General Trees and Binary Trees

EECS2011 X:
Fundamentals of Data Structures
Winter 2023

CHEN-WEI WANG

Learning Outcomes of this Lecture

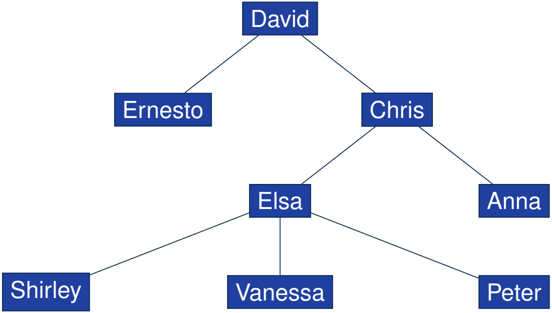
This module is designed to help you understand:

- ***Linar*** 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

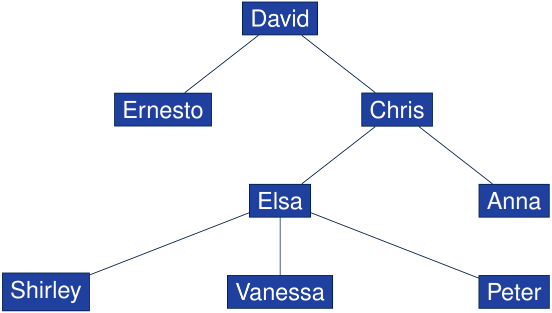
- A ***linear*** data structure is a sequence, where stored objects can be related via notions of "**predecessor**" and "**successor**".
  - e.g., arrays
  - e.g., Singly-Linked Lists (SLLs)
  - e.g., Doubly-Linked Lists (DLLs)
- The *Tree ADT* is a ***non-linear*** collection of nodes/positions.
  - 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)



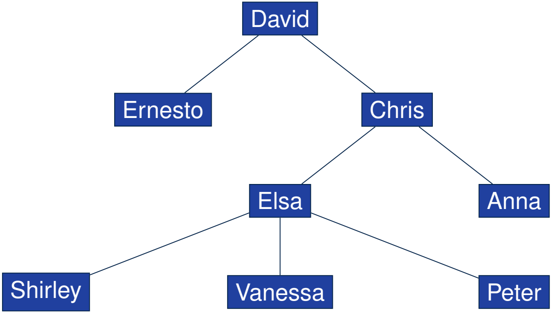
- ***top*** element of the tree [***root of tree***]
  e.g., root of the above family tree: David
- **the** node ***immediately above*** node *n* [***parent of*** *n*]
  e.g., parent of Vanessa: Elsa
- **all** nodes ***immediately below*** node *n* [***children of*** *n*]
  e.g., children of Elsa: Shirley,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

General Trees: Terminology (2)



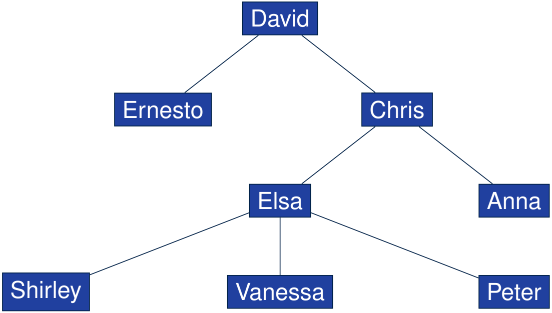
- Union of *n*, *n*'s ***parent***, *n*'s ***grand parent***, . . . , ***root*** [*n's* ***ancestors***]
  e.g., ancestors of Vanessa: Vanessa, Elsa, Chris, and David
  e.g., ancestors of David: David
- Union of *n*, *n*'s ***children***, *n*'s ***grand children***, . . . [*n's* ***descendants***]
  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***.

General Trees: Terminology (3)



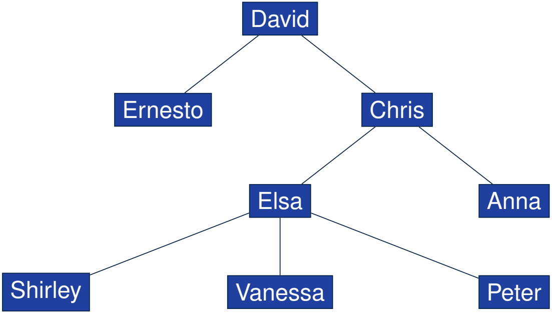
- **all** nodes with the ***same parent*** as *n*'s [***siblings of node*** *n*]
  e.g., siblings of Vanessa: Shirley and Peter
- **the** tree formed by ***descendants*** of *n* [***subtree rooted at*** *n*]
- nodes with ***no children*** [***external nodes (leaves)***]
  e.g., leaves of the above tree: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

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 ⟩

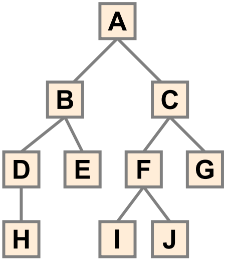
General Trees: Terminology (5)

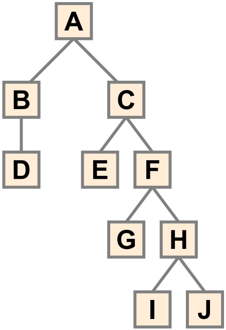


- number of ***edges*** from the ***root*** to node *n* [***depth of*** *n*]
  alternatively: number of *n*'s ***ancestors*** of *n* minus one
  e.g., depth of David (root): 0
  e.g., depth of Shirley, Vanessa, and Peter: 3
- ***maximum depth*** among all nodes [***height of tree***]
  e.g., Shirley,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

General Trees: Example Node Depths

d=0
d=1
d=2
d=3





d=0
d=1
d=2
d=3
d=4

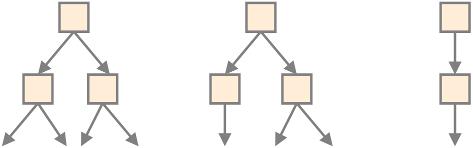
General Tree: Definition

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

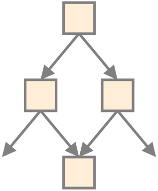
1. If T is ***empty***, then it does not contain any nodes.
2. If T is ***nonempty***, then:
   - T contains at least its ***root*** (a special node with no parent).
   - Each node n of T that is not the root has ***a unique parent node*** w.
   - Given two nodes n and w,
     if w is the ***parent***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

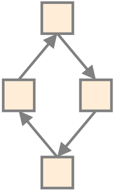
General Tree: Important Characteristics

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



legal tree organization

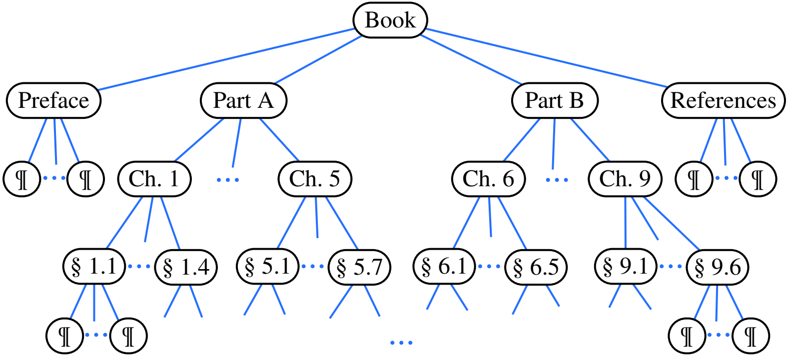




illegal tree organization (nontrees)

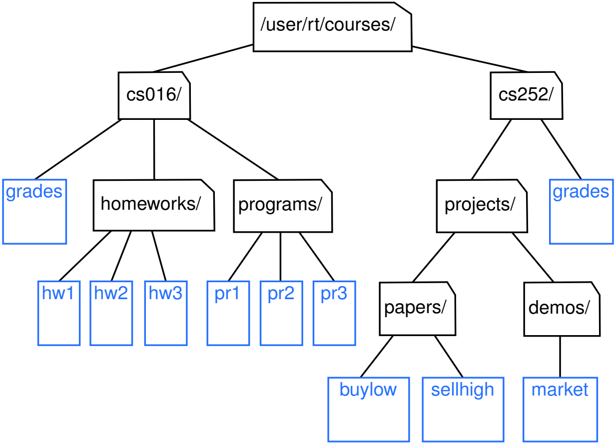
General Trees: Ordered Trees

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



General Trees: Unordered Trees

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



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];
```

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)`.

Testing: Connected Tree Nodes

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

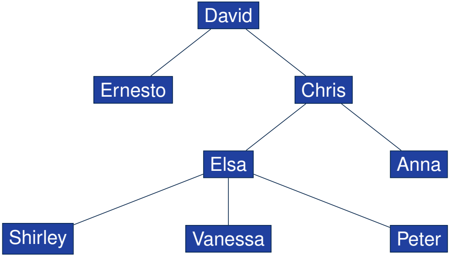
```
@Test
public void test_general_trees_construction() {
  TreeNode<String> agnarr = new TreeNode<>("Agnarr");
  TreeNode<String> elsa = new TreeNode<>("Elsa");
  TreeNode<String> anna = new TreeNode<>("Anna");

  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);
}
```

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>`:

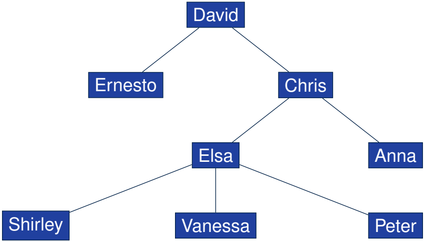
```
public int depth(TreeNode<E> n) {
  if(n.getParent() == null) {
    return 0;
  }
  else {
    return 1 + depth(n.getParent());
  }
}
```

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));
}
```

Unfolding: Computing a Node's Depth



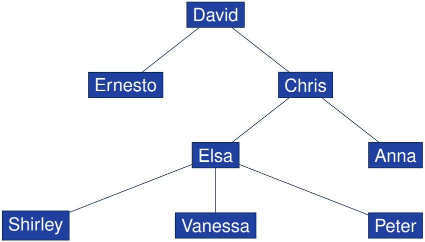
$$
\begin{array}{ll}
 & \textbf{\textit{depth}}(\texttt{vanessa}) \\
= & \{\ \textit{vanessa.getParent() == elsa}\ \} \\
 & 1 + \textbf{\textit{depth}}(\texttt{elsa}) \\
= & \{\ \textit{elsa.getParent() == chris}\ \} \\
 & 1 + 1 + \textbf{\textit{depth}}(\texttt{chris}) \\
= & \{\ \textit{chris.getParent() == david}\ \} \\
 & 1 + 1 + 1 + \textbf{\textit{depth}}(\texttt{David}) \\
= & \{\ \textit{David is the root}\ \} \\
 & 1 + 1 + 1 + 0 \\
= & 3
\end{array}
$$

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>`:

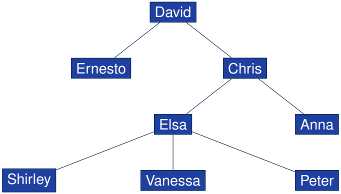
```
public int height(TreeNode<E> n) {
  TreeNode<E>[] children = n.getChildren();
  if(children.length == 0) { return 0; }
  else {
    int max = 0;
    for(int i = 0; i < children.length; i ++) {
      int h = 1 + height(children[i]);
      max = h > max ? h : max;
    }
    return max;
  }
}
```

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));
}
```

Unfolding: Computing a Tree's Height



$$
\begin{aligned}
& \textbf{height}(\texttt{subtree rooted at chris}) \\
= \ & \{ \textit{chris is not a leaf} \} \\
& MAX\begin{pmatrix} 1 + \textbf{height}(\texttt{subtree rooted at elsa}), \\ 1 + \textbf{height}(\texttt{subtree rooted at anna}) \end{pmatrix} \\
= \ & \{ \textit{elsa is not a leaf, anna is a leaf} \} \\
& MAX\begin{pmatrix} 1 + MAX\begin{pmatrix} 1 + \textbf{height}(\texttt{subtree rooted at shirley}), \\ 1 + \textbf{height}(\texttt{subtree rooted at vanessa}), \\ 1 + \textbf{height}(\texttt{subtree rooted at peter}) \end{pmatrix}, \\ 1 + 0 \end{pmatrix} \\
= \ & \{ \textit{shirley, vanessa, and peter are all leaves} \} \\
& MAX\begin{pmatrix} 1 + MAX\begin{pmatrix} 1 + 0, \\ 1 + 0, \\ 1 + 0 \end{pmatrix}, \\ 1 + 0 \end{pmatrix} \\
= \ & 2
\end{aligned}
$$

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`.

Binary Trees (BTs): Definitions

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

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

A *binary tree (BT)* is either:

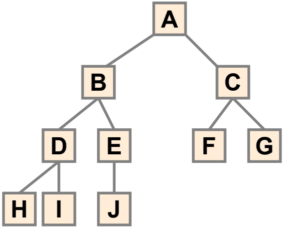
- An ***empty*** tree; or
- A ***nonempty*** tree with a ***root*** node *r* which has:
  - a *left subtree* rooted at its ***left child***, if any
  - a *right subtree* rooted at its ***right child***, if any

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.,

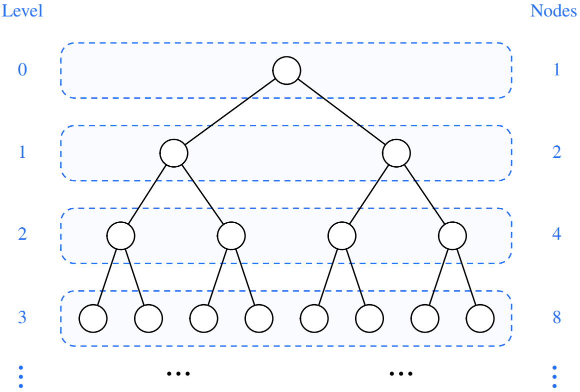


Node A has:

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

BT Terminology: Depths, Levels

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



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:

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

[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 = $1 + 2 + 4 + 8 + 16 = 1 \cdot (\frac{2^5 - 1}{2 - 1}) = 31$

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:

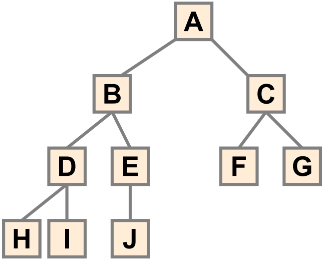
Maximum total number of nodes:

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

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? $(2^h - 1) + 1 = 2^h$

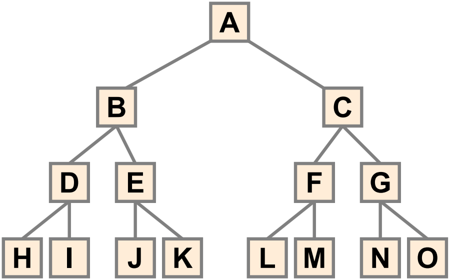
Q2: ***Maximum*** # of nodes of a ***complete*** BT? $2^{h+1} - 1$

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:

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

- Shape of BT with ***minimum*** height?

  A tree completely filled at each level

$$\begin{array}{lcl} & n & = & 2^{h+1} - 1 \\ \iff & n+1 & = & 2^{h+1} \\ \iff & log(n+1) & = & h+1 \\ \iff & log(n+1) - 1 & = & h \end{array}$$

- 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

BT Properties: Bounding # of Int. Nodes

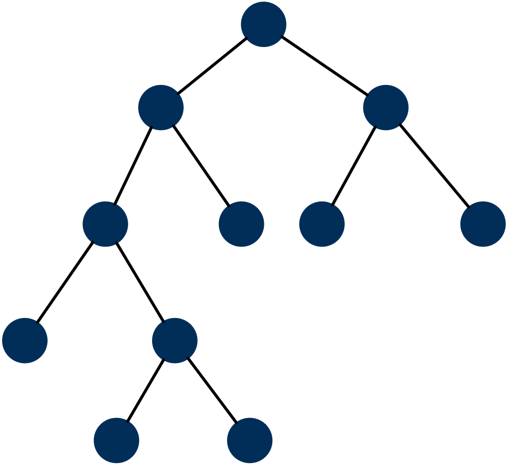
Given a *binary tree* with ***height*** h, the ***number of internal nodes*** n_I is bounded as:

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

- 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: $\underbrace{2^0 + 2^1 + \cdots + 2^{h-1}}_{n \text{ terms}} = 2^h - 1$

BT Terminology: Proper BT

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



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: $\mathbf{n_E} = \mathbf{n_I} + 1$

Proof by *mathematical induction*:

- **Base Case:**

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

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

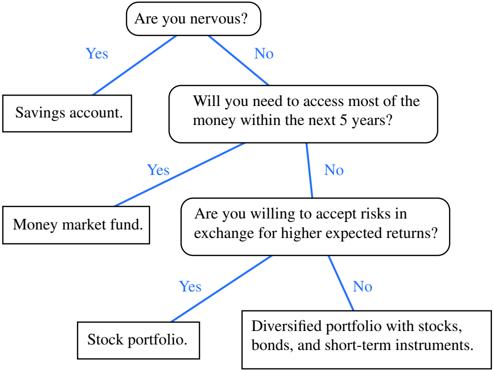
    Convert an external node into an ***internal*** node.

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

Binary Trees: Application (1)

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

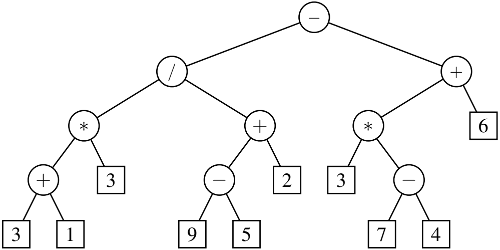
- Each ***internal node*** denotes a decision point: yes or no.
- Each ***external node*** denotes the consequence of a decision.



Binary Trees: Application (2)

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

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



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

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.

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
```

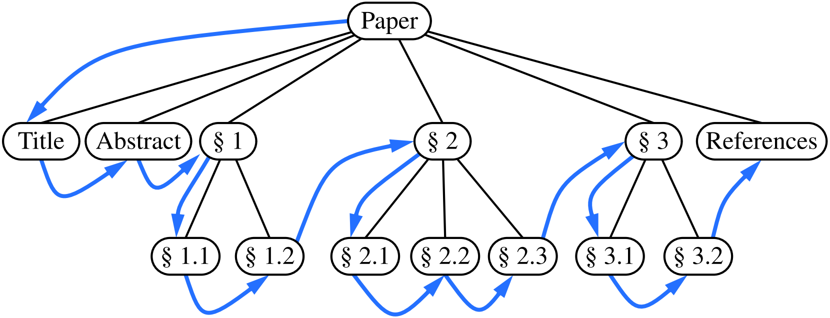
- **Inorder** (for *BT*): Visit left subtree, then parent, then right subtree.

```
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) }
```

Tree Traversal Algorithms: Preorder

Preorder: Visit parent, then visit child subtrees.

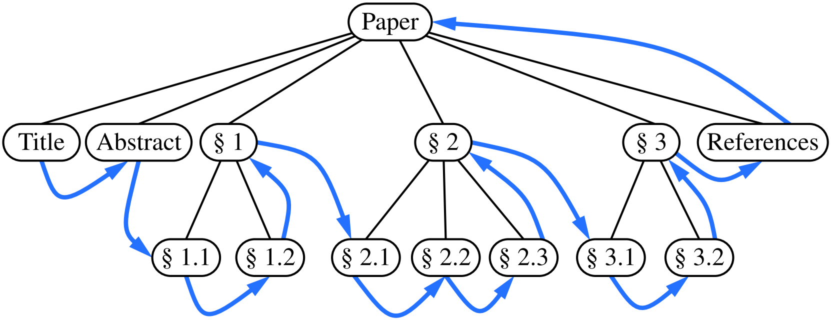
```
preorder (n)
  visit and act on position n
  for child c: children(n) { preorder (c) }
```



Tree Traversal Algorithms: Postorder

Postorder: Visit child subtrees, then visit parent.

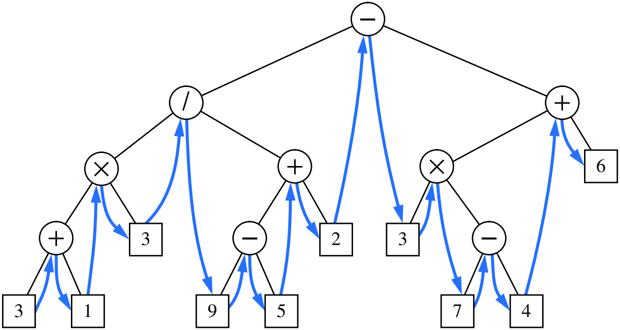
```
postorder (n)
  for child c: children(n) { postorder (c) }
  visit and act on position n
```



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) }
```



Index (1)

Index (2)

Index (3)

Index (4)