CS 600 A - Advanced Algorithms – Homework 3

Name: Deepali Nagwade | 20013393

Question 5.7.11: Is there a heap T storing seven distinct elements such that a preorder traversal of T yields the elements of T in sorted order? How about an inorder traversal? How about a postorder traversal?

Answer: In the case of a binary heap, elements are arranged such that the parent node is always greater than or equal to (in a max heap) or less than or equal to (in a min heap) its child nodes. However, there isn't a strict order between left and right child nodes. With this structure in mind, let's re-examine the traversal orders for a heap containing seven distinct elements:

1. Preorder Traversal (Root, Left, Right):

   - In a max heap, the preorder traversal will start with the maximum element (the root), but as there's no particular order between left and right child nodes, this traversal won't yield a sorted sequence.

   - In a min heap, the root is the smallest element, but again, the lack of order between left and right child nodes will prevent a sorted sequence.

2. Inorder Traversal (Left, Root, Right):

   - Neither a max heap nor a min heap will yield a sorted sequence with an inorder traversal due to the reasons outlined above.

3. Postorder Traversal (Left, Right, Root):

   - Similarly, neither heap type will yield a sorted sequence in postorder traversal. Only reversed sorted order can be obtained in this case.

```
          98
        /    \
      32      77
      / \     / \
    9   20  40   43
```

The inherent structure of a binary heap does not lend itself to producing a sorted sequence through standard tree traversal methods. For obtaining a sorted sequence from a heap, heap sort algorithm can be used, which repeatedly extracts the minimum (or maximum) element from the heap and reconstructs the heap until it is empty.

Question 5.7.24: Let T be a heap storing n keys. Give an efficient algorithm for reporting all the keys in T that are smaller than or equal to a given query key x (which is not necessarily in T). For example, given the heap of Figure 5.6 and query key x = 7, the algorithm should report 4, 5, 6, 7. Note that the keys do not need to be reported in sorted order. Ideally, our algorithm should run in O(k) time, where k is the number of keys reported.

Answer: To efficiently report all keys in a heap T that are smaller than or equal to a given query key x, we can perform a modified version of a heap traversal. Here's an algorithm that runs in O(k) time, where k is the number of keys reported:

Algorithm: ReportKeysSmallerThanOrEqualToX

Input:

heap: A binary heap storing n keys.

x: The query key for which we want to find keys in the heap that are smaller than or equal to x.

Output:

result: A list of keys from the heap that are smaller than or equal to x.

Procedure:

- Initialize an empty list result to store the keys that meet the condition.
- Create an empty stack stack to perform a modified heap traversal.
- Push the root node of the heap onto the stack (stack.push(heap[0])).
- While the stack is not empty:
    - ➢ Pop a node from the stack
    - ➢ Find the indices of the left and right children of node in the heap:

      left_child_idx = 2 * index_of(node) + 1

      right_child_idx = 2 * index_of(node) + 2

- If left_child_idx is within the bounds of the heap, push the left child onto the stack (stack.push(heap[left_child_idx])).
- If right_child_idx is within the bounds of the heap, push the right child onto the stack (stack.push(heap[right_child_idx])).

- Retune result of the output

This Algorithm runs in O(k) time

Question 5.7.28: In a discrete event simulation, a physical system, such as a galaxy or solar system, is modeled as it changes over time based on simulated forces. The objects being modeled define events that are scheduled to occur in the future. Each event, e, is associated with a time, te,in the future. To move the simulation forward, the event, e, that has smallest time, te, in the future needs to be processed. To process such an event, e is removed from the set of pending events, and a set of physical forces are calculated for this event, which can optionally create a constant number of new events for the future, each with its own event time. Describe a way to support event processing in a discrete event simulation in O(logn) time, where n is the number of events in the system.

Answer: To support event processing in a discrete event simulation in O(log n) time, where n is the number of events in the system, we can use a priority queue data structure. A priority queue allows we to efficiently retrieve and process events with the smallest future time (te) in logarithmic time complexity.

To process events in a discrete event simulation, we can use the following algorithm:

- **Maintain priority Queue:** Use a priority queue data structure, such as a binary heap or a Fibonacci heap, to maintain the set of pending events. The key for each element in the priority queue is the event time (te). The event with the smallest te will be at the top of the priority queue.

- **Inserting Events:** When a new event is created and needs to be scheduled in the future, insert it into the priority queue with its associated event time (te).

- **Processing Events:** To move the simulation forward, perform the following steps repeatedly until the simulation's end condition is met: Pop the event with the smallest te from the priority queue. This can be done in O(log n) time since we are using a priority queue. Remove the event from the set of pending events.

- **Creating New Events:** Process the selected event by calculating the physical forces associated with it. Optionally, generate a constant number of new events for the future, each with its own event time. These new events can be inserted into the priority queue.

  By using a priority queue, we ensure that we always process the event with the smallest event time efficiently. The insertion and removal of events from the priority queue takes O(log n) time in most commonly used priority queue implementations, which allows we to maintain good performance as the number of events (n) in the system grows.

  Question 6.7.13: Dr. Wayne has a new way to do open addressing, where, for a key k, if the cell h(k) is occupied, then he suggests trying (h(k)+i · f(k)) mod N, for i =1 ,2,3,...,until finding an empty cell, where f(k) is a random hash function returning values from 1 to N −1. Explain what can go wrong with Dr. Wayne's scheme if N is not prime.

Answer: Using Dr. Wayne's open addressing scheme, which employs the formula (h(k) + i * f(k)) mod N to resolve collisions and find an empty cell, can face various complications when N is not a prime number. Here are some of the potential challenges associated with this approach:

- **Heightened Collision Incidence:** Due to the limitations, employing a non-prime N increases the probability of collisions. As the probing sequence explores a restricted set of slots due to factors and non-coprime values, it becomes more probable that multiple keys will collide and map to the same slots, resulting in an upsurge in collision frequency.
- **Complexity in Tuning Parameters:** Determining the appropriate parameters, such as the choice of N and the characteristics of the random hash function f(k), can be more challenging when N is not prime. This can make it harder to optimize the performance of the open addressing scheme.
- **Common Factors:** If N is not a prime number, it may have factors other than 1 and itself. Consider the presence of a factor, denoted as 'm,' where 'm' is a positive integer greater than 1 and less than N. In such cases, when the hash function f(k) produces values within the range [1, N-1], it may result in values that are not coprime with N.

- **Limited Range:** When f(k) yields values in the interval [1, N-1], the potential step sizes (i) are constrained. If N lacks primality, there might be a scarcity of distinct values that f(k) can generate. This restriction in step sizes can give rise to discernible patterns in the probing sequence, leading to suboptimal collision handling and the risk of performance bottlenecks.
- **Uneven Distribution**: In open addressing, the objective is to evenly distribute data items across the hash table. When N is a prime number, it offers better prospects for achieving a uniform distribution and minimizing the formation of clusters. Conversely, non-prime values for N can lead to uneven distributions, undermining the hash table's performance.

Question 6.7.17: Suppose you are working in the information technology department for a large hospital. The people working at the front office are complaining that the software to discharge patients is taking too long to run. Indeed, on most days around noon there are long lines of people waiting to leave the hospital because of this slow software. After you ask if there are similar long lines of people waiting to be admitted to the hospital, you learn that the admissions process is quite fast in comparison. After studying the software for admissions and discharges, you notice that the set of patients currently admitted in the hospital is being maintained as a linked list, with new patients simply being added to the end of this list when they are admitted to the hospital. Describe a modification to this software that can allow both admissions and discharges to go fast. Characterize the running times of your solution for both admissions and discharges.

Answer: To enhance the efficiency of the hospital software for admissions and discharges, consider implementing a double-ended queue (deque) data structure. A deque allows for quick insertions and removals at both ends, facilitating fast patient management. Here's how to modify the software:

- Utilize a Deque for Patient Management: Instead of maintaining the list of admitted patients as a linked list, employ a deque. Deques offer O(1) time complexity for insertions and removals at both the front and rear ends.

- Admissions: When admitting a new patient, simply add them to the rear end of the deque using an O(1) operation. This ensures speedy admissions as new patients can swiftly join the queue.
- Discharges: For patient discharges, remove them from the front end of the deque, also with O(1) complexity. Discharges become efficient because you can promptly remove the patient at the front of the queue.
- Runtime Characteristics:
  - Admissions: The deque data structure ensures rapid admissions with O(1) time complexity for adding patients to the rear end.
  - Discharges: Discharges are also expedited, with O(1) time complexity for removing patients from the front end of the deque.

By adopting a deque for patient management, you guarantee that both admissions and discharges operate with constant time complexity. This modification eliminates the need to traverse linked lists, resulting in swift operations for both front-end and rear-end actions, significantly improving software efficiency.