MP11

MP 11 - Anagrams

Introduction / Motivation

In this MP you will be learning about and implementing logic for the binary search tree. This MP at hand involves two parts. First you need to write and test new methods for Binary Search Trees: **insert_node, find_node, find_parent_node, delete_node, pre_order traversal, in_order traversal, and post_order traversal**. In addition, you will need to create a new class called AnagramTree, which will construct a tree of words that can be searched for anagrams. An anagram is a word, phrase, or name formed by rearranging the letters of another. For example "cat" and "act". By creating an AnagramTree we will be able to contain a list of words that follow the same anagram pattern.

Given the words

Words for Dictionary

cat

dog

bat

god

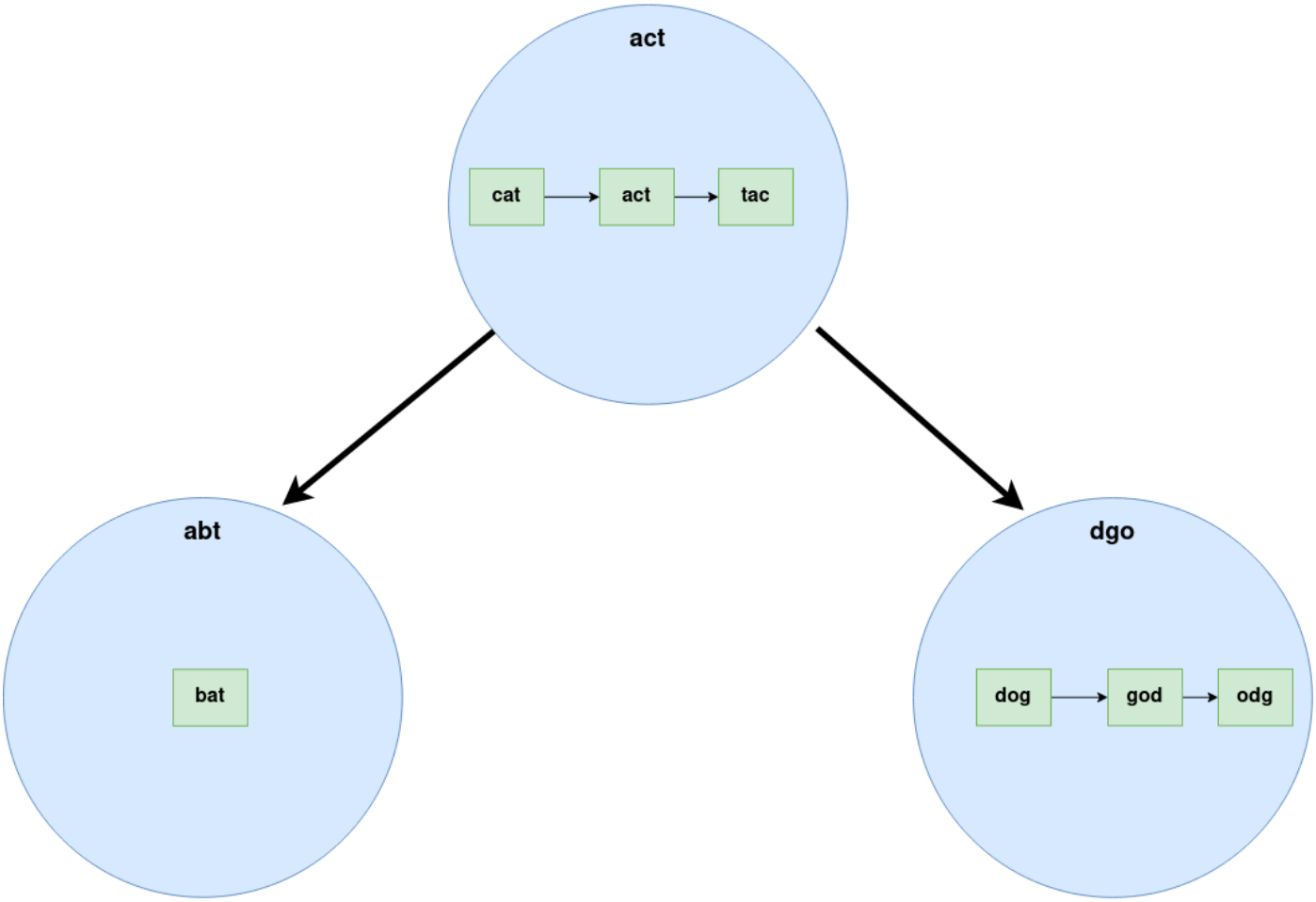
odg

act

tac

cat

The following anagram tree becomes. The anagram tree contains a key which are the words in lexicographic order, and the data is a linked list of the various words.



Part 1 - The Pieces

- `binary.cpp` - contains the functions you have to write for the binary tree. The code in this file will be graded.
- `binary.h` - contains the function definitions and some descriptions of the functions for the binary tree you have to write (DO NOT MODIFY)
- `anagram.cpp` - contains the functions you have to write to create the Anagram Dictionary. The code in this file will be graded.
- `anagram.h` - contains the functions definitions and some descriptions of the functions you have to write for the Anagram Dictionary (DO NOT MODIFY).
- `main.c` - contains the code to call and test the functions you implement in anagram.cpp

Details

Text Files

The data for the anagram trees are save and loaded in `.txt` files in the `/dictionaries` folder. Every line contains a word you will be inserting into your Anagram Tree.

The input text file would look like

Words for Dictionary

cat

dog

bat

god

odg

act

tac

cat

The outputted text file after the save_anagrams function is called would look like

inorder on inputted text file

bat

cat act tac

dog god odg

postorder on inputted text file

bat

dog god odg

cat act tac

preorder on inputted text file

cat act tac

bat

dog god odg

Part 1 - The BST Methods

The header file of a BinaryTree Node is given. Your task is to implement the member functions of the class.

```
template <class T, class Y>

class Node{
   public:
      Node<T, Y> (T key, Y data);
      T getKey();
      void setKey(T key);
      Y getData();
      void setData(Y data);
      Node<T, Y> *left;
      Node<T, Y> *right;

   private:
      T key_;
      Y data_;
};
```

The `Node` class is a template class that represents a single node in a binary search tree. Each node stores data of two types, `T` and `Y`, which represent the key and value of the node, respectively. We use a templated class as it allows for us to create one class that works for multiple data types. A user wouldn't have to create separate classes for different key and value data types.

The class provides the following public methods:

- `Node<T, Y>(T key, Y data)`: A constructor that creates a new node with the specified key and data.
- `T getKey()`: A method that returns the key stored in the node.
- `void setKey(T key)`: A method that sets the key stored in the node to the specified value.
- `Y getData()`: A method that returns the data stored in the node.
- `void setData(Y data)`: A method that sets the data stored in the node to the specified value.
- `Node<T, Y> *left`: A pointer to the left child of the node.
- `Node<T, Y> *right`: A pointer to the right child of the node.

The class also has the following private member variables:

- `T key_`: The key stored in the node.
- `Y data_`: The data stored in the node.

In order to use this class, you need to create an instance of the class with the appropriate `T` and `Y` types, and provide the key and data to be stored in the node. Once created, the public methods can be used to access or modify the key and data stored in the node, as well as access the left and right child nodes.\

The header file of a BinaryTree is given. Your task is to implement the member functions of the class.

```
template <class T, class Y>
class BinaryTree{

    public:
        BinaryTree();
      ~BinaryTree();
        void insert_node(Node<T,Y> *parent_node,Node<T,Y> *data);
        Node<T,Y> * find_node_parent(T key);
        Node<T,Y> * find_node(T key);
        void delete_node(T key);
        void pre_order(Node<T,Y> *node, std::list<Node<T,Y> > &list);
        void in_order(Node<T,Y> *node, std::list<Node<T,Y> > &list);
        void post_order(Node<T,Y> *node, std::list<Node<T,Y> > &list);
        Node<T,Y> * getRoot();
    private:
        Node<T,Y> *root;
        Node<T,Y> * find_node_parent(Node<T,Y> *node,T key);
        Node<T,Y> * find_node(Node<T,Y> *node,T key);
        Node<T,Y> * delete_node(Node<T,Y>* node, T key);
};
```

The `BinaryTree` class is a template class that represents a binary search tree, where each node has a maximum of two children. The tree stores data of two types, `T` and `Y`, which represent the key and value of each node, respectively. **The binary tree is sorted small-to-large where smaller values exist on the left of the tree and larger values exist to the right of the tree**. The tree is sorted based on the key.

The class provides the following public methods:

- `BinaryTree()`: A constructor that creates a new binary search tree.
- `~BinaryTree()`: A destructor that deallocates the memory used by the binary search tree.
- `void insert_node(Node<T,Y> *parent_node,Node<T,Y> *data)`: A method that inserts a new node with the specified data as a descendant of the given parent node.
- `Node<T,Y> * find_node_parent(T key)`: A method that searches the tree for the parent of a node with the specified key, returning a pointer to the parent node or NULL if the node is not found. The parent of the root node is also NULL. **You will need to use the private helper function to complete this function.**
- `Node<T,Y> * find_node(T key)`: A method that searches the tree for a node with the specified key, returning a pointer to the node or NULL if the node is not found. You will need to use the private helper function to complete this function.
- `void delete_node(T key)`: A method that deletes the node with the specified key from the tree. You will need to use the private helper function to complete this function.
- `void pre_order(Node<T,Y> *node, std::list<Node<T,Y> > &list)`: A method that performs a pre-order traversal of the tree, adding each visited node to the specified list.
- `void in_order(Node<T,Y> *node, std::list<Node<T,Y> > &list)`: A method that performs an in-order

traversal of the tree, adding each visited node to the specified list.

- `void post_order(Node<T,Y> *node, std::list<Node<T,Y> > &list)`: A method that performs a post-order traversal of the tree, adding each visited node to the specified list.
- `Node<T,Y> * getRoot()`: A method that returns a pointer to the root node of the tree.

The class also has the following private member variables:

- `Node<T,Y> *root`: A pointer to the root node of the tree.

In order to use this class, the following needs to be implemented:

- The `Node` class, which represents a single node in the binary search tree and contains the `T` and `Y` data types.
- The implementation of the public and private methods in the `BinaryTree` class, which should adhere to the standard binary search tree algorithms.

Below is the pseudocode for the helper function on how to delete a Node from a binary tree

```
delete_node(node, key):

    if node is NULL:

          return node

    else if key less then "node key":

         "node left" = delete_node("node left", key)

    else if key greater then "node key":

         "node right" = delete_node("node right", key)

    else: # Node has been found

          if node has no children:

             delete node

             return NULL

          else if node has no left child:

               set temporary node equal to right child

               delete node

               return temporary node

          else if node has no right child:

               set temporary node equal to left child

               delete node

               return temporary node

          else: # has two children
```

```
                temp = node.right

                while temp has a left child: # get the left most child that's greater then node you are
trying to delete

                      temp = temp.left

                node.setKey(temp.getKey())

                node.setData(temp.getData())

                node.right = delete_node(node right, temp key)

     return node
```

Part II - The Anagram Generator

The header file of an Anagram Tree is given. Your task is to implement the member functions of the class.

```
lass AnagramDict {

    public:

         AnagramDict(std::string filename);

         std::string sortWord(std::string word);

         void addWord(std::string word);

         void saveAnagrams(std::string order);

         BinaryTree<std::string, std::list<std::string> > tree;

         std::string filename_;

};
```

Class Members

- `tree`: A binary tree instance of `BinaryTree<std::string, std::list<std::string>>` used to store the anagrams.
- `filename_`: A string representing the name of the file you are testing.

Class Methods

- `AnagramDict(std::string filename)`: A constructor that creates an instance of the `AnagramDict` class with the given filename. It loads the anagrams from the file into the binary tree instance.
- `std::string sortWord(std::string word)`: A method that takes a string and returns a sorted string by sorting the letters in the string alphabetically. You may use the std::sort function or come up with your own sort algorithm.
- `void addWord(std::string word)`: A method that takes a string and adds it to the binary tree instance. It does this by first sorting the string, then checking if the sorted string already exists

as a key in the binary tree. If it does, it appends the new word to the list of anagrams under that key. If not, it creates a new node in the binary tree with the sorted string as the key and a list containing the new word as the value. The tree should not contain any repeating words

- `void saveAnagrams(std::string order)`: A method that takes a string (can be "pre", "post" or "in") indicating the order in which to save the anagrams and saves the anagrams to a file in that order. It does this by traversing the binary tree and appending the anagrams to a list in the specified order. It then writes the list to the file.

Building and Testing

A Makefile is provided to you.

```
make
```

compiles your code. To execute:

```
./mp11 test1.txt pre
```

Replace `test1.txt` with other files in the directory and check out how the outputs differ.

Replace "pre" with "pre", "in", "post" to do different traversals on the dataset.

We have given you a couple of gold outputs for the testcases to help check your functionality. **These test cases are not comprehensive and you may have to come up with your own testcases.**

The complete set of test cases used by the autograder is provided to you as the following makeable test:

To execute the test:

```
./mp11_test
```

Grading

Test cases only test up to 59 points of the functionality, the rest of the 41 points come from hidden tests.

Functionality(100%)

- BinaryTree(); - (3%)
- ~BinaryTree(); - (3%)
- void insert_node(Node *parent_node,Node* data); - (8%)

- Node * find_node_parent(T key); - (3%)
- Node * find_node(T key); - (3%)
- void delete_node(T key); - (3%)
- void pre_order(Node *node, std::list > &list); - (7%)
- void in_order(Node *node, std::list > &list); - (7%)
- void post_order(Node *node, std::list > &list); - (7%)
- Node * getRoot(); - (3%)
- Node (T key, Y data); - (3%)
- T getKey(); - (3%)
- void setKey(T key); - (3%)
- Y getData(); - (3%)
- void setData(Y data); - (3%)
- AnagramDict(std::string filename); - (3%)
- std::string sortWord(std::string word); - (3%)
- void addWord(std::string word); - (8%)
- void saveAnagrams(std::string order); - (8%)
- Hidden Test Case on Dictionary Output; - (16%)

If your code does not compile, you will receive 0.

This code has been adapted and taken from CMU's intro to Data Structures Class https://www.cs.cmu.edu/~mjs/121 (https://www.cs.cmu.edu/~mjs/121)