Introduction to Algorithms

Module Code: CELEN086

Lecture 7

(18/11/24)

Lecturer & Convenor: Manish Dhyani
Email: manish.dhyani@nottingham.edu.cn

Coursework : Introduction to Algorithm

- Weighting: 25%
- CW will be available on Moodle for the access

  From 22/11/2024 on Moodle .
- Deadline for submission - 06/12/2024 by 4pm
- Late submission will be penalized according to

  university policy.
- **You may use any algorithm from the lectures and seminars. Any algorithm you use must be written out in full.**

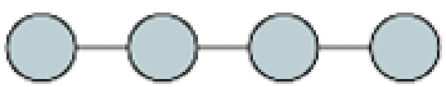
Linear and non-linear data structure

Data structures are conceptual tools for storing, sorting and manipulating various forms of data.

Linear data structure:

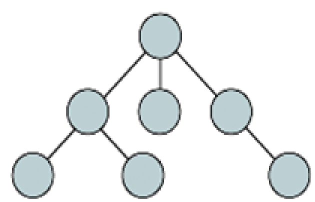
data elements are sequentially connected.

array/list

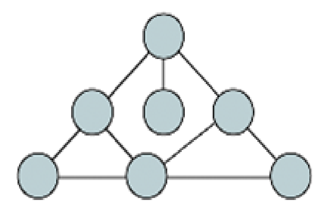


Non-linear data structure:

data elements are hierarchically connected and are present at various levels.



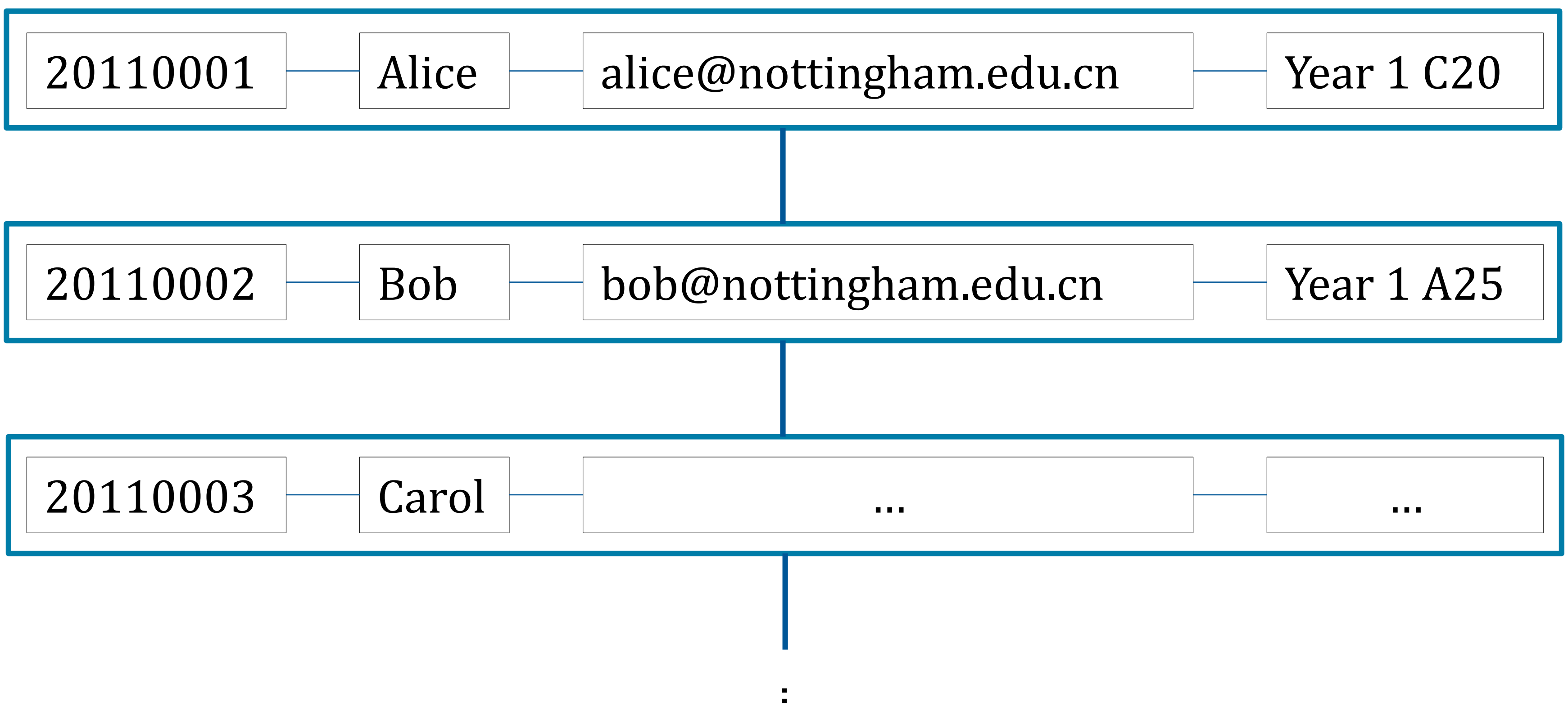
tree



graph

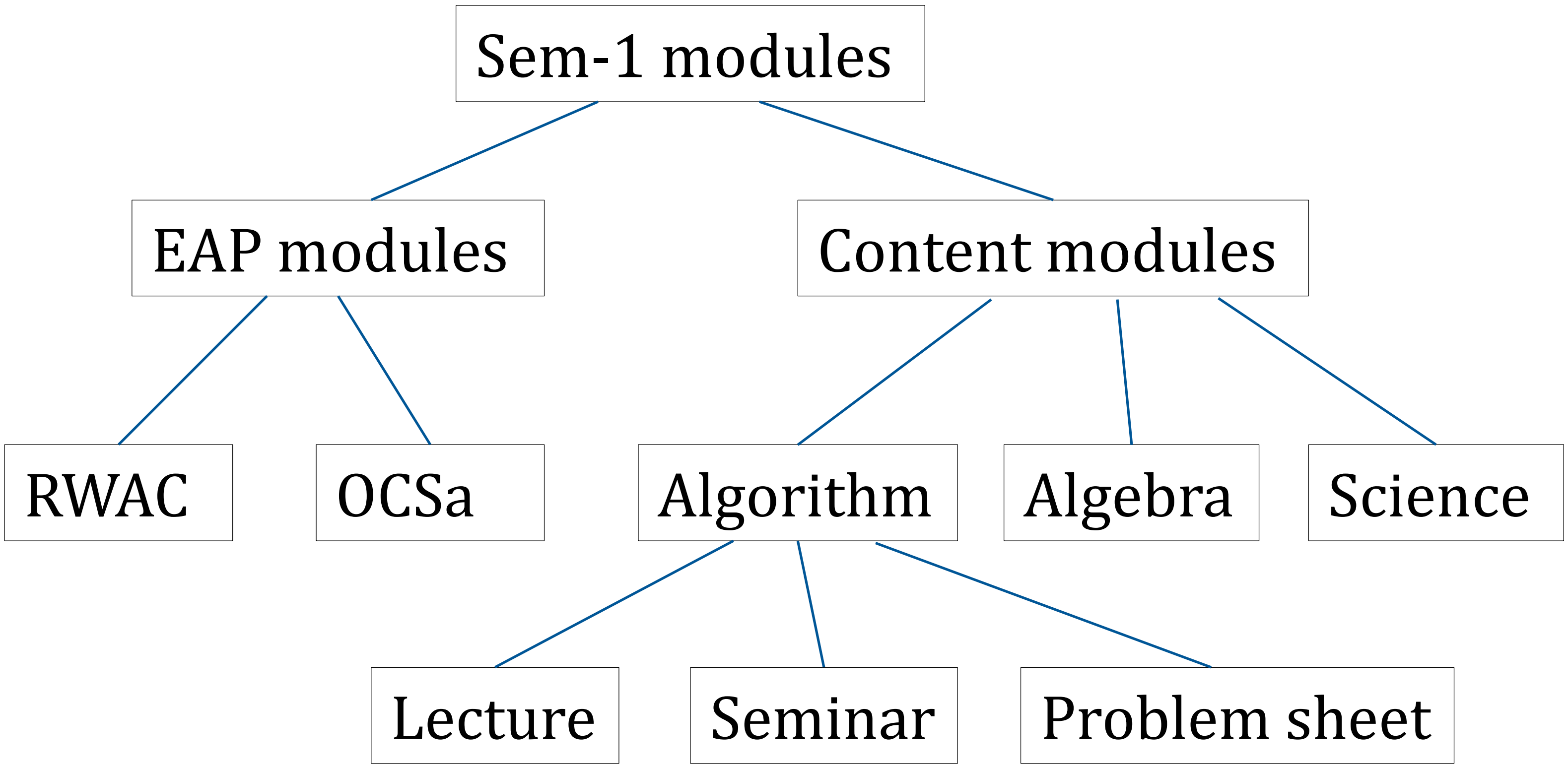
Example of linear data structure

Students register list

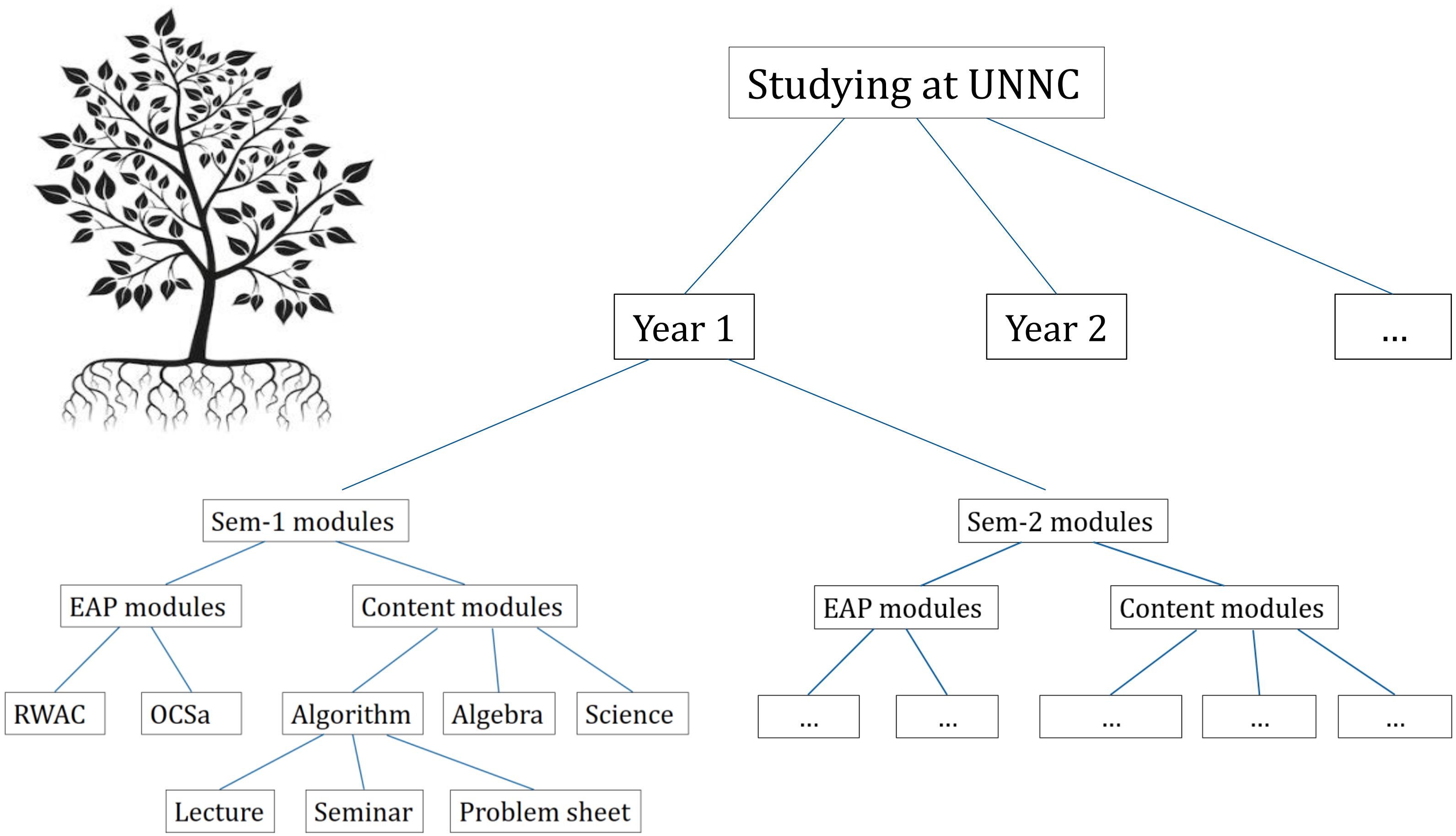


Example of non-linear data structure

Students registered modules



Example of non-linear data structure



Tree data structure

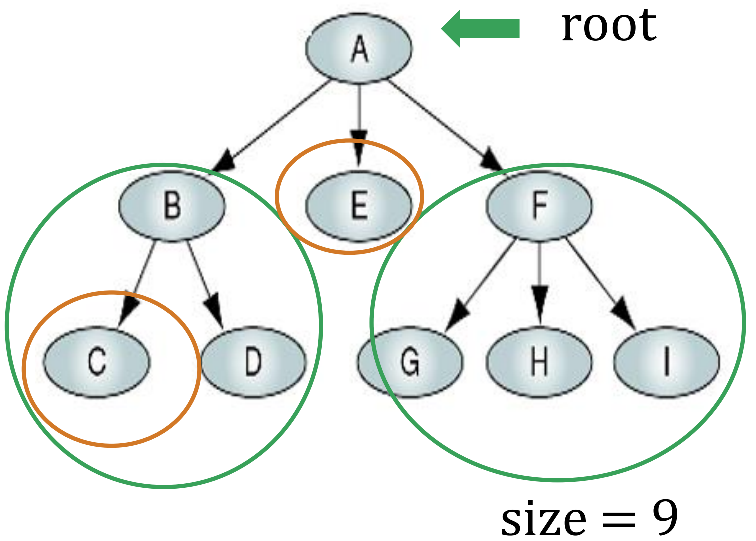
A tree is a set of nodes that are either empty or store a value.

Nodes are connected via branches (or called edges).

Root of a tree:
a principal node from which all other nodes and branches develop

Size of a tree:
total number of nodes in the tree

Subtrees:
smaller trees that descend from root or other lower nodes



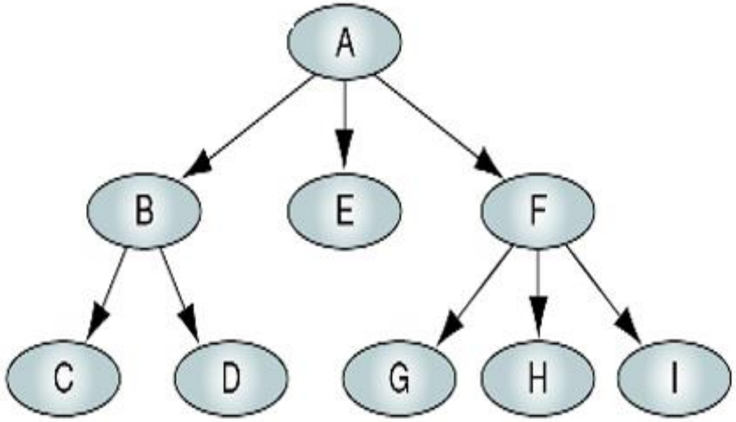
Node

Parent node: A is the parent node of B, E, F

the node with a branch from itself to any other successive node

Child node: C, D are children nodes of B

a descendant of any node



Sibling nodes: G,H,I are sibling nodes

nodes that belong to the same parent

Leaf nodes: E, C, D, G, H, I are all the leaf nodes

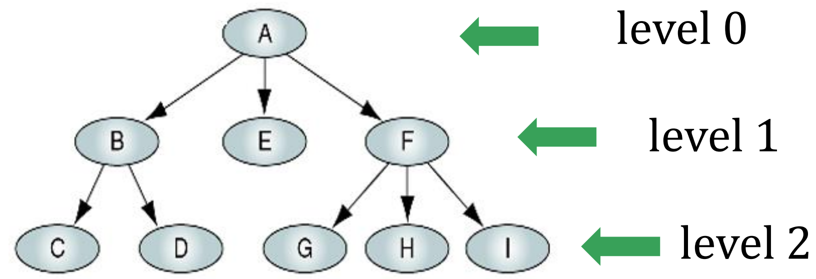
nodes with no child

Degree of a node: degree(A) = 3 degree(B) = 2 degree(C) = 0

total number of children of a node

Height and Depth

tree depth = tree height = 2



<u>Level</u>

Root node is at level 0; root node's children are at level 1,… and so on.

<u>Height</u> of a node

The number of edges from the leaf node upwards to the particular node in the longest path.

Maximum height of nodes is called Height of Tree.

Height of C: 0

Height of B: 1

Height of E: 0

Height of A: 2

(one of the longest paths: CB-BA)

<u>Depth</u> of a node

Total number of edges from the root node to the particular node.

Maximum depth of nodes is called Depth of Tree.

Depth of A: 0

Depth of B: 1

Depth of E: 1

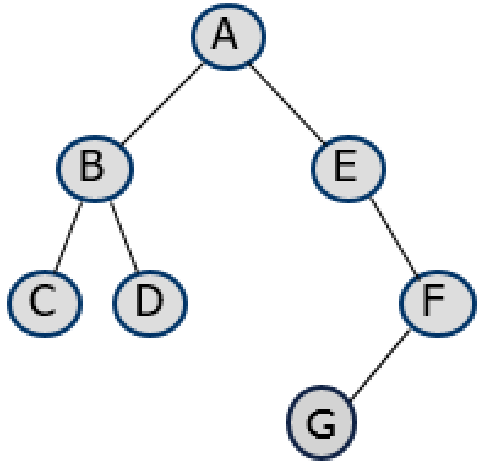
Depth of C: 2

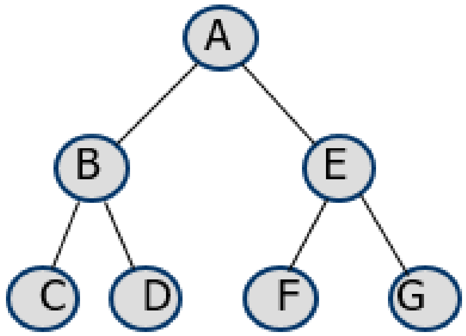
Binary tree

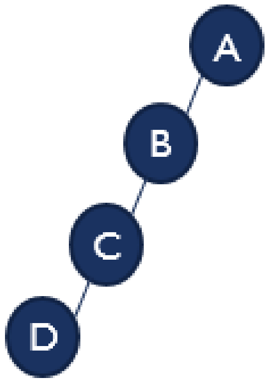
A binary tree is a tree in which:

each node has at most two children nodes (maximum degree =2)

Examples of binary trees:







Create a binary tree

Note:
Leaf and leaf node are different!

- leaf   nil

  Creating/representing an empty tree

- node(leaf, x, leaf)   cons(x,nil)

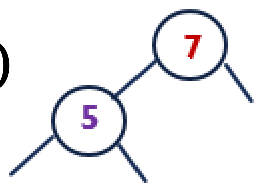
  node(leaf,5,leaf)

  Creating a leaf node that stores value x

- node(left-subtree, x, right-subtree)

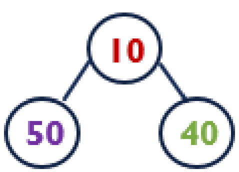
  $$\text{node}(\underbrace{\text{node(leaf,5,leaf)}}_{\text{left}},7,\underbrace{\text{leaf}}_{\text{right}})$$

  cons(x,list)

  Making a larger tree
  (storing x in parent node of two given subtrees)

$$\text{node}(\underbrace{\text{node(leaf,50,leaf)}}_{\text{left}},10,\underbrace{\text{node(leaf,40,leaf)}}_{\text{right}})$$



Basic functions

Like list commands, we have basic functions that work on binary trees.

- isLeaf(tree) — to return Boolean value True if the tree is empty (a leaf); False if the tree is non-empty

  isEmpty(list)

- root(tree) — to return the value stored in the root

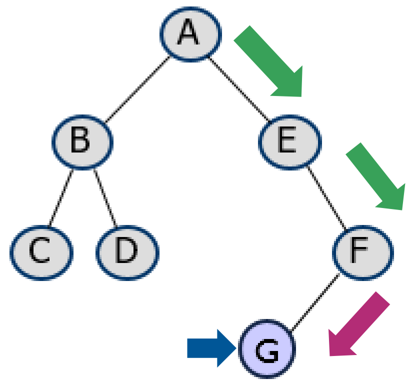
  head(list)

- left(tree) — to return the left subtree
- right(tree) — to return the right subtree

  tail(list)

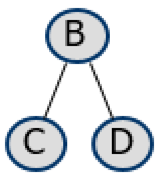
Example

Consider the binary tree T:

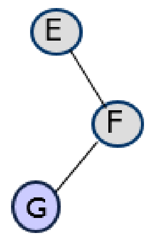


root(T) = A

left(T) =



right(T) =



right(left(T)) = D

root(right(left(T))) = D

root(root(left(T))) = Not valid!

left(right(left(T))) = leaf

How to obtain the value stored in the leaf node G ?

root(left(right(right(T)))) = G

Use left/right to walk down in a binary tree and use root to retrieve the value.

Algorithm: tree size

Design a recursive algorithm that computes the size (total number of nodes) of a binary tree.

Analysis: Decomposing the problem into

smaller instances of the same problem.

Tree sizes of smaller trees: left/right subtrees

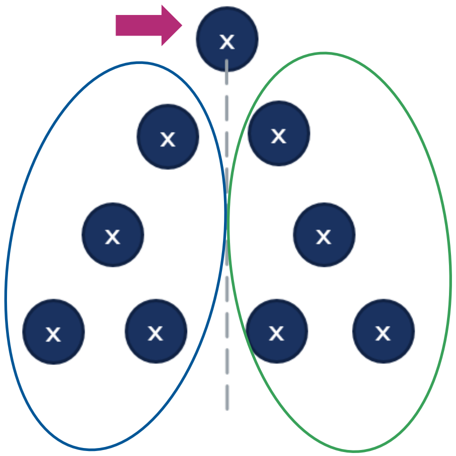
Recursions stop at leafs.

numbers of nodes in current tree

= numbers of nodes in left subtree

+ numbers of nodes in right subtree

+ 1



Algorithm: tree size

Algorithm: size(T)
Requires: a binary tree T
Returns: total number of nodes in T (size of T)

```
1.  if  isLeaf(T)
2.        return  0
3.  else
4.        return    size(left(T))+size(right(T))+1
5.  endif
```

Question to think:

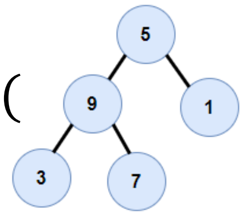
Can we replace Lines 1&2 by following statements, and maintain the rest of above algorithm? Why?

```
1.  if  isLeaf(left(T)) && isLeaf(right(T)) // checking leaf node
2.        return  1
```

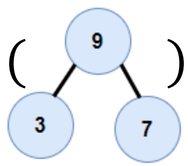
Trace

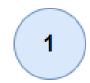
Algorithm: size(T)
Requires: a binary tree T
Returns: total number of nodes in T (size of T)

```
if isLeaf(T)
    return 0
else
    return size(left(T))+size(right(T))+1
endif
```

Trace: size (

) = 5

return size (

) +size (

) + 1 =3+1+1=5

return size (

)+size (

) + 1

=1+1+1=3

return size(leaf) + size (leaf) +1

= 0+0+1=1

return size(leaf) + size (leaf) +1

= 0+0+1=1

return size(leaf) + size (leaf) +1

= 0+0+1=1

(backtracking)

Algorithm: search in a binary tree

Design a recursive algorithm that searches for a node value in a binary tree.

```
Algorithm: search(x, T)
Requires: a binary tree T and an element x
Returns: True if x occurs in T; False otherwise

1. if  isLeaf(T)
2.      return  False
3. elseif    x==root(T)
4.      return  True
5. else
6.      return   search(x, left(T)) || search(x, right(T))
7. endif
```

Note:
In general, when we describe the time complexity of algorithms without any particular specifications (best/average/worst), we are aiming on the worst case scenario.

What is the time complexity of this algorithm? $O(n)$

(Assume the size of tree is n and the height of tree is h.)

Algorithm: sum of all node values in a binary tree

Design a recursive algorithm that find sum of all node values in a binary tree.

Algorithm: sumBT(T)
Requires: a binary tree T
Returns: a number i.e. sum of all node values in T

```
if  isLeaf(T)
     return  0
else
     return sumBT(left(T)) + sumBT( right(T))  + root(T)
endif
```

Review Tree Data structure

Nonlinear Data Structure Binary Tree