第 8 章

栈 (STACK)

本章内容

- 8.1 定义和应用
- 8.2 抽象数据类型
- 8.3 数组描述
- 8.4 链表描述
- 8.5 应用

8.1 定义和应用

- 定义[栈]：栈（stack）是一个线性表，其插入（也称为添加）和删除操作都在表的同一端进行。
  - 其中允许插入和删除的一端被称为栈顶（top）
  - 另一端被称为栈底（bottom）

- $e_0, e_1, e_2, \ldots, e_i, \ldots, e_{n-2}\ e_{n-1}$
  ↑ ↑
  bottom top

举例：打印机的打印纸托盘
自助餐厅的一摞餐盘

栈结构

D ←top
C
B
A ←bottom

E ←top
D
C
B
A ←bottom

D ←top
C
B
A ←bottom

C ←top
B
A ←bottom

●栈是一个后进先出(LIFO (Last-In, First-Out))表.

8.2 抽象数据类型

抽象数据类型 stack
{
实例
元素线性表，一端为栈底，另一端为栈顶
操作
empty()：//栈为空时返回true，否则返回false
size()：//返回栈中元素个数
top()：//返回栈顶元素
pop()：//删除栈顶元素
push(x)：//将元素x压入栈
}

C++抽象类stack

```
template <class T>
class stack
{
 public:
   virtual ~stack() {}
   virtual bool empty() const = 0;
         //栈为空时返回true，否则返回false
   virtual int size() const = 0;
         //返回栈中元素个数
   virtual T& top() = 0;
         //返回栈顶元素
   virtual void pop() = 0;
         //删除栈顶元素
   virtual void push(const T& theElement) = 0;
         //将元素theElement压入栈
}
```

栈的描述方法

- 栈可以使用任何一种线性表的描述方法
  - 数组描述
  - 链表描述

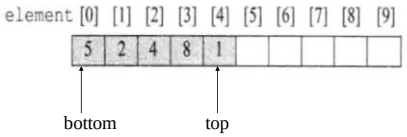
8.3 数组描述

- 栈使用数组描述，有两种实现方法:
  - 1. 使用数组描述的线性表arrayList， 通过arrayList类的派生得到数组描述的栈类derivedArrayStack
  - 2.定制数组描述的栈类arrayStack类

8.3.1 从arrayList派生实现

```
template <class T>
class arrayList : public linearList<T>
{
public:
    arrayList(int initialCapacity = 10);
    ……
    bool empty() const ;
    int size() const ;
    T& get(int theIndex) const;
    int indexOf(const T& theElement) const;
    void erase(int theIndex);
    void insert(int theIndex, const T& theElement);
    ……
protected:
    T *element;       //存储线性表元素的一维数组
    int  arrayLength;  //一维数组的容量
    int  listSize;        //线性表的元素个数
};
```


从arrayList派生derivedArrayStack



- 栈顶元素的索引： arrayList<T>::size()-1
- 应用arrayList类中的方法实现

从arrayList派生derivedArrayStack

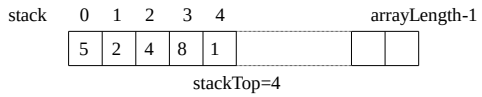
```
template<class T>
class derivedArrayStack : private arrayList <T>,
                          public  stack<T>;
{public:
    derivedArrayStack(int initialCapacity = 10) :
                    arrayList<T> (initialCapacity) {}
    bool empty() const  {return arrayList<T>::empty();}
    int size() const      {return (arrayList<T>::size() ;}
    T& top()     {if (arrayList<T>::empty()) throw  StackEmpty();
                    return get(arrayList<T>::size()-1);}
    void pop()   {if (arrayList<T>::empty()) throw  StackEmpty();
                  erase(arrayList<T>::size()-1);}
    void push(const T& theElement)
            {insert(arrayList<T>::size(), theElement);}
};
```


对类derivedArrayStack的评价

- 从arrayList派生derivedArrayStack：
  - 优点：
    - 大大减少了编码量。
    - 使程序的可靠性得到很大提高 。
  - 缺点：
    - 运行效率降低。
      - 例： push(const T& theElement) 。

类arrayStack

- 定制数组描述的栈



- 栈容量: arrayLength
- 栈中元素个数：stackTop+1
- 栈空：stackTop=-1
- 栈满：stackTop=arrayLength-1

8.3.2 类arrayStack

```
template<class T>
class arrayStack ： public  stack<T>;
{public:        arrayStack(int initialCapacity = 10);
                ~ arrayStack() { delete [] stack; }
                bool empty() const { return stackTop == -1; }
                int size() const { return stackTop+1; }
                T& top();
                void pop();
                void push(const T& theElement);

 private:       int stackTop;  //当前栈顶
                int arrayLength;  //栈容量
                T *stack;    //元素数组
};
```


arrayStack构造函数

```
template<class T>
arrayStack<T>::arrayStack(int initialCapacity = 10);
{ // 构造函数
   if (initialCapacity < 1)  …../输出错误信息，抛出异常
   arrayLength = initialCapacity;
   stack = new T[arrayLength];
   stackTop = -1;//栈空
}
```


arrayStack方法top，pop

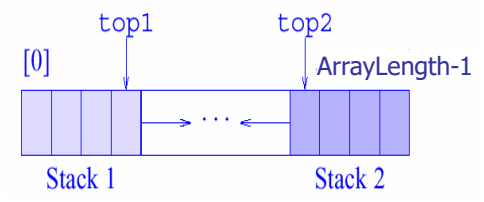
```
template<class T>
T& arraystack<T>::top()
  {
       if (stackTop == -1) throw StackEmpty();
       return stack[stackTop];
  }

template<class T>
void arrayStack<T>::pop()
{
  if (stackTop == -1) throw StackEmpty();
 stack[stackTop--].~T();//T的析构函数
}
```


arrayStack方法push

```
template<class T>
void arrayStack<T>:: push(const T& theElement);
 {
  //如果栈已满，则容量加倍.
  if (stackTop == arrayLength-1)
       {changeLength1D(stack, arrayLength, 2*arrayLength);
         arrayLength*=2;}

   //在栈顶插入元素
   stack[++stackTop] = theElement;
}
```


- 当同时使用多个栈时:
  - 浪费大量的空间
- 若仅同时使用两个栈，则是一种例外。
- 如何在一个数组中描述两个栈?

在一个数组中描述两个栈

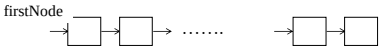


8.4 链表描述

- 栈使用链表描述，有两种实现方法:
  - 1. 使用链表描述的线性表chain， 通过chain类的派生得到链表描述的栈类derivedLinkedStack
  - 2.定制链表描述的栈类linkedStack类

8.4.1类derivedLinkedStack

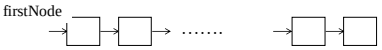
- 从chain派生类derivedLinkedStack
- 链表的哪一端对应于栈顶?



- 把链表的右端作为栈顶
  - stack: top() →chain: get(size()-1)
  - stack: push(theElement) → chain: Insert(size(), theElement)
  - stack: pop() → chain: erase(size()-1)
  - 时间复杂性 : **Θ(size())**

8.4.1类derivedLinkedStack

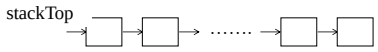
- 从chain派生类derivedLinkedStack
- 链表的哪一端对应于栈顶?



- 把链表的左端作为栈顶
  - stack: top() →chain: get(0)
  - stack: push(theElement) → chain: Insert(0, theElement)
  - stack: pop() → chain: erase(0)
  - 时间复杂性 : **Θ(1)**

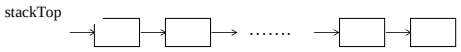
8.4.2类LinkedStack

- 定制链表栈



- 成员：
  - stackTop：栈顶指针
  - stackSize：栈中元素个数
- 空栈：stackTop=NULL，stackSize=0
- 栈顶元素：stackTop->element

8.4.2类LinkedStack




```
template<class T>
class LinkedStack : public stack<T>;
{ public:   LinkedStack(int initialCapacity = 10);
                {stackTop=NULL; stackSize=0; }
            ~ LinkedStack();
            bool empty() const { return stackSize == 0; }
            int size() const { return stackSize; }
            T& top();
            void pop();
            void push(const T& theElement);
  private:  chainNode<T>* stackTop; //栈顶指针
            int  stackSize;   //栈中元素个数
};
```

与数组表述对比

```
private:    int stackTop;  //当前栈顶
            int arrayLength;  //栈容量
            T *stack;    //元素数组
```

读程序 8-5

8.5 应用

- **8.5.1 括号匹配**
- 8.5.2 汉诺塔
- **8.5.3 列车车厢重排**
- 8.5.4 开关盒布线
- 8.5.5 离线等价类问题
- **8.5.6 迷宫老鼠**

8.5.1 括号匹配

- 问题:匹配一个字符串中的左、右括号
- (a*(b+c)+d)
  - 3 7
  - 0 10
- (a+b))*((c+d)
  - 0 4
  - No match for right parenthesis at 5
  - 8 11
  - No match for left parenthesis at 7

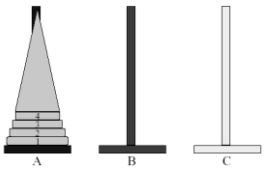
从左到右扫描一个字符串，每一个右括号都与**最近扫描的**那个未匹配的左括号相匹配

括号匹配C++实现

```
void PrintMatchedPairs(string expr)
{// 括号匹配
 arrayStack<int> s;
 int length = (int) expr.size();//输入表达式的长度
 // 扫描表达式expr，寻找‘(’ ‘和)’
 for (int i = 0; i<length; i++)
  {if (expr.at(i) = =' ( ' ) s.push(i);
   else if (expr.at(i) = =' ) ' )
        try {//从栈中删除匹配的左括号
             cout << s.top() <<' ' << i << endl;
             s.pop(); }
        catch (stackEmpty)
          { //栈空，没有匹配的左括号
            cout<<"No match for right parenthesis"<<" at "<<i<<endl;}
   }
// 栈不为空，栈中所剩下的左括号都是未匹配的
while(!s.empty()) {
     cout << "No match for left parenthesis at " << s.top() < endl;
     s.pop();
     }
}
```

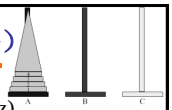
时间复杂度O(n)

8.5.2 汉诺塔



- 已知n个碟子和3座塔。初始时所有的碟子按从大到小次序从塔A的底部堆放至顶部，我们需要把碟子都移动到塔B，过程中可以借助塔C，但要求:
  - 每次移动一个碟子.
  - 任何时候都不能把大碟子放到小碟子的上面。

汉诺塔（递归方法）



```
void towersOfHanoi(int n, int x, int y, int z)
{ / /把n 个碟子从塔x 移动到塔y，可借助于塔z
   if (n > 0) {
   towersOfHanoi(n-1, x,z,y);//n-1个碟子从x移到z，借助y
   cout << "Move top disk from tower " << x
        <<“ to top of  tower ” << y << endl;//x最大的碟子移到y
   towersOfHanoi(n-l, z, y, x); //n-1个碟子从z移到y，借助x     }
}
```

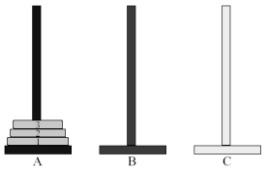

汉诺塔（递归方法）

•碟子的移动次数：

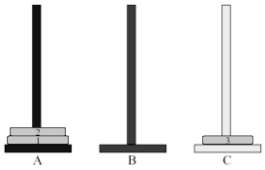
$$moves(n)=\begin{cases}0 & n=0\\ 2moves(n-1)+1 & n>0\end{cases}$$

- $moves(n)= 2^n-1$
- 时间复杂性: $\Theta(2^n)$

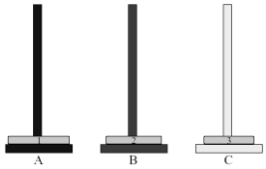
塔的布局



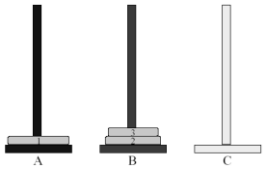
塔的布局



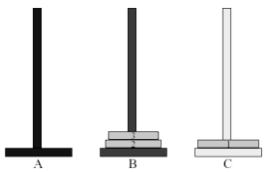
塔的布局



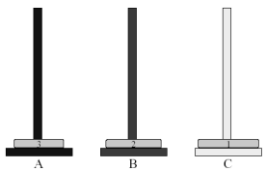
塔的布局



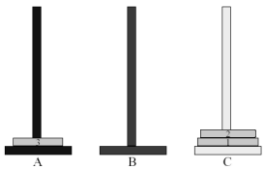
塔的布局



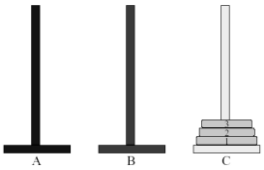
塔的布局



塔的布局



塔的布局



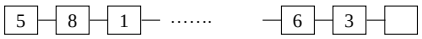
汉诺塔（栈模拟塔）

```
//全局变量，tower[1:3]表示三个塔
arrayStack<int> tower[4];
void moveAndShow(int n, int x, int y, int z);

void towersOfHanoi(int n)
{// 函数moveAndShow的预处理程序
    for (int d = n; d > 0; d--) // 初始化
        tower[1].push(d); // 把碟子d 放到塔1上

  //把塔1上的n个碟子移动到塔2上，借助于塔3 的帮助
    moveAndShow(n, 1, 2, 3);
}
```


汉诺塔 （栈模拟塔）

```
void moveAndShow(int n, int x, int y, int z)
{// 把n 个碟子从塔x 移动到塔y，可借助于塔z
if (n > 0) {
    moveAndShow(n-l, x, z, y); //把n-1个碟子从x挪到z
    int d= tower[x].top(); // 获得x最大那个碟子的编号d
    tower[x].pop(); //从x中移走最大那个碟子
    tower[y].push(d); //把x最大那个碟子碟子放到y 上
    ShowState(); //显示塔的布局
    moveAndShow(n-l, z, y, x); //把n-1个碟子从z挪到y}
}
```


8.5.3 列车车厢重排



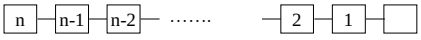
货运列车共有*n*节车厢，每节车厢将停放在不同的车站

*n*个车站的编号分别为1到*n*

货运列车按照**第*n*站至第1站**的次序经过这些车站

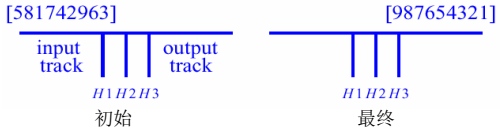
车厢的编号与它们的目的地相同。

重新排列车厢，使各车厢从前至后按编号1到*n*的次序排列。



这样排列之后，在每个车站只需卸掉最后一节车厢即可

列车车厢重排例（3个缓冲铁轨）



初始　　最终

- 缓冲铁轨是按照LIFO的方式使用的
- 在重排车厢过程中，仅允许以下移动：
  - 车厢可以从入轨的前部（即右端）移动到一个缓冲铁轨的顶部或出轨的左端。
  - 车厢可以从一个缓冲铁轨的顶部移动到出轨的左端。

3个缓冲铁轨中间状态

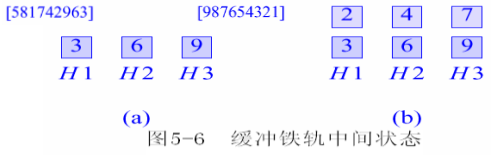


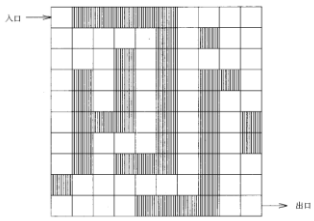
图5-6 缓冲铁轨中间状态

- 选择缓冲铁轨的分配规则
  - 新的车厢u应送入这样的缓冲铁轨：其顶部的车厢编号v满足v > u，且v是所有满足这种条件的缓冲铁轨顶部车厢编号中最小的一个编号。
- k个链表形式的栈来描述k个缓冲铁轨。

实现思路

int NowOut=1； // NowOut:下一次要输出的车厢号
for （int i=1;i<=n;i++）//从前至后依次检查的所有车厢
{1.车厢 p[i] 从入轨上移出
2.If (p[i] == NowOut)
①把p[i]放到出轨上去； NowOut++;
② while (minH(当前缓冲铁轨中编号最小的车厢)== NowOut ）
{把minH 放到出轨上去；
更新 minH ;NowOut++;}
else 按照分配规则将车厢p[i]送入某个缓冲铁轨。
}
读程序 8-9——8-12

8.5.6 迷宫老鼠

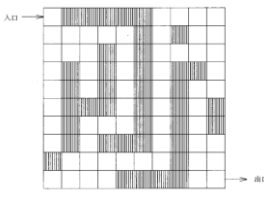


- 迷宫是一个矩阵区域
- 迷宫老鼠(rat in a maze)问题要求寻找一条从入口到出口的路径。

迷宫的描述

- 假定用**n×m的矩阵**来描述**迷宫**，位置(1,1)表示入口，(n,m)表示出口，n和m分别代表迷宫的行数和列数。
- 迷宫中的每个位置都可用其行号和列号来指定。在矩阵中，当且仅当在位置(i,j)处有一个障碍时其值为1，否则其值为0。

迷宫的描述



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

8.5.6 迷宫老鼠

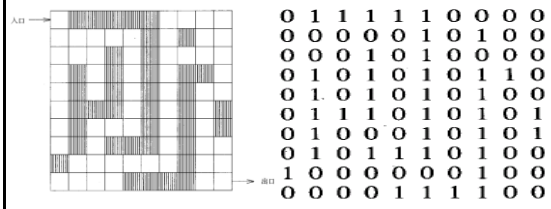
- 迷宫老鼠(rat in a maze)问题要求寻找一条从入口到出口的路径。
- 路径是由一组位置构成的，每个位置上都没有障碍，且每个位置(第一个除外)都是前一个位置的东、南、西或北的邻居。

迷宫老鼠程序设计

图 8-12　迷宫老鼠程序的模块化结构

```
void main()
{
   welcome();
   inputMaze();
   if(findPath())
      outputPath();
   else
      cout<< "No path "<<endl;
}
```


寻找路径



寻找路径设计思路

- 首先把迷宫的入口作为当前位置。
- 如果当前位置是迷宫出口，那么已经找到了一条路径，搜索工作结束。
- 如果当前位置不是迷宫出口，则在当前位置上放置障碍物，以便阻止搜索过程又绕回到这个位置。
- 检查相邻的位置中是否有空闲的(即没有障碍物)，如果有，就移动到这个新的相邻位置上，然后从这个位置开始搜索通往出口的路径。如果不成功，选择另一个相邻的空闲位置，并从它开始搜索通往出口的路径。在进入新的相邻位置之前，把**当前位置保存在一个栈**中，…..。
- **如果相邻的位置中没有空闲的，则回退到上一位置**。
- 如果所有相邻的空闲位置都已经被探索过，并且未能找到路径，则表明在迷宫中不存在从入口到出口的路径。

寻找路径

```
1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 0 0 0 0 1
1 0 0 0 0 0 1 0 1 0 0 1
1 0 0 0 0 1 0 1 0 0 0 1
1 0 1 0 1 0 1 0 1 1 0 1
1 0 1 0 1 0 1 0 1 0 0 1
1 0 1 1 1 1 0 1 0 1 1 1
1 0 1 0 0 0 1 0 1 0 1 1
1 0 1 0 1 1 1 0 1 0 0 1
1 1 0 0 0 0 0 0 1 0 0 1
1 0 0 0 0 1 1 1 1 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

图 8-15　设有一圈障碍物的图 8-9 的迷宫

寻找路径算法

```
bool FindPath()
{ //寻找从位置(1,1)到出口(m,m)的路径
增加一圈障碍物;
//对跟踪当前位置的变量进行初始化
Position here;
here.row = 1;
here.col = 1;
maze[1][1] = 1; // 阻止返回入口
```



```
//寻找通往出口的路径
while (here不是出口) do {
  选择here的下一个可行的相邻位置;
  if (存在这样一个相邻位置neighbor) {
    把当前位置here 放入堆栈path ;
    //移动到相邻位置，并在当前位置放上障碍物
    here = neighbor;
    maze[here.row][here.col] = 1;}
  else {
    //不能继续移动，需回溯
    if (堆栈path为空) return false;
    回溯到path栈顶中的位置here; }
}
return true;
}
```

寻找路径

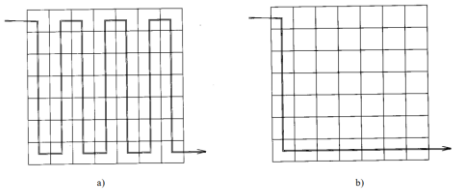


图5-16 没有障碍物的迷宫中的路径
a) 一条长路径 b) 一条短路径

- 现在的方法无法保证能够找到最短路径

- P184 12