**Day13 – June 28th**

**TASK1 – Pointers**

**#include <iostream>**

**int main() {**

**int age; // variable**

**int \*ptr;// pointer variable**

**age = 10;**

**ptr = &age;**

**std::cout<<"value of age "<<age;**

**std::cout<<"ptr is pointing to"<<\*ptr;**

**std::cout<<"address of age "<<&age;**

**std::cout<<"value of ptr "<<ptr;**

**std::cout<<"value of &ptr "<<&ptr;**

**return 0;**

**}**

**Output**

value of age 10

ptr is pointing to10

address of age 0x7ffdc59e59dc

value of ptr 0x7ffdc59e59dc

value of &ptr 0x7ffdc59e59d0

**TASK2 – Linked List and its Operations**

**#include <bits/stdc++.h>**

**using namespace std;**

**// Define a Node class**

**class Node {**

**public:**

**int data; // Data part of the node**

**Node\* next; // Pointer to the next node**

**// Constructor for convenience**

**Node(int value) : data(value), next(nullptr) {}**

**};**

**// Class for singly linked list**

**class LinkedList {**

**private:**

**Node\* head; // Pointer to the head of the list**

**public:**

**// Constructor to initialize an empty list**

**LinkedList() {**

**head = nullptr;**

**}**

**// Function to insert a node at the end**

**void insertAtEnd(int value) {**

**Node\* newNode = new Node(value);**

**if (head == nullptr) {**

**head = newNode; // If list is empty, make newNode the head**

**} else {**

**Node\* temp = head;**

**while (temp->next != nullptr) {**

**temp = temp->next; // Traverse to the last node**

**}**

**temp->next = newNode; // Link the last node to newNode**

**}**

**}**

**// Function to delete a node by value**

**void deleteByValue(int value) {**

**if (head == nullptr) return;**

**if (head->data == value) {**

**Node\* temp = head;**

**head = head->next; // Move head to next node**

**delete temp;**

**return;**

**}**

**Node\* temp = head;**

**while (temp->next && temp->next->data != value) {**

**temp = temp->next;**

**}**

**if (temp->next) {**

**Node\* nodeToDelete = temp->next;**

**temp->next = temp->next->next;**

**delete nodeToDelete;**

**}**

**}**

**// Function to display the list**

**void display() {**

**Node\* temp = head;**

**while (temp != nullptr) {**

**cout << temp->data << " -> ";**

**temp = temp->next;**

**}**

**cout << "NULL" << endl;**

**}**

**// Destructor to free all allocated memory**

**~LinkedList() {**

**Node\* temp;**

**while (head) {**

**temp = head;**

**head = head->next;**

**delete temp;**

**}**

**}**

**};**

**// Main function**

**int main() {**

**LinkedList list;**

**list.insertAtEnd(10);**

**list.insertAtEnd(20);**

**list.insertAtEnd(30);**

**cout << "Linked List: ";**

**list.display();**

**list.deleteByValue(20);**

**cout << "After Deleting 20: ";**

**list.display();**

**return 0;**

**}**

**Output**

Linked List: 10 -> 20 -> 30 -> NULL

After Deleting 20: 10 -> 30 -> NULL

----------------------------------------------------------------------------------------------------------------------------------------

**TASK 3 – Linkedlist in JAVA**

package Day13;  
  
class Node {  
 int data;  
 Node next;  
  
 public Node(int data)  
 {  
 this.data = data;  
 this.next = null;  
 }  
}  
class LinkedList1{  
 private Node head;  
  
 // Insert at the end  
 public void insertAtEnd(int value) {  
 Node newNode = new Node(value);  
 if (head == null) {  
 head = newNode;  
 } else {  
 Node temp = head;  
 while (temp.next != null) {  
 temp = temp.next;  
 }  
 temp.next = newNode;  
 }  
 }  
  
 // Delete by value  
 public void deleteByValue(int value) {  
 if (head == null) return;  
  
 if (head.data == value) {  
 head = head.next;  
 return;  
 }  
  
 Node temp = head;  
 while (temp.next != null && temp.next.data != value) {  
 temp = temp.next;  
 }  
  
 if (temp.next != null) {  
 temp.next = temp.next.next;  
 }  
 }  
  
 // Display the list  
 public void display() {  
 Node temp = head;  
 while (temp != null) {  
 System.*out*.print(temp.data + " -> ");  
 temp = temp.next;  
 }  
 System.*out*.println("NULL");  
 }  
}  
  
class TASK3\_LinkedListDemo {  
 public static void main(String[] args) {  
 LinkedList1 list = new LinkedList1();  
  
 list.insertAtEnd(10);  
 list.insertAtEnd(20);  
 list.insertAtEnd(30);  
  
 System.*out*.println("Linked List:");  
 list.display();  
  
 list.deleteByValue(20);  
  
 System.*out*.println("After deleting 20:");  
 list.display();  
 }  
}

**Output**

Linked List:

10 -> 20 -> 30 -> NULL

After deleting 20:

10 -> 30 -> NULL

**TASK4 – Node with accepts any type of data**

package Day13;  
  
import java.util.\*;  
import java.util.ArrayList;  
class Node1<T> {  
 T data;  
 Node1<T> next;  
  
 public Node1(T data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
public class TASK4\_LinkedListWithAnyDataType<T> {  
 private Node1<T> head;  
 private int size = 0;  
  
 public void add(T data) {  
 Node1<T> newNode = new Node1<>(data);  
 if (head == null) {  
 head = newNode;  
 } else {  
 Node1<T> current = head;  
 while (current.next != null) {  
 current = current.next;  
 }  
 current.next = newNode;  
 }  
 size++;  
 }  
  
 public void addFirst(T data) {  
 Node1<T> newNode = new Node1<>(data);  
 newNode.next = head;  
 head = newNode;  
 size++;  
 }  
  
 public T removeFirst() {  
 if (head == null) {  
 throw new NoSuchElementException("List is empty");  
 }  
 T removedData = head.data;  
 head = head.next;  
 size--;  
 return removedData;  
 }  
  
 public T get(int index) {  
 checkBounds(index);  
 Node1<T> current = head;  
 for (int i = 0; i < index; i++) {  
 current = current.next;  
 }  
 return current.data;  
 }  
  
 public int size() {  
 return size;  
 }  
  
 private void checkBounds(int index) {  
 if (index < 0 || index >= size) {  
 throw new IndexOutOfBoundsException("Index out of bounds");  
 }  
 }  
}  
  
  
class Task004\_DS\_CustomLinkedList {  
 public static void main(String[] args) {  
 TASK4\_LinkedListWithAnyDataType<String> liobj = new TASK4\_LinkedListWithAnyDataType<>();  
 liobj.add("Anitha");  
 liobj.add("Verma");  
 liobj.addFirst("Jack");  
  
 System.*out*.println("First Element: " + liobj.get(0));  
 System.*out*.println("Size: " + liobj.size());  
  
 liobj .removeFirst();  
  
 System.*out*.println("First Element after removal: " + liobj.get(0));  
 System.*out*.println("Size after removal: " + liobj.size());  
 }  
}

**Output**

First Element: Jack

Size: 3

First Element after removal: Anitha

Size after removal: 2

**TASK5- LinkedList with all Operations**

package Day13;  
  
// Node class (Generic)  
class Node2<T> {  
 T data;  
 Node2<T> next;  
  
 public Node2(T data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
  
// Generic Linked List  
class LinkedList2<T> {  
 private Node2<T> head;  
 private int size;  
  
 public LinkedList2() {  
 head = null;  
 size = 0;  
 }  
  
 // Add element at end  
 public void add(T value) {  
 Node2<T> newNode = new Node2<>(value);  
 if (head == null) {  
 head = newNode;  
 } else {  
 Node2<T> temp = head;  
 while (temp.next != null)  
 temp = temp.next;  
 temp.next = newNode;  
 }  
 size++;  
 }  
  
 // Remove node by value  
 public void remove(T value) {  
 if (head == null) return;  
  
 if (head.data.equals(value)) {  
 head = head.next;  
 size--;  
 return;  
 }  
  
 Node2<T> current = head;  
 while (current.next != null && !current.next.data.equals(value)) {  
 current = current.next;  
 }  
  
 if (current.next != null) {  
 current.next = current.next.next;  
 size--;  
 } else {  
 System.*out*.println("Element not found: " + value);  
 }  
 }  
  
 // Display all elements  
 public void display() {  
 Node2<T> temp = head;  
 while (temp != null) {  
 System.*out*.print(temp.data + " → ");  
 temp = temp.next;  
 }  
 System.*out*.println("NULL");  
 }  
  
 // Return size  
 public int getSize() {  
 return size;  
 }  
  
 // Get value by index (with index check)  
 public T get(int index) {  
 if (index < 0 || index >= size)  
 throw new IndexOutOfBoundsException("Index out of bounds: " + index);  
  
 Node2<T> temp = head;  
 for (int i = 0; i < index; i++)  
 temp = temp.next;  
  
 return temp.data;  
 }  
 public class TASK5\_LinkedListWithOpertaions {  
 public static void main(String[] args) {  
 LinkedList2<String> list = new LinkedList2<>();  
  
 list.add("Apple");  
 list.add("Banana");  
 list.add("Cherry");  
  
 System.*out*.println("List elements:");  
 list.display(); // Apple → Banana → Cherry → NULL  
  
 list.remove("Banana");  
 System.*out*.println("After removing Banana:");  
 list.display(); // Apple → Cherry → NULL  
  
 System.*out*.println("Size of list: " + list.getSize()); // 2  
  
 try {  
 System.*out*.println("Element at index 1: " + list.get(1)); // Cherry  
 System.*out*.println("Element at index 5: " + list.get(5)); // Throws exception  
 } catch (IndexOutOfBoundsException e) {  
 System.*out*.println(e.getMessage());  
 }  
 }  
 }  
  
}

**Output**

List elements:

Apple → Banana → Cherry → NULL

After removing Banana:

Apple → Cherry → NULL

Size of list: 2

Element at index 1: Cherry

Index out of bounds: 5

**TASK5\_1 – List down all the methods in Linked List**

| Method Name | Description |
| --- | --- |

|  |  |
| --- | --- |
| add(E e) | Adds element to the end of the list |

|  |  |
| --- | --- |
| add(int index, E e) | Adds element at a specific index |

|  |  |
| --- | --- |
| remove(Object o) | Removes the first occurrence of the element |

|  |  |
| --- | --- |
| remove(int index) | Removes the element at the specified position |

|  |  |
| --- | --- |
| get(int index) | Retrieves the element at the specified index |

|  |  |
| --- | --- |
| set(int index, E e) | Replaces the element at the specified index |

|  |  |
| --- | --- |
| clear() | Removes all elements |

**TASK6 – Create linked list using Pre defined class and add elements to it.**

package Day13;  
import java.util.LinkedList;  
  
public class TASK6\_PredefinedLinkedList {  
 public static void main(String[] args) {  
 // Create a LinkedList of Strings  
 LinkedList<String> fruits = new LinkedList<>();  
  
 // Add elements to the LinkedList  
 fruits.add("Apple");  
 fruits.add("Banana");  
 fruits.add("Mango");  
 fruits.add("Grapes");  
  
 // Display the elements  
 System.*out*.println("LinkedList elements: " + fruits);  
 }  
}

**Output**

LinkedList elements: [Apple, Banana, Mango, Grapes]

**TASK7 – Get 1st and last elements**

package Day13;  
import java.util.LinkedList;  
  
public class TASK6\_PredefinedLinkedList {  
 public static void main(String[] args) {  
 // Create a LinkedList of Strings  
 LinkedList<String> fruits = new LinkedList<>();  
  
 // Add elements to the LinkedList  
 fruits.add("Apple");  
 fruits.add("Banana");  
 fruits.add("Mango");  
 fruits.add("Grapes");  
 System.*out*.println("LinkedList elements: " + fruits);  
 System.*out*.println("1st element of Linkedlist : " + fruits.get(0));  
 System.*out*.println("1st element of Linkedlist : " + fruits.get(2));  
  
 }  
}

**Output**

LinkedList elements: [Apple, Banana, Mango, Grapes]

1st element of Linkedlist : Apple

last element of Linkedlist : Grapes

**TASK8 – Remove 2 elements and display remaining.**

package Day13;  
  
import java.util.LinkedList;  
  
public class TASK8\_RemoveElements {  
 public static void main(String[] args) {  
 // Create a LinkedList of Strings  
 LinkedList<String> fruits = new LinkedList<>();  
  
 // Add elements to the LinkedList  
 fruits.add("Apple");  
 fruits.add("Banana");  
 fruits.add("Mango");  
 fruits.add("Grapes");  
 System.*out*.println("LinkedList elements: " + fruits);  
 System.*out*.println("Remove 1st element of Linkedlist : " + fruits.remove(0));  
 System.*out*.println("Remove last element of Linkedlist : " + fruits.remove(2));  
  
 }  
}

**Output**

LinkedList elements: [Apple, Banana, Mango, Grapes]

Remove 1st element of Linkedlist : Apple

Remove last element of Linkedlist : Grapes

**(or)**

package Day13;  
  
import java.util.LinkedList;  
  
public class TASK8\_RemoveElements {  
 public static void main(String[] args) {  
 // Create a LinkedList of Strings  
 LinkedList<String> fruits = new LinkedList<>();  
  
 // Add elements to the LinkedList  
 fruits.add("Apple");  
 fruits.add("Banana");  
 fruits.add("Mango");  
 fruits.add("Grapes");  
 System.*out*.println("LinkedList elements: " + fruits);  
 for(String s : fruits)  
 {  
 System.*out*.println(s);  
 }  
 System.*out*.println("Remove 1st element of Linkedlist : " + fruits.removeFirst());  
 System.*out*.println("Remove last element of Linkedlist : " + fruits.removeLast());  
  
 }  
}

**Output**

LinkedList elements: [Apple, Banana, Mango, Grapes]

Apple

Banana

Mango

Grapes

Remove 1st element of Linkedlist : Apple

Remove last element of Linkedlist : Grapes

**TASK9 – Update the first elements to a new value ?**

package Day13;  
  
import java.util.LinkedList;  
class TASK9\_UpdateNewElement  
{  
 public static void main(String[] args) {  
 LinkedList<String> fruits = new LinkedList<>();  
 fruits.add("Apple");  
 fruits.add("Banana");  
 fruits.add("Mango");  
 fruits.add("Grapes");  
 System.*out*.println("LinkedList elements: " + fruits);  
 fruits.set(1, "Fig");  
 System.*out*.println("LinkedList elements After Updating : " + fruits);  
  
 }  
  
}

**Output**

LinkedList elements: [Apple, Banana, Mango, Grapes]

LinkedList elements After Updating : [Apple, Fig, Mango, Grapes]

**TASK10**

package Day13;  
  
import java.util.LinkedList;  
  
public class TASK10 {  
 public static void main(String[] args) {  
 LinkedList<String> items = new LinkedList<>();  
  
 // Add elements to the LinkedList  
 items.add("Pen");  
 items.add("Notebook");  
 items.add("Pencil");  
 items.add("Eraser");  
  
 System.*out*.println("Display using for loop with get():");  
 for (int i = 0; i < items.size(); i++) {  
 System.*out*.println("Item at index " + i + ": " + items.get(i));  
 }  
  
 System.*out*.println("\nDisplay using for-each loop:");  
 for (String item : items) {  
 System.*out*.println("Item: " + item);  
 }  
 }  
}

**Output**

Display using for loop with get():

Item at index 0: Pen

Item at index 1: Notebook

Item at index 2: Pencil

Item at index 3: Eraser

Display using for-each loop:

Item: Pen

Item: Notebook

Item: Pencil

Item: Eraser

**TASK11 – Display all the elements in the list without loops**

package Day13;  
import java.util.LinkedList;  
  
public class TASK11{  
 public static void main(String[] args) {  
 // Create a LinkedList of Strings  
 LinkedList<String> fruits = new LinkedList<>();  
  
 // Add elements to the LinkedList  
 fruits.add("Apple");  
 fruits.add("Banana");  
 fruits.add("Mango");  
 fruits.add("Grapes");  
 System.*out*.println("LinkedList elements: " + fruits);  
 }  
}

**Output**

LinkedList elements: [Apple, Banana, Mango, Grapes]

**TASK12 – Convert LinkedList to Array**

package Day13;  
  
import java.util.LinkedList;  
  
public class TASK12\_LinkedListToArray {  
 public static void main(String[] args) {  
 // Create and populate the LinkedList  
 LinkedList<String> names = new LinkedList<>();  
 names.add("Pen");  
 names.add("Book");  
 names.add("Paper");  
 System.*out*.println(names);  
  
  
 // Convert LinkedList to Object array  
 Object[] arr = names.toArray();  
  
 // Display elements of the array  
 System.*out*.println("Elements in Object[] array:");  
 for (Object obj : arr) {  
 System.*out*.println(obj);  
 }  
 }  
}

**Output**

[Pen, Book, Paper]

Elements in Object [] array:

Pen

Book

Paper

**TASK13 – Cloning**

package Day13;  
  
import java.util.LinkedList;  
  
public class TASK13\_Cloning {  
 public static void main(String[] args) {  
 LinkedList<String> original = new LinkedList<>();  
 original.add("Arun");  
 original.add("Teddy");  
  
 LinkedList<String> clone = (LinkedList<String>) original.clone();  
 System.*out*.println("Original LinkedList: " + original);  
 System.*out*.println("Cloned LinkedList: " + clone);  
 }  
}

**Output**

Original LinkedList: [Arun, Teddy]

Cloned LinkedList: [Arun, Teddy]

🡪Cloning creates the shallow copy of Linked List.

**TASK14 – Using Push and POP**

package Day13;  
  
import java.util.LinkedList;  
  
public class TASK14\_PushAndPop {  
 public static void main(String[] args) {  
 // Create a LinkedList to use as a stack  
 LinkedList<String> stack = new LinkedList<>();  
  
 // Push elements onto the stack (adds to the front)  
 stack.push("First");  
 stack.push("Second");  
 stack.push("Third");  
  
 System.*out*.println("Stack after pushes: " + stack);  
  
 // Pop element from the stack (removes from the front)  
 String popped = stack.pop();  
 System.*out*.println("Popped element: " + popped);  
  
 System.*out*.println("Stack after pop: " + stack);  
 }  
}

**Output**

Stack after pushes: [Third, Second, First]

Popped element: Third

Stack after pop: [Second, First]

**TASK15 – Spliterator example using forEachRemaining**

package Day13;  
  
import java.util.LinkedList;  
import java.util.Spliterator;  
  
public class TASK15\_SpliteratorExampled {  
 public static void main(String[] args) {  
 LinkedList<String> names = new LinkedList<>();  
 names.add("Arun");  
 names.add("Kumar");  
 names.add("Arun Kumar");  
  
 Spliterator<String> sp = names.spliterator();  
 System.*out*.println("Using Spliterator:");  
 sp.forEachRemaining(System.*out*::println);  
 }  
}

**Output**

Using Spliterator:

Arun

Kumar

Arun Kumar

* Spliterator can be used to traverse collections.
* forEachRemaining() applies the action to remaining elements.

**TASK16 – Split list into two spliterators using trySplit()**

package Day13;  
  
import java.util.LinkedList;  
import java.util.Spliterator;  
  
public class TASK16\_Splits {  
 public static void main(String[] args) {  
 LinkedList<String> data = new LinkedList<>();  
 data.add("Nirupa");  
 data.add("Anu");  
 data.add("Shruthika");  
 data.add("Advika");  
  
 Spliterator<String> s1 = data.spliterator();  
 Spliterator<String> s2 = s1.trySplit();  
  
 System.*out*.println("Spliterator 1:");  
 while (s1.tryAdvance(System.*out*::println));  
  
 System.*out*.println("Spliterator 2:");  
 while (s2.tryAdvance(System.*out*::println));  
 }  
}

**Output**

Spliterator 1:

Nirupa

Anu

Spliterator 2:

Shruthika

Advika

**Notes:**

trySplit() divides the Spliterator into two parts for potential parallel processing.

tryAdvance() processes elements one at a time.

**TASK17 – Create a Doubly Linked List**

package Day13;  
  
public class TASK17\_DoublyLinkedList {  
 static class Node {  
 int data;  
 Node prev, next;  
  
 Node(int data) {  
 this.data = data;  
 }  
 }  
  
 Node head = null, tail = null;  
  
 void add(int data) {  
 Node newNode = new Node(data);  
 if (head == null) {  
 head = tail = newNode;  
 } else {  
 tail.next = newNode;  
 newNode.prev = tail;  
 tail = newNode;  
 }  
 }  
  
 void displayForward() {  
 Node current = head;  
 while (current != null) {  
 System.*out*.print(current.data + " ");  
 current = current.next;  
 }  
 }  
  
 void displayBackward() {  
 Node current = tail;  
 while (current != null) {  
 System.*out*.print(current.data + " ");  
 current = current.prev;  
 }  
 }  
  
 public static void main(String[] args) {  
 TASK17\_DoublyLinkedList list = new TASK17\_DoublyLinkedList();  
 list.add(10);  
 list.add(20);  
 list.add(30);  
 System.*out*.print("Forward: ");  
 list.displayForward();  
 System.*out*.print("\nBackward: ");  
 list.displayBackward();  
 }  
}

**Output**

Forward: 10 20 30

Backward: 30 20 1

🡪A doubly linked list stores links to both previous and next nodes. This allows traversal in both directions. It’s helpful when we want efficient backward movement through data.

**TASK18 – Copy data from one map to another map**

package Day13;  
import java.util.HashMap;  
  
public class TASK18\_CopyMap {  
 public static void main(String[] args) {  
 HashMap<String, Integer> original = new HashMap<>();  
 original.put("Arun", 10);  
 original.put("Teddy", 20);  
  
 HashMap<String, Integer> copy = new HashMap<>(original);  
 System.*out*.println("Copied Map: " + copy);  
 }  
}

**Output**

Copied Map: {Arun=10, Teddy=20}

* We can copy the contents of one map into another using the constructor. It helps quickly duplicate data and create backups or new modified versions.

**TASK19 – Created HashMap with Capacity and Load Factor**

package Day13;  
  
import java.util.HashMap;  
public class TASK19\_HashMapWithLoadFactor {  
 public static void main(String[] args) {  
 HashMap<String, Integer> hm = new HashMap<>(10, 0.75f);  
 hm.put("A", 1);  
 hm.put("B", 2);  
 System.*out*.println(hm);  
 }  
}

**Output**

{A=1, B=2}

**TASK20 – Rehashing using Capacity and Load Factor**

package Day13;  
import java.util.HashMap;  
public class TASK20\_Rehashing {  
 public static void main(String[] args) {  
 HashMap<Integer, String> map = new HashMap<>(4, 0.75f);  
  
 map.put(1, "A");  
 map.put(2, "B");  
 map.put(3, "C");  
 map.put(4, "D");  
 map.put(5, "E"); // triggers rehashing internally  
  
 System.*out*.println(map);  
 }  
}

**Output**

{1=A, 2=B, 3=C, 4=D, 5=E}

* When the number of entries exceeds capacity × load factor, the map resizes automatically. Here, rehashing happens after 3 entries (4 × 0.75 = 3), so on the 4th insert, internal rehashing happens.

**TASK21 – Different ways of creating HashMap**

package Day13;  
  
import java.util.HashMap;  
public class TASK21\_DifferentHashMap {  
 public static void main(String[] args) {  
 HashMap<String, Integer> hm1 = new HashMap<>(); // default  
 HashMap<String, Integer> hm2 = new HashMap<>(10); // with capacity  
 HashMap<String, Integer> hm3 = new HashMap<>(hm2); // copy  
 HashMap<String, Integer> hm4 = new HashMap<>(10, 0.75f); // capacity + load factor  
  
 System.*out*.println("HashMaps created using different constructors");  
 }  
}

**Output**

HashMaps created using different constructors.

**TASK22 – Created a Circular Linked List and Display elements**

package Day13;  
  
public class TASK22\_CircularLinkedList {  
 static class Node {  
 int data;  
 Node next;  
  
 Node(int data) {  
 this.data = data;  
 }  
 }  
  
 Node head = null, tail = null;  
  
 void add(int data) {  
 Node newNode = new Node(data);  
 if (head == null) {  
 head = tail = newNode;  
 tail.next = head;  
 } else {  
 tail.next = newNode;  
 tail = newNode;  
 tail.next = head;  
 }  
 }  
  
 void display() {  
 if (head == null) return;  
 Node current = head;  
 do {  
 System.*out*.print(current.data + " ");  
 current = current.next;  
 } while (current != head);  
 }  
  
 public static void main(String[] args) {  
 TASK22\_CircularLinkedList list = new TASK22\_CircularLinkedList();  
 list.add(10);  
 list.add(20);  
 list.add(30);  
 list.display();  
 }  
}

**Output**

10 20 30

**HOME TASKS**

* **Advantages of LinkedList**

| **Advantage** | **Explanation** |
| --- | --- |
| **Dynamic Size** | Can grow or shrink in size without memory reallocation. |
| **Efficient Insert/Delete** | Insertions and deletions are fast (O(1)) if done at the beginning or with a pointer to the node. |
| **No Wasted Memory** | Uses memory only as needed for elements (no pre-allocation like arrays). |
| **Useful for Queue/Stack Implementations** | Ideal for FIFO/LIFO structures where element order matters. |

* **Disadvantages of LinkedList**

| **Disadvantage** | **Explanation** |
| --- | --- |
| **Slow Access Time** | Random access is not possible (O(n) time to reach a node). |
| **More Memory Overhead** | Extra memory required for storing pointers (next, and possibly prev). |
| **Cache-unfriendly** | Due to non-contiguous memory allocation, it doesn't benefit from CPU cache. |
| **Complexity in Operations** | Reversal, sorting, or manipulation can be trickier than with arrays. |

* **Applications of Linked List**
* Used for dynamic data structures like stacks, queues, and deques.
* Adjacency lists in graphs are often implemented using linked lists.
* OS uses linked lists for memory management (free memory blocks).
* In editors and software, to keep a sequence of previous states.
* Useful where dynamic memory allocation and frequent insert/delete is required.