NOTE TO STUDENTS: This file contains sample solutions to the term test. Please read the solutions carefully. Make sure that you understand why the solutions given here are correct, that you understand the mistakes that you made (if any), and that you understand *why* your mistakes were mistakes. Remember that although you may not agree completely with the marking scheme used to grade your paper, it was followed the same way for all students. We will remark your test only if you clearly demonstrate that there is a need for this. For all remark requests:

- Please submit your request directly to Noha Sinno (noha.sinno@alum.utoronto.ca)
- Your request must be a document which contains a clear description of why there is a need for remark and a scanned copy of the solution you had provided to the question in doubt
- Please remember that your question will be completely remarked and there is a chance that more marks maybe deducted

Question 1. [30 marks]

Please provide short answers to each one of the following sub-questions.

Part (a) [1 mark]

What is the purpose of function pointers?

Sample Solution:

> Function pointers store addresses of function handles. These are useful for generic implementation and abstraction.

Part (b) [2 marks]

Consider the following code snippet. What will be the result stored in the variable `res`. Specify your answer in decimal base.

```
int res;
int leftOpr = 13;
int rightOpr = 10;
res=leftOpr & (rightOpr << 1);
```

Sample Solution:

```
The result is 4.
```

Part (c) [3 marks]

Suppose that `struct Cart *c` is a pointer pointing to a valid structure variable in memory consisting of two members `int weight` and `struct Cart * nextCart` that are also populated with valid values. List at least three different ways in which you can access the weight of the `nextCart` member starting from `c`.

Sample Solution:

```
c->nextCart->weight;
*c.nextCart->weight;
*(*c.nextCart).weight;
*(c->nextCart).weight;
```

Part (d) [1 mark]

Consider the following code snippet in which the structure `Cart` has the same definition as in the previous question. Are there any issues with it? Why or why not?

```
struct Cart c;
struct Cart * cPtr=&c;
cPtr=(struct Cart *)malloc(sizeof(struct Cart));
free(cPtr);
```

Sample Solution:

> Nothing is wrong with the code. `c` is a local variable which will be removed from memory once this function completes execution. Hence, reassigning the pointer variable which originally pointed to the local variable will not cause any problems as reference to that variable is still available via the variable name `c`. The dynamically created variable is also freed immediately. Hence, there will be no issues associated with dangling pointers or memory leaks.

Part (e) [2 MARKS]

What is the purpose of asymptotic complexity analysis (big-oh)? When a problem is NP-hard, what does that imply?

SAMPLE SOLUTION:

> Complexity analysis provides a good idea about the amount of resourced requires especially when the problem is large. NP-hard means that the problem cannot be solved in deterministic time when n is large.

Part (f) [4 MARKS]

What are the advantages of using a linked list over arrays and vice versa? Include in your answer the complexity of insertion, deletion and accessing content in $O(.)$ notation and n where n is the number of elements already inserted.

SAMPLE SOLUTION:

> **Linked lists:** dynamic growth, no restriction on the size. Insertion is $O(n)$, deletion is $O(n)$ and accessing is $O(p)$ if you know it is located at the p^{th} node. $O(n)$ is also acceptable. **Arrays:** easier to access and but are fixed in size. Insertion is $O(n)$ or O(n-p) is the element is located at index p, deletion is $O(n-p)$ or $O(n)$ and accessing an element is $O(1)$ if you know it is located at the p^{th} index.

Part (g) [4 MARKS]

What are the differences and similarities between queues and stacks?

SAMPLE SOLUTION:

> **Similarities**: Queues and stacks are linear data structures. Operations are $O(1)$. **Differences**: Due to the different queueing policies (FIFO and LIFO), insertion and deletion operations are different.

The remaining questions are based on Fig. 1. Root in each tree is the top-most node.

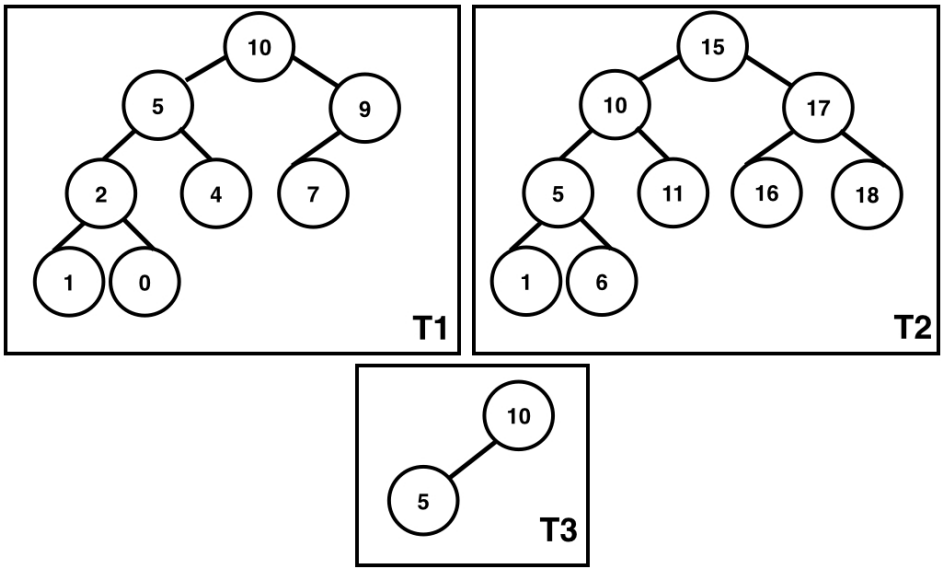

Figure 1: Tree diagrams

Part (h) [4 MARKS]

Consider T1 in this question:

- Is this a heap, binary search tree, neither or both? Why or why not?
- What is the height of this tree?
- Is this tree full, complete or neither? Why or why not?
- List the order in which elements will be printed when the tree is traversed in pre-order.

SAMPLE SOLUTION:

- Neither: Not a heap as the tree is not complete and not a BST as the elements are not inserted in the BST order.
- 3
- Neither: Not full as node 9 has one child. Not complete as level 2 is not completely filled with nodes.
- 10-5-2-1-0-4-9-7

Part (i) [5 MARKS]

Consider T2 in this question:

- Is this a heap, binary search tree, neither or both? Why or why not?
- What is the height of this tree?
- Is this tree full, complete or neither? Why or why not?
- List the order in which elements will be printed when the tree is traversed in in-order. Do you notice anything unique?

SAMPLE SOLUTION:

- BST: nodes are listed in the BST order. Not a heap.
- 3
- Full and complete: All nodes have 2 or 0 children. All levels are filled except for the last level and in the last level nodes are filled from left to right.
- 1-5-6-10-11-15-16-17-18. Printed in ascending order

Part (j) [4 MARKS]

Consider T3 in this question:

- Is this a heap, binary search tree, neither or both? Why or why not?
- What is the height of this tree?
- Is this tree full, complete or neither? Why or why not?
- List the order in which elements will be printed when the tree is traversed in level-order.

SAMPLE SOLUTION:

- BST and heap: Node 5 has a smaller value than the parent and this ordering heeds both the BST and heap ordering properties.
- 1
- Complete: Not full as node 10 has only one child. This is a complete tree as all levels (i.e. level 0) is filled with the maximal number of nodes possible (1 node) and in the last level there is only one node in the left most position.
- 10-15.

Question 2. [10 MARKS]

In this question, you will derive the computational complexity class $g(n)$ of the provided code snippet with respect to n. Assume that $n = 2^k$ where k is a positive integer. Do ensure that you include all steps and a formal proof to show that the cost $f(n)$ of executing this function indeed falls into the complexity class $g(n)$.

SAMPLE SOLUTION: $O(n)$. Solution is similar to Review Problem (1f).

Question 3. [10 MARKS]

Implement the `void pop(struct Stack * s, struct Data * d)` operation that can be performed on a stack using only queues. The `Stack` structure is defined as follows:

```
struct Stack{
   struct Queue * q1;
   struct Queue * q2;
   int count;
};
```

The `Data` structure is defined as follows:

```
struct Data{
   int data;
};
```

If the stack is empty, then set the `data` member of the content pointed to by `d` to -1. You can assume that you are provided with the definitions of the following interface functions:

- `void enqueue(struct Queue * q, struct Data d);`
  - This function inserts into the queue `q` a node containing the data `d`
- `void dequeue(struct Queue * q, struct Data * d);`
  - This function removes from the queue `q` a node and indirectly returns the content to the original function call via `d` which points to valid space in memory
- `int isEmpty(struct Queue * q);`
  - This function returns 1 if the `q` contains no nodes and 0 otherwise

You can assume that the arguments passed to the `pop` function are pointers to valid spaces in memory which are initialized appropriately. There are **two restrictions**. One is that you must use two queues and the other is that only the interface functions provided above must be used to access the queues (i.e. you must not directly access the content of the queues).

```
void pop(struct Stack * s, struct Data * d){
    struct Queue * tempQ;
    int numNodes=s->count;
    if (s->count!=0) {
        while (numNodes>1) {
            dequeue(s->q1, d);
            enqueue(s->q2,*d);
            numNodes--;
        }
        dequeue(s->q1, d);
        tempQ=s->q1;
        s->q1=s->q2;
        s->q2=tempQ;
        s->count--;
    }
    else{
        (*d).data=-1;
    }
}
```

Question 4. [10 MARKS]

In this question, you will implement the function `void convertBSTtoHeap(struct Node * bstRoot, struct Data * heapArray)` which copies data contained within each node in a Binary Search Tree (BST) (represented by the root pointer `bstRoot`) into an array represented by the single pointer `heapArray` so that the resulting array is a sequential representation of a heap. You can assume that `heapArray` is a dynamically allocated array containing sufficient space to store `MAXSIZE` number of `struct Data` variables and that `MAXSIZE` is much larger than the number of nodes in the BST. The `Node` and `Data` structures are defined as follows:

```
struct Node{
   struct Node * lChild;
   struct Node * rChild;
   Struct Data d;
};

struct Data{
   int data;
};
```

You can implement helper functions if necessary. Note that more than one heap can be constructed using the provided BST (i.e. there is no unique heap representation). Just ensure that the resulting sequential representation preserves the heap ordering and structural properties. Following is an example of a BST.

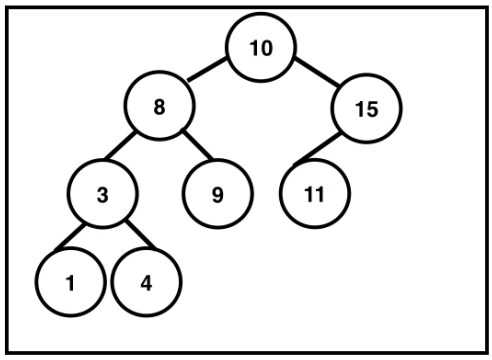

Figure 2: Tree diagrams

Hint: What tree traversal results in the listing of elements in a BST in ascending order? How can these elements be stored into the array so that this is a sequential representation of a heap?

SAMPLE SOLUTION:

```
void insertRevInOrder(struct Node * root, struct Data * hA, int * count)
{
    if (root!=NULL) {
        insertRevInOrder(root->rChild, hA, count);
        hA[*count]=root->d;
        *count=(*count)+1;
        printf("hello %d data %d\n", *count,root->d.data);
        insertRevInOrder(root->lChild, hA, count);
    }
```

```
}

void convertBSTtoHeap(struct Node * bstRoot, struct Data * hA)
{
    int count=1;
    insertRevInOrder(bstRoot, hA, &count);
}
```