EECE 2560: Fundamentals of Engineering Algorithms

Project #3 - Part I

Write a program that solves a word search puzzle. The program reads an $n \times n$ grid of letters from a file and prints out all the words that can be found in the grid. Words can be found in the array by starting from any letter and reading left, right, up, down, or along any of the four diagonals. Words can also wrap around the edges of the array. Letters can be used more than once in a particular word. Words must be at least 5 characters long.
The list of k possible words is included in the file `wordlist`. Several sample word search puzzles are also provided.
The goal is to find an algorithm that solves this problem that runs as quickly as possible for large n and k.

1. Implement a class called `wordList` that stores the word list in a vector, and which includes:
   (a) a function to read the words from the wordlist file
   (b) an overloaded output operator to print the word list
   (c) functions that sort the words using 1) insertionSort, 2) quickSort, and 3) mergeSort
   (d) a function to handle word lookups using binary search
2. Implement a class called `grid` that reads the letters in the grid from a file and stores them in a matrix.
3. Implement a **global** function `findMatches()` that is passed the word list and the grid as parameters and which prints out all words that can be found in the grid.
4. Implement a **global** function `search(int)` which reads the name of the grid file from the keyboard and prints out all words from the word list that can be found in the grid. The function should also print out the CPU time to sort the words, the CPU time to find the words, and the total time. The integer parameter is used to select the sorting algorithm used.

The code you submit should demonstrate the correct behavior of your code using the three sorting algorithms. After each sorting algorithm sorts the original word list, print out all the words that can be found in the grid and the timing information described above.

Project #3 - Part II

In this part, evaluate the effectiveness of sorting algorithms based on heaps and hashtables.

1. Implement the template class `heap<T>` that stores objects in a heap of type `vector<T>`, and which includes:

   (a) functions `parent(int)`, `left(int)`, `right(int)`, and `getItem(int n)` which returns the nth item in the heap.

   (b) for a max-heap, functions `initializeMaxHeap()`, `maxHeapify()`, and `buildMaxHeap()`.

   (c) the equivalent functions for a min-heap. You can implement the max-heap and min-heap data structures within the same class and convert the stored data into a min- or max-heap by calling the appropriate member functions.

   (d) function `heapSort()`

   Since the heap is only used to sort the word list: You can declare the heap within the `wordList::heapSort` function, copy the unsorted words into the heap, sort the words, and then copy the words out. Then the `wordList::binarySearch` function can be used to look up words.

2. Implement the template class `hashTable<T>` that stores objects in a hash table of type `vector<vector<T> >`, and which includes:

   (a) functions `addItem()`, `deleteItem()`, and `inList()`

   (b) function `hash()` which returns the hash value for an item Since the Hash Table will be used to look up words, you can copy the unsorted words into the hash table, and then the `hashTable::inList` function can be used to look up words. You can make `findMatches` a template function that can be passed data structures of different types.

3. Measure the runtimes of word search using all the sorting algorithms for all the sample grids. **Write a short paragraph summarizing the algorithms' relative performance and submit it with your project**.