Algorithms and Data Structures (ADS2)

Laboratory Sheet 3

This lab sheet contains material based on Lectures 10-12. This exercise is not assessed but should be completed to gain sufficient experience in the implementation of elementary abstract data types, and the testing thereof.

Exercise

You are to implement the Dequeue abstract data type (ADT) using two different data structures. Then, you will use this ADT to define generic Stack and Queue ADTs. Finally, you will use a stack to implement a non-recursive quicksort.

Part 1

a) Implement in Java the merge sort algorithm for linked lists introduced in Lecture 10 (slide 36). Use the following class structure to implement linked lists.

```
public class LinkedList<Item>{
  private Node<Item> head;

  private static class Node<Item>{
    private Item key;
    private Node<Item> next;
  }

  public LinkedList(){
    head = null;
  }

...
```

The implementation is straightforward following the pseudocode in the slides.

Part 2

a) Implement in Java the Dequeue abstract data type introduced in Lecture 12 (slide 35). Use resizable arrays in a circular fashion (slide 26) in the class below. What is the time complexity of each operation (i.e. PUSH-BACK, PUSH-FRONT, POP-BACK, POP-FRONT)?

```
public class ResizingDequeue<Item>{
  private Item[] q;
  private int n;  // number of elements in the dequeue
  private int tail;
  private int head;

  public ResizingDequeue (){
    q = (Item[]) new Object[2];
    n = 0;
    head = 0;
    tail = 0;
  }

...
```

See Lecture 12, slide 27 for implementation details. All operations have amortised O(1) time complexity.

b) Implement the Dequeue abstract data type using a circular doubly linked list. Modify the class structure given in 1a for singly linked lists to include `prev` pointers and a sentinel. What is the time complexity of each operation?

```
public class LinkedDequeue<Item>{
  private Node<Item> nil;  // sentinel
  private int n;  // number of elements in the dequeue

  private static class Node<Item>{
    private Item key;
    private Node<Item> next;
    private Node<Item> prev;
  }

  public LinkedDequeue(){
    nil = new Node();
    nil.prev = nil;
    nil.next = nil;
    n = 0;
  }

...
```

See Lecture 10 for the implementation details of a circular doubly linked list with sentinel. All operations have O(1) time complexity.

Part 3

a) Implement the Stack ADT using the Dequeue you implemented in 2b.

Implement `Pop` with `PopFront` and `Push` with `PushFront`.

b) Implement the Queue ADT using the Dequeue you implemented in 2b.

Implement `Dequeue` with `PopFront` and `Enqueue` with `PushBack`.

c) Implement the Queue ADT using two stacks. What is the time complexity of each operation?

There are several possible ways to do this. Below, the sketch of one solution in which stack S1 always stores the oldest element on the top.

ENQUEUE(Q,x)
1. While S1 is not empty, push everything from S1 to S2
2. Push x to S1
3. Push everything back to S1

DEQUEUE(Q)
1. If S1 is empty, then underflow error
2. Pop an item from S1 and return it

In this implementation, ENQUEUE is O(n) while DEQUEUE is O(1).

Part 4
Using an auxiliary stack, implement an iterative version of quicksort. To reduce the stack size, first push the indexes of the smaller subarray.

Below is a simple solution without the optimisation on the stack size.

```
void quicksortIter (int a[], int p, int r){
  Stack<int> stack = new Stack<int>();

  stack.push(p);
  stack.push(r);

  while (!stack. isEmpty()){
    r = stack.pop();
    p = stack.pop();
    int q = partition(a, p, r);

    if (q-1 > p){
      stack.push(p);
      stack.push(q-1);
    }

    if (q+1 < r){
      stack.push(q+1);
      stack.push(r);
    }
  }
}
```