Binary Search Tree (V): Deletion

CSCD 300 – Data Structures

Eastern Washington University

Goal: learn and implement BST's delete operation

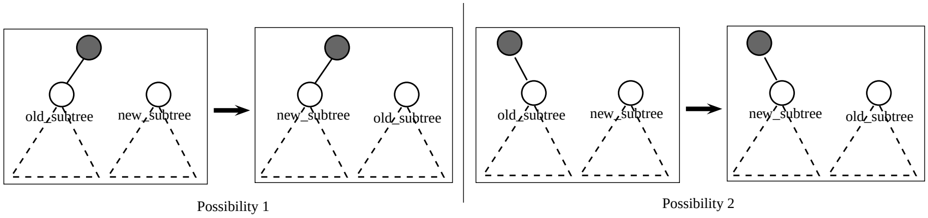
```
class BST{
   BST_Node root;   //the root of the BST

   BST(){ root = null; }  /* Constructor */

   BST_Node delete(int k){...} /* If the given key k already exists in the BST,
      delete the node that contains k from the BST and return the reference to
      the deleted node; otherwise, return null. */

   void delete(BST_Node node){...} /* A helper for the above delete(int).
      This helper method deletes the tree node referenced by ‘node’ */

   void transplant(BST_Node old_subtree, BST_Node new_subtree){...} /* A helper
      method that is to replace the subtree rooted on node ‘old_subtree’ by the
      subtree rooted on node ‘new_subtree’ */

   /* Other methods will follow here  */
}
```

Outline

Task

Delete the given key from the BST if the key exists, so that the resulting tree is still a BST.

Preparation: subtree transplant



```
/* Note: old_subtree cannot be null; new_subtree can be null. */

void BST::transplant(BST_Node old_subtree, BST_Node new_subtree){
   if(old_subtree.parent == null)
      root = new_subtree
   else if(old_subtree == old_subtree.parent.left)
      old_subtree.parent.left = new_subtree
   else
      old_subtree.parent.right = new_subtree

   if(new_subtree != null)
      new_subtree.parent = old_subtree.parent
}
```

The big picture of the algorithmic idea for deletion

Search for the node (denoted as z) that contains the to-be-deleted key.

1. If such node z does not exist, return `null`.
2. Else, delete node z from the BST and return the reference z:
    1. If z has no child: just delete z from the BST.
    2. If z has only one child: just link z's only child to z's parent. [1]
    3. If z has two children: (1) replace z by its successor, which is the minimum node in z's right subtree [2]. (2) delete z's successor [3].

[1] Of course, if z has no parent, z's only child becomes the global root.

[2] We can also choose to replace z by its predecessor, which is the maximum node in z's left subtree, but we often choose to replace z by its successor.

[3] z's successor has at most one child (why ?), so deleting z's successor is easy.

The delete operation

```
BST_Node BST::delete(int k) {
   BST_Node z = search(k);
   if(z != null)
      delete(z);
   return z;
}

void BST::delete(BST_Node z) {
   if(z.left == null and z.right == null) //z has no child
      transplant(z, null);
   else if(z.left == null) //z has only a right child
      transplant(z, z.right);
   else if(z.right == null) //z has only a left child
      transplant(z, z.left);
   else  //z has two children and is replaced by its successor
      y = Min(z.right);
      if(y.parent != z)
         transplant(y, y.right); y.right = z.right; y.right.parent = y;
      transplant(z, y); y.left = z.left; y.left.parent = y;
}
```

The time complexity of the delete operation

Theorem

The time complexity of the BST's delete operation takes $O(h)$ time.

- The first search operation takes $O(h)$ time.
- The actual delete mechanism after the search also takes $O(h)$ time.

How to maintain a “balanced” BST ?

It is possible that a sequence of `delete` operations can make the shape of a BST very skewed, leading to bad search performance.

There are more complicated “balanced” versions of BST, whose delete operation is more carefully designed and is more complicated, so that after any delete operation, the resulting BST is always guaranteed to be “balanced”. We will study such “balanced” BSTs in CSCD320.