# 1 Binary Search Trees

Binary search trees are binary trees that satisfies the search tree property that left children are less than or equal to the parent and right children are greater than the parent. Note that because root choice is arbitrary, the height of an unbalanced BST can be anywhere from $\approx \log_2 n \rightarrow \approx n$, which means any algorithms operating on binary search trees have to take into account average-case and worst-case runtimes.

| Operation | Time |
|---|---|
| Select | $O(\log n)$ |
| Search | $O(\log n)$ |
| Min/Max | $O(\log n)$ |
| Pred-/Succ-essor | $O(\log n)$ |
| Rank | $O(\log n)$ |
| Output | $O(n)$ |
| Insert | $O(\log n)$ |
| Delete | $O(\log n)$ |

# 2 Operations

**Minimum** Traverse the tree going left every time until a leaf node.

**Maximum** Traverse the tree going right every time until a leaf node.

**Predecessor** Called on node `k`, if `k` has a successor, return `max(left)`. Otherwise, follow parent pointers until reaching a node where `key <= cur`.

**Successor** Called on node `k`, if `k` has a successor, return `min(right)`. Otherwise, follow parent pointers until reaching a node where `key > cur`.

**In-Order Traversal** Recursively, at the parameter node, recurse on the left subtree, append the current node, then recurse on the right subtree.

**Selection and Rank** To select the $i^{th}$ order statistic of a BST, first augment each node with a `size` field for the size of the subtree rooted at the node. The size of a node will be equal to one plus the sum of its child sizes.

```
def select(x = root, rank):
    a = x.left.size
    if a >= i: select(x.left, rank)
    if a < i - 1: select(x.right, i - a - 1)
    if a == i - 1: return x
```

**Deletion** Deleting a node can go three different ways.

```
def delete(key):
    k = find(key)
    if k.left is NULL and k.right is NULL:
        remove(k)
    else if k.left is NULL xor k.right is NULL:
        k = child
        remove(old_k)
    else:
        l = predecessor(k)
        swap(k, l)
        remove(k)
```