COS30008 Semester 1, 2023 Dr. Markus Lumpe

# 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: Md Redwan Ahmed Zawad Your student id:103501849

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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:

1

// 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(); //return size of the fHeap

}

std::optional<T> front() noexcept

{

if (fHeap.empty()) { //Check if fHeap is empty

return std::optional<T>(); //return no value wrapped in optional

}

Pair lEaf = fHeap.back(); /\*get the back element of fHeap\*/

fHeap.erase(fHeap.end()-1); //erase the last element

if (!fHeap.empty())

{

std::swap(fHeap[0], lEaf); /\*exchange the first item with

the last item and call pushdown \*/

pushDown();

}

return std::optional<T>(lEaf.payload);

}

void insert(const T& aPayload, const P& aPriority) noexcept

{

fHeap.push\_back(Pair(aPriority,aPayload)); //add new element given priority value

bubbleUp(fHeap.size() - 1); // move to new last element

}

void update(const T& aPayload, const P& aNewPriority) noexcept

{

auto fPchk = [&aPayload](const Pair& pair) // lambda function to compare aPayload with the payload of each pair in fHeap

{

return pair.payload == aPayload; //returns bool

};

auto fupit = std::find\_if(fHeap.begin(), fHeap.end(), fPchk); //iterates over the fHeap till fPchk returns true and returns that element to fupit

if (fupit != fHeap.end()) //Checks that a match was found

{

P lprevious = fupit->priority; /\*saving the oldpriority and

and setting new priority\*/

fupit->priority = aNewPriority;

size\_t index = std::distance(fHeap.begin(), fupit); //get the index

if (aNewPriority > lprevious) //checks if new priority is greater than previous one

{

bubbleUp(index);

}

else

{

pushDown(index);

}

}

}

};