Week 03 Lab Exercise

Binary Search Trees

Objectives

- To explore the implementation of binary search trees
- To get some practice with binary search trees
- To implement a level-order traversal
- To perform complexity analysis on some binary search tree functions

---

Admin

Marks	5 (see Assessment section for more details)
Demo	in the Week 03 or Week 04 lab
Submit	see the Submission section
Deadline	submit by 5pm Sunday of Week 03
Late penalty	0.5 marks per day

Background

In lectures, we examined a binary search tree data type `BSTree` and implemented some operations on it. In this week's lab, we will add some additional operations to that ADT.

Recall that a binary search tree is an ordered binary tree with the following properties:

- the tree consists of a (possibly empty) collection of linked *nodes*
- each node contains a single integer value, and has links to two subtrees
- either or both subtrees of a given node may be empty
- all values in a node's left subtree will be less than the value in the node
- all values in a node's right subtree will be greater than the value in the node

---

Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/21T1/labs/week03/downloads/lab.zip
```

If you're working at home, download `lab.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`bst.c`	a client program to read values into a tree and then display the tree
`mkpref.c`	a program to print number sequences that produce balanced BSTs
`mkrand.c`	a program to generate random number sequences, for further BST testing
`BSTree.h`	interface for the `BSTree` ADT
`BSTree.c`	implementation of the `BSTree` ADT
`Queue.h`	interface for the `Queue` ADT

`Queue.c`	implementation of the `Queue` ADT
`tests/`	a sub-directory containing a collection of test cases

There is quite a lot of code provided, but most of it is complete, and you don't necessarily need to read it... although reading other people's code is generally a useful exercise. The main code to look at initially is `bst.c`. This is the main program that will be used for testing the additions you make to the `BSTree` ADT.

The other two small programs (`mkpref.c` and `mkrand.c`) are there simply to provide data to feed into `bst` for testing. You can compile the `mkpref` and `mkrand` programs and try running them on a few sample inputs. You may need to read their source code to find out how to use them properly ... especially if they're not behaving as you expect. The `bst` program will also compile, but won't produce correct output until you've done your lab tasks.

Once you've read `bst.c`, the next things to look at are the header files for the ADTs, to find what operations they provide. Finally, you should open an editor on the `BSTree.c` file, since that's the file you need to modify for the tasks below.

The supplied `BSTree` ADT has slightly different operations to the one looked at in lectures, although the underlying data representation is the same in both cases. Another difference is that it includes four traverse-and-print functions, one for each of the different traversal orders (infix, prefix, postfix and level-order). Finally, it adds a function to count leaf nodes, in addition to the existing function to count all nodes.

The `make` command will build the supplied versions of the code:

```
$ make
gcc -Wall -Werror -g   -c -o bst.o bst.c
gcc -Wall -Werror -g   -c -o BSTree.o BSTree.c
gcc -Wall -Werror -g   -c -o Queue.o Queue.c
gcc -Wall -Werror -g -o bst bst.o BSTree.o Queue.o
gcc -Wall -Werror -g    mkpref.c   -o mkpref
gcc -Wall -Werror -g    mkrand.c   -o mkrand
```

You can test your solutions using the supplied `runtests` script. There are 4 test cases provided in the `tests` directory.

The following command will compile and run the second test case (2), and will produce `tests/2.out` from your output. If `tests/2.out` is the same as `tests/2.exp` you pass the test, otherwise you fail the test.

As soon as any test fails, it shows you the differences between your output (*.out) and the expected output (*.exp) and then quits.

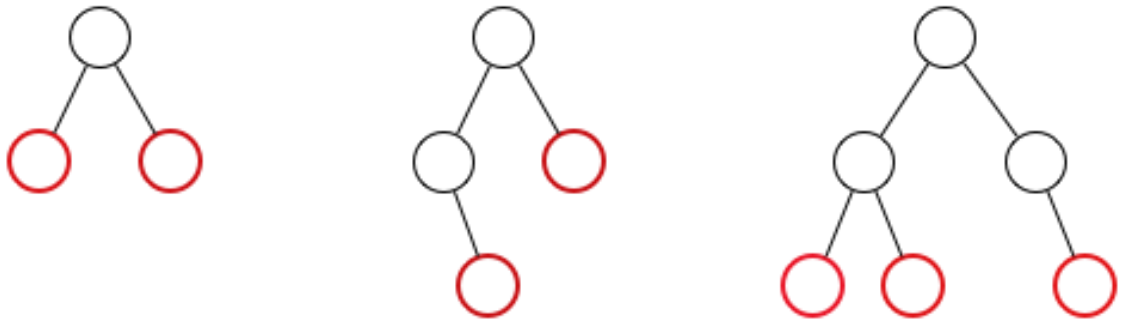
```
$ ./runtests 2
***                 ***
***  Testing lab03  ***
-----------------------------
** Failed Test 2

> Your output (in tests/2.out):
Tree:
      5
     / \
    /   \
   /     \
  3       7
 / \     / \
1   4   6   9

#nodes = 7,  #leaves = 0
Infix  : 1 3 4 5 6 7 9
Prefix : 5 3 1 4 7 6 9
Postfix: 1 4 3 6 9 7 5
ByLevel:

> Expected output (in tests/2.exp):
Tree:
      5
     / \
    /   \
   /     \
  3       7
 / \     / \
1   4   6   9

#nodes = 7,  #leaves = 4
Infix  : 1 3 4 5 6 7 9
Prefix : 5 3 1 4 7 6 9
Postfix: 1 4 3 6 9 7 5
ByLevel: 5 3 7 1 4 6 9

> Compare files tests/2.exp and tests/2.out to see differences
-----------------------------
```

Similarly you can test other cases, using commands like - `./runtests 1`, `./runtests 2`, `./runtests 3` and `./runtests 4`. Alternatively, run the command `./runtests` to test all the test cases.

Note that it will actually pass the first test (an empty tree), because the empty tree has no leaves and no values, which happens to be what the supplied code assumes. When you implement your code, make sure that you handle the empty tree correctly, or else this test might also start failing.

Task 1

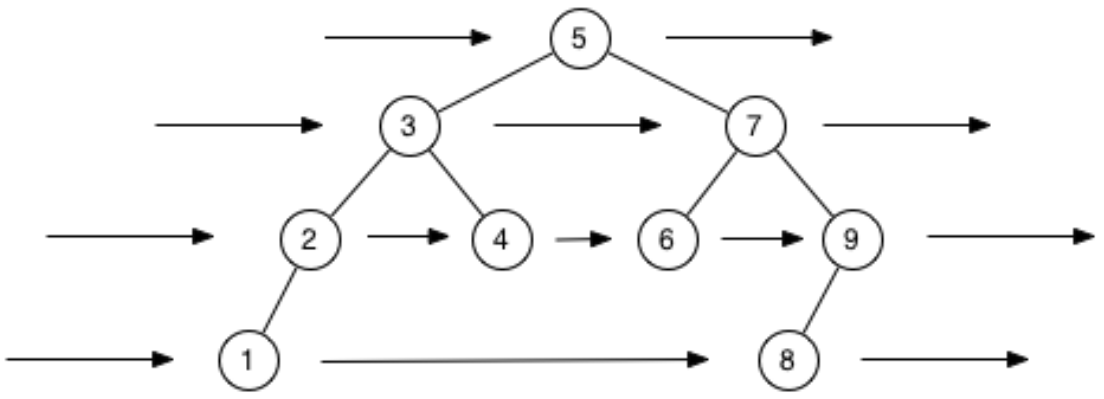
Implement `BSTreeNumLeaves()` which returns a count of the number of leaf nodes in the given `BSTree`. A leaf node is any node whose left and right subtrees are empty. The following diagram shows some sample trees, with the leaf nodes highlighted in red.



After you have implemented the function, determine its time complexity and add this to the comment above the function. The time complexity should be in terms of n, where n is the size (i.e., number of nodes) of the tree.

Task 2

Implement `BSTreeLevelOrder()` which prints the values in the `BSTree` in level-order on a single line separated by spaces (i.e. the same format as the other traverse-and-print functions). The following diagram aims to give an idea of how level-order traversal scans the nodes in a tree:



The output from this traversal would be: `5 3 7 2 4 6 9 1 8`.

Level-order traversal cannot be done recursively (at least not easily) and is typically implemented using a queue. The algorithm is roughly as follows:

```
Level Order Traversal(BSTree T):
    initialise queue
    add T's root node to queue
    WHILE the queue still has some entries DO
        remove the BSTree node at the front of the queue
        print the value in the BSTree node
        add its left child (if any) to the queue
        add its right child (if any) to the queue
    END WHILE
```

You must implement this algorithm for the `BSTree` data type by making use of the `Queue` ADT provided. This implements a queue of pointers to `BSTree` nodes.

> **NOTE:**
>
> The algorithm above doesn't contain *all* the logic that you need to include in your function. You will need to work out the additional parts yourself.

Note that you may *not* change any of the ADT interfaces. The *only* file you are allowed to change is `BSTree.c`. You can add extra local functions and data to `BSTree.c` if you wish.

After you have implemented the function, determine its time complexity and add this to the comment above the function. The time complexity should be in terms of n, where n is the size (i.e., number of nodes) of the tree.

Task 3

As discussed earlier, the `tests` directory contains 4 test cases and you can test them using the command `runtests`. You should use these to test your `BSTree` additions.

If the tests fail, you shouldn't rely solely on the test outputs to try and work out what the problem is. You will most likely need to add some diagnostic output or use GDB to try to work out *why* the tests failed.

If you check the output for the given tests, you might find that the provided tests are not very interesting. All of the nodes in the given tests either have zero or two children, and the trees are all reasonably balanced. Your task is to identify four more test cases that aren't covered by the given tests and put them in the files `tests/5.in`, `tests/6.in`, `tests/7.in` and `tests/8.in`. You should then collect the output from running `bst` on these four test cases. From within your `lab03` directory (not from within the `tests` directory) run the following commands to collect this output:

```
$ ./bst < tests/5.in > test5.txt
$ ./bst < tests/6.in > test6.txt
$ ./bst < tests/7.in > test7.txt
$ ./bst < tests/8.in > test8.txt
```

Open these output files to check that your functions produced the correct result/output. If the outputs seem correct, you can proceed to

submit these. Otherwise, fix your code and regenerate the output files.

Submission

Make the relevant changes to `BSTree.c` to complete the first two tasks, and then run the `bst` program on your new test cases to produce the `test5.txt`, `test6.txt`, `test7.txt` and `test8.txt` files.

You need to submit five files: `BSTree.c`, `test5.txt`, `test6.txt`, `test7.txt` and `test8.txt`. You can submit these via the command line using `give` using the command:

```
$ give cs2521 lab03 BSTree.c test5.txt test6.txt test7.txt test8.txt
```

or you can submit them via WebCMS. After submitting them (either in this lab class or the next) show your tutor, who'll give you feedback on your code, ask you about your results, and award a mark.

> **NOTE:**
>
> If you did not complete all the tasks, you must still submit *all* the required files. This could mean creating an empty file if a required file did not exist before.

Assessment

To receive a mark for this lab, you *must* demonstrate your lab to your tutor during your Week 03 or Week 04 lab session. You will be marked based on the following criteria:

Code correctness (2 marks)
This is the correctness of your code for Task 1 and Task 2. The correctness of your code will be determined by tests that we will run against your submission.

Complexity analysis (2 mark)
This is whether you were accurate with the time complexities that you obtained in Task 1 and Task 2 and whether you can explain your answers.

Test cases (1 marks)
This is whether you could identify four test cases that weren't covered by the given tests. You should be prepared to explain to your tutor what your tests cases cover that the given tests do not.
