数据结构
Data Structure

2017年秋季学期

刘鹏远

线性表常见问题解答

多项式求值

约瑟夫问题

线性表(顺序表及链表)总结

新地图开启------栈

```
typedef struct link_node{
        ElemType data;
        struct link_node * next;
}link_node;
```

Q1：这个struct写不写有什么区别？

Q2：最后的link_node为什么要呢？反正没有的话，也能编译出来。

Q3： 为什么要定义一个这样的结构体？建立之后是表示一个结点还是是整个链表？

Q4：为什么整个单链表可用一个指向第一个节点的指针来定义？

```
bool Insert ( link_list *first, Elem_type x, int i ) {
    link_node * p = *first;   int k = 1;
        //...
}
```

Q5. insert函数中：为什么这用*first？*号不是取值符号嘛？

Q6. 为什么要这样定义LinkList *first，如果要定义成LinkList first可行吗？

7.函数参数里什么时候用指针什么时候不用?

8.有点不懂为什么init函数的形参里还是要用指针变量 ?linklist不是本身就是一个指针类型吗?

9.Next域里是存放了下一个节点的data值嘛?

10. struct list_node *next 这个定义为啥又是结构体?

```
typedef struct  list_node{
  ElemType data;
  struct list_node *next;
}list_node;
```

如果多项式是按照降幂排列，按照既定顺序求值即可。

$$((\ldots((c_n x + c_{n-1})\,x + c_{n-2})\,x + \ldots)\,x + c_1)\,x + c_0$$

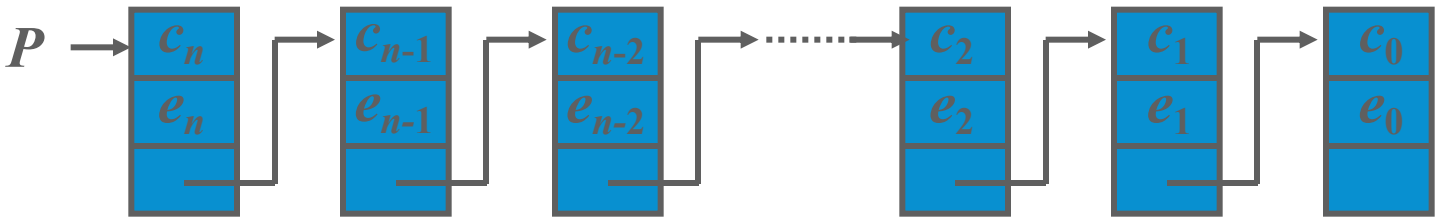
如果多项式有很多零系数项，如

$$25\,x^{101} + 15\,x^{54} + 18\,x^{17} + 2\,x^5 + 6$$

可以考虑改造一下公式，形如：

$$(((\circ\circ\circ(((c_m x^{e_m - e_{m-1}} + c_{m-1})x^{e_{m-1} - e_{m-2}} + c_{m-2})x^{e_{m-2} - e_{m-3}} +$$

$$+ \circ\circ\circ + c_1)x^{e_1 - e_0} + c_0$$



每项需要两个节点数据

然后，设计一个计算 x^i 的函数。

```
float Power ( float x, int i ) {
    float mul = x;
    for ( int j = 1; j < i; j++ ) mul * = x;
    return mul;
}//该函数大多数语言已经内置在库中
```

将指数大小将各项降幂链接，C语义可以直接使用 pow() 函数和上述公式计算多项式的值。#include <math.h>

```
float Evaluate ( poly_nomial pl, float x ) {//伪码
 //计算多项式 pl  在给定 x 时的值
      poly_node *p = pl->next;   //跳过表头
      float rst = p->data.coef;     elem_type a, b;
      if ( p->next== NULL )                  //只有一项
            return  rst*pow(x, p->data.exp);
      while ( p != NULL && p->next!= NULL ) {
            a = p->data;   b = p->next->data;
            rst = rst*Power(x, a.exp-b.exp)+b.coef;
            p = p->next;}
return rst;}
```

实际上，多项式求值，采用顺序结构即可(P40)

多项式乘法可以转化为加法（P43）

如果多项式链表按照指数大小将各项升幂链接怎么办？

对一元多项式，用“有序链表”进行表示与存储更好(P41)

- 应用四：约瑟夫问题模拟

据说著名犹太历史学家 Josephus有过以下的故事：在罗马人占领乔塔帕特后，39 个犹太人与Josephus及他的朋友躲到一个洞中，39个犹太人决定宁愿死也不要被敌人抓到，于是决定了一个自杀方式，41个人排成一个圆圈，由第1个人开始报数，每报数到第3人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。Josephus要他的朋友先假装遵从，他将朋友与自己安排在第16个与第31个位置，于是逃过了这场死亡游戏。

后来衍生了一系列的游戏。

问题可形式化表达为：

N个人围成一圈，从第一个开始报数，第M个将被选择（死亡），直到剩下1（r）个，其余人都已被选择。求选择(死亡)顺序。

逻辑上是相邻关系，因此可用线性表结构。

以顺序表及循环单链表为例来实现。

顺序表算法策略：

初始化位置表(data域置0)，报数变量=0，死亡计数变量=0；

如果死亡人数还小于预定死亡人数（N-r）：

位置表位置++，直到第一个data域非零位置

报数变量++

如果报数变量等于M，则

位置表当前节点data=1，输出当前位置

报数变量置0，死亡计数变量++

节点data域为0表示没死，1表示已死

时间复杂度？

可以证明：$O(N^2)$

循环链表算法策略：

1、可仿照顺序表基本一致来实现。2、还可以：

链表初始化，data域为位置序号。指针、计数器等初始化。

如果死亡人数还小于预定死亡人数（N-r）：

报数变量++

如果报数变量等于M，则

输出当前节点data域，删除该节点

报数变量置0，死亡计数变量++

定位下一个节点

前提是要实现循环链表(带头/不带头节点)的next操作。

时间复杂度？

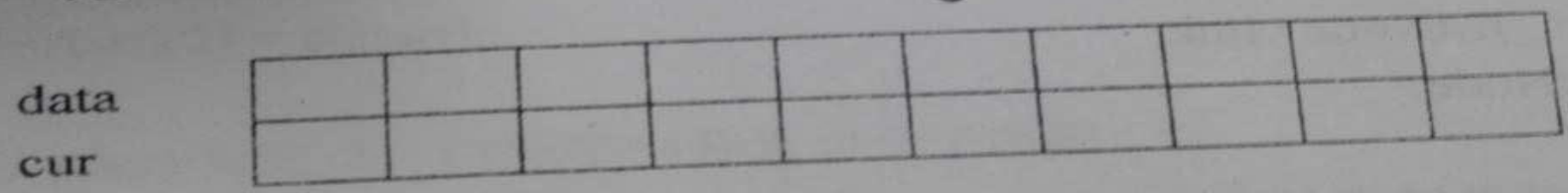
O(NM)　　由M*(N-1)可得

静态链表考试真题

3. 三个逻辑相邻元素（'a'，'b'，'c'），存储在一个最大长度为 MAXSIZE 的静态链表中，假设该静态链表首结点为备用链表的头结点，末结点为当前链表头结点，元素'a'在静态链表中下标为 3 的位置，请在表中画出当前该静态链表状态示意图。（6 分）

静态链表定义如下：

```
#define MAXSIZE 10
typedef struct static_list_node{
    char data;
    int  cur;
}s_list;
s_list my_list[MAXSIZE];
```



静态链表

```
#define      MAXSIZE      1000
typedef struct static_list_node
{
        elem_type         data;
        int     cursor;      //游标cursor
        //int next;  继续用next也可
}s_list;
s_list my_list[MAXSIZE];
```

小问题：是否有动态静态链表，^-^？

应用，用静态链表

问题：利用静态链表计算A-B并B-A

```
int main(void){//伪码
        init(static_list my_list[]);
        while(条件1) insert(my_list[]);//输入A中元素为结果集合my_list
        while(条件2) {
                scanf(“%d”,&b);//依次输入B中元素
                difference(my_list[], b);}//计算A-B并B-A
        visit(my_list);//visit自行实现，遍历打印元素
return 1;}
```

```
int difference(static_list L[], elem_type b)   #类python伪码
        p=L[MAX_SIZE-1]
        while p.cursor!=0 && p.data!=b:                #找b
                p=L(p.cursor)
        if p.cursor!=0:          #如果b存在，则删除A中b
                del(L[], p, e)
        else:                    #如果b不存在，则向A中插入b
                insert(L[], 1, p.data)//还是insert(L[], p, p.data)？哪个好？
        return 1;} //静态链表看以后如有时间，作为作业实现
```

- 小节-----线性结构之线性表
  - 线性结构逻辑推理过程
  - 顺序表---利用一维数组实现
  - 链表---利用指针/游标实现（带头/不带头）
    - 单链表
    - 双向链表
    - 循环链表
    - 双向循环链表
    - 静态链表---数组+游标

ADT、逻辑结构、存储结构、操作及应用

线性结构中顺序表与链表比较

逻辑相邻，物理相邻

实现

随机存取？

插入 删除 定位 移动元素（分别的时间复杂度）

应用：

具体事物---->抽象表示

视需求选择结构与设计操作及算法

视时间复杂度决定各种具体实现

作业（附加，算1/2次作业）：
增加多项式打印函数、多项式求值函数

多项式打印函数，参数为一个多项式的顺序表/链表
可以打印形如：2*x+3*x**4+4.5*x**7+....

求值函数，参数：多项式顺序表/链表（按指数倒序排列的多项式），int x
函数返回该多项式的值

线性结构之

栈

队列

栈
队列
栈与队列的应用
栈与递归

栈

概念

ADT定义

顺序存储结构

操作实现

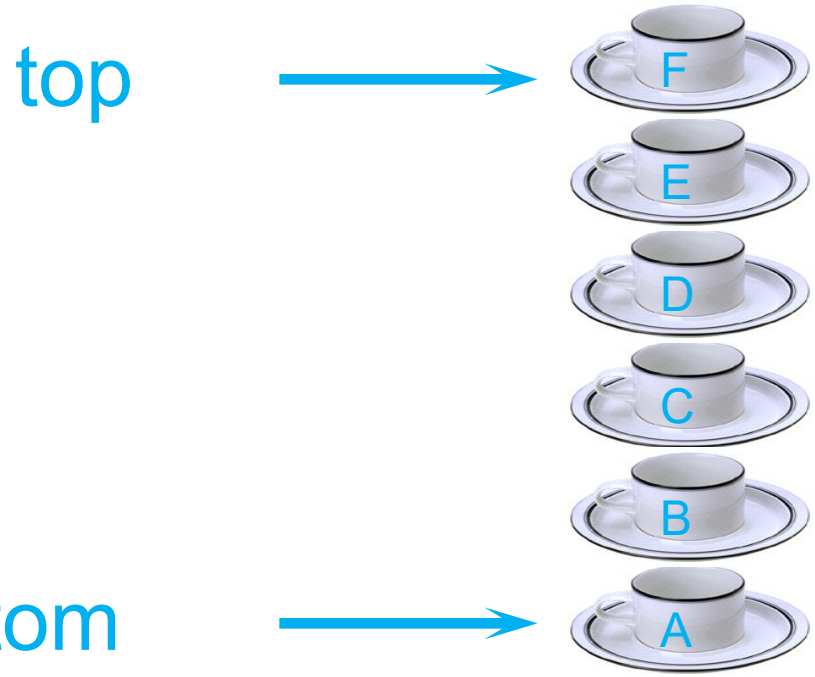
链式存储结构

操作实现

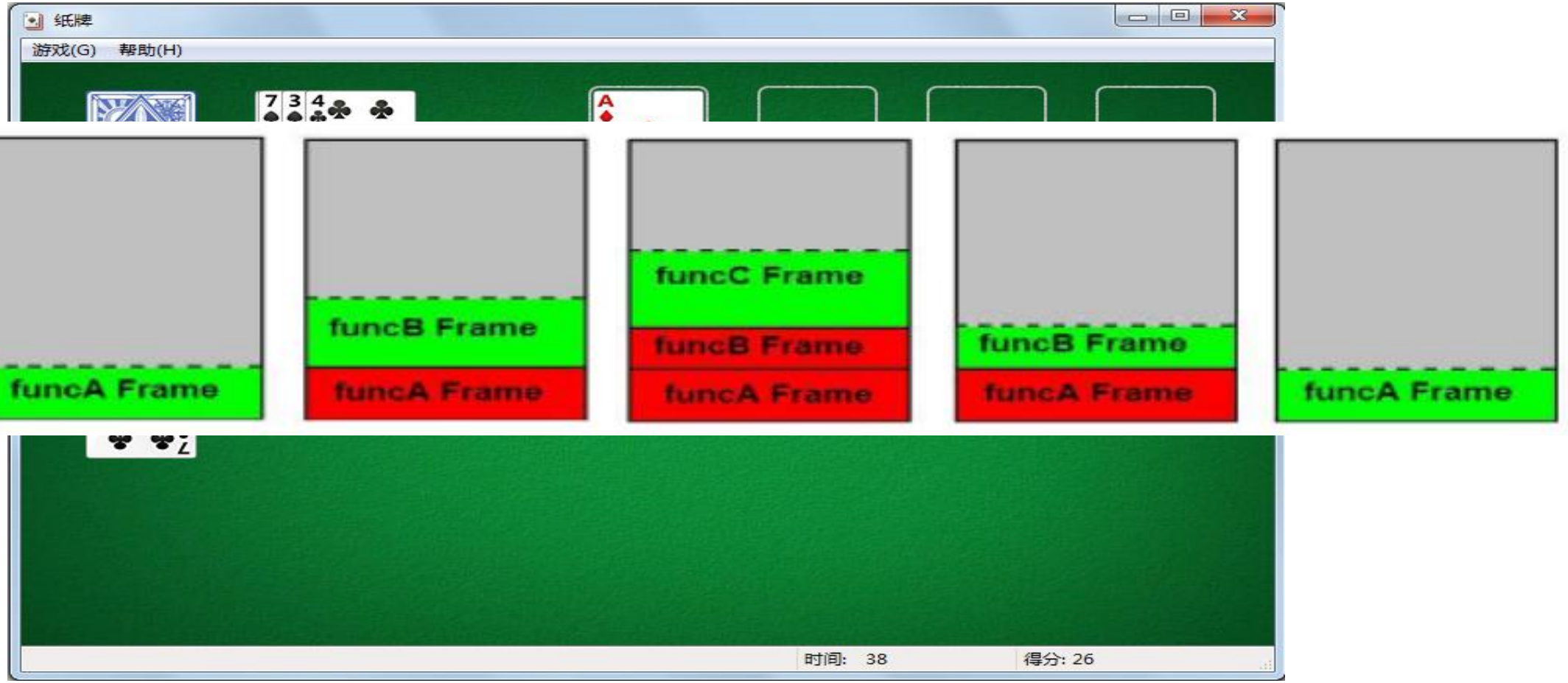
概念

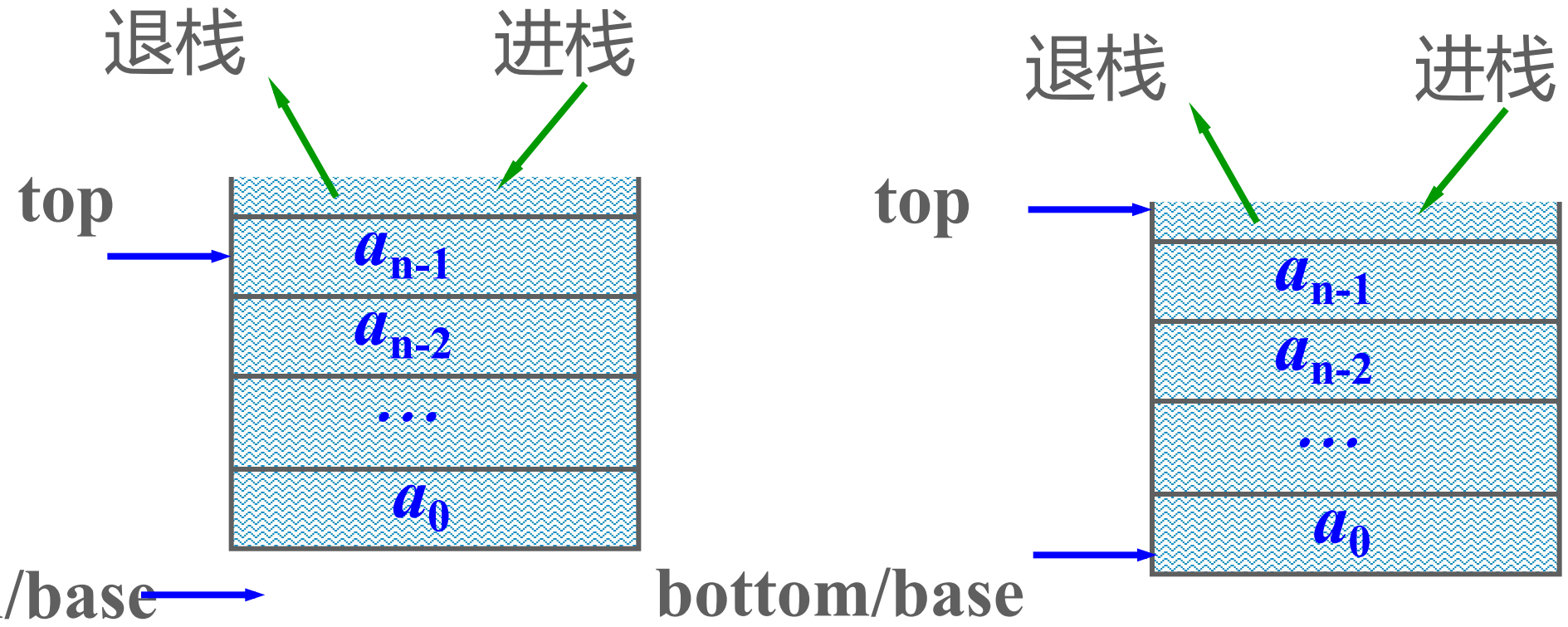
- 栈(Stack)：是限制在表的一端进行插入和删除操作的线性表。
- 逻辑结构是线性结构
- 对特定操作进行限定，其他方面均与线性表类似，因此称为限定性线性表

- 栈顶(Top)：
  允许进行插入、删除操作的一端
  用栈顶指针/游标(top)来指示栈顶。
- 栈底(Bottom)：
  是固定端。



- 特点：后进先出LIFO (Last In First Out)/先进后出FILO (First In Last Out)





操作要实现push入（进）栈与pop出（退）栈，代替线性表中的insert与delete两个操作。push与pop其实也可以视为是特殊/限定的insert与delete操作。

栈的长度：　栈中元素个数

空栈：　栈的长度为0

栈顶元素：　栈中第一个元素

入(进)栈：　向栈中加入元素

出(退)栈：　从栈中删除元素

ADT定义

ADT Stack

{

数据对象：D ={ ai|ai∈ElemSet，i=1,2,…,n，n≥0 }

数据关系：R ={<ai-1, ai>|ai-1，ai∈D，i=2,3,…,n }

基本操作：

初始化、入栈、出栈、清空、

判断栈是否空、栈长度、取栈顶元素

…

}//　　参考教材

顺序存储结构

顺序存储

(动态数组)

```
typedef struct
{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sq_stack;//教材
```

（静态数组）：

```
typedef struct
{
    ElemType data[MAXSIZE];
    int top;
}sq_stack;
```

顺序存储结构

思考：顺序栈，数组两端应设哪端为栈顶哪端为栈底？

```
//操作
Status init(sq_stack*);
Status pop(sq_stack*,ElemType*);
Status push(sq_stack*,ElemType);
Status clear(sq_stack*);
Status destory(sq_stack*);
Status is_empty(sq_stack);
int length(sq_stack);
Status get_top(sq_stack,ElemType*);
```

操作实现

```
Status init(sq_stack *S)
{S.base = (ElemType*)malloc(INIT_SIZE*sizeof(ElemType));
    if(!S.base) exit(OVERFLOW);
    S.top = S.base;
    S.stacksize = INIT_SIZE;
    return OK;
}
```

//该方式栈空为top=base，栈满为top-base>=stacksize

//若采用静态数组顺序表， 栈满为top >= MAXSIZE或MAXSIZE-1。（初始top与base指向0或-1）

操作实现

```
Status push(sq_stack *S, ElemType e)//伪码
{
    检测入栈条件（判断栈容量）
    *(S->top)= e;
    S->top++;
    return OK;
}O（1）    数组尾部插入
```

//画图，看栈顶元素与top指针的关系（重要）

操作实现

```
//入栈条件，判断是否栈满
 if(S.top-S.base >= S.stacksize)
 {
     S.base =
        (ElemType*)realloc(S.base,(Stack_size+INCREM)*
        sizeof(ElemType));
     if(!S.base) exit(OVERFLOW);
     S.top = S.stacksize+S.base;
     S.stacksize += INCREMENT;
 }
```

操作实现

```
Status pop(sq_stack *S,
ElemType *e)
{先检测出栈条件
    S.top--;
    e = *S.top;
    return OK;
}//O(1)
```

```
Status get_top(sq_stack S,
ElemType *e)
{先检测边界条件
    e = *(S.top-1);
    return OK;
}O（1）
```

出栈，看栈顶两个操作，出栈/边界条件为判断是否为空：

```
if(S.top == S.base)        return ERROR;
```

操作实现

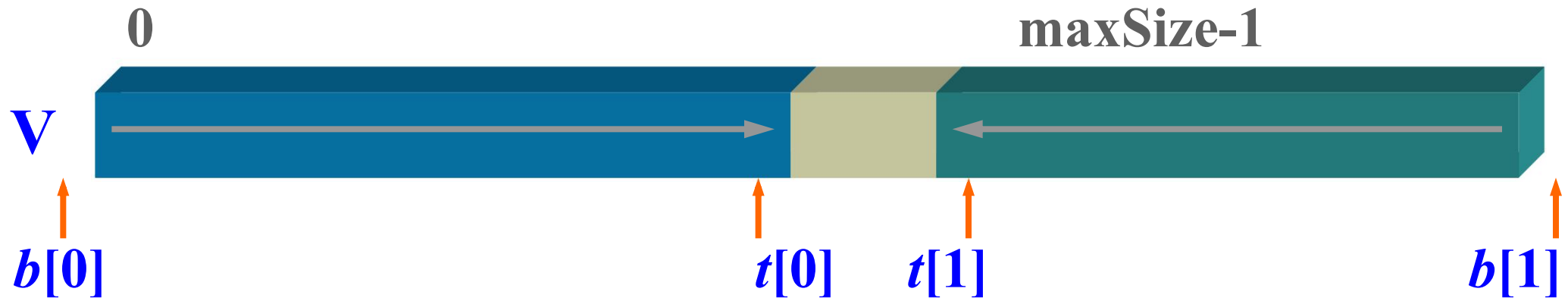
数组描述缺陷：空间利用率低。

如同时有两个栈，可能均有不少未利用的存储空间。

优化：双栈共享一个栈空间

如何做？大家画图

操作实现



两个栈共享一个数组空间V[maxSize]

设立栈顶指针数组 t[2] 和栈底指针数组 b[2]

t[i]和b[i]分别指示第 i 个栈的栈顶与栈底

初始 t[0] = b[0] = -1， t[1] = b[1] = maxSize

栈满条件：t[0]+1 == t[1]　　//栈顶指针相遇

栈空条件：t[0] = b[0]或t[1] = b[1] //退到栈底

链式存储结构

利用单链表
思考：top应该设置在单链表的哪端？

链式存储结构

TOP设为尾元素节点

Push(x)——Insert(n, x)： O(n)

Pop(x)——Delete(n, x)： O(n)

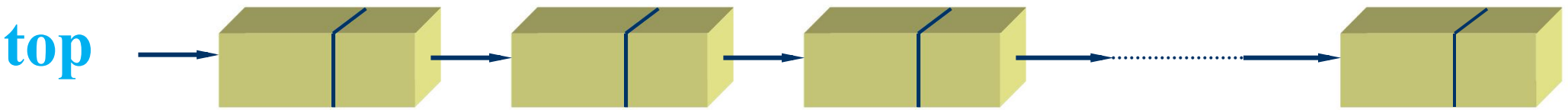
TOP设为首元素节点（头结点）

Push(x)——Insert(1, x)： O(1)

Pop(x)——Delete(1, x)： O(1)

其实，主要是尾部删除元素麻烦

操作实现




```
typedef  struct  stack_node
{
    ElemType    data ;
    struct stack_node  *next ;
} stack_node, *link_stack ;
```

带不带头均可。我们实现不带头结点的，带头结点的自行实现。

```
Status Init ( link_stack* S ) {
        //栈初始化
      //初始化指向或头结点
        return OK;
}//不带头的,其实可以不用init，用init主要是为了与其他形式的各项操作一致
```

操作实现

```
BOOL is_empty (  link_stack S ) {    //判栈空否
    return S == NULL;
}
void Traverse(link_stack S)//遍历
{
    while(S){
        printf("%d\n",S->data);
        S=S->next;
    }
}
```

操作实现

```
void Push (  link_stack* S, ElemType e ) {     //进栈
    stack_node *new_node =
  (stack_node*)malloc(sizeof(stack_node));
   if(!new_node) return OVERFLOW;
   new_node->data = e;  new_node->next = *S;
   *S=new_node;
   return OK;}
//栈顶指针S每次都变化指向
```

操作实现

```
Status pop (link_stack* S, ElemType* e ) { //退栈
    if(!Is_empty)       return ERROR;
  link_stack_node *p = *S;*S = (*S)->next; *e = p->data;
  free(p);  return OK;                    //释放原栈顶结点}

Status get_top( link_stack S, ElemType *e) {  //看栈顶
     if ( is_empty(S)  ) return ERROR;
     *e = S->data;    return OK;}
```

操作实现

对带头结点的栈，可令TOP指向头结点
栈空为TOP->next=NULL
push操作即在TOP节点后插入元素
pop操作即删除TOP节点后元素

实现各项操作也比较简单，课后可自行实现

应用

应用较多

除了P48-58外还有

待队列之后讲部分应用