数据结构

线性表、链表

- 顺序存储：物理相邻，逻辑相邻

  特点：插入删除慢，查询快

- 链接存储：物理不相邻，逻辑相邻

  特点：插入删除快，查询慢

  单链表的私有成员

```
private:
    struct node{...};
    node* head;
    int len;

    //this function is effective
    node *move(int i)const
    {
        node* p=head;
        while(i-->=0)p=p->nxt;
        return p;
    }
    //返回第i个节点的地址
```

  双链表

  私有成员比单链表多一个尾结点

  单循环链表

  不需要头结点，只需要指向线性表起始节点的指针head，如果head为空，表示空表。

- 链表的数组实现

```
    struct node{
        int val;//也可以用数组下标表示存储的值
        int l,int r;
    }a[max];
```

栈

- 实现

  线性栈、链接栈

- 应用

  递归消除、括号配对、算术表达式的运算

  中缀表达式与后缀表达式的相互转化：

  1. 中缀表达式 -->后缀表达式：

     > 将运算符放置在运算数之后

     如：5*(7-2*3)-8/2 --> 5.7.2.3.*-*8.2./-

     //下划线或者.来分割不同的数字

  2. 后缀表达式 -->中缀表达式：

     > 根据原理逆向推导

  3. 中缀表达式转化为后缀表达式的程序实现

     关键在于依据优先级和结合性重新排列运算符

     运算符优先级：

     右括号 > 乘方（右结合） > 乘除 > 加减（左结合） > 左括号

     ```
     enum token
     {OPAREN,ADD,SUB,MULTI,DIV,EXP,CPAREN,VALUE,EOL}

     按照上述方式定义可以解决优先级的问题，token可以直接比较大小
     ```

     核心思想： 依据栈的特性，当遇到优先级低的运算符，表明前面的运算符可以计算，输出；遇到优先级高的运算符，不能确定能否进行计算，先压入栈中保存。

     ```
     步骤：
     从左至右读取中缀表达式
     {
         if:
         {
             数字：直接输出
         运算符：
             if:
             比栈顶优先级高：意味着尚且不能计算
             比栈顶优先级底：意味着前面的运算符可以计算，弹出
             栈顶优先级同级：按照结合性决定是否弹出
                 if
                 左结合：说明前面的都能运算
                 右结合：不能运算
         左括号：
             直接入栈
         右括号：
     ```

```
            一直弹出直到左括号
        }
        入栈读取的运算符
    }
    按顺序依次出栈剩余运算符
```

队列

- 实现

  循环队列 共MaxSize 个单元的连续空间

```
//入队操作
rear=(rear+1)%MaxSize;
elem[rear]=x;

//出队操作
front=(front+1)%MaxSize;
```

  循环队列为了区分队列满和队列空的情况，通常采取牺牲一个单元的方法。(font/rear指向空单元)

- 应用

  火车车厢重排系统和单服务台排队系统模拟器

并查集

- 应用

  处理不相交的集合的合并和查询问题,凡是涉及分组的问题都可以尝试用并查集解决。

- 实现

初始化

```
int fa[MAXN]
void init(int n)
{
    for(int i=1;i<=n;++i)
        fa[i]=i;
}
```

查询（包含路径压缩）

判断任意两个元素是否属于同一个集合，只需要判断是否拥有同一个祖先即可。路径压缩以提高并查集效率

```
int find(int x)
{
    int tmp=x;
    while(fa[x]!=x)
    {
        x=fa[x];
    }

    //路径压缩
    fa[tmp]=x;

    return x;
}
```

递归版本

```
int find(int x)
{
    return x == fa[x] ? x : (fa[x] = find(fa[x]));
}
```

合并

```
void merge(int i,int j)
{
    fa[find(j)]=find(i);
}
```

扩展域

P2024 食物链 P1525 关押罪犯

用于增加并查集的更多分类关系

```
merge(x,y+n)
可以用于表示x 和 y的天敌是同一类。
```

树状结构

遍历

种类

前序遍历、中序遍历、后序遍历 （根节点的遍历位置决定）

层序遍历

构造

由前序/后序遍历+中序遍历可以得到完整二叉树

- 方法

  以前序遍历+中序遍历为例： 按照前序遍历获得根节点，由中序遍历查找根节点的位置，其左右两边则代表着左子树和右子树

顺序实现

lch: k<<1

rch: k<<1|1

parent:k>>1

前中后序遍历的非递归实现

1. 前序遍历

```
void preOrder()
{
    linkStack<Node*>s;
    Node* current;
    s.push(root)
    while(!s.empty())
    {
        current=s.pop();
        cout<< current->data;
        if(current->right)s.push(current->right);
        if(current->left)s.push(current->left);
    }
}
```

2. 中序遍历

```
struct StNode
{
    Node* node;
    int TimesPop;
    StNode(Node*N):node(N),TimesPop(0){}
}

void midOreder()
{
    linkList<StNode*>s;
    StNode current(root);
```

```
    s.push(current);
    while(!s.empty())
    {
        current=s.pop();
        if(++current->TimesPop==2)//根节点第二次出栈
        {
            //关键点 优先级高的节点后入栈
            cout<<current->data;
            if(current->right)
                s.push(StNode(current->right));
        }
        else
        {
            s.push(current);
            if(current->left)
                s.push(StNode(current->left));
        }
    }
}
```

3. 后序遍历

```
void postOrder()
{
    linkList<StNode*>s;
    StNode* current(root);
    s.push(current);

    while(!s.empty())
    {
        current=s.pop();
        if(++current->TimesPop==3)//第三次出栈
            cout<< current->data;
         s.push(current);
         if(current->Times==2)//第二次入栈
         {
            if(current->right)
                s.push(StNode(current->right));
         }
         else
         {
            if(current->left)
                s.push(StNode(current-.left));
         }
    }
}
```

应用

中缀表达式 -> 表达式树 -> 后序遍历计算结果

哈夫曼树

- 定义 构建一棵树，所有字符都包含在叶结点上，权值大的叶子应当尽量靠近树根，使它的编码尽可能地短。从根到存储该字符的叶节点的路径即为哈夫曼编码，可定义左支为0，右支为1。

> 只要每个字符的编码和其他字符的编码前缀不同，此编码成为前缀编码。

- 实现细节

  对于n个元素，需要2n大小的数组，[n,2n-1]存储叶子节点，1为根节点，归并生成的节点从n-1逐步递推到根节点。

树和森林

优先队列（堆）

- 实现原理

  1. enQueue(const T& x):插入元素x；放在末尾，逐渐和父节点递归交换至合适位置。
  2. deQueue():删除顶部元素；将顶部元素变成末尾元素，找子女中最小的（小根堆）节点，交换位置至合适位置。
  3. buildHeap():将一棵完全二叉树变成堆，复杂度O（n）;从叶结点逆向层次调用percolateDown(i),使得堆自底向上逐步满足条件。叶结点天生满足堆的有序性，因此只需要从编号最大的非叶子结点开始调用percolateDown(i)函数即可。 【i=currentSize/2】

- 代码实现

```
template<class T>
class priorityQueue
{
private:
    T* elem;
    int currentSize,maxSize;

    void percolateDown(int i)
    {
        T tmp=elem[i];
        int child,hole=i;
        for(;hole*2<=currentSize;hole=child)
        {
            child=hole*2;
            if(child!=currentSize&&elem[child+1]<elem[child])
                child++;
            if(elem[child]<tmp)elem[hole]=elem[child];
            else break;
        }

        elem[hole]=tmp;
    }

    void buildHeap()
    {
```

```
        for(int i=currentSize/2;i>0;--i)
            percolateDown(i);
    }

public:
    priorityQueue():currentSize(0),maxSize(1000000){elem=new T[maxSize];}
    priorityQueue(const T* items,int size):maxSize(size+10),currentSize(size)
    {
        elem=new T[maxSize];
        for(int i=1;i<=currentSize;++i)
            elem[i]=items[i-1];
        buildHeap();
    }

    ~priorityQueue(){delete[]elem;}

    void enQueue(const T&x)
    {
        int hole=++currentSize;
        for(;hole>1&&x<elem[hole>>1];hole>>=1)
            elem[hole]=elem[hole>>1];
        elem[hole]=x;
    }

    T& top()const{return elem[1];}

    T deQueue()
    {
        T minItem=elem[1];
        elem[1]=elem[currentSize--];
        percolateDown(1);
        return minItem;
    }
}
```

- 升级版堆
    1. D堆：每个结点存在D个儿子。主要被应用于插入操作比删除操作多很多的应用中，生成的堆较二叉堆更矮，插入操作会更快。
    2. 归并堆：左堆、斜堆、二项堆。归并两个堆的速率更快。

集合

查找

【内部查找】，一般以比较次数作为衡量时间性能的标准；外部查找一般以外存储器的访问次数作为衡量标准。

1. 二分查找
2. 插值查找
3. 分块查找

二叉查找树

【分类】
查找树：处理动态查找表的树
散列表：专门用于集合查找的数据结构

【二叉查找树】
def
1．若左子树不为空，则左子树[所有]元素比根节点键值小
2．若右子树不为空，则右子树[所有]元素比根节点键值大
3．它的左右子树同样是二叉查找树

重要性质
中序遍历一颗二叉查找树即得到按键值递增次序的排列

插入　递归过程
1．若根节点不存在，则插入为根节点
2．若根节点关键字大于插入元素关键字，则在左子树上插入
3．反之，在右子树上插入

删除
1．若删除关键字小于根节点，在左子树上删除结点
2．若删除关键字大于根节点，在右子树上删除结点
3．删除根节点。（1）若根结点是叶子，直接删除；（2）若根节点只有至多一个子节点，则将子结点直接连接在它的父亲节点；（3）若左右儿子都存在，则选取左子树的最大值或右子树的最小值作为根节点的替代，并删除该结点。

时间代价
最坏情况会退化为单链表，复杂度O（N），平均复杂度为O（logN）

AVL树

定义

一棵每个节点的左右子树高度差最多为1的二叉查找树。

特性

树中最少的结点树是其高度的指数函数，即高为H的树至少有C^H个结点

最坏情况下的时间复杂度是对数级的

插入操作

> AVL树的平衡策略保证了插入后至多只要调整一个节点的平衡!

```
工具函数 LL LR RL RR 函数

LL RR镜像对称，LR RL镜像对称;

LL函数：（单旋转）
当出现A的左子树比右子树高，插入发生在A的左儿子B的左子树中。（外侧）

操作：让B作为树根，让A作为B的右子树，B的右子树作为A的左子树
注意： t结点是引用传递
void LL(AvlNode*& t)
{
    AvlNode* t1=t->left;//未来的树根
    t->left=t1->right;
    t1->right=t;

    t->height=max(height(t->left),height(t->right))+1;
    t1->height=max(height(t1->left),height(t1->right))+1;

    t=t1;
}

LR函数：（双旋转）
当出现A的左子树比右子树高，插入发生在A的左儿子B的右子树中。（内侧）

操作：先对B做一次左旋转，再对A做一次右旋转

void LR(AvlNode*& t)
{
    RR(t->left);
    LL(t);
}
```

Avl树模板代码

```
template<class KEY,class OTHER>
struct SET
{
    KEY key;
    OTHER other;
};


template<class KEY,class OTHER>
class AvlTree {
 struct AvlNode
 {
```

```
        SET<KEY,OTHER> data;
        AvlNode *left;
        AvlNode* right;
        int height;

        AvlNode(const SET<KEY,OTHER>&elem,AvlNode* lt,AvlNode* rt,int h=1)
        :data(elem),left(lt),right(rt),height(h){}
    };

    AvlNode* root;

public:
    AvlTree(){root=NULL;}
    ~AvlTree(){clear(root);}
    SET<KEY,OTHER>*find(const KEY& x)const
    {
        AvlNode*t=root;

        while(t!=NULL&&t->data.key!=x)
        {
            t=(t->data.key<x)?t->right:t->left;
        }
        if(t==NULL)return NULL;
        else return t->data;
    };

    void insert(const SET<KEY,OTHER>&x)
    {
        insert(x,root);
    }

    void remove(const KEY& x)
    {
        remove(x,root);
    }

private:
    void insert(const SET<KEY,OTHER>&x,AvlNode*& t)
    {
        if(t==NULL)
            t=new AvlNode(x,NULL,NULL);
        else if(t->data.key>x.key)
        {
            insert(x,root->left);
            if(height(t->left)-height(t->right)==2)
            {
                if(x.key<t->left->data.key)LL(t);
                else LR(t);
            }
        }
        else if(t->data.key<x.key)
        {
            insert(x,root->right);
            if(height(t->left)-height(t->right)==2)
```

```
            {
                if(height(x.key<t->right->data.key))RL(t);
                else RR(t);
            }
        }

        t->height=max(height(t->left),height(t->right))+1;
    }

    bool remove(const KEY&x,AvlNode*&t)
    {
        if(t==NULL)return true;
        if(x==t->data.key)
        {
            if(t->left==NULL||t->right==NULL)
            {
                AvlNode* oldNode=t;
                t=(t->left)?t->left:t->right;
                delete oldNode;
                return false;
            }
            else
            {
                AvlNode* tmp=t->right;
                while(tmp->left)tmp=tmp->left;
                t->data=tmp->data;
                if(remove(tmp->data.key,t->right))return true;//删除后右子树没有变
矮
                return adjust(t,1);
            }
        }

        if(x<t->data.key)
        {
            if(remove(x,t->left))return true;
            return adjust(t,0);
        }
        else
        {
            if(remove(x,t->right))return true;
            return adjust(t,1);
        }
    }

    void clear(AvlNode*t)
    {
        if(t==NULL)return;
        clear(t->left);
        clear(t->right);
        delete t;
        t=NULL;
    }

    int height(AvlNode*t)const{return t==NULL?0:t->height;}
```

```
void LL(AvlNode*&t)
{
    AvlNode* t1=t->left;
    t->left=t1->right;
    t1->left=t;
    t->height=max(t->left->height,t->right->height)+1;
    t1->height=max(t1->left->height,t1->right->height)+1;
    t=t1;
}

void LR(AvlNode*&t)
{
    RR(t->left);
    LL(t);
}

void RL(AvlNode*&t)
{
    LL(t->right);
    RR(t);
}

void RR(AvlNode*&t)
{
    AvlNode* t1=t->right;
    t->right=t1->left;
    t1->left=t;
    t->height=max(t->left->height,t->right->height)+1;
    t1->height=max(t1->left->height,t1->right->height)+1;
    t=t1;
}
int max(int a,int b){return (a>b)?a:b;}
bool adjust(AvlNode*&t,int subTree)//调整树平衡的函数
{
    if(subTree)
    {
        if(height(t->left)-height(t->right)==1)return true;
        if(height(t->left)==height(t->right)){--t->height;return false;}
        if(height(t->left->right)>height(t->left->left))
        {
            LR(t);
            return false;
        }
        LL(t);
        if(height(t->right)==height(t->left))return false;
        else return true;
    }
    else
    {
        if(height(t->right)-height(t->left)==1)return true;
        if(height(t->right)==height(t->left)){--t->height;return false;}
        if(height(t->right->left)>height(t->right->right))
        {
```

```
                RL(t);
                return false;
            }
            RR(t);
            if(height(t->right)==height(t->left))return false;
            else return true;
        }
    }
};
```

私有的remove()函数返回bool类型，false表示需要检查目标节点的父结点的平衡度。 而adjust函数表示对所给结点的五种情况进行处理，如果需要修改父结点，返回false。

红黑树

【定义】

$红黑树本质是2-3-4树的BST实现,红结点理解为红链，红链代表同一个结点的两个键值$ 红黑树是一棵具有如下特点的二叉查找树。

（1）每个结点都被染成红色或黑色

（2）根节点是黑色的

（3）如果一个结点是红的，那么它的儿子结点必须是黑色的

（4）从任何一个结点出发到空结点（一般认为空结点为黑色结点）的路径上，必须包含相同数目的黑色结点。

性质3、4表明各条路径的长度差是有限的，最长的路径是最短路径的一倍，最长的路径是由红黑结点相间组成的。

【插入操作】

新结点总是作为叶结点插入，与二叉查找树一致，但是新插入的结点只能是红色（若新插入结点是黑色的，将违反性质4）。若插入结点的父结点是黑色，结束；若为红色，则需要通过修改结点的着色和旋转维持红黑树。其主要分为两种情况： 设红结点为X，父结点为P且也是红色，P的父结点为G，且根据红黑树定义G一定为黑色。

1. P的兄弟结点S是黑色的。
    1. X是G的外侧结点。LLb()&&RRb()函数，其中b表示P的兄弟结点是黑色的。对G执行一次单旋转，最后的旋转结果中令根结点为黑色，左右儿子结点为红色。
    2. X是G的内侧结点。LRb()&&RLb()函数。对G执行一次双旋转，并令根节点为黑色，左右儿子结点为红色。
2. P的兄弟结点S是红色的。此时G是黑色，P、S全为红色。将G变成红色，P、S变成黑色，此时问题解决了但没完全解决，因为G的父亲结点可能也是红色，则需要递归向上回溯。 为了插入过程不再向上回溯，采取自顶向下的处理过程，若遇到结点t为黑色，左右孩子非空且均为红色，将t结点变成红色，左右孩子变成黑色。通过上述操作保证插入时只存在父亲节点为黑色，或者兄弟结点S为黑色两种情况，不会存在回溯。

综上所述，插入的逻辑过程如下：

```
if 根节点空：创建新树//特判条件

while true:
    if 结点非空:
        if:左右孩子非空且为红色:
            黑色下沉
            检查一次t，t->paret,t->parent->parent结点是否存在情况1。
        else:
            t结点向下一层
    else:
        插入结点
        insertAdjust(granP,parent,t)
        root->color=BLACK//为满足性质2
```

删除操作

图论

查找所有强连通分量

实现

1. 从图G任意节点开始执行dfs,并按照遍历的顺序给每个结点编号。
2. 将图G的每条边逆向，从编号最大的结点开始dfs
3. 由此得到的每棵树就是G的强连通分量

拓扑排序（AOV网）

定义

将有向无环图按照如下规则排序：如果有一条从u到v的路径，那么结点v在拓扑排序中必须出现在结点u之后。事件定义在结点上（Acativity on Vers）

实现关键

类似BFS实现，不同的是，一个节点只有当所有可以到达该节点的其他结点都被访问后才能访问。 关键是使用数组inDegree保存每个结点的入度，并将入度为0的结点压入队列中。

关键路径（AOE网）

定义

1. AOE（Acativity on Edges）关键事件定义在边上，可用于估算工程的完成时间。主要解决完成工程至少需要多少时间以及哪些活动是影响的关键。
2. 最早发生时间：从源点到某个结点的最长路径称为最早发生时间。
3. 最迟发生时间：不推迟整个工程完成的前提下，从顶点出发的最迟开始时间。
4. 时间余量：最迟和最早发生时间之差。
5. 关键路径：从源点到收点的最长路径。该上的活动称之为关键活动

特性

1. 完成工程至少需要的时间对应的是从源点到收点的最长路径的长度。
2. 时间余量之差为0则为关键活动
3. 每个顶点的最早发生时间=max(每个直接前驱最早发生时间+该前驱到该点活动时间)
4. 最迟发生时间=min(每个直接后继最迟发生时间-该点到该后继活动时间)

找出关键路径实现

1. 找出拓扑序列
2. 从头到尾遍历拓扑序列找出最早发生时间
3. 从尾到头遍历拓扑序列找最迟发生时间
4. 从头到尾遍历拓扑序列，找最早发生时间和最迟发生时间相等的顶点，组成关键路径。

最小生成树

定义

1. 最小生成树：加权无向连通图众多生成树中权值和最小的生成树。
2. 生成最小生成树的常用算法为：Prim算法和Kruskal算法

特性

1. n个结点的无向连通图最大存在$$\frac{n*(n-1)}{2}$$条边，最小生成树即从中挑出n-1条边使得权值最小并连通所有结点。
2. Kruskal算法适用于稀疏图；Prim算法适用于稠密图。

Kruskal算法

基本思想

从边的角度出发。依次选择图中权值最小的边，若加入这条边不会产生回路，则加入；否则考虑下一条边。直至选择出n-1条边

实现

1. 选择权值最小的边采取优先队列
2. 判读是否产生回路使用并查集确定两个结点是否属于同一棵树

Prim算法

基本思想

从顶点的角度出发，将一个个结点加入生成树，直到所有结点被加入

实现

1. 初始时，生成树节点集U为空

2. 随意选择一个结点，加入结点集。重复下列工作至U=V (1)选择两个端点分别位于U和V-U的代价最小的边(u,v) (2)把(u,v)加入生成树边集，v加入U。

最短路径问题

定义

1. 单源最短路径：某一个结点到所有结点的最短路径。常用Dijkstra算法
2. 结点对的最短路径：求每一对结点之间的最短路径。常用Floyd算法

非加权单源最短路径

1. 算法： 通过BFS算法直到所有结点都找到最短路径即可结束 需要distance数组和prev数组，prev数组记录到达该节点的上一个结点。 输出路径使用递归函数输出prev数组
2. 伪代码

```
unweightedShortDistance(start)
{
    for->distance(v)=无穷大;
    distance[start]=0;
    start->queue;
    while(!queue.empty())
    {
        u=queue.pop();
        for(u的所有邻接点v)
        {
            if(distance[v]==无穷大)
            {
                distance[v]=distance[u]+1;
                pre[v]=u;
                queue.push(v);
            }
        }
    }

}
```

Dijkstra算法

本质是BFS加数组不断更新最短路径

- 带有负权值的图 注：不能让每条边的权值都增加一个值直到所有的权值都变成正！ 实际解决办法：采取加权图和非加权图的方法结合。在遇到比之前路径更短的结果时，要继续检查后续结点，后续结点同样可能会存在更短的路径。而Dijstra算法遇到更短路径只需要更新即可。

ST表

它是解决RMQ问题(区间最值问题)的一种强有力的工具

它可以做到O(nlogn)预处理，O(1)查询最值

局限性：对于修改的区间无能为力

算法思想

1. 预处理 对于st表 `f[i][j]` 表示$$f[i][j]=Max[i, i+2^j-1]$$即从i开始，区间长度为2^j的最大值。 核心是dp的思想。状态转移方程为： $$f[i][j]=Max(f[i][j-1],f[i+2^{(j-1)}][j-1])$$ `f[i][0]`就是需要处理的数组 `a[i]`

2. 查询 对于查询的区间`[L,R]`: $$k=log(R-L+1)$$ $$Max[L,R]=max(f[L][k],f[R-2^k+1][k])$$ `f[L][k]`表示从左端点开始的区间 推导： $$L+2^k-1<=R$$ $$k<=log(R-L+1)$$ 对于右端点，取相同长度，则起点为： $$R-2^k+1$$

3. tips 求log采取

   ```
   log[0]=-1
   for(int i=1;i<=n;++i)
        log[i]=log[i>>1]+1;
   ```

   求2^k采取 1<<(k)

快读

```
inline int read()
{
    char c=getchar();int x=0,f=1;
    while(c<'0'||c>'9'){if(c=='-')f=-1;c=getchar();}
    while(c>='0'&&c<='9'){x=x*10+c-'0';c=getchar();}
    return x*f;
}
```