Week 7: Linked List

4009 DATA STRUCTURES & ALGORITHM (DSA)

Learning Outcomes

- Understand and implement Linked Lists in Java
- Identity and explain when linked list should be used instead of Arrays
- Able to create, access and understand either Singly Linked List or Doubly Linked List
- Differentiate between a Singly Linked List and a Doubly Linked List
- Understand and compare how computers store variables, arrays and linked list in memory (RAM)

Content Overview

- Data structure trade-offs
- Linked list introduction
- Linked list key-terms
- Linked List
- Computer Memory (RAM)
- Variables in Memory
- Arrays in Memory
- Linked List in Memory
- Different types of Linked Lists
- Singly Linked List
- Doubly Linked List

Data Structure Trade-offs

Data structures	Advantages	Disadvantages
Array	Quick insertion, very fast access to index known	Slow search, slow deletion, fixed size
Ordered Array	Quicker Search than unsorted array	Slow insertion and deletion, fixed size
Stack	Provide Last-in First-out access	Slow access to other items
Queue	Provide First-in First out access	Slow access to other items
Linked List	Quick Insertion, Quick deletion	Slow search
Binary Tree	Quick search, insertion, deletion (if tree balanced)	Deletion algorithm is complex
Hash Table	Very fast access if key known, fast insertion	Slow deletion, access slow if key not known, inefficient memory usage
Heap	Fast insertion, deletion, access to large item	Slow access to other items
Graphs	Model real world situations	Some algorithms are slow and complex

Linked List introduction

Arrays had certain disadvantages as data storage structures. In an unordered array, searching is slow, whereas in an ordered array, insertion is slow. In both kinds of arrays, deletion is slow. Also, the size of an array can't be changed after it's created.

Linked list is the solution to disadvantages of Array as data storage structures.

Linked lists are probably the second most commonly used general purpose storage structures after arrays.

The linked list is a versatile mechanism suitable for use in many kinds of general-purpose databases. It can also replace an array as the basis for other storage structures such as stacks and queues.

Linked lists aren't the solution to all data storage problems, but they are surprisingly versatile and conceptually simpler than some other popular structures such as trees.

Linked List key-terms

Node: A basic unit of a Linked List containing data and a reference to the next node.

Head: The first node in the Linked List.

Tail: The last node, which points to null.

Next Pointer: A reference or link to the next node in the list.

Each node stores two things:

1. data – the value or element of the node
2. next – a reference to the next node in the list.

Linked List Definitions

First definition: A linked list class is a collection which can contain many objects of the same type like an ArrayList (chapter 7 – DSA book 2). LinkedList implement the List interface and have methods to add items, change items, remove items and clear the list.

Second definition: A linked list is a collection of nodes that together form a linear ordering. The ordering is determined as in the child's game "follow the leader" to which each node is an object that stores a reference to an element and a reference called next, to another node.

Links and nodes

As shown in figure 1 below, there are four nodes and five links. Starting from the head node, each node references another node using a link or pointer. Only the tail node references null which indicates the end of the list.

Moving from one node to another by following a next reference is known as link hopping or pointer hopping

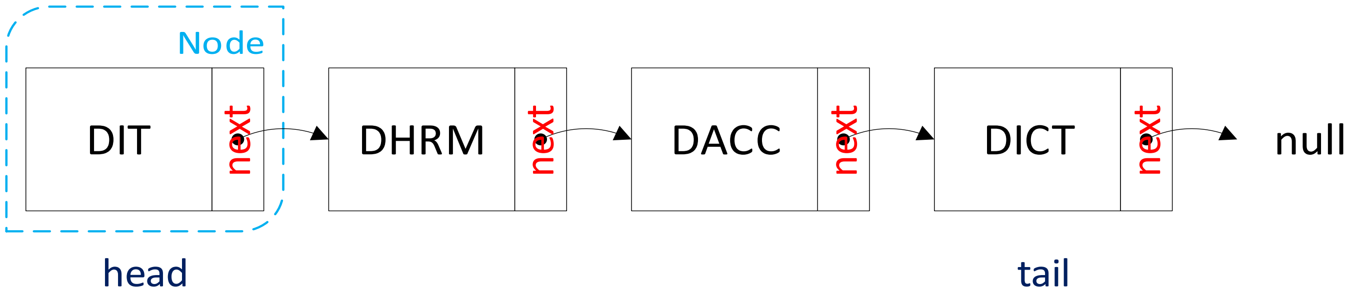


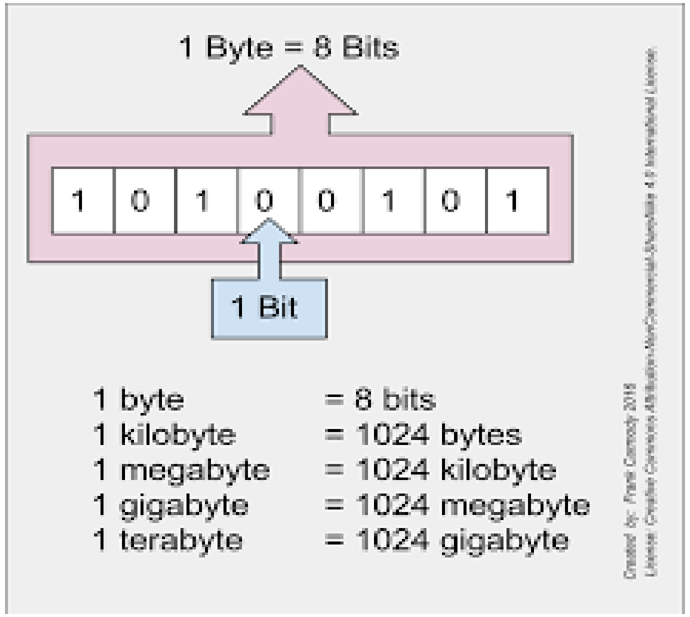
Figure 1: Diagram relating to Java code on slide 8

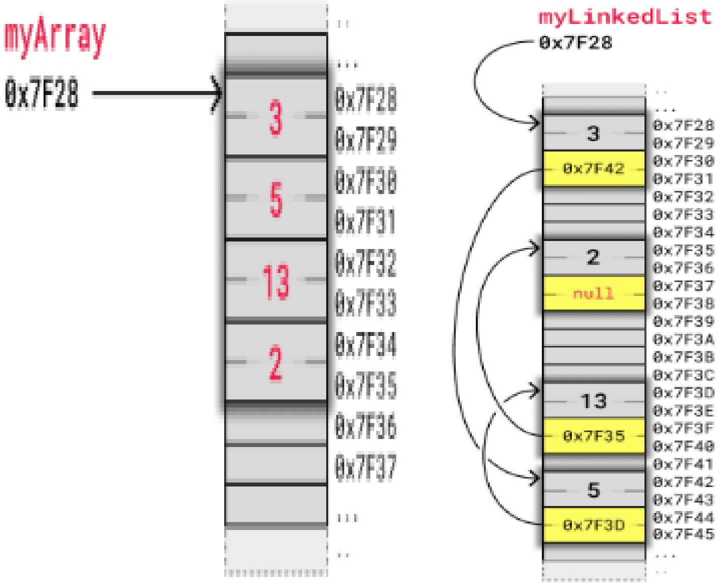
```java
// Import the LinkedList class
import java.util.LinkedList;

public class LinkedListExample {

  public static void main(String[] args) {

    //Create Linked List object called itiCourses
    LinkedList<String> itiCourses = new LinkedList<String>();

       //add nodes into the Linked List
       itiCourses.add("DIT");
       itiCourses.add("DHRM");
       itiCourses.add("DACC");
       itiCourses.add("DICT");

       //print to the console the Linked List
       System.out.println(itiCourses);
    }
}
```

Figure 2: Implementation of Linked Link in JGrasp

Computer Memory (RAM)

Computer memory or Random-Access Memory (RAM) is where your variables, arrays and linked lists are stored on a computer.



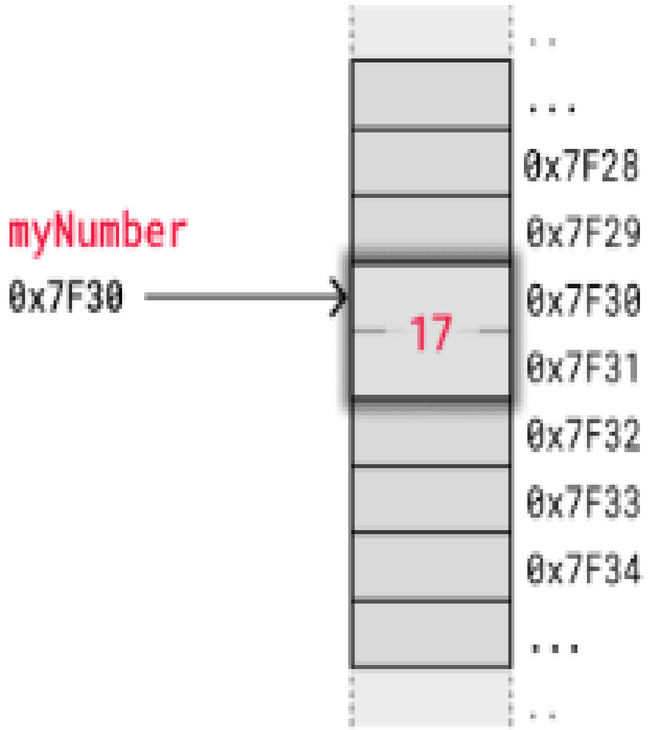


Variable in Memory

Let's imagine that we want to store the integer "17" in a variable myNumber. For simplicity, let's assume the integer is stored as two bytes (16 bits), and the address in memory to myNumber is 0x7F30.

0x7F30 is actually the address to the first of the two bytes of memory where the myNumber integer value is stored. When the computer goes to 0x7F30 to read an integer value, it knows that it must read both the first and the second byte, since integers are two bytes on this specific computer.

The image below shows how the variable myNumber = 17 is stored in memory.



Source: [DSA Linked Lists in Memory]

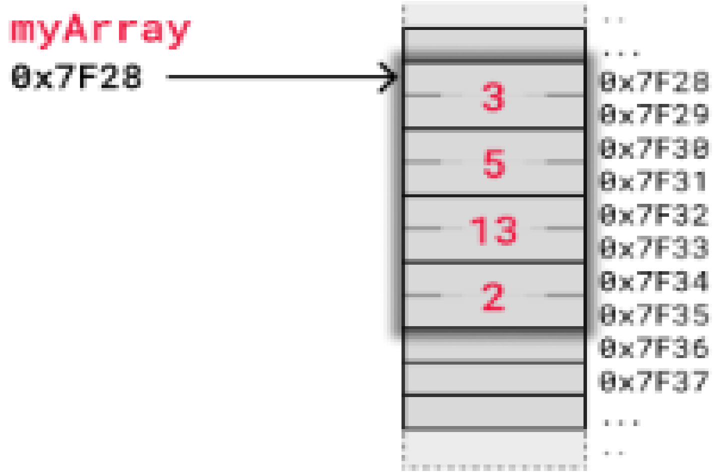
Arrays in memory

Source: DSA Linked Lists in Memory

To understand linked lists, it is useful to first know how arrays are stored in memory.

Elements in an array are stored contiguously in memory. That means that each element is stored right after the previous element.

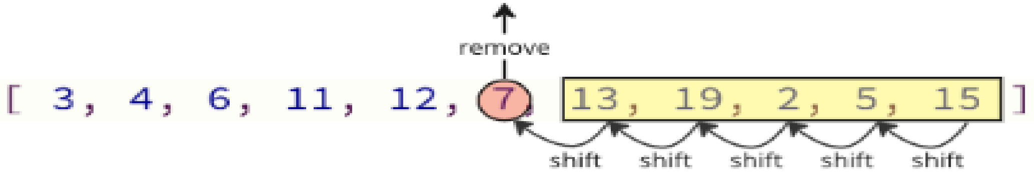
The image below shows how an array of integers myArray = [3,5,13,2] is stored in memory. We use a simple kind of memory here with two bytes for each integer, like in the previous example, just to get the idea.



The computer has only got the address of the first byte of myArray, so to access the 3rd element with code myArray[2] the computer starts at 0x7F28 and jumps over the two first integers. The computer knows that an integer is stored in two bytes, so it jumps 2x2 bytes forward from 0x7F28 and reads value 13 starting at address 0x7F32.

When removing or inserting elements in an array, every element that comes after must be either shifted up to make place for the new element, or shifted down to take the removed element's place. Such shifting operations are time consuming and can cause problems in real-time systems for example.

The image below shows how elements are shifted when an array element is removed.



Linked List in Memory (1/2)

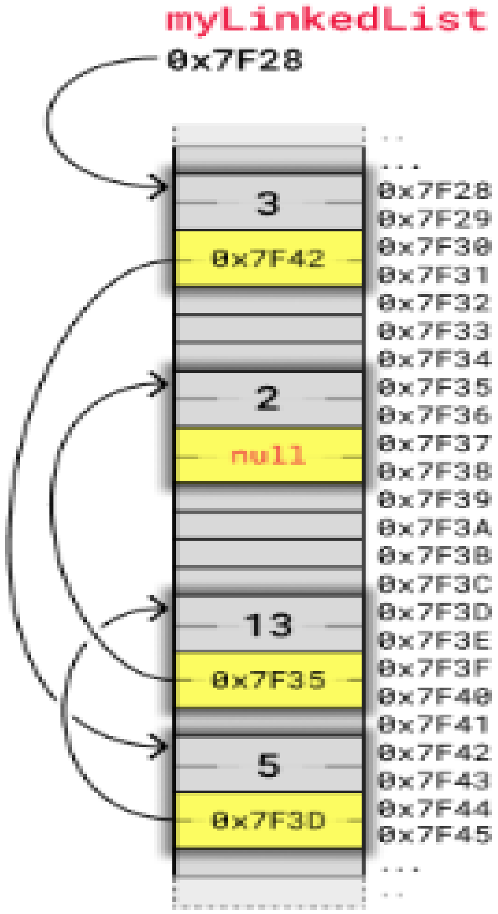
Instead of storing a collection of data as an array, we can create a linked list.

Linked lists are used in many scenarios, like dynamic data storage, stack and queue implementation or graph representation, to mention some of them.

A linked list consists of nodes with some sort of data, and at least one pointer, or link, to other nodes.

A big benefit with using linked lists is that nodes are stored wherever there is free space in memory, the nodes do not have to be stored contiguously right after each other like elements are stored in arrays. Another nice thing with linked lists is that when adding or removing nodes, the rest of the nodes in the list do not have to be shifted.

The image below shows how a linked list can be stored in memory. The linked list has four nodes with values 3, 5, 13 and 2, and each node has a pointer to the next node in the list.

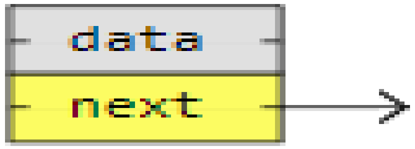


Source: DSA Linked Lists in Memory

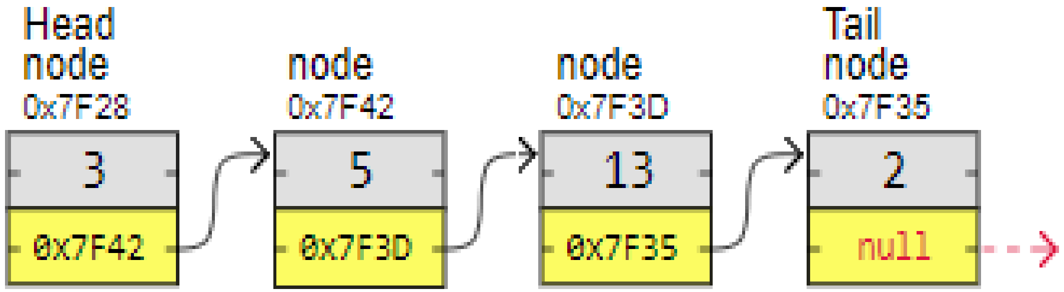
Linked List in Memory (2/2)

Each node takes up four bytes. Two bytes are used to store an integer value, and two bytes are used to store the address to the next node in the list.

To make it easier to see how the nodes relate to each other, we will display nodes in a linked list in a simpler way, less related to their memory location, like in the image below:



If we put the same four nodes from the previous example together using this new visualization, it looks like this:



Source: [DSA Linked Lists in Memory]

Different types of Linked Lists

Singly Linked List: A list where each node has a reference to the next node.

Doubly Linked List: A list where each node has references to both the next and the previous node.

Circular Linked List: A variation where the last node points back to the first node, forming a circle.

Note we will teach only ***Singly and doubly linked list in this course***.

Singly Linked List

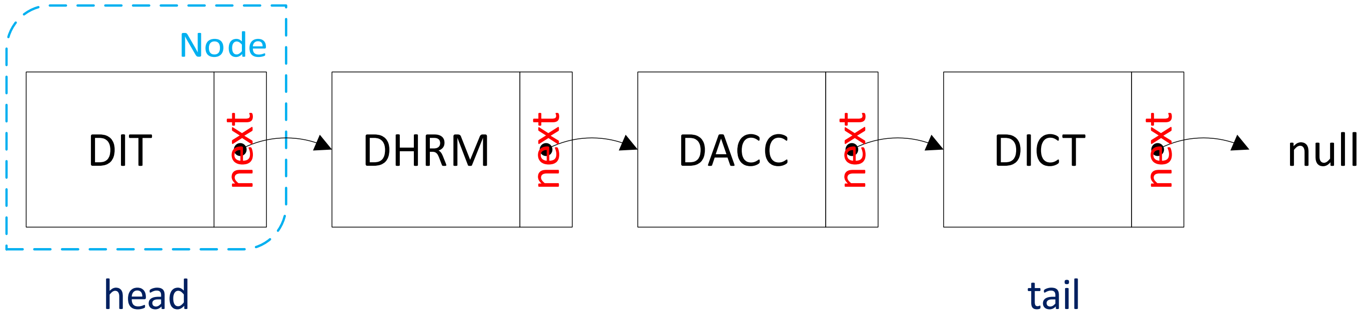
A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer.

Singly linked list is the simplest kind of linked list that takes up less space in memory because each node has only one address to the next node.

Each node stores:

Data also known as element or value of the node. For example: Head node's data is assigned the String value = DIT.

Link to the next node. Example: Head node is linked to next node which holds DHRM data.



Implementing a Singly Linked List (1/5)

```
1  class Node {
2     String data;
3     Node next;
4
5     // Constructor to create a new node
6     public Node(String data) {
7        this.data = data;
8        this.next = null;  // Initially, the next is null
9     }
10 }
11
```

Define Node class that contains data of type String and Node next to link the next node. Followed is the Node constructor to define Node's variables and assigned values.

Note: Node class statement block starts at line 1 and ends on line 10.

Implementing a Singly Linked List (2/5)

Next, we define public class SinglyLinkedList.

Note: SinglyLinkedList class statement block starts at line 12 and ends at line 107.

This class creates the nodes in the Singly linked list and defines the methods to be used to manipulate the Linked List.

Firstly, there is a constructor method that defines the head assigned to null;

There are five methods:

1. insertAtHead(String data) = method to insert data into the head nodes.
2. InsertAtEnd(String data) = method to insert data into the tail nodes.
3. display() = method to display full Linked List to the console
4. deleteByValue(String value) = method to delete a node by the value or data passed in the method's parameter
5. size() = method to calculate the size of Linked List

Implementing a Singly Linked List (3/5)

```java
public class SinglyLinkedList {
    Node head;  // Head of the list (points to the first node)

    // Constructor
    public SinglyLinkedList() {
        this.head = null;
    }

    // Method to insert a node at the beginning
    public void insertAtHead(String data) {
        Node newNode = new Node(data);
        newNode.next = head;  // New node points to the previous head
        head = newNode;       // Head is updated to the new node
    }

    // Method to insert a node at the end
    public void insertAtEnd(String data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;  // If the list is empty, make new node the head
            return;
        }
        Node temp = head;
        while (temp.next != null) {
            temp = temp.next;  // Traverse until the last node
        }
        temp.next = newNode;  // Link the last node to the new node
    }

    // Method to display the list
    public void display() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("null");  // End of the list
    }

    // Method to delete a node by value
    public void deleteByValue(String value) {
        if (head == null) {
            System.out.println("List is empty.");
            return;
        }

        // If the head is the node to be deleted
        if (head.data == value) {
            head = head.next;
            return;
        }

        Node temp = head;
        while (temp.next != null) {
            if (temp.next.data == value) {
                temp.next = temp.next.next;  // Bypass the node to delete
                return;
            }
            temp = temp.next;
        }
        System.out.println("Value " + value + " not found in the list.");
    }

    // Method to find the size of the list
    public int size() {
        int size = 0;
        Node temp = head;
        while (temp != null) {
            size++;
            temp = temp.next;
        }
        return size;
    }
```

Implementing a Singly Linked List (4/5)

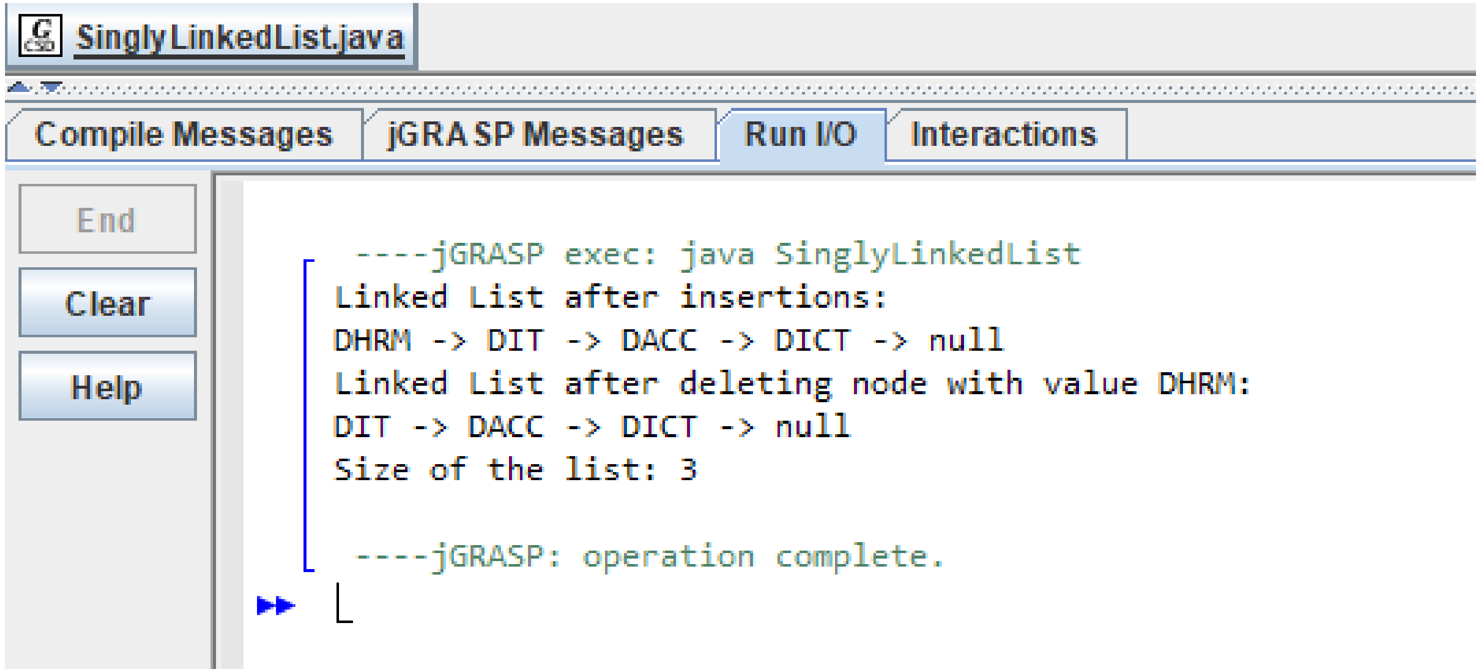
Lastly, we create the main method of the program.

Note: The main method statement block starts at line 87 and ends at line 106.

The main method will call the SinglyLinkedList class and execute the methods in the SinglyLinkedList class.

```java
87      public static void main (String[] args) {
88          SinglyLinkedList list = new SinglyLinkedList();
89
90          // Insert nodes at the head and end
91          list.insertAtHead("DIT");     // List: DIT
92          list.insertAtHead("DHRM");     // List: DIT -> DHRM
93          list.insertAtEnd("DACC");      // List: DIT -> DHRM -> DACC
94          list.insertAtEnd("DICT");      // List: DIT -> DHRM -> DACC -> DICT
95
96          System.out.println("Linked List after insertions:");
97          list.display();  // Expected output: DIT -> DHRM -> DACC -> DICT -> null
98
99          // Delete a node
100         list.deleteByValue("DHRM");  // List: DIT -> DACC -> DICT
101         System.out.println("Linked List after deleting node with value DHRM:");
102         list.display();  // Expected output: DIT -> DACC -> DICT -> null
103
104         // Print size of the list
105         System.out.println("Size of the list: " + list.size());  // Expected output: 3
106     }
107 }
```

Implementing a Singly Linked List (5/5)



Output from SinglyLinkedList program

Doubly Linked List

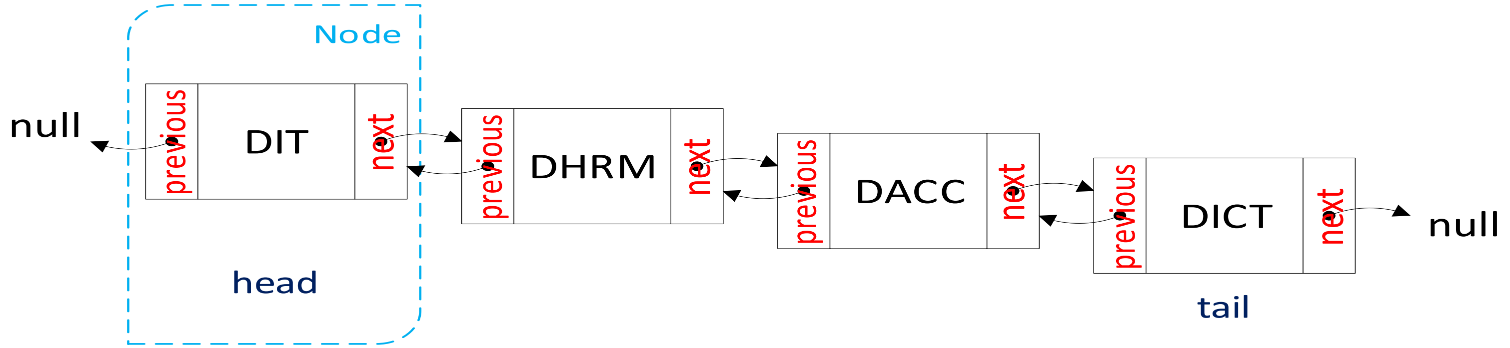
Doubly linked list has nodes with addresses to both the previous and next node, hence, takes up more memory. However, this is useful when locating data up and down the list.

Each node stores:

Data also known as element or value of the node. For example: Head node's data is assigned the String value = DIT.

Link to the next node. Example: the second node (holds DHRM data) is linked to next node that nodes the DACC data.

Link to the previous node. Example: the second node (holds DHRM data) is linked to previous node that nodes the DIT data.



Implementing a Doubly Linked List (1/5)

To implement a doubly linked list like a singly linked list we must first define a Node class which specifies the type of objects stored at the nodes of the list.
Note: If you would to implement the code below, save the program file as DoublyLinkedList.java in JGrasp.

```
1  class Node {
2     String data;
3     Node next;
4     Node prev;  // Reference to the previous node
5
6     public Node(String data) {
7        this.data = data;
8        this.next = null;
9        this.prev = null;  // Initially, the prev is null
10    }
11 }
```

Define Node class that contains data of type String, Node next to link the next node and node prev to link the previous node. Followed is the Node constructor to define Node's variables and assigned values.

Note: Node class statement block starts at line 1 and ends on line 11.

Implementing a Doubly Linked List (2/5)

Next, we define public class DoublyLinkedList.

Note: DoublyLinkedList class statement block starts at line 12 and ends at line 145.

This class creates the nodes in the Doubly linked list and defines the methods to be used to manipulate the Linked List.

Firstly, there is a constructor method that defines the head and the tail assigned to null;

There are five methods:

1. insertAtHead(String data) = method to insert data into the head nodes.
2. InsertAtEnd(String data) = method to insert data into the tail nodes.
3. deleteByValue(String value) = method to delete a node by the value or data passed in the method’s parameter
4. displayForward() = method to display the full Linked List from head to tail printing to the console
5. displayForward() = method to display the full Linked List from tail to head (reverse order) printing to the console

Implementing a Doubly Linked List (3/5)

```java
public class DoublyLinkedList {
    Node head;  // Head of the list
    Node tail;  // Tail of the list

    // Constructor to initialize the list
    public DoublyLinkedList() {
        this.head = null;
        this.tail = null;
    }

    // Method to insert a node at the beginning
    public void insertAtHead(String data) {
        Node newNode = new Node(data);
        if (head == null) {  // If the list is empty
            head = newNode;
            tail = newNode;
        } else {
            newNode.next = head;  // New node points to the current head
            head.prev = newNode;  // Current head points back to the new node
            head = newNode;       // Head is updated to the new node
        }
    }

    // Method to insert a node at the end
    public void insertAtEnd(String data) {
        Node newNode = new Node(data);
        if (tail == null) {  // If the list is empty
            head = newNode;
            tail = newNode;
        } else {
            tail.next = newNode;  // Last node's next points to the new node
            newNode.prev = tail;  // New node's prev points to the current tail
            tail = newNode;       // Tail is updated to the new node
        }
    }

    // Method to delete a node by value
    public void deleteByValue(String value) {
        if (head == null) {
            System.out.println("List is empty.");
            return;
        }

        Node temp = head;
        // If the node to be deleted is the head node
        if (head.data.equals(value)) {
            head = head.next;
            if (head != null) {
                head.prev = null;  // Head's previous is null after deletion
            }
            return;
        }

        // Traverse the list to find the node to delete
        while (temp != null) {
            if (temp.data.equals(value)) {
                // If the node to be deleted is the tail node
                if (temp == tail) {
                    tail = temp.prev;
                    tail.next = null;
                } else {
                    temp.prev.next = temp.next;  // Bypass the node to delete
                    if (temp.next != null) {
                        temp.next.prev = temp.prev;
                    }
                }
                return;
            }
            temp = temp.next;
        }
        System.out.println("Value '" + value + "' not found in the list.");
    }

    // Method to display the list from head to tail
    public void displayForward() {
        if (head == null) {
            System.out.println("The list is empty.");
            return;
        }

        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " <-> ");
            temp = temp.next;
        }
        System.out.println("null");
    }
```

Implementing a Doubly Linked List (4/5)

```java
100 // Method to display the list from tail to head (reverse order)
101 public void displayBackward() {
102    if (tail == null) {
103       System.out.println("The list is empty.");
104       return;
105    }
106
107    Node temp = tail;
108    while (temp != null) {
109       System.out.print(temp.data + " <-> ");
110       temp = temp.prev;
111    }
112    System.out.println("null");
113 }
114
```

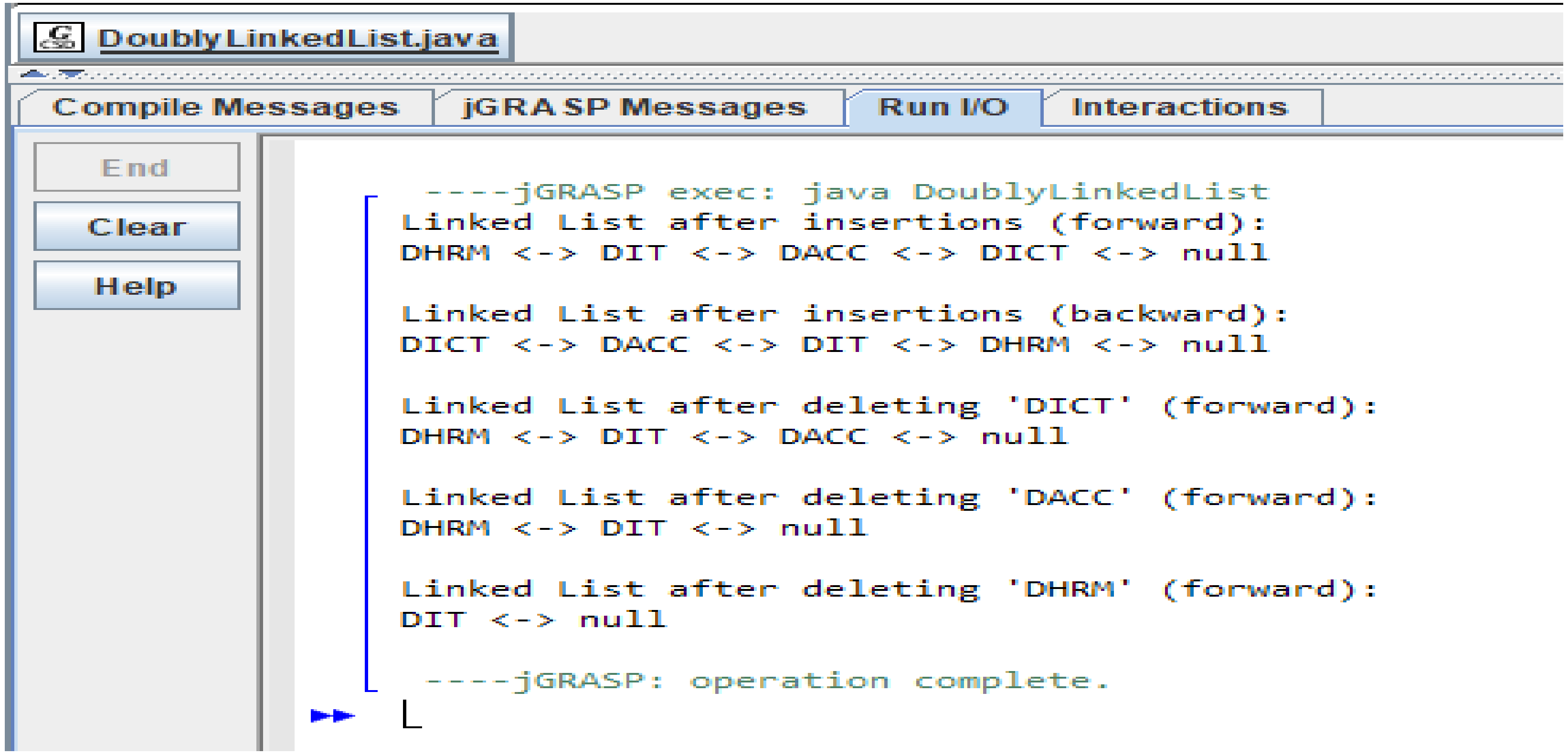
```java
115    public static void main (String[] args) {
116       DoublyLinkedList list = new DoublyLinkedList();
117
118       // Insert nodes at the head and end
119       list.insertAtHead("DIT");    // List: DIT
120       list.insertAtHead("DHRM");    // List: DIT -> DHRM
121       list.insertAtEnd("DACC");     // List: DIT -> DHRM -> DACC
122       list.insertAtEnd("DICT");     // List: DIT -> DHRM -> DACC -> DICT
123
124       System.out.println("Linked List after insertions (forward):");
125       list.displayForward();  // Expected output: DHRM <-> DIT <-> DACC <-> DICT <-> null
126
127       System.out.println("\nLinked List after insertions (backward):");
128       list.displayBackward();  // Expected output: DICT <-> DACC <-> DIT <-> DHRM <-> null
129
130       // Delete a node
131       list.deleteByValue("DICT");  // List: DHRM <-> DIT <-> DACC
132       System.out.println("\nLinked List after deleting 'DICT' (forward):");
133       list.displayForward();  // Expected output: DHRM <-> DIT <-> DACC <-> null
134
135       // Delete the last node (tail)
136       list.deleteByValue("DACC");   // List: DHRM <-> DIT
137       System.out.println("\nLinked List after deleting 'DACC' (forward):");
138       list.displayForward();  // Expected output: DHRM <-> DIT <-> null
139
140       // Delete the first node (head)
141       list.deleteByValue("DHRM"); // List: DIT
142       System.out.println("\nLinked List after deleting 'DHRM' (forward):");
143       list.displayForward();  // Expected output: DIT <-> null
144    }
145 }
```

Lastly, we create the main method of the program.

Note: The main method statement block starts at line 115 and ends at line 144.

The main method will call the DoublyLinkedList class and execute the methods in the DoublyLinkedList class

Implementing a Doubly Linked List (5/5)



Output from DoublyLinkedList program

Topic next week

Week 8 Stacks & Queue

DSA Week 7 activities

Refer to print out materials.

- This week, you are required to complete the questions and two labs.
  - In your DSA textbook 1, answer questions 1, 2, 3, 4, 5, 9 & 13
  - Refer to the print out, answer all week 7 questions.
  - Refer to the print out, complete the two labs activities using the lab computers.

Note: You can complete the activities in any order, however, make afford to complete and understand everything which prepares you for well for test 2, Mid Semester Exam & Final Exam.