7.1 Binary Search Trees

An (internal) binary search tree stores the elements in a binary tree. Each tree-node corresponds to an element. All elements in the left sub-tree of a node \( v \) have a smaller key-value than \( \text{key}[v] \) and elements in the right sub-tree have a larger-key value. We assume that all key-values are different.

(External Search Trees store objects only at leaf-vertices)

Examples:

```
6
2 7
1 5 8
1
2
5
6
7
8
```

Algorithm 5 TreeSearch \((x, k)\)

1: if \( x = \text{null} \) or \( k = \text{key}[x] \) return \( x \)
2: if \( k < \text{key}[x] \) return TreeSearch(left\( [x] \), \( k \))
3: else return TreeSearch(right\( [x] \), \( k \))
### Binary Search Trees: Minimum

**Algorithm 6 TreeMin(x)**

1. if \( x = \text{null} \) or \( \text{left}[x] = \text{null} \) return \( x \)
2. return TreeMin(left[\( x \)])

### Binary Search Trees: Successor

**Algorithm 7 TreeSucc(x)**

1. if \( \text{right}[x] \neq \text{null} \) return TreeMin(right[\( x \)])
2. \( y \leftarrow \text{parent}[x] \)
3. while \( y \neq \text{null} \) and \( x = \text{right}[y] \) do
   1. \( x \leftarrow y; y \leftarrow \text{parent}[x] \)
4. return \( y; \)

### Binary Search Trees: Insert

**Algorithm 8 TreeInsert(x, z)**

1. if \( x = \text{null} \) then
2. \( \text{root}[T] \leftarrow z; \text{parent}[z] \leftarrow \text{null} \);
3. return;
4. if \( \text{key}[x] > \text{key}[z] \) then
   1. if \( \text{left}[x] = \text{null} \) then
   2. \( \text{left}[x] \leftarrow z; \text{parent}[z] \leftarrow x \);
   3. else TreeInsert(left[\( x \)], z);
5. else
   1. if \( \text{key}[x] < \text{key}[z] \) then
   2. if \( \text{right}[x] = \text{null} \) then
   3. \( \text{right}[x] \leftarrow z; \text{parent}[z] \leftarrow x \);
   4. else TreeInsert(right[\( x \)], z);
5. else
   6. if \( \text{left}[x] = \text{null} \) then
   7. \( \text{left}[x] \leftarrow z; \text{parent}[z] \leftarrow x \);
   8. else TreeInsert(right[\( x \)], z);
Binary Search Trees: Delete

Case 1:
Element does not have any children
- Simply go to the parent and set the corresponding pointer to null.

Case 2:
Element has exactly one child
- Splice the element out of the tree by connecting its parent to its successor.

Case 3:
Element has two children
- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Algorithm 9 TreeDelete(z)
1: if left[z] = null or right[z] = null
2: then y ← z else y ← TreeSucc(z);
3: if left[y] ≠ null
4: then x ← left[y] else x ← right[y]; x is child of y (or null)
5: if x ≠ null then parent[x] ← parent[y]; parent[x] is correct
6: if parent[y] = null then
7: root[T] ← x
8: else
9: if y = left[parent[y]] then
10: left[parent[y]] ← x
11: else
12: right[parent[y]] ← x
13: if y ≠ z then copy y-data to z
Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $O(h)$, where $h$ denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $O(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps
similar: SPLAY trees.

---

**Binary Search Trees (BSTs)**

**Bibliography**


Binary search trees can be found in every standard text book. For example Chapter 7.1 in [MS08] and Chapter 12 in [CLRS90].