

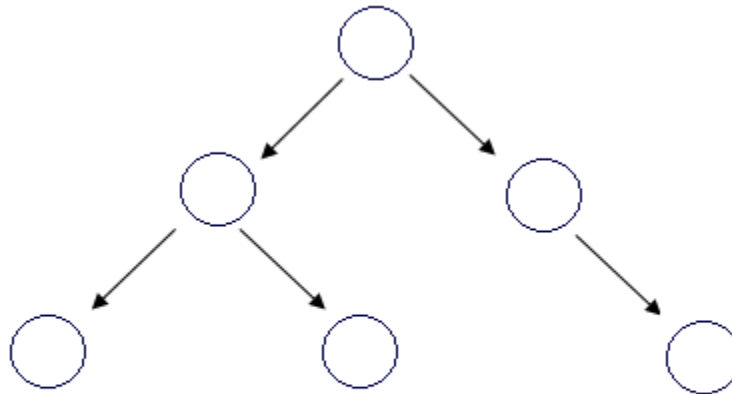
Tree

Tree is a collection of nodes in which there is a root node and all other nodes are recursively children of this root node. What does it mean is that root node has some child nodes, those child nodes have their child nodes and so on.

We represent each node of a tree by an object. As with linked lists, we assume that each node contains a key field. The remaining fields of interest are pointers to other nodes, and they vary according to the type of tree.

Binary Tree

A tree in which each node can have at max only two children is called Binary tree. Binary Search Tree is binary tree. We shall discuss more about BST later. we use the fields parent, leftC, and rightC to store pointers to the parent, left child, and right child of each node in a binary tree T . If $\text{parent}[\text{node}] = \text{NIL}$, then 'node' is the root. If 'node' has no left child, then $\text{leftC}[\text{node}] = \text{NIL}$, and similarly for the right child. The root of the entire tree T is pointed to by the attribute $\text{root}[T]$. If $\text{root}[T] = \text{NIL}$, then the tree is empty.

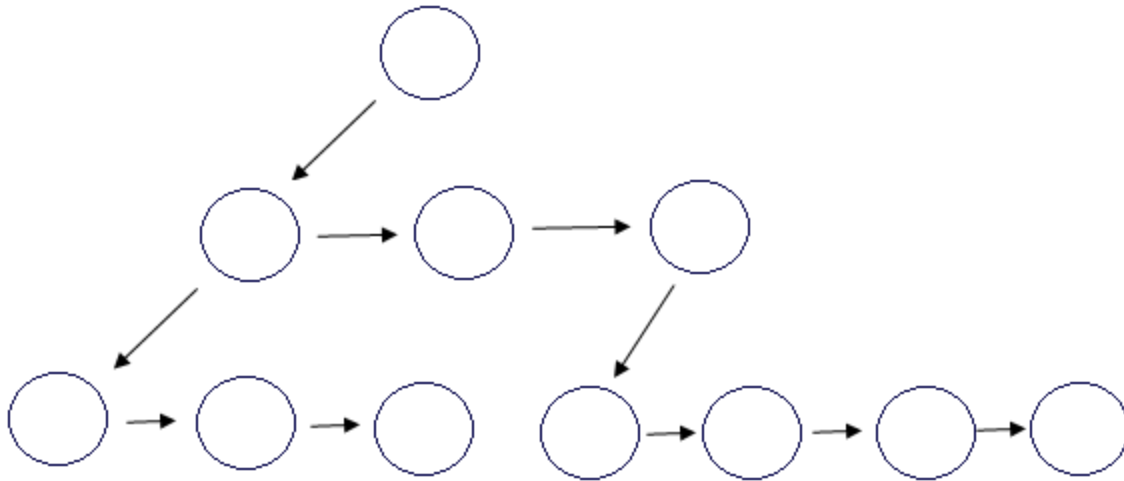


Binary Tree

Rooted trees with unbounded branching

The scheme for representing a binary tree can be extended to any class of trees in which the number of children of each node is at most some constant k : we replace the left and right fields by $\text{child}_1, \text{child}_2, \dots, \text{child}_k$. This scheme no longer works when the number of children of a node is unbounded, since we do not know how many fields (arrays in the multiple-array representation) to allocate in advance. Moreover, even if the number of children k is bounded by a large constant but most nodes have a small number of children, we may waste a lot of memory.

Fortunately, there is a clever scheme for using binary trees to represent trees with arbitrary numbers of children. It has the advantage of using only $O(n)$ space for any n -node rooted tree. The left-child, right-sibling representation is shown below. As before, each node contains a parent pointer `parent`, and `root[T]` points to the root of tree T . Instead of having a pointer to each of its children, however, each node x has only two pointers:



Rooted trees with unbounded branching

The left-child, right-sibling representation of a tree T . Each node x has fields `parent[x]`, `left-child[x]`, and `right-sibling[x]`.

1. `left-child[x]` points to the leftmost child of node x , and
2. `right-sibling[x]` points to the sibling of x immediately to the right. If node x has no children, then `left-child[x] = NIL`, and if node x is the rightmost child of its parent, then `right-sibling[x] = NIL`.

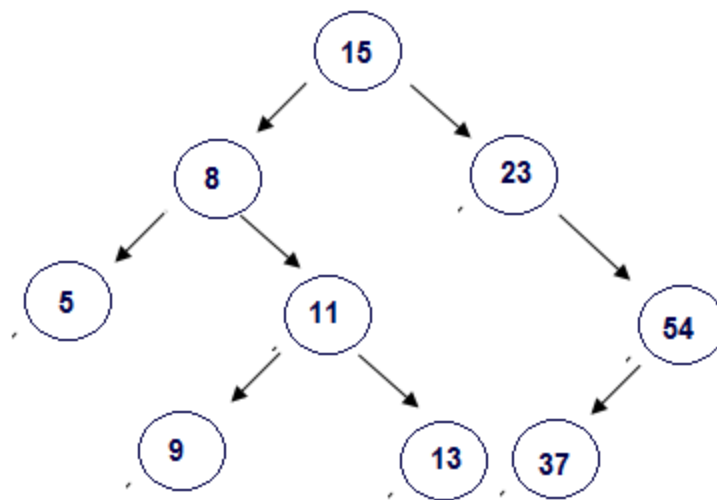
Binary Search Tree

Search trees are data structures that support many dynamic-set operations, including `SEARCH`, `MINIMUM`, `MAXIMUM`, `PREDECESSOR`, `SUCCESSOR`, `INSERT`, and `DELETE`. Thus, a search tree can be used both as a dictionary and as a priority queue. Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time. The expected height of a randomly built binary search tree is $O(\lg n)$, so that basic dynamic-set operations on such a tree take $\Theta(\lg n)$ time on average. In practice, we can't always guarantee that binary search trees are built randomly, but there are variations of binary search trees whose worst-case performance on basic operations can be guaranteed to be

good. One such variation is red-black trees, which have height $O(\lg n)$. Another example of variation is B-trees, which are particularly good for maintaining databases on random-access, secondary (disk) storage.

What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in below figure. Such a tree can be represented by a linked data structure in which each node is an object. In addition to a key field and satellite data, each node contains fields left, right, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is NIL.



Binary Search Tree

The keys in a binary search tree are always stored in such a way as to satisfy the binary search-tree property:

- Let x be a node in a binary search tree. If y is a node in the left subtree of x , then $\text{key}[y] \leq \text{key}[x]$. If y is a node in the right subtree of x , then $\text{key}[x] \leq \text{key}[y]$.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an inorder tree walk. This algorithm is so named because the key of the root of a subtree is printed between the values in its left subtree and those in its right subtree. (Similarly, a preorder tree walk prints the root before the values in either subtree, and a postorder tree walk prints the root after the values in its subtrees.) To use the following procedure to print all the elements in a binary search tree T , we call INORDER-TREE-WALK ($\text{root}[T]$).

```
INORDER-TREE-WALK(x)
1. if x ≠ NIL
2.   then INORDER-TREE-WALK(left[x])
3.   print key[x]
4.   INORDER-TREE-WALK(right[x])
```

Querying a binary search tree

A common operation performed on a binary search tree is searching for a key stored in the tree. Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR. In this section, we shall examine these operations and show that each can be supported in time $O(h)$ on a binary search tree of height h .

Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key k , TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

```
TREE-SEARCH(x, k)
1. if x = NIL or k = key[x]
2.   then return x
3. if k < key[x]
4.   then return TREE-SEARCH(left[x], k)
5. else return TREE-SEARCH(right[x], k)
```

The procedure begins its search at the root and traces a path downward in the tree. For each node x it encounters, it compares the key k with $\text{key}[x]$. If the two keys are equal, the search terminates. If k is smaller than $\text{key}[x]$, the search continues in the left subtree of x , since the binary-search-tree property implies that k could not be stored in the right subtree. Symmetrically, if k is larger than $\text{key}[x]$, the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

The same procedure can be written iteratively by "unrolling" the recursion into a while loop. On most computers, this version is more efficient.

```
ITERATIVE-TREE-SEARCH(x, k)
1. while x ≠ NIL and k ≠ key[x]
2.   do if k < key[x]
3.     then x ← left[x]
4.     else x ← right[x]
5. return x
```

Minimum and maximum

An element in a binary search tree whose key is a minimum can always be found by following left child pointers from the root until a NIL is encountered. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x .

```
TREE-MINIMUM(x)
1. while left[x] ≠ NIL
2.   do x ← left[x]
3. return x
```

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If a node x has no left subtree, then since every key in the right subtree of x is at least as large as $\text{key}[x]$, the minimum key in the subtree rooted at x is $\text{key}[x]$. If node x has a left subtree, then since no key in the right subtree is smaller than $\text{key}[x]$ and every key in the left subtree is not larger than $\text{key}[x]$, the minimum key in the subtree rooted at x can be found in the subtree rooted at $\text{left}[x]$.

The pseudocode for TREE-MAXIMUM is symmetric.

```
TREE-MAXIMUM(x)
1. while right[x] ≠ NIL
2.   do x ← right[x]
3. return x
```

Both of these procedures run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a path downward from the root.

Successor and predecessor

Given a node in a binary search tree, it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than $\text{key}[x]$. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

```
TREE-SUCCESSOR(x)
1. if right[x] ≠ NIL
2.   then return TREE-MINIMUM(right[x])
3. y ← p[x]
4. while y ≠ NIL and x = right[y]
5.   do x ← y
6.   y ← p[y]
```

7. return y

The code for TREE-SUCCESSOR is broken into two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in the right subtree, which is found in second line by calling TREE-MINIMUM(right[x]).

On the other hand, if the right subtree of node x is empty and x has a successor y , then y is the lowest ancestor of x whose left child is also an ancestor of x . To find y , we simply go up the tree from x until we encounter a node that is the left child of its parent; this is accomplished by 3rd to 7th lines of TREE-SUCCESSOR. The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since we either follow a path up the tree or follow a path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.

Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold. As we shall see, modifying the tree to insert a new element is relatively straight-forward, but handling deletion is somewhat more intricate.

Insertion

To insert a new value v into a binary search tree T , we use the procedure TREE-INSERT. The procedure is passed a node z for which $\text{key}[z] = v$, $\text{left}[z] = \text{NIL}$, and $\text{right}[z] = \text{NIL}$. It modifies T and some of the fields of z in such a way that z is inserted into an appropriate position in the tree.

```
TREE-INSERT(T, z)
1 y ← NIL
2 x ← root[T]
3 while x ≠ NIL
4   do y ← x
5   if key[z] < key[x]
6     then x ← left[x]
7   else x ← right[x]
8 p[z] ← y
9 if y = NIL
10  then root[T] ← z // Tree T was empty
11 else if key[z] < key[y]
12  then left[y] ← z
13 else right[y] ← z
```

Just like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and traces a path downward. The pointer x traces the path, and the pointer y is

maintained as the parent of x . After initialization, the while loop in lines 3-7 causes these two pointers to move down the tree, going left or right depending on the comparison of $\text{key}[z]$ with $\text{key}[x]$, until x is set to NIL. This NIL occupies the position where we wish to place the input item z . Lines 8-13 set the pointers that cause z to be inserted. Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h .

Deletion

The procedure for deleting a given node z from a binary search tree takes as an argument a pointer to z . The procedure considers the three cases. If z has no children, we modify its parent $p[z]$ to replace z with NIL as its child. If the node has only a single child, we "splice out" z by making a new link between its child and its parent. Finally, if the node has two children, we splice out z 's successor y , which has no left child and replace z 's key and satellite data with y 's key and satellite data.

```
TREE-DELETE(T, z)
1. if left[z] = NIL or right[z] = NIL
2.   then y ← z
3. else y ← TREE-SUCCESSOR(z)
4. if left[y] ≠ NIL
5.   then x ← left[y]
6. else x ← right[y]
7. if x ≠ NIL
8.   then p[x] ← p[y]
9. if p[y] = NIL
10.  then root[T] ← x
11. else if y = left[p[y]]
12.   then left[p[y]] ← x
13.   else right[p[y]] ← x
14. if y ≠ z
15.   then key[z] ← key[y]
16.   copy y's satellite data into z
17. return y
```

In lines 1-3, the algorithm determines a node y to splice out. The node y is either the input node z (if z has at most 1 child) or the successor of z (if z has two children). Then, in lines 4-6, x is set to the non-NIL child of y , or to NIL if y has no children. The node y is spliced out in lines 7-13 by modifying pointers in $p[y]$ and x . Splicing out y is somewhat complicated by the need for proper handling of the boundary conditions, which occur when $x = \text{NIL}$ or when y is the root. Finally, in lines 14-16, if the successor of z was the node spliced out, y 's key and satellite data are moved to z , overwriting the previous key and satellite data. The node y is returned in line 17 so that the calling procedure can recycle it via the free list. The procedure runs in $O(h)$ time on a tree of height h .

Source:

<http://www.learnalgorithms.in/#>