Trees: Basic terminology

Tree = set of nodes and edges, \( T = (V, E) \).

Nodes \( v \in V \) store data items in a parent-child relationship.

A parent-child relation between nodes \( u \) and \( v \) is shown as a directed edge \((u, v) \in E\), where the direction is from \( u \) to \( v \).

Each node has at most one parent node; may have many siblings.

There is exactly one node that has no parent – the root node.

The degree of a node \( v \) is the number of children.

A node with 0 children is a leaf node or external node.
The other nodes are internal.

Example: \((a, b, c, d), ((a,b),(a,c),(c,d))\)

- File system in UNIX
- Hierarchical classification systems
- Decision trees (e.g., binary search analysis Lecture 4)
- Hierarchical organization of...
  - Organizations: department, division, group, ...
  - Documents: book, chapter, section, ...
  - XML documents
- Representing order or priority...
  - expression tree for e.g. \(4 + 5 * 3!\)
Trees: Basic terminology (cont.)

A path is a sequence of nodes \( (v_1, v_2, \ldots, v_k) \) such that \( (v_i, v_{i+1}) \) is an edge for \( i = 1, \ldots, k-1 \).

A node \( n \) is an ancestor of a node \( v \) if there exists a path from \( n \) to \( v \) in \( T \).

A node \( n \) is a descendant of a node \( v \) if there exists a path from \( v \) to \( n \) in \( T \).

The depth \( d(v) \) of a node \( v \) is the length of the path from the root to \( v \).

The height \( h(v) \) of a node \( v \) is the length of the longest path from \( v \) to a descendant of \( v \).

The height \( h(T) \) of a tree \( T \) is the height of the root.

Special kinds of trees

- Ordered tree: linear order among the children of each node
- Binary tree: ordered tree with degree \( \leq 2 \) for each node (left child, right child...)
- Empty binary tree (\( \lambda \)): binary tree with no nodes
- Full binary tree: nonempty; degree is either 0 or 2 for each node
  - Fact: number of leaves = 1 + number of interior nodes (proof by induction)
- Perfect binary tree: full, all leaves have the same depth
  - Fact: number of leaves = \( 2^h \) for a perfect binary tree of height \( h \) (proof by induction on \( h \))
- Complete binary tree: approximation to a perfect tree for \( 2^h \leq n < 2^{h+1} - 1 \) (important and useful property!)

ADT Tree

Operations on a single tree node \( v \) of a tree \( T \):
- parent \( (v) \) returns parent of \( v \), error if \( v \) root
- children \( (v) \) returns set of children of \( v \)
- firstChild \( (v) \) returns first child of \( v \), or \( \lambda \) if \( v \) leaf
- rightSibling \( (v) \) returns right sibling of \( v \), or \( \lambda \) if none
- leftSibling \( (v) \) returns left sibling of \( v \), or \( \lambda \) if none
- isLeaf \( (v) \) returns true iff \( v \) is a leaf (an external node)
- isInternal \( (v) \) returns true iff \( v \) is a non-leaf node
- isRoot \( (v) \) returns true iff \( v \) is the root
- depth \( (v) \) returns depth of \( v \) in \( T \)
- height \( (v) \) returns height of \( v \) in \( T \)

ADT Tree (cont.)

Operations on entire tree \( T \):
- size \((v)\) returns number of nodes of \( T \)
- root \((v)\) returns root node of \( T \)
- height \((v)\) returns the height of \( T \)

In addition for a binary tree:
- left \((v)\) returns the left child of \( v \), or error
- right \((v)\) returns the right child of \( v \), or error
- hasLeft \((v)\) test if \( v \) has the left child
- hasRight \((v)\) test if \( v \) has the right child
**Tree representation 1: pointers...**

Type `Tnode` denotes a pointer to a structure storing node information:

```
record node_record
    nchilds: integer
    child: table<Tnode>[1..nchilds]
    info: infotype
```

- For binary trees: 2 pointers per node, `LC` and `RC`.
- Alternatively, the pointers to a node’s children can be stored in a linked list.
- If required, a “backward” pointer to the parent node can be added.

**Tree representation 2: Sequential memory**

For a complete binary tree holds:

There is exactly one complete binary tree with \( n \) nodes.

Implicit representation of edges:

Numbering of nodes \( \rightarrow \) index positions

\[ \text{Numbering starts at 1: root numbered 1: extends to binary trees which are not complete} \]

---

**2: Sequential memory (cont.)**

Use a table<key,info>[0..\( n-1 \)]

- `leftChild(i) = 2i+1` (\( \) if \( 2i+1 \geq n \))
- `rightChild(i) = 2i+2` (\( \) if \( 2i+1 \geq n \))
- `isLeaf(i) = i < n` and `2i+1 > n`
- `leftSibling(i) = i-1` (\( \) if \( i=0 \) or odd(\( i \)))
- `rightSibling(i) = i+1` (\( \) if \( i \) \( \) or even(\( i \)))
- `parent(i) = [i-1]/2` (\( \) if \( i \) \( \))
- `isRoot(i) = (i = 0)`

---

**Tree Traversals**

Regard a tree \( T \) as a building:

- nodes as rooms, edges as doors, root as entry
- How to explore an unknown (acyclic) labyrinth and get out again?
- Proceed by always keeping a wall to the right!

Generic tree traversal routine:

```
procedure visit(node v)
    for all u ∈ children(v) do
        visit(u)
```

Call visit(Root(\( T \))) and each node in \( T \) will be visited exactly once!
### Tree Traversals (3 of 4)

**procedure** `preorder_visit(node v)`

- `do_something(v)` {before any children}
- `for all u ∈ children(v)` do `preorder_visit(u)`

**procedure** `postorder_visit(node v)`

- `for all u ∈ children(v)` do `postorder_visit(u)`
- `do_something(v)` {after all children}

**procedure** `inorder_visit(node v)` {binary trees only!}

- `inorder_visit(leftChild(v))`
- `do_something(v)` {after all children}
- `inorder_visit(rightChild(v))`

### Tree Traversals (#4)

**procedure** `level_order_visit(node v)`

- `Q ← mkEmptyQueue()`
- `enqueue(v, Q)`
- `while not isEmpty(Q)` do
  - `v ← dequeue(Q)`
  - `do_something(v)`
  - `for all u ∈ children(v)` do `enqueue(u, Q)`

### Binary Search Tree (G&T 10.1)

A binary search tree (BST) is a binary tree such that:
- Information associated with a node includes a key, → linear ordering of nodes determined by keys.
- The key of each node is:
  - greater than (or equal) the keys of all left descendents, and
  - smaller than (or equal) the keys of all right descendents.
- The leaves do not store any information

### ADT Map as Binary Search Tree...

**lookUp(k, v):** comparison controlled traversal

- if `key(v) = k` then return `k`
- else if `k < key(v)` then `lookUp(k, leftChild(v))`
- else `lookUp(k, rightChild(v))`

- worst-case: `height(T)+1` comparisons

**insert(k, x):** add `(k, x)` as new leaf on `lookUp` failure or update the node on `lookUp` success

- worst-case: `height(T)+1` comparisons

**remove(k):** `lookUp`, then...

- if `v` is a leaf, remove `v`
- if `v` has one child `u`, replace `v` by `u`
- if `v` has two children, replace `v` by its inorder successor (alternatively: by its inorder predecessor)

- worst-case: `height(T)+1` comparisons
BST: s are not unique
Same data may generate different BST

insert: 1, 2, 4, 5, 8

insert: 5, 2, 1, 4, 8

Successful Lookup: Worst and Average case

Worst Case BST
- BST degenerated to a linear sequence
- Expected number of comparisons is \((n + 1)/2\)

Balanced BST
- The depths of leaves differ by at most 1
- \(O(\log_2 n)\) comparisons.

Thus – Let’s keep ‘em balanced!
Some commonly used balanced trees (to be discussed):
- AVL Trees
- (2,3) Trees (or (2,4) Trees), (a,b)-Trees,
- …Red-Black Trees, B-Trees
- Splay Trees

Exercise
Discuss implementation of ADT Dictionary as [G&T] variant of BST
- \(\text{Insert}(k,v)\) - no update of entries with the same key
- \(\text{findAll}(k)\) – \(k\) may occur in many nodes