```java
package MMS;

/**
 * Represents a list of Nodes. Each node holds a reference to a memory block.
 * <br>
 * (Part of Homework 10 in the Intro to CS course, Efi Arazi School of CS)
 */
public class List {

    private Node first = null; // The first (dummy) node of this list
    private Node last = null; // The last node of this list
    private int size = 0; // Number of elements (nodes) in this list

    /**
     * Constructs a new list of Node objects, each holding a memory block (MemBlock
     * object)
     */
    public List() {
        // Creates a dummy node and makes first and last point to it.
        first = new Node(null);
        last = first;
    }

    /**
     * Adds the given memory block to the end of this list.
     * Executes efficiently, in O(1).
     *
     * @param block The memory block that is added at the list's end
     */
    public void addLast(MemBlock block) {
        Node newNode = new Node(block);
        size++;
        last.next = newNode;
        last = newNode;
    }

    /**
     * Adds the given memory block at the beginning of this list.
     * Executes efficiently, in O(1).
     *
     * @param block The memory block that is added at the list's beginning
     */
    public void addFirst(MemBlock block) {
        Node newNode = new Node(block);
        newNode.next = first.next;
        first.next = newNode;
        if (size == 0) {
            last = newNode;
        }
        size++;
    }

    /**
     * Gets the node located at the given index in this list.
     *
     * @param index The index of the node to get, between 0 and size - 1
     * @return The node at the given index
     * @throws IllegalArgumentException
     *                                  If index is negative or greater than size -
     *                                  1
```

```
61      */
62     public Node getNode(int index) {
63         if (index < 0 || ((size > 0) && index > (size - 1)) || ((size == 0) && index >
   size)) {
64             throw new IllegalArgumentException("index must be between 0 and (size - 1)");
65         } else {
66             Node nodeAtIndex = first.next;
67             for (int i = 0; i < index; i++) {
68                 nodeAtIndex = nodeAtIndex.next;
69             }
70             return nodeAtIndex;
71         }
72     }
73 
74     /**
75      * Gets the memory block located at the given index in this list.
76      *
77      * @param index The index of the memory block to get, between 0 and size - 1
78      * @return The memory block at the given index
79      * @throws IllegalArgumentException
80      *                                  If index is negative or greater than size -
81      *                                  1
82      */
83     public MemBlock getBlock(int index) {
84         if (index < 0 || ((size > 0) && index > (size - 1))) {
85             throw new IllegalArgumentException("index must be between 0 and (size - 1)");
86         } else {
87             Node nodeAtIndex = getNode(index);
88             return nodeAtIndex.block;
89         }
90     }
91 
92     /**
93      * Gets the index of the node containing the given memory block.
94      *
95      * @param block The given memory block
96      * @return The index of the memory block, or -1 if the memory block is not in
97      *         this list
98      */
99     public int indexOf(MemBlock block) {
100        Node current = first.next;
101        for (int i = 0; i < size; i++) {
102            if (current.block.equals(block)) {
103                return i;
104            } else {
105                current = current.next;
106            }
107        }
108 
109        return -1;
110    }
111 
112    /**
113     * Adds a new node to this list, as follows:
114     * Creates a new node containing the given memory block,
115     * and inserts the node at the given index in this list.
116     * For example, if this list is (m7, m3, m1, m6), then
117     * add(2,m5) will make this list (m7, m3, m5, m1, m6).
118     * If the given index is 0, the new node becomes the first node in this list.
119     * If the given index equals the list's size - 1, the new node becomes the last
120     * node in this list.
```

```
121     * If the new element is added at the beginning or at the end of this list,
122     * the addition's runtime is O(1). Othewrise is it O(size).
123     *
124     * @param block The memory block to add
125     * @param index Where to insert the memory block
126     * @throws IllegalArgumentException
127     *                                  If index is negative or greater than the
128     *                                  list's size - 1
129     */
130    public void add(int index, MemBlock block) {
131        if (index < 0 || ((size > 0) && index > (size)) || ((size == 0) && index >
    (size))) {
132            throw new IllegalArgumentException("index must be between 0 and (size)");
133        } else {
134            if (index == 0) {
135                addFirst(block);
136            } else if (index == size) {
137                addLast(block);
138            } else {
139                Node newNode = new Node(block);
140                newNode.next = getNode(index);
141                getNode(index - 1).next = newNode;
142                size++;
143            }
144        }
145    }
146
147    /**
148     * Removes the first memory block from this list.
149     * Executes efficiently, in O(1).
150     *
151     * @throws IllegalArgumentException
152     *                                  If trying to remove from an empty list
153     */
154    public void removeFirst() {
155        if (size == 0) {
156            throw new IllegalArgumentException("Memory is empty, cannot remove block");
157        } else {
158            first.next = first.next.next;
159        }
160    }
161
162    /**
163     * Removes the given memory block from this list.
164     *
165     * @param block The memory block to remove
166     */
167    public void remove(MemBlock block) {
168        Node current = first;
169        while (current != null) {
170            if (current.next.block.equals(block)) {
171                current.next = current.next.next;
172                break;
173            }
174            current = current.next;
175        }
176    }
177
178    /**
179     * Returns an iterator over this list, starting with the first element.
180     *
```

```
     * @return A ListIterator object
     */
    public ListIterator iterator() {
        return new ListIterator(first.next);
    }

    /**
     * A textual representation of this list.
     *
     * @return A string representing this list
     */
    public String toString() {
        // Replace the following code with code that usese
        // StringBuilder and has the same effect.
        StringBuilder s = new StringBuilder("[ ");
        Node current = first.next; // Skips the dummy
        while (current != null) {
            s.append(current.block).append(" ");
            current = current.next;
        }
        s.append("]");
        return s.toString();
    }
}
```

```java
package MMS;

import java.util.Iterator;

/**
 * Represents a managed memory space (also called "heap"). The memory space is
 * managed by three
 * methods: <br>
 * <b> malloc </b> allocates memory blocks, <br>
 * <b> free </b> recycles memory blocks,
 * <br>
 * <b> defrag </b> reorganizes the memory space, for better allocation and
 * rescheduling.
 * <br>
 * (Part of Homework 10 in the Intro to CS course, Efi Arazi School of CS)
 */
public class MemorySpace {

    // A list that keeps track of the memory blocks that are presently allocated
    private List allocatedList;

    // A list that keeps track of the memory blocks that are presently free
    private List freeList;

    // check what is the last space that was allocated
    private int previousLength = 0;

    /**
     * Constructs a managed memory space ("heap") of a given maximal size.
     *
     * @param maxSize The size of the memory space to be managed
     */
    public MemorySpace(int maxSize) {
        // Constructs and intilaizes an empty list of allocated memory blocks, and a
        // free list containing
        // a single memory block which represents the entire memory space. The base
        // address of this single
        // memory block is zero, and its length is the given memory size (maxSize).
        allocatedList = new List();
        freeList = new List();
        freeList.addLast(new MemBlock(0, maxSize));
    }

    /**
     * Allocates a memory block.
     *
     * @param length The length (in words) of the memory block that has to be
     *               allocated
     * @return the base address of the allocated block, or -1 if unable to allocate
     */
    public int malloc(int length) {
        // Scans the freeList, looking for the first free memory block whose length
        // equals at least
        // the given length. If such a block is found, the method performs the following
        // operations:
        //
        // (1) A new memory block is constructed. The base address of the new block is
        // set to
        // the base address of the found free block. The length of the new block is set
        // to the value
```

```
61          // of the method's length parameter.
62          //
63          // (2) The new memory block is appended to the end of the allocatedList.
64          //
65          // (3) The base address and the length of the found free block are updated, to
66          // reflect the allocation.
67          // For example, suppose that the requested block length is 17, and suppose that
68          // the base
69          // address and length of the the found free block are 250 and 20, respectively.
70          // In such a case, the base address and length of of the allocated block are set
71          // to 250 and 17,
72          // respectively, and the base address and length of the found free block are
73          // updated to 267 and 3, respectively.
74          //
75          // (4) The base address of the new memory block is returned.
76          //
77          // If the length of the found block is exactly the same as the requested length,
78          // then the found block is removed from the freeList, and appended to the
79          // allocatedList.
80          Node current = freeList.getNode(0);
81          while (current != null) {
82              if (current.block.length >= length) {
83                  MemBlock newMemBlock = new MemBlock(current.block.baseAddress, length);
84                  allocatedList.addLast(newMemBlock);
85                  if (current.block.length == length) {
86                      freeList.remove(current.block);
87                  } else {
88                      current.block.baseAddress += length;
89                      current.block.length -= length;
90                  }
91                  return newMemBlock.baseAddress;
92              }
93              current = current.next;
94          }
95          if (length != previousLength) {
96              previousLength = length;
97              defrag();
98              return malloc(length);
99          }
100         return -1;
101     }
102
103     /**
104      * Frees the memory block whose base address equals the given address
105      *
106      * @param address The base address of the memory block to free
107      */
108     public void free(int address) {
109         // Adds the memory block to the free list, and removes it from the allocated
110         // list.
111         Node current = allocatedList.getNode(0);
112         while (current != null) {
113             if (current.block.baseAddress == address) {
114                 freeList.addLast(current.block);
115                 allocatedList.remove(current.block);
116                 break;
117             }
118             current = current.next;
119         }
120     }
121
```

```
122     /**
123      * A textual representation of this memory space
124      *
125      * @return a string representation of this memory space.
126      */
127     public String toString() {
128         // Returns the textual representation of the free list, a new line, and then
129         // the textual representation of the allocated list, as one string
130         String s = (freeList.toString() + "\n" + allocatedList.toString());
131         return s;
132     }
133 
134     /**
135      * Performs a defragmantation of the memory space.
136      * Can be called periodically, or by malloc, when it fails to find a memory
137      * block of the requested size.
138      */
139     public void defrag() {
140         List defragList = new List();
141         ListIterator curIterator = new ListIterator(freeList.getNode(0));
142         while (curIterator.hasNext()) {
143             ListIterator scanIterator = new ListIterator(freeList.getNode(0));
144             MemBlock temp = curIterator.current.block;
145             while (scanIterator.hasNext()) {
146                 if (temp.baseAddress + temp.length ==
    scanIterator.current.block.baseAddress) {
147                     temp.length += scanIterator.current.block.length;
148                     scanIterator = freeList.iterator();
149                 }
150                 scanIterator.next();
151             }
152             curIterator.next();
153             defragList.addLast(temp);
154         }
155         freeList = defragList;
156     }
157 
158 }
```