Smarter Priority Queue (SPQ) ADT Implementation

Contents

1 Introduction

This document outlines the implementation of a Smarter Priority Queue (SPQ) Abstract Data Type (ADT) using both min- and max-heaps. The SPQ is more adaptable and flexible than a standard priority queue, featuring operations that allow dynamic modification and efficient management of entries.

2 Project Specifications

2.1 Heap Implementation

- The heap must be implemented from scratch using an array that is dynamically extendable.
- The same code must be used for both min- and max-heap constructions, ensuring flexibility through parameterization.

2.2 SPQ Operations

The SPQ supports the following operations:

- **toggle()**: Switches between min-heap and max-heap configurations.
- **removeTop()**: Removes and returns the entry object with the smallest or largest key based on the current state of the queue.
- **insert(k,v)**: Inserts a key-value pair into the priority queue and returns the corresponding entry.
- **top()**: Returns the top entry (min or max key) without removing it.
- **remove(e)**: Removes a specific entry object from the queue.
- **replaceKey(e, k)**: Replaces the key of an entry and returns the old key.
- **replaceValue(e, v)**: Replaces the value of an entry and returns the old value.
- **state()**: Returns the current state (Min or Max) of the queue.
- **isEmpty()**: Checks if the queue is empty.
- **size()**: Returns the current number of entries in the queue.

3 Pseudocode Implementation

The following pseudocode outlines the structure and methods for the SPQ ADT, which is implemented using a parameterized heap:

3.1 Entry Class Pseudocode

```
Class Entry<K extends Comparable<K>, V>
    // Attributes
    key: K  // Key used for comparisons
    value: V  // Value associated with the key
    current_index: Integer  // Index in the container structure

    // Constructor
    Constructor Entry(newK: K, newV: V):
        Initialize key with newK
        Initialize value with newV

    // Methods
    Method getKey() -> K:
        Return key
    Method getValue() -> V:
        Return value
    Method setKey(newK: K):
        Set key to newK
    Method setValue(newV: V):
        Set value to newV
    Method getIndex() -> Integer:
        Return current_index
    Method setIndex(newIndex: Integer):
        Set current_index to newIndex
    Method equals(obj: Object) -> Boolean:
        Return true if this and obj have same keys and values
    Method print() -> String:
        Return string representation of the entry
    Method compareTo(other: Entry<K, V>) -> Integer:
        Compare this entry's key with other's key
```

3.2 ExpandingArray Class Pseudocode

```
Class ExpandingArray<K extends Comparable<K>, V>
    // Attributes
    Array: List of Entry<K, V>  // Internal dynamic array
    size: Integer  // Current number of elements
    DefaultCapacity: Integer  // Default capacity of the array

    // Constructor
    Constructor ExpandingArray():
        Initialize with DefaultCapacity

    Constructor ExpandingArray(starting_size: Integer):
        Initialize Array with given size
        Set all elements to null
```

```
    // Methods
    Method size() -> Integer:
        Return size
    Method Capacity() -> Integer:
        Return the length of Array
    Method isEmpty() -> Boolean:
        Return true if size is 0
    Method clear():
        Set all elements in Array to null
        Set size to 0
    Method ensureCapacity():
        If array needs expansion:
            Double the size of Array
            Copy old elements to new array
    Method swapIndex(index1: Integer, index2: Integer):
        Swap elements at index1 and index2 in Array
    Method get(index: Integer) -> Entry<K, V>:
        Return the element at index if valid
    Method set(index: Integer, entry: Entry<K, V>):
        Assign entry to Array at index
    Method remove(index: Integer) -> Entry<K, V>:
        Store the element at index, then set it to null
        Decrement size and return stored element
```

3.3 PriorityQueueHeap Class Pseudocode

```
Class PriorityQueueHeap <K extends Comparable<K>, V>
    // Attributes
    heapType: HeapType  // Enum with values Min or Max
    expandingArray: ExpandingArray<Entry<K, V>>
    DefaultCapacity: int

    // Constructors
    Constructor PriorityQueueHeap():
        Call main constructor with Max heap type and default capacity

    Constructor PriorityQueueHeap(startingSize: int):
        Call main constructor with Max heap type and specified starting siz

    Constructor PriorityQueueHeap(heapType: HeapType):
        Call main constructor with specified heapType and default capacity

    Constructor PriorityQueueHeap(heapType: HeapType, startingSize: int):
        Initialize expandingArray with starting size
        Set currentHeapType to heapType

    // Methods
```

```
Method size() -> integer:
    Return the number of elements in expandingArray

Method isEmpty() -> boolean:
    Return true if expandingArray is empty

Method top() -> Entry<K, V>:
    If isEmpty() then Return null
    Else Return expandingArray[0]

Method toggle():
    If currentHeapType is HeapType.Max then Set to HeapType.Min
    Else Set to HeapType.Max
    Perform heapSortAndInsert()

Method insert(key: K, value: V) -> Entry<K, V>:
    Ensure capacity of expandingArray
    Add new Entry<K, V> at the end of expandingArray
    Perform upHeap starting from last element

Method removeTop() -> Entry<K, V>:
    If isEmpty() then Return null
    Swap first and last element in expandingArray
    Remove last element
    Perform downHeap starting from the first element
    Return the removed element

Method remove(entry: Entry<K, V>) -> Entry<K, V>:
    indexFound = linearSearch(entry)
    If indexFound is not -1 then
        Swap entry at indexFound with last element
        Remove last element
        Perform downHeap and upHeap as necessary
        Return the removed entry
    Else Return null

Method replaceKey(entry: Entry<K, V>, newKey: K) -> K:
    indexFound = linearSearch(entry)
    If indexFound is not -1 then
        oldKey = get key of entry at indexFound
        Set newKey at indexFound
        Adjust heap using downHeap and upHeap
        Return oldKey
    Else Return null

Method replaceValue(entry: Entry<K, V>, newValue: V) -> V:
    indexFound = linearSearch(entry)
    If indexFound is not -1 then
```

```
            oldValue = get value of entry at indexFound
            Set newValue at indexFound
            Return oldValue
        Else Return null

    Method state() -> String:
        Return currentHeapType as string
```

4 Big-O Time Complexities

- **toggle()**: O(1) for toggling behavior, but O(n log n) due to heap reordering.
- **remove(e)**: O(log n) for heap reordering after removal.
- **replaceKey(e, k)**: O(log n) for heap reordering, but could be O(n) in the worst case.
- **replaceValue(e, v)**: Expected O(1) for replacing the value, but worst-case O(n) if a linear search is needed.