Thanks Marina <3

COLLABORATIVE SOLUTIONS FOR PRACTICE PROBLEMS
CSE 12

FINAL IN < 1 HOUR GL FAM ⏚ *- Alberto Einstein*

In order to maintain a productive study resource please:
use the google comment system to ask questions or leave comments.

- How to use comments: https://support.google.com/docs/answer/65129?hl=en
- add the question along with the answer
- add an explanation for your answer
- Always doubt everyone else's answers until you confirm for yourself they're correct

---

Questions from Final Review Discussion Slides

Given a Sorted Doubly Linked List with a head:

1. What is the asymptotic complexity of finding the smallest element in the List?
   → Θ(1)
   (Doesn't this depend on the definition of sorted ? if it's B-> S it'll be Θ(n))
2. What is the asymptotic complexity of finding the largest element in the list?
   → Θ(n), we don't have a tail pointer so we have to iterate through the whole list to reach to the end of the list which will have the largest element as the list is sorted.
3. What is the asymptotic complexity of determining whether a given element e appears in the list?
   → Best Case Θ(1), the element we're searching for is in the first index. Worst Case Θ(n), the element we're searching for is in the last index or it is not in the list. Average Case Θ(n).
4. What is the asymptotic complexity of finding the median element in the list?
   → Getting the element at the midpoint is Θ(n).
5. What is the asymptotic complexity of deleting a given element e in the list?
   → Best Case Θ(1), the element we're deleting is in the first index. Worst Case Θ(n), the element we're deleting is in the last index or it is not in the list. Average Case Θ(n).

True or False:

1. If class A implements interface I, class B extends A, and class C extends B, then C implements I. True
    a. Inheriting (extending a class) means that you have access to methods of the class you are extending from and you are probably adding some of your own. I think this is true because since C extends B that extends A that implements I, C must implement I.
    b. Adapter design pattern is not inheriting. That is making an instance of another class and using methods only on that instance variable
2. If class A implements interface I, class B extends A, class C extends B, and interface J extends interface I, then C implements J. False.
    a. There might/can be a method defined in J that C might/won't need to implement, therefore C implements J should be false.
3. 3^n is $O(2^n)$. False
    a. 3^n grows faster than 2^n. So 2^n cannot be upper bound for 3^n.
4. The finally block in a try-catch-finally statement is executed even if no exception is thrown and there is a return statement in the try block
    a. https://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html. True - "allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated."
5. A stack is a FIFO structure and a queue is a LIFO structure.
    a. False. Stack is LIFO and Queue is FIFO.
6. A collision occurs when two identical keys result in two identical hash values
    a. False. Description on comments →

What is a hash function?

→ It is a function that generates a unique index/value for a specific key.

What are the properties of a good hash function? What is the load factor of a hash Table?

→ Has to be deterministic (give same value always), uniform (distributed evenly) and fast.

→ Load factor = (number of elements added to the hash table / the size of the array)

What is the problem with Linear Probing and Quadratic Probing respectively and how does Double Hashing solve this issue?

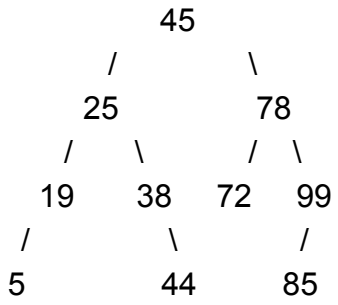
→ Linear probing can create primary clustering. Also, if bridge value is not used we might not get valid answer to specific methods. Quadratic probing can create secondary clustering, and there is a chance of having infinite loop while implementing quadratic hashing function.

→ Bridge value - Its when you add 53, and 43 and remove 53. The index 3 might be empty but we still have 43 on the hash table. When searching for 43 we might get false.

→ while quadratic probing results in secondary clustering (for when keys hash to the exact same location and probe by the same quadratic). Double hashing solves this issue because it continues to use the unique key as part of how to find the new index, so as long as keys are unique there is less of a chance of clustering.

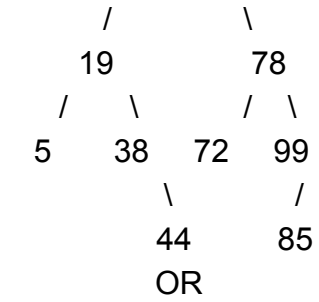
Complexity of Different Data Type methods:
https://docs.google.com/document/d/163Iv3TCOYiljG1dYxWJuqjUsKX1MxBTNH-RBb5e6I3U/edit

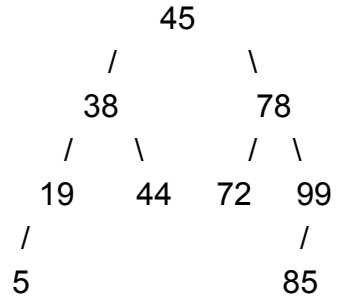
Draw the BST that results from inserting 45, 78, 25, 19, 38, 99, 72, 44, 85, 5 in that Order



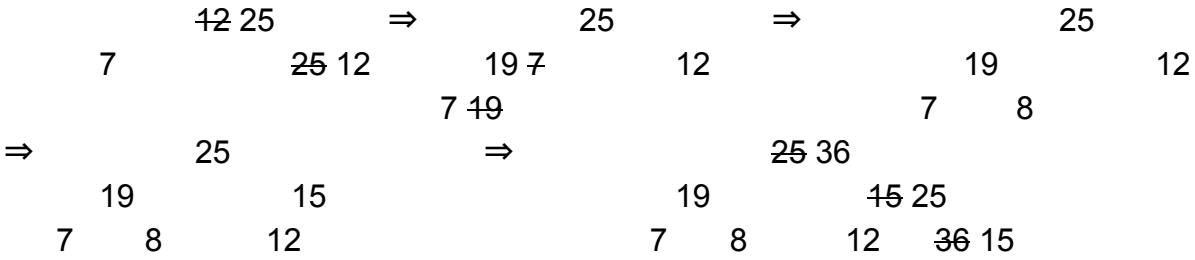
Height of the tree, the number of edges between the root and the farthest leaf from the root- 3
Leaf count: 4

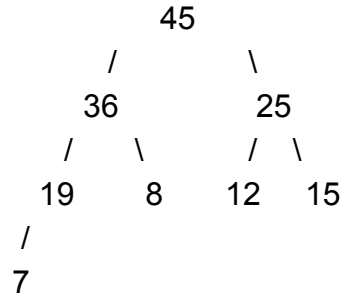
After deleting 25 the tree will be





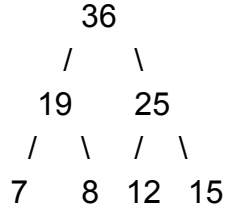
Draw the 2-max-heap that results from inserting 12, 7, 25, 19, 8, 15, 36, 45 in that Order.





Inorder traversal: 7, 19, 36, 8, 45, 12, 25, 15 (Left, Root, Right)
Preorder traversal: 45, 36, 19, 7, 8, 25, 12, 15 (Root, Left, Right)
Postorder traversal: 7, 19, 8, 36, 12, 15, 25, 45 (Left, Right, Root)

After one remove operation:



Coding Questions

Write a RECURSIVE method to reverse a Singly Linked List with a head.

(Method call with head and a null pointer as parameters)

```
Void reverse(Node currentNode, Node previousNode){
     if(currentNode == null) {
          if(previousNode == null) // list is empty

          else // making the end of the list the head
               head = previousNode;
     }
     else{
          reverse(currentNode.getnext(), currentNode);
          currentNode.setNext(previousNode);
     }
}
```

Write a method to count the number of nodes in a BST with Integer values, whose value is greater than k

```
public int countNodes(BSTNode root, int k) {
     if (root == NULL)
          return 0;
     k += 1 + countNodes(root.getLChild(), k) +
     countNodes(root.getRChild(), k) ;

     Return k;
```

```
}
```

// Here's my code. The above code needs to only count nodes that have values greater than k.

```
public int countNodes(BSTree tree, int k) {
     return helperCount(tree.getRoot(), k);
}

private int helperCount(Node root, int k) {
     // returns 0 if the node is null
     if (root == null) {
          return 0;
     }
     // returns 1 and the amount of nodes greater than k on both
sides below it
     if (root.getValue() > k) {
          return 1 + helperCount(root.getLeft(), k) +
helperCount(root.getRight(), k);
     }
     // returns just the amount of nodes greater than k on both
sides below it
     else {
          return helperCount(root.getLeft(), k) +
helperCount(root.getRight(), k);
     }
}
```

Given a list, print all the pairs whose difference is k. Your solution must be as time-efficient as possible. (What data structure do you think is useful here?)

→ Hashtable?? Add k to each element on the list and search for that key in the hashTable.

Given two DoublyLinkedLists with head, write a method that returns an intersection of the lists: public intersection(DoublyLinkedList1, DoublyLinkedList2)

→ What does intersection mean here. My guess was all the numbers that are in both the list, but its intersection instead of intersections so I'm a,, little confused here.

```
public DoublyLinkedList<E> intersection(DoublyLinkedList<E> doublyLinkedList1,
DoublyLinkedList<E> doublyLinkedList2) {
	DoublyLinkedList<E> intersect = new DoublyLinkedList<E>();
	Node current = doublyLinkedList1.getHead();

	while (current != null) {
		if (doublyLinkedList2.contains(current.getElement())) {
			intersect.add(current.getElement());
		}
		current = head.getNext();
	}
	return intersect;
}
```

Write a method to determine whether the contents of a Singly Linked List with a head forms a palindrome. Assume that each node contains a char data. Your solution must be time-efficient and you may use any additional data structures you wish

→

```
Public boolean isPalindrome(SinglyLinkedList myList){
	Stack newStack = new Stack();
	Queue newQueue = new Queue();
	Boolean areSame = true;
	for(int i = 0; i< myList.size(); i++){
		newStack.push(myList.get(i));
		newQueue.push(myList.get(i));
	}
	while(areSame){  //shouldn't this be while(!newStack.isempty())
		if( ! myStack.pop().equals(myQueue.pop()) )
			areSame = false;
	}
	Return areSame;
}
```

Given two strings A and B, find if one is a permutation of the other.
○abb and bba - true
○abcd and acdb - true
○acc and ccb - false

→ I feel like we should do this kinda like we did anagram extra credit for the assignment 8 ⇒ Sort the string first, and then do .equals in this case. Anyone else sees a better solution.

You are given a list of n-1 integers and these integers are in the range of 1 to n. There are no duplicates in list. One of the integers is missing in the list. Find the missing number.

Add all numbers in the list and subtract by the sum from 1 to n.

Find if a string contains all unique characters. Your solution must be as efficient as possible.

→ Use hash table to store all the string, using unique characters' ASCII value. If there's a collision then the strings is not unique.

→ (Answer, when we cannot use additional O(n) space)

Write a method to find the element in a BST that is one larger than the root.

→ (Assuming we get head as the parameter on method call)

```
Public E nextLargestFromRoot(BSTree tree){
    // Check if tree is not null
    Int elementToSearch = (tree.getElement() + 1);
    tree = tree.getRight(); // Since we're looking at larger element on BST
    while(tree != null){
        if elementToSearch is less than the element that is referenced by tree
            tree = tree.getLeft
        Else if elementToSearch is greater
            tree = tree.getRight
        Else
            Return tree.getElement
    }
    Return null
}
```

Write a method to print the levelOrderTraversal of a Binary Tree.

→ Assuming we get head as the parameter on method call

```
Public void levelOrderTraversal(BSTree tree){
        // Check if tree is null
        Queue newQueue = new Queue();
        newQueue.push(tree);
        while( ! newQueue.isEmpty() ){
                tree = newQueue.pop
                S.o.p( tree )
                If (tree != null && tree.leftnode != null) { newQueue.push( tree.getLeft ) }
                If (tree != null && tree.rightnode != null) { newQueue.push( tree.getRight ) }

        }
}
```

Write a method to find the sum of all leaf nodes in a BST with Integer data.

→

```
Public void sumofLeaf(BSTree tree){
        // Check if tree is null
        Queue newQueue = new Queue();
        newQueue.push(tree);
        Int sum = 0;
        while( ! newQueue.isEmpty() ){
                tree = newQueue.pop
                if(tree != null && tree.leftnode == null && tree.rightnode ==null)
                        ~~Sum++;~~ sum+= tree.data;
                If (tree != null && tree.leftnode != null) { newQueue.push( tree.getLeft ) }
                If (tree != null && tree.rightnode != null) { newQueue.push( tree.getRight ) }

        }
}
```

Practice Final Questions:

Complexity

1. Consider a small array a and large array b. Accessing the first element of a takes more/less/the same amount of time as accessing the first element of b
    a. The same amount of time
2. True or False? Big-O is used as an upper bound. Prove your answer
    a. $2^{N/2} = O(2^N)$
        i. True
        ii. 2^(N/2) = sqrt(2^N), so for N = 1 and c = 1 this is true. (2^(N/2) < 2^N)
    b. $N^2 + N^2 = O(N^3)$
        i. True
        ii. N^2 < N^3 as N approaches infinity because 1 < N. Therefore, for N = 1 and c = 1000000 this is true
    c. $100\ N^3 = \Omega(N^2)$
        i. True
        ii. N^3 > N^2 as N approaches infinity, so for c = 1 and N = 1 this is true (lower bound of N^3 is 1*N^2)
    d. $N \log N + N^2 = \Theta(N^2)$
        i. True
        ii. log N < N as N approaches infinity, so N^2 > N log N as N approaches infinity. Since the statement is true and N^2 = N^2, as 2N approaches infinity c*N^2 can be pushed above or below N^2 + N log N. For N = 1 and c = 100000000, O(N^2) is true and for N = 1 and c = 0.0000000001, $\Omega(N^2)$ is true.

What is the complexity of the mergeStep() method (merges two sorted arrays into one)?

→ Θ(N), <u>(where N is the size of the input arrays in total, total != adding the size of the two arrays.)</u>. Have to compare each element of each array to the other array's elements to see where to put.

→The O(n log n) of a mergeSort algorithm comes from splitting the array into 2 a bunch of times before merging

Big-Theta complexity?

```
for (int j = 1 ; j < n ; j *= 2)
    for (int i = 1; i<n; i++)
        Some_statement;
```

→ Θ(log(n) * n). Inner loop has Θ(n), and the outer loop has Θ(log(n)).

$$\sum_{j=1}^{log_2(n)} \left(\sum_{i=1}^{n} 2 \right) + 2 = \sum_{j=1}^{log_2(n)} 2n + 2 = log_2(n)\ (2n + 2) = \Theta(\ n\ log_2(n)\ )$$

Insertion,deletion for sorted and unsorted array

Sorted: Insertion - O(n); Deletion - O(n)

Unsorted: Insertion - O(1); Deletion - O(1) if we know the size O(n) if we don't

Generics

1. Does public static void myMethod(Collection<List> arg) {}... **no**

Change the method to public static void myMethod(Collection<? extends List > arg) {}

-> use a bounded wildcards that any thing that extend list

Change new Collection<LinkedList>(); to new LinkedList<LinkedList>();

-> You cannot make instance of Interface

-> It doesn't have to be LinkedList, any concrete class that extends List

2. Which of the following will compile?
   - Collection<List> c = new LinkedList<List>();
   - LinkedList<List> myL = new LinkedList<List>();
   - LinkedList<List> myL = new LinkedList<ArrayList>(); ⇐
   - LinkedList<? extends List> myL = new LinkedList<ArrayList>();
   - LinkedList<? super List> myL = new LinkedList<List>();
   - LinkedList<? super List> myL = new LinkedList<Collection>();
   - LinkedList<Collection> myL = new LinkedList<Collection>();

   myL.add(new ArrayList());

Except 3, with the arrow, all will compile. Description is in the final review discussion slides.

<? super List> could mean List and superclass of List

3. What are generic classes? Why do we use them? Clearly explain the advantage(s). Give an example generic class from the standard Java collection library.

→ Definition : a parameter that allows us to specify the type that should be accepted

A generic class seems to simply be a class with a type parameter section after it.

Ex. public class Thing<E> {}

→ Compile time check, instead of getting exception during runtime.

→ Advantages:

Elimination of casts

Stronger type checks at compile time.

→ Stack class, LinkedList class

C

.

1. Consider:

```
int a[3];
int *p = a;
```

The value stored at the memory address found in p is:

C: the value of the first element of the array

2. The array access a[3] can be rewritten as:

^wait, doesn't a[3] only initialize array to size of 3? So shouldn't the range be from 0-2?(Resolved: Question #2 is different from #1)

C. *(a + 3)

-> *a is at a[0]

*(a+3) move it over to a[3]

*(a) +3 add 3 to a[0]

3. Consider:

```
int a[] = { 10, 100, 100 };
int *p = a;
```

Question) The expression `p + 1`:

A. points to the second slot of a
B. Evaluates to 101
C. Points to the first slot of a
D. Evaluates to 11

Ans: A

4.) The string "dog" is stored as:

A. three non-contiguous memory locations
B. 4 contiguous locations in memory
C. a single location in memory
D. 3 contiguous locations in memory

Ans: B, (3 for each character and 1 for the null terminator)

Heaps.

1. For any given node at position i:

Its Left Child is at *[2*i+1]* if available.

Its Right Child is at *[2*i+2]* if available.

Its Parent Node is at *[(i - 1)/2]*if available.

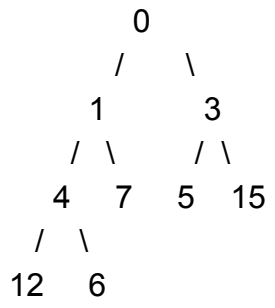
```
int deleteMin(int[] A, int size) {
	int removedElement = A[0];
	int lastElement = A[size - 1];
	A[0] = lastElement;
	A[size - 1] = null; // not necessary but for safety
	size--;
	trickleDown(A, size, 0);
	return removedElement; //supposed to return the smallest value
} // end deleteMin


void BubbleUp(int[] A, int size, int i) {
	int parentIndex = (i - 1) / 2;
	int toBubbleIndex = i;
	int temp;
	while (parentIndex != 0 && toBubbleIndex != 0) {
```

```
            if (A[toBubbleIndex] < A[parentIndex]) {
                  temp = A[toBubbleIndex];  // swap operation
                  A[toBubbleIndex] = A[parentIndex];
                  A[parentIndex] = temp;
            } else { break; }
            parentIndex = (i - 1) / 2 <--
            toBubbleIndex = parentIndex;
      }
} // end bubbleUp
```

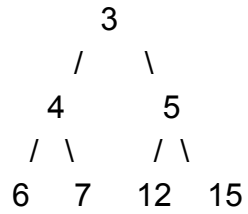
2. Give one reason why inserting a value into a d-heap containing N items might be faster than inserting a value into a binary heap of the same size.

Ans: when d > 2 with N items it is more likely to have fewer level than a binary heap given the height is log base d of N rather than log base 2 of N. insertion into a heap take generally big theta of height. When inserting a value, you insert at the end and then bubble up as needed. If we implement a d-heap, then there is less operations for Bubble up because the height of the heap is shorter than other heap that has 2 children per node.

3. Draw the binary min-heap that results from inserting 12, 1, 3, 7, 4, 5, 15, 0, 6 in that order into an initially empty binary heap.
Ans: 0,1,3,4,7,5,15,12,6



4. Draw the result of doing 2 delete mins on the heap you created in problem 3.
Ans: 3,4,5,6,7,12,15

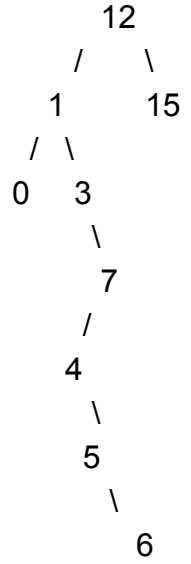


5. Give the worst possible case estimate for finding the maximum of a binary min-heap. Explain.

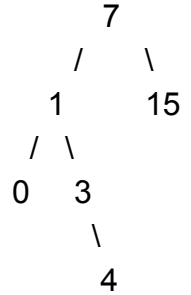
O(number of leaves) loop through the leaves compare to find max
Number of leaves in heap is ceiling of N/2
Looping from the end of the array where heap ended to the size/ 2 index to find max
The big O will be O(N/2) = O(N)

BST

1. Draw the BST that results from inserting 12, 1, 3, 7, 4, 5, 15, 0, 6
in that order into an initially empty tree.



2)



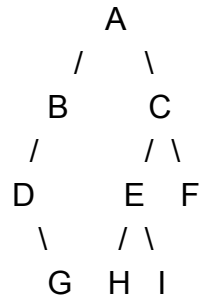
\
5
\
6 Alternative Ans: replace the root with the rightest left child

3)

Postorder: 0 5 4 6 3 1 15 7 (Left, Right, Print) yeah u right x2
Preorder: 7 1 0 3 6 4 5 15 (Print, Left, Right) yes
Inorder: 0 1 3 4 5 6 7 15 (Left, Print, Right)

If 15 is your root: (I think)
Postorder: 0, 6, 5, 4, 7, 3, 1, 15
Preorder: 15, 1, 0, 3, 7, 4, 5, 6
Inorder: 0, 1, 3, 4, 5, 6, 7, 15
Yeah Print = Root I just checked the lecture slides with it

)



HashTables

Consider a set of objects with hash codes 40, 5, 18, 29, 35, 7 and a hashTable of size 11

1a)

		35			5		40 -> 18-> 29			

							-> 7			

1b)

		35			5		40	18	29	7

1c)

29	7	35			5		40	18		

2. C. Two entries with different keys have the same exact hash value.

3.

Convert word “coffee” into an integer (hash table index) using Horner’s method. (Use base = 32).

c = 3, o = 15, f = 6, e = 5
Coffee: length of string = 6
key coffee = (3 * 32^5) + (15* 32^4) + (6* 32^ 3) +(6* 32^ 2) +(5*32^1) + (5*32^0)
Key (Horner's method) = ((((3 * 32 + 15)*32 + 6)*32 + 6)*32 + 5)*32 + 5
Index of coffee = key mod table size
Do we have to know the horner’s method in the final? yes

Coding

3.
```
int sameStack(StackLi s1, StackList2){
    Int same=0;
          if(s1.isEmpty() && s2.isEmpty()){
            return 1;
        }
    if (s1.isEmpty() || S2.isEmpty()){
      Return 0;
    }

          Int s1v = s1.pop();
    Int s2v = s2.pop();
        If ( s1v != s2v)
           Return 0;
```

```
        }
         else{
          same= sameStack(s1, s2);
          }
     push(s1, s1v); // for not destroying the stack push after
recursive call
     push(s2, s2v); // same as above (without these two line the
method still work but
                                  // Will destroyed the stack)
     Return same;
      }
```

You recurse before you push 1 back in.

Yes, so since we're popping to the last element in the stack and pushing the elements back in last to first we'll still have the same order as before

Write a recursive Java method that compare the two stack and see if they are equal

Equal is defined as have same items in same order

Sorting

1. It's value must be chosen randomly. Also, it's value would best around the average of the values in the entire list (if that makes any sense…it's also not that easily controllable)
    a. I think median is preferred.
    b. Yeah but that takes a while to find. Although I guess in this case that kind of thing doesn't matter.
    c. Don't pick the minimum or max element as the pivot. If it is the min or max, it will be O(n^2). It is best to pick a random element or the middle. According to Marina, it is best not to pick the first element because in a lot of cases, in many cases, the first element is the minimum or max.
    d. median-of-3 method

2. a) 100 (linear search)
   and c) 7 (log base 2 of 100 ceiling)
3. b) n-1
4. c) comparison sorts
5. A. The algorithm might be either selectionsort or insertionsort.

    a. Only if you start from the back. (for selection sort?)
        i. Couldnt you assume it started like how it is shown since selection sort has n^2 complex. in all cases?
6. d) O(n^2)
7. A. The pivot could be either the 7 or the 9.
8. E. None of the above. Wait isn't BubbleSort and InsertionSort both O(n) in the best case? Which one should I choose then?
    a. All of the above (But SelectionSort is O(n^2)). I think if the array is sorted it will be O(n) because no matter what it will iterate through the whole array at least 1 time
    b. I agree that both bubble sort and insertion sort are O(n)
    c. I don't think the best case for bubble sort is O(n). Since bubblesort knows nothing about the array, its best and worst case are both O(n^2). Similarly, selection sort knows nothing about the rest of the array, so no matter what it will keep searching for the minimum (maximum) value in the unsorted part, resulting in a running time of O(n^2). Insertion sort knows that the part of the array before the current element is sorted, so if the array is originally sorted, the running time will be O(n). So the answer is (a).

    Dissent for BubbleSort: BubbleSort looks at consecutive elements. If everything is already in order, then each comparison does not end up with swaps, which means that it moves on to the next set. This means that the algorithm never moves backward to the start. So it runs in O(n)

Miscellaneous

1) B) Hash Table O(1) insert and access

2) Abstract Data type, a way of storing, accessing manipulating data that gives a programmer a basic outlines but allows them to implement it in

any way they want. ADT uses data structures. (I thought data structures implement ADTs though???) Stack with a LL. ADT is what a blueprint of data structure **Data Structure** is an implementation of ADT

3) Stacks add and remove from the same location in a LL (usually from the head), while Queues add from one place and remove from another (which is why head/tail are both necessary).
4) If given index, use array. V. Hashtable

VV This stuff down here is messy af VV
(clarification of the term Data Structure)
According to the slides, array and linked list are the only data structure, right?
Yeah true, true.

But than she also mentioned that the term is abused like the big-O notation.

Correct me if I'm wrong but isn't data structure anything that store data ?
Including container, collection , ADT, array, linked list, hash table, trees , graphs and all? I'm now confused by the person who wrote the answer for the last 4 question.
Uh...yes??? ADT's aren't things that can be stored though...same with graphs too I think I wrote the little a) part
3 is correct though
+But isn't graph more like the concept of heap and bst ?
So from what I got from searching data structures, "data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate way
So hashtable, stacks, queues, and trees should fit into this, and only the hashtable fits the requirements for question 4 anyway

4 is probably referring to the fast access when given the index. So if index is not given it will be none of above, if index is given, it should be array.

The underlined part of 2 is confusing

Can someone explain this please?
Is it because 43 should be on the left of 47 instead of the left of 68?

Yes the structure of the BST should remain valid. You can try drawing the BST for c then the BST itself is wrong.

Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences CANNOT be the sequence of nodes examined?
a)
{10, 75, 64, 43, 60, 57, 55}
b)
{90, 12, 68, 34, 62, 45, 55}
c)
{9, 85, 47, 68, 43, 57, 55}
d)
{79, 14, 72, 56, 16, 53, 55}

Answer: c)

Good to see you guys make it here. Imma go to bed now.
Good luck all of u :)