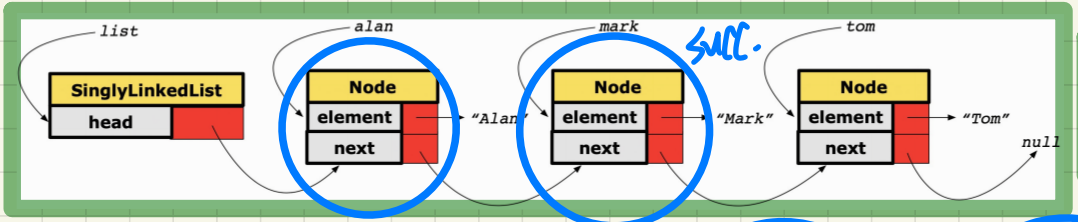
Lecture 5a

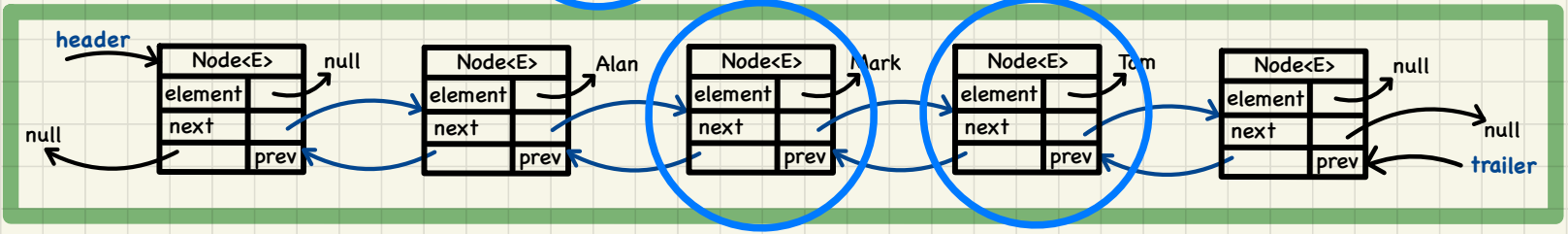
Part A

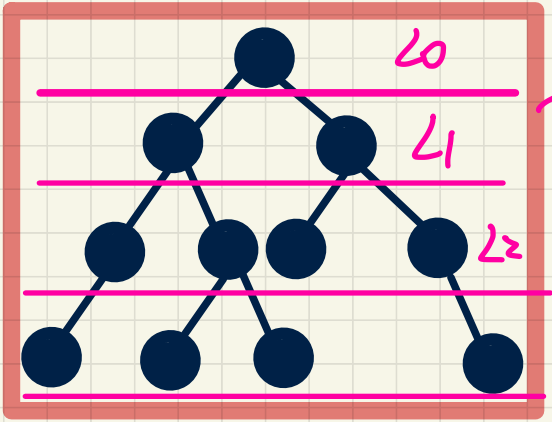
General Trees
Terminology, Applications

Linear vs. Non-Linear Structures

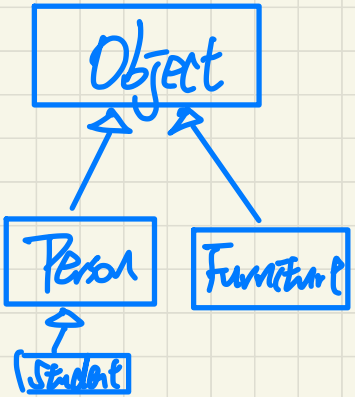
linear DS (can't branch out when traversing)



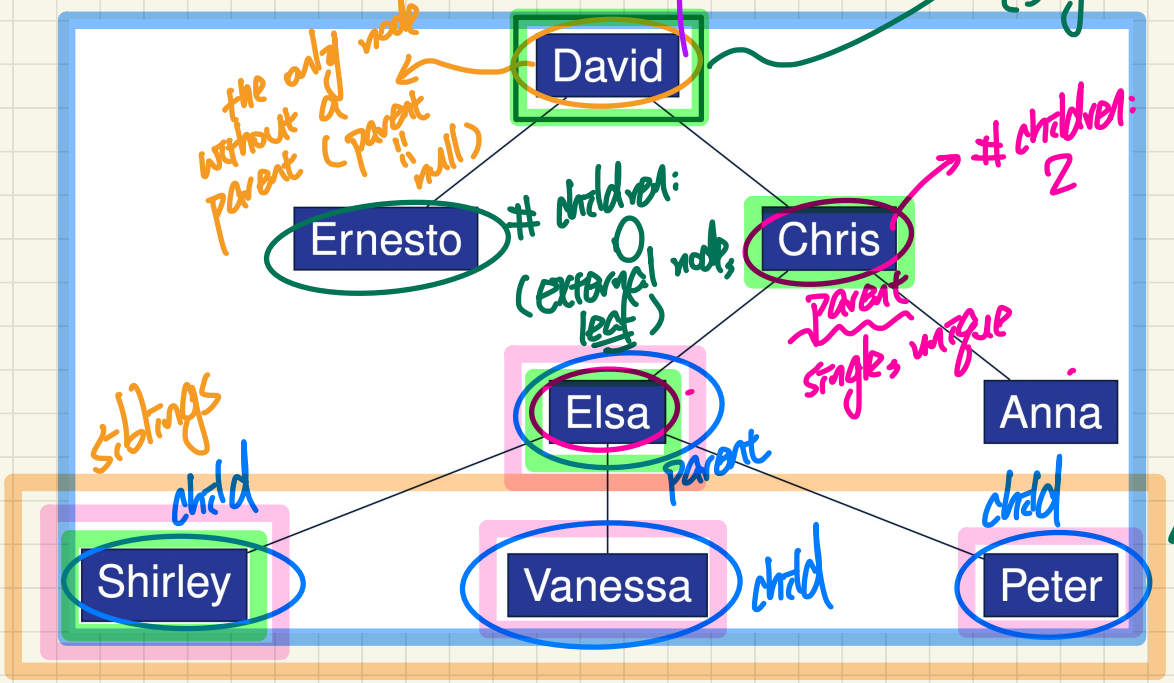




→ non-linear DS (levels = branching out structure) & hierarchical (e.g. inheritance)



General Trees: Terminology (1)



descendants of root covers the entire tree.

root (single, unique)

the only node without a parent (parent = null)

children: 0 (external node, leaf)

children: 2

parent single, unique

siblings

child

parent

child

child

- root
- parent
- children
- ancestors
- descendants
- siblings

ancestors of shirley:
shirley, elsa, chris
david

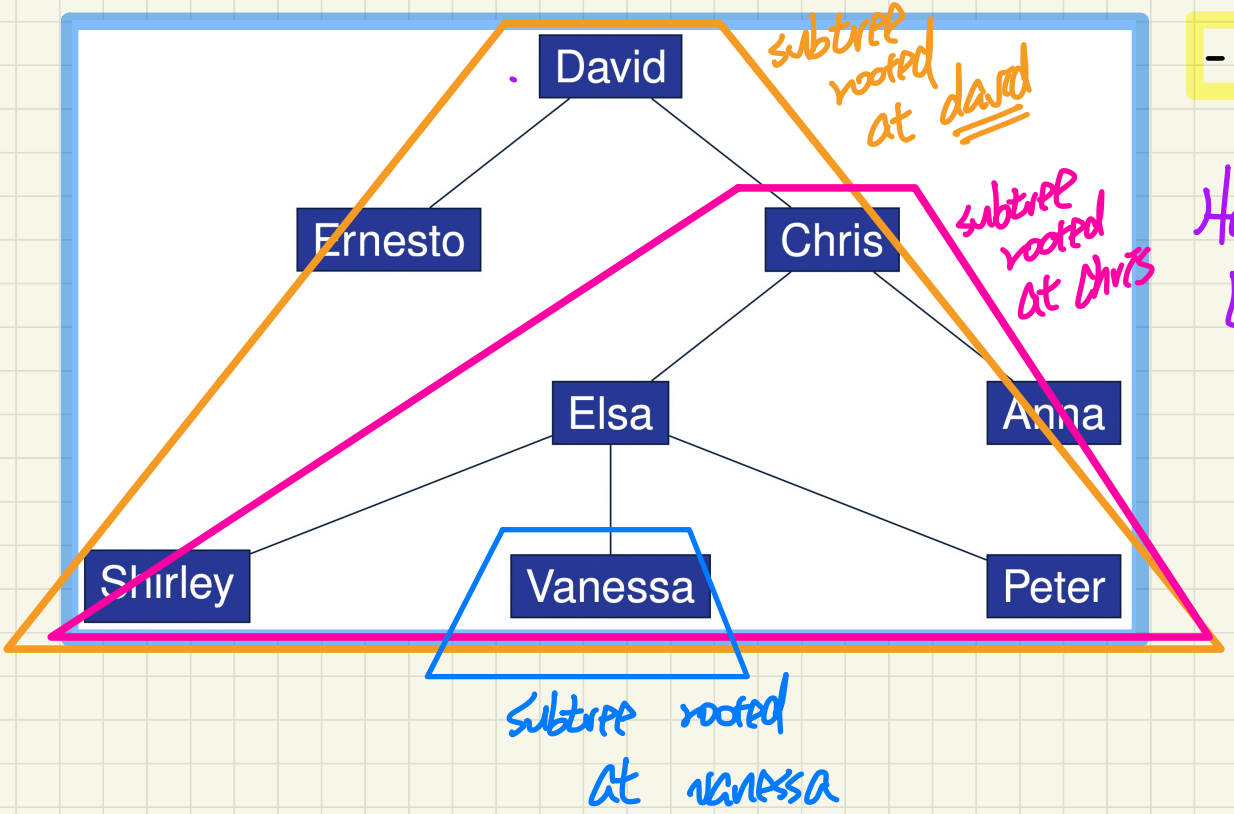
descendants of elsa:
elsa, shirley, vanessa, peter

Elsa is parent of vanessa
Chris is parent of Elsa
⇒ Chris is parent of vanessa ✗

General Trees: Terminology (2)

subtrees ≈ sub-arrays.

- subtree

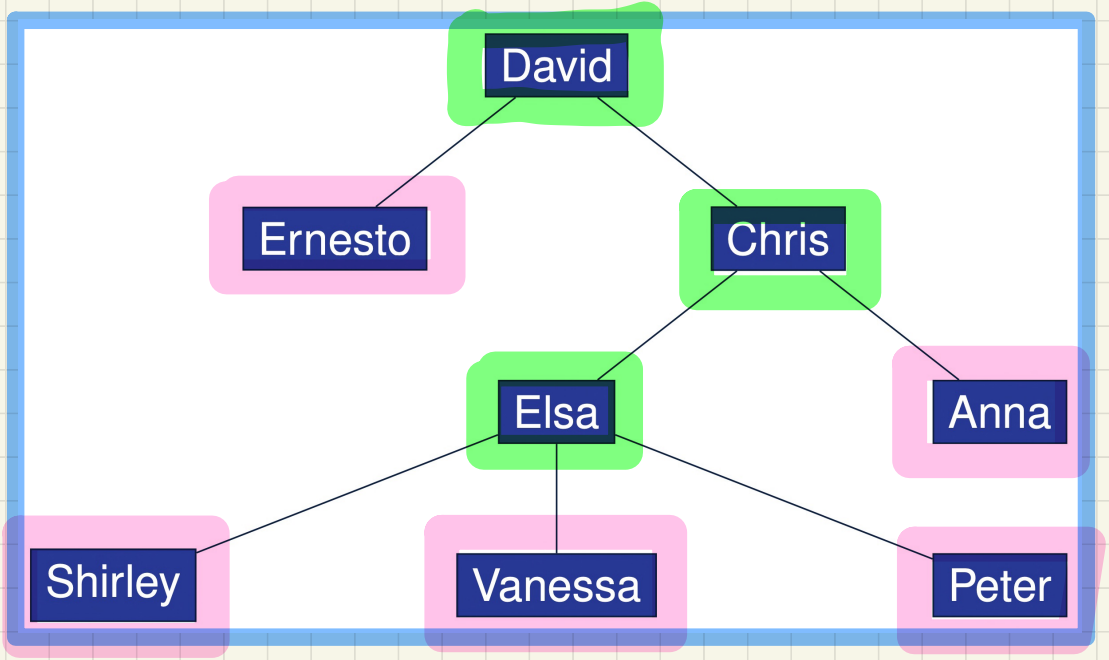


How many subtrees?

↳ count # node

8

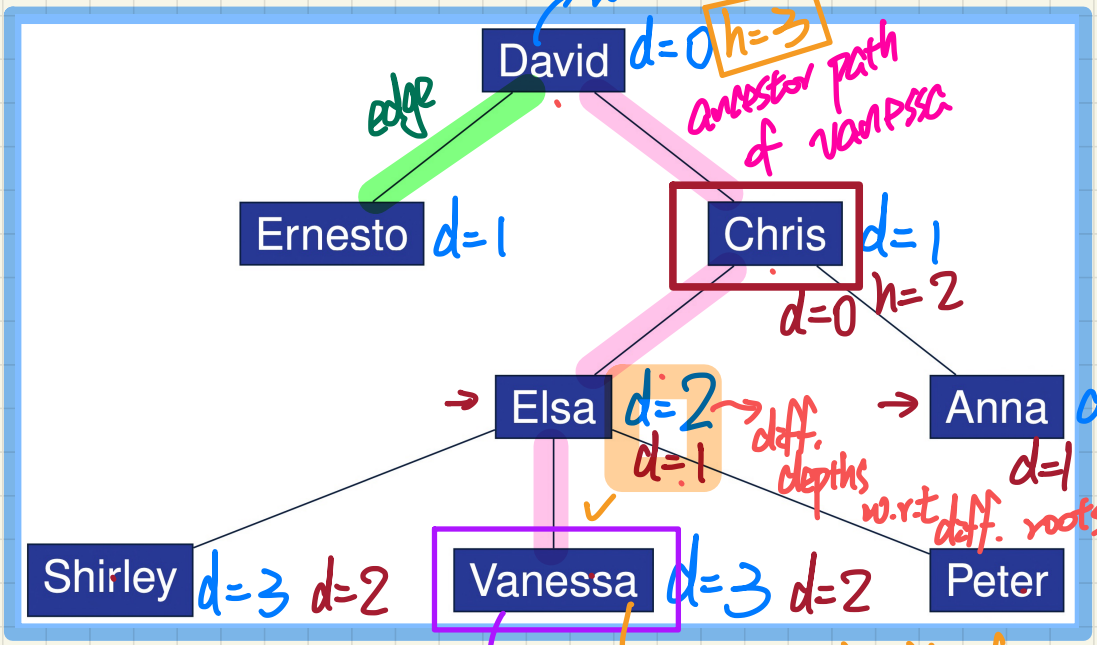
General Trees: Terminology (3)



- external nodes (5)
- internal nodes (3)

not necessarily at the bottom level.

General Trees: Terminology (4)



height of subtree rooted at David.

ancestor path of vanessa

- edge
- path
- depth → # edges from root
- height → max depth among all nodes

diff. depths w.r.t diff. roots

d=0 & h=0 → relative to the subtree root (vanessa)

height of subtree rooted at external node vanessa

* heights of subtrees rooted at external nodes are 0.

General Trees: Recursive Definition

- root
- size

Case 1: Empty Tree

$\emptyset$ root → null

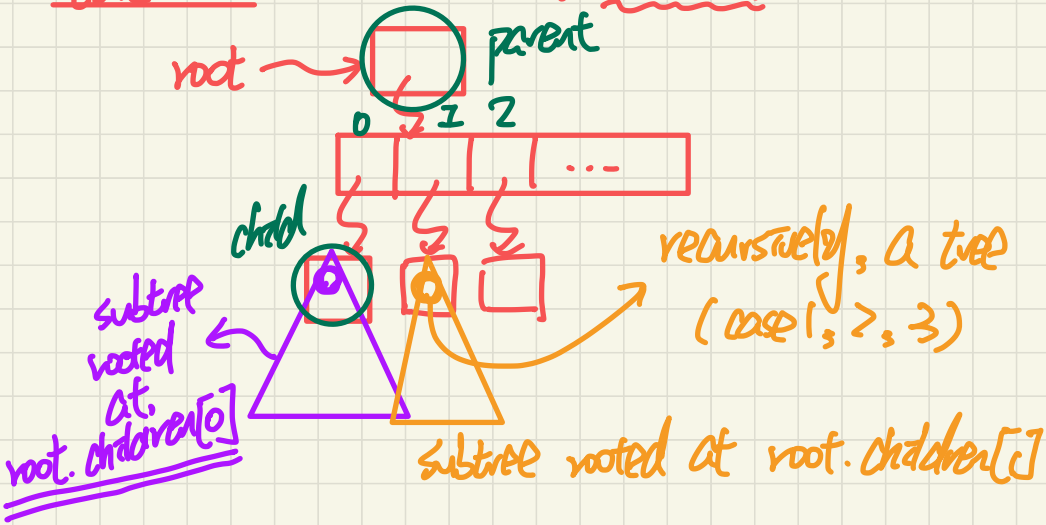
size = 0

Case 2

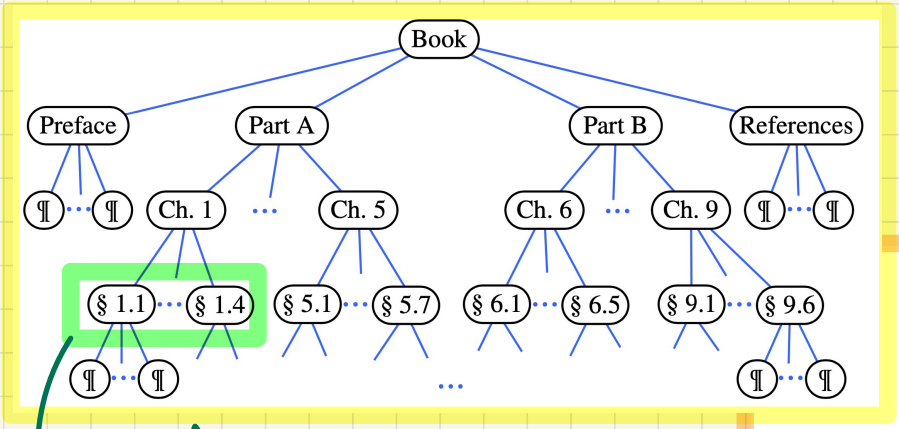
root → "alan"

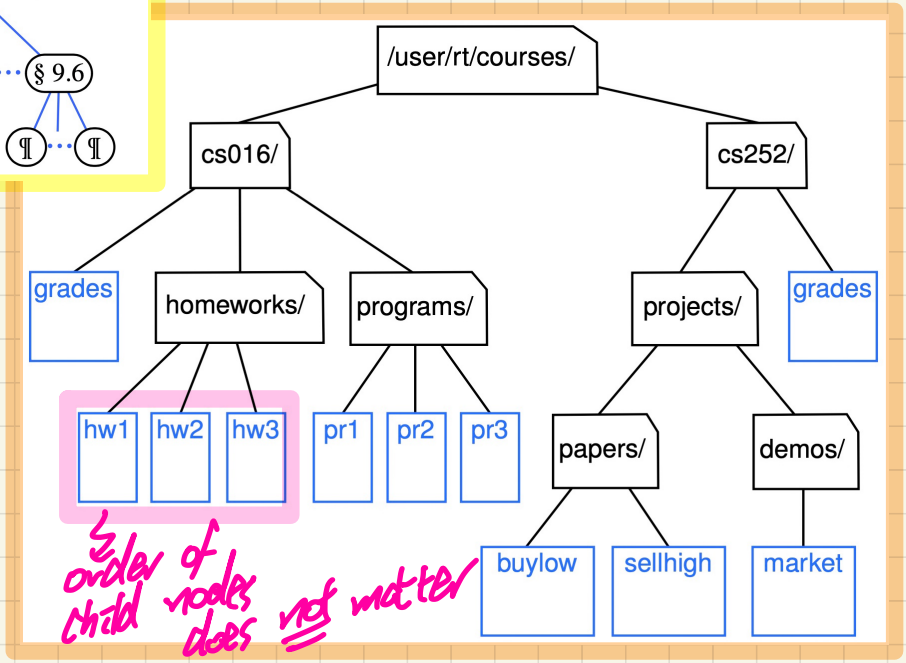
size = 1

Case 3 (Recursive Case) size > 1



General Trees: Ordered vs. Unordered Trees





Lecture 5a

Part B

General Trees
Implementing a Generic Tree in Java

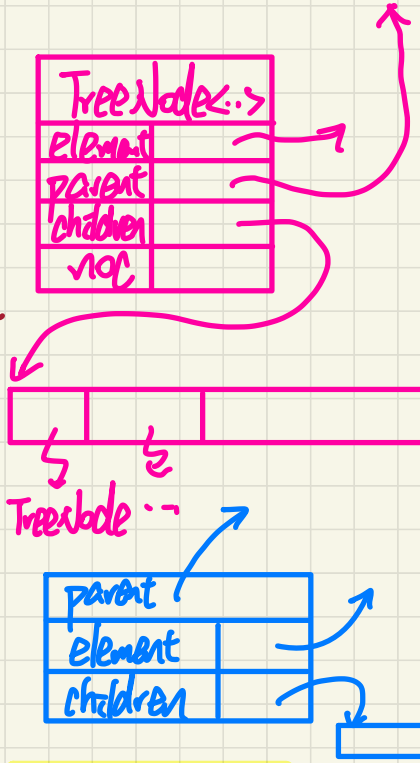
Generic, General Tree Nodes

```
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 TreeNode(E element) {
    this.element = element;
    this.parent = null;
    this.children = (TreeNode<E>[])
      Array.newInstance(this.getClass(), MAX_NUM_CHILDREN);
    this.noc = 0;
  }
  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) { ... }
}
```

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

Class Cast Exception

TreeNode<E>



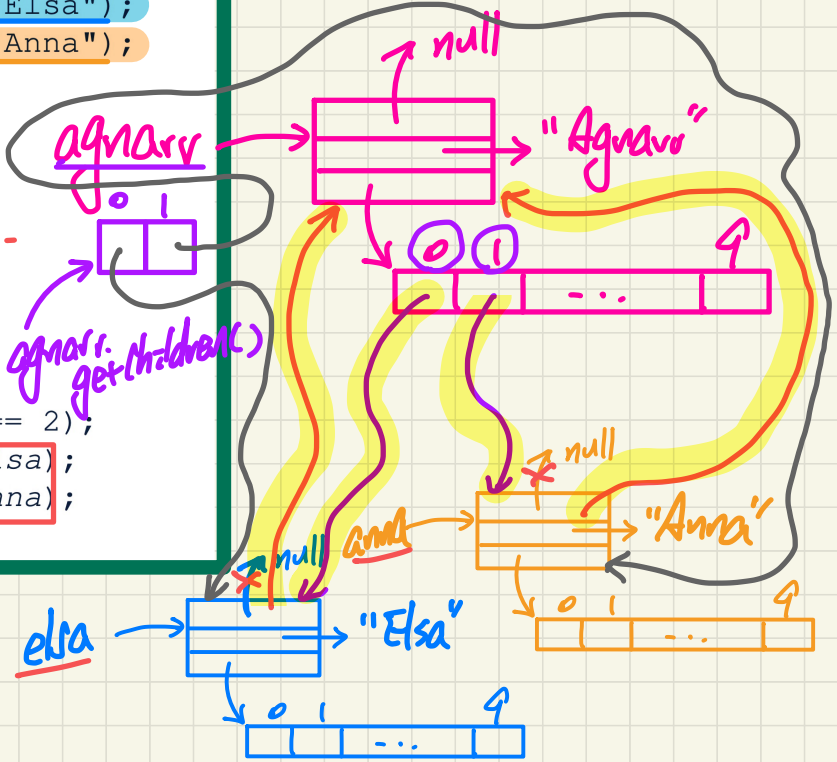
Compare:

+ prev ref.
+ next ref.

in a DLN.

Tracing: Constructing a Tree

```java
@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);
}
```



Tracing: Computing a Node's Depth

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

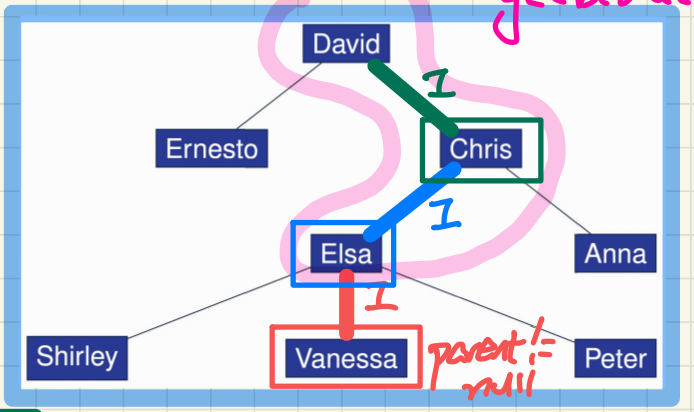
n is the root

edge from n to its parent

strictly smaller subproblem

```
@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));
}
```

trace: depth(anna)



depth(vanessa)

= 1 + depth(elsa)

= 1 + 1 + depth(chris)

= 1 + 1 + 1 + depth(david)

depth(david) = 0

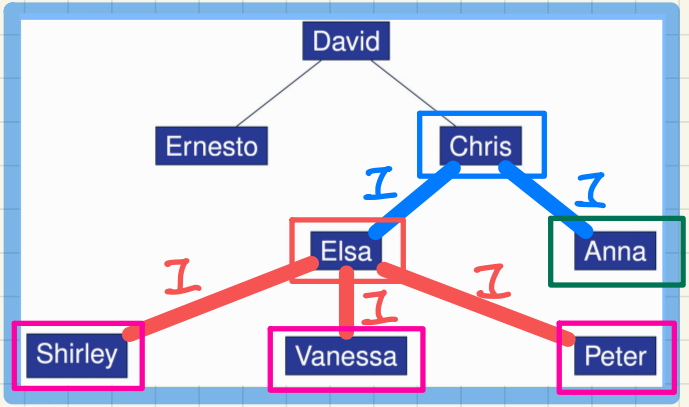
= 3

Tracing: Computing a Tree's Height

```
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;
  }
}
```

n is external (leaf)

↳ strictly small problem: height of a child node.




```
@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));
}
```

Exercise

height(chris)

$= 1 + \text{MAX}(\text{height(elsa)}, \text{height(anna)})$

$= 1 + \text{MAX}(1 + \text{MAX}(h(s), h(v), h(p)), 0)$

$= 2$

Lecture 5a

Part C

Binary Trees
Definition, Terminology, Properties

Binary Trees: Recursive Definition

- root
- size

Case 1: Empty BT

size = 0

$\emptyset$ root → null

Case 2: one-node BT

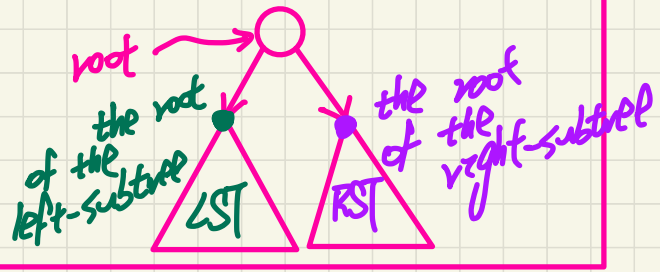
size = 1

root → ?

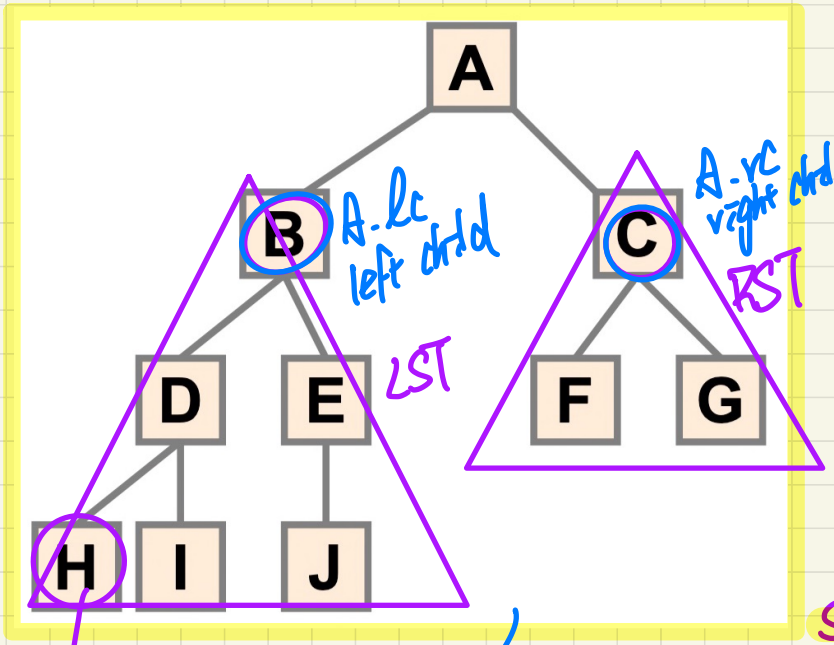
algorithm on a BT should be recursive

given that a BT's structure is recursive.

Case 3: size of BT > 1



BT Terminology: LST vs. RST



Strategy of Recursion on BT:

+ Do something on root
+ Recur on LST
+ Recur on RST

e.g.,

+ counting size
+ searching item

size(A) = 1 + size(A.lc) + size(A.rc)

size(H) = 1

ext. node (no further recursion necessary).

BT without satisfying the "search property"

search(A, item) = A.equals(item) || search(A.lc, item) || search(A.rc, item)

search(H, item) = H.equals(item)

BT Terminology: Depths, Levels, Max # of Nodes

Max # of nodes in a tree with height h:

$2^0 + 2^1 + 2^2 + \cdots 2^h$

$= 1 \cdot \left(\frac{2^{h+1}-1}{2-1}\right)$

$= 2^{h+1} - 1$

Level? $\stackrel{d}{=}$ depth

Max # Nodes at Level?

$1 = 2^0$

r

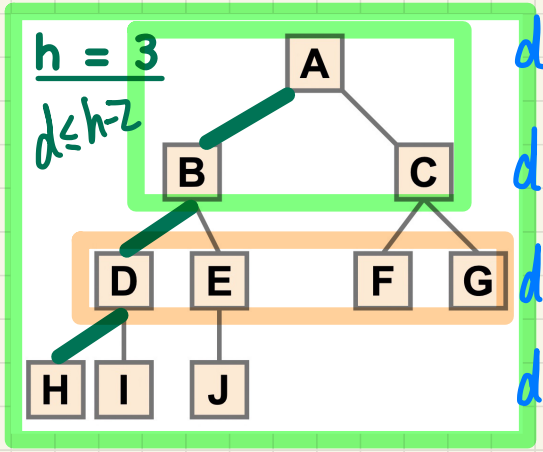
$1 * 2 = 2 = 2^1$

$2 * 2 = 4 = 2^2$

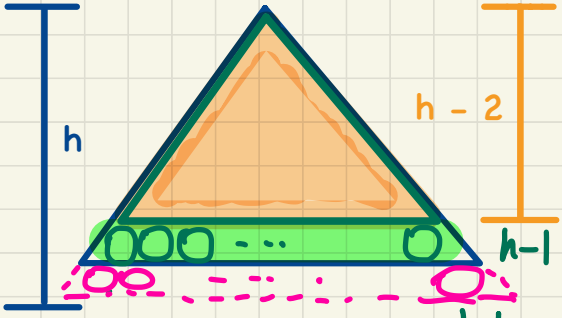
$4 * 2 = 8 = 2^3$

2^h

BT Terminology: Complete vs. Full BTs

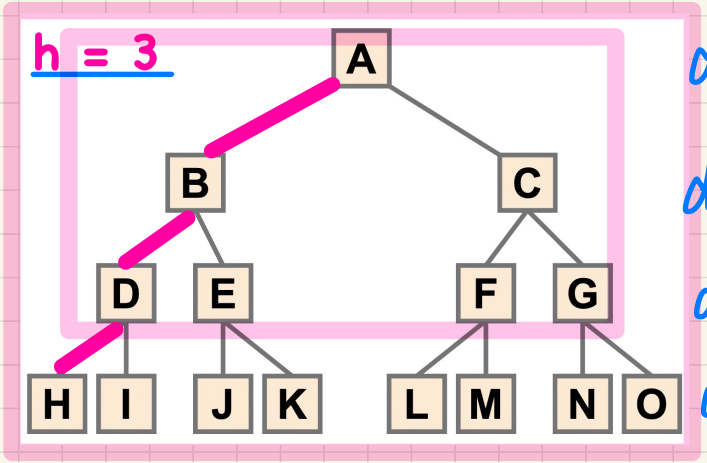


d=0
d=1 h-2
d=2 h-1
d=3 h

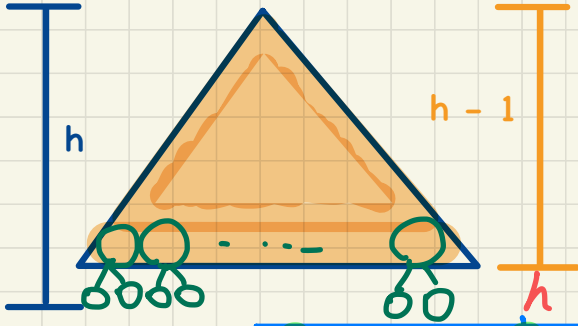


Min # nodes? $(1+2+4\cdots+2^{h-1})+1=2^h$

Max # nodes? $(1+2+4\cdots+2^{h-1})+2^h=(2^h-1)+2^h=2^{h+1}-1$



d=0
d=1
d=2 h-1
d=3 h



Min # nodes?

Max # nodes?

$2^0+2^1+\cdots+2^h=2^{h+1}-1$

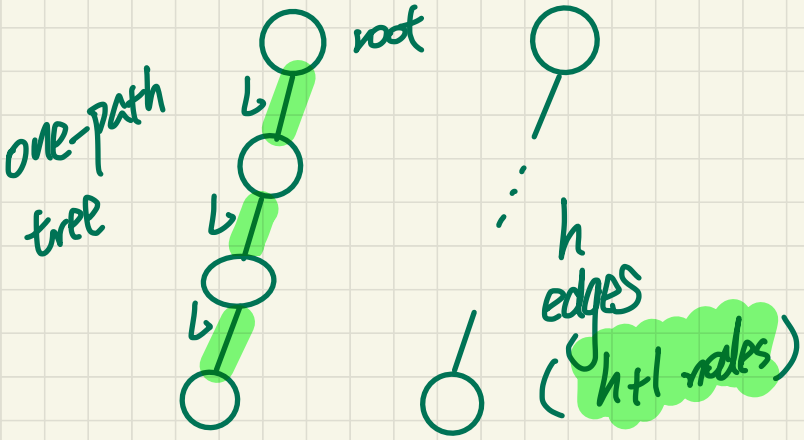
BT Properties: Bounding # of Nodes

Given a *binary tree* with ***height*** $h \geq 0$, the ***number of nodes*** n is bounded as:

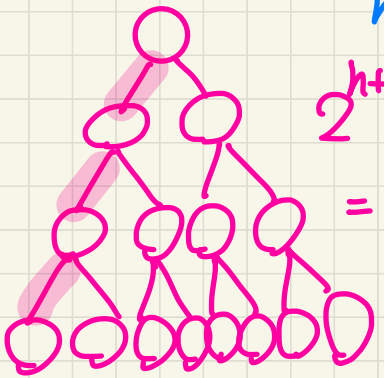
$$h + 1 \leq n \leq 2^{h+1} - 1$$

For example, say h = 3

Minimum # of nodes



Maximum # of nodes

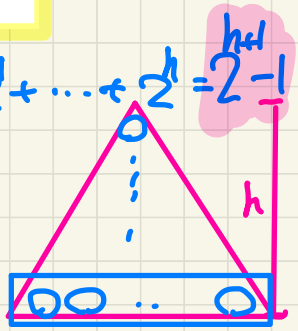


$2^{h+1} - 1$

$= 2^{3+1} - 1$

$= 15$

$2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$



BT Properties: Bounding Height of Tree

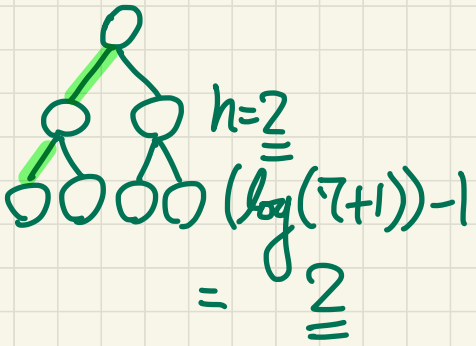
Given a *binary tree* with $n \geq 0$ ***nodes***, the ***height*** h is bounded as:

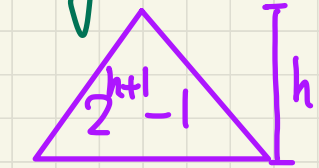
$$\log(n+1) - 1 \leq h \leq n - 1$$

For example, say n = 7.

Minimum height

(branch out two ways whenever possible)





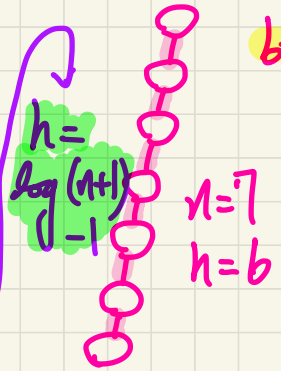
$n = 2^{h+1} - 1$

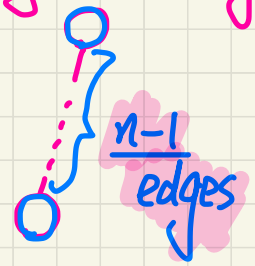
$n+1 = 2^{h+1}$

$\log(n+1) = h+1$

Maximum height

(don't waste any nodes branching two ways)





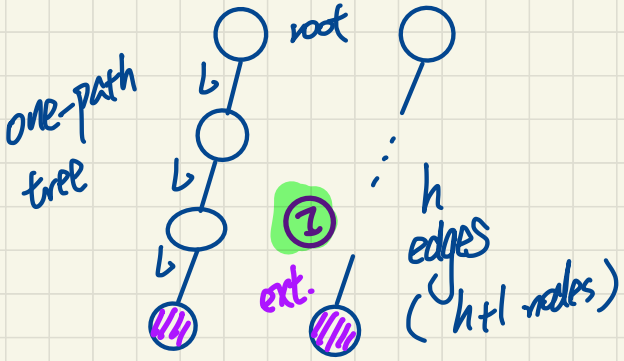
BT Properties: Bounding # of External Nodes

Given a *binary tree* with *height* h (≥ 0) the *number of external nodes* n_E is bounded as:

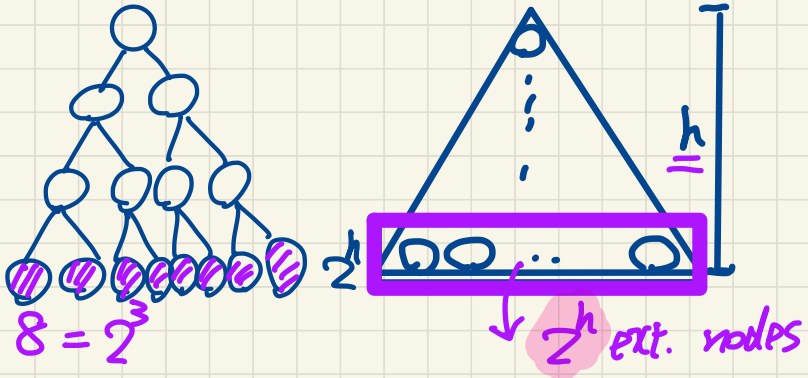
$$1 \leq n_E \leq 2^h$$

For example, say h = 3

Minimum # of External Nodes



Maximum # of External Nodes



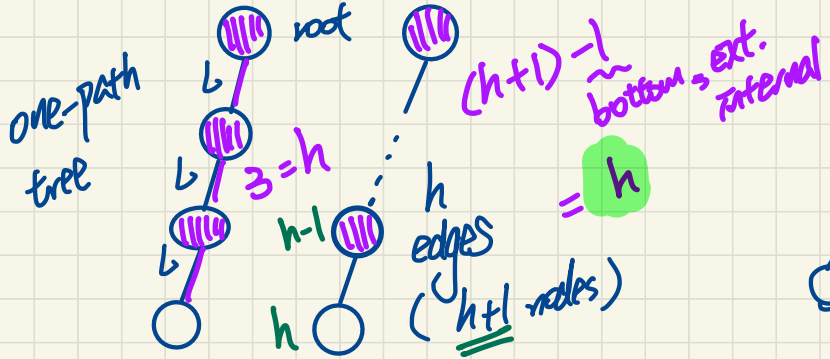
BT Properties: Bounding # of Internal Nodes

Given a *binary tree* with *height* $h \geq 0$, the *number of internal nodes* n_I is bounded as:

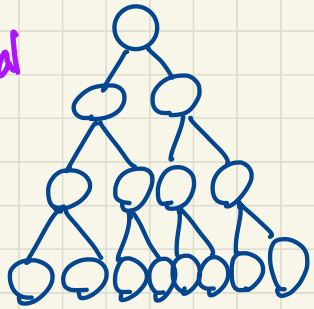
$$h \leq n_I \leq 2^h - 1$$

For example, say h = 3

Minimum # of Internal Nodes



Maximum # of Internal Nodes



$2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = 2^h - 1$