

skywang12345

導航

[博客園](#)[首頁](#)[新隨筆](#)[聯繫](#)[訂閱](#)[管理](#)

2021年5月						
日	一	二	三	四	五	六
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

統計

隨筆- 278

文章- 0

評論- 1598

閱讀- 730萬

搜索

找找看

常用鏈接

[我的隨筆](#)[我的評論](#)[我的參與](#)

二叉查找樹(一)-圖文解析和C語言的實現

概要

本章先對二叉樹的相關理論知識進行介紹，然後給出C語言的詳細實現。關於二叉樹的學習，需要說明的是：它並不難，不僅不難，而且它非常簡單。初次接觸樹的時候，我也覺得它似乎很難；而之所產生這種感覺主要是由於二叉樹有一大堆陌生的概念、性質等內容。而當我真正的實現了二叉樹再回過頭來看它的相關概念和性質的時候，覺得原來它是如此的簡單！因此，建議在學習二叉樹的時候：先對二叉樹基本的概念、性質有個基本了解，遇到難懂的知識點，可以畫圖來幫助理解；在有個基本的概念之後，再親自動手實現二叉查找樹（這一點至關重要！）；最後再回過頭來總結一下二叉樹的理論知識時，你會發現——它的確很簡單！在代碼實踐中，我以“二叉查找樹，而不是單純的二叉樹”為例子進行說明，單純的二叉樹非常簡單，實際使用很少。況且掌握了二叉查找樹，二叉樹也就自然掌握了。

本篇實現的二叉查找樹是C語言版的，後面章節再分別給出C++和Java版本的實現。您可以根據自己熟悉的語言進行實踐學習！

請務必深刻理解、實踐並掌握“二叉查找樹”！它是後面學習AVL樹、伸展樹、紅黑樹等相關樹結構的基石！

目錄

1. 樹的介紹
2. 二叉樹的介紹
3. 二叉查找樹的C實現
4. 二叉查找樹的C測試程序

最新評論
我的標籤

隨筆分類 (275)

Android(7)
Android NDK編程(9)
Android 系統層(5)
Android 應用層(46)
Computer Culture(2)
Java(111)
Linux/Ubuntu(5)
UML(5)
Windows(1)
設計模式(1)
數據結構_算法(79)
索引(4)

最新評論

1. Re:Java多線程系列-- "JUC線程池"
02之線程池原理(一)

不錯,點個贊!

--真番茄雞蛋麵

2. Re:AVL樹(三)之Java的實現

贊一個

--Honglixi

3. Re:Java 集合系列11之Hashtable詳細介紹(源碼解析)和使用示例

@BiuBiuBong 遍歷的效率比迭代器更高...

--Resta

4. Re:紅黑樹(一)之原理和算法詳細介紹

瞎幾把寫

--我想我是鳥

5. Re:紅黑樹(四)之C++的實現

@漂泊於天邊的雲你把return 去掉即可, 和前面的二叉搜索樹一樣的嘛...

轉載請註明出處: <http://www.cnblogs.com/skywang12345/p/3576328.html>

更多内容: 數據結構與算法系列目錄

(01). 二叉查找樹(一)之圖文解析和C語言的實現

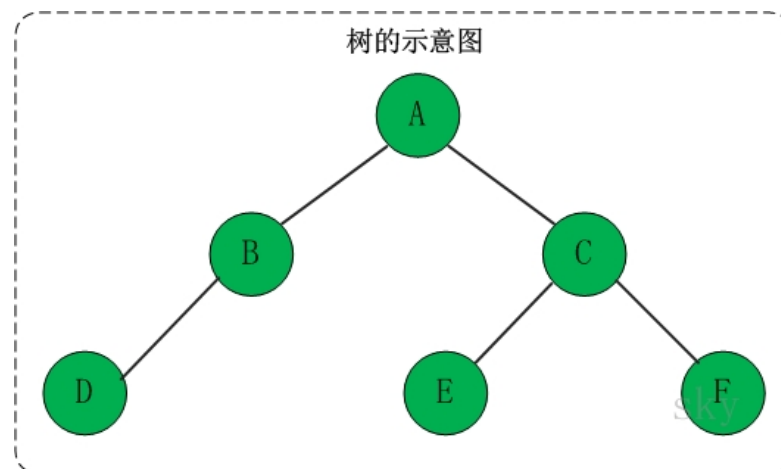
(02). 二叉查找樹(二)之C++的實現

(03). 二叉查找樹(三)之Java的實現

樹的介紹

1. 樹的定義

樹是一種數據結構, 它是由 n ($n \geq 1$) 個有限節點組成一個具有層次關係的集合。



把它叫做“樹”是因為它看起來像一棵倒掛的樹, 也就是說它是根朝上, 而葉朝下的。它具有以下的特點:

(01)每個節點有零個或多個子節點;

(02)沒有父節點的節點稱為根節點;

(03)每一個非根節點有且只有一個父節點;

(04)除了根節點外, 每個子節點可以分為多個不相交的子樹。

--one-rabbit

閱讀排行榜

1. 紅黑樹(一)之原理和算法詳細介紹(592232)
2. Java 集合系列10之HashMap詳細介紹(源碼解析)和使用示例(318616)
3. 數據結構與算法系列目錄(226148)
4. 圖的遍歷之深度優先搜索和廣度優先搜索(192170)
5. Java 集合系列12之TreeMap詳細介紹(源碼解析)和使用示例(185589)

2. 樹的基本術語

若一個結點有子樹，那麼該結點稱為子樹根的"雙親"，子樹的根是該結點的"孩子"。有相同雙親的結點互為"兄弟"。一個結點的所有子樹上的任何結點都是該結點的後裔。從根結點到某個結點的路徑上的所有結點都是該結點的祖先。

結點的度：結點擁有的子樹的數目。

葉子：度為零的結點。

分支結點：度不為零的結點。

樹的度：樹中結點的最度的度。

層次：根结点的层次为1，其余结点的层次等于该结点的双亲结点的层次加1。

树的高度：树中结点的最大层次。

无序树：如果树中结点的各子树之间的次序是不重要的，可以交换位置。

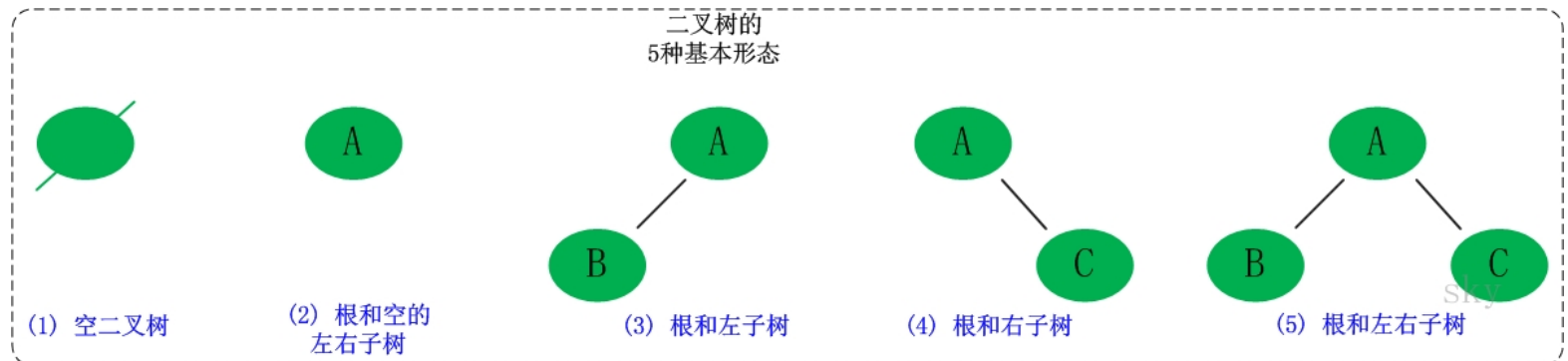
有序树：如果树中结点的各子树之间的次序是重要的，不可以交换位置。

森林：0个或多个不相交的树组成。对森林加上一个根，森林即成为树；删去根，树即成为森林。

二叉树的介绍

1. 二叉树的定义

二叉树是每个节点最多有两个子树的树结构。它有五种基本形态：**二叉树可以是空集；根可以有空的左子树或右子树；或者左、右子树皆为空。**



2. 二叉树的性质

二叉树有以下几个性质：TODO(上标和下标)

性质1：二叉树第*i*层上的结点数最多为 2^{i-1} ($i \geq 1$)。

性质2：深度为*k*的二叉树至多有 $2^k - 1$ 个结点($k \geq 1$)。

性质3：包含*n*个结点的二叉树的高度至少为 $\log_2(n+1)$ 。

性质4：在任意一棵二叉树中，若终端结点的个数为 n_0 ，度为2的结点数为 n_2 ，则 $n_0 = n_2 + 1$ 。

2.1 性质1：二叉树第*i*层上的结点数最多为 2^{i-1} ($i \geq 1$)

证明：下面用"数学归纳法"进行证明。

(01) 当*i* = 1时，第*i*层的节点数目为 $2^{i-1} = 2^0 = 1$ 。因为第1层上只有一个根结点，所以命题成立。

(02) 假设当*i* > 1，第*i*层的节点数目为 2^{i-1} 。这个是根据(01)推断出来的！

下面根据这个假设，推断出"第(*i* + 1)层的节点数目为 2^i "即可。

由于二叉树的每个结点至多有两个孩子，故"第(*i* + 1)层上的结点数" 最多是 "第*i*层的结点数目的2倍"。

即，第(*i* + 1)层上的结点数最大值 = $2 \times 2^{i-1} = 2^i$ 。

故假设成立，原命题得证！

2.2 性质2：深度为*k*的二叉树至多有 $2^k - 1$ 个结点($k \geq 1$)

证明：在具有相同深度的二叉树中，当每一层都含有最大结点数时，其树中结点数最多。利用"性质1"可知，深度为*k*的二叉树的结点数至多为：

$$2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$$

故原命题得证！

2.3 性质3：包含*n*个结点的二叉树的高度至少为 $\log_2(n+1)$

证明：根据"性质2"可知，高度为h的二叉树最多有 2^h-1 个结点。反之，对于包含n个结点的二叉树的高度至少为 $\log_2(n+1)$ 。

2.4 性质4：在任意一棵二叉树中，若终端结点的个数为 n_0 ，度为2的结点数为 n_2 ，则 $n_0=n_2+1$

证明：因为二叉树中所有结点的度数均不大于2，所以结点总数(记为n)="0度结点数(n_0)" + "1度结点数(n_1)" + "2度结点数(n_2)"。由此，得到等式一。

$$(等式一) \quad n = n_0 + n_1 + n_2$$

另一方面，0度结点没有孩子，1度结点有一个孩子，2度结点有两个孩子，故二叉树中孩子结点总数是： $n_1 + 2n_2$ 。此外，只有根不是任何结点的孩子。故二叉树中的结点总数又可表示为等式二。

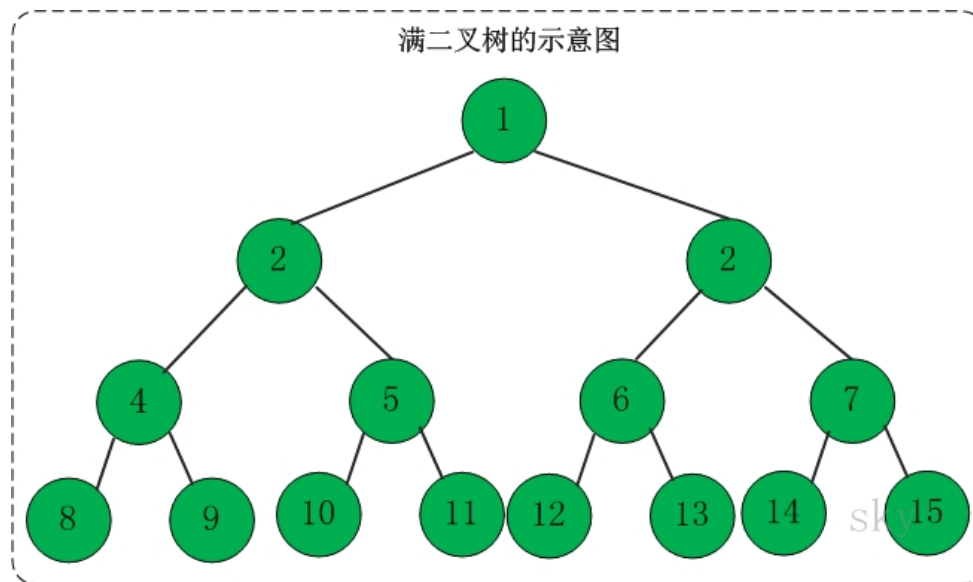
$$(等式二) \quad n = n_1 + 2n_2 + 1$$

由(等式一)和(等式二)计算得到： $n_0 = n_2 + 1$ 。原命题得证！

3. 满二叉树，完全二叉树和二叉查找树

3.1 满二叉树

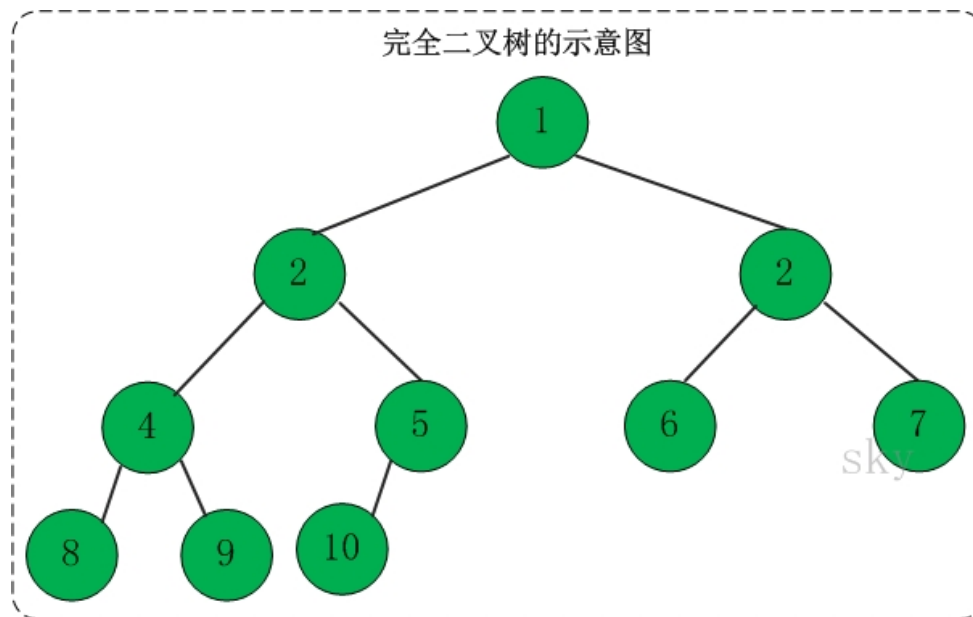
定义：高度为h，并且由 2^h-1 个结点的二叉树，被称为满二叉树。



3.2 完全二叉树

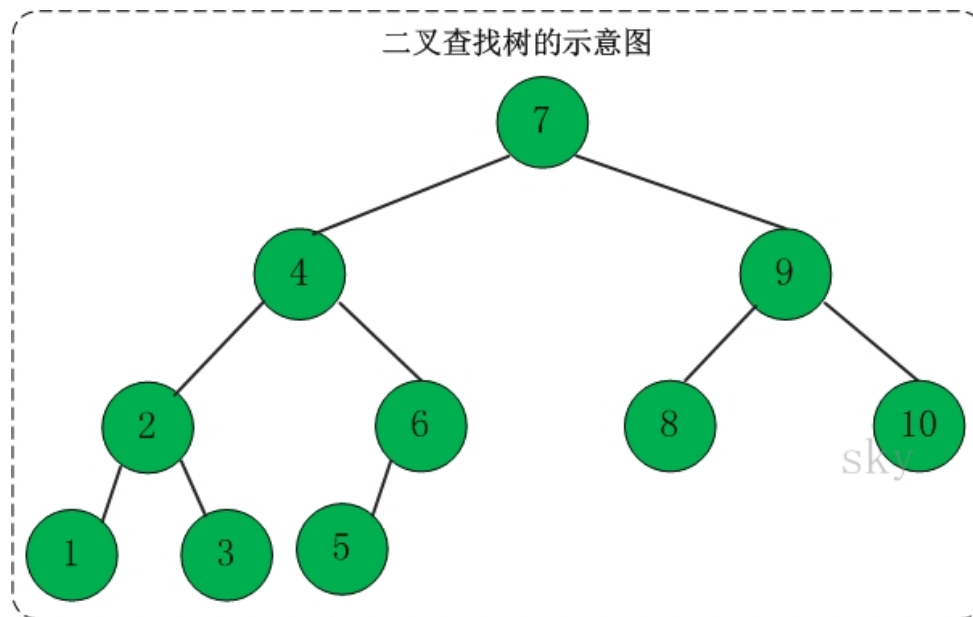
定义：一棵二叉树中，只有最下面两层结点的度可以小于2，并且最下一层的叶结点集中在靠左的若干位置上。这样的二叉树称为完全二叉树。

特点：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。显然，一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。



3.3 二叉查找树

定义：二叉查找树(Binary Search Tree)，又被称为二叉搜索树。设 x 为二叉查找树中的一个结点， x 节点包含关键字 key ，节点 x 的 key 值记为 $key[x]$ 。如果 y 是 x 的左子树中的一个结点，则 $key[y] \leq key[x]$ ；如果 y 是 x 的右子树的一个结点，则 $key[y] \geq key[x]$ 。



在二叉查找树中：

- (01) 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- (02) 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- (03) 任意节点的左、右子树也分别为二叉查找树。
- (04) 没有键值相等的节点（no duplicate nodes）。

在实际应用中，二叉查找树的使用比较多。下面，用C语言实现二叉查找树。

二叉查找树的C实现

1. 节点定义

1.1 节点定义



```
typedef int Type;

typedef struct BSTreeNode{
```



```
Type    key;                // 关键字 (键值)
struct BSTreeNode *left;    // 左孩子
struct BSTreeNode *right;   // 右孩子
struct BSTreeNode *parent;  // 父结点
}Node, *BSTree;
```



二叉查找树的节点包含的基本信息：

- (01) **key** -- 它是关键字，是用来对二叉查找树的节点进行排序的。
- (02) **left** -- 它指向当前节点的左孩子。
- (03) **right** -- 它指向当前节点的右孩子。
- (04) **parent** -- 它指向当前节点的父结点。

1.2 创建节点

创建节点的代码



```
static Node* create_bstree_node(Type key, Node *parent, Node *left, Node* right)
{
    Node* p;

    if ((p = (Node *)malloc(sizeof(Node))) == NULL)
        return NULL;
    p->key = key;
    p->left = left;
    p->right = right;
    p->parent = parent;

    return p;
}
```



2 遍历


这里讲解**前序遍历**、**中序遍历**、**后序遍历**3种方式。

2.1 前序遍历


若二叉树非空，则执行以下操作：

- (01) 访问根结点；
- (02) 先序遍历左子树；
- (03) 先序遍历右子树。

前序遍历代码



```
void preorder_bstree(BSTree tree)
{
    if(tree != NULL)
    {
        printf("%d ", tree->key);
        preorder_bstree(tree->left);
        preorder_bstree(tree->right);
    }
}
```




2.2 中序遍历

若二叉树非空，则执行以下操作：

- (01) 中序遍历左子树；
- (02) 访问根结点；
- (03) 中序遍历右子树。

中序遍历代码



```
void inorder_bstree(BSTree tree)
{

```

```
if(tree != NULL)
{
    inorder_bstree(tree->left);
    printf("%d ", tree->key);
    inorder_bstree(tree->right);
}
```



2.3 后序遍历

若二叉树非空，则执行以下操作：

- (01) 后序遍历左子树；
- (02) 后序遍历右子树；
- (03) 访问根结点。

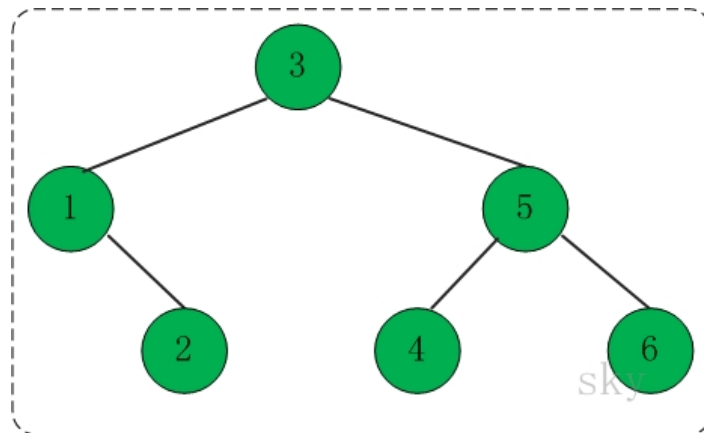
后序遍历代码



```
void postorder_bstree(BSTree tree)
{
    if(tree != NULL)
    {
        postorder_bstree(tree->left);
        postorder_bstree(tree->right);
        printf("%d ", tree->key);
    }
}
```



下面通过例子对这些遍历方式进行介绍。



对于上面的二叉树而言,

(01) 前序遍历结果: 3 1 2 5 4 6

(02) 中序遍历结果: 1 2 3 4 5 6

(03) 后序遍历结果: 2 1 4 6 5 3

3. 查找

递归版本的代码

```
Node* bstree_search(BSTree x, Type key)
{
    if (x==NULL || x->key==key)
        return x;

    if (key < x->key)
        return bstree_search(x->left, key);
    else
        return bstree_search(x->right, key);
}
```

非递归版本的代码



```
Node* iterative_bstree_search(BSTree x, Type key)
{
    while ((x!=NULL) && (x->key!=key))
    {
        if (key < x->key)
            x = x->left;
        else
            x = x->right;
    }

    return x;
}
```



4. 最大值和最小值

查找最大值的代碼



```
Node* bstree_maximum(BSTree tree)
{
    if (tree == NULL)
        return NULL;

    while(tree->right != NULL)
        tree = tree->right;

    return tree;
}
```




查找最小值的代碼



```
Node* bstree_minimum(BSTree tree)
{
    if (tree == NULL)
        return NULL;

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




5. 前驱和后继

节点的前驱：是该节点的左子树中的最大节点。

节点的后继：是该节点的右子树中的最小节点。

查找前驱节点的代码



```
Node* bstree_predecessor(Node *x)
{
    // 如果x存在左孩子，则"x的前驱结点"为 "以其左孩子为根的子树的最大结点"。
    if (x->left != NULL)
        return bstree_maximum(x->left);

    // 如果x没有左孩子，则x有以下两种可能：
    // (01) x是"一个右孩子"，则"x的前驱结点"为 "它的父结点"。
    // (01) x是"一个左孩子"，则查找"x的最低的父结点，并且该父结点要具有右孩子"，找到的这个"最低的父结点"就是"x的前驱结点"。
    Node* y = x->parent;
    while ((y!=NULL) && (x==y->left))
    {
        x = y;
        y = y->parent;
    }

    return y;
}
```



查找后继节点的代码



```
Node* bstree_successor(Node *x)
{
    // 如果x存在右孩子，则"x的后继结点"为 "以其右孩子为根的子树的最小结点"。
    if (x->right != NULL)
        return bstree_minimum(x->right);

    // 如果x没有右孩子。则x有以下两种可能：
    // (01) x是"一个左孩子"，则"x的后继结点"为 "它的父结点"。
    // (02) x是"一个右孩子"，则查找"x的最低的父结点，并且该父结点要具有左孩子"，找到的这个"最低的父结点"就是"x的后继结点"。
    Node* y = x->parent;
    while ((y!=NULL) && (x==y->right))
    {
        x = y;
        y = y->parent;
    }

    return y;
}
```



6. 插入

插入节点的代码



```
static Node* bstree_insert(BSTree tree, Node *z)
{
    Node *y = NULL;
    Node *x = tree;

    // 查找z的插入位置
```

```
while (x != NULL)
{
    y = x;
    if (z->key < x->key)
        x = x->left;
    else
        x = x->right;
}

z->parent = y;
if (y==NULL)
    tree = z;
else if (z->key < y->key)
    y->left = z;
else
    y->right = z;

return tree;
}

Node* insert_bstree(BSTree tree, Type key)
{
    Node *z;    // 新建结点


    // 如果新建结点失败，则返回。
    if ((z=create_bstree_node(key, NULL, NULL, NULL)) == NULL)
        return tree;

    return bstree_insert(tree, z);
}
```



bstree_insert(tree, z)是内部函数，它的作用是：将结点(z)插入到二叉树(tree)中，并返回插入节点后的根节点。
insert_bstree(tree, key)是对外接口，它的作用是：在树中新增节点，key是节点的值；并返回插入节点后的根节点。

注：本文实现的二叉查找树是允许插入相同键值的节点的！若用户不希望插入相同键值的节点，将bstree_insert()修改为以下代码即可。




```
static Node* bstree_insert(BSTree tree, Node *z)
{
    Node *y = NULL;
    Node *x = tree;

    // 查找z的插入位置
    while (x != NULL)
    {
        y = x;
        if (z->key < x->key)
            x = x->left;
        else if (z->key > x->key)
            x = x->right;
        else
        {
            free(z); // 释放之前分配的系统。
            return tree;
        }
    }

    z->parent = y;
    if (y==NULL)
        tree = z;
    else if (z->key < y->key)
        y->left = z;
    else
        y->right = z;

    return tree;
}
```



7. 删除

删除节点的代码



```
static Node* bstree_delete(BSTree tree, Node *z)
{
    Node *x=NULL;
    Node *y=NULL;

    if ((z->left == NULL) || (z->right == NULL) )
        y = z;
    else
        y = bstree_successor(z);

    if (y->left != NULL)
        x = y->left;
    else
        x = y->right;

    if (x != NULL)
        x->parent = y->parent;

    if (y->parent == NULL)
        tree = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;

    if (y != z)
        z->key = y->key;

    if (y!=NULL)
        free(y);

    return tree;
}

Node* delete_bstree(BSTree tree, Type key)
```

```

{
    Node *z, *node;

    if ((z = bstree_search(tree, key)) != NULL)
        tree = bstree_delete(tree, z);

    return tree;
}

```

`bstree_delete(tree, z)`是内部函数，它的作用是：删除二叉树(tree)中的节点(z)，并返回删除节点后的根节点。
`delete_bstree(tree, key)`是对外接口，它的作用是：在树中查找键值为key的节点，找到的话就删除该节点；并返回删除节点后的根节点。

8. 打印

打印二叉树的代码

```

void print_bstree(BSTree tree, Type key, int direction)
{
    if(tree != NULL)
    {
        if(direction==0)    // tree是根节点
            printf("%2d is root\n", tree->key);
        else                // tree是分支节点
            printf("%2d is %2d's %6s child\n", tree->key, key, direction==1?"right" : "left");

        print_bstree(tree->left, tree->key, -1);
        print_bstree(tree->right, tree->key, 1);
    }
}


```

`print_bstree(tree, key, direction)`的作用是打印整颗二叉树(tree)。其中，tree是二叉树节点，key是二叉树的键值，而direction表示该节点的类型：

direction为 0, 表示该节点是根节点;
direction为-1, 表示该节点是它的父结点的左孩子;
direction为 1, 表示该节点是它的父结点的右孩子。

9. 销毁二叉树


销毁二叉树的代码



```
void destroy_bstree(BSTree tree)
{
    if (tree==NULL)
        return ;



    if (tree->left != NULL)
        destroy_bstree(tree->left);
    if (tree->right != NULL)
        destroy_bstree(tree->right);

    free(tree);
}
```



完整的实现代码

二叉查找树的头文件(bstree.h)



```
1 #ifndef _BINARY_SEARCH_TREE_H_
2 #define _BINARY_SEARCH_TREE_H_
3
4 typedef int Type;
5
```

```
6 typedef struct BSTreeNode{
7     Type    key;                // 关键字 (键值)
8     struct BSTreeNode *left;    // 左孩子
9     struct BSTreeNode *right;   // 右孩子
10    struct BSTreeNode *parent;   // 父结点
11 }Node, *BSTree;
12
13 // 前序遍历"二叉树"
14 void preorder_bstree(BSTree tree);
15 // 中序遍历"二叉树"
16 void inorder_bstree(BSTree tree);
17 // 后序遍历"二叉树"
18 void postorder_bstree(BSTree tree);
19
20 // (递归实现) 查找"二叉树x"中键值为key的节点
21 Node* bstree_search(BSTree x, Type key);
22 // (非递归实现) 查找"二叉树x"中键值为key的节点
23 Node* iterative_bstree_search(BSTree x, Type key);
24
25 // 查找最小结点：返回tree为根结点的二叉树的最小结点。
26 Node* bstree_minimum(BSTree tree);
27 // 查找最大结点：返回tree为根结点的二叉树的最大结点。
28 Node* bstree_maximum(BSTree tree);
29
30 // 找结点 (x) 的后继结点。即· 查找"二叉树中数据值大于该结点"的"最小结点"。
31 Node* bstree_successor(Node *x);
32 // 找结点 (x) 的前驱结点。即· 查找"二叉树中数据值小于该结点"的"最大结点"。
33 Node* bstree_predecessor(Node *x);
34
35 // 将结点插入到二叉树中· 并返回根节点
36 Node* insert_bstree(BSTree tree, Type key);
37
38 // 删除结点 (key为节点的值) · 并返回根节点
39 Node* delete_bstree(BSTree tree, Type key);
40
41 // 销毁二叉树
42 void destroy_bstree(BSTree tree);
43
44 // 打印二叉树
45 void print_bstree(BSTree tree, Type key, int direction);
```

```
46
47 #endif
```



二叉查找树的实现文件(bstree.c)



```
1  /**
2   * 二叉搜索树(c语言): c语言实现的二叉搜索树。
3   *
4   * @author skywang
5   * @date 2013/11/07
6   */
7
8 #include <stdio.h>
9 #include <stdlib.h>
10 #include "bstree.h"
11
12
13 /*
14  * 前序遍历"二叉树"
15  */
16 void preorder_bstree(BSTree tree)
17 {
18     if(tree != NULL)
19     {
20         printf("%d ", tree->key);
21         preorder_bstree(tree->left);
22         preorder_bstree(tree->right);
23     }
24 }
25
26 /*
27  * 中序遍历"二叉树"
28  */
29 void inorder_bstree(BSTree tree)
30 {
31     if(tree != NULL)
```

```
32     {
33         inorder_bstree(tree->left);
34         printf("%d ", tree->key);
35         inorder_bstree(tree->right);
36     }
37 }
38
39 /*
40  * 后序遍历"二叉树"
41  */
42 void postorder_bstree(BSTree tree)
43 {
44     if(tree != NULL)
45     {
46         postorder_bstree(tree->left);
47         postorder_bstree(tree->right);
48         printf("%d ", tree->key);
49     }
50 }
51
52 /*
53  * (递归实现) 查找"二叉树x"中键值为key的节点
54  */
55 Node* bstree_search(BSTree x, Type key)
56 {
57     if (x==NULL || x->key==key)
58         return x;
59
60     if (key < x->key)
61         return bstree_search(x->left, key);
62     else
63         return bstree_search(x->right, key);
64 }
65
66 /*
67  * (非递归实现) 查找"二叉树x"中键值为key的节点
68  */
69 Node* iterative_bstree_search(BSTree x, Type key)
70 {
71     while ((x!=NULL) && (x->key!=key))
```

```
72     {
73         if (key < x->key)
74             x = x->left;
75         else
76             x = x->right;
77     }
78
79     return x;
80 }
81
82 /*
83  * 查找最小结点：返回tree为根结点的二叉树的最小结点。
84  */
85 Node* bstree_minimum(BSTree tree)
86 {
87     if (tree == NULL)
88         return NULL;
89
90     while(tree->left != NULL)
91         tree = tree->left;
92     return tree;
93 }
94
95 /*
96  * 查找最大结点：返回tree为根结点的二叉树的最大结点。
97  */
98 Node* bstree_maximum(BSTree tree)
99 {
100     if (tree == NULL)
101         return NULL;
102
103     while(tree->right != NULL)
104         tree = tree->right;
105     return tree;
106 }
107
108 /*
109  * 找结点(x)的后继结点。即，查找"二叉树中数据值大于该结点"的"最小结点"。
110  */
111 Node* bstree_successor(Node *x)
```



```
112 {
113     // 如果x存在右孩子，则"x的后继结点"为 "以其右孩子为根的子树的最小结点"。
114     if (x->right != NULL)
115         return bstree_minimum(x->right);
116
117     // 如果x没有右孩子。则x有以下两种可能：
118     // (01) x是"一个左孩子"，则"x的后继结点"为 "它的父结点"。
119     // (02) x是"一个右孩子"，则查找"x的最低的父结点，并且该父结点要具有左孩子"，找到的这个"最低的父结点"就是"x的后继结点"。
120     Node* y = x->parent;
121     while ((y!=NULL) && (x==y->right))
122     {
123         x = y;
124         y = y->parent;
125     }
126
127     return y;
128 }
129
130 /*
131  * 找结点(x)的前驱结点。即，查找"二叉树中数据值小于该结点的"最大结点"。
132  */
133 Node* bstree_predecessor(Node *x)
134 {
135     // 如果x存在左孩子，则"x的前驱结点"为 "以其左孩子为根的子树的最大结点"。
136     if (x->left != NULL)
137         return bstree_maximum(x->left);
138
139     // 如果x没有左孩子。则x有以下两种可能：
140     // (01) x是"一个右孩子"，则"x的前驱结点"为 "它的父结点"。
141     // (01) x是"一个左孩子"，则查找"x的最低的父结点，并且该父结点要具有右孩子"，找到的这个"最低的父结点"就是"x的前驱结点"。
142     Node* y = x->parent;
143     while ((y!=NULL) && (x==y->left))
144     {
145         x = y;
146         y = y->parent;
147     }
148
149     return y;
150 }
151
```

```
152 /*
153  * 创建并返回二叉树结点。
154  *
155  * 参数说明：
156  *     key 是键值。
157  *     parent 是父结点。
158  *     left 是左孩子。
159  *     right 是右孩子。
160  */
161 static Node* create_bstree_node(Type key, Node *parent, Node *left, Node* right)
162 {
163     Node* p;
164
165     if ((p = (Node *)malloc(sizeof(Node))) == NULL)
166         return NULL;
167     p->key = key;
168     p->left = left;
169     p->right = right;
170     p->parent = parent;
171
172     return p;
173 }
174
175 /*
176  * 将结点插入到二叉树中
177  *
178  * 参数说明：
179  *     tree 二叉树的根结点
180  *     z 插入的结点
181  * 返回值：
182  *     根节点
183  */
184 static Node* bstree_insert(BSTree tree, Node *z)
185 {
186     Node *y = NULL;
187     Node *x = tree;
188
189     // 查找z的插入位置
190     while (x != NULL)
191     {
```

```
192     y = x;
193     if (z->key < x->key)
194         x = x->left;
195     else
196         x = x->right;
197 }
198
199 z->parent = y;
200 if (y==NULL)
201     tree = z;
202 else if (z->key < y->key)
203     y->left = z;
204 else
205     y->right = z;
206
207 return tree;
208 }
209
210 /*
211  * 新建结点(key) · 并将其插入到二叉树中
212  *
213  * 参数说明：
214  *     tree 二叉树的根结点
215  *     key 插入结点的键值
216  * 返回值：
217  *     根节点
218  */
219 Node* insert_bstree(BSTree tree, Type key)
220 {
221     Node *z;    // 新建结点
222
223     // 如果新建结点失败，则返回。
224     if ((z=create_bstree_node(key, NULL, NULL, NULL)) == NULL)
225         return tree;
226
227     return bstree_insert(tree, z);
228 }
229
230 /*
231  * 删除结点(z) · 并返回根节点
```

```
232 *
233 * 参数说明 :
234 *     tree 二叉树的根结点
235 *     z 删除的结点
236 * 返回值 :
237 *     根节点
238 */
239 static Node* bstree_delete(BSTree tree, Node *z)
240 {
241     Node *x=NULL;
242     Node *y=NULL;
243
244     if ((z->left == NULL) || (z->right == NULL) )
245         y = z;
246     else
247         y = bstree_successor(z);
248
249     if (y->left != NULL)
250         x = y->left;
251     else
252         x = y->right;
253
254     if (x != NULL)
255         x->parent = y->parent;
256
257     if (y->parent == NULL)
258         tree = x;
259     else if (y == y->parent->left)
260         y->parent->left = x;
261     else
262         y->parent->right = x;
263
264     if (y != z)
265         z->key = y->key;
266
267     if (y!=NULL)
268         free(y);
269
270     return tree;
271 }
```

```
272
273 /*
274  * 删除结点 (key为节点的键值) · 并返回根节点
275  *
276  * 参数说明 :
277  *     tree 二叉树的根结点
278  *     z 删除的结点
279  * 返回值 :
280  *     根节点
281  */
282 Node* delete_bstree(BSTree tree, Type key)
283 {
284     Node *z, *node;
285
286     if ((z = bstree_search(tree, key)) != NULL)
287         tree = bstree_delete(tree, z);
288
289     return tree;
290 }
291
292 /*
293  * 销毁二叉树
294  */
295 void destroy_bstree(BSTree tree)
296 {
297     if (tree==NULL)
298         return ;
299
300     if (tree->left != NULL)
301         destroy_bstree(tree->left);
302     if (tree->right != NULL)
303         destroy_bstree(tree->right);
304
305     free(tree);
306 }
307
308 /*
309  * 打印"二叉树"
310  *
311  * tree      -- 二叉树的节点
```

```

312 * key          -- 节点的键值
313 * direction    -- 0·表示该节点是根节点;
314 *              -1·表示该节点是它的父结点的左孩子;
315 *              1·表示该节点是它的父结点的右孩子。
316 */
317 void print_bstree(BSTree tree, Type key, int direction)
318 {
319     if(tree != NULL)
320     {
321         if(direction==0)    // tree是根节点
322             printf("%2d is root\n", tree->key);
323         else                // tree是分支节点
324             printf("%2d is %2d's %6s child\n", tree->key, key, direction==1?"right" : "left");
325
326         print_bstree(tree->left, tree->key, -1);
327         print_bstree(tree->right, tree->key, 1);
328     }
329 }

```

二叉查找树的测试程序(btree_test.c)

```

1 /**
2  * C 语言：二叉查找树
3  *
4  * @author skywang
5  * @date 2013/11/07
6  */
7
8 #include <stdio.h>
9 #include "btree.h"
10
11 static int arr[] = {1, 5, 4, 3, 2, 6};
12 #define TBL_SIZE(a) ( (sizeof(a)) / (sizeof(a[0])) )
13
14 void main()
15 {

```

```
16     int i, ilen;
17     BSTree root=NULL;
18
19     printf("== 依次添加: ");
20     ilen = TBL_SIZE(arr);
21     for(i=0; i<ilen; i++)
22     {
23         printf("%d ", arr[i]);
24         root = insert_bstree(root, arr[i]);
25     }
26
27     printf("\n== 前序遍历: ");
28     preorder_bstree(root);
29
30     printf("\n== 中序遍历: ");
31     inorder_bstree(root);
32
33     printf("\n== 后序遍历: ");
34     postorder_bstree(root);
35     printf("\n");
36
37     printf("== 最小值: %d\n", bstree_minimum(root)->key);
38     printf("== 最大值: %d\n", bstree_maximum(root)->key);
39     printf("== 树的详细信息: \n");
40     print_bstree(root, root->key, 0);
41
42     printf("\n== 删除根节点: %d", arr[3]);
43     root = delete_bstree(root, arr[3]);
44
45     printf("\n== 中序遍历: ");
46     inorder_bstree(root);
47     printf("\n");
48
49     // 销毁二叉树
50     destroy_bstree(root);
51 }
```



二叉查找树的C测试程序

上面的btree_test.c是二叉查找树的测试程序，它的运行结果如下：



```
== 依次添加: 1 5 4 3 2 6
== 前序遍历: 1 5 4 3 2 6
== 中序遍历: 1 2 3 4 5 6
== 后序遍历: 2 3 4 6 5 1
== 最小值: 1
== 最大值: 6
== 树的详细信息:
1 is root
5 is 1's right child
4 is 5's left child
3 is 4's left child
2 is 3's left child
6 is 5's right child

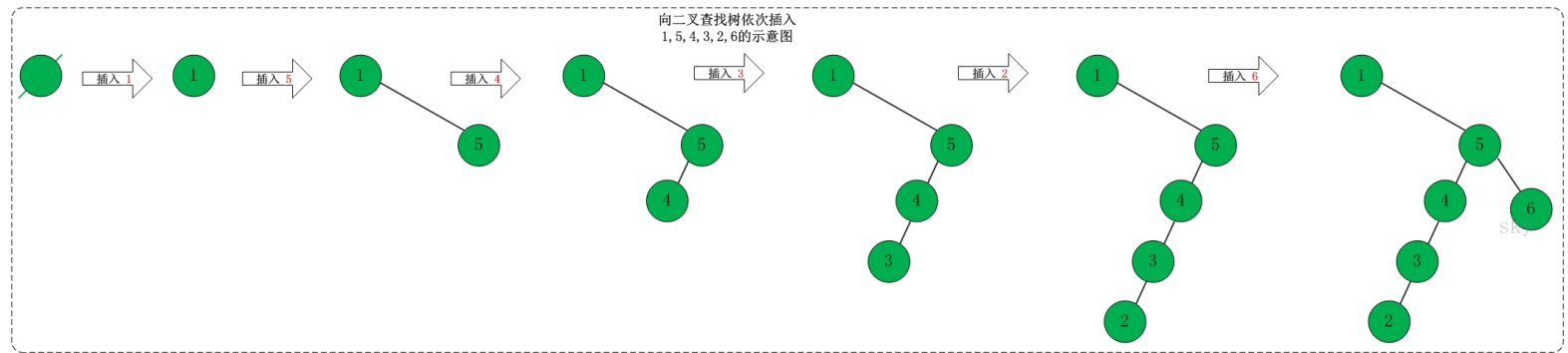
== 删除根节点: 3
== 中序遍历: 1 2 4 5 6
```



下面对测试程序的流程进行分析！

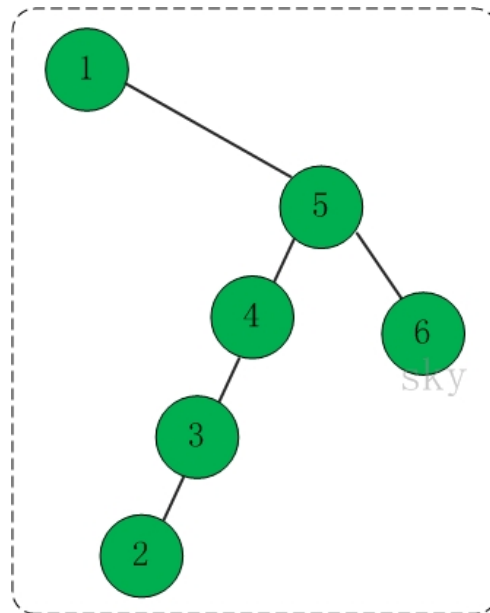
(01) 新建"二叉查找树"root。

(02) 向二叉查找树中依次插入1,5,4,3,2,6。如下图所示：



(03) 打印树的信息

插入1,5,4,3,2,6之后，得到的二叉查找树如下：



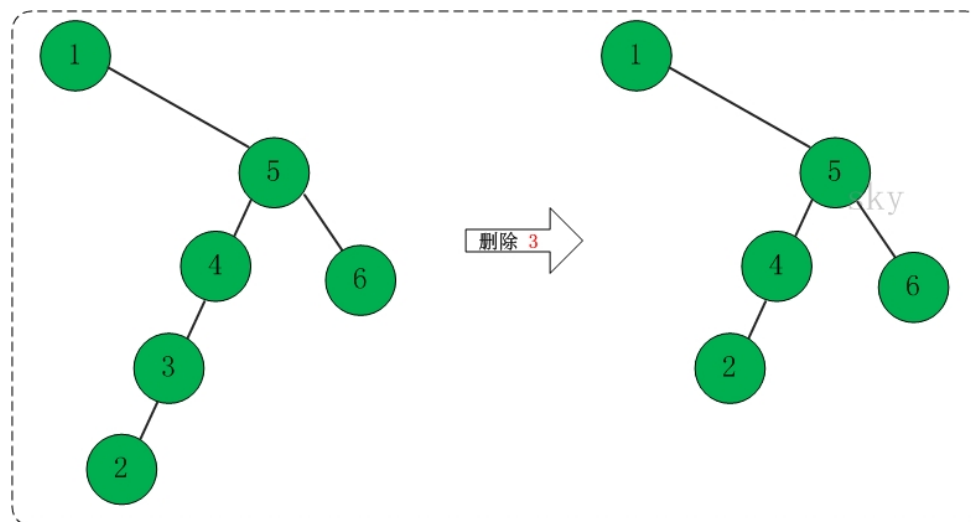
前序遍历结果: 1 5 4 3 2 6

中序遍历结果: 1 2 3 4 5 6

后序遍历结果: 2 3 4 6 5 1

最小值是1，而最大值是6。

(04) 删除节点3。如下图所示:



(05) 重新遍历该二叉查找树。

中序遍历结果: 1 2 4 5 6



生活的悲欢离合永远在地平线以外，而眺望是一种青春的姿态...

PS.文章是笔者分享的学习笔记，若你觉得可以、还行、过得去、甚至不太差的话，可以“推荐”一下的哦。就此谢过！

分类: 数据结构_算法

标签: 实现, C语言, 图解, 二叉树, 二叉查找树

好文要顶

关注我

收藏该文





如果天空不死

关注 - 9

粉丝 - 3371

[+加关注](#)[« 上一篇: 队列的图文解析 和 对应3种语言的实现\(C/C++/Java\)](#)[» 下一篇: 二叉查找树\(二\)之 C++的实现](#)posted on 2014-03-27 09:43 [如果天空不死](#) 阅读(57142) 评论(22) [编辑](#) [收藏](#)[刷新评论](#) [刷新页面](#) [返回顶部](#)登录后才能查看或发表评论, 立即 [登录](#) 或者 [逛逛](#) 博客园首页[【推荐】云上创新 2021阿里云峰会免费抢票, 期待您的到来!](#)[【推荐】大型组态、工控、仿真、CAD\GIS 50万行VC++源码免费下载!](#)[【推荐】100个HarmonyOS 2.0开发者Beta公测名额, 限时认领!](#)[【推荐】阿里云爆品销量榜单出炉, 精选爆款产品低至0.55折](#)[【推荐】限时秒杀! 国云大数据魔镜, 企业级云分析平台](#)

园子动态:

- [致园友们的一封检讨书: 都是我们的错](#)
- [数据库实例 CPU 100% 引发全站故障](#)
- [发起一个开源项目: 博客引擎 fluss](#)



最新新闻:

- [美国对亚马逊发起反垄断诉讼：阻止第三方低价出售商品](#)
 - [在这个豆瓣小组里 有20万人能帮你决定午饭吃什么](#)
 - [三巨头3nm、2nm大乱斗！Intel：我在哪儿？](#)
 - [暴雪免费大作《暗黑破坏神：不朽》国服将开测：你期待吗？](#)
 - [超级红月亮观赏攻略：5月26日现身、我国绝大多数地区肉眼可见](#)
- » [更多新闻...](#)

Powered by:

[博客園](#)

Copyright © 2021如果天空不死

Powered by .NET 5.0 on Kubernetes