```
/*
 * Name: Chao Huang, Yeung-Kit Wong
 * Date  : Aug 21, 2016
 * File   : FinalReport.pdf
 */
```

PA3 Final Report

1. Expect running time function for each class's find method

Given: N = number of words, C = number of unique chars in dictionary, D = word length

<**DictionaryBST** class> find() - **O($\log_2 N$)**

In worst case, a BST find method needs to search from root to the farthest leaf. Building the BSTree from a dictionary, it has O($\log_2$N) height. Therefore, it takes O($\log_2$N) to search a word in BST.

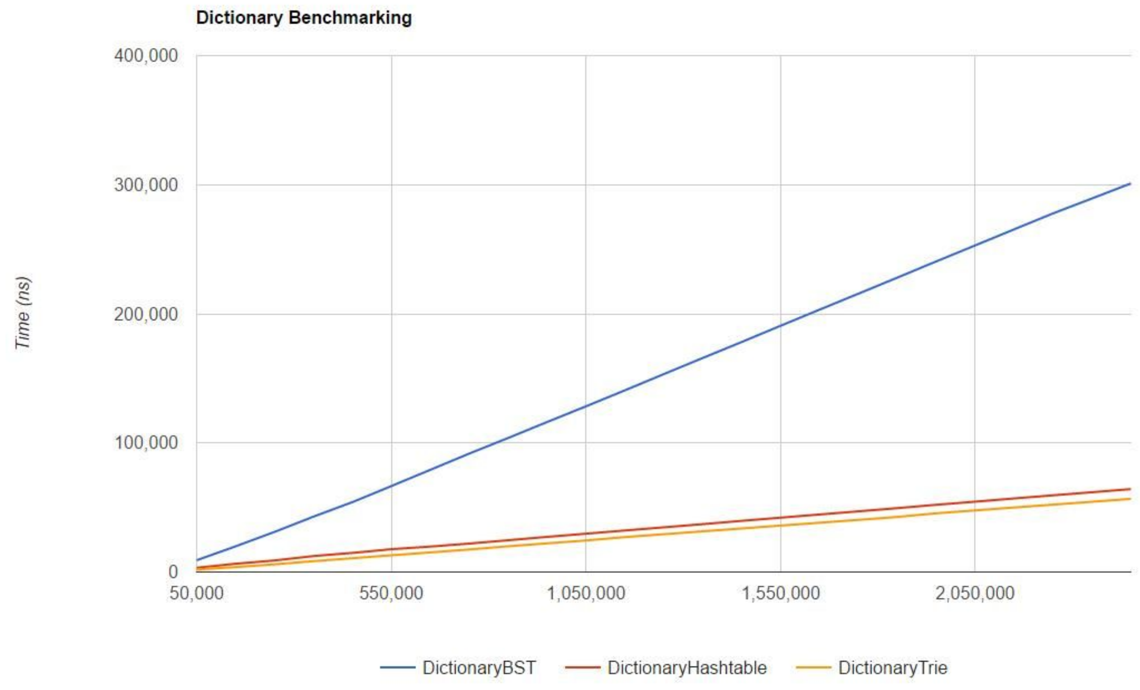
<**DictionaryHashtable** class> find() – **O(1)**

Hashtable takes O(N) to find a element in worst case, but it only happens when too many elements are hashed into the same key or the hashtable is full and has to rehash. In benchdict, we are searching for the words not stored in the hashtable, which is not the worst case of Hashtable search, therefore it takes constant time O(1) to find.

<**DictionaryTrie** class> find() – **O(D)**

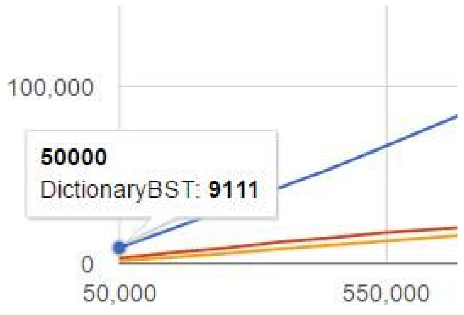
We implemented the Trie class as a multiway trie, each node has 27 children. On average, it takes O(1) to search one character. Therefore, to search a word with length D, on average it takes O(D) which is a constant time, independent of N.

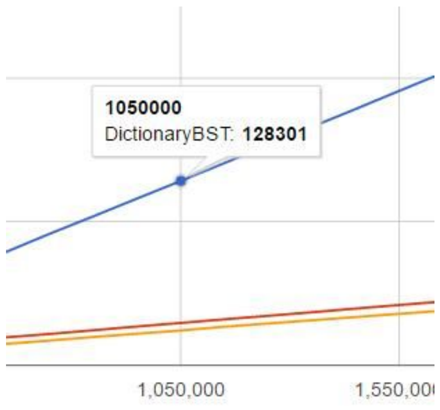
2. Graph Analysis



Runtime Comparison

As represented on the graph, the find() method runtime of DictionaryHashtable and DictionaryTrie class is much faster than DictionaryBST class as we expected.





BST runtime (N = 50000)

BST runtime (N = 1050000)

For DictionaryBST class, when the size of N increases 21 times (from 50,000 to 1,050,000), the runtime only increases 14 times.

Since $\log_2 21 = 4.392$, $14 \approx 3.2(\log_2 21) \in O(\log_2 N)$

For DictionaryHashtable and DicionaryTrie class, it shows that the growth of runtime is independent of the size of N. The line of BST does not look very logarithmic, while Hashtable and Trie also show a trend that they grow a little bit more than O(1) should do. We assume that linear behavior is caused by the necessary runtime for the computer and ieng6 server to execute the program. In addition, we are calculating the runtime of find( ) to find ten words instead of finding one word; the runtime is actually running find( ) method 10 times, therefore the runtime is increased by a factor of 10.

3. Algorithm of predictCompletions method

In our algorithm, we used several things to improve the method's performance:

i. We added an **unsigned int max** member in each Trie node to represent the highest frequency of word in the sub-trie rooted by this node.

ii. Use a STL container **set<Node>** with fixed size **[num_completions]** (number of words needed to be found) to store the **top num_completions highest frequency words** (with the **given prefix string)** found during the trie traversal.

Algorithm (Multiway Trie) :

```
1. Check if the given prefix is an empty string                ----  O(1)
2. Check if there are such prefix in our dictionary trie,      ----  O(D)
      Return a pointer to that node if prefix found
3. Go to the node that represent the prefix
4. Find num_completions highest frequency words recursively
   Recursion( Node* n, set<Node>& s, unsigned int num_completions )
      if n not exist return;
      if Node n is a word
          if less than num_completions words in s, then save n into s  -- O(1)
          else (s has com_completions words)                          --O(1)
             only replace the lowest frequency word in s, if the word in Node n has
             higher frequency
```

```
**if**  less than num_completions stored in s
  **for** each child node c (a-z and space) of Node n              ----O(27)
    **if** c exist, call recursive function Rec(c, s, num_completions)    ----$O(\frac{N}{27^D})$
**else** (s has com_completions words)
    **for** each child node c (a-z and space) of Node n              ----O(27)
      **if** c exist and the max frequency in sub-trie rooted by c is greater than
         the lowest frequency in s
       then call recursive function Rec(c, s, num_completions)        ----$O(\frac{N}{27^D})$
```

During the recursion call, the Node member variable **max** in each Trie node helps reduces lots of runtime. Since we know there is no higher frequency word in certain sub-trie, this algorithm will only traverses a sub-trie that possibly contains a higher frequency word. We can simply skip low frequency sub-tries and continue checking the next sub-trie, until finish traversing the whole Trie.

Expected runtime for predictCompletion: O(D) + 27*O($\frac{N}{27^D}$)

Since we implemented DicionaryTire as a multiway trie, each Trie node represents a character and has 27 child nodes(a-z, and space) containing all the words in dictionary. It takes O(D) to go to the trie node representing the last character of the word to search for completions. Assuming a large enough dictionary, all the words are evenly distributed among 26 characters. As it traverses to the node representing the first character of the word, there are $\frac{N}{26}$ words left to search; and traverses to the second character, there are ($\frac{N}{26} * \frac{1}{27}$) words left to search; third character ($\frac{N}{26} * \frac{1}{27} * \frac{1}{27}$).... Starting to search the child nodes of the last character of the word, there are $\frac{N}{27^D}$ elements left to search. Therefore it takes O($\frac{N}{27^D}$) to finish recursion call for each child node of the last character. For all 27 child nodes of the last character of word to search, that is 27*O($\frac{N}{27^D}$)

O(D) ---to traverse to the last character of word to search completions

27*O($\frac{N}{27^D}$) ---to search all 27 child nodes of last character for possible completions.

Expected runtime = O(D) + 27*O($\frac{N}{27^D}$)