5511 hw3

Tony Siu

September 2024

1 Data Structure Design

We will implement a min-heap to store English words in alphabetical order. The heap will be represented as a list (heap), and we'll maintain a variable (size) to keep track of the number of words.

1.1 Variables Involved

- **heap**: A list to store the words following the min-heap property.
- **size**: An integer representing the number of words in the heap.

2 Algorithms for Required Operations

To analyze the worst-case time complexity of the min-heap priority queue algorithms in terms of T(n), where n is the number of elements in the heap, we need to consider the exact number of operations performed during each method execution.

2.1 Report the number of words

Algorithm 1 GET_SIZE

```
1: function GetSize
2:     return size
3: end function
```

This method returns the number of words in the heap where size counter is incremented in the extract_min and add methods. The time complexity T(n) is 1.

2.2 Remove the First Word in Dictionary Order

Algorithm 2 EXTRACT_MIN

```
function ExtractMin
    if size = 0 then
        return null
    end if
    minWord ← heap[0]
    heap[0] ← heap[size - 1]
    size ← size - 1
    HeapifyDown(0)
    return minWord
end function
```

Algorithm 3 HEAPIFY_DOWN

```
function HeapifyDown(index)
    while true do
        smallest ← index
        left ← 2 × index + 1
        right ← 2 × index + 2
        if left < size and heap[left] < heap[smallest] then
            smallest ← left
        end if
        if right < size and heap[right] < heap[smallest] then
            smallest ← right
        end if
        if smallest ≠ index then
            Swap(heap[index], heap[smallest])
            index ← smallest
        else
            break
        end if
    end while
end function
```

The above method swaps the root of the heap with the last element and then removes the last element in the heap. The size variable is decreased by 1 and "heapify_down" is performed to maintain the heap property. The time complexity $T(n) = log(n)$ is from the call to "heapify_down" traversing the necessary children to maintain the heap property.

2.3 Search for specific word

Algorithm 4 SEARCH

```
function SEARCH(word, index)
    if index $\geq$ size then
        return -1
    end if
    if heap[index] = word then
        return index
    end if
    if heap[index] > word then
        return -1
    end if
    leftIndex $\leftarrow 2 \times$ index + 1
    rightIndex $\leftarrow 2 \times$ index + 2
    result $\leftarrow$ SEARCH(word, leftIndex)
    if result $\neq$ -1 then
        return result
    end if
    return SEARCH(word, rightIndex)
end function
```

The above method recursively traverses the heap until the word is found or the current root level is greater than the target word. The assumption is that since all roots must be smaller than its children, there doesn't need additional search into the children nodes once the target word is greater than the root. The time complexity $T(n) = n$ in the worst case as traveral may require traversing all nodes. Best case is $T(n) = 1$ where the word is in the root. There is the conditional case where early stopping to search is performed which is bounded by the worst and best case time complexity.

2.4 Add a new word if not already in the queue

Algorithm 5 ADD

```
function ADD(word)
    if SEARCH(word, 0) $\neq$ -1 then
        return
    end if
    size $\leftarrow$ size + 1
    heap[size - 1] $\leftarrow$ word
    HEAPIFYUP(size - 1)
end function
```

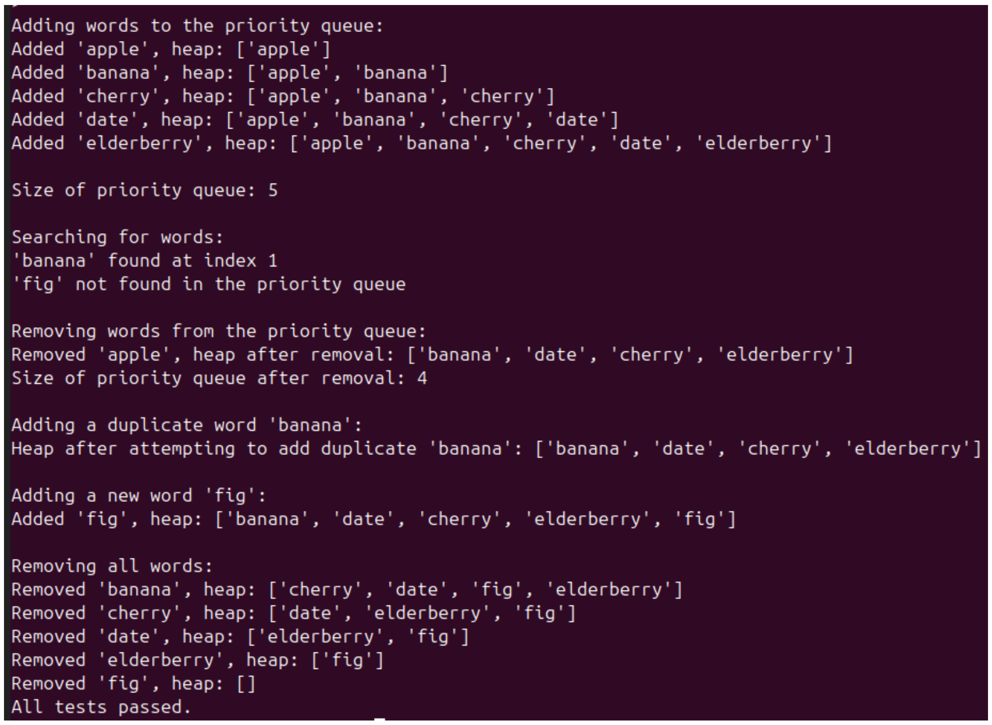
Algorithm 6 HEAPIFY_UP

```
1: function HEAPIFYUP(index)
2:     while index > 0 do
3:         parent ← (index - 1) div 2
4:         if heap[parent] > heap[index] then
5:             Swap(heap[parent], heap[index])
6:             index ← parent
7:         else
8:             break
9:         end if
10:    end while
11: end function
```

The above method first searches for the word. If the word is not found, the word is appended to the heap. The size variables increments by 1 and "heapify_up" is performed from the last index to main the heap property. The time complexity to add a new word comprises of first searching the word if its in the heap, insertion which is $T(n) = log(n)$ due to "heapify_up" to maintain heap property. That can be expressed as $T(n) = n + log(n) \approx n$.

3 Test program outputs

The code implementation can be found here. The below screenshot is a sample output from the test program from test_pq.py that imports the priority queue from min_heap_pq.py.



```
Adding words to the priority queue:
Added 'apple', heap: ['apple']
Added 'banana', heap: ['apple', 'banana']
Added 'cherry', heap: ['apple', 'banana', 'cherry']
Added 'date', heap: ['apple', 'banana', 'cherry', 'date']
Added 'elderberry', heap: ['apple', 'banana', 'cherry', 'date', 'elderberry']

Size of priority queue: 5

Searching for words:
'banana' found at index 1
'fig' not found in the priority queue

Removing words from the priority queue:
Removed 'apple', heap after removal: ['banana', 'date', 'cherry', 'elderberry']
Size of priority queue after removal: 4

Adding a duplicate word 'banana':
Heap after attempting to add duplicate 'banana': ['banana', 'date', 'cherry', 'elderberry']

Adding a new word 'fig':
Added 'fig', heap: ['banana', 'date', 'cherry', 'elderberry', 'fig']

Removing all words:
Removed 'banana', heap: ['cherry', 'date', 'fig', 'elderberry']
Removed 'cherry', heap: ['date', 'elderberry', 'fig']
Removed 'date', heap: ['elderberry', 'fig']
Removed 'elderberry', heap: ['fig']
Removed 'fig', heap: []
All tests passed.
```