Assignment 4: Collection Implementation—Sets

Assigned: Thursday, February 27, 2014
Due: Sunday, March 16, 2014, 11:59 P.M.
Type: Individual

Problem Overview

This assignment requires you to implement a *set* collection using a doubly-linked list as the underlying data structure. You are provided the `Set` interface and a shell of the `LinkedSet` implementing class. You must not change anything in the `Set` interface; you must only provide correct implementations of the methods in the `LinkedSet` class. You must use the provided inner class `Node` to build the internal linked list, and you may not change the `Node` class in any way. You may add any number of inner classes that you need (e.g., for iteration), but you may not create or use any other top-level class. You must also use without modification the three fields of the `LinkedSet` class: `front`, `rear`, and `size`.

Set Methods

Each method from the `Set` interface that you must implement in the `LinkedSet` class is described below and in comments in the provided source code. You must read both. Note that in addition to correctness, your code will also be graded on the stated performance requirements.

add(T element)

The `add` method ensures that this set contains the specified element. Neither duplicates nor null values are allowed. The method returns true if this set was modified (i.e., the element was added) and false otherwise. Note that the constraints on the generic type parameter `T` of the `Set` interface require that any type bound by a client to `T` be a class that implements the Comparable interface for that type. Thus, there is a *natural order* on the values stored in a set.

The interface states that no specific order can be assumed (by a client), and therefore allows the `add` method to maintain any order we choose or none at all. In the `LinkedSet` implementation, you must maintain the internal linked list in *ascending natural order* at all times. The time complexity of the `add` method must be $O(N)$, where N is the number of elements in the set.

remove(T element)

The `remove` method ensures that this set does not contain the specified element. The method returns true if this set was modified (i.e., an existing element was removed) and false otherwise. The `remove` method must maintain the ascending natural order of the linked list. The time complexity of the `remove` method must be $O(N)$, where N is the number of elements in the set.

contains(T element)

The `contains` method searches for the specified element in this set, returning true if the element is in this set and false otherwise. The time complexity of the `contains` method must be $O(N)$, where N is the number of elements in the set.

size()

The `size` method returns the number of elements in this set. *This method is provided for you and must not be changed*. The time complexity of the `size` method is O(1).

isEmpty()

The `isEmpty` method returns true if there are no elements in this set and false otherwise. *This method is provided for you and must not be changed*. The time complexity of the `isEmpty` method is O(1). Any set for which `isEmpty()` returns true is considered the "empty set" (∅) for purposes of union, intersection, and complement.

equals(Set<T> s)

Two sets are equal if and only if they contain exactly the same elements, regardless of order. If $A = \{1,2,3\}, B = \{3,1,2\}$, and $C = \{1,2,3,4\}$, then $A = B$ and $A \neq C$. The `equals` method returns true if this set contains exactly the same elements as the parameter set, and false otherwise. The time complexity of the `equals` method must be $O(N \times M)$ where N is the size of this set and M is the size of the parameter set.

union(Set<T> s)

The *union* of set A with set B, denoted $A \cup B$, is defined as $\{x \mid x \in A \text{ or } x \in B\}$. Note that $A \cup B = B \cup A$ and $A \cup \emptyset = A$. The `union` method returns a set that is the *union* of this set and the parameter set; that is, the set that contains the elements of both this set and the parameter set. The result set must be in ascending natural order. The time complexity of the `union` method must be $O(N \times M)$ where N is the size of this set and M is the size of the parameter set.

intersection(Set<T> s)

The *intersection* of set A with set B, denoted $A \cap B$, is defined as $\{x \mid x \in A \text{ and } x \in B\}$. Note that $A \cap B = B \cap A$ and $A \cap \emptyset = \emptyset$. The `intersection` method returns a set that is the *intersection* of this set and the parameter set; that is, the set that contains the elements of this set that are also in the parameter set. The result set must be in ascending natural order. The time complexity of the `intersection` method must be $O(N \times M)$ where N is the size of this set and M is the size of the parameter set.

complement(Set<T> s)

The *relative complement* of set A with respect to set B, denoted $A \backslash B$, is defined as $\{x \mid x \in A \text{ and } x \notin B\}$. Note that $A \backslash B \neq B \backslash A, A \backslash \emptyset = A$, and $\emptyset \backslash A = \emptyset$. The `complement` method returns a set that is the *relative complement* of this set with respect to the parameter set; that is, the set that contains the elements of this set that are not in the parameter set. The result set must be in ascending natural order. The time complexity of the `complement` method must be $O(N \times M)$ where N is the size of this set and M is the size of the parameter set.

iterator()

The `iterator` method returns an Iterator over the elements in this set. Although the interface specifies that no particular order can be assume (by a client), this implementation must ensure that the resulting iterator returns the elements in ascending natural order. The associated time complexities are as follows: `iterator()`: $O(1)$; `hasNext()`: $O(1)$; `next()`: $O(1)$.

LinkedSet Methods

In addition to the methods from the `Set` interface, the `LinkedSet` class also implements its own methods, typically to take advantage of the underlying representation. Each of these class-specific methods is described below, as well as in comments in the provided source code. You must read both. Note that in addition to correctness, your code will also be graded on the stated performance requirements.

Constructors

The only public constructor that is allowed has been provided for you and you must not change it in any way. You may, however, find it helpful to write your own *private* constructor; one that offers direct support for doubly-linked lists.

equals(LinkedSet<T> s)

The semantics of this overloaded method is identical to the `equals` method described above. However, since the parameter is typed as a `LinkedSet`, this method can directly access the doubly-linked list in this set as well as in the parameter set. Having access to the underlying representation allows a more efficient algorithm for this method. The time complexity of this `equals` method must be $O(N)$ where N is the size of this set.

union(LinkedSet<T> s)

The semantics of this overloaded method is identical to the `union` method described above. However, since the parameter is typed as a `LinkedSet`, this method can directly access the doubly-linked list in this set as well as in the parameter set. Having access to the underlying representation allows a more efficient algorithm for this method. The time complexity of this `union` method must be $O(max(N, M))$ where N is the size of this set and M is the size of the parameter set.

intersection(LinkedSet<T> s)

The semantics of this overloaded method is identical to the `intersection` method described above. However, since the parameter is typed as a `LinkedSet`, this method can directly access the doubly-linked list in this set as well as in the parameter set. Having access to the underlying representation allows a more efficient algorithm for this method. The time complexity of this `intersection` method must be $O(min(N, M))$ where N is the size of this set and M is the size of the parameter set.

complement(LinkedSet<T> s)

The semantics of this overloaded method is identical to the `complement` method described above. However, since the parameter is typed as a `LinkedSet`, this method can directly access the doubly-linked list in this set as well as in the parameter set. Having access to the underlying representation allows a more efficient algorithm for this method. The time complexity of this `complement` method must be $O(N)$ where N is the size of this set.

descendingIterator()

The `descendingIterator` method returns an Iterator over the elements in this set in descending natural order. The associated time complexities are as follows: `iterator()`: $O(1)$; `hasNext()`: $O(1)$; `next()`: $O(1)$.

powersetIterator()

The *power set* of a set S, denoted $\mathcal{P}(S)$, is defined as $\{T \mid T \subseteq S\}$; that is, the set of all subsets of S. There are 2^N members of $\mathcal{P}(S)$ where N is the number of elements in S. For example, if $S = \{A, B, C\}$, then $\mathcal{P}(S) = \{\emptyset, \{A\}, \{B\}, \{C\}, \{A, B\}, \{B, C\}, \{A, C\}, \{A, B, C\}\}$. (Note that the empty set $\emptyset$ is a member of every set.) The `powersetIterator` method returns an Iterator over the elements in the *power set* of this set. The iterator makes to guarantees regarding the order in which the elements of $\mathcal{P}(S)$ will be returned. The associated time complexities are as follows: `iterator()`: $O(N)$; `hasNext()`: $O(1)$; `next()`: $O(N)$, where N is the size of this set.

Assignment Submission

You must turn in only the `LinkedSet.java` file to Web-CAT no later than the published deadline. Submissions made within the 24 hour period after the published deadline will be assessed a late penalty of 15 points. No submissions will be accepted more than 24 hours after the published deadline.