**ORDERS OF GROWTH  
Definitions**

if for all if for alliff

**Properties**  
Let and   
1.   
2.   
3.   
4. Cost of if/else statements:   
5.   
**Notes**1. is   
2.   
3. Sterling’s approx.  
4.   
5.   
6.7.8.9.10.11.  
**Master Theorem**

|  |  |  |
| --- | --- | --- |
| **SORTING ALGO** | **Description** | **Invariant** |
| **Bubble Sort** | Compare adjacent (1st 2nd,2nd 3rd ) items and swap. | Largest i elements are sorted |
| **Selection Sort** | Select minimum element from range of low/high index, swaps into position. Repeat with increasing low index until all elements have been selected. Has least swaps needed | Smallest i elements are sorted |
| **Insertion Sort** | Compare the key with the previous elements. If the previous elements are greater than key, swap key to left until it is smaller. Start from index 1 to array size. | Subarray A[0 to i-1] is always sorted |
| **Heap Sort** | Repeatedly extractMax() element from heap, place it at end of array, update heap | In max heap, last i elements are sorted, vice versa min heap |
| **Merge Sort** | Divide array into half, recursively sort, then merge | Each subarray is already sorted when merging |
| **Quick Sort** | Partition around chosen/random element. Low/high index move left/right until element is bigger/smaller than pivot, swap low and high, then repeat on subarrays. | All elements to the left/right of pivot are smaller/larger.  Partition is in right position |

**Quick Sort**Partition algorithm: O(n)  
Stable quicksort: O(log n) space  
> 1st element as partition, 2 pointers from left to right  
 - left pointer moves until index > pivot  
 - right pointer moves until index < pivot  
 - swap elements until left = right  
> swap partition and left=right index  
**Quick Sort Optimizations**  
1. 3-way partition w/ dup array w/o dup array  
- 4 pointers - in-progress, < pivot, = pivot and > pivot  
 > If A[i] < pivot -> swap in-progress pointer with < pivot pointer  
 > If A[i] = pivot -> swap in-progress pointer with = pivot pointer  
 > If A[i] > pivot -> swap in-progress pointer with > pivot pointer  
2. Stable if partitioning is also stable  
3. Extra memory for stable quick sort  
**Choice of Pivot**  
1. : 1st/last/middle element  
2. : median/random element  
 > same if split by fractions  
3. Choose at random- random var runtime  
**Quick Select**  
O(n) to find the kth smallest element  
1. After partition, pivot is always in correct position  
2. Recurse left/right of pivot if kth is smaller/bigger.  
Duplicates works on quick select.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ALGO** | **Best** | **Average** | **Worst** | **Stable** |
| **Bubble** |  |  |  | ✓ |
| **Selection** |  |  |  | ✕ |
| **Insertion** |  |  |  | ✓ |
| **Heap** |  |  |  | ✕ |
| **Merge** |  |  |  | ✓ |
| **Quick** |  |  |  | ✕ |
| **Quick Select** |  |  |  | ✕ |

Table

Description automatically generated  
Text

Description automatically generated with medium confidenceTable

Description automatically generated  
Text

Description automatically generated with medium confidence

**TREES  
Binary Search Tree (BST)**1. Either empty, or node pointing to 2 BST  
2. Tree balance depends on insertion order  
3. Balanced tree:   
4. For full tree of size  
**BST Operations**  
1. height(n) = max(height(n.left), height(n.right))  
2. search(n),insert(n):   
3. delete(n):

1. no children – remove node  
2. 1 child – remove node, connect parent to child  
3. 2 children – delete successor, replace node with successor.  
4. searchMin(n): recurse left tree

5. successor(n):   
 1. If node has right subtree,   
 searchMin(n.right)else: traverse   
 upwards, return 1st parent that contains key   
 in left subtree.

Shape, polygon

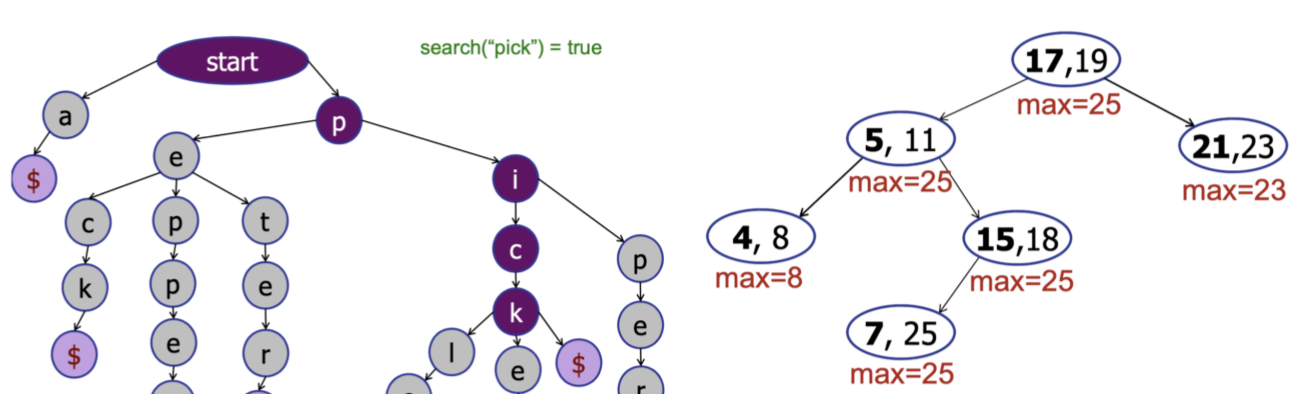
Description automatically generated**AVL Trees**  
1. Height-balanced iff | n.left.height -n.right.height|   
2. Node is augmented with its height  
3. Space complexity: for strings of length  
**Rebalancing**  
Shape, polygon

Description automatically generated **Updating nodes after Rotation** Shape

Description automatically generatedShape

Description automatically generatedA diagram of different colored arrows

Description automatically generated with low confidence  
1. Worst case Insertion: max 2 rotations  
2. Worst case Deletion: O(log n)  
3. Rotations can create every possible tree shape  
**Tries**  
1. search(n),insert(n):   
2. Space:

**Interval Trees**  
1. search(key):  
 > if value in root interval, return  
 > if value > max(left subtree), recurse right  
 > else recurse left (only when you can’t go  
 right)  
2. All overlaps:for overlapping   
 intervals (repeat algo until no more intervals)  
  
**Orthogonal Range Searching**1. BST: leaves store points, parent nodes store max value in left subtree

Diagram

Description automatically generated

2. build(points[])  
3. query(a, b): for points  
 1. Find Node between low/high, starting at root  
 =findSplit(low,high)  
 2. Output all node in right subtree & recurse left,  
 or recurse right = leftTraverse(n):  
 3. Symmetric to left traversal  
 rightTraverse(n):4. insert(key)  
5. nodeCount(v,low,high): Left-traverse but count weight of right subtree instead of traversing.**Order Statistics**1. Finding rank k in an augmented AVL  
 1. Rank = left.weight + 1  
 2. If k is ranked, return node  
 3. If k<rank, recurse left subtree with rank, else  
 recurse right subtree with rank-1  
2. Find rank given node n  
 1. If n has left child, rank = left.weight + 1  
 2. Else set node as rank = 1, traverse upwards  
 3. Go to parent, if node is parent’s left child, keep  
 rank  
 4. If node is right child, rank += parent.left.weight  
 +1, continue traversal upwards  
3. Maintain weight during insertions  
 > Add item, then traverse upwards and add 1 to   
 each node until root is reached  
 > If tree is not balanced, rotate to balance  
 - When doing right rotation, only need to   
 update the root and parent node for weight  
**Tree Traversal**  
Pre Order: print, root , left, right  
Diagram

Description automatically generated  
In Order: left, print, root, , right

|  |
| --- |
|  |

Post Order: left, right, print, root  
**Diagram

Description automatically generated**  
**KD Tree**  
1. Median using quick select as root, alternates splitting via x and y coordinates  
2. construct(points[])  
3. search(points)  
4. minimum(points)  
**Priority Queue**

|  |  |  |
| --- | --- | --- |
| **Data** | **Insert** | **ExtractMax** |
| Sorted Array |  |  |
| Unsorted |  |  |
| AVL Tree |  |  |

**Heap**1. Max heap: Stores biggest in root, smallest in leaf   
 Min heap: Stores smallest in root, biggest in leaf  
2. Priority of parent always >= child in max heap  
3. It is a complete binary tree: every level is full (has both left/right child) | all leaves are far left  
4. height(n)= floor(log n)  
5. insert(n): far left if priority of n > parent: Bubbleup/swap. Check 2. And 3.  
6. extractMax(n): return root and delete root  
7. increase(n,a): increase n to a, Bubbleup a  
8. decrease(n,b): decrease n to b, move b down, bubbleDown to child with bigger priority  
9. delete(n): swap n with last(), remove last(). bubbleDown /up depending on heap order  
Mapping heap into array:  
1. Start from root, at each level, insert from left/right  
2. Insert: insert into empty array slot, bubble up by swapping indexes  
3. Left(x) = | Right(x) =   
4. Parent(x) =   
Mapping unsorted array into heap:  
1. Iterate from end of array, bubbleDown current index and array: