数据结构 *Data Structure*

- 数据结构 *Data Structure*
- 结构*Struct*
  - 结构声明和变量定义
  - `-> .`和其他运算符优先级
  - 向函数传递参数
  - 结构赋值
  - 结构数组指针
- 链表*Linked List*
  - 基本链表类型
    - 单向单头链表
    - 单向双头链表
    - 单向有哨兵
  - 基本操作及其复杂度
    - 创建链表 - O(1)
    - 头插法 - O(1)
    - 尾插法 - 单头O(n),双头O(1)
    - 按值删除所有结点 - O(n)
    - 按值/按位置搜索某一结点 (Linear) - O(n)
    - 销毁 - O(n)
  - 简单程序及其复杂度
    - 奇偶结点重组*-19A*
    - 分离奇偶值结点
    - 链表实现Merge - 时间O(n)，空间O(1)
    - 链表逆置 - O(n)
    - 在递增链表中插入新结点 - O(n)
    - 用单向链表完成多项式计算
    - 循环链表之猴子选大王
- 队列
  - 队列的基本概念
  - 数据结构
  - 循环队列
    - CreateQueue
    - EmptyQueue
    - FullQueue
    - EnQueue
    - DeQueue
- 栈*Stack*
  - 栈的基本概念
  - 数据结构
  - Pop
  - Push

结构*Struct*

结构声明和变量定义

- 结构体类型不占内存，定义变量占内存

在c语言中，不允许有常量的数据类型是（结构 ）

若程序有以下的说明和定义：

```
struct abc
{ int x;char y; } //没加;
 struct abc s1,s2;
```

则会发生的情况是（）

- 嵌套结构

如果结构变量s中的生日是“1984年11月11日”，下列对其生日的正确赋值是（）。

```
struct student
{
  int no;
  char name[20];
  char sex;
  struct{
    int year;
    int month;
    int day;
  }birth;
};
struct student s;
```

`-> .`和其他运算符优先级

- 单目运算符 `[] () . ->`优先级最高，这四个结合律左到右
- 其他单目右到左

For the following declarations of structure and variables, the correct description of the expression `*p->str++;` is __.

```
struct {
    int no;
    char *str;
} a={1,"abc"}, *p=&a;
```

```
++ acts on the pointer str
```

向函数传递参数

- 可以传递整个结构
- 可以传递结构指针
- 可以传递结构成员

以下`scanf`函数调用语句中不正确的是__。

```
struct pupil {
    char name[20];
    int age;
    int sex;
} pup[5], *p=pup;
```

A. `scanf("%s", pup[0].name);`　　*数组名本身是一指针*

B. `scanf("%d", &pup[0].age);`

C. `scanf("%d", p->age);`　　`p->age` *是一个int*

D. `scanf("%d", &(p->sex));`

> `scanf(format,指针)`

结构赋值

- 可以两个结构赋值
- 可以结构内成员赋值
- 注意数组和指针的区别

For the following declarations, assignment expression __ is not correct.

```
struct Student {
    long num;
    char name[20];
} st1, st2={101, "Tom"}, *p=&st1;
```

```
A.  st1 = st2
```

```
B.  p->name = st2.name √(数组不等于指针，不能直接复制)
```

```
C.  p->num = st2.num
```

```
D.  *p=st2
```

结构数组指针

```
The value of expression *((int *)(p+1)+2) is __.

static struct {
    int x, y[3];
} a[3] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}}, *p;
p = a+1;
```

```
After executing the following code fragment, the value of variable z is ______.

static struct{ int x, y[3];
} a[3]={{0},{5,6,7},{10,12}}, *p=a+3; int z;
z=*((int *)(p-1)-3);
```

链表*Linked List*

基本链表类型

单向单头链表

- 数据结构

```
typedef struct _Node {
  int value;
  struct _Node *next;
} Node;

typedef struct {
  Node *head;//仅有头指针
} List;

List list;
List *plist = &list;
```

- 头插法

```c
void insert_head (List *plist,int x) {
    Node *p = (Node *) malloc(sizeof(Node));
    p->value = x;
    p->next = plist->head;
    plist->head = p;
}
```

- 尾插法

```c
//appendtail:Boundary-空表
void append_tail (List *plist,int x) {
    Node *tail = (Node *)malloc(sizeof(struct _Node));
    tail->value = x;
    tail->next = NULL;
    if (plist->head) {
        Node *p = plist->head;
        for (;p->next;p=p->next) ;
        p->next = tail;
    }
    else {
        plist->head = tail;
    }
}
```

- 按值删除所有结点

```c
void list_remove(List *list, int value) {
    Node *p=list->head,*q=list->head;
    while(p) {
        if (p->value == value) {
            if (list->head == p) {//删除头结点
                list->head = q = p->next;
                free(p);
                p = q;
            } else {//删除中间结点
                q->next = p->next;
                free(p);
                p = q->next;
            }
        } else {//不删除结点
            q = p;
            p = p->next;
        }
    }
}
```

- 遍历*iterate*

查找实质是遍历

```
void list_iterate(List *list, void (*func)(int v)) {
    for (Node*p = list->head;p;p=p->next) {
        func(p->data);
    }
}
```

- 销毁

```
void clear (List *plist) {
    for (Node *p = plist->head,*q = NULL;p;p = q) {
        q = p->next;
        free(p);
    }
}
```

单向双头链表

- 数据结构

```
typedef struct _node Node;
typedef struct {
    Node *head;
    Node *tail;//比单头链表多尾指针
} List;
```

- 创建链表（多尾指针）

```
List list_create() {
    List list;
    list.head = list.tail = NULL; //头尾指针置为NULL
    return list;
}
```

- 尾插法（有尾指针，尾插方便许多）

```
void list_append(List *list, int v) {
    Node *p = (Node *)malloc(sizeof(Node));
    p->data = v,p->next = NULL;//创建并初始化新结点
    if (list->tail) {//情况1:链表非空
        list->tail->next = p;//变更尾指针位置
        list->tail = p;
    } else {//情况2:空链表
        list->head = list->tail = p;//变更头尾指针位置
    }
}
```

- 头插法（空链表时多维护尾指针）

```
void list_insert(List *list, int v) {
    Node *p = (Node *)malloc(sizeof(Node));
    p->data = v,p->next = NULL; //创建并初始化新结点
    if (list->head) {//情况1:链表非空
        p->next = list->head;//变更头指针位置
        list->head = p;
    } else {//情况2:空链表
        list->head = list->tail = p; //变更头尾指针位置
    }
}
```

- 按值删除某结点（多维护尾指针。分两大类，四小种）

```
void list_remove(List *list, int v) {
if (list->head && list->head != list->tail) {//假定链表非空且至少有两个结点
/*以下这段代码实际上也可以放在for循环中，没必要单独讨论该情况*/
        if (list->head->data == v) {//情况1:如果要删除的结点是头结点
            Node *p = list->head;
            list->head = p->next;//改变头指针位置
            free(p);
            return;
        }
        for (Node *p = list->head->next,*q = list->head;p;q = p,p = p->next) {
            if (p->data == v) {
                if (p == list->tail) {//情况2:如果要删除尾结点
                    list->tail = q;
                    q->next = NULL;//这里很重要，使尾结点后继为NULL
                    free(p);
                } else {
                    q->next = p->next;//情况3:中间结点
                    free(p);
                    p = q->next;
                }
                break;
            }
```

```
        }
    }
}
```

单向有哨兵

- 创建哨兵链表（头结点）

```
void create_head(List *plist) {
    Node *p = (Node*)malloc(sizeof(Node));
    p->value=0,p->next=NULL;
    plist->head = p;
}
```

- 头插法（实际的头指针是`head->next`）

```
void insert_head(List *plist,int x) {
    Node *p = (Node*) malloc(sizeof(Node));
    p->value = x,p->next = NULL;
    p->next = plist->head->next;
    plist->head->next = p;
    /*比较一下无头结点的写法
    p->next = plist->head;
    plist->head = p;
    */
}
```

- 尾插法（不用考虑空表情况）

```
void append_tail(List *plist,int x) {
    Node *tail = (Node*)malloc(sizeof(Node));
    tail->value=x,tail->next=NULL;
    Node*p=plist->head;
    for (;p->next;p=p->next) ;
    p->next = tail;
    /*比较无头结点，空表头指针为空，需要单独考虑（而设置了哨兵后，即使是空表，头指针也不为空）
    if (plist->head) {
        Node *p = plist->head;
        for (;p->next;p=p->next) ;
        p->next = tail;
    }
    else {
        plist->head = tail;
    }
```

- 删除（不用单独考虑删除头指针情况）

```
void remove(List *plist,int x) {
    for (Node*p = plist->head,*q = plist->head->next;p;q = p,p = p->next) {
        if (p->value == x) {
            q->next = p->next;
            free(p);
        }
    }
}
```

基本操作及其复杂度

创建链表 - O(1)

```
List head;
head = NULL;
```

头插法 - O(1)

```
Node *p = (Node *)malloc(sizeof(struct Node));
p->data = val,p->next = NULL;//create a node

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

尾插法 - 单头O(n),双头O(1)

```
Node *p = (Node *)malloc(sizeof(struct Node));
p->data = val,p->next = NULL;

if (tail) {
  tail->next = p;
  tail = p;
}
else {
  head = tail = p;
}
```

按值删除所有结点 - O(n)

```
void list_remove(List *list, int value) {
    Node *p=list->head,*q=list->head;
    while(p) {
        if (p->value == value) {
            if (list->head == p) {//删除头结点
```

```
                list->head = q = p->next;
                free(p);
                p = q;
            } else {//删除中间结点
                q->next = p->next;
                free(p);
                p = q->next;
            }
        } else {//不删除结点
            q = p;
            p = p->next;
        }
    }
}
```

按值/按位置搜索某一结点 (Linear) - O(n)

```
int loc = 0;
for (Node *p=head;p;p = p->next) {
  if (p->data == x) {
    return loc;
  }
  loc++;
}
```

销毁 - O(n)

```
for (Node *p = head,*q;p;p = q){
  q = p->next;
  free(p);
}
```

- 注意`->`左边不能是`NULL`

简单程序及其复杂度

奇偶结点重组-*19A*

- 要求重排后`1-3-5-2-4`
- 要求空间复杂度为`O(1)`，即利用原有结点，至多创建了一个哨兵结点
- 已知`CreateNode(int data)`

```
Linklist Rearrange(Linklist head) {
  ListNode* current = head;
  Linklist even = CreateNode(0);
  ListNode* even_tail = even;
  ListNode* odd_tail = NULL;
```

```
  int even = 0;
  while (odd_tail) {
    if (!even) //current指向奇数
    {
      odd_tail = current;
    }
    else {
      even_tail->next = current;
      even_tail = current;//尾插法
      odd_tail->next = current->next;
    }
    current = current->next;
    even = 1 - even;
  }
  even_tail->next = NULL;
  current->next = even->next;//不是even--相当于一个哨兵结点
  return head;
}
```

分离奇偶值结点

- 空间复杂度O(1)，利用原结点
- 十分类似于上题

```
struct ListNode *getodd( struct ListNode **L ) {
    Node *odd = (Node*)malloc(sizeof(struct ListNode));
    odd->data = 0,odd->next = NULL;
    Node *odd_tail = odd;
    Node *cur = *L;
    Node *even_tail = NULL;
    while (cur) {
        if (cur->data%2) {
            odd_tail->next = cur;
            odd_tail = cur;
            if (cur == *L) {//判断第一个是否为奇数
                *L = (*L)->next;
            }
            else {
                even_tail->next = cur->next;
            }
        }
        else {
            even_tail = cur;
        }
        cur = cur->next;
    }
    return odd->next;//odd本身是哨兵结点
}
```

链表实现Merge - 时间O(n)，空间O(1)

- 数组merge，空间复杂度O(n),必须要新开辟`b[n]`
- 与数组显著不同的是，链表实现利用原结点，只新建了哨兵结点

```
typedef struct Node Node;
List Merge( List L1, List L2 ) {
    List merge = (List)malloc(sizeof(Node));
    merge->Data = 0,merge->Next = NULL;
    Node *merge_tail = merge;
    Node *tail1 = L1->Next,*tail2 = L2->Next;
    while (tail1 && tail2) {
        if (tail1->Data < tail2->Data) {
            merge_tail->Next = tail1;
            merge_tail = tail1;//尾插法
            tail1 = tail1->Next;
        } else {
            merge_tail->Next = tail2;
            merge_tail = tail2;//尾插法
            tail2 = tail2->Next;
        }
    }
    merge_tail->Next = tail1 ? tail1 : tail2;
    L1->Next = NULL,L2->Next = NULL;
    return merge;
}
```

链表逆置 - O(n)

- 利用头插法

```
typedef struct ListNode Node;
void insert_head(Node **head,int x) {
    Node *p = (Node*)malloc(sizeof(Node));
    p->data=x,p->next=NULL;
    p->next=*head;
    *head = p;
}
struct ListNode *reverse( struct ListNode *head ){
    Node *head2 = NULL;
    for (Node*p = head;p;p=p->next) {
        insert_head(&head2,p->data);
    }
    return head2;
}
```

在递增链表中插入新结点 - O(n)

- 插入排序的一趟，链表实现

```
List Insert( List L, ElementType X ) {
//思路：先定位最后一个比X小的结点q（while循环）即q->Data<X && q->Next->Data>X，然后把X插在该结点后面
    List p = (List) malloc(sizeof(struct Node));
    p->Data = X,p->Next = NULL;//创建新结点
    List q = L;
    if (L) {
        while (q->Next && q->Next->Data < X) q = q->Next;
        p->Next = q->Next;//在链表中间插入一结点
        q->Next = p;
    } else {
        L = p;//特殊情况
    }
    return L;
}
```

用单向链表完成多项式计算

- 因式分解

```
struct node {
  int coe;
  int exp;
  struct node *next;
} ;
typedef struct node node;
int polynomial(node *h,int x) {
  if (h == NULL) return 0;
  int result = 0;
  int last = h->exp,cur;
  for (node *p = h;p;last = cur,p = p->next)
    cur = p->exp;
    for (int i=last;i>cur;i--) result *= x;
    result += p->coe;
  }
  for (int i=last;i>0;i--) result *= x;
  return result;
}
```

循环链表之猴子选大王

- 与单向链表差别在`tail->next = head`

```
linklist *CreateCircle( int n ) {
    linklist *head = NULL,*last = NULL;
    for (int i=1;i<=n;i++) {
        linklist * p = (linklist*) malloc(sizeof(linklist));
        p->number = i,p->next = NULL;
        scanf("%d",&(p->mydata));

        if (head) {
            last->next = p;
            last = p;
        } else {
            head = last = p;
        }
    }
    last->next = head;
    return head;
}

int KingOfMonkey(int n,linklist *head) {
        linklist *p = head,*q = head;
        int cnt = 0;
        for (int i=0;i<n-1;i++) {
            q = q->next;//找到尾结点
        }
        int d = q->mydata;

        while (p->next != p) //循环退出条件,链表中只剩一个元素
        {
            cnt++;
            if (cnt == d) {
                d = p->mydata;
                cnt = 0;
                printf("Delete No:%d\n",p->number);
                q->next = p->next;
                free(p);
                p = q->next;
            }
            else {
                q = p;
                p = p->next;
            }
        }
        return p->number;
}
```

队列

队列的基本概念

- 队列："先进先出"（FIFO）线性表
- 插入操作只能在队尾(rear)进行，删除操作只能在队首(front)进行
- 储存结构：顺序存储结构/链表结构实现

数据结构

- 单端队列

```
struct _queue {
  int *pBase;
  int front;
  int rear;
  int maxsize;
} QUEUE,*PQUEUE;
```

或者单向双头链表

- 双端队列deque　*英标 [dek]*

de -- double ended 双端队列（两边都可以插入和删除），双向双头链表

循环队列

引入循环队列的原因

- 线性队列浪费front以前的空间

CreateQueue

```
void CreateQueue (PQUEUE Q,int maxsize) {
    Q->pBase = (int*) malloc(sizeof(int)*maxsize);
    front = rear = 0;
    Q->maxsize = maxsize;
}
```

EmptyQueue

```
int EmptyQueue(PQUEUE Q) {
    return Q->front == Q->rear; //队列空的唯一情况
}
```

- `front == rear`

FullQueue

```
int FullQueue(PQUEUE Q) {
    return (Q->rear+1)%(Q->maxsize) == Q->front; //括号是不必要的。实质是(rear+1)%size == front
}
```

- `(rear+1)%size == front`

EnQueue

```
int EnQueue(PQUEUE Q,int val) {
    if (FullQueue(Q)) return 0;
    Q->pBase[Q->rear] = val;
    Q->rear = (Q->rear+1)%Q->maxsize;// rear = (rear+1)%size
    return 1;
}
```

- `q[rear] = val;`
- `rear = (rear+1)%size`

DeQueue

```
int DeQueue(PQUEUE Q,int *val) {
    if (EmptyQueue(Q)) return 0;
    *val = Q->pBase[Q->front];
    Q->front = (Q->front+1)%Q->maxsize; // front = (front+1)%size
    return 1;
}
```

- `*val = q[front];`
- `front = (front+1)%size;`

栈*Stack*

栈的基本概念

FILO 先进后出线性表

数据结构

常用数组

Pop

Push

C2-程序结构