

```

1 package MMS;
2
3 /**
4  * Represents a list of Nodes. Each node holds a reference to a memory block.
5  * <br>
6  * (Part of Homework 10 in the Intro to CS course, Efi Arazi School of CS)
7  */
8 public class List {
9
10     private Node first = null; // The first (dummy) node of this list
11     private Node last = null; // The last node of this list
12     private int size = 0; // Number of elements (nodes) in this list
13
14     /**
15      * Constructs a new list of Node objects, each holding a memory block (MemBlock
16      * object)
17      */
18     public List() {
19         // Creates a dummy node and makes first and last point to it.
20         first = new Node(null);
21         last = first;
22     }
23
24     /**
25      * Adds the given memory block to the end of this list.
26      * Executes efficiently, in O(1).
27      *
28      * @param block The memory block that is added at the list's end
29      */
30     public void addLast(MemBlock block) {
31         Node newNode = new Node(block);
32         size++;
33         last.next = newNode;
34         last = newNode;
35     }
36
37     /**
38      * Adds the given memory block at the beginning of this list.
39      * Executes efficiently, in O(1).
40      *
41      * @param block The memory block that is added at the list's beginning
42      */
43     public void addFirst(MemBlock block) {
44         Node newNode = new Node(block);
45         newNode.next = first.next;
46         first.next = newNode;
47         if (size == 0) {
48             last = newNode;
49         }
50         size++;
51     }
52
53     /**
54      * Gets the node located at the given index in this list.
55      *
56      * @param index The index of the node to get, between 0 and size - 1
57      * @return The node at the given index
58      * @throws IllegalArgumentException
59      *         If index is negative or greater than size -
60      *         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             ListIterator current = new ListIterator(first.next());
67             for (int i = 0; i < index; i++) {
68                 current.next();
69             }
70             return current.current;
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             return getNode(index).block;
88         }
89     }
90
91     /**
92      * Gets the index of the node containing the given memory block.
93      *
94      * @param block The given memory block
95      * @return The index of the memory block, or -1 if the memory block is not in
96      *         this list
97      */
98     public int indexOf(MemBlock block) {
99         ListIterator current = new ListIterator(getNode(0));
100         for (int i = 0; i < size; i++) {
101             if (current.current.block.equals(block)) {
102                 return i;
103             }
104             current.next();
105         }
106
107         return -1;
108     }
109
110     /**
111      * Adds a new node to this list, as follows:
112      * Creates a new node containing the given memory block,
113      * and inserts the node at the given index in this list.
114      * For example, if this list is (m7, m3, m1, m6), then
115      * add(2,m5) will make this list (m7, m3, m5, m1, m6).
116      * If the given index is 0, the new node becomes the first node in this list.
117      * If the given index equals the list's size - 1, the new node becomes the last
118      * node in this list.
119      * If the new element is added at the beginning or at the end of this list,
120      * the addition's runtime is O(1). Otherwise is it O(size).

```

```

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

```

```
181 public ListIterator iterator() {
182     return new ListIterator(first.next);
183 }
184
185 /**
186  * A textual representation of this list.
187  *
188  * @return A string representing this list
189  */
190 public String toString() {
191     // Replace the following code with code that uses
192     // StringBuilder and has the same effect.
193     StringBuilder s = new StringBuilder("[ ");
194     ListIterator current = new ListIterator(getNode(0)); // Skips the dummy
195     while (current.hasNext()) {
196         s.append(current.current.block).append(" ");
197         current.next();
198     }
199     s.append("]");
200     return s.toString();
201 }
202 }
```

```

1 package MMS;
2
3 import java.util.Iterator;
4
5 /**
6  * Represents a managed memory space (also called "heap"). The memory space is
7  * managed by three
8  * methods: <br>
9  * <b> malloc </b> allocates memory blocks, <br>
10 * <b> free </b> recycles memory blocks,
11 * <br>
12 * <b> defrag </b> reorganizes the memory space, for better allocation and
13 * rescheduling.
14 * <br>
15 * (Part of Homework 10 in the Intro to CS course, Efi Arazi School of CS)
16 */
17 public class MemorySpace {
18
19     // A list that keeps track of the memory blocks that are presently allocated
20     private List allocatedList;
21
22     // A list that keeps track of the memory blocks that are presently free
23     private List freeList;
24
25     // check what is the last space that was allocated
26     private int previousLength = 0;
27
28     /**
29      * Constructs a managed memory space ("heap") of a given maximal size.
30      *
31      * @param maxSize The size of the memory space to be managed
32      */
33     public MemorySpace(int maxSize) {
34         // Constructs and intilaizes an empty list of allocated memory blocks, and a
35         // free list containing
36         // a single memory block which represents the entire memory space. The base
37         // address of this single
38         // memory block is zero, and its length is the given memory size (maxSize).
39         allocatedList = new List();
40         freeList = new List();
41         freeList.addLast(new MemBlock(0, maxSize));
42     }
43
44     /**
45      * Allocates a memory block.
46      *
47      * @param length The length (in words) of the memory block that has to be
48      *               allocated
49      * @return the base address of the allocated block, or -1 if unable to allocate
50      */
51     public int malloc(int length) {
52         // Scans the freeList, looking for the first free memory block whose length
53         // equals at least
54         // the given length. If such a block is found, the method performs the following
55         // operations:
56         //
57         // (1) A new memory block is constructed. The base address of the new block is
58         // set to
59         // the base address of the found free block. The length of the new block is set
60         // 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 ListIterator current = new ListIterator(freeList.getNode(0));
81 while (current.hasNext()) {
82     if (current.current.block.length >= length) {
83         MemBlock newMemBlock = new MemBlock(current.current.block.baseAddress,
length);
84         allocatedList.addLast(newMemBlock);
85         if (current.current.block.length == length) {
86             freeList.remove(current.current.block);
87         } else {
88             current.current.block.baseAddress += length;
89             current.current.block.length -= length;
90         }
91         return newMemBlock.baseAddress;
92     }
93     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     ListIterator current = new ListIterator(allocatedList.getNode(0));
112     while (current.hasNext()) {
113         if (current.current.block.baseAddress == address) {
114             freeList.addLast(current.current.block);
115             allocatedList.remove(current.current.block);
116             break;
117         }
118         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 defragmentation 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 ==
147 scanIterator.current.block.baseAddress) {
148                 temp.length += scanIterator.current.block.length;
149                 scanIterator = freeList.iterator();
150             }
151             scanIterator.next();
152         }
153         curIterator.next();
154         defragList.addLast(temp);
155     }
156     freeList = defragList;
157 }
158 }

```