DAA HW 2

Pranav Tushar Pradhan N18401944 pp3051@nyu.edu

Udit Milind Gavasane N16545381 umg215@nyu.edu

Rohan Mahesh Gore N19332535 rmg9725@nyu.edu

1)a) $$S_n = \begin{cases} 1 & \to n=0 \\ 2 & \to n=1 \\ 3 & \to n=2 \\ S_{n-1} + S_{n-3} & \to n \geq 3 \end{cases} \qquad n \geq 0$$

To prove by induction - $\sum_{i=3}^{n} S_{i-3} = S_{n-3} \quad \forall\ n \geq 3$

(I) Induction basis :-

- when $n=3$

To prove - $\sum_{i=3}^{3} S_{i-3} = S_3 - 3$

LHS - $\sum_{i=3}^{3} S_{3-3} = \sum_{i=3}^{3} S_0 = 1$

RHS - $S_3 - 3 = (S_2 + S_0) - 3 = (3+1) - 3 = 1$

$\therefore$ LHS = RHS

Hence $\sum_{i=3}^{3} S_{i-3} = S_3 - 3$ is ok. $\because$ Basis case holds true

(II) Induction hypothesis :- Assuming the statement to be true for $n = k \geq 3$

$\therefore \sum_{i=3}^{k} S_{i-3} = S_k - 3$

(II) Induction step :- when $n = k+1$

To prove $\sum_{i=3}^{k+1} S_{i-3} = S_{k+1} - 3$

$LHS - \sum_{i=3}^{k+1} S_{i-3} = \sum_{i=3}^{k} S_{i-3} + S_{(k+1)-3}$

Using hypothesis,

$\sum_{i=3}^{k+1} S_{i-3} = (S_k - 3) + S_{k-2}$

$RHS - S_{k+1} - 3 = (S_k + S_{k-2}) - 3$, using recurrence relation

$LHS = RHS$

Hence by principle of induction, statement is true for all integers $n \geq 3$

b) To prove by induction - $S_{n+3} \geq \hat{\phi}^n \quad \forall n \geq 0$, $\hat{\phi} \in (1,2)$ is a root of $x^3 - x^2 - 1 = 0$

(I) Induction basis :-

when $n = 0$, $S_3 = 3 \geq \hat{\phi}^0 = 1$

when $n = 1$, $S_4 = 4 > \hat{\phi}^1$ as $\hat{\phi} < 2$

when $n = 2$, $S_5 = 5 > \hat{\phi}^2$ as $\hat{\phi} < 2$

(II) Inductive hypothesis :- Assuming the statement is true for all k, $0 \leq k \leq n$ ie $S_{k+3} \geq \hat{\phi}^k$ for $0 \leq k \leq n$

(iii) Induction step:-

To prove- $S_{(n+1)+3} \geq \hat{\phi}^{n+1}$

$S_{n+4} = S_{n+3} + S_{n+1}$ using recurrent relation

$\geq \hat{\phi}^{n} + \hat{\phi}^{n-2}$ using inductive hypothesis

$= \hat{\phi}^{n-2}(\hat{\phi}^2 + 1)$

$= \hat{\phi}^{n-2}(\hat{\phi}^3)$ as $\hat{\phi}^3 = \hat{\phi}^2 + 1$

$= \hat{\phi}^{n+1}$

$\therefore S_{n+4} \geq \hat{\phi}^{n+1}$ proving statement

Hence by principle of mathematic induction, statement is true for all integers $n \geq 0$

2) The problem is to identify the flaws in the induction steps

- Basis step-

The basis proof is correct as when only one horse is available, & all the horses (1) will have the same color.

- Induction hypothesis-

If n horses are assumed to be of same color, induction hypothesis remains true.

- Induction step-

Induction step of the problem is wrong.

- After removing 1 horse, we have n horses.
- Coming back to n=1, basis is true

For n=2,

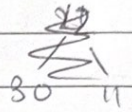
- we remove 1 horse, hence 1 horse remains which by basis is true.
- After we put that horse back and remove the earlier horse, statement isn't true all the time. Hence we cannot conclude that horses would have the same color.
- Proof assumes that there is a 'in between' horse

that remains in both sub groups of n horses. This is true for $n \geq 3$ but fails for $n=2$
Hence, proof fails as it cannot prove the case for $n=2$ for horses.
The base case was too small to support the inductive step.

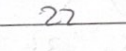
3) sequence of items → S_n = 22, 30, 11, 46, 35, 14, 25, 62, 40, 83, 55, 8

After 3 ~~insert opⁿ~~ ~~Endmk step~~

A = 22, 30, 11

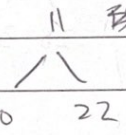


Initial step - A = 22
0



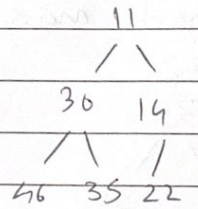
i) After 3 insert -

A = 11 30 22
0 1 2



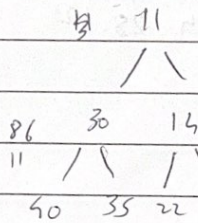
ii) After 3 inserts -

A = 11 30 14 46 35 22
0 1 2 3 4 5



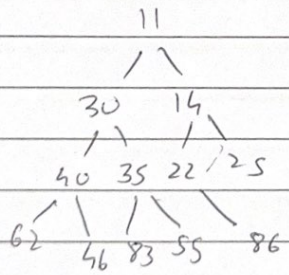
iii) After 3 inserts -

A = 11 30 14 40 35 22 25 62 46
0 1 2 3 4 5 6 7 8



iv) After 3 inserts -

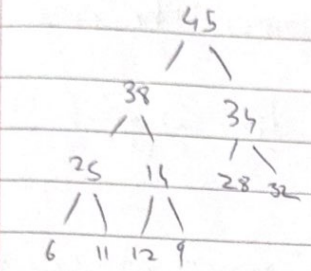
A = 11 30 14 40 35 22 25 62 46 83 55 86
0 1 2 3 4 5 6 7 8 9 10 11



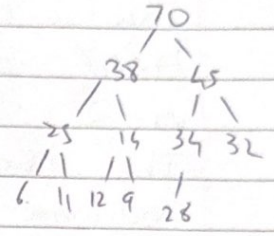
Final sorted min heap - 11, 30, 14, 40, 35, 22, 25, 62, 46, 83, 55, 8

4) First extract max

extract 70



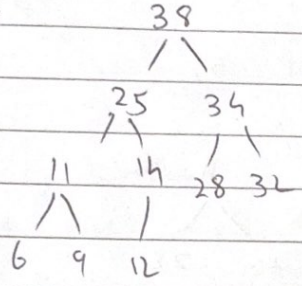
OG



second extract max

extract 45

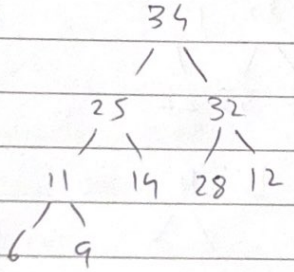
→



third extract max

extract 38

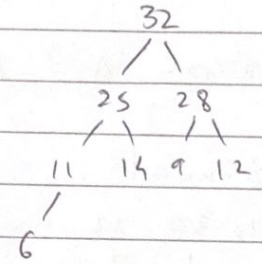
→



Fourth extract max

extract 34

→



We extract the root, swap it with the rightmost child node and perform heapify.

Q.5.a) Given n, we can find the binary representation of n.

(1) No. of bits in the binary representation of $n = \lceil \log_2(n) \rceil$.

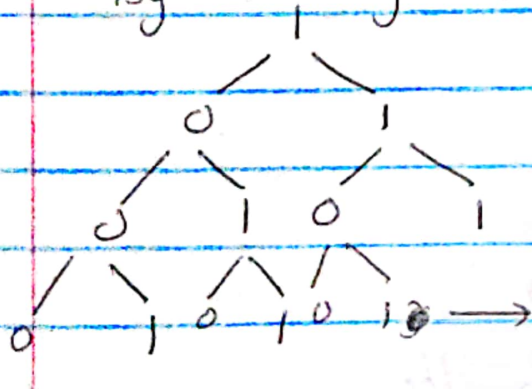
We know that a heap represented as a binary is a complete binary tree filled from left to right at the level while all other levels are filled.

Now, we need to traverse the given binary-tree from the binary string which represents n.

Let $n = 13$

Then $n = 1101$ (Binary form)

By following the rule Left → 0 & Right → 1



13^{th} element which is also the last as the tree is complete.

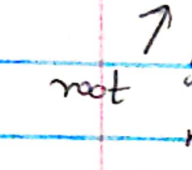
Pseudo-code of the algorithm:

1) Read the binary string from right to left i.e. from MSB to LSB.

MSB is the root.

Let the string be: 1011101....

We will store the binary string in an string array & traverse the tree as follows:




```
while (string[i] != '\0') {
    let node = root
    If string[i] = 0
        node = node → left
    If string[i] = 1
        node = node → right }
    return node;
```

Algorithm to traverse to last element

No. of bits in string = $O(\log n)$

∴ Time complexity for finding the last element = $O(\log n)$ as it takes $O(\log n)$ to reach the last element.

b.) In a min-heap, extracting min is a complex operation since the root is the smallest element, so we need to replace root with a new root.

Extract_min (Q) ⇒

1) We know root is the smallest element.
2) Replace the root node with the last element as new root
3) Restore the heap property by moving the last element to it appropriate position.

→ Now, from part a) we know to find last element in O(log n) time.

→ Replace it with root and perform swap operations to shift the new root to its correct position.
→ To do this, compare it with both its children and swap with the smallest one
→ Repeat this till heap is restored.
This will take a max of O(log n) steps since height of tree is O(log n).

∴ Total Time complexity of Extract-min(Q):

$O(\log n) + O(\log n) = O(\log n)$

$O(\log n)$ → Find last node

$O(\log n)$ → Restore heap for new root

Similarly for Insert (Q, x)

1) Find the last element in $O(\log n)$
2) Insert in next position of last node
3) Satisfy heap property with new element:
   a] Compare with parent & swap if parent is greater than x.
   b] Repeat till x is in its correct position.

∴ Time complexity of Insert (Q, x):

$O(\log n) + O(\log n) = O(\log n)$

$O(\log n)$ → Find last node

$O(\log n)$ → Restore heap for new element

DAA HW2 Question 6

```
#include <iostream>
#include <vector>
using namespace std;

void Traverse(const vector<int> &maxHeap, const int &k, const int &x, int &counter, int index)
{
    if (counter == k) // found k greatest elements
        return;

    if (index < maxHeap.size())
    {
        if (maxHeap[index] >= x) // accessing the element = cost O(1)
        {
            counter++; // found a element >= x
            Traverse(maxHeap, k, x, counter, index * 2);       // traverse left child : No Cost
            Traverse(maxHeap, k, x, counter, (index * 2) + 1); // traverse right child : No Cost
        }
    }
}

int main()
{
    // starting heap from index 1 for easy readability of alorithm and code
    vector<int> maxHeap = {-1, 100, 50, 70, 1,2,3,4};

    int x = 2;
    int k = 7;
    int counter = 0;
    Traverse(maxHeap, k, x, counter, 1); // 1 = root element (largest number)
    if (counter == k)
        cout << "yes\n";
    else
        cout << "no\n";
    return 0;
}
```

Time Complexity = O(k)
Space Complexity = O(1)
The algorithm's approach and description are handwritten and are given below.

Q6

← The algorithm is demonstrated in the above C++ code.

← "We are assuming array indexes from "1" rather than "zero" for easy readability of code / algorithm.

== Algo. Logic

1. For "k^{th}" greatest no. to be $\geq x$, even $(k-1)^{th}$ greatest no. should be $\geq x$, and so on, until $(k-i)^{th}$ no., where $i = 1$ to $k-1$.

∴ For $i = k-1$

$k - i \Rightarrow k - (k-1) \Rightarrow +1$ — root element.

∴ the root element at index 1 should also be greater than/equal to x.

2. Considering the property of "implicit binary max-heap" that: Parent $\geq$ any child.

&

∴ if root $< x \Rightarrow$ all its children $< x$

∴ No use of exploring the sub-tree further.

↓

This applies to any parent node & any sub-tree in the max-heap.

3. Also, as we DO NOT NEED TO find the k^{th} largest number in the heap and only focus on its relationship with x,
Therefore we "FIND & COUNT Numbers $\geq x$"

= This is done using DFS (Depth-First-Search) to traverse the heap.
- It will start at root
- Then it will visit left_child with its sub_tree
- and then right_child with its sub_tree
→ [If only they satisfy the condition $\geq x$]

- If current node $\geq x$ → Increment counter(c) and visit its children.

- If counter(c) == k, then we have found "k" numbers $\geq x$.
It is possible that it may not contain the "k^{th} largest element in max-heap" but in that case
- we have a no. that is $\geq x$ and smaller than "k^{th} largest element".
∴ "k^{th} largest element" would OBVIOUSLY be greater than equal to x. - (According to Point 1.).

4. Time Complexity

- Algo. runs max for "2k-1" elements or "N" elements (if N < 2k-1 or N < C).

- No. of nodes (C) visited with values ≥ x will be "at most k".

- A scenario where current_node < x is when it is a child of a visited_node in C whose value ≥ x.

=

- ∴ For each "C" we will visit its 2 children ∴ 2C
Also, for the last C^{th} node when we have C == k, we dont need to visit C^{th} node's children.

∴ Total nodes visited = All C-1 nodes with their 2 children + last C^{th} node

= 2 * (C-1) + 1

= 2C - 2 + 1

= 2C - 1 elements

OR 2k - 1 elements.

∴ TC = O(2C-1) = O(2k-1) = O(C) = O(k)

because C ≤ k.

5. Space Complexity = O(1). because we are using constant memory space for "counter" variable - denoted by "c" in our algorithm, which does not change according to array size.