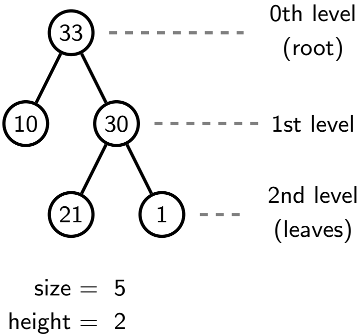
Trees

Trees

A tree is a very flexible and powerful data structure that, like a linked list, involves connected nodes, but has a hierarchical structure instead of the linear structure of linked lists.

Depending on the number of child nodes that each node has:

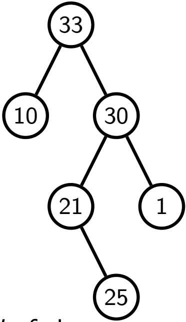
- Unary trees (0–1 children) = Linked Lists,
- Binary trees (0–2),
- Ternary trees (0–3),
- Quad trees (0–4), ...



Size = number of nodes

Height = length of longest path from the root to a leaf

Tree Terminology



- *Root*: the unique node at the base of the tree.
- Each node is connected by a link, called an *edge*, to each of its *child* nodes.
- Each *child* node has exactly one *parent* node.
- *Siblings* are nodes with the same *parent*.
- A node with no child nodes is called a *leaf*
- An *ancestor/descendent* of a node is the *parent/child* of the node or (inductively) the *ancestor/descendent* of that *parent/child*.
- A path is a sequence of connected edges between two nodes.

Tree Terminology

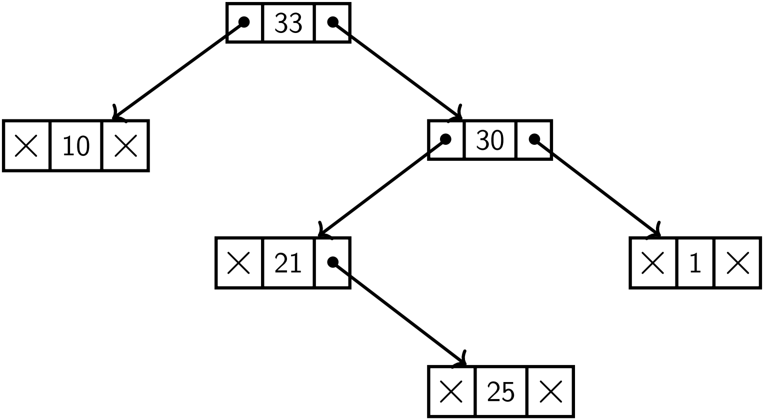
- Trees have the property that there is exactly 1 path between each node and the root
- The *depth* or *level* of a node is the length of the path from the node to the *root* (*root* has *level* 0).
- The *height* of a tree is the length of the longest path from the *root* to a *leaf*.
- The *size* of a tree is the number of nodes in the tree.
- A tree with one node has *size* 1 and *height* 0.
- An empty tree (with no nodes) has *size* 0 and, by convention, a *height* of -1.

Tree Implementation Options

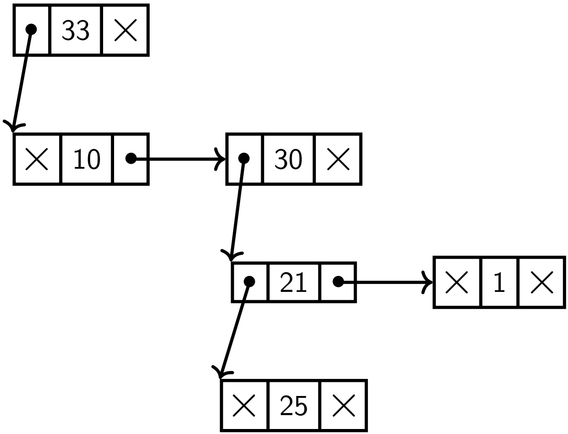
There are 3 common approaches to implementing trees:

1. Basic: Use nodes like doubly linked list nodes with a value field, and left and right child pointers
2. Sibling List: Use nodes with a value field, a single *children* pointer, and a pointer to the next sibling. This is good for trees with a variable number of children in each node.
3. Array: For binary trees, use arrays with a layout based on storing the root at index 1, then the children of the node at index i is stored at index $2 * i$ and $2 * i + 1$.

Tree Implementation Options: Basic



Tree Implementation Options: Sibling List



Tree Implementation Options: Array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	33	10	30			21	1					25			

Binary Tree ADT

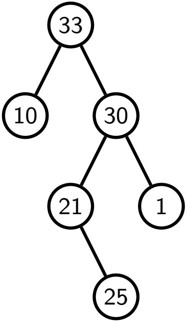
Just as with lists, we can define the *Binary Tree Abstract Data Type* inductively:

- Constructors:
  - `EmptyTree`: returns an empty tree
  - `MakeTree(v, l, r)`: returns a new tree where the root node has value `v`, left subtree `l` and right subtree `r`
- Accessors:
  - `isEmpty(t)`: return true if `t` is the empty tree, otherwise returns false
  - `root(t)`: returns the value of the root node of the tree `t` [1]
  - `left(t)`: returns the left subtree of the tree `t` [2]
  - `right(t)`: returns the right subtree of the tree `t` [2]
- Convenience Constructor:
  - `Leaf(v)` = `MakeTree(v, EmptyTree, EmptyTree)`

[1] Triggers error if the tree is empty

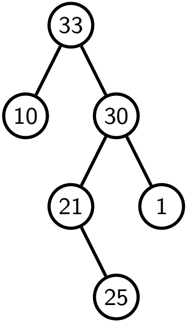
Example: Construct a Tree

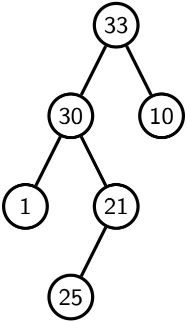
```
MakeTree(33,
         Leaf(10),
         MakeTree(30,
                  MakeTree(21,
                           EmptyTree,
                           Leaf(25)),
                  Leaf(1)))
```



Example: Reverse a tree

```
reverseTree(t) {
  if ( isEmpty(t) )
     return (t)
  else
    return (MakeTree(root(t),
                reverseTree(right(t)),
                reverseTree(left(t))))
```





Example: Flatten a tree into a list

Here we assume that we have the code to append two lists (see the handout *dsa-slides-02-01-intro-ADT.pdf*) and that `isEmpty(...)` can distinguish between the list and the tree version (e.g. by qualifying it with the ADT name):

```
flatten(t) {
  if Tree.isEmpty(t)
    return EmptyList
  else
   return append(flatten(left(t)),
                 MakeList(root(t), flatten(right(t)))))
}
```