Priority Queues

1. What is a priority queue?

Queues are commonly used in scheduling where tasks follow the first-in-first-out (FIFO) order. There are several situations that such an order must be overruled using some priority criteria. For example, at flight check-in counters, elderly and pregnant customers may have priority over others; on roads with toll booths, some vehicles may be put through immediately, even without paying (police cars, ambulances, fire engines, etc.). For a sequence of processes, process P_2 may need to be executed before process P_1 for the proper functioning of a system, even though P_1 went to the waiting list before P_2. In situations like those, a *priority queue* becomes a better choice than standard queue.

Priority queue (PQ) is an extension of queue in which every item associates with a priority value. An element with high priority is dequeued before an element with low priority. Elements of the same priority are either sorted following the order in which they were enqueued or left unsorted. A **priority value** indicates, for example, a patient's priority for treatment or a task's priority for completion. The quantity for this value can be arbitrary chosen as long as it defines a simple ranking.

Formally, a PQ is an ADT that contains a finite number of objects, not necessarily distinct, having the same data type and ordered by priority. This ADT supports the following basic operations [1]:

- Test whether a PQ is empty

`isEmpty()`	Task: Sees whether this PQ is empty. Input: None. Output: True if the PQ is empty; otherwise false.

- Add a new entry to the PQ in its sorted position based on priority value.

`enqueue(newEntry)`	Task: Adds newEntry to this PQ. Input: newEntry. Output: True if the operation is successful; otherwise false

- Remove from the PQ the entry with the highest priority value.

`dequeue()`	Task: Removes the entry with the highest priority from this PQ. Input: None. Output: True if the operation is successful; otherwise false.

- Get the entry in the PQ with the highest priority value.

`peek()`	Task: Returns the entry in this PQ with the highest priority. The operation does not change the priority queue. Input: None. Output: The entry with the highest priority.

2. Applications of priority queues

- **In a hospital's emergency room (ER) [1].** Patients waiting for treatment are normally put in a queue according to the order of their arrival. However, it is unreasonable if Ms. Zither, who was just rushed to the ER with acute appendicitis, would have to wait for Mr. Able to have a splinter removed. Clearly, the ER staff should assign some measure of urgency, or priority, to the patients. The next available doctor should treat the patient with the highest priority.
- **Discrete event simulation or scheduling.** The events are added to a queue with their simulation time used as the priority. The execution of the simulation proceeds by repeatedly pulling the top of the queue and executing the event thereon.
  - Google Mail helps users prioritize their emails by automatically suggesting frequently read emails as important emails or letting users choose which type of emails to appear first (unread emails, starred emails, or everything).
  - Students makes schedules for completing their assignments according to the assignments' difficulty and submission deadlines.
  - An onscreen messaging system has three kinds of messages, each of which has different priority. Alerts always get displayed immediately including removing whatever was on screen. Instructions could be queued up and only be displayed once the ones in front of it has been dismissed. Reminders only get displayed after all the instructions were gone and they too get queued up in the order in which they came in. (from Quora)
- **Bandwidth management [3].** Priority queue can be used to manage limited resources such as bandwidth on a transmission line from a network router. In the event of outgoing traffic queuing due to insufficient bandwidth, all other queues can be halted to send the traffic from the highest priority queue upon arrival. This ensures that the prioritized traffic (such as real-time traffic, e.g. an RTP stream of a VoIP connection) is forwarded with the least delay and the least likelihood of being rejected due to a queue reaching its maximum capacity. Many modern protocols for local area networks also include the concept of priority queues at the media access control (MAC) sub-layer to ensure that high-priority applications (such as VoIP or IPTV) experience lower latency than other applications which can be served with best effort service. Examples include IEEE 802.11e and ITU-T G.hn. Usually a limitation (policer) is set to limit the bandwidth that traffic from the highest priority queue can take to prevent high priority packets from choking off all other traffic.
- **Dijkstra's algorithm and Best-first search algorithms.** Those algorithms find the shortest path between two vertices of a weighted graph by trying out the most promising routes first. A PQ (aka the fringe) is used to keep track of unexplored routes; the one for which the estimate (a lower bound in the case of A*) or the calculation (in Dijkstra) of the total path length is smallest is given highest priority.
- **Prim's algorithm for minimum spanning tree.** The weight of the edges is used to decide the priority of the vertices. Lower the weight, higher the priority and higher the weight, lower the priority.
- **Huffman coding.** Huffman coding requires one to repeatedly obtain the two lowest-frequency trees. A priority queue is one method of doing this.

3. Implementations in C++

There are different ways to define a priority queue and all these approaches target to the efficiency of enqueuing and dequeuing. Since elements may arrive randomly to the queue, there is no guarantee that the front elements will be the most like to be dequeued as that the elements put at the end will be the last candidates for dequeuing. In addition, the definition of priority value may also be tricky due to a wide spectrum of possible priority criteria that can be used in difference cases, such as levels of customer satisfaction (higher values means higher priority), time of scheduled execution (higher values, on the other hand, indicates lower priority), status (an elderly or handicapped person is privileged) and others.

3.1 Linked-list-based Priority Queues

Priority queues can be represented by two variations of linked lists. The first variation simply arranges all elements according to their order of entry, while in another, the order is maintained by putting a new element in its proper position according to its priority. The total operational times are *O(n)* in both cases because, for an unordered list, adding an element is *O(1)* but searching is *O(n)*, and in a sorted list, taking an element is *O(1)* but adding an element is *O(n)*. More efficient queue representations are listed in [2], in which the implementation of J. O. Hendriksen (1977, 1983) operates consistently well with queues of any size and takes $O(\sqrt{n})$.

The following example demonstrate a sequence of operations on a Linked-list-based priority queue. Each element is an uppercase character. *Larger the ASCII value, higher the priority*. Elements are sorted in descending order of their priority.

operation	argument	return value	size	contents: head (left) to tail (right)
enqueue	P		1	P
enqueue	Q		2	Q → P
enqueue	E		3	Q → P → E
dequeue		Q	2	P → E
enqueue	X		3	X → P → E
enqueue	A		4	X → P → E → A
enqueue	M		5	X → P → M → E → A
dequeue		X	4	P → M → E → A

The class SLPriorityQueue defines a PQ by using linked sorted list as underlying data structure. Elements of the PQ are arranged in descending order of priority, from head to tail of the linked list. In this way, the operations of a PQ correspond to those of the associated linked list as follows:

- Test whether a PQ is empty = Test whether the linked list contains no node.
- Add a new entry to the PQ in its sorted position based on priority value = Insert the new entry into the linked list such that the descending order of priority from head to tail is not violated, which takes *O(n)*.
- Remove from the PQ the entry with the highest priority value = Remove the first node (head) of the linked list, which takes *O(1)*.
- Get the entry in the PQ with the highest priority value: Get the content of the head of the linked list, which takes *O(1)*.

The below source codes are modified from those supported by [1]. Key functions are highlighted in yellow. Please find in the *.h and *.cpp files associated with this document for further details.

SLPriorityQueue.h

```
#pragma once
#ifndef _SL_PRIORITY_QUEUE
#define _SL_PRIORITY_QUEUE
#include "LinkedSortedList.h"
class SLPriorityQueue
{
private:
	LinkedSortedList* slistPtr;	// Pointer to sorted list of integers

public:
	SLPriorityQueue();
	~SLPriorityQueue();

	bool isEmpty() const;
	bool enqueue(const int& newEntry);
	bool dequeue();

	/** @throw PrecondViolatedExcep if priority queue is empty. */
	int peek() const throw(PrecondViolatedExcep);
}; // end SLPriorityQueue
#endif
```

SLPriorityQueue.cpp

```
#include "SLPriorityQueue.h"
SLPriorityQueue::SLPriorityQueue()
{
	slistPtr = new LinkedSortedList();
}  // end default constructor

SLPriorityQueue::~SLPriorityQueue()
{
}  // end destructor

bool SLPriorityQueue::isEmpty() const
{
	return slistPtr->isEmpty();
}  // end isEmpty

bool SLPriorityQueue::enqueue(const int& newEntry)
{
	slistPtr->insertSorted(newEntry);
	return true;
}  // end add

bool SLPriorityQueue::dequeue()
{
	// The highest priority item is at the beginning of the sorted list
	return slistPtr->remove(1);
}  // end remove

int SLPriorityQueue::peek() const throw(PrecondViolatedExcep)
{
	if (isEmpty())
		throw PrecondViolatedExcep("peekFront() called with empty queue.");

	// Priority queue is not empty; return highest priority item;
	// it is at the beginning of the sorted list
	return slistPtr->getEntry(1);
}  // end peek
```

LinkedSortedList.cpp

```
#include "LinkedSortedList.h"  // Header file
void LinkedSortedList::insertSorted(const int& newEntry)
{
	Node* newNodePtr = new Node(newEntry);
	Node* prevPtr = getNodeBefore(newEntry);

	if (isEmpty() || (prevPtr == nullptr)) // Add at beginning
	{
		newNodePtr->setNext(headPtr);
		headPtr = newNodePtr;
	}
	else // Add after node before
	{
		Node* aftPtr = prevPtr->getNext();
		newNodePtr->setNext(aftPtr);
		prevPtr->setNext(newNodePtr);
	} // end if

	itemCount++;
} // end insertSorted

bool LinkedSortedList::remove(int position)
{
	bool ableToDelete = (position >= 1) && (position <= itemCount);
	if (ableToDelete)
	{
		Node* curPtr = nullptr;
		if (position == 1){
			// Delete the first node in the chain
			curPtr = headPtr; // save pointer to node
			headPtr = headPtr->getNext();
		}
		else{
			// Find node that is before the one to delete
			Node* prevPtr = getNodeAt(position - 1);

			// Point to node to delete
			curPtr = prevPtr->getNext();

			// Disconnect indicated node from chain by connecting the
			// prior node with the one after
			prevPtr->setNext(curPtr->getNext());
		}  // end if

		   // Return deleted node to system
		curPtr->setNext(nullptr);
		delete curPtr;
		curPtr = nullptr;

		itemCount--;  // Decrease count of entries
	}  // end if

	return ableToDelete;
}  // end remove

Node* LinkedSortedList::getNodeBefore(const int& anEntry) const
{
	Node* curPtr = headPtr;
	Node* prevPtr = nullptr;

	while ((curPtr != nullptr) && (curPtr->getItem() > anEntry)){
		prevPtr = curPtr;
		curPtr = curPtr->getNext();
	} // end while

	return prevPtr;
} // end getNodeBefore
```

3.2 Heap-based Priority Queues

Defining a priority queue using heap results in a more time-efficient implementation. It is straightforward because PQ operations are exactly analogous to heap operations and the priority value of an item in a PQ corresponds to the value of an item in a heap. It is also worth noting that while priority queues are often implemented with heaps, they are conceptually distinct from heaps.

Let's redo the example in Section 3.1 with a (binary) Heap-based priority queue. Again, each element is an uppercase character; *larger the ASCII value, higher the priority*.

operation	argument	return value	size	contents (max element appears on the left)
enqueue	P		1	P
enqueue	Q		2	Q – P
enqueue	E		3	Q – P – E
dequeue		Q	2	P → E
enqueue	X		3	X → E → P
enqueue	A		4	X → E → P → A
enqueue	M		5	X → M → P → A → E
dequeue		X	4	P → M → E → A

The class HeapPriorityQueue defines a PQ by using an instance of array-based max (binary) heap as underlying data structures. Because heap is considered as a complete binary tree in this case, a simple array-based implementation of the heap is sufficient if the maximum number of items is known in advance. In general, the operations of a PQ correspond to those of the associated linked list as follows:

- Test whether a PQ is empty = Test whether the heap contains no element.
- Add a new entry to the PQ in its sorted position based on priority value = Insert the new entry into the heap, which takes *O(log n)*.
- Remove from the PQ the entry with the highest priority value = Remove the item in the root of this heap, which takes *O(log n)*.
- Get the entry in the PQ with the highest priority value: Get the data that is in the root (top) of this heap, which takes *O(1)*.

HeapPriorityQueue.h

```
#pragma once
#ifndef _HEAP_PRIORITY_QUEUE
#define _HEAP_PRIORITY_QUEUE
#include "ArrayMaxHeap.h"
class HeapPriorityQueue
{
private:
	ArrayMaxHeap* heapPtr;   // Pointer to heap of integers in the PQ
public:
	HeapPriorityQueue();
	~HeapPriorityQueue();

	bool isEmpty() const;
	bool enqueue(const int& newEntry);
	bool dequeue();

	/** @pre The priority queue is not empty. */
	int peek() const throw(PrecondViolatedExcep);
}; // end HeapPriorityQueue
#endif
```

HeapPriorityQueue.cpp

```
#include "HeapPriorityQueue.h"
bool HeapPriorityQueue::isEmpty() const
{
   return heapPtr->isEmpty();
}  // end isEmpty

bool HeapPriorityQueue::enqueue(const int& newEntry)
{
   return heapPtr->add(newEntry);
}  // end add

bool HeapPriorityQueue::dequeue()
{
   return heapPtr->remove();
}  // end remove

int HeapPriorityQueue::peek() const throw(PrecondViolatedExcep)
{
   try
   {
      return heapPtr->peekTop();
   }
   catch (PrecondViolatedExcep e)
   {
      throw PrecondViolatedExcep("Attempted peek into an empty PQ.");
   }  // end try/catch
}  // end peek
```

ArrayMaxHeap.cpp

```
#include "ArrayMaxHeap.h"
bool ArrayMaxHeap::add(const int& newData)
{
   bool isSuccessful = false;
   if (itemCount < maxItems)
   {
      items[itemCount] = newData;

      bool inPlace = false;
      int newDataIndex = itemCount;
      while ((newDataIndex > 0) && !inPlace)
      {
         int parentIndex = getParentIndex(newDataIndex);
         if (items[newDataIndex] < items[parentIndex])
         {
            inPlace = true;
         }
         else
         {
            swap(items[newDataIndex], items[parentIndex]);
            newDataIndex = parentIndex;
         }  // end if
      }  // end while

      itemCount++;
      isSuccessful = true;
   }  // end if

   return isSuccessful;
}  // end add

bool ArrayMaxHeap::remove()
{
   bool isSuccessful = false;
   if (!isEmpty())
```

```
    {
        items[ROOT_INDEX] = items[itemCount - 1];
        itemCount--;
        heapRebuild(ROOT_INDEX);
        isSuccessful = true;
    }  // end if

    return isSuccessful;
}  // end remove
```

4. Examples with SLPriorityQueue

This example shows how to use SLPriorityQueue with integers. You can also use this PQ with another data type by replacing int with some data type you want.

main.cpp

```
#include <iostream>
#include <vector>
#include "SLPriorityQueue.h" // ADT Priority Queue operations
using namespace std;
int main()
{
	SLPriorityQueue* pqPtr = new SLPriorityQueue();
	cout << "Empty: " << pqPtr->isEmpty() << endl;

	bool success;
	// Insert 80 and peek the max element
	success = pqPtr->enqueue(80);
	if (!success)
		cout << "Failed to add to the priority queue." << endl;
	cout << "Peek: " << pqPtr->peek() << endl;

	// Insert 81 and peek the max element
	success = pqPtr->enqueue(81);
	if (!success)
		cout << "Failed to add to the priority queue." << endl;
	cout << "Peek: " << pqPtr->peek() << endl;

	// Insert 69 and peek the max element
	success = pqPtr->enqueue(69);
	if (!success)
		cout << "Failed to add to the priority queue." << endl;
	cout << "Peek: " << pqPtr->peek() << endl;

	// Remove max and peek the max element
	cout << "Remove: " << pqPtr->dequeue() << endl;
	cout << "Peek: " << pqPtr->peek() << endl;

	// Insert 88 and peek the max element
	success = pqPtr->enqueue(88);
	if (!success)
		cout << "Failed to add to the priority queue." << endl;
	cout << "Peek: " << pqPtr->peek() << endl;

	// Insert 65 and peek the max element
	success = pqPtr->enqueue(65);
	if (!success)
		cout << "Failed to add to the priority queue." << endl;
	cout << "Peek: " << pqPtr->peek() << endl;

	// Insert 77 and peek the max element
	success = pqPtr->enqueue(77);
	if (!success)
		cout << "Failed to add to the priority queue." << endl;
	cout << "Peek: " << pqPtr->peek() << endl;

	// Remove max and peek the max element
	cout << "Remove: " << pqPtr->dequeue() << endl;
	cout << "Peek: " << pqPtr->peek() << endl << endl;

	// Remove all items
```

```
        while (!pqPtr->isEmpty())
        {
                try
                {
                        cout << "Peek: " << pqPtr->peek() << endl;
                }
                catch (PrecondViolatedExcep e)
                {
                        cout << e.what() << endl;
                }  // end try/catch
                cout << "Remove: " << pqPtr->dequeue() << endl;
        }  // end for

           // Check possible exceptions
        cout << "remove with an empty priority queue: " << endl;
        cout << "Empty: " << pqPtr->isEmpty() << endl;
        cout << "remove: " << pqPtr->dequeue() << endl; // nothing to remove!
        try
        {
                cout << "peek with an empty priority queue: " << endl;
                cout << "peek: " << pqPtr->peek() << endl; // nothing to see!
        }
        catch (PrecondViolatedExcep e)
        {
                cout << e.what();
        }  // end try/catch

        getchar();
        return 0;
}  // end main
```

```
Empty: 1
Peek: 80
Peek: 81
Peek: 81
Remove: 1
Peek: 80
Peek: 88
Peek: 88
Peek: 88
Remove: 1
Peek: 80

Peek: 80
Remove: 1
Peek: 77
Remove: 1
Peek: 69
Remove: 1
Peek: 65
Remove: 1
remove with an empty priority queue:
Empty: 1
remove: 0
peek with an empty priority queue:
Precondition Violated Exception: peekFront() called with empty queue.
```

5. References

[1] Prichard, J., & Carrano, F. (2010). Data Abstraction and Problem Solving with Java: Walls and Mirrors. Addison-Wesley Publishing Company.

[2] Drozdek, Adam. Data Structures and algorithms in C++. Cengage Learning, 2012.

[3] Wikipedia: https://en.wikipedia.org/wiki/Priority_queue