DATASTRUCT

目录

索引：

第二阶段课程内容：数据结构和算法

目标：掌握以下内容

数据结构：栈,队列,单链表,双链表,二叉树

算法：2 个查找算法和 5 个排序算法

1.基本概念(了解)

数据结构：描述计算机中数据之间的关系和存储方式

2.数据结构分类(了解)

a)逻辑结构：描述数据之间的关系

b)物理结构：描述数据的存储方式

c)运算结构：描述数据的运算过程,例如：+,-等

3)逻辑结构(了解),又分：

a)集合结构：强调数据的总体,不强调数据之间的关系

例如：CSD/ESD1912 班学生,强调班集体,不强调学生关系

b)线性结构：描述数据一对一的前后关系

例如：公交车排队,食堂打饭

c)树形结构：描述数据一对多的关系

例如：树,家谱

d)网状结构：描述数据多对多的关系

例如：网球拍,蜘蛛网

4)物理结构(了解),又分：

a)顺序存储结构：

采用数组来存储数据

b)链式存储结构：

采用单链表,双链表,二叉树来存储数据

5) 常用的数据结构：栈,队列,单链表,双链表,二叉树

一、数据结构之栈的特性

具有后进先出的特性(Last In First Out:LIFO)

或者

具有先进后出的特性(First In Last Out:FILO)

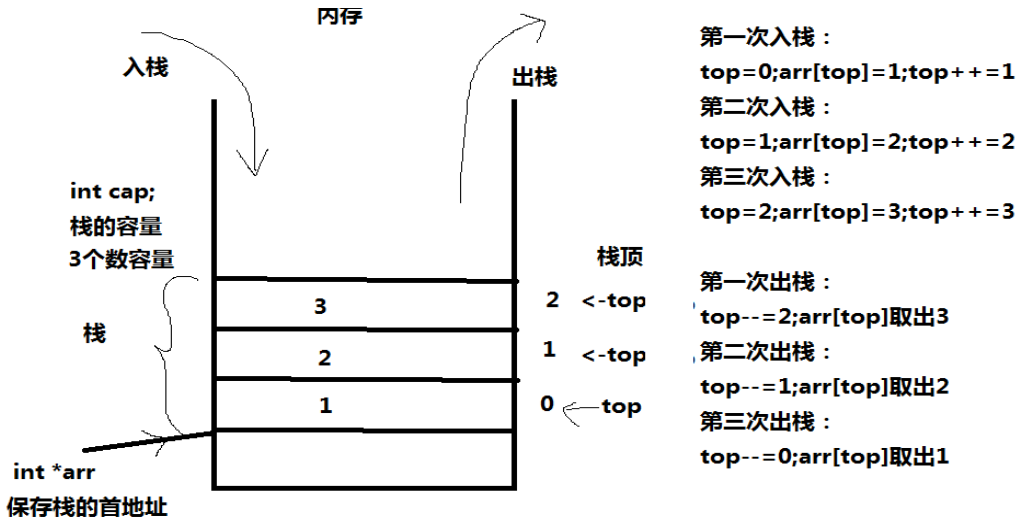
例如：放书和取书(不能抽),鸡尾酒(没吸管)

具体参见:栈.png

入栈又称压栈：push

出栈又称弹栈: pop

图解：




```
1 /*栈的演示*/
2 #include <stdio.h>
3 #include <stdlib.h> //为了用 malloc 函数
4 /*声明结构体数据类型,用于描述栈的属性*/
5 typedef struct stack {
6         int *arr; //栈的首地址
7         int cap; //栈的容量
8         int top; //栈顶,便于入栈和出栈
9 }stack_t;
10 /*定义初始化栈的函数*/
11 void stack_init(stack_t *stack, int cap){
12         stack->arr = (int *)malloc(cap * sizeof(int)); //分配 10 个数的内存
13         stack->cap = cap; //指定容量为 10 个数
14         stack->top = 0; //一开始为空栈
15 }
16 /*定义释放栈内存资源函数*/
17 void stack_del(stack_t *stack){
```

```
18          free(stack->arr); //释放内存
19          stack->cap = 0;
20          stack->top = 0;
21 }
22 /*定义判断栈满的函数*/
23 int stack_full(stack_t *stack){
24          return stack->top >= stack->cap; //满返回 1,不满返回 0
25 }
26 /*定义判断栈空的函数*/
27 int stack_empty(stack_t *stack) {
28          return !stack->top; //空(top=0)返回 1,不空返回 0
29 }
30 /*定义入栈函数,将数据保存到栈中*/
31 void stack_push(stack_t *stack, int data){
32          stack->arr[stack->top++] = data;
33 }
34 /*定义出栈函数,从栈中取出数据*/
35 int stack_pop(stack_t *stack) {
36          return stack->arr[--stack->top];
37 }
38 int main(void){
39          //1.定义栈
40          stack_t stack;
41          //2.初始化栈
42          stack_init(&stack, 10); //栈的容量为 10 个数
43          //3.判断栈满和栈空
44          int ret = 0;
45          ret = stack_full(&stack);
46          printf("%s\n", ret ?"满":"不满");
47          ret = stack_empty(&stack);
48          printf("%s\n", ret ?"空":"不空");
49          //4.向栈中保存 10 个数据
50          int i = 250;
51          while(!stack_full(&stack)) {
52                  stack_push(&stack, i++);
53          }
54          //5.从栈中取出数据并且打印
55          while(!stack_empty(&stack)) {
56                  printf("%d\n", stack_pop(&stack));
57          }
58          //释放栈
59          stack_del(&stack);
60          return 0;
61 }
```

二、数据结构之队列

1.队列特点

先进先出:First In First Out:FIFO

实现原理：队列.png

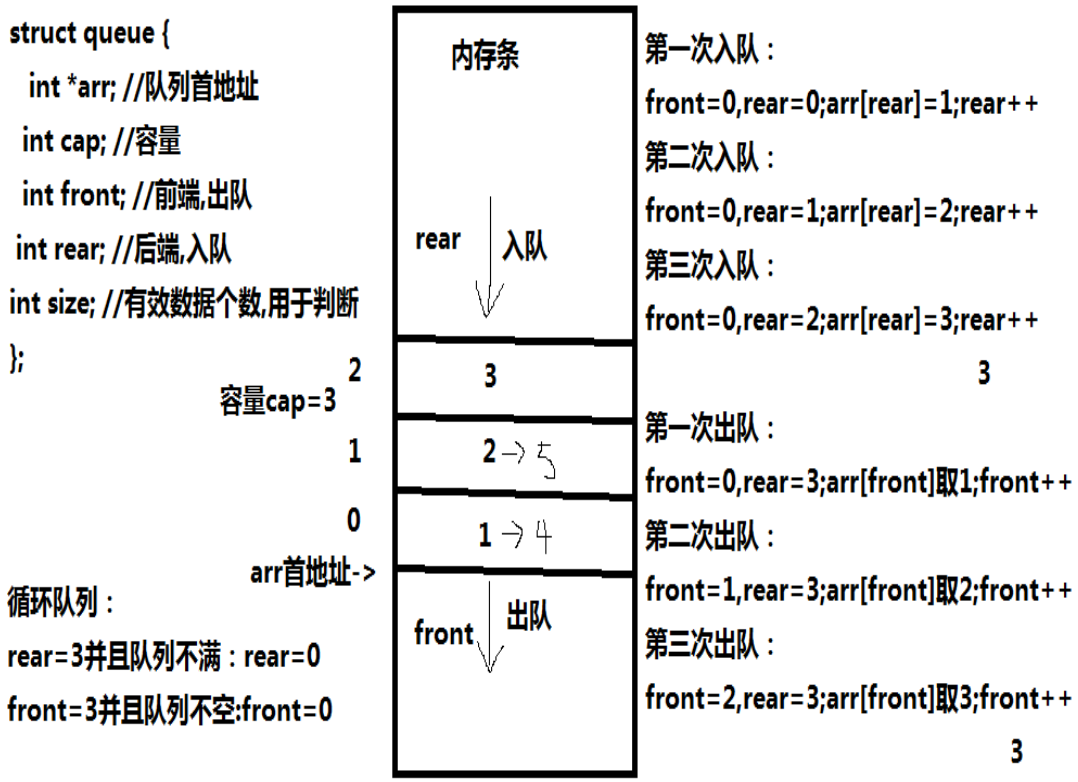
2.队列的经典应用：linux 系统的经典的三大队列

消息队列：实现程序与程序之间的交互

工作队列：实现任务延后执行

等待队列：实现程序休眠和唤醒

图解：



queue.h

```
 1 /*头文件卫士*/
 2 #ifndef __QUEUE_H
 3 #define __QUEUE_H
 4
 5 #include <stdio.h>
 6 #include <stdlib.h>
 7
 8 /*声明结构体数据类型描述队列的属性*/
 9 typedef struct queue {
10         int *arr; //首地址
11         int cap; //容量
12         int size; //有效数据的个数
```

```
13          int front; //前端,出队
14          int rear; //后端,入队
15 }queue_t;
16
17 /*声明初始化队列函数*/
18 extern void queue_init(queue_t *, int);
19
20 /*声明释放队列内存函数*/
21 extern void queue_del(queue_t *);
22
23 /*判断是否满*/
24 extern int queue_full(queue_t *);
25
26 /*判断是否空*/
27 extern int queue_empty(queue_t *);
28
29 /*定义入队函数*/
30 extern void queue_push(queue_t *, int);
31
32 /*定义出队函数*/
33 extern int queue_pop(queue_t *);
34
35 #endif //结束
```

Queue.c

```
c    1 #include "queue.h" //包含自己的头文件
   2
   3 /*定义初始化队列函数*/
   4 void queue_init(queue_t *queue, int cap)
   5 {
   6          queue->arr = (int *)malloc(cap * sizeof(queue->arr[0])); //分配内存
   7          queue->cap = cap; //指定容量
   8          queue->size = 0; //没有数
   9          queue->front = 0; //没有数
  10          queue->rear = 0; //没有数
  11 }
  12 /*定义释放队列内存函数*/
  13 void queue_del(queue_t *queue)
  14 {
  15          free(queue->arr);
  16          queue->cap = 0;
  17          queue->size = 0;
  18          queue->rear = 0;
  19          queue->front = 0;
```

```
20 }
21 /*判断是否满*/
22 int queue_full(queue_t *queue)
23 {
24         return queue->size >= queue->cap; //满返回 1,不满返回 0
25 }
26 /*判断是否空*/
27 int queue_empty(queue_t *queue)
28 {
29         return !queue->size; //空返回 1,不空返回 0
30 }
31 /*定义入队函数*/
32 void queue_push(queue_t *queue, int data){
33         if(queue->rear >= queue->cap)
34                 queue->rear = 0; //构造一个循环队列
35
36         queue->arr[queue->rear++] = data;
37         queue->size++; //更新有效数据的个数,队列不满,前面有空位置
38 }
39 /*定义出队函数*/
40 int queue_pop(queue_t *queue){
41         if(queue->front >= queue->cap)
42             queue->front = 0; //构造一个循环队列,队列不空并且前面有数据
43              queue->size--; //更新有效数据的个数
44         return queue->arr[queue->front++]; //根据 front 下标取数
45 }
```

Main.c

```
1#include "queue.h" //各种声明
2
3 int main(void)
4 {
5         //定义队列
6         queue_t queue;
7         //初始化队列
8         queue_init(&queue, 4);
9         printf("%s\n", queue_full(&queue) ? "满":"不满");
10        printf("%s\n", queue_empty(&queue) ? "空":"不空");
11        //入队:10 20 30 40
12        for(int i = 10; i <= 40; i += 10) {
13                if(!queue_full(&queue)) {
14                        queue_push(&queue, i);
15                }
16        }
```

```
17        //出队:10 20
18        for(int i = 0; i < 2; i++) {
19            if(!queue_empty(&queue)) {
20                printf("%d\n", queue_pop(&queue));
21            }
22        }
23        //入队:50 60 30 40
24        for(int i = 50; i <= 60; i += 10) {
25            if(!queue_full(&queue)) {
26                queue_push(&queue, i);
27            }
28        }
29        //整体出队
30        while(!queue_empty(&queue)) {
31            printf("%d ", queue_pop(&queue));
32        }
33        printf("\n");
34
35        //释放内存,删除队列
36        queue_del(&queue);
37        return 0;
38 }
```

Strcmp.c

```
 1 /*字符串比较函数*/
 2 #include <stdio.h>
 3 //cs="abc", ct="abc"/"aba"/"abd"
 4 int my_strcmp(const char *cs, const char *ct){//strcmp:标准库函数,此代码执行
   效率比咱们的代码执行效率低很多
 5     unsigned char c1, c2;
 6     while(1) {
 7         c1 = *cs++;
 8         c2 = *ct++;
 9         if(c1 != c2)
10             return c1 < c2 ? -1 : 1;
11         if(!c1)
12             break;
13     }
14     return 0;
15 }
16 //再编写一个字符串比较函数,但是指定比较字符的个数："abcd","abad",要求
       只比较前 3 个字符
17 int my_strncmp(const char *cs, const char *ct, int count){
18     unsigned char c1, c2;
```

```
19        while(count) {
20              c1 = *cs++;
21              c2 = *ct++;
22              if(c1 != c2)
23                    return c1 < c2 ? -1 : 1;
24              if(!c1)
25                    break;
26              count--;
27        }
28        return 0;
29 }
30 int main(void){
31        int ret = 0;
32        ret = my_strcmp("abc", "abc");
33        printf("ret = %d\n", ret); //0
34        ret = my_strcmp("abc", "aba");
35        printf("ret = %d\n", ret); //1
36        ret = my_strcmp("abc", "abd");
37        printf("ret = %d\n", ret); //-1
38        ret = my_strncmp("abcd", "abac", 2); //只比较前 3 个字符
39        printf("ret = %d\n", ret); //1
40        return 0;
41 }
```

三、Makefile 提升

1.Makefile 的变量

类似 C 语言的#define,可以提高代码的可移植性,将来让 Makefile 改动的工作量减少。

建议：变量名用大写

定义变量的形式：

(1)OBJ=helloworld.o #定义变量并且初始化为 helloworld.o

(2)OBJ:=helloworld.o #定义变量并且初始化为 helloworld.o

(3)OBJ = helloworld.o

OBJ += test.o #在 OBJ 变量的基础上继续增加一个值

结果是 OBJ=helloworld.o test.o

(4)OBJ ?= helloworld.o #如果 OBJ 之前赋过值,此赋值失效,否则起效

例如：

OBJ=test.o

OBJ ?= helloworld.o #结果：OBJ=test.o

(5)获取 Makefile 变量值的语法：$(变量名)

例如：$(OBJ)等于 helloworld.o

案例：优化 queue 队列代码的 Makefile

注意：命令前面加@表示隐藏命令的执行打印

2.伪目标

概念：没有依赖的目标

例如：

clean:

rm hellworld helloworld.o

当执行 make clean 时,自动执行 clean 伪目标对应的命令 rm helloworld helloworld.o

案例：优化 queue 队列代码的 Makefile,添加 clean 删除功能

Makefile 工具书：<<跟我一起写 Makefile>>

```
1 #定义变量
2 BIN=list
3 OBJ=main.o list.o
4
5 #CROSS_COMPILE=arm-linux-
6 CC=$(CROSS_COMPILE)gcc
7
8 INSTALL_PATH=/home/tarena/stdc/
9
10 #制定规则 1
11 $(BIN):$(OBJ)
12          $(CC) -o $(BIN) $(OBJ)
13          @echo "程序编译完成,恭喜!"
14
15 #制定规则 2
16 %.o:%.c
17          @$(CC) -c -o $@ $<
18
19
20 #制定规则 3 #执行 make clean
21 clean:
22          @rm $(BIN) $(OBJ)
23
24 #制定规则 4 #执行 make install
25 install:
26          @cp $(BIN) $(INSTALL_PATH)
```

四、数据结构之单链表

1.任务：编写程序,让三个学生信息连接起来

具体参见：单链表.png

2.单链表特性

单链表的操作方式朝一个方向

每个节点的结构体：

struct node {

```
        节点包含的数据信息;
        struct node *next; //保存下一个节点的首地址
};
```

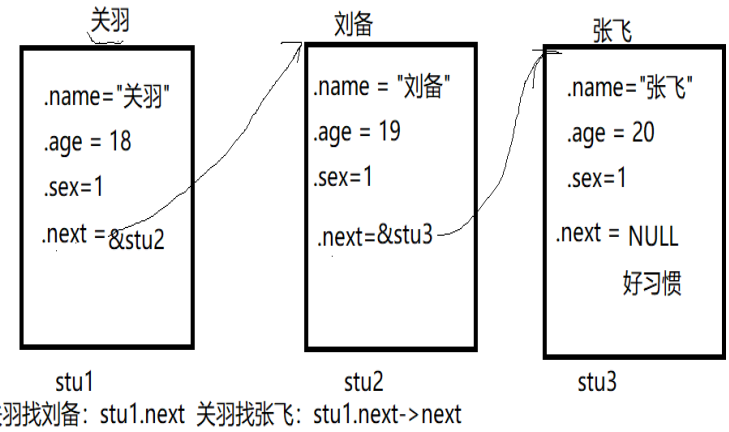
整个链表的结构体

```
struct list {
        struct node *head; //头结点
        struct node *tail; //尾结点
};
```

参考代码：list.h list.c main.c Makefile

图解：

```
struct student {
    char name[32];
    int age;
    int sex;
struct student *next;
};
```



关羽找刘备：stu1.next 关羽找张飞：stu1.next->next

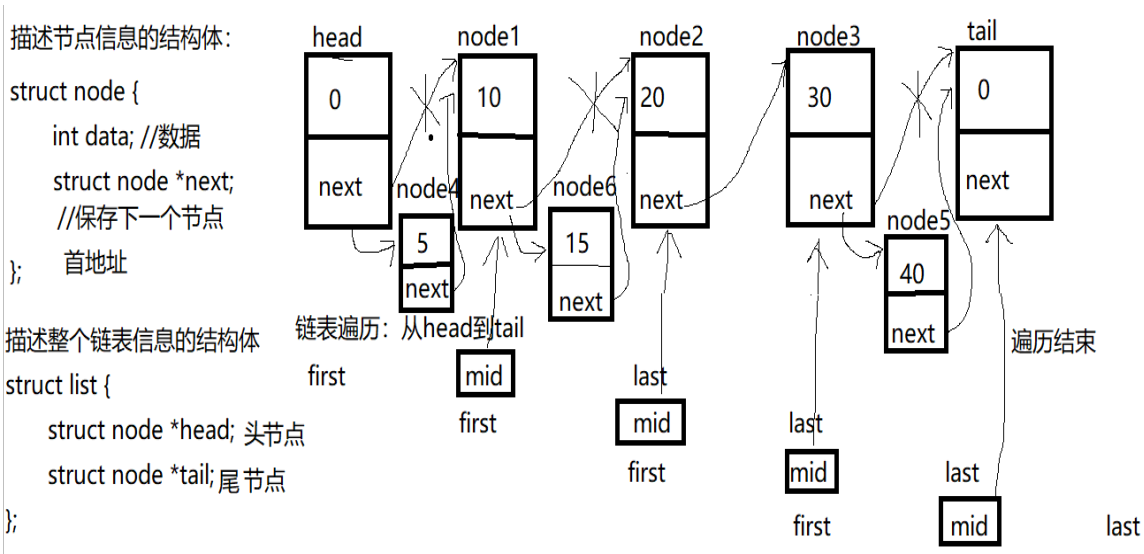
打印名字:

```
printf("%s %s %s\n", stu1.name, stu1.next->name, stu1.next->next->name)
```

```
1 /*简陋链表操作演示*/
2 #include <stdio.h>
3 /*声明描述学生信息的结构体*/
4 typedef struct student {
5         char name[32]; //姓名
6         int age; //年龄
7         struct student *next; //保存下一个学生结构体变量的首地址
8 }stu_t;
9
10 int main(void)
11 {
12        //1.定义初始化三个学生信息的结构体变量
13        stu_t stu1 = {.name = "刘备", .age = 20};
14        stu_t stu2 = {.name = "关羽", .age = 21};
15        stu_t stu3 = {.name = "张飞", .age = 22};
16
17        //2.将三个学生连接起来
18        stu1.next = &stu2;
```

```
19          stu2.next = &stu3;
20          stu3.next = NULL;
21
22          //3.打印名字
23        printf("%s %s %s\n", stu1.name, stu2.name, stu3.name);
24        printf("%s %s %s\n", stu1.name, stu1.next->name, stu1.next->next->name);
25          return 0;
26 }
```

图解：



List.h:

```
1 /*单链表演示：list.h*/
2 #ifndef __LIST_H
3 #define __LIST_H
4
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 /*声明描述每个节点属性信息的结构体*/
9 typedef struct node {
10          int data; //保存节点的数据
11          struct node *next; //保存下一个节点的首地址
12 }node_t;
13
14 /*声明描述整个链表属性信息的结构体*/
15 typedef struct list {
16          struct node *head; //指向头节点
17          struct node *tail;//指向尾节点
18 }list_t;
```

```
19
20 /*声明遍历链表打印节点数据的函数*/
21 extern void list_travel(list_t *);
22 /*声明添加函数：有小到大*/
23 extern void list_add(list_t *, int);
24 /*声明前插函数：新节点永远跟在头节点 head 的后面*/
25 extern void list_add_head(list_t *, int);
26 /*声明后插函数：新节点永远跟在尾节点 tail 的前面*/
27 extern void list_add_tail(list_t *, int);
28 /*声明删除节点函数*/
29 extern void list_del(list_t *, int);
30 /*声明初始化链表函数*/
31 extern void list_init(list_t *);
32 /*声明删除链表释放内存函数*/
33 extern void list_deinit(list_t *);
34 #endif
```

List.c:

```
 1 /*单链表演示：list.c*/
 2 #include "list.h"
 3 /*定义创建节点并且初始化节点的函数*/
 4 static node_t *create_node(int data){
 5         //分配节点对应的内存
 6         node_t *p = (node_t *)malloc(sizeof(node_t));
 7         //初始化创建的新节点
 8         p->data = data;
 9         p->next = NULL;
10         return p;
11 }
12 /*单链表初始化函数*/
13 void list_init(list_t *list){
14         //给头节点分配内存并且初始化
15         list->head = create_node(0);
16         //给尾节点分配内存并且初始化
17         list->tail = create_node(0);
18         //将头节点和尾节点连接起来
19         list->head->next = list->tail;
20         list->tail->next = NULL;
21 }
22 /*释放链表节点内存函数*/
23 void list_deinit(list_t *list){
24         node_t *pnode = list->head;
25         while(pnode != list->tail) {
26                 node_t *ptmp = pnode->next;//暂存下一个节点首地址
```

```
27              free(pnode); //释放节点内存
28              pnode = ptmp; //准备释放下一个节点内存
29          }
30 }
31 /*遍历链表节点函数*/
32 void list_travel(list_t *list){
33          for(node_t *pnode = list->head; pnode != list->tail; pnode=pnode->next) {
34              //定义三个游标
35              node_t *pfirst = pnode;
36              node_t *pmid = pfirst->next;
37              node_t *plast = pmid->next;
38              if(pmid != list->tail)
39                  printf("%d ", pmid->data);
40          }
41          printf("\n");
42 }
43
44 /*添加节点函数：有小到大*/
45 void list_add(list_t *list, int data)
46 {
47          //1.创建初始化一个新节点
48          node_t *ptmp = create_node(data);
49          //2.遍历链表找到新节点的合适位置然后插进入
50          for(node_t *pnode = list->head;
51                  pnode != list->tail; pnode=pnode->next) {
52              //3.定义三个游标
53              node_t *pfirst = pnode;
54              node_t *pmid = pfirst->next;
55              node_t *plast = pmid->next;
56              //4.找位置插入,插入的位置就是在 pfirst 和 pmid 中间
57              if(pmid->data >= ptmp->data
58                  || pmid == list->tail) {
59                  pfirst->next = ptmp;
60                  ptmp->next = pmid;
61                  break;
62              }
63          }
64 }
65
66 /*前插*/
67 void list_add_head(list_t *list, int data)
68 {
69          //1.创建新节点
70          node_t *ptmp = create_node(data);
```

```
71          //2.临时保存第一个节点的首地址
72          node_t *pnode = list->head->next;
73          //3.插入新节点
74          list->head->next = ptmp;
75          ptmp->next = pnode;
76  }
77  /*后插*/
78  void list_add_tail(list_t *list, int data){
79          //1.创建新节点
80          node_t *ptmp = create_node(data);
81          //2.遍历找到最后一个节点
82          for(node_t *pnode = list->head;
83                          pnode != list->tail; pnode=pnode->next) {
84                  //3.定义三个游标
85                  node_t *pfirst = pnode;
86                  node_t *pmid = pfirst->next;
87                  node_t *plast = pmid->next;
88                  //4.找最后节点,pfirst 指向最后节点
89                  //pmid 指向尾节点
90                  if(pmid == list->tail) {
91                          //将新节点插入到 pfirst 和 pmid 之间
92                          pfirst->next = ptmp;
93                          ptmp->next = pmid;
94                          break;
95                  }
96          }
97  }
98  /*删除节点*/
99  void list_del(list_t *list, int data){
100         //1.遍历找到要删除的节点
101         for(node_t *pnode = list->head;
102                         pnode != list->tail; pnode=pnode->next) {
103                 //2.定义三个游标
104                 node_t *pfirst = pnode;
105                 node_t *pmid = pfirst->next;
106                 node_t *plast = pmid->next;
107                 //3.判断,让 pmid 的前节点 first->next 指向
108                 //pmid 的后节点 plast,这样 pmid 对应的节点错开
109                 if(pmid->data == data && pmid != list->tail) {
110                         pfirst->next = plast;
111                 }
112         }
113 }
```

Main.c:

```
 1 /*单链表演示:main.c*/
 2 #include "list.h"
 3 int main(void){
 4         //1.创建单链表
 5         list_t list;
 6         //2.初始化单链表
 7         list_init(&list);
 8         //3.添加三个节点并遍历打印
 9         list_add(&list, 10);
10         list_add(&list, 20);
11         list_add(&list, 30);
12         list_travel(&list);
13         //4.添加新节点并打印
14         list_add(&list, 15);
15         list_travel(&list);
16         //5.前插
17         list_add_head(&list, 5);
18         list_add_head(&list, 3);
19         list_travel(&list);
20         //6.后插
21         list_add_tail(&list, 40);
22         list_add_tail(&list, 50);
23         list_travel(&list);
24         //7.删除节点
25         list_del(&list, 20);
26         list_travel(&list);
27         //删除整个链表
28         list_deinit(&list);
29         return 0;
30 }
```

Makefile:

```
 1 #定义变量
 2 BIN=list
 3 OBJ=main.o list.o
 4
 5 #编译生成手机上运行程序
 6 #CROSS_COMPILE=arm-linux-
 7 #编译生成电脑运行的程序
 8 CROSS_COMPILE=
 9 CC=$(CROSS_COMPILE)gcc
10
```

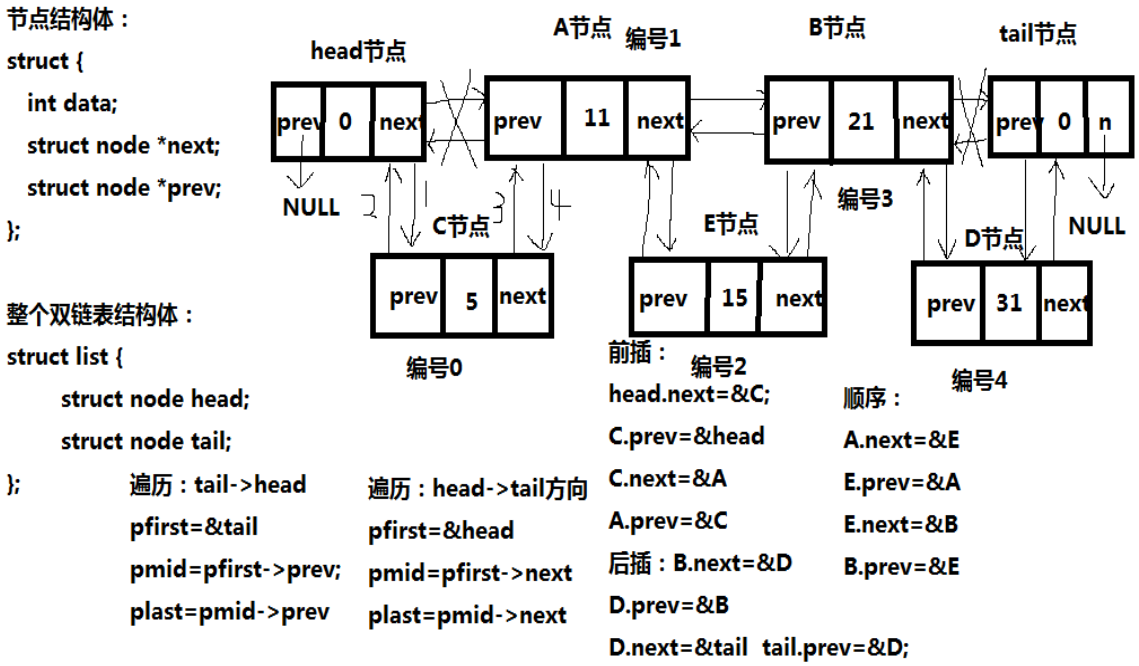
```
11 #制定编译规则 1
12 $(BIN):$(OBJ)
13          $(CC) -o $(BIN) $(OBJ)
14
15 #制定编译规则 2
16 %.o:%.c
17          $(CC) -c -o $@ $<
18
19 #伪目标
20 clean:
21          rm $(BIN) $(OBJ)
```

五、数据结构之双链表

双链表每个节点的结构体：

```
Struct node{
    数据;
    Struct node *next;//保存上一个节点的首地址
    Struct node *prev;//保存下一个节点的首地址
};
```

图解：



List.h

```
1 /*双链表演示*/
2 #ifndef __LIST_H
3 #define __LIST_H
4 #include <stdio.h>
```

```
 5 #include <stdlib.h>
 6 /*声明描述节点信息的结构体*/
 7 typedef struct node {
 8         int data; //数据
 9         struct node *next; //指向下一个节点
10         struct node *prev; //指向前一个几点
11 }node_t;
12 /*声明描述整个链表的结构体*/
13 typedef struct list {
14         struct node head; //头结点
15         struct node tail; //尾节点
16 }list_t;
17 /*声明初始化函数*/
18 extern void list_init(list_t *);
19 /*声明清除链表函数*/
20 extern void list_deinit(list_t *);
21 /*判断链表是否为空，空返回 1，非空返回 0*/
22 extern int list_empty(list_t *);
23 /*获取链表节点个数*/
24 extern int list_size(list_t *);
25 /*后插*/
26 extern void list_add_tail(list_t *, int);
27 /*前插*/
28 extern void list_add_head(list_t *, int);
29 /*顺序插*/
30 extern void list_add(list_t *, int);
31 /*只删除第一个节点*/
32 extern void list_del_head(list_t *);
33 /*只删除最后一个节点*/
34 extern void list_del_tail(list_t *);
35 /*删除某个数字所在的节点*/
36 extern void list_del(list_t *, int);
37 /*通过节点编号获取节点数据*/
38 extern int list_get(list_t *, int);
39 #endif
```

List.c

```
 1 /*双链表演示*/
 2 #include "list.h"
 3 /*初始化链表函数*/
 4 void list_init(list_t *list)
 5 {
 6       list->head.next = &list->tail; //头指向尾
 7       list->tail.prev = &list->head; //尾指向头
```

```
 8       list->head.prev = NULL;
 9       list->tail.next = NULL;
10 }
11
12 /*判断链表是否为空：空返回 1,非空返回 0*/
13 int list_empty(list_t *list)
14 {
15       return list->head.next == &list->tail;
16 }
17
18 /*获取链表节点个数*/
19 int list_size(list_t *list)
20 {
21       int count = 0; //计数
22       for(node_t *pnode = &list->head; pnode != &list->tail; pnode = pnode->next)
{
23             node_t *pfirst = pnode;
24             node_t *pmid = pfirst->next;
25             node_t *plast = pmid->next;
26             if(pmid != &list->tail)
27                   count++; //只要没有到 tail,加一个
28       }
29       return count; //返回节点个数
30 }
31 /*创建初始化节点*/
32 static node_t *create_node(int data)
33 {
34       //给新节点分配内存
35       node_t *p = (node_t *)malloc(sizeof(node_t));
36       //初始化节点
37       p->next = NULL;
38       p->prev = NULL;
39       p->data = data; //数据
40       return p;
41 }
42 /*将新节点插入到 pfirst 和 pmid 之间*/
43 static void insert_node(node_t *pfirst, node_t *pmid, node_t *pnode)
44 {
45       pfirst->next = pnode;
46       pnode->prev = pfirst;
47       pnode->next = pmid;
48       pmid->prev = pnode;
49 }
50 /*前插*/
```

```
51 void list_add_head(list_t *list, int data)
52 {
53         //1.创建新节点
54         node_t *pnode = create_node(data);
55         //2.定义游标
56         node_t *pfirst = &list->head; //pfirst 指向头节点
57         node_t *pmid = pfirst->next; //pmid 指向第一个节点
58         node_t *plast = pmid->next; //plast 指向第二节点
59         //3.插入
60         insert_node(pfirst, pmid, pnode);
61 }
62 /*后插*/
63 void list_add_tail(list_t *list, int data)
64 {
65         //1.创建新节点
66         node_t *pnode = create_node(data);
67         //2.定义游标
 68          node_t *pfirst = list->tail.prev; //pfirst 指向最后一个节点
69         node_t *pmid = pfirst->next; //pmid 指向 tail
70         node_t *plast = pmid->next; //plast 指向 NULL
71         //3.插入
72         insert_node(pfirst, pmid, pnode);
73 }
74 /*由小到大插*/
75 void list_add(list_t *list, int data)
76 {
77         //1.创建新节点
78         node_t *pnode = create_node(data);
79         //2.遍历
80         for(node_t *ptmp = &list->head; ptmp != &list->tail; ptmp = ptmp->next) {
81                 node_t *pfirst = ptmp;
82                 node_t *pmid = pfirst->next;
83                 node_t *plast = pmid->next;
84                 //3.查找位置并且插入
85                 if(pmid->data > pnode->data || pmid == &list->tail) {
86                         insert_node(pfirst, pmid, pnode);
87                         break;
88                 }
89         }
90 }
91 /*删除节点*/
92 static void del_node(node_t *pfirst, node_t *pmid , node_t *plast)
93 {
94         //1.连接 pfirst 和 plast,干掉 pmid(pmid 指向要删除的节点)
```

```
95          pfirst->next = plast;
96          plast->prev = pfirst;
97          //2.释放内存
98          free(pmid);
99          pmid = NULL;
100 }
101 /*只删除最后一个节点*/
102 void list_del_tail(list_t *list)
103 {
104         //1.判断链表是否为空
105         if(list->head.next == &list->tail) {
106                 printf("链表空了.\n");
107                 return;
108         }
109
110         //2.定义游标
111         node_t *plast = &list->tail; //plast 指向 tail
112         node_t *pmid = plast->prev; //pmid 指向最后一个节点
113         node_t *pfirst = pmid->prev; //pfirst 指向倒数第二个节点
114
115         //3.删除 pmid 指向的最后一个节点
116         del_node(pfirst, pmid, plast);
117 }
118 /*只删除第一个节点*/
119 void list_del_head(list_t *list)
120 {
121         //1.判断链表是否为空
122         if(list->head.next == &list->tail) {
123                 printf("链表空了.\n");
124                 return;
125         }
126
127         //2.定义游标
128         node_t *pfirst = &list->head; //pfirst 指向头>节点
129         node_t *pmid = pfirst->next; //pmid 指向第一个节点
130         node_t *plast = pmid->next; //plast 指向第二个节点
131
132         //3.删除 pmid 指向的第一个节点
133         del_node(pfirst, pmid, plast);
134 }
135 /*删除某个数字所在的节点*/
136 void list_del(list_t *list, int data)
137 {
138         //1.遍历
```

```
139         for(node_t *ptmp = &list->head; ptmp != &list->tail; ptmp = ptmp->next) {
140             node_t *pfirst = ptmp;
141             node_t *pmid = pfirst->next;
142             node_t *plast = pmid->next;
143             //2.找数字所在的节点
144             if(pmid->data == data && pmid !=    &list->        tail)
145                 del_node(pfirst, pmid, plast);//删除所在的节点
146         }
147 }
148 /*根据节点编号获取对应的数据*/
149 int list_get(list_t *list, int index)
150 {
151         int count = 0; //计数
152
153         //2.找节点
154         for(node_t *ptmp = &list->head; ptmp != &list ->tail; ptmp = ptmp->next) {
155             node_t *pfirst = ptmp;
156             node_t *pmid = pfirst->next;
157             node_t *plast = pmid->next;
158             //3.判断 count 和 index
159             if(count == index && pmid != &list->tail)
160                 return pmid->data; //返回编号所在节点的数据
161             count++;
162         }
163 }
164 /*清理链表*/
165 void list_deinit(list_t *list)
166 {
167         while(list->head.next != &list->tail) {
168             //定义游标
169             node_t *pfirst = &list->head;
170             node_t *pmid = pfirst->next;
171             node_t *plast = pmid->next;
172             //先连接 pfirst 和 plast 然后释放 pmid 内存
173             del_node(pfirst, pmid, plast);//删除所在的节点
174         }
175 }
```

Main.c

```
1 /*双链表测试*/
2 #include "list.h"
3 int main(void){
4         //1.创建链表
5         list_t list;
```

```
6       //2.初始化链表
7       list_init(&list);
8       //3.前插
9       list_add_head(&list, 50);
10      list_add_head(&list, 20); //20 50
11      //4.后插
12      list_add_tail(&list, 70);
13      list_add_tail(&list, 100); //20 50 70 100
14      //5.小到大插
15      list_add(&list, 80);
16      list_add(&list, 30);
17      list_add(&list, 40);
18      list_add(&list, 60);
19      list_add(&list, 90);
20      list_add(&list, 10);
21      //6.获取节点个数
22      int size = list_size(&list);
23      printf("节点个数是%d\n", size);
24      //7.打印节点的数据
25      for(int i = 0; i < size; i++)
26              printf("%d ", list_get(&list, i));
27      printf("\n");
28      //删除第一个节点
29      list_del_head(&list);
30      //删除最后一个节点
31      list_del_tail(&list);
32      //删除 50 所在的节点
33      list_del(&list, 50);
34      size = list_size(&list);
35      printf("节点个数是%d\n", size);
36      //打印节点的数据
37      for(int i = 0; i < size; i++)
38              printf("%d ", list_get(&list, i));
39      printf("\n");
40      //清除链表
41      list_deinit(&list);
42      return 0;
43 }
```

六、递归函数(recursive)

特性：自己调用自己,重复相同的事情

例如：

```
void print(void)
{
        printf("1\n");
        print(); //自己调用自己
        printf("2\n");
}
int main(void)
{
        print();
        return 0;
}
```

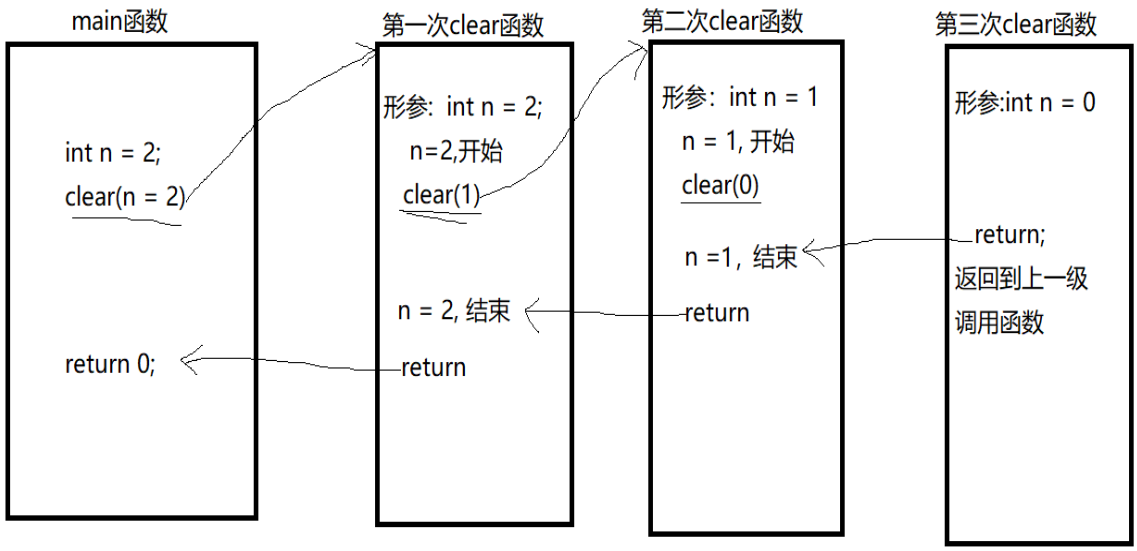
注意：递归函数要有退出的条件

Recursive1.c

```
/*递归函数演示*/
#include <stdio.h>
/*
main
    clear(2)
        n=2,开始
        clear(1)
            n=1,开始
            clear(0)
                return 返回
            n=1,结束
            return 返回
        n=2,结束
        return 返回
return 0
*/
void clear(int n){
    if(n != 0) {
            printf("n = %d,  开始.\n", n);
            clear(n-1);
            printf("n = %d,  结束.\n", n);
            return;
    }
    return;
}
int main(void){
    int n = 2;
```

```
28          clear(n);
29          return 0;
30 }
```

图解：



Recur2.c

```
 1 #include <stdio.h>
 2 /*
 3   main
 4         print(3)
 5               print(2)
 6                     print(1)
 7                           1
 8                           return
 9                     2
10               3
11 \n
12   */
13 void print(int max)
14 {
15         if(max == 1) {
16               printf("1 ");
17               return;
18         }
19         print(max-1);
20         printf("%d ", max);
21 }
22
23 int main(void)
```

```
24 {
25          int max = 3;
26          print(max);
27          printf("\n");
28          return 0;
29 }
```

七、二叉树

1.二叉树是一种特殊的树(具有一对多)

2.二叉树的特点

(1)每个节点最多有两个子节点(可以没有,可以有一个)

(2)单根性,每个子节点有且仅有一个父节点,整棵树只有一个根节点

左子树：根节点左边的子树

右子树：根节点右边的子树

(3)一般用递归函数处理

3.有序二叉树(核心的二叉树)

(1)定义：一般来说,当左子树不为空时,左子树的元素值小于根节点，当右子树不为空时,右子树的元素值大于根节点。

(2)例如： 现在有这么一组数(类似礼物),要求按照有序二叉树的特点将这些数挂接到树上。

(3)三种遍历方式：

先序遍历：处理节点自己的数据->处理左节点->处理右节点

中序遍历: 处理左节点->处理节点自己的数据->处理右节点

后续遍历：处理左节点->处理右节点->处理节点自己的数据

(4)有序二叉树应用：

用于搜索和查找数据极其的方便,又称二叉查找树(各种砍半)

4.有序二叉树的结构体：

声明描述有序二叉树每个节点的结构体

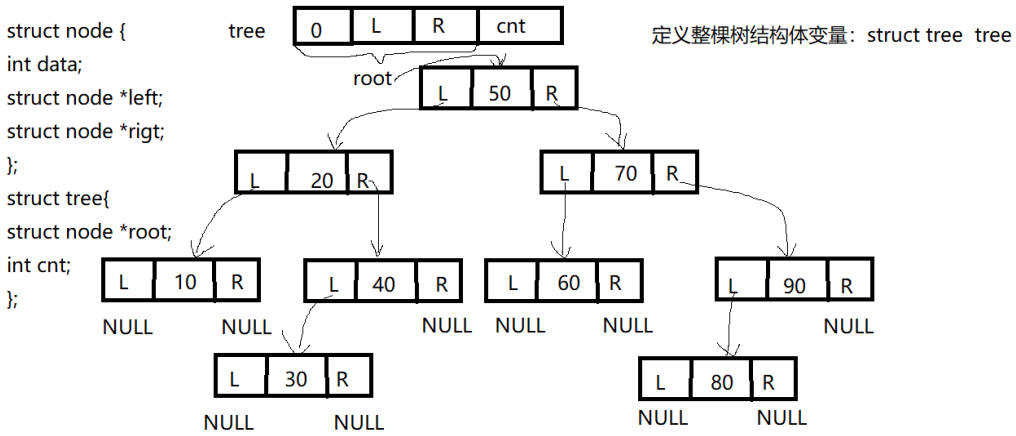
```
struct node {
        数据;
        struct node *left; //保存左节点的地址
        struct node *right; //保存右节点的地址
};

描述树的结构体
struct tree {
        struct node *root; //记录数的"根节点"
        int cnt; //记录数的节点个数
};
```

注意：我这里的数不仅仅是整棵树,也可以说是整棵树中的其中某棵子树

参考代码：tree.h tree.c main.c Makefile

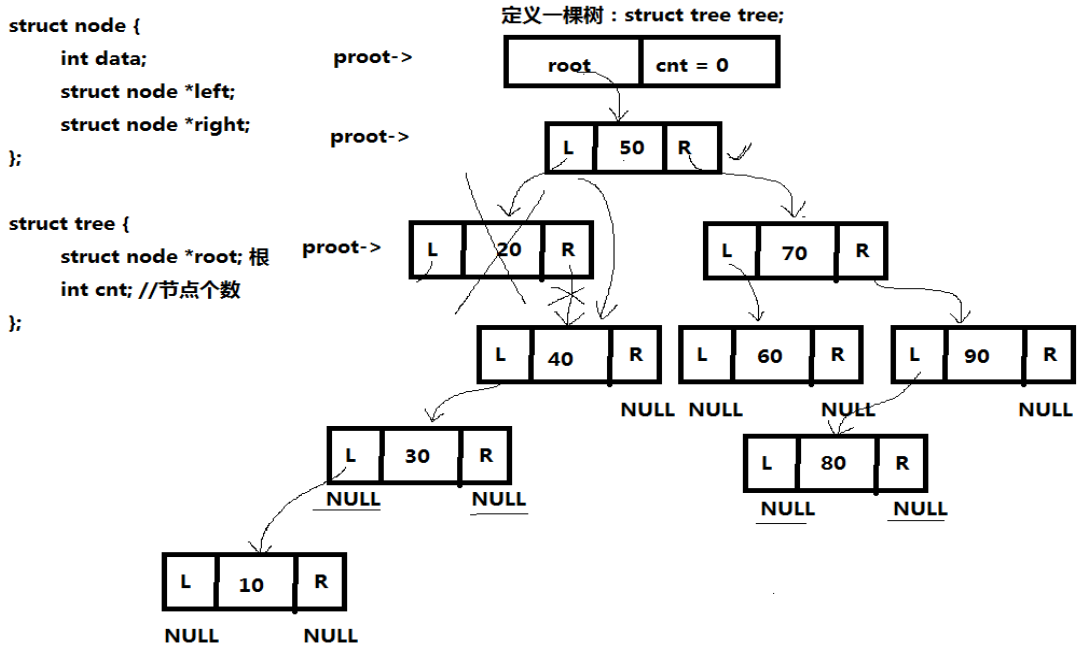
5. 注意：删除二叉树节点的三步骤
(1) 找到删除的节点
(2) 将要删除的节点的左子树放到要删除的节点的右子树上去
(3) 将要删除的节点的父节点的左子树的指针指向要删除的节点的右子树



Tree.h

```
 1 /*有序二叉树演示*/
 2 #ifndef __TREE_H
 3 #define __TREE_H
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 //声明描述节点的结构体
 7 typedef struct node {
 8         int data;
 9         struct node *left;
10         struct node *right;
11 }node_t;
12
13 //声明描述整课树的结构体
14 typedef struct tree {
15         struct node *root; //根节点地址
16         int cnt; //节点个数
17 }tree_t;
18
19 //声明操作函数
20 extern void insert_data(tree_t *tree, int data);//插入节点
21 extern void travel_data(tree_t *tree);//遍历
22 extern void clear_data(tree_t *tree); //清空
23 extern void del_data(tree_t *tree, int data); //删除
24 extern void modify_data(tree_t *tree, int old_data, int new_data); //需改
25 #endif
```

删除节点图示：



Tree.c

```
1 #include "tree.h"
2 //定义创建节点的函数
3 static node_t *create_node(int data){
4         //分配内存
5         node_t *p = (node_t *)malloc(sizeof(node_t));
6         //初始化新节点
7         p->data = data;
8         p->left = NULL;
9         p->right = NULL;
10        return p;
11 }
12 //定义递归函数:负责插入新节点向 root:*proot=root
13 static void insert(node_t **proot, node_t *pnode){
14 //递归函数的退出条件:"根节点"L,R 为空,如果为空,将新节点插入到"根节点"
      的 R 和 L
15        if(*proot == NULL) {
16                *proot = pnode;
17                return;
18        }
19        //插入到左子树
20        if((*proot)->data > pnode->data) {
```

```
21              insert(&(*proot)->left, pnode);
22              return;
23          } else {//插入到右子树
24              insert(&(*proot)->right, pnode);
25              return;
26          }
27 }
28 //定义向有序二叉树插入新节点函数
29 void insert_data(tree_t *tree, int data){
30          //1.创建新节点
31          node_t *pnode = create_node(data);
32
33          //2.调用递归函数将新节点 pnode 插入 tree 的 root
34          insert(&tree->root, pnode); //一开始从最开始的根节点 root(50)插入
35
36          //3.更新计数
37          tree->cnt++;
38 }
39 //定义递归函数:按照中序遍历方式打印节点的值
40 static void travel(node_t *proot){
41          if(proot != NULL) {
42              /*中序遍历*/
43              travel(proot->left); //处理左节点
44              printf("%d ", proot->data); //处理节点自己的数据
45              travel(proot->right); //处理右几点
46              /*
47                先序遍历
48              printf("%d ", proot->data); //处理节点自己的数据
49              travel(proot->left); //处理左节点
50              travel(proot->right); //处理右几点
51                */
52              /*
53                后序遍历
54              travel(proot->left); //处理左节点
55              travel(proot->right); //处理右几点
56              printf("%d ", proot->data); //处理节点自己的数据
57                */
58              return;
59          }
60          return;
61 }
62 //定义遍历函数
63 void travel_data(tree_t *tree){
64          //调用递归函数
```

```
65          travel(tree->root);
66          printf("\n");
67 }
68 //删除:20 70 50
69 /*
70 clear_data()
71          clear(50)
72                  clear(50 左边:20)
79                          free(70)
80                  free(50)
81  返回到 clear_data*/
82 //定义清除节点递归函数
83 void clear(node_t **proot){
84          if(*proot != NULL) {
85                  clear(&(*proot)->left); //清空左子树
86                  clear(&(*proot)->left); //清空右子树
87                  free(*proot);
88                  *proot = NULL;
89          }
90 }
91 //定义清空二叉树节点函数
92 void clear_data(tree_t *tree){
93        //调用递归函数实现清空节点
94        clear(&tree->root);
95        tree->cnt = 0;
96 }
97 /*
98 find_data(50, 5)
99          find(50, 5)
100                 find(20, 5)
101                         find(10, 5)
102                                 find(NULL, 5)
103    NUll; //没有找到
104
105 find_data(50, 10)
106         find(50, 10)
107                 find(20, 10)
108                     find(10, 10)
109                       return 10 节点的 2 级指针
110 10 节点 2 级指针  != NULL
111
112 find_data(50, 80)
113         find(50, 80)
114                 find(70, 80)
```

```
115                     find(90, 80)
116                         find(80, 80)
117                             return 80 节点的 2 级指针
118 80 节点 2 级指针  != NULL
119   */
120 //定义查找节点的递归函数
121 static node_t **find(node_t **proot, int data){
122         //1.如果"根节点"(L,R)为空,直接返回,没有找到
123         if(*proot == NULL)
124                 return proot; //返回 NULL
125         //2.比较要删除的节点和目标值,相等返回节点
126         if(data == (*proot)->data)
127                 return proot; //找到了
128         else if(data < (*proot)->data) {
129                 return find(&(*proot)->left, data); //左子树找
130         } else {
131                 return find(&(*proot)->right, data);//右子树找
132         }
133 }
134 static node_t **find_data(tree_t *tree, int data){
135         return find(&tree->root, data);
136 }
137 //定义删除节点函数
138 //先查找,后找爹,提一档
139 void del_data(tree_t *tree, int data){
140         //1.首先找到要删除的节点,返回这个节点的首地址
141         node_t **ppnode = find_data(tree, data);
142         if(*ppnode == NULL) {
143                 printf("要找的目标不存在.\n");
144                 return;
145         }
146         //2.如果要删除的节点有左子树,把左子树并到右子树
147         if((*ppnode)->left != NULL) {
148                 insert(&(*ppnode)->right, (*ppnode)->left);
149         }
150         //3.将要删除的节点的右子树给要删除的节点的父节点的左子树
151         node_t *ptmp = *ppnode; //暂存要删除节点的首地址
152         *ppnode = (*ppnode)->right;
153
154         //4.释放内存
155         free(ptmp);
156         ptmp = NULL;
157
158         //5.更新计数
```

```
159         tree->cnt--;
160 }
161 /*定义修改节点数值的函数*/
162 void modify_data(tree_t *tree, int old, int new)
163 {
164         //1.先删除节点
165         del_data(tree, old);
166         //2.插入新节点
167         insert_data(tree, new);
168 }
```

Main.c

```
  1 /*有序二叉树测试*/
  2 #include "tree.h"
  3
  4 int main(void)
  5 {
  6         //1.定义一个二叉树
  7         tree_t tree;
  8
  9         //2.初始化二叉树
 10         tree.root = NULL;
 11         tree.cnt = 0;
 12
 13         //3.添加新节点
 14         insert_data(&tree, 50);
 15         insert_data(&tree, 70);
 16         insert_data(&tree, 30);
 17         insert_data(&tree, 20);
 18         insert_data(&tree, 10);
 19         insert_data(&tree, 90);
 20         insert_data(&tree, 80);
 21         insert_data(&tree, 60);
 22         insert_data(&tree, 40);
 23
 24         //4.遍历打印
 25         travel_data(&tree);
 26
 27         //5.删除 40
 28         del_data(&tree, 40);
 29         del_data(&tree, 40);
 30         travel_data(&tree);
 31
 32         //6.修改
```

```
33          modify_data(&tree, 10, 250);
34          travel_data(&tree);
35
36          //7.清空
37          clear_data(&tree);
38          travel_data(&tree);
39          return 0;
40 }
```

Makefile

```
1 #定义变量
2 BIN=tree
3 OBJ=tree.o main.o
4
5 CROSS_COMPILE=
6 CC=$(CROSS_COMPILE)gcc
7
8 #制定编译规则
9 $(BIN):$(OBJ)
10              $(CC) -o $(BIN) $(OBJ)
11
12 %.o:%.c
13              $(CC) -c -o $@ $<
14
15 clean:
16              rm $(BIN) $(OBJ)
```

八、大名鼎鼎的代码调试工具：gdb(单步跟踪调试程序)

使用步骤：

1.编译程序时,一定要加-g 选项(调试选项)

例如：gcc -g -o helloworld helloworld.c

2.调程序需要启动 gdb 命令,立马出现 gdb 的命令行提示符:(gdb)

gdb 命令格式：gdb 可执行程序文件名

例如：gdb helloworld

3.掌握几个 gdb 的命令

(gdb)l //l=list:罗列源代码

(gdb)b 15 //b=breakpoint=断点:将来 CPU 执行到这个断点的位置就立刻停止不动,等待着下一个命令，b 15:就是在源码的第 15 行设置一个断点

(gdb)r //r=run,启动运行程序,CPU 就从 main 函数开始依次向下运行,直到遇到断点

(gdb)s //s=step,下一步,让 CPU 继续向下执行一条语句,如果碰到函数,会让 CPU 进入函数继续跟踪

如果输入 n(next),也是下一步,但是 CPU 不会进入函数内部跟踪,直接调用函数完毕

(gdb)p 变量名 //查看变量的值
(gdb)p /x 变量名 //查看变量的值,按照 16 进制格式输出
(gdb)q //quit,退出 gdb 命令

```
1 #include <stdio.h>
2 int add(int x, int y){
3         int z = 0;
4         z = x + y;
5         return z;
6 }
7 int main(void){
8         int a = 10;
9         int b = 20;
10        int c = 0;
11        c = add(a, b);
12        printf("c = %d\n", c);
13        return 0;
14 } //10:15 继续上课
15 /*使用步骤:gcc -g -o helloworld helloworld.c 一定添加-g 选项
16        1)调程序需要启动 gdb 命令,立马出现 gdb 的命令行提示符:(gdb)
17        gdb 命令格式: gdb 可执行程序文件名
18        例如: gdb helloworld
19        2)掌握几个 gdb 的命令
20        (gdb)l    //l=list:罗列源代码
21        (gdb)b    15 //b=breakpoint=断点:将来 CPU 执行到这个断点的位置就立刻停止不动,等待着下一个命令
22        b 15:就是在源码的第 15 行设置一个断点
23        (gdb)r    //r=run,启动运行程序,CPU 就从 main 函数开始依次向下运行,直到遇到断点
24        (gdb)s    //s=step,下一步,让 CPU 继续向下执行一条语句,如果碰到函数,会让 CPU 进>        入函数继续跟踪
25        如果输入 n(next),也是下一步,但是 CPU 不会进入函数内部跟踪,直接调用函数完毕
26        (gdb)p 变量名 //查看变量的值
27        (gdb)p /x 变量名 //查看变量的值,按照 16 进制格式输出
28        (gdb)q //quit,退出 gdb 命令
29    */
```

九、常用算法: 4 个排序算法和 2 个查找算法

1.冒泡排序算法

例如:

初始状态: 9 7 5 3 1
第一趟: 7 5 3 1 9(4)
第二趟: 5 3 1 7 9(3)
第三趟: 3 1 5 7 9(2)

第四趟：1 3 5 7 9(1)
目前来看：N 个数需要 N-1 趟排序,每趟需要 N-1 次比较
第一步优化：N 个数需要 N-1 趟排序,每趟需要 N-1-i 次比较
第二步优化：例如：
初始状态：0 7 5 3 1
第一趟: 0 5 3 1 7
第二趟: 0 3 1 5 7
第三趟: 0 1 3 5 7 *
如果发现某一趟的排序中,没有发生数据的交互,那么就认为排序完成

2.插入排序算法
例如：
初始状态:10 30 20 15 5 //无序的
原始想法：5 10 15 20 30 //有序的
原始想法思路：从无序的数列中取出数据单独存放成一个有序的数列
弊端：需要额外开辟内存空间来存放有序数据列
解决办法：原地完成插入,不需要额外开辟大量的内存空间
原地实现插入流程：
初始状态:10 30 20 15 5 //无序的
定义一个变量保存要被插入的数据,从第二个数据开始,思路：
5 10 15 20 30
优点：节省了内存空间,只需一个变量保存被插入的数据即可原地完成排序

3.选择排序算法：
例如：
初始状态：9 5 7 3 1 //无序
原始想法：1 3 5 7 9 //有序
思路：从一个数列中找最小数放到第一个位置,
然后从剩余的无序数列中再找一个最小数
放在无序的第一个位置,依次排下去
弊端：需要额外开辟内存空间来存放有序数据列
解决办法：
初始状态: 9 5 7 3 1
原地解决：1 5 7 3 9
————
1 3 7 5 9
——
1 3 5 7 9
——
1 3 5 7 9

4. 快速排序算法：

例如：

0 10 80 30 60 50 40 70 20 90 100

思路：不管站在哪个数上去看,它左边的数小于它,它右边的数据大于它(跟有序二叉树一样)。

原始实现：站在 50 来看,进行一次分组得到,把比 50 小的放左边,大的放右边：

0 10 30 40 20 50 60 70 90 80 100 //更接近有序

然后再对 50 左边的数进行再次分组,把比 30 小的放左边,大的放右边,

再然后对 50 右边的数进行再次分组,把比 90 小的放左边,大的放右边：

0 20 10 30 40 50 60 80 70 90 100 //又接近有序

以此类推,对 30 左边右边进行各种分组,对 90 左边和右边各种分组

直到分到没有数或者只有一个数

弊端：需要额外开辟内存空间来存放不断趋近有序数列

0 10 80 30 60 50 40 70 20 90 100

50 基准 0 10 30 20 40 50 60 70 90 80 100

30,90 基准 0 20 10 30 40 50 60 80 70 90 100

20,80 基准 0 10 20 30 40 50 60 70 80 90 100

10,70 基准 0 10 20 30 40 50 60 70 80 90 100

问：能够原地实现分组么?

答：定义三个游标来实现原地分组(p=pivot=基准)

初始：0 10 80 30 60 50 40 70 20 90 100

i

p

j

以 50 基准进行一次(将来定义变量来暂存):结果

0 10 20 30 40 50 80 70 60 90 100

i

p

j

参考代码：sort.h sort.c main.c Makefile

```
1 /*sort.c:排序算法函数定义*/
2 //冒泡
3 void bubble_sort(int data[], int size){
4         int i, j;
5         for(i = 0; i < size - 1; i++) { //趟次
6                 int ordered = 1; //判断是否发生交换
7                 for(j = 0; j < size - 1 - i; j++) { //一趟排序
8                         if(data[j+1] < data[j]) {
9                                 int swap = data[j];
10                                data[j] = data[j+1];
11                                data[j+1] = swap;
12                                ordered=0; //只要交换,给 0
13                        }
14                }
```

```
15              if(ordered)
16                  break; //跳出外层 for 循环,后续趟次不用跑
17          }
18 }
19 //插入:10 30 20 15 5
20 void insert_sort(int data[], int size){
21          int i;
22          for(i = 1; i < size; i++) {
23              int inserted = data[i]; //临时保存被插入的数据
24              int j;
25              for(j = i; j > 0 && inserted < data[j-1]; --j)
26                  data[j] = data[j-1];
27              if(j != i) //如果前面的数都比被插入的数要小
28                  data[j] = inserted;
29          }
30 }
31 //选择排序:9 5 7 3 1
32 void select_sort(int data[], int size){
33      int i;
34      for(i = 0; i < size - 1; i++) { //找最小数
35              int min = i; //假设无序数列中的第 1 个数据是最小的
36              int j;
37              for(j = i + 1; j < size; j++) { //找最小数
38                  if(data[j] < data[min])
39                      min = j; //重新更新最小数的下标
40              }
41              if(min != i) { //然后交换
42                  int swap = data[i];
43                  data[i] = data[min];
44                  data[min] = swap;
45              }
46      }
47 }
48 //快速排序
49 void quick_sort(int data[], int left, int right) {
50          int p = (left+right)/2; //随意定义基准下标
51          int pivot = data[p]; //获取基准值
52          int i = left; //left=0
53          int j = right; //right=size-1
54          //递归退出条件
55          while(i < j) { //条件成立即可分组
56              //操作左边
57              for(; !(i >= p || pivot < data[i]); i++); //空循环,让 i++,i++不加加的
    前提是 i=p 重合,或者 i 对应的值大于基准值
```

```
58              if(i < p) { //此代码表示上面的循环是因为 i 对应的值大于基准值退
                            出的
59                  data[p] = data[i]; //放到 p 位置
60                  p = i; //p 和 i 重合
61              }
62              //同理操作右边
63              for(; !(j <= p || data[j] < pivot); j--); //空循环
64              if(j > p) {
65                  data[p] = data[j];
66                  p = j;
67              }
68          }
69          data[p] = pivot; //i,j,p 重合后将基准值放到 p 位置
70          if(p - left > 1) { //左边至少两个数,重复分组
71              quick_sort(data, left, p-1); //不用包含 p
72          }
 73          if(right - p > 1) { //右边至少两个数,重复分组
74              quick_sort(data, p+1, right); //不用包含 p
75          }
76 }
```

Main.c

```
1 /*main.c:排序算法测试*/
 2 #include "sort.h"
 3
 4 int main(void)
 5 {
 6          int data[] = {9, 0, 7, 2, 5, 4, 3, 6, 1, 8};
 7          int size = sizeof(data)/sizeof(data[0]);
 8
 9          //冒泡排序
10          //bubble_sort(data, size);
11          //插入排序
12          //insert_sort(data, size);
13          //选择排序
14          //select_sort(data, size);
15          //快速排序
16          quick_sort(data, 0, size -1);
17          //打印排序的结果
18          for(int i = 0; i < size; i++)
19              printf("%d ", data[i]);
20          printf("\n");
21          return 0;
22 }
```

5.查找算法：线性查找,二分查找(折半查找)
(1)线性查找：在数列中挨个匹配要找的数据,对数据的有序性无要求
(2)二分查找(折半查找)
切记：查找的前提是数据必须有序,否则用不了
原理：目标找 60
10 20 30 40 50 60 70
对数列对半分,然后拿 60 跟 40 比较,如果相等,说明找到了,如果不相等,发现 60 大于 40,说明 60 肯定在 40 的右边,然后对 40 右边的 50 60 70 再对半分,然后拿 60 跟中间的 60 进行比较,发现相等,说明找到了,如果不相等并且小于在左边继续对半找,如果不相等并且大于在右边继续对半找。
参考代码：find.h find.c main.c Makefile

Find.c

```
#include "find.h"
//定义线性查找函数
int line_find(int data[], int size, int key){
    for(int i = 0; i < size; i++)
        if(data[i] == key)
            return i; //返回下标
    return -1; //没有找到
}
/*递归二分查找*/
static int recu_find(int data[], int left, int right, int key) {
    if(left <= right) {
        int mid = (left+right)/2;
        if(key < data[mid])
            return recu_find(data, left, mid-1, key);
        else if(data[mid] < key)
            return recu_find(data, mid+1, right, key);
        else
            return mid;
    }
    return -1;
}

//定义二分查找函数
int half_find(int data[], int size, int key) {
    //1.采用递归实现
    return recu_find(data, 0, size-1, key);

    //2.采用常规方法
    //定义两个游标
    int left = 0;
```

```
31      int right = size -1;
32
33      while(left <= right) {
34          int mid = (left+right)/2; //折半
35          if(key < data[mid]) //左边找去
36              right = mid-1; //中间值不需要
37          else if(data[mid] < key) //右边找去
38              left = mid + 1; //中间值不需要
39          else
40              return mid; //找到并且返回下标
41      }
42      return -1; //没有找到
43 }
```

Find.h

```
1 #ifndef __FIND_H
2 #define __FIND_H
3 #include <stdio.h>
4 /*声明线性查找函数*/
5 extern int line_find(int data[], int size, int key);
6 /*声明二分成查找函数*/
7 extern int half_find(int data[], int size, int key);
8 #endif
```

Main.c

```
1 #include"find.h"
2 #include"sort.h"
3 int main()
4 {
5       int data[]={9,0,7,2,5,4,3,6,1,8};
6       int size = sizeof(data)/sizeof(data[0]);
7       int key = 7;
8       int index = 0;
9       //1.线性查找 7，找到返回数字所在的下标
10      index = line_find(data,size,key);
11      if(index == -1){
12          printf("没有找到%d\n",key);
13          return -1;
14      }
15      printf("data[%d] = %d\n",index,data[index]);
16      //2.快速排序
17      quick_sort(data,0,size-1);
18      //3.二分查找
19      index = half_find(data,size,key);
```

```
20      if(index == -1){
21          printf("没有找到%d\n",key);
22          return -1;
23      }
24      printf("data[%d] = %d\n",index,data[index]);
25      return 0;
26 }
```