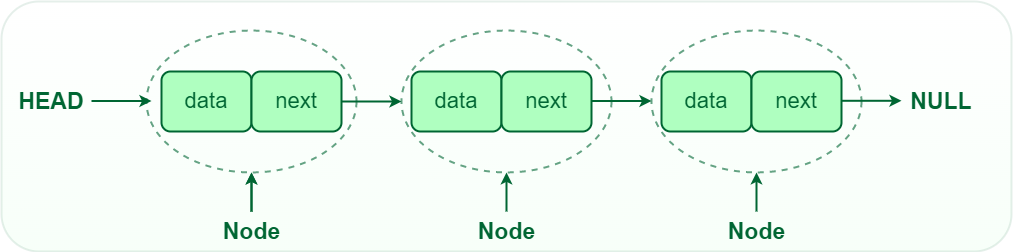
**Linked List**

**Linked List** là một cấu trúc dữ liệu tuyến tính, trong đó các phần tử không được lưu trữ ở một vị trí liền kề mà chúng được liên kết bằng con trỏ. Linke List tạo thành một chuỗi các node được kết nối, trong đó mỗi node lưu trữ dữ liệu và địa chỉ của node tiếp theo.



**Node Structure**: Một node trong danh sách liên kết thường bao gồm hai thành phần:

* ***Data***: Nó chứa giá trị thực hoặc dữ liệu được liên kết với node.
* ***Next Pointer***: Nó lưu trữ địa chỉ bộ nhớ (tham chiếu) của node tiếp theo trong chuỗi.

**Head and Tail**: Linked List được truy cập thông qua head node, node này trỏ đến first node trong danh sách. Last node trong danh sách trỏ tới NULL hoặc nullptr, cho biết kết thúc danh sách. Node này được gọi là tail node.

**Ưu điểm:**

* Cấu trúc dữ liệu động: Kích thước của bộ nhớ có thể được phân bổ hoặc hủy phân bổ trong thời gian chạy dựa trên thao tác chèn hoặc xóa.
* Dễ dàng chèn/xóa: Việc chèn và xóa các phần tử đơn giản hơn mảng vì không cần thay đổi phần tử nào sau khi chèn và xóa, chỉ cần cập nhật địa chỉ.
* Sử dụng bộ nhớ hiệu quả: Như chúng ta đã biết Linked List là một cấu trúc dữ liệu động, kích thước tăng hoặc giảm theo yêu cầu để tránh lãng phí bộ nhớ.
* Triển khai: Các cấu trúc dữ liệu nâng cao khác nhau có thể được triển khai bằng cách sử dụng Linked List như stack, queue, graph, hash map, v.v.

**Các loại danh sách liên kết:**

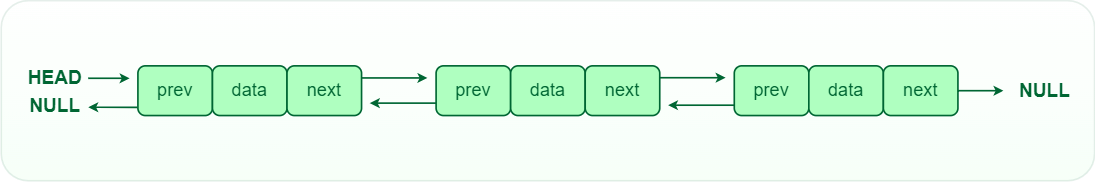
1. **Singly linked list (Danh sách liên kết đơn)**

**Singly linked list** là cấu trúc dữ liệu tuyến tính trong đó các phần tử không được lưu trữ ở các vị trí bộ nhớ liền kề và mỗi phần tử chỉ được kết nối với phần tử tiếp theo bằng con trỏ.



1. **Double – linked list (Danh sách liên kết đôi)**

Trong **Double – linked list**, mỗi node chứa các tham chiếu đến cả node tiếp theo và node trước đó. Điều này cho phép di chuyển theo cả hai hướng tiến và lùi, nhưng nó yêu cầu bộ nhớ bổ sung để tham chiếu lùi



Việc chèn một node mới vào danh sách liên kết đôi rất giống với việc chèn node mới vào danh sách liên kết. Cần phải thực hiện thêm một chút công việc để duy trì liên kết của node trước đó. Một node có thể được chèn vào **Double – linked list** theo bốn cách:

* Ở phía trước của DLL.
* Ở giữa hai node

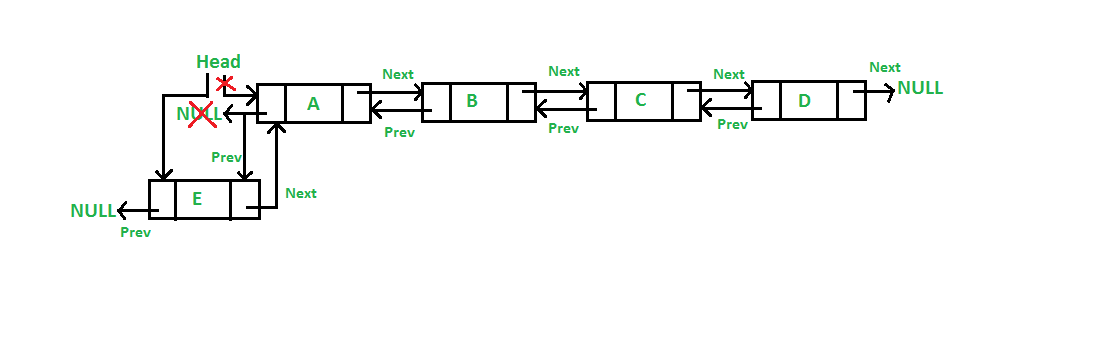
Sau một node nhất định.

Trước một node nhất định.

* Ở cuối tệp DLL.
  1. **Thêm một node ở phía trước trong DDL**

Node mới luôn được thêm vào trước phần đầu của Danh sách liên kết đã cho. Tác vụ có thể được thực hiện bằng cách sử dụng 5 bước sau:

* Đầu tiên, phân bổ một node mới (giả sử new\_node ).
* Bây giờ đặt dữ liệu cần thiết vào node mới.
* Đặt điểm tiếp theo của new\_node vào phần đầu hiện tại của danh sách liên kết đôi.
* Đặt điểm trước của điểm đầu hiện tại thành new\_node .
* Cuối cùng, trỏ tới new\_node



**void** push(**struct** Node\*\* head\_ref, **int** new\_data)

{

    // 1. allocate node

**struct** Node\* new\_node

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

    // 2. put in the data

    new\_node->data = new\_data;

    // 3. Make next of new node as head and previous as NULL

    new\_node->next = (\*head\_ref);

    new\_node->prev = NULL;

    // 4. change prev of head node to new node

**if** ((\*head\_ref) != NULL)

        (\*head\_ref)->prev = new\_node;

    // 5. move the head to point to the new node

    (\*head\_ref) = new\_node;

}

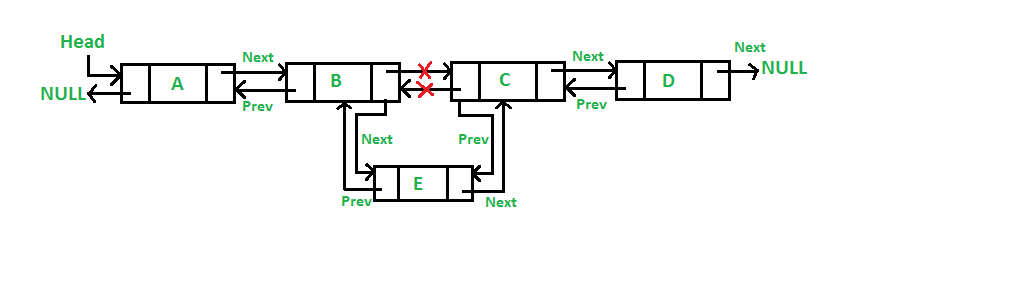
* 1. **Thêm một node ở giữa hai node**

Nó còn được phân loại thành hai phần sau:

* + 1. ***Thêm một node sau một node nhất định trong Danh sách liên kết đôi:***

Chúng ta được cấp một con trỏ tới một node là prev\_node và node mới được chèn sau node đã cho. Điều này có thể được thực hiện bằng 6 bước sau:

* Đầu tiên tạo một node mới (giả sử new\_node ).
* Bây giờ hãy chèn dữ liệu vào node mới.
* Trỏ phần tiếp theo của new\_node vào phần tiếp theo của prev\_node.
* Trỏ phần tiếp theo của prev\_node vào new\_node.
* Trỏ phần trước của new\_node vào prev\_node.
* Thay đổi con trỏ trước đó của node mới thành new\_node.



**void** insertAfter(**struct** Node\* prev\_node, **int** new\_data)

{

    // Check if the given prev\_node is NULL

**if** (prev\_node == NULL) {

**printf**("the given previous node cannot be NULL");

**return**;

    }

    // 1. allocate new node

**struct** Node\* new\_node

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

    // 2. put in the data

    new\_node->data = new\_data;

    // 3. Make next of new node as next of prev\_node

    new\_node->next = prev\_node->next;

    // 4. Make the next of prev\_node as new\_node

    prev\_node->next = new\_node;

    // 5. Make prev\_node as previous of new\_node

    new\_node->prev = prev\_node;

    // 6. Change previous of new\_node's next node

**if** (new\_node->next != NULL)

        new\_node->next->prev = new\_node;

}

* + 1. **Thêm một node trước một node nhất định trong DDL**

Đặt con trỏ tới node đã cho này là next\_node . Điều này có thể được thực hiện bằng cách sử dụng 6 bước sau.

* Cấp phát bộ nhớ cho node mới, đặt tên là new\_node .
* Đặt dữ liệu vào new\_node .
* Đặt con trỏ trước của new\_node này làm node trước của next\_node .
* Đặt con trỏ trước của next\_node làm new\_node .
* Đặt con trỏ tiếp theo của new\_node này làm next\_node .
* Bây giờ hãy đặt con trỏ trước đó của new\_node .

Nếu node trước của new\_node không phải là NULL thì hãy đặt con trỏ tiếp theo của node trước đó là new\_node .

Ngược lại, nếu prev của new\_node là NULL thì đó sẽ là node đầu mới.

A diagram of a computer code

Description automatically generated

**void** insertBefore(**struct** Node\* next\_node, **int** new\_data)

{

    // Check if the given next\_node is NULL

**if** (next\_node == NULL) {

**printf**("the given next node cannot be NULL");

**return**;

    }

    // 1. Allocate new node

**struct** Node\* new\_node

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

    // 2. Put in the data

    new\_node->data = new\_data;

    // 3. Make previous of new node as previous of next\_node

    new\_node->prev = next\_node->prev;

    // 4. Make the previous of next\_node as new\_node

    next\_node->prev = new\_node;

    // 5. Make next\_node as next of new\_node

    new\_node->next = next\_node;

    // 6. Change next of new\_node's previous node

**if** (new\_node->prev != NULL)

        new\_node->prev->next = new\_node;

**else**

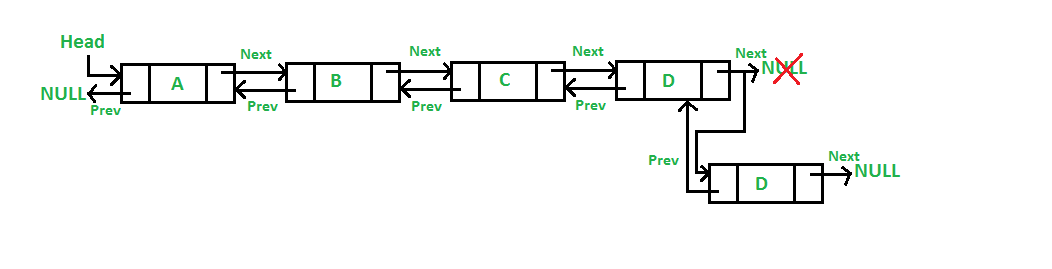
        head = new\_node;

}

* 1. **Thêm một node ở cuối trong DDL**

Node mới luôn được thêm vào sau node cuối cùng của Danh sách liên kết đã cho. Điều này có thể được thực hiện bằng 7 bước sau:

* Tạo một node mới (giả sử new\_node ).
* Đặt giá trị vào node mới.
* Đặt con trỏ tiếp theo của new\_node thành null.
* Nếu danh sách trống, hãy đặt new\_node làm phần đầu.
* Nếu không, hãy di chuyển đến cuối danh sách liên kết.
* Bây giờ hãy tạo con trỏ tiếp theo của node cuối cùng trỏ tới new\_node .
* Thay đổi con trỏ trước của new\_node thành node cuối cùng của danh sách.



**void** append(**struct** Node\*\* head\_ref, **int** new\_data)

{

    // 1. allocate node

**struct** Node\* new\_node

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

**struct** Node\* last = \*head\_ref; /\* used in step 5\*/

    // 2. put in the data

    new\_node->data = new\_data;

    // 3. This new node is going to be the last node, so

    // make next of it as NULL

    new\_node->next = NULL;

    // 4. If the Linked List is empty, then make the new

    // node as head

**if** (\*head\_ref == NULL) {

        new\_node->prev = NULL;

        \*head\_ref = new\_node;

**return**;

    }

    // 5. Else traverse till the last node

**while** (last->next != NULL)

        last = last->next;

    // 6. Change the next of last node

    last->next = new\_node;

    // 7. Make last node as previous of new node

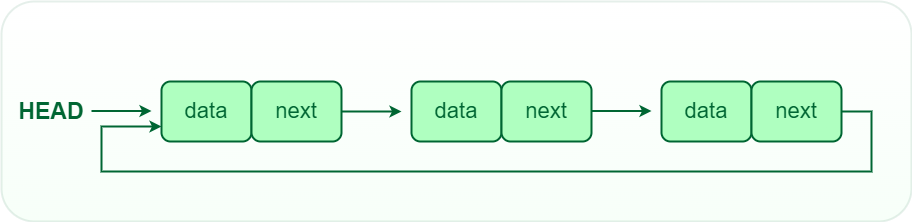
    new\_node->prev = last;

**return**;

}

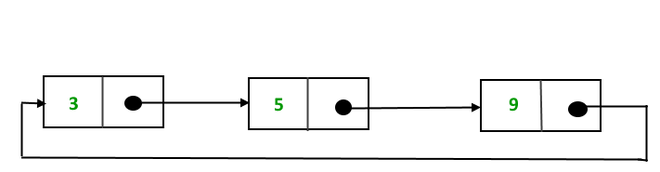
1. **Circular linked list:**

Trong Circular linked list, node cuối cùng trỏ về node đầu, tạo ra cấu trúc vòng tròn. Nó có thể được liên kết đơn hoặc đôi. Không có NULL ở cuối.

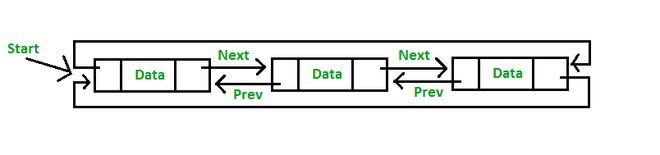


Nhìn chung có hai loại danh sách liên kết vòng:

* **Circular singly linked list** (Danh sách liên kết đơn vòng): Trong danh sách liên kết đơn vòng, node cuối cùng của danh sách chứa một con trỏ tới node đầu tiên của danh sách. Chúng ta duyệt qua danh sách liên kết đơn vòng tròn cho đến khi đến được node mà chúng ta đã bắt đầu. Danh sách liên kết đơn vòng tròn không có điểm bắt đầu và kết thúc. Không có giá trị null nào xuất hiện trong phần tiếp theo của bất kỳ node nào.



* **Circular Doubly linked list** (Danh sách liên kết đôi tròn): Danh sách liên kết đôi tròn có các thuộc tính của cả danh sách liên kết đôi và danh sách liên kết vòng trong đó hai phần tử liên tiếp được liên kết hoặc kết nối bởi con trỏ trước và con trỏ tiếp theo và node cuối cùng trỏ đến node đầu tiên bởi con trỏ tiếp theo và cũng node đầu tiên trỏ đến node cuối cùng bằng con trỏ trước đó.

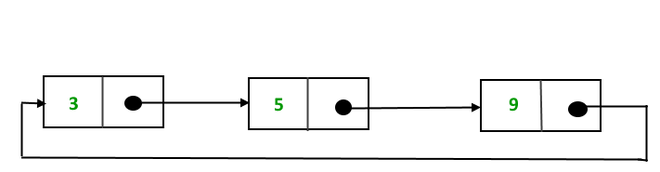


**Lưu ý:** Chúng ta sẽ sử dụng danh sách liên kết vòng đơn để thể hiện hoạt động của danh sách liên kết vòng.

**Biểu diễn danh sách liên kết vòng:**

Danh sách liên kết vòng tương tự như Danh sách liên kết đơn ngoại trừ việc kết nối node cuối cùng với node đầu tiên.

Biểu diễn node của Danh sách liên kết vòng:



Node\* one = createNode(3);

Node\* two = createNode(5);

Node\* three = createNode(9);

// Connect nodes

one->next = two;

two->next = three;

three->next = one;

**Các thao tác trên danh sách liên kết vòng:**

Chúng ta có thể thực hiện một số thao tác trên danh sách liên kết vòng tương tự như danh sách liên kết đơn đó là: Chèn và Xóa.

* 1. **Chèn vào danh sách liên kết vòng:**

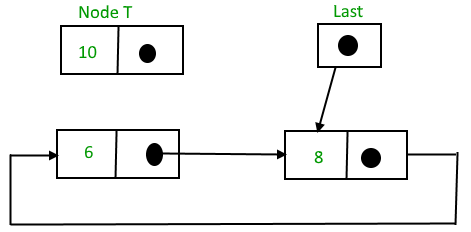
***Một node có thể được thêm vào theo ba cách:***

* Chèn vào đầu danh sách
* Chèn vào cuối danh sách
* Chèn vào giữa các node
  + 1. **Chèn vào đầu danh sách:**

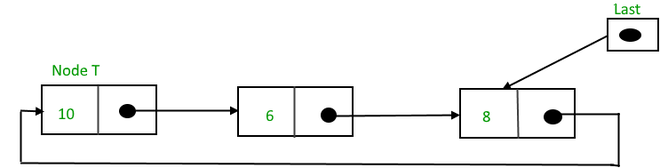
Để chèn một node vào đầu danh sách, hãy làm theo các bước sau:

* Tạo một node, ví dụ T.
* Tạo T -> next = last -> next.
* last -> next = T.

Danh sách liên kết vòng trước khi chèn



Và sau đó,

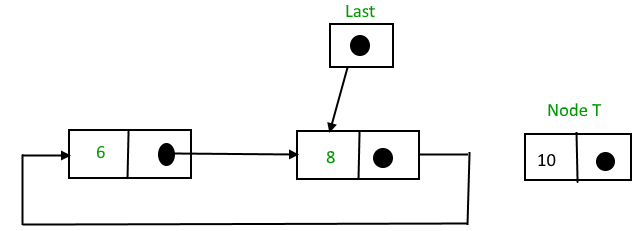


* + 1. **Chèn vào cuối danh sách:**

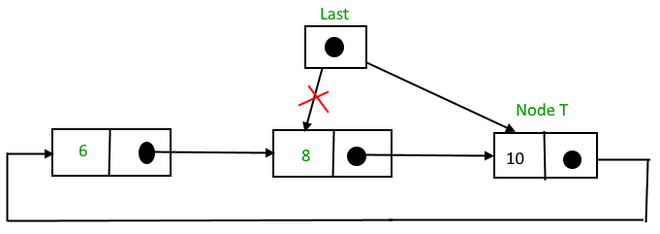
Để chèn một node vào cuối danh sách, hãy làm theo các bước sau:

* Tạo một node, ví dụ T.
* Tạo T -> next = last -> next;
* last -> next = T.
* last = T

Trước khi chèn,



Sau khi chèn,

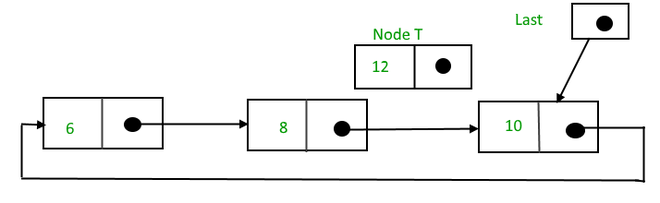


* + 1. **Chèn vào giữa các node:**

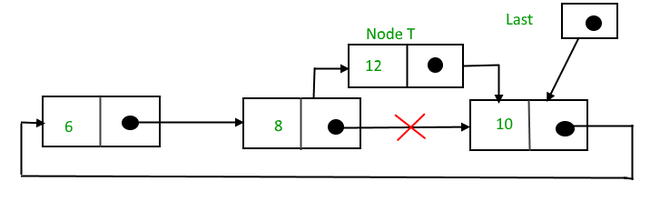
Để chèn một node vào giữa hai node, hãy làm theo các bước sau:

* Tạo một node, ví dụ T.
* Tìm kiếm node mà sau đó T cần được chèn vào, giả sử node đó là P.
* Tạo T -> next = P -> next;
* P -> next = T.

Giả sử 12 cần được chèn sau node có giá trị 10,



Sau khi tìm kiếm và chèn,



* 1. **Xóa trong danh sách liên kết vòng:**
     1. **Chỉ xóa node nếu đó là node duy nhất trong danh sách liên kết vòng:**
* Giải phóng bộ nhớ của node
* Giá trị cuối cùng phải là NULL Một node luôn trỏ đến một node khác, do đó việc gán NULL là không cần thiết.
* Bất kỳ node nào cũng có thể được đặt làm điểm bắt đầu.
* Các node được duyệt nhanh chóng từ node đầu tiên đến node cuối cùng.
  + 1. **Xóa node cuối cùng:**
* Xác định vị trí node trước node cuối cùng (đặt nó là temp)
* Giữ địa chỉ của node bên cạnh node cuối cùng trong temp
* Xóa bộ nhớ cuối cùng
* Đặt temp ở cuối
  + 1. **Xóa node bất kỳ khỏi danh sách liên kết vòng:**

Chúng ta sẽ được cấp một node và nhiệm vụ của chúng ta là xóa node đó khỏi danh sách liên kết vòng.

**Thuật toán:**

***Trường hợp 1:*** Danh sách trống.

Nếu danh sách trống, chúng tôi sẽ quay lại.

***Trường hợp 2:*** Danh sách không trống

* Nếu danh sách không trống thì chúng ta xác định hai con trỏ curr và prev và khởi tạo curr con trỏ bằng head node.
* Duyệt danh sách bằng cách sử dụng curr để tìm node cần xóa và trước khi chuyển sang node tiếp theo, mỗi lần đặt prev = curr.
* Nếu node được tìm thấy, hãy kiểm tra xem đó có phải là node duy nhất trong danh sách hay không. Nếu có, đặt head = NULL và free(curr).
* Nếu danh sách có nhiều node, hãy kiểm tra xem đó có phải là node đầu tiên của danh sách hay không. Điều kiện để kiểm tra điều này (curr == head). Nếu có thì di chuyển trước cho đến khi đến node cuối cùng. Sau khi prev đến node cuối cùng, đặt head = head -> next và prev -> next = head. Xóa curr.
* Nếu curr không phải là node đầu tiên, chúng tôi sẽ kiểm tra xem đó có phải là node cuối cùng trong danh sách hay không. Điều kiện để kiểm tra cái này là (curr -> next == head).
* Nếu Curr là node cuối cùng. Đặt prev -> next = head và xóa node hiện tại bằng free(curr).
* Nếu node cần xóa không phải là node đầu tiên cũng không phải node cuối cùng thì đặt prev -> next = curr -> next và xóa curr.
* Nếu node không có trong danh sách, hãy trả về head và không làm gì cả.

Dưới đây là cách thực hiện cho phương pháp trên:

#include <stdio.h>

#include <stdlib.h>

// Structure for a node

**struct** Node {

**int** data;

**struct** Node\* next;

};

// Function to insert a node at the

// beginning of a Circular linked list

**void** push(**struct** Node\*\* head\_ref, **int** data)

{

    // Create a new node and make head

    // as next of it.

**struct** Node\* ptr1 = (**struct** Node\*)**malloc**(**sizeof**(**struct** Node));

    ptr1->data = data;

    ptr1->next = \*head\_ref;

    // If linked list is not NULL then

    // set the next of last node

**if** (\*head\_ref != NULL) {

        // Find the node before head and

        // update next of it.

**struct** Node\* temp = \*head\_ref;

**while** (temp->next != \*head\_ref)

            temp = temp->next;

        temp->next = ptr1;

    }

**else**

        // For the first node

        ptr1->next = ptr1;

    \*head\_ref = ptr1;

}

// Function to print nodes in a given

// circular linked list

**void** printList(**struct** Node\* head)

{

**struct** Node\* temp = head;

**if** (head != NULL) {

**do** {

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

            temp = temp->next;

        } **while** (temp != head);

    }

**printf**("\n");

}

// Function to delete a given node

// from the list

**void** deleteNode(**struct** Node\*\* head, **int** key)

{

    // If linked list is empty

**if** (\*head == NULL)

**return**;

    // If the list contains only a

    // single node

**if** ((\*head)->data == key && (\*head)->next == \*head) {

**free**(\*head);

        \*head = NULL;

**return**;

    }

**struct** Node \*last = \*head, \*d;

    // If head is to be deleted

**if** ((\*head)->data == key) {

        // Find the last node of the list

**while** (last->next != \*head)

            last = last->next;

        // Point last node to the next of

        // head i.e. the second node

        // of the list

        last->next = (\*head)->next;

**free**(\*head);

        \*head = last->next;

**return**;

    }

    // Either the node to be deleted is

    // not found or the end of list

    // is not reached

**while** (last->next != \*head && last->next->data != key) {

        last = last->next;

    }

    // If node to be deleted was found

**if** (last->next->data == key) {

        d = last->next;

        last->next = d->next;

**free**(d);

    }

**else**

**printf**("Given node is not found in the list!!!\n");

}

// Driver code

**int** main()

{

    // Initialize lists as empty

**struct** Node\* head = NULL;

    // Created linked list will be

    // 2->5->7->8->10

    push(&head, 2);

    push(&head, 5);

    push(&head, 7);

    push(&head, 8);

    push(&head, 10);

**printf**("List Before Deletion: ");

    printList(head);

    deleteNode(&head, 7);

**printf**("List After Deletion: ");

    printList(head);

**return** 0;

}