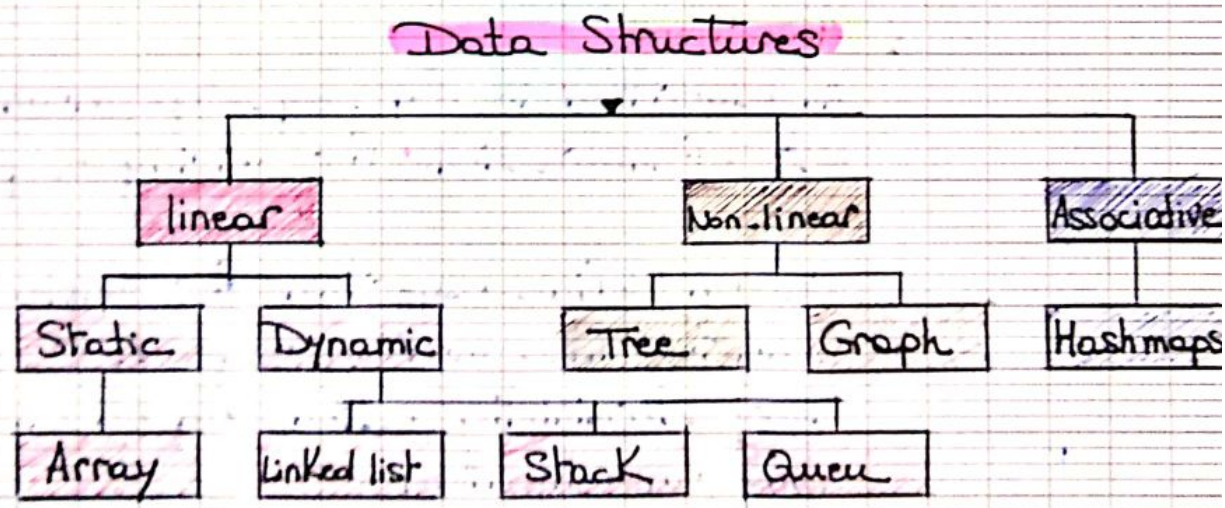
Data Structures In Java

→ **What is Data Structure (DSA)**

A **data structure** is a meaningful way of **arranging** and **sorting** data in a computer (RAM) so as to use it efficiently

Some commonly used data structures include:

- Arrays, Dynamic Arrays (List), LinKed Lists
- Stacks, Queus
- Priority Queus
- HashTables, Hashsets
- Trees, Graphs

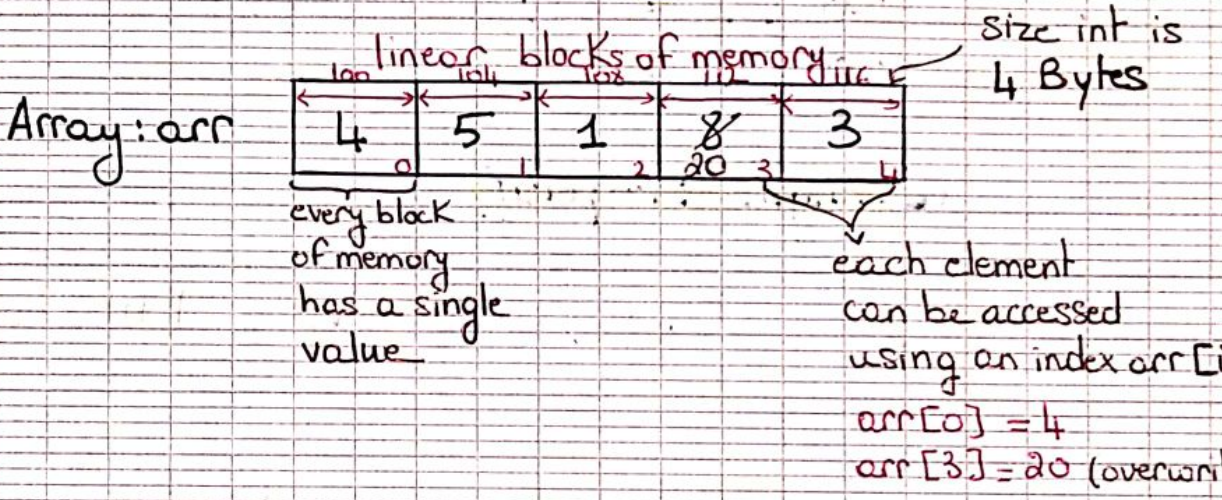


→ Arrays

An array is linear collection of element of the same type.

Arrays are used to store multiple values in a single variable, instead of declaring separate variable for each value.

Each array location is accessed using an index (i), the indexing starts from 0



↳ Initalising Arrays

```
// Init an Array of Strings
String[] billPayments = {"Electricity", "Mobile", "CC"

// Init an Array of Integers
int [] myNum = { 10, 20, 30, 40 }
```

Updating Arrays

```
// Init an fixed size array
int [] arr = new int[5];

// update the data at ith index
arr [0] = 10;
arr [1] = 20;
arr [2] = 30;
arr [3] = 40;
arr [4] = 50;
```

Length Property

```
String [] cars = {"Volvo", "BMW", "Ford", "Mazda"};

system.out.println (cars.length); // output : 4
                                  ↳ gives the size of the array
```

Looping over Arrays

```
// For Loop
for (int i=0; i<myArray.length ; i++){
        system.out.println (myArray [i]);
}
// output: Volvo
           BMW
           Ford
           Mazda
```

```
// Enhanced For loop
for (String val : cars) {
    system.out.println (val);
}
//output: Volvo
          BMW
          Ford
          Mazda
```

Arrays Class Methods

The Arrays Class in java.util package is a part of Java Collection Framework. This class provides static methodes to dynamically creat and access Java arrays.

There are some of the Arrays class Methodes:

asList (arrayname) : Transforming the array to arraylist

binarySearch (array, val) : Returns the index of the specified value

copyOf (originalArray, newLength): Copies the specified array

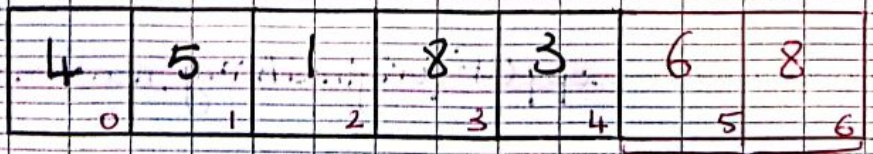
equals (array1, array 2): Checks if both the array's are equal or not

sort(Array): Sort the specified Array in ascending order

toString(Array): Returns a String representation of the content of this array

→ ArrayList

- An array list is dynamic array for storing elements. It is like an array, but with no size limit.
- Arraylist class maintains insertion order (i.e. I add an element it will always get added at the end)
- We can add, remove, or search elements
- In Java it implement the List Interface so that we can use all methodes of List interface here.



It's a dynamic array we can always add more elements

↳ Array List Operations

- Random access takes $\Theta(1)$ Time
- Adding element take amortised constant time $\Theta(1)$
- Inserting / Deleting take $\Theta(n)$ Time
- Searching takes $\Theta(n)$ time for unsorted array and $\Theta(\log n)$ for sorted one.

↳ ArrayList Declaration and Creation

```
//General Syntax                                  we can specify the capacity
ArrayList<obj-type> name = new ArrayList<obj-type>();

//ArrayList of Integers with capacity =10
ArrayList<Integer> arr = newArrayList<Integer>(10);
```

↳ Loop through an Array List

1)
```
for(int i= 0; i< list.size() ; i++){
    Systeme.out.println(list.get(i));
} // for loop
```

2)
```
for (Integer x : list){
    System.out.println(x);
} // for-each loop
```

The ArrayList Class Methods

java.util.ArrayList :

add (element) : adds new element at the end of the list

add (index, element) : adds new element at the specified index in the list

Clear (-) : Removes all element from the list

Contains (element) : Return true if this list contaiht the specified element

get (index) : Return the element from the list at the specified index

indexOf (element) : Return the inder of the first matching element in the list

lastInder (element) : Relūn the inder of the last matching element in the list

isEmpty (-) : Return true if the list contains no elements

remove (element) : Remove the first match of element

size(): Return the number of element in the list

remove(index): Remove the element at the specified index and return it

set(index, element): Sets the element at the specified index

↳ Array List From / to Arrays

It is also possible to create an equivalent list from an array list using the static method Arrays.asList(array)

Example: ArrayList from an Array of objects:

```
String[] array = {"red", "green", "blue"};
ArrayList<String> list = new ArrayList<String>
                         (Arrays.asList(array));
```

It is possible to start with a List and create an equivalent array containing the same elements in the same order by using toArray() method

Example: ArrayList to Array

```
String [] array1 = new String [list.size()];

list.toArray(array1);
```

Collections Utility Class

- Collections class, is a treasure store of class (static) methods for doing things to collections, such as sorting or shuffling them, picking out the largest or smallest item, and so on.

- Collections methods can be used with ArrayList object

- We need to import java.util.Collections

Methods of collections class

- Reversing:

```
Collections.reverse(list);//reverse the order
```

- Sorting

```
Collections.sort(list);//Sort alphabetically
                       or in asc order
```

- Finding max/min

```
collections.max/min(list);//retur values, not index
```

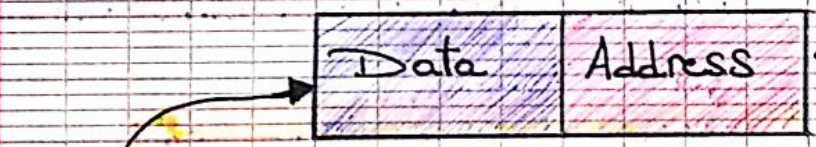
Shuffling:

```
Collections.Shuffle(list);
// put the list in random order
```

→ **LinKed List**

LinKed List is a linear data structure, in wich element are represented as object and Stored in non-contiguous (غير متجاورة) memory location.

It consist of a group of nodes in sequence. Each node hold two pieces of data, the first one hold the "value". and the second one hold the "address" of the next node



contains the actual data

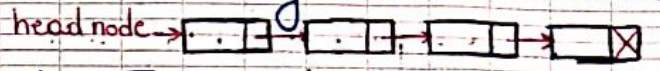
NODE

contain the address of the next node of the list

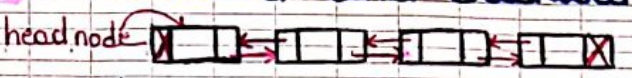
Each node references the next node, we call the first node the head and the last node the tail.

Types of Linked list

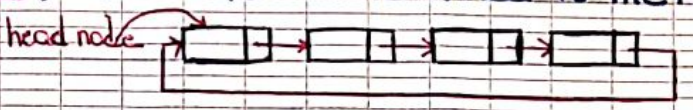
- **Single Linked List**: Navigation is forward only



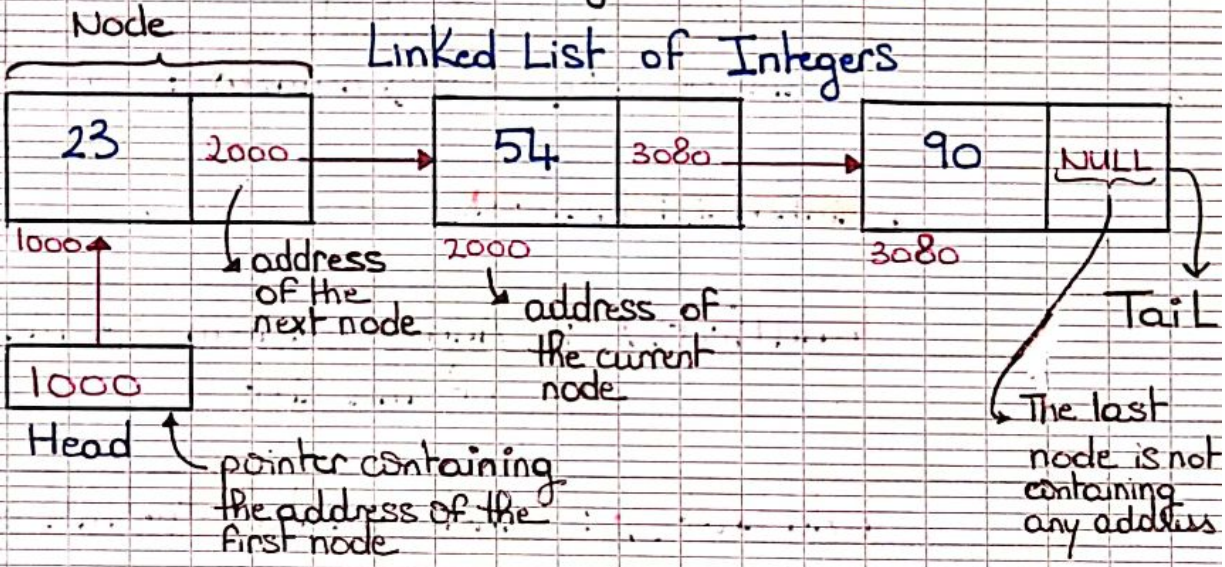
- **Doubly Linked List**: Forward and Backward navigate



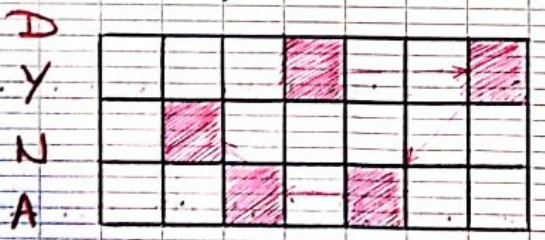
- **Circular Linked List**: Last element is linked to the first one



Representation of a single linked list



Memory Allocation



Linked List don't need to be contiguous in memory. So nodes don't need to be stored in sequential order in memory, address are randomly stored in the memory locations.

LinkedList class in Java

Like as Arrays and ArrayList there is a predefined class in Java for LinkedList. The LinkedList class is a collection which can contain many object of the same type, just like ArrayList.

```
// Import the LinkedList class
import java.util.LinkedList;
// Creation of LinkedList
LinkedList<Integer> numbers = new LinkedList<>();
```

LinkedList Methods

addFirst() : Adds an item to the beginning of the List

addLast() : Adds an item to the end of the list

removeFirst() : Remove an item from the beginning

removeLast() : Remove an item from the end

add(int index, E element) : Inserts an element at the specified position in the list

remove(int index) : remove the element at the specified position in the list

contains(object o) : return true if the list contains the specified element

get(int index) : returns the element at the specified position

getFirst() : return the first element

getLast() : return the last element

indexOf(object o) : return the index of the first occurence of the specified element or -1 if the list does'nt contain the element

set(int index, E element) : replace the element at the specified position with the specified element

clear() : remove all elements from the list

size() : return the number of element in the list

toString() : return a string representation of the list

LinKed List Implementation in Java

To implement a linKed list in Java, we organize our code into three distinct classes, each serving a specifice purpose.
The first class 'NodeData', encapsulates the value of the data stored in a node.
The second class 'Node', acts as the building block of our linKed list, representing an individual node.
Finally, the third class, 'LinKedList', manages the linKed list structure.
By compartmentalizing our code into these three classes, we create a modular and organized approch to building and maintaining linKed list.

1. NodeData Class

- purpose: Represents the value of the data stored in a node.
- Attribute: 'value': Holds the actual data value. It can have any data type
- Constructor: Initializes the data value when an instance is created
- Methods: 'compareTo (NodeData nd) : int' method designed to implement 'Comparable' interface. This method is commonly used in sorting and ordering operations.

- Code:

```
public class NodeData {
    private int value; //int can be replaced by any type

    //constructor
    public NodeData (int value) {
        this.value = value;
    }

    // Compare the value of two different nodes
    public int compareTo (NodeData nd) {
        if (this.value == nd.value)
            return 0;
        if (this.value < nd.value)
            return -1;
        return 1;
    }
}
```

2. Class Node

- purpose: Represents a node in the linked list, containing both data and a reference to the next node
- Attribute: 'NodeData data' : Holds the data value using the NodeData class.
  'Node next' : Points to the next node in the list
- Constructor: Initializes the data and sets the next reference initially to 'null'

- Code:

```
public classe Node {
    private NodeData data;
    private Node next;

    // Constructor
    public Node (NodeData nd) {
        this.data = nd ;
        this.next = null ;
    }
}
```

3. LinKed List Class

- purpose: Manages the linKed list by providing methods for insertion and display
- Attributes: 'Node head' : Point to the first node
- Constructor: Initializes the head initially to 'null'
- Methods: addFirst ( ) : Insert a new node at the beginning
  addLast ( ) : Insert a new node at the end
  addAt ( ) : Insert a new node at a specific place
  removeFirst( ) : Remove from the beginning
  removeLast( ) : Remove from the end

- Code:

```
public class LinkedList {
    private Node head;
    //Constructor
    public LinkedList() {
        this.head = null;
    }
    //methods for insertion, display and others...
}
```

→ LinkedList class methods implementation

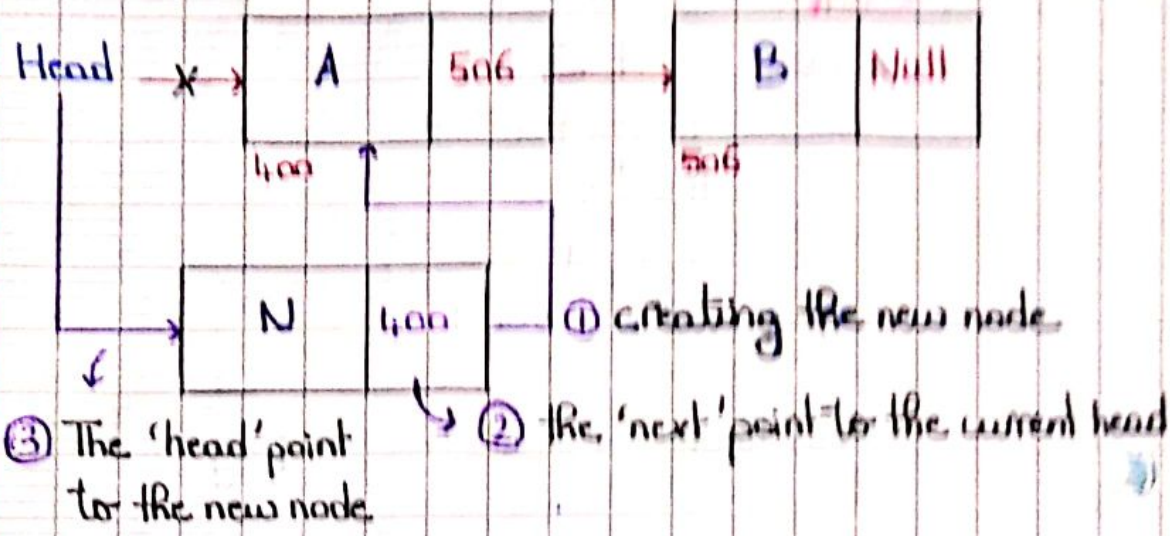
↳ Insertion Methods

In a linked list we can insert a new node at 3 possible positions:

1. At the beginning of the linked list (head)
2. At a specific position (in the middle)
3. At the end of the linked list (tail)

1. How to insert a Node at the beginning of linked list

- Approach:
  1. Create a new node the given data
  2. Set "next" reference of the new node to point to the current head of the linked list
  3. Update the "head" reference to point to the new node



- Code:

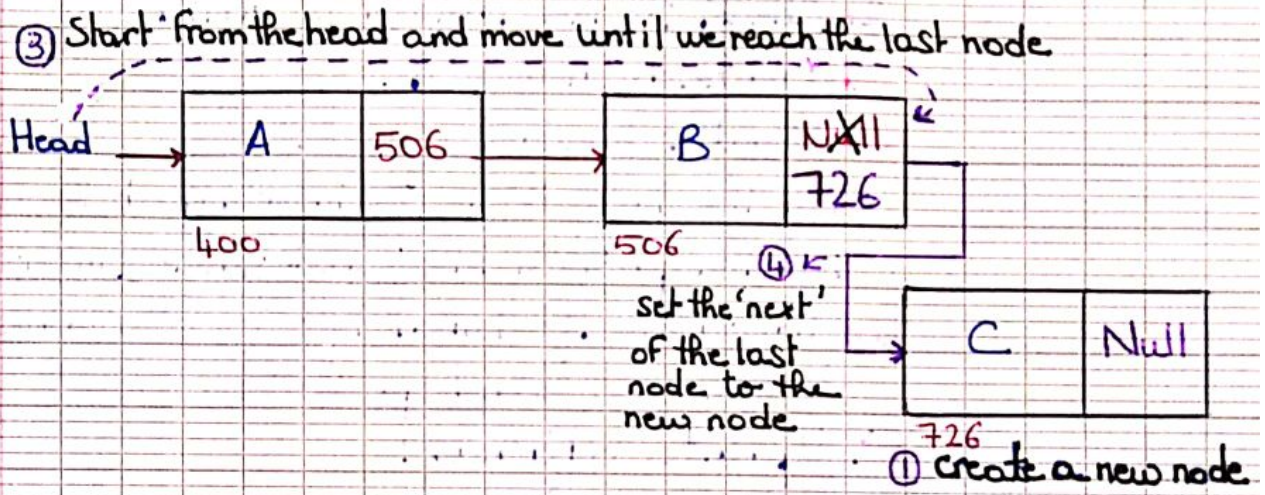
```
public void addHead (NodeData item) {
// creating a new node
Node newNode = new Node (item);
// first possibility : the linked list is empty
if (head == null)
    head = newNode;

// Second possibility : the linked list isn't empty
else {
    newNode.next = head;
    head = newNode;
}
}
```

- Time Complexity:

O(1), because the operation involves a constant number of basic operations, regardless of the size of the linked list.

2- How to insert a node at the end of linked list

- Approach:
  1. Create a new node
  2. Check if the list is empty (if this is the case set the 'head' to the new node)
  3. If the list isn't empty, traverse it to the last node
  4. Attach the new node to the last node

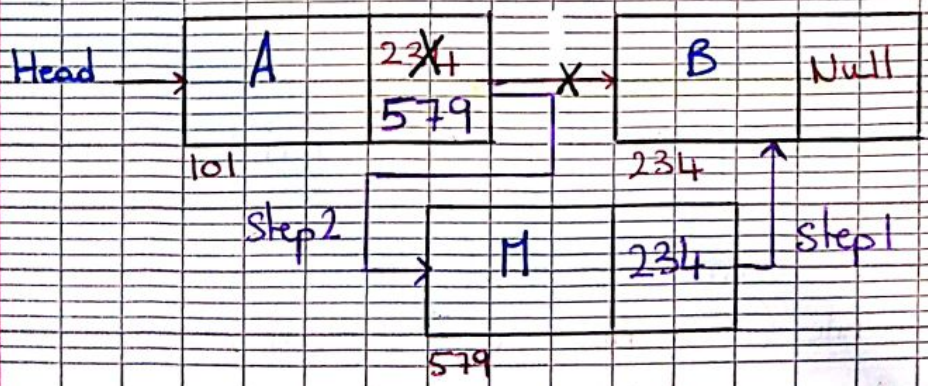


- Code:

```
public void addTail (NodeData item){
//Creating a new node
Node newNode = new Node (item);
//First possibility : the linked list is empty
if (head == null)
    head = newNode;
//Second possibility : the linked list isn't empty
else{
Node current = head;
While (current.next != null)
        current = current.next;
current.next = newNode;
}
```

- Time Complexity :
  O(N), where N is the number of nodes in the linked list. Since there is a loop from head to end, the function does O(n) work.

3. How to insert a node at a specific position
- Approach:
  1. Check if the position is valid
  2. Create a new node
  3. Handle special cases:
     - if the position is 0 (inserting at the beginning)
     - if the position is the last index (inserting at the end)
  4. Traverses the linked list to the node just before the target position
  5. Update references:
     - Adjust the 'next' reference of the new node to point to the node at the target position
     - Update the 'next' reference of the node just before the target position to point to the new node



- code:

```
public void insert ( NodeData item, int index) {
 // Create a new node
  Node newNode = new Node (item);
 // Check if the position is valid
  if (index < 1 || index > countNodes ()+ 1){
      System.out.println (" Invalid Position");
  }
 // Insert into an empty list
  else if (isEmpty()){
        head = newNode;
  }
 // Add at the beginning
  else if ( index == 1){
      addHead(item);
  }
// Add at the end
  else if ( index == countNodes() + 1) {
      addTail (item);
  }
// Add in the middle
  else {
      Node previous = head;
      for (int i = 1; i < index - 1 ; i++) {
          previous = previous.next;
      }
      newNode.next = previous.next;
      previous.next = newNode;
}
}
```

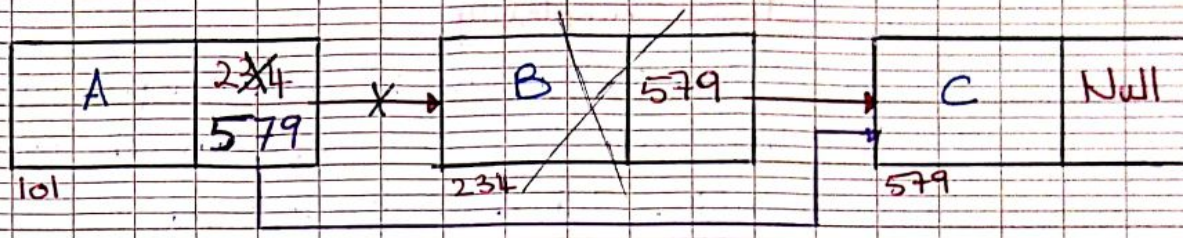
- Time Complexity
  O(n), where n is the number of nodes in the linked list. This is due to the need to traverse the list to find the specific position.

↳ Deletion Methods

1) How to delete node at a specific position

- Approach

1. Handle Edge Cases
   - Check if the linked list is empty, then no deletion is possible
   - Check if the position is valid. Ensure that the position is within the bounds of the linked list. (not less than 1 and not greater than the size of the list)
   - Check if the position is 1, and delete the head in this case.
2. Traverse to the Node Before the target Node
   - Traverse the linked list to the node just before the target node position. Keep track of the previous and current node.
3. Adjust pointers
   - Update the 'next' reference of the previous node to skip the larget node.
   -



Code

```
public void delete (int index ) {
  // Check if the List is empty
   if (isEmpty ()) {
      System.out.println ("Deletion failed: Empty List");
   }
  // Check if the Position is valid
   else if ( index < 1 || index > countNodes()) {
      System.out.println (" Invalid Position");
   }
  // Deleting at the beginning
   else if (index == 1) {
      head = head.next;
   }
  // Deleting from the middle or at the end
   else {
      Node previous = head;
      Node current;
      for (int i = 1 ; i < index-1 ; i++) {
         previous = previous.next;
      }
      current = previous.next;
      previous.next = current.next;
   }
}
```

- Time Complexity
  O(n), where n is the length of the linked list. This is because we need to traverse the entire linked list to find the node to be deleted

2) Two other useful methods for deleting nodes

→ Delete the head node

```
public void deleteHead() {
    if (!isEmpty())
        head = head.next;
}
```

→ Delete all nodes in the list

```
public void clear() {
    head = null;
}
```

Basic Methods

In the implementation of the linked list class we have incorporated several essential methods to enhance the functionality and managment of the linked list. These methods serve as the backbone of our data structure, supporting other operation like delction and insertion.

1) isEmpty()

This method determines whether the linked list is empty, playing a crucial role in guarding against operations on non-existent elements.

code:

```
public boolean isEmpty(){
    if (head == null)
        return true;

    return false;
}
```

2) countNodes()

Counting the number of nodes in our linked list provides valuable insights into the size and scale of our data structure.

code:

```
public int countNodes(){
    int count = 0;
    Node current = head;
    while (current != null){
        count++;
        current = current.next;
    }
    return count;
}
```

3) indexOf()

By determining the index of a specific value within the linked list, this method facilitates targeted operation on elements

- Code:

```
public int indexOf (int item){
 int index;
 Node current = head;
 While ( current != null){
     if ( current.data.value ==item){
         return index;
      }
   current = current.next
    index ++;
  }
 return -1;
}
```

4) Search()

This method allows for the efficient retrieval of a node based on its data value, enhancing the accessibility and utility of our linked list

- Code:

```
public boolean search (NodeData item) {
  Node current = head;
  while (current != null) {
    if (current.data.compareTo(item) == 0)
      return true;
    current = current.next
  }
return false;
}
```

5) display ()

This method enables us to visualize the elements of the linked list

- Code:

```
public void display () {
  Node current = head;
  while (current != null) {
    System.out.println (current.data.value + " ");
    current = current.next;
  }
system.out.println ();
}
```

6) getHead()

code:

```
public Node getHead(){
    return head;
}
```

7) equals()

This method facilitates the comparison of two linked lists, determining whether their content are identical.

code:

```
public boolean equals (LinkedList list){
  // Different number of nodes
    if (countNodes() != list.countNodes()){
      return false;
    }
  // If the two list are of equal length
    Node current1 = head;
    Node current2 = list.head;
    while ( current1 != null ){
      if(current1.data.compareTo (current2.data) !=0)
        return false
      }
    current1.next;
  }current2.next;
  return true;
}
```

8) copy()

With this method we can create a duplicate of the linked list, preserving the structure and content.

code:

```
public LinkedList copy (){
 LinkedList list2 = new LinkedList ();
 Node current1 = head;
 while ( current1 != null ){
     list2. addTail ( current1. data);
      current1= current1.next;
 }
 return list2
}
```

9) reverse()

code

```
public LinkedList reverse (){
 LinkedList reversedList = new LinkedList ();
 if (isEmpty()) { return reversedList; }
 Node current = head;
 while ( current != null) {
   reversedList. addHead ( current. data);
   current = current .next;
 }
 return reversedList;
}
```

Merging two sorted list

Merging two sorted lists involves combining the elements of two lists while maintaining the sorted order

. approach:

1. Create a new empty linked list to store the merged result.
2. Initialize pointers (current) for the head of the linked lists
3. Compare the current nodes of "list 1" and 'list 2'.
4. take the smaller of the two nodes and add it to the result list (addTail)
5. Move the pointer in the respective list to the next node
6. Continue comparing and adding nodes to the result list until one of the input lists is exhausted
7. If one of the list still remaining nodes append the remaining to the result list
8. Return the result (merged sorted list)

code:

```
public LinkedList mergeSorted(LinkedList list2){
  LinkedList mergedList = new LinkedList();
  Node current1 = head;
  Node current2 = list2.head;
  // Compare and Merge
  while (current1 != null && current2 != null){
    if (current1.data.compareTo(current2.data) < 0){
      mergedList.addTail(current1.data);
      current1 = current1.next;
    }
    else{
      mergedList.addTail(current2.data);
      current2 = current2.next;
    }
  }
  // Append remaining nodes
  while (current1 != null){
    mergedList.addTail(current1.data);
    current1 = current1.next;
  }
  while (current2 != null){
    mergedList.addTail(current2.data);
    current2 = current2.next;
  }
  // Return the result
  return mergedList;
}
```

Sorting a Linked List Using Insert Method

Sorting a linked list is a common operation that involves arranging its element in a specific order, such as ascending or desending. One approach to achieve this is by using the "sorted insert" method, which involves inserting each element into its proper sorted position in the list.

1) Sorted Insert Method

The sorted insert method is a fundamental procedure for inserting a new node into a sorted linked list while preserving the order.

- approach:
  1. Creat a new node with the data to be inserted into the linked list
  2. Base case Handling:
     If the linked list is empty or the new node's data should be place at the beginning, designate the new node as the new head
  3. Traverse the list:
     Otherwise, traverse the list until finding the node after wich the new node should be inserted. This involves comparing the data of the current node with the data of the new node.

4. Insert the New Node

Adjust the 'next' references to insert the new node in its proper sorted position.

Code:

```
public void SortedInsert (NodeData value) {
  Node newNode = new        (value);
  // If the list is empty or the new node should
    be inserted at the beginning
  if (isEmpty () || head.data.compareTo(value)>=0
     addHead (value);
  }
 // Find the node after which the new node should
    be inserted
  else {
  Node current = head;
  while (current.next != null && current.data.compareTo
                                        (value) <0){

    current = current.next;
  }
  // Insert the new Node
    newNode.next = current.next;
    current.next = newNode;
  }
}
```

2) Sorting the entire LinkedList (.sort() method)

We use the sortedInsert method to sort the linked list

.approach

1. Create a new empty linked list in wich we store the result
2. Initialise the pointer (current) to the head
3. Travers the insorted linked list and for each element, perform the sorted insert operation
4. return the result

.code:

```
public LinkedList sort() {
    LinkedList sortedList = new LinkedList();
    Node current = head;

    while (current != null)
        sortedList.sortedInsert(current.data);
        current = current.next;
    }
    return sortedList;
}
```

↳ LinKed List Applications

. Palindrome

A palindrom is a word, number, or sequence that reads the same backwards as forwards.
e.g. damnmad, madam, abcba, 1234321 ...
We can use linKed list to detect palindroms

. approach:

1. Store the original phrase in a linKed list 'list 1', one character per node.
2. Reverse 'list 1' and the reversed list will be 'list 2'
3. Compare 'list 1' with 'list 2' using the method 'equals'. If they are equal, then we have a palindrome.

. Application code :

```
public class Application {
  public static void main (String [] args) {
    Scanner input = new Scanner (System.in)
    LinkedList list1 = new LinkedList ()
    System.out.println ("Enter a String");
    String word = in. .next();
    // Store the characters of the String in a linkedList
    for (int i=0 ; i<word.length(); i++) {
        list1. addTail (new NodeData (word.charAt(i)))
    }
    if (isPalindrome (list1))
      system. out. println (word + "is a palindrom") ;
```

=>

```
else
    System.out.println(word + " is not a palindrom");
} //end main

public static boolean isPalindrom(LinkedList list1){
    LinkedList list2 = list1.reverse();
    return (list1.equals(list2));
}

} //end Application
```

Anagram

An anagram is a word or phrase formed by rearranging the letters of another word or phrase, typically using all the original letters exactly once. For example "listen and "silent" are anagrams
we can use linked list to detect anagrams

approach:

1. Define a method 'splitword' that take a String as input and create a linked list where each node represents a character in the string.
2. In the 'aseAnagrams' method, first we create two linked lists by splitting the input strings using 'splitWord' method

3. Compare the nb of nodes in both linked lists IF they are not equal, the word cannot be anagram, so return 'false'
4. Otherwise, while the first list is not empty, get the data of the current node in the first list. Search for the data in the second list using the 'search' method. IF the data is not found in l2, return false (not anagrams).
IF the data is found, delete the head of l1
5. IF the loop completes without returning 'false' it means that each character in l1 was found in l2.
6. Return 'true' to indicate that the word are anagrams.

- Application Code:

```
public class Anagrams {                    public static void main(String[] args)
    Scanner input = new Scanner (System.in);
    // Scan words to be detected
      System.out.print ("Enter the first word");
      String word1 = input.next();
      System.out.print("Enter the second word");
      String word2 = input.next();
    // Check if words are anagrams
      if (areAnagram (word1, word2))
          System.out.println (word1 + "and" + word2 + "are anag
      else
          System.out.println (word1 + "and" + word2 + "are not anagrams");
} // end main
```

=>

```
public static LinkedList splitWord (String word){
  LinkedList result = new LinkedList();
  for (int i = 0 ; i<word.length() ; i++){
    result.addTail (new NodeData (word.charAt(i)));
  }
  return result;
}

public static boolean areAnagram (String s1, String s2){
 LinkedList l1 = splitWord (s1);
 LinkedList l2 = splitWord (s2);
 // Check if the two words have the same nb of letters
 if (l1.countNode() != l2.countNodes ()){
    return false;
 }
 else {
 while (! l1.isEmpty()){
    boolean searchResult = l2.search(l1.head.data);
    if (SearchResult == false )
       return false;
    else
       l1.deleteHead();
 }
}
return true ;
}
} // end Application
```