**Trees**

**Trees**

**Table of contents**

1. [Introduction](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#intro)
2. [Types of trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#types)
   * [Free trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#freeTrees)
   * [Rooted trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#rootedTrees)
   * [Ordered trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#orderedTrees)
   * [M-ary treesar](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#MaryTrees)
     + [Binary trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#binaryTrees)
3. [Recursive definitions and representations](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#recurDef)
   * [Binary trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#binaryTreesDef)
   * [M-ary trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#MaryTreesDef)
   * [Free trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#freeTreesDef)
   * [Rooted trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#rootedTreesDef)
   * [Ordered trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#orderedTreesDef)
4. [Graphs](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#graphs)
5. [Mathematical properties of binary trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#mathProps)
6. [Tree traversal](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#traversal)
7. [Some recursive tree algorithms/methods](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#recurMethods)
8. [Discussion](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#discussion)

**Introduction**

Before defining what a *tree* is, let's take a look at examples of different types of trees we use everyday.

* Files and directories on a computer (starting from '/' on Unix-like systems, or "C:\" on Windows); structure of the CSE 220 website.

|  |
| --- |
| CSE 220  |  --------------------------------  / | | | \  Messages Lectures Exams Assign References |

* positions in an organizations (starting from say the managing director or the CEO).
* **Arithmetic expression trees** — remember how we used postfix expression to implement a simple stack-based calculator? We can most naturally convert a prefix expression to a tree, and implement a calculator. For example, the infix expression (5 + 3) \* 8 is \* + 5 3 8 in prefix form, which can be represented as the following (expression) tree.

|  |
| --- |
| \*  / \  + 8  / \  5 3 |

* the structure of the contents in a book - the book has chapters, chapters have sections, and so on.
* specialized trees - binary search trees, heap ordered trees, and many many others.

A **tree** is made up of **nodes**, with **edges** or **links** connecting the nodes. Defined recursively:

|  |
| --- |
| A **tree** is either an empty tree, or a node connected to a set of  **subtrees**. |

Back to [Table of contents](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#toc)

**Types of trees**

The textbook discusses the following types of trees:

* [Free trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#freeTrees)
* [Rooted trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#rootedTrees)
* [Ordered trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#orderedTrees)
* [M-ary trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#MaryTrees)
  + [Binary trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#binaryTrees)

**Free trees**

A free tree (or just "a tree") is a set of nodes/vertices, with links/edges connecting pairs of distinct nodes. The defining property is that each pair of nodes must have exactly one simple path (simple path means that the nodes in a path are distinct, except perhaps the first and the last). The other way to say this is that each node is connected to every other node, and there are no cycles. This is the most general type of trees.

If there are none or multiple paths (cycles), then it's a **graph**! **Forests** are disjoint set of trees.

**Rooted trees**

A Rooted tree has a **distinguished node** that acts as the **root** (much like the **head** of a linked list), where it all starts. There is exactly one simple path between the root and every other node in the tree ("connected"), and there is no direction implied in these paths. Some definitions/concepts:

* **above**, **below**, **ancestor** and **descendant**: a node y is considered "below" node x if there is a path from root to y that touches x, and x is "above" y consequently. Y is a descendant of x (y below x) or x is an ancestor of y (x above x).
* **sibling**, **parent**, **children**, **grandparent**, **grandchildren**: the sibling nodes share the same parent. The immediate ancestor is called the parent, and immediate descendants children; grandparent and grandchildren are one step away in either direction. A node may have many children and many siblings, but only ONE parent.
* **leaves/terminal** nodes, **non-terminal** nodes: nodes with no children are called leaves or terminal nodes (because it's the end of the road for the links!). Similarly, nodes with at least one child are called non-terminal nodes. In trees representing recursion, the recursion is represented by the non-terminal nodes, and the non-recursive part by the terminal nodes.

**Ordered trees**

An ordered trees is a rooted tree in which the children of a node are ordered in a particular way. For example, the chapters in a book (ordered from earlier to later), the children in a family tree (may be ordered according to age), the order of the arguments to matrix multiplication (in expression trees), etc.

**M-ary trees**

An M-ary tree is an ordered tree in which each node must have **exactly** M children, which are also M-ary trees. For these trees, we use special nodes called **external** nodes (essentially *sentinel* nodes), which have no children. These external nodes representing leaves are essentially dummy nodes to satisfy the M-ary property. The **non-terminal** nodes are those that have at least one child that is not "external". The "leaf" in an M-ary tree is then an internal node with M children all of which are external.

**Binary trees**

A binary tree is the simplest (and most widely used) M-ary tree with M=2. We'll spend lots of time on binary trees.

Looking at the "is-a" hierarchy -

|  |
| --- |
| Binary/M-ary tree -> ordered tree -> rooted tree -> (free) tree |

Back to [Table of contents](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#toc)

**Recursive definitions and representations**

* [Binary trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#binaryTreesDef)
* [M-ary trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#MaryTreesDef)
* [Free trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#freeTreesDef)
* [Rooted trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#rootedTreesDef)
* [Ordered trees](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#orderedTreesDef)

**Binary trees**

Let's start with the recursive definition of a **binary tree**.

|  |
| --- |
| A binary is either: an external node (ie., an empty tree)  or: or a an internal node connected to an ordered pair of  binary trees, called the left and right subtrees. |

Given this recursive definition of a binary tree, the question now is how to represent it. One obvious way to represent the tree structure in a node is to use references to the children subtrees, and using a "sentinel" to represent an external node or the empty tree. In Java for example, we can use the null as the sentinel for an external. When using linked representation, we can maintain two different references - one to the left subtree and one to the right subtree. We can of course also use an array of two references, with the 1st element referring to the left and the 2nd element referring to the right subtree respectively. The first choice seems more natural.

|  |
| --- |
| public class Node {  Object item; // the key within the node  Node left; // reference to the left child  Node right; // reference to the right child  public Node(Object i, Node l, Node r) {  item = i; left = l; right = r;  }  } |

With this representation, one can move to the left subtree with a reference assignment such as n = n.left, or to the right subtree with n = n.right.

We may also need to move "up" the tree, for which it may be convenient to also maintain a **parent** reference in each node. The root node does not have a parent, so we'll use the null as a sentinel. We'll see a need for that later on when we discuss **binary search trees**. Adding the **parent** reference to the Node class above, we get the following:

|  |
| --- |
| public class Node {  Object item; // the key within the node  Node left; // reference to the left child  Node right; // reference to the right child  Node parent; // reference to the parent node  public Node(Object i, Node l, Node r, Node p) {  item = i; left = l; right = r, parent = p;  }  } |

This is the so-called **linked** representation of a binary tree, where we use references to nodes as the links. There is also an **array** representation of trees, in which each node is represented by a specific index into an array. The idea is very simple: the root node is kept at index 1, and it's left and right children in indices 2 and 3. In general, any node at index k have children at indices 2k and 2k+1, and parent at floor(k/2). If 2k+1 is > N, where N is the number of internal nodes, then the node does not have a right child; if 2k > N, then it does not have a left child either. There is potentially a lot of wasted space unless the tree is "almost complete" - a binary tree that has all the levels filled, except possibly for the last level which is filled from the left. Much more on "almost complete" binary trees and array representation when we study "heap ordered" trees later this semester.

**M-ary trees**

The recursive definition of a **M-ary** tree is almost the same as the binary tree (remember that a binary tree is just an M-ary tree with M=2):

|  |
| --- |
| An M-ary tree is either: an external node (empty tree)  or: a node connected to an ordered sequence of M  trees, all of which are M-ary trees. |

The representation is typically linked, using either explicit references to the children (left, middle and right for a ternary tree), or using an array of references for the children. If M > 3, it's probably easier to use an array.

|  |
| --- |
| public class Node {  Object item; // the key within the node  Node[] children; // references to the left-to-right children  public Node(Object i, Node[] c) {  item = i;  children = new Node[c.length];  System.arraycopy(c, 0, children, 0, c.length);  }  } |

**Ordered trees**

A node (called the root) connected to an ordered sequence of disjoint trees. The difference between this and an M-ary tree is that a rooted tree can have nodes with any number of children. Since the number of children varies from node to node, it may be better to use a linked list to represent the children in a particular node.

**Rooted trees**

Same as an ordered rooted tree, just that there no ordering in the children, so there are many representations of the same tree (eg., permuting the order of the children does not change the tree).

**Free trees**

Since free trees do not have a distinguished node that acts as the root, it is typically best to represent this type of tree using a graph representation. We'll see more about this when we study graphs.

Back to [Table of contents](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#toc)

**Graphs**

Since we're talking about more and more general types of trees, good to introduce graphs. We've already seen graphs when discussing the Internet (or the FriendsNet assignment, which looks at friendships among people).

A graph is a set of nodes or vertices, with a set of edges that connect pairs of distinct nodes (with at most one edge connecting a pair of nodes).

A tree is nothing more than a graph with specific properties, so each tree is a graph. Is every graph a tree? Before answering that, let's define a few terms that we'll be using everywhere from now on.

|  |
| --- |
| **path**  a path is sequence of edges starting from a source vertex leading to a target vertex.  **simple path**  a path is \*simple\* if no vertex appears more than once in the path. In a tree, every path is simple.  **cycle**  a cycle is a simple path that has the same first and last vertex. A tree cannot have a cycle by definition, since there must be a single simple path between every pair of nodes.  **connected-ness**  if there is a path between every pair of vertices, then it is connected. A tree is connected by definition. |

A graph G with N vertices is a tree if any of the following are satisfied (in other words, the following are all equivalent):

1. There are N-1 edges and no cycles.
2. There are N-1 edges and G is connected.
3. Exactly one path connecting each pair of vertices.
4. G is connected, but does not remain connected if any edge is removed.

Back to [Table of contents](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#toc)

**Mathematical properties of binary trees**

1. A binary tree with N internal nodes contains N+1 external nodes.

Proof: We prove this by induction. For the case of an empty tree, there is 1 external node, so this holds for N = 0. For N > 0, a binary tree with N internal nodes has a left subtree with k internal nodes, and a right subtree with N-1-k internal nodes, for some k between 0 and N-1 (the root is an internal node). By the inductive hypothesis, the left subtree has k+1 external nodes and the right subtree has N-1-k+1 = N-k external nodes, resulting in a total of k+1+N-k=N+1 external nodes.

1. A binary tree with N internal nodes has 2N internal links/edges, N-1 to internal nodes and N+1 to external nodes.

Proof: Each internal node, except for the root, contributes one edge to its parent, which means that there are N-1 links to internal nodes. The N+1 external nodes also contribute one edge each, so there are N+1 links to external nodes, bringing the total number of links to N-1+N+1 = 2N.

1. Levels in a tree: the level of a node in a tree is one higher than the level of its parent, with the root at level 0. For a rooted tree, you can easily picture this by having horizontal lines that partition the nodes such that nodes at the same level are in the same partition. The level is also called the "depth" of a node. It's not defined for an empty tree.

For a node `n', it's level is defined recursively in terms of its parent:

|  |
| --- |
| -  | 0, if n is the root (parent(n) == null)  level(n) = |  | 1 + level(parent(n)), otherwise  - |

1. height of a tree: The height of a tree is the maximum of the levels of the tree's nodes. This however leads to an inefficient implementation, since we have to compute the levels for all the nodes, and take the maximum value.

Alternatively, the height of a tree is the maximum height of the root's subtrees plus 1 (for the link to the root itself). An empty tree has a height of -1 (which means that a tree with a single node, ie., just the root, has a height of 0).

For a tree rooted at `n', it's height is defined recursively in terms the heights of its subtrees:

|  |
| --- |
| -  | -1, if tree is empty (n == null)  height(n) = |  | 1 + max(height(n.left), height(n.right)), otherwise  - |

Note that this guarantees that the height of a tree with a single node is 0.

1. The height of a binary tree with N internal nodes is at least lg N and at most N-1.

Proof: The worst case (the maximum height) is when each node at all the levels except the last one has one internal and one external children. The leaf node at the last level has both external children. The height is obviously N-1 in this case.

The best case (the minimum height) is when each node at all but perhaps the last level has exactly two internal children; the nodes at the last level may have a mix of internal and external children. Level 0 has 1 (or 2^0) node, level 1 has 2^1 nodes, level 2 has 2^2 nodes, and so on until level h-1 which has 2^{h-1} nodes. The last level, level h, may have just one internal node or up to 2^h internal nodes. Given that the tree has N+1 external nodes, we have the following:

|  |
| --- |
| 2^h < N + 1 <= 2^{h+1}  h < lg(N+1) and h >= lg(N+1) - 1  so, h = floor(lg(N)) |

Another way to look at it is if you consider the following question: what is the shortest possible tree with N internal nodes? The maximum number of internal nodes in a binary tree of height h (without increasing its height) occurs when all the leaves are completely full, and adding another node would increase the height to h+1. That indeed is the shortest tree possible! Instead of keeping the last level full, we can decrease the number of nodes so that there is a single node at the last level, still keeping the height to be h.

|  |
| --- |
| N\_max = 2^0 + 2^1 + 2^2 + ... + 2^{h-1} + 2^h = 2^{h+1} - 1  N\_min = 2^0 + 2^1 + 2^2 + ....+ 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h  N <= N\_max  N <= 2^{h+1} - 1  N+1 <= 2^{h+1}  lg(N+1) <= h+1  h >= lg(N+1) - 1  N >= N\_min  N >= 2^h  lg(N) >= h  h <= lg(N)  \*\* Note: a quick review of sums - arithmetic, geometric, etc. |

Back to [Table of contents](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#toc)

**Tree traversal**

Traversal in trees is analogous to "iteration" in sequences, in which each node is "visited" exactly once. Unlike in a sequence, there is no "natural" order in which to proceed to the next element, but there are a few conventional way of doing it. The difference is the choice is advancing to the next node in the tree, so there are different possible "visitation" orders or linearizations.

**Depth order**: The depth order traversals recursively visit the children of a node until reaching an external nodes, and then backing up and visiting the next children, until all the children are exhausted. There are 3 common ones:

|  |
| --- |
| **pre-order** traversal : visit self, then left and then right (slr)  **in-order** traversal : visit left, then self, then right (lsr)  **post-order** traversal : visit left, then right, and then self (lrs) |

The pre-order traversal is commonly used when searching for a key within a node in a tree. The pre-order traversal is commonly known as **depth-first** when dealing with graphs.

The depth-order traversals are naturally recursive, and most easily implemented as such. However, these can also be implemented iteratively using stacks (which makes sense given the connection between recursion and stacks). The following shows a recursive implementation of pre-order traversal of a binary tree.

|  |
| --- |
| public static preOrderVisit(Node n) {  // Done if tree is empty  if (n == null)  return;  else {  // Visit self, and then visit left and right subtrees recursively  visit(n);  preOrderVisit(n.left);  preOrderVisit(n.right);  }  } |

We start traversing a tree by calling preOrderVisit(root) where root is a reference to the root node of the binary tree.

We can get the other two depth-order traversals – in-order and post-order – by simply re-arranging the orders of the method calls.

|  |
| --- |
| public static inOrderVisit(Node n) {  // Done if tree is empty  if (n == null)  return;  else {  // Visit left subtree, visit self, and then visit right subtree  inOrderVisit(n.left);  visit(n);  inOrderVisit(n.right);  }  }  public static postOrderVisit(Node n) {  // Done if tree is empty  if (n == null)  return;  else {  // Visit left subtree and right subtrees, and then visit self  postOrderVisit(n.left);  postOrderVisit(n.right);  visit(n);  }  } |

Level order: Starting from level 0 (only one node - the root), visit the nodes in level 1 (perhaps in left-to-right order), and then level 2 and so on. Can be easily implemented (iteratively) using a queue. In fact, if you replace the stack used in (iterative) pre-order traversal with a queue, you'll get level order traversal. The level order is commonly known as **breadth-first** traversal when dealing with graphs.

Back to [Table of contents](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#toc)

**Some recursive tree algorithms/methods**

Since tree is an inherently recursive data structure, the operations on a tree are often easily described by recursion as opposed to by iteration.

We've already seen some of the examples earlier (height, and levels or depth), so let's take a look a few more.

1. Counting the nodes: the number of nodes in a binary tree is the sum of the number of nodes in the two subtrees, plus one (for the root of the tree).

For a tree rooted at `n', it's size (i.e., the number of nodes) is defined recursively in terms the sizes of its subtrees:

|  |
| --- |
| -  | 0, if tree is empty (n == null)  size(n) = |  | 1 + size(n.left) + size(n.right) otherwise  -  public static int size(Node n) {  if (n == null)  return 0;  else  return 1 + size(n.left) + size(n.right);  } |

1. Copying a binary tree: To copy a tree, you can copy the root node, and then attach a copy of the left subtree to the copy's left and a copy of it's right subtree to the copy's right. Copying an empty tree returns an empty tree of course.

For a tree rooted at `n', it's copy is defined recursively in terms the copies of its subtrees:

|  |
| --- |
| -  | null, if tree is empty (n == null)  copy(n) = |  | c = make a copy of node `n'  | (i.e., create new node with `n''s element)  | c.left = copy(n.left)  | c.right = copy(n.right) otherwise  -  public static Node copy(Node n) {  if (n == null)  return null;  else {  Node c = new Node(n.element, copy(n.left), copy(n.right));  return c;  }  } |

1. Comparing if two trees are the same (structure and values): If both trees are empty, they are the same; otherwise, if the two root's have equal elements, AND if the left and right subtrees are the same, then the trees are the same.

For trees rooted at `n1' and `n2', their "same-ness" or equality is is defined recursively in terms the equality of the respective subtrees:

|  |
| --- |
| -  | true if both n1 and n2 are empty  | false if n1 is empty, but n2 is not  same(n) = |  | true if nodes n1 and n2 have equal  | elements, AND if  | same(n1.left, n2.left) AND  | same(n1.right, n2.right)  |  | false otherwise  -  public static boolean same(Node n1, Node n2) {  if (n1 == null || n2 == null) {  if (n1 == null && n2 == null)  return true;  else  return false;  } else {  return n1.element.equals(n2.element)  && same(n1.left, n2.left)  && same(n1.right, n2.right);  }  } |

1. Finding the leftmost node of a binary tree: the leftmost node is the one you get by taking a left each time starting from the root until there is more left.

|  |
| --- |
| public static Node leftmost(Node n) {  if (n.left == null)  return n;  else  return leftmost(n.left);  } |

You can get the rightmost by simply replacing "left" with "right" above.

Back to [Table of contents](http://moodle.bracu.ac.bd/mod/page/view.php?id=5310#toc)