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:

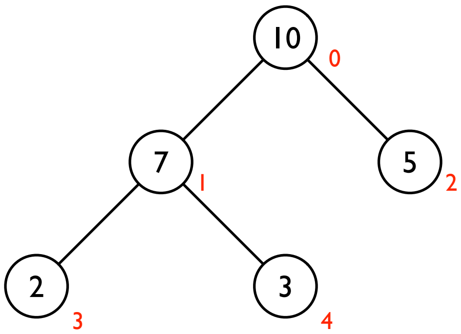
This assignment has been given an extension and is now due on __________________

Signature of Convener:_________________________

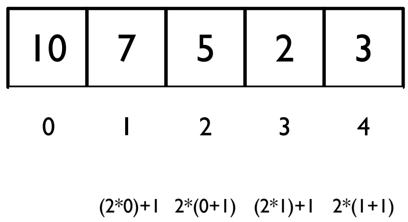
Problem Set 4: A Tree-like Priority Queue

The aim of this problem set is to define a basic priority queue data type. We have previously studied stacks and queues, which are linear data types using arrays as underlying representation. A priority queue can be defined in terms of a simply linear queue. However, it is more efficient to employ a tree structure to implement a priority queue and use an array to store the tree.

The solution is to "heapify" the tree. We assume a binary tree as underlying representation. In the array representation, if we are starting to count indices from 0, then 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). For example, the binary tree



may yield the following array layout:



Every node in the tree is mapped to a specific location in the array.

When using the array representation, we insert new elements in the priority queue at the end of the array and move them to the correct position along the parent chain if necessary. Similarly, when we remove the first element from the priority queue, we update the array by moving elements towards the leaves if necessary. This guarantees that elements in the array are always ordered according to their specified priority.

Somebody else has already started the implementation. We define a priority queue as a class template:

```
#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;

  void bubbleUp( size_t aIndex ) noexcept;
  void pushDown( size_t aIndex = 0 ) noexcept;

public:

  size_t size() const noexcept;                                    // (2)

  std::optional<T> front() noexcept;                               // (28)
  void insert( const T& aPayload, const P& aPriority ) noexcept;    // (10)
  void update( const T& aPayload, const P& aNewPriority ) noexcept; // (26)
};
```

Class template `PriorityQueue` defines an object adapter for `std::vector`. Standard library vectors define dynamic arrays whose elements can be accessed and manipulated using indexers and iterators.

The class template `PriorityQueue` is parameterized over the payload type `T` and priority type `P`. Internally, we rely of `Pair`, a public class (or structure) that associates a priority with a payload, and two private service functions, `bubbleUp()` and `pushdown()`, that implement the necessary infrastructure to move elements according to their priority into place.

The public interface of `PriorityQueue` defines four methods:

- `size()`:
  This method returns the number of items in the priority queue.
- `front()`:
  The method `front()` extracts the root item and returns it to the caller. The return value is wrapped into `std::optional`, which allows to return no-value when the priority queue is empty. Optional is a C++-17 feature.
  To extract the root item, we first remove the last item from `fHeap`. This requires two operations: first access the back element of `fHeap`, next erase the last element using

a vector iterator positioned at the last element (i.e., `fHeap.begin()` + offset to last element).
If the removed item is the last item in the priority queue, return its payload to the caller.
If there are still more items in `fHeap`, then exchange the first item in `fHeap` with the removed item and call `pushDown()`. The last item likely had a low priority and now sits at the root, so we need to move it towards the leaves, until both its children have a lower priority than it. Finally, return the payload of the extracted item to the caller. (It is actually the payload of the root item in this instance.)

- `insert()`:
  This method adds a new item into the priority queue using the given priority value. Insert first emplaces a new item at the back of `fHeap`. Next, it calls `bubbleUp()` to move to newly created last element into place along the parent chain.
- `update()`:
  We can use method `update()` to change the priority of a given item in the priority queue. First, we have to determine the array index of the item (i.e., `aPayload`). If no such item exists in the priority queue, `update()` finishes. Otherwise, we have to change the priority of the item to `aNewPriority`. This requires saving the old priority in a local variable. Once the new priority has been set, the item may have to be moved to a different position in the priority queue. If the new priority in greater than the old one, then we use `bubbleUp()` to move the item along the parent chain. If the new priority is less than the old one, then we use `pushDown()` to move the item towards the leaves. As argument to both method calls, we use the current array index of the item.

The implementations of `bubbleUp()` and `pushDown()` are given:

- `bubbleUp()`:

```
void bubbleUp( size_t aIndex ) noexcept
{
    if ( aIndex > 0 )
    {
        // set lCurrent to item to be moved into position
        Pair lCurrent = fHeap[aIndex];

        do
        {
            size_t lParentIndex = (aIndex - 1) / 2;

            // move parent item down if necessary, or stop
            if ( fHeap[lParentIndex].priority < lCurrent.priority )
            {
                fHeap[aIndex] = fHeap[lParentIndex];
                aIndex = lParentIndex;
            }
            else
            {
                break;
            }
        } while (aIndex > 0);

        // move item into position
        fHeap[aIndex] = lCurrent;
    }
}
```

- pushDown():

```
void pushDown( size_t aIndex = 0 ) noexcept
{
    if ( fHeap.size() > 1 )
    {
        // limit search to first leaf
        size_t lFirstLeafIndex = ((fHeap.size() - 2) / 2) + 1;

        if ( aIndex < lFirstLeafIndex )
        {
            // set lCurrent to item to be moved into position
            Pair lCurrent = fHeap[aIndex];

            do
            {
                // indices of children
                size_t lChildIndex = (2 * aIndex) + 1;
                size_t lRight = 2 * (aIndex + 1);

                // pick child node with highest priority
                if ( fHeap[lChildIndex].priority <
                                            fHeap[lRight].priority )
                {
                    lChildIndex = lRight;
                }

                // move parent item down if necessary, or stop
                if ( fHeap[lChildIndex].priority >
                                            lCurrent.priority )
                {
                    fHeap[aIndex] = fHeap[lChildIndex];
                    aIndex = lChildIndex;
                }
                else
                {
                    break;
                }

            } while ( aIndex < lFirstLeafIndex );

            // move item into position
            fHeap[aIndex] = lCurrent;
        }
    }
}
```

Start with the header file provided on Canvas and implement the public methods.

The test driver should produce the following output:

```
Test Priority Queue:
Fetch 6 elements:
To
be
or
not
to
be.
Elements in priority queue: 0
Fetch 6 elements:
To
```

```
be
or
not
to
be.
Elements in priority queue: 0
Test Priority Queue complete.
```

Submission deadline: Friday, May 26, 2023, 23:59.

Submission procedure: PDF of printed code for `PriorityQueue.h` and the source of `PriorityQueue.h`.