Programming Techniques project, Spring 2017
Binary Search Tree

Vasilescu Rodica - Mihaela

May 12, 2017

First Year
Group: C.E.N 1.2B
Where: Room S6N
Web: Programming Techniques

1 Introduction

The goal of my assignment is to develop a software for experimenting with algorithms. The project is focused on the development of skills for good programming of basic algorithms, covering coding, design, documentation and presentation of results. At the end of my project I must produce a set of deliverables including: a technical report, source code and experimental data.

2 Problem statements

A library for binary search trees (BST)
The library should provide operations for: creating an empty tree, inserting a node into a tree, deleting a node and at least two different traversal strategies the traversal functions must accept a function which will be passed the current element.

3 General aspects

In computer science, binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of container: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

4 Definition

A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted left and right. The tree additionally satisfies the binary search tree property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree.[1]:287 (The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another. Leaves are commonly represented by a special leaf or nil symbol, a NULL pointer, etc.)

5 Pseudocode Algorithms

5.1 Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees. Eventually, we will reach an external node and add the new key-value pair as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

```
INSERT ($value$, $root$)
1. if $root$ = NULL then
2.     $info[root] \leftarrow value$
3. else  if $value < info[root]$ then
4.          $\triangleright$ INSERT ($value$ , $left[root]$)
5. else $\triangleright$ INSERT ($value$ , $right[root]$)
```

Figure 1: Insertion of nodes in the binary tree

5.2 Searching

Searching a binary search tree for a specific key can be programmed recursively or iteratively. We begin by examining the root node. If the tree is null, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is null. If the searched key is not found after a null subtree is reached, then the key is not present in the tree.

```
SEARCH ($value$, $root$)
1. if $root$ != NULL then
2.     if $info[root] = value$ then
3.         return $info[root]$
4.     else  if $value < info[left]$ then
5.             return ⊳ SEARCH ( $value$ , $left[root]$ )
6.     else  return ⊳ SEARCH ( $value$ , $rightt[root]$ )
```

Figure 2: Searching of nodes in the binary tree

5.3 Deletion

When removing a node from a binary search tree it is mandatory to maintain the in-order sequence of the nodes. There are many possibilities to do this. However, the following method guarantees that the heights of the subject subtrees are changed by at most one. There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted C. Do not delete C. Instead, choose either its in-order predecessor node or its in-order successor node as replacement node Temp.

In all cases, when D happens to be the root, make the replacement node root again. Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have only one or no child at all. Delete it according to one of the two simpler cases above.

```
DELETE($del$ , $root$)
1. $c \leftarrow root$
2. while $del$ != $info[c]$ do
3.         $p \leftarrow c$
4.         if $del < info[c]$ then
5.             $c \leftarrow left[c]$
6.         else $c \leftarrow right[c]$
7. if  $left[c] = NULL$ and $right[c] = NULL$ then
8.     if $info[c] < info[p]$ then
9.         $left[p] \leftarrow NULL$
10.    else $right[p] \leftarrow NULL$
11. else if $right[c] = NULL$ then
12.         if $info[c] < info[p]$ then
13.             $left[p] \leftarrow left[c]$
14.         else $right[p] \leftarrow left[c]$
15. else if $left[c] = NULL$ then
16.         if $info[c] < info[p]$ then
17.             $left[p] \leftarrow right[c]$
18.         else $right[p] \leftarrow right[c]$
19. else $temp \leftarrow right[c]$
20.        while $left[temp]$ != NULL do
21.                $temp = left[temp]$
22.        $aux = info[temp]$
23.        $\triangleright$ DELETE (aux , root)
24.        $info[c] = aux$
```

Figure 3: Deletion of nodes in the arbory tree

5.4 Preorder Traversal

Algorithm Preorder

1 Check if the current node is empty / null.

2 Display the data part of the root (or current node).

3 Traverse the left subtree by recursively calling the pre-order function.

4 Traverse the right subtree by recursively calling the pre-order function.

Uses of Preorder
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.

```
PREORDER ( $root$)
1. if $root$ != NULL then
2.    write $info[root]$
3.    if $left[root]$ != NULL then
4.       ▷ PREORDER $left[root]$
5.    else  if $right[root]$ != NULL then
6.             ▷ PREORDER $right[root]$
```

Figure 4: Preorder Traversal

5.5 Inorder Traversal

Algorithm Inorder

1 Check if the current node is empty / null.

2 Traverse the left subtree by recursively calling the in-order function.

3 Display the data part of the root (or current node).

4 Traverse the right subtree by recursively calling the in-order function.

Uses of Inorder
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder itraversal s reversed, can be used.

```
INORDER ( $root$)
1. if $root$ != NULL then
2.    if $left[root]$ != NULL then
3.       ▷ INORDER $left[root]$
4.    write $info[root]$
5.    else  if $right[root]$ != NULL then
6.             ▷ INORDER $right[root]$
```

Figure 5: Inorder Traversal

5.6 Postorder

Algorithm Postorder

1 Check if the current node is empty / null.

2 Traverse the left subtree by recursively calling the post-order function.

3 Traverse the right subtree by recursively calling the post-order function.

4 Display the data part of the root (or current node).

Uses of Postorder
Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree.

```
POSTORDER ( $root$)
1. if $root$ != NULL then
2.    if $left[root]$ != NULL then
3.       ▷ POSTORDER $left[root]$
4.    else  if $right[root]$ != NULL then
5.             ▷ POSTORDER $right[root]$
6.    write $info[root]$
```

Figure 6: Postorder Traversal

References

[1] https://en.wikipedia.org/wiki/Binary_search_tree *Binary Search Trees.*

[2] http://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/ *Traversal preorder, inorder, postorder.*

[3] LaTeXproject site, http://latex-project.org/