COSC1285/2123: Algorithms & Analysis
Laboratory 5

Topic

Binary Search Trees.

Objective

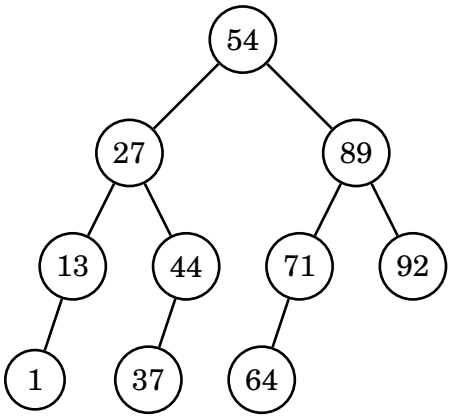
Students who complete this lab should learn how to implement binary search trees in Java and gain greater understanding of them.

Binary Search Tree

A binary search tree (BST) is a data structure that facilitates search using a binary tree to store data. A search for an element X in a BST is successful if there exists a node in the binary tree corresponding to the value X. Values in a BST are stored to satisfy the following property: Each internal node x stores an element such that the elements stored in the left subtree of x are less than x and elements stored in the right subtree of x are greater than x. This is the called **binary search tree property**.

Traversal

Tree-traversal refers to the process of recursively visiting (examining and/or updating) each node in a tree data structure, exactly once, in a *systematic* way. Such traversals are classified by the **order** in which the nodes are visited. Starting at the **root** of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (referred to as "visiting" the node), traversing to the left child node, and traversing to the right child node. Although we didn't go through this in lectures, it is interesting to know about traversals. Consider the following traversal approaches and example:



Pre: 54,27,13,1,44,37,89,71,64,92
In: 1,13,27,37,44,54,64,71,89,92
Post: 1,13,37,44,27,64,71,92,89,54

Traversal methods

In **preorder** traversal, the root is visited **before** the left and right subtrees are visited:

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

In **inorder** traversal, the root is visited **after** the left subtree and **before** the right subtree is visited:

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

In **postorder** traversal, the root is visited **after** the left and right subtrees are visited:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

Implementation

Your task in this laboratory is to experiment with the graph traversals and implement a number of different operations. Download the provided source code `BSTDemo.java` and `BST.java` from Blackboard.

The following methods are provided in BST.java:

- `insert()` — insert an element into the BST.
- `inorder()` — traverse the BST in **inorder**, printing out the element of nodes as we visit them
- `preorder()` — traverse the BST in **preorder**, printing out the element of nodes as we visit them
- `postorder()` — traverse the BST in **postorder**, printing out the element of nodes as we visit them

You are to implement the following methods for a binary search tree in BST.java:

- `search()` — search for an element in the BST. Hint: Use insert() as your initial guide on how to implement search.
- `min()` — return the minimum element in the BST. Hint: Remember the semantics of a binary search tree.

- `max()` — return the maximum element in the BST. Hint: Remember the semantics of a binary search tree.
- `height()` — calculate the height of the BST. Hint: height of a node = height of parent + 1.

You can compile the finished program using the command:

```
javac BSTDemo.java BST.java
```

Running your code

The provided skeleton code allows you to dynamically modify the BST using a "mini" command shell:

```
$ java BSTDemo
insert 9
insert 4
insert 12
insert 7
insert 3
insert 6
insert 18
insert 25
print_ascii
      9
     / \
    /   \
   4     12
  / \      \
 3   7     18
    /        \
   6          25
end
```

Testing your code

The provided skeleton code reads data from `stdin` and executes operations accordingly. Two test files `test01.in` and `test02.in` the corresponding expected outputs `test01.exp` and `test02.exp` are provided for you to test your program.

First run your program reading the inputs from stdin and write the output to file using the following command:

```
java BSTDemo < test01.in > test01.out
```

Then compare your output to the expected output:

```
diff test01.exp test01.out
```

Challenges

- All recursive functions can be unwound. How would you implement an iterative tree traversal?