Queues

Dhruba Das

Spring Fall 2023

Introduction

This assignment was much more simple compared to binary trees; there are two problems in this assignment; implementing the different kinds of stacks and implementing a queue in the Binary tree iterator instead of a stack.

Linked List Queue

This was quite easy to implement; for the first one, I only had to create a reference to the next node and the head/first node, and implement methods add and remove. In the second linked list, we improve on the first one by adding a pointer to the end of the queue (i.e node which has queued least). Below is the code snippet for the improved and final linked list:

Node structure

The node structure is the same as the first linked list, but now has a pointer to the last (ultimo) node. Below is the code snippet:

```
public class QueueLinkedList {
    Node first;
    Node ultimo;

    private class Node{
        Integer item;
        Node next;

        private Node(Integer itm, Node nxt){
            this.item = itm;
            this.next = nxt;
        }
    }
```

```
    public QueueLinkedList(){
        first = null;
        ultimo = null;
    }
```

Enqueue method

This method adds a node to the end of the queue; includes special cases and is more efficient in traversal as we know have a pointer to the last node. Below is the code snippet:

```
public void enqueue(Integer itm){
        Node last = new Node(itm, null);
        if(ultimo != null){
            ultimo.next = last;
        }
        ultimo = last;
        if(first == null){
            first = last;
        }
    }
```

Dequeue method

This method removes the first node (i.e node which has queued the longest); includes special cases and is more efficient in traversal as we know have a pointer to the last node: Below is the code snippet:

```
public Integer dequeue(){
        if(first == null)
          return null;

        Integer ret = first.item;
        first = first.next;

        if(first == ultimo)
          first = ultimo = null;

        return ret;
    }
```

Breadth first traversal

In this section, we implement a queue in the Binary Tree iterator from the previous assignment instead of a stack; Code snippet below shows the

changes which were made in order to do this:

```
private class TreeIterator implements Iterator<Integer> {
      private Node next;
      private QueueLLGeneric <Node> q;

      public TreeIterator() {
                q = new QueueLLGeneric<Node>(); //initializes a new queue
                next = root; //start at root
                q.enqueue(root);
         }

        @Override
            public boolean hasNext() {
                return next != null;
                // returns true if next isn't null; returns false if next is null;
            }


            @Override
        public Integer next() { // returns value of next node
            if (!hasNext()){
               throw new NoSuchElementException(); //Checks if there is next node
                }
                Integer val = next.value;

                q.enqueue(next.left);//Add left branch to queue
                q.enqueue(next.right);//Add right branch to queue

                q.dequeue();
                next = q.firstItem();

                return val;
            }

            @Override
            public void remove() {
                throw new UnsupportedOperationException();
            }
```

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.

Array Queues: Static vs Dynamic

Here, we implement queues using array data-structures instead of a linked list.

This is the first implementation of the array; below is the code snippet for the constructor and the class:

```
public class QueueArray {

    Integer[] queue;
    int first, last;
    int size;

    public QueueArray(){
        size = 4;

        queue = new Integer[size];

        first = 0;
        last = 0;
    }
```

We the move on to the method enqueue implementation, which has a time complexity of O(1).

```
public void enqueue(Integer itm){
        queue[last] = itm;
        last = (last + 1) % size;
        if(last == first){
          System.out.println("Array full");
        }
    }
```

Here in the static array implementation, we face a problem when the array is full (here complexity is the same, i.e O(1)). This problem is circumvented by implementing the enqueue method as so (Dynamically):

```
public void enqueue(Integer itm){
        queue[last] = itm;
        last = (last + 1) % size; //last points to next empty slot
        if(last == first){
            Integer[] copy = new Integer[size*2];
            int c = 0;
            for(int i = first; i < size; i++){
                copy[c++] = queue[i];
```

```
            }

            queue = copy;
            first = 0;
            last = c;
            size = size*2;
        }
    }
```

Here, we double the size of the array (i.e dynamic) once the array is full. This allows us to keep enqueuing items without having to worry about the size of the array.

For the dequeue method, it's the same for both Static and Dynamic, with a time complexity of O(1):

```
public Integer dequeue(){
        if(isEmpty())
          return null;
        Integer ret = queue[first];
        queue[first] = null;

        if(first == last){
            first = 0;
        }
        else{
            first++;
        }
        return ret;
    }
```

For dequeue, we first check whether the array is empty first; if so, we return null. We then make the first item in the array a null after storing it in the ret variable. If first is the same as last, we make first to be zero, thus indicating that there aren't any more items in the array.