```
 1: import java.util.concurrent.locks.ReentrantLock;
 2:
 3: public class Node {
 4:
 5:         public static int MAX_LEVEL = 10;
 6:         final ReentrantLock Lock = new ReentrantLock();
 7:         final int key;
 8:         final Node [] next ;
 9:         volatile boolean marked = false;
10:         volatile boolean fullyLinked = false;
11:         public int toplevel;
12:
13:         public Node(int key){   //sentinel constructor
14:                 this.key = key;
15:                 next = new Node[MAX_LEVEL+1];
16:                 this.toplevel = MAX_LEVEL;
17:         }
18:
19:         public Node (int key, int level){//non sentinel constructor
20:                 this.key = key;
21:                 next = new Node[level+1];
22:                 this.toplevel = level;
23:         }
24:
25:         public void lock(){
26:                 Lock.lock();
27:         }
28:
29:         public void unlock(){
30:                 Lock.unlock();
31:         }
32: }
```

```
    1: import java.util.Random;
    2: import java.util.concurrent.locks.Lock;
    3: import java.util.concurrent.locks.ReentrantLock;
    4: public class TestConc implements Runnable{
    5:
    6:         static LazyList l;
    7:         public static Lock printLock = new ReentrantLock();
    8:         public static void main(String[] args){
    9:                 l = new LazyList();
   10:                 for(int i=0;i<3;i++)
   11:                         new Thread(new TestConc()).start();
   12:         }
   13:         public void run(){
   14:                 Random rnd = new Random();
   15:                 for(int loop = 0; loop < 10; loop++){
   16:                         int seed = rnd.nextInt(50);
   17:                         if(seed%3 == 0){
   18:                                 if(l.contains(seed) == false){
   19:                                         l.add(seed);
   20:                                 }
   21:                                 else{
   22:                                         printLock.lock();
   23:                                         System.out.println(seed+ " Already exists"
);
   24:                                         printLock.unlock();
   25:                                 }
   26:
   27:                         }
   28:                         else if(seed%3 == 1){
   29:                                 if (l.contains(seed) != false)
   30:                                 {
   31:                                         l.remove(seed);
   32:                                 }
   33:                                 else
   34:                                 {
   35:                                         printLock.lock();
   36:                                         System.out.println(seed + " Not found!");
   37:                                         printLock.unlock();
   38:                                 }
   39:                         }
   40:                         else{
   41:                                 printLock.lock();
   42:                                 System.out.println("Printing the skipList");
   43:                                 l.printList();
   44:                                 System.out.println("Printed the skipList");
   45:                                 printLock.unlock();
   46:                         }
   47:
   48:         }
   49: }}
```

```
 1: import java.util.Random;
 2: public class LazyList {
 3:         public static final int MAX_LEVEL = 10;
 4:         final Node head = new Node(Integer.MIN_VALUE);
 5:         final Node tail = new Node(Integer.MAX_VALUE);
 6:         public LazyList(){
 7:                 for(int i =0 ; i<head.next.length; i++){
 8:                         head.next[i] = tail;
 9:                 }
10:         }
11:
12:         public int find(int key, Node[] preds, Node[] succs){
13:                 //iterates from the top of the list to the bottom most level
14:                 //locates the node
15:                 //fills preds and succs array
16:                 Node curr,pred;
17:                 pred = head;
18:                 int lFound = -1;
19:                 for(int level= MAX_LEVEL; level >= 0; level--){
20:                         curr = pred.next[level];
21:                         while(curr.key < key){
22:                                 pred = curr;
23:                                 curr = pred.next[level];
24:                         }
25:                         if (lFound == -1 && curr.key == key)
26:                                 lFound = level;
27:
28:                         preds[level] = pred;
29:                         succs[level] = curr;
30:                 }
31:
32:                 return lFound;
33:         }
34:
35:         public boolean add(int key){
36:                 Random rnd = new Random();
37:                 //get the height by coin flip
38:                 int topLevel=0; //stores the topmost level where this node is adde
d
39:                 while( rnd.nextInt(2)%2 == 1)   //till you keep getting heads
40:                         topLevel+=1;
41:
42:                 topLevel = Math.min(topLevel, MAX_LEVEL);
43:
44:                 Node [] preds = new Node[MAX_LEVEL+1];
45:                 Node [] succs = new Node[MAX_LEVEL+1];
46:                 Node pred,succ ;
47:                 while(true){
48:                         int lFound = find(key, preds, succs);
49:                         if (lFound != -1){//node is found
50:                                 Node nodeFound = succs[0];
51:                                 if (!nodeFound.marked){
52:                                         while(nodeFound.fullyLinked != true){
53:                                         //wait till fully linked
54:                                         }
55:                                         return false;
56:                                 }
57:                                 continue;
58:                         }
59:                         //if not found
60:                         //* lock all preds
61:                         boolean valid = true;
62:                         int highestLocked=0;
63:                         try{
64:                                 pred = head;
65:                                 for(int level = 0; valid && (level <= topLevel) ;
level++){
66:                                         pred = preds[level];
67:                                         succ = succs[level];
68:                                         pred.lock();
69:
70:                                         highestLocked = level;
71:
72:                                         //* validate all preds
73:                                         valid = !pred.marked && !succ.marked && (p
red.next[level] == succ);
74:                                 }
75:                                 if (!valid)//valid criteria failed
76:                                         continue;
77:
78:                                 Node newNode = new Node(key, topLevel);
79:                                 //assign pointers
80:                                 for(int level = 0; level <= topLevel; level++){
81:                                         newNode.next[level] = succs[level];
82:                                 }
83:                                 for(int level = 0 ; level <= topLevel; level++){
84:                                         preds[level].next[level] = newNode;
85:                                 }
86:                                 //* set linearization point
87:                                 newNode.fullyLinked = true;
88:                                 return true;
89:                         }
90:                         finally{
91:                                 for(int level = 0; level <= highestLocked; level++
){
92:                                 //      System.out.println("@level: "+ level);
93:                                         pred = preds[level];
94:                                         pred.unlock();
95:                                 }
96:                         }
97:                 }
98:         }
99:
100:        public boolean remove(int key){
101:                Node[] preds = new Node[MAX_LEVEL+1];
102:                Node[] succs = new Node[MAX_LEVEL+1];
103:                Node victim = null;
104:                boolean isMarked = false;
105:                int topLevel = -1;
106:
107:                while(true){
108:                        int lFound = find(key, preds, succs);
109:                        if (lFound != -1)//if node is found
110:                                victim = succs[lFound];
111:                        if(isMarked | (lFound != -1 && (victim.fullyLinked && vict
im.toplevel == lFound && !victim.marked) )){
112:
113:                                if(!isMarked){
114:                                        topLevel = victim.toplevel;
115:                                        victim.lock();
116:                                        if(victim.marked){
117:                                                victim.unlock();
118:                                                return false;
119:                                        }
120:
121:                                        victim.marked= true;
122:                                        isMarked = true;
123:                                }
124:
125:                                int highestLocked = -1;
126:                                try{
127:
128:                                        Node pred;
129:                                        boolean valid = true;
```

```
  130:                                        for(int level = 0; valid && (level <= topL
evel); level++){
  131:                                                pred = preds[level];
  132:                                                pred.lock();
  133:                                                highestLocked = level;
  134:                                                valid = !pred.marked && pred.next[
level] == victim;
  135:                                        }
  136:                                        if(!valid)
  137:                                                continue;
  138:
  139:                                        for(int level = topLevel; level >= 0; leve
l--){
  140:                                                preds[level].next[level] = victim.
next[level];
  141:                                        }
  142:                                        victim.unlock();
  143:                                        return true;
  144:                                }
  145:                                finally{
  146:                                        for(int i =0 ; i<= highestLocked; i++)
  147:                                                preds[i].unlock();
  148:                                }
  149:                        }
  150:                        else
  151:                                return false;
  152:                }
  153:        }
  154:
  155:        public boolean contains(int key){
  156:                Node [] preds = new Node[MAX_LEVEL+1];
  157:                Node [] succs = new Node[MAX_LEVEL+1];
  158:                int lFound = find(key, preds,succs);
  159:                return (lFound != -1 && succs[lFound].fullyLinked && !succs[lFound
].marked);
  160:        }
  161:
  162:        void printList(){
  163:                Node curr;
  164:                for(int level = MAX_LEVEL; level >=0 ; level--){
  165:                        curr = head;
  166:                        while(curr.next[level].key < tail.key){
  167:                                System.out.print(curr.next[level].key + " ");
  168:                                curr = curr.next[level];
  169:                        }
  170:                        System.out.println();
  171:                }
  172:                return;
  173:        }
  174: }
```