

QUESTION ONE

Red-Black Trees

```
package edu.princeton.cs.algs4;

import java.util.NoSuchElementException;

public class RedBlackBST<Key extends Comparable<Key>, Value> {

    private static final boolean RED = true;
    private static final boolean BLACK = false;

    private Node root;    // root of the BST

    // BST helper node data type

    private class Node {

        private Key key;        // key
        private Value val;      // associated data
        private Node left, right; // links to left and right subtrees
        private boolean color;   // color of parent link
        private int size;        // subtree count

        public Node(Key key, Value val, boolean color, int size) {

            this.key = key;
            this.val = val;
            this.color = color;
            this.size = size;
        }
    }

    /**
     * Initializes an empty symbol table.
     */
    public RedBlackBST() {

    }

    /**
     * Node helper methods.
     */
}
```

```

*****/

// is node x red; false if x is null ?
private boolean isRed(Node x) {
    if (x == null) return false;
    return x.color == RED;
}

// number of node in subtree rooted at x; 0 if x is null
private int size(Node x) {
    if (x == null) return 0;
    return x.size;
}

/**
 * Returns the number of key-value pairs in this symbol table.
 * @return the number of key-value pairs in this symbol table
 */

public int size() {
    return size(root);
}

/**
 * Is this symbol table empty?
 * @return {@code true} if this symbol table is empty and {@code false} otherwise
 */

public boolean isEmpty() {
    return root == null;
}

/*****
 * * Standard BST search.
 *****/
/**

```

```

public Value get(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to get() is null");
    return get(root, key);
}

// value associated with the given key in subtree rooted at x; null if no such key
private Value get(Node x, Key key) {
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if    (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else      return x.val;
    }
    return null;
}

public boolean contains(Key key) {
    return get(key) != null;
}

/*****
 *
 * Red-black tree insertion.
 *****/
/  /**

public void put(Key key, Value val) {
    if (key == null) throw new IllegalArgumentException("first argument to put() is null");
    if (val == null) {
        delete(key);
        return;
    }

    root = put(root, key, val);
    root.color = BLACK;
    // assert check();
}

```

```

// insert the key-value pair in the subtree rooted at h
private Node put(Node h, Key key, Value val) {
    if (h == null) return new Node(key, val, RED, 1);
    int cmp = key.compareTo(h.key);
    if (cmp < 0) h.left = put(h.left, key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else h.val = val;
    // fix-up any right-leaning links
    if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right)) flipColors(h);
    h.size = size(h.left) + size(h.right) + 1;
    return h;
}

/*****
 *
 * Red-black tree deletion.
 *****/
/ /**
 * Removes the smallest key and associated value from the symbol table.
 * @throws NoSuchElementException if the symbol table is empty
 */
public void deleteMin() {
    if (isEmpty()) throw new NoSuchElementException("BST underflow");
    // if both children of root are black, set root to red
    if (!isRed(root.left) && !isRed(root.right))
        root.color = RED;
    root = deleteMin(root);
    if (!isEmpty()) root.color = BLACK;
    // assert check();
}

```

// delete the key-value pair with the minimum key rooted at h

```
private Node deleteMin(Node h) {  
    if (h.left == null)  
        return null;  
    if (!isRed(h.left) && !isRed(h.left.left))  
        h = moveRedLeft(h);  
    h.left = deleteMin(h.left);  
    return balance(h);  
}
```

```
public void deleteMax() {  
    if (isEmpty()) throw new NoSuchElementException("BST underflow");  
    // if both children of root are black, set root to red  
    if (!isRed(root.left) && !isRed(root.right))  
        root.color = RED;  
    root = deleteMax(root);  
    if (!isEmpty()) root.color = BLACK;  
    // assert check();  
}
```

** Red-black tree helper functions.*

/

// make a left-leaning link lean to the right

```
private Node rotateRight(Node h) {  
    // assert (h != null) && isRed(h.left);  
    Node x = h.left;  
    h.left = x.right;  
    x.right = h;  
    x.color = x.right.color;  
    x.right.color = RED;  
    x.size = h.size;  
    h.size = size(h.left) + size(h.right) + 1;
```

```

        return x;

    /* put() and delete with iteration */
    Map<String, String> map = new HashMap<String, String>() {
        {
            put("test", "test123");
            put("test2", "test456");
        }
    };

    for(Iterator<Map.Entry<String, String>> it = map.entrySet().iterator(); it.hasNext(); ) {
        Map.Entry<String, String> entry = it.next();
        if(entry.getKey().equals("test")) {
            it.remove();
        }
    }

    /* Use a real iterator */
    Iterator<RedBlackBST> it = map.keySet().iterator();
    while (it.hasNext())
    {
        it.next();
        if (BST)
            it.remove(); }

    public static void main(String[] args) {
        RedBlackBST<String, Integer> st = new RedBlackBST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        StdOut.println();
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }

```

```

        StdOut.println();
    }
}

/*****
****

```

This marks the end of the program. The iteration order of a value in the map is the iteration order of the key with which it used. Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics. Basically, the Creation of Iterator in Java involve the following steps that have been exclusively used in the code above:

1. The first step is to obtain an iterator to the beginning of the collection. ...
2. Next would be to set up a loop that makes a call to `hasNext( )` and then have the loop iterate as long as `hasNext( )` returns true.
3. Finally, within that loop, obtain each element by calling `next( )`.

```

****
****/

```

QUESTION TWO

Hash Tables

Part a: If you have an empty hash table with N slots, the probability that there is at least one collision when N keys are inserted is given by:

$$\alpha = n/N = \Theta(N)/N = O(1)$$

where $O(1)$ is the worst case running time

Algorithm `get(k)`

`i` $\leftarrow$ `h(k)`

`p` $\leftarrow$ 0

repeat

`c` $\leftarrow$ `A[i]`

if `c` = $\emptyset$

return null

else if `c.getKey ()` = `k`

return `c.getValue()`

else

`i` $\leftarrow$ (`i` + 1) mod `N`

`p` $\leftarrow$ `p` + 1

until `p` = `N`

return null

Part b:

```
package edu.princeton.cs.algs4;

public class SeparateChainingHashST<Key, Value> {
    private static final int INIT_CAPACITY = 4;
    private int n;                // number of key-value pairs
    private int m;                // hash table size
    private SequentialSearchST<Key, Value>[] st; // array of linked-list symbol tables
```

```

/**
 * Initializes an empty symbol table.
 */
public SeparateChainingHashST() {
    this(INIT_CAPACITY);
}

/**
 * Initializes an empty symbol table with {@code m} chains.
 * @param m the initial number of chains
 */
public SeparateChainingHashST(int m) {
    this.m = m;
    st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[m];
    for (int i = 0; i < m; i++)
        st[i] = new SequentialSearchST<Key, Value>();
}

// resize the hash table to have the given number of chains,
// rehashing all of the keys
private void resize(int chains) {
    SeparateChainingHashST<Key, Value> temp = new SeparateChainingHashST<Key,
Value>(chains);
    for (int i = 0; i < m; i++) {
        for (Key key : st[i].keys()) {
            temp.put(key, st[i].get(key));
        }
    }
    this.m = temp.m;
    this.n = temp.n;
    this.st = temp.st;
}

```

// hash value between 0 and m-1

```
private int hash(Key key) {  
    return (key.hashCode() & 0x7fffffff) % m;  
}
```

*/***

** Returns the number of key-value pairs in this symbol table.*

** @return the number of key-value pairs in this symbol table*

**/*

```
public int size() {
```

```
    return n;
```

```
}
```

*/***

** Returns true if this symbol table is empty.*

** @return {@code true} if this symbol table is empty;*

** {@code false} otherwise*

**/*

```
public boolean isEmpty() {
```

```
    return size() == 0;
```

```
}
```

```
public boolean contains(Key key) {
```

```
    if (key == null) throw new IllegalArgumentException("argument to contains() is null");
```

```
    return get(key) != null;
```

```
}
```

```
public Value get(Key key) {
```

```
    if (key == null) throw new IllegalArgumentException("argument to get() is null");
```

```
    int i = hash(key);
```

```
    return st[i].get(key);
```

```
}
```

```

public void put(Key key, Value val) {
    if (key == null) throw new IllegalArgumentException("first argument to put() is null");
    if (val == null) {
        delete(key);
        return;
    }
    // double table size if average length of list >= 10
    if (n >= 10*m) resize(2*m);
    int i = hash(key);
    if (!st[i].contains(key)) n++;
    st[i].put(key, val);
}

public void delete(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to delete() is null");
    int i = hash(key);
    if (st[i].contains(key)) n--;
    st[i].delete(key);
    // halve table size if average length of list <= 2
    if (m > INIT_CAPACITY && n <= 2*m) resize(m/2);
}

// return keys in symbol table as an Iterable
public Iterable<Key> keys() {
    Queue<Key> queue = new Queue<Key>();
    for (int i = 0; i < m; i++) {
        for (Key key : st[i].keys())
            queue.enqueue(key);
    }
    return queue;
}

```

```

public static void main(String[] args) {
    SeparateChainingHashST<String, Integer> st = new SeparateChainingHashST<String,
Integer>();
    for (int i = 0; !StdIn.isEmpty(); i++) {
        String key = StdIn.readString();
        st.put(key, i);
    }
    // print keys
    for (String s : st.keys())
        StdOut.println(s + " " + st.get(s));
}

class SeparateChainingHashST {
    String web;
    SeparateChainingHashST (String w){
        web = w;
    }

    /* This is the Constructor, it
    * copies the values of one object
    * to the another object (the object
    * that invokes this constructor)
    */
    SeparateChainingHashST (SeparateChainingHashST je){
        web = je.web;
    }
    void disp(){
        }
    public static void main(String args[]){
        SeparateChainingHashST obj1 = new SeparateChainingHashST ();

        /* Passing the object as an argument to the constructor

```

```
    * This will invoke the copy constructor
    */
    SeparateChainingHashST = new SeparateChainingHashST (obj1);
    obj1.disp();
    obj2.disp();
}
}
}
```

Part c:

// hashCode for point2D

```
public void hashCode () {
    int MAX_VALUE = 1000;
    HashSet<Integer> hashCodes = new HashSet<Integer>();
    int collisions = 0;
    for (int x = -MAX_VALUE; x < MAX_VALUE; ++x) {
        for (int y = -MAX_VALUE; y < MAX_VALUE; ++y) {
            final int hashCode = ((x+MAX_VALUE)<<16)|((y+MAX_VALUE)&0xFFFF);
            if (hashCodes.contains(hashCode))
                collisions++;
            hashCodes.add(hashCode);
        }
    }
    System.out.println("Collisions: " + collisions + " // Buckets: " + hashCodes.size());
}
```

//hashCode for Interval

```
static int hashCode(Interval i) {return (int)((i >>> 32) + (int) i);}

static int hashCode(Double d) {
    long bits = Double.doubleToLongBits(d);
    return (int) (bits ^ (bits >>> 32));
}
```

// hashCode for Date

```
// Java code to demonstrate
// hashCode() function of Date class
import java.util.Date;
import java.util.Calendar;

public class Date_hashCode {
    // main method
    public static void main(String[ ] args)
```

```
{  
    // creating a Calendar object  
    Calendar c1 = Calendar.getInstance();  
    // set Month  
    // MONTH starts with 0 i.e. ( 0 - Jan)  
    c1.set(Calendar.MONTH, 05);  
    // set Date  
    c1.set(Calendar.DATE, 20);  
    // set Year  
    c1.set(Calendar.YEAR, 2020);  
    // creating a date object with specified time.  
    Date dateOne = c1.getTime();  
    System.out.println("Date: " + dateOne);  
    // Prints hash Code  
    System.out.println("HashCode: "  
        + dateOne.hashCode());  
}  
}
```

QUESTION THREE

Minimum Spanning Tree

PRIM-MST

The algorithm is as follows;

Input: Graph $G = (V, E)$, cost function w , root vertex r

Output: A minimum spanning tree of G

foreach $v \in V$ do

$\text{key}(v) \leftarrow \infty$

$\pi(v) \leftarrow \text{nil}$ /* π keeps track of the parent of a vertex in the tree. */

end

$\text{key}(r) \leftarrow 0$

$Q \leftarrow V$ /* Priority queue consists of vertices with their key values */

 while $Q \neq \emptyset$ do

$u \leftarrow \text{Extract-Min}(Q)$

 foreach v adjacent to u do

 if $v \in Q$ and $w(u, v) < \text{key}(v)$ then

$\pi(v) \leftarrow u$

$\text{key}(v) \leftarrow w(u, v)$

 end

 end

end

```

// A Java program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

import java.util.*;
import java.lang.*;
import java.io.*;

class PrimMST {
    // Number of vertices in the graph
    private static final int V = 5;

    // A utility function to find the vertex with minimum key
    // value, from the set of vertices not yet included in MST
    int minKey(int key[ ], Boolean mstSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index = -1;
        for (int v = 0; v < V; v++)
            if (mstSet[v] == false && key[v] < min) {
                min = key[v];
                min_index = v;
            }
        return min_index;
    }

    // A utility function to print the constructed MST stored in
    // parent[ ]
    void printMST(int parent[ ], int graph[ ][ ])
    {
        System.out.println("Edge \tWeight");
        for (int i = 1; i < V; i++)
            System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);
    }
}

```

// Function to construct and print MST for a graph represented

// using adjacency matrix representation

```
void primMST(int graph[ ][ ])
```

```
{
```

```
    // Array to store constructed MST
```

```
    int parent[ ] = new int[V];
```

```
    // Key values used to pick minimum weight edge in cut
```

```
    int key[ ] = new int[V];
```

```
    // To represent set of vertices not yet included in MST
```

```
    Boolean mstSet[ ] = new Boolean[V];
```

```
    // Initialize all keys as INFINITE
```

```
    for (int i = 0; i < V; i++) {
```

```
        key[i] = Integer.MAX_VALUE;
```

```
        mstSet[i] = false;
```

```
    }
```

```
    // Always include first 1st vertex in MST.
```

```
    key[0] = 0; // Make key 0 so that this vertex is
```

```
    // picked as first vertex
```

```
    parent[0] = -1; // First node is always root of MST
```

```
    // The MST will have V vertices
```

```
    for (int count = 0; count < V - 1; count++) {
```

```
        // Pick the minimum key vertex from the set of vertices
```

```
        // not yet included in MST
```

```
        int u = minKey(key, mstSet);
```

```
        // Add the picked vertex to the MST Set
```

```
        mstSet[u] = true;
```

```
        // Update key value and parent index of the adjacent
```

```
        // vertices of the picked vertex. Consider only those
```

```
        // vertices which are not yet included in MST
```

```
        for (int v = 0; v < V; v++)
```

```

        // graph[u][v] is non zero only for adjacent vertices of m
        // mstSet[v] is false for vertices not yet included in MST
        // Update the key only if graph[u][v] is smaller than key[v]
        if (graph[u][v] != 0 && mstSet[v] == false && graph[u][v] < key[v]) {
            parent[v] = u;
            key[v] = graph[u][v];
        }
    }

    // print the constructed MST
    printMST(parent, graph);
}

public static void main(String[] args)
{
    //The code below creates a Graph
    MST t = new MST();
    int graph[ ][ ] = new int[ ][ ] { { 0, 2, 0, 6, 0 },
                                         { 2, 0, 3, 8, 5 },
                                         { 0, 3, 0, 0, 7 },
                                         { 6, 8, 0, 0, 9 },
                                         { 0, 5, 7, 9, 0 } };

    // Print the solution
    t.primMST(graph);
}
}

```

QUESTION FOUR

Shortest Graphs

This problem can be solved using Dijkstra's shortest path algorithm. Minimum spanning tree won't give shortest path in all cases because source and destination should be reached. In the above scenario, one subset can be taken as source and the other as destination.

If we apply Dijkstra's shortest path algorithm, we can get a shortest path in $O(E + V \log V)$ time. The idea is to use Breadth-first search (BFS). One important observation about BFS is, the path used in BFS always has least number of edges between any two vertices. So if all edges are of same weight, we can use BFS to find the shortest path. For this problem, we can modify the graph and split all edges of weight two into two edges of weight one each. In the modified graph, we can use BFS to find the shortest path. In worst case, all edges are of weight two and we need to do $O(E)$ operations to split all edges, so the time complexity becomes $O(E) + O(V+E)$ which is $O(V+E)$. The algorithm consists of the below steps;

function Dijkstra (Graph,pseudocode):

for each vertex v in Graph:	// Initialization
dist[s] := infinity	// initial distance from source to vertex s is set to infinite
previous[s] := undefined	// Previous node in optimal path from source
dist[source] := 0	// Distance from source to source
Q := the set of all nodes in Graph	// all nodes in the graph are unoptimized - thus are in Q
while Q is not empty:	// main loop
t := node in Q with smallest dist[]	
remove t from Q	
for each neighbor s of u:	// where s has not yet been removed from Q.
alt := dist[t] + dist_between(s, t)	
if alt < dist[s]	// Reduce (s,t)
dist[s] := alt	
previous[s] := t	
return previous[]	