

Data Structures G5029

Lecture 3 Queues and Linked Lists

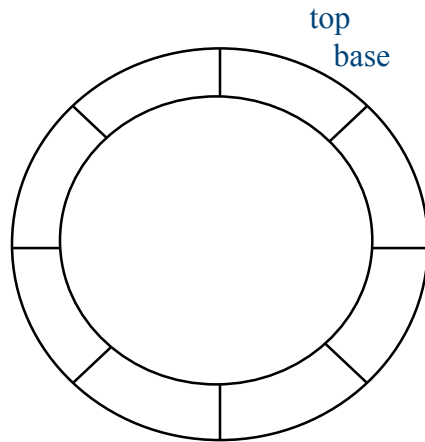
Kingsley Sage
Room 5C16, Pevensey III
khs20@sussex.ac.uk

© University of Sussex 2006

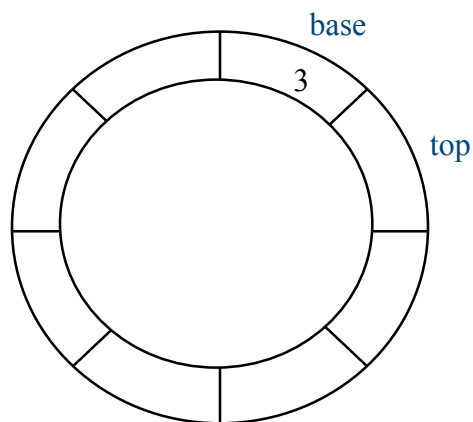
Lecture 3

- Queues
 - A typical analogy is the “queue of people”.
 - A way of buffering a stream of objects, in which the the First In is the First Out (FIFO).
 - What might we use a queue for?
- Functional requirements of a queue.
 - A queue is similar to a stack, except that you join the queue at one end and leave it at the other.
 - We shall again use an array to storing the items in the queue.
 - We shall need two pointers, one to indicate where the next item to be placed in the queue should be stored (`top`), and another to indicate the location of the next item to be retrieved (`base`).
 - We shall treat the underlying array as a circular buffer.

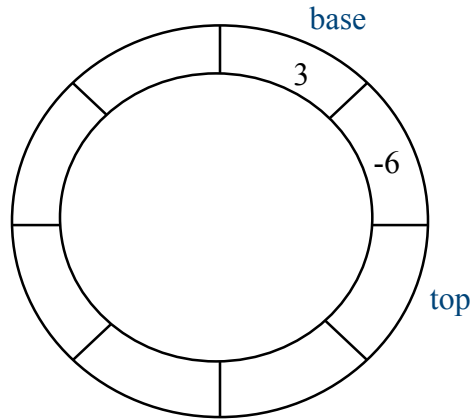
Queues – Conceptual diagram



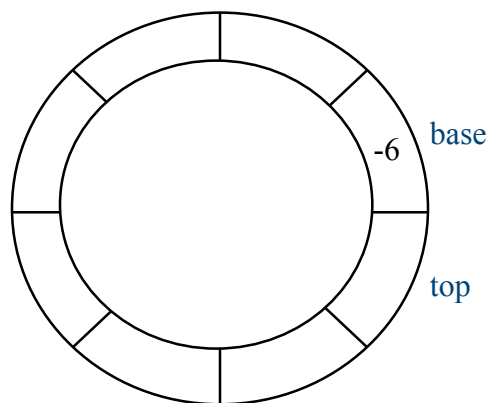
Queues – Conceptual diagram Adding an item



Queues – Conceptual diagram Adding another item



Queues – Conceptual diagram Using an item (de-queueing)



Simple implementation

- Items to be enqueued will be written in the array at `queue[top]`.
- Items to be dequeued will be read from `queue[base]`.
- After each write `top` is incremented by one.
- After each read `base` is incremented by one.
- The amount by which `top` is “in the lead” is given by a variable `size` which will always lie between 0 when the queue is empty and `queue.length` when the queue is full.
- Incrementing of `top` and `base` is always modulo `queue.length`.

```
public class SimpleQueue {
    private Object[] queue;
    private int size; // size of the queue
    private int top; // for next insertion
    private int base; // for next removal

    public SimpleQueue(int capacity) {
        queue = new Object[capacity];
        size = top = base = 0;
    }

    public boolean isEmpty() {
        return (size == 0);
    }
    public boolean isFull() {
        return (size == queue.length);
    }
    public void enqueue(Object item) {
        if (isFull()) {
            throw new RuntimeException("queue overflow");
        } else {
            queue[top++] = item;
            if (top == queue.length) top = 0;
            ++size;
        }
    }

    public Object dequeue() {
        if (isEmpty()) {
            throw new RuntimeException("queue underflow");
        } else {
            Object item = queue[base++];
            if (base == queue.length) base = 0;
            --size;
            return item;
        }
    }
}
```

Simple implementation

- We can use modulo arithmetic as an alternative bounds length checking ..

```
if (top == queue.length) {
    top = 0;
}

// Can use modulo arithmetic instead ..
top = top % queue.length;

// which can also be written as ..
top %= queue.length
```

Simple implementation

- We can use techniques similar to the ones we saw in the previous lecture for stacks to define an interface class for queues, and to increase the queue size by copying it to a larger one if required. See on-line course notes for details.

Linked list implementations

- Up to now we have been using arrays as the basis for our implementations of stacks and queues.
- We have relied on statically declared array size and techniques to copy arrays to new larger arrays if required.
- This is not optimally efficient although it works just fine. What would be more efficient would be a data structure that dynamically adjusted its size such that it provided precisely the right amount of data storage for the task at any time.
- We achieve this using the concept of linked lists ...

Linked lists

- Consider a linked list of integers ...
- Each list member consists of two parts, the data item and a link to the next member of the list.
- The final member in the list uses a null reference to show that there are no further members in the list.



Linked lists implementation

- We define a Java class to represent a list member (or node) ...

```
class ListNode {
    Object data;
    ListNode next;

    ListNode(Object data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}

// this operator is self referential ...
```

Stacks using linked lists

```
import java.util.NoSuchElementException;
public class StackList implements Stack {
    private ListNode top;
    public StackList() {
        top = null;
    }
    public boolean isEmpty() {
        return (top == null);
    }
    public void push(Object item) {
        top = new ListNode(item, top);
    }
    public Object pop() {
        if (top == null) {
            throw new NoSuchElementException();
        } else {
            Object item = top.data;
            top = top.next;
            return item;
        }
    }
}
```

Linked lists

- We shall see more about linked lists in the next lecture.
- For details of the linked list implementation of the queue interface, see the on-line course notes.

Next time ...

- Enumerations and iterators.
- More on linked lists.