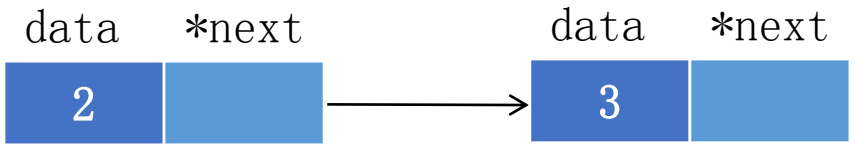
基础数据结构

湖南师大附中 许力

链表（linked list）

- 链表的结构顾名思义：链状的线性数据结构
- 比数组复杂的是：链表中的每个节点除了记录自身的数据外，还需要记录后继节点是谁



插学：指针

```
7405124
10

--------------------------------

Process exited after 0.235 seconds wi
th return value 0
请按任意键继续. . .
```

- 学习链表之前先补一点指针
- 指针其实也是变量，不过不是普通变量
- 是专用来存普通变量的地址的变量

```cpp
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a = 10;
    int *p = &a;
    printf("%d\n", p);
    printf("%d\n", *p);
    return 0;
}
```

- 注意这里：

```
*p = a
```

- 会因类型不匹配而报错

动态存储

- 什么是动态存储？
- 我们自己手动分配内存空间（供自己使用），使用完毕后又自己释放该内存空间
- 相对的，静态存储就是一切被系统安排得明明白白，只管用

- 动态存储有什么好处呢？就是可以在程序里，自由决定什么时候需要多大的内存空间，而不是像开静态数组那样直接开一个足够大的先放在那儿备用

malloc 开内存函数

- malloc函数的作用是向系统申请内存空间

```
malloc(4);  //开4个字节内存;

malloc(sizeof(double)); //开一个double所需的内存
```

- 头文件<stdlib>

malloc 开内存函数

- 但是这一片内存开好了之后，怎么用？
- 这时就必须用一个指针变量，来指向这片内存的首地址

```
double *p;
p = (double *) malloc(sizeof(double));
```

malloc 开内存函数

```
136192
10

--------------------------------

Process exited after 0.1943 seconds w
ith return value 0
请按任意键继续. . .
```

- 现在再玩一下：

- 注意这时的地址随着每次调用，会出现变化
- 这点和之前的静态地址不一样

- 然后用完内存之后，还可以释放内存

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a = 10;
    int *p;
    p = (int *) malloc(sizeof(int));
    *p = a;
    printf("%d\n", p);
    printf("%d\n", *p);
    free(p);  //释放内存
    return 0;
}
```

new 操作符

- 除了malloc函数外，我们也可以用new操作符来动态申请内存

```
int *p;  //定义指针变量 p
p = new int;
//为 p新开一个大小可以存放 int的内存空间
```

new 操作符

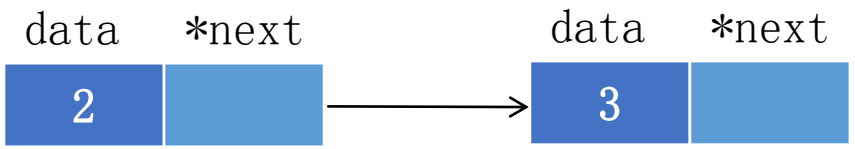
- 继续玩
- 注意释放内存是delete

```cpp
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a = 10;
    int *p;
    p = new int;
    *p = a;
    printf("%d\n", p);
    printf("%d\n", *p);
    delete(p);  //释放内存
    return 0;
}
```

链表

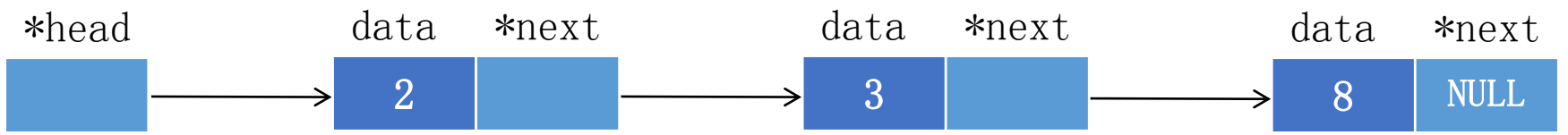
- 链表的结构顾名思义：链状的线性数据结构
- 比数组复杂的是：链表中的每个节点除了记录自身的数据外，还需要记录后继节点是谁
- 那显然我们需要开结构体才可以做到这一点

```
struct node
{
    int data;
    node *next;   //next存后继节点的地址
};
```



链表

- 那显然我们需要开结构体才可以做到这一点
- 一般链表需要一个头指针head，存放第一个元素的地址，没有数据域
- 如果next里存的是NULL（空指针，全大写），那说明当前节点是链尾



链表的代码实现

```
#include<bits/stdc++.h>
using namespace std;
struct node
{
    int data;
    node *next;
};
void creat(int x)  //创建值为 x的链表节点
{
    node *p;
    p = new node;
    p -> data = x;
    p -> next = NULL;
    delete(p);
}
```

```
int main()
{
    int n, x;
    scanf("%d", &n);
    while (n --)
        scanf("%d", &x), creat(x);
    return 0;
}
```

链表的代码实现

- 这个符号：

```
p -> data = x;
```

- 用于以指针形式访问结构体元素

- 注意非指针形式访问结构体元素我们用的是：

```
student.score = x;
```

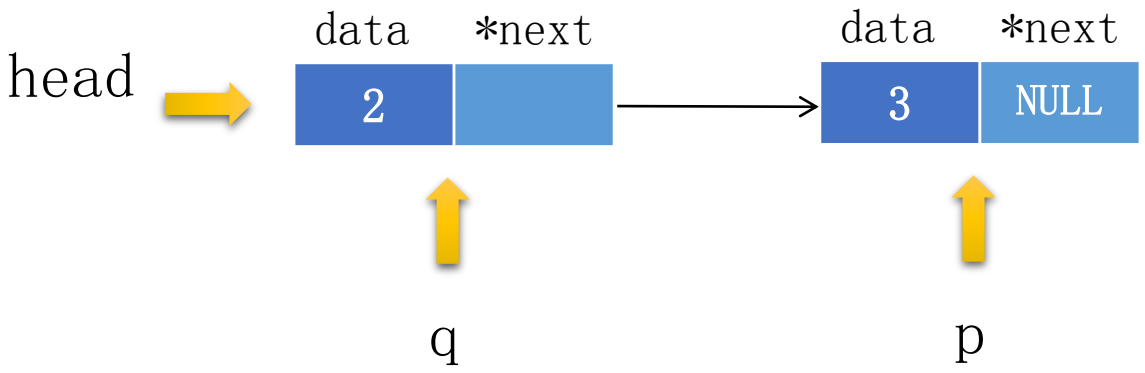
链表的代码实现

- 但是刚才这段代码并没有什么用
- 因为这些节点散落着并未串成“链”，所以我们也无法顺利地访问它们

```
void creat(int x)
{
    node *p;
    p = (node *) malloc(sizeof(node));
    p -> data = x;
    p -> next = NULL;
    free(p);
}
```

链表的代码实现

- 我们还需要建立前后指针

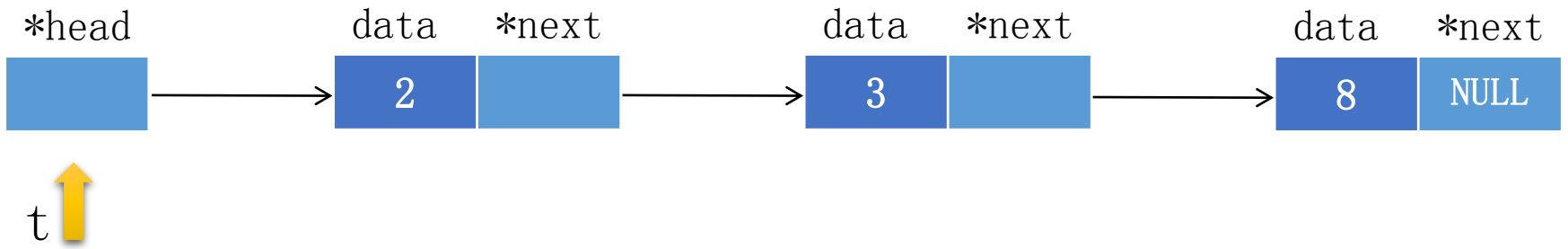



```
void creat(int x)
{
    node *p, *q, *head;
    p = (node *) malloc(sizeof(node));
    p -> data = x;
    p -> next = NULL;
    if (head == NULL) head = p;  //设立头指针方便访问
    else q -> next = p;  //上一个节点 q指向当前节点 p
    q = p; // 最后 q也要指向当前节点,以便顺序遍历
    //free(p);
}
```

链表的代码实现

- 访问时，可以依序进行

```
node *t;
while (t != NULL)
{
    printf("%d ", t -> data);
    t = t -> next;  //继续访问下一个节点
}
```

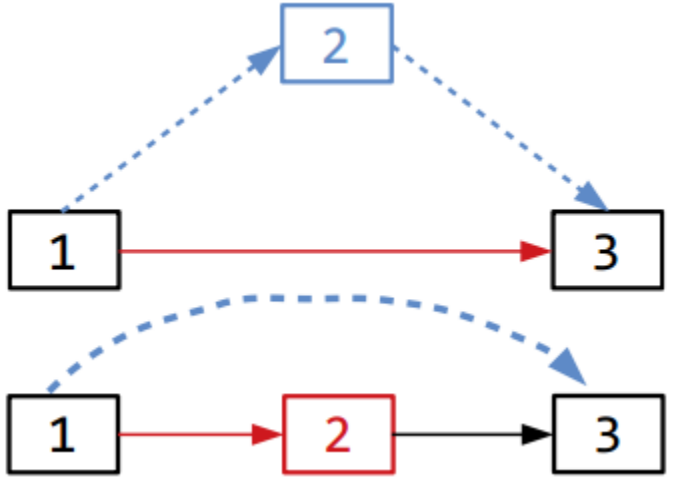


链表的插入和删除

- 链表支持O(1)复杂度的插入和删除操作

- 删除节点node2：

1. node1->next = node2->next;



链表的插入和删除

- 链表的删除

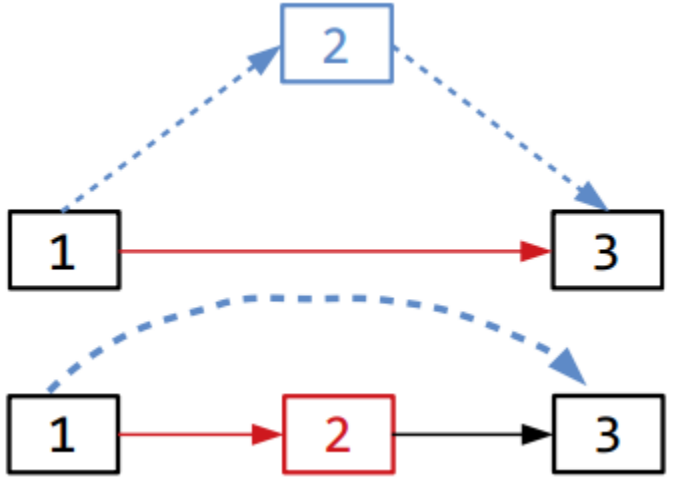
```
void del(node *p)  //删除节点 p后面的节点
{
    node *q;
    q = p -> next;
    p -> next = q -> next;
    delete(q);
}
```

链表的插入和删除

- 链表支持O(1)复杂度的插入和删除操作

- 插入节点node2到node1和node3之间：

```
1.  node1->next = node2;
2.  node2->next = node3;
```



链表的插入和删除

- 插入操作

```
void add(node *p, int x)  //插入元素 x到节点 p后
{
    node *q;
    q = new node;
    q -> data = x;
    q -> next = p -> next;
    p -> next = q;
}
```

链表的插入和删除

- 数组的缺点刚好是链表的优点
- 链表的缺点刚好是数组的优点

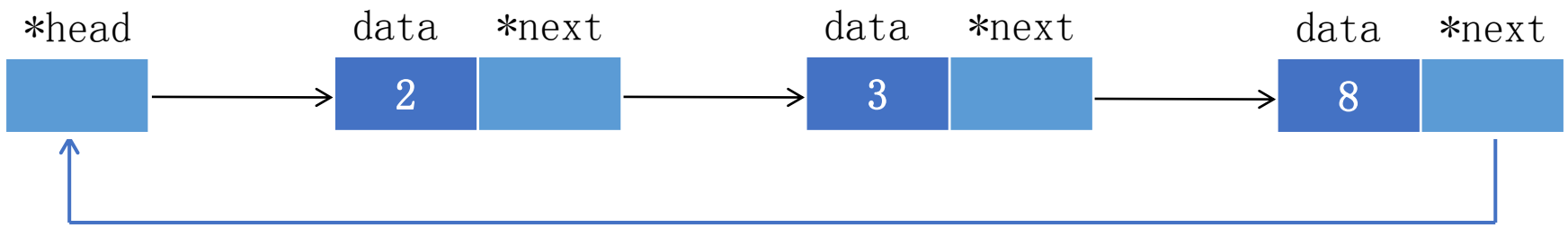
链表的拓展：双向链表

- 每个节点除了记录后继节点外，还记录前驱节点，使得遍历链表时可以在链表中双向移动

```
struct node
{
    int data;
    node *pre, *next;
};
```

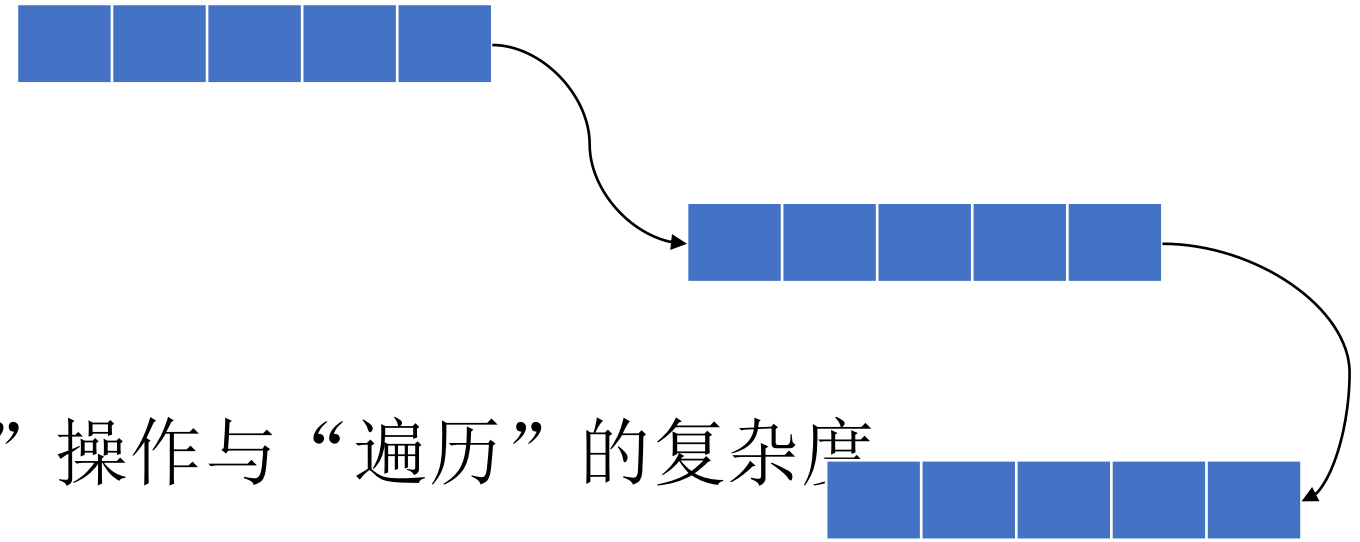
链表的拓展：循环链表

- 链尾节点的*next不指向NULL，而是指向链首节点，使得遍历链表时可以从链表中任意一个元素开始遍历整个链表



链表的拓展：块状链表

- 块状链表是链表和数组混合的产物：



- 这样可以平衡“插入/删除”操作与“遍历”的复杂度
- 复杂度均摊为$\sqrt{n}$

课外加练

- luogu 1996　　约瑟夫问题
- luogu 1160　　队列安排

树形数据结构

二叉树/堆/哈夫曼树/并查集/线段树/树状数组

树（tree）

- 树是一种非线性的数据结构
- **父节点/儿子节点**

1号节点是2的父节点

5号节点是2的右儿子节点，同时又是10的父节点

12号节点是6的左儿子节点

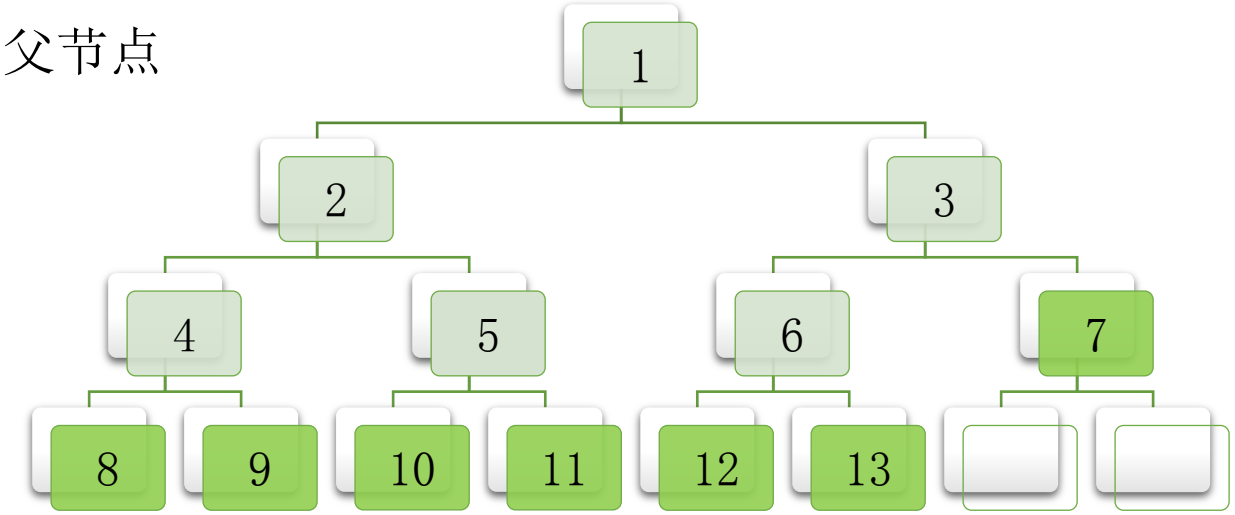
……

- **根节点**

没有父节点的节点

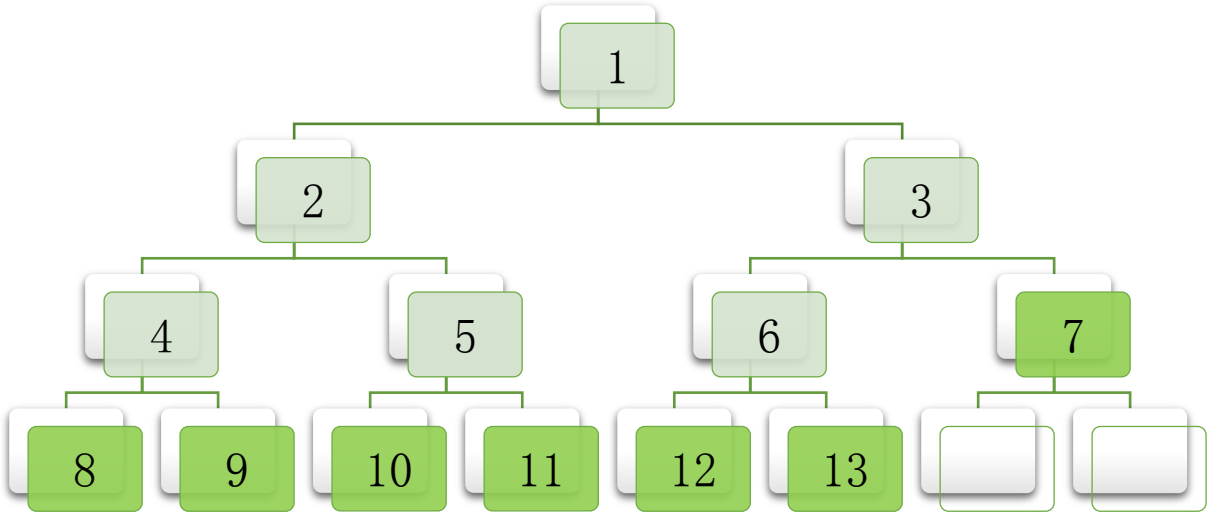
- **叶节点**

没有儿子节点的节点



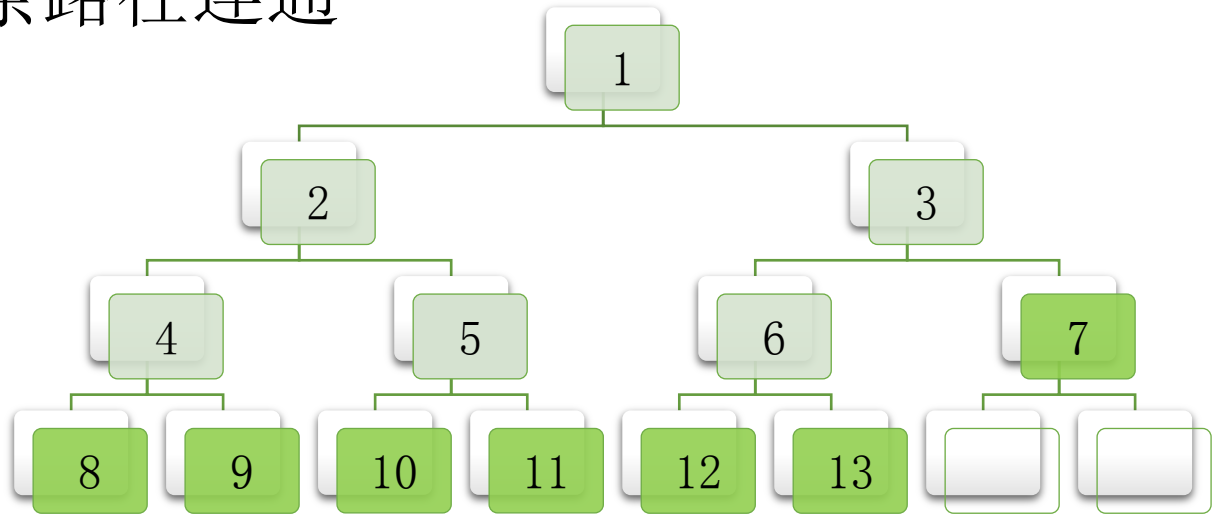
树

- 树是一种非线性的数据结构
- 树具有天然的层次和递归特点



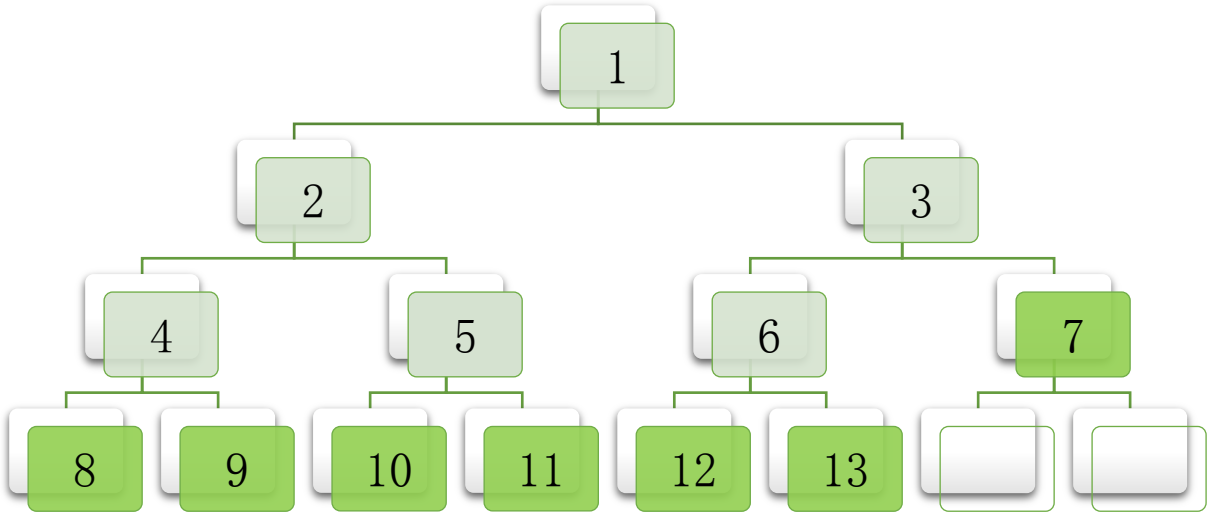
树

- 树也可以这么定义：
- 有n个节点，n-1条边的无向无环连通图
- 任意两个节点间有且仅有一条路径连通



二叉树（binary tree）

- 任意节点至多有两个儿子节点的树
- 满二叉树
- 完全二叉树



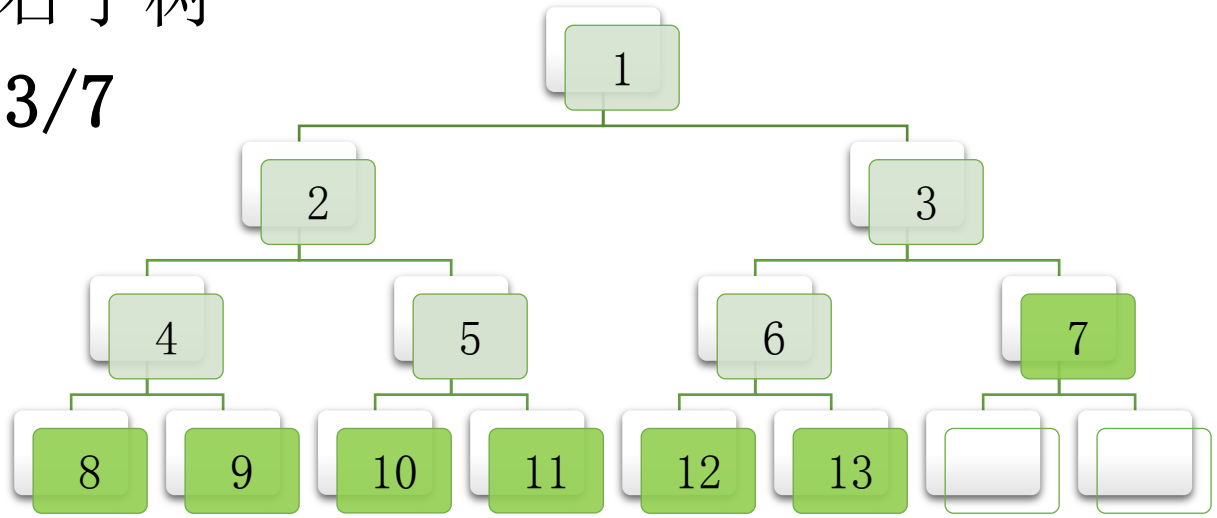
二叉树的遍历方式

- 二叉树的遍历方式有四种：

1. 前序（先根）遍历

先访问根节点，再左子树，再右子树

1/2/4/8/9/5/10/11/3/6/12/13/7



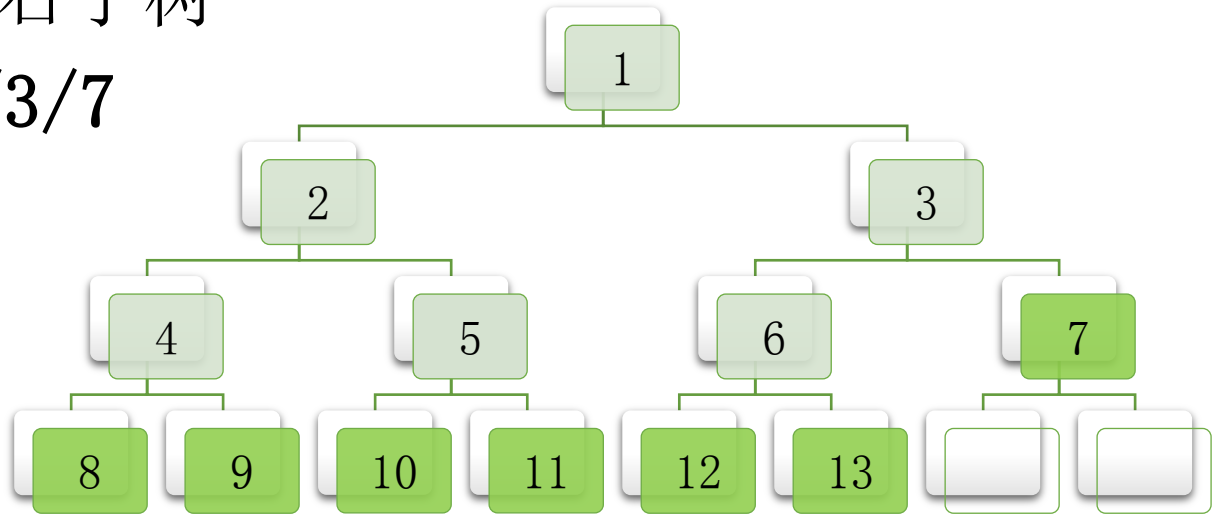
二叉树的遍历方式

- 二叉树的遍历方式有三种：

2. 中序（中根）遍历

先访问左子树，再根节点，再右子树

8/4/9/2/10/5/11/1/12/6/13/3/7



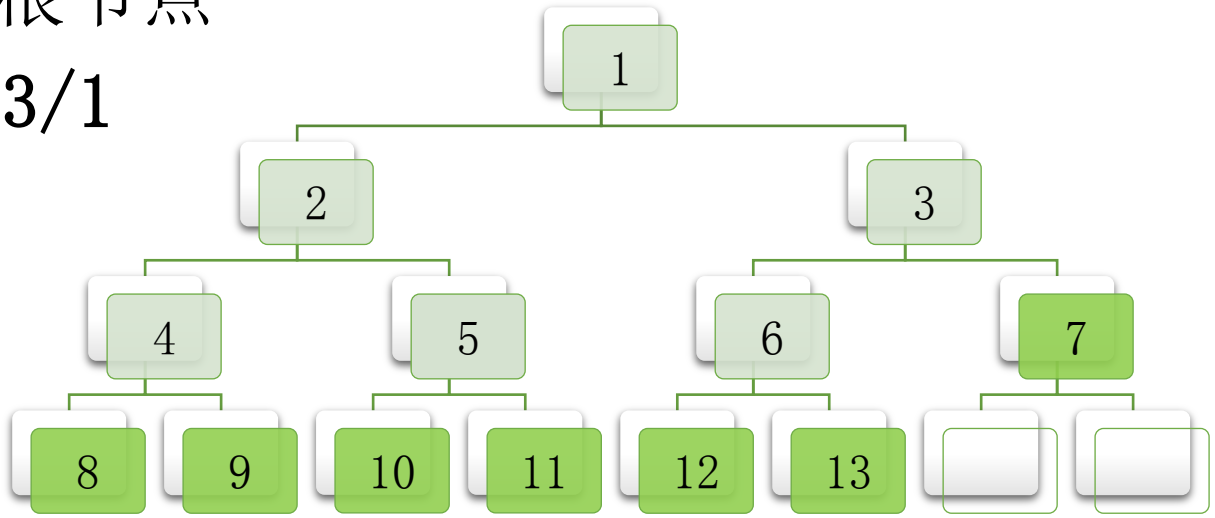
二叉树的遍历方式

- 二叉树的遍历方式有三种：

3. 后序（后根）遍历

先访问左子树，再右子树，再根节点

8/9/4/10/11/5/2/12/13/6/7/3/1



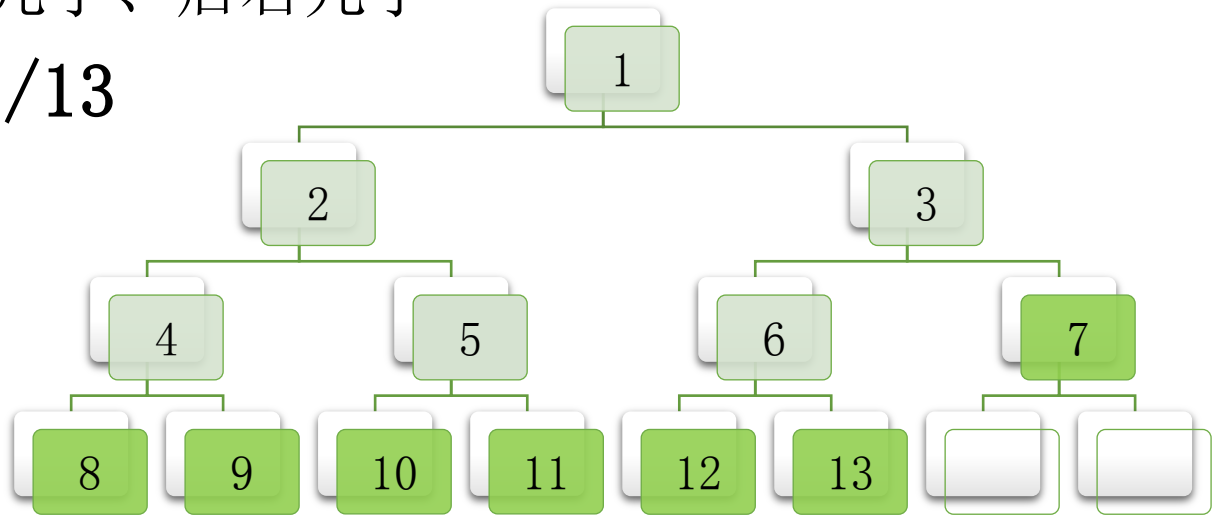
二叉树的遍历方式

- 二叉树的遍历方式有三种：

4. 层次遍历

按照儿子节点的层次先根节点，左儿子、后右儿子

1/2/3/4/5/6/7/8/9/10/11/12/13



问

- 一棵二叉树的前序遍历序列是ABCDEFG，后序遍历序列是CBFEGDA，则根节点的左子树的节点个数可能是？

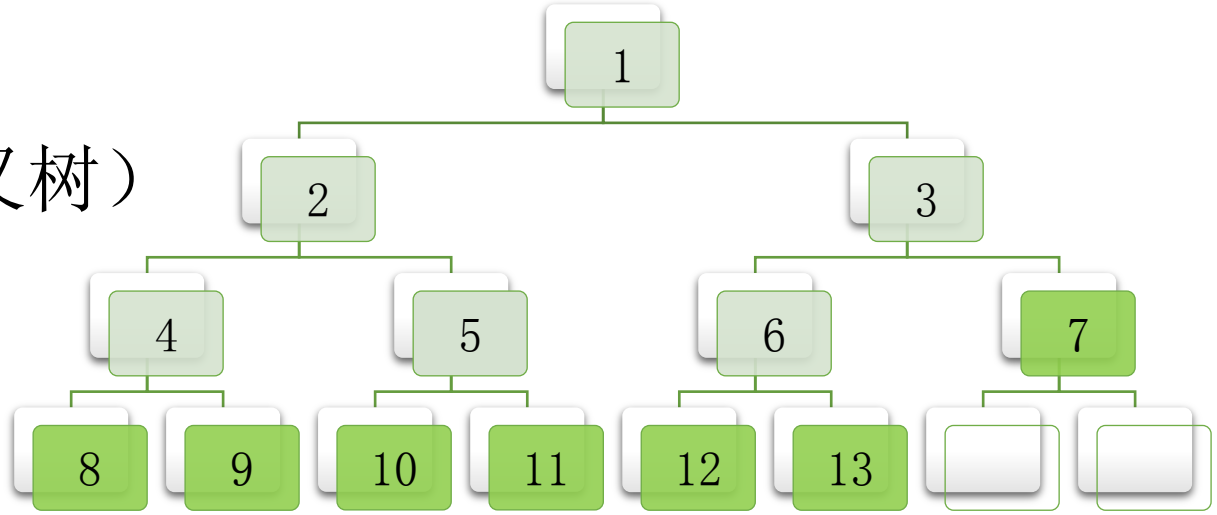
2

二叉树的存储

- 树非线性结构，要想完整存下一棵树，需要从根节点开始，记录每个节点的左右儿子节点

- 两个办法

1. 纯数组模拟（仅限完全二叉树）
2. 每个节点开一个结构体

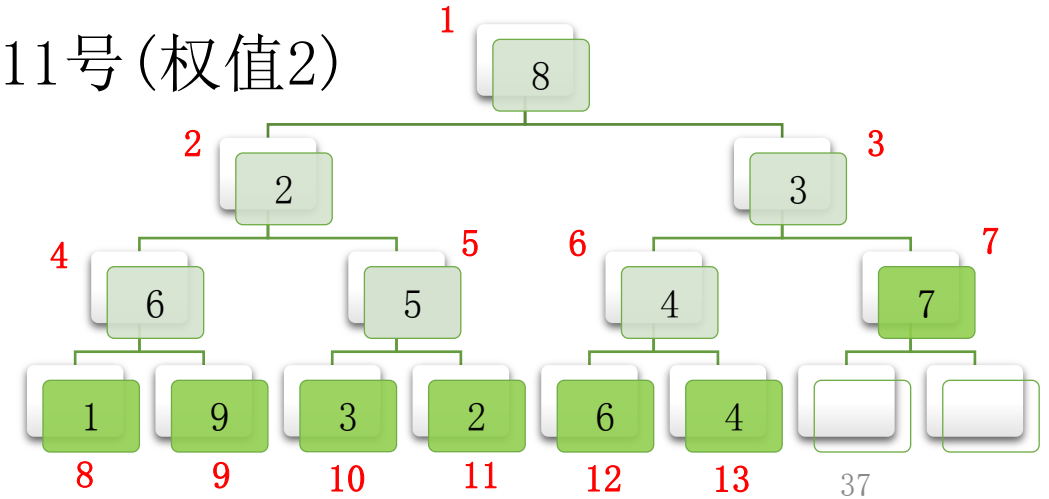


二叉树的存储

1. 以这棵树为例：我们用数组这样存：

1	2	3	4	5	6	7	8	9	10	11	12	13
8	2	3	6	5	4	7	1	9	3	2	6	4

- 2号节点(权值2)的左右儿子是4号(权值6)和5号(权值5)
- 5号节点(权值5)的左右儿子是10号(权值3)和11号(权值2)



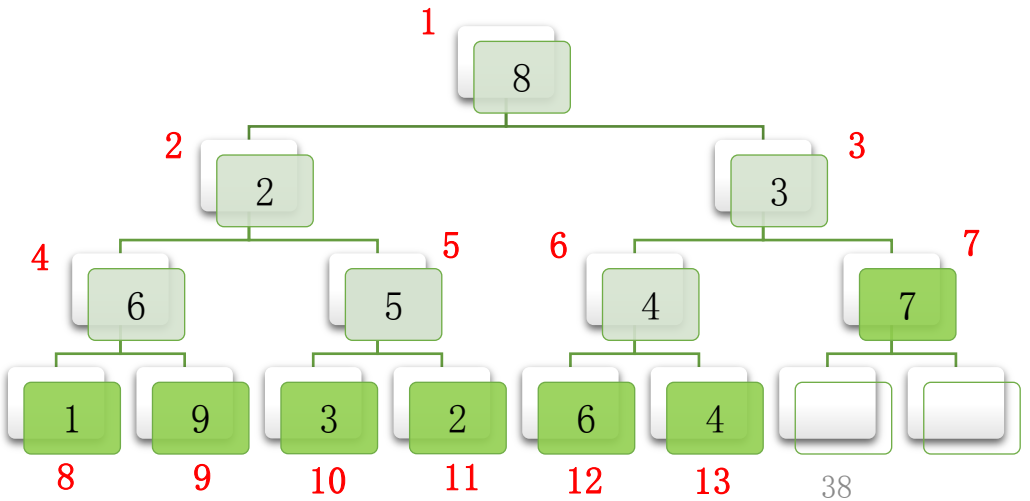
二叉树的存储

1. 以这棵树为例：我们用数组这样存：

1	2	3	4	5	6	7	8	9	10	11	12	13
8	2	3	6	5	4	7	1	9	3	2	6	4

（完全二叉树的数组表示中）

- 父节点a[i]的左儿子节点是a[2*i]
- 右儿子是a[2*i+1]

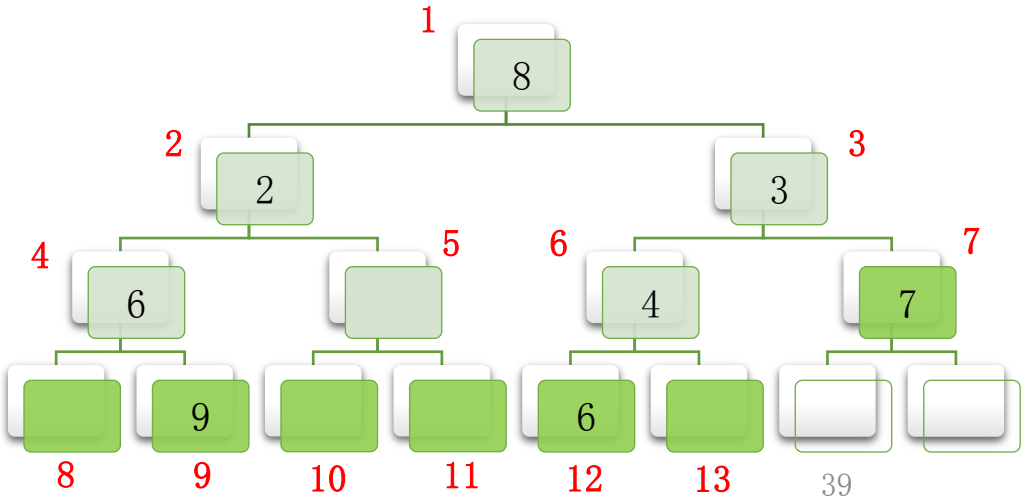


二叉树的存储

1. 以这棵树为例：我们用数组这样存：

1	2	3	4	5	6	7	8	9	10	11	12	13
8	2	3	6		4	7		9			6	

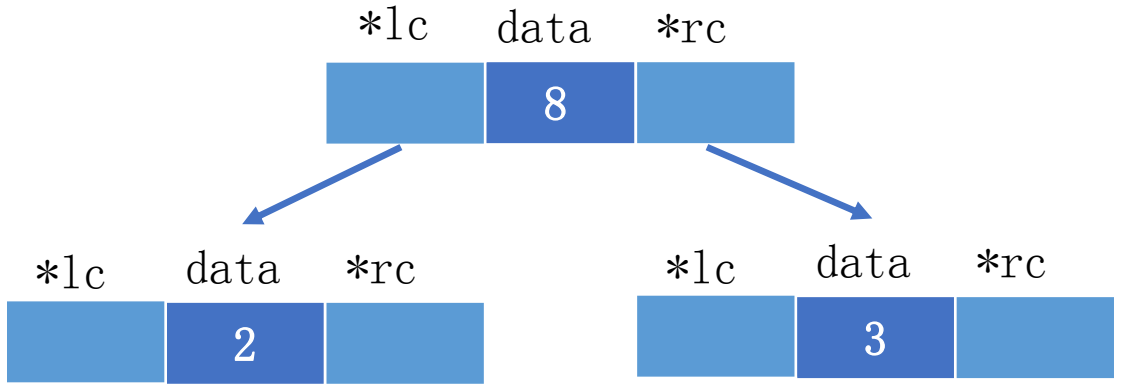
但是这样的存储方式对于非完全二叉树来说，就太浪费



二叉树的存储

2. 每个节点开一个结构体，然后既可以链式存储，也可以线性存储

```
struct node
{
    int data;
    node *lc, *rc;
};
```



二叉树的存储

- 建树递归进行
- 分别建立根节点、左子树、右子树

```
typedef node *tree;  //定义 tree为 node型节点
void build(tree root)
{
    int x;
    scanf("%d", &x);
    if (root)
    {
        root = new node;  //建立根节点
        root -> data = x;
        build(root -> lc);  //递归建左子树
        build(root -> rc);  //递归建右子树
    }
    else root = NULL;
}
```

二叉树的遍历

- 从根节点开始先序遍历

```
typedef node *tree;  //定义 tree为 node型节点
void pre(tree root)
{
    if (root)
    {
        printf("%d/", root -> data);
        pre(root -> lc);
        pre(root -> rc);
    }
}
```

二叉树的遍历

- 从根节点开始中序遍历

- 大家仔细观察会发现：

二叉搜索树的中序遍历将得到单调递增的访问序列

```
typedef node *tree;  //定义 tree为 node型节点
void mid(tree root)
{
    if (root)
    {
        mid(root -> lc);
        printf("%d/", root -> data);
        mid(root -> rc);
    }
}
```

二叉树的遍历

- 从根节点开始后序遍历

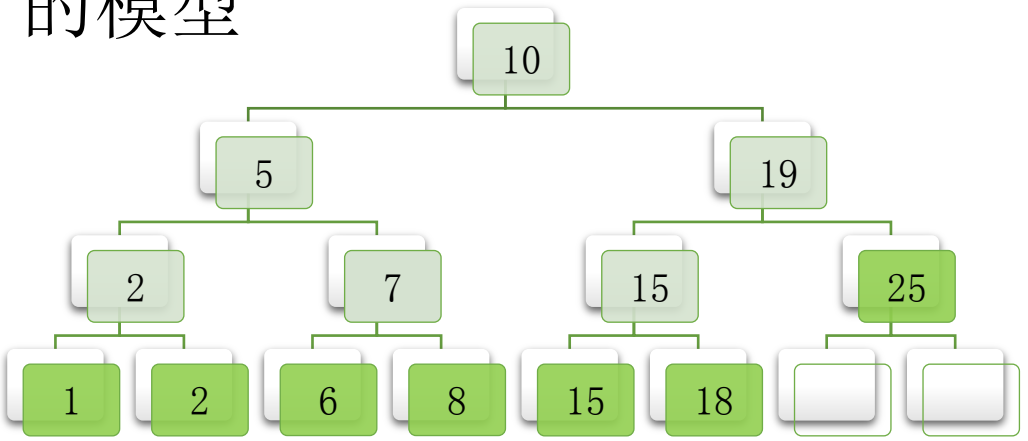
```
typedef node *tree;  //定义 tree为 node型节点
void next(tree root)
{
    if (root)
    {
        next(root -> lc);
        next(root -> rc);
        printf("%d/", root -> data);
    }
}
```

二叉搜索树

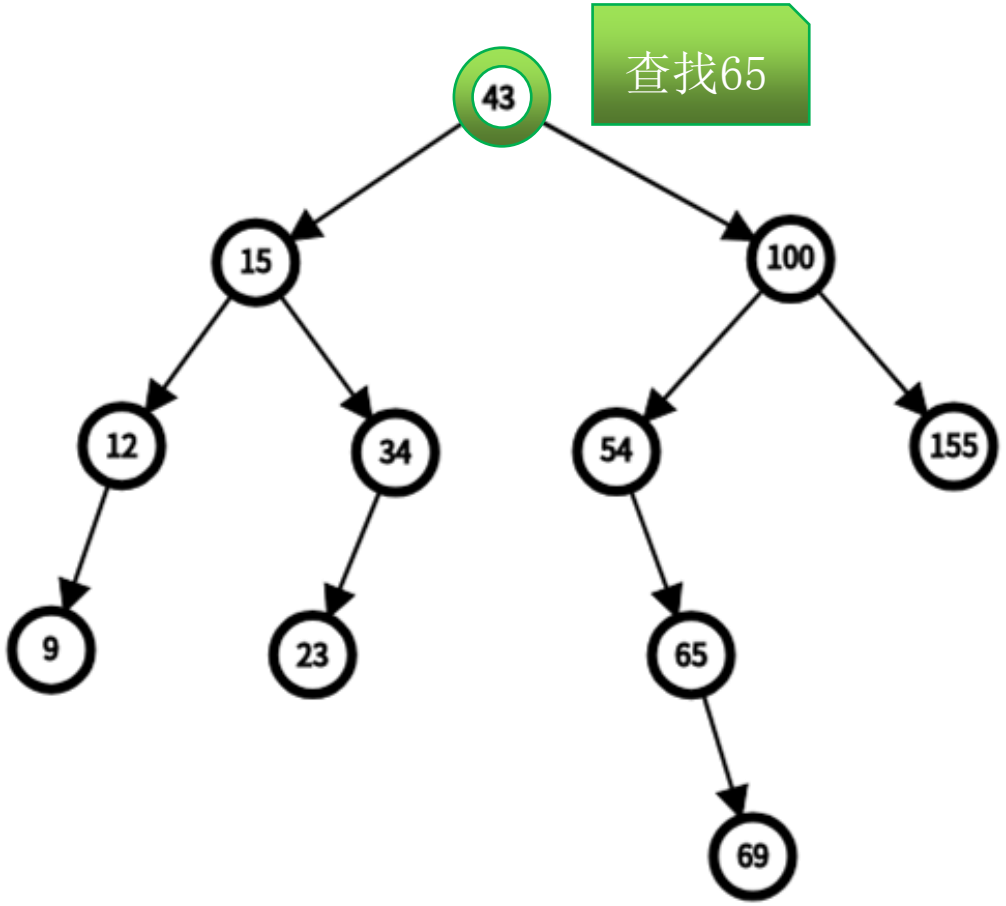
- 也叫二叉查找树，是满足特殊性质的二叉树：

1. 如果左子树不空，则不大于父节点，如果右子树不空，则不小于父节点
2. 左、右子树均符合递归定义

- 二叉搜索树其实就是“猜数游戏”的模型



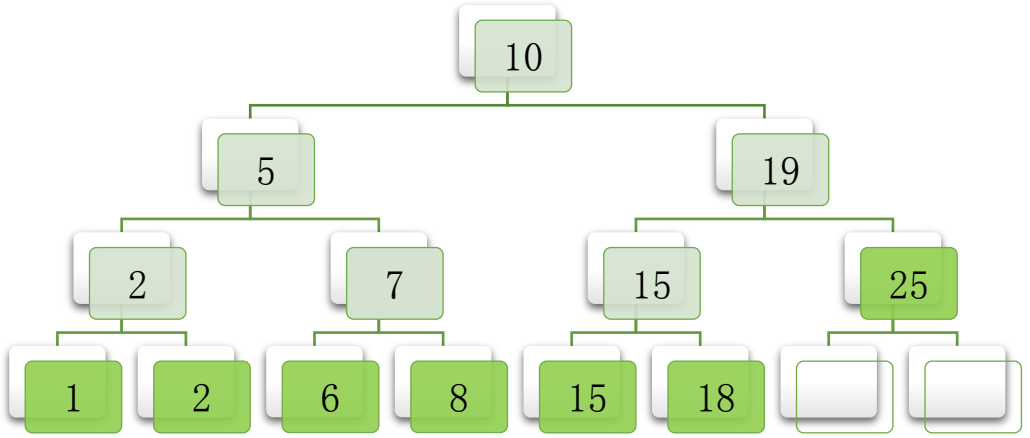
猜数游戏与二叉搜索树



二叉搜索树的建立

- 递归建树

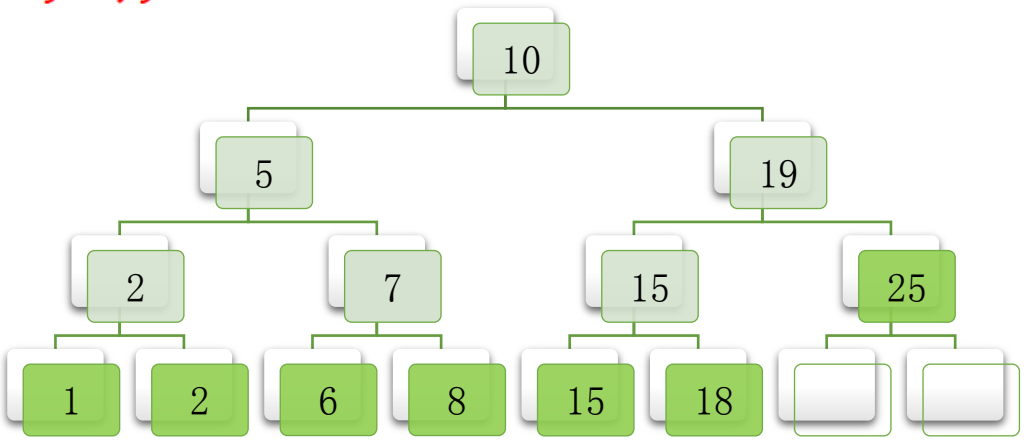
```
if (x <= root -> data) build(root -> lc);
else build(root -> rc);
```



二叉搜索树的查找

- 二分查找

```
tree find(tree root, int n)
{
    if (root)
    {
        if (n < root -> data) find(root -> lc, n);
        else if (n > root -> data) find(root -> rc, n);
        else return root;
    }
    else return NULL;
 }
```



课外加练

- luogu 1035　　新二叉树
- luogu 1030　　求先序排列
- luogu 1087　　FBI树
- luogu 5018　　对称二叉树