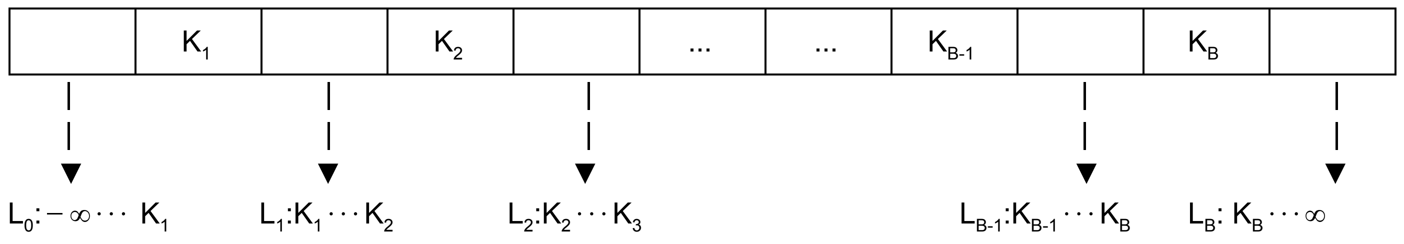
Splay trees: a self-adjusting BST with a new operation called "splay". splay(t) rearranges the tree with rotations such that t is at the root of the tree. Splay trees do not store any additional information at the nodes to maintain balance information. All operations start with applying the corresponding BST algorithm. The operations contains, get, min, max, and add apply splay(t) on the node t at which the operation ended. If the remove operation failed because x is not in the tree, it applies splay(t) where t is the node at which find(x) stopped (i.e., the node at which it was discovered that x is not in the tree). If x was in the tree, then the remove operation bypassed some node t. Let p_t be the parent node of that node before the remove operation. The algorithm calls splay(p_t) before returning the removed element.

```
splay( t ):
    while t is not the root of the tree do
        if t is left (or right) child of root then  // Zig
            Perform a single rotation [R] (or [L]) at root to bring t to the root of the tree
        else if g_t → p_t → t is LL (or RR) then  // Zig-Zig
            Perform a double rotation [RR] (or [LL]) at g_t
        else  // Zig-Zag
            Perform a double rotation [LR] or [RL] at g_t due to t
```

B-trees: multi-way branching trees used for implementing dictionary operations. Multi-way branching does not provide any advantage over binary trees when the entire data resides in memory. B-trees are used in external dictionaries (databases), where indexing is used to allow fast access to data. In these applications, the actual data resides on the disk, in "main" files. Blocks of the main file are the leaf nodes of the B-tree. Internal nodes of the tree, known as index blocks or directory entries, are stored in index files. The branching factor ("B" in the B-tree) is chosen based on how many directory entries can fit into one block of the disk. The tree is organized so that every path from its root to any of its leaves has the same length. Internal nodes of the tree are kept at least 50% full. When a block overflows during an add operation, it is split into 2 blocks that are half full. When a block becomes less than half full after a remove operation, entries are borrowed from adjacent blocks (balancing) or blocks are merged into fewer blocks.



Search starts at the root of the tree. An index block is fetched from the disk, and a binary search on the keys stored in that block is used to find a value of i such that $K_i \leq x \leq K_{i+1}$. The link L_i is the reference of the next index block, which directs us towards the part of the main file records with all keys between K_i and K_{i+1}. The link from the last level index refers to a block of the main file. If a main file occupies N blocks of the disk, then a B-tree index accesses $O(\log_B N)$ blocks of the disk for each operation. In external dictionary applications, disk access takes many orders of magnitude more time than computation, and therefore, external algorithms are evaluated based on the number of disk accesses, rather than the number of operations. B-trees improve the running times of operations from $O(\log_2 N)$ to $O(\log_B N)$ in sorted files, and from $O(N)$ to $O(\log_B N)$ in unsorted files. The choice of B depends on the number of bits needed for keys, size of block pointers, and the block size of the disk. In practice, it ranges from about 50-1000, giving a speed up of about 10-20 in sorted files and millions in unsorted files.