Revision

[1] Big-O Analysis

AVERAGE(n)	**Statement**	no. of times executed
1. sum ← 0	1	1
2. i ← 0	2	1
3. **while** i < n	3	n+1
4. number ← input_number()	4	n
5. sum ← sum + number	5	n
6. i ← i + 1	6	n
7. mean ← sum / n	7	1
8. **return** mean	8	1

T(n) = 4n + 5, ***growth rate*** of n.

We use ***asymptotic notation***, such as big-Oh. So,T(n) = O(n), i.e., it is of O(n) **complexity**.

Compare algorithms: Linear search vs Binary search

Linear search

```
Set found to false.
Set position to -1.
Set index to 0.
While found is false and index < number of elements
If list [Index] is equal to search value
        found = true.
        position = index.
End If
Add 1 to index.
End While.
Return position
```

Binary search

```
Preconditions:  a is an array sorted in ascending order,
        first is the index of the first element to search,
        last is the index of the last element to search,
        target is the item to search for.
If first > last
        return failure
mid ← (first+last)/2
if a[mid] is equal to target
        return mid
else if target < a[mid]
  return result of recursive search of a from first up to mid-1
else
        return result of recursive search of a from mid+1 up to last
Postcondition: value returned is position of target in a, otherwise return failure
```

Example1- For an array a = {22, 11, 13, 81, 5}, trace linearSearch (a, 5,11).

a = {22, 11, 13, 81, 5}

- **linearSearch (a, 5,11)**
  - **n == 5; target == 11. Since a[4] != target, return linearSearch(a, 4, 11)**
- **linearSearch (a, 4,11)**
  - **n == 4; target == 11. Since a[3] != target, return linearSearch(a, 3, 11)**
- **linearSearch (a, 3, 11)**
  - **n == 3; target == 11. Since a[2] != target, return linearSearch(a, 2, 11)**
- **linearSearch(a, 2, 11)**
  - **n == 2; target == 11. Since a[l] == target, return 1**

Example2-For an array a = {22, 11, 13, 81, 5}, trace linearSearch (a, 5,50).

a = {22, 11, 13, 81, 5}

- **linearSearch (a, 5, 50)**
  - **n == 5; target == 50. Since a[4] != target, return linearSearch(a, 4, 50)**
- **linearSearch (a, 4,50)**
  - **n == 4; target == 50. Since a[3] != target, return linearSearch(a, 3, 50)**
- **linearSearch (a, 3, 50)**
  - **n == 3; target == 50. Since a[2] != target, return linearSearch(a, 2, 50)**
- **linearSearch (a, 2,50)**
  - **n == 2; target == 50. Since a[1] != target, return linearSearch(a, 1, 50)**
- **linearSearch (a, 1, 50)**
  - **n == 1; target == 50. Since a[0] != target, return linearSearch(a, 0, 50)**

1. **linearSearch (a, 0, 50)**
   - **n <= 0; return -1**

Example3- For an array a = {1, 6, 15, 22, 37, 45, 52}, trace binarySearch(a, 0, 6, 37).

- **binarySearch (a, 0, 6, 37)**
  first=0; last=6; target=37; mid=3; target > a[3], return binarySearch(a,4,6, 37)

- **binarySearch (a, 4, 6, 37)**
  first=4; last=6; target=37; mid=5; target < a[5], return binarySearch(a,4,4, 37)

- **binarySearch (a, 4, 4, 37)**
  first=4; last=4; target=37; mid=4; target == a[5], return 4

Example4- For an array a = {1, 6, 15, 22, 37, 45, 52}, trace binarySearch(a, 0, 6, 40).

- **binarySearch (a, 0, 6, 40)**

first=0; last=6; target=40; mid=3; target > a[3], return binarySearch(a,4,6, 40)

- **binarySearch (a, 4, 6, 40)**

first=4; last=6; target=40; mid=5; target < a[5], return binarySearch(a,4,4, 40)

- **binarySearch (a, 4, 4, 40)**

first=4; last=4; target=40; mid=4; target > a[4], return binarySearch(a,5,4, 40)

- **binarySearch (a, 5, 4, 40)**

first=4; last=4; target=40; first > last , return -1

What is the complexity of Linear and Binary Search?

Linear (Sequential, or serial) Search , if the size of array is n

- Best-Case: **O(1)** if item in [0]
- Worst-Case: **O(n)** if item in [n-1] or not found
- Average-Case **O(n/2)**

- Binary Search
  - average-case **O($\log_2$ n)**

[2*] Abstract data types (ADTs)

Example 1 Using Data Structures, write the **Abstract Data Type (ADT)** for Pair **of Dice** (**Characteristics** and **Operations**)

Dice ADT
Characteristics The value of the first die, a random integer value from 1 to 6, (dice_1), The value of the second die, a random integer value from 1 to 6 (dice_2), The total sum of the two readings (sum)
Operations int roll() randomize the values of the two dice and return the total sum int die_1() return the reading of the first die int die_2() return the reading of the second die

[3] Linked list ADT

What are the basic linked list operations.

- **appending a node,**
- **traversing the list,**
- **inserting a node,**
- **deleting a node, and**
- **destroying the list.**

What is the difference between appending and inserting a node to a list?

Append a node means adding it to the end of a list.

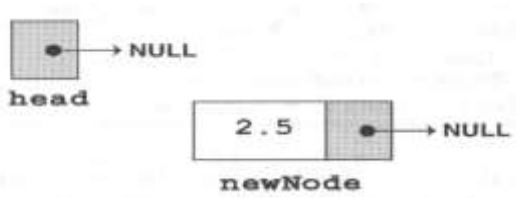
Create a new node.
Store data in the new node.
If there are no nodes In the list
  Make the new node the first node.
Else
  Traverse the list to find the last node.
  Add the new node to the end of the list.
End If.

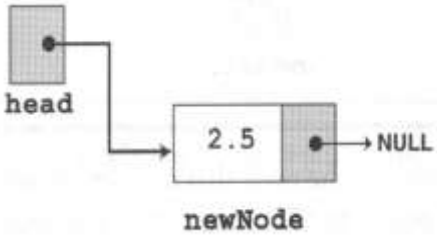
Insert a node means adding it to a list, but not necessarily to the end.

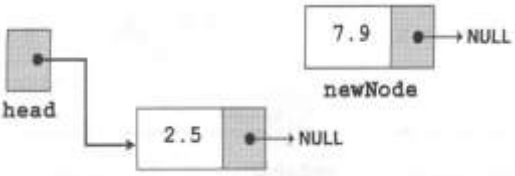
Create a new node.
Store data In the new node.
If there are no nodes In the list
  Make the new node the first node.
Else
Find the first node whose value is > or = the new value, or the end of the list
Insert the new node before the found node, or at the end of the list if not found.
End If.

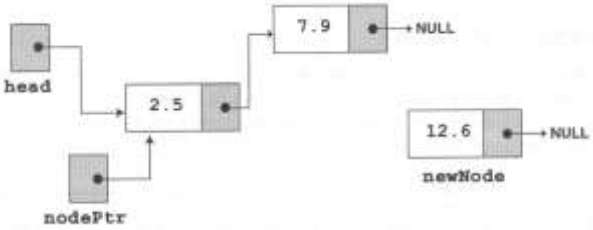
Trace following program to append 2.5, 9.7, and 12.5 to draw the form of linked list.

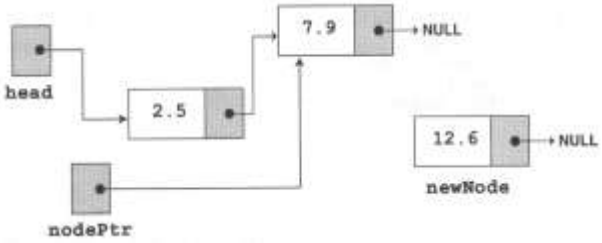
```
void main(void){
        FloatList list;
        list.appendNode(2.5);
        list.appendNode(7.9);
        list.appendNode(12.6);
}
```

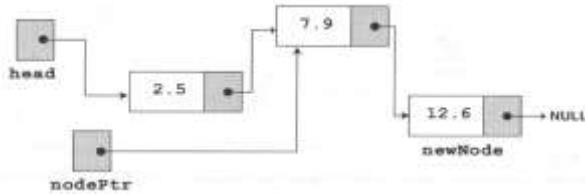






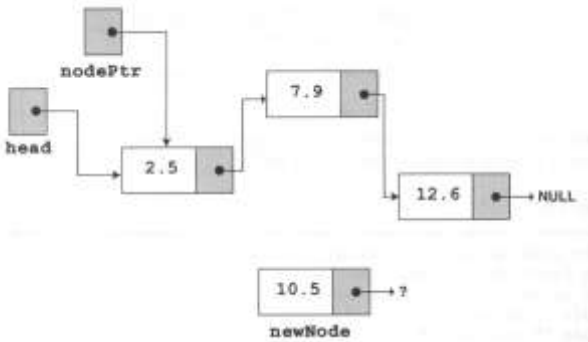


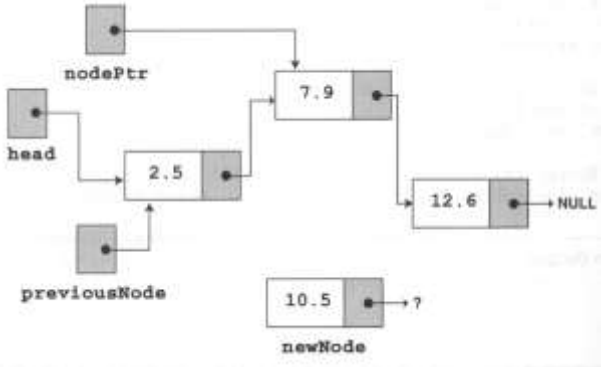


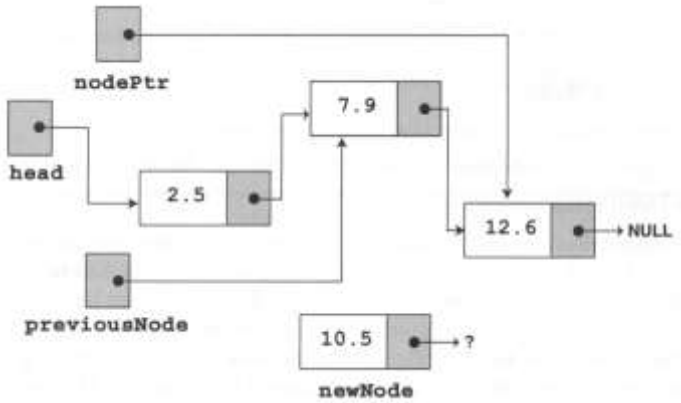


What is output of the following program and draw the corresponding linked list structure.

```
void main(void)
{
        FloatList list; // Build the list
        list.appendMode(2.5); list.appendNode(7.9); list.appendNode(12.6) ;
        list.insertNode(10.5); // Insert a node in the middle of the list.
        list.displayList();// Dispay the list
}
```







write the algorithms of **Traversing and Deleting a Node**

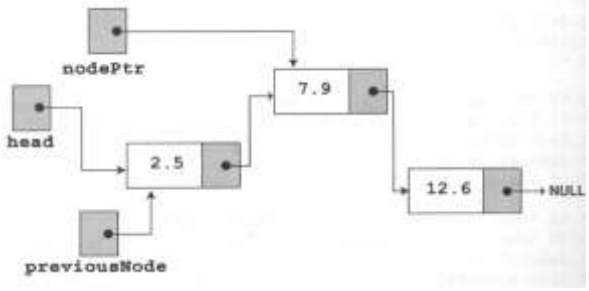
Traversing

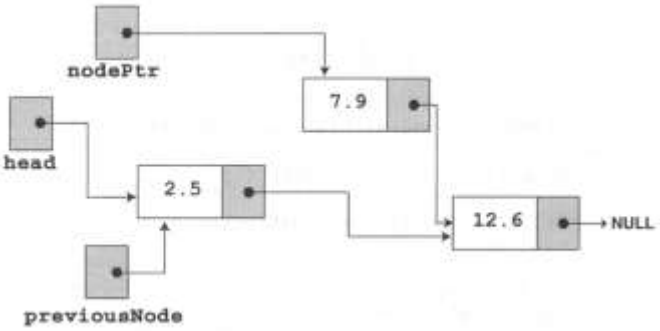
Assign List head to node pointer.
While node pointer is not NULL
    Display the value member of the node pointed to by node pointer.
    Assign node pointer to its own next member
End While.

Deleting a Node

- Remove the node from list without breaking the links created by the next pointers
- Deleting the node from memory (delete nodePtr)

And show **List.deleteNode(7.9);**





[4] STACK ADT:

A stack S stores items of some type in Last-In, First-Out (LIFO) order.

Basic Operations

PUSH	POP
1. If stack is not full then : 2. Add 1 to the stack pointer, 3. Store item at stack pointer location	1. If stack is not empty then : 2. read item at stack pointer location, 3. subtract 1 from the stack pointer

Example: Push 5, push 10, push 15, push 20, push 25, pop, pop, pop, pop

stack size =5 ,
top =-1

initially

	[0]
	[1]
	[2]
	[3]
	[4]

stack size =5 ,
top =0

Push 5

5	[0]
	[1]
	[2]
	[3]
	[4]

stack size =5 ,
top = 1

Push 10

5	[0]
10	[1]
	[2]
	[3]
	[4]

stack size =5 ,
top = 4

after all Push

5	[0]
10	[1]
15	[2]
20	[3]
25	[4]

stack size =5 ,
top = 3

pop

5	[0]
10	[1]
15	[2]
20	[3]
	[4]

Q2. Given an initially empty stack, the following operations are performed:
Push T, Push E, Push X, Pop, Push R, Push V, Pop, Push Y, Pop, Pop
(a) Write out the composition of the stack after these operations.
(b) What sequence of letters was popped off the stack? → XVYR

Reverse Polish Notation (RPN)

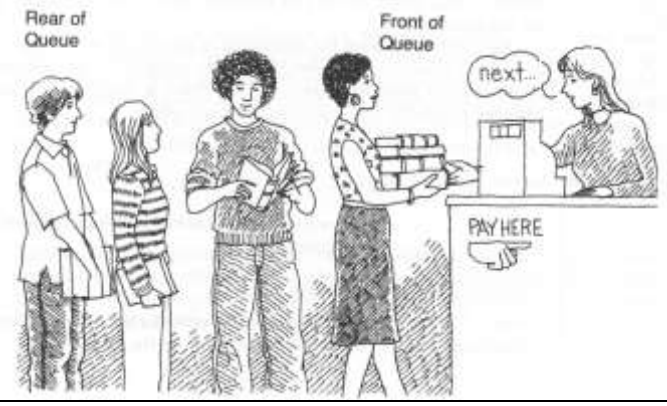
Stack ADT is used to evaluate a mathematical expression written in reverse Polish notation (RPN= postfix notation).

(A+B)*C → AB+C*
infix postfix

Notice that the postfix expression **HAS NO AMBIGUITY**, so it does not need parentheses to ensure that A+B is computed before the multiplication of C.

AB+C*	AB+C*	AB+C*	AB+C*	AB+C*
1	2 1	3	3 3	9
push A	push B	pop B pop A push A+B	push C	pop C pop A+B push (A+B)*C

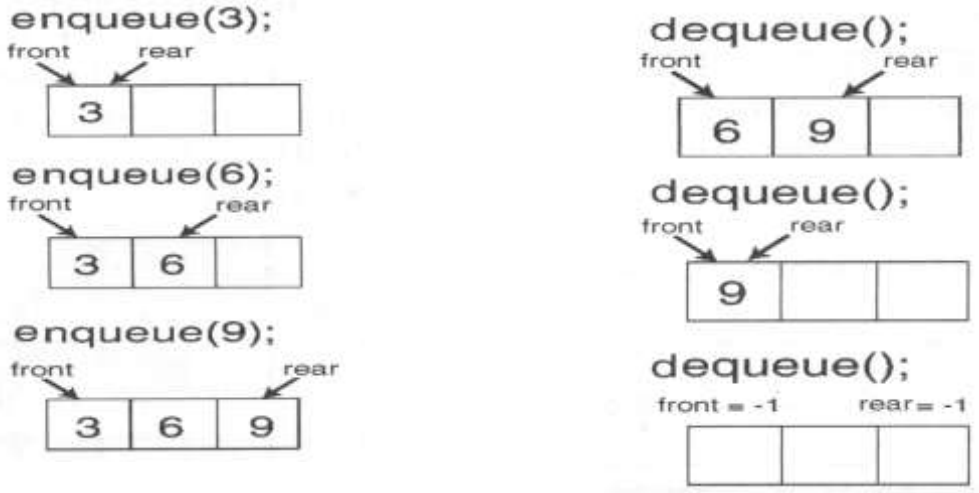
[4.5] QUEUE ADT :



Operations allowed on a queue:
- Creating an empty queue.
- Testing if a queue is empty.
- Adding data to the tail of the queue (enqueue).
- Removing data from the head of the queue (dequeue).

A Queue Q stores items of some type, with First-In, First-Out (FIFO) access

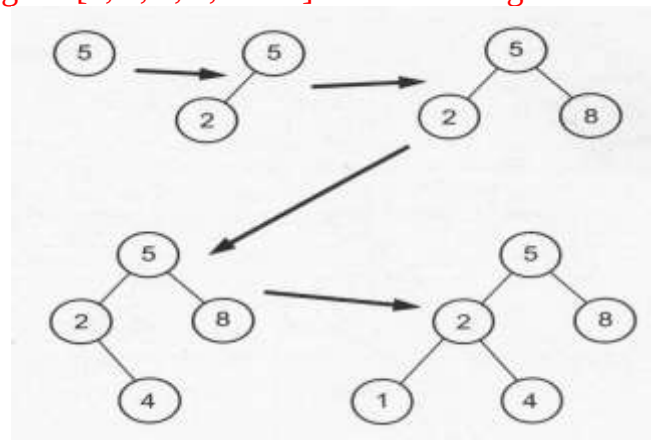
example: Identify the queue structure after execute the following enqueue operations.
enqueue(3),enqueue(6), enqueue(9), and three consecutive dequeue



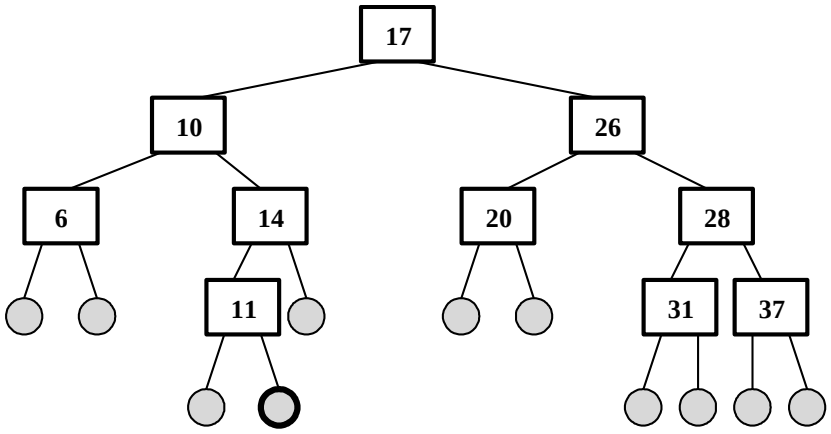
[5] Binary search trees (BST)

Basic operations
- *Inserting a new node.*
- *Deleting a node.*
- *Listing or visiting the nodes of the tree* (traversal)

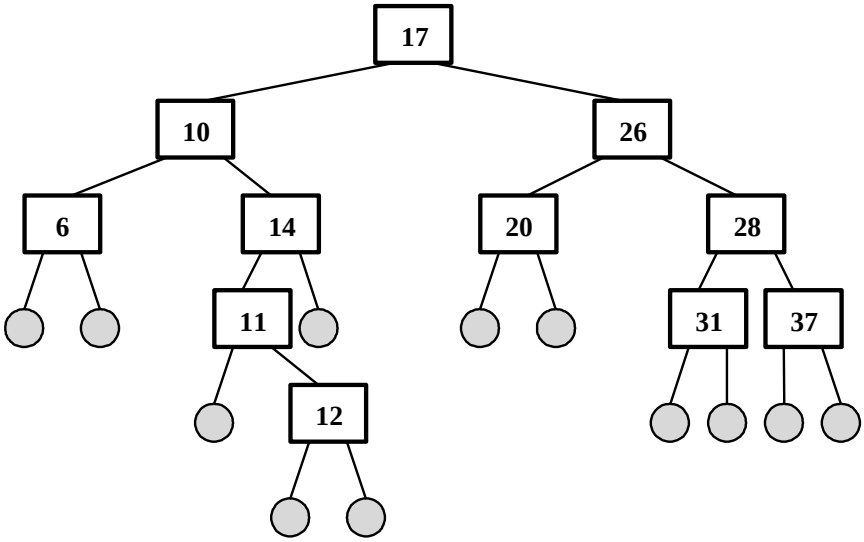
How to sort list of integers [5, 2, 8, 4, and 1] into ascending order using BST



How to insert 12 in the tree below



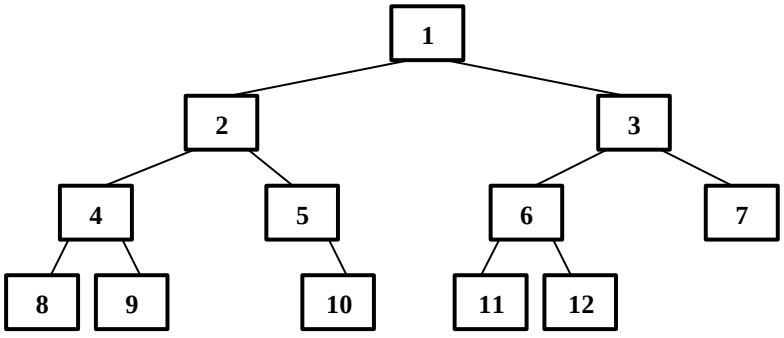
The Binary Search Tree after 12 inserted

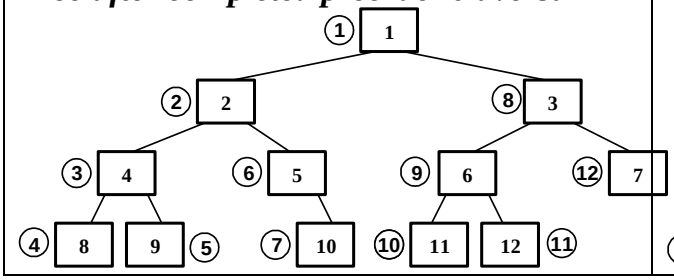
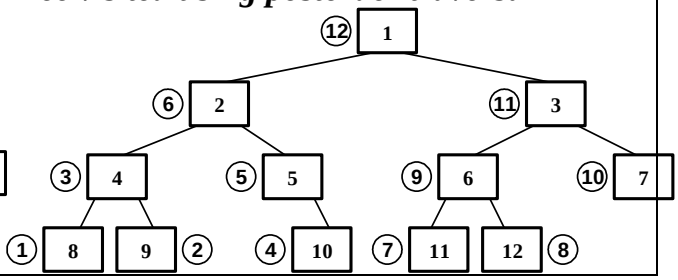


Traversal

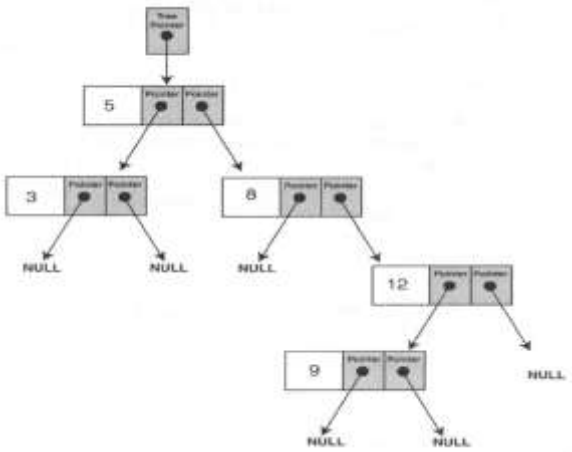
The 3 common methods for traversing a binary tree and processing the value of each node:

- ***Inorder traversal:***
  1. The node's left subtree is traversed.
  2. The node's data is processed.
  3. The node's right subtree is traversed
- ***Preorder traversal:***
  1. The node's data is processed
  2. The node's left subtree is traversed.
  3. The node's right subtree is traversed.
- ***Postorder traversal:***
  1. The node's left subtree is traversed.
  2. The node's right subtree is traversed.
  3. The node's data is processed.



Tree after completed preorder traversal	*Tree visited using postorder traversal*
	

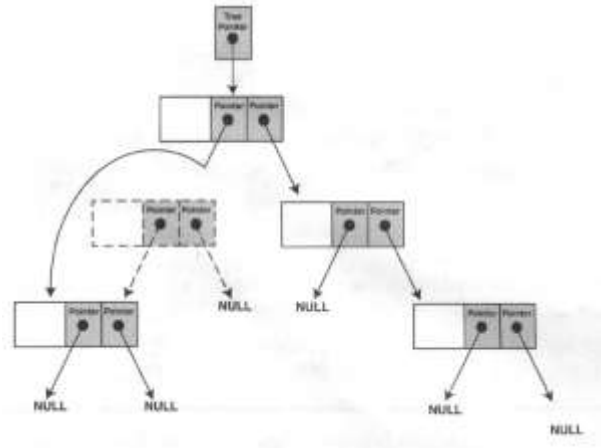
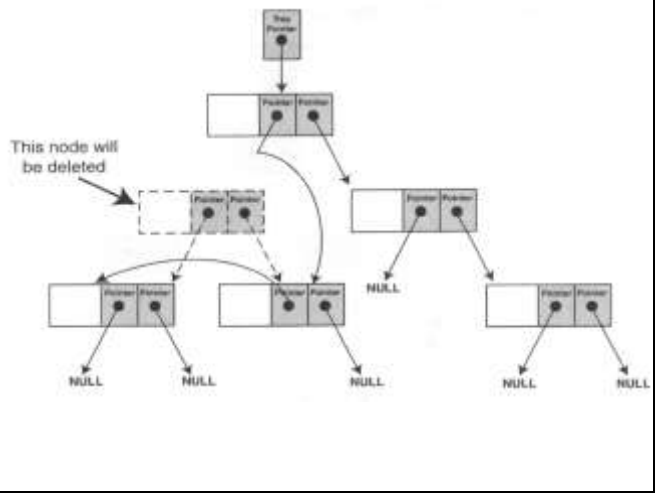
```
void main(void)
{
        BinaryTree   tree;
        tree.insertNode(5);
        tree.insertNode(8);
        tree.insertNode(3);
        tree.insertNode(12);
        tree.insertNode(9);
}
```



Inorder traversal:	**Preorder traversal:**	**Postorder traversal:**
3	5	3
5	3	9
8	8	12
9	12	8
12	9	5

Deleting a node

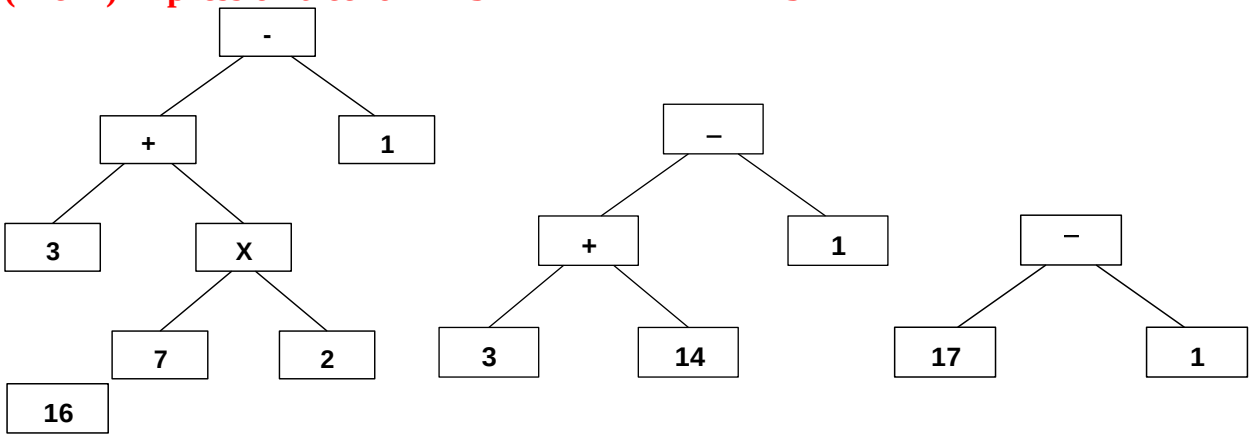
There are 2 possible situations to face when deleting a non-leaf node:

A) the node has one child, or	B) the node has two children.
	

Q1 Suppose the following values are inserted into a binary tree, in the order given: 2,17,9,10,22,24,30,18, 3,14, 20 , Draw a diagram of the resulting binary tree.

Q2 Draw a diagram of the resulting binary tree after the following operations-: insert 5, 8, 3 , 12, 9, remove 8, 12 insert 10.

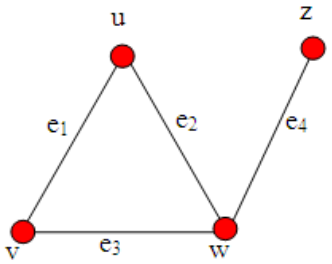
(Prefix) Expression tree for $3 + 7 \times 2 - 1$ **=** $- + 3 \times 7\ 2\ 1$



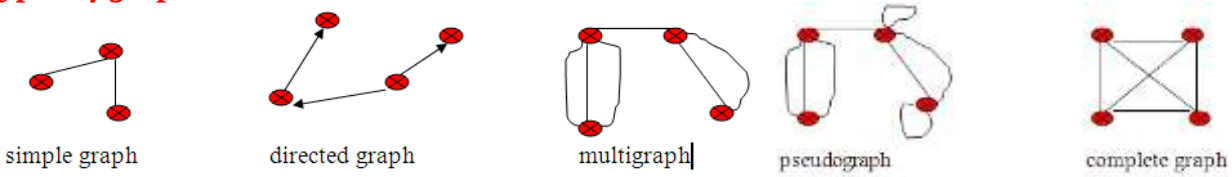
Q1 Create an expression tree for the following expression: 6 * 4 / 2 + 7 - 5 * 3

[6] Graphs

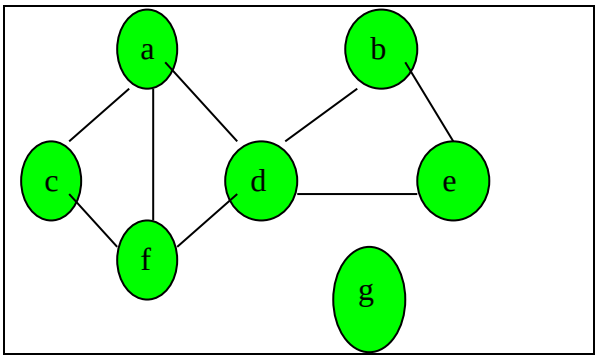
vertex-set V(G) = {u, v, w, z}
edge-set E(G) = {e_1, e_2, e_3, e_4}
G = {V(G), E(G)}
= {{u, v, w, z}, {e_1, e_2, e_3, e_4}}



Types of graphs



Representing graphs



The adjacency matrix:

	a	b	c	d	e	f	g
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

The incident matrix:

	ac	ad	af	bd	be	cf	de	df
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	1	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	0	1
g	0	0	0	0	0	0	0	0

The adjacency list.