**Common data structures in Java**

**Arrays** are the most commonly used data structure. Arrays are of fixed size, indexed, and all containing elements are of the same type (i.e. a homogenous collection). For example, storing employee details just read from the database as EmployeeDetail[ ], converting and storing a string as a byte array for further manipulation or processing, etc. Wrap an array in a class to protect it from being inadvertently altered. This would be true for other data structures as well.

**Lists** are known as arrays that can grow. These data structures are generally backed by a fixed sized array and they re-size themselves as necessary. A list can have duplicate elements. For example, adding new line items to an order that stores its line items as a list, removing all expired products from a list of products, etc. Initialize them with an appropriate initial size to minimize the number of re-sizes.

**Sets** are like lists but they do not hold duplicate elements. Sets can be used when you have a requirement to store unique elements.

**Stacks** allow access to only one data item, which is the last item inserted (i.e. Last In First Out - LIFO). If you remove this item, you can access the next-to-last item inserted, and so on. The LIFO is achieved through restricting access only via methods like peek( ), push( ), and pop( ). This is useful in many programing situations like parsing mathematical expressions like (4+2) \* 3, storing methods and exceptions in the order they occur, checking your source code to see if the brackets and braces are balanced properly, etc.

The LIFO access mechanism used by a stack has many practical uses. For example, Evaluation of expressions / syntax Parsing, validating and parsing XML, undo sequence in a text editor, pages visited history in a web browser, etc. Java interview questions and answers on stack data structure.  
**Queues** are somewhat like a stack, except that in a queue the first item inserted is the first to be removed (i.e. First In First Out – FIFO). The FIFO is achieved through restricting access only via methods like peek( ), offer( ), and poll( ). For example, waiting in a line for a bus, a queue at the bank or super market teller, etc.

**LinkedLists** are data structures made of nodes, where each node contains data and a reference to the next node, and possibly to the previous node as well for a doubly linked list. For example, a stack or queue can be implemented with a linked list or a doubly linked list because you can insert and delete at both ends. There would also be other situations where data will be frequently inserted and deleted from the middle. The Apache library provides a TreeList implementation, which is a good replacement for a LinkedList as it performs much better than a LinkedList at the expense of using a little more memory. This means a LinkedList is rarely a good choice of

implementation.

**ArrayList** is a good general purpose list implementation. An ArrayList is faster than a TreeList for most operations except inserting and removing in the middle of the list. A TreeList implementation utilizes a tree structure internally to ensure that all insertions and removals are O(log n). This provides much faster performance than both an ArrayList and a LinkedList where elements are inserted and removed repeatedly from anywhere in the list.

class Link {   
private int id; // data   
private String name; // data   
private Link next; // reference to next link   
}

**HashMaps** are amortized constant-time access data structures that map keys to values. This data structure is backed by an array. It uses a hashing functionality to identify a location, and some type of collision detection algorithm is used to handle values that hash to the same location. For example, storing employee records with employee number as the key, storing name/value pairs read from a properties file, etc. Initialize them with an appropriate initial size to minimize the number of re-sizes.

Trees are the data structures that contain nodes with optional data elements and one or more child elements, and possibly each child element references the parent element to represent a hierarchical or ordered set of data elements. For example, a hierarchy of employees in an organization, storing the XML data as a hierarchy, etc. If every node in a tree can have utmost 2 children, the tree is called a binary tree. The binary trees are very common because the shape of a binary tree makes it easy to search and insert data. The edges in a tree represent quick ways to navigate from node to node.

Java does not provide an implementation for this but it can be easily implemented as shown below. Just make a class Node with an ArrayList holding links to other Nodes.

**Graphs** are data structures that represent arbitrary relationships between members of any data sets that can be represented as networks of nodes and edges. A tree structure is essentially a more organized graph where each node can only have one parent node. Unlike a tree, a graph's shape is dictated by a physical or abstract problem. For example, nodes (or vertices) in a graph may represent cities, while edges may represent airline flight routes between the cities.

To make a Java graph class, you will have to work out the way in which the information can be stored and accessed. A graph will be using some of the data structures mentioned above. The Java API does not provide an implementation for this. But there are a number of third party libraries like JUNG, JGraphT, and JDSL (does not seem to support generics).

**Program to Represent Tree in java**

class **Node** {

private String name;

private List<Node> children = new ArrayList<Node>( );

private Node parent;

public Node getParent( ) {

return parent;

}

public void setParent(Node parent) {

this.parent = parent;

}

public Node(String name) {

this.name = name;

}

public List<Node> getChildren(){

return children;

}

public Node addChild(Node child) {

children.add(child);

return child;

}

public void removeChild(Node child) {

children.remove(child);

}

public String toString( ) {

return name;

}

}

public class TestMap{

public static void main(String[] args){

Node root = new Node("A");

Node n1 = root.addChild(new Node("B"));

Node n2 = root.addChild(new Node("C"));

Node n3 = root.addChild(new Node("D"));

Node n11 = n1.addChild(new Node("E"));

Node n21 = n2.addChild(new Node("F"));

root.removeChild(n2);

printTree(root," ");

}

private static void printTree(Node root, String app){

System.out.println(app+root);

root.getChildren().forEach(each->printTree(each,app+app));

}

}

**Generic Tree data structure to accept values**

class **Node<T>** {

private T data = null;

private List<Node<T>> children = new ArrayList<>();

private Node<T> parent = null;

public Node(T data) {

this.data = data;

}

public Node<T> addChild(Node<T> child) {

child.setParent(this);

this.children.add(child);

return child;

}

/\*public void addChildren(List<Node<T>> children) {

children.forEach(each -> each.setParent(this));

this.children.addAll(children);

}\*/

public List<Node<T>> getChildren() {

return children;

}

public T getData() {

return data;

}

public void setData(T data) {

this.data = data;

}

private void setParent(Node<T> parent) {

this.parent = parent;

}

public Node<T> getParent() {

return parent;

}

//to Get parent root node

public Node getRoot() {

if(parent == null){

return this;

}

return parent.getRoot();

}

//Deleting a Node

public void deleteNode() {

if (parent != null) {

int index = this.parent.getChildren().indexOf(this);

this.parent.getChildren().remove(this);

for (Node<T> each : getChildren()) {

each.setParent(this.parent);

}

this.parent.getChildren().addAll(index, this.getChildren());

} else {

deleteRootNode();

}

this.getChildren().clear();

}

public Node<T> deleteRootNode() {

if (parent != null) {

throw new IllegalStateException("deleteRootNode not called on root");

}

Node<T> newParent = null;

if (!getChildren().isEmpty()) {

newParent = getChildren().get(0);

newParent.setParent(null);

getChildren().remove(0);

for (Node<T> each : getChildren()) {

each.setParent(newParent);

}

newParent.getChildren().addAll(getChildren());

}

this.getChildren().clear();

return newParent;

}

public String toString( ) {

return data.toString();

}

}

public class testTree {

public static void main(String[] args) {

Node<Integer> root = createTree();

printTree(root, " ");

}

private static Node<Integer> createTree() {

Node<Integer> root = new Node<>(100);

Node<Integer> node1 = root.addChild(new Node<Integer>(2));

Node<Integer> node11 = node1.addChild(new Node<Integer>(5));

Node<Integer> node111 = node11.addChild(new Node<Integer>(11));

Node<Integer> node112 = node11.addChild(new Node<Integer>(12));

Node<Integer> node12 = node1.addChild(new Node<Integer>(6));

Node<Integer> node2 = root.addChild(new Node<Integer>(3));

Node<Integer> node21 = node2.addChild(new Node<Integer>(7));

Node<Integer> node22 = node2.addChild(new Node<Integer>(8));

Node<Integer> node3 = root.addChild(new Node<Integer>(4));

Node<Integer> node31 = node3.addChild(new Node<Integer>(9));

Node<Integer> node32 = node3.addChild(new Node<Integer>(10));

Node<Integer> node311 = node31.addChild(new Node<Integer>(13));

Node<Integer> node312 = node31.addChild(new Node<Integer>(14));

Node<Integer> node321 = node32.addChild(new Node<Integer>(15));

Node<Integer> node322 = node32.addChild(new Node<Integer>(16));

Node<Integer> node323 = node32.addChild(new Node<Integer>(17));

System.out.println(node322.getRoot());

node31.deleteNode();

return root;

}

private static <T> void printTree(Node<T> node, String appender) {

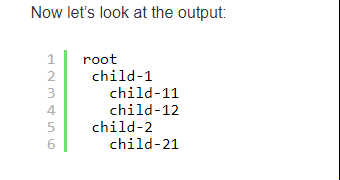
System.out.println(appender + node.getData());

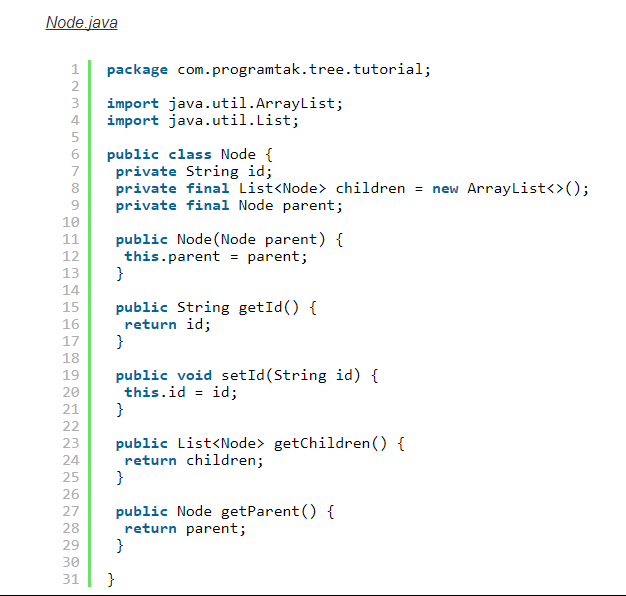
node.getChildren().forEach(each -> printTree(each, appender + appender));

}

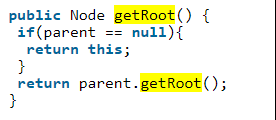
}

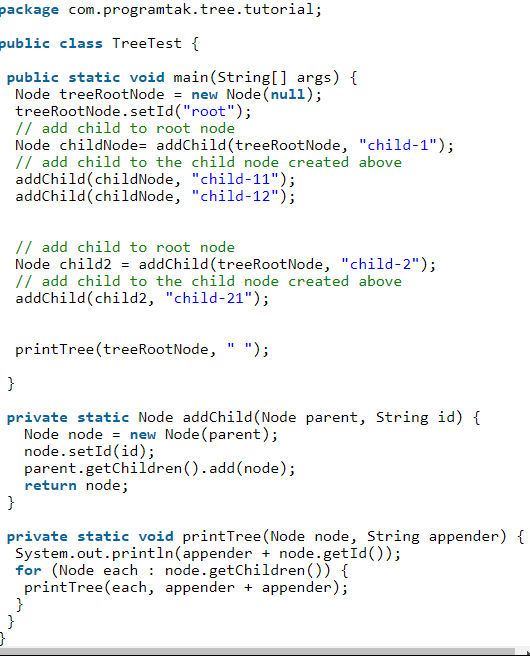
**Modified version of Tree Data Structure**



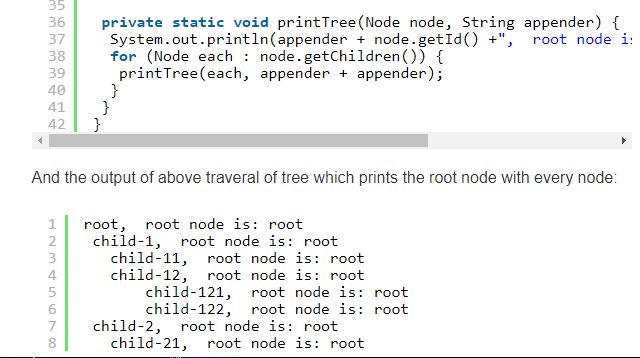


Add below method to Nod Class to get the root node





Updated printTree() with getNode()



**Deleting a Node from Tree**

