Assignment Trees: Binary Search Trees

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- Learn about binary tree data structures and their performance characteristics.
- Practice recursion on tree structures
- More practice with pointers, dynamic memory allocation and linked node data structures.
- Experience with $\mathcal{O}(n \log_2 n)$ search time complexity.

Description

In this assignment we will be implementing some member methods of a `BinaryTree` data structure. This week we will only have and work with a pointer/linked node based implementation, so you have a `BinaryTree.hpp` header file that defines the base class abstraction for our binary tree data type, but only a `LBinaryTree.[hpp|cpp]` header and implementation for a pointer/node based binary tree. It is possible to use an array to manage a tree of nodes, see our Shaffer 3.2 C++ edition textbook section 5.3.3 for a discussion of an array based implementation. The only real reason to implement a binary tree as an array would be to save space. There can be a lot of overhead for each key/value pair we store as we need two 64-bit pointers for each storage node, pointers for the left and right sub-children. So if the tree gets even a little bit unbalanced, you quickly end up having a lot more empty/wasted space in an array based implementation than in a linked pointer implementation. Array based trees though are useful for heaps, as also discussed in our text, as a heap data structure will always be a balanced tree, and thus any wasted space will be minimal using an array to hold a heap tree structures. As with previous assignments, an API of the binary search tree data abstraction is available from our Shaffer 3.2 C++ edition, pg. 171. We will be basing our assignment for this unit on the Shaffer example, as our Malik textbook does not cover tree data structures.

As another wrinkle this week, we are using the concept of data as combinations of key/value pairs. We will discuss this in more detail in our next unit on hashing and dictionaries. The basic idea is that, for many applications we need to store a bunch of related information as a record (or structure), and we want to have an efficient way to search for and retrieve this record of information. Think of a typical database application, where you a record is normally thought of as a row of a table, and each row has some unique index or key that can be used to retrieve a row in an efficient manner.

We could have used key/value pairs in our `List`, `Stack` and `Queue` data structures from previous units. Lists in particular are often used to hold a collection of records, and it would be useful to be able to retrieve the record efficiently from a list. We did not do this explicitly for our unit on the `List` data type. For example, if the list were an unordered array or an unordered linked list, as we discussed, we could make it efficient to add new items onto the list in some manner (a $\mathcal{O}(1)$ operation). However, if we had the requirement that we need to search the list for a particular record, we would have to search through the whole list from the beginning, which would be $\mathcal{O}(n)$ performance. We could use a binary search as we discussed, which would be $\mathcal{O}(\log_2 n)$, but this would require us to use an array based implementation, and to keep the items in the array sorted by the key we want to look up records with. If the list were sorted, we could then perform a search relatively efficiently. But to keep the list always sorted so search is efficient, the insertion has to become a $\mathcal{O}(n)$ operation, like we saw for our priority queue assignment.

A binary search tree will allow us to have a data structure where search has performance $\mathcal{O}(log_2 n)$. Further, insertion and removal will also be $\mathcal{O}(log_2 n)$ into the binary tree (though in all cases this assumes the tree remains relatively balanced, as discussed in our materials for this unit). A $\mathcal{O}(log_2 n)$ operation is very good, even with huge amounts of data (billions and trillions of records), all operations will be very fast, including search. While for a list the best you can do is either a $\mathcal{O}(n)$ for either a linear search, or to do a linear traversal to perform an insertion to keep the list in

order. For very large n the performance for a list, one way or the other, will be noticeably slower and unacceptable for truly large amounts of data.

A `BinaryTree` has 3 basic operations `insert()`, `remove()` and `find()`, to add items to the collection, remove existing items, and search for an item. If you look at the `BinaryTreeNode.[hpp|cpp]` file, you will see that our nodes now have 2 pieces of information, a `Key` and a `Value`. The `Key` data type needs to be order-able by the standard operators, like `operator<()`. So for example `int` types can be used as keys, since you can tell which `int`s are smaller and bigger than which others. Same for `string` data types. User defined types can also be used as keys, as long as you overload and define needed operations for your type like `operator<()`.

You will be implementing these 3 basic `BinaryTree` API methods in this assignment. The `remove()` from a `BinaryTree` is a tricky operation, it requires at times rearranging nodes in the tree structure. Thus you will be working on the `remove()` function last, and we will implement two helper methods that will make the work of `remove()` easier, a `getMinimum()` method and a `deleteMinimum()` method.

Overview and Setup

For this assignment you will be given the following files that you will be using and adding code to for this assignment.

File Name	Description
`src/test-BinaryTreeNode.cpp`	Unit tests of the `BinaryTreeNode` structure that holds key/value pairs for our `BinaryTree` implementation
`src/test-LBinaryTree.cpp`	Unit tests of the link/pointer based `LBinaryTree` implementation of the `BinaryTree` API
`include/BinaryTree.hpp`	Header file of the ADT base `BinaryTree` class that defines the `BinaryTree` interface / abstraction
`include/LBinaryTree.hpp`	Header file of the concrete pointer/node based implementation of the `BinaryTree` abstract data type
`include/BinaryTreeNode.hpp`	Header file of the Node structure used by the linked pointer based implementation
`src/BinaryTree.cpp`	Implementation file of common methods of the `BinaryTree` base class
`src/LBinaryTree.cpp`	Implementation file for the `LBinaryTree` member functions that implement the concrete linked/pointer based `BinaryTree`

This week you will mainly be working exclusively on a linked node based `BinaryTree` implementation. The nodes are defined in the `BinaryTreeNode.[hpp|cpp]` files. This node is a full class with accessor methods to help manage the key/value pair of data, and the left and right links to child nodes in the tree structure.

You will be adding all of your new code and methods to the `LBinaryTree.[hpp|cpp]` files. The member methods to insert, remove and find records in the binary tree are missing from this implementation and need to be added.

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment Recursion' for our current class semester and section.
2. Clone the repository using the SSH URL to your local class DevBox development environment.
3. Configure the project by running the `configure` script from a terminal.
4. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
5. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment. Also make sure that you link the tasks with the open `Feedback` pull request.

Assignment Tasks

Task 1: Implement the `LBinaryTree` `insert()` Method

As usual make sure that you have created Task 1 on your GitHub repository for this assignment and have linked the Task 1 issue with the open `Feedback` pull request.

There are two tests cases in `test-LBinaryTree.cpp` for the implementation of the `insert()` member method. You can start by just uncommenting the first one, they both do similar tests, the second test case simply uses different data types for the key and value to test the template types are working for the `BinaryTree` container.

As usual, start by defining the `insert()` function in the `LBinaryTree.hpp` header file, and put in a stub function in the `LBinaryTree.cpp` implementation file. You should uncomment the `virtual` declaration of `insert()` in the `BinaryTree.hpp` abstract base class. This declaration should give you an idea of the signature of your insert function. Insert takes two parameters, a `Key` and a `Value` (both `const` reference parameters). Insert is a `void` function, so your initial stub function can do nothing. Make sure that you can compile your project and run the uncommented tests before beginning your implementation.

You will be implementing all of the member functions for this assignment as recursive functions. In order to do `insert()` as a recursive function, we need a second private version of `insert()`. This function should take 3 parameters, where the second and third parameters are the same `Key` and `Value`, but the first parameter is a `BinaryTreeNode<Key, Value>*` to a node in the tree. This pointer will initially be the `root` of the binary tree. Additionally this private version will need to return a pointer of the same type.

So you actually need two versions of the `insert()` function. You can probably figure out the signature of the public version from the definition given in the `BinaryTree.hpp` interface. But just to be clear and get you started, here are the signatures of the two functions you need in your `LBinaryTree.hpp` header file:

```
private:
  // this is a private member function, it will be a recursive function
  // that performs the actual work of inserting new key/value pairs into the
  // tree structure
  BinaryTreeNode<Key, Value>*
    insert(BinaryTreeNode<Key, Value>* node, const Key& key, const Value& value);

public:
  // this is the public insert() function defined by the BinaryTree API and
  // implemented in this concrete LBinaryTree class.  It will be what is used
  // by code that creates LBinaryTree instances.  It will call the private
  // insert() function to do the actual work
  void insert(const Key& key, const Value& value);
```

Having member functions with the same name in a class is a perfectly legal thing to do. This is a type of overloading, which we have made use of before. Here because the signatures are different (the private function takes 3 parameters instead of 2, and returns a result), there is no confusion between the two. Also notice that the private function needs to be declared in the `private` section of the `LBinaryTree` class declaration. This function cannot be used by someone external to the class. This is because, we need to start the recursive search by calling this function with the `root` of the tree. The `root` is a private member variable of the `LBinaryTree` implementation, so it is impossible for external code to get a reference to the root node of the tree in order to call the private insert directly.

The general recursive algorithm that you need to implement in the private `insert()` method is as follows:

Base Case: If the node we are evaluating is the `nullptr`, create a new node and return this new node. The `BinaryTreeNode` class is a full class this time, and it has defined constructors and helper function in order to use it. So for example, when creating a new node, you should use the standard constructor, and do something like the following:

```
BinaryTreeNode<Key, Value>* newNode = new BinaryTreeNode<Key, Value>(key, value);
```

and return this `newNode` as the result of the base case of the `insert()`.

General Case: If the node is not null, you need to perform a recursive search.

1. If `key <= node->getKey()` then you need to search in the left subtree.
2. Otherwise the `key` should be greater than current nodes `key` so you need to search to the right subtree.
3. in both cases you should return this `node` as the result, since the recursive `insert()` function returns a node as the result.

By "searching" we mean you need to call the private `insert()` method recursively on the `left` or `right` subtree as appropriate. The `left` and `right` members are private in the `BinaryTreeNode` class we are using, so to access them you have to call `node->getLeft()` and `node->getRight()` accessors respectively.

There is still more to do though. The base case creates and returns a new node once we reach a `nullptr` portion of the tree. So when you call `insert()` recursively, a new node might be returned. You should set the left or right child to the returned result of calling `insert()`. If a new node is created, it means that left or right children were `nullptr`, so in that case you will be adding the new node into the correct position of the binary tree. If the subtree you are searching down is not the bottom of the tree, then by step 3 above, the node should just return itself back to the caller. So you can again just set the `left` or `right` child to the return result of this recursive call and nothing happens in the case of an already existing subtree.

Once you have the private recursive `insert()` implemented, you still need to provide an implementation for the public `insert()`. This should be relatively simply. In your public `insert()` function, simply call the private version, passing in the `root` as the first parameter, and passing along the `key` and `value` you were given to insert. The private `insert()` will return back either a new node when the tree is initially empty, or return back the `root` if the new node is inserted somewhere lower in the tree. In either case, the node that is returned back to you should be assigned to be the value of the `root` of the binary tree. Additionally, after inserting a new node, you need to increase the `size` of the tree by 1. As we have seen in previous assignment, the `size` member variable is defined in the base class of the binary tree, so because this is a template, you will need to be explicit about this parameters namespace, for example by doing `BinaryTree<Key, Value>::size += 1` to increment the size.

Once you have the tests passing in the first unit test for `insert()` uncomment the next unit test for Task 1. This tests on a binary tree with some different data types for the key and value. Your implementation should pass these hopefully as well. Then, there is an array based constructor that needs to use your `insert()` method. Uncomment the call to your insert in the constructor, and uncomment the next two test cases. If your `insert()` is working it should be able to pass all of the first 6 test cases for task 0 and task 1 now for the assignment

When you are satisfied with your work for task 1, commit your changes and push them to the `Feedback` pull request of your class repository.

Task 2: Implement the `LBinaryTree` `find()` Method

We implemented `insert()` first so we could use it in the remaining tests. The `find()` method has a similar structure to `insert()`, and will probably be a bit simpler.

The `find()` method takes a `Key` as its input parameter (as a `const` reference parameter again). And it returns the `Value` it finds associated with the `key`, if the search succeeds. The `find()` method is a `const` member function, the tree will not be modified when you call `find()` to search the tree. You need to uncomment the virtual declaration of `find()` in the `BinaryTree.hpp` header, and the declaration here will be the same signature you need for the `find()` method (minus the `virtual` and the `=0` parts).

The `find()` method will again need a private version of it that will implement a recursive algorithm to perform the search of the tree. Your private function will still return a `Value` like the public member function. But like for `insert()` you need an additional first parameter of type `BinaryTreeNode<Key, Value>*` so you can pass in a pointer to a node in the tree while performing the recursive search. Add these two function stubs and uncomment the next unit test, and make sure you can compile and run the tests still. Again there are two unit tests for task 2, but you can start by just uncommenting and working with the first of the two unit tests.

The recursive algorithm for the recursive private `find()` is as follows:

Base Case: If you ever call the `find()` funciton with a `nullptr` node as the first parameter, this means the search has failed. In that case, you should throw a `BinaryTreeKeyNotFoundException` to inform the caller that the key they were expecting to find was not in the tree.

General Case 1: key found: Otherwise, you should first test for the case where the search succeeds. The search succeeds when the key in this node is the same as the key we were asked to find. If the keys match, you should return

the `value` from the node where you found the `key`.

General Case 2: keep searching: Otherwise the search hasn't failed yet, and you didn't find what you were looking for in this node. So search the left subtree if the key is less than the key of this node, or search the right subtree if the key is greater. In both cases you will be calling the recursive `find()` recursively, and it will return a result. You need to return that result of calling `find()` recursively as your own result.

The public `find()` method is even simpler than for `insert()`. You simply need to call the private method with the `root` of the tree as the first parameter, and return the `Value` result it gives back to you as the result of calling the public `find()` API implementation.

You should uncomment the other test case for task 2 and make sure that your implementation passes all of the tests of the `find()` method. When you have the tests passing, commit your changes and push them to the `Feedback` pull request of your assignment repository.

Task 3: Implement the `LBinaryTree` Private `getMinimum()` Helper Method

The only remaining public API method of our binary tree is the `remove()` method. Remove can be a bit tricky in some cases. If you ask to remove a leaf node, this method is very simply, simply delete the node and set the pointer to it from the parent to the `nullptr`. Likewise, it is still relatively simpe if the node you need to remove has only a left or a right subtree. In either case with only 1 subtree, delete the node to remove, and point the parents pointer that was pointing to the node you removed to its remaining left or right subtree. You should be able to convince yourself with some example trees by hand that this results in a correct binary tree.

The difficult case is when we are asked to remove a node that has both `left` and `right` child subtrees. We will discuss this case in more detail in task 5. But the solution involves finding and deleting the node with the minimum value in the subtree under the node we need to remove.

So in order to implement the difficult case for `remove()`, we will first implement two private helper functions that will be used by `remove()` for the case where the node to remove has both child subtrees.

The first of these private helper methods will be called `getMinimum()`. You will notice if you look in `BinaryTree.hpp` that this method is not defined as part of the API. Private methods will never be part of the abstract API of a data structure. For example, for an array based implementation, we would not need this method to help us manipulate the tree. So in short there is no public version of `getMinimum()`, it is not something that is generally useful to users of binary tree containers.

`getMinimum()` needs to take a `BinaryTreeNode<Key, Value>*` pointer to a node as its input, and it returns the this same type of value as its result. Basically what this function does is, it takes a starting node of a subtree that we need to find the minimum node of, and then it returns this minimum node it finds.
The minimum node from any starting node is easy to find, just keep following the `left` subtree until you get to a node with no `left` subtree, at which point this node must be the node with the minimum key of the starting subtree.

So the recursive algorithm for `getMinimum()` is fairly simple:

Base Case: If the node does not have a left subtree pointer, then just return this node.

General Case: Otherwise the node has a left subtree, so recursive call `getMinimum()` on the left subtree and return whatever this recursive call returns as your result.

There are again two test cases that test the `getMinimum()` function for task 4. You should uncomment the first one, and create a stub function that compiles and runs the tests before beginning your implementation. Once you can pass the tests in the first test case, uncomment and check you can pass them for the second test case of task 4 as well. Once you are satisfied with your results, commit your work for task 4 and push your commit to the `Feedback` pull request of your assignment repository.

Task 4: Implement the `LBinaryTree` Private `deleteMinimum()` Helper Method

The `remove()` member functions needs to first get a reference to the minimum node in its right subtree, then it will delete the minimum node after copying the values from the minimum node to itself. For many reasons it is easier to simply have the delete as a separate function. This method is thus similar to `getMinimum()`, we have to first navigate to the minimum node. But for delete, instead of returning what we find we want to remove the node we find in the subtree with the minimum value.

The `deleteMinimum()` function has the same signature as `getMinimum()` it takes a pointer to a node as input and will return a pointer to the node as its result. The recursive algorithm for `deleteMinimum()` is as follows:

Base Case: If the node has no left child subtree, it means we have recursively traversed to the minimum node in the subtree. In this case, return the right subtree of this node. We return the right subtree because we are going to delete this node, so pointing the parent node to this nodes right subtree is the correct thing to do to maintain the tree structure.

General Case: Otherwise continue searching left. But unlike for the `getMinimum()` we need to set our left subtree to what is returned from calling ourself recursively. This is because, if the left subtree is not the node to be deleted it should return itself. But if the left node is the node being deleted, it will instead return its right subtree, which if we set to our new left child will maintain the tree structure. Also once this method sets its left pointer to the return from the recursive call, it should return itself as the result of the general case.

There are again two test cases for the task 4 `getMinimum()` function. You can uncomment and compile with only the first of these two while working on implementing your function. And then when it is working, uncomment the second test case for task 4 and make sure your function works for different data types.

Once you are satisfied with your task 4 implementation, commit your changes and push them to the `Feedback` pull request of your assignment repository.

Task 5: Implement the `LBinaryTree` `remove()` Method

We had already begun describing the algorithm for the `remove()` method in task 3. There are a set of relatively easy cases to check for, then the hard case which will reuse your `getMinimum()` and `deleteMinimum()` private functions. The easy base cases are:

Base Case: Failed Search

1. First we need to test for a request to delete a `key` that is not in the tree. If the `node` we are currently testing is the `nullptr` this means the search failed. In that case you should throw a `BinaryTreeKeyNotFoundException` to inform the caller the expected `key` was not in the tree.

General Cases: Key not yet found, continue searching left or right

2. Otherwise if the `key` we want to remove is less than this current `node`s key, call `remove()` recursively on the left subtree. Again `remove()` returns a `node` result, which could be the result of deleting a node, so you should assign the return result from the recursive call back to this nodes `left` pointer.
3. Likewise, if the `key` is greater then this nodes key we need to search the right subtree. So call `remove()` recursive on the `right` node, and assign the return result back to the `right` pointer.

General Case: Key found, this is the node to delete

4. If we checked and the key was not less or greater then this nodes key, then we found the key so this is the node to delete.
    a. Make a temporary pointer and point it to this `node`. This will hold the `node` to return, which might be this `node`.
    b. If this node does not have a `left` child subtree, it means this `node` is either a leaf node (no children) or it has a right subtree. In either case we want to set the node to return to be the `right` subtree of this `node` and then delete this `node`.
    c. Else if this node does not have a `right` child, that means it only has a `left` child since we just check if it didn't have a `left` child. In that case, we want to get the `left` subtree and set it as the node to be returned, and then delete this `node`.
    d. Otherwise, we checked and the tree has both a `left` and a `right` subtree. This is the difficult case. When we have both subtrees, we want to find the minimum node in the right subtree, and swap this nodes values with the values of our minimum node. So first use `findMinimum()` to find the minimum node of the `right` subtree of this node. Then use the `setKey()` and `setValue()` methods to copy the value from the minimum node into the key and value of this node. Then finally set our right node to the result of calling `deleteMinimum()` on our `right` subtree. The effect is as if we copied the minimum values in our right subtree (which is the next closest value to this node) and then delete that minimum node in the right subtree. You should try this out by hand to convince yourself this works correctly to delete the asked for node and maintain the binary search tree structure. **NOTE**: The `deleteMinimum()` function removes the

node from the tree, but it should not actually deallocate the memory. It is better to deallocate the memory in `remove()`. At this point, after calling `deleteMinimum()` recursively, you should delete the minimum node's allocation using the C++ `delete` command.

This is a complex set of steps, and could maybe even use some more helper functions. For example, the whole general case for when we have found the node to remove might be better to remove to another private member function? But these steps are based on our `Shaffer` textbook implementation, so as a hint, try reading the textbook and looking at the explanation and code given there for a better understanding of how to implement the full `remove()` function.

When you are done, make sure you uncomment the second test case for task 5. When you are able to get all of the tests to pass for your `remove()` implementation, commit your work and push it to your assignment repository `Feedback` pull request.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 25 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 15 points are awarded for completing each of the 5 tasks. However you should note that the autograder awards either all points for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
   - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
   - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).

4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Lecture U11-1 Properties of Binary Search Trees
- Data Structures - Full Course Using C and C++ The videos 5:10:48 Introduction to Trees through 7:59:27 Inorder Successor in a Binary Search Tree are all excellent introductions to the fundamentals of implementing Binary trees and manipulating them.
- C++ Classes and Objects
- C++ Inheritance
- C++ Templates