LAB_05

ICS 202

Jalal Ali Zainaddin | 202154790 | 10/11/2023

Task 1.1: Recursive contains(Element e):

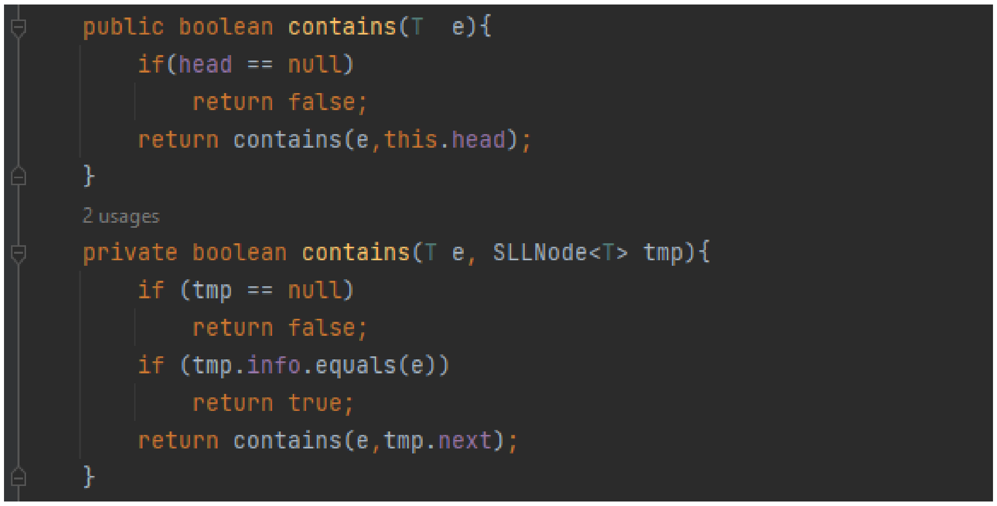
The previous iterative code:

> At first it checks if the head is null (empty), if it is, then return false. Then initialize a node as a head initially in order to be able to traverse, and continue checking if it is not empty to continue iterating through while loop. Inside the loop, we check whether the info of the node is equal to the given value, if nothing matches, return false.

The Recursive code:

We used helper method to check for special cases, such as list is empty, else we go through overloaded method, which holds the provided element and temporary node initialized as head, each time we check whether the current temp info matches the elem or not, else we call the method again with the next node, until it reaches the end.

Code:



```
public boolean contains(T  e){
    if(head == null)
        return false;
    return contains(e,this.head);
}
2 usages
private boolean contains(T e, SLLNode<T> tmp){
    if (tmp == null)
        return false;
    if (tmp.info.equals(e))
        return true;
    return contains(e,tmp.next);
}
```

Task 1.2: Recursive toString():

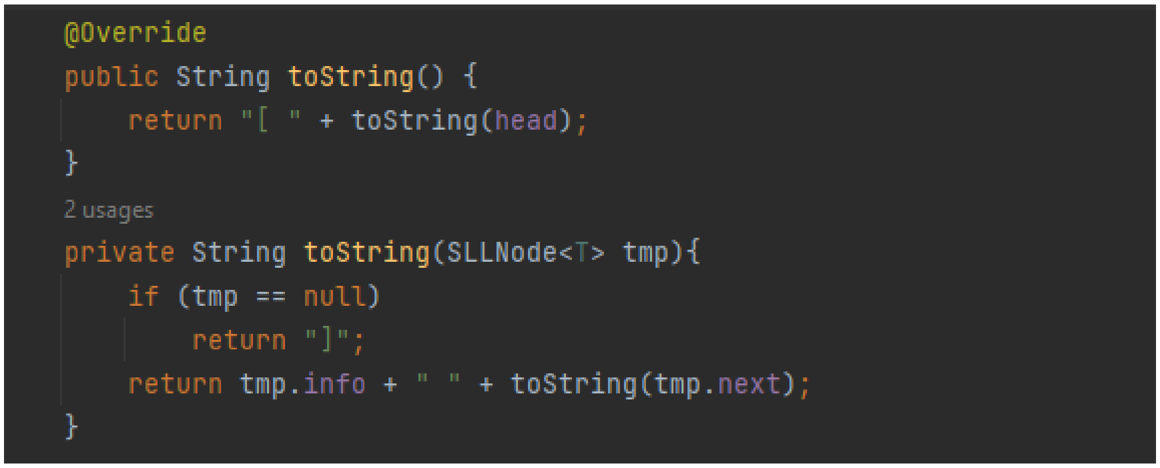
The previous iterative code:

At first it checks if the head is null (empty), if it is, then return empty list “[]”. Then create a string variable with starting bracket “[” then initialize temp node as a head initially in order to traverse, and continue adding info to the string and checking if it is not empty to continue iterating through while loop. If the node is null, add closing bracket “]” and return the string.

The Recursive code:

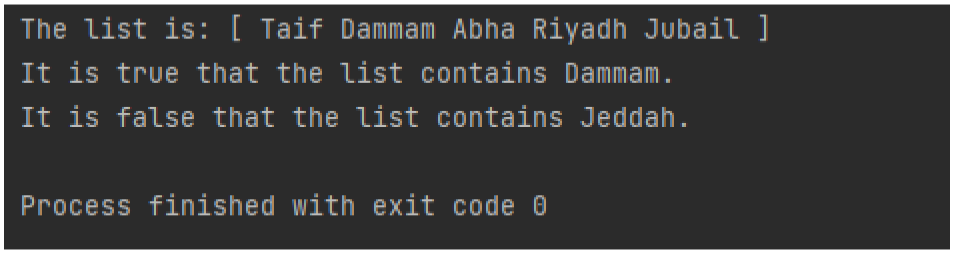
We used helper method to call the overloaded method with node as parameter as head initially and with starting bracket “[”, then we traverse through the nodes and we add info to the return statement each time until the node is empty, we add closing bracket “]”.

Code:



```
@Override
public String toString() {
    return "[ " + toString(head);
}
2 usages
private String toString(SLLNode<T> tmp){
    if (tmp == null)
        return "]";
    return tmp.info + " " + toString(tmp.next);
}
```

Output:



```
The list is: [ Taif Dammam Abha Riyadh Jubail ]
It is true that the list contains Dammam.
It is false that the list contains Jeddah.

Process finished with exit code 0
```

Task 2: Recursive dequeue():

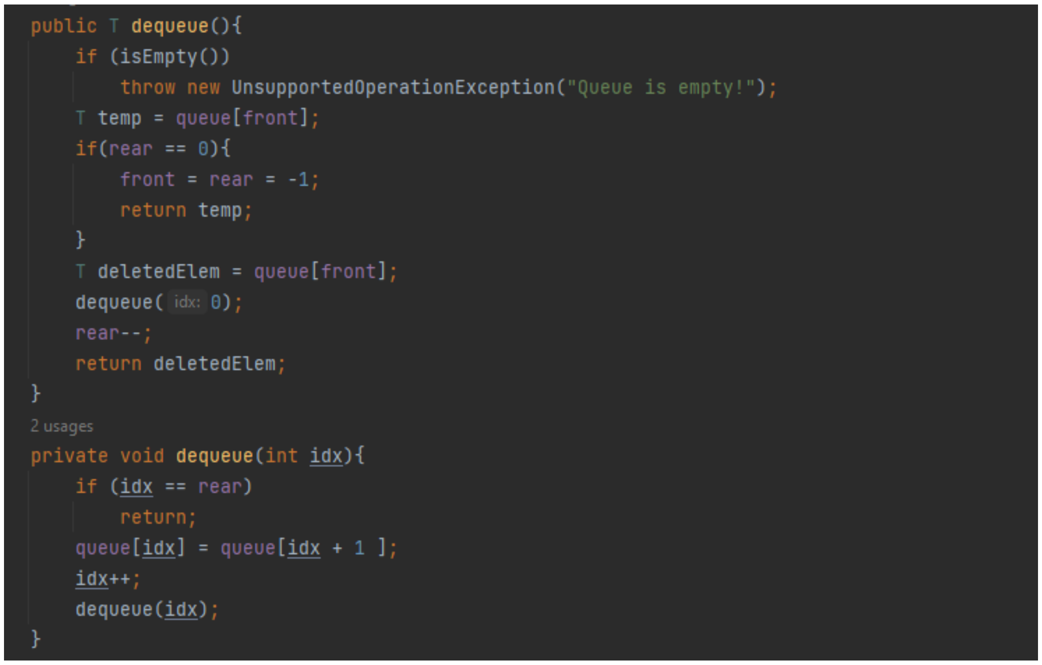
The previous iterative code:

Firstly, we check whether the queue is empty, if it is, throw an exception, then store the first element to be removed in a variable. Then if the queue has only one element, we make the front and rear equal to -1, and return the removed value, else, we use for loop to shift all elements from right to left through queue array, and decrement the rear by 1.

The Recursive code:

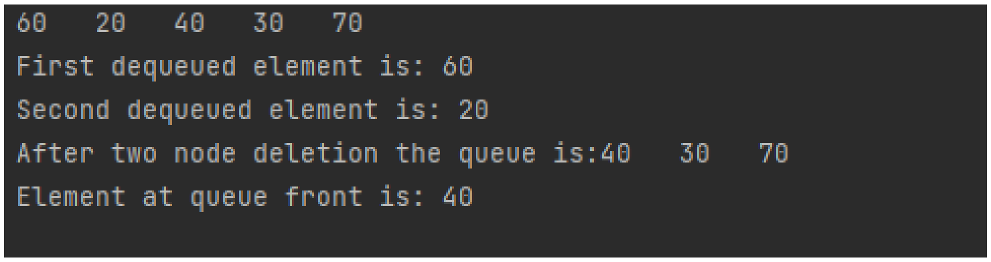
We used helper method to throw an exception if queue is empty or it has only one element, and store and return the first element to be removed after iterate recursively, we call overloaded void dequeue with idx integer in the parameter, and increment idx until it equals to the rear, we return;.

Code:



```
public T dequeue(){
    if (isEmpty())
        throw new UnsupportedOperationException("Queue is empty!");
    T temp = queue[front];
    if(rear == 0){
        front = rear = -1;
        return temp;
    }
    T deletedElem = queue[front];
    dequeue( idx: 0);
    rear--;
    return deletedElem;
}
2 usages
private void dequeue(int idx){
    if (idx == rear)
        return;
    queue[idx] = queue[idx + 1 ];
    idx++;
    dequeue(idx);
}
```

Output:



```
60   20   40   30   70
First dequeued element is: 60
Second dequeued element is: 20
After two node deletion the queue is:40   30   70
Element at queue front is: 40
```