ENG 346
Data Structures and Algorithms for Artificial Intelligence
Trees

Dr. Mehmet PEKMEZCİ

mpekmezci@gtu.edu.tr

https://github.com/mehmetpekmezci/GTU-ENG-346

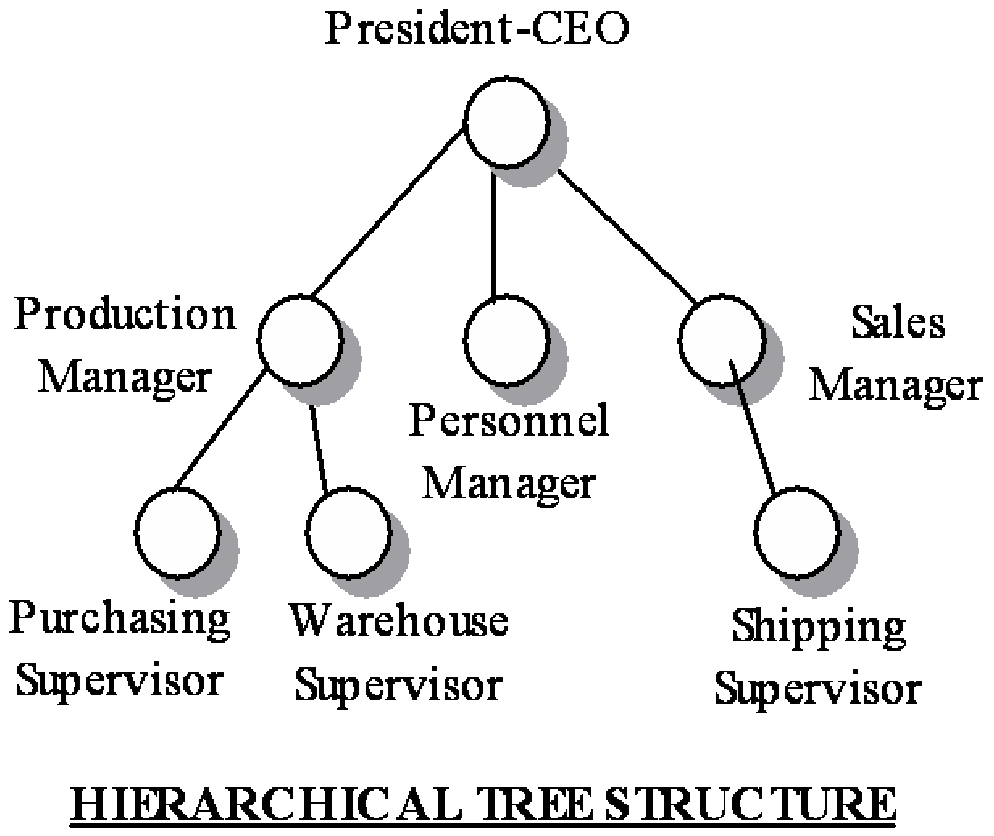
ENG-346 Teams code is **0uv7jlm**

Agenda

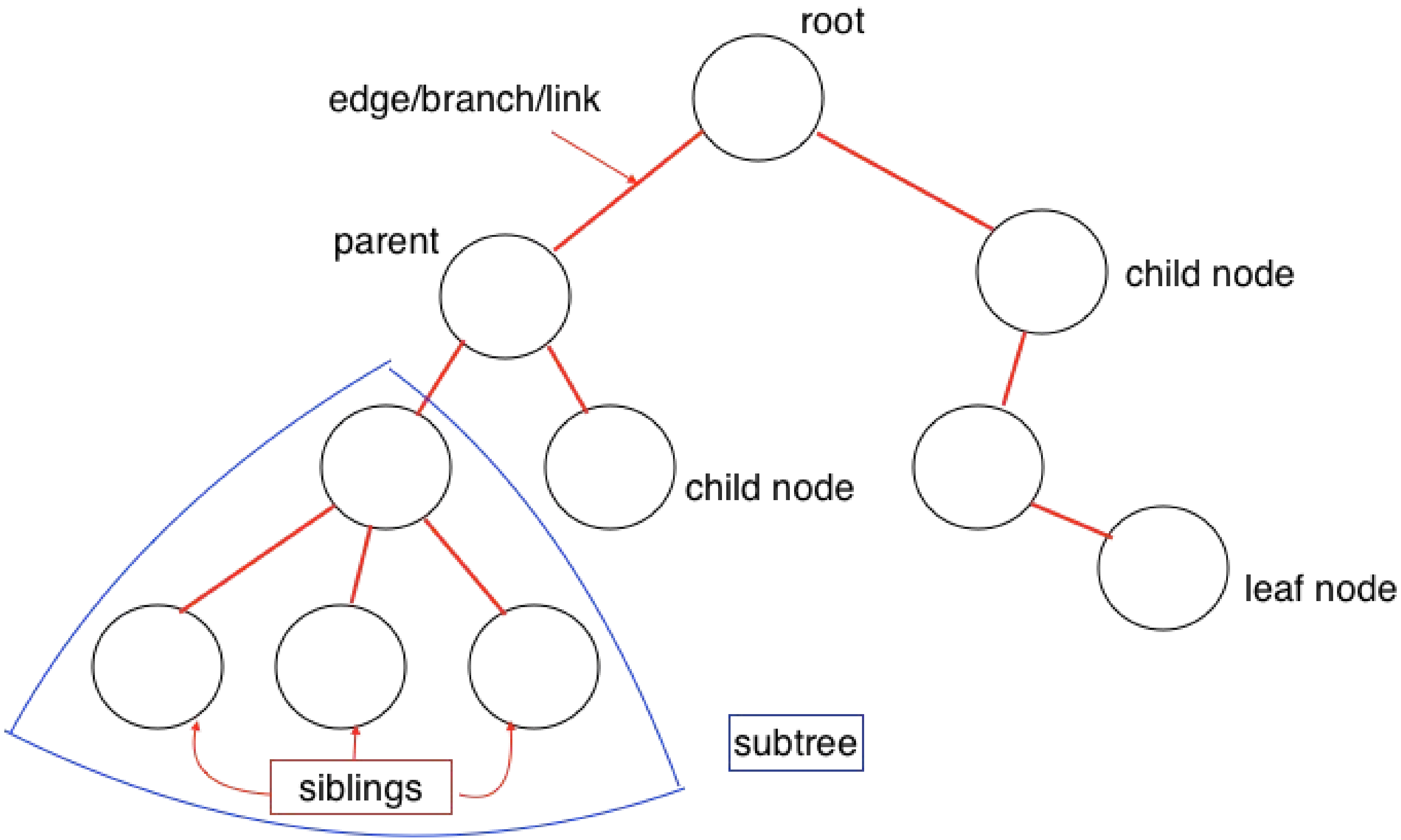
- Basic Concepts

What is a Tree?

- A *tree* is a *hierarchical* data structure consisting of *nodes* connected by *edges*, with a designated *root node* and *branches* that form a directed acyclic graph.

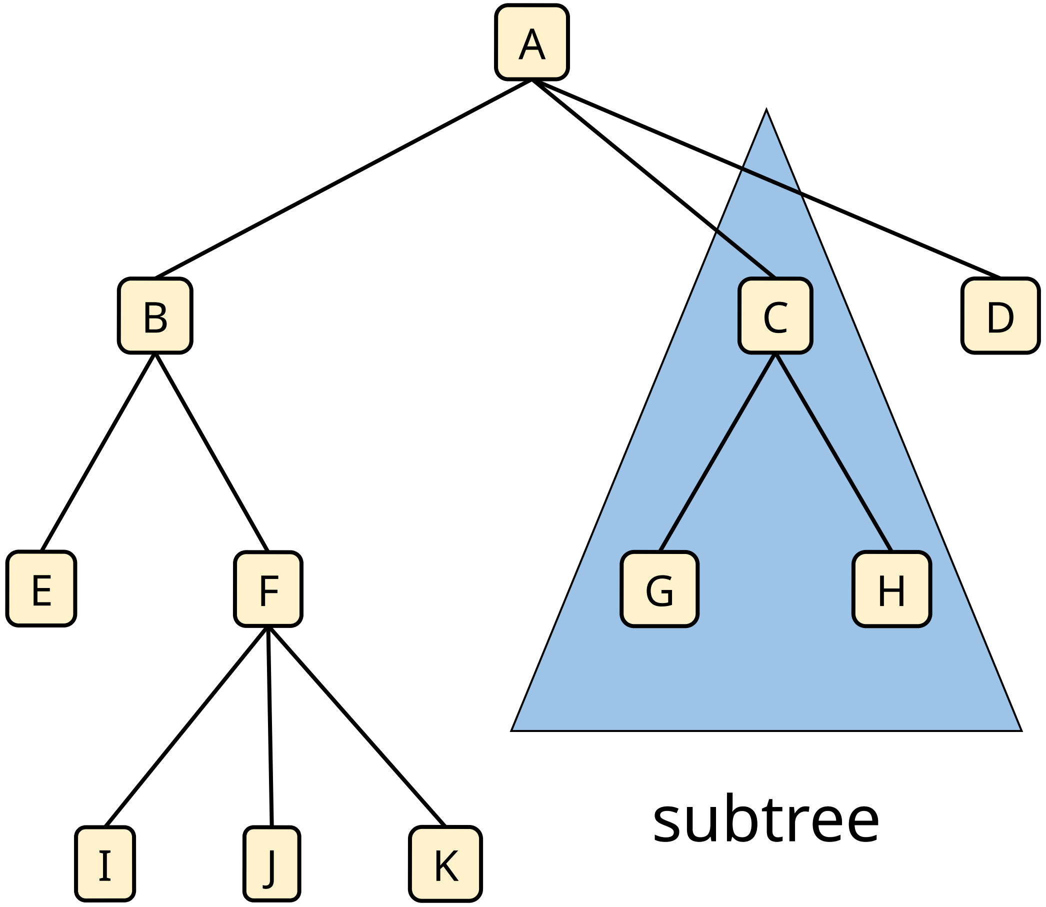


Definitions



Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Subtree: tree consisting of a node and its descendants



Applications of Trees

- Organization Charts
- Decision Trees in Machine Learning
- File Systems
- Expression Parsing
- DOM Model of Web pages

Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - Integer len()
  - Boolean is_empty()
  - Iterator positions()
  - Iterator iter()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterator children(p)
  - Integer num_children(p)

- Query methods:
  - Boolean is_leaf(p)
  - Boolean is_root(p)
- Update method:
  - element replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Tree ADT

- Assume T is a tree, and p is a position (node of the tree).

Functions/Operations	Description
p.element()	Data stored in p
Accessors	
T.root()	Return the position of the root of the tree. None is T is empty.
T.is_root(p)	True if p is the root, False otherwise.
T.parent(p)	Return the parent of position p. None if p is the root.
T.children(p)	Return list of children of position p.
T.num_children(p)	Number of children of position p.
Queries	
T.is_leaf(p)	True if position p has no children.
T.is_root(p)	True if position p is the root.

Abstract Tree Class

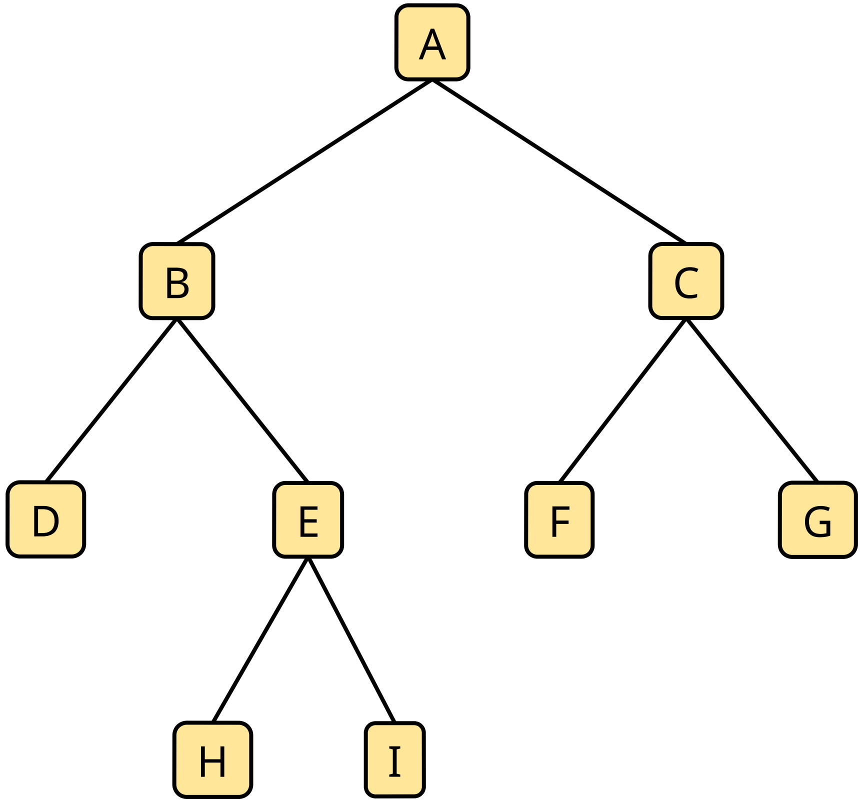
```
class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------ nested Position class ------------------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def element(self):
      """Return the element stored at this Position."""
      raise NotImplementedErr

    def __eq__(self, other):
      """Return True if other P
      raise NotImplementedErr

    def __ne__(self, other):
      """Return True if other c
      return not (self == oth

```

```
  # ---------- abstract methods that concrete subclass must support ----------
  def root(self):
    """Return Position representing the tree's root (or None if empty)."""
    raise NotImplementedError('must be implemented by subclass')

  def parent(self, p):
    """Return Position representing p's parent (or None if p is root)."""
    raise NotImplementedError('must be implemented by subclass')

  def num_children(self, p):
                                     ition p has."""
                                     lemented by subclass')

                                     senting p's children."""
                                     lemented by subclass')

                                     the tree."""
                                     lemented by subclass')
```

```
  # ---------- concrete methods implemented in this class ----------
  def is_root(self, p):
    """Return True if Position p represents the root of the tree."""
    return self.root( ) == p

  def is_leaf(self, p):
    """Return True if Position p does not have any children."""
    return self.num_children(p) == 0

  def is_empty(self):
    """Return True if the tree is empty."""
    return len(self) == 0
```

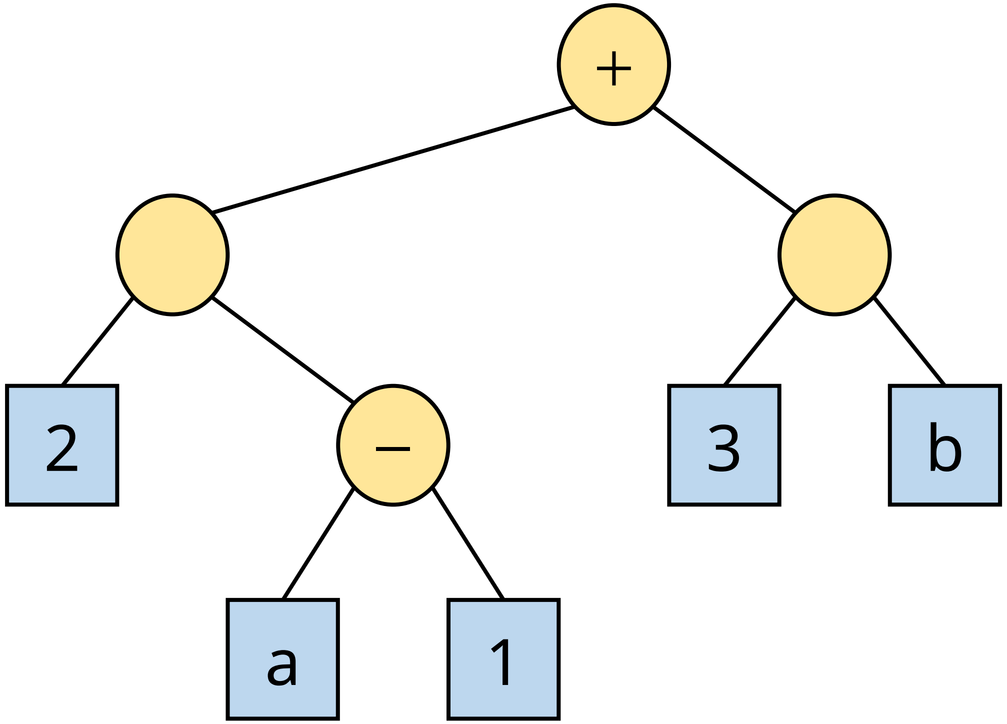
Binary Trees

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children
  - The children of a node are an ordered pair
- The children are called left child and right child
- Applications
  - arithmetic expressions
  - decision processes
  - searching



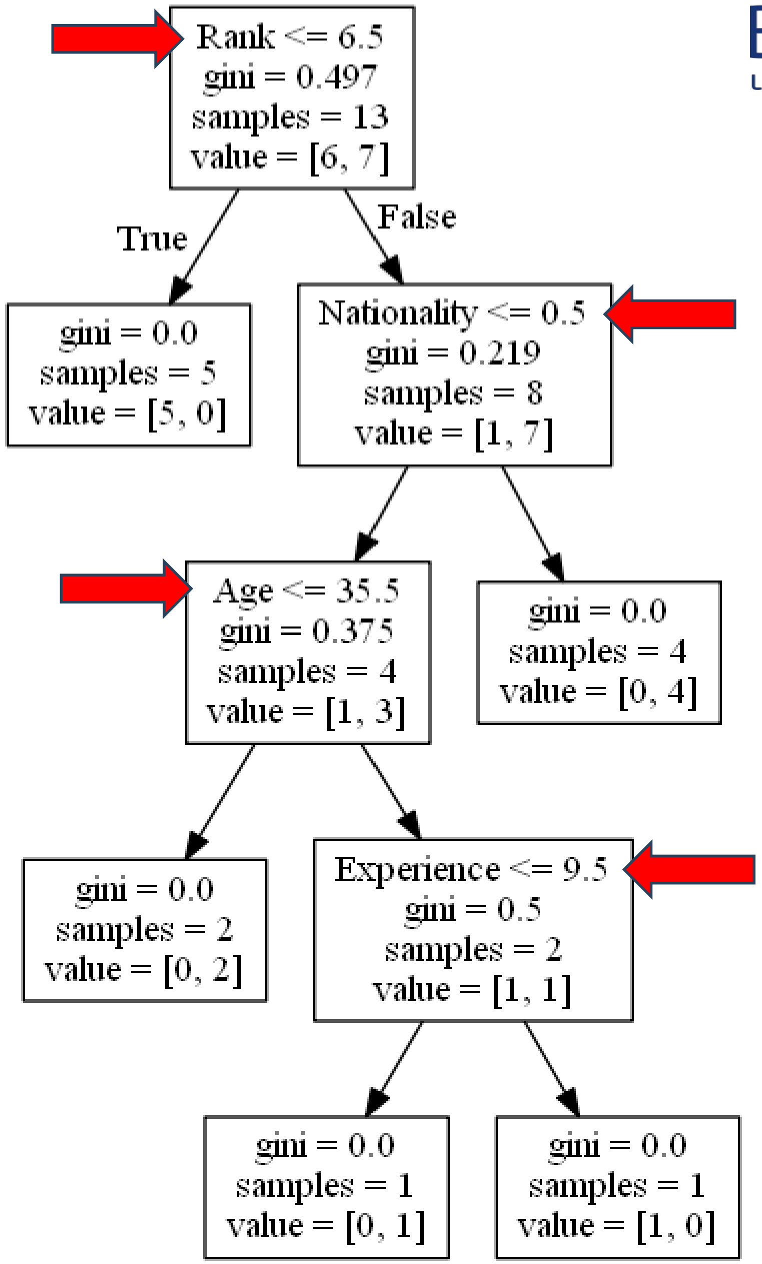
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - Internal nodes: operators
  - Leaf nodes: operands
- Example: arithmetic expression tree for the expression (2   (a – 1) + (3   b))



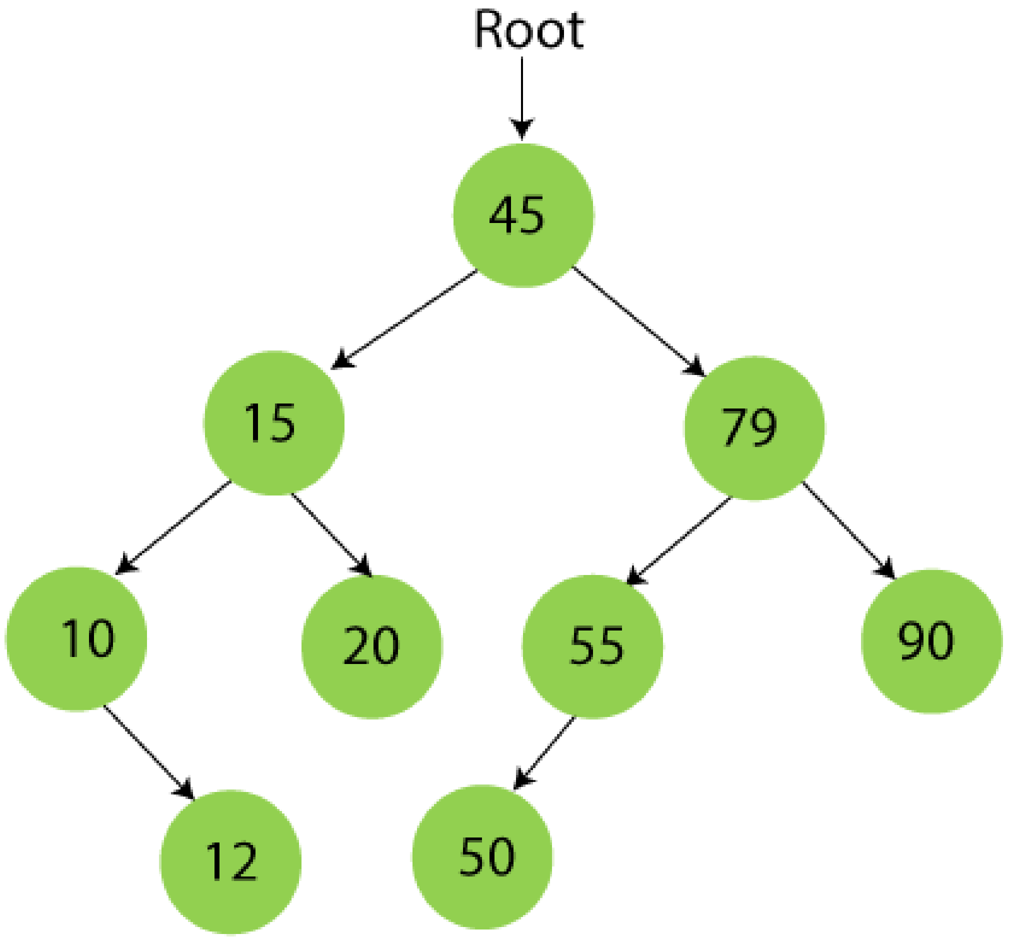
Decision Tree

- Binary tree associated with a decision process
  - Internal nodes: questions with yes/no answer
  - Leaf nodes: decisions



Searching

- Binary search tree



Binary Tree ADT

- The Binary Tree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT

- Additional methods:
  - position left(p)
  - position right(p)
  - position sibling(p)

Abstract BinaryTree Class

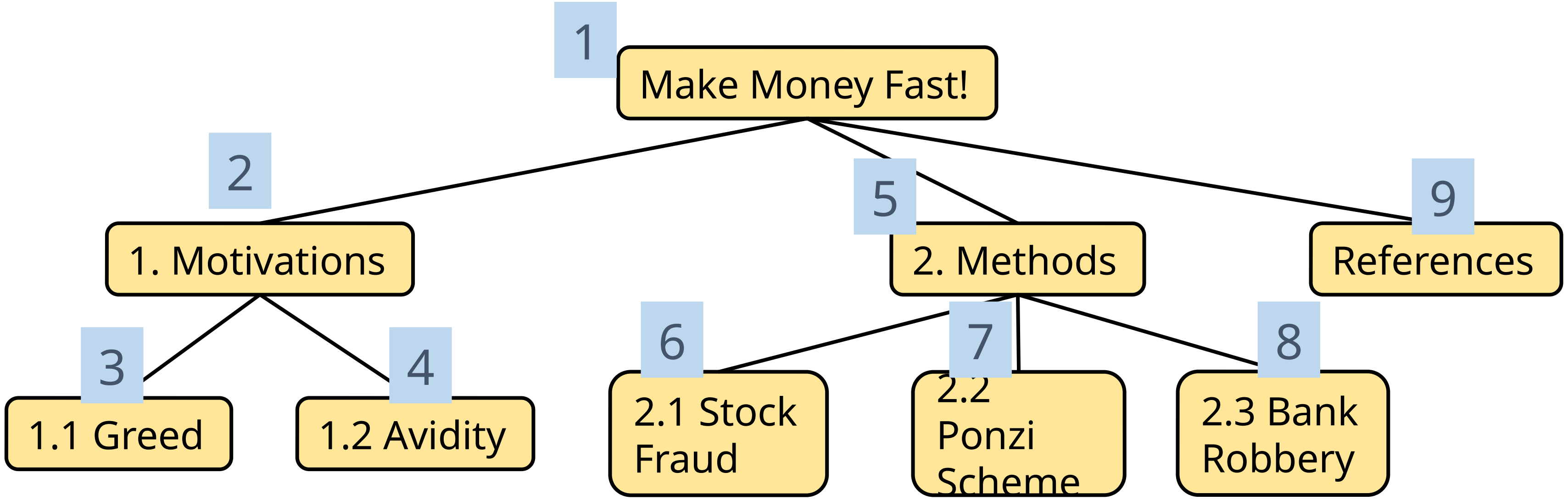
GEBZE TECHNICAL UNIVERSITY

```
class BinaryTree(Tree):
  """Abstract base class representing a binary tree structure."""

  # --------------------- additional abstract methods ---------------------
  def left(self, p):
    """Return a Position representing p's left child.

    Return None if p does not have a left child.
    """
    raise NotImplementedError('must be implemented by subclass')

  def right(self, p):
    """Return a Position representing p's right child.

    Return None if p does not have a right child.
    """
    raise NotImplementedError('must be implemented by subclass')

  # ---------- concrete methods implemented in this class ----------
  def sibling(self, p):
    """Return a Position representing p's sibling (or None if no sibling)."""
    parent = self.parent(p)
    if parent is None:                    # p must be the root
      return None                         # root has no sibling
    else:
      if p == self.left(parent):
        return self.right(parent)         # possibly None
      else:
        return self.left(parent)          # possibly None

  def children(self, p):
    """Generate an iteration of Positions representing p's children."""
    if self.left(p) is not None:
      yield self.left(p)
    if self.right(p) is not None:
      yield self.right(p)
```

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

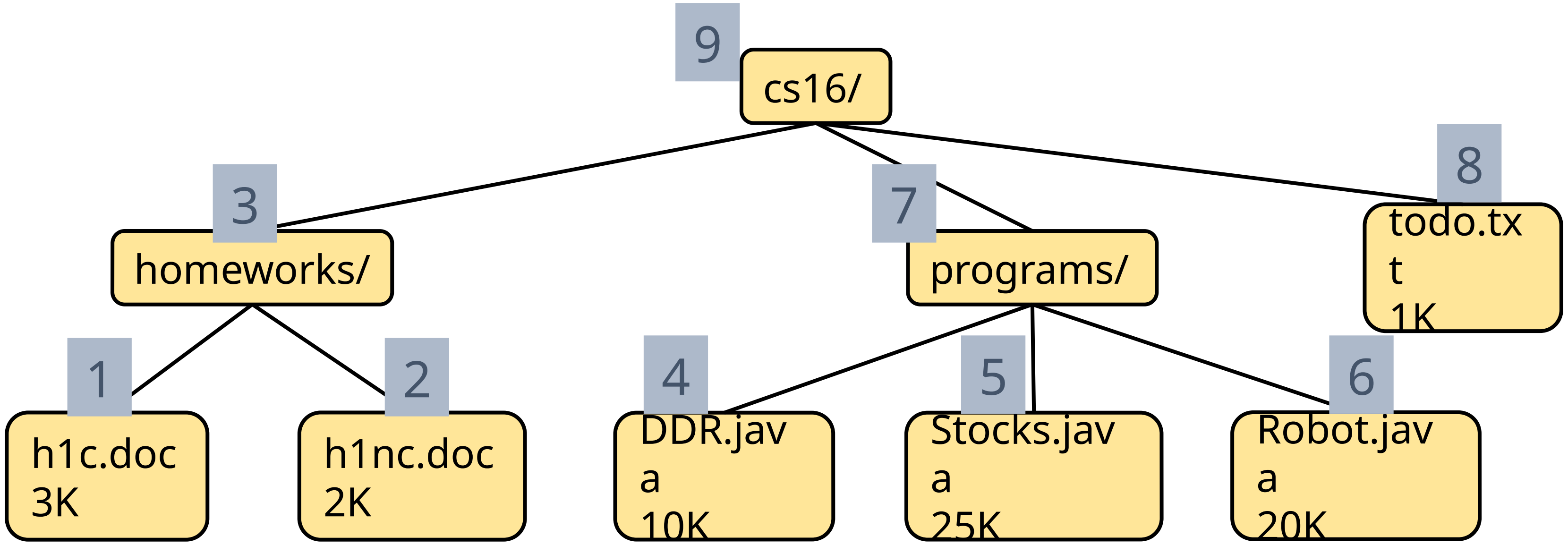
Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

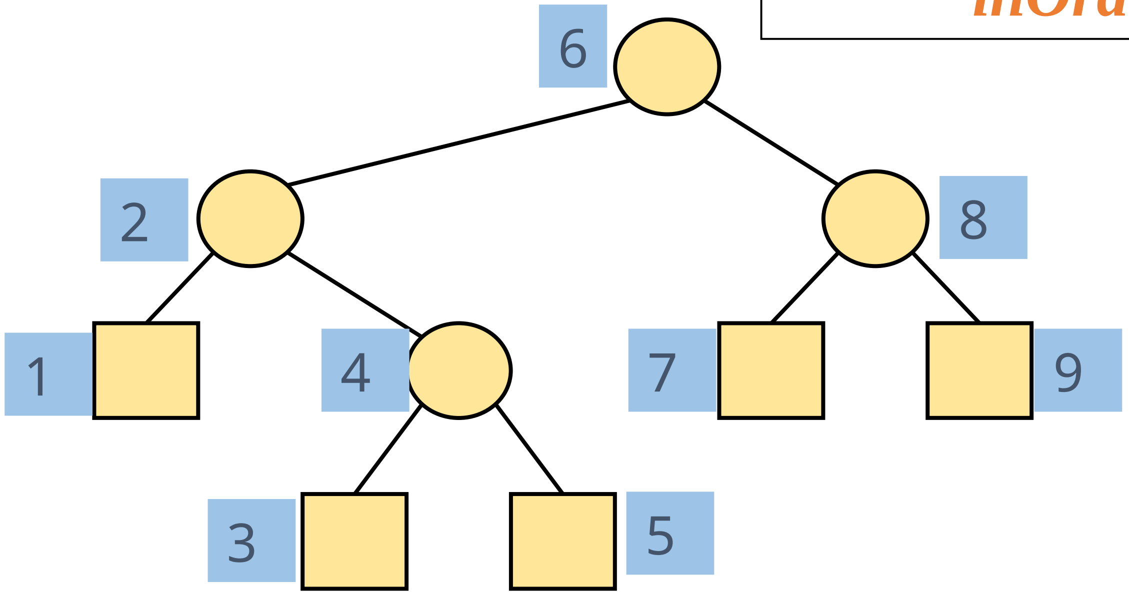
Algorithm *postOrder*(*v*)
    for each child *w* of *v*
        postOrder (*w*)
    visit(*v*)



Inorder Traversal

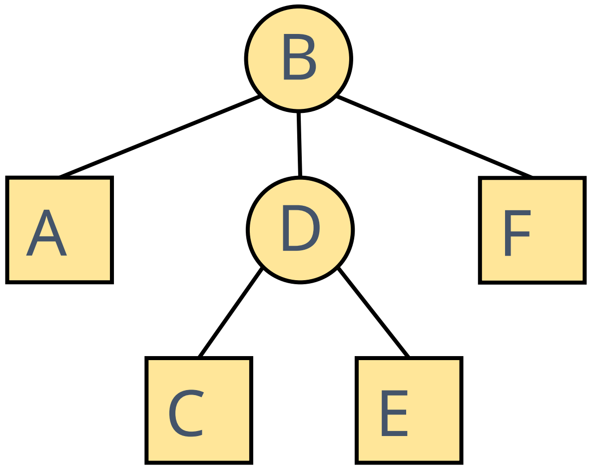
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

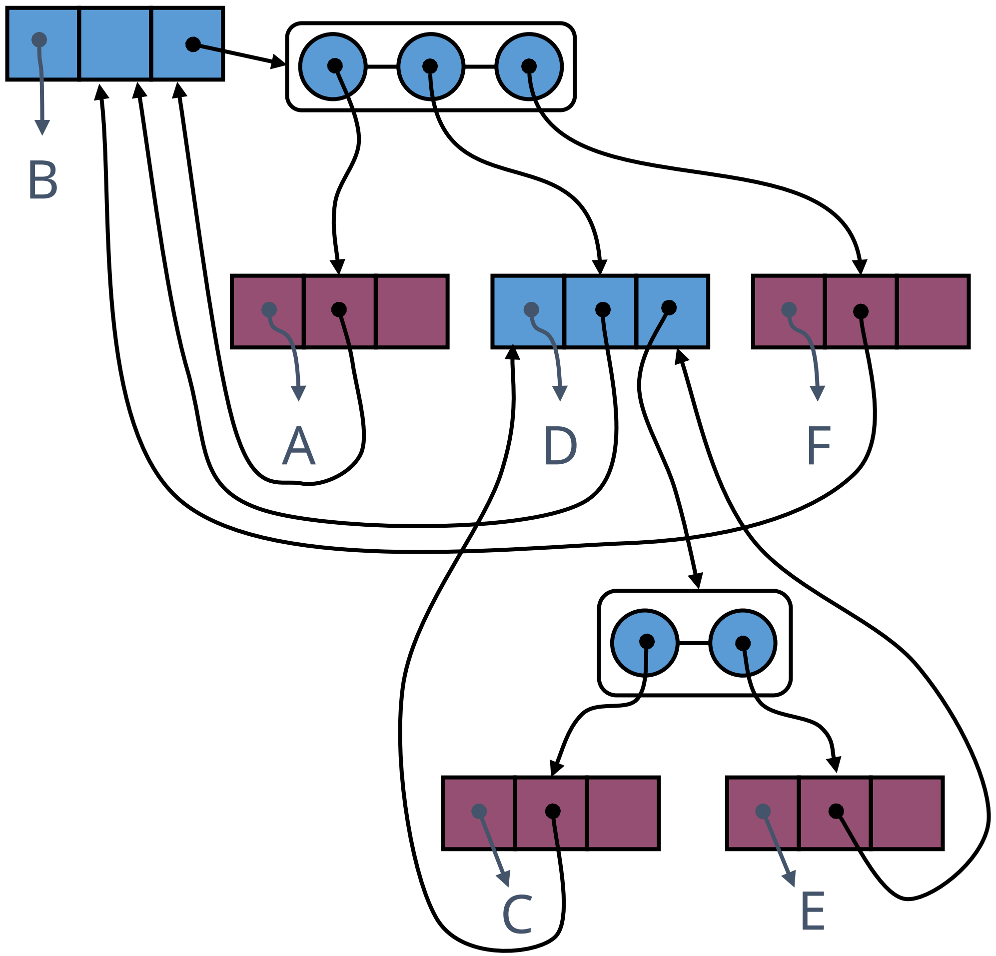
Algorithm *inOrder*(*v*)
    if *v* **has a left child**
        inOrder (*left* (*v*))
    visit(*v*)
    if *v* **has a right child**
        inOrder (*right* (*v*))



Linked Structure for Trees

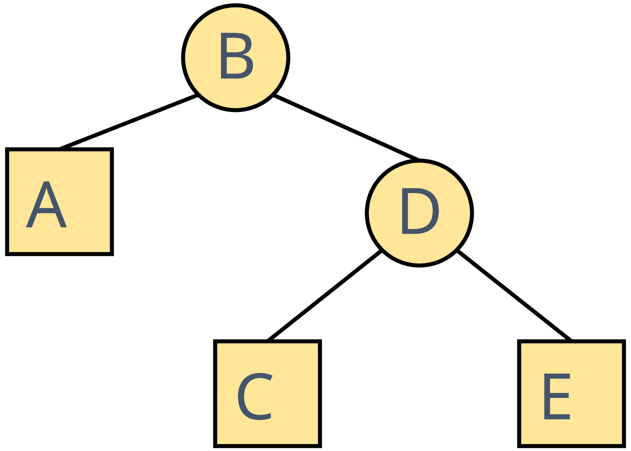
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes

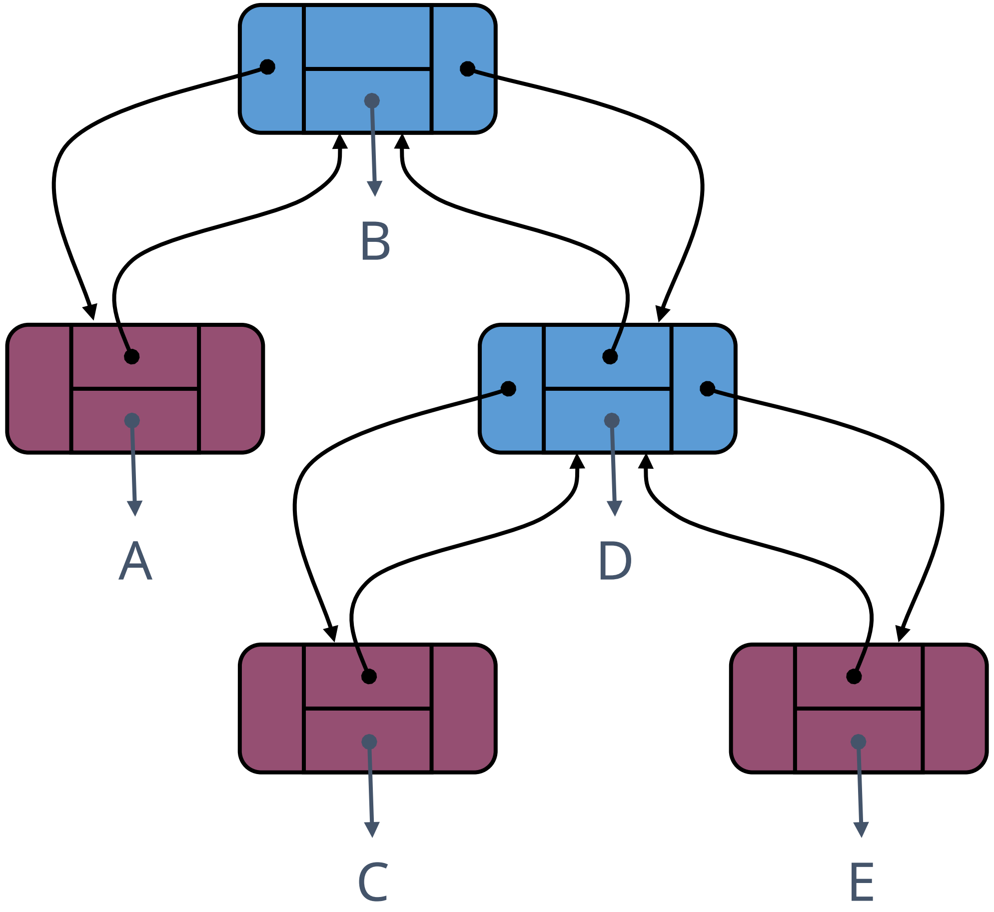




Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node





Binary Search Trees

- A Binary Search Tree is a binary tree data structure with the following properties:
  - Binary Tree Structure: It is a binary tree, meaning each node has at most two children: a left child and a right child.
  - Ordering Property: For every node n, all nodes in its left subtree have values less than n, and all nodes in its right subtree have values greater than n.

- This property ensures that the tree maintains a specific ordering, making it suitable for efficient searching.

Tree Traversal

- The process of visiting each node in a tree exactly once.
- Fundamental for accessing and processing data stored in a tree.

- Use cases:
  - Search / Data retrieval
  - Tree operations (insert, delete, update)
  - Serialization / deserialization of a tree
  - Backtracking (puzzle, maze solving)

Depth-First Search

- Traverse a tree such that the deepest node is visited and then backtrack to its parent node if no sibling of that node exists.

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

- https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

- Application of DFS Algorithm
  - For finding the path
  - To test if the graph is bipartite
  - For finding the strongly connected components of a graph
  - For detecting cycles in a graph

- https://www.programiz.com/dsa/graph-dfs

Breath-First Search

- Traverse a Tree such that all nodes present in the same level are traversed completely before traversing the next level.
- Naive solution: find the height of the tree and traversing each level and printing the nodes of that level.
- Efficient solution: Use a queue.
  - Start at the root node and add it to a queue.
  - While the queue is not empty, dequeue a node and visit it.
  - Enqueue all of its children (if any) into the queue.
  - Repeat steps 2 and 3 until the queue is empty.
- https://www.geeksforgeeks.org/level-order-tree-traversal/

- BFS Algorithm Applications
  - To build index by search index
  - For GPS navigation
  - Path finding algorithms
  - In Ford-Fulkerson algorithm to find maximum flow in a network
  - Cycle detection in an undirected graph
  - In minimum spanning tree

- https://www.programiz.com/dsa/graph-bfs

DFS vs BFS

	Depth First Search (DFS)	Breadth First Search (BFS)
Data Structure	DFS uses Stack data structure.	BFS uses Queue data structure
Definition	DFS traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.
Conceptual Difference	DFS builds the tree sub-tree by sub-tree.	BFS builds the tree level by level.
Approach used	It works on the concept of LIFO (Last In First Out).	It works on the concept of FIFO (First In First Out).
Suitable for	DFS is more suitable when there are solutions away from source.	BFS is more suitable for searching vertices closer to the given source.
Applications	DFS is used in various applications such as acyclic graphs and finding strongly connected components etc.	BFS is used in various applications such as bipartite graphs, shortest paths, etc.

ENG 346
Data Structures and Algorithms for Artificial Intelligence
Trees

Dr. Mehmet PEKMEZCİ

mpekmezci@gtu.edu.tr

https://github.com/mehmetpekmezci/GTU-ENG-346

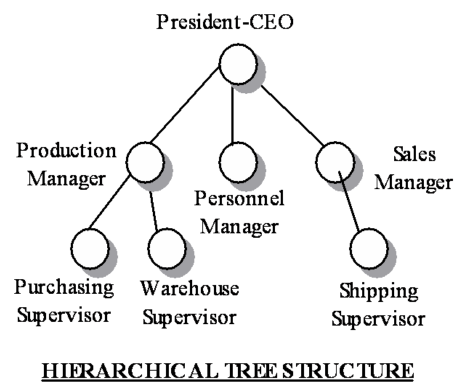
ENG-346 Teams code is **0uv7jlm**

Agenda

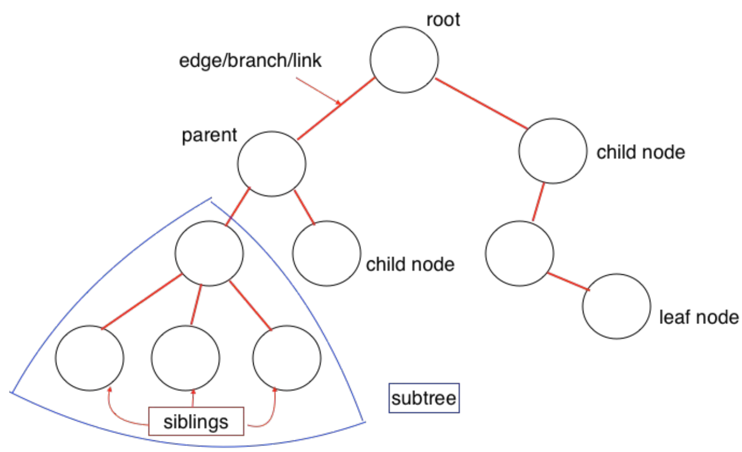
- Basic Concepts

What is a Tree?

- A *tree* is a *hierarchical* data structure consisting of *nodes* connected by *edges*, with a designated *root node* and *branches* that form a directed acyclic graph.



Definitions



ENG 346 – Data Structures and Algorithms for Artificial Intelligence

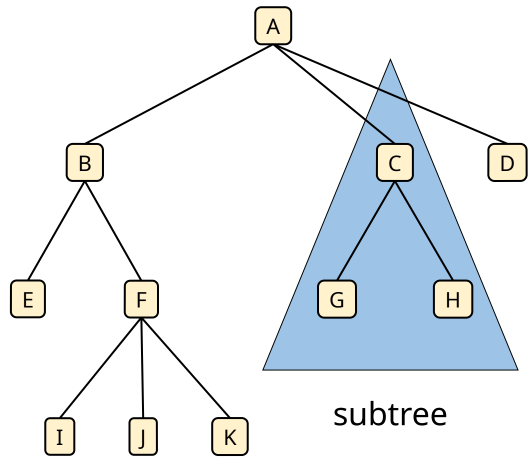
Node:
A fundamental building block of a tree.
Contains data or information.
May have zero or more child nodes.

Edge:
Connects two nodes, representing a relationship between them.
In a tree, edges depict the hierarchical connection between parent and child nodes.

Root:
The topmost node in a tree.
A tree has exactly one root.
All other nodes are descendants of the root.

Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Subtree: tree consisting of a node and its descendants



Applications of Trees

- Organization Charts
- Decision Trees in Machine Learning
- File Systems
- Expression Parsing
- DOM Model of Web pages

Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - Integer len()
  - Boolean is_empty()
  - Iterator positions()
  - Iterator iter()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterator children(p)
  - Integer num_children(p)

- Query methods:
  - Boolean is_leaf(p)
  - Boolean is_root(p)
- Update method:
  - element replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Tree ADT

- Assume T is a tree, and p is a position (node of the tree).

Functions/Operations	Description
p.element()	Data stored in p
Accessors	
T.root()	Return the position of the root of the tree. None is T is empty.
T.is_root(p)	True if p is the root, False otherwise.
T.parent(p)	Return the parent of position p. None if p is the root.
T.children(p)	Return list of children of position p.
T.num_children(p)	Number of children of position p.
Queries	
T.is_leaf(p)	True if position p has no children.
T.is_root(p)	True if position p is the root.

Abstract Tree Class

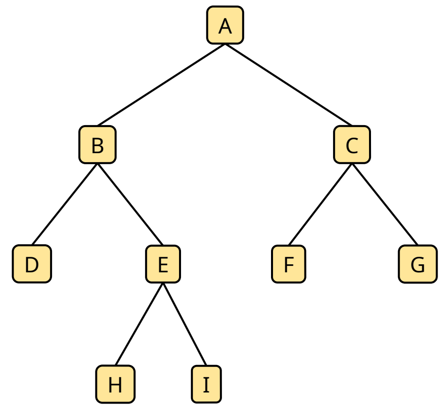
```
class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------ nested Position class ------------------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def element(self):
      """Return the element stored at this Position."""
      raise NotImplementedErr

    def __eq__(self, other):
      """Return True if other P
      raise NotImplementedErr

    def __ne__(self, other):
      """Return True if other d
      return not (self == oth
```

```
  # ---------- abstract methods that concrete subclass must support ----------
  def root(self):
    """Return Position representing the tree's root (or None if empty)."""
    raise NotImplementedError('must be implemented by subclass')

  def parent(self, p):
    """Return Position representing p's parent (or None if p is root)."""
    raise NotImplementedError('must be implemented by subclass')

  def num_children(self, p):
                                        ition p has."""
                                        lemented by subclass')

                                        senting p's children."""
                                        lemented by subclass')

                                        the tree."""
                                        lemented by subclass')
```

```
  # ---------- concrete methods implemented in this class ----------
  def is_root(self, p):
    """Return True if Position p represents the root of the tree."""
    return self.root( ) == p

  def is_leaf(self, p):
    """Return True if Position p does not have any children."""
    return self.num_children(p) == 0

  def is_empty(self):
    """Return True if the tree is empty."""
    return len(self) == 0
```

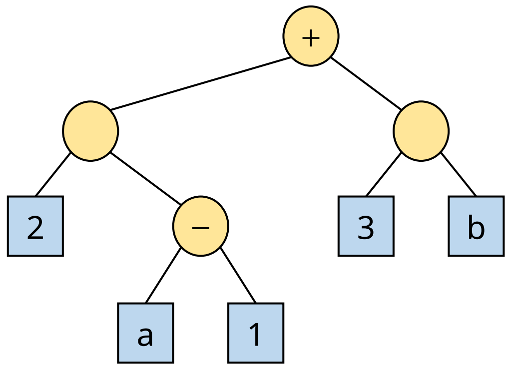
Binary Trees

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children
  - The children of a node are an ordered pair
- The children are called left child and right child
- Applications
  - arithmetic expressions
  - decision processes
  - searching



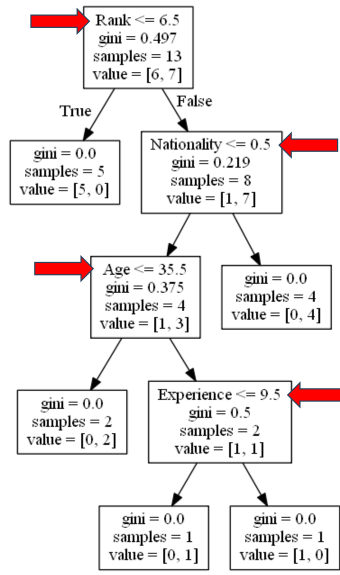
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - Internal nodes: operators
  - Leaf nodes: operands
- Example: arithmetic expression tree for the expression (2   (a – 1) + (3   b))



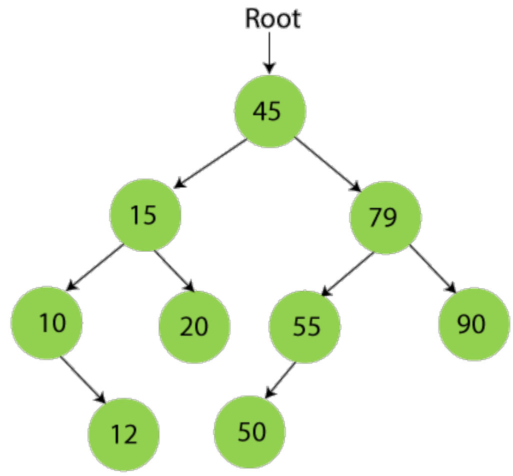
Decision Tree

- Binary tree associated with a decision process
  - Internal nodes: questions with yes/no answer
  - Leaf nodes: decisions



Searching

- Binary search tree



Binary Tree ADT

- The Binary Tree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT

- Additional methods:
  - position left(p)
  - position right(p)
  - position sibling(p)

Abstract BinaryTree Class

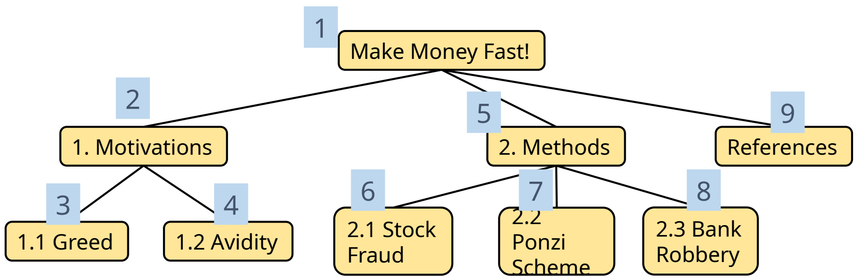
```
class BinaryTree(Tree):
  """Abstract base class representing a binary tree structure."""

  # -------------------- additional abstract methods --------------------
  def left(self, p):
    """Return a Position representing p's left child.

    Return None if p does not have a left child.
    """
    raise NotImplementedError('must be implemented by subclass')

  def right(self, p):
    """Return a Position representing p's right child.

    Return None if p does not have a right child.
    """
    raise NotImplementedError('must be implemented by subclass')
```

```
  # ---------- concrete methods implemented in this class ----------
  def sibling(self, p):
    """Return a Position representing p's sibling (or None if no sibling)."""
    parent = self.parent(p)
    if parent is None:                    # p must be the root
      return None                         # root has no sibling
    else:
      if p == self.left(parent):
        return self.right(parent)         # possibly None
      else:
        return self.left(parent)          # possibly None

  def children(self, p):
    """Generate an iteration of Positions representing p's children."""
    if self.left(p) is not None:
      yield self.left(p)
    if self.right(p) is not None:
      yield self.right(p)
```

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

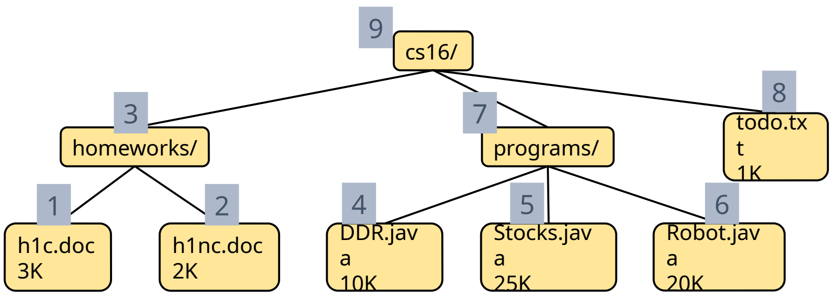
Algorithm *preOrder*(v)
    visit(v)
    for each child w of v
        preorder (w)



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

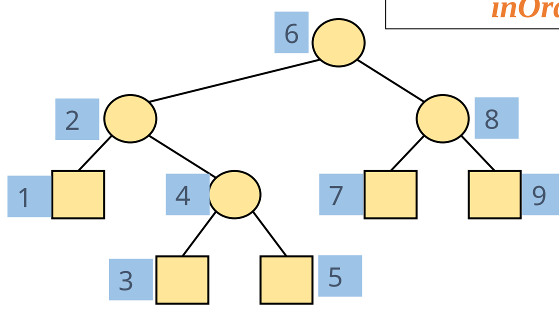
Algorithm *postOrder*(v)
  for each child w of v
    postOrder (w)
  visit(v)



Inorder Traversal

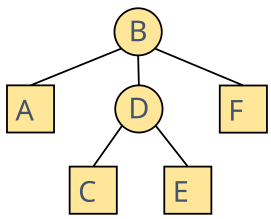
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

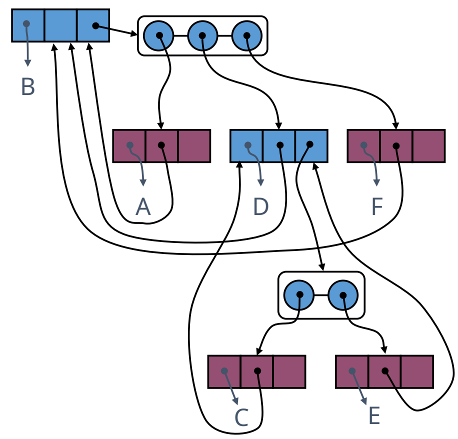
Algorithm *inOrder*(*v*)
  if *v* **has a left child**
    inOrder (*left* (*v*))
  visit(*v*)
  if *v* **has a right child**
    inOrder (*right* (*v*))



Linked Structure for Trees

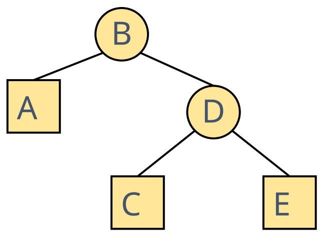
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes

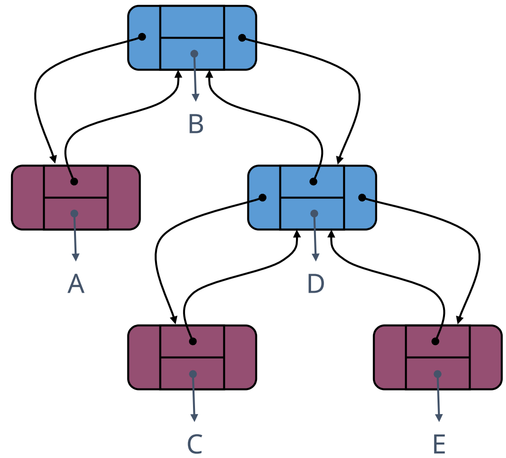




Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node





Binary Search Trees

- A Binary Search Tree is a binary tree data structure with the following properties:
  - Binary Tree Structure: It is a binary tree, meaning each node has at most two children: a left child and a right child.
  - Ordering Property: For every node n, all nodes in its left subtree have values less than n, and all nodes in its right subtree have values greater than n.

- This property ensures that the tree maintains a specific ordering, making it suitable for efficient searching.

Tree Traversal

- The process of visiting each node in a tree exactly once.
- Fundamental for accessing and processing data stored in a tree.

- Use cases:
  - Search / Data retrieval
  - Tree operations (insert, delete, update)
  - Serialization / deserialization of a tree
  - Backtracking (puzzle, maze solving)

Depth-First Search

- Traverse a tree such that the deepest node is visited and then backtrack to its parent node if no sibling of that node exists.

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

- https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

- Application of DFS Algorithm
    - For finding the path
    - To test if the graph is bipartite
    - For finding the strongly connected components of a graph
    - For detecting cycles in a graph

- https://www.programiz.com/dsa/graph-dfs

Breath-First Search

- Traverse a Tree such that all nodes present in the same level are traversed completely before traversing the next level.
- Naive solution: find the height of the tree and traversing each level and printing the nodes of that level.
- Efficient solution: Use a queue.
  - Start at the root node and add it to a queue.
  - While the queue is not empty, dequeue a node and visit it.
  - Enqueue all of its children (if any) into the queue.
  - Repeat steps 2 and 3 until the queue is empty.
- https://www.geeksforgeeks.org/level-order-tree-traversal/

- BFS Algorithm Applications
  - To build index by search index
  - For GPS navigation
  - Path finding algorithms
  - In Ford-Fulkerson algorithm to find maximum flow in a network
  - Cycle detection in an undirected graph
  - In minimum spanning tree
- https://www.programiz.com/dsa/graph-bfs

DFS vs BFS

	Depth First Search (DFS)	Breadth First Search (BFS)
Data Structure	DFS uses Stack data structure.	BFS uses Queue data structure
Definition	DFS traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.
Conceptual Difference	DFS builds the tree sub-tree by sub-tree.	BFS builds the tree level by level.
Approach used	It works on the concept of LIFO (Last In First Out).	It works on the concept of FIFO (First In First Out).
Suitable for	DFS is more suitable when there are solutions away from source.	BFS is more suitable for searching vertices closer to the given source.
Applications	DFS is used in various applications such as acyclic graphs and finding strongly connected components etc.	BFS is used in various applications such as bipartite graphs, shortest paths, etc.