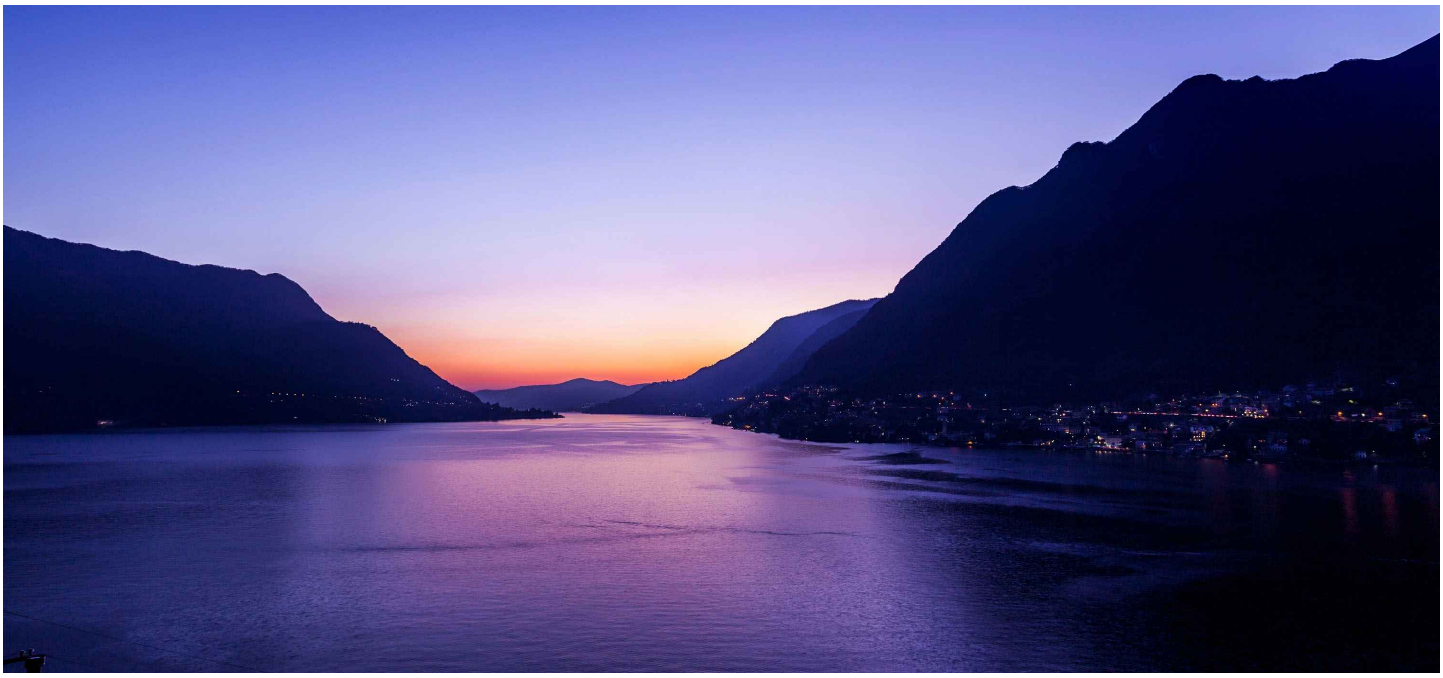


23-DSA Complexity

Author: Vincent Lau

Learning Objectives

> Get familiar with basic data structures and the time complexity of their operations.

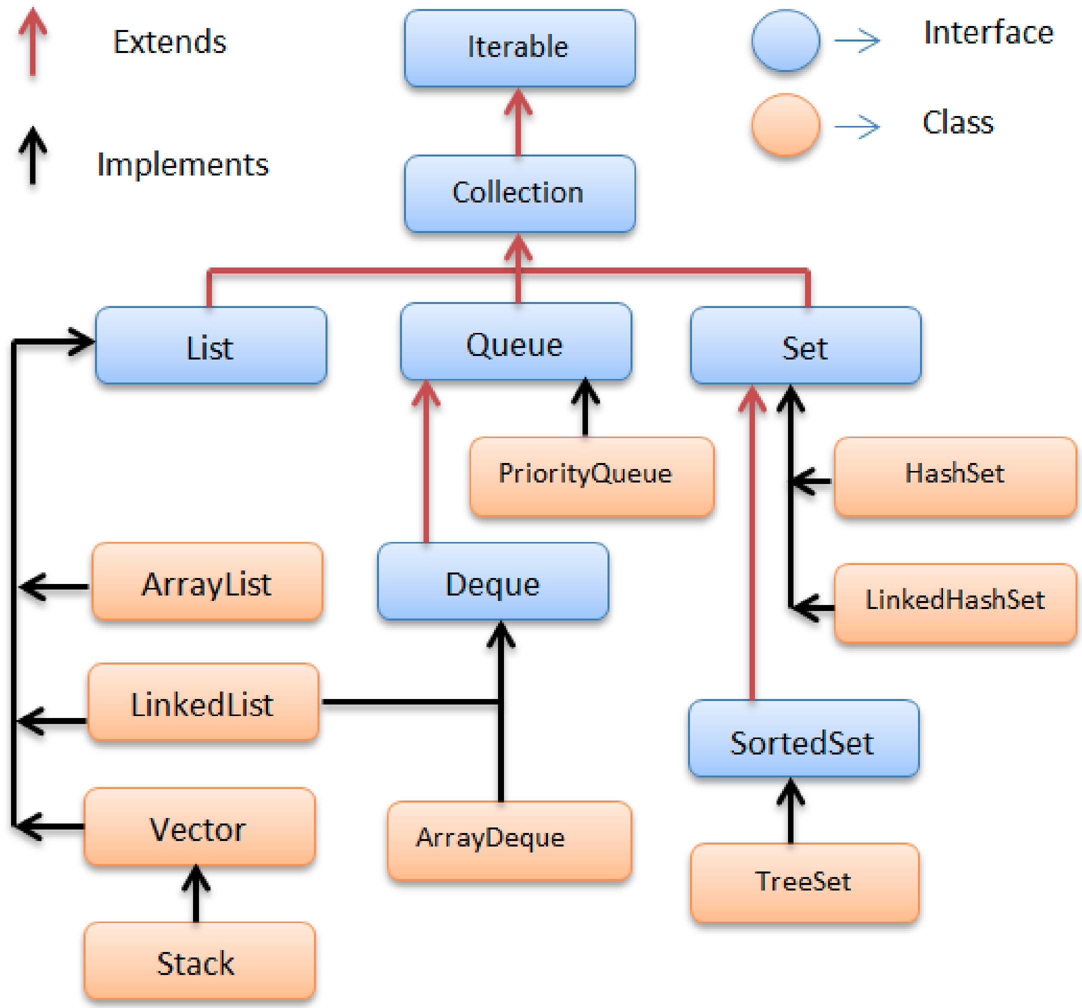
> Understand the difference between Array and Linked List in terms of memory allocation.

> Understand how a Hash Table works.

> Understand how stacks and queues work.

> Understand why modifying a String is expensive.

Data Structures



- **Data structures** are ways of organizing data on a computer so that it can be accessed and manipulated effectively and efficiently.
- Note that this chapter is language-agnostic and does not only apply to Java.

Arrays

- An array is a data structure that contains a group of elements.
- Arrays are stored in contiguous back-to-back memory slots.

Static Arrays (Array)

- **An array that is declared with the static keyword is known as static array**.
- It allocates memory at **compile-time** whose size is fixed. We cannot alter the static array.
- If we want an array to be sized based on input from the user, then we cannot use static arrays.

```
public class StaticArray {
    private static String[] arr; // declaration

    static {
```

```java
      arr = new String[3]; // initialization
      arr[0] = "Hi";
      arr[1] = "Hello";
      arr[2] = "How are you?";
    }

  public static void main(String[] args) {
    for (int i = 0; i < arr.length; i++) {
      System.out.println(arr[i]);
    }
  }
}
```

Dynamic Arrays (ArrayList)

- Size determined during runtime.
- Array can be resized (larger or smaller)
- An ArrayList is considered a dynamic array because it can change in size, although it still uses Array under the hood and creates a larger Array (i.e. resizing) as need be.

```java
public class DynamicArray {

  public static void main(String[] args) {
    // 1. Constructor ArrayList(), Create an empty array of integers
    ArrayList<Integer> integers = new ArrayList<>();

    // Add numbers from 0 to 9 to an array
    for (int i =0 ; i < 10; i++) {
        integers.add(i);
    }

    System.out.println(integers); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    // 2. Create an array of strings based on another array of strings, constructor ArrayList()
    // 2.1. Create a source array
    ArrayList<String> strings = new ArrayList<>();
    strings.add("Winter");
    strings.add("Spring");
    strings.add("Autumn");
    strings.add("Summer");
    System.out.println(strings);  // [Winter, Spring, Autumn, Summer]

    // 2.2. Use constructor ArrayList(Collection<? extends E>)
```

```
    ArrayList<String> strings2 = new ArrayList<>(strings); // copy array
    System.out.println(strings2);  // [Winter, Spring, Autumn, Summer]
  }
}
```

Time Complexity

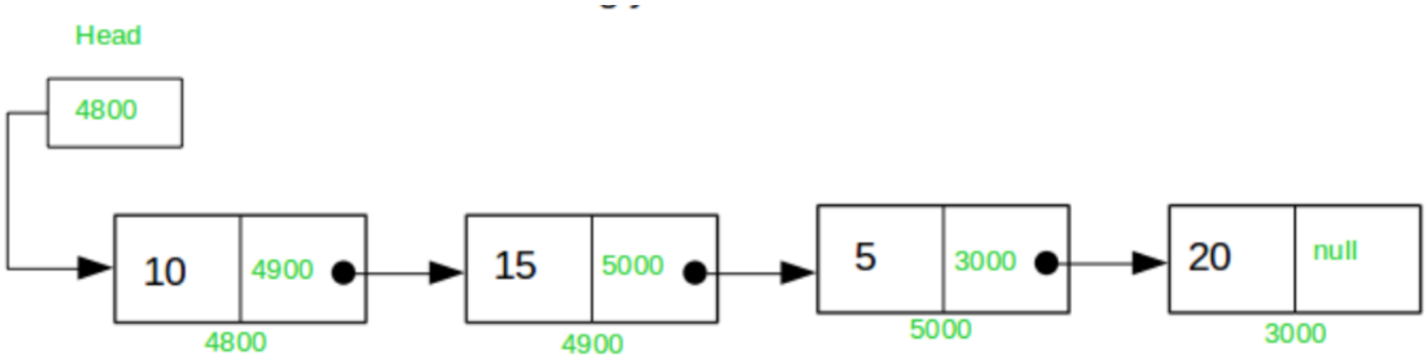
- Retrieving or modifying an element inside an array by **index** has a time complexity of `O(1)`.
- Initializing an array has a time complexity of `O(n)`.
- Inserting a new element in a static array entails `O(n)` time complexity.

Linked List

- Unlike an Array, the elements of a Linked List can be stored anywhere in memory.
- A Linked List connects a group of elements using **pointers** by storing their address. Each node in a Linked List has a value and the pointer to the next node.
- The first node in a Linked List is called the head, and the last node is called the tail.

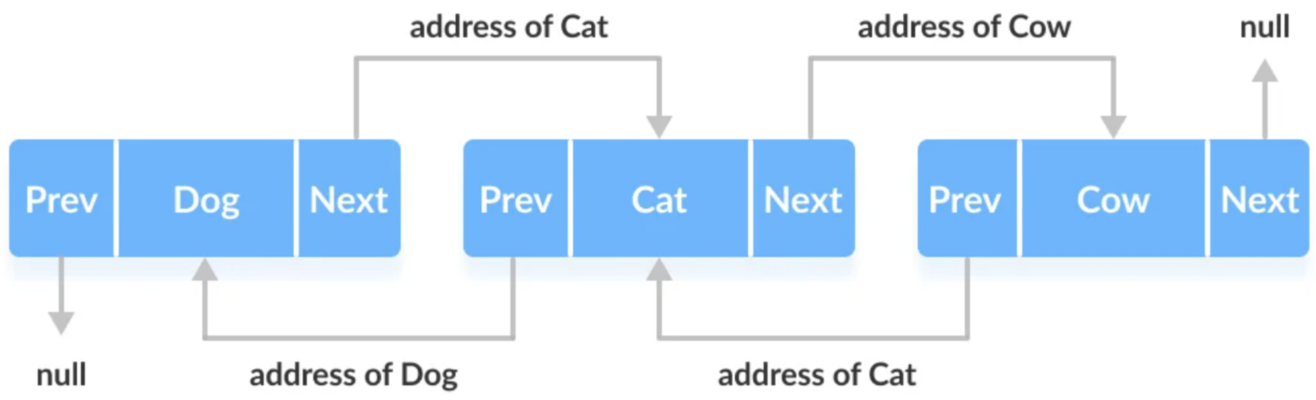
Types of Linked List

Singly Linked Lists

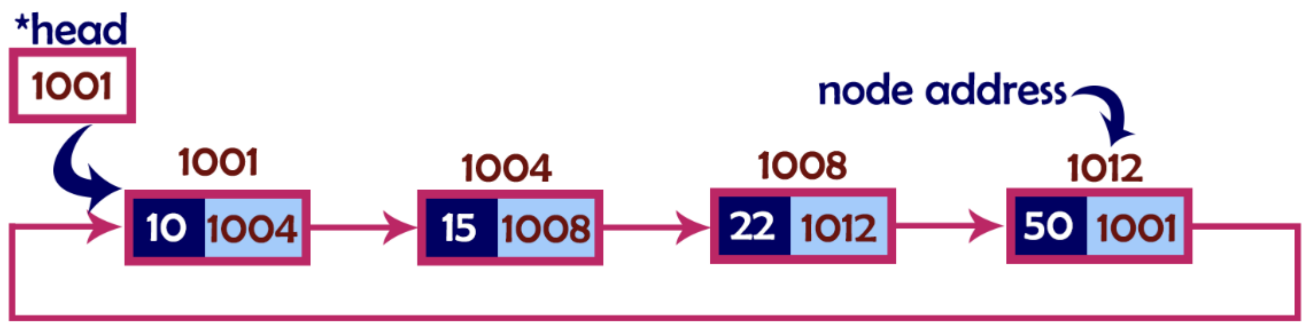


Doubly Linked Lists

The `LinkedList` in Java is a Doubly Linked List, where each node in the list has a pointer to the previous node and a pointer to the next node.



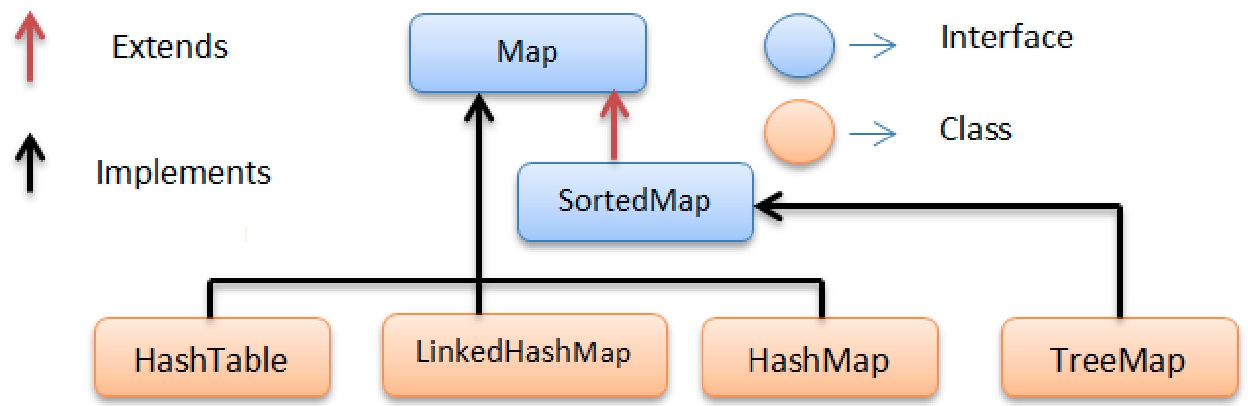
Circular Linked Lists



Time Complexity

- Retrieving and modifying an element in a Linked List has a time complexity of `O(n)`.
- Initializing an array has a time complexity of `O(n)`.
- **Inserting a new element at the beginning in a (singly) Linked List entails `O(1)` time complexity; inserting at the end has `O(n)` time complexity. Inserting somewhere in the middle has `O(m)` complexity, where `m` is the number of existing elements before the new element.**
- For **Doubly** Linked List, inserting a new element at the beginning is also with O(1), and **inserting it at the end would be O(1), but not O(n).**

HashTables / HashMaps



- A Hash Table is a data structure comprised of key-value pairs.
- We can access a value given a key, but the reverse is not possible.
- Under the hood, a Hash Table is built on top of an Array, i.e. an Array of Linked Lists.
- The key is transformed into an index by taking modulo of the output of a hash function.
- When two keys get hashed to the same index, we call this a ***hash collison***. The new element is appended to the end of the underlying Linked List, where each node of the Linked List points back to the original key.

Basic Operations

```
public class HashtableExample {
    public static void main(String[] args) {
        //1. Create Hashtable
        Hashtable<Integer, String> hashtable = new Hashtable<>();

        //2. Add mappings to hashtable , self-define the key
        hashtable.put(1,  "A");
        hashtable.put(2,  "B" );
        hashtable.put(3,  "C");
        hashtable.put(3,  "D"); // overwrite the key 3 with value "C"

        System.out.println(hashtable); // {3=D, 2=B, 1=A}

        //3. Access a mapping by key
        String value = hashtable.get(1);       //A
        System.out.println(value);

        //4. Remove a mapping
        hashtable.remove(3);            // 3 is deleted

        //5. Iterate over mappings
        Iterator<Integer> itr = hashtable.keySet().iterator();
```

```java

        while(itr.hasNext()) {
            Integer key = itr.next();
            String mappedValue = hashtable.get(key);

            System.out.println("Key: " + key + ", Value: " + mappedValue);
        }
    }
}
/*
Output:
{3=D, 2=B, 1=A}
A
Key: 2, Value: B
Key: 1, Value: A
*/
```

- HashMap is non-synchronized where Hashtable is synchronized. That's why HashMap performs much faster.
- HashMap allows Null for both key and value while Hashtable doesn't allow null for both key and value. Otherwise, we will get a null pointer exception.

```java
class HashMapExample {
    public static void main(String args[]) {
        //----------hashtable -------------------------
        Map<Integer,String> ht = new Hashtable<>();
        ht.put(101,"Ajay");
        ht.put(101,"Vijay");
        ht.put(102,"Ravi");
        ht.put(103,"Rahul");
        System.out.println("-------------Hash table--------------");
        for (Map.Entry<Integer, String> m : ht.entrySet()) {
            System.out.println(m.getKey()+" "+m.getValue());
        }

        //----------------hashmap--------------------------------
        Map<Integer, String> hm = new HashMap<>();
        hm.put(100,"Amit");
        hm.put(104,"Amit");
        hm.put(101,"Vijay");
        hm.put(102,"Rahul");
        System.out.println("-----------Hash map-----------");
        for (Map.Entry<Integer, String> m: hm.entrySet()) {
            System.out.println(m.getKey()+" "+m.getValue());
```

```
        }
    }
}
/*
-------------Hash table--------------
103 Rahul
102 Ravi
101 Vijay
-----------Hash map-----------
100 Amit
101 Vijay
102 Rahul
104 Amit
*/
```

Time Complexity

- Insertion, deletion and retrieval all have a time complexity of `O(1)`.
- Initialization has a time complexity of `O(n)`.

Queues & Stacks

- A queue is a data structure that follows FIFO (First-In-First-Out).

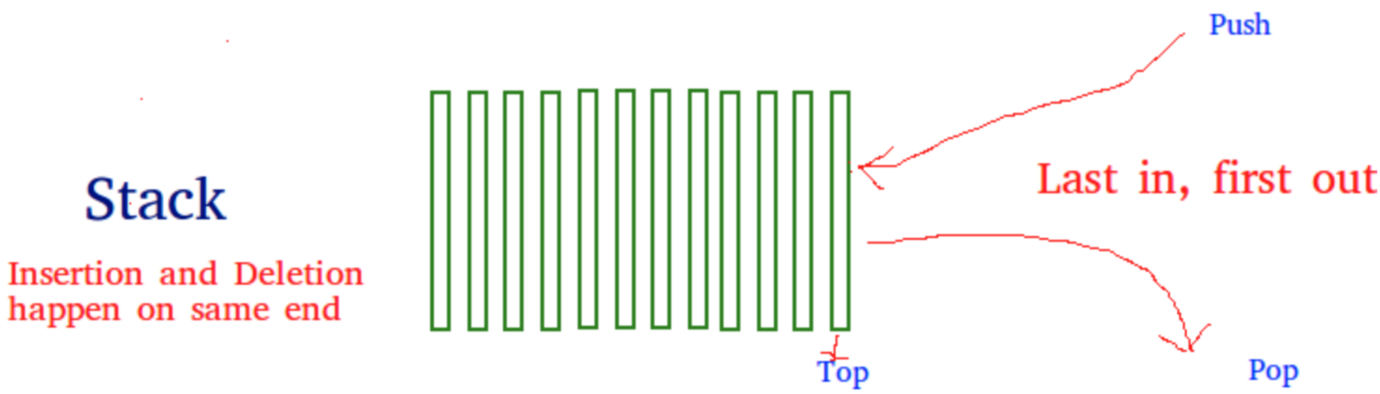
Basic Operations

- Stacks and Queues are usually implemented with a LinkedList in Java.

```java
public class QueueExample {

    public static void main(String[] args) {
        Queue<Integer> q = new LinkedList<>();
```

```java

        // Adds elements {0, 1, 2, 3, 4} to the queue
        for (int i = 0; i < 5; i++) {
            q.add(i);
        }
        // Display contents of the queue.
        System.out.println("Elements of queue " + q);

        // To remove the head of queue.
        int removedElement = q.remove();
        System.out.println("removed element-" + removedElement);
        // queue after removal
        System.out.println(q);

        // Return the head of queue
        int head = q.peek();
        System.out.println("head of queue-" + head);

        // To remove head of queue and return it
        System.out.println("removed head of queue-" + q.poll());
        // queue after removal
        System.out.println(q);

        // Rest all methods of collection interface like size and contains
        // can be used with this implementation.
        int size = q.size();
        System.out.println("Size of queue-" + size);
    }
}
/*
Output:
Elements of queue [0, 1, 2, 3, 4]
removed element-0
[1, 2, 3, 4]
head of queue-1
removed head of queue-1
[2, 3, 4]
Size of queue-3
*/
```

- A stack is a data structure that follows LIFO (Last-In-First-Out).




```java
public class StackExample {
    public static void main(String a[]){
        //declare a stack object
        Stack<Integer> stack = new Stack<>();
        //print initial stack
        System.out.println("Initial stack : "  + stack);
        //isEmpty()
        System.out.println("Is stack Empty? : "  + stack.isEmpty());
        //push() operation
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.push(40);
        //print non-empty stack
        System.out.println("Stack after push operation: "  + stack);
        //pop() operation
        System.out.println("Element popped out:"  + stack.pop());
        System.out.println("Stack after Pop Operation : "  + stack);
        //search() operation
        System.out.println("Element 10 found at position: "  + stack.search(10));
        System.out.println("Is Stack empty? : "  + stack.isEmpty());

        System.out.println("Stack elements using Java 8 forEach:");
        //Get a stream for the stack
        Stream stream = stack.stream();
        //traverse though each stream object using forEach construct of Java 8
        stream.forEach((element) -> {
            System.out.print(element + " ");  // print element
        });
    }
}
/*
output:
```

```
Initial stack : []
Is stack Empty? : true
Stack after push operation: [10, 20, 30, 40]
Element popped out:40
Stack after Pop Operation : [10, 20, 30]
Element 10 found at position: 3
Is Stack empty? : false
Stack elements using Java 8 forEach:
10 20 30
*/
```

Time Complexity

- Insertion and deletion of an element is `O(1)`.
- Searching a specific element is `O(n)`.
- Initialization is `O(n)`.

Reading Exercise

Linked Lists

Delete Node

```
/* Linked list Node*/
class Node {
    int data;
    private Node next; // next is storing the reference of the node

    Node(int d) {
        data = d;
        next = null;
    }

    public Node getNext() {
        return this.next;
    }

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

// Demonstrate deletion in singly linked list
```

```java
class LinkedList {
    Node head; // head of list

    /* Given a key, deletes the first
       occurrence of key in
     * linked list */
    void deleteNode(int key) { // delete by node value
        // Store head node
        Node temp = head;
        Node prev = null;

        // If head node itself holds the key to be deleted
        if (temp != null && temp.data == key) {
            head = temp.getNext(); // Changed head
            return;
        }

        // Search for the key to be deleted, keep track of
        // the previous node as we need to change temp.next
        while (temp != null && temp.data != key) {
            prev = temp;
            temp = temp.getNext();
        }

        // If key was not present in linked list
        if (temp == null)
            return;

        // Unlink the node from linked list
        prev.setNext(temp.getNext());
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data) // insert to tail by node data
    {
        Node new_node = new Node(new_data);
        new_node.setNext(head);
        this.head = new_node;
    }

    /* This function prints contents of linked list starting
       from the given node */
    public void printList()
    {
        Node tnode = head;
        while (tnode != null) {
            System.out.print(tnode.data + " ");
```

```
                tnode = tnode.getNext();
            }
    }

    /* Driver program to test above functions.*/
    public static void main(String[] args)
    {
        LinkedList llist = new LinkedList();

        llist.push(7);
        llist.push(1);
        llist.push(3);
        llist.push(2);

        System.out.println("\nCreated Linked list is:");
        llist.printList(); // 2 3 1 7

        llist.deleteNode(1); // Delete node with data 1

        Node node = llist.getNext().getNext();

        System.out.println(
            "\nLinked List after Deletion of 1:");
        llist.printList(); // 2 3 7
    }
}
```

Insert Node (Append)

```
class LinkedList
{
    Node head;  // head of list

    /* Linked list Node*/
    class Node {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* Appends a new node at the end. */
    public void append(int new_data) {
        /* 1. Allocate the Node &
           2. Put in the data
           3. Set next as null */
```

```
        Node new_node = new Node(new_data);

        /* 4. If the Linked List is empty, then make the
              new node as head */
        if (head == null) {
            head = new Node(new_data);
            return;
        }

        /* 4. This new node is going to be the last node, so
              make next of it as null */
        new_node.next = null;

        /* 5. Else traverse till the last node */
        Node last = head;
        while (last.next != null)
            last = last.next;

        /* 6. Change the next of last node */
        last.next = new_node;
        return;
    }

    /* This function prints contents of linked list starting from
        the given node */
    public void printList() {
        Node tnode = head;
        while (tnode != null)
        {
            System.out.print(tnode.data+" ");
            tnode = tnode.next;
        }
    }

    /* Driver program to test above functions. */
    public static void main(String[] args) {
        /* Start with the empty list */
        LinkedList llist = new LinkedList();
        // Insert 6.  So linked list becomes 6->NUllist
        llist.append(6);

        // Insert 4 at the end. So linked list becomes
        // 6->4->NUllist
        llist.append(4);

        System.out.println("\nCreated Linked list is: ");
        llist.printList();
```

```
    }
}
```

Questions

- What is the difference between Array and Linked List in terms of memory allocation?
- How does a Hash Table work?
- How do stacks and queues work respectively?
- Why is modifying a String expensive?