Big-O

- $T(n) = O(f(n))$ means that T grows no faster than $f(n)$ $(T(n) \leq k * f(n))$
- $T(n) = \Omega(f(n))$ means that T grows no slower than $f(n)$ $(T(n) \geq k * f(n))$
- $T(n) = \Theta(f(n))$ means that T grows at the same rate as $f(n)$ $(k' * f(n) \leq T(n) \leq k * f(n))$
- $T(n) + S(n) = O(f(n) + g(n))$
- $T(n) * S(n) = O(f(n) * g(n))$
- If the time complexities are added, then only the greatest x related term count
- If adding together time complexities (completing one action after another), take the max of the time complexities
- If doing one operation concurrently with another (for loop in a for loop, etc.), take the product of the complexities
- If there are multiple *if* conditions, then add up the time complexities of the branches
- Given $f(n) = O(g(n))$ and $f(n) = \Omega(h(n))$, then $h(n) = \Theta(g(n))$ **and** $h(n) = O(g(n))$
- $f(n)$ lower-bounded by $h(n)$ and upper-bounded by $g(n)$, thus $h(n) \leq f(n) \leq g(n)$
- $T(n) = 2T(n/4) + O(1) \rightarrow \sqrt{n}$
- $T(n) = T(n/2) + O(1) \equiv O(logn)$
- $T(n) = \sqrt{n}T(\sqrt{n}) + \sqrt{n} \equiv O(n)$
- $T(n) = T(n/2) + O(n) \equiv O(n)$
- $T(n) = 2T(n/2) + O(1) \equiv O(n)$
- $T(n) = T(n-1) + O(1) \equiv O(n)$
- $T(n) = 2T(n/4) + O(n) \rightarrow O(n)$
- $T(n) = T(n/2) + T(\sqrt{n}) + O(n) \equiv O(n)$ $(T(\sqrt{n}) \leq T(n/4)$
- $T(n) = \mathbf{k}T(n/\mathbf{k}) + \mathbf{b} \equiv O(n)$
- $T(n) = 2T(n/2) + O(n) \rightarrow O(nlogn)$ (QuickSort)
- $T(n) = \mathbf{k}T(n/\mathbf{k}) + \mathbf{b}n \equiv O(nlogn)$
- $T(n) = 2T(n/2) + O(n) \equiv O(nlogn)$
- $T(n) = T(n-1) + O(logn) \equiv O(nlogn)$
- $T(n) = 2T(n/2) + O(nlogn) \rightarrow O(nlog^2n)$
- $O(\sqrt{n}\log^2 n) = O(n)$ but $O(n) \neq O(\sqrt{n}\log^2 n)$
- $T(n) = 3T(n/2) + O(n^2) \equiv O(n^2)$
- $T(n) = T(n-1) + O(n) \equiv O(n^2)$
- $T(n) = T(n-1) + O(n^k) \equiv O(n^{k+1})$
- $T(n) = \sum_i^{n-1} T(i) + O(1) \equiv O(2^n)$
- $T(n) = 2T(n-1) + O(1) \equiv O(2^n)$
- $T(n) = T(n-1) + T(n-2) + \mathbf{d} \equiv O(2^n)$
- $T(n) = T(n-1) + T(n-2) + 1 \equiv O(\phi^n)$ (fibo)
- $O(log(n!)) \equiv O(nlogn)$
- $O(n) > O(\log^k n) > O(\log^4 n) > O(\sqrt{n}) > O(\log^3 n) > O(\log^2 n) > O(\sqrt[4]{n}) > O(\log n) > O(\log(\log n))$
- Note: *alogb*: a is amt of work done at each level and b is the number of levels taken to reach base case

```
public static int recur(int n) {
    // O(n^2)
    for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        System.out.println("Hello");
    }
    }
    // if n: even, it will turn odd in
    // next call
    // if n: odd, then it will invoke
    // recur(n - 2) consistently
    // T(n) = O(n^2) + T(n - 2)
    if (n <= 2) {
    return 1;
    } else if (n % 2 == 0) {
    return recur(n + 1);
    } else {
    return recur(n - 2);
    }
}
```

Divide-and-Conquer: Bin Search

- Invariant 1: A[low] $\leq$ key $\leq$ A[high]
- Invariant 2: No. of iterations $= n/2^k = \log(n)$
- Array/Function/Condition must be **monotonic**
- Be wary of off-by-one errors
- For normal binary search, the array must have the guarantee that for all elements at index $> i$, that they must be $\geq$ element at i
- For binary search on an array guaranteeing pairwise order $(A[i] \leq A[i+2])$, still cannot do bin search just by checking if the pairs are increasing or decreasing and recurse from there, since the unaccounted for elements may have the key we want (e.g. $A[i+1]$ is the value we want
- For a modified B-Search that returns when an item is found, if there are dupes, then the original time complexity of finding an item changes from $\Theta(\log n)$ to $\Theta(\log n/m)$, where m is the number of times a key appears
- $[1, 2, 3, x, x, x, ..., 4]$, imagine splitting the array into partitions of m, and we can stop the search when item x is not found within each partition cuz we can guarantee that x will be found in at least one partition of m if at least one occurrence of x is found
- Time complexity is $O(n \log m)$, since m splits, and n mergers
- Must check that $end = mid - 1$ or it might not terminate
- All the fake B-Search algos that iterate on the left and right of the array have time complexities of $O(n)$, and should terminate when $start \geq end$ and check if $A[start] == key$

```
binSearch(A, key) { // O(log n)
  low = 0 (or 1)
  high = A.length - 1 (or searchspace)
  while (low < high) {
    mid = low + (high - low) / 2
    if (condition(mid)) {
      // usually key <= A[mid]
      high = mid
    } else {
      low = mid + 1
    }
  }
  return A[low];
  // or
  return A[low] == key
         ? low
         : -1;
}
```

Peak Finding

- Reduce and Conquer it, start from the middle, then head in direction of increasing object

```
// O(log n), finds just peaks
findPeak(A, n) {
    if A[n/2 + 1] > A[n / 2]
        findPeak(A[n/2 + 1, ..., n],
            n/2)
    else if (A[n/2 - 1] > A[n / 2])
        findPeak(A[1..., n/2 - 1],
            n/2)
    else
        // found peak
        return n/2
}
// O(n) for stiff peaks
```

- Invariant 1: If we recurse in the right half, then there exist a peak in the right half (same for the left)
- Invaraint 2: Every peak in [begin, end] is a peak in $[0, n-1]$
- For stiff peaks, there is no info on where to recurse, think of an array with same elements except one

Sorting

Invariants

- **Note: Algorithms are usually not stable when swapping non-adjacent items**
- If given 2 sorts to sort a list by key and value, use the sort that is not stable first to build an ordering, then use the stable one to maintain it
- To sort a queue with a queue, do something like iterative merge sort (dequeue $i/2$, compare first $i/2$ elements, dequeue and enqueue smaller item into the original list)
- Bubble: The largest k items will be at the correct spot after k iterations (check the ends) (just need to swap things out of order) (unlikely for smallest elements to be at the front)
- Selection: The smallest k items will be in the correct spot after k iterations (check the beginning, and that the order of other array elements are not changed, except for the swaps (should be swapped with the swapped element))
- Insertion: The first k items will be in sorted order (check that the order of the other elements are not changed)
  - Invariant: For all $k < i$, $A[k] \leq A[k+1]$

```
void sort(int[] array) {
    int size = array.length;
    for (int i = 1; i < size; i++) {
    for (int j = 1; j > 0; j--) {
        if (array[j - 1] > array[j]) {
        swap(array, j - 1, j);
        } else {
        break;
        }
    }
    }
}
```

- QuickSort: The pivots will be in the correct sorted position after one iteration, and elements to the left/right of the pivot are smaller/larger
  - Not-inplace
    - Invariant 1: For all $i <$ low: $A[i] <$ pivot
    - Invariant 2: For all $j >$ high: $A[j] >$ pivot
  - Inplace: use 2 pointers, keep track of low and high points, if low $>$ pivot, stop, and if high $<$ pivot stop, then swap; at the end swap pivot with the low index
    - Invariant 1: $A[high] >$ pivot at the end of each loop
    - Invariant 2: For all $i \geq high$, $A[i] >$ pivot
    - Invariant 3: For all $1 < j < low$, $A[j] <$ pivot (1 indexed)
  - Inplace 3 way: Partition normally, then partition the $\leq$ *pivot* side **or** maintain 4 pointers in the array [< pivot | = pivot | wip | > pivot]
    - If $A[i] <$ pivot, swap < pivot with wip
    - If $A[i] ==$ pivot, swap == pivot with wip
    - If $A[i] >$ pivot, swap > pivot with wip
    - Stop when all pointers except < pivot are equal
  - Can optimise by using InsertionSort on small arrays
- MergeSort: Check that subarrays of size that are powers of 2 are in sorted order (and that the positions of the elements don not change drastically)
  - For a merge sort that does a reversal of the subarray and sorts only 0s and 1s, then the total cost of sorting the list is $O(n_1logn$, where n_1 is the number of elements equal to 1 and n is the total number of elements in the list
  - To sort iteratively, sort in groups of powers of 2. For each given size i, copy the first $i/2$ elements into the left array and next $i/2$ elements into the right. Set left and right pointers to the beginning of both arrays and arrayPointer to the original position of the main array. Repeat until there is no element in the new array: Check the left and right pointer elements, place the smaller element into the arrayPointer. Increment the left or right pointer and arrayPointer

```
mergeSort(A, n)
  if (n == 1)
    return;
  else
    X = mergeSort(A[1...n/2], n/2)
    Y = mergeSort(A[n/2 + 1...n], n/2)
    return merge(X, Y, n/2)
```

QuickSelect

- $O(n)$ time: $T(n) = T(n/2) + O(n)$
- Used for finding k order statistics
- Idea is to partition the array and search the correct half
- When recursing on the left, we don't have to change the kth item to find, but if we recurse on the right, we need to update to $k - p$, where p is the index of the pivot

```
quickSelect(A, n, k)
  if (n == 1)
    return A[1]
  else
    pivot = random()
    p = partition(A[1...n], n, pivot)
    if (k == p)
      return A[p]
    else if (k < p)
      return quickSelect(A[1...p-1],
        k)
    else
      return quickSelect(A[p+1...],
        k - p)
```

Trees

- Binary trees are trees with at most 2 children for each node
- Binary Search Trees are binary trees, all left nodes < key < all right nodes
- Complete tree every node has 2 child, except maybe last layer, flush left. Perfect every node 2 child.
- **Height**: No. of edges of the longest path from a node to a leaf, height = max(left.height, right.height) +1, leaf is height 0

```
height()
  l = -1
  r = -1
  if (node.left != null)
    l = node.left.height;
  if (node.right != null)
    r = node.right.height;
  return max(l, r) + 1
```

- $O(h)$ operations, if balanced $O(\log n)$
- Max key: Traverse all the way to the right
- Min key: Traverse all the way to the left
- Search: Compare to current key, if query < key go left, if query > key go right, else found it; if null, return not found
- Insert: Search first, then insert into tree if an empty node is encountered
- Ways to insert: $n!$, no. of shapes: 4^n
- $O(n)$ operation
- In-order: L > self > R: think of an item below the nodes, then trace from root to left subtree and right subtree
- Pre-order: self > L > R: think of an item left of the nodes, then trace from root to left subtree and right subtree

- Post-order: L > R > self: think of an item right of the nodes, then trace from root to left subtree and right subtree
- Successor: $O(n)$ - if have right, return min of right, if not, go up the parent and find a node with a right tree to find min for

```
succ()
  if (node.right != null)
    return min_key(node.right)
  // walk up the tree, parent == null
  // means reached root
  parent = parentTree
  child = this
  while (parent != null &&
   child == parent.right)
    child = parent
    parent = child.parentTree
  return parent
```

```
altsucc()
  if (root == null || p == null) {
    return null;
  }
  if (p.right != null) {
    return findMin(p.right);
  }
  // Case 2: p.right == null
  TreeNode succ = null;
  TreeNode q = root;
  while (q != null) {
    if (q.val > p.val) {
      succ = q;
      q = q.left;
    } else if (q.val < p.val) {
      q = q.right;
    } else {
      break;
    }
  }
  return succ;
}
```

- Predecessor: $O(n)$

```
pred(root, node)
  if (root == null) {
    return null
  }
  if (node.left != null)
    return max(node.left)
  predec = null
  curr = root;
  while curr != null
    if (curr.val > node.val)
      curr = curr.left
    else if (curr.val < node.val)
      predec = curr
      curr = curr.right
    else
      break
  return predec
```

- Delete: $O(n)$
  - If no children, just delete the node
  - If have one child, assign the child to the parent of the node
  - If have two child, find the in-order successor, and swap it with the node to be deleted, then deleted the

Balanced Tree & AVL

- For height h, $n \leq 2^{h+1} - 1$
- $\log(n) - 1 \leq h \leq n$
- $\min(n_h) = n_{h-1} + n_{h-2} + 1$, where $n_0 = 1$ and $n_1 = 2$
- **A BST is balanced if h = O(log(n))**
- AVL nodes are augmented with height, and must be maintained on insertion and deletion (height updated by 1 along traversed path)
- Can just store the difference in height of children in the parent (-1 for left heavy, 0 for balanced, 1 for right heavy)
- **Depth**: no. of edges connecting root to node
- **Weight**: Total number of nodes on the left + right + 1
- **Rank**: The order of the element if we were to do in-order traversal (computed by node.left.weight + 1)
- **Invariant: Height-balanced**: |v.left.height − v.right.height| ≤ 1
- Min no of nodes for a tree of height h > max no of nodes for a tree of height $h - 1$
  - $n_h \geq 1 + n_{h-1} + n_{h-2} \geq 2n_{h-2} \geq 4n_{h-4} \geq \ldots \geq 2^k n_0 \geq 2^{h/2} n_0 = 2^{h/2}$
- Delete: $O(\log n)$
  - Delete the key from BST
  - Walk up from the point of deletion and check for balance, if imbalanced, rotate, then continue to the root
  - Can also do logical deletes (mark node as deleted) but needs to be cleaned up over time
- Rotation only possible if it has a left(for right) or right(for left) child
- Max no of rotations to balance AVL on insertion is 2
- Height balanced tree with height h has at least $2^{h/2}$ nodes ≡ height balanced tree with n nodes has at most height $h < 2\log(n)$
  - AVL balanced after insertion
  - Maximum increase in height after one insertion is 1
  - Consider all cases: v.left is balanced (occurs only on deletion, do right rot), v.left is left-heavy (right rot), v.left is right-heavy (left rot + right rot)
  - As long as we can prove that the rotation **reduces the height of the most imbalanced node by 1**, we are done
- Max no of rotations to balance AVL on deletion is $O(\log n)$
- Search, Insertion have time complexity of $O(L \log n)$, where $L = 1$ if no processing is done at each node
- AVL has the fastest expected search time $O(\log n)$
- No guarantee on the height of the tree at each node, they can differ greatly (not always true that height of node is always one less than its parents)
- Depth in an AVL tree can also differ greatly
- Similarly, inserting into an AVL tree may or may not take $< \log n$ time, since not all leaves are at depth $< \log n$ but also may not sometimes
- Invariants
  - If node u and v are siblings (same parents), then $|(\text{height}(u) - \text{height}(v)| < 2$ or $|(\text{height}(u) - \text{height}(v)|) \leq 1)$
  - If node u and v have the same grandparent, then $|(\text{height}(u) - \text{height}(v)| \leq 2$
  - If node u is the parent of node v, then $|\text{height}(u) - \text{height}(v)| > 0$ (height diff must be > 1)
  - Height of leaf is 0
  - Height of tree is always less than the number of nodes of the tree (max height is $O(\log n)$)
  - After insertion or deletion, for every node at height $h(u)$, the subtree rooted at u contains at most $2^{h(u)+1}$ nodes (including itself) → the total number of nodes rooted at a subtree in an AVL tree is 2^{height}, adding one in increases this value by 1
- To find median in a AVL tree, is $O(logn)$ if the weight of the tree at the node is augmented, otherwise is $O(n)$
- A fibonacci tree is always height balanced, but it is a maximally imbalanced AVL tree (height diff between nodes = 1); subtree with minimum possible number of nodes has 2 subtrees of heights $h-1$ and $h-2$, thus height of current node should be given by $S(h) = S(h-1) + S(h-2) + 1$
- **k**AVL trees are trees in which the nodes are **k**$h - good$ (the heights of all the child of node **u** is at least $height(u) - k$)
  - Max height is $\leq k * (\log_2 n + 1)$ (the max difference in height is leaf to root at height $k * height$, and the max height of a normal tree is $O(\log n)$, we +1 to this value to account for self)

Tries

- Cost to insert, query is $O(L)$, L is length of word
- Space to store everything is O(size of corpus) (with some overhead attached to it [* overhead])
- If we need to keep track of different categories of stuff, and the different categories is small, we can just use a few tries, no need complex algos

Tournament Trees

- Group elements into pairs, and compare, then winners will be paired up and compared again, until there is only 1 winner
- $O(\log n)$ height (max no. of comparisons for the winner)
- $O(n)$ total number of comparisons

kd-trees

- Search is $T(n) = 2T(n/4) + O(1) \equiv O(\sqrt{n})$, $2T(n/4)$ is due to the fact that we are recursing the 2 subtrees 2 levels down
- Tree Depth is $2 \log n$
- Recursion Depth is $\log(n)/2$; buildQuickSel $O(n \log n)$
- No. of nodes is $O(2^{\log(n)/2})$

Dynamic Order Statistics

- Use an AVL tree, augment with weight (weight is like the quickSelect partition count thing)
- Rank is costly to update so don't use it
- Select(k)

```
select(k)
  rank = node.left.weight + 1
  if (k == rank)
    return node
  else if k < rank
    return node.left.select(k)
  else if k > rank
    return node.right.select(k - rank)
```

- Rank(node)

```
  // traverse up the tree and add
  // weights if we go left
  rank(node)
    r = node.left.weight + 1
    while node != null
      if node is left child
        do nothing
      else if
        r += node.parent.left.weight + 1
      node = node.parent
    return r
```

Interval Trees

- Create a tree sorted by the left end point
- Each node is augmented with the max value in the left subtree
- It is always safe to just traverse left if cannot decide
- Invariants
  - If a search goes right, then there is no overlaps in the left subtree
  - If a search goes left and fails, then key < every interval in the right subtree

```
// O(log n)
interval(x)
  c = root
  while (c != null && x ! in x.interval)
    if c.left == null
      c = c.right
    else if (x > c.left.max)
      c = c.right
    else
      c = c.left
  return c.interval
// For all overlaps, O(k log n)
// best: O(k + log n)
while (intervals not empty)
  search for interval
  add to a list
  delete interval from intervals
for interval in list
  add interval back into intervals
```

(a, b)-tree - B-trees (B, 2B)-tree

- Rule 1: (a, b) children per node ($a \leq (b+1)/2$) [root can have 2 children min (1 key); no. of keys = a or b - 1]
- Rule 2: Key ranges (internal nodes have one more child than no. of keys)
- Rule 3: Leaves must be at the same depth
- DS: An array of (key, subtree) pairs, or let non-leaf nodes store pivots for navigating search while leaves store keys
- Let deg(v) be the number of children node v has, and height(leaf) = 0
- maxHt= $O(\log_a n + 1)$, minHt = $O(\log_b n)$
- Invariants
  - If node u and v are siblings, then $|deg(u) - deg(v)| \leq b$
  - If node u and v are siblings, then $|height(u) - height(v)| < 1$
  - If node u and v are siblings, then $|deg(u) - deg(v)| < 2$
  - If node u has height h, then subtree rooted at u contains at least a^h nodes
  - $2 \leq$ root.children $\leq b$
  - $a \leq$ internal.children $\leq b$
  - leaf.children = 0
- Searching for **a key** in the tree has recurrence relation $T(n) \leq T(n/c) + 1 = O(\log n)$
- Search time is $O(\log_2 b \cdot \log_a n) = O(\log_a n) = O(\log n)$ (using linear search, but asymptotically equivalent)
- Insertion
  - Find the leaf to insert into, if unbalanced, then find median, split node into 2 (< median, > median), push median to parent, link the new nodes to the parent, then continue to split upwards the tree if rules are still violated
  - For the root, do the same, except that the median is promoted to the root and the split nodes become the left and right child of the new root
  - Can be done proactively during search phase
  - For each level - $O(\log_a n)$ times
  - Split at most once - $O(b)$
  - Insert 1 item at the leaf - $O(1)$
- Deletion
  - Search for the element to delete
  - Case 1: Deletion causes no violations - just delete it
  - Case 2: Causes violations - merge, then split if necessary
  - Merge: Split away the parent key that separates the children, join with the children node, then update the links
  - To delete a key in the root node, swap with predecessor or successor, then delete it as usual
- Block Transfers
  - No. of block transfers needed to do a linear search on blocks for an item: n/b

  - ... to do a binary search on blocks for an item: $\log[n/b]$
  - Cost of searching a keylist, splitting a B-tree node and merging or sharing B-tree nodes is $O(1)$
  - Cost of searching a B-tree, inserting or deleting in a B-tree is $O(\log_b n)$

Scapegoat Tree

- Self-balancing BST with $O(\log n)$ lookup, insertion and delete
- Less node overhead, no need augment
- Usually α-weight-balanced, find highest scapegoat node to $O(n)$ rebalance when out-of-balance

Leetcode Questions?

Finding the longest non-increasing subarray

- Find a Prefix Max array ($[1, 3, 2, 1] -> [1, 3, 3, 3]$) $[O(n)]$

```
computePrefixMax(A)
  M = new array of size A.length
  max = 0
  for j = 0 to A.length - 1
    if (A[j] > max) {
      max = A[j]
    }
    M[j] = max
  return M
```

- Then do B-Search to find the smallest index j in M such that $M[j] > key$ $[O(logn)]$

```
findFirst(M, key)
  low = 0
  // no need to check the ends
  high = M.length

  // return if no max val
  if M[high] < key
    return -1
  while (low < high)
    mid = (low + high) / 2
    // <= finds the first next
    // index
    if (M[mid] <= key)
      low = mid + 1
    else if (M[mid] > key)
      high = mid
  return low
```

- To find the longest constant subarray, we need to use the previous functions $[O(nlogn)]$

```
computeMaxRecoveryPeriod(data)
  M = computePrefixMax(data)
  max = 0
  begin = -1
  end = -1
  for j = 0 to A.length - 1
    k = findFirst(M, data[j])
    if k >= 0 && k < j
      if j - k > max
        begin = k
        end = j
        max = end - begin
  return (begin, end)
```

- Basic idea: Iterate through all endpoints, and find possible beginning of recovery period (non-increasing subarray); find smallest index in *data* that maximises length of recovery interval, if M[j] > j, some point in [1, t] must be larger than j, hence we must find smallest t
- Invariants: There is at least one element in the range $>$ key
- Invariants: In every iteration, every value at an index $<$ low is $\leq$ key
- Every iteration reduces range between low and high, and low = high when it completes, ensures that we find the smallest index larger than the specified key
- Time complexity: $O(n) + O(nlogn) = O(nlogn)$

Range Queries

Expensive to insert and delete nodes for 2D range query as rotations may have to rebuild the entire y-tree, costing $O(n)$

- $O(log^d n + k)$ query
- $O(nlog^{d-1} n)$ buildTree
- $O(nlog^{d-1} n)$ space
- built by storing a $d-1$ dim range tree in each node of a 1D range tree, then build recursively
- Good only for static trees
- **When searching for a particular range, the query function below will only check the periphery nodes that surround the region, internal nodes within range will not be checked**
- If we are finding total number of elements in range and we have the weights of the nodes augmented, running time of *query* is $O(n)$

```
findSplit(low, high) {  // O(log n)
  v = root
  done = false
  while (!done) {
    if (high <= v.key) v = v.left
    else if (low > v.key) v = v.right
    else done = true
  }
  return v
}
// k is the number of items found
LTraversal(v, low, high) {  // O(k)
  if (low <= v.key) {
    all_leaf_traverse(v.right)
    LTraversal(v.left, low, high)
  } else {
    LTraversal(v.right, low, high)
  }
}

RTraversal(v, low, high) {  // O(k)
  if (v.key <= high) {
    all_leaf_traverse(v.left)
    LTraversal(v.right, low, high)
  } else {
    LTraversal(v.left, low, high)
  }
}

query(low, high) {  // O(k + log n)
  v = findSplit(low, high)
  left = LTraversal(v, low, high)
  right = RTraversal(v, low, high)
  return append(left, right)
}

// 2D, Query time is O(log^2 n + k),
// O(log n) to find split node, recursing
// steps and y-tree-searches of cost
// O(log n) and O(k) enumeration of output
// Space is O(n log n) (each point appear
// in at most 1 y-tree per level, O(log n)
// level
LTraversal(v, low, high) {  // O(k)
  if (v.key.x >= low.x) {
    ytree.search(low.y, high.y)
    LTraversal(v.left, low, high)
  } else {
    LTraversal(v.right, low, high)
  }
}

// Building trees O(n log n)
// (or O(n log^2 n)) at worst
```

Priority Queue

- Maintain a set of prioritized objects, must support **insert**, **increaseKey**, **extractMax**, **decreaseKey**, **extractMin**
- Can implement PQ with AVL tree, $O(logn)$ insertion and extractMax

Heap (Binary Heap / MaxHeap)

- Store biggest item at root, smallest at leaves
- Invariants
  - Heap Ordering: $priority[parent] \geq priority[child]$
  - Complete Binary Tree: Every level is full, except possibly the last; nodes should be as far left as possible
- Max height of Heap with n elements is $floor(logn) \approx O(logn)$
- Build size k maxHeap from n elem $O(n \log k)$

```
// All operations are O(log n)
// Same asymptotic costs, no rotations,
// better concurrency
bubbleUp(node) {
  while (node != null) {
    if (priority(node) > priority(parent)) {
      swap(v, parent(v))
    } else {
      return
    }
    node = parent(node)
  }
}

insert(priority, key) {
  // insert into the next available slot
  // lowest level, on the left
  node = tree.insert(priority, k)
  bubbleUp(node)
}

increaseKey(oldPrior, newPrior, key) {
  node = tree.query(oldPrior, key)
  node.priority = newPrior
  bubbleUp(node)
}

bubbleDown(node) {
  while (!isLeaf(node)) {
    left = priority(left(v))
    right = priority(right(v))
    max = max(left, right, priority(v))

    if (left == max) {
      swap(v, left(v))
      v = left(v)
    } else if (right == max) {
      swap(v, right(v))
      v = right(v)
    } else {
      return
    }
  }
}

decreaseKey(oldPrior, newPrior, key) {
  node = tree.query(oldPrior, key)
  node.priority = newPrior
  bubbleDown(node)
}

delete(key) {
  // swap key with the last element
  // remove the last element
  // bubbledown the swapped node
}

extractMax() {
  // save the root
  // delete the root from tree
  // return the root
}
```

- Can use an array to store a heap
- left(x): $2x + 1$ (x is index in array)
- right(x): $2x + 2$
- parent(x): $floor((x-1)/2)$
- AVL tree should not be stored in array, as rotations are no longer $O(1)$

HeapSort

- Start from unsorted list, make it into a heap, then extractMax and insert into the last vacant position in the array
- $O(nlogn)$ time complexity worst case, $O(n)$ space
- Deterministic: will always complete in $O(nlogn)$
- Unstable
- 3way heapsort faster
- Invariant: The last ith item of the array contains the largest ith item in the list in sorted order

```
// Naive Heapify O(n log n)
for (int i = 0; i < n; i++) {
  int value = nodeArray[i]
  nodeArray[i] = null
  // O(log n)
  insertIntoHeap(value,
    nodeArray, 0, i)
}

// Recursion (assume that leaves are
// heaps, and we combine heaps together
// by bubbling down nodes) O(n)
for (int i = n - 1; i >= 0; i--) {
  bubbleDown(i, nodeArray)
}

// HeapSort, O(n log n)
for (int i = (n - 1); i >= 0; i--) {
  int value = extractMax(heap)
  heap[i] = value
}
```

Expectation

- $E(X) = \sum x_i p_i$

Rotations

- If v is out of balance and LEFT HEAVY
  - v.left is balanced: right rotate v
  - v.left is left-heavy: right rotate v
  - v.left is right-heavy: left rotate v.left, then right rotate v
- If v is out of balance and RIGHT HEAVY
  - v.right is balanced: left rotate v
  - v.right is right-heavy: left rotate v
  - v.right is left-heavy: right rotate v.right, then left rotate v

Pancake Sort

- No. of flips: $O(2(n-2)+1)$
- If we have an extra condition that causes another flip for each pancake, then total no. of flips is $O(3(n-1)+1)$
- If have only 2 sized pancakes, worst case is when two pancakes of the same size are never touching each other (smaller pancake on the bottommost so there is one final flip)
- Lower bound: Any pancake sort algo requires at least n flips
  - Worst case would be that we have to insert our spatula between each pancake at least once to flip it (consider the table as a pancake with infinite size and is unflappable)

Random Shuffle

- Invariant? : Each permutation of the array must have $1/n!$ probability of occuring
- Invariant?? : The k^{th} element has equal chance of being any previously unselected item $x_j \equiv 1/(n-k+1)$
- Invariant??? : The first k items are a random permutation of the first k items in *xs*
- Knuth Shuffle

```
for i from 2 to n
  r = random(1, i)
  swap(A, i, r)
```

- Sorting shuffle (give keys random values, sort based on values)

Reservoir Sampling

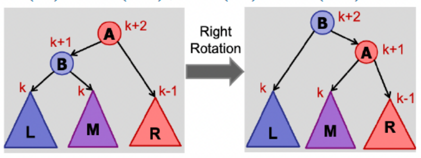
- Given a stream of data how do we ensure we fairly pick elements from it? We have a bag that stores only 1 element at time.
- Invariant: $P(x = x_i) = 1/k$, where i is in $1...k$, maintained by using the new item with probability $1/k+1$ and keeping the old item with probability $k/k+1$
- This gives us an overall probability of selecting any i^{th} element from the stream with probability $1/n$
- No. of expewcted swap of items: $O(logn)$
- If now we want to select k items from it, then the probability of selecting an item fairly is k/n
- Probability of picking item and swapping with a specific slot in the bag is now $k/i * 1/k = 1/i$, probability of not having a slot replaced by another item is $1 - 1/i$

Manhattan Distance

- sum $+ = i \cdot a[i] - (N - 1 - i) \cdot a[i]$ is $O(n)$

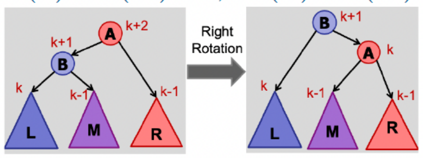
[case 1] B is **balanced: right-rotate**

$$h(L) = h(M), \quad h(R) = h(M) - 1$$



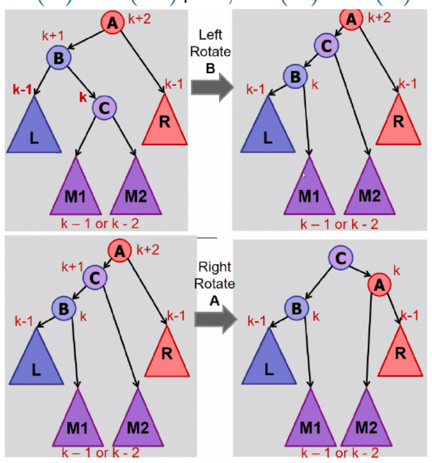
[case 2] B is **left-heavy: right-rotate**

$$h(L) = h(M) + 1, \quad h(R) = h(M)$$



[case 3] B is **right-heavy: left-rotate(v.left), right-rotate(v)**

$$h(L) = h(M) - 1, \quad h(R) = h(L)$$



Sort	Sorted	Almost Sorted	Arrays with small no. of unique elements	Best	Average	Worst	Space	Stable?	In-place?
Bogo	O(n * n!)	O(n * n!)	O(n * n!)	O(n)	O(n * n!)	O(n * n!)	O(1)	No	Depends on shuffle
Bubble	O(n)	O(n^2)	O(n^2)	O(n)	O(n^2)	O(n^2)	O(1)	Yes	Yes
Insertion	O(n)	O(n * k) k is no. of out of order pairs	O(n^2)	O(n) (> bubble)	O(n^2)	O(n^2)	O(1)	Yes	Yes
Selection	O(n^2)	O(n^2)	O(n^2)	O(n^2)	O(n^2)	O(n^2)	O(1)	No	Yes
Merge	O(n log n)	O(n log n)	O(n log n)	O(n log n)	O(n log n)	O(n log n)	O(1) (inplace) O(n) OR O(n log n)	Yes (take left list first)	No (possible but hard)
QuickSort	O(n)	O(n log n)	O(n) (3-way part) O(n log n) (usual)	O(n log n)	O(n log n)	O(n^2)	O(1) (inplace) O(n) (new array)	Possible (3-way partitioning)	Possible (In-place partitioning)

sort	**best**	**average**	**worst**	**stable?**	**memory**
bubble	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
selection	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$	×	$O(1)$
insertion	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
merge	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓	$O(n)$
quick	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	×	$O(1)$

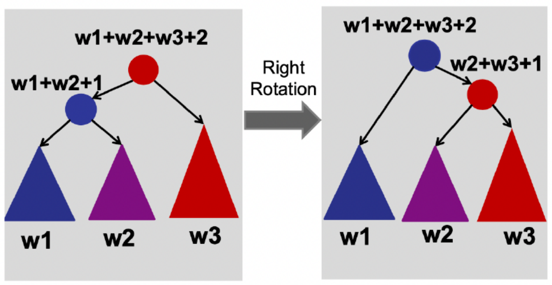
sorting invariants

sort	**invariant** (after k iterations)
bubble	largest k elements are sorted
selection	smallest k elements are sorted
insertion	first k slots are sorted
merge	given subarray is sorted
quick	partition is in the right position

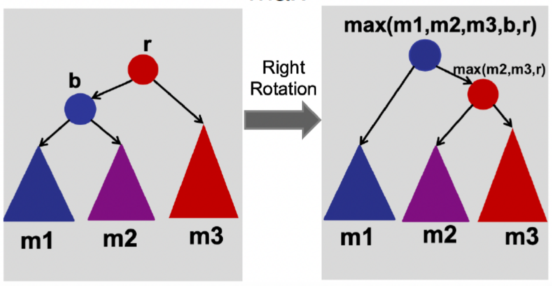
searching

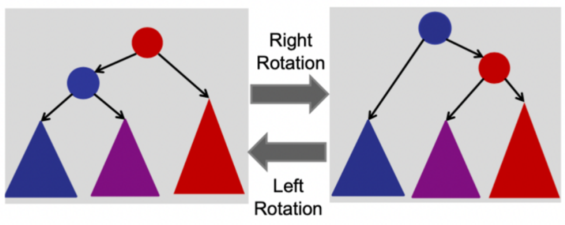
search	**average**
linear	$O(n)$
binary	$O(\log n)$
quickSelect	$O(n)$
interval	$O(\log n)$
all-overlaps	$O(k \log n)$
1D range	$O(k + \log n)$
2D range	$O(k + \log^2 n)$

weights



max



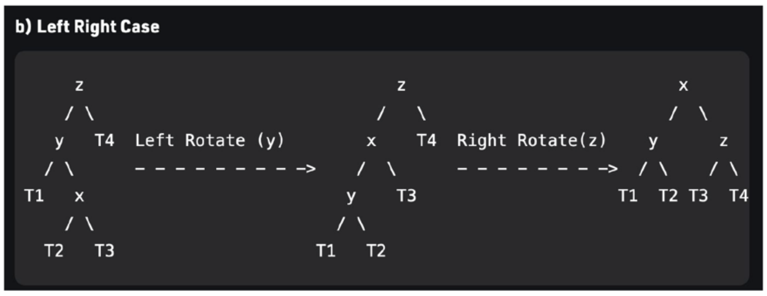


data structures assuming $O(1)$ comparison cost

data structure	**search**	**insert**
sorted array	$O(\log n)$	$O(n)$
unsorted array	$O(n)$	$O(1)$
linked list	$O(n)$	$O(1)$
tree (kd/(a, b)/binary)	$O(\log n)$ or $O(h)$	$O(\log n)$ or $O(h)$
trie	$O(L)$	$O(L)$
dictionary	$O(\log n)$	$O(\log n)$
symbol table	$O(1)$	$O(1)$
chaining	$O(n)$	$O(1)$
open addressing	$\frac{1}{1-\alpha} = O(1)$	$O(1)$

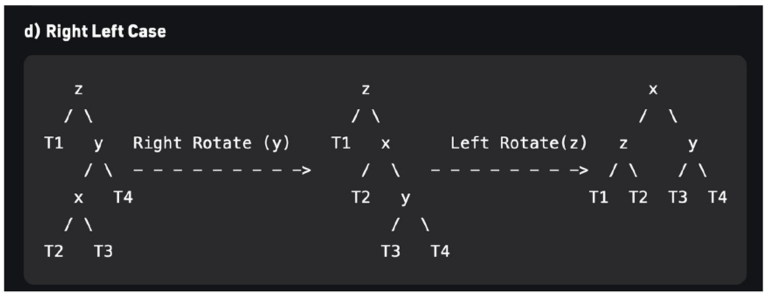
orders of growth

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < 2^{2n}$$
$$\log_a n < n^a < a^n < n! < n^n$$



b) Left Right Case

```
     z                               z                                   x
    / \                             /  \                                /  \
   y    T4   Left Rotate (y)       x    T4   Right Rotate(z)          y      z
  / \        - - - - - - - - ->   /  \       - - - - - - - ->        / \    / \
T1    x                          y    T3                           T1   T2 T3   T4
     / \                        / \
   T2   T3                    T1   T2
```



d) Right Left Case

```
   z                            z                            x
  / \                          / \                          /  \
T1    y   Right Rotate (y)   T1    x      Left Rotate(z)   z      y
     / \  - - - - - - - - ->      /  \    - - - - - - - ->/ \    / \
    x   T4                      T2    y                 T1   T2 T3   T4
   / \                               /  \
 T2   T3                           T3    T4
```

Master Theorem

For general form $T(n) = aT(n/b) + f(n)$,
where $a \geq 1$, $b > 1$, $f(n) = \Theta(n^k \log^p n)$
$\boxed{\text{Case 1}}$ if $\log_b a > k$, then $\Theta(n^{\log_b a})$
$\boxed{\text{Case 2}}$ if $\log_b a = k$,
if $p > -1$, then $\Theta(n^k \log^{p+1} n)$
if $p = -1$, then $\Theta(n^k \log(\log n))$
if $p < -1$, then $\Theta(n^k)$
$\boxed{\text{Case 3}}$ if $\log_b a < k$
if $p \geq 0$, then $\Theta(n^k \log^p n)$
if $p < 0$, then $O(n^k)$

For general form $T(n) = aT(n - b) + f(n)$,
where $a > 0$, $b > 0$, $f(n) = O(n^k)$, $k \geq 0$
$\boxed{\text{Case 1}}$ if $a < 1$, then $O(f(n))$
$\boxed{\text{Case 2}}$ if $a = 1$, then $O(n \cdot f(n))$
$\boxed{\text{Case 3}}$ if $a > 1$, then $O(a^{n/b} \cdot f(n))$

2 cases

T1, T2, etc are subtrees

inserted node is somewhere in the leaves

Can come up with example for both cases, but we will just pick RIGHT-LEFT for the sake of this question (results are actually the same if you trace for both cases)

Potentially infer the answer from this via pattern recognition.

The parent of the newly inserted node becomes the root of the balanced subtree