SCSSE, University of Wollongong
CSCI124 Applied Programming
Autumn 2015

Assignment 2 (Due 11:59 PM Wed 13th May)

Objectives

To implement a binary search tree.
To work with pointers and dynamic memory.
To structure a small multi-file program correctly.
To work with generic data and generic functions.

Part 1 - Data

A linked list is formed when each node can have at most one child. This is done by having each node contain a single child pointer, typically called next. A binary tree is formed when each node can have at most two children.This is done by having each node contian two child pointers, typically called left and right.

Create a struct to define a node of the binary search tree. For now, we will assume that the tree is designed to hold a single letter, which will be used as the search key, and an integer count.

Part 2 - Interface

There are a number of things we may want to do with a binary search tree.

Initialise

Write a function to initialse your tree. It should take a pointer to the root of the tree and initialise it to NULL.

Add

Write a function to add a node to the tree. It should take the letter and count for the new node, along with the pointer to the root of the tree (passed by reference). There are 3 options, each worth more marks.

1. This is easiest to do recursively: If root is NULL, insert the new node at the root. If the letter to be inserted is less than the letter in the root, then we should add somehwere in the left subtree so call add again on the left subtree. If the letter to be inserted is greater than the letter in the root, then we should add somehwere in the right subtree call add again on the right subtree If the letter to be inserted is equal to the letter in the root, then a node with that letter already exists. In this case, just return false.*Note this is not the same as your linked list lab. The expectation is that the counts have already been established so we may try to add 'x' with a count of 50, if 'x' is already there (e.g. with a count of 23) then it should simply fail to add.*

2. However, as you know, recursive algorithms have a some overhead. Much like we saw with binary search, we can eliminate this overhead by writing an interative version:

   If root is NULL, we can insert the new node at the root.

   If not, then we need to search for the correct location, so start with a current node pointer set at the root. Then you will need to repeatedly move through the tree until you find the right location to insert the new node. You could use while(true) and return immediately after you add a node, you could use while(current node exists) which allows you to deal with the root at the end rather than the start, or you may prefer a more structured approach like while(location not found).

   If the letter to be inserted is equal to the letter in the current node, then a node with that letter already exists. In this case, just return false. If the letter to be inserted is less than the letter in the current node, then we should add somehwere in the left subtree, so check to see if the left child exists. If there is space, then we should insert the new node at the left child of the current node. However, if the current node already has a left child, then we will need to compare with that, so set the current pointer to be the left child and repeat the process

   If the letter to be inserted is greater than the letter in the current node, then we should add somehwere in the right subtree, so check to see if the left child exists. If there is space, then we should insert the new node at the right child of the current node However if the current node already has a right child, then we will need to compare with that, so set the current pointer to be the right child and repeat the process

   If the letter to be inserted is equal to the letter in the current node, then a node with that letter already exists. In this case, just return false. *Note this is not the same as your linked list lab. The expectation is that the counts have already been established so we may try to add 'x' with a count of 50, if 'x' is already there (e.g. with a count of 23) then it should simply fail to add.*

   Once you have found the correct location, you will need to create a new node at that location, set its children to NULL, and copy the appropriate data into the node.

3. You will notice that the above algorithm has to check the root, and later the left child and right child to see if they are empty, so there are three possible insertion locations. This means that, unlike the iterative algorithm where you always just update 'root', you need to write the code for creating the new node in 3 different places (or you create a node at the start but have to delete it when you discover you don't actually need it). This works but is not particularly elegant.

   While there are different ways to avoid this (including a function call or a reference variable), in this case we will use another level of indirection. At the start of the algorithm, create a pointer to the pointer to a node

When you find the right place to insert, use that to store the address of the insertion location. Once the the loop ends (assuming you have not returned), link in the newly created node at the location pointed to by the stored address.

Print

To check that your add function is working, you will want to print the nodes in your tree. Since writing this iteratively requires the use of a stack anyway, we will just do it recursively.

- Write a recursive function as follows: If the root is NULL, then return. Otherwise, call the function again passing it the left pointer (to print all the nodes in the left subtree), then output the data in the current node, then call the function again passing it the right pointer (to print the nodes in the right subtree).

  Check that everything is working as expected. If it is, then all letters should be printed in alphabetical order, regardless of the order in which you add them.

Remove

- Write a function to remove a node from the tree. It should take the target letter for node we want to remov and a reference to a location to store the count, along with the pointer to the root of the tree (passed by reference).

  Writing an iterative version is annoying, so just do a recursive remove function.

  If the root is NULL, then we can't find the node we want to remove, so return false. If the target letter is less than the letter in the root, then we should remove from the left subtree so call remove again on the left subtree. If the target letter is greater than the letter in the root, then we should remove somehwere from the right subtree call remove again on the right subtree If the target letter is equal to the letter in the root, then we have found the node we need to remove.

  Once the node has been found, we need to delete it. The process will be different depending on the children it has. If the node has no left child, then we can just set root to point to its right subtree. If the node has a left child, but no right child, then we can just set root to point to its left subtree.

  Unfortunately, if the node has both children, then the situation is a bit more complex. One way to do this, is to attach the left child somewhere else in the tree and then treat it as the first case. This is the way we will use in this assignment, since it is completed entirely with pointer manipulations. Find the leftmost node in the right subtree (it will have no left child), then update its left pointer to point to the left child of the node we plan to delete. Then we can set root to the right child of the node we plan to delete. Once the tree has been relinked, you should delete the node. The count associated with the removed node should be returned by reference.

Part 3 - bst

- Write a main driver to demonstrate that your binary search tree works correctly. Try to ensure that your test covers the different branches in each of your functions. Note that your test must not require any user input whatsoever (you may use files with input data provided you submit them).
- The resulting program should be named `bst`.
- Also ensure that your code is appropriately modular, with everything in the correct files.

Part 4 - Generic Data and Functions

This section will involve changing your program, and temporarily breaking part 3. So you may want to save a copy of your working code and/or comment out the existing data-specific functions.

Storing a copy of the given letter and count is not very flexible, so instead we will store a copy of whatever is provided.

Data

- Replace the letter and count in your node definition with a void pointer called data.
- We will also need a function to copy the data, one to compare two data items, and another to delete the data. A function for printing a data item would also be useful. Since this assignment doesn't cover classes, you can save the functions using four global function pointers with static(internal) linkage (see the lecture notes for an example) . Prototypes for the functions you will need to store are given in the section 'Main' below.

Initialise

- The functions for copying, comparing, deleting and printing will be data specific so they will need to be supplied to the initialisation function, which should save them in the function pointers you declared.

Add

- When creating a new node, call the copying function to make a copy of the given data and set the data field of the new node to point to it.

Print

- Rather than printing the letter and count directly, call the printing function.

Remove

- When searching for a new node, use the comparison function rather than > and <.
- When removing a new node, call the copying function to copy the data in the node, and return the address of the copy.
- Also call the deleting function to correctly delete the data.

Main

- Change your main program to test your new generic binary search tree by using a cstring instead of a letter and count. Note that your test must not require any user input whatsoever (you may use files with input data provided you submit them).

  ```
  const int MAX_WORD = 100;

  struct Word
  {
          char word[MAX_WORD];
          double frequency;
  };
  ```

  You will need to define the following functions to pass into initialise:

  ```
  //creates a new copy of the Word and returns its address as dest.
  void clone_word(void* &dest, const void* source);

  //deletes the given Word
  void destroy_word(void* data);

  //compares two words based on their frequency
  //returns 0 if they are equal, 1 if a > b and -1 if a < b
  int compare_word(const void* a, const void* b);

  //prints the frequency and string of a Word
  void print_word(const void* data);
  ```

Makefile

- Write a makefile named **makefile** to make your test program.
  Your test program must be named `bst`.
  It should run using the command `make` and should only recompile/relink those files which need it.

Submission instructions

Submit all .cpp and .h files for your assignment, along with your makefile.
If you have done part 4, you don't need to submit the previous version (at the end of part 3). `submit -u <username> -c CSCI124 -a 3 <source files> makefile`
from the directory containing your file, where `<username>` is your SOLS username.

1. Late submissions will be marked with a 25% deduction for each day.
2. Submissions more than three days late will not be marked, unless an extension has been granted.
3. If you need an extension apply through SOLS, if possible **before** the assignment deadline.
4. Plagiarism is treated seriously. If we suspect any work is copied, all students involved are likely to receive zero for the entire assignment.