General Trees

Tree - a hierarchical datastructure
- nodes (vertices) - "bubbles", store information
- parent/children relationship
- root ✓ topmost node
- leaves (external nodes) - have no children, e.g. we have 7 leaves
- ancestor - any node "above" me, e.g. for A it is 3 ancestors
- descendant - anybody "below" me is my descendant, e.g. 4 has 10 descendants

[Section 7.1]
General Trees

Tree, cont.

- **subtree** - take a node, and take all its descendants
- **edges** - a connection between two nodes
- **paths** - a set of connecting edges from one node to another, e.g., exactly one path between every pair of nodes
- **ordered**
- **size** = # of nodes

- **height/depth**
  
  = # levels or the length of the longest path from the root to any other node
  
  → most books would count the height/depth of our example as 4 (not 5, but really no difference with Big-O notation)

  \[
  \text{height}(v) = \text{the length of the path from the root to } v \quad (\text{distance from the root})
  \]

  \[
  \text{depth}(v) = \text{the max length of a path from } v \text{ to any of its descendants}
  \]

  \(v\) - a node
How to implement?

- Which classes we need? And methods?

- **NodeT class**
  - (book uses a different name than Node)
  - key
  - parent
  - linked list of all its children
  - + all set/get methods

- **Tree class**
  - root \(\rightarrow\) NodeT
  - maybe size
  - - methods: getSize, getHeight, getDepth, traversals - how to list/print all nodes etc. - depending on the specific application
How to compute the depth of a node v?

Pseudocode: (detailed)

DEPTH (input: NodeTv)
Output: depth(v)

1. if v.getChildren() == null then
2. return 0;
3. let l = v.getChildren();
4. max = 0;
5. let tmp = l.getHead();
6. while (tmp != null) { //while I haven't gone through all children
7. ch = tmp.getNext();
8. d = DEPTH(ch);
9. if (max < d) then max = d;
10. tmp = tmp.getNext();
11. }
12. return max + 1;

Redefine the depth(v):
- maximum depth of v's children + 1
- 0 if v is a leaf

Children: w1 -> w2 -> w3

Depth of v = 3

\( d(w_1) = 2 \)
\( d(w_2) = 1 \)
\( d(w_3) = 0 \)
\( d(w_4) = 0 \)
\( d(w_c) = 0 \)
How to compute the height of a node $v$?

Pseudocode:

```
Pseudocode: \text{ input}

\text{HEIGHT(Node\text{v})}
\text{ Output: height(v)}
1. $h = 0$
2. $tmp = v$
3. \textbf{while} ($tmp \neq \text{null}$) \textbf{do}
4. \hspace{1em} $tmp = tmp \cdot \text{getParent()}$
5. \hspace{1em} $h++$
6. \textbf{end while}$
7. \textbf{return} $h-1$
```

Melinda: can change to $tmp = root$ and return $h$ (careful with $v=\text{null}$).
How to compute the height of the tree?
(nonrecursive vs recursive approach)
Tree Traversal Algorithms

Preorder Traversal

Goal: visit every node and perform an "action" on each node

TRAVERSALS

PREORDERTRAV

Input: tree
Output: no specific output but action performed on every node
1. perform action on the root; // e.g., print root.getKey()
2. recursively traverse the trees with roots = children of the root

PREORDERTRAV (Input: node v)
1. print v.getKey()
2. for every child a of v do
3.   PREORDERTRAV (a)

Note: also known as the depth-first-search (DFS) on trees
Tree Traversal Algorithms

Preorder and Postorder Traversals

**POSTORDERTRAV** (Input: nodeT v)
1. for every child a of v do
2.  **POSTORDERTRAV** (a)
3.  print v.getKey () // or any other action

call: **POSTORDERTRAV**(root)

where \( n = \) #nodes

**Running time:**
- all steps 3: \( O(n) \) steps (assuming \( O(1) \) action per node)
- all steps 2: 1 call per node: total \( O(n) \) recursive calls \( \rightarrow O(n) \) steps
- all steps 1: sum over all nodes of the number of children of the nodes \( = n-1 \) \( \Rightarrow \) \( O(n) \) steps

**OVERALL:** \( O(n) \) steps

prints for this input:
5, 6, 2, 11, 12, 7, 3, 8, 9, 13, 14, 10, 4, 1

postorder:
- first visit children,
  then do the action
Binary Trees

- every node has at most 2 children (left, right)
- proper/full: every node has 0 or 2 children

Suppose we have a binary tree of height \( d \):

- What is the maximum number of its leaves?
- What is the maximum number of its nodes?

\[
\sum_{i=0}^{d} 2^i = 2^0 + 2^1 + 2^2 + \cdots + 2^d = 2^{d+1} - 1 \quad \Rightarrow \text{therefore} \quad n \leq 2^{d+1} - 1
\]

\[
\Rightarrow n \leq 2^{d+1} - 1 \quad \Rightarrow \quad d+1 \geq \log n
\]
Binary Search Trees

- a nonlinear datastructure for storing keys
- for every node the following holds:
  - the left child’s key is smaller than the key of its parent
  - the right child’s key is larger than the key of its parent
Binary Search Trees

Searching

Recursive Pseudocode:

```
SEARCH (Input: a key x, a nodeT v)
0. if v == null
   print "NOT FOUND"; exit;
1. let y = v.getkey()
2. if x < y then
3.   SEARCH(x, v.getLeft())
4. else
5.   if x > y then
6.   SEARCH(x, v.getRight())
7. else
8.   print "FOUND"
```

Non-recursive Pseudocode:

```
SEARCH (Input: a key x)
1. tmp = root
2. found = false
3. while (tmp != null && !found) {
4.   let y = tmp.getkey()
5.   if x < y then tmp = tmp.getLeft()
6. else if x > y then tmp = tmp.getRight()
7. else found = true
8. }
9. return found
```

If the tree is so-called balanced, then searching takes \( O(\log n) \) time.

Essentially a (sorted) linked list.

- If searching for the largest, we'll go through everybody.
- \( O(n) \) time.

Run time:
- \( O(\text{depth}) \)
Binary Search Trees

Inserting an element

Idea: follow left-right relationships to find the position of a new key $x$ then insert $x$ as a new leaf

Pseudocode: very similar to previous slide, except instead of the return, we insert a new leaf
Binary Search Trees

Deleting an element
Binary Search Trees

Running time of the operations search, insert, delete