CS-GY 6033: Homework #2

Due on October 4, 2023

Professor Yi-Jen Chiang

Runze Li
Nxxxxxxxx
rl50xx@nyu.edu

Tzu-Yi Chang
Nxxxxxxxx
tc39xx@nyu.edu

Jiayi Li
Nxxxxxxxx
jl156xx@nyu.edu

September 30, 2023

1. (a) Basis: when $n=4$

$\sum_{i=4}^{4} S_{i-4} = S_0 = 1 \quad S_4 - 4 = S_3 + S_0 - 4 = 4 + 1 - 4 = 1$

$\therefore \sum_{i=4}^{4} S_{i-4} = S_4 - 4$ is true

Assume: it's true for $n=k$

$\sum_{i=4}^{k} S_{i-4} = S_k - 4$

Induction Step: $\sum_{i=4}^{k+1} S_{i-4} = \sum_{i=4}^{k} S_{i-4} + S_{k+1-4}$

$= S_k - 4 + S_{k-3}$

$\because S_n = S_{n-1} + S_{n-4}$

$\therefore \sum_{i=4}^{k+1} S_{i-4} = S_k + S_{k-3} - 4$

$= S_{k+1} - 4$

So the statement is true for $n=k+1$

$\therefore \sum_{i=4}^{n} S_{i-4} = S_n - 4$ for all $n \geq 4$

(b) Basis: ① when $n=0$

$S_{0+4} = S_4 = S_3 + S_0 = 5 \quad \hat{\phi}^0 = 1$

$S_{0+4} > \hat{\phi}^0$

② when $n=3$

$S_{3+4} = S_6 + S_3 = S_5 + S_2 + S_3 = S_4 + S_1 + S_2 + S_3$

$= S_3 + S_0 + S_1 + S_2 + S_3$

$= 14$

$\hat{\phi}^3 \in (1^3, 2^3) \quad \hat{\phi}^3 < 8$

$\therefore S_{3+4} \geq \hat{\phi}^3$ is true

③ when $n=1$

$S_{1+4} = S_5 = S_4 + S_1 = S_3 + S_0 + S_1 = 7$

$\hat{\Phi}^1 \in (1, 2)$

$\therefore S_{1+4} \geq \hat{\Phi}^1$ is true

④ when $n=2$

$S_{2+4} = S_6 = S_5 + S_2 = 8$ $\quad \hat{\Phi}^2 \in (1, 4)$

$\therefore S_{2+4} \geq \hat{\Phi}^2$ is true

Assume: it's true for k, $k-3$

$S_{k+4} \geq \hat{\Phi}^k \quad S_{k-3+4} \geq \hat{\Phi}^{k-3}$

Induction: when $n=k+1$

$$S_{k+1+4} = S_{k+5} = S_{k+4} + S_{k+1}$$
$$\geq \hat{\Phi}^k + \hat{\Phi}^{k-3}$$

$\because \hat{\Phi}^k + \hat{\Phi}^{k-3} = (\hat{\Phi}^3 + 1) \cdot \hat{\Phi}^{k-3}$

$\because \hat{\Phi}$ is a root of $x^4 - x^3 - 1 = 0$

$\therefore \hat{\Phi}^4 - \hat{\Phi}^3 - 1 = 0$

$\therefore \hat{\Phi}^3 + 1 = \hat{\Phi}^4$

$\therefore \hat{\Phi}^k + \hat{\Phi}^{k-3} = \hat{\Phi}^4 \cdot \hat{\Phi}^{k-3} = \hat{\Phi}^{k+1}$

$\therefore S_{k+1+4} \geq \hat{\Phi}^k + \hat{\Phi}^{k-3} = \hat{\Phi}^{k+1}$

$\therefore S_{n+4} \geq \hat{\Phi}^n$ for all $n \geq 0$

2. Mistake is in the induction step.

We can assume $n=1$, so the group of 2 horses consists of horse 1 and horse 2.

If we discard horse 2, it's true that horse 1 has same color.

If we discard horse 1, it's also true that horse 2 has same color, which is always true as there is only one horse.

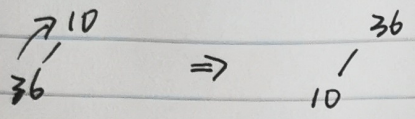
However, in this case, we can not prove that horse 1 and horse 2 have same color. The reasoning process in "Induction Step" is not applicable for this situation. Because there is no such horse that can serve as the "intermediary" to prove the color of the divided sub-groups.

Q3 [10, 36, 25, 54, 37, 12, 75, 68, 42, 86, 72, 90]

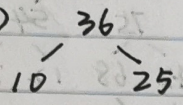
(1) 10

Process:

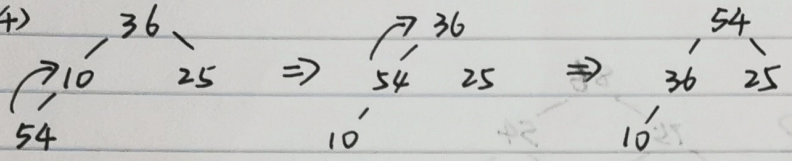
(2)



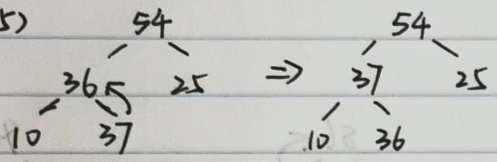
(3)



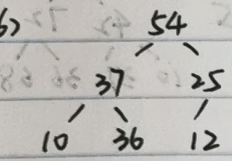
(4)



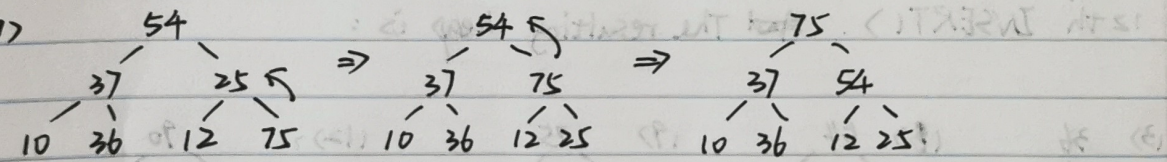
(5)



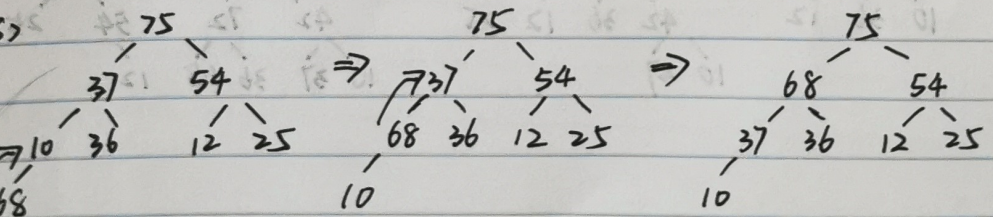
(6)



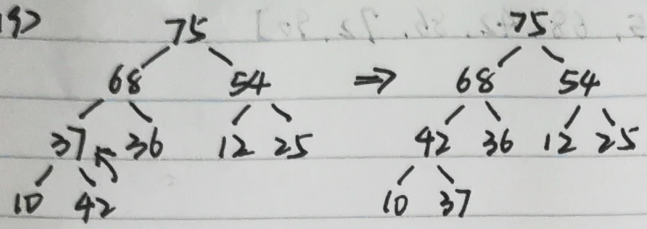
(7)



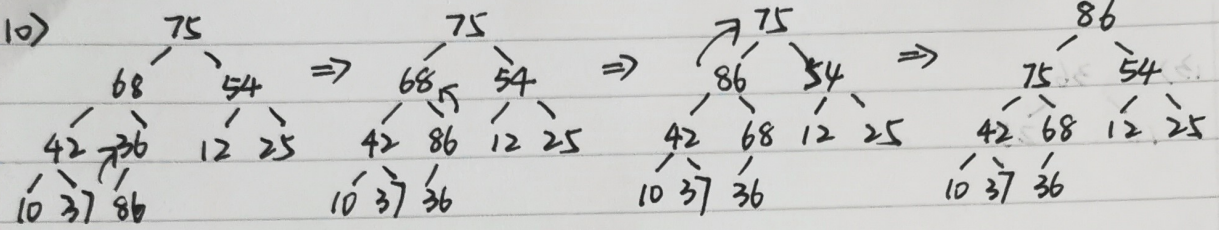
(8)



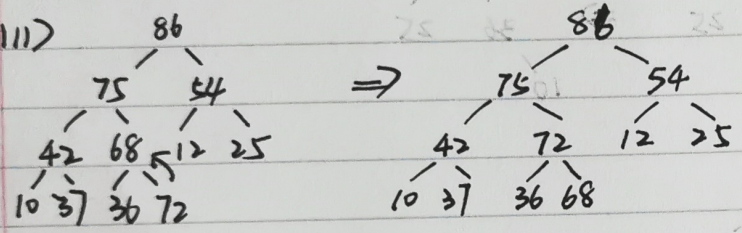
△ (9)



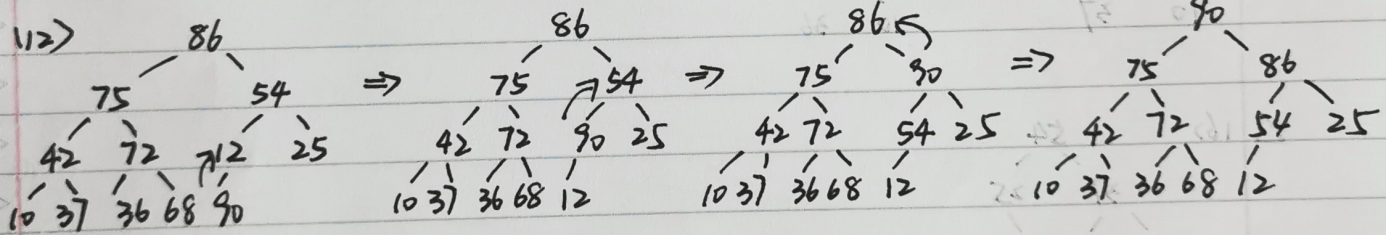
(10)



(11)

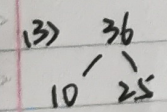


△ (12)

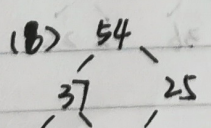


The process has been shown above, and I indicate "△" on the 3rd, 6th, 9th a
12th INSERT( ). ~~And~~ The resulting heap is:

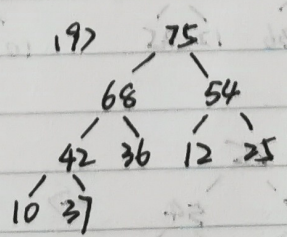
(3)



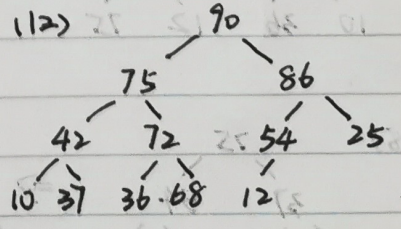
(6)



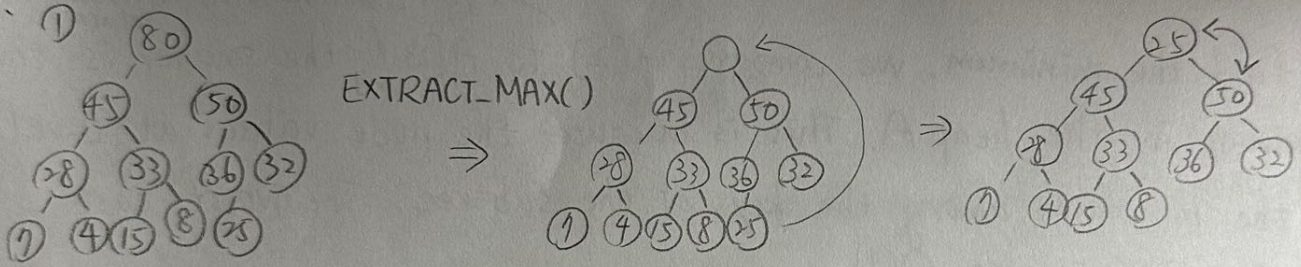
(9)



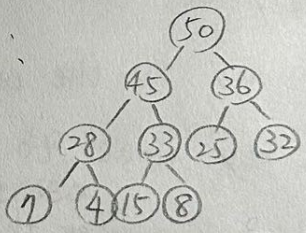
(12)



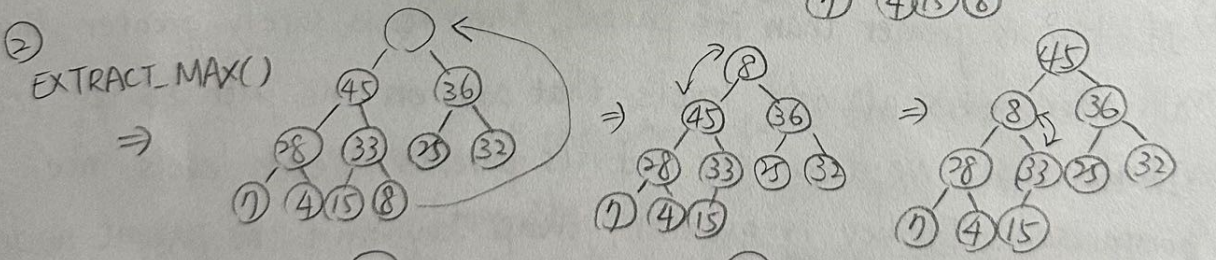
4. ① EXTRACT_MAX()

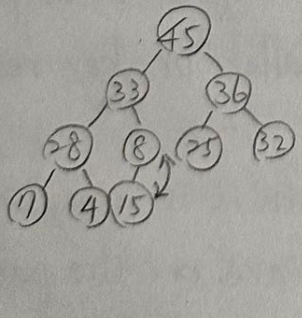


Extract 80 and get max-heap

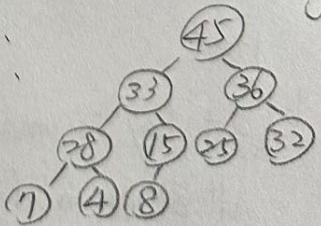


② EXTRACT_MAX()



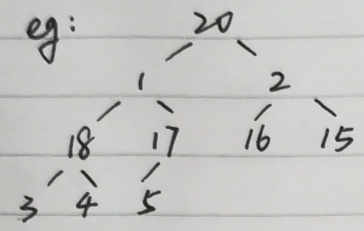


Extract 50 and get max-heap



Q5

(a) eg:



(b) If we want to find the maximum in the heap A, we can just return A[1], which is the largest one. The worse-case time is O(1).
The pseudocode is as follows:

```
HEAP_MAXIMUM (A)
    if A.heap-size < 1
        error "heap underflow"
    return A[1]
```

If we want to find the minimum in the heap A, we can return the smaller one between A[2] and A[3]. The worse-case time is O(1).
The pseudocode is as follows:

```
HEAP_MINIMUM (A)
    if A.heap-size < 1
        error "heap underflow"
    min = A[1]
    if A.heap-size ≥ 2  and  A[2] ≤ min
        min = A[2]
    if A.heap-size ≥ 3  and  A[3] ≤ min
        min = A[3]
    return min.
```

(c) INSERT(A, key)

```
if A.heap_size == n
    error "heap overflow"
A.heap_size = A.heap_size + 1
A[A.heap_size] = key
INCREASE_KEY(A, key)
```

INCREASE_KEY(A, key)

```
find the index i in array A where A[i] == key
level = log2(i)
while level > 0
    if level % 2 == 0                                   // even level
        if A[i] > A[PARENT(PARENT(i))]
            exchange A[i] with A[PARENT(PARENT(i))]
            i = PARENT(PARENT(i))
            level = level - 2
        else if A[i] < A[PARENT(i)]
            exchange A[i] with A[PARENT(i)]
            i = PARENT(i)
            level = level - 1
        else break
    else                                                // odd level
        if level > 1 and A[i] < A[PARENT(PARENT(i))]
            exchange A[i] with A[PARENT(PARENT(i))]
            i = PARENT(PARENT(i))
            level = level - 2
        else if A[i] > A[PARENT(i)]
            exchange A[i] with A[PARENT(i)]
            i = PARENT(i)
            level = level - 1
        else break
```

The running time of INSERT(A, key) is $O(\log n)$.

(d) Heapify (A, i):

```
if i is at even level:
    Let A[largest] = the largest node in either left or right subtree.
    if A[largest] is a grandchild of A[i]:
        if A[largest] > A[i]:
            Swap A[largest] and A[i].
            if new A[largest] < A[parent of largest]:
                Swap A[largest] and A[parent of largest].
            endif
            Heapify (A, largest)
        endif
    else if A[largest] > A[i]: # A[largest] is a child of A[i]
        Swap A[largest] and A[i].
    endif
endif
if i is at odd level:
    ... # The algorithm is identical to the above, but with all of the
          comparison operators reversed.
endif
```

The running time of the worst-case for Heapify (A, i) would be $O(\log n)$ because running from the root till the leaf nodes level would be the worst case, with $\log n$ being the height of the heap A.

(e)

BUILD_HEAP(A, n)

A.heap_size = n

for i = $\lfloor \frac{n}{2} \rfloor$ downto 1

HEAPIFY(A, i)

Analysis of BUILD_HEAP(A, n):

In level 0, there is 1 node, whose height is $\log n$

In level 1, there are 2 nodes, whose height is $\log n - 1$

In the last level, there are at most $\frac{n}{2}$ nodes, whose height is 1.

So the total time $\leq 1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{2^2} + \cdots = \sum_{i=1}^{\log n} i \cdot \frac{n}{2^i} = S$

$S = 1 \cdot \frac{n}{2^1} + 2 \cdot \frac{n}{2^2} + 3 \cdot \frac{n}{2^3} + \cdots$

$\frac{1}{2}S = \quad 1 \cdot \frac{n}{2^2} + 2 \cdot \frac{n}{2^3} + \cdots$

$\frac{1}{2}S = S - \frac{1}{2}S = \frac{n}{2^1} + \frac{n}{2^2} + \frac{n}{2^3} = n(\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots) \leq n$

$\Rightarrow S = 2n = O(n)$

As a result, the running time of BUILD_HEAP() is $O(n)$.

(f) EXTRACT_MAX(A)

max = HEAP_MAXIMUM(A)

A[1] = A[A.heap_size]

A.heap_size = A.heap_size − 1

HEAPIFY(A, 1)

return max

EXTRACT_MIN(A)

min = HEAP_MINIMUM(A)

find the index i in array A where A[i] == min

A[min] = A[A.heap-size]

A.heap_size = A.heap_size − 1

HEAPIFY(A, i)

return min

The running time of EXTRACT_MAX(A) and EXTRACT_MIN(A) is $O(\log n)$.

6. We assume input array is A. the value k is what we need to compare x to the k-th largest element, and i is the index of array.
We'll call the function on the loop.
The pseudo code is as follows:

```
compare (array A, int i, int k, int x, int &count)
    if i > A.heap_size
        return
    if A[index] ≥ x
        return
    count = count + 1
    if count ≥ k
        return .
    compare (A, LEFT(i), k, x, count);
    compare (A, RIGHT(i), k, x, count);
    return;

main_function (array A, int k, int x)
    count = 0
    compare (A, 0, k, x, count)
    if count ≥ k
        return true
    else
        return false
```

We call the function on the loop until count is larger or equal to k, or A[i] or A[index] ≥ x.
So the running time is at most O(k), and the extra space is at most O(k).