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 choses 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 [3].** 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. Prioritizing courses' assignments according their deadlines could be a nice example.
- **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 [3].** Search algorithms in graph theory and Artificial Intelligence find the shortest path between two vertices of a weighted graph by trying out the most promising routes first. A PQ (also known as 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 the case of Dijkstra) of the total path length is smallest is given highest priority.
- **Prim's algorithm for minimum spanning tree [3].** 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 [3].** 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 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 source code is written in C++ template to serve various data types. Key functions (highlighted in yellow) will be further explained while other functions can be found in the *.h and *.cpp files associated with this document.

SLPriorityQueue.h

```
/** This source code is a modified version of which provided in the Data
Structures and Problem Solving 6th edition book **/
#pragma once
#ifndef _SL_PRIORITY_QUEUE
#define _SL_PRIORITY_QUEUE
#include "LinkedSortedList.cpp"
template<class ItemType>
class SL_PriorityQueue
{
private:
	LinkedSortedList<ItemType>* slistPtr;  // Pointer to sorted list of items

public:
	/** constructors and destructors */

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

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

SLPriorityQueue.cpp

```
#include "SLPriorityQueue.h"
/** There are some codes above **/
template<class ItemType>
bool SL_PriorityQueue<ItemType>::isEmpty() const
{
	return slistPtr->isEmpty();
}  // end isEmpty

template<class ItemType>
bool SL_PriorityQueue<ItemType>::enqueue(const ItemType& newEntry)
{
	slistPtr->insertSorted(newEntry);
	return true;
}  // end add

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

template<class ItemType>
ItemType SL_PriorityQueue<ItemType>::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
/** There are some codes above **/
void LinkedSortedList<ItemType>::insertSorted(const ItemType& newEntry)
{
	Node<ItemType>* newNodePtr = new Node<ItemType>(newEntry);
	Node<ItemType>* prevPtr = getNodeBefore(newEntry);

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

	itemCount++;
} // end insertSorted

template<class ItemType>
bool LinkedSortedList<ItemType>::remove(int position)
{
	bool ableToDelete = (position >= 1) && (position <= itemCount);
	if (ableToDelete)
	{
		Node<ItemType>* 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<ItemType>* 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

template<class ItemType>
Node<ItemType>* LinkedSortedList<ItemType>::getNodeBefore(const ItemType&
anEntry) const
{
      Node<ItemType>* curPtr = headPtr;
      Node<ItemType>* prevPtr = nullptr;

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

      return prevPtr;
} // end getNodeBefore

/** There are some codes below**/
```

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.

The class HeapPriorityQueue defines a PQ by using an instance of array-based max 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

```
/** This source code is a modified version of which provided in the Data
Structures and Problem Solving 6th edition book **/
#pragma once
#ifndef _HEAP_PRIORITY_QUEUE
#define _HEAP_PRIORITY_QUEUE
#include "ArrayMaxHeap.cpp"

template<class ItemType>
class HeapPriorityQueue
{
private:
	ArrayMaxHeap<ItemType>* heapPtr;   // Pointer to heap of items in the PQ

public:
	/** constructors and destructors */

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

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

HeapPriorityQueue.cpp

```
#include "HeapPriorityQueue.h"
/** There are some codes above **/
bool HeapPriorityQueue<ItemType>::isEmpty() const
{
   return heapPtr->isEmpty();
}  // end isEmpty

template<class ItemType>
bool HeapPriorityQueue<ItemType>::enqueue(const ItemType& newEntry)
{
   return heapPtr->add(newEntry);
}  // end add

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

template<class ItemType>
ItemType HeapPriorityQueue<ItemType>::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

```cpp
#include "ArrayMaxHeap.h"
/** There are some codes above **/

template<class ItemType>
bool ArrayMaxHeap<ItemType>::add(const ItemType& 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

template<class ItemType>
bool ArrayMaxHeap<ItemType>::remove()
{
   bool isSuccessful = false;
   if (!isEmpty())
   {
      items[ROOT_INDEX] = items[itemCount - 1];
      itemCount--;
      heapRebuild(ROOT_INDEX);
      isSuccessful = true;
   }  // end if

   return isSuccessful;
}  // end remove

/** There are some codes below **/
```

4. Examples with SLPriorityQueue

This example demonstrates how to use the class SLPriorityQueue with strings. Strings that come earlier according to the lexicographical order has lower priority than those come afterward.

main.cpp

```
/** This source code is a modified version of which provided in the Data Structures and Problem
Solving 6th edition book **/
#include <iostream>
#include <string>
#include <vector>
#include "SLPriorityQueue.cpp" // ADT Priority Queue operations
using namespace std;
int main()
{
	// This is the data to be inserted
	vector<string> items = { "kiwi", "apple", "banana", "mango", "watermelon",
"coconut" };

	// Insert every element to the PQ
	SLPriorityQueue<string>* pqPtr = new SLPriorityQueue<string>();
	cout << "Empty: " << pqPtr->isEmpty() << endl;
	for (int i = 0; i < items.size(); i++)
	{
		cout << "Adding " << items[i] << endl;
		bool success = pqPtr->enqueue(items[i]);
		if (!success)
			cout << "Failed to add " << items[i] << " to the PQ." << endl;
	}  // end for

	// Iteratively get the most prioritized element from PQ
	cout << "Empty?: " << pqPtr->isEmpty() << endl;
	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
Adding kiwi
Adding apple
Adding banana
Adding mango
Adding watermelon
Adding coconut
Empty?: 0
Peek: watermelon
Remove: 1
Peek: mango
Remove: 1
Peek: kiwi
Remove: 1
Peek: coconut
Remove: 1
Peek: banana
Remove: 1
Peek: apple
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.
```

Another example shows that SLPriorityQueue also works with user-defined data types. Consider a PQ whose elements are of the Fruit data type and they are prioritized based on their degrees of sourness. Ties are solved by lexicographical order of fruits' names.

Fruit.h

```
#pragma once
#ifndef _FRUIT
#define _FRUIT
#include <string>
using namespace std;
class Fruit
{
private:
	string fruit_name;
	int degree_of_sourness;
public:
	/** Constructors and member functions */

	// Override the relational operator >
	friend bool operator >(Fruit const& lhs, Fruit const& rhs)
	{
		if (lhs.degree_of_sourness > rhs.degree_of_sourness)
			return true;
		else
		{
			if (lhs.degree_of_sourness == rhs.degree_of_sourness)
				return (lhs.fruit_name < rhs.fruit_name);
			return false;
		}
	}
	friend bool operator >(Fruit const& lhs, Fruit const& rhs)
	{
		// ... ...
	}
};
#endif
```

(Another) main.cpp

```
int main()
{
	// This is the data to be inserted
	vector<Fruit> items = { {"kiwi",4}, {"apple",3}, {"banana",3},
				{"mango",5}, {"watermelon",1}, {"coconut",1} };

	// Insert every element to the PQ
	SLPriorityQueue<Fruit>* pqPtr = new SLPriorityQueue<Fruit>();
	for (int i = 0; i < items.size(); i++)
	{
		bool success = pqPtr->enqueue(items[i]);
		if (!success)
			cout << "Failed to add item " << i << " to the PQ." << endl;
	}  // end for

	// Iteratively get the most prioritized element from PQ
	while (!pqPtr->isEmpty())
	{
		try
		{
			Fruit f = pqPtr->peek();
			cout << "peek: " << f.getFruitName() << " " <<
f.getSourDegree() << endl;
		}
		catch (PrecondViolatedExcep e)
		{
			cout << e.what() << endl;
		}  // end try/catch

		cout << "Empty: " << pqPtr->isEmpty() << endl;
		cout << "Remove: " << pqPtr->dequeue() << endl;
	}  // end for

	cout << "remove with an empty priority queue: " << endl;
	cout << "Empty: " << pqPtr->isEmpty() << endl;
	cout << "remove: " << pqPtr->dequeue() << endl; // nothing to remove!
	getchar();
	return 0;
}  // end main
```

```
peek: mango 5
Empty: 0
Remove: 1
peek: kiwi 4
Empty: 0
Remove: 1
peek: apple 3
Empty: 0
Remove: 1
peek: banana 3
Empty: 0
Remove: 1
peek: coconut 1
Empty: 0
Remove: 1
peek: watermelon 1
Empty: 0
Remove: 1
remove with an empty priority queue:
Empty: 1
remove: 0
```

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