

## CSE 321 - Homework 2

### 1. Master Theorem:

Let  $T(n)$  be an eventually non-decreasing function that satisfies the recurrence relation:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) ; \quad n = b^k \quad (k = 1, 2, \dots)$$

$$T(1) = c$$

$$\text{where } a \geq 1, \quad b \geq 2, \quad c > 0$$

If  $f(n) \in \Theta(n^d)$  where  $d \geq 0$ , then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \text{ for all } n, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

a)  $T(n) = 2 \cdot T\left(\frac{n}{4}\right) + \sqrt{n \log n}$

$f(n) = \sqrt{n \log n} \in \Theta(\sqrt{n \log n}) \Rightarrow$  Since we can't write  $\sqrt{n \log n}$  as  $n^d$ , we can not use Master Theorem here.  
 $a=2, b=4$

b)  $T(n) = 9 \cdot T\left(\frac{n}{3}\right) + 5n^2$

$a=9, b=3, f(n) = 5n^2 \in \Theta(n^2)$   
 $d=2, b^d = 3^2 = 9$

$a = b^d = 9 \therefore T(n) \in \Theta(n^d \log n) = \Theta(n^2 \log n)$

c)  $T(n) = \frac{1}{2} \cdot T\left(\frac{n}{2}\right) + n$

$a = \frac{1}{2} < 1 \therefore$  We can not use Master Theorem here.

d)  $T(n) = 5 \cdot T\left(\frac{n}{2}\right) + \log n$

$a=5, b=2, f(n) = \log n \in \Theta(\log n) \Rightarrow$  We can not write as  $n^d$ , so it is not suitable for Master Theorem.

e)  $T(n) = 4^n \cdot T\left(\frac{n}{3}\right) + 1$

$a = 4^n$  is not a constant, so we can not use Master Theorem.

f)  $T(n) = 7 \cdot T\left(\frac{n}{4}\right) + n \log n$

$a=7, b=4, f(n) = n \log n \in \Theta(n \log n) \Rightarrow$  We can not write  $n \log n$  as  $n^d$ , so we can not use Master Theorem.

g)  $T(n) = 2 \cdot T\left(\frac{n}{3}\right) + \frac{1}{n}$

$a=2, b=3, f(n) = n^{-1} \Rightarrow d = -1 < 0$ , so we can not use Master Theorem.

h)  $T(n) = \frac{2}{5} \cdot T\left(\frac{n}{5}\right) + n^5$

$a = \frac{2}{5} < 1$ , Therefore, we can not use Master Theorem here.

$$2) A = \{3, 6, 2, 1, 4, 5\}$$

Algorithm:

→ Mark the first element as sorted and, for each unsorted element  $x$ , extract the element  $x$ , compare with all sorted element. For every element in sorted part, if the element is greater than  $x$ , move to right. If not put the  $x$  there and count it as in the sorted part.

- ① Sorted  $\rightarrow \{3\}$ , Array  $\rightarrow \{3, 6, 2, 1, 4, 5\}$   
 $3 < 6$ , then no need to change.
- ② Sorted  $\rightarrow \{3, 6\}$ , Array  $\rightarrow \{3, 6, 2, 1, 4, 5\}$   
 $6 > 2$ , move 6 to the right.  
 $3 > 2$ , move 3 to the right.
- ③ Sorted  $\rightarrow \{2, 3, 6\}$ , Array  $\rightarrow \{2, 3, 6, 1, 4, 5\}$   
 $6 > 1$ , move 6 to the right.  
 $3 > 1$ , " 3 " " " "  
 $2 > 1$ , " 2 " " " "
- ④ Sorted  $\rightarrow \{1, 2, 3, 6\}$ , Array  $\rightarrow \{1, 2, 3, 6, 4, 5\}$   
 $6 > 4$ , move 6 to the right.  
 $3 < 4$ , no need to move, put 4 in between 3 and 6
- ⑤ Sorted  $\rightarrow \{1, 2, 3, 4, 6\}$ , Array  $\rightarrow \{1, 2, 3, 4, 6, 5\}$   
 $6 > 5$ , move 6 to the right.  
 $4 < 5$ , no need to move, put 5 in between 4 and 6.
- ⑥ Sorted  $\rightarrow \{1, 2, 3, 4, 5, 6\}$ , Array  $\rightarrow \{1, 2, 3, 4, 5, 6\}$   
 No unsorted element left, sorting is complete.

3) a) i. Arrays access their every element in constant time with the given index. They have the capability of random accessing since it knows its elements' location in memory from the fact that every element of the array  $L[i]$ , locates  $L[i-1] + \text{sizeof}(\text{elem})$

$$W_a = 1 \in \Theta(1)$$

Linked Lists do not know every element's location from the beginning, we have to trace from the root element to the desired element. But in this scenario, accessing the first element is constant, since we know the root element's address.

$$W_l = 1 \in \Theta(1)$$

ii. For arrays accessing the first or last does not matter from the reason explained on i.

$$W_a = 1 \in \Theta(1)$$

Linked List should trace from the beginning to the end to access the last element. Every element has the intel of next element's location. We can not know from beginning.

$$W_l = n \in \Theta(n)$$

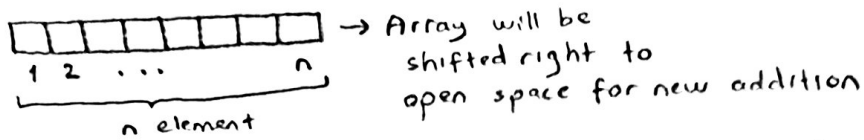
iii. For arrays it does not matter.

$$W_a = 1 \in \Theta(1)$$

Linked List should travers from root to middle element to access it. Let's say the middle element is located in  $\frac{n}{2}$  th node:

$$W_l = \frac{n}{2} \in \Theta(n)$$

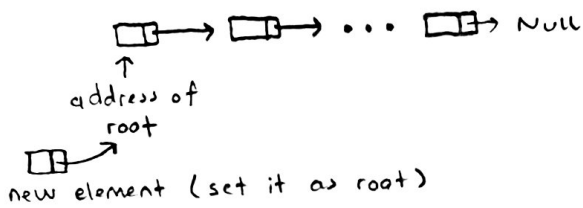
iv. To add element at the beginning of an array, all the element of the array should be shifted right.



There are  $n$  elements to be shifted, so  $n$  operations.

$$W_a = n \in \Theta(n)$$

- In linked list we know the address of the first element. To add element at the beginning it is enough to set this new element's next element as current first element and set this new element as the first node of the linked list.



new element (set it as root)

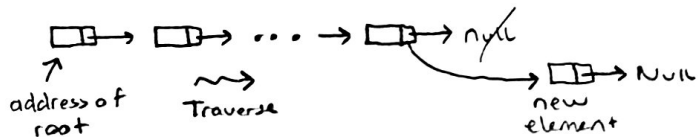
Setting next element of the new element and marking it as root element are constant procedure.

$$W_l = 1 \in \Theta(1)$$

v. If there is space at the end of an array, we can simply put the new element in constant time without any operation.

$$W_a = 1 \in \Theta(1)$$

In worst case scenario we don't have the address of the last element of the linked list. So, to put new element at the end, we should first traverse to the end and set the next element of the last element as the new element.



Traversing the linked list with  $n$  elements to the end is linear time operation,

$$W_l = n \in \Theta(n)$$

vi. To add new element in the middle, we should shift half of the array to the right.

It is  $\sim \frac{n}{2}$  operations,

$$W_a = \frac{n}{2} \in \Theta(n)$$

To add new node in the middle we should traverse to the middle node ( $\frac{n}{2}$ th node), set its next element as new node, set new node's next element as  $(\frac{n}{2} + 1)$ th node.

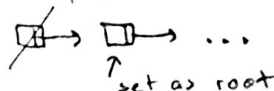
$$W_l = \frac{n}{2} + 2 \in \Theta(n)$$

vii. To delete the first element of the array, we should shift  $(n-1)$  elements to the left.

$$W_a = n-1 \in \Theta(n)$$

To delete the first element of linked list, simply set the second element as root and remove the first element from memory.

$$W_l = 2 \in \Theta(1)$$



### Viii. Deleting the last element

→ Set the size of the array as  $(n-1)$ . It is a constant operation.

For a worst case scenario, we can initialize a new array with size of  $(n-1)$  and copy the elements of the array to new array from 1st element to  $(n-1)$ th element. It is  $(n-1)$  operations.

$$W_a = (n-1) \in \Theta(n)$$

→ To delete the last node of the linked list, we should traverse to the  $(n-1)$ th node, set its next node as Null, decrease the size of the list, deallocate the memory for last element.

$$W_l = (n-1) + 3 \in \Theta(n)$$

### ix. Deleting any element in the middle.

→ Shift all the elements beyond  $\lfloor \frac{n}{2} \rfloor$ th element. It is  $n - \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil$  operations.

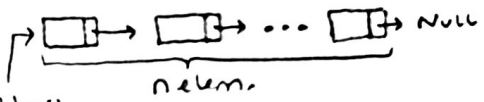
$$W_a = \frac{n}{2} \in \Theta(n)$$

→ Traverse the list to the  $\lfloor \frac{n}{2} \rfloor$ th node, set its next element as  $(\lfloor \frac{n}{2} \rfloor + 2)$ th element. Deallocate  $(\lfloor \frac{n}{2} \rfloor + 1)$ th element.

$$W_l = \frac{n}{2} + 2 \in \Theta(n)$$

### b) Space requirements:

→ Arrays:  →  $n \text{ spaces} \in \Theta(n)$   
n elem.

→ Linked List:  →  $n + n + 1$   
↓ ↓ ↓  
elements pointers address of root  
 $= 2n + 1 \text{ spaces} \in \Theta(n)$

### 4. Following is a three step solution:

1. Create a temp array `arr[]` that stores inorder traversal of the tree.
2. Sort the temp array.
3. Again, do inorder traversal of tree and copy sorted array elements to tree nodes one by one.

#### Pseudocode:

function `binaryTreeToBST (root, n)`:

`arr[] ← [n]`

for each inorder element in tree

    append element to `arr`

end for

`quickSort(arr)`

for each element in `arr`

    copy element to inorder element in tree

endfor

return

When we traverse the tree in order and replace its element with the corresponding sorted elements, the tree will become also sorted.

Best case: Traversing the tree will take  $\sim n$  iterations in each case.  
Complexity depends on the sorting algorithm used on the arr.

$$T_b = n + \underbrace{n \log n}_{\text{Best case for quick sort}} + n = 2n + n \log n \in \Theta(n \log n)$$

Traversing

Worst case:

$$T_w = n + \underbrace{n^2}_{\text{Worst case for quick sort}} + n = n^2 + 2n \in \Theta(n^2)$$

Average case:

$$T_a = n + \underbrace{n \log n}_{\text{Avg. case for quick sort}} + n = n \log n + 2n \in \Theta(n \log n)$$

5. function findPair (arr[], size, x):

i ← 0

j ← 1

while (i < size) & (j < size)

if (i != j) && ((arr[i] - arr[j] = x) || (arr[j] - arr[i] = x))

return (i, j)

end if

else if arr[j] - arr[i] < x

j++

end else if

else

i++

end else

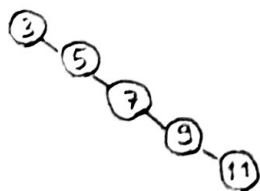
end while

return

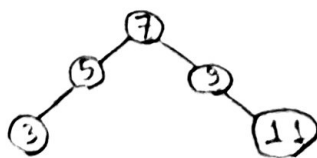
The algorithm above assumes it is a sorted list. It looks for two indices to find a pair in the array. If the sub of two pairs are smaller than the x value, we need a bigger upper value arr[j], so we increment j. If the sub is greater than the x value, we need a bigger lower value arr[i], so we increment i. Finally we will find a pair whose sub is equal to x, if any.

6. a) True. For instance lets say we have these values: {3, 5, 7, 9, 11}

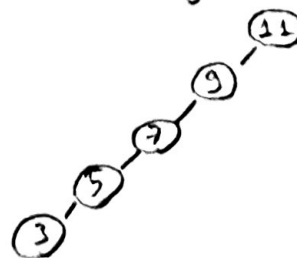
Insertion from left to right to BST:



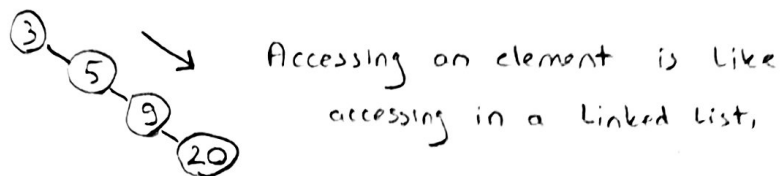
Insertion order: 7 → 5 → 3 → 9 → 11



Insertion from right to left

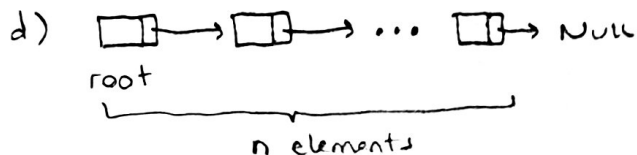


b) True, if it is shaped like that below:



There are no left, right nodes, only in one direction.

c) True, if it is a sorted array. Minimum element will be the  $L[1]$ , and the maximum element will be  $L[n]$ . Accessing these elements in array is constant time.



To apply binary search on linked lists, we should traverse to the middle node from root first. It is  $n/2$  operations and  $\theta(n)$

Searched element is smaller  $\rightarrow$  Traverse to  $n/4$ th node :  $\theta(n)$

" " " greater  $\rightarrow$  Traverse  $(n - \frac{n}{2})/2$  times :  $\theta(n)$

$W_T = \frac{n}{2} + \frac{n}{4} + \dots + \frac{n}{2^k} + \dots + 1 \rightarrow$  First element or Last element reached (worst case)

$2^k \rightarrow$  operations  
 $2^k = n$

$\log_2 n = k \Rightarrow \in \theta(\log n) \therefore \text{True}$

e) Let's say we have  $\text{arr}[ ] = \{21, 15, 8\}$ ,  $n=3$

① Sorted  $\rightarrow \{21\}$  Array  $\rightarrow \{21, \{15, 8\}\}$   
 $21 > 15$ , move 21 to right  
 No element left, put 15 at the beginning } 1 shift

② Sorted  $\rightarrow \{15, 21\}$  Array  $\rightarrow \{15, 21, \{8\}\}$   
 $21 > 8$ , move 21 to right  
 $15 > 8$ , move 15 " " } 2 shift  
 No element left, put here

③ Sorted  $\rightarrow \{8, 15, 21\}$

$1 + 2 = 3$  operations  $= \frac{2 \cdot 3}{2}$

If the given array's length  $= 4 = n$

$1 + 2 + 3 = 6$  operations  $= \frac{3 \cdot 4}{2}$

If  $n=5$

$1 + 2 + 3 + 4 = 10 = \frac{4 \cdot 5}{2}$

$$T_w = \frac{(n-1)n}{2} = \frac{n^2 - n}{2} \in \theta(n^2)$$

$\therefore \text{False}$