9/15/2024

Assignment 4

Heap Data Structures and Priority Queue: Implementation and Applications

Sujan Dumaru

Student ID: 005030060

MSCS 532: Algorithms and Data Structures

Dr. Vanessa Cooper

INTRODUCTION

Heap data structures are fundamental in computer science, commonly used for efficient sorting and priority management. These tree-based structures consist of parent and child nodes and satisfy one of two heap properties: in a **Max-Heap**, the largest element is always at the root, while in a **Min-Heap**, the smallest element is at the root. This report examines the implementation and analysis of Heapsort, a sorting algorithm that utilizes a binary heap to achieve *O (n logn)* time complexity. Heapsort's performance is compared to other sorting algorithms, such as Quicksort and Merge Sort, across various input types. Additionally, the report covers the implementation of a priority queue using a binary heap, especially min-heap, with applications in task scheduling, analyzing the efficiency of key operations like insertion, priority adjustment, and extraction.

HEAPSORT IMPLEMENTATION

Heapsort is a comparison-based sorting technique based on a Binary Heap data structure. Heapsort can be implemented by converting an unsorted array into a max heap using the **heapify** method, which restructures the heap to maintain its property. In this case, Heapsort will utilize a max-heap, while a min-heap will later be used for the priority queue implementation. The ability to perform heapify is a key characteristic of heap data structures, as it ensures that after insertion or deletion, the largest element always remains at the root in a max heap. First, the maximum element is obtained through heapify, and it is placed at the end. After that, the same process is repeated for the remaining elements, which is a recursive approach, making the implementation straightforward. The Heapsort implementation is shown below.

```python
def max_heapify(array, heap_size, index):
    """ Function to heapify a subtree rooted with node index """
    left = 2 * index + 1                        # Left child index with parent index
    right = 2 * index + 2                       # Right child index with parent index

    # Check to see if the left child of root exists and if it does,
    # check to see if the value is greater than the root.
    # Set the largest to the largest value index between left child and root
    if (left < heap_size) and (array[left] > array[index]):
        largest = left
    else:
        largest = index

    # Check to see if the right child of root exists and if it does,
    # check to see if the value is greater than the root.
    # Set the largest to the right child index in order to swap
    if (right < heap_size) and (array[right] > array[largest]):
        largest = right

    # If the largest is either left or right child, swap it with the root
    if largest is not index:
        array[index], array[largest] = array[largest], array[index]

        # Recursively heapify the affected subtree
        max_heapify(array, heap_size, largest)
```

Figure 1: Implementation of heapify for Max Heap

```python
def build_max_heap(array, array_size):
    """ Function to build a max heap from the array """
    # Since last parent will be at (array_size//2) it's the start point.
    for index in range(array_size // 2, -1, -1):
        max_heapify(array, array_size, index)

# @profile
def max_heap_sort(array):
    """ Function to sort the array using max heap """
    array_size = len(array)
    build_max_heap(array, array_size)

    # Extract the largest number one by one from the heap
    for index in range(array_size - 1, 0, -1):
        # The largest value element will go at the end of array and
        # heapify will be done with rest of the elements excluding last one
        array[index], array[0] = array[0], array[index]
        max_heapify(array, index, 0)

    return array

```

Figure 2: Heapsort Implementation on Max Heap

Analysis of Implementation

The time-complexity for the heap sort is always *O (n logn)* in all cases because of how it executes. The implementation of the heapsort can be broken down into two main stages: **build_max_heap(arr)** and extracting each element through **max_heapify(arr)**.

During the **build_max_heap(arr)** phase, all non-leaf nodes (approximately *n // 2)* are heapified, starting from the bottom-most level, where **n** is the total size of the array. Heapifying a single node takes *O (n logn)* time because it involves adjusting the node in a binary tree structure. The overall time to build the heap is *O (n)*, as heapifying from the bottom to the top is more efficient than heapifying from the top down.

The process of heapifying can be described using the recurrence relation:

$$T(n) = 2T(n/2) + O(n)$$

By applying the master theorem, this resolves to $T(n) = O(\lg n)$ (Cormen et al., 2009). Therefore, building the heap takes *O(n)* time. After building the heap, each element is extracted and placed at the end of the array. For each extraction, **max_heapify(arr)** is called to maintain the heap property, reducing the heap size each time. Since there are *n* elements and each extraction requires *O (log n)* time, the total time for the extraction phase is:

$$n \text{ extractions} \times O(\log n) \text{ time per extraction} = O(n \log n)$$

Thus, the overall time complexity of Heapsort is dominated by the extraction step, giving it a time complexity of *O (n logn)* time in all cases.

Since Heapsort is an in-place sorting algorithm, it does not require additional space for sorting, resulting in a space complexity of *O (1)*.

Comparison with other sorting algorithms

Heapsort offers both advantages and disadvantages when compared to algorithms like Quicksort and Merge Sort. Below is a theoretical comparison of these algorithms in terms of time complexity, space complexity, and their performance on different input types:

Algorithms	Best Case	Average Case	Worst Case
Heapsort	Ω (n logn)	Θ (n logn)	O (n logn)
Quicksort	Ω (n logn)	Θ (n logn)	O (n^2)
Merge sort	Ω (n logn)	Θ (n logn)	O (n logn)

Table 1: Time Complexities between different sorting algorithms

```
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> python -u "z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding\MSCS532_Assignment2\quick_sort.py"
Random array:
Total execution time: 0.021849799999472452
Sorted array:
Total execution time: 0.040444499998557 15
Reversed sorted array:
Total execution time: 0.03384430000005523
Huge random array:
Total execution time: 2.738786499998241
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding>
```

Figure 3: Quick sort execution times on different dataset

```
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> python -u "z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding\MSCS532_Assignment2\merge_sort.py"
Random array:
Total execution time: 0.06952799999999115
Sorted array:
Total execution time: 0.06753639999806182
Reversed sorted array:
Total execution time: 0.0678769000005559
Huge random array:
Total execution time: 6.521810199999891
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding>
```

Figure 4: Merge sort execution times on different dataset

```
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> py
"
RANDOM ARRAY total execution time: 0.039991200000258686
SORTED ARRAY total execution time: 0.044530200000281184
REVERSED SORTED ARRAY total execution time: 0.03389690000039991
HUGE RANDOM ARRAY total execution time: 5.726385599999958
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding>
```

Figure 5: Heapsort execution times on different dataset

From the above figures, we can conclude that Heapsort is faster than merge sort but slower than Quick sort. However, the space complexity makes Heapsort more suitable in memory-constrained environment. The space complexity between different algorithms is shown below:

Algorithm	Space Complexity
Heapsort	*O (1)*
Quicksort	*O (n logn)*
Merge sort	*O (n)*

Table 2: Space Complexities between different sorting algorithms

PRIORITY QUEUE IMPLEMENTATION

A priority queue is an abstract data structure where each element is associated with a priority, and elements are served based on their priority rather than the order of insertion. Priority queues are commonly used in task scheduling, ensuring higher-priority tasks are executed before lower-priority ones. For example, they are used in operating systems to manage processes and in airlines to prioritize Business or First-class passengers. In this report, the priority queue is implemented using an array, as arrays provide indexed access, making the construction and management of the priority queue more straightforward. A min-heap priority queue is used, meaning tasks with the lowest priority are served first. The implementation for priority queue using Python is given below along with its output:

```
""" Understanding Priority Queue """

You, 16 hours ago | 1 author (You)
class Task:
    """ Task class to represent individual tasks, and storing relevant information """
    def __init__(self, task_id, priority, arrival_time, deadline):
        self.task_id = task_id
        self.priority = priority
        self.arrival_time = arrival_time
        self.deadline = deadline

```

Figure 6: Task class to store relevant information

```python
class MinHeapPriorityQueue:
    """ Minimum heap priority queue which will store lowest priority element first """
    def __init__(self):
        self.heap = []

    def shift_up(self, index):
        """ Function to restore min heap property """
        parent = (index - 1) // 2

        # Make sure that the lowest priority is at the root
        while index > 0 and self.heap[parent].priority > self.heap[index].priority:
            self.heap[parent], self.heap[index] = self.heap[index], self.heap[parent]
            index = parent
            parent = (index - 1) // 2

    def insert(self, task):
        """ Insert task into the priority queue as a heap """
        self.heap.append(task)
        self.shift_up(len(self.heap) - 1)

    def heap_minimum_element(self):
        """ Return the minimum element from the heap """
        if self.is_empty():
            raise IndexError("Minimum Heap Priority queue is empty.")
        return self.heap[0]

    def shift_down(self, index):
        """ Function to shift minimum a subtree rooted with node index """
        heap_size = len(self.heap)
        left = 2 * index + 1                        # Left child index with parent index
        right = 2 * index + 2                       # Right child index with parent index

        # Check to see if the left child of root exists and if it does,
        # check to see if the priority is smaller than the root.
        # Set the smallest to the smaller value index between left child and root
        if (left < heap_size) and (self.heap[left].priority < self.heap[index].priority):
            smallest = left
        else:
            smallest = index

        # Check to see if the right child of root exists and if it does,
        # check to see if the value is smaller than the root.
        # Set the smallest to the right child index in order to swap
        if (right < heap_size) and (self.heap[right].priority < self.heap[smallest].priority):
            smallest = right

        # If the smallest is either left or right child, swap it with the root
        if smallest is not index:
            self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]

            # Recursively heapify the affected subtree
            self.shift_down(smallest)
```

Figure 7: Min heap priority queue implementation

```python
    def extract_min(self):
        """ Function to get the minimum element from the heap """
        min_value = self.heap_minimum_element()         # Get the minimum priority element from the heap
        self.heap[0] = self.heap[-1]
        self.heap.pop()                                 # Remove the last task
        self.shift_down(0)                              # Get the lowest priority element to top

        return min_value

    def decrease_key(self, task_id, new_priority):      You, 17 hours ago • Added implementation for heap_s
        """ Function to decrease the priority of a element in the heap """
        for index, task in enumerate(self.heap):
            if task.task_id == task_id:
                if new_priority < task.priority:
                    task.priority = new_priority
                    self.shift_up(index)                # Maintain heap property
                else:
                    raise ValueError("New priority must be smaller than current priority.")
                break

    def is_empty(self):
        """ Return if the heap is empty or not """
        return len(self.heap) == 0

task_priority_queue = MinHeapPriorityQueue()

# Insert tasks in the queue
task_priority_queue.insert(Task(1, 3, '10:00', '12:00'))
task_priority_queue.insert(Task(2, 5, '10:05', '12:05'))
task_priority_queue.insert(Task(3, 2, '10:10', '12:10'))
task_priority_queue.insert(Task(4, 3, '10:23', '12:15'))
task_priority_queue.insert(Task(5, 1, '10:16', '12:20'))
task_priority_queue.insert(Task(6, 5, '10:30', '12:30'))

# Extract heap_minimum_element
# Should extract the task with the lowest priority (Task_ID: 5, Priority: 1)
print("Extracted task with task_id:", task_priority_queue.extract_min().task_id)

# Should extract the task with the lowest priority (Task_ID: 3, Priority: 2)
print("Extracted task with task_id:", task_priority_queue.extract_min().task_id)

# Decrease priority
task_priority_queue.decrease_key(2, 2)  # Decrease task 2's priority to 2
task_priority_queue.decrease_key(6, 4)  # Decrease task 6's priority to 3

# Should extract the task with the lowest priority (Task_ID: 3, Priority: 2)
print("Extracted task with task_id:", task_priority_queue.extract_min().task_id)

# Check queue state
print("Queue:")
for i in task_priority_queue.heap:
    print("Task ID: ", i.task_id, " Priority: ", i.priority)
```

Figure 8: Min heap priority queue implementation 2

- **insert(task)** – As the name implies, this function inserts tasks into the priority queue, placing them at the bottom of the heap. The newly added element is then moved up using the **shift_up()** function, depending on its priority, to maintain the min-heap property. The time complexity of this operation is *O (log n)*, as it depends on the height of the tree for shifting the newly inserted element. In the worst case, the element may need to travel from the bottom to the root all the way to top, which still takes *O (log n)* time.
- **extract_min()** – This function extracts the task with the lowest priority, which is the root of the heap. After the root is removed, the last element in the heap is moved to the root and then shifted down using the **shift_down()** method to restore the heap property, ensuring the element with the lowest priority remains at the root. The time complexity of this operation is *O (log n)*, similar to the **shift_up()** method as it depends on the height of the tree to move the element down.
- **decrease_key(task_id, new_priority)** – This function decreases the priority of a task that matches the given task ID and repositions it in the queue based on the new, lower priority using the **shift_up()** method. Since the new priority is lower, the task will move up in the tree. The time complexity of this operation is *O (log n)*, due to the **shift_up()** process.
- **is_empty()** – This is a simple function to check if the priority queue is empty or not. It takes *O (1)* time to complete.

These are the main functions used to implement the priority queue in this report. A min-heap priority queue can also be implemented using Python's built-in *heapq* library. The output for the provided code block is shown in the figure below. From the figure, we can see that the task with **task_id 5**, which had the lowest priority 1, is positioned at the root before extraction.

```
for i in task_priority_queue.heap:
    print("Task ID: ", i.task_id, " Priority: ", i.priority)

# Extract heap_minimum_element
# Should extract the task with the lowest priority (Task_ID: 5, Priority: 1)
print("Extracted task with task_id:", task_priority_queue.extract_min().task_id)

# Should extract the task with the lowest priority (Task_ID: 3, Priority: 2)
print("Extracted task with task_id:", task_priority_queue.extract_min().task_id)

# Decrease priority
task_priority_queue.decrease_key(2, 2)  # Decrease task 2's priority to 2
task_priority_queue.decrease_key(6, 4)  # Decrease task 6's priority to 3

# Should extract the task with the lowest priority (Task_ID: 3, Priority: 2)
print("Extracted task with task_id:", task_priority_queue.extract_min().task_id)

# Check queue state
print("Queue after extraction:")
for i in task_priority_queue.heap:
    print("Task ID: ", i.task_id, " Priority: ", i.priority)

# Check if empty
print("Is queue empty?", task_priority_queue.is_empty())

# Should extract the task with the lowest priority (Task_ID: 4, Priority: 3)
print("Extracted task with task_id:", task_priority_queue.extract_min().task_id)

# Should extract the task with the lowest priority (Task_ID: 1, Priority: 3)
print("Extracted task with task_id:", task_priority_queue.extract_min().task_id)

# Should extract the task with the lowest priority (Task_ID: 6, Priority: 4)
print("Extracted task with task_id:", task_priority_queue.extract_min().task_id)

# Check if empty
print("Is queue empty?", task_priority_queue.is_empty())
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

```
Queue before extracting lowest element:
Task ID:  5  Priority:  1
Task ID:  3  Priority:  2
Task ID:  1  Priority:  3
Task ID:  2  Priority:  5
Task ID:  4  Priority:  3
Task ID:  6  Priority:  5
Extracted task with task_id: 5
Extracted task with task_id: 3
Extracted task with task_id: 2
Queue after extraction:
Task ID:  4  Priority:  3
Task ID:  6  Priority:  4
Task ID:  1  Priority:  3
Is queue empty? False
Extracted task with task_id: 4
Extracted task with task_id: 1
Extracted task with task_id: 6
Is queue empty? True
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding>
```

Figure 9: Min heap priority queue implementation 3 and Output result

The **extract_min()** function is called three times, extracting the tasks with the lowest priorities first. After these extractions, three tasks remain, with the lowest priority task (**task_id 4**) still at the root, maintaining the heap property. Subsequent extractions reveal that **task_id 6**, having the highest priority, is the last to be extracted.

CONCLUSION

In this report, we explored the implementation and analysis of heap data structures, focusing on their applications in Heapsort and priority queues. Heapsort was implemented using a max-heap, and its time complexity of *O(n logn)* in all cases was thoroughly analyzed and compared with other sorting algorithms like Quicksort and Merge Sort. The priority queue was implemented using a min-heap, showcasing efficient insertion, extraction, and priority adjustment operations, each maintaining *O(log n)* time complexity.

Through this project, we demonstrated the practicality of heap data structures in solving real-world problems, such as task scheduling, where tasks are processed based on their priority. The efficient management of priority queues, along with the robust performance of Heapsort, highlights the importance of heaps in both theoretical and applied computing. Overall, this report underscores the versatility and efficiency of heaps in both sorting and priority management scenarios.

REFERENCES

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (4th ed.).