Binary search trees (BST)
A tree is a structure that represents a parent-child relation on a set of objects.

An element of a tree is called a node or vertex.

The root of a tree is the unique node that does not have a parent (node A in the example below).

A node is a leaf if it doesn’t have children (C,F,G,H,I).

A node which is not a leaf is called an inner node.
A node $w$ is a **descendant** of a node $v$ if there are nodes $v_1, v_2, \ldots, v_t$, with $v_1 = v$ and $v_t = w$, such that $v_{i+1}$ is a child of $v_i$ for all $i$. $v$ is called an **ancestor** of $v$.

The **subtree** rooted at $v$ is a tree containing $v$ and its descendants.
The **depth** of a node $v$ is the number of edges in the path from the root to $v$.

In the example below, the depth of B is 1.
The height of a node \( v \) is the maximum number of edges in a path from \( v \) to a descendant of \( v \). In the example below, the height of B is 2.

The height of a tree is the height of its root = the maximum depth of a leaf.

The height of an empty tree is \(-1\).
The degree of a node $v$ is the number of children of $v$. For example, degree(A) = 3.
A **binary tree** is a tree in which each node has at most two children.

More precisely, a node can have a **left child** or not, and have a **right child** or not.
A full binary tree is a binary tree in which each internal node has exactly two children.
A perfect binary tree is a full binary tree in which all leaves have the same depth.
A complete binary tree is either a perfect binary tree, or a tree that is obtained from a perfect binary tree by deleting consecutive leaves of the tree from right to left.
A dynamic set ADT is a structure that stores a set of elements. Each element has a (unique) key and satellite data. The structure supports the following operations.

- **Search**($S$, $k$) Return the element whose key is $k$.
- **Insert**($S$, $x$) Add $x$ to $S$.
- **Delete**($S$, $x$) Remove $x$ from $S$ (the operation receives a pointer to $x$).
- **Minimum**($S$) Return the element in $S$ with smallest key.
- **Maximum**($S$) Return the element in $S$ with largest key.
- **Successor**($S$, $x$) Return the element in $S$ with smallest key that is larger than $x$.key.
- **Predecessor**($S$, $x$) Return the element in $S$ with largest key that is smaller than $x$.key.
The dynamic set ADT can be implemented using linked list.

Is this implementation good?
A binary search tree (BST) is an implementation of the dynamic set ADT. The elements of the set are stored in the nodes of a binary tree (exactly one element in each node) such that the following property is satisfied for every node $x$.

- For every node $y$ in the left subtree of $x$, $y$.key $\leq$ $x$.key.
- For every node $z$ in the right subtree of $x$, $z$.key $> x$.key.
Each node $x$ in the tree has the following fields.

- key: The key of the element of $x$.
- Satellite data.
- left: pointer to the left child of $x$ (can be NULL).
- right: pointer to the right child of $x$ (can be NULL).
- $p$: pointer to the parent of $x$ (NULL for the root).

$T$.root is a pointer to the root of tree $T$. 
**Tree walk**

**Inorder(x)**

1. if x ≠ NULL
2. Inorder(x.left)
3. print x.key
4. Inorder(x.right)

Time complexity: $\Theta(n)$, where $n$ is the number of nodes.

We also define two additional procedures.

- **Preorder** Print $x$.key before the two recursive calls.
- **Postorder** Print $x$.key after the two recursive calls.

Inorder: 2 3 5 6 7 8
Preorder: 6 3 2 5 8 7
Postorder: 2 5 3 7 8 6
Recursive search

**Search**$(x, k)$

(1) **if** $x = \text{NULL}$ **OR** $k = x.key$
(2) **return** $x$
(3) **if** $k < x.key$
(4) **return** Search$(x.\text{left}, k)$
(5) **else**
(6) **return** Search$(x.\text{right}, k)$

Time complexity: $\Theta(h)$ where $h$ is the height of the tree.
Recursive search

\textbf{Search}(x, k)

(1) \textbf{if } x = \text{NULL OR } k = x.\text{key} \\
(2) \hspace{0.5cm} \text{return } x \\
(3) \textbf{if } k < x.\text{key} \\
(4) \hspace{0.5cm} \text{return } \text{Search}(x.\text{left}, k) \\
(5) \textbf{else} \\
(6) \hspace{0.5cm} \text{return } \text{Search}(x.\text{right}, k)

Time complexity: $\Theta(h)$ where $h$ is the height of the tree.
**Recursive search**

**Search**($x, k$)

1. if $x = \text{NULL}$ OR $k = x\.key$
2. return $x$
3. if $k < x\.key$
4. return **Search**($x\.left, k$)
5. else
6. return **Search**($x\.right, k$)

Time complexity: $\Theta(h)$ where $h$ is the height of the tree.
Iterative search

\textbf{SEARCH}(x, k)

(1) \textbf{while} x \neq \text{NULL} \text{ AND } k \neq x.\text{key}
(2) \textbf{if} \; k < x.\text{key}
(3) \quad x \leftarrow x.\text{left}
(4) \textbf{else}
(5) \quad x \leftarrow x.\text{right}
(6) \textbf{return} \; x

Time complexity: $\Theta(h)$. 
**Minimum(x)**

1. while $x$.left $\neq$ NULL
2. $x \leftarrow x$.left
3. return $x$

**Maximum(x)**

1. while $x$.right $\neq$ NULL
2. $x \leftarrow x$.right
3. return $x$

Time complexity: $\Theta(h)$. 

Binary search trees (BST)
Successor

To find the successor of $x$:

- If $x$ has a right child, the successor is the node with minimum key in the subtree of the right child.
- Otherwise, the successor is the lowest ancestor of $x$ whose left child is also an ancestor of $x$ (if no such ancestor exists, the successor is NULL).

Time complexity: $\Theta(h)$.

**Successor**($x$)

1. if $x$.right $\neq$ NULL
2. return Minimum($x$.right)
3. $y \leftarrow x$.p
4. while $y \neq$ NULL AND $x = y$.right
5. $x \leftarrow y$
6. $y \leftarrow y$.p
7. return $y$
To find the successor of \( x \):

- If \( x \) has a right child, the successor is the node with minimum key in the subtree of the right child.
- Otherwise, the successor is the lowest ancestor of \( x \) whose left child is also an ancestor of \( x \) (if no such ancestor exists, the successor is NULL).

Time complexity: \( \Theta(h) \).

\[
\text{Successor}(x) \\
(1) \text{ if } x.\text{right} \neq \text{NULL} \\
(2) \text{ return } \text{Minimum}(x.\text{right}) \\
(3) y \leftarrow x.\text{p} \\
(4) \text{ while } y \neq \text{NULL} \text{ AND } x = y.\text{right} \\
(5) x \leftarrow y \\
(6) y \leftarrow y.\text{p} \\
(7) \text{ return } y
\]
When inserting a new element $x$ to the tree, it is inserted as a leaf.

The location of the new leaf is chosen such that the binary search tree property is maintained.

This is done by going down from the root using the same algorithm as in the search procedure.

We need to keep a “trailing pointer”, namely a pointer that points to the previous node visited by the algorithm.

Time complexity: $\Theta(h)$. 
**Insertion**

**INSERT(T, z)**

\[
y \leftarrow \text{NULL} \\
x \leftarrow T.\text{root} \\
\text{while } x \neq \text{NULL} \\
\quad y \leftarrow x \\
\quad \text{if } z.\text{key} < x.\text{key} \\
\quad\quad x \leftarrow x.\text{left} \\
\quad \text{else} \\
\quad\quad x \leftarrow x.\text{right} \\
\]

\[
z.\text{p} \leftarrow y \\
\text{if } y = \text{NULL} \\
\quad T.\text{root} \leftarrow z \quad \text{// } T \text{ was empty} \\
\text{else if } z.\text{key} < y.\text{key} \\
\quad y.\text{left} \leftarrow z \\
\text{else} \\
\quad y.\text{right} \leftarrow z
\]
**Insertion**

**Insert** \((T, z)\)

\[
y \leftarrow \text{NULL} \\
x \leftarrow T.\text{root} \\\	ext{while } x \neq \text{NULL} \\
\quad y \leftarrow x \\
\quad \text{if } z.\text{key} < x.\text{key} \\
\quad \quad x \leftarrow x.\text{left} \\
\quad \text{else} \\
\quad \quad x \leftarrow x.\text{right} \\
\]

\[
z.\text{p} \leftarrow y \\
\text{if } y = \text{NULL} \\
\quad T.\text{root} \leftarrow z \quad \text{// } T \text{ was empty} \\
\text{else if } z.\text{key} < y.\text{key} \\
\quad y.\text{left} \leftarrow z \\
\text{else} \\
\quad y.\text{right} \leftarrow z
\]
**Insertion**

**Insert** \((T, z)\)

\[
y \leftarrow \text{NULL} \\
x \leftarrow T.\text{root} \\
\text{while } x \neq \text{NULL} \\
\quad y \leftarrow x \\
\quad \text{if } z.\text{key} < x.\text{key} \\
\quad \quad x \leftarrow x.\text{left} \\
\quad \text{else} \\
\quad \quad x \leftarrow x.\text{right} \\
z.\text{p} \leftarrow y \\
\text{if } y = \text{NULL} \\
\quad T.\text{root} \leftarrow z \quad // T \text{ was empty} \\
\text{else if } z.\text{key} < y.\text{key} \\
\quad y.\text{left} \leftarrow z \\
\text{else} \\
\quad y.\text{right} \leftarrow z
\]
**Insertion**

**INSERT**\( (T, z) \)

\[
y \leftarrow \text{NULL}
\]

\[
x \leftarrow T.\text{root}
\]

**while** \( x \neq \text{NULL} \)

\[
y \leftarrow x
\]

**if** \( z.\text{key} < x.\text{key} \)

\[
x \leftarrow x.\text{left}
\]

**else**

\[
x \leftarrow x.\text{right}
\]

\[
z.\text{p} \leftarrow y
\]

**if** \( y = \text{NULL} \)

\[
T.\text{root} \leftarrow z \quad // \ T \text{ was empty}
\]

**else if** \( z.\text{key} < y.\text{key} \)

\[
y.\text{left} \leftarrow z
\]

**else**

\[
y.\text{right} \leftarrow z
\]
Insertion

**Insert** \((T, z)\)

\[
y \leftarrow \text{NULL} \\
x \leftarrow T.\text{root} \\
\text{while } x \neq \text{NULL} \\
\quad y \leftarrow x \\
\quad \text{if } z.\text{key} < x.\text{key} \\
\quad \quad x \leftarrow x.\text{left} \\
\quad \text{else} \\
\quad \quad x \leftarrow x.\text{right} \\
z.\text{p} \leftarrow y \\
\text{if } y = \text{NULL} \\
\quad T.\text{root} \leftarrow z \quad /\!/ T \text{ was empty} \\
\text{else if } z.\text{key} < y.\text{key} \\
\quad y.\text{left} \leftarrow z \\
\text{else} \\
\quad y.\text{right} \leftarrow z
\]
To delete a node $z$ in a binary search tree, there are 3 cases to consider:

- **Case 1** $z$ has no children.
- **Case 2** $z$ has 1 child.
- **Case 3** $z$ has 2 children.

Time complexity: $\Theta(h)$. 
If $z$ is a leaf, delete it from the tree.
Case 2

If $z$ has a single child $x$, delete $z$ from the tree, and make $x$ a child of the parent of $z$. 

![Diagram of a binary search tree with a node labeled 'z' being deleted and an edge from 'z' to its parent being redrawn to connect to the child of 'z'.]
Case 3

If $z$ has two children:

- $y \leftarrow \text{SUCCESSOR}(z)$
- Delete $y$ from the tree (using Case 1 or 2).
- Make $y$ take the place of $z$ in the tree.
All the dynamic set operation take $\Theta(h)$ time (worst case) on a tree of height $h$.

In the worst case, $h = n - 1$.

In the best case, $h = \lceil \log n \rceil$ (in an optimal tree, level $i$ of the tree contains exactly $2^i$ nodes, except perhaps the last level).
A **randomly built binary search tree** is a tree that is built by inserting the elements in random order into an initially empty tree (each of $n!$ permutations of the elements is equally likely).

**Theorem**

*The expected height of a randomly built binary search tree is $\Theta(\log n)$.*

Little is known about the average height of a binary search tree when both insertion and deletion are used to create it.