二叉查找树 (Binary Search Tree)

张晓平

2016 年 11 月 8 日

1 定义

定义 1. 二叉查找树是一棵二叉树，可以是空树，否则满足如下性质：

1. 树中结点有一个唯一的关键字；
2. 如果有左子树，则左子树的所有关键字小于根的关键字；
3. 如果有右子树，则右子树的所有关键字大于根的关键字；
4. 左、右子树都是二叉查找树。

以上定义是递归的。由性质 2、3、4 可推出所有关键字都不相同。

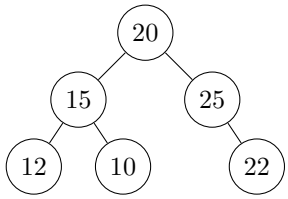

图 1: 不是 BST

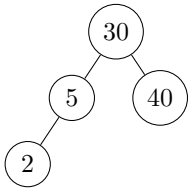

图 2: BST

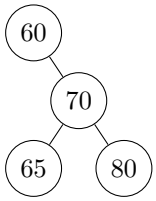

图 3: BST

注 1. 二叉查找树特别适合同时需要查找、插入和删除三种操作的应用。二叉查找树可以按关键字值操作，也可以按关键字序操作。如：可在查找树中查找或删除关键字为 *key* 的结点，也可以删除关键字排第 *5* 小的结点；还可以插入一个结点，同时获得该结点关键字的大小位置。

BST 结点描述

```
typedef int iType;
typedef struct {
  int key;
  iType item;
} ElemType;

typedef struct BSTNode {
  ElemType data;
  struct BSTNode * lchild, * rchild;
} BSTree, BSTNode;
```

2 查找

假设要查找关键字是 key 的结点。

2.1 递归实现

从根开始，若根是 NULL，树中无结点，查找失败；否则，比较 key 与根的关键字：

- 如果 key 等于根的关键字，查找成功，结束；
- 如果 key 小于根的关键字，因右子树的所有关键字不可能等于 key，故应查找根的左子树；
- 如果 key 大于根的关键字，应查找根的右子树。

BST 的查找（递归实现）

```
ElemType * search(BSTree root, int key)
{
  if (!root) return NULL;
  if (key == root->data.key) return &(root->data);
  if (key < root->data.key)
    return search(root->lchild, key);
  return search(root->rchild, key);
}
```

2.2 非递归实现

用 while 结构代替递归结构。

BST 的查找（非递归实现）

```
ElemType * iter_search(BSTree root, int key)
{
  while (root) {
    if (key == root->data.key) return &(root->data);
    if (key < root->data.key)
      root = root->lchild;
    else
      root = root->rchild;
  }
  return NULL;
}
```

3 插入

往 BST 中插入关键字为 key 的结点，首先应检查该结点是否已经出现在树中，故应先执行查找操作，若查找不成功，则在该位置执行插入操作。

如，在图 4中插入 80，首先在树中查找 80，查找不成功，最后找到结点 40，插入 80 使之成为结点 40 的右孩子。继续插入 35，查找不成功，最后仍找到结点 40，插入 35 使之成为结点 40 的左孩子。

在整个过程中，若查找不成功，则必须获取最后找到的结点，该功能可以对 iter_search 函数略加改动而得到。

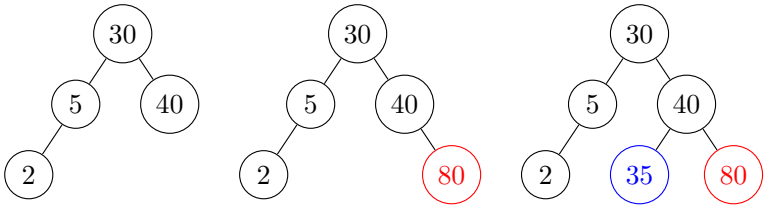

图 4: 插入操作

BST 的查找（修正）

```
BSTree mod_search(BSTree root, int key)
{
  if (!root) return NULL;
  while (root) {
    if (key == root->data.key) return NULL;
    if (key < root->data.key) {
      if (!root->lchild)
        return root;
      root = root->lchild;
    }
    else {
      if (!root->rchild)
        return root;
      root = root->rchild;
    }
  }
}
```

BST 的插入

```
void insert(BSTree * node, int key, iType item)
{
  BSTree ptr, tmp = mod_search(*node, key);
  if (tmp || !(*node)) {
    ptr = BSTree malloc(sizeof(*ptr));
```

```
    ptr->data.key  = key;
    ptr->data.item = item;
    ptr->lchild = ptr->rchild = NULL;
    if (*node) {
      if (key < tmp->data.key)
        tmp->lchild = ptr;
      else
        tmp->rchild = ptr;
    }
    else
      *node = ptr;
  }
}
```

4 删除

删除操作分为以下三种情形：

1. 删除叶子结点特别简单。如删除结点 35，只需把其父亲的左孩子域置为 NULL，然后释放被删除结点的空间即可。

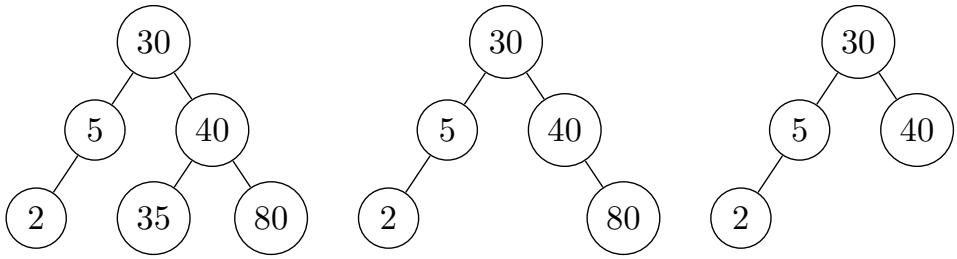

图 5: 删除叶子结点

例如，在图 5中删除 35，只要把其父亲结点 40 的左孩子域置为 NULL，然后释放被删除结点 35 的空间；接着删除 80，将其父亲结点 40 的右孩子域置为 NULL，释放结点 80 的空间。

2. 若待删结点不是叶子，但只有一个孩子，先把待删结点的空间释放，然后把它唯一的孩子放在删除结点原来的位置即可。

   如在图 6中删除结点 5，只要把其父亲结点 30 原本指向结点 5 的指针改为指向结点 2 即可。

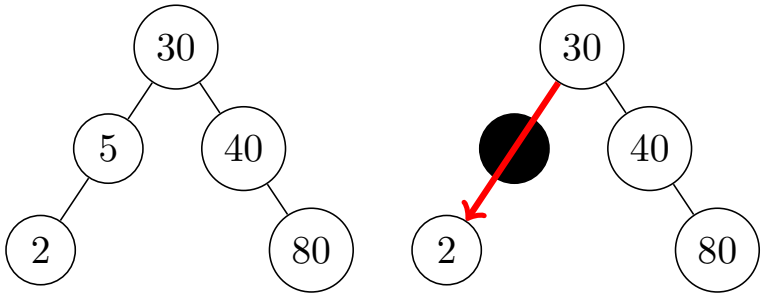

图 6: 删除只有一个孩子的结点

3. 若待删结点有两个孩子，先用左子树中关键字最大的孩子的 data 或右子树中关键字最小的孩子的 data 替换待删结点的 data，然后删除子树中的那个结点。

   例如，要删除图 7中的结点 30，一种做法是用左子树中最大关键字为 5 的结点替换，另一种做法是用右子树中最小关键字为 40 的结点替换。以第一种做法为例，将结点 5 的 data 存入根结点，然后删除根左边关键字为 5 的结点，最后将根的左孩子域指向结点 2。

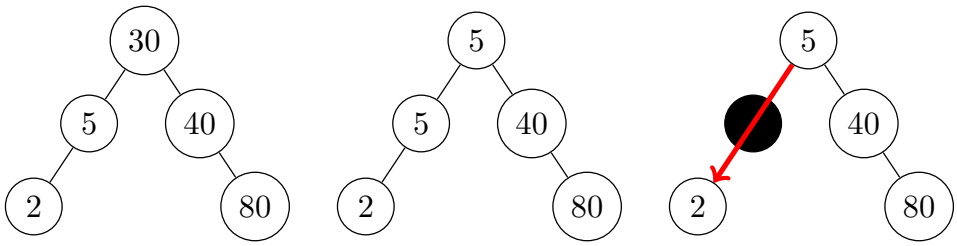

图 7: 删除有两个孩子的结点

习题 1. 编制程序，实现 *BST* 的删除操作。