DICTIONARY
(Notes by Glenn Mertens)
**A.D.T. DICTIONARY**

**Object:** A set of elements that can be ordered.

**Operations:** `MAKENULL` `INSERT` `DELETE` `SEARCH` `{basic operations}`

`MAXIMUM`, `MINIMUM`, `SUCCESSOR`, `PREDECESSOR` `{browsing operations}`

**SORT**

**Implementations:**
- Binary Search Tree (BST)
- Balanced Search Trees (e.g., red-black tree)
- B-Trees
- Hashing
- Tries / Suffix Trees
BINARY SEARCH TREE

A binary search tree is a data structure that stores a set of `n` items in a tree whose nodes have two children, a left and a right child. Each node in the tree contains a `key` value along with a pointer to its left and right children.

The key property of a binary search tree is that for any node `u` in the tree, all keys in the left subtree of `u` are less than `key[u]`, and all keys in the right subtree of `u` are greater than `key[u].` This property holds for all nodes in the tree.

The search tree & property

Ghiyas k£6.
IN ORDER TRAVERSAL

Visits all nodes in order, using $O$ element comparisons.

The entropy scale:

- $m \log m$
- $m$
- unordered list
- Binary heap
- Effort needed ($= \# \text{uses of oracle}$)
- order:
  - sorted list,
  - BST
BASIC OPERATIONS

(TREE) SEARCH \((t, \triangleright)\)

\[
2 \leftarrow t \\
\text{while } (x \neq \text{nil}) \text{ and } (\text{key}[x] \neq k) \text{ do :} \\
\quad \text{if } k < \text{key}[x] \text{ then } x \leftarrow \text{left}[x] \\
\quad \text{else } x \leftarrow \text{right}[x] \\
\text{return } x
\]

Time = \(O(1) \text{ (Height of Tree)}\)

Worst-case
\[
\text{INSERT } (t, x) \text{ key to be inserted}
\]
\[
(a, y) \leftarrow (t, \text{nil})
\]
\[
\text{while } a \neq \text{nil} \text{ do :}
\]
\[
y \leftarrow a
\]
\[
\text{if } k < \text{key}(x) \text{ then } a \leftarrow \text{left}(x)
\]
\[
\text{else } a \leftarrow \text{right}(x)
\]

\[
\text{make new cell } x
\]
\[
\text{key } [x] \leftarrow k
\]
\[
\text{parent } [x] \leftarrow y
\]
\[
\text{if } k < \text{key } [y] \text{ then } \text{left } [y] \leftarrow x
\]
\[
\text{else } \text{right } [y] \leftarrow x
\]
\[
\text{left } [x] \leftarrow \text{nil}
\]
\[
\text{right } [x] \leftarrow \text{nil}
\]

\[
\text{creates : }
\]
DELETE(α)

\[ |L| = 0 \text{ or } |R| = 0 \]

\[ \Rightarrow \]

Else let \( l = \text{maximum key}[α] \)

\[ \Rightarrow \]

DELETE(\( l \))

(on, equivalently do this with \( x \))
TREE(SUCCESSOR(x))

If right(x) ≠ nil
Then return minimum(right(x))
else y = parent(x)
    While y ≠ nil and x = right(y):
        x = y
        y = parent(x)
    return y

Time = O(Height of tree)
Browsing: Repeat successor $k$ times starting at $x$.

Towards root

Least common ancestor (LCA)

There are $k-1$ keys between $x$ and $y$.

Path followed by successor
COMPLEXITY CONSISTS OF least common ancestor
and several subtree visits

TOTAL COMPLEXITY ≤ 2 * Height + 2 * (k-1)

So, browsing the entire tree from min to max takes time O(n).
BRANDS OF SEARCH TREES

• ORDINARY (no guarantee about height)

• STATIC SEARCH TREES (no INSERT DELETE):
  optimal binary search trees, sometimes complete binary trees

• RANDOM BINARY SEARCH TREES (trecps, etc.):
  Height $O(\log n)$ in an expected sense

• BALANCED BINARY SEARCH TREES: Height $O(\log n)$

Examples: Red-Black trees (2-3-4 trees)
            AVL trees
            HB-k trees
            &-3 trees
            B trees
            etc...

• SPLAY TREES, WEIGHT-BALANCED TREES: Amortized $O(\log n)$ time per operation.