This entrance exam booklet is being re-used for internal exams.
Please ignore the printed material on the sheets.

CHENNAI MATHEMATICAL INSTITUTE
https://www.cmi.ac.in

End Semester Examination, Aug–Nov 2023

Name: Arittra Mayumdar	Roll Number: U01910801005
Date: 27th Nov	Subject: Haskell
Course & Year: MCS202304	Total No. of Pages:

100/100

	Points	Remarks
Part A	40	
Part B	60	
Total	100	

B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	Total
20	20	20								
B11	**B12**	**B13**	**B14**	**B15**	**B16**	**B17**	**B18**	**B19**	**B20**	**Total**

Part B

For questions in part (B), you have to write your answer with a short explanation in the space provided below. For numerical answers, the following forms are acceptable: fractions, decimals, symbolic e.g.: $\binom{n}{r}$, ${}^nP_r$, $n!$ etc.

---

(1)

~~1> Nothing~~
~~2> Just Nothing~~
~~3> Just True~~
~~4> Just False~~

1> Nothing
2> Just (False, Nothing)
3> Just (False, Just False)
4> Just (False, Just True)
5> Just (True, Nothing)
6> Just (True, Just False)
7> Just (True, Just True)

8

---

(2)

~~(North, False)~~

(North, Nothing)

8

(3) main :: IO string

~~act :: IO Bool~~

act :: Int → IO Bool

8

---

(4)

multiplyAll :: BinTree → Int

multiplyAll Nil = 1

multiplyAll (Node left curr right) =

multiplyAll (left) * curr
* (multiplyAll right)

8

(5) f xs = foldr (\x l → x:[sum l]) [] xs

f [1,2,3,4] = foldr (\x l → x:[sum l]) [] [1,2,3,4]

= foldr (..) (\4 [] → 4:[0]) [3,..]

= foldr (..) [4,0] [1,2,3]

= foldr (..) (\3 [4,0] → 3:[sum [4,0]]) [1,2]

= [10,0]

for calculating sum of all items of 2 elements

So, in each iteration / recursion call it spends ●(1) + ●(2) times on it

for appending

and go to the next lower element.

---

(6)

So, $T(n) = T(n-1) + O(1)$

$T(n) \leq T(n-1) + c$

$T(2) \leq T(1) + c$

$T(1) = 1$

$T(n) \leq cn$

$T(n) = O(n)$

8

PTO

Part B

(7)(1) map (+1) (map (^2) [1.. 5])

= map (+1) ((1^2) : map (^2) [2.. 5])

= map (+1) (1 : map (^2) [2.. 5])

= [redacted] 2 : map (+1) (map (^2) [2.. 5])

= 2 : 5 : map (+1) (map (^2) [3.. 5])

= 2 : 5 : 10 map (+1) (map (^2) [4.. 5])

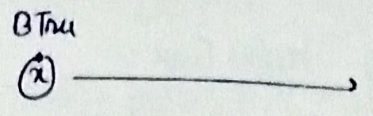
= 2 : 5 : 10 : 17 : 26

20

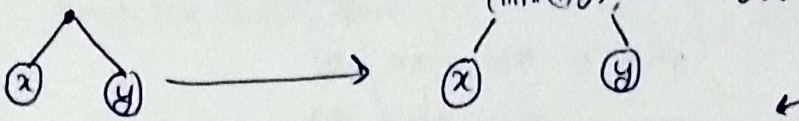
8)

(9) (2) data DBTree = DLeaf Int | DFork DBTree (Int, Int) DBTree

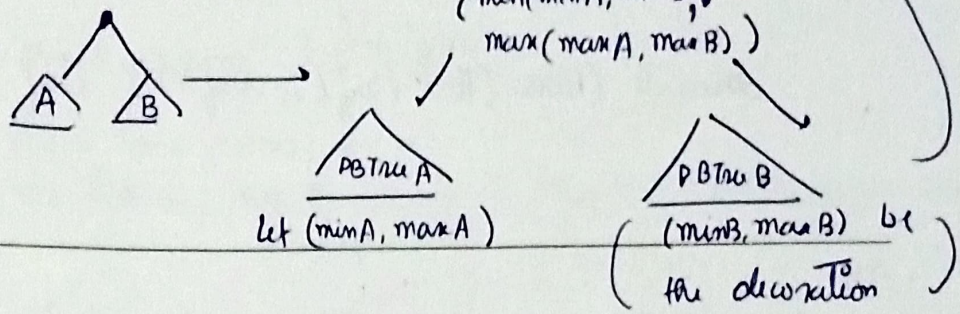
Intuition: 1) we have a BTree instance which is leaf. So, it would be same



2) Now, if we have a fork node with two leaves,



3) if we have a fork node with two fork nodes, (not leaves)



(10)

4) Similarly forks with fork one node and one fork can be handled similarly

Code:

```
decorate :: BTree ⟶ DBTree

decorate (Leaf a) = (DLeaf a)

decorate (Fork (Leaf l) (Leaf r)) =

    (DFork (DLeaf L) (min (l,r), max (l,r)) (DLeaf r))
```


P70

(11) decorate (Fork (Leaf a) (Fork [redacted] (Leaf l) (Leaf r))) =

(DFork (DLeaf a) (mn, mx) right) where

right = decorate rightFork

(_, [redacted] (mn_, mx_), _) = right

mn = min (mn_, a)

mx = max (mx_, a)

decorate (Fork (Fork (Leaf l) (Leaf r)) (Leaf

---

(12) decorate (Fork (Leaf a) (Fork rightfork)) =

(DFork (DLeaf a) (mn, mx) right) where

right = decorate rightFork

(_ (mn_, mx_), _) = right

mn = min (mn_, a)

mx = max (mx_, a)

This will be similar for the case when leaf is in right.

(13)

decorate (Fork (Fork left) (Fork right)) =

(DFork left (mn, mx) right) where

left = decorate leftFork

right = decorate rightFork

~~((mn_l, mx_l), _)~~

(_, (mn_l, mx_l), _) = left

(_, (mn_r, mx_r), _) = right

mn = min (mn_l, mn_r)

mx = max (mx_l, mx_r)

(14)

(17)

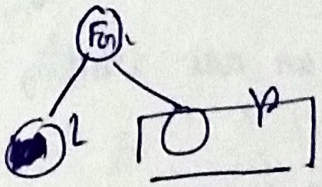
(3) leaves Naive approach : Inorder

```
leaves :: BTree → [Int]
leaves (Leaf a) = [a]
leaves (Fork left right) = (leaves left) ++ (leaves right)
```

But this may take $O(n^2)$ time
where n = number of leaves

leaves optimal approach :-

(18) Intuition : we will keep an array that stores on a Fork, all the right children.



append each element of L to ↑ (we know the answer)

PTO

(19)

```
leaves :: BinTree → [Int]
leaves root = go root []

go :: BinTree → [Int] [Int]
go (Leaf a) am = a : am
go (Fork l r) am = go l (go r am)
```

calculate the am for right

calculate the answers for whole tree

(20)

Only operation we are doing is (a : am) leaf

and we are visiting each leaves once.

So, it will take O(n) time

n = number of leaves

Rough work

Naive

buildBTree :: [Int] → BTree

can be implemented using length (=n) and splitAt (n/2) (n `div` 2)

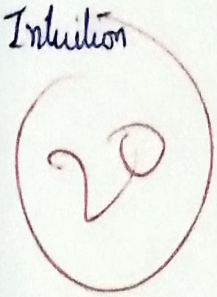
and passing them (divided arrays) into recursive calls

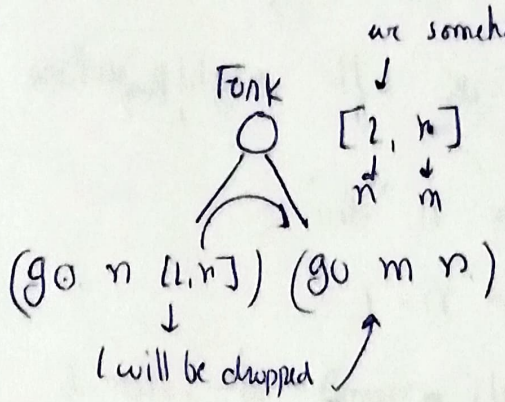
Done later

But this takes $O(n^2)$ time for length and splitAt

Better solution :-

Intuition





So while going left down the tree, we are removing one element from the front

So, go will be function that takes the size and the remaining array and returns a tree and the remaining array after the operation

* This def of go in (go rem size) unlike the prev page

Rough work

buildTree :: [Int] ⟶ BTree

buildTree arr = fst . (go arr n) where

n = length arr

go :: [Int] ⟶ Int ⟶ (BTree, [Int])

go (x:xs) 1 = ((Leaf x), xs)

go rem n = (Fork left right, pass) where

l = n `div` 2

r = n - l

(left, rem_) = go . rem l

(rem_)

(right, pass) = go rem_ r

Keeps the remaining after building the left subtree

Pass variable: Passes it to the next right subtree on fork

Time complexity:

This takes O(n) Times because in each case we are removing the first element and adding it to the tree

length takes O(n) which is called once and

Rough work

Here is also the naive approach :

```
buildTree :: [Int] → BTree

buildTree [a] = (leaf a)

buildTree arr = Fork (buildTree left)(buildTree right)
                where
                  n = length arr
                  (left, right) = splitAt (n `div` 2) arr
```