**Programming Assignment 3**

Penalty for late submission: 10% less credit/day.

Each student group should choose a collection of problems with total number of points not exceeding 120. All points you gain above 100 count for extra credit and may compensate your lower performance on programming assignments 1,2.

Implement the following:

Dictionary ADT (4.5-4.8) with open hash tables (fig. 4.12) for problem 1 and closed hash tables (fig 4.14) for problem 2. Verify experimentally the following statements. (INSERT, DELETE, MEMBER, MAKENULL)

1.     (20 points) For Dictionary ADT implemented with open hash tables the average number of probes required to make either a deletion or an insertion is O(1+a). Find best numerical constants for deletion and insertion.

2.     (20 points) For Dictionary ADT implemented with closed hash tables and the linear hashing as the rehash strategy the average number of probes required to make a deletion is approximately equal to (1+1/(1-a))/2 and the average number of probes required to make an insertion is approximately equal to (1+1/(1-a)2)/2.

In statements of problems 1 and 2 a = N/B is the load factor, N is the number of elements in the set, and B is the size of the bucket/hash table. It is recommended that you use library hash functions in your experiments.

3.      (20 points) Dictionary ADT with BST's (5.1-5.2, figs 5.2-5.5). Verify experimentally that the average number of nodes on the path from the root to another node, in a random BST obtained by an iteration of INSERT operation is O(log2n), where n is the number of items inserted into an initially empty BST. Assume that all permutations of elements are equally probable. It is recommended that you use library generators of random permutations.

4.     (20 points) Trie (5.3) with a list representation for trie nodes (pages 167-168). Store the strings of the document [**Alice in Wonderland**](https://www.cs.drexel.edu/~knowak/cs260_summer_2017/alice30.txt) in your trie by iterating the INSERT operation and print out the size of the resulting trie (provide the exact definition of size that you use in your count).

5.      (10 points) Dijkstra's shortest paths algorithm (6.3) with the adjacency matrix as the representation of the graph. Test your implementation on the graph of problem 6 of review 2.

6.     (30 points) Dijkstra's shortest paths algorithm (6.3) with a partially ordered tree as a priority queue and linked adjacency lists as the representation of the graph. Test your implementation on the graph of problem 6 of review 2.  
An implementation in C is described in section 9.8 of our CS 270 textbook *Foundations of Computer Science* by Alfred Aho and Jeffrey Ullman.  
book chapters: http://infolab.stanford.edu/~ullman/focs.html   
C code: http://infolab.stanford.edu/~ullman/fcsc-figures.html

7.     (10 points) Floyd's algorithm (6.4) with the option of recovering the paths. Test your implementation on the graph of problem 6 of review 2.

8.     (10 points) Depth-first search algorithm (6.5) together with depth-first numbering of nodes. Test your algorithm on the graph of fig. 6.38, page 226.

9.     (30 points) Fast Fourier Transform in two versions ([W] 2.5): dyadic size ([W] p.53) and any size ([W] p.55). Verify correctness of your implementations by comparing the results of your program with the results obtained with FFT's of standard mathematical packages.  
Symbol [W] refers to the book *Algorithms and Complexity* by Herbert Wilf.

Each of your programs should come together with testing data and the code running the tests. If you submit programs corresponding to problems 1,2 or 3 you need to include the data you were using in your experiments and a document containing tables with the display of your numerical results.

## Programming Assignment 3 Submission Guidelines

Due date and instructions are available on the course website: https://www.cs.drexel.edu/~knowak/cs260\_summer\_2019/cs\_260\_links.htm

Please follow the submission guidelines below. There are specific instructions for each problem. Failure to do so may result in points being deducted. If you have any questions, please email Reza (rm976@drexel.edu) or ask at office hours.

* 1. Submit via [learn.drexel.edu](http://learn.drexel.edu/)
     + **Only the group leader submits**
     + **To submit, go to the "Assignments" link and select "Programming Assignment 3"**
     + You may re-submit until the due date - your latest submission will be the one graded
     + If you re-submit, please upload ALL of your files (not just the ones you updated)
     + Do not put comments in the text area with the submission, they will not be read
     + **You must compress your files into a single .zip or .tar.gz**
     + DO NOT email submissions
  2. **Submit a file called problems**
     + A plain text file called **problems** (not Problems, nor problems.txt, etc)
     + Indicate which problems you want graded via a newline-delimited list of problem numbers
     + For example, if I did problems 1, 2, 4, 5, 6, 7, and 8 I would submit this file:
       - [problems](https://www.cs.drexel.edu/~eb452/CS260summer18/problems)
     + Do not include anything else in this file (not even your name)
     + **If your list of problems exceeds 120 points, I will arbitrarily choose which problems I want to grade**
  3. **Submit a file called group**
     + A plain text file called **group** (not Group, nor group.txt, etc)
     + Indicate your group members via a newline-delimited list of usernames (including yours)
     + For example, if I worked in a group with abc123 and xyz789, I would submit this file:
       - [group](https://www.cs.drexel.edu/~eb452/CS260summer18/group)
     + Do not include anything else in this file (not even your names)
  4. **Submit a README**
     + A plain text file named **README** (not readme, nor README.txt, etc)
     + Include anything you want me to know before I grade
     + Clearly note which features of your program do and do not work, as well as any design decisions unique to your implementation
  5. **Submit a Makefile**
     + A plain text file named **Makefile** (not makefile.txt, nor Makefile.mak, etc)
     + All compiled files (executables and .o) will be deleted before grading. Your Makefile should compile and give execute permissions (if necessary) and run your program.
     + **Include targets for the problems you did as specified below**
     + For examples of python and C++ makefiles that run a program based on multiple files, see [Makefile](https://www.cs.drexel.edu/~eb452/CS260summer18/A3/make)
  6. **Document your source code**
     + Comment your code so that a programmer could figure out how to use your data structures (describe parameters, return values, etc)
     + Don't go overboard - it is not necessary to comment every line of code
  7. **All code MUST run on tux.cs.drexel.edu**
     + If your code does not run on tux, then it does not run for grading purposes
     + You may not use any libraries that are not standard on tux

1. **Do not under any circumstances prompt for user input**

## Specific Instructions for Each Problem

* 1. **Dictionary - Open Hash**
     + Put implementation in **openhash.cpp** or **openhash.py**
     + Makefile target **openhash** should run your experiments
  2. **Dictionary - Closed Hash**
     + Put implementation in **closedhash.cpp** or **closedhash.py**
     + Makefile target **closedhash** should run your experiments
  3. **Dictionary - Binary Search Tree**
     + Put implementation in **bst.cpp** or **bst.py**
     + Makefile target **bst** should run your experiments
  4. **Trie**
     + Put implementation in **trie.cpp** or **trie.py**
     + **Your program should read from a file called**[**alice30.txt**](http://www.umich.edu/~umfandsf/other/ebooks/alice30.txt)**in the current directory, split at spaces, build the trie, and output its size**
     + You may optionally strip punctuation including the # sign
     + Makefile target **trie** should run your program
  5. **Dijkstra with Adjacency Matrix**
     + Put implementation in **dijkstra.cpp** or **dijkstra.py**
     + **Hard code the graph of**[**review 2 problem 6**](https://www.cs.drexel.edu/~knowak/cs260_winter_2018/review_2.pdf)**as an adjacency matrix**
     + **Output the shortest distance (D) for each vertex, using vertex 1 as the source**
     + Makefile tarket **dijkstra** runs your program
  6. **Dijkstra with Heap and Adjacency Lists**
     + Put implementation in **dijkstra2.cpp** or **dijkstra2.py**
     + **Hard code the graph of**[**review 2 problem 6**](https://www.cs.drexel.edu/~knowak/cs260_summer_2018/review_2.pdf)**as linked adjacency lists**
     + **Output the shortest distance (D) for each vertex, using vertex 1 as the source**
     + Makefile tarket **dijkstra2** runs your program
  7. **Floyd**
     + Put implementation in **floyd.cpp** or **floyd.py**
     + **Hard code the graph of**[**review 2 problem 6**](https://www.cs.drexel.edu/~knowak/cs260_summer_2018/review_2.pdf)**as an adjacency matrix**
     + **Output the shortest distance and predecessor matrices (A and P)**
     + Makefile tarket **floyd** runs your program
  8. **Depth-first Search**
     + Put implementation in **dfs.cpp** or **dfs.py**
     + **Hard code the graph of fig 6.38 in the graph representation of your choice**
     + **Output the depth-first numbering of vertices, using vertex a as the source**
     + Makefile tarket **dfs** runs your program

1. **Fast Fourier Transform**
   * Put code in **fft.cpp** or **fft.py**
   * **You may have two seperate functions for dyadic and arbitrary input sizes**
   * Makefile target **fft** runs your test code