12.1 Binary Trees

A tree is a container of positions arranged in child–parent relationship. A tree consists of nodes: we speak of parent and child nodes. In a binary tree, each node has two possible children: a left and right child. A leaf node is one without children; otherwise it is an internal node. There is one special node called the root.

Examples include:
- father/family tree
- UNIX file system: each node is a level of grouping
- decision/taxonomy tree: each internal node is a question

For example, here is an expression tree that stores the expression \((7 + 3) \times (8 - 2)\):

The descendants of a node are its children, their children etc. A node and its descendants form a subtree. A node \(u\) is ancestor of \(v\) if and only if \(v\) is descendant of \(u\). The depth of a node is the number of ancestors (excluding itself); that is, how many steps away from the root it is. Here is a binary tree with the nodes’ depths marked.

Special trees: A binary tree is proper/full if every internal node has two children. A binary tree is complete if it is full and every leaf has the same depth. (NOTE: different books have different definitions.)
Note that:

- A complete tree of depth $d$ has $2^d$ leaves and $2^{d+1} - 1$ nodes in total.
- A full tree has one more leaf than internal node.

(Exercise to reader: prove these by induction.)

12.2 Implementation with Links

Each node contains some data and pointers to its two children. The overall tree is represented as a pointer to the root node.

```c
struct BTNode {
    <type> data;
    BTNode *left;
    BTNode *right;
};
```

If there is no child, then that child pointer is `nullptr`. It is common for tree methods to return `nullptr` when a child does not exist (rather than print an error message or throw an Exception).

Methods might include:
- get’s and set’s (data and children)
- isLeaf
- modification methods: add or remove nodes

For a general tree, there are two standard approaches:

- each node contains a collection of references to children, or
- each node contains references to `firstChild` and `nextSibling`.
12.3 Animal Guessing Example

(Based on Main.) The computer asks a series of questions to determine a mystery animal. The data is stored as a **decision tree**. This is a full binary tree where each internal node stores a question: one child is associated with yes, one with no. Each leaf stores an animal.

The program moves down the tree, asking the question and moving to the appropriate child. When a leaf is reached, the computer has identified the animal. The cool idea is that if the program is wrong, it can automatically update the decision tree: If the program is unsuccessful in a guess, it prompts the user to provide a question that differentiates its answer from the actual answer. Then it replaces the relevant node by a guess and two children.

Code for such a method might look something like:

```cpp
void replace(Node *v, string quest, string yes, string no) {
    v->data = quest;
    v->left = new Node(yes);
    v->right = new Node(no);
}
```

assuming a suitable constructor for the class **Node**.

12.4 Tree Traversals

A **traversal** is a systematic way of accessing or visiting all nodes. The three standard traversals are called preorder, inorder, and postorder. We will discuss **inorder** later.

In a **preorder traversal**, a node is visited before children (so the root is first). It is simplest when expressed using recursion. The main routine calls **preorder(root)**

```cpp
preorder(Node *v) {
    visit node v
    preorder ( left child of v )
    preorder ( right child of v )
}
```

Here is a tree with the nodes labeled by preorder:
The standard application of a preorder traversal is printing a tree in a special way: for example, the indented printout below:

\[
\begin{array}{c}
\text{root} & \rightarrow & \text{left} & \rightarrow & \text{leftLeft} \\
& & & \rightarrow & \text{leftRight} \\
& & \rightarrow & \text{right} & \rightarrow & \text{rightLeft} \\
& & & \rightarrow & \text{rightRight} \\
\end{array}
\]

The most common traversal is a **postorder traversal**. In this, each node is visited after its children (so the root is last). Here is a tree labeled with postorder:

```
1
2 4
3 5
10 9
6 8 7
```

Examples include computation of disk-space of directories, or maximum depth of a leaf. For the latter:

```c
int maxDepth(Node *v) {
    if( v->isLeaf() )
        return 0;
    else {
        int leftDepth=0, rightDepth=0;
        if( v->left )
            leftDepth = maxDepth (v->left) ;
        if( v->right )
            rightDepth = maxDepth (v->right) ;
        return 1 + max( leftDepth, rightDepth );
    }
}
```

For the code, the time is proportional to the size of the tree, that is, it is \( O(n) \).

**Practice.** Calculate the size (number of nodes) of the tree using recursion.