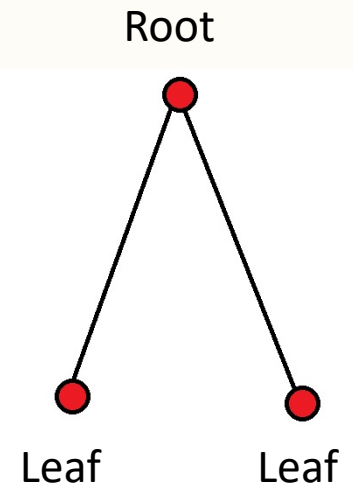
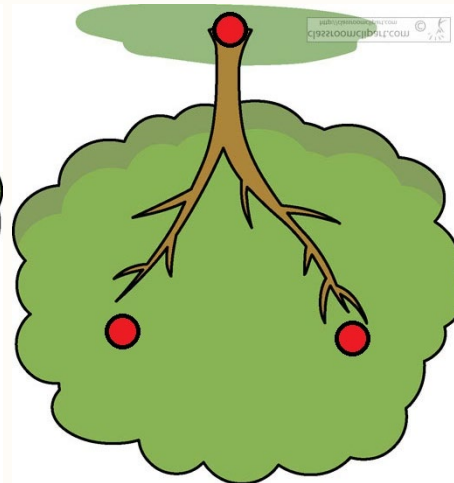
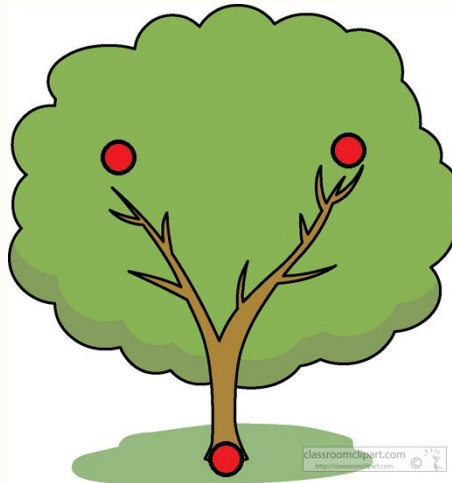
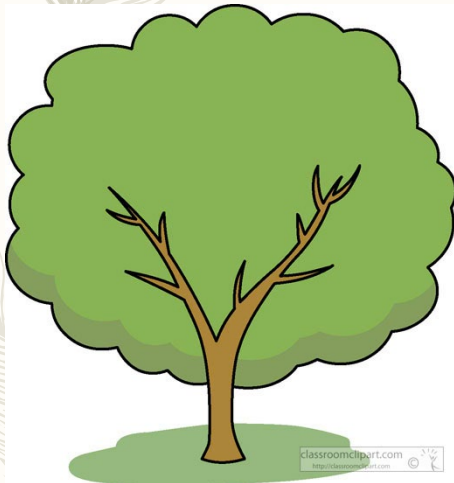


# Trees

- Data structure which looks like an upside down tree (or the root system of a tree)
- Nodes have **parents** and **children**
- **No loops**



# Trees

---

- Collection of **nodes** and **edges**
  - **Any shape**, but can't have a loop
  - **Acyclic** means “no cycles” (i.e. no loops)
- Nodes have:
  - **data**
  - (possibly) a “**key**” to sort/search by
  - (possibly) **pointer** to **children**
  - (possibly) **pointer** to **parent**



---

# Tree Definitions

Note: Unfortunately, due to the manner in which the field of computer science grew out of many other fields (math, engineering, etc.)... there are no completely-agreed-upon definitions of some of these terms.

We will use the definitions given in these slides for CS310.

# Tree Definitions 1/3

---

- The **root** node is the top most node in the tree
- The **descendants** of a node are all the nodes below it (and sometimes includes the node itself)
- The **ancestors** of a node are the nodes on the path from the node to the **root** (and sometimes includes the node itself)
- Nodes are **siblings** if they have the same parent



# Tree Definitions 2/3

---

- The **leaf** nodes have no children, the **root** node has no parent
- An **inner** node has at least one child (i.e. not a leaf)
- A **null link** describes an empty link, typically child link
- The **depth** of a node is the length (number of edges) of the path from the node to the root
- The **node height** is the length of the path from the node to the deepest leaf.
- The **tree height** is the maximum **depth** of any node in the tree



# Tree Definitions 3/3

---

## – Full tree

- every node other than the leaves has the max number of children

## – Perfect tree (sometimes called "complete")

- all leaves have the same depth
- every node other than the leaves has the max number of children

## – Nearly complete tree (sometimes called "complete")

- last level is not completely filled

## – Balanced tree

- height of the left and right sub trees of every node differ by 1 or less (as used by AVL tree)

## – Degenerate tree

- each parent node has only one associated child node
- equiv. to linked list, maximum height?



# Common Tree Operations

---

- Searching for an item
- Adding items
- Deleting items (synonymous with removing)
- Balancing
- Iterating = mention things one by one
  - all the items (in some order)
  - a section of a tree



# k-ary Trees

---

- aka. **n-ary** and **m-ary** trees
- each **parent** can have **only k children**
  - k is the “**branching factor**”
- **number of nodes** in a **perfect** k-ary tree
  - $(k^{h+1}-1)/(k-1)$ 
    - *h is the **height** of the tree*
- **number of leaves** in a **perfect** k-ary tree
  - first level  $k^0$ , second  $k^1$ , third  $k^2$ ...  $k^h$
- **height** of a **perfect** k-ary tree of with n nodes
  - $\log_k((k-1)*(n)+1)-1$



# Binary Trees (k-ary where k=2)

---

- each **parent** can have only **two children**
- **number of nodes** in a **nearly complete** binary tree
  - min:  $2^h + 1$       max:  $2^{h+1} - 1$ 
    - *max from k-ary tree formula:*  $(k^{h+1} - 1) / (k - 1)$ ,  $k=2 \therefore 2^{h+1} - 1 / (2 - 1) = 2^{h+1} - 1$
- **number of leaves** in **perfect** binary tree
  - first level  $2^0$ , second  $2^1$ , third  $2^2 \dots 2^h$
- **number of internal nodes** in **perfect** binary tree of n nodes
  - $\lfloor n/2 \rfloor$
- **height** of a **balanced** binary tree of n nodes
  - $\lceil \lg(n+1) \rceil$

# Binary Tree Storage: Arrays

---

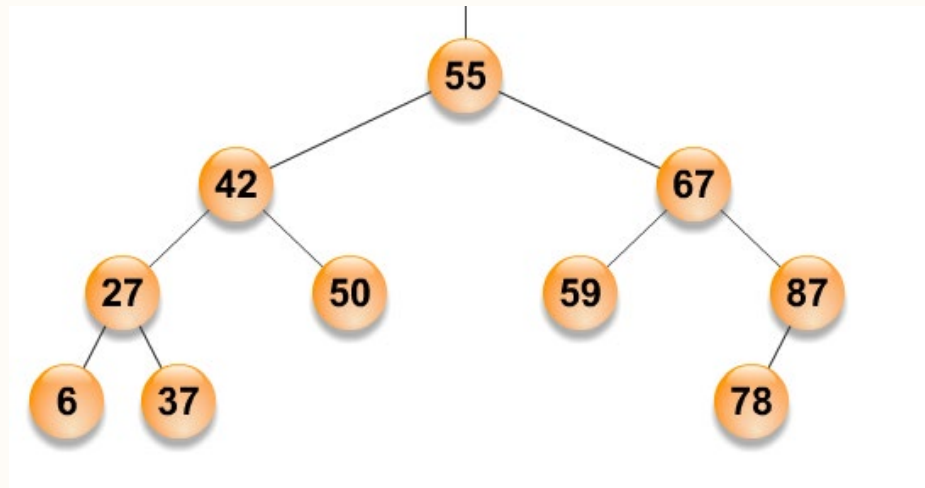
- **Root** at index 0
- **Children** at index:
  - $\text{parentIndex} * 2 + 1$
  - $\text{parentIndex} * 2 + 2$
  - e.g. root at 0, children of root at index 1 and 2
- **Parent** at index
  - $\lfloor (\text{childIndex} - 1) / 2 \rfloor$
  - e.g. parent of item at index 2 =  $\lfloor 1/2 \rfloor = 0$
- **Common variant** is to start root at index 1

# Array Storage Example

---



– Draw the binary tree for:

[55, 42, 67, 27, 50, 59, 87, 6, 37, null, null, null, null, 78]



# Binary Tree Storage: Links

---

- 
- 
- (optional) key
  - value
  - link to child 1
  - link to child 2
  
  - Do we need a link from child to the parent?

```
class Node<K,V> {  
    K key;  
    V value;  
    Node<K,V> left;  
    Node<K,V> right;  
}
```

```
class Node {  
    int value;  
    Node[] children;  
}
```

```
class Node<V> {  
    V value;  
    Node<V> left;  
    Node<V> right;  
}
```

# Other Tree Storage

---

## K-ary Tree Storage

- node structure
  - array/list of children
- array structure
  - root, children of root, grandchildren of root, etc.
  - same as binary tree, but different math

## Arbitrary Tree Storage

- node structure
  - list of children (dynamic or linked)
- array structure
  - first-child-next-sibling storage (see textbook 18.1.2)



# Tree Storage: Arrays vs. Linked

---

## – Arrays

- Need to know where each item is
- How? Need to limit number of children and/or use other methods of storage (see first-child-next-sibling storage)
- Most common for balanced trees (very little wasted space)
- Not good for degenerate trees (lots of wasted space)
- Fast memory access (compared to linked list)

## – Linked Data Structures

- easy to add, delete, and swap around parts of the tree

