Homework Assignment: Heap based Priority Queue Algorithm Analysis and Implementation

Mauricio Estrella

September 30, 2025

1 Task 1 - Data Structure Construction

Design a heap-based priority queue to store some English words by modifying the algorithms in the lecture note. The data structure should support the following operations:

- Report the number of words in the data structure
- Remove the first word in dictionary order if the data structure is not empty
- Search for a specific word and return its index or -1
- Add a new word if it is not already in the queue (otherwise do nothing)

1.1 Data Structure Design

The core of this data structure is a binary min-heap. This heap will be implemented using a 1-indexed array to simplify the calculation of parent and child node indices. The following table shows the signature of each method.

Variables:

- private String[] pq: An array to store the words. We use 1-based indexing (index 0 is unused), so pq[1] is the root of the heap. The word at the root is always the smallest in dictionary order.
- private int n: A counter that holds the current number of words in the priority queue.

Method	Description	Time Complexity
size()	Returns the size of priority queue	O(1)
search(Key word)	Return index of word or -1 if not in queue	O(n)
add(Key word)	Add a word to priority queue if it does not already exist	O(n)
extractMin()	Remove the first word in dictionary order if the data structure is not empty	O(log n)
sink(int k)	Helper method to restore the heap invariant	N/A
swim(int k)	Helper method to restore the heap invariant	N/A
exch(int i, int j)	Helper method to swap	N/A
greater(int i, int j)	Helper method to compare	N/A
resize(int capacity)	A helper method for resizing an array (used when pq[] is full)	N/A

Table 1: Word Priority Queue

Algorithm 1 size()

```
  Function SIZE()
1: return $n$
```

Algorithm 2 Extract Minimum

```
Function ExtractMin()
1: if pq is empty then
2:     return -1
3: min = pq[1]
4: Exch(1, $n--$)
5: Sink(1)
6: return min
```

Algorithm 3 Search

```
Function Search(word)
1: return searchRecursive(1, word)
```

Algorithm 4 SearchRecursive

```
Function Search($k$, $word$)
1: if $k > n$ then
2:     return -1
3: if pq[k] equals $word$ then
4:     return k
5: if pq[k] > $word$ then
6:     return -1
7: leftSearchResult = SearchRecursive($2 * k$, $word$)
8: if leftSearchResult != -1 then
9:     return leftSearchResult
10: return SearchRecursive($2 * k + 1$, $word$)
```

Algorithm 5 Add

```
Function Add(word)
1: if Search(word) != -1 then
2:     return
3: if $n == pq.length - 1$ then
4:     resize($2 * pq.length$)
5: pq[$++n$] = $word$
6: Swim($n$)
```

2 Analyzing Time Complexity of each Method

1. Report the number of words
   - **Algorithm**: The operation simply returns the value of the counter variable n.
   - **Method Name**: size()
   - **Worst-Case Time Complexity**: O(1) - This is a constant time operation as it only involves retrieving the value of a variable.
2. Remove the first word in dictionary order if the data structure is not empty
   - **Algorithm**: This is the standard extract-min operation for a min-heap.
     (a) Retrieve the word at the root (pq[1]), which is the smallest element.
     (b) Swap the root element with the last element in the heap (pq[n]).
     (c) Decrement the word count n.
     (d) Restore the heap property by "sinking" the new root element down to its correct position. The sink operation repeatedly compares the element with its children and swaps it with the smaller child until the heap property is satisfied.
   - **Method Name**: extractMin()
   - **Worst-Case Time Complexity**: O(log N) - The complexity is determined by the sink operation, which traverses the height of the heap. The height of a balanced binary heap with N nodes is log N.
3. Search for a specific word and return its index or -1
   - **Algorithm**: Searches for a word using a recursive heap-style traversal. It traverses the heap like a tree and prunes branches where the target word cannot exist.
     (a) Start at the root (index 1).
     (b) If the current node's word is what we're looking for, we're done.
     (c) If the current node's word is greater than the word we're searching for (e.g., current is "dog", search is "cat"), we can stop searching that entire branch. Because of the min-heap property, all children below "dog" must be greater than or equal to "dog", so they cannot possibly be "cat". This is how we "prune" the search.
     (d) If the current node's word is smaller than the search word, the target word could still be in the branches below, so we continue searching recursively on its children.
   - **Method Name**: search(Key word)
   - **Worst-Case Time Complexity**: O(N) - If you are searching for a word that is large alphabetically (like "zebra"), you will likely never be able to prune any branches and will have to visit every single node.
4. Add a new word if it is not already in the queue (otherwise do nothing)
   - **Algorithm**: This operation combines a search with a standard heap insertion.
     (a) Perform a linear scan of the array pq from index 1 to n.
     (b) If the word is found, terminate the operation.
     (c) If the word is not found, it is added to the end of the heap at index ++n. Then, the heap's structure is restored by "swimming" the new element up to its correct position. The swim operation repeatedly compares the new element with its parent and swaps them if the new element is smaller.
   - **Method Name**: add(Key word)
   - **Worst-Case Time Complexity**: O(N) - The search step requires checking every element in the worst case, resulting in O(N) complexity. The heap insertion (swim) takes O(log N) time. The overall complexity is dominated by the search.

A word on our search's time complexity: O(N). To improve our data structure's efficiency we could augment it with a Hash Map to provide constant time, O(1) searching. This map will store each word and its current index in the pq array, thus, instead of scanning the array, we look up the word in our hash map.

3 Testing

Below is a small testing program that adds 5 words to our priority queue and tests each method:

```
--- Testing Add and Size Operations ---
Is queue empty? true
Initial size: 0

Adding words: 'fox', 'dog', 'cat', 'bear', 'elk'
Current size: 5
Is queue empty? false

--- Testing Duplicate Prevention ---
Attempting to add 'dog' again...
Size after attempting to add duplicate: 5 (Should be 5)

--- Testing Search Operation ---
Searching for 'cat': Found at index 2
Searching for 'zebra': Not found

--- Testing ExtractMin Operation ---
Extracting words in dictionary order:
Extracted: bear, New size: 4
Extracted: cat, New size: 3
Extracted: dog, New size: 2
Extracted: elk, New size: 1
Extracted: fox, New size: 0

Final size: 0
Is queue empty? true
```

As you can see, I test the size function of my priority queue first. With an initial size of 0. Then I add 5 words and test size again. After, I test out the duplicate add functionality. Next, I test search on a word in the queue and a word not in the queue. Finally, I begin extracting the minimum element in the array so the words come out in dictionary order.