**LRU Cache**

// Implementation of custom queue to handle all LRU calls and methods;

public class LRUCache {

public class Node{

int key;

int value;

Node prev;

Node next;

public Node(int k,int v) {

key=k;

value=v;

next=prev=null;

}

// getters

public int getKey() {

return key;

}

public int getValue() {

return value;

}

public boolean setValue(int v) {

value=v;

return true;

}

}

Node head;

Node tail;

Map<Integer,Node> map; // to check for the value if present in the cache;

// NODE is given as value for easier O(1) get functions.

int capacity;

// int size;

public LRUCache(int capacity) {

this.capacity=capacity;

head=tail=null;

// size=0;

map=new HashMap<>(capacity); // stores the key of the cache and the node to which it points

}

public int get(int key) {

/\* The value to be fetched may be at

HEAD of the queue

MIDDLE

Tail

\*/

Node cur=map.get(key);

if(cur!=null) {

System.out.println(cur.getKey());

System.out.println("map Size: " + map.size());

// check location of the node;

if(cur.prev==null) { // it means the head

return cur.getValue();

}else if(cur.next==null) { // it means the tail

cur.prev.next=null;

tail= cur.prev;

}else { //it is in between

// remove the node from the middle and join the previous and next;

cur.prev.next=cur.next;

cur.next.prev=cur.prev;

}

cur.next=head; //join the current element at the head

cur.prev=null;

head.prev= cur;

head=cur;

return head.getValue();

}else {

return -1;

}

}

public void put(int key, int value) {

/\*

check

1. if the capacity is null

2. if the capacity is reached

3. if the cache already has the key but value differs.

4. else continue with the size < capacity content

\*/

Node cur=new Node(key,value);

if(head==null) { // when the cache is empty

head=tail=cur;

map.put(key,cur);

// size+=1;

return;

}else if (map.containsKey(key)) { // when cache already contains the key

cur=map.get(key); // Get the node from the map

cur.setValue(value); // update the old value with the new value

get(key); // call get(key) to push it to the the front of the queue

return;

} else if (map.size()==capacity) { // when the cache is full

// System.out.println(" when capacity == size");

map.remove(tail.getKey());

// System.out.println("tail value: " + tail.getKey());

if(tail.prev!=null) {

tail=tail.prev;

// System.out.println("tail value: " + tail.getKey());

tail.next=null;

}

// set current node as the head of the queue

head.prev=cur;

cur.next=head;

head=cur;

map.put(key,cur);

//head.prev=null;

return;

} else { // if the size < capacity

map.put(key, cur); // update the map to tell the chache has the new value

cur.next=head; // put the new node in front of the queue

// cur.prev=null;

head.prev=cur;

head=cur;

// size+=1;

return;

}

}

}

/\*\*

\* Your LRUCache object will be instantiated and called as such:

\* LRUCache obj = new LRUCache(capacity);

\* int param\_1 = obj.get(key);

\* obj.put(key,value);

\*/

**Reflections**

Design to sort a list of objects of a **Contact** class having multiple data members

The below design specifies two functionalities

1. To compare and sort an array of **objects depending on multiple data members at the same time**, one after the other.
2. To compare and sort the array of objects depending on the **field specified by the end user** during run time.

**CONTACTS CLASS:**

**package sortDesign;**

**import java.util.\*;**

**public class Contact {**

**int c\_ID;**

**String name;**

**int age;**

**Date startDate;**

**String hobby;**

**// For simplicity purposes only 5 data members are considered, the code words for n number of data members too**

**public Contact(int c\_ID, String name, int age, Date startDate, String hobby) {**

**super();**

**this.c\_ID = c\_ID;**

**this.name = name;**

**this.age = age;**

**this.startDate = startDate;**

**this.hobby = hobby;**

**}**

**public int getC\_ID() {**

**return c\_ID;**

**}**

**public void setC\_ID(int c\_ID) {**

**this.c\_ID = c\_ID;**

**}**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**public int getAge() {**

**return age;**

**}**

**public void setAge(int age) {**

**this.age = age;**

**}**

**public Date getStartDate() {**

**return startDate;**

**}**

**public void setStartDate(Date startDate) {**

**this.startDate = startDate;**

**}**

**public String getHobby() {**

**return hobby;**

**}**

**public void setHobby(String hobby) {**

**this.hobby = hobby;**

**}**

**public String toString() {**

**return String.*format*("%d\t%s\t%d\t%s\t%s",c\_ID,name,age,startDate.toString(),hobby);**

**}**

**}**

**// Chained comparator to sort the objects various data members at the same time**

**package** sortDesign;

**import** java.util.\*;

**public** **class** EmbededComparator **implements** Comparator<Contact> {

List<Comparator<Contact>> comparatorList;

@SafeVarargs

**public** EmbededComparator(Comparator<Contact>... comparators){

comparatorList=Arrays.*asList*(comparators);

}

// multiple comparators

@Override

**public** **int** compare(Contact c1, Contact c2) {

// **TODO** Auto-generated method stub

// move to the next comparator only when it returns 0

**for**(Comparator<Contact> comparator : comparatorList){

**int** result=comparator.compare(c1, c2);

**if**(result!=0)

**return** result;

}

**return** 0;

}

}

**// Execution class to retrieve all the data and execute both functionalities**

**package** sortDesign;

**import** java.lang.reflect.Field;

**import** java.util.ArrayList;

**import** java.util.Calendar;

**import** java.util.Collections;

**import** java.util.Comparator;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Random;

**import** java.util.Scanner;

// utility function to compare depending on the field

**class** ContactsComparator **implements** Comparator<Contact> {

Field sortField;

**public** ContactsComparator(Field sf) {

sortField = sf;

// **TODO** Auto-generated constructor stub

}

**public** **int** compare(Contact c1, Contact c2) {

**try** {

//only comparable has the compareTo function inside

Comparable f1 = (Comparable)sortField.get(c1);

Comparable f2 = (Comparable)sortField.get(c2);

**return** f1.compareTo(f2);

}

**catch**(IllegalAccessException e) {

**return** 0;

}

}

}

**public** **class** SortContact {

**public** **static** **void** main(String args[]) **throws** IllegalAccessException{

List<Contact> contactList=**new** ArrayList<>();

Random rand = **new** Random();

//populating the list with test data

**for**(**int** i=0;i<10;i++){

**char** a= (**char**)(rand.nextInt(25) + 97);

**char** b=(**char**)(rand.nextInt(25) + 97);

String name=String.*valueOf*(a)+"Name";

String hob=String.*valueOf*(b)+"Hobby";

contactList.add(**new** Contact(i,name,rand.nextInt(100), (Calendar.*getInstance*().getTime()),hob));

}

// check display before sorting

**for**(Contact c:contactList){

System.***out***.println(c.toString());

}

// to sort the contacts with respect to id,name,hobby,

// EmbededComparator ec=new EmbededComparator(

//// Give all the different types of comparators on which the list has to be sorted

//

//

//// compare by id

// new Comparator<Contact>(){

// @Override

// public int compare(Contact c1,Contact c2){

return c1.getC\_ID()-c2.getC\_ID();

}

// };

//// compare by name

// new Comparator<Contact>(){};

//// compare by age

// new Comparator<Contact>(){};

//// compare by date

// new Comparator<Contact>(){};

//

//

// );

// Collections.sort(contactList,new EmbededComparator(new

// ));

Contact c = contactList.get(0);

// Determine the class to which the object belongs to

Class<?> objClass = c.getClass();

// gets the declared fields in the specified class

Field[] fields = objClass.getDeclaredFields();

// to match the user input to the corresponding field

Map<String, Field> fieldNameToField = **new** HashMap<>();

System.***out***.println("Enter the field to be sorted with");

**for**(Field field : fields) {

fieldNameToField.put(field.getName(), field);

System.***out***.println(field.getName());

}

Scanner s = **new** Scanner(System.***in***);

String choice = s.nextLine();

Field sortField = fieldNameToField.get(choice);

Collections.*sort*(contactList, **new** ContactsComparator(sortField));

System.***out***.println("After sorting");

**for**(Contact cont :contactList){

System.***out***.println(cont.toString());

}

}

}

Serialize and de-serialize binary tree

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Codec {

// Encodes a tree to a single string.

public String serialize(TreeNode root) {

if(root==null) {

return null;

}

Stack<TreeNode> stack=new Stack<>();

stack.push(root);

StringBuilder sb=new StringBuilder();

while(!stack.isEmpty()){

TreeNode top=stack.pop();

if(top!=null){

sb.append(top.val+",");

stack.push(top.right);

stack.push(top.left);

}

else{

sb.append("#,");

}

}

return sb.toString().substring(0,sb.length()-1);

}

// Decodes your encoded data to tree.

public TreeNode deserialize(String data) {

if(data==null) {

return null;

}

int[]t={0};

// int k=0;

String[] temp=data.split(",");

return helper(temp,t);

}

public TreeNode helper(String[] trims,int[] k) {

if(trims[k[0]].equals("#")) {

return null;

}

TreeNode node=new TreeNode(Integer.parseInt(trims[k[0]]));

k[0]=k[0]+1;

node.left=helper(trims,k);

k[0]=k[0]+1;

node.right=helper(trims,k);

// if(trims[k].equals("#")) {

// return null;

// }

// TreeNode node=new TreeNode(Integer.parseInt(trims[k]));

// node.left=helper(trims,k++);

// node.right=helper(trims,k++);

return node;

}

}

// Your Codec object will be instantiated and called as such:

// Codec codec = new Codec();

// codec.deserialize(codec.serialize(root));

**Edit Distance**

**Given two words *word1* and *word2*, find the minimum number of steps required to convert *word1* to *word2*. (each operation is counted as 1 step.)**

**You have the following 3 operations permitted on a word:**

**a) Insert a character  
b) Delete a character  
c) Replace a character**

public int minDistance(String word1, String word2) {

// return helper(word1,word2,word1.length(),word2.length());

int[][] dp=new int[word1.length()+1][word2.length()+1];

for(int i=0;i<=word1.length();i++) {

dp[i][0]=i;

}

for(int i=0;i<=word2.length();i++) {

dp[0][i]=i;

}

for(int i=0;i<word1.length();i++) {

for(int j=0;j<word2.length();j++) {

if(word1.charAt(i)==word2.charAt(j)){

dp[i+1][j+1]=dp[i][j];

}

else {

// dp[i+1][j+1]=1+(min(dp[i+1][j],dp[i][j],dp[i][j+1]));

int a=dp[i+1][j];

int b=dp[i][j];

int c=dp[i][j+1];

dp[i+1][j+1]=1+(a<b?(a<c?a:c):(b<c)?b:c);

}

}

}

return dp[word1.length()][word2.length()];

}