CS251
Simple Search Engine using Red-Black Tree
(Part 2 due 03/26/2018 at 11:59pm)

Red Black Tree Lecture

Overview:

A red–black tree is a kind of self-balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Balance is preserved by painting each node of the tree with either red or black in a way that satisfies certain properties -

1. Each node is either red or black.
2. The root is black.
3. All leaves (NIL) are black.
4. If a node is red, then both its children are black.
5. Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.

When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

The balancing of the tree is not perfect, but it is good enough to allow it to guarantee searching in O(log n) time, where n is the total number of elements in the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in O(log n) time.
Black depth - the number of black nodes from the root to a node
Black height - the uniform number of black nodes in all paths from root to the leaves

In this project you will implement Red-Black (RB) tree. Then we will parse a list of URLs and store the **words in the URL** using that RB tree. Later we will search the RB tree for retrieval of links in which a word (or, words) appear(s).

This project is comprised of two parts, each has its deadline. The parts are dependent on each other, e.g., you will not be able to implement part2 without successfully completing part1.
Before starting the lab, make sure you review the lecture slides to completely understand RB tree.

Setting up your environment:

Remote login to data by typing:
ssh <your-user-name>@data.cs.purdue.edu

Copy the initial project-4 files by typing:
cp /homes/cs251/Spring2018/project4/project4-src.tar .
tar -xvf project4-src.tar
cd project4-src

What's in the tarball:

You are provided with all the skeleton files you need to implement your project. The header files (*.h) contain the class declarations while the *.cpp files contain the definitions of the class methods. You are provided with some method declarations in the header files, but you need to add more methods to them (the provided header files are incomplete).

Reference implementation:

You are also provided with the solution's executables in this project. You should run 'scan.org' to get familiar with the ***URL parse*** task and 'search.org' to get familiar with the ***search*** part. Your final program should produce files that are identical to those produced by '*scan.org'* and '*search.org'*.

Tests:

In the tests/ directory, there is a script "testall.sh" that you will use to test your project. You can also run it to test different parts of the project separately, for example "./testall.sh -p1" will test part-1 of the project only.

Milestone 1: Implementation of Red Black Tree and parsing URLs to populate the RB tree

(Due 03/19/2018 at 11:59pm)

Part 1: Implementation of Red Black Tree

The first step towards our goal is to implement the Red-Black Tree (RBTree). We want our implementation of RBTree to support polymorphism. One way to do it is to implement it - using ***templates***.

First you need to implement the 'Node' data structure which will be the nodes of our RBTree. And, then you implement the actual 'Red Black Tree' data structure. Both of these are **templated with two arguments - Key and Element.** Using template means we can make our tree contain any kind of data pairs (including non-primitive objects), such as:

- `<string, int>` : Key is *string* and Element is of type *int*
- `<int, string>` : Key is *int* and Element is of type *string*
- `<string, vector<string, int>>` : Key is *string* and Element is of type *vector<string, int>* type (vector is a data type implemented in STL)

Step 1: Implement the Node class

The **Node** class is the building block for the Red Black tree. A RBTree node contains: **key** and **element**, which represent the key you structure your RBTree with and the element of each node respectively. It has an additional member variable - **color**, which controls the color constraint in RBTree. Initially the value of color should be RED. It also contains three pointers (parent, left and right) to the parent, left and right child. These pointers initially point to **NULL**. Complete the implementation of the methods in **Node.h**.

Additionals:

- **enum** is a data type in C++ consisting of a set of named values. You can use these to hold the color member variable. You can use this type as any class object, for example, *color c = RED*.
- **Operator overloading** is when you have different implementations of the same operators depending on their arguments. Normally the operators (+/-/*/<< or >>, and so on) handle the primitive types. But when you want different operations with non-primitive objects you can also overload the operators to handle them. That is what we did in two last functions of Node.h - `operator<<` and `operator>>`.
  You should not change anything in these two functions.

  Example 1:
  To print an object of `Node < Key, Element >*, root` to an outputstream (say, ***out***), you do the following:

  `out << * root`

  Example 2:
  To take an object of `Node < Key, Element >, root` as input from an inputstream (say, ***in***), you do the following:

  `in >> root`

Step 2: Implement the class for Red Black Tree

Now, we would like to implement a RBTree data structure from scratch to store the Node objects. The RBTree object has the ***Node<Key, Element>**** and ***size*** as member variables. Implement the methods in **RBTree.h**. The tree will initially contain only the root node, which will be null initially and size will be 0. The RBTree should grow dynamically as we add more elements to it. It should **not** have a fixed capacity.

The implementation of the delete method - *del( )* is optional. You do not need this method for the searching and sorting part. It will be worth some extra credit which is 10% of the total grade for the project

To implement `inorder(ostream&)` and `levelorder(ostream&)`, example 1 may prove helpful to you.

In the method `search(Key&)` , if you cannot find the key, it returns a NULL, else it returns the node containing the key.

Do not change the two operator overloading functions - `operator<<` and `operator>>` given to you.

Part 2: Populating Red Black Tree with words from URLs

This is the part you will make use of your RBTree for the first time. You will be implementing the ***scan.cpp*** file. In short, we will be going through a list of url, get the document from that url, parse the document to get the words and then insert them in the RBTree. Each node for this RBTree will be of following type-

	What are we storing	Data type
Key	Words	string
Element	A list of 'urls' with 'frequency' of that particular word	vector<pair<string, int>>

How to get the document from each url:

- ***SimpleHTMLParser.h*** and ***SimpleHTMLParser.cpp*** has method for fetching a HTML file from an URL and parse it to return a 'string of lowercase words separated by spaces'. **You do not need to do anything in these two files.**

    - Note that these files depend on the library libcurl in order to fetch HTML pages from the internet. The path to this library is in the Makefile. If you want to compile on your own machine, you will need to install libcurl and modify your Makefile accordingly.
- ***void parse(string url)*** - parses the HTML file in ***url***
- ***string getText()*** - returns a string of lowercase words separated by spaces

Additionals:
Pair is a simple container defined in C++ STL. For this project you only need to know the following -

- **Syntax :**
    - `pair (data_type1, data_type2) Pair_name;`
    - `pair <int, char> PAIR1 ;`
- **Initializing a pair / creating a pair object :**

```
make_pair(1, 'a');
```

- **Access the elements :**
    - Use variable name followed by dot operator followed by the keyword first or second.
    - Code Snippet:

```
PAIR1.first = 100;
PAIR1.second = 'G' ;
cout << PAIR1.first << " " ;
cout << PAIR1.second << endl ;
```

- You can learn more about pair from here -
    - http://www.cplusplus.com/reference/utility/pair/

What you need to implement in this part:
You will write your code in the ***scan.cpp*** file. Given a url-list and an output file, you need to do the following-

```
1. while there is url in the url-list
     a. parse the url
     b. get the document
     c. process the document, for each word in the document
        i.   search for the word in the RBTree
       ii.   if the word is not in the tree
```

```
                1. create a new list (vector) for this word
                2. add this url in the list with frequency 1
                3. insert this list in the tree
        iii.   Else
                1. get the element(the url-list) from the tree
                2. search for this specific url
                3. if found
                        i.   update the second element
                             (frequency)in the pair
                4. else
                        i.   make a pair object with this url
                             and 1 as freq
                       ii.   insert it in the url-list
2. print the newly created RBTree in the output file with the
   overloaded << operator.
```

Do not change the operator overloading function - `operator<<` given to you.

Turning in milestone 1 (Due 03/19/2018 at 11:59pm)

Execute the following turnin command:
turnin -c cs251 -p project4-1 <your_folder_name>
(Eg: turnin -c cs251 -p project4-1 john_smith)
(Important: previous submissions are overwritten with the new ones. Your last submission will be the official one and therefore graded).
Verify what you have turned in by typing
turnin -v -c cs251 -p project4-1
(Important: Do not forget the -v flag, otherwise your submission would be replaced with an empty one).

Milestone 2: Searching for words and Sorting the result (Due 03/26/2018 at 11:59pm)

In this part of your project, given an input file and a group of words, you will search for words in a RBTree and sort the results based on the frequency. **The number of words will be between 1 to 5 words.** The process is simple, but a little tricky. You will be writing your code for this part of the project in ***search.cpp*** file.

The steps involved in this part are as follows:

1. Read the RBTree from the input file using the overloaded >> operator. The `operator>>` method takes an input stream as an argument. So if you have a filename, you create an inputstream with it (say, fin). Then you read a RBTree (say, wordTree) with this line of code -

   `fin >> wordTree;`
2. While there is word in the search-word-list
   a. Convert the word to lowercase (since, we made our tree with lowercase words)
   b. Search the RBTree for the word
      i. If found
         1. Get the element from the node and put that in a list (a list of **elements** which is `vector<vector<pair<string,int>>>`)
      ii. Else
         1. Put an empty element in the list
3. At the end of last step, we will have a list of the elements (a list of 'list of urls'). But we want to find the common occurrence in webpages. So next thing we need to do is - we need to find the ***intersection*** between all the 'list of urls'. At the end of this step, we will have a single list of intersected urls

   `vector <pair <string, int>>`
4. Now we need to ***sort this list by frequency in descending order.*** The descending order ensures that we are getting the 'most relevant' results first. If two or more URLs have the same frequency, they should be ordered alphabetically by URL, in *ascending* order.

5. Final step
    a. If the 'list of URLs' is empty, you should print `Not Found` with a newline.
    b. print each pair of the sorted 'list of URLs' to *stdout* in the following format:

`URL<space>Frequency`

Additionals:

- Sorting

For sorting, you do not need to implement a sorting algorithm from the scratch. You can use the function template `sort` for that purpose. You need to write a compare function for the `sort` function to use. `vector` class has iterators `begin()` and `end()` which gives access to the begin and last element of the vector. Here is an example of using `sort` function. Say, you have a vector of type `int` and you want to sort it in ascending order by its' value. Take a look at the sample code -

```
bool myfunction (int i,int j) { return (i<j); }   //Ascending
.....................
std::vector<int> myvector;
// Populated with - 32 71 12 45 26 80 53 33
.....................
std::sort (myvector.begin(), myvector.end(), myfunction);
```

Now in this project our vector is of type `pair <string, int>` and the compare function should return true if first pairs' frequency is greater than second pairs' (descending).

A compare function is a binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered greater than the second in the strict weak ordering it defines. The function shall not modify any of its arguments.

- Intersection
    - Say, we have 5 words, so we would have a **`vector`** (list) of **`vector<pair<string,int>>`** with size 5, say, < `a1, a2, a3, a4, a5` >.
    - Now find iteratively the intersection of <`a4, a5`> (store in say, `b1`), then <`b1, a3`> and so on.
    - To find intersection between all **`pair<string,int>`**s in two **`vector<pair<string,int>>`**s (<`a4, a5`>), if only two **`pair<string,int>`** is of equal URL, add the frequencies and put them under same url, and add that **`pair<string,int>`** in the newly build **`pair<string,int>`**(b1).

- Do not change the operator overloading functions given to you.

Testing

You are provided a testing script (testall.sh) to test your program against the reference solutions. Running "testall.sh" will run all the test cases against the reference solutions. You can also test each milestone separately by running "./testall.sh [-p1 | -p2]" (from the tests directory).

Turning in part 2 (Due 03/26/2018 at 11:59pm)

Execute the following turnin command:
turnin -c cs251 -p project4-2 <your_folder_name>
(Eg: turnin -c cs251 -p project4-2 john_smith)
(Important: previous submissions are overwritten with the new ones. Your last submission will be the official one and therefore graded).

Verify what you have turned in by typing
turnin -v -c cs251 -p project4-2
(Important: Do not forget the -v flag, otherwise your submission would be replaced with an empty one).