5511 hw3

Tony Siu

September 2024

1 Data Structure Design

We will design a priority queue that stores English words in alphabetical order using a min-heap augmented with a hash table (dictionary). This combination allows efficient retrieval, insertion, and deletion operations while maintaining the heap property and enabling fast searches.

1.1 Variables Involved

- **heap**: A list (array) that represents the heap structure where each element is a word.
- **position map**: A dictionary (hash table) that maps each word to its index in the heap list. This allows O(1) average-time searches and updates.
- **heap size**: The number of elements in the heap can be obtained using len(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. This includes comparisons, swaps, and hash table operations (lookups, insertions, deletions). I made the following assumptions:

- The heap is implemented as a list (array) with zero-based indexing.
- The hash table "position map" is implemented as a python dictionary
- I assume that the**Average-case** with Hash table operations "get", "set", "del" take O(1) time.
- I assume the **worse-case** hash table operations due to hash collisions can be degraded to O(n) time.

2.1 Report the number of words

function SIZE
    return length(self.heap)
end function

The above function returns the length of the heap in the Priority Queue. This method returns the number of words in the priority queue by returning the heap list. The time complexity T(n) is 1 as retrieving the length of a list in python is a constant time operation.

2.2 Remove the First Word in Dictionary Order

function REMOVEMIN
    if not self.heap **then**
        return None
    end if
    $minWord \leftarrow self.heap[0]$
    $lastWord \leftarrow self.heap$.pop()
    delete $self.positionMap[minWord]$
    if self.heap **then**
        $self.heap[0] \leftarrow lastWord$
        $self.positionMap[lastWord] \leftarrow 0$
        $self$.siftDown(0)
    end if
    return $minWord$
end function

The above algorithm first checks if the heap is not empty. Then the root of the heap is removed and replaced with the last element in the heap. The position map is updated by removign the deleted word and updating the position of the moved word. Finally, sift down is called to maintain the min heap. This is a O(logn) operation.

The worst case operations include hash table operations, comparisons, swapping and hash updates. This can be expressed as

$$T(n) = 2n(initialhashoperations) + \lfloor log_2 n \rfloor \times (3 + 2n)$$

. Simplifying this becomes;

$$T(n) = 2n + \lfloor log_2 n \rfloor (3 + 2n)$$

2.3 Search for a Specific Word

function SEARCH(word)
    return self.positionMap.get($word, -1$)
end function

By constructing a min heap and using a hash map to map the position of the min heap to the word, searching for a specific word is only that of querying

the hash map to get the word. The time complexity for the search function is entirely founded upon the hash complexity of the hash table. The worst case is $T(n) = n$ due to low level hash collisions that degrades to O(n) time.

2.4 Add a new word if not already in the queue

```
function ADD(word)
    if word ∈ self.positionMap then
        return                                  ▷ Word already in the queue
    end if
    self.heap.append(word)
    index ← length(self.heap) − 1
    self.positionMap[word] ← index      self.siftUp(index)
```

For this function, it first check if the word is already in the position map. If it is not, the word will be append to the end of the heap array. The position map is updated with the new word and its index in the min heap. Sift up is finally performed to maintain the heap property.

The worst case time complexity includes hash table operations, comparison, swapping and hash table updates. This is expressed as

$$T(n) = n(lookup) + 1(append) + n(hashupdate) + \lfloor log_2 n \rfloor \times (2 + 2n)$$

Simplifying this becomes;

$$T(n) = 2n + 1 + 2\lfloor log_2 n \rfloor + 2n\lfloor log_2 n \rfloor$$

2.5 Helper Methods

The number of iterations in sift up is proportional to the height of the heap which is $\lfloor log_2 n \rfloor$. In each iteration, a constant number of comparisons and swaps are performed. Hash table updates can be O(1) on average but O(n) in the worst case.

2.5.1 Sift up

```
function SIFTUP(index)
    while index > 0 do
        parent ← (index − 1) ÷ 2
        if self.heap[index] ≥ self.heap[parent] then
            break
        end if
        self.swap(index, parent)
        index ← parent
    end while
end function
```

The above sift up operations first moves the element at "index" up the heap until the heap property is restored. The position map is updated in the swap method call.

2.5.2 sift down

```
function SiftDown(index)
    size ← length(self.heap)
    while true do
        smallest ← index
        left ← 2 × index + 1
        right ← 2 × index + 2
        if left < size and self.heap[left] < self.heap[smallest] then
            smallest ← left
        end if
        if right < size and self.heap[right] < self.heap[smallest] then
            smallest ← right
        end if
        if smallest = index then
            break
        end if
        self.swap(index, smallest)
        index ← smallest
    end while
end function
```

The sift down operation first moves the element at "index" down the heap until the heap property is restored. Same with sift up, the position map is updated during word swaps.

2.5.3 Swap

```
function Swap(i, j)
    self.positionMap[self.heap[i]], self.positionMap[self.heap[j]] ← j, i
    self.heap[i], self.heap[j] ← self.heap[j], self.heap[i]
end function
```

The core operation in this swap function is to update the hash map for the word and position index for search purposes and also swap the positions in the heap between words.

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 priority_queue.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: []
```