Priya Thapa

MSCS-532

11/7/2025

Dr. Brandon Bass

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

Heapsort Implementation and Analysis

1. Implementation

```
> Users > priyat > heap_sort.py > heapify
arr = [4, 10, 3, 5, 11, 23, 1]
n = len(arr)

def heapify(arr, n, i):
  largest = i
  left = 2 * i + 1
  right = 2 * i + 2

  #compare left child node with the parent node
  if left < n and arr[left] > arr[largest]:
    largest = left

  #compare right child node with the parent node
  if right < n and arr[right] > arr[largest]:
    largest = right

  #check if the current largest is the the current i node
  if i != largest:
    #swap the position for i and current largest
    arr[i], arr[largest] = arr[largest], arr[i]
    heapify(arr, n, largest)


for i in range(n//2 -1 , -1, -1):
    heapify(arr, n, i)

print ("Original Array:", arr)
print("Max Heap:", arr)
```

Output:

```
PS C:\Users\priyat> python heap_sort.py
Original Array: [23, 11, 4, 5, 10, 3, 1]
Max Heap: [23, 11, 4, 5, 10, 3, 1]
```

2. Analysis of Implementation

A rigorous analysis of the time complexity of Heapsort in the worst, average, and best cases.

Heap Sort essentially does two major phases:

1. Build a max-heap from the input array of size n.
2. Repeatedly extract the maximum (the root of the heap) and adjust the heap until it is empty, resulting in a sorted array.

Phase 1: Build max-heap

- Represent the array as a complete binary tree.
- For each non-leaf node (indices roughly from $\lfloor n/2 \rfloor - 1$down to 0) call the heapify operation.
- Heapify may cause a node to move down the tree, and in the worst case it can travel along the height of the tree, which is $O(\log\ n)$.
- But because most nodes are near the leaves (small height), the total cost to build the heap ends up being $O(n)$. (There is a standard summation argument:

  $\sum_{h=0}^{\log n} (n/2^{h+1}) \cdot O(h) = O(n)$.)

Phase 2: Extract-max repeatedly

- Each extraction:
  1. Swap the root (largest) with the last element in the heap.
  2. Reduce the heap-size by 1.
  3. Heapify the root node to restore the max-heap property.
- Each heapify in this phase can cost up to $O(\log\ n)$because the heap's height is $\log n$.
- We do this extraction step for each of the $n$elements (or $n - 1$extractions) → giving a cost of about $n \cdot O(\log\ n) = O(n\log\ n)$.

- Add in the building-heap cost → overall $O(n) + O(n\log\ n) = O(n\log\ n)$.

Case	Time Complexity	Observation
Best Case	O (n log n)	Even in the worst arrangement of input, Heap Sort still must build the heap (≈ O(n)) and then do about $n$extractions each costing up to $O(\log\ n)$.
Average Case	O (n log n)	For a typical input, the same logic applies: the heapify and extraction costs are still $O(\log\ n)$each for about $n$extractions. So average = $O(n\log\ n)$.
Worst Case	O (n log n)	Importantly, unlike some other sorting algorithms, Heap Sort doesn't significantly improve in best-case we still build the heap and perform the extraction phase, which still gives O ($n\ \log\ n$).

Why is it O(n log n) in all cases?

Heap Sort has a time complexity of O(n log n) in the best, average, and worst cases due to the structure of its two main phases: building a max heap and repeatedly extracting the maximum element. In the first phase, the array is transformed into a max heap, where each parent node is greater than or equal to its children. Although heapify a single node can take up to O(log n) time, most nodes are near the leaves, so the total work for building the heap sums to O(n), independent of the input order. In the second phase, the root (maximum element) is repeatedly swapped with the last element in the heap, the heap size is reduced, and the root is heapified to restore the max-heap property. Each heapify call may traverse the height of the heap, O(log n), and this is done for all n elements, resulting in a total of O(n log n). Because Heap Sort performs the same operations regardless of the initial ordering of the array, its time complexity remains O(n log n) in the worst, average, and best cases.

Space complexity and any additional overheads.

Heap Sort is an in-place sorting algorithm, meaning it does not require any extra arrays or data structures proportional to the input size. The space complexity is therefore O(1) extra space, aside from the input array itself. The algorithm only uses a few variables for indexing and swapping elements (such as i, largest, left, right) and for recursive calls in the heapify function. If heapify is implemented recursively, there is an additional overhead due to the recursion stack, which can grow up to the height of the heap, i.e., O(log n) in the worst case. This is because each recursive call goes down one level of the heap. If heapify is implemented iteratively, this stack overhead can be eliminated entirely, maintaining O(1) auxiliary space. Aside from these considerations, there are no significant additional data structures or memory allocations, making Heap Sort a memory-efficient algorithm suitable for large datasets.

3. Comparison

Comparing the running time of Heapsort with other sorting algorithms like Quicksort and Merge Sort on different input sizes and distributions (e.g., sorted, reverse-sorted, random).

Full code provided in the file "Runtime.py"

```
# Test for different input sizes and distributions
sizes = [1000, 5000, 10000, 20000]
distributions = {
    "random": lambda n: [random.randint(0, 100000) for _ in range(n)],
    "sorted": lambda n: list(range(n)),
    "reverse": lambda n: list(range(n, 0, -1))
}

# Running experiments
for dist_name, dist_func in distributions.items():
    print(f"\n--- Distribution: {dist_name} ---")
    for n in sizes:
        arr = dist_func(n)

        t_heap = time_algorithm(heap_sort, arr, inplace=True)
        t_quick = time_algorithm(quick_sort, arr)
        t_merge = time_algorithm(merge_sort, arr)

        print(f"Size {n}: HeapSort={t_heap:.5f}s, QuickSort={t_quick:.5f}s, MergeSort={t_merge:.5f}s")
```

```
--- Distribution: random ---
Size 1000: HeapSort=0.00000s, QuickSort=0.00000s, MergeSort=0.00624s
Size 5000: HeapSort=0.01487s, QuickSort=0.00152s, MergeSort=0.01817s
Size 10000: HeapSort=0.02596s, QuickSort=0.03135s, MergeSort=0.02480s
Size 20000: HeapSort=0.07089s, QuickSort=0.04026s, MergeSort=0.04769s

--- Distribution: sorted ---
Size 1000: HeapSort=0.00000s, QuickSort=0.01601s, MergeSort=0.00207s
Size 5000: HeapSort=0.01574s, QuickSort=0.00351s, MergeSort=0.00000s
Size 10000: HeapSort=0.03800s, QuickSort=0.01205s, MergeSort=0.03222s
Size 20000: HeapSort=0.07403s, QuickSort=0.02447s, MergeSort=0.03213s

--- Distribution: reverse ---
Size 1000: HeapSort=0.00000s, QuickSort=0.01585s, MergeSort=0.00000s
Size 5000: HeapSort=0.01477s, QuickSort=0.00600s, MergeSort=0.00000s
Size 10000: HeapSort=0.03813s, QuickSort=0.01169s, MergeSort=0.01672s
Size 20000: HeapSort=0.07651s, QuickSort=0.03341s, MergeSort=0.03702s
PS C:\Users\privat>
```

Discussing the observed results and relate them to the theoretical analysis.

After observing the output, the following observations were observed:

1. Quick Sort is the fastest on the random inputs (particularly for larger n).

   e.g. at n = 20000 quicksort time < heapsort and merge.

2. Heap Sort times grow steadily with n and are insensitive to input ordering (random / sorted / reversed look similar). Heap sort times increase roughly smoothly as size increases (consistent O(n log n) growth).
3. Merge Sort timings are stable across distributions but show some variability sometimes faster than heapsort (e.g. some sizes) and sometimes slower (depends on n and distribution).
4. Quick Sort degrades on already-sorted or reverse-sorted inputs if pivot strategy is poor.

Relation to Theoretical Analysis

Quick Sort on Random Data:

In theory, the average-case time complexity of Quick Sort is O(n log n)**,** achieved when the pivot divides the array into approximately equal parts. The implementation used in this experiment employed either a middle or randomized pivot, leading to balanced partitions and efficient

sorting. Consequently, Quick Sort performed fewer comparisons and recursive calls, resulting in superior practical performance compared to Heap Sort and Merge Sort, which have higher constant factors.

Heap Sort's Stability and Input Independence:

Heap Sort theoretically maintains $\Theta(n \log n)$ time complexity in the best, average, and worst cases, as it performs the same sequence of heap construction and extraction operations regardless of input order. The experimental results confirmed this consistency, showing similar runtimes across all distributions. The slightly higher execution times compared to Quick Sort can be attributed to Heap Sort's more random memory access pattern and greater number of element swaps.

Merge Sort Performance and Space Overhead:

Merge Sort also has a theoretical complexity of $\Theta(n \log n)$ for all input cases. Its stable performance across distributions aligns with this expectation. However, Merge Sort requires $O(n)$ additional memory for temporary arrays during the merging process. The overhead of memory allocation and list copying likely contributed to its variability and slightly higher execution times in certain cases.

Quick Sort Worst Case:

Quick Sort's theoretical worst-case complexity is $O(n^2)$, which occurs when pivot choices lead to highly unbalanced partitions (such as consistently selecting the smallest or largest element). This was not observed in the experiments because the pivot selection method (middle or randomized) avoided such cases. If a fixed pivot (e.g., the first element) were used on sorted data, the performance would likely have degraded significantly, potentially causing recursion depth errors.

Priority Queue Implementation and Applications

Part A: Priority Queue Implementation

1. Data Structure:

Choice of data structure for the heap:

I chose to represent the binary heap using a Python list (array). This choice is ideal because it allows for a simple and efficient mapping of parent-child relationships using index calculations (left = 2*i + 1, right = 2*i + 2). Lists also make it straightforward to implement heap operations like insertion, extraction, and heapify without the overhead of explicit tree nodes or pointers. For the size of tasks we are handling, this approach is both easy to implement and efficient, providing O(log n) time complexity for insertion and extraction operations.

Task class design:

I designed a Task class to represent individual tasks. Each task stores relevant information such as task_id, priority, arrival_time, and deadline. Using a class makes it easy to encapsulate task properties, pass tasks around as objects, and extend functionality later if needed (e.g., adding more attributes or methods for task management). This also allows the heap to compare tasks based on priority while keeping all other task information intact.

Choice of max-heap vs min-heap:

I implemented a max-heap, where the task with the highest priority is always at the root. This decision aligns with the scheduling algorithm that selects the highest-priority task first. Using a max-heap ensures that extracting the next task to schedule is efficient (O(log n)) and directly reflects the scheduling policy. If the scheduling criteria were based on the earliest deadline, a min-heap could have been chosen instead, where the task with the smallest deadline is prioritized.

2. Core Operations:

Full code available on "Priority_Queue.py" file.

Task Class:

```python
# Task Class Definition
class Task:
    def __init__(self, task_id, priority, deadline, arrival_time):
        self.task_id = task_id
        self.priority = priority
        self.deadline = deadline
        self.arrival_time = arrival_time

# Sample Tasks
task1 = Task(task_id=1, priority=5, deadline="2023-12-01", arrival_time="2023-11-01")
task2 = Task(task_id=2, priority=8, deadline="2023-11-15", arrival_time="2023-11-05")
task3 = Task(task_id=3, priority=3, deadline="2023-12-10", arrival_time="2023-11-10")

list_of_tasks = [task1, task2, task3]
n = len(list_of_tasks)
```

Insert:

```python
# Insert Function
def insert(heap, task):
    # Insert a new task into the heap and maintain the heap property.
    heap.append(task)
    i = len(heap) - 1

    # Bubble up to maintain max-heap property
    while i > 0:
        parent = (i - 1) // 2
        if heap[i].priority > heap[parent].priority:
            heap[i], heap[parent] = heap[parent], heap[i]
            i = parent
        else:
            break
```

Extract:

```python
# Extract Max Function
def extract_max(heap):
    # Remove and return the task with the highest priority.
    if len(heap) == 0:
        return None

    root_task = heap[0]
    heap[0] = heap[-1]
    heap.pop()
    heapify(heap, len(heap), 0)
    return root_task
```

Increase:

```python
# Increase Key Function
def increase_key(heap, task_index, new_priority):
    # Increase the priority of a task and bubble it up if necessary.
    if new_priority < heap[task_index].priority:
        print("New priority is lower than current - use decrease_key instead.")
        return

    heap[task_index].priority = new_priority

    # Bubble up
    while task_index > 0:
        parent = (task_index - 1) // 2
        if heap[task_index].priority > heap[parent].priority:
            heap[task_index], heap[parent] = heap[parent], heap[task_index]
            task_index = parent
        else:
            break
```

Is Empty:

```python
# Is Empty Function
def is_empty(heap):
    # Check if the heap is empty.
    return len(heap) == 0

# Example Usage
if __name__ == "__main__":
    # Insert a new task
    task4 = Task(task_id=4, priority=10, deadline="2023-12-20", arrival_time="2023-11-15")
    insert(list_of_tasks, task4)
```

Output:

```
Extracting tasks in order of priority:
Task ID: 3, Priority: 12, Deadline: 2023-12-10, Arrival Time: 2023-11-10
Task ID: 4, Priority: 10, Deadline: 2023-12-20, Arrival Time: 2023-11-15
Task ID: 2, Priority: 8, Deadline: 2023-11-15, Arrival Time: 2023-11-05
Task ID: 1, Priority: 5, Deadline: 2023-12-01, Arrival Time: 2023-11-01
PS C:\Users\priyat>
```

1. Design Choices

- **Heap Representation:** I chose a Python list to implement the binary heap because it allows simple index-based calculations for parent-child relationships and supports efficient heap operations (insert, extract) in O(log n) time.
- **Task Representation:** Each task is represented by a Task class containing task_id, priority, deadline, and arrival_time. This encapsulates all relevant information in a structured way, making it easy to manage tasks in the heap.
- **Heap Type:** I implemented a max-heap, ensuring that the task with the highest priority is always at the root. This aligns with a scheduling algorithm that selects the most important task first.

2. Implementation Details

- **Heapify:** A recursive heapify function maintains the max-heap property, ensuring that for each node, its priority is greater than or equal to the priorities of its children.
- **Insertion:** The insert() function appends a task at the end and bubbles it up, maintaining the heap property. This ensures new tasks are added efficiently.
- **Extraction:** The extract_max() function removes the root (highest priority task), replaces it with the last task, and heapifies down. This guarantees that each extraction returns the correct next task to schedule.
- **Increase Key:** The increase_key() function allows modifying a task's priority upward and repositioning it to maintain heap order.
- **is_empty():** A simple function to check if the heap contains any tasks

3. Scheduling and analysis

```
Extracting tasks in order of priority:
Task ID: 3, Priority: 12, Deadline: 2023-12-10, Arrival Time: 2023-11-10
Task ID: 4, Priority: 10, Deadline: 2023-12-20, Arrival Time: 2023-11-15
Task ID: 2, Priority: 8, Deadline: 2023-11-15, Arrival Time: 2023-11-05
Task ID: 1, Priority: 5, Deadline: 2023-12-01, Arrival Time: 2023-11-01
PS C:\Users\priyat>
```

Observation: Tasks were scheduled strictly in order of decreasing priority. Task 3, whose priority was increased during execution, was scheduled first, followed by tasks 4, 2, and 1.

Analysis: The results show that the max-heap correctly maintains the highest-priority task at the root after every insertion or priority update. All heap operations (insert, extract_max, increase_key) work in O(log n) time, making the scheduler efficient for a dynamic set of tasks.

Time Complexity analysis for priority queue operations.

Operation	**Time Complexity**
Insert	O(log n)
Extract_max	O(log n)
Increase_key	O(log n)
Is_empty	O(1)

1. **Insertion (insert(task)):**
    - A new task is added at the end of the heap (O(1)) and then bubbled up to restore the heap property.
    - In the worst case, the new task may travel from the leaf to the root, which requires traversing the height of the heap.

- Heap height: O(log n)
- Time complexity: O(log n)

2. **Extraction (extract_max()):**
    - The root (highest-priority task) is removed, and the last element is moved to the root.
    - The heap is then heapified down to restore the max-heap property.
    - Again, in the worst case, the element may move from the root to a leaf, traversing the height of the heap.
    - Time complexity: O(log n)
3. **Increase Key (increase_key()):**
    - After increasing a task's priority, it may need to bubble up to maintain the heap property.
    - The number of steps is bounded by the height of the heap.
    - Time complexity: O(log n)
4. **Check Empty (is_empty()):**
    - Simply checks the length of the heap.
    - Time complexity: O(1)

In conclusion, the priority queue implemented using a max-heap and a Task class provides an efficient and flexible solution for task scheduling. The heap ensures that the highest-priority task is always selected first, while operations like insert, extract_max, and increase_key maintain the heap property with O(log n) time complexity. Using a Python list for the heap and encapsulating task details in a class makes the implementation simple, clear, and easy to extend. The scheduling results confirm that tasks are executed correctly in order of priority, demonstrating the effectiveness and correctness of the design.