LastElement>H->Elements[child])
```

```
            H->Elements[i]=H->Elements[child];
        else break;
    }
    H->Element[i]=LastElement;
```

## Disjoint Set并查集

负数表示没父亲节点，并且该集合里面包括该节点一共有多少节点。例如-1表示只有这么一个节点。




### 合并disjoint

代码实现

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

### Find

代码实现

```
SetType Find(ElementType X, DisjSet S)
{
    for(; S[X]>0;X=S[X])//每次都去找father结点，直到找到负数（root
    return X;
}
```

### Union by Size

```
void SetUnion (DisjSet S,SetType root1, SetType root2)
{
    if(S[root1]<=S[root2])//root1比root2小，但是他们又都是负数；说明root1比root2集合要大
    {
        S[root1]+=S[root2];
        S[root2]=S[root1];
    }
    else
    {
        S[root2]+=S[root1];
        S[root1]=S[root2];
    }
}
```

### Union by height

```
void SetUnion (DisjSet S,SetType root1, SetType root2)
{
    if(S[root2]<S[root1])//root2比root1要更深
        S[root1]=root2;//root2是新根
    else
    {
        if(S[root1]==R[root2])//两个深度相等的时候
          S[root1]--;//随便找一个树放在下方
        S[root2]=root1;//root1是新根
    }
}
```

### path compression

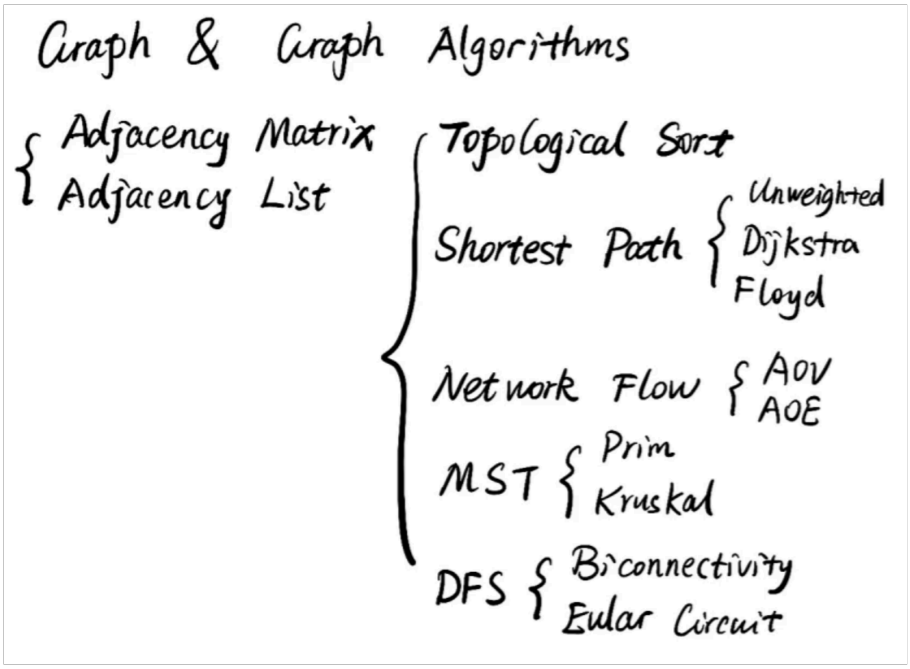
X到root路径上的每个节点都使他的父节点变成 root。

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

### 代码实现

```
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




## 强连通图/弱连通图（都是有向图）

**连通图**

在无向图中, 若从顶点v1到顶点v2有路径, 则称顶点v1与v2是连通的。如果图中任意一对顶点都是连通的,则称此图是连通图。

**强连通图**：即有向图$G=(V,E)$ 中，若对于V中任意两个不同的顶点$x$和$y$，都存在从$x$到$y$以及从$y$到$x$的路径，则称$G$是强连通图。相应地有强连通分量的概念。强连通图只有一个强连通分量，即是其自身；非强连通的有向图有多个强连分量。

**弱连通图**：将有向图的所有的有向边替换为无向边，所得到的图称为原图的基图。如果一个有向图的基图是连通图，则有向图是弱连通图。

无向图的连通分量即为自身， 有向图需考虑从强连通关系去分析划分。

## Topological Sort拓扑排序

要求**有向无环图**

**算法1**：普通的拓扑排序复杂度$O(|V|^2)$. 每次找到**入度为0**的点然后与它相邻的点的入度分别-1.

```
void TopSort(Graph g)
{
    int counter;
    vertex v,w;
    for(center=0;center<numvertex;counter++)
    {
        v=FindNewVertexDegreeZero();//找到入度为0的节点
        if(v==notavertex)
        {
            printf("error!Graph has a cycle!");
            break;
        }
        Topnum[v]=counter;//mark the sequence
        for(each w adjacent to v)
          indegree[w]--;
    }
}
```

**算法2:**优化的拓扑排序复杂度$O(|E|+|V|)$ 如果入度为0，则放到队列当中

```
void TopSort(graph g)
{
    Queue Q;
    int count=0;
    vertex v,w;
    Q= CreateQueue(NumVertex);
    MakeEmpty(Q);
    for(each vertex v)
      if(indegree[v]==0) enqueue(Q,v);//入队
    while(isEmpty(Q))//BFS
    {
        v=dequeue(Q);
        top[Num]=++count;
        for(each w adjacent to v)
          if(--indegree[w]==0) enqueue(Q,w);
    }
    if(count!=numvertex)
       printf("Graph has a cycle!");
```

```
    DisposeQueue(Q);//free memory
}
```

注意拓扑排序不适用于有环图，此时会无法遍历每一个节点。

# 最短路算法

## Single-Source Shortest Path单源最短路径

**松弛操作（Relaxation）**

两个节点v1,v2之间的距离初始化为无穷大，然后每次找到最短路之后进行修改–松弛。

## Unweighted Graph

**BFS(无权图)**

用队列实现 – 松弛思想 – 贪心算法

- Table[i].Dist ::=distance from to $s$ to $v_i$ /* initialized to be ∞ except for $s$*/此处有用到松弛
- Table[ i ].Known ::= 1 if $v_i$ is checked; or 0 if not
- Table[ i ].Path ::= for tracking the path /* initialized to be 0 */

贪心算法（分成2部分，一部分已知与该点的最短路距离

worst case：所有节点在同一条线上　　$T = O(|V|^2)$

**Improvement:**用队列来解决问题

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

## Weighted Graph

**1. Dijkstra 算法（有权图-不考虑有负边情况）**

2个数组，一个叫做Dist[],path[]　　**贪心算法**

Dist[] 初始化为∞，用于维护原点到当前节点的最短路距离。

Path[] 用于记录前一个节点。

- Dijkstra算法对于稠密图而言较为好，时间复杂度是$T = O(|V|^2 + |E|)$
- 优化方式：堆优化，时间复杂度可以达到$T = O(log|V|)$

`DeleteMin`操作直接通过删除堆顶实现

- 更新处理方式：
  - Method1: `DecreaseKey`——$O(log|V|)$

  $T = O(|V|log|V| + |E|log|V|)$
  - Method2: keep doing `DeleteMin` until an unknown vertex emerges

    $T = O(|E|log|V|)$ but requires |E| to `DeleteMin` with `|E|` space.

**代码实现：**

```
void Dijkstra(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)//更新操作
                  T[w].path=v;//path[]数组用来标记前一个节点
              }
        }
    }
}
```

堆优化：

```
void Dijkstra(Table T)
{
    Queue Q;
    Vertex v,w;
    Q=CreateQueue(numvertex);
    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 already in Q)
             enqueue(Q);
        }
    }
    DisposeQueue(Q);//free memory
}
```

**2. Floyd 算法（有权图-可以解决有负边的情况）**

Floyd算法可以处理边权为正或者为负的图。 负权边 – 会导致有无限的圈

多元最短路，可以求出来所有点到所有点之间的距离 DP

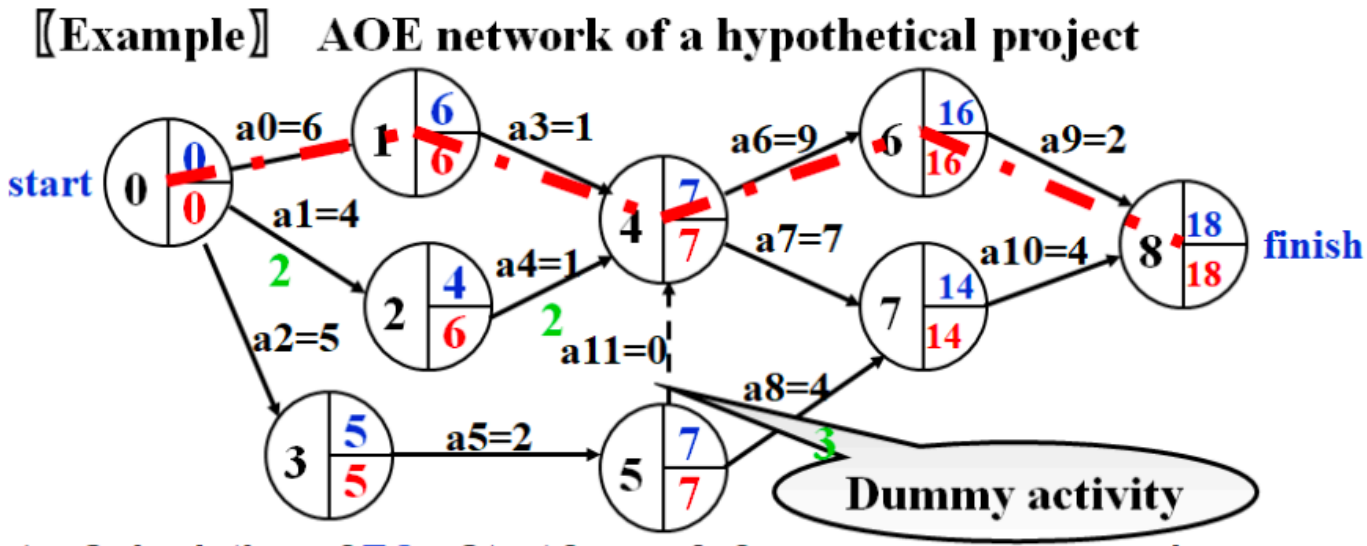
Floyd算法适用于APSP(AllPairsShortestPaths)，是一种动态规划算法，对于稠密图而言效果最佳。此算法简单有效，由于三重循环结构紧凑，对于稠密图，效率要高于执行|V|次Dijkstra算法。

- **APSP(All Pairs Shortest Paths)**
  - Method 1

    Use **single-source algorithm** for time。
  - Method 2–动态规划
    - works fast on sparse graph.
    - algorithm given in Chapter 10, works faster on dense graphs.

**AOE(Activity On Edge Network)：**

若在**带权**有向图中，以顶点表示事件，以有向边表示活动，边上的**权值表示活动的开销**（如该活动持续的时间），则此带权的有向图称为AOE网。




- Calculation of EC: Start from v0, for any $a_i$ = <v, w>, we have

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

- Calculation of LC: Start from the last vertex v8, for any $a_i$ = <v, w>, we have $LC[v] = \min_{(v,w)\in E}\{LC[w] - C_{v,w}\}$
- Slack Time of <v,w> = $LC[w] - EC[v] - C_{v,w}$
- Critical Path ::= path consisting entirely of zero-slack edges.

特别需要注意：EC总是从起点开始计算，但是LC总是从终点倒着减回去。

EC找从起点开始到这个点最长的时间；LC找从终点减回去剩余最短的时间。

闲暇时间Slack Time <v,w>：$= LC[w] - EC[v] - C_{v,w}$

**AOV(Activity On Vertex Network)：**

在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系。这样的有向图为顶点表示活动的网，我们称为AOV网。

# 网络流

`maximum flow==minimum cut`最大流和最小割是一样的。只割最少条边就可以断开S和D之间的路

# Minimum Spanning Tree最小生成树

- 两种都是贪心算法，但是复杂度不一样（与稠密度有关
- 最小生成树可能不唯一

**Prim's Algorithm “加点法”**

每次找到与v相邻的最短且不在已选集合中的那个顶点，加入生成树中。

**Kruskal's Algorithm “加边法”**

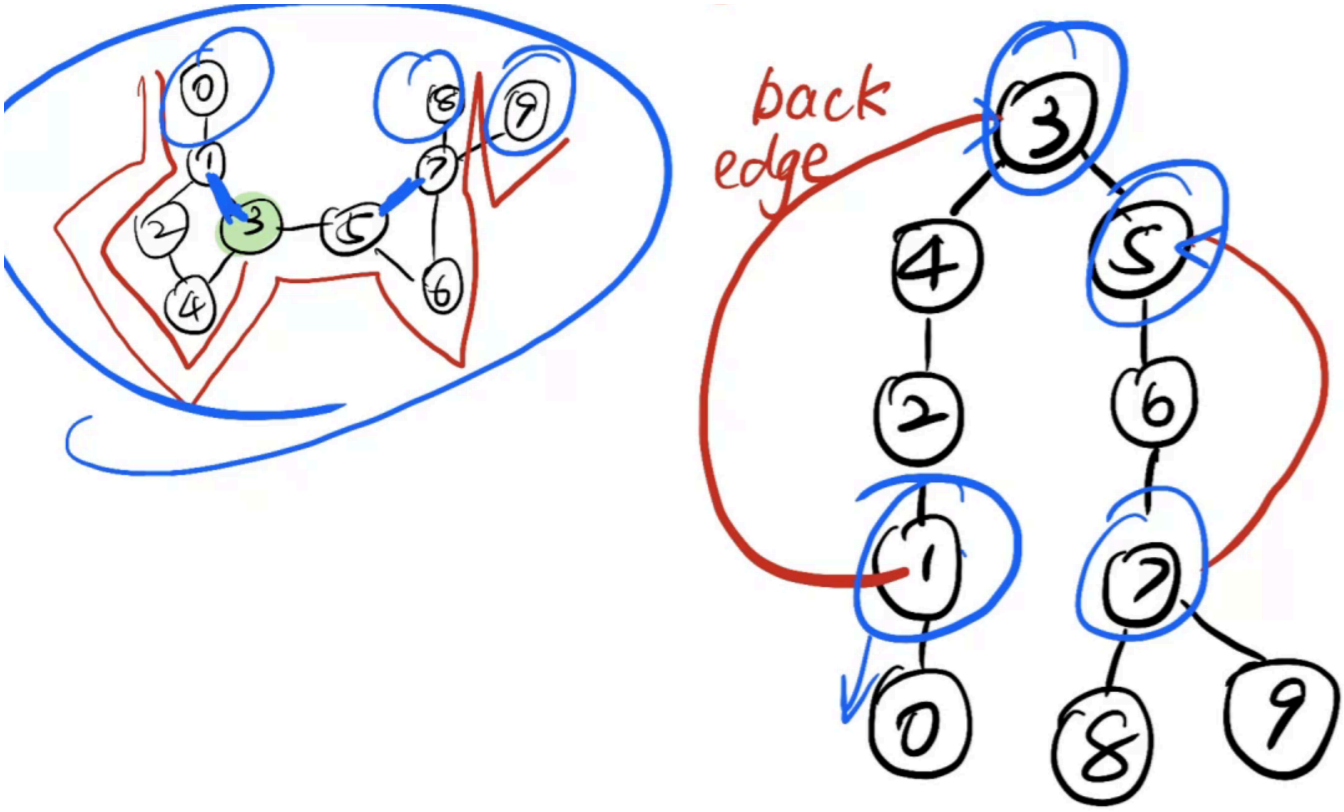
把所有边按照边长**排序**后，依次放在queue中。把边dequeue之后，判断当前的两个点是否已经在生成树中，如果该点已经在最小生成树中了(会形成环)，则该边要被舍弃。

```
void Kruskal(Graph g)
{
    T={};
    while(T contains less than |V|-1edges &&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);
    }
}
```

# DFS

DFS生成的树可以用来判断Biconnectivity。

connectivity连通分量，如果删除一个节点形成多个连通分量，则该点叫做articulation point。这些点可以用dfs找到。




联通节点(articulation point)的判定：如果往下走无法回到自己的father节点，则该节点就是articulation point

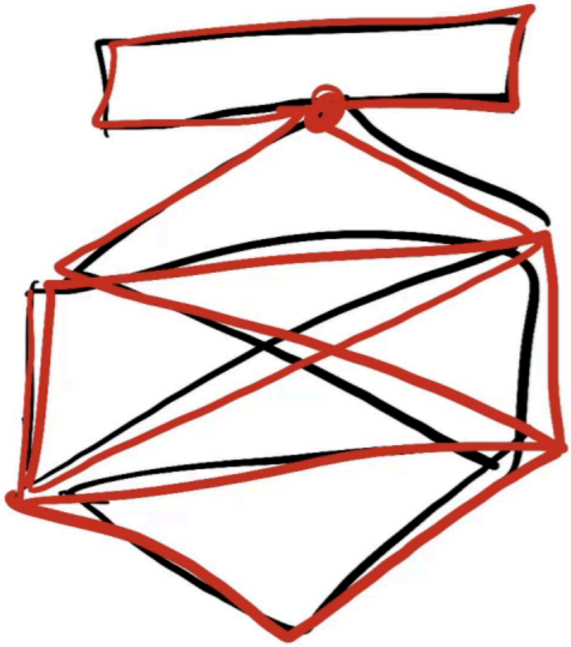
算法思想：递归

对于深度优先搜索生成树上的每一个顶点，计算编号最低的顶点，称之为Low[u]

*Low[u]=min{Num(u),Min{Low(w)|w is a child of u},min{Num(w)|(u,m)is a back edge}}*

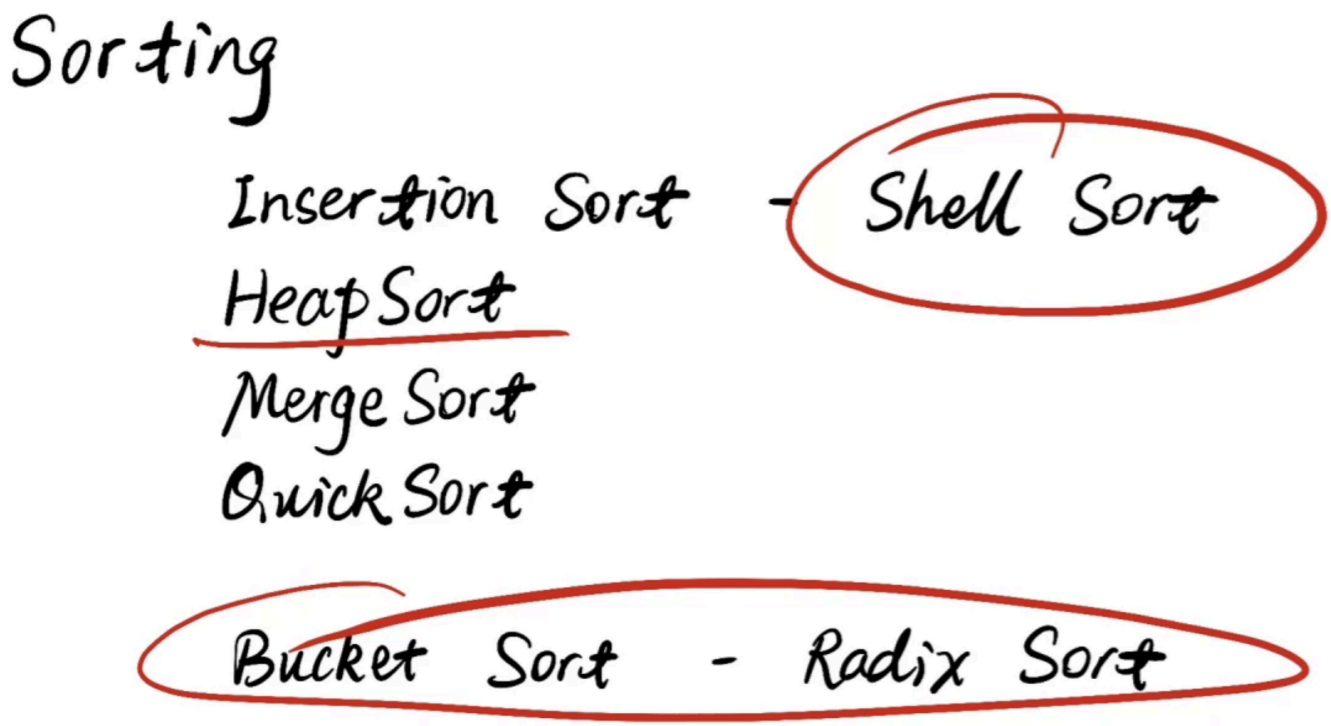
```
void FindArt(Vertex v)
{
    Vertex W;
    Visited[V]=True;
    Low[V]=Num[V]=Counter++;
    for(each W adgecent to V)
    {
        if(!Visited[W])
        {
            Parent[W]=V;
            FindArt(W);
            if(Low[W]>=Num[V])
              printf("%v is an articulation point\n",v);
            Low[V]=Min(Low[V],Low[W]);
        }
        else if(Parent[V]!=W)
            Low[V]=Min(Low[V],Num[W]);
    }
}
```

## Euler Circuit 欧拉回路



如果欧拉回路第一遍走完还有一部分没走到，但是剩余部分也满足欧拉回路，则可以并入原来的图中。（例如上述的矩形。欧拉回路必须满足每一个点都是**偶数条**的degree。

## Sorting




- 选择、快排、希尔、堆是不稳定的
- 冒泡、插入、归并、基数是稳定的

## Insertion Sort

```
void Insertion(ElementType A[],int N)
{
    int j,p;
    ElementType Tmp;
    for(p=1;p<N;p++)
    {
        Tmp=A[p];//先记录下当前这个节点的大小
        for(j=p;j>0&&A[j-1]>Tmp;j--)//遍历找到应该放他的位置，然后每一个数字向后平移
            A[j]=A[j-1];
        A[j]=Tmp;
    }
}
```

worst case: $T = O(N^2)$，此种情况下刚好是逆序的。

best case: $T = O(N)$

## Shell Sort

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

Shell's increment sequence

$$h_t = 「N/2」, \ h_k = 「h_{k+1}/2」 \text{两个都是取下整}$$

```
void ShellSort(ElementType A[],int N)
{
    int i,j,increment;
    ElementType Tmp;
    for(increment=N/2;increment>0;increment/=2)
        for(i=increment;i<N;i++)
        {
            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;
        }
}
```

worst case: $T(N)=O(N^2)$

**Hibbard's increment sequence:** $h_k=2^k-1$

In this way, $T(N)=\Theta(N^{\frac{3}{2}})$

## Heap Sort

两种算法：

第一种算法是需要用一个额外的数组，将每次DeleteMin的数存在TmpArray中，则空间复杂度上升。

第二种算法是每次DeleteMin时，交换A[0]&&A[i]的位置。

```
void HeapSort(ElementType A[], int N)
{
    int i;
    for(i=N/2;i>=0;i--)
        PercDown(A,i,N);
    for(i=N-1;i>0;i--)
    {
        Swap(&A[0],&A[i]);//这里注意是位置交换！！可能因为i考虑到ElementType这个变量是指针吧
        PercDown(A,0,i);
    }
}
```

Average comparisions: $2NlogN-O(NloglogN)$

## Merge Sort

```
void MSort(ElementType A[], ElementType TmpArray[],int left,int right)
{
    int center;
    if(left<right)
    {
       center=(left+right)/2;
       MSort(A,TmpArray,left,center);
       MSort(A,TmpArray,center+1,right);
       Merge(A,TmpArray,left,center+1,right)
    }
}
void Mergesort(ElementType A[],int N)
```

```
{
    ElementType *TmpArray;
    TmpArray=malloc(N*sizeof(struct ElementType));
    if(TmpArray!=NULL)
    {
        MSort(A,TmpArray,0,N-1);
        free(TmpArray);
    }
    else printf("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)
    {
        if(A[Lpos]<=A[Rpos])
          TmpArray[Tmppos++]=A[Lpos++];
        else
          TmpArray[Tmppos++]=A[Rpos++];
        while(Lpos<=leftend)
          TmpArray[Tmppos++]=A[Lpos++];
        while(Rpos<=rightend)
          TmpArray[Tmppos++]=A[Rpos++];
        for(i=0;i<NumElements;i++,rightend--)
          A[rightend]=TmpArray[rightend];
    }
}
```

## Quick Sort

最坏情况的时间复杂度 $O(N^2)$，平均情况是$O(NlogN)$

优化方法：

1.快排和其他算法融合；小数据用insertion sort，大数据用快排

2.随机选一个数开始

```
void QuickSort(ElementType A[], int N)
{
    Qsort(A,0,N-1);
  /*A:the array*/
  /*0:Left index*/
  /*N-1:Right index*/
}
```

```
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]);
  //上面三个操作保证了center位置上的数字是median的
    swap(&A[center],&[right-1]);//因为right位置上的数肯定比pivot要大
    return A[right-1];
}
```

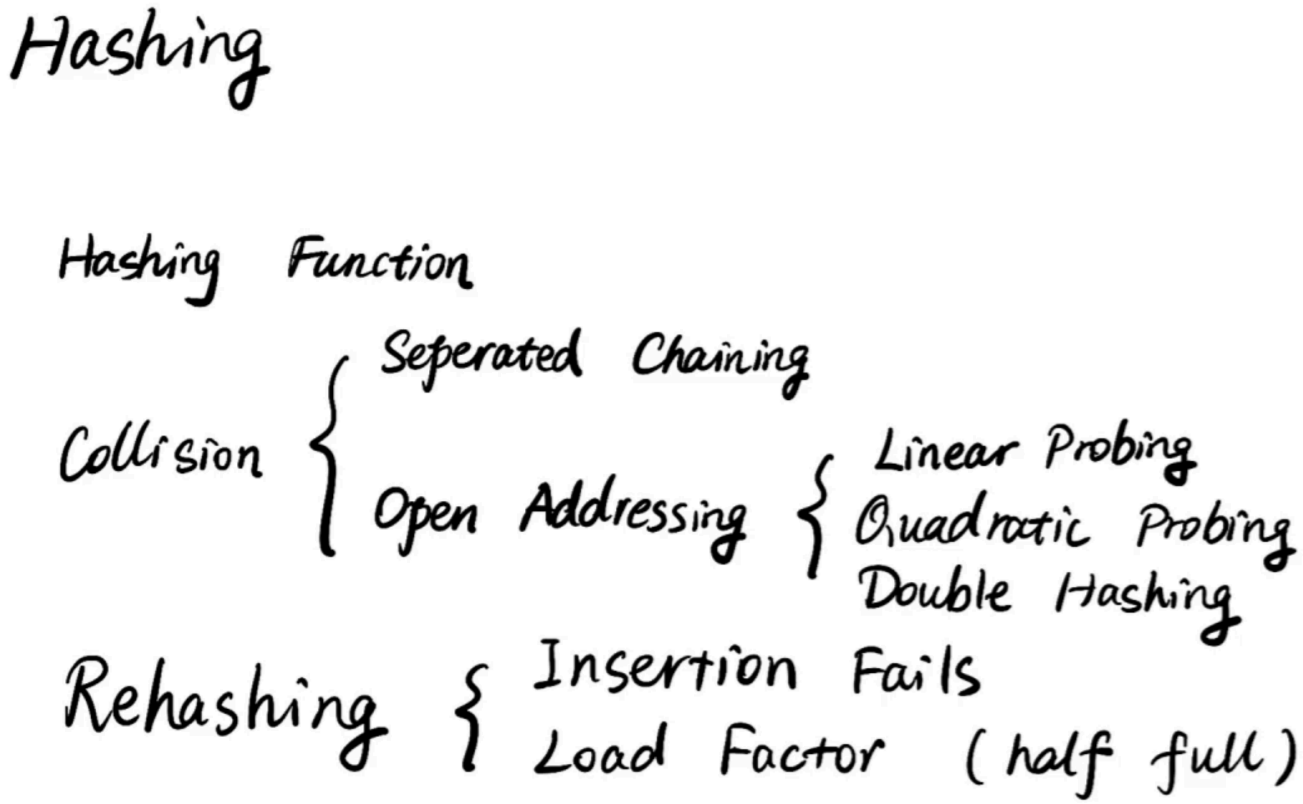
```
void QSort(ElementType A[],int left, int right)
{
      int i,jl
      ElementType Pivot;
      if(left+cutoff<=right)//数据规模较大时用快排
      {
          pivot=median3(A,left,right);
          i=left;
          j=right-1;
          for(;;)
          {
              while(A[++i]>pivot){}
              while(A[--j]<pivot){}
              if(i<j)
                swap(&A[i],&A[j]);
              else break;//i==j时候退出循环
          }
          Swap(&A[i],&A[right-1]);//pivot放到ij重合的位置上
          QSort(A,left,i-1);
          QSort(A,i+1,right);
      }
      else//数据规模较小时用插入算法
        InsertionSort(A+left,right-left+1);
}
```

## Bucket Sort

不基于比较的排序 案例：扑克牌按照❤️♦️♠️♣️排列

## Radix Sort

不基于比较的排序 案例：扑克牌按照1234.....排列




Hashing

## 几种数据结构的定义

## pta错题

**HW1**

1-2The Fibonacci number sequence {*FN*} is defined as: *F*0=0, *F*1=1, *FN*=*FN*−1+*FN*−2, *N*=2, 3, .... The time complexity of the function which calculates *FN* recursively is Θ(*N*!). **F**

斐波那且数列. if(N!=1) Fib(N)=Fib(N-1)+Fib(N-2)

T(N)=T(N-1)+T(N-2)+2 所以时间复杂度满足 $O(\frac{3}{2})^n \leq T(N) \leq O(\frac{5}{3})^n$

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

2-3The recurrent equations for the time complexities of programs P1 and P2 are:

- P1: $T(1) = 1, T(N) = T(N/3) + 1$
- P2: $T(1) = 1, T(N) = 3 \cdot T(N/3) + 1$

Then the correct conclusion about their time complexities is: ***O*(log*N*) for P1, *O*(*N*) for P2**

2-2Let *n* be a non-negative integer representing the size of input. The time complexity of the following piece of code is:

```
x = 0;
while ( n >= (x+1)*(x+1) )
    x = x+1;
```

$O(n^{\frac{1}{2}})$

**HW2**

1-1For a sequentially stored linear list of length *N*, the time complexities for deleting the first element and inserting the last element are $O(1)$ and $O(N)$, respectively. **F**

sequentially sorted linear list 相当于数组，因为是连续的排列

插入是O(1),查找是O(N)

2-2If 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?

A. doubly linked list

B. singly linked circular list

C. doubly linked circular list with a dummy head node

D. sequential list✅

**HW3**

2-1

Push 5 characters `ooops` onto a stack. In how many different ways that we can pop these characters and still obtain `ooops`?

A. 1 B. 3

C. 5 ✅ D. 6

ooo字符的push和pop顺序不固定，可以随意；等ooo全部输出完成后，再依次push和pop ‘p’和‘s’。

Push/pop/push/pop/push/pop

Push/pop/push/push/pop/pop

Push/push/pop/push/pop/pop

Push/push/pop/pop/push/pop

Push/push/push/pop/pop/pop

2-3Suppose that an array of size 6 is used to store a circular queue, and the values of `front` and `rear` are 0 and 4, respectively. Now after 2 dequeues and 2 enqueues, what will the values of `front` and `rear` be?

A. 2 and 0 ✅ `front==rear`说明queue为空，`(rear-front+1+maxsize)%maxsize==0`说明queue满了

B. 2 and 2

C. 2 and 4

D. 2 and 6

2-4 Suppose that all the integer operands are stored in the stack *S*1, and all the operators in the other stack *S*2. The function `F()` does the following operations sequentially:

- (1) Pop two operands `a` and `b` from *S*1;
- (2) Pop one operator `op` from *S*2;
- (3) Calculate `b op a`; and
- (4) Push the result back to *S*1.

Now given { 5, 8, 3, 2 } in *S*1 (where 2 is at the top), and { `*`, `-`, `+` } in *S*2 (where `+` is at the top). What is remained at the top of *S*1 after `F()` is executed 3 times?

A. -15 B. 15 ✅ 看清楚栈的顶端是谁/ `b op a`作减号的时候，应该是8-5=3，而不是-3

C. -20 D. 20

[补充] Suppose that an array of size m is used to store a circular queue. If the head pointer front and the current size variable size are used to represent the range of the queue instead of front and rear, then the maximum capacity of this queue can be:

A.cannot be determined
B.m+1
C.m ✅
D.m-1
当空队列的时候，rear指向第一个前一个，即最后一个，front指向第一个，当满队列的时候，发现rear和front也是这样指的，因此环队列应该要少存一个，防止搞不清。但是当用元素个数的时候，发现不会有这种情况，故是m个

**HW4**

1-1It is always possible to represent a tree by a one-dimensional integer array. **T**每个数组记录当前node的parent node

1-2There exists a binary tree with 2016 nodes in total, and with 16 nodes having only one child. **F**

2-1Given a tree of degree 3. Suppose that there are 3 nodes of degree 2 and 2 nodes of degree 3. Then the number of leaf nodes must be .错误答案7

答案：**8** tree中有一个等式：结点总数=边数+1

2-2If 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*?

A. Pre-order traversal

B. In-order traversal ✅

C. Post-order traversal

D. Level-order traversal

**In-order traversal**

**HW5**

1-1In a binary search tree, the keys on the same level from left to right must be in sorted (non-decreasing) order. **F**

因为左子树的数不可能大于右子树，所以the same level的数也有大小区别

1-2In a binary search tree which contains several integer keys including 4, 5, and 6, if 4 and 6 are on the same level, then 5 must be their parent. **F**

2-3Given a binary search tree with its preorder traversal sequence { 8, 2, 15, 10, 12, 21 }. If 8 is deleted from the tree, which one of the following statements is FALSE?

A. One possible preprder traversal sequence of the resulting tree may be { 2, 15, 10, 12, 21 } 左子树最大值-2移到要删除的节点

B. One possible preprder traversal sequence of the resulting tree may be { 10, 2, 15, 12, 21 }右子树最小值10移到要删除的节点

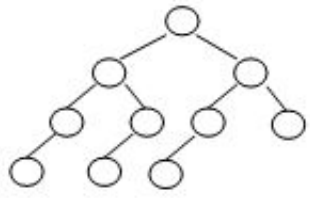
C. One possible preprder traversal sequence of the resulting tree may be { 15, 10, 2, 12, 21 } ✅

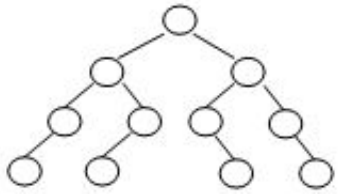
D. It is possible that the new root may have 2 children

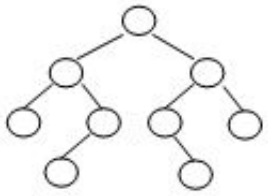
**BST删除节点的3种情况**：

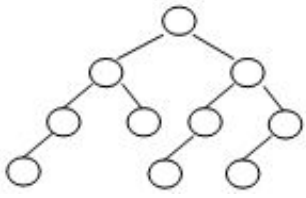
1. leaf node，则可以直接被删除
2. 有1个son，则可以直接将son以及其子树连到原来的节点上
3. 有2个son，一般的删除策略是用其右子树关键字最小的元素或左子树关键字最大的元素代替该节点的数据并递归的删除那个节点。

2-5Among the following binary trees, which one can possibly be the decision tree (the external nodes are excluded) for binary search? **A** decision树的特点：决策树的特征是非叶节点存放一个可能性，叶节点存放结果，因此叶子必须是唯一的孩子（不可能有两个结果）。并且最后一行的节点必须从左往右排列，不可以中间跳过。

A. 

B. 

C. 

D. 

**HW6**

1-2 The inorder traversal sequence of any min-heap must be in sorted order. **F**

2-1 In a max-heap with $n$ (>1) elements, the array index of the minimum key may be __.

A. 1　　B. $\lfloor n/2 \rfloor - 1$

C. ⌊*n*/2⌋　　D. ⌊*n*/2⌋+2 ✅ 这个堆是最大堆，找最小元素的可能位置

最小元素的位置 ⌊*n*/2⌋+1是最小值，且如果最小元素后面没有别的元素了，则其index越大

2-2 Using the linear algorithm to build a min-heap from the sequence {15, 26, 32, 8, 7, 20, 12, 13, 5, 19}, and then insert 6. Which one of the following statements is FALSE?

注意BST和min-heap的区别。min-heap建堆是先按照sequence随意插入，然后percolate down，并且没有左右子树大小关系的比较。

A. The root is 5

B. The path from the root to 26 is {5, 6, 8, 26}

C. 32 is the left child of 12 ✅ 32 is the right child of 12.

D. 7 is the parent of 19 and 15

2-4Insert {5, 2, 7, 3, 4, 1, 6} one by one into an initially empty min-heap. The preorder traversal sequence of the resulting tree is:

A. 1, 3, 2, 5, 4, 7, 6

B. 1, 2, 3, 4, 5, 7, 6

C. 1, 2, 5, 3, 4, 7, 6

D. 1, 3, 5, 4, 2, 7, 6 ✅

这个题是一个一个插进去，相当于一遍插一遍percolate。

一个一个边插入边渗透//全插完之后再渗透，两种建堆方式要区分开来

2-6If a binary search tree of N nodes is complete, which one of the following statements is FALSE?

**C** 最大数不一定，只要没有右节点就行
A.the average search time for all nodes is O(logN)

B.the minimum key must be at a leaf node

C.the maximum key must be at a leaf node

D.the median node must either be the root or in the left subtree

**HW7**

1-1In *Union/Find* algorithm, if *Unions* are done by size, the depth of any node must be no more than *N*/2, but not $O(logN)$. **F**

每做一次归并，都会使得小的集合深度加1，但是总的深度还是看大的集合。只有深度相同的归并才能使得总的深度加1。例如2,2归并，深度变为3；3,3归并，深度变为4。因此深度最大为 $log_2N+1$。所以深度就是$O(logN)$.

The array representation of a disjoint set containing numbers 0 to 8 is given by { 1, -4, 1, 1, -3, 4, 4, 8, -2 }. Then to union the two sets which contain 6 and 8 (with union-by-size), the index of the resulting root and the value stored at the root are: **4and-5**

程序填空题

5-1Please fill in the blanks in the program which performs `Find` as a Union/Find operation with path compression.（路径压缩）

```
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;
}
```

**HW8**

1-1If a directed graph G=(V,E) is weakly connected, then there must be at least |V| edges in G. **F**

*at least |V-1| edges*，所以不需要一定大于

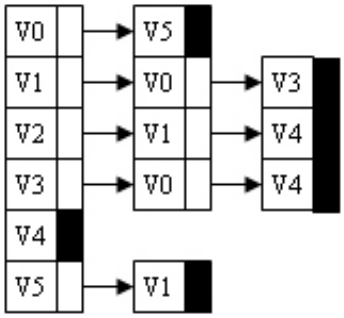
2-1If graph G is NOT connected and has 35 edges, then it must have at least  vertices. **10**

因为C(9,2)=36，即9个顶点的连通图最多有36条边，35条边也能符合。C(8,2)=28，即8个顶点的联通图最多有28条边，显然是达不到35条边的，因此如果是不连通的，必须加上一个独立的顶点，9 + 1 = 10

2-2 A graph with 90 vertices and 20 edges must have at least __ connected component(s)连通分量. **70**

根据欧拉公式，对于一个联通图 r=v-e+2，r是区域的数量。将一个图的一个联通分量看作一个联通图，假如有k个联通分量，求和可以得到，**R=V-E+2k**。但是注意到所有联通分量中最外部的那个区域被计算了k次，其实只需计算一次，因此需要减掉。R=V-E+2k-(k-1)=V-E+k+1
这就是任意图都能使用的欧拉公式。
代入可得：k=R+69，为了让联通分量最小，取R=1,即k=70

2-3 Given the adjacency list of a directed graph as shown by the figure. There is(are) __ strongly connected component(s).




A. 4 {{0, 1, 5}, {2}, {3}, {4}}

B. 3 {{2}, {4}, {0, 1, 3, 5}} ✅强连通概念：x->y, y->x的路径都可以找到，否则为一个连通分量

C. 1 {0, 1, 2, 3, 4, 5}

D. 1 {0, 5, 1, 3}

2-4 Given an undirected graph G with 16 edges, where 3 vertices are of degree 4, 4 vertices are of degree 3, and all the other vertices are of degrees less than 3. Then G must have at least __ vertices. **11**

不妨设有n个vertex，则边数：$[3*4+4*3+2(n-7)]/2=16$ 解得 $n=11$.

**2019-2020fds期中卷**

If a binary search tree of *N* nodes is also a complete binary tree, then among the following, which one is FALSE?(5分) **B**

A.The smallest key must be at a leaf node.否则节点的left leaf就比它小

B.The largest key must be on the last level 也不用是最后一层，只要是leaf就行

C.The average search time is *O*(*logN*)

D.The median key must be at either the root or in the left subtree of the root

**期中考试错题**

判断题

1-1$(logN)^2 is O(N)$. **T**

1-6The Fibonacci number sequence {*F***N*} is defined as: *F*0=0, *F*1=1, *FN*=*FN*−1+*FN*−2, *N*=2, 3, .... The time complexity of the function which calculates *F***N* recursively is *O*(*FN*)。 **T**

1-7In a circular queue which is implemented by an array, the `front` value must always be no larger than the `rear` value. **F**

1-8In a binary search tree which contains several integer keys including 4, 5, and 6, if 4 and 6 are on the same level, then 5 must be their parent. **F** *There is no need for 4 and 6 to be at the same level.*

选择题

2-6What is the major difference among lists, stacks, and queues?

A.Lists are linear structures while stacks and queues are not.

**B.**Stacks and queues are lists with insertion/deletion constraints.

C.Lists use pointers, and stacks and queues use arrays.

D.Lists and queues can be implemented using circularly linked lists, but stacks cannot.❌

2-9Since the speed of a printer cannot match the speed of a computer, a buffer is designed to temperarily store the data from a computer so that later the printer can retrieve data in order. Then the proper structure of the buffer shall be a:

**A**.queue

B.tree

C.stack❌

D.graph

2-10Suppose that a polynomial is represented by a linked list storing its non-zero terms. Given two polynimials with $N1$and $N2$ non-zero terms, and the highest exponents being $M1$ and $M2$, respectively. Then the time complexity for **adding them up** is:

**A**.$O$(N1+N2)

B.$O(M1+M2)$

C.$O(N1\times N2)$❌

D.$O(M1\times M2)$

**程序填空题**

The function is to increase the value of the integer key at position `P` by a positive amount `D` in a max-heap `H`.

```
void IncreaseKey( int P, int D, PriorityQueue H )
{
   int i, key;
   key = H->Elements[P] + D;
   for ( i = ++D; H->Elements[i/2] < key; i/=2 )//i==++D blank
      H->Elements[i]=H->Elements[i/2];//blank
   H->Elements[i] = key;
}
```

**HW9**

Let P be the shortest path from S to T. If the weight of every edge in the graph is incremented by 2, P will still be the shortest path from S to T. **F**

带负边的图的解决途径->可以把所有的边都加上一个正值然后解决，但是也存在问题：不同路的edge数量是不一定，所以最短路可能变成不是最短的了。

If besides finding the shortest path from `S` to every other vertices, we also need to count the number of different shortest paths, we can modify the Dijkstra algorithm in the following way: add an array `count[]` so that `count[V]` records the number of different shortest paths from `S` to `V`. Then `count[V]` shall be initialized as:

**A**.`count[S]=1;` and `count[V]=0` for other V ✅ count[S]=1表示这个路是可行的

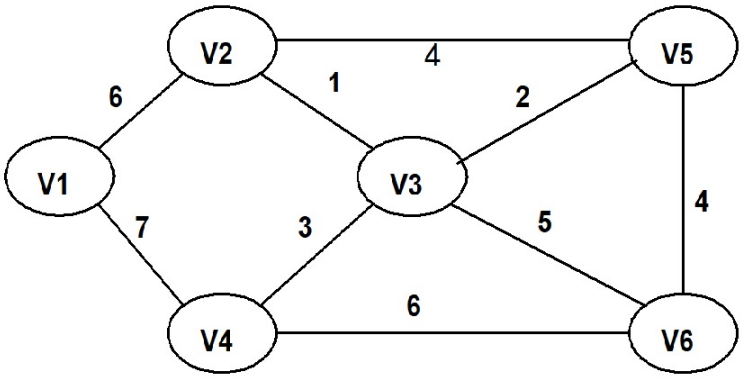
B.`count[S]=0` and `count[V]=1` for other V count[V]=0表示这两个点之间的距离

C.`count[V]=1` for all vertices

D.`count[V]=0` for all vertices

**HW10**

2-1To find the minimum spanning tree with Kruskal's algorithm for the following graph. Which edge will be added in the final step?




A. (v1,v4)　　B.(v1,v2)✅

C. (v4,v6)　　D. uncertain

Kruskal算法是将边按照长度排序，然后从最短边开始，标记该边相连的顶点。如果该边的2个节点都被标记，则跳过这条边；直到所有点都被标记visit。

2-2The minimum spanning tree of any weighted graph

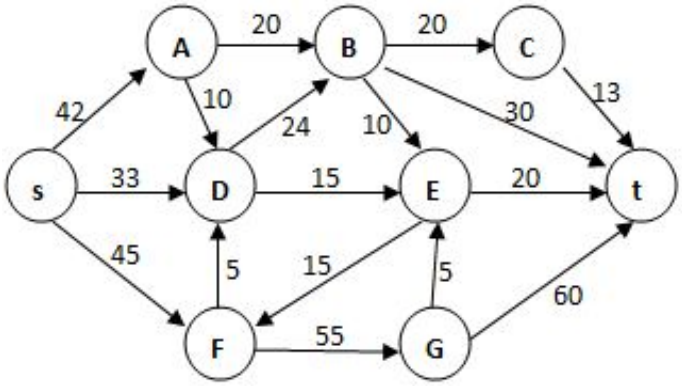
A. must be unique

B. must not be unique

C. exists but may not be unique

D.may not exist ✅可能有生成树，但是没有最小的（因为不唯一

2-3The maximum flow in the network of the given Figure is:



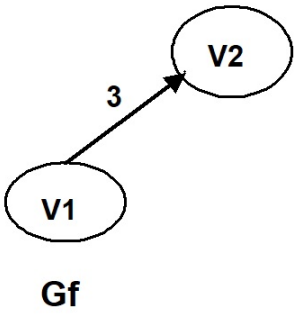

A. 104 ✅

B. 123

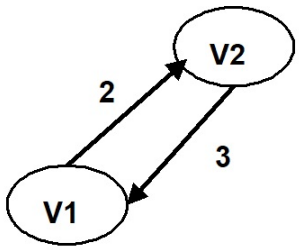
C. 120

D. 97

2-4 When solving the maximum flow problem for graph *G*, if partial states of the $G_f$ ( will be the maximum flow when the algorithm terminates) and $G_r$ (residual graph) are shown as the following, what must be the capacity of (v1, v2) or of (v2,v1) in the original graph *G*?




Gf




Gr(residula graph)

A. the capacity of (v1, v2) is 2

B. the capacity of (v1, v2) is 3

C. the capacity of (v1, v2) is 5 ✅ residual graph用反向边来标记当前的流量，而方向是v1->v2

D. the capacity of (v2, v1) is 5

**HW11**

For a graph, if each vertex has an even degree or only two vertexes have odd degree, we can find a cycle that visits every edge exactly once. **F** 欧拉回路需要是连通图，并且节点全是偶数节点，或者起点终点两个点为奇数点。

After the first run of Insertion Sort, it is possible that no element is placed in its final position. **T**

2-2Graph G is an undirected completed graph of 20 nodes. Is there an Euler circuit in G? If not, in order to have an Euler circuit, what is the minimum number of edges which should be removed from G?

A. Yes, Graph G has an Euler circuit

B. No, Graph G has no Euler circuit. 10 edges should be removed. ✅

C. No, Graph G has no Euler circuit. 20 edges should be removed.

D. No, Graph G has no Euler circuit. 40 edges should be removed.

20个点的完全图，每个点的度数为19。欧拉图要求每个点的度数都必须是偶数，因此为了使得每一个点的度数是18，减少的边为：20*1/2=10。

2-4 Use simple insertion sort to sort 10 numbers from non-decreasing to non-increasing, the possible numbers of comparisons and movements are:

A. 100, 100 B. 100, 54

C. 54, 63 D. 45, 44 ✅ 逆序对的个数不超过n(n-1)/2个，即交换次数不超过45

**HW12**

2-1To sort { 8, 3, 9, 11, 2, 1, 4, 7, 5, 10, 6 } by Shell Sort, if we obtain ( 4, 2, 1, 8, 3, 5, 10, 6, 9, 11, 7 ) after the first run, and ( 1, 2, 3, 5, 4, 6, 7, 8, 9, 11, 10 ) after the second run, then the increments of these two runs must be __ , respectively.

A. 3 and 1 B. 3 and 2 ✅

C. 5 and 2 D. 5 and 3

2-2To sort *N* elements by heap sort, the extra space complexity is:

A.*O*(1) ✅ B.*O*(logN)

C.*O*(*N*) D.*O*(*NlogN*)

2-3To sort *N* records by merge sort, the worst-case time complexity is:

A.O(logN) B.*O*(*N*)

C.*O*(NlogN)✅ D.$O(N^2)$

**HW13**

During the sorting, processing every element which is not yet at its final position is called a "run". To sort a list of integers using quick sort, it may reduce the total number of recursions by processing the small partion first in each run. **F** recursion的次数是不变的

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 ✅

Each run will find different number of pivots. 2 runs will settle down **3** pivots. 但是如果pivot在末尾的话，可以只有2个pivot。

When running internal sorting, if merge sort is chosen instead of insertion sort, the possible reason should be:

1. The code of merge sort is shorter
2. Merge sort takes less space
3. Merge sort runs faster

A. 2 only

**B.** 3 only ✅ Merge Sort 要有额外的array来存储结果

C. 1 and 2

D. 1 and 3

Among the following sorting methods, which ones will be slowed down if we store the elements in a linked structure instead of a sequential structure?

1. Insertion sort; 2. Selection Sort; 3. Bubble sort; 4. Shell sort; 5. Heap sort

A. 1 and 2 only

B. 2 and 3 only

C. 3 and 4 only

**D.** 4 and 5 only ✅

用链表存储元素，则会导致元素的插入变快O(1), 但是查找变慢O(N). 希尔排序需要去找第k个元素会变慢；heap需要去找father节点或者son节点访问数组，会变慢。

If quick sort is implemented recursively to sort an array of records, then which one of the following statements is TRUE?

A.After each partition, handling the longer sublist first will reduce the number of recursions.

B.After each partition, handling the shorter sublist first will reduce the number of recursions.

**C.**The number of recursions has nothing to do with the order of handling the sublists after each partition.✅

D.The number of recursions has nothing to do with the initial condition of the input.

递归和首先处理哪个子列的顺序无关

For the quicksort implementation with the left pointer stops at an element with the same key as the pivot during the partitioning, but the right pointer does not stop in a similar case, what is the running time when all keys are equal?

A.$O(logN)$

B.$O(N)$

C.$O(NlogN)$

**D.**$O(N^2)$

如果都相同的话：$T(N) = T(N-1) + N$，解出来是$O(N^2)$

Quick Sort can be implemented by a recursive function void Qsort( ElementType A[ ], int Left, int Right ). If we are to implement the function Qsort() in a non-recursive way with a stack, which of the following should be packed as the elements of the stack?

A.value of pivot

B.index of pivot

C.Left and Right ✅

D.only Left or Right

这道题的意思是说如果用快排递归次数太多，会导致栈溢出。因此我们可以自己维护一个栈。栈顶永远是left,下一个是right,然后按照left和right进行弹出。

Given input { 18, 92, 47, 51, 64, 9, 32 }. After the first partition (with the median-of-three as the pivot) of quick sort, the resulting sequence is D

A.{ 18, 9, 32, 51, 64, 92, 47 }

B.{ 18, 9, 47, 32, 51, 92, 64 }

C.{ 9, 18, 47, 51, 64, 92, 32 }

**D**.{ 18, 9, 32, 92, 64, 47, 51 } ✅

首先对1 8　5 1　3 2排序：

18, 92, 47, **32**, 64, 9, 51　pivot为32

18, 92, 47, 9, 64, **32**, 51　pivot和right-1位置上的数交换;

i从18开始，碰到比32大的数停下；j从64开始，碰到比32小的数停下➡️i停在92，j停在9

18, 9, 47, 92, 64, 32, 51

i从9开始，j从64开始➡️i停在47，j停在9 (产生交错了)➡️i和32交换

18, 9, 32, 92, 64, 47, 51

第一轮划分结束⬅️END

**HW14**

2-1The average search time of searching a hash table with *N* elements is:

A.$O(1)$

B.$O(logN)$

C.$O(N)$

D.cannot be determined ✅ 哈希表的搜索时间与collision的处理有关，所以不能确定

2-6Suppose that the range of a hash table is [0, 18], and the hash function is H(Key)=Key%17. If linear probing is used to resolve collisions, then after inserting { 16, 32, 14, 34, 48 } one by one into the hash table, the index of 48 is:

A. 14　　B. 0

C. 17✅　　D. 1 这道题不同的地方在于数组范围是[0,18]，但是模取值只到17，但是linear collision可以让48填到17下标这个位置。

|34| 1 2 3 4 5 6 7 8 9 10 11 12 13 |14| |32| |16| 17->48

**HW15**

2-1Given a hash table of size 13 and the hash function *h*(*x*)=*x*%13. Assume that **quadratic probing** is used to solve collisions. After filling in the hash table one by one with input sequence { 10, 23, 1, 36, 19, 5 }, which number is placed in the position of index 0?

A. 23　　B. 36

C. 10　　D. none✅

0 |1| 2 3 4 |5| |36| |19| 8 9 |10| |23| 12

2-2 Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function *h*(*X*)=*X*%10. If the collisions are solved by **separate chaining with table size being 10**, then the indices of the input numbers in the hash table are: (-1 means the insertion cannot be successful)

A. 1, 3, 3, 9, 4, 9, 9✅

B. 1, 3, 4, 9, 7, 5, -1

C. 1, 3, 4, 9, 5, 0, 8

D. 1, 3, 4, 9, 5, 0, 2

2-3Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function *h*(*X*)=*X*%10. If the collisions are solved by open addressing hash table with **second hash function** *h*2(*X*)=7−(*X*%7) and table size being 10, then the indices of the input numbers in the hash table are: (-1 means the insertion cannot be successful)

A.1, 3, 3, 9, 4, 9, 9

B.1, 3, 4, 9, 7, 5, -1✅

C.1, 3, 4, 9, 5, 0, 8

D.1, 3, 4, 9, 5, 0, 2

0 |4371| 2 |1323| |6173| |9879| 6 |4344| 8 |4199|

6173发生collision,则h2=7-6=1　所以f(i)=1*1=1,偏移量为1，到4号下标的位置

4344发生collision,则h2=7-4=3　所以f(i)=1*3=3,偏移量为3，到7号下标的位置

9679发生collision,则h2=7-5=2　所以$f(i)=1*2=2$,但是此时1下标位置也有数字，则再偏移$4=2*2$位到5号下标位置。

1989发生collision,则h2=7-1=6　所以$f(i)=1*6=6$,偏移量为6，到5号下标的位置,被占了;再偏移$f(2)=2*6=12$到7号位；再偏移$f(3)=3*6=18$到5号位；.....

2-4 Suppose that the numbers {4371, 1323, 6173, 4199, 4344, 9679, 1989} are hashed into a table of size 10 with the hash function $h(X)$=$X\%10$, and hence have indices {1, 3, 4, 9, 5, 0, 2}. What are their indices after **rehashing** using $h(X)$=$X\%TableSize$ with linear probing?

A. 11, 3, 13, 19, 4, 0, 9

B. 1, 3, 4, 9, 5, 0, 2

C. 1, 12, 9, 13, 20, 19, 11✅

D. 1, 12, 17, 0, 13, 8, 14

rehashing即是double扩大原表长，使得loading density下降一半的过程。因为装载因子loading density和期望查找长度有反相关的关系，所以降低α这样可以增加散列查找的效率。

本题中再散列后的表长应该是20，但是要取大于20的第一个素数，所以表长取23较适宜。

4371%23=1　1323%23=12 6173%23=9　4199%23=13 4344%23=20 9679%23=19 1989%23=11

2-5 Rehashing is required when an insertion to a hash table fails. Which of the following is **NOT** necessary to rehashing?

A. change the collision resolution strategy ✅

B. use a new function to hash all the elements into the new table

C. build another table that is bigger than the original one

D. scan down the entire original hash table for non-deleted elements