If you are inserting a new node into an empty linked list, the behavior depends on how the insertion is handled. Here's what typically happens:

1. **Create the new node:** You create the new node with the desired value.
2. **Update the head pointer:** Since the list is empty, the head of the list (or the first node) will be null or None. You now update the head to point to the new node, making it the first node in the list.
3. **Set the next pointer of the new node:** Since the new node is the only node in the list, its next pointer will be null or None, indicating the end of the list.

The new node becomes the head of the linked list. After this insertion, the list contains exactly one node, and the head points to that node, with the next pointer of the node being null (or None).

4o mini

how will i get the base address of circular linked list

In a **circular linked list**, the "base address" typically refers to the memory address of the first node (i.e., the head node), which serves as the starting point for traversal or accessing the entire list. However, since a circular linked list does not have a null pointer at the end (instead, the last node points back to the head), the term "base address" could mean:

1. The memory address of the first node (head).
2. The address where the list can be accessed or iterated starting from the head.

Here are the typical steps to obtain the "base address" (the starting point) of a circular linked list:

**Steps to Access the Base Address of a Circular Linked List:**

1. **Access the head pointer**: The circular linked list generally maintains a pointer (or reference) to the first node in the list, often called head or start.
   * In this case, head points to the first node in the circular linked list.
   * The "base address" is simply the address stored in the head pointer.
2. **Node structure in a Circular Linked List**: Each node in the circular linked list contains:
   * A data value.
   * A pointer to the next node.

In the last node, instead of next being null, it points back to the first node (head), creating a loop.

**Example in C-like pseudocode:**

c

Copy code

struct Node {

int data;

struct Node\* next;

};

// Function to get the base address

struct Node\* getBaseAddress(struct Node\* head) {

return head; // The base address is simply the address of the head node

}

**Key Points:**

* If the list is **empty**, the head pointer will be null (or None in Python), indicating there is no base address.
* If the list is **not empty**, the head pointer holds the base address of the first node in the circular list.

In a circular linked list, once you have the head node, you can traverse the list by following the next pointers, and because it’s circular, you’ll eventually return to the head node after a complete cycle.

4o mini

**which will become base address if we delete first node**

If you delete the **first node** in a **circular linked list**, the "base address" (or the starting point) of the list will change. Specifically, the new **base address** will become the second node in the list (or the node that was previously the second node). The steps involved in updating the base address when the first node is deleted are as follows:

**Steps to Update the Base Address After Deleting the First Node:**

1. **Identify the first node** (head) and the second node.
   * The first node is pointed to by the head pointer.
   * The second node is the node that the first node points to (i.e., head->next).
2. **Update the head pointer**:
   * After deleting the first node, the head pointer should be updated to point to the second node (head->next).
   * If the second node exists, the head will now point to this new node, which becomes the new base address for the circular linked list.
3. **Handle the circular reference**:
   * The last node in the list should now point to the new first node (which was previously the second node).
   * Specifically, the next pointer of the last node should be updated to point to the new head.
4. **Edge case: Deleting the only node**:
   * If the list only contains one node and you delete it, the list becomes empty, and the head pointer will be null or None. In this case, there is no base address since the list is empty.

**Example (in pseudocode):**

Suppose we have a circular linked list with nodes 1 → 2 → 3 → 1 (back to the first node):

* Initially, the head points to node 1 (the base address).

After deleting the first node (node 1):

* We update the head to point to the second node, which is node 2.
* We ensure the last node (node 3) points back to the new head (node 2), maintaining the circular nature.

**C-like pseudocode for Deletion and Base Address Update:**

c

Copy code

struct Node {

int data;

struct Node\* next;

};

void deleteFirstNode(struct Node\*\* head) {

if (\*head == NULL) { // List is empty

return;

}

struct Node\* temp = \*head;

if ((\*head)->next == \*head) { // Only one node in the list

free(\*head);

\*head = NULL;

return;

}

// Update head to point to the second node

\*head = (\*head)->next;

// Find the last node

struct Node\* last = \*head;

while (last->next != temp) {

last = last->next;

}

// Last node now points to the new head

last->next = \*head;

// Free the original first node

free(temp);

}

* **Before deletion:** The head points to the first node, and the last node points to the first node.
* **After deletion:** The head points to the second node, and the last node now points to the new head (previously the second node), maintaining the circular reference.

If there is only one node, deleting it will make the list empty, and there will be no base address (head will be null or None).

**in doubly linked list**

In a **doubly linked list**, deleting the **first node** changes the base address (head pointer) of the list. The structure of a doubly linked list means that each node has two pointers:

* A prev pointer pointing to the previous node.
* A next pointer pointing to the next node.

When the first node is deleted, the head pointer of the list will be updated to point to the next node (the second node), and the prev pointer of that second node should be updated to null since it will become the new first node.

**Steps to Update the Base Address After Deleting the First Node in a Doubly Linked List:**

1. **Identify the first node**: This is the node that the head pointer points to.
   * The first node has no prev pointer (it’s null).
   * The second node (if it exists) is the node that the head->next pointer refers to.
2. **Update the head pointer**: After deleting the first node:
   * The head should be updated to point to the second node (head->next).
   * If the second node exists, its prev pointer should be set to null (because it's the new first node in the list).
3. **Handle the case of an empty list**:
   * If the list contains only one node and that node is deleted, the head should be set to null, and the list will be empty.
4. **Memory management**: The memory occupied by the deleted node needs to be freed to prevent memory leaks.

**Example (in pseudocode):**

Suppose we have a doubly linked list with nodes 1 ↔ 2 ↔ 3:

* Initially, head points to node 1.

After deleting the first node (node 1):

* The head pointer is updated to point to node 2.
* The prev pointer of node 2 is set to null.
* Node 1 is removed from the list and its memory is freed.

**C-like pseudocode for Deleting the First Node in a Doubly Linked List:**

c

Copy code

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

// Function to delete the first node in the doubly linked list

void deleteFirstNode(struct Node\*\* head) {

if (\*head == NULL) { // List is empty

return;

}

struct Node\* temp = \*head;

if ((\*head)->next == NULL) { // Only one node in the list

free(\*head);

\*head = NULL;

return;

}

// Update head to point to the second node

\*head = (\*head)->next;

// Set the prev pointer of the new first node to NULL

(\*head)->prev = NULL;

// Free the original first node

free(temp);

}

**Key Points:**

* **Before deletion:** The head points to the first node, and the first node’s prev pointer is null, while the second node's prev pointer points to the first node.
* **After deletion:** The head now points to the second node, the second node's prev pointer is updated to null, and the memory for the first node is freed.

**Edge Cases:**

1. **List with one node**: If there is only one node in the list, deleting it will make the list empty, and the head will become null.
2. **List with multiple nodes**: The head will be updated to point to the second node, and the previous pointer of the new first node (second node) will be set to null.

In a doubly linked list, deleting the first node involves reassigning the head and adjusting the prev pointer of the new first node, ensuring that the doubly linked structure is maintained.

**STACK**

Last in first out

 **LIFO** (Last In, First Out): Last item added is the first to be removed.

* Real-world analogy: Stack of plates, undo in text editors, function calls.

 **FIFO** (First In, First Out): First item added is the first to be removed.

* Real-world analogy: Queue at a ticket counter, print job scheduling, customer service lines.

the size of the stack is considered **fixed** because the underlying data structure (array) has a predefined, constant size. This means that once you declare an array to hold the stack, you cannot dynamically resize it as you add or remove elements.

**Count the no. of nodes in linked list:**

**Steps to Count Nodes in a Linked List:**

1. Start from the **head** of the linked list.
2. Initialize a counter to **0**.
3. Traverse the list node by node, incrementing the counter for each node.
4. Stop when you reach the **end** of the list (when the next pointer of the current node is NULL).
5. The counter will hold the number of nodes in the linked list.

**Example Code: Counting Nodes in a Linked List**

Here’s a C program to demonstrate how to count the number of nodes in a singly linked list:

c

Copy code

#include <stdio.h>

#include <stdlib.h>

// Define a structure for the linked list node

struct Node {

int data;

struct Node\* next;

};

// Function to count the number of nodes in the linked list

int countNodes(struct Node\* head) {

int count = 0; // Initialize the counter

struct Node\* current = head; // Start from the head node

// Traverse the list until we reach the end (NULL)

while (current != NULL) {

count++; // Increment the count for each node

current = current->next; // Move to the next node

}

return count; // Return the total count of nodes

}

// Function to create a new node and insert it at the end

void append(struct Node\*\* head, int newData) {

// Allocate memory for a new node

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

struct Node\* last = \*head; // Used to traverse the list to the last node

newNode->data = newData; // Assign the data to the new node

newNode->next = NULL; // New node will be the last node, so next is NULL

// If the list is empty, make the new node the head

if (\*head == NULL) {

\*head = newNode;

return;

}

// Otherwise, traverse to the last node

while (last->next != NULL) {

last = last->next;

}

// Link the last node to the new node

last->next = newNode;

}

// Function to display the list

void display(struct Node\* head) {

if (head == NULL) {

printf("The list is empty.\n");

return;

}

struct Node\* temp = head;

while (temp != NULL) {

printf("%d -> ", temp->data);

temp = temp->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL; // Initialize an empty list

// Add some nodes to the list

append(&head, 10);

append(&head, 20);

append(&head, 30);

append(&head, 40);

// Display the linked list

printf("Linked List: ");

display(head);

// Count the number of nodes in the list

int nodeCount = countNodes(head);

printf("Number of nodes in the linked list: %d\n", nodeCount);

return 0;

}

**Stack**

A stack follows the **Last In, First Out (LIFO)** principle, meaning that the last element inserted is the first one to be removed.

**Stack Operations:**

* **Push**: Add an element to the top of the stack.
* **Pop**: Remove the element from the top of the stack.
* **Peek/Top**: Retrieve the element from the top without removing it.
* **isEmpty**: Check if the stack is empty.
* **isFull**: Check if the stack is full (in case of array implementation).

**Queue**

A queue follows the **First In, First Out (FIFO)** principle, meaning that the first element inserted is the first one to be removed.

**Queue Operations:**

* **Enqueue**: Add an element to the rear of the queue.
* **Dequeue**: Remove the element from the front of the queue.
* **Front**: Retrieve the element from the front without removing it.
* **isEmpty**: Check if the queue is empty.
* **isFull**: Check if the queue is full (in case of array implementation).