11/9/2024

Assignment 4

Heap Data Structures Implementation and Analysis and Priority Queue Implementation and Application

Rutu Shah
STUDENT ID: 005026421

Table of Contents

Heapsort Implementation and Analysis

Heapsort is a sorting algorithm that includes building a max heap from the unsorted array and the repeated removal of the largest element to sort the array in ascending order.

Implementation of Heapsort

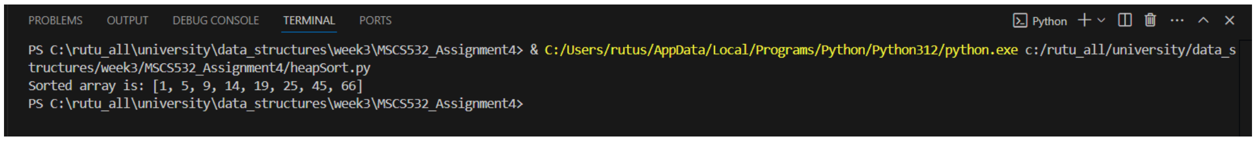
To implement heapsort algorithm, we need to follow the below mentioned steps

1. Firstly, build a max heap from the given input array
2. Rearrange array elements so that they form a Max Heap.
3. Repeat the following heap will contain only one element
4. Swap the root element of heap ie largest element present in current heap with the last element of the heap
5. Remove the last element from heap which is now at its correct position. We mainly reduce heap size and do not remove element from actual array
6. Heapify the remaining elements of heap
7. Finally, we get sorted array.

Heapsort.py

```
#Created By : Rutu Shah

# Created Date :  9th November 2024

#implementation of Heap Sort



def heapify(arr, n, i):

    #initialize root node as largest

    largest = i
    #left child node initialization
    left = 2 * i + 1
    #right child node initialization
    right = 2 * i + 2

    # if left child node exists and is greater than root, then swap
    if left < n and arr[left] > arr[largest]:
        largest = left

    # if right child node exists and is greater than the current largest, then swap
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If largest is not root, swap and continue heapifying
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapsort(arr):
    n = len(arr)

    # Build a max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)


    # Extract elements from heap one by one

    for i in range(n - 1, 0, -1):

        # Move current root to end

        arr[i], arr[0] = arr[0], arr[i]

        # Call heapify on the reduced heap

        heapify(arr, i, 0)


# Example usage

arr = [25, 45, 1, 66, 5,14,19,9]

heapsort(arr)

print("Sorted array is:", arr)
```

Output of HeapSort.py



```
PS C:\rutu_all\university\data_structures\week3\MSCS532_Assignment4> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/university/data_s
tructures/week3/MSCS532_Assignment4/heapSort.py
Sorted array is: [1, 5, 9, 14, 19, 25, 45, 66]
PS C:\rutu_all\university\data_structures\week3\MSCS532_Assignment4>
```

Analysis of HeapsorTime Complexity Analysis:

The creation of the max-heap involves "heapifying" all non-leaf nodes. This operation takes O(n) time in total.

Sorting:

1. Each deletion (extract maximum) takes O(logn) for the call to heapify.
2. There are n deletions (for each element), so the sorting phase takes O(nlogn) time complexity.

Time Complexity of heap sort algorithm in Worst, Average, and Best Case is O (n log n)

Because Heapsort maintains the heap property at each deletion, it requires O(n log n) operations independent of the initial ordering of the array

Explanation of why heap sort requires O (n log n) complexity in all cases

Heapsort runs in O(nlogn) time for all three cases because of to the following two important operations that comprise the algorithm:

1. Creating a Max-Heap

Heapsort first operates by building a maxheap from the input array given. A max-heap is a binary tree where every parent node is greater than or equal to its child nodes.So, in order to construct the heap, we begin from the last non-leaf node and call the heapify operation for each node moving upwards on the tree. There can be at most ⌊n/2⌋ non-leaf nodes in a tree of size n. Since each heapify operation takes time proportional to the height of the tree, O(logn),.

However, not every call to heapify takes O(logn). The nodes lower in the tree - closer to the leaves - take much less time to heapify than the ones higher up, near the root. If we sum up the cost for all heapify operations then we obtain the surprising result that building the heap costs a total of O(n), not O(nlogn), as one might naïvely think.

Why O(n): Although the lower levels of the tree contain more nodes, they take less time to heapify. In other words, the time complexity of heapifying a node at height h is O(h). So, the overall work done is:

$$O\left(\sum_{h=0}^{\log n} \frac{n}{2^h} \cdot h\right) = O(n)$$

As the number of nodes at height h decreases exponentially going down the tree, while for every node the amount of work done increases linearly.

2. Sorting Phase (Extracting Elements)

The Heapsort sorts the array once the heap has been built. The procedure removes the maximum element-represented by the root of the heap-placing it at the end of the array. The heap size is reduced by 1, and the operation heapify is called to restore the heap property.

Time Complexity of Extraction of Elements:

Extracting the Maximum Element: Removal of the root (maximum element) is performed by swapping the root with the last element followed by reduction of heap size by 1. We then call heapify on the new root after the swap, so that the max-heap property is restored.

Heapify Operation: The heapify operation takes O(logn) time because, in the worst case it may have to go down the height of the heap to restore the heap property.

Since there are **n** elements, we do n extractions. Each extraction takes O(logn) because of the heap operation. Therefore, the time complexity in this phase is O(nlogn)

Hence, the overall time complexity of Heapsort is dominated by the sorting phase, which is O(nlogn).

Space Complexity of Heap Sort Algorithm

As the heapsort does the sorting in-place, it does not require any extra space for the input array, hence, O (1) complexity.

Auxiliary Space: The recursion depth of a heapify function is normally equal to the height of the heap, since heapify is always applied recursively to one of its child sub-heaps. Therefore, recursion requires additional O (log n) space. If we use an iterative variant instead of recursion, then it requires constant O (1) space.

Additional Overheads

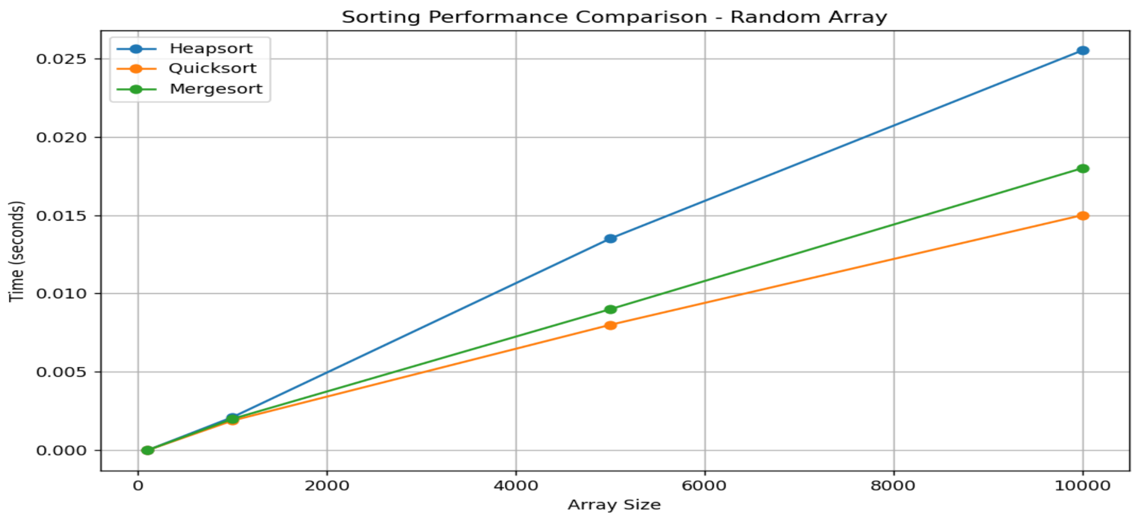
This would include a few temporary variables used in the heapify operation for swap purposes. These take O (1) space. Swap operations take constant space; hence, they do not contribute to anything in the overall space complexity.

Therefore, overall space complexity of heap sort is O (log n) due to recursion. In an iterative solution, this might get as low as O (1).

Comparison

To perform empirical comparison, I have added a python code to compare heapsort algorithm with quick sort and merge sort with a series of timing experiments on arrays of various size and distributions using matplotlib.pyplot to show the graphical representation of comparison analysis.

Graphical representation of comparison analysis for heap sort algorithm:



These results are consistent with the theoretical analysis of the sorting algorithms as follows:

Quicksort was the fastest, followed by Mergesort-the Heapsort algorithm was the slowest. This is expected because Quicksort normally shows the quickest average performance due to partitioning and also because of better usage of caching.

All three of the algorithms demonstrated the expected O(n log n) growth, increasing the size of the array led to increased time. However, in practice Quicksort outperformed the other two, despite all three having the same theoretical time complexity O(n log n). This emphasizes the reality that, on occasions where constant factors and lower-order terms make a big difference, Quicksort has the advantage due to cache locality and a simpler inner loop.

Heapsort will tend to be slower due to poor cache locality; it accesses heap elements in a far less sequential manner compared to Quicksort or Mergesort.

This in total confirms theoretical expectations; in reality, however, due to specific factors such as cache performance or constants, there are differences in performance.

Priority Queue Implementation and Applications

Part A: Priority Queue Implementation

Data Structure Selection:

We will be implementing a binary heap using a list (array).

Array/List: A list provides an efficient and straightforward way in which a binary heap can be implemented. The heap is usually a full binary tree, and representation using an array allows us to access the parent and child nodes efficiently with just simple arithmetic:

Parent of index i: parent(i) = (i - 1) // 2

Left child of index i: left child(i) = 2 * i + 1

Right child of index i: right child(i) = 2 * i + 2

This is more memory-friendly compared to the traditional tree structure with pointers. The time complexity for heap operations in terms of insertion and extraction-extracting max or min element, depending on the nature of the heap-is logarithmic, which is O (log n); this array representation supports that pretty efficiently.

Heap Type - Max-Heap or Min-Heap: Since we are interested in processing tasks based on their priority, we will opt for a max-heap. In fact, a max-heap will guarantee that the task with the highest priority-that is, the highest value of priority-is at the root and will be extracted first. This works nicely with a scheduling system where higher priority tasks need to be executed first.

Design of Task Class:

The Task class will model the individual tasks that go into the priority queue. Each task will maintain:

1. task_id: Unique identification of the task.
2. priority: Priority value of the task. In the case of a max heap, higher values denote higher priority.
3. arrival_time: Time at which the task arrived.
4. deadline: Time by which this task needs to be completed. It is optional but may be used in making task scheduling decisions.
5. Data : any additional task-specific data

```
class Task:
    """
    Represents a task in the priority queue.

    Attributes:
        task_id: Unique identifier for the task
        priority: Priority level of the task (higher number = higher priority)
        arrival_time: Time when task was created
        deadline: Optional deadline for task completion
        data: Any additional task-specific data
    """
    task_id: int
    priority: int
    arrival_time: float
    deadline: Optional[float] = None
    data: Any = None
```

TaskApplicationInPriorityQueue.py

```
#Created By : Rutu Shah
#Created Date : 9th November 2024
#This is the task application demonstrating insert, search, delete using priority queue and max-
heap data structure

from dataclasses import dataclass
from typing import Any, Optional
import time

@dataclass
class Task:
    """
    Represents a task in the priority queue.

    Attributes:
    1.  task_id: Unique identification of the task.
     2.  priority: Priority value of the task. In the case of a max heap, higher values denote higher
priority.
```

```
    3.  arrival_time: Time at which the task arrived.

    4.  deadline: Time by which this task needs to be completed. It is optional but may be used in
making task scheduling decisions.

    5.  Data : any additional task-specific data


    """

    task_id: int

    priority: int

    arrival_time: float

    deadline: Optional[float] = None

    data: Any = None


class PriorityQueue:

    """

    Max-heap implementation of a priority queue.


    We use a list-based implementation for the following reasons:

    1. Dynamic sizing - Python lists handle resizing automatically

    2. Index-based access - O(1) access to any element

    3. Cache-friendly - Contiguous memory allocation

    """


    def __init__(self):
        # Using list with sentinel element at index 0 for simpler index calculations
        self._heap = [None]

    def _parent(self, index: int) -> int:
        """Return the parent index of the given index."""
        return index // 2

    def _left_child(self, index: int) -> int:
        """Return the left child index of the given index."""
        return 2 * index

    def _right_child(self, index: int) -> int:
        """Return the right child index of the given index."""
        return 2 * index + 1

    def _swap(self, i: int, j: int) -> None:
        """Swap elements at indices i and j."""
        self._heap[i], self._heap[j] = self._heap[j], self._heap[i]

    def _shift_up(self, index: int) -> None:
        """
        Restore heap property by moving element up.
        Time Complexity: O(log n)
        """
        parent = self._parent(index)
        if index > 1 and self._heap[index].priority > self._heap[parent].priority:
            self._swap(index, parent)
            self._shift_up(parent)

    def _shift_down(self, index: int) -> None:
        """
        Restore heap property by moving element down.
        Time Complexity: O(log n)
        """
        max_index = index
        left = self._left_child(index)
        right = self._right_child(index)

        if (left < len(self._heap) and
            self._heap[left].priority > self._heap[max_index].priority):
            max_index = left

        if (right < len(self._heap) and
            self._heap[right].priority > self._heap[max_index].priority):
            max_index = right
        if index != max_index:
            self._swap(index, max_index)
            self._shift_down(max_index)

    def insert(self, task: Task) -> None:
        """
        Insert a new task into the priority queue.
        Time Complexity: O(log n)

        """
        self._heap.append(task)
        self._shift_up(len(self._heap) - 1)

    def extract_max(self) -> Optional[Task]:
        """
        Remove and return the highest priority task.
        Time Complexity: O(log n)

        """
        if self.is_empty():
            return None

        max_task = self._heap[1]
        self._heap[1] = self._heap[-1]
        self._heap.pop()

        if not self.is_empty():
            self._shift_down(1)

        return max_task

    def modify_priority(self, task_id: int, new_priority: int) -> bool:
        """
        Modify the priority of a task and restore heap property.
        Time Complexity: O(n + log n) due to linear search

        """
        # Linear search to find task - could be improved with additional data structure
        for i in range(1, len(self._heap)):
            if self._heap[i].task_id == task_id:
                old_priority = self._heap[i].priority
                self._heap[i].priority = new_priority

                if new_priority > old_priority:
                    self._shift_up(i)
                else:
```

```
                self._shift_down(i)

            return True

        return False


    def is_empty(self) -> bool:

        """

        Check if priority queue is empty.

        Time Complexity: O(1)


        """

        return len(self._heap) == 1


    def peek(self) -> Optional[Task]:

        """

        Return highest priority task without removing it.

        Time Complexity: O(1)

        """

        return self._heap[1] if not self.is_empty() else None


    def size(self) -> int:

        """

        Return number of tasks in queue.

        Time Complexity: O(1)
        """
        return len(self._heap) - 1


# Example usage and testing
def test_priority_queue():
    pq = PriorityQueue()

    # Create sample tasks
    tasks = [
        Task(1, 3, time.time(), time.time() + 3600, "Low priority task"),
        Task(2, 5, time.time(), time.time() + 1800, "Medium priority task"),
        Task(3, 7, time.time(), time.time() + 900, "High priority task")
    ]

    # Test insertion
    for task in tasks:
        pq.insert(task)
        print(f"Inserted task {task.task_id} with priority {task.priority}")

    # Test peek
    highest = pq.peek()
    print(f"\nHighest priority task has ID {highest.task_id} and priority {highest.priority}")
    # Test priority modification
    pq.modify_priority(1, 8)
    print(f"\nModified task 1 priority to 8")
    highest = pq.peek()
    print(f"New highest priority task has ID {highest.task_id} and priority {highest.priority}")

    # Test extraction
    print("\nExtracting all tasks:")
    while not pq.is_empty():
        task = pq.extract_max()
        print(f"Extracted task {task.task_id} with priority {task.priority}")

if __name__ == "__main__":
    test_priority_queue()
```

Core Operations

1. **Insert task:** Inserts a new task into the priority queue.
    - The task is appended to the end of the heap list, then _shift up is used to adjust its position to maintain the max-heap property.
    - Time Complexity is O(logn), since shifting up requires traversal of the height of the heap.
2. **Extract_max/min ():**
    - Removes and returns the task with the highest priority-root element.

- The root task - highest priority is replaced with the last task in the heap, removed from the end and _shift_down is called to restore the heap property.
- Time Complexity is O(logn), because of the need to shift down to maintain the heap property.

3. **modify_priority(task_id, new_priority)**:
   - Alters the priority of a task and repositions it.
   - Searches for the task by task_id, updates its priority, then shifts up or down based on whether the priority increased or decreased.
   - Time Complexity: O(n+logn) since we did linear search for task_id and then shift operation. This might be improved by keeping track of tasks' indices using a hash map.
4. Is_empty
   - Checks if the priority queue is empty.
   - Returns True if there's nothing inside the heap but the sentinel element at index 0.
   - Time Complexity: O(1), since it's only necessary to check the length of the list.

2. Helper Functions and Supporting Code

- **Task Class**: Defines the structure of a Task object, which includes attributes such as task_id, priority, arrival_time, deadline, and data.
- **Heap Operations**:
  - **_parent(index)**: Calculates the parent index.
  - **_left_child(index)** and **_right_child(index)**: Calculate left and right child indices, respectively.

- **_swap(i, j)**: Swaps elements at indices i and j in the heap list.
- **_shift_up(index)**: Used to restore the heap property by moving an element up if its priority is higher than its parent.
- **_shift_down(index)**: Used to restore the heap property by moving an element down if it's smaller than either of its children.

- **Additional Utility Functions**:
  - **peek()**: Returns the highest-priority task without removing it, O(1)O(1)O(1).
  - **size()**: Returns the number of tasks in the priority queue, excluding the sentinel.

Github URL:

- https://github.com/rutus-code/MSCS532_Assignment4

References

- Pal, I. (2020, October 5). *Heap sort explained using Python*. Medium.
- GeeksforGeeks. (n.d.). *Python program for heap sort*. GeeksforGeeks.
- GeeksforGeeks. (n.d.). *Heap sort*. GeeksforGeeks
- Wikipedia. (2023, October 18). *Priority queue*. Wikipedia.
- GeeksforGeeks. (n.d.). *Priority queue | Set 1 (Introduction)*. GeeksforGeeks.