Algorithms and Data Structures

Assignment 4: Heap Data Structures: Implementation, Analysis and Applications

Nguyen The Duy Khanh

005019181

Table of Contents

1. Heap Sort Implementation and Analysis

a. Practical Implementation

- The Heap Sort algorithm [1] has two steps:
    - **Step 1:** Building a max heap using heapify

```python
# heapify will build the max-heap based on the current node
# compared to its leaf node and heapify at each level
def heapify(array: list[int], numberOfElements: int, currentIndex: int) -> None:
  largestIndex = currentIndex
  leftNodeIndex = 2 * currentIndex + 1
  rightNodeIndex = 2 * currentIndex + 2

  if leftNodeIndex < numberOfElements and array[largestIndex] < array[leftNodeIndex]:
    largestIndex = leftNodeIndex

  if rightNodeIndex < numberOfElements and array[largestIndex] < array[rightNodeIndex]:
    largestIndex = rightNodeIndex

  if largestIndex != currentIndex:
    array[currentIndex], array[largestIndex] = array[largestIndex], array[currentIndex]
    heapify(array, numberOfElements, largestIndex)
```

```python
def heapSort(array: list[int]) -> None:
  numberOfElements = len(array)
  # Step 1: Build a max heap
  for currentIndex in range(numberOfElements//2 - 1, -1, -1):
      heapify(array, numberOfElements, currentIndex)
```

Figure 1: Build Max Heap

- **Step 2:** Remove the root element (which has the largest value) as a max heap by swapping it with the last element. Afterwards, perform a shift down to maintain the heap structure by replacing the root node with the second largest node. Then performing the same operation with the remaining nodes.

```python
# heapSort will sort based on the following steps:
# Step 1: Build a max heap using heapify
# Step 2: The root node is the largest element and it is the first element.
# Therefore, we will remove the largest element by swapping it with the last element
# and heapify again in order to replace the root node with the second largest element
# https://www.programiz.com/dsa/heap-sort      You, yesterday • Add comments and readme for runnir
# Time complexity: O(n log n)
# Space complexity: O (log n)
def heapSort(array: list[int]) -> None:
    numberOfElements = len(array)
    # Step 1: Build a max heap
    for currentIndex in range(numberOfElements//2 - 1, -1, -1):
        heapify(array, numberOfElements, currentIndex)
    # Remove the largest element iteratively and heapify to replace the root node
    # with the second largest element
    for maxElement in range(numberOfElements-1, 0, -1):
        array[maxElement], array[0] = array[0], array[maxElement]
        heapify(array, maxElement, 0)
```

Figure 2: Heap Sort

- As a result of Heap Sort, we will have the following result:

```
MSCS532_Assignment4 on  main via  v3.12.5 on  kel.nguyen@vietnamtechsociety.org(us-east1)
❯ python3 main.py
Before sorting
44 185 61 28 160 68 69 98 144 39

After sorting
28 39 44 61 68 69 98 144 160 185
```

Figure 3: Heap Sort's Result

b. Theoretical analysis

- There are two stages of heap sort:
    - **Step 1:** Building a max heap by using a shift down or shift up [2].
        - Shift down swaps a node that is too small with its largest child by moving it down until it is at least as both nodes below it and vice versa for shift up.
        - The number of shift operations required for both shift operations are proportional to the distance the node may have to move. For shift Down, it is the distance to the bottom of the tree, so shift Down is

expensive for nodes at the top of the tree and vice versa for shift up. Although both operations are O(log n) in the worst case, in a heap, only one node is at the top where as half the nodes lie in the bottom layer. Therefore, shift down is more preferable than shift up.

- Therefore, when building a max heap out of an array, the time complexity is O(n) with shift down.

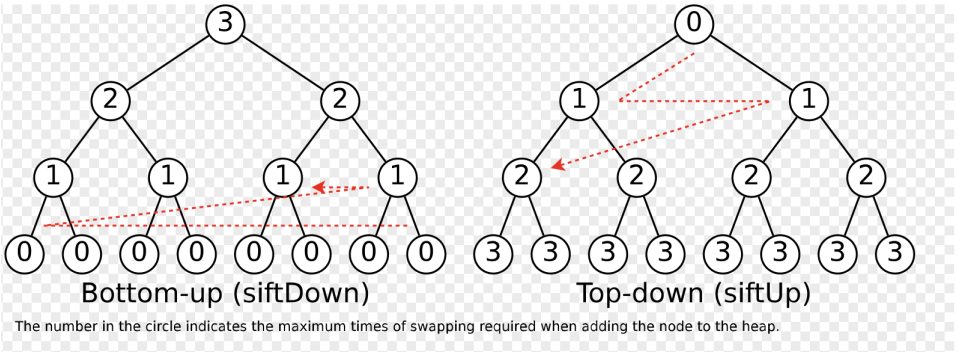

Figure 4: Shift up and shift down's cost

- **Step 2:** Remove the root element (which has the largest value) as a max heap by swapping it with the last element. Afterwards, perform a shift down to maintain the heap structure by replacing the root node with the second largest node. Then performing the same operation with the remaining nodes. Since we perform the shift down process to every element in the array after swapping the largest element to the last element of the array, the time complexity would be O (n log n) for average, best, and worst complexity.

- Since we don't create any extra auxiliary space, the only that cause additional overhead would be the recursive stack calls which cause O(log n) since the height is half because of binary heap's characteristics. However, HeapSort has an additional overheads when compared to Quick Sort since the inner loop of shift up operation is

longer than QuickSort (because each node gets compared with it's two leaf and two compares each done a N Log N times [3])

- The general recurrence relation for Heap Sort is:

$$T(n) = T(\frac{n}{2}) + O(log\ n)$$

where $\frac{n}{2}$ comes from the fact that we shift down one half of the tree and for every element while O(log n) is the shift down's time complexity.

- With *Substitution Method[4]* and we must prove that T(n) ≤ c n log n, the equation would be :

$$T(n) \leq c\frac{n}{2} log\ \frac{n}{2} + clogn$$

$$\leq cn\ log\ n$$

-

c. Comparison

- When experimenting with Heap Sort, Merge Sort and Quick Sort with different datasets, the difference between them can be clearly found:
  - With **random data** from n = 100 to 100,000 elements

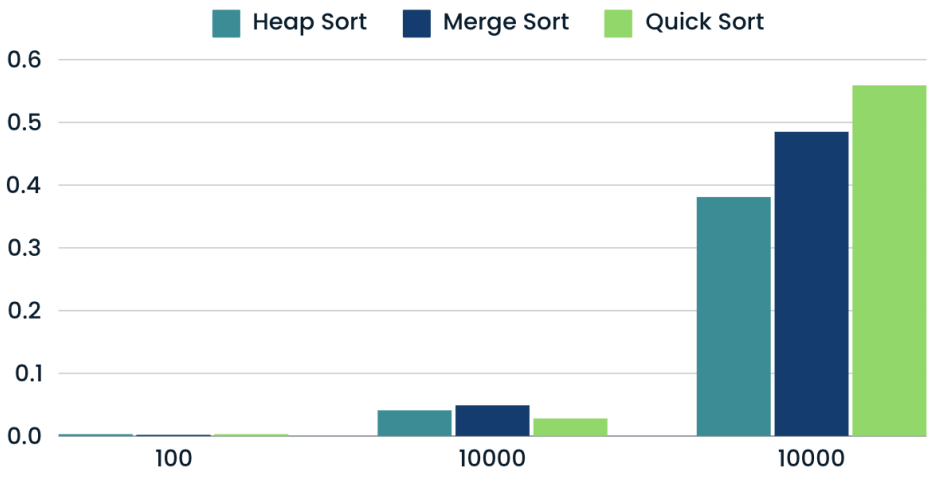

Figure 5: Random Dataset's Profiler with random data(CPU(s)))

- With **sorted order data** from n = 100 to 100,000 elements

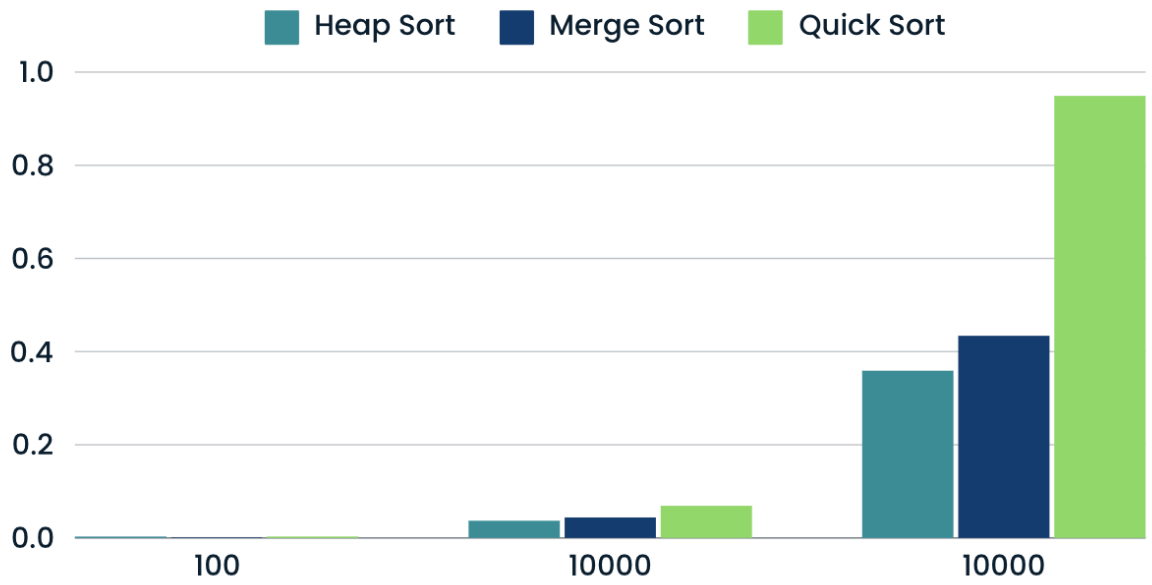

Figure 6: Random Dataset's Profiler with sorted order data (CPU(s)))

- With **reverse sorted order data** from n = 100 to 100,000 elements

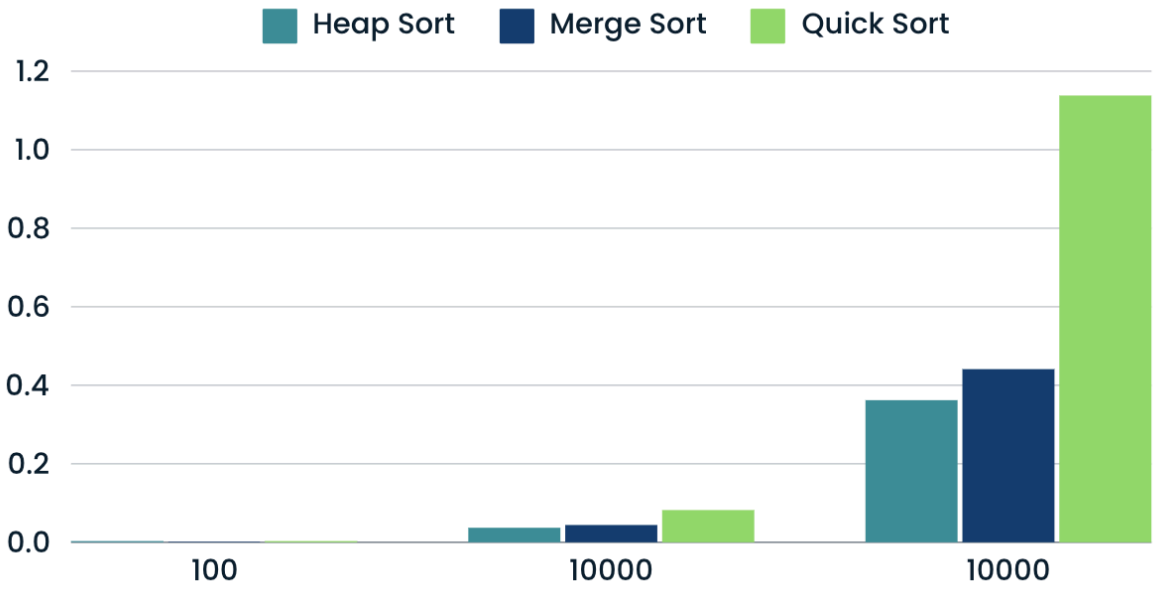

Figure 7: Random Dataset's Profiler with reverse sorted order data (CPU(s)))

- As a result of the experiment, Heap Sort has slightly the same operation time with Merge Sort. For all case, Heap Sort has 0.38s for performing a sort with all cases while QuickSort has an increased time when sorting with reverse sorted order data (from 0.05s to 1.12s for 100,000 elements) since the time complexity for this case would be O(n^2). Additionally, we expect Quick Sort will perform better in case of randomness data than HeapSort since the inner loop of shift up operation is longer than QuickSort (because each node gets compared with it's two leaf and two compares each done a N Log N times.

2. Priority Queue Implementations and Applications

- Priority Queue[5] can use different kinds of structures to implement its internal operations. Each structure will have its disadvantages and advantages, and the same can be said between array and linked list. When considering between LinkedList and Array [6], we need to know the following facts:
    - It's easier to store the data of different sizes in the linked list and scale the size organically, while an array can only store data at the same size, and its size needs to be known beforehand; otherwise, each insertion to the array can be costly (resizing the array leads to time complexity of O(n) [7]). However, LinkedList comes with a cost of extra memory overhead for storing data.
    - Array allows random access, while linked list allow only sequential access to elements. Therefore, it is unsuitable for application where it is useful to look up an index quickly (e.g quick sort, priority queue) and it will decrease the implementation's complexity.
- Since we know the size of the array for task scheduling beforehand, the time complexity of the array and linked list will be similar. Additionally, the memory

overhead that is incurred for a linked list won't apply to an array with an extra easy implementation. As we need random access when performing a shift up and shift down, an array is better with random access. Therefore, the array is more suitable for task scheduling with a priority queue.

	Array	Sorted Array	Linked List
Insert	O(1) (worst case is O(n) for resizing)	O(log n) (worst case is O(n) for resizing)	O(1)
Search	O(n)	O(log n)	O(n)
Remove	O(n)	O(n)	O(n)
Easy of implementation	Lowest	Medium	Highest

Figure 8: Analysis for different data structures

- With an array as a data structure for storing Task as a heap within the priority queue, we will have the following implementations [8] (an example for the implementation would be: Task 0 - priority 199, Task 1 - priority 93, Task 2 - priority 120, Task 3 - priority 20, Task 4 - priority 56, Task 5 - priority 49, Task 6 - priority 55, Task 7 - priority 193, Task 8 - priority 120, Task 9 - priority 41)
    - **Insert:** Since to maintain the structure of the heap, we would need to add the new task and perform a shift up to determine the task's index by comparing the task's priority with its parent's priority. If the task's priority is higher, the task will swap it with its location with the parent task and perform the same shift up operation again till it's no larger than the parent's task priority as a Max-Heap. Since we only traverse a single leaf of a binary heap each time, the time complexity for this would be O(log n) but can be O(n) in case of resizing in array[7]).

- With the following Task 0 - priority 199, Task 1 - priority 93, Task 2 - priority 120, when we insert Task 3 - priority 20 it will compare with Task 1's priority, it won't swap since Task 3 has a smaller priority. However, if Task 3 has priority of 150, it will swap the location with Task 1 and stay at that location since it's priority is smaller than Task 0.

```
# Time complexity: O(log n) (worst case is O(n) when resizie the array for appending the elements
# https://stackoverflow.com/a/77296220)
def insert(self, task: Task) -> None:
  self.heap.append(task)
  self.heapifyUp(len(self.heap) - 1)

# heapifyUp will compare the current elements with the parent node and swap it if the
# parent node is larger. Moreover, continue to do the same operations with the parent node.
# Time complexity: O(log n)
def heapifyUp(self, currentIndex: int) -> None:
  while currentIndex > 0:
    parentIndex = (currentIndex - 1) // 2

    if self.heap[currentIndex].priority > self.heap[parentIndex].priority:
      self.heap[currentIndex], self.heap[parentIndex] = self.heap[parentIndex], self.heap[currentIndex]
      currentIndex = parentIndex
    else:
      break
```

Figure 9: Insertion with Priority Queue

- **Extract Max:** To extract the max from the priority queue with max heap, we will need to remove the root node (which is the largest priority task) and perform a shift down to maintain the heap structure by replacing the root node with the 2nd largest priority task. Since we only traverse a single leaf of a binary heap each time, the time complexity for this would be O(log n).
    - With the following Task 0 - priority 199, Task 1 - priority 93, Task 2 - priority 120, when we extract the highest priority task, we will shift down by comparing Task 0 with Task 1's and Task 2's priority. Since we need to maintain the heap's characteristics, we would need to take

the largest priority out of Task 1 and Task 2 which is Task 2. Afterwards, Task 2 will be the root node.

```
def remove(self):
    return None

  if len(self.heap) == 1:
    return self.heap.pop()

  largestPriority = self.heap[0]
  self.heap[0] = self.heap.pop()
  self.heapifyDown(0)
  return largestPriority

# heapifyDown will compare the current node with the leaf node
# to replace the current node if the leaf node is larger.
# Then, heapifyDown the leaf node operation if the current node is smaller
# Time complexity: O(log n)
def heapifyDown(self, currentIndex: int) -> None:
  while currentIndex * 2 + 1 < len(self.heap):
    largestIndex = currentIndex
    leftNodeIndex = currentIndex * 2 + 1
    rightNodeIndex = currentIndex * 2 + 2

    if leftNodeIndex < len(self.heap) and self.heap[leftNodeIndex].priority > self.heap[largestIndex].priority:
        largestIndex = leftNodeIndex

    if rightNodeIndex < len(self.heap) and self.heap[rightNodeIndex].priority > self.heap[largestIndex].priority:
        largestIndex = rightNodeIndex

    if self.heap[currentIndex].priority < self.heap[largestIndex].priority:
        self.heap[currentIndex], self.heap[largestIndex] = self.heap[largestIndex], self.heap[currentIndex]
        currentIndex = largestIndex
    else:
        break
```

Figure 10: Extract Max with Priority Queue

- **Change priority:** When changing the priority, we would need to determine if the new priority is larger than the current priority and perform a shift up if the new priority is larger than the current priority and vice versa. Otherwise, it will violate the characteristics of max heap. Since we need to determine the task's index, we would need a time complexity of O(n) and only O (log n) to perform a shift up or down.
    - With the following Task 0 - priority 199, Task 1 - priority 93, Task 2 - priority 120, when we change Task 2 - priority to 200, it will perform a shift up and compare with Task 0's priority. Since Task 2 priority is

larger, it will swap the location with Task 0 and stay as a largest priority task.

```
# changePriority will change the priortiy of the current index
# if the priority is higher than the current index's priority,
# then perform a heapify up to maintain the heap structure and vice versa
# Time complexity: O(log n) if not including search, if including search it would be O(n)
def changePriority(self, taskName: str, priority: int) -> None:
    index, task = self.priorityQueue.search(taskName=taskName)
    if index == -1:
        return

    oldPriority = task.priority
    task.priority = priority
    if priority > oldPriority:
        self.priorityQueue.heapifyUp(index)
    else:
        self.priorityQueue.heapifyDown(index)

# printPriorityQueue will print all the elements in the heap
```

Figure 11: Change priority with Priority Queue

- **Is Empty:** When checking if the heap is empty, we only need to confirm if there are any remaining tasks in the heap or not. The time complexity would be O(1)
    - Since we have 10 elements in the heap, we will return not empty for the array.

```
# isEmpty will check if the
# Time complexity: O(1)
def isEmpty(self) -> bool:
    return len(self.priorityQueue.heap) == 0
```

Figure 12: Checking Priority Queue is empty

- As a result of each operation in heap,

```
MSCS532_Assignment3 on  main [$!?] via  v3.12.5 on  kel.nguyen@vietnamtechsociety.org(us-east1)
❯ python3 main.py
Number of elements at bucket index 0  is  1
Number of elements at bucket index 1  is  2
Number of elements at bucket index 4  is  1
Number of elements at bucket index 6  is  2
Number of elements at bucket index 9  is  1
Number of elements at bucket index 10  is  1
Number of elements at bucket index 15  is  1
Number of elements at bucket index 25  is  2
Removing Tingyun from HashMap
Value of 35: -1
Value of 35: FeiXiao
Value of 6: -1
Value of 89: Firefly
```

Figure 13: Operation's Result with Priority Queue

3. References

- [1] *Richmond Alake* (2023). Heap Sort Explained. Builtin. https://builtin.com/data-science/heap-sort
- [2] *Jeremy West* (2013). How can building a heap be on time O(n) on time complexity. Stack over flows. https://stackoverflow.com/questions/9755721/how-can-building-a-heap-be-on-time-complexity
- [3] *Omar Elgabry* (2017). The angry birds - mighty eagle. Medium. https://medium.com/omarelgabrys-blog/the-angry-birds-mighty-eagle-2db211cc4019
- [4] *GeeksForGeeks* (2024) How to analyze the complexity of the recurrence equation https://www.geeksforgeeks.org/how-to-analyse-complexity-of-recurrence-relation/
- [5] *GeeksForGeeks* (2024). Priority Queue Set 1 Introduction https://www.geeksforgeeks.org/priority-queue-set-1-introduction/
- [6] *Jonas Klemming (2008)* Array Versus Linked List. StackOverFlow. https://stackoverflow.com/a/166966
- [7] *Karoly Horvath* (2014). Time complexity of append operation in simple array. Stackoverflow.

https://stackoverflow.com/questions/22173998/time-complexity-of-append-operation-in-simple-array

- [8] *Kunal Pratap Singh (2020). Complexity analysis of various operations of binary min heap. GeeksForGeeks.* https://www.geeksforgeeks.org/complexity-analysis-of-various-operations-of-binary-min-heap/