Introduction to Algorithm

- Why study algorithms
- Learning outcomes
- Preliminaries
- Some aspects of algorithms
  - ✓ Mechanization of abstraction
  - ✓ Analysis of algorithm
  - ✓ Data structure
- Organization of the lectures

Why study algorithms

Definition: An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. All algorithms must satisfy the following criteria:

1. Input: Zero or more quantities are externally supplied.
2. Output: At least one quantity is produced.
3. Definiteness : Each instruction is clear and unambiguous.
4. Finiteness: The algorithm terminates after a finite number of steps.
5. Effectiveness: Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper.

- To understand and appreciate their impact
- Technology: Internet, Web search, CPU design, VLSI routing, Computer Graphics, games, simulations...
- Advancement of Science: Gene Sequence Alignment, Phylogeny, Galaxy formations, particle collisions, medical applications, signal processing
- Encounter in other parts of CS discipline: OS, Compilers, Networks, Parallel Computing, Quantum Computing ...

Learning outcome

On completing the module

- Identify the fundamental strategies in algorithm design
- Distinguish which strategy is appropriate to solve a given problem
- Classify different algorithmic strategies
- Analyse a given algorithm and assess its efficiency.
- Apply techniques of proof by induction to verify certain properties of algorithms

Preliminaries

- Basic concepts of algorithms from Part 1
  - Programs = data structures + algorithms
  - Pseudo code (control abstraction)

- Programming ability
  - Use of modules, procedures, functions
  - Recursion

- Basic grasp of complexity
  - Analysis of Algorithms
  - Big Oh, worst case, expected case, ...

Aspects of algorithms

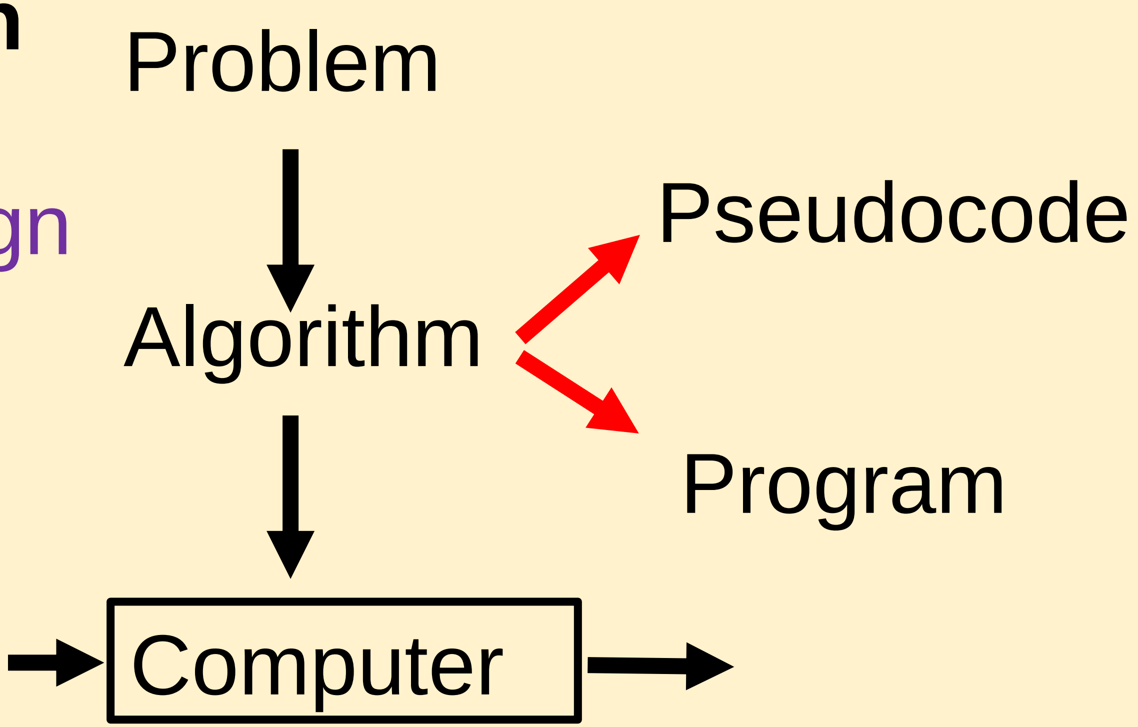
1. How to devise algorithms
   - ✓ A major goal of this course is to study various design techniques that have been proven to be useful in that they have often yielded to good algorithms.
   - ✓ By mastering these design strategies, it will become easier for you to devise new and useful algorithms.
2. How to validate algorithms:
   - ✓ Once an algorithm is devised, it is necessary to show that it computes the correct answer
   - ✓ for all possible legal inputs.
   - ✓ The algorithm need not as yet expressed as a program.

3. How to analyse algorithms
    - ✓ refers to the task of determining how much computing time and storage an algorithm requires.
    - ✓ This is a challenge area sometimes requiring great mathematical skills.
4. How to test a program
    - ✓ Debugging and profiling

➢ We will concentrate on design and analysis of algorithms

Mechanisation of abstraction

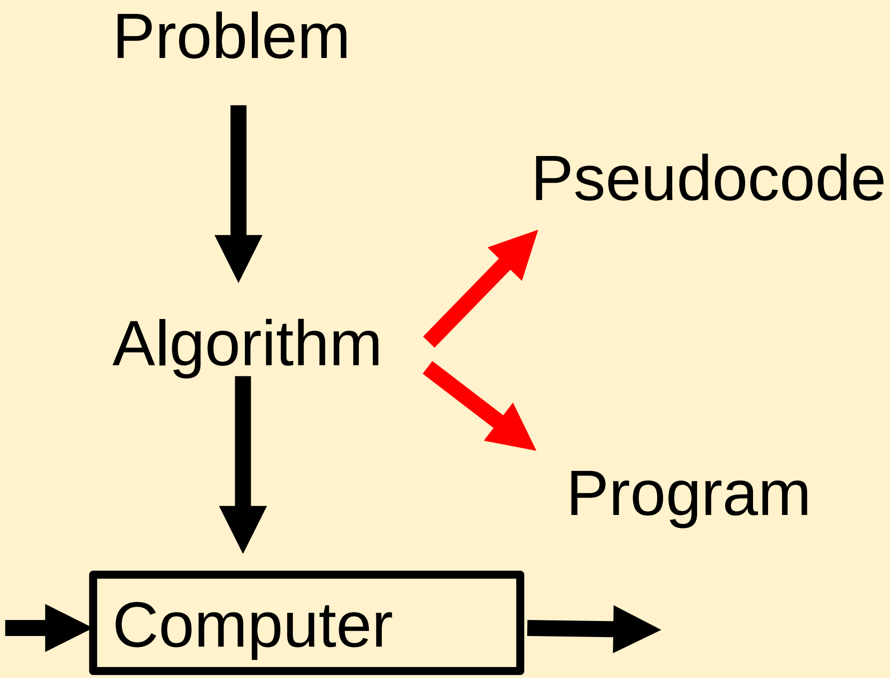
We will concentrate on design and analysis of algorithms.



- Algorithms will be at the level of pseudocode.
- We will translate some algorithms into actual code as practical exercises, and to understand better the “Definiteness” criterion.

- Algorithms are NOT Programs
- A “program” is just a way of expressing Algorithms formally in a computer language
- Satisfying the “definiteness” criteria
- And (therefore) executable on a “computer”.
- Definition by Nickalus Wirth

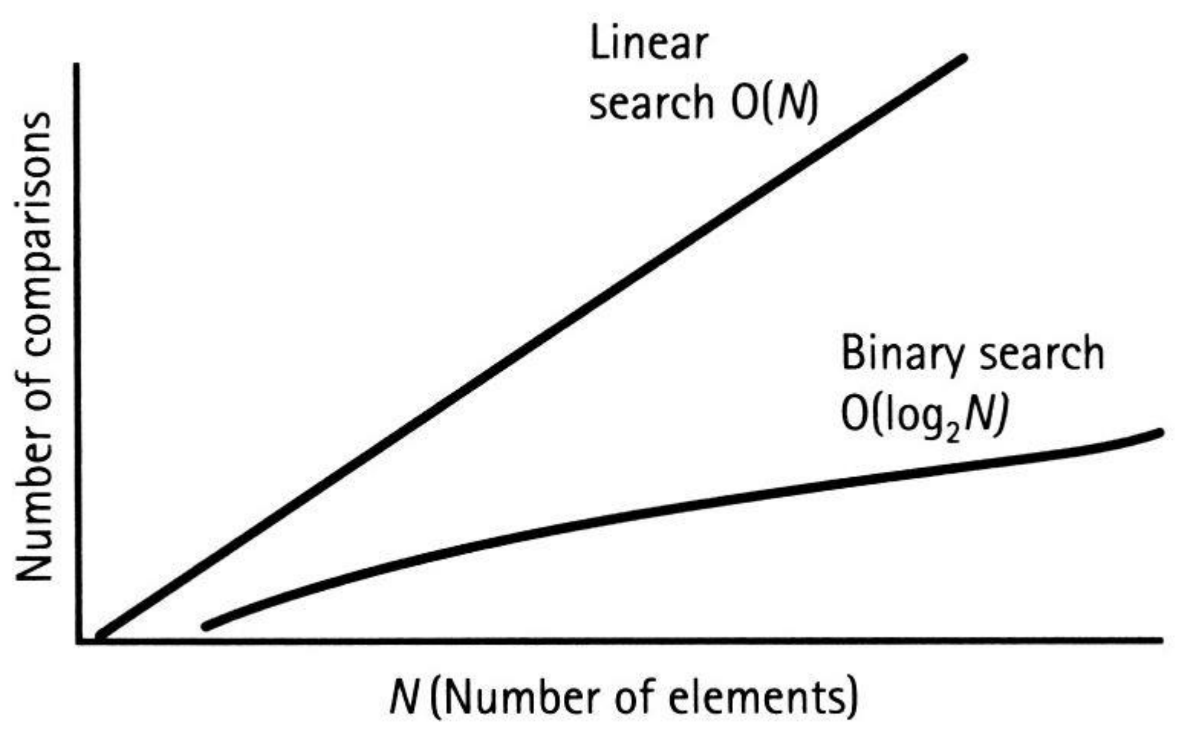
  Program = Algorithms + Data structures
- all algorithms manipulate data in one form or another.



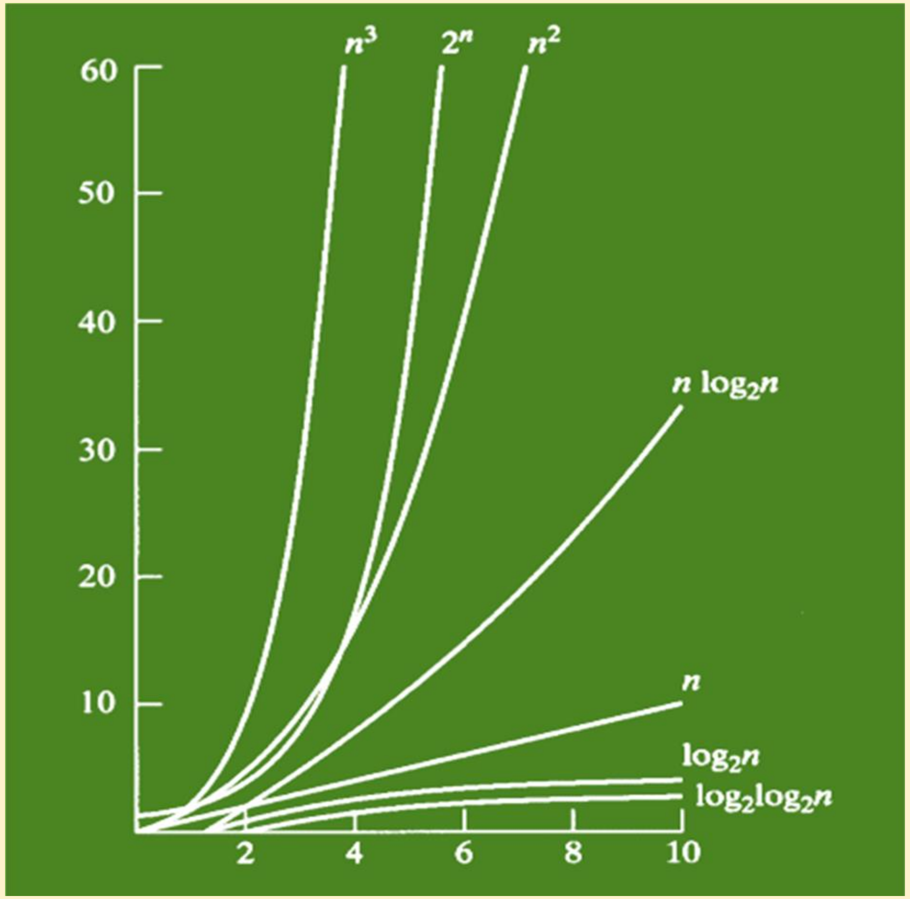
Analysis of algorithm using time complexity

- We want to decide which algorithm to choose from the possible solutions.

  e.g. Linear search or Binary Search ?



- Think about locating a given page in a book



- We are interested in “good” algorithms, i.e. efficient algorithms.
- Time complexity is a “tool”
- We will use to help our decision process without actually implementing an algorithm as a “program”.
- Only “efficient” solutions need be considered for the problem.

Data Structure

- Data structure is used to denote a particular way of organizing data for particular types of operation.
- The ability to formulate an efficient algorithm for a complex problem depends on being able to organize the data appropriately.
- Abstract data types(ADT) can be defined only by the operations that may be performed on them, without having to worry about all the implementational details.
- At higher level, control abstraction hides implementational details to speed up the development of algorithm.
- In this module data structure is studied with the algorithm where needed.

Types of data structure

- Primitive data structure
  - Integer
  - Float
  - Character
  - Boolean
- Non- Primitive data structure
  - Linear
    - Arrays
  - Non-Linear

- Linked list: A list with elements comprising of two items - the data and a reference to the next node.
- Stack is last-in-first-out structure
    1. Push: inserts a data item onto the stack;
    2. Pop: removes a data item from it;
    3. Peek or top: accesses a item on top   without removal.
- Queue is first-in-first-out structure
    1. Enqueue: inserts a data item into the queue
    2. Dequeue: removes the first data item from it
    3. Front: accesses and serves the first data item in the queue.

More details and Tree and Graph will be covered later in more details.

Divide and Conquer

Matrix multiplication

Strassen Matrix multiplication

Convex hull

Master theorem

max-min

Selection

Multiplication of two integers

Greedy

Kruskal MST

Fractional Knapsack

Prim MST

Dijkstra shortest path

Dynamic Programming

Floyd's all pairs shortest path

String edit

Warshall transitive closure

Traveling saleman

Tree traversal,
Heap
(data strutrure)

Graph,
Queue,
Stack. Traversal
(data structure)

Organisation of the lectures

- Each week learning materials are in a file containing two topics, together with a few videos under Week no., and more
- Midterm revision I and final revision II
- Mid term mock tests for learning outcome feedback
- Three classes of algorithms are covered in the order of
  1. Divide and conquer
  2. Greedy
  3. Dynamic programming
- Additional data structures (tree and graph) are covered in the first five weeks, prior for Greedy and dynamic programming
- An online tests open in Week 10, counts 15% of the module mark.

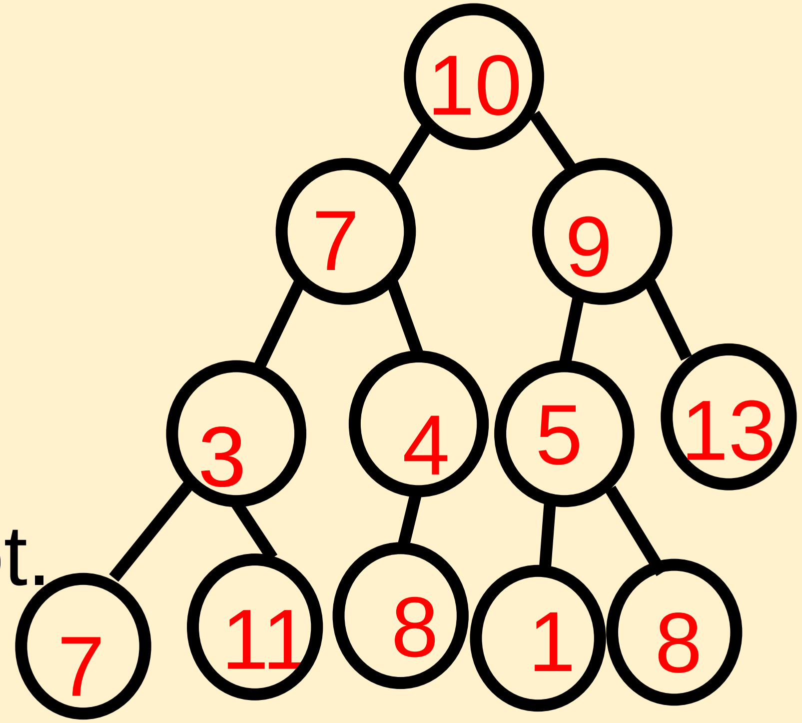
Useful books:

- Computer Algorithms C++ by Ellis Horowitz, Sartaj Sahni, and S. Rajasekaran, May 1996 –Available in the library
- Fundamentals of Computer Algorithms – Horowitz & Sahni, Pitman 1978 (a classic),
- Data Structures and Algorithms in Java (Goodrich and Tamassia)
- Analysis of Algorithms -- An active Learning Approach. J.J.McConnell. Jones and Bartlett 2001
- Computational Geometry in “C”, Joseph O’Rourke

Tree traversal

- Binary tree data structure
- Binary tree traversal
  - Recursion
  - Inorder
  - Preorder
  - Postorder

Binary tree (data structure)



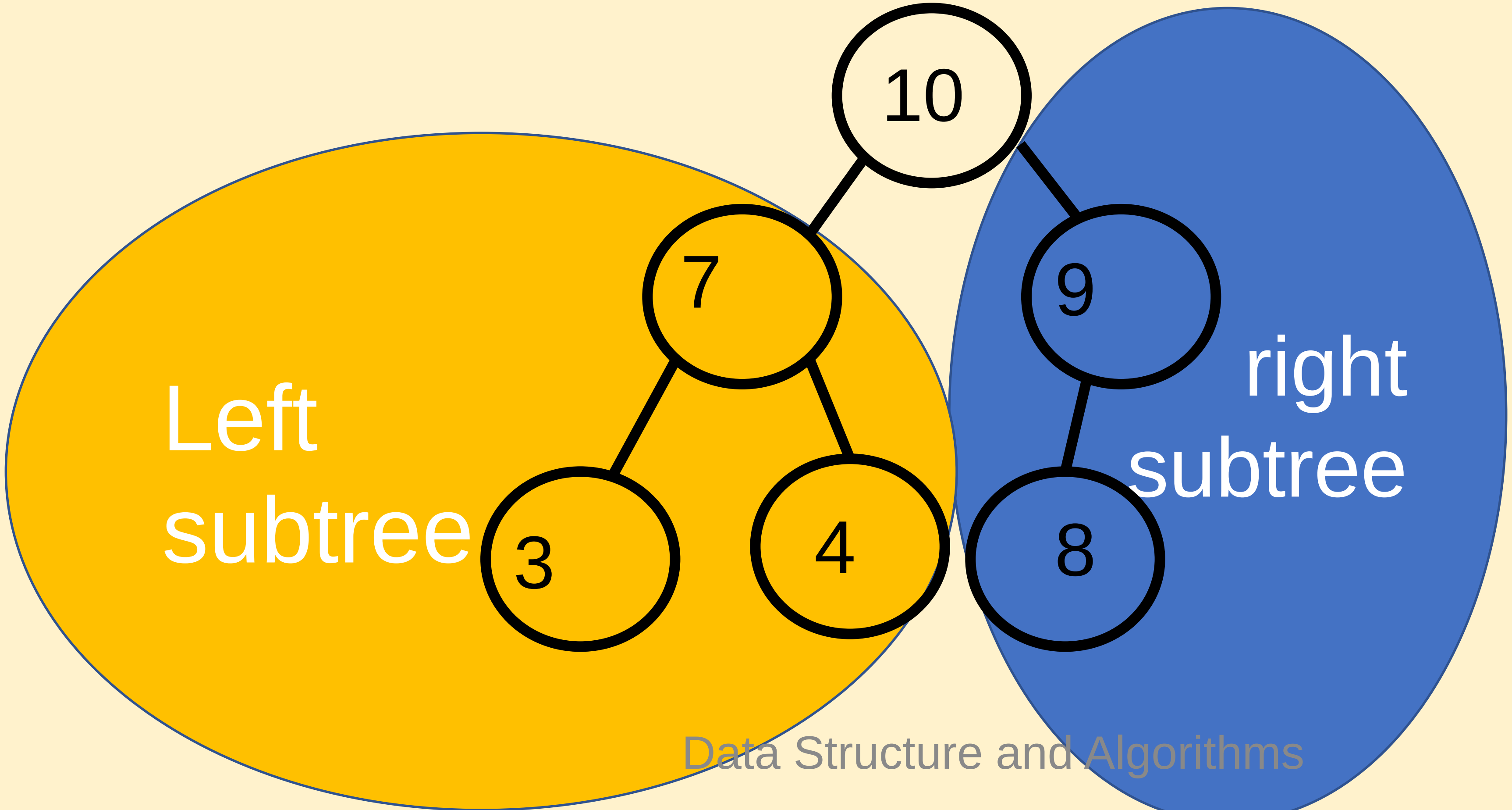
- A binary tree is a finite set of nodes.
- The set might be empty.
- When its not empty, it follows there rules:

a) There is one special node, called the root.
b) Each node can be associated with up to two other different nodes, called its *left child* and its *right child*.
a) Each node, except the root, has exactly one parent; the root has no parent.
b) Starting at a node, move to the node's parent and then move again to that node's parent, and keep moving upward, eventually the root is reached.

- It's a nonlinear organization of elements (the components do not form a simple sequence of first element, second element, and so on).
- Depth of a tree: the maximum depth of any of its leaves.
- Leaves :The nodes without children are leaves.
- Full binary tree: every leaf has the same depth and every non-leaf has two children.
- Complete binary tree: every level except for the deepest level must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.
- Traversal: an organized way to visit every member in the structure.

- Subtree: A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself.

  Any node in a tree T, together with all the nodes below it, comprise a subtree of T.



Binary Tree Traversal

- Trees are models on which algorithmic solutions for many problems are constructed.
  - Traversal: All nodes of a tree are examined/ evaluated.
  - Search: Only a subset of vertices (nodes) are examined.
- Binary tree is a tree structure in which there can be at most two children for each parent.
- A single node is the root.
- Tree traversal is an organized way to visit every member in the structure.

- A parent can have a left child subtree and a right child subtree
- A node may be a record of information.
- Visiting a node involves computation with one or more of the data fields at the node (e.g. ‘print the node’)
  Not just passing by it
- refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once.
- A node is visited when it is operated in the traversal.
- Types of Binary Tree traversals: In-order, Pre-order, Post-order (all use **recursion**)
- Such traversals can be classified by the order in which the nodes are visited.

Recursion

- Recursion is a computer programming technique involving the use of a procedure of invoking itself.
- Example: Fibonaccoi numbers

$$x_n = x_{n-1} + x_{n-2}$$

- A function can be defined as

  Function Fib(n)

  $Fib(n) = Fib(n-1) + Fib(n-2)$

  ….
- but what is missing?
- We need to terminate the program at base cases $Fib(0)=0$, $Fib(1)=1$

```
function x = Fib(n)
  if n<=0
  x = 0;
    else if n=1
       x= 1;
    else
      x = Fib(n-1) +Fib(n-2);
  end
```

Example: A factorial function n! $=1\times 2 \times \ldots \times n$, write two codes using iterative and recursion respectively for integers n >0

Iteration:

```
funcion x = fact(n)
  A(1)=1;
   for i=2:n
     A(i)= i * A(i-1);
   end;
  x =A(n);
end;
```

- Use recursion

$$n! = 1 \times 2 \times \ldots \times (n-1) \times n$$
$$= (n-1)! \times n$$

Let $fac(n) = n!$
$$fac(n) = fac(n-1) \times n$$

```
function x = fact(n)
    if n<=1
        x = 1;
    else
        x = n .* fact(n-1) ;
end;
```

In class exercise: A sum function $sum(n)=1+2+\ldots+n$, write a code using recursion, for integer n >0

Note : sum(n) =1+2+ …+(n-1)+n = sum(n-1) +n

function x = sum(n)

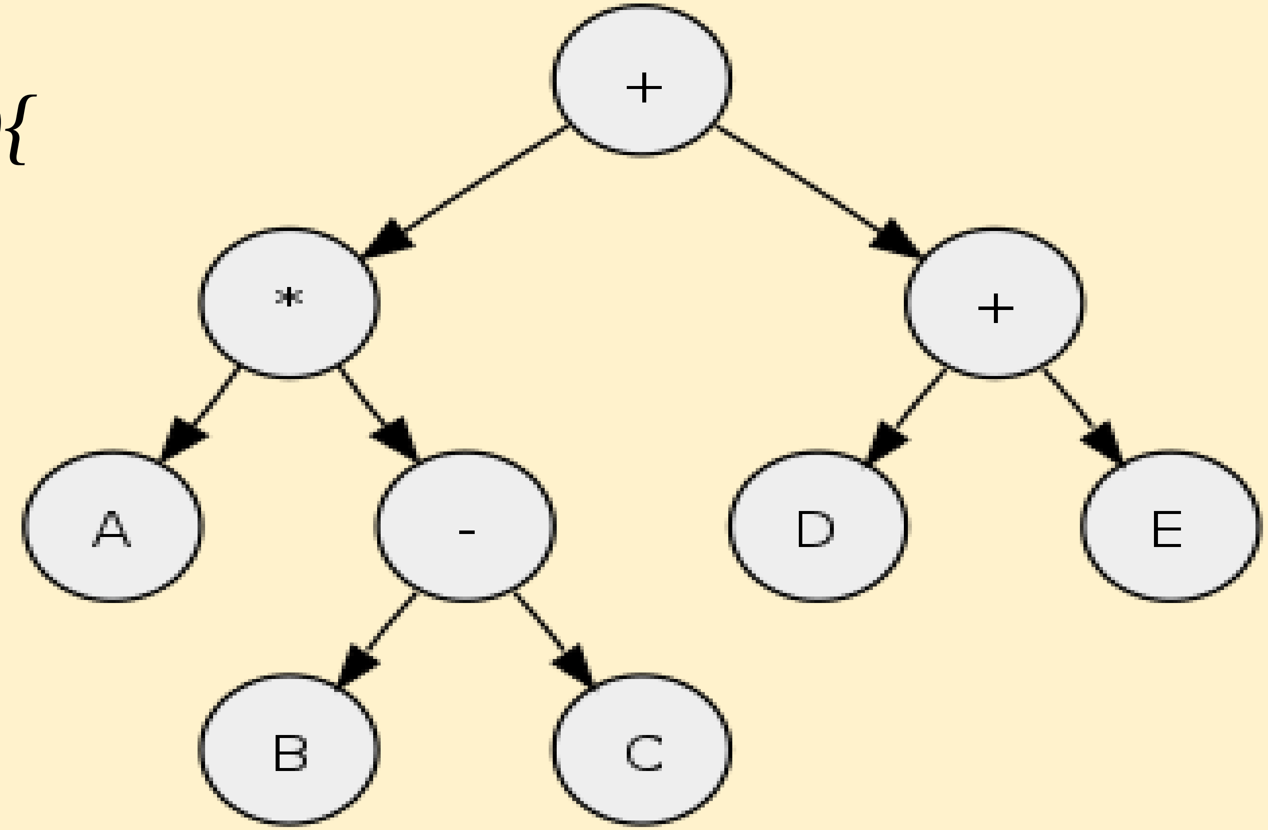
else

end;

In-Order

process in between the two subtrees.

Algorithm Inorder (Tree T){
if Not IsEmpty(T){
Inorder(Lchild(T));
Process(Data(T));
Inorder(Rchild(T));
}
}



Example: Tree representing the arithmetic expression

$A*(B-C) + (D+E)$

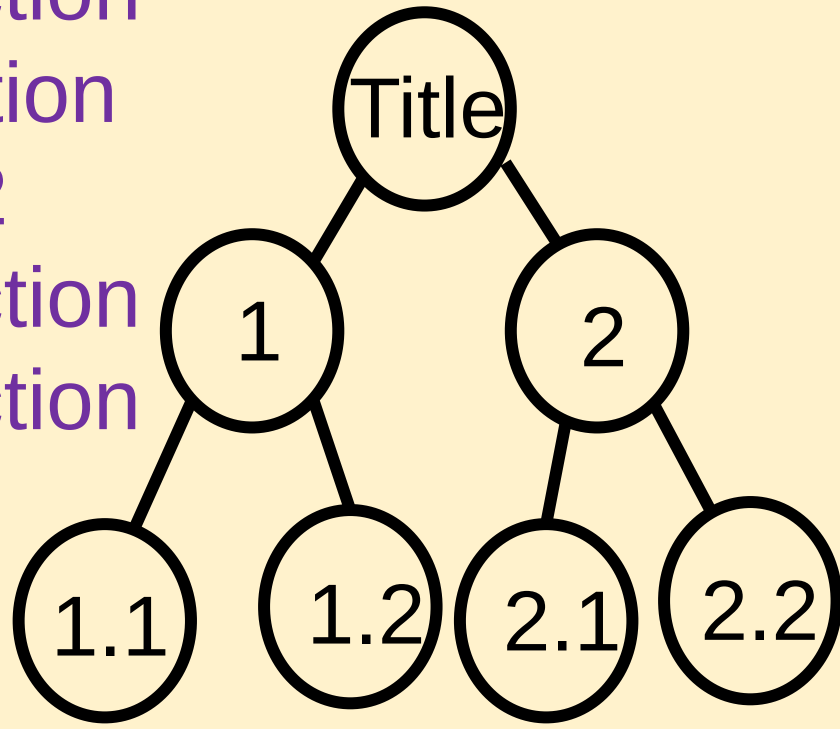
Example: Tree representing structure of a thesis content list

Pre-Order

process before anything else.

```
Algorithm Preorder (Tree  T){
  if  Not  IsEmpty(T){
        Process(Data(T));
        Preorder(Lchild(T));
        Preorder(Rchild(T));
    }
}
```

Title
1 Chapter 1
 1.1 a section
 1.2 a section
2 Chapter 2
 2.1 a section
 2.2 a section

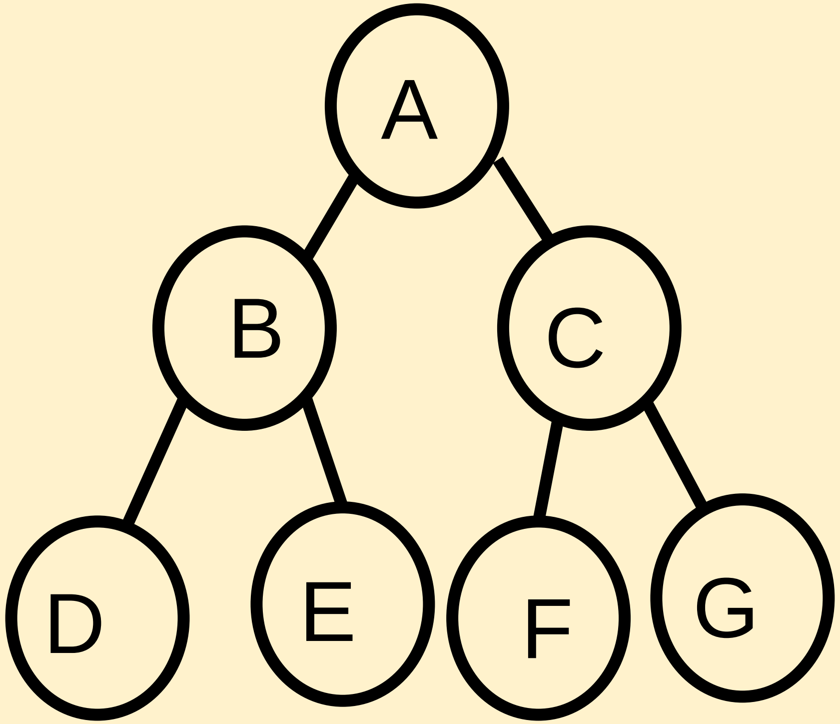


Post-Order
process after anything else.

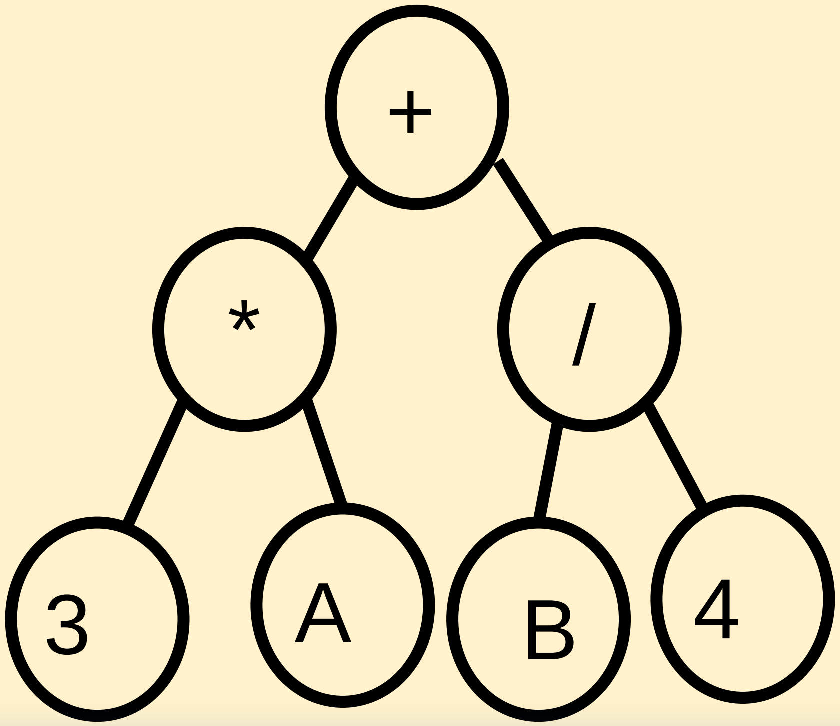
Example: Tree representing the order of deleting a tree.

Algorithm Postorder (Tree T){
if Not IsEmpty(T){
Postorder(Lchild(T));
Postorder(Rchild(T));
Process(Data(T));
}
}

D
E
B
F
G
C
A



In class exercise : Print out the nodes.



Inorder: [redacted]
Preorder:
Postorder: