

栈和队列

第3章

3.1 实战题解析

【实战 3.1】 POJ1363——铁轨问题

时间限制：1000ms；内存限制：65536KB。

问题描述：A 市有一个著名的火车站，那里的山非常多。该站建于 20 世纪，不乐观的是该火车站是一个死胡同，并且只有一条铁轨，如图 3.1 所示。

当地的传统是从 A 方向到达的每列火车都继续沿 B 方向行驶，需要以某种方式进行车厢重组。假设从 A 方向到达的火车有 n ($n \leq 1000$) 个车厢，按照递增的顺序 $1, 2, \dots, n$ 编号。火车站负责人必须知道是否可以通过重组得到 B 方向的车厢序列。

输入格式：输入由若干块组成。除了最后一个块之外，每个块描述了一个列车以及可能更多的车厢重组要求。在每个块的第一行中为上述的整数 n ，下一行是 $1, 2, \dots, n$ 的车厢重组序列。每个块的最后一行仅包含 0。最后一个块只包含一行 0。

输出格式：输出包含与输入中具有车厢重组序列对应的行。如果可以得到车厢

对应的重组序列，输出一行“Yes”，否则输出一行“No”。此外，在输入的每个块之后有一个空行。

输入样例：

```
5
1 2 3 4 5
5 4 1 2 3
0
6
```

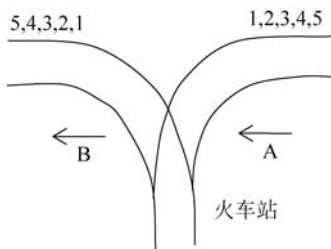


图 3.1 铁轨问题示意图



视频讲解

```

6 5 4 3 2 1
0
0

```

输出样例:

```

Yes
No

Yes

```

解: 先考虑这样的情况,假设 a 和 b 数组中均含 n 个整数(n 为大于 1 的正整数),都是 $1 \sim n$ 的某个排列,现在要判断 b 是否为以 a 作为输入序列的一个合法出栈序列,即 a 序列经过一个栈是否得到 b 的出栈序列。

求解思路是,建立一个整型栈 st ,用 i, j 分别遍历 a, b 序列(初始值均为 0),在 a 序列没有遍历完时循环:

① 将 $a[i]$ 进栈, $i++$ 。

② 栈不空并且栈顶元素与 $b[j]$ 相同时循环: 出栈元素 $e, j++$ 。

在上述过程结束后,如果栈空返回 `true`,表示 b 序列是 a 序列的出栈序列,否则返回 `false`,表示 b 序列不是 a 序列的出栈序列。

例如, $a=[1,2,3,4,5], b=[3,2,1,5,4], i=0, j=0$,判断过程如下:

① 初始时栈空, $a[0]$ 进栈($i=1$)。 $b[0] \neq$ 栈顶元素, $a[1]$ 进栈($i=2$)。 $b[0] \neq$ 栈顶元素, $a[2]$ 进栈($i=3$),如图 3.2(a)所示。

② $b[0]=$ 栈顶元素,出栈一次, j 增 1($j=1$)。 $b[1]=$ 栈顶元素,出栈一次, j 增 1($j=2$)。 $b[2]=$ 栈顶元素,出栈一次, j 增 1($j=3$),如图 3.2(b)所示。

③ 此时栈空, $a[3]$ 进栈($i=4$)。 $b[3] \neq$ 栈顶元素, $a[4]$ 进栈($i=5$, a 序列遍历完毕),如图 3.2(c)所示。

④ $b[3]=$ 栈顶元素,出栈一次, j 增 1($j=4$)。 $b[4]=$ 栈顶元素,出栈一次, j 增 1($j=5$),如图 3.2(d)所示。

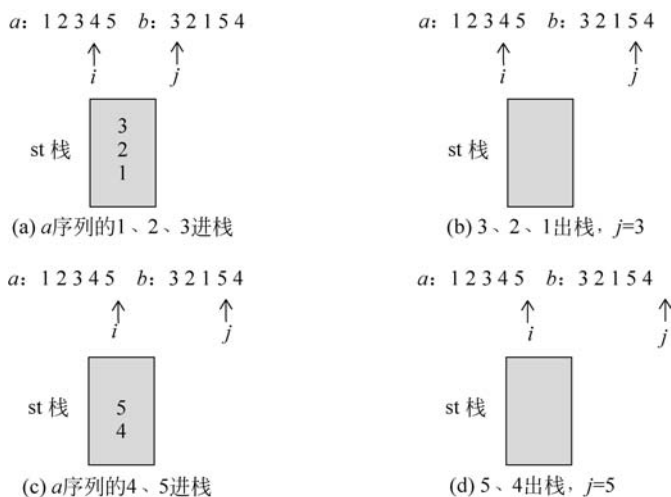


图 3.2 判断 $b=[3,2,1,5,4]$ 是否为出栈序列的过程

此时 a 序列遍历完毕, 栈空返回 true, 表示 b 序列是 a 序列的出栈序列。

又例如, $a=[1,2,3], b=[3,1,2], i=0, j=0$, 判断过程如下:

① 初始时栈空, $a[0]$ 进栈 ($i=1$)。 $b[0] \neq$ 栈顶元素, $a[1]$ 进栈 ($i=2$)。 $b[0] \neq$ 栈顶元素, $a[2]$ 进栈 ($i=3, a$ 序列遍历完毕), 如图 3.3(a) 所示。

② $b[0]=$ 栈顶元素, 出栈一次, j 增 1 ($j=1$)。 $b[1] \neq$ 栈顶元素, 如图 3.3(b) 所示。

此时 a 序列遍历完毕, 栈不空返回 false, 表示 b 序列不是 a 序列的出栈序列。



图 3.3 判断 $b=[3,1,2]$ 是否为出栈序列的过程

再回到本题, 实际上就是有多多个测试用例, 每个测试用例给一个正整数 n 和若干由 $1 \sim n$ 排列构成的 b 序列, 判断 $1, 2, \dots, n$ 作为输入序列经过一个栈是否得到 b 序列。采用上述思路, 直接将 a 数组用 $1, 2, \dots, n$ 替代, 对应的完整程序如下:

```
#include <iostream>
#include <stack>
using namespace std;
const int MAXN=1005;
bool isSerial(int b[], int n)                //判断 b 序列是否为 1~n 的出栈序列
{ stack<int> st;                             //建立一个栈对象
  int i=1, j=0;
  while (i <= n)                             //输入序列没有遍历完
  { st.push(i);
    i++;
    while (!st.empty() && st.top() == b[j])
    { st.pop();                               //b[j] 与栈顶匹配时出栈
      j++;
    }
  }
  return st.empty();                         //栈空返回 true, 否则返回 false
}

int main()
{ int n;
  int b[MAXN];
  while (cin >> n && n)                     //n=0 时输入结束
  { while(true)                             //处理一个块
    { cin >> b[0];
      if (b[0] == 0)                         //一个块结束
      { cout << endl;                       //输出一个空行
        break;
      }
    }
  }
}
```

```

else
{ for (int i=1;i<n;i++)          //输入一个 b 序列的其他 n-1 个整数
  cin >> b[i];
}
if (isSerial(b,n))
  cout << "Yes" << endl;
else
  cout << "No" << endl;
}
}
}

```

上述程序是 AC 代码,运行时间为 176ms,运行占用的空间为 391KB,编译语言为 C++。

【实战 3.2】 POJ1208——箱子操作

问题描述参见本书第 2 章中的实战 2.4,这里改为用栈求解。

解:如图 3.4(a)所示为 $n=4$ 的某个状态表示,box[1]表示位置 1 的箱子链,实际上箱子链应该是箱子栈,上方是栈顶,当执行 move 0 onto 2 命令时,将箱子 0 和 2 上面的箱子(只有箱子 3)放回初始位置,并将 0 放到 2 上,结果如图 3.4(b)所示。

为此将 box 数组元素由链表改为栈,即 box[i]表示位置 i 的箱子栈,所有操作执行完毕,借助 vector<int>向量输出每个栈中从栈底到栈顶的所有元素即可。

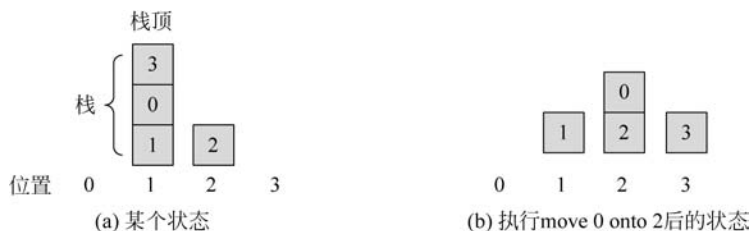


图 3.4 执行 move 0 onto 2 操作命令

对应的程序如下:

```

#include <iostream>
#include <vector>
#include <string>
#include <stack>
using namespace std;
const int MAXN=30;
stack<int> box[MAXN];          //box[i]表示位置 i 的箱子栈
stack<int> tmp;                //临时栈
int pos[MAXN];                //pos[i]表示编号为 i 的箱子的位置
void restore(int x)            //将 x 上面的箱子放回初始位置
{ int i=pos[x];                //找 x 所在的位置 i
  while(box[i].top()!=x)        //在位置 i 的箱子栈中从尾部找到 x 为止
  { int t=box[i].top();          //取出箱子 t
    box[t].push(t);              //将箱子 t 添加到位置 t 的箱子栈中
    pos[t]=t;                    //设置箱子 t 的位置为 t
    box[i].pop();                //将箱子 t 从位置 i 的箱子栈中删除
  }
}

```



视频讲解

```

}
void over(int x,int y)                                //将箱子 x 及其上方的箱子放到箱子 y 所在的箱子栈中
{ int i=pos[x];                                       //找 x 所在的位置 i
  int j=pos[y];                                       //找 y 所在的位置 j
  while(box[i].top()!=x)                               //在位置 i 的箱子栈中从尾部找到 x 为止
  { tmp.push(box[i].top());                           //将箱子 x 上方的箱子退出并存放在临时 tmp 中
    box[i].pop();
  }
  box[j].push(box[i].top());                           //将箱子 x 放在位置 j 的箱子栈中
  pos[x]=j;                                           //置箱子 x 的位置为 j
  box[i].pop();                                       //从位置 i 的箱子栈中删除 x
  while(!tmp.empty())                                //将 tmp 中的箱子放在 x 的上方
  { box[j].push(tmp.top());
    pos[tmp.top()]=j;
    tmp.pop();
  }
}
int main()
{ int n;
  cin >> n;
  for(int i=0;i<n;i++)                                //将箱子 i 放在位置 i
  { box[i].push(i);
    pos[i]=i;
  }
  string str1,str2;
  int a,b;
  while(cin >> str1 && str1!="quit" )
  { cin >> a >> str2 >> b;
    if(str1=="move" && str2=="onto") //move a onto b
    { restore(a);
      restore(b);
      over(a,b);
    }
    else if(str1=="move" && str2=="over") //move a over b
    { restore(a);
      over(a,b);
    }
    else if(str1=="pile" && str2=="onto") //pile a onto b
    { restore(b);
      over(a,b);
    }
    else //pile a over b
      over(a,b);
  }
  vector<int> ans;
  vector<int>::reverse_iterator rit;
  for(int i=0;i<n;i++)                                //输出结果
  { cout << i << ' ';
    ans.clear();
    while(!box[i].empty())                            //将 box[i] 栈元素出栈并存放在 ans 中
    { ans.push_back(box[i].top());

```

```

        box[i].pop();
    }
    for (rit=ans.rbegin(); rit!=ans.rend(); rit++)
        cout << " " << *rit;           //反向输出 ans 构成该箱子栈的结果
    cout << endl;
}
return 0;
}

```

上述程序是 AC 代码,运行时间为 16ms,运行占用的空间为 220KB,编译语言为 C++。从中看出,由于 vector 和 list 容器都提供了快捷的尾端操作 push_back/pop_back,用它们模拟 stack 十分方便,实际上实战 2.4 就是用 list 容器模拟栈来求解的。

【实战 3.3】 LeetCode225——用队列实现栈

问题描述:使用队列实现栈的下列操作。

void push(int x): 元素 x 进栈。

int pop(): 删除并且返回栈顶元素。

int top(): 返回栈顶元素。

bool empty(): 返回栈是否为空。

解: 将一个队列作为栈的示意图如图 3.5 所示,以 queue<int>容器 qu 为队列,模拟栈的操作如下。

void push(int x): 元素 x 进栈和 qu 进队的操作一致,直接用 qu.push(x)实现。

int pop(): 出栈的元素为 a_{n-1} ,将 qu 中的前 $n-1$ 个元素出队并立刻进队,再出队列元素 a_{n-1} 并返回该元素。

int top(): 类似 pop(),但第 n 个元素 d 仍然需要进队,返回 d 。

bool empty(): 返回队列 qu 是否为空。

对应的程序代码如下:

```

class MyStack
{
    queue<int> qu;
public:
    MyStack(){}           //构造函数
    void push(int x)       //元素 z 进栈
    {
        qu.push(x);
    }
    int pop()              //删除并且返回栈顶元素
    {
        int m=qu.size()-1,d;
        for(int i=0;i<m;i++)
        {
            d=qu.front();
            qu.pop();
            qu.push(d);
        }
    }
}

```

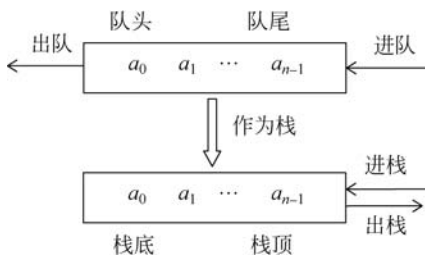


图 3.5 将一个队列作为栈



视频讲解

```

        d=qu.front();
        qu.pop();
        return d;
    }

    int top()                //获取栈顶元素
    { int n=qu.size(),d;
      for(int i=0;i<n;i++)
      { d=qu.front();
        qu.pop();
        qu.push(d);
      }
      return d;
    }

    bool empty()            //返回栈是否空
    {
        return qu.empty();
    }
};

```

上述程序是 AC 代码,运行时间为 4ms,运行占用的空间为 8.6MB,编译语言为 C++。

【实战 3.4】 HDU1276——士兵队列训练问题

时间限制: 1000ms; 内存限制: 32768KB。

问题描述: 某部队进行新兵队列训练,将新兵从 1 开始按顺序依次编号,并排成一行横队。训练的规则如下: 从头开始按 1~2 报数,凡报到 2 的出列,剩下的向小序号方向靠拢; 再从头开始按 1~3 报数,凡报到 3 的出列,剩下的向小序号方向靠拢; 继续从头开始按 1~2 报数,……; 之后从头开始轮流按 1~2 报数、按 1~3 报数,直到剩下的人数不超过 3 个为止。

输入格式: 本题有多个测试数据组,第一行为组数 n ,接着为 n 行新兵人数,新兵人数不超过 5000。

输出格式: 共有 n 行,分别对应输入的新兵人数,每行输出剩下的新兵的最初编号,编号之间有一个空格。

输入样例:

```

2
20
40

```

输出样例:

```

1 7 19
1 19 37

```

解法 1: 本题实际上是约瑟夫问题的变形,与 (n, m) 约瑟夫问题相比,一方面 m 是可变的(这里轮流做 $m=2$ 和 $m=3$ 的报数),另外每次报数都是从头开始。其求解思路与《教程》中的例 3.13 类似,这里用 queue 容器代替队列。对应的程序如下:

```

#include <iostream>
#include <queue>
using namespace std;

```



视频讲解

```
//求解算法

//m 个士兵排成一行

//是否做 1~2 报数
//剩余人数超过 3 时循环
//1~2 报数
//剩余人数

//处理前面满两人的组

//每组中第 2 个人只出不进

//处理最后一人的组
//这些人都保留

//做 1~3 报数
//剩余人数

//处理前面满 3 人的组

//每组中第 3 个人只出不进

//处理最后一或者两人的组
//这些人都保留

//输出最后剩下的士兵

//输入测试用例的个数
```

```

{ scanf("%d",&n);           //输入一行的人数 n
  solve(n);                 //求解一个测试用例的问题
}
return 0;
}

```

上述程序是 AC 代码,运行时间为 15ms,运行占用的空间为 1840KB,编译语言为 C++。

解法 2: 直接采用链表模拟,这里采用 `list<int>` 链表容器 `lst` 存放一行的 n 个新兵(编号为 $1 \sim n$),依次报数并删除出列的新兵,直到剩下的人数不超过 3 人为止,再输出 `lst` 中的所有新兵。

参考博客: <https://www.cnblogs.com/randy-lo/p/12607874.html>

对应的程序如下:

```

#include <iostream>
#include <list>
using namespace std;
void solve(int n)           //求解算法
{ list<int> lst;
  list<int>::iterator it;
  for (int i=1;i<=n;i++)    //m 个士兵排成一行
    lst.push_back(i);
  int m=2;                 //报数的人数
  while (lst.size()>3)
  { int i=1;
    it=lst.begin();
    while (it!=lst.end())
    { if (!(i++%m))
      it=lst.erase(it);    //删除之后返回下一个位置的指针
      else
        i++;
    }
    if (m==2) m=3;
    else m=2;
  }
  it=lst.begin();          //输出最后剩下的士兵
  while (it!=lst.end())
  { if (it==lst.begin())
    printf("%d", *it);
    else
      printf(" %d", *it);
    i++;
  }
  printf("\n");
}
int main()
{ int t,n;
  scanf("%d",&t);          //输入测试用例的个数
  while (t--)
  { scanf("%d",&n);        //输入一行的人数 n
    solve(n);              //求解一个测试用例的问题
  }
}

```

```

    }
    return 0;
}

```

上述程序是 AC 代码,运行时间为 15ms,运行占用的空间为 1876KB,编译语言为 C++。

【实战 3.5】 LeetCode84——柱状图中最大的矩形

问题描述: 给定 n 个非负整数,用来表示柱状图中各个柱子的高度。每个柱子彼此相邻,且宽度为 1,求在该柱状图中能够勾勒出来的矩形的最大面积。例如,如图 3.6(a)所示的柱状图,其中每个柱子的宽度为 1,给定的高度 $\text{heights}[] = \{2, 1, 5, 6, 2, 3\}$,求出的最大面积的矩形如图 3.6(b)所示,其面积为 10。要求设计相应的 $\text{largestRectangleArea}()$ 函数:

```

class Solution
{
public:
    int largestRectangleArea(vector<int> & heights)
    {
        ...
    }
};

```

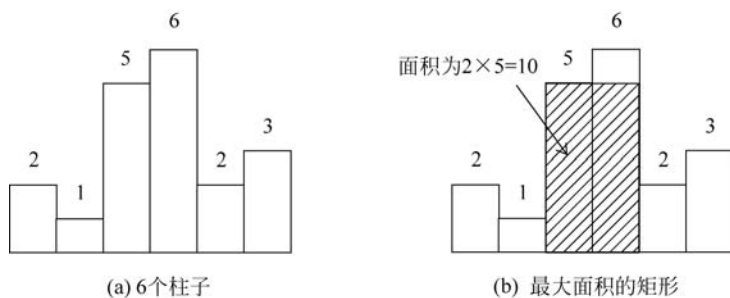


图 3.6 一个柱状图及其求解结果

解法 1: 采用枚举法,对于 $\text{heights}[i..j]$ ($j \geq i$) 的 $j-i+1$ 个柱子,构成的矩形面积为 $(\text{heights}[i..j]$ 中的最小高度) $\times (j-i+1)$ 。在所有这些矩形面积中比较求最大值。对应的程序如下:

```

class Solution
{
public:
    int largestRectangleArea(vector<int> & heights)
    {
        if (heights.size() == 0) return 0; //处理两种特殊情况
        if (heights.size() == 1) return heights[0];
        int ans = 0; //存放最大矩形面积
        for (int j = 0; j < heights.size(); j++) //j 从 0 到 n-1 遍历
        {
            int minh = heights[j];
            for (int i = j; i >= 0; i--) //i 从 0 到 j 反向遍历
            {
                if (heights[i] < minh) //求 heights[i..j] 中的最小值 minh
                    minh = heights[i];
                int s = (j-i+1) * minh; //求每个 heights[i..j] 的面积
                if (ans < s) ans = s; //比较求最大面积
            }
        }
    }
}

```

```

    }
    return ans;
}
};

```

上述程序是 AC 代码,运行时间为 1916ms,运行占用的空间为 9.2MB,编译语言为 C++。显然该解法是低效的。

解法 2: 采用单调栈求解。设 $\text{maxr}([i..j])$ 表示 $\text{heights}[i..j]$ 中的最大矩形面积,假设 $\text{heights}[i..j]$ 中所有柱子的高度是递增的,采用上述解法需要通过枚举 i, j 来求最大的矩形面积,实际上可以利用所有柱子的宽度为 1 并且高度递增推出 $\text{maxr}([i..i]) < \dots < \text{maxr}([i..j-1]) < \text{maxr}([i..j])$ 。当 $\text{heights}[j+1] > \text{heights}[j]$ 时一定有 $\text{maxr}([i..j]) < \text{maxr}([i..j+1])$, 所以没有必要求前面的 $\text{maxr}([i..i]), \dots, \text{maxr}([i..j-1]), \text{maxr}([i..j])$, 只有当 $\text{heights}[j+1] > \text{heights}[j]$ 不成立时再求这些面积,从中比较求出最大矩形面积。这样减少了枚举 i 的次数,从而提高了效率。

为此用一个栈 st 来维护高度递增的柱子序列(即单调递增栈),栈中存储这些柱子的下标(索引)而不是柱子的高度,每个柱子的下标仅进栈一次,从栈底到栈顶柱子的高度是递增的。用 j 从 0 开始遍历 heights : 当栈空或者柱子 j 的高度大于或等于栈顶柱子的高度时,将 j 进栈并且执行 $j++$,即维护栈中柱子高度的递增性,否则对栈中所有高度大于 $\text{heights}[j]$ 的柱子求矩形面积 S 。计算过程是退栈柱子 tmp ,求该柱子到 $j-1$ 位置为止构成的最大矩形面积,显然其高度是 $\text{heights}[tmp]$,那么最大矩形的长度 $length$ 呢? 分为两种情况:

① 退栈 tmp 后栈不空, $st.top()$ 为新栈顶柱子,求以柱子 $j-1$ 结尾、以 tmp 为最小高度的矩形 S 的面积,按照单调栈的操作有两种情况,情况 1 是 $st.top() = tmp-1$ (即 tmp 柱子和栈顶 $st.top()$ 柱子是相邻的,并且 $\text{heights}[st.top()] < \text{heights}[tmp]$), 矩形 S 是由柱子 tmp 到柱子 $j-1$ 构成的,其面积为 $\text{heights}[tmp] \times (j - tmp) = \text{heights}[tmp] \times (j - st.top() - 1)$; 情况 2 是 $st.top() < tmp-1$ (即柱子 $st.top()+1$ 到柱子 $tmp-1$ 的所有柱子在 tmp 出栈之前已经出栈,说明它们的高度都大于柱子 tmp 的高度,需要计入矩形 S 中), 矩形 S 是由柱子 $st.top()+1$ 到柱子 $j-1$ 构成的,其面积为 $\text{heights}[tmp] \times (j - st.top() - 1)$ 。合并起来矩形 S 的面积 $area = \text{heights}[tmp] \times (j - st.top() - 1)$, 求宽度 $length = j - st.top() - 1$ 。

② 退栈后栈空,说明在所有 $\text{heights}[0..j-1]$ 中 tmp 柱子的高度最小,宽度 $length = j$, 其面积 $area = \text{heights}[tmp] \times j$ 。

对于每次求出的 $area$,再求 $ans = \max(ans, area)$ 。当所有 heights 序列遍历完毕,需要对栈中的全部柱子求矩形面积,为了触发这个计算过程,先在 heights 末尾添加一个高度为 0 的虚柱子。

例如,题目中的样例在末尾添加 0 后为 $\text{heights}[] = \{2, 1, 5, 6, 2, 3, 0\}$,用 ans 存放最大矩形面积(初始为 0),求解过程如下:

① $j=0$,柱子 0 的高度为 2,直接进栈,如图 3.7(a)所示,栈中元素“2[0]”表示该柱子的下标为 0,高度为 2。

② $j=1$,柱子 1 的高度为 1,其高度小于栈顶柱子的高度,计算当前最大面积,出栈

$tmp=0$, 栈为空, 说明柱子 2 是 $heights[0..j-1]$ 中高度最小的, 长度 $length=j=1$, 求出面积 $area=heights[0] \times length=2 \times 1=2$, 则 $ans=2$ 。再将 j 进栈, 如图 3.7(b) 所示。

③ $j=2$, 柱子 2 的高度为 5, 其高度大于栈顶柱子的高度, 直接进栈。 $j=3$, 柱子 3 的高度为 6, 其高度大于栈顶柱子的高度, 直接进栈, 结果如图 3.7(c) 所示。

④ $j=4$, 柱子 4 的高度为 2, 其高度小于栈顶柱子的高度, 计算当前最大面积, 出栈 $tmp=3$, 栈不空, 说明新栈顶之后一直到 $j-1$ 所有柱子中 tmp 柱子的高度最小, 长度 $length=j-st.top()-1=4-2-1=1$, 求出面积 $area=heights[3] \times length=6 \times 1=6$, 则 $ans=6$ 。柱子 4 的高度仍小于栈顶柱子的高度, 再计算当前最大面积, 出栈 $tmp=2$, 栈不空, 计算长度 $length=j-st.top()-1=4-1-1=2$, 求出面积 $area=heights[2] \times length=5 \times 2=10$, 则 $ans=10$ 。再将 j 进栈, 如图 3.7(d) 所示。

⑤ $j=5$, 柱子 5 的高度为 3, 其高度大于栈顶柱子的高度, 直接进栈, 如图 3.7(e) 所示。

⑥ $j=6$, 柱子 6 的高度为 0, 其高度小于栈顶柱子的高度, 计算当前最大面积, 出栈 $tmp=5$, 栈不空, 计算长度 $length=j-st.top()-1=6-4-1=1$, 求出面积 $area=heights[5] \times length=3 \times 1=3$, $ans=10$ 。

柱子 6 的高度仍小于栈顶柱子的高度, 再计算当前最大面积, 出栈 $tmp=4$, 栈不空, 说明新栈顶之后一直到 $j-1$ 所有柱子中 tmp 柱子的高度最小, 长度 $length=j-st.top()-1=6-1-1=4$ (主要此时是 $heights[2..5]$ 计算面积), 求出面积 $area=heights[4] \times length=2 \times 4=8$, $ans=10$ 。

柱子 6 的高度仍小于栈顶柱子的高度, 再计算当前最大面积, 出栈 $tmp=1$, 栈空, 计算长度 $length=j=6$, 求出面积 $area=heights[1] \times length=1 \times 6=6$, $ans=10$ 。再将 j 进栈, 如图 3.7(f) 所示。

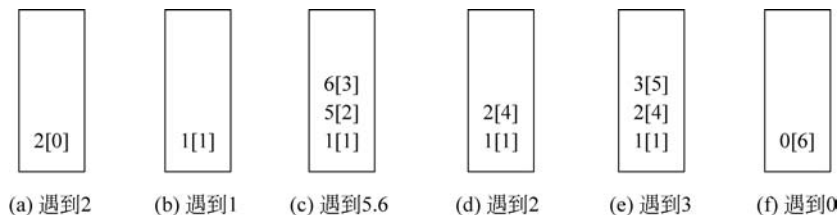


图 3.7 {2,1,5,6,2,3,0} 的求解过程

对应的程序如下:

```
class Solution
{
public:
    int largestRectangleArea(vector<int> & heights)
    { if (heights.size()==0) return 0;           //处理两种特殊情况
      if (heights.size()==1) return heights[0];
      heights.push_back(0);                     //在末尾添加一个高度为0的虚柱子
      int n=heights.size();
      stack<int> st;
      int ans=0;                                //存放最大矩形面积,初始为0
      for (int j=0;j<n;j++)
      { if (st.empty() || heights[j]>heights[st.top()])
```

```

        st.push(j);                                //栈空或高度是单调递增的,直接进栈
    else
    { while (!st.empty() && heights[st.top()]>=heights[j])
        { int tmp=st.top(); st.pop();              //退栈 tmp
          int length;
          if(st.empty())
              length=j;
          else
              length=j-st.top()-1;
          int area=heights[tmp]*length;             //求 heights[st.top()+1..j-1]的矩形面积
          ans=max(ans,area);                        //维护 ans 为最大矩形面积
        }
        st.push(j);                                //j 进栈
    }
}
return ans;
};

```

上述程序是 AC 代码,运行时间为 12ms,运行占用的空间为 15.5MB,编译语言为 C++。

【实战 3.6】 POJ2823——滑动窗口

时间限制: 12000ms; 内存限制: 65536KB。

问题描述: 给出一个长度为 $n \leq 10^6$ 的整数数组。有一个大小为 k 的滑动窗口,它从数组的最左边移动到最右边,用户只能在窗口中看到 k 个整数。每次滑动窗口向右移动一个位置。在如表 3.1 所示的例子中,该数组为 $\{1, 3, -1, -3, 5, 3, 6, 7\}$, k 为 3。



视频讲解

表 3.1 一个 $k=3$ 的滑动窗口例子

窗口位置	最小值	最大值
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7

确定每个位置的滑动窗口中的最大值和最小值。

输入格式: 输入包含两行,第一行包含两个整数 n 和 k ,它们是数组和滑动窗口的长度,第二行有 n 个整数。

输出格式: 输出中有两行,第一行按从左到右的顺序给出每个位置的窗口中的最小值,第二行为对应的最大值。

输入样例:

```

8 3
1 3 -1 -3 5 3 6 7

```

输出样例:

```

-1 -3 -3 -3 3 3
3 3 5 5 6 7

```

解：设计两个单调队列，minqu 为单调递增队列，用于维护序列的最小值（队头为最小元素），maxqu 为单调递减队列，用于维护序列的最大值（队头为最大元素）。队列中的每个元素记录对应的整数 x 和输入的编号 i 。

由于本题是求 k 区间内的最大值和最小值，以使用 minqu 求最小值序列为例，对于当前整数 x ，只关心它自己和它前面 $k-1$ 个整数。 i 标记输入的整数 x 的编号，当 $i \leq k-1$ （处理前面的 $k-1$ 个整数）时不存在区间最小值，直接做进队处理：

① 将队尾 $\geq x$ 的元素从队尾出队。

② x 从队尾进队。

当 $i \geq k$ （处理第 k 个整数以及后面的整数）时存在区间最小值，此时除了做上述直接进队的处理外，还需要将队头下标超出 k 区间范围的元素出队，即将满足 $i - \text{minqu.front}() > k$ 的整数从队头出队，目的是保持队头元素为区间内的最小值，再置 $\text{mina}[i] = \text{minqu.front}()$ ， x 取队首元素。

求 k 区间内的最大值序列与上述过程类似，最后输出最小值序列 mina 和最大值序列 maxa。对应的程序如下：

```

#include <iostream>
#include <deque>
using namespace std;
const int MAXN=1000005;

struct Node                                //队列元素类型
{ int x;                                  //元素值
  int i;                                  //元素的编号
  Node(int x1,int i1):x(x1),i(i1) {}     //构造函数
};

deque<Node> minqu;                         //单调递增队列
deque<Node> maxqu;                         //单调递减队列
int mina[MAXN];                          //存放最小元素序列
int maxa[MAXN];                          //存放最大元素序列

int main()
{ int n,k,x;
  scanf("%d%d",&n,&k);
  for (int i=1;i<=n;i++)                  //按编号输入整数
  { scanf("%d",&x);                      //输入编号为 i 的整数 x
    while (!minqu.empty() && minqu.back().x >= x) //维护 minqu 为单调递增队列
      minqu.pop_back();                  //队尾较大元素出队
    minqu.push_back(Node(x,i));          //x 从队尾进队
    if(i >= k)
    { while (!minqu.empty() && i - minqu.front().i >= k) //把队头下标超出 k 区间范围
      //的元素出队
      minqu.pop_front();
      mina[i] = minqu.front().x;         //取队首元素
    }
    while (!maxqu.empty() && maxqu.back().x <= x) //维护 maxqu 为单调递减队列
      maxqu.pop_back();                 //队尾较小元素出队
  }
}

```

```

maxqu.push_back(Node(x,i));           //x 从队尾进队
if (i >= k)
{ while (!maxqu.empty() && i-maxqu.front().i >= k) //把队头下标超出 k 区间范围的
//元素出队

    maxqu.pop_front();
    maxa[i] = maxqu.front().x;           //取队首元素
}
}
for (int i=k;i<=n;i++)                 //输出最小元素序列
if (i==k)
    printf("%d",mina[i]);
else
    printf(" %d",mina[i]);
printf("\n");
for (int i=k;i<=n;i++)                 //输出最大元素序列
if (i==k)
    printf("%d",maxa[i]);
else
    printf(" %d",maxa[i]);
printf("\n");
}

```

上述程序是 AC 代码,运行时间为 9094ms,运行占用的空间为 9088KB,编译语言为 C++。

3.2 在线编程题解析

3.2.1 LeetCode150——逆波兰表达式求值



视频讲解

问题描述: 根据逆波兰表示法求表达式的值,有效的运算符包括+、-、*、/。每个运算对象可以是整数,也可以是另一个逆波兰表达式。假设给定的逆波兰表达式总是有效的,换句话说,表达式总会得出有效数值且不存在除数为 0 的情况。其中整数除法只保留整数部分。

示例 1:

输入: ["2", "1", "+", "3", "*"]

输出: 9

解释: $((2+1)*3)=9$

示例 2:

输入: ["4", "13", "5", "/", "+"]

输出: 6

解释: $(4+(13/5))=6$

示例 3:

输入: ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]

输出: 22

解释:

$((10*(6/((9+3)*-11)))+17)+5$

```

=((10 * (6/(12 * -11)))+17)+5
=((10 * (6/-132))+17)+5
=((10 * 0)+17)+5
=(0+17)+5
=17+5
=22

```

设计求逆波兰表达式 tokens 的值:

```

class Solution
{
public:
    int evalRPN(vector< string> & tokens)
    { ... }
};

```

解: 利用《教程》第3章中3.1.7节求后缀表达式值的思路求解。设计一个运算数栈st,遍历逆波兰表达式tokens,当遇到运算符时出栈两个运算数a和b,做相应运算并将结果进栈,若遇到数字串转换为整数并进栈,tokens处理完毕返回栈顶整数。对应的程序如下:

```

class Solution
{
public:
    int evalRPN(vector< string> & tokens)
    { int a, b;
      stack< int> st;
      for (int i=0; i< tokens.size(); i++)
      { string s=tokens[i];
        if (s=="+")
        { a=st.top(); st.pop(); //出栈整数 a
          b=st.top(); st.pop(); //出栈整数 b
          st.push(b+a); //执行 b+a 并将结果进栈
        }
        else if(s=="-")
        { a=st.top(); st.pop(); //出栈整数 a
          b=st.top(); st.pop(); //出栈整数 b
          st.push(b-a); //执行 b-a 并将结果进栈
        }
        else if(s=="*")
        { a=st.top(); st.pop(); //出栈整数 a
          b=st.top(); st.pop(); //出栈整数 b
          st.push(b * a); //执行 b * a 并将结果进栈
        }
        else if(s=="/")
        { a=st.top(); st.pop(); //出栈整数 a
          b=st.top(); st.pop(); //出栈整数 b
          st.push(b/a); //执行 b/a 并将结果进栈
        }
        else //遇到数字串,转换为整数后进栈
          st.push(stoi(s));
      }
    }
};

```

```

    }
    return st.top();
}
};

```

运行通过,执行用时为 4ms,内存消耗为 12.2MB,所用语言为 C++。

3.2.2 LeetCode622——设计循环队列



视频讲解

问题描述:设计循环队列的实现。循环队列是一种线性数据结构,其操作表现基于FIFO(先进先出)原则并且队尾被连接在队首之后,以形成一个循环,所以它也被称为“环形缓冲器”。循环队列的一个好处是用户可以利用这个队列之前用过的空间。在一个普通队列中,一旦一个队列满了就不能插入下一个元素,即使在队列前面仍有空间,但是在使用循环队列时可以使用这些空间去存储新的值。该实现应该支持如下操作。

- ① MyCircularQueue(k): 构造器,设置队列长度为 k 。
- ② Front(): 从队首获取元素。如果队列为空,返回-1。
- ③ Rear(): 获取队尾元素。如果队列为空,返回-1。
- ④ enqueue(value): 向循环队列中插入一个元素。如果成功插入,则返回真。
- ⑤ dequeue(): 从循环队列中删除一个元素。如果成功删除,则返回真。
- ⑥ isEmpty(): 检查循环队列是否为空。
- ⑦ isFull(): 检查循环队列是否已满。

例如:

```

MyCircularQueue circularQueue = new MyCircularQueue(3); //设置长度为 3
circularQueue.enqueue(1); //返回 true
circularQueue.enqueue(2); //返回 true
circularQueue.enqueue(3); //返回 true
circularQueue.enqueue(4); //返回 false, 队列已满
circularQueue.Rear(); //返回 3
circularQueue.isFull(); //返回 true
circularQueue.dequeue(); //返回 true
circularQueue.enqueue(4); //返回 true
circularQueue.Rear(); //返回 4

```

提示:所有的值都在 0~1000,操作数将在 1~1000,不要使用内置的队列库。

解:循环队列的原理参见《教程》第3章中的3.2.2节,为了简单,在这里循环队列中除了data数组、队头指针front和队尾指针rear外,增加了容量capacity和长度length数据成员,同样让front指向队头元素的前一个位置,rear指向队尾元素的位置。对应的程序如下:

```

class MyCircularQueue
{
public:
    int * data; //存放队中元素
    int front; //队头指针
    int rear; //队尾指针
    int capacity; //容量

```

```

    int length;                                //长度
public:
    MyCircularQueue(int k)                    //构造函数
    {
        data = new int[k];
        capacity = k;
        front = rear = 0;
        length = 0;
    }
    ~MyCircularQueue()                        //析构函数
    {
        delete [] data;
    }
    bool isEmpty()                            //判断是否为空队
    {
        return length == 0;
    }
    bool isFull()                             //判断是否队满
    {
        return length == capacity;
    }
    bool enqueue(int value)                   //元素 value 进队
    {
        if (isFull()) return false;
        rear = (rear + 1) % capacity;
        data[rear] = value;
        length++;
        return true;
    }
    bool dequeue()                            //出队一个元素
    {
        if (isEmpty()) return false;
        front = (front + 1) % capacity;
        length--;
        return true;
    }
    int Front()                              //返回队头元素
    {
        if (isEmpty()) return -1;
        int head = (front + 1) % capacity;
        return data[head];
    }
    int Rear()                               //返回队尾元素
    {
        if (isEmpty()) return -1;
        return data[rear];
    }
};

```

运行通过,执行用时为 28ms,内存消耗为 17.1MB,所用语言为 C++。

3.2.3 HDU5818——合并栈操作

时间限制: 4000ms; 内存限制: 65536KB。

问题描述: 栈是一种后进先出的数据结构,可合并栈是支持“merge”操作的栈。3 种操



视频讲解

作的说明如下。

- ① push A x : 将 x 插入栈 A 中。
- ② pop A: 删除栈 A 的顶部元素。
- ③ merge A B: 合并栈 A 和 B。

其中,“merge A B”操作后栈 A 包含 A 和 B 之前的所有元素,B 变为空,新栈中的元素根据先前的进栈时间重新排列,就像在一个栈中重复“push”操作一样。给定两个可合并栈 A 和 B,请执行上述操作。

输入格式: 有多个测试用例,测试用例的第一行包含一个整数 n ($0 < n \leq 10^5$) 表示操作个数,接下来的 n 行每行包含一条指令 push、pop 或 merge,栈元素是 32 位整数。A 和 B 最初都是空的,并且保证不会对空栈执行 pop 操作。以 $n=0$ 表示输入结束。

输出格式: 对于每个测试用例,先输出一行“Case # t :”,其中 t 是测试用例编号(从 1 开始),对于每个“pop”操作,在一行中输出对应的出栈元素。

输入样例:

```
4
push A 1
push A 2
pop A
pop A
9
push A 0
push A 1
push B 3
pop A
push A 2
merge A B
pop A
pop A
pop A
9
push A 0
push A 1
push B 3
pop A
push A 2
merge B A
pop B
pop B
pop B
0
```

输出样例:

```
Case #1:
2
1
Case #2:
1
```

```

2
3
0
Case #3:
1
2
3
0

```

解法 1: 本题表面上是栈操作,由于“merge”操作产生新栈,其中元素是根据先前的进栈时间重新排列的,所以每个栈元素增加一个时间戳 timeid 表示进栈时间,这样的栈可以看成按 timeid 越大越优先出栈的优先队列(体现后进先出的特性)。这样设计 3 个优先队列,A 和 B 为题目中的栈,C 为公共栈,存放合并后的结果。栈元素采用 STL 内置 pair 结构体,两个成员分别为 timeid 和进栈元素,默认的 operator 按 timeid 越大越优先。

① push A/B x: 将 x 插入 A/B 优先队列(具有栈特性)。

② pop A/B: 先从 A/B 中去找,如果 A/B 为空,则说明存放在公共栈 C 中了(题目保证不会对空栈 pop)。

③ merge A B: 把当前 A 和 B 全部清空并放到公共栈 C 中即可。

对应的程序如下:

```

#include <iostream>
#include <queue>
using namespace std;
typedef pair<int,int> PA;
priority_queue<PA> A,B,C;
int main()
{ int n,t=1,x;
  PA e;
  char com[10],g; //com 存放命令
  while(scanf("%d",&n) != EOF && n)
  { printf("Case # %d:\n",t++);
    while(!A.empty()) A.pop(); //初始化 3 个优先队列
    while(!B.empty()) B.pop();
    while(!C.empty()) C.pop();
    int timeid=0; //表示时间戳
    while(n-->0)
    { scanf("%s %c",com,&g);
      if(com[1] == 'u') //push A x 命令
      { scanf("%d",&x);
        if(g == 'A')
          A.push(make_pair(timeid++,x));
        else
          B.push(make_pair(timeid++,x));
      }
      else if(com[1] == 'o') //pop A
      { if(g == 'A')
        { if(!A.empty()) //A 不空时从 A 出栈元素
          { e=A.top(); A.pop();

```

```

        printf("%d\n", e.second);
    }
    else //A 空时从公共栈 C 出栈元素
    { e=C.top(); C.pop();
      printf("%d\n", e.second);
    }
}
else //B 不空时从 B 出栈元素
{ if(!B.empty())
  { e=B.top(); B.pop();
    printf("%d\n", e.second);
  }
  else //B 空时从公共栈 C 出栈元素
  { e=C.top(); C.pop();
    printf("%d\n", e.second);
  }
}
}
else //merge A B
{ char tmp[10];
  gets(tmp); //忽略 B 参数,将 A 和 B 中的所有元素合并到 C 中
  while(!A.empty())
  { e=A.top(); A.pop();
    C.push(e);
  }
  while(!B.empty())
  { e=B.top(); B.pop();
    C.push(e);
  }
}
}
return 0;
}

```

运行通过,执行用时为 951ms,内存消耗为 2632KB,所用语言为 C++。

解法 2: 同样设置 3 个栈, A 和 B 为题目中的栈, C 为公共栈, 存放合并后的结果。每个栈采用数组表示, 例如 $A[0..lena-1]$ 存放栈 A 中的元素 (长度为 lena), $A[0]$ 为栈底, $A[lena-1]$ 为栈顶。由于元素进栈是按时间戳有序的, 所以每个栈元素包含元素值 x 和时间戳 timeid 数据成员, 每个栈从栈底到栈顶如 $A[0..lena-1]$ 按 timeid 递增有序, 在栈合并操作中采用二路归并将 $A[0..lena-1]$ 和 $B[0..lenb-1]$ 合并到 C 中 (如果之前 C 非空, $C[0..lenc-1]$ 一定有序, 并且它们的时间戳均早于 A 和 B 中所有元素的时间戳)。对应的程序如下:

```

#include <iostream>
using namespace std;
const int MAXN=100005;
struct Node //栈元素类型
{ int x; //元素值

```

```

int timeid; //时间戳
Node() {} //构造函数
Node(int x1, int t1):x(x1),timeid(t1) {} //重载构造函数
};
Node A[MAXN], B[MAXN], C[MAXN]; //全局变量: 3 个栈
int lena, lenb, lenc; //全局变量: 3 个栈的长度
int main()
{ int n, t=1, x;
  char com[10], g; //com 存放命令
  while(scanf("%d", &n) != EOF && n)
  { printf("Case # %d:\n", t++);
    lena=lenb=lenc=0; //初始化 3 个栈
    int timeid=0; //表示时间戳
    while(n-->0)
    { scanf("%s %c", com, &g);
      if(com[1]=='u') //push A x 命令
      { scanf("%d", &x);
        if(g=='A')
          A[lena++] = Node(x, timeid++);
        else
          B[lenb++] = Node(x, timeid++);
      }
      else if(com[1]=='o') //pop A
      { if(g=='A')
        { if(lena!=0) //A 不空时从 A 出栈元素
          { lena--;
            printf("%d\n", A[lena].x);
          }
          else //A 空时从公共栈 C 出栈元素
          { lenc--;
            printf("%d\n", C[lenc].x);
          }
        }
        else
        { if(lenb!=0) //B 不空时从 B 出栈元素
          { lenb--;
            printf("%d\n", B[lenb].x);
          }
          else //B 空时从公共栈 C 出栈元素
          { lenc--;
            printf("%d\n", C[lenc].x);
          }
        }
      }
    }
  }
  else //merge A B
  { char tmp[10];
    gets(tmp); //忽略 B 参数, 将 A 和 B 队列中的所有元素合并到 C 中
    int i=0, j=0;
    while (i < lena && j < lenb) //按 timeid 二路归并
    { if (A[i].timeid < B[j].timeid) //归并 timeid 较小的 A 中元素
      { C[lenc] = Node(A[i].x, A[i].timeid);

```

```

        i++; lenc++;
    }
    else //归并 timeid 较小的 B 中元素
    { C[lenc]=Node(B[j].x,B[j].timeid);
      j++; lenc++;
    }
}
while (i < lena) //归并 A 中的剩余元素
{ C[lenc]=Node(A[i].x,A[i].timeid);
  i++; lenc++;
}
while (j < lenb) //归并 B 中的剩余元素
{ C[lenc]=Node(B[j].x,B[j].timeid);
  j++; lenc++;
}
lena=lenb=0; //A、B 清空
}
}
return 0;
}

```

运行通过,执行用时为 858ms,内存消耗为 2140KB,所用语言为 C++。从中看出解法 2 的时空性能稍好于解法 1。

3.2.4 HDU6215——暴力排序

问题描述参见本书第 2 章中的 2.2.7 节,这里采用队列求解。

参考博客: <https://www.cnblogs.com/sbfhy/p/7576974.html>

解: 直接用 queue<int>类型的队列 qu 替代 list<int>类型的 lst。对应的程序如下:

```

#include <iostream>
#include <cstring>
#include <queue>
using namespace std;
#define MAXN 200005

struct DNode //静态双链表结点类型
{ int val; //存放整数
  int prior; //前驱位置
  int next; //后继位置
} a[MAXN]; //a 数组作为静态双链表

int n,m;
bool vis[MAXN]; //表示元素是否已删除

int main()
{ int t;
  scanf("%d",&t);
  while (t--)
  { scanf("%d",&n);
    memset(vis,false,sizeof(vis));
    queue<int> qu; //定义一个队列
  }
}

```



视频讲解

```

vis[0] = vis[n+1] = true;
a[0].val = 0; a[0].next = 1;           //a[0]为头结点
a[n+1].val = INT_MAX;
for (int i = 1; i <= n; i++)           //建立双链表
{ scanf("%d", &a[i].val);
  a[i].next = i+1;
  a[i].prior = i-1;
}
for (int i = 1; i <= n; i++)           //将 a[i-1] <= a[i] 且 a[i] > a[i+1] 的无序元素进队
  if(a[i-1].val <= a[i].val && a[i].val > a[i+1].val)
    qu.push(i);
m = n;
while(!qu.empty())                   //处理无序元素
{ int u = qu.front();                //出队元素
  qu.pop();
  if (vis[u])                        //a[u]已删除时跳过
    continue;
  if (a[u].val <= a[a[u].next].val)   //a[u]正序时跳过
    continue;
  vis[u] = true;                     //a[u]反序时标记 qu 中的 a[u]被删除
  m--;
  int pos;
  for (pos = a[u].next; pos <= n; pos = a[pos].next) //依次处理 a[u]后面的元素
  { if(a[a[pos].prior].val <= a[pos].val) //正序时退出循环
    break;
    vis[pos] = true;                 //反序时标记 qu 中的 a[pos]被删除
    m--;
  }
  a[a[u].prior].next = pos;           //链表操作:删除 a[u]到 a[pos]之前的所有结点
  a[pos].prior = a[u].prior;
  qu.push(a[u].prior);               //将 a[u].prior 进队
}
printf("%d\n", m);                   //输出结果
for(int i = a[0].next; i <= n; i = a[i].next)
  printf("%d ", a[i].val);
printf("\n");
}
return 0;
}

```

上述程序是 AC 代码,运行时间为 327ms,占用的空间为 3120KB,编译语言为 C++。



视频讲解

3.2.5 HDU4699——编辑器

问题描述参见本书第 2 章中的 2.2.8 节,这里采用栈求解。

参考博客: <https://www.cnblogs.com/ZhenghangHu/p/9759435.html>

解: 设计两个栈,prestack 栈存放当前光标的前面部分元素,poststack 存放当前光标的后面部分元素,其他与 2.2.8 节的思路相同。对应的程序如下:

```
#include <iostream>
```

```

#include <stack>
#include <algorithm>
using namespace std;
#define MAXN 1000010
int presum[MAXN];           //presum[pos]存放 pos 位置的前缀和
int maxpresum[MAXN];        //maxpresum[pos]存放 pos 位置的最大前缀和
stack<int> prest;            //存放当前光标之前的序列
stack<int> postst;          //存放当前光标之后的序列
int main()
{ int n;
  while (~scanf("%d", &n))
  { while(!prest.empty()) //清空 prest 栈
    prest.pop();
    while(!postst.empty()) //清空 postst 栈
    postst.pop();
    int pos=0;             //当前位置
    int len=0;
    maxpresum[0]=-INT_MAX;
    presum[0]=0;
    for(int i=1;i<=n;i++) //处理 n 个指令
    { char op[5];
      scanf("%s", op);
      if (op[0]=='I') //I x 指令
      { int x;
        scanf("%d", &x);
        prest.push(x); //把 x 进 prest 栈
        pos++;
        presum[pos]=presum[pos-1]+x; //求 pos 位置的前缀和
        maxpresum[pos]=max(maxpresum[pos-1], presum[pos]); //求 pos 位置的最大前缀和
      }
      else if (op[0]=='D') //D 指令
      { prest.pop();
        pos--;
      }
      else if (op[0]=='L') //L 指令
      { if (prest.empty()) continue;
        pos--;
        int tmp=prest.top();
        prest.pop();
        postst.push(tmp);
      }
      else if (op[0]=='R') //R 指令
      { if (postst.empty()) continue;
        pos++;
        int e=postst.top();
        postst.pop();
        prest.push(e);
        presum[pos]=presum[pos-1]+prest.top();
        maxpresum[pos]=max(maxpresum[pos-1], presum[pos]);
      }
      else if (op[0]=='Q') //Q k 指令

```

```

        { int k;
          scanf("%d",&k);
          printf("%d\n",maxpresum[k]);
        }
      }
    }
    return 0;
}

```

上述程序是 AC 代码,运行时间为 1232ms,占用的空间为 15172KB,编译语言为 C++。



视频讲解

3.2.6 HDU6375——度度熊学队列

时间限制: 1500ms; 内存限制: 131072KB。

问题描述: 度度熊正在学习双端队列,它对双端队列的翻转和合并产生了很大的兴趣。初始时有 N 个空的双端队列(编号为 $1 \sim N$),用户要支持度度熊的 Q 次操作。

① $1\ u\ w\ val$: 在编号为 u 的队列中加入一个权值为 val 的元素($w=0$ 表示加在最前面, $w=1$ 表示加在最后面)。

② $2\ u\ w$: 询问编号为 u 的队列中的某个元素并删除它($w=0$ 表示询问并操作最前面的元素, $w=1$ 表示询问并操作最后面的元素)。

③ $3\ u\ v\ w$: 把编号为 v 的队列“接在”编号为 u 的队列的最后面。 $w=0$ 表示顺序接(队列 v 的开头和队列 u 的结尾连在一起,队列 v 的结尾作为新队列的结尾), $w=1$ 表示逆序接(先将队列 v 翻转,再顺序接在队列 u 的后面)。该操作完成后,队列 v 被清空。

输入格式: 有多组测试用例,对于每组测试用例,第一行读入两个数 N 和 Q 。接下来有 Q 行,每行 3~4 个数,意义如上。规定 $N \leq 150000$, $Q \leq 400000$, $1 \leq u, v \leq N$, $0 \leq w \leq 1$, $1 \leq val \leq 100000$,所有数据的和不超过 500000。

输出格式: 对于每个测试用例的每一个操作②,输出一行表示答案。注意,如果操作②的队列是空的,就输出 -1 且不执行删除操作。

输入样例:

```

2 10
1 1 1 23
1 1 0 233
2 1 1
1 2 1 2333
1 2 1 23333
3 1 2 1
2 2 0
2 1 1
2 1 0
2 1 1

```

输出样例:

```

23
-1
2333

```

233

23333

参考博客：https://blog.csdn.net/weixin_30274627/article/details/99221872

解：直接用双端队列 `deque<int>` 容器数组 `dqu` 来模拟，根据输入的指令编号 x 执行要求的操作。对应的程序如下：

```
#include <deque>
#include <iostream>
using namespace std;
#define MAXN 150010
deque<int> dqu[MAXN];
int main()
{ int N, Q;
  while(~scanf("%d %d", &N, &Q))
  { for(int i=0; i<=N; i++)
    dqu[i].clear();
    int x, u, v, w, val;
    for(int i=0; i<Q; i++)           //处理 Q 条指令
    { scanf("%d", &x);
      if(x==1)                       //1 u w val
      { scanf("%d %d %d", &u, &w, &val);
        if(w==0)                   //加在最前面
          dqu[u].push_front(val);
        else if(w==1)              //在最后面
          dqu[u].push_back(val);
      }
      else if(x==2)                 //2 u w
      { scanf("%d %d", &u, &w);
        if(dqu[u].empty())
        { cout << -1 << endl;
          continue;
        }
        if(w==0)                   //询问并操作 del 最前面的元素, 询问要求输出
        { cout << dqu[u].front() << endl;
          dqu[u].pop_front();
        }
        else if(w==1)              //最后面
        { cout << dqu[u].back() << endl;
          dqu[u].pop_back();
        }
      }
    }
    else if(x==3)                   //3 u v w, 该操作完成后队列 v 被清空
    { scanf("%d %d %d", &u, &v, &w);
      if(w==0)                     //顺序接
      { while(!dqu[v].empty())
        { dqu[u].push_back(dqu[v].front());
          dqu[v].pop_front();
        }
      }
      else if(w==1)                //逆序接
```

```

        { while(!dqu[v].empty())
          { dqu[u].push_back(dqu[v].back());
            dqu[v].pop_back();
          }
        }
      }
    }
  }
  return 0;
}

```

上述程序是 AC 代码,运行时间为 936ms,占用的空间为 25380KB,编译语言为 C++。



视频讲解

3.2.7 HDU4393——扔钉子

时间限制: 2000ms; 内存限制: 32768KB。

问题描述: 年度学校自行车比赛开始了,ZL 是这所学校的学生,他太无聊了,因为他不能骑自行车! 因此他决定干预比赛,他通过以前的比赛视频获得了选手的信息,一个选手第一秒可以跑 F 米,然后每秒跑 S 米。每个选手有一条直线跑道,ZL 每秒向跑的最远的运动员的跑道扔一个钉子,在自行车胎爆了之后,该选手将被淘汰。如果有多个选手是 NO. 1, 则他总是选择 ID 最小的选手扔钉子。

输入格式: 在第一行中有一个整数 T ($T \leq 20$),表示测试用例的数量。每个测试用例的第一行包含一个整数 n ($1 \leq n \leq 50000$),表示选手的人数,接着是 n 行,每行包含第 i 个选手的两个整数 F_i ($0 \leq F_i \leq 500$)、 S_i ($0 < S_i \leq 100$),表示该选手第一秒可以跑 F_i 米,然后每秒跑 S_i 米, i 是玩家从 1 开始的 ID。

输出格式: 对于每个测试用例,第一行的输出均为“Case #c:”, c 是测试用例的编号(从 1 开始)。第二行输出 n 个数字,以空格分隔,第 i 个数字是选手的 ID,该选手将在第 i 秒结束时被淘汰。

输入样例:

```

2
3
100 1
100 2
3 100
5
1 1
2 2
3 3
4 1
3 4

```

输出样例:

```

Case #1:
1 3 2
Case #2:
4 5 3 2 1

```

参考博客：https://blog.csdn.net/rssj_chlh/article/details/8040023

解：题目大意是给出 n 个人，每个人有两个速度，第一个是第一秒的速度，第二个是第一秒以后的平均速度。找到每一秒输出跑的路程最长的一个人，然后淘汰他，直到所有人输出完毕为止。如果路程相同就输出 ID 最小的那一个。

采用优先队列，每次出队一个最远选手即可。由于选手人数 n 可达 50000，如果建立一个优先队列会出现超时，为此按选手的平均速度划分，由于选手的平均速度最大为 100，这样建立 105 个优先队列，用 qu 数组表示，每次遍历所有队列，通过比较找到路径长度最长的选手 pid 出队。对应的程序如下：

```
#include <iostream>
#include <algorithm>
#include <queue>
using namespace std;
struct QNode
{
    int id; //选手 ID
    int len; //选手跑了路径长度
    bool operator < (const QNode & s) const
    {
        if (len == s.len) //路径长度相同, ID 越小越优先
            return id > s.id;
        else //路径长度不同, 其值越大越优先
            return len < s.len;
    }
};
priority_queue<QNode> qu[105];
int main()
{
    int t;
    scanf("%d", &t);
    int cas=1;
    while (t--)
    {
        int n;
        scanf("%d", &n);
        cout << "Case #" << cas++ << ": " << endl;
        for(int i=0; i<n; i++) //处理 n 个选手
        {
            int f, s;
            QNode p;
            cin >> f >> s; //输入 f 和 s
            p.len=f;
            p.id=i+1;
            qu[s].push(p); //所有选手按 p 速度划分
        }
        for(int i=0; i<n; i++) //扔 n 次钉子(每秒一次)
        {
            int pos, pid;
            int maxs=-1; //最长的长度
            for(int j=1; j<101; j++) //遍历每个优先队列
            {
                if (!qu[j].empty())
                {
                    QNode p=qu[j].top();
                    if (p.len+j*i > maxs || (p.len+j*i == maxs && p.id < pid))
                    {
                        pos=j; //记录是哪个优先队列
                        pid=p.id; //记录选手的 ID
                    }
                }
            }
        }
    }
}
```

```

        maxs=p.len+j*i;           //记录其连接长度
    }
}
if (i!=n-1)
    cout << pid << " ";
else
    cout << pid;
qu[pos].pop();                   //淘汰的选手 pid 出队
}
cout << endl;
}
}

```

上述程序是 AC 代码,运行时间为 920ms,占用的空间为 2996KB,编译语言为 C++。



视频讲解

3.2.8 POJ3032——纸牌戏法

时间限制: 1000ms; 内存限制: 65536KB。

问题描述: 魔术师随机抽取一小叠纸牌,将其面朝下握住,然后执行以下步骤。

① 第 1 步: 将顶部的一张纸牌移到底部,新的顶部纸牌面朝上放在桌子上,该纸牌是黑桃 1(A)。

② 第 2 步: 做两次移动,每次将顶部的一张纸牌移到底部,两次做完后将新的顶部纸牌面朝上放在桌子上,该纸牌是黑桃 2。

③ 第 3 步: 做 3 次移动,每次将顶部的一张纸牌移到底部,3 次做完后将新的顶部纸牌面朝上放在桌子上,该纸牌是黑桃 3。

④ 以此类推,直到第 n 步做 n 次移动后新的顶部纸牌面朝上放在桌子上,该纸牌是黑桃 n 。

如果魔术师知道如何事先排列纸牌,则此技巧非常有用。编写程序在给定纸牌数量 $n(1 \leq n \leq 13)$ 的情况下确定纸牌的初始顺序。

输入格式: 输入的第一行是一个正整数,表示测试用例数。每个测试用例由包含整数 n 的一行组成。

输出格式: 对于每个测试用例,输出一行为 $1 \sim n$ 的某个排列,以空格分隔,第一个整数为顶部纸牌的数字,以此类推。

输入样例:

```

2
4
5

```

输出样例:

```

2 1 4 3
3 1 4 5 2

```

解: 例如,当 $n=4$ 时,求解结果序列是 2143,魔术师的操作步骤如下。

① 第 1 步,将 2 放到队列末端变成 1432,拿走 1,剩下 432。

② 第2步,将4、3分别放到底部末端,变成243,拿走2,剩下43。

③ 第3步,把4放到末端,把3放到末端,再把4放到末端,变成34,拿走3,剩下4。

④ 第4步,只有4,做4次,拿走4。

用队列模拟,按与上述相反的步骤操作,因为魔术师操作 n 步拿走的纸牌依次是 $1\sim n$,所以按 $n\sim 1$ 的顺序进队,整个模拟过程如下:

① 4进队。

② 3进队,出、进队做3遍,即4出队/4进队,3出队/3进队,4出队/4进队。

③ 2进队,出、进队做2遍,即3出队/3进队,4出队/4进队。

④ 1进队,出、进队做1遍,即2出队/2进队。

此时从队头到队尾是3412,反向便是求解结果2143。用`queue<int>`容器`qu`作为队列,对应的程序如下:

```
#include <iostream>
#include <queue>
using namespace std;
const int MAXN=15;           //最大 n
queue<int> qu;
int ans[MAXN];               //存放结果
int main()
{ int t, n, d;
  scanf("%d", &t);           //测试用例的个数
  while(t--)
  { while (!qu.empty())      //队列清空
    { qu.pop();
      scanf("%d", &n);
      qu.push(n);
      for(int i=n-1; i>=1; i--)
      { qu.push(i);
        for(int j=1; j<=i; j++)
        { d=qu.front();
          qu.pop();
          qu.push(d);
        }
      }
    }
    for(int i=0; i<n; i++)    //按出队顺序放在 ans[1..n] 中
    { ans[n-i]=qu.front();
      qu.pop();
    }
    for(int i=1; i<n; i++)    //输出结果 ans
    { printf("%d ", ans[i]);
      printf("%d\n", ans[n]);
    }
  }
  return 0;
}
```

上述程序是 AC 代码,运行时间为 16ms,运行占用的空间为 128KB,编译语言为 C++。



视频讲解

3.2.9 POJ2259——团队队列

时间限制：2000ms；内存限制：65536KB。

问题描述：在团队队列中每个成员都属于一个团队，如果一个成员进入队列，他首先从头到尾搜索队列，以检查他的一些队友（同一队的成员）是否已经在队列中，如果是，他会进入到该团队的后面；如果不是，他会从尾部进入队列并成为新的最后一个成员。成员出队是按常规队列操作，按照出现在队列中的顺序从头到尾进行处理。请编写一个模拟这样的团队队列的程序。

输入格式：包含一个或多个测试用例，每个测试用例都以团队个数 t 开始 ($1 \leq t \leq 1000$)，然后是团队描述，每个描述包含属于团队的成员个数和成员编号列表，成员编号为 $0 \sim 999999$ 的整数，一个团队最多可以包含 1000 个成员。然后是一系列命令，有以下 3 种不同的命令。

① ENQUEUE p ：成员 p 进入队列。

② DEQUEUE：队列中的第一个成员出来并将其从队列中删除。

③ STOP：当前测试用例结束。

输入以 $t=0$ 结束。

输出格式：对于每个测试用例，首先输出一行“Scenario #cas”，其中 cas 是测试用例的编号，然后对于每个 DEQUEUE 命令，以单独一行输出出队的成员。在每个测试用例之后输出一个空白行，即使是在最后一行之后。

输入样例：

```
2
3 101 102 103
3 201 202 203
ENQUEUE 101
ENQUEUE 201
ENQUEUE 102
ENQUEUE 202
ENQUEUE 103
ENQUEUE 203
DEQUEUE
DEQUEUE
DEQUEUE
DEQUEUE
DEQUEUE
DEQUEUE
STOP
2
5 259001 259002 259003 259004 259005
6 260001 260002 260003 260004 260005 260006
ENQUEUE 259001
ENQUEUE 260001
ENQUEUE 259002
ENQUEUE 259003
ENQUEUE 259004
```

```
ENQUEUE 259005
DEQUEUE
DEQUEUE
ENQUEUE 260002
ENQUEUE 260003
DEQUEUE
DEQUEUE
DEQUEUE
DEQUEUE
STOP
0
```

输出样例：

```
Scenario #1
101
102
103
201
202
203
```

```
Scenario #2
259001
259002
259003
259004
259005
260001
```

解：采用如图 3.8 所示的结构模拟,每个团队用一个队列表示,由于团队的编号是 1~1000,所以用 `termqu[1001]` 数组表示,即 `termqu[k]` 表示编号为 k 的团队队列。另外设置一个总队列 `totalqu`,其元素为团队队列的编号。当一个团队队列中有一个成员时将其团队编号进入总队列,如果一个团队队列的成员数为 0,则从总队列中删除。所以每个团队 `termqu[k]` 包含对应的团队队列和是否出现在总队列中的标志 `vis`。每个成员用成员编号表示,进队后有一个对应的团队队列,设置数组 `termno`,`termno[p]=k` 表示成员编号 p 的团队队列的编号为 k 。

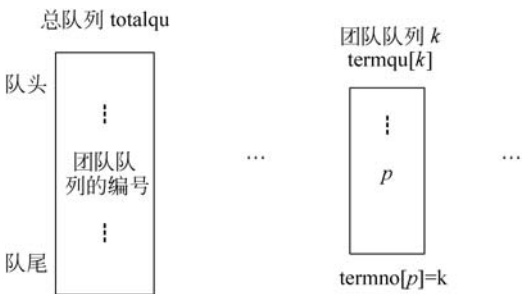


图 3.8 团队队列的结构

先初始化上述结构,处理各个命令如下。

① ENQUEUE p : 将成员 p 进入团队编号为 $\text{termno}[p]$ 的队列($\text{termqu}[\text{termno}[p]]$),如果该团队队列在总队列中不存在,将 $\text{termno}[p]$ 进入总队列 totalqu 并且设置其 vis 为 true 。

② DEQUEUE: 总队列中的队头为 $\text{totalqu.front}()$ 团队队列,出队其中的一个成员并且输出,如果该团队队列为空,则从总队列 totalqu 中出队并且设置其 vis 为 false 。

对应的程序如下:

```
#include <iostream>
#include <queue>
#include <string>
#include <cstring>
using namespace std;
const int MAXN=1000000;           //最大成员编号
int termno[MAXN];                 //每个成员所属团队的编号
int n;
queue<int> totalqu;                //总队列
struct
{ queue<int> qu;                  //队列
  bool vis;                       //表示这个团队是否在总队列中
} termqu[1001];                  //每个团队队列
void solve(int cas)               //求解算法
{ cout << "Scenario #" << cas << endl;
  string com;
  while (cin >> com && com!="STOP")
  { if (com=="ENQUEUE")           //ENQUEUE p
    { int p;
      cin >> p;
      termqu[termno[p]].qu.push(p); //成员 p 进入 termqu[termno[p]] 队列
      if (!termqu[termno[p]].vis)   //若 p 是该团队的第一个成员
      { totalqu.push(termno[p]);    //将该团队编号添加到总队列中
        termqu[termno[p]].vis=true; //置该团队在总队列中
      }
    }
    else if (com=="DEQUEUE")       //DEQUEUE
    { cout << termqu[totalqu.front()].qu.front() << endl;
      termqu[totalqu.front()].qu.pop(); //总队列中的第一个团队出队
      if (termqu[totalqu.front()].qu.empty()) //该团队为空的情况
      { termqu[totalqu.front()].vis=false;
        totalqu.pop();
      }
    }
    else break;                    //STOP
  }
}

void init()                        //初始化
{ memset(termno,0,sizeof(termno)); //成员所属团队清 0
  while(!totalqu.empty())          //总队列清空
    totalqu.pop();
  for(int i=1;i<=n;i++)            //每个团队队列清空
```

```

    { while(!termqu[i].qu.empty())
      { termqu[i].qu.pop();
        termqu[i].vis=false;
      }
    }
}

int main()
{ int cas=1; //测试用例编号
  int t,p;
  while(cin >> n && n != 0) //输入 n(团队个数)
  { init();
    for(int i=1;i<=n;i++) //输入 n 个团队
    { scanf("%d",&t); //输入团队人数 t
      for(int j=1;j<=t;j++) //输入 t 个成员
      { scanf("%d",&p);
        termno[p]=i;
      }
    }
    solve(cas++); //求解
    cout << endl;
  }
  return 0;
}

```

上述程序是 AC 代码,运行时间为 469ms,运行占用的空间为 4544KB,编译语言为 C++。

3.2.10 POJ2559——最大矩形面积

时间限制: 1000ms; 内存限制: 65536KB。

问题描述: 与本书中的实战 3.5 求柱状图的最大矩形面积类似。

输入格式: 输入包含几个测试用例。每个测试用例描述一个直方图,并以整数 n ($1 \leq n \leq 100000$) 开头,表示它所组成的矩形个数,然后跟随 n 个整数,即 $h_1, h_2, \dots, h_n$ ($0 \leq h_i \leq 1000000000$),这些整数表示从左到右顺序的直方图矩形的高度,每个矩形的宽度为 1。以输入 n 为 0 结束。

输出格式: 对于每个测试用例输出一行,指定直方图中最大矩形的面积。

输入样例:

```

7 2 1 4 5 1 3 3
4 1000 1000 1000 1000
0

```

输出样例:

```

8
4000

```

解: 采用单调栈 st,ans 表示最大矩形面积(初始为 0),求解思路参见本书中实战 3.5 的解法 2,这里有两点修改。

(1) 在所有柱子高度输入完毕后,再对单调栈中的所有柱子做一次计算,而不是通过添加 0 高度的柱子自动触发计算。



视频讲解

(2) 每个柱子都进栈一次、出栈一次,栈中的每个柱子除了高度 h 成员外还带一个 len 成员表示该柱子对应矩形的宽度,在进栈和出栈过程中累加当前柱子的宽度,在出栈中求对应矩形的面积,比较求出 ans 。

以输入柱子高度 $heights[] = \{2, 1, 5, 6, 2, 3\}$ 为例:

① $i=0, [2, 1]$ 进栈 ($[2, 1]$ 中的前者 2 为高度,后者 1 为该柱子当前对应矩形的宽度)。 $st = ([2, 1])$ 。

② $i=1$, 出栈 $[2, 1]$, $tmp=0$, 栈顶柱子对应的宽度 $= 1 + tmp = 1$ (出栈时 tmp 表示当前柱子的后面宽度), 求出面积 $= 2 \times 1 = 2$ 。置 $tmp=1$, 将 $[1, 1 + tmp] = [1, 2]$ 进栈 (进栈时 tmp 表示当前柱子的前面宽度)。 $st = ([1, 2])$ 。

③ $i=2, [5, 1]$ 进栈, $st = ([1, 2], [5, 1])$ 。

④ $i=3, [6, 1]$ 进栈, $st = ([1, 2], [5, 1], [6, 1])$ 。

⑤ $i=4$, 出栈 $[6, 1]$, $tmp=0$, 栈顶柱子对应的宽度 $= 1 + tmp = 1$, 求出面积 $= 6 \times 1 = 6$ 。置 $tmp=1$, 出栈 $[5, 1]$, 栈顶柱子对应的宽度 $= 1 + tmp = 2$, 求出面积 $= 5 \times 2 = 10$ 。置 $tmp=2$, 将 $[2, 1 + tmp] = [2, 3]$ 进栈。 $st = ([1, 2], [2, 3])$ 。

⑥ $i=5, [3, 1]$ 进栈。 $st = ([1, 2], [2, 3], [3, 1])$ 。

⑦ 最后计算, $tmp=0$, 出栈 $[3, 1]$, 栈顶柱子对应的宽度 $= 1 + tmp = 1$, 求出面积 $= 3 \times 1 = 3$ 。置 $tmp=1$, 出栈 $[2, 3]$, 栈顶柱子对应的宽度 $= 3 + tmp = 4$, 求出面积 $= 2 \times 4 = 8$ 。置 $tmp=4$, 出栈 $[1, 2]$, 栈顶柱子对应的宽度 $= 2 + tmp = 6$, 求出面积 $= 1 \times 6 = 6$ 。栈空结束。

得到最大面积 $= 10$ 。对应的程序如下:

```
#include <iostream>
#include <stack>
using namespace std;

struct Node                                //栈元素类型
{ long long h;                             //柱子的高度
  long long len;                           //当前柱子高度的矩形宽度
};

stack<Node> st;
int main()
{ long long tmp, ans, area, n;
  Node q;
  while (scanf("%lld", &n) != EOF)
  { if (n == 0) break;
    ans = 0;
    for (int i = 0; i < n; i++)
    { scanf("%lld", &q.h);                  //输入一个柱子高度
      q.len = 1;
      if (st.empty())                       //栈空时直接进栈
        st.push(q);
      else if (q.h <= (st.top()).h)         //反序时
      { tmp = 0;                            //tmp 表示前一个柱子对应矩形的宽度
        while (!st.empty() && q.h <= (st.top()).h)
        { st.top().len += tmp;              //栈顶柱子较高,累加其矩形宽度
          area = st.top().h * st.top().len; //求矩形面积
          if (area > ans) ans = area;        //求最大矩形面积
        }
      }
    }
  }
}
```

```

        tmp=st.top().len;
        st.pop();
    }
    q.len+=tmp;
    st.push(q);
}
else st.push(q);
}
tmp=0;
while (!st.empty())                //输入完毕做最后一次计算
{ st.top().len+=tmp;
  area=st.top().h * st.top().len;
  if (area > ans) ans=area;
  tmp=st.top().len;
  st.pop();
}
cout << ans << endl;
}
}

```

上述程序是 AC 代码,运行时间为 360ms,运行占用的空间为 1304KB,编译语言为 C++。

3.2.11 POJ3984——迷宫问题

时间限制：1000ms；内存限制：65536KB。

问题描述：定义一个二维数组。



视频讲解

```

int maze[5][5] = {
    0, 1, 0, 0, 0,
    0, 1, 0, 1, 0,
    0, 0, 0, 0, 0,
    0, 1, 1, 1, 0,
    0, 0, 0, 1, 0,
}

```

它表示一个迷宫,其中的 1 表示墙壁,0 表示可以走的路,注意只能横着走或竖着走,不能斜着走,要求编程找出从左上角到右下角的最短路线。

输入格式：一个 5×5 的二维数组,表示一个迷宫。数据要保证有唯一解。

输出格式：从左上角到右下角的最短路径,格式如样例所示。

输入样例：

```

0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0

```

输出样例：

```

(0, 0)
(1, 0)

```

```
(2, 0)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(3, 4)
(4, 4)
```

解法 1: 由于求的是最短路径,采用队列求迷宫问题的方法,与《教程》中 3.2.7 节的原理完全相同。在输出迷宫路径时,通过一个 path 数组将反向路径反向输出即得到正向路径。对应的程序如下:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
const int MAX=5; //迷宫的最大行、列数
int dx[]={-1,0,1,0}; //x 方向的偏移量
int dy[]={0,1,0,-1}; //y 方向的偏移量
int mg[MAX][MAX];
int m=5,n=5;

struct Box //队列中的方块元素类型
{ int i; //方块的行号
  int j; //方块的列号
  Box * pre; //本路径中上一方块的地址
  Box() {} //构造函数
  Box(int i1,int j1) //重载构造函数
  { i=i1;
    j=j1;
    pre=NULL;
  }
};

void disppath(queue<Box *> &qu) //输出一条迷宫路径
{ vector<Box> apath; //存放一条迷宫路径
  Box * b;
  b=qu.front(); //从队头开始向入口方向搜索
  while (b!=NULL)
  { apath.push_back(Box(b->i,b->j)); //将搜索的方块添加到 apath 中
    b=b->pre;
  }
  for (int i=apath.size()-1;i>=0;i--) //反向输出构成一条正向迷宫路径
    printf("(%d, %d)\n",apath[i].i,apath[i].j);
}

void mgpath(int xi,int yi,int xe,int ye) //求一条从(xi,yi)到(xe,ye)的迷宫路径
{ Box * b, * b1;
  queue<Box *> qu; //定义一个队列 qu
  b=new Box(xi,yi); //建立入口的对象 b
  qu.push(b); //入口对象 b 进队,其 pre 置为 NULL
  mg[xi][yi]=-1; //为避免来回找相邻方块置 mg 值为-1
  while (!qu.empty()) //队不空时循环
  { b=qu.front(); //取队头方块 b
```

```

    if (b->i==xe && b->j==ye)           //找到了出口,输出路径
    {   disppath(qu);                   //输出一条迷宫路径
        return;
    }
    qu.pop();                           //出队方块 b
    for (int di=0;di<4;di++)            //循环扫描每个方位,把每个可走的方块进队
    {   int i=b->i+dx[di];               //找 b 的 di 方位的相邻方块(i,j)
        int j=b->j+dy[di];
        if (i>=0 && i<m && j>=0 && j<n && mg[i][j]==0)
        {   b1=new Box(i,j);           // (i,j) 方块有效且可走
            b1->pre=b;                  //将该相邻方块进队,并置其 pre 为前驱方块
            qu.push(b1);
            mg[i][j]=-1;                //为避免来回找相邻方块置 mg 值为-1
        }
    }
}
}
}
int main()
{   for (int i=0;i<MAX;i++)
    {   for (int j=0;j<MAX;j++)
        {   scanf("%d",&mg[i][j]);
            mngpath(0,0,4,4);
        }
    }
    return 0;
}

```

上述程序是 AC 代码,运行时间为 0ms,运行占用的空间为 168KB,编译语言为 C++。

解法 2: 解法 1 中队列 qu 元素是创建的 Box 对象的指针,通过 pre 变量表示前驱关系,算法设计相对复杂,这里采用 pre 二维数组存放搜索中的前驱方块,例如 pre[3][4].i=1, pre[3][4].j=2 表示方块(3,4)的前驱方块是(1,2),当找到出口时求最短路径。对应的程序如下:

```

#include <iostream>
#include <vector>
#include <queue>
using namespace std;
const int MAX=5;           //迷宫最大的行、列数
int dx[]={-1,0,1,0};      //x 方向的偏移量
int dy[]={0,1,0,-1};      //y 方向的偏移量
int mg[MAX][MAX];
int m=5,n=5;

struct Box                 //队列中的方块元素类型
{   int i;                 //方块的行号
    int j;                 //方块的列号
    Box() {}               //构造函数
    Box(int i1,int j1)      //重载构造函数
    {   i=i1;
        j=j1;
    }
};

Box pre[MAX][MAX];        //表示前驱关系

```

```

void disppath(int xi,int yi,int xe,int ye)           //输出一条迷宫路径
{ vector<Box> apath;                                //存放一条迷宫路径
  int i,j,il,jl;
  i=xe; j=ye;                                       //从(xe,ye)反推逆路径
  while (i!=xi || j!=yi)
  { apath.push_back(Box(i,j));                      //将搜索的方块添加到 apath 中
    il=pre[i][j].i;
    jl=pre[i][j].j;
    i=il; j=jl;
  }
  apath.push_back(Box(xi,yi));                      //将入口添加到 apath 中
  for (int i=apath.size()-1;i>=0;i--)              //反向输出构成一条正向迷宫路径
    printf("(%d, %d)\n", apath[i].i, apath[i].j);
}

void mgpath(int xi,int yi,int xe,int ye)            //求一条从(xi,yi)到(xe,ye)的迷宫路径
{ Box b, b1;
  queue<Box> qu;                                     //定义一个队列 qu
  b=Box(xi,yi);                                     //建立入口的对象 b
  qu.push(b);                                       //入口对象 b 进队
  mg[xi][yi]=-1;                                    //为避免来回找相邻方块置 mg 值为-1
  while (!qu.empty())                               //队不空时循环
  { b=qu.front();                                   //取队头方块 b
    if (b.i==xe && b.j==ye)                         //找到了出口,输出路径
    { disppath(xi,yi,xe,ye);                       //输出一条迷宫路径
      return;
    }
    qu.pop();                                       //出队方块 b
    for (int di=0;di<4;di++)                       //循环扫描每个方位,把每个可走的方块进队
    { int i=b.i+dx[di];                             //找 b 的 di 方位的相邻方块(i,j)
      int j=b.j+dy[di];
      if (i>=0 && i<m && j>=0 && j<n && mg[i][j]==0)
      { b1=Box(i,j);                               //(i,j)方块有效且可走
        pre[i][j].i=b.i; pre[i][j].j=b.j;          //置其前驱方块为(b.i,b.j)
        qu.push(b1);                                //将该相邻方块进队
        mg[i][j]=-1;                                //为避免来回找相邻方块置 mg 值为-1
      }
    }
  }
}

int main()
{ for (int i=0;i<MAX;i++)
  for (int j=0;j<MAX;j++)
    scanf("%d",&mg[i][j]);
  mgpath(0,0,4,4);
  return 0;
}

```

上述程序是 AC 代码,运行时间为 0ms,运行占用的空间为 168KB,编译语言为 C++。



视频讲解

3.2.12 POJ1686——算术式子是否等效

时间限制: 1000ms; 内存限制: 10000KB。

问题描述：数学老师太懒了，没能在试卷中给问题打分，在试卷中学生给出了求解问题的复杂的公式，不同的学生可能给出不同的形式，但都是正确的答案，这使得评分非常困难。因此，数学老师需要计算机程序员的帮助。请编写一个程序读取不同的公式，并确定它们在算术上是否等效。

输入格式：输入的第一行包含一个整数 n ($1 \leq n \leq 20$)，它是测试用例的数量。在第一行之后，每个测试用例都有两行。一个测试用例由两个算术表达式组成，每个算术表达式位于单独的行中，最多 80 个字符。在输入中没有空行，表达式包含以下一项或多项：

- ① 单字母变量(不区分大小写)。
- ② 一位数字。
- ③ 匹配的左、右括号。
- ④ 二元运算符+、-和*分别用于加、减和乘。
- ⑤ 上述标记之间的任意空白或制表符字符。

注意：对于所有运算符，表达式在语法上都是正确的，并且从左到右具有相同的优先级。保证变量的系数和指数适合 16 位整数。

输出格式：程序必须为每个测试用例输出一行，如果每个测试数据的输入表达式在算术上等效，则输出“YES”，否则输出“NO”，输出应全部使用大写字符。

输入样例：

```
3
(a+b-c)*2
(a+a)+(b*2)-(3*c)+c
a*2-(a+c)+((a+c+e)*2)
3*a+c+(2*e)
(a-b)*(a-b)
(a*a)-(2*a*b)-(b*b)
```

输出样例：

```
YES
YES
NO
```

参考博客：<https://www.cnblogs.com/lxm940130740/p/3289003.html>

解：采用《教程》第3章中3.1.7节用栈求简单表达式值的思路，对于输入的两个算术表达式求出值分别为 ans1 和 ans2，若两者相同，输出“YES”，否则输出“NO”。对应的程序如下：

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <stack>
using namespace std;
#define MAXN 90
int priority(char c) //求运算符 c 的优先级
{ if(c == '(')
    return 0;
```

```

        else if(c == ' * ')
            return 2;
        else
            return 1;
    }
void convert(char * exp, char * postexp)           //转换为后缀表达式
{
    int len = strlen(exp), t = 0;
    char ch;
    stack < char > st;                               //定义一个栈
    for(int i = 0; i < len; i++)
        if(exp[i] != ' ')
        {
            ch = exp[i];
            if ((ch <= 'z' && ch >= 'a') || (ch >= '0' && ch <= '9'))
                postexp[t++] = ch;
            else
            {
                if (st.empty() || ch == '(')
                    st.push(ch);
                else if (ch == ')')
                {
                    while (!st.empty() && st.top() != '(')
                    {
                        postexp[t++] = st.top();
                        st.pop();
                    }
                    st.pop();
                }
            }
            else
            {
                while (!st.empty() && priority(ch) <= priority(st.top()))
                {
                    postexp[t++] = st.top();
                    st.pop();
                }
                st.push(ch);
            }
        }
    }
    while (!st.empty())
    {
        postexp[t++] = st.top();
        st.pop();
    }
    postexp[t] = '\0';
}

int calculate(char * postexp)           //求后缀表达式值
{
    int len = strlen(postexp), a, b, c;
    char ch;
    stack < int > st;
    for(int i = 0; i < len; i++)
    {
        ch = postexp[i];
        if (ch >= '0' && ch <= '9')
            st.push(ch - '0');
        else if (ch <= 'z' && ch >= 'a')
            st.push(int(ch));
        else
        {
            a = st.top(); st.pop();

```

```
b=st.top(); st.pop();
switch(ch)
{
case '*':
    c=b*a;
    break;
case '+':
    c=b+a;
    break;
case '-':
    c=b-a;
    break;
}
st.push(c);
}
}
return st.top();
}
int main()
{ char exp[MAXN], postexp[MAXN];
  int n;
  scanf("%d", &n);
  getchar();
  while(n-->0)
  { gets(exp);
    convert(exp, postexp);
    int ans1=calculate(postexp);
    gets(exp);
    convert(exp, postexp);
    int ans2=calculate(postexp);
    if(ans1==ans2)
        printf("YES\n");
    else
        printf("NO\n");
  }
}
```

上述程序是 AC 代码,运行时间为 0ms,运行占用的空间为 104KB,编译语言为 C++。