Node Deletion in BSTs

March 30, 2020

Deleting Nodes In BSTs

Deletion of nodes is more complicated than the insertion of nodes in Binary Search Trees.

Why?

Simple Algorithm

A simple approach to deletion may be described as follows:

```
if the item is in the tree
    remove it
```

Deleting Nodes — Not That Simple

Unfortunately, things are not quite that simple. There are *three cases* to consider:

1. N is a leaf.
2. N has only one child.
3. N has two children.

Example Tree Construction

Recall the tree building code fragment:

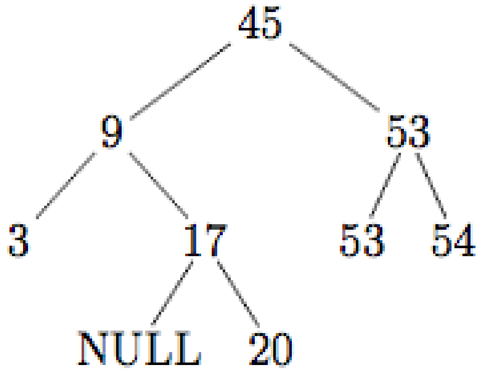
```
static DATA_TYPE A[] = { 45,  9, 53,  3,
                         17, 53, 20, 54 };
static int nA = sizeof(A)/sizeof(DATA_TYPE);

for( int i = 0 ; i < nA ; i++ )
   t1.AddNode( A[i] );
```

Example Tree

```
t1:
                54
        53
                53
45
                        20
                17
         9
                 3
```

Consider the idea of deleting nodes labeled: 3, 17, and 45.

Sample Tree



Case 1: Leaf Node

This case is easy:

To remove a leaf, all we need to do is set the pointer from its parent to `NULL`, and release (*delete*) the dynamically allocated memory.

The node with value 3 is clearly a leaf.

Case 2: One Child

This case is a bit more difficult, since there are two possibilities:

- N has only a left child.
- N has only a right child.

It can be shown that all we need to do is set the child to N's parent.

The node with value 17 falls into this category.

Case 3: Two Children

The last case is the most difficult. Since there are two children, N's parent cannot take them both, since it has one other child.

Case 3: Two Children (continued)

Another strategy:

1. Locate another node M that is easier to remove from the tree than N.
2. Copy the info that is in M to N (deleting orginal N).
3. Remove M from the tree.

Note that M can't be just any node—the binary search tree properties must be preserved!

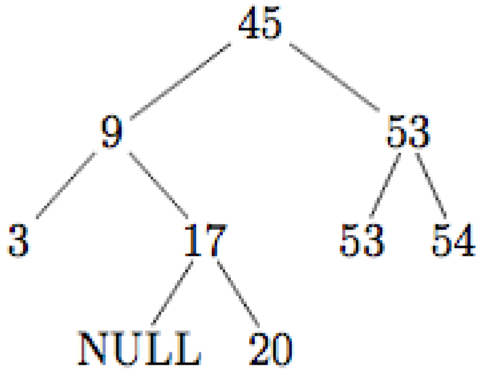
The node with value 45 (the root) clearly matches this case.

Node Deletion Algorithm

A more detailed description of the algorithm:

```
DeleteItem( TreePtr, key )
   if( key is in node N )
       DeleteNodeItem( N )
   else
       do nothing / error message

DeleteNodeItem( TreePtr N )
   if( N is a leaf )
       remove N from the tree
   else if( N has only one child (C) )
       if( N is the left child of its parent P )
           make C the left child of P
       else
           make C the right child of P
   else  //  N has two children
       find M, the node that is N's inorder successor
       copy info from M into N
       remove M from the tree.
```

Sample Tree



DeleteNode()


```
void DeleteNode( TreePtr& t, DATA_TYPE val )
{
   if( t == NULL ) {
      return;
   }
   else if( val == t->data ) {
      DeleteNodeItem( t );
   }
   else if( val < t->data ) {
      DeleteNode( t->left, val );
   }
   else {
      DeleteNode( t->right, val );
   }
}
```

Note: Recursion is used to traverse the tree.

DeleteNodeItem()


```
void DeleteNodeItem( TreePtr& t )
{
    TreePtr delPtr;

    if( IsLeaf(t) ) {
       delete t;
       t = NULL;
    } else if( t->left == NULL ) {
       delPtr = t;
       t = t->right;
       delPtr->right = NULL;
       delete delPtr;
    } else if( t->right == NULL ) {
       delPtr = t;
       t = t->left;
       delPtr->left = NULL;
       delete delPtr;
    } else {
       DATA_TYPE  replacementItem;
       ProcessLeftMost( t->right, replacementItem );
       t->data = replacementItem;
    }
```

ProcessLeftMost()


```
void ProcessLeftMost( TreePtr& t, DATA_TYPE& theItem )
{
   if( t->left != NULL )
      ProcessLeftMost( t->left, theItem );
   else
   {
      theItem = t->data;

      TreePtr delPtr = t;
      t = t->right;
      delPtr->right = NULL;
      delete  delPtr;
   }
}
```

Node Deletion in Modula-2

Unfortunately, removal of an element is not generally as simple as insertion. It is straightforward if the element to be deleted is a terminal node (leaf) or one with a single descendant. The difficulty lies in removing an element with two descendants, for we cannot point in two directions with a single pointer.

In this situation, the deleted element is to be replaced by either the rightmost element of its left subtree or by the leftmost node of its right subtree, both of which have at most one descendant. The details are shown in the recursive procedure `delete()`.

Node Deletion in Modula-2


```
PROCEDURE delete( x : INTEGER; VAR p : TreePtr );
  VAR  q : TreePtr
    PROCEDURE del( VAR r : TreePtr );
    BEGIN
        IF r^.right # NIL  THEN
            del( r^.right )
        ELSE
            q^.key := r^.key;
            q := r;
            r := r^.left;
        END
    END del;
BEGIN  (* delete *)
    IF p = NIL THEN ; (* word is not in tree *)
    ELSIF x < p^.key THEN delete( x, p^.left )
    ELSIF x > p^.key THEN delete( x, p^.right )
    ELSE  (* delete p^  *)
       q := p;
       IF    q^.right = NIL THEN p := q^.left
       ELSIF q^.left  = NIL THEN p := q^.right
       ELSE del(q^.left)
    END;
```

A few comments on Modula-2

Niklaus Wirth created Pascal, Modula, and Oberon.

Note the use of `BEGIN` and `END` for grouping statements. [Ada uses this convention also.]

Note the use of `^.` (instead of `->`) for accessing fields in the pointer variables.

Note: In Modula-2, procedures/functions may be defined *inside* another procedure/function.

Why is this useful?

Let's look at the Modula-2 procedures translated into C++.

delete() in C++

```
void delete( TreePtr &p, int x )
{
    TreePtr  q;

    if( p == NULL )
        ;   /* item is not in tree */
    else if( x < p->key )
       delete( p->left, x  );
    else if( x > p->key )
       delete( p->right, x );
    else    /* delete p   */
    {
       q = p;
       if(      q->right == NULL)  p = q->left;
       else if( q->left  == NULL)  p = q->right;
       else del( q, q->left );
    }
}
```

del() in C++

```
void del( TreePtr& q, TreePtr& r )
{
   if( r->right != NULL )
       del( q, r->right );
   else
   {
       q->key = r->key;
       q = r;
       r = r->left;
   }
}
```

Difference?

What's the difference between the two algorithms?

- The first algorithm promotes the *inorder successor*.
- The second algorithm promotes the *inorder predecessor*.