

General Trees and Binary Trees



EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

Learning Outcomes of this Lecture



This module is designed to help you understand:

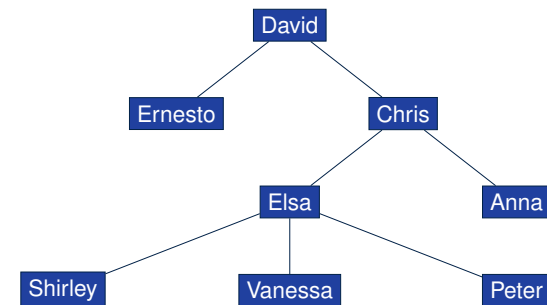
- **Linear** DS (e.g., arrays, LLs) vs. **Non-Linear** DS (e.g., trees)
- Terminologies: **General** Trees vs. **Binary** Trees
- Implementation of a **Generic** Tree
- Mathematical Properties of Binary Trees
- Tree **Traversals**

General Trees



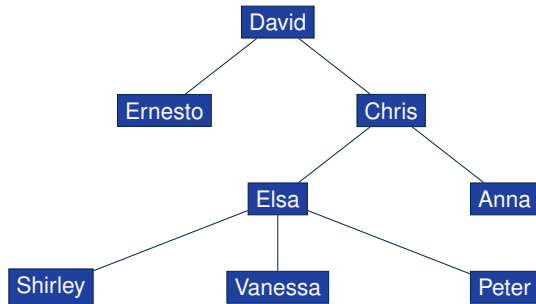
- A **linear** data structure is a sequence, where stored objects can be related via notions of “**predecessor**” and “**successor**”.
 - e.g., arrays
 - e.g., Singly-Linked Lists (SLLs)
 - e.g., Doubly-Linked Lists (DLLs)
- The **Tree ADT** is a **non-linear** collection of nodes/positions.
 - Each node stores some data object.
 - **Nodes** in a **tree** are organized into **levels**: some nodes are “**above**” others, and some are “**below**” others.
 - Think of a **tree** forming a **hierarchy** among the stored **nodes**.
- Terminology of the **Tree ADT** borrows that of **family trees**:
 - e.g., root
 - e.g., parents, siblings, children
 - e.g., ancestors, descendants

General Trees: Terminology (1)



- **top** element of the tree
e.g., root of the above family tree: David [**root of tree**]
- **the** node **immediately above** node *n*
e.g., parent of Vanessa: Elsa [**parent of n**]
- **all** nodes **immediately below** node *n*
e.g., children of Elsa: Shirley, Vanessa, and Peter [**children of n**]
e.g., children of Ernesto: ∅

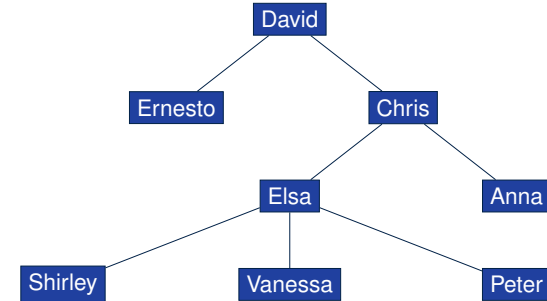
General Trees: Terminology (2)



- Union of n , n 's **parent**, n 's **grand parent**, ..., **root** [n 's **ancestors**]
e.g., ancestors of Vanessa: Vanessa, Elsa, Chris, and David
e.g., ancestors of David: David
- Union of n , n 's **children**, n 's **grand children**, ... [n 's **descendants**]
e.g., descendants of Vanessa: Vanessa
e.g., descendants of David: the entire family tree
- By the above definitions, a **node** is both its **ancestor** and **descendant**.

5 of 47

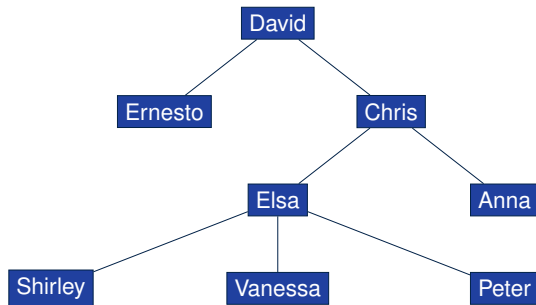
General Trees: Terminology (4)



- a pair of **parent** and **child** nodes [**an edge of tree**]
e.g., (David, Chris), (Chris, Elsa), (Elsa, Peter) are three edges
- a **sequence** of nodes where any two consecutive nodes form an **edge** [**a path of tree**]
e.g., (David, Chris, Elsa, Peter) is a path
e.g., Elsa's **ancestor path**: (Elsa, Chris, David)

7 of 47

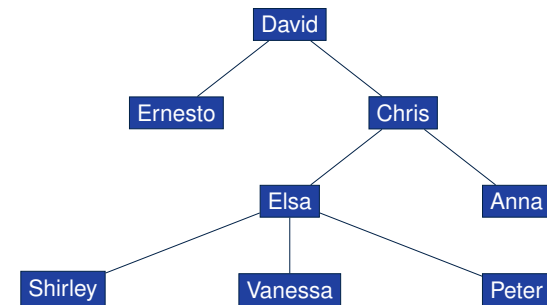
General Trees: Terminology (3)



- all** nodes with the **same parent** as n 's [**siblings of node n**]
e.g., siblings of Vanessa: Shirley and Peter
- the** tree formed by **descendants** of n [**subtree rooted at n**]
- nodes with **no children** [**external nodes (leaves)**]
e.g., leaves of the above tree: Ernesto, Anna, Shirley, Vanessa, Peter
- nodes with **at least one child** [**internal nodes**]
e.g., non-leaves of the above tree: David, Chris, Elsa

6 of 47

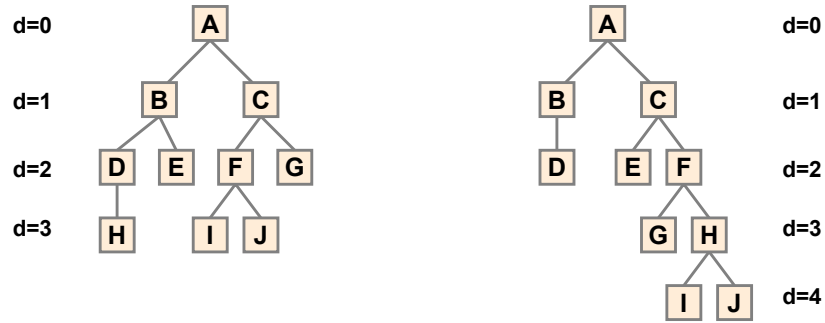
General Trees: Terminology (5)



- number of **edges** from the **root** to node n [**depth of n**]
alternatively: number of n 's **ancestors** of n minus one
e.g., depth of David (root): 0
e.g., depth of Shirley, Vanessa, and Peter: 3
- maximum depth** among all nodes [**height of tree**]
e.g., Shirley, Vanessa, and Peter have the maximum depth

8 of 47

General Trees: Example Node Depths

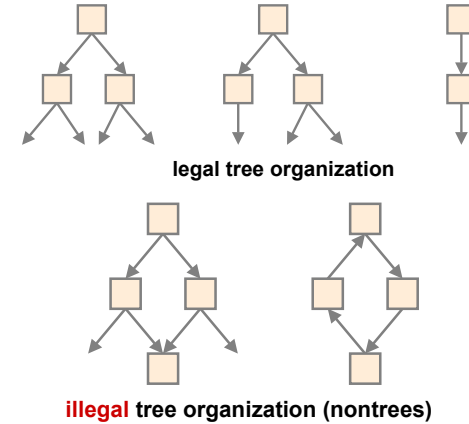


9 of 47

General Tree: Important Characteristics



There is a **single, unique path** from the **root** to any particular node in the same tree.



11 of 47

General Tree: Definition



A **tree T** is a set of **nodes** satisfying **parent-child** properties:

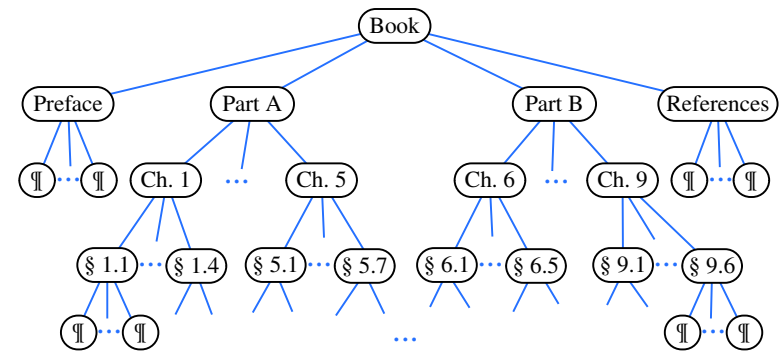
1. If T is **empty**, then it does not contain any nodes.
2. If T is **nonempty**, then:
 - T contains at least its **root** (a special node with no parent).
 - Each node n of T that is not the root has a **unique parent node w** .
 - Given two nodes n and w , if w is the **parent** of n , then **symmetrically**, n is one of w 's **children**.

10 of 47

General Trees: Ordered Trees



A tree is **ordered** if there is a meaningful **linear order** among the **children** of each **internal node**.

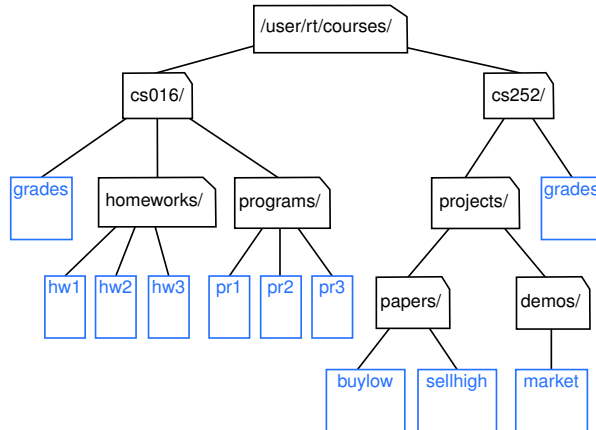


12 of 47

General Trees: Unordered Trees



A tree is **unordered** if the order among the **children** of each **internal node** does **not** matter.



13 of 47

Implementation: Generic Tree Nodes (2)



```
public class TreeNode<E> {
    private E element; /* data object */
    private TreeNode<E> parent; /* unique parent node */
    private TreeNode<E>[] children; /* list of child nodes */

    private final int MAX_NUM_CHILDREN = 10; /* fixed max */
    private int noc; /* number of child nodes */

    public E getElement() { ... }
    public TreeNode<E> getParent() { ... }
    public TreeNode<E>[] getChildren() { ... }

    public void setElement(E element) { ... }
    public void setParent(TreeNode<E> parent) { ... }
    public void addChild(TreeNode<E> child) { ... }
    public void removeChildAt(int i) { ... }
}
```

Exercise: Implement void removeChildAt(int i).

15 of 47

Implementation: Generic Tree Nodes (1)



```
1 public class TreeNode<E> {
2     private E element; /* data object */
3     private TreeNode<E> parent; /* unique parent node */
4     private TreeNode<E>[] children; /* list of child nodes */
5
6     private final int MAX_NUM_CHILDREN = 10; /* fixed max */
7     private int noc; /* number of child nodes */
8
9     public TreeNode(E element) {
10        this.element = element;
11        this.parent = null;
12        this.children = (TreeNode<E>[])
13            Array.newInstance(this.getClass(), MAX_NUM_CHILDREN);
14        this.noc = 0;
15    }
16    ...
17 }
```

Replacing L13 with the following results in a **ClassCastException**:

```
this.children = (TreeNode<E>[]) new Object[MAX_NUM_CHILDREN];
```

14 of 47

Testing: Connected Tree Nodes



Constructing a **tree** is similar to constructing a **SLL**:

```
@Test
public void test_general_trees_construction() {
    TreeNode<String> agnarr = new TreeNode<>("Agnarr");
    TreeNode<String> elsa = new TreeNode<>("Elsa");
    TreeNode<String> anna = new TreeNode<>("Anna");

    agnarr.addChild(elsa);
    agnarr.addChild(anna);
    elsa.setParent(agnarr);
    anna.setParent(agnarr);

    assertNull(agnarr.getParent());
    assertTrue(agnarr == elsa.getParent());
    assertTrue(agnarr == anna.getParent());
    assertTrue(agnarr.getChildren().length == 2);
    assertTrue(agnarr.getChildren()[0] == elsa);
    assertTrue(agnarr.getChildren()[1] == anna);
}
```

16 of 47

Problem: Computing a Node's Depth

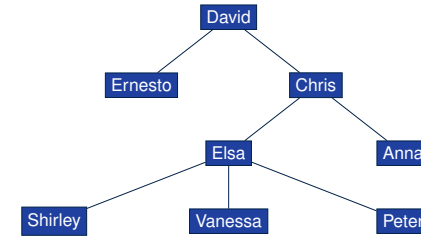


- Given a node n , its **depth** is defined as:
 - If n is the **root**, then n 's depth is 0.
 - Otherwise, n 's **depth** is the **depth** of n 's parent plus one.
- Assuming under a **generic** class `TreeUtilities<E>`:

```
1 public int depth(TreeNode<E> n) {
2     if(n.getParent() == null) {
3         return 0;
4     }
5     else {
6         return 1 + depth(n.getParent());
7     }
8 }
```

17 of 47

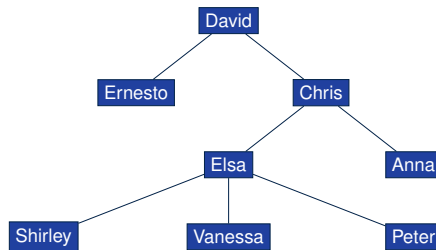
Unfolding: Computing a Node's Depth



```
depth(vanessa)
= { vanessa.getParent() == elsa }
  1 + depth(elsa)
= { elsa.getParent() == chris }
  1 + 1 + depth(chris)
= { chris.getParent() == david }
  1 + 1 + 1 + depth(David)
= { David is the root }
  1 + 1 + 1 + 0
= 3
```

19 of 47

Testing: Computing a Node's Depth



```
@Test
public void test_general_trees_depths() {
    ... /* constructing a tree as shown above */
    TreeUtilities<String> u = new TreeUtilities<>();
    assertEquals(0, u.depth(david));
    assertEquals(1, u.depth(ernesto));
    assertEquals(1, u.depth(chris));
    assertEquals(2, u.depth(elsa));
    assertEquals(2, u.depth(anna));
    assertEquals(3, u.depth(shirley));
    assertEquals(3, u.depth(vanessa));
    assertEquals(3, u.depth(peter));
}
```

18 of 47

Problem: Computing a Tree's Height

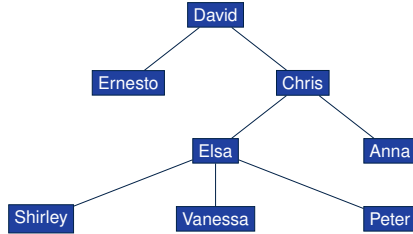


- Given node n , the **height** of subtree rooted at n is defined as:
 - If n is a **leaf**, then the **height** of subtree rooted at n is 0.
 - Otherwise, the height of subtree rooted at n is one plus the **maximum height** of **all** subtrees rooted at n 's children.
- Assuming under a **generic** class `TreeUtilities<E>`:

```
1 public int height(TreeNode<E> n) {
2     TreeNode<E>[] children = n.getChildren();
3     if(children.length == 0) { return 0; }
4     else {
5         int max = 0;
6         for(int i = 0; i < children.length; i++) {
7             int h = 1 + height(children[i]);
8             max = h > max ? h : max;
9         }
10        return max;
11    }
12 }
```

20 of 47

Testing: Computing a Tree's Height



```

@Test
public void test_general_trees_heights() {
    ... /* constructing a tree as shown above */
    TreeUtilities<String> u = new TreeUtilities<>();
    /* internal nodes */
    assertEquals(3, u.height(david));
    assertEquals(2, u.height(chris));
    assertEquals(1, u.height(elsa));
    /* external nodes */
    assertEquals(0, u.height(ernesto));
    assertEquals(0, u.height(anna));
    assertEquals(0, u.height(shirley));
    assertEquals(0, u.height(vanessa));
    assertEquals(0, u.height(peter));
}
    
```

21 of 47

Exercises on General Trees



- Implement and test the following *recursive* algorithm:

```
public TreeNode<E>[] ancestors(TreeNode<E> n)
```

which returns the list of *ancestors* of a given node *n*.

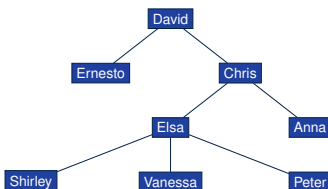
- Implement and test the following *recursive* algorithm:

```
public TreeNode<E>[] descendants(TreeNode<E> n)
```

which returns the list of *descendants* of a given node *n*.

23 of 47

Unfolding: Computing a Tree's Height



$$\begin{aligned}
 & \text{height}(\text{subtree rooted at chris}) \\
 = & \{ \text{chris is not a leaf} \} \\
 & \text{MAX} \left(1 + \text{height}(\text{subtree rooted at elsa}), \right. \\
 & \quad \left. 1 + \text{height}(\text{subtree rooted at anna}) \right) \\
 = & \{ \text{elsa is not a leaf, anna is a leaf} \} \\
 & \text{MAX} \left(1 + \text{MAX} \left(1 + \text{height}(\text{subtree rooted at shirley}), \right. \right. \\
 & \quad \left. \left. 1 + \text{height}(\text{subtree rooted at vanessa}), \right. \right. \\
 & \quad \left. \left. 1 + \text{height}(\text{subtree rooted at peter}) \right), \right) \\
 = & \{ \text{shirley, vanessa, and peter are all leaves} \} \\
 & \text{MAX} \left(1 + \text{MAX} \left(\begin{matrix} 1+0, \\ 1+0, \\ 1+0 \end{matrix} \right), \right) \\
 = & 2
 \end{aligned}$$

22 of 47

Binary Trees (BTs): Definitions



A *binary tree (BT)* is an *ordered tree* satisfying the following:

- Each node has **at most two** (≤ 2) children.
- Each *child node* is labeled as either a *left child* or a *right child*.
- A *left child* precedes a *right child*.

A *binary tree (BT)* is either:

- An *empty tree*; or
- A *nonempty tree* with a *root node* *r* which has:
 - a *left subtree* rooted at its *left child*, if any
 - a *right subtree* rooted at its *right child*, if any

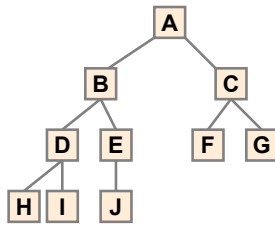
24 of 47

BT Terminology: LST vs. RST



For an **internal** node (with at least one child):

- Subtree rooted at its **left child**, if any, is called **left subtree**.
 - Subtree rooted at its **right child**, if any, is called **right subtree**.
- e.g.,



Node A has:

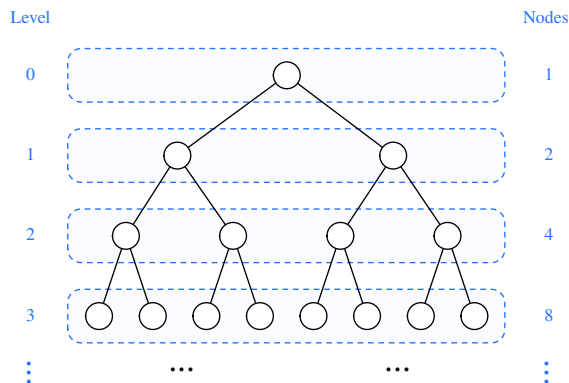
- a **left subtree** rooted at node B
- a **right subtree** rooted at node C

25 of 47

BT Terminology: Depths, Levels



The set of nodes with the same **depth** d are said to be at the same **level** d .



26 of 47

Background: Sum of Geometric Sequence



- Given a **geometric sequence** of n terms, where the initial term is a and the common factor is r , the **sum** of all its terms is:

$$\sum_{k=0}^{n-1} (a \cdot r^k) = a \cdot r^0 + a \cdot r^1 + a \cdot r^2 + \dots + a \cdot r^{n-1} = a \cdot \left(\frac{r^n - 1}{r - 1} \right)$$

[See [here](#) to see how the formula is derived.]

- For the purpose of **binary trees**, **maximum** numbers of nodes at all **levels** form a **geometric sequence**:
 - $a = 1$ [the **root** at **Level 0**]
 - $r = 2$ [≤ 2 children for each **internal node**]
 - e.g., **Max** total # of nodes at **levels** 0 to 4 = $1 + 2 + 4 + 8 + 16 = 1 \cdot \left(\frac{2^5 - 1}{2 - 1} \right) = 31$

27 of 47

BT Properties: Max # Nodes at Levels



Given a **binary tree** with **height** h :

- At each level:
 - **Maximum** number of nodes at **Level 0**: $2^0 = 1$
 - **Maximum** number of nodes at **Level 1**: $2^1 = 2$
 - **Maximum** number of nodes at **Level 2**: $2^2 = 4$
 - ...
 - **Maximum** number of nodes at **Level h**: 2^h
- Summing all levels:

Maximum total number of nodes:

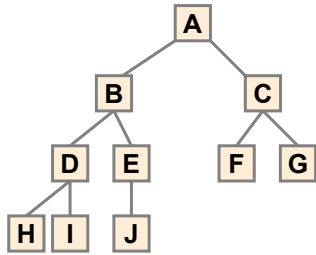
$$\underbrace{2^0 + 2^1 + 2^2 + \dots + 2^h}_{h+1 \text{ terms}} = 1 \cdot \left(\frac{2^{h+1} - 1}{2 - 1} \right) = 2^{h+1} - 1$$

28 of 47

BT Terminology: Complete BTs

A **binary tree** with **height** h is considered as **complete** if:

- Nodes with **depth** $\leq h - 2$ has two children.
- Nodes with **depth** $h - 1$ may have zero, one, or two child nodes.
- **Children** of nodes with **depth** $h - 1$ are filled from left to right.



Q1: **Minimum** # of nodes of a **complete** BT? $(2^h - 1) + 1 = 2^h$

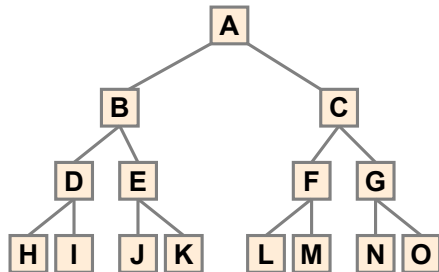
Q2: **Maximum** # of nodes of a **complete** BT? $2^{h+1} - 1$

BT Terminology: Full BTs

A **binary tree** with **height** h is considered as **full** if:

Each node with **depth** $\leq h - 1$ has two child nodes.

That is, all **leaves** are with the same **depth** h .



Q1: **Minimum** # of nodes of a **complete** BT? $2^{h+1} - 1$

Q2: **Maximum** # of nodes of a **complete** BT? $2^{h+1} - 1$

BT Properties: Bounding # of Nodes

Given a **binary tree** with **height** h , the **number of nodes** n is bounded as:

$$h + 1 \leq n \leq 2^{h+1} - 1$$

- Shape of BT with **minimum** # of nodes?
A “one-path” tree (each **internal node** has exactly one child)
- Shape of BT with **maximum** # of nodes?
A tree completely filled at each level

BT Properties: Bounding Height of Tree

Given a **binary tree** with n nodes, the **height** h is bounded as:

$$\log(n + 1) - 1 \leq h \leq n - 1$$

- Shape of BT with **minimum** height?
A tree completely filled at each level

n	=	$2^{h+1} - 1$
$\Leftrightarrow n + 1$	=	2^{h+1}
$\Leftrightarrow \log(n + 1)$	=	$h + 1$
$\Leftrightarrow \log(n + 1) - 1$	=	h
- Shape of BT with **maximum** height?
A “one-path” tree (each **internal node** has exactly one child)

BT Properties: Bounding # of Ext. Nodes



Given a **binary tree** with **height** h , the **number of external nodes** n_E is bounded as:

$$1 \leq n_E \leq 2^h$$

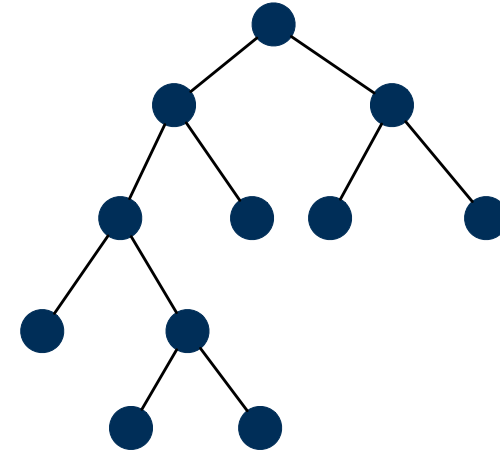
- Shape of BT with **minimum** # of external nodes?
A tree with only one node (i.e., the **root**)
- Shape of BT with **maximum** # of external nodes?
A tree whose bottom level (with **depth** h) is completely filled

33 of 47

BT Terminology: Proper BT



A **binary tree** is **proper** if each **internal node** has two children.



35 of 47

BT Properties: Bounding # of Int. Nodes



Given a **binary tree** with **height** h , the **number of internal nodes** n_I is bounded as:

$$h \leq n_I \leq 2^h - 1$$

- Shape of BT with **minimum** # of internal nodes?
 - Number of nodes in a “one-path” tree ($h + 1$) minus one
 - That is, the “deepest” leaf node excluded
- Shape of BT with **maximum** # of internal nodes?
 - A tree whose $\leq h - 1$ **levels** are all completely filled
 - That is: $\underbrace{2^0 + 2^1 + \dots + 2^{h-1}}_{n \text{ terms}} = 2^h - 1$

34 of 47

BT Properties: #s of Ext. and Int. Nodes



Given a **binary tree** that is:

- **nonempty** and **proper**
- with n_I **internal nodes** and n_E **external nodes**

We can then expect that: $n_E = n_I + 1$

Proof by **mathematical induction**:

• Base Case:

A **proper** BT with only the **root** (an **external node**): $n_E = 1$ and $n_I = 0$.

• Inductive Case:

- Assume a **proper** BT with n nodes ($n > 1$) with n_I **internal nodes** and n_E **external nodes** such that $n_E = n_I + 1$.
- Only one way to create a **larger** BT (with $n + 2$ nodes) that is still **proper** (with n'_E **external nodes** and n'_I **internal nodes**):

Convert an external node into an **internal** node.

$$n'_E = (n_E - 1) + 2 = n_E + 1 \wedge n'_I = n_I + 1 \Rightarrow n'_E = n'_I + 1$$

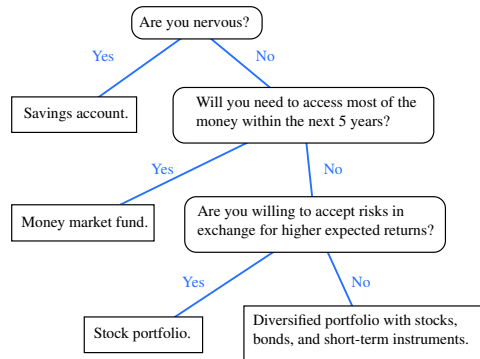
36 of 47

Binary Trees: Application (1)



A **decision tree** is a **proper** binary tree used to express the decision-making process:

- Each **internal node** denotes a **decision point**: yes or no.
- Each **external node** denotes the **consequence** of a decision.



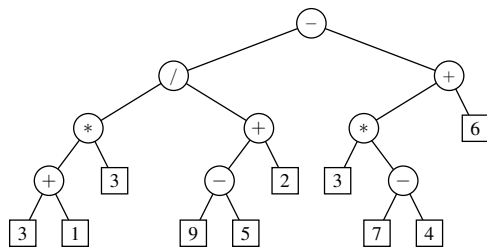
37 of 47

Binary Trees: Application (2)



An **infix arithmetic expression** can be represented using a binary tree:

- Each **internal node** denotes an **operator** (unary or binary).
- Each **external node** denotes an **operand** (i.e., a number).



- To evaluate the expression that is represented by a binary tree, certain **traversal** over the entire tree is required.

38 of 47

Tree Traversal Algorithms: Definition



- A **traversal** of a **tree T** systematically **visits all T's** nodes.
- Visiting each **node** may be associated with an **action**: e.g.,
 - Print** the node element.
 - Determine** if the node element satisfies certain property (e.g., positive, matching a key).
 - Accumulate** the node element values for some **global** result.

39 of 47

Tree Traversal Algorithms: Common Types



Three common traversal orders:

- Preorder**: Visit parent, then visit child subtrees.

```
preorder (n)
  visit and act on position n
  for child c: children(n) { preorder (c) }
```

- Postorder**: Visit child subtrees, then visit parent.

```
postorder (n)
  for child c: children(n) { postorder (c) }
  visit and act on position n
```

- Inorder** (for **BT**): Visit left subtree, then parent, then right subtree.

```
inorder (n)
  if (n has a left child lc) { inorder (lc) }
  visit and act on position n
  if (n has a right child rc) { inorder (rc) }
```

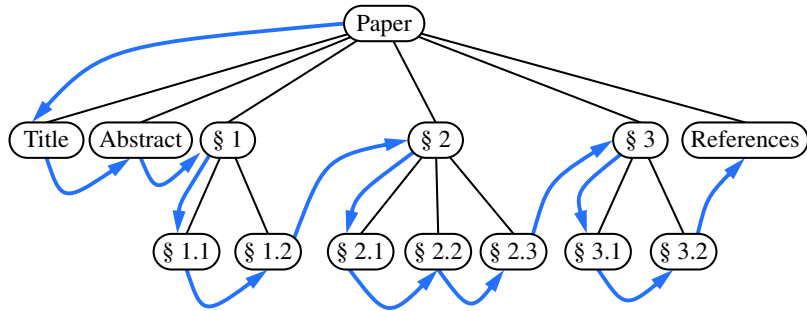
40 of 47

Tree Traversal Algorithms: Preorder



Preorder: Visit parent, then visit child subtrees.

```
preorder(n)
  visit and act on position n
  for child c: children(n) { preorder(c) }
```



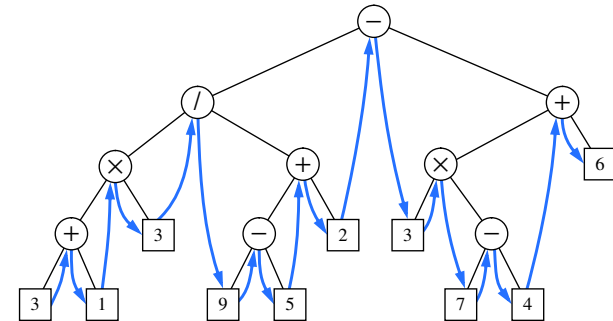
41 of 47

Tree Traversal Algorithms: Inorder



Inorder (for BT): Visit left subtree, then parent, then right subtree.

```
inorder(n)
  if (n has a left child lc) { inorder(lc) }
  visit and act on position n
  if (n has a right child rc) { inorder(rc) }
```



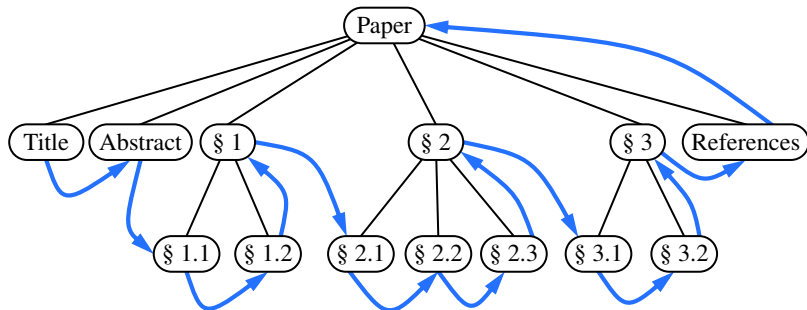
43 of 47

Tree Traversal Algorithms: Postorder



Postorder: Visit child subtrees, then visit parent.

```
postorder(n)
  for child c: children(n) { postorder(c) }
  visit and act on position n
```



42 of 47

Index (1)



Learning Outcomes of this Lecture

General Trees

General Trees: Terminology (1)

General Trees: Terminology (2)

General Trees: Terminology (3)

General Trees: Terminology (4)

General Trees: Terminology (5)

General Trees: Example Node Depths

General Tree: Definition

General Tree: Important Characteristics

General Trees: Ordered Trees

44 of 47



Index (2)

General Trees: Unordered Trees

Implementation: Generic Tree Nodes (1)

Implementation: Generic Tree Nodes (2)

Testing: Connected Tree Nodes

Problem: Computing a Node's Depth

Testing: Computing a Node's Depth

Unfolding: Computing a Node's Depth

Problem: Computing a Tree's Height

Testing: Computing a Tree's Height

Unfolding: Computing a Tree's Height

Exercises on General Trees

45 of 47



Index (3)

Binary Trees (BTs): Definitions

BT Terminology: LST vs. RST

BT Terminology: Depths, Levels

Background: Sum of Geometric Sequence

BT Properties: Max # Nodes at Levels

BT Terminology: Complete BTs

BT Terminology: Full BTs

BT Properties: Bounding # of Nodes

BT Properties: Bounding Height of Tree

BT Properties: Bounding # of Ext. Nodes

BT Properties: Bounding # of Int. Nodes

46 of 47



Index (4)

BT Terminology: Proper BT

BT Properties: #s of Ext. and Int. Nodes

Binary Trees: Application (1)

Binary Trees: Application (2)

Tree Traversal Algorithms: Definition

Tree Traversal Algorithms: Common Types

Tree Traversal Algorithms: Preorder

Tree Traversal Algorithms: Postorder

Tree Traversal Algorithms: Inorder

47 of 47