

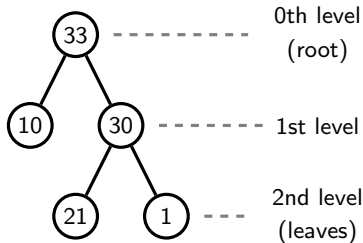
Trees

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), ...



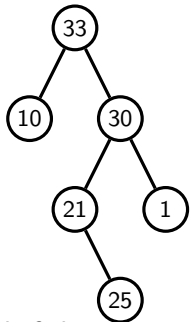
size = 5
height = 2

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

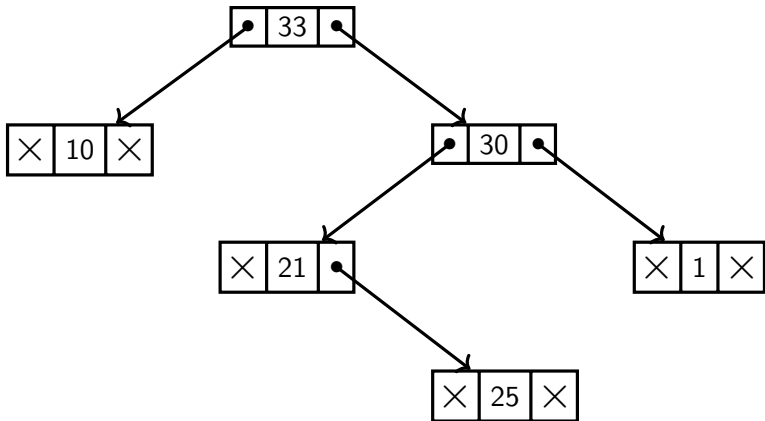
- Trees have the property that there is exactly 1 path between each node and the root
- The *depth* or *level* of a node is the length of the path from the node to the *root* (*root* has *level* 0).
- The *height* of a tree is the length of the longest path from the *root* to a *leaf*.
- The *size* of 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

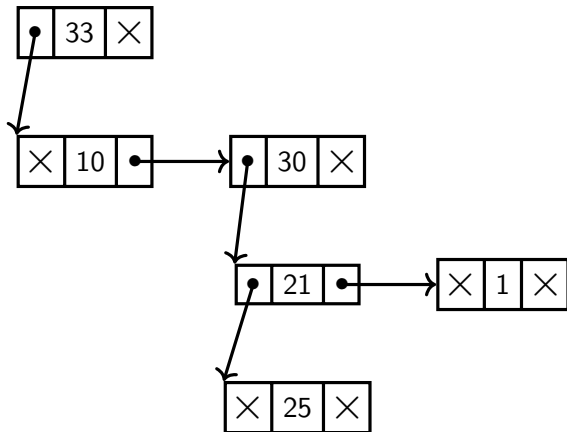
There are 3 common approaches to implementing trees:

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

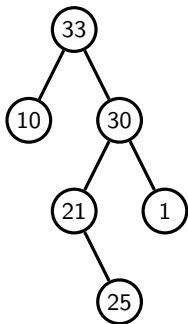
Just as with lists, we can define the *Binary Tree Abstract Data Type* inductively:

- 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`¹
 - `right(t)` : returns the right subtree of the tree `t`¹
- Convenience Constructor:
 - `Leaf(v) = MakeTree(v, EmptyTree, EmptyTree)`

¹Triggers error if the tree is empty

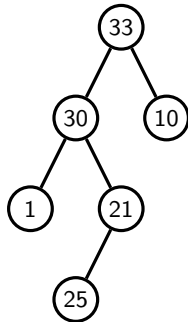
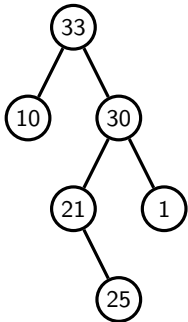
Example: Construct a Tree

```
1  MakeTree(33,  
2      Leaf(10),  
3      MakeTree(30,  
4          MakeTree(21,  
5              EmptyTree,  
6                  Leaf(25)),  
7          Leaf(1)))
```



Example: Reverse a tree

```
1 reverseTree(t) {  
2   if ( isEmpty(t) )  
3     return (t)  
4   else  
5     return (MakeTree(root(t),  
6                   reverseTree(right(t)),  
7                   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):

```
1 flatten(t) {  
2     if Tree.isEmpty(t)  
3         return EmptyList  
4     else  
5         return append(flatten(left(t)),  
6                        MakeList(root(t), flatten(right(t))))  
7 }
```