

Trees

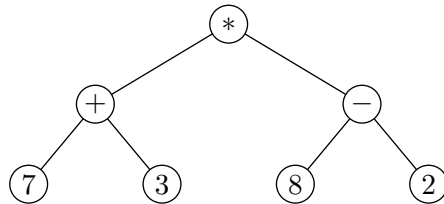
13.1 Binary Trees

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*.

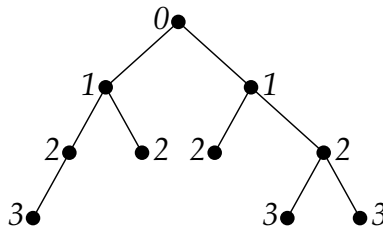
Examples include:

- 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 $(7 + 3) * (8 - 2)$:



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.)

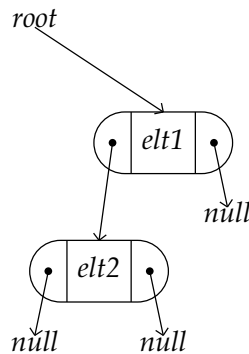


13.2 Implementation with Links

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

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

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



Methods might include:

- gets and sets (data and children)
- isLeaf
- modification methods: add or remove nodes

For a general tree, either each node contains a collection of references to children, or each node contains references to *firstChild* and *nextSibling*.

13.3 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:

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

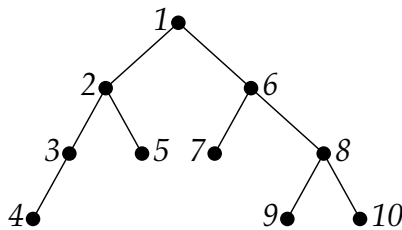
assuming a suitable constructor for the class Node.

13.4 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)`

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

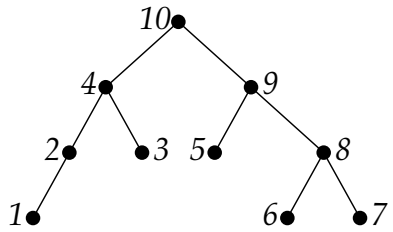


The standard application of a preorder traversal is printing a tree in a special way: for example, the indented printout below:

```

root  — left   — leftLeft
      — right  — leftRight
      — right  — rightLeft
      — right  — rightRight
  
```

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



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

```

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

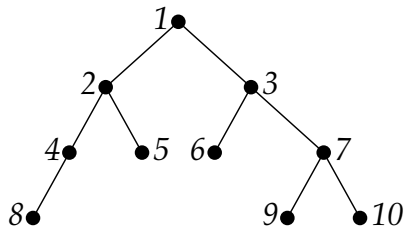
```

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

13.5 Breadth-first and Depth-first Search

A *search* is a systematic way of searching through the nodes for a specific node. The two standard searches are breadth-first search and depth-first search.

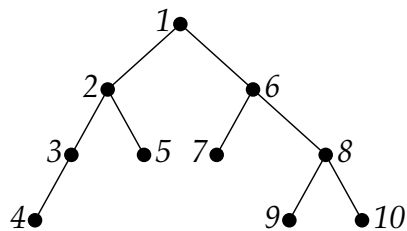
In a *breadth-first search*, the tree is visited one level at a time. It uses a *queue*: each time a node is visited, one adds its children to the queue of nodes to be visited. The next node to be visited is extracted from the front of the queue. Thus one visits the root, then the root's children, then the nodes at depth 2, and so on.



```
Add the root node to the queue.
While (queue not empty) and (goal not found) {
  current = front node removed from queue
  enqueue all of current's children (left then right)
}
```

In a *depth-first search*, the search continues going deeper into the tree whenever possible. When the search reaches a leaf, it backtracks to the last (visited) node that has un-visited children, and continues searching from there. A depth-first-search can use a *stack*: each time a node is visited, its right child is pushed onto the stack for later use while its left child is explored next. When one reaches a leaf-node, one pops off the stack.

```
Add the root node to the stack.
While (Stack not empty) and (Goal not reached) {
  current = node popped from stack
  push all of current's children (right then left)
}
```



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