**Problem Sheet #2 Karl Parks**

Solutions

1. Given the following s**tack** class declaration, write the following methods using an arraybased implementation: **CODE AT END OF QUESTION**

* public void push(E data)
* public E pop()
* public boolean isFull()
* public boolean isEmpty()

*Note: there is no currentSize variable here. Do we need it?* **We do not require currentSize because we only care about the top item of the stack when we are manipulating the data structure.**

class Stack<E> { private int head; private int maxSize; private Object [] stack;

public Stack(int size) { maxSize = size; head = -1;

stack = new Object[maxSize];

}

public void push(E data) {

public E pop()

public boolean isFull()

public boolean isEmpty()

}

**class** Stack<E> {

**private** **int** head;

**private** **int** maxSize;

**private** Object [] stack;

**public** Stack(**int** size) {

maxSize = size;

head = -1;

stack = **new** Object[maxSize];

}

**public** **void** push(E data) {

**if** (isFull()) {

System.***out***.println("Stack is Full");

**return**;

}

head++;

stack[head] = data;

System.***out***.println("Pushed Element");

}

@SuppressWarnings("unchecked")

**public** E pop() {

**if** (isEmpty()) {

System.***out***.println("Stack is Empty");

**return** **null**;

}

Object tmp = stack[head];

stack[head] = **null**;

head--;

System.***out***.println("Popped Element");

**return** (E) tmp;

}

**public** **boolean** isFull() {

**return** maxSize-1==head;

}

**public** **boolean** isEmpty() {

**return** head<0;

}

**public** **static** **void** main(String[] args) {

System.***out***.println("Hello Karl");

Stack<String> stack1;

stack1 = **new** Stack<String>(4);

stack1.push("One");

}

}

2. Given the following array based **Queue** class declaration, write the methods specified. Both the enqueue and dequeue method must run in O(1) time. **CODE AT END OF QUESTION**

public void enqueue(E n);

* public E dequeue();
* public boolean isFull();
* public boolean isEmpty();

public class Queue<E> { private int maxSize; private int currentSize; private Object [] storage; private int front, rear;

public Queue(int size) { maxSize = size; currentSize = 0;

storage = (E[]) new Object[maxSize]; front = rear = 0;

}

public void enqueue(E obj)

public E dequeue()

public boolean isFull()

public boolean isEmpty()

}

**public** **class** Queue<E> {

**private** **int** maxSize;

**private** **int** currentSize;

**private** Object [] storage;

**private** **int** front, rear;

@SuppressWarnings("unchecked")

**public** Queue(**int** size) {

maxSize = size;

currentSize = 0;

storage = (E[]) **new** Object[maxSize];

front = rear = 0;

}

**public** **void** enqueue(E obj) {

//insert in back

**if** (isFull()) {

System.***out***.println("Queue is Full");

**return**;

}

storage[rear] = obj;

rear = (rear + 1) % maxSize;

currentSize++;

System.***out***.println("Queued Element");

}

@SuppressWarnings("unchecked")

**public** E dequeue() {

//grab from front

**if** (isEmpty()) {

System.***out***.println("Queue is Empty");

**return** **null**;

}

Object tmp = storage[front];

storage[front] = **null**;

front = (front + 1) % maxSize;

currentSize--;

System.***out***.println("Dequeue Element");

**return** (E) tmp;

}

**public** **boolean** isFull() {

**return** currentSize==maxSize;

}

**public** **boolean** isEmpty() {

**return** currentSize==0;

}

**public** **void** printVals() {

System.***out***.println("currentSize: " + currentSize);

System.***out***.println("front: " + front);

System.***out***.println("rear: " + rear);

**int** count = 0;

**for** (**int** i = front; count < currentSize; count++) {

System.***out***.println("storage:[" + i + "]: " + storage[i]);

i = (i + 1) % maxSize;

}

}

**public** **static** **void** main(String[] args) {

System.***out***.println("Hello Karl");

Queue<String> queue1;

queue1 = **new** Queue<String>(4);

queue1.enqueue("first");

}

}

1. Some circular queue implementations use the **mod** operator % in enqueue and dequeue operations. Explain why this is inefficient.

**I used the mod operator above. This sincerely improves code readability. However, it is inefficient because it requires an underlying computation that is not required when we simply can wrap an index when it reaches the maxSize.**

1. If a queue is implemented using a singly linked list with a head and tail pointer, you should always insert at the tail and remove from the head. Explain why this is so.

**This is a Queue. Linked Lists can be tricky in their implementation because it is very easily to “lose a link”. In a singly linked list, removing at the tail would require iterating through the entire array and saving the node right before the last, which is tricky and hard to do with a large O(n);**

1. What is a Priority Queue, and how does it differ from a standard queue?

**A priority queue is exactly what it sounds like, an ordered (prioritized) queue where you can insert and remove at specific spots along the data structure. A standard queue, you simply dequeue and enqueue at the ends of the structure with no manipulation in the center/middle.**

1. Priority Queues are almost always implemented with an ordered data structure. Why?

**A priority queue is a prioritized (ordered) structure of data.**

1. Write the following methods for a standard unordered singly linked list. There is a head pointer, but no tail pointer. **CODE AT END OF QUESTION**

*// inserts the given key into the first position in the list (head)* public void insertFirst(int key)

*// inserts the given key at the end of the list (there is no tail pointer)* public void insertLast(int key)

*inserts the key1 into the position immediately following the last instance of key2* *// if key2 exists in the list. Otherwise, the key1 is inserted at the end of the list.*

public void insertAfter(int key1, int key2)

*// removes the given key from the list if it exists, otherwise does nothing* public void delete(int key)

*// removes the first element (head) in the list* public void deleteFirst()

*// deletes the last instance of the key in the list* public void deleteLastInstance(int key)

*// removes the last element in the list (there is no tail pointer).*

public void deleteLast() {

*/ / reverses the order of the nodes in the list. i.e. if the list contains:*

*// HEAD->A->B->C->D*

*// then the method modifies the list so that it becomes:*

*// HEAD->D->C->B->A*

*// The method does not create a new list, but reverses the nodes by manipulating the links in the existing list.*

public void reverseList(){

**public** **void** insertFirst(E obj) {

Node<E> newNode = **new** Node<E>(obj);

newNode.next = head;

head = newNode;

currentSize++;

}

**public** **void** insertLast(E obj) {

Node<E> newNode = **new** Node<E>(obj);

Node<E> tmp = head;

**if** (head == **null**) {

head = newNode;

currentSize++;

**return**;

}

**while** (tmp.next != **null**) {

tmp = tmp.next;

}

tmp.next = newNode;

currentSize++;

}

**public** **void** insertAfter(E key1, E key2) {

Node<E> newNode = **new** Node<E>(key1);

Node<E> tmp = head;

Node<E> tmpDesired = **null**;

**boolean** found = **false**;

**for** (**int** i = 0; i < currentSize; i++) {

**if** (tmp.data == key2) {

tmpDesired = tmp;

found = **true**;

}

tmp = tmp.next;

}

**if** (!found) {

insertLast(key1);

**return**;

}

newNode.next = tmpDesired.next;

tmpDesired.next = newNode;

currentSize++;

}

**public** **void** delete(E key) {

Node<E> tmp = head;

Node<E> tmpDesired = **null**;

Node<E> beforeNode = **null**;

**boolean** found = **false**;

**for** (**int** i = 0; i < currentSize; i++) {

**if** (tmp.data == key) {

tmpDesired = tmp;

found = **true**;

System.***out***.println("Found Node to Delete");

}

**if** (i == 0) {

beforeNode = head;

**if** (found) {

head = tmpDesired.next;

currentSize--;

**return**;

}

}

**else** **if** (!found) {

beforeNode = beforeNode.next;

}

tmp = tmp.next;

}

**if** (!found) {

**return**;

}

beforeNode.next = tmpDesired.next;

currentSize--;

}

**public** **void** deleteFirst() {

**if** (currentSize != 0) {

head = head.next;

currentSize--;

}

**return**;

}

**public** **void** deleteLast() {

**if** (currentSize != 0) {

Node<E> tmp = head;

**for** (**int** i = 0; i < currentSize; i++) {

tmp = tmp.next;

}

tmp = **null**;

currentSize--;

}

**return**;

}

**public** **void** deleteLastInstace(E key) {

delete(key);

}

**public** **void** reverseList() {

Node<E> tmp = head;

Node<E> tmpHead = head;

Node<E> beforeNode = **null**;

System.***out***.println("--- Reversing List ---");

**for** (**int** i = 0; i < currentSize; i++) {

**if** (i == 0) {

beforeNode = head;

}

**else** **if** (i > 0) {

beforeNode = beforeNode.next;

}

**if** (i < currentSize) {

head = tmp;

}

tmp = tmp.next;

head.next = beforeNode;

}

}

8. Both Stacks and Queues can be implemented with either arrays or linked lists. Discuss the advantages and disadvantages of each implementation.

**Arrays are easy to implement, however one of their biggest cons is their allocated max size.**

**Linked Lists are harder to implement, however they do not have an allocated max size. Unfortunately, because of their extra information contained in each node, that can take up a larger portion of memory.**