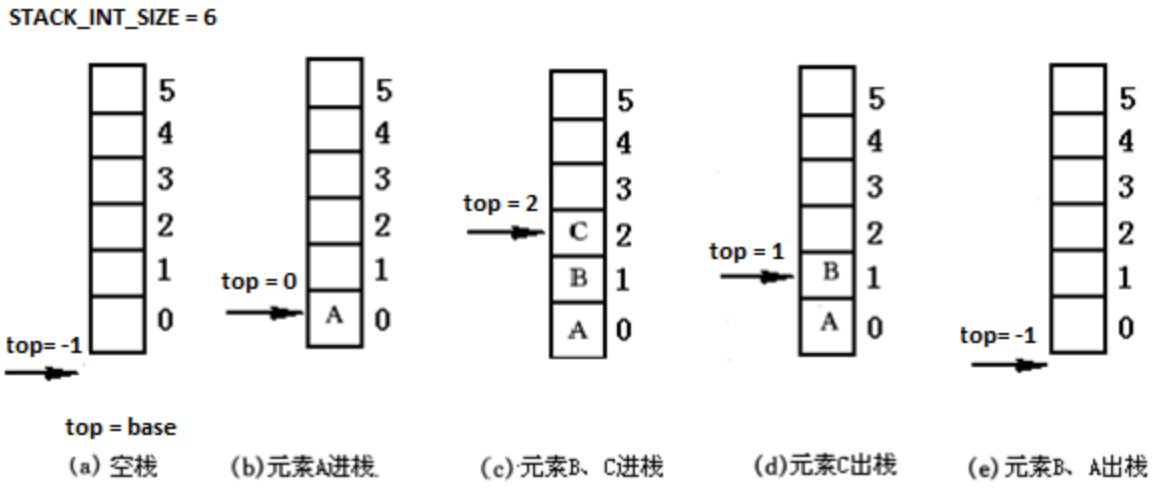
1、堆栈(stack)

定义

- 是一种特殊的线性表，这种表只能在固定的一端进行插入与删除操作。
- 固定插入的一端叫**栈顶(top)**，而另一端称为**栈底(bottom)**。位于栈顶和栈底的元素分别称为**顶元**和**底元**。当表中无元素时，称为空栈

图示



注意事项

- 堆栈也叫做先进后出，如图中进栈顺序为:ABC,而出栈顺序为CBA
- 栈底指针base是固定的，而栈顶指针top随着插入和删除的操作而不断变化。
- 堆栈只是对线性表的插入和删除的位置做了限定，并没有限定插入和删除操作的时间和次序，即随时都可进行进栈和出栈操作
- 后插入的元素永远比先插入的元素先出栈

抽象数据类型ADT

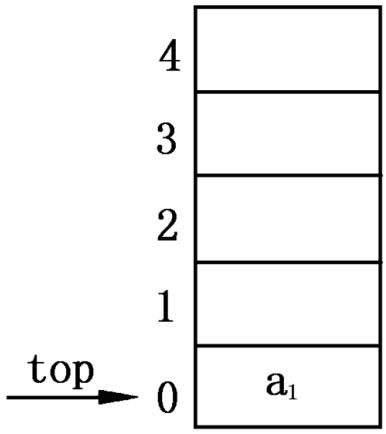
```
ADT List
{
    Data:
    Operation:
        InitStack(&S,maxsize,incresize)
        ClearStack(&S)
        StackLength(S)
        Push(&S,e)
        Pop(&S,&e)
        GetTop(S,&e)
        StackTraverse(S)
        StackEmpty(S)
        DestroyStack(&S)
    }//ADT List
```

2、顺序栈

定义

- **顺序栈**:堆栈的顺序存储结构，利用一组地址连续的存储单元依次存放自栈底到栈顶之间的数据元素。

逻辑图示

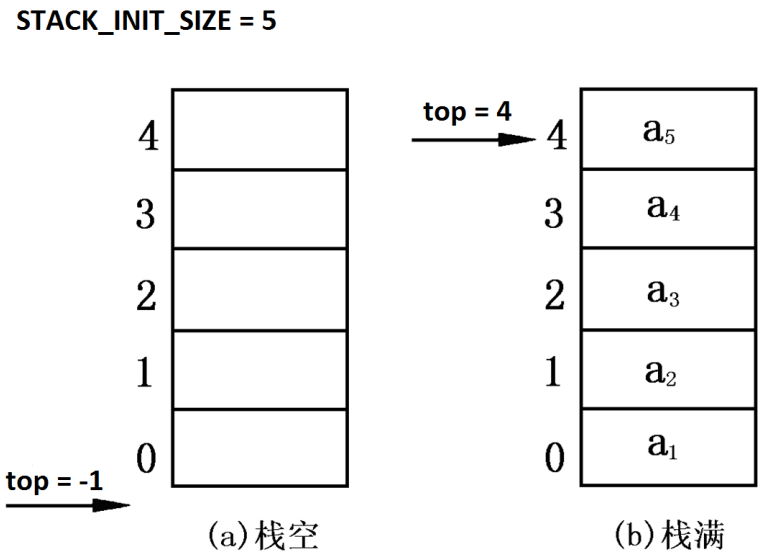


注意事项

- 实质上是顺序表的简化。
- 唯一的约束条件是顺序表的表头(a[0])的一端作为栈底base(base一般数据结构省略掉)，附设top表示**栈顶元素**，top称为**栈顶指针**。
- top = -1表示空栈。
- top = n表示栈中有n+1个元素(n>=0)
- 在顺序栈中，top起到指示栈顶元素的作用，它的值是数组中的下标，因此top是一个相对指针

上溢与下溢

- **上溢(overflow)** : 栈满的情况下还入栈。
- **下溢(underflow)**: 栈空的情况下还出栈。



- 因此，在对顺序栈进行插入元素之前，需要判断是否“栈满”，否则上溢；
- 对顺序栈进行删除元素之前，需要判断是否“栈空”，否则下溢。

- 也就是说：设STACK_INT_SIZE = m,top=m-1时，栈满，此时入栈，则上溢。top = -1时，栈空，此时出栈，则下溢。

顺序栈的数据结构

```
[in SqStack.h]
# define STACK_INIT_SIZE  100       // 顺序栈 (默认的)的初始分配最大容量
# define STACKINCREMENT  10         // (默认的)增补空间量
typedef struct {
    ElemType  *stack;                    // 存储数据元素的一维数组
    int top;                           // 栈顶指针(然而并非真正意义上的指针,只是一个计数器),
    //注意，这里对于顺序表的差别只是将length换成了top
    int stacksize;                     // 当前分配的数组容量（以ElemType为单位）
    int incrementsize;                 // 增补空间量（以ElemType为单位）
}SqStack;
```

顺序栈初始化

```
[in SqStack.h]
void InitStack_Sq(SqStack &S, int maxsize=STACK_INIT_SIZE, int incresize=STACKINCREMENT )
{
    S.stack=(ElemType *)malloc(maxsize*sizeof(ElemType)); // 为顺序栈分配初始存储空间
    if(!S.stack)  exit(1);                                 // 存储空间分配失败
    S.top=-1;                                             // 置栈空
    S.stacksize=maxsize;                                  // 顺序栈的当前容量
    S.incrementsize=incresize;                            // 增补空间
}// InitStack_Sq
```

求顺序栈的长度

```
[in SqStack.h]
int StackLength_Sq(SqStack S)
{
    return S.top+1;
}// StackLength_Sq
```

进栈操作

//在顺序栈的栈顶插入元素e

```
[in SqStack.h]
bool Push_Sq(SqStack &S,ElemType e)
{
    if(S.top==S.stacksize-1)  {
        S.stack =(ElemType *)realloc(S.stack,
(S.stacksize+S.incrementsize)*sizeof(ElemType)); // 栈满，给顺序栈增补空间
        if(!S.stack)     return false;                // 分配存储空间失败
        S.stacksize+=S.incrementsize;
    }
    S.stack[++S.top]=e;                                 // 栈顶指针上移，元素e进栈
    return true;
}// Push_Sq
```

出栈操作

// 删除顺序栈栈顶元素，并让e返回其值

```
[in SqStack.h]
bool Pop_Sq(SqStack &S,ElemType &e)
{
    if(S.top==-1)
        return false;
    e=S.stack[S.top--];
    return true;
}// Pop_Sq
```

取栈顶元素操作

//取顺序栈栈顶元素，并让e返回其值

```
[in SqStack.h]
bool GetTop_Sq(SqStack S, ElemType &e)
{
    if(S.top==-1)
        return false;
    e=S.stack[S.top];
    return true;
}// GetTop_Sq
```

判断栈空操作

在顺序表L的第i个元素之**前**插入新的元素e，若表中当前容量不足，则按预定义的增量扩容

```
[in SqStack.h]
bool StackEmpty_Sq(SqStack S)
{
    if(S.top==-1)
        return true;
    else
        return false;
}// StackEmpty_Sq
```

顺序栈遍历输出各元素

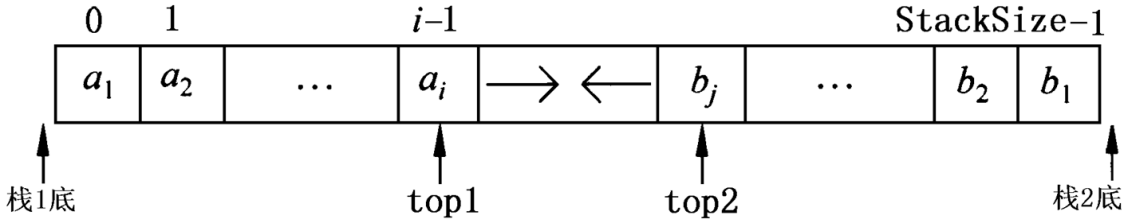
```
[in SqStack.h]
void StackTraverse_Sq(SqStack S)
{
    int i;
    for(i=0;i<StackLength_Sq(S);i++)
        cout<<setw(6)<<S.stack[i];
    cout<<endl;
}// StackTraverse_Sq
```

撤销顺序栈操作

```
[in SqStack.h]
void DestroyStack_Sq(SqStack &S )
{
   free(S.stack);
   S.stack=NULL;
   S.stacksize=0;
   S.top=-1;
}// DestroyStack_Sq
```

3、多栈共享邻接空间

- 定义：即两个栈栈底位置为两端，两个栈顶在中间不断变化，由两边往中间延伸。动态变化（想象把两个花瓶口对口连起来）。
- 目的：若多个栈同时使用，可能出现一个栈的空间被占满而其它栈空间还有大量剩余，因此采用这个方法。
- 特点：设有top1和top2两个指针，top1 = -1表示左栈为空，top2 >= StackSize表示右栈为空; top1 + 1 >= top2时表示栈满。



多站共享数据结构

```
# define StackSize 100          //顺序栈最大容量
typedef struct
{
  ElemType stack[StackSize];
  int top1,top2;
}SqStack_Du;
```

多栈共享初始化

```
void InitStack_DuSq(SqStack_Du&S)
{
   S.top1 = -1;
   S.top2 = StackSize;
}//InitStack_DuSq
```

多栈共享入栈

```
book Push_DuSq(SqStack_Du & S, char WhichStack,ElemType e)
{
   if(S.top1 >= S.top2 - 1)
   {
     cout<<"栈已满!"<<endl;
     return false;
   }
   if(WhichStack != 'L' && WhichStack != 'R')
   {
     cout<<"参数错误"<<endl;
     return false;
   }
   if(WhichStack == 'L')
        S.stack[++S.top1]=e;    //左栈进栈
   else
        S.stack[--S.top2] = e;
   return true;
}
```

多栈共享出栈

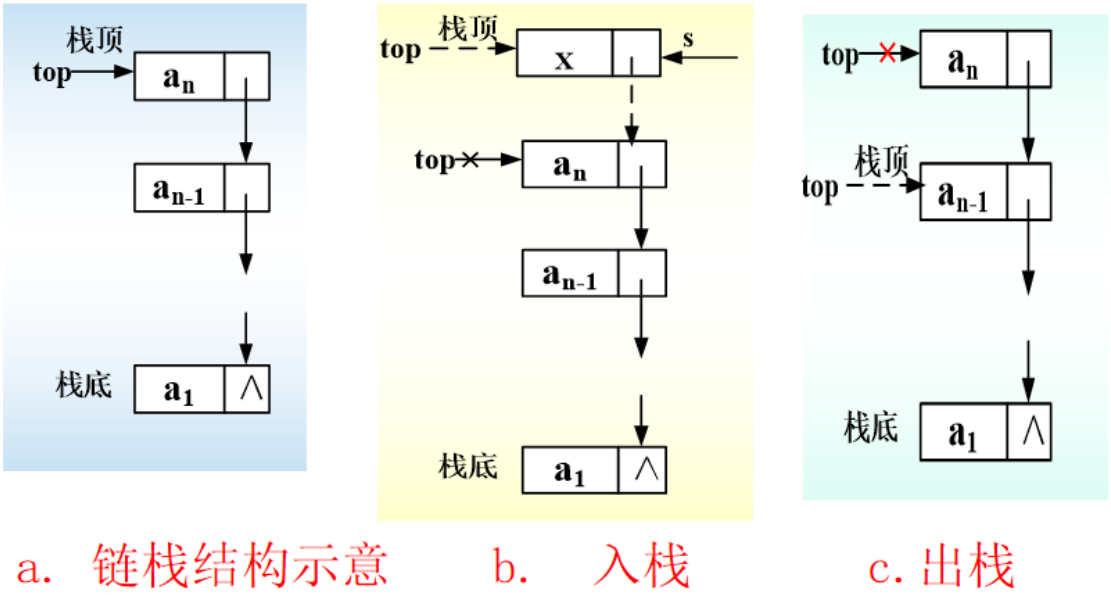
```
book Pop_DuSq(SqStack_Du & S, char WhichStack)
{
   if(S.top1 ==-1 || S.top2 == StackSize)
   {
     cout<<"栈空!"<<endl;
     return false;
   }
   if(WhichStack != 'L' && WhichStack != 'R')
   {
     cout<<"参数错误"<<endl;
     return false;
   }
   if(WhichStack == 'L')
        ElemType e = S.stack[--S.top1];
   else
        ElemType e = S.stack[++S.top2];
   return true;
}//Push_DuSq
```

4、链栈

定义

- 堆栈的链式存储结构简称**链栈**，链栈中每一个数据元素用一个节点表示，实际上就是一个单链表(但是操作上有约束)。

逻辑图示



a. 链栈结构示意　　b. 入栈　　c. 出栈

注意事项

- 实际上就是一个单链表(但是操作上有约束)
- **链栈的栈顶指针就是单链表的头指针**
- 唯一的约束条件是头指针的操作在头部执行
- 一般不需要像单链表那样为了运算方便附加一个头结点。

判定栈空

- 链栈一般不需要判定栈满，只需要判定栈是否为空。
- 栈空判定条件是: S = NULL

链栈的数据结构

```
[in LinkStack.h]
typedef LinkList LinkStack;
```

链栈的初始化

```
[in LinkStack.h]
void InitStack_L(LinkStack &S)
 {
   S=NULL;
  }// InitStack_L
```

求链栈的长度

// 统计链栈S中数据元素的个数，并返回统计结果

```
[in LinkStack.h]
int StackLength_L(LinkStack S)
{
  int k=0;
  LinkStack p=S;
    while(p)
    { k++;
      p=p->next;                          // 访问下一个结点。
    }
   return k;
}// StackLength_L
```

进栈操作

在链栈的栈顶插入元素e

```
[in LinkStack.h]
bool Push_L( LinkStack &S, ElemType e)
{
      LinkStack p;
      if((p=(LNode *)malloc(sizeof(LNode)))==NULL) return false;  // 存储分配失败
      p->data=e;
      p->next=S;                          // 插入新的栈顶元素
      S=p;                                // 修改栈顶指针
      return true;
     }// Push_L
```

出栈操作

// 删除链栈栈顶元素，并让e返回其值

```
[in LinkStack.h]
bool Pop_L( LinkStack &S, ElemType &e)
{
    LinkStack p;
    if(S)                               // 栈非空
    { p=S;S=S->next;                    // 修改栈顶指针
        e=p->data;                      // 元素e返回其值
        free(p);                        // 释放结点空间
        return true;
    }
   else return false;                   // 栈空，出栈失败
 }// Pop_L
```

取栈顶元素操作

```
[in LinkStack.h]
bool GetTop_L(LinkStack S,ElemType &e)
{
    if(S)                       // 栈非空
    { e=S->data;               // 元素e返回其值
        return true;
    }
   else return false;          // 栈空，取栈顶元素失败
}// GetTop_L
```

判栈空操作

判断链栈S是否为空，若为空则返回true,否则返回false

```
[in LinkStack.h]
bool StackEmpty_L(LinkStack S)
{
 if(!S) return true;
 else return false;
}// StackEmpty_L
```

撤销链栈操作

释放链栈所占空间

```
[in LinkStack.h]
void DestroyStack_L(LinkStack &S )
{
  LinkStack p,p1;
  p=S;
  while(p)
  { p1=p;
    p=p->next;
    free(p1);                                  // 释放p1所指的空间
   }
   S=NULL;                                      // S置空
 }// DestroyStack_L
```

5、典型应用

表达式求值（结合真题）

运算过程中栈的变化

中缀表达式转后缀表达式

递归

举例：斐波那契数列（数列的前面相邻两项之和，构成下一项）

下标	1	2	3	4	5	6	7	8	9	0
值	1	1	2	3	5	8	13	21	34	55

值										

迭代

```
array[0] = 1;
array[1] = 1;
System.out.print(array[0] + " " + array[1] + " ");
for (i = 2; i < 20; i++) {
    array[i] = array[i - 1] + array[i - 2];
    System.out.print(array[i] + " ");
}
```

递归

```
int i;
for (i = 0; i < 20; i++){
    System.out.println(Fbi(i));
}
static int Fbi(int i) {
    if (i < 2)
        return 1;
    return Fbi(i - 1) + Fbi(i - 2); // 这里Fbi函数就是自己，它在调用自己
}
```