```
    1: import java.lang.management.ThreadInfo;
    2: import java.util.Random;
    3: import java.util.concurrent.locks.Lock;
    4: import java.util.concurrent.locks.ReentrantLock;
    5: import java.util.concurrent.*;
    6:
    7: public class Test implements Runnable{
    8:
    9:         private int RANGE = 100;
   10:         public static int counter=0;
   11:         public static volatile FineGrainedHeap FGH;
   12:         public static volatile Lock printLock = new ReentrantLock();    // to lock
 print to prevent interleaved printing of heap
   13:
   14:         public static void main(String[] args){
   15:
   16:                 int capacity = 100;
   17:                 FGH = new FineGrainedHeap(capacity);
   18:                 int t = 5;
   19:                 for(int j =0 ;j < t;j++){
   20:                         (new Thread(new Test())).start();
   21:                 }
   22:                 return;
   23:         }
   24:
   25:         public void run(){
   26:                 Random random = new Random();
   27:                 for(int loop = 0 ;loop < 10; loop++){
   28:                         int seed = random.nextInt(RANGE);
   29:                         if (seed%3==0){
   30:                                 try{
   31:
   32:                                         int next = random.nextInt(RANGE);
   33:                                         printLock.lock();
   34:                                         System.out.println("Enq (" + next + ") :"
+ java.lang.Thread.currentThread().getId());
   35:                                         printLock.unlock();

   36:                                         FGH.add(next);
   37:                                         printLock.lock();
   38:                                         System.out.println("Ok ():" + java.lang.Th
read.currentThread().getId());
   39:                                         printLock.unlock();
   40:                                 }
   41:                                 catch(Exception e){}
   42:                         }
   43:                         else if (seed%3 == 1){
   44:
   45:                                 try{
   46:                                         printLock.lock();
   47:                                         System.out.println("Deq (): " + java.lang.
Thread.currentThread().getId());
   48:                                         printLock.unlock();
   49:
   50:                                         HeapNode min = FGH.removeMin();
   51:
   52:                                         printLock.lock();
   53:                                         if (min == null){
   54:                                                 System.out.println(java.lang.Threa
d.currentThread().getId() + " : Underflow");
   55:                                         }
   56:                                         else{
   57:                                                 System.out.println("Ok("+ min.item
 +"): " + java.lang.Thread.currentThread().getId());
   58:                                         }
   59:                                         printLock.unlock();
   60:                                 }
   61:                                 catch(Exception e){}
   62:                         }
   63:                         else{
   64:
   65:                                 printLock.lock();
   66:                                 FGH.print();
   67:                                 printLock.unlock();
   68:
   69:                         }
   70:                 }
   71:         }
   72: }
```

```
    1: import java.util.concurrent.locks.Lock;
    2: import java.util.concurrent.locks.ReentrantLock;
    3: import java.util.concurrent.*;
    4:
    5: public class FineGrainedHeap {
    6:
    7:         public int ROOT = 1;
    8:         public static final int NO_ONE = -1;
    9:         private Lock heapLock;
   10:         private int next;
   11:         HeapNode [] heap;
   12:         private Lock nextLock;
   13:
   14:
   15:
   16:         public FineGrainedHeap(int capacity){
   17:                 heapLock = new ReentrantLock();
   18: //              nextLock  = new ReentrantLock();
   19:                 next = ROOT;
   20:                 heap = new HeapNode [capacity+1];
   21:                 for(int i =0;i < capacity+1;i++)
   22:                         heap[i] = new HeapNode();
   23:         }
   24:
   25:         public void add(int data){
   26:
   27:                 System.out.println(java.lang.Thread.currentThread().getId() + " wa
nts to acquire the heaplock to enqueue");
   28:                 heapLock.lock();
   29:                 int child = next++;
   30:                 System.out.println(java.lang.Thread.currentThread().getId() + " no
w acquired the heaplock to enqueue");
   31:
   32:
   33:                 heap[child].lock();
   34:                 heap[child].init(data);
   35:                 System.out.println(java.lang.Thread.currentThread().getId() + " no
w released the heaplock");
   36:                 heapLock.unlock();
   37:
   38:                 heap[child].unlock();
   39:
   40:                 while(child > ROOT){
   41:                         int parent = child/2;
   42:                         heap[parent].lock();
   43:                         heap[child].lock();
   44:                         int oldChild = child;
   45:                         try{
   46:                                 if (heap[parent].tag == Status.AVAILABLE && heap[c
hild].amOwner() ){
   47:                                         if(heap[child].item < heap[parent].item){
   48:
   49:                                                 //swap item
   50:                                                 int temp = heap[parent].item;
   51:                                                 heap[parent].item = heap[child].it
em;
   52:                                                 heap[child].item = temp;
   53:
   54:                                                 //swap owner
   55:                                                 long tempOwner = heap[parent].owne
r;
   56:                                                 heap[parent].owner = heap[child].o
wner;
   57:                                                 heap[child].owner = tempOwner;
   58:
   59:                                                 //swap status
   60:                                                 Status tempStatus = heap[parent].t
ag;
   61:                                                 heap[parent].tag = heap[child].tag
;
   62:                                                 heap[child].tag = tempStatus;
   63:
   64:                                                 child = parent;
   65:                                         }
   66:                                         else{
   67:                                                 heap[child].tag = Status.AVAILABLE
;
   68:                                                 heap[child].owner = NO_ONE;
   69:                                                 return;
   70:                                         }
   71:                                 }
   72:                                 else if(!heap[child].amOwner()){
   73:                                         child = parent;
   74:                                 }
   75:                         }
   76:                         finally{
   77:                                 heap[oldChild].unlock();
   78:                                 heap[parent].unlock();
   79:                         }
   80:                 }
   81:
   82:                 if (child == ROOT){
   83:                         heap[ROOT].lock();
   84:                         if(heap[ROOT].amOwner()){
   85:                                 heap[ROOT].tag = Status.AVAILABLE;
   86:                                 heap[child].owner = NO_ONE;
   87:                         }
   88:                         heap[ROOT].unlock();
   89:                 }
   90:         }
   91:
   92:
   93:         public HeapNode removeMin(){
   94:                 System.out.println(java.lang.Thread.currentThread().getId() + " wa
nts to acquire heaplock to dequeue");
   95:                 heapLock.lock();
   96:
   97:                 if(next <= 1){
   98:                         return null;
   99:                 }
  100:                 else{
  101:                 HeapNode retNode = heap[ROOT];
  102:                 int bottom = --next;
  103:
  104:                         System.out.println(java.lang.Thread.currentThread().getId(
) + " now has the heaplock to dequeue");
  105:
  106:
  107:                         heap[bottom].lock();
  108:                         heap[ROOT].lock();
  109:
  110:                         System.out.println(java.lang.Thread.currentThread().getId(
) + " now released the heaplock");
  111:                         heapLock.unlock();
  112:
  113:                         retNode = heap[ROOT];
  114:                         heap[ROOT].tag = Status.EMPTY;
  115:                         heap[ROOT].owner = NO_ONE;
  116:
  117:                         heap[ROOT].item = heap[bottom].item;
  118:                         heap[ROOT].tag = heap[bottom].tag;
  119:                         heap[ROOT].owner = heap[bottom].owner;
```

```
  120:
  121:                        heap[bottom].unlock();
  122:
  123:                        if (heap[ROOT].tag == Status.EMPTY){    //heap had only ro
ot item
  124:                                heap[ROOT].unlock();
  125:                                return retNode;
  126:                        }
  127:
  128:                        int HeapLength = next;
  129:                        int child = 0;
  130:                        int parent = ROOT;
  131:
  132:                        while(parent <= HeapLength/2){
  133:                                int left = parent*2;
  134:                                int right = parent*2+1;
  135:                                heap[left].lock();
  136:                                heap[right].lock();
  137:                                if(heap[left].tag == Status.EMPTY){
  138:                                        heap[right].unlock();
  139:                                        heap[left].unlock();
  140:                                        break;
  141:                                }
  142:                                else if (heap[right].tag == Status.EMPTY || heap[r
ight].item > heap[left].item){
  143:                                        heap[right].unlock();
  144:                                        child = left;
  145:                                }
  146:                                else{
  147:                                        heap[left].unlock();
  148:                                        child = right;
  149:                                }
  150:
  151:                                if(heap[child].item < heap[parent].item){
  152:
  153:                                        //swap all fields of parent and child
  154:                                                // item swap
  155:                                                int temp1 = heap[child].item;
  156:                                                heap[child].item = heap[parent].it
em;
  157:                                                heap[parent].item = temp1;
  158:
  159:                                                //status swap
  160:                                                Status tempStatus = heap[child].ta
g;
  161:                                                heap[child].tag = heap[parent].tag
;
  162:                                                heap[parent].tag = tempStatus;
  163:
  164:                                                //owner swap
  165:                                                long tempOwner = heap[child].owner
;
  166:                                                heap[child].owner = heap[parent].o
wner;
  167:                                                heap[parent].owner = tempOwner;
  168:
  169:                                                heap[parent].unlock();
  170:                                                parent = child;
  171:                                        }
  172:                                        else{
  173:                                                heap[child].unlock();
  174:                                                break;
  175:                                        }
  176:                                }
  177:                                heap[parent].unlock();
  178:                                return retNode;
  179:                        }
  180:
  181:                }
  182:
  183:        public synchronized void print(){
  184:                heapLock.lock();
  185:                for (int i=1;i<next;i++){
  186:                        System.out.println(heap[i].item);
  187:                }
  188:                heapLock.unlock();
  189:        }
  190: }
```

```
1: public enum Status {EMPTY, AVAILABLE, BUSY};
```

```
 1: import java.util.concurrent.*;
 2: import java.util.concurrent.locks.Lock;
 3: import java.util.concurrent.locks.ReentrantLock;
 4: import java.lang.*;
 5:
 6: public class HeapNode{
 7:         private static final int ROOT = 1;
 8:         private static final int NO_ONE = -1;
 9:         Status tag;
10:         int item;
11:         Lock lock;
12:         long owner;
13:
14:         public void init(int data)      {
15:                 this.item = data;
16:                 tag = Status.BUSY;
17:                 owner = java.lang.Thread.currentThread().getId();
18:         }
19:
20:         public HeapNode(){
21:                 tag = Status.EMPTY;
22:                 lock = new ReentrantLock();
23:         }
24:
25:         public void lock(){ lock.lock(); }
26:         public void unlock(){ lock.unlock(); }
27:         public boolean amOwner(){
28:                 return (this.owner == java.lang.Thread.currentThread().getId());
29:         }
30: }
```