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: Dhanveer Ramnauth **Your student ID:** 103866373

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:_____

Figure 1: SortablePair.h

```cpp
#pragma once

#include <ostream>

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

    SortablePair(const K& aFirst = K{}, const V& aSecond = V{})
    noexcept
        : fFirst(aFirst), fSecond(aSecond) {}

    const K& first() const noexcept
    {
        return fFirst;
    }

    const V& second() const noexcept
    {
        return fSecond;
    }

    bool operator==(const SortablePair& aOther) const noexcept
    {
        return fFirst == aOther.fFirst && fSecond ==
    aOther.fSecond;
    }

    bool operator<(const SortablePair& aOther) const noexcept
    {
        return fFirst > aOther.fFirst;
    }

    friend std::ostream& operator<<(std::ostream& aOStream, const
    SortablePair& aPair)
    {
        aOStream << "(" << aPair.fFirst << "," << aPair.fSecond
    << ")";
        return aOStream;
```

```
37      }
38
39  private:
40      K fFirst;
41      V fSecond;
42  };
```

Figure 2: PriorityQueue.h

```cpp
#pragma once

#include "SortablePair.h"

#include <optional>
#include <cassert>
#include <algorithm>

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

    using value_type = SortablePair<P, T>;

    PriorityQueue() noexcept
        : fElements(new value_type[1]), fHead(0), fTail(0),
    ↪ fCapacity(1) {}

    ~PriorityQueue() noexcept
    {
        delete[] fElements;
    }

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

    size_t count() const noexcept
    {
        return fTail - fHead;
    }

    size_t capacity() const noexcept
    {
        return fCapacity;
    }

    std::optional<T> top() const noexcept
    {
        if (count() == 0) return std::nullopt;
```

```
40          return fElements[fHead].second();
41      }
42
43      void push(const T& aValue, const P& aPriority) noexcept
44      {
45          ensure_capacity();
46          new (&fElements[fTail++]) value_type(aPriority, aValue);
47          sort();
48      }
49
50      void pop() noexcept
51      {
52          if (count() > 0)
53          {
54              ++fHead;
55              adjust_capacity();
56          }
57      }
58
59  private:
60
61      value_type* fElements;
62      size_t fHead;
63      size_t fTail;
64      size_t fCapacity;
65
66      void sort() noexcept
67      {
68          std::sort(&fElements[fHead], &fElements[fTail]);
69      }
70
71      void resize(size_t aCapacity)
72      {
73          value_type* newArray = new value_type[aCapacity];
74          size_t n = count();
75          for (size_t i = 0; i < n; ++i)
76          {
77              newArray[i] = fElements[fHead + i];
78          }
79          delete[] fElements;
```

```
80          fElements = newArray;
81          fHead = 0;
82          fTail = n;
83          fCapacity = aCapacity;
84      }
85
86      void ensure_capacity()
87      {
88          if (fTail == fCapacity)
89          {
90              resize(fCapacity * 2);
91          }
92      }
93
94      void adjust_capacity()
95      {
96          if (count() <= fCapacity / 4 && fCapacity > 1)
97              resize(fCapacity / 2);
98      }
99  };
```