Data Structures Project 1

Jakub Podmokly
Nhi Pham Le Yen
Lab Section: Wednesday 1:15pm - 3:15pm

February 2019

1 Project

- Our project file includes **main.cpp**, **binarySearch.cpp**, **binarySearch.h**, **Makefile**, and the **Dictionaries.**
- The main function in the whole project is the function **binarySearch** (because the dictionary is already sorted), which optimizes the time complexity from O(n) to O(logn) by reducing the number of comparisons needed to find a word.

2 Algorithm

- There are three key functionalities, which we divide into 3 types:
  - Type 1: search for a *full word*.
  - Type 2: search for and list the words that match the *prefix* $(*)$.
  - Type 3: search for and list the words that match the *prefix* and *postfix* (?).
- For our function **binarySearch**, it takes 8 arguments: vector$<$ *string* $>$data[i], string word, int left, int right, int comparisons, int number, int pos, int type.

  - *vector< string >*data[dataPos]: contains the words in the dictionary. The index *dataPos* ranging from 0 to 26, which means that from 0 - 25 represents the letter beginning with a-z, and the last one to store the special characters.
  - *word*: the word that the user inputs.
  - *left*: the first element in the interval we choose to find the word.
  - *right*: the last element in the interval we choose to find the word.
  - *comparisons*: number of comparisons that are carried out to find the matching word.
  - *number*: the maximum number of words the user wants to output (in case 2 and 3).
  - *pos*: the pos of $*$ in type 2, or pos of ? in type 3. Pos has the value of -1 in type 1.
  - *type*: determines the type 1, 2, or 3.

- Before we call the recursive function binarySearch, we find the vector that contains all the words beginning with the same character with the searchString that the user inputs in. The purpose of this is to reduce the range we need to search for the word.

- We implement recursive binarySearch function in our project.
  - If (right < left), there is no match, and we output "word not found."
  - Find the middle element (mid) , and compare the word with mid.
  - If the word == mid:
    - If **type 1**, we output "Word found" and number of comparisons.
    - Else if **type 2** or **type 3**, we run 2 while loops to the left and to the right to find the range that matches the prefix.
      - If type 2, from the range we find, we compare with the maximum number we need to output to print the maximum possible number of words needed and print the number of comparisons. The counting for the number of comparisons is stopped when we reach the first element that

matches in the binary search, not necessarily the first element in the dictionary that matches the prefix.

    · If type 3, from the range we find, we go through the range to find if there is any words that match the postfix and increase the number of comparisons every time we compare. We do linear search in this step.

- Else:
    - if word < data[mid], we call function binarySearch for the range (left, mid-1)
    - else if word > data[mid], we call function binarySearch for the range (mid+1, right)