

§ 3. 栈和队列

3. 1. 栈

3. 1. 1. 栈的定义

栈是一种仅限定在表的一端(表尾)进行插入/删除操作的线性表, 操作按后进先出的原则进行, 称为LIFO结构

★ LIFO = Last In First Out

★ 基本术语 设 $S=(a_1, a_2, \dots, a_n)$

栈顶: 表尾(进行插入/删除的一端) a_n

栈底: 表头 a_1

空栈: 不含任何元素的空表

进栈: 在栈顶增加一个元素, 新元素成为新的栈顶, 栈中元素的个数+1

出栈: 将栈顶元素移除, 次栈顶元素成为新的栈顶, 栈中元素个数-1

上溢: 在栈满时进行进栈操作而产生的错误

下溢: 在栈空时进行出栈操作而产生的错误

★ 栈的形式化定义及基本操作

P. 45

§ 3. 栈和队列

3.1. 栈

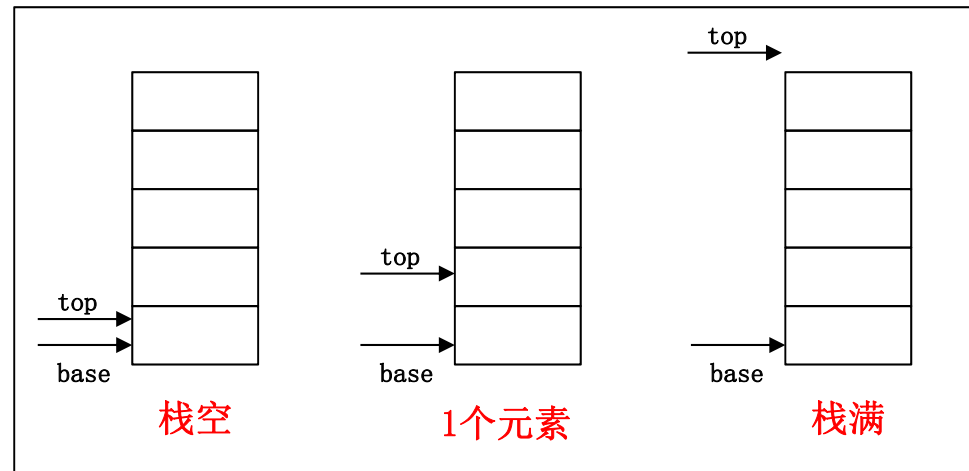
3.1.2. 栈的表示和实现

3.1.2.1. 顺序栈的表示和实现

★ 顺序栈的含义

用一组地址连续的存储单元(数组)依次存放自栈底到栈顶的元素, 同时附设指针base和top, 其中base指向栈底不变, top指示栈顶元素在顺序栈中的位置

- 因为数组使用时必须声明大小, 因此不直接使用数组, 而采用动态内存申请模拟数组的方式, 方便增加大小(同顺序表)
- 考虑插入/删除的效率, 选择下标0做为栈底, 下标n为栈顶
- 考虑一般的习惯, 采用 $top=base$ ($top=0$) 表示空栈, 因此top应始终指向栈顶元素的下一个元素
- 元素进栈后, top指针+1
- 元素出栈后, top指针-1
- top和base指针的差值, 就是之间元素的个数, 即栈的长度



§ 3. 栈和队列

3. 1. 栈

3. 1. 2. 栈的表示和实现

3. 1. 2. 1. 顺序栈的表示和实现

★ 顺序栈的含义

★ 顺序栈的定义与实现

● C语言的定义与实现

/* sqstack.h 的组成 */

/* P.10 的预定义常量和类型 */

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define LOVERFLOW -2 //避免与<math.h>中的定义冲突

typedef int Status;

/* P.46 结构体定义 */

#define STACK_INIT_SIZE 100 //初始大小为100

#define STACKINCREMENT 10 //若空间不够，每次增长10

typedef int SElemType; //可根据需要修改元素的类型

typedef struct {

SElemType *base; //存放动态申请空间的首地址(栈底)

SElemType *top; //栈顶指针

int stacksize; //当前分配的元素个数

} SqStack;

/* sqstack.h 的组成 */

/* P. 46-47的抽象数据类型定义转换为实际的C语言 */

```
Status InitStack(SqStack *S);  
Status DestroyList(SqStack *S);  
Status ClearStack(SqStack *S);  
Status StackEmpty(SqStack S);  
int    StackLength(SqStack S);  
Status GetTop(SqStack S, SElemType *e);  
Status Push(SqStack *S, SElemType e);  
Status Pop(SqStack *S, SElemType *e);  
Status StackTraverse(SqStack S, Status (*visit)(SElemType e));
```

★ C++的引用在C中都表示为指针

/ sqstack.c 的组成 */*

```
#include <stdio.h>
#include <stdlib.h>           //malloc/realloc函数
#include <unistd.h>           //exit函数
#include "sqstack.h"          //形式定义
```

/ 初始化栈 */*

```
Status InitStack(SqStack *S)
{
    S->base = (SElemType *)malloc(STACK_INIT_SIZE * sizeof(SElemType));
    if (S->base == NULL)
        exit(LOVERFLOW);

    S->top = S->base;  //栈顶指针指向栈底，表示栈空
    S->stacksize = STACK_INIT_SIZE;  //置初始大小

    return OK;
}
```

/* sqstack.c 的组成 */

/* 删除栈 */

Status DestroyStack(SqStack *S)

{

/* 若未执行 InitStack，直接执行本函数，则可能出错，
因为指针初始值未定 */

if (S->base)

free(S->base);

S->top = NULL;

S->stacksize = 0;

return OK;

}

/ sqstack.c 的组成 */*

/ 清空栈（已初始化，不释放空间，只清除内容） */*

Status ClearStack(SqStack *S)

{

/ 如果栈曾经扩展过，恢复初始的大小 */*

if (S->stacksize > STACK_INIT_SIZE) {

 S->base = (SElemType *)realloc(S->base,

 STACK_INIT_SIZE * sizeof(SElemType));

 if (S->base==NULL)

 exit(LOVERFLOW); *//一般不会，但仍加上*

 S->stacksize = STACK_INIT_SIZE; *//初始大小*

}

S->top = S->base; *//栈顶指针指向栈底，表示栈空*

return OK;

}

`/* sqstack.c 的组成 */`

`/* 判断是否为空栈 */`

`Status StackEmpty(SqStack S)`

`{`

`if (S.top == S.base)`

`return TRUE;`

`else`

`return FALSE;`

`}`

/* sqstack.c 的组成 */

/* 求栈的长度 */

int StackLength(SqStack S)

{

 return S.top - S.base; // 指针相减，值为相差的元素个数

}

/ sqstack.c 的组成 */*

/ 取栈顶元素 */*

Status GetTop(SqStack S, SElemType *e)

{

 if (S.top == S.base)

 return ERROR; *//空栈则返回*

 *e = *(S.top-1); *//下标从0开始，top是实际栈顶+1*

 return OK;

}

/* sqstack.c 的组成 */

/* 元素入栈 */

Status Push(SqStack *S, SElemType e)

{

/* 如果栈已满，则扩充空间 */

if (S->top - S->base >= S->stacksize) {

S->base = (SElemType *)realloc(S->base,
(S->stacksize+STACKINCREMENT) * sizeof(SElemType));

if (S->base==NULL)

return LOVERFLOW;

//因为S->base可能会变，因此要修正S->top的值

S->top = S->base + S->stacksize;

S->stacksize += STACKINCREMENT; //空间增加

}

S->top++ = e; //先(S->top)，再S->top++

return OK;

}

```
/* sqstack.c 的组成 */
```

```
/* 元素出栈 */
```

```
Status Pop(SqStack *S, SElemType *e)
```

```
{
```

```
    int length;
```

```
    if (S->top == S->base)
```

```
        return ERROR; //空栈则返回
```

```
    *e = *--S->top; //先--S->top, 再*(--S->top)
```

```
/* 如果栈缩小, 则缩小动态申请空间的大小 */
```

```
length = S->top - S->base;
```

```
if (S->stacksize > STACK_INIT_SIZE &&
```

```
    S->stacksize - length >= STACKINCREMENT) {
```

```
    S->base = (SElemType *)realloc(S->base,
```

```
        (S->stacksize-STACKINCREMENT) * sizeof(SElemType));
```

```
    if (S->base==NULL)
```

```
        return LOVERFLOW;
```

```
    S->top = S->base + length; //若S->base改变则修正S->top的值
```

```
    S->stacksize -= STACKINCREMENT;
```

```
}
```

```
return OK;
```

```
}
```

```
/* sqstack.c 的组成 */
```

```
/* 遍历栈 */
```

```
Status StackTraverse(SqStack S, Status (*visit)(SElemType e))
```

```
{
```

```
    extern int line_count; //main中定义的打印换行计数器(与算法无关)
```

```
    SElemType *t = S.base; //栈底指针
```

```
    line_count = 0; //计数器恢复初始值(与算法无关)
```

```
    while(t<S.top && (*visit)(*t)==TRUE)
```

```
        t++;
```

```
    if (t<S.top)
```

```
        return ERROR; //遍历过程有错误，一般不可能出现
```

```
    printf("\n"); //最后打印一个换行，只是为了好看，与算法无关
```

```
    return OK;
```

```
}
```

§ 3. 栈和队列

3. 1. 栈

3. 1. 2. 栈的表示和实现

3. 1. 2. 1. 顺序栈的表示和实现

★ 顺序栈的含义

★ 顺序栈的定义与实现

- C语言的定义与实现
- C++语言的定义与实现

/ sqstack.h 的组成 */*

/ P. 10 的预定义常量和类型 */*

```
#define TRUE          1
#define FALSE        0
#define OK           1
#define ERROR        0
#define INFEASIBLE   -1
#define LOVERFLOW    -2  //避免与<math.h>中的定义冲突
```

```
typedef int Status;
```

/ P. 46 结构体定义 */*

```
#define STACK_INIT_SIZE  100    //初始大小为100
#define STACKINCREMENT   10    //若空间不够，每次增长10

typedef int SElemType;        //可根据需要修改元素的类型
```


/ sqstack.h 的组成 */*

```
class SqStack {  
    protected:  
        SElemType *base;    //存放动态申请空间的首地址(栈底指针)  
        SElemType *top;     //栈顶指针  
        int stacksize;      //当前分配的元素个数  
    public:  
        /* P. 46-47的抽象数据类型定义转换为实际的C++语言 */  
        SqStack();          //构造函数，替代InitStack  
        ~SqStack();         //析构函数，替代DestroyList  
        Status ClearStack();  
        Status StackEmpty();  
        int StackLength();  
        Status GetTop(SElemType &e);  
        Status Push(SElemType e);  
        Status Pop(SElemType &e);  
        Status StackTraverse(Status (*visit)(SElemType e));  
};
```

/ sqstack.cpp 的组成 */*

```
#include <iostream>
#include <stdlib.h>           //malloc/realloc函数
#include <unistd.h>           //exit函数
#include <string.h>           //memcpy
#include "sqstack.h"          //形式定义
using namespace std;
```

/ 构造函数（初始化栈） */*

```
SqStack::SqStack()
{
    base = new SElemType[STACK_INIT_SIZE];
    if (base == NULL)
        exit(LOVERFLOW);
    top = base; //栈顶指针指向栈底，表示栈空
    stacksize = STACK_INIT_SIZE;
}
```

```
/* sqstack.cpp 的组成 */
```

```
/* 析构函数（删除栈） */
```

```
SqStack::~~SqStack()
```

```
{
```

```
    /* 析构函数，不可能出现未初始化就调用的情况 */
```

```
    if (base)
```

```
        delete base;
```

```
    top = NULL;
```

```
    stacksize = 0;
```

```
}
```

```
/* sqstack.cpp 的组成 */
/* 清空栈（已初始化，不释放空间，只清除内容） */
Status SqStack::ClearStack()
{
    /* 如果栈曾经扩展过，恢复初始的大小 */
    if (stacksize > STACK_INIT_SIZE) {
        /* 释放原空间并申请 */
        delete base;
        base = new SElemType[STACK_INIT_SIZE];
        if (base == NULL)
            exit(LOVERFLOW);
        stacksize = STACK_INIT_SIZE; // 恢复初始大小
    }

    top = base; // 栈顶指针指向栈底，表示栈空
    return OK;
}
```

`/* sqstack.cpp 的组成 */`

`/* 判断是否为空栈 */`

```
Status SqStack::StackEmpty()
{
    if (top == base)
        return TRUE;
    else
        return FALSE;
}
```

```
/* sqstack.cpp 的组成 */
```

```
/* 求栈的长度 */
```

```
int SqStack::StackLength()
```

```
{
```

```
    return top - base; // 指针相减，值为相差的元素个数
```

```
}
```

`/* sqstack.cpp 的组成 */`

`/* 取栈顶元素 */`

```
Status SqStack::GetTop(SElemType &e)
{
    if (top == base)
        return ERROR; //空栈直接返回

    e = *(top-1); //下标从0开始，top是实际栈顶+1
    return OK;
}
```

/* sqstack.cpp 的组成 */

/* 元素入栈 */

Status SqStack::Push(SElemType e)

{

/* 如果栈已满，则扩充空间 */

if (top - base >= stacksize) {

SElemType *newbase;

newbase = new SElemType[stacksize+STACKINCREMENT];

if (!newbase)

return LOVERFLOW;

/* 原来的listsize个ElemType空间进行复制 */

memcpy(newbase, base, stacksize*sizeof(SElemType));

/* 释放旧空间，base和top指向新空间(自己做realloc的工作) */

delete base;

base = newbase;

top = base + stacksize; //base与原来不同，top也要移动

stacksize += STACKINCREMENT;

}

*top++ = e; //先*top=e，再top++

return OK;

}


```
/* sqstack.cpp 的组成 */
```

```
/* 元素出栈 */
```

```
Status SqStack::Pop(SElemType &e)
```

```
{
```

```
    int length;
```

```
    if (top == base)
```

```
        return ERROR; //空栈直接返回
```

```
    e = *--top; //先--top, 再 e*(--top)
```

```
/* 如果栈缩小, 则缩小动态申请空间的大小 */
```

```
length = top - base;
```

```
if (stacksize > STACK_INIT_SIZE && stacksize - length >= STACKINCREMENT) {
```

```
    SElemType *newbase;
```

```
    /* 申请新空间 */
```

```
    newbase = new SElemType[stacksize - STACKINCREMENT];
```

```
    if (newbase == NULL)
```

```
        return LOVERFLOW;
```

```
    /* 原来的listsize个ElemType空间进行复制(自己做realloc的工作) */
```

```
    memcpy(newbase, base, (stacksize - STACKINCREMENT) * sizeof(SElemType));
```

```
    /* 释放旧空间, base和top指向新空间 */
```

```
    delete base;
```

```
    base = newbase;
```

```
    top = base + length; //base变化, 修正top的值
```

```
    stacksize -= STACKINCREMENT;
```

```
}
```

```
return OK;
```

```
}
```

```
/* sqstack.cpp 的组成 */
```

```
/* 遍历栈 */
```

```
Status SqStack::StackTraverse(Status (*visit)(SElemType e))
```

```
{
```

```
    extern int line_count; //main中定义的打印换行计数器(与算法无关)
```

```
    SElemType *t = base; //栈底指针
```

```
    line_count = 0; //计数器恢复初始值(与算法无关)
```

```
    while(t<top && (*visit)(*t)==TRUE)
```

```
        t++;
```

```
    if (t<top)
```

```
        return ERROR; //遍历过程有错误，一般不可能出现
```

```
    cout << endl; //最后打印一个换行，只是为了好看，与算法无关
```

```
    return OK;
```

```
}
```

§ 3. 栈和队列

3.1. 栈

3.1.2. 栈的表示和实现

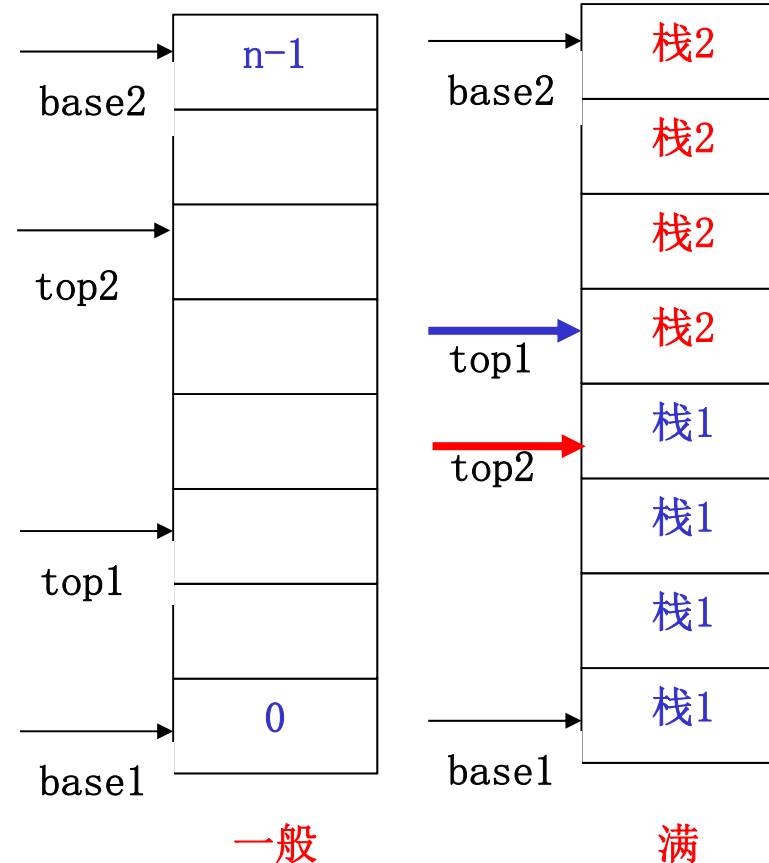
3.1.2.1. 顺序栈的表示和实现

★ 两个顺序栈共享空间

当需要使用多个栈时，为了提高效率，减少空间浪费而使两个栈共享存储空间

- 适用于某些无动态内存申请的高级程序设计语言

- 栈1进栈则 $top1++$ ，
栈2进栈则 $top2--$ ，
 $top2+1==top1$ (已交叉) 则栈满



§ 3. 栈和队列

3. 1. 栈

3. 1. 2. 栈的表示和实现

3. 1. 2. 2. 链栈的表示和实现

★ 链栈的含义

用地址不连续的存储单元来存放元素，通过指针域指向下一个元素

- 为操作方便，插入和删除均在表头进行

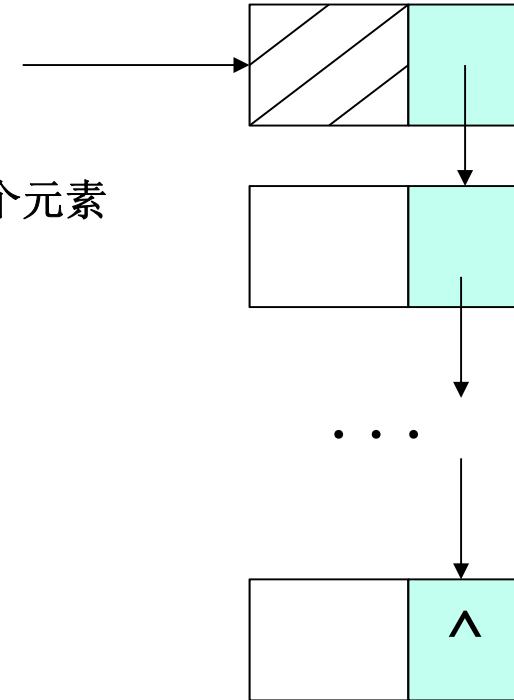
(next指向前一元素，与链表含义相反)

- 可以用带头/不带头结点的单链表来实现

(单链表，只进行删除首元和插入新首元的操作)

★ 链栈的定义与实现

- C语言的定义与实现



`/* linkstack.h 的组成 */`

`/* P.10 的预定义常量和类型 */`

```
#define TRUE          1
#define FALSE        0
#define OK           1
#define ERROR        0
#define INFEASIBLE   -1
#define LOVERFLOW    -2  //避免与<math.h>中的定义冲突
```

```
typedef int Status;
```

`/* 结构体定义 */`

```
typedef int SElemType;    //可根据需要修改元素的类型
```

```
typedef struct LNode {
    SElemType    data;    //存放数据
    struct LNode *next;    //存放直接后继的指针
} LNode, *LinkStack;
```

/* linkstack.h 的组成 */

/* P. 46-47的抽象数据类型定义转换为实际的C语言 */

```
Status InitStack(LinkStack *S);
Status DestroyStack(LinkStack *S);
Status ClearStack(LinkStack *S);
Status StackEmpty(LinkStack S);
int StackLength(LinkStack S);
Status GetTop(LinkStack S, SElemType *e);
Status Push(LinkStack *S, SElemType e);
Status Pop(LinkStack *S, SElemType *e);
Status StackTraverse(LinkStack S, Status (*visit)(SElemType e));
```

★ C++的引用在C中都表示为指针

/ linkstack.c 的组成 */*

#include <stdio.h>

#include <stdlib.h> //malloc/realloc函数

#include <unistd.h> //exit函数

#include "linkstack.h" //形式定义

/ 初始化栈 */*

Status InitStack(LinkStack *S)

{

/ 申请头结点空间，赋值给头指针 */*

*S = (LinkStack)malloc(sizeof(LSNode));

if (*S == NULL)

exit(LOVERFLOW);

(*S)->next = NULL; *//头结点next域为NULL*

return OK;

}

```
/* linkstack.c 的组成 */
/* 销毁栈 */
Status DestroyStack(LinkStack *S)
{
    LinkStack q, p = *S; //指向头结点

    /* 整个链表(含头结点)依次释放 */
    while(p) {
        q = p->next; //抓住链表的下一个结点
        free(p);
        p = q;
    }

    (*S) = NULL; //头指针置NULL
    return OK;
}
```



```
/* linkstack.c 的组成 */
/* 清空栈（保留头结点） */
Status ClearStack(LinkStack *S)
{
    LinkStack q, p = (*S)->next; //指向首元结点

    /* 从首元结点开始依次释放 */
    while(p) {
        q = p->next; //抓住链表的下一个结点
        free(p);
        p = q;
    }

    (*S)->next = NULL; //头结点的next域置NULL
    return OK;
}
```

`/* linkstack.c 的组成 */`

`/* 判断是否为空栈 */`

`Status StackEmpty(LinkStack S)`

`{`

`/* 判断头结点的next域即可 */`

`if (S->next==NULL)`

`return TRUE;`

`else`

`return FALSE;`

`}`

`/* linkstack.c 的组成 */`

`/* 求栈的长度 */`

```
int StackLength(LinkStack S)
{
    int    len = 0;
    LinkStack p = S->next; //指向首元结点

    while(p) {
        len++;
        p = p->next;
    }

    return len;
}
```

/* linkstack.c 的组成 */

/* 取栈顶元素 */

Status GetTop(LinkStack S, SElemType *e)

{

 if (S->next==NULL)

 return ERROR; //无首元则返回

/* 取首元结点的值 */

 *e = S->next->data;

 return OK;

}

/ linkstack.c 的组成 */*

/ 元素入栈 */*

```
Status Push(LinkStack *S, SElemType e)
{
```

/ 申请空间 */*

```
LinkStack p;
```

```
p = (LinkStack)malloc(sizeof(LSNode));
```

```
if (p==NULL)
```

```
    return LOVERFLOW;
```

/ 给新结点复制并插入为首元 */*

```
p->data = e;
```

```
p->next = (*S)->next;
```

```
(*S)->next = p;
```

```
return OK;
```

```
}
```

```
/* linkstack.c 的组成 */
```

```
/* 元素出栈 */
```

```
Status Pop(LinkStack *S, SElemType *e)
```

```
{
```

```
    LinkStack p;
```

```
    if ((*S)->next==NULL)
```

```
        return ERROR; //空栈直接返回
```

```
    /* 元素出栈，释放结点空间 */
```

```
    p = (*S)->next;
```

```
    (*S)->next = p->next;
```

```
    *e = p->data;
```

```
    free(p);
```

```
    return OK;
```

```
}
```

```
/* linkstack.c 的组成 */
```

```
/* 遍历栈 */
```

```
Status StackTraverse(LinkStack S, Status (*visit)(SElemType e))
```

```
{
```

```
    extern int line_count; //main中定义的打印换行计数器(与算法无关)
```

```
    LinkStack p = S->next;
```

```
    line_count = 0;           //计数器恢复初始值（与算法无关）
```

```
    while(p && (*visit)(p->data)==TRUE)
```

```
        p=p->next;
```

```
    if (p)
```

```
        return ERROR;
```

```
    printf("\n"); //最后打印一个换行，只是为了好看，与算法无关
```

```
    return OK;
```

```
}
```

§ 3. 栈和队列

3.1. 栈

3.1.2. 栈的表示和实现

3.1.2.2. 链栈的表示和实现

★ 链栈的含义

用地址不连续的存储单元来存放元素，通过指针域指向下一个元素

- 为操作方便，插入和删除均在表头进行

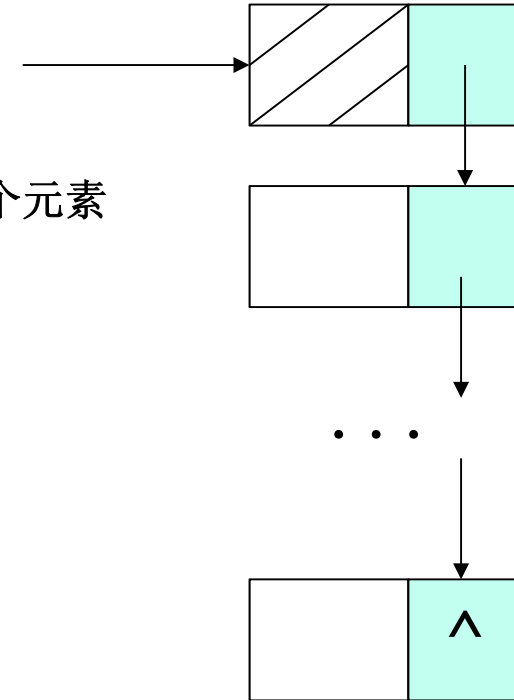
(next指向前一元素，与链表含义相反)

- 可以用带头/不带头结点的单链表来实现

(单链表，只进行删除首元和插入新首元的操作)

★ 链栈的定义与实现

- C语言的定义与实现
- C++语言的定义与实现




```
/* linkstack.h 的组成 */
/* P.10 的预定义常量和类型 */
#define TRUE          1
#define FALSE        0
#define OK           1
#define ERROR        0
#define INFEASIBLE   -1
#define LOVERFLOW    -2  //避免与<math.h>中的定义冲突

typedef int Status;

typedef int SElemType;    //可根据需要修改元素的类型

class LinkStack; //提前声明，因为定义友元要用到
class LNode {
    protected:
        SElemType data; //数据域
        LNode *next;    //指针域
    public:
        friend class LinkStack;
        //不定义任何函数，相当于struct LNode
};
```

/* linkstack.h 的组成 */

```
class LinkStack {
    protected:
        LSNode *head;    //头指针
    public:
        /* P. 46-47的抽象数据类型定义转换为实际的C++语言 */
        LinkStack();      //构造函数，替代InitStack
        ~LinkStack();     //析构函数，替代DestroyStack
        Status ClearStack();
        Status StackEmpty();
        int StackLength();
        Status GetTop(SElemType &e);
        Status Push(SElemType e);
        Status Pop(SElemType &e);
        Status StackTraverse(Status (*visit)(SElemType e));
};
```

```
/* linkstack.cpp 的组成 */
```

```
#include <iostream>
```

```
#include <stdlib.h> //malloc/realloc函数
```

```
#include <unistd.h> //exit函数
```

```
#include "linkstack.h" //形式定义
```

```
using namespace std;
```

```
/* 初始化栈 */
```

```
LinkStack::LinkStack()
```

```
{
```

```
    /* 申请头结点空间，赋值给头指针 */
```

```
    head = new LSNode;
```

```
    if (head == NULL)
```

```
        exit(LOVERFLOW);
```

```
    head->next = NULL;
```

```
}
```

```
/* linkstack.cpp 的组成 */
```

```
/* 销毁栈 */
```

```
LinkStack::~LinkStack()
```

```
{
```

```
    LSNODE *q, *p = head; //指向头结点
```

```
    /* 整个链表(含头结点)依次释放 */
```

```
    while(p) {
```

```
        q=p->next; //抓住链表的下一个结点
```

```
        delete p;
```

```
        p=q;
```

```
    }
```

```
    head = NULL; //头指针置NULL
```

```
}
```

```
/* linkstack.cpp 的组成 */
/* 清空栈（保留头结点） */
Status LinkStack::ClearStack()
{
    LSNode *q, *p = head->next; //指向首元结点
    /* 从首元结点开始依次释放 */
    while(p) {
        q = p->next; //抓住链表的下一个结点
        delete p;
        p = q;
    }

    head->next = NULL; //头结点的next域置NULL
    return OK;
}
```

```
/* linkstack.cpp 的组成 */
```

```
/* 判断是否为空栈 */
```

```
Status LinkStack::StackEmpty()
```

```
{
```

```
    /* 判断头结点的next域即可 */
```

```
    if (head->next==NULL)
```

```
        return TRUE;
```

```
    else
```

```
        return FALSE;
```

```
}
```

`/* linkstack.cpp 的组成 */`

`/* 求栈的长度 */`

```
int LinkStack::StackLength()
{
    int    len = 0;
    LSNode *p = head->next; //指向头结点

    while(p) {
        len++;
        p = p->next;
    }
    return len;
}
```

```
/* linkstack.cpp 的组成 */
```

```
/* 取栈顶元素 */
```

```
Status LinkStack::GetTop(SElemType &e)  
{
```

```
    if (head->next==NULL)
```

```
        return ERROR;    //无首元则返回
```

```
/* 取首元结点 */
```

```
e = head->next->data;
```

```
return OK;
```

```
}
```



```
/* linkstack.cpp 的组成 */  
/* 元素入栈 */  
Status LinkStack::Push(SElemType e)  
{  
    /* 申请新结点 */  
    LNode *p;  
    p = new LNode;  
    if (p==NULL)  
        return OVERFLOW;  
  
    /* 加入成为新首元 */  
    p->data = e;  
    p->next = head->next;  
    head->next = p;  
  
    return OK;  
}
```

```
/* linkstack.cpp 的组成 */  
/* 元素出栈 */  
Status LinkStack::Pop(SElemType &e)  
{  
    LSNode *p;  
    if (head->next==NULL)  
        return ERROR; //空栈直接返回  
  
    /* 首元出栈并释放 */  
    p = head->next;  
    head->next = p->next;  
    e = p->data;  
    delete p;  
  
    return OK;  
}
```

```
/* linkstack.cpp 的组成 */
/* 遍历栈 */
Status LinkStack::StackTraverse(Status (*visit)(SElemType e))
{
    extern int line_count; //main中定义的打印换行计数器(与算法无关)
    LSNode *p = head->next;

    line_count = 0;          //计数器恢复初始值（与算法无关）
    while(p && (*visit)(p->data)==TRUE)
        p=p->next;

    if (p)
        return ERROR;

    cout << endl; //最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```

§ 3. 栈和队列

3.1. 栈

3.1.2. 栈的表示和实现

3.1.2.2. 链栈的表示和实现

★ 链栈的含义

用地址不连续的存储单元来存放元素，通过指针域指向下一个元素

- 为操作方便，插入和删除均在表头进行
(next指向前一元素，与链表含义相反)
- 可以用带头/不带头结点的单链表来实现
(单链表，只进行删除首元和插入新首元的操作)

★ 链栈的定义与实现

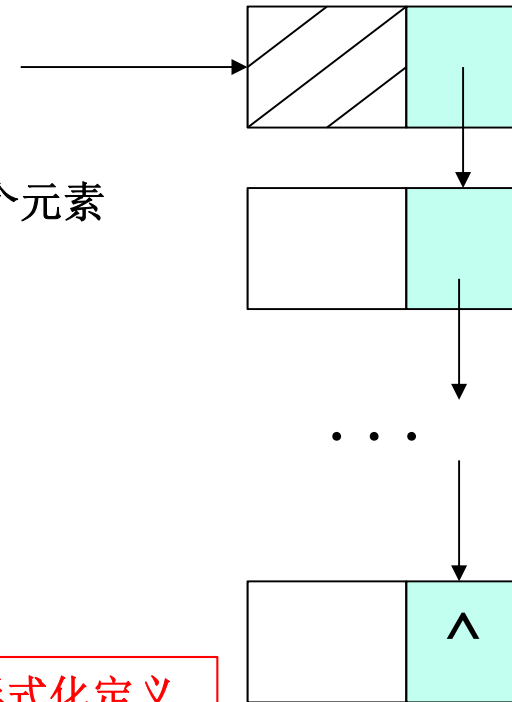
- C语言的定义与实现
- C++语言的定义与实现

★ 以上两种实现方式，StackTraverse均是从栈顶到栈底，和形式化定义以及顺序栈是反向的，如何改为从栈底到栈顶遍历？

- 单链表 : 可以，时间复杂度为 $O(n^2)$
 - 单循环链表 : 可以，时间复杂度为 $O(n^2)$
 - 带尾指针的单循环链表: 可以，时间复杂度为 $O(n^2)$
- 假设上面三个单向链表都是头结点指向栈顶
- 双向循环链表 : 可以，时间复杂度为 $O(n)$

结论:

用单链表表示链栈且StackTraverse的时间复杂度为 $O(n)$ ，必须头结点指向栈底，且pop的时间复杂度为 $O(n)$



§ 3. 栈和队列

3.2. 栈的应用举例

3.2.1. 数制转换(十进制转其它进制)

规则：连除，**逆向**取余

$$N = (N \text{ div } d) \times d + N \text{ mod } d$$

其中：N - 十进制数

d - 其它进制

div - 整除

mod - 求模

```
void conversion() //P.48 算法3.1
{
    InitStack(S);
    cin >> N >> d; //读入十进制数及要转换的进制(书：1个)
    while(N) {
        Push(S, N%d); //此处是d进制(书：8进制)
        N = N/d;       //此处是d进制(书：8进制)
    }
    while(!StackEmpty(S)) {
        Pop(S, e);
        cout << e;
    }
}
```

如果16进制，输出是
10-15变为A-F即可

§ 3. 栈和队列

3.2. 栈的应用举例

3.2.1. 数制转换(十进制转其它进制)

3.2.2. 括号匹配的检查

规则: (1) 右括号与上面最近出现的左括号匹配
(2) 多种括号不能交叉(即不允许 `([)]`)

例: 假设键盘输入的一行中包含小括号、中括号
以及其它字符, 其它字符的合理性不做检查,
仅要求检查括号的匹配

★ 栈中元素类型应为char型

```
Status match()
{
    InitStack(S);
    while((ch=getchar())!='\n') {
        switch(ch) {
            case '(':
                Push(S, '(');
                break;
            case '[':
                Push(S, '[');
                break;
            case ')':
                if (Pop(S, e)==OK)
                    if (e!='(')
                        return ERROR;
                    else
                        break;
                else //pop错, 可能是栈空
                    return ERROR;
            case ']':
                if (Pop(S, e)==OK)
                    if (e!='[')
                        return ERROR;
                    else
                        break;
                else //pop错, 可能是栈空
                    return ERROR;
            default:
                break;
        } //end of switch
    } //end of while
    if (StackEmpty()) //栈空
        return OK;
    else
        return ERROR;
}
```

§ 3. 栈和队列

3.2. 栈的应用举例

3.2.1. 数制转换(十进制转其它进制)

3.2.2. 括号匹配的检查

3.2.3. 行编辑程序

规则：键盘输入一行

- 退格

@ - 本行作废

例：whli##ilr#e(s#*s)

=> while(*s)

outcha@putchar(*s=#++);

=> putchar(*s++);

★ 正常输入中不会出现#和@

```
void LineEdit()
{
    InitStack(S); //初始化
    ch = getchar();
    while(ch != EOF) { //外层while持续读到输入结束(EOF)
        while(ch!=EOF && ch!='\n') { //内层while读一行
            switch(ch) {
                case '#':
                    Pop(S, c); //即使pop错(栈空)也继续
                    break;
                case '@':
                    ClearStack(S);
                    break;
                default:
                    Push(S, ch);
            } //end of switch
            ch = getchar();
        }
        StackTraverse(S); //从栈底到栈顶，正好是该行
        ClearStack(S); //清空栈，准备读下一行
        if (ch!=EOF)
            ch = getchar();
    }
    DestroyStack(S); //全部读完，销毁栈
}
```

§ 3. 栈和队列

3.2. 栈的应用举例

3.2.1. 数制转换(十进制转其它进制)

3.2.2. 括号匹配的检查

3.2.3. 行编辑程序

3.2.4. 迷宫求解(略, 后面第7章图遍历时有类似方法)

3.2.5. 表达式求值(参见前面的课程)

§ 2. 数据类型与表达式

2.8. C++的表达式求值(补充)

表达式：由若干操作数和操作符构成的符合C++语法的算式

表达式的求值：整个表达式从左到右依次分析，分析原则如下：

★ 若只有一个运算符，则求值

$a+b$

★ 若有两个运算符，若

① 左边运算符的优先级高于右边运算符 或

② 左边运算符的优先级等于右边运算符，且该级别运算符是左结合，

则对左边运算符求值，其值再参与后续的运算

$a*b+c$

$a+b+c$

★ 若有两个运算符，若

① 左边运算符的优先级低于右边运算符 或

② 左边运算符的优先级等于右边运算符，且该级别运算符是右结合，

则先忽略左边运算符，继续向后分析，直到右边运算符被求值后再次分析左边运算符

$a+=a-=a*a$

$a+b*c$

$a+b*c-d$

$a+b*c*d$

§ 2. 数据类型与表达式

2.8. C++的表达式求值(补充)

★ 用栈(LIFO)的形式理解表达式求值过程

运算数栈：存放运算数

运算符栈：存放运算符

规则：(1) 运算数/运算符分别进各自的栈

(2) 若欲进栈的运算符级别高于或等于(右结合)栈顶运算符，则进栈

(3) 若欲进栈的运算符级别低于或等于(左结合)栈顶运算符，则先将栈顶运算符计算完成，计算数为运算数栈的两个元素

(4) 重复上述步骤至运算符栈为空，运算数栈只有一个元素，否则认为语法错

10+' a' +i*f-d/e (P.33 解释)

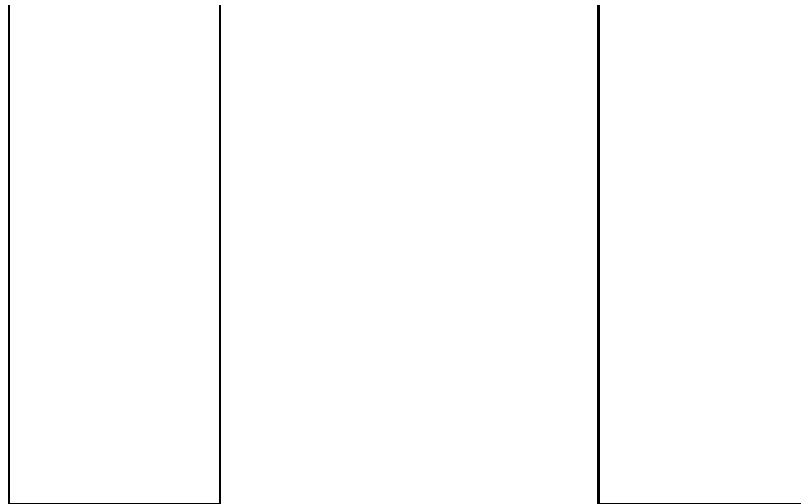
- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行



初始：两栈均为空

10+'a'+i*f-d/e (P.33 解释)

- ① 10+'a' int + int
- ② i*f double * double
- ③ ①+② double + double
- ④ d/e double / double
- ⑤ ③-④ double - double

注意：不是将整个算式中优先级最高的最先计算，而是分步进行



10进栈

10+'a'+i*f-d/e (P.33 解释)

- ① 10+'a' int + int
- ② i*f double * double
- ③ ①+② double + double
- ④ d/e double / double
- ⑤ ③-④ double - double

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

10

+

+进栈

10+'a'+i*f-d/e (P.33 解释)

- ① 10+'a' int + int
- ② i*f double * double
- ③ ①+② double + double
- ④ d/e double / double
- ⑤ ③-④ double - double

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

'a'
10

+

'a' 进栈

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

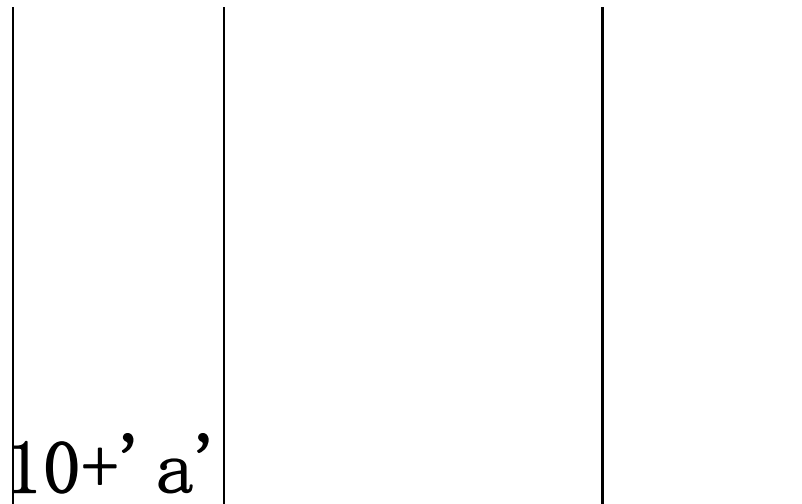


要进栈的(+)等于栈顶(+), 且左结合, 求值

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行



要进栈的(+)等于栈顶(+), 且左结合, 求值

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

10+' a'

+

+进栈

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

i
10+' a'

i进栈

+

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

i
10+' a'

*
+

*进栈 (要进栈的*高于栈顶的+)

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

f
i
10+' a'

f进栈

*

+

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

f
i
10+' a'

*
+

要进栈的(-)低于栈顶的(*), 先计算

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

i*f
10+' a'

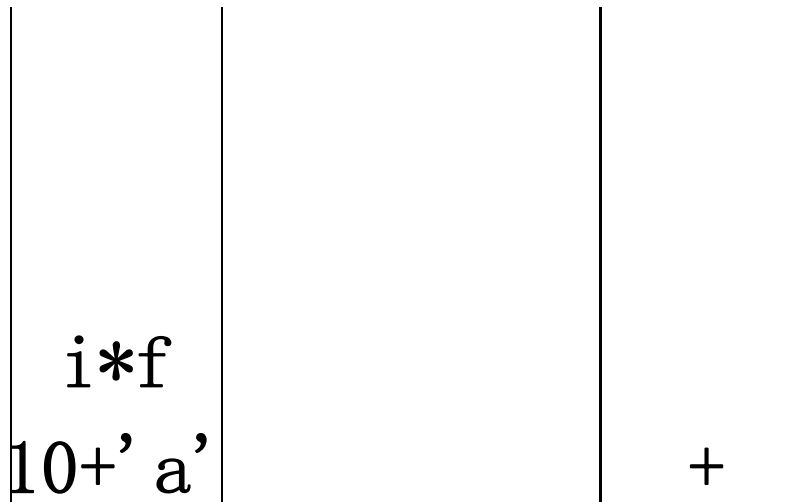
+

要进栈的(-)低于栈顶的(*), 先计算

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

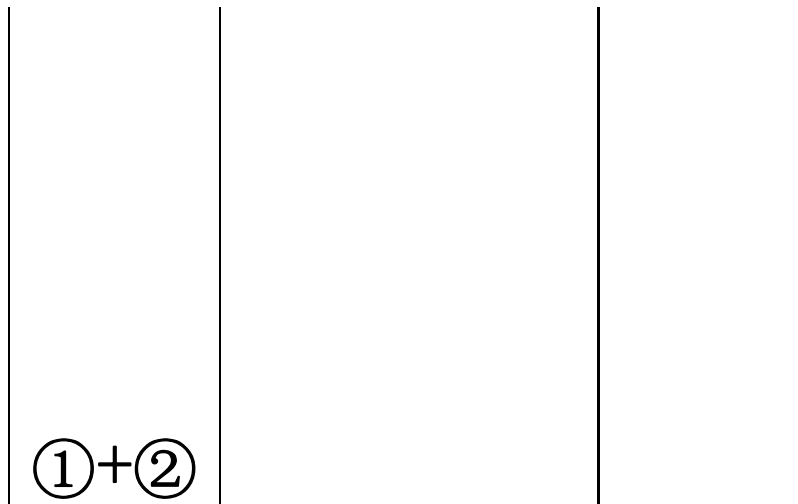


要进栈的(-)等于栈顶的(+), 左结合, 先计算

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行



要进栈的(-)等于栈顶的(+), 左结合, 先计算

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

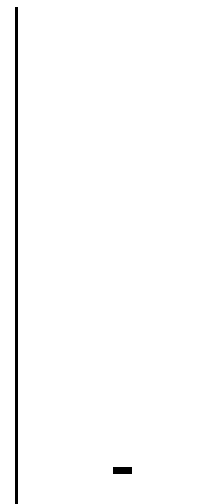
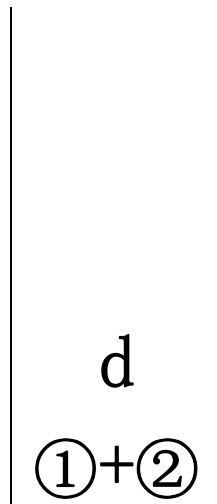


-进栈

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

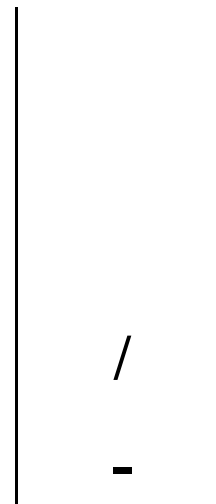
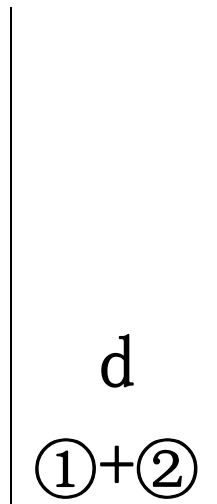


d进栈

10+' a' +i*f-d/e (P.33 解释)

- ① 10+' a' int + int
- ② i*f double * double
- ③ ①+② double + double
- ④ d/e double / double
- ⑤ ③-④ double - double

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

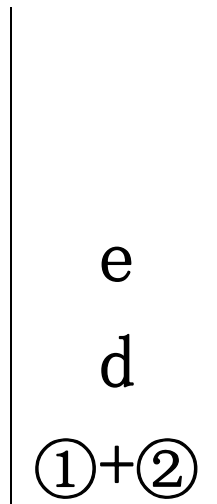


/进栈 (要进栈的/高于栈顶的-)

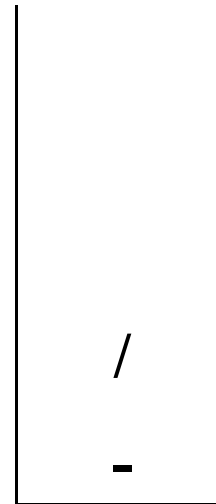
10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行



e进栈



10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

d/e
①+②

-

计算 d/e

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

③-④

计算 ③-④

10+' a' +i*f-d/e (P.33 解释)

- | | | |
|---|---------|-----------------|
| ① | 10+' a' | int + int |
| ② | i*f | double * double |
| ③ | ①+② | double + double |
| ④ | d/e | double / double |
| ⑤ | ③-④ | double - double |

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

③-④

表达式分析并求值完成

转为算法时需要考虑的问题：

- 1、如何判断表达式结束
- 2、如何处理各种错误情况
- 3、如何进行表达式求值
(例：-左右不能互换)
- 4、表达式含括号如何处理
- 5、含单目运算符如何处理
- 6、运算符二义性如何处理
(例：-负号/减)

§ 3. 栈和队列

3.3. 栈和递归的实现(参见前面的课程)

§ 4. 函数与预处理

4.5. 函数的嵌套调用

4.5.1. C++程序的执行过程

- (1) 从main函数的第一个执行语句开始依次执行
- (2) 若执行到函数调用语句，则保存调用函数当前的一些系统信息(保存现场)
- (3) 转到被调用函数的第一个执行语句开始依次执行
- (4) 被调用函数执行完成后，返回到调用函数的调用处，恢复调用前保存的系统信息(恢复现场)
- (5) 若被调用函数中仍有调用其它函数的语句，则嵌套执行步骤(2) - (4)
- (6) 所有被调用函数执行完后，顺序执行main函数的后续部分直到结束

P. 56 第2行起

保存现场的工作：

- 1、将所有的实在参数、返回地址等信息传递给被调用函数保存
- 2、为被调用函数的局部变量分配存储区
- 3、将控制转移到被调用函数的入口

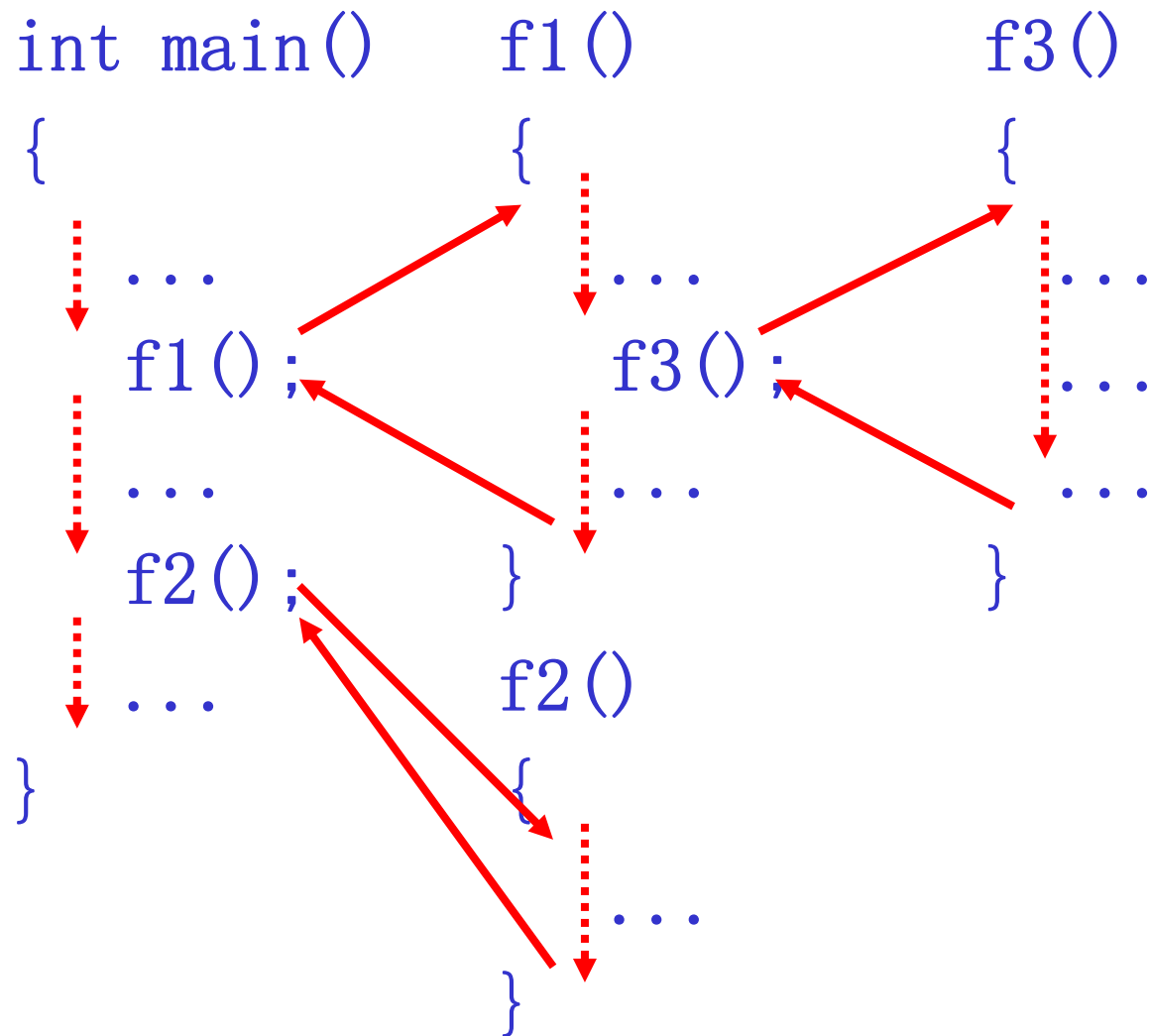
恢复现场的工作：

- 1、保存被调用函数的计算结果
- 2、释放被调用函数的数据区
- 3、依照被调函数保存的返回地址将控制转移到调用函数

§ 4. 函数与预处理

4. 5. 函数的嵌套调用

4. 5. 1. C++程序的执行过程



§ 4. 函数与预处理

4.5. 函数的嵌套调用

4.5.1. C++程序的执行过程

4.5.2. 特点

- ★ 嵌套的层次、位置不限
- ★ 遵循**后进先出**的原则(栈)
- ★ 调用函数时，被调用函数与其所调用的函数的关系是透明的，适用于大程序的分工组织

P. 56 第6行起

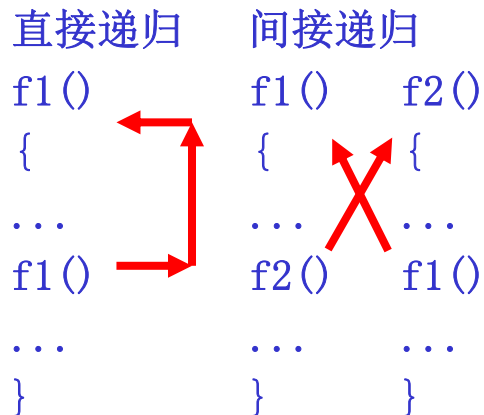
当有多个函数构成嵌套调用时，按照“后调用先返回”的原则，上述函数之间的信息传递和控制转移必须通过“栈”来实现，即系统将整个程序运行时所需的数据空间安排在一个栈中，每当调用一个函数时，就为它在栈顶分配一个存储区，每当从一个函数退出时，就释放它的存储区，则当前正运行函数的数据区必在栈顶。

§ 4. 函数与预处理

4.6. 函数的递归调用

4.6.1. 含义

函数直接或间接地调用本身



必然有条件判断是否进行下次递归调用!!!

4.6.2. 递归的求解过程

回推：到一个确定值为止(递归不再调用)

递推：根据回推得到的确定值求出要求的解

4.6.3. 使用

确定终止条件

P. 56 最后一段起

一个递归函数的运行过程类似于多个函数的嵌套调用，只是调用函数和被调用函数是同一个函数，因此，和每次调用相关的一个重要的概念是递归函数运行的“层次”。

P. 57 第1行

为了保证递归函数正确执行，系统需设立一个“递归工作栈”做为整个递归函数运行期间使用的数据存储区，每一层递归所需信息构成一个“工作记录”，其中包括所有的实在参数、所有的局部变量以及上一层的返回地址。每进入一层递归，就产生一个新的工作记录压入栈顶。每退出一层递归，就从栈顶弹出一个工作记录，则当前执行层的工作记录必是递归工作栈栈顶的工作记录，称这个记录为“活动记录”，并称指示活动记录的栈顶指针为“当前环境指针”。

§ 4. 函数与预处理

4. 6. 函数的递归调用

请参考C++第4章PDF中关于递归函数理解的例子

例1: 写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
}
```

例2: 写出程序的运行结果

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例3: 写出程序的运行结果及功能

```
void f(int n, int k)
{   if (n>=k)
        f(n/k, k);
    cout << n%k;
}
int main()
{   f(14, 2);
    f(65, 8);
}
```

§ 3. 栈和队列

3. 4. 队列

3. 4. 1. 队列的含义

队列是一种仅限在线性表的一端进行插入，另一端进行删除的线性表，按先进先出的原则进行，称为FIFO结构

★ FIFO = First In First Out

★ 基本术语

队尾：允许插入的一端

队头：允许删除的一端

空队列：队列中无元素

入队列：在队尾增加一个元素，新元素成为新的队尾，队列中元素的个数+1

出队列：将队头元素移除，次队头元素成为新的队头，队列中元素个数-1

上溢：在队列满时进行入队列操作而产生的错误

下溢：在队列空时进行出队列操作而产生的错误

★ 队列及形式化定义

P. 59-60

§ 3. 栈和队列

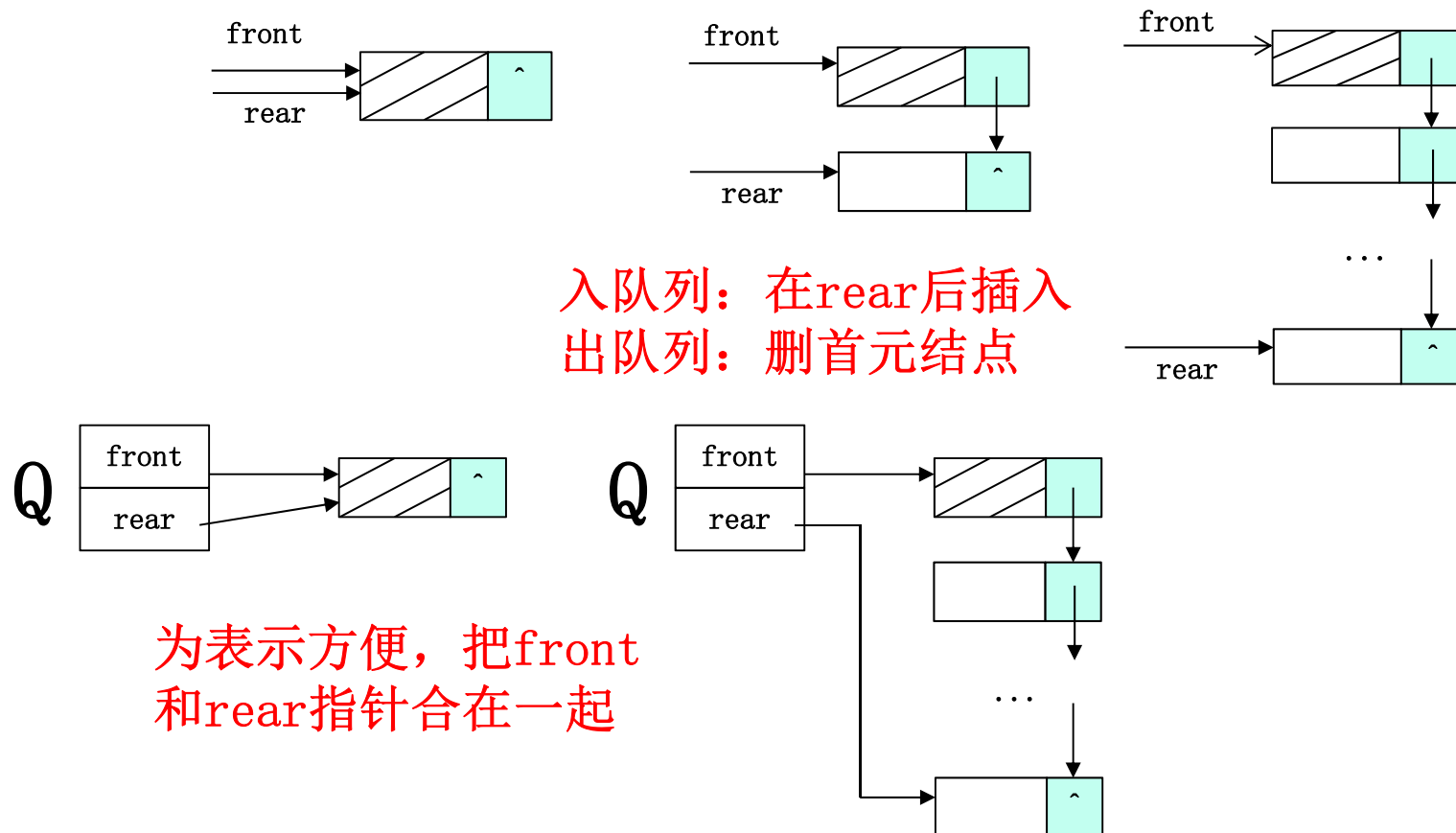
3. 4. 队列

3. 4. 2. 链队列-队列的链式表示与实现

含义：用单链表(带头结点)表示队列，分设两个指针

头指针：指向队头结点，用于删除

尾指针：指向队尾结点，用于插入



§ 3. 栈和队列

3. 4. 队列

3. 4. 2. 链队列-队列的链式表示与实现

含义：用单链表(带头结点)表示队列，分设两个指针

头指针：指向队头结点，用于删除

尾指针：指向队尾结点，用于插入

链队列的定义与实现

★ C语言的定义与实现

```
/* linkqueue.h 的组成 */
/* P.10 的预定义常量和类型 */
#define TRUE          1
#define FALSE         0
#define OK            1
#define ERROR         0
#define INFEASIBLE    -1
#define LOVERFLOW     -2  //避免与<math.h>中的定义冲突

typedef int Status;
/* P.61 形式定义 */
typedef int QElemType;    //可根据需要修改元素的类型

typedef struct QNode {
    QElemType    data;    //存放数据
    struct QNode *next;    //存放直接后继的指针
} QNode, *QueuePtr;

typedef struct {
    QueuePtr front;
    QueuePtr rear;
} LinkQueue;
```

/* linkqueue.h 的组成 */

/* P. 59-60的抽象数据类型定义转换为实际的C语言 */

```
Status InitQueue(LinkQueue *Q);
Status DestroyQueue(LinkQueue *Q);
Status ClearQueue(LinkQueue *Q);
Status QueueEmpty(LinkQueue Q);
int    QueueLength(LinkQueue Q);
Status GetHead(LinkQueue Q, QElemType *e);
Status EnQueue(LinkQueue *Q, QElemType e);
Status DeQueue(LinkQueue *Q, QElemType *e);
Status QueueTraverse(LinkQueue Q, Status (*visit)(QElemType e));
```

/ linkqueue.c 的组成 */*

```
#include <stdio.h>
#include <stdlib.h>           //malloc/realloc函数
#include <unistd.h>           //exit函数
#include "linkqueue.h"       //形式定义
```

/ 初始化队列 */*

```
Status InitQueue(LinkQueue *Q)
{
    /* 申请头结点空间，赋值给头/尾指针 */
    Q->rear = Q->front = (QueuePtr)malloc(sizeof(QNode));
    if (Q->front == NULL)
        exit(LOVERFLOW);

    Q->front->next = NULL; //头结点的next域
    return OK;
}
```

`/* linkqueue.c 的组成 */`

`/* 销毁队列 */`

`Status DestroyQueue(LinkQueue *Q)`

`{`

`/* 整个链表(含头结点)依次释放，没有像链表、栈等`

`借助 QNode *p, *q; 而直接借用了front和rear */`

`while(Q->front) {`

`Q->rear = Q->front->next; //抓住链表的下一个结点`

`free(Q->front);`

`Q->front = Q->rear;`

`}`

`Q->rear = Q->front = NULL;`

`return OK;`

`}`

/ linkqueue.c 的组成 */*

/ 清空队列（保留头结点） */*

Status ClearQueue(LinkQueue *Q)

{

QueuePtr q, p = Q->front->next; *//指向首元*

/ 从首元结点开始依次释放 */*

while(p) {

q = p->next; *//抓住链表的下一个结点*

free(p);

p = q;

}

Q->front->next = NULL; *//头结点的next域置NULL*

Q->rear = Q->front; *//必须要，否则尾指针指向不正确*

return OK;

}

`/* linkqueue.c 的组成 */`

`/* 判断是否为空队列 */`

`Status QueueEmpty(LinkQueue Q)`

`{`

`/* 判断front和rear指针是否相等 */`

`if (Q.front == Q.rear)`

`return TRUE;`

`else`

`return FALSE;`

`}`

`//也可以判断头指针的next域`

`if(Q.front->next==NULL)`

`return TRUE;`

`else`

`return FALSE;`

`/* linkqueue.c 的组成 */`

`/* 求队列的长度 */`

```
int QueueLength(LinkQueue Q)
{
    int    len = 0;
    QueuePtr p = Q.front->next;    //指向首元

    while(p) {
        len++;
        p = p->next;
    }
    return len;
}
```



```
/* linkqueue.c 的组成 */
```

```
/* 取队头元素 */
```

```
Status GetHead(LinkQueue Q, QElemType *e)
```

```
{
```

```
    /* 空队列则返回 */
```

```
    if (Q.front == Q.rear) //也可以用Q.front->next==NULL
```

```
        return ERROR;
```

```
    /* 取首元结点的值 */
```

```
    *e = Q.front->next->data;
```

```
    return OK;
```

```
}
```

/ linkqueue.c 的组成 */*

/ 元素入队列 */*

Status EnQueue(LinkQueue *Q, QElemType e)

{

QueuePtr p;

/ 申请新结点 */*

p = (QueuePtr)malloc(sizeof(QNode));

if (p==NULL)

return LOVERFLOW;

p->data = e;

p->next = NULL; *//新结点的next必为NULL*

Q->rear->next = p; *//接在当前队尾的后面*

Q->rear = p; *//尾指针指向新的队尾*

return OK;

}

```
/* linkqueue.c 的组成 */
```

```
/* 元素出队列 */
```

```
Status DeQueue(LinkQueue *Q, QElemType *e)
```

```
{
```

```
    QueuePtr p;
```

```
    /* 空队列则返回 */
```

```
    if (Q->front == Q->rear)
```

```
        return ERROR;
```

```
    /* 对首元结点操作 */
```

```
    p = Q->front->next;
```

```
    //指向首元
```

```
    Q->front->next = p->next;
```

```
    //front指向新首元(可能NULL)
```

```
    /* 如果只有一个结点，则必须修改尾指针 */
```

```
    if (Q->rear == p)
```

```
        Q->rear = Q->front;
```

```
    /* 返回数据并释放结点 */
```

```
    *e = p->data;
```

```
    free(p);
```

```
    return OK;
```

```
}
```

`/* linkqueue.c 的组成 */`

`/* 遍历队列 */`

`Status QueueTraverse(LinkQueue Q, Status (*visit)(QElemType e))`

`{`

`extern int line_count; //main中定义的打印换行计数器(与算法无关)`

`QueuePtr p = Q.front->next; //指向首元`

`line_count = 0; //计数器恢复初始值(与算法无关)`

`while(p && (*visit)(p->data)==TRUE)`

`p=p->next;`

`if (p) //p!=NULL说明visit出错`

`return ERROR;`

`printf("\n"); //最后打印一个换行，只是为了好看，与算法无关`

`return OK;`

`}`

§ 3. 栈和队列

3. 4. 队列

3. 4. 2. 链队列-队列的链式表示与实现

含义：用单链表(带头结点)表示队列，分设两个指针

头指针：指向队头结点，用于删除

尾指针：指向队尾结点，用于插入

链队列的定义与实现

★ C语言的定义与实现

★ C++语言的定义与实现

```

/* linkqueue.h 的组成 */
/* P.10 的预定义常量和类型 */
#define TRUE          1
#define FALSE        0
#define OK           1
#define ERROR        0
#define INFEASIBLE   -1
#define LOVERFLOW    -2 //避免与<math.h>中的定义冲突

typedef int Status;
/* 由P.61 形式定义转换而来 */
typedef int QElemType;    //可根据需要修改元素的类型

class LinkQueue; //提前声明，因为定义友元要用到
class QNode {
    protected:
        QElemType  data;    //数据域
        QNode      *next;   //指针域
    public:
        friend class LinkQueue;
        //不定义任何函数，相当于struct QNode
};

```

/* linkqueue.h 的组成 */

```
class LinkQueue {  
    protected:  
        QNode *front;    //头指针  
        QNode *rear;    //尾指针  
    public:  
        /* P. 59-60的抽象数据类型定义转换为实际的C++语言 */  
        LinkQueue();      //构造函数，替代InitQueue  
        ~LinkQueue();     //析构函数，替代DestroyQueue  
        Status    ClearQueue();  
        Status    QueueEmpty();  
        int       QueueLength();  
        Status    GetHead(QElemType &e);  
        Status    EnQueue(QElemType e);  
        Status    DeQueue(QElemType &e);  
        Status    QueueTraverse(Status (*visit)(QElemType e));  
};
```

```
/* linkqueue.cpp 的组成 */
#include <iostream>
#include <stdlib.h>           //malloc/realloc函数
#include <unistd.h>          //exit函数
#include "linkqueue.h"       //形式定义
using namespace std;

/* 初始化队列，替代InitQueue */
LinkQueue::LinkQueue()
{
    /* 申请头结点空间，赋值给头/尾指针 */
    rear = front = new QNode;
    if (front == NULL)
        exit(LOVERFLOW);

    front->next = NULL;    //头结点的next域
}
```



```
/* linkqueue.cpp 的组成 */
/* 销毁队列，替代DestroyQueue */
LinkQueue::~LinkQueue()
{
    /* 整个链表(含头结点)依次释放，没有像链表、栈等借助
       QNode *p, *q; 而直接借用了front和rear */
    while(front) {
        rear = front->next; //抓住链表的下一个结点
        delete front;
        front = rear;
    }

    rear = front = NULL;
}
```

```
/* linkqueue.cpp 的组成 */
/* 清空队列（保留头结点） */
Status LinkQueue::ClearQueue()
{
    QNode *q, *p = front->next; //指向首元结点

    /* 从首元结点开始依次释放 */
    while(p) {
        q = p->next; //抓住链表的下一个结点
        delete p;
        p = q;
    }

    front->next = NULL; //头结点的next域置NULL
    rear = front;      //必须要，否则尾指针指向不正确
    return OK;
}
```

/* linkqueue.cpp 的组成 */

/* 判断是否为空队列 */

Status LinkQueue::QueueEmpty()

{

/* 判断front和rear指针是否相等 */

if (front == rear)

return TRUE;

else

return FALSE;

}

//也可以判断头结点的next域

if (front->next==NULL)

return TRUE;

else

return FALSE;

`/* linkqueue.cpp 的组成 */`

`/* 求队列的长度 */`

```
int LinkQueue::QueueLength()
{
    int    len = 0;
    QNode *p = front->next;    //指向首元结点

    while(p) {
        len++;
        p = p->next;
    }

    return len;
}
```

```
/* linkqueue.cpp 的组成 */
```

```
/* 取队头元素 */
```

```
Status LinkQueue::GetHead(QElemType &e)
```

```
{
```

```
    /* 空队列则返回 */
```

```
    if (front == rear) //用front->next==NULL也可以
```

```
        return ERROR;
```

```
    /* 取首元结点的值 */
```

```
    e = front->next->data;
```

```
    return OK;
```

```
}
```

```
/* linkqueue.cpp 的组成 */
```

```
/* 元素入队列 */
```

```
Status LinkQueue::EnQueue(QElemType e)
```

```
{
```

```
    QNode *p;
```

```
    /* 申请新结点 */
```

```
    p = new QNode;
```

```
    if (p==NULL)
```

```
        return LOVERFLOW;
```

```
    /* 加入队尾 */
```

```
    p->data = e;
```

```
    p->next = NULL;           //新结点的next必为NULL
```

```
    rear->next = p;          //接在当前队尾的后面
```

```
    rear = p;                 //指向新的队尾
```

```
    return OK;
```

```
}
```

```
/* linkqueue.cpp 的组成 */
/* 元素出队列 */
Status LinkQueue::DeQueue(QElemType &e)
{
    QNode *p;
    /* 空队列则返回 */
    if (front == rear) //用front->next==NULL也可以
        return ERROR;

    /* 处理首元结点 */
    p = front->next;           //指向首元
    front->next = p->next; //front指向新首元(可能为NULL)

    /* 如果只有一个结点，则必须修改尾指针 */
    if (rear == p)
        rear = front;

    /* 返回数据并释放结点 */
    e = p->data;
    delete p;
    return OK;
}
```

```
/* linkqueue.cpp 的组成 */
```

```
/* 遍历队列 */
```

```
Status LinkQueue::QueueTraverse(Status (*visit)(QElemType e))
```

```
{
```

```
    extern int line_count; //main中定义的打印换行计数器(与算法无关)
```

```
    QNode *p = front->next; //指向首元结点
```

```
    line_count = 0; //计数器恢复初始值(与算法无关)
```

```
    while(p && (*visit)(p->data)==TRUE)
```

```
        p=p->next;
```

```
    if (p) //p!=NULL表示visit出错
```

```
        return ERROR;
```

```
    cout << endl; //最后打印一个换行，只是为了好看，与算法无关
```

```
    return OK;
```

```
}
```


§ 3. 栈和队列

3.4. 队列

3.4.3. 循环队列-队列的顺序表示与实现

顺序队列：基于顺序存储结构，采用一维数组来依次存放队列中的数据元素

类型定义：

```
#define MAXQSIZE 8 //仅为了后面图示方便
typedef struct {
    ElemType base[MAXQSIZE];
    int front; //指向队头元素(出队列方向)
    int rear;  //指向队尾元素(入队列方向)
} sequeue;
```

§ 3. 栈和队列

3. 4. 队列

3. 4. 3. 循环队列-队列的顺序表示与实现

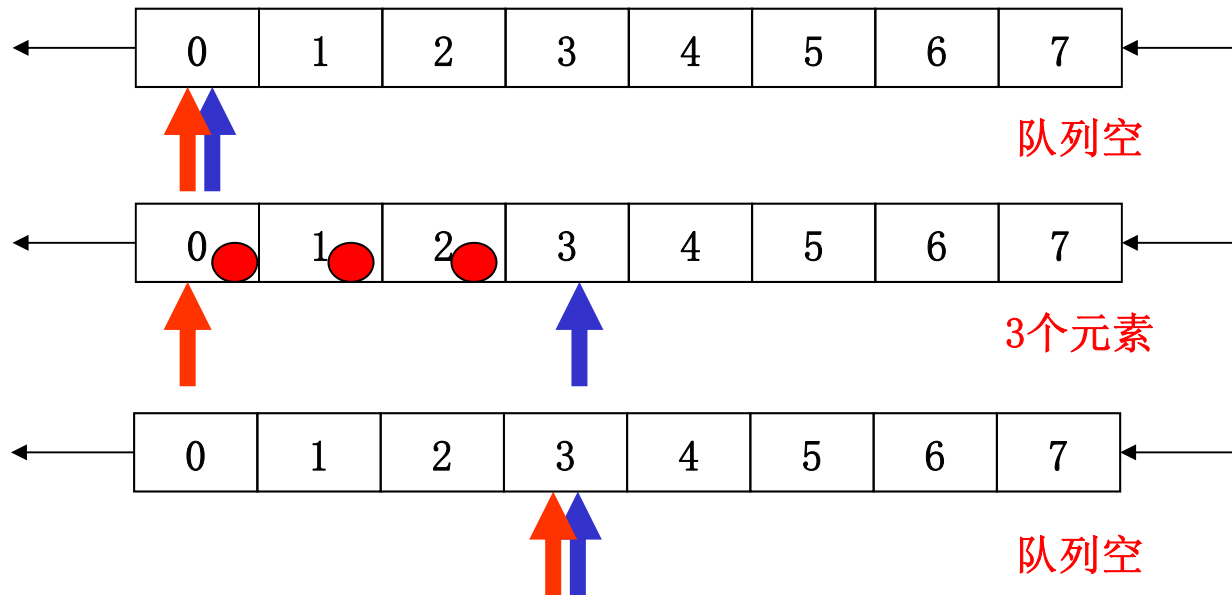
问题1: 如何设置头尾指针?

答: 和栈类似, front指针指向队头

rear 指针指向队尾+1 (即插入位置)

front==rear表示队列为空

rear - front = 队列长度



§ 3. 栈和队列

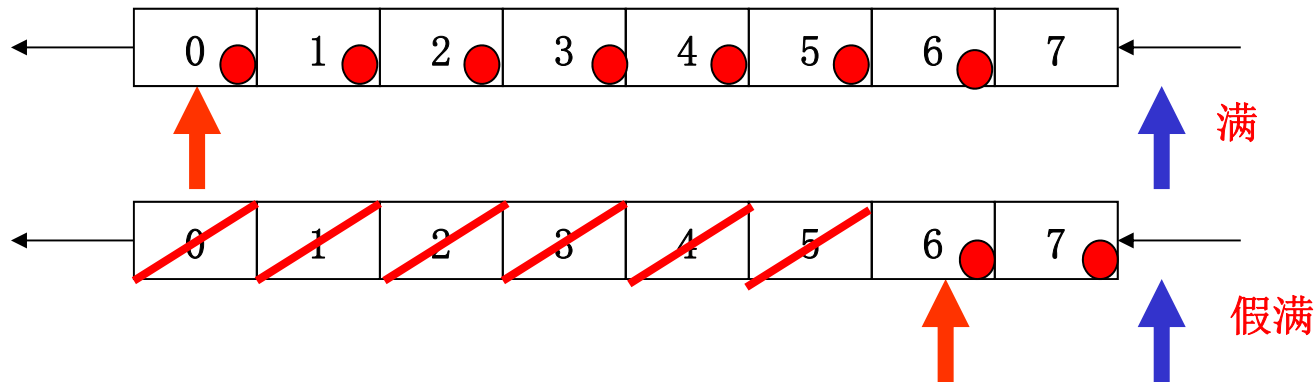
3. 4. 队列

3. 4. 3. 循环队列-队列的顺序表示与实现

问题1：如何设置头尾指针？

问题2：如何处理出现的假满？

假满：队头元素因为删除而顺序向队尾移动，因前面空间无法利用而导致虽有空间但无法入队列的情况



§ 3. 栈和队列

3. 4. 队列

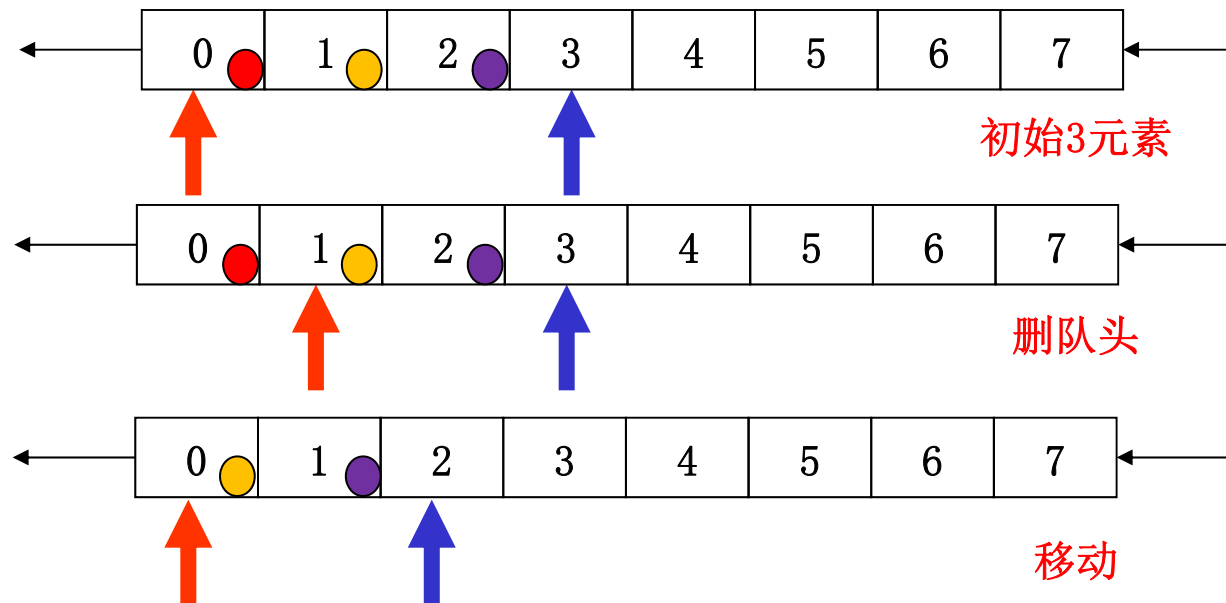
3. 4. 3. 循环队列-队列的顺序表示与实现

问题1：如何设置头尾指针？

问题2：如何处理出现的假满？

假满：队头元素因为删除而顺序向队尾移动，因前面空间无法利用而导致虽有空间但无法入队列的情况

解决方法1：每次删除后所有元素向队头移动，即保持队头始终是下标0
(造成大量数据的移动)



§ 3. 栈和队列

3. 4. 队列

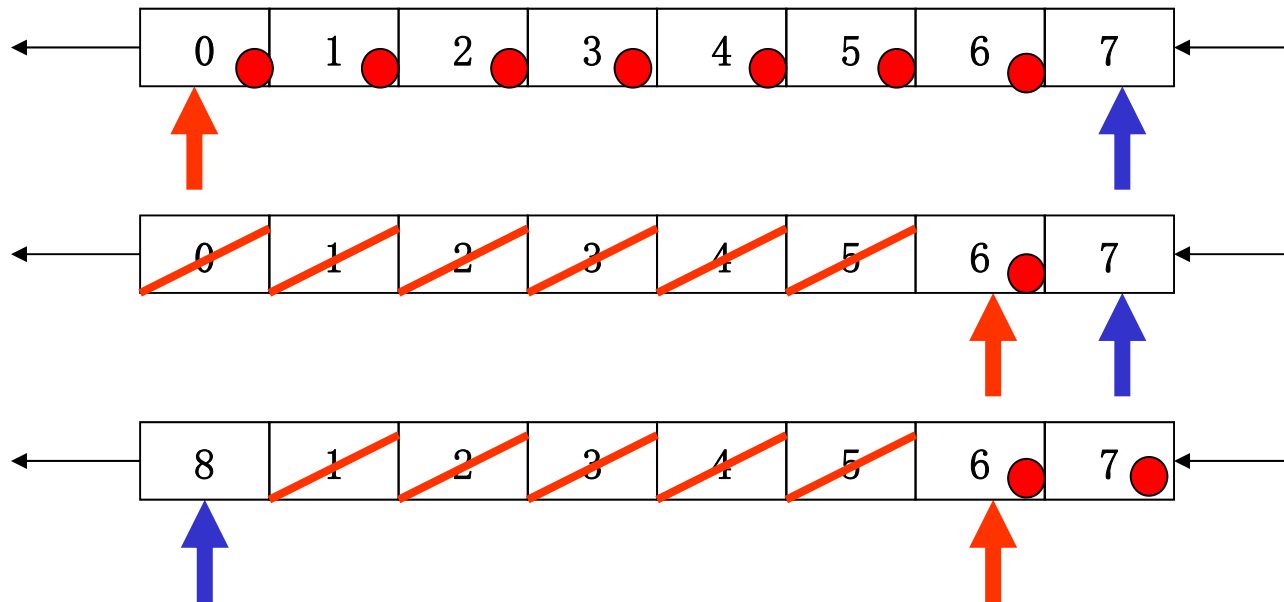
3. 4. 3. 循环队列-队列的顺序表示与实现

问题1：如何设置头尾指针？

问题2：如何处理出现的假满？

假满：队头元素因为删除而顺序向队尾移动，因前面空间无法利用而导致虽有空间但无法入队列的情况

解决方法2：循环队列



§ 3. 栈和队列

3. 4. 队列

3. 4. 3. 循环队列-队列的顺序表示与实现

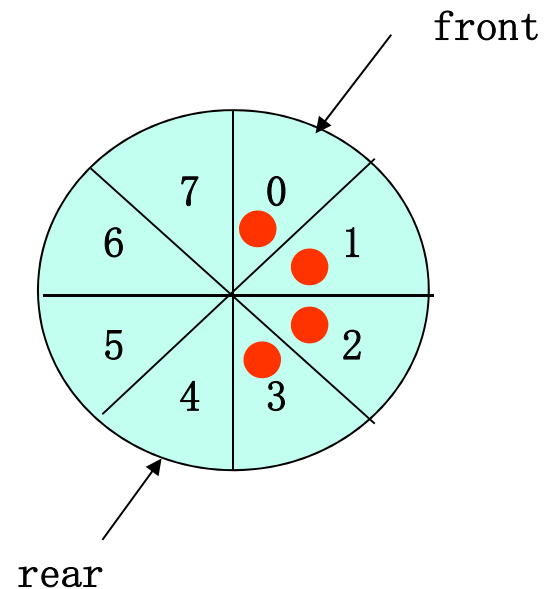
问题1：如何设置头尾指针？

问题2：如何处理出现的假满？

假满：队头元素因为删除而顺序向队尾移动，因前面空间无法利用而导致虽有空间但无法入队列的情况

解决方法2：循环队列

[0..MAXQSIZE-1] 当 $\text{front}/\text{rear} == \text{MAXQSIZE}$ 时
令 $\text{front}/\text{rear} = 0$



§ 3. 栈和队列

3. 4. 队列

3. 4. 3. 循环队列-队列的顺序表示与实现

问题1：如何设置头尾指针？

问题2：如何处理出现的假满？

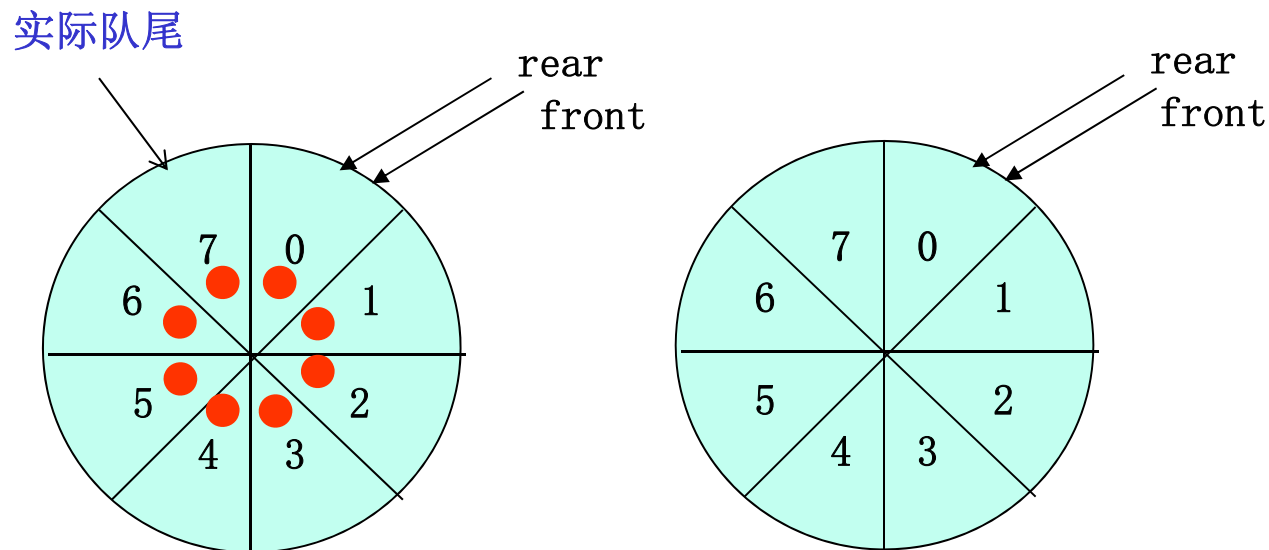
假满：队头元素因为删除而顺序向队尾移动，因前面空间无法利用而导致虽有空间但无法入队列的情况

解决方法2：循环队列

[0..MAXQSIZE-1] 当 $\text{front}/\text{rear} == \text{MAXQSIZE}$ 时

令 $\text{front}/\text{rear} = 0$

问题：无法区分满/空



§ 3. 栈和队列

3. 4. 队列

3. 4. 3. 循环队列-队列的顺序表示与实现

问题1: 如何设置头尾指针?

问题2: 如何处理出现的假满?

假满: 队头元素因为删除而顺序向队尾移动, 因前面空间无法利用而导致虽有空间但无法入队列的情况

解决方法2: 循环队列

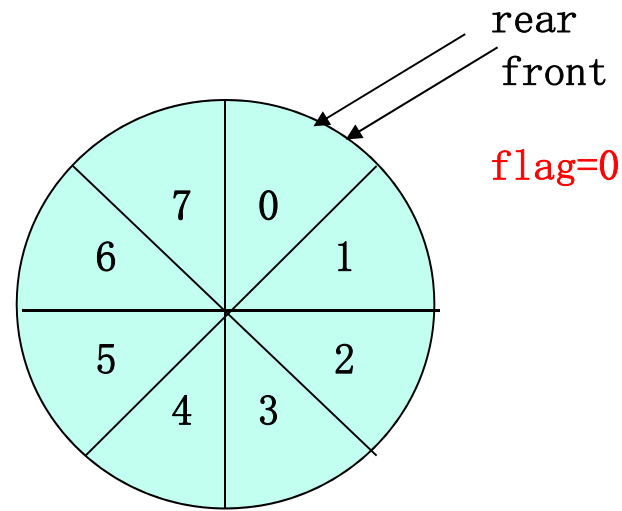
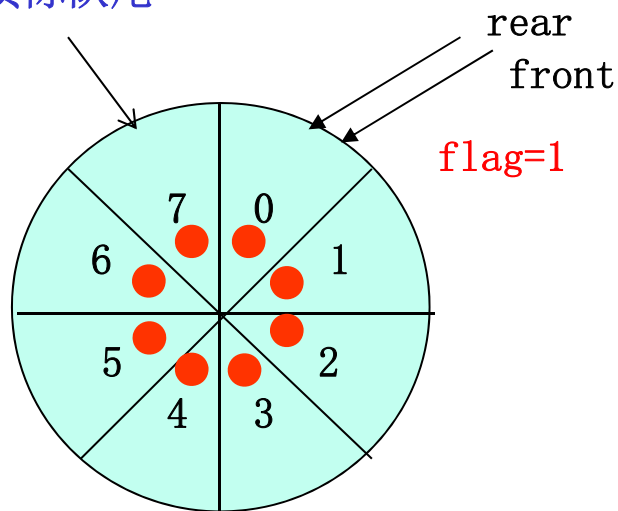
[0..MAXQSIZE-1] 当 front/rear==MAXQSIZE 时
令 front/rear=0

问题: 无法区分满/空

1、另设标记

```
if (front==rear) {  
    if (flag==1)  
        cout << "队列满";  
    else  
        cout << "队列空";  
}
```

实际队尾



§ 3. 栈和队列

3. 4. 队列

3. 4. 3. 循环队列-队列的顺序表示与实现

问题1：如何设置头尾指针？

问题2：如何处理出现的假满？

假满：队头元素因为删除而顺序向队尾移动，因前面空间无法利用而导致虽有空间但无法入队列的情况

解决方法2：循环队列

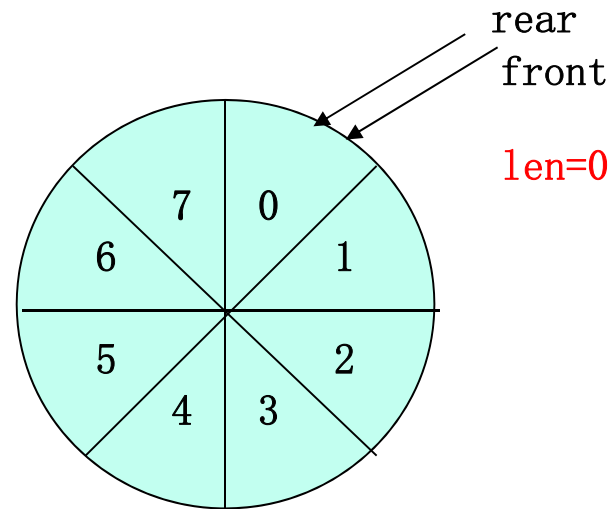
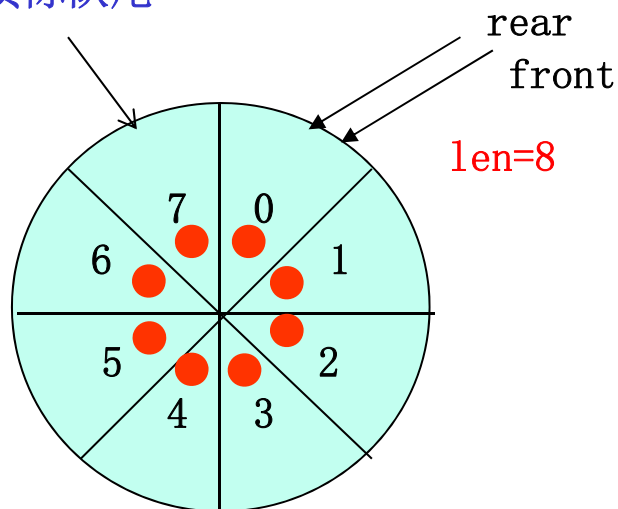
[0..MAXQSIZE-1] 当 front/rear==MAXQSIZE 时
令 front/rear=0

问题：无法区分满/空

- 1、另设标记
- 2、另设长度

```
if (front==rear) {  
    if (len!=0)  
        cout << "队列满";  
    else  
        cout << "队列空";  
}
```

实际队尾



§ 3. 栈和队列

3. 4. 队列

3. 4. 3. 循环队列-队列的顺序表示与实现

问题1：如何设置头尾指针？

问题2：如何处理出现的假满？

假满：队头元素因为删除而顺序向队尾移动，因前面空间无法利用而导致虽有空间但无法入队列的情况

解决方法2：循环队列

[0..MAXQSIZE-1] 当 $\text{front}/\text{rear} == \text{MAXQSIZE}$ 时

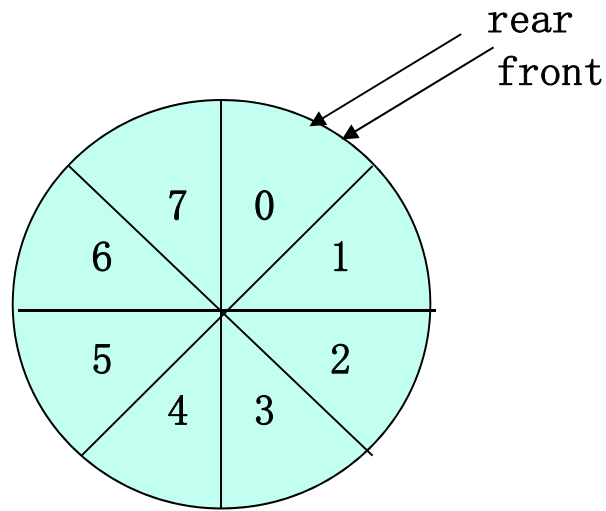
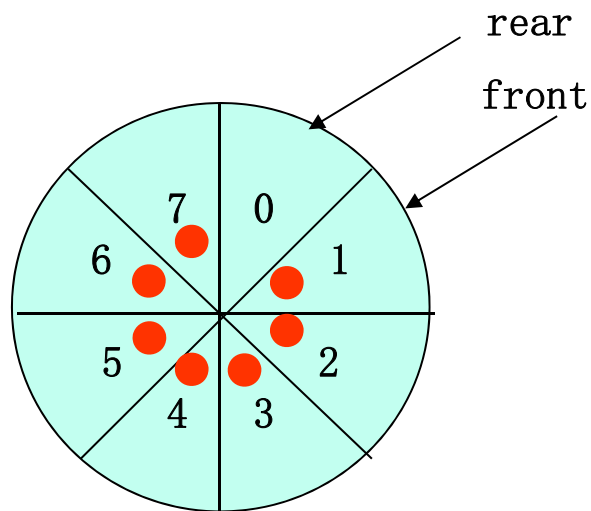
令 $\text{front}/\text{rear} = 0$

问题：无法区分满/空

- 1、另设标记
- 2、另设长度
- 3、少用一个元素

若 $\text{front} == \text{rear}$ ，表示空

若 $(\text{rear} + 1) \% \text{MAXQSIZE} == \text{front}$ ，表示满



§ 3. 栈和队列

3. 4. 队列

3. 4. 3. 循环队列-队列的顺序表示与实现

循环队列的定义与实现

★ C语言的定义与实现

`/* sqqueue.h 的组成 */`

`/* P.10 的预定义常量和类型 */`

```
#define TRUE          1
#define FALSE         0
#define OK            1
#define ERROR         0
#define INFEASIBLE    -1
#define LOVERFLOW     -2  //避免与<math.h>中的定义冲突
```

```
typedef int Status;
```

`/* P.64 结构体定义 */`

```
#define MAXQSIZE 100  //按需修改，但一旦确定则无法扩充
```

```
typedef int QElemType;  //可根据需要修改元素的类型
```

```
typedef struct {
    QElemType *base;  //存放动态申请空间的首地址
    int        front;  //队头指针（指向队头元素的下标）
    int        rear;  //队尾指针（指向队尾元素的下标）
} SqQueue;
```

/* sqqueue.h 的组成 */

/* P. 59-60的抽象数据类型定义转换为实际的C语言 */

```
Status InitQueue(SqQueue *Q);
Status DestroyQueue(SqQueue *Q);
Status ClearQueue(SqQueue *Q);
Status QueueEmpty(SqQueue Q);
int    QueueLength(SqQueue Q);
Status GetHead(SqQueue Q, QElemType *e);
Status EnQueue(SqQueue *Q, QElemType e);
Status DeQueue(SqQueue *Q, QElemType *e);
Status QueueTraverse(SqQueue Q, Status (*visit)(QElemType e));
```

/ sqqueue.c 的组成 */*

```
#include <stdio.h>
#include <stdlib.h>           //malloc/realloc函数
#include <unistd.h>           //exit函数
#include "sqqueue.h"          //形式定义
```

/ 初始化队列 */*

```
Status InitQueue(SqQueue *Q)
{
    Q->base = (QElemType *)malloc(MAXQSIZE * sizeof(QElemType));
    if (Q->base == NULL)
        exit(LOVERFLOW);

    Q->front = Q->rear = 0;  //只要front和rear相等
                           //指向0 ~ MAXQSIZE-1的任意值均可

    return OK;
}
```

`/* sqqueue.c 的组成 */`

`/* 销毁队列 */`

`Status DestroyQueue(SqQueue *Q)`

`{`

`if (Q->base)`

`free(Q->base);`

`Q->front = Q->rear = 0; //可写成 Q->front=Q->rear;`

`return OK;`

`}`

/ sqqueue.c 的组成 */*

/ 清空队列（已初始化，不释放空间，只清除内容） */*

Status ClearQueue(SqQueue *Q)

{

 Q->front = Q->rear = 0; *//只要front和rear相等即可*
 //即 Q->front = Q->rear;
 //或 Q->rear = Q->front;

 return OK;

}

/ sqqueue.c 的组成 */*

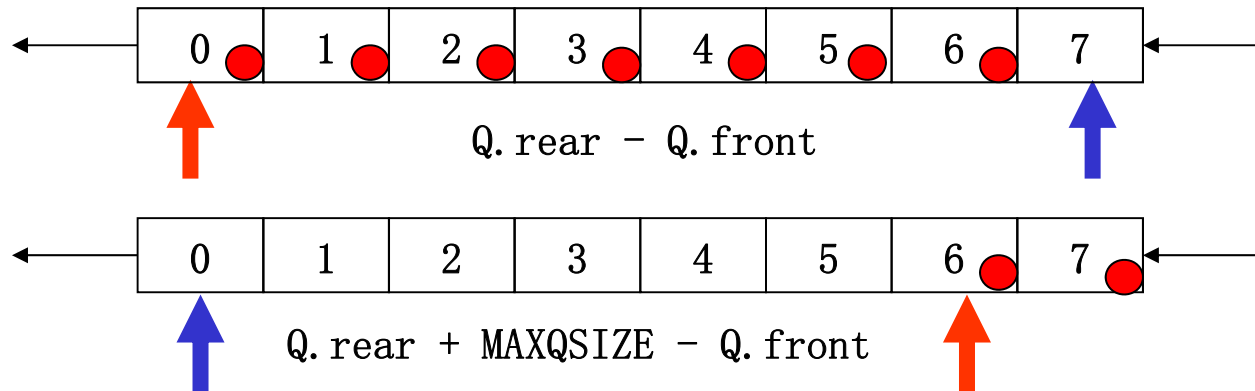
/ 判断是否为空队列 */*

```
Status QueueEmpty(SqQueue Q)
{
    if (Q.front == Q.rear)
        return TRUE;
    else
        return FALSE;
}
```

/* sqqueue.c 的组成 */

/* 求队列的长度 */

```
int QueueLength(SqQueue Q)
{
    return (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE;
}
```



/ sqqueue.c 的组成 */*

/ 取队头元素 */*

```
Status GetHead(SqQueue Q, QElemType *e)
{
    /* 队列为空则直接返回 */
    if (Q.front == Q.rear)
        return ERROR;

    *e = Q.base[Q.front]; //取队头
    return OK;
}
```

/* sqqueue.c 的组成 */

/* 元素入队列 */

```
Status EnQueue(SqQueue *Q, QElemType e)
{
```

/* 队列满则直接返回 */

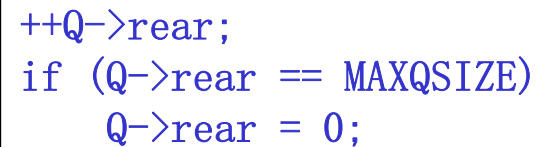
```
    if ( (Q->rear+1) % MAXQSIZE == Q->front)
        return ERROR;
```

```
    Q->base[Q->rear] = e; //rear指向队尾+1，就是插入位置
```

```
    Q->rear = (++Q->rear) % MAXQSIZE; //队尾指针向后移动
```

```
    return OK;
```

```
}
```



```
++Q->rear;
if (Q->rear == MAXQSIZE)
    Q->rear = 0;
```

/ sqqueue.c 的组成 */*

/ 元素出队列 */*

Status DeQueue(SqQueue *Q, QElemType *e)

{

/ 队列空则直接返回 */*

if (Q->front == Q->rear)

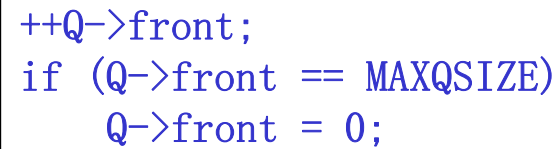
return ERROR;

*e = Q->base[Q->front]; *//取队头元素*

Q->front = (++Q->front)%MAXQSIZE; *//队头指针向后移动*

return OK;

}



```
++Q->front;  
if (Q->front == MAXQSIZE)  
    Q->front = 0;
```

/* sqqueue.c 的组成 */

/* 遍历队列 */

Status QueueTraverse(SqQueue Q, Status (*visit)(QElemType e))

{

extern int line_count; //main中定义的打印换行计数器(与算法无关)

int pos = Q.front;

line_count = 0; //计数器恢复初始值(与算法无关)

/* pos从队头开始, 循环到队尾

(pos==Q.rear时不做visit, 恰好Q.rear指向队尾+1) */

while(pos!=Q.rear && (*visit)(Q.base[pos])==TRUE)

pos = (++pos)%MAXQSIZE;

if (pos!=Q.rear) //表示visit出现了错误

return ERROR;

printf("\n"); //最后打印一个换行, 只是为了好看, 与算法无关

return OK;

}

§ 3. 栈和队列

3. 4. 队列

3. 4. 3. 循环队列-队列的顺序表示与实现

循环队列的定义与实现

★ C语言的定义与实现

★ C++语言的定义与实现

`/* sqqueue.h 的组成 */`

`/* P.10 的预定义常量和类型 */`

```
#define TRUE          1
#define FALSE        0
#define OK           1
#define ERROR        0
#define INFEASIBLE   -1
#define LOVERFLOW    -2 //避免与<math.h>中的定义冲突
```

```
typedef int Status;
```

`/* P.64 结构体定义 */`

```
#define MAXQSIZE 100 //按需修改，但一旦确定则无法扩充
```

```
typedef int QElemType; //可根据需要修改元素的类型
```


/* sqqueue.h 的组成 */

```
class SqQueue {  
    protected:  
        QElemType *base;    //存放动态申请空间的首地址  
        int front;    //队头指针（指向队头元素的下标）  
        int rear;    //队尾指针（指向队尾元素的下标）  
    public:  
        /* P. 59-60的抽象数据类型定义转换为实际的C++语言 */  
        SqQueue();    //构造函数，替代InitQueue  
        ~SqQueue();    //析构函数，替代DestroyQueue  
        Status ClearQueue();  
        Status QueueEmpty();  
        int QueueLength();  
        Status GetHead(QElemType &e);  
        Status EnQueue(QElemType e);  
        Status DeQueue(QElemType &e);  
        Status QueueTraverse(Status (*visit)(QElemType e));  
};
```

/ sqqueue.cpp 的组成 */*

```
#include <iostream>
#include <stdlib.h>           //malloc/realloc函数
#include <unistd.h>           //exit函数
#include "sqqueue.h"          //形式定义
using namespace std;
```

/ 构造函数（初始化队列） */*

```
SqQueue::SqQueue()
{
    /* 申请头结点 */
    base = new QElemType[MAXQSIZE];
    if (base == NULL)
        exit(LOVERFLOW);
    front = rear = 0; //只要front和rear相等
                     //指向0 ~ MAXQSIZE-1的任意值均可
                     //但此处不能直接 front = rear 或 rear = front;
                     //因为此时front/rear的值尚不确定
}
```

```
/* sqqueue.cpp 的组成 */
```

```
/* 析构函数（销毁队列） */
```

```
SqQueue::~SqQueue()
```

```
{
```

```
    if (base)
```

```
        delete base;
```

```
    front = rear = 0;  //用 front = rear; 即可
```

```
                        //或 rear = front; 也可
```

```
                        //或者干脆不要
```

```
}
```

`/* sqqueue.cpp 的组成 */`

`/* 清空队列（已初始化，不释放空间，只清除内容） */`

`Status SqQueue::ClearQueue()`

`{`

`front = rear = 0; //用 front = rear; 即可`

`//或 rear = front; 也可`

`//因为此时front/rear保证`

`//位于 [0 .. MAXQSIZE-1] 之间`

`return OK;`

`}`

`/* sqqueue.cpp 的组成 */`

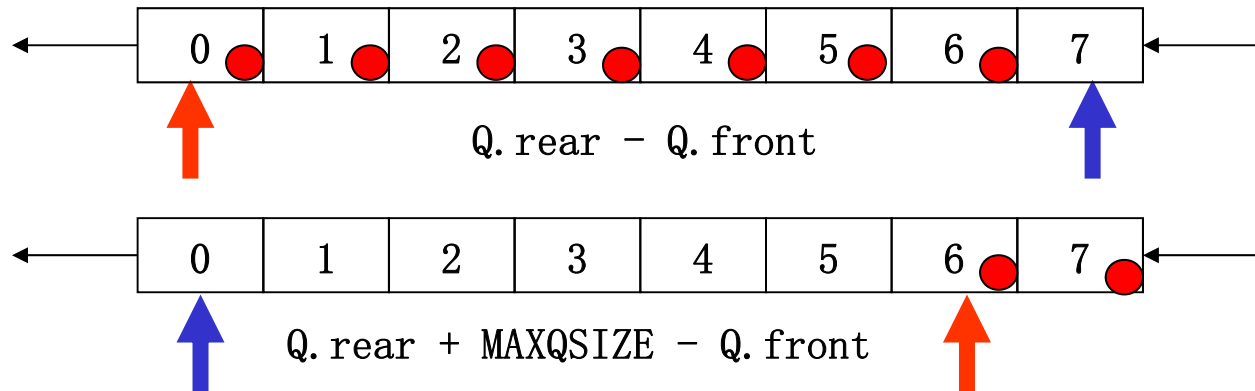
`/* 判断是否为空队列 */`

```
Status SqQueue::QueueEmpty()
{
    if (front == rear)
        return TRUE;
    else
        return FALSE;
}
```

/* sqqueue.cpp 的组成 */

/* 求队列的长度 */

```
int SqQueue::QueueLength()
{
    return (rear - front + MAXQSIZE) % MAXQSIZE;
}
```



```
/* sqqueue.cpp 的组成 */
```

```
/* 取队头元素 */
```

```
Status SqQueue::GetHead(QElemType &e)
```

```
{
```

```
    /* 空队列则直接返回 */
```

```
    if (front == rear)
```

```
        return ERROR;
```

```
    e = base[front];    //取队头
```

```
    return OK;
```

```
}
```

/* sqqueue.cpp 的组成 */

/* 元素入队列 */

Status SqQueue::EnQueue(QElemType e)

{

/* 队列满则直接返回 */

if ((rear+1) % MAXQSIZE == front)

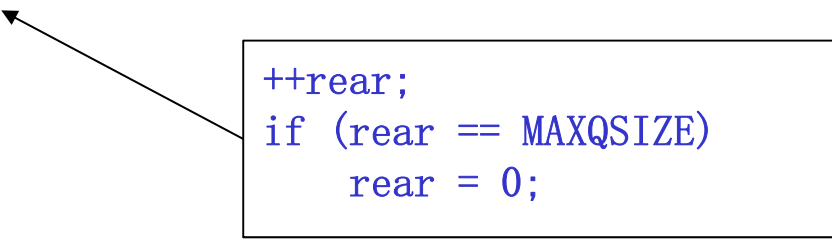
return ERROR;

base[rear] = e; //rear指向队尾+1, 正好是插入位置

rear = (++rear) % MAXQSIZE; //队尾指针向后移动

return OK;

}



```
++rear;  
if (rear == MAXQSIZE)  
    rear = 0;
```


/* sqqueue.cpp 的组成 */

/* 元素出队列 */

Status SqQueue::DeQueue(QElemType &e)

{

/* 队列空则直接返回 */

if (front == rear)

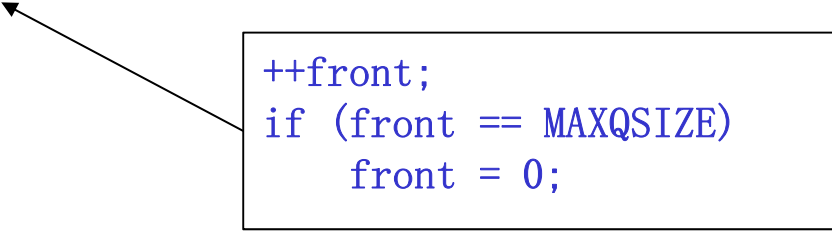
return ERROR;

e = base[front];

front = (++front) % MAXQSIZE; //队头指针向后移动

return OK;

}



```
++front;  
if (front == MAXQSIZE)  
    front = 0;
```

```
/* sqqueue.cpp 的组成 */
```

```
/* 遍历队列 */
```

```
Status SqQueue::QueueTraverse(Status (*visit)(QElemType e))
```

```
{
```

```
    extern int line_count; //main中定义的打印换行计数器(与算法无关)
```

```
    int pos = front;
```

```
    line_count = 0;                //计数器恢复初始值（与算法无关）
```

```
    /* pos从队头开始，循环到队尾
```

```
       (pos==Q.rear时不做visit，恰好Q.rear指向队尾+1) */
```

```
    while(pos!=rear && (*visit)(base[pos])==TRUE)
```

```
        pos = (++pos)%MAXQSIZE;
```

```
    if (pos!=rear)    //表示visit出现了错误
```

```
        return ERROR;
```

```
    cout << endl; //最后打印一个换行，只是为了好看，与算法无关
```

```
    return OK;
```

```
}
```

§ 3. 栈和队列

3. 4. 队列

3. 4. 4. 双端队列 (P. 60 略)

含义：在两端均可以插入/删除的队列，有多种形式

§ 3. 栈和队列

3. 5. 离散事件模拟 (略)