Swinburne University of Technology

School of Science, Computing and Emerging Technologies

ASSIGNMENT COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures & Patterns
Assignment number and title:	3 – Amortized Analysis & Abstract Data Types
Due date:	Sunday, May 18, 2025, 23:59
Lecturer:	Dr. Markus Lumpe

Your name:____________________________ **Your student ID:**_______________

Marker's comments:

Problem	Marks	Obtained
1	34	
2	112	
Total	146	

Extension certification:

This assignment has been given an extension and is now due on ____________

Signature of Convener:_________________________

Problem Set 3: Amortized Analysis & Abstract Data Types

In this task, you define a priority queue, an important data type in software engineering. Priority queues represent a FIFO data structure where the addition of elements is governed by a sorting criterion: the priority of the added element. Queues are a linear data type allowing insertion at one end and deletion at the opposite end. Additionally, priority queues order elements upon insertion. The element with the highest priority is moved to the head of the queue. There are many known approaches to constructing priority queues. In this problem set, we use an array of pairs and denote the corresponding queue ends by two indices into the array. The approach taken here also guarantees the *order of arrival*. If two elements are assigned the same priority, they are arranged sequentially based on arrival. This requirement has implications for the sorting of arrays. We must employ a stable sorting technique, in which elements with the same priority remain ordered based on their arrival. Array sorting algorithms do not guarantee this behavior by default.

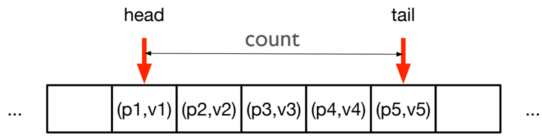

Figure 1: Priority Queue.

Figure 1 illustrates the principal features of a priority queue, as defined in the problem set. We use an array to store the elements of a priority queue. The pointers (or indices) *head* and *tail* refer to the current ends of the priority queue. Please note that the position of head and tail is not fixed. Their positions vary in response to performing queue operations. The priority queue is ordered from left to right. The distance between head and tail, called *count*, denotes the number of elements in the priority queue. It is zero when head and tail refer to the same position in the array. Otherwise, the value of *count* is greater than zero.

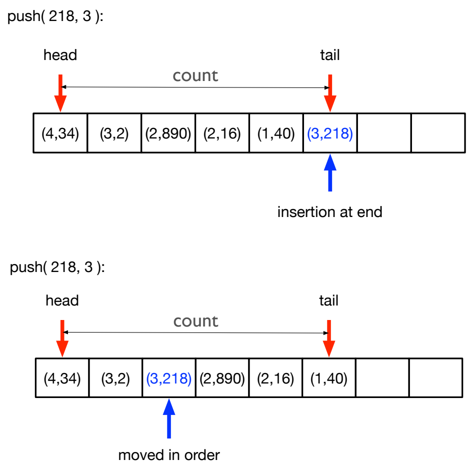

Figure 2: Add a new element to the priority queue.

Figure 2 depicts how we add elements with a given priority to the priority queue via the *push* operation. First, we insert the new element at the end. The new element becomes the tail of the priority queue. Next, we move the element into position. This is achieved by sorting the array. Please note that elements with the same priority are arranged in the order of arrival. As the priority queue already contains an element with priority 3, element 218 must be placed after element 2 in the priority queue. Technically, the comparison function for sorting must guarantee that (3,2) is ordered *before* (3,218).

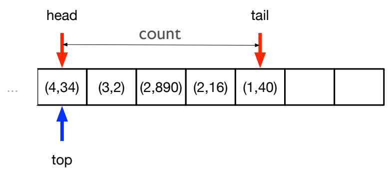

Figure 3: Top of priority queue.

Figure 3 shows what element is referred to when we employ the *top* operation. The *top* method returns the element at index head. Naturally, the method *top* can return an element only if *count* is greater than zero.

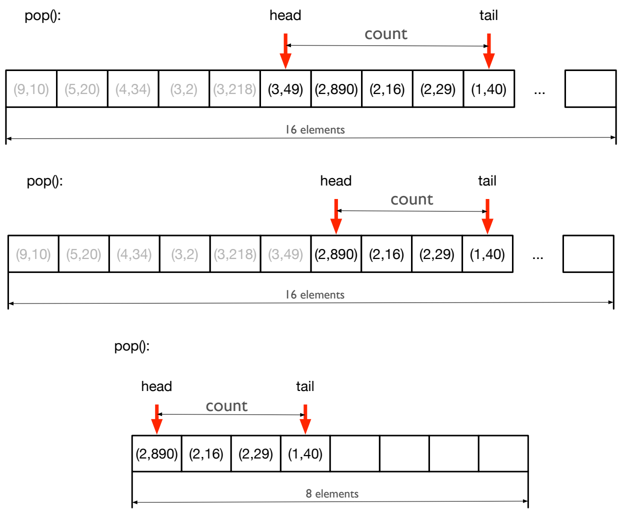

Figure 4: The pop operation triggering a resize.

Figure 4 illustrates the *pop* operation and a resize of the priority queue. A *pop* operation does not remove elements from the queue per se. It moves the *head* one position to the right (via increment). The inaccessible elements are grayed out. The operation *pop* may need to adjust the capacity of the priority queue. (The operation *push* has to ensure a sufficient capacity of the queue, not shown in Figure 2.) If the load factor for the priority queue reaches ¼, a resize is triggered. The size of the priority queue is halved. The elements are copied from the old array into the new array. The copy operation realigns the *head* and *tail* indices. After a resize, head points to the first element in the array. Resizing preserves the distance between head and tail. In Figure 4, the value is 4 before and after resizing.

The implementation requires two data types: `SortablePair`, an auxiliary data type for representing sortable pairs, and `PriorityQueue`, the data type for realizing a priority queue.

The `SortablePair` class defines the basic infrastructure for pairs. It has a `first()` and `second()` getter. In addition, it declares the output operator as a friend (we can assign a `SortablePair` object a textual representation via an output operator). `SortablePair` objects can be compared for equality and support the less-than operator that defines a weak ordering over `SortablePair` objects. The latter solely defines a relation over the first component.

The `PriorityQueue` class defines the basic infrastructure for a priority queue. A dynamic `SortablePair` array serves as the representation for `PriorityQueue`. The class `PriorityQueue` satisfies the properties of an abstract data type. Many of the implementation details for `PriorityQueue` mimic those used in the definition of `DynamicStack`, which we studied in Tutorial 8. There are, however, a few subtle differences that require special care.

Problem 1:

You must start with the `SortablePair` class. `SortablePair` is a template class with two template type arguments `K` and `V`, where `K` is the type of the first component (or key) and `V` is the type of the second component (or value). The template class for `SortablePair` is given below.

```
#pragma once

#include <ostream>

template<typename K, typename V>
class SortablePair
{
public:

  SortablePair( const K& aFirst = K{}, const V& aSecond = V{} ) noexcept;

  const K& first() const noexcept;
  const V& second() const noexcept;

  bool operator==( const SortablePair& aOther ) const noexcept;

  bool operator<( const SortablePair& aOther ) const noexcept;

  friend std::ostream& operator<<( std::ostream& aOStream,
                                   const SortablePair& aPair );

private:

  K fFirst;
  V fSecond;
};
```

The definition of template class `SortablePair` follows standard C++ class-building practice for templates. Class `SortablePair` defines key-value pairs. The textual representation for `SortablePair` objects is a comma-separated list enclosed in the symbols '`(`' and '`)`'.

Two `SortablePair` objects are equivalent if all components are pairwise equivalent.

A `SortablePair` object `obj1` is less than a `SortablePair` object `obj2`, `obj1 < obj2`, if `obj1`'s first component is greater than `obj2`'s first component. This semantics allows `SortablePair` objects to be arranged in decreasing order of priority while preserving the order of arrival.

The file `Main.cpp` contains a test function to check your implementation. Uncomment **`#define`** `P1` and compile your solution. Your program should produce the following output:

```
Test Sortable Pair:
[(4,34), (3,2), (2,890), (1,40), (2,16), (3,218), (5,20), (3,49), (9,10), (2,29)]
Test getter:
4 - 34
3 - 2
2 - 890
1 - 40
2 - 16
3 - 218
5 - 20
3 - 49
9 - 10
2 - 29
Test operator== :
0: Yes
1: No
2: No
```

```
3: No
4: No
5: No
6: No
7: No
8: No
9: No
Test operator<:
0:     No     Yes    Yes    Yes    Yes    Yes    No     Yes    No     Yes
1:     No     No     Yes    Yes    Yes    No     No     No     No     Yes
2:     No     No     No     Yes    No     No     No     No     No     No
3:     No     No     No     No     No     No     No     No     No     No
4:     No     No     No     Yes    No     No     No     No     No     No
5:     No     No     Yes    Yes    Yes    No     No     No     No     Yes
6:     Yes    Yes    Yes    Yes    Yes    Yes    No     Yes    No     Yes
7:     No     No     Yes    Yes    Yes    No     No     No     No     Yes
8:     Yes    Yes    Yes    Yes    Yes    Yes    Yes    Yes    No     Yes
9:     No     No     No     Yes    No     No     No     No     No     No
Test Sortable Pair completed.
```

Problem 2:

The template class `PriorityQueue` implements a dynamic priority queue.

```
#pragma once

#include "SortablePair.h"

#include <optional>
#include <cassert>

template<typename T, typename P>
class PriorityQueue
{
public:

  using value_type = SortablePair<P, T>;

  PriorityQueue() noexcept;

  ~PriorityQueue() noexcept;

  PriorityQueue( const PriorityQueue& ) = delete;
  PriorityQueue& operator=( const PriorityQueue& ) = delete;

  size_t count() const noexcept;
  size_t capacity() const noexcept;

  std::optional<T> top() const noexcept;
  void push( const T& aValue, const P& aPriority ) noexcept;
  void pop() noexcept;

private:

  value_type* fElements;
  size_t fHead;
  size_t fTail;
  size_t fCapacity;

  void sort() noexcept
  {
    std::sort( &fElements[fHead], &fElements[fTail] );
  }

  void resize( size_t aCapacity );

  void ensure_capacity();
  void adjust_capacity();
};
```

`PriorityQueue` is a template class with two template type arguments `T` and `P`, where `T` is the value type and `P` is the priority type. The definition of template class `PriorityQueue` follows standard C++ class-building practice for templates.

The `sort()` method is predefined. We use `std::sort()` to place new items in the correct position. The function `std::sort()` takes two iterators (or pointers) to denote the range of the array to be sorted. We use the indices `fHead` and `fTail` for this purpose and pass the addresses of the corresponding array elements to `std::sort()`. The function `std::sort()` uses the predicate less-than to arrange the elements and yields a decreasing order of priority while preserving the order of arrival for elements of type `SortablePair`.

Most features can be defined following the approach that we used for `DynamicStack`. There are two exceptions, though. In the `push()` method, we must insert a new `SortablePair` object into the elements array at the tail index, and a naïve implementation results in an unnecessary copy. Instead, we can use the in-place **new** operator (presented in the lecture *Abstract Data Types*) to reinitialize an object of type `SortablePair`. Use the following expression to insert a new `SortablePair` at the end of the priority queue:

```
new (&fElements[fTail++]) value_type( aPriority, aValue );
```

Here, the expression `(&fElements[fTail])` is the address of the tail element in the queue. We reinitialize it with `value_type( aPriority, aValue )`, the constructor for `SortablePair`. The name `value_type` is a nested member type name for the specialization `SortablePair<P, T>`.

The resize logic uses the member variable `fCapacity` as a value to determine the load factor of the priority queue. In the `resize()` method, remember that the head of the queue may not be at array index 0. When moving elements from the old array to the new one, you must offset the copy index by `fHead`. For instance, if the loop variable `i` equals 2 and `fHead` equals 6, the source index is 8 or `i + fHead` (i.e., `(2,29)` in Figure 4). This pair becomes the third element in the new array (cf. Figure 4).

The file `Main.cpp` contains a test function to check your implementation. Uncomment **`#define`** `P2` and compile your solution. Your program should produce the following output:

```
Test Priority Queue:
Queue capacity: 1, count: 0
Add (4,34), top: 34, capacity: 1, count: 1
Add (3,2), top: 34, capacity: 2, count: 2
Add (2,890), top: 34, capacity: 4, count: 3
Add (1,40), top: 34, capacity: 4, count: 4
Add (2,16), top: 34, capacity: 8, count: 5
Add (3,218), top: 34, capacity: 8, count: 6
Add (5,20), top: 20, capacity: 8, count: 7
Add (3,49), top: 20, capacity: 8, count: 8
Add (9,10), top: 10, capacity: 16, count: 9
Add (2,29), top: 10, capacity: 16, count: 10
Access all elements:
10, capacity: 16, count: 9
20, capacity: 16, count: 8
34, capacity: 16, count: 7
2, capacity: 16, count: 6
218, capacity: 16, count: 5
49, capacity: 8, count: 4
890, capacity: 8, count: 3
16, capacity: 4, count: 2
29, capacity: 2, count: 1
40, capacity: 1, count: 0
Test Priority Queue complete.
```

Submission deadline: Sunday, May 18, 2025, 23:59.
Submission procedure:

Follow the instructions on Canvas. Submit electronically the PDF of the printed source files `SortablePair.h` and `PriorityQueue.h`. Upload the source files `SortablePair.h` and `PriorityQueue.h` to Canvas.

The sources will be assessed and compiled in the presence of the solution artifacts provided on Canvas.