11. 数据结构

11.1 链表

链表分为单链表和双链表.多个单链表组成的邻接表常用于存树和图.双链表一般用于优化.

可用指针和结构体来实现链表,即动态链表,代码:

```
struct Node {
  int val;
  Node* nxt;
};
```

这种实现方式每次插入一个节点都要调用new函数,非常慢,一般不用这种方式.一般用数组模拟链表,即静态链表.一般不用结构体数组,因为代码多.

11.1.1 单链表

用变量$head$表示头指针,再开一个数组$val[]$用于存各节点的数值,再开一个数组$nxt[]$存各节点的next指针,即该节点的后继节点的下标.两者通过各节点的下标联系,其中节点的下标一般从0开始,空指针记为-1.最后用一个从0开始的idx表示当前用到的节点的编号,相当于指针.

题意

实现一个单链表,初始为空,支持如下三种操作:①向链表头插入一个数;②删除第k个插入的数后面的一个数;③在k个插入的数后插入一个数.若操作过程共插入n个数,则按插入顺序,这n个数依次为:第1个插入的数、第2个插入的数、$\cdots$.

现对链表进行M $(1 \leq M \leq 1e5)$次操作,进行完所有操作后从头到尾输出整个链表,操作命令有如下三种:

①H x,表示向链表头插入一个数x.

②D k,表示删除第k个插入的数后面的一个数,$k = 0$时表示删除头节点.

③I k x,表示在第k个插入的数后面插入一个数x,数据保证$k > 0$.

数据保证所有操作合法.

思路

①操作:先将新节点的值存到$val[idx]$,后将新节点的next指向原头节点指向的节点,再将头节点的next指向新节点,最后idx后移一位.

②操作:因节点的编号按插入的顺序依次为$0, 1, \cdots$,则删除第k个插入的数后面的一个数即删除第$(k-1)$个节点的后继,用将第k个插入的节点的next指向其下一个节点的后继实现.

$k = 0$,即删除头节点时,只需让head指向原头节点的后继.

③操作:先将新节点的值存到$val[idx]$,后将新节点的next指向第k个插入的节点指向的节点,再将第k个插入的节点的next指向新节点,最后idx后移一位.

事实上,单链表可在任一位置插入,但为保证$O(1)$的时间复杂度,只能插入到第k个插入的节点之后,因为单链表能在$O(1)$的时间内找到某个节点的后继,但无法找到某个节点的前驱,为找到前驱只能从头遍历链表.

注意②、③操作中函数传参要传$k-1$,因为节点编号从0开始.

代码

```cpp
const int MAXN = 1e5 + 5;
int head;  // 头指针
int val[MAXN];  // 存各节点的数值
int nxt[MAXN];  // 存各节点的后继
int idx;  // 当前用到的节点的编号

void init() {  // 初始化
  head = -1, idx = 0;
}

void add_to_head(int x) {  // 将x插入到头节点
  val[idx] = x, nxt[idx] = head, head = idx++;
}

void add(int k, int x) {  // 将x插入到第k个插入的节点后
  val[idx] = x, nxt[idx] = nxt[k], nxt[k] = idx++;
}

void remove(int k) {  // 删除第k个插入的节点后的一个数
  nxt[k] = nxt[nxt[k]];
}

void print() {  // 从前往后打印链表
  for (int i = head; ~i; i = nxt[i])
    cout << val[i] << ' ';
  cout << endl;
}

int main() {
  init();

  while (m--) {
    char op; cin >> op;
    int k, x;
    switch (op) {
      case 'H':
        cin >> x;
        add_to_head(x);
        break;
      case 'D':
        cin >> k;
        if (!k) head = nxt[head];  // k=0时表示删除头节点
        else remove(k - 1);
        break;
      case 'I':
        cin >> k >> x;
        add(k - 1, x);
        break;
    }
  }
  print();
}
```

11.1.2 双链表

与单链表相比,每个节点有两个指针,分别指向其前驱和后继,故开两个数组$pre[]$和$nxt[]$分别存节点的前驱和后继.

双链表中可不定义头节点,让下标为0的点作头节点head,下标为1的节点作尾节点tail,初始化时让0号点的右边为1号点,1号点的左边为0号点,已用两个点,故idx从2开始.

题意

实现一个双链表,初始为空,支持如下五种操作:①在最左端插入一个数;②在最右端插入一个数;③删除第k个插入的数;④在第k个插入的数的左侧插入一个数;⑤在第k个插入的数的右侧插入一个数.

现对链表进行M $(1 \leq M \leq 1e5)$次操作,进行完所有操作后从左到右输出整个链表,操作命令有如下五种:

①L x,表示在链表最左端插入数x.

②R x,表示在链表最右端插入数x.

③D k,表示删除第k个插入的数;

④IL k x,表示在第k个插入的数的左侧插入一个数.

⑤RL k x,表示在第k个插入的数的右侧插入一个数.

思路

(1)显然插入操作都可用操作⑤完成:

操作⑤:先将新节点的值存到$val[idx]$,后将新节点的前驱指向第k个节点,新节点的后继指向原第k个节点的后继,再将第k个节点的后继指向新节点,将原第k个节点的后继的前驱指向新节点(注意顺序),最后idx后移一位.

操作④:操作⑤的函数传节点k的后驱.

(2)操作③:将第k个插入的数的前驱的后继指向原第k个插入的数的后继,将第k个插入的数的后继的前驱指向原第k个插入的数的前驱.

代码

```
const int MAXN = 1e5 + 5;
int val[MAXN];  // 存各节点的数值
int pre[MAXN], nxt[MAXN];  // 存各节点的前驱和后继
int idx;  // 当前用到的节点的编号

void init() {  // 初始化
  nxt[0] = 1, pre[1] = 0;
  idx = 2;
}

void insert(int k, int x) {  // 在第k个插入的节点的右侧插入x
  val[idx] = x;
  pre[idx] = k, nxt[idx] = nxt[k];
  pre[nxt[k]] = idx, nxt[k] = idx;
  idx++;
}

void remove(int k) {  // 删除第k个插入的节点
  pre[nxt[k]] = pre[k], nxt[pre[k]] = nxt[k];
}

void print() {  // 从前往后打印链表
```

```
  for (int i = nxt[0]; i != 1; i = nxt[i])
    cout << val[i] << ' ';
  cout << endl;
}

int main() {
  init();

  while (m--) {
    string op; cin >> op;
    int k, x;
    if (op == "L") {
      cin >> x;
      insert(0, x);
    }
    else if (op == "R") {
      cin >> x;
      insert(pre[1], x);
    }
    else if (op == "D") {
      cin >> k;
      remove(k + 1);
    }
    else if (op == "IL") {
      cin >> k >> x;
      insert(pre[k + 1], x);
    }
    else {
      cin >> k >> x;
      insert(k + 1, x);
    }
  }
  print();
}
```

11.1.3 合并两有序链表

题意

给定两升序排序的链表(长度≤ 500),合并它们使得新链表的节点升序排列.

思路

先合并再排序的时间复杂度太高.

考虑用归并排序中合并两有序序列的方法,即分别让两个指针指向两链表头,每次比较两个节点的大小,将小的节点连到答案链表中,对应的指针后移一位.待一个链表为空时,将另一链表接到答案链表中.

代码

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* merge(ListNode* l1, ListNode* l2) {
        auto ans = new ListNode(-1);  // 答案链表
        auto cur = ans;  // 当前遍历到的答案链表中的节点

        while(l1 && l2){
            if(l1->val < l2->val){
                cur->next = l1;
                cur = l1;
                l1 = l1->next;
            }
            else{
                cur->next = l2;
                cur = l2;
                l2 = l2->next;
            }
        }

        if(l1) cur->next = l1;
        else cur->next = l2;

        return ans->next;
    }
};
```

11.1.4 反转链表

题意

输入一个链表(长度≤ 30)的头节点,反转该链表并输出反转后的链表的头节点.

思路

因单链表不能从前往后遍历到前驱节点,故开一个变量记录前驱节点.

代码

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
```

```
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* pre = nullptr;
        auto cur = head;

        while(cur){  // 从前往后遍历链表
            auto next = cur->next;
            cur->next = pre;
            pre = cur;
            cur = next;
        }

        return pre;
    }
};
```

11.1.5 两链表的第一个公共节点

题意

输入两个链表(长度$\leq$ 2000),返回它们的第一个公共节点.若不存在公共节点,返回空节点.

思路

两指针p和q最初分别指向两链表头,若一个链表遍历完则该指针指向另一链表的头节点.显然它们的经过的距离相同.若两链表有公共节点,则会在第一个公共节点相遇,否则两指针分别遍历完两个链表后都走到空节点.

代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *findFirstCommonNode(ListNode *headA, ListNode *headB) {
        auto p = headA, q = headB;
        while (p != q) {
            if (p) p = p->next;
            else p = headB;
            if (q) q = q->next;
            else q = headA;
        }

        return p;
    }
};
```

11.1.6 删除链表中重复的节点

题意

输入一个有序的含重复节点的链表(长度≤ 100,节点的值的范围$[0,100]$),删去其中重复的节点,重复的节点不保留.

思路

注意到有序链表中重复值相邻.用指针p记录上一段的终点,用指针q遍历下一段.

注意可能删去头节点,可定义一个虚头节点防止特判.

代码

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* deleteDuplication(ListNode* head) {
        auto dummy = new ListNode(-1);
        dummy->next = head;

        auto p = dummy;  // 记录上一段的终点
        while(p->next){  // 注意不是while(p)
            auto q = p->next;  // 遍历下一段
            while(q && p->next->val == q->val) q = q->next;

            if(p->next && p->next->next == q) p = p->next;
            else p->next = q;  // 删去重复的部分
        }

        return dummy->next;
    }
};
```

11.1.7 倒序打印链表

题意

给定一个链表(长度≤ 1000)的头节点,倒序返回各节点的值.

思路

顺序遍历链表,将节点的值记录在一个数组中,再将数组反转,时间复杂度$O(n)$.

代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> printListReversingly(ListNode* head) {
        vector<int> ans;
        while(head){
            ans.push_back(head->val);
            head = head->next;
        }

        return vector<int>(ans.rbegin(), ans.rend());
    }
};
```

11.2 栈

栈的特点:先进后出.

11.2.1 模拟栈

题意

实现一个栈,初始为空,支持如下四种操作:①将某元素压入栈;②弹出栈顶元素:③判断栈是否为空;④查询栈顶元素.

现对栈进行M $(1 \leq M \leq 1e5)$次操作,操作命令有下面四种:

①$push\ x$,表示向栈顶插入一个数x $(1 \leq x \leq 1e9)$.

②$pop\ x$,表示从栈顶弹出一个数.

③$empty$,判断栈是否为空,是则输入"YES",否则输出"NO".

④$query$,查询栈顶元素,输出栈顶元素的值.

数据保证操作合法.

思路

用一个数组$stk[]$模拟栈,再用一个从1开始的变量cnt表示栈中元素的数量,同时作为栈中元素的下标.

①操作,$stk[++cnt] = x$.

②操作,$cnt--$.

③操作,$if(cnt)$.

④操作,栈顶元素即$stk[cnt]$.

代码

```cpp
const int MAXN = 1e5 + 5;
int stk[MAXN];  // 栈
int cnt = 0;  // 栈中元素个数

void push(int x) {  // 压栈
  stk[++cnt] = x;
}

void pop() {  // 弹出栈顶元素
  cnt--;
}

bool empty() {  // 询问栈是否为空
  if (cnt) return false;
  return true;
}

int top() {  // 查询栈顶元素
  return stk[cnt];
}

int main() {
  while (m--) {
    string op; cin >> op;
    int x;
    if (op == "push") { cin >> x; push(x); }
    else if (op == "pop") pop();
    else if (op == "query") cout << top() << endl;
    else {
      if (empty()) cout << "YES" << endl;
      else cout << "NO" << endl;
    }
  }
}
```

11.2.2 表达式求值

一般表达式求值中出现的运算符都是二元运算符.

题意

给定一长度不超过$1e5$表达式,其中运算符仅包含$+,-,*,/$(加、减、乘、整除),可能包含括号,输出改表达式的值.

数据保证:

①表达式合法.

②"$-$"只作为减号出现,不作为负号出现.

③表达式中的数都为正整数.

④表达式在中间过程和计算结果中的值都不超过$2^{31}-1$.

⑤整除指对0取整,即大于0的结果向下取整,小于0的结果向上取整.如$5/3=1,5/(1-4)=-1$.

思路

将表达式化为一棵表达式树,其中内部节点为运算符,叶子节点为数字.

以表达式$(2+2)*(1+1)$为例:

(1)中缀表达式树,对应树的中序遍历:$*\begin{cases}+\begin{cases}2\\2\end{cases}\\+\begin{cases}1\\1\end{cases}\end{cases}$,但这样对应的是$2+2\times1+1$,与原来运算符的优先级不同,要注意加括号使得加法在乘法之前计算.显然,中序遍历遍历到某棵子树的父亲节点时,该子树的表达式的值已被计算出来,只需将该父亲节点的值置为子树表达式的值.但遍历到该子树的父亲节点的父亲节点时,后者的右子树的值还未被计算,故先计算右子树的表达式的值,最后更新该父节点的值,重复该过程直至根节点.

判断子树被遍历完:深度减小,则已遍历完子树;深度增加,则未遍历完子树.显然深度深的运算符优先级高,先被计算,则判断子树被遍历完的条件等价于当前运算符的优先级高于上一个运算符的优先级.

若表达式中有括号,因中序遍历子树的过程是从根节点往叶子节点走,故括号内的表达式中运算符优先级递增,故从右括号往左括号计算该括号内的表达式的值.

可递归求左右子树的值即可求得根节点的值,也可用栈来实现类似的效果,以用栈实现为例:

写一个函数$cal()$实现用最后一个运算符操作最后两个数:先取出最后两个数(注意倒着取,因为右叶子节点的数后进),分别处理各运算符,最后将结果压入栈.

从左往右依次读入表达式:

①遇到数字时将其取出并压入栈,取出过程可用双指针实现.

②遇到左括号时将其压入栈.

③遇到右括号时,将栈内的数和运算符从后往前计算直至遇到左括号.

④遇到其他运算符时,若栈非空,且栈顶运算符优先级$\geq$当前运算符的优先级,则进行$cal()$操作,否则将当前数压入栈.最后将栈中未操作的运算符从右往左操作一遍,最终答案是栈顶元素.

(2)也可用后缀表达式树,对应树的后序遍历,如上面的树的后续遍历为$2\ 2\ +\ 1\ 1\ +\ *$,无需添加括号.中缀表达式需存运算符,但后缀表达式不需要.

代码

```cpp
unordered_map<char, int> pro{ {'+',1}, {'-',1},{'*',2}, {'/',2},{'^',3} };  // 运算符的优先级
stack<int> nums;  // 数
stack<char> op;  // 运算符

void cal() {
  auto b = nums.top(); nums.pop();  // 注意倒着取
  auto a = nums.top(); nums.pop();
  auto c = op.top(); op.pop();

  int res;
  switch (c) {
    case '+':
      res = a + b;
      break;
    case '-':
      res = a - b;
      break;
    case '*':
      res = a * b;
```

```
      break;
    case '/':
      res = a / b;
      break;
    case '^':
      res = pow(a, b);
      break;
  }
  nums.push(res);  // 将结果压入栈
}

int main() {
  string str; cin >> str;

  for (int i = 0; i < str.size(); i++) {
    auto c = str[i];

    if (isdigit(c)) {  // 取出数字,压入栈
      int tmp = 0;
      int j = i;  // 双指针
      while (j < str.size() && isdigit(str[j]))
        tmp = (tmp << 3) + (tmp << 1) + str[j++] - '0';

      i = j - 1;  // 更新指针i的位置
      nums.push(tmp);
    }
    else if (c == '(') op.push(c);
    else if (c == ')') {
      while (op.top() != '(') cal();  // 从右往左计算括号内的表达式直至遇到左括号
      op.pop();  // 弹出左括号
    }
    else {  // 一般运算符
      while (op.size() && op.top() != '(' && pro[op.top()] >= pro[c]) cal();
      op.push(c);
    }
  }

  while (op.size()) cal();  // 将栈中剩下的数和运算符从后往前操作
  cout << nums.top();
}
```

11.2.3 用两个栈实现队列

题意

用栈实现一个队列,支持如下四种操作:

①$push(x)$,表示将元素x插到队尾.

②$pop()$,表示弹出队首元素,并返回该元素.

③$peek()$,返回队首元素.

④$empty()$,返回队列是否为空.

数据保证所有操作合法,每组数据操作命令数≤ 100.

思路

②操作,开一个新的栈,将除了队首元素外的元素放到新栈中,弹出队首元素后再将剩余元素放回栈中.③操作同理.

①操作时间复杂度$O(1)$,②、③和④操作时间复杂度$O(n)$.

代码

```
class MyQueue {
public:
    /** Initialize your data structure here. */
    stack<int> stk, cache;  // cache用于暂存原栈的元素

    MyQueue() {

    }

    /** Push element x to the back of queue. */
    void push(int x) {
        stk.push(x);
    }

    void copy(stack<int>& a, stack<int>& b){
        while(a.size()){
            b.push(a.top());
            a.pop();
        }
    }

    /** Removes the element from in front of queue and returns that element. */
    int pop() {
        copy(stk, cache);
        int res = cache.top();
        cache.pop();
        copy(cache, stk);

        return res;
    }

    /** Get the front element. */
    int peek() {
        copy(stk, cache);
        int res = cache.top();
        copy(cache, stk);

        return res;
    }

    /** Returns whether the queue is empty. */
    bool empty() {
        return stk.empty();
    }
};

/**
 * Your MyQueue object will be instantiated and called as such:
```

```
 * MyQueue obj = MyQueue();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.peek();
 * bool param_4 = obj.empty();
 */
```

11.2.4 双向排序

原题指路: https://www.lanqiao.cn/problems/1458/learning/

题意

有一个长度为$n\ \ (1 \leq n \leq 1e5)$的序列$a = [a_1, \cdots, a_n]$, 其中$a_i = i\ \ (1 \leq i \leq n)$.

现有如下两种操作:

①0 q, 表示将前缀$a[1...q]\ \ (1 \leq q \leq n)$降序排列.

②1 q, 表示将后缀$a[q \cdots n]\ \ (1 \leq q \leq n)$升序排列.

给定$m\ \ (1 \leq m \leq 1e5)$个操作, 求操作后的序列.

思路

考察冗余操作.

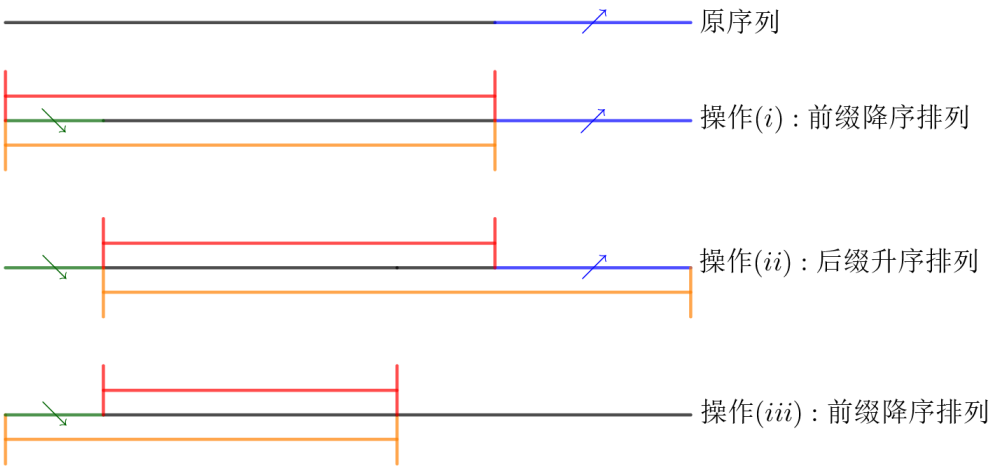
(1)对操作①, 若有操作"0 x"和"0 y", 且$x < y$, 则操作"0 x"可删去. 若有操作"0 y"和"0 x", 且$x < y$, 则操作"0 x"可删去. 同理可讨论操作②. 故连续的同种操作只需保留区间长度最大的, 此时操作为两种类型交替进行.

(2)不妨设先进行①操作. 对一个①操作A和它下一个的②操作B, 若后续存在比A的区间更长的①操作C, 则可将A, B删去. 此时操作①的区间长度依次递减, 即q递减. 同理操作②的区间长度递减, 即q递增. 若先进行②操作, 则不改变原序列.

(3)如下图, 以先进行①操作为例, 考察各个操作的等效操作.

约定: 橙色表示操作区间, 红色表示等效操作区间.

记操作(i)的操作区间为o_i, 等效操作区间为r_i, 绿色部分为g_i, 蓝色部分为b_i.



· 操作(i)为将前缀o_i降序排列, 则后缀b_i与原序列对应部分相同. 该操作等价于翻转r_i.

· 操作(ii)为将后缀o_{ii}升序排列, 则前缀$g_{ii} = g_i$.

因后缀$b_{ii} = b_i$, 且b_{ii}中的数$> (o_{ii} \backslash b_{ii})$中的数, 而$b_{ii}$降序排列, 则该操作等价于翻转$r_{ii}$.

· 操作(iii)为将前缀o_{iii}降序排列.

因前缀$g_{iii}=g_{ii}$, 且g_{iii}中的数$>(o_{iii}\backslash g_{iii})$中的数, 而$g_{iii}$中的数降序排列, 则该操作等价于翻转$r_{iii}$.

综上, 去除所有冗余操作后, 只需将每个操作与其上一个操作(若存在)的交集部分反转即可.

区间反转可用线段树或Splay实现, 但注意到每次操作后都会有一段与上个序列(若存在)相同的区间, 且每个不变的区间中的数连续, 则可用变量cur记录当前用到的数, 每次操作时填入不变的区间中即可. 如上图操作(i)不改变b_i, 不妨设b_i中的数为$[n-2,n-1,n]$. 操作(ii)不改变g_{ii}, 而$g_{ii}=g_i$, 则g_{ii}中的数为$[n-3,n-4,\cdots]$.

实现时, 用栈存每个操作, 维护当前等效修改区间$[l,r]$的左右端点, 每次操作时右移l或左移r, 并填入与上个序列相同的区间中的数. 若过程中$l>r$则退出. 所有操作结束后, 根据栈中的操作个数的奇偶性升序或降序将剩余的数填入区间$[l,r]$. 具体地, 若栈中有奇数个操作, 如只有一个操作①, 则区间$[l,r]$中的数为降序; 否则, 区间$[l,r]$中的数为升序.

代码

```cpp
const int MAXN = 1e5 + 5;
pair<int, int> stk[MAXN];  // first为op, second为pos
int top;
int ans[MAXN];

void solve() {
    int n, m; cin >> n >> m;

    // 去除冗余操作
    while (m--) {
        int op, pos; cin >> op >> pos;

        if (!op) {  // 前缀降序排列
            while (top && !stk[top].first)  // 连续的同种操作保留区间最长的
                pos = max(pos, stk[top--].second);
            while (top >= 2 && stk[top - 1].second <= pos)  // 去除冗余的上一个同种操作及其下一个异种操作
                top -= 2;
            stk[++top] = { 0, pos };
        }
        else if (top) {  // 后序升序排列, 注意前面有前缀操作时才需考察后缀操作
            while (top && stk[top].first)  // 连续的同种操作保留区间最长的
                pos = min(pos, stk[top--].second);
            while (top >= 2 && stk[top - 1].second >= pos)  // 去除冗余的上一个同种操作及其下一个异种操作
                top -= 2;
            stk[++top] = { 1, pos };
        }
    }

    int cur = n;  // 当前填到的数
    int l = 1, r = n;  // 等效修改区间
    for (int i = 1; i <= top; i++) {
        if (!stk[i].first)  // 前缀降序排列
            while (l <= r && r > stk[i].second) ans[r--] = cur--;
        else  // 后缀升序排列
            while (l <= r && l < stk[i].second) ans[l++] = cur--;

        if (l > r) break;
    }

    if (top & 1)
```

```
41          while (l <= r) ans[l++] = cur--;
42      else
43          while (l <= r) ans[r--] = cur--;
44
45      for (int i = 1; i <= n; i++)
46          cout << ans[i] << " \n"[i == n];
47  }
48
49  int main() {
50      solve();
51  }
```

11.2.5 Minimal string

原题指路: https://codeforces.com/problemset/problem/797/C

题意

给定一个长度不超过$1e5$的、只包含小写英文字母的字符串s. 将其字符从前往后入栈, 求字典序最小的出栈序列.

思路

设s的长度为n. 预处理s的后缀min数组$suf[]$.

贪心策略: 若当前的栈顶元素s_i不大于后缀$s[(i+1)\cdots n]$中的最小元素, 则将其出栈. 将所有元素都入栈后, 再逐个弹出栈中的元素即可.

代码

```
1   void solve() {
2       string s; cin >> s;
3       int n = s.length();
4       s = " " + s;
5
6       vector<char> suf(n + 2);  // 后缀min
7       suf[n + 1] = 127;
8       for (int i = n; i; i--)
9           suf[i] = min(suf[i + 1], s[i]);
10
11      stack<char> stk;
12      for (int i = 1; i <= n; i++) {
13          stk.push(s[i]);
14          while (stk.size() && stk.top() <= suf[i + 1]) {
15              cout << stk.top();
16              stk.pop();
17          }
18      }
19
20      for (; stk.size(); stk.pop())
21          cout << stk.top();
22      cout << endl;
23  }
24
25  int main() {
26      solve();
```

```
}
```

11.3 队列

队列的特点:先进先出.

11.3.1 模拟队列

题意

实现一个队列,初始为空,支持如下四种操作:①向队尾插入一个数;②从队首弹出一个数;③判断队列是否为空;④查询队首元素.

现对队列进行M $(1 \leq M \leq 1e5)$次操作,操作命令有如下四种:

①$push\ x$,表示向队尾插入一个数x $(1 \leq x \leq 1e9)$.

②$pop\ x$,表示从对头弹出一个数.

③$empty$,判断队列是否为空,若是则输出"YES",否则输出"NO".

④$query$,输出队首元素.

数据保证操作合法.

思路

用一个数组模拟$que[]$模拟队列,用两个变量$head$和$tail$分别表示队首和队尾,其中$tail$初始值为-1.

代码

```
const int MAXN = 1e5 + 5;
int que[MAXN];  // 队列
int head, tail = -1;  // 队首和队尾的指针

void push(int x) {  // 在队尾插入一个数x
  que[++tail] = x;
}

void pop() {  // 弹出队首元素
  head++;
}

bool empty() {  // 判断队列是否为空
  return head > tail;
}

int front() {  // 查询队首元素
  return que[head];
}

int back() {  // 查询队尾元素
  return que[tail];
}

```

```cpp
int main() {
  while (m--) {
    string op; cin >> op;
    int x;
    if (op == "push") {
      cin >> x; push(x);
    }
    else if (op == "pop") pop();
    else if (op == "empty") {
      if (empty()) cout << "YES" << endl;
      else cout << "NO" << endl;
    }
    else if (op == "query") cout << front() << endl;
    else cout << back() << endl;
  }
}
```

11.3.2 逛画展

题意

给定n $(1 \leq n \leq 1e6)$幅画作,它们由m $(1 \leq m \leq 2000)$位画师创作,其中第i幅画作由第$a[i]$ $(1 \leq a[i] \leq m)$个画师创作.求一个最短的连续区间$[l, r]$ $s.t.$ 每位都在其中出现过,下标从1开始.

思路

$cnt[i]$表示第i位画师出现的次数.第一次遍历时区间右端点不断右移直至所有画师都出现,然后画师出现次数$>$ 1的画师可出队,即区间左端点右移.第二次遍历从第一次遍历的终点开始,遍历剩下的画作,区间右端点右移并更新画师出现的次数.因本次遍历时刻保证区间内每个画师都出现过,则找到更短的区间时更新答案.

代码

```cpp
const int MAXN = 1e6 + 5;
int n, m;  // 图画数、画师数
int a[MAXN];  // 每幅画的画师
int cnt[MAXN];  // 记录每个画师出现的次数
int ansl, ansr;  // 答案区间

int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];

  int i = 1;
  int num = 0;  // 当前看了多少个画师
  int l = 1, r = 0;  // 当前区间,注意右端点从0开始,第一次右移后区间[l,r]=[1,1]
  while(num != m) {  // 区间右端点不断右移直至所有画师都出现
    if (!cnt[a[i]]) num++;  // 还没看过的画师
    r++, cnt[a[i++]]++;  // 区间右端点右移,更新画师出现的次数
  }

  while (cnt[a[l]] > 1) cnt[a[l++]]--;  // 出现次数多于1次的画师可以去掉

  ansl = l, ansr = r;  // 记录当前答案区间

  while(i <= n) {
```

```
    r++, cnt[a[i++]]++;  // 区间右端点右移,更新画师出现的次数

    while (cnt[a[l]] > 1) cnt[a[l++]]--;  // 出现次数多于1次的画师可以去掉

    if (ansr - ansl > r - l) ansl = l, ansr = r;  // 找到更短的区间
  }

  cout << ansl << ' ' << ansr;
}
```

11.4 单调栈

11.4.1 单调栈1

题意

给定一长度为n $(1 \le n \le 1e5)$的整数数列$a = [a_1, \cdots, a_n]$, 其中元素的范围为$[1, 1e9]$, 输出每个数左边第一个比它小的数, 若不存在则输出-1.

思路

朴素做法: 对每个数, 从当前位置往前遍历序列直至找到一个比它小的数.

用一个栈存每个数左边的数. 注意到若栈中存在一对$a_x \ge a_y$且$x < y$(即x比y更靠近栈底), 则可删去a_x, 故最后栈中的序列从栈底到栈顶必为严格递增的序列.

查询$a[i]$左边第一个比它小的数时, 从栈顶元素$stk[cnt]$开始找, 若$stk[cnt] \ge a[i]$, 则弹出栈顶元素, 直至找到一个比$a[i]$小的数, 再将$a[i]$压入栈. 若将栈清空都找不到一个符合的数, 则不存在符合的数.

代码

```
void solve() {
    stack<int> stk;
    CaseT {
        int a; cin >> a;

        while (stk.size() && stk.top() >= a) stk.pop();
        cout << (stk.empty() ? -1 : stk.top()) << ' ';
        stk.push(a);
    }
    cout << endl;
}

int main() {
    solve();
}
```

11.4.2 单调栈2

题意

给定有n $(1 \le n \le 3e6)$项的整数数列$a[]$,下标从1开始.定义函数$f(i)$为数列中第i个元素之后第一个大于a_i的元素的下标,即$f(i) = \min\limits_{i<j\le n, a_j>a_i} \{j\}$,若不存在,定义$f(i) = 0$.对每个$i \in [1, n]$,求$f(i)$.

代码

```cpp
const int MAXN = 3e6 + 5;
int n;  // 数组长度
int a[MAXN];  // 原数组
int f[MAXN];  // 答案
stack<int> stk;

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> a[i];

  for (int i = n; i >= 1; i--) {
    while (stk.size() && a[stk.top()] <= a[i]) stk.pop();
    f[i] = stk.empty() ? 0 : stk.top();
    stk.push(i);
  }

  for (int i = 1; i <= n; i++) cout << f[i] << ' ';
}
```

11.4.3 Look Up S

题意

n $(1 \le n \le 1e5)$头奶牛站成一排,编号$1 \sim n$,其中第i头奶牛身高为h_i $(1 \le h_i \le 1e6)$.现每只奶牛向右看齐.对奶牛i, j,若$i < j$且$h_i < h_j$,则称奶牛i仰望奶牛j.队每头奶牛,求离其最近的仰望对象的编号,若该奶牛无仰望对象,输出0.

思路I:单调栈

维护一个从栈底到栈顶单调增的单调栈.倒序遍历一遍$h[]$,对每个h,检查其是否不小于当前的栈顶元素,若是,则h是当前栈顶元素的仰望对象,记录答案并将当前栈顶元素出栈,最后将h入栈.

代码I

```cpp
const int MAXN = 1e6 + 5;
int n;  // 牛数
int h[MAXN];
stack<int> stk;
int ans[MAXN];

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> h[i];

  for (int i = n; i >= 1; i--) {
    while (stk.size() && h[stk.top()] <= h[i]) stk.pop();
```

```
    if (stk.size()) ans[i] = stk.top();  // 若栈为空,则该奶牛无仰望对象
    stk.push(i);
  }

  for (int i = 1; i <= n; i++) cout << ans[i] << endl;
}
```

思路II:单调队列

用队列模拟样例:3 2 6 1 1 2.先将它们编号$(3,1),(2,2),(6,3),(1,4),(1,5),(2,6)$.

先$(3,1)$入队(从队尾),然后$(2,2)$入队,因$2<3$,队列不变.

$(6,3)$入队,因$6>3,6>2$,则3号奶牛是1、2号奶牛的仰望对象,记录答案,1、2号奶牛出队.

$(1,4)$入队,队列不变.$(1,5)$入队,队列不变.

$(2,6)$入队,因$2>1,2>1$,则6号奶牛是4、5号奶牛的仰望对象,记录答案,4、5号出队.

代码II

```
const int MAXN = 1e6 + 5;
int n;  // 牛数
deque<pii> que;  // 元素是牛的身高、编号
int ans[MAXN];

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) {
    int h; cin >> h;
    while (que.size() && que.back().first < h) {
      ans[que.back().second] = i;
      que.pop_back();
    }
    que.push_back({ h,i });
  }

  for (int i = 1; i <= n; i++) cout << ans[i] << endl;
}
```

思路III:递推

因$ans[n]=0$,从$(n-1)$开始倒着扫一遍$h[]$,对每个$h[i]$,先与其右边的第一个数$h[j]$比较,若$h[i]$不小于$h[j]$,则$h[i]$与$h[j]$的仰望对象比较,重复该过程直至找到比$h[i]$大的数或确定i号奶牛无仰望对象.

代码III

```
const int MAXN = 1e6 + 5;
int n;  // 牛数
int h[MAXN];  // 身高
int ans[MAXN];

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> h[i];

```

```
  for (int i = n - 1; i >= 1; i--) {
    int j = i + 1;  // 与该数的右边的第一个数比较
    while (h[i] >= h[j] && h[j] > 0) j = ans[j];  // h[i]不小于h[j],则将其与h[j]的仰望对象比较
    ans[i] = j;  // 记录仰望对象
  }

  for (int i = 1; i <= n; i++) cout << ans[i] << endl;
}
```

11.4.4 发射站

例题

n $(1 \leq n \leq 1e6)$个基站排成一行,编号$1 \sim n$,其中第i个基站高度为h_i $(1 \leq h_i \leq 2e9)$,且能向两边(两端的发射站只能向一边)同时发射v_i $(1 \leq v_i \leq 1e4)$的能量,发出的能量只能被两边最近且比它高的发射站接收.求接受能量最多的发射站接收的能量.

基站用n行输入描述,其中第i行包含两个整数h_i, v_i,分别表示第i个基站的高度和发射的能量.

思路

维护一个从栈底到栈顶单调增的单调栈.

对每个h_i,若当前栈顶元素的高度不超过h_i,则栈顶元素的能量只能传给i号基站,更新能量并弹出栈顶元素.重复该过程直至栈为空或找到一个高度$>$ h_i的基站,该基站向i号基站传递能量.最后i号基站入栈.

代码

```
const int MAXN = 1e6 + 5;
int n;  // 基站数
int h[MAXN], v[MAXN];  // 基站的高度、发射的能量
stack<int> stk;
int sum[MAXN];  // sum[i]表示i号基站接收的能量之和

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> h[i] >> v[i];

  for (int i = 1; i <= n; i++) {
    while (stk.size() && h[stk.top()] < h[i]) {
      sum[i] += v[stk.top()];
      stk.pop();
    }
    if (stk.size()) sum[stk.top()] += v[i];
    stk.push(i);
  }

  int ans = 0;
  for (int i = 1; i <= n; i++) ans = max(ans, sum[i]);
  cout << ans;
}
```

11.4.5 Patrik音乐会的等待

题意

n $(1 \le n \le 5e5)$排成一行,编号$1 \sim n$,每个人的身高为h_i $(1 \le h_i < 2^{31})$.对任意两人a, b $(a \ne b)$,若他们相邻或他们之间没有人比a或b高,则a与b能互相看见.求有多少对人能互相看见.

思路

先考虑每个人身高不同的情况.注意到从i号人往右,若遇到一个比i高的人j,则i不能再看见j之后的人.维护一个从栈底到栈顶单调递增的单调栈.对每个高度h_i,若它不小于当前的栈顶元素,则更新答案并弹出栈顶元素.

因可能存在相同的身高,则还需记录相同身高的人数.

代码

```cpp
const int MAXN = 1e6 + 5;
int n;  // 人数
stack<pii> stk;  // 身高、该身高的人数

int main() {
  cin >> n;

  ll ans = 0;
  while (n--) {
    int h; cin >> h;
    pii tmp{ h,1 };  // 统计当前身高的人数
    while (stk.size() && stk.top().first <= h) {
      ans += stk.top().second;
      if (stk.top().first == h) tmp.second += stk.top().second;  // 更新当前身高的人数
      stk.pop();
    }

    if (stk.size()) ans++;
    stk.push(tmp);
  }
  cout << ans;
}
```

11.4.6 奶牛排队

题意

有n $(2 \le n \le 1e5)$头牛排成一行,编号$1 \sim n$,其中第i头牛身高h_i $(1 \le h_i < 2^{31})$.现找出一个区间的牛,使得最左端的牛A最矮,最右端的牛B最高,且B高于A.若它们间还有其他牛,则中间的牛身高不能与A、B的身高相同.求区间长度最大值.

思路

枚举区间右端点B.因左端点A处的牛最矮,则A是$h[1 \cdots B]$的后缀的最小值的下标.因A、B间不存在比B高的元素,则A的右边有且仅有B可作为$h[1 \cdots B]$的后缀的最大值的下标.只需找到从后往前数第二个后缀的最大值的下标k,则A是k右边的序列的后缀的最小值.

用单调栈维护当前序列后缀的最值.每次枚举到一个B时将新位置入栈,此时最大值栈顶是从后往前数第二个后缀最大值的下标.因最小值栈的下标单调,可二分出最靠左的A的位置.时间复杂度$O(n \log n)$.

代码

```cpp
const int MAXN = 1e5 + 5;
int n;  // 牛数
int h[MAXN];  // 牛的身高
int stkmax[MAXN], tailmax = 0;  // 最大值栈
int stkmin[MAXN], tailmin = 0;  // 最小值栈
int ans;

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> h[i];

  for (int i = 1; i <= n; i++) {
    while (tailmax && h[stkmax[tailmax]] < h[i]) tailmax--;  // 维护最大值栈
    while (tailmin && h[stkmin[tailmin]] >= h[i]) tailmin--;  // 维护最小值栈
    int k = upper_bound(stkmin + 1, stkmin + tailmin + 1, stkmax[tailmax]) - stkmin;  // k是从后往前数第二个后缀的最大值的下标
    if (k != tailmin + 1) ans = max(ans, i - stkmin[k] + 1);  // 若找到,则更新最长区间长度
    stkmax[++tailmax] = stkmin[++tailmin] = i;  // 入栈
  }
  cout << ans;
}
```

11.4.7 Mike and Feet

原题指路:https://codeforces.com/problemset/problem/547/B

题意

给定长度为n $(1 \le n \le 2e5)$的序列$a_1, \cdots, a_n$ $(1 \le a_i \le 1e9)$,对每个$x \in [1, n]$,求长度为x的区间中元素最小值的最大值.

思路

对每个i,找到最大的$j\ s.t.\ a_j < a_i$,记这样的j为l_i(若无这样的j,记$l_i = 0$).对每个i,找到最小的$j\ s.t.\ a_j < a_i$,记这样的j为r_i(若无这样的j,记$r_i = n + 1$).上述两过程可用单调栈在$O(n)$的时间复杂度内完成.

设长度为x的答案为ans_x.注意到$ans_1 \ge ans_2 \ge \cdots \ge ans_n$,最后倒着扫一遍更新即可.

代码

```cpp
const int MAXN = 2e5 + 5;
int n;
int a[MAXN];
int l[MAXN], r[MAXN];
int ans[MAXN];

void solve() {
  cin >> n;
  for (int i = 0; i < n; i++) cin >> a[i];

  stack<pii> stk;  // first为元素,second为下标
  stk.push({ -1,-1 });  // 哨兵
  for (int i = 0; i < n; i++) {
```

```
    while (stk.top().first >= a[i]) stk.pop();

    l[i] = stk.top().second;
    stk.push({ a[i],i });
  }

  stk = stack<pii>();  // 清空
  stk.push({ -1,n });  // 哨兵
  for (int i = n; i >= 0; i--) {
    while (stk.top().first >= a[i]) stk.pop();

    r[i] = stk.top().second;
    stk.push({ a[i],i });

    int len = r[i] - l[i] - 1;
    ans[len] = max(ans[len], a[i]);
  }

  for (int i = n - 1; i >= 0; i--) ans[i] = max(ans[i], ans[i + 1]);
  for (int i = 1; i <= n; i++) cout << ans[i] << " \n"[i == n];
}

int main() {
  solve();
}
```

11.4.8 移掉 K 位数字

原题指路: https://leetcode.cn/problems/remove-k-digits/

题意

给定一个用字符串表示的非负整数 num 和一个整数 k $(1 \le k \le |num| \le 1e5)$, 移除该数的恰 k 个数码, 使得剩下的数最小.

思路

因两长度相等的数的大小关系取决于最靠前的不同的数码, 为使得剩下的数最小, 应使得靠前的数码尽量小.

设 $num = \overline{d_1 \cdots d_n}$. 贪心策略: 从前往后求得第一个下标 $i \in [2, n]$ $s.t.$ $d_{i-1} > d_i$, 删除数码 d_{i-1}. 若不存在, 则序列不降, 删除最后一个数码即可. 总时间复杂度 $O(nk)$, 会TLE.

考虑优化, 从前往后遍历 num 的数码, 用栈维护删除后的数字. 具体地, 维护一个从栈底到栈顶不降的单调栈. 若当前数码 $<$ 栈顶元素, 则不断弹出栈顶元素, 直至栈为空或当前数码 $\ge$ 栈顶元素或已弹出了 k 个数码. 删除完成后, 从栈底到栈顶的元素即剩下的数.

特判如下情况:

① 栈空, 则剩下的数为 0.

② 剩下的数有前导零时需删去.

③ 删除了 m 个数码且 $m < k$, 需再删除末尾的 $(k - m)$ 个数码.

总时间复杂度 $O(n)$.

代码

```cpp
class Solution {
public:
    string removeKdigits(string num, int k) {
        vector<char> stk;
        for (auto ch : num) {
            while (stk.size() && stk.back() > ch && k) {
                stk.pop_back();
                k--;
            }
            stk.push_back(ch);
        }

        for (; k > 0; k--) stk.pop_back();

        int idx = 0;
        while (idx < stk.size() && stk[idx] == '0') idx++;

        string ans;
        for (; idx < stk.size(); idx++)
            ans += stk[idx];
        return ans == "" ? "0" : ans;
    }
};
```

11.5 单调队列

11.5.1 滑动窗口

题意

给定一个大小为n $(1 \leq n \leq 1e6)$的数组和一个大小为k的滑动窗口, 它从数组的最左边移动到最右边, 你只能在窗口中看到k个数字, 每次滑动窗口向右移动一个位置. 求滑动窗口位于每个位置时, 窗口中的最大值和最小值.

思路

朴素做法: 用一个队列维护当前窗口内的数, 每次扫描一遍当前队列求出最值.每次滑动时弹出队首元素, 将一个新元素入队. 每次扫描需k次, 共扫描n次, 总时间复杂度$O(nk)$.

考察某一时刻窗口的值, 以$3, -1, -3$为例. 因$-3 < -1$, 则只要-3在序列中, 最小值不会取到-1, 故可将-1删去. 显然最后的序列从队首到队尾单调增, 每个时刻的最小值时即队首元素, 每次查询的时间复杂度$O(1)$. 每次滑动时, 检查队首元素是否在窗口内, 若不在则弹出; 检查新加入的元素是否<队尾元素, 若是则队尾元素出队, 直至队尾元素≤新加入的元素, 再将新加入的元素入队.

求最大值时只需将上述过程对称地做一遍.

代码

```cpp
const int MAXN = 1e6 + 5;
int a[MAXN];
int que[MAXN], head, tail;

void solve() {
    int n, k; cin >> n >> k;
    for (int i = 0; i < n; i++) cin >> a[i];

    head = 0, tail = -1;
    for (int i = 0; i < n; i++) {
        if (head <= tail && i - k + 1 > que[head]) head++;

        while (head <= tail && a[que[tail]] >= a[i]) tail--;
        que[++tail] = i;

        if (i >= k - 1) cout << a[que[head]] << ' ';
    }
    cout << endl;

    head = 0, tail = -1;
    for (int i = 0; i < n; i++) {
        if (head <= tail && i - k + 1 > que[head]) head++;

        while (head <= tail && a[que[tail]] <= a[i]) tail--;
        que[++tail] = i;

        if (i >= k - 1) cout << a[que[head]] << ' ';
    }
    cout << endl;
}

int main() {
    solve();
}
```

11.5.2 好消息，坏消息

原题指路:https://www.luogu.com.cn/problem/P2629

题意

有n $(1 \leq n \leq 1e6)$条按顺序排列的消息,编号$1 \sim n$,其中第i条消息有好坏度a_i $(-1000 \leq a_i \leq 1000)$.现要将消息都告诉老板,初始时老板的心情为$0$,每被告知一条信息后老板的心情等于原心情加上该信息的好坏度.现可选择一个k,按$k, k+1, k+2, \cdots, n, 1, 2, \cdots, k-1$的顺序把消息告诉老板,求有多少个 $s.t.$ 老板任意时刻的心情非负的k.

思路I

展成链后求前缀和$sum[]$,枚举分割点查询区间和会T.

注意到对每个k,查询区间$[k, k+n-1]$中每个分割点i对应的区间和时,只要有一个点的区间和为负时,该区间已不合法,显然只需判断最小的$sum[i]$减去$sum[k-1]$是否非负即可.可用单调队列维护区间$[k, k+n-1]$中的最小值.

代码I

```cpp
const int MAXN = 2e6 + 5;
int n;  // 消息数
int a[MAXN];  // 消息的好坏度
ll sum[MAXN];  // a[]的前缀和
deque<int> que;

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
    a[n + i] = a[i];  // 倍增,展成链
  }

  for (int i = 1; i <= 2 * n - 1; i++) sum[i] = sum[i - 1] + a[i];  // 前缀和

  int ans = 0;
  for (int i = 1; i <= 2 * n - 1; i++) {
    while (que.size() && max(i - n + 1, 1) > que.front()) que.pop_front();  // 队首离开区间
    while (que.size() && sum[i] <= sum[que.back()]) que.pop_back();  // 队尾不是最小值
    que.push_back(i);
    if (i - n + 1 > 0 && sum[que.front()] - sum[i - n] >= 0) ans++;  // 心情非负
  }
  cout << ans;
}
```

思路II

显然只需要前缀和$sum[]$的最小值非负即可.不展成链,直接维护$sum[]$每个前缀的最小值$pre[]$和每个后缀的最小值$suf[]$.

代码II

```cpp
const int MAXN = 1e6 + 5;
int n;  // 消息数
ll sum[MAXN];  // 好坏度a[]的前缀和
ll pre[MAXN], suf[MAXN];  // sum[]的前缀最小值、后缀最小值

int main() {
  cin >> n;
  pre[0] = suf[n + 1] = INFF;

  for (int i = 1; i <= n; i++) {
    int a; cin >> a;
    sum[i] = sum[i - 1] + a;  // 预处理出a[]的前缀和
    pre[i] = min(pre[i - 1], sum[i]);  // 更新sum[]每个前缀的最小值
  }
  for (int i = n; i >= 1; i--) suf[i] = min(suf[i + 1], sum[i]);  // 更新sum[]每个后缀的最小值

  int ans = 0;
  for (int i = 1; i <= n; i++)  // 枚举分割点
    if ((pre[i - 1] + sum[n] - sum[i - 1] >= 0) && (suf[i] - sum[i - 1] >= 0)) ans++;  // 心情非负
  cout << ans;
```

```
21 }
```

11.6 堆

堆是一棵完全二叉树,常用于维护数的集合.

堆分为大根堆和小根堆.小根堆中,每个节点小于等于其左右儿子,根节点是整个集合中的最小值;大根堆中,每个节点大于等于左右儿子,根节点是集合中的最大值.

堆用一个一维数组存储,1号节点为根节点,节点x的左儿子为$x << 1$(即$2x$),右儿子为$x << 1|1$(即$2x+1$).

堆有两个基本操作$down()$和$up()$,分别表示将一个节点下移和上移.以大根堆为例:

①减小一个节点的值,可能需将其下移.$down()$的实现:将当前节点的值与其左右节点中的最小值交换,重复该操作直至无需交换.

②增大一个节点的值,可能需将其上移.$up()$的实现:若当前节点的值大于等于父亲节点,则将当前节点与父亲节点交换,重复该操作直至无需交换或当前节点为根节点.

$down()$和$up()$的执行次数与树的高度成正比,故时间复杂度$O(\log n)$.

11.6.1 模拟堆

题意

维护一个集合,初始时为空,支持如下五种操作:①插入一个数;②输出当前集合中的最小值;③删除当前集合中的最小值;④删除一个数;⑤修改一个数.(其中④和⑤在STL的优先队列中无现成的函数)

先对集合进行N $(1 \le N \le 1e5)$次操作,操作命令有如下五种:

①$I\ x$,表示插入一个数x $(-1e9 \le x \le 1e9)$.

②PM,输出当前集合中的最小值.

③DM,删除当前集合中的最小值,数据保证最小值唯一.

④$D\ k$,表示删除第k个插入的数.

⑤$C\ k\ x$,表示修改第k个插入的数为x.

数据保证操作合法.

思路

五种操作都可通过$down()$和$up()$的组合来完成.

用一个数组$heap[]$表示堆,下标从1开始(若从0开始,则其左儿子还是0).用一个变量cnt表示当前堆中节点的数量,初始时$cnt=0$.

①操作,将数x插到数组的末尾,即$heap[++cnt]=x$,再用$up(cnt)$维护堆,总时间复杂度$O(\log n)$.

②操作,当前集合中的最小值即根节点$heap[1]$,时间复杂度$O(1)$.

③操作,因用一维数组存,删除头节点不方便,但删除尾节点方便,故分三步:

1)用$heap[]$的最后一个元素覆盖根节点,即$heap[1]=heap[cnt]$.

2)删去最后一个节点,即$cnt--$.

3)维护堆,$down(1)$.

④操作,用tmp存$heap[k]$,后续将其与新值比较,根据两者的大小关系选择$down()$还是$up()$.与③操作类似,先用$heap[]$的最后一个元素覆盖第k个节点,即$h[k]=heap[cnt]$,再删去最后一个节点,即$cnt--$.

1)新值比原值大,为维护小根堆,则将其下移,$down(k)$.

2)新值比原值小,则将其上移,$up(k)$.

事实上可不管新值与原值的大小关系,每次都做$down(k)$和$up(k)$,但实际只会执行一个.

⑤操作,与④操作类似,$heap[k]=x,down(k),up(k)$.

用长度为n的数列建堆时,若执行n次插入操作,则时间复杂度$O(n\log n)$.事实上,可将数列的前$\left\lfloor\frac{n}{2}\right\rfloor$项都$down()$一次即可,时间复杂度$O(n)$,理由:以树的高度为4为例(根节点深度为1).第$\left\lfloor\frac{n}{2}\right\rfloor=3$层的节点数为$\frac{n}{4}$,它们$down()$一层至叶子节点,而叶子节点不需要$down()$;第2层的节点数为$\frac{n}{8}$,需$down()$两层,$\cdots$,则共需做$\frac{n}{4}\times1+\frac{n}{8}\times2+\frac{n}{16}\times3+\cdots=n\left(\frac{1}{2^2}+\frac{2}{2^3}+\frac{3}{2^4}+\cdots\right)$次,错位相减得:$\frac{1}{2^2}+\frac{2}{2^3}+\frac{3}{2^4}+\cdots=\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}+\cdots<1$,故总时间复杂度$O(n)$.

因$up()$操作需对第k个插入的数进行操作,故开两个数组$hp[]$和$ph[]$分别存第k个插入的点是哪个节点、各节点是第几次插入的,则交换节点的值时,需同时交换$hp[]$和$ph[]$指向的值.显然,若$ph[j]=k$,则$hp[k]=j$,即在第k个插入的点和堆的节点间建立双射.

代码

```
const int MAXN = 1e5 + 5;
int m = 0;  // 记录当前数是第几个插入的数
int heap[MAXN];  // 堆
int ph[MAXN], hp[MAXN];  // 第k个插入的点与堆的节点的双射
int cnt = 0;  // 节点数
int ls(int u) { return u << 1; }  // 左儿子
int rs(int u) { return u << 1 | 1; }  // 右儿子

void heap_swap(int a, int b) {  // 交换值的同时交换双射
  swap(ph[hp[a]], ph[hp[b]]);
  swap(hp[a], hp[b]);
  swap(heap[a], heap[b]);
}

void down(int u) {  // down操作
  int tmp = u;  // u节点
  if (ls(u) <= cnt && heap[ls(u)] < heap[tmp]) tmp = ls(u);  // 注意检查左节点是否存在,下同
  if (rs(u) <= cnt && heap[rs(u)] < heap[tmp]) tmp = rs(u);

  if (tmp != u) {  // 有交换
    heap_swap(u, tmp);
    down(tmp);
  }
}

void up(int u) {  // up操作
  while (u / 2 && heap[u] < heap[u / 2]) {
```

```
    heap_swap(u, u / 2);
    u >>= 1;
  }
}

void insert(int x) {  // 插入一个数x
  cnt++, m++;  // 更新节点数

  ph[m] = cnt, hp[cnt] = m;  // 建立双射
  heap[cnt] = x;
  up(cnt);  // 维护堆
}

int top() {  // 查询堆顶元素
  return heap[1];
}

void remove_min() {  // 删除集合中的最小值
  heap_swap(1, cnt);  // 用尾节点覆盖根节点
  cnt--;  // 更新节点数
  down(1);  // 维护堆
}

void remove(int k) {  // 删除第k个插入的数
  k = ph[k];  // 找到该元素在堆中的位置
  heap_swap(k, cnt);  // 用尾节点覆盖第k个插入的节点
  cnt--;  // 更新节点数
  down(k); up(k);  // 维护堆
}

void modify(int k, int x) {  // 修改第k个插入的数为x
  k = ph[k];  // 找到该元素在堆中的位置
  heap[k] = x;  // 修改值
  down(k); up(k);  // 维护堆
}

int main() {
  while (n--) {
    string op; cin >> op;
    int k, x;
    if (op == "I") { cin >> x; insert(x); }
    else if (op == "PM") cout << top() << endl;
    else if (op == "DM") remove_min();
    else if (op == "D") { cin >> k; remove(k); }
    else { cin >> k >> x; modify(k, x); }
  }
}
```

11.6.2 堆排序

题意

给定一个长度为n $(1 \leq n \leq 1e5)$的整数数列,其中元素范围为$[1, 1e9]$,从小到大输出前m $(1 \leq m \leq 1e5)$小的数.

思路

将整个数组建堆,后输出前m个节点的值.

代码

```cpp
const int MAXN = 1e5 + 5;
int heap[MAXN];  // 堆
int cnt = 0;  // 节点数
int ls(int u) { return u << 1; }  // 左儿子
int rs(int u) { return u << 1 | 1; }  // 右儿子

void down(int u) {  // down操作
  int tmp = u;  // u节点
  if (ls(u) <= cnt && heap[ls(u)] < heap[tmp]) tmp = ls(u);  // 注意检查左节点是否存在,下同
  if (rs(u) <= cnt && heap[rs(u)] < heap[tmp]) tmp = rs(u);

  if (tmp != u) {  // 有交换
    swap(heap[tmp], heap[u]);
    down(tmp);
  }
}

void up(int u) {  // up操作
  return;  // 本题用不到
}

void build(int n) {  // 建堆
  cnt = n;
  for (int i = n / 2; i; i--) down(i);
}

void remove(int k) {  // 删除节点u
  heap[k] = heap[cnt--];  // 用尾节点覆盖当前节点
  down(k); up(k);  // 维护堆
}

int main() {
  for (int i = 1; i <= n; i++) cin >> heap[i];

  build(n);  // 建堆

  while (m--) {
    cout << heap[1] << ' ';
    remove(1);
  }
}
```

11.6.3 动态中位数

题意

依次读入一个整数序列,每当读入的整数个数为奇数时,输出已读入的整数构成的序列的中位数.

有t $(1 \leq t \leq 1000)$组测试数据.每组测试数据第一行输入一个奇数m $(1 \leq m \leq 99999)$,表示序列中的元素个数.接下来的若干行每行输入10个整数,最后一行的输入量可能小于10个.数据保证所有测试数据的m之和不超过5e5.

对每组测试数据,第一行输入两整数,分别表示序列中的元素个数(m)和输出的中位数的个数(序列的元素个数加一除以二).接下来的若干行每行输出10个中位数,最后一行的输出量可能少于10个.

思路

注意到只关心中位数,则无需对较小的一半和较大的一半排序,只需对中间的几个数排序.考虑维护上面是小根堆,下面是大根堆的对顶堆,上面存较大的一半的数,下面存较小的一半的数,使得读入的整数个数为偶数时上下两堆中的元素个数相等,读入个数为奇数时下面的元素个数比上面的元素个数多1,这样每次的中位数都是下面堆的堆顶元素.

对顶堆的维护:①小根堆up的堆顶元素≥大根堆$down$的堆顶元素,这样小根堆中是序列中较大的一半,大根堆中是序列中较小的一半;②插入一个元素x时,若$x \leq down.top()$,则将其插入大根堆中;否则将其插入小根堆中.初始时两个堆都为空,将第一个元素插入大根堆;③若up的元素过多,则将$up.top()$插入到$down$中;若$down$的元素过多,则将$down.top()$插入到up中.

每次维护的时间复杂度都为$O(\log n)$,故每组测试数据的时间复杂度为$O(n \log n)$.

代码

```cpp
int main() {
  CaseT{
    int n, m; cin >> m >> n;
    cout << m << ' ' << (n + 1) / 2 << endl;

    // 对顶堆
    pque<int> down;
    pque<int, vi, greater<int>> up;

    int cnt = 0;  // 读入的元素个数
    for (int i = 1; i <= n; i++) {
      int x; cin >> x;

      if (down.empty() || x <= down.top()) down.push(x);
      else up.push(x);

      if (down.size() > up.size() + 1) up.push(down.top()), down.pop();
      if (up.size() > down.size()) down.push(up.top()), up.pop();

      if (i & 1) {
        cout << down.top() << ' ';
        if (++cnt % 10 == 0) cout << endl;
      }
    }
    if (cnt % 10) cout << endl;
  }
}
```

11.6.4 Productive Meeting

原题指路:https://codeforces.com/problemset/problem/1579/D

题意 (2 s)

给定一个长度为n的非负整数序列$a_1,\cdots,a_n$.现有操作:选择两个正数,将它们都-1.求最大操作次数.

有t $(1\le t\le 1000)$组测试数据.每组测试数据第一行输入一个整数n $(2\le n\le 2\text{e}5)$.第二行输入n个整数$a_1,\cdots,a_n$ $(0\le a_i\le 2\text{e}5)$.数据保证所有测试数据的n之和不超过$2\text{e}5$,所有a_i之和不超过$2\text{e}5$.

思路

贪心策略:每次取最大和次大的正数进行操作,直至序列中不存在至少两个正数.

[**证**] 设$s=\sum_{i=1}^{n}a_i$.考察如下两种情况:

(1)若序列中的最大元素$\ge$其他元素之和,即$\exists a_i\ s.t.\ a_i\ge\sum_{j\ne i}a_j=s-a_i$时,

显然$ans\le s-a_i$且能取得最大值$s-a_i$.

(2)下证$ans\le\left\lfloor\frac{s}{2}\right\rfloor$.若不然,因每次操作需要令两个正数$-1$,则$s\ge 2\left(\left\lfloor\frac{s}{2}\right\rfloor+1\right)>s$,矛盾.

下证根据上述贪心策略可取得最大值$\left\lfloor\frac{s}{2}\right\rfloor$或$\left\lfloor\frac{s-1}{2}\right\rfloor$,考察最后一次操作后序列的情况.

①若序列所有元素都为0,则进行了$\left\lfloor\frac{s}{2}\right\rfloor$次操作.

②若序列有一个元素为1,则进行了$\left\lfloor\frac{s-1}{2}\right\rfloor$次操作.

②若序列有且仅有一个元素$a_i>1$,则该元素参与之前的所有操作,

这是因为每次取序列中最大和次大的正数进行操作,而$a_i\ge 2$,则序列中至多存在一个值为1的元素,

进而该情况为(1).

代码

```cpp
void solve() {
  int n; cin >> n;
  priority_queue<pair<int, int>> heap;
  for (int i = 1; i <= n; i++) {
    int x; cin >> x;
    heap.push({ x,i });
  }

  vector<pair<int, int>> ans;
  while (heap.top().first) {
    auto [a1, idx1] = heap.top(); heap.pop();
    auto [a2, idx2] = heap.top(); heap.pop();
    if (!a2) break;

    ans.push_back({ idx1,idx2 });
    heap.push({ a1 - 1,idx1 }), heap.push({ a2 - 1,idx2 });
  }

  cout << ans.size() << endl;
```

```
  for (auto [u, v] : ans) cout << u << ' ' << v << endl;
}

int main() {
  CaseT
    solve();
}
```

11.6.5 Merge Equals

原题指路: https://codeforces.com/problemset/problem/962/D

题意 (2 s)

给定一个长度为n $(2 \le n \le 15000)$的序列$a = [a_1, \cdots, a_n]$ $(1 \le a_i \le 1e9)$. 现重复下述操作, 直至序列不存在至少出现两次的元素: 选择当前序列中最小的至少出现两次元素x, 设其第一、二次出现的下标分别为fir、sec, 删除$a[fir]$, 令$a[sec] * = 2$. 求上述过程结束后的序列.

思路

用元素为pair<int, int>的小根堆维护序列中的元素, 其中first为元素值, second为元素下标, 注意first要开long long. 模拟上述过程即可.

代码

```
typedef pair<ll, int> pli;

void solve() {
    int n; cin >> n;
    priority_queue<pli, vector<pli>, greater<pli>> heap;  // val, idx
    for (int i = 1; i <= n; i++) {
        int a; cin >> a;
        heap.push({ a, i });
    }

    vector<pair<int, ll>> ans;  // idx, val
    while (heap.size() >= 2) {
        auto [a, fir] = heap.top(); heap.pop();
        auto [b, sec] = heap.top();
        if (a == b) {
            heap.pop();
            heap.push({ a + b, sec });
        }
        else ans.push_back({ fir, a });
    }
    ans.push_back({ heap.top().second, heap.top().first });  // 最后一个元素
    sort(all(ans));

    cout << ans.size() << endl;
    for (auto [_, ai] : ans)
        cout << ai << " \n"[ai == ans.back().second];
}

int main() {
    solve();
```

```
31 }
```