CITS2200 Data Structures and Algorithms

Topic 11

# Trees

- Why trees?
- Binary trees
  - definitions: size, height, levels, skinny, complete
- Trees, forests and orchards
- Tree traversal
  - depth-first, level-order
  - traversal analysis

Reading: Lambert and Osborne, Chapter 11

# 1. Why Study Trees?

Wood. . .

**“Trees are ubiquitous.”**

Examples. . .

| | |
|---|---|
| genealogical trees | organisational trees |
| biological hierarchy trees | evolutionary trees |
| population trees | book classification trees |
| probability trees | decision trees |
| induction trees | design trees |
| graph spanning trees | search trees |
| planning trees | encoding trees |
| compression trees | program dependency trees |
| expression/syntax trees | gum trees |
| ⋮ | ⋮ |

Also, *many other data structures are based on trees!*

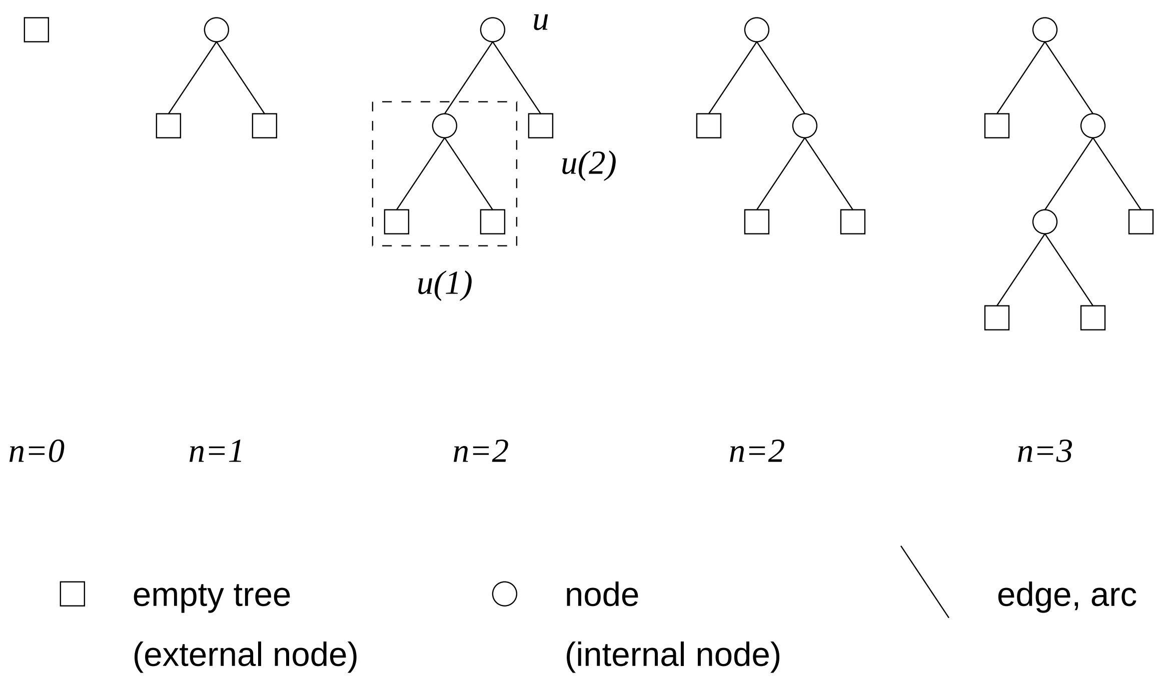
# 2. Binary Trees

## Definition

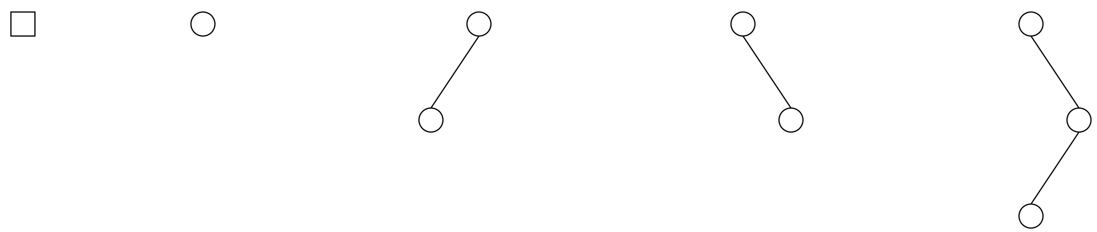
A *binary (indexed) tree* $T$ of $n$ nodes, $n \geq 0$, either:

- *is empty*, if $n = 0$, *or*
- *consists of a* root node $u$ *and two binary trees* $u(1)$ and $u(2)$ of $n_1$ and $n_2$ nodes respectively such that $n = 1 + n_1 + n_2$.
  - $u(1)$: *first* or *left subtree*
  - $u(2)$: *second* or *right subtree*

  The function $u$ is called the *index*.




We will often omit external nodes...



More terminology...

## Definition

Let $w_1$, $w_2$ be the roots of the subtrees $u_1$, $u_2$ of $u$. Then:

- $u$ is the *parent* of $w_1$ and $w_2$.
- $w_1$, $w_2$ are the (*left* and *right*) *children* of $u$. $u(i)$ is also called the $i^{th}$ child.
- $w_1$ and $w_2$ are *siblings*.

*Grandparent*, *grandchild*, etc are defined as you would expect.

A *leaf* is an (internal) node whose left and right subtrees are both empty (external nodes).

The external nodes of a tree define its *frontier*.

In the following assume $T$ is a tree with $n \geq 1$ nodes.

## Definition

Node $v$ is a *descendant* of node $u$ in $T$ if:

1. $v$ is $u$, or
2. $v$ is a child of some node $w$, where $w$ is a descendant of $u$.

*Proper descendant*: $v \neq u$

*Left descendant*: $u$ itself, or descendant of left child of $u$

*Right descendant*: $u$ itself, or descendant of right child of $u$

**Q**: How would you define "$v$ is *to the left of* $u$"?

**Q**: How would you define descendant without using recursion?

## 2.1 Size and Height of Binary Trees

The *size* of a binary tree is the number of (internal) nodes.

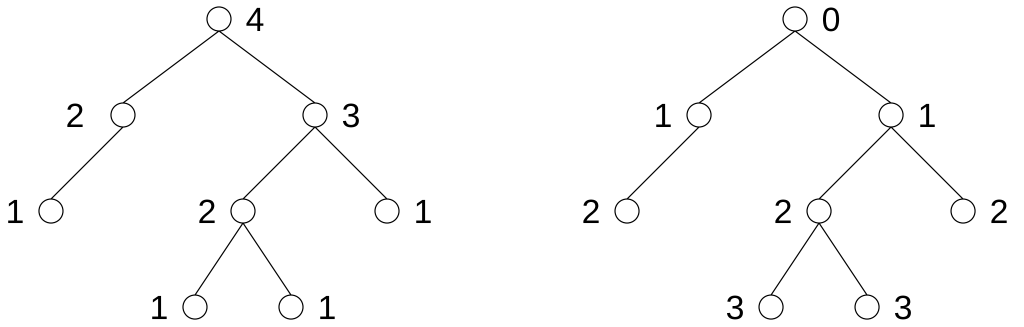
The *height* of a binary tree $T$ is the length of the longest chain of descendants. That is:

- $0$ if $T$ is empty,
- $1 + \max(height(T_1), height(T_2))$ otherwise, where $T_1$ and $T_2$ are subtrees of the root.

The height of a node $u$ is the height of the subtree rooted at $u$.

The *level* of a node is the "distance" from the root. That is:

- $0$ for the root node,
- $1$ plus the level of the node's parent, otherwise.

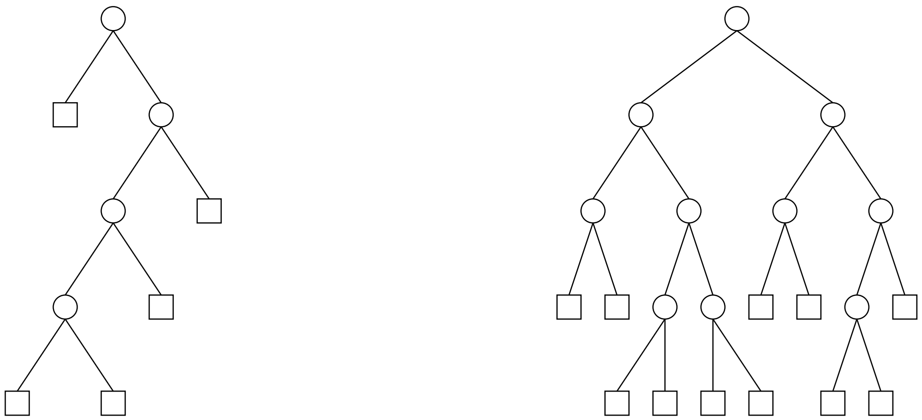


## 2.2 Skinny and Complete Trees

Since we will be doing performance analyses of tree representations, we will be interested in worst cases for height vs size.

*skinny* — every node has at most one child (internal) node

*complete (fat)* — external nodes (and hence leaves) appear on at most two adjacent levels

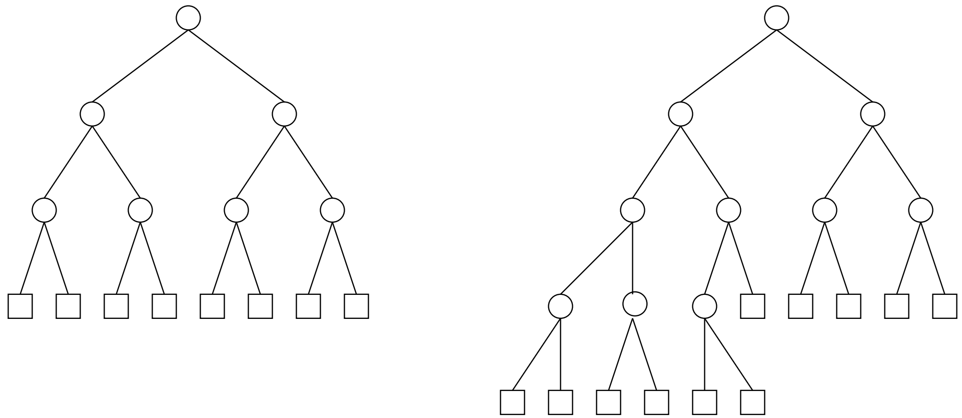


For a given size, skinny trees are the highest possible, and complete trees the lowest possible.

We also identify the following subclasses of complete:

*perfect* — all external nodes (and leaves) on one level

*left-complete* — leaves at lowest level are in leftmost position



## 2.3 Relationships between Height and Size

The above relationships can be formalised/extended to the following:

1. A binary tree of height $h$ has size at least $h$.
2. A binary tree of height $h$ has size at most $2^h - 1$.
3. A binary tree of size $n$ has height at most $n$.
4. A binary tree of size $n$ has height at least $\lceil \log(n + 1) \rceil$.

### Exercise

For each of the above, what class of binary tree represents an upper or lower bound?

### Exercise

Prove (2).

# 3. Trees, Forests, and Orchards

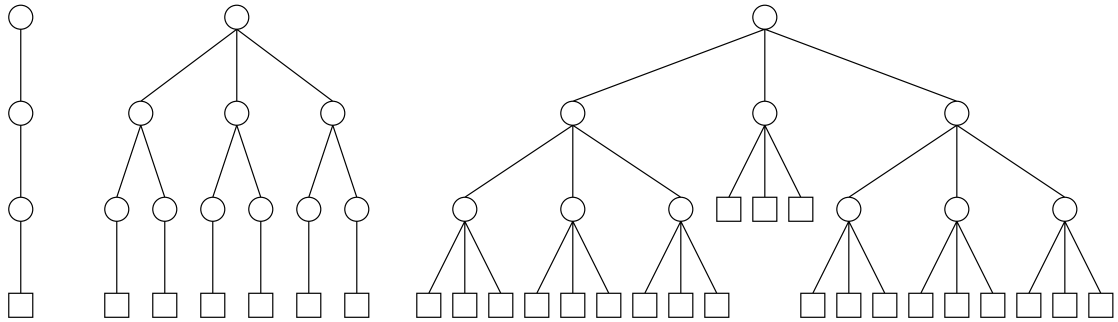
A general *tree* or *multiway (indexed) tree* is defined in a similar way to a binary tree except that a parent node does not need to have exactly two children.

## Definition

A *multiway (indexed) tree* $T$ of $n$ nodes, $n \geq 0$, either:

- is empty, if $n = 0$, or
- consists of a root node $u$, an integer $d \geq 1$ called the *degree* of $u$, and $d$ multiway trees $u(1), u(2), \ldots, u(d)$ with sizes $n_1, n_2, \ldots, n_d$ respectively such that

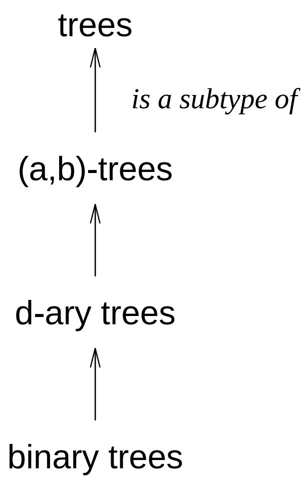
$$n = 1 + n_1 + n_2 + \cdots + n_d.$$

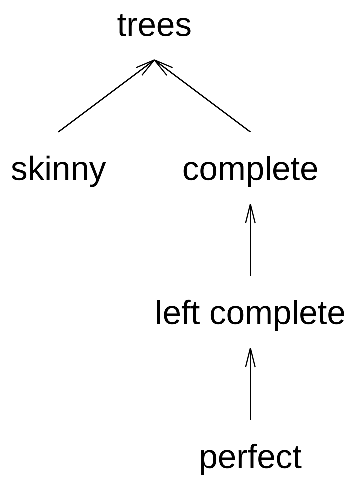


A tree is a $d$-*ary tree* if $d_u = d$ for all (internal) nodes $u$. We have already looked at binary (2-ary) trees. Above is a unary (1-ary) tree and a ternary (3-ary) tree.

A tree is an $(a, b)$-*tree* if $a \leq d_u \leq b$, $(a, b \geq 1)$, for all $u$. Thus the above are all (1,3)-trees, and a binary tree is a (2,2)-tree.

# Some trees of tree types!



## 3.1 Forests and Orchards

Removing the root of a tree leaves a collection of trees called a *forest*. An ordered forest is called an *orchard*. Thus:

*forest* — (possibly empty) set of trees

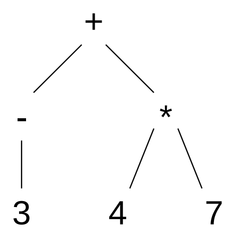
*orchard* — (possibly empty) queue or list of trees

## 3.2 Annotating Trees

The trees defined so far have no values associated with nodes. In practice it is normally such values that make them useful.

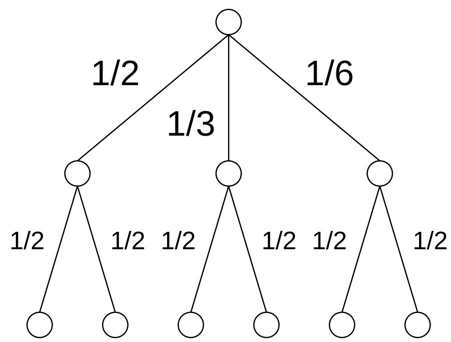
We call these values *annotations* or *labels*.

eg. a *syntax* or *formation* tree for the expression $-3 + 4 * 7$



eg. The following is a probability tree for a problem like:

> "Of the students entering a language course, one half study French, one third Indonesian, and one sixth Warlpiri. In each stream, half the students choose project work and half choose work experience. What is the probability that Björk, a student on the course, is doing Warlpiri with work experience?"




In examples such as this one, it often seems more natural to associate labels with the "arcs" joining nodes. However, this is equivalent to moving the values down to the nodes.

As with the list ADT, we will associate elements with the nodes.

# 4. Tree Traversals

Why traverse?

- search for a particular item
- test equality (isomorphism)
- copy
- create
- display

We'll consider two of the simplest and most common techniques:

*depth-first* — follow branches from root to leaves

*breadth-first (level-order)* — visit nodes level by level

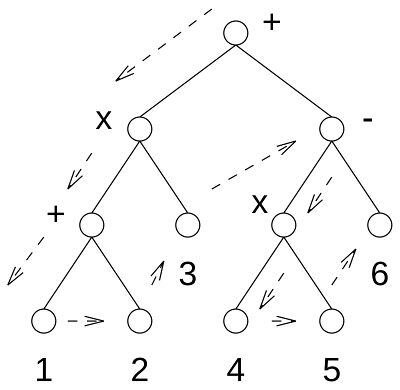
(More in Algorithms or Algorithms for AI. . . !)

## 4.1 Depth-first Traversal

### Preorder Traversal

(Common garden "left to right", "backtracking", depth-first search!)

```
if(!t.isEmpty()) {
   visit root of t;
   perform preorder traversal of left subtree;
   perform preorder traversal of right subtree;
}
```



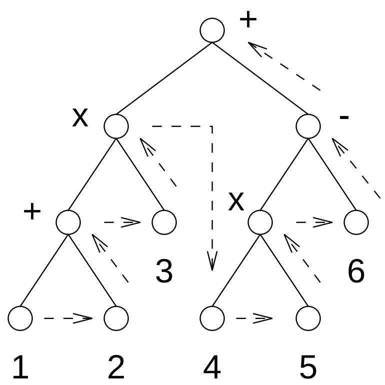
(Generates a *prefix expression*

$$+ \times + 1\,2\,3 - \times 4\,5\,6$$

Sometimes used because no brackets are needed — no ambiguity.)

## Postorder Traversal

```
if(!t.isEmpty()) {
   perform postorder traversal of left subtree;
   perform postorder traversal of right subtree;
   visit root of t;
}
```



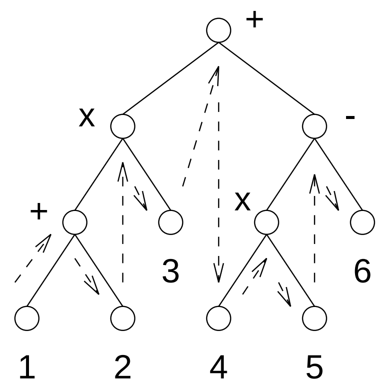

(Generates a *postfix expression*

$$1\ 2 + 3 \times 4\ \ 5 \times 6 - +$$

Also non-ambiguous — as used by, eg. HP calculators.)

# Inorder Traversal

```
if(!t.isEmpty()) {
   perform inorder traversal of left subtree;
   visit root of t;
   perform inorder traversal of right subtree;
}
```



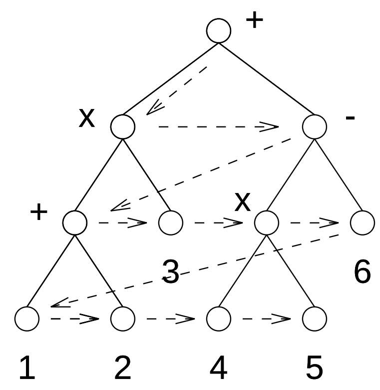

(Generates an *infix expression*

$$1 + 2 \times 3 + 4 \times 5 - 6$$

Common, easy to read, but ambiguous.)

## 4.2 Level-order (Breadth-first) Traversal

Starting at root, visit nodes level by level (left to right):




Doesn't suit recursive approach. Have to jump from subtree to subtree.

Solution:

- need to keep track of subtrees yet to be visited — ie need a data structure to hold (windows to) subtrees (or Orchard)
- each internal node visited spawns two new subtrees
- new subtrees visited *only after* those already waiting

⇒ Queue of (windows to) subtrees!

## Algorithm

```
place tree (root window) in empty queue q;
while (!q.isEmpty()) {
   dequeue first item;
   if (!external node) {
      visit its root node;
      enqueue left subtree (root window);
      enqueue right subtree (root window);
   }
}
```

## 4.3 Traversal Analysis

### Time

The traversals we have outlined all take $O(n)$ time for a binary tree of size $n$.

Since all $n$ nodes must be visited, we require $\Omega(n)$ time
$\Rightarrow$ asymptotic performance cannot be improved.

## Space

*Depth-first*: Recursive implementation requires memory (from Java's local variable stack) for each method call $\Rightarrow$ proportional to height of tree

- worst case: skinny, size $n$ implies height $n$
- expected case: much better (depends on distribution considered — see Wood Section 5.3.3)
- best case: *exercise*...

Iterative implementation is also possible.

*Level-order*: Require memory for queue.

Depends on tree *width* — maximum number of nodes on a single level.

Maximum length of queue is bounded by twice the width.

- best case: skinny, width 2
- worst case: *exercise*...

## 5. Summary

- Trees are not only common "in their own right", but form a basis for many other data structures.
- Definitions — binary trees, trees, forests, orchards, annotated trees
- Properties — size, height, level, skinny, complete, perfect, $d$-ary, $(a, b)$
- Covered important, common traversal strategies
  - depth-first: preorder, postorder, inorder
  - level-order (breadth-first)

Next — tree representations. . .