Question 1. [26 MARKS]

The following are multiple choice questions. For each question, please circle an option that is correct. Note that if you circle an option that is not correct, you will **not be awarded** any marks for that question.

Part (a) [2 MARKS]

How are members in a structure variable stored in memory?

(i) Members are stored at random locations in memory

(ii) Members are stored at continuous addresses in memory

(iii) Members such as arrays are stored continuously and other non-array members are not

SOLUTION: **ii**

Part (b) [2 MARKS]

Consider the following definition of the structure `Node`:

```
struct Node{
    int m1;
    struct Node * mPtr;
};
```

Suppose that `ptr` is a pointer pointing to a valid `struct Node` variable which in turn has valid and accessible data stored in its members. Which one of the following statements is valid?

(i) `ptr->mPtr->m1;`

(ii) `ptr->mPtr.m1;`

(iii) `ptr.mPtr->m1;`

(iv) `*ptr.mPtr.m1;`

SOLUTION: **i**

Part (c) [2 MARKS]

Which one of the following is an equivalent representation of the decimal number 61?

(i) Base 16: 3D

(ii) Base 16: 3C

(iii) Base 2: 1101111

(iv) Base 2: 1111101

(v) Base 10: 60

SOLUTION: **i**

Part (d) [2 marks]
Suppose `int i = 61;` and `int j = 8;`. Which one of the following statements will clear bit 3 of `i`?

(i) `j=j>>1; i=i ^ j;`

(ii) `i=i ^ j`

(iii) `i=i && j;`

(iv) `i=i || j;`

Solution: **ii**

Part (e) [2 marks]
Suppose that $c_1 = 300n$, $c_2 = 3n^3/sqrt(n)$, $c_3 = log_5(n)$, $c_4 = 10000n^3$. Which one of the following sequences list the asymptotic complexity of these functions in increasing order?

(i) O(c_3), O(c_1), O(c_4), O(c_2)

(ii) O(c_3), O(c_1), O(c_2), O(c_4)

(iii) O(c_3), O(c_2), O(c_1), O(c_4)

(iv) O(c_1), O(c_3), O(c_2), O(c_4)

Solution: **ii**

Part (f) [2 marks]
Which statement is true of Abstract Data Types (ADTs)?

(i) Underlying implementation of an ADT is not hidden

(ii) Serve as building blocks

(iii) Are not portable

Solution: **ii**

Part (g) [2 marks]
Which statement is true of Linked Lists?

(i) Nodes in a linked list are stored continuously in memory

(ii) Linked lists grow dynamically

(iii) If the position of a node with a particular key is known beforehand, it is more efficient to use linked lists rather than arrays

(iv) It is possible to directly access nodes located in the middle of the linked list

Solution: **ii**

Part (h) [2 MARKS]
Which statement is true of Queues?

(i) Nodes are added to the front of the queue and removed from the back of the queue

(ii) Nodes can be added directly into the middle of the queue

(iii) Nodes are added to the back of the queue and removed from the front of the queue

(iv) Queues can be implemented via circular arrays as the front node remains static

SOLUTION: **iii**

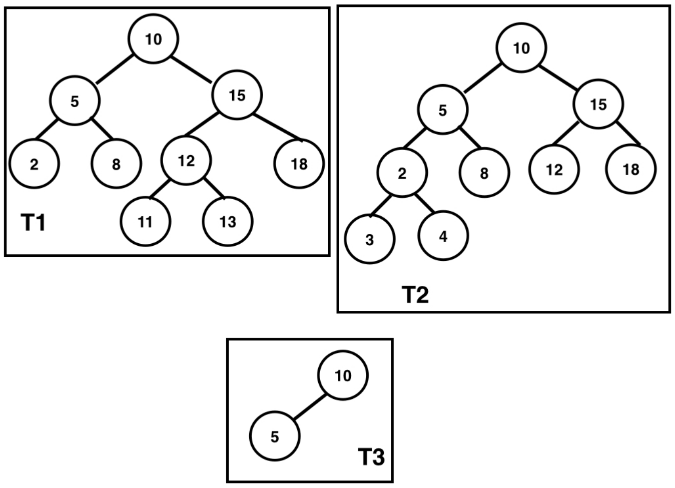
Part (i) [2 MARKS]
Which statement is true of Stacks?

(i) Stacks are non-linear data structures

(ii) Items are removed from the bottom of the stack all the time

(iii) Stacks are ideal for processing postponed obligations

SOLUTION: **iii**



Consider the above figure for the next five questions.

Part (j) [2 MARKS]
The following statement is true of T1:

(i) It is complete and is a heap

(ii) It is full and is a binary search tree

(iii) The tree is complete and is a binary search tree

(iv) The tree is not full but is a heap

Solution: **ii**

Part (k) [2 marks]

The following statement is true of T2:

(i) It is complete and full

(ii) It is full and is a binary search tree

(iii) The tree is a heap

(iv) The tree is a binary search tree and is complete

Solution: **i**

Part (l) [2 marks]

The following statement is true of T3:

(i) It is complete and is a heap

(ii) It is full and is a binary search tree

(iii) The tree is complete and is not a binary search tree

(iv) The tree is full and is a heap

Solution: **i**

Part (m) [2 marks]

Post-order traversal of T1 results in the sequence:

(i) 2 5 8 10 11 12 13 15 18

(ii) 10 5 15 2 8 12 18 11 13

(iii) 2 8 5 11 13 12 18 15 10

(iv) 2 5 8 10 12 15 18 11 13

Solution: **iii**

Question 2. [20 marks]

Following is an implementation of a *search* function. This algorithm searches for a specific value or key K in an already sorted array A. Array A contains n integer elements in increasing order. L and R are integer values that indicate the left index and right index of a sub-array of A. Once the key is found, this function will return the index at which the key resides. If the key is not found, then the function returns -1.

```
int search(int *A, int L, int R, int K){
    int middle, u=R-L;
    if (u>=0) {
        middle=(L+R)/2;
        if (A[middle]==K) {
            return middle;
        }
        else if(A[middle]>K){
            return search(A,L,middle-1,K);
        }
        else{
            return search(A,middle+1,R,K);
        }
    }
    return -1;
}
```

An example of an initial call to this function will be search(A,0,5,3) where A={0,1,3,4,5,6}. This function call will return 2 as key 3 is located at index 2 in array A. This search algorithm divides the array into two. Since the array is already sorted, the algorithm will know in which half the key being searched for resides. If the element located at the middle of the array is lesser than K, the key must reside in the right half of the array. Otherwise K resides in the left half. Please turn the page over for the first question regarding the above code snippet.

Part (a) [4 MARKS]

Following is a recurrent relation that defines the worst case cost of the search function implementation provided in the previous page.

$$\begin{aligned} f(n) &= \alpha c + f(n/2) \\ f(0) &= \beta c \end{aligned}$$

Assume that the cost of executing every declaration, assignment, return and conditional statement is a constant `c`. α and β are other numerical constants that are greater than 0. **Justify** why the above recurrent relation captures the worst case cost of the search algorithm.

SAMPLE SOLUTION:

Answer: n refers to the size of the array being searched. In the worst case, the key may not be found. What this means is that the search will not terminate before exploring all n elements. The algorithm will terminate when the indices are undefined. This base case occurs when there are no more elements left to be explored. Every non-recursive statement costs c, since there are α such statements in the non-base case, the cost of executing these is αc. In the base case, these statements occur β times and therefore the cost of executing the base case is βc. The recursive call divides the array explored into half and occurs only once in the function `search`.

Part (b) [6 MARKS]

Unroll the above relation and express the cost $f(n)$ in its non-recurrent form.

SAMPLE SOLUTION:

$$\begin{aligned} f(n) &= \alpha c + f(n/2) = k\alpha c + f(n/2^k) \\ &\textbf{as } n/2^k = 1 \text{ when } 2^k = n \text{ solving for } k,\ k = \lfloor log_2 n \rfloor \\ f(n) &= \lfloor log_2 n \rfloor \alpha c + f(1) \\ &= \lfloor log_2 n \rfloor \alpha c + \alpha c + f(0) \\ &= \lfloor log_2 n \rfloor \alpha c + \alpha c + \beta c \\ &= v \lfloor log_2 n \rfloor + u \end{aligned}$$

where $v = \alpha c$ and $u = \alpha c + \beta c$.

Part (c) [10 MARKS]
What is the asymptotic complexity f(n)=O(g(n)) of this search algorithm? Provide a formal proof.
SAMPLE SOLUTION: Need to find a $g(n)$ that satisfies the following definition:

O-Notation: $f(n) \in O(g(n))$ if there exist two positive constants K and n_0 such that

$$|f(n)| \leq K|g(n)| \;\forall\; n \geq n_0$$

Since

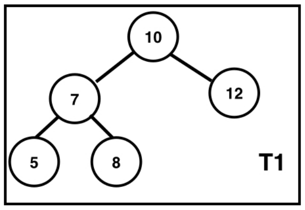
$$\begin{aligned} \lfloor log_2 n \rfloor &\leq log_2 n \;\forall\; n \geq 2 \\ u &\leq u log_2 n \;\forall\; n \geq 2 \\ v &\leq v log_2 n \;\forall\; n \geq 2 \end{aligned}$$

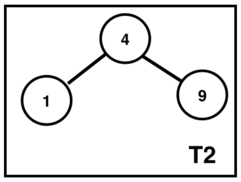
$$\begin{aligned} v \lfloor log_2 n \rfloor + u &\leq v log_2 n + u && \forall\; n \geq 2 \\ v \lfloor log_2 n \rfloor + u &\leq v log_2 n + u log_2 n && \forall\; n \geq 2 \\ v \lfloor log_2 n \rfloor + u &\leq (v+u) log_2 n && \forall\; n \geq 2 \\ v \lfloor log_2 n \rfloor + u &\leq \frac{(v+u)}{log 2} log(n) && \forall\; n \geq 2 \\ f(n) &\leq \frac{v+u}{log 2} log(n) \end{aligned}$$

where $K = \frac{v+u}{log2}$ and $n_0 = 2$

Question 3. [14 MARKS]

Suppose that you are given two pointers that point to the root nodes of two binary search trees. Each node in both trees stores an integer value as data/key. You are required to complete the implementation of function `void f(struct Node * r1, struct Node * r2)` which can print out all elements in both binary search trees from the largest to the smallest value. Following is an example of two binary search trees.





Your function should print the following values to the console: 12 10 9 8 7 5 4 1. The only condition imposed in your implementation is that you must use at least one stack. You can define helper functions if needed. Each node in the binary search tree is defined as:

```
struct Node{
    int data;
    struct Node * lChild;
    struct Node * rChild;
};
```

Assume that you have at your disposal the following interface functions (i.e. you do not have to implement these functions) and that prototypes of these functions and helper functions (if any) are listed in a header file that is available to you:

- struct Stack * initializeStack();
- void push(struct Stack * S, struct Node * n);
- struct Node * pop(struct Stack * S);
- int isEmpty(struct Stack * S);
- void freeStack(struct Stack * S);

The function `initalizeStack` dynamically allocates space for a `struct Stack` variable and returns a pointer to it. `push` inserts a `struct Node` pointer `n` into the stack `S` and `pop` performs a removal from the stack and returns a `struct Node` pointer `n`. `isEmpty` checks whether the stack `S` has any nodes (i.e. returns 1 if there are no nodes and 0 otherwise). `freeStack` deallocates all remaining elements in the stack pointed to by `s`.

Hints: You do not necessarily have to follow these tips if you are aware of an alternate approach. You can use two stacks. Each stack can represent each tree. You can traverse each tree using in-order traversal and store nodes encountered through this traversal in the stack corresponding to that tree. After both stacks are populated, values in nodes can be compared and printed accordingly.

Define helper functions here if needed:

Sample Solution:

```
void inOrder(struct Stack * s, struct Node * n){
    if (n!=NULL) {
        inOrder(s,n->lChild);
        push(s, n);
        inOrder(s,n->rChild);
    }
}

void f(struct Node * r1, struct Node * r2){
    struct Stack * s1 = initializeStack();
    struct Stack * s2 = initializeStack();
    struct Node * n1, * n2;
    int pFlag1=0, pFlag2=0;

    inOrder(s1, r1);
    inOrder(s2, r2);

    while (isEmpty(s1)!=1 && isEmpty(s2)!=1) {

        if (pFlag1==0) {
            n1=pop(s1);
            pFlag1=1;
        }
        if (pFlag2==0) {
            n2=pop(s2);
            pFlag2=1;
        }

        if (n2->data<n1->data) {
            printf("%d ", n1->data);
            n1=NULL;
            pFlag1=0;
        }
        else {
            printf("%d ", n2->data);
            n2=NULL;
            pFlag2=0;
        }
    }
    if (n1!=NULL) {
        printf("%d ",n1->data);
    }
    if (n2!=NULL) {
        printf("%d ",n2->data);
    }
    while (isEmpty(s1)==0) {
        n1=pop(s1);
        printf("%d ",n1->data);
    }
    while (isEmpty(s2)==0) {
        n2=pop(s2);
        printf("%d ",n2->data);
    }
    freeStack(s1);
    freeStack(s2);
}
```