Assignment N*2

Name: Harlee Ramos **Due Date:** 06/09/2024

1. In the algorithm **SELECT**, we achieved a worst-case linear time by grouping elements in groups of size 5. What is the worst-case time complexity of an alternative version of **SELECT** where we group elements in groups of size $2m + 1$ (where $m \geq 2$ is an integer constant)? Show your work.

Alternative Pseudocode based on lecture 4:

```
SELECT (A[1...n], i)
   1. Divide the n elements of A into $\left\lfloor \frac{n}{2m+1} \right\rfloor$ groups of 2m+1 elements each and at most
      1 group with fewer than 2m+1 elements.
   2. Sort 2m+1 elements in each group and find the median of each group.
   3. Let M [1... $\left\lfloor \frac{n}{2m+1} \right\rfloor$ ]be the array made up with the medians of each group.
   4. x = SELECT ($M, \left\lfloor \frac{\lceil \frac{n}{2m+1} \rceil + 1}{2} \right\rfloor$ )        // x is the “median of all the medians”.
   5. l = location of x in A.
   6. swap A[l] and A[n]  // swap x to the tail of A.
   7.  k = PARTITION (A[1...n])   // partition A with pivot = x, and x is the k-th
      smallest number in A.
   8. if (i == k)    return x
   9. if (i < k)     return SELECT (A[1...k-1], i)
   10.if (i > k)     return SELECT (A[k+1...n], i - k)
```

Time complexity analysis:

Line 1. Divide the n elements of Array takes $\Theta(n)$.

Line 2. Sort 2m+1 elements on each group takes $\Theta(1)$ time considering 2m+1 as a constant, then the time complexity is $\Theta(n)$.

Line 3. Let M [1... $\left\lfloor \frac{n}{2m+1} \right\rfloor$]be the array made up with the medians and **c**onsidering 2m+1 as a constant, then the time complexity is $\Theta(n)$.

Line 4. Recursion step for selecting the medias: this next involves recursively finding the median of medians in a subarray of size $\left\lfloor \frac{n}{2m+1} \right\rfloor$, This recursive call essentially reduces the problem size by a factor of 2m + 1.
then the time complexity is T($\frac{n}{2m+1}$).

Line 5. l = location of x in A, for this step the time complexity takes $\Theta(n)$.

Line 6. For swap A[l], takes $\Theta(1)$ time.

Line 7. For applying the partitioning takes $\Theta(n)$ time.

Line 8. For this if (i == k) statement, it is not necessary to apply a recursive call to the method SELECT, because it will only return x, then takes $\Theta(1)$ time.

Line 9. if (i < k) return **SELECT** (A[1...k-1], i), the time complexity can be expressed as $\mathrm{T}(k-1) \leq T\left(\frac{2m}{2m+1}n\right)$

Line 10. if (i > k) return **SELECT** (A[k+1...n], i - k), the time complexity can be expressed as $\mathrm{T}(n-k) \le T\left(\frac{2m}{2m+1}n\right)$

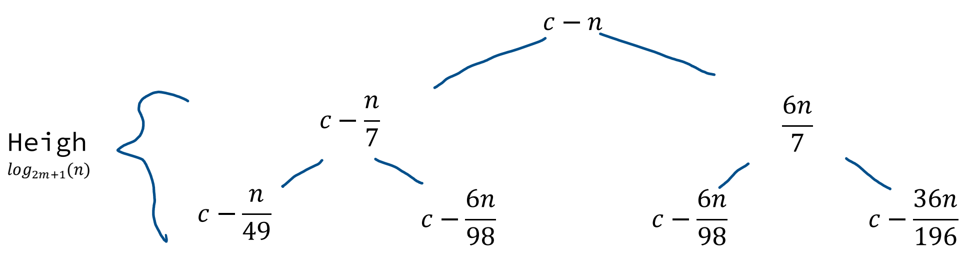
Tree Analysis

Subproblems:

- The size of the first subproblem is at most $\frac{n}{2m+1}$
- The size of the second subproblem is at most $\left(1-\frac{1}{2m+1}\right)\mathrm{n}$

$$1-\frac{1}{2\mathrm{m}+1}=\frac{2\mathrm{m}+1-1}{2\mathrm{m}+1}=\frac{2m}{2\mathrm{m}+1}=\left(\frac{2m}{2\mathrm{m}+1}\right)n$$

If m is an odd number. (m=3), then 2(3)+1= 7



Common ratio of the recursion tree 1

$$T(n) \le T\left(\frac{n}{7}\right)+T\left(\frac{6n}{7}\right)+\Theta(n)$$

Thus, $T(n) \le \Theta(n)$ which means $T(n)$ is $O(n)$.

The recursion tree for the modified SELECT algorithm uses a geometric progression with a common ratio of 1. Then the time complexity sums up to $T(n)=\Theta(nlogn)$, as the height of the tree is $log_{2m+1}(n)$ and the cost at each level is $\Theta(n)$.

2. Give an $O(n \lg k)$-time algorithm to merge k sorted arrays into one sorted array, where n is the total number of elements in all the input arrays. (Hint: Use a min-heap for k-way merging.)

```
1. MERGE-K-SORTED-ARRAYS(A)
2. Create an empty min-heap H
3. Initialize result as an empty array
4. for each array A[i] in arrays:
5. if A[i] is not empty:
6. Insert (A[i][0], i, 0) into H
7. while H is not empty:
8. (min_element, i, j) ← EXTRACT-MIN(H)
9. Append min_element to result
10.if j + 1 < length(A[i]):
11.Insert (A[i][j + 1], i, j + 1) into H
12.return result
```

```
1. EXTRACT-MIN(H)
2. if H is empty:
3. then error "heap underflow"
4. min_element ← H[1]
5. H[1] ← H[H.heap-size]
6. H.heap-size ← H.heap-size - 1
```

```
1. MIN-HEAPIFY(H, 1)
2. return min_element
3. MIN-HEAPIFY(H, i)
4. l ← LEFT(i)
5. r ← RIGHT(i)
6. if l ≤ H.heap-size and H[l] < H[i]
7. then smallest ← l
8. else smallest ← i
9. if r ≤ H.heap-size and H[r] < H[smallest]
10.then smallest ← r
11.if smallest ≠ i
12.then exchange H[i] with H[smallest]
13.MIN-HEAPIFY(H, smallest)
14.LEFT(i)
15.return 2 * i
16.RIGHT(i)
17.return 2 * i + 1
```

Time Complexity Analysis for the MERGE-K-SORTED-ARRAYS
Line 2: It takes O(1) time to create an empty min-heap H.
Line 3: It takes O(1) time to initialize the result array.
Lines 4-6: In order to loop across all k arrays. Then insert the first entry into the min-heap for each array that isn't empty. It takes $O(\text{logk})$time for each insertion

operation into the min-heap. This section requires O(logk) time since there are k insertions.
Line 7: Until the heap is empty, the while loop is executed. The while loop will run n times since we extract one element at a time and there are n elements total (sum of the lengths of all arrays).
Line 8: The EXTRACT-MIN procedure requires O(logk) time.
Line 9: It takes O(1) time to add an element to the result array.
Lines 10-11: We add the subsequent element to the min-heap if it exists in the array A[i]. Every insertion process requires O(logk) time.

Therefore, each iteration of the while loop takes O(logk)time for extraction and potentially another O(logk)time for insertion. Since the while loop runs n times, this part takes O(nlogk)time.
The total time complexity is $T(n) = \mathrm{O}(klogk) + \mathrm{O}(nlogk) = \mathrm{O}((n + k)logk)$, then considering k as a value smaller than n, it can be not considered in the total time complexity ,resulting in a time complexity of $\mathrm{O}(nlogk)$.

3. Provide pseudo-code for the operation **MAX − HEAP − DELETE** (A, i) that deletes the element in $A[i]$ from a binary max-heap A. In your code, you can call **MAX − HEAPIFY** and/or **HEAP − INCREASE − KEY** from the textbook/lecture notes directly if you want to. Analyze the time complexity of your algorithm.

Pseudocodes provided by the textbook:

- The **MAX-HEAPIFY** algorithm ensures that a binary tree maintains the max-heap property starting from a given node.

```
MAX-HEAPIFY(A, i)
1. l ← LEFT(i)
2. r ← RIGHT(i)
3. if l ≤ A.heap-size and A[l] > A[i]
4.    largest = l
5.    else largest = i
6. if r ≤ A.heap-size and A[r] > A[largest]
7.    largest = r
8. if largest ≠ i
9.    then exchange A[i] with A[largest]
10.        MAX-HEAPIFY(A, largest)
```

Time complexity for **MAX-HEAPIFY** $O(\log n)$

- The objective of the **HEAP-INCREASE-KEY** algorithm is to increase the value of element A[i] to a new key.

```
HEAP-INCREASE-KEY(A, i, key)
1. if key < A[i]
2.    then error "new key is smaller than current key"
3. A[i] = key
4. while i > 1 and A[PARENT(i)] < A[i]
5.     do exchange A[i] with A[PARENT(i)]
6.        i = PARENT(i)
```

Time complexity for **HEAP-INCREASE-KEY** $O(\log n)$

Pseudocode for MAX-HEAP-DELETE

```
MAX-HEAP-DELETE(A, i)
1. if i > A.heap-size or i < 1
2.     error "index out of bounds"
3. A[i] = A[A.heap-size]         // Replace A[i] with the last element
4. A.heap-size = A.heap-size - 1 // Reduce heap size
5. MAX-HEAPIFY(A, i)             // Ensure the heap property
6. if i > 1 and A[PARENT(i)] < A[i]
7.     HEAP-INCREASE-KEY(A, i, A[i]) // Fix the heap upwards if necessary
```

Time Complexity Analysis MAX-HEAP-DELETE
Line 1-2. In these lines, the if statement is just checking if the index i is valid and follows the criteria, and then this action just takes a constant time of $O(1)$.
Line 3. consists of assigning the key to the array, and this action just takes a constant time of $O(1)$.
Line 4. The reduction of the heap size is also an assignment; this action just takes a constant time of $O(1)$.
Line 5. takes $O(\log n)$ tie, because it consists in the implementation of the MAX-HEAPIFY method, and the time $O(\log n)$ is the worst case of transversing a node from the node in the index i to a down position on a leaf node position.
Line 6. In these lines, the if statement is just checking conditions, and then this action just takes a constant time of $O(1)$.
Line 7. This line is implementing the method HEAP-INCREASE-KEY, which will guarantee that after the deletion the heap will follow the max heap properties. In the worst case, this line will take $O(\log n)$ time.

Considering the weight of the operations with more time complexity, it can be concluded that the time complexity is $O(\log n)$.

4. Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A, b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give the smallest possible (with fewest nodes) counterexample to the professor's conjecture.

   As an aside, think about how to prove that your counterexample is the smallest, you don't need to show this proof in your solution.

Test with an A as an empty set

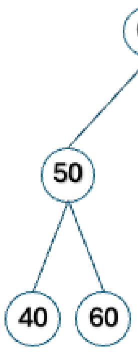
Key to search k=40

A (keys to the left of the search path): {} (empty set)

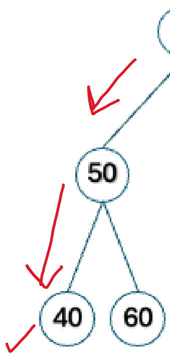
B (keys on the search path): {65,50,40}

C (keys to the right of the search path): {60}

Binary search tree representation



Search path



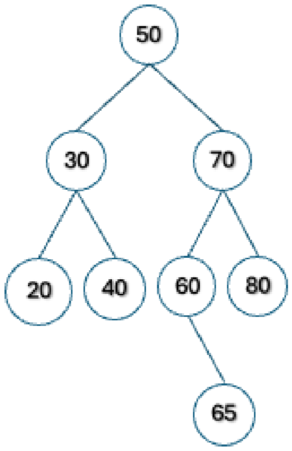
If any three keys $a∈A$, $b∈B$, and $c∈C$ satisfy $a≤b≤c$:

- Since A is empty, we don't have an a to test the conjecture. However, if we look at the keys in B and C,
  - $b∈B$: b can be 65, 50, or 40.
  - $c∈C$: c can be 60.
  - Evaluation of all combinations:
    - 50≤b≤c
- By looking at the path and the keys, we can see that 65>60, which does not satisfy $a≤b≤c$ if b=65 (from B) and c=60 (from C). As a result, the BST created above provides the smallest possible counterexample using the provided nodes {65, 50, 40, 60} and the search key 40. In this instance, the configuration disproves Professor Bunyan's conjecture.

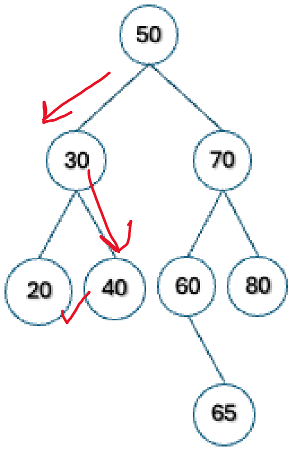
Test with not empty sets

Key to search k=40
A (keys to the left of the search path): {20}
B (keys on the search path): {50,30,40}
C (keys to the right of the search path): {70,60,80,65}

Binary search tree representation



Search path



Professor Bunyan's conjecture states that we must have $a \le b \le c$ for every $a \in A$, $b \in B$, and $c \in C$. We will select a=20∈ A, b=30∈ B, and c=60∈ C. The conjecture is satisfied by 20≤30≤60, according to the sets provided. However, if we examine a=20∈A, b=50∈B, and c=40∈C, we find that 20≤50 but 50≤40, proving the claim to be wrong. As a result, we have refuted Professor Bunyan's conjecture by showing that the sets do not meet $a \le b \le d$ with a=20, b=50, and c=40.