$T(n) = T(n-1) + O(1) = O(n)$
$T(n) = T(n-1) + O(n) = O(n^2)$
$T(n) = T(n/2) + O(1) = O(\log n)$ (Binary search)
$T(n) = T(n/2) + O(n) = O(n)$ (Quickselect)
$T(n) = kT(n/k) + O(1) = O(n)$
$T(n) = kT(n/k) + O(n) = O(n\log n)$, (Quicksort)
$T(n) = T(n-1) + T(n-2) = O(2^n)$ (Fibonacci)
$T(n) = T(n-1) + O(nk) = O(nk+1)$
$T(n) = T(n-1) + O(\log n) = O(n\log n)$ (Stirling Approx)
$T(n) = 2T(n-1) + 1 = O(2^n)$
$T(n) = 2T(n/4) + O(1) = O(n^{0.5})$
$n^b > (\log n)^a$ always
$n^b < a^m$ always
$O(n^{9.9}) < O(1.01^n)$

Binary search

```
int search(A, key, n)
begin = 0
end = n-1
while begin < end do:
mid = begin + (end-begin)/2;
if key <= A[mid] then
 end = mid
else begin = mid+1
return (A[begin]==key) ? begin : -1
Invariant: A[begin] <= key <= A[end]
if key is in A.
T(n) = T(n/2) + O(1) = O(logn)
```

Peak finding

```
FindPeak(A, n)
if A[n/2] is a peak then return n/2
else if A[n/2+1] > A[n/2] then
Search for peak in right half.
else if A[n/2–1] > A[n/2] then
Search for peak in left half.
Invariant:
1. If we recurse in the right half,
then there exists a peak in the right half.
2. If we recurse in the right half,
then every peak in the right half is a peak
         in the array.
T(n) = T(n/2) + O(1) = O(logn)
```

Sorting Algorithms	Time Complexities			Input Types			Stable	Advantages	Disadvantages	Invariant / Sorting Jumble Hax
	Best Case	Worst Case	Average Case	Best Case	Worst Case	Average Case				
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Already sorted/ almost sorted	Reverse sorted	Input chosen in random	Yes	Fast for almost sorted	Slow average case	After j iterations, first j elements sorted (not final), back part untouched
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	-	-	-	No	-	Slow for almost sorted array	After j iterations, first j elements correctly sorted in their final positions, original element swapped away
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Already sorted	Reverse sorted	Input chosen in random	Yes	Fast for almost sorted	Significantly slower for most inputs	After j iterations, last j elements correctly sorted in their final positions
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	-	-	-	Yes	Fast on average	Slow for small n and sorted array Extra space required	After k iterations, the first k elements in the final array are sorted, each pair of elements are correctly sorted.
Quick Sort	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	Random pivot	Every pivot is the smallest/biggest element	Random pivot	No	Optimizable, many partition and pivot choosing algorithms	Worst case is $O(n^2)$	After the partition step, every: arr[i] < pivot, where i < low arr[j] < pivot, where j > high Pivots always sorted

Quicksort Partition Algo O(n):

```
partition(A[0..n], n, pIndex) // Assume no duplicates, n>1
    pivot = A[pIndex]; // pIndex is the index of pivot
    swap(A[0], A[pIndex]); // store pivot in A[0]
    low = 1; // start after pivot in A[1]
    high = n+1; // Define: A[n+1] = inf
    while (low < high)
        while (A[low] < pivot) and (low < high) do low++;
        while (A[high] > pivot) and (low < high) do high– – ;
    if (low < high) then swap(A[low], A[high]);
    swap(A[1], A[low–1]); //swap the pivot back in
    return low–1;
```

To deal with duplicates, 3-way partition algo.

Things to note:

- BST may not be balanced
- Worst case runtime for Query Operations is O(n)
- For deletion of node v:
  - If no children, just delete v
  - If 1 child,
    - delete v + connect v.child to v.parent
  - If 2 children,
    - x = successor(v)
    - delete(x)
    - delete(v)
    - connect x to v.left, v.right, v.parent

Quickselect kth smallest element

```
Select(A[1..n], n, k)
if (n == 1) then return A[1];
else choose random pivot index pIndex.
  p = partition(A[1..n], n, pIndex)
if (k == p) then return A[p];
else if (k < p) then
return Select(A[1..p–1], k)
else if (k > p)
then return Select(A[p+1], k – p)
Invariant: The pivots will be in their
final positions.
```

BST

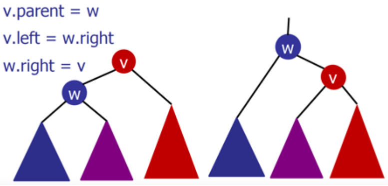
```
public TreeNode successor(){
    if (rightTree != null)
        return rightTree.searchMin();
    TreeNode parent = parentTree;
    TreeNode child = this;
    while ((parent != null) && (child == parent.rightTree))
        child = parent;
    parent = child.parentTree;
    }
    return parent;
}
```

Choice of Pivot:

- Pivot is good when it divides the array to 2 pieces, each size at least 1/10
- Median/First/Last: Worst case is the same $O(n^2)$
- Paranoid Quicksort:
  - Repeat the partition step until p > n/10 and p <9n/10. Expected runtime is O(nlogn) with high probability.

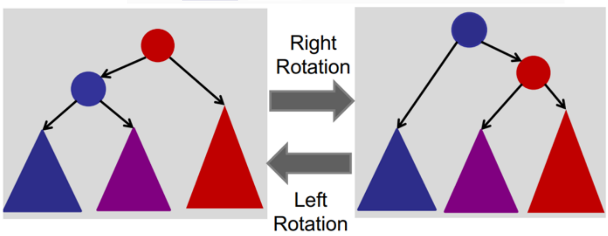
AVL trees: Height balanced if every node is height balanced
|v.left.height – v.right.height| <= 1
and has at most h < 2log(n), and n > 2^h/2.
Insertion:
– Insert key in BST.
– Walk up tree:
• At every step, check for balance.
• If out-of-balance, use rotations to rebalance.

```
right-rotate(v)        // assume v has left != null
    w = v.left
    w.parent = v.parent
    v.parent = w
    v.left = w.right
    w.right = v
```



If v is out of balance and left heavy:
1. v.left is balanced: right-rotate(v)
2. v.left is left-heavy: right-rotate(v)
3. v.left is right-heavy: left-rotate(v.left), right-rotate(v)

delete(v)
1. If v has two children, swap it with its successor.
2. Delete node v from binary tree (and reconnect children).
3. For every ancestor of the deleted node:
– Check if it is height-balanced.
– If not, perform a rotation.
– Continue to the root.
Deletion may take up to O(log(n)) rotations.



If v is out of balance and right heavy:
1. v.right is balanced: left-rotate(v)
2. v.right is right-heavy: left-rotate(v)
3. v.right is left-heavy: right-rotate(v.right), left-rotate(v)

Trie vs AVL:
Time:
Trie tends to be faster: O(L) vs O(Lh).
Does not depend on size of total text.
Does not depend on number of strings.
Space:
Trie tends to use more space.
BST and Trie use O(text size) space.
But Trie has more nodes and more overhead.

Augmenting Data Structures
Most of the time, we will make use of existing Data structures, but store additional Data so that it works as we intend.
E.g. weight, rank, max, min
1. Maintain the Data Structure
(BST, array, linked list etc)
2. Modifying Operations
(Insert, delete)
3. Query Operations
Search, Select
Important!!
Make sure the additional data is updated during Insertion/Deletion.

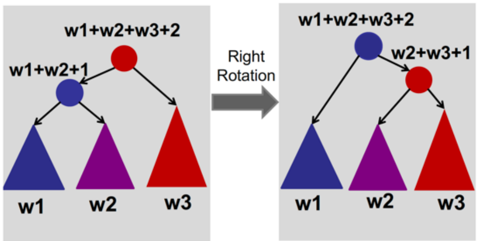
Order statistics in BST:
Idea: Store the size of the subtree in each node
Select(k): select node of rank(k):
Time Complexity: O(logn)

```
select(k)
    rank = m_left.weight + 1;
    if (k == rank) then
        return v;
    else if (k < rank) then
        return m_left.select(k);
    else if (k > rank) then
        return m_right.select(k–rank);
```

Rank(v): return the rank of node v
Time Complexity: O(logn)

```
rank(node)
    rank = node.left.weight + 1;
    while (node != null) do
        if node is left child then
            do nothing
    else if node is right child then
            rank += node.parent.left.weight + 1;
    node = node.parent;
    return rank;
```

Maintain weight during rotations:



Node type	#Keys		#Children	
	Min	Max	Min	Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

More Algorithm Analysis
Sum of AP:
$d = Common\ Difference$
$a = first\ term, then\ a + (n-1) \times d\ is\ the\ last\ term$
$$Sn = \frac{n}{2}(2a + (n-1) \times d) = \frac{n}{2}(first\ term + last\ term)$$

Sum of GP:
$r = Common\ Ratio$
$a = first\ term$
$$Sn = \frac{a(r^n - 1)}{r - 1} = \frac{a(1 - r^n)}{1 - r}$$

If loop index x= or /=, there will be $\log_2 n$ number of steps
$$T(n) = kT\left(\frac{n}{k}\right) + O(n^2) = O(n^2)$$
$$T(n) = kT\left(\frac{n}{k}\right) + O(\sqrt{n}) = O(n)$$
$$T(n) = kT\left(\frac{n}{k}\right) + O(nlogn) = O(nlogn^2)$$
$$T(n) = kT\left(\frac{n}{k}\right) + O(logn) = O(logn^2)$$

(a, b)-trees
B-trees (special type of (a,b)-trees where a=B and b = 2B)
Search: check keylist. If key is not in keylist, need to look inside the relevant subtree.
If found, return node.
- An (a,b)-B-tree with n **nodes** has $O(\log_a n)$ height
- Binary search for a key at every node takes $O(\log_2 b)$ time
- Hence total search cost: $O(\log_2 b \cdot \log_a n)$ (Cost of searching a node × Height)
= $O(\log_a n)$ since $\log_2 b$ is a constant = O(logn)

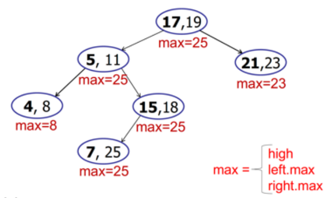
Insertion: Traverse to the relevant node, then insert into keylist.
Rule 1: (a,b) child policy.
Rule 2: Internal nodes have 1 more child than its number of keys
Rule 3: All leaves have the same depth.
If keylist has > b-1 keys, perform split
1. Choose median key from keylist
2. Use that to split keylist into 2 halves.
3. Put median key to parent. Left half is left child, right half is right child.
- If parent is also over capacity, continue splitting upwards.
- To split root node, create a new node (the new root) to store the median.

Deletion:
1.Find successor of key to be deleted
2. Replace the key with its successor.
3. If node with less than a-1 keys after deletion, demote separating key from parent and join with sibling

Interval Trees:
Each node is an interval (a,b), sorted by left end-point (a).
Augment the maximum endpoint in subtree rooted at the node.
interval-search(x) : find interval containing x (O (logn))

Augment: maximum endpoint in subtree

```
interval-search(x)
c = root;
while (c != null and x is not in c.interval) do
    if (c.left == null) then
        c = c.right;
    else if (x > c.left.max) then
        c = c.right;
    else c = c.left;
return c.interval;
```



if search goes right, then there is no interval in the left tree (key > max of left subtree)
if search goes left, then there is no interval in the right tree (key < left endpoint in left tree, so confirm < anything in the right tree)
All-Overlaps Algorithm: (O(klogn))
Repeat until no more intervals:
– Search for interval.
–Add to list.
–Delete interval.
Repeat for all intervals on list:
–Add interval back to tree.

Orthogonal Range Searching
Find split node:
Highest node where search includes both left and right subtree
Algorithm:
– v = FindSplit(low, high);
– LeftTraversal(v, low, high);
– RightTraversal(v, low, high);

```
FindSplit(low, high) O(logn)
    v = root;
    done = false;
    while !done {
        if (high <= v.key) then v=v.left;
        else if (low > v.key) then v=v.right;
        else (done = true);
    }
    return v;
LeftTraversal(v, low, high)
    if (low <= v.key) {
        all-leaf-traversal(v.right);
        LeftTraversal(v.left, low, high);
    }
    else {
        LeftTraversal(v.right, low, high);
    }
Right is symmetric
```

Orthogonal Searching Analysis:
Recursion max logn steps
Output all subtree max O(k)
Query Time Complexity = O(k + logn)
Build Tree: O(nlogn)
If just find number of points, augment weight.
Fix rotations?
2D range enquiry: $O(\log^2 n + k)$
Each node in the x-tree has a y-tree, representing nodes in the subtree but keyed in y

Hashing
Symbol table has no easy way to sort: is not comparison based.
Direct Access Table: convert anything in the world into bits. Space required is $n^{\text{no. of bits}}$
Idea of Hashing: map n keys to m (usually less than n) buckets
Hash function: maps key to bucket h : U -> {1…m}
Time Complexity: O(cost of finding h(k) + cost of accessing bucket) Usually O(1)
Collision: when $h(k_1) = h(k_2)$.
Unavoidable: table size < universe size. By pigeonhole principle, collisions are bound to occur

Chaining
Each bucket contains a linked list to store multiple entries: Total space = O(m+n), m = table size, n = linked list size.
Insert: compute h(key) O(cost(h))and add (key, value) to linked list O(1). Total: O(1 + cost(h))
Search: compute h(key) O(cost(h)) and look inside linked list O(n) (worst case). Total: O(n + cost(h))
Simple Uniform Hashing Assumption:
- All keys equally likely to map to every bucket
- Keys are mapped independently

Expect number of entries per bucket = n/m.
Hence, expected search time = O(1 + n/m) = O(1)