Section-A,B,C,D **Subcode:18EC643**

1. Define a binary Tree and state and prove four properties of binary tree. Draw the binary expression trees for the following expressions

11.3 PROPERTIES OF BINARY TREES

Property 11.1 *The drawing of every binary tree with n elements, $n > 0$, has exactly $n - 1$ edges.*

Proof Every element in a binary tree (except the root) has exactly one parent. There is exactly one edge between each child and its parent. So the number of edges is $n - 1$. ■

Property 11.2 *A binary tree of height h, $h \geq 0$, has at least h and at most $2^h - 1$ elements in it.*

Proof Since each level has at least one element, the number of elements is at least h. As each element can have at most two children, the number of elements at level i is at most 2^{i-1}, $i > 0$. For $h = 0$, the total number of elements is 0, which equals $2^0 - 1$. For $h > 0$, the number of elements cannot exceed $\sum_{i=1}^{h} 2^{i-1} = 2^h - 1$. ■

Property 11.3 *The height of a binary tree that contains n, $n \geq 0$, elements is at most n and at least $\lceil \log_2(n+1) \rceil$.*

Proof Since there must be at least one element at each level, the height cannot exceed n. From Property 11.2 we know that a binary tree of height h can have no more than $2^h - 1$ elements. So $n \leq 2^h - 1$. Hence $h \geq \log_2(n+1)$. Since h is an integer, we get $h \geq \lceil \log_2(n+1) \rceil$. ■

A binary tree of height h that contains exactly $2^h - 1$ elements is called a **full binary tree**. The binary tree of Figure 11.5(a) is a full binary tree of height 3. The binary trees of Figures 11.5(b) and (c) are not full binary trees. Figure 11.6 shows a full binary tree of height 4.

i) (a * b) + (c/d) ii) (a + b)/(c – d) + e + g * h/a

2. Mention four common ways to traverse binary tree.Implement C++ functions for each traversal method.

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.
- **Search** − Searches an element in a tree.
- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

Inorder Traversal

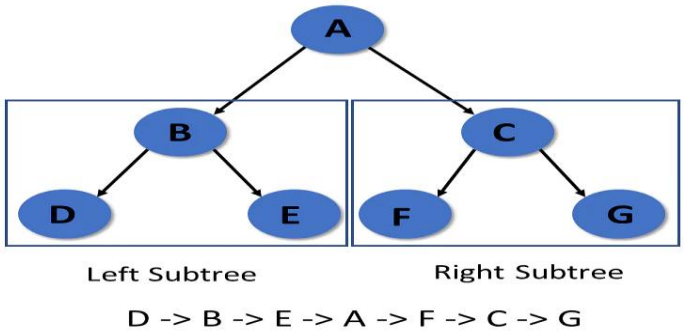
In the in-order traversal, the left subtree is visited first, then the root, and later the right subtree.

Algorithm:

Step 1- Recursively traverse the left subtree

Step 2- Visit root node

Step 3- Recursively traverse right subtree



Pre-Order Traversal

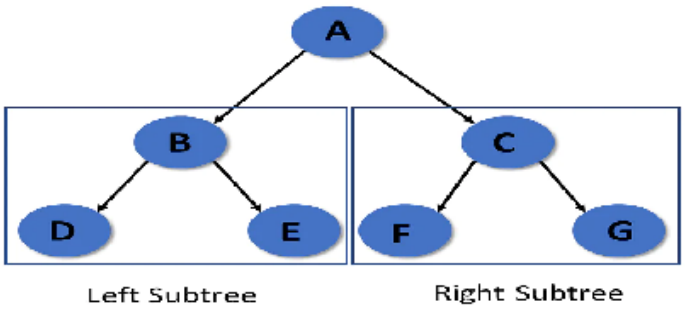
In pre-order traversal, it visits the root node first, then the left subtree, and lastly right subtree.

Algorithm:

Step 1- Visit root node

Step 2- Recursively traverse the left subtree

Step 3- Recursively traverse right subtree



A -> B -> D -> E -> C -> F -> G

Post-Order Traversal

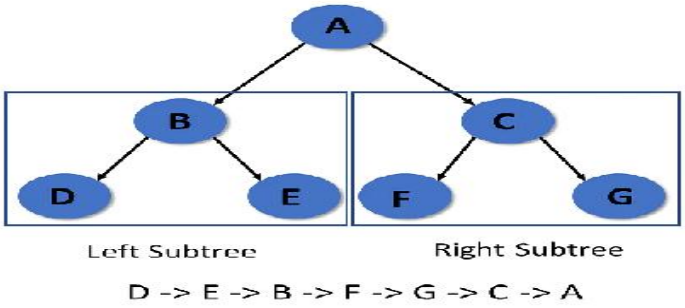
It visits the left subtree first in post-order traversal, then the right subtree, and finally the root node.

Algorithm:

Step 1- Recursively traverse the left subtree

Step 2- Visit root node

Step 3- Recursively traverse right subtree



D -> E -> B -> F -> G -> C -> A

Pre-order traversal

```
template <class T>
Void preOrder (binaryTreeNode <T> * t)
{
   if (t != NULL)
   {
      visit (t);   // visit tree root
      preOrder (t -> leftChild); // do left subtree
      preOrder (t -> rightChild); // do right subtree
   }
}
```

In order Traversal

```
template <class T>
Void inOrder (binaryTreeNode <T> * t)
{
   if (t != NULL)
   {
      inOrder (t -> leftChild); // do left subtree
      visit (t); // visit tree root
      inOrder (t -> rightChild); // do right subtree
   }
}
```

Post traversal.

```
template <class T>
Void postOrder(binaryTreeNode<T> * t)
{
   if (t != NULL)
   {
     postOrder (t -> leftchild);
     postOrder (t -> rightchild);
     visit(t);
   }
}
```

Visit function.

```
template <class T>
Void Visit (binaryTreeNode<T> * x)
{
   Cout << x -> element;
}
```

3. Develop a C++ template class to implement circular queue in array representation. Define member functions for push and pop operations.

4. Write and explain a C++ program that inputs a string and outputs the pair of matched parenthesis as well as those parenthesis for which there is no match.

In this problem we are to match the left and right parentheses in a character string. For example, the string (a*(b+c)+d) has left parentheses at positions 0 and 3 and right parentheses at positions 7 and 10. The left parenthesis at position 0 matches the right at position 10, while the left parenthesis at position 3 matches the right parenthesis at position 7. In the string (a+b))(, the right parenthesis at position

```
void printMatchedPairs(string expr)
{// Parenthesis matching.
   arrayStack<int> s;
   int length = (int) expr.size();

   // scan expression expr for ( and )
   for (int i = 0; i < length; i++)
      if (expr.at(i) == '(')
         s.push(i);
      else
         if (expr.at(i) == ')')
       try
         {// remove location of matching '(' from stack
            cout << s.top() << ' ' << i << endl;
            s.pop();  // unstack match
         }
```

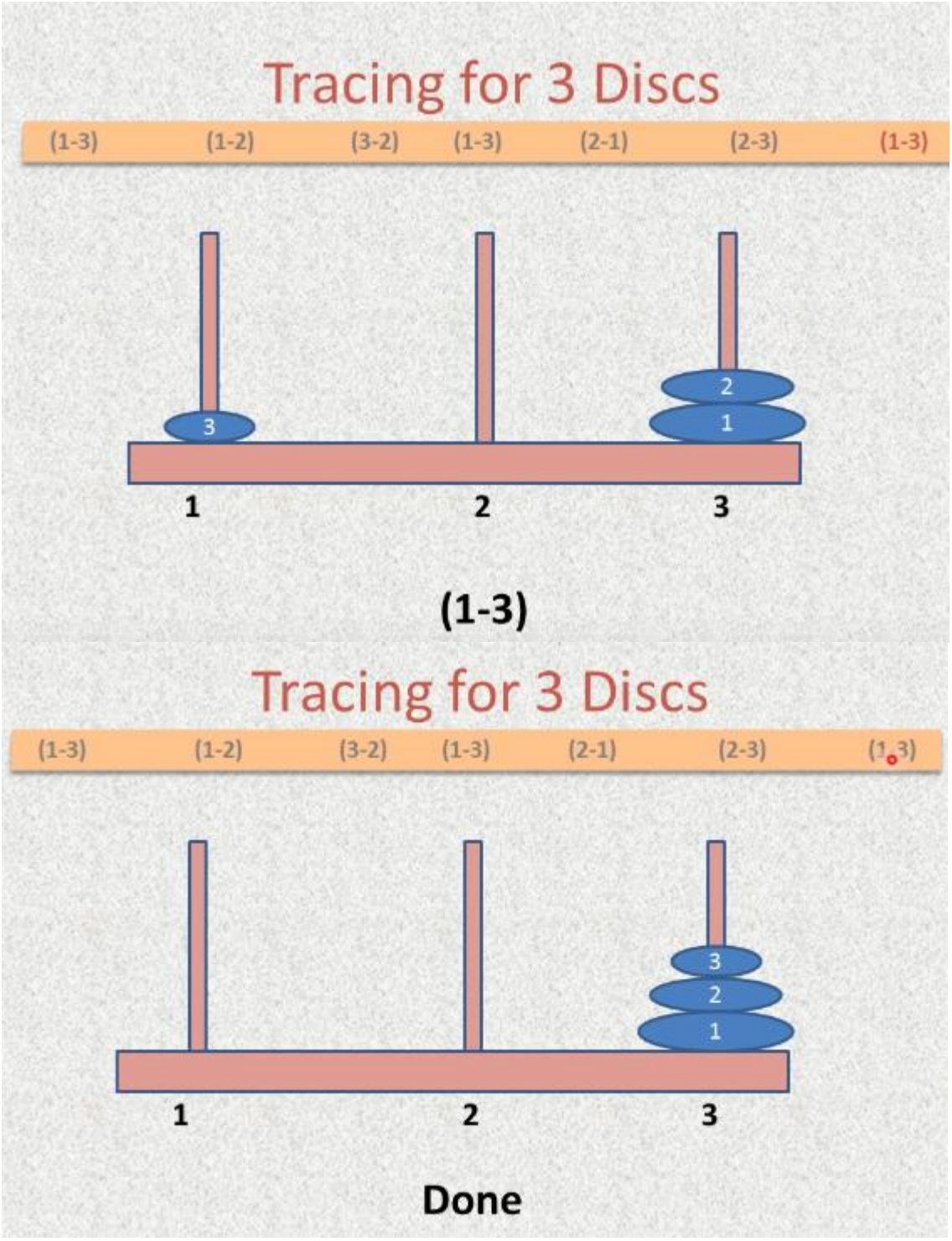
5. Explain stack application Tower Hanoi with neat diagrams and c++ program.

- Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
  No disk may be placed on top of a smaller disk.

Objective of Tower of Hanoi:

- The Objective of the puzzle is to move all the discs from one Rod (Source Rod) to another Rod (Destination Rod) with the help of third Rod (Auxiliary Rod) but they must follow the listed rules below.
- During single iteration only one disc can be moved,i.e. you cannot move more than one discs at a time.
- You cannot place a larger disc over a smaller disc.

```
void towersOfHanoi(int n, int x, int y, int z)
{// Move the top n disks from tower x to tower y.
 // Use tower z for intermediate storage.
   if (n > 0)
   {
      towersOfHanoi(n-1, x, z, y);
      cout << "Move top disk from tower " << x
           << " to top of tower " << y << endl;
      towersOfHanoi(n-1, z, y, x);
   }
}
```



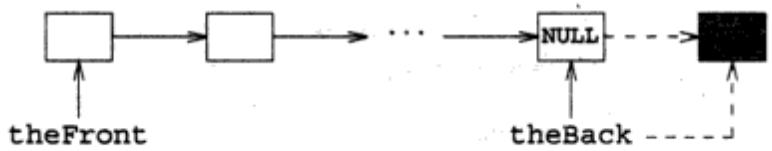
6. Draw the binary expression trees corresponding to each of the following expressions:

   (a) $(a+b)/(c-d)+e+g*h/a$

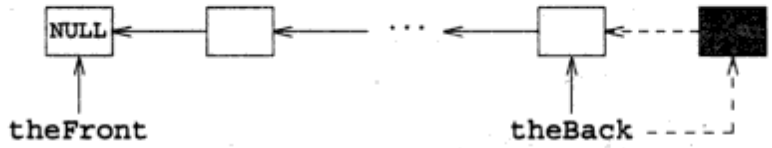
   (b) $-x-y*z+(a+b+c/d*e)$

   (c) $((a+b)>(c-e))||a<b\&\&(x<y||y>z)$

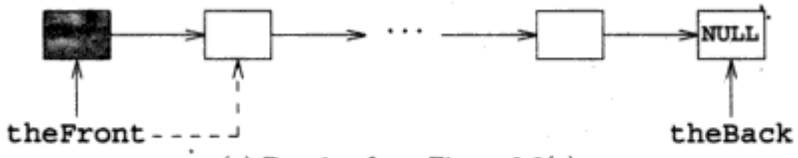
6. Write a method for 'push' and 'pop' for linked queue.



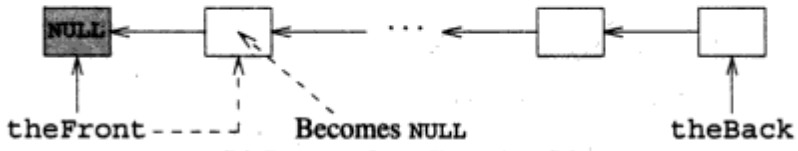
(a) Pushing into Figure 9.8(a)



(b) Pushing into Figure 9.8(b)



(a) Popping from Figure 9.8(a)



(b) Popping from Figure 9.8(b)

Figure 9.10 Popping an element from a linked queue

```
template<class T>
void linkedQueue<T>::push(const T& theElement)
{// Add theElement to back of queue.

   // create node for new element
   chainNode<T>* newNode = new chainNode<T>(theElement, NULL);

   // add new node to back of queue
   if (queueSize == 0)
      queueFront = newNode;        // queue empty
   else
      queueBack->next = newNode;  // queue not empty
   queueBack = newNode;

   queueSize++;
}
```

```
template<class T>
void linkedQueue<T>::pop()
{// Delete front element.
   if (queueFront == NULL)
      throw queueEmpty();

   chainNode<T>* nextNode = queueFront->next;
   delete queueFront;
   queueFront = nextNode;
   queueSize--;
}
```

7. Write a structure of binary tree node.

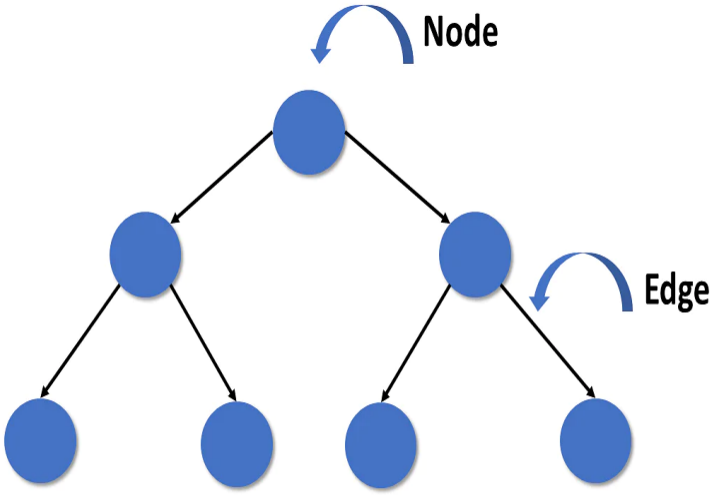
```
template <class T>
Struct binaryTreeNode
{
  T element;
  binaryTreeNode<T> *leftChild, * rightChild;
C1 binaryTreeNode() {leftChild = rightChild = NULL;}
C2 binaryTreeNode(const T& theElement)
   {
     element = theElement;
     leftChild = rightChild = NULL;
   }
C3 binaryTreeNode(const T& theElement, binaryTreeNode *theLeftChild,
                                       binaryTreeNode *theRightChild)
  { element = theElement;
    leftChild = theLeftChild;
    rightChild = theRightChild;
  }
};
```

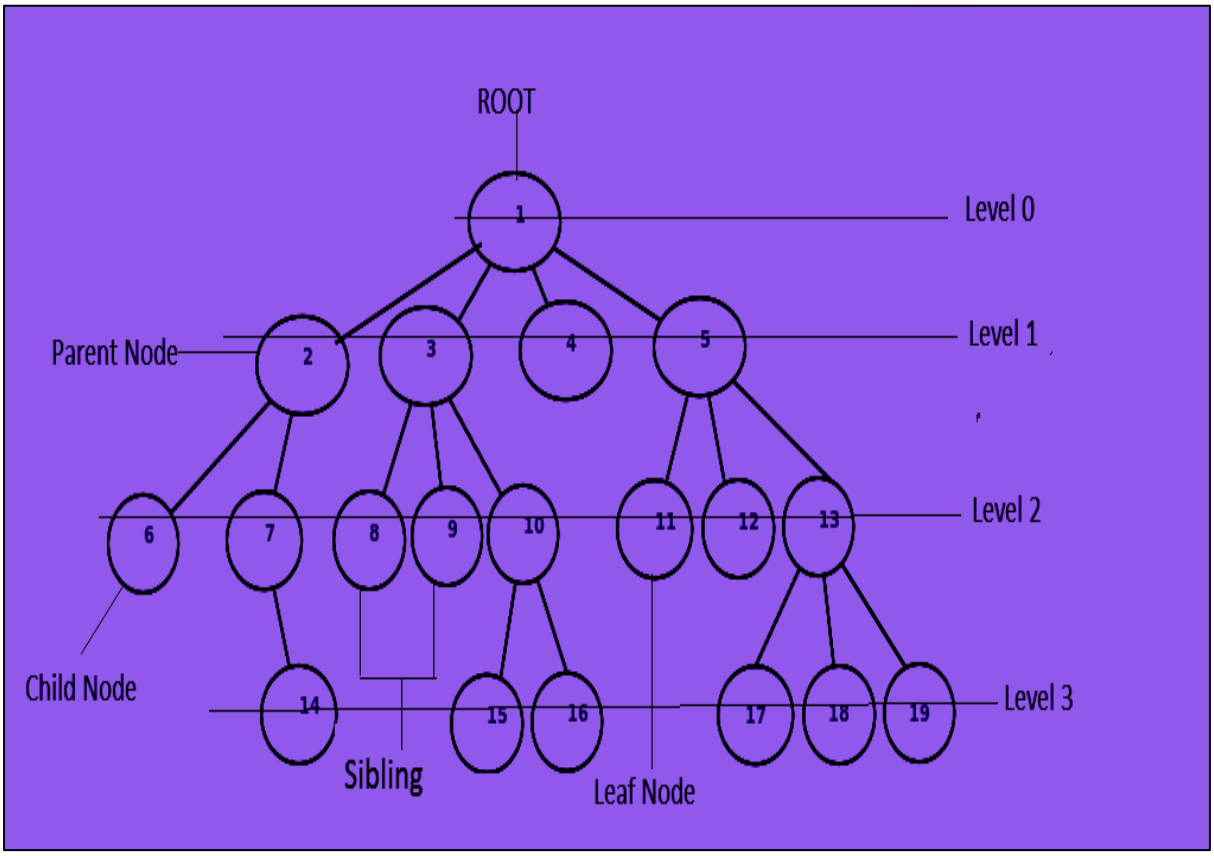
8. Define Heap. Explain insert operations using max.heap.

9. What is Heap data structure explains with flow diagrams.
   A heap is a complete binary tree in which the value of a node is less than all the values in its sub trees.
   By convention, the smallest element is the one with the highest priority

10. What is Tree explain types of tree explain terminology with neat diagrams.

The tree is a nonlinear hierarchical data structure and comprises a collection of entities known as nodes. It connects each node in the tree data structure using "edges", both directed and undirected.





Parent Node: The node which is a predecessor of a node is called the parent node of that node. {**2**} is the parent node of {**6, 7**}.

Child Node: The node which is the immediate successor of a node is called the child node of that node. Examples: {**6, 7**} are the child nodes of {**2**}.

Root Node: The topmost node of a tree or the node which does not have any parent node is called the root node. {**1}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree. **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree. The degree of the node {**3}** is **3**.
Leaf Node or External Node: The nodes which do not have any child nodes are called leaf nodes. {**6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19}** are the leaf nodes of the tree.
Ancestor of a Node: Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {**1, 2}** are the parent nodes of the node {**7}**
Descendant: Any successor node on the path from the leaf node to that node. {**7, 14}** are the descendants of the node. {**2}**.
Sibling: Children of the same parent node are called siblings. {**8, 9, 10}** are called siblings.
Depth of a node: The count of edges from the root to the node. Depth of node {**14}** is **3**.
Height of a node: The number of edges on the longest path from that node to a leaf. Height of node {**3}** is **2**.
Height of a tree: The height of a tree is the height of the root node i.e the count of edges from the root to the deepest node. The height of the above tree is **3**.
Level of a node: The count of edges on the path from the root node to that node. The root node has level **0**.
Internal node: A node with at least one child is called Internal Node.
Neighbour of a Node: Parent or child nodes of that node are called neighbors of that node.
Subtree: Any node of the tree along with its descendant

Terminology	Description
Root	Root is a special node in a tree. The entire tree originates from it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Leaf	Node which does not have any child is called as leaf
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.
Siblings	Nodes with the same parent are called Siblings.
Path / Traversing	Path is a number of successive edges from source node to destination node.
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.
Levels of node	Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on
Degree of Node	Degree of a node represents the number of children of a node.
Sub tree	Descendants of a node represent subtree.

11. Write push and pop operation of queue using an array

```
template<class T>
void arrayQueue<T>::push(const T& theElement)
{// Add theElement to queue.

   // increase array length if necessary
   if ((queueBack + 1) % arrayLength == queueFront)
   {// double array length
      // code to double array size comes here
   }

   // put theElement at the queueBack of the queue
   queueBack = (queueBack + 1) % arrayLength;
   queue[queueBack] = theElement;
}
```

```
void pop()
   {// remove queueFront element
      if (queueFront == queueBack)
         throw queueEmpty();
      queueFront = (queueFront + 1) % arrayLength;
      queue[queueFront].~T();  // destructor for T
   }
```

Draw the binary expression trees corresponding to each of the following expressions:

(a) $(a+b)/(c-d)+e+g*h/a$

(b) $-x-y*z+(a+b+c/d*e)$

(c) $((a+b)>(c-e))||a<b\&\&(x<y||y>z)$