

第八章:引用

8.2 引用变量

8.2.1 创建引用变量

8.2.1.1 创建引用变量

前面讲过，C 和 C++ 使用`&`符号来指示变量的地址。C++ 给`&`符号赋予了另一个含义，将其用来声明引用。例如，要将 `rodents` 作为 `rats` 变量的别名，可以这样做：

```
int rats;
int & rodents = rats; // makes rodents an alias for rats
```

其中，`&`不是地址运算符，而是类型标识符的一部分。就像声明中的 `char*` 指的是指向 `char` 的指针一样，`int &` 指的是指向 `int` 的引用。上述引用声明允许将 `rats` 和 `rodents` 互换——它们指向相同的值和内存单元，程序清单 8.2 表明了这一点。

请注意，下述语句中的`&`运算符不是地址运算符，而是将 `rodents` 的类型声明为 `int &`，即指向 `int` 变量的引用：

```
int & rodents = rats;
```

但下述语句中的`&`运算符是地址运算符，其中`&rodents` 表示 `rodents` 引用的变量的地址：

```
cout << ", rodents address = " << &rodents << endl;
```

指针和引用的区别

对于 C 语言用户而言，首次接触到引用时可能也会有些困惑，因为这些用户很自然地会想到指针，但它们之间还是有区别的。例如，可以创建指向 `rats` 的引用和指针：

```
int rats = 101;
int & rodents = rats; // rodents a reference
int * prats = &rats; // prats a pointer
```

这样，表达式 `rodents` 和 `*prats` 都可以同 `rats` 互换，而表达式 `&rodents` 和 `prats` 都可以同 `&rats` 互换，从

不同于指针的。除了表示法不同外，还有其他的差别。例如，差别之一是，必须在声明引用时将其初始化，而不能像指针那样，先声明，再赋值：

```
int rat;
int & rodent;
rodent = rat; // No, you can't do this.
```

注意：必须在声明引用变量时进行初始化。

8.2.2 作为函数参数

引用经常被用作函数参数，使得函数中的变量名成为调用程序中的变量的别名。这种传递参数的方法称为按引用传递。按引用传递允许被调用的函数能够访问调用函数中的变量。C++ 新增的这项特性是对 C 语言的超越，C 语言只能按值传递。按值传递导致被调用函数使用调用程序的值的拷贝（参见图 8.2）。当然，C 语言也允许避开按值传递的限制，采用按指针传递的方式。

按值传递

```
void sneezy(int x);
int main()
{
    int times = 20;
    sneezy(times);
    ...
}

void sneezy(int x)
{
    ...
}
```

→ 创建times变量，将值20赋给它

→ 创建x变量，将传递来的值20赋给它

20
times

20
x

2个变量
2个名称

按引用传递

```
void grumpy(int &x);
int main()
{
    int times = 20;
    grumpy(times);
    ...
}

void grumpy(int &x)
{
    ...
}
```

→ 创建times变量，将值20赋给它

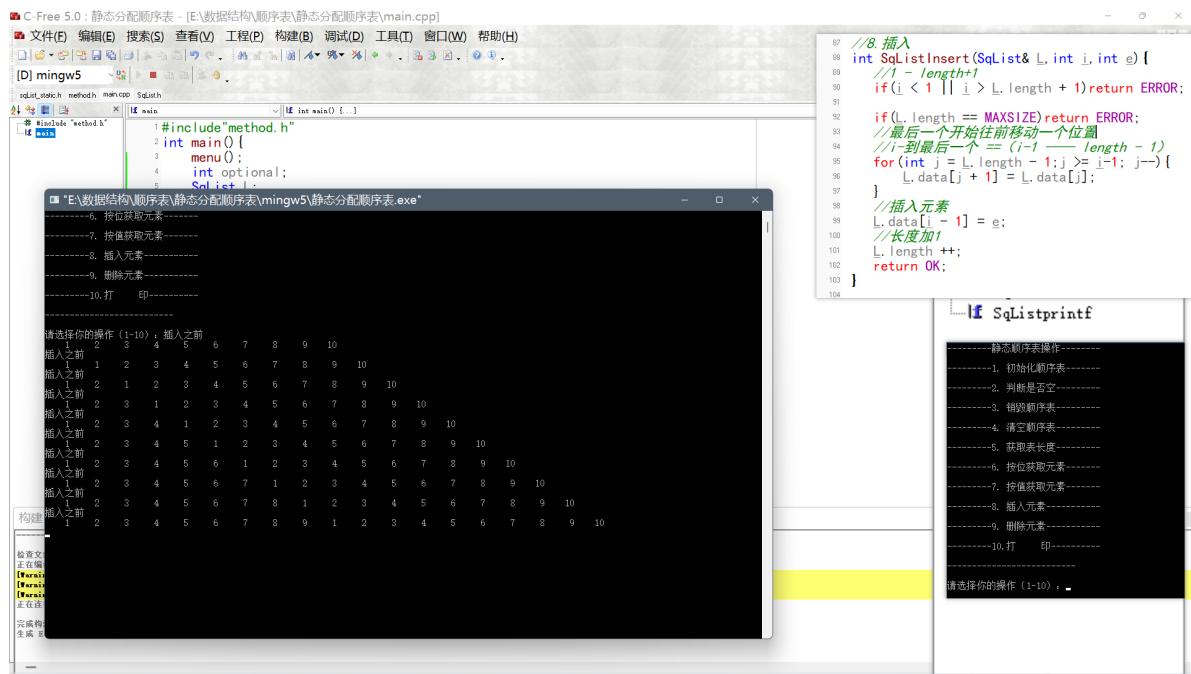
→ 使x成为times的别名

20
times, x

1个变量
2个名称

1.顺序表

1.静态分配



1.SqList.h

```
//SqList.h
//#####
//为什么直接传顺序表不可以，网上传的指针；
//直接传L过去，那它就是直接拷贝，不会影响实参

//Status InitList_Sq(SqList L)

#include<cstdio>
#include<stdlib.h>
//#####
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2

//#####
typedef int ElemType;

//顺序表最大值
#define MAXSIZE 100
//增量+最大值的

typedef struct{
    //静态分配，数组大小和空间固定，空间占满会溢出，导致崩溃
    ElemType data[MAXSIZE];
    ElemType length; //当前长度
}SqList;
```

2. 初始化InitSqList

```
//1.顺序表初始化
int InitSqList(SqList& L){

    for(int i = 0; i < MAXSIZE; i++){
        L.data[i] = 0;
        //printf("%d ",L.data[i]);
    }

    L.length= 0;
    return OK;
}
```

3. DestroySqList

```
//2.销毁顺序表
void DestroySqList(SqList& L){

    //静态顺序表中，只要将长度设置为0即可，操作系统会回收空间
    L.length = 0;

}
```

4. ClearSqList

```
//3.清空顺序表
void ClearSqList(SqList& L){

    L.length = 0;
    printf("Clear顺序表已清空现在长度为: %d\n",L.length);

}
```

5. GetSqListLength

```
//4.获取顺序表长度
int GetSqListLength(SqList& L){
    printf("顺序表的长度为: %d\n",L.length);
    return L.length;
}
```

6. SqListIsEmpty

```
//5.判断顺序表是否空
void SqListIsEmpty(SqList& L){
    if(L.length == 0)printf("空\n");
    else{
        printf("非空\n");
    }
}
```

7.SqListGetElem按位

```
//6.按位查找（时间复杂度O(1) 随机存取，每一条只运行一次）
int SqListGetElem(SqList& L,int i,int& e){
    //1-length范围
    if(i < 1 || i > L.length) return ERROR;
    e = L.data[i - 1];
    printf("第%d位置的元素是: %d\n",i,e);
    return OK;
}
```

8.SqListLocateElem按值

```
//7.按值查找，顺序查找
int SqListLocateElem(SqList& L,int& e){
    for(int i = 0; i < L.length; i++){
        if(L.data[i] == e) return i + 1;
    }
    return 0;
}
```

9.SqListInsert

```
//8.插入
int SqListInsert(SqList& L,int i,int& e){
    //1 - length+1
    if(i < 1 || i > L.length + 1) return ERROR;

    if(L.length == MAXSIZE) return ERROR;
    //最后一个开始往前移动一个位置
    //i-到最后一个 == (i-1 -- length - 1)
    for(int j = L.length - 1;j >= i-1; j--){
        L.data[j + 1] = L.data[j];
    }
    //插入元素
    L.data[i - 1] = e;
    //长度加1
    L.length++;
    return OK;
}
```

10.SqListDelete

```

//9.删除
int SqListDelete(SqList &L, int &i){
    if((i < 1) || (i > L.length)) return ERROR;
    //从i-最后一个均往前移动
    for(int j = i; j <= L.length - 1; j++){
        L.data[j - 1] = L.data[j];
    }
    //长度减1
    L->length--;
    return OK;
}

```

11.SqListprintf

```

//10.打印
void SqListprintf(SqList &L){
    if(L.length == 0){
        printf("顺序表长度为0\n");
        return;
    }
    for(int i = 0; i < L.length; i++){
        printf("%5d ", L.data[i]);
    }
    printf("\n");
}

```

2.动态分配

1.SqList.h

```

//#####头文件#####
#include<cstdio>
#include<cstdlib>

//#####函数返回状态#####
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#define InitSize 10
//#####Status函数类型#####

typedef int ElemType;
//顺序表最大值

//增量+最大值的

typedef struct{
    //顺序表元素类型 float.char
    int *elem;
    int length; //当前长度
}

```

```
    int MaxSize;//当前最大值  
}SqList;
```

2. 初始化InitSqList

```
//1.顺序表初始化  
int InitSqList(SqList& L){  
    //申请初始空间  
    L.elem = (int*)malloc(sizeof(int)*InitSize);  
    if(!L.elem)exit(OVERFLOW);  
    //清除脏数据  
    for(int i = 0; i < 10; i++){  
        L.elem[i] = 0;  
    }  
    //初始化成员  
    L.length = 0;  
    L.MaxValue = InitSize;  
    return OK;  
}
```

3. DestroySqList

```
//2.销毁顺序表  
void DestroySqList(SqList& L){  
    printf("顺序表已经销毁\n");  
    L.length = 0;  
    L.MaxValue = 0;  
    //释放指针  
    if(L.elem != NULL){  
        free(L.elem);  
        L.elem = NULL;  
    }  
}
```

4. ClearSqList

```
//3.清空顺序表  
void ClearSqList(SqList& L){  
    L.length = 0;  
    L.MaxValue = InitSize;  
    printf("Clear顺序表已清空现在长度为: %d\n", L.length);  
}
```

5. GetSqListLength

```
//4.获取顺序表长度  
int GetSqListLength(SqList& L){  
    printf("顺序表的长度为: %d\n", L.length);  
    return OK;  
}
```

6.SqListIsEmpty

```
//5.判断顺序表是否空
void SqListIsEmpty(SqList& L){
    if(L.length == 0)printf("空\n");
    else{
        printf("非空\n");
    }
}
```

7.SqListGetElem按位

```
//6.获取顺序表中位置i的元素的内容（时间复杂度O(1) 随机存取，每一条只运行一次）
int SqListGetElem(SqList& L,int i,int &e){
    //1-length范围
    if(i < 1 || i > L.length) return ERROR;
    e = L.elem[i - 1];
    printf("第%d位置的元素是: %d\n",i,e);
    return OK;
}
```

8.SqListLocateElem按值

```
//7.按值查找，顺序查找
int SqListLocateElem(SqList& L,int e){
    int flag = -1;
    for(int i = 0; i < L.length; i++){
        if(L.elem[i] == e){
            flag = i + 1;
            printf("元素%d在顺序表中的位置是%d\n",e,flag);
            return OK;
        }
    }
    if(flag == -1){
        printf("顺序表中没有元素%d\n",e);
    }
    return 0;
}
```

9.SqListInsert

```
//8.插入
int SqListInsert(SqList& L,int i,int e){
    if(i < 1 || i > L.length + 1) return ERROR;
    //空间满，增加空间
    if(L.length >= L.MaxValue){
        L.MaxValue = L.MaxValue + InitSize;
    }
    for(int j = L.length - 1;j >= i-1; j--){
        L.elem[j + 1] = L.elem[j];
    }
    L.elem[i - 1] = e;
    L.length++;
}
```

```
    return OK;  
}
```

10.SqListDelete

```
//9.删除  
int SqListDelete(SqList& L,int i){  
    if((i < 1)|| (i > L.length))return ERROR;  
    for(int j = i; j <= L.length - 1; j++){  
        L.elem[j - 1] = L.elem[j];  
  
    }  
    L.length --;  
    return OK;  
}
```

11.SqListprintf

```
//10.打印  
void SqListPrintf(SqList& L){  
    if(L.length == 0){  
        printf("顺序表长度为0\n");  
        return;  
    }  
    for(int i = 0; i < L.length; i++){  
        printf("%5d ",L.elem[i]);  
    }  
    printf("\n");  
}
```

2.单链表

1.带头节点

1.初始化：InitList

```
//1. 初始化  
bool initList(LinkList &L) {  
    L = (LinkList)malloc(sizeof(LNode));  
    L->next = NULL;  
    return true;  
}  
  
//2. 判断表是否为空  
bool listEmpty(LinkList L) {  
    if(L->next) {  
        return true;  
    }  
    else {  
        return false;  
    }  
}  
  
//3. 销毁  
bool destroyList(LinkList &L) {  
    LNode *p; //指向一个节点LinkList p也可以，不过为了便于理解用前一个  
    while(L)
```

1.申请头节点，L指向
2.将L->next设置为空

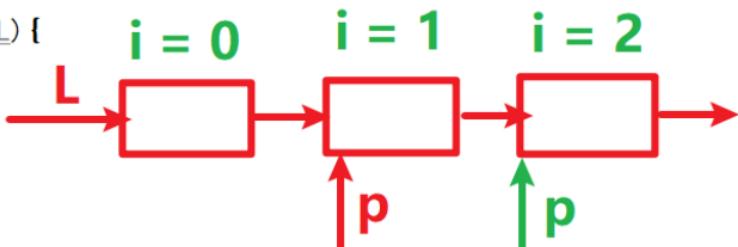


便于理解为一个链表：LNode *也可以

```
//1. 初始化：申请头节点，设置next为空
bool InitList(LinkList &L){
    L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;
    return true;
}
```

2.求表长：Length

```
50 //5. 求表长
51 int Length(LinkList L) {
52     LNode* p;
53     p = L->next;
54     int i = 0;
55     while(p) {
56         i++;
57         p = p->next;
58     }
59     return i;
60 }
```

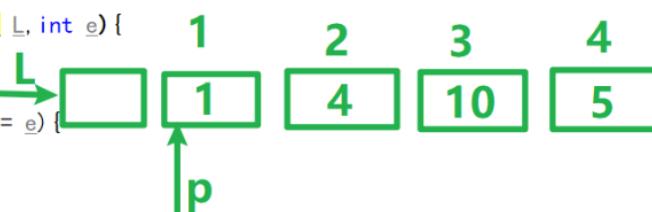


```
//5. 求表长：从L->next遍历累加
int Length(LinkList L){
    LNode* p;
    p = L->next;
    int i = 0;
    while(p){
        i++;
        p = p->next;
    }
    return i;
}
```

3.按值查找：LocateElem

从L->next遍历，计数从1开始记录当前节点，并查找

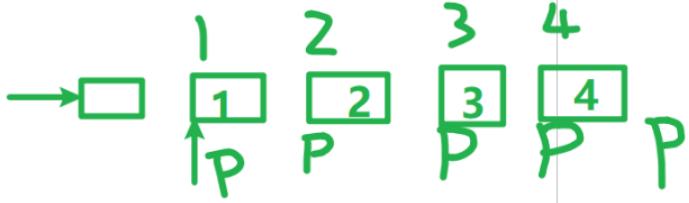
```
7 //7. 按值查找
8 int locateElem(LinkList L, int e) {
9     LNode *p;
10    p = L->next;
11    int j = 1;
12    while(p && p->data != e) {
13        p = p->next;
14        j++;
15    }
16    if(p) {
17        return j;
18    }
19    else {
20        return 0;
21    }
22 }
```



查找5

1.有这个值

```
//7.按值查找
int locateElem(LinkList L, int e) {
    LNode *p;
    p = L->next;
    int j = 1;
    while(p && p->data != e) {
        p = p->next;
        j++;
    }
    if(p) {
        return j;
    } else {
        return 0;
    }
}
```



查找122, 不存在

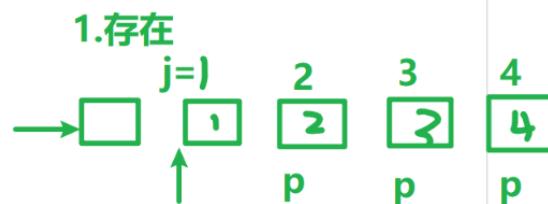
2.这个值不存在

```
//7.按值查找
int LocateElem(LinkList L,int e){
    LNode *p;
    p = L->next;
    int j = 1;
    while(p && p->data != e){
        p = p->next;
        j++;
    }
    if(p){
        return j;
    } else {
        return 0;
    }
}
```

4.按位查找：GetElem

遍历记位置： $j > i$,即*i*不合法

```
//6.按位取值
int GetElem(LinkList L, int i) {
    LNode *p;
    p = L->next;
    int j = 1;
    while(p && j < i) {
        p = p->next;
        j++;
    }
    if(!p || j > i) {
        return 0;
    }
    return p->data;
}
```

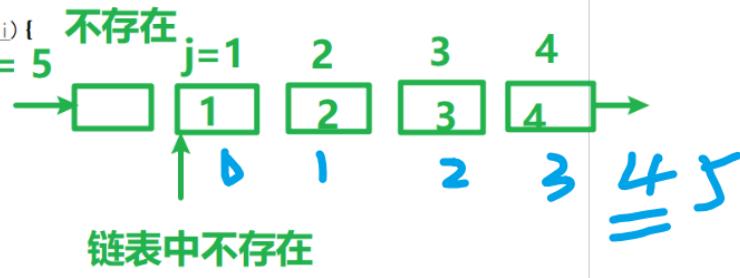


1.该位置链表中存在一定能找到

```

62 //6. 按位取值
63 int GetElem(LinkList L, int i) {
64     LNode *p;
65     p = L->next;
66     int j = 1;
67     while(p && j < i) {
68         p = p->next;
69         j++;
70     }
71     if(!p || j > i) {
72         return 0;
73     }
74     return p->data;
75 }

```



2. 链表中没有该位置

```

//6. 按位取值
int GetElem(LinkList L, int i){
    LNode *p;
    p = L->next;
    int j = 1;
    while(p && j < i){
        p = p->next;
        j++;
    }
    if(!p || j > i){
        return 0;
    }
    return p->data;
}

```

5. 插入 ListInsert

1. 判断合法: 1 —— length+1
2. 循环找到 $i - 1$
3. 判断是否找到
4. 申请空间插入节点

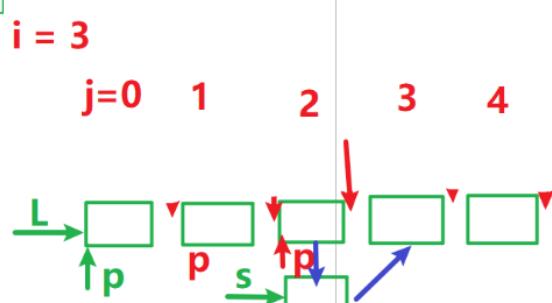
```

94 //8. 插入节点: 前插和后插, 前插可以转换为后插找到i-1个节点
95 bool ListInsert(LinkList &L, int i, int e) {
96     if(i < 1) {
97         return false;
98     }
99     LNode *p; //当前指针扫描到的节点
100    int j = 0; //当前p指向的是第几个节点
101    p = L; //L指向头节点, 头节点是第0个节点
102    while(p != NULL && j < i - 1) {
103        p = p->next;
104        j++;
105    }
106    if(p == NULL) { //i值不合法
107        return false;
108    }
109    LNode *s = (LNode *)malloc(sizeof(LNode));
110    s->data = e;
111    s->next = p->next;
112    p->next = s;
113    return true;
114 }
115 //删除节点

```

2. 后面插入, 3. 前面插入

1. 判断位置是否合法
2. 可能在第一个位置插入



//思考: 当 $i = 10$, 是, 仍然满足条件, 下一次, p变成了最后一个元素的后面为空;

```

//8.插入节点: 前插和后插, 前插可以转换为后插找到i-1个节点
bool ListInsert(LinkList &L,int i,int e){
    if(i < 1){
        return false;
    }
    LNode *p;//当前指针扫描到的节点
    int j = 0;//当前p指向的是第几个节点
    p = L;//L指向头节点, 头节点是第0个节点
    while(p != NULL && j < i - 1){
        p = p->next;
        j++;
    }
    if(p == NULL){//i值不合法
        return false;
    }
    LNode *s = (LNode *)malloc(sizeof(LNode));
    s->data = e;
    s->next = p->next;
    p->next = s;
    return true;
}

```

6.删除ListDelete

- 1.判断位置是否合法: 1-length
- 2.找到i-1个位置
- 3.判断链表中是否找到
- 4.判断是否为最后一个节点, 如果是那么, 他后面没有节点, 自然也没有next, 需特殊
- 5.p指向i-1, q指向i, 跳过q释放q即可

//9.删除节点

```

116 bool ListDelete(LinkList &L,int i,int &e) {
117     if(i < 1) {
118         return false;
119     }
120     LNode *p;
121     int j = 0;
122     p = L;
123     while(p != NULL && j < i - 1) {
124         p = p->next;
125         j++;
126     }
127     if(p == NULL) {
128         return false;
129     }
130     if(p->next == NULL) {
131         return false;
132     }
133     LNode *q = p->next;
134     e = q->data;
135     printf("被删除的节点是: %d\n", e);
136     p->next = q->next;
137     free(q);
138     return true;
139 }

```

```

//9.删除节点
bool ListDelete(LinkList &L,int i,int &e){
    if(i < 1){
        return false;
    }
    LNode *p;
    int j = 0;
    p = L;
    while(p != NULL && j < i - 1){
        p = p->next;
    }

```

```

        j++;
    }
    if(p == NULL){//超过了最后一个节点
        return false;
    }
    if(p->next == NULL){//到最后一个节点，我们找的是i - 1，说明没有i
        return false;
    }
    LNode *q = p->next;
    e = q->data;
    printf("被删除的节点是: %d\n",e);
    p->next = q->next;
    free(q);
    return true;
}

```

7.输出操作：PrintList

遍历输出

```

//13.打印
void PrintList(LinkList L)
{
    L = L->next;
    printf("表中的数据为: ");
    for(L,L->next != NULL,L=L->next)
    {
        printf("%d ", L->data);
        while (L->next != NULL)
        {
            L = L->next;
            printf("%d ", L->data);
        }
    }
    printf("\n");
}

```

```

//13.打印
void PrintList(LinkList L)
{
    L = L->next;
    printf("表中的数据为: ");
    //不为空打印
    if (L != NULL)
    {
        printf("%d ", L->data);
        while (L->next != NULL)
        {
            L = L->next;
            printf("%d ", L->data);
        }
    }
    printf("\n");
}

```

8. 判空: Empty

L->next == null

```
8 //2. 判断表是否为空
9 bool listEmpty(LinkList L) {
10     if(L->next) {
11         return true;
12     }
13     else{
14         return false;
15     }
16 }
```

```
//2.判断表是否为空
bool Empty(LinkList L){
    if(L->next){
        return true;
    }
    else{
        return false;
    }
}
```

9. 销毁: DestoryList

1. 用p记录L

2. L后移

3. 释放p

```
27 //3. 销毁
28 bool destroyList(LinkList &L) {
29     LNode *p; //指向一个节点LinkList p也可以，不过为了便于理解用前一个
30     while(L) {
31         p = L;
32         L = L->next;
33         free(p); //delete p也行
34     }
35     return true;
36 }
```

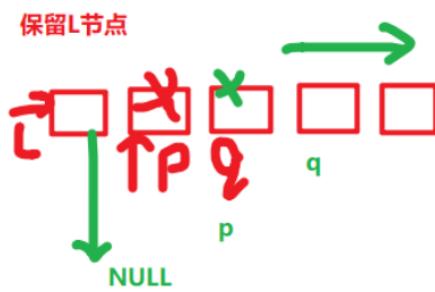


```
//3. 销毁
bool DestroyList(LinkList &L){
    LNode *p; //指向一个节点LinkList p也可以，不过为了便于理解用前一个
    while(L){
        p = L;
        L = L->next;
        free(p); //delete p也行
    }
    return true;
}
```

10.清空：ClearList

- 1.p = L->next开始
- 2.q = p->next
- 3.释放p, 将q赋值给p
- 4.最后L->next设置为NULL

```
38 //4. 清空
39 bool ClearList(LinkList &L) {
40     LNode *p, *q;
41     p = L->next;
42     while(p) {
43         q = p->next;
44         free(p); //delete p;
45         p = q;
46     }
47     L->next = NULL;
48     return true;
49 }
```



```
//4.清空
bool ClearList(LinkList &L){
    LNode *p,*q;
    p = L->next;
    while(p){
        q = p->next;
        free(p); //delete p;
        p = q;
    }
    L->next = NULL;
    return true;
}
```

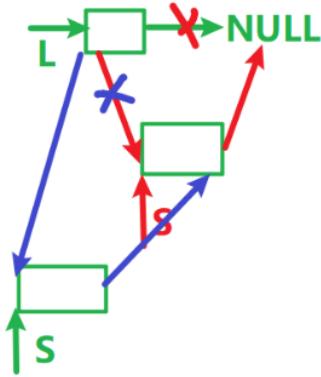
11.头插法：ListHeaderInsert

- 1.申请头节点L, L->next == NULL
- 2.申请节点S, s->next = L->next
- 3.L->next = s

```

155     return L;
156 }
157 //11. 头插法
158 LinkList List_HeadInsert(LinkList &L) {
159     LNode *s;
160     int x;
161     L = (LinkList)malloc(sizeof(LNode));
162     L->next = NULL;
163     scanf("%d", &x);
164     while(x != 9999) {
165         s = (LNode *)malloc(sizeof(LNode));
166         s->data = x;
167         s->next = L->next;
168         L->next = s;
169         scanf("%d", &x);
170     }
171     return L;
172 }
173 //13. 打印
174 void PrintList(LinkList L)
175 {
176     L = L->next;
177     printf("表中的数据为: ");
178     if (L != NULL)
179     {

```



```

//11.头插法
LinkList List_HeadInsert(LinkList &L){
    LNode *s;
    int x;
    L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;
    scanf("%d",&x);
    while(x != 9999){
        s = (LNode *)malloc(sizeof(LNode));
        s->data = x;
        s->next = L->next;
        L->next = s;
        scanf("%d",&x);
    }
    return L;
}

```

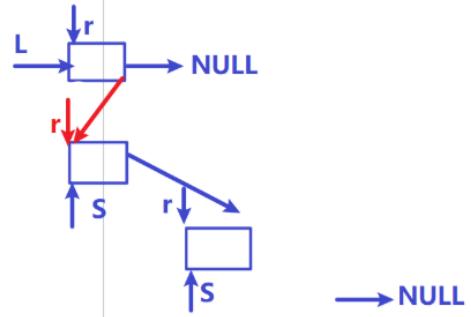
12.尾插法：ListTailInsert

- 1.申请头节点L,R尾指针同时指向头节点
- 2.申请节点S
- 3.R->next = s
- 4.移动R到S: r = s
- 5.设置r->next == NULL

```

140 }
141 //10尾插法
142 LinkList List_TailInsert(LinkedList &L) {
143     int x;
144     L = (LinkList)malloc(sizeof(LNode));
145     LNode *s,*r = L;
146     scanf("%d",&x);
147     while(x != 9999) {
148         s = (LNode *)malloc(sizeof(LNode));
149         s->data = x;
150         r->next = s;
151         r = s;
152         scanf("%d",&x);
153     }
154     r->next = NULL;
155     return L;
156 }
157 //11.头插法
158 LinkList List_HeadInsert(LinkedList &L) {
159     LNode *s;
160     int x;
161     L = (LinkList)malloc(sizeof(LNode));
162     L->next = NULL;
163     scanf("%d",&x);
164     while(x != 9999) {

```



```

//10尾插法
LinkList List_TailInsert(LinkedList &L){
    int x;
    L = (LinkList)malloc(sizeof(LNode));
    LNode *s,*r = L;
    scanf("%d",&x);
    while(x != 9999){
        s = (LNode *)malloc(sizeof(LNode));
        s->data = x;
        r->next = s;
        r = s;
        scanf("%d",&x);
    }
    r->next = NULL;
    return L;
}

```

2.不带头节点

1.初始化：InitList

指针设置为空即可

<pre> //1. 不带头节点的初始化 void InitList(LinkedList &L) { L = NULL; } //2. 获取单链表长度 int Length(LinkedList L) { int length = 0; if (L == NULL) //首先判断第一个结点是否为空 return length; else length++; while (L->next != NULL) length++; </pre>	1.头指针直接设置为空 $L \rightarrow \text{NULL}$
---	---

```

void InitList(LinkedList &L)
{
    L = NULL;
}

```

2.求表长：Length

1.判断是否为空，如果为空直接返回，否则length = 1；

2.遍历剩下的length - 1个，累加

```
16 ]
17
18 //2.获取单链表长度
19 int Length(LinkList L)
20 {
21     int length = 0;
22     if (L == NULL) //首先判断第一个结点是否为空
23         return length;
24     else
25         length++;
26     while (L->next != NULL)
27     {
28         length++;
29         L = L->next;
30     }
31     return length;
32 }
33
34 //3.判断的单链表是否为空表
35 bool Empty(LinkList L)
36 {
37     if (L == NULL)
38         return true;
39     else
40         return false;
41 }
```

```
//2.获取单链表长度
int Length(LinkList L)
{
    int length = 0;
    if (L == NULL) //首先判断第一个结点是否为空
        return length;
    else
        length++;
    while (L->next != NULL)
    {
        length++;
        L = L->next;
    }
    return length;
}
```

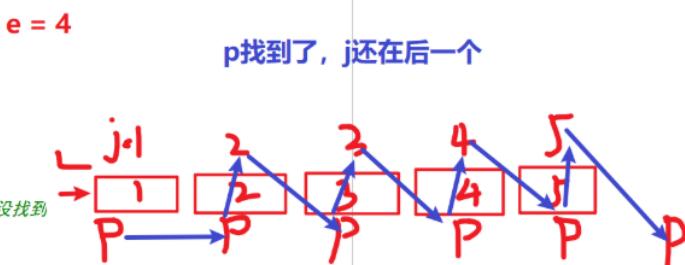
3.按值查找：LocateElem

1.判断第一个是不是要查找的元素，如果是直接返回，不是则下一个

2.循环找到e

3.返回的j+1是不是有什么问题，最后一个的这个是否判断条件需要加上p != NULL;

```
117 int LocatedElem(LinkList L, ELEMType e)
118 {
119     int j = 1;
120     LNode *p = L;
121     if (p->data == e)
122         return j;
123     else
124         p = p->next;
125     while (p != NULL && p->data != e)
126     {
127         p = p->next;
128         j++;
129     }
130     if (j == Length(L)) //到最后一位依然没找到
131     {
132         return 0;
133     }
134     return j + 1;
135 }
136
137 //得到表中某一位置的结点
138 LNode *GetElem_LNode(LinkList L, int i)
139 {
140     LNode *p = L;
141     if (i < 0 || i > Length(L))
```



```

//在表中查找第一个与查找元素相同的结点的位置
int LocatedElem(LinkList L, ElemType e)
{
    int j = 1;
    LNode *p = L;
    if (p->data == e)
        return j;
    else
        p = p->next;
    while (p != NULL && p->data != e)
    {
        p = p->next;
        j++;
    }
    if (j == Length(L)) //到最后一位依然没找到
    {
        return 0;
    }
    return j + 1;
}

```

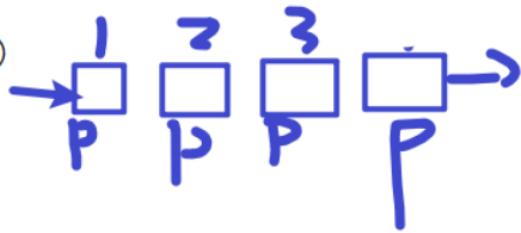
4.按位查找：GetElem

- 1.判断位置是否合法
- 2.循环找到位置
- 3.返回i位置的元素

```

149 //获得某一位置元素的值
150 ELEMType GetElem(LinkList L, int i)
151 {
152     LNode *p = L;
153     if (i < 0 || i > Length(L))
154         return 0;
155     for (int j = 1; j < i; j++)
156     {
157         p = p->next;
158     }
159     return p->data;
160 }

```



```

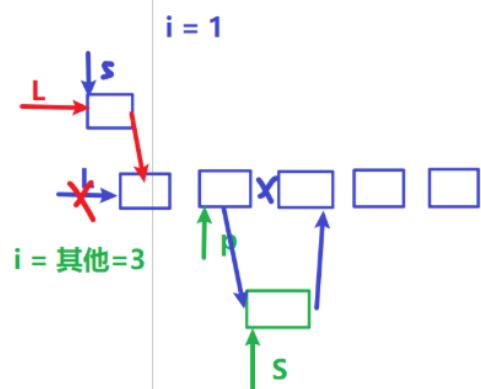
//获得某一位置元素的值
int GetElem(LinkList L, int i)
{
    LNode *p = L;
    if (i < 0 || i > Length(L))
        return 0;
    for (int j = 1; j < i; j++)
    {
        p = p->next;
    }
    return p->data;
}

```

5.插入ListInsert

- 1.判断位置是否合法
- 2.申请节点放入数据
- 3.特殊处理第一位置: S->next = L; L = S;
- 4.其他所有节点处理方式一致: 找到 i - 1 节点; s->next = p -> next; p->next = s;

```
159     return p->data;
160 }
161
162 //向表中插入数据
163 bool ListInsert(LinkList &L, int i, ElemType e)
164 {
165     if (i < 1 || i > Length(L) + 1)
166         return false;
167     LNode *s = (LNode *)malloc(sizeof(LNode));
168     s->data = e;
169     if (i == 1)
170     {
171         s->next = L;
172         L = s;
173         return true;
174     }
175     LNode *p = GetElem_LNode(L, i - 1);
176     s->next = p->next;
177     p->next = s;
178     return true;
179 }
180
181 //删除表中某位置的数据
182 bool ListDelete(LinkList &L, int i, ElemType &e)
183 {
```



```
//向表中插入数据
bool ListInsert(LinkList &L, int i, ElemType e)
{
    if (i < 1 || i > Length(L) + 1)
        return false;
    LNode *s = (LNode *)malloc(sizeof(LNode));
    s->data = e;
    if (i == 1)
    {
        s->next = L;
        L = s;
        return true;
    }
    LNode *p = GetElem_LNode(L, i - 1);
    s->next = p->next;
    p->next = s;
    return true;
}
```

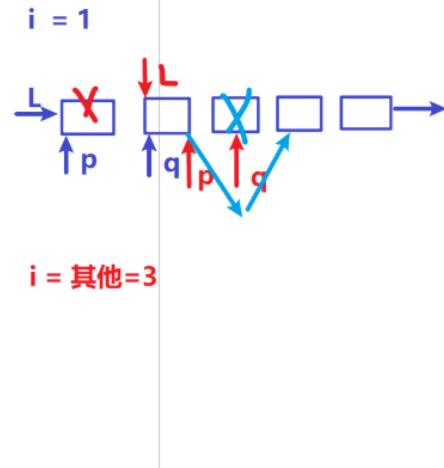
6.删除ListDelete

- 1.判断位置是否合法
- 2.判断表是否为空
- 3.p和q
- 4.特殊处理第一个节点: p指向L, q指向L->next; 释放p, 将q赋值给L
- 5.其他节点统一: 找到i - 1 节点; , q 为 节点, 跳过节点

```

190 //删除表中某位置的数据
191 bool ListDelete(LinkList &L, int i, ElemtType &e)
192 {
193     if (i < 1 || i > Length(L)) //位置不在表中则返回false
194         return false;
195     if (L == NULL) //表为空则返回false
196         return false;
197     LNode *p;
198     LNode *q;
199     if (i == 1) //删除第一个结点的情况
200     {
201         p = L;
202         e = p->data;
203         q = L->next;
204         free(p); //释放p结点
205         L = q;
206         return true;
207     }
208     p = GetElem_LNode(L, i - 1);
209     q = p->next;
210     e = q->data;
211     p->next = q->next;
212     free(q);
213     return true;
214 }

```



```

//删除表中某位置的数据
bool ListDelete(LinkList &L, int i, ElemtType &e)
{
    if (i < 1 || i > Length(L)) //位置不在表中则返回false
        return false;
    if (L == NULL) //表为空则返回false
        return false;
    LNode *p;
    LNode *q;
    if (i == 1) //删除第一个结点的情况
    {
        p = L;
        e = p->data;
        q = L->next;
        free(p); //释放p结点
        L = q;
        return true;
    }
    p = GetElem_LNode(L, i - 1);
    q = p->next;
    e = q->data;
    p->next = q->next;
    free(q);
    return true;
}

```

7.输出操作：PrintList

1.先判断L是否为空，不为空打印再循环输出其他

```

207 //打印单链表的所有数据
208 void PrintList(LinkList L)
209 {
210     printf("表中的数据为: ");
211     if (L != NULL)
212     {
213         printf("%d ", L->data);
214         while (L->next != NULL)
215         {
216             L = L->next;
217             printf("%d ", L->data);
218         }
219     }
220     printf("\n");
221 }

```

```

//打印单链表的所有数据
void PrintList(LinkList L)
{
    printf("表中的数据为: ");
    if (L != NULL)
    {
        printf("%d ", L->data);
        while (L->next != NULL)
        {
            L = L->next;
            printf("%d ", L->data);
        }
    }
    printf("\n");
}

```

8. 判空：Empty

1. L == NULL

```

34 //3. 判断的单链表是否为空表
35 bool Empty(LinkList L)
36 {
37     if (L == NULL)
38         return true;
39     else
40         return false;
41 }

```

```
//3.判断的单链表是否为空表
bool Empty(LinkList L)
{
    if (L == NULL)
        return true;
    else
        return false;
}
```

9.销毁：DestoryList

1.p记录L，从L开始

```
222 //销毁
223 bool DestoryList(LinkList &L) {
224     LNode *p; //指向一个节点LinkList p也可以，不过为了便于理解用前一个
225     while(L) {
226         p = L;
227         L = L->next;
228         free(p); //delete p也行
229     }
230     return true;
231 }
232
```

```
//3.销毁
bool DestoryList(LinkList &L){
    LNode *p; //指向一个节点LinkList p也可以，不过为了便于理解用前一个
    while(L){
        p = L;
        L = L->next;
        free(p); //delete p也行
    }
    return true;
}
```

10.清空：ClearList

1.判断是否为空表

2.删除除第一个外的节点：判断L->next是否为空，p=L->next； L->next移动到下一个，删除p

3.删除第一个节点

```

233 //清空表的数据
234 bool ClearList(LinkList &L)
235 {
236     if (L == NULL) //表空则返回false
237         return false;
238     LNode *p;
239     while (L->next != NULL) //删除第一个结点之后的所有结点
240     {
241         p = L->next;
242         L->next = p->next;
243         free(p);
244     }
245     L = NULL; //删除第一个结点
246     return true;
247 }

```

```

233 //清空表的数据
234 bool ClearList(LinkList &L)
235 {
236     if (L == NULL) //表空则返回false
237         return false;
238     LNode *p;
239     while (L->next != NULL) //删除第一个结点之后的所有结点
240     {
241         p = L->next;
242         L->next = p->next;
243         free(p);
244     }
245     L = NULL; //删除第一个结点
246     return true;
247 }

```



```

//清空表的数据
bool ClearList(LinkList &L)
{
    if (L == NULL) //表空则返回false
        return false;
    LNode *p;
    while (L->next != NULL) //删除第一个结点之后的所有结点
    {
        p = L->next;
        L->next = p->next;
        free(p);
    }
    L = NULL; //删除第一个结点
    return true;
}

```

11.头插法：ListHeaderInsert

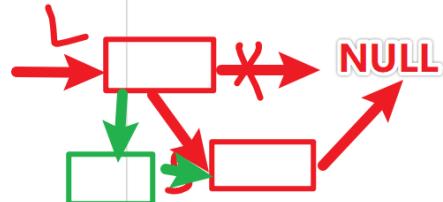
- 1.申请节点L，并赋值
- 2.循环插入节点和带头节点的一样

```

43 //4.头插法建立单链表
44 LinkList List_HeadInsert(LinkList &L)
{
    InitList(L); //初始化
    LNode *s;
    ElemType e;
    printf("输入数据9999停止: \n");
    scanf("%d", &e);
    if (e != 9999) // 首先对第一个结点赋值
    {
        L = (LinkList)malloc(sizeof(LNode));
        L->next = NULL;
        L->data = e;
        scanf("%d", &e);
        while (e != 9999)
        {
            s = (LNode *)malloc(sizeof(LNode));
            s->data = e;
            s->next = L->next;
            L->next = s;
            scanf("%d", &e);
        }
    }
    return L;
}
//尾插法建立单链表

```

第一个节点赋值，这样下面的就相当于带有都节点



```

43 //4.头插法建立单链表
44 LinkList List_HeadInsert(LinkList &L)
{
    InitList(L); //初始化
    LNode *s;
    ElemType e;
    printf("输入数据9999停止: \n");
    scanf("%d", &e);
    if (e != 9999) // 首先对第一个结点赋值
    {
        L = (LinkList)malloc(sizeof(LNode));
        L->next = NULL;
        L->data = e;
        scanf("%d", &e);
        while (e != 9999)
        {
            s = (LNode *)malloc(sizeof(LNode));
            s->data = e;
            s->next = L->next;
            L->next = s;
            scanf("%d", &e);
        }
    }
    return L;
}
//尾插法建立单链表

```

第一个节点赋值，这样下面的就相当于带有都节点

```

//4.头插法建立单链表
LinkList List_HeadInsert(LinkList &L)
{
    InitList(L); //初始化
    LNode *s;
    ElemType e;
    printf("输入数据9999停止: \n");
    scanf("%d", &e);
    if (e != 9999) // 首先对第一个结点赋值
    {
        L = (LinkList)malloc(sizeof(LNode));
        L->next = NULL;
        L->data = e;
        scanf("%d", &e);
        while (e != 9999)
        {
            s = (LNode *)malloc(sizeof(LNode));
            s->data = e;
            s->next = L->next;
            L->next = s;
            scanf("%d", &e);
        }
    }
    return L;
}

```

12. 尾插法：ListTailInsert

1. 给第一个节点赋值，其他统一视为带头节点的一样

```
//尾插法建立单链表
LinkList List_TailInsert(LinkList &L)
{
    InitList(L); //初始化
    LNode *s, *r;
    ElemType e;
    printf("输入数据9999停止: \n");
    scanf("%d", &e);
    if (e != 9999) //首先对第一个结点赋值
    {
        L = (LinkList)malloc(sizeof(LNode));
        L->data = e;
        r = L;
        scanf("%d", &e);
        while (e != 9999)
        {
            s = (LNode *)malloc(sizeof(LNode));
            s->data = e;
            r->next = s;
            r = s;
            scanf("%d", &e);
        }
    }
    r->next = NULL;
    return L;
}
```

第一个节点赋值，其他统一

```
//尾插法建立单链表
LinkList List_TailInsert(LinkList &L)
{
    InitList(L); //初始化
    LNode *s, *r;
    ElemType e;
    printf("输入数据9999停止: \n");
    scanf("%d", &e);
    if (e != 9999) //首先对第一个结点赋值
    {
        L = (LinkList)malloc(sizeof(LNode));
        L->data = e;
        r = L;
        scanf("%d", &e);
        while (e != 9999)
        {
            s = (LNode *)malloc(sizeof(LNode));
            s->data = e;
            r->next = s;
            r = s;
            scanf("%d", &e);
        }
    }
    r->next = NULL;
    return L;
}
```

3. 静态链表

静态链表

静态链表：线性存储结构的一种，兼顾顺序表和链表的优点，是顺序表和链表的升级；静态链表的数据全部存储在数组中（顺序表），但存储的位置是随机的，数据直接的一对一关系是通过一个整型变量（称为“游标”，类似指针的功能）维持。

1. 节点

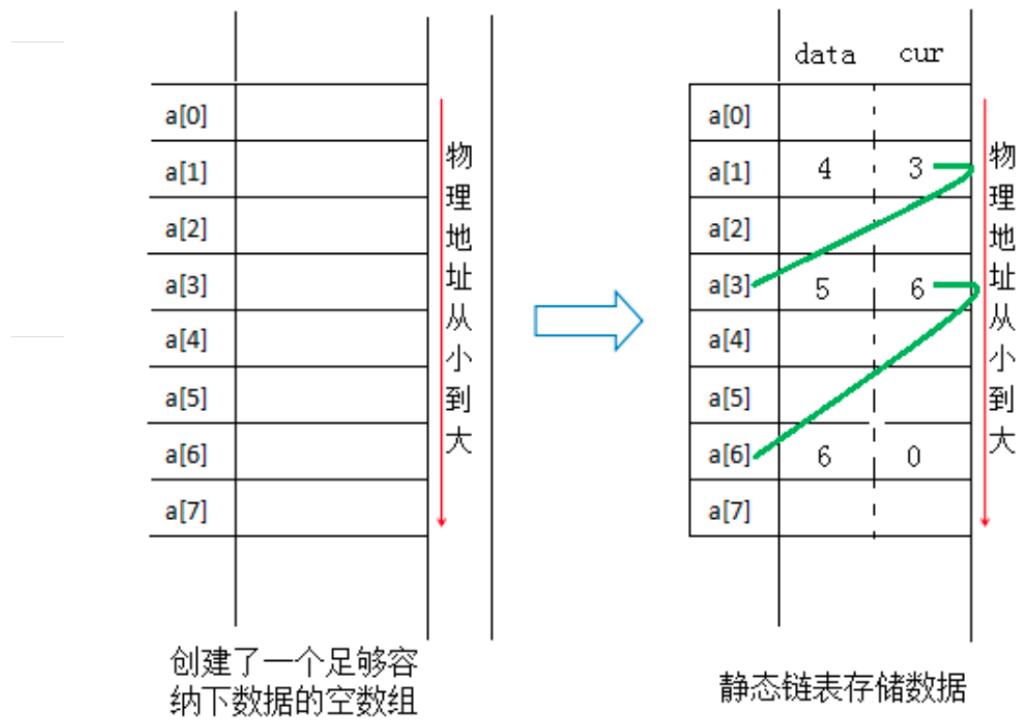
1. 静态链表中的节点

数据域：用于存储数据元素的值

游标：即数组下标，表示直接后继元素所在数组中的位置

```
typedef struct
{
    int data; //静态链表节点中的数据
    int cur;   //静态链表节点中的游标
}component;
```

例：使用静态链表存储数据元素4、5、6，过程如下：



注：通常静态链表会将第一个数据元素放到数组下标为1(即 $a[1]$)的位置中。

图中从 $a[1]$ 存储的数据元素4开始，通过存储的游标变量3，可以在 $a[3]$ 中找到元素4的直接后继元素5；通过元素 $a[3]$ 存储的游标变量6，可在 $a[6]$ 中找到元素5的直接后继元素6；这样一直到某元素的游标变量为0截止($a[0]$ 默认不存储数据元素)

2. 备用链表

2. 备用链表

静态链表中，除了数据本身通过游标组成链表外，还需要有一条连接各个空闲位置的链表，称为备用链表。

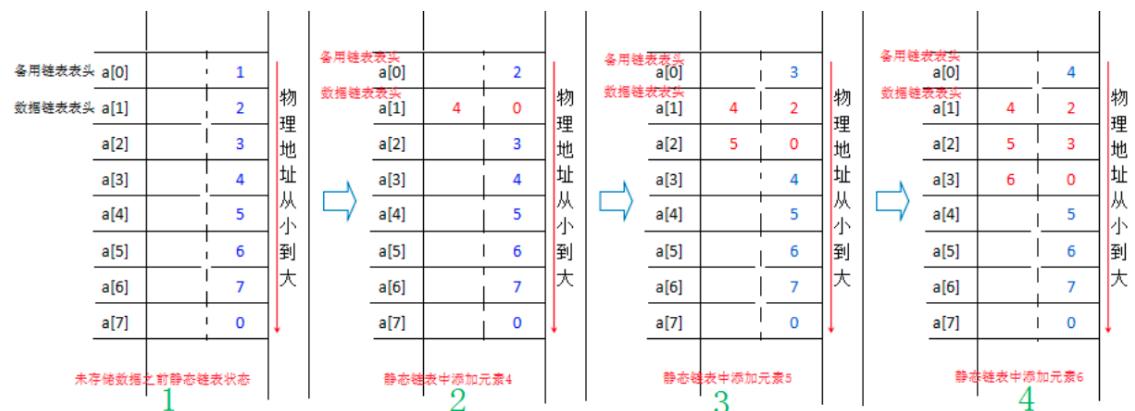
作用：回收数组中未使用或者之前使用过(现在不用)的存储空间，留待后期使用。即静态链表使用数组申请的物理空间中，存在两个链表，一条连接数据，另一条连接数组中为使用的空间。

注：通常备用链表的表头位于数组下标为0($a[0]$)的位置，而数据链表的表头位于数组下标为1($a[1]$)的位置。

静态链表中设置备用链表的好处是，可以清楚地知道数组中是否有空闲位置，以便数据链表添加新数据时使用。比如，若静态链表中数组下标为0的位置上存有数据，则证明数组已满。

3. 静态链表的实现

3 静态链表的实现



在数据链表未初始化之前，数组中所有位置都处于空闲状态，所以都链接在备用链表上。(图1)

1. 在数据链表未初始化之前，数组中所有位置都处于空闲状态，所以都链接在备用链表上。(图1)
2. 向静态链表中添加数据时，需提前从备用链表中摘除结点，让新数据使用。
3. 备用链表摘除节点最简单的方法是摘除 $a[0]$ 的直接后继节点(即摘除 $a[1]$ 的游标2)：获取 $a[0]$ 指向的空闲的第一个节点

同样，向备用链表中添加空闲节点也是添加作为 $a[0]$ 新的直接后继节点(

图2中 $a[1]$ 为 $a[0]$ 新的直接后继点(游标为2)、

图3中 $a[2]$ 为 $a[0]$ 新的直接后继点(游标为3)、

图4中 $a[3]$ 为 $a[0]$ 新的直接后继点(游标为4))。

因为 $a[0]$ 是备用链表的第一个节点，我们知道它的位置，操作它的直接后继节点相对容易，无需遍历备用链表，耗费的时间复杂度为 $O(1)$ 。

```

/***********************
创建备用链表
************************/
void reserveArr(component *array) {
    int i = 0;
    for (i = 0; i < maxSize; i++) {
        array[i].cur = i + 1;//将每个数组分量链接到一起
        array[i].data = 0;
    }
    array[maxsize - 1].cur = 0;//链表最后一个结点的游标值为0
}//做的事如图

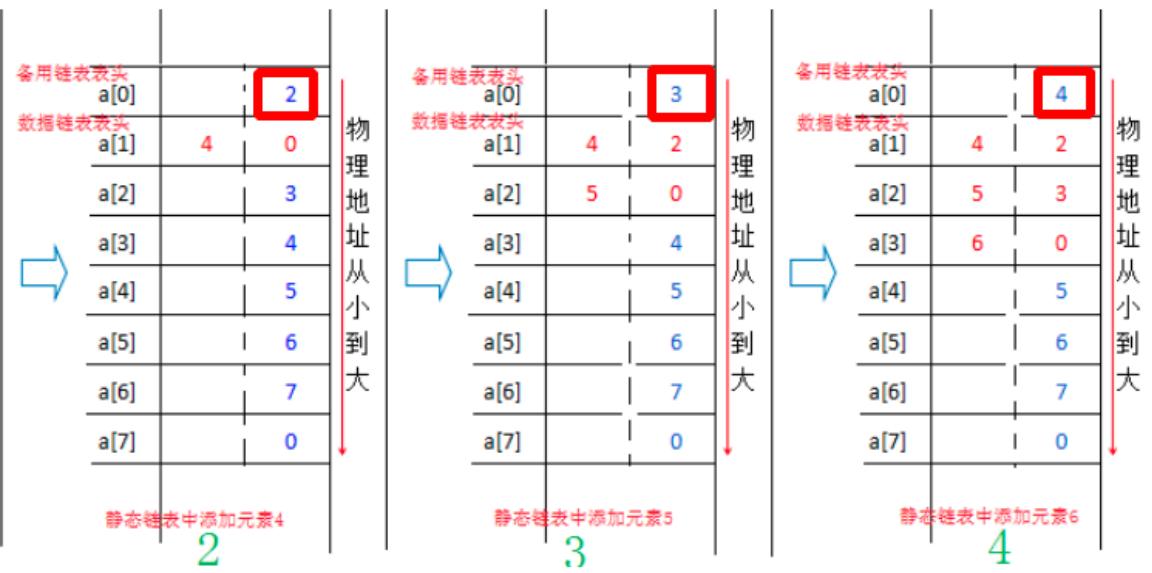
```



```

/***********************
提取分配空间
若备用链表为非空，则返回分配的节点下标，否则返回0(当分配最后一个节点时，该节点的游标值为0)
************************/
int mallocArr(component * array) {
    //若备用链表非空，则返回分配的结点下标，否则返回 0 (当分配最后一个结点时，该结点的游标值为
    0)
    int i = array[0].cur;//a[0]指向第一个空闲的空间
    //更新a[0]指向下一个空闲的空间
    if (array[0].cur) {
        array[0].cur = array[i].cur;
    }
    return i;
}//通过a[0]找到第一个空闲的空间， cur为0时为最后一个元素

```



```

/*
*****初始化静态链表*****
int initArr(component *array) {
    // 尾指针
    int tail = 0;
    // 头指针
    int head = 0;

    int i = 0;
    // 初始化备用链表
    reserveArr(array);
    // 提取分配空间
    head = mallocArr(array); //第一次时body=1, a【0】.cur = 1;

    //建立首元结点
    array[head].data = 1;
    array[head].cur = 0; //只有一个元素时设置a【1】为最后一个节点

    //声明一个变量，把它当指针使，指向链表的最后的一个结点，当前和首元结点重合
    tail = head; //第一遍时设置尾指针指向a[1]

    // 分配2-5的空间
    for (i = 2; i < 6; i++) {
        // a【0】.cur即指向空闲的第一个空间
        int j = mallocArr(array); //从备用链表中拿出空闲的分量

        printf("获取的备用链表空间j是%d\n\n", j);
        array[j].data = i; //初始化新得到的空间结点

        printf("存入j空间的元素是%d\n\n", i);
        printf("tail的位置是%d\n\n", tail);
        array[tail].cur = j; //将新得到的结点链接到数据链表的尾部

        tail = j; //将指向链表最后一个结点的指针后移
    }
    array[tail].cur = 0; //新的链表最后一个结点的指针设置为0
    return head; //第一个元素所在的位置
}

```

```

// 初始化静态链表
int initArr(component *array) {
    int tempBody = 0;
    int body = 0;
    int i = 0;

    // 初始化备用链表
    reserveArr(array);
    // 申请空闲空间
    body = mallocArr(array); //body = 1
    // 建立首元结点
    array[body].data = 1;
    array[body].cur = 0;

    // 声明一个变量，把它当指针使，指向链表的最后的一个结点，当前和首元结点重合
    tempBody = body;
    for (i = 2; i < 4; i++) {
        /*
         * 1. 循环中j = 2, 3
         */
        int j = mallocArr(array); // 从备用链表中拿出空闲的分量
        array[j].data = i; // 初始化新得到的空间结点
        array[tempBody].cur = j; // 将新得到的结点链接到数据链表的尾部
        tempBody = j; // 将指向链表最后一个结点的指针后移
    }
    array[tempBody].cur = 0; // 新的链表最后一个结点的指针设置为0
    return body;
}

```

静态链表为：
1,2
2,3
3,0
在第3的位置上插入元素4:
1,2
2,3
3,4
4,0
删除数据域为1的结点:
2,3
3,4
4,0
查找数据域为4的结点的位置:
4
将结点数据域为4改为5:
2,3
3,4
5,0
[1] + Done

```

// 输出静态链表，通过游标遍历，直到游标等于0停止
void displayArr(component * array, int head) {
    int index = head; // index准备做遍历使用，从a【1】开始
    while (array[index].cur) {
        printf("%d,%d\n", array[index].data, array[index].cur);
        index = array[index].cur;
    }
    // 输出最后等于0的时候的数组元素
    printf("%d,%d\n", array[index].data, array[index].cur);
}

```

```

datastruct > 10 静态链表 > static.cpp > displayArr(component *, int)
86     array[j].data = i; // 初始化新得到的空间结点
87
88     printf("存入j空间的元素是%d\n", i);
89     printf("tail的位置是是%d\n", tail);
90     array[tail].cur = j; // 将新得到的结点链接到数据链表的尾部
91
92     tail = j; // 将指向链表最后一个结点的指针后移
93 }
94 array[tail].cur = 0; // 新的链表最后一个结点的指针设置为0
95 return head; // 第一个元素所在的位置
96 }
97
98 void displayArr(component * array, int head) {
99     int index = head; // index准备做遍历使用，从a【1】开始
100    while (array[index].cur) {
101        printf("%d,%d\n", array[index].data, array[index].cur);
102        index = array[index].cur;
103    }
104    printf("%d,%d\n", array[index].data, array[index].cur);
105 }
106 }


```

获取的备用链表空间j是4
存入j空间的元素是4
tail的位置是是3
获取的备用链表空间j是5
存入j空间的元素是5
tail的位置是是4
静态链表为：
1,2
2,3
3,4
4,5
5,0
[1] + Done

4. 添加元素

静态链表添加元素

例如，在图 1 的基础，将元素 4 添加到静态链表中的第 3 个位置上，实现过程如下：

1. 从备用链表中摘除一个节点，用于存储元素 4；
2. 找到表中第 2 个节点（添加位置的前一个节点，这里是数据元素 2），将元素 2 的游标赋值给新元素 4；
3. 将元素 4 所在数组中的下标赋值给元素 2 的游标；

```

datastruct > 10静态链表 > staticAll.cpp > insertArr(component * int, int, int)
89     int j = mallocArr(array); //从备链表中拿出空闲的分量
90     array[j].data = i; //初始化新得到的空间结点
91     array[tempBody].cur = j; //将新得到的结点链接到数据链表的尾部
92     tempBody = j; //将指向链表最后一个结点的指针后移
93 }
94 array[tempBody].cur = 0; //新的链表最后一个结点的指针设置为0
95 return body;
96 }

//向链表中插入数据, body表示链表的头结点在数组中的位置, add表示插入元素的位置, num表示要插入的数据
97 void insertArr(component * array, int body, int add, int num) {
98     int tempBody = body; //tempBody做遍历结构体数组使用
99     int i = 0, insert = 0;
100    //找到要插入位置的上一个结点在数组中的位置:找到前驱的位置
101    for (i = 1; i < add - 1; i++) {
102        tempBody = array[tempBody].cur;
103    }
104    insert = mallocArr(array); //申请空间, 准备插入
105    array[insert].data = num;
106    array[insert].cur = array[tempBody].cur; //新插入结点的游标等于其直接前驱结点的游标
107
108    array[tempBody].cur = insert; //直接前驱结点的游标等于新插入结点所在数组中的下标
109
110 }
111
112 }

//删除结点函数, num表示被删除结点中存放的数据
113 void deleteArr(component * array, int body, int num) {
114     int tempBody = body;
115     int del = 0;
116     int newbody = 0;
117
118     //找到被删除结点的位置
119     while (array[tempBody].data != num) {
120         tempBody = array[tempBody].cur;
121     }
122 }

//终端将被任务重用, 按任意键关闭。

```

```

文件 编辑 选择 查看 转到 运行 终端 帮助
资源管理器 ... 多文件 < BF.cpp < static.cpp < staticAll.cpp < SqList.cpp .../01静态分配顺序表 < method.h .../01静态分配顺序表 < main.cpp .../01静态分配顺序表 < SqList.h .../01静态分配顺序表
< C语言
> OS队列
> 06栈
> 第04章: 串
> 09双链表
< 10静态链表
< static.cpp
< staticAll.cpp
readme.md
header.out
header.out
SqList.out
SqQueue.out
SqStack.out
大明
maxSize
component typedef
< _unnamed_struct...
reserveArr(component...
initArr(component...
displayArr(component...
mallocArr(component...
insertArr(component...
main()
reserveArr(component...
mallocArr(component...
initArr(component...
displayArr(component...
Insert(component...
tail的位置是是4
head is 1
静态链表为:
1,2
2,3
3,4
4,5
5,0
当前节点的cur is 3
空闲空间是:6
插入位置元素的游标为:3
1,2
2,6
7,3
3,4
4,5
5,0
[1] + Done
"/usr/bin/gdb" --interpreter=mi
] 0<"./tmp/Microsoft-MIEngine-In-eZptvz4l.omb" 1>"./tmp/Microsoft-MIEngine-Out
0tdj2.log"
按任意键继续...

```

```

//向链表中插入数据, body表示链表的头结点在数组中的位置, add表示插入元素的位置, num表示要插入的数据
void insertArr(component * array, int body, int add, int num) {
    int tempBody = body; //tempBody做遍历结构体数组使用
    int i = 0, insert = 0;
    //找到要插入位置的上一个结点在数组中的位置:找到前驱的位置
    for (i = 1; i < add - 1; i++) {
        tempBody = array[tempBody].cur;
    }
    insert = mallocArr(array); //申请空间, 准备插入
    array[insert].data = num;

    array[insert].cur = array[tempBody].cur; //新插入结点的游标等于其直接前驱结点的游标

    array[tempBody].cur = insert; //直接前驱结点的游标等于新插入结点所在数组中的下标
}

```

5. 删除元素

静态链表中删除指定元素，只需实现以下 2 步操作：

1. 将存有目标元素的节点从数据链表中摘除；
2. 将摘除节点添加到备用链表，以便下次再用；

比较特殊的是，对于无头结点的数据链表来说，如果需要删除头结点，则势必会导致数据链表的表头不再位于数组下标为 1 的位置，换句话说，删除头结点之后，原数据链表中第二个结点将作为整个链表新的首元结点。

若问题中涉及大量删除元素的操作，建议读者在建立静态链表之初创建一个带有头节点的静态链表，方便实现删除链表中第一个数据元素的操作。

```
datastruct > 10静态链表 > static.cpp > freeArr(component * array)
141     printf("空闲空间是:%d\n", insert);
142
143     array[insert].data = a;
144
145     array[insert].cur = array[p].cur; //新插入结点的游标等于其直接前驱结点的游标
146     printf("插入位置元素的游标为 :%d\n", array[insert].cur);
147
148     array[p].cur = insert; //直接前驱结点的游标等于新插入结点所在数组中的下标
149 }
150
151 //备用链表回收空间的函数，其中array为存储数据的数组，k表示未使用节点所在数组的下标
152 void freeArr(component * array, int k) {
153     /*
154      | 1. 如果第一个节点就是,
155      |   a[1].cur = 5
156      |   a[0].cur = 1;
157      */
158     array[k].cur = array[0].cur;
159     array[0].cur = k;
160 }
161
162 //删除结点函数，num表示被删除结点中数据域存放的数据，函数返回新数据链表的表头位置
163 int deleteArr(component * array, int head, int num) {
164     int p = head;
165     int del = 0;
166     int newhead = 0;
167 }
```



```
//删除结点函数，num表示被删除结点中数据域存放的数据
int deleteArr(component * array, int body, int num) {
    // 用于遍历
    int tempBody = body;
    // 标记删除节点位置
    int del = 0;
    // 用于无头结点的，删除第一个节点
    int newbody = 0;

    //找到被删除结点的位置
    while (array[tempBody].data != num) {
        tempBody = array[tempBody].cur;
        //当tempBody为0时，表示链表遍历结束，说明链表中没有存储该数据的结点
        if (tempBody == 0) {
            printf("链表中没有此数据");
            return body;
        }
    }

    //运行到此，证明有该结点
    del = tempBody;
    tempBody = body;

    //删除首元结点，需要特殊考虑
    if (del == body) {
        newbody = array[del].cur;
        freeArr(array, del);
        return newbody;
    }

    else
    {
        //找到该结点的上一个结点，做删除操作
        while (array[tempBody].cur != del) {
```

```
    tempBody = array[tempBody].cur;
}
//将被删除结点的游标直接给被删除结点的上一个结点
array[tempBody].cur = array[del].cur;
//回收被摘除节点的空间
freeArr(array, del);
return body;
}
}
```

```
//备用链表回收空间的函数，其中array为存储数据的数组，k表示未使用节点所在数组的下标
void freeArr(component * array, int k) {
    array[k].cur = array[0].cur;//第一个空闲的空间和被删除的节点链接起来
    array[0].cur = k;//将a[0]和被删除节点的空间链接起来
}
```

6.查找元素

静态链表查找元素

静态链表查找指定元素，由于我们只知道静态链表第一个元素所在数组中的位置，因此只能通过逐个遍历静态链表的方式，查找存有指定数据元素的节点。

```
//在以body作为头结点的链表中查找数据域为elem的结点在数组中的位置
int selectNum(component * array, int body, int num) {
    //当游标值为0时，表示链表结束
    while (array[body].cur != 0) {
        if (array[body].data == num) {
            return body;
        }
        body = array[body].cur;
    }
    //判断最后一个结点是否符合要求
    if (array[body].data == num) {
        return body;
    }
    return -1;//返回-1，表示在链表中没有找到该元素
}
```

7.修改元素

静态链表中更改数据

更改静态链表中的数据，只需找到目标元素所在的节点，直接更改节点中的数据域即可。

实现此操作的 C 语言代码如下：

```

//在以body作为头结点的链表中将数据域为oldElem的结点，数据域改为newElem
void amendElem(component * array, int body, int oldElem, int newElem) {
    int add = selectNum(array, body, oldElem);
    if (add == -1) {
        printf("无更改元素");
        return;
    }
    array[add].data = newElem;
}

```

8.完整代码

```

#include <stdio.h>
#define maxSize 7
typedef struct {
    int data;
    int cur;
}component;

//将结构体数组中所有分量链接到备用链表中
void reserveArr(component *array);

//初始化静态链表
int initArr(component *array);

//向链表中插入数据，body表示链表的头结点在数组中的位置，add表示插入元素的位置，num表示要插入的数据
void insertArr(component * array, int body, int add, int num);

//删除链表中存有num的结点，返回新数据链表中第一个节点所在的位置
int deletArr(component * array, int body, int num);

//查找存储有num的结点在数组的位置
int selectNum(component * array, int body, int num);

//将链表中的字符oldElem改为newElem
void amendElem(component * array, int body, int oldElem, int newElem);

//输出函数
void displayArr(component * array, int body);

//从备用链表中摘除空闲节点的实现函数
int mallocArr(component * array);

//将摘除下来的节点链接到备用链表上
void freeArr(component * array, int k);

int main() {
    component array[maxSize];
    int body = initArr(array);
    int selectAdd;
    printf("静态链表为: \n");
    displayArr(array, body);

    printf("在第3的位置上插入元素4:\n");
}

```

```

insertArr(array, body, 3, 4);
displayArr(array, body);

printf("删除数据域为1的结点:\n");
body = deletArr(array, body, 1);
displayArr(array, body);

printf("查找数据域为4的结点的位置:\n");
selectAdd = selectNum(array, body, 4);
printf("%d\n", selectAdd);
printf("将结点数据域为4改为5:\n");
amendElem(array, body, 4, 5);
displayArr(array, body);
return 0;
}

//创建备用链表
void reserveArr(component *array) {
    int i = 0;
    for (i = 0; i < maxSize; i++) {
        array[i].cur = i + 1;//将每个数组分量链接到一起
    }
    array[maxSize - 1].cur = 0;//链表最后一个结点的游标值为0
}

//初始化静态链表
int initArr(component *array) {
    int tempBody = 0, body = 0;
    int i = 0;
    reserveArr(array);
    body = mallocArr(array);
    //建立首元结点
    array[body].data = 1;
    array[body].cur = 0;
    //声明一个变量，把它当指针使，指向链表的最后的一个结点，当前和首元结点重合
    tempBody = body;
    for (i = 2; i < 4; i++) {
        int j = mallocArr(array); //从备用链表中拿出空闲的分量
        array[j].data = i; //初始化新得到的空间结点
        array[tempBody].cur = j; //将新得到的结点链接到数据链表的尾部
        tempBody = j; //将指向链表最后一个结点的指针后移
    }
    array[tempBody].cur = 0;//新的链表最后一个结点的指针设置为0
    return body;
}

//向链表中插入数据，body表示链表的头结点在数组中的位置，add表示插入元素的位置，num表示要插入的数据
void insertArr(component * array, int body, int add, int num) {
    int tempBody = body;//tempBody做遍历结构体数组使用
    int i = 0, insert = 0;
    //找到要插入位置的上一个结点在数组中的位置
    for (i = 1; i < add; i++) {
        tempBody = array[tempBody].cur;
    }
    insert = mallocArr(array);//申请空间，准备插入
    array[insert].data = num;

    array[insert].cur = array[tempBody].cur;//新插入结点的游标等于其直接前驱结点的游标
}

```

```

array[tempBody].cur = insert;//直接前驱结点的游标等于新插入结点所在数组中的下标
}

//删除结点函数，num表示被删除结点中数据域存放的数据
int deletArr(component * array, int body, int num) {
    int tempBody = body;
    int del = 0;
    int newbody = 0;

    //找到被删除结点的位置
    while (array[tempBody].data != num) {
        tempBody = array[tempBody].cur;
        //当tempBody为0时，表示链表遍历结束，说明链表中没有存储该数据的结点
        if (tempBody == 0) {
            printf("链表中没有此数据");
            return body;
        }
    }

    //运行到此，证明有该结点
    del = tempBody;
    tempBody = body;

    //删除首元结点，需要特殊考虑
    if (del == body) {
        newbody = array[del].cur;
        freeArr(array, del);
        return newbody;
    }

    else
    {
        //找到该结点的上一个结点，做删除操作
        while (array[tempBody].cur != del) {
            tempBody = array[tempBody].cur;
        }
        //将被删除结点的游标直接给被删除结点的上一个结点
        array[tempBody].cur = array[del].cur;
        //回收被摘除节点的空间
        freeArr(array, del);
        return body;
    }
}

//在以body作为头结点的链表中查找数据域为elem的结点在数组中的位置
int selectNum(component * array, int body, int num) {
    //当游标值为0时，表示链表结束
    while (array[body].cur != 0) {
        if (array[body].data == num) {
            return body;
        }
        body = array[body].cur;
    }
    //判断最后一个结点是否符合要求
    if (array[body].data == num) {
        return body;
    }
    return -1;//返回-1，表示在链表中没有找到该元素
}

```

```

}

//在以body作为头结点的链表中将数据域为oldElem的结点，数据域改为newElem
void amendElem(component * array, int body, int oldElem, int newElem) {
    int add = selectNum(array, body, oldElem);
    if (add == -1) {
        printf("无更改元素");
        return;
    }
    array[add].data = newElem;
}

void displayArr(component * array, int body) {
    int tempBody = body;//tempBody准备做遍历使用
    while (array[tempBody].cur) {
        printf("%d,%d\n", array[tempBody].data, array[tempBody].cur);
        tempBody = array[tempBody].cur;
    }
    printf("%d,%d\n", array[tempBody].data, array[tempBody].cur);
}

//提取分配空间
int mallocArr(component * array) {
    //若备用链表非空，则返回分配的结点下标，否则返回0（当分配最后一个结点时，该结点的游标值为0）
    int i = array[0].cur;
    if (array[0].cur) {
        array[0].cur = array[i].cur;
    }
    return i;
}

//备用链表回收空间的函数，其中array为存储数据的数组，k表示未使用节点所在数组的下标
void freeArr(component * array, int k) {
    array[k].cur = array[0].cur;
    array[0].cur = k;
}

```

4. 双向链表

1. 不带头结点

双向链表及其创建（C语言）详解

◀ 如何判断链表中有环？

双向链表的基本操作 >

目前我们所学到的链表，无论是动态链表还是静态链表，表中各节点中都只包含一个指针（游标），且都统一指向直接后继节点，通常称这类链表为单向链表（或单链表）。

虽然使用单链表能 100% 解决逻辑关系为“一对多”数据的存储问题，但在解决某些特殊问题时，单链表并不是效率最优的存储结构。比如说，某场景中需要大量地查找某结点的前趋结点，这种情况下使用单链表无疑是灾难性的，因为单链表更适合“从前向后”找，“从后往前”找并不是它的强项。

对于逆向查找（从后往前）相关的问题，使用本节讲解的双向链表，会更加事半功倍。

双向链表，简称双链表。从名字上理解双向链表，即链表是“双向”的，如图 1 所示：



图 1 双向链表结构示意图

所谓双向，指的是各节点之间的逻辑关系是双向的，但通常头指针只设置一个，除非实际情况需要，可以为最后一个节点再设置一个“头指针”。

1. 节点

根据图 1 不难看出，双向链表中各节点包含以下 3 部分信息（如图 2 所示）：

1. 指针域：用于指向当前节点的直接前驱节点；
2. 数据域：用于存储数据元素；
3. 指针域：用于指向当前节点的直接后继节点。



图 2 双向链表的节点构成

```
typedef struct line{  
    struct line * prior; //指向直接前趋  
    int data;  
    struct line * next; //指向直接后继  
}line;
```

2. 尾插法创建

双向链表的创建

同单链表相比，双链表仅是各节点多了一个用于指向直接前驱的指针域。因此，我们可以在单链表的基础轻松实现对双链表的创建。

和创建单链表不同的是，创建双向链表的过程中，每一个新节点都要和前驱节点之间建立两次链接，分别是：

- 将新节点的 prior 指针指向直接前驱节点；
- 将直接前驱节点的 next 指针指向新节点；



图 1 双向链表结构示意图

```
line* initLine(line * head) {  
    int i = 0;  
    line * list = NULL;  
    //创建一个首元节点，链表的头指针为head  
    head = (line*)malloc(sizeof(line));
```

```

//对节点进行初始化
head->prior = NULL;
head->next = NULL;
head->data = 1;
//声明一个指向首元节点的指针，方便后期向链表中添加新创建的节点
list = head;

for (i = 2; i <= 5; i++) {
    //创建新的节点并初始化
    line * body = (line*)malloc(sizeof(line));
    body->prior = NULL;
    body->next = NULL;
    body->data = i;
    //新节点与链表最后一个节点建立关系
    list->next = body;
    body->prior = list;
    //list永远指向链表中最后一个节点
    list = list->next;
}

//返回新创建的链表
return head;
}

```

```

...  C line.cpp  x
datastruct > 01线性表 > 07双向链表 > C line.cpp > initLine(line *)
31
32 line* initLine(line * head) {
33     //尾插法
34     int i = 0;
35     line * list = NULL;
36     //创建一个首元节点，链表的头指针为head
37     head = (line*)malloc(sizeof(line));
38
39     //对节点进行初始化
40     head->prior = NULL;
41     head->next = NULL;
42     head->data = 1;
43
44     //声明一个指向首元节点的指针，方便后期向链表中添加新创建的节点
45     list = head;
46
47     for (i = 2; i <= 5; i++) {
48         //创建新的节点并初始化
49         line * body = (line*)malloc(sizeof(line));
50         body->prior = NULL;
51         body->next = NULL;
52         body->data = i;
53         //新节点与链表最后一个节点建立关系
54         list->next = body;
55         body->prior = list;
56         //list永远指向链表中最后一个节点
57         list = list->next;
58
59     }
59     //返回新创建的链表
60     return head;
61 }
62

```

1 2 3 4 5
链表中第 4 个节点的直接前驱是: 3[1] + Done "/usr/bin/gdb"
--Interpreter=ml --tty=\$(DbgTerm) 0<"tmp/Microsoft-MIEngine-In-Seqtfchx.tsq" 1<"tmp/Microsoft-MIEngine-Out-a4rzq4gv.gdw"
按任意键继续...[]

3.添加

1.表头添加

1) 添加至表头

将新数据元素添加到表头，只需要将该元素与表头元素建立双层逻辑关系即可。

换句话说，假设新元素节点为 temp，表头节点为 head，则需要做以下 2 步操作即可：

1. `temp->next=head; head->prior=temp;`；
2. 将 head 移至 temp，重新指向新的表头；

例如，将新元素 7 添加至双向链表的表头，则实现过程如图 2 所示：

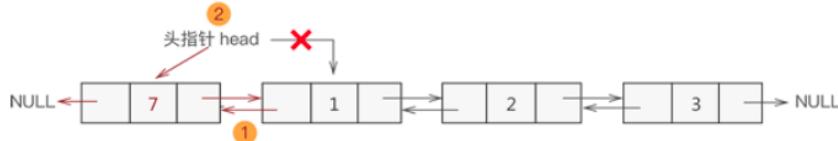


图 2 添加元素至双向链表的表头

2. 添加到表中间位置

2) 添加至表的中间位置

同单链表添加数据类似，双向链表中间位置添加数据需要经过以下 2 个步骤，如图 3 所示：

1. 新节点先与其直接后继节点建立双层逻辑关系；
2. 新节点的直接前驱节点与之建立双层逻辑关系；

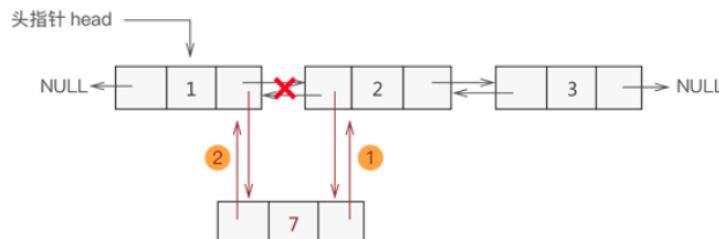


图 3 双向链表中间位置添加数据元素

3. 表尾添加

3) 添加至表尾

与添加到表头是一个道理，实现过程如下（如图 4 所示）：

1. 找到双向链表中最后一个节点；
2. 让新节点与最后一个节点进行双层逻辑关系；

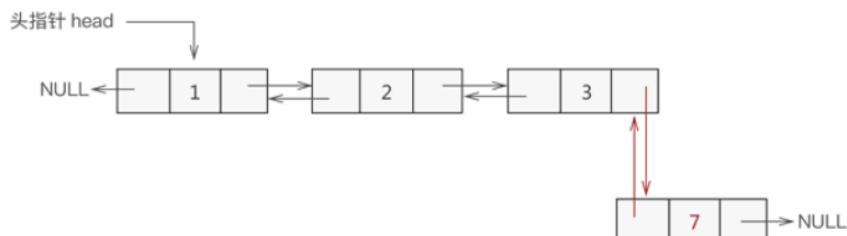


图 4 双向链表尾部添加数据元素

```
//data 为要添加的新数据，add 为添加到链表中的位置
line * insertLine(line * head, int data, int add) {

    //新建数据域为data的结点
    line * temp = (line*)malloc(sizeof(line));
    temp->data = data;
    temp->prior = NULL;
    temp->next = NULL;

    //插入到链表头，要特殊考虑
}
```

```

if (add == 1) {
    temp->next = head;
    head->prior = temp;
    head = temp;
}
else {
    int i = 0;//用于遍历
    Line * body = head;
    //找到要插入位置的前一个结点

    for (i = 1; i < add - 1; i++) {
        body = body->next;
        if (body == NULL) {
            //只要不是表头就肯定有前驱，否则位置出错
            printf("插入位置有误\n");
            break;
        }
    }

    if (body) {
        //判断条件为真，说明插入位置为链表尾
        if (body->next == NULL) {
            body->next = temp;
            temp->prior = body;
        }
        else {
            body->next->prior = temp;
            temp->next = body->next;
            body->next = temp;
            temp->prior = body;
        }
    }
}

return head;
}

```

4. 删除节点

双向链表删除节点

双链表删除结点时，只需遍历链表找到要删除的结点，然后将该节点从表中摘除即可。

例如，从图 1 基础上删除元素 2 的操作过程如图 5 所示：

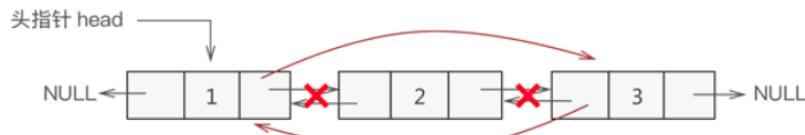


图 5 双链表删除元素操作示意图

```

//删除结点的函数，data为要删除结点的数据域的值
Line * delLine(Line * head, int data) {
    Line * temp = head;
    //遍历链表

```

```

while (temp) {
    //判断当前结点中数据域和data是否相等，若相等，摘除该结点
    if (temp->data == data) {
        temp->prior->next = temp->next;
        temp->next->prior = temp->prior;
        free(temp);
        return head;
    }
    temp = temp->next;
}
printf("链表中无该数据元素\n");
return head;
}

```

4.按值查找

双向链表查找节点

通常，双向链表同单链表一样，都仅有一个头指针。因此，双链表查找指定元素的实现同单链表类似，都是从表头依次遍历表中元素。

```

//head为原双链表，elem表示被查找元素
int selectElem(Dlink * head, int elem) {
    //新建一个指针t，初始化为头指针 head
    Dlink * t = head;
    int i = 1;
    while (t) {
        if (t->data == elem) {
            return i;
        }
        i++; //记录节点
        t = t->next;
    }
    //程序执行至此处，表示查找失败
    return -1;
}

```

5.修改

双向链表更改节点

更改双链表中指定结点数据域的操作是在查找的基础上完成的。实现过程是：通过遍历找到存储有该数据元素的结点，直接更改其数据域即可。

```

//更新函数，其中，add 表示更改结点在双链表中的位置，newElem 为新数据的值
Dlink *amendElem(Dlink * p, int add, int newElem) {
    int i = 0;
    Dlink * temp = p;
    //遍历到被删除结点，位置最多到最后一个节点
    for (i = 1; i < add; i++) {
        temp = temp->next;
        if (temp == NULL) {
            printf("更改位置有误！\n");
            break;
        }
    }
    if (temp) {

```

```
    temp->data = newElem;
}
return p;
}
```

```
//完整代码
#include <stdio.h>
#include <stdlib.h>
typedef struct line {
    struct line * prior;
    int data;
    struct line * next;
}line;

//双链表的创建
line* initLine(line * head);

//双链表插入元素，add表示插入位置
line * insertLine(line * head, int data, int add);

//双链表删除指定元素
line * delLine(line * head, int data);

//双链表中查找指定元素
int selectElem(line * head, int elem);

//双链表中更改指定位置节点中存储的数据，add表示更改位置
line * amendElem(line * p, int add, int newElem);

//输出双链表的实现函数
void display(line * head);

int main() {
    line * head = NULL;
    //创建双链表
    printf("初始链表为: \n");
    head = initLine(head);
    display(head);
    //在表中第 3 的位置插入元素 7
    printf("在表中第 3 的位置插入新元素 7: \n");
    head = insertLine(head, 7, 3);
    display(head);
    //表中删除元素 2
    printf("删除元素 2: \n");
    head = delLine(head, 2);
    display(head);

    printf("元素 3 的位置是: %d\n", selectElem(head, 3));
    //表中第 3 个节点中的数据改为存储 6
    printf("将第 3 个节点存储的数据改为 6: \n");
    head = amendElem(head, 3, 6);
    display(head);
    return 0;
}

line* initLine(line * head) {
    int i = 0;
```

```

line * list = NULL;
head = (line*)malloc(sizeof(line));
head->prior = NULL;
head->next = NULL;
head->data = 1;
list = head;
for (i = 2; i <= 3; i++) {
    line * body = (line*)malloc(sizeof(line));
    body->prior = NULL;
    body->next = NULL;
    body->data = i;

    list->next = body;
    body->prior = list;
    list = list->next;
}
return head;
}

line * insertLine(line * head, int data, int add) {
    //新建数据域为data的结点
    line * temp = (line*)malloc(sizeof(line));
    temp->data = data;
    temp->prior = NULL;
    temp->next = NULL;
    //插入到链表头，要特殊考虑
    if (add == 1) {
        temp->next = head;
        head->prior = temp;
        head = temp;
    }
    else {
        int i = 0;
        line * body = head;
        //找到要插入位置的前一个结点
        for (i = 1; i < add - 1; i++) {
            body = body->next;
            if (body == NULL) {
                printf("插入位置有误\n");
                break;
            }
        }
        if (body) {
            //判断条件为真，说明插入位置为链表尾
            if (body->next == NULL) {
                body->next = temp;
                temp->prior = body;
            }
            else {
                body->next->prior = temp;
                temp->next = body->next;
                body->next = temp;
                temp->prior = body;
            }
        }
    }
    return head;
}

```

```

}

line * delLine(line * head, int data) {
    line * temp = head;
    //遍历链表
    while (temp) {
        //判断当前结点中数据域和data是否相等, 若相等, 摘除该结点
        if (temp->data == data) {
            temp->prior->next = temp->next;
            temp->next->prior = temp->prior;
            free(temp);
            return head;
        }
        temp = temp->next;
    }
    printf("链表中无该数据元素\n");
    return head;
}

//head为原双链表, elem表示被查找元素
int selectElem(line * head, int elem) {
    //新建一个指针t, 初始化为头指针 head
    line * t = head;
    int i = 1;
    while (t) {
        if (t->data == elem) {
            return i;
        }
        i++;
        t = t->next;
    }
    //程序执行至此处, 表示查找失败
    return -1;
}

//更新函数, 其中, add 表示更改结点在双链表中的位置, newElem 为新数据的值
line * amendElem(line * p, int add, int newElem) {
    int i = 0;
    line * temp = p;
    //遍历到被删除结点
    for (i = 1; i < add; i++) {
        temp = temp->next;
        if (temp == NULL) {
            printf("更改位置有误! \n");
            break;
        }
    }
    if (temp) {
        temp->data = newElem;
    }
    return p;
}

//输出链表的功能函数

```

```
void display(line * head) {  
    line * temp = head;  
    while (temp) {  
        if (temp->next == NULL) {  
            printf("%d\n", temp->data);  
        }  
        else {  
            printf("%d->", temp->data);  
        }  
        temp = temp->next;  
    }  
}
```

2.带头结点

5.循环链表

1.双循环链表

2.循环链表

6. 栈

同顺序表和链表一样，栈也是用来存储逻辑关系为“一对多”数据的线性存储结构，如图 1 所示。



图 1 栈存储结构示意图

从图 1 我们看到，栈存储结构与之前所学的线性存储结构有所差异，这缘于栈对数据“存”和“取”的过程有特殊的要求：

1. 栈只能从表的一端存取数据，另一端是封闭的，如图 1 所示；
2. 在栈中，无论是存数据还是取数据，都必须遵循“先进后出”的原则，即最先进栈的元素最后出栈。拿图 1 的栈来说，从图中数据的存储状态可判断出，元素 1 是最先进的栈。因此，当需要从栈中取出元素 1 时，根据“先进后出”的原则，需提前将元素 3 和元素 2 从栈中取出，然后才能成功取出元素 1。

因此，我们可以给栈下一个定义，即栈是一种只能从表的一端存取数据且遵循“先进后出”原则的线性存储结构。

通常，栈的开口端被称为**栈顶**；相应地，封口端被称为**栈底**。因此，栈顶元素指的就是距离栈顶最近的元素，拿图 2 来说，栈顶元素为元素 4；同理，栈底元素指的是位于栈最底部的元素，图 2 中的栈底元素为元素 1。

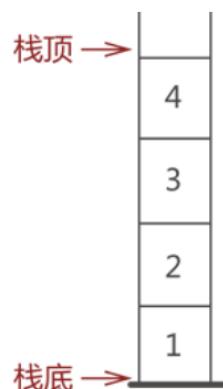


图 2 栈顶和栈底

InitStack(&S) 初始化操作

操作结果：构造一个空栈 S。

DestroyStack(&S) 销毁栈操作

初始条件：栈 S 已存在。

操作结果：栈 S 被销毁。

StackEmpty(S) 判定 S 是否为空栈

初始条件：栈 S 已存在。

操作结果：若栈 S 为空栈，则返回 TRUE，
否则 FALSE。

StackLength(S) 求栈的长度

初始条件：栈 S 已存在。

操作结果：返回 S 的元素个数，即栈的长度。

GetTop(S, &e) 取栈顶元素

初始条件：栈 S 已存在且非空。

操作结果：用 e 返回 S 的栈顶元素。

ClearStack(&S) 栈置空操作

初始条件：栈 S 已存在。

操作结果：将 S 清为空栈。

Push(&S, e) 入栈操作

初始条件：栈 S 已存在。

操作结果：插入元素 e 为新的栈顶元素。

Pop(&S, &e) 出栈操作

初始条件：栈 S 已存在且非空。

操作结果：删除 S 的栈顶元素 a_n ，并用 e 返回其值。

进栈和出栈

基于栈结构的特点，在实际应用中，通常只会对栈执行以下两种操作：

- 向栈中添加元素，此过程被称为“进栈”（入栈或压栈）；
- 从栈中提取出指定元素，此过程被称为“出栈”（或弹栈）；

栈的具体实现

栈是一种“特殊”的线性存储结构，因此栈的具体实现有以下两种方式：

1. **顺序栈**：采用**顺序存储结构**可以模拟栈存储数据的特点，从而实现栈存储结构；
2. **链栈**：采用**链式存储结构**实现栈结构；

两种实现方式的区别，仅限于数据元素在实际物理空间上存放的相对位置，顺序栈底层采用的是数组，链栈底层采用的是链表。有关顺序栈和链栈的具体实现会在后续章节中作详细讲解。

栈的应用

基于栈结构对数据存取采用“先进后出”原则的特点，它可以用于实现很多功能。

例如，我们经常使用浏览器在各种网站上查找信息。假设先浏览了页面 A，然后关闭了页面 A 跳转到页面 B，随后又关闭页面 B 跳转到了页面 C。而此时，我们如果想重新回到页面 A，有两个选择：

- 重新搜索找到页面 A；
- 使用浏览器的“回退”功能。浏览器会先回退到页面 B，而后再回退到页面 A。

浏览器“回退”功能的实现，底层使用的就是栈存储结构。当你关闭页面 A 时，浏览器会将页面 A 入栈；同样，当你关闭页面 B 时，浏览器也会将 B 入栈。因此，当你执行回退操作时，才会首先看到的是页面 B，然后是页面 A，这是栈中数据依次出栈的效果。

不仅如此，栈存储结构还可以帮我们检测代码中的括号匹配问题。多数编程语言都会用到括号（小括号、中括号和大括号），括号的错误使用（通常是丢右括号）会导致程序编译错误，而很多开发工具中都有检测代码是否有编辑错误的功能，其中就包含检测代码中的括号匹配问题，此功能的底层实现使用的就是栈结构。

同时，栈结构还可以实现数值的进制转换功能。例如，编写程序实现从十进制数自动转换成二进制数，就可以使用栈存储结构来实现。

以上也仅是栈应用领域的冰山一角，这里不再过多举例。在后续章节的学习中，我们会大量使用到栈结构。接下来，我们学习如何实现顺序栈和链栈，以及对栈中元素进行入栈和出栈的操作。

1. 顺序栈

3.3.2 顺序栈的表示和实现



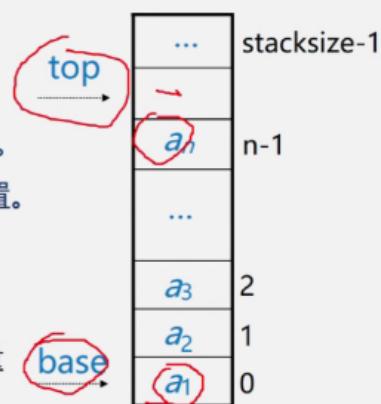
存储方式：同一般线性表的顺序存储结构完全相同，

利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。栈底一般在低地址端。

- 附设**top**指针，指示栈顶元素在顺序栈中的位置。
- 另设**base**指针，指示栈底元素在顺序栈中的位置。

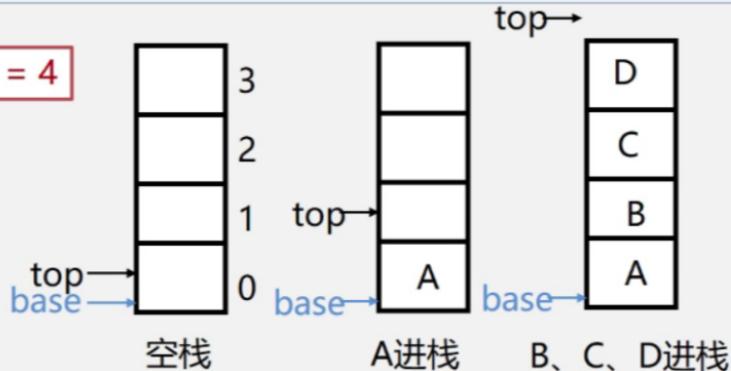
但是，为了方便操作，通常**top** 指示真正的栈顶元素之上的下标地址

- 另外，用**stacksize**表示栈可使用的最大容量



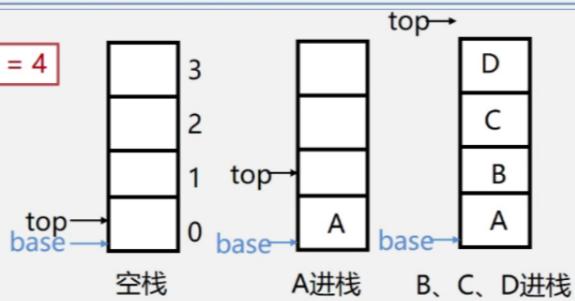
3.3.2 顺序栈的表示和实现

stacksize = 4



3.3.2 顺序栈的表示和实现

stacksize = 4



空栈: base == top 是栈空标志

栈满 top-base==stacksize

栈满时的处理方法:

1. 报错, 返回操作系统。
2. 分配更大的空间, 作为栈的存储空间, 将原栈的内容移入新栈。

3.3.2 顺序栈的表示和实现

使用数组作为顺序栈存储方式的特点:

简单、方便、但易产生溢出 (数组大小固定)

- 上溢(overflow): 栈已经满, 又要压入元素
- 下溢(underflow): 栈已经空, 还要弹出元素

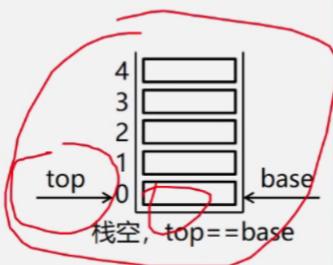


注: 上溢是一种错误, 使问题的处理无法进行; 而下溢一般认为是一种结束条件, 即问题处理结束。

3.3.2 顺序栈的表示和实现



【算法3.1】顺序栈的初始化

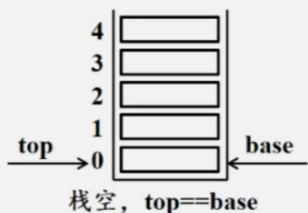


```
Status InitStack(SqStack &S){ //构造一个空栈
    S.base = new SElemType[MAXSIZE]; //或
    S.base = (SElemType*)malloc(MAXSIZE*sizeof(SElemType));
    if (!S.base) exit (OVERFLOW); //存储分配失败
    S.top = S.base; //栈顶指针等于栈底指针
    S.stacksize = MAXSIZE;
    return OK;
}
```

3.3.2 顺序栈的表示和实现



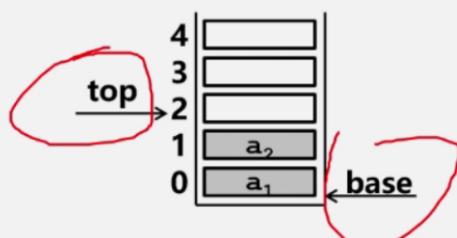
【算法补充】顺序栈判断栈是否为空



```
Status StackEmpty(SqStack S){
    //若栈为空, 返回TRUE; 否则返回FALSE
    if (S.top == S.base)
        return TRUE;
    else
        return FALSE;
}
```

3.3.2 顺序栈的表示和实现

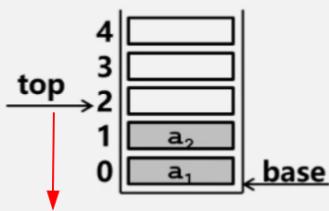
【算法补充】求顺序栈长度



```
int StackLength( SqStack S )
{
    return S.top - S.base;
}
```

3.3.2 顺序栈的表示和实现

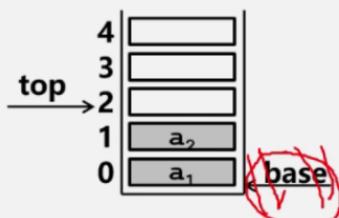
【算法补充】清空顺序栈



```
Status ClearStack( SqStack S ) {  
    if( S.base ) S.top = S.base;  
    return OK;  
}
```

3.3.2 顺序栈的表示和实现

【算法补充】销毁顺序栈

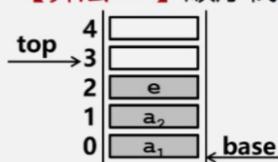


```
Status DestroyStack( SqStack &S ) {  
    if( S.base ) {  
        delete S.base ;  
        S.stacksize = 0;  
        S.base = S.top = NULL;  
    }  
    return OK;  
}
```

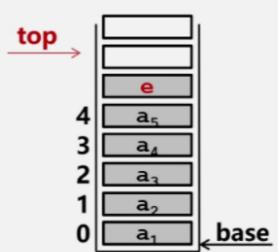
3.3.2 顺序栈的表示和实现



【算法3.2】顺序栈的入栈



- (1)判断是否栈满，若满则出错（上溢）
- (2)元素e压入栈顶
- (3)栈顶指针加1



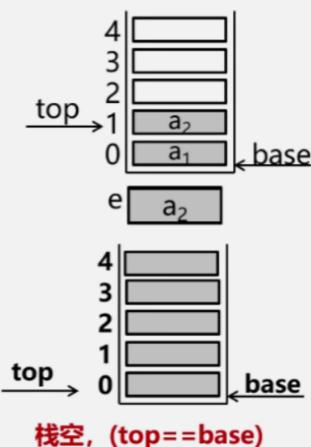
```
Status Push( SqStack &S, SElemType e ) {  
    if( S.top - S.base == S.stacksize ) // 栈满  
        return ERROR;  
    *S.top++ = e;  
    return OK;  
}
```

*S.top=e;
S.top++;

3.3.2 顺序栈的表示和实现



【算法3.3】顺序栈的出栈



(1)判断是否栈空，若空则出错（下溢）
(2)获取栈顶元素e
(3)栈顶指针减1

```
Status Pop(SqStack &S, SElemType &e){  
    //若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK；  
    //否则返回ERROR  
    if(S.top == S.base) //等价于 if(StackEmpty(S))  
        return ERROR;  
    e = *--S.top; //从栈顶移出一个元素  
    return OK;  
}
```

```
#include<stdio.h>  
#include<stdlib.h>  
#define MAXSIZE 100  
  
#define TRUE 1  
#define FALSE 0  
#define OK 1  
#define ERROR 0  
#define INFEASIBLE -1  
#define OVERFLOW -2  
  
typedef int Status;  
  
typedef struct{  
    int *base; //栈底指针  
    int *top; //栈顶指针  
    int stacksize; //可用最大容量  
} SqStack;  
  
void menu(){  
    printf("-----栈的基本操作-----\n\n");  
    printf("-----1. 初始化栈-----\n\n");  
    printf("-----2. 判断栈空-----\n\n");  
    printf("-----3. 进栈操作-----\n\n");  
    printf("-----4. 出栈操作-----\n\n");  
    printf("-----5. 清空栈-----\n\n");  
    printf("-----\n\n");  
    printf("请选择你的操作 (1-5) : ");  
}  
  
// 初始化  
Status InitStack(SqStack &s){  
    s.base = (int *)malloc(MAXSIZE*sizeof(int));  
    if(!s.base) exit(OVERFLOW);  
    s.top = s.base; //设置为空  
    s.stacksize = MAXSIZE;  
    printf("初始化成功\n");  
}
```

```
    return OK;
}

Status StackEmpty(SqStack s){
    if(s.top == s.base){
        printf("NULL\n");
        return TRUE;
    }
    else{
        printf("Not NULL\n");
        return FALSE;
    }
}

int StackLength(SqStack s){
    printf("长度为:%d\n",s.top - s.base);
    return s.top - s.base;
}

Status ClearStack(SqStack s){
    if(s.base){
        s.top = s.base;
        printf("清空成功\n");
    }
    return OK;
}

Status DestroyStack(SqStack &s){
    if(s.base){
        // s.base 存在则销毁
        delete s.base;
        s.stacksize = 0;
        s.base = s.top = NULL;
        printf("销毁成功\n");
    }
    return OK;
}

Status Push(SqStack &s,int e){
    if(s.top - s.base == s.stacksize){//栈满
        printf("栈满\n");
        return ERROR;
    }
    *s.top = e;
    s.top++;
    return OK;
}

Status Pop(SqStack &s,int &e){
    if(s.top == s.base){
        printf("栈空\n");
        return ERROR;
    }
}
```

```

    }
    --s.top;
    e = *s.top;
    printf("出栈元素是%d\n",e);
    return OK;
}

int main(){
    SqStack s;
    menu();
    int op = -1;
    scanf("%d",&op);
    while(1){
        system("clear");
        menu();
        switch(Op){
            case 1:
                printf("\n");
                InitStack(s);
                break;
            case 2:
                printf("\n");
                StackEmpty(s);
                break;
            case 3:
                printf("\n");
                int PushNum;
                printf("\n");
                printf("Please input the Push number:");
                scanf("%d",&PushNum);
                Push(s,PushNum);
                break;
            case 4:
                printf("\n");
                int PopNum;
                // printf("Please input the Pop number:");
                // scanf("%d",&PopNum);
                Pop(s,PopNum);
                break;
            case 5:
                printf("\n");
                ClearStack(s);
                break;
            default:
                printf("\n");
                printf("ERROR\n");
        }
        scanf("%d",&op);
    }
    return 0;
}

```

2.链栈

链栈，即用链表实现栈存储结构。

链栈的实现思路同顺序栈类似，顺序栈是将数[顺序表](#)（数组）的一端作为栈底，另一端为栈顶；链栈也如此，通常我们将链表的头部作为栈顶，尾部作为栈底，如图 1 所示：

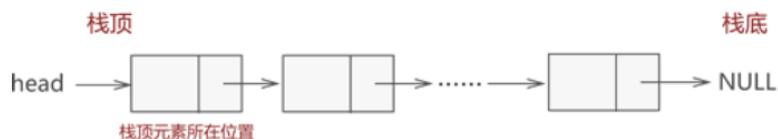


图 1 链栈示意图

将链表头部作为栈顶的一端，可以避免在实现数据“入栈”和“出栈”操作时做大量遍历链表的耗时操作。

链表的头部作为栈顶，意味着：

- 在实现数据“入栈”操作时，需要将数据从链表的头部插入；
- 在实现数据“出栈”操作时，需要删除链表头部的首元节点；

因此，链栈实际上就是一个只能采用头插法插入或删除数据的链表。

链栈元素入栈

例如，将元素 1、2、3、4 依次入栈，等价于将各元素采用头插法依次添加到链表中，每个数据元素的添加过程如图 2 所示：

- 刚开始： head → NULL
- 添加元素 1： head → 1 → NULL
- 添加元素 2： head → 2 → 1 → NULL
- 添加元素 3： head → 3 → 2 → 1 → NULL
- 添加元素 4： head → 4 → 3 → 2 → 1 → NULL

图 2 链栈元素依次入栈过程示意图

链栈元素出栈

例如，图 2e) 所示的链栈中，若要将元素 3 出栈，根据“先进后出”的原则，要先将元素 4 出栈，也就是从链表中摘除，然后元素 3 才能出栈，整个操作过程如图 3 所示：

- 初始链栈： head → 4 → 3 → 2 → 1 → NULL
- 元素 4 出栈： head → 3 → 2 → 1 → NULL
- 元素 3 出栈： head → 2 → ~~3~~ → NULL

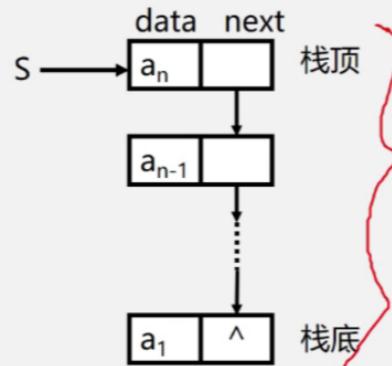
图 3 链栈元素出栈示意图

3.3.3 链栈的表示和实现

链栈的表示

- 链栈是运算受限的单链表，只能在链表头部进行操作

```
typedef struct StackNode{  
    SElemType data;  
    struct StackNode *next;  
} StackNode, *LinkStack;  
LinkStack S;
```



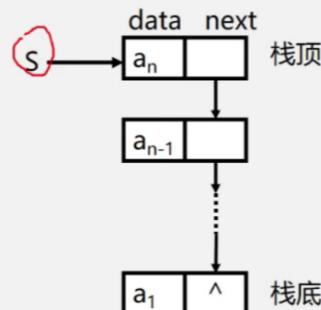
注意：链栈中指针的方向

3.3.3 链栈的表示和实现

链栈的表示

- 链栈是运算受限的单链表，只能在链表头部进行操作

```
typedef struct StackNode{  
    SElemType data;  
    struct StackNode *next;  
} StackNode, *LinkStack;  
LinkStack S;
```



- 链表的头指针就是栈顶
- 不需要头结点
- 基本不存在栈满的情况
- 空栈相当于头指针指向空
- 插入和删除仅在栈顶处执行

注意：链栈中指针的方向

3.3.3 链栈的表示和实现

(算法3.5) 链栈的初始化

```
void InitStack(LinkStack &S ) {  
    //构造一个空栈，栈顶指针置为空  
    S=NULL;  
    return OK;  
}
```

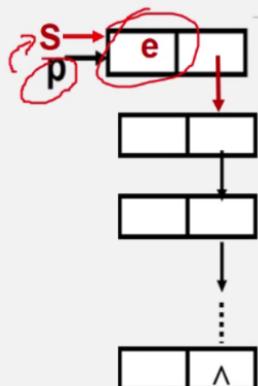
3.3.3 链栈的表示和实现

【补充算法】判断链栈是否为空

```
Status StackEmpty(LinkStack S)
    if (S==NULL) return TRUE;
    else return FALSE;
}
```

3.3.3 链栈的表示和实现

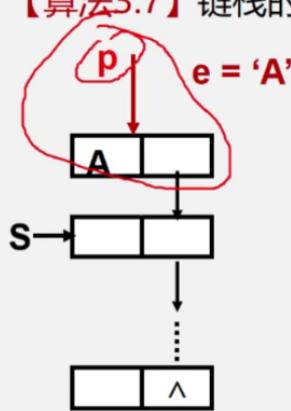
【算法3.6】链栈的入栈



```
Status Push(LinkStack &S , SElemType e){
    p=new StackNode; //生成新结点p
    p->data=e; //将新结点数据域置为e
    p->next=S; //将新结点插入栈顶
    S=p; //修改栈顶指针
    return OK;
}
```

3.3.3 链栈的表示和实现

【算法3.7】链栈的出栈



```
Status Pop (LinkStack &S,SElemType &e){
    if (S==NULL) return ERROR;
    e = S-> data;
    p = S; 保存S
    S = S-> next;
    delete p;
    return OK;
}
```

```
#include<stdio.h>
#include<stdlib.h>
#define MAXSIZE 100

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2

typedef int Status;
typedef struct StackNode{
    int data;
    struct StackNode *next;
}StackNode,*LinkStack;

void menu(){
    printf("-----栈的基本操作-----\n\n");
    printf("-----1.初始化栈-----\n\n");
    printf("-----2.判断栈空-----\n\n");
    printf("-----3.进栈操作-----\n\n");
    printf("-----4.出栈操作-----\n\n");
    printf("-----5.读栈顶数-----\n\n");
    printf("-----\n\n");
    printf("请选择你的操作 (1-5) : ");
}

int InitStack(LinkStack &s){
    s = NULL;
    printf("初始化成功\n");
    return OK;
}

Status StackEmpty(LinkStack s){
    if(s == NULL){
        printf("栈空\n");
        return true;
    }
    else{
        printf("非空");
        return false;
    }
}

Status Push(LinkStack &s,int e){
    StackNode *p = (StackNode *)malloc(sizeof(StackNode));
    p->data = e;
    p->next = s;
    // 头插
    s = p;
    printf("插入成功\n");
    return OK;
}
```

```
status Pop(LinkStack &s, int &e){  
    StackNode *p;  
    if(s == NULL){  
        printf("栈空\n");  
        return ERROR;  
    }  
    e = s->data;  
    p = s;//保存  
    printf("出栈元素是%d", e);  
    s = s->next;  
    delete p;  
    return OK;  
}  
  
int GetTop(LinkStack s){  
    if(s != NULL){  
        printf("栈顶元素是%d\n", s->data);  
        return s->data;  
    }  
}  
  
int main(){  
    StackNode *S;  
    menu();  
    int op = -1;  
    scanf("%d", &op);  
    while(1){  
        system("clear");  
        menu();  
        switch(op){  
            case 1:  
                printf("\n");  
                InitStack(S);  
                break;  
            case 2:  
                printf("\n");  
                StackEmpty(S);  
                break;  
            case 3:  
                printf("\n");  
                int PushNum;  
                printf("\n");  
                printf("Please input the Push number:");  
                scanf("%d", &PushNum);  
                Push(S, PushNum);  
                break;  
            case 4:  
                printf("\n");  
                int PopNum;  
                // printf("Please input the Pop number:");  
                // scanf("%d", &PopNum);  
                Pop(S, PopNum);  
                break;  
            case 5:  
                printf("\n");  
                GetTop(S);  
                break;  
        }  
    }  
}
```

```

    default:
        printf("\n");
        printf("ERROR\n");
    }
    scanf("%d",&op);
}
return 0;
}

```

7. 队列

3.5 队列的表示和操作的实现

▪ 相关术语

- 队列 (Queue) 是仅在表尾进行插入操作，在表头进行删除操作的线性表。
- 表尾即 a_n 端，称为队尾；表头即 a_1 端，称为队头。
- 它是一种先进先出 (FIFO) 的线性表。

例如：队列 $Q = (a_1, a_2, a_3, \dots, a_{n-1}, a_n)$



插入元素称为入队；删除元素称为出队。

队列的存储结构为链队或顺序队 (常用循环顺序队)

3.5 队列的表示和操作的实现

▪ 队列的常见应用

- 脱机打印输出：按申请的先后顺序依次输出。
- 多用户系统中，多个用户排成队，分时地循环使用CPU和主存
- 按用户的优先级排成多个队，每个优先级一个队列
- 实时控制系统中，信号按接收的先后顺序依次处理
- 网络电文传输，按到达的时间先后顺序依次进行



3.5.1 队列的抽象数据类型定义



ADT Queue{

数据对象: $D = \{a_i | a_i \in \text{ElemSet}, i=1, 2, \dots, n, n \geq 0\}$

数据关系: $R = \{<a_{i-1}, a_i> | a_{i-1}, a_i \in D, i=2, \dots, n\}$ 约定其中 a_1 端为队列头, a_n 端为队列尾。

基本操作:

InitQueue(&Q) 操作结果: 构造空队列Q

DestroyQueue(&Q) 条件: 队列Q已存在; 操作结果: 队列Q被销毁

ClearQueue(&Q) 条件: 队列Q已存在; 操作结果: 将Q清空

QueueLength(Q) 条件: 队列Q已存在 操作结果: 返回Q的元素个数, 即队长

GetHead(Q, &e) 条件: Q为非空队列 操作结果: 用e返回Q的队头元素

EnQueue(&Q, e) 条件: 队列Q已存在 操作结果: 插入元素e为Q的队尾元素

DeQueue(&Q, &e) 条件: Q为非空队列 操作结果: 删除Q的队头元素, 用e返回值

..... 还有将队列置空、遍历队列等操作.....

} ADT Queue

1.顺序队列

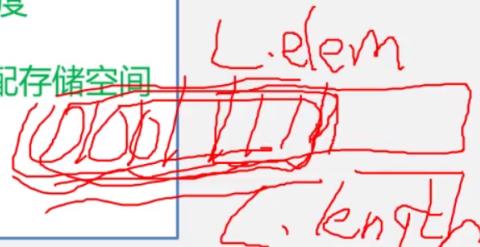
3.5.2 队列的顺序表示和实现



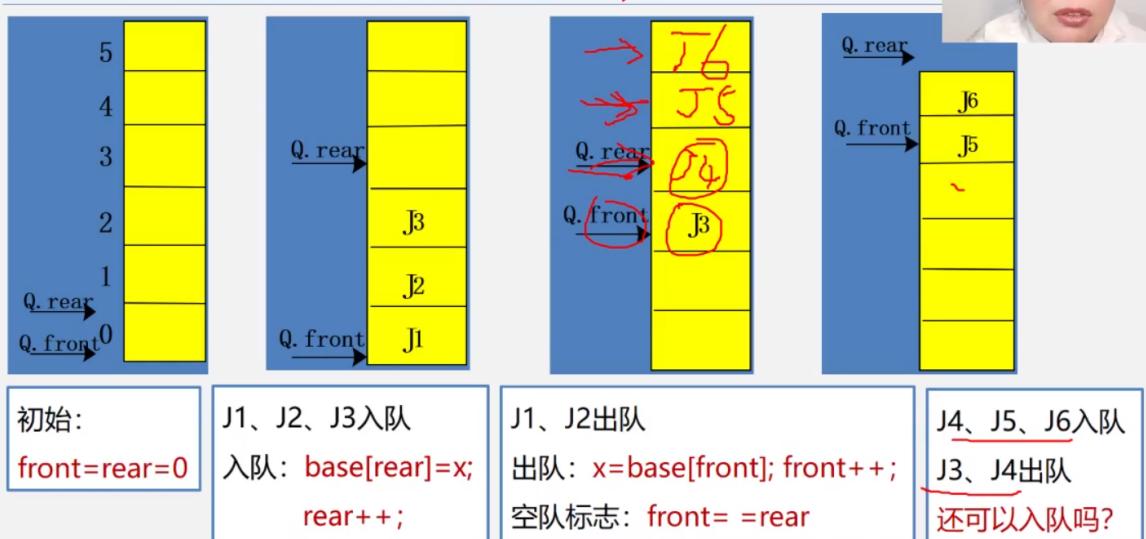
- 队列的物理存储可以用顺序存储结构, 也可用链式存储结构。相应地, 队列的存储方式也分为两种, 即顺序队列和链式队列。

- 队列的顺序表示——用一维数组base[MAXQSIZE]

```
#define MAXQSIZE 100 //最大队列长度
TypeDef struct {
    QElemType *base; //初始化的动态分配存储空间
    int front; //头指针
    int rear; //尾指针
} SqQueue;
```



3.5.2 队列的顺序表示和实现



3.5.2 队列的顺序表示和实现



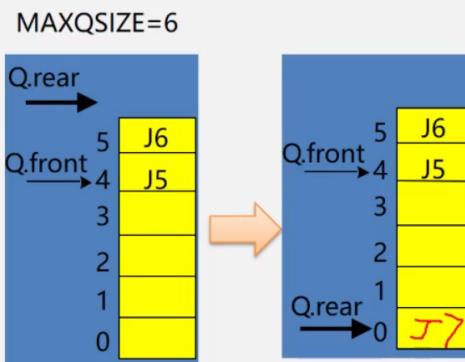
3.5.2 队列的顺序表示和实现

❖ 解决假上溢的方法

1、将队中元素依次向队头方向移动。

缺点：浪费时间。每移动一次，队中元素都要移动。

2、将队空间设想成一个循环的表，即分配给队列的m个存储单元可以循环使用，当rear为maxqsize时，若向量的开始端空着，又可从头使用空着的空间。当front为maxqsize时，也是一样。



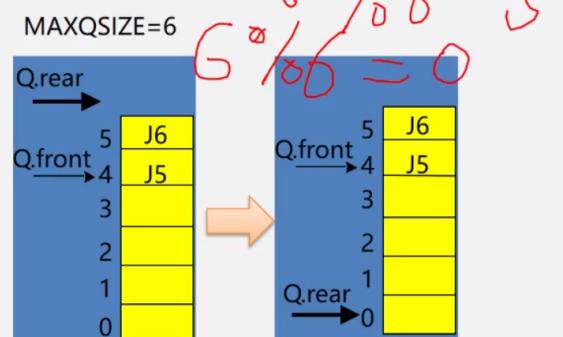
3.5.2 队列的顺序表示和实现

❖ 解决假上溢的方法

1、将队中元素依次向队头方向移动。

缺点：浪费时间。每移动一次，队中元素都要移动。

2、将队空间设想成一个循环的表，即分配给队列的m个存储单元可以循环使用，当rear为maxqsize时，若向量的开始端空着，又可从头使用空着的空间。当front为maxqsize时，也是一样。



3.5.2 队列的顺序表示和实现



❖ 解决假上溢的方法——引入循环队列

base[0]接在base[MAXQSIZE - 1]之后，若rear+1==M，则令rear=0;

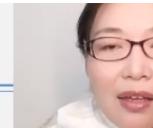
实现方法：利用模(mod, C语言中：%) 运算。

插入元素： Q.base[Q.rear]=x;

Q.rear=(Q.rear+1)% MAXQSIZE;

$$(5+1) \% 6 = 0$$

3.5.2 队列的顺序表示和实现



❖ 解决假上溢的方法——引入循环队列

base[0]接在base[MAXQSIZE - 1]之后，若rear+1==M，则令rear=0;

实现方法：利用模(mod, C语言中：%) 运算。

插入元素： Q.base[Q.rear]=x;

Q.rear=(Q.rear+1)% MAXQSIZE;

删除元素： x=Q.base[s.front]

Q.front=(Q.front+1)% MAXQSIZE

3.5.2 队列的顺序表示和实现



❖ 解决假上溢的方法——引入循环队列

base[0]接在base[MAXQSIZE - 1]之后，若rear+1==M，则令rear=0;

实现方法：利用模(mod, C语言中：%) 运算。

插入元素： Q.base[Q.rear]=x;

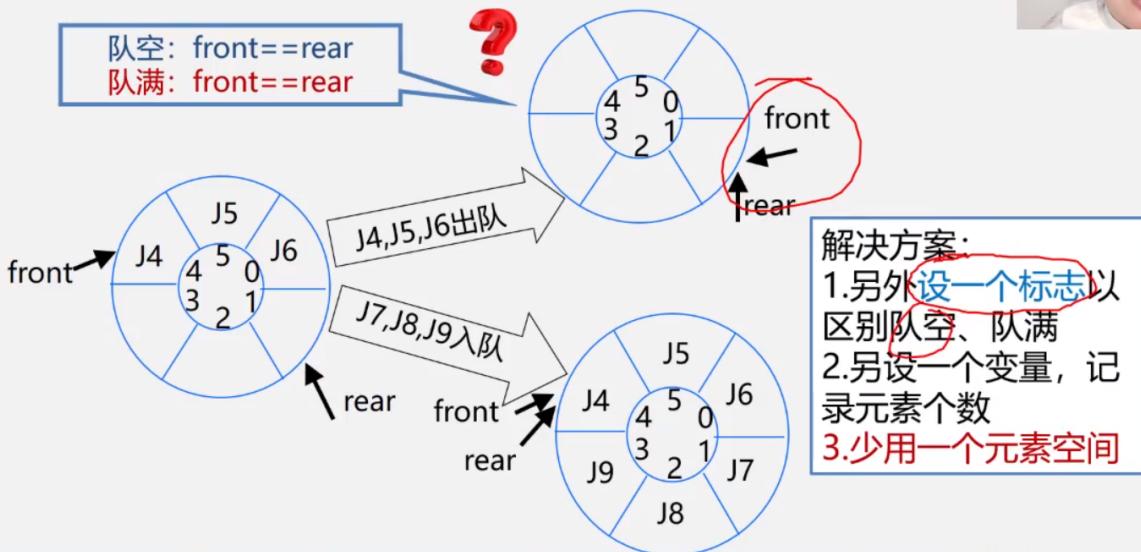
Q.rear=(Q.rear+1)% MAXQSIZE;

删除元素： x=Q.base[s.front]

Q.front=(Q.front+1)% MAXQSIZE

循环队列：循环使用为队列分配的存储空间。

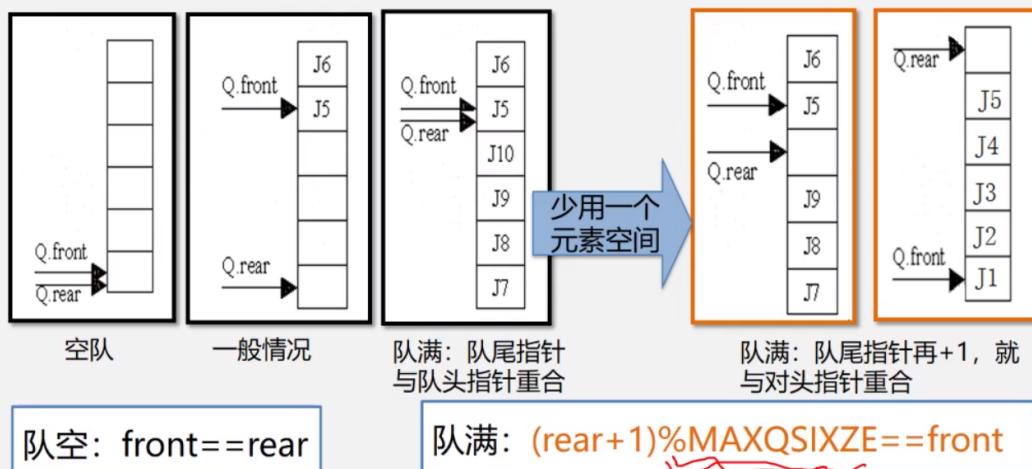
3.5.2 队列的顺序表示和实现



3.5.2 队列的顺序表示和实现



- 循环队列解决队满时判断方法——少用一个元素空间:



3.5.2 队列的顺序表示和实现



- ❖ 循环队列的操作——队列的初始化 (算法3.11)

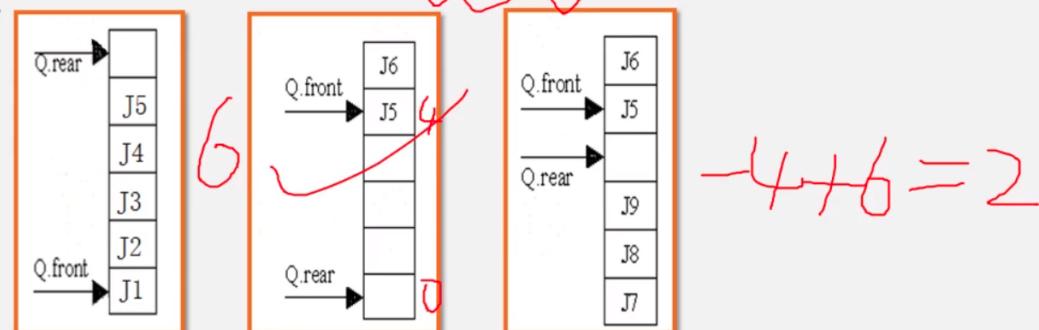
```
Status InitQueue (SqQueue &Q){  
    Q.base = new QELEMType[MAXQSIXZE]; //分配数组空间  
    //Q.base = (QELEMType*)  
    malloc(MAXQSIXZE*sizeof(QELEMType));  
    if (!Q.base) exit(OVERFLOW); //存储分配失败  
    Q.front=Q.rear=0; //头指针尾指针置为0, 队列为空  
    return OK;  
}
```

3.5.2 队列的顺序表示和实现



❖ 循环队列的操作——求队列的长度 (算法3.12)

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

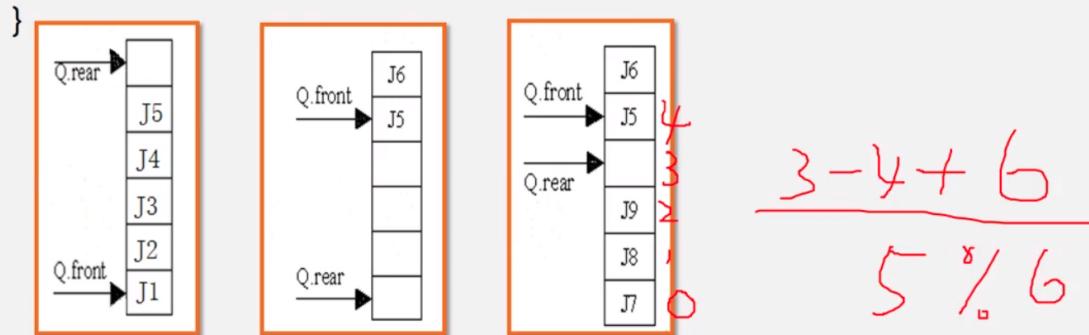


3.5.2 队列的顺序表示和实现



❖ 循环队列的操作——求队列的长度 (算法3.12)

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



3.5.2 队列的顺序表示和实现



❖ 解决假上溢的方法——引入循环队列

base[0]接在base[MAXQSIZE - 1]之后，若rear+1==M，则令rear=0;

实现方法：利用模(mod, C语言中：%) 运算。

插入元素： $Q.base[Q.rear]=x;$

$Q.rear=(Q.rear+1)\% MAXQSIZE;$

3.5.2 队列的顺序表示和实现

❖ 循环队列的操作——循环队列入队 (算法3.13)

```
Status EnQueue(SqQueue &Q, QElemType e){  
    if((Q.rear+1)%MAXQSIZE==Q.front) return ERROR; //队满  
    Q.base[Q.rear]=e; //新元素加入队尾  
    Q.rear=(Q.rear+1)%MAXQSIZE; //队尾指针+1  
    return OK;  
}
```

3.5.2 队列的顺序表示和实现

❖ 解决假上溢的方法——引入循环队列

base[0]接在base[MAXQSIZE - 1]之后，若rear+1==M，则令rear=0；
实现方法：利用模(mod, C语言中：%) 运算。

插入元素： Q.base[Q.rear]=x;
 Q.rear=(Q.rear+1)% MAXQSIZE;
删除元素： x=Q.base[s.front]
 Q.front=(Q.front+1)% MAXQSIZE

3.5.2 队列的顺序表示和实现

❖ 循环队列的操作——循环队列出队 (算法3.14)

```
Status DeQueue (SqQueue &Q,QElemType &e){  
    if(Q.front==Q.rear) return ERROR; //队空  
    e=Q.base[Q.front]; //保存队头元素  
    Q.front=(Q.front+1)%MAXQSIZE; //队头指针+1  
    return OK;  
}
```

3.5.2 队列的顺序表示和实现

❖ 循环队列的操作——取队头元素 (算法3.15)

```
SElemType GetHead(SqQuere Q){  
    if(Q.front!=Q.rear) //队列不为空  
        return Q.base[Q.front]; //返回队头指针元素  
    的值, 队头指针不变
```

```
#include<stdio.h>  
#include<stdlib.h>  
#define MAXSIZE 7  
  
#define TRUE 1  
#define FALSE 0  
#define OK 1  
#define ERROR 0  
#define INFEASIBLE -1  
#define OVERFLOW -2  
  
typedef int Status;  
  
typedef struct{  
    int *base;//栈底指针  
    int front;//头指针  
    int rear;//尾指针  
}SqQueue;  
  
void menu(){  
    printf("-----顺序队列的基本操作-----\n\n");  
    printf("-----1.初始化队列-----\n\n");  
    printf("-----2.判断队列空-----\n\n");  
    printf("-----3.入队 操作-----\n\n");  
    printf("-----4.出队 操作-----\n\n");  
    printf("-----\n\n");  
    printf("请选择你的操作 (1-4) : ");  
}  
  
// 初始化  
Status InitQueue(SqQueue &q){  
    //首元素的地址就是一个指针  
    q.base = (int *)malloc(MAXSIZE*sizeof(int));  
    if(!q.base)exit(OVERFLOW);  
    q.front = q.rear = 0;//头尾指针设置为0, 队列空  
    printf("初始化成功\n");  
    return OK;
```

```
}
```

```
// 求长度
int QueueLength(SqQueue q){
    printf("线性表的长度尾%d\n", (q.rear - q.front + MAXSIZE)%MAXSIZE);
    return((q.rear - q.front + MAXSIZE) % MAXSIZE);
}
```

```
//判断队列是否为空
bool Sq_QueueEmpty(SqQueue Q){
    if(Q.rear == Q.front){
        return true;
    }
    else{
        printf("队列不为空\n");
        return false;
    }
}
```

```
// 入队
Status EnQueue(SqQueue &q, int e){
    // 少用一个元素形成的解决方法
    if((q.rear + 1) % MAXSIZE == q.front){
        printf("栈满、上溢\n");
        return ERROR;
    }
    q.base[q.rear] = e;//新元素插入到队尾
    printf("入队元素是%d\n", q.base[q.rear]);
    q.rear = (q.rear + 1)%MAXSIZE;//队尾指针加1
    return OK;
}
```

```
// 出队
Status DeQueue(SqQueue &q){
    if(q.front == q.rear){
        printf("队空、下溢\n");
        return ERROR;
    }
    printf("出队元素是: %d\n", q.base[q.front]);
    q.front = (q.front + 1) % MAXSIZE;//队头+1
    return OK;
}
```

```
Status GetHead(SqQueue q){
    if(q.front != q.rear){//队列不为空
        printf("队头元素是: %d\n", q.base[q.front]);
        return q.base[q.front];
    }
    return OK;
}
```

```
int main(){
    SqQueue Q;
    int e = 0;
```

```

//初始化
InitQueue(Q);
//判断是否为空
if(Sq_QueueEmpty(Q)){
    printf("队列为空\n");
}
//入队
for(int i = 0; i < 8; i++){
    EnQueue(Q, i + 1);
}
printf("\n");
//出队
for(int i = 0; i < 8; i++){
    DeQueue(Q);
    //printf("出队元素是: %d\n\n", e);
}
if(!EnQueue(Q, 100)){
    printf("上溢\n");
}
GetHead(Q);
//if(Sq_QueueEmpty(&Q)){
//    printf("队列为空\n");
//}
printf("\n");
system("pause");
return 0;
}

```

2. 链队

3.5.3 链队——队列的链式表示和实现

❖ 若用户无法估计所用队列的长度，则宜采用链队列

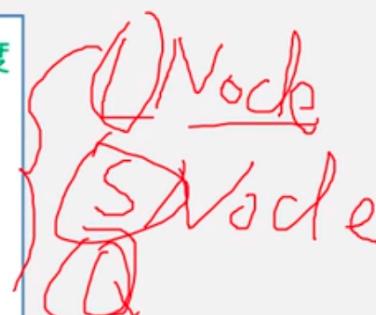


❖ 链队列的类型定义

```

#define MAXQSIZE 100 //最大队列长度
typedef struct Qnode {
    QElemType data;
    struct Qnode *next;
}QNode, *QuenePtr;

```



3.5.3 链队——队列的链式表示和实现



❖ 若用户无法估计所用队列的长度，则宜采用链队列

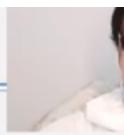


❖ 链队列的类型定义

```
#define MAXQSIZE 100 //最大队列长度  
typedef struct Qnode {  
    QElemType data;  
    struct Qnode *next;  
}QNode, *QueuePtr;
```

```
typedef struct {  
    QueuePtr front; // 队头指针  
    QueuePtr rear; // 队尾指针  
}LinkQueue;
```

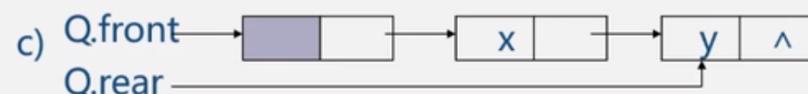
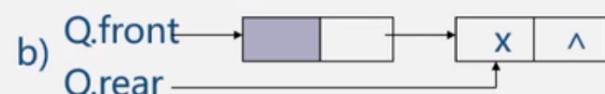
3.5.3 链队——队列的链式表示和实现



❖ 链队列运算指针变化状况



- a) 空队列
- b) 元素x入队列
- c) y入队列
- d) x出队列



3.5.3 链队——队列的链式表示和实现

我以为我电脑响了。。。

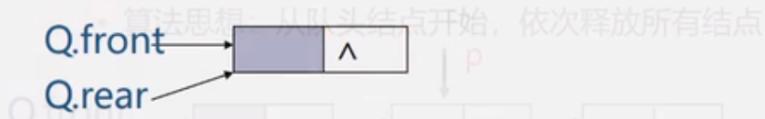
没错你电脑响了

我尼玛，吓死老子了

我是手机哦！咋没有人骗我
我电脑响了

??? Status

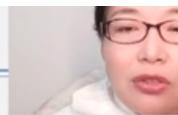
❖ 链队列的操作——**链队列初始化** (算法3.16)



```
Status InitQueue (LinkQueue &Q){
```

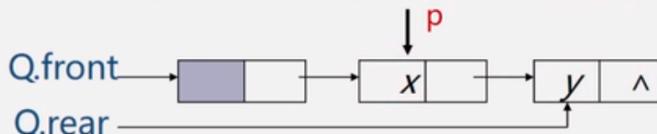
```
    Q.front=Q.rear=(QueuePtr) malloc(sizeof(QNode));  
    if(!Q.front) exit(OVERFLOW);  
    Q.front->next=NULL;  
    return OK;  
}
```

3.5.3 链队——队列的链式表示和实现



❖ 链队列的操作——销毁链队列 (补充)

- 算法思想：从队头结点开始，依次释放所有结点

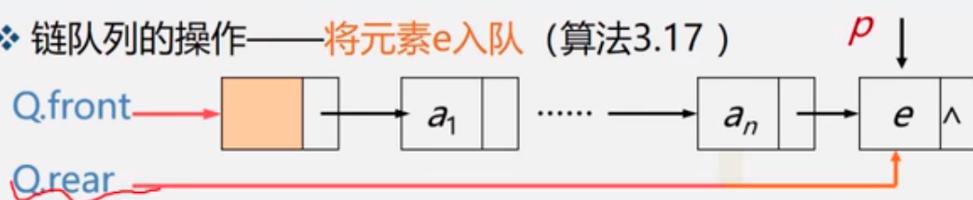


```
Status DestroyQueue (LinkQueue &Q){  
    while(Q.front){  
        p=Q.front->next; free(Q.front); Q.front=p;  
    }  
    return OK;  
}
```

3.5.3 链队——队列的链式表示和实现



❖ 链队列的操作——将元素e入队 (算法3.17)

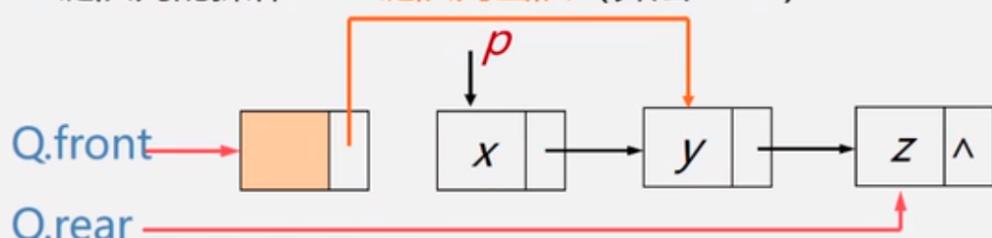


```
Status EnQueue(LinkQueue &Q, QElemType e){  
    p=(QueuePtr)malloc(sizeof(QNode));  
    if(!p) exit(OVERFLOW);  
    p->data=e; p->next=NULL;  
    Q.rear->next=p;  
    Q.rear=p;  
    return OK;  
}
```

青岛大学 | 数据科学

3.5.3 链队——队列的链式表示和实现

❖ 链队列的操作——链队列出队 (算法3.18)



$p=Q.front->next;$ $e=p->data;$

$Q.front->next=p->next;$



3.5.3 链队——队列的链式表示和实现

❖ 链队列的操作——链队列出队 (算法3.18)

```

Status DeQueue (LinkQueue &Q,QElemType &e){
    if(Q.front==Q.rear) return ERROR;
    p=Q.front->next;
    e=p->data;
    Q.front->next=p->next;
    if(Q.rear==p) Q.rear=Q.front;
    delete p;
    return OK;
}

```

3.5.3 链队——队列的链式表示和实现

❖ 链队列的操作——求链队列的队头元素 (算法3.19)

```

Status GetHead (LinkQueue Q, QElemType &e){
    if(Q.front==Q.rear) return ERROR;
    e=Q.front->next->data;
    return OK;
}

```

8.模式匹配算法

1.BF算法

串的模式匹配算法，通俗地理解，是一种用来判断两个串之间是否具有“主串与子串”关系的算法。

主串与子串：如果串 A（如 "shujujiegou"）中包含有串 B（如 "ju"），则称串 A 为主串，串 B 为子串。主串与子串之间的关系可简单理解为一个串“包含”另一个串的关系。

实现串的模式匹配的算法主要有以下两种：

1. 普通的模式匹配算法；
2. 快速模式匹配算法；

BF算法原理

普通模式匹配算法，其实现过程没有任何技巧，就是简单粗暴地拿一个串同另一个串中的字符一一比对，得到最终结果。

例如，使用普通模式匹配算法判断串 A ("abcac") 是否为串 B ("ababcabacbab") 子串的判断过程如下：

首先，将串 A 与串 B 的首字符对齐，然后逐个判断相对的字符是否相等，如图 1 所示：

B : a **b** a b c a b c a c b a b
A : a **b** c a c

图 1 串的第一次模式匹配示意图

图 1 中，由于串 A 与串 B 的第 3 个字符匹配失败，因此需要将串 A 后移一个字符的位置，继续同串 B 匹配，如图 2 所示：

B : a **b** a b c a b c a c b a b
A : a **b** c a c

图 2 串的第二次模式匹配示意图

图 2 中可以看到，两串匹配失败，串 A 继续向后移动一个字符的位置，如图 3 所示：

B : a b a **b** c a b c a c b a b
A : a b c a c

图 3 串的第三次模式匹配示意图

图 3 中，两串的模式匹配失败，串 A 继续移动，一直移动至图 4 的位置才匹配成功：

B : a b a b c a b c a c b a b
A : a b c a c

图 4 串模式匹配成功示意图

由此，串 A 与串 B 以供经历了 6 次匹配的过程才成功，通过整个模式匹配的过程，证明了串 A 是串 B 的子串（串 B 是串 A 的主串）。

串的模式匹配算法——BF算法

例如，设目标串S= "aaaaab"，模式串T= "aaab"。S的长度为n (n=6)，T的长度为m (m=4)。

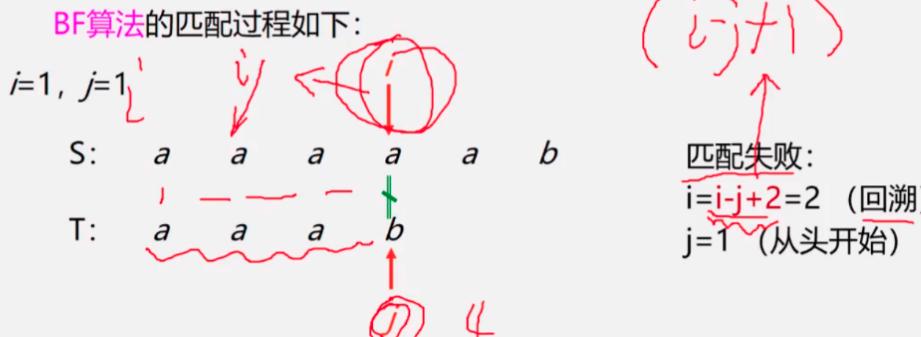
BF算法的匹配过程如下：

$i=1, j=1$

S: a a a a a b
T: a a a b

($i-j+1$)

匹配失败：
 $i=i-j+2=2$ (回溯)
 $j=1$ (从头开始)



BF算法时间复杂度



例: S= '0000000001' , T= '0001' , pos=1

若n为主串长度, m为子串长度, 最坏情况是

- ✓ 主串前面 $n-m$ 个位置都部分匹配到子串的最后一位, 即这 $n-m$ 位各比较了m次
- ✓ 最后m位也各比较了1次

$$\text{总次数为: } (n-m)*m + m = (n-m+1)*m$$

若 $m < n$, 则算法复杂度 $O(n*m)$

```
#include <stdio.h>
#include <string.h>
#include "./SString.h"
int BF(SqString s, SqString t){
    int i = 1, j = 1; // 我这里字符从1开始存储
    while(i <= s.length && j <= t.length){ // 不超过两个字符串的长度
        if(s.data[i] == t.data[j]){ // 相同则往下
            ++i;
            ++j;
        }
        else{ // 不相同则回溯
            i = i - j + 2; // 从1移动到j个位置, 移动了j - 1
            // 个单位, 要回到原始位置的下一个则是i - (j - 1) + 1
            j = 1;
        }
    }
    if(j > t.length){ // 返回匹配的第一个字符的下标, 匹配完成时j >= 子串的长度
        printf("匹配到的子串第一个字母的下标是: %d\n", i - t.length);
        return i - t.length;
    }
    else{
        printf("没有与之相匹配的子串\n");
        return 0;
    }
}
int main(){
    SqString s1;
    SqString s2;
    // 初始化字符串
    char a[100] = "heAAAAalloworld";
    char b[100] = "AAAA";
    StrAssign(s1, a);
    StrAssign(s2, b);
    printf("s1: \n");
    DispStr(s1);
    printf("s2: \n");
    DispStr(s2);
    // 调用BF算法
    BF(s1, s2);
}
```

```
    return 0;
}
```

The screenshot shows a code editor with several tabs at the top: sort.cpp, 03选择排序.cpp, 04student.cpp, students.c, 01统计大写字母数.cpp, and BF.cpp. The BF.cpp tab is active, displaying the following C code:

```
1 #include <stdio.h>
2 #include <string.h>
3 #include "./SString.h"
4 int BF(SqString s, SqString t){
5     int i = 1, j = 1; // 我这里字符从1开始存储
6     while(i <= s.length && j <= t.length){//不超过两个字符串的长度
7         if(s.data[i] == t.data[j]){//相同则往下
8             ++i;
9             ++j;
10        }
11        else{//不相同则回溯
12            i = i - j + 2;//从1移动到j个位置，移动了j - 1
13            //个单位，要回到原始位置的下一个则是i - (j - 1) + 1
14            j = 1;
15        }
16    }
17    if(j > t.length){//返回匹配的第一个字符的下标，匹配完成时j>=子串的长度
18        printf("匹配到的子串第一个字母的下标是: %d\n", i - t.length);
19        return i - t.length;
20    }
21    else{
22        printf("没有与之相匹配的子串\n");
23        return 0;
24    }
25 }
26 int main(){
27     SqString s1;
```

On the right side, there is a terminal window showing the execution of the program. It prints:

```
s1:
h e A A A A l l o w o r l d
s2:
A A A A
匹配到的子串第一个字母的下标是: 3
[1] + Done
} 0<" /tmp/Microsoft-MIEngine-In-j:
3zxya.f5t "
按任意键继续... 
```

2.*KMP算法

0.完整演示过程

1.现象

仅仅后移模式串，比较指针不回溯

- 第一个现象：某个位置发生不匹配时比较指针左边是匹配的



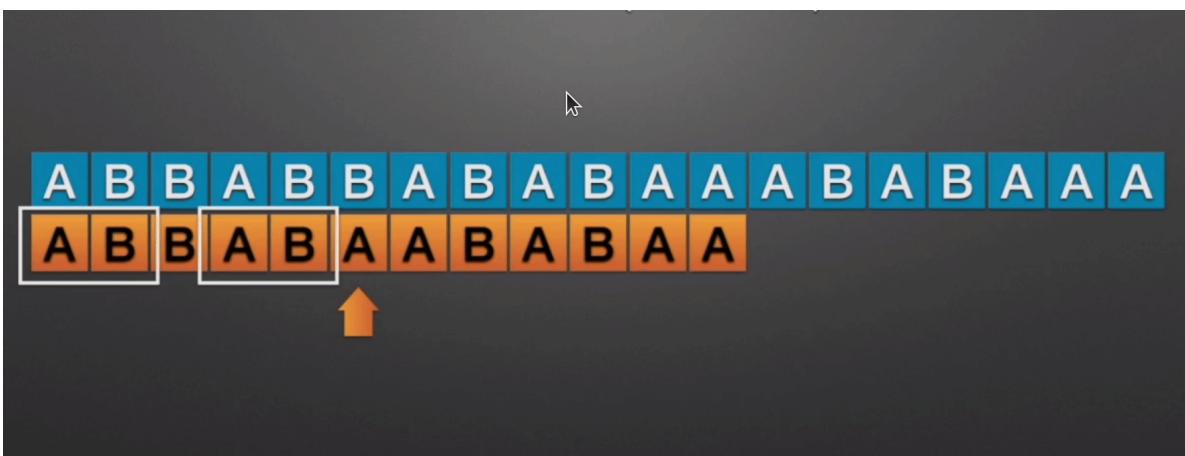
- 第二个现象：模式串前缀后缀相匹配，最长公共前后缀AB和AB

KMP算法(易懂版)



2.核心步骤

移动模式串的前缀到模式串的后缀位置，这样就能保证左边的串和主串匹配



KMP算法(易懂版)

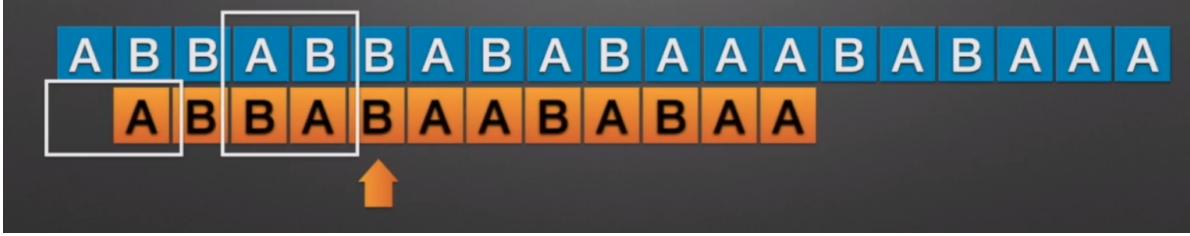


KMP算法(易懂版)

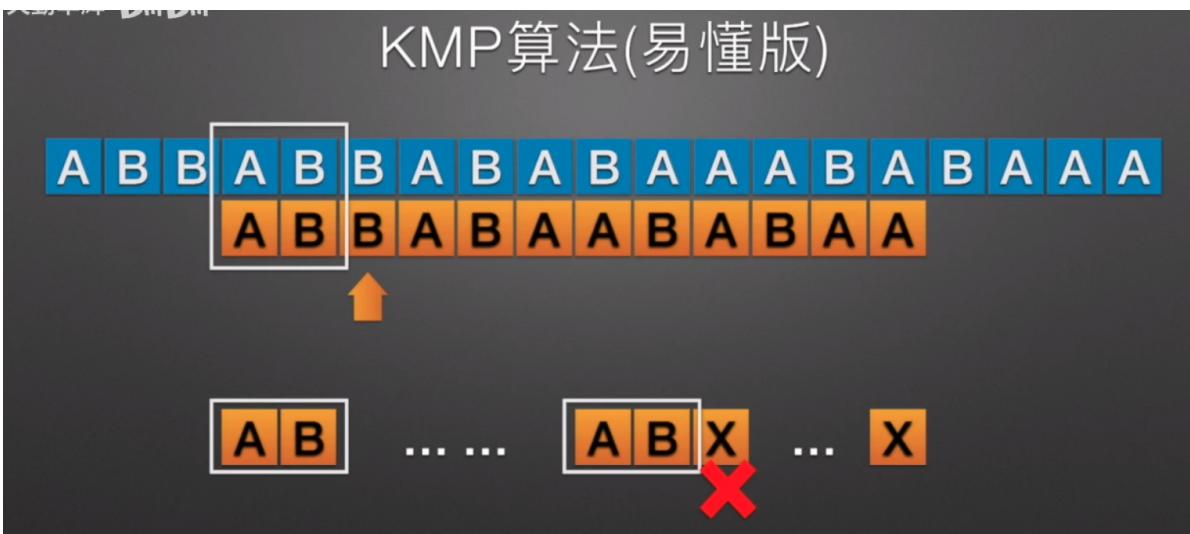


- 中间并没有匹配的模式串

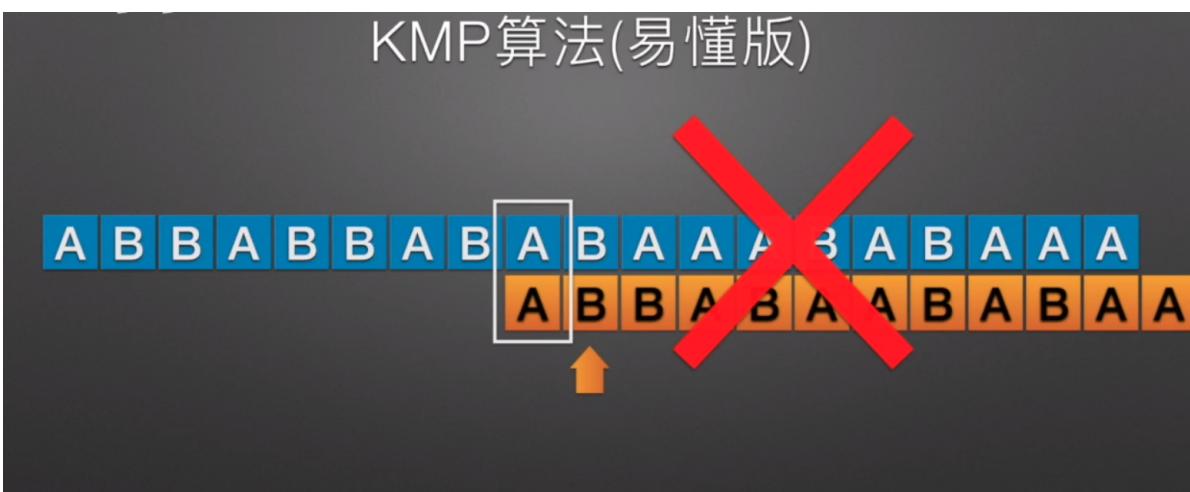
KMP算法(易懂版)



- 一般模式：可以不用看主串，衍生到串对于所有位置都有可能不匹配的现象



模式串超出主串的位置，判断结束不匹配



- 不匹配找最长公共前后缀

KMP算法(易懂版)



- 长度不能等于子串长度，无意义

KMP算法(易懂版)



KMP算法(易懂版)



3.去掉主串

KMP算法(易懂版)



- 存入数组，从1开始存

KMP算法(易懂版)

0 1 2 3 4 5 6 7 8 9 10 11 12

A B A B A A A B A B A B A A



- 第一个位置不匹配

X B B A B B B A B A B A A A A B A B A A A

A B A B A A A A B A B A A



X B B A B B B A B A B A A A A B A B A A A

A B A B A A A A B A B A A



让模式串中一号位置的字符和主串的下一位比较

KMP算法(易懂版)

0 1 2 3 4 5 6 7 8 9 10 11 12

A B A B A A A A B A B A A



1号位与主串下一位比较

一号位不匹配

天勤率

- 二号位不匹配

B	X	B	A	B	B	A	B	A	B	A	A	A	A	A	A
A	B	A	B	A	A	A	A	B	A	B	A	A	A	A	A



不匹配的位置前后缀的长度为1，但是公共前后缀的长度<不匹配位置前的串长度，所以最长公共前后缀为0，所以前移一位

模式串的1号位和主串的当前位置比较

B	X	B	A	B	B	A	B	A	B	A	A	A	A	A	A
A	B	A	B	A	A	A	A	B	A	B	A	A	A	A	A



B	X	B	A	B	B	A	B	A	B	A	A	A	A	A	A
A	B	A	B	A	A	A	A	B	A	B	A	A	A	A	A



- 3号位不匹配

KMP算法(易懂版)

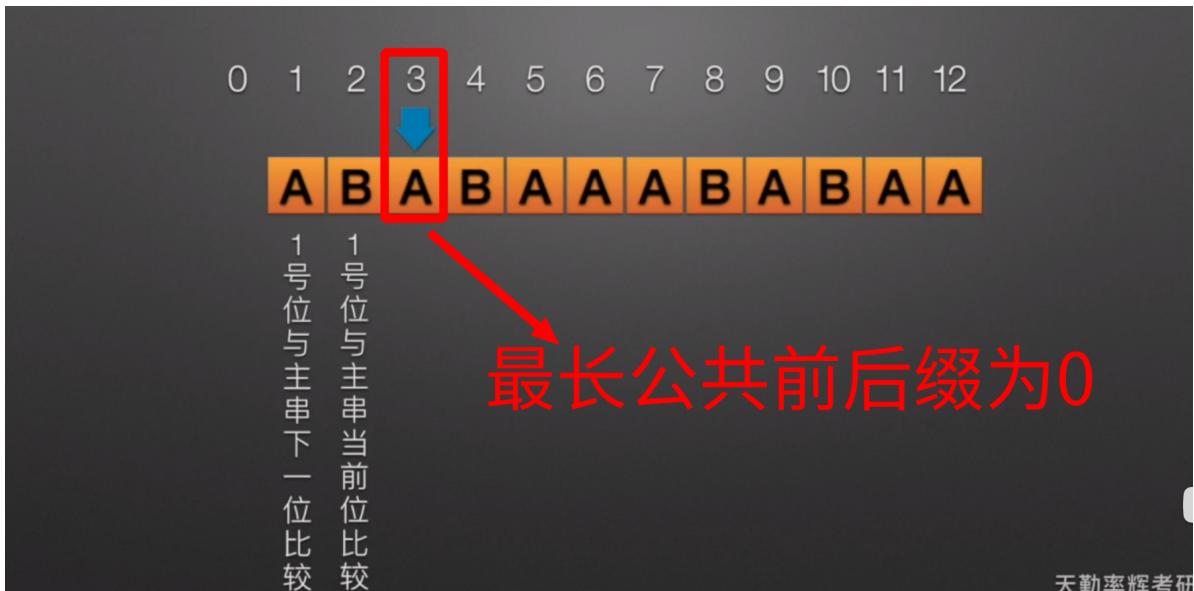
0 1 2 3 4 5 6 7 8 9 10 11 12



A	B	A	B	A	A	A	A	B	A	B	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---

1号位
与主串下一位
比较
1号位
与主串当前位
比较



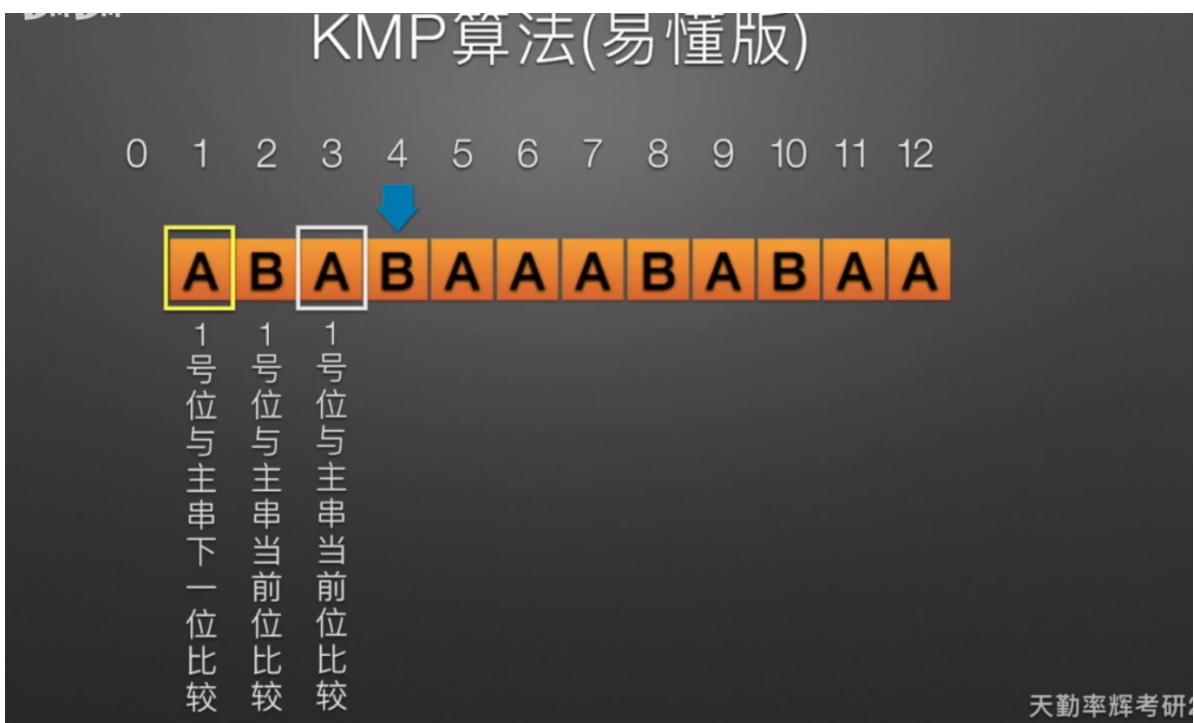


天勤率辉考研

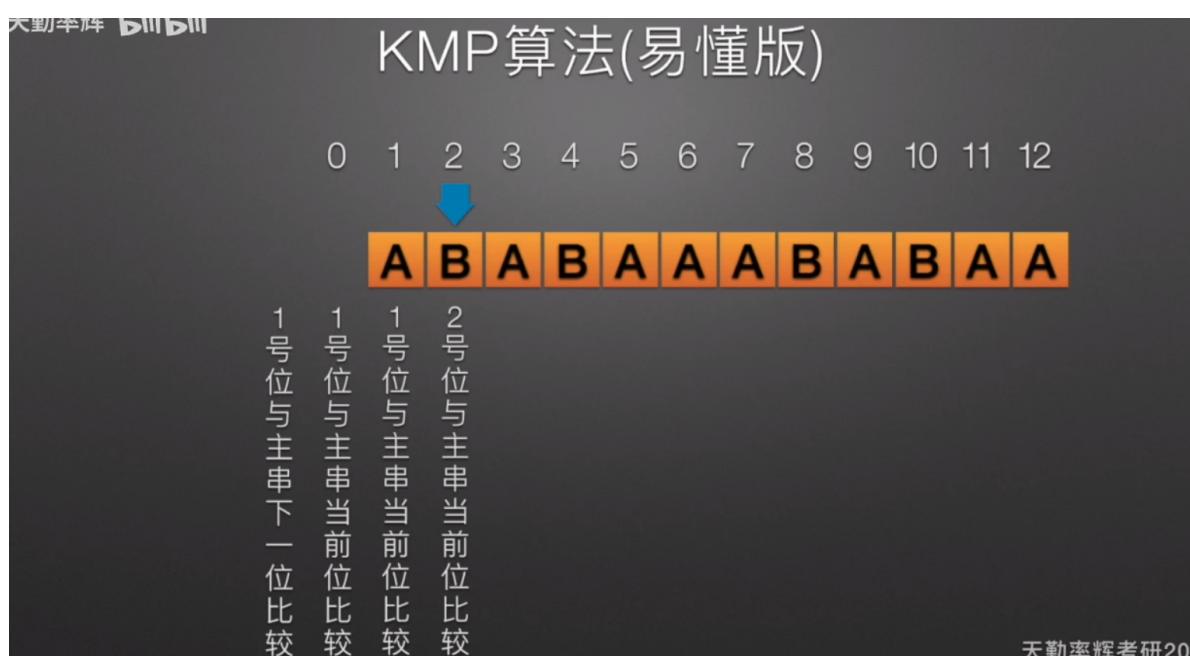
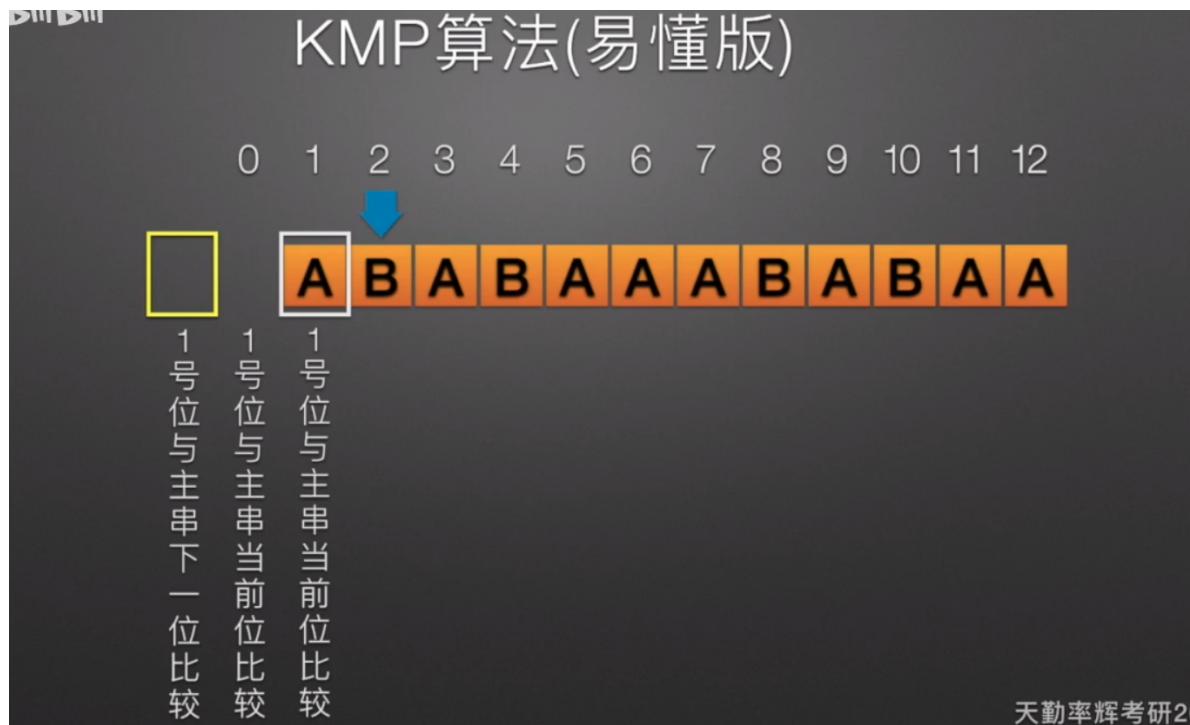


天勤率辉考研202

- 4号位不匹配，最长公共前后缀为1

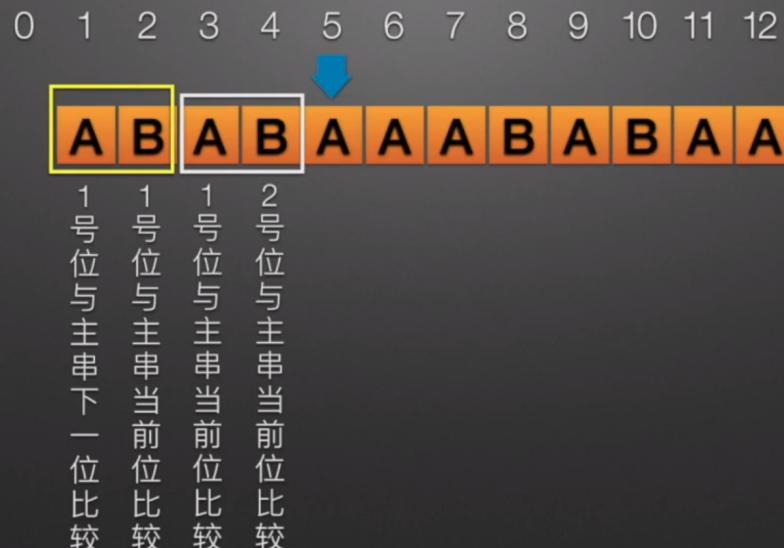


天勤率辉考研2



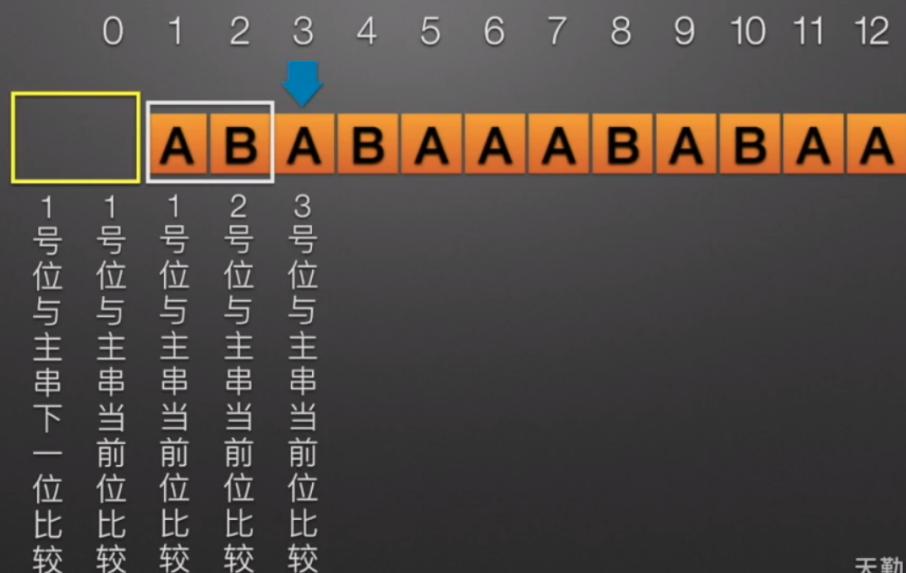
- 5号位不匹配

KMP算法(易懂版)



天勤率辉者研

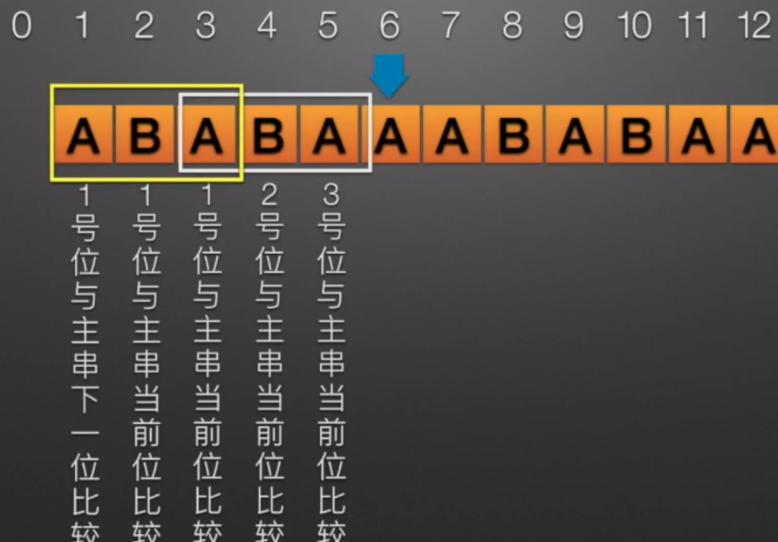
KMP算法(易懂版)



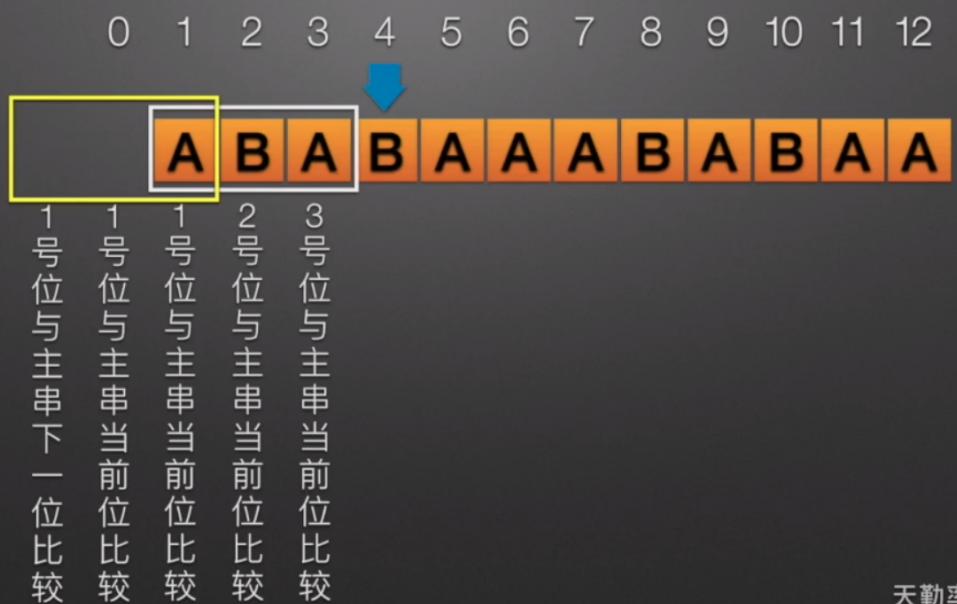
天勤率辉者

- 6号位不匹配的情况

KMP算法(易懂版)

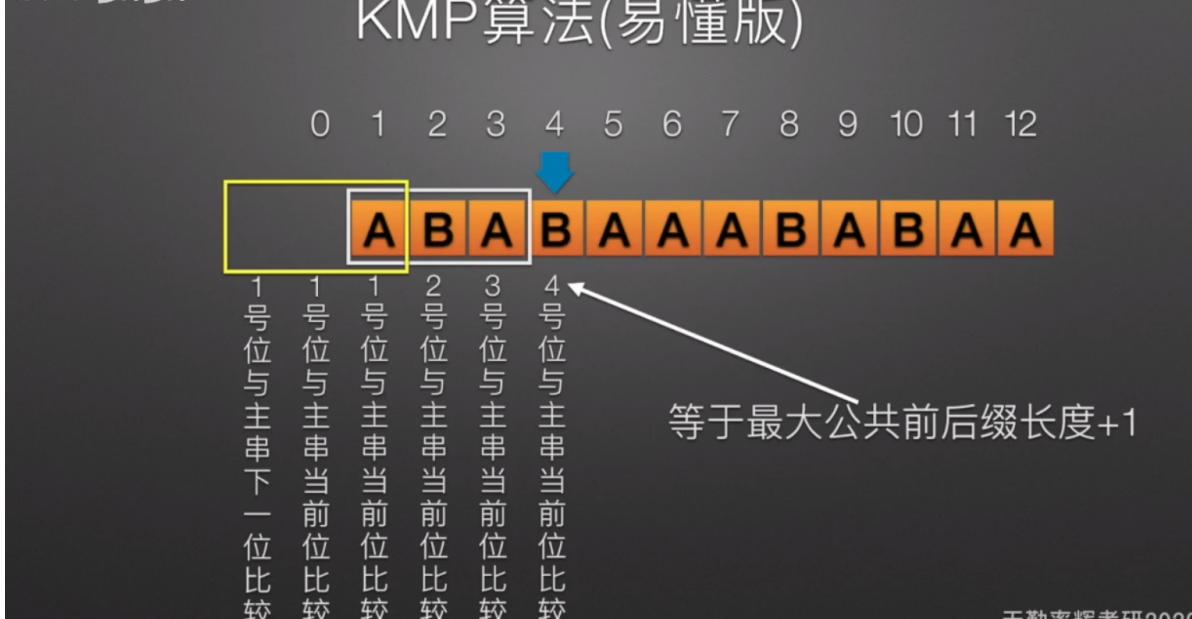


KMP算法(易懂版)

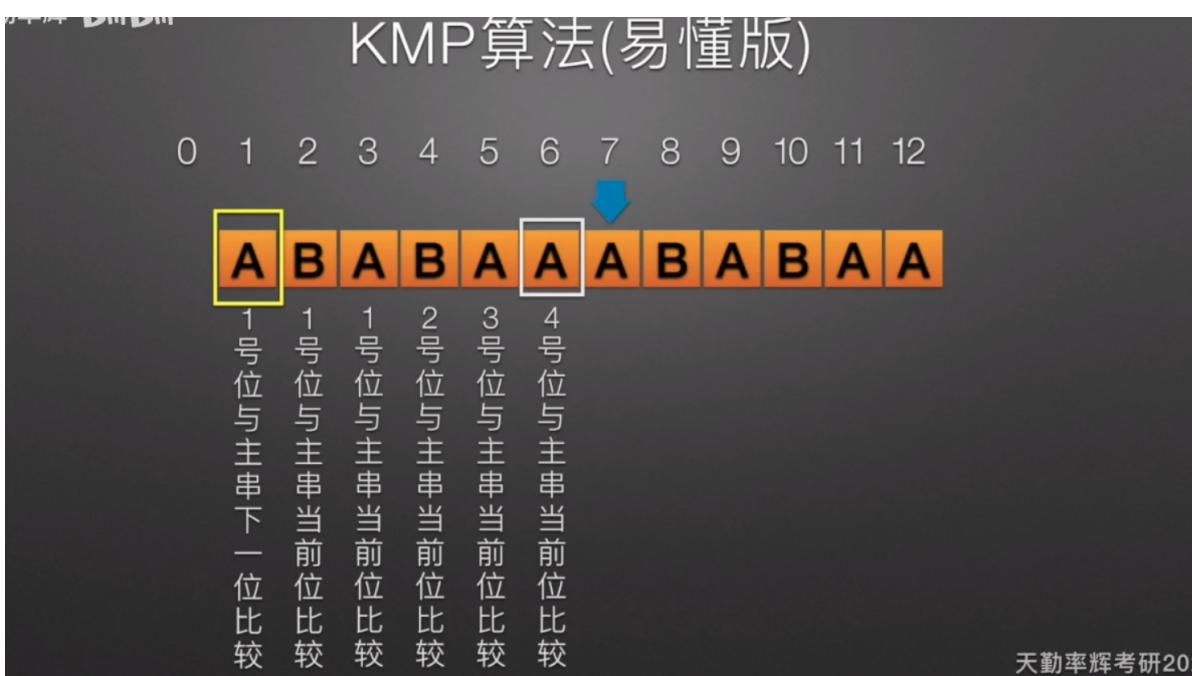


规律出来了

KMP算法(易懂版)

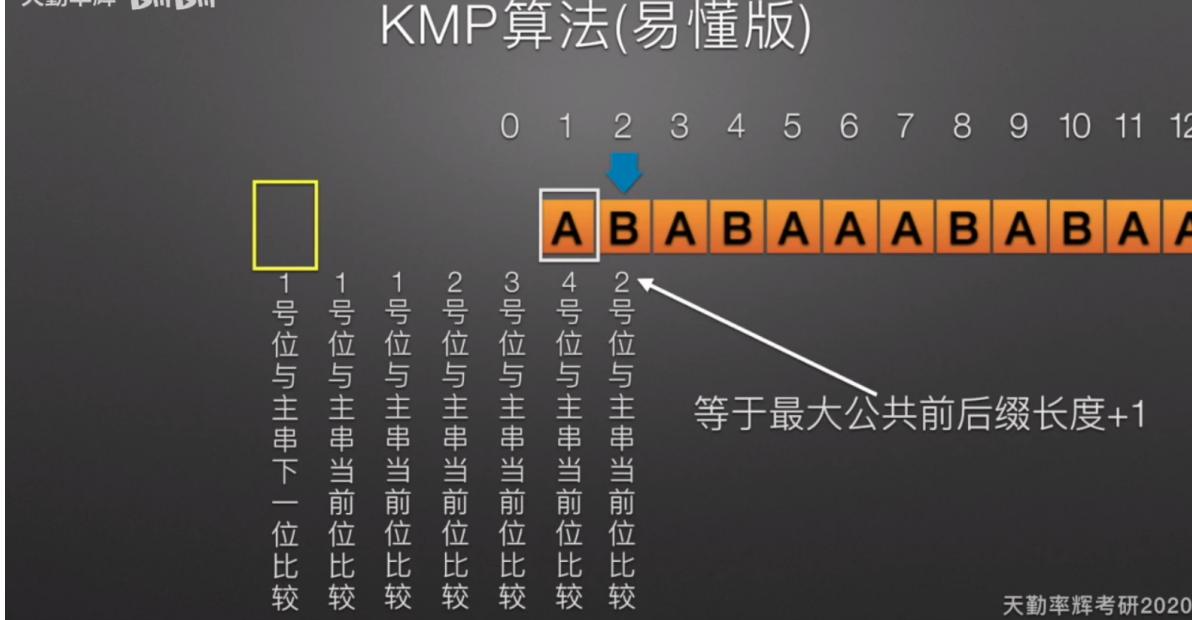


- 7号位不匹配



天勤率辉考研2020

KMP算法(易懂版)



天勤率辉考研2020

- 8号位不匹配

KMP算法(易懂版)

0 1 2 3 4 5 6 7 8 9 10 11 12

A B A B A A A B A B A A

1号位与主串下一位比较	1号位与主串当前位比较	1号位与主串当前位比较	2号位与主串当前位比较	3号位与主串当前位比较	4号位与主串当前位比较	2号位与主串当前位比较
-------------	-------------	-------------	-------------	-------------	-------------	-------------

勤率辉 真的爱了！
感人，俺懂了

吴金 真呼牛皮 吴金年代
妙啊，大师我悟了

KMP算法(易懂版)

0 1 2 3 4 5 6 7 8 9 10 11 12

A B A B A A A B A B A A

1号位与主串下一位比较	1号位与主串当前位比较	1号位与主串当前位比较	2号位与主串当前位比较	3号位与主串当前位比较	4号位与主串当前位比较	2号位与主串当前位比较
-------------	-------------	-------------	-------------	-------------	-------------	-------------

天勤率辉考研

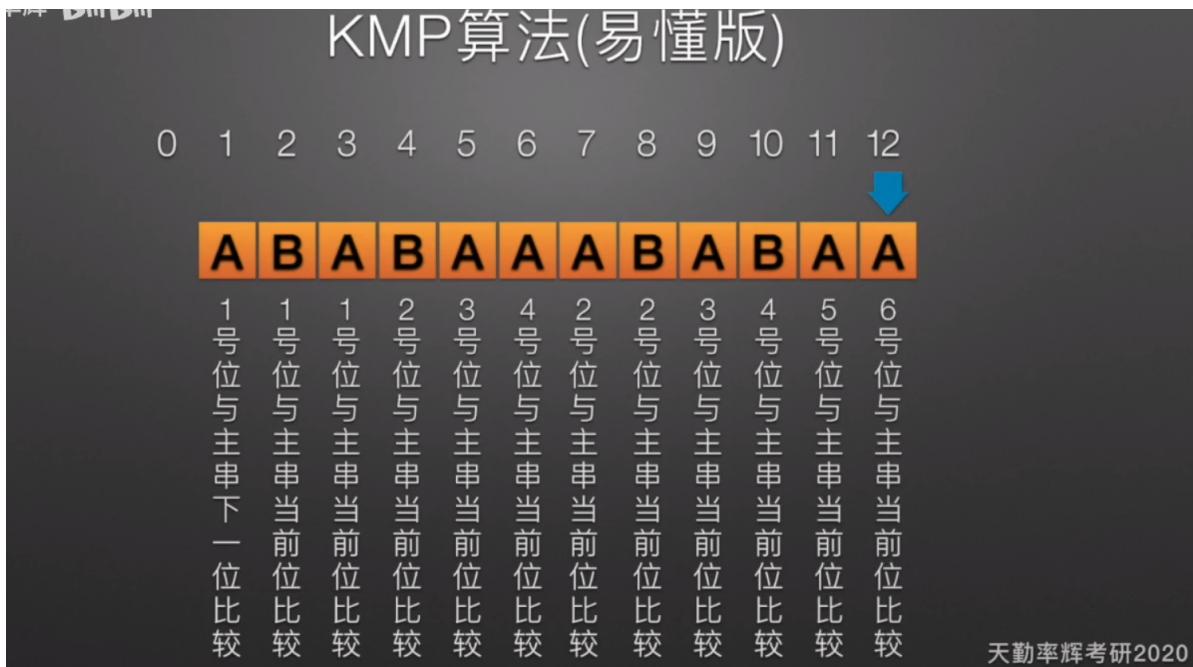
4.总结

KMP算法

为此，定义 $\text{next}[j]$ 函数，表明当模式中第 j 个字符与主串中相应字符“失配”时，在模式中需重新和主串中该字符进行比较的字符的位置。

$$\text{next}[j] = \begin{cases} 0 & \max\{k | 1 < k < j, \text{且 } "p_1 \dots p_{k-1}" = "p_{j-k+1} \dots p_{j-1}"\} \\ 1 & \text{当 } j=1 \text{ 时} \\ & \text{其他情况} \end{cases}$$

算法的关键是求next【j】，除了下标1位置之外，其他位置描述都是一样的，都是最长公共前后缀
+1



- 将第一句话标记位0，看到0按照第一句话的方式处理



- 将每句话的数字抽取出来作为代号放入数组中

KMP算法(易懂版)

值	0	1	1	2	3	4	2	2	3	4	5	6	
下标	0	1	2	3	4	5	6	7	8	9	10	11	12

KMP算法(易懂版)

A B A B A A A B A B A A

值	0	1	1	2	3	4	2	2	3	4	5	6
下标	0	1	2	3	4	5	6	7	8	9	10	11

next数组

5.next数组代码说明

求解next数组前要懂得以下几个概念：

- 1、前缀：包含首位字符但不包含末位字符的子串。
- 2、后缀：包含末位字符但不包含首位字符的子串。
- 3、next数组定义：当主串与模式串的某一位字符不匹配时，模式串要回退的位置。
- 4、next[j]：其值 = 第j位字符前面j-1位字符组成的子串的前后缀重合字符数+1

手算Next数组

j: 1 2 3 4 5 6 7 8
P: a b a a b c a c
Next[j]: 0 1 1 2 2 3 1 2

当j = 1时，规定next[1] = 0
当j = 2时，j前子串为“a”，next[2] = 1
当j = 3时，j前子串为“ab”，next[3] = 1
当j = 4时，j前子串为“aba”，next[4] = 2
当j = 5时，j前子串为“abaa”，next[5] = 2
当j = 6时，j前子串为“abaab”，next[6] = 3
当j = 7时，j前子串为“abaabc”，next[7] = 1
当j = 8时，j前子串为“abaabca”，next[8] = 2

- 有7时

```
int GetNext(char ch[], int length, int next[]){ // length为串ch的长度
    next[1] = 0;
    int i = 1, j = 0; // i当前主串正在匹配的字符位置，也是next数组的索引
    while(i <= length){
        if(j == 0 || ch[i] == ch[j]) next[++i] = ++j;
        else j = next[j];
    }
}
```

j: 1 2 3 4 5 6
P: a b a b a ?
Next[j]: 0 1 1 2 3 4

当你理解不了一段代码的时候，就跟着代码走一次

- ◆ 第1次循环：i = 1, j = 0, j==0成立，i = 2, j = 1, next[2] = 1;
- ◆ 第2次循环：i = 2, j = 1, ch[i]==ch[j] 不成立，j = next[j] = next[1] = 0;
- ◆ 第3次循环：i = 2, j = 0, j==0成立，i = 3, j = 1, next[3] = 1;
- ◆ 第4次循环：i = 3, j = 1, ch[i]==ch[j] 成立，i = 4, j = 2, next[4] = 2;
- ◆ 第5次循环：i = 4, j = 2, ch[i]==ch[j] 成立，i = 5, j = 3, next[5] = 3;
- ◆ 第6次循环：i = 5, j = 3, ch[i]==ch[j] 成立，i = 6, j = 4, next[6] = 4;

```

int GetNext(char ch[], int length, int next[]){ // length为串ch的长度
    next[1] = 0;
    int i = 1, j = 0; // i当前主串正在匹配的字符位置，也是next数组的索引
    while(i <= length){
        if(j==0 || ch[i]==ch[j]) next[++i] = ++j;
        else j = next[j];
    }
}

```

j: 1 2 3 4 5 6 7
P: a b a b a c ?

当你理解不了一段代码的时候，就跟着代码走一次

Next[j]: 0 1 1 2 3 4 1

- ◆ 第1次循环: $i = 1, j = 0, j == 0$ 成立, $i = 2, j = 1, \text{next}[2] = 1;$
- ◆ 第2次循环: $i = 2, j = 1, \text{ch}[i] == \text{ch}[j]$ 不成立, $j = \text{next}[j] = \text{next}[1] = 0;$
- ◆ 第3次循环: $i = 2, j = 0, j == 0$ 成立, $i = 3, j = 1, \text{next}[3] = 1;$
- ◆ 第4次循环: $i = 3, j = 1, \text{ch}[i] == \text{ch}[j]$ 成立, $i = 4, j = 2, \text{next}[4] = 2;$
- ◆ 第5次循环: $i = 4, j = 2, \text{ch}[i] == \text{ch}[j]$ 成立, $i = 5, j = 3, \text{next}[5] = 3;$
- ◆ 第6次循环: $i = 5, j = 3, \text{ch}[i] == \text{ch}[j]$ 成立, $i = 6, j = 4, \text{next}[6] = 4;$
- ◆ 第7次循环: $i = 6, j = 4, \text{ch}[i] == \text{ch}[j]$ 不成立, $j = \text{next}[j] = \text{next}[4] = 2;$
- ◆ 第8次循环: $i = 6, j = 2, \text{ch}[i] == \text{ch}[j]$ 不成立, $j = \text{next}[j] = \text{next}[2] = 1;$
- ◆ 第9次循环: $i = 6, j = 1, \text{ch}[i] == \text{ch}[j]$ 不成立, $j = \text{next}[j] = \text{next}[1] = 0;$
- ◆ 第10次循环: $i = 6, j = 0, j == 0$ 成立, $i = 7, j = 1, \text{next}[7] = 1;$



探索规律背后的原理

1、 $\text{next}[j+1]$ 的最大值为 $\text{next}[j]+1$ 。

2、如果 $P_{k_1} \neq P_j$, 那么 $\text{next}[j+1]$ 可能的次大值为 $\text{next}[\text{next}[j]]+1$, 以此类推即可高效求出 $\text{next}[j+1]$ 。 (重点)



从头走一遍流程

- ①求 $\text{next}[j+1]$, 则已知 $\text{next}[1], \text{next}[2], \dots, \text{next}[j]$
- ②假设 $\text{next}[j]=k_1$, 则有 $P_1 \dots P_{k_1-1} = P_{j-k_1+1} \dots P_{j-1}$ (前 k_1-1 位字符与后 k_1-1 位字符重合)
- ③如果 $P_{k_1}=P_j$, 则 $P_1 \dots P_{k_1-1} P_{k_1} = P_{j-k_1+1} \dots P_{j-1} P_j$, 则 $\text{next}[j+1]=k_1+1$, 否则进入下一步
- ④假设 $\text{next}[k_1]=k_2$, 则有 $P_1 \dots P_{k_2-1} = P_{k_1-k_2+1} \dots P_{k_1-1}$
- ⑤第二第三步联合得到: $P_1 \dots P_{k_2-1} = P_{k_1-k_2+1} \dots P_{k_1-1} = P_{j-k_1+1} \dots P_{k_2-k_1+j-1} = P_{j-k_2+1} \dots P_{j-1}$ 即四段重合
- ⑥这时候, 再判断如果 $P_{k_2}=P_j$, 则 $P_1 \dots P_{k_2-1} P_{k_2} = P_{j-k_2+1} \dots P_{j-1} P_j$, 则 $\text{next}[j+1]=k_2+1$; 否则再取 $\text{next}[k_2]=k_3 \dots$ 以此类推

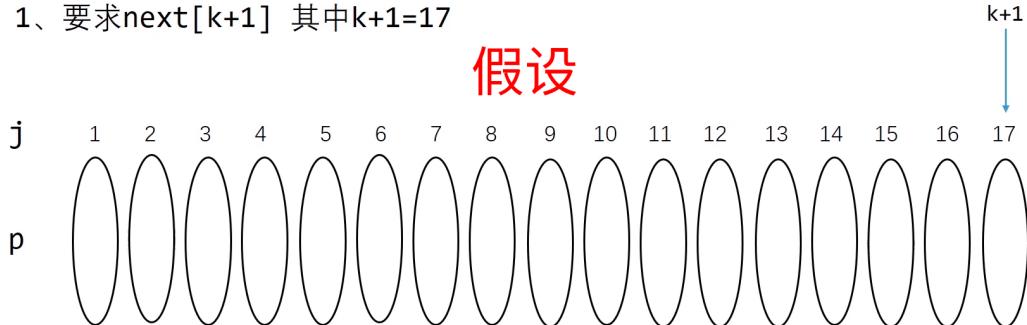
- 假设 $k+1$ 等于17



图解原理

1、要求 $\text{next}[k+1]$ 其中 $k+1=17$

假设



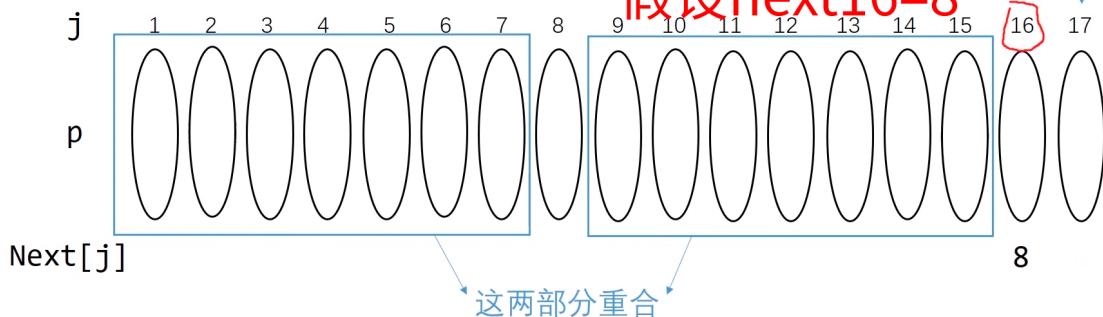
- 假设 $\text{next}[16]=8$

3 已知next[16]

2、已知next[16]=8，则元素有以下关系：

如果 $P_8 = P_{16}$, 则明显 $\text{next}[17] = 8 + 1 = 9$

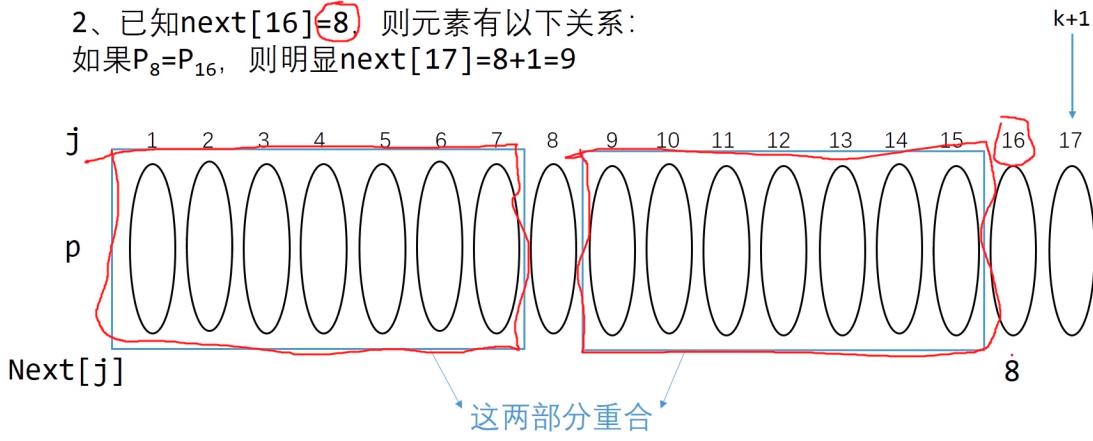
假设next16=8



2. 已知next[16]

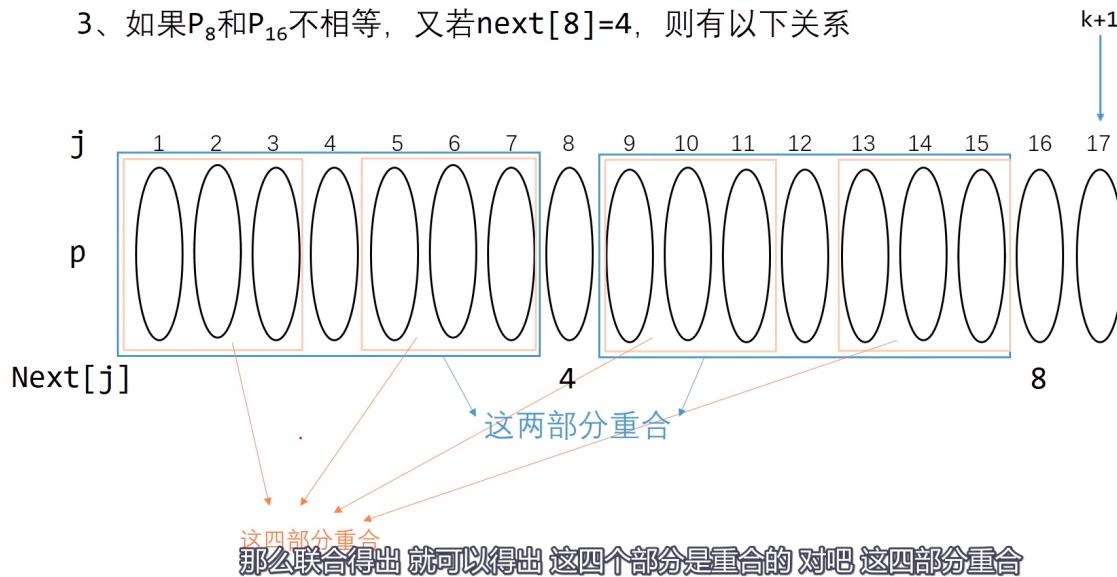
2、已知next[16]=8，则元素有以下关系：

如果 $P_8 = P_{16}$, 则明显 $\text{next}[17] = 8 + 1 = 9$

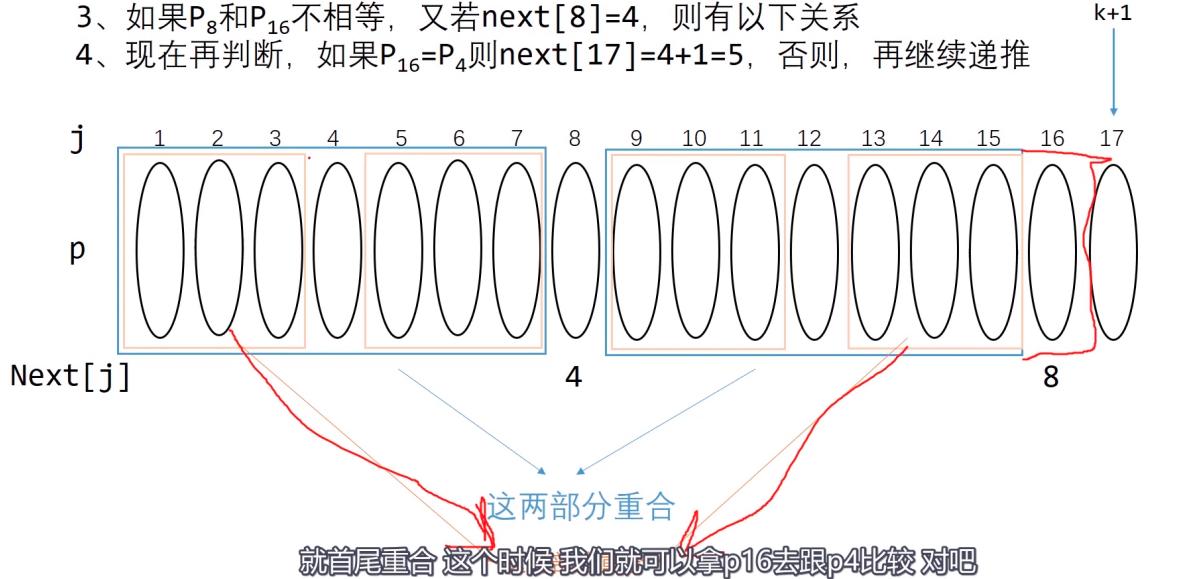


对吧 因为你这个是8 8=7+1 证明你前缀有七位字符跟后面的七位字符是重合的

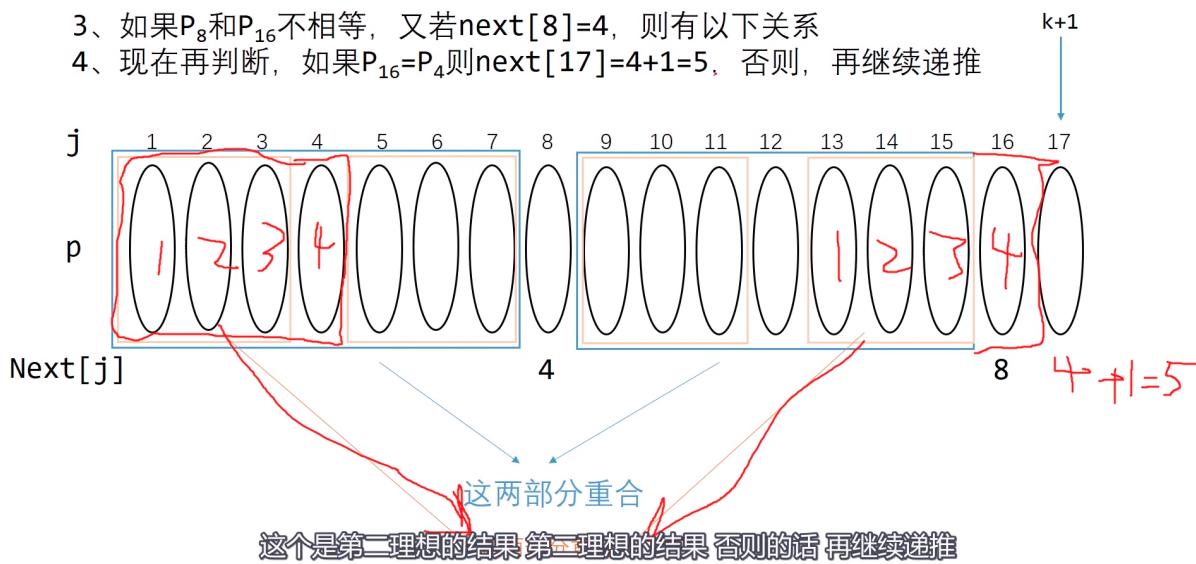
3、如果 P_8 和 P_{16} 不相等，又若 $next[8]=4$ ，则有以下关系



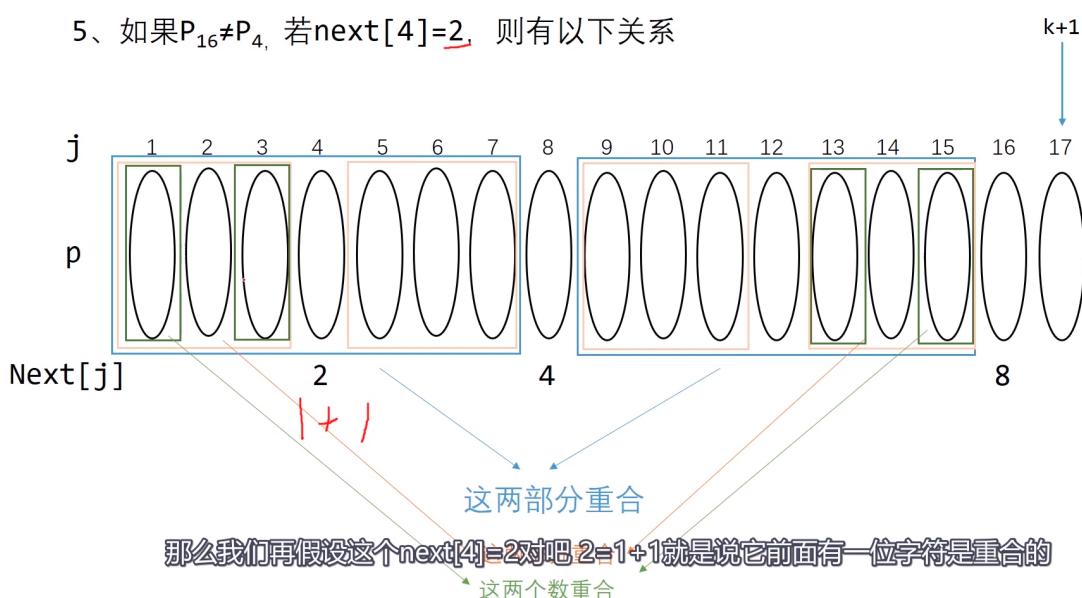
- 3、如果 P_8 和 P_{16} 不相等，又若 $\text{next}[8]=4$ ，则有以下关系
 4、现在再判断，如果 $P_{16}=P_4$ 则 $\text{next}[17]=4+1=5$ ，否则，再继续递推



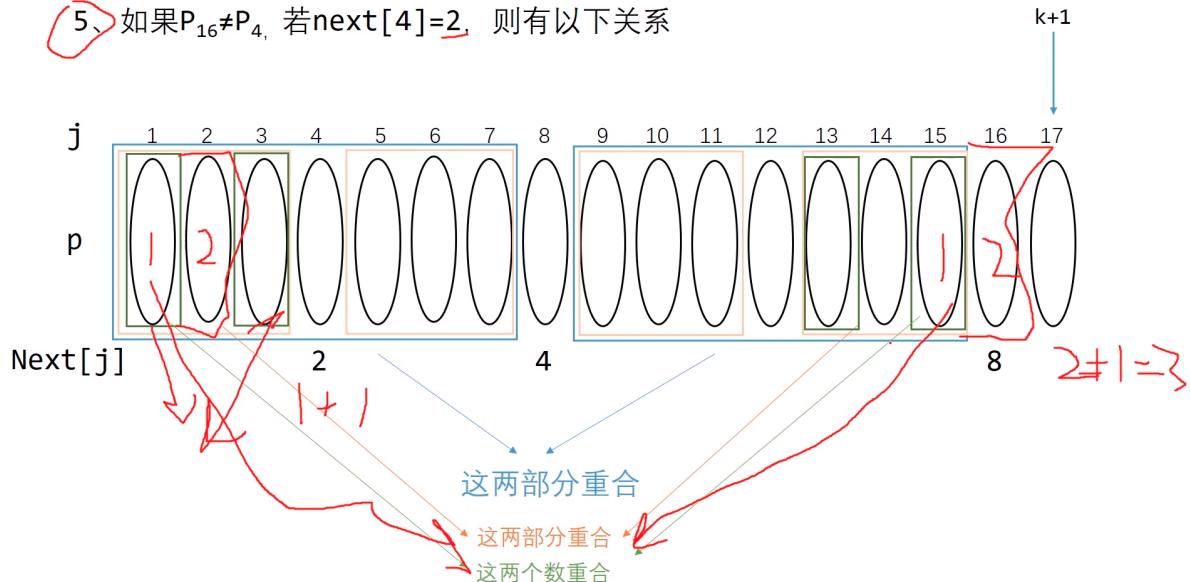
- 3、如果 P_8 和 P_{16} 不相等，又若 $\text{next}[8]=4$ ，则有以下关系
 4、现在再判断，如果 $P_{16}=P_4$ 则 $\text{next}[17]=4+1=5$ ，否则，再继续递推



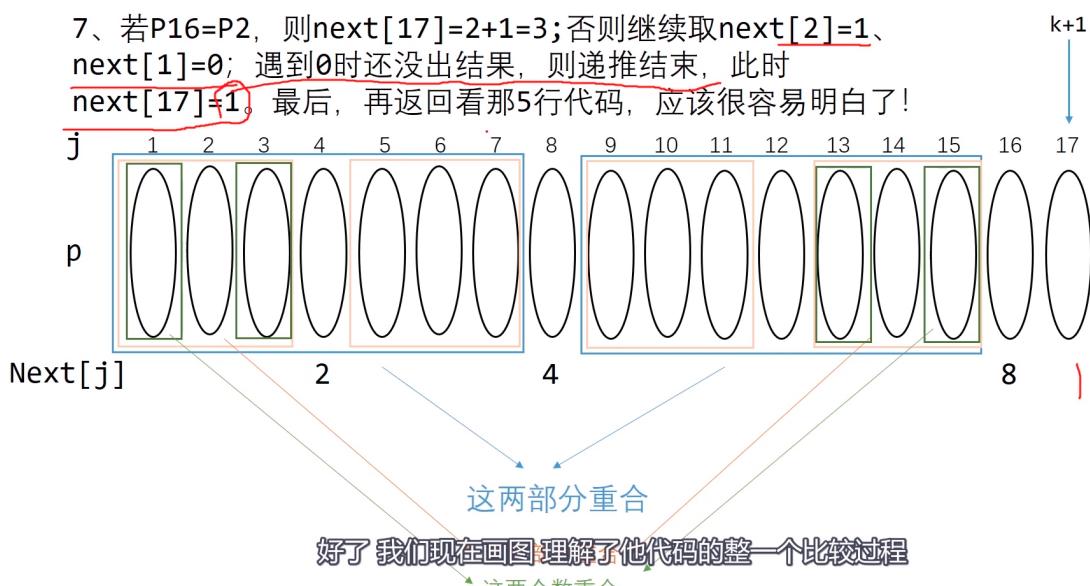
- 5、如果 $P_{16} \neq P_4$, 若 $\text{next}[4]=2$ ，则有以下关系



5、如果 $P_{16} \neq P_4$, 若 $\text{next}[4] = 2$, 则有以下关系



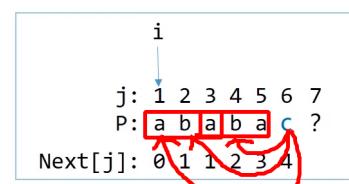
7、若 $P_{16}=P_2$, 则 $\text{next}[17]=2+1=3$; 否则继续取 $\text{next}[2]=1$ 、
 $\text{next}[1]=0$; 遇到0时还没出结果, 则递推结束, 此时
 $\text{next}[17]=1$ 。最后, 再返回看那5行代码, 应该很容易明白了!



好了 我们现在画图理解了他代码的整个比较过程

* 这两个数重合

```
int GetNext(char ch[], int length, int next[]){ // length为串ch的长度
    next[1] = 0;
    int i = 1, j = 0; // i当前主串正在匹配的字符位置, 也是next数组的索引
    while(i <= length){
        if(j==0 || ch[i]==ch[j]) next[++i] = ++j;
        else j = next[j];
    }
}
```



- ◆ 第1次循环: $i = 1, j = 0$, $j==0$ 成立, $i = 2, j = 1$, $\text{next}[2] = 1$;
- ◆ 第2次循环: $i = 2, j = 1$, $\text{ch}[i]==\text{ch}[j]$ 不成立, $j = \text{next}[j] = \text{next}[1] = 0$;
- ◆ 第3次循环: $i = 2, j = 0$, $j==0$ 成立, $i = 3, j = 1$, $\text{next}[3] = 1$;
- ◆ 第4次循环: $i = 3, j = 1$, $\text{ch}[i]==\text{ch}[j]$ 成立, $i = 4, j = 2$, $\text{next}[4] = 2$;
- ◆ 第5次循环: $i = 4, j = 2$, $\text{ch}[i]==\text{ch}[j]$ 成立, $i = 5, j = 3$, $\text{next}[5] = 3$;
- ◆ 第6次循环: $i = 5, j = 3$, $\text{ch}[i]==\text{ch}[j]$ 成立, $i = 6, j = 4$, $\text{next}[6] = 4$;
- ◆ 第7次循环: $i = 6, j = 4$, $\text{ch}[i]==\text{ch}[j]$ 不成立, $j = \text{next}[j] = \text{next}[4] = 2$;
- ◆ 第8次循环: $i = 6, j = 2$, $\text{ch}[i]==\text{ch}[j]$ 不成立, $j = \text{next}[j] = \text{next}[2] = 1$;
- ◆ 第9次循环: $i = 6, j = 1$, $\text{ch}[i]==\text{ch}[j]$ 不成立, $j = \text{next}[j] = \text{next}[1] = 0$;
- ◆ 第10次循环: $i = 6, j = 0$, $j==0$ 成立, $i = 7, j = 1$, $\text{next}[7] = 1$;

9. 树的递归算法

10.树的非递归

1.先序

5 二叉树遍历的非递归算法

前序遍历——非递归算法（也称先序遍历）

二叉树前序遍历的非递归算法的**关键**：在前序遍历完某结点的整个左子树后，如何找到该结点的**右子树**的根指针。

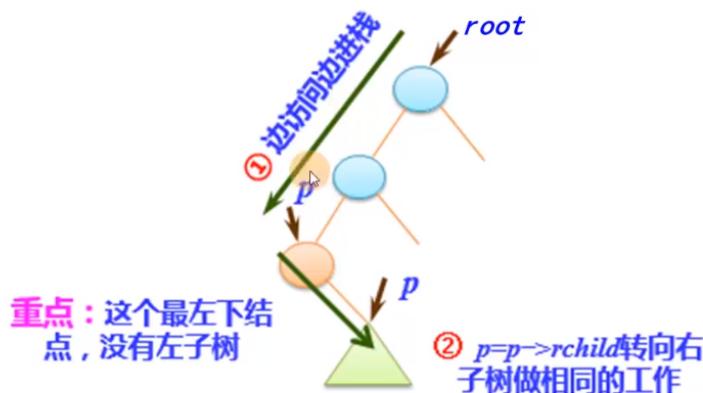
解决办法：在访问完该结点后，将该结点的指针保存在**栈**中，以便以后能通过它找到该结点的右子树。

在前序遍历中，设要遍历二叉树的根指针为root，则有两种可能：

- (1) 若 $\text{root} \neq \text{NULL}$ ，则表明？如何处理？
- (2) 若 $\text{root} = \text{NULL}$ ，则表明？如何处理？

前序遍历的**非递归**实现

p 用于结点遍历，初始时 $p=\text{root}$



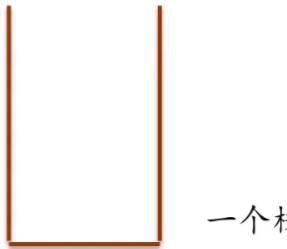
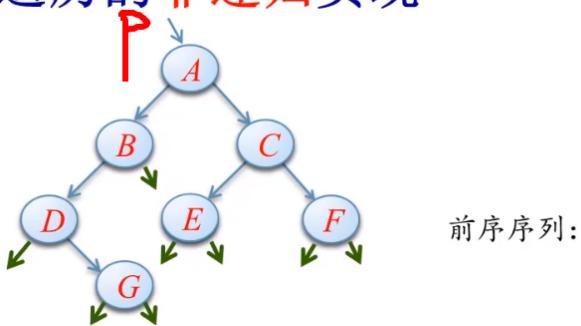
- 栈中结点均已经访问
- p 指向刚刚出栈结点的右子树



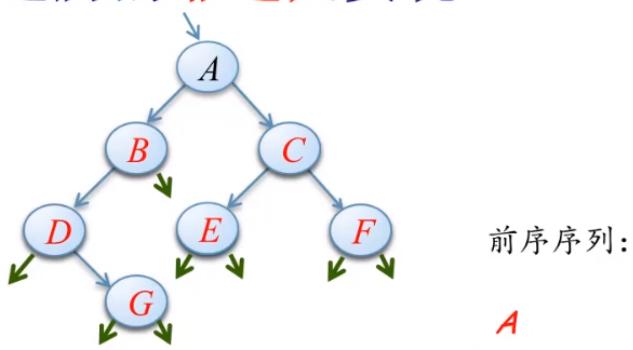
→ 栈空且 $p=\text{NULL}$ 结束

- 过程演示

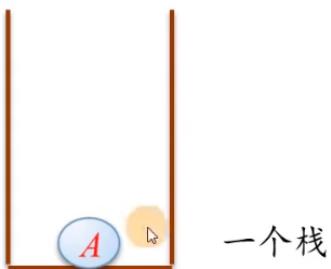
前序遍历的非递归实现



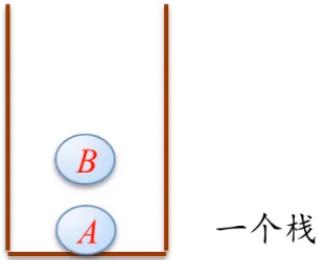
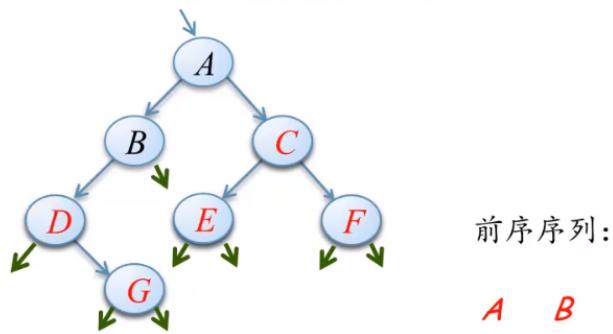
前序遍历的非递归实现



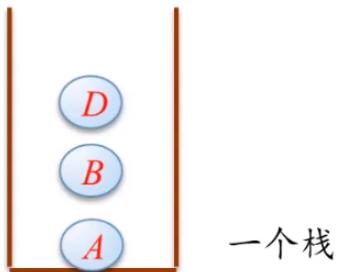
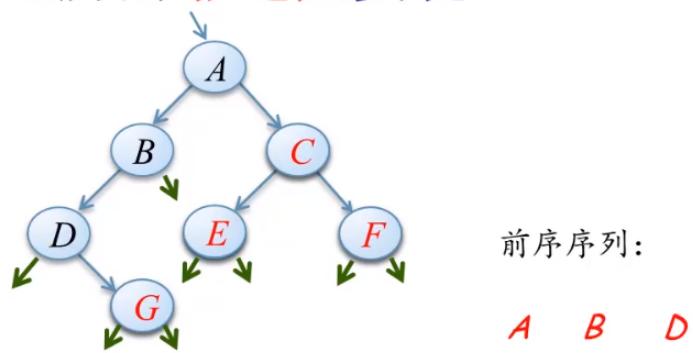
A



前序遍历的非递归实现

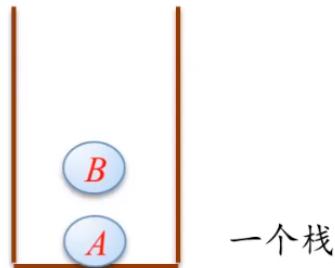
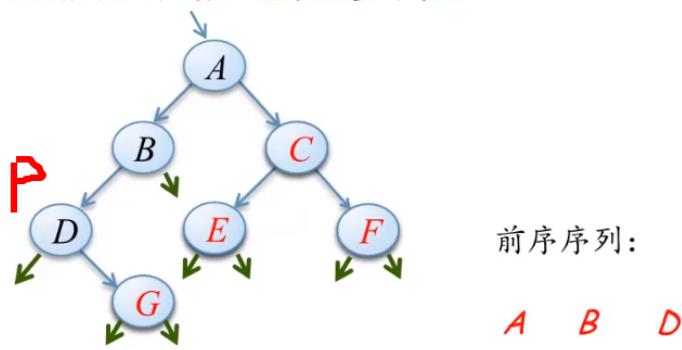


前序遍历的非递归实现

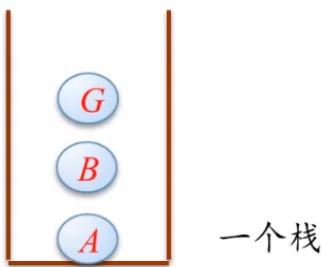
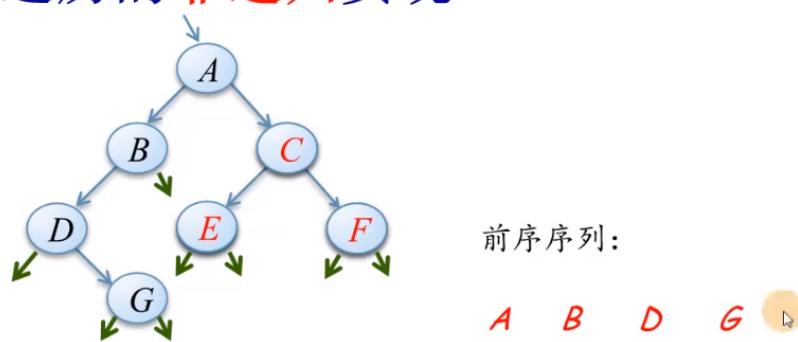


D的左子树为空，则栈顶POP，指针P的回到D位置，然后遍历右子树

前序遍历的非递归实现

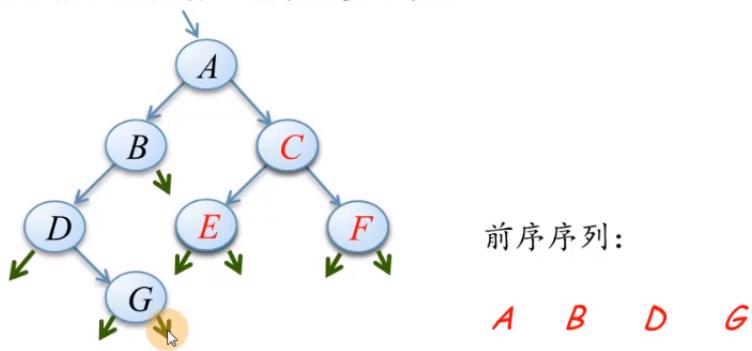


前序遍历的非递归实现

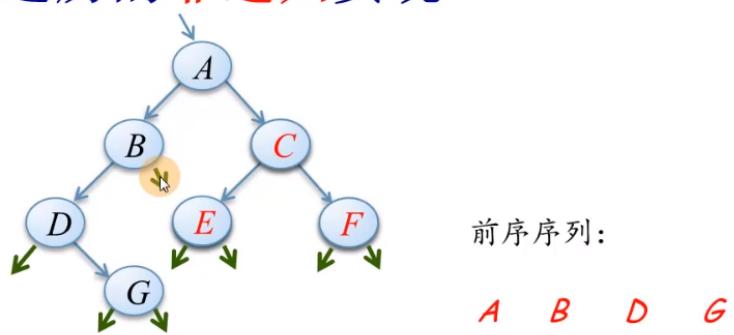


G的左节点为空，则栈顶的G出栈；P指针指向回到G节点，然后遍历右节点；又因为右节点为空，则栈顶元素B出栈；P指针指向B节点

前序遍历的非递归实现

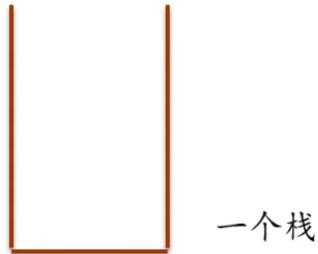
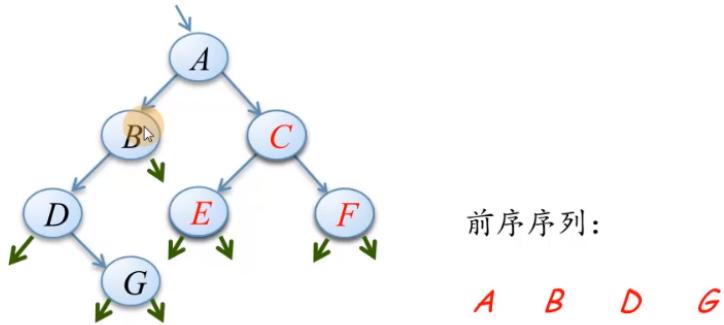


前序遍历的非递归实现



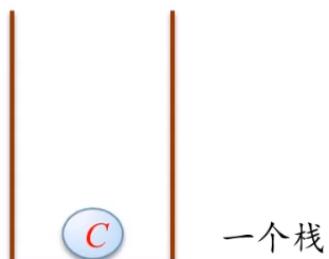
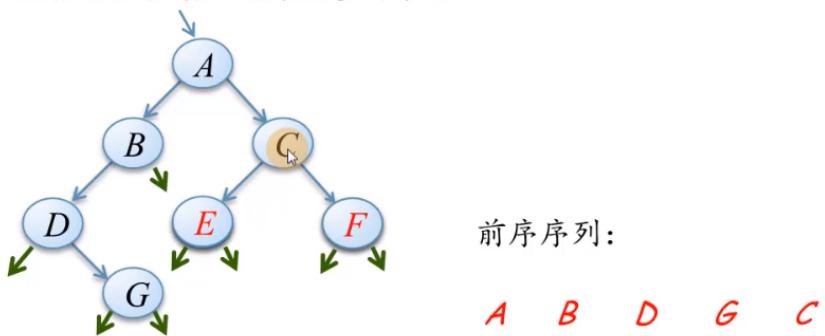
对于B元素来说,右子树为空,又要出栈;则A出栈,P指针回退到A节点

前序遍历的非递归实现

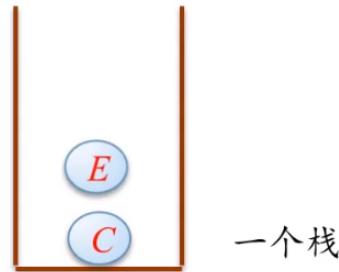
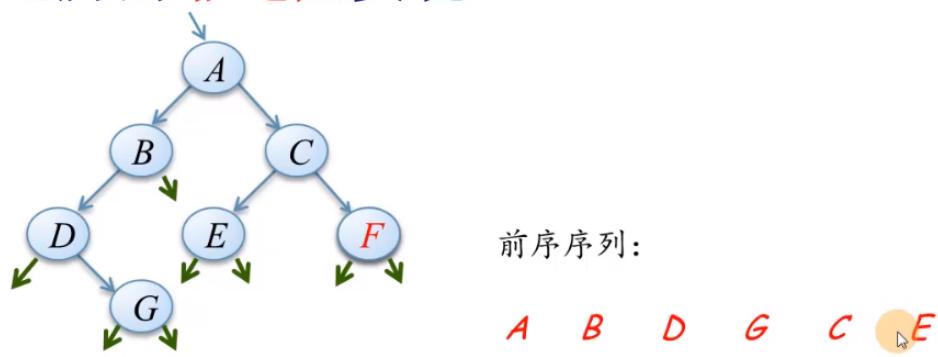


- 其他步骤

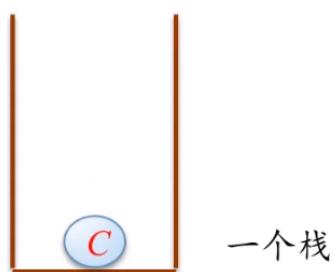
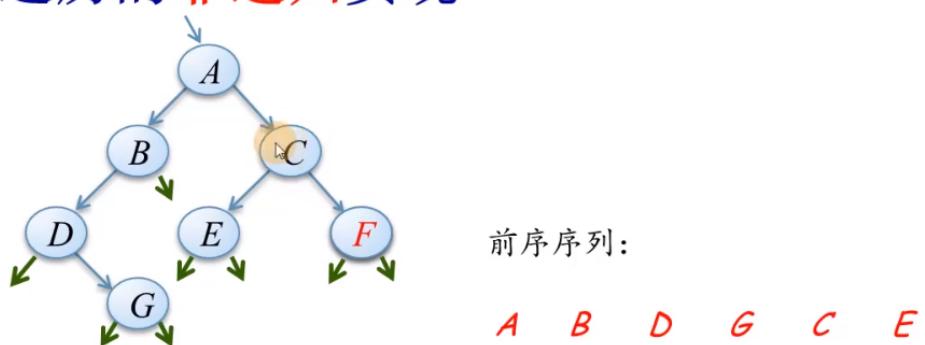
前序遍历的非递归实现



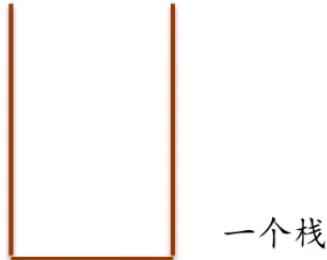
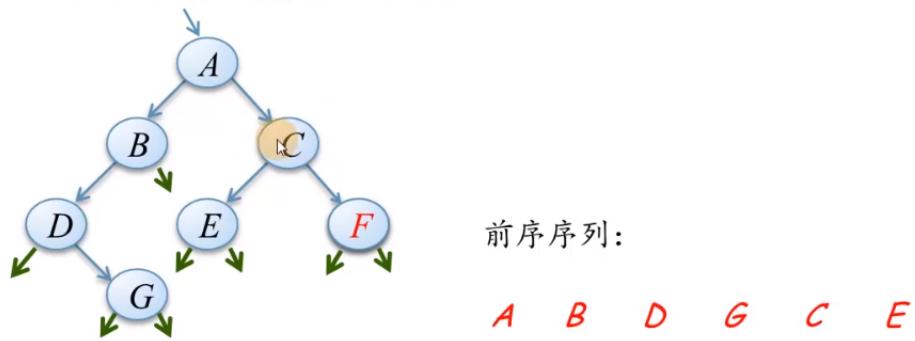
前序遍历的非递归实现



前序遍历的非递归实现

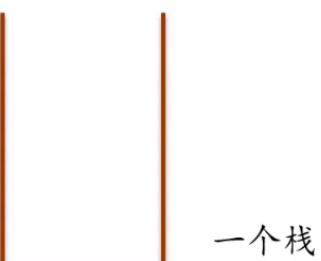
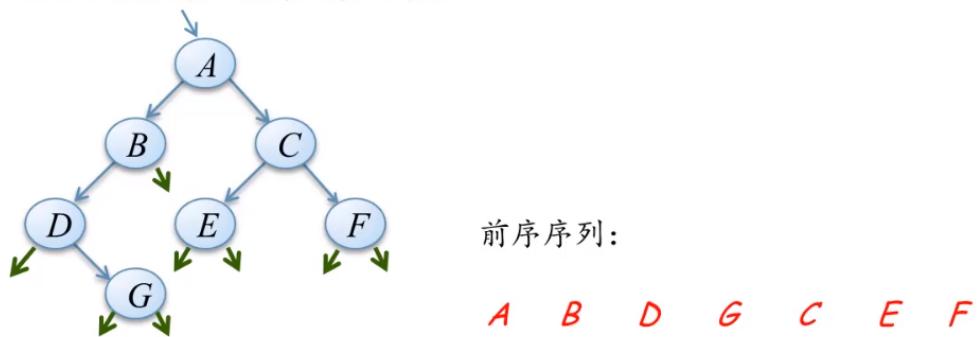


前序遍历的非递归实现



当访问到F的右节点时,栈中的元素也为空,此时循环终止

前序遍历的非递归实现



栈空 且 $p=NULL$

前序遍历完毕



```
//非递归先序遍历
void PreOrder1(Tree t)
{
    Tree p = t;
    Sqstack s;
    InitStack(s);

    while (p || !StackEmpty(s))
    {
        // 不为空遍历左子树
        if (p)
        {
```

```

        printf("%c", p->data); //输出根节点
        Push(s, p); //根节点进栈,方便左子树没有时返回
        p = p->lchild; //遍历左子树
    }
    // 为空遍历右子树
    else
    {
        Pop(s, p); //回退
        p = p->rchild;
    }
}
}

```

2.中序

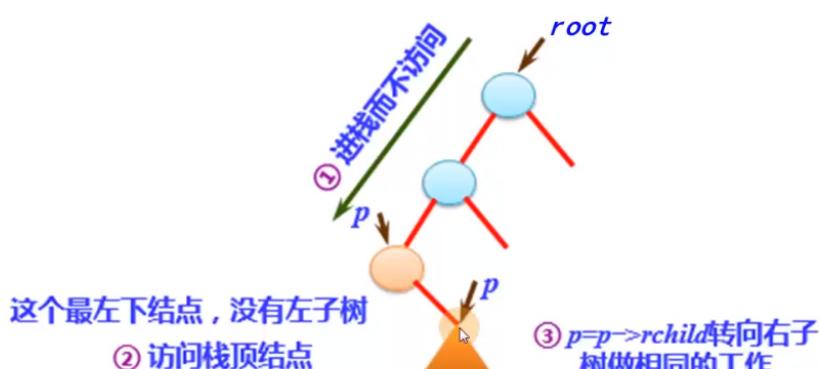
中序遍历——非递归算法

在二叉树的中序遍历中，访问结点的操作发生在该结点的左子树遍历完毕并准备遍历右子树时，所以，在遍历过程中遇到某结点时并不能立即访问它，而是将它压栈，等到它的左子树遍历完毕后，再从栈中弹出并访问之。中序遍历的非递归算法只需将前序遍历的非递归算法中的输出语句visit(bt->data)移到p = s.pop();之后即可。

中序遍历——非递归算法

在先序遍历非递归算法的基础上改进而来的

p 用于结点遍历，初始时 $p=root$, 当 $p=NULL$ 并且栈为空结束

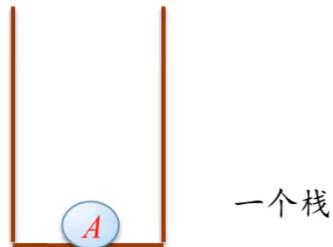
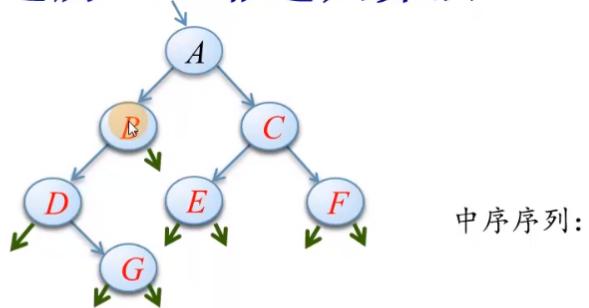


- 栈中结点均没有访问
- p 指向刚刚出栈结点的右子树

- 过程演示

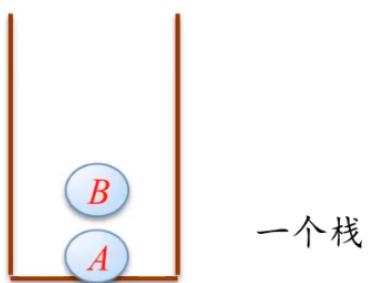
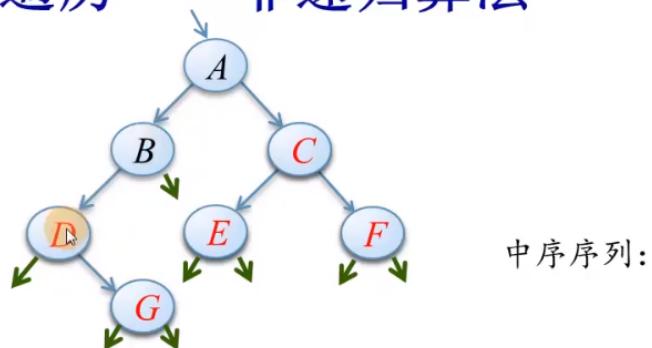
1. 根节点进栈但是不访问,遍历左子树

中序遍历——非递归算法



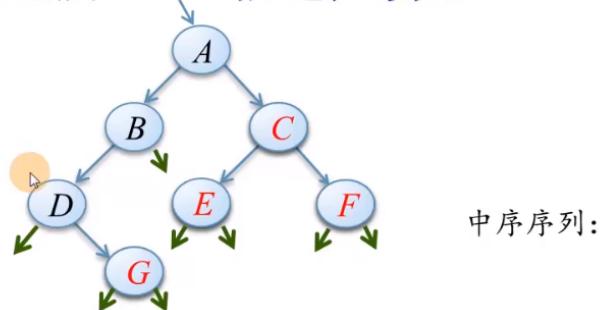
2.B节点入栈,但是不访问,遍历左子树

中序遍历——非递归算法

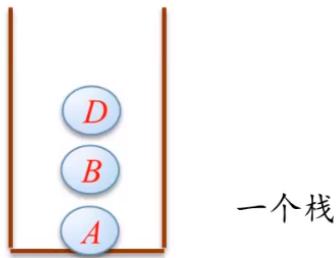


3.D节点进栈, 访问左子树为空

中序遍历——非递归算法

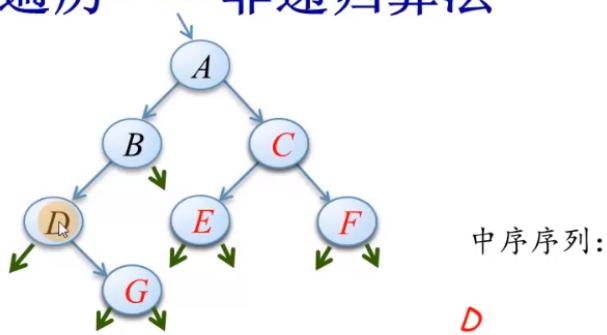


中序序列:



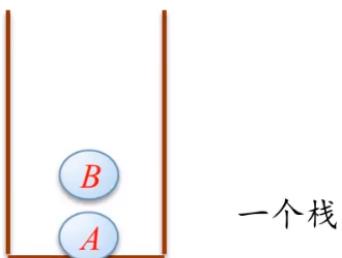
4.D出栈，输出D节点；P指针指向D节点，开始访问右子树

中序遍历——非递归算法



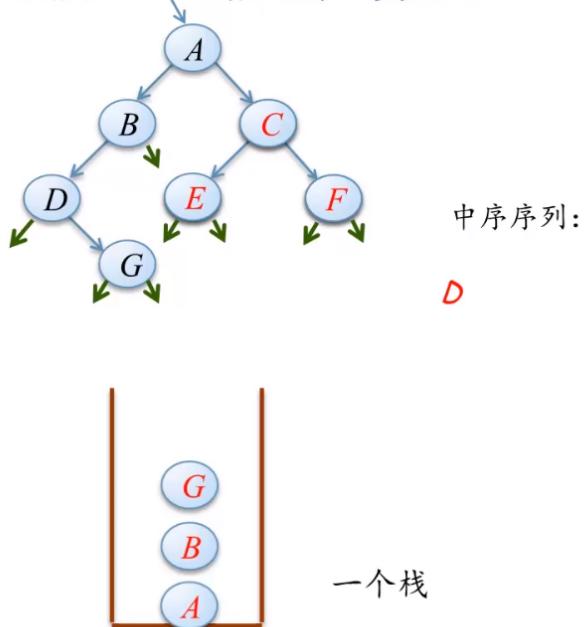
中序序列:

D

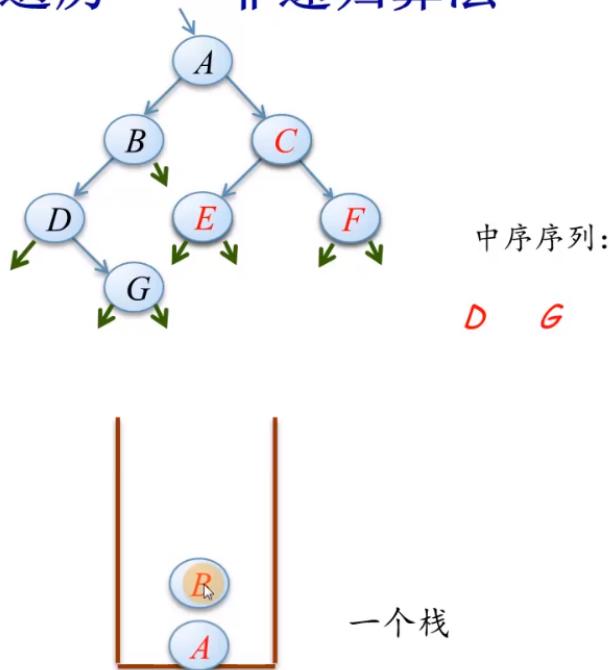


5.G节点入栈，访问左子树，为空，则G节点出栈，P回退到G，访问右节点

中序遍历——非递归算法

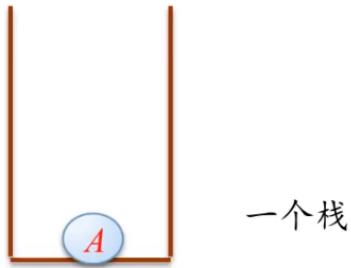
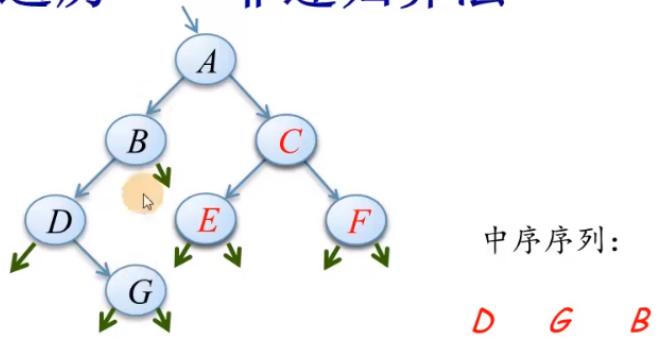


中序遍历——非递归算法



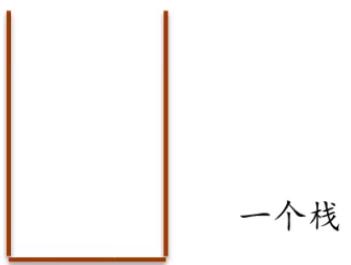
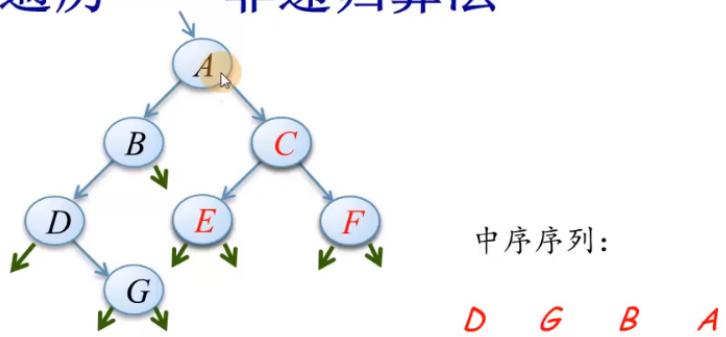
6.右子树为空，栈顶元素出栈，P回退到B节点，输出B节点

中序遍历——非递归算法



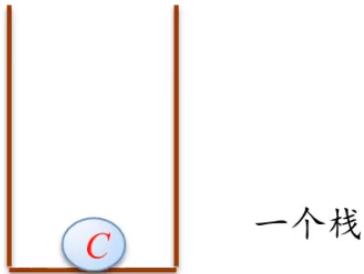
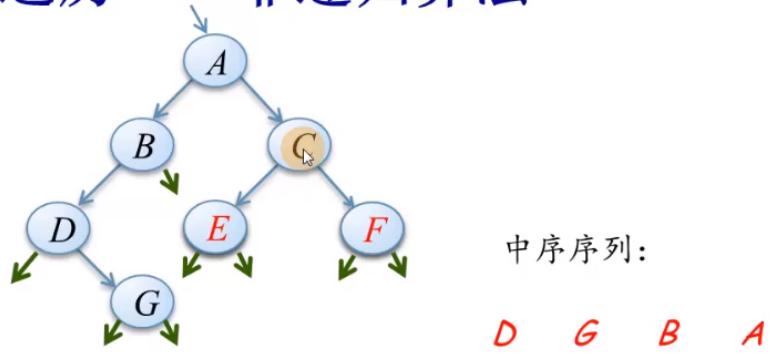
7.访问右子树，也为空，则A出栈，P指针指向A，输出A节点

中序遍历——非递归算法

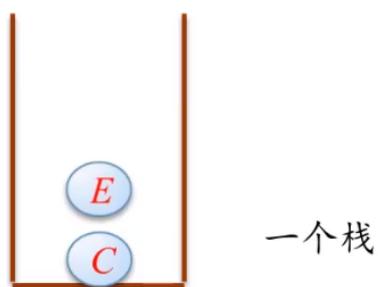
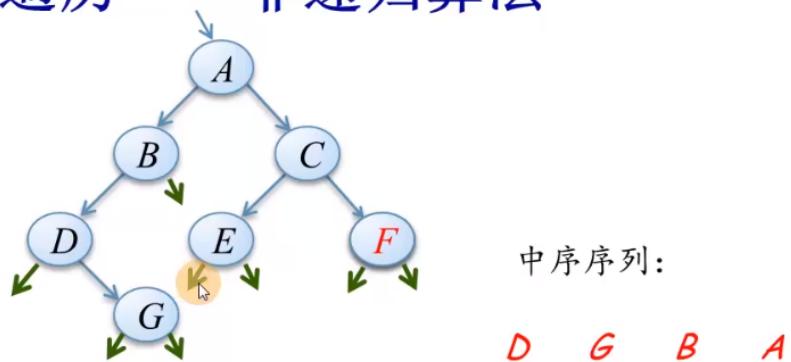


- 其他步骤

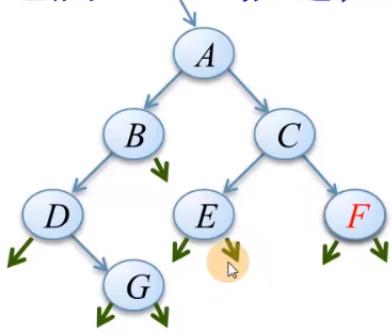
中序遍历——非递归算法



中序遍历——非递归算法

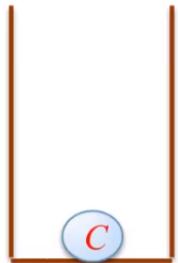


中序遍历——非递归算法



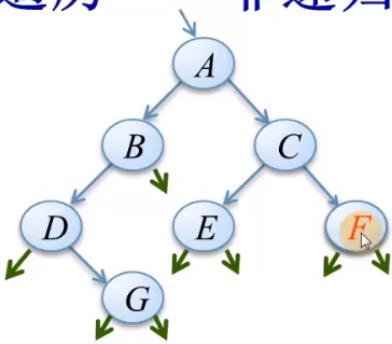
中序序列:

D G B A E



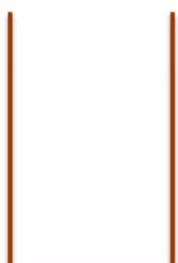
一个栈

中序遍历——非递归算法



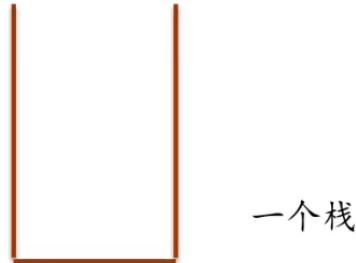
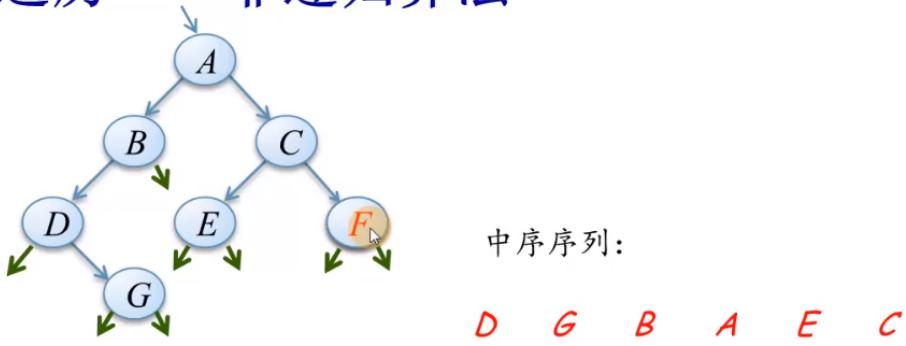
中序序列:

D G B A E C

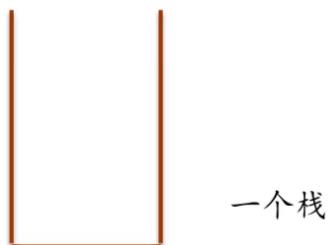


一个栈

中序遍历——非递归算法

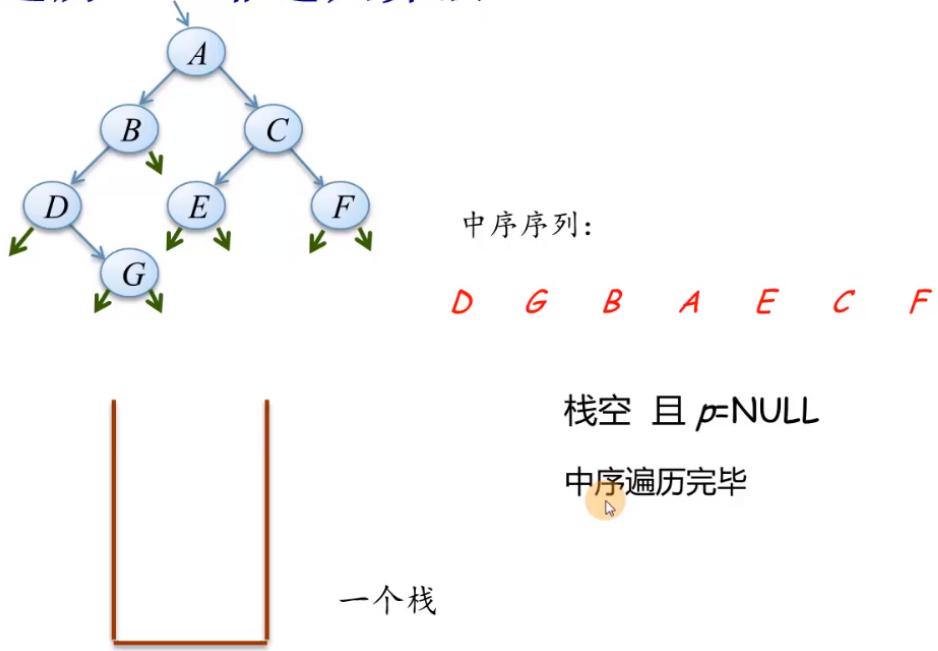


中序遍历——非递归算法



访问到F的右子树时，此时栈为空，并且P为空，循环停止，遍历结束

中序遍历——非递归算法



```
//非递归中序遍历
void InOrder1(Tree t)
{
    Tree p = t;
    Sqstack s;
    InitStack(s);

    while (p || !StackEmpty(s))
    {
        if (p)
        {
            Push(s, p);
            p = p->lchild;
        }
        else
        {
            Pop(s, p);
            printf("%c", p->data); //和先序的区别
            p = p->rchild;
        }
    }
}
```

3.后序

后序遍历首先访问左子树，然后退栈，访问右子树，进栈，退栈，用Debug走一遍

后序遍历——非递归算法

在后序遍历过程中，结点要入两次栈，出两次栈：

- (1) 第一次出栈：只遍历完左子树，该结点不出栈，利用栈顶结点找到它的右子树，准备遍历它的右子树；
 - (2) 第二次出栈：遍历完右子树，将该结点出栈，并访问它。
- 因此，为了区别同一个结点的两次出栈，设置标志**flag**。

11.二叉树层序遍历算法

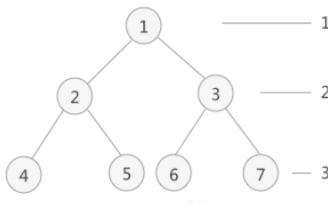


图1 二叉树

层次遍历的实现过程

例如，层次遍历图 1 中的二叉树：

- 首先，根结点 1 入队；
- 根结点 1 出队，出队的同时，将左孩子 2 和右孩子 3 分别入队；
- 队头结点 2 出队，出队的同时，将结点 2 的左孩子 4 和右孩子 5 依次入队；
- 队头结点 3 出队，出队的同时，将结点 3 的左孩子 6 和右孩子 7 依次入队；
- 不断地循环，直至队列内为空。