1. True/False (2 pts each) To get credit, you must (very briefly) for your answers!!

(1.1) T/F The maximum number of nodes in a tree that has $N$ levels is $2^L - 1$.

(1.2) T/F A queue should be used when implementing Breadth First Search (BFS).

Using a queue allows us to "remember" the most immediate neighbors.

(1.3) T/F The largest value of a binary search tree is always stored at the root of the tree.

This is true for heaps only! Counter example:

(1.4) T/F A complete tree is also a full tree.

The leaf nodes of a complete tree are not at the same level.
(1.5) True. In a binary tree, every node has exactly two children.

(1.6) False. Tree operations typically run in $O(d)$ time where $d$ is the number of nodes in the tree.

$d$ is the height of the tree,

that is, $d = \log N$ where $N$ is the number of nodes.

(1.7) True. To delete a dynamically allocated tree, the best traversal method is postorder first, we delete the children, then the parent.

(1.8) False. In a heap, the left child of a node is always less than the right child of a node.

Counter-example

(1.9) True. Implementing a priority queue using heaps is more efficient than using linked lists.

$\begin{align*}
\text{Enqueue} & \quad \text{take } O(\log n) \\
\text{in the case of heaps} & \\
\text{Enqueue} & \quad \text{take } O(n) \\
\text{in the case of lists} & \\
\text{Dequeue} & \quad \text{take } O(1) \quad \text{in the case of lists} ...
\end{align*}$
(1.10) The ancestors of node 10 are nodes 11, 2, and 14.

(1.11) Every binary tree is either complete or full.

(1.12) Heaps are useful for searching binary trees efficiently.

(1.13) A complete directed graph with 8 vertices has 64 edges.

(1.14) The linked-list implementation of a graph is more efficient in finding whether two vertices are directly connected or not.
(1.15) The order in which elements are inserted in a binary search tree is unimportant. It is very important: if the elements are inserted in sorted order, then the height of the tree will become \(O(N)\).

2. Short answers (3 pts each)

(2.1) What is the number of nodes in a full tree with \(L\) levels? Prove it (show all the steps carefully).

\[
\begin{align*}
0 & \quad \text{levels} \quad N = 2^0 + 2^1 + \cdots + 2^{L-1} = \frac{2^L - 1}{2-1} = 2^L - 1 \\
\end{align*}
\]

(2.2) What is the maximum number of levels (height) of a tree with \(N\) nodes? What is the minimum number of levels (height) of a tree with \(N\) nodes? Justify your answers.

\[
\min \text{ height} = \log(N+1) \\
\text{tree is as compressed as possible...} \\
\text{max height}=N
\]

(2.3) Assume \(A\) is an array-based tree with 70 nodes. \(N=70\)

What is the index of the first leaf node? \(\text{Leaves: } \frac{N}{2} \text{ to } N-1\)

Who is the parent of \(A[50]\)? \(\left\lfloor \frac{50-1}{2} \right\rfloor = \left\lfloor \frac{49}{2} \right\rfloor = \left\lfloor 24 \right\rfloor \)
(2.4) We have discussed two different approaches to implement a priority queue. Which are these two approaches? How do they compare in terms of efficiency (time wise)? Justify your answer.

1st approach using heaps
- Enqueue: \(O(\log n)\) \(\approx O(\log n)\) on the average
- Dequeue: \(O(\log n)\) on the average

2nd approach using linked list
- Enqueue: \(O(n)\) worst case
- Dequeue: \(O(1)\) on the average

(2.5) Given the graph below, draw its adjacency matrix representation (store the vertices in alphabetical order)
(2.6) Using the graph in (2.5), is there a path from A to D? Demonstrate how Depth First Search (DFS) solves this problem (to get credit, you need to show all the steps clearly).

\[ A \quad E \quad C \quad D \]

Show the stack in each iteration.

(2.7) A graph can be represented using either an adjacency list or an adjacency matrix representation. Compare the two approaches (list their advantages/disadvantages).

<table>
<thead>
<tr>
<th>Matrix</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory: ( O(V+E)^2 \approx O(V^2) )</td>
<td>( O(V+E) )</td>
</tr>
<tr>
<td>Better for dense graphs</td>
<td>Better for sparse graphs</td>
</tr>
<tr>
<td>Easier to check if two vertices are connected</td>
<td>Easier to find the vertex adjacent to another vertex</td>
</tr>
</tbody>
</table>

(2.8) Label the following binary tree with numbers from the set \( \{6, 2, 9, 14, 13, 1, 8\} \) so that it is a legal binary search tree (choose the numbers in any order).

```
  13
 /   \
6     14
/   \
1     9
```

```
     22
```

```
      8
```

- 6 -
(2.9) Show how the tree in (2.8) would look like after each of the following operations:

(i) delete 22

(ii) insert 34 (use the tree from step (i))

(2.10) Given the tree shown below, show the order in which nodes in the tree are processed by preorder traversal.

1 2 4 5 7 3 6 8
(2.11) A priority queue is implemented as a heap. Show how the heap shown below would look like after each of the following operations:

(i) pq.Enqueue(38);

(ii) pq.Enqueue(102); (use the heap from step (i))

(2.12) Continue problem (2.11)

(iii) pq.Dequeue(x); What is the value of x? (use the heap from step (ii))

\[ x = 102 \]

(iv) pq.Dequeue(y); What is the value of y? (use the heap from step (iii))

\[ y = 66 \]
(2.13) Heaps are usually implemented using arrays. Why? (be specific). What property of heaps allow us to implement them using arrays?

- Save memory
- We can compute the parent of a node easily
- The children node can be computed easily too.

(2.14) Suppose \( N \) elements are inserted in order, from smallest to largest, into a binary search tree. Describe the efficiency of searching for an element in the tree in terms of Big-O notation.

This will create a linked-list

\( O(N) \) time

(2.15) Trace the function below and describe what it does.

```cpp
template<class ItemType>
int Mystery(TreeType<ItemType> *tree, int &n)
{
if(tree != NULL) {
    n++;
    Mystery(tree->left, n);
    Mystery(tree->right, n);
}
}
```
3. Code

(3.1) (10 pts) Write a function that returns the largest value in a binary search tree (full credit will be given only to the most efficient solutions).

```cpp
template <class Item_Type>
Item_Type Tree_Type<Item_Type>::Largest()
{
    return Largest_Value(root);
}

template <class Item_Type>
Item_Type Largest_Value(TreeNode<Item_Type>* tree)
{
    if (tree->right != NULL) // Assumes that the tree is not empty
    
        tree = tree->right;
        Largest_Value(tree);
    
    return tree->info;
}
(3.2) (10 pts) Write a boolean member function `IsBST` that determines if a binary tree is a binary search tree.

```cpp
bool TreeType<ItemType>::IsBST()
{
    return IsTrue(TreeNode<ItemType> * tree);
}

bool IsTrue(TreeNode<ItemType> * tree)
{
    if (tree == NULL)
        return true;
    else if (tree->left != NULL &&
             tree->left->info > tree->info)
        return false;
    else if (tree->right != NULL &&
             tree->right->info <= tree->info)
        return false;
    else
        return IsTrue(tree->left) && IsTrue(tree->right);
}
```
(3.3) (5 pts) Give the pseudo-code of the Depth-First-Search approach. How is it different from the Breadth-First-Search approach?

```plaintext
found = false;
Stack.Push(Start Vertex)
do
    Stack.Pop(vertex)
    if vertex == end Vertex
        found = true
    else
        Push all adjacent vertices onto Stack
while !Stack.IsEmpty() && !found
if (!found)
    Write "Path does not exist"
```

**BFS**

Use a queue instead of a stack!!
Look at all possible paths at the same depth before going to a deeper level!