|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm analysis** | | | | | |
| **Big O:** Let and be functions that map positive integers to real positive numbers.  is if for some and | | | | | |
| **Big Omega ()**: Let and be functions that map positive integers to real positive numbers.  is if for some and | | | | | |
| **Big Theta()**: Let and be functions that map positive integers to real positive numbers.  is if for some and | | | | | |
| **Loop invariant**: Any predicate/condition that holds true for every iteration of a loop | | | | | |
| **Recursion** | | | | | |
| - Understand differences between linear recursion, binary recursion, multiple recursion in terms how many recursive class each function makes | | | | | |
| **Trees** | | | | | |
| **Tree**: If a tree is non-empty, there exists a root node with no parent. Each child of that root is in turn the root of another sub-tree | | | | | |
| **Descendant**: Any node lower by 2 or more levels | | | **Ancestor:** Any node higher by 2 or more levels | | |
| **Depth:** Number of levels separating node from the root | | | **Height**: The maximum levels of the tree (excluding root) | | |
| **Proper binary tree**: Each node must have either 0 or 2 children | | | | | |
| **Complexities**: depth*,* height | | | | | |
| **Preorder:** Visit node before left-right children  **Postorder:** Visit left-right children before node  **Inorder:** Visit left child, node, right child | | | | | |
| **Permutations of n elements in a BST**: Given by the *nth* Catalan number, | | | | | |
| **Sorted maps and Balanced Search Trees** | | | | | |
| **Map**: Stores key-value pairs. Also known as an associative array | | | | | |
| **Balanced search tree**: A binary tree is balanced if for every internal position p, the heights of p’s children differ at most by 1 | | | | | |
| **Deletion policy**: If internal node with 2 children, replace with inorder predecessor | | | | | |
| **AVL tree**: Balance after each insertion/deletion with trinode restructuring | | | | | |
| **2,4 tree**: *Size* *property –* Every internal node has at most 4 children. *Depth property –* all external nodes have the same depth  **Insertion**: On overflow, promote third child (k3 is promoted from (k1, k2, k3, k4) )  **Deletion**: Remove node, replace and cascade if internal, until we remove a key from an internal node whose children are external nodes. If underflow and sibling is a 3/4-node, transfer. Otherwise if sibling is a 2-node, merge both to form a 3-node | | | | | |
| **Red-black tree**: *Root property –* root is black. *External property* – every external sentinel node is black. *Red property* – the children of a red node are black. *Depth property* – All external nodes have the same black depth  **Insertion**: = inserted node, = parent, = grandparent. If is red, double red. If has black sibling, trinode restructuring on to form new 4-node. Otherwise, recolor , ’s sibling, and , propagating if necessary.  **Deletion**: Delete normally and cascade until reaching a node with external child. If red, no issue. If black with one red child, promote red child and color black. Otherwise, double black. = promoted child, = sibling of , = common parent of and .  1: If is black with a red child , trinode restructuring on (2,4 transfer)  2 :If is black with black children, recolor to red, *p* to black and *z* black or double black and propagate (2,4 fusion)  3: If is red, must be black. Rotate such that is the parent of , recolor to black, to red. Go to case 1 or 2 | | | | | |
| **Complexities**: search, insertion, removal for AVL (restructuring) and RBT (recoloring)  For red-black tree insertion, trinode restructure. For red-black tree deletion, trinode restructures | | | | | |
| **Hash Tables and Maps** | | | | | |
| **Hashcode implementations**: XOR, polynomial function, bitwise cyclic shift | | | | | |
| **Probing methods**: Given a hashcode , we probe for :  Linear probing , Quadratic probing , Double hashing | | | | | |
| **Heaps and Priority Queues** | | | | | |
| **Heap:** *Heap-order property* – for every position p,the key is greater than its parent. *Complete binary tree property –* every level of the tree has the maximal number of nodes possible, and the remaining noes reside in the leftmost possible positions. | | | | | |
| **Complexities**: insertion+removal for list implementations, for heap. heap insertion, heapify | | | | | |
| **Graphs** | | | | | |
| **Graph:** A tuple where is the set of vertices and is the set of edges | | | | | |
| **Path**: A set of alternating vertices and edges from u to v, where each edge is incident on the immediate predecessor and successor vertices. *Simple* if no repeated vertices | | | | | |
| **Cycle**: A path from u to itself, involving at least one other vertex. *Simple* if no repeated vertices | | | | | |
| **Connectedness**: A graph is connected if there is a path between any 2 vertices. *Strongly connected* if for any pair of vertices and , is reachable from and is reachable from | | | | | |
| **Subgraph**:is a subgraph of if and . *Spanning subgraph* if , | | | | | |
| **Degree**: Number of edges incident on a vertex. *In-degree* - incoming edges, *Out-degree* – outgoing edges | | | | | |
| **Edge list**: vertices and edges stored in separate unordered lists. Limitations in processing edges for a given vertex  **Adjacency list**: vertices stored in an unordered list, where each vertex maintains its own unordered list of all incident edges  **Adjacency map**: vertices stored in an unordered list, each vertex maintains a map where key=adjacent vertex and value=edge  **Adjacency matrix**: 2D array of , where holds a references to the edge if it exists | | | | | |
| **Topological ordering**:Any givengraph has a topological ordering if and only if it is acyclic. Its vertices are ordered such that for every edge of , we have | | | | | |
| **Minimum spanning tree**: Given an undirected, weighted graph , a minimum spanning tree of is a tree containing all the vertices in , that minimizes the sum of weights, . If all edges of has distinct weights, the minimum spanning tree is unique | | | | | |
| **Method** | **Edge List** | **Adj List** | | **Adj Map** | **Adj Matrix** |
| numVertices(), numEdges() |  |  | |  |  |
| vertices() |  |  | |  |  |
| edges() |  |  | |  |  |
| getEdge() |  | O(1) | | O(1) expected | O(1) |
| outDegree(), indegree() |  |  | |  |  |
| outgoingEdges(), incomingEdges() |  |  | |  |  |
| removeVertex(v) |  |  | |  |  |
| insertVertex(), insertEdge(), removeEdge() |  |  | |  |  |
| **Prim-Jarnik**: Start from any vertex as its own graph, build up by adding lowest cost edges to undiscovered nodes | | | | | |
| **Kruskal**: Each node is its own cluster at the beginning. Iteratively consume lowest cost edges that connect different clusters | | | | | |