Threaded binary search trees.

A binary search tree is threaded by making all left child pointers, which are normally null to point to their inorder predecessors, and all right child pointers, which are normally null to point to their inorder successors, respectively.

The inorder traversal of this tree yields: A B C D E F G H I.
Since C’s inorder predecessor is B, the left thread from C points to B. The right thread from C points to D which is the inorder successor of D. Since A has no predecessor, A’s left thread doesn’t exist (dangling thread) as does for G whose right thread doesn’t exist.

Some variants of threaded trees:

- Right-threaded tree: Only the thread to one’s successor is maintained.
- Left-threaded tree: Only the thread to one’s predecessor is maintained.
Rationale for threading:

- Via threading, we could dispense with the recursive inorder-traversal to search and print the three (recursion is costly since it’d use one or more stacks), and do an iterative (or via an iterator) traversal.

- A binary tree with \( n \) nodes would have \( 2n \) links, most of which would be unused.

  Total number of non-null links for an \( n \) node tree = \( n - 1 \)

  Therefore, total number of unused null links =

  \[
  2n - (n - 1) = n + 1
  \]

That’s a lot of wasted memory for large \( n \).

Threads could be set up on those unused links allowing one to traverse the tree iteratively.

How do we insert a new node in a threaded tree?

A new node \( D \) is to be inserted next as the right child of \( B \) which has no right child. The alignment of threads now point in the following way:
Let $p$ = pointer for B, and $q$ = new Node(D,null, null);
Then

$q \rightarrow Lthread = p$;
$q \rightarrow Rthread = p \rightarrow Rthread$;
$p \rightarrow Rthread = NULL$;
$p \rightarrow right = q$

Another case. Suppose now node X to be inserted as the right child of B with D as X’s right child.

The tree now changes to
The pointers and threads now change to the following allocations:

Let $p =$ Pointer for B, and $q =$ New node (X, null, null). Then,

$$q \rightarrow \text{right} = p \rightarrow \text{right};$$
$$p \rightarrow \text{right} = q;$$
$$q \rightarrow \text{Lthread} = p;$$
$$q \rightarrow \text{right} \rightarrow \text{left} \rightarrow \text{Lthread} = q$$

One could look into other insertions, deletions in a similar manner.

**Ordered trees.**

An ordered rooted tree is one where a node may have a number of children (not just two), but with a “sense” of order among them (ordered from “oldest” to “youngest”).
Position trees.

In this case, the children are indicated by their “position” rather than by “age”. A $k$-ary position tree is a position tree with maximum $k$ branches. Binary tree is a position tree with $k = 2$.

An ordered tree of $n$ nodes can be transformed into a binary tree of $(n - 1)$ nodes. This is shown next.

If we retain the pointer to the oldest child from its parent and configure the linked list from the oldest child to its siblings as a branch emanating from the oldest child, we would get its binary tree equivalent.
In the same way we could convert a Forest or an orchard (set of trees) into one binary tree.

**Splay trees.**

A self-balancing binary tree is the one where

- recently inserted node
- recently accessed node

could be quickly accessed again. Insertions, look-ups and deletions are done in $O(\lg n)$ amortized time.

What is an amortized time? It is the average running time for an algorithm using the worst-case sequence scenarios given that worst-case scenarios are relatively rare. In binary tree searching, only about 10% of the stored data are accessed 90% of the time (90-10 rule). The amortized time complexity is not probability driven computation; it is the guaranteed upper bound time that would be expected about 10% of the cases.

The basic operation: splaying.

Splaying a tree for a certain node places the latter at the root of the tree through its rearrangement via rotations.
Example. The node $x$ is accessed. It now moves to the root of the tree.

Splaying: sequence of splay steps
each step brings $x$ closer to root
previously accessed nodes stay around $x$

The three factors predicating splaying:

- Is $x$ left or right child of its parent $p$?
- Is $p$ the root?
- Is $p$ the left or right child of its parent $g$?

Zig step: In this case $p$ is the root. Zig step is done only as the last step of splay operation and when it has odd depth at the beginning of operation.

Zig-Zig step: $p$ is not the root, and $x$ and $p$ are on the same side.
Zig-Zag step: In this case, $p$ is not the root, and $x$ and $p$ are on the opposite sides.

Deletion of a node from a Splay tree:

Step 1. Make $x$ grandfather with $g$ as its right child. Make $b$ the right child of $p$. 
Step 2. Make $b$ to be parent of $p$ with the latter $b$’s left child.

Step 3: Delete $x$ from the root and promote the first left-child of $x$ to be root.