

```

set nu
set sw=4
set ts=4
set st=4
set bs=2
set cul
set ai
set ls=2
map <F5> gT
imap <F5> <ESC>gT
map <F6> gt
imap <F6> <ESC>gt
imap {<CR> {<CR><END><CR>}<UP><END>
au FileType cpp map <F9> <ESC>:w<CR>:!g++<Space>—Wall<Space>%&&./a.out<CR>
au FileType cpp imap <F9> <ESC>:w<CR>:!g++<Space>—Wall<Space>%&&./a.out<CR>
set encoding=UTF-8

```

code/.vimrc

```

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

public class Main{

    public static void main(String[] args){
        Scan scan = new Scan();
        int testcases = scan.nextInt();
        while(testcases — != 0){
            int n = scan.nextInt();
            Coordinate[] vertex = new Coordinate[n];
            for(int i=0;i<n;i++) vertex[i] = new Coordinate(scan.nextDouble(), scan.nextDouble());
            Arrays.sort(vertex);
            //     for(Coordinate c : vertex){
            //         System.out.println(c.x+" "+c.y);
            //     }
            Coordinate[] list = new Coordinate[n+1];
            int index = 0;
            for(int i=0;i<n;i++){
                while(index >= 2 && ABcrossAC(list[index-2], list[index-1], vertex[i]) <= 0) index--;
                list[index++] = vertex[i];
            }
            int half_point = index+1;
            for(int i=n-2;i>=0;i--){
                while(index >= half_point && ABcrossAC(list[index-2], list[index-1], vertex[i]) <= 0)
index--;
                list[index++] = vertex[i];
            }
            double result = 0.0;
            //System.out.println(list[0].x+" "+list[0].y);
            for(int i=1;i<index;i++){
                //System.out.println(list[i].x+" "+list[i].y);
                result += Math.sqrt((list[i].x-list[i-1].x)*(list[i].x-list[i-1].x) + (list[i].y-list[i-1].y)*(list[i].y-list[i-1].y));
            }
            System.out.println(result);
        }

        static double ABcrossAC(Coordinate A, Coordinate B, Coordinate C){
            return (B.x-A.x) * (C.x-A.x) - (B.y-A.y) * (C.y-A.y);
        }

        static class Coordinate implements Comparable<Coordinate>{

            double x,y;

```

```

Coordinate(double x, double y){
    this.x = x;
    this.y = y;
}

@Override
public int compareTo(Coordinate o){
    if(x < o.x) return -1;
    if(x > o.x) return 1;
    if(y < o.y) return -1;
    if(y > o.y) return 1;
    return 0;
}
}

}

class Scan implements Iterator<String>{

    BufferedReader buffer;
    StringTokenizer tok;

    Scan(){
        buffer = new BufferedReader(new InputStreamReader(System.in));
    }

    @Override
    public boolean hasNext(){
        while(tok == null || !tok.hasMoreElements()){
            try{
                tok = new StringTokenizer(buffer.readLine());
            }catch(Exception e){
                return false;
            }
        }
        return true;
    }

    @Override
    public String next(){
        if(hasNext()) return tok.nextToken();
        return null;
    }

    @Override
    public void remove(){
        throw new UnsupportedOperationException();
    }

    int nextInt(){
        return Integer.parseInt(next());
    }

    long nextLong(){
        return Long.parseLong(next());
    }

    double nextDouble(){
        return Double.parseDouble(next());
    }

    String nextLine(){

```

```

        if (hasNext()) return tok.nextToken("\n");
        return null;
    }
}

```

code/ConvexHull.java

```

/* build: O(VlogV), query: O(logV) */
#include <iostream>
#include <vector>
#include <cstdio>
#define MAX 50010

using namespace std;

int a[MAX][160]; /* 160 = log2(MAX/2) */
int parent[MAX], tin[MAX], tout[MAX];
int num, root, timestamp;
bool visit[MAX];
vector<int> adj[MAX];

int log2(int n) {
    int i = 0;
    while ((1<<i) <= n) ++i;
    return i - 1;
}

/* when x == y, it's be true */
bool ancestor(int x, int y) {
    return (tin[x] <= tin[y]) && (tout[x] >= tout[y]);
}

void dfs(int x, int px) {
    tin[x] = timestamp++;
    visit[x] = true;
    a[x][0] = px;
    for (int i = 1; i < log2(num); ++i) {
        a[x][i] = a[a[x][i-1]][i-1];
    }

    for (int i = 0; i < adj[x].size(); ++i) {
        int target = adj[x][i];
        if (!visit[target]) {
            parent[target] = x;
            dfs(target, x);
        }
    }
    tout[x] = timestamp++;
}

int lca(int x, int y) {
    if (ancestor(x, y)) return x;
    if (ancestor(y, x)) return y;

    for (int i = log2(num); i >= 0; --i) {
        if (!ancestor(a[x][i], y)) {
            x = a[x][i];
        }
    }
    return a[x][0];
}

int main () {
    timestamp = 0;

    /* init */

```

```

for (int i = 0; i < num; ++i) {
    parent[i] = i;
    visit[i] = false;
    adj[i].clear();
}

for (int i = 0; i < num-1; ++i) {
    int x, y;
    scanf("%d%d", &x, &y);
    adj[x].push_back(y);
    adj[y].push_back(x);
}

dfs(0, 0);
cin >> x >> y;
cout << lca(x, y);
}

```

code/double\_lca.cpp

```

#include <iostream>
#include <vector>
#include <cstring>
#include <cstdio>

using namespace std;

class HashMap {
private:
    int MAX;
    vector<int> table;
    vector<bool> selector;
    vector<long long> values;
    int hashing1(int);
    int hashing2(int);
    void expand();
    long long default_values;
public:
    HashMap();
    HashMap(int);
    void put(int, long long);
    long long get(int);
    bool remove(int);
    long long operator [] (int);
};

HashMap::HashMap() {
    MAX = 16;
    table.assign(1<<MAX, -1);
    selector.assign(1<<MAX, false);
    default_values = 0;
    values.assign(1<<MAX, default_values);
}

HashMap::HashMap(int capacity) {
    MAX = capacity;
    table.assign(1<<MAX, -1);
    selector.assign(1<<MAX, false);
    default_values = 0;
    values.assign(1<<MAX, default_values);
}

void HashMap::put(int key, long long value) {
    int hash, activeKey = key, nextKey;
    bool activeSelect = true, nextSelect;
    long activeValue = value, nextValue;
}

```

```

hash = hashing1(key);
if(table[hash] == key) {
    values[hash] = value;
    return;
}

hash = hashing2(key);
if(table[hash] == key) {
    values[hash] = value;
    return;
}

do {
    hash = activeSelect ? hashing1(activeKey) : hashing2(activeKey);

    nextKey = table[hash];
    nextSelect = selector[hash];
    nextValue = values[hash];

    table[hash] = activeKey;
    selector[hash] = !activeSelect;
    values[hash] = activeValue;

    activeKey = nextKey;
    activeSelect = nextSelect;
    activeValue = nextValue;

    if (activeKey==key && activeSelect) break;
} while (activeKey != -1);

if (activeKey == key) {
    expand();
    put(key, value);
}
}

long long HashMap::get(int key) {
    int hash = hashing1(key);
    if(table[hash] == key) return values[hash];

    hash = hashing2(key);
    if(table[hash] == key) return values[hash];

    put(key, default_values);
    return default_values;
}

bool HashMap::remove(int key) {
    int hash = hashing1(key);
    if(table[hash] == key){
        table[hash] = -1;
        return true;
    }

    hash = hashing2(key);
    if(table[hash] == key){
        table[hash] = -1;
        return true;
    }

    return false;
}

int HashMap::hashing1(int x){
    x = (x+0x7ed55d16) + (x<<12);

```

```

    x = (x^0xc761c23c) ^ (x>>19);
    x = (x+0x165667b1) + (x<<5);
    x = (x+0xd3a2646c) ^ (x<<9);
    x = (x+0xfd7046c5) + (x<<3);
    x = (x^0xb55a4f09) ^ (x>>16);
    return x & 0x7fffffff >>(32-MAX);
}

int HashMap::hashing2(int x) {
    x -= (x<<6);
    x ^= (x>>17);
    x -= (x<<9);
    x ^= (x<<4);
    x -= (x<<3);
    x ^= (x<<10);
    x ^= (x>>15);
    return x & 0x7fffffff >>(32-MAX);
}

void HashMap::expand() {
    ++MAX;
    vector<int> oldTable = table;
    vector<long long> oldValues = values;

    table.reserve(1<<MAX);
    for (int i = 0; i < (1<<MAX); ++i)
        table[i] = -1;

    selector.reserve(1<<MAX);
    values.reserve(1<<MAX);

    for(int i = 0; i < oldTable.size(); ++i){
        if(oldTable[i] == -1) continue;
        put(oldTable[i], oldValues[i]);
    }
}

long long HashMap::operator [] (int key) {
    return get(key);
}

int main () {
    HashMap test;
    const int maxi = 1000000;
    for (int i = 0; i < maxi; ++i) {
        test.put(i, i);
    }
    return 0;
}

```

code/Hash.cpp

```

#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>
#define MAX 404
#define INF 0x7fffffff

using namespace std;

int num; // total num of node
int path[MAX][MAX];
bool visit_x[MAX], visit_y[MAX];
int parent[MAX], weight_x[MAX], weight_y[MAX];

```

```

bool find(int i) {
    visit_x[i] = true;
    for (int j = 0; j < num; ++j) {
        if (visit_y[j]) continue;
        if (weight_x[i] + weight_y[j] == path[i][j]) {
            visit_y[j] = true;
            if (parent[j] == -1 || find(parent[j])) {
                parent[j] = i;
                return true;
            }
        }
    }
    return false;
}

int weighted_hungarian() {
    /* remember to initial weight_x (max weight of node's edge)*/
    /* initialize */
    for (int i = 0; i < num; ++i) {
        weight_y[i] = 0;
        parent[i] = -1;
    }

    for (int i = 0; i < num; ++i) {
        while (1) {
            memset(visit_x, false, sizeof(visit_x));
            memset(visit_y, false, sizeof(visit_y));
            if (find(i)) break;

            int lack = INF;
            for (int j = 0; j < num; ++j) {
                if (visit_x[j]) {
                    for (int k = 0; k < num; ++k) {
                        if (!visit_y[k]) {
                            lack = min(lack, weight_x[j] + weight_y[k] - path[j][k]);
                        }
                    }
                }
            }
            if (lack == INF) break;
            // renew label
            for (int j = 0; j < num; ++j) {
                if (visit_x[j]) weight_x[j] -= lack;
                if (visit_y[j]) weight_y[j] += lack;
            }
        }
    }

    int ans = 0;
    for (int i = 0; i < num; ++i) {
        ans += weight_x[i];
        ans += weight_y[i];
    }
    return ans;
}

```

code/km.cpp

```

import java.util.*;

class MyHashMap{

    int MAX;
    int [] table;
}

```

```

boolean[] selector;
long[] values;

MyHashMap(){
    MAX = 16;
    table = new int[1<<MAX];
    Arrays.fill(table, -1);
    selector = new boolean[1<<MAX];
    values = new long[1<<MAX];
}

MyHashMap(int capacity){
    MAX = capacity;
    table = new int[1<<MAX];
    Arrays.fill(table, -1);
    selector = new boolean[1<<MAX];
    values = new long[1<<MAX];
}

void put(int key, long value){

    int hash, activeKey = key, nextKey;
    boolean activeSelect = true, nextSelect;
    long activeValue = value, nextValue;

    hash = hashing1(key);
    if(table[hash] == key){
        values[hash] = value;
        return;
    }
    hash = hashing2(key);
    if(table[hash] == key){
        values[hash] = value;
        return;
    }

    do{
        hash = activeSelect ? hashing1(activeKey) : hashing2(activeKey);
        nextKey = table[hash];
        nextSelect = selector[hash];
        nextValue = values[hash];
        table[hash] = activeKey;
        selector[hash] = !activeSelect;
        values[hash] = activeValue;
        activeKey = nextKey;
        activeSelect = nextSelect;
        activeValue = nextValue;
        if(activeKey==key && activeSelect) break;
    }while(activeKey != -1);

    if(activeKey == key){
        expand();
        put(key, value);
    }
}

long get(int key){
    int hash = hashing1(key);
    if(table[hash] == key) return values[hash];
    hash = hashing2(key);
    if(table[hash] == key) return values[hash];
    put(key, 01);
    return 01;
}

boolean remove(int key){

```



```

    int hash = hashing1(key);
    if(table[hash] == key){
        table[hash] = -1;
        return true;
    }
    hash = hashing2(key);
    if(table[hash] == key){
        table[hash] = -1;
        return true;
    }
    return false;
}

int hashing1(int x){
    x = (x+0x7ed55d16) + (x<<12);
    x = (x^0xc761c23c) ^ (x>>19);
    x = (x+0x165667b1) + (x<<5);
    x = (x+0xd3a2646c) ^ (x<<9);
    x = (x+0xfd7046c5) + (x<<3);
    x = (x^0xb55a4f09) ^ (x>>16);
    return x & 0x7fffffff >>(32-MAX);
}

int hashing2(int x){
    x -= (x<<6);
    x ^= (x>>17);
    x -= (x<<9);
    x ^= (x<<4);
    x -= (x<<3);
    x ^= (x<<10);
    x ^= (x>>15);
    return x & 0x7fffffff >>(32-MAX);
}

void expand(){
    MAX++;
    int[] oldTable = table;
    long[] oldValues = values;
    table = new int[1<<MAX];
    Arrays.fill(table, -1);
    selector = new boolean[1<<MAX];
    values = new long[1<<MAX];
    for(int i = 0; i<oldTable.length; i++){
        if(oldTable[i] == -1) continue;
        put(oldTable[i], oldValues[i]);
    }
}
}

```

code/MyHashMap.java

```

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

public class Scan{

    BufferedReader buffer;
    StringTokenizer tok;

    Scan(){
        buffer = new BufferedReader(new InputStreamReader(System.in));
    }

    boolean hasNext(){
        while(tok==null || !tok.hasMoreElements()){

```

```

        try{
            tok = new StringTokenizer(buffer.readLine());
        }catch(Exception e){
            return false;
        }
    }
    return true;
}

String next(){
    if(hasNext()) return tok.nextToken();
    return null;
}

String nextLine(){
    if(hasNext()) return tok.nextToken("\n");
    return null;
}

int nextInt(){
    return Integer.parseInt(next());
}

long nextLong(){
    return Long.parseLong(next());
}

double nextDouble(){
    return Double.parseDouble(next());
}
}

```

code/Scan.java

```

class SegmentTree{

    int rank, capacity;
    int [] input, tree;

    SegmentTree(int [] input){
        this.input = input;
        int length = input.length;
        rank = 1;
        while((1<<rank++) < length);
        capacity = 1<<(rank-1);
        //System.out.println("rank = "+rank+", capacity = "+capacity);
        tree = new int[capacity << 1];
        build(1, capacity, capacity<<1);
    }

    int build(int index, int left, int right){
        if(index >= left){
            //System.out.println("getInput("+index+") = "+getInput(index));
            return tree[index] = getInput(index);
        }
        int middle = (left+right) >> 1;
        int left_value = build(lc(index), left, middle);
        int right_value = build(rc(index), middle, right);
        return tree[index] = left_value + right_value;
    }

    int query(int start, int finish){
        return query(1, capacity, capacity<<1, capacity+start, capacity+finish);
    }

    int query(int index, int left, int right, int start, int finish){

```

```

    if(left == start && right == finish) return tree[index];
    int middle = (left+right) >> 1;
    if(finish <= middle) return query(lc(index), left, middle, start, finish);
    if(start >= middle) return query(rc(index), middle, right, start, finish);
    int left_value = query(lc(index), left, middle, start, middle);
    int right_value = query(rc(index), middle, right, middle, finish);
    return left_value+right_value;
}

void update(int target, int result){
    int diff = result - input[target];
    input[target] = result;
    target += capacity;
    while(target > 0){
        tree[target] += diff;
        target >>= 1;
    }
}

int getInput(int index){
    index -= capacity;
    if(index < input.length) return input[index];
    return 0;
}

int lc(int x){
    return x<<1;
}

int rc(int x){
    return (x<<1)+1;
}
}

```

code/SegmentTree.Basic.java

```

public class SegmentTree{

    int [] input;
    Entry [] tree;
    int rank, capacity;

    SegmentTree(int [] input){
        this.input = input;
        rank = 1;
        while(1<<(rank++) < input.length);
        capacity = 1<<rank>>1;
        System.out.println("rank = "+rank+", cap = "+capacity);
        tree = new Entry[1<<rank];
        build(0, 1, capacity);
    }

    int operate(int resultL, int resultR){
        return resultL + resultR;
    }

    int build(int index, int left, int right){
        Entry now = tree[index] = new Entry(left, right, index);
        if(left == right) return now.value = input[index-1];
        int middle = (left+right) >> 1;
        return now.value = operate(build(lc(index), left, middle), build(rc(index), middle+1, right));
    }

    int query(int index, int start, int finish){
        if(tree[index].lb == start && tree[index].rb == finish) return tree[index].value;
        int middle = (tree[index].lb+tree[index].rb) >> 1;
    }
}

```

```

    if(finish <= middle) return query(lc(index), start, finish);
    else if(middle < start) return query(rc(index), start, finish);
    else{
        return operate(query(lc(index), start, middle), query(rc(index), middle+1, finish));
    }
}

void update(int target, int value){
    int index = target-1+capacity;
    int diff = value - tree[index].value;
    maintain(index, diff);
}

void maintain(int index, int diff){
    tree[index].value += diff;
    if(index == 1) return;
    maintain(index<<1, diff);
}

int lc(int x){
    return x<<1;
}

int rc(int x){
    return (x<<1)+1;
}

class Entry{

    int lb, rb, id; //Left Bound, Right Bound and index in Array
    int value;

    Entry(int lb, int rb, int id){
        this.lb = lb;
        this.rb = rb;
        this.id = id;
        value = -1;
    }

}
}

```

code/SegmentTree.java

```

public class SplayTree{

    Node root;
    int size;

    SplayTree(){
        root = null;
        size = 0;
    }

    public boolean containsKey(int target){
        return splay(target);
    }

    public void add(int target){
        // System.out.println("add "+target);
        if(root == null){
            root = new Node(null, target);
            return;
        }
        Node now = root;
    }
}

```

```

while(true){
    if(now.key == target) break;
    if(target < now.key){
        if(now.lchild == null){
            now.lchild = new Node(now, target);
            break;
        }else now = now.lchild;
    }else{
        if(now.rchild == null){
            now.rchild = new Node(now, target);
            break;
        }else now = now.rchild;
    }
}
splay(target);
}

public void delete(int target){
//    System.out.println("delete "+target);
    if(!containsKey(target)) return;
    Node l = root.lchild;
    Node r = root.rchild;
    if(l == null){
        root = r;
    }else l.parent = null;
    if(r == null){
        root = l;
    }else r.parent = null;
    if(root == null || root.key != target) return;
    Node lMax = l;

    while(lMax.rchild != null) lMax = lMax.rchild;
    splay(lMax.key);
    lMax.rchild = r;
}

private boolean splay(int target){
//    System.out.println("splay "+target);
    while(true){
        if(root == null) return false;
        if(root.key == target) return true;
        if(target < root.key){
            if(root.lchild == null) return false;
            Node l = root.lchild;
            if(l.key == target){
                root = l;
                rightRoatation(l);
                return true;
            }
            if(target < l.key){
                if(l.lchild == null) return false;
                Node a = l.lchild;
                root = a;
                rightRoatation(l);
                rightRoatation(a);
            }else{
                if(l.rchild == null) return false;
                Node b = l.rchild;
                root = b;
                leftRoatation(b);
                rightRoatation(b);
            }
        }else{
            if(root.rchild == null) return false;
            Node r = root.rchild;
            if(r.key == target){

```

```

        root = r;
        leftRoatation(r);
        return true;
    }
    if(target>r.key){
        if(r.rchild == null) return false;
        Node d = r.rchild;
        root = d;
        leftRoatation(r);
        leftRoatation(d);
    }else{
        if(r.lchild == null) return false;
        Node c = r.lchild;
        root = c;
        rightRoatation(c);
        leftRoatation(c);
    }
}
}
}

void print(Node now){
    if(now == null){
        System.out.print("-1 ");
        return;
    }
    System.out.print(now.key+" ");
    print(now.lchild);
    print(now.rchild);
}

void rightRoatation(Node x){
    Node r = x.parent.parent;
    Node p = x.parent;
    Node b = x.rchild;

    x.rchild = p;
    if(p != null) p.parent = x;
    if(p != null) p.lchild = b;
    if(b != null) b.parent = p;
    x.parent = r;
    if(r != null) r.lchild = x;
}

void leftRoatation(Node x){
    Node r = x.parent.parent;
    Node p = x.parent;
    Node b = x.lchild;

    x.lchild = p;
    if(p != null) p.parent = x;
    if(p != null) p.rchild = b;
    if(b != null) b.parent = p;
    x.parent = r;
    if(r != null) r.rchild = x;
}

class Node{

    Node parent, lchild, rchild;
    int key;

    Node(Node parent, int key){
        this.parent = parent;
    }
}

```

```

        lchild = rchild = null;
        this.key = key;
    }
}
}

```

code/SplayTree.java

```

/*
Time Complexity=2*n*log(n)*log(n)
*/
#include <cstdio>
#include <algorithm>
using namespace std;

class Weight{
public:
    Weight(int a=0,int b=0,int c=0):id(a),first(b),second(c){}
    int id,first,second;
    bool operator<(const Weight &rhs)const{
        return first<rhs.first||(first==rhs.first&&second<rhs.second);
    }
    bool operator==(const Weight &rhs)const{
        return first==rhs.first&&second==rhs.second;
    }
    bool operator!=(const Weight &rhs)const{
        return !((*this)==rhs);
    }
};

class SuffixArray{
public:
    SuffixArray(char *r):refer(r){
        for(length=0;refer[length]!='\0';length++);
        rankOfIndex=new int[length];
        indexOfRank=new int[length];
        texi=new Weight[length];
        firstsort();
        for(int know=1;know<=length;know<<=1) doublesort(know);
    }

    void firstsort(){
        for(int i=0;i<length;i++){
            texi[i]=Weight(i,refer[i]);
        }
        sort(&texi[0],&texi[length-1]+1);

        indexOfRank[rankOfIndex[texi[0].id]=0]=texi[0].id;
        int current=0;
        for(int i=1;i<length;i++){
            if(texi[i]!=texi[i-1]) current++;
            indexOfRank[i]=texi[i].id;
            rankOfIndex[texi[i].id]=current;
        }
    }

    void doublesort(int known){
        for(int i=0;i<length;i++){
            texi[i]=Weight(i,rankOfIndex[i],(i+known<length)?rankOfIndex[i+known]:-1);
        }

        sort(&texi[0],&texi[length-1]+1);

        indexOfRank[rankOfIndex[texi[0].id]=0]=texi[0].id;
        int current=0;
        for(int i=1;i<length;i++){

```

```

        if (texi[i] != texi[i-1]) current++;
        indexOfRank[i] = texi[i].id;
        rankOfIndex[texi[i].id] = current;
    }

}

void print(int i, bool newline=0){
    printf("%s", &refer[indexOfRank[i]]);
    if (newline) printf("\n");
}

void printall(){
    for (int i=0; i<length; i++) print(i, 1);
}

int *indexOfRank, *rankOfIndex, length;
char *refer;
Weight *texi;
};

int main()
{
    SuffixArray a("aaaaaaaa");
    a.printall();
    return 0;
}

```

code/SuffixArray.cpp

```

/* O(V^2) */
#include <iostream>
#include <cstdio>
#define MAX 10000

using namespace std;

/* p for dfs, parent for tree */
int p[MAX], parent[MAX];
int lca[MAX][MAX];
int num, root;
bool visit[MAX];

int dis_find(int x) {
    if (x == p[x]) return x;
    return p[x] = dis_find(p[x]);
}

void dfs(int x) {
    if (visit[x]) return;
    visit[x] = true;

    for (int i = 0; i < num; ++i) {
        if (visit[i]) {
            lca[x][i] = lca[i][x] = dis_find(i);
        }
    }

    for (int i = 0; i < num; ++i) {
        if (parent[i] == x) {
            dfs(i);
            p[i] = x;
        }
    }
}

```



```

int main () {
    /* init */
    for (int i = 0; i < num; ++i) {
        p[i] = i;
        visit[i] = false;
        parent[i] = -1;
    }

    /* build tree first */
    /* use parent[x] = px to build tree */
    dfs(root);
    cin >> x >> y;
    cout << lca[x][y] << endl;
}

```

code/tarjan\_lca.cpp

```

public class Tester{

    public static void main(String[] args){
        Scan scan = new Scan();
        int[] array = new int[]{1,2,3,4,5,6,7,8};
        SegmentTree tree = new SegmentTree(array);
    }
}

```

code/Tester.java

```

import java.util.*;

/**
 * A magical data structure.
 * Written on 103.08.19
 */
public class Treap<K, V>{

    Random priorityGenerator;
    int time, size;
    Entry root;

    /**
     * Default Constructor
     */
    Treap(){
        root = null;
        time = size = 0;
        priorityGenerator = new Random();
    }

    /**
     * Find the Entry associated with key
     * @param key the key of the entry you are looking for
     * @return Entry
     */
    Entry find(K key){
        Entry now = root;
        Comparable<? super K> cmp = (Comparable<? super K>)key;
        int situation;
        while((now != null) && (situation=cmp.compareTo(now.key)) != 0){
            if(situation == -1) now = now.lchild;
            else now = now.rchild;
        }
        return now;
    }

    /**

```

```

* Split the treap based on the key
* Behavior undefined if the specified key is already in the tree
* @param cmp Comparable based on the key
* @return an array consists of two elements, the left subtree and the right
*/
Entry[] split(Comparable<? super K> cmp){
    Entry leftTree = null, rightTree = null, left = null, right = null;
    Entry current = root;
    while(current != null){
        if(cmp.compareTo(current.key) == -1){
            if(right == null){
                right = rightTree = current;
            }else{
                current.parent = right;
                right = right.lchild = current;
            }
            current = current.lchild;
            right.lchild = null;
            if(current != null) current.parent = null;
        }else{
            if(left == null){
                left = leftTree = current;
            }else{
                current.parent = left;
                left = left.rchild = current;
            }
            current = current.rchild;
            left.rchild = null;
            if(current != null) current.parent = null;
        }
    }
    return new Treap.Entry[]{ leftTree, rightTree };
}

/**
* Merge two Treaps into one.
* All keys of the entries in the left must be smaller than all keys of the entries in the right
* @param left the left Treap, it must be smaller than the right Treap
* @param right the right Treap, it must be greater than the left Treap
* @return root of the resulting Treap
*/
Entry merge(Entry left, Entry right){
    if(left == null) return right;
    if(right == null) return left;
    if(left.compareTo(right) == -1){
        if(right.lchild == null){
            right.lchild = left;
            left.parent = right;
        }else if(right.lchild.compareTo(left) == -1){
            Entry temp = right.lchild;
            right.lchild = left;
            left.parent = right;
            temp.parent = null;
            merge(left, temp);
        }else{
            merge(left, right.lchild);
        }
        return right;
    }else{
        if(left.rchild == null){
            left.rchild = right;
            right.parent = left;
        }else if(left.rchild.compareTo(right) == -1){
            Entry temp = left.rchild;
            left.rchild = right;
            right.parent = left;
        }
    }
}

```

```

        temp.parent = null;
        merge(temp, right);
    }else{
        merge(left.rchild, right);
    }
    return left;
}
}

/**
 * Insert a new Entry into the Treap if the key doesn't exists
 * Else replace the value with the new one and return the old value
 * @param key the key of the entry to be inserted or modified
 * @param value the new value of the entry
 * @return The original value if entry already exists, else return null;
 */
V puts(K key, V value){
    if(root == null){
        root = new Entry(key, value);
        size++;
        return null;
    }
    Entry position = find(key);
    if(position != null){
        V temp = position.value;
        position.value = value;
        return temp;
    }
    Entry newEntry = new Entry(key, value);
    Comparable<? super K> cmp = ((Comparable<? super K>)key);
    Entry[] subtree = split(cmp);
    newEntry = merge(subtree[0], newEntry);
    root = merge(newEntry, subtree[1]);
    size++;
    return null;
}

/**
 * Remove the entry associated with the specified key
 * return the according value upon removing
 * @param key the key of the entry to be destroyed
 * @return the value associated with the specified key, return null if no such key exists
 */
V remove(K key){
    Entry target = find(key);
    if(target == null) return null;
    if(target.lchild!=null) target.lchild.parent = null;
    if(target.rchild!=null) target.rchild.parent = null;
    Entry child = merge(target.lchild, target.rchild);
    if(child != null) child.parent = target.parent;
    if(target.parent != null){
        if(target == target.parent.lchild) target.parent.lchild = child;
        else if(target == target.parent.rchild) target.parent.rchild = child;
        else throw new AssertionError("remove fail");
    }else if(root == target) root = child;
    else throw new AssertionError("What is this?");
    size--;
    return target.value;
}

/**
 * This is a debugger
 * @param now the node doing a in order traversal
 * @return the size of the subtree rooted at now
 */
int iterate(Entry now, Entry parent){

```

```

    if(now == null) return 0;
    //System.out.println("Iterate "+now.key);
    if(now.parent != parent) System.out.println("Parent Check Fail!!!");
    int result = 1;
    result += iterate(now.lchild, now);
    //System.out.println("Entry : "+now.key);
    result += iterate(now.rchild, now);
    return result;
}

/**
 * The class storing all the entries of Treap
 * each Entry consists of a key and a value and a random generated priority
 * also stores its parent and children as well
 */
class Entry implements Comparable<Entry>{
    Entry parent, lchild, rchild;
    Integer priority, timestamp;
    K key;
    V value;

    Entry(K key, V value){
        this.key = key;
        this.value = value;
        parent = lchild = rchild = null;
        priority = priorityGenerator.nextInt();
        timestamp = time++;
    }

    @Override
    public int compareTo(Entry rhs){
        int result = priority.compareTo(rhs.priority);
        if(result == 0) return timestamp.compareTo(rhs.timestamp);
        return result;
    }
}
}

```

code/Treap.java