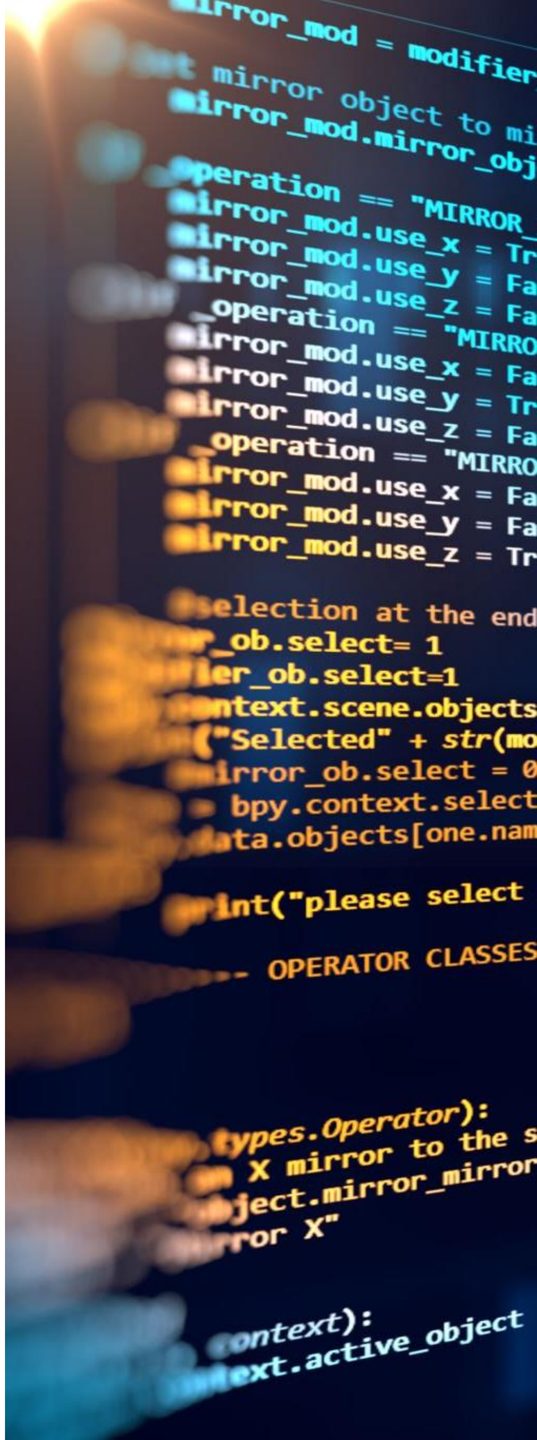
Please use the following QR code to check in and record your attendance.

CS 1037
Fundamentals of Computer Science II

Linked Data Structures

Ahmed Ibrahim

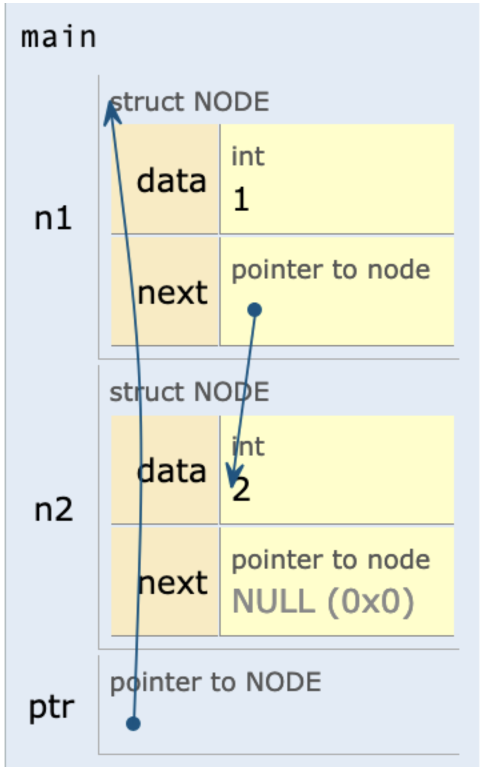


Recall: Self-referential Structures

- Self-referential structures include a reference to data of their <u>own type</u>.
- In the **struct** node definition, the **next** variable acts as a pointer to a **struct** node type.
- The method of self-referential structures is the foundation for building linked data structures such as **linked lists** and **trees**.

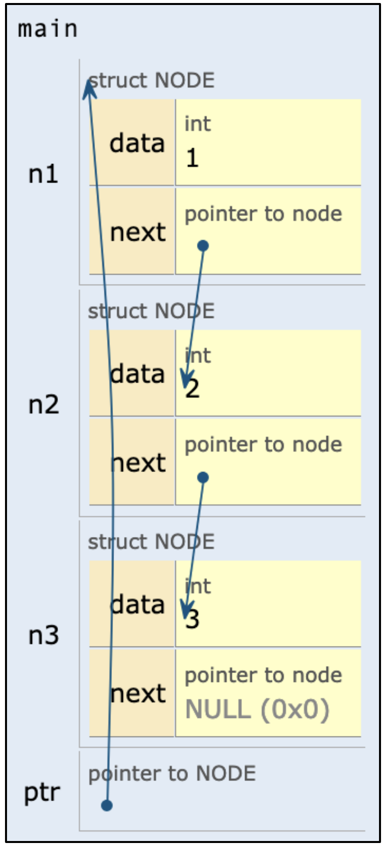
```c
#include <stdio.h>

int main() {

    typedef struct node {
    int data;
    struct node *next;
    } NODE;

     NODE n1, n2, *ptr = &n1;
     n1.data = 1;
     n1.next = &n2;
     n2.data = 2;
     n2.next = NULL;

     // output: 1
     printf("%d\n", ptr->data);

     // output: 2
     printf("%d\n", ptr->next->data);

     return 0;
}
```

Memory:



Recall: The concept of the Linked List

```c
#include <stdio.h>

int main() {

    typedef struct node {
      int data;
      struct node *next;
    } NODE;
    NODE n1, n2, n3, *ptr = &n1;
    n1.data = 1;
    n1.next = &n2;
    n2.data = 2;
    n2.next = &n3;
    n3.data = 3;
    n3.next = NULL;

    return 0;
}
```



Advantages of Linked Lists

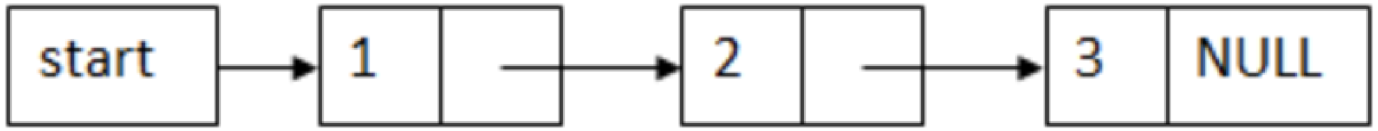
- The items **do not** have to be stored in consecutive memory locations; the successor can be anywhere physically.
  - So, you can insert and delete items without **shifting** data
  - Can **increase the size** of the data structure easily
- Linked lists can grow dynamically (i.e. at run time) – the amount of memory space allocated can grow and shrink as needed.

Real-World Applications

- Music and Video Playlists
- Undo/Redo Functionality in Text Editors
- Browser History
- Memory Management in Operating Systems
- Image Viewing Software
- Dynamic Data Structures in Gaming
- Simple To-do list applications

Linear Linked Data Structures

- A Linear or Singly linked list is a data structure in which each element (node) contains:
  - **Data**: The actual value stored in the node.
  - **Next Pointer**: A reference (or pointer) to the next node in the sequence.
- The last node's next pointer is set to null, indicating the end of the list.



Conceptual Diagram of a Singly-Linked List

Create a Simple To-Do List

- Imagine we want to create a simple program to represent a to-do list using a linked list, where each task is a node.
- Each node will **contain a description** of the task and a **reference to the next task** in the sequence.
- The last task will point to NULL, indicating the end of the list.

What do we need to do?

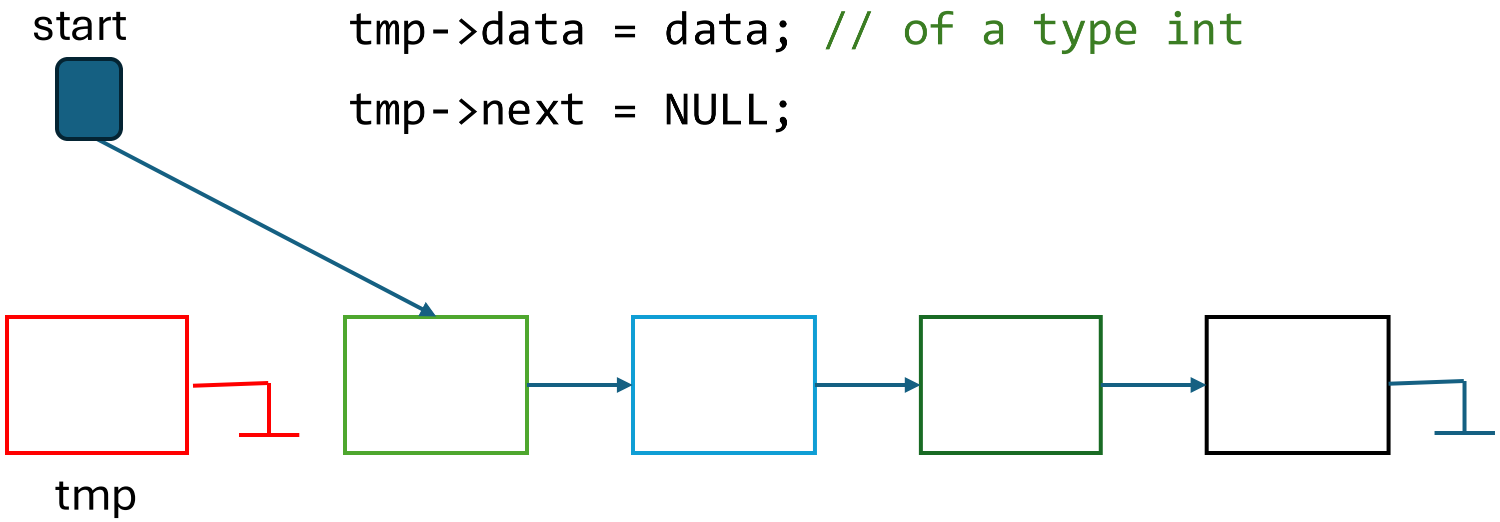
1. Define a Node Structure
2. Create a Function to Make a New Node
3. Add a New Task to the List
4. Print All Tasks in the List
5. Test you program

Linked List Operations

Insert Node to a Linked List

1. Create a new node:

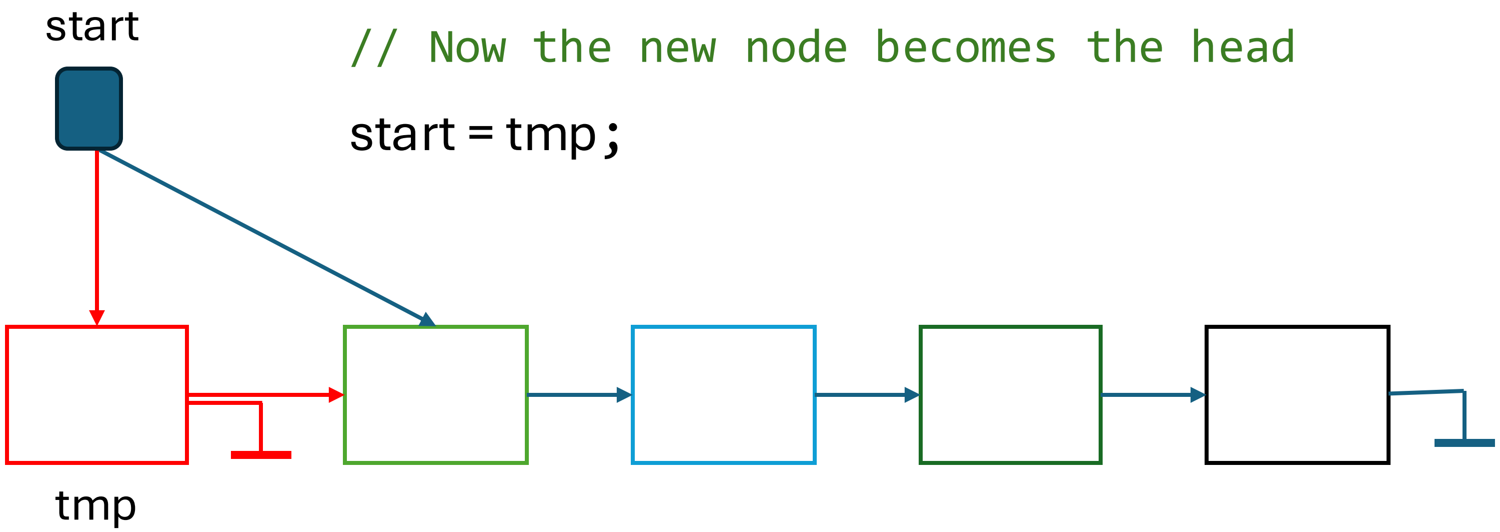
```
// Create a new node with data
      Node *tmp = (Node*)malloc(sizeof(Node));
      tmp->data = data; // of a type int
      tmp->next = NULL;
```



Insert Node to a Linked List (cont.)

2. Insert the new node:

```
// New node points to the current front
tmp->next = start;
// Now the new node becomes the head
start = tmp;
```

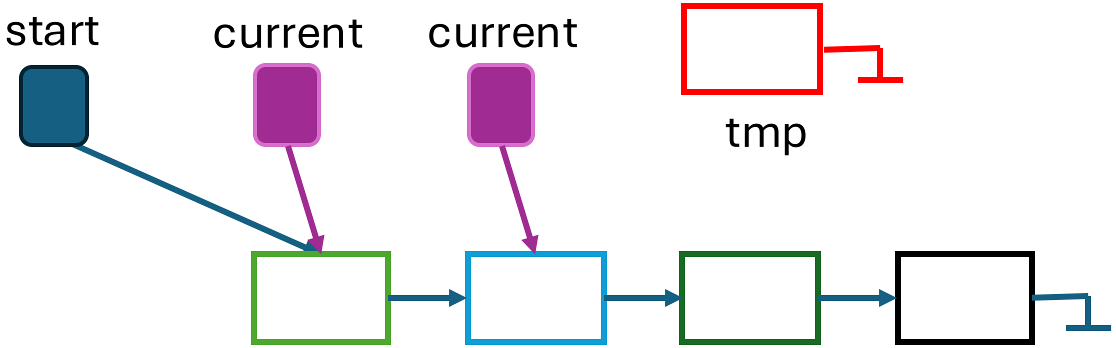


Linked lists can grow and shrink dynamically (i.e., at run time).

Insert Node in the Middle

Find the Position to Insert the Node:

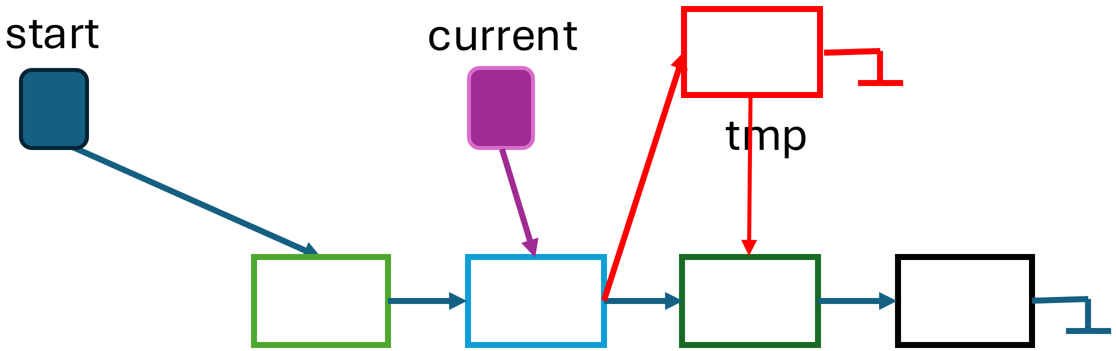
```
// assume that the tmp node has already been created and the position is given (#3)
    // Find the node before the desired position
    Node *current = start;
    for (int i = 0; i < position - 1 && current != NULL; i++) {
    // Traverse to the node before the insertion point
        current = current->next;
    }
```



Insert Node in the Middle

Insert the New Node:

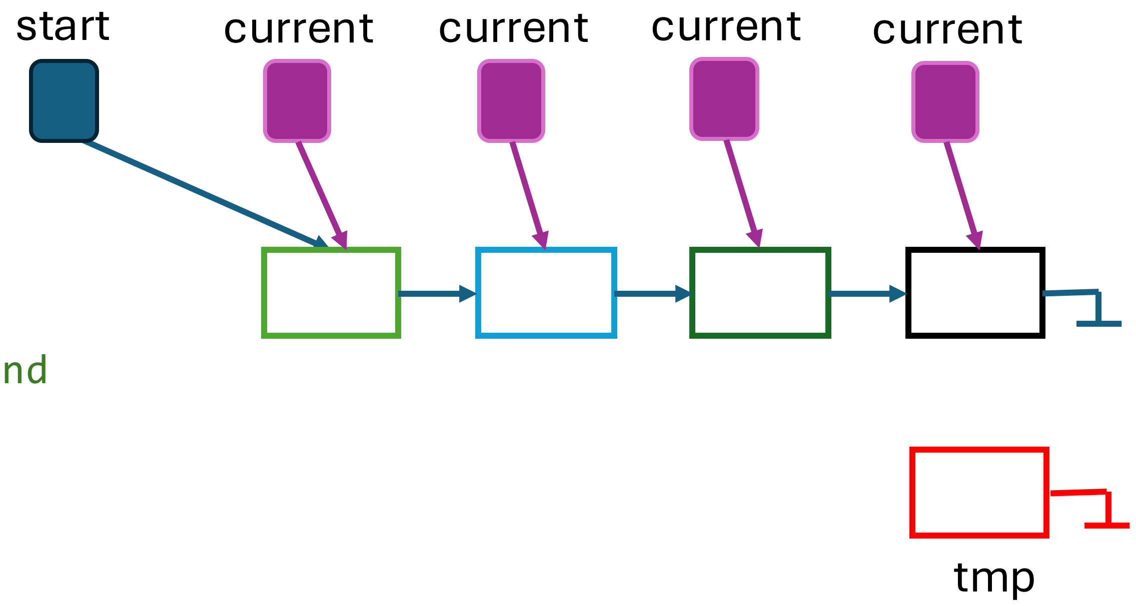
```
// If the current node is not NULL, insert the new node
    if (current != NULL) {
        tmp->next = current->next;
        current->next = tmp;
    }
```



Insert Node at the End

- Find the Last Node in the List:

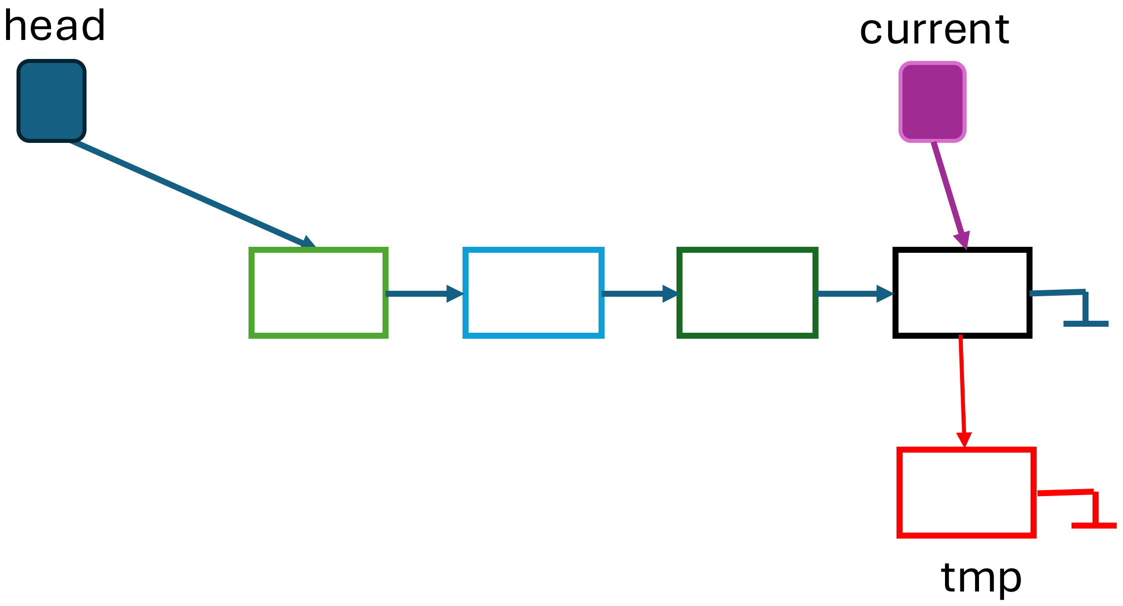
```
…
// Traverse to the last node in the list
Node *current = start;
while (current->next != NULL) {
// Move to the next node until we reach the end
    current = current->next;
}
…
```



Insert Node at the End

- Insert the New Node:

```
// Set the last node's next to the new node
   current->next = tmp;
```



Question!

What happens if we try to insert a node at a position larger than the current length of the list?

A) Insert the node at the end of the list.

B) Display an error message indicating the position is out of bounds.

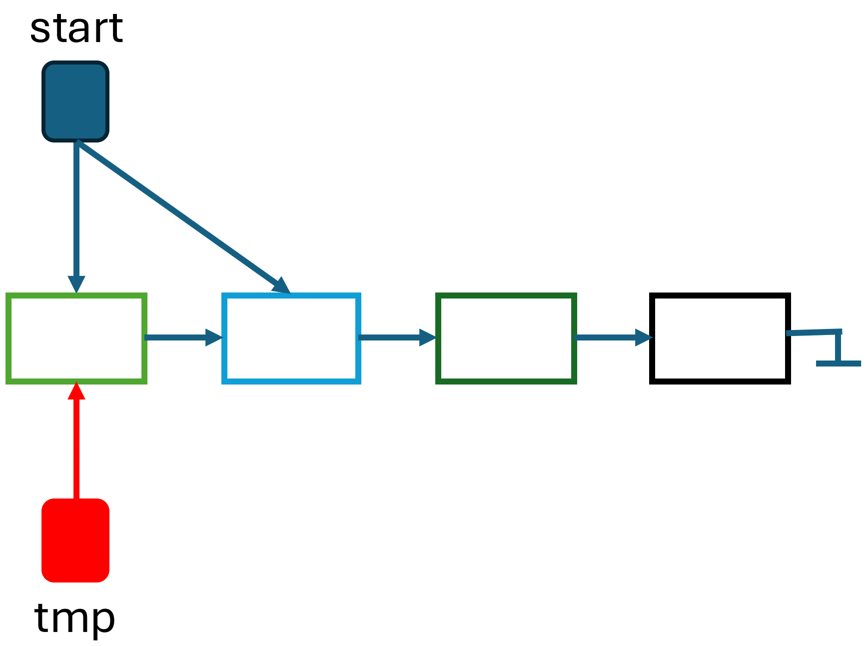
C) Do nothing.

What steps should we take to handle this?

Delete Node from the Front

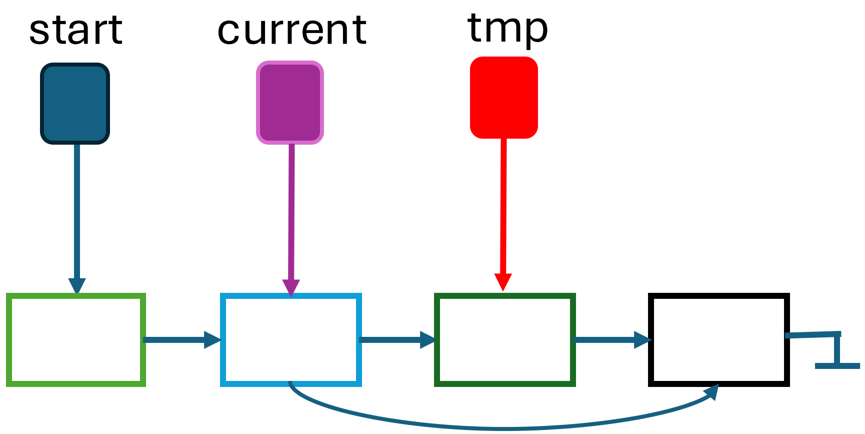
- Delete the First Node (Head):

```
…
    // Move head to the next node
    Node *tmp = start;  // Store the current head
    // Update head to the next node
    start = start->next;
    free(tmp);  // Free the old head node
…
```



Delete Node from the Middle

```
…
// Traverse the list to find the node just
//before the desired position
Node *current = start;
for (int i = 0; i < position - 1 && current != NULL; i++) {
    current = current->next;}
// Delete the node at the desired position
Node *tmp = current->next;
current->next = current->next->next;
// Update the link to skip the deleted node
free(tmp);  // Free the node
…
```

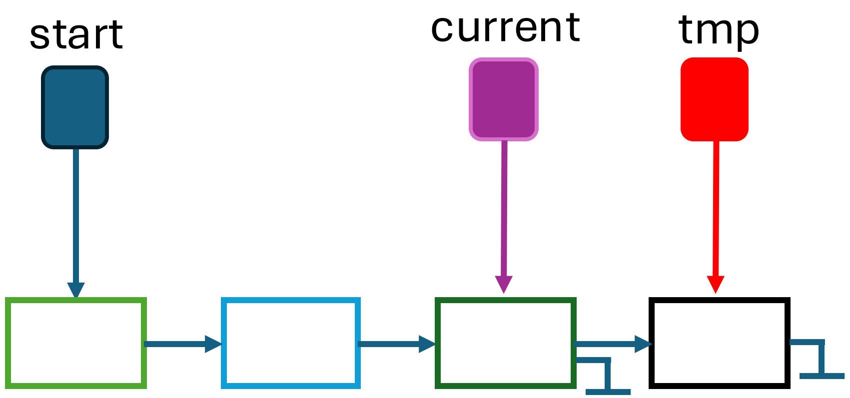


Delete Node from the End

…

```
    // Traverse to the second-to-last node
    Node *current = start;
    while (current->next->next != NULL) {
        current = current->next;
    }
    // Free the last node and set the second-to-last
    //node's next to NULL
    Node *tmp = current->next;
    current->next = NULL;
    free(tmp);
```

…



Question!

You are given the following C code snippet:

```
Node*start = createNode(10);
Start->next = createNode(20);
Start->next->next = createNode(30);
insertAtPosition(&start, 25, 2);
deleteAtPosition(&start, 1);
```

What will the final linked list look like after performing both operations, assuming the initial state and the insertion and deletion operations succeed without errors?

A) 10 -> 25 -> 30 -> NULL

B) 10 -> 30 -> NULL

C) 25 -> 30 -> NULL

D) 10 -> 20 -> 25 -> 30 -> NULL

Memory Management in Linked Lists

- Linked lists in C often use dynamic memory allocation (e.g., malloc) to create new nodes.
- To avoid memory leaks, it is essential to deallocate memory using free() when a node is no longer needed.
- Do not call free() on the same pointer more than once; this can cause undefined behavior.

Linked List Traversal

- This operation involves visiting each node in the list and is essential for printing, searching, or processing each node's data.

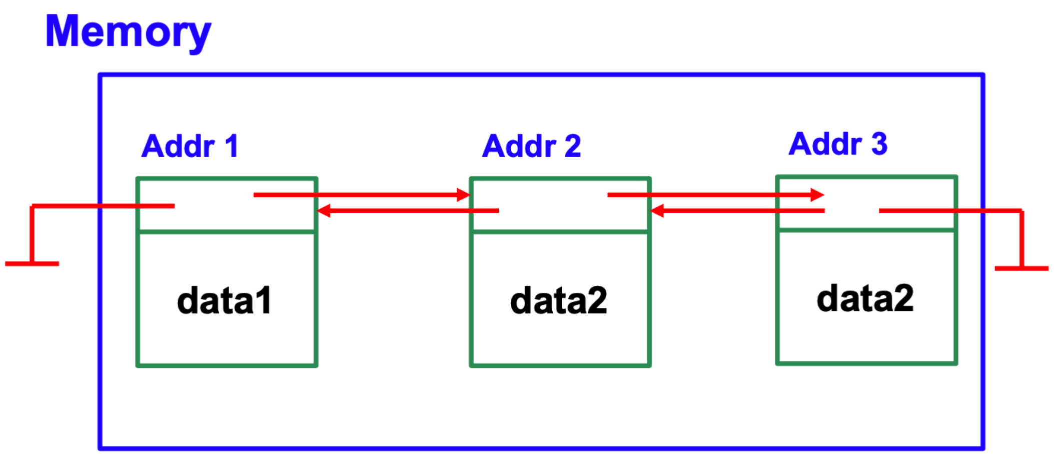
```
void traverse(Node*start) {
    Node*current = start;
    while (current != NULL) {
        printf("%d\n", current->data);
        current = current->next;
    }
}
```

- Searching allows you to find if a particular data element exists in the list.

```
int search(Node*start, char*data) {
    Node *current = start;
    while (current != NULL) {
  // Compare the current node's
  // data with the target data
        if (strcmp(current->data, data) == 0) {
          return 1; // Data found }
        // Move to the next node
        current = current->next;}
      return 0; // Data not found
}
```

Doubly Linked Data Structures

- A doubly linked list is a type of linked list where each node contains three fields:
  - **Data**: The value stored in the node.
  - **Next Pointer**: A reference to the next node in the sequence.
  - **Previous Pointer**: A reference to the previous node in the sequence.
- The first node's previous pointer is NULL, and the last node's next pointer is NULL, signifying the list's boundaries.




Thank you