**VII. Tree & Graph**

1. **Tree Data Structure**

* **binary tree**
* A tree data structure in which each node has at most two children, which are referred to as the left child and the right child.
* **Full tree**: every node other than the leaves has two children.
* **Complete** tree: every level except possibly the last, is completely filled, and all nodes are as far left as possible.
* **Perfect tree**: a complete BT whose all leaves are on the same level

|  |  |  |
| --- | --- | --- |
|  | Average | Worst case |
| **Space** | Θ(*n*) | O(*n*) |
| **Search** | Θ(log *n*) | O(*n*) |
| **Insert** | Θ(log *n*) | O(*n*) |
| **Delete** | Θ(log *n*) | O(*n*) |

* **AVL tree**
* self-balancing binary search tree.
* heights of the two child sub trees of any node differ by at most one
* Ensures O(log N) depth

|  |  |  |
| --- | --- | --- |
|  | Average | Worst case |
| **Space** | O(*n*) | O(*n*) |
| **Search** | O(log *n*) | O(log *n*) |
| **Insert** | O(log *n*) | O(log *n*) |
| **Delete** | O(log *n*) | O(log *n*) |
|  |  |  |

* Single rotation: left child’s left subtree, or right child’s right subtree. (outside)
* Double rotation: left child’s right subtree, or right child’s left subtree. (inside)
* As a result, lookup in an AVL tree is typically faster, but slower insertion and deletion due to more rotation operations.
* So use an AVL tree if the number of lookups > updates to the tree.
* **Red–black tree**
* A kind of self-balancing binary search tree.
* Each node of the binary tree has an extra bit color(red or black) of the node.
* These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.
* A node is either red or black.
* The root is black.
* All leaves (NIL) are black.
* If a node is red, then both its children are black.
* Every [path](https://en.wikipedia.org/wiki/Path_(graph_theory)) from a given node to any of its descendant NIL nodes contains the same number of black nodes.

|  |  |  |
| --- | --- | --- |
|  | Average | Worst case |
| **Space** | O(*n*) | O(*n*) |
| **Search** | O(log *n*) | O(log *n*) |
| **Insert** | O(log *n*) | O(log *n*) |
| **Delete** | O(log *n*) | O(log *n*) |

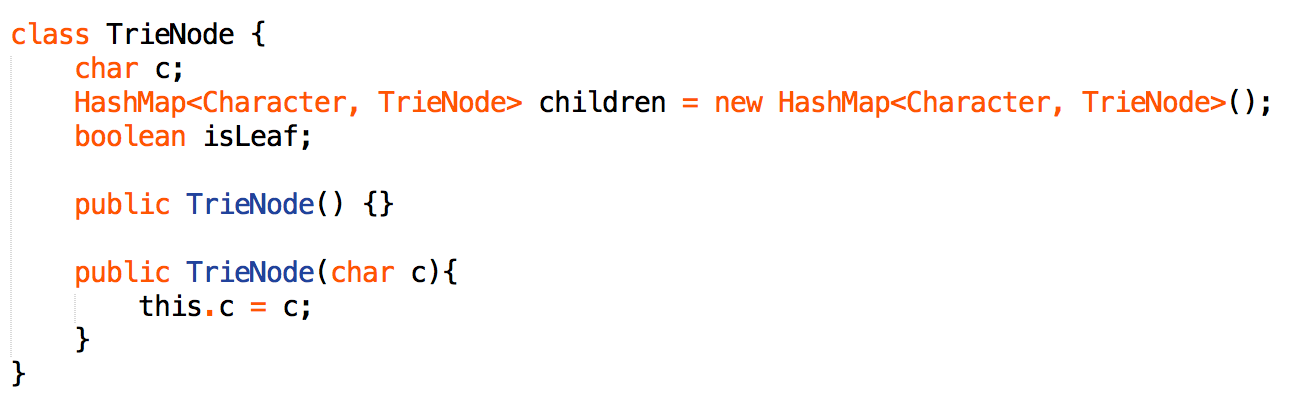
* **splay tree**
* a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again

|  |  |  |
| --- | --- | --- |
|  | Average | Worst case |
| **Space** | O(n) | O(n) |
| **Search** | O(log n) | amortized O(log n) |
| **Insert** | O(log n) | amortized O(log n) |
| **Delete** | O(log n) | amortized O(log n) |

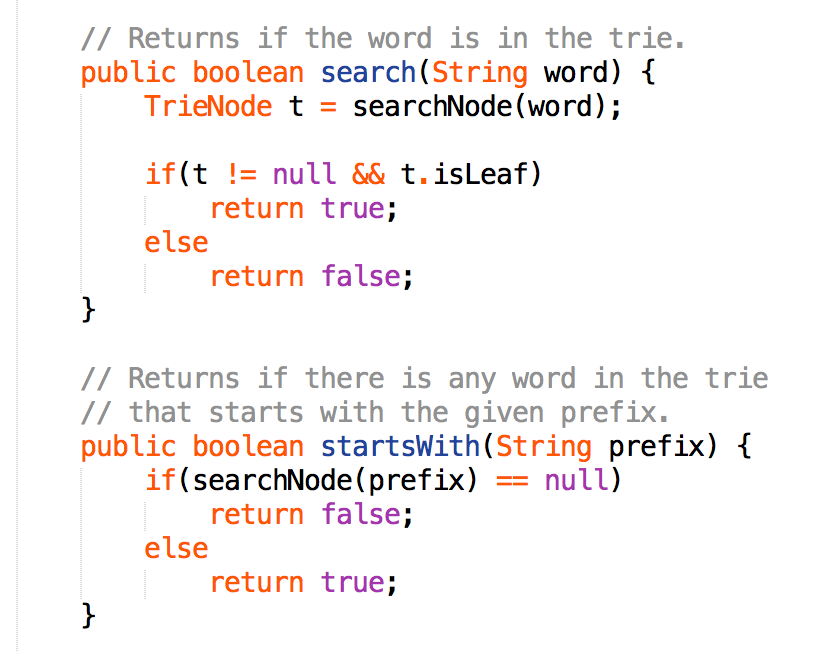
* Child of root: rotate child with root
* zig-zag: inside, use zig-zag rotation (same as AVL double rotation)
* zig-zig: outside, use zig-zig rotation
* A red-black tree and a splay tree can serve the same purpose. They are both self-balancing BSTs with O(logn) complexity for insertion, deletion, traversal.
* **B-Tree / B+-Tree**
* B-tree is a M-ary search tree uses M-1 keys for M branches.
* It has height of logM N.
* Unlike BST, B-tree is optimized for systems that read and write large blocks of data (B-Tree: disk-based solution; AVL Tree: memory-based solution )
* The B+-tree stores data only in the leaf nodes

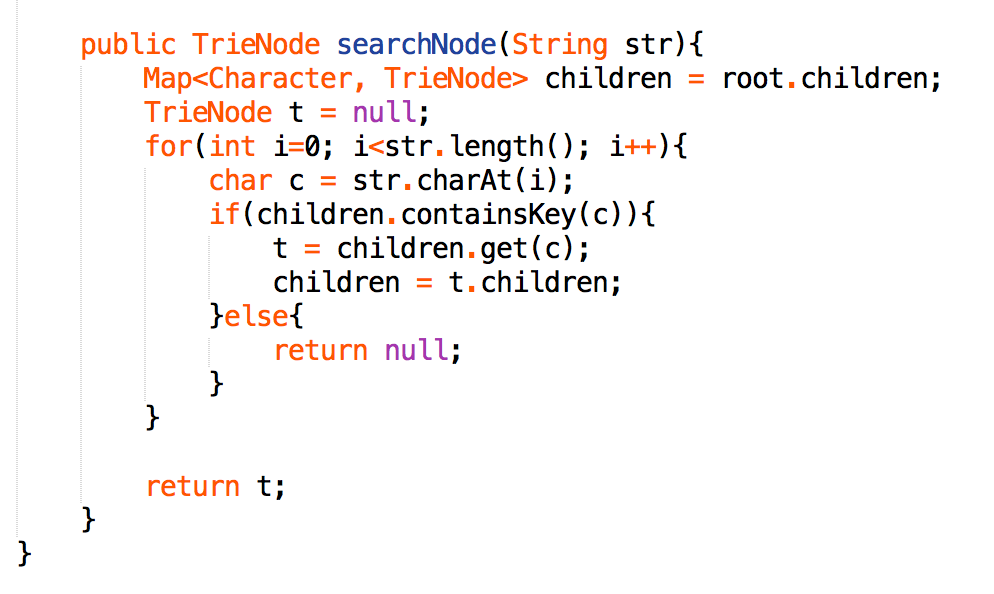
|  |  |  |
| --- | --- | --- |
|  | Average | Worst case |
| **Space** | O(*n*) | O(*n*) |
| **Search** | O(log *n*) | O(log *n*) |
| **Insert** | O(log *n*) | O(log *n*) |
| **Delete** | O(log *n*) | O(log *n*) |

* **Trie Tree**
* used to store a [dynamic set](https://en.wikipedia.org/wiki/Set_(abstract_data_type)) or associative array where the keys are usually [strings](https://en.wikipedia.org/wiki/String_(computer_science)).
* instead, its position in the tree defines the key with which it is associated node
* The complexity of creating a trie is O(W\*L)
* Search a word with size=L takes O(L)
* Java Implement
* Can optimize by use array instead of HashMap when limited character/number



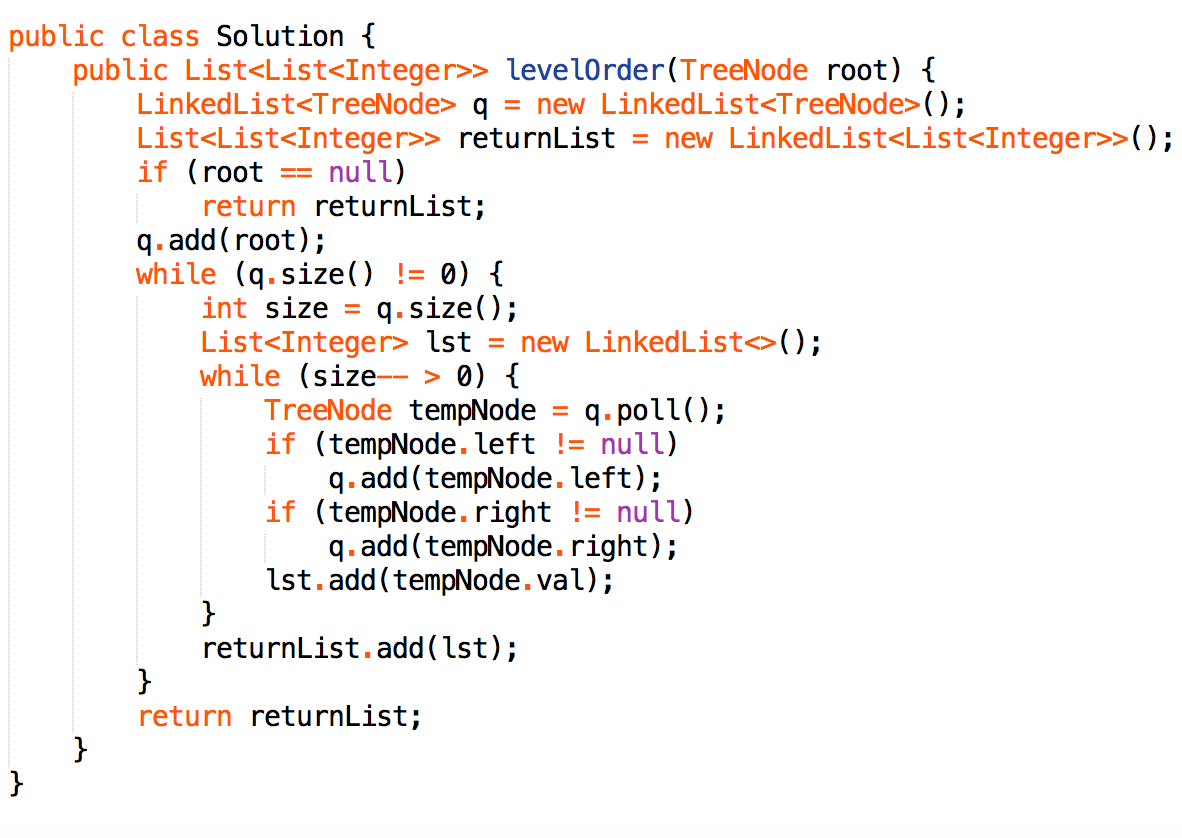
****



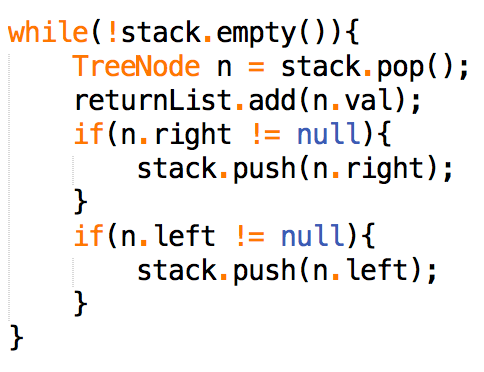


1. **Trees Problems**

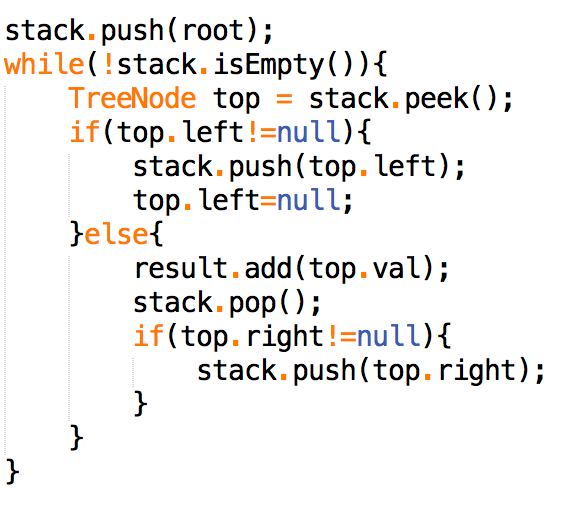
* **Level traversal (BFS)**

****

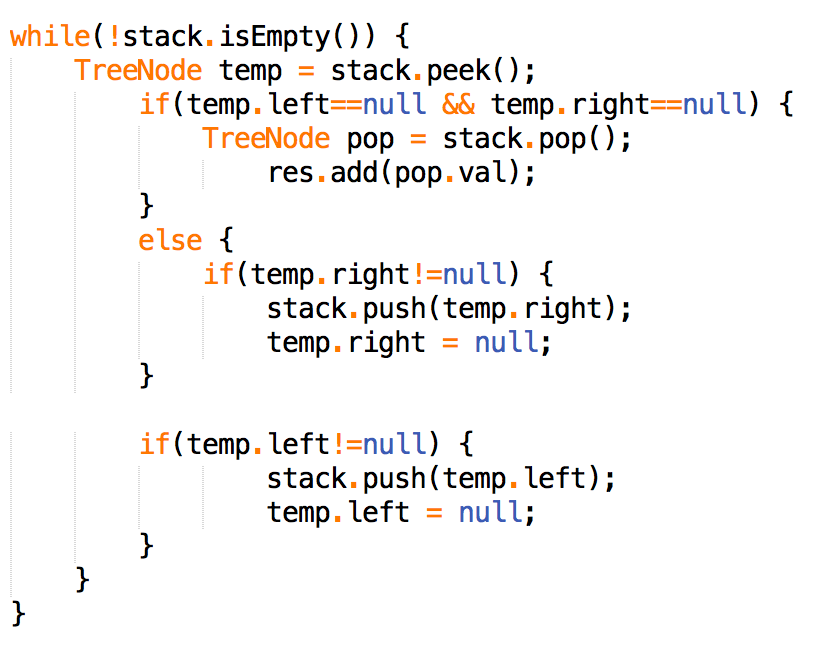
* **Preorder traversal (binary tree & iterative)**



* **Inorder traversal (binary tree & iterative)**



* **Postorder traversal (binary tree & iterative)**



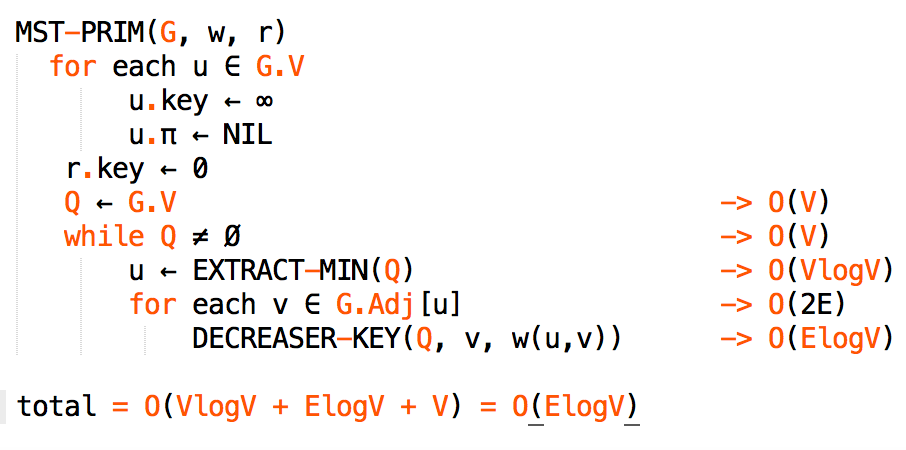
* **Tree Problems**
* First Common Ancestor
* Is it a BST? Easy for BST
* Is it to use Hash Map? Need extra pointers to parent
* Use Pre-order traversal
* TreeNode left = LCA(root.left, p, q)
* TreeNode right = LCA(root.right, p, q)
* If(left != null && right != null) return root;//already found
* else //return non-null node
* Path Sum
* From root to leaf? Easy with recursion
* From any node to leaf? Return all path? Use Backtracking
* Path.add(root.val)
* If (isLeaf) for(each valid sub path) add to res
* Else recursive on left & right node
* Path.remove(path.size() - 1)
* Balance binary tree
* use Post-order traversal
* return height of sub-tree

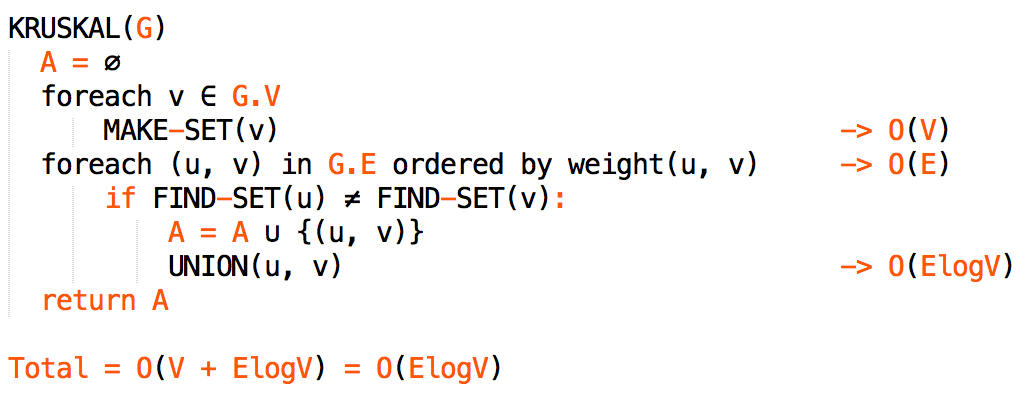
(DP are rarely used in tree, cause usually nodes are accessed only once)

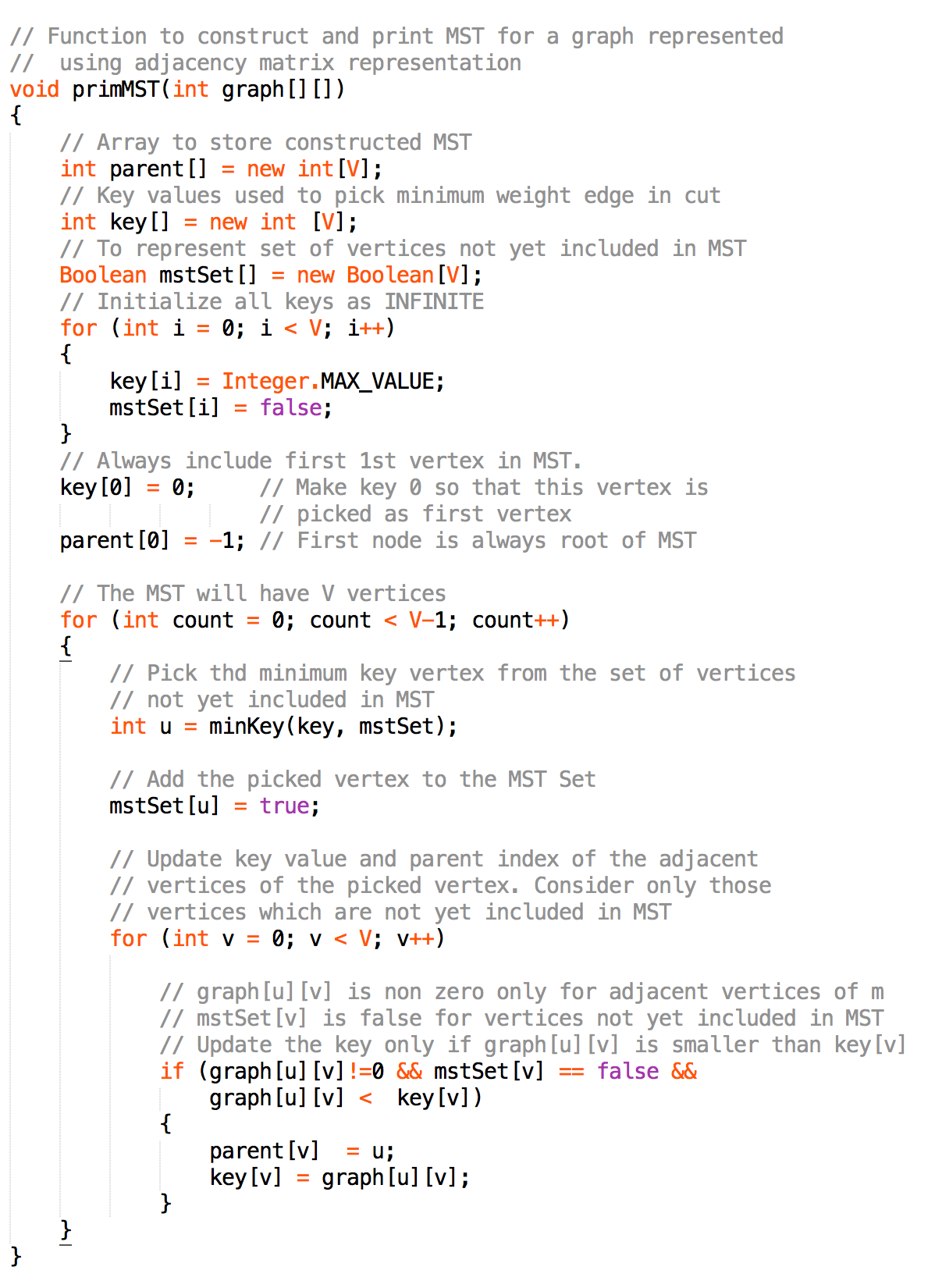
* Or -1 represent not balanced
* Tree compare
* Use level traversal
* Tree to linked list
* To dynamic initiate head of linked list, declare ‘Flag/ null, then use if (keyFactor) within iteration
* To form a cycle linked list, keep updating head.next to current tail.prev

1. **Graph**

* **Graph representation**
* **objects and pointer**
* commonly used for object oriented implementations
* more readable and convenient for object oriented programing
* **Matrix**
* commonly used for dense graphs where index access is necessary.
* can represent a un/directed and weighted structure with this.
* Add an edge: O(n)/O(1) depend on if new vertex
* Check if 2 vertex adjacent: O(1)
* Iterate through all adjacent vetex to p: O(n)
* **Adjacency list**
* used for representing sparse graphs, web graph is sparse.
* implement this using a HashMap<Vertex, List<Vertex>>
* Add an edge: O(1)
* Check if 2 vertex adjacent: O(k)
* Iterate through all adjacent vetex to p: O(k)
* **Minimum Spanning Tree**
* **Prim**
* Complexity: O(ElogV) with Binary Heap
* Step 1: always keep a connected component, starting with a single vertex.
* Step 2: look at all edges from the current component to other vertices and find the smallest among them
* Step 3: then add the neighboring vertex to the component
* Step 4: In N-1 steps, every vertex would be merged to the current one if we have a connected graph.
* Pseudo code



* **Krustral**
* Complexity: O(ElogV)
* Step 1: At each stage, look at the globally smallest edge that does not create a cycle in the current forest.
* Step 2: Such an edge must necessarily merge two trees in the current forest into one.
* Step 3: In N-1 steps, they would all have merged into one.
* Pseudo code
* Prim Java Implementation



* **Shortest Path Tree**
* find shortest paths between nodes in a graph.
* **Dijkstra's algorithm**
* It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.
* **Difference** between **Dijkstra** and **Prim**
* Prim **stores** a minimum cost edge: dist[v] = graph[u][v]; Dijkstra stores the total cost from source vertex to the current vertex: dist[v] = dist[u] + graph[u][v]
* Prim's algorithm works on undirected graphs only; Dijkstra works for both
* **Complexity**: Use adjacency list can be reduced to O(E log V) with binary heap
* **Difference** between **Dijkstra** and **A\***
* Dijkstra is a special case for A\* (when the heuristics is zero)
* A\* has two function
* g(x): real cost function, same as Dijkstra
* h(x): heuristic function, approximate cost from node x to goal node.
* The total cost of each node is calculated by f(x)=g(x)+h(x)
* Better efficiency if heuristic function is good
* Java Implementation

