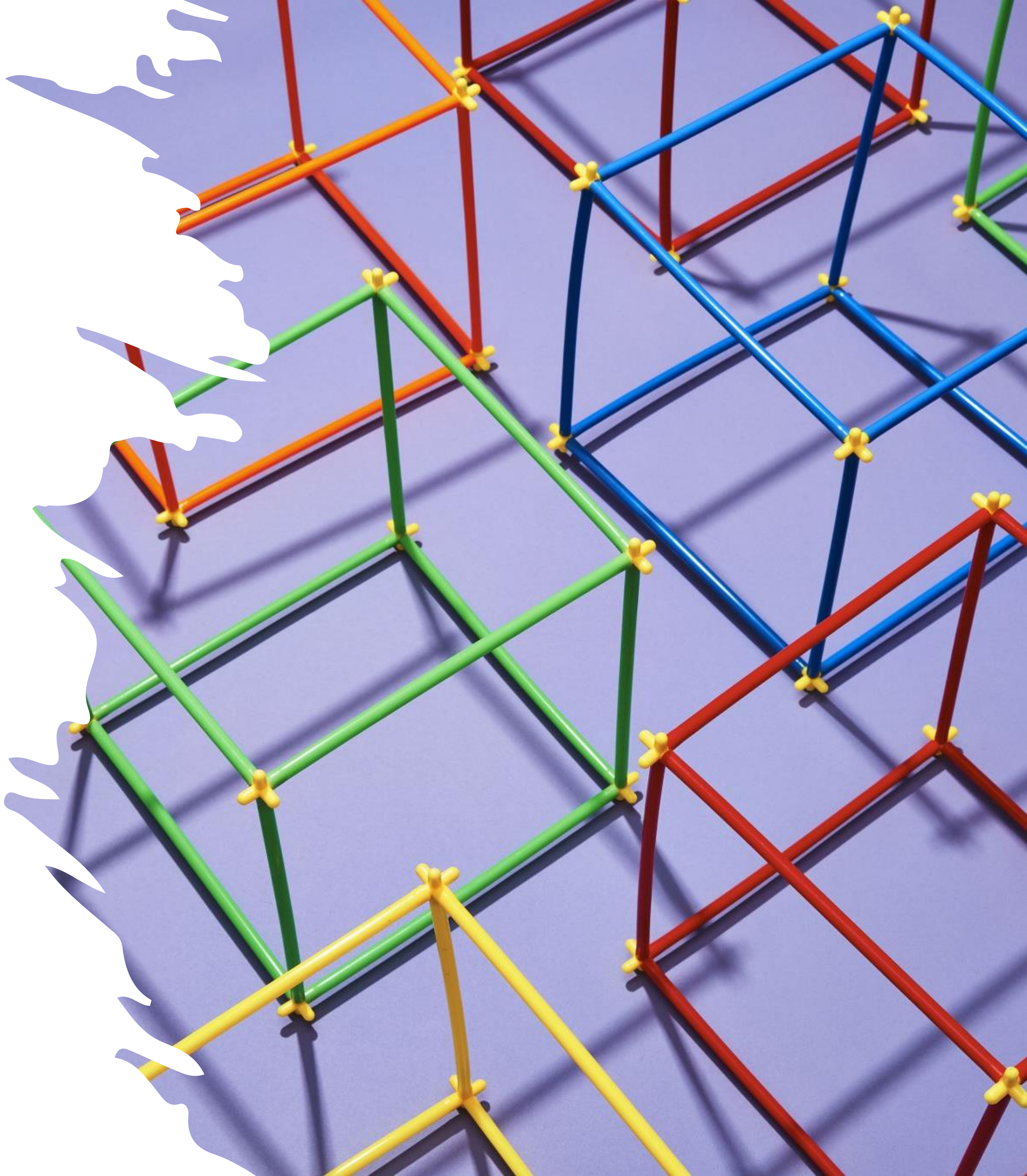
Data Structure
Lecture 4: Linked List

Prepared by

Dr. Mohammed Salah Al-Obiadi

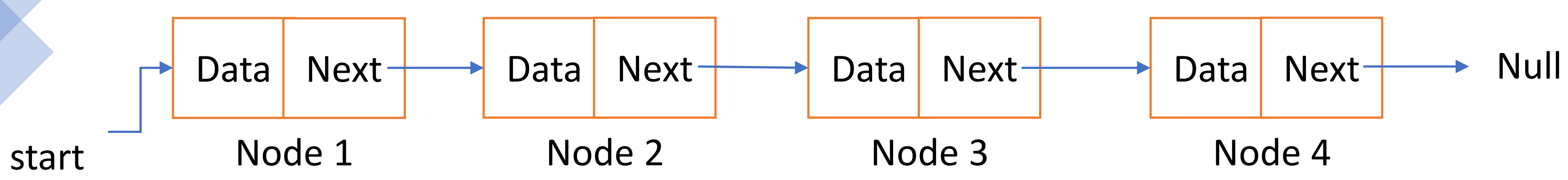


What is a Linked List?

A linked list is a data structure used for storing collections of data.

A linked list has the following properties:

1. Successive elements are connected by pointers.
2. The last element points to NULL.
3. Can grow or shrink in size during execution of a program.
4. Can be made just as long as required (until systems memory exhausts).
5. Does not waste memory space. It allocates memory as list grows.



Linked List vs Arrays?

Array	Linked list
Array elements store in a contiguous memory location.	Linked list elements can be stored anywhere in the memory
Array works with a static memory and cannot be changed at the run time.	The Linked list works with dynamic memory means memory size can be changed at the run time.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list takes less time while performing any operation like insertion, deletion, etc.
Accessing any element in an array is faster as the element in an array can be directly accessed through the index.	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time.	In the case of a linked list, memory is allocated at run time.
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements then the rest of the space will be unused.	Memory utilization is efficient as the memory can be allocated or deallocated at the run time.
Arrays take O(1) for access to an element.	Linked lists take O(n) for access to an element.

Operation on Linked List

1- Traversal: To traverse all the nodes one after another.

2- Insertion: To add a node at the given position.

3- Deletion: To delete a node.

4- Searching: To search an element(s) by value.

5- Updating: To update a node.

6- Sorting: To arrange nodes in a linked list in a specific order.

7- Merging: To merge two linked lists into one.

Types of Link List

1- Single Link List

2- Double Link List

3- Circular Link List

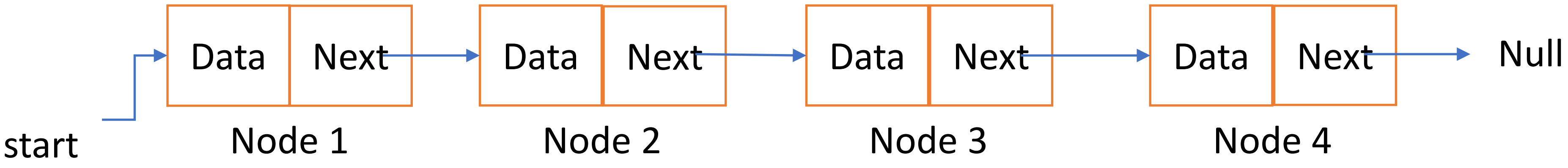
4- Doubly Circular linked list

Single Link List

Generally “linked list” means a single linked list.

This list consists of a number of nodes in which each node has a *next* pointer to the following element.

The link of the last node in the list is NULL, which indicates the end of the list.



STRUCTURE OF THE NODE OF A LINKED LIST

Struct **tagname**

```
{
    Data type member1;
    Data type member2;
    …………………….
    ……………………
    …………………..
    Data type membern;
    Struct tagname *var;
};
```

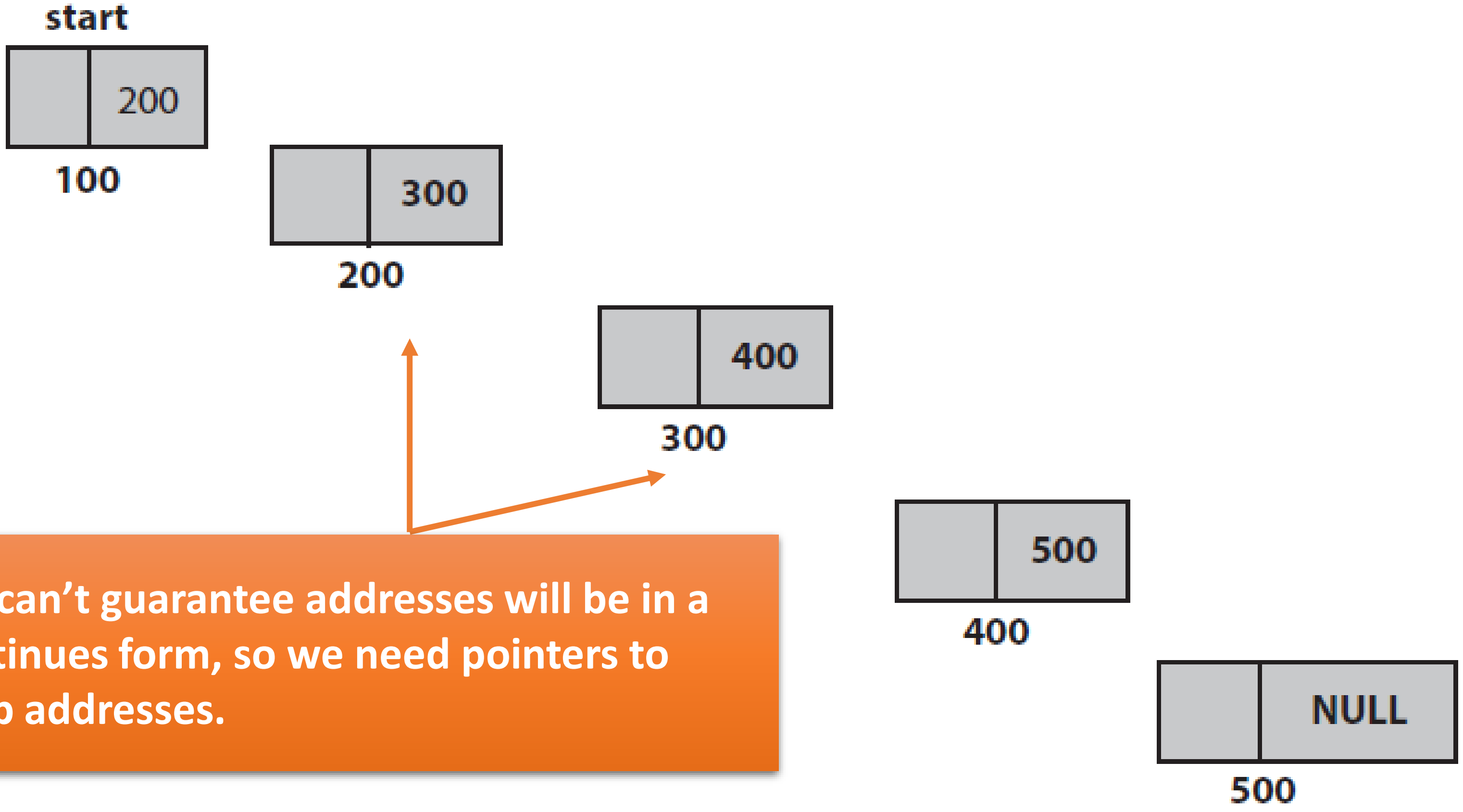
Example:

struct **link**

```
{
int info;
struct link *next;
};
```

LOGIC FOR CREATION

struct link start, *node;



We can’t guarantee addresses will be in a continues form, so we need pointers to keep addresses.

Algorithm For Creation Of Single Link List

```
Struct link start, *node
create(start,node) [start is the structure type of variable][node is the structure type of pointer]
    step-1 : node = &start
    step-2 : node → next = new link() //allocate memory of size struct link for the node
             node = node → next
             input : node → info
             node → next = null
    step-3 : repeat step-2 to create more nodes
    step-4 : return
```

Algorithm For Traversing Of Single Link List

```
struct link start, *node;
traverse(start,node) [start is the structure type of variable]  [node is the structure type of pointer]
step-1 : node = start.next
step-2 : repeat while (node!=null )
                write : node → info
                node = node → next
                end of loop
step-3 : return
```

Insertion Into Linked List

The insertion process with link list can be discussed in four different ways:

1. Insertion at Beginning.
2. Insertion at End.
3. Insertion when node number is known.
4. Insertion when information is known.

Algorithm For Insertion At Beginning

struct start, *first, *node,* newnode

insbeg(start,first,node, newnode) [start is the structure variable] [node and first is the structure pointer]

```
step-1 :  first = &start //first saves start's address
          node = start.next
step-2 :  newnode = new link()
          input : newnode → info
          first → next = newnode
          newnode → next := node
step-3 : return
```

Algorithm For Insertion At Last

```
struct start, *last, *node,* newnode
inslast(start,last,node,newnode)
step-1 : last = &start //last's pointer saves start's address
         node = start.next
step-2 : repeat while(node != null)
             node = node → next
             last = last → next
step-3 : newnode →next=new link() //allocate a memory to newnode
         input : newnode → info
         last → next = newnode
         newnode → next = null
step-4 : return
```