As mentioned earlier, one of the main purposes of sorting is to facilitate fast searching. However, while a sorted sequence stored in a linear array is good for searching, it is expensive to add items to the sequence and to delete items from it. Binary search trees give you the best of both worlds: fast search and fast update. We first introduce binary trees and then explain how to sort them and thus turn them into binary search trees.

**Binary trees.** We have used binary trees repeatedly and now return to a more formal and systematic introduction. There are many ways to define a binary tree, and the most compact one is recursive: a *binary tree* is either empty or a node with a binary tree as left subtree and binary tree as right subtree. As an example consider an arithmetic expression used in the solution of quadratic equations: $\sqrt{b^2 - 4ac}$. This expression can be represented by a binary tree as in Figure 14. When we write pseudo-code for trees we assume each node is a record storing an item and pointers to two children.

```
struct Node { item info; Node *l, *r; }
typedef Node *Tree.
```

Sometimes it is convenient to also store a pointer to the parent, but for now we will do without.

**Terminology and Properties.** Terms for relations between family members such as *child*, *parent*, *sibling* are also used for nodes in a tree, as illustrated in Figure 15. In a tree, every node has one parent, except the root, which has no parent. A *leaf* is a node without children, and all other nodes are *internal*. A node $v$ is a *descendent* of $\mu$ if

![Figure 15: Every node has a parent and two children (some of which may be NULL). The top node is the root and the nodes without children are the leaves.](image)

$\nu = \mu$ or $\nu$ is a descendent of a child of $\mu$. $\mu$ is an *ancestor* of $\nu$ if $\nu$ is a descendent of $\mu$. The *subtree* of $\mu$ consists of all descendents of $\mu$. An *edge* is a parent-child pair. The *size* of the tree is the number of nodes. A binary tree is *full* if every internal node has two children. Every full binary tree has one more leaf than internal node. To count its edges we can either count 2 for each internal node or 1 for every node other than the root. Either way, the total number of edges is one less than the size of the tree.

A *path* is a sequence of contiguous edges with no edge repeated. Usually we only consider paths that descend or paths that ascend. The *length* of a path is the number of edges. For every node $\mu$ there is a unique path from the root to $\mu$. The length of that path is the *depth* of $\mu$. The *height* of the tree is the maximum depth of any node. The *internal path length* is the sum of depths over all internal nodes and, similarly, the *external path length* is the sum of
depths over all leaves. In a full binary tree, the number of internal paths covering any one edge is one less than the number of external paths. It follows that the external path length is equal to the internal path length plus the number of edges in the tree.

Traversal. There is no unique or best sequence to visit the nodes of a tree, and the following three are most common:

- **preorder**: 1. root, 2. left subtree, 3. right subtree.
- **inorder**: 1. left subtree, 2. root, 3. right subtree.
- **postorder**: 1. left subtree, 2. right subtree, 3. root.

As an example consider the three orders of the tree in Figure 14.

As an example consider the three orders of the tree in Figure 14.

\[
\begin{align*}
\text{Lisp notation} & : \quad (b) \quad (k) \quad (l) \quad (m) \quad (n) \quad (s) \quad (x) \quad (y) \\
\text{symmetric notation} & : \quad b \quad k \quad l \quad m \quad n \quad s \quad x \quad y \\
\text{Polish notation} & : \quad b^k l^m n^s x^y
\end{align*}
\]

It is straightforward to write recursive functions that traverse a binary tree in any one of the three orders:

```c
void PREFORM(Node * t)
    if t != NULL then print (t \rightarrow info);
        PREFORM(t \rightarrow \ell);
        PREFORM(t \rightarrow r)
    endif.
```

The functions for inorder and postorder simply permute the print statement and the recursive calls. A traversal takes constant time per node and therefore time \(O(n)\) altogether, where \(n\) is the number of nodes.

Searching and inserting. A binary tree is sorted if its inorder sequence is sorted. A **binary search tree** is a sorted binary tree. For example, the tree in Figure 16 is sorted in alphabetical order.

![Figure 16: Items are stored in the internal nodes with sorted inorder sequence: b, k, l, m, r, s, x, y. The shaded edges indicate the path from the root down to the new shaded node storing v.](image)

We can search in a binary search tree by tracing a path starting at the root.

```c
Node * SEARCH(Tree t, item K)
    case t = NULL: return NULL;
    case K < t \rightarrow info: return SEARCH(t \rightarrow \ell, K);
    case K = t \rightarrow info: return t;
    case K > t \rightarrow info: return SEARCH(t \rightarrow r, K).
```

The running time depends on the length of the path, which is at most the height of the tree. Let \(n\) be the size. In the worst case the tree is a linked list, like the left tree in Figure 17, and searching takes time \(O(n)\). In the best case the tree is perfectly balanced, like the right tree in Figure 17, and search takes only time \(O(\log n)\).

![Figure 17: A perfectly unbalanced tree of height 3 to the left and a perfectly balanced tree of height 3 to the right.](image)

To add a new item is similarly straightforward: follow a path from the root to a leaf as in function SEARCH and replace that leaf by a new node storing the item, as illustrated in Figure 16.

```c
Tree * INSERT(Tree t, item K)
    case t = NULL: return Node(K, NULL, NULL);
    case K < t \rightarrow info: return INSERT(t \rightarrow \ell, K);
    case K = t \rightarrow info: print (K already here);
    case K > t \rightarrow info: return INSERT(t \rightarrow r, K).
```

Again the running time depends on the length of the path.

**Expected Search Time.** A binary search tree constructed by \(n\) insertions can take on all kinds of shapes. If the insertions come in a random order then the tree is usually close to being perfectly balanced. We consider the expected number of comparison for searching and distinguish between a successful and an unsuccessful search. The \(n\)-th harmonic number is \(H_n = \sum_{i=1}^{n} \frac{1}{i}\). We have

\[
\ln(n+1) = \int_{1}^{n+1} \frac{dx}{x} \leq H_n \leq 1 + \int_{1}^{n} \frac{dx}{x} = 1 + \ln n.
\]

Since \(\ln n \approx 0.693 \ldots \log_2 n\), we have \(2H_2 - 2 \approx 1.386\log_2 n\).
CLAIM. The expected number of comparisons for a successful and an unsuccessful search are $S_n = 2(1 + \frac{1}{n}) \cdot H_n - 3$ and $U_n = 2H_{n+1} - 2$.

PROOF. In the analysis we refer to the $n$ nodes of the tree as internal nodes and stipulate $n + 1$ external nodes, although these are not stored. If the searched key is the $i$-th one inserted then the expected number of comparisons is $U_{i-1} + 1$. Hence,

$$S_n = \frac{1}{n} \sum_{i=1}^{n} (U_{i-1} + 1) = 1 + \frac{1}{n} \sum_{i=0}^{n-1} U_i.$$  

We get a second equation from the relation between internal and external path lengths. Let $I_n$ and $E_n$ be their expected values. We have $S_n = I_n + 2n$, because there are $n$ internal paths and a successful search takes one more comparison than there are edges in the path it traverses. We have $U_n = E_n$ because there are $n + 1$ external paths. Since $E_n = I_n + 2n$, we have

$$S_n = 1 + \frac{E_n - 2n}{n} = \frac{n+1}{n} \cdot U_n - 1.$$  

Using this together with the first equation we get

$$(n + 1) \cdot U_n = 2n + \sum_{i=0}^{n-1} U_i,$$

$$n \cdot U_{n-1} = 2(n - 1) + \sum_{i=0}^{n-2} U_i,$$

and therefore $(n + 1) \cdot U_n - n \cdot U_{n-1} = 2 + U_{n-1}$. This is the same as $U_n = U_{n-1} + \frac{2}{n+1}$. We reduce the recurrence relation to a sum and solve it:

$$U_n = \frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{2} = 2 \cdot H_{n+1} - 2$$

because $U_0 = 0$. Using the relation between $U_n$ and $S_n$ we get

$$S_n = \frac{n+1}{n} \cdot (2H_{n+1} - 2) - 1,$$

$$= 2 \cdot \frac{n+1}{n} \cdot \left( H_n + \frac{1}{n+1} - 1 \right) - 1,$$

$$= 2 \left( 1 + \frac{1}{n} \right) \cdot H_n - 3.$$  

Deleting. The main idea for deleting an item is the same as for inserting: follow the path from the root to the node $\nu$ that stores the item.

Case 1. $\nu$ has no internal node as a child. In this case we can simply remove $\nu$, as shown in Figure 18.

![Figure 18: Substitute a leaf for $\nu$.](image18)

Case 2. $\nu$ has one internal child. Make that child the child of the parent of $\nu$, as in Figure 19.

![Figure 19: Connect the parent with the child.](image19)

Case 3. $\nu$ has two internal children. Find the rightmost internal node in the left subtree, remove it, and substitute it for $\nu$, as illustrated in Figure 20.

![Figure 20: Store $J$ in $\nu$ and delete the node that used to store $J$.](image20)

The analysis of the expected search time in a binary search tree constructed by a random sequence of insertions and deletions is considerably more challenging than if no deletions are present. Even the definition of a random sequence is ambiguous in this case.

We thus just proved that if we add items in a random order to a binary search tree then the expected search time is close to optimal. Compare this result with the earlier analysis of quicksort.