目录

[链表 2](#_Toc5270)

[▪ 结构体+指针 2](#_Toc12624)

[▪ 数组模拟(静态链表) 2](#_Toc8097)

[栈 3](#_Toc27234)

[队列 4](#_Toc24439)

[单调队列 5](#_Toc31969)

[KMP 5](#_Toc30870)

[Trie 6](#_Toc27598)

[并查集 7](#_Toc30862)

[堆 8](#_Toc4802)

[哈希表 9](#_Toc26553)

[▪ C++ STL简介 10](#_Toc3916)

# 链表

## 结构体+指针

## 数组模拟(静态链表)

### 单链表

#### 常考：邻接表 ；

*// head存储链表头，e[]存储节点的值，ne[]存储节点的next指针，idx表示当前用到了哪个节点*

int head, e[N], ne[N], idx;

*// 初始化*

void init()

{

    head = -1;

    idx = 0;

}

*// 在链表头插入一个数a*

void insert(int a)

{

    e[idx] = a, ne[idx] = head, head = idx ++ ;

}

*//将x插入到k点后面*

void add(int k, int x)

{

    e[idx] = x, ne[idx] = ne[k], ne[k] = idx ++ ;

}

*// 将下标是k的点后面的点删掉*

void remove(int k)

{

    ne[k] = ne[ne[k]];

}

*// 将头结点删除，需要保证头结点存在*

void remove()

{

    head = ne[head];

}

### 双链表

*// e[]表示节点的值，l[]表示节点的左指针，r[]表示节点的右指针，idx表示当前用到了哪个节点*

int e[N], l[N], r[N], idx;

*// 初始化*

void init()

{

*//0是左端点，1是右端点*

    r[0] = 1, l[1] = 0;

    idx = 2;

}

*// 在节点a的右边插入一个数x*

void insert(int a, int x)

{

    e[idx] = x;

    l[idx] = a, r[idx] = r[a];

    l[r[a]] = idx, r[a] = idx ++ ; //顺序不可交换

}

*// 删除节点a*

void remove(int a)

{

    l[r[a]] = l[a];

    r[l[a]] = r[a];

}

# 栈

*// tt表示栈顶*

int stk[N], tt = 0;

*// 向栈顶插入一个数*

stk[ ++ tt] = x;

*// 从栈顶弹出一个数*

tt -- ;

*// 栈顶的值*

stk[tt];

*// 判断栈是否为空*

if (tt > 0)

{

}

# 队列

1. 普通队列：

// hh 表示队头，tt表示队尾

int q[N], hh = 0, tt = -1;

// 向队尾插入一个数

q[ ++ tt] = x;

// 从队头弹出一个数

hh ++ ;

// 队头的值

q[hh];

// 判断队列是否为空

if (hh <= tt)

{

}

2. 循环队列

// hh 表示队头，tt表示队尾的后一个位置

int q[N], hh = 0, tt = 0;

// 向队尾插入一个数

q[tt ++ ] = x;

if (tt == N) tt = 0;

// 从队头弹出一个数

hh ++ ;

if (hh == N) hh = 0;

// 队头的值

q[hh];

// 判断队列是否为空

if (hh != tt)

{

}

### 单调栈

常见模型：找出每个数左边离它最近的比它大/小的数

维护栈内的元素单调递增

int tt = 0;

for (int i = 1; i <= n; i ++ )

{

    while (tt && check(stk[tt], i)) tt -- ;

    stk[ ++ tt] = i;

}

# 单调队列

常见模型：找出滑动窗口中的最大值/最小值

int hh = 0, tt = -1;

for (int i = 0; i < n; i ++ )

{

    while (hh <= tt && check\_out(q[hh])) hh ++ ;  *// 判断队头是否滑出窗口*

    while (hh <= tt && check(q[tt], i)) tt -- ;

    q[ ++ tt] = i;

}

# KMP

*// s[]是长文本，p[]是模式串，n是s的长度，m是p的长度*

*//求模式串的Next数组：*

for (int i = 2, j = 0; i <= m; i ++ )

{

    while (j && p[i] != p[j + 1]) j = ne[j];

    if (p[i] == p[j + 1]) j ++ ;

    ne[i] = j;

}

*// 匹配*

for (int i = 1, j = 0; i <= n; i ++ )

{

    while (j && s[i] != p[j + 1]) j = ne[j];

    if (s[i] == p[j + 1]) j ++ ;

    if (j == m)

    {

        j = ne[j];

*// 匹配成功后的逻辑*

    }

}

# Trie

#### 高效的存储和查找字符串

int son[N][26], cnt[N], idx;

*// 0号点既是根节点，又是空节点*

*// son[][]存储树中每个节点的子节点*

*// cnt[]存储以每个节点结尾的单词数量*

*// 插入一个字符串*

void insert(char \*str)

{

    int p = 0;

    for (int i = 0; str[i]; i ++ )

    {

        int u = str[i] - 'a';

        if (!son[p][u]) son[p][u] = ++ idx;

        p = son[p][u];

    }

    cnt[p] ++ ;

}

*// 查询字符串出现的次数*

int query(char \*str)

{

    int p = 0;

    for (int i = 0; str[i]; i ++ )

    {

        int u = str[i] - 'a';

        if (!son[p][u]) return 0;

        p = son[p][u];

    }

    return cnt[p];

}

# 并查集

#### 面试易考，思路精巧

#### 作用：1.合并两个集合 2.询问两个元素是否在一个集合

#### 优化：路径压缩

(1)朴素并查集：

    int p[N]; *//存储每个点的祖宗节点*

*// 返回x的祖宗节点*

    int find(int x)

    {

        if (p[x] != x) p[x] = find(p[x]);

        return p[x];

    }

*// 初始化，假定节点编号是1~n*

    for (int i = 1; i <= n; i ++ ) p[i] = i;

*// 合并a和b所在的两个集合：*

    p[find(a)] = find(b);

(2)维护size的并查集：

    int p[N], size[N];

*//p[]存储每个点的祖宗节点, size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的数量*

*// 返回x的祖宗节点*

    int find(int x)

    {

        if (p[x] != x) p[x] = find(p[x]);

        return p[x];

    }

*// 初始化，假定节点编号是1~n*

    for (int i = 1; i <= n; i ++ )

    {

        p[i] = i;

        size[i] = 1;

    }

*// 合并a和b所在的两个集合：*

    size[find(b)] += size[find(a)];

    p[find(a)] = find(b);

(3)维护到祖宗节点距离的并查集：

    int p[N], d[N];

*//p[]存储每个点的祖宗节点, d[x]存储x到p[x]的距离*

*// 返回x的祖宗节点*

    int find(int x)

    {

        if (p[x] != x)

        {

            int u = find(p[x]);

            d[x] += d[p[x]];

            p[x] = u;

        }

        return p[x];

    }

*// 初始化，假定节点编号是1~n*

    for (int i = 1; i <= n; i ++ )

    {

        p[i] = i;

        d[i] = 0;

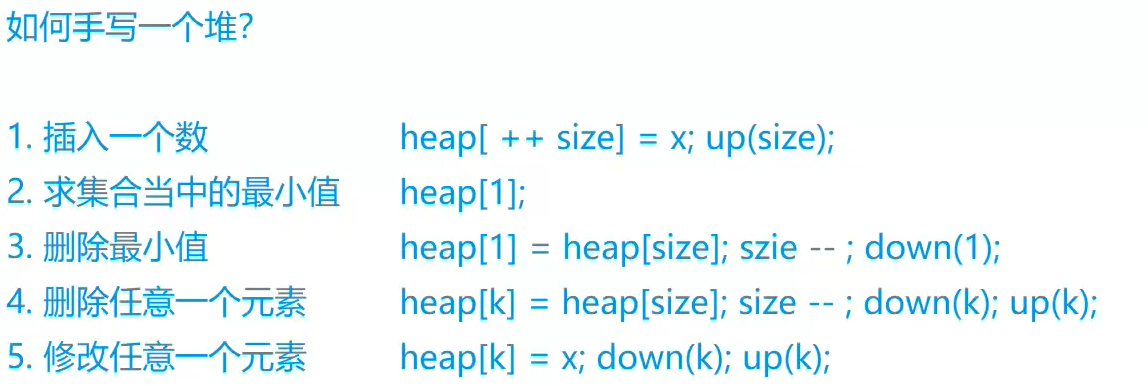
    }

*// 合并a和b所在的两个集合：*

    p[find(a)] = find(b);

    d[find(a)] = distance; *// 根据具体问题，初始化find(a)的偏移量*

# 堆



*// h[N]存储堆中的值, h[1]是堆顶，x的左儿子是2x, 右儿子是2x + 1*

*// ph[k]存储第k个插入的点在堆中的位置*

*// hp[j]存储堆中下标是j的点是第几个插入的*

int h[N], ph[N], hp[N], size;

*// 交换两个点，及其映射关系*

void heap\_swap(int a, int b)

{

    swap(ph[hp[a]],ph[hp[b]]);

    swap(hp[a], hp[b]);

    swap(h[a], h[b]);

}

void down(int u)

{

    int t = u;

    if (u \* 2 <= size && h[u \* 2] < h[t]) t = u \* 2;//左儿子存在且比左儿子大

    if (u \* 2 + 1 <= size && h[u \* 2 + 1] < h[t]) t = u \* 2 + 1;

    if (u != t) heap\_swap(u, t), down(t);

}

void up(int u)

{

    while (u / 2 && h[u] < h[u / 2])//父结点存在且比父结点小

    {

        heap\_swap(u, u / 2);

        u >>= 1;

    }

}

# 哈希表

#### 核心思想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低

#### 小技巧：取模的数用2^64，这样直接用unsigned long long存储，溢出的结果就是取模的结果

#### 存储方式

## 一般哈希

### 拉链法

    int h[N], e[N], ne[N], idx;

*// 向哈希表中插入一个数*

    void insert(int x)

    {

        int k = (x % N + N) % N;

        e[idx] = x;

        ne[idx] = h[k];

        h[k] = idx ++ ;

    }

*// 在哈希表中查询某个数是否存在*

    bool find(int x)

    {

        int k = (x % N + N) % N;

        for (int i = h[k]; i != -1; i = ne[i])

            if (e[i] == x)

                return true;

        return false;

    }

### 开放寻址法

    int h[N];

*// 如果x在哈希表中，返回x的下标；如果x不在哈希表中，返回x应该插入的位置*

    int find(int x)

    {

        int t = (x % N + N) % N;

        while (h[t] != null && h[t] != x)

        {

            t ++ ;

            if (t == N) t = 0;

        }

        return t;

    }

## 字符串哈希

typedef unsigned long long ULL;

ULL h[N], p[N]; *// h[k]存储字符串前k个字母的哈希值, p[k]存储 P^k mod 2^64*

*// 初始化*

p[0] = 1;

for (int i = 1; i <= n; i ++ )

{

    h[i] = h[i - 1] \* P + str[i];

    p[i] = p[i - 1] \* P;

}

*// 计算子串 str[l ~ r] 的哈希值*

ULL get(int l, int r)

{

    return h[r] - h[l - 1] \* p[r - l + 1];

}

## C++ STL简介

vector, 变长数组，倍增的思想

    size()  返回元素个数

    empty()  返回是否为空

    clear()  清空

    front()/back()

    push\_back()/pop\_back()

    begin()/end()

    []

    支持比较运算，按字典序

pair<int, int> //存储一个元素的两种属性

    first, 第一个元素

    second, 第二个元素

    支持比较运算，以first为第一关键字，以second为第二关键字（字典序）

string，字符串

    size()/length()  返回字符串长度

    empty()

    clear()

    substr(起始下标，(子串长度))  返回子串

    c\_str()  返回字符串所在字符数组的起始地址//printf(“%s”,a.c\_str());

queue, 队列

    size()

    empty()

    push()  向队尾插入一个元素

    front()  返回队头元素

    back()  返回队尾元素

    pop()  弹出队头元素

priority\_queue, 优先队列，默认是大根堆

    size()

    empty()

    push()  插入一个元素

    top()  返回堆顶元素

    pop()  弹出堆顶元素

    定义成小根堆的方式：priority\_queue<int, vector<int>, greater<int>> q; // -x

stack, 栈

    size()

    empty()

    push()  向栈顶插入一个元素

    top()  返回栈顶元素

    pop()  弹出栈顶元素

deque, 双端队列

    size()

    empty()

    clear()

    front()/back() //返回第一/最后一个元素

    push\_back()/pop\_back()

    push\_front()/pop\_front()

    begin()/end()

    []

set, map, multiset, multimap, 基于平衡二叉树（红黑树），动态维护有序序列

    size()

    empty()

    clear()

    begin()/end()

    ++, -- 返回前驱和后继，时间复杂度 O(logn)

    set/multiset

        insert()  插入一个数

        find()  查找一个数

        count()  返回某一个数的个数

        erase()

            (1) 输入是一个数x，删除所有x   O(k + logn) //

            (2) 输入一个迭代器，删除这个迭代器

        lower\_bound()/upper\_bound()

            lower\_bound(x)  返回大于等于x的最小的数的迭代器

            upper\_bound(x)  返回大于x的最小的数的迭代器

    map/multimap

        insert()  插入的数是一个pair

        erase()  输入的参数是pair或者迭代器

        find()

        []  注意multimap不支持此操作。 时间复杂度是 O(logn)//取值类似python的字典

        lower\_bound()/upper\_bound()

unordered\_set, unordered\_map, unordered\_multiset, unordered\_multimap, 哈希表

    和上面类似，增删改查的时间复杂度是 O(1)

    不支持 lower\_bound()/upper\_bound()， 迭代器的++，--

bitset, 圧位 //bool 压8位

    bitset<10000> s;

    ~, &, |, ^

    >>, <<

    ==, !=

    []

    count()  返回有多少个1

    any()  判断是否至少有一个1

    none()  判断是否全为0

    set()  把所有位置成1

    set(k, v)  将第k位变成v

    reset()  把所有位变成0

    flip()  等价于~

    flip(k) 把第k位取反