Queues

Algorithms and data structures ID1021

Johan Montelius

Fall term 2022

Introduction

We're now ready to take a look at a very common data structure that I'm sure your familiar with from your every day life. We're going to implement a queue i.e. a row of items waiting in line. The line is of course ordered so if you add an item to the queue you will have to wait for all other items in the queue to be removed before your new item is in the front of the queue. Queues are also called FIFO, *first-in-first-out*, that describes the functionality. This can be compared to a stack that is also referred to as *last-in-first-out*.

We will implement a queue using two different approaches, using an array and using a linked list. The linked list implementation is as you will see less complicated and although it might not be as efficient the simplicity is often an advantage.

A linked list

The simplest way to implement a queue is to simply use a linked list. A queue then has a only one property, the `head`, and new elements are simply added to the end of this list. Your first queue could look like this:

```
public class Queue {

    Node head;

    public class Node {

        public Integer item;
        public Node tail;

        public Node(Integer item, Node list) {
            this.item = item;
            this.tail = list;
        }
    }

    public Queue() {
      :
    }

    public void add(Integer item) {
      :
    }

    public Integer remove() {
      :
    }
```

What is the drawback of this implementation? What is the cost of removing the next element? What is the cost of adding a new element? Can we do better?

Change the structure of the queue so it holds a pointer to the first element, the `head`, of the queue but also a pointer to the last element. When adding a new item you can then access the last node directly and attach the new node immediately instead of traversing the list to find the last node. You have to be careful when removing the last element and when adding the first element but you should be able to do the implementation in a few lines of code.

Breadth first traversal

In the previous assignment you implemented a depth first traversal of a tree using a stack. What happens if we use a queue instead? Imagine you enter a node and then add the left and right branches to a queue. When it is time to process the next node you take the first node in the queue so you will of course move to the left branch. What will happen of we repeat this procedure?

Reuse the implementation of a binary tree but now use a queue when constructing the iterator. As before you will have one node that is the next node in the sequence but now as you process a node (in `next()`) you add both the left branch and the right branch (if they are not null) to the queue before removing the next node from the queue.

Note that we have two kinds of `Nodes`: the ones that make up the linked list in the queue (`Queue.Node`), and the ones that make up the tree

(`BinaryTree.Node`). The nodes of the queue should of course refer to the nodes of the tree.

The traversal order is called *breadth first* since we traverse all nodes of one *level* before going to the next level that is deeper. Compare to *depth first* where the depth is given priority before continuing on the same level.

Using a breadth first strategy is often a good choice if you're looking for a node in a tree with some special property but you want to find the node that is closest to the root. One could of course use a depth first strategy but imagine if the left branch is a thousand nodes deep but the solution can be found in the first node to the right.

Breadth first approaches are also useful when we have more general structures where some paths might be circular. A depth first strategy could then result in an endless loop where the breadth first strategy would find what your looking for.

An array implementation

In the same way as you implemented a stack both by using a linked list and using an array you can implement a queue using an array. The idea is simple but the implementation will probably take you some time since there are so many corner cases. Before you start coding you need to draw pictures of the different situations and only then do the coding. You could of course cheat and google for "java queue array implementation" to look at any of the six million examples but if you do it from scratch yourself you will learn so much more so give it a try.

the basic idea

The basic idea is of course to have the item in the queue represented in an array. Let's say that the array has a length of n and that the first element of the queue is at position 0 and the last element at position $k-1$ (k is the first free slot). Adding a new item to the queue can then be implemented simply by writing the new item at position k and increment k by one (yes, k could then be equal to n which is not good but that's something you need to solve).

What should we do when we remove an item from the array? One solution would be to move all items from 1 to $k-1$ one step closer to the beginning. This would work and you can have this as your first solution but it is of course a very costly operation. A better approach is to keep track of the first element in the queue using an index in the array. Let's say that the first element is at position i, removing and element is then simply done by returning the element at position i and increment i by one.

So far so good, this is the general plan but this is where it starts to be complicated.

wrap around

What should you do when k is equal to n, should we give up? Should you allocate a new larger array and copy the items over to the new array? Hmm ... the first item is at position i so the slots between 0 and $i-1$ are free. Could we add the next item at position 0, let k be 1 and then carry on as before?

This is what you should implement but as you can imagine things are now slightly more complicated. If you consider the following cases you will be fine.

- If i is equal to k the queue is empty. The initial queue will have both set to 0. Trying to remove an item from an empty queue will simply return `null`.
- When increment k (after adding a new item) it should be set to 0 (not to n) if k is equal to $n-1$.
- If k is equal to i after being incremented the queue is full and you have problem.
- When increment i (after remove) it should be set to 0 (not to n) if i is equal to $n-1$.

Note - when you remove an item from the queue you should also set the position to `null`. We do not want to have references to items that are no longer needed.

Do some tests where you increase the complexity and try to catch the different corner cases when either k or i needs to wrap around. The only thing left is to handle arbitrary large queues.

dynamic queue

As with the dynamic stack you should now handle the case where the queue is full. The solution, once you get it right, is rather simple. Let's say that you after increment k realize that $k$is equal to i. The size of the array that you have is n so you simply allocate a new array of size $2*n$. The copying is then done as follow:

- All items from i to $n-1$ are copied staring at 0 and the index i is set to 0.
- Then, all item from 0 to $k-1$ are copied and k is set to n.

An optional task is to also implement a shrinking queue in the same way as the stack could be reduced in size. You need to keep track of the total number of elements in the queue and if this number drops below let's say

$n/4$ then you allocate an array of size $n/2$ and copy the queue over to the new array.

Note that you now have two cases, one where $0 < i < k < n$ and one where $0 < k < i < n$. When you copy you need to keep track of what you're doing.

Queuing theory

The most common usage of queues is in concurrent programs i.e. programs with more than one executing thread. Think about a printing service that should be open to receive requests for printing documents and then print them in the order they arrived. The way to solve this is having to threads of control, one that receives new orders and places them in a queue and one that takes the next order from the queue and sends it to the printer.

The usage of queues in larger systems is so important that it has become a computer science topic by its own: with a given set of resources, arrival of new requests with a given distribution in time, what is the maximum length of the queue, how long time does a request in average stay in the queue before being process? These are the questions of *queuing theory* and could easily take up a whole course.