```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
     */
    public Node getNode(int index) {
        if (index < 0 || ((size > 0) && index > (size - 1)) || ((size == 0) && index > size)) {
            throw new IllegalArgumentException("index must be between 0 and (size - 1)");
```

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

```
117         Node theNodeBefore = getNode(index - 1);
118         theNodeBefore.next = newNode;
119         newNode.next = theNodeBefore.next;
120     }
121 
122     /**
123      * Removes the first memory block from this list.
124      * Executes efficiently, in O(1).
125      * @throws IllegalArgumentException
126      *         If trying to remove from an empty list
127      */
128     public void removeFirst() {
129         if (size == 0) {
130             throw new IllegalArgumentException("Trying to delete value from an
empty list");
131         }
132         first = first.next;
133     }
134 
135     /** Removes the given memory block from this list.
136      * @param block The memory block to remove
137      */
138     public void remove(MemBlock block) {
139         int blockIndex = indexOf(block);
140         if (blockIndex != -1) {
141             if (blockIndex == 0) {
142                 removeFirst();
143                 return;
144             }
145             Node theNodeBefore = getNode(blockIndex - 1);
146             theNodeBefore.next = getNode(blockIndex).next;
147         } else {
148             System.out.println("" + block);
149         }
150     }
151 
152     /**
153      * Returns an iterator over this list, starting with the first element.
154      * @return A ListIterator object
155      */
156     public ListIterator iterator() {
157         return new ListIterator(first.next);
158     }
159 
160     /**
161      * A textual representation of this list.
162      * @return A string representing this list
163      */
164     public String toString() {
165         StringBuilder str = new StringBuilder("[ ");
166         // Creates a pointer to the first element
167         Node current = first.next; // skips the dummy
168         while (current != null) {
169             str.append(current.block + " ");
170             current = current.next;
171         }
172         str.deleteCharAt(str.length()-1);
173         str.append(" ]");
174         return str.toString();
175     }
176 }
```

```java
// import java.util.ListIterator;

// package MMS;
/**
 * 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;

    private int oldLength = 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:
        // ListIterator iterator = freeList.iterator();
        ListIterator iterator = new ListIterator(freeList.getNode(0));
        MemBlock freeMemBlock = null;
        while (iterator.hasNext() && freeMemBlock == null) {
            MemBlock nextMeM = iterator.current.block;
            if (nextMeM.length >= length) {
                freeMemBlock = nextMeM;
            }
            iterator.next();
        }

        if (freeMemBlock != null) {
```

```
55              // (1) A new memory block is constructed. The base address of the new
    block is set to
56              //     the base address of the found free block. The length of the new
    block is set to the value
57              //     of the method's length parameter.
58              MemBlock newMemBlock = new MemBlock(freeMemBlock.baseAddress, length);
59
60              // (2) The new memory block is appended to the end of the
    allocatedList.
61              allocatedList.addLast(newMemBlock);
62
63              // (3) The base address and the length of the found free block are
    updated, to reflect the allocation.
64              //     For example, suppose that the requested block length is 17, and
    suppose that the base
65              //     address and length of the the found free block are 250 and 20,
    respectively.
66              //     In such a case, the base address and length of of the allocated
    block are set to 250 and 17,
67              //     respectively, and the base address and length of the found free
    block are updated to 267 and 3, respectively.
68              if (length == freeMemBlock.length) {
69                  freeList.remove(freeMemBlock);
70              } else {
71                  freeMemBlock.baseAddress = freeMemBlock.baseAddress + length;
72                  freeMemBlock.length = freeMemBlock.length - length;
73              }
74
75              // (4) The base address of the new memory block is returned.
76              return newMemBlock.baseAddress;
77          }
78
79          if (length != oldLength) {
80              oldLength = length;
81              defrag();
82              return malloc(length);
83          }
84          return -1;
85      }
86
87      /**
88       * Frees the memory block whose base address equals the given address
89       *
90       * @param address The base address of the memory block to free
91       */
92      public void free(int address) {
93          // Adds the memory block to the free list, and removes it from the
    allocated list.
94          // allocatedList.
95          ListIterator iterator = new ListIterator(allocatedList.getNode(0));
96          while (iterator.hasNext()) {
97              MemBlock nextMeM = iterator.current.block;
98              if (nextMeM.baseAddress == address) {
99                  freeList.addLast(nextMeM);
100                 allocatedList.remove(nextMeM);
101                 break;
102             }
103
104             iterator.next();
105         }
106     }
107
108     /**
```

```
109      * A textual representation of this memory space
110      * @return a string representation of this memory space.
111      */
112     public String toString() {
113         // Returns the textual representation of the free list, a new line, and
    then
114         // the textual representation of the allocated list, as one string
115         StringBuilder str = new StringBuilder(freeList.toString());
116         str.append("\n");
117         str.append(allocatedList.toString());
118         return str.toString();
119     }
120 
121     /**
122      * Performs a defragmantation of the memory space.
123      * Can be called periodically, or by malloc, when it fails to find a memory
    block of the requested size.
124      */
125     public void defrag() {
126         List newFreeList = freeList;
127         ListIterator iterator1 = freeList.iterator();
128         while (iterator1.hasNext()) {
129             MemBlock firstListMeM = iterator1.next();
130             MemBlock secondListMeM = null;
131             ListIterator iterator2 = newFreeList.iterator();
132             while (iterator2.hasNext()) {
133                 MemBlock tempMeM = iterator2.next();
134                 if (firstListMeM.baseAddress + firstListMeM.length ==
    tempMeM.baseAddress) {
135                     secondListMeM = tempMeM;
136                 }
137             }
138 
139             if (secondListMeM != null) {
140                 MemBlock combinedMeM = new MemBlock(firstListMeM.baseAddress,
    firstListMeM.length + secondListMeM.length);
141                 newFreeList.remove(firstListMeM);
142                 newFreeList.remove(secondListMeM);
143                 newFreeList.addLast(combinedMeM);
144             }
145         }
146 
147         freeList = newFreeList;
148     }
149 }
```