CS 2420 Section 2 and Section 3
Spring 2016
Assignment 6: Heaps (Priority Queues)
Due: 11:59 p.m. March 30 (Wednesday), 2016
Total Points: 100 points

Part I [15 points; 5 points per question]: Submit your solution via Canvas.

1. Insert a new key 18 into the heap shown in Figure 1. Show all the necessary intermediate results (e.g., after inserting the new key 18 and after each necessary UpHeap swap operation) and draw the final new heap after the insertion. Use an array as discussed in class to store this heap starting with index 1. Show all the intermediate results in the array and the final array after the insertion.
2. Perform the DeleteMin operation on the heap shown in Figure 1. Show all the necessary intermediate results (e.g., after removing the smallest key and after each necessary DownHeap swap operation) and draw the final new heap after the DeleteMin operation. Use an array as discussed in class to store this heap starting with index 1. Show all the intermediate results in the array and the final array after the DeleteMin operation.
3. Use the linear-time algorithm (i.e., the O(n) algorithm) discussed in class (refer to slides 32-37 of the class notes for details) to build a heap from the following elements: 40 25 14 75 10 34 8 33 60 28 17 3 19 6 42 15. Figure 2 shows a complete binary tree formed by these numbers. Show all the necessary intermediate results (e.g., after each necessary DownHeap swap operation) and draw the final heap generated by the linear-time heap construction algorithm. Use an array as discussed in class to store this heap starting with index 1. Show all the intermediate results in the array and the final array after the linear-time heap construction operations.

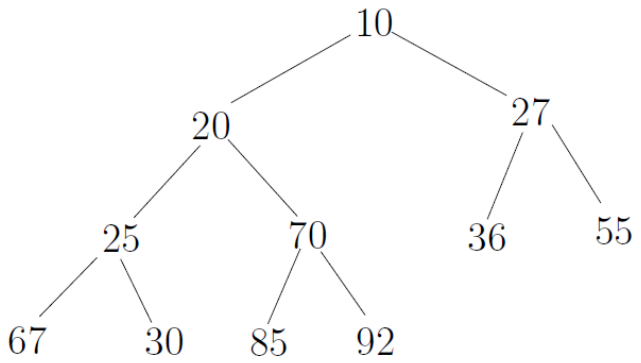


Figure 1: A Heap

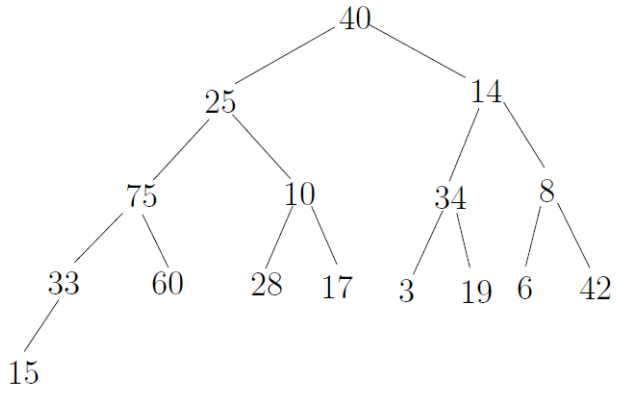
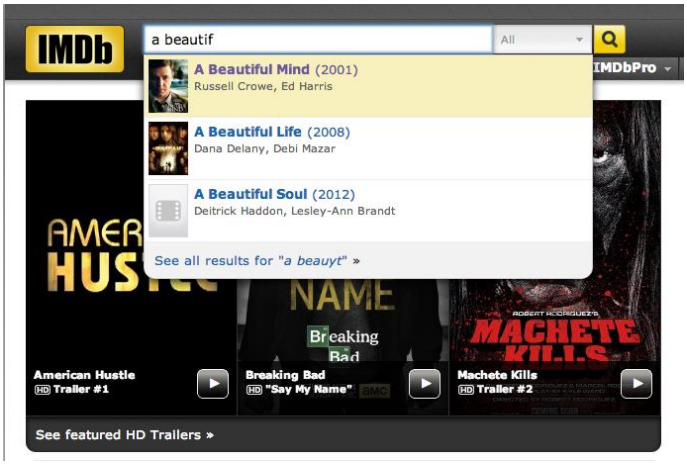
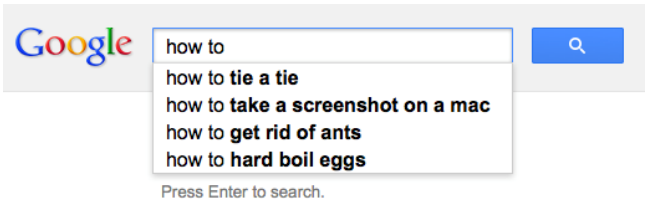


Figure 2: A Complete Binary Tree

Part II [85 points]:

Autocomplete is an important feature of many modern applications. As the user types, the program predicts the complete *query* (typically a word or phrase) that the user intends to type. *Autocomplete* is the most effective when there are a limited number of likely queries. For example, the Internet Movie Database uses *autocomplete* to display the names of movies as the user types; search engines use *autocomplete* to display suggestions as the user enters web search queries; cell phones use *autocomplete* to speed up text input.





In these examples, the application predicts how likely it is that the user is typing each query and presents to the user a list of the top-matching queries, in descending order of weight. These weights are determined by historical data, such as box office revenue for movies, frequencies of search queries from other Google users, or the typing history of a cell phone user. The performance of *autocomplete* functionality is critical in many systems. According to one study, the application has only about 50*ms* to return a list of suggestions for it to be useful to the user. Moreover, in principle, it must perform this computation *for every keystroke typed into the search bar* and *for every user*!

In this assignment, your task is to implement a simple *autocomplete* system for a given set of *N* strings and nonnegative weights. That is, given a prefix or two prefixes, quickly produce top *k* possible strings in the set that start with the prefix or prefixes (e.g., prefix a or prefix b for the case of two prefixes), in descending order of weight (e.g., a larger weight has a higher priority).

You will have access to a file named "SortedWords.txt", which stores a set of all possible queries and associated weights **(and these queries and weights will not change).** The first line lists the number of all possible queries. The remaining lines list a sorted word followed by its associated weight line by line.

[40 points] You must create a max-heap (a.k.a., a max priority queue) class, where the key of each node is larger than the key of either of its children and the heap is stored in an array as discussed in class starting with index 1. This max-heap supports the following public methods:

- **[2 points]** MaxHeap: Generate a max-heap using an array of a specified size, set the array size, and initialize the heap size as 0.
- **[2 points]** MaxHeap: Generate a max-heap using an array of a specified size, set the heap size and the array size, sequentially copy a set of elements into the array starting from index 1, and build the heap using the linear-time algorithm as discussed in class. It may call PercolateDown function to make sure that the heap-order property is maintained. Note: These two MaxHeap functions are overloading functions (i.e., they have different input parameters as shown in MaxHeap.h).
- **[1 point]** ~MaxHeap**:** Destructor.
- **[2 points]** Insert: Insert a new entry containing a query and its associated weight into the heap (assuming the new entry is not in the heap). Here, weight is considered as the key and a larger weight means a higher priority.
- **[3 points]** DeleteMax: Find, return, and remove the entry with the maximum weight in the heap.
- **[5 points]** PrintHeap: Print the heap prettily so the tree structure can be seen on the screen.

- **[7 points]** Merge: Combine the current heap with another heap to form a larger heap when two prefixes are input by the user. You may need to release the dynamic array and reallocate enough memory spaces for a large enough dynamic array to hold the two combined heaps.
- **[5 points]** FindTopMatches: Return the top *k* words with the prefix matching the target word.

Several private functions may be needed:

- **[10 points]** PercolateDown: Perform the DownHeap operation. It is preferably called in MaxHeap and DeleteMax functions to make the implementation more concise and modularized.
- **[1 point]** Left: Find the index of the left child of a given node.
- **[1 point]** Right: Find the index of the right child of a given node.
- **[1 point]** Parent: Find the index of the parent of a given node.

Below is the MaxHeap.h file, which contains C++ Max-Heap class interface, for your reference.

```
class MaxHeap
{
   private:
        int arraySize ;      // the size of the array, index from 0 to array_size-1
        int heapSize ;       // number of elements in the heap; heapSize is smaller than arraySize
        Element * H ;        // elements of heap are in H[1]…H[heapSize], cell at index 0 is not used
        int Left(int i) ;    // return the index of the left child of node i
        int Right(int i) ;   // return the index of the right child of node i
        int Parent(int i) ;  // return the index of the parent of node i
        void PercolateDown(int) ;  // DownHeap method.  It will be called in MaxHeap and DeleteMax

   public:
        MaxHeap(int arraySize=30) ;  // Generate an empty heap with the default array size of 30.
        MaxHeap(Element *A, int heapSize, int arraySize) ;  // A contains a sequence of elements
        ~MaxHeap() ;
        void Insert(const Element &a) ;   // Insert a new element containing word and its weight
        Element DeleteMax();    // Find, return, and remove the element with the maximum weight
        void PrintHeap();   // Print the heap in tree structure; each node containing word and weight
        void Merge(const MaxHeap &newHeap) ;  // Merge with another heap to form a larger heap
        Element * FindTopMatches(int count) ;  // return top "count" matching words based on weights
};
```

[45 points] Write a driver code in main program to perform the following tasks:

1. **[15 points]** Demonstrate the correctness of all the public member functions using a reasonable number of data. Specifically, you must create a small test case to verify your priority queue works perfectly by using various combinations of at least the following operations:
   - Create an instance of the queue using the first constructor in MaxHeap.h
   - Call Insert function to insert a new element into the queue.
   - Call Insert function multiple times to insert different new elements into the same queue.
   - Call PrintHeap function to print the queue prettily after each insert operation
   - Call DeleteMax function to remove the element with the maximum weight
   - Call PrintHeap function to print the queue prettily

- Call DeleteMax function to remove the element with the maximum weight
- Call PrintHeap function to print the queue prettily
- Call FindTopMatches to return top "count" matches for one prefix
- Create another instance of the queue using the second constructor in MaxHeap.h to insert a sequence of new elements into the queue
- Call PrintHeap function to print the queue prettily
- Call DeleteMax function to remove the element with the maximum weight
- Call PrintHeap function to print the queue prettily
- Call DeleteMax function to remove the element with the maximum weight
- Call PrintHeap function to print the queue prettily
- Call FindTopMatches to return top "count" matches for one prefix
- Call Merge function to merge the two queues
- Call PrintHeap function to print the queue prettily
- Call FindTopMatches to return top "count" matches for two prefixes

The grader will use his own driver code to test the correctness of all the public member functions.

2. **[5 points]** Read in the file "SortedWords.txt" and store all possibly queries and their weights in a dynamic array of Element, where the size of the array is the same as the number stored in the first line of the file and Element is an ADT containing a possible query (word) and its weight.
3. **[5 points]** Create a simple user interface that allows the user to choose whether running *autocomplete* or quitting *autocomplete*. When running *autocomplete*, it allows the user to input (a) the prefix of a word (e.g., goo) or the prefixes of two words (e.g., goo & te) (b) the number of outputs s/he wants to see (e.g., count). When quitting *autocomplete*, the program terminates. Feel free to add other user options if you want.
4. For each target word (of length *r*) and provided count, perform the following operations:
   4.1. **[5 points]** Apply the binary search algorithm on the array to find the index of a word whose first *r* letters match the target word.
   int BinarySearch(Element a[], int arraySize, string target)
   4.2. **[3 points]** Use the returned index from BinarySearch (e.g., foundIndex) to find the first location of word in a[] whose first *r* letters match the target word.
   int FindFirstIndex(Element a[], int arraySize, int foundIndex, string target)
   4.3. **[2 points]** Use the returned index from BinarySearch (e.g., foundIndex) to find the last location of word in a[] whose first *r* letters match the target word.
   int FindLastIndex(Element a[], int arraySize, int foundIndex, string target)
   Note: After calling the above three functions, the return value from calling FindFirstIndex is the first location of word in the array (i.e., the smallest index) whose first *r* letters match the target word and the return value from calling FindLastIndex is the last location of word in the array (i.e., the largest index) whose first *r* letters match the target word. In other words, using these two returned values (i.e., the smallest index and the largest index), we can find all words whose first *r* letters match the target word.
   4.4.**[5 points]** Put all elements found to be matching the target word into a priority queue. Using the weight as a priority, output the top "count" terms.
   4.5. **[5 points]** Allow merging of two queues to return the top "count" terms for two queries.

The output will look something like the following. Black letters are printed by the program, bolded blue letters are the input from the user, and green letters are the output from the *autocomplete* program.

```
****************************************
Menu:
    S:  Search top words with one prefix or two prefixes
    Q: Quit Search
****************************************
S
Please input your prefix or prefixes: acc
Please input the number of top matches you want to return: 9
  according
  accept
  account
  access
  accident
  accuse
  accomplish
  accompany
  accurate
****************************************
Menu:
    S:  Search top words with one prefix or two prefixes
    Q: Quit Search
****************************************
S
Please input your prefix or prefixes: acc & all
Please input the number of top matches you want to return: 11
    all
    allow
    according
    accept
    account
    access
    accident
    accuse
    accomplish
    ally
    accompany
****************************************
Menu:
    S:  Search top words with one prefix or two prefixes
    Q: Quit Search
****************************************
Q
Thank you for using autocomplete software developed by xxxx.
```