230704——数据结构

day1

```
数据结构：数据与数据之间的关系
逻辑结构：
    根据数据的直接前驱数和直接后继数分为：
      线性结构 和 非线性结构
存储结构：
    顺序存储、链式存储 、 索引存储、 散列存储
运算：增、删、改、查、...
```

2.1

```
逻辑结构：
     线性结构 和非线性结构
线性结构：
     线性表、栈、队列
     线性表：
         顺序表、链表
 非线性结构：
   树、图
线性表的特点：
   表头无前驱、表尾无后继；
   其他元素有且仅有一个直接前驱和直接后继；
```

顺序表

```
逻辑结构：线性
存储结构：顺序存储

抽象顺序表：
  #define SIZE 100
  typedef int data_t;
  typedef struct{
    data_t buf[SIZE];//数据
    int index;//表尾指针
  }sqlist_t;
运算：
  创建表
  sqlist_t *sqlistCreate();
  判空
  int sqlistIsEmpty(sqlist_t *sqlist);
  判满：
```

```
  int sqlistIsFull(sqlist_t *sqlist);
  按照位置插入:
int sqlistInsertPos(sqlist_t *sqlist, int pos, data_t data)
 求长度:
  int sqlistGetLength(sqlist_t *sqlist);
  打印顺序表
  void SqlistShow(sqlist_t *sqlist);

//按位置删除
int sqlistDeletePos(sqlist_t *sqlist, int pos);
//按数据查找
int sqlistSearchData(sqlist_t *sqlist, data_t data);
//更新
int sqlistUpdataData(sqlist_t *sqlist, data_t olddata, data_t newdata);
//表的清空
int sqlistClear(sqlist_t *sqlist);
//表的销毁
int sqlistDestroy(sqlist_t **sqlist);

//按数据删除
void sqlistDeleteData(sqlist_t *sqlist, data_t data);

//按位置查找
data_t sqlistSearchPos(sqlist_t *sqlist, int pos);
//按位置更新
int sqlistUpdataPos(sqlist_t *sqlist, int pos, data_t data);
//排序
void sqlistSort(sqlist_t *sqlist);

顺序表的优缺点:
  缺点:
   1.可能会存在浪费空间的情况或者空间不够;
   2.在删除或者插入数据的时候，数据会发生成片的移动，效率低;
  有点:
     查找很方便;


```

链表

```
逻辑线性;
存储: 链式存储

抽象数据结构:
  typedef int data_t;
typedef struct Node{
   data_t data; //数据域
   struct Node *next; //指针域
```

```
}link_t;

1.创建一个链表
  link_t *linkCreate()
{
   link_t * head = (link_t *)malooc(sizeof(link_t));
   if(head == NULL){
      return NULL;
   }
   head->data = -1;
   head->next = NULL;

   return head;
}
2.int linkGetLength(link_t *head)
{
  if(head == NULL){
    return -1;
  }
   int len = 0;
   link_t *p = head->next;
   while(p != NULL){
      len++;
      p = p->next;
  }
  return len;
}

3.判空
  int linkIsEmpty(link_t *head)
{
  if(head == NULL){
      return -1;
  }
  return head->next == NULL;
}

4.遍历链表
  void linkShow(link_t *head)
{
  if(head == NULL){
    return;
  }
  link_t *p = head->next;
  while(p != NULL){
    printf("%d ", p->data);
    p = p->next;
  }
}
5.按位置插入
  int linkInsertPos(link_t *head, int pos, data_t
data)
```

```c
{
  //1. 判断链表是否存在
  if(head == NULL){
    return -1;
  }
  //2.判断位置是否合法
  len = linkGetLength(head);
  if(pos < 0 || pos > len){
    printf("pos error\n");
    return -1;
  }
  //3.找到要插入位置的前一个节点
  link_t *p = head;
  int i = 0;
  for(; i < pos; i++){
    p = p->next;
  }
  //4.创建新节点
  link_t *new = (link_t *)malloc(sizeof(link_t));
  if(new == NULL){
    printf("malloc failed...\n");
    return -1;
  }
  new->data = data;
  //插入：
  //让新节点指向后一个节点
  new->next = p->next;
  //让前一个节点指向新节点
  p->next = new;

  return 0;

}
6.按位置删除
  int linkDeletePos(link_t *head, int pos)
{
  //1.判断链表是否为空
  if(linkIsEmpty(head)){
    printf("head is empty\n");
    return -1;
  }
  //2.判断位置是否合法
  int len = linkGetLength(head);
  if(pos < 0 || pos > len-1){
    printf("pos error\n");
    return -1;
  }
  //3.删除
    //3.1 找到要删除的前一个节点
    int i = 0;
  link_t *p = head;
  for(; i < pos; i++){
```

```
    p = p->next;
  }
  //保存要删除的节点的地址
  link_t *q = p->next;
  p->next = q->next;
  free(q);
  q = NULL;
  return 0;
}
7.按数据查找
 link_t *linkSearchData(link_t *head, data_t data)
{
  if(head == NULL){
    return -1;
  }
  link_t *p = head->next;
  while(p != NULL){
    if(p->data == data){
      return p;
    }
    p = p->next;
  }
  return NULL;
}
8.int linkUpdateData(link_t *head, \
                        data_t olddata, data_t
newdata)
{
  if(head == NULL){
      return -1;
  }
  link_t *p = linkSearchData(head, olddata);
  p->data = newdata;
  return 0;
}
9.void linkClear(link_t *head)
{
  while(!linkIsEmpty(head)){
    linkDeletePos(head, 0);
  }
  return;
}
10.void linkDestroy(link_t **head)
{
  linkClear(*head);
  free(*head);
  *head = NULL;

  return;
}

链表的优缺点:
```

```
 插入删除比较方便，不用考虑预分配空间的问题
缺点：
 访问效率低，单独分配一个指针的空间保存下一个节点的地址
```

单向循环链表

```
typedef struct Node{
   data_t data;
    struct Node *next;
}link_t;
link_t * cir_linkCreate()
{
  link_t *head = (link_t *)malloc(sizeof(link_t));
  if(head == NULL)
  {
    printf("malloc failed....\n");
    head->data = -1;
    head->next = head;
  }
  return head;
}

约瑟夫环问题？
```

双向链表

```
typedef int data_t;
typedef struct Node{
   data_t data;
  struct Node *next;
  struct Node *prior;
}link_t;
//1.创建一个双向链表
link_t * D_linkCreate()
{
  link_t *head = (link_t *)malloc(sizeof(link_t));
  if(head == NULL){
      printf("malloc failed...\n");
      return NULL;
  }
  head->data = -1;
  head->next = NULL;
  head->prior = NULL;

  return head;

}
```

栈

```
堆栈简称栈：
  线性结构；
  定义：只允许一端进行插入或者删除的线性表;允许插入或者删除的一端
       称为栈顶,另一端称为栈底；
  特点:后进先出；
抽象数据结构：
  顺序栈：
  #define SIZE 100
  typedef int data_t;
typedef struct{
    data_t data[SIZE];
    int top; // 栈顶"指针"
}stack_t;

运算：
   //1.创建栈
   //2.入栈
   //3.出栈
   //4.获取栈顶元素
   //5.判空
   //6.判满
   //7.获取长度

   // 遍历
   //清空
   //销毁
```

链式栈

```
typedef int data_t;
typedef struct Node{
    data_t data;
    struct Node *next;
}lstack_t;
将链表的头节点作为栈顶，方便入栈或者出栈；

运算：
   //1.创建栈
   //2.入栈
   //3.出栈
   //4.获取栈顶元素
   //5.判空
   //6.判满
   //7.获取长度

```

```
    // 遍历
    //清空
    //销毁
```

队列

```
允许在两端进行插入或者删除的线性表称为队列;允许插入的一端称为队尾;
允许删除的一端称为队头;
特点:
     先进先出;

  逻辑关系: 线性
  存储:
     顺序存储 顺序队
     链式存储 链队
 顺序队列:
    抽象数据结构:
     typedef int data_t;
    typedef struct{
          data_t data[100];
          int front; // 队头"指针"
          int rear;  //队尾 "指针"
    }sq_t;

//1.创建队列
//2.判空
//3.判满
//4.队列的长度
//5.入队
//6.出队
//7.清空队列
//8.销毁

```

链式队列

```
抽象数据结构:
  typedef int data_t ;
  typedef struct Node{
       data_t data;
       struct Node *next;
  }link_t;

typedef struct{
    link_t *front;
    link_t *rear;
}lqueue_t;
```