Reliability, usability, efficiency, extensibility, maintainability, correctness and completeness. Depth of k is the length of the unique path from the root to k. Height of k is the length of the longest path from k to a leaf. Tree Traversals: Preorder (Rt, L, R), Inorder (L, Rt, R), Postorder (L, R, Rt). Balanced BT(the height diff. of L&R subtrees < 2), Full BT (every node has its L&R subtrees of the same height), Complete BT (all filled, bottom level filled left to right). BST - avg. depth over all nodes is O(log n). All operations on BST take O(log n) time on avg. if there are no deletions and insertions are done randomly. B-BST - all operations take O(log n) in all cases. B-BST are made: by recreating trees (glob. app.), maintaining the balance condition (loc. app.) AVL trees, self adjusting (glob. app.). B-BST, recreating, (right rotation, left rotation code). DSW has 2 steps: BST to Backbone (creating a backbone code) which requires at most n -1 rotations, O(n), Backbone to B-BST (create balanced).
AVL tree = a BST with AVL pity = a B-BST, depth of an AVL is O(log n), node k is an imbalance node if h. diff of L&R subtrees >=2)

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$
$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$
$$\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}.$$

CreateBackBone(root, n) { --temp = root; --while(temp != null) --if (temp.left != null) --//has a left child --right-rotate temp.left about temp --temp = temp.left; --else -- temp = temp.right;}
CreateBalanced(n) { m = 2floor(log2(n+1)) - 1; --//RHS is # of nodes in the closest full BT --make (n-m) left rots from the top of backbone --//treat overflowing nodes separately --while (m > 1) { m = m/2; --make m left rots from the top of backbone --//each pass decreases the size of backbone --//by one-half --} }

node AVLInsert(data x, node t) { if (t==null) //base case --return new node(x, null, null); else if (x<t.elt) --{ t.left=AVLInsert(x, t.left); --if (height(t.left)-height(t.right)==2) --if (x<t.left.elt) //left-left --t=singleRight(t); --else //left-right --t=doubleRight(t); --else if (x>t.elt) --NOTE: need to keep "height" info for each node --else if (x>t.elt) --{ t.right=AVLInsert(x, t.right); --if (height(t.right)-height(t.left)==2) --if (x>t.right.elt) //right-right --t=singleLeft(t); --else //right-left --t=doubleLeft(t);}--else; //duplicate; do nothing --t.height=max(height(t.left),height(t.right))+1; --return t; }

RotateRight(G, P, C) { if (P != root) // G != null G.right = C; --P.left = C.right; --C.right = P; }
RotateRight(G, P, C) { if (P != root) // G != null G.right = C; --P.right = C.left; --C.left = P; }

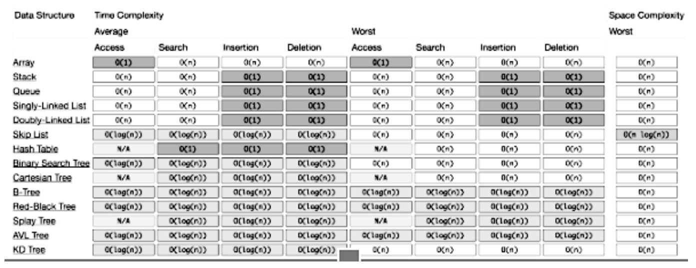
collision - when a hash fctn. maps more than one key into the same index. perfect hash function - 1-1 mapping. A table size is prime so collisions and clustering are minimized. Resolving collisions: 1. restructuring the hash table by; using buckets T[i] is an array and by separate chaining (an array of linked lists) (arrays and linked lists are secondary data structures. (code for find insert and delete) 2. Open Addressing (no secondary data structure), use a hash function to generate a probing sequence. Hash function h(x), where x is the key is h_i (x) = (hash (x) + f(i)) mode TS (%TS), f is a collision resolution strategy. The probing sequence 1. is to be produced efficiently 2. needs to be reproducible. Some basic operations, 1. deletion, 2. retrieve, each entry has three states, " occupied", "empty" or "deletion", 3. insertion when state is empty or deleted
Problem with linear probing: primary clustering. It can be shown that if TS is prime, and the table is at least half empty then we can always insert a new item into the table. Problem with quadratic probing: secondary clustering. insert 7597 into TS 101.7597mod101 = 22, insert 4567 into TS 101.4567mod101= 22 + 1^2 = 23 etc.

```
/*function to insert element in binary tree */
static void insert(Node temp, int key)
{
    Queue<Node> q = new LinkedList<Node>();
    q.add(temp);

    // Do level order traversal until we find
    // an empty place.
    while (!q.isEmpty()) {
        temp = q.peek();
        q.remove();

        if (temp.left == null) {
            temp.left = new Node(key);
            break;
        } else
            q.add(temp.left);

        if (temp.right == null) {
            temp.right = new Node(key);
            break;
        } else
            q.add(temp.right);
    }
}
```



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)