Assignment 3 COL106

Name - Shrey Patel
Entry No. - 2019CS10400
Group - 9

A 3.2 Assuming n-operations have already been performed, determine the worst case time complexity of Free, Allocate and Defragment using BST.

① **Free:**

→ Free(startAddr) performs an exact search in allocBlk for a block with address equal to startAddr, and if found, removes it from allocBlk and inserts it into freeBlk.

→ All these operations take worst time when size of allocBlk is O(n) [After n allocate operations on freeBlk], as well as the size of freeBlk is O(n)

Suppose

h_f = Height of freeBlk BST

h_a = Height of allocBlk BST

→ Search for startAddr = $O(h_a)$
Delete from allocBlk = O(1) (after searching)
Insert to freeBlk = $O(h_f)$

→ **Total time = $O(h_a) + O(h_f)$**
In worst case, height of both the trees can be as large as $O(n)$

∴ **Worst case time = $O(n)$**

② Allocate:

→ Allocate(blockSize) performs a best fit search for a block of size > blockSize in freeBlk, and if found, splits or deletes that block from freeBlk and inserts it into allocBlk.

→ All these operations take maximum time when size of freeBlk and allocBlk is maximum i.e. $O(n)$. Note that size of freeBlk can be made $O(n)$ by n free operations on allocBlk.

Again suppose h_a and h_f as in Free

Best Fit
→ Search for blockSize = $O(h_f)$
Delete/Split the found block = O(1) after searching
Insert to allocBlk = $O(h_a)$

Total time = $O(h_f) + O(h_a)$

In worst case, height of both the trees can be as large as $O(n)$

∴ Worst case time = $O(n)$

(3) Defragment

→ In defragment, we traverse through all the blocks of freeBlk, all of which are sorted by their size, and now we sort them by their address by maintaining an auxiliary tree. So, extra space used = $O(n)$ if size of freeBlk = $O(n)$

→ Defragment takes maximum time when size of freeBlk is maximum i.e. $O(n)$ (after n free operations on allocBlk)

(1) Traversal (In-order) of freeBlk:

By Euler Walk, in-order traversal of any BST takes $O(n)$ time where n = no. of nodes

(2) Insertion into auxiliary tree:

→ If the auxiliary tree is currently of height h, then insertion of next node takes $O(h)$ which can be as large as $O(m)$ where m = no. of nodes currently in auxiliary

→ So, total time of insertion will be maximum when the nodes are inserted in already sorted order, in which case the tree grows by a height of 1 in every step.
So, total time of insertion:

$$0 + 1 + 2 + \cdots\cdots (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

③ Merging contiguous blocks

→ This requires us to traverse the blocks (nodes) of the auxiliary tree (in order) and combine two adjacent blocks if they are contiguous. Again, by Euler walk, worst time = $O(n)$

→ Note that we also delete the already visited blocks from the auxiliary tree after simultaneously inserting the merged blocks into freeBlk.
(Deletion takes $O(1)$ time if we already have the pointer to that node)

④ Insertion of merged blocks to freeBlk

→ Both freeBlk and auxiliary tree are BSTs and will have same no. of nodes n in the worst case, because it may happen that no blocks are contiguous, so no. of merged blocks = no. of original blocks = n

So, for similar to (2), worst case time of inserting = $O(n^2)$

→ Total worst case time of defragment in BST,

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

A 3.3 Determine the worst case time complexity of Free, Allocate and Defragment using AVL Trees.

(1) Free:

→ Similar to BST Free, Free(startAddr) first performs an exact search in allocBlk for startAddr and if found, inserts it into freeBlk and deletes from allocBlk.

→ Assuming freeBlk and allocBlk to have size $O(n)$ and

h_f = Height of freeBlk AVL Tree

h_a = Height of allocBlk AVL Tree

→ Search for startAddr = $O(h_a)$

Delete from allocBlk = $O(1)$ (after searching)

Insert to freeBlk = $O(h_f)$

→ $\boxed{\text{Total time} = O(h_a) + O(h_f)}$

In worst case, h_a and h_f will be no larger than $\log n$, because of height balancing property of AVL Trees.

So, worst case of free operation = $O(\log n)$

(2) Allocate :

→ As in BST, Allocate(blockSize) performs a best fit search for blockSize, deletes/splits it from freeBlk and inserts that block to allocBlk.

→ Assuming size of freeBlk and allocBlk to be $O(n)$ (for worst case time complexity) and taking h_a and h_f as heights of allocBlk and freeBlk.

→ Best Fit search for blockSize = $O(h_f)$

Delete/split the found block = $O(1)$ after searching

Insert to allocBlk = $O(h_a)$

→ Total time = $O(h_f) + O(h_a)$

And in worst case, $h = O(\log n)$, from AVL property

∴ Worst case time of allocate = $O(\log n)$

③ Defragment

→ In defragment, similar to BST, we traverse in-order through the blocks of freeBlk, which are sorted by size, and now we sort them by their address by maintaining an auxiliary tree.

→ Defragment takes maximum time when size of freeBlk is maximum i.e. O(n) (after n-operations of free on allocBlk)

→ So, size of auxiliary tree is also O(n)

Extra space used = O(n)

① Traversal (In-order) of freeBlk :

→ Traversal (In order) of any binary tree takes O(n) time at worst by Euler walk

② Insertion into auxiliary tree :

→ If auxiliary tree is currently containing n nodes and has height h, then insertion of new node takes O(h) time. But, from AVL property, $h = O(\log n)$ in worst case.

→ So, worst case time of insertion :

$\log 1 + \log 2 + \cdots \log n = \log(n!)$

$= \boxed{O(n \log n)}$

(3) Merging contiguous blocks:

→ This requires us to traverse the blocks (nodes) of the auxilliary tree (in order) and combine two adjacent blocks if they are contiguous. Again, by Euler Walk, worst time = $O(n)$

→ We simultaneously also insert the merged block into the freeBlk after which we delete it from the auxiliary tree. (This deletion takes $O(1)$ time as we already have the pointer to that node)

(4) Insertion of merged blocks to freeBlk:

→ No. of merged blocks can be equal to original no. of nodes (i.e. $O(n)$) in worst case, if no two blocks are contiguous.

→ So, similar to case (2), worst case of insertion = $\boxed{O(n \log n)}$

→ Total worst case time of defragment in AVL tree = $O(n) + O(n \log n) + O(n) + O(n \log n)$

= $\boxed{O(n \log n)}$