Implementing a `SortedLinkedSet` in Java

Data Structures. Pepe Gallardo, 2025.

Dpto. Lenguajes y Ciencias de la Computación. University of Málaga

Objective

In the last lab session, we delved into the world of sorted sets by implementing a `SortedArraySet` using an array-based structure. This week, we'll build upon that knowledge and implement a `SortedLinkedSet` in Java. This data structure utilizes a sorted linked structure to store elements, ensuring that all elements are unique and sorted in ascending order. By creating a `SortedLinkedSet`, you will gain firsthand experience with sorted linked structures and maintaining invariants in a sorted linked data structure.

Introduction to SortedLinkedSet

In our previous lab, we learned that sorted sets are specialized versions of sets that maintain their elements in a specific order. We implemented this concept using an array, which offered advantages such as:

1. Simple random access to elements
2. Efficient use of memory for a fixed number of elements

However, the array-based implementation also had some limitations:

1. Fixed size, requiring resizing operations
2. Potentially inefficient insertions and deletions, especially for large sets as elements need to be shifted

This week, we'll address these limitations by implementing a `SortedLinkedSet` using a linked structure. This approach offers several advantages:

1. Dynamic size without explicit resizing operations
2. Insertions and deletions at any position without shifting elements
3. Good performance for large, frequently modified sets

Learning Objectives

By the end of this lab session, you will:

1. Understand the differences between array-based and linked implementations of sorted sets
2. Implement core operations (insert, delete, contains) for a `SortedLinkedSet`
3. Analyze the time complexity of these operations compared to the `SortedArraySet`
4. Gain practical experience in working with linked structures in Java

Real-World Applications

The applications of sorted sets remain the same as in our previous lab. However, the choice between an array-based or linked implementation can significantly impact performance depending on the specific use case.

As we progress through this lab, consider how the characteristics of a linked implementation might be advantageous or disadvantageous for different applications compared to the array-based approach we explored last week.

Instructions

Class Definition

Begin by defining a class `SortedLinkedSet<T>` that implements the `SortedSet<T>` interface. This class will use a generic type `T` and will accept a `Comparator<T>` during construction to compare elements and establish the sorting order. The main goal of this class is to manage a collection of **unique** elements in a sorted order within a linked structure.

Attributes

Your `SortedLinkedSet` will require the following key attributes:

```
private Node<T> first;             // Reference to the first node
private Comparator<T> comparator; // Comparator to define the order of elements
private int size;                  // Number of elements in the set
```

You will also need to define an inner `Node` class to represent the nodes in the linked structure. This class should contain an element of type `T` and a reference to the next node in the sequence:

```
private static final class Node<E> {
    E element;    // Element stored in the node
    Node<E> next; // Reference to the next node

    Node(E element, Node<E> next) {
        this.element = element;
        this.next = next;
    }
}
```

Constructors

The constructor of `SortedLinkedSet` will initialize the comparator with the one provided as an argument, set the `first` reference to null, and set the initial size to zero. This setup ensures that the set starts empty and is ready to sort elements as they are added.

```
public SortedLinkedSet(Comparator<T> comparator) {
    this.comparator = comparator;
    this.first = null;
    this.size = 0;
}
```

Invariants

It is crucial to maintain certain invariants in your implementation after every operation:

- The linked structure should always be sorted in ascending order according to the comparator.
- No duplicate elements should be present in the set.
- The `first` reference should always point to the node containing the smallest element in the set (or null if the set is empty).
- The `next` reference of the last node should always be null.
- The `size` attribute should accurately reflect the number of elements in the set.

The diagram in figure 1 illustrates a `SortedLinkedSet` containing four elements.

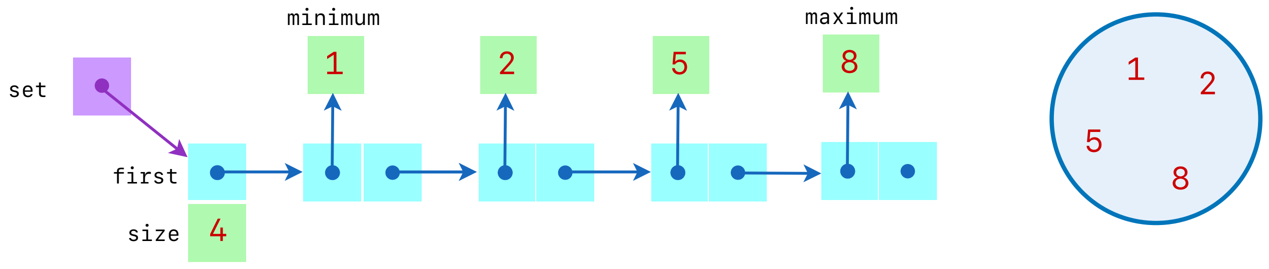


Figure 1: A `SortedLinkedSet` instance representing the sorted set on the right.

The Finder Class: A Crucial Auxiliary Tool

The `Finder` inner class is an essential component of the `SortedLinkedSet` implementation. It serves as a powerful utility for efficiently locating elements or determining insertion points within the sorted linked structure. Understanding the `Finder` class is key to implementing effective insert, delete and search operations in the `SortedLinkedSet`. This class is defined as follows:

```java
private final class Finder {
    boolean found;             // Indicates if the target element was found
    Node<T> previous, current; // References to the previous and current nodes

    Finder(T element) { // Constructor takes the target element
        previous = null;
        current = first;

        int cmp = 0;
        while (current != null && (cmp = comparator.compare(element, current.element)) > 0) {
            previous = current;
            current = current.next;
        }

        found = current != null && cmp == 0;
    }
}
```

Attributes Explained. The `Finder` class has three key attributes:

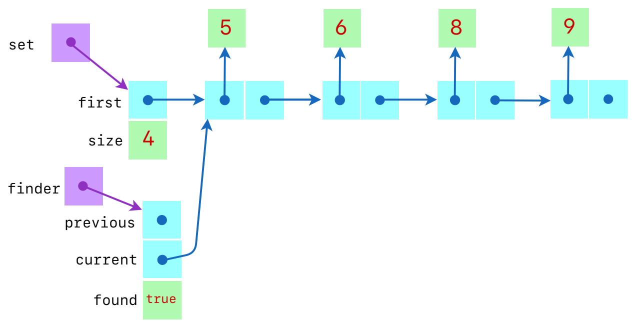
1. `boolean found`:
   - This flag indicates whether the target element was found in the set.
   - It is set to `true` if an exact match is found, `false` otherwise.
2. `Node<T> previous`:
   - This is a reference to the node that comes before the target element or its insertion point.
   - It is useful for insertion and deletion operations, as it allows direct manipulation of the links in the structure.
   - If the target element should be (or is) at the beginning of the list, `previous` will be `null`.
3. `Node<T> current`:
   - This is a reference to the node containing the target element or its insertion point.
   - After the search, if the target element is found, `current` points to the node containing the element.
   - If the target element is not found, `current` points to the first node greater than the target (or `null` if the target would be inserted at the end).

Constructor Behavior. The constructor of the `Finder` class takes a target element as a parameter and performs a traversal of the linked structure to search for the element or to determine its insertion point. Here's a step-by-step breakdown of its behavior:

1. Initialize `previous` to `null` and `current` to the first (minimum) node of the set.
2. Enter a while loop that continues as long as:
   - `current` is not `null` (we haven't reached the end of the list), and
   - The target element is greater than the current element (according to the comparator).
3. Inside the loop:
   - Move `previous` to the current node.
   - Move `current` to the next node.
4. After the loop ends:
   - If `current` is not `null` and the target element equals the current element, set `found` to `true`.
   - Otherwise, set `found` to `false`.

Possible Output States. The `Finder` class can end up in one of three possible states after its construction:

1. **Target Element Found**:
   - `found` is `true`.
   - `current` points to the node containing the target element.
   - `previous` points to the node before the target, or `null` if the target element is in the first node (see figure 2 where the target element is 5).
2. **Target Element Not Found, Insertion Point Determined**:
   - `found` is `false`.
   - `current` points to the first node greater than the target element, or `null` if the target should be inserted at the end.
   - `previous` points to the last node less than the target element, or `null` if the target should be inserted at the beginning (see figures 3 and 4 (left) where the target element is 5).
3. **Empty List**:
   - `found` is `false`.
   - `current` is `null`.
   - `previous` is `null` (see figure 4 (right) where the target is any value).



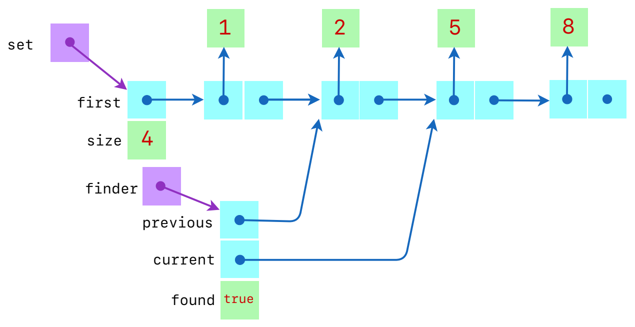


Figure 2: The `Finder` class when the target element (5) is found and it is the first one (left) or when it is not the first one (right).

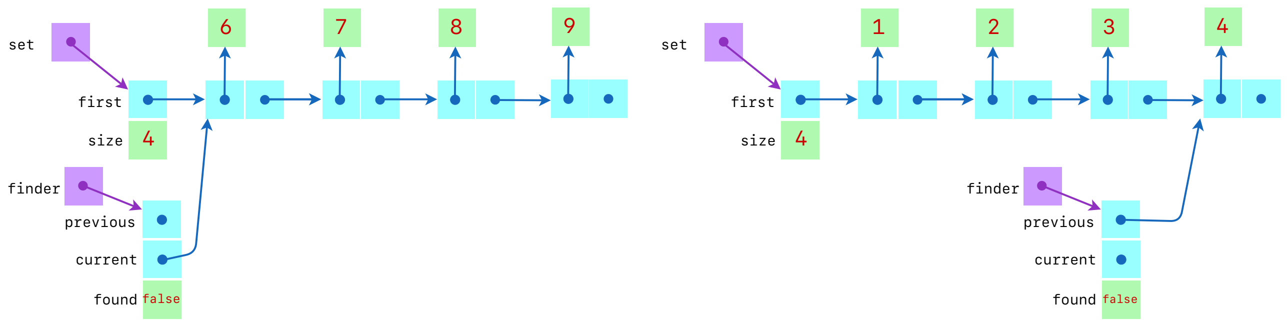


Figure 3: The `Finder` class when the target element (5) is not found and it should be inserted at the beginning (left) or at the end (right).

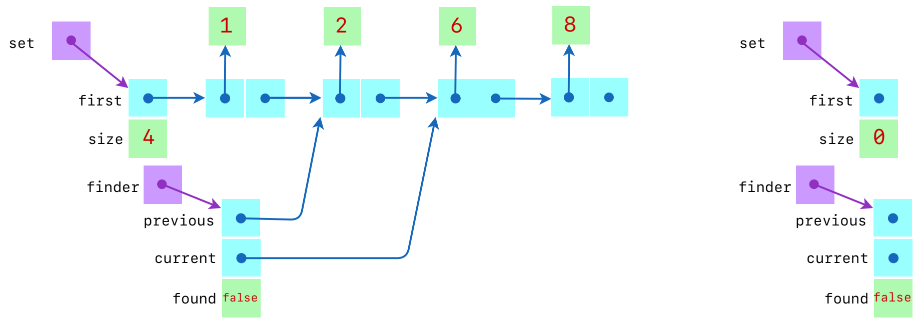


Figure 4: The `Finder` class when the target element (5) is not found and it should neither be inserted at the end or at the beginning (left) or when the list is empty (right).

Utilizing the Finder Class. The `Finder` class simplifies several key operations in the `SortedLinkedSet`:

1. **Insertion**:
   - If `found` is `true`, replace the existing element with the new one.
   - If `found` is `false` and `previous` is not `null`, insert the new element between `previous` and `current`.
   - If `found` is `false` and `previous` is `null`, update the `first` reference of the set.
2. **Deletion**:
   - If `found` is `true` and `previous` is not `null`, remove the `current` node by updating the `next` reference of `previous`.
   - If `found` is `true` and `previous` is `null`, update the `first` reference of the set.
3. **Search**:
   - The `contains` method can simply return the value of `found`.

By encapsulating the traversal logic, the `Finder` class promotes code reuse and simplifies the implementation of the `SortedLinkedSet` operations. It is a prime example of how auxiliary classes can enhance the design and efficiency of data structures.

Methods

The class `SortedLinkedSet` implements the `SortedSet<T>` interface, which requires specific methods to be implemented. This interface extends the `Set<T>` interface and adds additional methods related to sorting, so methods from both interfaces should be implemented:

```
public interface Set<T> extends Iterable<T> {
    void insert(T element);
    void delete(T element);
    boolean contains(T element);
    boolean isEmpty();
    int size();
    void clear();
}

public interface SortedSet<T> extends Set<T> {
    Comparator<T> comparator();
    T minimum();
    T maximum();
}
```

Here is a detailed explanation of each method you need to implement:

- **`void insert(T element)`**. The `insert` method adds the specified element to the set while maintaining the sorted order and ensuring no duplicates. If the element already exists, the element in the set will be replaced by the new one (ensuring uniqueness).
- **`void delete(T element)`**. The `delete` method removes the specified element from the set if it exists, maintaining the sorted order of the remaining elements. If the element is not found, no action is taken and no exception should be thrown.
- **`boolean contains(T element)`**. The `contains` method checks if the specified element is present in the set and returns `true` if the element is found, otherwise returns `false`.
- **`boolean isEmpty()`**. The `isEmpty` method checks if the set is empty and returns `true` if there are no elements in the set, otherwise returns `false`.
- **`int size()`**. The `size` method returns the number of elements currently in the set.
- **`void clear()`**. The `clear` method removes all elements from the set, resetting its size to zero and setting the `first` reference to null.
- **`Comparator<T> comparator()`**. The `comparator` method returns the comparator being used to order the elements in the set.
- **`T minimum()`**. The `minimum` method returns the smallest element in the set according to the comparator. If the set is empty, this method should handle it appropriately by throwing a `NoSuchElementException`.
- **`T maximum()`**. The `maximum` method returns the largest element in the set according to the comparator. If the set is empty, this method should handle it appropriately by throwing a `NoSuchElementException`.

Factory Methods

Factory methods provide a flexible way to create instances of `SortedLinkedSet` with different initial configurations. These methods enhance usability by offering multiple ways to instantiate the set, catering to various use cases and preferences. You should implement the following factory methods:

- `empty()`.
  - Purpose: Creates an empty `SortedLinkedSet` using the natural ordering of elements. .
  - Use case: When you need an empty set and the elements have a natural ordering (they implement `Comparable`) that you want to use.
- `empty(Comparator<T> comparator)`.

    - Purpose: Creates an empty `SortedLinkedSet` with a custom comparator.
    - Use case: When you need an empty set with a specific ordering that differs from the natural ordering of elements.

- `of(T... elements)`.
    - Purpose: Creates a `SortedLinkedSet` containing the specified elements, using their natural ordering.
    - Use case: When you want to create a set with initial elements that have a natural ordering.

- `of(Comparator<T> comparator, T... elements)`.
    - Purpose: Creates a `SortedLinkedSet` with a custom comparator, containing the specified elements.
    - Use case: When you want to create a set with initial elements and a specific ordering.

- `copyOf(SortedSet<T> set)`.
    - Purpose: Creates a new `SortedLinkedSet` containing all elements from an existing `SortedSet` and using the provided set's comparator.
    - Use case: When you need to create a copy of an existing sorted set, potentially to modify it without affecting the original.

Making SortedLinkedSet Iterable

Create a private inner `SortedLinkedSetIterator` class that implements the `Iterator<T>` interface. It should return set elements in ascending order. The iterator should traverse the linked structure, starting from the `first` node and moving to the next node on each call to `next()`. For this purpose, the `SortedLinkedSetIterator` class should have an attribute to keep track of the current node being visited. Implement the `iterator()` method in class `SortedLinkedSet` to return an `Iterator<T>` object (a new instance of `SortedLinkedSetIterator`).

The SortedLinkedSetBuilder Class: Optimizing Set Operations

When implementing set operations like union, intersection, and difference for `SortedLinkedSet`, we face a challenge. While the sorted nature of our sets allows for efficient comparisons, building the result set by appending elements to the end becomes costly (O(n) for each append) because `SortedLinkedSet` only has a reference to the first node.

To overcome this limitation and optimize set operations, we introduce the `SortedLinkedSetBuilder` inner class. This auxiliary class allows for efficient construction of a new sorted linked structure when the elements are already sorted and is defined as follows:

```java
private static final class SortedLinkedSetBuilder<T> {
    Node<T> first, last;
    int size;
    Comparator<T> comparator;

    SortedLinkedSetBuilder(Comparator<T> comparator) {
        this.first = null;
        this.last = null;
        this.size = 0;
        this.comparator = comparator;
    }

    void append(T element) {
        assert first == null || comparator.compare(element, last.element) > 0;

        Node<T> node = new Node<>(element, null);
        if (first == null) { // builder was empty
            first = node;
        } else {
            last.next = node;
        }
        last = node;
        size++;
    }

    SortedLinkedSet<T> toSortedLinkedSet() { // turns the builder into a SortedLinkedSet in O(1) time
        return new SortedLinkedSet<>(this);
    }
}
```

Class Structure. Let's break down the key components of this class:

1. **Attributes:**
   - `first` and `last`: References to the first and last nodes of the sorted linked structure being built.
   - `size`: Keeps track of the number of elements in the structure.
   - `comparator`: Defines the order of elements.
2. **Constructor:**
   - Initializes an empty structure with the given comparator.
3. **`append` method:**
   - Adds a new element to the end of the linked structure in O(1) time.
   - Uses an assert statement to ensure the appended element is greater than the last element, maintaining the sorted order.
   - Updates `first` and `last` references as needed.
4. **`toSortedLinkedSet` method:**
   - Creates and returns a new `SortedLinkedSet` from the built structure in O(1) time.

Using SortedLinkedSetBuilder. To use this builder in the `SortedLinkedSet` class, a new constructor is added:

```
private SortedLinkedSet(SortedLinkedSetBuilder<T> builder) {
    this.first = builder.first;
    this.size = builder.size;
    this.comparator = builder.comparator;
}
```

This constructor allows for the efficient creation of a `SortedLinkedSet` from a completed `SortedLinkedSetBuilder`.

Advantages of SortedLinkedSetBuilder

1. **Efficiency:** Allows O(1) appends to the end of the structure, significantly improving performance for building new sets.
2. **Maintaining Order:** Optimizes the construction of sorted sets when elements are already sorted, avoiding additional sorting steps.
3. **Simplicity:** Simplifies the implementation of set operations by providing a clean interface for building new sets.

Set Operations

Add static methods to perform set operations:

- `union`. Implement a static method `union` that takes two `SortedSet` instances and returns a new `SortedLinkedSet` containing all elements from both sets, without duplicates.
- `intersection`. Create a static method `intersection` that takes two `SortedSet` instances and returns a new `SortedLinkedSet` containing only the elements present in both sets.
- `difference`. Develop a static method `difference` that takes two `SortedSet` instances (`sortedSet1` and `sortedSet2`) and returns a new `SortedLinkedSet` containing elements that are in `sortedSet1` but not in `sortedSet2`.

With the `SortedLinkedSetBuilder`, we can now implement these set operations more efficiently. The key to these implementations is to take advantage of the sorted nature of the input sets and the efficiency of the builder class.

Optimizing Set Operations.

1. **Union:**
   - Create a new `SortedLinkedSetBuilder`.
   - Traverse both input sets simultaneously, comparing elements.
   - Append the smaller element (or one of them if equal) to the builder.
   - After traversal, append any remaining elements from the longer set.
2. **Intersection:**
   - Create a new `SortedLinkedSetBuilder`.
   - Traverse both input sets simultaneously.
   - When equal elements are found, append to the builder.
   - Advance the pointer in the set with the smaller current element.
3. **Difference:**
   - Create a new `SortedLinkedSetBuilder`.
   - Traverse both sets simultaneously.
   - Append elements from the first set that are not in the second set.
   - Advance pointers based on element comparisons.

After building the result using `SortedLinkedSetBuilder`, call `toSortedLinkedSet()` to create the final `SortedLinkedSet`.

By utilizing `SortedLinkedSetBuilder`, we can implement union, intersection, and difference operations with a time complexity of O(n + m), where n and m are the sizes of the input sets. This is a significant improvement over naive implementations that might require O(n log n) time for sorting the resulting sets or repeated O(n) insertions.

Conclusion

This lab session has built upon the foundation laid in our previous exploration of the `SortedArraySet`, providing you with a deeper understanding of sorted set implementations and the trade-offs between array-based and linked structures.

By implementing both array-based and linked versions of a sorted set, you've gained valuable insights into the strengths and weaknesses of each approach:

1. **Dynamic Size**: Your `SortedLinkedSet` can grow and shrink dynamically without the need for explicit resizing operations, unlike the `SortedArraySet` which required careful management of array capacity.

2. **Locating an Element or Insertion Point**: The `Finder` class in the array-based implementation allowed for efficient binary search, while the linked implementation required a linear traversal. This difference impacts the time complexity of operations like `contains`, `insert`, and `delete`.
3. **Insertion and Deletion Efficiency**: You've seen how the linked structure allows for more efficient insertions and deletions, especially in the middle of the set, compared to the potentially costly element shifting in the array-based version.
4. **Memory Usage**: While the `SortedArraySet` had a more compact memory footprint for a fixed number of elements, your `SortedLinkedSet` uses memory more flexibly, allocating only what's needed for each element.

Congratulations on your achievement, and may your future data structures be ever efficient and elegantly implemented!