



12 Pages

1. Please give the information requested in the boxes on the right.
2. Enter your name in the space provided to the right, fold over and seal down (using the label provided) to hide your name.
3. The answers must be legibly written. The Examiners can either ignore illegible script or have it typed and the expense of typing charged to the candidate.
4. Write your answers in ink.
5. Begin each answer on a fresh page. Do all rough work in the answer book and cross it through.
6. No part of this book is to be torn out; it must be handed in complete.
7. If more books than one are used, fasten them together using the tags available from the invigilator(s).
8. **Every book must be handed in. In no circumstances may answer books, used or unused, be taken out of the examination venue.**

1	Student Number	2306895
2	Title of Examination Paper	Advanced Software Development
3	Module Code	ACCB5011
4	Date and Time of Examination	11/01/2024 9am
5	Examination Venue	301b
6	Seat Number	
7	Number of Answer Books handed in	1

[illegible]

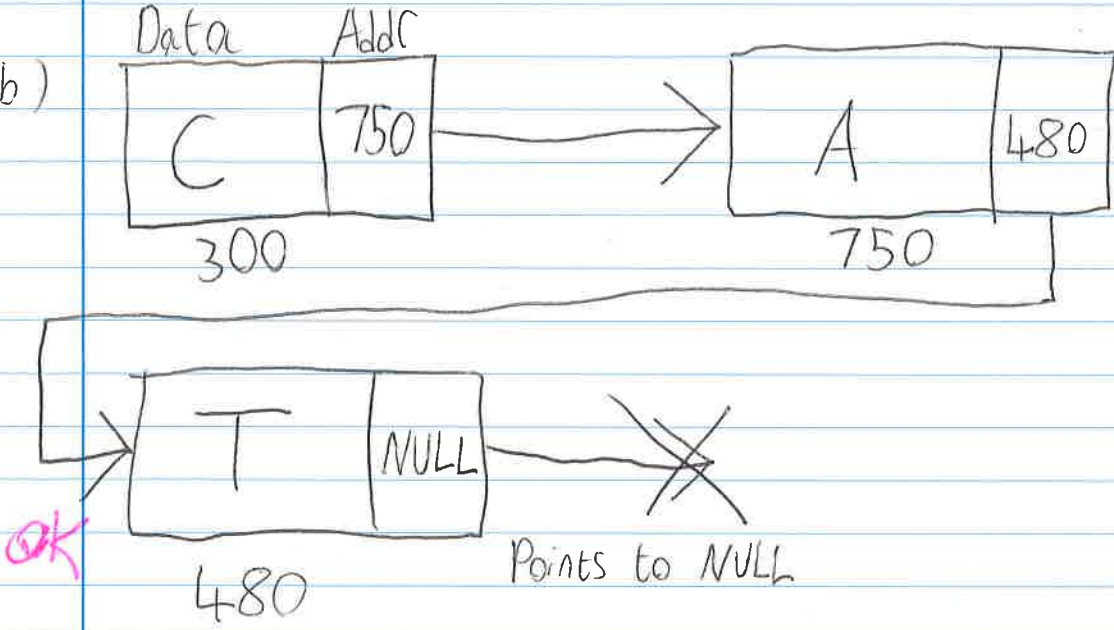
(Gweler yr ochr arall am y fersiwn Cymraeg)

**Complete, fold and seal
(see note 2 below)**

SURNAME: Roberts

FORENAME(S): Scott

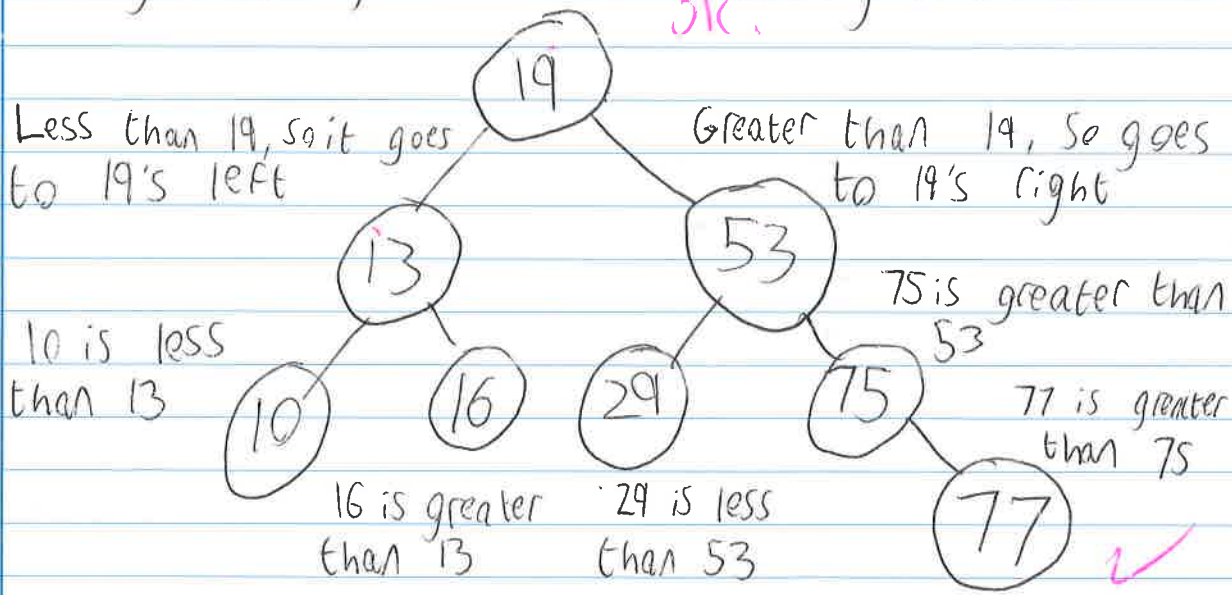
2b)



4a)

Sorted List = 10, 13, 16, 19, 29, 53, 75, 77
No of elements = 8
Middle element = 19

Selecting 19 as my root Node for balancing purposes



By splitting the sorted list into equal sub-lists and picking the middle (median) element, I have made my binary search tree balanced, making it faster to work on.

good.

4a = 6/6

Question Two

- a) Static memory allocation is where a program's memory is fixed and allocated at compile-time. Dynamic memory allocation is where memory is allocated at runtime on an as-needed basis. Data structures which are statically allocated cannot grow larger than the memory allocated to them, whereas data structures that are dynamically allocated can have more memory allocated to them and grow larger as the program runs. *points, e.g.s. diag? 3 1/4 / 4*
- b) A node based linked list is a data structure in which each data point (node) contains a piece of data and a link to another node in the list. Nodes can be placed anywhere in memory and do not have to be placed sequentially in memory. Essential operations include:
- Insertion – It must be possible to insert new nodes into the linked list, taking advantage of the fact that linked lists can be dynamically allocated (i.e. they can grow and shrink). It could be argued that if new nodes do not need to be inserted into a given linked list, a programmer would be better served using a statically allocated array.
 - Traversal – It must be possible to traverse a linked list. Unlike an array, it is not possible to jump to a given element by multiplying the element width by the distance from the start of the array, due to the fact that linked list nodes could be placed anywhere in memory. Therefore, we must traverse the linked list to find a given node where we do not know the address at which it is stored. *delete? 5 / 7*

See written diagram 2b *✓ OK.*

- c) Example of an algorithm for a function that will add an item to a double linked list:

```
Function AddToDoublyLinkedList(node* nodeToInsertAfter, char dataItem){
    nextNode = nodeToInsertAfter.nextNode
    newNodeToInsert = new Node(dataItem)
    newNodeToInsert.next = nextNode
    newNodeToInsert.previous = nodeToInsertAfter

    nodeToInsertAfter.next = newNodeToInsert
    nextNode.previous = newNodeToInsert
}
```

a def a diag 20 both? 7 / 10

This implementation does not take into account that nodeToInsertAfter could be the last node in the list. If this happens, the function will naively try to manipulate an invalid nextNode and the program will crash. This could be avoided by using a conditional statement to check whether nodeToInsertAfter.nextNode is NULL.

the tree. On average, the space complexity is $O(h)$ where h is the height of the tree. One potential problem with this algorithm is that very tall subtrees may overflow the function stack, causing a crash. It is important to consider balancing trees to reduce their height. ✓.

c)

Iteration No.	Stack Contents	Cumulative Output
1	F	F
2	FA	FA
3	FE	FAE
4	FEB	FAEB
5	FEC	FAEBC
6	FD	FAEB CD

Question 4

- a) Answered on paper
- b) The first node I visited in each traversal was 19
 Post-order traversal: 10,16,13,29,77,75,53,19 ✓
 In-order traversal: 10,13,16,19,29,53,75,77 ✓
 Pre-order traversal: 19,13,10,16,53,29,75,77 ✓
- c) As node 16 does not have any descendants, we can easily delete it from the tree without worrying about reconnecting child nodes to the parent node.

We would have to first find the address of 16's parent node:

careful with marking function

```

Function findParentNodePreorder(Node* node, int data){
    If (node == NULL){
        Return NULL;
    }
    If (node.left.data == data || node.right.data == data){
        Return node;
    }
    findParentNode(node.left);
    findParentNode(node.right);
}
  
```

Now that we know the address of 16's parent node, we free up the memory used by the node we want to delete and we remember to set the right pointer of the parent node to nullptr to ensure the program does not try to dereference a null pointer later on.

```

Function deleteRightChildWithNoDescendants(Node* node){
    delete node.right;
    node.right = nullptr;
}
  
```


- d) A dynamic linearly linked list can be modified to implement a circular (ring) data structure by setting the head of the list to point back to the tail (and vice versa in a doubly-linked list). An important thing to do when traversing a circular data structure is to keep track of the starting node to avoid infinite loops.

Question 3

- a) A stack data structure is one where the last element to be added is also the first element to be removed when retrieving data from it. This is called 'last in, first out' or LIFO. It is helpful to view a stack data structure like a literal stack of paper. You can only see the second sheet of paper by removing the first and so on. It is important to consider stacks when choosing a data structure, as they can be useful when making 'breadcrumb trails' of a user's operation such as undo history or browsing history. There are a number of essential operations when working with a stack data structure. One must be able to push to the stack, which is the process of adding new items to the top of the stack. One must be able to pop from the stack which is the process of taking items from the top of the stack. One must also be able to peek at the stack, which simply means to look at the item that is at the top of the stack.

A queue data structure is one where the first element to be added is also the first element to be removed when retrieving data from it. This is called 'first in, first out' or FIFO. A queue data structure represents a typical queue seen in real life. The first person to join a queue at a coffee shop is the first person to be served. It is important to consider including a queue when selecting a data structure, particularly in programs where multiple "jobs" are expected to be performed. For example, a programmer designing a web server would want to ensure that requests come through a queue, serving the first requests first rather than processing the most recently received request as soon as possible. There are a number of essential operations when working with a queue data structure. One must be able to enqueue items to the back of the queue and dequeue items from the front of the queue.

- b) An example of an algorithm that implements an iterative depth first search using a stack data structure

```
Function preorder(Node* node){  
    if (node == NULL){  
        // Base case  
        return NULL;  
    }  
    cout << node.data;  
    Preorder(node.left);  
    Preorder(node.right);  
}
```

This algorithm makes use of the function stack to recursively search a tree. It is a preorder search, so it first outputs the data stored in the node. The function then calls itself, passing the left subtree. The base iteration of the function will pause, and the new function call is added to the function stack. It will only resume once every consequent function call has finished executing. The function then calls itself again using the right subtree. In terms of both time and space complexity, the worst case is $O(n)$ where n is the number of nodes in

