No.

Date

老邱 数据结构 (严云鹏)

程序是算法的实现

常用算法设计方法：穷举法 回溯法 分治法
递归法 贪心法 动态规划法

算法分析 { 时 / 空 } 复杂度

1. 时间复杂度

求解问题的基本操作次数 — 算法时间的度量

算法基本操作的次数与输入次数有关

(1) 平均时间复杂度 (2) 最坏情况时间复杂度

第二章 线性表

分配空间

(1) void *malloc(int size) （size：字节数）

在系统中分配size个存储单位，返回空间基地址

(重新分配) realloc

(2) void free(p) int m=100

float *p （强制类型转换）

p=(float*) malloc(m*sizeof(float))

若p所占空间不用，则释放

free(p)

malloc分配空间时 若想返回 OK, ERROR 先#define

里面常随机数 动态数组：指针 每一个算法用一个函数表达

1. 初始化
2. 销毁：回收空间 函数 free
3. 置空：令 L.length=0，下次再重新定义长度，并赋值

Status (LinkList sqlist &L) 判断空表：length是否为0

{ if (!L.elem) exit(-2); }

4. 插入，在i处插入一个元素，先后移，i处元素仍存在，把它覆盖

最后一个元素位置：

$$\begin{cases} p = \&(L.elem[L.length-1]); \\ p = L.elem + L.length - 1; \end{cases}$$

i处 元素位置：

$$\begin{cases} q = \&(L.elem[i-1] \\ q = L.elem + i - 1 \end{cases}$$

前一个元素后移：

```
*(p+1) = *p
```

```
while(p>=q)
{ *(p-1) = *p;
  p--;
}
L.length+1;
```

p=q时不得移，因为要算出插入位置

```
free(L.elem)
L.elem = NULL
```

(以后实验先写基本操作)

realloc 重新分配空间

```
L.length >= L.listsize
```

超出 满

```
p2 = (ElemType*)realloc(p1, 空间)
```

新指针（防分配不成功

原指针

原理：重新分配另一块已知长的空间

把数据搬过去，收回原空间

“插入”时间复杂度 = O(n) ① 表长 n ② 插入位置

$$\begin{cases} i = L.length+1 & \text{插在尾} \\ & \text{不移 } O(1) \\ i = 1 & \text{全移 } O(n) \end{cases}$$

按查找 Locate(L, x)

```
int Locate (SqList L, Elemtype x)
{ Elemtype *p = L.elem;
  int i = 1;
  while (*p != x && i <= L.length)
  {p++; i++;}
  if (i > L.length) return 0;
  else return i; }
```

作业习题册 P17. 2.12

P18 2.21 ↳支撑首尾

6. 删除

二 线性表的链式表示和实现

1. 链表

结点: | 数据域 | 指针域 |

单链表由1个头指针唯一确定

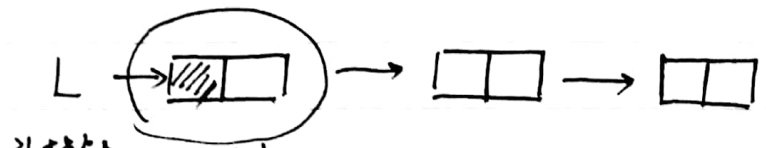
关心逻辑关系而非物理

头结点中data一般不放数据

第0个结点

LNode 类型变量 ↓自定义

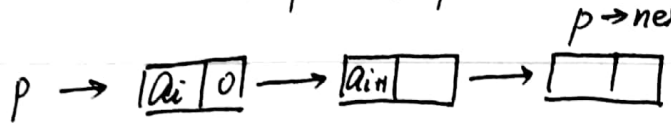
LinkList → | data | next | →



(data部分可空
next指针指向第一个)

空表 L → | | ∧ | 空. 无地址

头结点指向空



*p) ai p→next→data

p→ai

Getelem

```
p = L->next
j = 1
while (j<i && p!=NULL)
{ p = p->next;
  j++;
}
```

4. 比大小，移小的

在表中找值为key的结点，返回其地址

```
Lnode * Locate (Linklist L, Elemtype key)
 p = L->next;
while (p && t!=p->data)
{ p=p->next;
```

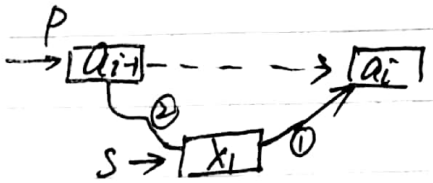
在L中找值key的结点，返回其地址

```
Lnode * Locate( LinkList L, Elemtype key)
  p = L->next
while(t!= p->data)
{   p = p->next;
}
if ct==(p->data))
 { k = p;
  return key;
 }
else  return 0;
```

} 改 return(p);

2. 单链表的插入

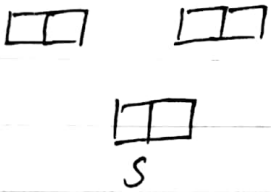
顺序一定 { 1. S->next = p->next
2、 p->next = S



S = (LinkList) malloc(sizeof(LNode))
‖
Lnode *

表Sq 在p后插S结点 { S->next = p->next;
p->next = S; 前提 (知道插入位置)

在p前插S结点, q = Sq; 前1个结点, j < i-1

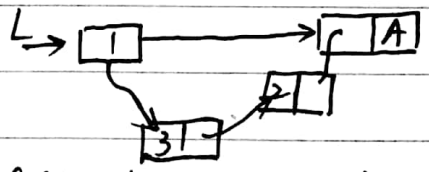


while(q->next != p)
{ q = q->next; }
q->next = S;
S->next = p;

在表尾插入 (q = Sq) q = p

while(q->next != NULL)
{ q = q->next; }
q->next = S;
S->next = NULL;

单链表的建立 (一) 从尾到头



每次都在L后插入

动态链表

L = (LinkList) malloc(sizeof(LNode))
L->next = NULL;
for(i=1; i<=n; i++)
{ p = (LinkList) new LNode
根据规定的数n, 输当n为止

cin >> p->data;
p->next = L->next;
L->next = p;
}

do {…} → p->data
while(□ != x);

p=L j=0 p为空时也可以表示

但p=L->next j=1 若为空，空表也计数，不可

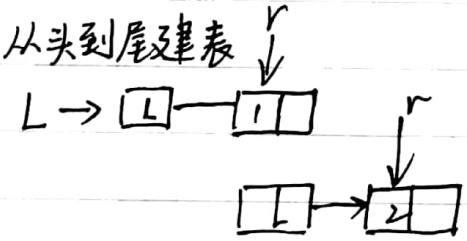
构造一个空的单链表（带头结点）

```
void InitList(LinkList &L)
{ L=(LinkList)malloc(sizeof(ElemType));
  L->next=NULL;}
```

2. 销毁单链表

```
void DestroyList(LinkList &L)
{ while(L)
  {q=L->next; 要有一个指针指向下一个结点
   free(L);
   L=q;
  }
}
```

法二：从头到尾建表




```
L->next=NULL
r=L;
for
{ p=(p->next);  p->next=NULL; r->next=p;
  r=p(r->next);
}
```

p=L j=0（p为空时也可表

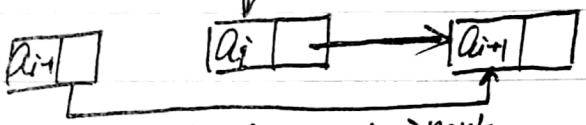
但p=L->next j=1 若为空，空表也计

free( )
只能释放一个结点

3. 将单链表置空（只留头结点）

q = L->next; 第1个元素结点
L->next = NULL;
DestroyList(q);

q = p->next
free(q)



3. 元素删除 → a_{i-1}

p = p->next->next

q

(1) 删p后继结点
(2) 删p直接前驱结点

找 {a->next->next != NULL} a->next = NULL;

删x: {a->next->next != p} a->next = p;

找到x前一个结点 ⇒ 指向x下一个

数据结构 Data structures

E-mail wuyl@njau.edu.cn

线性表　图
栈和队列　查找
树和二叉树　内部排序

数据结构 {①数据的逻辑结构 ②数据的存储结构 {顺序 链式 ③数据的运算

检索、排序、插入、删除、修改

带链表就地逆置
不用每次找表尾

当作两个链表

当表头　L->next = NULL

p = L->next
L->next = NULL

```
void Reverse
{ p = L->next;
  L->next = NULL;
  while(p)
  {q = p->next;
   q->next = L->next;
   L->next = q;
   p = q ;}
```

循环链表：表中最后一个结点的指针域指向头结点

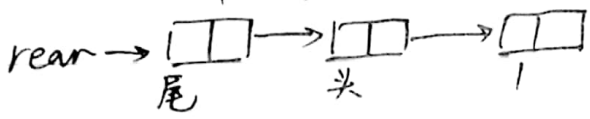
循环条件不同 ：（p且p->next != L）

循环链表　　↳头指针

尾指针rear 指向尾结点

rear->next->next

头指针　第一个元素结点



约瑟夫环，(不要头结点，用尾指针)

防止它指第一个时，无法知道它上一个

双向链表

链表结点中有2个指针域，一个指直接后继，一个指直接前驱

有单尾指针的链表

```
L.head = (  ) malloc(  )
L.tail = L.head
L.head->next = NULL
L.len = 0
```

两个表比大小

```
pa = A.elem
pb = B.elem
while(pa<(A.elem+A.length)&& pb<(B.elem+B.length)
{ if (*pa<*pb)    return (-1) // B大
  if (*pa>*pb)    return (1) // A大
  pa++;
  pb++;                          如果没比出来
} return (A.length-B.length) // 看谁到表尾了   + A大
                                               - B大
```

删范围

```
Status ListDelete(LinkList &L, ElemType mink, ElemType maxk)
{ LinkList p,q, prev=NULL;
  if(mink>maxk) return ERROR;
  p=L;
  prev=p;
  while(p&&p->data<maxk)
  { if(p->data<=mink)
    {prev=p;
     p=p->next;
    }
   else
     {prev->next=p->next;
      q=p;
      p=p->next;
      free(q);
     }
  }
  return OK;
}
```

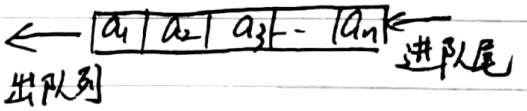
栈

栈是限定了只能从一端操作(一头进,出)的线性表

函数(LA, & Lc)
不加& ↘会变成函数中操作后的样子

在main中原来是什么,现在还是什么

先进先出



队列=只能在一端插入,另一端删除
(队尾) (队头)

P21-24 3.1, 3.4, 3.17

P25 3.29, 3.30.

如果 front, rear 为指针

```
typedef struct
{   ElemType   *base;
    int * front;
    int * rear;
}SqQueue
```

P25. 3.29 3.30

树和二叉树

树：非线性关系，具有层次关系

树：空　非空（一定有根结点）

叶子结点：度为0（无后继）

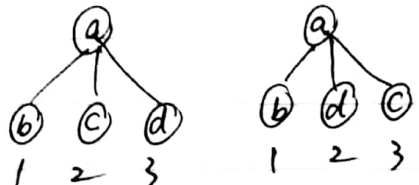
森林：m棵不相交的树 m可=0

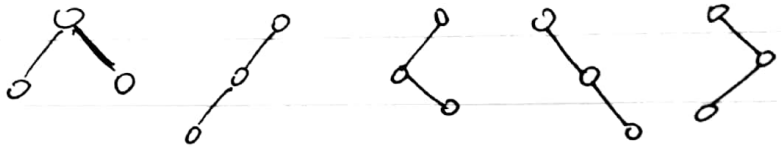
树结构：一对多

每个结点的度 ≤ 2

即每个结点最多有2棵树

3个结点





有序树 ：二者不同

无序树 ：二者相同

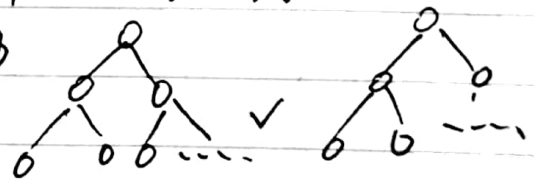
二叉树的性质 1. i层上最多有 2^{i-1} 个结点

N总结点数

2. 深度为k，最多 2^k-1 个结点

3. 深度为k，最多 2^k-1 个结点，满二叉树

完全二叉树 ：最后一层右边缺少



有n个结点的完全二叉树

深度为 $(\log_2 n)+1$

二叉树的存储结构

n层 多 2^n-1 个
少 2^{n-1} 个

1. 顺序 ：一般二叉树，浪费空间
2. 链式

二叉链表 ：存储：数据+左右孩子位置

三叉链表