A&DS

Author

BOZENA LEKVAROVA

Created by WikiCourseBuilder

1. Data structure

1.1.

A data structure known as a hash table.
In computer science, a data structure is a data organization, management and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.TEMPLATE[Cite_book, url http://dl.acm.org/citation.cfm?id 1074100.1074312, title Encyclopedia of Computer Science, last Wegner, first Peter, last2 Reilly, first2 Edwin D., publisher John Wiley and Sons, isbn 978-0470864128, location Chichester, UK, pages 507–512, date 2003-08-29]

1.2. Usage

Data structures serve as the basis for abstract data types (ADT). The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

Different types of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.

Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory.

1.3. Implementation

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by a pointer—a bit string, representing a memory address, that can be itself stored in memory and manipulated by the program. Thus, the array and record data structures are based on computing the addresses of data items with arithmetic operations, while the linked data structures are based on storing addresses of data items within the structure itself.

Many data structures use both principles, sometimes combined in non-trivial ways (as in XOR linking).

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).

1.4. Examples

There are numerous types of data structures, generally built upon simpler primitive data types:

In addition, graphs and binary trees are other commonly used data structures.

1. An array is a number of elements in a specific order, typically all of the same type (depending on the language, individual elements may either all be forced to be the same type, or may be of almost any type). Elements are accessed using an integer index to specify which element is required. Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or resizable.
2. A linked list (also just called list) is a linear collection of data elements of any type, called nodes, where each node has itself a value, and points to the next node in the linked list. The principal advantage of a linked list over an array, is that values can always be efficiently inserted and removed without relocating the rest of the list. Certain other operations, such as random access to a certain element, are however slower on lists than on arrays.
3. A record (also called tuple or struct) is an aggregate data structure. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called fields or members.
4. A union is a data structure that specifies which of a number of permitted primitive types may be stored in its instances, e.g. float or long integer. Contrast with a record, which could be defined to contain a float and an integer; whereas in a union, there is only one value at a time. Enough space is allocated to contain the widest member datatype.
5. A tagged union (also called variant, variant record, discriminated union, or disjoint union) contains an additional field indicating its current type, for enhanced type safety.
6. An object is a data structure that contains data fields, like a record does, as well as various methods which operate on the data contents. An object is an in-memory instance of a class

from a taxonomy. In the context of object-oriented programming, records are known as plain old data structures to distinguish them from objects.

1.5. Language support

Most assembly languages and some low-level languages, such as BCPL (Basic Combined Programming Language), lack built-in support for data structures. On the other hand, many high-level programming languages and some higher-level assembly languages, such as MASM, have special syntax or other built-in support for certain data structures, such as records and arrays. For example, the C (a direct descendant of BCPL) and Pascal languages support structs and records, respectively, in addition to vectors (one-dimensional arrays) and multi-dimensional arrays.

Most programming languages feature some sort of library mechanism that allows data structure implementations to be reused by different programs. Modern languages usually come with standard libraries that implement the most common data structures. Examples are the C++ Standard Template Library, the Java Collections Framework, and the Microsoft .NET Framework.

Modern languages also generally support modular programming, the separation between the interface of a library module and its implementation. Some provide opaque data types that allow clients to hide implementation details. Object-oriented programming languages, such as C++, Java, and Smalltalk, typically use classes for this purpose.

Many known data structures have concurrent versions which allow multiple computing threads to access a single concrete instance of a data structure simultaneously.TEMPLATE[cite_web, author1 Mark Moir and Nir Shavit, title Concurrent Data Structures, url https://www.cs.tau.ac.il/~shanir/concurrent-data-structures.pdf, website cs.tau.ac.il]

2. binary search tree

2.1.

right|200px|A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

In computer science, binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of container: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

Several variants of the binary search tree have been studied in computer science; this article deals primarily with the basic type, making references to more advanced types when appropriate.

2.2. Definition

A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted left and right. The tree additionally satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree. The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another.

Frequently, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records. The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order

traversal can be very efficient; they are also easy to code.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

Binary search requires an order relation by which every element (item) can be compared with every other element in the sense of a total preorder. The part of the element which effectively takes place in the comparison is called its key. Whether duplicates, i.e. different elements with same key, shall be allowed in the tree or not, does not depend on the order relation, but on the application only.

In the context of binary search trees a total preorder is realized most flexibly by means of a three-way comparison subroutine.

1. When inserting or searching for an element in a binary search tree, the key of each visited node has to be compared with the key of the element to be inserted or found.
2. The shape of the binary search tree depends entirely on the order of insertions and deletions, and can become degenerate.
3. After a long intermixed sequence of random insertion and deletion, the expected height of the tree approaches square root of the number of keys, , which grows much faster than .
4. There has been a lot of research to prevent degeneration of the tree resulting in worst case time complexity of (for details see section Types).

2.3. Operations

Binary search trees support three main operations: insertion of elements, deletion of elements, and lookup (checking whether a key is present).

Searching a binary search tree for a specific key can be programmed recursively or iteratively.

We begin by examining the root node. If the tree is null, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is null. If the searched key is not found after a null subtree is reached, then the key is not present in the tree. This is easily expressed as a recursive algorithm (implemented in Python):

```
def search_recursively(key, node):

if node is None or node.key    key:
      return node
  if key  node.key
  return search_recursively(key, node.right)
```

The same algorithm can be implemented iteratively:

```
def search_iteratively(key, node):

current_node   node
  while current_node is not None:
      if key    current_node.key:
          return current_node
      if key  current_node.key:
          current_node   current_node.right
  return current_node
```

These two examples rely on the order relation being a total order.

If the order relation is only a total preorder a reasonable extension of the functionality is the following: also in case of equality search down to the leaves in a direction specified by the user. A binary tree sort equipped with such a comparison function becomes stable.

Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's height (see tree terminology). On average, binary search trees with nodes have height. However, in the worst case, binary search trees can have height, when the unbalanced tree resembles a linked list (degenerate tree).

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

Here's how a typical binary search tree insertion might be performed in a binary tree in C++:

```
void insert(Node*& root, int key, int value) {
```

```
if (!root)
   root   new Node(key, value);
 else if (key    root->key)
   root->value   value;
 else if (key key)
   insert(root->left, key, value);
 else  // key > root->key
   insert(root->right, key, value);

}
```

The above destructive procedural variant modifies the tree in place. It uses only constant heap space (and the iterative version uses constant stack space as well), but the prior version of the tree is lost. Alternatively, as in the following Python example, we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a persistent data structure:

```
def binary_tree_insert(node, key, value):

if node is None:
      return NodeTree(None, key, value, None)
  if key    node.key:
      return NodeTree(node.left, key, value, node.right)
  if key
```

The part that is rebuilt uses space in the average case and in the worst case.

In either version, this operation requires time proportional to the height of the tree in the worst case, which is time in the average case over all trees, but time in the worst case.

Another way to explain insertion is that in order to insert a new node in the tree, its key is first compared with that of the root. If its key is less than the root's, it is then compared with the key of the root's left child. If its key is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its key: if the key is less than the leaf's key, then it is inserted as the leaf's left child, otherwise as the leaf's right child.

There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

When removing a node from a binary search tree it is mandatory to maintain the in-order

sequence of the nodes.
There are many possibilities to do this. However, the following method which has been proposed by T. Hibbard in 1962s. Robert Sedgewick, Kevin Wayne: Algorithms Fourth Edition. Pearson Education, 2011, , p. 410. guarantees that the heights of the subject subtrees are changed by at most one.
There are three possible cases to consider:

Deleting a node with two children from a binary search tree. First the leftmost node in the right subtree, the in-order successor E, is identified. Its value is copied into the node D being deleted. The in-order successor can then be easily deleted because it has at most one child. The same method works symmetrically using the in-order predecessor C.

In all cases, when D happens to be the root, make the replacement node root again.

Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have only one or no child at all. Delete it according to one of the two simpler cases above.

Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can lead to an unbalanced tree, so some implementations select one or the other at different times.

Runtime analysis: Although this operation does not always traverse the tree down to a leaf, this is always a possibility; thus in the worst case it requires time proportional to the height of the tree. It does not require more even when the node has two children, since it still follows a single path and does not visit any node twice.

```
def find_min(self):   # Gets minimum node in a subtree

current_node   self
  while current_node.left_child:
      current_node   current_node.left_child
  return current_node

def replace_node_in_parent(self, new_value None):

if self.parent:
      if self    self.parent.left_child:
          self.parent.left_child   new_value
      else:
```

```
            self.parent.right_child   new_value
    if new_value:
        new_value.parent   self.parent

def binary_tree_delete(self, key):

if key  self.key:
        self.right_child.binary_tree_delete(key)
        return
    # delete the key here
    if self.left_child and self.right_child: # if both children are present
        successor   self.right_child.find_min()
        self.key   successor.key
        successor.binary_tree_delete(successor.key)
    elif self.left_child:   # if the node has only a *left* child
        self.replace_node_in_parent(self.left_child)
    elif self.right_child:  # if the node has only a *right* child
        self.replace_node_in_parent(self.right_child)
    else:
        self.replace_node_in_parent(None) # this node has no children
```

Once the binary search tree has been created, its elements can be retrieved in-order by recursively traversing the left subtree of the root node, accessing the node itself, then recursively traversing the right subtree of the node, continuing this pattern with each node in the tree as it's recursively accessed. As with all binary trees, one may conduct a pre-order traversal or a post-order traversal, but neither are likely to be useful for binary search trees. An in-order traversal of a binary search tree will always result in a sorted list of node items (numbers, strings or other comparable items).

The code for in-order traversal in Python is given below. It will call callback (some function the programmer wishes to call on the node's value, such as printing to the screen) for every node in the tree.

```
def traverse_binary_tree(node, callback):

if node is None:
        return
    traverse_binary_tree(node.leftChild, callback)
    callback(node.value)
    traverse_binary_tree(node.rightChild, callback)
```

Traversal requires time, since it must visit every node. This algorithm is also , so it is asymptotically optimal.

Traversal can also be implemented iteratively. For certain applications, e.g. greater equal search, approximative search, an operation for single step (iterative) traversal can be very useful. This is, of course, implemented without the callback construct and takes on average and in the worst case.

Sometimes we already have a binary tree, and we need to determine whether it is a BST. This problem has a simple recursive solution.

The BST property—every node on the right subtree has to be larger than the current node and every node on the left subtree has to be smaller than the current node—is the key to figuring out whether a tree is a BST or not. The greedy algorithm—simply traverse the tree, at every node check whether the node contains a value larger than the value at the left child and smaller than the value on the right child—does not work for all cases. Consider the following tree:

```
20
   /  \
 10    30
     /  \
    5    40
```

In the tree above, each node meets the condition that the node contains a value larger than its left child and smaller than its right child hold, and yet it is not a BST: the value 5 is on the right subtree of the node containing 20, a violation of the BST property.

Instead of making a decision based solely on the values of a node and its children, we also need information flowing down from the parent as well. In the case of the tree above, if we could remember about the node containing the value 20, we would see that the node with value 5 is violating the BST property contract.

So the condition we need to check at each node is:

A recursive solution in C++ can explain this further:

```
struct TreeNode {

int key;
   int value;
   struct TreeNode *left;
```

```
   struct TreeNode *right;

};

bool isBST(struct TreeNode *node, int minKey, int maxKey) {

if (node    NULL) return true;
   if (node->key key > maxKey) return false;

   return isBST(node->left, minKey, node->key-1) && isBST(node->right, node->key+1, maxKey);

}
```

node->key+1 and node->key-1 are done to allow only distinct elements in BST.

If we want same elements to also be present, then we can use only node->key in both places.

The initial call to this function can be something like this:

```
if (isBST(root, INT_MIN, INT_MAX)) {

puts("This is a BST.");

} else {

puts("This is NOT a BST!");

}
```

Essentially we keep creating a valid range (starting from) and keep shrinking it down for each node as we go down recursively.

As pointed out in section #Traversal, an in-order traversal of a binary search tree returns the nodes sorted. Thus we only need to keep the last visited node while traversing the tree and check whether its key is smaller (or smaller/equal, if duplicates are to be allowed in the tree) compared to the current key.

1. Deleting a node with no children: simply remove the node from the tree.
2. Deleting a node with one child: remove the node and replace it with its child.

3. Deleting a node with two children: call the node to be deleted D. Do not delete D. Instead, choose either its in-order predecessor node or its in-order successor node as replacement node E (s. figure). Copy the user values of E to D.Of course, a generic software package has to work the other way around: It has to leave the user data untouched and to furnish E with all the BST links to and from D. If E does not have a child simply remove E from its previous parent G. If E has a child, say F, it is a right child. Replace E with F at E's parent.

1. if the node is the left child of its parent, then it must be smaller than (or equal to) the parent and it must pass down the value from its parent to its right subtree to make sure none of the nodes in that subtree is greater than the parent
2. if the node is the right child of its parent, then it must be larger than the parent and it must pass down the value from its parent to its left subtree to make sure none of the nodes in that subtree is lesser than the parent.

2.4. Examples of applications

A binary search tree can be used to implement a simple sorting algorithm. Similar to heapsort, we insert all the values we wish to sort into a new ordered data structure—in this case a binary search tree—and then traverse it in order.

The worst-case time of build_binary_tree is —if you feed it a sorted list of values, it chains them into a linked list with no left subtrees. For example, build_binary_tree() yields the tree (1 (2 (3 (4 (5))))).

There are several schemes for overcoming this flaw with simple binary trees; the most common is the self-balancing binary search tree. If this same procedure is done using such a tree, the overall worst-case time is , which is asymptotically optimal for a comparison sort. In practice, the added overhead in time and space for a tree-based sort (particularly for node allocation) make it inferior to other asymptotically optimal sorts such as heapsort for static list sorting. On the other hand, it is one of the most efficient methods of incremental sorting, adding items to a list over time while keeping the list sorted at all times.

Binary search trees can serve as priority queues: structures that allow insertion of arbitrary key as well as lookup and deletion of the minimum (or maximum) key. Insertion works as previously explained. Find-min walks the tree, following left pointers as far as it can without hitting a leaf:

```
// Precondition: T is not a leaf
function find-min(T):
    while hasLeft(T):
```

```
    T ? left(T)
  return key(T)
```

Find-max is analogous: follow right pointers as far as possible. Delete-min (max) can simply look up the minimum (maximum), then delete it. This way, insertion and deletion both take logarithmic time, just as they do in a binary heap, but unlike a binary heap and most other priority queue implementations, a single tree can support all of find-min, find-max, delete-min and delete-max at the same time, making binary search trees suitable as double-ended priority queues.TEMPLATE[cite_book, last1 Mehlhorn, first1 Kurt, author1-link Kurt Mehlhorn, first2 Peter, last2 Sanders, author2-link Peter Sanders (computer scientist), title Algorithms and Data Structures: The Basic Toolbox, publisher Springer, year 2008, url http://people.mpi-inf.mpg.de/~mehlhorn/ftp/Toolbox/SortedSequences.pdf]

2.5. Types

There are many types of binary search trees. AVL trees and red-black trees are both forms of self-balancing binary search trees. A splay tree is a binary search tree that automatically moves frequently accessed elements nearer to the root. In a treap (tree heap), each node also holds a (randomly chosen) priority and the parent node has higher priority than its children. Tango trees are trees optimized for fast searches.
T-trees are binary search trees optimized to reduce storage space overhead, widely used for in-memory databases

A degenerate tree is a tree where for each parent node, there is only one associated child node. It is unbalanced and, in the worst case, performance degrades to that of a linked list. If your add node function does not handle re-balancing, then you can easily construct a degenerate tree by feeding it with data that is already sorted. What this means is that in a performance measurement, the tree will essentially behave like a linked list data structure.

D. A. Heger (2004)TEMPLATE[Citation, title A Disquisition on The Performance Behavior of Binary Search Tree Data Structures, first1 Dominique A., last1 Heger, year 2004, journal European Journal for the Informatics Professional, volume 5, url http://www.cepis.org/upgrade/files/full-2004-V.pdf, issue 5, pages 67–75] presented a performance comparison of binary search trees. Treap was found to have the best average performance, while red-black tree was found to have the smallest amount of performance variations.

Tree rotations are very common internal operations in binary trees to keep perfect, or near-to-perfect, internal balance in the tree.
If we do not plan on modifying a search tree, and we know exactly how often each item will be

accessed, we can constructTEMPLATE[cite_web, last Gonnet, first Gaston, title Optimal Binary Search Trees, url http://linneus20.ethz.ch:8080/4_7_1.html, work Scientific Computation, publisher ETH Zürich, accessdate 1 December 2013, deadurl yes, archiveurl https://web.archive.org/web/20141012033537/http://linneus20.ethz.ch:8080/4_7_1.html, archivedate 12 October 2014, df] an optimal binary search tree, which is a search tree where the average cost of looking up an item (the expected search cost) is minimized.

Even if we only have estimates of the search costs, such a system can considerably speed up lookups on average. For example, if you have a BST of English words used in a spell checker, you might balance the tree based on word frequency in text corpora, placing words like the near the root and words like agerasia near the leaves. Such a tree might be compared with Huffman trees, which similarly seek to place frequently used items near the root in order to produce a dense information encoding; however, Huffman trees store data elements only in leaves, and these elements need not be ordered.

If we do not know the sequence in which the elements in the tree will be accessed in advance, we can use splay trees which are asymptotically as good as any static search tree we can construct for any particular sequence of lookup operations.

Alphabetic trees are Huffman trees with the additional constraint on order, or, equivalently, search trees with the modification that all elements are stored in the leaves. Faster algorithms exist for optimal alphabetic binary trees (OABTs).

3. Tree (data structure)

3.1.

right|192px|A simple unordered tree; in this diagram, the node labeled 7 has two children, labeled 2 and 6, and one parent, labeled 2. The root node, at the top, has no parent.
In computer science, a tree is a widely used abstract data type (ADT)—or data structure implementing this ADT—that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

Alternatively, a tree can be defined abstractly as a whole (globally) as an ordered tree, with a value assigned to each node. Both these perspectives are useful: while a tree can be analyzed mathematically as a whole, when actually represented as a data structure it is usually represented and worked with separately by node (rather than as a set of nodes and an adjacency list of edges between nodes, as one may represent a digraph, for instance). For example, looking at a tree as a whole, one can talk about "the parent node" of a given node, but in general as a data structure a given node only contains the list of its children, but does not contain a reference to its parent (if any).

3.2. Preliminary definition

A tree is a data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the null or empty tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

3.3. TEMPLATE[anchor, definition, Definition]Mathematical definition

Mathematically, an unordered tree

(or "algebraic tree")
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.

Another equivalent definition is that of a set-theoretic tree that is singly-rooted and whose height is at most . 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

(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" TEMPLATE[cite_web, author Bruce Fields, url http://www.fieldses.org/~bfields/kernel/vfs.txt, title Notes on the Linux kernel].

The concept of an instantiation tree

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.

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).

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}. TEMPLATE[efn, Alternatively, a "partial" version can be employed by excluding {r}.]
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.

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

∅ ∉ ℱ.
 (That is, (U, ℱ) is a hypergraph.)

U ∈ ℱ.

For every X, Y from ℱ,
X ∩ Y ∈
{∅, X, Y}.
 (That is, ℱ is a laminar family.)

For every X from ℱ,
there are only finitely many
Y from ℱ such that
X ⊆ Y.

Then the structure (ℱ, ⊆) is an unordered tree
whose root equals U.
Conversely, if (U, ≤) is an unordered tree,
and ℱ is the set {↓x | x ∈ U}
of all principal ideals of (U, ≤)
then the set system (U, 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 corresponent 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.

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. The correspondent expansion of the previously described tree structures (X, ≤) can be defined as follows.

An ordered tree is a structure
(X,
≤V,
≤S)
where
X is a non-empty set of nodes
and
≤V
and
≤S
are relations on X
called vertical (or also hierarchical) order
and
sibling order, respectively.
The structure is subject to the following conditions:

(X, ≤V)
is a partial order that is an unordered tree as defined in the previous subsection.

(X, ≤S)
is a partial order.

The partial orders
≤V and
≤S
are orthogonal:
((<V)
∪
(>V))
∩
((<S)
∪
(>S))

∅.
That is, distinct nodes cannot be comparable both in
≤V and
≤S.

For every sibling set S of (X, ≤V),
the restriction (S, ≤S)
is a singly-branching tree.
That is, (S, ≤S) is a linear order,
and if S is infinite,
then (S, ≤S)
must be isomorphic to
(ℕ, ≤), the usual ordering of natural numbers.

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

:{| cellpadding 3 style "border:0"

| (<H)
|
| (≤V) ○ (<S) ○ (≥V)
| (the horizontal order),
|-
| (<L⁻)
|
| (>V) ∪ (<H)
| (the "discordant" linear order),
|-
| (<L⁺)
|
| (<V) ∪ (<H)
| (the "concordant" linear order).
|}

This amounts to a "V-S-H-L±" system of five partial orders
≤V,
≤S,
≤H,
≤L⁺,
≤L⁻

on the same set X of nodes, in which, except for the pair
{ ≤S,
≤H },
any two relations uniquely determine the other three.

Notes about notational conventions:

The relation composition symbol ○ used in this subsection is to be interpreted left-to-right (as \circ_l).

Symbols < and ≤ express the strict and non-strict versions
of a partial order.

Symbols > and ≥ express the converse relations.

The ≺ symbol is used for the
covering relation of ≤ which is the immediate version of a partial order.

This yields six versions
≺,
<, ≤,
≻,
>, ≥ for a single partial order relation.
Except for
≺
and
≻,
each version uniquely determines the others.
Passing from ≺ 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⁻.

The partial orders
≤V and

≤H
are complementary:
(<V)
⊎
(>V)
⊎
(<H)
⊎
(>H)

X × X
∖
idX.
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
≺L⁻ and
≺L⁺ correspond to pre-order traversal and
post-order traversal, respectively.
If x
≺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 (x,y) of the latter case form the relation

(≺L⁻)
∖
(<H)

which is a partial map that assigns each non-leaf node its first child node.
Similarly,
(≻L⁺) ∖ (>H)
assigns each non-leaf node with finitely many children its last child node.

The table on the right shows a correspondence of introduced relations
to XPath axes.
For a context node

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
≤L⁻.
A "concordance" would be achieved, if the vertical order ≤V was defined oppositely, with the bottom-up direction outwards the root like in set theory in accordance to natural trees.

Below is the list of partial maps that are typically used for ordered tree traversal.

Each map is a distinguished functional subrelation of
≤L⁻ or of its opposite.

≺V
… the parent-node partial map,

≻S
… the previous-sibling partial map,

≺S
… the next-sibling partial map,

(≺L⁻) ∖ (<H)
… the first-child partial map,

(≻L⁺) ∖ (>H)
… the last-child partial map,

≻L⁻
… the previous-node partial map,

≺L⁻
… the next-node partial map.

The traversal maps constitute a partial unary algebraTEMPLATE[cite_web, url

http://www.math.chapman.edu/~jipsen/structures/doku.php/unary_algebras, title Unary Algebras]
(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.TEMPLATE[cite_journal, url http://citeseerx.ist.psu.edu/viewdoc/download?doi 10.1.1.156.7781&rep rep1&type pdf#page 3, author J.T. Mühlberg, G. Lüttgen, title Verifying compiled file system code, volume Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, publisher Springer, Berlin, Heidelberg, year 2009, format PDF]

Similarly to the "V-S-H-L±" 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.

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

An ordered tree is a structure (X, lc, rs) 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:

The partial maps lc and rs are disjoint, i.e.
(lc) ∩ (rs)

∅
.

The inverse of (lc) ∪ (rs) is a partial map p
such that the partial algebra (X, p) is an unordered tree.

The partial order structure (X, ≤V, ≤S)
is obtained as follows:

:{| cellpadding 3 style "border:0"

| (≺S)
|
| (rs),
|-
| (≻V)
|
| (lc) ○ (≤S).
|}

3.4. Terminology used in trees

TEMPLATE[defn, A forest is a set of n 0 disjoint trees.]

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. Nodes in a tree could have next/previous references or references to their parent nodes.

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.

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

(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 a forest (a list of trees), where a tree consists of a value and a forest (the subtrees of its children):

f:
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

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

This data structure defines a directed graph, 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.

As an ADT, 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 E

children: T F

nil: () F

node: E × F 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).

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):

together with:

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 tree must include fork operation.

1. A rooted tree with the "away from root" direction (a more narrow term is an "arborescence"), meaning:
2. 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),

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

3.5. Terminology

A node is a structure which may contain a value or condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more child nodes, which are below it in the tree (by convention, trees are drawn growing downwards). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent.

An internal node (also known as an inner node, inode for short, or branch node) is any node of a tree that has child nodes. Similarly, an external node (also known as an outer node, leaf node, or terminal node) is any node that does not have child nodes.

The topmost node in a tree is called the root node. Depending on definition, a tree may be required to have a root node (in which case all trees are non-empty), or may be allowed to be empty, in which case it does not necessarily have a root node. Being the topmost node, the root node will not have a parent. It is the node at which algorithms on the tree begin, since as a data structure, one can only pass from parents to children. Note that some algorithms (such as post-order depth-first search) begin at the root, but first visit leaf nodes (access the value of leaf nodes), only visit the root last (i.e., they first access the children of the root, but only access the value of the root last). All other nodes can be reached from it by following edges or links. (In the formal definition, each such path is also unique.) In diagrams, the root node is conventionally drawn at the top. In some trees, such as heaps, the root node has special properties. Every node in a tree can be seen as the root node of the subtree rooted at that node.

The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The depth of a node is the length of the path to its root (i.e., its root path). This is commonly needed in the manipulation of the various self-balancing trees, AVL Trees in particular. The root node has depth zero, leaf nodes have height zero, and a tree with only a single node (hence both a root and leaf) has depth and height zero. Conventionally, an empty tree (tree with no nodes, if such are allowed) has height 1.

A subtree of a tree T is a tree consisting of a node in T and all of its descendants in T.TEMPLATE[efn, 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).] Nodes thus correspond to subtrees (each node corresponds to the

subtree of itself and all its descendants) – the subtree corresponding to the root node is the entire tree, and each node is the root node of the subtree it determines; the subtree corresponding to any other node is called a proper subtree (by analogy to a proper subset).

3.6. 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").

3.7. 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.

3.8. Generalizations

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: v f

3.9. Traversal methods

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.

3.10. Common operations

1. Enumerating all the items
2. Enumerating a section of a tree
3. Searching for an item
4. Adding a new item at a certain position on the tree
5. Deleting an item
6. Pruning: Removing a whole section of a tree
7. Grafting: Adding a whole section to a tree
8. Finding the root for any node
9. Finding the lowest common ancestor of two nodes

3.11. Common uses

1. Representing hierarchical data such as syntax trees
2. Storing data in a way that makes it efficiently searchable (see binary search tree and tree traversal)
3. Representing sorted lists of data
4. As a workflow for compositing digital images for visual effects