COMP2521 20T2 Data Structures and Algorithms

Week 03 Lab Exercise

Binary Search Trees

Objectives

- to explore the implementation of binary search trees
- to implement a level-order traversal
- to develop test data

Admin

Marks	5=outstanding, 4=very good, 3=adequate, 2=sub-standard, 1=hopeless
Demo	in the Week 03 Lab or in the Week 04 Lab
Submit	`give cs2521 lab03 BSTree.c test6.txt test7.txt` or via WebCMS
Deadline	submit by 11:59pm Sunday of Week 3

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

Set up a directory for this lab under your `cs2521/labs` directory, change into that directory, and run the following command:

```
$ unzip /web/cs2521/20T2/labs/week03/lab.zip
```

If you're working at home, download `lab.zip` and then work on your local machine.

If you've done the above correctly, you should now find the following files in the directory:

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

You can test your solutions using the supplied script `autotest`. There are 5 test cases provided in the directory `tests`.

The following command will compile and run autotest for the case 2, it 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 expected output (*.exp) and then quits.

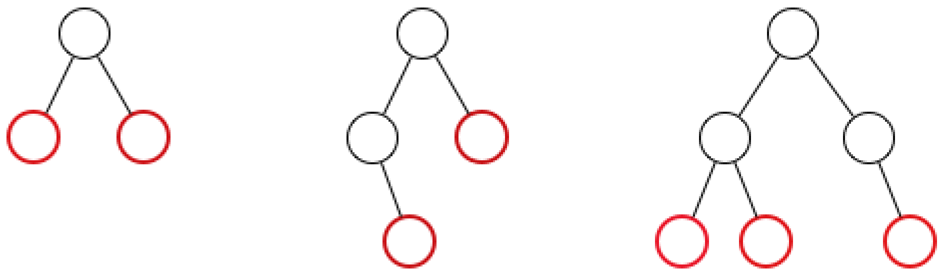
```
$ ./autotest  2
***                  ***
***  Testing lab03  ***
------------------------------
** Failed Test 2

> 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 - `./autotest 1`, `./autotest 2`, `./autotest 3`, `./autotest 4` and `./autotest 5`. Alternatively, run the command `./autotest` 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.

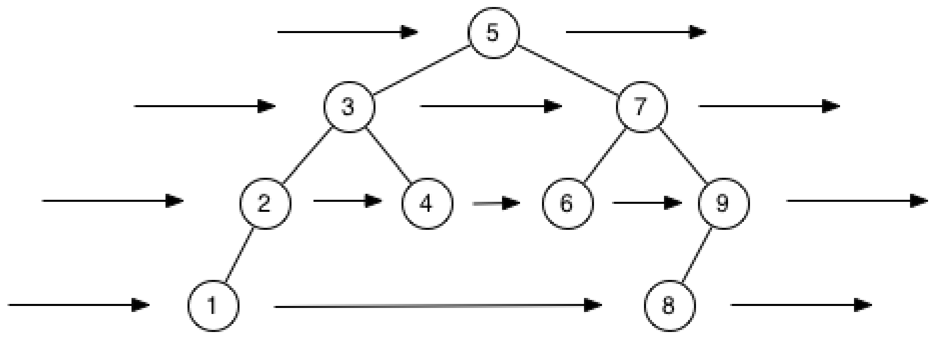
Task 1

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



Task 2

Implement the `BSTreeLevelOrder()` function 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
        take the head of the queue
        print the value from its BSTree node
        add the left child (if any) to the queue
        add the right child (if any) to the queue
    END
END
```

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

Task 3

As discussed earlier,the `tests` directory contains 5 test cases and you can test them using the command `autotest`. 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.

Once your code is working OK, think of two more *interesting* test cases and put them in the files `tests/6.in` and `tests/7.in`. What you *are* required to do is to collect the output from running `bst` with your code on these two test cases. From within the main lab directory (not from within the `tests` directory) run the following commands to collect this output:

```
$ ./bst < tests/6.in > test6.txt
$ ./bst < tests/7.in > test7.txt
```

Include the above two `test6.txt` and `test7.txt` files in your submission.

Submission

Make the relevant changes to `BSTree.c` to complete the first two tasks, and then run your `bst` program on your new test cases to produce the `test6.txt` and `test7.txt` files. You need to submit three files: `BSTree.c`, `test6.txt` and `test7.txt`. Submit them from the command line via `give`, and then show your tutor, who'll give you feedback on your coding style, your test quality, and award a mark.

Have fun, *jas*