Pavin Wu 26916592

Prac 8: task 1

We will implement the trie with a help of object oriented programming, i.e. we define a class *trie*. To implement the trie, we also need a specialised node object, so we also need to define a class *trieNode*. In *trieNode*, we need to have attributes:

- *self.pointers*: an array of size 26+1 to store a link to the children of the current node. The size 26+1 comes from the size English alphabet + the '$' to denote an end of a string
- *self.max_cNode*: store a location in the *self.pointers* array of a max child of this current node
- *self.max_cNode_freq*: store the frequency of the word constituted with the max child node
- *self.descp*: description of a word. This will be a value other than None only at the node '$'.

Now for the trie class, we need an attribute *self.root* as an object *trieNode*, the method *insertWord()* to insert a word into a trie, and the method *suggestWord()* to get the most likely prediction from the trie based on the prefix input.

To implement *insertWord()*, all we have to do is go down through the nodes of the trie, based on the characters sequentially extracted from the input word. We check for a node to go through by looking up the node in *self.*pointers array of a current node. Only if a node corresponding to the next character does not exist (meaning it's a new word), in *self.pointers* array of the current node, we create a new node at the index corresponding to the order of character in the alphabet e.g. if character of this new node should be 'a', a new node should be created at the index 0. Now, we have a node to go through.

Note that since we also have a requirement to get the word with highest frequency, with smaller alphabetical order, we need to store the frequency of the maximum child character of the current node. These are stored in the variable *self.max_cNode_freq.* This is so we will be able to compare it with future words to find out which have higher frequency. Then update it if the new word is of the higher frequency. We do something similar by storing the index of the maximum child character of the current node. This is stored in *self.max_cNode.* Not only does this allow us to compare alphabetical order, it also allows us to pick the word to go through when we are finding the best word suggestion for a prefix. This will be implemented in *suggestWord()*.

Next, regardless of whether current word is a new word, we set this node we are going to go through as a new current node.

Then we simply repeat this process in a for loop for every character in the input word until we reach the end character. Call this c. Now we no longer have the next character to define where we should put a new node at. However, if there is going to be exact same word inserted into the trie later, but with some extra characters after c, we need a way to tell the program that there is this word with the final character c. Otherwise, it would mean that the only word we have is the word we inserted later (with characters after c). Therefore, by convention, we mark this position with '$' in the last index of *self.pointers* array of the final character node. Also, we update the value of *self.descp* only when we reach this node. This is because we can tell that a word is unique only when we reach the end of the word.

Now, to implement *suggestWord()*, we simply do the same running through process as *insertWord(),* but based on the prefix, with some additional processes after reaching prefix, and additional return statements.

First, we go through trie based on the characters in the prefix input. We return None and 0 when there is no child corresponding to a character in the *self.pointers* of current node, as this means there is no word with the same prefix as the input in the trie. Once we reach the end of the prefix, and there's still no return, it means we have word(s) with the prefix.

So next, we go the trie forward, but based on the value stored in *self.max_cNode* of the current node instead. We go through in the same manner until *self.max_cNode* points to the character '$'. We then return the word we have gone through as the suggested word for this prefix input.

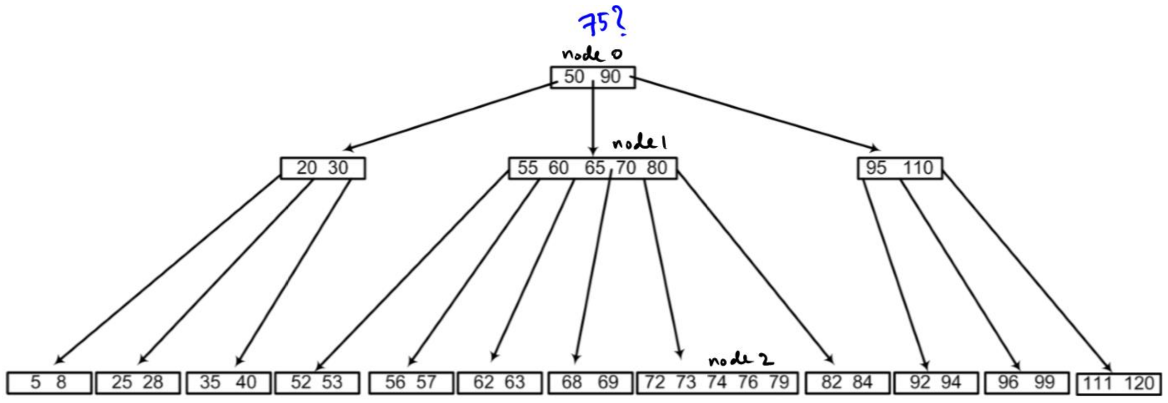
Worst-case space: space used in a trie depends on the number of repeating prefixes. Let total number of repeated prefixes be P, total number of unique characters after prefix be U, and the total number of characters in every definition of every word be D, then the total space complexity is O(P+U+D).

Worst-case time: For insertion method, we need to go through all characters of all words in the dictionary (excluding definition). So O(T) is the total time complexity for insertion where T is total number of words.

For suggestion method, we need to go through all the characters in the prefix, then all remaining character of the word with the highest frequency, then the definition of the word. So if we let these M and N, the complexity is O(M+N). Therefore, O(M+N) is the overall time complexity for suggestion.

Prac 8: task 2

a) insert 75



1. First check node 0, we see we must go down middle child to reach position for 75 (since 50 < 75 < 90)

   This middle child is full (since maximum number of keys in a node is 2t-1=5). So we need to split.
   To split, we take median: 65, and insert it to the parent node in the position of the link. Node 0 is now changed.
   Now node 1 is split into node 1.1, and node 1.2 with the links from node 0.

2. From the new node 0, we see we need to go to node 1.2 to reach position for 75.

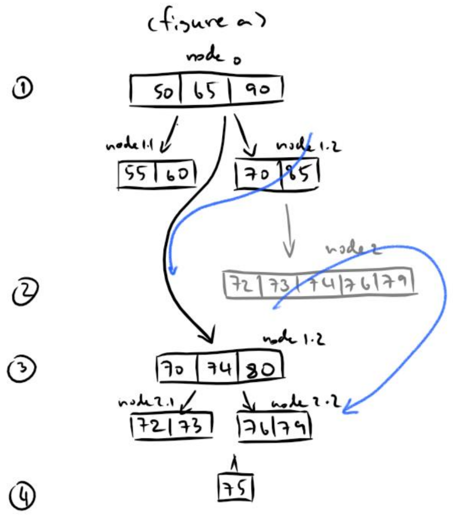
   From the node 1.2, we see we need to go to node 2.

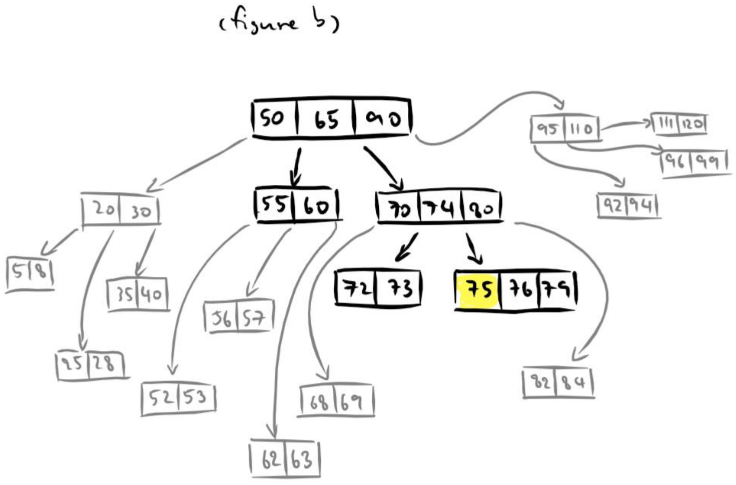
3. Again, this node is full, so we need to split if we want to insert a key.
   To split, we take median: 74, and insert it to its parent node in the position of the link. Node 1.2 is now changed.

   Now node 2 is split into node 2.1, and node 2.2 with the links from node 1.2.

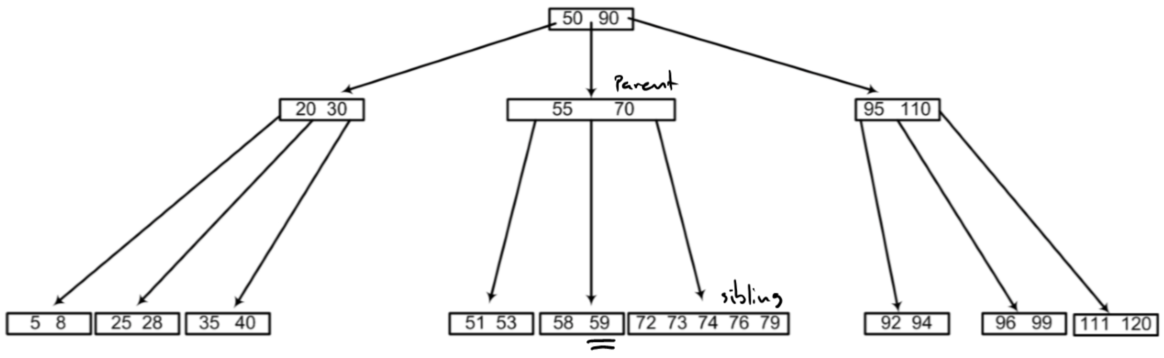
4. From the new node 1.2, we see we need to go to node 2.2 to reach position for 75.
   Since this node is not full, we insert 75 in its relevant position.

   The process can be illustrated in figure a, and the final tree in figure b:





b) delete 59



Looking at the tree, we see 59 is in the child node with a poor parent, and a rich sibling. This corresponds to the case 3a from lecture notes.

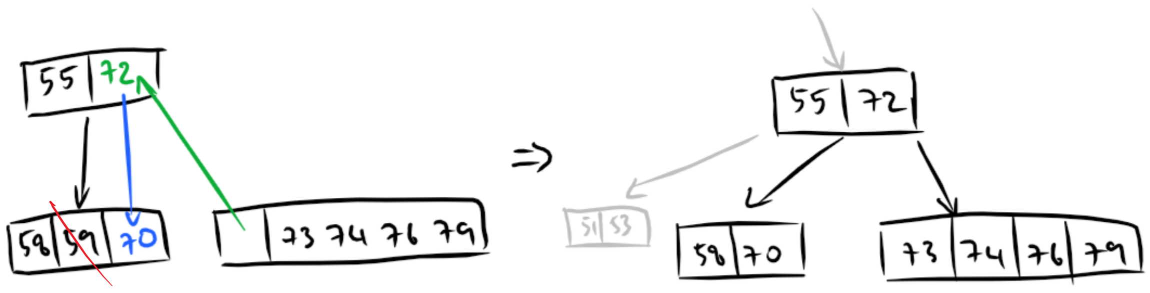
To allow deletion of 59 from a poor node, we need an extra key to fill its place. Since the key from the rich sibling is greater than the immediate parent key (70), we cannot use the extra key from rich sibling.

Therefore, we have to use the immediate parent key. We achieve this by 1. we insert the immediate parent key (70) to the poor child node with 59. This child node is no longer poor, but the parent is now underflowed.

Now, 2. we take the smallest key from the rich sibling (72), and 3. add that to the position of the moved immediate parent. The parent now has one additional key, so it is no longer violating the constraint requiring the number of keys in the node to be lower than t-1=2.

4. Since the child node with 59 is no longer poor, and no other nodes will violate the constraints, we can safely delete 59.

The process can be illustrated by the diagram below. Note that all other nodes and links in grey or not shown will not be changed:


55 | 72
58 | ~~59~~ | 70
73 74 76 79
=>
55 | 72
51 53
58 | 70
73 | 74 | 76 | 79