

# 数据结构和算法代码汇总

alpha0x00

leetiankai@gmail.com

2020 年 9 月 16 日

本文档来自于对 HarvestWu 的《数据结构算法汇总》文档的整理和完善，代码严格按照 C99 标准来书写，并不会 C++ 代码的任何特性（包括引用）。

# 目录

<b>1 线性表</b>	<b>3</b>
1.1 数据结构定义	3
1.1.1 顺序表	3
1.1.2 单链表	6
1.1.3 双链表	7
1.1.4 静态链表	10
1.2 例题	11
<b>2 栈</b>	<b>14</b>
2.1 数据结构定义	14
2.1.1 顺序存储的栈	14
2.1.2 链式存储的栈	17
2.2 例题	18
<b>3 队列</b>	<b>19</b>
3.1 数据结构定义	19
3.1.1 顺序存储的队列	19
3.1.2 链式存储的队列	21
3.2 例题	23
<b>4 树</b>	<b>23</b>
4.1 数据结构定义	23
4.1.1 双亲存储结构	23
4.1.2 孩子兄弟存储结构	27
4.1.3 二叉树、二叉排序数和平衡二叉树	27
4.1.4 线索二叉树	33
4.2 例题	38
<b>5 广义表 *</b>	<b>53</b>
5.1 数据结构定义	53
5.2 例题	55

<b>6 图</b>	<b>56</b>
6.1 数据结构定义	56
6.2 例题	57
<b>7 排序和查找</b>	<b>57</b>
7.1 排序	57
7.1.1 插入排序	57
7.1.2 折半插入排序	57
7.1.3 希尔排序 *	57
7.1.4 冒泡排序	58
7.1.5 快速排序	59
7.1.6 选择排序	59
7.1.7 堆排序 *	60
7.1.8 归并排序	60
7.1.9 基数排序 *	62
7.2 查找	63
7.2.1 顺序查找	63
7.2.2 二分查找	63

# 1 线性表

## 1.1 数据结构定义

### 1.1.1 顺序表

**静态分配** 静态分配是指初始化链表时内部不需要来自头文件 `<stdlib.h>` 的 `malloc` 或 `calloc` 函数动态分配空间。其结构定义如下：

---

```

1  #define MaxSize 1024
2  typedef struct SeqList {
3      int data[MaxSize];
4      int length; /* 记录存储元素的个数 */
5  } SeqList;

```

---

**动态分配** 相对于静态分配，初始化链表时需要 `malloc` 或 `calloc` 函数。

---

```

1  typedef struct SeqList {
2      int *data;
3      int length, capacity; /* capacity 记录顺序表可存储元素最大个数 */
4  } SeqList;

```

---

### 基本操作

1. 初始化顺序表 `bool InitList(SeqList *list, int capacity)`；成功返回 `true`，失败返回 `false`。注意：下面没有使用 `list->data = (int *)malloc(capacity * sizeof(int))` 风格，即强制转换 `malloc` 返回值类型为 `int *` 类型，原因参见<sup>1</sup>。

---

```

1  /**
2   * bool 在 C99 以标准类型添加到 C 语言中，来自头文件 <stdbool.h>
3   */
4  bool InitList(SeqList *list, int capacity)
5  {
6      if (NULL == list || capacity <= 0) return false;
7
8      list->data = malloc(capacity * sizeof(int));
9      if (NULL == list->data) return false;
10
11     list->length = 0;
12     list->capacity = capacity;
13     return true;
14 }

```

---

<sup>1</sup><https://stackoverflow.com/questions/605845/do-i-cast-the-result-of-malloc>

如果是静态分配的顺序表，初始化函数可以按照如下定义：

---

```
1 bool InitList(SeqList *list)
2 {
3     if (NULL == list) return false;
4     list->length = 0;
5     return true;
6 }
```

---

2. 销毁顺序表 `void DestroyList(SeqList *list)`；对于静态顺序表，可以没有销毁操作，或者简单地把 `list->length` 置为 0 即可

---

```
1 void DestroyList(SeqList *list)
2 {
3     free(list->data);
4 }
```

---

3. 求顺序表长度 `int ListLength(SeqList *list)`；

---

```
1 int ListLength(SeqList *list)
2 {
3     if (NULL == list) return 0;
4     return list->length;
5 }
```

---

4. 获取顺序表容量 `int ListCapacity(SeqList *list)`；

---

```
1 int ListCapacity(SeqList *list)
2 {
3     if (NULL == list) return 0;
4     return list->capacity;
5     /* 对于静态顺序表 */
6     //return MaxSize;
7 }
```

---

5. 顺序表是否为空 `bool ListEmpty(SeqList *list)`；

---

```
1 bool ListEmpty(SeqList *list)
2 {
3     return (0 == list->length);
4 }
```

---

6. 顺序表是否已满 `bool ListFull(SeqList *list)`；

---

```
1 bool ListFull(SeqList *list)
2 {
```

---

```

3     return (ListLength(list) == ListCapacity(list));
4 }

```

---

7. 查找某个元素的下标 `int LocateElem(SeqList *list, int ele);`

```

1  /**
2   * 返回 -1 表明没有找到或者出错
3   */
4  int LocateElem(SeqList *list, int ele)
5  {
6      if (NULL == list) return -1;
7      for (int i = 0; i < Length(list); i++) {
8          if (ele == list->data[i])
9              return i;
10         /* 对于静态顺序表 */
11         //if (ele == list.data[i])
12         //    return i;
13     }
14     /* 没有找到 */
15     return -1;
16 }

```

---

8. 根据下标获取元素 `int GetElem(SeqList *list, int idx);`

```

1  int GetElem(SeqList *list, int idx)
2  {
3      /* 假定元素都是大于 0, 返回 -1 表明发生下标越界错误
4       * 如果存储的元素是任意整数, 那么返回 -1 就不合理, 可以修改 GetElem 函数为
5       * bool GetElem(SeqList *list, int idx, int *store);
6       * 返回 true, 表明正确获取数据, 存储到指针 *store 里,
7       * 返回 false, 表明下标越界
8       */
9      if (idx < 0 || idx > ListLength(list))
10         return -1;
11     return list->data[idx];
12     /* 对于静态顺序表 */
13     //return list.data[idx];
14 }

```

---

9. 在指定下标处插入元素 `bool ListInsert(SeqList *list, int idx, int ele);`

```

1  bool ListInsert(SeqList *list, int idx, int ele)
2  {
3      if (idx < 0 || idx > ListLength(list)) return false;
4      /* 存储空间已满 */
5      if (ListFull(list)) return false;
6      for (int i = Length(list); i > idx; i--) {
7          list->data[i] = list->data[i-1];
8          /* 对于静态顺序表 */
9          // list.data[i] = list.data[i-1];

```

```

10     }
11     list->data[idx] = ele;
12     return true;
13 }

```

10. 删除下标处的元素 `bool ListDelete(SeqList *list, int idx, int *ele);`

```

1  bool ListDelete(SeqList *list, int idx, int *ele)
2  {
3      if (idx < 0 || idx > ListLength(list)) return false;
4      for (int i = idx; i < ListLength(list)-1; i++) {
5          list->data[i] = list->data[i+1];
6          /* 对于静态顺序表 */
7          // list.data[i] = list.data[i+1];
8      }
9      list->length = list->length-1;
10     return true;
11 }

```

### 1.1.2 单链表

```

1  typedef struct LinkNode {
2      int data;
3      struct LinkNode *next;
4  } LinkNode;
5  typedef LinkNode *LinkList;
6
7  /* 分配一个节点 */
8  LinkList node = malloc(sizeof(LinkNode));
9  node->next = NULL;

```

#### 基本操作

1. 初始化一个单链表 `bool InitList(LinkList *list);` 单链表常常有两种组织方式，带有头节点和不带头节点。带有头节点的单链表很多操作可以方便一下，不带头节点的单链表虽然也可以向带有头节点的一样，但是要稍稍繁琐一点。

- 带头节点的单链表初始化

```

1  /**
2   * 带头节点的初始化
3   * LinkList 实际上是指向 LinkNode 类型的指针
4   * 所以 list 指针也就是一个指向 LinkNode 类型的 “二级指针”，
5   * 也就是所谓的指向指针的指针，list 指向 LinkList 类型的指针
6   */
7  bool InitList(LinkList *list)
8  {

```

---

```

9     if (NULL == list) return false;
10    *list = malloc(sizeof(LinkNode));
11    (*list)->next = NULL;
12    return true;
13 }

```

---

- 不带头节点的单链表初始化

---

```

1  /**
2   * 不带头结点的单链表初始化
3   */
4  bool InitList(LinkList *list)
5  {
6      if (NULL == list) return false;
7      *list = NULL;
8      return true;
9  }
10 /**
11  * 对于不带头节点的单链表也可以不使用函数来初始化链表，而是按照如下方式
12  * 因为无头节点，直接以空指针当作空链表
13  */
14  LinkList list = NULL;

```

---

2. 销毁单链表，释放节点占据的空间 `void DestroyList(LinkList list);`

---

```

1  /**
2   * 无论有没有头节点，释放方法都一样
3   */
4  void DestroyList(LinkList list)
5  {
6      LinkList next;
7      for (; NULL != list; list = next) {
8          next = list->next;
9          free(list);
10     }
11 }

```

---

### 1.1.3 双链表

双链表不像单链表有带头节点和不带头节点之分，因为不带头节点处理链表通常更繁琐一些，双链表插入删除涉及的指针数量是单链表的两倍，所以双链表实际中一般都是带有头节点。此外，双链表有普通的双链表和循环双链表两种实现。

---

```

1  typedef struct DLinkNode {
2      int data;
3      struct DLinkNode *prev, *next;
4  } DLinkNode;
5  typedef DLinkNode *DLinkList;

```

---



**基本操作** 这里以循环双链表为主要例子作为演示，同时注释里会注明普通双链表的实现。

1. 初始化双链表 `bool InitDLinkedList(DLinkedList *dlist);`

---

```

1  bool InitDLinkedList(DLinkedList *dlist)
2  {
3      if (NULL == dlist) return false;
4      *dlist = malloc(sizeof(DLinkedList));
5      if (NULL == *dlist) return false;
6      /* 循环双链表 */
7      (*dlist)->prev = (*dlist)->next = *dlist;
8      /* 普通双链表 */
9      // (*dlist)->prev = (*dlist)->next = NULL;
10     return true;
11 }

```

---

按照如下方式定义双链表并初始化：

---

```

1  DLinkedList list;          /* 定义双链表 list */
2  InitDLinkedList(&list);    /* 初始化双链表 list */

```

---

2. 判断是否是空链表 `bool DLinkedListEmpty(DLinkedList list);`

---

```

1  bool DLinkedListEmpty(DLinkedList list)
2  {
3      if (NULL == list) return true;
4      return (list->next == list) && (list->prev == list);
5      /* 普通双链表 */
6      //return (list->next == NULL) && (list->prev == NULL);
7  }

```

---

3. 求表长度 `int DLinkedListLength(DLinkedList list);`

---

```

1  int DLinkedListLength(DLinkedList list)
2  {
3      if (NULL == list) return 0;
4      int len = 0;
5      /* 让 p = list->next 是为了去掉头节点 */
6      for (DLinkedList p = list->next; list != p; p = p->next)
7          len++;
8
9      /* 普通双链表 */
10     //for (DLinkedList p = list->next; NULL != p; p = p->next)
11         // len++;
12
13     return len;
14 }

```

---

4. 在双链表指定节点后插入元素 `bool InsertDLinkedListAt(DLinkedList at, int ele);`

- 在循环双链表中插入元素

---

```

1  /**
2   * 插入元素 ele 到插入点 at 之后
3   */
4  bool InsertDLinkedListAt(DLinkedList at, int ele)
5  {
6      if (NULL == at) return false;
7      /* 分配好新插入的节点 node */
8      DLinkedListNode *node = malloc(sizeof(DLinkedListNode));
9      if (NULL == node) return false;
10     node->data = ele;
11
12     DLinkedList next = at->next;
13     /**
14      * 插入后节点构成顺序 at, node, next
15      * 所以上述语句 next = at->next, 是为了获取 node 的后继节点
16      * 然后直接连接起来即可
17      */
18     node->next = next;
19     node->prev = at;
20     next->prev = node;
21     at->next = node;
22     return true;
23 }

```

---

- 在非循环双链表中插入元素

---

```

1  /**
2   * 插入元素 ele 到插入点 at 之后
3   */
4  bool InsertDLinkedListAt(DLinkedList at, int ele)
5  {
6      if (NULL == at) return false;
7      /* 分配好新插入的节点 node */
8      DLinkedListNode *node = malloc(sizeof(DLinkedListNode));
9      if (NULL == node) return false;
10     node->data = ele;
11
12     DLinkedList next = at->next;
13     /**
14      * 插入后节点构成顺序 at, node, next
15      * 所以上述语句 next = at->next, 是为了获取 node 的后继节点
16      * 然后直接连接起来即可
17      */
18     node->next = next;
19     node->prev = at;
20     /**
21      * 当 at 就是最后一个节点时, at->next 肯定是空指针
22      * 所以没有“后继”节点指向 node
23      */

```

---

---

```

24     if (NULL != next)
25         next->prev = node;
26     at->next = node;
27     return true;
28 }

```

---

#### 5. 删除双链表中指定节点 `bool DeleteDLinkedList(DLinkedList node);`

- 从循环双链表中删除元素

---

```

1  bool DeleteDLinkedList(DLinkedList node)
2  {
3      if (NULL == node) return false;
4      /* 获取被删除节点的前驱和后继 */
5      DLinkedList prev = node->prev;
6      DLinkedList next = node->next;
7      /* 连接前驱和后继元素 */
8      prev->next = next;
9      next->prev = prev;
10     /* 释放被删除的节点 */
11     free(node);
12     return true;
13 }

```

---

- 从非循环双链表中删除元素

---

```

1  bool DeleteDLinkedList(DLinkedList node)
2  {
3      if (NULL == node) return false;
4      /* 获取被删除节点的前驱和后继 */
5      DLinkedList prev = node->prev;
6      DLinkedList next = node->next;
7      /**
8       * 连接前驱和后继元素
9       * 注意：如果 node 是第一个元素，那么 prev 是空指针 NULL
10      * 同理，如果 node 是最后一个元素，那么 next 是 NULL
11      */
12     if (NULL != prev)
13         prev->next = next;
14     if (NULL != next)
15         next->prev = prev;
16     /* 释放被删除的节点 */
17     free(node);
18     return true;
19 }

```

---

#### 1.1.4 静态链表

静态链表（或者叫「游标」的技术）在 C 语言中使用的比较少，但是对于没有指针概念的编程语言中使用会使用。与之类似且在 C 语言中使用较多的技术，叫「池」。比如链表节点池：

为了减少 malloc 函数的调用以提高效率，预先分配一大块内存，自己在这块内存中分配各个节点。指向这些节点的时候可以不使用指针，而是使用节点在这一大块节点池中的编号。

## 1.2 例题

1. 删除不带头结点单链表 list 中所有值为 x 的结点

---

```

1  #include "list.h"
2  void DeleteAllxFromList(LinkList *plist, int x)
3  {
4      /**
5       * 使用二级指针后代码虽然简洁，但是理解有点复杂，尝试画图理解吧
6       * 二级指针的思想就是：把节点想象成汤圆，指针就是用来插汤圆的牙签，
7       * 二级指针就是握住牙签的手
8       * 空链表：就是一根光溜溜的牙签
9       */
10     if (NULL == plist) return;
11
12     while (*plist != NULL) {
13         /**
14          * plist 是手，*plist 就是手上的牙签，(*plist)->next 下一个汤圆的牙签
15          * 让“手”握住下一个“汤圆”的“牙签”
16          */
17         LinkList *pNext = &(*plist)->next; /* -> 优先级高于 & */
18         if (x == (*plist)->data) {
19             free(*plist);
20             *plist = *pNext;
21         } else {
22             plist = pNext;
23         }
24     }
25 }
26

```

---

2. 删除带头结点单链表 list 中所有值为 x 结点

---

```

1  #include "list.h"
2  void DeleteAllxFromList2(LinkList list, int x)
3  {
4      if (NULL == list || NULL == list->next) return;
5      while (NULL != list->next) {
6          LinkList curr = list->next;
7          if (x == curr->data) {
8              free(curr);
9              list->next = curr->next;
10         } else {
11             list = list->next;
12         }
13     }
14 }

```

---

## 3. 反向输出带有头结点单链表 list 中的所有值

- 方法一：函数递归输出，由于链表长度可以很长，所以存在栈溢出的问题

---

```

1  #include "list.h"
2  /**
3   * 注意这里带有头节点
4   */
5  void _PrintListReversely(LinkList list)
6  {
7      /* 按照没有头节点单链表来处理 */
8      if (NULL == list) return;
9      _PrintListReversely(list->next);
10     printf("%d ", list->data);
11 }
12 void PrintListReversely(LinkList list)
13 {
14     /* 对于空链表 (带有头节点) 直接返回不进行处理 */
15     if (NULL == list || NULL == list->next) return;
16     list = list->next; /* 去除头节点 */
17     _PrintListReversely(list);
18 }

```

---

- 方法二：用一个数组暂存元素然后逆向输出（就是后面会讲到的栈的思想）

---

```

1  void PrintListReversely(LinkList list)
2  {
3      if (NULL == list || NULL == list->next) return;
4      list = list->next; /* 去除头节点 */
5      /* 存储元素的地址，减少直接复制元素带来的开销 */
6      int values[MaxSize];
7      int cnt;
8      for (cnt = 0; cnt < MaxSize && NULL != list; list = list->next) {
9          values[cnt] = list->data;
10         cnt++;
11     }
12     if (cnt > MaxSize) {
13         printf("the amount of data in list is greater than MaxSize(%d).",
14             MaxSize);
15         return;
16     }
17     /* 反向输出 */
18     while (cnt-- > 0)
19         printf("%d ", values[cnt]);
20 }

```

---

## 4. 删除带头结点单链表 list 中最小值的节点

---

```

1  #include "list.h"
2  void DeleteMinimalValueFromList(LinkList list)
3  {
4      if (NULL == list || NULL == list->next) return;

```

---

```

5      /* INT_MAX 来自头文件 limits.h, 典型值为 2^31 (2 的 31 次方) */
6      int minValue = INT_MAX; /* 存储最小值 */
7      LinkList minPrev;      /* 指向最小值节点的前驱节点 */
8      /* 找到最小值的节点的前驱节点 */
9      for (; NULL != list->next; list = list->next) {
10         LinkList curr = list->next;
11         if (curr->data < minValue) {
12             minValue = curr->data;
13             minPrev = list;
14         }
15     }
16     /* 删除最小值的节点 */
17     LinkList minNode = minPrev->next;
18     minPrev->next = minNode->next;
19     free(minNode);
20 }

```

#### 5. 就地逆置带头结点单链表 list

```

1  #include "list.h"
2  void ReverseListInPlace(LinkList list)
3  {
4      if (NULL == list || NULL == list->next) return;
5      LinkList node, next;
6      /* 把 list 断开成两个链表: list 空链表, node 余下节点 */
7      node = list->next;
8      list->next = NULL;
9      for (; NULL != node; node = next) {
10         next = node->next;
11         /* 插入头节点之后, 普通节点之前 */
12         node->next = list->next;
13         list->next = node;
14     }
15 }

```

#### 6. 排序带头结点单链表 list 为升序

```

1  #include "list.h"
2  /* 采用插入排序法 */
3  void SortList(LinkList list)
4  {
5      if (NULL == list || NULL == list->next) return;
6      LinkList node = list->next;
7      list->next = NULL;
8      /* 把余下节点 node, 插入到新链表 list */
9      LinkList next;
10     for (; NULL != node; node = next) {
11         next = node->next;
12         /* 寻找 */
13         LinkList prev = list;
14         while (node->data > prev->data && NULL != prev->next)

```

---

```

15         prev = prev->next;
16         /* 插入 */
17         node->next = prev->next;
18         prev->next = node;
19     }
20 }

```

---

7. 删除带头结点单链表 `list` 中介于给定的两个值 `lo`, `hi` （大于等于 `lo`, 小于等于 `hi`）中的元素

---

```

1  #include "list.h"
2  /**
3   * 整体思路和从 list 中删除 x 一样，只是条件不同
4   */
5  void DeleteValuesBetweenLoHiFromList(LinkList list, int lo, int hi)
6  {
7      if (NULL == list || NULL == list->next || lo > hi) return;
8      while (NULL != list->next) {
9          LinkList curr = list->next;
10         if (lo <= curr->data && curr->data <= hi) {
11             list->next = curr->next;
12             free(curr);
13         } else {
14             list = list->next;
15         }
16     }
17 }

```

---

## 2 栈

### 2.1 数据结构定义

栈可以采用数组连续存储或者单链表来实现，其中使用动态分配的顺序存储来实现更为常见。下面按照动态分配的连续存储实现的栈演示。

#### 2.1.1 顺序存储的栈

**静态分配** 类似于静态分配的顺序表。

---

```

1  #define MaxSize 1024
2  typedef struct SeqStack {
3      int data[MaxSize];
4      int top;
5  } SeqStack;

```

---

## 动态分配

---

```

1 typedef struct SeqStack {
2     int *data;
3     int top, capacity;
4 } SeqStack;

```

---

## 基本操作

1. 以最大容量 `capacity` 初始化栈 `bool InitStack(SeqStack *stack, int capacity);`

---

```

1 bool InitStack(SeqStack *stack, int capacity)
2 {
3     if (NULL == stack || capacity <= 0) return false;
4     stack->top = -1;
5     stack->capacity = capacity;
6     stack->data = malloc(capacity * sizeof(int));
7     if (NULL == stack->data) return false;
8     return true;
9 }

```

---

2. 销毁栈 `void DestroyStack(SeqStack *stack);` 和销毁动态分配的顺序表类似，就不再赘述

3. 获取栈内元素个数 `int StackSize(SeqStack *stack);`

---

```

1 int StackSize(SeqStack *stack)
2 {
3     if (NULL == stack) return 0;
4     return (stack->top + 1);
5 }

```

---

4. 获取栈最大容量 `int StackCapacity(SeqStack *stack);`

---

```

1 int StackCapacity(SeqStack *stack)
2 {
3     if (NULL == stack) return 0;
4     return stack->capacity;
5     /* 对于静态分配的栈 */
6     //return MaxSize;
7 }

```

---

5. 判断栈是否为空 `bool StackEmpty(SeqStack *stack);`

---

```

1 bool StackEmpty(SeqStack *stack)
2 {

```

---



```
3     if (NULL == stack) return true;
4     return (0 == StackSize(stack));
5     /* 或者 */
6     //return (-1 == stack->top);
7 }
```

---

6. 判断栈是否为满 `bool StackFull(SeqStack *stack);`

```
1 bool StackFull(SeqStack *stack)
2 {
3     if (NULL == stack) return false;
4     return (StackSize(stack) == StackCapacity(stack));
5     /* 或者 */
6     //return (stack->top + 1 == stack->capacity);
7 }
```

---

7. 进栈 `bool Push(SeqStack *stack, int ele);`

```
1 bool Push(SeqStack *stack, int ele)
2 {
3     if (NULL == stack || StackFull(stack)) return false;
4     stack->top += 1;
5     stack->data[stack->top] = ele;
6     return true;
7 }
```

---

8. 出栈 `bool Pop(SeqStack *stack, int *store);`

```
1 bool Pop(SeqStack *stack, int *store)
2 {
3     if (NULL == stack || NULL == store) return false;
4     if (StackEmpty(stack)) return false;
5     *store = stack->data[stack->top];
6     stack->top -= 1;
7     return true;
8 }
```

---

9. 获取栈顶元素 `bool GetTop(SeqStack *stack, int *store);`

```
1 bool GetTop(SeqStack *stack, int *store)
2 {
3     if (NULL == stack || NULL == store) return false;
4     if (StackEmpty(stack)) return false;
5     *store = stack->data[stack->top];
6     return true;
7 }
```

---

### 2.1.2 链式存储的栈

```
1 typedef struct LinkStack {  
2     int size; /* 存储了多少个元素 */  
3     LinkList top;  
4 } LinkStack;
```

#### 基本操作

1. 初始化栈 `bool InitStack(LinkStack *stack);`

```
1 bool InitStack(LinkStack *stack)  
2 {  
3     if (NULL == stack) return false;  
4     stack->size = 0;  
5     stack->top = NULL; /* 不带头节点的单链表 */  
6     return true;  
7 }
```

2. 获取栈内元素个数、判断栈是否为空等函数不再赘述

3. 进栈 `bool Push(LinkStack *stack, int ele);`

```
1 bool Push(LinkStack *stack, int ele)  
2 {  
3     if (NULL == stack) return false;  
4     /* 链表实现的栈的容量为无穷大, 所以不存在栈满的情况 */  
5     LinkNode *node = malloc(sizeof(LinkNode));  
6     if (NULL == node) return false;  
7     node->data = ele;  
8  
9     node->next = stack->top;  
10    stack->top = node;  
11  
12    stack->size += 1;  
13    return true;  
14 }
```

4. 出栈 `bool Pop(LinkStack *stack, int *store);`

```
1 bool Pop(LinkStack *stack, int *store)  
2 {  
3     if (NULL == stack || NULL == store) return false;  
4     if (StackEmpty(stack)) return false;  
5     LinkList node = stack->top;  
6     stack->top = node->next;  
7  
8     *store = node->data;
```

```

9     free(node);
10    return true;
11 }

```

5. 获取栈顶元素 `int GetElem(LinkStack *stack)`; 无须再给出吧

## 2.2 例题

栈的应用更多是在数和图里面作为辅助数据结构使用，所以单独考察变体不多。这里以判断括号是否匹配为例子，演示栈数据结构的使用。注意，这里栈内存储元素类型为 `char`，而不是前文示例所讲的元素类型 `int`，也就是说下面函数中的栈需要做适当的修改。

```

1  #include "stack.h"
2  #define CapacityMax 4096
3  bool IsBracketsValid(char *brackets)
4  {
5      bool valid = true;
6      SeqStack stack;
7      InitStack(&stack, CapacityMax);
8      int ch;
9      for (int i = 0; '\0' != brackets[i]; i++) {
10         switch (brackets[i]) {
11             /* 左“括号”入栈 */
12             case '(': Push(&stack, '('); break;
13             case '[': Push(&stack, '['); break;
14             case '{': Push(&stack, '{'); break;
15             /* 右“括号”出栈 */
16             case ')':
17                 if (StackEmpty(&stack)) {
18                     DestroyStack(&stack);
19                     return false;
20                 }
21                 Pop(&stack, &ch);
22                 if ('(' != ch) {
23                     DestroyStack(&stack);
24                     return false;
25                 }
26                 break;
27             case ']':
28                 if (StackEmpty(&stack)) {
29                     DestroyStack(&stack);
30                     return false;
31                 }
32                 Pop(&stack, &ch);
33                 if ('[' != ch) {
34                     DestroyStack(&stack);
35                     return false;
36                 }
37                 break;
38             case '}':

```

---

```

39         if (StackEmpty(&stack)) {
40             DestroyStack(&stack);
41             return false;
42         }
43         Pop(&stack, &ch);
44         if ('{' != ch) {
45             DestroyStack(&stack);
46             return false;
47         }
48         break;
49     default:
50         break;
51 }
52 }
53 /* 栈不为空表明全是左括号 ((({[[[[ 的情况 */
54 if (!StackEmpty(&stack)) {
55     DestroyStack(&stack);
56     return false;
57 }
58
59 DestroyStack(&stack);
60 return true;
61 }

```

---

## 3 队列

### 3.1 数据结构定义

#### 3.1.1 顺序存储的队列

同顺序表和顺序存储的栈，顺序存储的队列也有两种结构。下文只介绍动态分配的顺序存储队列的实现。

#### 动态分配

---

```

1 typedef struct SeqQueue {
2     int *data;
3     int capacity;
4     int front, rear;
5     /* 通过 size 记录存储元素的个数，进而作为另一种判断队列是否为空的方法 */
6     // int size;
7 } SeqQueue;

```

---

#### 基本操作

1. 初始化队列 `bool InitQueue(SeqQueue *queue, int capacity);`

```
1 bool InitQueue(SeqQueue *queue, int capacity)
2 {
3     if (NULL == queue || capacity <= 0) return false;
4     queue->data = malloc(capacity * sizeof(int));
5     if (NULL == queue->data) return false;
6     queue->capacity = capacity;
7     queue->front = queue->rear = 0;
8     //queue->size = 0;
9     return true;
10 }
```

2. 判断队列是否为空 `bool QueueEmpty(SeqQueue *queue);`

```
1 bool QueueEmpty(SeqQueue *queue)
2 {
3     if (NULL == queue) return true;
4     return (queue->front == queue->rear);
5     /* 如果采用 size 记录存储元素的个数 */
6     //return (0 == queue->size);
7 }
```

3. 判断队列是否为满 `bool QueueFull(SeqQueue *queue);`

```
1 bool QueueFull(SeqQueue *queue)
2 {
3     if (NULL == queue) return false;
4     return (queue->front == (queue->rear+1) % queue->capacity);
5     //return (queue->capacity == queue->size);
6 }
```

4. 获取队列当前元素个数 `int QueueSize(SeqQueue *queue);`

```
1 int QueueSize(SeqQueue *queue)
2 {
3     if (NULL == queue) return 0;
4     return (queue->rear - queue->front + queue->capacity) % queue->capacity;
5     //return queue->size;
6 }
```

5. 入队 `bool EnQueue(SeqQueue *queue, int ele);`

```
1 bool EnQueue(SeqQueue *queue, int ele)
2 {
3     if (NULL == queue || QueueFull(queue)) return false;
4     queue->data[queue->rear] = ele;
5     queue->rear = (queue->rear+1) % queue->capacity;
6     //queue->size += 1;
```

```

7     return true;
8 }

```

6. 出队 `bool DeQueue(SeqQueue *queue, int *store);`

```

1 bool DeQueue(SeqQueue *queue, int *store)
2 {
3     if (NULL == queue || NULL == store) return false;
4     if (QueueEmpty(queue)) return false;
5     *store = queue->data[queue->front];
6     queue->front = (queue->front+1) % queue->capacity;
7     //queue->size -= 1;
8     return true;
9 }

```

7. 获取队首元素 `bool GetHead(SeqQueue *queue, int *store);`

```

1 bool GetHead(SeqQueue *queue, int *store)
2 {
3     if (NULL == queue || NULL == store) return false;
4     if (QueueEmpty(queue)) return false;
5     *store = queue->data[queue->front];
6     return true;
7 }

```

### 3.1.2 链式存储的队列

单链表实现的队列，如果要求得当前队列元素个数，必须要遍历队列，效率较低，也可以通过一个变量 `size` 记录存储元素的个数。

```

1 typedef struct LinkQueue {
2     LinkList rear, front;
3     int size;
4 } LinkQueue;

```

#### 基本操作

1. 初始化队列 `bool InitQueue(LinkQueue *queue);`

```

1 bool InitQueue(LinkQueue *queue)
2 {
3     if (NULL == queue) return false;
4     LinkNode *dummy = malloc(sizeof(LinkNode)); /* 头节点 */
5     if (NULL == dummy) return false;
6     dummy->next = NULL;
7 }

```

```

8      /**
9      * front->[dummy]->NULL
10     * rear ----^
11     */
12     queue->front = queue->rear = dummy;
13     queue->size = 0;
14     return true;
15 }

```

---

2. 销毁队列 `void DestroyQueue(LinkQueue *queue);`

```

1 void DestroyQueue(LinkQueue *queue)
2 {
3     if (NULL == queue) return;
4     DestroyList(queue->front);
5 }

```

---

3. 判断队列是否为空 `bool QueueEmpty(LinkQueue *queue);`

```

1 bool QueueEmpty(LinkQueue *queue)
2 {
3     if (NULL == queue) return true;
4     return (queue->front == queue->rear);
5 }

```

---

4. 入队 `bool EnQueue(LinkQueue *queue, int ele);`

```

1 bool EnQueue(LinkQueue *queue, int ele)
2 {
3     if (NULL == queue) return false;
4     LinkNode *node = malloc(sizeof(LinkNode));
5     if (NULL == node) return false;
6     node->data = ele;
7     node->next = NULL;
8
9     /**
10    * front->[dummy]->[node1]->...->[ele]->NULL
11    * rear -----^
12    */
13     queue->rear->next = node;
14     queue->rear = node;
15
16     queue->size += 1;
17     return true;
18 }

```

---

5. 出队 `bool DeQueue(LinkQueue *queue, int *store);`

---

```

1  bool DeQueue(LinkQueue *queue, int *store)
2  {
3      if (NULL == queue || NULL == store) return false;
4      if (QueueEmpty(queue)) return false;
5
6      /**
7       * front->[dummy]->[node2]->...->[noden]->NULL
8       * rear -----^
9       * node ->[node1]
10     */
11     LinkList node = queue->front->next;
12     queue->front->next = node->next;
13     /* 如果是最后一个元素 */
14     if (node == queue->rear)
15         queue->rear = queue->front;
16     queue->size -= 1;
17
18     *store = node->data;
19     free(node);
20     return true;
21 }

```

---

## 3.2 例题

后续章节树和图的遍历会大量使用队列和栈，所以这里不再单独练习。

# 4 树

## 4.1 数据结构定义

### 4.1.1 双亲存储结构

**数组形式** 数组形式存储的  $k$  ( $k \geq 2$ ) 叉树，通常孩子节点下标  $n$  和双亲节点下标  $m$  满足如下关系（下标从 1 开始计数）：

- 通过孩子节点下标计算双亲节点下标： $m = \left\lceil \frac{n-1}{k} \right\rceil$
- 计算双亲节点的第  $i$  ( $1 \leq i \leq k$ ) 个孩子的下标： $n = (m-1)k + i + 1$

下面证明如何通过孩子节点计算双亲节点公式正确：

证明. 对于任意孩子节点  $n$ ，假设其双亲节点为  $m$ 。那么双亲  $m$  的第一个孩子节点下标为



$(m-1)k+2$ ，最后一个孩子节点下标为  $mk+1$ ，那么有不等式

$$\begin{aligned} (m-1)k+2 &\leq n \leq mk+1 \\ \Rightarrow m-1+\frac{2}{k} &\leq \frac{n}{k} \leq m+\frac{1}{k} \\ \Rightarrow m+\frac{1}{k}-1 &\leq \frac{n-1}{k} \leq m \\ \Rightarrow m &\leq \left\lceil \frac{n-1}{k} \right\rceil \leq m \\ \Rightarrow m &= \left\lceil \frac{n-1}{k} \right\rceil \end{aligned}$$

□

特别地，对于二叉树上述公式也是成立，但是此外有更为便捷的方法。考虑到对计算机而言，下取整比上取整更为方便，对上述证明中采取如下计算方法：

$$\begin{aligned} (m-1)k+2 &\leq n \leq mk+1 \\ \Rightarrow m &\leq \frac{n}{k}+1-\frac{2}{k} \leq m+1-\frac{1}{k} \\ \Rightarrow m &\leq \left\lfloor \frac{n}{k}+1-\frac{2}{k} \right\rfloor \leq \left\lfloor m+1-\frac{1}{k} \right\rfloor = m \\ \Rightarrow m &= \left\lfloor \frac{n}{k}+1-\frac{2}{k} \right\rfloor \text{ (考虑到 } k=2 \text{)} \\ \Rightarrow m &= \left\lfloor \frac{n}{2} \right\rfloor \end{aligned}$$

即由双亲节点得孩子节点  $n=2m$  或  $2m+1$ ，由孩子节点得双亲节点  $m=\left\lfloor \frac{n}{2} \right\rfloor$

---

```

1  #define TreeNodeMax 2048
2  /* Parent Tree Node */
3  typedef struct PTNode {
4      int data;
5      int parent;    /* -1 表明没有双亲节点 */
6  } PTNode;
7  /* Parent Tree */
8  typedef struct PTree {
9      PTNode *nodes;
10     int capacity;   /* nodes 的总个数 */
11 } PTree;

```

---

## 应用

1. 并查集 (Union Find Set): 并查集使用来判断两个元素是否在同一个集合的数据结构。顺序存储的树结构可以用于实现并查集, 不同之处在于并查集不需要通过双亲节点找到孩子节点, 因此可以更为灵活。注意: 下列代码中空集定义为双亲元素为 -1, 单元素集合定义为双亲元素为自身。

- 结构的定义

---

```

1 typedef struct UFSet {
2     int *nodes;
3     int capacity;
4 } UFSet;

```

---

- 初始化并查集 `bool InitUFSet(UFSet *set, int capacity);`

---

```

1 bool InitUFSet(UFSet *set, int capacity)
2 {
3     if (NULL == set || capacity <= 0) return false;
4     set->nodes = malloc(capacity * sizeof(int));
5     if (NULL == set->nodes) return false;
6     set->capacity = capacity;
7     /* 所有集合初始化为空集 */
8     for (int i = 0; i < capacity; i++)
9         set->nodes[i] = -1;
10    return true;
11 }

```

---

- 销毁并查集 `void DestroyUFSet(UFSet *set);`

---

```

1 void DestroyUFSet(UFSet *set)
2 {
3     if (NULL == set) return;
4     free(set->nodes);
5 }

```

---

- 添加单元素集合 {id} `bool MakeSet(UFSet *set, int id);`

---

```

1 bool MakeSet(UFSet *set, int id)
2 {
3     if (NULL == set) return false;
4     if (id < 0 || id >= set->capacity) return false;
5     /* 元素 id 已经添加 */
6     if (set->nodes[id] != -1) return false;
7     set->nodes[id] = id;
8     return true;
9 }

```

---

- 查找元素 id 所在集合的代表元素 `int Find(UFSet *set, int id);`

---

```

1 /**
2  * 返回 -1 表示 id 不在任何集合内

```

```

3  * 返回 >= 0 表示 id 所在集合的代表元素
4  */
5  int Find(UFSet *set, int id)
6  {
7      if (NULL == set) return -1;
8      if (id < 0 || id >= set->capacity) return -1;
9      while (true) {
10         int parent = set->nodes[id];
11         if (parent == id || parent == -1)
12             return parent;
13         id = parent;
14     }
15 }

```

- 判断两个元素是否同一集合 `bool Same(UFSet *set, int x, int y);`

```

1  bool Same(UFSet *set, int x, int y)
2  {
3      int xparent = Find(set, x);
4      int yparent = Find(set, y);
5      if (-1 == xparent || -1 == yparent)
6          return false;
7      return (xparent == yparent);
8  }

```

- 合并 `x` 和 `y` 所在的集合 `bool Union(UFSet *set, int x, int y);` 下面单纯的合并算法会导致查找函数 `Find` 效率降低, 实际中有两种方法优化: 按秩合并和路径压缩的方法。优化方法具体实现请参见<sup>2</sup>。

```

1  bool Union(UFSet *set, int x, int y)
2  {
3      int xparent = Find(set, x);
4      int yparent = Find(set, y);
5      if (-1 == xparent || -1 == yparent)
6          return false;
7      /* y 所在集合的代表元素 yparent 指向 x 集合代表元素 xparent */
8      set->nodes[yparent] = xparent;
9      return true;
10 }

```

## 2. 存储满二叉树

## 3. 大堆或小堆

**指针形式** 指针形式通常不能直接找到双亲节点的孩子节点, 通过孩子节点寻找双亲节点很方便。

<sup>2</sup>[https://en.wikipedia.org/wiki/Disjoint-set\\_data\\_structure](https://en.wikipedia.org/wiki/Disjoint-set_data_structure)

---

```
1 typedef struct PTreeNode {
2     int data;
3     struct PTreeNode *parent;
4 } PTreeNode;
5 typedef PTreeNode *PTree;
```

---

#### 4.1.2 孩子兄弟存储结构

---

```
1 /* Child Sibling Tree */
2 typedef struct CSTreeNode {
3     int data;
4     struct CSTreeNode *firstchild, *nextsibling;
5 } CSTreeNode;
6 typedef CSTreeNode *CSTree;
```

---

#### 4.1.3 二叉树、二叉排序数和平衡二叉树

---

```
1 /* Binary Tree */
2 typedef struct BiTreeNode {
3     int data;
4     struct BiTreeNode *lchild, *rchild;
5 } BiTreeNode;
6 typedef BiTreeNode *BiTree;
```

---

### 基本操作

#### 二叉树遍历

##### 1. 先序遍历 `void PreOrder(BiTree tree);`

- 递归实现

---

```
1 void PreOrder(BiTree tree)
2 {
3     if (NULL == tree) return;
4     /* 访问节点 tree */
5     //visit(tree);
6     PreOrder(tree->lchild);
7     PreOrder(tree->rchild);
8 }
```

---

- 非递归实现

- 版本一

---

```

1 void PreOrder(BiTree tree)
2 {
3     #define TreeHeightMax 2048
4     SeqStack stack;
5     InitStack(&stack, TreeHeightMax);
6
7     Push(&stack, tree);
8     while (!StackEmpty(&stack)) {
9         BiTree node;
10        Pop(&stack, &node);
11        if (NULL == node)
12            continue;
13        /* 访问节点 node */
14        //visit(node);
15        /* 先右节点入栈，然后是左节点入栈 */
16        Push(&stack, node->rchild);
17        Push(&stack, node->lchild);
18    }
19
20    DestroyStack(&stack);
21 }

```

---

○ 版本二 \*

---

```

1 void PreOrder(BiTree tree)
2 {
3     #define TreeHeightMax 2048
4     SeqStack stack;
5     InitStack(&stack, TreeHeightMax);
6
7     BiTree curr = tree;
8     while (NULL != curr || !StackEmpty(&stack)) {
9         if (NULL != curr) {
10            /* 访问当前节点 curr */
11            //visit(curr);
12            Push(&stack, curr);
13            curr = curr->lchild;
14        } else {
15            Pop(&stack, &curr);
16            curr = curr->rchild;
17        }
18    }
19
20    DestroyStack(&stack);
21 }

```

---

2. 中序遍历 `void InOrder(BiTree tree);`

- 递归实现

---

```

1 void InOrder(BiTree tree)
2 {

```

---

```

3     if (NULL == tree) return;
4     InOrder(tree->lchild);
5     /* 访问节点 tree */
6     //visit(tree);
7     InOrder(tree->rchild);
8 }

```

---

- 非递归实现

---

```

1 void InOrder(BiTree tree)
2 {
3     #define TreeHeightMax 2048
4     SeqStack stack;
5     InitStack(&stack, TreeHeightMax);
6
7     BiTree curr = tree;
8     while (NULL == curr || !StackEmpty(&stack)) {
9         if (NULL != curr) {
10             Push(&stack, curr);
11             curr = curr->lchild;
12         } else {
13             Pop(&stack, &curr);
14             /* 访问节点 node */
15             //visit(node);
16             curr = curr->rchild;
17         }
18     }
19     DestroyStack(&stack);
20 }
21

```

---

### 3. 后序遍历 `void PostOrder(BiTree tree);`

- 递归实现

---

```

1 void PostOrder(BiTree tree)
2 {
3     if (NULL == tree) return;
4     PostOrder(tree->lchild);
5     PostOrder(tree->rchild);
6     /* 访问节点 tree */
7     //visit(tree);
8 }

```

---

- 非递归实现<sup>3</sup>

---

```

1 void PostOrder(BiTree tree)
2 {
3     #define TreeHeightMax 2048
4     SeqStack stack;

```

---

<sup>3</sup>[https://en.wikipedia.org/wiki/Tree\\_traversal#Post-order\\_\(LRN\)](https://en.wikipedia.org/wiki/Tree_traversal#Post-order_(LRN))

```

5     InitStack(&stack, TreeHeightMax);
6     BiTree lastVisited = NULL;
7     BiTree curr = tree;
8
9     while (NULL != curr || !StackEmpty(&stack)) {
10         if (NULL != curr) {
11             Push(&stack, curr);
12             curr = curr->lchild;
13         } else {
14             GetTop(&stack, &curr); /* 只获取栈顶, 不弹栈 */
15             if (NULL != curr->rchild && lastVisited != curr->rchild) {
16                 curr = curr->rchild;
17             } else {
18                 /* 访问当前节点 */
19                 //visit(curr);
20                 Pop(&stack, &lastVisited);
21             }
22         }
23     }
24
25     DestroyStack(&stack);
26 }

```

#### 4. 层次遍历 `void LevelOrder(BiTree tree);`

```

1 void LevelOrder(BiTree tree)
2 {
3     #define TreeWidthMax 4096
4     if (NULL == tree) return;
5     SeqQueue queue;
6     InitQueue(&queue, TreeNodesMax);
7
8     EnQueue(&queue, tree);
9     while (!QueueEmpty(&queue)) {
10         BiTree node;
11         DeQueue(&queue, &node);
12         /* 访问节点 */
13         //visit(node);
14         if (NULL != node->lchild)
15             EnQueue(&queue, node->lchild);
16         if (NULL != node->rchild)
17             EnQueue(&queue, node->rchild);
18     }
19
20     DestroyQueue(&queue);
21 }

```

**二叉查找树** 二叉查找树 (Binary Search Tree, BST), 也叫排序二叉树或二叉排序树

##### 1. 查找元素 `bool BiTreeSearch(BiTree tree, int data);`

- 递归实现

---

```

1 bool BiTreeSearch(BiTree tree, int data)
2 {
3     if (NULL == tree) return false;
4     if (data == tree->data)
5         return true;
6     if (data < tree->data)
7         return BiTreeSearch(tree->lchild, data);
8     else
9         return BiTreeSearch(tree->rchild, data);
10 }

```

---

- 非递归实现

---

```

1 bool BiTreeSearch(BiTree tree, int data)
2 {
3     while (NULL != tree) {
4         if (data == tree->data)
5             return true;
6         if (data < tree->lchild)
7             tree = tree->lchild;
8         else
9             tree = tree->rchild;
10    }
11
12    return false;
13 }

```

---

## 2. 插入元素 `bool BiTreeInsert(BiTree *ptree, int data);`

---

```

1 #include "tree.h"
2 bool BiTreeInsert(BiTree *ptree, int data)
3 {
4     /* 和向不带头节点单链表中插入数据极其类似 */
5     while (NULL != *ptree) {
6         if (data == (*ptree)->data)
7             return true;
8         if (data < (*ptree)->data) {
9             ptree = &(*ptree)->lchild;
10        } else {
11            ptree = &(*ptree)->rchild;
12        }
13    }
14
15    BiTreeNode *node = malloc(sizeof(BiTreeNode));
16    if (NULL == node) return false;
17    node->data = data;
18    node->lchild = node->rchild = NULL;
19    *ptree = node;
20
21    return true;
22 }

```

---



3. 二叉排序树构造 `BiTree CreateBSTree(int *array, int n);`


---

```

1 BiTree CreateBSTree(int *array, int n)
2 {
3     if (NULL == array || n <= 0) return NULL;
4
5     BiTree tree = NULL;
6     for (int i = 0; i < n; i++)
7         BiTreeInsert(&tree, array[i]);
8
9     return tree;
10 }

```

---

4. 删除元素 `*bool BiTreeDelete(BiTree *ptree, int data);`


---

```

1 #include "tree.h"
2 /**
3  * 从树中偷取其最大值的节点，并存储到 store 里
4  */
5 void StealMaxNode(BiTree *ptree, BiTree *store)
6 {
7     while (NULL != (*ptree)->rchild)
8         ptree = &(*ptree)->rchild;
9     *store = *ptree;
10    *ptree = (*store)->lchild;
11 }
12 bool BiTreeDelete(BiTree *ptree, int data)
13 {
14     if (NULL == ptree) return false;
15     while (NULL != *ptree && data != (*ptree)->data) {
16         if (data < (*ptree)->data)
17             ptree = &(*ptree)->lchild;
18         else
19             ptree = &(*ptree)->rchild;
20     }
21     /* 没有值为 data 的节点 */
22     if (NULL == *ptree) return false;
23
24     BiTree lsubtree = (*ptree)->lchild;
25     BiTree rsubtree = (*ptree)->rchild;
26     free(*ptree);
27
28     /* 左右都有子树的节点的删除 */
29     if (NULL != lsubtree && NULL != rsubtree) {
30         BiTree maxNode;
31         StealMaxNode(&lsubtree, &maxNode);
32         *ptree = maxNode;
33         (*ptree)->lchild = lsubtree;
34         (*ptree)->rchild = rsubtree;
35         return true;

```

---

```

36     }
37
38     /* 只有左子树的节点的删除 */
39     if (NULL != lsubtree) {
40         *ptree = lsubtree;
41         return true;
42     }
43
44     /* 只有右子树的节点的删除 */
45     if (NULL != rsubtree) {
46         *ptree = rsubtree;
47         return true;
48     }
49
50     /* 没有左右子树的节点 */
51     *ptree = NULL;
52     return true;
53 }

```

**平衡二叉树** 平衡二叉树 AVL-Tree，查找算法和二叉查找树一样，但是插入和删除算法不同而且代码实现较为复杂（比二叉查找树删除元素还要复杂），这里暂时不给出。

## 应用

### 实现集合 \*

#### 4.1.4 线索二叉树

```

1  typedef struct ThreadTreeNode {
2      int data;
3      struct ThreadTreeNode *lchild, *rchild;
4      #define TagChild 0
5      #define TagPrevNode 1
6      #define TagNextNode 1
7      int ltag, rtag;
8  } ThreadTreeNode;
9  /**
10 * ltag 为 TagChild (0) 时, lchild 指向左孩子节点
11 * ltag 为 TagPrevNode (1) 时, lchild 指向前驱节点
12 * rtag 为 TagChild (0) 时, rchild 指向右孩子节点
13 * rtag 为 TagNextNode (1) 时, rchild 指向后继节点
14 */
15 typedef ThreadTreeNode *ThreadTree;

```

线索化的思想：任意一个节点有前驱节点且左孩子指针 `lchild` 没有指向左孩子，那么就指向此节点的前驱节点；任意一个节点有后继节点且右孩子指针 `rchild` 没有指向右孩子，那么就

指向后继节点。

## 基本操作

1. 先序遍历线索化 \* `void CreatePreThreadTree(ThreadTree tree);`

---

```

1  #include "tree.h"
2  void _CreatePreThreadTree(ThreadTree tree, ThreadTree *pprev)
3  {
4      if (NULL == tree) return;
5
6      /* 前驱指向当前节点 */
7      if (NULL != *pprev && NULL == (*pprev)->rchild) {
8          (*pprev)->rchild = tree;
9          (*pprev)->rtag = TagNextNode;
10     }
11     /* 当前节点指向前驱 */
12     if (NULL == tree->lchild) {
13         tree->lchild = *pprev;
14         tree->ltag = TagPrevNode;
15     }
16     pprev = &tree;
17
18     if (TagChild == tree->ltag)
19         _CreatePreThreadTree(tree->lchild, pprev);
20     if (TagChild == tree->rtag)
21         _CreatePreThreadTree(tree->rchild, pprev);
22 }
23 void CreatePreThreadTree(ThreadTree tree)
24 {
25     ThreadTree prev = NULL;
26     _CreatePreThreadTree(tree, &prev);
27 }
```

---

2. 中序遍历线索化 \* `void CreateInThreadTree(ThreadTree tree);`

---

```

1  #include "tree.h"
2  void _CreateInThreadTree(ThreadTree tree, ThreadTree *pprev)
3  {
4      if (NULL == tree) return;
5
6      if (TagChild == tree->ltag)
7          _CreateInThreadTree(tree->lchild, pprev);
8
9      if (NULL != *pprev && NULL == (*pprev)->rchild) {
10         (*pprev)->rchild = tree;
11         (*pprev)->rtag = TagNextNode;
12     }
13     if (NULL == tree->lchild) {
14         tree->lchild = *pprev;
15         tree->ltag = TagPrevNode;

```

---

```

16     }
17     pprev = &tree;
18
19     if (TagChild == tree->rtag)
20         _CreateInThreadTree(tree->rchild, pprev);
21 }
22 void CreateInThreadTree(ThreadTree tree)
23 {
24     ThreadTree prev = NULL;
25     _CreateInThreadTree(tree, &prev);
26 }

```

---

### 3. 后序遍历线索化 \* void CreatePostThreadTree(ThreadTree tree);

```

1  #include "tree.h"
2  void _CreatePostThreadTree(ThreadTree tree, ThreadTree *pprev)
3  {
4      if (NULL == tree) return;
5
6      if (TagChild == tree->lchild)
7          _CreatePostThreadTree(tree->lchild, pprev);
8      if (TagChild == tree->rchild)
9          _CreatePostThreadTree(tree->rchild, pprev);
10
11     if (NULL != *pprev && NULL == (*pprev)->rchild) {
12         (*pprev)->rchild = tree;
13         (*pprev)->rtag = TagNextNode;
14     }
15     if (NULL == tree->lchild) {
16         tree->lchild = *pprev;
17         tree->ltag = TagPrevNode;
18     }
19     pprev = &tree;
20 }
21 void CreatePostThreadTree(ThreadTree tree)
22 {
23     ThreadTree prev = NULL;
24     _CreatePostThreadTree(tree, &prev);
25 }

```

---

### 4. 先序遍历先序线索树 void PreOrderOfPreThreadTree(ThreadTree tree);

```

1  #include "tree.h"
2  /* 在先序序列中，找给定节点的后继元素 */
3  ThreadTree NextNode(ThreadTree node)
4  {
5      if (TagChild == node->ltag)
6          return node->lchild;
7      return node->rchild;
8  }
9  /* 在先序序列中，找给定节点的前驱元素 */
10 ThreadTree PrevNode(ThreadTree node)

```

```

11 {
12     /**
13     * 注意：比较困难，可能需要遍历整棵树
14     * 因为，如果 node->ltag == TagPrevNode,
15     *     只需要返回 node->lchild
16     *     否则前驱节点就是 node 的父节点，
17     *     然而在二叉树孩子节点表示法中，难以找到父节点
18     */
19 }
20 /* 在先序线索树中，找先序序列的最后一个节点 */
21 ThreadTree LastNode(ThreadTree tree)
22 {
23     ThreadTree nextNode;
24     while (true) {
25         nextNode = NextNode(tree);
26         if (NULL == nextNode)
27             return tree;
28         tree = nextNode;
29     }
30 }
31 /* 在先序线索树中，找先序序列的第一个节点 */
32 ThreadTree FirstNode(ThreadTree tree)
33 {
34     return tree;
35 }
36 void PreOrderOfPreThreadTree(ThreadTree tree)
37 {
38     /**
39     * 把线索化后的节点看成可以通过 PrevNode() 找到前驱和 NextNode() 找到后继的
40     * 非循环双链表，如果某一个节点 PrevNode() 为 NULL，说明这个节点为第一个节
41     * 点；同理，如果 NextNode() 为 NULL，说明这个节点为最后一个节点
42     */
43     ThreadTree node;
44     for (node = FirstNode(tree); NULL != node; node = NextNode(node)) {
45         /* 访问节点 */
46         //visit(node);
47     }
48 }

```

#### 5. 中序遍历中序线索树 void InOrderOfInThraedTree(ThreadTree tree);

```

1 #include "tree.h"
2 /* 在中序线索树中，求给定节点的后继 */
3 ThreadTree NextNode(ThreadTree node)
4 {
5     if (TagNextNode == node->rtag)
6         return node->rchild;
7     /* 右子树的第一个节点就是当前节点的后继节点 */
8     return FirstNode(node->rchild);
9 }
10 /* 在中序线索树中，求给定节点的前驱 */
11 ThreadTree PrevNode(ThreadTree node)
12 {

```

```

13     if (TagPrevNode == node->ltag)
14         return node->lchild;
15     /* 左子树的最后一个节点解释当前节点的前驱节点 */
16     return LastNode(node->lchild);
17 }
18 /* 在中序线索树中，求中序序列第一个节点 */
19 ThreadTree FirstNode(ThreadTree tree)
20 {
21     if (NULL == tree) return NULL;
22     /* 左边“链”往左下的最后一个节点 */
23     while (TagPrevNode != tree->ltag)
24         tree = tree->lchild;
25     return tree;
26 }
27 /* 在中序线索树中，求中序序列最后一个节点 */
28 ThreadTree LastNode(ThreadTree tree)
29 {
30     if (NULL == tree) return NULL;
31     /* 右边“链”往右下的最后一个节点 */
32     while (TagNextNode != tree->rtag)
33         tree = tree->rchild;
34     return tree;
35 }
36 void InOrderOfInThraedTree(ThreadTree tree)
37 {
38     ThreadTree node;
39     for (node = FirstNode(tree); NULL != node; node = NextNode(node)) {
40         /* 访问节点 node */
41         //visit(node);
42     }
43     /* 如果是按照中序序列逆序 */
44     //for (node = LastNode(tree); NULL != node; node = PrevNode(node)) {
45     //    /* 访问节点 node */
46     //    //visit(node);
47     //}
48 }

```

对中序线索树中序遍历时，某一节点的前驱或者后继节点一定是其子树中的节点，所以一定方便找到前驱和后继。对比先序线索树的先序遍历，某一节点的前去节点可能是其父节点，所以难以找到前驱节点。

#### 6. 后序遍历后序线索树 `void PostOrderOfPostThread(ThreadTree tree);`

```

1  #include "tree.h"
2  ThreadTree NextNode(ThreadTree node)
3  {
4      /**
5       * 注意：和先序遍历寻找前驱类似，也是比较困难，原因一样
6       * 如果 node->rtag == TagNextNode, 直接返回 node->rchild 即可
7       * 否则就是 node 的父节点，父节点难以求得
8       */

```

```

9  }
10 ThreadTree PrevNode(ThreadTree node)
11 {
12     if (TagChild == node->rtag)
13         return node->rchild;
14     return node->lchild;
15 }
16 ThreadTree FirstNode(ThreadTree tree)
17 {
18     if (NULL == tree) return tree;
19     /* 第一个节点是“最左边”的第一个叶节点 */
20     while (true) {
21         if (TagChild == tree->ltag)
22             tree = tree->lchild;
23         else if (TagChild == tree->rtag)
24             tree = tree->rchild;
25         else
26             break;
27     }
28     return tree;
29 }
30 ThreadTree LastNode(ThreadTree tree)
31 {
32     return tree;
33 }
34 void PostOrderOfPostThread(ThreadTree tree)
35 {
36     /**
37      * 后序线索树想要后序遍历，因为 NextNode() 函数难以实现，
38      * 所以后序遍历也难以实现
39      * 但是，逆后序遍历是容易的，下面给出逆后序遍历
40      */
41     ThreadTree node;
42     for (node = LastNode(tree); NULL != node; node = PrevNode(node)) {
43         /* 访问节点 */
44         //visit(node);
45     }
46 }

```

对比先序线索树的先序遍历，可以发现后序线索树的后序遍历与其有一定的对称性。

## 4.2 例题

### 1. 求二叉树的高度

```

1  #include "tree.h"
2  /* 这里给出递归求树的高度，也可以通过非递归的任意遍历求树高 */
3  int BiTreeHeight(BiTree tree)
4  {
5      if (NULL == tree) return 0;
6      int lheight = BiTreeHeight(tree->lchild);

```

```

7     int rheight = BiTreeHeight(tree->rchild);
8     return (lheight>rheight)? (lheight+1): (rheight+1);
9 }

```

2. 求二叉树宽度和高度，二叉树宽度定义为同一层中节点数最大值

```

1  #include "tree.h"
2  #include "queue.h"
3  int max(int x, int y)
4  {
5      return (x>y)? x: y;
6  }
7  bool BiTreeWidthAndHeight(BiTree tree, int *width, int *height)
8  {
9      #define TreeWidthMax 10240
10     if (NULL == height || NULL == width) return false;
11     *height = 0;
12     *width = 0;
13     if (NULL == tree) return true;
14
15     SeqQueue queue;
16     BiTree lastEnQueue;
17
18     InitQueue(&queue, TreeWidthMax);
19     EnQueue(&queue, tree);
20     lastEnQueue = tree;          /* 每一层的最后一个入队列的元素 */
21
22     while (!QueueEmpty(&queue)) {
23         int width2 = 0;
24         BiTree thisLayerLastEnQueue = NULL;
25
26         /* 处理一层 */
27         while (true) {
28             BiTree node;
29             DeQueue(&queue, &node);
30             width2 ++;
31             if (NULL != node->lchild) {
32                 EnQueue(&queue, node->lchild);
33                 thisLayerLastEnQueue = node->lchild;
34             }
35             if (NULL != node->rchild) {
36                 EnQueue(&queue, node->rchild);
37                 thisLayerLastEnQueue = node->rchild;
38             }
39             if (lastEnQueue == node)
40                 break;
41         }
42
43         lastEnQueue = thisLayerLastEnQueue;
44         *height += 1;
45         *width = max(*width, width2);
46     }
47

```



```

48
49     DestroyQueue(&queue);
50     return true;
51 }

```

3. 求左子树中节点经过根节点到右子树中节点的最长路径的长度

```

1  #include "tree.h"
2  int MaxDistThroughRoot(BiTree tree)
3  {
4      /**
5       * 注意：树的高度等于根节点到最远叶节点距离加一
6       * 高度和路劲长度不是完全一回事
7       */
8      if (NULL == tree) return 0;
9      int lheight = BiTreeHeight(tree->lchild);
10     int rheight = BiTreeHeight(tree->rchild);
11     if (lheight >= 1) lheight -= 1;
12     if (rheight >= 1) rheight -= 1;
13     return lheight + rheight;
14 }

```

4. 输出根节点到每个叶子结点的路径

```

1  #include "tree.h"
2  #define BiTreeHeightMAX 4096
3  int trace[BiTreeHeightMAX];
4  int idx = 0;
5  /* DFS 深度遍历 */
6  void PrintAllTraces(BiTree tree)
7  {
8      if (NULL == tree) {
9          for (int i = 0; i < idx; i++)
10             printf("%d ", trace[i]);
11             printf("\n");
12             return;
13     }
14     trace[idx] = tree->data;
15     idx++;
16     PrintAllTraces(tree->lchild);
17     PrintAllTraces(tree->rchild);
18     idx--;
19 }

```

5. 输出二叉树给定节点的所有祖先

```

1  #include "tree.h"
2  /* 如果是二叉排序树 */
3  void PrintAncestors(BiTree tree, int x)
4  {

```

```

5     while (NULL == tree || x == tree->data) {
6         printf("%d ", tree->data);
7         if (x < tree->data)
8             tree = tree->lchild;
9         else
10            tree = tree->rchild;
11    }
12 }
13
14 /* 一般二叉树 */
15 bool _PrintAncestors(BiTree tree, int x)
16 {
17     if (NULL == tree) return false;
18     if (x == tree->data)
19         return true;
20     /* 如果左子树或者右子树找到 x, 那么当前节点是 x 的祖先, 那么输出它 */
21     if (_PrintAncestors(tree->lchild, x) || _PrintAncestors(tree->rchild, x)) {
22         printf("%d ", tree->data);
23         return true;
24     }
25     return false;
26 }
27 void PrintAncestors(BiTree tree, int x)
28 {
29     _PrintAncestors(tree, x);
30 }

```

## 6. 判断给定二叉树是否为完全二叉树

```

1  #include "tree.h"
2  #include "queue.h"
3  /**
4   * 完全二叉树: 除了最有一层右边部分外, 其余层都是满的
5   * 层次遍历时, 如果出现空节点, 那么其后续元素都应当是空节点
6   */
7  bool IsCompleteBiTree(BiTree tree)
8  {
9      #define BiTreeWidthMax 4096
10     if (NULL == tree) return true;
11     SeqQueue queue;
12     InitQueue(&queue, BiTreeWidthMax);
13     EnQueue(&queue, tree);
14
15     while (!QueueEmpty(&queue)) {
16         BiTree node;
17         DeQueue(&queue, &node);
18         if (NULL != node) {
19             EnQueue(&queue, node->lchild);
20             EnQueue(&queue, node->rchild);
21         } else {
22             while (!QueueEmpty(&queue)) {
23                 DeQueue(&queue, &node);
24                 if (NULL != node) {

```

```

25         DestroyQueue(&queue);
26         return false;
27     }
28 }
29 DestroyQueue(&queue);
30 return true;
31 }
32 }
33
34 DestroyQueue(&queue);
35 return true;
36 }

```

#### 7. 判断二叉树 subtree 是否为 tree 的一棵子树

```

1  #include "tree.h"
2  bool IsEqualBiTree(BiTree tree1, BiTree tree2)
3  {
4      if (NULL == tree1 && NULL == tree2)
5          return true;
6      if ((NULL != tree1 && NULL != tree2) && (tree1->data == tree2->data)) {
7          return IsEqualBiTree(tree1->lchild, tree2->lchild)
8              && IsEqualBiTree(tree1->rchild, tree2->rchild);
9      }
10     return false;
11 }
12
13 bool IsSubTree(BiTree tree, BiTree subtree)
14 {
15     if (IsEqualBiTree(tree, subtree))
16         return true;
17     if (NULL != tree) {
18         if (IsSubTree(tree->lchild, subtree))
19             return true;
20         if (IsSubTree(tree->rchild, subtree))
21             return true;
22     }
23     return false;
24 }

```

#### 8. 判断二叉树 part 是否为 tree 的子结构，子结构是指能在 tree 找到 part 这样的树结构，并不一定是子树。

```

1  #include "tree.h"
2  bool IsPartOf(BiTree tree, BiTree part)
3  {
4      if (NULL == part)
5          return true;
6      if (NULL == tree)    /* part 一定不等于 NULL */
7          return false;
8      if (tree->data != part->data)

```

```

9         return false;
10        return IsPartOf(tree->lchild, part->lchild)
11                && IsPartOf(tree->rchild, part->rchild);
12    }
13    bool IsPartOfBiTree(BiTree tree, BiTree part)
14    {
15        if (IsPartOf(tree, part))
16            return true;
17        if (NULL != tree) {
18            if (IsPartOf(tree->lchild, part))
19                return true;
20            if (IsPartOf(tree->rchild, part))
21                return true;
22        }
23        return false;
24    }

```

9. 按照从下往上、从右往左的方式遍历二叉树（逆层次遍历）

```

1  #include "tree.h"
2  #include "stack.h"
3  #include "queue.h"
4  void ReLevelOrder(BiTree tree)
5  {
6      /* 层次遍历加逆序输出 */
7      #define BiTreeNodeMax 10240
8      if (NULL == tree) return;
9      SeqStack stack;
10     SeqQueue queue;
11     InitQueue(&queue, BiTreeNodeMax);
12     InitStack(&stack, BiTreeNodeMax);
13
14     EnQueue(&queue, tree);
15     while (!QueueEmpty(&queue)) {
16         BiTree node;
17         DeQueue(&queue, &node);
18         Push(&stack, node->data);
19         if (NULL != node->lchild)
20             EnQueue(&queue, node->lchild);
21         if (NULL != node->rchild)
22             EnQueue(&queue, node->rchild);
23     }
24
25     while (!StackEmpty(&stack)) {
26         int data;
27         Pop(&stack, &data);
28         printf("%d ", data);
29     }
30     printf("\n");
31     DestroyStack(&stack);
32 }
33

```

## 10. 通过先序序列和中序序列重构二叉树 \*

---

```

1  #include "tree.h"
2  /**
3   * 重构过程也是先序的过程，所以即从 preArr 中不断取元素构建即可，
4   * 同时根据 inArr 数组来确定某个节点有没有左右子树，
5   * 进而决定 preArr 余下的的节点在这个节点左子树还是右子树
6   */
7  BiTree _ReBuildBiTree(int *preArr, int preIdx, int len,
8                        int *inArr, int inIdx, int inEnd)
9  {
10     if (preIdx >= len || inIdx >= inEnd)
11         return NULL;
12
13     int rootData = preArr[preIdx];
14     BiTree root = malloc(sizeof(BiTreeNode));
15     if (NULL == root) return NULL;
16     root->data = rootData;
17
18     /* 在中序数组 inArr 中找到 rootData，然后划分成两部分 */
19     int i;
20     for (i = inIdx; i < inEnd; i++) {
21         if (rootData == inArr[i])
22             break;
23     }
24     /* 数据有误，在中序数组中没能找到先序数组中的元素 */
25     if (i == inEnd) {
26         printf("Invalid input data in preArr or inArr.\n");
27         return NULL;
28     }
29
30     int lpreIdx = preIdx+1;
31     int rpreIdx = preIdx+1 + (i-inIdx);
32     root->lchild = _ReBuildBiTree(preArr, lpreIdx, len, inArr, inIdx, i);
33     root->rchild = _ReBuildBiTree(preArr, rpreIdx, len, inArr, i+1, inEnd);
34
35     return root;
36 }
37 BiTree ReBuildBiTree(int *preArr, int *inArr, int len)
38 {
39     if (len <= 0 || NULL == preArr || NULL == inArr) return NULL;
40     return _ReBuildBiTree(preArr, 0, len, inArr, 0, len);
41 }

```

---

## 11. 由满二叉树的先序序列求出其后序序列，也就是对顺序存储的满二叉树数组进行后续遍历

---

```

1  #include "tree.h"
2  /* 这里的顺序存储的满二叉树下标从 0 开始，所以和前面所讲有所差异 */
3  void postOrder(int *preArr, int currNodeIdx, int len, int *postArr, int *idx)
4  {
5     if (currNodeIdx >= len) return;

```

---

```

6     postOrder(preArr, 2*currNodeIdx+1, len, postArr, idx);
7     postOrder(preArr, 2*currNodeIdx+2, len, postArr, idx);
8     postArr[*idx] = preArr[currNodeIdx];
9     (*idx) += 1;
10 }
11 void FromPreOrderToPostOrder(int *preArr, int *postArr, int len)
12 {
13     if (len <= 0 || NULL == preArr || NULL == postArr) return;
14     int idx = 0;
15     postOrder(preArr, 0, len, postArr, &idx);
16 }

```

---

## 12. 统计二叉树的节点数、叶子数、双分叉节点数

```

1  #include "tree.h"
2  int CountNodes(BiTree tree)
3  {
4      if (NULL == tree) return 0;
5      return (1 + CountNodes(tree->lchild) + CountNodes(tree->rchild));
6  }
7
8  int CountLeafs(BiTree tree)
9  {
10     if (NULL == tree) return 0;
11     if (NULL == tree->lchild && NULL == tree->rchild)
12         return 1;
13     return CountLeafs(tree->lchild) + CountLeafs(tree->rchild);
14 }
15
16 int CountForks(BiTree tree)
17 {
18     int cnt = 0;
19     if (NULL == tree) return 0;
20     if (NULL != tree->lchild && NULL != tree->rchild)
21         cnt = 1;
22     return (cnt + CountForks(tree->lchild) + CountForks(tree->rchild));
23 }

```

---

## 13. 交换左右子树

```

1  #include "tree.h"
2  void SwapBiTree(BiTree tree)
3  {
4      if (NULL == tree) return;
5      BiTree t = tree->lchild;
6      tree->lchild = tree->rchild;
7      tree->rchild = t;
8      SwapBiTree(tree->lchild);
9      SwapBiTree(tree->rchild);
10 }

```

---

## 14. 删除二叉树所有值为 x 的子树（不是节点）

---

```

1  #include "tree.h"
2  void DestroyBiTree(BiTree tree)
3  {
4      if (NULL == tree) return;
5      DestroyBiTree(tree->lchild);
6      DestroyBiTree(tree->rchild);
7      free(tree);
8  }
9  void DeleteAllSubtreeXFromBiTree(BiTree *ptree, int x)
10 {
11     if (NULL == ptree || NULL == *ptree) return;
12     if (x == (*ptree)->data) {
13         DestroyBiTree(*ptree);
14         *ptree = NULL;
15         return;
16     }
17     DeleteAllSubtreeXFromBiTree(&(*ptree)->lchild, x);
18     DeleteAllSubtreeXFromBiTree(&(*ptree)->rchild, x);
19 }

```

---

## 15. 删除二叉树（不是二叉排序树）所有值为 x 的节点 \*，由于不是二叉排序树，所以删除节点后的树不唯一，只需要保证是二叉树即可

---

```

1  #include "tree.h"
2  /**
3   * 从二叉树中取下某一个节点，存储到 store 指向的空间里
4   */
5  bool StealNodeFromBiTree(BiTree *ptree, BiTree *store)
6  {
7      /* 空树，没节点可取 */
8      if (NULL == *ptree) return false;
9      if (NULL != (*ptree)->lchild)
10         return StealNodeFromBiTree(&(*ptree)->lchild, store);
11     if (NULL != (*ptree)->rchild)
12         return StealNodeFromBiTree(&(*ptree)->rchild, store);
13     /* 只有根节点 */
14     *store = *ptree;
15     *ptree = NULL;
16     return true;
17 }
18 void DeleteAllXFromBiTree(BiTree *ptree, int x)
19 {
20     if (NULL == ptree || NULL == *ptree) return;
21
22     DeleteAllXFromBiTree(&(*ptree)->lchild, x);
23     DeleteAllXFromBiTree(&(*ptree)->rchild, x);
24
25     if (x == (*ptree)->data) {
26         BiTree lsubtree = (*ptree)->lchild;
27         BiTree rsubtree = (*ptree)->rchild;

```

```

28     free(*ptree);
29
30     /* 从左子树或者右子树偷一个节点来替代被删除的根节点 */
31     BiTree node;
32     if (StealNodeFromBiTree(&lsubtree, &node)
33         || StealNodeFromBiTree(&rsubtree, &node)) {
34         *ptree = node;
35         (*ptree)->lchild = lsubtree;
36         (*ptree)->rchild = rsubtree;
37         return;
38     }
39     /* 左右子树都没有节点可偷, 说明这棵树只有一个节点, 直接删除即可 */
40     *ptree = NULL;
41 }
42 }

```

#### 16. 查找二叉树节点 p 和 q 的最近公共祖先

```

1  #include "tree.h"
2  /**
3   * 按照从节点 node 往上到根 tree 的顺序,
4   * 储所有祖先到 ancestorArr 数组内 (包含 node 节点),
5   * 并且存储祖先数量到 arrLen 指针指向的整数里
6   * 返回 true, 表明获取成功
7   * false, 表明 node 不是树 tree 的节点
8   * 思路和 PrintAncestors 一样
9   */
10 bool GetAllAncestors(BiTree tree, BiTree node, BiTree *ancestorArr, int *arrLen)
11 {
12     if (NULL == tree) return false;
13     if (tree == node
14         || GetAllAncestors(tree->lchild, node, ancestorArr, arrLen)
15         || GetAllAncestors(tree->rchild, node, ancestorArr, arrLen)) {
16         ancestorArr[*arrLen] = tree;
17         (*arrLen) += 1;
18     }
19     return false;
20 }
21 BiTree CommonAncestor(BiTree tree, BiTree p, BiTree q)
22 {
23     #define BiTreeHeightMax 4096
24     if (NULL == tree || NULL == p || NULL == q) return NULL;
25
26     int pLen = 0, qLen = 0;
27     BiTree pAncestors[BiTreeHeightMax], qAncestors[BiTreeHeightMax];
28
29     /* 如果没有获取成功, p 就不是 tree 的节点 */
30     if (!GetAllAncestors(tree, p, pAncestors, &pLen))
31         return NULL;
32     if (!GetAllAncestors(tree, q, qAncestors, &qLen))
33         return NULL;
34
35     do {

```



```

36     pLen--;
37     qLen--;
38 } while (pAncestors[pLen] == qAncestors[qLen]);
39 /**
40  * 循环结束, 此时 pAncestors[pLen] != qAncestors[qLen]
41  * 却有, pAncestors[pLen+1] == qAncestors[qLen+1]
42  */
43
44     return pAncestors[pLen+1];
45 }

```

17. 将二叉树的叶子节点从左向右顺序连接, 即把所有叶子节点组织成非循环双链表

```

1  #include "tree.h"
2
3  /* pprevLeaf: pointer on prevLeaf */
4  void _LinkLeafs(BiTree tree, BiTree *pprevLeaf)
5  {
6      if (NULL == tree) return;
7      if (NULL == tree->lchild && NULL == tree->rchild) {
8          if (NULL != *pprevLeaf)
9              (*pprevLeaf)->rchild = tree;
10         tree->lchild = *pprevLeaf;
11         pprevLeaf = &tree;
12         return;
13     }
14     _LinkLeafs(tree->lchild, pprevLeaf);
15     _LinkLeafs(tree->rchild, pprevLeaf);
16 }
17 void LinkLeafs(BiTree tree)
18 {
19     BiTree prevLeaf = NULL; /* 存储上一片叶子节点地址 */
20     _LinkLeafs(tree, &prevLeaf);
21 }

```

18. 判断两棵二叉树结构是否相似 (只判断形状不判断节点值)

```

1  #include "tree.h"
2  bool IsSameBiTree(BiTree tree1, BiTree tree2)
3  {
4      if (NULL == tree1 && NULL == tree2)
5          return true;
6      if (NULL != tree1 && NULL != tree2) {
7          return IsSameBiTree(tree1->lchild, tree2->rchild)
8              && IsSameBiTree(tree1->lchild, tree2->rchild);
9      }
10     return false;
11 }

```

19. 在中序线索树查找给定节点在后序遍历序列中的前驱节点

思想：若此节点有右子树，那么前驱就是右节点；若有左子树，那么就是左节点；如果左右子树都没有，那么节点左指针指向某一个祖先节点，如果双亲节点有左子树，那么前驱就是祖先节点的左节点，如果左节点没有则继续往上寻找，知道某一个祖先节点满足具有左子树，除非左子树为空指针，那么这个节点是后序遍历的第一个节点，没有前驱。

---

```

1  #include "tree.h"
2  ThreadTree PrevNodeOfPostOrderInThreadTree(ThreadTree tree, ThreadTree node)
3  {
4      if (NULL == tree || NULL == node) return NULL;
5      if (TagChild == node->rtag)
6          return node->rchild;
7      if (TagChild == node->ltag)
8          return node->lchild;
9      while (NULL != node->lchild) {
10         node = node->lchild;
11         if (TagChild == node->ltag)
12             return node->lchild;
13     }
14     return NULL;
15 }

```

---

#### 20. 计算二叉树的带权路径长度 (Weighted Path Length, WPL)

---

```

1  #include "tree.h"
2  int _WPLofBiTree(BiTree tree, int depth)
3  {
4      if (NULL == tree) return 0;
5      if (NULL == tree->lchild && NULL == tree->rchild)
6          return tree->data * (depth+1);
7      return _WPLofBiTree(tree->lchild, depth+1)
8             + _WPLofBiTree(tree->rchild, depth+1);
9  }
10 int WPLofBiTree(BiTree tree)
11 {
12     return _WPLofBiTree(tree, 0);
13 }

```

---

#### 21. 输出给定表达树的中缀表达式子（通过括号体现优先级）

---

```

1  #include "tree.h"
2  /**
3   * 表达式：叶子节点为数据，非叶子节点为运算符
4   * 实际上就是中序遍历
5   */
6  void PrintExpressionBiTree(BiTree tree)
7  {
8      if (NULL == tree) return;
9
10     /* 操作数 */

```

```

11     if (NULL == tree->lchild && NULL == tree->rchild) {
12         printf("%d", tree->data);
13         return;
14     }
15
16     printf("(");
17     PrintExpressionBiTree(tree->lchild);
18     printf("%c", tree->data); /* 操作符 */
19     PrintExpressionBiTree(tree->rchild);
20     printf(")");
21 }

```

---

## 22. 求指定值在二叉排序树中的高度

```

1  #include "tree.h"
2  int HeightOfValueInBSTTree(BiTree tree, int value)
3  {
4      if (NULL == tree) return 0;
5
6      int height = 1;
7      while (NULL != tree) {
8          if (value == tree->data)
9              return height;
10         if (value < tree->data)
11             tree = tree->lchild;
12         else
13             tree = tree->rchild;
14         height += 1;
15     }
16
17     return -1; /* 在树 tree 中没有找到值为 value 的节点 */
18 }

```

---

## 23. 判断给定二叉树是否是二叉排序树

```

1  #include "tree.h"
2  /* 中序遍历看是否有序 */
3  bool _IsBSTTree(BiTree tree, int *pprev)
4  {
5      if (NULL == tree) return true;
6      if (!_IsBSTTree(tree->lchild, pprev))
7          return false;
8      if (*pprev > tree->data)
9          return false;
10     *pprev = tree->data;
11     if (!_IsBSTTree(tree->rchild, pprev))
12         return false;
13     /* 上述条件都满足才是二叉排序树 */
14     return true;
15 }
16 bool IsBSTTree(BiTree tree)
17 {

```

```

18     /* INT_MIN 来自 limits.h, 最小整数 */
19     int prev = INT_MIN;
20     return _IsBSTTree(tree, &prev);
21 }

```

---

#### 24. 判断给定二叉树是否是平衡二叉树

```

1  #include "tree.h"
2  int abs(int x)
3  {
4      return (x<0)? (-x): x;
5  }
6  int max(int x, int y)
7  {
8      return (x>y)? x: y;
9  }
10 /**
11  * pprev: 指向存储当前节点的前驱节点的值
12  * pheight: 指向存储子树高度的变量
13  * 这个函数实际上就是通过中序遍历即判断了是否是二叉排序树,
14  * 同时求出左右子树高度并判断高度之差是否小于等于 1
15  * 也就是组合了判断是否是二叉排序树的函数和求树高的函数
16  */
17 bool _IsAVLTree(BiTree tree, int *pprev, int *pheight)
18 {
19     if (NULL == tree) {
20         *pheight = 0;
21         return true;
22     }
23
24     int lheight, rheight;
25     if (!_IsAVLTree(tree->lchild, pprev, &lheight))
26         return false;
27     if (*pprev > tree->data)
28         return false;
29     *pprev = tree->data;
30     if (!_IsAVLTree(tree->rchild, pprev, &rheight))
31         return false;
32
33     *pheight = 1 + max(lheight, rheight);
34
35     return (abs(lheight - rheight) <= 1);
36 }
37 bool IsAVLTree(BiTree tree)
38 {
39     int prev = INT_MIN;
40     int height;
41     return _IsAVLTree(tree, &prev, &height);
42 }

```

---

#### 25. 求二叉排序树的最大值和最小值

---

```

1  #include "tree.h"
2  int MaxValue(BiTree tree)
3  {
4      /* 返回 INT_MIN 表示没有最大值, INT_MAX 来自头文件 limits.h */
5      if (NULL == tree)
6          return INT_MIN;
7
8      while (NULL != tree->rchild)
9          tree = tree->rchild;
10     return tree->data;
11 }
12 bool MinValue(BiTree tree)
13 {
14     if (NULL == tree) return INT_MAX;
15
16     while (NULL != tree->lchild)
17         tree = tree->lchild;
18     return tree->data;
19 }

```

---

26. 从大到小输出二叉排序树中大于等于 lo 小于等于 hi 的节点的值

- 版本一：遍历整棵树，根据判断是否输出节点值

---

```

1  #include "tree.h"
2  /* 类似于中序遍历，只是先右子树后左子树 */
3  void PrintFromHiToLoInBSTTree(BiTree tree, int lo, int hi)
4  {
5      if (NULL == tree || lo > hi) return;
6      PrintFromHiToLoInBSTTree(tree->rchild, lo, hi); /* 右子树 */
7      if (lo <= tree->data && tree->data <= hi) /* 只输出满足条件的 */
8          printf("%d ", tree->data);
9      PrintFromHiToLoInBSTTree(tree->lchild, lo, hi); /* 左子树 */
10 }

```

---

- 版本二：仅仅遍历要输出的节点

---

```

1  #include "tree.h"
2  void PrintFromHiToLoInBSTTree(BiTree tree, int lo, int hi)
3  {
4      if (NULL == tree || lo > hi) return;
5
6      if (tree->data <= hi)
7          PrintFromHiToLoInBSTTree(tree->rchild, lo, hi);
8
9      printf("%d ", tree->data);
10
11     if (tree->data >= lo)
12         PrintFromHiToLoInBSTTree(tree->lchild, lo, hi);
13 }

```

---

27. 以  $O(\log_2(n))$  时间复杂度查找二叉树（假定二叉树没有退化成链表）中序遍历序列中第  $k$  ( $1 \leq k \leq n$ ) 个节点。二叉树节点中有附加域 `nodes` 表明当前子树的节点个数，比如空树没有节点，所以无法存储 `nodes`，就认为空树节点数为 0；只有一个节点的子树其根节点 `nodes` 为 1；其他树的根节点的 `nodes` 等于左右子树的 `nodes` 之和再加 1

---

```

1  #include "tree.h"
2  BiTree FindKthNodeInBiTree(BiTree tree, int k)
3  {
4      if (NULL == tree || k <= 0) return NULL;
5      int lsubtreeNodes = 0;
6      if (NULL != tree->lchild)
7          lsubtreeNodes = tree->nodes;
8      if (lsubtreeNodes + 1 == k)
9          return tree;
10     if (k <= lsubtreeNodes)
11         return FindKthNodeInBiTree(tree->lchild, k);
12     else
13         return FindKthNodeInBiTree(tree->rchild, k - lsubtreeNodes - 1);
14 }

```

---

## 5 广义表 \*

### 5.1 数据结构定义

广义表的字段 `isAtom` 用于表明此节点是一个原子节点还是普通节点。当 `isAtom` 为真时，此节点为原子节点，此时 `left` 和 `right` 字段没有意义；同理，当 `isAtom` 为假时，此节点为普通节点 `data` 字段没有意义<sup>4</sup>。

---

```

1  /* General List Node */
2  typedef struct GListNode {
3      int data;
4      struct GListNode *left, *right;
5      bool isAtom;
6  } GListNode;
7  /* General List */
8  typedef GListNode *GList;

```

---

### 基本操作

1. 判断一个节点是不是原子 `bool IsAtom(GList list);`

---

```

1  bool IsAtom(GList list)
2  {

```

---

<sup>4</sup>在 C 语言中更常用联合体 `union` 来表示这种结构

```

3     /* 空列表不是原子 */
4     if (NULL == list) return false;
5     return list->isAtom;
6 }

```

---

2. 判断一个节点是不是“节点对” (pair) `bool IsPair(GList list);`

```

1 bool IsPair(GList list)
2 {
3     /* 空列表不是节点对 */
4     if (NULL == list) return false;
5     return !IsAtom(list);
6 }

```

---

3. 判断一个列表是不是空列表 `bool IsNull(GList list);`

```

1 bool IsNull(GList list)
2 {
3     return (NULL == list);
4 }

```

---

4. 根据数据构建原子节点 `GList MakeAtom(int data);`

```

1 GList MakeAtom(int data)
2 {
3     GList node = malloc(sizeof(GListNode));
4     if (NULL == node) return NULL;
5     node->data = data;
6     node->isAtom = true;
7     return node;
8 }

```

---

5. 构建对 (pair) `GList MakePair(GList a, GList b);` 或者命名为 `GList cons(GList a, GList b);`。  
想知道为何是 `cons` 见 <sup>5 6</sup>。

```

1 GList MakePair(GList a, GList b)
2 {
3     GList node = malloc(sizeof(GListNode));
4     if (NULL == node) return NULL;
5     node->left = a;
6     node->right = b;
7     node->isAtom = false;
8     return node;
9 }

```

---

<sup>5</sup>[https://en.wikipedia.org/wiki/Lisp\\_\(programming\\_language\)#Conses\\_and\\_lists](https://en.wikipedia.org/wiki/Lisp_(programming_language)#Conses_and_lists)

<sup>6</sup><https://en.wikipedia.org/wiki/Cons>

6. 构建列表 GList `MakeList(int *arr, int len)`; 把数组中的元素构建成列表

---

```

1 GList MakeList(int *arr, int len)
2 {
3     if (NULL == arr || len <= 0) return NULL;    /* 空列表 */
4     GList list = NULL;
5     for (int i = len-1; i >= 0; i--)
6         list = MakePair(MakeAtom(arr[i]), list);
7     return list;
8 }

```

---

7. 取表头 GList `GetHead(GList list)`; 或者命名为 GList `car(GList list)`;

---

```

1 GList GetHead(GList list)
2 {
3     if (IsNull(list)) return NULL;    /* 空表表头还是空表 */
4     if (IsAtom(list)) return NULL;   /* 原子没法取“表头” */
5     return list->left;
6 }

```

---

8. 取除去表头剩下的部分 GList `GetTail(GList list)`; 或者命名为 GList `cdr(GList list)`;

---

```

1 GList GetTail(GList list)
2 {
3     if (IsNull(list)) return NULL;    /* 空表表尾还是空表 */
4     if (IsAtom(list)) return NULL;
5     return list->right;
6 }

```

---

9. 判断是不是列表 `bool IsList(GList list)`;

---

```

1 bool IsList(GList list)
2 {
3     if (IsNull(list)) return true;    /* 空表 */
4     /* 顺着列表往后找到最后一个非原子节点 */
5     while (IsPair(list))
6         list = GetTail(list);
7     /* 这个非原子节点必须是空表，那么才是列表 */
8     return IsNull(list);
9 }

```

---

## 5.2 例题

1. 构建广义列表 `list = (A, (G, H, (M)), D)`

---

```

1 GList listM = MakePair(MakeAtom('M'), NULL);    /* (M) */
2 GList listHM = MakePair(MakeAtom('H'), listM);   /* (H, (M)) */

```

---



---

```

3  GList listGHM = MakePair(MakeAtom('G'), listHM); /* (G, H, (M)) */
4  GList listD = MakePair(MakeAtom('D'), NULL);      /* (D) */
5  GList listGHMD = MakePair(listGHM, listD);        /* ((G, H, (M)), D) */
6  GList list = MakePair(MakeAtom('A'), listGHMD);   /* (A, (G, H, (M)), D) */

```

---

2. 在广义表  $list = (A, B, (F, C), D)$  中获取 C

---

```

1  GetTail(list);                                /* (B, (F, C), D) */
2  GetTail(GetTail(list));                       /* ((F, C), D) */
3  GetHead(GetTail(GetTail(list)));              /* (F, C) */
4  GetTail(GetHead(GetTail(GetTail(list))));     /* (C) */
5  GetHead(GetTail(GetHead(GetTail(GetTail(list))))); /* C */

```

---

3. 求广义表的深度，即 left 嵌套的深度，直观来说就是左括号 ( 的嵌套的最大深度。比如本章第 1 题表深度为 3

---

```

1  int DepthOfGList(GList list)
2  {
3      if (IsNull(list) || IsAtom(list)) return 0;
4      int leftDepth = DepthOfGList(GetHead(list));
5      int rightDepth = DepthOfGList(GetTail(list));
6      return max(1 + leftDepth, rightDepth);
7  }

```

---

4. 求广义表的长度，比如本章第 1 题表长度为 3

---

```

1  int LengthOfGList(GList list)
2  {
3      if (!IsList(list)) return 0;
4      int len = 0;
5      while (!IsNull(list)) {
6          len++;
7          list = GetTail(list);
8      }
9      return len;
10 }

```

---

## 6 图

### 6.1 数据结构定义

#### 遍历

1. 深度优先遍历
2. 广度优先遍历

## 6.2 例题

# 7 排序和查找

## 7.1 排序

### 7.1.1 插入排序

```
1 void InsertSort(int *arr, int len)
2 {
3     if (NULL == arr || len <= 0) return;
4     for (int i = 1; i < len; i++) {
5         /* 把 arr[i] 存储到 tmp, 留出 arr[i] 空位 */
6         int tmp = arr[i];
7         int j;
8         /* 把比 tmp 大的元素往后移, 给 tmp 留出空位 */
9         for (j = i; j-1 >= 0 && tmp < arr[j-1]; j -= 1) {
10             arr[j] = arr[j-1];
11         }
12         /* 插入 tmp 元素到空位 */
13         arr[j] = tmp;
14     }
```

### 7.1.2 折半插入排序

仅仅在查找元素的时候采用二分查找，但是移动元素开销不能避免，同时代码也会比简单的插入排序复杂一点点，所以不给出代码。

### 7.1.3 希尔排序 \*

希尔排序实际上是插入排序的范化版本，插入排序可以看成增量固定为 1 的希尔排序。希尔排序像是多次使用插入排序，每次插入排序的增量不断减小直到为 1。

- 以增量减半作为增量序列的希尔排序

```
1 void ShellSort(int *arr, int len)
2 {
3     if (NULL == arr || len <= 0) return;
4     int inc = len;
5     /* 最外层循环控制增量 */
6     for (int inc = len; inc >= 1; inc /= 2) {
7         /* 内层增量为 inc 的普通插入排序 */
8         for (int i = inc; i < len; i++) {
9             int tmp = arr[i];
10            int j;
11            for (j = i; j-inc >= 0 && tmp < arr[j-inc]; j -= inc)
```

```

12         arr[j] = arr[j-inc];
13         arr[j] = tmp;
14     }
15 }
16 }

```

- 执行指定增量序列的希尔排序

```

1  /**
2   * inc: 增量数组，按照从增量从大到小排序，最后一个元素一定为 1
3   * incLen: 增量数组的大小
4   */
5  void ShellSortWithInc(int *arr, int len, int *inc, int incLen)
6  {
7      if (NULL == arr || len <= 0 || NULL == inc || incLen <= 0) return;
8      /* 最后一个增量必须是 1 */
9      if (1 != inc[incLen-1])
10         return;
11     /* 控制增量 */
12     for (int k = 0; k < incLen; k++) {
13         for (int i = inc[k]; i < len; i++) {
14             int tmp = arr[i];
15             int j;
16             for (j = i; j-inc[k] >= 0 && tmp < arr[j-inc[k]]; j -= inc[k])
17                 arr[j] = arr[j-inc[k]];
18             arr[j] = tmp;
19         }
20     }
21 }

```

#### 7.1.4 冒泡排序

```

1  void Swap(int *a, int *b)
2  {
3      int t = *a;
4      *a = *b;
5      *b = t;
6  }
7  void BubbleSort(int *arr, int len)
8  {
9      if (NULL == arr || len <= 0) return;
10     for (int i = 0; i < len; i++) {
11         for (int j = len-1; j-1 >= i; j--) {
12             if (arr[j-1] > arr[j]) {
13                 /* &arr[j-1] 等价于 &(arr[j-1]) 表示求元素 arr[j-1] 的地址 */
14                 Swap(&arr[j-1], &arr[j]);
15             }
16         }
17     }
18 }

```

### 7.1.5 快速排序

```
1 int Partition(int *arr, int lo, int hi)
2 {
3     int pivot = arr[lo];
4     while (lo < hi) {
5         while (lo < hi && arr[hi] >= pivot)
6             -- hi;
7         arr[lo] = arr[hi];
8         while (lo < hi && arr[lo] <= pivot)
9             ++ lo;
10        arr[hi] = arr[lo];
11    }
12    arr[lo] = pivot;
13    return lo;
14 }
15 void _QuickSort(int *arr, int lo, int hi)
16 {
17     if (lo >= hi) return;
18     int pivotIdx = Partition(arr, lo, hi);
19     _QuickSort(arr, lo, pivotIdx-1);
20     _QuickSort(arr, pivotIdx+1, hi);
21 }
22 void QuickSort(int *arr, int len)
23 {
24     if (NULL == arr || len <= 0) return;
25     _QuickSort(arr, 0, len-1);
26 }
```

**应用** 快速排序中 Partition 是一个重要的想法。通过这个方法可以做到高效地求得无序数组中第  $k$  大的元素，采取的方法类似于树中第27题。

1. 求无序数组中第  $k$  ( $1 \leq k \leq n$ ) 大的元素。

### 7.1.6 选择排序

```
1 void SelectSort(int *arr, int len)
2 {
3     if (NULL == arr || len <= 0) return;
4     for (int i = 0; i < len; i++) {
5         int minIdx = i;
6         for (int j = i; j < len; j++) {
7             if (arr[j] < arr[minIdx])
8                 minIdx = j;
9         }
10        Swap(&arr[i], &arr[minIdx]);
11    }
12 }
```

### 7.1.7 堆排序 \*

TODO 实现堆排序

### 7.1.8 归并排序

1. 递归版本实现，也是最常见的实现方式

```
1 void CopyArray(int *arr, int lo, int hi, int *tmpArr)
2 {
3     for (int i = 0; i <= hi-lo; i++)
4         arr[lo+i] = tmpArr[i];
5 }
6 /**
7  * 版本一：递归实现的归并排序
8  */
9 void _MergeSort(int *arr, int lo, int hi, int *tmpArr)
10 {
11     if (lo >= hi) return;
12     int mid = lo+(hi-lo)/2;
13     int lo1 = lo;
14     int lo2 = mid+1;
15     _MergeSort(arr, lo1, mid, tmpArr);
16     _MergeSort(arr, lo2, hi, tmpArr);
17     /* 合并 */
18     int idx = 0;
19     while (lo1 <= mid && lo2 <= hi) {
20         if (arr[lo1] < arr[lo2]) {
21             tmpArr[idx] = arr[lo1];
22             lo1++;
23         } else {
24             tmpArr[idx] = arr[lo2];
25             lo2++;
26         }
27         idx++;
28     }
29     while (lo1 <= mid) {
30         tmpArr[idx] = arr[lo1];
31         idx++;
32         lo1++;
33     }
34     while (lo2 <= hi) {
35         tmpArr[idx] = arr[lo2];
36         idx++;
37         lo2++;
38     }
39     /* 把合并后的数组复制回原来数组的区间 [lo, hi] 里 */
40     CopyArray(arr, lo, hi, tmpArr);
41 }
42 void MergeSort(int *arr, int len)
43 {
44     if (NULL == arr || len <= 0) return;
```

```

45     int *tmpArr = malloc(len * sizeof(int));
46     if (NULL == tmpArr) return;
47     MergeSort(arr, 0, len-1, tmpArr);
48     free(tmpArr);
49 }

```

---

## 2. 非递归实现 \*, 从底向上合并的思想

```

1  int min(int x, int y)
2  {
3      return (x<y)? x: y;
4  }
5  /**
6   * 版本二: 非递归实现的归并排序, 自底向上
7   */
8  void MergeSort(int *arr, int len)
9  {
10     if (NULL == arr || len <= 0) return;
11     int *tmpArr = malloc(len * sizeof(int));
12     if (NULL == tmpArr) return;
13     for (int step = 1; step < len; step *= 2) {
14         for (int i = 0; i < len; i += 2*step) {
15             int idx = 0;
16             int lo1 = i;
17             int hi1 = min(i+step, len) - 1;
18             int lo2 = i+step;
19             int hi2 = min(i+2*step, len) - 1;
20             while (lo1 <= hi1 && lo2 <= hi2) {
21                 if (arr[lo1] < arr[lo2]) {
22                     tmpArr[idx] = arr[lo1];
23                     lo1 ++;
24                 } else {
25                     tmpArr[idx] = arr[lo2];
26                     lo2 ++;
27                 }
28                 idx ++;
29             }
30             while (lo1 <= hi1) {
31                 tmpArr[idx] = arr[lo1];
32                 idx ++;
33                 lo1 ++;
34             }
35             while (lo2 <= hi2) {
36                 tmpArr[idx] = arr[lo1];
37                 idx ++;
38                 lo2 ++;
39             }
40             CopyArray(arr, i, hi2, tmpArr);
41         }
42     }
43     free(tmpArr);
44 }

```

---

### 7.1.9 基数排序 \*

基数排序只能用于特定类型的数据的排序，比如整数或字符串，并不是一个通用型排序算法，并且实现需要基数个队列。

```
1  #include "queue.h"
2  /* 基数选择为 10 */
3  #define Radix 10
4  /**
5   * 按照最低位优先 LSD 的基数排序
6   */
7  void RadixSort(int *arr, int len)
8  {
9      if (NULL == arr || len <= 0) return;
10     /* 需要 Radix 个队列 */
11     LinkQueue queues[Radix];
12     int radixLo = 1;
13     for (int i = 0; i < Radix; i++)
14         InitQueue(&queues[i]);
15
16     bool doned = false;
17     while (!doned) {
18         /* 把元素按照数位上的数放入对应的队列中 */
19         for (int i = 0; i < len; i++) {
20             int digit = (arr[i] / radixLo) % Radix;
21             EnQueue(&queues[digit], arr[i]);
22         }
23
24         /* 收集 */
25         int idx = 0;
26         int ele;
27         for (int i = 0; i < Radix; i++) {
28             while (!QueueEmpty(&queues[i])) {
29                 DeQueue(&queues[i], &ele);
30                 arr[idx] = ele;
31                 idx++;
32             }
33         }
34
35         radixLo *= Radix;
36         /**
37          * 如果所有元素都比 radixLo 小，那么说明所有元素的最高位已经排序，
38          * 否则元素还没有处理完，还需要继续排序 doned = false
39          */
40         doned = true;
41         for (int i = 0; i < len; i++) {
42             if (arr[i] > radixLo)
43                 doned = false;
44         }
45     }
46
47     for (int i = 0; i < Radix; i++)
48         DestroyQueue(&queues[i]);
```

49 }

## 7.2 查找

### 7.2.1 顺序查找

顺序查找一般用于无序的数组内查找元素。

```
1  /* 查找并返回元素下标 */
2  int SequenceSearch(int *arr, int len, int x)
3  {
4      if (NULL == arr || len <= 0) return -1;
5      for (int i = 0; i < len; i++) {
6          if (x == arr[i])
7              return i;
8      }
9      return -1;
10 }
```

### 7.2.2 二分查找

对于已经有序的数组，二分查找是更为高效的算法，一般假定数据从小到大排序。

```
1  int BinarySearch(int *arr, int len, int x)
2  {
3      if (NULL == arr || len <= 0) return;
4      int lo = 0, hi = len-1;
5      while (lo <= hi) {
6          int mid = lo+(hi-lo)/2;
7          if (x == arr[mid])
8              return mid;
9          if (x < arr[mid])
10             hi = mid-1;
11         else
12             lo = mid+1;
13     }
14     return -1;
15 }
```