

Swinburne University of Technology*School of Science, Computing and Engineering Technologies***ASSIGNMENT COVER SHEET**

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: 4, A Tree-like Priority Queue
Due date: Friday, May 26, 2023, 23:59
Lecturer: Dr. Markus Lumpe

Your name: _____ **Your student id:** _____

Check Tutorial	Tues 08:30	Tues 10:30	Tues 12:30 BA603	Tues 12:30 ATC627	Tues 14:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30	Thurs 08:30	Thurs 10:30

Marker's comments:

Problem	Marks	Obtained
1	66	
Total	66	

Extension certification:

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

Signature of Convener: _____

```
// COS30008, Problem Set 4, 2023
```

```
#pragma once
```

```
#include <vector>
```

```
#include <optional>
```

```
#include <algorithm>
```

```
template<typename T, typename P>
```

```
class PriorityQueue
```

```
{
```

```
private:
```

```
    struct Pair
```

```
    {
```

```
        P priority;
```

```
        T payload;
```

```
        Pair( const P& aPriority, const T& aPayload ) :
```

```
            priority(aPriority),
```

```
            payload(aPayload)
```

```
        {}
```

```
    };
```

```
    std::vector<Pair> fHeap;
```

```
    /*
```

In the array representation, if we are starting to count indices from 0, the children of the i -th node are stored in the positions $(2 * i) + 1$ and $2 * (i + 1)$, while the parent of node i is at index $(i - 1) / 2$ (except for the root, which has no parent).

```
    */
```

```
void bubbleUp( size_t aIndex ) noexcept
```

```
{
```

```
    if ( aIndex > 0 )
```

```
    {
```

```
        Pair lCurrent = fHeap[aIndex];
```

```
    do
```

```
    {
```

```
        size_t lParentIndex = (aIndex - 1) / 2;
```

```
        if ( fHeap[lParentIndex].priority < lCurrent.priority )
```

```
        {
```

```
            fHeap[aIndex] = fHeap[lParentIndex];
```

```
            aIndex = lParentIndex;
```

```
        }
```

```
    else
```

```

        {
            break;
        }
    } while (aIndex > 0);

    fHeap[aIndex] = lCurrent;
}
}

void pushDown( size_t aIndex = 0 ) noexcept
{
    if ( fHeap.size() > 1 )
    {
        size_t lFirstLeafIndex = ((fHeap.size() - 2) / 2) + 1;

        if ( aIndex < lFirstLeafIndex )
        {
            Pair lCurrent = fHeap[aIndex];

            do
            {
                size_t lChildIndex = (2 * aIndex) + 1;
                size_t lRight = 2 * (aIndex + 1);

                if ( lRight < fHeap.size() && fHeap[lChildIndex].priority < fHeap[lRight].priority )
                {
                    lChildIndex = lRight;
                }

                if ( fHeap[lChildIndex].priority > lCurrent.priority )
                {
                    fHeap[aIndex] = fHeap[lChildIndex];
                    aIndex = lChildIndex;
                }
                else
                {
                    break;
                }

            } while ( aIndex < lFirstLeafIndex );

            fHeap[aIndex] = lCurrent;
        }
    }
}

```

public:

```

size_t size() const noexcept
{
    return fHeap.size();
}

std::optional<T> front() noexcept

```

```

{
    if (fHeap.size() >= 1)
    {
        Pair lastItem = fHeap.back();
        fHeap.pop_back();

        if (fHeap.empty())
        {
            return lastItem.payload;
        }
        else
        {
            std::swap(fHeap[0], lastItem);
            pushDown();
        }

        return lastItem.payload;
    }
    else
    {
        return std::optional<T>();
    }
}

void insert(const T& aPayload, const P& aPriority) noexcept
{
    Pair newpair(aPriority, aPayload);
    fHeap.emplace_back(newpair);
    bubbleUp(fHeap.size()-1);
}

void update(const T& aPayload, const P& aNewPriority) noexcept
{
    auto item = std::find_if(fHeap.begin(), fHeap.end(), [&aPayload](const Pair& pair) {
        return pair.payload == aPayload;
    });

    if (item == fHeap.end())
    {
        return;
    }
    else
    {
        size_t index = std::distance(fHeap.begin(), item);

        P oldPriority = fHeap[index].priority;

        fHeap[index].priority = aNewPriority;

        if (aNewPriority > oldPriority)
        {
            bubbleUp(index);
        }
    }
}

```

```
    }  
    else if(aNewPriority < oldPriority)  
    {  
        pushDown(index);  
    }  
  
}  
  
}  
  
};
```

```
// COS30008, Problem Set 4, 2023
```

```
#include <iostream>
```

```
#include <string>
```

```
#include "PriorityQueue.h"
```

```
int main()
```

```
{
```

```
    using PQ = PriorityQueue<std::string, size_t>;
```

```
    std::cout << "Test Priority Queue:" << std::endl;
```

```
    PQ lPriorityQueue;
```

```
    std::cout << "Elements in priority queue: " << lPriorityQueue.size() << std::endl;
```

```
    lPriorityQueue.insert( "be.", 2 );
```

```
    lPriorityQueue.insert( "or", 7 );
```

```
    lPriorityQueue.insert( "not", 5 );
```

```
    lPriorityQueue.insert( "to", 3 );
```

```
    lPriorityQueue.insert( "be", 7 );
```

```
    lPriorityQueue.insert( "To", 10 );
```

```
    std::cout << "Fetch " << lPriorityQueue.size() << " elements:" << std::endl;
```

```
    std::optional<std::string> lValue1 = lPriorityQueue.front();
```

```
    while ( lValue1 )
```

```
    {
```

```
        std::cout << lValue1.value() << std::endl;
```

```
        lValue1 = lPriorityQueue.front();
```

```
    }
```

```
    std::cout << "Elements in priority queue: " << lPriorityQueue.size() << std::endl;
```

```
    lPriorityQueue.insert( "to", 2 );
```

```
    lPriorityQueue.insert( "be", 7 );
```

```
    lPriorityQueue.insert( "or", 5 );
```

```
    lPriorityQueue.insert( "not", 3 );
```

```
    lPriorityQueue.insert( "To", 7 );
```

```
    lPriorityQueue.insert( "be.", 10 );
```

```
    lPriorityQueue.update( "be.", 1 );
```

```
    std::cout << "Fetch " << lPriorityQueue.size() << " elements:" << std::endl;
```

```
std::optional<std::string> lValue2 = lPriorityQueue.front();

while ( lValue2 )
{
    std::cout << lValue2.value() << std::endl;

    lValue2 = lPriorityQueue.front();
}

std::cout << "Elements in priority queue: " << lPriorityQueue.size() << std::endl;

std::cout << "Test Priority Queue complete." << std::endl;

return 0;
}
```