LINKED LISTS

LINKED LISTS

LINKED LISTS ARE **LIST DATA STRUCTURES** - THEY CAN STORE MULTIPLE ELEMENTS AS A LIST

EACH ELEMENT IS **LINKED TO OR REFERENCES THE NEXT ELEMENT** - THE ELEMENTS ARE CHAINED TOGETHER

THIS DATA STRUCTURE IS NOT VERY COMMONLY USED IN JAVA AS JAVA HAS AN AWESOME LIBRARY OF LISTS WHICH CAN BE USED OUT OF THE BOX

LINKED LISTS

LINKED LISTS ARE **LIST DATA STRUCTURES** - THEY CAN STORE MULTIPLE ELEMENTS AS A LIST

EACH ELEMENT IS **LINKED TO OR POINTS TO THE NEXT ELEMENT** - THE ELEMENTS ARE CHAINED TOGETHER

ARRAYLIST IS THE MOST COMMON BUT **LINKEDLIST** IS ALSO AVAILABLE TO USE AS PART OF THE STANDARD JAVA LIBRARIES

LINKED LISTS

LINKED LISTS ARE **LIST DATA STRUCTURES** - THEY CAN STORE MULTIPLE ELEMENTS AS A LIST

EACH ELEMENT IS **LINKED TO OR POINTS TO THE NEXT ELEMENT** - THE ELEMENTS ARE CHAINED TOGETHER

ARRAYLIST IS THE MOST COMMON BUT **LINKEDLIST** IS ALSO AVAILABLE TO USE AS PART OF THE STANDARD JAVA LIBRARIES

THE WAY TO ACCESS LINKED LIST IS VIA THE VERY FIRST ELEMENT IN THE LIST CALLED THE "**HEAD**"

LINKED LISTS

LINKED LISTS ARE **LIST DATA STRUCTURES** - THEY CAN STORE MULTIPLE ELEMENTS AS A LIST

EACH ELEMENT IS **LINKED TO OR POINTS TO THE NEXT ELEMENT** - THE ELEMENTS ARE CHAINED TOGETHER

ARRAYLIST IS THE MOST COMMON BUT **LINKEDLIST** IS ALSO AVAILABLE TO USE AS PART OF THE STANDARD JAVA LIBRARIES

THE WAY TO ACCESS LINKED LIST IS VIA THE VERY FIRST ELEMENT IN THE LIST CALLED THE "**HEAD**"

EACH ELEMENT REFERENCES TO THE NEXT ELEMENT IN THE CHAIN - THE **LAST ELEMENT POINTS TO NULL**

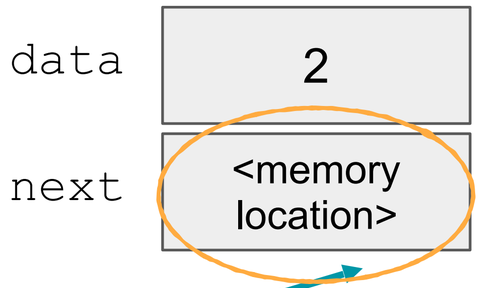
LINKED LISTS

LINKED LISTS IN JAVA ARE PRETTY STRAIGHTFORWARD - IN C THEY CAN BE PRETTY COMPLEX AND ARE INTERVIEW FAVORITES

HOWEVER, AS THE SIMPLEST AND MOST BASIC DATA STRUCTURE IT'S WORTH OUR WHILE TO DO A BRIEF STUDY OF HOW LINKED LISTS WORK

LINKED LISTS

INFORMATION HELD IN EACH ELEMENT OF A LINKED LIST, HERE ASSUME IT'S AN INTEGER



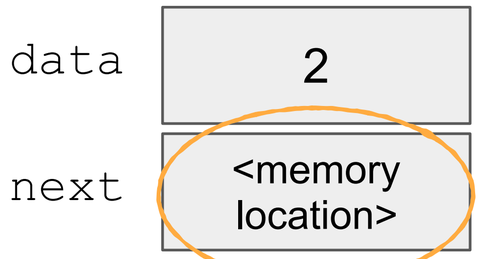
THIS IS JUST A REFERENCE IN JAVA - A REFERENCE TO THE NEXT NODE

POINTER TO THE NEXT ELEMENT IN THE LIST - THE LAST ELEMENT WILL POINT TO NULL

THIS INFORMATION DATA + NEXT REFERENCE CAN BE CONSIDERED TO BE A SINGLE NODE

LINKED LISTS

INFORMATION HELD IN EACH ELEMENT OF A LINKED LIST, HERE ASSUME IT'S AN INTEGER



THIS IS JUST A REFERENCE IN JAVA - A REFERENCE TO THE NEXT NODE

POINTER TO THE NEXT ELEMENT IN THE LIST - THE LAST ELEMENT WILL POINT TO NULL

THIS NODE IS IMPLEMENTED AS A **CLASS IN JAVA** - A **GENERIC CLASS** IS PREFERRED SO IT CAN HOLD DATA OF ANY TYPE

LINKED LISTS

```java
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;

    public Node(T data) {
        this.data = data;
        setNext(null);
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> next) {
        this.next = next;
    }

    public T getData() {
        return data;
    }

    @Override
    public String toString() {
        return String.valueOf(data);
    }

}
```

LINKED LISTS

```java
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;

    public Node(T data) {
        this.data = data;
        setNext(null);
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> next) {
        this.next = next;
    }

    public T getData() {
        return data;
    }

    @Override
    public String toString() {
        return String.valueOf(data);
    }

}
```

A GENERIC CLASS WHICH CAN HOLD DATA OF ANY TYPE

THE DATA SHOULD BE COMPARABLE - THIS IS FOR EQUALITY CHECKS SO YOU CAN DO THINGS LIKE FIND WHAT INDEX A PARTICULAR ELEMENT IS LOCATED AT IN A LIST

LINKED LISTS

```java
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;

    public Node(T data) {
        this.data = data;
        setNext(null);
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> next) {
        this.next = next;
    }

    public T getData() {
        return data;
    }

    @Override
    public String toString() {
        return String.valueOf(data);
    }

}
```

THE INFORMATION WITHIN ANY NODE IS THE ACTUAL **DATA** AS WELL AS A **REFERENCE** TO THE NEXT NODE

LINKED LISTS

```java
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;

    public Node(T data) {
        this.data = data;
        setNext(null);
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> next) {
        this.next = next;
    }

    public T getData() {
        return data;
    }

    @Override
    public String toString() {
        return String.valueOf(data);
    }

}
```

THE CONSTRUCTOR TAKES IN THE DATA ASSOCIATED WITH THIS NODE

THE NEXT REFERENCE IS SET TO NULL INITIALLY - IT CAN BE SET SEPARATELY LATER

LINKED LISTS

```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;

    public Node(T data) {
        this.data = data;
        setNext(null);
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> next) {
        this.next = next;
    }

    public T getData() {
        return data;
    }

    @Override
    public String toString() {
        return String.valueOf(data);
    }

}
```

SIMPLE GETTERS AND SETTERS FOR THE DATA AND THE NEXT REFERENCE

LINKED LISTS

```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;

    public Node(T data) {
        this.data = data;
        setNext(null);
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> next) {
        this.next = next;
    }

    public T getData() {
        return data;
    }

    @Override
    public String toString() {
        return String.valueOf(data);
    }

}
```

THE STRING REPRESENTATION OF THE NODE IS SIMPLY THE STRING REPRESENTATION OF THE DATA STORED IN THE NODE

LINKED LISTS

```
public class LinkedList<T extends Comparable<T>> implements Cloneable {

    private Node<T> head = null;

    public LinkedList() {
    }

}
```

A LINKED LIST CLASS HOLDS THE HEAD OF THE LINKED LIST - THIS CAN HOLD ALL THE METHODS WHICH OPERATE ON THE LIST

LINKED LISTS

```
public class LinkedList<T extends Comparable<T>> implements Cloneable {

    private Node<T> head = null;

    public LinkedList() {
    }

}
```

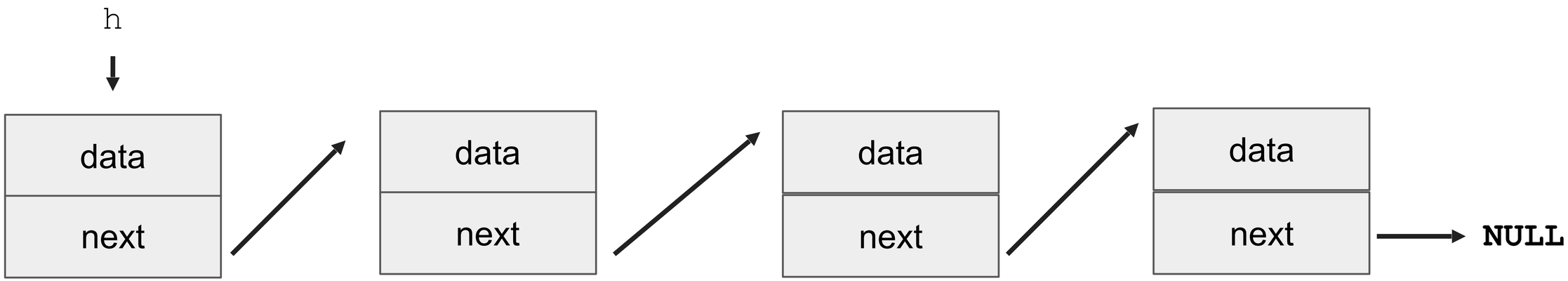
A LINKED LIST CLASS HOLDS THE HEAD OF THE LINKED LIST - THIS CAN HOLD ALL THE METHODS WHICH OPERATE ON THE LIST

LINKED LISTS

```
public class LinkedList<T extends Comparable<T>> implements Cloneable {

    private Node<T> head = null;

    public LinkedList() {
    }

}
```

INITIALLY THE HEAD WILL BE NULL - AS YOU CREATE AND ADD NODES TO THE LIST THE HEAD WILL REFERENCE THE FIRST ELEMENT IN THE LIST

HOW DO LINKED LISTS LOOK IN MEMORY?




```
Node<Integer> h = head;
while (h != null) {
  h = h.getNext();
}
```

ASSUME THAT `head` POINTS TO THE FIRST ELEMENT - THIS IS HOW WE TRAVERSE A LINKED LIST TILL NULL IS ENCOUNTERED

LINKED LISTS

```
public class LinkedList<T extends Comparable<T>> implements Cloneable {

    private Node<T> head = null;

    public LinkedList() {
    }

}
```

INITIALLY THE HEAD WILL BE NULL - AS YOU CREATE AND ADD NODES TO THE LIST THE HEAD WILL REFERENCE THE FIRST ELEMENT IN THE LIST

LINKED LISTS

COMPLEXITY OF COMMON OPERATIONS

ADDING A NEW ELEMENT TO THE END OF A LIST

O(N)

LINKED LISTS

COMPLEXITY OF COMMON OPERATIONS

ADDING A NEW ELEMENT TO THE BEGINNING OF A LIST

O(1)

LINKED LISTS

COMPLEXITY OF COMMON OPERATIONS

FIND AN ELEMENT IN A LINKED LIST

O(N)

LINKED LISTS

COMPLEXITY OF COMMON OPERATIONS

DELETING THE FIRST ELEMENT IN A LINKED LIST

O(1)

LINKED LISTS

COMPLEXITY OF COMMON OPERATIONS

DELETING A RANDOM ELEMENT IN A LINKED LIST

O(N)

COUNT NODES

COUNT NODES

COUNT THE NUMBER OF NODES IN A LINKED LIST

THIS INVOLVES WALKING THROUGH THE LINKED LIST TO SEE HOW MANY NODES ARE PRESENT

THE MOST EFFICIENT WAY OF DOING THIS IS BY HOLDING A COUNTER WHICH YOU INCREMENT EACH TIME YOU ADD AN ELEMENT AND DECREMENT EACH TIME YOU DELETE AN ELEMENT

COUNT NODES

```
public int countNodes() {
    if (head  == null) {
       return  0;
    } else {
        Node<T> curr = head;
        int count = 0;
        while (curr != null) {
            curr = curr.getNext();
            count++;
        }
        return count;
    }
}
```

COUNT NODES

```
public int countNodes() {
    if (head  == null) {
        return  0;
    } else {
        Node<T> curr = head;
        int count = 0;
        while (curr != null) {
            curr = curr.getNext();
            count++;
        }
        return count;
    }
}
```

IF THE HEAD IS NULL IT MEANS NO NODE EXISTS IN THE LINKED LIST

COUNT NODES

```
public int countNodes() {
    if (head  == null) {
        return  0;
    } else {
        Node<T> curr = head;
        int count = 0;
        while (curr != null) {
            curr = curr.getNext();
            count++;
        }
        return count;
    }
}
```

WALK THROUGH THE LIST TILL YOU REACH NULL

COUNT NODES

```
public int countNodes() {
    if (head  == null) {
        return  0;
    } else {
        Node<T> curr = head;
        int count = 0;
        while (curr != null) {
            curr = curr.getNext();
            count++;
        }
        return count;
    }
}
```

INCREMENT A COUNTER FOR EVERY ELEMENT IN THE LIST AND RETURN THE COUNT OF ELEMENTS

ADD A NODE

ADD A NODE

APPENDS A NEW NODE WITH SOME DATA TO THE VERY END OF A LINKED LIST

INVOLVES TRAVERSING A LINKED LIST TO THE VERY END AND THEN ADDING A NODE

REMEMBER THE NEWLY ADDED NODE SHOULD HAVE THE NEXT SET TO NULL

ADD A NODE

```
public void addNode(T data) {
    if (head == null) {
        head = new Node<T>(data);
    } else {
        Node<T> curr = head;
        while (curr.getNext() != null) {
            curr = curr.getNext();
        }
        curr.setNext(new Node<T>(data));
    }
}
```

ADD A NODE

```
public void addNode(T data) {
    if (head == null) {
        head = new Node<T>(data);
    } else {
        Node<T> curr = head;
        while (curr.getNext() != null) {
            curr = curr.getNext();
        }
        curr.setNext(new Node<T>(data));
    }
}
```

IF THIS IS THE FIRST NODE IN THE LINKED LIST CREATE A NEW NODE AND ASSIGN THE HEAD REFERENCE TO THAT NODE

ADD A NODE

```
public void addNode(T data) {
    if (head == null) {
        head = new Node<T>(data);
    } else {
        Node<T> curr = head;
        while (curr.getNext() != null) {
            curr = curr.getNext();
        }
        curr.setNext(new Node<T>(data));
    }
}
```

FOLLOW THE LINKED ELEMENTS TILL WE GET TO THE VERY LAST ELEMENT OF THE CURRENT LIST

ADD A NODE

```
public void addNode(T data) {
    if (head == null) {
        head = new Node<T>(data);
    } else {
        Node<T> curr = head;
        while (curr.getNext() != null) {
            curr = curr.getNext();
        }
        curr.setNext(new Node<T>(data));
    }
}
```

FOLLOW THE LINKED ELEMENTS TILL WE GET TO THE VERY LAST ELEMENT OF THE CURRENT LIST

ADD A NODE

```
public void addNode(T data) {
    if (head == null) {
        head = new Node<T>(data);
    } else {
        Node<T> curr = head;
        while (curr.getNext() != null) {
            curr = curr.getNext();
        }
        curr.setNext(new Node<T>(data));
    }
}
```

CREATE A NEW NODE AND POINT THE VERY LAST ELEMENT TO IT

ADD A NODE

```
public void addNode(T data) {
    if (head == null) {
        head = new Node<T>(data);
    } else {
        Node<T> curr = head;
        while (curr.getNext() != null) {
            curr = curr.getNext();
        }
        curr.setNext(new Node<T>(data));
    }
}
```

REMEMBER THAT THE NEXT POINTER IS SET TO NULL IN THE CONSTRUCTOR OF NODE

RETURN THE FIRST ELEMENT IN A LIST I.E POP ELEMENT

RETURN THE FIRST ELEMENT IN A LIST I.E POP ELEMENT

WE WANT TO ACCESS THE FIRST ELEMENT IN THE LIST AND REMOVE THE FIRST ELEMENT FROM THE LIST AS WELL

CHANGING THE FIRST ELEMENT INVOLVES UPDATING THE HEAD REFERENCE

RETURN THE FIRST ELEMENT IN A LIST I.E POP ELEMENT

```
public T popElement() {
    if (head != null) {
        T topElement = head.getData();

        head = head.getNext();

        return topElement;
    }

    return null;
}
```

RETURN THE FIRST ELEMENT IN A LIST I.E POP ELEMENT

```
public T popElement() {
    if (head != null) {
        T topElement = head.getData();

        head = head.getNext();

        return topElement;
    }

    return null;
}
```

IF THE HEAD IS NULL IT MEANS THERE ARE NO ELEMENTS IN THE LINKED LIST

RETURN THE FIRST ELEMENT IN A LIST I.E POP ELEMENT

```
public T popElement() {
    if (head != null) {
        T topElement = head.getData();

        head = head.getNext();

        return topElement;
    }

    return null;
}
```

IF THERE ARE SOME NODES IN THE LINKED LIST WE ACCESS THE FIRST ELEMENT VIA THE HEAD REFERENCE

RETURN THE FIRST ELEMENT IN A LIST I.E POP ELEMENT

```
public T popElement() {
    if (head != null) {
        T topElement = head.getData();

        head = head.getNext();

        return topElement;
    }

    return null;
}
```

ACCESS THE TOP ELEMENT AND STORE IT SO WE CAN RETURN IT

RETURN THE FIRST ELEMENT IN A LIST I.E POP ELEMENT

```
public T popElement() {
    if (head != null) {
        T topElement = head.getData();

        head = head.getNext();

        return topElement;
    }

    return null;
}
```

JUST MOVE THE HEAD POINTER TO REFERENCE THE NEXT ELEMENT!