DAA HW 4

Pranav Tushar Pradhan N18401944 pp3051@nyu.edu

Udit Milind Gavasane N16545381 umg215@nyu.edu

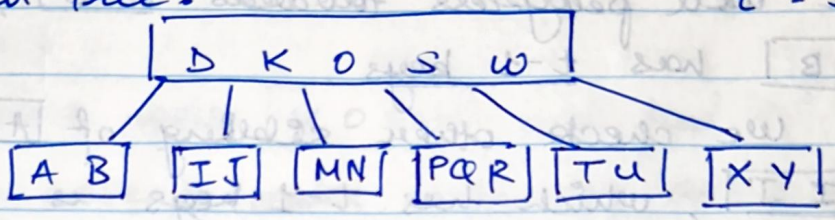
Rohan Mahesh Gore N19332535 rmg9725@nyu.edu

Saavy Singh N16140420 ss19170@nyu.edu

Sashank Badri Narayan N14628839 sb10192@nyu.edu

Q. 1.

- Given Tree: $t = 3$



== ~~Phase 1: Delete~~ B

As we have $t = 3$

∴ For root:

children = 2 ~ 6

keys stored = 1 ~ 5

∴ For children / internal node:

children = 3 ~ 6

keys stored = 2 ~ 5

== Phase 1: Delete B

Step 1: Start from node [D K O S W] and check if it has > 1 key.

∴ 5 > 1 ~~Hence case~~

Also it does not have "B" ∴ we need to move to it children.

∴ [Case 3 is clear for [D K O S W]]

Based on structure of tree we will approach node [A, B] next.

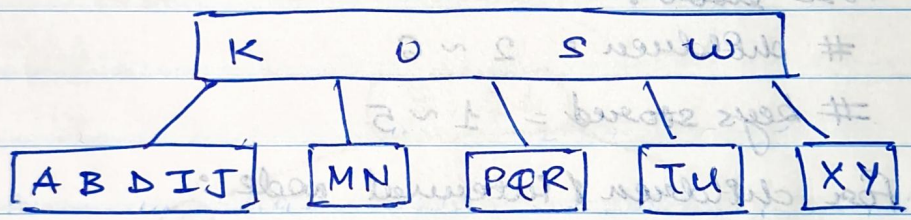
Step 2: As we can see - based on [D] - that we will progress towards [A B], but, [A B] has t-1 keys,

∴ We check other sibling of [A B] which is [I J], which has t-1 keys as well.

- As both children of D have t-1 keys

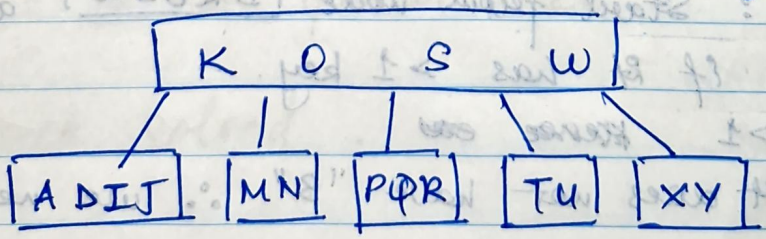
∴ we implement Case 3.b

∴ Tree becomes.



Step 3: We can now visit the node [A B D I J] and can delete B "Case 1".

∴ We have tree after deleting B



Phase 2 : Delete O

Step 1 :

We check node [K O S W] and find "O"

∴ we have to ~~make sure its children~~ replace "O" by predecessor

- But "predecessor" containing node → [M N] has "t-1" keys

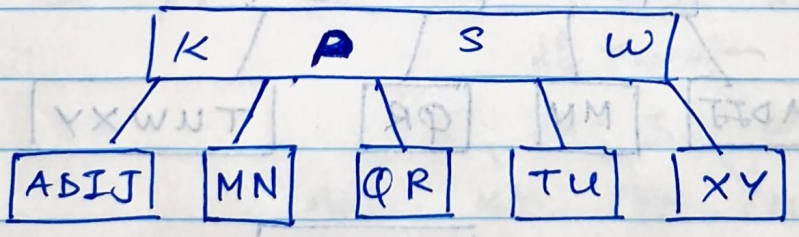
  ∴ Case 2a FAILED

So, we look for successor

- ~~But~~ "successor containing node has t keys

∴ Implement Case 2b.

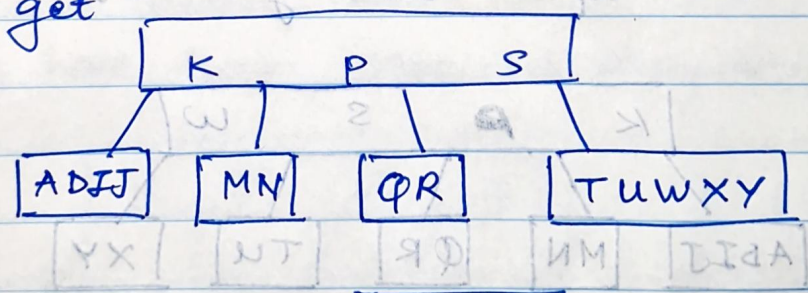
∴ Tree after deleting O is:



Phase 3 : Delete U

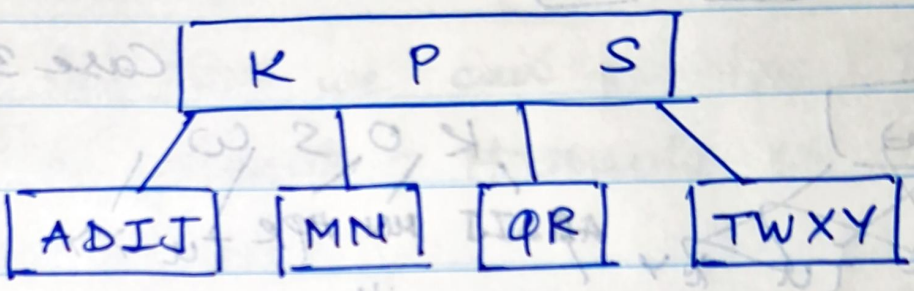
Step 1 : Find the Node [K, P, S, W] ⇒ After visiting key "W" we can confirm that, if "U" is present - it should be in left subtree of W.
Also, has [K, P, S, W] node has >t-1 keys
∴ Case 3 verified

Step 2 : Explore left subtree of W
- The left child has 2 keys i.e $< t = 3$ [T, U]
- we can check right sibling i.e [X, Y]
  - but it also has 2 keys i.e $< t = 3$
∴ we need to IMPLEMENT Case 3b
∴ we get



Step 3 : Visit node [TUWXY]
- we found key "U" and as this node is leaf node,
∴ we can delete "U" by using Case 1.

∴ final tree is



2) a) $T(n) = 4T(n/2) + n^2$

For this recurrence

$a=4,\ b=2$ and $d=2$

Comparing $\frac{a}{b^d}$ with 1

$\therefore \frac{a}{b^d} = \frac{4}{2^2} = 1$

$\therefore$ By using baby master th^m

$T(n) = \Theta(n^d \log n) = \Theta(n^2 \log n)$

b) $T(n) = 2T(7n/10) + n^3$

$a=2,\ b=\frac{10}{7},\ d=3$

$\therefore \frac{a}{b^d} = \frac{2}{(\frac{10}{7})^3} \approx 0.686 < 1$

$\therefore$ By using baby master th^m

$T(n) = \Theta(n^d) = \Theta(n^3)$

c) $T(n) = 6T(n/4) + n\sqrt{n} = 6T(n/4) + n^{3/2}$

$\therefore a=6,\ b=4,\ d=3/2$

$\therefore \frac{a}{b^d} = \frac{6}{(4)^{3/2}} = 0.75 < 1$

$\therefore$ By using baby master th^m

$T(n) = \Theta(n^d) = \Theta(n^{3/2})$

d) $T(n) = 9T(n/2) + n^3$

$\therefore a=9,\ b=2,\ d=3$

$\frac{a}{b^d} = \frac{9}{2^3} = 1.125 > 1$

$\therefore$ By using baby master th^m

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 9}) = \Theta(n^{3.1699})$

3) a) $n \leq 2$ for $T(n)$

a) $T(n) = T(n-3) + n^3$

$\therefore T(n) = T(n-3) + n^3$

$= T(n-6) + (n-3)^3 + n^3$

$= T(n-9) + (n-6)^3 + (n-3)^3 + n^3$

$\vdots$

$= T(n-3k) + (n-3(k-1))^3 + \cdots + (n-3)^3 + n^3$

recursion stops at $n - 3k \leq 2$

$k = \frac{n-2}{3} \simeq k = \frac{n}{3} - \frac{2}{3} \approx \frac{n}{3}$

$\therefore T(n) \doteq \sum_{i=0}^{\frac{n}{3}-1} (n-3i)^3 \approx \boxed{\underline{\Theta(n^4)}}$ ... as it is dominated by n^3

b) $T(n) = 3T(n/3) + n\log^2 n$

$\therefore T(n) = 3T\left(\frac{n}{3}\right) + n\log^2 n$

$= 3\left(3T\left(\frac{n}{9}\right)\right) + \left(\frac{n}{3}\right)\log^2\left(\frac{n}{3}\right) + n\log^2 n$

$= 9\left(3T\left(\frac{n}{27}\right)\right) + \frac{n}{9}\log^2\left(\frac{n}{9}\right) + \left(\frac{n}{3}\right)\log^2\left(\frac{n}{3}\right) + n\log^2 n$

$\vdots$

$= 3^k T\left(\frac{n}{3^k}\right) + \sum_{i=0}^{k-1} \left(n\log^2\left(\frac{n}{3^i}\right)\right)$

recursion stops at $n/3^k = 1$ (as $T(n)$ is constant at $n \leq 2$)

$\therefore k = \log_3 n$

$T(n) = n \cdot O(1) + \sum_{i=0}^{\log_3 n - 1} n\log^2\left(\frac{n}{3^i}\right)$ $\quad \sum (\log n - i\log 3)^2$

↑ insignificant here

$\sum_{i=0}^{\log_3 n - 1} n\left(\log^2(n) - 2i\log n + i^2\log^2 3\right)$

Dominant term here is $n\log^2 n$

$\therefore T(n) = \boxed{\underline{\Theta(n\log^3 n)}}$

4) $T(n) = 2\sqrt{n}\, T(\sqrt{n}) + cn$

n reduces to $\sqrt{n}$. There are going to be $2\sqrt{n}$ subproblems

Structure of tree -

Root → initial problem, size n → cost cn

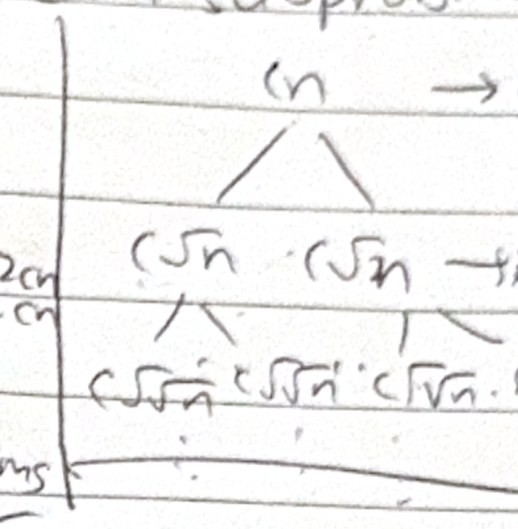
Level 1 → $2\sqrt{n}$ subproblems, size $\sqrt{n}$ → cost $2\sqrt{n} \cdot c\sqrt{n} = 2cn = cn$

Level 2 → For each $\sqrt{n}$ subproblem there are $2\sqrt{\sqrt{n}}$ subproblem of size $\sqrt{\sqrt{n}}$ · No. of subproblems is $2\sqrt{n} \cdot 2\sqrt{\sqrt{n}} = 2^2 n^{1/4}$ → Cost $2^2 n^{1/4} \cdot c\sqrt{n}$

$= 4cn = cn$

Level k → No. of subproblems is $2^k \cdot n^{1/2^k}$ with size $n^{1/2^k}$ → cost $2^k n^{1/2^k} \cdot cn^{1/2^k}$

$= 2^k \cdot c \cdot n = cn$



Depth of the tree is depended as follows -

$n \xrightarrow{\text{reduce}} \sqrt{n} \xrightarrow{\text{reduce}} \sqrt{\sqrt{n}}$ until $n \leq 2$

$\therefore \quad n^{1/2^k} = 2$

$\frac{1}{2^k} \log_2 n = \log_2 2 = 1$

$\log_2 n = 2^k$

$\boxed{\log_2 \log_2 n = k}$

Method 0:

$\therefore$ Total cost $= cn + 2cn + 4cn + \cdots + 2^{\log_2 \log_2 n} cn = cn + 2cn + 4cn + \cdots + 2^{\log_2 \log_2 n} cn$

$= cn \cdot \frac{2^{\log_2 \log_2 n} - 1}{2-1}$ ... Geometric progression

$= cn \cdot (2^{\log_2 \log_2 n} - 1)$

$= cn \cdot (\log_2 n - 1)$

Dominating term $= n \log_2 n$

$\therefore$ Total cost $= \boxed{\Theta(n \log_2 n)}$

Method 1:

$\therefore$ Total cost $= cn * \Theta(\log_2 \log_2 n) = \boxed{\Theta(n \log_2 \log_2 n)}$

Both answers are asymptotically correct. M0 uses GP (product) M1 uses constant cn. ~~as the answer for 2^k as it~~ will

DAA HW-4

5. To determine whether the deterministic quick selection algorithm, modified to divide the input items into groups of 9 instead of 5, still runs in worst-case linear time, we need to analyze the recurrence for the algorithms worst-case running time $T(n)$.

The SELECT algorithm, as described in the textbook, operates by

1. Dividing the input into groups (initially of size 5, now of size 9)
2. Finding the median of each group
3. Recursively selecting the median of the medians
4. Partitioning the array based on this median.
5. Recursively selecting the k-th smallest element in the relevant partition.

Recurrence relation:

1. Divide the items into groups of 9:
   - The number of groups is $\frac{n}{9}$.

2. Find the median of each ~~grou~~ group:
   - Finding the median within each group of 9 takes constant time (Since 9 is a fixed small number, sorting each group can be done in constant time).

   - Therefore, finding the medians of all groups takes $O(n)$ time.

3. Recursively find the median of the medians:
   - Since there are $\frac{n}{9}$ medians, finding the median of the medians involves solving the selection problem recursively on $\frac{n}{9}$ elements.

. The time for this recursive step is $T(\frac{n}{9})$

4. Partition the array based on the median of medians:
. Partitioning
. Partitioning the array around the median takes $O(n)$ time (since partitioning is linear).

5. Recursive selection in one of the partitions:
. In the worst case, the median of medians divide the array such that at least $\frac{1}{9}n$ elements remain in the larger partition (Since the median of medians is a "good" pivot, it guarantees a reasonably balanced partition

. The worst-case size of the remaining subproblem is at most $\frac{7}{9}n$.

Recurrence:

Based on the steps above, the recurrence for the worst-case running time $T(n)$ is:

$$T(n) \leq T(\frac{n}{9}) + T(\frac{7}{9}n) + O(n)$$

Solving the recurrence:

We solve this recurrence using the substitution method or by applying the master theorem

1. The recurrence consists of 2 recursive calls on $\frac{n}{9}$ and $\frac{7n}{9}$, both of which decrease the size of the problem but still involve linear work $O(n)$ at each level.
2. The recursive tree has $O(\log n)$ levels (since at each level the problem size is reduced by a constant fraction)

and the total work done at each level is $O(n)$ due to the partitioning and median-finding steps.

Thus the overall time complexity of the algorithm is

$$T(n) = O(n).$$

6. (a) Sorting algorithm for $n \leq p$:

When $n \leq p$, we can take full advantage of the hardware priority queue, which can store up to p records & allows $O(1)$ worst-case time for both INSERT and EXTRACT-MIN operations.

Algorithm:

1. Insert all n elements into the priority queue:
Since $n \leq p$, we can fit all n elements into the hardware device.

- Inserting each element takes $O(1)$ time, and since there are n elements, this step takes $O(n)$ time.

2. Extract the elements in sorted order: since the priority queue allows $O(1)$ time EXTRACT-MIN operations, extract the minimum element n times in sorted order.

- Extracting all elements takes $O(n)$ time.

Total time complexity:

- The total time for the algorithm is the sum of inserting and extracting operations, which gives: $O(n) + O(n) = O(n)$

Thus the algorithm runs in $O(n)$ worst-case time when $n \leq p$

(b) Sorting algorithm for $n > p$ (divide and conquer approach):

When $n > p$, the number of records exceeds the capacity of the hardware priority queue. We can use a divide and conquer strategy, similar to merge sort, and also make use of the hardware priority queue for efficient merging.

High-level idea:

- Divide the input into smaller subarrays, each of size at most p, sort each subarray using the hardware priority queue, and then merge them using the hardware device.

- The hardware priority queue will be useful during the merging process, enabling fast sorting of the divided parts.

Algorithm:

1. Divide the records into chunks of size at most p:
   - Break the n records into $\lceil \frac{n}{p} \rceil$ subarrays, each of size at most p.
   - Sorting each subarray can be done using the approach in part (a), in $O(p)$ time for each chunk.
   - There are $\lceil \frac{n}{p} \rceil$ chunks, so the total time for sorting all the chunks is $O(\frac{n}{p} \times p) = O(n)$.

2. Merge the sorted subarrays:
   - After sorting the subarrays, we need to merge them to get the final sorted order.
   - This is where we use the hardware priority queue. We can treat the priority queue as a min-heap & store the first

element of each subarray in the queue (up to p elements at a time)

- Use the EXTRACT-MIN operation to repeatedly extract using the priority queue for a total of n elements.
- Each merge operation involves O(1) inserts and extracts using the priority queue for a total of n elements.
- There are $\lceil \frac{n}{p} \rceil$ subarrays to merge, and the merging step at each level takes O(n) time using the priority queue.

3. Recursive merging:
- The algorithm operates like a multi-way merge sort, merging $\lceil \frac{n}{p} \rceil$ sorted chunk in each level of recursion. Each level takes O(n) time, and there are $O(\log_p n)$ levels (since at each level, the number of subarrays reduces by a factor of p).

Total time complexity:
- Sorting the individual chunks takes O(n).
- Merging the sorted subarrays takes $O(n \log_p n)$. Thus the overall time complexity is:

$$O(n \log_p n).$$

Result:
- For part (a) when $n \leq p$, the sorting algorithm runs in O(n) time.
- For part (b), when $n > p$, the divide-and-conquer sorting algorithm runs in $O(n \log_p n)$ time.

7. To solve the problem of finding the minimum sum contiguous subsequence using a divide-and-conquer approach, we can design an algorithm inspired by the well-known maximum subarray problem (Kadane's algorithm), but adapted to find the minimum sum subsequence.

Problem Restatement:
Given an unsorted sequence $x_1, x_2, \ldots, x_n$ of real numbers, we are tasked with finding the subsequence (of at least one element) with the minimum sum in $O(n \log n)$ time using a divide-and-conquer approach.

Approach:
We will divide the array into two halves, solve for the minimum sum subsequence in the left half, the right half, and across the middle. Then, we combine the results to find the overall minimum.

Algorithm:
1. Best Case:
- If the array contains only one element, the minimum sum subsequence is simply that element. Thus, for $n = 1$, the result is the value of the single element, i.e., return x_1.

2. Divide Step:
Divide the array into 2 steps:
- Left half : $x_1, x_2, \ldots, x_{mid}$
- Right half : $x_{mid+1}, x_{mid+2}, \ldots, x_n$

3. Conquer Step:
- Recursively find the minimum sum subsequence in the left half and the right half.

· Find the minimum sub sequence crossing the middle. This is the key part, and we can compute it efficiently in linear time.

4. Combine step:
· The overall minimum sum subsequence will be the minimum of
1) The minimum sum subsequence entirely in the left half.
2) The minimum sum subsequence entirely in the right half.
3) The minimum sum subsequence that crosses the middle.

Finding the minimum sum crossing the middle:
To find the minimum sum subsequence crossing the middle we need to consider 2 parts:
· Left part: The subsequence that ends at the middle.
. Start at the middle and move leftwards, keeping track of the running sum and recording the minimum sum encountered.

· Right part: The subsequence that starts just after the middle.
· Start just after the middle and move rightwards, keeping track of the running sum and recording the minimum sum encountered.

The crossing subsequence is the combination of the left minimum and the right minimum subsequences.

Recurrence relation: Let $T(n)$ be the running time of the algorithm for an input of size n.
· The divide step takes $O(1)$ time.
· The conquer step involves two recursive calls on subarrays of size $n/2$, which gives us $2T(n/2)$.

• The combine step involves finding the minimum sum crossing the middle, which takes $O(n)$ time.

Thus the recurrence is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

This is a standard divide-and-conquer recurrence, and by the Master Theorem, it solves to:

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

Algorithm Summary:

1. Best case: If the subarray has one element, return the element.
2. Divide: Split the array into two halves.
3. Conquer: Recursively find the minimum sum subsequence in each half.

4. Combine: Find the minimum sum subsequence that crosses the middle and combine the results.

5. Return the minimum of the three subsequences (left half, right half, crossing the middle).

Summary:

This divide-and-conquer algorithm correctly finds the subsequence with the minimum sum in $O(n \log n)$ time by splitting the array, solving each half recursively, and combining the results.