分治——计算完全二叉树节点数

原创 田郑书媛 LOA算法学习笔记 2021-02-19 19:35

计算完全二叉树的节点数是一个很简单的问题，只要像计算普通的二叉树一样遍历整个二叉树即可，这种算法的时间复杂度为O(n)。但是实际上，计算完全二叉树节点数的时间复杂度可以缩小到 $O((logn)^2)$ ，我们要用什么样的算法达到这个要求呢？下面介绍两种方法求完全二叉树的节点数，保证时间复杂度为 $O((logn)^2)$ ，n为节点数。

01 方法一

1. 思路

分析发现，完全二叉树具有这样的性质：

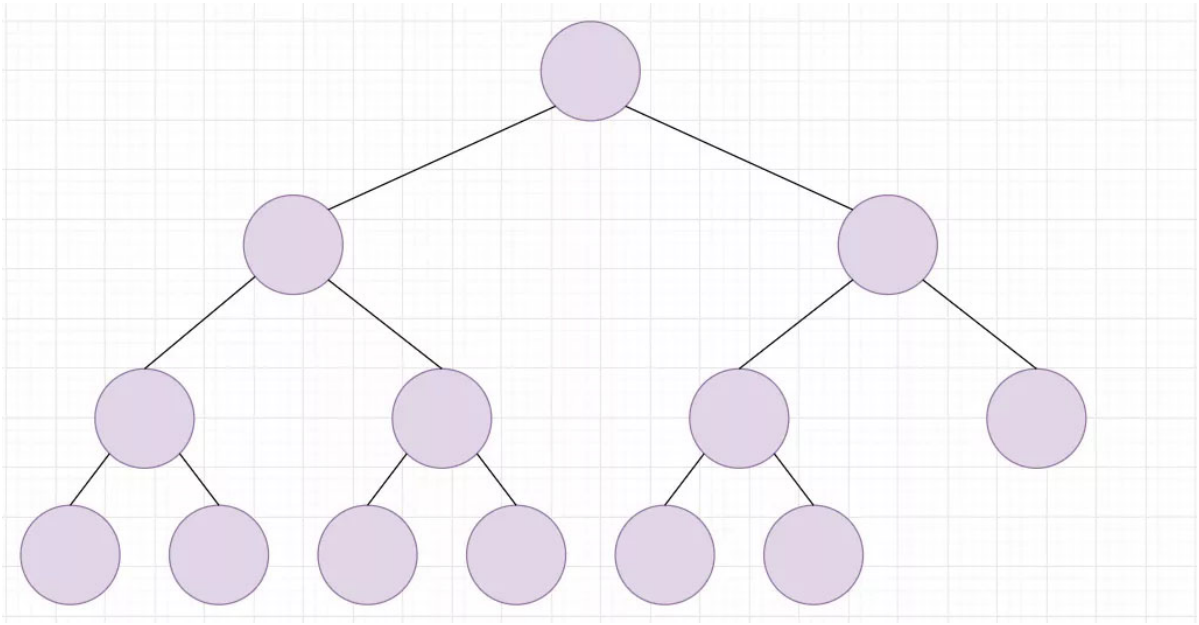

图1

1) 如果完全二叉树的右子树高度比整个二叉树高度少1，则左子树是满二叉树，如图1所示，左子树为满二叉树，整个二叉树高度为4，右子树高度为3。满二叉树的节点数可以直接根据其高度求出来，这样就有了左子树的节点数；而对于右子树，可以递归计算。

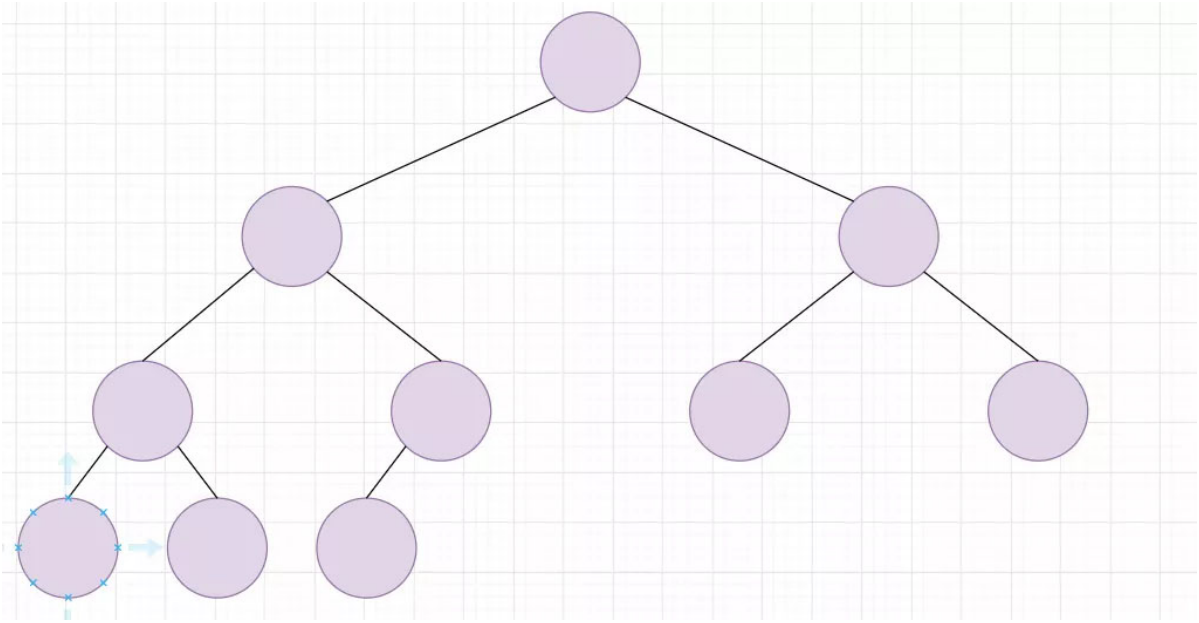

图2

2) 如果完全二叉树的右子树高度比整个二叉树高度少2，则右子树是满二叉树，如图2所示，右子树为满二叉树，整个二叉树高度为4，右子树高度为2。在这种情况下，右子树高度可以直接根据其高度求出，而左子树的节点数可以递归计算。

另外，计算满二叉树的高度可以从根节点一直往左下方走，计算经过的节点数，所需时间为log n。

2. C语言代码

```
#include
#include
#define NODE_NUM 100
char binary[NODE_NUM];
struct node{
    int value;
    struct node* left;
    struct node* right;
}list[NODE_NUM];


void init(){
    int i;
    for(i=0;binary[i]!=0;i++)
        list[i].value=binary[i]-'0';
    while(i<NODE_NUM){
        list[i].value=-1;
        i++;
    }
    for(i=0;i<NODE_NUM;i++){
        if((2*i+1)<NODE_NUM&&list[2*i+1].value==1){
            list[i].left=&list[2*i+1];
        }else{
            list[i].left=NULL;
        }
        if((2*i+2)<NODE_NUM&&list[2*i+2].value==1){
            list[i].right=&list[2*i+2];
        }else{
            list[i].right=NULL;
        }
    }
    return;
}
int height(struct node* point){
    int height=1;
    while(point->left!=NULL){
        point=point->left;
        height++;
    }
    return height;
}
int power(int d,int z){
    int i;
    int result=1;
```

```
    for(i=0;i<z;i++){
        result*=d;
    }
    return result;
}
int count(struct node* point){
    int whole_height,right_height;
    int left_num,right_num,num;
    whole_height=height(point);
    if(point->right!=NULL){
        right_height=height(point->right);
    }else{
        right_height=0;
    }
    if(whole_height==(right_height+1)){
        left_num=power(2,whole_height-1)-1;
        if(point->right==NULL){
            num=left_num+1;
        }else{
            num=left_num+count(point->right)+1;
        }
    }else{
        right_num=power(2,right_height)-1;
        if(point->left==NULL){
            num=right_num+1;
        }else{
            num=count(point->left)+right_num+1;
        }
    }
    return num;
}


int main(){
    int i,num;
    printf("input the binary tree:\n");
    scanf("%s",&binary);
    init();
    num=count(&list[0]);
    printf("there is totally %d nodes\n",num);
    return 0;
}
```

在main函数中提示用户以层次遍历的方式输入二叉树，然后调用init函数以链表的方式构建二叉树，接着调用count函数，输入的参数是整个二叉树的根节点。这个函数首先调用height函数计算整个二叉树和右子树的高度，如果右子树高度比整个二叉树高度少1，代表左子树是满二叉树，这样的话左子树里的节点数可以直接调用计算幂的power函数得到，然后递归调用count函数，输入的参数是右子树的根节点；如果右子树高度比整个二叉树的高度少2，代表右子树是满二叉树，这样的话右子树节点数可以直接根据其高度算出，而左子树的高度则需要递归调用count，并以左子树根节点为输入。

3. 子问题简化图

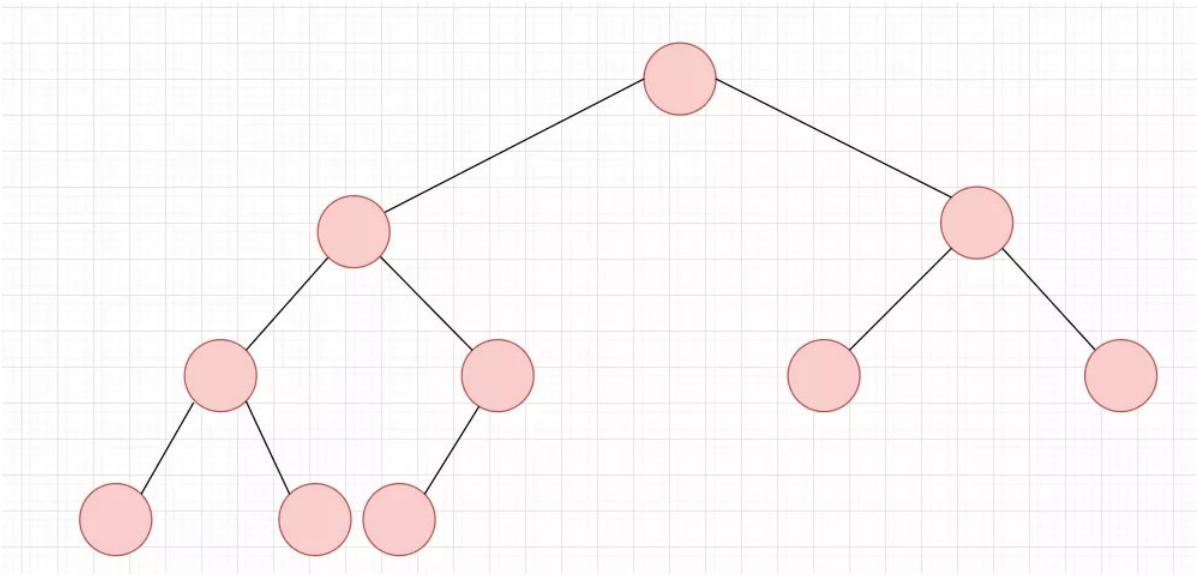

图3

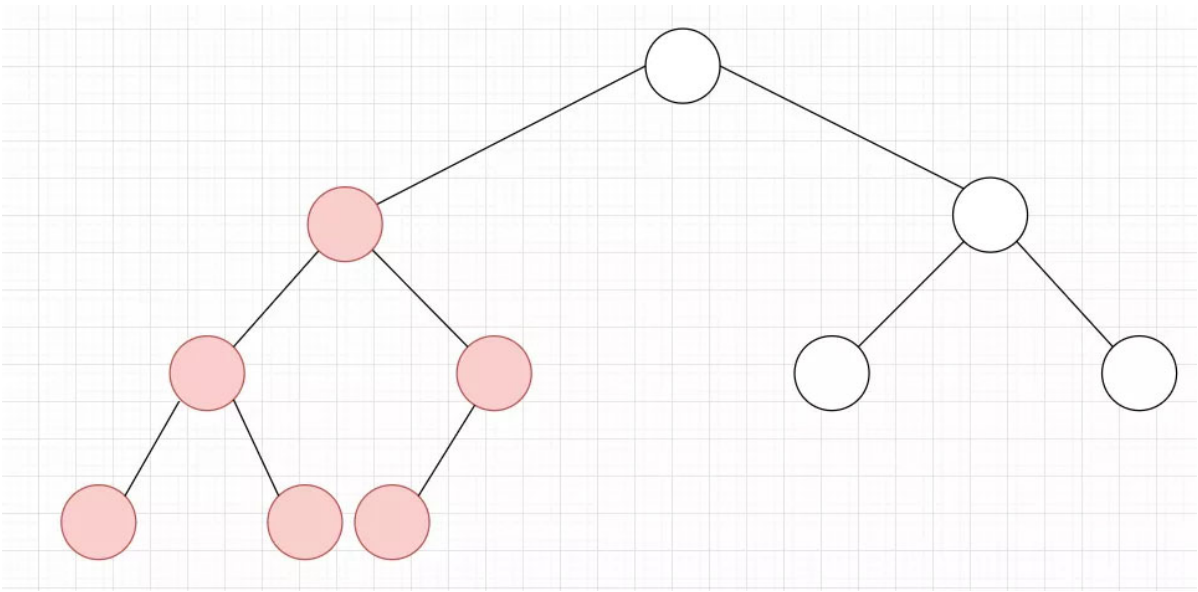

图4

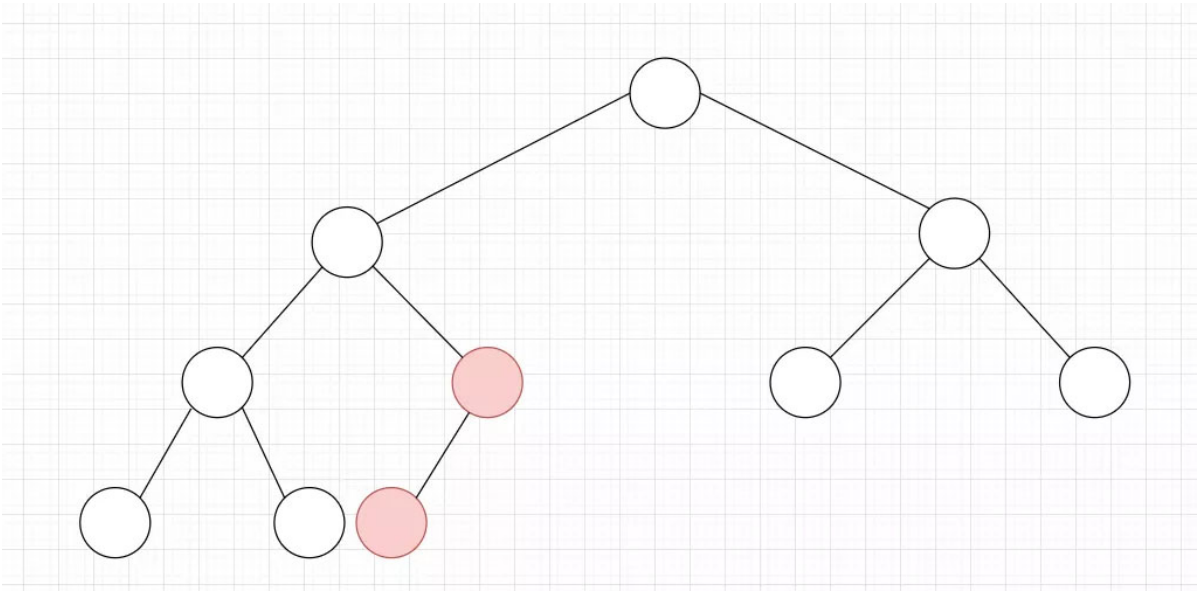

图5

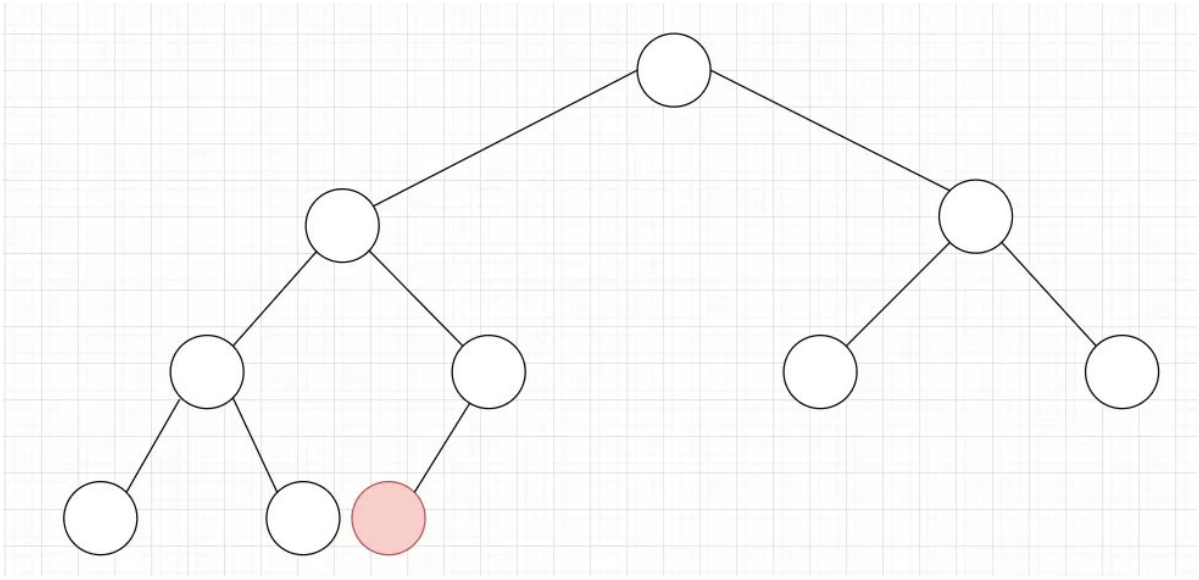

图6

上面的图3~图6是该算法的子问题简化图，粉色的节点代表仍然需要计算的节点数。

4. 正确性

算法的核心思想是完全二叉树的左子树和右子树中一定有一个是满二叉树，通过计算这个满二叉树的高度可以直接得到其节点数，而对另一个子树，使用同样的方法递归求其节点数。将求解一棵完全二叉树节点数的问题不断地分解为求其左右子树节点数的问题，得到问题的答案，可见其逻辑正确性。

5. 复杂度

对于一个有n个节点的二叉树，其高度为log n，所以求其高度的算法复杂度为O(log n)，而每次递归后问题的规模都会变成原来的一半，所以该算法求解节点个数的复杂度为 $O((logn)^2)$

02 方法二

1. 思路：

根据完全二叉树的定义，只有最底层可能没有填满，且对于没有填满的最底层，所有的节点都集中在该层的最左边。因此求解这个问题只需要知道最底层最靠右边的节点是哪一个。

设最底层为第d层，则该层最多包含 2^{d-1} 个节点。使用二分查找的思想解题如下：

1) 计算二叉树的高度d。因为完全二叉树最底层至少有一个节点（即最左边一定有节点），所以求解二叉树高度的方法是不断地令node=node→left，直至node为空，该步的时间复杂度是O(log n)。

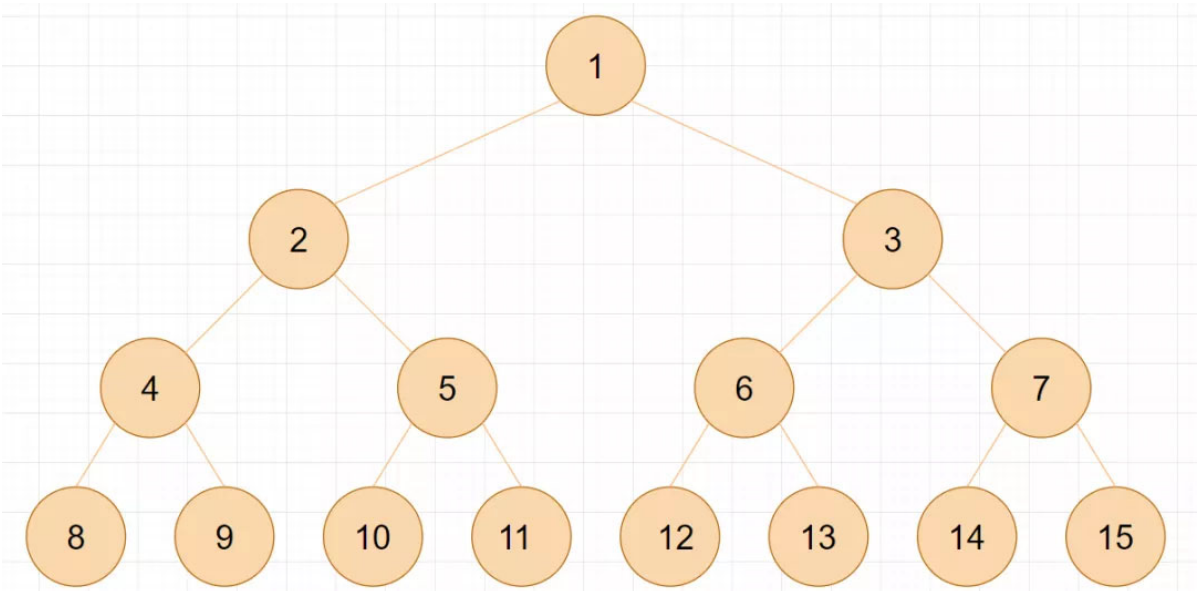

图7

2) 完全二叉树第层的节点数最多为 2^{d-1} 个。二叉树最底层的每一个节点都可以用一个d-1位的二进制数表示，从左到右依次为0~ $2^{d-1}-1$ ，通过从高位到低位观察这个二进制数，我们可以得到从根节点到某个最底层节点x的路径。对于这d-1位，0表示移动到左子节点，1表示移动到右子节点。当我们想要知道最底层某个节点是否存在时，首先我们得到这个节点在最底层是第几个节点，假设是k，然后通过将 k 转换为二进制我们可以得到对应的路径。例如在图7中d=4，对于序号8的节点，它是最底层第0个节点，可以表示为，表示从根节点1出发，向左走三次，对于序号为10的节点，它是最底层第2个节点，可以表示为010，表示从根节点1出发，向左、向右、向左。得到路径后顺着路径走就可以判断该路径对应的节点是否存在了。使用二分查找的方法时，先判断最底层中间的点是否存在，不存在则查看左半部分中点，存在则查看右半部分中点，直到找到最底层序号最大的点。

2. C语言代码：

```
#include
#include
#define NODE_NUM 100
char binary[NODE_NUM];
struct node{
    int value;
    struct node* left;
    struct node* right;
}list[NODE_NUM];


void init(){
    int i;
    for(i=0;binary[i]!=0;i++)
        list[i].value=binary[i]-'0';
    while(i<NODE_NUM){
        list[i].value=-1;
        i++;
    }
    for(i=0;i<NODE_NUM;i++){
        if((2*i+1)<NODE_NUM&&list[2*i+1].value==1){
            list[i].left=&list[2*i+1];
        }else{
            list[i].left=NULL;
        }
        if((2*i+2)<NODE_NUM&&list[2*i+2].value==1){
            list[i].right=&list[2*i+2];
        }else{
            list[i].right=NULL;
        }
    }
    return;
}


int height(struct node* node){
    int height=0;
    while(node->left!=NULL){
        node=node->left;
        height++;
```

```
    }
    return height;
}


int  exist(struct node* node,int height,int x){
    int bits=1<<(height-1);
    while(node!=NULL && bits>0){
        if((bits & x)==0){
            node=node->left;
        }else{
            node=node->right;
        }
        bits= bits>>1;
    }
    return node!=NULL;
}


int count(struct node* root){
    int whole_height,low,high,mid;
    struct node* node=root;
    whole_height=height(node);
    low=  1<< whole_height;
    high=(1<<(whole_height+1))-1;
    while(low<high){
        mid=(high-low+1)/2+low;
        node=root;
        if(exist(node,whole_height,mid)){
            low=mid;
        }else{
            high=mid-1;
        }
    }
    return low;
}


int main(){
    int i,num;
    printf("input the binary tree:\n");
    scanf("%s",&binary);
    init();
    num=count(&list[0]);
    printf("there is totally %d nodes\n",num);
    return 0;
}
```

3. 正确性：

完全二叉树除去最底层的所有层都是满的，因此问题可以看作求最底层最右侧的节点。使用二分法的思想，首先查看最底层正中间的节点是否存在，若存在则查看右半部分的正中间节点，否则查看左半部分的正中间节点，由此保证正确性和时间复杂度的要求。

4. 复杂度：

- 计算二叉树高度的时间复杂度是O(log n)
- 寻找最底层最右侧节点时，根据序号对应的路径寻找这个节点的时间复杂度为O(log n)，又由于使用二分法，故该步的时间复杂度为 $O((logn)^2)$

总的时间复杂度为 $O((logn)^2)$