```
 1: import java.util.concurrent.atomic.*;
 2:
 3: public class Tree {
 4:         public Internal root;
 5:         public static int Inf1 = 10001;
 6:         public static int Inf2 = 10000;
 7:
 8:         //init tree with inf1 and inf2
 9:         public Tree(){
10:                 root = new Internal(Inf1);
11:                 root.left = new AtomicMarkableReference<Node>(new Leaf(Inf2),true)
;
12:                 root.right = new AtomicMarkableReference<Node>(new Leaf(Inf1),true
);
13:         }
14:
15:         public SearchRet Search(int key){
16:                 //returns the nearest leaf node to the key value
17:                 Internal gp=root, p=root;
18:                 Internal l = root;
19:                 Leaf L=null;
20:                 Update pupdate = p.update.getReference();
21:                 Update gpupdate = gp.update.getReference();
22:                 while(l.type == Type.INTERNAL ){
23:                         gp = p;
24:                         p = l;
25:                         gpupdate = pupdate;
26:                         pupdate = p.update.getReference();
27:                         if(key < l.key){
28:                                 if(l.left.getReference() instanceof Internal){

29:                                         l = (Internal)l.left.getReference();

30:                                 }
31:                                 else if(l.left.getReference() instanceof Leaf ){
32:                                         L = (Leaf) (l.left.getReference());
33:                                         L = new Leaf(l.left.getReference().key);
34:                                         L.key = l.left.getReference().key;
35:                                         L.type = Type.EXTERNAL;
36:                                         break;
37:                                 }
38:                         }
39:                         else{
40:                                 if(l.right.getReference().type == Type.INTERNAL)
41:                                         l = (Internal)l.right.getReference();
42:                                 else if(l.right.getReference().type == Type.EXTERN
AL){
43:                                         L = (Leaf) l.right.getReference();
44:                                         L.key = l.right.getReference().key;
45:                                         L.type = Type.EXTERNAL;
46:                                         break;
47:                                 }
48:                         }
49:                 }
50:                 SearchRet result = new SearchRet(gp,p,L,pupdate, gpupdate);
51:                 return result;
52:         }
53:
54:         public boolean find(int key){
55:                 SearchRet result = Search(key);
56:                 if (result.l.key == key)
57:                         return true;
58:                 return false;
59:         }
60:
61:         public boolean insert(int key){
62:                 while(true){
63:                         SearchRet result = Search(key);
64:                         if (result.l.key == key){
65:                                 return false;
66:                         }
67:                         if (result.pupdate.state != State.CLEAN)
68:                                 help(result.pupdate);
69:                         else{
70:                                 Leaf newSibling = new Leaf(result.l.key);
71:                                 Leaf newNode = new Leaf(key);
72:                                 Internal newInternal = new Internal(Math.max(resul
t.l.key, key));
73:
74:                                 if(key < result.l.key){
75:                                         newInternal.left.set(newNode, true);
76:                                         newInternal.right.set(newSibling, true);

77:                                 }
78:                                 else{
79:                                         newInternal.left.set(newSibling, true);
80:                                         newInternal.right.set(newNode, true);
81:                                 }
82:                                 IInfo op = new IInfo(result.p, result.l, newIntern
al);
83:                                 boolean output = result.p.update.compareAndSet(res
ult.pupdate, new Update(State.iFLAG,op),true, true);///doubt prove correctness
84:                                 if (output == true){
85:                                         helpInsert(op);
86:                                         return true;
87:                                 }
88:                                 else
89:                                         help(result.pupdate);
90:                         }
91:                 }
92:         }
93:
94:         public void CASChild(Internal parent, Node old, Node newNode){
95:
96:                 if(newNode.key < parent.key){
97:                         parent.left.compareAndSet(old, newNode, true, true);

98:                 }
99:                 else{
100:                        parent.right.compareAndSet(old, newNode, true, true);
101:                }
102:        }
103:
104:        public void helpInsert(IInfo op){
105:                CASChild(op.p, op.l, op.newInternal);
106:                op.p.update.set(new Update(State.CLEAN,op), true);
107:        }
108:
109:
110:        public void help(Update update){
111:                if(update.state == State.iFLAG)
112:                        helpInsert((IInfo)update.info);
113:                else if(update.state == State.MARK)
114:                        helpMarked((DInfo)update.info);
115:                else if (update.state == State.dFLAG)
116:                        helpDelete((DInfo)update.info);
117:        }
118:
119:        public boolean delete(int key){
120:
121:                SearchRet result;
122:                while(true){
123:                        result = Search(key);
124:                        if (result.l.key != key)
```

```
  125:                            return false;
  126:                    if (result.gpupdate.state != State.CLEAN){
  127:                            help(result.gpupdate);
  128:                    }
  129:
  130:
  131:                    else if(result.pupdate.state != State.CLEAN){
  132:                            help(result.pupdate);
  133:                    }
  134:
  135:                    else{
  136:                            DInfo op = new DInfo(result.p, result.gp, result.l
, result.pupdate, result.gpupdate );
  137:                            boolean out = op.gp.update.compareAndSet(result.gp
update, new Update(State.dFLAG,op), true, true);
  138:                            if (out)
  139:                            {       if (helpDelete(op))
  140:                                    return true;
  141:                            }
  142:                            else
  143:                                    help(result.gpupdate);
  144:                    }
  145:            }
  146:    }
  147:
  148:    public boolean helpDelete(DInfo op){
  149:
  150:            boolean res = op.p.update.compareAndSet(op.pupdate, new Update(Sta
te.MARK,op), true, true);
  151:            if(res==true){
  152:                    helpMarked(op);
  153:                    return true;
  154:            }
  155:            else{
  156:                    help(op.pupdate);
  157:                    op.gp.update.compareAndSet(op.gp.update.getReference(), ne
w Update(State.CLEAN,op), true, true);
  158:                    return false;
  159:            }
  160:    }
  161:
  162:    public void helpMarked(DInfo op){
  163:            if(op != null){
  164:                    Node other;
  165:                    if(op.p.right.getReference() == op.l)
  166:                            other = op.p.left.getReference();
  167:                    else
  168:                            other = op.p.right.getReference();
  169:
  170:                    CASChild(op.gp, op.p,other);
  171:                    op.gp.update.compareAndSet(op.gp.update.getReference(), ne
w Update(State.CLEAN,op), true, true);
  172:            }
  173:    }
  174:
  175:
  176:
  177:    void printTree(Internal n){
  178:            if (n.left.getReference().type == Type.EXTERNAL)
  179:                    System.out.println(n.left.getReference().key + " c/o " + n
.key);
  180:            else
  181:                    printTree((Internal) n.left.getReference());
  182:
  183:            if(n.right.getReference().type == Type.EXTERNAL)
  184:                    System.out.println(n.right.getReference().key + " c/o " +
n.key);
  185:            else
  186:                    printTree((Internal) n.right.getReference());
  187:    }
  188: }
```

```
 1:
 2: public class IInfo extends Info{
 3:
 4:         Internal newInternal;
 5:
 6:         public IInfo(Internal p, Leaf l, Internal newInternal ){
 7:                 super(p,l);
 8:                 this.newInternal = newInternal;
 9:         }
10: }
```

```
1:
2: public enum Type {
3:         INTERNAL, EXTERNAL
4: }
```

```
    1:
    2: public class SearchRet {
    3:                 Internal gp;
    4:                 Internal p;
    5:                 Update pupdate;
    6:                 Update gpupdate;
    7:                 Leaf l;
    8:
    9:                 public SearchRet(Internal gp, Internal p, Leaf l, Update pupdate,
Update gpupdate ){
   10:                         this.gp = gp;
   11:                         this.p = p;
   12:                         this.l = l;
   13:                         this.pupdate = pupdate;
   14:                         this.gpupdate = gpupdate;
   15:                 }
   16: }
```

```
    1: import java.util.concurrent.atomic.*;
    2: public class Internal extends Node {
    3:
    4:         public AtomicMarkableReference<Update> update;
    5:
    6:
    7:         public Internal(int key){
    8:                 super(key);
    9:                 update = new AtomicMarkableReference<Update>(new Update(State.CLEA
N, new Info(null, null)),true);
   10:                 super.type = Type.INTERNAL;
   11:         }
   12: }
```

```
 1:
 2: public class Leaf extends Node{
 3:
 4:         int key;
 5:
 6:         public Leaf(int key) {
 7:                 super(key);
 8:                 super.type = Type.EXTERNAL;
 9:         }
10:
11: }
```

```
 1:
 2: public class Update {
 3:         public Info info;
 4:         public State state;
 5:
 6:         public Update(State st, Info info){
 7:                 this.info = info;
 8:                 this.state = st;
 9:         }
10: }
```

```
 1:
 2: public class Info {
 3:
 4:         Internal p;
 5:         Leaf l;
 6:
 7:         public Info(Internal p, Leaf l){
 8:                 this.p = p;
 9:                 this.l = l;
10:         }
11: }
```

```
 1: import java.util.concurrent.atomic.AtomicMarkableReference;
 2:
 3:
 4: public class Node{
 5:                 public int key;
 6:                 public Type type;
 7:                 public AtomicMarkableReference<Node> left;
 8:                 public AtomicMarkableReference<Node> right;
 9:
10:                 public Node(int key){
11:                         this.key = key;
12:                         left = new AtomicMarkableReference<Node>(null, true);
13:                         right = new AtomicMarkableReference<Node>(null, true);
14:                 }
15: }
```

```
    1:
    2: public class DInfo extends Info{
    3:
    4:         public Internal gp;
    5:         public Update pupdate;
    6:
    7:         public DInfo(Internal p, Internal gp, Leaf l, Update pupdate, Update gpupd
ate){
    8:                 super(p,l);
    9:                 this.gp = gp;
   10:                 this.pupdate = pupdate;
   11:                 this.p = p;
   12:                 this.l = l;
   13:                 this.p.update.set(pupdate, true);
   14:                 this.gp.update.set(gpupdate, true);
   15:         }
   16: }
```

```
1:
2: public enum State {
3:
4:         CLEAN, MARK, iFLAG, dFLAG
5:
6: }
```

```
    1: import java.util.Random;
    2: import java.util.concurrent.locks.Lock;
    3: import java.util.concurrent.locks.ReentrantLock;
    4: import java.lang.management.*;
    5:
    6: public class ConTest implements Runnable{
    7:         static Tree T;
    8:         public static Lock printLock = new ReentrantLock();
    9:         public static volatile int I= 300;
   10:         public static volatile int D = 100;
   11:         public static volatile int S = 600;
   12:         //number of threads
   13:         public static int N = 1000;
   14:         //time measurement
   15:         public static long [] startTime = new long[N];
   16:         public static long [] endTime = new long[N];
   17:         public static volatile long threadTime = 0;
   18:         final ThreadMXBean bean = ManagementFactory.getThreadMXBean();
   19:
   20:         public static void main(String[] args){
   21:                 T = new Tree();
   22:                 long start = System.currentTimeMillis();
   23:                 for(int i=0;i<N;i++){
   24:
   25:                         startTime[i] = startTime[i] = System.currentTimeMillis();
   26:                         new Thread(new ConTest()).start();
   27:                         endTime[i] = System.currentTimeMillis() - startTime[i];
   28:                 }
   29:
   30:                 long maxEndTime = endTime[0];
   31:                 double avEndTime = 0;
   32:                 for(int i = 0 ; i < N; i++){
   33:                         if (endTime[i] > maxEndTime)
   34:                                 maxEndTime = endTime[i];
   35:                         avEndTime += endTime[i];
   36:                 }
   37:                 long end = System.currentTimeMillis() - start;
   38:                 avEndTime /= N;
   39:                 System.out.println("total program running time: " + end);
   40:                 System.out.println("Av threadTime: " + threadTime/1000000 + " ms"
);
   41:                 return;
   42:         }
   43:
   44:                 public void run(){
   45:                         Random rnd = new Random();
   46:                         for(int loop = 0; loop < 300; loop++){
   47:                                 int seed = rnd.nextInt(5000);
   48:                                 if(I >= 0){
   49:                                                 T.insert(seed);
   50:                                                 I--;

   51:                                 }
   52:                                 if(D >=0 ){
   53:                                                 T.delete(seed);
   54:                                                 D--;
   55:                                 }
   56:                                 if(S >= 0){
   57:                                         T.find(seed);
   58:                                         S--;
   59:                                 }
   60:                         }
   61:                         threadTime += bean.getCurrentThreadCpuTime();
   62:                 }
   63:         }
```