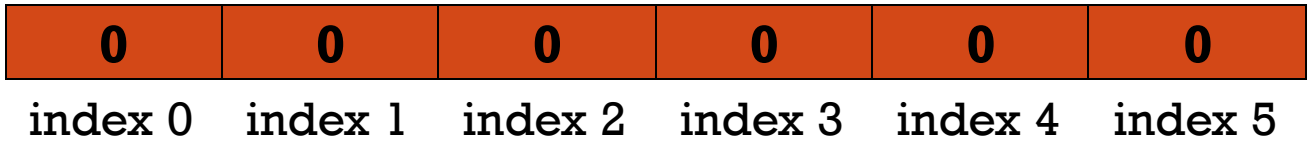
ARRAYS & VECTORS

ARRAYS

- Primitive variables are designed to hold only one value at a time.
- Arrays allow us to create a collection of like values that are indexed.
- An array can store any type of data but only one type of data at a time.

```
type name [elements];
```

```
int foo [6];
  //creates a new array that will hold 6 integers.
```



Array element values are initialized to 0.
Array indexes always start at 0.

CREATING ARRAYS

Initializing Arrays:

```
int arr[5]; // Array of 5 integers with undefined values
int arr[5] = {1, 2}; // Initializes first two elements, others are set to 0
int arr[5] = {1, 2, 3, 4, 5}; // Initializes all elements
int arr[] = {1, 2, 3, 4, 5}; // Compiler deduces size as 5
std::array<int, 5> arr = {1, 2, 3, 4, 5}; // Array of 5 integers
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}}; // 2D array with 2 rows and 3 columns
int arr[5] = {}; // All elements are initialized to 0
int arr[5]{1, 2, 3, 4, 5}; // Aggregate initialization C++ 20 onwards
```

Using Pointers:

```
int* arr = new int[5]; // Dynamically allocates an array of 5 integers
delete[] arr; // Free the dynamically allocated memory
```

PROPERTIES OF ARRAYS

- The array size must be a non-negative number.
- It may be a literal value or be derived from a constant or variable.

```
const int ARRAY_SIZE = 3;
int[ARRAY_SIZE]={1,2,3};
```

- Once created, an array size is fixed and cannot be changed.
- Processing data in an array is the same as any other variable.

```
grossPay = hours[3] * payRate;
```

- Pre and post increment works the same:

```
int score[] = {7, 8, 9, 10, 11};
++score[2]; // Pre-increment operation
score[4]++; // Post-increment operation
```

UNSORTED ARRAY

- **Searching:**
  Searching can be performed by a Linear Search Algorithm which is O(n)
- **Inserting:**
  Insertion is faster than in an unsorted array, since the position does not matter.
  Worst case : O(1) – Always inserted at the end of the array.
- **Deleting:**
  To delete a particular element from the array, use Linear Search to find it.
  After deletion, the remaining elements need to shifted to the left.
  Worst Case: O(n) – All elements may have to be shifted.

SORTED ARRAYS

- **Searching:**
  Searching for a particular element can be done using Binary search which is O(logn)
- **Inserting:**
  Insertion needs to be done at a particular index in a sorted array.
  Elements after the position must be shifted to the right
  Worst case : O(n) – All elements may have to be shifted.
- **Deleting:**
  To delete a particular element from the array, use Binary search to find it.
  After deletion, the remaining elements need to shifted to the left.
  Worst Case: O(n) – All elements may have to be shifted.

DISADVANTAGES OF ARRAYS

When you declare a standard array, you have to tell C++ a fixed number of memory locations that you want the array to contain. During the run of a program, this cannot be changed unless you create a new standard array and copy all of the elements to it. This would be awkward to do and implement in every program that uses standard arrays.

When you declare a Vector, you have the option of changing the size of the memory locations. Performance wise other than the ease of changing sizes, Vector does not outperform Arrays.

```
std::vector<int> arr = {1, 2, 3, 4, 5}; // Dynamic array with 5 elements
```

ABSTRACT DATA TYPES

ABSTRACT DATA TYPES

- Concept that defines a data type logically
- Specifies set of data and set of operations that can be performed on that data.
- Does not mention anything about how operation will be implemented.
- Exists as an idea but does not have a physical implementation.

LIST ADT

- void add(Type newItem)
  - Add items to the end of the list
- void add(int position, Type newItem)
  - Add item at a certain position
- boolean contains(Type item)
  - Check if an Item exists in the list
- int size()
  - Return length of the list
- boolean isEmpty()
  - Delete the list

- Type get(int position)
  - Get node at a particular position
- Type delete(int position)
  - Delete a node at a particular position
- void printList()

ADTS AND DATA STRUCTURES

Data structure is the physical implementation of Abstract Data Type.

ADT	Data Structure
Logical view of data and the operations to manipulate the data ADT specifies WHAT TO DO Used by an Interface	Actual representation of data and algorithms to manipulate the data Data Structures specifies HOW TO DO Used by the Client Program

LINKED LISTS

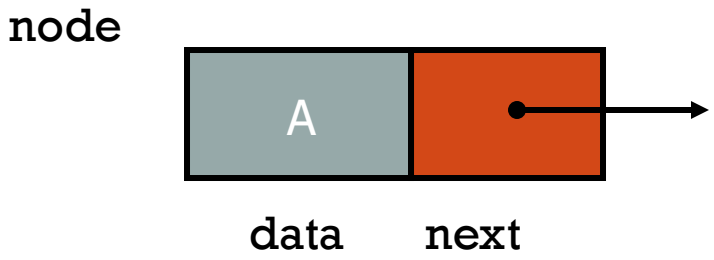
LINKED LISTS

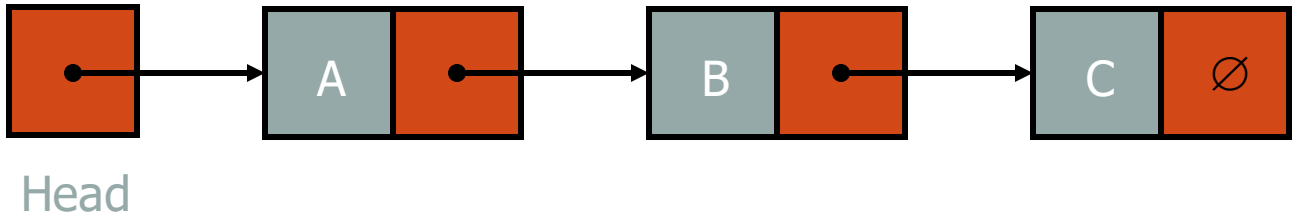
- Dynamic Data structures made up of nodes
- Data is not stored in contiguous memory Locations
- Insertion and deletion of elements is easier
- Can be used to implement abstract data types like Stacks and Queues
- Efficient random access is not possible
- Implementation requires extra memory

TYPES OF LINKED LISTS

- Single Linked Lists
- Double Linked Lists
- Circular Linked Lists

SINGLE LINKED LIST





NODES : OBJECTS TO STORE ELEMENTS

- A special "node" type of object that represents an element of a list
- Each node will keep a reference to the node after it (the "next" node)
- The last node will have `next -> nullptr,` signifying the end of the list

LINKED LIST IMPLEMENTATION

- **Linked List**: a list implemented using a linked sequence of nodes

  - The list needs to keep a reference to the first node (we might name it `head`)
  - We can also keep reference to the last node (we might name it `tail`)

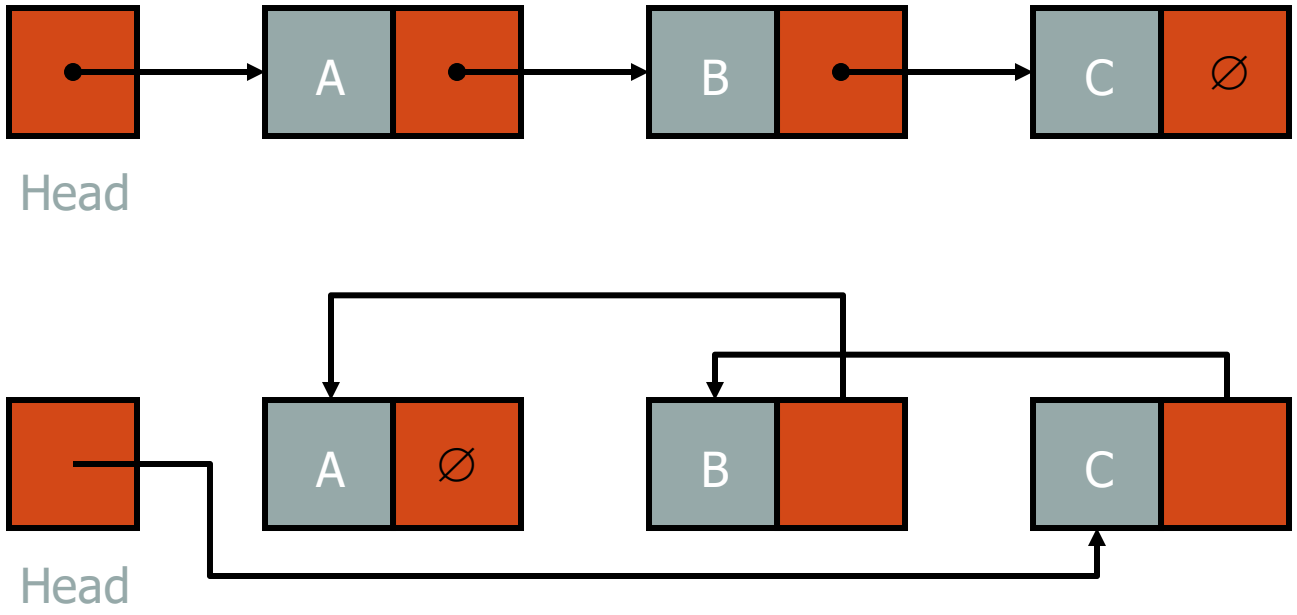
INSERTION IN LINKED LIST

- Insert at the beginning of the list
  - O(1)
- Insert at the end of the List
  - O(n)
- Insert at a given position
  - O(n)

DELETION IN LINKED LIST

- Delete at the beginning of the list
  - O(1)
- Delete at the end of the List
  - O(n)
- Delete at a given position
  - O(n)

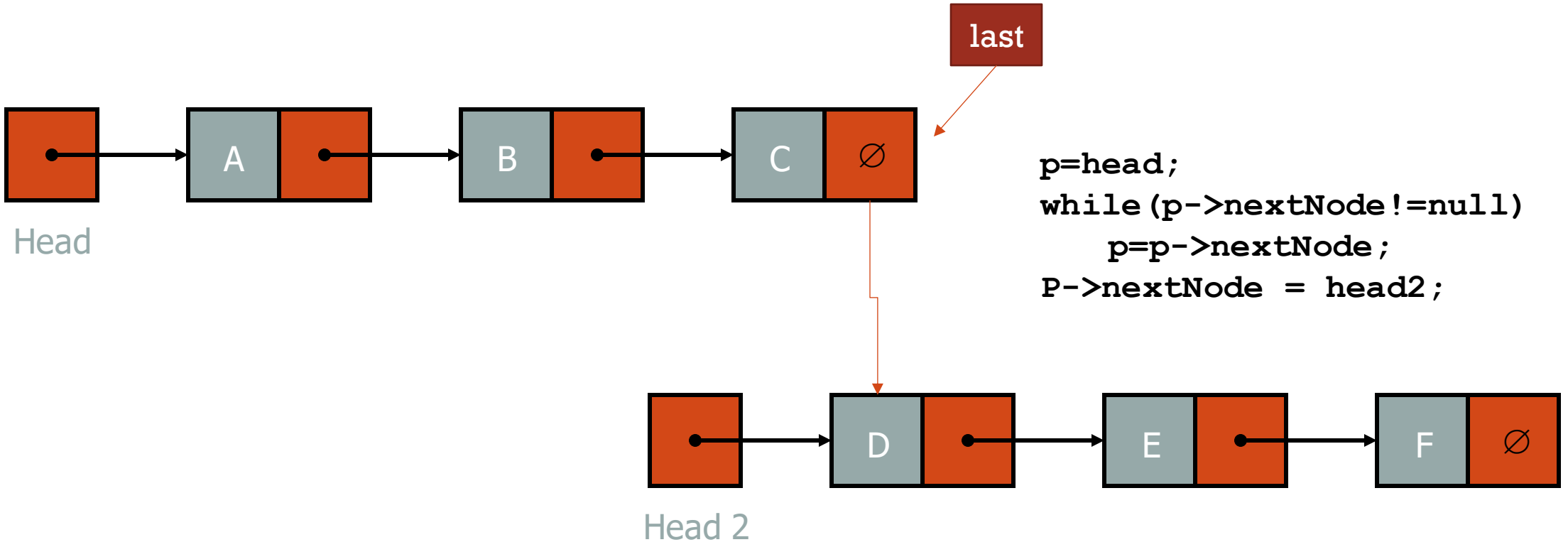
REVERSING A LINKED LIST




```
Node* prev, curr,  next;
prev = null;
curr = head;
while(curr!=null)
{
  next = curr->nextNode;
  Curr->nextNode = prev;
  prev = curr;
  curr = next;
}
head = prev;
```

CONCATENATION – SINGLY LINKED LISTS

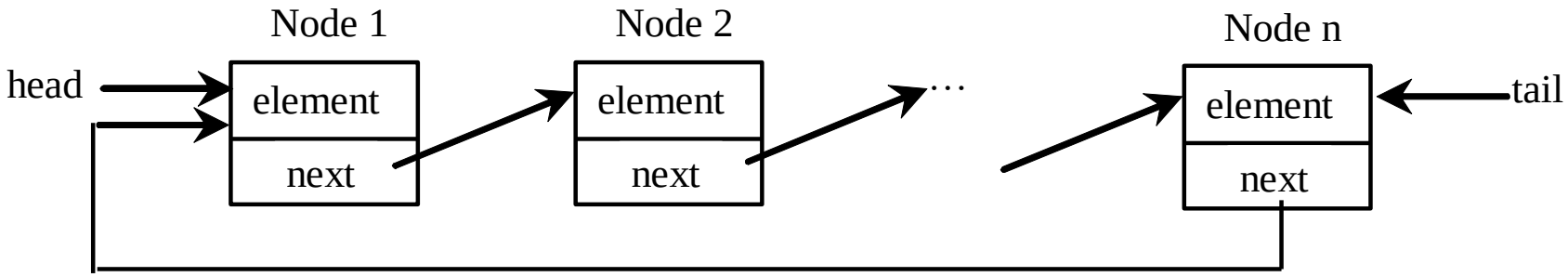
- Joining One linked List to the end of another linked list.



CIRCULAR LINKED LISTS

A circular, singly linked list is like a singly linked list, except that the pointer of the last node points back to the first node.

If `curr->nextNode = head` , you have reached the last node in the list.



CIRCULAR LINKED LISTS

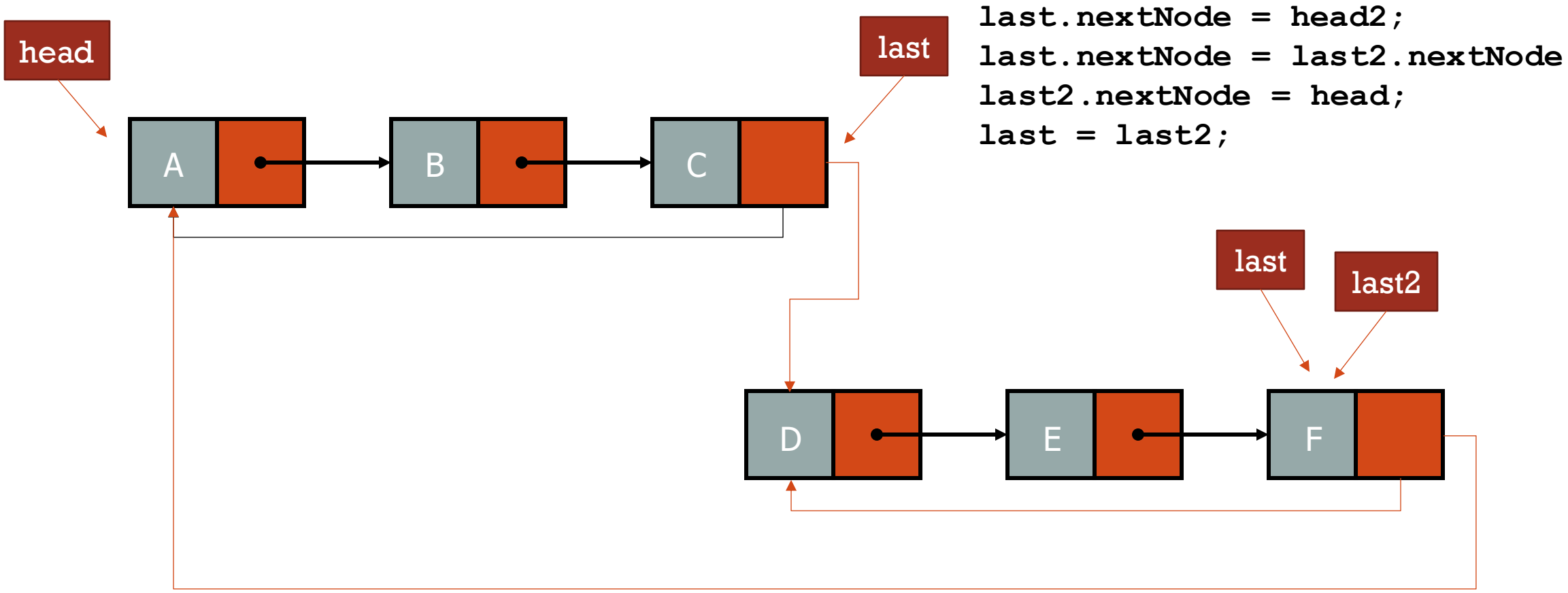
Advantages

- Entire Linked List can be traversed starting at any node.
- Fewer special cases for insertion and deletion, since all nodes have a previous node and a next node.

Disadvantages

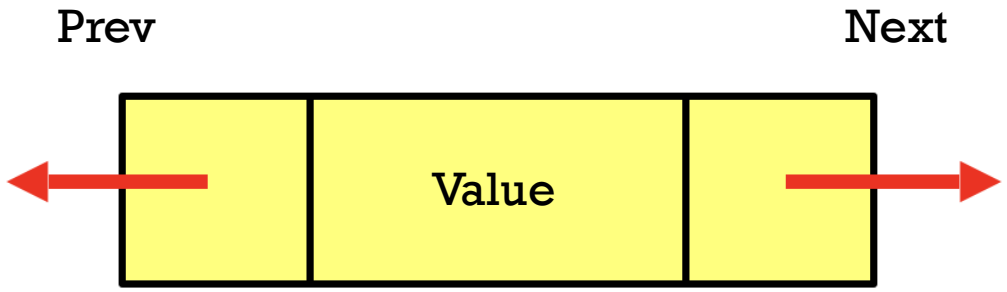
- If not executed properly, searches could lead to infinite loop.
- It is more complicated to execute when the Linked List is doubly linked.
- Accessing individual nodes takes longer due to additional traversals

CONCATENATION – CIRCULAR LINKED LISTS




```
last.nextNode = head2;
last.nextNode = last2.nextNode
last2.nextNode = head;
last = last2;
```

DOUBLY LINKED LIST NODE




```
Class Node {
  DataType value;
  Node* prev;
  Node* next;
};
```

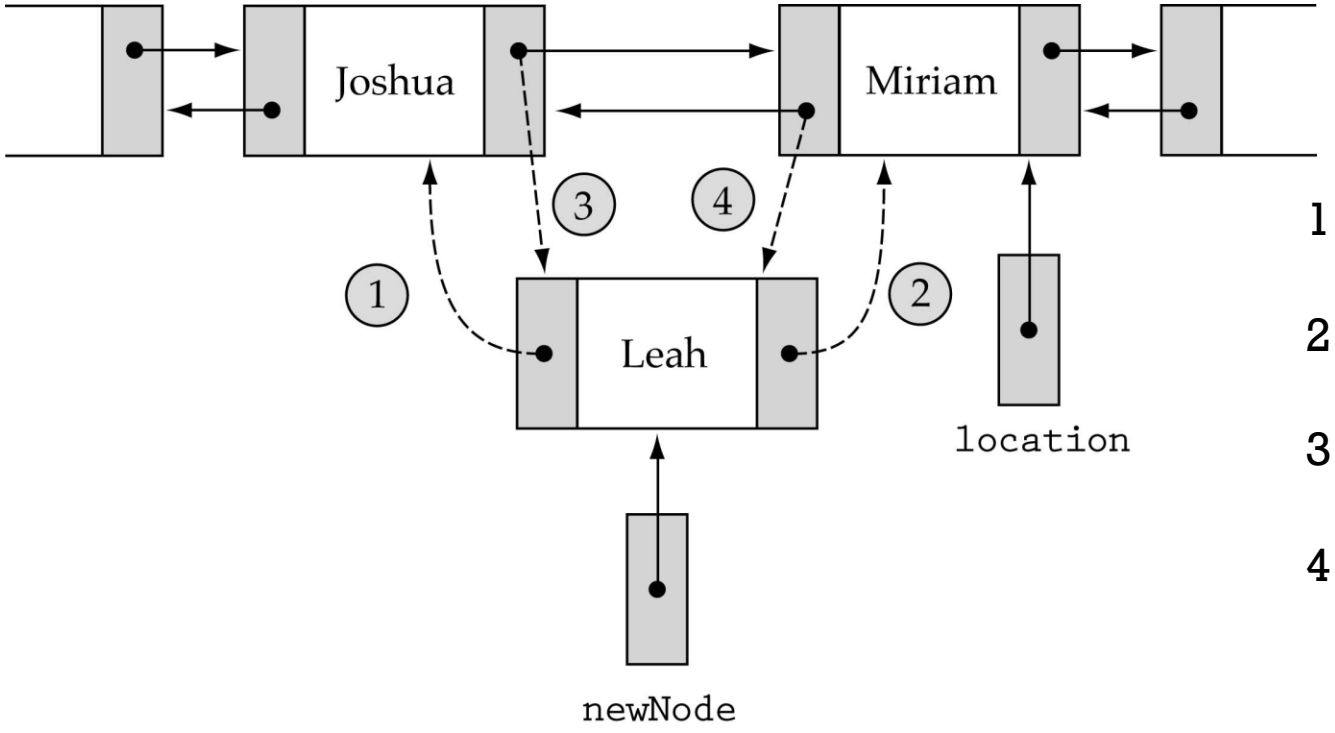
INSERTION IN DOUBLY LINKED LIST

- Insert at the beginning of the list
  - O(1)
- Insert at the end of the List
  - O(n)
- Insert at a given position
  - O(n)

DELETION IN DDOUBLY LINKED LIST

- Delete at the beginning of the list
  - O(1)
- Delete at the end of the List
  - O(n)
- Delete at a given position
  - O(n)

INSERTION IN DOUBLY LINKED LIST



1. newNode->back = location->back;
2. newNode->next = location
3. location->back->next=newNode;
4. location->back = newNode;

DELETION IN A DOUBLY LINKED LIST