Problem Set 3

Ekkapot Charoenwanit
Efficient Algorithms

Problem 3.1. Stack and Queue

1) How do you implement a queue with two stacks? Demonstrate your idea with pseudo-code.

Solution: Let $s1$ and $s2$ be stacks. The enqueue and the dequeue operations can be implemented as follows:

First Alternative Implementation:

Algorithm 1 implements the enqueue operation

```
1: procedure enQueue(x)
2:     while ¬s1.empty() do
3:         s2.push(s1.top())
4:         s1.pop()
5:     s2.push(x)
6:     while ¬s2.empty() do
7:         s1.push(s2.top())
8:         s2.pop()
```

Algorithm 2 implements the dequeue operation

```
1: procedure deQueue
2:     if s1.empty() then
3:         error: Q is empty!
4:     x = s1.top()
5:     s1.pop()
6:     return x
```

Second Alternative Implementation:

Algorithm 3 implements the enqueue operation

```
1: procedure enQueue(x)
2:     s1.push(x)
```

2) Analyze the time complexity of the enqueue and dequeue operation of your two-stack queue.

Analysis: We will first analyze the time complexity of the first implementation.

The enqueue operation is relatively costly as we will see shortly. Suppose there are n items currently stored in the queue when an enqueue operation is called. Therefore, the first while loop runs for n iterations, each of which runs in $\Theta(1)$ time, until $s1$ is empty. Therefore, the first while loop costs $\Theta(n)$ time.
The enqueue operation then pushes the new item x onto $s2$, hence requiring $\Theta(1)$ time. The second while

Algorithm 4 implements the dequeue operation

```
1:  procedure DEQUEUE
2:      if s1.EMPTY() ∧ s2.EMPTY() then
3:          ERROR: Q IS EMPTY!
4:      if s2.EMPTY() then
5:          while ¬s1.EMPTY() do
6:              s2.PUSH(s1.TOP())
7:              s1.POP()
8:      x = s2.TOP()
9:      s2.POP()
10:     return x
```

loop runs for $n + 1$ iterations, each of which takes $\Theta(1)$ time. Therefore, the second while loop costs $\Theta(n + 1) = \Theta(n)$ time. Summing up all the work, the running time is $\Theta(n)$.

The dequeue operation performs a couple of constant time operations to retrieve the item at the top of the stack and pop it off the stack, hence requiring $\Theta(1)$ time, independently of the number of items currently in the queue.

Note: The enqueue operation is costly whereas the dequeue is cheap in this implementation.

Analysis: We will now analyze the time complexity of the second implementation.

The dequeue operation is relatively costly as we will see shortly. Suppose there are n items currently stored in the queue when a dequeue operation is called. The conditional branches in lines 2 and 4 cost $\Theta(1)$ time. Suppose that $s2$ is empty while $s1$ contains all the n items. In this situation, the while loop will runs for n iterations, each of which takes $\Theta(1)$ time. Therefore, the execution of the while loop costs $\Theta(n)$ time. Once outside of the while loop, the dequeue operation performs a couple of constant-time operations as shown in lines $8, 9$ and 10. Therefore, the dequeue operation costs $\Theta(n)$ time.

The enqueue operation simply pushes the new item x onto $s1$, hence requiring $\Theta(1)$ time, independently of the number of items currently in the queue.

Note: The dequeue operation is costly whereas the enqueue is cheap in this implementation.

Problem 3.2. Binary Heap

1) Verify whether the array $[15, 13, 9, 5, 12, 18, 7, 4, 0, 16, 2, 1]$ is a max heap. If not, show how to convert it to a max heap with BUILDMAXHEAP.

Solution: The nodes with the keys 9 and 12 violate the max heap property. Therefore, the given array is not a max heap. To turn this array into a max heap, we have to run the BUILDMAXHEAP routine on this array. Let us call this array A.

We run BUILDMAXHEAP from the node with index $\lfloor \frac{n}{2} \rfloor = \lfloor \frac{12}{2} \rfloor = 6$ (i.e. the internal node with the largest index) down to 1.

When $i = 6$, the binary tree rooted at $i = 6$ is already a max heap so the recursive call does nothing and returns.

When $i = 5$, the binary tree rooted at $i = 5$ is not a max heap as $A[5] < A[10]$ so we swap $A[5]$ and $A[10]$. After the swap, the tree rooted at $A[10]$ is a leaf so it is trivially a max heap and the recursive call

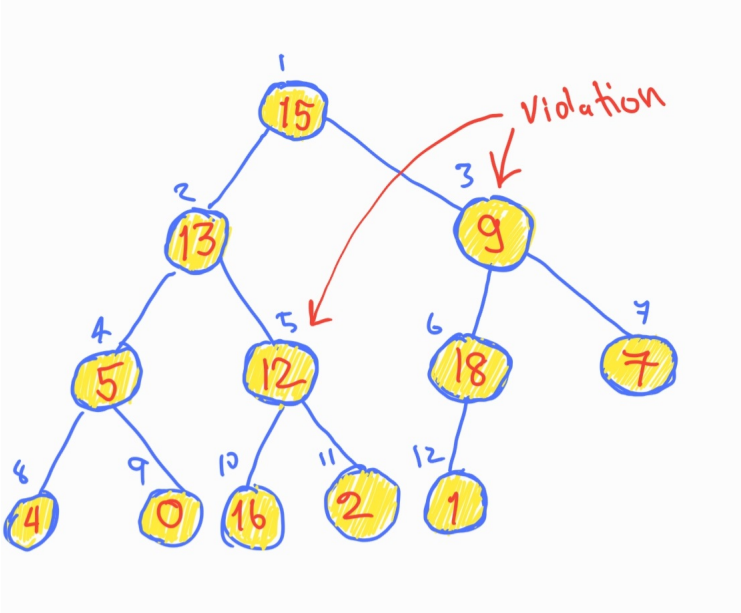

Figure 1: The nodes with keys 9 and 12 violate the max heap property.

returns.

When $i = 4$, the binary tree rooted at $i = 4$ is already a max heap so the recursive call does nothing and returns.

When $i = 3$, the binary tree rooted at $i = 3$ is not a max heap as $A[3] < A[6]$ so we swap $A[3]$ and $A[6]$. After the swap the tree rooted at $A[6]$ is a max heap so there is no need to recurse and the recursive call returns.

When $i = 2$, the binary tree rooted at $i = 3$ is not a max heap as $A[2] < A[5]$ so we swap $A[2]$ and $A[5]$. After the swap, the tree rooted at $A[5]$ is a max heap so there is no need to recurse and the recursive call returns.

When $i = 1$, the binary tree rooted at $i = 1$ is not a max heap as $A[1] < A[3]$ so we swap $A[1]$ and $A[3]$After the swap, the tree rooted at $A[3]$ is a max heap so there is no need to recurse and the recursive call returns. The entire array $A[1...12]$ is now a max heap.

2) Compute the minimum number of elements of a heap of height h.

Hint: Use the fact that the maximum number of elements of a heap of height h is $2^{h+1} - 1$. (We proved this in class.)

Proof: As the maximum number of elements of a heap of height h is $2^{h+1} - 1$, the maximum number of elements of a heap of height $h - 1$ is therefore $2^h - 1$.
Therefore, the minimum number of elements of a heap of height h is one more than the maximum number of elements of height $h - 1$.

$$h_{min} = (2^h - 1) + 1 = 2^h \qquad \square$$

3) Show that a binary heap with n elements has height $\lfloor \log_2 n \rfloor$.

Proof: Based on the result from 2), we have

$$2^h \leq n \leq 2^{h+1} - 1$$

$$2^h \leq n < 2^{h+1}$$

Taking log of both sides gives

$$h \leq \log n < h + 1$$

$$h = \lfloor \log n \rfloor \qquad \square$$

4) Show that the node with index $\lfloor \frac{n}{2} \rfloor$ is not a leaf in a binary heap tree with n elements.

Hint: Split your consideration into two cases: (Case 1) n is even and (Case 2) n is odd.

Reminder: In class, we proved that the elements in the subarray $A[\lfloor \frac{n}{2} \rfloor + 1, ..., n]$ are all leaves.

Proof: We have two cases to consider as follows:

Case I: n is even, i.e., $n = 2k$ for some $k \in Z$.

$$\left\lfloor \frac{n}{2} \right\rfloor = \left\lfloor \frac{2k}{2} \right\rfloor = \lfloor k \rfloor = k$$

If this node with index k has a left child, the index of the left child would be $2k = n$, which happens to be the last leaf with index n. Therefore, if n is even, the node with index $\lfloor \frac{n}{2} \rfloor$ is not a leaf.

Case II: n is odd, i.e., $n = 2k + 1$ for some $k \in Z$.

$$\left\lfloor \frac{n}{2} \right\rfloor = \left\lfloor \frac{2k+1}{2} \right\rfloor = \left\lfloor k + \frac{1}{2} \right\rfloor = k$$

If this node with index k has a left child, the index of the left child would be $2k = n$, which happens to be the last leaf with index n. Therefore, if n is odd, the node with index $\lfloor \frac{n}{2} \rfloor$ is not a leaf.

Therefore, the node with index $\lfloor \frac{n}{2} \rfloor$ cannot be a leaf. $\square$

5) Based on your work in 4), compute the number of leaves in an n-element heap.

Proof: There are n nodes in total, $\lfloor \frac{n}{2} \rfloor$ of which are internal nodes. Therefore, $n - \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil$ are leaf nodes. $\square$

Problem 3.3. Priority Queue

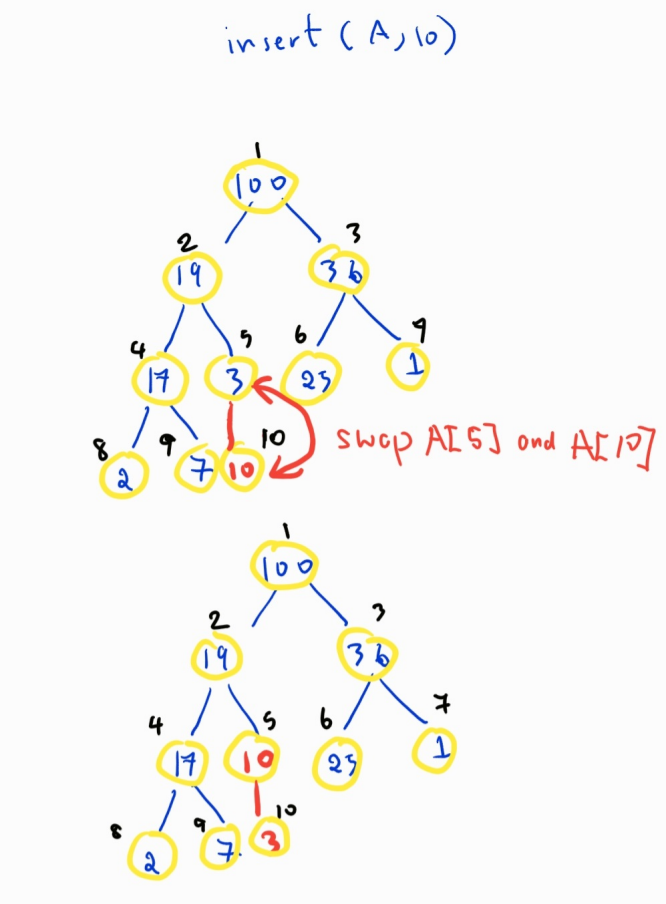

Figure 2: Inserting a node with key=10 into the max heap $A[1...9]$

1) Illustrate the operation of INSERT$(A, 10)$ on the max heap $[100, 19, 36, 17, 3, 25, 1, 2, 7]$.

Answer: As in Figure 2, the new value 10 is initially inserted in the newly reserved slot $A[10]$. After the insertion, the tree rooted at $A[5]$ is not a max heap. Therefore, we perform a swap between $A[5]$ and $A[10]$. After the swap, the tree rooted at $A[Parent(5)] \equiv A[2]$ is a max heap so the insertion operation terminates.

Note: The following operation is not part of the question. It is only for demonstration purposes.
Suppose that we want to insert a new value 20 into the heap immediately after the previous insertion of 10. The insertion is performed as shown in Figure 3.

2) Show how to implement a queue using a min priority queue.

Solution: Use a timestamp as a key value. When a new object is put into the min priority queue, associate the object being inserted with the current timestamp. Since time always monotonically increases, objects inserted earlier are guaranteed to always have smaller timestamps (key values) than objects inserted later. Therefore, the min priority queue will always remove the object with the smallest timestamp first. This way, the min priority queue can behave as a FIFO queue.

Note: This is only one of the many possible ways of implementing a FIFO queue using a min priority

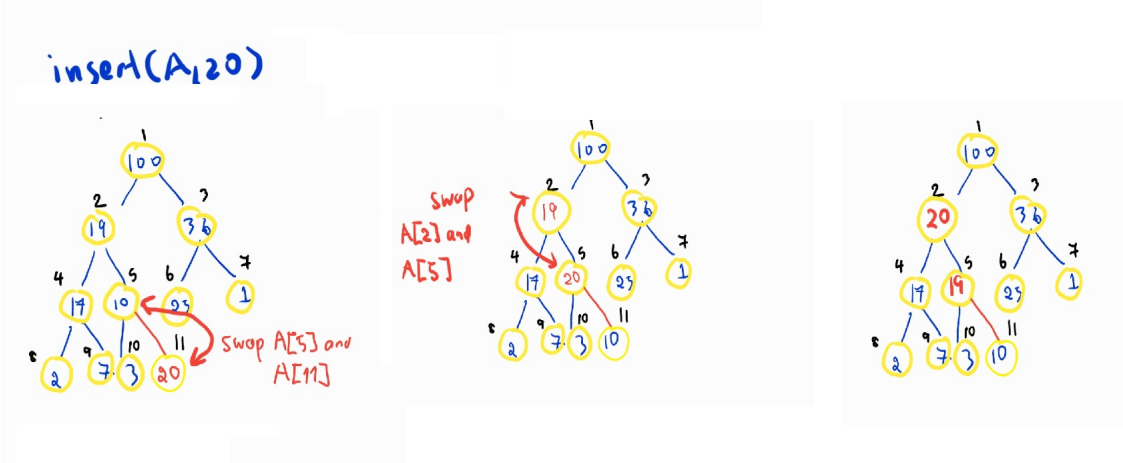

Figure 3: Inserting a node with key=20 into the max heap $A[1...10]$

queue.