2025-10-23

# Priority Queues

## Introduction

### Abstract Data Type

Described [abstractly](https:///princomp.github.io/lectures/data/intro#abstract-data-types), [a priority queue](https://en.wikipedia.org/wiki/Priority_queue) is (the differences with queue are **in bold**):

* a finite collection of elements, **endowed with a priority**
* in **no** particular order,
* that may contain the same element multiple times.

Generally, it has operations to…

* … create an empty priority queue,
* … test for emptiness,
* … add an element **with a given priority**,
* … remove the element **with the highest priority**,
* … **increase the priority of a particular element**.

Letting a greater priority means “more important” is called a *max-priority queue*, it is also possible to implement *min-priority queue*, where the most element important are the ones with the lowest priority. In both cases, a decision must be made if multiple elements have the same priority: we can decide arbitrarily, using the element value, take the “first one” in the structure, etc.

Exactly like a people waiting **at the ER**, priority queues implement a **“most-important-in-first-out”** principle.

## Possible Implementation

### Using Arrays

Here is an implementation of priority queues using arrays:

﻿using System; // This is required for the exception.

class PQueue<TPriority, TValue> where TPriority : IComparable<TPriority>
{
 class Cell
 {
 public TPriority Priority { get; set; }
 public TValue Value { get; set; }
 public Cell(TPriority priorityP, TValue valueP)
 {
 Priority = priorityP;
 Value = valueP;
 }
 public override string ToString()
 {
 return Value + " (priority: " + Priority + ")";
 }
 }
 private Cell[] mArray;
 public PQueue(int sizeP = 10)
 {
 mArray = new Cell[sizeP];
 }
 public void Add(TPriority priorityP, TValue valueP)
 {
 // slot is the index where we will add the element
 int slot = -1;
 // index is where we are currently looking for
 // a slot in the arry.
 int index = 0;
 while (index < mArray.Length && slot == -1)
 {
 if (mArray[index] == null)
 {
 slot = index;
 }
 else
 {
 index++;
 }
 }
 if (slot == -1)
 {
 throw new ApplicationException("Queue is full, cannot add " + valueP + " with priority " + priorityP + ".");
 }
 else
 {
 mArray[slot] = new Cell(priorityP, valueP);
 }
 }

 public int MinPriority()
 {
 int index = 0;
 // We begin by looking for a value
 // in mArray that is not null.
 bool notNull = false;
 while (index < mArray.Length && !notNull)
 {
 if (mArray[index] != null)
 {
 // We found a value that is not null.
 notNull = true;
 }
 else
 {
 index++;
 }
 }
 // If we exit and notNull is still false,
 // it means there is no non-null cell in
 // the array.
 if (!notNull)
 {
 throw new ApplicationException("Queue is empty, no index with minimal priority.");
 }
 // Minimal priority found "so far".
 TPriority minP = mArray[index].Priority;
 // Index of the minimal priority found "so far".
 int minI = index;
 while (index < mArray.Length)
 {
 // The following if is crucial: there may
 // be null values in the array, and we should
 // not try to access the Priority property
 // if mArray[index] is null.
 if (mArray[index] != null)
 {
 // If we found a lower priority,
 // we update the minP and minI
 // values.
 if (mArray[index].Priority.CompareTo(minP) < 0)
 {
 minP = mArray[index].Priority;
 minI = index;
 }
 }
 index++;
 }
 return minI;
 }

 public TValue Peek()
 {
 // Looking at the most urgent Cell
 // uses MinPriority.
 return mArray[MinPriority()].Value;
 }

 public string Extract()
 {
 // Removing the most urgent Cell
 // relies also on MinPriority().
 int minE = MinPriority();
 Cell cellE = mArray[minE];
 mArray[minE] = null;
 return cellE.ToString();
 }

 public override string ToString()
 {
 string ret = "";
 for (int i = 0; i < mArray.Length; i++)
 {
 if (mArray[i] != null)
 {
 ret += mArray[i].ToString();
 }
 else
 {
 ret += "(empty)";
 }
 ret += "\n";
 }
 return ret;
 }
}

[*(Download this code)*](https:///princomp.github.io/code/projects/PQueue_array.zip)

This implementation as the following performance:

* Add is $O\left(n\right)$, it may take $n$ steps to find an empty slot,
* MinPriority is also $O\left(n\right)$, we will have to go through the entire array to find the Cell with the highest priority.
* Peek and Extract both rely on MinPriority, so they are also $O\left(n\right)$.

### Using Lists

An implementation using lists would be very similar to the one using array, except that Add would be $O\left(c\right)$, since inserting in a list can simply be done at the beginning.

### Using Heaps

A maximally efficient implementation of priority queues is given by [heaps](https://en.wikipedia.org/wiki/Heap_%28data_structure%29), which is

* A complete binary tree[[1]](#footnote-28) (that we will represent in an array), – Such that the priority of every (non-root) node is less important than the priority of its parent.

Note that this is different from being a binary search tree.

﻿using System;
using System.Collections.Generic;

public class PQueue<TPriority, TValue> where TPriority : IComparable<TPriority>
{

 class Cell
 {
 public TPriority Priority { get; set; }
 public TValue Value { get; set; }
 public Cell(TPriority priorityP, TValue valueP)
 {
 Priority = priorityP;
 Value = valueP;
 }
 public override string ToString()
 {
 return Value + " (priority: " + Priority + ")";
 }
 }

 private Cell[] mArray;
 // Number of items in queue.
 private int count = 0;

 public PQueue(int size = 100)
 {
 if (size < 10)
 size = 10;
 mArray = new Cell[size];
 }

 public bool IsEmpty()
 {
 return count == 0;
 }

 public bool IsFull()
 {
 return (count == mArray.Length - 1);
 }

 public void Clear()
 {
 count = 0;
 }

 public TValue Peek()
 {
 if (IsEmpty()) throw new ApplicationException("Queue is empty, no most urgent value.");
 return mArray[1].Value;
 }

 public void Add(TPriority priorityP, TValue valueP)
 {
 if (IsFull()) throw new ApplicationException("Queue is full, cannot add " + valueP + " with priority " + priorityP + ".");

 // Otherwise, we will be able to add an element,
 // so count must increment.
 count++;
 // We now look for a place to insert the value.
 int hole = count;
 // As long as hole > 1 and priorityP is less than
 // the priority at hole / 2…
 while(hole > 1 && priorityP.CompareTo(mArray[hole / 2].Priority) < 0)
 {
 mArray[hole] = mArray[hole / 2];
 hole /= 2;
 // We divide hole by 2
 // and move the data at hole / 2 at hole.
 }
 // Once this is done, we can insert the new value.
 mArray[hole] = new Cell(priorityP, aValue);
 }

 public TValue Extract()
 {
 if (IsEmpty())
 throw new ApplicationException("Queue is empty, cannot extract from it.");

 // Save the data to be returned.
 TValue value = mArray[1].Value;

 // put the last item in the tree in the root
 mArray[1] = mArray[count];
 // We have one less element now
 count--;

 // Move the lowest child up until we've found the right spot
 // for the item moved from the last level to the root.
 PercolateDown(1);

 return value;
 }

 private void PercolateDown(int hole)
 {
 int child;
 // save the hole's cell in a tmp spot
 Cell pTmp = mArray[hole];

 // keep going down the tree until the last level
 for (; hole \* 2 <= count; hole = child)
 {
 child = hole \* 2; // get right child
 // check right and left child and put lowest one in the child variable
 if (child != count && mArray[child + 1].Priority.CompareTo(mArray[child].Priority) < 0)
 child++;
 // put lowest child in hole
 if (mArray[child].Priority.CompareTo(pTmp.Priority) < 0)
 {
 mArray[hole] = mArray[child];
 }
 else
 break;
 }
 // found right spot of hole's original value, put it back into tree
 mArray[hole] = pTmp;
 }

 /// <summary>
 /// Assumes all but last item in array is in correct order
 /// Shifts last item in array into correct location based on priority
 /// </summary>
 public void BuildHeap()
 {
 for (int i = count / 2; i > 0; i--)
 PercolateDown(i);
 }

 public override string ToString()
 {
 string returned = "";
 for (int i = 1; i <= count; i++)
 {
 returned += mArray[i].Value.ToString() + "; ";
 }
 return returned;
 }

 // return string with contents of array in order (e.g. left child, parent, right child)
 public string InOrder()
 {
 return InOrder(1);
 }
 private string InOrder(int position)
 {
 string returned = "";
 if (position <= count)
 {
 returned += (position \* 2) + "\t";
 returned += mArray[position].Value.ToString() + "\n ";
 returned += InOrder(position \* 2 + 1) + "\t";
 }
 return returned;
 }
}

[*(Download this code)*](https:///princomp.github.io/code/projects/PQueue_heap.zip)

1. A complete binary tree is such that all levels are filled completely except the lowest level, which is filled from as left as possible. [↑](#footnote-ref-28)