CMPT 115: Principles of Computer Science

Binary Search Tree ADT Implementation

Department of Computer Science
University of Saskatchewan

March 21–25, 2016

Today's Goal

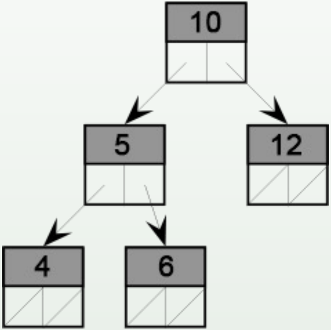
- To implement a Binary Search Tree ADT in C++ from the pseudocode given in the lectures.
- We will complete BSTree.cc in the lab

Exercises (to hand in with Assignment 9)

- `BSTree.cc`

Binary Search Trees

In a *binary search tree (BST)*, the left subtree contains key values less than the root and the right subtree contains key values greater than the root. It looks like this:



Lab Objective: build and test a binary search tree

In this lab you'll write a program that does the following:

1. Create a binary search tree.
2. Repeatedly:
    - ask the user for an integer;
    - put the integer into the binary search tree;

   Keep repeating until the user types -999.
3. Display the in-order traversal on the tree.
4. Search for two targets: one is in the tree, the other is not.
5. Optional: Delete one node from the tree.
6. Display the in-order traversal on the tree after deleting that node.
7. Destroy the tree.

Binary Search Tree Data Structure

- The BST data structure is slightly different than the Tree data structure that we have seen.
- It is more similar to the List data structure (two record types).
- In order to make the ADT more general, we can do this first: `typedef int Element;`
- Then the data structure is defined as:

```
Tree
  refToTreeNode root
end Tree

TreeNode
   Element data
   refToTreeNode left
   refToTreeNode right
end TreeNode
```

Interface: BSTree.h

ACTIVITY: Download `BSTree.h` from moodle.

We are going to implement the operations listed in this file:

- `Tree *CreateBST(); // create an empty tree and return its reference`
- `void DestroyBST(Tree*); // destroy a given tree, and return all memory to the heap`
- `bool SearchBST(const Tree*, const Element);`
- `void InsertBST(Tree *, const Element);`
- `void PrintInOrder(Node *);`

ACTIVITY:

- Implement operations in `BSTree.cc`.
- Create a test driver to test the functionality in `testBSTree.cc`.

insertNodeBST

Algorithm *insertNodeBST*(*rNode*, *nData*)
Inserts the new data into the TreeNodes of a binary search tree starting at node rNode
Pre: rNode is a reference to a TreeNode in rTree
nData is the data element to insert into a new node
Post: a new node is created with nData as data and inserted into the tree at rNode if the data is not already in tree, and rTree will remain a binary search tree.
Return: returns the root of the modified subtree.

if (*rNode* == NULL)
  refToNode rNew ← **allocate new** *Node*
  rNew ⤖ *data* ← *nData*
  rNew ⤖ *leftSubtree* ← NULL
  rNew ⤖ *rightSubtree* ← NULL
  return *rNew*
else if (*nData* == key of *rNode* ⤖ *data*)
  return *rNode* // already there, no change!
else if (*nData* < key of *rNode* ⤖ *data*)
  rNode ⤖ *leftSubtree* ← *insertNodeBST*(*rNode* ⤖ *leftSubtree*, *nData*)
  return *rNode*
else if (*nData* > key of *rNode* ⤖ *data*)
  rNode ⤖ *rightSubtree* ← *insertNodeBST*(*rNode* ⤖ *rightSubtree*, *nData*)
  return *rNode*
end if

insertBST

Algorithm *insertBST*(*rTree*, *nData*)
Insert a node into a binary search tree
Pre: rTree is a reference to a binary search tree
nData is the data element to insert into a new node
Post: a new node with nData as data and inserted a new node in the tree such that rTree remains a binary search tree.
Return: Nothing

$rTree \twoheadrightarrow \text{root} \leftarrow insertNodeBST(rTree \twoheadrightarrow root,\ nData)$

`insertNodeBST` is a helper function, only used internally.

Inorder Traversal for Binary Search Trees

We simply call the algorithm for TreeNodes.

```
Algorithm Inorder(rTree)
Processes each node of rTree exactly once using a inorder
    traversal using the Process() algorithm.
Pre:  rTree is a Tree.

   InorderTNodes( rTree ⤖ root)
```

This is the operation the applications programmer calls. The function InorderTNodes is only used internally.

searchNodeBST

Algorithm *searchNodeBST*(*rNode*, *target*, rDataOut)
Searches a binary search tree starting at node rNode for target
Pre: rNode :: TreeNode
target is the key of the data element to search for
rDataOut :: refToElement
Post: *rDataOut contains data of element with target as data if found
Return: true if found, false otherwise

if (rNode = NULL)
  return false
else if ($target <$ key of $rNode \twoheadrightarrow data$)
  return $searchNodeBST(rNode \twoheadrightarrow leftSubtree, target, rDataOut)$
else if ($target >$ key of $rNode \twoheadrightarrow data$)
  return $searchNodeBST(rNode \twoheadrightarrow rightSubtree, target, rDataOut)$
else
  $*rDataOut \leftarrow rNode \twoheadrightarrow data$
  return *true*
end if

searchBST

Algorithm *searchBST*(*rTree*, *target*, *rDataOut*)
Searches a binary search tree starting at node rRoot for target
Pre: rTree is a reference to a BST
target is the key of the data element to search for
rDataOut is a reference of type Element
Post: *rDataOut contains data of element with target as data if found
Return: true if round, false otherwise

return *searchNodeBST*(*rTree* *⇒ *root*, *target*, *rDataOut*)

`searchNodeBST` is a helper function, only used internally.

Study: deleteBST

Algorithm *deleteBST*(*rTree*, *nData*, *rDataOut*)
Delete a node from a binary search tree
Pre: rTree is a reference to a binary search tree
nData is the data element of the node to be deleted
rDataOut is a reference of type Element
Post: if there is a node containing nData, it is removed from tree, and its data place in *rDataOut
Return: true if node deleted, false otherwise.

```
// Search for node to delete
refToNode rNode ← rTree ⤃ root
refToNode rParent ← NULL
refToNode rNew
refToNode rBiggest
while (rNode != NULL AND key of rNode != nData)
    rParent ← rNode
    if (nData < key of rNode ⤃ data ) rNode ← rNode ⤃ leftSubtree
    else rNode ← rNode ⤃ rightSubtree
    end if
end while

// continued next slide
```

Optional: deleteBST

```
// Continued from previous slide

//rNode to be deleted; rNew will replace rNode as child of rParent
if (rNode == NULL) // the data was not in the tree
  return false
else // the data was found in the tree
   if (rNode ⤖ leftSubtree = NULL)        //case 1 or 2
      rNew ← rNode ⤖ rightSubtree         //rNew will replace rNode
   else if (rNode ⤖ rightSubtree = NULL)  //case 3
      rNew ← rNode ⤖ leftSubtree          //rNew will replace rNode
   else   //case 4
          rBiggest ← rNode ⤖ leftSubtree
          //find biggest in left subtree
          while (rBiggest ⤖ rightSubtree != NULL)
             rBiggest ← rBiggest ⤖ rightSubtree
          end while
          //attach right subtree at correct position
          rBiggest ⤖ rightSubtree ← rNode ⤖ rightSubtree
          rNew ← rNode ⤖ leftSubtree       //rNew will replace rNode
   end if

   // continued next slide
```

Study: deleteBST

```
// Continued from previous slide
// just tidy up now

    //if deleted node is root, make new root
    if (rNode = rTree ⤔ root)
        rTree ⤔ root ← rNew
    //if deleted node is left child, make rNew left child
    else if (rParent ⤔ leftSubtree = rNode)
        rParent ⤔ leftSubtree ← rNew
    //if deleted node was right child, make rNew right child
    else
        rParent ⤔ rightSubtree ← rNew
    end if
    (*rDataOut) ← rNode ⤔ data      //return deleted data
    delete(rNode)      //free deleted node
    return true
end if
```