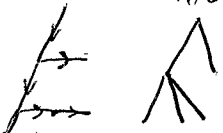
Binary Search Trees:

We see implicit representation of trees for heaps.

For explicit representation all nodes have m+1 pointers, one to the parent and m to the children where m is the maximal number of children in a tree. Alternatively, if m is unknown, we have two pointers one to the first child second to the next sibling (a linked-list or doubly-linked list)



Binary search trees is another way to implement dictionary operations without randomization and it is comparison-based (the bounds are the worst-case bounds)

Dictionary operations
- search(x): find x or determine x is not there (all elements are distinct for now)
- insert(x): unless it is already there, insert x.
- delete(x): delete x if it is there.

Binary search trees also implement more complicated operations efficiently.

We use explicit representation for binary trees by keeping a record containing at least three fields: key, left, and right(child). Due to explicit representation any node may be added or removed (unlike heaps that we only remove or insert leaves)

Each key in the binary search tree serves to divide the range of the keys below it. The keys in the left subtree are all smaller than it and the keys in the right subtree are all greater than it. We say a tree is consistent if all keys satisfy this condition.

Search operation: we first compare x against the root of the tree, whose value is say r. If $x=r$, then we are done. If $x<r$, we continue from the left child; otherwise we continue from the right child.

Insertion operation: first we search for x and abort insert if x is already in the tree. Otherwise, the search ends at a leaf and the new key can be then inserted below that leaf depending on the value of x (at left or right). The tree remains consistent.

Deletion operation: Deletion is more complicated. It is easy if it is a leaf (by making it nil). Even if the node has one child, it is easy by changing the child pointer of the parent to the child of x (instead of x).

What about if the node x has two children?

First we find the predecessor of x in the tree which is a node at least as large as all the keys in the left subtree of x and smaller than all the keys in the right subtree of x. To find it first we go the left child of x and then follow the right child as long as we can. When we end by finding p, since it has only at most one left child, we will delete it easily and replace x with p. The consistency is preserved since p is the largest element of the left subtree of x (i.e. the tree rooted at the left child of x).

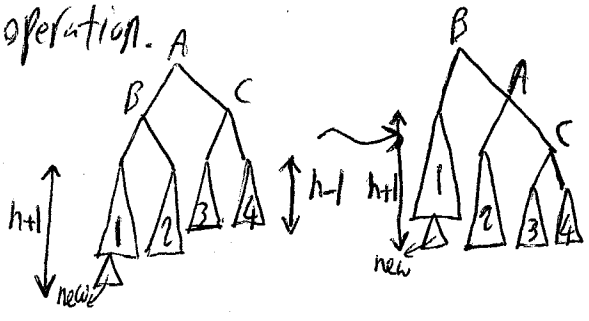
Time complexity: all running times depend on the shape of the tree and the location of the relevant node; in the worst case it is $O(\text{height}(T))$. If the tree is balanced, i.e. $\text{height}(T) = O(\log n)$, where n is the number of elements, all operations are in $O(\log n)$ and thus efficient. If the keys are inserted in a random order, one can prove the expected height of the tree is $O(\log n)$ (like depth of Quicksort) and thus dictionary operation are efficient in the average case. However if the insertions e.g. are in a sorted, or close to sorted, order, the tree would be a linked list (a long path) and the operations can take $\Omega(n)$ thus.

To prevent this worst case we can use:

AVL trees: Binary search trees such that for every node, the difference between the heights of its left and right subtrees is at most one (the height of an empty tree is defined as 0).

Thm: The height h of an AVL tree is $O(\log n)$.

The idea of AVL trees is that whenever the property above is violated, we rebalance the tree we perform a rotation operation. See details in the book.



All operations in AVL trees is O(height) = O(log n). Empirical studies have shown the average search time to be approximately $\log_2 n + 0.25$ comparisons. Note that we can delete_max() or delete_min() operations in AVL trees by following right or left childrens until we reach a leaf. Again the running time is O(height) = O(log n). Now we can do a sort using binary search trees or AVL trees as follows.

1. first insert all keys in the tree in the order of input.
2. do n delete_min() and insert the returned elements in the output array.

Both steps and thus the whole sort is in O(n log n) and so it is efficient. The sort here is a comparison-based sort, and is **not** an in-place sort.