CMSC 206: Data Structures
Final Exam Reference
May 2018

```
public interface BMCSet<E>
{
     /** Adds a new item to the set
      *  @param item The new item to add to the set
      *  @return true if the item is a new item added to the
      *          set; false if the item was already in the set
      */
     boolean add(E item);

     /** Checks whether this set contains an item
      *  @param item The item to check for
      *  @return true if the item is in the set; false otherwise
      */
     boolean contains(E item);

     /** Removes an item from the set
      *  @param item The item to remove
      *  @return true iff the item was in the set and removed
      */
     boolean remove(E item);

     /** Removes all items from the set */
     void clear();

     /** @return true if the set is empty; false otherwise */
     boolean isEmpty();

     /** @return the number of elements in this set */
     int size();
}

public class BinarySearchTree<E extends Comparable<E>>
  implements BMCSet<E>, Iterable<E>
{ public BinarySearchTree() { … } /* … */ }

public class BMCHashSet<E> implements BMCSet<E>, Iterable<E>
{ public BMCHashSet() { … } /* … */ }
```

```
public interface BMCMap<K,V>
{
     /** Associates the specified value with the specified key.
      *  @param key The key in the association
      *  @param value The value to associate that key with.
      */
     void put(K key, V value);

     /** Retrieves the value associated with a key.
      *  @param key The key to look up.
      *  @return The value associated with that key, or null if
      *          none exists
      */
     V get(K key);

     /** Removes a mapping from the given key.
      *  @param key The key whose mapping should be removed.
      */
     void remove(K key);

     /** Checks to see whether a key is mapped.
      *  @param key The key to check
      *  @return true if this map maps the given key; false
      *          otherwise
      */
     boolean containsKey(K key);

     /** @return true iff there are no associations in map */
     boolean isEmpty();

     /** @return The number of mappings in this map */
     int size();

     /** Removes all entries from this map. */
     void clear();
}

public class BinarySearchTreeMap<K extends Comparable<K>,V>
  implements BMCMap<K,V>
{ public BinarySearchTreeMap() { … } /* … */ }
```

```
public class BMCHashMap<K,V> implements BMCMap<K,V>
{
     // the array of linked lists
     private LinkedList<Entry<K,V>>[] table;

     private int size; // the number of items in the map

     // the default number of slots in the table
     private final static int NUM_INITIAL_SLOTS = 10;

     /** Construct an empty BMCHashMap */
     public BMCHashMap()
     {
          table = newArray(NUM_INITIAL_SLOTS);
          size = 0;
     }

     /** Choose the slot number an item should belong in
      *  @param k The item to look up
      *  @return The slot number.
      *          0 <= slot number < table.length
      */
     private int hash(K k)
     {
          return Math.abs(x.hashCode()) % table.length;
     }

     @Override
     public void put(K k, V v)
     {
          int hash = hash(k);

          if(table[hash] == null)
          {
               table[hash] = new LinkedList<>();
          }

          for(Entry<K,V> elem : table[hash])
          {
               if(k.equals(elem.getKey()))
               {
                    elem.setValue(v);
               }
          }

          table[hash].add(new Entry<>(k, v));
          size++;
```

```
}

@Override
public V get(K k)
{
     int hash = hash(k);

     if(table[hash] == null)
     {
          return null;
     }

     for(Entry<K,V> elem : table[hash])
     {
          if(k.equals(elem.getKey()))
          {
               return elem.getValue();
          }
     }

     return null;
}

@Override
public boolean containsKey(K k)
{
     int hash = hash(k);

     if(table[hash] == null)
     {
          return false;
     }

     for(Entry<K,V> elem : table[hash])
     {
          if(k.equals(elem.getKey()))
          {
               return true;
          }
     }

     return false;
}

@Override
public void remove(K k)
{
```

```
        int hash = hash(k);

        if(table[hash] == null)
        {
            return;
        }

        for(Iterator<Entry<K,V>> itor =
              table[hash].iterator(); itor.hasNext(); )
        {
            Entry<K,V> entry = itor.next();
            if(k.equals(entry.getKey()))
            {
                itor.remove();
                size--;
                return;
            }
        }
    }

    @Override
    public int size()
    {
        return size;
    }

    @Override
    public void clear()
    {
        table = newArray(NUM_INITIAL_SLOTS);
        size = 0;
    }

    @Override
    public boolean isEmpty()
    {
        return size == 0;
    }


    /** Allocate space for an array of linked lists.
     *  @param len The desired number of slots
     *  @return The array of linked lists to use as the table
     */
    @SuppressWarnings("unchecked")
    private static <E> LinkedList<E>[] newArray(int len)
    {
```

```
            return (LinkedList<E>[])new LinkedList<?>[len];
        }
}

public class Entry<K,V>
{
        private K key;
        private V value;

        /** Constructs an entry with the given key and value
         *  @param k The key
         *  @param v The value
         */
        public Entry(K k, V v)
        {
            key = k;
            value = v;
        }

        /** @return The key in this Entry */
        public K getKey()
        {
            return key;
        }

        /** @return The value in this Entry */
        public V getValue()
        {
            return value;
        }

        /** Updates the value in this Entry
         *  @param v The new value
         */
        public void setValue(V v)
        {
            value = v;
        }
}
```

```
public class Node
{
     public String data; // the data stored at this node
     public Node next; // the next node of data

     /** Creates a new node with a null next field
      *  @param dataItem The data stored
      */
     public Node(String dataItem)
     {
          data = dataItem;
          next = null;
     }

     /** Creates a new node that references another node.
      *  @param dataItem The data stored
      *  @param nodeRef The node referenced by new node
      */
     public Node(String dataItem, Node nodeRef)
     {
          data = dataItem;
          next = nodeRef;
     }
}
```

```
public class SingleLinkedList implements Iterable<String>
{
     private Node head = null; // reference to the list head
     private int size; // number of items in the list

     /** Add an item to the front of the list. Runs in constant
      *  time.
      *  @param item The item to be added
      */
     public void addFirst(String item) { … }

     /** Add a node after a given node. Runs in constant time.
      *  @param node The node preceding the new item
      *  @param item The item to insert
      */
     private void addAfter(Node node, String item) { … }

     /** Insert the specified item at index. Runs in O(n) time,
      *  where n is the index.
      *  @param index The position where item is to be inserted
      *  @param item The item to be inserted
      *  @throws IndexOutOfBoundsException if index is out
      *          of range
      */
     public void add(int index, String item) { … }

     /** Append item to the end of the list. Runs in O(n) time,
      *  where n is the size of the linked list.
      *  @param item The item to be appended
      *  @return true, always
      */
     public boolean add(String item) { … }

     /** Remove the first node from the list. Runs in constant
      *  time.
      *  @return The removed node's data or null if the list
      *          is empty
      */
     public String removeFirst() { … }

     /** Remove the node after a given node. Runs in constant
      *  time
      *  @param node The node before the one to be removed
      *  @return The data from the removed node, or null if
      *          there is no node to remove
      */
     private String removeAfter(Node node) { … }
```

```
    /** Remove the node at the given index. Runs in O(n) time,
     *  where n is the index.
     *  @param index The position of the element to remove
     *  @return The removed data
     *  @throws IndexOutOfBoundsException if the index is ou
     *          of bounds
     */
    public String remove(int index) { … }

    /** Get the data value at index. Runs in O(n) time, where
     *  n is the index.
     *  @param index The position of the element to return
     *  @return The data at index
     *  @throws IndexOutOfBoundsException if index is out of
     *          range
     */
    public String get(int index) { … }

    /** Set the data value at index. Runs in O(n) time, where
     *  n is the index.
     *  @param index The position of the item to change
     *  @param newValue The new value
     *  @return The data value previously at index
     *  @throws IndexOutOfBoundsException if index is out of
     *          range
     */
    public String set(int index, String newValue) { … }

    /** Find the node at a specified position. Runs in O(n)
     *  time, where n is the index
     *  @param index The position of the node sought
     *  @return The node at index or null if it does not exist
     */
    private Node getNode(int index) { … }

    /** Returns an Iterator over the strings in this list.
     *  Runs in O(1) time.
     *  @return an iterator over this list
     */
    public Iterator<String> iterator() { … }
}
```

```
public interface Iterable<E>
{
    /** Returns an iterator over elements of type E.
     *  @return an Iterator.
     */
    Iterator<E> iterator();
}

public class String
{
     // Returns the char value at the specified index
     public char charAt(int index) { … }

     // Compares two strings lexicographically
     public int compareTo(String other) { … }

     // Returns true if and only if this string contains the
     // specified string
     public boolean contains(String s) { … }

     // Returns a hash code for this string
     public int hashCode() { … }

     // Returns the length of this string
     public int length() { … }

     /** Returns a string that is a substring of this string
      *  The substring begins with the character at the
      *  specified index and extends to the end of this string.
      *  @param beginIndex The beginning index, inclusive
      *  @return The specified substring
      */
     public String substring(int beginIndex) { … }

     /** Returns a string that is a substring of this string.
      *  The substring begins at the specified beginIndex and
      *  extends to the character at index (endIndex - 1). Thus
      *  the length of the substring is endIndex - beginIndex.
      *  @param beginIndex the beginning index, inclusive
      *  @param endIndex the ending index, exclusive
      *  @return the specified substring
      */
     public String substring(int beginIndex, int endIndex) { … }
}

public class LinkedList<E> implements Iterable<E>, Queue<E>
{ … }
```

```
public interface Queue<E>
{
    /** Inserts the specified element into this queue.
     *  @param e the element to add
     *  @return true, always
     */
    boolean add(E e);

    /** Retrieves and removes the head of this queue.
     *  @return the head of this queue
     *  @throws NoSuchElementException if this queue is empty
     */
    E remove();


    /** @return true iff this collection contains no elements
     */
    boolean isEmpty();

    // other methods not shown
}

public interface StackInt<E>
{
     /** Pushes an item onto the top of the stack and
      *  returns the item pushed.
      *  @param obj The object to be inserted
      *  @return The object inserted
      */
     E push(E obj);

     /** Return the object at the top of the stack, removing it.
      *  Post-condition: The stack is one item smaller
      *  @return The object at the top of the stack
      *  @throws EmptyStackException if stack is empty
      */
     E pop();

     /** @return true if the stack is empty
      */
     boolean isEmpty();
}
```

```
public interface Iterator<E>
{
    /**
     * Returns true if the iteration has more elements.
     * (In other words, returns true if next() would
     * return an element rather than throwing an exception.)
     *
     * @return true if the iteration has more elements
     */
    boolean hasNext();

    /**
     * Returns the next element in the iteration.
     *
     * @return the next element in the iteration
     * @throws NoSuchElementException if the iteration has no
     *         more elements
     */
    E next();

    /**
     * Removes from the underlying collection the last element
     * returned by this iterator (optional operation).
     * This method can be called only once per call to next().
     *
     * @throws UnsupportedOperationException if the remove
     *         operation is not supported by this iterator
     *
     * @throws IllegalStateException if the next method has not
     *         yet been called, or the remove method has already
     *         been called after the last call to the next
     *         method
     */
    void remove();
}
```