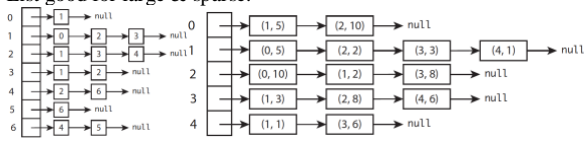


Insertion in 2 subtree PreC: -The BST Dictionary Invariant is satisfied. -A key $k \in K$, value $v \in V$, & a non-null node x in this BST have been given as input. -The BST Invariant would still be satisfied if this BST was changed by including a node s 2 ring k & v (at the appropriate place) 2 the subtree w / root x — either by changing the value s 2 red at a node s 2 ring k if there is one, or by adding a new node if no such node exists. PostC: -The BST Dictionary Invariant is still satisfied. -A node s 2 ring k & v has been included in the subtree w / root x —either by replacing the value at a node s 2 ring k , or by adding a new node if no such node existed • If the input value is less than the value at the root, recursively insert the input value in 2 the left subtree of the root. • If the input value is greater than the value at the root, recursively insert the input value in 2 the right subtree of the root. • Finally, if the input value is equal 2 the value at the root, throw an exception & do not change the tree at all.		Deletion on 2 a subtree PreC:-The BST Dictionary Invariant is satisfied -A key $k \in K$ & (possibly null) node x in this BST have been given as input. -If x is null then there is no node s 2 ring k in this BST. Otherwise, if there is a node s 2 ring k , then it is a node in the subtree w / root x PostC:-The BST Dictionary Invariant is still satisfied. -If $f(k)$ was defined, for the function $f: K \rightarrow V$ represented by this BST, then $f(k)$ has been set 2 be undefined, w/out making other changes. Otherwise a NoSuchElementException has been thrown C#1: This node is a leaf. In this case it should simply be deleted C# 2 : The L child of x is null but the R child of x is not null: In this case, the R child of x must replace x C#3: The R child of x is null but the L child of x is not null: In this case, the L child of x must replace x C#4: Neither the L nor R child of x is null: Let y be the node in the R subtree of x s 2 ring the smallest key. Let us call this node the successor of x . Copy the key & value s 2 red at y in 2 the node x — overwriting (& eliminating) the key-value pair s 2 red at x . Delete the node y whose key & value have just been copied in 2 x	
Red- Black Tree -Red-black trees always have depths that are at most logarithmic in the sizes of these trees -Every node is either red or black. -The root is black. -Every leaf (NIL node) is black. -If a node is red, then both its children are black. -For each node, all paths from the node 2 descendant leaves contain the same # of black nodes HTs w/ Chaining -A collision occurs if two keys k_1 & k_2 (used in the set or map) hash 2 the same location, that is, $h(k_1) = h(k_2)$ -Collisions are unavoidable if the size of the set or map is greater than the table size -In a HT w/ Chaining, we put all the keys (used in the set or map) that hash 2 the same location ℓ in 2 a linked list (NO NULL) -the worst case running time is equal 2 the size of the set being represented! -The average cost of these operations depends on:The likelihood of each kind of operation, & The shape of the HT. (The shape of the HT only depends on the locations 2 which keys are hashed — not on the values of the keys, themselves) HTs w/ Open Addressing -In a HT w/ open addressing, all elements are s 2 red in the HT itself (w/ each being a value, NIL or deleted) -Linear probing: If slot $\text{hash}(x)\%S$ is full, then we try $(\text{hash}(x) + 1) \% S$ -Quadratic probing: If slot $\text{hash}(x)\%S$ is full, then we try $(\text{hash}(x) + 1^2)\%S$ then we try $(\text{hash}(x) + 2^2)\%S$ <div> <div>0</div> <div>1</div> <div>2</div> <div>3</div> <div>4</div> <div>5</div> <div>6</div> <div>7</div> <div>8</div> <div>9</div> </div> <div> <div>0</div> <div>DEL</div> <div>NIL</div> <div>NIL</div> <div>10</div> <div>5</div> <div>NIL</div> <div>NIL</div> <div>30</div> </div> Linear Search $O(n)$ Binary Search $O(\log n)$ -If $\text{high} < \text{low}$ then a NoSuchElementException else $\text{mid} = \lfloor (\text{low} + \text{high}) / 2 \rfloor$ • If $k < A[\text{mid}]$ then the AL is called recursively; low is not changed, but high is replaced by $\text{mid} - 1$. The output generated (or exception thrown) by this recursive call is returned. • If $k = A[\text{mid}]$ then mid is returned as output. • If $k > A[\text{mid}]$ then the AL is called recursively: low is replaced by $\text{mid}+1$ but high is not changed. The output generated/exception by this recursive call is returned.		When you delete a node in a black & red tree the node deleted is either red or black (when the node does not exist then an exception is thrown) Case A: The colour of the deleted node y was red:• The parent of y must be black.• When x is promoted its parent (the former parent of y) will be black — & its black-height is still well-defined, & unchanged.• Then T is still a RB tree. \Rightarrow We are finished! Case B: the colour of the deleted node y was black: <u>The first problem:</u> Paths from nodes 2 leaves that included y are now missing one black node so that “black-height” is not well-defined <u>The second problem:</u> It is possible that either x & its parent might both be red, or x might be red & the root The fix is 2 use red-black node that count (it counts as a black node on the way when computing red & black height) or 2 use double black nodes (it counts as two black) - x is RB when x was a red child of the deleted black node - x is double black when x was a black child of the deleted black node - x is a red-black node. There are no other problems in the tree - x is a DB node at the root. There are no other problems in the tree - x is a DB node, not at the root. There are no other problems in the tree Solution 2 these Cases above -when x is a red-black node. Solution: Change x 2 a black node, & s 2 p! -When x is a DB node at the root. Solution: Change x 2 a black node, & s 2 p! -When is a DB node not at the root. A series of adjustments will be made that will either move the problem closer 2 the root, or 2 switch 2 one of the easier cases that we already know how 2 h&le -when x is the L or R child of its parent , go through all possible node combinations, find the impossible cases (according 2 RB tree properties), address possible cases by using rotations & changing colours breaking the problem down 2 easier subcases 2 establish the postcondition & RB tree properties	
Selection Sort $O(n^2)$ -sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part & putting it at the beginning	Insertion sort $O(n^2)$ -builds the final sorted array (or list) one item at a time	Bubble sort $O(n^2)$ -works by repeatedly swapping the adjacent elements if they are in wrong order	
Merge sort $O(n \log n)$ -much faster, in the worst case, than any of the classical ALs, when used 2 sort large arrays -it does not sort in place: Another array is produced as output — so the space requirements of this AL are significantly higher -it divides input array in two halves, calls itself for the two halves & then merges the two sorted halves		Quick sort $O(n^2)$ -The # of steps used by this AL 2 sort an array w/ length n is in $\Theta(n^2)$ worst case. -A deterministic linear-time 2 choose the median element in a subarray can be used 2 choose the partition element — in order 2 guarantee that when Quick Sort if recursively called, the subarray 2 be recursively sorted always has at most one-half the length of the original subarray $\Theta(n \log n)$	
Binary Heap -insert fr left 2 right -Max-Heap Property: The value at each node is greater than or equal 2 values at any children of the node [Bubble the problem node up 2ward the root (by exchanging its value with the value at its parent, if they are out of order)] Min-Heap Property: The value at each node is less than or equal 2 the values at any children of the node [Bubble the problem down 2ward the leaves of the heap (by changing its value with the value at one of its children, if they are not all in order)]		Heap Sort $O(n \log n)$ -comparison based sorting technique based on Binary Heap data structure. It is similar 2 selection sort where we first find the maximum element & place the maximum element at the end 1. Build a max heap from the input data 2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree. 3. Repeat above steps while size of heap is greater than 1 Partitioning: Speed & position	

<p>Priority Qs</p> <p>-an abstract data type that maintains a finite multiset S of elements, of some type V, each with an associated value (of some ordered type P) called a priority</p> <ul style="list-style-type: none"> • max-priority-Qs (which are easily implemented using Maxheaps), and • min-priority-Qs (which are easily implemented using Minheaps). <p>Graphs</p> <ul style="list-style-type: none"> • u and v are neighbors (or, “u is adjacent to v”) if $(u, v) \in E$ • If $u \in V$ then the degree of u is the number of neighbors of u <p>List good for large & sparse:</p>  <p>-If $G = (V, E)$ is a connected undirected graph, then a spanning tree of G is a graph $bG = (bV, bE)$ such that</p> <p>-$G = (V, E)$ is an induced subgraph of G if G is a subgraph of G and, furthermore $E = \{(u, v) \in E \mid u, v \in V\}$, that is, G includes all the edges from G that it possibly could</p> <p>-minimal spanning tree is a spanning tree with the minimum cost (is the sum of the weights of the edges)</p>	<p>Randomized Partitioning: A “randomized” solution that chooses random values during its computation — so that the output is not fixed, even when the input is $O(\log n)$</p> <p>Deterministic version uses $\Theta(n^2)$ steps to sort an array of size n worst case; $\Theta(n \log n)$ steps on average</p> <p>-Compared 2 other ALs that sort in place using only a constant amount of additional storage, this AL has the best asymptotic worst-case performance. We will shortly see another AL — Quick Sort — that is often faster “in practice,” but whose worst-case performance is asymptotically much worse</p> <p>-This is a rather complicated AL (if insert and deleteMax are considered)</p>
<p>Breadth First Search</p> <p>-Begin with s; expand the boundary between “discovered” and “undiscovered” vertices uniformly across the breadth of the boundary</p> <p>-go through each level and use a queue</p> <p>Minimum Cost Paths — Dijkstra’s Algorithm</p>	<p>Depth First Search</p> <p>-Given a graph G and a vertex s, this algorithm finds a tree with root s whose edges are chosen by searching as deeply down a path as possible before “backtracking”</p> <p>-go through by node by node with a stack</p>