数据结构

数据结构分类

数据结构主要就是研究数据存储的方式。

线性表

一对一对一

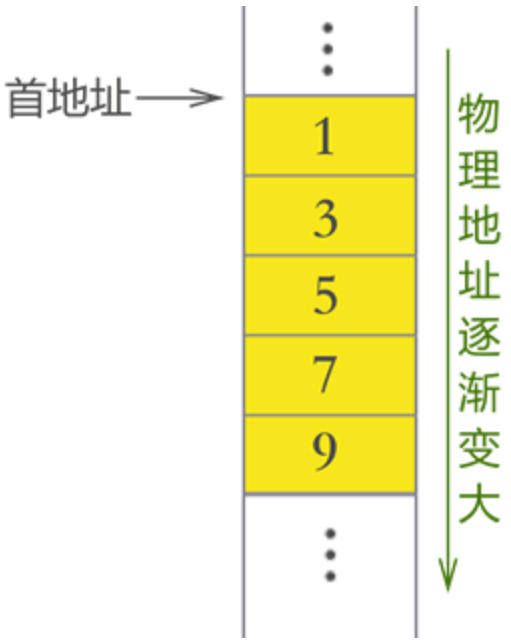
数组`int a[5]={1,2,3,4,5}`，各个元素依次排列，除了首尾元素除了首元素和尾元素，其他的元素前面和后面有且仅有一个元素与之相邻。类似于这样的就可以使用线性表的存储形式。

顺序存储结构：顺序表

链式存储结构：链表

顺序表

类似于数组。



> 顺序表结构的底层实现借助的就是数组，因此对于初学者来说，可以把顺序表完全等价于数组。但是要做到心中有数：数据结构研究的是数据存储方式，它囊括的都是各种存储结构，而数组仅仅只是编程语言中的一种数据类型，并不属于数据结构的范畴。

链表

使用顺序表（底层实现靠数组）时，需要提前申请一定大小的存储空间，这块存储空间的物理地址是连续的；
链表则完全不同，使用链表存储数据时，是随用随申请，因此数据的存储位置是相互分离的，换句话说，数据的存储位置是随机的；
为了给各个数据块建立 "依次排列" 的关系，链表给各数据块增设一个指针，每个数据块的指针都指向下一个数据块（最后一个数据块的指针指向NULL）就如同一个个小学生都伸手去拉住下一个小学生的手，这样，看似毫无关系的数据块就建立了 "依次排列" 的关系，也就形成了链表；

树

树结构适合存储“一对多”关系的数据

图

图结构适合存储“多对多”关系的数据

数据储存方式

逻辑结构

线性表 用于存储具有 "一对一" 逻辑关系的数据；

树结构 用于存储具有 "一对多" 逻辑关系的数据；

图结构 用于存储具有 "多对多" 逻辑关系的数据。

物理结构

如果选择集中存储，就使用顺序存储结构；如果选择的是分散存储，就使用链式存储结构。至于如何选择，主要取决于存储设备的状态以及数据的用途。

时间、空间复杂度

程序= 算法+数据结构

时间复杂度

时间复杂度的定义：在计算机科学中，算法的时间复杂度是一个函数(数学中带有未知表达式的函数)

它定量地描述了该算法的运行时间，一个算法的执行所耗费的时间，从理论上说 是不能算出来的。

所以为了解决这个麻烦，就有了时间复杂度的分析方式:**一个算法所花费的时间跟其中语句的执行次数成正比。**

算法的基本操作的执行次数为算法的时间复杂度。

大O渐进表示法

规则：

1. 用常数1表示运行过程中所有的常数
2. 如果阶项存在，在修改后的运行次数函数中，去掉所有的加法常数式子，只保留最高阶项，去除与这个项目相乘的常数 。

```
1  void fun(int n)
2  {
3      int x = 0;
4      for(int i = 0;i<2*n;i++)
5      {
6          x++;
7      }
8  }
9  //该算法的执行次数为2*n次
```

```c
void fun(int n)
{
    int x = 0;
    for(int i = 0;i<2;i++)
    {
        x++;
    }
    for(int i = 0;i<2;i++)
    {
        x++;
    }
    for(int i = 0;i<2*n;i++)
    {
        x++;
    }
}
//该算法的执行次数为4*n次
```

常数阶

```c
void fun(int n)
{
    int x = 0;
    for(int i = 0;i<2;i++)
    {
        x++;
    }
    for(int i = 0;i<2;i++)
    {
        x++;
    }
    for(int i = 0;i<2;i++)
    {
        x++;
    }
}
//该算法的执行次数为2+2+2次
//O(1)
```

线性阶

```
void fun(int n)
{
    int x = 0;
    for(int i = 0;i<2;i++)
    {
        x++;
    }
    for(int i = 0;i<2;i++)
    {
        x++;
    }
    for(int i = 0;i<2*n;i++)
    {
        x++;
    }
}
//O(N)
```

平方阶，立方阶

指数阶

```
long fun(int n)
{
    return n<2?n:fun(n-1)+fun(n-2);
}

//O(2^N)
```

对数阶

`O(1)` 常数阶 < `O(logn)` 对数阶 < `O(n)` 线性阶 < `O(n^2)` 平方阶 < `O(n^3)` 立方阶 < `O(2^n)` 指数阶

空间复杂度

空间复杂度就是一个算法在运行过程中临时占用空间大小的量度。

对算法的空间复杂度影响最大的，往往是程序运行过程中所申请的临时存储空间。不同的算法所编写出的程序，其运行时申请的临时存储空间通常会有较大不同。一般与内部的临时变量有关。

线性表--顺序表（顺序存储结构）

基本概念

顺序存储结构定义：
将具有与“一对一”逻辑关系的数据按照次序连续的存储到一整块内存空间上。

顺序表的初始化

使用顺序表存储数据之前，除了要申请一整块足够大小的物理空间之外；为了方便后期使用表中的数据，还需要实时记录以下 2 项数据：
1顺序表申请的存储容量；
2顺序表的长度，也就是表中存储数据元素的个数；
正常来说，顺序表申请的存储容量要大于顺序表的长度。

定义顺序表（静态分配）：

```
//#include "list.h"
#include <stdio.h>
#define Size 10
typedef struct List
{
    int data[Size];
    int len;    //顺序表当前的长度
}Slist;

void InitList();
int main()
{
    Slist l;
    InitList(&l);
    return 0;
}

void InitList(Slist *l)
{
    for(int i = 0;i<Size;i++)
    {
        l->data[i] = 0;
    }
    l->len = 0;
}
```

定义顺序表（动态分配）：

```c
#include <stdio.h>
#include <stdlib.h>

#define MAXSize 10   //默认最大长度
typedef struct List
{
    int *head;   //指向动态开辟的空间（数组）
    int MaxSize; //数组的最大容量
    int len;    //顺序表当前的长度
}Slist;

Slist InitList();        //顺序表的初始化
void display(Slist l);   //遍历顺序表中的元素
int main()
{
    Slist L;
    L = InitList();
    //向顺序表中添加元素
    for(int i = 0;i < MAXSize;i++)
    {
        L.head[i] = i;
        L.len++;
    }

    display(L);
    printf("%d\n",L.len);
    return 0;
}
//创建一个空的顺序表
Slist InitList()
{
    Slist l;

    l.head = (int*)malloc(MAXSize*sizeof(int));
    if(NULL == l.head)
    {
        printf("error");
        exit(1);     //退出
    }
    l.len = 0;
    l.MaxSize = MAXSize;

    return l;
}

void display(Slist l)
{
```

```
    for(int i =0;i<l.len;i++)
    {
        printf("%d",l.head[i]);
    }
    puts(" ");
}
```

```
0123456789
10
```

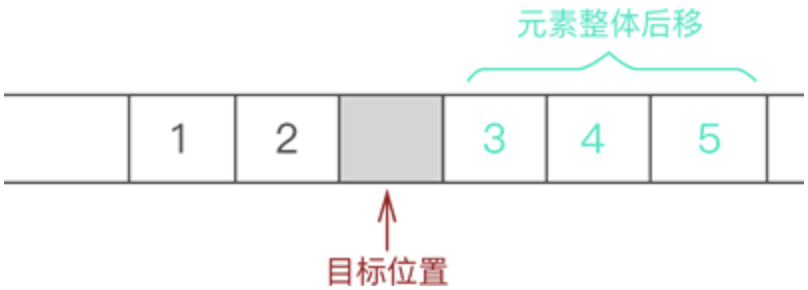
顺序表基本操作

插入元素

例如，在顺序表 {1,2,3,4,5} 的第 3 个位置上插入元素 6，实现过程如下：

●遍历至顺序表存储第 3 个数据元素的位置，如下图所示：

●将元素 3 以及后续元素 4 和 5 整体向后移动一个位置，如下图所示：



●将新元素 6 放入腾出的位置，如下图所示：

```c
#include <stdio.h>
#include <stdlib.h>

#define MAXSize 10   //默认最大长度
typedef struct List
{
    int *head;   //指向动态开辟的空间（数组）
    int MaxSize; //数组的最大容量
    int len;    //顺序表当前的长度
}Slist;

Slist InitList();        //顺序表的初始化
void display(Slist l);   //遍历顺序表中的元素
Slist InsertList(Slist l,int addr,int elem);   // addr表示插入的位置，elem表示插入的元素的值
int main()
{
    Slist L;
    L = InitList();
    //向顺序表中添加元素
    for(int i = 0;i < MAXSize-1;i++)
    {
        L.head[i] = i;
        L.len++;
    }
    L =InsertList(L,2,9);
    display(L);
    printf("%d\n",L.len);
    return 0;
}
//创建一个空的顺序表
Slist InitList()
{
    Slist l;

    l.head = (int*)malloc(MAXSize*sizeof(int));
    if(NULL == l.head)
    {
        printf("error");
        exit(1);     //退出
    }
    l.len = 0;
    l.MaxSize = MAXSize;

    return l;
}
//遍历顺序表中的元素
```

```c
void display(Slist l)
{
    for(int i =0;i<l.len;i++)
    {
        printf("%d",l.head[i]);
    }
    puts(" ");
}
//插入元素
Slist InsertList(Slist l,int addr,int elem)
{
    //判断插入的位置是否合理
    if(addr > l.len || addr < 0)
    {
        printf("addr error");
        exit(1);
    }
    //判断容量是否足够
    if(l.len == l.MaxSize)
    {
        l.head = (int *)realloc(l.head,(l.MaxSize+1)*sizeof(int));
        if(NULL == l.head)
        {
            printf("error");
            exit(1);     //退出
        }
    l.MaxSize++;
    }
    //元素插入
    for(int i = l.len-1;i>=addr;i--)
    {
        l.head[i+1] = l.head[i];
    }
    l.head[addr] = elem;
    l.len++;
    return l;
}
```

```
0192345678
10
```

删除元素

```c
#include <stdio.h>
#include <stdlib.h>

#define MAXSize 10   //默认最大长度
typedef struct List
{
    int *head;   //指向动态开辟的空间（数组）
    int MaxSize; //数组的最大容量
    int len;    //顺序表当前的长度
}Slist;

Slist InitList();        //顺序表的初始化
void display(Slist l);   //遍历顺序表中的元素
Slist delList(Slist l,int addr);   //addr表示删除那个位置上的元素
int main()
{
    Slist L;
    L = InitList();
    //向顺序表中添加元素
    for(int i = 0;i < MAXSize-1;i++)
    {
        L.head[i] = i;
        L.len++;
    }
    L = delList(L,4);
    display(L);
    //printf("%d\n",L.len);
    return 0;
}
//创建一个空的顺序表
Slist InitList()
{
    Slist l;

    l.head = (int*)malloc(MAXSize*sizeof(int));
    if(NULL == l.head)
    {
        printf("error");
        exit(1);     //退出
    }
    l.len = 0;
    l.MaxSize = MAXSize;

    return l;
}
//遍历顺序表中的元素
void display(Slist l)
```

```
{
    for(int i =0;i<l.len;i++)
    {
        printf("%d",l.head[i]);
    }
    puts(" ");
}
Slist delList(Slist l,int addr)
{
    //判断插入的位置是否合理
    if(addr > l.len || addr < 0)
    {
        printf("addr error");
        exit(1);
    }
    //元素删除
    for(int i = addr;i<l.len-1;i++)
    {
        l.head[i] = l.head[i+1];
    }
    l.len--;
    return l;
}
```

```
01235678
```

查找元素

```c
#include <stdio.h>
#include <stdlib.h>

#define MAXSize 10   //默认最大长度
typedef struct List
{
    int *head;   //指向动态开辟的空间（数组）
    int MaxSize; //数组的最大容量
    int len;    //顺序表当前的长度
}Slist;

Slist InitList();        //顺序表的初始化
void display(Slist l);   //遍历顺序表中的元素
int searchList(Slist l,int elem);
int main()
{
    Slist L;
    L = InitList();
    //向顺序表中添加元素
    for(int i = 0;i < MAXSize-1;i++)
    {
        L.head[i] = i;
        L.len++;
    }
    int a =searchList(L,7);
    printf("%d\n",a);
    display(L);
    //printf("%d\n",L.len);
    return 0;
}
//创建一个空的顺序表
Slist InitList()
{
    Slist l;

    l.head = (int*)malloc(MAXSize*sizeof(int));
    if(NULL == l.head)
    {
        printf("error");
        exit(1);     //退出
    }
    l.len = 0;
    l.MaxSize = MAXSize;

    return l;
}
//遍历顺序表中的元素
```

```
void display(Slist l)
{
    for(int i =0;i<l.len;i++)
    {
        printf("%d",l.head[i]);
    }
    puts(" ");
}
//查找顺序表中元素的位置
int searchList(Slist l,int elem)
{
    for(int i = 0;i<l.len;i++)
    {
        if(l.head[i] == elem)
        return i;
    }
    return -1;
}
```

更改元素

```c
#include <stdio.h>
#include <stdlib.h>

#define MAXSize 10   //默认最大长度
typedef struct List
{
    int *head;   //指向动态开辟的空间（数组）
    int MaxSize; //数组的最大容量
    int len;    //顺序表当前的长度
}Slist;

Slist InitList();        //顺序表的初始化
void display(Slist l);   //遍历顺序表中的元素
int searchList(Slist l,int elem);
void updataList(Slist l,int elem,int newelem);
int main()
{
    Slist L;
    L = InitList();
    //向顺序表中添加元素
    for(int i = 0;i < MAXSize-1;i++)
    {
        L.head[i] = i;
        L.len++;
    }
    updataList(L,4,9);
    //printf("%d\n",a);
    display(L);
    //printf("%d\n",L.len);
    return 0;
}
//创建一个空的顺序表
Slist InitList()
{
    Slist l;

    l.head = (int*)malloc(MAXSize*sizeof(int));
    if(NULL == l.head)
    {
        printf("error");
        exit(1);     //退出
    }
    l.len = 0;
    l.MaxSize = MAXSize;

    return l;
}
```

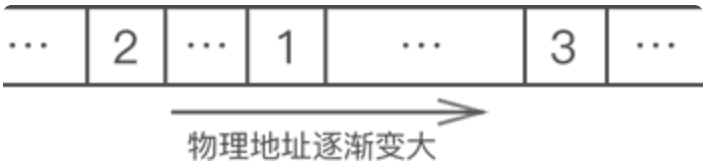
```c
//遍历顺序表中的元素
void display(Slist l)
{
    for(int i =0;i<l.len;i++)
    {
        printf("%d",l.head[i]);
    }
    puts(" ");
}
//查找顺序表中元素的位置
int searchList(Slist l,int elem)
{
    for(int i = 0;i<l.len;i++)
    {
        if(l.head[i] == elem)
        return i;
    }
    return -1;
}

void updataList(Slist l,int elem,int newelem)
{
    int a = searchList(l,elem);
    l.head[a] = newelem;
}
```

线性表--单链表（链式存储结构）

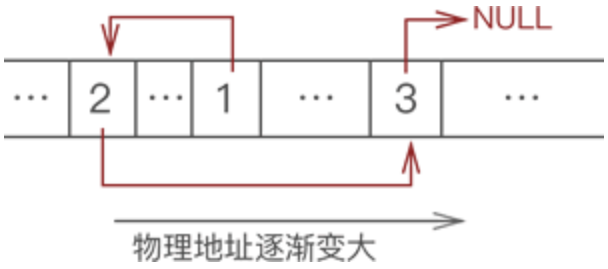
基本概念

链表，又名单链表，链式存储结构。用来存储逻辑关系为一对一的数据。与顺序表的不同点在于，链表不限制数据的物理存储状态。就是说，使用链表存储的数据元素，其物理存储位置是随机的。

使用链表存储 `{1，2，3，4，5}` ，那么他的物理存储状态可以为

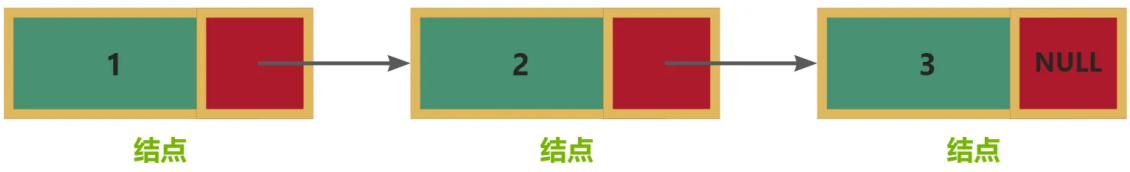


链表的解决方案是：每个元素在存储的时候都配备一个指针，用于指向自己的直接后继(元素)。



链表的数据结构：

将上述结构称之为**节点，**链表实际存储的是一个一个的**节点** `Node` ，真正的数据元素包含在这些**节点** `Node` 中。



链表中的结点

将链表的第一个节点称之为首元节点。

存在一个指向首元节点的指针，叫做头指针。

单链表的初始化

```c
#include <stdio.h>
#include <stdlib.h>
typedef int TypeElem;
typedef struct node
{
    TypeElem data;
    struct node *next;
}NODE;
NODE * Creat_list();
void Printf_list(NODE *first);
int main()
{
    NODE *pfirst = Creat_list();
    Printf_list(pfirst);
    return 0;
}
//创建链表，从键盘上输入，直到输入0结束输入
NODE * Creat_list()
{
    NODE *first = NULL;     //指向链表的首元结点
    NODE *last = NULL;      //指向链表的最后一个结点
    NODE *p;      //指向新分配的结点
    int x ;
    while(1)
    {
        scanf("%d",&x);
        if(x == 0)
        {
            break;
        }
        p = (NODE *)malloc(sizeof(NODE));
        p->data = x;
        p->next = NULL;
        //分情况，将新节点加入到链表中。
        if(first == NULL)  //表示此时链表中还没有节点，这个节点就作为链表的首元节点
        {
            first = p;
            last = p;
        }
        else         //此时链表中有节点
        {
            //尾插法
            last->next = p;
            last = p;
            //头插法
            //p->next = first;
            //first = p;
```

```
        }
    }
    return first;  //返回链表的地址

}

void Printf_list(NODE *first)
{
    if(first == NULL)
    {
        printf("list is empty");
    }

        NODE *p = first;    //p用来遍历链表（循环）
        while(p != NULL)
        {
            printf("%d",p->data);
            //Printf_list(first->next);
            p = p->next;
        }

    puts(" ");
}
```

查找元素

```c
int Find_node(NODE *first, TypeElem x)
{
    NODE *p = first;
    while(p != NULL)
    {
        if(p->data == x)
        {
            printf("%d",p->data);
            return 0;
        }
        else
        {
            p = p->next;
        }

    }
    puts(" ");
    return -1;
}
```

删除元素

1. 释放链表的每一个节点空间

```c
NODE * Free_list(NODE *first)
{
    NODE *p = first;
    while(p != NULL)
    {
        //保存first的下一个
        p = first->next;
        first->data = 0;
        first->next = NULL;
        free(first);
        //再让first指向下一个要释放的节点
        first = p;
    }
    return first;
}
```

2. 删除数据域为X的节点

```
NODE * Del_node(NODE *first,TypeElem x)
{
    if(first == NULL)
    {
        return NULL;
    }
    NODE *p = first;
    NODE *r = NULL;   //用来保存要删除的节点的前一个节点
    while(p != NULL)
    {
        if(p->data == x)
        {
            break;
        }
        r = p;
        p = p->next;
    }
    if(p != NULL)
    {
        if(p == first)     //如果删除的时第一个节点
        {
            first = p->next;
            free(p);
        }
        else
        {
            r->next = p->next;
            free(p);

        }
    }
    return first;
}
```

插入元素

在一个有序的(升序)链表中 插入一个节点 使其仍然有序

```
NODE * yxcr(NODE *first)
{
    TypeElem x;
    printf("请输入要插入的元素：");
    scanf("%d",&x);
    NODE *p = first;
    NODE *r = NULL;
    NODE * y=(NODE *)malloc(sizeof(NODE));
    y->data = x;
    while(p != NULL)
    {
        if(p->data > x)
        {
            break;
        }
        r = p;
        p = p->next;
    }
    if(p == first)
    {
        y->next = p;
        first = y;
    }
    else if(p == NULL)
    {
        y->next = NULL;
        r->next = y;
    }
    else
    {
        y->next = p;
        r->next = y;
    }
    return first;
}
```

线性表--双链表（链式存储结构）

概念：

在一个节点中，即保存它的下一个节点的地址，也保存它的上一个节点的地址。既然如此，这个代表节点的结构体就绪要如下定义：

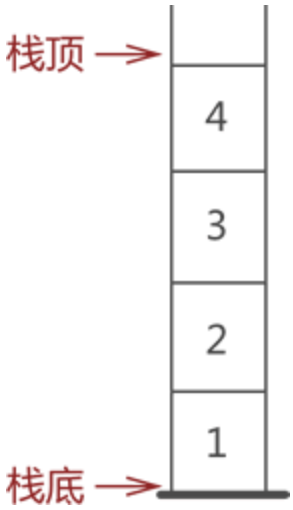
```
typedef struct node
{
    TypeElem data;          //数据域
    struct node *next;      //存储下一个元素的地址
    struct node *prev;      //存储上一个元素的地址
}NODE;吧
```

栈和队列

栈

栈是一种用来存储一对一逻辑的数据结构。

栈，对于存储数据和取出数据有要求：

1. 栈只允许从表的一端存取数据，另一端封闭。
2. 在栈中，无论是存数据还是取数据都要遵循一个原则叫作“**先进后出，后进先出**”，即最先进入栈的元素，最后才可以取出来。
3. 

只允许在栈顶进行数据的储存和读取工作，栈底不可以。

把栈内元素的存取叫做：

出栈：从栈中取出指定的元素

入栈：向栈中添加元素

顺序栈

用一组地址连续的存储单元来依次存放栈里面的每个元素。 即类似"数组"

代码示例：

```c
#ifndef __SEQSTACK_H__
#define __SEQSTACK_H__

#include <stdio.h>
#include <stdlib.h>

typedef int SElemType; //栈中元素的类型
#define STACK_MAX_LENGTH 1024

typedef struct seqstack
{
    //真正的栈空间 通过malloc申请得到
    SElemType * elem ;
    //栈顶的下标
    int top;
    //int max_len ; //elem指向的空间的最大的存放的元素个数

}SeqStack;

//初始化一个栈
SeqStack * InitStack();

//判断一个栈是否为空
//如果栈为空 返回1
//如果栈不为空 返回0
int StackIsEmpty(SeqStack * s);

//获取栈顶元素 但是元素不出栈
//GetTop函数的设计 : 既要考虑返回栈顶元素 还要考虑该函数获取栈顶元素成功与否
//返回值来表示成功与否  返回1表示成功 返回0表示是失败
//额外用一个空间来保存栈顶元素
int GetTop(SeqStack * s,SElemType * e);

//返回值为1 代表出栈成功  返回为0 表示出栈失败
//e指向的空间额外用来保存栈顶元素
int Pop(SeqStack * s,SElemType * e);

//入栈 把一个元素加入到栈中
//成功返回1 失败返回0
int Push(SeqStack * s , SElemType e) ;

#endif
```

```c
#include "sxstack.h"

//初始化一个栈
SeqStack * InitStack()
{
    //为所谓的头结点申请空间并初始化
    SeqStack * s = (SeqStack *)malloc(sizeof(SeqStack));

    s->elem  = (SElemType *)malloc(STACK_MAX_LENGTH * sizeof(SElemType));
    s->top = -1;//top 保存的栈顶元素的下标
                //top = -1时 表示是一个空栈(栈中什么元素都没有)
    return s; //把头节点的地址返回 后续通过头结点来管理这个栈。
}

//判断一个栈是否为空
//如果栈为空 返回1
//如果栈不为空 返回0
int StackIsEmpty(SeqStack * s)
{
    if(s == NULL)
    {
        return 1;
    }
    return s->top == -1 ? 1 : 0 ;
}

//返回栈中元素的个数
int StackLength(SeqStack * s)
{
    if(s ==NULL)
    {
        return 0;
    }
    return s->top + 1;
}

//返回值为1 代表出栈成功  返回为0 表示出栈失败
//e指向的空间额外用来保存栈顶元素
int Pop(SeqStack * s,SElemType * e)
{
    if(s == NULL || s->top == -1 || e == NULL)
    {
        return 0;
    }
    *e = s->elem[s->top];
    s->top--; //出栈操作
}
```

```c
//入栈 把一个元素加入到栈中
//成功返回1 失败返回0
int Push(SeqStack * s , SElemType e)
{
    if(s == NULL || s->top == STACK_MAX_LENGTH - 1)
    {
        return 0;
    }
    s->top++;
    s->elem[s->top] = e;
    return 1;
}
```

```c
#include "sxstack.h"

int main()
{
    SeqStack *s = InitStack();
    int x;
    while(1)
    {
        scanf("%d",&x);
        if(x == 0)
        {
            break;
        }
        PUsh(s,x);
    }
    int n = s->top+1;
    //for(int i = s->top+1;i>0;i--)
    for(int i = 0;i<n;i++)
    {
        Pop(s,&x);
        printf("%d",x);
    }
    puts(" ");
    return 0;
}
```

链式栈

用链表来构建一个栈

代码示例

```c
#include "sxstack.h"

//初始化一个栈
SHead* InitStack()
{
    SHead* s = (SHead*)malloc(sizeof(SHead));
    s->n = 0;
    s->Bottom = s->Top = NULL;
    return s;
}
//清空栈,释放所有的节点
void Clear_Stack(SHead *s)
{
    if(s == NULL || s->n == 0)
    {

    }
    LsStack * p = s->Bottom;
    while(p != NULL)
    {
        s->Bottom = p->next;
        free(p);
        p = s->Bottom;
    }
    s->Top = NULL;
    s->n = 0;
}
//出栈
int Pop(SHead * s,Elemtype *x)
{

    if(s->Top == NULL)
    {
        s->Bottom = NULL;
        return 0;
    }
    else{
        *x = s->Top->data;
        LsStack *p = s->Top;
        s->Top = s->Top->next;
        free(p);
        s->n--;
    }
    return 1;
}
//入栈
int Push(SHead * s,Elemtype x)
```

```
{
    if(s == NULL)
    {
        return 0;
    }
    LsStack * p = (LsStack *)malloc(sizeof(LsStack));
    p->data = x;
    p->next = NULL;
    if(s->Top == NULL)
    {
        s->Bottom = s->Top = p;
    }
    else{
        p->next = s->Top;
        s->Top = p;
    }
    s->n++;
    return 1;
}
```

队列

概念：队列是一种先进先出(FIFO:First In First Out)的线性表

它只允许在表的一端进行插入，在另外一端进行删除操作。

队尾(rear):允许插入元素的一端

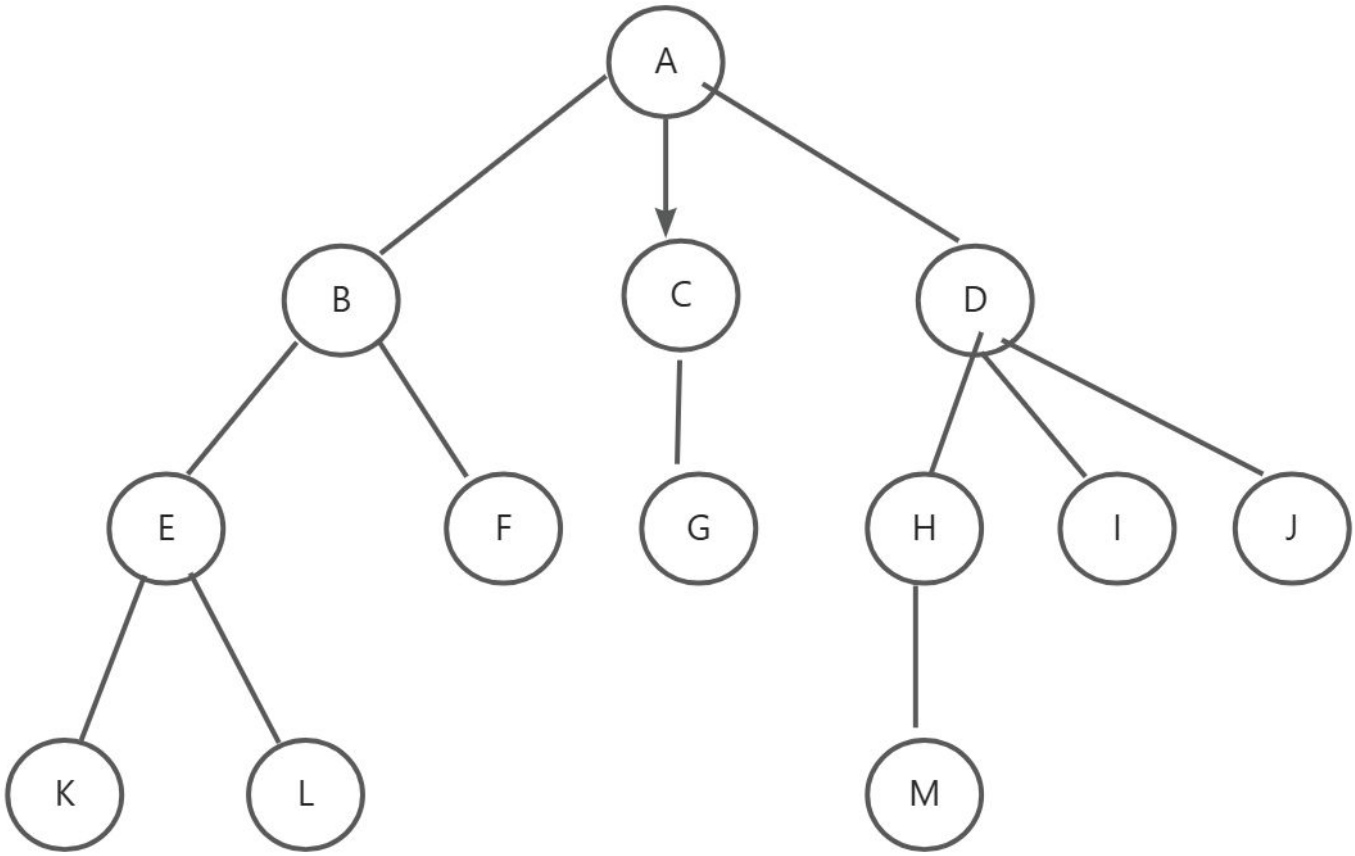
队头(front)；允许删除元素的那一端

“排队思想”：先进先出

“排队论”、

树结构

树也是一种非线性的是数据结构。用来存储具有“一对多”逻辑关系的数据。



树的结点

使用树结构存储的每一个数据元素都被称为 "结点"。

对于（A）中的结点 A、B、C、D 来说，A 是 B、C、D 结点的父结点（也称为 "双亲结点"），而 B、C、D 都是 A 结点的子结点（也称 "孩子结点"）。对于 B、C、D 来说，它们都有相同的父结点，所以它们互为兄弟结点。

树根结点（简称 "根结点"）：每一个非空树都有且只有一个被称为根的结点。

叶子结点：如果结点没有任何子结点，那么此结点称为叶子结点（叶结点）。比如：K,L,F,G。。。

子树和空树

子树：如图 1（A）中，整棵树的根结点为结点 A，而如果单看结点 B、E、F、K、L 组成的部分来说，也是棵树，而且节点 B 为这棵树的根结点。所以称 B、E、F、K、L 这几个结点组成的树为整棵树的子树；同样，结点 E、K、L 构成的也是一棵子树，根结点为 E。

空树：如果集合本身为空，那么构成的树就被称为空树。空树中没有结点。

结点的度和层次

结点的度：对于一个结点，拥有的子树数（结点有多少分支）称为**结点的度** `Degree` 。例如，图（A）中，根结点 A 下分出了 3 个子树，所以，结点 A 的度为 3。

一棵树的度是树内各结点的度的最大值。

结点的层次：从一棵树的树根开始，树根所在层为第一层，根的孩子结点所在的层为第二层，依次类推。

一棵**树的深度（高度）**是**树中结点所在的最大的层次**。上图树的深度为 4

有序树和无序树

如果树中结点的子树从左到右看，谁在左边，谁在右边，是有规定的，这棵树称为有序树；反之称为无序树。

森林

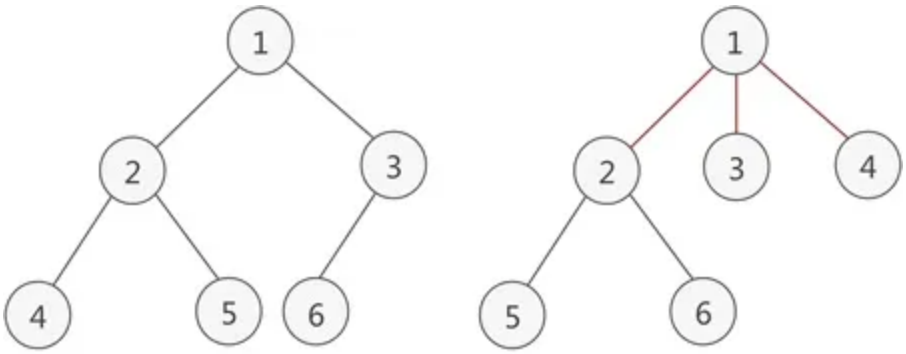
由 m（m >= 0）个互不相交的树组成的集合被称为森林。图（A）中，分别以 B、C、D 为根结点的三棵子树就可以称为森林。

> 树可以理解为由根结点和若干子树构成的，这些子树本身就是一个森林。所以树还可以理解为是由根结点和森林组成的。

二叉树

满足一下两个条件的就可以称之为二叉树：

1. 本身必须是有序树
2. 树中包含的各个结点的度不能超过2，只能是0，1，2



二叉树　　非二叉树

二叉树的性质：

1. 对于一个二叉树，第i层最多有多少个结点？ 2^{i-1}
2. 如果二叉树的深度为K，那么此二叉树最多有 2^K-1 个结点；
3. 二叉树中，终端结点数（叶子结点数）为 n_0 ，度为 2 的结点数为 n_2 ， $n_0 = n_2 + 1$

性质 3 的计算方法：

对于一个二叉树，除了度为 0 的叶子结点和度为 2 的结点，剩下的就是度为 1 的结点（设为 n_1 ）；

如此，**总结点数为：** $\mathbf{n = n_0 + n_1 + n_2}$ 。

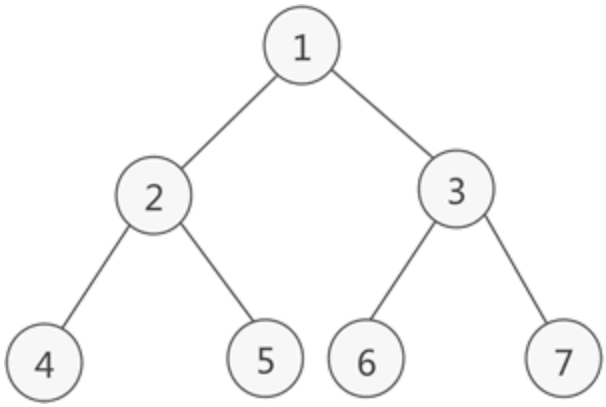
同时，对于每一个结点来说都是由其父结点分支表示的，假设树中分支数为 B，总结点数 $n = B + 1$ 。

分支数也可以通过 n_1 和 n_2 表示，即 $B = n_1 + 2\times n_2$ ，故 $\mathbf{n = n_1 + 2\times n_2 + 1}$ 。

联合红色粗体公式可得 $n_0 = n_2 + 1$

满二叉树

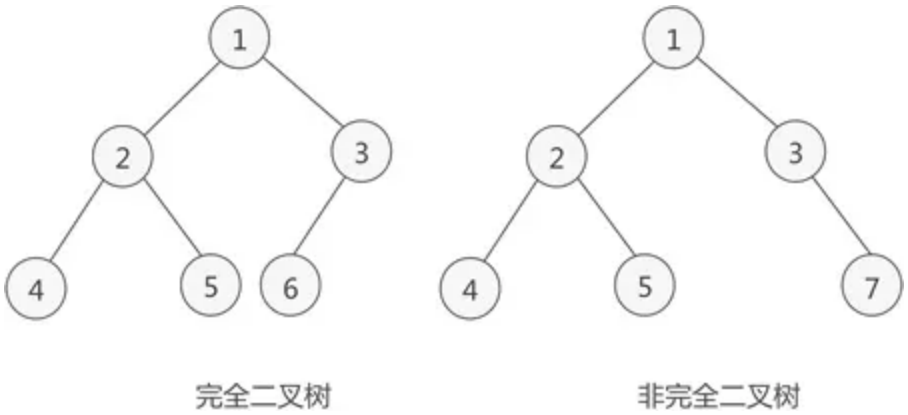
如果二叉树中除了叶子结点，每个结点的度都为 2，则此二叉树称为满二叉树。



1. 满二叉树的第 n 层，结点数为 2^{n-1}
2. 深度为 K 的满二叉树必有 $2^{K}-1$ 个节点，叶子数为 2^{K-1}
3. 满二叉树中不存在度为 1 的结点，每一个分支点中都两棵深度相同的子树，且叶子结点都在最底层。
4. 具有 n 个结点的满二叉树的深度为 $\log_2(n+1)$

完全二叉树

如果二叉树中除去最后一层结点为满二叉树，且最后一层的结点依次从左到右分布，则此二叉树被称为完全二叉树。



上图右侧所示为一颗完全二叉树，由于最后一层的结点没有按照从左向右分布，因此只能算作是普通的二叉树。

完全二叉树除了具有普通二叉树的性质，它自身也具有一些独特的性质。

比如说，n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor+1$ 。

$\lfloor \log_2 n \rfloor$ 表示取小于 $\log_2 n$ 的最大整数。例如，$\lfloor \log_2 4 \rfloor = 2$，而 $\lfloor \log_2 5 \rfloor$ 结果也是 2。

对于任意一个完全二叉树来说，如果将含有的结点按照层次从左到右依次标号，对于任意一个结点 `i`，完全二叉树还有以下几个结论成立：

1. 当 `i>1` 时，父亲结点为结点 `[i/2]` 。（`i=1` 时，表示的是根结点，无父亲结点）
2. 如果 `2*i > n` （`n` 为总结点的个数），则结点 `i` 肯定没有左孩子；否则其左孩子是结点 `2*i` 。
3. 如果 `2*i+1 > n` ，则结点 `i` 肯定没有右孩子；否则右孩子是结点 `2*i + 1` 。

二叉树实现

顺序结构（数组）

二叉树的顺序存储，指的是使用顺序表（数组）存储二叉树。需要注意的是，顺序存储只适用于完全二叉树。换句话说，只有完全二叉树才可以使用顺序表存储。

> 普通二叉树也可以存储，但是需要提前将普通二叉树转化为完全二叉树。

普通二叉树转完全二叉树的方法很简单，只需给二叉树额外添加一些节点，将其 "拼凑" 成完全二叉树即可。

完全二叉树的顺序存储，仅需从根结点开始，按照层次依次将树中结点存储到数组即可。

链式结构

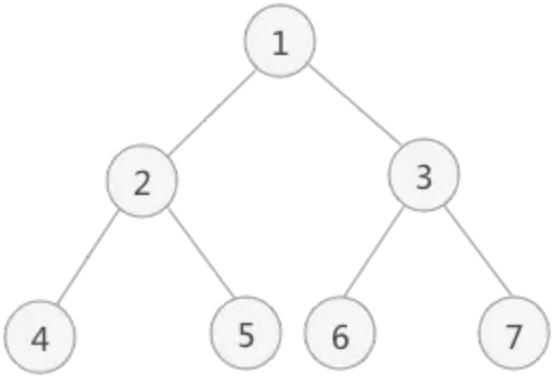
一个数据域用来保存数据

两个指针域，分别保存它的左右两个子节点。

```
typedef int TelemType
typedef struct tnode
{
    TelemType data;
    struct tnode *lchild;
    struct tnode *rchild;
}TNODE;
```

二叉树遍历

先序遍历



二叉树先序遍历的实现思想是：

1. 访问根节点；
2. 访问当前结点的左子树；
3. 若当前结点无左子树，则访问当前结点的右子树；

诀窍：根左右

```
1   1 2 4 5 3 6 7
```

以图 1 为例，采用先序遍历的思想遍历该二叉树的过程为：

1. 访问该二叉树的根结点，找到 1；
2. 访问结点 1 的左子树，找到结点 2；
3. 访问节点 2 的左子树，找到结点 4；
4. 由于访问结点 4 左子树失败，且也没有右子树，因此以结点 4 为根结点的子树遍历完成。但结点 2 还没有遍历其右子树，因此现在开始遍历，即访问结点 5；
5. 由于结点 5 无左右子树，因此结点 5 遍历完成，并且由此以结点 2 为根结点的子树也遍历完成。现在回到结点 1 ，并开始遍历该节点的右子树，即访问结点 3；
6. 访问结点 3 左子树，找到结点 6；
7. 由于结点 6 无左右子树，因此结点 6 遍历完成，回到结点 3 并遍历其右子树，找到结点 7；
8. 节点 7 无左右子树，因此以结点 3 为根结点的子树遍历完成，同时回归结点 1。由于结点 1 的左右子树全部遍历完成，因此整个二叉树遍历完成；

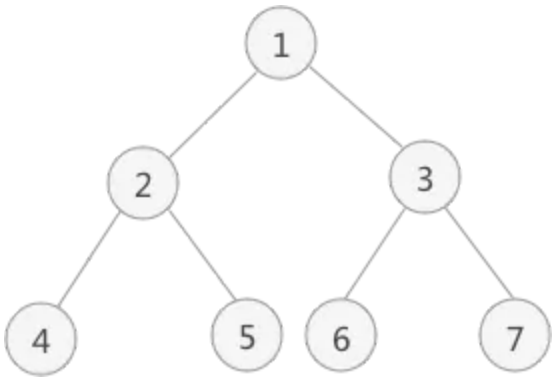
先序遍历的递归实现

```
void xx(TNODE *t)
{
    if(t != NULL)
    {
        printf("%d",t->data);   //根
        xx(t->lchild);          //左
        xx(t->rchild);          //右
    }
}
```

先序遍历的非递归实现

利用栈（请见二叉排序树）

中序遍历



二叉树中序遍历的实现思想是：

1. 访问当前节点的左子树；
2. 访问根节点；
3. 访问当前节点的右子树；

诀窍：左根右

```
4 2 5 1 6 3 7
```

以图 1 为例，采用中序遍历的思想遍历该二叉树的过程为：

1. 访问该二叉树的根节点，找到 1；
2. 遍历节点 1 的左子树，找到节点 2；
3. 遍历节点 2 的左子树，找到节点 4；

4. 由于节点 4 无左孩子，因此找到节点 4，并遍历节点 4 的右子树；
5. 由于节点 4 无右子树，因此节点 2 的左子树遍历完成，访问节点 2；
6. 遍历节点 2 的右子树，找到节点 5；
7. 由于节点 5 无左子树，因此访问节点 5 ，又因为节点 5 没有右子树，因此节点 1 的左子树遍历完成，访问节点 1 ，并遍历节点 1 的右子树，找到节点 3；
8. 遍历节点 3 的左子树，找到节点 6；
9. 由于节点 6 无左子树，因此访问节点 6，又因为该节点无右子树，因此节点 3 的左子树遍历完成，开始访问节点 3 ，并遍历节点 3 的右子树，找到节点 7；
10. 由于节点 7 无左子树，因此访问节点 7，又因为该节点无右子树，因此节点 1 的右子树遍历完成，即整棵树遍历完成；

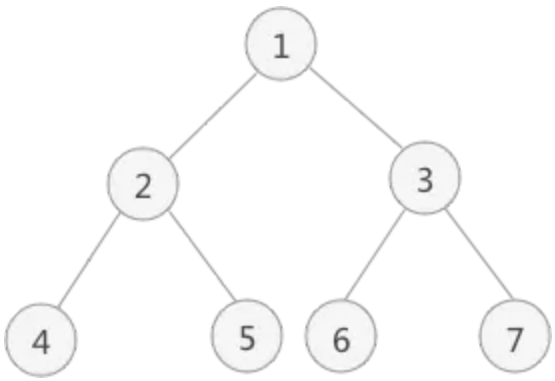
中序遍历的递归实现

```
void zx(TNODE *t)
{
    if(t != NULL)
    {
        xx(t->lchild);          //左
        printf("%d",t->data);   //根
        xx(t->rchild);          //右
    }
}
```

中序遍历的非递归实现

利用栈

后序遍历



二叉树后序遍历的实现思想是：

从根节点出发，依次遍历各节点的左右子树，直到当前结点左右子树遍历完成后，才访问该节点元素。

诀窍：左右根

```
1   4 5 2 6 7 3 1
```

如图 1 中，对此二叉树进行后序遍历的操作过程为：

- 从根节点 1 开始，遍历该节点的左子树（以节点 2 为根节点）；
- 遍历节点 2 的左子树（以节点 4 为根节点）；
- 由于节点 4 既没有左子树，也没有右子树，此时访问该节点中的元素 4，并回退到节点 2 ，遍历节点 2 的右子树（以 5 为根节点）；
- 由于节点 5 无左右子树，因此可以访问节点 5 ，并且此时节点 2 的左右子树也遍历完成，因此也可以访问节点 2；
- 此时回退到节点 1 ，开始遍历节点 1 的右子树（以节点 3 为根节点）；
- 遍历节点 3 的左子树（以节点 6 为根节点）；
- 由于节点 6 无左右子树，因此访问节点 6，并回退到节点 3，开始遍历节点 3 的右子树（以节点 7 为根节点）；
- 由于节点 7 无左右子树，因此访问节点 7，并且节点 3 的左右子树也遍历完成，可以访问节点 3；节点 1 的左右子树也遍历完成，可以访问节点 1；
- 到此，整棵树的遍历结束。

后序遍历的递归实现

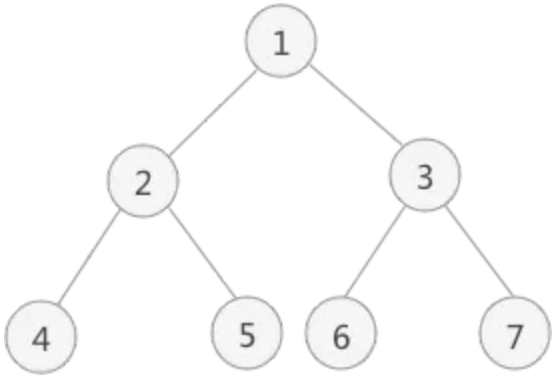
```
void zx(TNODE *t)
{
    if(t != NULL)
    {
        xx(t->lchild);          //左
        xx(t->rchild);          //右
        printf("%d",t->data);   //根
    }
}
```

后序遍历的非递归实现

利用栈

层次遍历

按照二叉树中的层次从左到右依次遍历每层中的结点。




```
1234567
```

二叉树反遍历

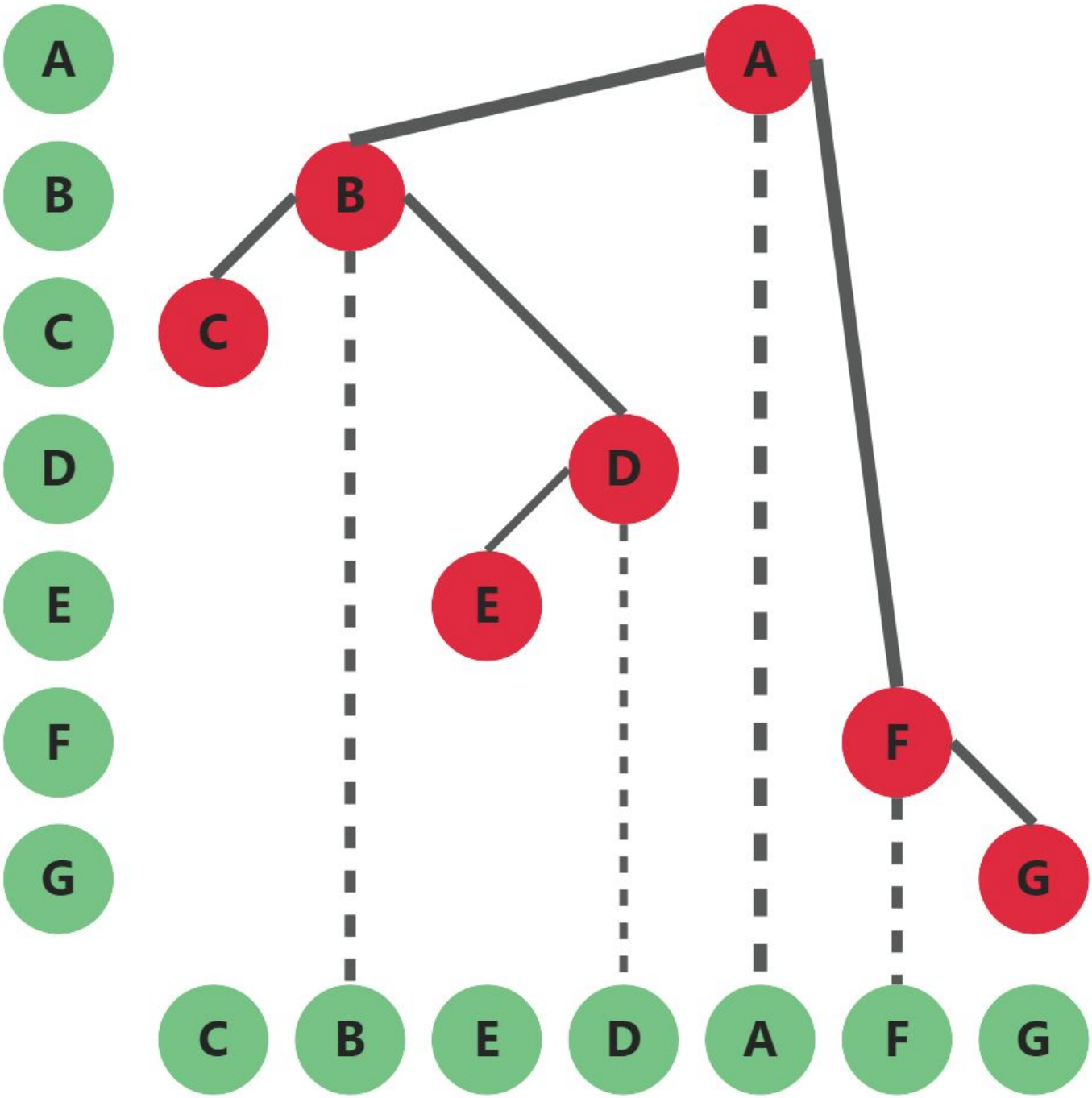
已知先/后序遍历与中序遍历，求后/先序遍历

先：`ABCDEFG`

中：`CBEDAFG`

先：ABCDEFG

中：CBEDAFG

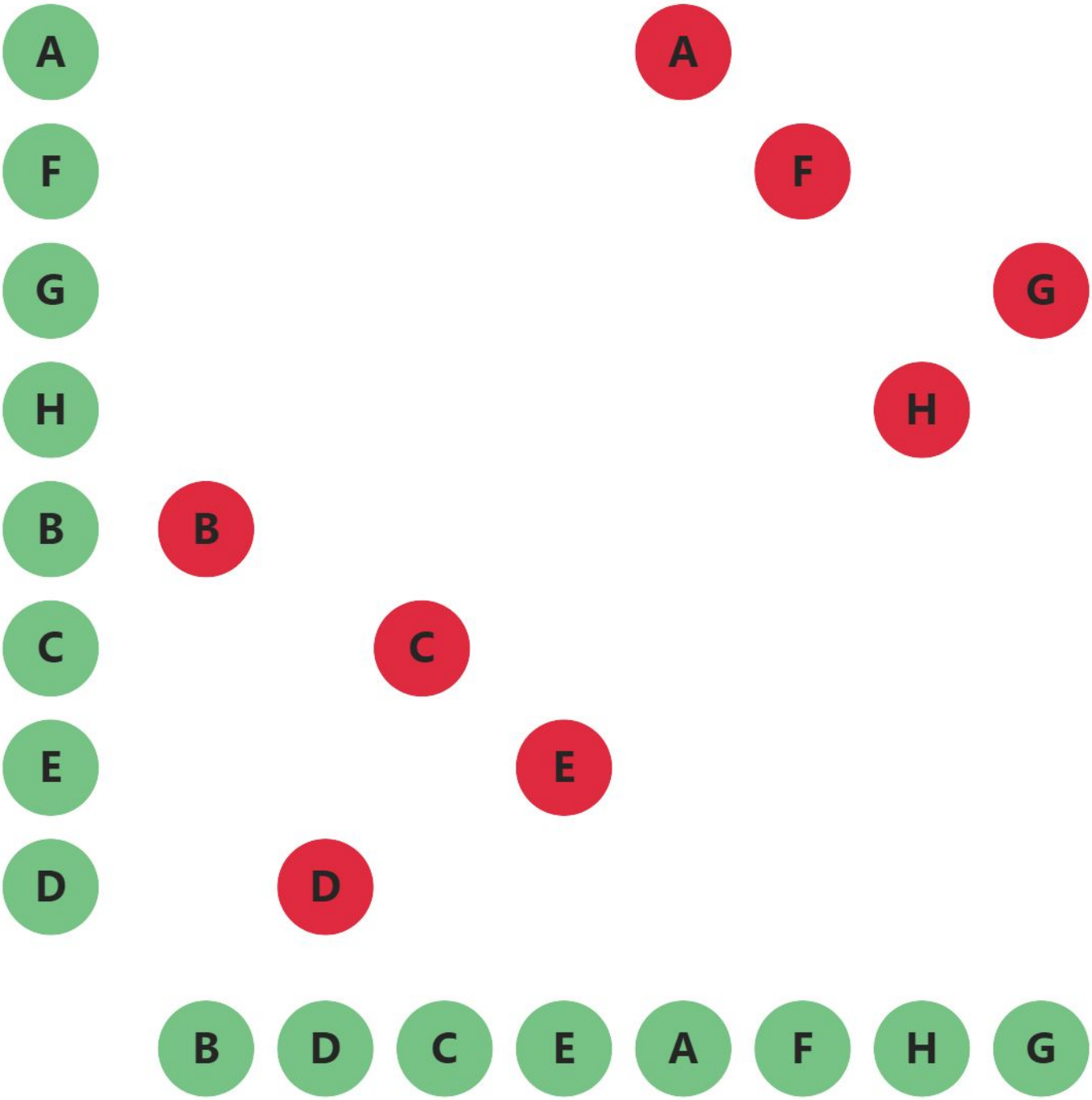


后：`DECBHGFA`

中：`BDCEAFHG`

后：DECBHGFA

中：BDCEAFHG



已知后续和中序时需要把后序倒过来的原因，应该是后序遍历的最后一个结点为根结点，为了保证画出来的二叉树的根结点在最上方，所以需要把后序倒过来（前序遍历第一个结点就是根结点，所以不用倒过来。）

二叉排序树

又名二叉查找树。

性质

1. 如果该树的左子树不为空，那么左子树上的所有节点的值都小于根节点的值
2. 如果该树的右子树不为空，那么右子树上的所有节点的值都大于根节点的值
3. 它的左右子树分别也是二叉排序树

```
#ifndef __TREE_H__
#define __TREE_H__

#include <stdio.h>
#include <stdlib.h>


typedef int TElemType; //树中结点的元素的类型

typedef struct binode
{
    TElemType data ; //保存结点的数据
    struct binode * lchild; //保存左子结点的指针
    struct binode * rchild; //保存右子结点的指针
}BINODE;

BINODE *Creat_Tree(char *s);
BINODE * Add_Node(BINODE *root,BINODE *n);
void Pre_Order(BINODE *t);
void Min_Order(BINODE *t);
void Post_Order(BINODE *t);
int Height(BINODE * root);
void Lever_Order(BINODE * root)

#endif
```

```c
#include "tree.h"
BINODE * Add_Node(BINODE *root,BINODE *n)
{
    if(root == NULL)
    {
        return n;
    }
    if(n == NULL)
    {
        return root;
    }
    BINODE * p = root;
    while(p)
    {
        //n要插到p的右边
        if(n->data > p->data)
        {
            //p有右子树
            if(p->rchild)
            {
                p = p->rchild;
            }
            else{
                p->rchild = n;
                break;
            }
        }
        else if(n->data < p->data)
        {
            //p有右子树
            if(p->lchild)
            {
                  p = p->lchild;
            }
            else{
                p->lchild = n;
                break;
            }
        }
        else{
            return root;
        }
    }
    return root;
}
```

```
//从键盘上输入节点的元素，创建一个二叉排序树
BINODE *Creat_Tree(char *s)
{
    BINODE *root = NULL;
    while(*s != '\0')
    {
        BINODE *p = (BINODE *)malloc(sizeof(BINODE));
        p->data = *s;
        p->lchild = p->rchild = NULL;
        //将节点p插入到二叉排序树中
        root = Add_Node(root,p);
        //printf("%c",p->data);
        s++;
    }
    return root;
}

//先序
void Pre_Order(BINODE *t)
{
    if(t == NULL)
    return;
    printf("%c",t->data);
    Pre_Order(t->lchild);
    Pre_Order(t->rchild);
}
//中序
void Min_Order(BINODE *t)
{
    if(t == NULL)
    return;
    Min_Order(t->lchild);
    printf("%c",t->data);
    Min_Order(t->rchild);

}
//后序
void Post_Order(BINODE *t)
{
    if(t == NULL)
    return;
    Post_Order(t->lchild);
    Post_Order(t->rchild);
    printf("%c",t->data);
```

```
}
//层次遍历（用到队列的相关函数，队列的初始化，入队，出队等）
//InitQueue()：队列的初始化
//EnQueue：入队函数
//DeQueue：出队函数
//LinkedQueue：队列相关结构体
//QueueIsEmpty：判断队列是否为空
void Lever_Order(BINODE * root)
{
    if(root == NULL) return ;
    //初始化一个队列
    LinkedQueue * lq = InitQueue();
    //把根节点入队
    EnQueue(lq,root);
    //如果队列不为空
    while(QueueIsEmpty(lq) == 0)
    {
        BINODE * temp;
        //出队 并访问出队元素
        DeQueue(lq,&temp); //temp就保存了出队元素
        printf("%c",temp->data);
        //将出队元素的左子结点(如果有)和右子结点(如果有)入队
        if(temp->lchild)
        {
            EnQueue(lq,temp->lchild);
        }
        if(temp->rchild)
        {
            EnQueue(lq,temp->rchild);
        }
    }
}
//求一棵树的高度
int Height(BINODE * root)
{
    if(root == NULL) return 0;

    int l = Height(root->lchild);
    int r = Height(root->rchild);
    return l>r? l+1 : r+1;
}
```

```c
#include "tree.h"

int main()
{
    char tree[100] = {0};
    scanf("%s",tree);
    BINODE *root = Creat_Tree(tree);

    printf("先序遍历: ");
    Pre_Order(root);
    puts(" ");

    printf("中序遍历: ");
    Min_Order(root);
    puts(" ");

    printf("后序遍历: ");
    Post_Order(root);
    puts(" ");
    return 0;
}
```

删除元素

```
BINODE *Del_Node(BINODE *root,TElemType x)
{
    if(root == NULL)
    {
        return root;
    }
    BINODE *px = root;    //px用于寻找要删除元素
    BINODE *pf = root;    //pf用来保存px的父节点
    while (px)
    {
        if(px->data == x)
        {
            break;
        }
        if(x > px->data)
        {
            pf = px;
            px = px->rchild;
        }
        else{
            pf = px;
            px = px->lchild;
        }
    }
    if(px == NULL)
    {
        return root;
    }
//delete:
    //分情况删除
    //1.要删除的那个节点是叶子节点
    if(px->lchild == NULL && px->rchild == NULL)
    {
        //可能是根节点
        if(px == root)
        {
            free(px);
            return NULL;
        }
        //判断删除的节点是它的父节点的左孩子还是右孩子
        if(px == pf->lchild)
        {
            pf->lchild = NULL;
        }
        else{
            pf->rchild = NULL;
        }
```

```
        free(px);
        return root;
    }
    //2.L为空，R不为空
    else if(px->lchild == NULL)
    {
        //要删除的节点是根节点
        if(px == root)
        {
            root = px->rchild;
            free(px);
            return root;
        }
        //判断删除的节点是它的父节点的左孩子还是右孩子,让删除节点的右孩子
        //接替要删除的节点，使其称为父节点新的左孩子或者右孩子
        if(pf->lchild == px)
        {
            pf->lchild = px->rchild;
            free(px);
            return root;
        }
        else
        {
            pf->rchild = px->rchild;
            free(px);
            return root;
        }
    }
    //3.L不为空，R为空
    else if(px->rchild == NULL)
    {
        //要删除的节点是根节点
        if(px == root)
        {
            root = px->lchild;
            free(px);
            return root;
        }
        //判断删除的节点是它的父节点的左孩子还是右孩子,让删除节点的右孩子
        //接替要删除的节点，使其称为父节点新的左孩子或者右孩子
        if(pf->lchild == px)
        {
            pf->lchild = px->lchild;
            free(px);
            return root;
        }
        else
        {
```

```
            pf->rchild = px->lchild;
            free(px);
            return root;
        }
    }
    //4.L,R都不为空
    else{
        BINODE * px1 = px->lchild;   //px1用来查找要删除节点的左子树的最右的节点
        pf = px;     ///保存px1的父节点
        while(px1->rchild)
        {
            pf = px1;
            px1 = px1->rchild;
        }
        px->data = px1->data;
        px = px1;
        //goto delete;
        pf->rchild = px->rchild;
        free(px);
        return root;
    }
}
```

先序遍历的非递归实现

```c
/*
    先序遍历的非递归实现之一(利用栈)
    约定：入栈之前先访问
    1.首先让根节点进栈
    2.进栈元素，让其左子树进栈，直到左子树为空。
    3.出栈，转向其右子树(把出栈元素的右子树进栈)  goto 2
*/
void Pre_Order_Stack(BINODE * root)
{
    if(root == NULL) return ;
    LinkedStack * s = InitStack(); //构建一个链式栈
    //1.首先让根节点进栈
    printf("%c",root->data); //入栈之前先访问
    Push(s,root);
    while(StackIsEmpty(s) == 0)
    {
        //2.入栈元素 如果其左子树存在 那么就让他的左子树一直入栈 直到左子树为空
        while(GetTop(s)->lchild)
        {
            //入栈之前先访问
            printf("%c",GetTop(s)->lchild->data);
            Push(s,GetTop(s)->lchild);
        }
        //3.不断出栈 直到出栈元素有右子树停止出栈 并将其右子树入栈 然后回到第2个操作
        //如果出栈到栈为空 就结束循环
        while(StackIsEmpty(s) == 0)
        {
            BINODE * temp;
            Pop(s,&temp); // 出栈 并temp就是保存了出栈元素
            if(temp->rchild) //出栈元素有右子树
            {
                //停止出栈 让其右子树入栈
                printf("%c",temp->rchild->data); //入栈之前先访问
                Push(s,temp->rchild);
                break;
            }
        }

    }
}
```