Fundamentals of Data Structures

Chapter 1 Preparation

Grading Policies

> Lecture Grade (75): Homework(10) + Quiz(10) + Mid_Term Exam(15) + Final Exam(40)
>
> Laboratory Grade (25/30) = $[\sum Lab(i) * 0.25(or\ 0.30)]/3$

Quizzes

Random Quizzes

10 minutes and 10 points each

Problems will be chosen from HW

Chapter 2 Algorithm Analysis

The Property of Algorithm

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

(1) Input There are zero or more quantities that are externally supplied.

(2) Output At least one quantity is produced.

(3) Definiteness Each instruction is clear and unambiguous.

(4) Finiteness If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after finite number of steps.

(5) Effectiveness Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in(3); it also must be feasible.

Tips: 至少一个输出，可以没有输入

What to analyse

§ 1 What to Analyze

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

• Assumptions:

① instructions are executed sequentially

② each instruction is simple, and takes exactly one time unit

③ integer size is fixed and we have infinite memory

• Typically the following two functions are analyzed:

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

Asymptotic Notation

四种渐进表达:

【Definition】 $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \le c \cdot f(N)$ for all $N \ge n_0$.

【Definition】 $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \ge c \cdot g(N)$ for all $N \ge n_0$.

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

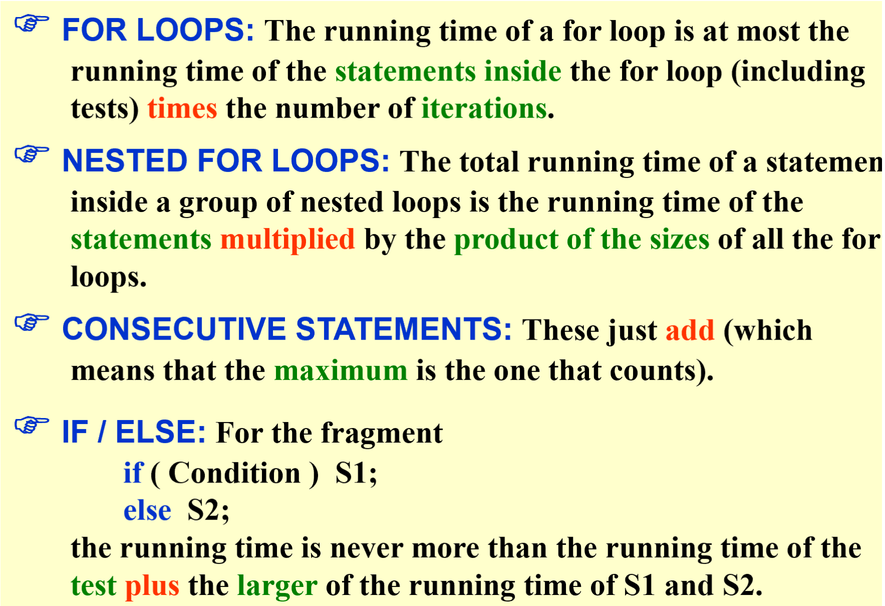
【Definition】 $T(N) = o(p(N))$ if $T(N) = O(p(N))$ and $T(N) \ne \Theta(p(N))$.

Rules Of Asymptotic

☞ If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
(a) $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$,
(b) $T_1(N) * T_2(N) = O(f(N) * g(N))$.

☞ If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(N^k)$.

☞ $\log^k N = O(N)$ for any constant k. This tells us that logarithms grow very slowly.



1、时间复杂度是指执行算法所需要的计算工作量。

2、空间复杂度是指执行这个算法所需要的内存空间。

对于Recursive Algotirithm来说：

递归算法的时间复杂度：递归的总次数*每次递归的数量。

递归算法的空间复杂度：递归的深度*每次递归创建变量的个数。

以Fibonacci数列为例，时间复杂度：(2^0 + 2^1 + ... + 2^n-2) * 2 = O(2 ^ n)

空间复杂度：n * 2 = O(n)

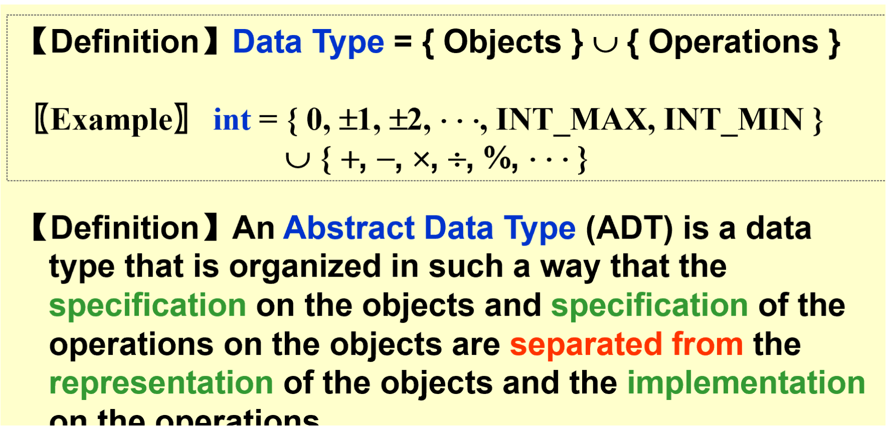
求解递推公式的复杂度

1、求解数列通项的方法

2、无限迭代消除n 例如log3n

Chapter 3 Lists, Stacks, and Queues

ADT![[Pasted image 20221030113848.png]]



LIST ADT

Objects: (item_0, item_1, ⋯ , item_{N-1})

Operations:

- **Finding the length**, N, of a list.
- **Printing** all the items in a list.
- **Making an empty** list.
- **Finding** the k-th item from a list, $0 \le k < N$.
- **Inserting** a new item after the k-th item of a list, $0 \le k < N$.
- **Deleting** an item from a list.
- **Finding next** of the current item from a list.
- **Finding previous** of the current item from a list.

数组:

array[i] = item_i

Sequential mapping

Address	Content
……	……
array+i	item_i
array+i+1	item_{i+1}
……	……

MaxSize has to be estimated.

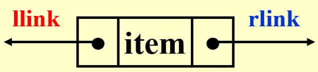
Find_Kth takes O(1) time.

Insertion and **Deletion** not only take O(N) time, but also involve a lot of data movements which takes time.

链表:

Doubly Linked Circular Lists:

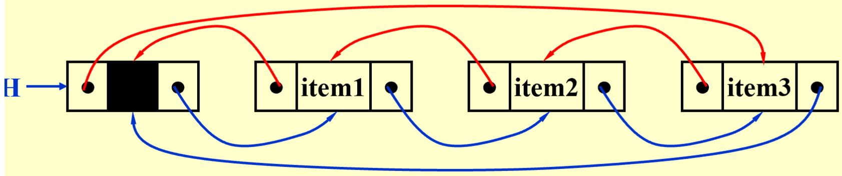
```
typedef struct node *node_ptr ;
typedef struct node {
      node_ptr llink;
      element  item;
      node_ptr rlink;
} ;
```



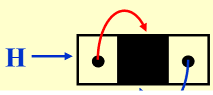
ptr = ptr->llink->rlink

= ptr->rlink->llink

A doubly linked circular list with head node:

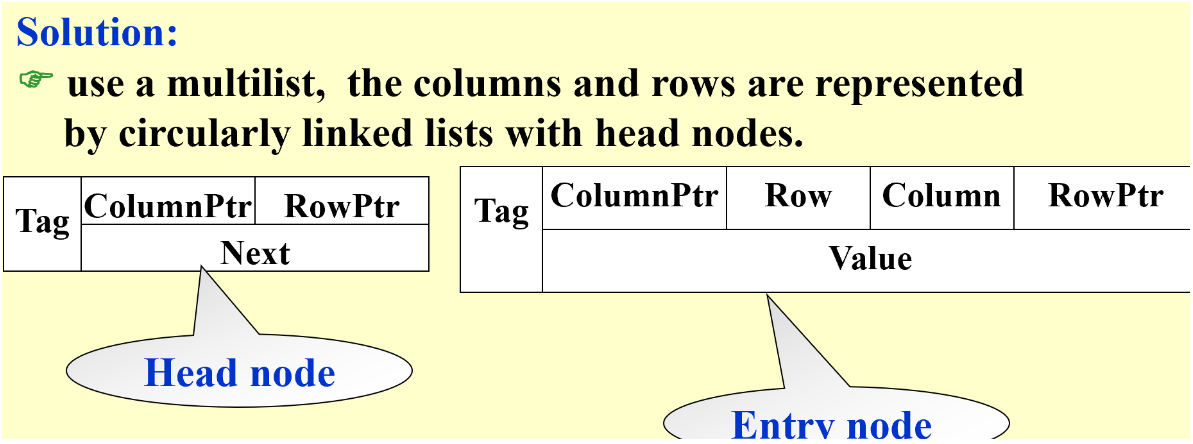


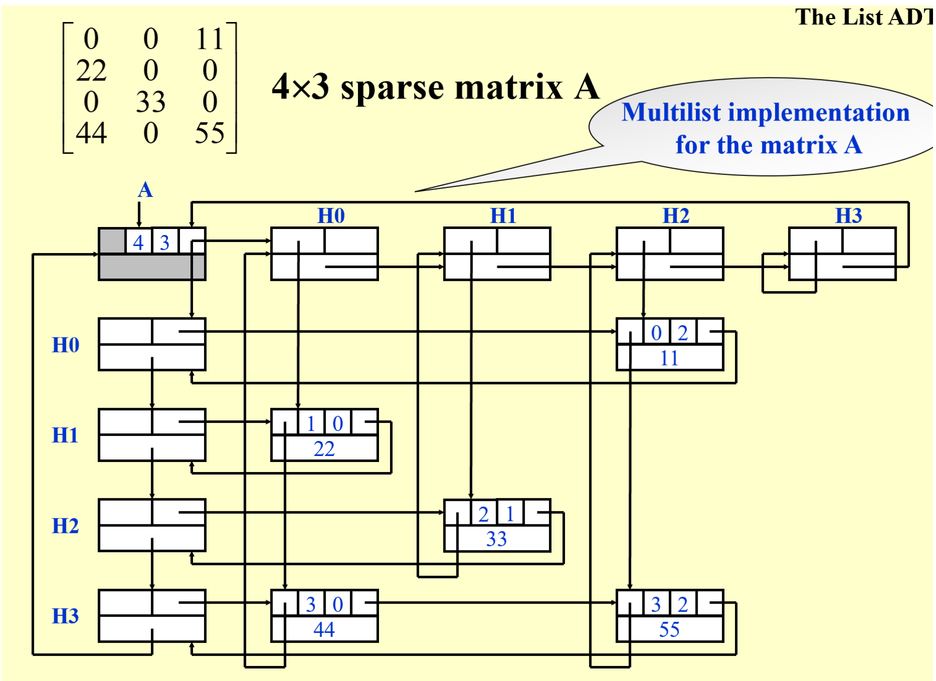
An empty list :



Multilists:

用多重链表表示sparse matrix（稀疏矩阵）





无指针实现：
![[Pasted image 20220926140505.png|500]]

STACK ADT

三种表达式:
![[Pasted image 20220926140728.png|500]]

中缀表达式$a+b*c+(d*e+f)*g$，其转换成后缀表达式则为$abc*+de*f+g*+$。
转换过程需要用到栈，具体过程如下：
1）如果遇到操作数，我们就直接将其输出。
2）如果遇到操作符，则我们将其放入到栈中，遇到左括号时我们也将其放入栈中。
3）如果遇到一个右括号，则将栈元素弹出，将弹出的操作符输出直到遇到左括号为止。（注意，左括号只弹出并不输出）
4）如果遇到任何其他的操作符，如（$+$，$*$，$($）等，从栈中弹出元素直到遇到发现更低优先级的元素(或者栈为空，或者遇到左括号)为止。弹出完这些元素后，才将遇到的操作符压入到栈中。有一点需要注意，只有在遇到") "的情况下我们才弹出"("，其他情况我们都不会弹出"("。
5）如果我们读到了输入的末尾，则将栈中所有元素依次弹出。

Chapter 4 Trees

4.1 一些基本概念

Preliminary:
节点的度(degree): 节点的子树个数
树的度(degree):树的所有节点中的最大的度数
Root:根节点
Subtree: 子树
Children:
Parent:
Siblings: parent相同的节点叫做siblings
Leaves: 没有child的节点叫做Leave
 Path: 从node n1 to nk的path是一段包含n1到nk的序列，其中前一个是后面一个节点的parent，path的length是这条path经过的edge的数目即k-1.
(ps: 有且仅有一条从root到各个节点的path)
Depth: ni节点的depth就是从根节点到该结点的path长度
Height: ni节点的height就是从ni到leaf的最长路径的长度。树的高度就是最深叶子节点的depth。
Ancestor | Descendent: 如果从n1到n2之间存在路径，那么n1是n2的ancestor，且n2是n1的descendent。如果n1 不等于 n2，那么变成proper ancestor|descendent

Implementation of trees
1、First child and next sibling实现:

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
    element_type element;
    tree_ptr first_child;
    tree_ptr next_sibling;
};
```

2、二叉树建立(先序遍历)

```
void CreatBiTree(BiTree &T)
{
    char ch;
    scanf("%c",&ch);
    if(ch == '#')
    {
        T = NULL;
    }
    else
    {
        T = (BiTree)malloc(sizeof(BiTNode));
        T->data = ch;
        CreatBiTree(T->lchild);
        CreatBiTree(T->rchild);
    }
}
```

3、求二叉树的深度

```
int TreeDeep(BiTree T)
{
    int deep = 0;
    if(T)
    {
        int leftdeep = TreeDeep(T->lchild);
        int rightdeep = TreeDeep(T->rchild);
        deep = leftdeep>=rightdeep?leftdeep+1:rightdeep+1;
    }
    return deep;
}
//孩子兄弟树
int depthCSTree(CSTree T)
{
    int maxd, d;
    CSTree p;
    if(!T) return 0; //空树
    else
    {
        for(maxd=0,p=T->firstchild; p; p=p->nextsibling)
        if((d=depthCSTree(p)) > maxd) maxd = d; //子树的最大深度
        return maxd + 1;
    }
}
```

4、孩子兄弟树遍历

```
void  InOrderTraverse_leaf(CSTree T)
{
    if(T)
    {
        if(!T->firstchild)
        {
            printf("%d ",T->data);
        }
        InOrderTraverse_leaf(T->firstchild);
        InOrderTraverse_leaf(T->nextsibling);
    }
}
```

4.2 Binary Trees

普通树转二叉树

1. 将树的根节点直接作为二叉树的根节点
2. 将树的根节点的第一个子节点作为根节点的左儿子，若该子节点存在兄弟节点，则将该子节点的第一个兄弟节点（方向从左往右）作为该子节点的右儿子
3. 将树中的剩余节点按照上一步的方式，依序添加到二叉树中，直到树中所有的节点都在二叉树中或者：
4. 在所有兄弟结点之间加一连线
5. 对每个结点，除了保留与其第一个儿子的连线外，去掉该结点与其它孩子的连线

   Traversal of Binary tree

   1、Preorder Traversal(前序遍历：根左右)

```
//递归版本
void preorder( tree_ptr  tree ){
    if ( tree ){
         visit ( tree );
         preorder(tree->left);
         preorder(tree->right);
  }
}
//迭代参考
vector<int> preorderTraversal(TreeNode* root)
{
    vector<int> v;//存储遍历结果的数组
    stack<TreeNode*> s; //栈，模拟搜索
    TreeNode* temp = root;
    //循环条件，只有当遍历完所有节点并且栈为空的时候才终止
    while(temp || !s.empty())
    {
        //当指向不为空节点时
        if(temp != nullptr)
        {
            v.emplace_back(temp->val);
            s.push(temp);  //将该结点入栈
            temp = temp->left; //根据前序遍历的要求，指向它的左子节点
        }else
        {
            //当左边已经完全搜完时，弹出栈顶节点并找到它的右子节点继续进行搜索
            temp = s.top()->right;
            s.pop();
        }
    }
    return v;
}
```

2、Postorder Traversal（后序遍历：左右根）

```
void postorder ( tree_ptr  tree ){
    if  ( tree )  {
        postorder(tree->left);
        postorder(tree->right);
        visit ( tree );
    }
}
```

3、Inorder Traversal（中序遍历：左根右）

```
void inorder ( tree_ptr  tree )
{  if  ( tree )   {
        inorder ( tree->Left );
        visit ( tree->Element );
        inorder ( tree->Right );
   }
}
//迭代版本
```

```
vector<int> inorderTraversal(TreeNode* root)
{
    vector<int>v; //存储结果语句
    stack<TreeNode*> s;
    TreeNode* temp = root;

    while(1)
    {
        //先找到最左边的节点，并将经过的节点入队
        while(temp)
        {
            s.push(temp);
            temp = temp->left;
        }
        //当栈为空时则退出循环
        if(s.empty()) break;
        //加入栈顶节点的值，即最左边的节点的值
        v.emplace_back(s.top()->val);
        //查找该节点的右子节点
        temp = s.top()->right;
        s.pop();
    }
    return v;
}
```

4、Levelorder Traversal（层序遍历）

```
void  levelorder ( tree_ptr  tree ){
    enqueue ( tree );
    while (queue is not empty) {
        visit ( T = dequeue ( ) );
        for (each child C of T )
            enqueue ( C );
    }
}
```

Threaded Binary Trees(线索二叉树)

---即把一个二叉树变成一个双向链表

Def: 利用二叉树中未利用的n+1个指针域，将中序遍历后得到的中序（左根右）二叉树保存下来，以便下次访问。

```
//声明
typedef  struct  ThreadedTreeNode  *PtrTo  ThreadedNode;
typedef  struct  PtrToThreadedNode  ThreadedTree;
typedef  struct  ThreadedTreeNode {
    int LeftThread;   /* if it is TRUE, then Left */
    ThreadedTree Left_child;      /* is a thread, not a child ptr.   */
    ElementType Element;
    int RightThread; /* if it is TRUE, then Right */
    ThreadedTree  Right_child;    /* is a thread, not a child ptr.   */
}
```

实现规则：
1、如果ptr->leftchild为空，那么存放中序遍历排列中该节点的前驱节点。该节点成为 ptr 的 中序前驱（inorder predecessor）。
2、 如果 ptr->rightChild 为空，则存放指向中序遍历序列中该结点的后继节点。这个节点称为 ptr 的中序后继（inorder successor）。
3、有一个头结点，头结点的lchild即left指向二叉树的根节点，rchild指向中序遍历访问的最后一个节点

如何实现线索化：

```
BThrNodePtr prev;/* 全局变量,始终指向刚刚访问过的结点 */
/* 中序遍历进行中序线索化 */
void InThreading(BThrNodePtr Tp)
{
    if (Tp)
    {
        InThreading(Tp->LChild);/* 在第一次左递归过程中绑定了如图的线条3 */
        if (!Tp->LChild)/* 没有左孩子 */
        {
            Tp->LTag = Thread;/* 前驱线索 */
            Tp->LChild = prev;/* 左孩子指针指向前驱 */
        }
        if (!prev->RChild)/* 前驱没有右孩子 */
        {
            prev->RTag = Thread;/* 后继线索 */
            prev->RChild = Tp;/* 前驱右孩子指针指向后继(当前结点Tp) */
        }

        prev = Tp;
        InThreading(Tp->RChild);/* 递归右子树线索化 */
    }
}
/* 中序遍历二叉树,并将其中序线索化,*Hpp指向头结点 */
bool InOrderThreading(BThrNodePtr *Hpp, BThrNodePtr Tp)
{
    cout << "InOrderThreading ..." << endl;
    *Hpp = (BThrNodePtr)malloc(sizeof(BThrNode));
    if (!(*Hpp))
        exit(1);
    (*Hpp)->LTag = Link;/* 建头结点 */
    (*Hpp)->RTag = Thread;
    (*Hpp)->RChild = (*Hpp);/* 右指针回指 */
    if (!Tp)
        (*Hpp)->LChild = *Hpp;/* 若二叉树空,则左指针回指 */
    else
    {
        (*Hpp)->LChild = Tp; /* 绑定如图的线1 */
        prev = (*Hpp); /* 头结点是第一个走过的点*/
        InThreading(Tp); /* 中序遍历进行中序线索化 */
        prev->RChild = *Hpp; /* 最后一个结点的后继指向头结点，即如图的线4*/
        prev->RTag = Thread;
        (*Hpp)->RChild = prev; /* 头结点的后继指向最后一个结点，即如图的线2*/
    }
}

/* 中序遍历二叉线索树(头结点)的非递归算法 */
```

```
//T指向头结点，头结点中lchild和rchild指向如上
//中序遍历二叉搜索链表表示的二叉树
Status InOrderTraverse_Thr(BiThrTree T){
    BiThrTree p;
    p = T->lchild; //p指向根节点
    while(p != T){ //空树或者遍历结束时，p == T
        while(p->LTag == Link) //循环到中序第一个结点
            p = p->lchild;
        printf("%c", p->data);
        whlie(p->RTag == Thread && p->rchild != T)
        {
            p = p->rchild;
            printf("%c", p->data);
        }
        p = p->rchild; //p前进至右子树根
    }
}
```

二叉树的性质

1. 二叉树中，第 i 层最多有2^{i-1} 个结点。
2. 如果二叉树的深度为 K，那么此二叉树最多有 2^K-1 个结点, $k >= 1$。
3. 二叉树中，终端结点数（叶子结点数）为 n0，度为 2 的结点数为 n2，则 $n0 = n2 + 1$。

满二叉树的性质
Def: 如果二叉树中除了叶子结点，每个结点的度都为 2，则此二叉树称为满二叉树。

1. 满二叉树中第 i 层的节点数为 2^{i-1} 个。
2. 深度为 k 的满二叉树必有 2^k-1 个节点 ，叶子数为 2^k-1。
3. 满二叉树中不存在度为 1 的节点，每一个分支点中都两棵深度相同的子树，且叶子节点都在最底层。
4. 具有 n 个节点的满二叉树的深度为 log2(n+1)。

4.3 Search Trees

- 在search tree中，孩子的顺序是重要的

Def:
![[Pasted image 20221017134409.png]]

标准ADT:

1. SearchTree MakeEmpty(SearchTree T);
2. Position Find(ElementType X, SearchTree T);
3. Position FindMin(SearchTree T);
4. Position FindMax(SearchTree T);
5. SearchTree Insert(ElementType X, SearchTree T);
6. SearchTree Delete(ElementType X, SearchTree T);
7. ElementType Retrieve(Position P);

4.4 AVL_Tree

最小不平衡子树：距离插入节点最近的，且平衡因子的绝对值大于1的 结点为根的子树。

5.1 Priority queue(HEAP)

Complete Binary Tree:

> 若设二叉树的深度为h，除第 h 层外，其它各层 (1 ~ h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树

完全二叉树的性质:
Def：如果二叉树中除去最后一层节点为满二叉树，且最后一层的结点依次从左到右分布，则此二叉树被称为完全二叉树。

1. n 个结点的完全二叉树的深度为 $\lfloor log2n \rfloor + 1$
2. 当 i>1 时，父亲结点为结点 [i/2] 。（i=1 时，表示的是根结点，无父亲结点)
3. 如果 2i>n（总结点的个数），则结点 i 肯定没有左孩子（为叶子结点）；否则其左孩子是结点 2i 。
4. 如果 2i+1>n，则结点 i 肯定没有右孩子；否则右孩子是结点 2i+1 。
5. 基本二叉树的性质

Heap:

> 堆也是一种完全二叉树。
> 最小堆：对于任意一个父结点来说，其子结点的值都大于这个父结点
> 最大堆：父节点的值比每一个子节点的值都要大

1. 插入操作(堆的shift up，以最大堆为例)：
   1、将数据添加到数组的最后一位
   2、依次与父结点进行比较并进行交换位置，逐渐上浮直到父结点大于该结点
2. 取出操作(堆的shift down)
   1、取出的元素是堆顶元素，即有最大优先级的元素
   2、如何填补：将数组的最后一个元素调整到堆顶，然后依次和子节点进行比较，并和较大的结点交换位置，直到所有的子节点都小于该结点
3. 构造堆的操作(堆排序)：
   （节点序号从1开始到n）从第一个非叶子节点（下标是[n/2]）开始，从它开始逐一向前对每一个元素作为根节点进行依次shift down操作
4. normal堆排序

- 将待排序的序列构造成一个最大堆，此时序列的最大值为根节点
- 依次将根节点与待排序序列的最后一个元素交换
- 再维护从根节点到该元素的前一个节点为最大堆，如此往复，最终得到一个递增序列

优先队列
其实就是一个最大堆.......

6.1 The Disjoint Set ADT（并查集）

简介：并查集是解决等价关系的一种数据结构，
并查集(Disjoint-Set或Union-Find Set)是一种表示不相交集合的数据结构，用于处理不相交集合的合并与查询问题。在不相交集合中，每个集合通过代表来区分，代表是集合中的某个成员，能够起到唯一标识该集合的作用。一般来说，选择哪一个元素作为代表是无关紧要的，关键是在进行查找操作时，得到的答案是一致的(通常把并查集数据结构构造成树形结构，根节点即为代表)

等价关系:
![[Pasted image 20221031133634.png]]

等价类
一个元素的等价类是a的一个子集， 包含所有与a有关系的元素。等价类也是对S的一个划分

并查集的操作
![[Pasted image 20221031134543.png]]
任意顺序的M次find和直到N-1次的Union最多花费O(M+NlogN)时间

数据结构实现方式
![[Pasted image 20221031135142.png|left|500]]
![[Pasted image 20221031135339.png|left|500]]
Union操作：

1. Union by-size（按大小求并）,把较小的集合合并到较大的集合中，最后形成树的深度不会超过log N

- 数组里面存储的是元素的个数，初始化为-1，每增加一个个数就-1

```
void unionset(int a[],int root1,int root2)
{
    int tmp=a[root1]+a[root2];
    if(a[root1]<=a[root2])
    {
        a[root2]=root1;
        a[root1]=tmp;
    }
    else
    {
        a[root1]=root2;
        a[root2]=tmp;
    }
}
```

2. Union by rank（按高度合并），把较小的集合合并到较大的集合中，只有两个集合的高度相同时，集合的高度才会更新

```
void unionset(int s[],int root1,int root2)
{
    if(s[root2]<s[root1])
        s[root1]=root2;
    else
    {
        if(s[root1]==s[root2])
            s[root1]--;
        s[root2]=root1;
    }
}
```

Find操作

```
int find(int a[],int x)
{
    if(a[x]<=-1)
        return x;
    else
        return find(a,a[x]);
}
```

Find操作的时间复杂度时O(logN)

Find过程实现路径压缩，压到一层：
![[Pasted image 20221104155501.png]]

路径压缩(和按大小求并适配，不和按rank适配)
从X到根路径上的每一个节点，如果不是根节点，就把该结点指向该子集的根节点

```
int find(int a[],int x)
{
if(a[x]<=-1)
    return x;
else
    return a[x] = find(a,a[x]);
}
```

![[Pasted image 20221103234205.png]]