Cosc 241
Programming and Problem Solving
Lecture 15 (15/4/2019)
Queues

Lech Szymanski
lechszym@cs.otago.ac.nz

Keywords: queue

UNIVERSITY of OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

Queue ADT

- A *queue* is an abstract data type that represents item processing in a first in, first out manner.
- Basic operations just like for a stack, except that the "remove" and "add" methods (pop and push for stacks) operate on opposite ends of the data, rather than the same end.
- A useful abstraction for:
  - breadth first search,
  - event or order processing,
  - process scheduling,
  - simulation.
- The *priority queue* (later!) is a common extension

Queue ADT

```
public interface Queue241<T> {

  public void add(T element);
  public T remove();
  public T peek();
  public int size();
  public boolean isEmpty();

}
```

Singly linked list as queue

- A singly linked list seems an obvious match for representing the queue ADT in a data structure.
- Adding a reference to the last element makes sure that both the add and remove operations can be $O(1)$.
- Let's take the `SinglyLinkedList` from L10 and convert it into an implementation of `Queue`.

Add and remove

```
public class SLLQ<T> implements Queue241<T>{
  private SLLNode<T> first; // The next item to be removed
  private SLLNode<T> last; // The last item added
  /* ... */

  public void add(T element) {
    SLLNode newNode = new SLLNode(element, null);
    if (this.first == null) {
        this.first = newNode;
        this.last = newNode;
    } else {
        this.last.next = newNode;
        this.last = newNode;
    }
  }

  public T remove() throws EmptyQueueException {
    if (this.first == null) throw new EmptyQueueException("remove");
    T element = this.first.value;
    this.first = this.first.next;
    return element;
  }

  /* ... */
}
```

Alternative queue representation

- It has to be asked: can we use an array to represent a queue?
- This would be sensible if we “knew” that our queues would never grow beyond a fixed size.
- One approach is to remove by returning the element at position 0 and moving everything else down one space.
- But that makes `remove` $O(n)$.
- Can we arrange for both `add` and `remove` to be $O(1)$?

The circular array

- If we think of the array elements as being arranged on a circle, then we need only keep track of ‘first’ and ‘last’ indices.
- Adding involves storing at the last index and incrementing it.
- Removing involves returning the element at the first index, and incrementing it.
- We need to remember to wrap around at the end.

Add and remove

```
public class CircularQ<T> implements Queue241<T>{
  private T[] values;
  private int first, last, size;
  /* ... */

  public void add(T element) {
    values[last] = element;
    last++; size++;
    if (last >= values.length) last = 0;
  }

  public T remove() throws EmptyQueueException {
    if (size == 0) throw new EmptyQueueException("remove");
    size--;
    T value = values[first];
    first++;
    if (first >= values.length) first = 0;
    return value;
  }

  /* ... */
}
```

What if?

We forget to enlarge the array storing a circular queue (because it is too much trouble) when the capacity is exceeded?

- Basically, we mess up completely
- How can we recover?
- Two possibilities:
  - Impose a hard limit on the size of such a queue and throw an exception when we overrun (probably also want to add a constructor that sets the size to something other than the default).
  - Bite the bullet and write the enlarge method.