Types of Tree

1. Binary Tree
2. Binary Search Tree
3. AVL Tree
4. B-Tree

Tree Traversal

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

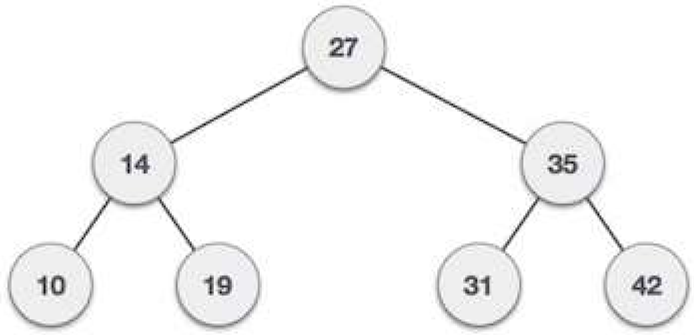
To learn more, please visit tree traversal.

Tree Applications

- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.
- **Search** − Searches an element in a tree.

- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree.

We shall learn about tree traversing methods in the coming chapter.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
If root is NULL
   then create root node
return

If root exists then
   compare the data with node.data

   while until insertion position is located

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

   endwhile
   insert data

end If
```

Implementation

The implementation of insert function should look like this −

```
void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty, create root node
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent  = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
            return;
          }
        }

        //go to right of the tree
        else {
          current = current->rightChild;

          //insert to the right
          if(current == NULL) {
            parent->rightChild = tempNode;
            return;
          }
        }
      }
    }
  }
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
  return root
else
  while data not found

    If data is greater than node.data
      goto right subtree
    else
      goto left subtree
      If data found
         return node
   endwhile

   return data not found

end if
```

The implementation of this algorithm should look like this.

```
struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
      if(current != NULL)
      printf("%d ",current->data);

      //go to left tree

      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }

      return current;
   }
```

}

To know about the implementation of binary search tree data structure, please click here

Drawing trees

Trees are often drawn in the plane. Ordered trees can be represented essentially uniquely in the plane, and are hence called *plane trees,* as follows: if one fixes a conventional order (say, counterclockwise), and arranges the child nodes in that order (first incoming parent edge, then first child edge, etc.), this yields an embedding of the tree in the plane, unique up to ambient isotopy. Conversely, such an embedding determines an ordering of the child nodes.

If one places the root at the top (parents above children, as in a family tree) and places all nodes that are a given distance from the root (in terms of number of edges: the "level" of a tree) on a given horizontal line, one obtains a standard drawing of the tree. Given a binary tree, the first child is on the left (the "left node"), and the second child is on the right (the "right node").

Common operations

Graph and tree search algorithms

- α–β
- A*
- B*

- Backtracking
- Beam
- Bellman–Ford
- Best-first
- Bidirectional
- Borůvka
- Branch & bound
- BFS
- British Museum
- D*
- DFS
- Dijkstra
- Edmonds
- Floyd–Warshall
- Fringe search
- Hill climbing
- IDA*
- Iterative deepening
- Johnson
- Jump point
- Kruskal
- Lexicographic BFS
- LPA*
- Prim
- SMA*

Listings

- *Graph algorithms*
- *Search algorithms*
- *List of graph algorithms*

Related topics

- Dynamic programming
- Graph traversal

- Enumerating all the items
- Enumerating a section of a tree
- Searching for an item
- Adding a new item at a certain position on the tree
- Deleting an item
- Pruning: Removing a whole section of a tree
- Grafting: Adding a whole section to a tree
- Finding the root for any node
- Finding the lowest common ancestor of two nodes

Traversal and search methods

Main article: Tree traversal

Stepping through the items of a tree, by means of the connections between parents and children, is called **walking the tree**, and the action is a *walk* of the tree. Often, an operation might be performed when a pointer arrives at a particular node. A walk in which each parent node is traversed before its children is called a **pre-order** walk; a walk in which the children are traversed before their respective parents are traversed is called a **post-order** walk; a walk in which a node's left subtree, then the node itself, and finally its right subtree are traversed is called an **in-order** traversal. (This last scenario, referring to exactly two subtrees, a left subtree and a right subtree, assumes specifically a binary tree.) A **level-order** walk effectively performs a breadth-first search over the entirety of a tree; nodes are traversed level by level, where the root node is visited first, followed by its direct child nodes and their siblings, followed by its grandchild nodes and their siblings, etc., until all nodes in the tree have been traversed.

Representations

There are many different ways to represent trees; common representations represent the nodes as dynamically allocated records with pointers to their children, their parents, or both, or as items in an array, with relationships between them determined by their positions in the array (e.g., binary heap).

Indeed, a binary tree can be implemented as a list of lists (a list where the values are lists): the head of a list (the value of the first term) is the left child (subtree), while the tail (the list of second and subsequent terms) is the right child (subtree). This can be modified to allow values as well, as in Lisp S-expressions, where the head (value of first term) is the value of the node, the head of the tail (value of second term) is the left child, and the tail of the tail (list of third and subsequent terms) is the right child.

In general a node in a tree will not have pointers to its parents, but this information can be included (expanding the data structure to also include a pointer to the parent) or stored separately. Alternatively, upward links can be included in the child node data, as in a threaded binary tree.

Generalizations

Digraphs

If edges (to child nodes) are thought of as references, then a tree is a special case of a digraph, and the tree data structure can be generalized to represent directed graphs by removing the constraints that a node may have at most one parent, and that no cycles are allowed. Edges are still abstractly considered as pairs of nodes, however, the terms parent and child are usually replaced by different terminology (for example, source and target). Different implementation strategies exist: a digraph can be represented by the same local data structure as a tree (node with value and list of children), assuming that "list of children" is a list of references, or globally by such structures as adjacency lists.

In graph theory, a tree is a connected acyclic graph; unless stated otherwise, in graph theory trees and graphs are assumed undirected. There is no one-to-one correspondence between such trees and trees as data structure. We can take an arbitrary undirected tree, arbitrarily pick one of its vertices as the root, make all its edges directed by making them point away from the root node – producing an arborescence – and assign an order to all the nodes. The result corresponds to a tree data structure. Picking a different root or different ordering produces a different one.

Given a node in a tree, its children define an ordered forest (the union of subtrees given by all the children, or equivalently taking the subtree given by the node itself and erasing the root). Just as subtrees are natural for recursion (as in a depth-first search), forests are natural for corecursion (as in a breadth-first search).

Via mutual recursion, a forest can be defined as a list of trees (represented by root nodes), where a node (of a tree) consists of a value and a forest (its children):

```
f: [n[1], ..., n[k]]
n: v f
```

Data type versus data structure

There is a distinction between a tree as an abstract data type and as a concrete data structure, analogous to the distinction between a list and a linked list.

As a data type, a tree has a value and children, and the children are themselves trees; the value and children of the tree are interpreted as the value of the root node and the subtrees of the children of the root node. To allow finite trees, one must either allow the list of children to be empty (in which case trees can be required to be non-empty, an "empty tree" instead being represented by a forest of zero trees), or allow trees to be empty, in which case the list of children can be of fixed size (branching factor, especially 2 or "binary"), if desired.

As a data structure, a linked tree is a group of nodes, where each node has a value and a list of references to other nodes (its children). There is also the requirement that no two "downward" references point to the same node. In practice, nodes in a tree commonly include other data as well, such as next/previous references, references to their parent nodes, or nearly anything.

Due to the use of references to trees in the linked tree data structure, trees are often discussed implicitly assuming that they are being represented by references to the root node, as this is often how they are actually implemented. For example, rather than an empty tree, one may have a null reference: a tree is always non-empty, but a reference to a tree may be null.

Recursive

Recursively, as a data type a tree is defined as a value (of some data type, possibly empty), together with a list of trees (possibly an empty list), the subtrees of its children; symbolically:

t: v [t[1], ..., t[k]]

(A tree *t* consists of a value *v* and a list of other trees.)

More elegantly, via mutual recursion, of which a tree is one of the most basic examples, a tree can be defined in terms of forest (a list of trees), where a tree consists of a value and a forest (the subtrees of its children):

f: [t[1], ..., t[k]]

t: v f

Note that this definition is in terms of values, and is appropriate in functional languages (it assumes referential transparency); different trees have no connections, as they are simply lists of values.

As a data structure, a tree is defined as a node (the root), which itself consists of a value (of some data type, possibly empty), together with a list of references to other nodes (list possibly empty, references possibly null); symbolically:

n: v [&n[1], ..., &n[k]]

(A node *n* consists of a value *v* and a list of references to other nodes.)

This data structure defines a directed graph,[b] and for it to be a tree one must add a condition on its global structure (its topology), namely that at most one reference can point to any given node (a node has at most a single parent), and no node in the tree point to the root. In fact, every node (other than the root) must have exactly one parent, and the root must have no parents.

Indeed, given a list of nodes, and for each node a list of references to its children, one cannot tell if this structure is a tree or not without analyzing its global structure and that it is in fact topologically a tree, as defined below.

Type theory

As an abstract data type, the abstract tree type *T* with values of some type *E* is defined, using the abstract forest type *F* (list of trees), by the functions:

value: $T \to E$

children: $T \to F$

nil: $() \to F$

node: $E \times F \to T$

with the axioms:

value(node(e, f)) = e

children(node(e, f)) = f

In terms of type theory, a tree is an inductive type defined by the constructors nil (empty forest) and node (tree with root node with given value and children).

Mathematical terminology

Viewed as a whole, a tree data structure is an ordered tree, generally with values attached to each node. Concretely, it is (if required to be non-empty):

- A rooted tree with the "away from root" direction (a more narrow term is an "arborescence"), meaning:
  - A directed graph,
  - whose underlying undirected graph is a tree (any two vertices are connected by exactly one simple path),
  - with a distinguished root (one vertex is designated as the root),
  - which determines the direction on the edges (arrows point away from the root; given an edge, the node that the edge points from is called the parent and the node that the edge points to is called the child),

together with:

- an ordering on the child nodes of a given node, and
- a value (of some data type) at each node.

Often trees have a fixed (more properly, bounded) branching factor (outdegree), particularly always having two child nodes (possibly empty, hence *at most* two *non-empty* child nodes), hence a "binary tree".

Allowing empty trees makes some definitions simpler, some more complicated: a rooted tree must be non-empty, hence if empty trees are allowed the above definition instead becomes "an empty tree or a rooted tree such that ...". On the other hand, empty trees simplify defining fixed branching factor: with empty trees allowed, a binary tree is a tree such that every node has exactly two children, each of which is a tree (possibly empty). The complete sets of operations on the tree must include fork operation.

Mathematical definition

This section **may contain an excessive amount of intricate detail that may interest only a particular audience**. Please help by spinning off or relocating any relevant information, and removing excessive detail that may be against Wikipedia's inclusion policy. *(August 2020) (Learn how and when to remove this template message)*

Unordered tree

Mathematically, an *unordered tree*[2] (or "algebraic tree")[3] can be defined as an algebraic structure (*X*, parent) where *X* is the non-empty carrier set of *nodes* and parent is a function on *X* which assigns each node *x* its "parent" node, parent(*x*). The structure is subject to the condition that every non-empty subalgebra must have the same fixed point. That is, there must be a unique "root" node *r*, such that parent(*r*) = *r* and for every node *x*, some iterative application parent(parent(⋯parent(*x*)⋯)) equals *r*.

There are several equivalent definitions.

As the closest alternative, one can define unordered trees as *partial* algebras (X, parent) which are obtained from the total algebras described above by letting parent(r) be undefined. That is, the root r is the only node on which the parent function is not defined and for every node x, the root is reachable from x in the directed graph (X, parent). This definition is in fact coincident with that of an anti-arborescence. The TAoCP book uses the term *oriented tree*.[4]

An *unordered tree* is a structure (X, $\prec$) where X is a set of *nodes* and $\prec$ is a *child-to-parent* relation between nodes such that:

(1) X is non-empty.

(2) X is weakly connected in $\prec$.

(3) $\prec$ is functional.

(4) $\prec$ satisfies ACC: there is no infinite sequence $x_1 \prec x_2 \prec x_3 \prec \cdots$.

The box on the right describes the partial algebra (X, parent) as a relational structure (X, $\prec$). If (1) is replaced by

1. X contains exactly one $\prec$-maximal node.

then condition (2) becomes redundant.

Using this definition, dedicated terminology can be provided for generalizations of unordered trees that correspond to distinguished subsets of the listed conditions:

- (1,2,3) – directed pseudotree,
- (3) – directed pseudoforest,
- (3,4) – unordered forest (whose components are unordered trees),
- (4) – directed acyclic graph, assumed that X is finite,
- (1',4) – acyclic accessible pointed graph (then condition (2) holds implicitly).

Another equivalent definition of an unordered tree is that of a set-theoretic tree that is singly-rooted and whose height is at most ω (a "finite-ish" tree).[5] That is, the algebraic structures (X, parent) are equivalent to partial orders (X, ≤) that have a top element r and whose every principal upset (aka principal filter) is a finite chain. To be precise, we should speak about an inverse set-theoretic tree since the set-theoretic definition usually employs opposite ordering.

The correspondence between (X, parent) and (X, ≤) is established via reflexive transitive closure / reduction, with the reduction resulting in the "partial" version without the root cycle.

The definition of trees in descriptive set theory (DST) utilizes the representation of partial orders (X, ≥) as prefix orders between finite sequences. In turns out that up to isomorphism, there is a one-to-one correspondence between the (inverse of) DST trees and the tree structures defined so far.

We can refer to the four equivalent characterizations as to *tree as an algebra*, *tree as a partial algebra*, *tree as a partial order*, and *tree as a prefix order*. There is also a fifth equivalent definition – that of a graph-theoretic rooted tree which is just a connected acyclic rooted graph.

The expression of trees as (partial) algebras (also called functional graphs) (*X*, parent) follows directly the implementation of tree structures using *parent pointers*. Typically, the partial version is used in which the root node has no parent defined. However, in some implementations or models even the parent(*r*) = *r* circularity is established. Notable examples:

- The Linux VFS where "The root dentry has a d_parent that points to itself".[6].
- The concept of an *instantiation tree*[7][8][9]

from object-oriented programming. In this case, the root node is the top metaclass – the only class that is a direct instance of itself.

Note that the above definition admits *infinite* trees. This allows for the description of infinite structures supported by some implementations via lazy evaluation. A notable example is the infinite regress of eigenclasses from the Ruby object model.[10] In this model, the tree established via superclass links between non-terminal objects is infinite and has an infinite branch (a single infinite branch of "helix" objects – see the diagram).

Sibling sets

In every unordered tree (*X*, parent) there is a distinguished partition of the set *X* of nodes into *sibling sets*. Two non-root nodes *x*, *y* belong to the same sibling set if parent(*x*) = parent(*y*). The root node *r* forms the singleton sibling set {*r*}.[c] A tree is said to be *locally finite* or *finitely branching* if each of its sibling sets is finite.

Each pair of distinct siblings is incomparable in ≤. This is why the word *unordered* is used in the definition. Such a terminology might become misleading when all sibling sets are singletons, i.e. when the set *X* of all nodes is totally ordered (and thus well-ordered) by

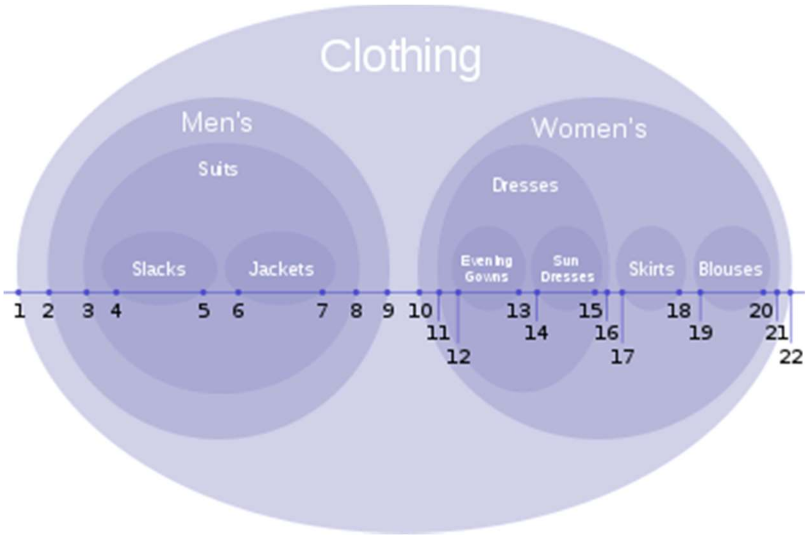
≤ In such a case we might speak about a *singly-branching tree* instead.

Using set inclusion

As with every partially ordered set, tree structures $(X, \leq)$ can be represented by inclusion order – by set systems in which ≤ is coincident with ⊆, the induced inclusion order. Consider a structure $(U, \mathcal{F})$ such that U is a non-empty set, and $\mathcal{F}$ is a set of subsets of U such that the following are satisfied (by Nested Set Collection definition):

1. $\emptyset \notin \mathcal{F}$. (That is, $(U, \mathcal{F})$ is a hypergraph.)
2. $U \in \mathcal{F}$.
3. For every X, Y from $\mathcal{F}$, $X \cap Y \in \{\emptyset, X, Y\}$. (That is, $\mathcal{F}$ is a *laminar* family.)[11]
4. For every X from $\mathcal{F}$, there are only finitely many Y from $\mathcal{F}$ such that $X \subseteq Y$.

Then the structure $(\mathcal{F}, \subseteq)$ is an unordered tree whose root equals U. Conversely, if $(U, \leq)$ is an unordered tree, and $\mathcal{F}$ is the set $\{\downarrow x \mid x \in U\}$ of all principal ideals of $(U, \leq)$ then the set system $(U, \mathcal{F})$ satisfies the above properties.



Tree as a laminar system of sets (Copied from Nested set model)

The set-system view of tree structures provides the default semantic model – in the majority of most popular cases, tree data structures represent containment hierarchy. This also offers a justification for order direction with the root at the top: The root node is a *greater* container than any other node. Notable examples:

- Directory structure of a file system. A directory contains its sub-directories.
- DOM tree. The document parts correspondent to DOM nodes are in subpart relation according to the tree order.
- Single inheritance in object-oriented programming. An instance of a class is also an instance of a superclass.
- Hierarchical taxonomy such as the Dewey Decimal Classification with sections of increasing specificity.
- BSP trees, quadtrees, octrees, R-trees and other tree data structures used for recursive space partitioning.

Well-founded trees

An unordered tree $(X, \leq)$ is *well-founded* if the strict partial order $<$ is a well-founded relation. In particular, every finite tree is well-founded. Assuming the axiom of dependent choice a tree is well-founded if and only if it has no infinite branch.

Well-founded trees can be defined recursively – by forming trees from a disjoint union of smaller trees. For the precise definition, suppose that X is a set of nodes. Using the reflexivity of partial orders, we can identify any tree $(Y, \leq)$ on a subset of X with its partial order $(\leq)$ – a subset of $X \times X$. The set $\mathcal{R}$ of all relations R that form a well-founded tree (Y, R) on a subset Y of X is defined in stages $\mathcal{R}_\mathrm{i}$, so that $\mathcal{R} = \bigcup\{\mathcal{R}_i \mid i \text{ is ordinal}\}$. For each ordinal number i, let R belong to the i-th stage $\mathcal{R}_i$ if and only if R is equal to

$$\bigcup\mathcal{F} \cup ((\mathrm{dom}(\bigcup\mathcal{F}) \cup \{x\}) \times \{x\})$$

where $\mathcal{F}$ is a subset of $\bigcup\{\mathcal{R}_k \mid k < i\}$ such that elements of $\mathcal{F}$ are pairwise disjoint, and x is a node that does not belong to dom($\bigcup\mathcal{F}$). (We use dom(S) to denote the domain of a relation S.) Observe that the lowest stage $\mathcal{R}_0$ consists of single-node trees $\{(x,x)\}$ since only empty $\mathcal{F}$ is possible. In each stage, (possibly) new trees R are built by taking a forest $\bigcup\mathcal{F}$ with components $\mathcal{F}$ from lower stages and attaching a new root x atop of $\bigcup\mathcal{F}$.

In contrast to the tree *height* which is at most ω, the *rank* of well-founded trees is unlimited,[12] see the properties of "unfolding".

Using recursive pairs

In computing, a common way to define well-founded trees is via recursive ordered pairs (F, x): a tree is a forest F together with a "fresh" node x.[13] A *forest* F in turn is a possibly empty set of trees with pairwise disjoint sets of nodes. For the precise definition, proceed similarly as in the construction of *names* used in the set-theoretic technique of forcing. Let X be a set of nodes. In the superstructure over X, define sets T, $\mathcal{F}$ of trees and forests, respectively, and a map nodes : $T \rightarrow \wp(X)$ assigning each tree t its underlying set of nodes so that:

(trees over X)	$t \in T$	↔	t is a pair (F, x) from $\mathcal{F} \times X$ such that for all $s \in F$, $x \notin$ nodes(s),
(forests over X)	$F \in \mathcal{F}$	↔	F is a subset of T such that for every $s,t \in F$, $s \neq t$, nodes(s) ∩ nodes(t) = ∅ }},
(nodes of trees)	$y \in$ nodes(t)	↔	$t = (F, x) \in T$ and either $y = x$ or $y \in$ nodes(s) for some $s \in F$ }}.

Circularities in the above conditions can be eliminated by stratifying each of T, $\mathcal{F}$ and *nodes* into stages like in the previous subsection. Subsequently, define a "subtree" relation $\leq$ on T as the reflexive transitive closure of the "immediate subtree" relation $\prec$ defined between trees by

$$s \prec t \leftrightarrow \qquad s \in \pi_1(t)$$

where $\pi_1(t)$ is the projection of t onto the first coordinate; i.e., it is the forest F such that $t = (F, x)$ for some $x \in X$. It can be observed that $(T, \leq)$ is a multitree: for every $t \in T$, the principal ideal $\downarrow t$ ordered by $\leq$ is a well-founded tree as a partial order. Moreover, for every tree $t \in T$, its "nodes"-order structure (nodes(t), $\leq_t$) is given by $x \leq_t y$ if and only if there are forests $F, G \in \mathcal{F}$ such that both (F, x) and (G, y) are subtrees of t and $(F, x) \leq (G, y)$.

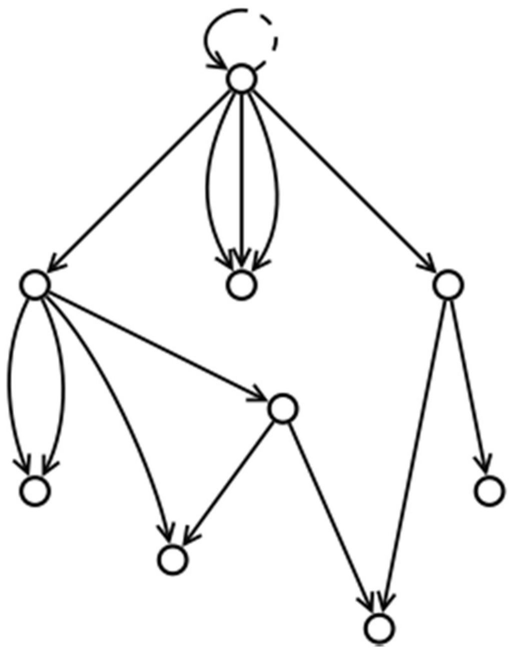
Using arrows

Another formalization as well as generalization of unordered trees can be obtained by reifying parent-child pairs of nodes. Each such ordered pair can be regarded as an abstract entity – an "arrow". This results in a multidigraph (X, A, s, t) where X is the set of nodes, A is the set of *arrows*, and s and t are functions from A to X assigning each arrow its *source* and *target*, respectively. The structure is subject to the following conditions:

1. $(A, s \circ t^{-1})$ is an unordered tree, as a total algebra.
2. The t map is a bijection between arrows and nodes.

In (1), the composition symbol $\circ$ is to be interpreted left-to-right. The condition says that inverse consecutivity of arrows[d] is a total child-to-parent map. Let this parent map between arrows be denoted p, i.e. $p = s \circ t^{-1}$. Then we also have $s = p \circ t$, thus a multidigraph satisfying (1,2) can also be axiomatized as (X, A, p, t), with the parent map p instead of s as a definitory constituent.

Observe that the root arrow is necessarily a loop, i.e. its source and target coincide.



Arrow tree: the hard-link structure of VFS

An important generalization of the above structure is established by allowing the target map *t* to be many-to-one. This means that (2) is weakened to

> (2') The *t* map is surjective – each node is the target of some arrow.

Note that condition (1) asserts that only leaf arrows are allowed to have the same target. That is, the restriction of *t* to the range of *p* is still injective.

Multidigraphs satisfying (1,2') can be called "arrow trees" – their tree characteristics is imposed on arrows rather than nodes. These structures can be regarded as the most essential abstraction of the Linux VFS because they reflect the hard-link structure of filesystems. Nodes are called inodes, arrows are dentries (or hard links). The parent and target maps *p* and *t* are respectively represented by d_parent and d_inode fields in the dentry data structure.[14] Each inode is assigned a fixed file type, of which the directory type plays a special role of "designed parents":

a. only directory inodes can appear as hard-link source and
b. a directory inode cannot appear as the target of more than one hard-link.

Using dashed style for the first half of the root loop indicates that, similarly to the parent map, there is a *partial* version for the source map s in which the source of the root arrow is undefined. This variant is employed for further generalization, see #Using paths in a multidigraph.

Using paths in a digraph

Unordered trees naturally arise by "unfolding" of accessible pointed graphs.[15]

Let $\mathcal{R} = (X, R, r)$ be a *pointed relational structure*, i.e. such that X is the set of nodes, R is a relation between nodes (a subset of $X \times X$), and r is a distinguished "root" node. Assume further that $\mathcal{R}$ is *accessible*, which means that X equals the preimage of $\{r\}$ under the reflexive transitive closure of R, and call such a structure an *accessible pointed graph* or *apg* for short.($\ast$) Then one can derive another apg $\mathcal{R}' = (X', R', r')$ – the *unfolding* of $\mathcal{R}$ – as follows:

- X' is the set of reversed *paths* to r, i.e. the set of non-empty finite sequences p of nodes (elements of X) such that (a) consecutive members of p are inversely R-related, and (b) the first member of p is the root r,
- R' is a relation between paths from X' such that paths p and q are R'-related if and only if $p = q \ast [x]$ for some node x (i.e. q is a maximal proper prefix of p, the "popped" p), and
- r' is the one-element sequence $[r]$.

Apparently, the structure (X', R') is an unordered tree in the "partial-algebra" version: R' is a partial map that relates each non-root element of X' to its parent by path popping. The root element is obviously r'. Moreover, the following properties are satisfied:

- $\mathcal{R}$ is isomorphic to its unfolding $\mathcal{R}'$ if and only if $\mathcal{R}$ is a tree (⁑). (In particular, unfolding is idempotent, up to isomorphism.)
- Unfolding preserves well-foundedness: If R is well-founded then so is R'.
- Unfolding preserves rank: If R is well-founded then the ranks of R and R' coincide.

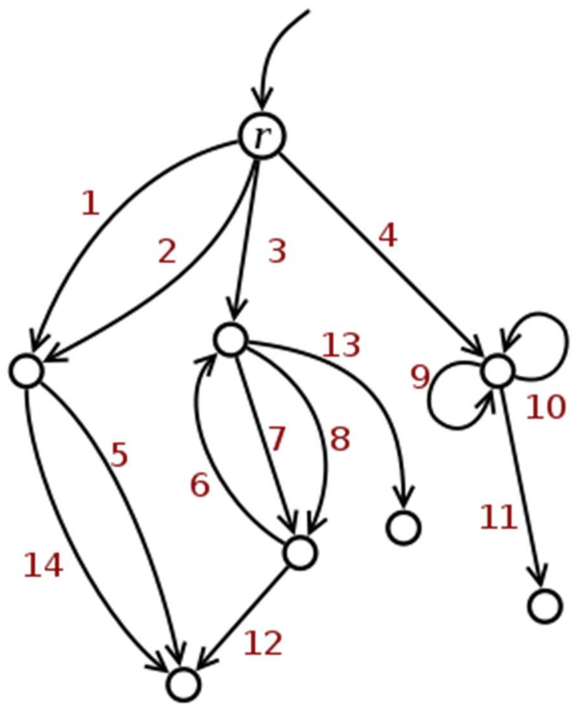
Notes:

($\ast$) To establish a concordancy between R and the *parent* map, the presented definition uses *reversed* accessibility: r is reachable from any node. In the original definition by P. Aczel[15], every node is reachable from r (thus, instead of "preimage", the word "image" applies).[e]

(⁑) We have implicitly introduced a definition of unordered trees as apgs: call an apg $\mathcal{R}$ = (X, R, r) a *tree* if the reduct (X, R) is an unordered tree as a partial algebra. This can be translated as: Every node is accessible by exactly one path.

Using paths in a multidigraph

As shown on the example of hard-link structure of file systems, many data structures in computing allow *multiple* links between nodes. Therefore, in order to properly exhibit the appearance of unordered trees among data structures it is necessary to generalize accessible pointed graphs to *multidigraph* setting. To simplify the terminology, we make use of the term *quiver* which is an established synonym for "multidigraph".



Accessible pointed quiver (apq): generalization of apg to multidigraphs.

Let an *accessible pointed quiver* or *apq* for short be defined as a structure

$\mathcal{M} = (X, A, s, t)$

where

X is a set of *nodes*,

A is a set of *arrows*,

s is a *partial* function from A to X (the *source* map), and

t is a total function from A to X (the *target* map).

Thus, $\mathcal{M}$ is a "partial multidigraph".

The structure is subject to the following conditions:

1. There is exactly one "root" arrow, a_r, whose source $s(a_r)$ is undefined.
2. Every node $x \in X$ is reachable via a finite sequence of consecutive arrows starting with the root arrow a_r.

$\mathcal{M}$ is said to be a *tree* if the target map t is a bijection between arrows and nodes. The *unfolding* of $\mathcal{M}$ is formed by the sequences mentioned in (2) – which are the *accessibility paths* (cf. Path algebra). As an apq, the unfolding can be written as

$$\mathcal{M}' = (X', A', s', t')$$

where

X' is the set of accessibility paths,

A' coincides with X',

s' coincides with path popping, and

t' is the identity on X'.

Like with apgs, unfolding is idempotent and always results in a tree.

The *underlying apg* is obtained as the structure

$$(X, R, t(a_r))$$

where

$$R = \{(t(a),s(a)) \mid a \in A \setminus \{a_r\}\}.$$

The diagram above shows an example of an apq with 1 + 14 arrows. In JavaScript, Python or Ruby, the structure can be created by the following (exactly the same) code:

```
r = {};
r[1] = {}; r[2] = r[1]; r[3] = {}; r[4] = {};
```

```
r[1][5]    = {};   r[1][14]    = r[1][5];
r[3][7]    = {};   r[3][8]     = r[3][7];  r[3][13] = {};
r[4][9]    = r[4]; r[4][10]    = r[4];     r[4][11] = {};
r[3][7][6] = r[3]; r[3][7][12] = r[1][5];
```

Using names

Unordered trees and their generalizations form the essence of naming systems. There are two prominent examples of naming systems: file systems and (nested) associative arrays. The multidigraph-based structures from previous subsections provided *anonymous* abstractions for both cases. To obtain naming capabilities, arrows are to be equipped with *names* as identifiers. A name must be locally unique – within each sibling set of arrows[f] there can be at most one arrow labelled by a given name.

source	**name**	**target**
$s(a)$	$\sigma(a)$	$t(a)$

This can be formalized as a structure

$$\mathcal{E} = (X, \Sigma, A, s, \sigma, t)$$

where

X is a set of *nodes*,

Σ is a set of *names*,

A is a set of *arrows*,

s is a partial function from A to X,

σ is a partial function from A to Σ, and

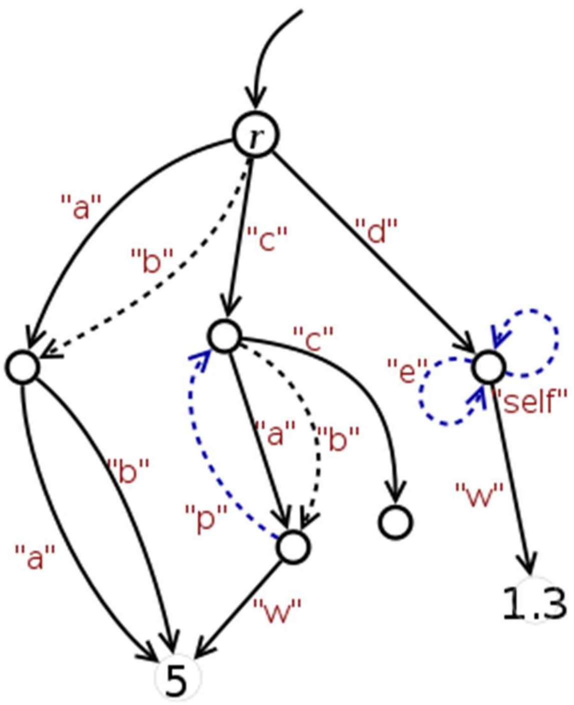
t is a total function from A to X.

For an arrow a, constituents of the triple $(s(a), \sigma(a), t(a))$ are respectively a's *source*, *name* and *target*. The structure is subject to the following conditions:

1. The reduct (X, A, s, t) is an accessible pointed quiver (apq) as defined previously.
2. The name function σ is undefined just for the source-less root arrow.
3. The name function σ is injective in the restriction to every sibling set of arrows, i.e. for every non-root arrows a, b, if $s(a) = s(b)$ and $\sigma(a) = \sigma(b)$ then $a = b$.

This structure can be called a *nested dictionary* or *named apq*. In computing, such structures are ubiquitous. The table above shows that arrows can be considered "un-reified" as the set $A' = \{(s(a), \sigma(a), t(a)) \mid a \in A \setminus \{a_r\}\}$ of source-name-target triples. This leads to a relational structure (X, Σ, A') which can be viewed as a relational database table. Underlines in source and name indicate primary key.

The structure can be rephrased as a deterministic labelled transition system: X is a set of "states", Σ is a set of "labels", A' is a set of "labelled transitions". (Moreover, the root node $r = t(a_r)$ is an "initial state", and the accessibility condition means that every state is reachable from the initial state.)



Nested dictionary

The diagram on the right shows a nested dictionary $\mathcal{E}$ that has the same underlying multidigraph as the example in the previous subsection. The structure can be created by the code below. Like before, exactly the same code applies for JavaScript, Python and Ruby.

First, a substructure, $\mathcal{E}_0$, is created by a single assignment of a literal {...} to r. This structure, depicted by full lines, is an "arrow tree" (therefore, it is a spanning tree). The literal in turn appears to be a JSON serialization of $\mathcal{E}_0$.

Subsequently, the remaining arrows are created by assignments of already existing nodes. Arrows that cause cycles are displayed in blue.

```
r = {"a":{"a":5,"b":5},"c":{"a":{"w":5},"c":{}},"d":{"w":1.3}}

r["b"]           = r["a"]; r["c"]["b"] = r["c"]["a"]
r["c"]["a"]["p"] = r["c"]; r["d"]["e"] = r["d"]["self"] = r["d"]
```

In the Linux VFS, the name function σ is represented by the d_name field in the dentry data structure.[14] The $\mathcal{E}_0$ structure above demonstrates a correspondence between JSON-representable structures and hard-link structures of file systems. In both cases, there is a fixed set of built-in types of "nodes" of which one type is a container type, except that in JSON, there are in fact two such types – Object and Array. If the latter one is ignored (as well as the distinction between individual primitive data types) then the provided abstractions of file-systems and JSON data are the same – both are arrow trees equipped with naming σ and a distinction of container nodes.

Pathnames

The naming function σ of a nested dictionary $\mathcal{E}$ naturally extends from arrows to arrow paths. Each sequence $p = [a_1, ..., a_n]$ of consecutive arrows is implicitly assigned a *pathname* (cf. Pathname) – the sequence $\sigma(p) = [\sigma(a_1), ..., \sigma(a_n)]$ of arrow names.[g] Local uniqueness carries over to arrow paths: different sibling paths have different pathnames. In particular, the root-originating arrow paths are in one-to-one correspondence with their pathnames. This correspondence provides a "symbolic" representation of the unfolding of $\mathcal{E}$ via pathnames – the nodes in $\mathcal{E}$ are globally identified via a tree of pathnames.

Ordered tree

The structures introduced in the previous subsection form just the core "hierarchical" part of tree data structures that appear in computing. In most cases, there is also an additional "horizontal" ordering between siblings. In search trees the order is commonly established by the "key" or value associated with each sibling, but in many trees that is not the case. For example, XML documents, lists within JSON files, and many other structures have order that does not depend on the values in the nodes, but is itself data —

sorting the paragraphs of a novel alphabetically would lose information.[dubious – discuss]

The correspondent expansion of the previously described tree structures $(X, \leq)$ can be defined by endowing each sibling set with a linear order as follows.[17][18]

An alternative definition according to Kuboyama[2] is presented in the next subsection.

An *ordered tree* is a structure $(X, \leq_V, \leq_S)$ where X is a non-empty set of nodes and $\leq_V$ and $\leq_S$ are relations on X called *vertical* (or also *hierarchical*[2]) order and *sibling* order, respectively. The structure is subject to the following conditions:

1. $(X, \leq_V)$ is a partial order that is an unordered tree as defined in the previous subsection.
2. $(X, \leq_S)$ is a partial order.
3. Distinct nodes are comparable in $<_S$ if and only if they are siblings:

   $(<_S) \cup (>_S) = ((\prec_V) \circ (\succ_V)) \setminus \mathrm{id}_X.$

4. Every node has only finitely many preceding siblings, i.e. every principal ideal of $(X, \leq_S)$ is finite. (This condition can be omitted in the case of finite trees.)

Conditions (2) and (3) say that $(X, \leq_S)$ is a component-wise linear order, each component being a sibling set. Condition (4) asserts that if a sibling set S is infinite then $(S, \leq_S)$ is isomorphic to $(\mathbb{N}, \leq)$, the usual ordering of natural numbers.

Given this, there are three (another) distinguished partial orders which are uniquely given by the following prescriptions:

$(<_H) = (\leq_V) \circ (<_S) \circ (\geq_V)$ (the *horizontal order*),

$(<_{L^-}) = (>_V) \cup (<_H)$ (the *"discordant" linear order*),

$(<_{L^+}) = (<_V) \cup (<_H)$ (the *"concordant" linear order*).

This amounts to a "V-S-H-L$^{\pm}$" system of five partial orders $\leq_V$, $\leq_S$, $\leq_H$, $\leq_{L^+}$, $\leq_{L^-}$ on the same set X of nodes, in which, except for the pair $\{\leq_S, \leq_H\}$, any two relations uniquely determine the other three, see the determinacy table.

Notes about notational conventions:

- The relation composition symbol $\circ$ used in this subsection is to be interpreted left-to-right (as).
- Symbols $<$ and $\leq$ express the *strict* and *non-strict* versions of a partial order.
- Symbols $>$ and $\geq$ express the converse relations.
- The $\prec$ symbol is used for the covering relation of $\leq$ which is the *immediate* version of a partial order.

This yields six versions $\prec$, $<$, $\leq$, $\succ$, $>$, $\geq$ for a single partial order relation. Except for $\prec$ and $\succ$, each version uniquely determines the others. Passing from $\prec$ to $<$requires that $<$ be transitively reducible. This is always satisfied for all of $<_V$, $<_S$ and $<_H$ but might not hold for $<_{L^+}$ or $<_{L^-}$ if X is infinite.

The partial orders $\leq_V$ and $\leq_H$are complementary:

$$(<_V) \uplus (>_V) \uplus (<_H) \uplus (>_H) = X \times X \setminus \mathrm{id}_X.$$

As a consequence, the "concordant" linear order $<_{L^+}$ is a linear extension of $<_V$. Similarly, $<_{L^-}$ is a linear extension of $>_V$.

The covering relations $\prec_{L^-}$ and $\prec_{L^+}$ correspond to pre-order traversal and post-order traversal, respectively. If $x \prec_{L^-} y$ then, according to whether *y* has a previous sibling or not, the *x* node is either the "rightmost" non-strict descendant of the previous sibling of *y* or, in the latter case, *x* is the first child of *y*. Pairs of the latter case form the relation $(\prec_{L^-}) \setminus (<_H)$ which is a partial map that assigns each non-leaf node its *first child* node. Similarly, $(\succ_{L^+}) \setminus (>_H)$ assigns each non-leaf node with finitely many children its *last* child node.

Definition using horizontal order

The Kuboyama's definition of "rooted ordered trees"[2] makes use of the horizontal order $\leq_H$ as a definitory relation.[h] (See also Suppes.[19])

Using the notation and terminology introduced so far, the definition can be expressed as follows.

An *ordered tree* is a structure $(X, \leq_V, \leq_H)$ such that conditions (1–5) are satisfied:

1. $(X, \leq_V)$ is a partial order that is an unordered tree. (The *vertical* order.)
2. $(X, \leq_H)$ is a partial order. (The *horizontal* order.)
3. The partial orders $\leq_V$ and $\leq_H$ are complementary: $(<_V) \uplus (>_V) \uplus (<_H) \uplus (>_H) = X \times X \setminus \mathrm{id}_X$.

   (That is, pairs of nodes that are incomparable in $(<_V)$ are comparable in $(<_H)$ and vice versa.)

4. The partial orders $\leq_V$ and $\leq_H$ are "consistent": $(<_H) = (\leq_V) \circ (<_H) \circ (\geq_V)$.

   (That is, for every nodes *x*, *y* such that $x <_H y$, all descendants of *x* must precede all the descendants of *y*.)

5. Every node has only finitely many preceding siblings. (That is, for every infinite sibling set S, $(S, \leq_H)$ has the order type of the natural numbers.) (Like before, this condition can be omitted in the case of finite trees.)

The sibling order ($\leq_S$) is obtained by $(<_S) = (<_H) \cap ((\prec_V) \circ (\succ_V))$, i.e. two distinct nodes are in sibling order if and only if they are in horizontal order and are siblings.

Determinacy table

The following table shows the determinacy of the "V-S-H-L$^{\pm}$" system. Relational expressions in the table's body are equal to one of $<_V$, $<_S$, $<_H$, $<_{L^-}$, or $<_{L^+}$ according to the column. It follows that except for the pair { $\leq_S$, $\leq_H$ }, an ordered tree (X, ...) is uniquely determined by any two of the five relations.

	$<_V$	$<_S$	$<_H$	$<_{L^-}$	$<_{L^+}$
V,S			$(\leq_V) \circ (<_S) \circ (\geq_V)$		
V,H		$(<_H) \cap ((\prec_V)\circ(\succ_V))$		$(>_V) \cup (<_H)$	$(<_V) \cup (<_H)$
V,L⁻		$(<_{L^-}) \cap ((\prec_V)\circ(\succ_V))$	$(<_{L^-}) \setminus (>_V)$		
V,L⁺		$(<_{L^+}) \cap ((\prec_V)\circ(\succ_V))$	$(<_{L^+}) \setminus (<_V)$		
H,L⁻	$(>_{L^-}) \setminus (<_H)$				
H,L⁺	$(<_{L^+}) \setminus (<_H)$				

L⁻,L⁺	$(>_{L^-}) \cap (<_{L^+})$	$(<_{L^-}) \cap (<_{L^+})$
S,L⁻	$x \prec_V y \leftrightarrow y = \inf_{L^-}(Y)$ where Y is the image of $\{x\}$ under $(\geq_S)\circ(\succ_{L^-})$	
S,L⁺	$x \prec_V y \leftrightarrow y = \sup_{L^+}(Y)$ where Y is the image of $\{x\}$ under $(\leq_S)\circ(\prec_{L^+})$	

In the last two rows, $\inf_{L^-}(Y)$ denotes the infimum of Y in $(X, \leq_{L^-})$, and $\sup_{L^+}(Y)$ denotes the supremum of Y in $(X, \leq_{L^+})$. In both rows, $(\leq_S)$ resp. $(\geq_S)$ can be equivalently replaced by the sibling equivalence $(\leq_S)\circ(\geq_S)$. In particular, the partition into sibling sets together with either of $\leq_{L^-}$ or $\leq_{L^+}$ is also sufficient to determine the ordered tree. The first prescription for $\prec_V$ can be read as: the parent of a non-root node x equals the infimum of the set of all immediate predecessors of siblings of x, where the words "infimum" and "predecessors" are meant with regard to $\leq_{L^-}$. Similarly with the second prescription, just use "supremum", "successors" and $\leq_{L^+}$.

The relations $\leq_S$ and $\leq_H$ obviously cannot form a definitory pair. For the simplest example, consider an ordered tree with exactly two nodes – then one cannot tell which of them is the root.

XPath axes

XPath axis	Relation
ancestor	$<_V$
ancestor-or-self	$\leq_V$
child	$\succ_V$
descendant	$>_V$

descendant-or-self	$\geq_V$
following	$<_H$
following-sibling	$<_S$
parent	$\prec_V$
preceding	$>_H$
preceding-sibling	$>_S$
self	id_X

The table on the right shows a correspondence of introduced relations to XPath axes, which are used in structured document systems to access nodes that bear particular ordering relationships to a starting "context" node. For a context node[20] x, its *axis* named by the specifier in the left column is the set of nodes that equals the image of $\{x\}$ under the correspondent relation. As of XPath 2.0, the nodes are "returned" in *document order*, which is the "discordant" linear order $\leq_{L^-}$. A "concordance" would be achieved, if the vertical order $\leq_V$ was defined oppositely, with the bottom-up direction outwards the root like in set theory in accordance to natural trees.[i]

Traversal maps

Below is the list of partial maps that are typically used for ordered tree traversal.[21] Each map is a distinguished functional subrelation of $\leq_{L^-}$ or of its opposite.

- $\prec_V$ ... the *parent-node* partial map,
- $\succ_S$ ... the *previous-sibling* partial map,
- $\prec_S$ ... the *next-sibling* partial map,
- $(\prec_{L^-}) \setminus (<_H)$ ... the *first-child* partial map,
- $(\succ_{L^+}) \setminus (>_H)$ ... the *last-child* partial map,
- $\succ_{L^-}$ ... the *previous-node* partial map,

- $\prec_{L^-}$ ... the *next-node* partial map.

Generating structure

The traversal maps constitute a partial unary algebra[22] (X, parent, previousSibling, ..., nextNode) that forms a basis for representing trees as linked data structures. At least conceptually,there are parent links, sibling adjacency links, and first / last child links. This also applies to unordered trees in general, which can be observed on the dentry data structure in the Linux VFS.[23]

Similarly to the "V-S-H-L$^\pm$" system of partial orders, there are pairs of traversal maps that uniquely determine the whole ordered tree structure. Naturally, one such generating structure is (X, $_V$, $_S$) which can be transcribed as (X, parent, nextSibling) – the structure of parent and next-sibling links. Another important generating structure is (X, firstChild, nextSibling) known as left-child right-sibling binary tree. This partial algebra establishes a one-to-one correspondence between binary trees and ordered trees.

Definition using binary trees

The correspondence to binary trees provides a concise definition of ordered trees as partial algebras.

An *ordered tree* is a structure where X is a non-empty set of nodes, and *lc*, *rs* are partial maps on X called *left-child* and *right-sibling*, respectively. The structure is subject to the following conditions:

1. The partial maps *lc* and *rs* are disjoint, i.e. $(lc) \cap (rs) = \emptyset$.
2. The inverse of $(lc) \cup (rs)$ is a partial map p such that the partial algebra (X, p) is an unordered tree.

The partial order structure (X, $\leq_V$, $\leq_S$) is obtained as follows:

$(\prec_S) = (rs)$,

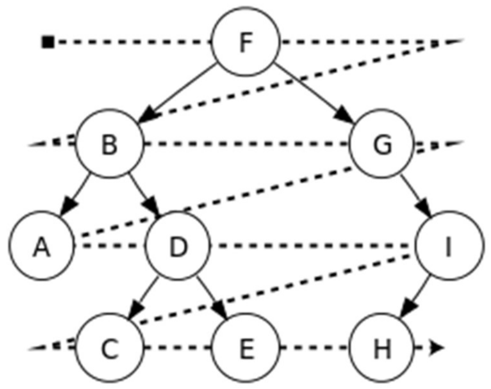
$(\succ_V) = (lc) \circ (\leq_S)$.

Encoding by sequences

Ordered trees can be naturally encoded by finite sequences of natural numbers.[24][i] Denote ω^* the set of all finite sequences of natural numbers. Then any non-empty subset W of ω^* that is closed under taking prefixes gives rise to an ordered tree: take the prefix order for $\geq_V$ and the lexicographical order for $\leq_{L^-}$. Conversely, for an ordered tree $T = (X, _V, \leq_{L^-})$ assign each node x the sequence

of sibling indices, i.e. the root is assigned the empty sequence and for every non-root node x, let $w(x) = w(\text{parent}(x)) * [i]$ where i is the number of preceding siblings of x and $*$ is the concatenation operator. Put $W = \{w(x) \mid x \in X\}$. Then W, equipped with the induced orders $\leq_V$ (the inverse of prefix order) and $\leq_{L^-}$ (the lexicographical order), is isomorphic to T.

Per-level ordering



Dashed line indicates the $<_{B^-}$ ordering)

As a possible expansion of the "V-S-H-L$^\pm$" system, another distinguished relations between nodes can be defined, based on the tree's level structure. First, let us denote by $\sim_E$ the equivalence relation defined by $x \sim_E y$ if and only if x and y have the same number of ancestors. This yields a partition of the set of nodes into *levels* L_0, L_1, ... (, L_n) – a coarsement of the partition into sibling sets. Then define relations $<_E$, $<_{B^-}$ and $<_{B^+}$ by

It can be observed that $<_E$ is a strict partial order and $<_{B^-}$ and $<_{B^+}$ are strict total orders. Moreover, there is a similarity between the "V-S-L$^\pm$" and "V-E-B$^\pm$" systems: $<_E$ is component-wise linear and orthogonal to $<_V$, $<_{B^-}$ is linear extension of $<_E$ and of $>_V$, and $<_{B^+}$ is a linear extension of $<_E$ and of $<_V$.

See also

- Tree structure
- Tree (graph theory)
- Tree (set theory)
- Cardinal tree and Ordinal tree
- Hierarchy (mathematics)
- Dialog tree
- Single inheritance
- Generative grammar
- Hierarchical clustering
- Binary space partition tree
- Recursion
- Fenwick tree

Other trees

- Trie
- Day–Stout–Warren algorithm
- Enfilade
- Left child-right sibling binary tree
- Hierarchical temporal memory

Notes

1.

□ This is different from the formal definition of subtree used in graph theory, which is a subgraph that forms a tree – it need not include all descendants. For example, the root node by itself is a subtree in the graph theory sense, but not in the data structure sense (unless there are no descendants).

□ □ Properly, a rooted, ordered directed graph.

□ □ Alternatively, a "partial" version can be employed by excluding .

□ □ Arrows *a* and *b* are said to be *consecutive*, respectively, if $t(a) = s(b)$.

□ □ However, some authors also introduce the definition with reversed reachability.[16]

□ □ I.e. arrows that have the same source node.

□ □ Here we assume that the root arrow a_r is not in *p*.

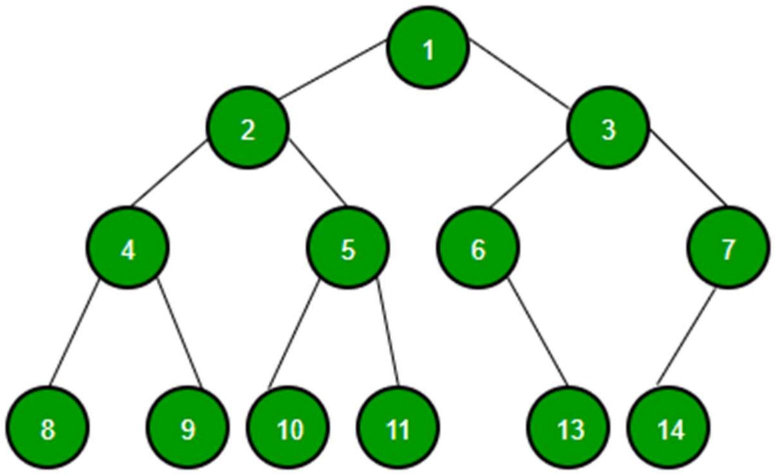
□ □ Unfortunately, the author uses the term *sibling order* for the horizontal order relation. This is non-standard, if not a misnomer.

□ □ This would also establish a concordance of the two possible directions of inequality symbols with the categorization of XPath axes into *forward axes* and *reverse axes*.

□ In general, any alphabet equipped with ordering that is isomorphic to that of natural numbers can be used.

Binary Tree Data Structure

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child