CSCE-629

Assignment #1

Fall 2017

Pentyala, Shiva Kumar – UIN : 127003995

9-26-2017

1. Answer the following questions, and give a brief explanation for each of your answers.

a) Quicksort takes time O (n log n)?

Answer: **False**

b) Quicksort takes time O (n^2)?

Answer: **True**

Explanation for (a), (b): Let us consider an array A of size n with start and end as its first and last element indexes.

Worst Case Analysis:

QuickSort (A, start, end): **//T(n)**

1) Check start<end if not return. **//O(1)**
2) pivot = Partition (A, start, end) **//O(n)**
3) QuickSort (A, start, pivot-1) **//T(n-1)**
4) QuickSort (A, pivot+1, end) **//T(0)**

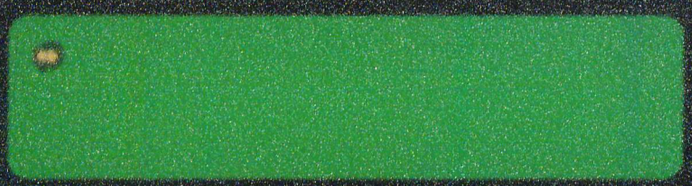
Partition (A, start, end): **//O(n)**

1) p <- choose pivot element **//O(1)**
2) j <- start-1 **//O(1)**
3) For i = start To end: **//O(n)**
   IF A[i] <= p: Swap A[i] and A[j], j++
4) Swap p with A[j+1] , return j+1 **//O(1)**

The worst case occurs when the partition process (Step 2) always picks greatest or smallest element as pivot. Suppose we pick last element as pivot always, the worst case would occur when the array is already sorted in increasing or decreasing order. So we observe the following in worst case:

- One subarray is always empty.
- The second subarray contains n – 1 elements. (all the elements other than the pivot)
- Quicksort is recursively called only on this second part.
- Hence we need at least n – 1 comparisons at the time of partitioning. At each next step, number of comparisons is one less. so on combining n + (n-1) + (n-2) +.....+ 1 = O (n^2). We can also analyze this as two recursive calls on arrays of size 0 and n-1. So

  T (n) = T(n–1) + T (0) + (n-1) = T (n-1) + O (n) [since T (0)=O (1) and n-1=O (n)]
- Solving this recurrence relation, we get T (n) = n (n-1)/2, which is of the order O (n^2).



c) Mergesort takes time O (n log n)?

Answer: **True**

d) Mergesort takes time O (n^2)?

Answer: **False**

Explanation for (c), (d): Let us consider a list L of size n.

```
MergeSort (LIST L): //T(n)

1) Check LEN( L ) >1 if not return L                    // O(1)
2) (List left, List right) <- DIVIDE ( L )              // O(n)
3) left  <- MergeSort (left)                            // T(n/2)
4) right <- MergeSort (right)                           // T(n/2)
5) Return Merge (left, right)   //Merge step            // O(n)
```

```
Merge (L, R): // O(n)

1) F = empty array of size n = LEN (L) + LEN (R)        //O(1)
2) Set i =j = k = 0                                     //O(1)
3) While i < LEN (L) and j < LEN (R):                   //O(n)
        IF L[ i ] <=R[ j ] : set F[ k ] = L[ i ] , i++ , k++
        ELSE :  set F[ k ] = R[ j ] , j++ ,k++
4) IF ( i == LEN (left) ): append ( F , rest (left))   //O(n)
5) ELSE: append( F , rest (right))                      //O(n)
6) Return F                                             //O(1)
```

The worst case scenario for Merge Sort is when, during every Merge step, exactly one value remains in the opposing array; in other words, no comparisons were skipped. This situation occurs when the two largest values in a Merge step are contained in opposing arrays. When this situation occurs, Merge Sort must continue comparing array elements in each of the opposing arrays until the two largest values are compared which would be of the order O (n). So we observe the following in worst case:

- Number of comparisons in merge step that we mentioned above are of the order O (n).
- We are making two recursive calls in step 3 and 4 of MergeSort (A), each on arrays of length n/2, so total time would be 2 T (n/2).
- Hence we get the recurrence relation as T (n) = 2T (n/2) + O (n)
- Solving this recurrence relation, we get T (n) = (n/2) log n, which is of the order O (n log n).

Q. 2) Solve the recurrence relations

a) $T(1) = O(1)$, and $T(n) = 2T(n/2) + O(n^2)$

Let $O(n^2) \leq cn^2$ for some constant $c > 0$

Now, our equation becomes

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn^2 \rightarrow (1)$$

From (1), on replacing n with $\frac{n}{2}$, $\frac{n}{4}$, $\frac{n}{8}$ we get

$$T\left(\frac{n}{2}\right) \leq 2T\left(\frac{n}{4}\right) + \frac{cn^2}{2^2} \rightarrow (a)$$

$$T\left(\frac{n}{4}\right) \leq 2T\left(\frac{n}{8}\right) + \frac{cn^2}{4^2} \rightarrow (b)$$

$$T\left(\frac{n}{8}\right) \leq 2T\left(\frac{n}{16}\right) + \frac{cn^2}{8^2} \rightarrow (c)$$

Substitute (a) in (1),

$$\Rightarrow T(n) \leq 2\left[2T\left(\frac{n}{4}\right) + \frac{cn^2}{2^2}\right] + cn^2$$

$$\Rightarrow T(n) \leq 2^2 T\left(\frac{n}{2^2}\right) + cn^2\left(1 + \frac{1}{2}\right)$$

Substitute (b) in above equation,

$$\Rightarrow T(n) \leq 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{cn^2}{2^4}\right] + cn^2\left(1 + \frac{1}{2}\right)$$

$$\Rightarrow T(n) \leq 2^3 T\left(\frac{n}{2^3}\right) + cn^2\left(1 + \frac{1}{2} + \frac{1}{2^2}\right)$$

Substitute (c) in above equation,

$$\Rightarrow T(n) \leq 2^3\left[2T\left(\frac{n}{2^4}\right) + \frac{cn^2}{2^6}\right] + cn^2\left[1 + \frac{1}{2} + \frac{1}{2^2}\right]$$

$$\Rightarrow T(n) \leq 2^4 T\left(\frac{n}{2^4}\right) + \frac{cn^2}{2^3} + cn^2\left[1 + \frac{1}{2} + \frac{1}{2^2}\right]$$

$$\Rightarrow T(n) \leq 2^4 T\left(\frac{n}{2^4}\right) + cn^2\left[1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3}\right]$$

Recursion depth: If we observe how long it takes until the subproblem has constant size

$$\Rightarrow \frac{n}{2^{4k}} \rightarrow 1$$

k^{th} term will be,

$$T(n) \leq 2^k T\left(\frac{n}{2^k}\right) + cn^2\left[1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right]$$

Idea is to make $\left(\frac{n}{2}\right)$ term smaller so that

$$\frac{n}{2^k} \rightarrow 1 \quad \text{or} \quad T\left(\frac{n}{2^k}\right) \rightarrow T(1)$$

So, $k = \log_2^n \rightarrow (2)$

On applying formula of sum of Geometric progression,

$$1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} = \frac{1\left[\left(\frac{1}{2}\right)^k - 1\right]}{\frac{1}{2} - 1} = 2\left(1 - \left(\frac{1}{2}\right)^k\right)$$

So, $T(n) \leq 2^k T\left(\frac{n}{2^k}\right) + 2cn^2\left(1 - \frac{1}{2^k}\right)$

Substituting (2) in the above equation we get,

$$T(n) \leq nT(1) + 2cn^2\left(1 - \frac{1}{n}\right)$$

Formulas used: $a^{\log_a b} = b$

$$\Rightarrow T(n) \leq nT(1) + 2cn(n-1)$$

Given $T(1) = O(1) \leq c_1 \cdot 1$ where c_1 is some constant

$$\Rightarrow T(n) \leq c_1 n + 2cn^2 - 2cn$$

$$\Rightarrow T(n) \leq 2cn^2 + n(c_1 - 2c) \leq c_3 n^2 \text{ for some constant } c_3$$

Hence $T(n) = O(n^2)$

Solution: $\boxed{T(n) = O(n^2)}$

Lets check our solution with master theorem: [I want to do this for to be on safer side, if I might had did some calculation mistake in Back Substitution method]

Cross checking my solution with Master theorem:

$T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$ is of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

where $a = 2$, $b = 2$, $\log_b^a = \log_2^2 = 1$

our $f(n) = c_1 n^2$ since $O(n^2) \leq c_1 n^2$ for some constant $c_1 > 0$

Comparing $n^{\log_2^2} = n^1$ with $f(n) = c_1 n^2$

$f(n) = \Omega(n^{\log_2^2 + \epsilon}) \Rightarrow$ case 3 (But we need to verify regularity condition)

Regularity condition check:

$af\left(\frac{n}{b}\right) \leq cf(n) \leq c \cdot c_1 \cdot n^2 \leq c_2 n^2$ where c_2 is other const.

$\Rightarrow \frac{2n^2}{4} \leq c_2 n^2 \Rightarrow \frac{1}{2} \leq c_2 < 1$

As for some constant $c_2 < 1$, $af\left(\frac{n}{b}\right) \leq cf(n)$

Hence by Case 3 of master Theorem we can say that,

$$T(n) = O(f(n)) = O(n^2)$$

$$\boxed{T(n) = O(n^2)}$$

b) $T(1) = O(1)$, and $T(n) = 2T(n-2) + O(n)$

Let $O(n) \leq cn$ for some constant $c > 0$

Now, our equation becomes

$$T(n) \leq 2T(n-2) + cn \rightarrow (1)$$

From (1), on replacing n with $(n-2)$, $(n-4)$, $(n-6)$ we get

$T(n-2) \leq 2T(n-4) + c(n-2) \rightarrow (a)$

$T(n-4) \leq 2T(n-6) + c(n-4) \rightarrow (b)$

$T(n-6) \leq 2T(n-8) + c(n-6) \rightarrow (c)$

Substitute (a) in (1),

$\Rightarrow T(n) \leq 2\left[2T(n-4) + c(n-2)\right] + cn$

$\Rightarrow T(n) \leq 2^2 T(n-4) + cn(1+2) - (2^2)$

$$\Rightarrow T(n) \leq 2^2 T(n-4) + cn(1+2) - (2^2)$$

Substituting (b) in above equation,

$$\Rightarrow T(n) \leq 2^2\left[2T(n-6) + c(n-4)\right] + cn(1+2) - (2^2)$$

$$\Rightarrow T(n) \leq 2^3 T(n-6) + cn(1+2+2^2) - (2^2+2^4)$$

Substitute (c) in above equation,

$$\Rightarrow T(n) \leq 2^3\left[2T(n-8) + c(n-6)\right] + cn\left[1+2+2^2\right] - \left[2^2+2^4\right]$$

$$\Rightarrow T(n) \leq 2^4 T(n-8) + cn\left[1+2+2^2+2^3\right] - \left[1\cdot 2^2 + 2\cdot 2^3 + 3\cdot 2^4\right]$$

k^{th} term will be,

$$T(n) \leq 2^K T(n-2K) + cn\left[1+2+2^2+2^3+\cdots+2^{K-1}\right] - \left[1\cdot 2^2 + 2\cdot 2^3 + 3\cdot 2^4 + \cdots (K-1)2^K\right]$$

on applying formula for sum of Geometric progression,

$$1+2+2^2+2^3+\cdots+2^{K-1} = \frac{1\,(2^K-1)}{2-1} = 2^K - 1 \rightarrow (d)$$

On applying formula of sum of Arithmetic-Geometric progression (AGP),

$a, (a+d)r, (a+2d)r^2, (a+3d)r^3, \cdots [a+(n-1)d]\cdot r^{n-1}$ → AGP series

Sum is given as $S_n = \frac{a-[a+(n-1)d]r^n}{1-r} + \frac{dr(1-r^{n-1})}{(1-r)^2} \rightarrow (A)$

we can write $1\cdot 2^2 + 2\cdot 2^3 + 3\cdot 2^4 + 4\cdot 2^5 + \cdots + (K-1)2^K$ as

$$2\underbrace{\left[1\cdot 2 + 2\cdot 2^2 + 3\cdot 2^3 + 4\cdot 2^4 + \cdots + (K-1)2^{K-1}\right]}_{\text{AGP}} = 2^2\underbrace{\left[1 + 2\cdot 2^1 + 3\cdot 2^2 + \cdots + (K-1)2^{K-2}\right]}_{\text{AGP}}$$

It's in AGP type with $a = 1$, $d = 1$, $r = 2$, $n = k-1$

so, sum becomes according to formula (A),

$$2^2 \left[1 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 + \cdots (1 + (k-2) \cdot 1)\, 2^{k-2}\right] =$$

$$2^2 \left[\left(\frac{1 - [1 + (k-2) \cdot 1] \cdot 2^{k-1}}{1-2}\right) + \frac{1 \cdot 2 \cdot (1 - 2^{k-2})}{(1-2)^2}\right]$$

$$= 2^2 \left[(k-1) \cdot 2^{k-1} - 1 + \frac{2 - 2^{k-1}}{1}\right]$$

$$= 2^2 \left[(k-1) 2^{k-1} - 1 + 2 - 2^{k-1}\right]$$

$$= 2^2 \left[(k-1) 2^{k-1} - 2^{k-1} + 1\right]$$

$$= (k-1) \cdot 2^{k+1} - 2^{k+1} + 4$$

$$= k \cdot 2^{k+1} - 2 \cdot 2^{k+1} + 4$$

$$= (k-2)\, 2^{k+1} + 4$$

~~so, [1+2~~

so, $\left[1 \cdot 2^2 + 2 \cdot 2^3 + 3 \cdot 2^4 + \cdots + (k-1) 2^k\right] = (k-2)\, 2^{k+1} + 4 \longrightarrow$ (e)

on substituting (d), (e) in k^{th} term expression we get,

$$\Rightarrow T(n) \leq 2^K T(n-2K) + cn\left[2^K - 1\right] - \left[(K-2)\,2^{K+1} + 4\right]$$

Recursion depth: If we observe how long it takes until the subproblem has constant size.

$$\Rightarrow n-2K \longrightarrow 1$$

Idea is to make $(n-2K)$ term smaller so that

$$(n-2K) \rightarrow 1 \quad \text{or} \quad T(n-2K) \rightarrow T(1)$$

So, $n-2K=1$

$$\Rightarrow K = \frac{n-1}{2} \longrightarrow (f)$$

Substituting (f) in our above expression,

$$\Rightarrow T(n) \leq 2^{\left(\frac{n-1}{2}\right)} T(1) + cn\left[2^{\left(\frac{n-1}{2}\right)} - 1\right] - \frac{(n-5)}{2} \cdot 2^{\left(\frac{n+1}{2}\right)} - 4$$

Given $T(1) = O(1) \leq c \cdot 1$

$$\Rightarrow T(n) \leq c \cdot 2^{\left(\frac{n-1}{2}\right)} + cn \cdot 2^{\frac{n-1}{2}} - cn - (n-5)\,2^{\frac{n-1}{2}} - 4$$

$$\Rightarrow T(n) \leq 2^{\left(\frac{n-1}{2}\right)} \left[c + cn - (n-5)\right] - cn - 4$$

$$\Rightarrow T(n) \leq 2^{\left(\frac{n-1}{2}\right)} \left[c + n(c-1) + 5\right] - cn - 4$$

$$\Rightarrow T(n) \leq c_2 \cdot n \cdot 2^{\left(\frac{n-1}{2}\right)} - c_3 n \text{ , For some constants } c_2, c_3$$

$\Rightarrow T(n) \leq c_4 \cdot n \cdot 2^{\left(\frac{n-1}{2}\right)}$, For some constant c_4

$\Rightarrow T(n) \leq c_4 \left[n \cdot 2^{\left(\frac{n-1}{2}\right)}\right] \leq c_4 \left[n \cdot 2^{\frac{n}{2}}\right]$

Hence $T(n) = O\left(n \cdot 2^{n/2}\right)$

Solution: $\boxed{T(n) = O\left(n \cdot 2^{\frac{n}{2}}\right)}$

cross checking my solution with Muster Theorem or in general Akra-Bazzi Theorem:

It states that, if $T(n)$ is having the property,

$$T(n) \leq \begin{cases} c & \text{if } n \leq 1 \\ aT(n-b) + f(n), & n > 1 \end{cases}$$

for some constants $c, a > 0, b > 0, d \geq 0$ and $f(n)$ is in $O(n^d)$, then

$$T(n) = \begin{cases} O(n^d), & \text{if } a < 1 \\ O(n^{d+1}) & \text{if } a = 1 \\ O(n^d a^{n/b}) & \text{if } a > 1 \end{cases}$$

In our problem, $a = 2$, $b = 2$, $f(n) = O(n)$ so $d = 1$.

This comes under case 3. so our $T(n)$ is of the order $O(n \cdot 2^{n/2})$

Hence $\boxed{T(n) = O\left(n 2^{\frac{n}{2}}\right)}$

Q.3) Use Mathematical Induction to prove that a 2-3 tree with n leaves has its height bounded by $\log_2 n$. The height of a 2-3 tree is defined to be the length of the path from its root to any of its leaves.

Statement: Height of a 2-3 Tree with n leaves is bounded by $\log_2 n$.

$\Rightarrow$ Number of leaves in 2-3 Tree $= n$

Basis: To see our statement holds for $n=1$.

For $n=1$, 2-3 Tree is a single Node Tree. So, From our statement it should have height $= \log_2 1 = 0$

$\Rightarrow$ Statement is True for $n=1$

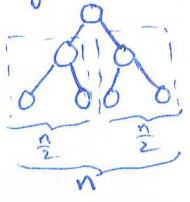
Inductive Step: For $n>1$, According to the definition of 2-3 Tree, if T is a 2-3 tree with n leaves the T should have atleast two children T_{c_1} and T_{c_2}.

T_{c_1}, T_{c_2} are subtrees of T once again so they are also 2-3 Trees.

Since our T has n leaves, T_{c_1} and T_{c_2} or at least one of T_{c_1} and T_{c_2} should not have more than $\frac{n}{2}$ leaves. This upper bound of $\frac{n}{2}$ case may occur if T is similar to a complete binary Tree.

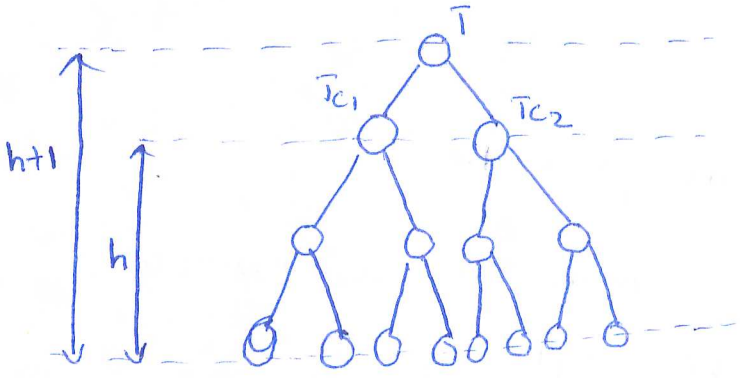
Lets say T_{c_1} has $\frac{n}{2}$ leaves. so According to our statement, height of T_{c_1} is bounded by $\log_2(n/2) = \log_2 n - 1$

$\Rightarrow$ Height of $T_{c_1} \leq \log_2 n - 1 \rightarrow (1)$



According to the definition of 2-3 Tree, All the leaves must be on the same level, which means that all paths from the root of the Tree to its leaves should have same length. In that case

height of T = height of T_{c_1} + 1



So, height of $T \leq \log_2^n - 1 + 1$ [Since From (1)]

$\Rightarrow T \leq \log_2^n$ → Our statement is valid.

Hence by mathematical Induction, we can say that Height of a 2-3 Tree with n leaves has its height bounded by $\log_2^n$.

Q-4) Let T_1 and T_2 be two 2-3 Trees, where T_1 has n elements in its leaves, and T_2 has n/logn elements in its leaves. Develop a linear-time algorithm that places all the elements in T_1 and T_2 in a single sorted list.

Problem Statement:

As T_2 is having less elements than T_1, I will try to remove elements from T_2 and insert them in T_1. Our final goal is to place all the elements of new tree T_1 in to a single sorted List L.

Algorithm Design:

Merge(T_1, T_2):

→ while T_2 != NULL do

① x = Min(T_2)

② Delete(T_2, x)

③ Insert(T_1, x)

→ return T_1

Delete(r, x): // Deletes x from r.

1. If (r == NULL) return
2. If r → leaf:
   If x == l(r) return NULL
   Else return
3. DEL (r, x, done, lson)
4. If (done == false) return
5. If (lson == true) return child1(r)
   Else return r

DEL (r, x, done, lson)

1. done = true
2. If r is parent of leaves
   delete x if it is in tree, update done, lson.
3. If (x <= l(r)) r' = child1(r)
   else if (x <= m(r)) or (child3(r) == null) r' = child2(r)
   Else r' = child3(r)
4. DEL (r', x, done', lson')

5. If (done' == false) done = false; return

6. If (ison' == true)

If r has at least 4 grandchildren:

reorganize so that each of r, children has either 2 or 3 childs

Else make r a 1-child node;

ison = true; return.

Insert (r, x): // r is the root // Inserts x in to r.

1. If less than 2 children return that child,

2. Addson (r, x, r')

3. If r' != NULL

create a new node v, (r, r' be children of v);

4. return v

Addson (r, x, r'): // r is not a leaf

1. r' = null;

2. If r is a parent of leaves

If r has 2 childs add x as a new child of r

Else // r has 3 children

order x and the 3 childs of r in linear order

return;

3. If (l(r) >= x) v = child1(r)

Else if ((m(r)) >= x or child3(r) == NULL) v = child2(r);

Else v = child3(r)

4. Addson (v, x, v')

5. If v' == null return

6. If v' != null, r has 2 child add v' as new child of r else process accordingly

Min(r): // gives minimum value in tree r.

① If r is Empty return;

② If r is leaf return value of r

③ Else return min(child1(r))

After performing Merge(T_1, T_2), T_1 will have $n+\frac{n}{\log n}$ elements and T_2 will have null. In order to copy final elements of T_1 in to a list L, we need to go till bottom of the Tree T_1. As Insert() Algorithm will Place all the elements in sorted order, Final Tree T_1 will have all of its leaves in sorted order. So we just need to copy and place one by one in the List L.

Main(T_1, T_2): // O(n) (shown in next page)

① Merge(T_1, T_2) // O(n)

② L = [to store $n+\frac{n}{\log n}$ elements], i=0 // O(1)

③ copy(T_1) // O(n)

copy(T): // copies leaves of T in to L recursively

① If T=null return;

② If T is a leaf:
i = i+1
A[i] = value(T)

③ copy(T_{c_1}) // child 1 of T

④ copy(T_{c_2}) // child 2 of T

⑤ If $T_{c_3} \neq$ null then copy(T_{c_3}) // child 3 of T

As our required elements are stored in the leaf level of T and are sorted from left to right, copy(T) just places them in to a List L. So thus L will have $n+\frac{n}{\log n}$ elements in sorted order.

Analysis of Algorithm: Main() has two big operations. 1) merge 2) copy.

① → Inorder to place one element in T_1 from T_2, Merge algorithm does one Min(), one Delete() operations on T_2 and one Insert() operation on T_1.

We know that, Min(), Delete() take $O(\log n)$ time where n is the number of leaves in T_2. At any time, T_1 will have max $n + \frac{n}{\log n}$ elements. As T_1 has $n + \frac{n}{\log n} \leq 2n$. So inorder to remove the correct element (maximum elements in T_1) from T_2 its taking $O(\log 2n) = O(\log n)$ and Insert take $O(\log n)$ $= O(\log n)$

while loop runs $\frac{n}{\log n}$ times in Merge(). So total comes $O(n)$.

In brief,

Inorder to place one element in T_1

① Min(), Delete() taking $O(\log n)$

② Insert() → $O(\log n)$

so, for $\frac{n}{\log n}$ elements, running Time of Merge() is

$$\frac{n}{\log n} \cdot \left[O(\log n) + O(\log n)\right] = O(n)$$

② In main, next operation Copy() is having 5 steps. each step other than recursive calls take constant time $O(1)$ on each node. As its spending constant time on each node, total Running time of Copy() will be of the order $O(n_{total})$. where n_{total} are the

total number of nodes in the Tree T_i.

Total number of nodes in T_i (n_{total}) = $n_{leafs} + n_{internal}$

we know that, leafs nodes (n_{leafs}) = $n + \frac{n}{\log n} \leq 2n$

internal nodes ($n_{internal}$) $\leq n_{leafs} \leq n + \frac{n}{\log n} \leq 2n$

$\Rightarrow$ Total nodes in T_i (n_{total}) $\leq 4n = O(n)$

$\Rightarrow$ copy( ) has time complexity of $O(n)$

Conclusion:

Overall, our Algorithm is taking $O(n)$ time to solve the problem statement discussed at the start of this question. so It's a linear-time Algorithm.

Q.4) Design Algorithms for Min(S), Insert(S, a), and Delete(S, h), where the set S is stored in a heap, 'a' is the element to be inserted into the heap S, and h is the index of the element in the heap S to be deleted. Analyze the complexity of your algorithms.

Problem statement:

Performing Min(S), Insert(S,a), Delete(S,h) operations on a heap 'S'.

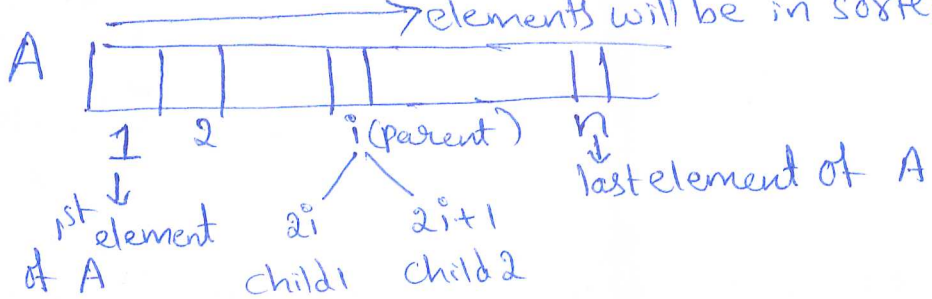
a → element to be inserted

h → Index of the element to be deleted

A → array representation of heap S.

~~Anak~~ Algorithm Design:

Let Heap S has n elements. It can be represented by an array A. If A starts with index 1, then for any index i, we can access the heap element with A[i] and its children using A[2i] and A[2i+1].



similarly, For any element in the heap A[i], its father will be $A[\lfloor i/2 \rfloor]$. so this array representation of heap is very useful in terms of Accessing the elements in heap, their respective children and Parents.

Min(A): // A is the array representing heap
// output: Minimum element in A
// Running time: O(1)

1. return A[1]

Insert(A, a): // Assumed Array entries are Indexed 1 to n.
// Input: an array representing the heap; element to be inserted.
// output: modified 'A' that represents a heap
// Running Time: O(log n)

1. n = n+1
2. A[n] = a
3. Min-Heapify(A, n)

Delete(A, h):

// Input: an array representing a heap; Index of the element to be deleted

// output: modified 'A' that represents a heap

// Running time: O(log n)

1. A[h] = A[n]
2. n = n-1
3. Min-Heapify(A, h)

Min-Heapify(A, x): // To shake till we reach correct state of heap.

// Input: an unsorted array, which can't be called as heap; Index

// of the element in the heap where there is some bug.

// output: modified array 'A' which represents a heap

// Running Time: O(log n)

1. IF $x > 1$ and $A[x] < A[\lfloor x/2 \rfloor]$: // child is lighter / pushing up operation on A[x].

   ↓ To check and make sure its not ROOT because root can't go up.

   $y = x$

   while $y > 1$ and $A[y] < A[\lfloor y/2 \rfloor]$ do

   $A[y] \xleftrightarrow{\text{swap}} A[\lfloor y/2 \rfloor]$

   $y = \lfloor y/2 \rfloor$

2. ELSE IF $x \leq \lfloor n/2 \rfloor$ and $A[x] >$ minimum $\{A[2x]$ and $A[2x+1]\}$ then:

   ↙ To make sure it's not leaf because leaf can't go down.

   $y = x$;

   while $y \leq \lfloor n/2 \rfloor$ and $A[y] > \min(A[2y], A[2y+1])$ do

   check which child is minimum = min $\{A[2y], A[2y+1]\}$

   swap corresponding child with A[y]

   y = Index of corresponding child.

Analysis of Algorithm:

As we know the index in array, we can access any element of the heap in $O(1)$ time.

① Min-Heapify (A, x): It takes the index x of array which is changing the heap property from heap to non-heap. That means $A[x]$ is either too large or small compared to its parent/children. So we run while loop inside this Algorithm inorder to repeat the process till we get back Heap → complete binary tree with possibly some right most leaves removed. ⇒ After min-Heapify, A' is a valid Heap.

For n nodes in the heap, we know its height is $O(\log_2 n)$. In the entire process, we are moving level by level and at each level we are spending constant time $O(1)$ so in the worst case, we may need at most one swap on each level of a heap on the path from the inserted node to the root or child. So worst case time of Min-Heapify is $O(\log_2 n)$ as its height is bounded by $\log_2 n$

② min(A): $O(1)$ as we can access using array index.

③ Insert(A, a): $O(\log_2 n)$ as 1,2 steps take $O(1)$ and 3rd take $O(\log_2 n)$

④ Delete(A, h): $O(\log_2 n)$ as 1,2 steps take $O(1)$ and 3rd take $O(\log_2 n)$

Conclusion: overall, our Algorithm for min(S) take $O(1)$ time and Insert(S, a), Delete(S, h) take $O(\log_2 n)$ time to solve our Problem statement mentioned at the start of the question.