Queues

Edwin Ahlstrand

Fall 2022

Introduction

This assignment is about implementing queues in different ways with the principle of first in first out (FIFO). The different implementations are with both single and double linked lists, and as an array. Lastly a queue is implemented for an iterator to a binary tree so it can use breadth-first traversal. The code for each implementation is explained in each step through comments for each row.

Queues

A queue is a linear data structure where elements are stored in their order of entry.

When adding an element, it is appended at the end of the queue, and when retrieving the next element in the queue, the first element item is returned and it's next element is assigned as the first element.

Single Linked List

The queue is implemented using a single linked list with a head node, where each node has access to the next node in the queue. The code is explained through the comments:

```
public class SingleLinkedQueue<T> implements Queue<T>{
    Node head;  // First node in queue
    @Override public void add(T item) {  // Adds item as node to queue
        Node node = new Node(item);  // Node to be added
        if (head!=null){  // Not empty, append node to bottom
            Node nxt = head;  // Next node in queue
            while(nxt.right != null){  // Step through as long as next isn't null
                nxt = nxt.right;  // Go to next node
            }
            nxt.right = node;  // Assign last nodes tail as given node
        }
        else{  // Empty, create new head
            head = node;  // The new item as a node is now start
        }
    }
    @Override public T remove() {  // Removes first item and returns it's value
        if(head!=null){  // If there is something to remove
            Node<T> node = head;  // Save head to be returned
            head = node.right;  // Assign new head as next one
            return node.item;  // Returns the previous heads item
        }
        return null;  // Else returns nothing
    }
}
```

Double Linked List

The double linked list implementation of a queue is similar to the single linked, but both the head and tail of the queue is saved. The code is explained through the comments:

```
public class DoubleLinkedQueue<T> implements Queue<T>{
    Node head;  // First node in queue
    Node tail = head;  // Last node in queue
    @Override public void add(T item){  // Adds element to tail of queue
        if(head != null){  // If list isn't empty, attach node to tail
            this.tail = new Node(tail, item);  // Set tail as new node
        }
        else{  // Set head and tail to first element
            head = new Node(item);  // Set head to node
            tail = head;  // Set tail to head
        }
    }
    @Override public T remove() {  // Removes first item and returns it's value
        if(head!=null){  // If there is something to remove
            Node<T> node = head;  // Node to be returned
            head = node.right;  // Set new head as old heads next node
            return node.item;  // Returns nodes item
        }
        return null;  // Else returns nothing
    }
}
```

Array

When implementing a queue as an array, lot of corner cases had to be considered. The main idea is to keep track of the position of the first and last element in the array, which is useful in wrapping around the array if it is filled with elements to the end but there are unused position in the beginning of the array.

If every position in the array is used, then it can be expanded by allocating a new array and filling it with the elements from first to last in the queue (which could wrap around).

When adding an element it has to check if it the array has wrapped back around or if the last position is greater than the first and act accordingly.

The indexes are carefully incremented or reset depending on the state of the array, which can be seen in the code for the add and remove method on the next page.

```
@Override public void add(T item) {  // Adds item to queue
    if(last >= first){  // Last is same or after first
        if(empty()){  // Empty
            queue[first] = item;  // Place item first place, no increment
        }
        else if(last != queue.length-1){  //  Free space to the right
            queue[++last] = item;  // Place item at last incremented
        }
        else{  // Last would overflow, wrap around?
            if(first != 0){  // First position is free, wrap around
                queue[last=0] = item;  // Place item at last=0
            }
            else{  // Array full, expand instead
                expand(item);  // Expands array and adds item
            }
        }
    }
    else{  // Last is before First, has wrapped around
        if((last+1)!=first){  // Won't overwrite first
            queue[++last] = item;  // Still unused space until First
        }
        else{  // Will overwrite First, expand
            expand(item);  // Expands array and adds item
        }
    }
}
@Override public T remove() {  // Removes first element in queue
    if(!empty()){  // Not empty
        if(first != queue.length-1){  // Won't wrap around
            return queue[first++];  // Returns item and increments First
        }
        else{  // Will wrap around
            first = 0;  // Start over, first at position 0
            return queue[queue.length-1];  // Returns last item
        }
    }
    else{  // Empty
        return null;  // Nothing
    }
}
```

Breadth First Traversal

Breadth first is going through every layer of the tree in order. This is done by using a queue and saving each layer from left to right to the queue, and then saving those nodes left and right nodes afterwards. The implementation is fairly simple compared to depth first traversal, which can be seen in the code:

```
public class TreeIterator<T> implements Iterator<T> {
    private Node next;  // Next node for iterator
    private Queue<Node> queue;  // Queue consisting of nodes
    public TreeIterator() {  // Constructor
        queue = new DoubleLinkedQueue();  // Creates queue
        next = BinaryTree.this.root;  // Start at root
        queue.add(next);  // Adds root to queue
        if(next == null) {return;}  // No nodes
    }
    @Override public boolean hasNext() {  // If there exists a value
        next = queue.remove();  // Select next now so null can be checked
        return next != null;  // Is return not null?
    }
    @Override public T next(){  // Selects the next value
        if(next.left != null){  // Add left node if it exists
            queue.add(next.left);  // Adds it's left node to queue
        }
        if(next.right != null){  // Add right node if it exists
            queue.add(next.right);  // Adds it's right node to queue
        }
        return (T)next.item;  // Return current item
    }
    @Override public void remove() {throw new UnsupportedOperationException();}
}
```