Universidad Carlos III de Madrid

Algorithms and Data Structures (ADS)
Bachelor in Informatics Engineering
Computer Science Department

Lists, Stacks and Queues.

Authors: Isabel Segura Bedmar

April 2011
Departamento de Informtica,
Laboratorio de Bases de Datos Avanzadas (LaBDA)
http://labda.inf.uc3m.es/

Lists, Stacks and Queues.

Most of the information has been sourced from the books [1, 2].

Lists

A *list* is a collection of n elements stored in a linear order. The most common way for storing lists is using an *array* data structure. Each element in a list can be refered by an index in the range $[0, n-1]$ indicating the number of elements that preced e in the list. This representation provides that all operations performed on a given element take O(1) time. An array stores all elements of the list contiguously in memory and requires to initially know the maximum size of the list. This may produce an unnecessary waste of memory and other cases insufficient memory. A possible solution is to use a dynamic array (*java.util.ArrayList*) that is able to be reallocated when the space reserved for the dynamic array is exceeded. Unfortunately, this reallocating of the elements of a dynamic array is a very expensive operation. [1]

A linked list is a data structure that consists of a sequence of nodes such that each node contains a reference to the next node in the list. This representation does not require that elements of the list are stored contiguously in memory. Also, it just uses the space actually needed to store the elements of the list. On the other hand, linked lists do not mantain index numbers for the nodes and allow only sequential access to elements, while arrays allow constant-time random access. As it will be shown in following sections, the linked list data structure allows us us to implement some important abstract data structures such as stacks, queues, deques.

An implementation of a linked list may include the following methods:

- *isEmpty()*: test whether or not a list is empty.
- *size()*: return the number of elements of the list.
- *first()*: return the first element of the list. An error occurs if the list is empty.

[1] Most of the information has been sourced from the books [1, 2].

- *last()*: return the last element of the list. An error occurs if the list is empty.
- *next(v)*: return the next node of v in the list.
- *prev(v)*: return the previous node of v in the list.
- *remove(v)*: removes the node v the list.
- *addFirst(v)*: add the node v at the beginning of the list.
- *addLast(v)*: add the node v at the end of the list.
- *addBefore(v,new)*: add the node new just before the v node.
- *addAfter(v,new)*: add the node new just after the v node.

Exercise: Look for more information on the main differences between arrays (static and dynamic) and linked lists. Write an outline that brings out the main advantages and disadvantages for each data structure.

Singly Linked Lists

A **linked list** is a sequence of **nodes**. Each node is an object that stores a reference to an element and a reference to the following node in the list. This link to the next node is called *next*. The order of the list is represented by the next links.

The first node of a linked list is the **head** of the list. The last node of a linked list is the **tail** of the list. The next reference of the tail node points to null. A linked list defined in this way is known as a **singly linked list**.

Figure 1: Example of a singly linked list containing my favorite series orderby preference. Each node contains a reference to the name of a TV serie and a reference to the next node (the following TV serie). The next reference of the last node (tail) links to *null*.

How can you implement a singly linked list

Firstly, we implement a *Node* class as shown in Figure 2. This implementation uses the generic parameterized type ¡E¿, which allows to store elements of any specified class (that is, you will use the Node class to create objects containing *String, Integer, Long*, etc and any other class that you specified).

```java
/**
 * Example of Node class for a singly linked list of Objects.
 */
public class Node<E> {
    private E element;
    private Node<E> next;
    /**Constructor*/
    public Node(E e, Node<E> n) {
        element=e;
        next=n;
    }
    /**This method returns the element of this node*/
    public E getElement() {     return element;     }
    /**
     * This method returns the next node to this node
     * @return Node
     */
    public Node<E> getNext() {      return next;    }
    /**
     * Methods to modifier the properties of the Node class
     */
    public void setElement(E e) {       element=e;  }
    public void setNext(Node<E> n) {        next=n; }
}
```

Figure 2: Implementation of a Node of a singly linked list.

Figure 3 shows the partial implementation for a singly linked lis that only uses the reference to the head of the list (*head*), an instance variable to store the number of elements of the list (*size*) and a constructor method that sets the head node to null. For example, you may modify this class adding a new constructor method that has a node as input parameter and links the head of the list to this parameter.

How can you insert a new element in a singly linked list

The easiest case is when the new element is inserted at the head of the list. For example, I would like to add the TV serie 'Heidi' at the head of the above list. This TV series is my all time favorite serie, so it must be the first of the list (see 4). The following steps describe the process of insertion at the head of the list:

1. Create a new node. The name of the new serie and a reference to the same object as head (that is, next links to node that contains the serie *Losts*) must be passed to the constructor method.

2. Once you have created the node, you must set *head* (property of SinglyLinkedList) to point to new node.

```java
/**A singly linked list*/
public class SinglyLinkedList<E> {
    /**Head node of the list*/
    protected Node<E> head;
    protected int size;
    /**Constructor that sets the head node to a node class*/
    public SinglyLinkedList() {
        head=null;
        size=0;
    }
    public int getSize() {return size;};
    public boolean isEmpty() {return (head==null);};

    /**Search and update methods
     * ..... */
```

Figure 3: We define the head of the list as a Node. The constructor sets this node to null.

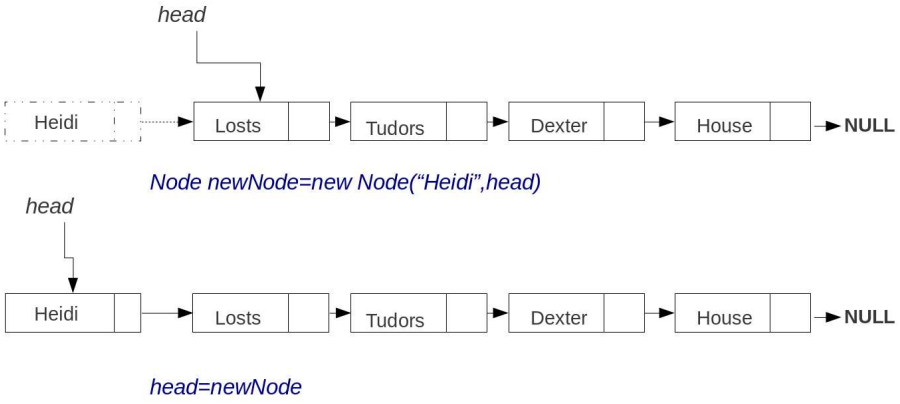

Figure 4: Insertion of an element at the head of a singly linked list.

Now, it is your turn. Please, write a new method called *insertHead* in the *SinglyLinkedList* class that inserts a new element at the head of the list. Figure **??** shows the implementation of this method.

To insert an element at the end of the list is very easy if you add a reference to the tail node, that is, an instance variable (with type Node¡E¿) to store the reference the last node in the list. For example, imagine that I like 'The Simpson', but I like it than less 'House', so I should insert it at the end of the list. I must follow the following steps:

1. Create a new node with the element 'The Simpson' and its next reference

```java
/**Inserts the node v at the beginning of the list*/
public void addFirst(Node<E> newHead) {
    /**This must points to the old head*/
    newHead.setNext(head);
    /**Now, we must make that the head points to the newHead*/
    head=newHead;
    /**Increase the size of the list*/
    size++;
}
```

Figure 5: This method implements the insertion operation of an element at the beginning of a singly linked list.

sets to null because this node will be the last node.

2. Then, the tail reference itself to this new node.

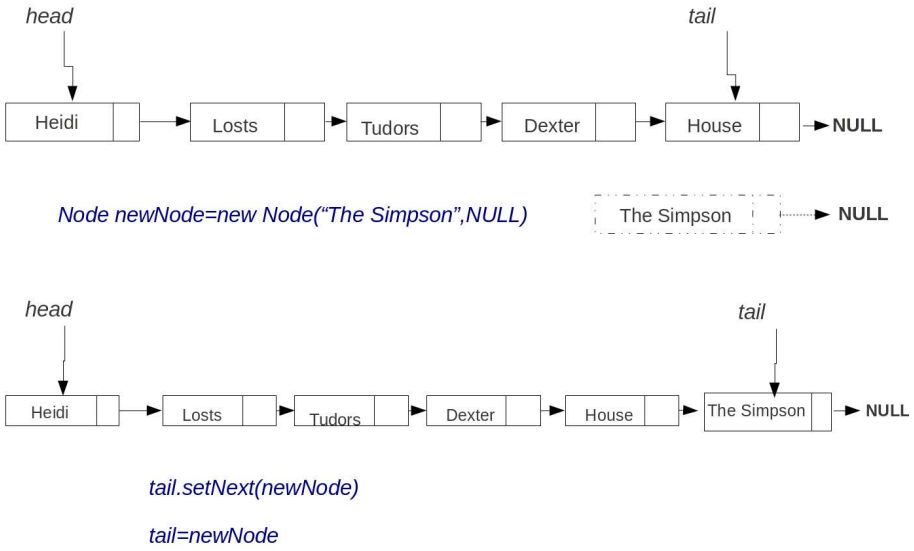

Figure 6: Insertion of an element at the end of a singly linked list. The class must have a property to store the tail of the list.

Figure 6 shows the above example. Please, try yourself defining the Node tail in the SinglyLinkedList class and adding the method *addLast*. You can find the solution in this new implementation in Figure ẽffig:addLastSList.

Now, take few minutes and think about how you can insert an element at the end of the list when you do not keep the tail reference in your implementation of the singly linked list. You should examine the list until you find node with a next

```java
/**A singly linked list*/
public class SinglyLinkedList<E> {
    /**Head and tail nodes of the list*/
    protected Node<E> head, tail;

    protected int size;

    public SinglyLinkedList() {
        head=tail=null;
        size=0;
    }

    /**Inserts the node newTail at the end of the list*/
    public void addLast(Node<E> newTail) {
        /**This must points to null by its reference next*/
        newTail.setNext(null);
        /**In order to link the list with the new node, the tail
         * points to the newHead by its reference next*/
        tail.setNext(newTail);
        /**Now, we must set the tail node to newTail*/
        tail=newTail;
        /**If the list was empty, the tail must points to the newHead*/
        if (head==null) head=newTail;
        /**Increase the size of the list*/
        size++;
    }

    /**Inserts the node newHead at the beginning of the list*/
    public void addFirst(Node<E> newHead) {
        /**This must points to the old head*/
        newHead.setNext(head);
        /**Now, we must make that the head points to the newHead*/
        head=newHead;
        /**If the list was empty, the tail must points to the newHead*/
        if (tail==null) tail=newHead;
        /**Increase the size of the list*/
        size++;
    }
```

Figure 7: Implementation of the insertion operation of an element at the end of a singly linked list.

reference to null. Do you dare to do it?. You can find a possible implementation in Figure 8.

How can you remove an element in a singly linked list

Now, let me show you how to remove at the head of the list. It is very easy!!!. You only need to set the head reference to its next node. This operation is illustrated in Figure 9 and its implementation is shown in Figure 10.

In order to remove the last node or an node at a give position of the list, we must access its previous node. Thus, the only way to find this previous node is to traverse the list from its beginning until to find it. Traversing the list may involve a big number of operations, taking a long time when the size of the list

```java
/**
 * This Method inserts a element at the end of the list.
 * Since we do not keep a reference to the last node of the list
 * (that is, an instance variable tail), we traverse the list until
 * finding a node whose next reference is null.
 */
public void insertTail(Node<E> newTail) {
    newTail.setNext(null);
    if (head==null) {
        /**The list is empty, we can insert the node
        at the beginning of the list*/
        addFirst(newTail);
    } else {
        Node<E> aux=head;
        /** The list is traversed from the beginning to the last node
         * using an auxiliary node that initially is set to head.*/
        while (aux.getNext()!=null) aux=aux.getNext();
        aux.setNext(newTail);

    }
    size++;
}
```

Figure 8: Implementation of the insertion operation of an element at the end of a singly linked list without keeping the tail reference.

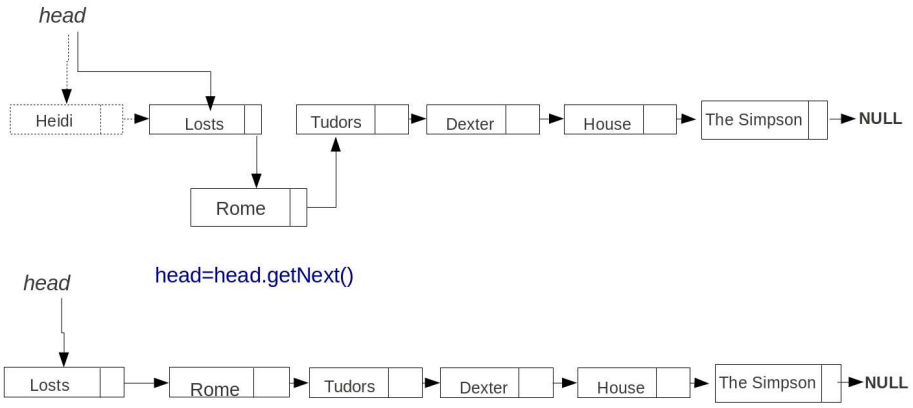

Figure 9: Removal of the head of the list.

is big. The following section presents an effective solution for this problem.

```java
/**Removes the head of the list.
 * If the list is empty an exception is throwing*/
public void removeFirst() throws EmptyListException {
    if (isEmpty()) throw new EmptyListException("List is empty");
    head=head.getNext();
    if (head==null) tail=null;
    size--;
}
```

Figure 10: This method removes the first element of the list.

Doubly Linked Lists

The main drawback of singly linked lists is that insering or removing a node at the middle or the end of a list, it is necessary to visit all nodes from its head until the node just before the place where you want to insert it or the node that you want to remove. This operation is time consuming because we do not have a quick access to the node before the one that you want to remove or the position where you want to insert.

Let me ask you the following question: *How can you implement a linked list to improve the access to nodes?*. Figure 11 gives you the key. This representation allows to traverse the list in both directions.

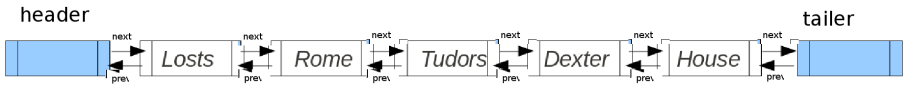

Figure 11: A doubly linked list storing my all time favorite TV series.

How can you define a node for a doubly linked list?. We need a mechanism that allow us to traverse the list from the beginning to the end and from the end to the beginning. The Node class (see Figure2) used in the implementation of a singly linked list only allowed us to go from left to right by the its instance variable *next*, which references to the following node in the list. Therefore, it would be very useful to define other instance variable to reference it previous node in the list. This variable is called *prev*. This implementation (see Figure 12) of a node for a doubly linked list makes easier to insert or remove an element at the end as well as in the middle of the list.

Now, we define the implementation of a doubly linked list. In order to facilitate the programming tasks, we can use two special nodes (called sentinels): *header* and *tailer*. The sentinel nodes do not store any reference to elements.

The *header* stores the node just before the head of the list (the header nodes points to the head of the list by its next reference; the prev reference of the head of the list must point to the header node). The *tailer* is the node after the tail of the list (the last element of the list must point to tailer by its next reference, while the tailer must poing to this last node by its prev reference) When the list is empty, the header and tailer nodes must point to each other.

Figure 15 describes the partial implementation of a doubly linked list. You can see how header and tailer sentinels are defined as objects of the above *DoublyNode* class. Also, a size property has been defined to store the number of elements at the list. We have defined a constructor that creates an empty list, that is, header and tailer sentinels are instantiated as *DoblyNode* objects (although, we do not give any value to their element property) and they point to each other.

Figures 14 and 13 show examples of removing and adding an element, respectively. As you can see in Figure **??** to insert a new element at the beginning

```java
package lists;

/**A class for nodes in a doubly linked list*/
public class DoublyNode<E> {
    protected E element;
    /**references to the next and previous nodes*/
    protected DoublyNode<E> next;
    protected DoublyNode<E> prev;
    /**Constructor*/
    public DoublyNode(E e, DoublyNode<E> p, DoublyNode<E> n) {
        element=e;
        prev=p;
        next=n;
    }
    /**This method returns the element of this node*/
    public E getElement() {return element;}
    /** This method returns a reference to its next node*/
    public DoublyNode<E> getNext() {return next;}
    /** This method returns a reference to its previous node*/
    public DoublyNode<E> getPrev() {return prev;}
    /**Methods set*/
    public void setElement(E e) {element=e;}
    public void setNext(DoublyNode<E> n) {next=n;}
    public void setPrev(DoublyNode<E> p) {prev=p;}
}
```

Figure 12: Implementation of a node for a doubly linked list.

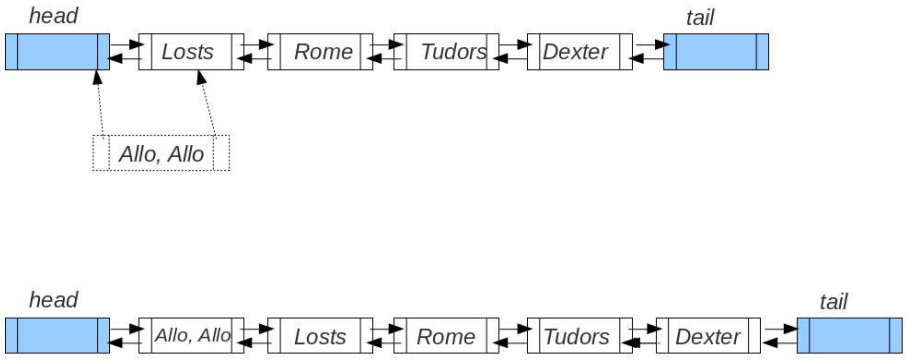

Figure 13: Inserting an element at the start of the list.

of the list or removing the last element of the list is very easy. Likewise, it is very easy to implement the methods *addLast()* and *removeFirst()*. *Do you dare*

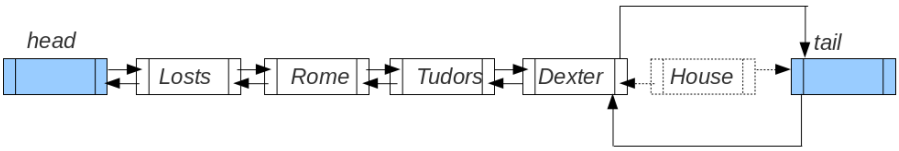

Figure 14: Removing an element from the end of the list.

```
public class DoublyLinkedList<E> {
    /**A sentinel node that is just before the head of the list. */
    protected DoublyNode<E> header;

    /**A sentinel node that is just after the last element of the list*/
    protected DoublyNode<E> tailer;

    protected int size;

    /**This constructor creates an empty list*/
    public DoublyLinkedList() {
        header=new DoublyNode<E>(null, null, null);
        tailer=new DoublyNode<E>(null, null, null);
        /**Sentinels must point to each other*/
        header.setNext(tailer);
        tailer.setPrev(header);
        size=0;
    }
    public int getSize() {      return size;    }
    public boolean isEmpty() { return (size==0);    }

    public void removeLast() throws EmptyListException {
        if (isEmpty()) throw new EmptyListException("List is empty");
        DoublyNode<E> lastNode=tailer.getPrev();
        DoublyNode<E> penultNode=lastNode.getPrev();
        tailer.setPrev(penultNode);
        penultNode.setNext(tailer);
        size--;
    }
    public void addFirst(DoublyNode<E> newNode){
        DoublyNode<E> oldNewNode=header.getNext();
        newNode.setNext(oldNewNode);
        newNode.setPrev(header);
        oldNewNode.setPrev(newNode);
        header.setNext(newNode);
        size++;
    }
```

Figure 15: Partial Implementation of a doubly linked list including the definition of the sentinel nodes *header* and *tailer*, the constructor method and the *addFirst()* and *removeLast()* methods.

to implement them?. You can find the implementation of these operation in Figure 16

```
public void addLast(DoublyNode<E> newNode) {
    DoublyNode<E> oldLastNode=tailer.getPrev();
    oldLastNode.setNext(newNode);
    newNode.setPrev(oldLastNode);
    newNode.setNext(tailer);
    tailer.setPrev(newNode);
    size++;
}


public void removeFirst() throws EmptyListException {
    if (isEmpty()) throw new EmptyListException("List is empty");
    DoublyNode<E> firstNode=header.getNext();
    DoublyNode<E> new1stNode=firstNode.getPrev();
    new1stNode.setPrev(header);
    header.setNext(new1stNode);
    size--;
}
```

Figure 16: Methods *addLast()* and *removeFirst()* of a doubly linked list.

How can you insert an element in the middle of a doubly linked list?

Doubly linked lists allow an efficient manner to access and modify their elements since they provide an easier way to insert and remove in the middle of the list than single linked lists. Figure 17 shows an example of a insertion in the middle of a doubly linked list. I have just watched the TV serie 'Bones'. I like more than 'Losts', but less than 'Allo, Allo", so I should add it just after the 'Allo, Allo' node. Firstly, I must define a new node with the element 'Bones', its next node must point to the node 'Losts' and its prev node to the node 'Allo, Allo'. Then, the nodes 'Allo, Allo' and 'Losts' must point to the node 'Bones' by their next and prev nodes respectively (see Figure 17). Now, imagine that I no longer like the 'Losts' serie. To remove it, their before node (that is, 'Allo, Allo') and after node ('Rome') must point to each other by their next and prev nodes respectively. Figure 18 includes the java code of the methods for inserting and removing elements in the middle of a doubly linked list. A full implementation of a doubly linked list can be found in the following link.

All methods in the implementation of a list using a doubly linked list take O(1). For a list of n elements, the space used is O(n).

Stacks

A **stack** is a collection of objects that are added and removed according to the the **Last-In First-out (LIFO)** principle. To understand better this principle, think about a stack of plates (Figure 19), *how do you add and take off plates from the stack?*. Normally, you add plates to the top of the stack and you take

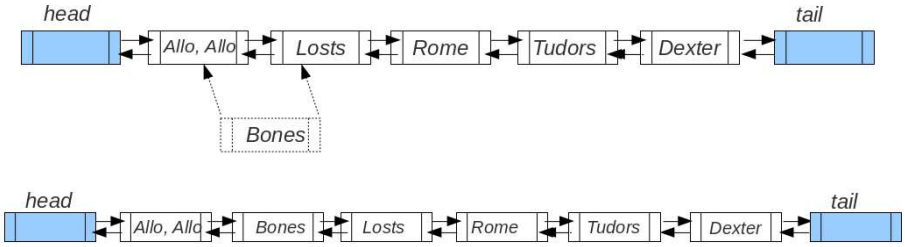

Figure 17: To add a new node after the .

```
/**Insert the node newNode before the v node*/
public void addBefore(DoublyNode<E> v, DoublyNode<E> newNode) throws IllegalStateException {
    DoublyNode<E> aux=getPrev(v);
    newNode.setPrev(aux);
    aux.setNext(newNode);
    newNode.setNext(v);
    v.setPrev(newNode);
    size++;
}

/**Insert the node vNode after the v node*/
public void addAfter(DoublyNode<E> v, DoublyNode<E> newNode) throws IllegalStateException {
    DoublyNode<E> aux=getNext(v);
    newNode.setNext(aux);
    aux.setPrev(newNode);
    newNode.setPrev(v);
    v.setNext(newNode);
    size++;
}

public void remove(DoublyNode<E> v) throws IllegalStateException {
    DoublyNode<E> sig=getNext(v);
    DoublyNode<E> ant=getPrev(v);
    ant.setNext(sig);
    sig.setPrev(ant);
    size--;
}
```

Figure 18: Methods addBefore, addAfter, remove for a doubly linked list.

off them from the top of the stack. This is just the *LIFO* principle.

Formally, a *stack* is an abstract data structure that is characterized by the following operations:

- *push(e)*: add the element e to the top of the stack.
- *pop()*: remove the top element from the stack and return it.

Other additionally operations are:

- *size()*: return the size of the stack.
- *isEmtpy()*: return true if the stack is empty; false eoc.
- *top()*: return the top element in the stack, without removing it.

Figure 19: Stack of plates.

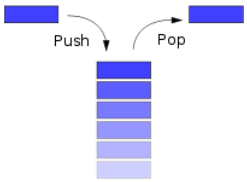

Figure 20: Push and pop operations.

Operation	**Stack**	**Output**
push('h')	(h)	-
push('e')	(h,e)	-
top()	(h,e)	e
push('l')	(h,e,l)	-
push('l')	(h,e,l,l)	-
push('o')	(h,e,l,l,o)	-
top()	(h,e,l,l,o)	o
push('!')	(h,e,l,l,o,!)	-
top()	(h,e,l,l,o,!)	!
size()	(h,e,l,l,o,!)	6
isEmpty()	(h,e,l,l,o,!)	false
pop()	(h,e,l,l,o)	!

Table 1: This table shows a secuence of operations on a stack of characters

This data structure is very useful for many applications which require to store the secuence of operations in order to reverse or undo them. For example, web browsers store the urls recently visited on a stack in order to allow users to visit the previously urls by pushing the back button. Likewise, stacks can be used to provide an *undo* mechanism to the text editors.

The *java.util* package already includes an implementation of the stack data structure due to its importance. It is recommended to use the *java.util.Stack* class, however in this section we design and implement ourselves a stack. First of all, we define an **interface** to declare the methods of the data structure (see Figure 21). You can note that this interface has been defined using the generic parameterized type E, which allows to store elements of any specified class. Also, we have defined the *EmptyStackException* class that will throw an exception when the methods *pop()* and *top()* are called on an empty stack.

```
package lists;
/** Interface for a stack: collection of elements that are inserted
 * and removed based on the LIFO principle.
 */
public interface Stack<E> {
    /**Returns the size of the stack*/
    public int size();

    /**Returns true if the stack is empty, otherwise false*/
    public boolean isEmpty();

    /**Returns the top element of the stack and remove it.
     * Throws an exception if the stack is empty. */
    public E pop() throws EmptyStackException;

    /**Returns the top element of the stack without remove it.
    * Throws an exception if the stack is empty. */
    public E top() throws EmptyStackException;

    /**Add the element e at the top of the stack*/
    public void push(E e);

}
```

Figure 21: Interface Stack. This interface uses the generic parameterized type E to contain elements of any specified class in the stack.

There are several ways to implement the Stack class. A simple way to represent a stack is to store its elements into an array. Thus, the stack consists of an array and an integer variable to indicate the index of the top element. Figure 23 shows a java class implementing an array-based stack. This class implements the interface Stack<E>. This implementation is based on the use of an array (for storing the elements of a stack. The instance variable *top* stores the index in which the top of the stack is stored in the array. Also, the maximum size of the array is defined in a constant *MAXCAPACITY*. Another instance variable (*capacity*) stores the actual capacity of the stack. The main drawback of this implementation is that it is necessary to initially know the maximum size of the stack. Thus, we have defined the *FullStackException* class that will throw an exception when the *push* method is called on a full stack (see Figure 22)

Now, you must write the code to build a stack containing the operations shown in Table 1. Table 2 summarizes the computational complexity for each

```java
package lists;
/**
 * This Runtime exception will be thrown when the methods
 * top and pop are performed on an empty stack.
 */
public class EmptyStackException extends RuntimeException {

    public EmptyStackException(String message) {
        super(message);
    }
}
```

Figure 22: An exception is thrown when the methods pop and top are performed on an empty stack.

method of the array-based implementation of a stack. The methods *size()* and *isEmpty* take O(1) time because they only access the instance variable *top*. The methods *top()* and *pop()* take constant time because they call the method *isEmpty()* and access the top element in the array (the method *pop()* also decreases the instance variable top). All of the previous operations take O(1). Likewise the method *push()* also takes O(1).

Methods	**Time**
isEmtpy(), size()	O(1)
top(), pop()	O(1)
push()	O(1)

Table 2: Performance of an array-based implementation of a stack

Therefore, the array-based implementation is simple and efficient. However, the main drawback of this implementation is that it is necessary to know the maximum size of the stack. In some cases, this may cause an unnecessary waste of memory or an exception when the stack reaches this maximum size and it is not possible to add store elements.

Another implementation that does not have the size limitation is to use a linked list to represent a stack. That is, the elements of a stack are stored into the nodes of a linked list. Of course, we may use the *java.util.ArrayList* class or any API java class implementing lists to represent the stack, however, we provide ourselves implementation of a linked list in order to improve your knowledge and practice about linked lists. First of all, we define a java class to implement a generic node for a singly linked list (see Figure 25). Figure 26 shows the code of the linked list-based implementation. This class defines the instace variable *top* that stores the top element of the stack. We have decided that the top of the

```java
package lists;
/**An array-based implementation to represent a stack * */
public class ArrayStack<E> implements Stack<E> {
    /**This constant is the maximum possible size of the array*/
    public final static int MAXCAPACITY=1000;

    /**Array for storing the elements of the stack*/
    protected E stack[];
    /**index for the top element in the stack*/
    protected int top=-1;
    /**stores the actual capacity (number of elements) of the stack*/
    protected int capacity=-1;

    /**Constructor. Initially, it creates an array of a given size*/
    public ArrayStack(int s) {
        capacity=s;
        stack=(E[]) new Object[s];
    }

    /**Returns the size of the stack*/
    public int size() { return (top+1); }
    /**Returns true if the stack is empty, eoc false*/
    public boolean isEmpty() { return (top<0);  }

    /**Returns the top element of the stack and remove it.
     * Throws an exception if the stack is empty. */
    public E pop() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException("Stack is empty");
        E temp=stack[top];
        stack[top]=null;
        top--;
        return temp;
    }

    /**Returns the top element of the stack without remove it.
    * Throws an exception if the stack is empty. */
    public E top() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException("Stack is empty");
        return stack[top];
    }

    /**Add the element e at the top of the stack*/
    public void push(E e) throws FullStackException {
        if (size()==capacity) throw new FullStackException("Stack is full");
        top++;
        stack[top]=e;
    }

}
```

Figure 23: An array-based implementation of a stack

stack is stored at the head of the linked list. This fact allows to the operations *pop(), top()* and *push()* take constant time (see Table 3) because they only need to access the first element (head) of the list. If the top element of the stack was stored at the end of the list, then it would be necessary to trasverse all elements of the list, every time you would need to access the top element. Thus, the previous methods would take O(n) time. We also note that the method *push()* does not throw an exception related to the size overflow problem since in this implementation the size of the stack is not limited.

```java
package lists;
/**
 * This Runtime exception will be thrown when the methods
 * push is called on stack that is full.
 */
public class FullStackException extends RuntimeException {

    public FullStackException(String message) {
        super(message);
    }
}
```

Figure 24: An exception is thrown when the method push is performed on a full stack.

```java
/**
 * Example of Node class for a singly linked list of Objects.
 */
public class Node<E> {
    private E element;
    private Node<E> next;
    /**Constructor*/
    public Node(E e, Node<E> n) {
        element=e;
        next=n;
    }
    /**This method returns the element of this node*/
    public E getElement() {     return element;     }
    /**
     * This method returns the next node to this node
     * @return Node
     */
    public Node<E> getNext() {      return next;    }
    /**
     * Methods to modifier the properties of the Node class
     */
    public void setElement(E e) {       element=e;  }
    public void setNext(Node<E> n) {        next=n; }
}
```

Figure 25: A java class for implementing a node of a generic singly linked list.

Stacks have many interesting applications. You can find some applications such as reversing arrays or matching parentheses and HTML tags in in Chapter 5 (Stacks and Queues) (pages 199-203) in the book [1].

```java
package lists;
/**A linked list-based implementation of a stack.
 * The top element of stack is stored at the beginning of the list*/
public class LinkedListStack<E> implements Stack<E> {
    /**This variable stores the reference to the top element of the stack*/
    protected Node<E> top;
    /**Stores the number of elements of the stack*/
    protected int size=-1;

    /**Constructor. It creates an empty stack*/
    public LinkedListStack() {
        size=0;
        top=null;
    }
    /**Returns the size of the stack*/
    public int size() { return (size); }
    /**Returns true if the stack is empty, eoc false*/
    public boolean isEmpty() { return (size<=0);    }
    /**Returns the top element of the stack and remove it.
     * Throws an exception if the stack is empty. */
    public E pop() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException("Stack is empty");
        E temp=top.getElement();
        top=top.getNext();
        size--;
        return temp;
    }
    /**Returns the top element of the stack without remove it.
    * Throws an exception if the stack is empty. */
    public E top() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException("Stack is empty");
        return top.getElement();
    }

    /**Add the element e at the top of the stack*/
    public void push(E e) {
        Node<E> newNode=new Node<E>(e,top);
        top=newNode;
        size++;
    }
```

Figure 26: A Linked-list based implementation of a stack.

Queues

Another important linear data structure is the queue. A **queue** is a collection of objects that are managed according to the the **First-In First-out (FIFO)** principle (see Figure 27), that is, only element at the front of the queue can be accessed or deleted and new elements must added at the end (*rear*) of the queue. In order to understand better this principle, think about a line of people

Methods	Time
isEmtpy(), size()	O(1)
top(), pop()	O(1)
push()	O(1)

Table 3: Running times of the array-based implementation of a stack

waiting a bus. Normally, the first person in the line will be the first one on getting onto the bus. If one arrives last, this should put oneself at the rear of the line. Queues are a nature option of many applications that requiere to process their requests according to FIFO principle, such as reservation systems for airlines, cinemas or many other public services.

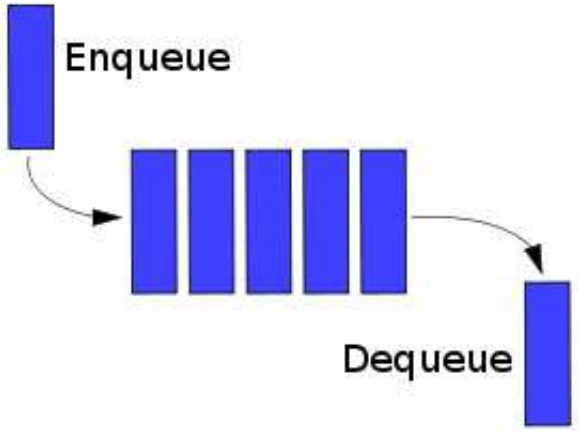

Figure 27: Representation of a FIFO Queue.

Formally, a *queue* is an abstract data structure that is characterized by the following operations:

- *enqueue()*: add a element at the rear (end) of the queue.
- *dequeue()*: return and remove the first element of the queue, that is, the element at the front of the queue. If this one is empty, then this method should throw an exception.
- *front()*: return the element at the front of the queue. If this one is empty, then this method should throw an exception.

In addition, similar to the *Stack* ADT, the queue ADT can also include the *size()* and *isEmpty()* methods.

Figure 28 shows a java interface for this ADT. It uses the generic parameterized type E, which allows to store elements of any specified class. Also, the

Operation	Queue	Output
enqueue('h')	(h)	-
enqueue('e')	(h,e)	-
front()	(h,e)	h
dequeue()	(e)	h
dequeue()	()	e
enqueue('h')	(h)	-
enqueue('o')	(h,o)	-
enqueue('l')	(h,o,l)	-
front()	(h,o,l)	h
enqueue('l')	(h,o,l,l)	-
front()	(h,o,l,l)	h
enqueue('a')	(h,o,l,l,a)	-
size	(h,o,l,l,a)	6
enqueue('!')	(h,o,l,l,a,!)	-
size	(h,o,l,l,a,!)	7
front()	(h,o,l,l,a,!)	h
dequeue()	(o,l,l,a,!)	h

Table 4: A secuence of operations on a queue of characters

EmptyQueueException class has been defined to throw an exception when the methods *dequeue()* and *front()* are called on an empty queue.

A circular array-based implementation of a queue

Likewise with the *Stack* ADT, we can use an array to represent a queue. Thus, elements of a queue are stored in an array. *What is the more efficient option for storing the front of the queue*:

1. at the first position of the array (that is, Array[0]) and adding the following elements from there.

2. as the last element of the array, that is, a new element is always inserted at the first position of the array and the

The former one is not an efficient solution because each time the method *dequeue()* is called, all elements must moved to its previous cell, taking O(n) time. The second one is also an inefficient solution since each time a new element will be inserted (*enqueue()*), elements in the array must be moved to their following position, taking O(n) time.

In order to achieve constant time for the methods of the *Queue* interface, we can use a circular array to store the elements and two instance variables *front* and *rear* to keep the index storing the first element of the queue and the index to store a new element. Each time we remove the first element of the queue, we should increase the variable *front*. Likewise, each time we add a new element,

```java
package lists;
/** Interface for a queue: collection of elements that are managed
 * based on the FIFO principle:first in- first out
 */
public interface Queue<E> {
    /**Returns its size */
    public int size();

    /**Returns true if the queue is empty, otherwise false*/
    public boolean isEmpty();

    /**Returns the element at the front of the queue and remove it.
     * Throws an exception if the queue is empty. */
    public E dequeue() throws EmptyQueueException;

    /**Returns the element at the front of the queue without remove it.
    * Throws an exception if the queue is empty. */
    public E front() throws EmptyQueueException;

    /**Add the element e at the rear of the queue*/
    public void enqueue(E e);

}
```

Figure 28: A java interface for the *Queue* ADT.

```java
package lists;

public class EmptyQueueException extends RuntimeException {
    public EmptyQueueException(String message) {
        super(message);
    }
}
```

Figure 29: An exception is thrown when the methods *dequeue()* and *front()* are performed on an empty queue.

we store it into the position *rear* at the array and increase the value of this variable.

Figure 30 shows three different configuration of a queue implemented using a circular array. The first case ($front \leq rear \leq length(array)$) is the normal configuration. The second and thirds examples illustrate the configuration in which the rear reaches the lenght of the array and it is necessary to store a new element at the first position of the array. When *rear* reaches *front*, it implies that the queue is empty. Each time we need to increase the *rear* or *front*, we must estimate their the module value

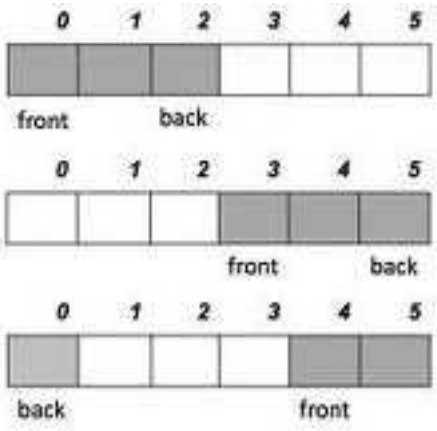

Figure 30: Three different configurations of a queue.

Each method in this implementation takes O(1) since they only involve a constant number of arithmetic opreations, comparisons and assignments. The only drawback of this implementation is that the size of the queueu is limited to the size of the array. However, if we are able to provide a good estimation of the size of the queue, this implementation is very efficient. Figure 31 shows a circular array-based implementation of a queue.

A linked list-based implementation of a queue

A linked list also provides an efficient implementation of a queue (see Figure 32). For efficiency reasons, the front of the queue is stored at the first node of the list and we also define a variable to store the tail of the list. These two variables allow all methods take O(1) time because they only need a constant number of simple statements. The main advantage of this implementation compared to the array-based implementation is that it is no necessary to specify a maximum size for the queue.

Please, write a java program to assign the turn to every journalist in the TV program '59 seconds'. You can find an interesting problem based on the use of a queue in in Chapter 5 (Stacks and Queues) (pages 212) in the book [1].

Double-Ended Queues (Dqueues)

Figure 33 compares the three ADT: stack (LIFO: last in, first out), queue (FIFO: first in, first out) and dqueue. A *double-ended queue* ADT (dqueue is pronunced like deck) is power than stack and queue because it supports insertion and deletion at both its front and its rear. The main methods of the dqueue ADT are:

- *addFirst(e)*: insert a new element at the head of the queue.

```
package lists;
public class ArrayQueue<E> implements Queue<E> {
    protected E queue[];
    protected int front=-1;
    protected int rear=-1;
    public ArrayQueue(int maxElem){
        queue=(E[]) new Object[maxElem];
    }
    public int size() {return (queue.length - front + rear) % queue.length; }
    public boolean isEmpty() {      return (front==rear);   }
    public E dequeue() throws EmptyQueueException {
        if (front == rear)
            throw new EmptyQueueException ("Queue is empty");
        E temp=queue [front];
        front = (front + 1) % queue.length;
        return temp;
    }
    public E front() throws EmptyQueueException {
        if (front == rear)
            throw new EmptyQueueException ("Queue is empty");
        return queue [front];
    }
    public void enqueue(E e) throws FullQueueException  {
        if (size()== queue.length-1) throw new FullQueueException("Queue is full");
        queue [rear] = e;
        rear = (rear + 1) % queue.length;
    }
}
```

Figure 31: A circular array-based implementation of a queue.

- *addLast(e)*: insert a new element at the end of the queue.
- *removeFirst()*: remove the element at the front of the queue. If the queue is empty, then an exception is thrown.
- *removeLast()*: remove the element at the end of the queue. If the queue is empty, then an exception is thrown.
- *getFirst()*: return the element at the head of the queue. If the queue is empty, then an exception is thrown.
- *getLast()*: return the element at the end of the queue. If the queue is empty, then an exception is thrown.
- size(): return the size of the queue.
- isEmpth(): return a boolean indicating if the queue is empty.

The *java.util.LinkedList¡E¿* class already defines all the methods of a dequeue. Of course, you can use this class when you need to use a dequeue in your future applications. But first, you must learn how the dequeue ADT can be defined and implemented. Figure 34 contains a java interface for the dqueue

```java
public class LinkedListQueue<E> implements Queue<E> {
    /**This variables point to the head and tail of the queue.
     * For efficiency reasons, the front is stored into the first node*/
    protected Node<E> front, tail;
    protected int size=-1;
    public LinkedListQueue() {
        size=0;
        front=tail=null;
    }
    public E dequeue() throws EmptyQueueException {
        if (isEmpty()) throw new EmptyQueueException("Queue is empty");
        E temp=front.getElement();
        front=front.getNext();
        size--;
        if (size==0) tail=null;
        return temp;
    }
    public void enqueue(E e) {
        Node<E> newNodo=new Node(e,null);
        if (isEmpty()) front=newNodo;
        else tail.setNext(newNodo);
        tail=newNodo;
        size++;
    }
```

Figure 32: A Linked List-based implementation of a queue.

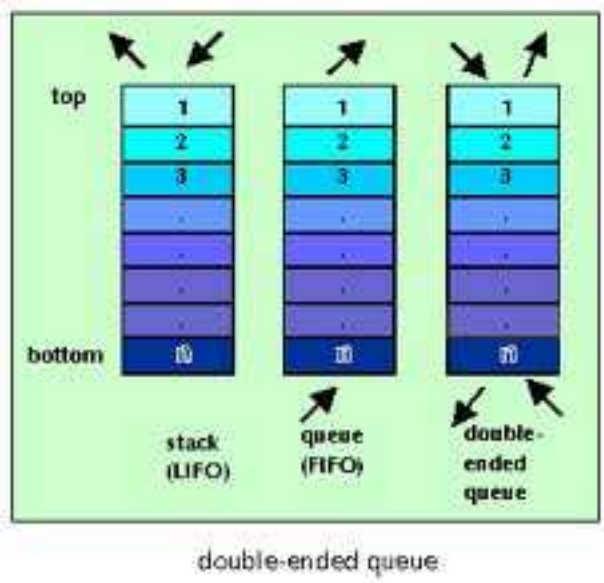

Figure 33: The dequeu ADT a queue that allows invertion and deletion at both its front and its rear (source: http://t3.gstatic.com/.

Operation	Queue	Output
addFirst('k')	(k)	-
addLast('l')	(k,l)	-
removeFirst()	(l)	-
addLast('o')	(l,o)	-
addFirst('a')	(a,l,o)	-
removeFirst()	(l,o)	-
removeLast()	(o)	-
removeLast()	empty	-
removeFirst()	empty	exception
isEmpty()	empty	true
addLast('s')	(s)	-
isEmpty()	(s)	false

Table 5: Secuence of operations on a dqueue of characters

ADT. A dequeu is a list of elements, hence we may use a linked list to implement a dqueue. A singly linked list is not an efficient solution because the dqueue allows insertion and removal at both the head and the tail of the list. While the insertion or removal of the first element at the dqueue just take O(1), the insertion or removal of the last element take O(n) because we should trasverse all nodes until to reach the last node. However, if we implement the dqueue ADT using a doubly linked list, all insertion and removal operations take O(1) times.

Figure 36 shows a fragment of the implementation of a deque using a doubly linked list. This class defines two sentinal nodes to reference the head and tail of the deque. Figure 35 shows the implementation for a node of a doubly linked list.

```java
package lists;
/**Interface for a double-ended queue.
 * The java.util.LinkedList class implements these methods*/
public interface Dqueue<E> {
    /**Returns its size */
    public int size();

    /**Returns true if the dqueue is empty, otherwise false*/
    public boolean isEmpty();

    /**Inserts an element at the beginning of the dqueue.*/
    public void insertFirst(E e);

    /**Inserts an element at the end of the dqueue.*/
    public void insertLast(E e);

    /**Returns the element at the front of the dqueue.
     * Throws an exception if the dqueue is empty. */
    public E removeFirst() throws EmptyDqueueException;

    /**Returns the element at the end of the dqueue.
     * Throws an exception if the dqueue is empty. */
    public E removeLast() throws EmptyDqueueException;

    /**Returns the element at the beginning of the dequeu.
    * Throws an exception if the dqueue is empty. */
    public E getFirst() throws EmptyDqueueException;

    /**Returns the element at the end of the dequeu.
    * Throws an exception if the dqueue is empty. */
    public E getLast() throws EmptyDqueueException;
```

Figure 34: An interface for a double-ended queue ADT.

```java
package lists;

/**A class for nodes in a doubly linked list*/
public class DoublyNode<E> {
    protected E element;
    /**references to the next and previous nodes*/
    protected DoublyNode<E> next;
    protected DoublyNode<E> prev;
    /**Constructor*/
    public DoublyNode(E e, DoublyNode<E> p, DoublyNode<E> n) {
        element=e;
        prev=p;
        next=n;
    }
    /**This method returns the element of this node*/
    public E getElement() {return element;}
    /** This method returns a reference to its next node*/
    public DoublyNode<E> getNext() {return next;}
    /** This method returns a reference to its previous node*/
    public DoublyNode<E> getPrev() {return prev;}
    /**Methods set*/
    public void setElement(E e) {element=e;}
    public void setNext(DoublyNode<E> n) {next=n;}
    public void setPrev(DoublyNode<E> p) {prev=p;}
}
```

Figure 35: This class implements a node of a doubly linked list.

```java
package lists;
/**A doubly-linked list based implementation of a deque*/
public class DoublyLListDeque<E> implements Dqueue<E> {
    /**We define two sentinal nodes to reference the head and tail of the list */
    protected DoublyNode<E> header, tailer;
    protected int size;
    /**Constructor*/
    public DoublyLListDeque() {
        header=new DoublyNode<E>();
        tailer=new DoublyNode<E>();
        /**header and tailer must point to  each other*/
        header.setNext(tailer);
        tailer.setPrev(header);
        size=0;
    }
    /**Returns the last element at the deque.*/
    public E getLast() throws EmptyDqueueException {
        if (isEmpty()) throw new EmptyDqueueException("Deque is empty");
        /**tailer points to the last node of the deque by its reference prev*/
        return tailer.getPrev().getElement();
    }
    /**Removes the first element at the deque.*/
    public E removeFirst() throws EmptyDqueueException {
        if (isEmpty()) throw new EmptyDqueueException("Deque is empty");
        /**header points to the first node of the deque by its reference next*/
        DoublyNode<E> firstNode=header.getNext();
        E temp=firstNode.getElement();
        /**The current second node will be the first node*/
        DoublyNode<E> secondNode=firstNode.getNext();
        header.setNext(secondNode);
        secondNode.setPrev(header);
        size--;
        return temp;
    }
    /**Inserts the element e at the end of the deque*/
    public void insertLast(E e) {
        DoublyNode<E> oldLastNode=tailer.getPrev();
        DoublyNode<E> newLastNode=new DoublyNode<E>(e,oldLastNode,tailer);
        oldLastNode.setNext(newLastNode);
        tailer.setPrev(newLastNode);
        size++;
    }
```

Figure 36: A doubly linked list class for implementing a deque.

Bibliography

[1] M. Goodrich and R. Tamassia, *Data structures and algorithms in Java*. Wiley-India, 2009.

[2] M. Weiss, *Data structures and problem solving using Java*. Addison Wesley Publishing Company, 2002.