DATA STRUCTURES AND ALGORITHMS

- LECTURE 6-

(2ND YEAR OF STUDY)

Skopje, 2023/24

Contents

6.1. Introduction

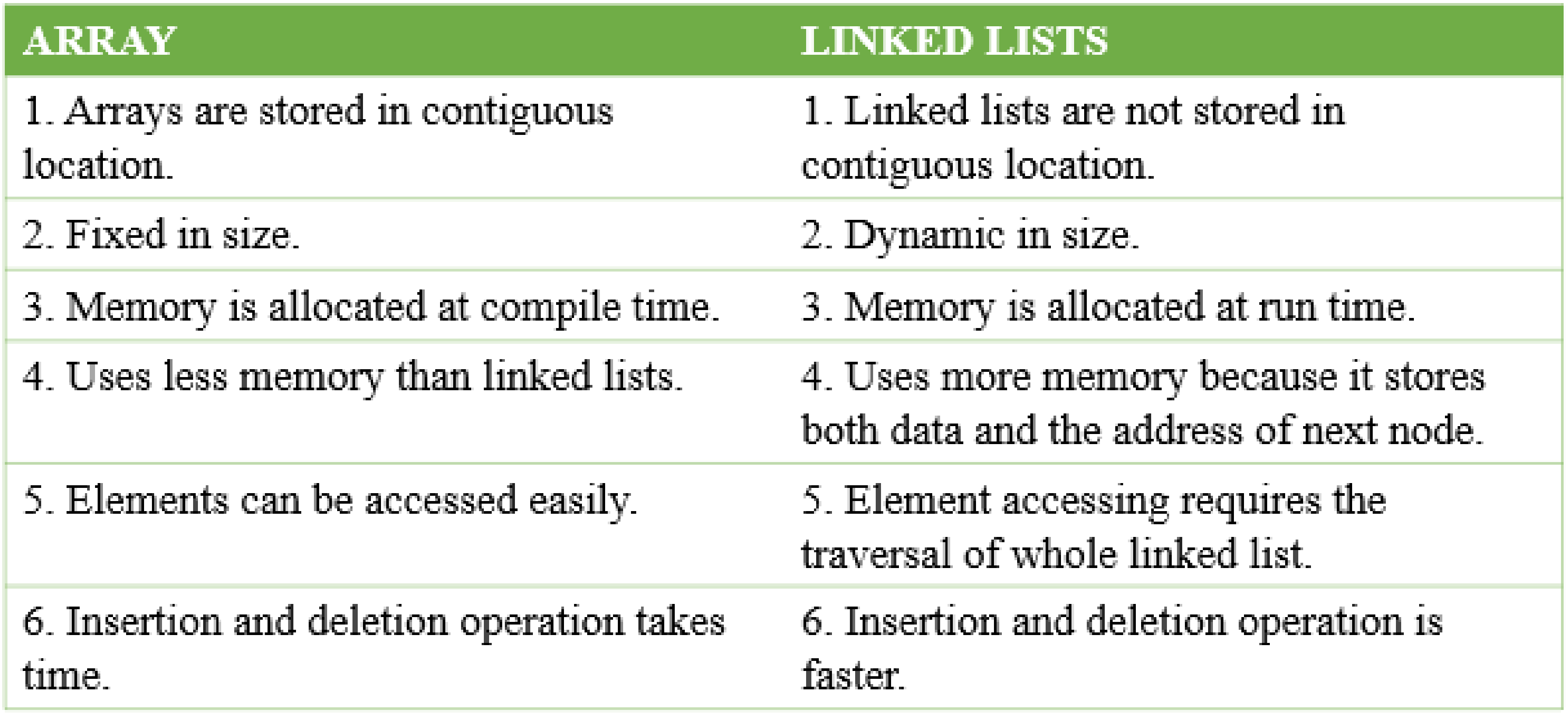
- A **Linked List** is a *linear data structure*, in which the elements are not stored at contiguous memory locations.
- The elements in a Linked List are linked using ***pointers***.
- In simple words, a Linked List forms a series of connected ***nodes***, where each node contains a **data field** and a **reference** (i.e. **link**) to the next node in the list.

6.1. Introduction

- **Node Structure:** A node in a linked list typically consists of two components:
  - **Data:** It holds the actual value or data associated with the node.
  - **Next Pointer:** It stores the memory address (reference) of the next node in the sequence.
- **Head and Tail:** The linked list is accessed through the ***head node***, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the ***tail node***.

6.1. Introduction

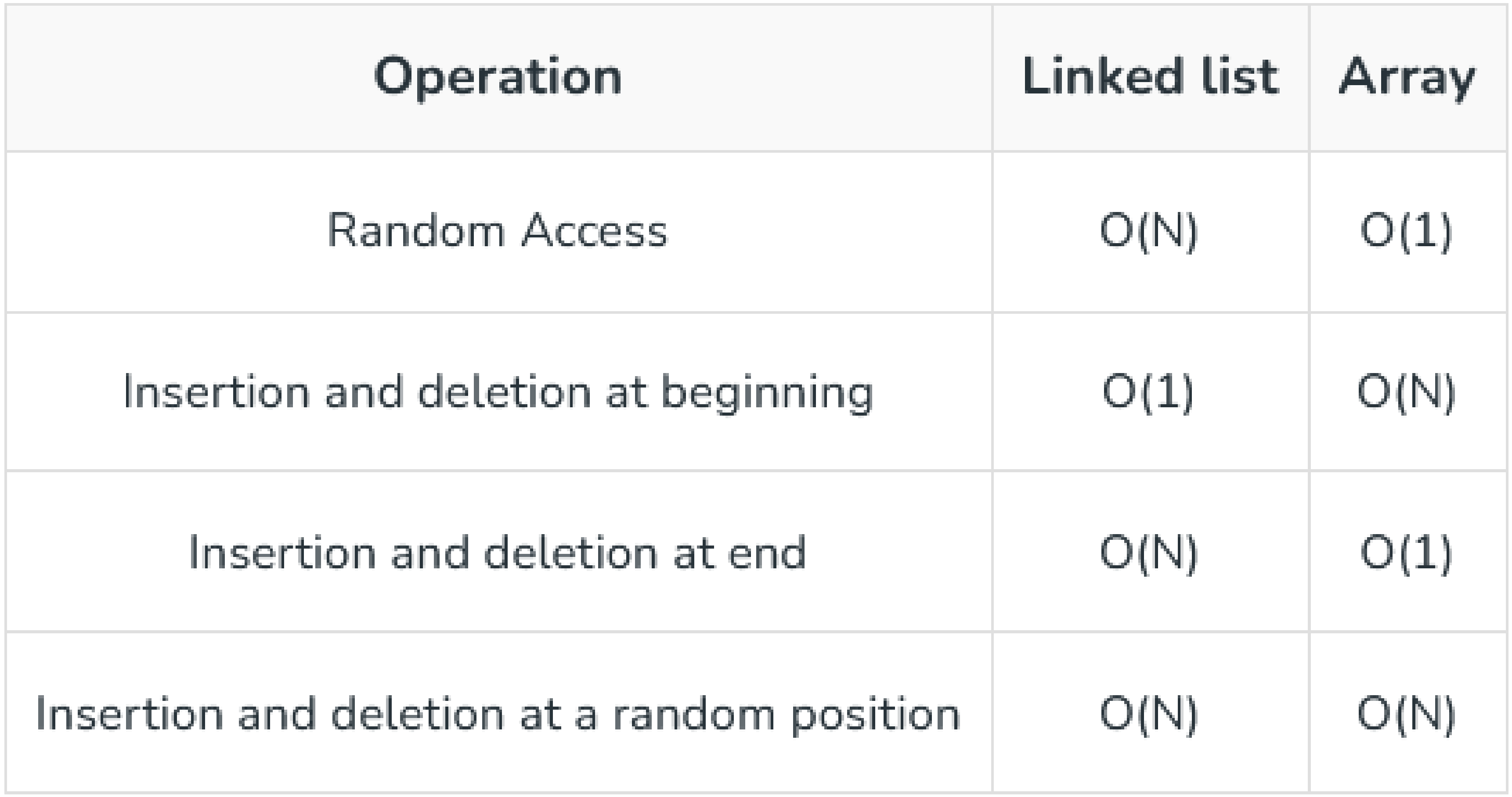
- **Linked List vs. Array**



ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

6.1. Introduction

- **Linked List vs. Array in Time Complexity**



Operation	Linked list	Array
Random Access	O(N)	O(1)
Insertion and deletion at beginning	O(1)	O(N)
Insertion and deletion at end	O(N)	O(1)
Insertion and deletion at a random position	O(N)	O(N)

6.1. Introduction

- **Advantages of Linked Lists:**
  - **Dynamic nature:** Linked lists are used for dynamic memory allocation.
  - **Memory efficient:** Memory consumption of a linked list is efficient as its size can grow or shrink dynamically according to our requirements, which means effective memory utilization, hence no memory wastage.
  - **Ease of Insertion and Deletion:** Insertion and deletion of nodes are easily implemented in a linked list at any position.
  - **Implementation:** For the implementation of stacks and queues, and for the representation of trees and graphs.

6.1. Introduction

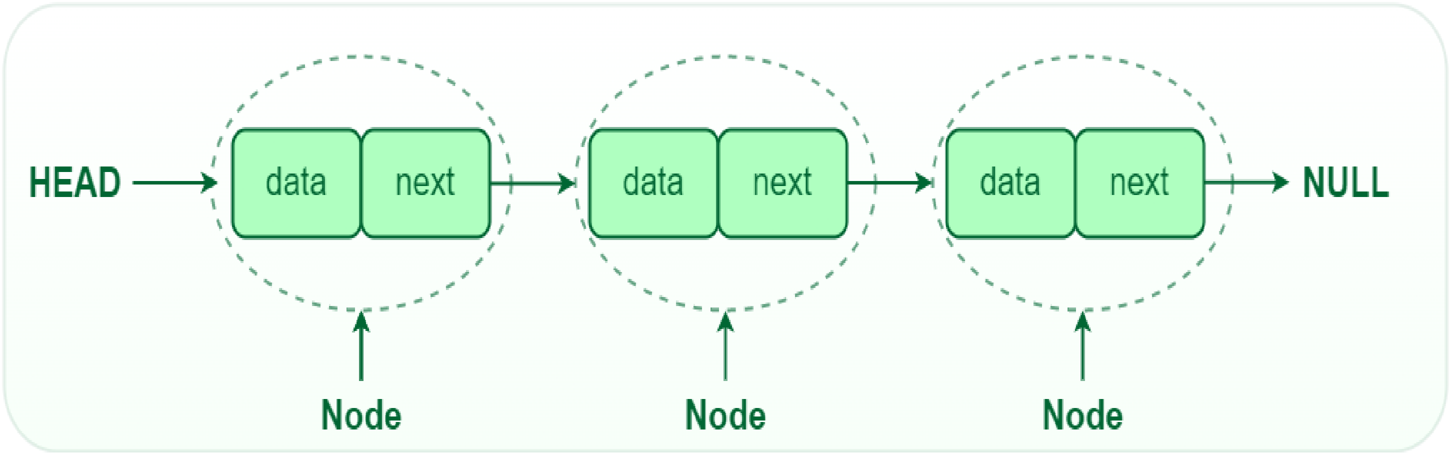
- **Disadvantages of Linked Lists:**
  - **Memory usage:** The use of pointers is more in linked lists, hence complex and requires more memory.
  - **Accessing a node:** Random access is not possible due to dynamic memory allocation.
  - **Search operation costly:** Searching for an element is costly and requires **O(*N*)** time complexity, where ***N*** is the number of nodes in the linked list.
  - **Traversing in reverse order:** Traversing is more time-consuming, and reverse traversing *is not possible* in singly linked lists.

6.2. Singly Linked List

- **Singly Linked List** is the simplest type of linked list in which every node contains some ***data*** and a ***pointer*** to the next node of the same data type.
- The node contains a pointer to the next node means that the node stores the address of the next node in the sequence.
- A *single linked list* allows the traversal of data only in one way.

6.2. Singly Linked List

- Below is the image for the same:



- The Node structure of a Singly Linked List

```
// Node of a singly linked list
class Node {
public:
    int data;

    // Pointer to next node in LL
    Node* next;
};
```

6.2. Singly Linked List

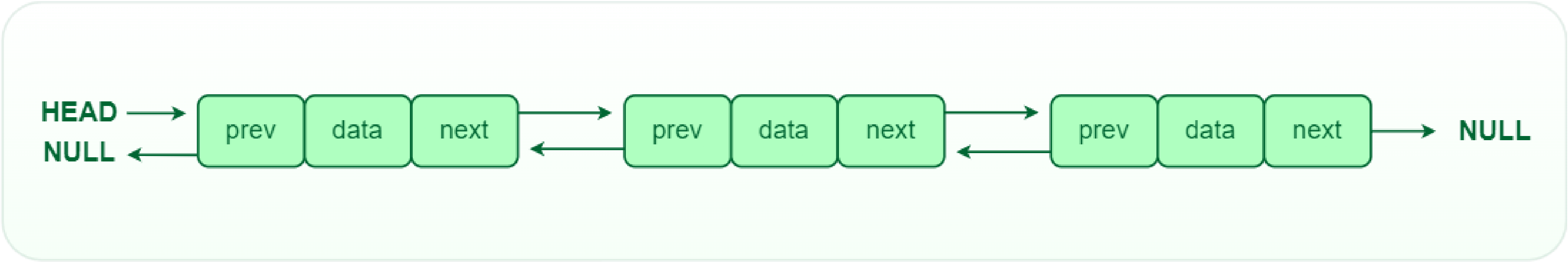
- **Characteristics of a Singly Linked List:**
  - Each node holds a single value and a reference to the next node in the list.
  - The list has a **head**, which is a *reference* to the *first* node in the list, and a **tail**, which is a *reference* to the *last* node in the list.
  - The nodes are not stored in a contiguous block of memory, but instead, each node holds the address of the next node in the list.
  - ***Accessing elements*** in a singly linked list requires ***traversing the list*** from the head to the desired node, as there is no direct access to a specific node in memory.

6.3. Doubly Linked List

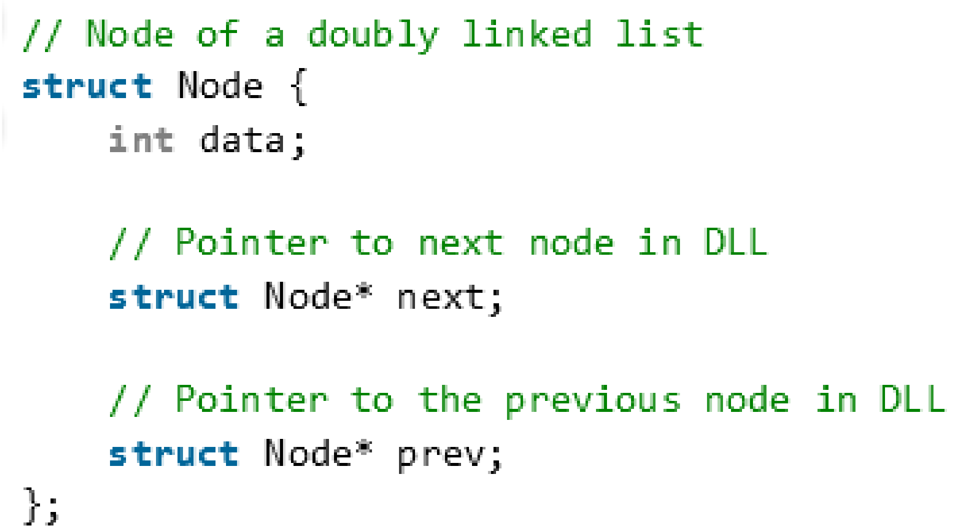
- In a **Doubly Linked List** (or, a *two-way linked list*) each node contains references to both the ***next*** and ***previous*** nodes.
- This allows for traversal in both ***forward*** and ***backward*** directions, but it requires additional memory for the backward reference.
- It contains 3 parts of data. This would enable us to traverse the list in the backward direction as well.

6.3. Doubly Linked List

- Below is the image for the same:



- The Node structure of a Doubly Linked List



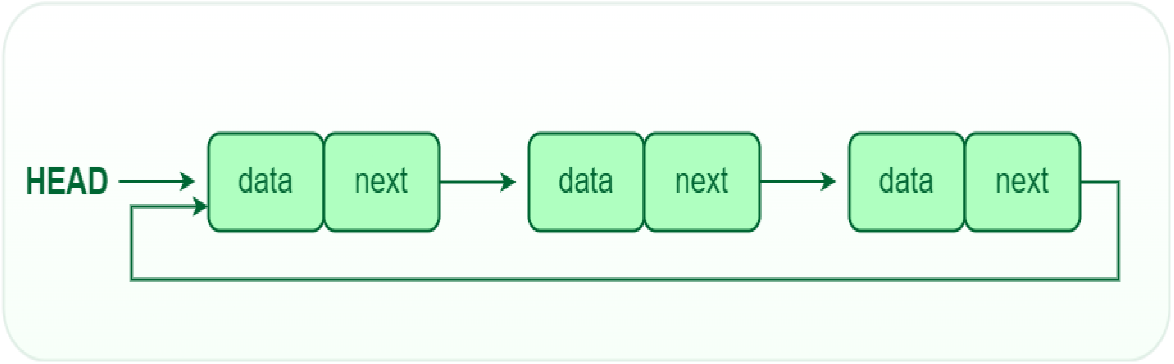
```
// Node of a doubly linked list
struct Node {
    int data;

    // Pointer to next node in DLL
    struct Node* next;

    // Pointer to the previous node in DLL
    struct Node* prev;
};
```

6.4. Circular Linked List

- A **Circular Linked List** is that in which the last node contains the pointer to the first node of the list.
- The *circular linked list* is a linked list where all nodes are connected to form a *circle*.
- In a *circular linked list*, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.

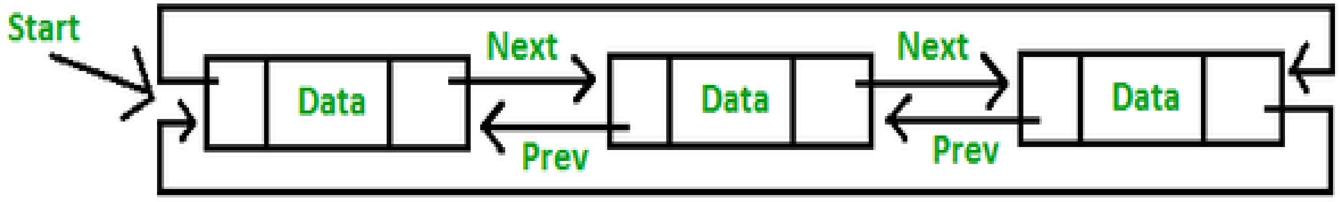
6.4. Circular Linked List

- There are generally **two types** of **Circular Linked Lists:**
  - **Circular Singly Linked List:** In a *Circular Singly Linked List*, the *last node* of the list contains a pointer to the *first node* of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. ***No null value*** is present in the next part of any of the nodes.



6.4. Circular Linked List

- **Circular Doubly Linked List:** *Circular Doubly Linked List* has properties of both *doubly linked list* and *circular linked list* in which two consecutive elements are linked or connected by the ***previous*** and ***next*** pointer, and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.



6.5. Application of Linked Lists

- **Applications of Linked Lists:**
  - Linear data structures such as: stack, queue, and non-linear data structures such as: hash maps, and graphs *can be implemented* using ***linked lists***.
  - *Dynamic memory allocation*: We use a ***linked list*** of free blocks.
  - *Implementation of graphs*: Adjacency list representation of graphs is the most popular in that it uses ***linked lists*** to store adjacent vertices.
  - In web browsers and editors, ***doubly linked lists*** can be used to *build a forward and backward navigation button.*
  - A ***circular doubly linked list*** can also be used for implementing data structures like Fibonacci heaps.

6.5. Application of Linked Lists

- **Applications of Linked Lists in real world:**
  - The *music player song list* is linked to the previous and next songs.
  - In a *web browser*, previous and next web page URLs are linked through the previous and next buttons.
  - In the *image viewer*, the previous and next images are linked with the help of the previous and next buttons.
  - Switching between two applications is carried out by using "**alt+tab**" in windows and "**cmd+tab**" in mac book. It requires the functionality of a ***circular linked list***.
  - In *mobile phones*, we save the contacts of people. The newly entered contact details will be placed at the correct alphabetical order.
  - The modifications that we made in the documents are actually created as nodes in ***doubly linked list***. We can simply use the undo option by pressing **Ctrl+Z** to modify the contents.