第 9 讲

动态数据组织

主要内容

9.1 动态内存管理

9.2 链表

9.3 栈

9.4 队列

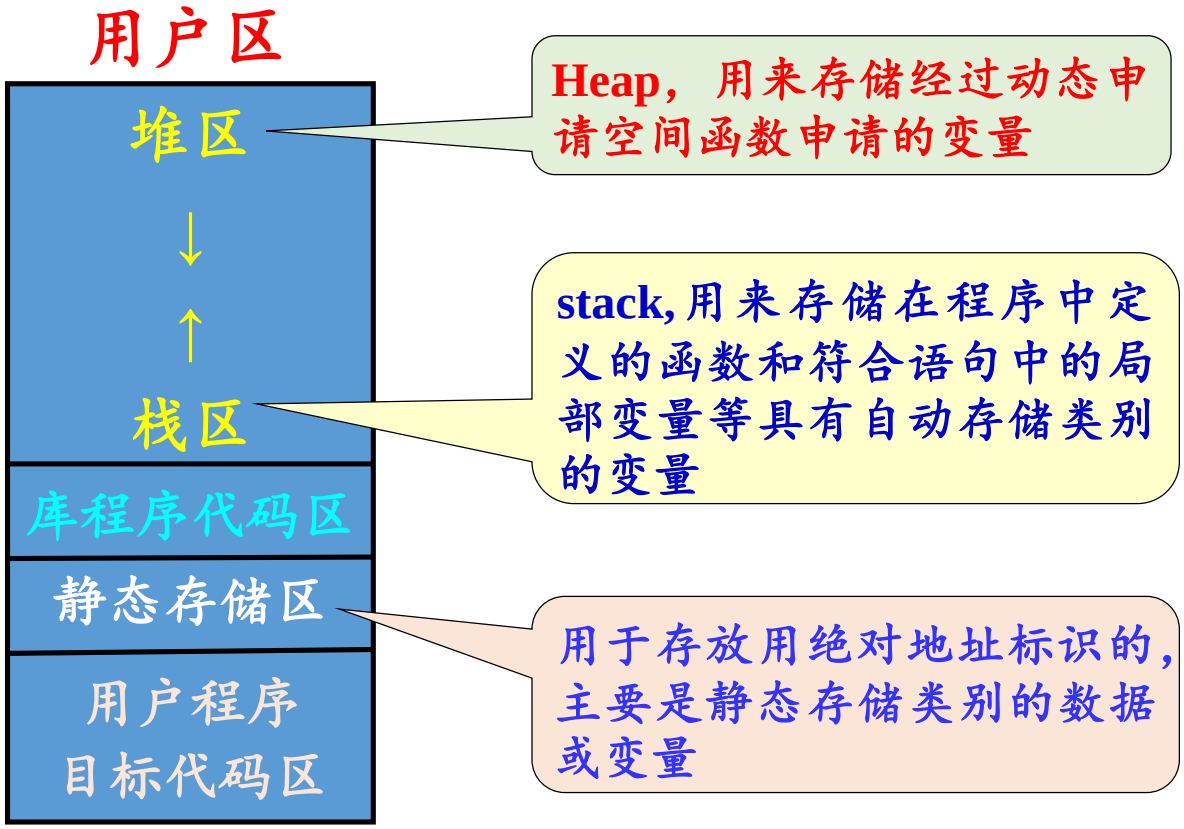
9.1 动态内存管理

静态数据结构 与 动态数据结构

- **静态数据结构**
  - ✓ 如数组，在程序执行时大小是固定的，必须用常量来定义，系统为其分配一段连续的内存空间
  - ✓ 使用静态数据结构时，要求事先了解数据集的大小，以免内存空间不够用或浪费

动态数据结构

- ✓ 在程序执行过程中，可以根据需要进行收缩或扩展
- ✓ 内存按需分配或释放，即进行内存动态管理
- ✓ 数据不一定存储在连续的内存空间，需要使用指针将数据连接起来
- ✓ 动态数据结构：链表、栈、队列


用户区
堆区
↓
↑
栈区
库程序代码区
静态存储区
用户程序
目标代码区
Heap，用来存储经过动态申请空间函数申请的变量
stack,用来存储在程序中定义的函数和符合语句中的局部变量等具有自动存储类别的变量
用于存放用绝对地址标识的，主要是静态存储类别的数据或变量

动态内存分配

- Dynamic Memory Allocation
- 一般的内存分配工作是在编译阶段进行。动态内存分配允许程序员在程序执行过程中进行内存分配
  - ✓ malloc( ) //分配一定字节的连续内存空间
  - ✓ calloc( ) //分配若干个具有固定字节的连续内存空间
  - ✓ realloc( )//修改已分配的内存区的大小
  - ✓ free( ) //释放已分配的内存区

函数原型	**void *malloc(unsigned long size);**
函数功能	向内存申请分配size字节的连续空间
参数	size—申请的内存空间的字节数
返回值	成　功：所分配的内存空间的首地址 不成功：NULL
头文件	#include<stdlib.h>
说明：	• 所分配空间位于堆中，按指定大小分配 • 不再使用该内存空间时，使用free函数释放 应用举例：int *pi; pi=malloc(50*sizeof(int)); //为50个整数申请内存空间，并使pi指向该空间

函数原型	**void *calloc(unsigned n, unsigned size);**
函数功能	向内存申请n个长度为size字节的连续空间
参数	n —数据项的个数 size —每个数据项的字节数
返回值	成　功：所分配的内存空间的首地址 不成功：NULL
头文件	#include<stdlib.h>
说明：	• 所分配空间位于堆中，每个字节都被置为0 • 不再使用该内存空间时，使用free函数释放 应用举例：int *pi; pi=calloc(50, sizeof(int)); //申请50个int类型的内存空间，使pi指向该空间

函数原型	**void *realloc(void *p, unsigned size);**
函数功能	将p所指向的已经分配的内存区大小改为size
参数	p—指向要修改大小的内存区的指针 size—修改后的字节数
返回值	成　功：新分配的内存空间的首地址 不成功：NULL
头文件	#include<stdlib.h>

说明：	• size可以比原来的内存区间扩大或缩小
	应用举例：int *pi; pi=malloc(50*sizeof(int));
	pi=realloc(pi,10*sizeof(int));
	//将空间缩小至存放10个整数

函数原型	**void free(void *p);**
函数功能	释放p所指向的内存区
参数	p—指向要释放内存空间的指针
返回值	无
头文件	#include<stdlib.h>
说明：	• 拟释放的内存是由malloc或calloc函数申请空间时返回的地址 • 所释放的空间可由系统重新分配 应用举例：free(p);

9.2 链表

- 一种动态数据结构
- 若干个结点由指针串在一起构成
- 结点数目无须事先指定，可以临时生成，每个结点有自己的存储空间，结点间的存储空间无须连续
- 插入和删除结点时方便，无须移动大批数据，只需修改指针的指向

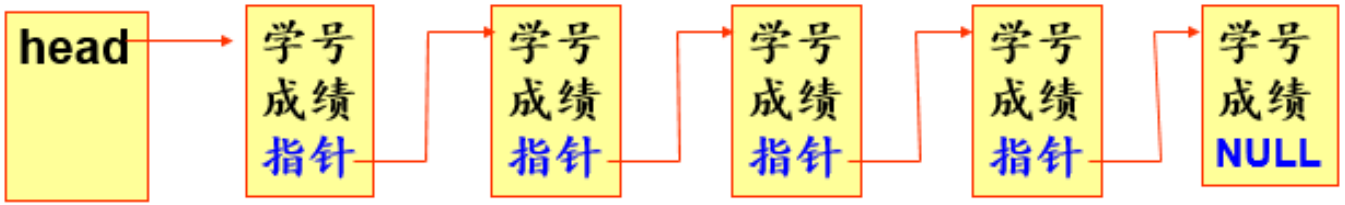
例：选择合适的数据结构来存放一批学生的学号及考试成绩，以便进一步处理。

- 由于学生人数未知，用静态数组不合适
- 用链表处理较恰当。

12

用链表处理该问题的基本思路：

将各学生的数据进行离散存放，来一个学生就分配一小块内存（结点）。并将各结点用指针依次连接起来——链表



- 每结点应包含下一结点的开始地址
- 最后一个结点中的指针为空
- 链头指针指向第一个结点,是访问链表的重要依据

这样的链表称单向链表

```
//每个结点可用如下结构描述：
struct  Student
{
   int  num;            // 学号
   int  score;          // 成绩
   struct  Student  *next;   // 下一结点的地址
};
```

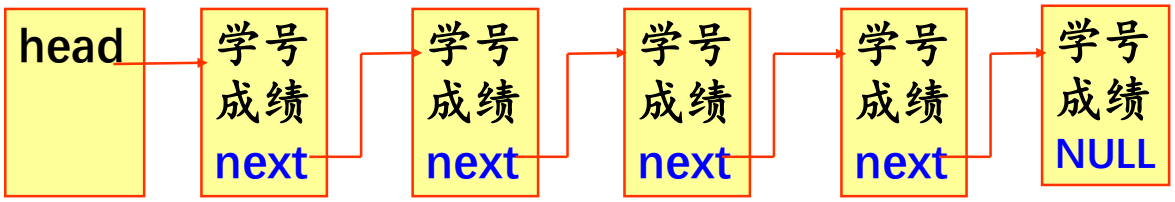
创建链表

①输入一个学生的数据

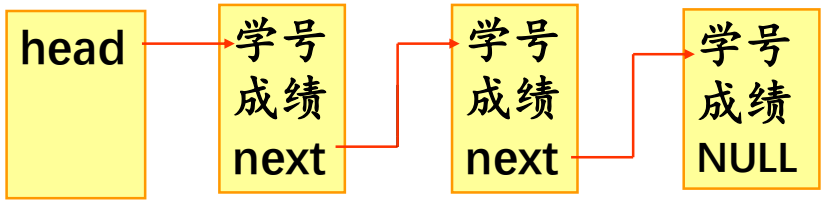
②分配结点空间，数据存入

③将该结点的首地址赋给上一结点的next，若该结点是第一个结点，则赋给头指针

④将该结点的next置为空（NULL），表示该结点为当前的最后结点



```
struct  Student   *creat ( )
{
   struct  Student  st, *p0=NULL, *p, *head=NULL;
   while(1)
    {  scanf("%d%d", &st.num, &st.score);
       if(st.num<0) break;
       p=(struct Student *) malloc(sizeof(struct  Student));
       *p=st;
       p->next=NULL;
       if(p0==NULL)  head=p;     //p0为前一结点的指针
       else  p0->next=p;
       p0=p;
    }
   return head;
}
```



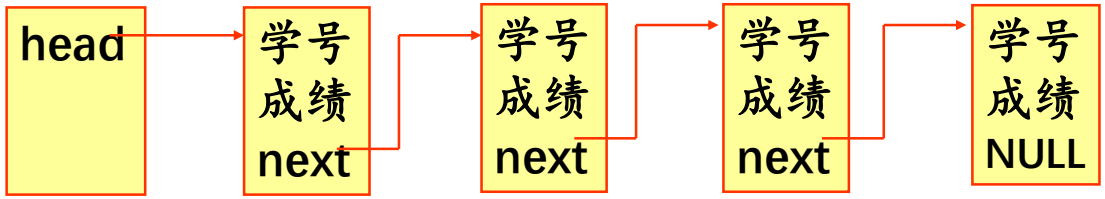
遍历链表

以输出为例

① 通过头指针找到第一个结点.

② 输出当前结点的内容，并通过next找到后继结点，…，直到next为空

```
void output(struct  Student  *head)
{
   struct  Student   *p=head;
    while(p)
    {
       printf("\n %d %d", p->num, p->score );
       p=p->next;
    }
}
```



删除结点

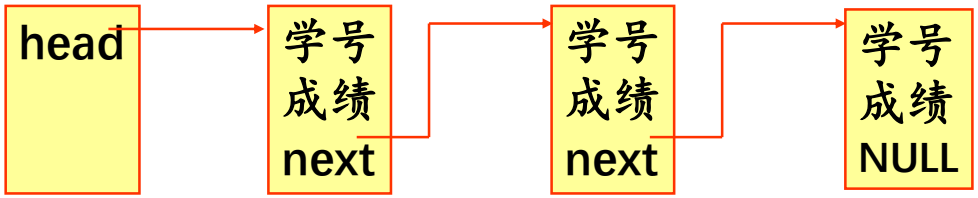
①按链表的访问方法找到相应结点。

②若该结点是第一个结点，则将后继结点指针赋给头指针；

若该结点是最后一个结点，则将前缀结点的next置为空；

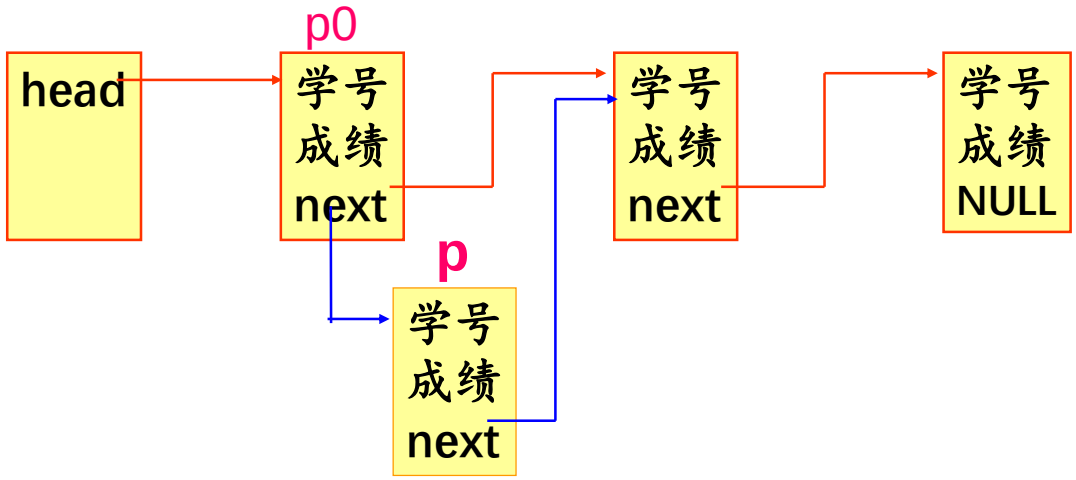
若该结点是中间结点，则将后继结点指针赋给前缀结点的next。

③释放该结点所占的内存单元。



```
struct  Student   *delete(struct  Student  *head, int num)
{  struct  Student  *p=head, *p0=NULL;
   while(p)
   {   if(p->num==num)  //假定要删除某一指定学号的结点
       {     if(p==head)   head=p->next;
             else if(p->next==NULL)   p0->next=NULL;
                  else   p0->next=p->next;
             free(p);    break;
       }
      else {p0=p; p=p->next;}
   }
  return head;
}
```

插入结点

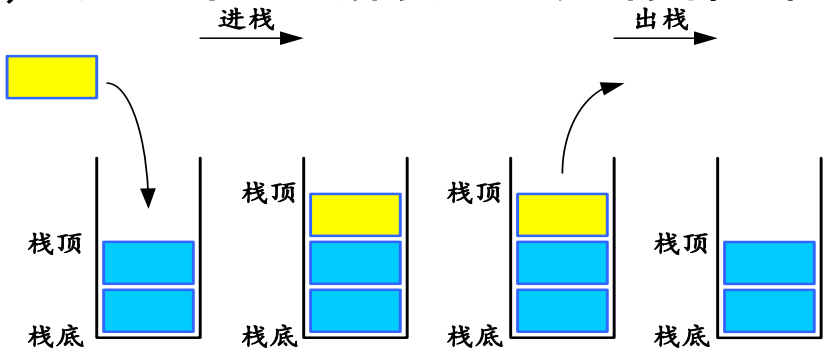


假定将结点p插入到结点p0的后面，则插入操作的关键为：

p->next=p0->next; p0->next=p;

9.3 栈

- 一种操作受限的线性表，仅允许在表的一端（栈顶）进行插入和删除操作，另一端为栈底
- 插入新元素到栈顶元素的上面，称作进栈、入栈或压栈；从一个栈删除元素又称作出栈或退栈



栈的操作：

- 压栈
- 出栈
- 计算栈长度
- 输出栈内容

```
//可用链表实现栈结构：
struct node
{
   int  val;                    // 存放数据
   struct node *next;    // 下一结点的地址
};

struct node *head=NULL;
```

```
struct node *create_node(int val)     //生成一个结点
{
    struct  node  *p=(struct  node  *)malloc(sizeof(struct
                                node));
    p->val=val;
    p->next=NULL;
    return p;
}
```

```
struct node *push(int val)      //压栈
{
    struct node *p=create_node(val);
    p->next=head;
    head=p;
    return head;
}
```

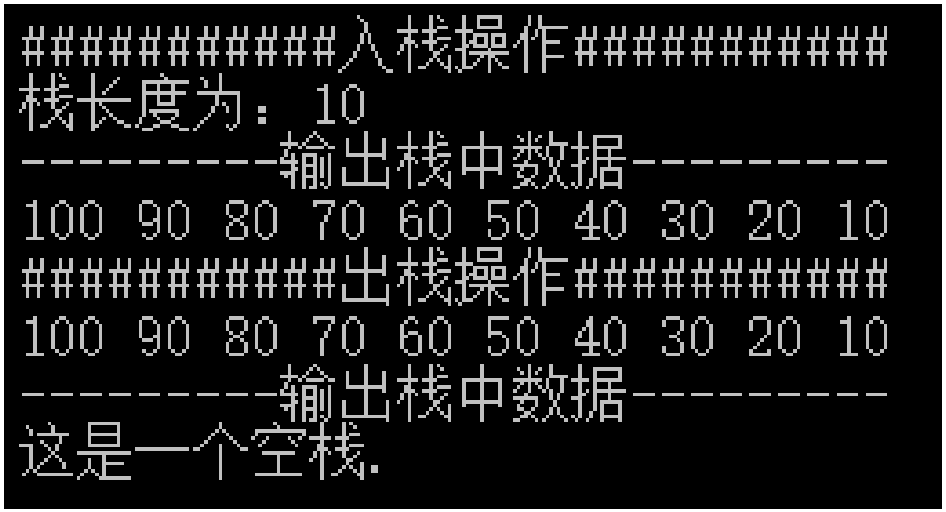
```
int pop(struct node *link)  //出栈，返回值为栈顶数据
{
    int val;
    struct node *p=head;
    val=p->val;
    head=head->next;
    free(p);
    return val;
}
```

```
int length_stack(struct node *link)   //计算栈长度
{
    int count=0;
    while(link){  count++;  link=link->next;    }
    return count;
}

void print_stack( )     //输出栈中的内容
{   if(link==NULL) printf(" 这是一个空栈.\n");
    else while(link){   printf(" %d", link->val);
                        link=link->next;  }
}
```

例9-1：

```
void main( )
{  int i, stackSize;
   printf("\n ############入栈操作############\n");
   for(i=1;i<=MAXSIZE;i++)   push(i*10);
   stackSize=length_stack(head);
   printf(" 栈长度为：%d\n", stackSize);
   printf(" ---------输出栈中数据---------\n");
   print_stack(head);
   printf("\n ############出栈操作############\n");
   for(i=0;i<stackSize;i++)    printf(" %d", pop(head));
   printf("\n ---------输出栈中数据---------\n");
   print_stack(head);
}
```


############入栈操作############
栈长度为： 10
----------输出栈中数据----------
100 90 80 70 60 50 40 30 20 10
############出栈操作############
100 90 80 70 60 50 40 30 20 10
----------输出栈中数据----------
这是一个空栈.

例9-2： 编写程序，输入两个个正整数n和m，实现将n转换为m进制的功能，输出转换后的数据。

- 分析

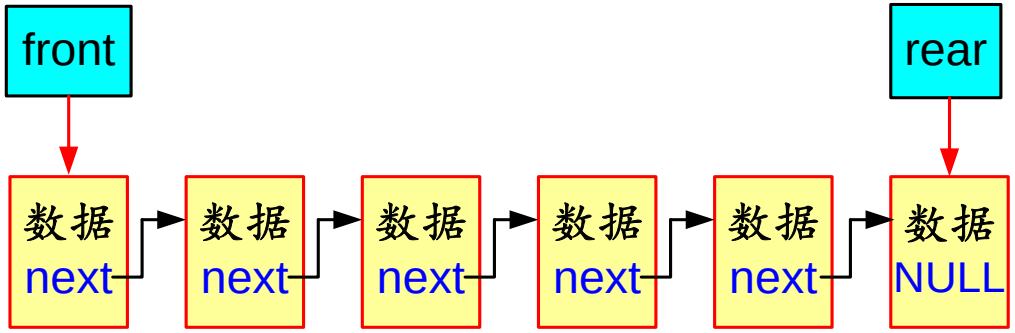
本题可用数组方法实现；本题用链表方法实现

```
void main( )
{   int n, m, nn;
    printf("请输入拟转换的数n及进制数m：");
    scanf("%d%d", &n, &m);
    nn=n;
    while(nn)
    {    push(nn%m);        nn=nn/m;    }
    printf("%d可以转换为%d进制", n, m);
    print_stack( );
}
```

```
请输入拟转换的数n及进制数m：159 7
159可以转换为7进制315
```

9.4 队列

- 一种操作受限制的特殊线性表，也叫**FIFO**结构
- 数据项从表的一端（rear，队尾）加入，而在表的另一端（front，队首）移除。因此，一个队列需要两个指针，分别指向队首和队尾



队列的操作：

- 入队（排在队尾）
- 出队（从队首删一项）

```
//可用链表实现队列结构：
struct queue
{
   int  val;                    // 存放数据
   struct queue *next;    // 下一结点的地址
};

struct queue *front=NULL, *rear=NULL;
```

```
void Inqueue(int x)    //入队
{
    struct queue *p;
    p=(struct queue *)malloc(sizeof(struct queue));
    p->val=x;
    p->next=NULL;
    if(rear==NULL) rear=front=p;
    else   {  rear->next=p;    rear=p;   }
}
```

```
int Outqueue( )    //出队
{
    int val;
    struct queue *p;
    if(front==NULL) { printf(" 队列空.\n");  val=-999;}
    else {   val=front->val;   p=front;   front=front->next;
                if(front==NULL)  rear=NULL;
                free(p);
          }
    return val;
}
```

例9-3：

```
void main( )
{
    int i;
    printf("\n 1. 入队操作 \n");
    for(i=1; i<=MAXSIZE; i++)  Inqueue(i*10);
    printf(" 2. 出队时输出队列内容\n");
    for(i=0; i<MAXSIZE/2; i++)  printf(" %d", Outqueue());
    printf("\n");
    for(i=1; i<MAXSIZE; i++)  Inqueue(i*1000);
    for(i=1; i<MAXSIZE+MAXSIZE/2; i++) printf(" %d", Outqueue());
    printf("\n");
}
```

```
1. 入队操作
2. 出队时输出队列内容
10 20 30 40 50
60 70 80 90 100 1000 2000 3000 4000 5000 6000 7000 8000 9000
```