```
 1: import java.util.concurrent.atomic.AtomicInteger;
 2: public class CASConsensus extends ConsensusProtocol{
 3: //The subclass of Consensus protocol, which implements Consensus
 4: //Its methods are called by LFUniversal class
 5:
 6:         private final int FIRST = -1;
 7:         AtomicInteger r = new AtomicInteger(FIRST);
 8:
 9:         public Node decide(Node value){
10:                 int i = ThreadID.get();
11:                 propose(value);
12:                 if (r.compareAndSet(FIRST,i)){
13:                         return proposed[i];
14:                 }
15:                 else
16:                         return proposed[r.get()];
17:         }
18:
19:         public TNode decide(TNode value){
20:                 int i = ThreadID.get();
21:                 propose(value);
22:                 if (r.compareAndSet(FIRST,i)){
23:                         return proposedT[i];
24:                 }
25:                 else
26:                         return proposedT[r.get()];
27:         }
28: }
```

```
   1: import java.util.concurrent.*;
   2: import java.lang.management.*;
   3: import java.util.concurrent.locks.Lock;
   4: import java.util.concurrent.locks.ReentrantLock;
   5: import java.util.Random;
   6: public class Test implements Runnable{
   7:         public static Lock printLock = new ReentrantLock();
   8:         //number of threads doing Insert, delete and search respectively
   9:         public static int I = 300;
  10:         public static int D = 100;
  11:         public static int S = 600;
  12:         //number of threads
  13:         public static int N = 1000;
  14:         WaitFreeConstruct WFC = new WaitFreeConstruct();
  15:         public static long [] startTime = new long[N];
  16:         public static long [] endTime = new long[N];
  17:         public static long [] startUserTime = new long[N];
  18:         public static long [] endUserTime = new long[N];
  19:         public static void main(String [] args){
  20:
  21:                 long start = System.currentTimeMillis();
  22:                 for(int i = 0 ; i < N;i++){
  23:
  24:                         Thread t = new Thread(new Test());
  25:                         startTime[i] = System.currentTimeMillis();
  26:                         t.run();
  27:
  28:                         endTime[i] = System.currentTimeMillis() - startTime[i];
  29:                 }
  30:                 long end = System.currentTimeMillis() - start;
  31:                 System.out.println("ThreadID " + "Start Time " + "End Time:");
  32:
  33:                 long maxEndTime = endTime[0];
  34:                 double avEndTime = 0;
  35:                 for(int i = 0 ; i < N; i++){
  36:                         System.out.println(i + " " + startTime[i] + " " +endTime[i
]);
  37:                         if (endTime[i] > maxEndTime)
  38:                                 maxEndTime = endTime[i];
  39:                         avEndTime += endTime[i];
  40:                 }
  41:                 System.out.println("Total time taken: " + avEndTime);
  42:                 avEndTime /= N;
  43:                 System.out.println("Max:" + maxEndTime + " | Av:"+avEndTime);
  44:                 System.out.println(end);
  45:                 return;
  46:         }
  47:
  48:         public void run(){
  49:
  50:                 Random rnd = new Random();
  51:                 for(int  j = 0; j  < 1000; j++){
  52:                 int seed = rnd.nextInt();
  53:                 if (seed < 0 )
  54:                         seed = seed * -1;
  55:                 seed = seed%100 ;
  56:                 int opId = seed%3+1;
  57:                 Invoc invoc;
  58:                 if(opId == 1 && I >= 0){
  59:                         invoc = new Invoc(1,seed);
  60:                         I--;
  61:                         WFC.apply(invoc);
  62:                 }
  63:                 else if(opId == 2 && D >= 0){
  64:                         invoc = new Invoc(2,seed);
  65:                         D--;
  66:                         WFC.apply(invoc);
  67:
  68:                 }
  69:                 else if(S >= 0){
  70:                         invoc = new Invoc(3,seed);
  71:                         S--;
  72:                         WFC.apply(invoc);
  73:                 }
  74:                 }
  75:         }
  76:
  77: }
```

```
1:
2: public class Invoc {
3:         int opId;
4:         int key;
5:         public Invoc(int opId, int key){
6:                 this.opId = opId;
7:                 this.key = key;
8:         }
9: }
```

```
 1:
 2: public class Node {
 3:         public int seq;
 4:         public CASConsensus decideNext;
 5:         Node next;
 6:         Invoc invoc;
 7:         boolean status;
 8:
 9:         public Node(){
10:                 this.seq = 1;
11:                 this.decideNext = new CASConsensus();
12:                 next = null;
13:                 invoc = null;
14:                 status = true;
15:         }
16:
17:         public Node (Invoc invoc){
18:                 this.seq = 0;
19:                 this.decideNext = new CASConsensus();
20:                 next = null;
21:                 this.invoc = invoc;
22:                 status = false;
23:         }
24:
25:         public static Node max(Node[] head){
26:                 Node retNode = head[0];
27:                 for(int i=0; i < head.length; i++){
28:                         if (head[i].seq > retNode.seq){
29:                                 retNode = head[i];
30:                         }
31:                 }
32:                 return retNode;
33:         }
34: }
```

```
 1: import java.util.concurrent.atomic.*;
 2: //independent of choice of N
 3: public class TNode {
 4:         public int key;
 5:         public int level;
 6:         public AtomicBoolean freeze;
 7:         public CASConsensus decideRight;
 8:         public TNode right;
 9:         public CASConsensus decideLeft;
10:         public TNode left;
11:
12:         public TNode(int key){
13:                 this.key = key;
14:                 level = 0;
15:                 freeze = new AtomicBoolean(false);
16:                 decideLeft = new CASConsensus();
17:                 left = null;
18:                 decideRight = new CASConsensus();
19:                 right = null;
20:         }
21:
22:
23:         public TNode [] search (int key){
24:                 TNode curr = this;
25:                 TNode parent=curr;
26:                 TNode [] ret = new TNode[2];
27:                 while(curr != null){
28:                         if (key == curr.key){
29:                                 ret[0] = curr;
30:                                 ret[1] = parent;
31:                                 return ret;
32:                         }
33:                         else if(key < curr.key){
34:                                 parent = curr;
35:                                 curr = curr.left;
36:                         }
37:                         else{
38:                                 parent = curr;
39:                                 curr = curr.right;
40:                         }
41:                 }
42:                 ret[0] = null;
43:                 ret[1] = parent;
44:                 return ret;
45:         }
46:
47: }
```

```
 1:
 2: public abstract class ConsensusProtocol implements Consensus{
 3:
 4:         static final int N = 100;
 5:         protected Node [] proposed = new Node [N];
 6:         protected TNode [] proposedT = new TNode [N];
 7:
 8:         abstract public Node decide(Node value);
 9:         abstract public TNode decide(TNode value);
10:         public void propose(Node value){
11:                 proposed[ThreadID.get()] = value;
12:         }
13:         public void propose(TNode value){
14:                 proposedT[ThreadID.get()] = value;
15:         }
16:
17: }
```

```
 1: import java.util.concurrent.locks.Lock;
 2: import java.util.concurrent.locks.ReentrantLock;
 3:
 4:
 5: public class WaitFreeConstruct {
 6:         public static TNode ROOT;
 7:         public static int N = 60;
 8:         public static Lock printLock = new ReentrantLock();
 9:         private static Node [] announce = new Node[N];
10:         private static Node [] head = new Node[N];
11:         private static Node tail = new Node();
12:         WaitFreeConstruct(){
13:                 ROOT = new TNode(1000);
14:                 for(int i = 0 ; i < N; i++){
15:                         announce[i] = tail;
16:                         head[i] = tail;
17:                 }
18:         }
19:
20:         boolean apply(Invoc invoc){
21:                 int i = ThreadID.get();
22:                 announce[i] = new Node(invoc);
23:                 head[i] = Node.max(head);
24:                 Node optiTail = head[i];
25:                 while(announce[i].seq == 0){
26:                         Node before = head[i];
27:                         Node prefer = announce[i];
28:                         Node help = announce[(before.seq+1)%N];
29:                         if(help.seq == 0)
30:                                 prefer = help;
31:
32:                         Node after = before.decideNext.decide(prefer);
33:                         before.next = after;
34:                         after.seq = before.seq+1;
35:                         head[i] = after;
36:                 }
37:
38:                 Node current = optiTail;
39:                 while(current != announce[i]){
40:                         if(current.status == false){
41:                                 if(current.invoc.opId == 1){
42:                                         insert(current.invoc.key);
43:                                 }
44:                                 else if(current.invoc.opId == 2){
45:                                         delete(current.invoc.key);
46:                                 }
47:                                 else
48:                                         find(current.invoc.key);
49:                                 current.status = true;
50:                         }
51:                         current = current.next;
52:                 }
53:
54:                 head[i] = announce[i];
55:                 boolean output = true;
56:                 if (announce[i].status == false){
57:                         if(invoc.opId == 1){
58:                                 output = insert(invoc.key);
59:                         }
60:                         else if(invoc.opId == 2)
61:                                 output = delete(invoc.key);
62:                         else
63:                                 output = find(invoc.key);
64:                 }
65:                 announce[i].status = true;
66:
67:                 return output;
68:         }
69:
70:         boolean insert(int key){
71:                 TNode [] ret = ROOT.search(key);
72:                 if(ret[0] != null){
73:                         return ret[0].freeze.compareAndSet(true, false);
74:                 }
75:                 //if node is not found in the tree already
76:                 TNode parent = ret[1];
77:                 TNode toInsert = new TNode(key);
78:                 TNode q;
79:                 while(toInsert.level == 0){
80:                         if(key > parent.key){
81:                                 q = parent.decideRight.decide(toInsert);
82:                                 parent.right = q;
83:                                 q.level = parent.level + 1;
84:                                 parent = q;
85:                         }
86:                         else if (key < parent.key){
87:                                 q = parent.decideLeft.decide(toInsert);
88:                                 parent.left = q;
89:                                 q.level = parent.level + 1;
90:                                 parent = q;
91:                         }
92:                         else{
93:                                 return false;
94:                         }
95:                 }
96:                 return true;
97:         }
98:
99:         public boolean delete(int key){
100:                TNode [] ret = ROOT.search(key);
101:                if(ret[0] == null)
102:                        return false;
103:                else{
104:                        return ret[0].freeze.compareAndSet(false, true);
105:                }
106:        }
107:
108:        public boolean find(int key){
109:                TNode [] ret = ROOT.search(key);
110:                if(ret[0] == null)
111:                        return false;
112:
113:                else{
114:                        return !ret[0].freeze.get();
115:                }
116:        }
117:
118:        public void print(){
119:                Node curr = tail.next;
120:                while(curr != null){
121:                        System.out.println("No. "+curr.seq + " | " + "ID: " + curr
.invoc.opId + " | " + "key: " + curr.invoc.key);
122:                        curr = curr.next;
123:                }
124:                TNode start = ROOT;
125:                System.out.println("Inorder printing...");
126:                inorder(start);
127:        }
128:
129:        private void inorder(TNode root){
130:                if(root == null)
131:                        return;
132:                inorder(root.left);
133:                if(root.freeze.get() == false)
```

```
134:                         System.out.println(root.key + " @ " + root.level);
135:                 inorder(root.right);
136:         }
137: }
```

```
1: public interface Consensus{
2:          Node decide(Node value);
3:          TNode decide(TNode value);
4: }
```

```
 1: public class ThreadID {
 2:     private static volatile int nextID = 0;
 3:     private static class ThreadLocalID extends ThreadLocal<Integer> {
 4:         protected synchronized Integer initialValue() {
 5:                 return nextID++;
 6:         }
 7:     }
 8:
 9:     private static ThreadLocalID threadID = new ThreadLocalID();
10:     public static int get() {
11:         return threadID.get();
12:     }
13:     public static void set(int index) {
14:         threadID.set(index);
15:     }
16: }
```