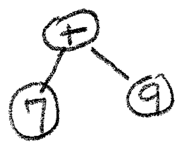
CS 315 SAMPLE FINAL ANSWERS F18

(1) RPN: 12 18 7 9 + - 5 % *

9
7
18
12
NODE STACK

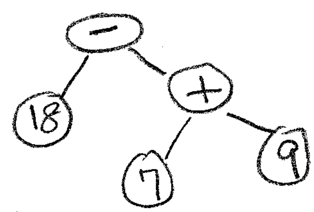
RPN: + - 5 % *

+
18
12
NODE STACK



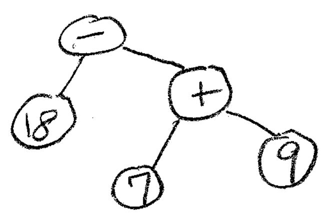
RPN: - 5 % *

−
12
NODE STACK



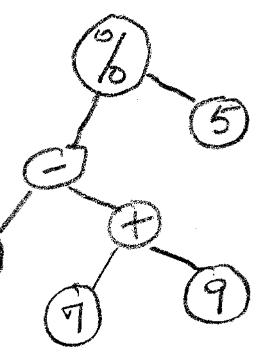
RPN 5 % *

5
12
NODE STACK



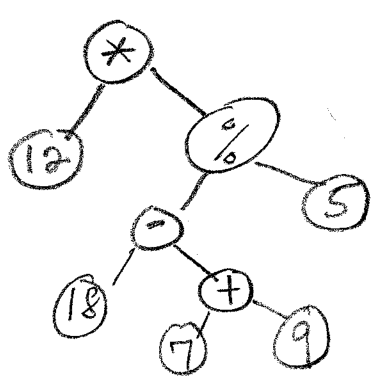
RPN: % *

%
12
NODE STACK



RPN: *

*
NODE STACK



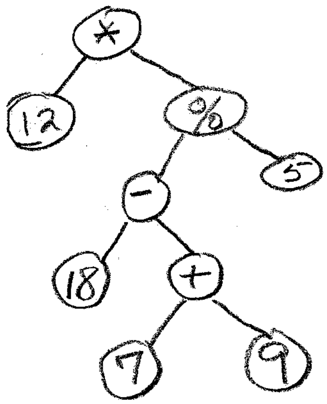
RPN: EMPTY

EMPTY
NODE STACK

2(a) postorder

(b) Stack

(3)



Using postorder with an operand stack.

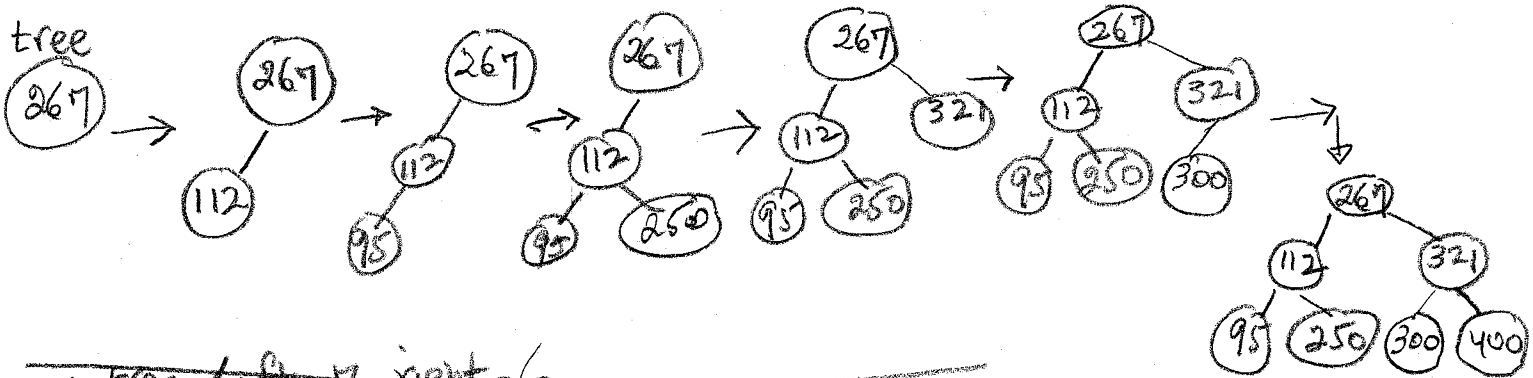
								answer
			9					
		7	7	16		5		
	18	18	18	18	2	2	2	24
12	12	12	12	12	12	12	12	
stack	stack	stack	stack	stack	stack	stack	stack	stack

(4)

0	1	2	3	4	5	6
95	112	250	267	300	321	400

[SEE PAGE 4 FOR CLEAN VERSION]

NOTE: THE VARIABLE 'array' WILL ALWAYS REFER TO THE ARRAY ABOVE.
THE VARIABLE 'tree' WILL ALWAYS REFER TO THE GRAPHICAL TREE BEING BUILT.



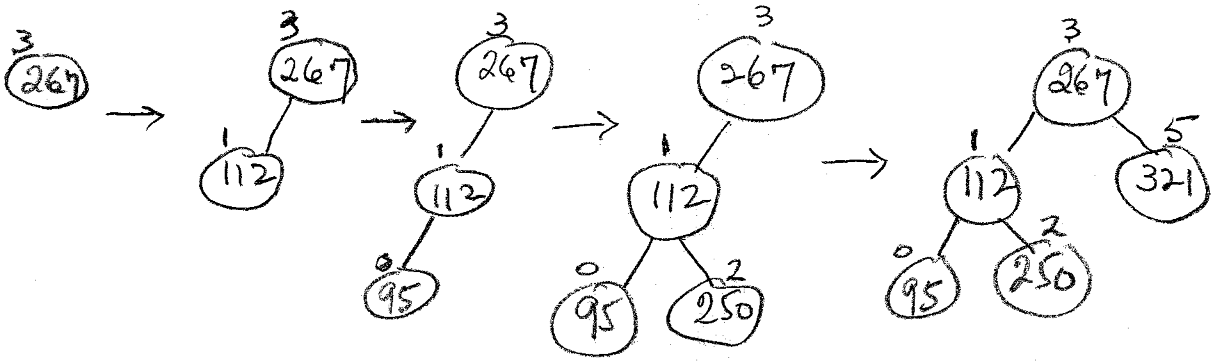
~~array tree left=7 right=6~~
~~array tree left=6 right=5~~
~~array tree left=6 right=6 mid=6 buildBST(array, tree, 6, 5) buildBST(array, tree, 7, 6)~~
~~array tree left=5 right=4~~
~~array tree left=4 right=3~~
~~array tree left=4 right=4 mid=4 buildBST(array, tree, 4, 3) buildBST(array, tree, 5, 4)~~
~~array tree left=4 right=6 mid=5 buildBST(array, tree, 4, 4) buildBST(array, tree, 6, 6)~~
~~array tree left=3 right=2~~
~~array tree left=2 right=1~~
~~array tree left=2 right=2 mid=2 buildBST(array, tree, 2, 1) buildBST(array, tree, 3, 2)~~
~~array tree left=1 right=0~~
~~array tree left=0 right=-1~~
~~array tree left=0 right=0 mid=0 buildBST(array, tree, 0, -1) buildBST(array, tree, 1, 0)~~
~~array tree left=0 right=2 mid=1 buildBST(array, tree, 0, 0) buildBST(array, tree, 2, 2)~~

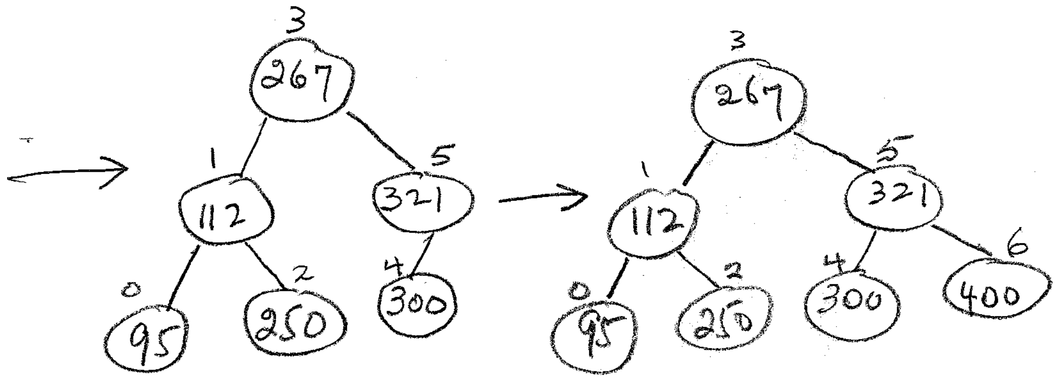
~~array tree left=0 right=6 mid=3 buildBST(array, tree, 0, 2)~~ buildBST(array, tree, 4, 6)

→ buildBST(array, tree, 0, 6)

Start

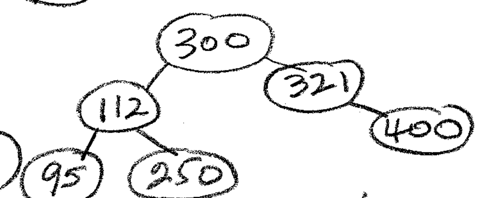
(4) THIS SHOWS HOW THE BST IS BUILT ONE NODE AT A TIME. THE NUMBERS ABOVE EACH NODE REPRESENT THE ARRAY INDICES.





(5) (a) 95, 250, 112, 300, 400, 321, 267

(b)



(6) Going one to the left and all the way to the right finds the biggest value in the left subtree. So swapping that number into the node to be deleted will still keep every node to the left smaller.

Similarly, going one to the right and then all the way left will find the smallest value in the right subtree. So swapping will again preserve the BST.

Example using tree from question (4)

Let's remove 267.

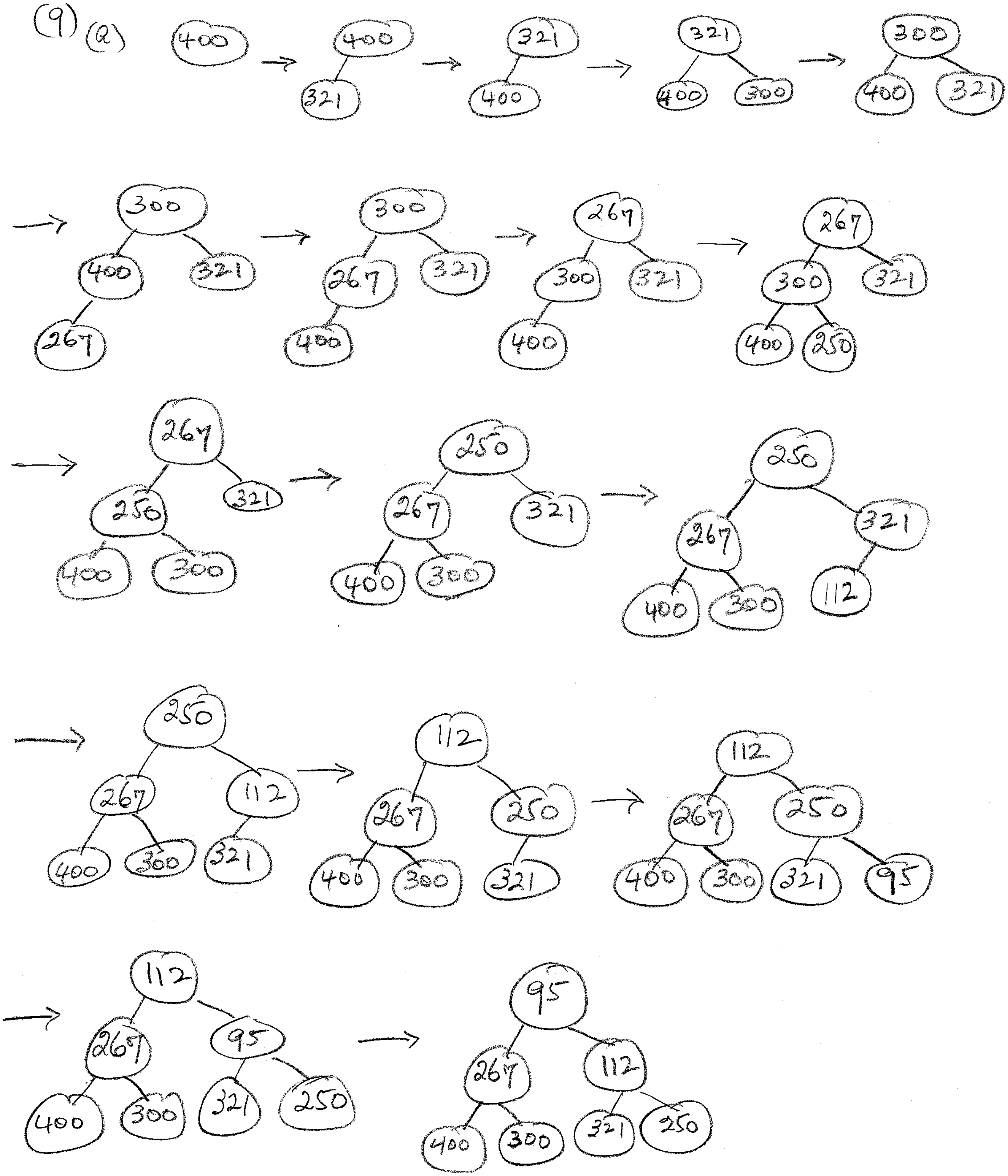
(a) go left once and then all the way right finds 250. Replacing 267 with 250 keeps the BST.

(b) go right once and then all the way left finds 300 which also keeps the BST.

(7)
```
int addNodesOnBST(TreeNode<Integer> root)
{
    if(root == null) return 0;
    int leftSum = addNodesOnBST(root.left);
    int rightSum = addNodesOnBST(root.right);
    return leftSum + rightSum + root.data;
}
```

(8)
```
int countPrimeNodesOnBST(TreeNode<Integer> root)
{
    if (root == null) return 0;
    int prime = 0;
    if (isPrime(root.data)) prime = 1;
    int leftCount = countPrimeNodesOnBST(root.left);
    int rightCount = countPrimeNodesOnBST(root.right);
    return leftCount + rightCount + prime;
}
```

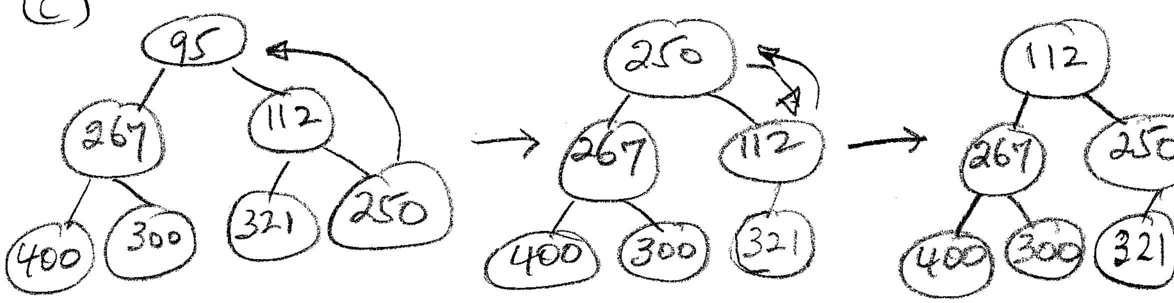
(9) (a)



9 (b) Heap assigned to an array

0	1	2	3	4	5	6
95	267	112	400	300	321	250

(c)



(d)

0	1	2	3	4	5	6
95	267	112	400	300	321	250

0	1	2	3	4	5	
250	267	112	400	300	321	

Smaller child of index 0 is at position 2 so swap

0	1	2	3	4	5
112	267	250	400	300	321

(10) (a)

Vertex		List
1	⟶	2, 3, 6
2	⟶	1, 3, 7
3	⟶	1, 2, 4
4	⟶	3, 5
5	⟶	4, 6, 7
6	⟶	1, 5, 7
7	⟶	2, 5, 6

(b)

	1	2	3	4	5	6	7
1	0	1	1	0	0	1	0
2	1	0	1	0	0	0	1
3	1	1	0	1	0	0	0
4	0	0	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	1
7	0	1	0	0	1	1	0

(10)(C) BREADTH-FIRST

(1) QUEUE: 4
VISITED: 4
EDGE LIST:

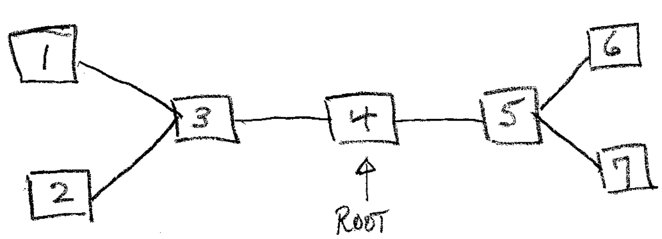
(2) QUEUE: 3, 5
VISITED: 4, 3, 5
EDGE LIST: 4→3, 4→5

(3) QUEUE: 5, 1, 2
VISITED: 4, 3, 5, 1, 2
EDGE LIST: 4→3, 4→5, 3→1, 3→2

(4) QUEUE: 1, 2, 6, 7
VISITED: 4, 3, 5, 1, 2, 6, 7
EDGE LIST: 4→3, 4→5, 3→1, 3→2, 5→6, 5→7

(5) THE NODES CURRENTLY ON THE QUEUE CANNOT BRING IN ANY OTHER NODES BECAUSE ALL HAVE BEEN VISITED SO EVERY OTHER STEP JUST BRINGS THEM OFF THE QUEUE

RESULT:
QUEUE: EMPTY
VISITED: 4, 3, 5, 1, 2, 6, 7



(10)(d) DEPTH-FIRST

STEPS

(1)

Stack
4

VISITED: 4
EDGE LIST:

(2)

Stack
3
4

VISITED: 4, 3
EDGE LIST: 4→3

(3)

Stack
1
3
4

VISITED: 4, 3, 1
EDGE LIST: 4→3, 3→1

(4)

Stack
2
1
3
4

VISITED: 4, 3, 1, 2
EDGE LIST: 4→3, 3→1, 1→2

(5)

Stack
7
2
1
3
4

VISITED: 4, 3, 1, 2, 7
EDGE LIST: 4→3, 3→1, 1→2, 2→7

(6)

Stack
5
7
2
1
3
4

VISITED: 4, 3, 1, 2, 7, 5
EDGE LIST: 4→3, 3→1, 1→2, 2→7, 7→5

(7)

Stack
6
5
7
2
1
3
4

VISITED: 4, 3, 1, 2, 7, 5, 6
EDGE LIST: 4→3, 3→1, 1→2, 2→7, 7→5, 5→6

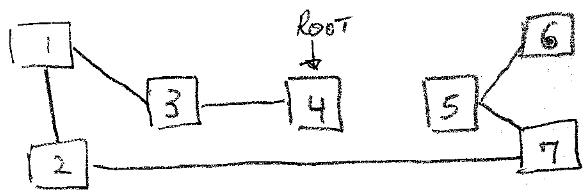
(10)(d)

STEPS 8 → 14

STACK IS POPPED ONE VERTEX AT A TIME AND EACH TIME YOU HAVE TO SEE IF ANY CONNECTED VERTEX HAS NOT BEEN VISITED AS YET.
IF THERE ARE UNVISITED VERTICES, YOU HAVE TO PUSH THEM ON THE STACK AND GO UP AGAIN.
IN THIS CASE, THE STACK WILL BE POPPED COMPLETELY.

RESULT:

VISITED: 4, 3, 1, 2, 7, 5, 6



(11) 'vertices' REFERS TO AN ArrayList OF WHATEVER TYPE A VERTEX IS.

'adjacencyLists' IS AN ArrayList OF ArrayList(S) WHERE EACH OF THE COMPONENT ArrayList(S) CORRESPONDS WITH A VERTEX STORED 'vertices'.
EVERY TIME A VERTEX OBJECT IS ADDED TO THE 'vertices' LIST A NEW ArrayList IS CREATED AND ADDED TO THE 'adjacencyLists' LIST.

Example: vertices → [A, B, C]

adjacencyLists → [[ ], [ ], [ ]]

ONCE CONNECTED VERTICES START TO BE ADDED THE EMPTY LISTS WILL START TO GET VERTICES.

(12) (a) THE TOP OF THE HEAP IS THE FIRST ELEMENT IN ANY REPRESENTATION OF A HEAP SO IT'S A DIRECT ACCESS. SO CONSTANT TIME = $O(1)$ or $O(K)$

(b) A SPANNING TREE MUST INCLUDE ALL n VERTICES AND THERE ARE NO LOOPS SO $O(n)$.

(c) THERE ARE n VERTICES TO START FROM AND EACH TREE BUILT IS $O(n)$ SO $O(n^2)$.

(d) AN ADJACENCY MATRIX IS A 2-D ARRAY AND ACCESSING ONE ELEMENT IN AN ARRAY TAKES CONSTANT TIME SO $O(1)$ OR $O(K)$.