Lab 6: "Release the Crack-ing"

Due: April 2nd at 11:59pm

Overview

The goal of this lab is to implement the key functionality of a Binary Search Tree (BST) using recursive functions and use your BST to implement two cryptanalysis tools to analyze and crack a substitution cipher. **This is a group lab, groups of two, but you must work with someone in your lab section.**

Deliverables

Your submission should include requisite code necessary to run your program, but the following files will be the focus of grading:

- `Makefile`
- `README`
- `cipher/*`
- `plain/*`
- `key/*`
- `sample/*`
- `crack.cpp`
- `freq.cpp`
- `BST.h`
- `linked_BST.h`
- `BST_node.inl`
- `linked_BST.inl`
- `linked_BST_private.inl`
- `pair.h`
- `test_BST.cpp`

All starter code is provided via `update35`.

Submission

You will submit using `handin35`. Be sure to place all relevant files in `cs35/labs/06`.

"Release the Crack-ing"

In this lab you will implement a Binary Search Tree by completing a set of recursive functions over subtrees. You will then use your implementation to complete two cryptanalysis programs, `freq` and `crack`. Cryptanalysis is the process of analyzing cipher text (or encrypted text) and trying to determine the key to reveal the private message, thus "cracking" the code. Both provided cipher texts are encrypted using a substitution cipher, where each letter in the cipher text maps to a unique letter in the plain text. By analyzing the frequency and occurrences of each letter in the cipher text, you should be able to determine the mapping to the plain text; for example, "E" is the most common letter, so the letter that occurs most frequently in the cipher text likely maps to "E". Below is an example of the cipher-texo

```
 >./crack cipher/cipher4.txt
Mapping:
Cracked Text:
NGKMXKP JFUM ESS XGK DEA XGPFYBG XGHM, LYX UFKM JFX PYUKSA HJXKPPYWX
PEJUA EM E AFYJBKP JKPU DFYSU. AFYP AFYJBKP JKPU XECKM FOOKJMK QYHNCSA
DGKJ MFTKFJK JKEP GHT LKBHJM XF YXXKP UKNSEPEXHVK MKJXKJNKM, LKNEYMK
GK PKEUM HJXF HX EJ EMMKPXHFJ XGEX GK, XGK JKPU, UFKM JFX ESPKEUA CJFD
XGK HJOFPTEXHFJ LKHJB HTWEPXKU. LYX AFYP FSUKP JKPU GEM TFPK MKSO
NFJOHUKJNK, EJU LKMHUKM, YJUKPMXEJUM XGEX OPKQYKJXSA WKFWSK JKKU XF
XGHJC FYX SFYU. EJU GHBGSA EUVEJNKU JKPUM DHSS OYPXGKPTFPK YJUKPMXEJU
XGEX YXXKPHJB UKNSEPEXHVK MKJXKJNKM DGFMK NFJXKJXM EPK ESPKEUA CJFDJ
XF ESS WPKMKJX HM WEPX FO XGK MFNHES WPFNKMM FO TECHJB NFJVKPMEXHFJ
EJU XGKPKOFPK MGFYSU JFX LK NFJMXPYKU EM EBBPKMMHFJ YJUKP EJA
NHPNYTMXEJNKM.
```

There is currently no mapping from cipher to plain text, so `crack` just outputs the raw cipher text. As you analyze the cipher text, you will add mappings to a key file, so that eventually, you will be able to read the plain text. There are plenty of cipher texts for you to crack, all sampled from literature (mainly in sci-fi/fantasy) – name the novels and win the prize!

Starter's Abstract As usual, this is an involved lab requiring you to touch many pieces of code, so below is an abstract of the work required. A more complete description of each of the provided files is found at the end of this document.

1. Implement the following private recursive functions for a LinkedBST in `linked_BST_private.inl`: `insertInSubtree()`, `containsInSubtree()`, `findInSubtree()`, `updateInSubtree()`, `getMinInSubtree()`, `getMaxInSubtree()`, `getHeightInSubtree()`.
2. Test your implementation using `test_BST`. Focus on one test at a time.
3. Implement `removeFromSubtree()` and write the test for it in `test_BST.cpp`.
4. Implement the tree traversal routines `buildPreOrder()`, `buildInOrder()`, `buldInOrder()` in `linked_BST_private.inl` and `getLevelOrder()` in `linked_BST.inl`.
5. Complete the `crack` and `freq` program in `crack.cpp` and `freq.cpp`. (Note that you do not need to complete `remove()` to finish these programs)
6. **(Challenge)** Write a program, `encrypt`, that uses your BST implementation to encrypt a file using a randomly choose substitution file.
7. **(Extra Credit)** Write a program, `the_cracken`, that automates the process of frequency analysis and cipher text cracking to iterate through a set of possible keys until a the plain text is found.

The Binary Search Tree Declaration and Implementation

There are four primary classes that you will need to understand and potentially edit to complete this lab: BST, LinkedBST, BSTNode, and Pair.

- **BST** is a (pure-virtual) abstract Binary Search Tree interface, declared `BST.h`. This defines the standard methods of a BST, documentation of these functions can be found in the header file.
- **LinkedBST** is the implementation of a Binary Search tree and inherits from BST. Documentation can be found in the header file, `linked_BST.h`. It stores a pointer to the root of the tree, a BSTNode (see below), and it also declares the standard BST functions. However, the LinkedBST declares private recursive functions that will actually do the task of manipulating and finding content in the tree. All the private recurisve functions have the same prefix as the standard functions with an "inSubtree" prefix. The standard BST functions are just wrappers to the recursive methods, like so:

```
/**
 * Given a key, returns the value associated with the key in the tree.
 * Throws a runtime_error if the key is not in the tree.
 */
template <typename K, typename V>
V LinkedBST<K,V>::find(K key) {
  return findInSubtree(root, key);
}
```

  You are tasked with implement the private recursive functions, and this code can be found in `linked_BST_private.inl`.

- **BSTNode** represents the standard recursive definition of a tree, where each node stores a key and a value with pointers to the left and right subtree:

```
template <typename K, typename V>
class BSTNode {
  private:
    K key; //The key
    V value;//the value

    BSTNode<K,V>* left; //pointer to left sub-tree
    BSTNode<K,V>* right;//pointer to right sub tree

   //default constructor inherited in c++
    BSTNode();

    //or a constuctor that set the key value
    BSTNode(K k, V v);

  friend class LinkedBST<K,V>;

};
```

  One thing to be aware of with respect to the BSTNode is that the LinkedBST is declared as a friend class. That means that in LinkedBST you can reference (and set) the private fields in the BSTNode. This is really handy when writing recursive functions in the LinkedBST that need to change the left or right pointer. The friend relationship also allows for the use of two *private* constructors that only the LinkedBST can call. One constructor is the default c++ constructor which will initialize everything to `NULL`. The other constructor will set the left and right pointer to `NULL` but initialize the key and value. The declaration of BSTNode can be found in `linked_BST.h`

- The **Pair** class is a template class to hold a key-value pair. You will use this class during traversal, adding a Pair's that stores the key-value's in the BST into a Queue (provided in the library) as you traverse the tree. The Queue is referred to as the *iterator* of the tree because it provides a linear mechanism to iterate through the nodes of tree based on the traversal order. The Pair class should not instantiated using the new operator, instead use stack instances only. The reasons for this is rather esoteric and having to do with handling memory leaks, but if you stick to using stack instances everything should work out fine. Below is the class declaration for a Pair as a reference and the declaration and definition can be found in `pair.h`:

```
/**
 * A Pair is an abstract container class for two pieces of data, which it
 * stores publicly.
 */
template <typename F, typename S>
class Pair {
  public:
    F first;   // The first item in the pair.
    S second;  // The second item in the pair.

    Pair() {};
    Pair(F f, S s) {first = f; second = s;};
};
```

Notes on insertInSubtree and removeFromSubtree

The two most challenging recursive functions will be `insert()` and `remove()` because they manipulate the tree:

- **insertInSubtree()**: As a reference, note the documentation for the return value of `insertInSubtree()`:

```
    /**
     *Recursive function that inserts a new node into a
     *sub-tree pointed to by current
     *
     *@error runtime_error : duplicate key
     *
     *@arg BSTNode<K,V> * current : a pointer to the current root-node of the sub-tree
     *@arg K key : the key for the new node being inserted
     *@arg V value : the value for the new node being inserted
     *
     *@return BSTNode<K,V> : the root of the sub-tree (on the last recursive
     *                       call this will be the new node in the
     *                       tree and the initial call, will return
     *                       the root of the whole tree)
     **/
    BSTNode<K,V>* insertInSubtree   (BSTNode<K,V>* current,
                                     K key,
                                     V value);
```

Given the return value, the way you should think about the recursive insert is that when `NULL` is reached in the recursive trace, this is the location where the new BSTNode should be. That is, the routine has recursed a path down the tree and found a place, indicated by `NULL`, where the key should be if it was pressent in the tree. At this point, the function can return the new BSTNode, which will be set in the previous recursive step, based on the path choice. For example, the recursive-case for this routine will look something like this:

```
* if key is less than current key:
   - set the current left to the insert on the left subtree
* if key is greater than current key:
   - set the current right to the insert on the right subtree
```

Thinking about this process further with respect to inserting: If the recursion reaches a `NULL` and a new BSTNode is returned, it will get set to either the right or left pointer of the parent. This can be conceptually difficult at first, but the code is rather simple and elegant. Follow your nose, and you'll probably get it right.

- **removeFromSubtree**(): When implementing the recursive remove, you need to carefully consider the cases. You could be removing a leaf; or you could be removing a parent with one child; or you could be removing an interior node, a parent with two children. The first two cases are relatively easy, but the third case will require some thought since you must find a node to replace the one being removed. Recall from lab, we described the node that should replaced the removed one as the maximal key/value in the left subtree, since it maintains the division between left and right subtrees. The challenging part is keeping all this straight in the code, and to help pseuo-code is provided in the skeleton code.

  Again, like in insert, the return of the recursive remove is the (potentially new) root of the subtree, so you will need to use recursive assignment like in `insertInSubtree()`.

Crypt-Analysis Using `freq` and `crack`

In this part of the lab, you will use your BST implementation to complete two cryptanalysis program. The `freq` program will calculate a letter frequency table of an input text, and `crack` will use a mapping from cipher-letters to plain-letters to decrypt a message.

Substitution Ciphers

You will be provided with a set of cipher texts that are encyrpted using a substitutiont cipher. A substitution cipher describes a process where plain text (unencrypted text) is encyrpted by mapping each letter in the plain-text to a letter in the cipher text. The unique mapping is described as the *key* to the cipher and with knowledge of the key, the cipher-text can be decrypted revealing the plain-text. As example of a substitution, consider the following cipher-text: `QLYYZ EZJYM`, and the following key (or mapping):

```
A F
B X
C O
D H
E I
F S
G N
H A
I L
J P
K R
L C
M E
N T
O B
P U
Q K
R Q
S V
T D
U J
V M
W G
X Y
Y Z
Z W
```

You can read the key such that, for each line, the first letter refers to a cipher-text letter and the second letter refers to a plain-text letter. So after working though decryption using the key, you can see that `QLYYZ EZJYM` will decrypt to "HELLO WORLD." The format above will be the format you will use for generating your own key files for `crack`.

Cracking Substitution Ciphers

The problem with substitution ciphers is that they are easily cracked without any prior knowledge of the key. This is due to features of human language: some letters are more frequent than other. If you were to inspect cipher-text encyrpted with a substitution cipher, you'll find that some letters occur more frequently than others, and it must be the case that these letters map to high frequency letters in the plain-text, e.g., "E" "O" "S" "R", since there is a unique, one-to-one mapping.

To crack the cipher text, you first need to have an understanding of the letter frequencies in plain text. You can learn this information by using the `freq` program on some known samples. There are some

provided to you in the `samples` directory, including samples from *Moby Dick* and *Oliver Twist*.

Next you will use the `freq` program to analyze the letter frequency in the cipher text, keeping an eye out for high frequency cipher-letters. One you have identified likely candidates for the common letters, e.g. "E", you can place that mapping in your key file, formatted like above. Run the `crack` program using the key on the cipher text hunting for plain text words, as you start to see partial words you can fill in the missing letters in the key, e.g., "TH" with out an "E". Eventually, as you iterate over this process, the plain-text will be revealed.

Place your plain text decryption in the directory `plain` and your key files in the directory `key`. Use the same number postfix (e.g., the 1 in `cipher1.txt`) to indicate which cipher-text map to which plain-text (`plain1.txt`) using which key (`key1.txt`).

Grading and Extra Credit

Rubric

This lab will be graded on a 50 point scale:

- (10 Points) Core BST tree access functionality (not including`remove()`)
- (10 Points) `remove()` implementation and test code
- (10 points) Implementing traversal routines
- (5 points) Implementing `crack` and `freq`
- (5 points) Cracking the cipher-text
- (10 Points) Coding style and memory management

Note that there are points dedicated to aspects of your lab that are not part of the solution. Do pay attention to coding style and managing memory, design and lay out your code effectively so that it is readable and easily debugged. Run your program through `valgrind` and plug any memory leaks.

Challenge and Extra Credit

- **(Challenge)** : Use your BST implementation to write a program `encrypt` that will apply a substitution cipher to some input plain text. You can copy the general format and code `crack.cpp` to do this, and you should be sure that your program also outputs the key in a format that `crack` can decrypt the file.
- **(Extra Credit 7pt)**: Write a new program `the_cracken` which will automate the process of apply frequency analysis and cracking techniques to a cipher-text message. The minimal input to the `the_cracken` is the cipher-text, and the output is a key and the plain-text. In order to receive full credit, your program should go beyond just trying all the possible mappings. It should use some heuristic or logic to try more likely key mappings first before less likely ones.

Provided Code

- `Makefile` : The Makefile. You will only need to modify the makefile if you attempt the challenge or extra credit
- `sample/*` : Directory of sample text for you to analyze using freq.
- `cipher/*` : Directory of cipher text for you to crack.
- `plain/*` : Directory to store your plain text decryption of the cipher text.
- `key/*` : Directory to store your key files.
- `crack.cpp` : The source code for `crack` You will need to edit this file.
- `freq.cpp` : The source code for the `freq` program. You will need to edit this file.
- `BST.h` : The header file for the abstract BST. You will not need to edit this file, but you should read it for the documentation.
- `linked_BST.h` : The header file for the linked BST implementation. This file contains the classes LinkedBST and BSTNode. You will not need to edit this file, but you will need to read it carefully.
- `BST_node.in;` : The inline implementation of the BSTNode. You will not need to edit this file.
- `linked_BST.inl` : The inline implementation of the public functions of the LinkedBST. You will need to edit this file for `getLevelOrder()`.
- `linked_BST_private.ink` : The inline implementation of the private recursive functions of the LinkedBST. You will need to edit large parts of this file.
- `pair.h` : The header file for a pair. You will not need to edit this file, but you shold read it.
- `test_BST.cpp` : The test file for your BST. You will need to edit this file, and use it to test your implementation.

Compilation

Compilation will occur using `make`. To compile the amazing

```
bash> make
```

If you add extra functionality contained in other files, you should edit the `Makefile`, but it is not necessary. To compile the test code:

```
bash> make test
```

Which will compile an executable `test_BST` at first, you can add more tests as you develop.

Execution

Here are the command line arguments for `crack`:

```
bash>./crack -h
freq [OPTIONS] file.txt
         Output a decryption of file.txt using the mapping
Options:
         -h      print this usage
         -m mapping      using the mapping file formatted with one
                         mapping per line, with input to output chars
                         seperated by white space from the first char
                         to the second
```

And here are the command line arguments for `freq`:

```
bash>.//freq -h
freq [OPTIONS] file.txt
         Generate a character frequency table for
         the input file, file.txt
Options:
         -h      print this usage
         -g      output gnuplot graph, pipe output (dflt: false)
                    ./freq -g file.txt | gnuplot -persist
```