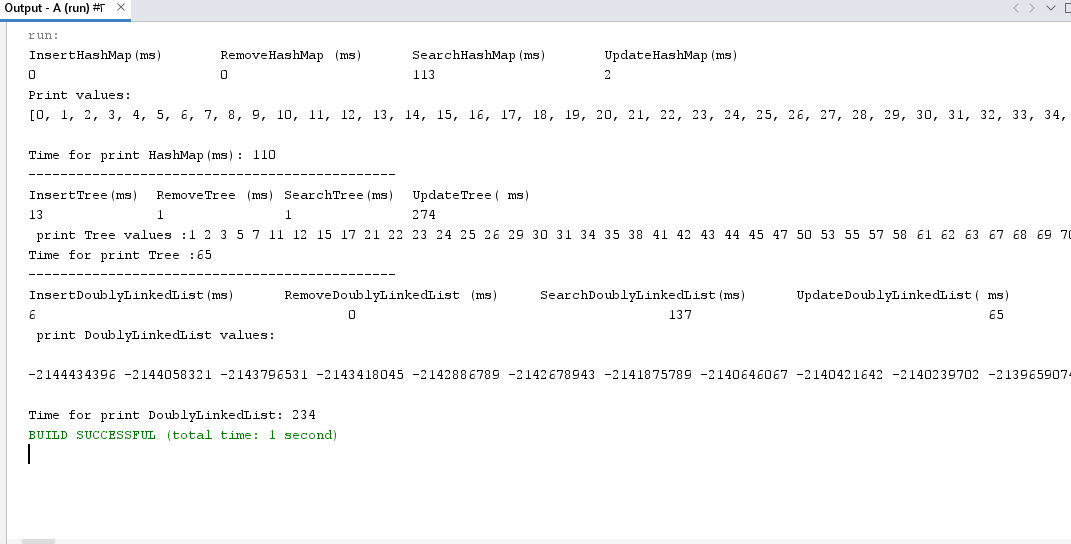
Assignment

Output:





import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.HashMap;

import java.util.HashSet;

import java.util.LinkedHashMap;

import java.util.List;

import java.util.Map;

import java.util.Random;

import java.util.Scanner;

import java.util.Set;

public class HashMap1 {

public static void insertTree ( BST t){

Set<Integer> values = new HashSet<>();

Random random = new Random();

while (values.size() < 5000) {

int val = random.nextInt(10000) + 1;

if (!values.contains(val)) {

values.add(val);

t.insert(val);

}

}

}

public static void DeleteTree ( BST tree){

Random r= new Random();

int[] values = new int[5000];

for (int i = 0; i < 5000; i++) {

values[i] = r.nextInt();

}

for (int i=0 ;i<5000;i++) {

if ( tree.contains(values[i])){

tree.delete(values[i]);

}

}

}

public static void printTree ( BST tree){

System.out.println();

System.out.print(" print Tree values :");

tree.inOrder(tree.root);

System.out.println();

System.out.print("Time for print Tree :");

}

public static void searchTree ( BST tree){

Random r=new Random(5000);

int found=0;

int Notfound=0;

for (int i = 0; i < 5000; i++) {

int val = r.nextInt();

if (tree.search(val)) {

found++;

// System.out.println(val + " was found in the tree");

} else {

Notfound++;

// System.out.println(val + " was not found in the tree");

}

}

}

public static void UpdateTree ( BST tree){

Random r= new Random();

int[] values = new int[5000];

for (int i = 0; i < 5000; i++) {

values[i] = r.nextInt();

}

for ( int j=0;j<5000;j++ ) {

tree.updateElement(tree.root,tree.getvalues(tree.root),values[j] );

}

}

public static void removeHashMap(HashMap<Integer,Integer>hp) {

Random r= new Random();

int[] keys = new int[5000];

for (int i = 0; i < 5000; i++) {

keys[i] = r.nextInt();

}

for (int key : keys) {

if ( hp.containsKey(key))

hp.remove(key);

}

}

public static void printHashMap(HashMap<Integer,Integer>hp) {

System.out.println();

System.out.println("Print values:");

LinkedHashMap<Integer, Integer> sortedValue = new LinkedHashMap<>(hp);

for(int i=0;i<5000;i++){

sortedValue.put(hp.get(i),i);

}

System.out.print(sortedValue.keySet());

System.out.println("---------------------------");

System.out.println();

System.out.print("Time for print HashMap(ms): ");

}

public static void insertHashMap(HashMap<Integer, Integer> hp) {

Set<Integer> values = new HashSet<>();

Random random = new Random();

for (int i=0;i<5000;i++) {

int val = random.nextInt(10000) + 1;

if (!values.contains(val)) {

values.add(val);

hp.put(i, val);

}

}

}

public static void searchHashMap(HashMap<Integer, Integer> hp) {

Random r=new Random(5000);

int found=0;

int Notfound=0;

for (int i=0;i < 5000;i++){

if (hp.containsValue(r.nextInt())){

found++;

}

else{

Notfound++;

}

}

// System.out.println( " Find "+ found +" value");

// System.out.println( " not Find "+ Notfound +" value");

}

public static void updateHashMap(HashMap<Integer, Integer> hp) {

Random r= new Random();

int[] values = new int[5000];

for (int i = 0; i < 5000; i++) {

values[i] = r.nextInt();

}

for ( int j=0;j<5000;j++ ) {

hp.put(j,values[j] );

}

}

public static void insertDoublyLink(DoublyLinkedList1 DoubLinkList){

Set<Integer> values = new HashSet<>();

Random random = new Random();

while (values.size() < 5000) {

int val = random.nextInt(10000) + 1;

if (!values.contains(val)) {

values.add(val);

DoubLinkList.insert(val);

}

}

}

public static void removeDoublyLink(DoublyLinkedList1 DoubLinkList){

Random r= new Random();

int[] values = new int[5000];

for (int i = 0; i < 5000; i++) {

values[i] = r.nextInt();

}

for (int i = 0; i < 5000; i++) {

if (DoubLinkList.contains(values[i]))

DoubLinkList.remove(values[i]);

}

}

public static void searchDoublyLink(DoublyLinkedList1 DoubLinkList) {

Random ran=new Random ();

int countFound=0;

int countNotFound=0;

for (int i = 0; i < 5000; i++) {

int value = ran.nextInt();

if (DoubLinkList.contains(value)) {

countFound++;

} else {

countNotFound++;

}

}

// System.out.println(countFound + " exists in the list");

// System.out.println(countNotFound+ " does not exist in the list");

}

public static void UpdateDoublyLink(DoublyLinkedList1 DoubLinkList) {

Random r= new Random();

int[] values = new int[5000];

for (int i = 0; i < 5000; i++) {

values[i] = r.nextInt();

}

for (int j= 0; j < 5000; j++) {

DoubLinkList.update( j,values[j]);

}

}

public static void printDoublyLink (DoublyLinkedList1 DoubLinkList) {

System.out.println();

System.out.println(" print DoublyLinkedList values:");

System.out.println();

DoubLinkList.sort();

DoubLinkList.printNodes();

System.out.println();

System.out.println();

System.out.print("Time for print DoublyLinkedList: ");

}

public static void main(String[] args) {

Scanner in=new Scanner (System.in );

HashMap <Integer, Integer> hp=new HashMap <>();

BST tree =new BST();

System.out.println("InsertHashMap(ms)" +"\t"+"RemoveHashMap (ms)" +"\t"+"SearchHashMap(ms)" +"\t"+"UpdateHashMap(ms)");

int start= (int)System.currentTimeMillis();

insertHashMap (hp);

int end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.print("\t" +"\t"+"\t");

start= (int)System.currentTimeMillis();

removeHashMap(hp);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.print("\t" +"\t"+"\t");

start= (int)System.currentTimeMillis();

searchHashMap(hp);

end=(int)System.currentTimeMillis();

System.out.print((end- start)+"");

System.out.print("\t" +"\t"+"\t");

start= (int)System.currentTimeMillis();

updateHashMap (hp);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.print("\t" +"\t"+"\t");

start= (int)System.currentTimeMillis();

printHashMap (hp);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.println();

System.out.println("----------------------------------------------");

System.out.println("InsertTree(ms)" +"\t"+"RemoveTree (ms)" +"\t"+"SearchTree(ms)" +"\t"+"UpdateTree( ms)" +"\t" );

start= (int)System.currentTimeMillis();

insertTree ( tree);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.print("\t" );

System.out.print("\t" );

start= (int)System.currentTimeMillis();

DeleteTree(tree);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.print("\t" );

System.out.print("\t" );

start= (int)System.currentTimeMillis();

searchTree(tree);

end=(int)System.currentTimeMillis();

System.out.print((end- start)+"");

System.out.print("\t" );

System.out.print("\t" );

start= (int)System.currentTimeMillis();

UpdateTree (tree);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.print("\t" );

System.out.print("\t" );

start= (int)System.currentTimeMillis();

printTree ( tree);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.println( );

System.out.println("----------------------------------------------");

DoublyLinkedList1 DoubLinkList = new DoublyLinkedList1();

System.out.println("InsertDoublyLinkedList(ms)" +"\t"+"RemoveDoublyLinkedList (ms)" +"\t"+"SearchDoublyLinkedList(ms)" +"\t"+"UpdateDoublyLinkedList( ms)" +"\t" );

start= (int)System.currentTimeMillis();

insertDoublyLink (DoubLinkList);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.print("\t" +"\t"+"\t"+"\t"+"\t");

start= (int)System.currentTimeMillis();

// removeDoublyLink(DoubLinkList);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.print("\t" +"\t"+"\t"+"\t"+"\t");

start= (int)System.currentTimeMillis();

searchDoublyLink(DoubLinkList);

end=(int)System.currentTimeMillis();

System.out.print((end- start)+"");

System.out.print("\t" +"\t"+"\t"+"\t"+"\t");

start= (int)System.currentTimeMillis();

UpdateDoublyLink (DoubLinkList);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.print("\t" +"\t"+"\t");

start= (int)System.currentTimeMillis();

printDoublyLink (DoubLinkList);

end=(int)System.currentTimeMillis();

System.out.print((end- start));

System.out.println();

}

}

class Node {

int key;

Node l,r,parent;

public Node(int key) {

this.key = key;

}

}

class BST {

Node root;

public BST() {

root = null;

}

void insert(int k) {

root = insertRec(root, k);

}

Node root() {

return root;

}

Node insertRec(Node root, int k) {

if (root == null) {

root = new Node(k);

return root;

}

if (k < root.key) {

root.l = insertRec(root.l, k);

}

if (k > root.key) {

root.r = insertRec(root.r, k);

}

return root;

}

public boolean contains(int value) {

return contains(root, value);

}

private boolean contains(Node node, int value) {

if (node == null) {

return false;

}

if (node.key == value) {

return true;

}

if (value < node.key) {

return contains(node.l, value);

} else {

return contains(node.r, value);

}

}

public int getvalues (Node root) {

if (root == null) {

return 0;

}

getvalues(root.l);

getvalues(root.r);

return root.key;

}

public void updateElement(Node root, int oldValue, int newValue) {

if (root == null) {

return;

}

if (root.key == oldValue) {

root.key = newValue;

} else if (root.key> oldValue) {

updateElement(root.l, oldValue, newValue);

} else {

updateElement(root.r, oldValue, newValue);

}

}

void delete(int k) {

root = deleteRec(root, k);

}

Node deleteRec(Node root, int k) {

if (root == null) {

return null;

}

if (k < root.key) {

root.l = deleteRec(root.l, k);

} else if (k > root.key) {

root.r = deleteRec(root.r, k);

} else {

if (root.l == null) {

return root.r;

} else if (root.r == null) {

return root.l;

} else {

root.key = minVal(root.r);

root.r = deleteRec(root.r, root.key);

}

}

return root;

}

int minVal(Node root) {

int min = root.key;

while (root.l != null) {

min = root.l.key;

root = root.l;

}

return min;

}

void inOrder(Node root) {

if (root == null) {

return;

}

inOrder(root.l);

System.out.print (root.key+" ") ;

inOrder(root.r);

}

boolean search(int val) {

return searchRec(root, val);

}

boolean searchRec(Node current, int val) {

if (current == null) {

return false;

}

if (val == current.key) {

return true;

}

return val < current.key ? searchRec(current.l, val): searchRec(current.r, val);

}

}

class DoublyLinkedList1 {

class Node {

int val;

Node prev;

Node next;

public Node(int val) {

this.val = val;

this.prev = null;

this.next = null;

}

}

Node head;

Node tail;

int size;

public DoublyLinkedList1() {

this.head = null;

this.tail = null;

}

public boolean contains(int value) {

Node current = head;

while (current != null) {

if (current.val == value) {

return true;

}

current = current.next;

}

return false;

}

public void insert(int val) {

Node newNode = new Node(val);

if (tail != null) {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

} else {

head = newNode;

tail = newNode;

}

size++;

}

public void update( int index,int value) {

if (index < 0 || index >= size) {

throw new IndexOutOfBoundsException();

}

Node current = head;

for (int i = 0; i < index; i++) {

current = current.next;

}

current.val = value;

}

public void sort() {

if (size <= 1) {

return;

}

boolean swap = true;

while (swap) {

swap = false;

Node current = head;

while (current.next != null) {

if (current.val> current.next.val) {

int temp = current.val;

current.val = current.next.val;

current.next.val = temp;

swap= true;

}

current = current.next;

}

}

}

public void remove(int index) {

if (index < 0 || index >= size) {

throw new IndexOutOfBoundsException();

}

Node current = head;

for (int i = 0; i < index; i++) {

current = current.next;

}

if (current.prev != null) {

current.prev.next = current.next;

} else {

head = current.next;

}

if (current.next != null) {

current.next.prev = current.prev;

} else {

tail = current.prev;

}

}

public void printNodes() {

Node current = head;

if(head == null) {

System.out.println("Doubly linked list is empty");

return;

}

while(current != null) {

System.out.print(current.val + " ");

current = current.next;

}

}

}