Elementary Data Structures: Linked Lists

Dr. Keyur Parmar

National Institute of Technology (NIT), Surat
Email: keyur@coed.svnit.ac.in

Outline

Introduction

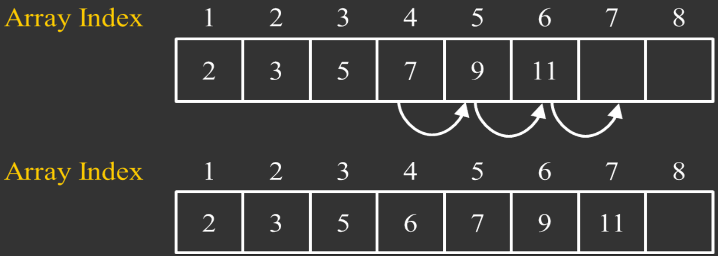
We can store the elements of finite dynamic sets using arrays. For example,

- array of integers
- array of characters
- array of structures, etc.

Then, why do we need linked lists to store the elements of finite dynamic sets?

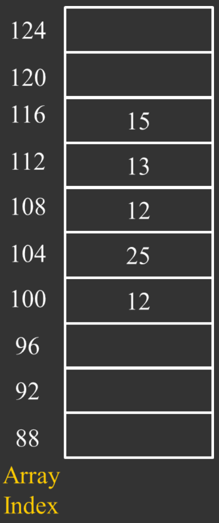
Arrays Vs. Linked Lists

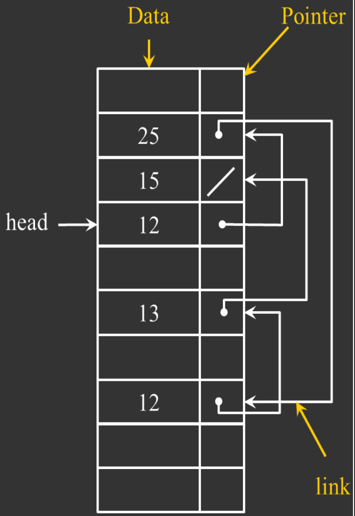
- Insertion and deletion at intermediate positions
  - Array: Time consuming if many elements need to be shifted. In the worst-case, it is $\theta(n)$.
  - Linked list: Efficient as only the pointers need to be updated. In the worst-case, it is $\theta(1)$.[1]
- For example, insert 6 in a sorted array.



1 - Searching the position may take time proportional to n.

Arrays Vs. Linked Lists





- A dynamic set S = {12, 25, 12, 13, 15}
- Order of elements (in memory)
  - Array: Stored consecutively
  - Linked list: Stored randomly

Arrays Vs. Linked Lists

- Memory utilization
  - Array: Not efficient (as it is very difficult to predict the size of the array at the beginning)
  - Linked list: Efficient

Why do we use arrays rather than linked lists to store the elements of finite dynamic sets?

Arrays Vs. Linked Lists

- Accessing an element
  - Array: Only takes $\theta(1)$ time to access an element.
  - Linked list: Takes $\theta(n)$ time to access an element.

Arrays Vs. Linked Lists

- Size
  - Array: Size of the array is constant and decided at the time of compilation.
  - Linked list: Size of the linked list is dynamic and can be modified during the runtime.

- Searching an element
  - Array: Array elements are stored sequentially and therefore we can search an element efficiently using the binary search (if elements are stored in sorted order).
  - Linked list: To search an element, linked list can use only linear search (and cannot use binary search).

Arrays Vs. Linked Lists

- Extra memory/space per element
  - Array: NULL
  - Linked list: Extra space/memory is required to store/represent links between elements. For example, a pointer to the next element of the list, head, etc.

Linked Lists

What is a linked list?

Linked Lists

- Linked list: A data structure in which the objects/nodes are arranged in a linear order.
- A linked list is an alternative to an array-based structure.
- A linked list is a collection of nodes.
- Each node consists of
  1. Data
  2. Reference to the next and/or previous node of the list

Linked Lists - Example

- A linked list representing a finite dynamic set $S = \{1, 3, 5, 7\}$ is as follows.

- If the node is a last element of the list, the reference to the next node of the list is set to NULL (represented by "/").

Linked Lists

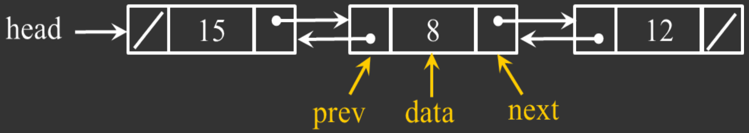
- Array index is used to find the location of an element in the array in constant time.
- Elements of a linked list are not consecutive in the memory, and to access an element of a linked list, we have to traverse the links.
- To access the first element of a linked list, set an attribute "head" that points to the first element.

Types of Linked Lists

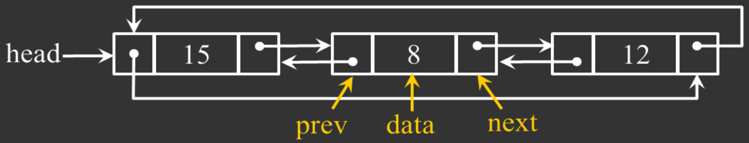
- Singly linked lists
- Doubly linked lists
- Circular linked lists
- Sorted linked lists

Singly Linked Lists - No “prev” Pointer

Doubly Linked Lists - Both "next" and "prev" Pointer



Circular Doubly Linked Lists



Sorted Singly Linked Lists

- The “data” attribute of nodes is used to sort the linked list.

Singly Linked List

Singly Linked Lists

- Each node of the linked list consists of
  1. Data
  2. A link to access the “next” node in the list. – A variable that store the address of the “next” node in the list.

Singly Linked Lists

- Given a node `x` in the list
  - `x.next` points to the successor of `x` in the list.
  - If `x.next` is `NULL`, then `x` is the last node of the list.
- Attribute "`head`" points to the first node of the list.
- If `head` is `NULL`, then the linked list is empty.

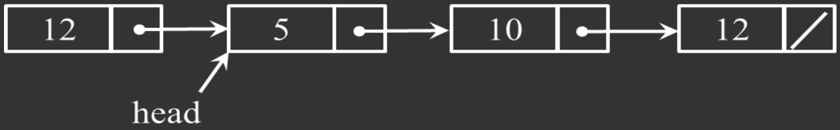
Singly Linked Lists - Insert a Node at the Front

Goal: Insert “12” into the linked list at the front of the list.

1. Create a node.

Singly Linked Lists - Insert a Node at the Front

2. Set a node's "next" attribute points to the node the "head" points.



3. Set the "head" to point to the newly created node.

Singly Linked List - Insert a Node at the Front

- Goal: Insert a node into the linked list at the front of the list.

```
List_Insert_First(L, x)
x.next = head
head = x
```

- The running time of `List_Insert_First` operation is $\theta(1)$.

Singly Linked List - Insert at Front - Function in C Prog. Language

```c
#include <stdio.h>
#include <stdlib.h>

struct node  // Structure of each node
{
    int data;
    struct node* next;
};

struct node* head = NULL; // Head pointer of a singly linked list
```

Singly Linked List - Insert at Front - Function in C Prog. Language

```
void list_insert_first(int x) // Insert a
    node at the beginning of the list
{
    struct node* tmp = (struct node*) malloc
        (sizeof(struct node));

    (*tmp).data = x;
    (*tmp).next = head;
    head = tmp;
}
```

Singly Linked List - Remove a Node From the Front

Goal: Remove a node containing "7" from the linked list.

Singly Linked List - Remove a node From the Front

- Goal: Remove a node from the linked list.

```
List_Remove_First(L)
if head == NULL then
    print "List is empty."
else
    head = head.next
```

- The running time of `List_Remove_First` operation is $\theta(1)$.

Singly Linked List - Remove From Front - Function in C

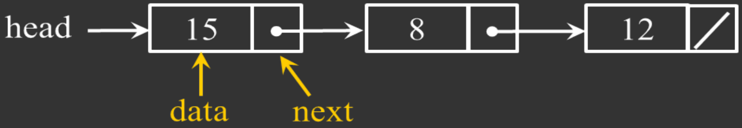
```c
#include <stdio.h>
#include <stdlib.h>

struct node  // Structure of each node
{
    int data;
    struct node* next;
};

struct node* head = NULL; // Head pointer of a singly linked list
```

Singly Linked List - Remove From Front - Function in C

```c
void list_remove_first() // Delete a node from the beginning of the list
{
    if (head == NULL)
    {
        printf("\nLinked list is empty.");
    }
    else
    {
        struct node* tmp = head;
        head = (*head).next;
        free(tmp);
    }
}
```

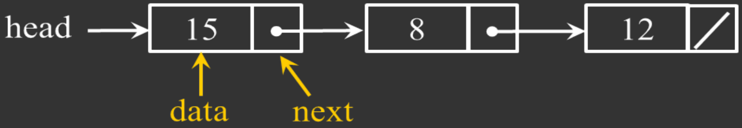
Singly Linked Lists - Disadvantage

- It is easy to remove the front node from the singly linked list.



- Can we easily remove the last node from the singly linked list?
    - No, to remove the last node from the linked list, we need to have access of the "next" pointer of the previous node to the last node. However, we cannot access the "next" pointer of the previous node (easily).

Singly Linked Lists - Disadvantage



- Can we easily remove the intermediate node from the singly linked list?
  - No, to remove the intermediate node from the singly linked list, we need to have access of the "next" pointer of the previous node to the intermediate node. However, we cannot access the "next" pointer of the previous node to the intermediate node (easily).

Singly Linked Lists - Disadvantage

- In a singly linked list, we only have "next" pointers. A singly linked list does not have "prev" pointers to access the previous node from the current node.
- Along with "head" pointer if we implement a "tail" pointer, then also we cannot access the previous node of the last/intermediate node easily.
- To efficiently support such operations, i.e., to remove the last/intermediate node from the linked list, we need a "Doubly Linked List".

Doubly Linked List

Doubly Linked List

- Each node of the doubly linked list consists of
  1. Data
  2. Link to the "`next`" node in the list
  3. Link to the "`prev`" node in the list

Doubly Linked List

- Given a node `x` of the linked list

  1. `x.next` points to the successor of `x` in the linked list.
  2. `x.prev` points to the predecessor of `x` in the linked list.
  3. If `x.next` is `NULL`, then `x` is the last node of the linked list.
  4. If `x.prev` is `NULL`, then `x` is the first node of the linked list.

Search an Element in the Doubly Linked List

- Given a node `x` of the linked list
  1. `x.key`: data stored/represented by `x`.
  2. `x.next`: points to the successor of `x` in the linked list.
- Input:
  1. A linked list `L`.
  2. A key `k`.
- Output:
  1. Return a node with key `k` or return NULL.

Doubly Linked List - `List_Search(L, k)`

```
List_Search(L, k)
x = head
while x ≠ NULL and x.key ≠ k do
    x = x.next
return x
```

- In the best case, the running time of `List_Search` operation is $\theta(1)$ – when the node is at the head/front of the linked list.
- In the worst case, the running time of `List_Search` operation is $\theta(n)$ – when the node is not in the list.
- The `List_Search` operation remains same for singly and doubly linked lists.

Doubly Linked List - Search a Node - Function in C

```c
#include <stdio.h>
#include <stdlib.h>

struct node  // Structure of each node
{
    int data;
    struct node* next;
    struct node* prev;
};

struct node* head = NULL; // Head pointer of a singly linked list
```

Doubly Linked List - Search a Node - Function in C

```c
struct node* list_search(int x) // Search a node in the linked list
{
    struct node* tmp = head;

    while ((tmp!=NULL) && ((*tmp).data!=x))
    {
        tmp = (*tmp).next;
    }

    return tmp;
}
```

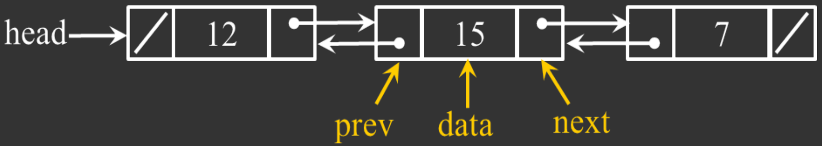
Insert a Node in the Linked List

- Given a node `x` where attributes
  - `x.key = value`
  - `x.next = NULL`
  - `x.prev = NULL`

  insert `x` at the front of the linked list `L`.

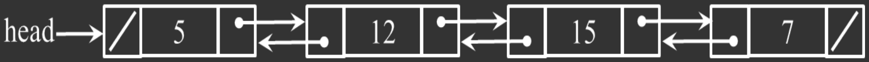
Insert a Node in the Linked List

- Linked List:



- Goal: Insert a node.

- Output:



List_Insert(L, x)

- Goal: Insert a node at the front of the linked list.

List_Insert(L, x)

```
x.next = head
if head ≠ NULL then
    head.prev = x
head = x
x.prev = NULL
```

- The running time of List_Insert operation is $\theta(1)$.

Doubly Linked List - Insert a Node - Function in C

```c
#include <stdio.h>
#include <stdlib.h>

struct node  // Structure of each node
{
    int data;
    struct node* next;
    struct node* prev;
};

struct node* head = NULL; // Head pointer of a doubly linked list
```

Doubly Linked List - Insert a Node - Function in C

```c
void list_insert_first(int x) // Insert a node at the beginning of the list
{
    struct node* tmp = (struct node*) malloc(sizeof(struct node));
    (*tmp).next = head;
    if (head != NULL)
    {
        (*head).prev = tmp;
    }
    (*tmp).data = x;
    (*tmp).prev = NULL;
    head = tmp;
}
```

Delete a Node From the Linked List

- Given a node `x` where attributes
  - `x.key`
  - `x.next`
  - `x.prev`

  delete a node `x` from the linked list `L`.

Delete a Node From the Linked List

- Goal: Delete a node `x` from the linked list `L`.

```
List_Delete(L, x)

if x.prev ≠ NULL then
    x.prev.next = x.next
else
    head = x.next
if x.next ≠ NULL then
    x.next.prev = x.prev
```

- The running time of `List_Delete` operation is $\theta(1)$.
- However, in the worst-case, searching an item (to be deleted) takes $\theta(n)$ time.

Doubly Linked List - Delete a Node - Function in C

```
#include <stdio.h>
#include <stdlib.h>

struct node  // Structure of each node
{
int data;
struct node* next;
struct node* prev;
};

struct node* head = NULL; // Head pointer of a doubly linked list
```

Doubly Linked List - Delete a Node - Function in C

```
void list_remove(int x) // Delete a node
    from the list
{
    if (head == NULL)
    {
        printf("\nLinked list is empty.");
    }
    else
    {
        struct node* tmp = list_search(x);

        if (tmp == NULL)
        {
            printf("\nElement is not in the
                linked list.");
```

Doubly Linked List - Delete a Node - Function in C

```
            return;
        }

        if ((*tmp).prev != NULL)
        {
            (*(*tmp).prev).next = (*tmp).next;
        }
        else
        {
            head = (*tmp).next;
        }

        if ((*tmp).next != NULL)
        {
            (*(*tmp).next).prev = (*tmp).prev;
```

Doubly Linked List - Delete a Node - Function in C

```
        }

        free(tmp);
    }
}
```

Circular Linked Lists

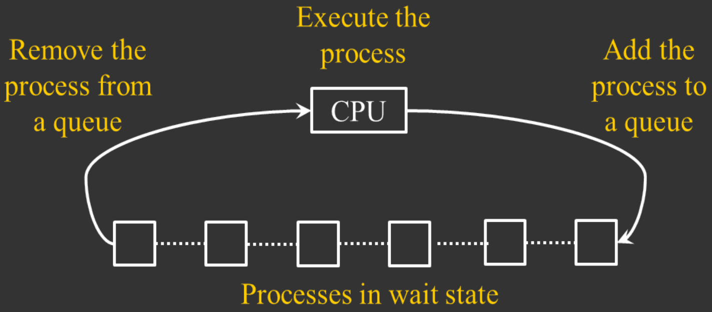
Circular Linked Lists - Applications

- In many real-world applications, data may not be arranged in a linear order but rather arranged in a cyclic order without a fixed beginning and/or end.
- For example, two or more active processes share the use of central processing unit (CPU) in a cyclic order.
- Round-robin scheduling:
  - Each process is given a time slice to execute on a CPU.
  - As long as the processes remain active, they will get the time slice in a cyclic order.
  - New processes are added to the list and completed processes are removed from the list.

Round Robin Scheduling

1. Remove the process from the head of a linked list.
2. Execute the process for a particular time slice.
3. If the process remains incomplete, add the process at the tail of a linked list.

Round Robin Scheduling



Circular Linked Lists

- If the linked list is not circular, we remove a node (i.e., a process) from one end of the linked list and add the same node (i.e., a process) to the other end of the linked list.
- The above mechanism is inefficient.
- An efficient implementation of Round-Robin scheduling requires a "circular linked list".

Types of Circular Linked Lists

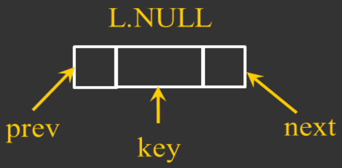
- Types of circular linked lists
  - Circular singly linked list
  - Circular doubly linked list without sentinels
  - Circular doubly linked list with sentinels
- Let's design a circular doubly linked list with sentinels.
- The use of sentinels makes the code simpler.

Sentinel

- To avoid special cases when operating near the boundaries of a linked list, add "dummy" node(s), i.e., sentinels, at the beginning and/or end of the linked list.
- To create a sentinel for a linked list `L`, create a node `L.NULL` which has attributes similar to other nodes of the linked list `L`.

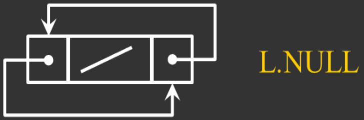
Circular Doubly Linked List - Sentinels

- The attributes of a sentinel `L.NULL`
  - `key = NULL`
  - `next` – points to the first node of the linked list.
  - `prev` – points to the last node of the linked list.



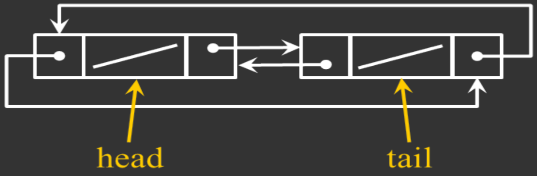
Circular Doubly Linked List - Sentinels

- As "`next`" attribute of the sentinel points to the first node of the linked list, we can eliminate the usual "`head`" attribute.
- In an empty linked list, there is only a sentinel.
- In an empty linked list, `next` and `prev` attributes of a sentinel point to the same node `L.NULL`.



Sentinels - Advantages

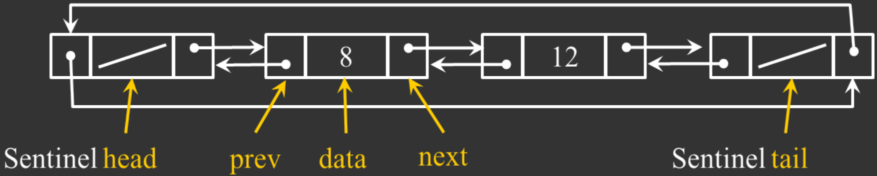
- To keep the logic/code simple, we can add two "`head`" and "`tail`" sentinels.



- Now, we can insert/delete a node without considering boundary conditions.
- A node added or removed from the linked list will always have nodes (either regular or sentinels) on both sides.
- As we do not have to consider boundary conditions, the code becomes simple.

Circular Doubly Linked List - Search

- Goal: Search a node x in the circular doubly linked list.
- Example:



List_Search(L, k) – Sentinels

- Goal: Search a node x in the circular doubly linked list.

List_Search(L, k)

```
x = head.next
while x ≠ tail and x.key ≠ k do
    x = x.next
return x
```

- In the worst case, the running time of List_Search operation is $\theta(n)$ – when the node x is not in the linked list.

Note: If head.next = tail, it is an empty list with only sentinels - head and tail.

Circular Doubly Linked List - Search a Node - Function in C

```c
// Search a node in the linked list having head and tail sentinels.

#include<stdio.h>
#include<stdlib.h>

struct node  // Structure of each node
{
    int data;
    struct node* next;
    struct node* prev;
};
```

Circular Doubly Linked List - Search a Node - Function in C

```c
/* As we have to create only two sentinels,
    we can create it without using malloc.
    Here, sentinels are created in the
    Global variable section of the RAM. We
    use malloc if we have to dynamically
    create n number of sentinels. */

struct node head; // Sentinel - head
struct node tail; // Sentinel - tail

struct node* list_search(int x) // Search a
    node in the linked list
{
    struct node* tmp = head.next;
```

Circular Doubly Linked List - Search a Node - Function in C

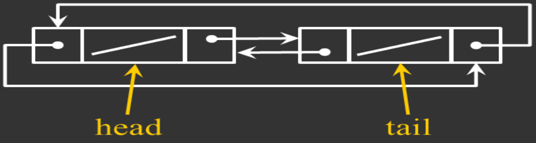
```
    while ((tmp != &tail) && ((*tmp).data !=
        x))
    {
        tmp = (*tmp).next;
    }

    return tmp;
}

int main()
{
    /* Initially create an empty list with
        only two sentinels, head and tail. */
```

Circular Doubly Linked List - Search a Node - Function in C

```
    head.next = &tail; // &tail - head.next stores the address of tail.
    head.prev = &tail;

    tail.next = &head;
    tail.prev = &head;

    ... Other lines of main Function ...

    return 0;
}
```

Circular Doubly Linked List - Insert with Sentinels

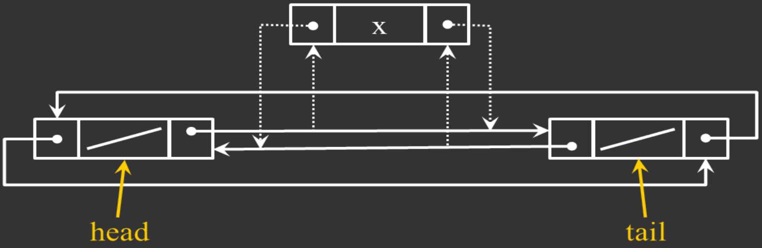
- Goal: To insert a node `x` in the circular doubly linked list (at any position)
- The linked list has two sentinels `head` and `tail`.
- The attributes of sentinels are `key`, `next`, and `prev`.



Note: Delete operation with either one sentinel or two sentinels remains the same.

Circular Doubly Linked List - Insert with Sentinels

- Goal: Insert with head and tail sentinels.




```
List_Insert(L, x)                          // Two sentinels
x.next = x.prev.next
x.prev = x.next.prev
x.prev.next = x
x.next.prev = x
```

- The running time of `List_Insert` operation is $\theta(1)$.

Circular Doubly Linked List - Insert a Node - Function in C

```c
// Insert a node after any node n in the linked list.

#include<stdio.h>
#include<stdlib.h>

struct node  // Structure of each node
{
    int data;
    struct node* next;
    struct node* prev;
};
```

Circular Doubly Linked List - Insert a Node - Function in C

```c
/* As we have to create only two sentinels,
    we can create it without using malloc.
    Here, sentinels are created in the
    Global variable section of the RAM. We
    use malloc if we have to dynamically
    create n number of sentinels. */

struct node head; // Sentinel - head
struct node tail; // Sentinel - tail

void list_insert(struct node* n, int x) //
    Insert a node after the node n
{
    struct node* tmp = (struct node*) malloc
        (sizeof(struct node));
```

```c
    (*tmp).next = (*n).next;
    (*tmp).prev = n;

    (*(*n).next).prev = tmp;
    (*n).next = tmp;

    (*tmp).data = x;
}

int main()
{
    /* Initially create an empty list with
       only two sentinels, head and tail. */
```

Circular Doubly Linked List - Insert a Node - Function in C

```
    head.next = &tail; // &tail - head.next stores the address of tail.
    head.prev = &tail;

    tail.next = &head;
    tail.prev = &head;

    ... Other lines of main Function ...

    return 0;
}
```

Circular Doubly Linked List - Delete

- Goal: Delete a node `x` from a circular doubly linked list.

```
List_Delete(L, x)                    // Sentinels
x.prev.next = x.next
x.next.prev = x.prev
```

- The running time of `List_Delete` operation is $\theta(1)$.

Note: We can delete any node `x` (except sentinels) irrespective of its position in the linked list.

Circular Doubly Linked List - Delete a Node - Function in C

```c
// Delete a node from any position in the
    linked list (all node can be deleted
    except head and tail sentinels.)

#include<stdio.h>
#include<stdlib.h>

struct node  // Structure of each node
{
    int data;
    struct node* next;
    struct node* prev;
};
```

Circular Doubly Linked List - Delete a Node - Function in C

```c
/* As we have to create only two sentinels,
    we can create it without using malloc.
    Here, sentinels are created in the
    Global variable section of the RAM. We
    use malloc if we have to dynamically
    create n number of sentinels. */

struct node head; // Sentinel - head
struct node tail; // Sentinel - tail

void list_remove(int x) // Delete a node
    from the list
{
    if (head.next == &tail)
    {
```

Circular Doubly Linked List - Delete a Node - Function in C

```c
        printf("\nLinked list is already empty.");
    }
    else
    {
        struct node* tmp = list_search(x);

        if (tmp == &tail)
        {
            printf("\nElement is not in the linked list.");
            return;
        }

        (*(*tmp).prev).next = (*tmp).next;
```

Circular Doubly Linked List - Delete a Node - Function in C

```
        (*(*tmp).next).prev = (*tmp).prev;

        free(tmp);
    }
}


int main()
{
    /* Initially create an empty list with only two sentinels, head and tail. */

    head.next = &tail; // &tail - head.next stores the address of tail.
    head.prev = &tail;
```

```
    tail.next = &head;
    tail.prev = &head;

    ... Other lines of main Function ...

    return 0;
}
```

References

References

- Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms, MIT Press.
- Gilberg and Forouzan, Data Structures: A Pseudocode Approach with C, Course Technology Inc.,
- Youtube Channel: mycodeschool, Introduction to Data Structures.

Thank You.