数据结构和算法代码汇总

αlpha0x00
leetiankai@gmail.com

2020 年 9 月 17 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

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：

```c
#define MaxSize 1024
typedef struct SeqList {
    int data[MaxSize];
    int length; /* 记录存储元素的个数 */
} 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。

```c
typedef struct SeqList {
    int *data;
    int length, capacity;   /* capacity 记录顺序表可存储元素最大个数 */
} 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 [1]。

```c
/**
 * bool 在 C99 以标准类型添加到 C 语言中，来自头文件 <stdbool.h>
 */
bool InitList(SeqList *list, int capacity)
{
    if (NULL == list || capacity <= 0) return false;

    list->data = malloc(capacity * sizeof(int));
    if (NULL == list->data) return false;

    list->length = 0;
    list->capacity = capacity;
    return true;
}
```

[1] 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：

```
bool InitList(SeqList *list)
{
    if (NULL == list) return false;
    list->length = 0;
    return true;
}
```

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

```
void DestroyList(SeqList *list)
{
    free(list->data);
}
```

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

```
int ListLength(SeqList *list)
{
    if (NULL == list) return 0;
    return list->length;
}
```

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

```
int ListCapacity(SeqList *list)
{
    if (NULL == list) return 0;
    return list->capacity;
    /* 对于静态顺序表 */
    //return MaxSize;
}
```

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

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

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

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

```
    return (ListLength(list) == ListCapacity(list));
}
```

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

```
/**
 * 返回 -1 表明没有找到或者出错
 */
int LocateElem(SeqList *list, int ele)
{
    if (NULL == list) return -1;
    for (int i = 0; i < Length(list); i++) {
        if (ele == list->data[i])
            return i;
        /* 对于静态顺序表 */
        //if (ele == list.data[i])
        //    return i;
    }
    /* 没有找到 */
    return -1;
}
```

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

```
int GetElem(SeqList *list, int idx)
{
    /* 假定元素都是大于 0，返回 -1 表明发生下标越界错误
     * 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
     * bool GetElem(SeqList *list, int idx, int *store);
     * 返回 ture，表明正确获取数据，存储到指针 *store 里，
     * 返回 false，表明下标越界
     */
    if (idx < 0 || idx > ListLength(list))
        return -1;
    return list->data[idx];
    /* 对于静态顺序表 */
    //return list.data[idx];
}
```

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

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

```
    }
    list->data[idx] = ele;
    return true;
}
```

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

```
bool ListDelete(SeqList *list, int idx, int *ele)
{
    if (idx < 0 || idx > ListLength(list)) return false;
    for (int i = idx; i < ListLength(list)-1; i++) {
        list->data[i] = list->data[i+1];
        /* 对于静态顺序表 */
        // list.data[i] = list.data[i+1];
    }
    list->length = list->length-1;
    return true;
}
```

1.1.2 单链表

```
typedef struct LinkNode {
    int data;
    struct LinkNode *next;
} LinkNode;
typedef LinkNode *LinkList;

/* 分配一个节点 */
LinkList node = malloc(sizeof(LinkNode));
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

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

```
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 InitDLinkList(DLinkList *dlist);`

```
bool InitDLinkList(DLinkList *dlist)
{
    if (NULL == dlist) return false;
    *dlist = malloc(sizeof(DLinkNode));
    if (NULL == *dlist) return false;
    /* 循环双链表 */
    (*dlist)->prev = (*dlist)->next = *dlist;
    /* 普通双链表 */
    // (*dlist)->prev = (*dlist)->next = NULL;
    return true;
}
```

按照如下方式定义双链表并初始化：

```
DLinkList list;          /* 定义双链表 list */
InitDLinkList(&list);    /* 初始化双链表 list */
```

2. 判断是否是空链表 `bool DLinkListEmpty(DLinkList list);`

```
bool DLinkListEmpty(DLinkList list)
{
    if (NULL == list) return true;
    return (list->next = list) && (list->prev = list);
    /* 普通双链表 */
    //return (list->next == NULL) && (list->prev == NULL);
}
```

3. 求表长度 `int DLinkListLength(DLinkList list);`

```
int DLinkListLength(DLinkList list)
{
    if (NULL == list) return 0;
    int len = 0;
    /* 让 p = list->next 是为了去掉头节点 */
    for (DLinkList p = list->next; list != p; p = p->next)
        len++;

    /* 普通双链表 */
    //for (DLinkList p = list->next; NULL != p; p = p->next)
    //    len++;

    return len;
}
```

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

   - 在循环双链表中插入元素

```c
/**
 * 插入元素 ele 到插入点 at 之后
 */
bool InsertDLinkListAt(DLinkList at, int ele)
{
    if (NULL == at) return false;
    /* 分配好新插入的节点 node */
    DLinkNode *node = malloc(sizeof(DLinkNode));
    if (NULL == node) return false;
    node->data = ele;

    DLinkList next = at->next;
    /**
     * 插入后节点构成顺序 at, node, next
     * 所以上述语句 next = at->next, 是为了获取 node 的后继节点
     * 然后直接连接起来即可
     */
    node->next = next;
    node->prev = at;
    next->prev = node;
    at->next = node;
    return true;
}
```

   - 在非循环双链表中插入元素

```c
/**
 * 插入元素 ele 到插入点 at 之后
 */
bool InsertDLinkListAt(DLinkList at, int ele)
{
    if (NULL == at) return false;
    /* 分配好新插入的节点 node */
    DLinkNode *node = malloc(sizeof(DLinkNode));
    if (NULL == node) return false;
    node->data = ele;

    DLinkList next = at->next;
    /**
     * 插入后节点构成顺序 at, node, next
     * 所以上述语句 next = at->next, 是为了获取 node 的后继节点
     * 然后直接连接起来即可
     */
    node->next = next;
    node->prev = at;
    /**
     * 当 at 就是最后一个节点时, at->next 肯定是空指针
     * 所以没有“后继”节点指向 node
     */
```

```
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 DeleteDLinkList(DLinkList node);`

   - 从循环双链表中删除元素

```
1   bool DeleteDLinkList(DLinkList node)
2   {
3       if (NULL == node) return false;
4       /* 获取被删除节点的前驱和后继 */
5       DLinkList prev = node->prev;
6       DLinkList 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 DeleteDLinkList(DLinkList node)
2   {
3       if (NULL == node) return false;
4       /* 获取被删除节点的前驱和后继 */
5       DLinkList prev = node->prev;
6       DLinkList 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

```c
#include "list.h"
void DeleteAllxFromList(LinkList *plist, int x)
{
    /**
     * 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
     * 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，
     * 二级指针就是握住牙签的手
     * 空链表：就是一根光溜溜的牙签
     */
    if (NULL == plist) return;

    while (*plist != NULL) {
        /**
         * plist 是手，*plist 就是手上的牙签，(*plist)->next 下一个汤圆的牙签
         * 让“手”握住下一个“汤圆”的“牙签”
         */
        LinkList *pnext = &(*plist)->next;  /* -> 优先级高于 & */
        if (x == (*plist)->data) {
            free(*plist);
            *plist = *pnext;
        } else {
            plist = pnext;
        }

    }
}
```

2. 删除带头结点单链表 list 中所有值为 x 结点

```c
#include "list.h"
void DeleteAllxFromList2(LinkList list, int x)
{
    if (NULL == list || NULL == list->next) return;
    while (NULL != list->next) {
        LinkList curr = list->next;
        if (x == curr->data) {
            free(curr);
            list->next = curr->next;
        } else {
            list = list->next;
        }
    }
}
```

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

```c
#include "list.h"
/**
 * 注意这里带有头节点
 */
void _PrintListRevsersely(LinkList list)
{
    /* 按照没有头节点单链表来处理 */
    if (NULL == list) return;
    _PrintListRevsersely(list->next);
    printf("%d ", list->data);
}
void PrintListReversely(LinkList list)
{
    /* 对于空链表（带有头节点）直接返回不进行处理 */
    if (NULL == list || NULL == list->next) return;
    list = list->next;  /* 去除头节点 */
    _PrintListRevsersely(list);
}
```

- 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）

```c
void PrintListReversely(LinkList list)
{
    if (NULL == list || NULL == list->next) return;
    list = list->next;  /* 去除头节点 */
    /* 存储元素的地址，减少直接复制元素带来的开销 */
    int values[MaxSize];
    int cnt;
    for (cnt = 0; cnt < MaxSize && NULL != list; list = list->next) {
        values[cnt] = list->data;
        cnt ++;
    }
    if (cnt > MaxSize) {
        printf("the amount of data in list is greater than MaxSize(%d).",
                MaxSize);
        return;
    }
    /* 反向输出 */
    while (cnt-- > 0)
        printf("%d ", values[cnt]);
}
```

4. 删除带头结点单链表 list 中最小值的节点

```c
#include "list.h"
void DeleteMinimalValueFromList(LinkList list)
{
    if (NULL == list || NULL == list->next) return;
```

```
    /* INT_MAX 来自头文件 limits.h, 典型值为 2^31 (2 的 31 次方) */
    int minValue = INT_MAX; /* 存储最小值 */
    LinkList minPrev;       /* 指向最小值节点的前驱节点 */
    /* 找到最小值的节点的前驱节点 */
    for (; NULL != list->next; list = list->next) {
        LinkList curr = list->next;
        if (curr->data < minValue) {
            minValue = curr->data;
            minPrev = list;
        }
    }
    /* 删除最小值的节点 */
    LinkList minNode = minPrev->next;
    minPrev->next = minNode->next;
    free(minNode);
}
```

5. 就地逆置带头结点单链表 list

```
#include "list.h"
void ReverseListInPlace(LinkList list)
{
    if (NULL == list || NULL == list->next) return;
    LinkList node, next;
    /* 把 list 断开成两个链表: list 空链表, node 余下节点 */
    node = list->next;
    list->next = NULL;
    for (; NULL != node; node = next) {
        next = node->next;
        /* 插入头节点之后, 普通节点之前 */
        node->next = list->next;
        list->next = node;
    }
}
```

6. 排序带头结点单链表 list 为升序

```
#include "list.h"
/* 采用插入排序法 */
void SortList(LinkList list)
{
    if (NULL == list || NULL == list->next) return;
    LinkList node = list->next;
    list->next = NULL;
    /* 把余下节点 node, 插入到新链表 list */
    LinkList next;
    for (; NULL != node; node = next) {
        next = node->next;
        /* 寻找 */
        LinkList prev = list;
        while (node->data > prev->data && NULL != prev->next)
```

```
            prev = prev->next;
        /* 插入 */
        node->next = prev->next;
        prev->next = node;
    }
}
```

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

```
#include "list.h"
/**
 * 整体思路和从 list 中删除 x 一样，只是条件不同
 */
void DeleteValuesBetweenLoHiFromList(LinkList list, int lo, int hi)
{
    if (NULL == list || NULL == list->next || lo > hi) return;
    while (NULL != list->next) {
        LinkList curr = list->next;
        if (lo <= curr->data && curr->data <= hi) {
            list->next = curr->next;
            free(curr);
        } else {
            list = list->next;
        }
    }
}
```

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。

```
#define MaxSize 1024
typedef struct SeqStack {
    int data[MaxSize];
    int top;
} SeqStack;
```

动态分配

```
typedef struct SeqStack {
    int *data;
    int top, capacity;
} SeqStack;
```

基本操作

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

```
bool InitStack(SeqStack *stack, int capacity)
{
    if (NULL == stack || capacity <= 0) return false;
    stack->top = -1;
    stack->capacity = capacity;
    stack->data = malloc(capacity * sizeof(int));
    if (NULL == stack->data) return false;
    return true;
}
```

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);`

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

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

```
int StackCapacity(SeqStack *stack)
{
    if (NULL == stack) return 0;
    return stack->capacity;
    /* 对于静态分配的栈 */
    //return MaxSize;
}
```

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

```
bool StackEmpty(SeqStack *stack)
{
```

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

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

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

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

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

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

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

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

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

2.1.2 链式存储的栈

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

基本操作

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

```c
bool InitStack(LinkStack *stack)
{
    if (NULL === stack) return false;
    stack->size = 0;
    stack->top = NULL;  /* 不带头节点的单链表 */
    return true;
}
```

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);`

```c
bool Push(LinkStack *stack, int ele)
{
    if (NULL == stack) return false;
    /* 链表实现的栈的容量为无穷大，所以不存在栈满的情况 */
    LinkNode *node = malloc(sizeof(LinkNode));
    if (NULL == node) return false;
    node->data = ele;

    node->next = stack->top;
    stack->top = node;

    stack->size += 1;
    return true;
}
```

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

```c
bool Pop(LinkStack *stack, int *store)
{
    if (NULL == stack || NULL == store) return false;
    if (StackEmpty(stack)) return false;
    LinkList node = stack->top;
    stack->top = node->next;

    *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。

```
#include "stack.h"
#define CapacityMax 4096
bool IsBracketsValid(char *brackets)
{
    bool valid = true;
    SeqStack stack;
    InitStack(&stack, CapacityMax);
    int ch;
    for (int i = 0; '\0' != brackets[i]; i++) {
        switch (brackets[i]) {
        /* 左“括号”入栈 */
        case '(': Push(&stack, '('); break;
        case '[': Push(&stack, '['); break;
        case '{': Push(&stack, '{'); break;
        /* 右“括号”出栈 */
        case ')':
            if (StackEmpty(&stack)) {
                DestroyStack(&stack);
                return false;
            }
            Pop(&stack, &ch);
            if ('(' != ch) {
                DestroyStack(&stack);
                return false;
            }
            break;
        case ']':
            if (StackEmpty(&stack)) {
                DestroyStack(&stack);
                return false;
            }
            Pop(&stack, &ch);
            if ('[' != ch) {
                DestroyStack(&stack);
                return false;
            }
            break;
        case '}':
```

```
            if (StackEmpty(&stack)) {
                DestroyStack(&stack);
                return false;
            }
            Pop(&stack, &ch);
            if ('{' != ch) {
                DestroyStack(&stack);
                return false;
            }
            break;
        default:
            break;
        }
    }
    /* 栈不为空表明全是左括号 (((({{{{[[[[ 的情况 */
    if (!StackEmpty(&stack)) {
        DestroyStack(&stack);
        return false;
    }

    DestroyStack(&stack);
    return true;
}
```

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

```
typedef struct SeqQueue {
    int *data;
    int capacity;
    int front, rear;
    /* 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 */
    // int size;
} SeqQueue;
```

基本操作

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

```
bool InitQueue(SeqQueue *queue, int capacity)
{
    if (NULL == queue || capacity <= 0) return false;
    queue->data = malloc(capacity * sizeof(int));
    if (NULL == queue->data) return false;
    queue->capacity = capacity;
    queue->front = queue->rear = 0;
    //queue->size = 0;
    return true;
}
```

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

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

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

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

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

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

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

```
bool EnQueue(SeqQueue *queue, int ele)
{
    if (NULL == queue || QueueFull(queue)) return false;
    queue->data[queue->rear] = ele;
    queue->rear = (queue->rear+1) % queue->capacity;
    //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，如果需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);`

```c
bool DeQueue(LinkQueue *queue, int *store)
{
    if (NULL == queue || NULL == store) return false;
    if (QueueEmpty(queue)) return false;

    /**
     * front->[dummy]->[node2]->...->[noden]->NULL
     * rear ---------------------------^
     * node ->[node1]
     */
    LinkList node = queue->front->next;
    queue->front->next = node->next;
    /* 如果是最后一个元素 */
    if (node == queue->rear)
        queue->rear = queue->front;
    queue->size -= 1;

    *store = node->data;
    free(node);
    return true;
}
```

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

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

□

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：

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

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 $m=\left\lfloor \frac{n}{2} \right\rfloor$

```c
#define TreeNodeMax 2048
/* Parent Tree Node */
typedef struct PTNode {
    int data;
    int parent;     /* -1 表明没有双亲节点 */
} PTNode;
/* Parent Tree */
typedef struct PTree {
    PTNode *nodes;
    int capacity;   /* nodes 的总个数 */
} 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

```
typedef struct UFSet {
    int *nodes;
    int capacity;
} UFSet;
```

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

```
bool InitUFSet(UFSet *set, int capacity)
{
    if (NULL == set || capacity <= 0) return false;
    set->nodes = malloc(capacity * sizeof(int));
    if (NULL == set->nodes) return false;
    set->capacity = capacity;
    /* 所有集合初始化为空集 */
    for (int i = 0; i < capacity; i++)
        set->nodes[i] = -1;
    return true;
}
```

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

```
void DestroyUFSet(UFSet *set)
{
    if (NULL == set) return;
    free(set->nodes);
}
```

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

```
bool MakeSet(UFSet *set, int id)
{
    if (NULL == set) return false;
    if (id < 0 || id >= set->capacity) return false;
    /* 元素 id 已经添加 */
    if (set->nodes[id] != -1) return false;
    set->nodes[id] = id;
    return true;
}
```

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

```
/**
 * 返回 -1 表示 id 不在任何集合内
```

```
 * 返回 >= 0 表示 id 所在集合的代表元素
 */
int Find(UFSet *set,  int id)
{
    if (NULL == set) return -1;
    if (id < 0 || id >= set->capacity) return -1;
    while (true) {
        int parent = set->nodes[id];
        if (parent == id || parent == -1)
            return parent;
        id = parent;
    }
}
```

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

```
bool Same(UFSet *set, int x, int y)
{
    int xparent = Find(set, x);
    int yparent = Find(set, y);
    if (-1 == xparent || -1 == yparent)
        return false;
    return (xparent == yparent);
}
```

- 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[2]。

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

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。

[2] https://en.wikipedia.org/wiki/Disjoint-set_data_structure

```
typedef struct PTreeNode {
    int data;
    struct PTreeNode *parent;
} PTreeNode;
typedef PTreeNode *PTree;
```

4.1.2 孩子兄弟存储结构

```
/* Child Sibling Tree */
typedef struct CSTreeNode {
    int data;
    struct CSTreeNode *firstchild, *nextsibling;
} CSTreeNode;
typedef CSTreeNode *CSTree;
```

4.1.3 二叉树、二叉排序数和平衡二叉树

```
/* Binary Tree */
typedef struct BiTreeNode {
    int data;
    struct BiTreeNode *lchild, *rchild;
} BiTreeNode;
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

   ```
   void PreOrder(BiTree tree)
   {
       if (NULL == tree) return;
       /* 访问节点 tree */
       //visit(tree);
       PreOrder(tree->lchild);
       PreOrder(tree->rchild);
   }
   ```

   - 非递归实现
     - 版本一

```
void PreOrder(BiTree tree)
{
#define TreeHeightMax 2048
    SeqStack stack;
    InitStack(&stack, TreeHeightMax);

    Push(&stack, tree);
    while (!StackEmpty(&stack)) {
        BiTree node;
        Pop(&stack, &node);
        if (NULL == node)
            continue;
        /* 访问节点 node */
        //visit(node);
        /* 先右节点入栈，然后是左节点入栈 */
        Push(&stack, node->rchild);
        Push(&stack, node->lchild);
    }

    DestroyStack(&stack);
}
```

- 版本二 *

```
void PreOrder(BiTree tree)
{
#define TreeHeightMax 2048
    SeqStack stack;
    InitStack(&stack, TreeHeightMax);

    BiTree curr = tree;
    while (NULL != curr || !StackEmpty(&stack)) {
        if (NULL != curr) {
            /* 访问当前节点 curr */
            //visit(curr);
            Push(&stack, curr);
            curr = curr->lchild;
        } else {
            Pop(&stack, &curr);
            curr = curr->rchild;
        }
    }

    DestroyStack(&stack);
}
```

2. 中序遍历 `void InOrder(BiTree tree);`

- 递归实现

```
void InOrder(BiTree tree)
{
```

```
    if (NULL == tree) return;
    InOrder(tree->lchild);
    /* 访问节点 tree */
    //visit(tree);
    InOrder(tree->rchild);
}
```

- 非递归实现

```
void InOrder(BiTree tree)
{
#define TreeHeightMax 2048
    SeqStack stack;
    InitStack(&stack, TreeHeightMax);

    BiTree curr = tree;
    while (NULL == curr || !StackEmpty(&stack)) {
        if (NULL != curr) {
            Push(&stack, curr);
            curr = curr->lchild;
        } else {
            Pop(&stack, &curr);
            /* 访问节点 node */
            //visit(node);
            curr = curr->rchild;
        }
    }

    DestroyStack(&stack);
}
```

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

- 递归实现

```
void PostOrder(BiTree tree)
{
    if (NULL == tree) return;
    PostOrder(tree->lchild);
    PostOrder(tree->rchild);
    /* 访问节点 tree */
    //visit(tree);
}
```

- 非递归实现 [3]

```
void PostOrder(BiTree tree)
{
#define TreeHeightMax 2048
    SeqStack stack;
```

[3] https://en.wikipedia.org/wiki/Tree_traversal#Post-order_(LRN)

```
    InitStack(&stack, TreeHeightMax);
    BiTree lastVisited = NULL;
    BiTree curr = tree;

    while (NULL != curr || !StackEmpty(&stack)) {
        if (NULL != curr) {
            Push(&stack, curr);
            curr = curr->lchild;
        } else {
            GetTop(&stack, &curr);  /* 只获取栈顶，不弹栈 */
            if (NULL != curr->rchild && lastVisited != curr->rchild) {
                curr = curr->rchild;
            } else {
                /* 访问当前节点 */
                //visit(curr);
                Pop(&stack, &lastVisited);
            }
        }
    }

    DestroyStack(&stack);
}
```

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

```
void LevelOrder(BiTree tree)
{
#define TreeWidthMax 4096
    if (NULL == tree) return;
    SeqQueue queue;
    InitQueue(&queue, TreeNodesMax);

    EnQueue(&queue, tree);
    while (!QueueEmpty(&queue)) {
        BiTree node;
        DeQueue(&queue, &node);
        /* 访问节点 */
        //visit(node);
        if (NULL != node->lchild)
            EnQueue(&queue, node->lchild);
        if (NULL != node->rchild)
            EnQueue(&queue, node->rchild);
    }

    DestroyQueue(&queue);
}
```

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

```
bool BiTreeSearch(BiTree tree, int data)
{
    if (NULL == tree) return false;
    if (data == tree->data)
        return true;
    if (data < tree->data)
        return BiTreeSearch(tree->lchild, data);
    else
        return BiTreeSearch(tree->rchild, data);
}
```

- 非递归实现

```
bool BiTreeSearch(BiTree tree, int data)
{
    while (NULL != tree) {
        if (data == tree->data)
            return true;
        if (data < tree->lchild)
            tree = tree->lchild;
        else
            tree = tree->rchild;
    }

    return false;
}
```

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

```
#include "tree.h"
bool BiTreeInsert(BiTree *ptree, int data)
{
    /* 和向不带头节点单链表中插入数据极其类似 */
    while (NULL != *ptree) {
        if (data == (*ptree)->data)
            return true;
        if (data < (*ptree)->data) {
            ptree = &(*ptree)->lchild;
        } else {
            ptree = &(*ptree)->rchild;
        }
    }

    BiTreeNode *node = malloc(sizeof(BiTreeNode));
    if (NULL == node) return false;
    node->data = data;
    node->lchild = node->rchild = NULL;
    *ptree = node;

    return true;
}
```

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

```
BiTree CreateBSTree(int *array, int n)
{
    if (NULL == array || n <= 0) return NULL;

    BiTree tree = NULL;
    for (int i = 0; i < n; i++)
        BiTreeInsert(&tree, array[i]);

    return tree;
}
```

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

```
#include "tree.h"
/**
 * 从树中偷取其最大值的节点，并存储到 store 里
 */
void StealMaxNode(BiTree *ptree, BiTree *store)
{
    while (NULL != (*ptree)->rchild)
        ptree = &(*ptree)->rchild;
    *store = *ptree;
    *ptree = (*store)->lchild;
}
bool BiTreeDelete(BiTree *ptree, int data)
{
    if (NULL == ptree) return false;
    while (NULL != *ptree && data != (*ptree)->data) {
        if (data < (*ptree)->data)
            ptree = &(*ptree)->lchild;
        else
            ptree = &(*ptree)->rchild;
    }
    /* 没有值为 data 的节点 */
    if (NULL == *ptree) return false;

    BiTree lsubtree = (*ptree)->lchild;
    BiTree rsubtree = (*ptree)->rchild;
    free(*ptree);

    /* 左右都有子树的节点的删除 */
    if (NULL != lsubtree && NULL != rsubtree) {
        BiTree maxNode;
        StealMaxNode(&lsubtree, &maxNode);
        *ptree = maxNode;
        (*ptree)->lchild = lsubtree;
        (*ptree)->rchild = rsubtree;
        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

```
typedef struct ThreadTreeNode {
    int data;
    struct ThreadTreeNode *lchild, *rchild;
#define TagChild    0
#define TagPrevNode 1
#define TagNextNode 1
    int ltag, rtag;
} ThreadTreeNode;
/**
 * ltag 为 TagChild    (0) 时, lchild 指向左孩子节点
 * ltag 为 TagPrevNode (1) 时, lchild 指向前驱节点
 * rtag 为 TagChild    (0) 时, rchild 指向右孩子结点
 * rtag 为 TagNextNode (1) 时, rchild 指向后继节点
 */
 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);

```
#include "tree.h"
void _CreatePreThreadTree(ThreadTree tree, ThreadTree *pprev)
{
    if (NULL == tree) return;

    /* 前驱指向当前节点 */
    if (NULL != *pprev && NULL == (*pprev)->rchild) {
        (*pprev)->rchild = tree;
        (*pprev)->rtag = TagNextNode;
    }
    /* 当前节点指向前驱 */
    if (NULL == tree->lchild) {
        tree->lchild = *pprev;
        tree->ltag = TagPrevNode;
    }
    pprev = &tree;

    if (TagChild == tree->ltag)
        _CreatePreThreadTree(tree->lchild, pprev);
    if (TagChild == tree->rtag)
        _CreatePreThreadTree(tree->rchild, pprev);
}
void CreatePreThreadTree(ThreadTree tree)
{
    ThreadTree prev = NULL;
    _CreatePreThreadTree(tree, &prev);
}
```

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

```
#include "tree.h"
void _CreateInThreadTree(ThreadTree tree, ThreadTree *pprev)
{
    if (NULL == tree) return;

    if (TagChild == tree->ltag)
        _CreateInThreadTree(tree->lchild, pprev);

    if (NULL != *pprev && NULL == (*pprev)->rchild) {
        (*pprev)->rchild = tree;
        (*pprev)->rtag = TagNextNode;
    }
    if (NULL == tree->lchild) {
        tree->lchild = *pprev;
        tree->ltag = TagPrevNode;
```

```
    }
    pprev = &tree;

    if (TagChild == tree->rtag)
        _CreateInThreadTree(tree->rchild, pprev);
}
void CreateInThreadTree(ThreadTree tree)
{
    ThreadTree prev = NULL;
    _CreateInThreadTree(tree, &prev);
}
```

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

```
#include "tree.h"
void _CreatePostThreadTree(ThreadTree tree, ThreadTree *pprev)
{
    if (NULL == tree) return;

    if (TagChild == tree->lchild)
        _CreatePostThreadTree(tree->lchild, pprev);
    if (TagChild == tree->rchild)
        _CreatePostThreadTree(tree->rchild, pprev);

    if (NULL != *pprev && NULL == (*pprev)->rchild) {
        (*pprev)->rchild = tree;
        (*pprev)->rtag = TagNextNode;
    }
    if (NULL == tree->lchild) {
        tree->lchild = *pprev;
        tree->ltag = TagPrevNode;
    }
    pprev = &tree;
}
void CreatePostThreadTree(ThreadTree tree)
{
    ThreadTree prev = NULL;
    _CreatePostThreadTree(tree, &prev);
}
```

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

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

```
{
    /**
     * 注意：比较困难，可能需要遍历整棵树
     * 因为，如果 node->ltag == TagPrevNode,
     *          只需要返回 node->lchild
     *       否则前驱节点就是 node 的父节点，
     *       然而在二叉树孩子节点表示法中，难以找到父节点
     */
}
/* 在先序线索树中，找先序序列的最后一个节点 */
ThreadTree LastNode(ThreadTree tree)
{
    ThreadTree nextNode;
    while (true) {
        nextNode = NextNode(tree);
        if (NULL == nextNode)
            return tree;
        tree = nextNode;
    }
}
/* 在先序线索树中，找先序序列的第一个节点 */
ThreadTree FirstNode(ThreadTree tree)
{
    return tree;
}
void PreOrderOfPreThreadTree(ThreadTree tree)
{
    /**
     * 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
     * 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
     * 点；同理，如果 NextNode() 为 NULL，说明这个节点为最后一个节点
     */
    ThreadTree node;
    for (node = FirstNode(tree); NULL != node; node = NextNode(node)) {
        /* 访问节点 */
        //visit(node);
    }
}
```

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

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

```
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        */
```

```
}
ThreadTree PrevNode(ThreadTree node)
{
    if (TagChild == node->rtag)
        return node->rchild;
    return node->lchild;
}
ThreadTree FirstNode(ThreadTree tree)
{
    if (NULL == tree) return tree;
    /* 第一个节点是“最左边”的第一个叶节点 */
    while (true) {
        if (TagChild == tree->ltag)
            tree = tree->lchild;
        else if (TagChild == tree->rtag)
            tree = tree->rchild;
        else
            break;
    }
    return tree;
}
ThreadTree LastNode(ThreadTree tree)
{
    return tree;
}
void PostOrderOfPostThread(ThreadTree tree)
{
    /**
     * 后序线索树想要后序遍历，因为 NextNode() 函数难以实现，
     * 所以后序遍历也难以实现
     * 但是，逆后序遍历是容易的，下面给出逆后序遍历
     */
    ThreadTree node;
    for (node = LastNode(tree); NULL != node; node = PrevNode(node)) {
        /* 访问节点 */
        //visit(node);
    }
}
```

对比先序线索树的先序遍历，可以发现后序线索树的后续遍历与其有一定的对称性。

4.2 例题

1. 求二叉树的高度

```
#include "tree.h"
/* 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 */
int BiTreeHeight(BiTree tree)
{
    if (NULL == tree) return 0;
    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   {
```

```
    while (NULL == tree || x == tree->data) {
        printf("%d ", tree->data);
        if (x < tree->data)
            tree = tree->lchild;
        else
            tree = tree->rchild;
    }
}

/* 一般二叉树 */
bool _PrintAncestors(BiTree tree, int x)
{
    if (NULL == tree) return false;
    if (x == tree->data)
        return true;
    /* 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 */
    if (_PrintAncestors(tree->lchild, x) || _PrintAncestors(tree->rchild, x)) {
        printf("%d ", tree->data);
        return true;
    }
    return false;
}
void PrintAncestors(BiTree tree, int x)
{
    _PrintAncestors(tree, x);
}
```

6. 判断给定二叉树是否为完全二叉树

```
#include "tree.h"
#include "queue.h"
/**
 * 完全二叉树：除了最有一层右边部分外，其余层都是满的
 * 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
 */
bool IsCompleteBiTree(BiTree tree)
{
#define BiTreeWidthMax  4096
    if (NULL == tree) return true;
    SeqQueue queue;
    InitQueue(&queue, BiTreeWidthMax);
    EnQueue(&queue, tree);

    while (!QueueEmpty(&queue)) {
        BiTree node;
        DeQueue(&queue, &node);
        if (NULL != node) {
            EnQueue(&queue, node->lchild);
            EnQueue(&queue, node->rchild);
        } else {
            while (!QueueEmpty(&queue)) {
                DeQueue(&queue, &node);
                if (NULL != node) {
```

```
                    DestroyQueue(&queue);
                    return false;
                }
            }
            DestroyQueue(&queue);
            return true;
        }
    }

    DestroyQueue(&queue);
    return true;
}
```

7. 判断二叉树 subtree 是否为 tree 的一棵子树

```
#include "tree.h"
bool IsEqualBiTree(BiTree tree1, BiTree tree2)
{
    if (NULL == tree1 && NULL == tree2)
        return true;
    if ((NULL != tree1 && NULL != tree2) && (tree1->data == tree2->data)) {
        return IsEqualBiTree(tree1->lchild, tree2->lchild)
            && IsEqualBiTree(tree1->rchild, tree2->rchild);
    }
    return false;
}

bool IsSubTree(BiTree tree, BiTree subtree)
{
    if (IsEqualBiTree(tree, subtree))
        return true;
    if (NULL != tree) {
        if (IsSubTree(tree->lchild, subtree))
            return true;
        if (IsSubTree(tree->rchild, subtree))
            return true;
    }
    return false;
}
```

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。

```
#include "tree.h"
bool IsPartOf(BiTree tree, BiTree part)
{
    if (NULL == part)
        return true;
    if (NULL == tree)    /* part 一定不等于 NULL */
        return false;
    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 BiTreeNodesMax  10240
8       if (NULL == tree) return;
9       SeqStack stack;
10      SeqQueue queue;
11      InitQueue(&queue, BiTreeNodesMax);
12      InitStack(&stack, BiTreeNodesMax);
13
14      EnQueue(&queue, tree);
15      while (!QueueEmpty(&tree)) {
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
32      DestroyStack(&stack);
33  }
```

10. 通过先序序列和中序序列重构二叉树 *

```c
#include "tree.h"
/**
 * 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，
 * 同时根据 inArr 数组来确定某个节点有没有左右子树，
 * 进而决定 preArr 余下的的节点在这个节点左子树还是右子树
 */
BiTree _ReBuildBiTree(int *preArr, int preIdx, int len,
        int *inArr, int inIdx, int inEnd)
{
    if (preIdx >= len || inIdx >= inEnd)
        return NULL;

    int rootData = preArr[preIdx];
    BiTree root = malloc(sizeof(BiTreeNode));
    if (NULL == root) return NULL;
    root->data = rootData;

    /* 在中序数组 inArr 中找到 rootData，然后划分成两部分 */
    int i;
    for (i = inIdx; i < inEnd; i++) {
        if (rootData == inArr[i])
            break;
    }
    /* 数据有误，在中序数组中没能找到先序数组中的元素 */
    if (i == inEnd) {
        printf("Invalid input data in preArr or inArr.\n");
        return NULL;
    }

    int lpreIdx = preIdx+1;
    int rpreIdx = preIdx+1 + (i-inIdx);
    root->lchild = _ReBuildBiTree(preArr, lpreIdx, len, inArr, inIdx, i);
    root->rchild = _ReBuildBiTree(preArr, rpreIdx, len, inArr, i+1, inEnd);

    return root;
}
BiTree ReBuildBiTree(int *preArr, int *inArr, int len)
{
    if (len <= 0 || NULL == preArr || NULL == inArr) return NULL;
    return _ReBuildBiTree(preArr, 0, len, inArr, 0, len);
}
```

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

```c
#include "tree.h"
/* 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 */
void postOrder(int *preArr, int currNodeIdx, int len, int *postArr, int *idx)
{
    if (currNodeIdx >= len) return;
```

```
    postOrder(preArr, 2*currNodeIdx+1, len, postArr, idx);
    postOrder(preArr, 2*currNodeIdx+2, len, postArr, idx);
    postArr[*idx] = preArr[currNodeIdx];
    (*idx) += 1;
}
void FromPreOrderToPostOrder(int *preArr, int *postArr, int len)
{
    if (len <= 0 || NULL == preArr || NULL == postArr) return;
    int idx = 0;
    postOrder(preArr, 0, len, postArr, &idx);
}
```

12. 统计二叉树的节点数、叶子数、双分叉节点数

```
#include "tree.h"
int CountNodes(BiTree tree)
{
    if (NULL == tree) return 0;
    return (1 + CountNodes(tree->lchild) + CountNodes(tree->rchild));
}

int CountLeafs(BiTree tree)
{
    if (NULL == tree) return 0;
    if (NULL == tree->lchild && NULL == tree->rchild)
        return 1;
    return CountLeafs(tree->lchild) + CountLeafs(tree->rchild);
}

int CountForks(BiTree tree)
{
    int cnt = 0;
    if (NULL == tree) return 0;
    if (NULL != tree->lchild && NULL != tree->rchild)
        cnt = 1;
    return (cnt + CountForks(tree->lchild) + CountForks(tree->rchild));
}
```

13. 交换左右子树

```
#include "tree.h"
void SwapBiTree(BiTree tree)
{
    if (NULL == tree) return;
    BiTree t = tree->lchild;
    tree->lchild = tree->rchild;
    tree->rchild = t;
    SwapBiTree(tree->lchild);
    SwapBiTree(tree->rchild);
}
```

14. 删除二叉树所有值为 x 的子树（不是节点）

```
#include "tree.h"
void DestroyBiTree(BiTree tree)
{
    if (NULL == tree) return;
    DestroyBiTree(tree->lchild);
    DestroyBiTree(tree->rchild);
    free(tree);
}
void DeleteAllSubtreeXFromBiTree(BiTree *ptree, int x)
{
    if (NULL == ptree || NULL == *ptree) return;
    if (x == (*ptree)->data) {
        DestroyBiTree(*ptree);
        *ptree = NULL;
        return;
    }
    DeleteAllSubtreeXFromBiTree(&(*ptree)->lchild, x);
    DeleteAllSubtreeXFromBiTree(&(*ptree)->rchild, x);
}
```

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

```
#include "tree.h"
/**
 * 从二叉树中取下某一个节点，存储到 store 指向的空间里
 */
bool StealNodeFromBiTree(BiTree *ptree, BiTree *store)
{
    /* 空树，没节点可取 */
    if (NULL == *ptree) return false;
    if (NULL != (*ptree)->lchild)
        return StealNodeFromBiTree(&(*ptree)->lchild, store);
    if (NULL != (*ptree)->rchild)
        return StealNodeFromBiTree(&(*ptree)->rchild, store);
    /* 只有根节点 */
    *store = *ptree;
    *ptree = NULL;
    return true;
}
void DeleteAllXFromBiTree(BiTree *ptree, int x)
{
    if (NULL == ptree || NULL == *ptree) return;

    DeleteAllXFromBiTree(&(*ptree)->lchild, x);
    DeleteAllXFromBiTree(&(*ptree)->rchild, x);

    if (x == (*ptree)->data) {
        BiTree lsubtree = (*ptree)->lchild;
        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。

```c
#include "tree.h"
ThreadTree PrevNodeOfPostOrderInThreadTree(ThreadTree tree, ThreadTree node)
{
    if (NULL == tree || NULL == node) return NULL;
    if (TagChild == node->rtag)
        return node->rchild;
    if (TagChild == node->ltag)
        return node->lchild;
    while (NULL != node->lchild) {
        node = node->lchild;
        if (TagChild == node->ltag)
            return node->lchild;
    }
    return NULL;
}
```

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

```c
#include "tree.h"
int _WPLOfBiTree(BiTree tree, int deepth)
{
    if (NULL == tree) return 0;
    if (NULL == tree->lchild && NULL == tree->rchild)
        return tree->data * (deepth+1);
    return _WPLOfBiTree(tree->lchild, deepth+1)
         + _WPLOfBiTree(tree->rchild, deepth+1);
}
int WPLOfBiTree(BiTree tree)
{
    return _WPLOfBiTree(tree, 0);
}
```

21. 输出给定表达树的中缀表达式子（通过括号体现优先级）

```c
#include "tree.h"
/**
 * 表达树：叶子节点为数据，非叶子节点为运算符号
 * 实际上就是中序遍历
 */
void PrintExpressionBiTree(BiTree tree)
{
    if (NULL == tree) return;

    /* 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

```c
#include "tree.h"
int MaxValue(BiTree tree)
{
    /* 返回 INT_MIN 表示没有最大值，INT_MAX 来自头文件 limits.h */
    if (NULL == tree)
        return INT_MIN;

    while (NULL != tree->rchild)
        tree = tree->rchild;
    return tree->data;
}
bool MinValue(BiTree tree)
{
    if (NULL == tree) return INT_MAX;

    while (NULL != tree->lchild)
        tree = tree->lchild;
    return tree->data;
}
```

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

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

- 版本二：仅仅遍历要输出的节点

```c
#include "tree.h"
void PrintFromHiToLoInBSTTree(BiTree tree, int lo, int hi)
{
    if (NULL == tree || lo > hi) return;

    if (tree->data <= hi)
        PrintFromHiToLoInBSTTree(tree->rchild, lo, hi);

    printf("%d ", tree->data);

    if (tree->data >= lo)
        PrintFromHiToLoInBSTTree(tree->lchild, lo, hi);
}
```

27. 以 $O(\log_2(n))$ 时间复杂度查找二叉树（假定二叉树树没有退化成链表）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

```
#include "tree.h"
BiTree FindKthNodeInBiTree(BiTree tree, int k)
{
    if (NULL == tree || k <= 0) return NULL;
    int lsubtreeNodes = 0;
    if (NULL != tree->lchild)
        lsubtreeNodes = tree->nodes;
    if (lsubtreeNodes + 1 == k)
        return tree;
    if (k <= lsubtreeNodes)
        return FindKthNodeInBiTree(tree->lchild, k);
    else
        return FindKthNodeInBiTree(tree->rchild, k - lsubtreeNodes - 1);
}
```

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[4]。

```
/* General List Node */
typedef struct GListNode {
    int data;
    struct GListNode *left, *right;
    bool isAtom;
} GListNode;
/* General List */
typedef GListNode *GList;
```

基本操作

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

```
bool IsAtom(GList list)
{
```

[4]在 C 语言中更常用联合体 union 来表示这种结构

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

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

```
bool IsPair(GList list)
{
    /* 空列表不是节点对 */
    if (NULL == list) return false;
    return !IsAtom(list);
}
```

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

```
bool IsNull(GList list)
{
    return (NULL == list);
}
```

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

```
GList MakeAtom(int data)
{
    GList node = malloc(sizeof(GListNode));
    if (NULL == node) return NULL;
    node->data = data;
    node->isAtom = true;
    return node;
}
```

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 [5] [6]。

```
GList MakePair(GList a, GList b)
{
    GList node = malloc(sizeof(GListNode));
    if (NULL == node) return NULL;
    node->left = a;
    node->right = b;
    node->isAtom = false;
    return node;
}
```

[5] https://en.wikipedia.org/wiki/Lisp_(programming_language)#Conses_and_lists

[6] 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

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

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

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

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

```
GList GetTail(GList list)
{
    if (IsNull(list)) return NULL;  /* 空表表尾还是空表 */
    if (IsAtom(list)) return NULL;
    return list->right;
}
```

9. 判断是不是列表 bool IsList(GList list);

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

5.2 例题

1. 构建广义列表 list = (A, (G, H, (M)), D)

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

```
GList listGHM = MakePair(MakeAtom('G'), listHM);/* (G, H, (M)) */
GList listD = MakePair(MakeAtom('D'), NULL);    /* (D) */
GList listGHMD = MakePair(listGHM, listD);      /* ((G, H, (M)), D) */
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

```
GetTail(list);                                          /* (B, (F, C), D) */
GetTail(GetTail(list));                                 /* ((F, C), D) */
GetHead(GetTail(GetTail(list)));                        /* (F, C) */
GetTail(GetHead(GetTail(GetTail(list))));               /* (C) */
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

```
int DepthOfGList(GList list)
{
    if (IsNull(list) || IsAtom(list)) return 0;
    int leftDepth = DepthOfGList(GetHead(list));
    int rightDepth = DepthOfGList(GetTail(list));
    return max(1 + leftDepth, rightDepth);
}
```

4. 求广义表的长度，比如本章第 1 题表长度为 3

```
int LengthOfGList(GList list)
{
    if (!IsList(list)) return 0;
    int len = 0;
    while (!IsNull(list)) {
        len ++;
        list = GetTail(list);
    }
    return len;
}
```

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

```c
void InsertSort(int *arr, int len)
{
    if (NULL == arr || len <= 0) return;
    for (int i = 1; i < len; i ++) {
        /* 把 arr[i] 存储到 tmp，留出 arr[i] 空位 */
        int tmp = arr[i];
        int j;
        /* 把比 tmp 大的元素往后移，给 tmp 留出空位 */
        for (j = i; j-1 >= 0 && tmp < arr[j-1]; j -= 1) {
            arr[j] = arr[j-1];
        /* 插入 tmp 元素到空位 */
        arr[j] = tmp;
    }
}
```

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

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

```
                arr[j] = arr[j-inc];
            arr[j] = tmp;
        }
    }
}
```

- 执行指定增量序列的希尔排序

```
/**
 * 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
 * incLen: 增量数组的大小
 */
void ShellSortWithInc(int *arr, int len, int *inc, int incLen)
{
    if (NULL == arr || len <= 0 || NULL == inc || incLen <= 0) return;
    /* 最后一个增量必须是 1 */
    if (1 != inc[incLen-1])
        return;
    /* 控制增量 */
    for (int k = 0; k < incLen; k++) {
        for (int i = inc[k]; i < len; i ++) {
            int tmp = arr[i];
            int j;
            for (j = i; j-inc[k] >= 0 && tmp < arr[j-inc[k]]; j -= inc[k])
                arr[j] = arr[j-inc[k]];
            arr[j] = tmp;
        }
    }
}
```

7.1.4 冒泡排序

```
void Swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
void BubbleSort(int *arr, int len)
{
    if (NULL == arr || len <= 0) return;
    for (int i = 0; i < len; i++) {
        for (int j = len-1; j-1 >= i; j--) {
            if (arr[j-1] > arr[j]) {
                /* &arr[j-1] 等价于 &(arr[j-1]) 表示求元素 arr[j-1] 的地址 */
                Swap(&arr[j-1], &arr[j]);
            }
        }
    }
}
```

7.1.5 快速排序

```
int Partition(int *arr, int lo, int hi)
{
    int pivot = arr[lo];
    while (lo < hi) {
        while (lo < hi && arr[hi] >= pivot)
            -- hi;
        arr[lo] = arr[hi];
        while (lo < hi && arr[lo] <= pivot)
            ++ lo;
        arr[hi] = arr[lo];
    }
    arr[lo] = pivot;
    return lo;
}
void _QuickSort(int *arr, int lo, int hi)
{
    if (lo >= hi) return;
    int pivotIdx = Partition(arr, lo, hi);
    _QuickSort(arr, lo, pivotIdx-1);
    _QuickSort(arr, pivotIdx+1, hi);
}
void QuickSort(int *arr, int len)
{
    if (NULL == arr || len <= 0) return;
    _QuickSort(arr, 0, len-1);
}
```

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。

```
int Partition(int *arr, int lo, int hi)
{
    int pivot = arr[lo];
    while (lo < hi) {
        while (lo < hi && arr[hi] >= pivot)
            hi --;
        arr[lo] = arr[hi];
        while (lo < hi && arr[lo] <= pivot)
            lo ++;
        arr[hi] = arr[lo];
    }
    arr[lo] = pivot;
    return lo;
}
int _FindKthElement(int *arr, int lo, int hi, int k)
{
    int pivot_idx = Partition(arr, lo, hi);
```

```
    /* 程序一定是在这里结束递归 */
    if (pivot_idx == k) return arr[pivot_idx];
    if (k > pivot_idx)
        return _FindKthElement(arr, pivot_idx+1, hi, k);
    else
        return _FindKthElement(arr, lo, pivot_idx-1, k);
}
int FindKthElement(int *arr, int len, int k)
{
    if (NULL == arr || len <= 0) return -1;
    if (k <= 0 || k > len) return -1;
    return _FindKthElement(arr, 0, len-1, k-1);
}
```

7.1.6 选择排序

```
void SelectSort(int *arr, int len)
{
    if (NULL == arr || len <= 0) return;
    for (int i = 0; i < len; i++) {
        int minIdx = i;
        for (int j = i; j < len; j++) {
            if (arr[j] < arr[minIdx])
                minIdx = j;
        }
        Swap(&arr[i], &arr[minIdx]);
    }
}
```

7.1.7 堆排序 *

```
void BuildMaxHeap(int *arr, int len)
{
    for (int k = (len-1)/2; k >= 0; k--)
        AdjustHeap(arr, k, len);
}
/* 调整下标为 k 的元素 */
void AdjustHeap(int *arr, int len, int k)
{
    int t = arr[k];
    int i = k;
    while (true) {
        i = 2*i + 1;
        if (i >= len)       /* 调整到叶子节点了 */
            break;
        if (i+1 < len && arr[i] < arr[i+1])
            i++;
        if (t >= arr[i])    /* 余下的已经调整好了 */
            break;
        arr[k] = arr[i];    /* k 是父节点, i 是当前节点 */
```

```
        k = i;
    }
    arr[k] = t;
}
/**
 * 元素下标从 0 开始的
 */
void HeapSort(int *arr, int len)
{
    if (NULL == arr || len <= 0) return;
    BuildMaxHeap(arr, len);
    for (int i = len-1; i > 0; i--) {
        Swap(&arr[i], &arr[0]);
        AdjustHeap(arr, 0, i-1);
    }
}
```

7.1.8 归并排序

1. 递归版本实现，也是最常见的实现方式

```
void CopyArray(int *arr, int lo, int hi, int *tmpArr)
{
    for (int i = 0; i <= hi-lo; i++)
        arr[lo+i] = tmpArr[i];
}
/**
 * 版本一：递归实现的归并排序
 */
void _MergeSort(int *arr, int lo, int hi, int *tmpArr)
{
    if (lo >= hi) return;
    int mid = lo+(hi-lo)/2;
    int lo1 = lo;
    int lo2 = mid+1;
    _MergeSort(arr, lo1, mid, tmpArr);
    _MergeSort(arr, lo2,  hi, tmpArr);
    /* 合并 */
    int idx = 0;
    while (lo1 <= mid && lo2 <= hi) {
        if (arr[lo1] < arr[lo2]) {
            tmpArr[idx] = arr[lo1];
            lo1 ++;
        } else {
            tmpArr[idx] = arr[lo2];
            lo2 ++;
        }
        idx ++;
    }
    while (lo1 <= mid) {
        tmpArr[idx] = arr[lo1];
        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  }
```