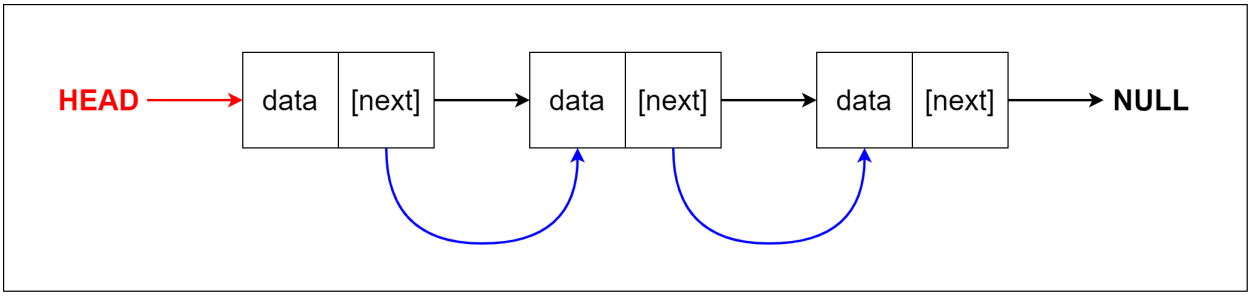
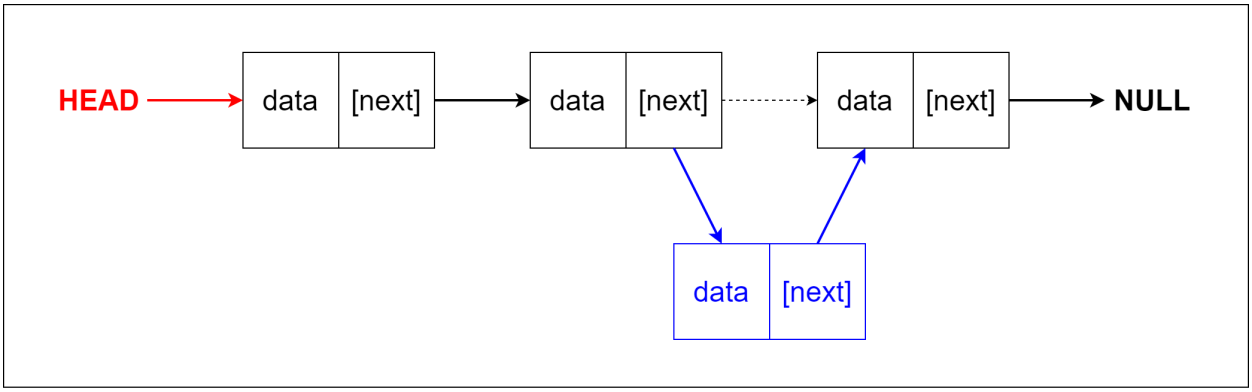
Alright, let's delve into the world of linked list operations. These operations allow us to perform various tasks with linked lists, a fascinating data structure we've been exploring.

In this section, we're going to focus on the fundamental operations that can be performed on linked lists. These operations help us manage and manipulate the data within the linked list efficiently. Let's take a look at them:

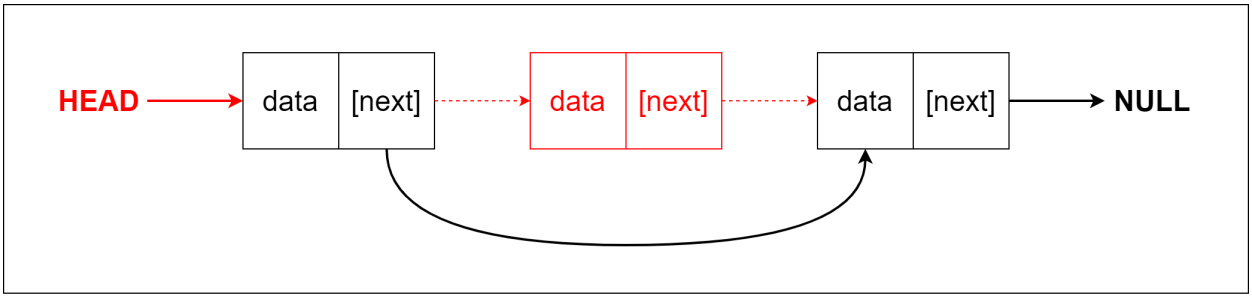
- First, we have **Traversal**, which involves accessing each element of the linked list. This helps us examine and work with the data stored in the list.



- Next, there's **Insertion**, a key operation. It lets us add a new element to the linked list. This is essential for expanding the list and accommodating new data.



- Then, we have **Deletion**, which removes existing elements from the linked list. This operation is crucial for maintaining the content of the list.



- **Search** is another operation. It allows us to find a specific node within the linked list. This is useful when we want to locate and work with particular data.

- Lastly, there's **Sort**. This operation helps us organize the nodes of the linked list in a particular order, making it easier to work with the data.

Before diving into the detailed exploration of these operations, make sure you have a solid understanding of what a linked list is. It's like setting the foundation before building a structure. Let's get started!

Things to Remember about Linked List

let's go over some important points to keep in mind about linked lists.

- First, we have the concept of **head**, which is like a signpost. It points to the very first node in the linked list, helping us start our journey through the list.
- Then, there's the **next** pointer. This is like a path leading us to the next node in the sequence. But here's a trick: When we reach the last node, this pointer points to something special - it points to **NULL**. This tells us that there's no further node to move to, and we've reached the end of our linked list adventure.

Now, let's look at a specific example to get a clearer picture. Imagine we have a linked list with three nodes, labeled as 1, 2, and 3. The sequence goes like this: 1 -> 2 -> 3. Each node has a structure that holds two things: its **data** (which is the number it carries) and its **next** (which is the pointer to the next node).

So, if you think of it like a train, the "data" is the passenger's ticket, and the "next" is the next station where the train is headed.

Remember this structure, as we're going to use it in our examples:

```
struct node {
  int data;
  struct node *next;
};
```

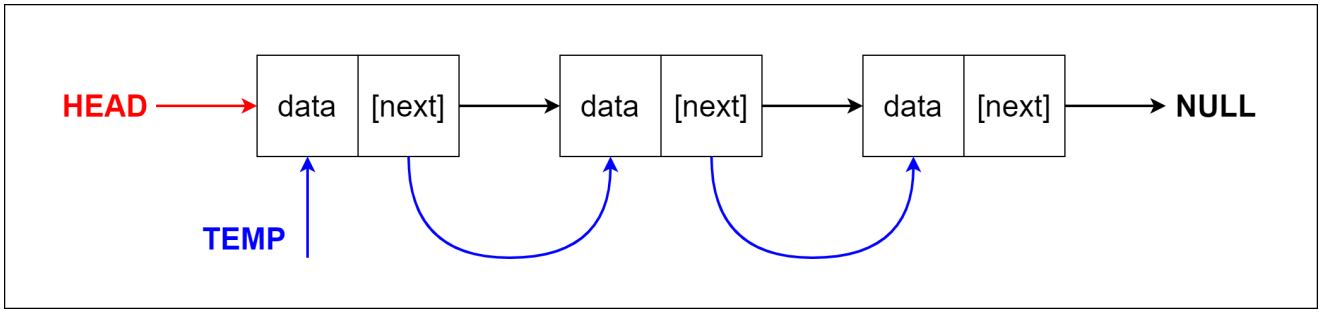
Alright, with these concepts in mind, we're ready to dive deeper into linked list operations. Let's continue our journey!

Traverse a Linked List

let's understand how to traverse or move through a linked list.

Imagine you have a train of nodes, each carrying a number. To see all the numbers, you need to walk through the entire train, right? Traversing a linked list is just like that!

Here's how it works:



1. We start at the first node, which is called **head**. This is like the starting station of our journey.

2. We create a temporary node, let's call it **temp**, and make it point to the same station where the head points. So, our journey starts at the beginning.
3. We're going to take a look at the contents of each station. For example, at the first station, we note down the number (data) it holds, which might be 1. Then we move to the next station by following the **next** pointer.
4. We keep doing this until we reach a station where there's no next station, meaning our **temp** has become **NULL**. This indicates we've reached the end of our journey, or the end of the linked list.

Now, let's look at a piece of code that does this:

```
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL) {
  printf("%d --->", temp->data);
  temp = temp->next;
}
```

Here, **temp** is like a passenger who walks through each station, and when **temp** becomes **NULL**, our journey is complete.

So, if your linked list has the numbers 1, 2, and 3, this code will display:

```
List elements are -
1 ---> 2 ---> 3 --->
```

And that's how we traverse a linked list to see all its contents!

Insert Elements to a Linked List

Let's dive into adding elements to a linked list.

Think of our linked list as a train where each carriage has a number. Now, we'll learn how to add new carriages with numbers.

There are three ways to do this:

1. **Add to the Beginning**: It's like adding a new carriage right at the front of the train. The new number becomes the first one, and the old ones follow.
2. **Add to the Middle**: Imagine you want to insert a new carriage between two existing ones. It's like adding a new number in the middle of the train.
3. **Add to the End**: If you want to extend the train, you just add a new carriage at the very back.

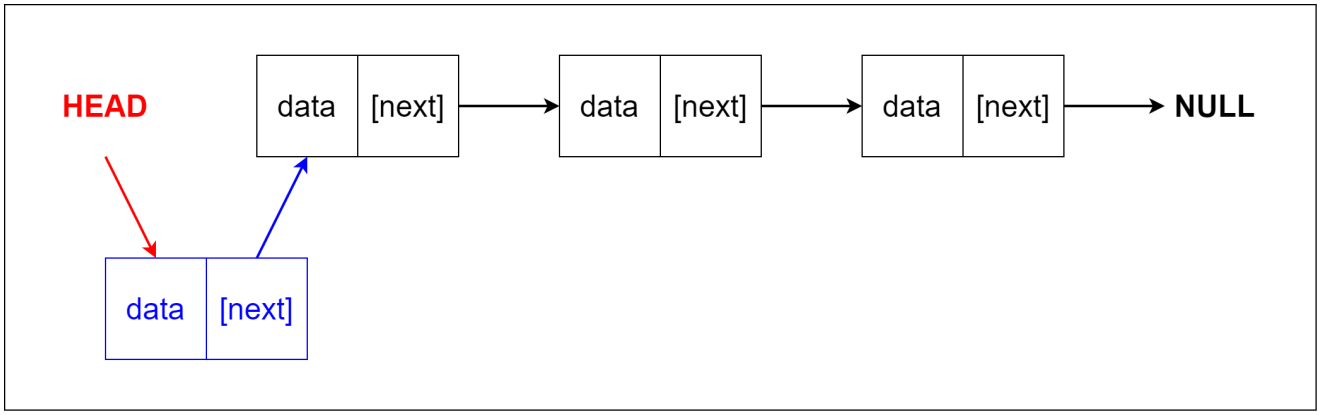
So, depending on where you want your new number, you'll use one of these methods.

Remember, each carriage has a number and a pointer to the next one. When we add a new number, we adjust the pointers to make sure the train is still connected.

That's how we insert elements into a linked list!

1. Insert at the beginning

Let's learn how to add a new element at the beginning of a linked list. Imagine you have a train of numbers, and you want to add a new number right at the front.



Here's how you do it step by step:

1. **Allocate Memory**: You create a new train carriage, or node, for the new number. It's like finding an empty carriage for your new passenger.
2. **Store Data**: You put the new number inside the carriage. This is the value you want to add to the linked list.
3. **Change Pointers**: The magic happens here. You make the new carriage's pointer point to the carriage that was at the front before. In other words, you connect the new carriage to the old front carriage.
4. **Update Head**: Finally, you update the front of the train. The pointer that used to point to the old front carriage now points to the new carriage. So, the new number becomes the new front.

If you put this in code, it would look something like this:

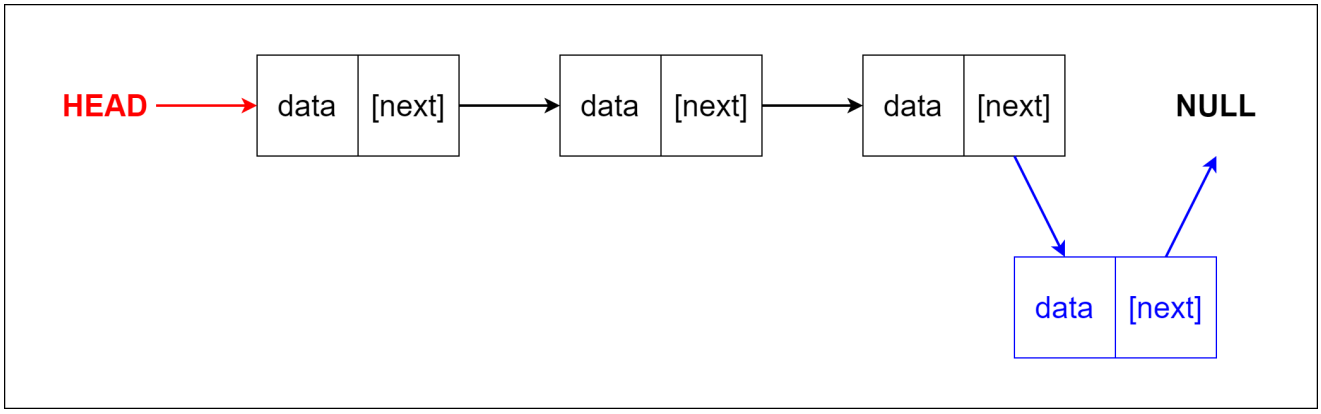
```
struct node *newNode; // Create a new carriage
newNode = malloc(sizeof(struct node)); // Allocate memory
newNode->data = 4; // Put your new number inside
newNode->next = head; // Connect the new carriage to the old front
head = newNode; // Update the front to the new carriage
```

And just like that, you've added a new number to the beginning of your linked list!

2. Insert at the End

Let's move on to adding an element at the end of a linked list. Imagine you have a train of numbers, and you want to add a new number right at the back.

Here's the process broken down:



1. **Allocate Memory**: You prepare a new train carriage, or node, for the new number. It's like getting a new carriage ready at the end of the train.
2. **Store Data**: You put the new number inside the carriage. This is the value you want to add to the linked list.
3. **Traverse to Last**: Now, you need to find the very last carriage in the train. You start from the front and keep moving from one carriage to the next until you reach the last one.
4. **Change Pointers**: Once you're at the last carriage, you change its pointer. The pointer that was pointing to nothing (NULL) now points to the new carriage you just created. This connects the new carriage to the train.

If you put this in code, it would look something like this:

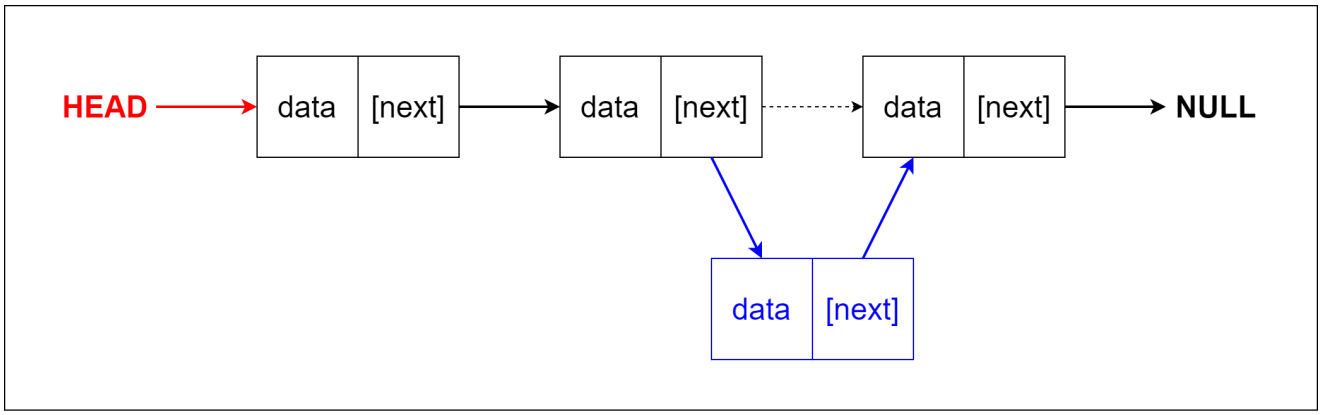
```
struct node *newNode; // Create a new carriage
newNode = malloc(sizeof(struct node)); // Allocate memory
newNode->data = 4; // Put your new number inside
newNode->next = NULL; // Set the new carriage's pointer to NULL

struct node *temp = head; // Start from the front
while(temp->next != NULL){ // Keep moving until you reach the last carriage
  temp = temp->next;
}

temp->next = newNode; // Connect the new carriage to the last one
```

And that's how you add a new number to the end of your linked list!

3. Insert at the Middle

Let's understand how to insert a new element in the middle of a linked list. Imagine you have a sequence of numbers, and you want to slide in a new number right between two existing ones.

Here's the process explained step by step:



1. **Allocate Memory and Store Data**: Just like before, you create a new carriage or node for your number. Then, you put your new number inside the carriage.
2. **Traverse to the Right Spot**: Now, you need to find the carriage just before where you want to add the new one. It's like finding the two carriages you want to slot your new one between. You start from the front and move through the carriages.
3. **Change Pointers**: Once you're at the right carriage, you change its pointer. You make it point to your new carriage, and you make your new carriage point to the carriage that used to come after the one you're inserting in between. This effectively inserts your new carriage in the middle.

If we translate this into code, it would look like this:

```
struct node *newNode; // Create a new carriage
newNode = malloc(sizeof(struct node)); // Allocate memory
newNode->data = 4; // Put your new number inside

struct node *temp = head; // Start from the front

for(int i=2; i < position; i++) { // Find the carriage before the spot you want
  if(temp->next != NULL) {
    temp = temp->next;
  }
}
newNode->next = temp->next; // Make your new carriage point to the next carriage
temp->next = newNode; // Make the previous carriage point to your new carriage
```

That's how you insert a new number right in the middle of your linked list!

Delete from a Linked List

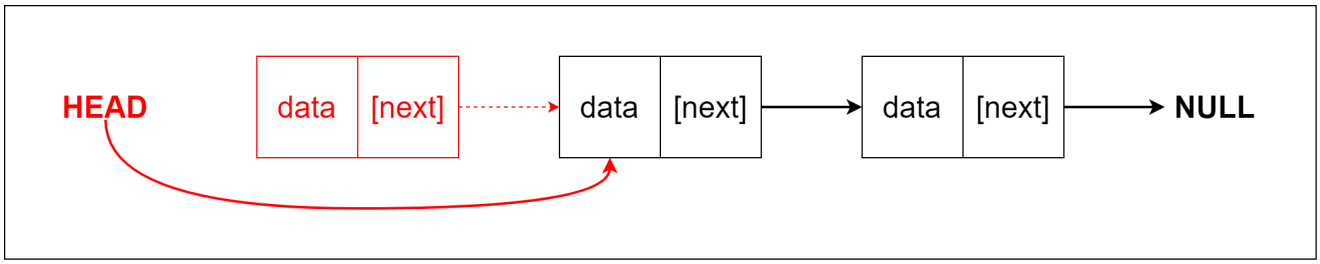
Alright, let's delve into how you can remove elements from a linked list. Imagine you have a list of items, and you want to remove one item. It's like taking out a piece from a puzzle.

Here's the breakdown of the process:

1. **From the Beginning**: Just like how you start solving a puzzle from the edges, you can remove an item from the start of the linked list. You change where the front points to, effectively skipping the first item.
2. **From the End**: Similar to finishing a puzzle and removing the last piece, you can remove an item from the end of the linked list. You change the pointer of the second-to-last item to point to nothing, effectively disconnecting the last piece.
3. **From a Particular Position**: This is like removing a specific piece from the middle of the puzzle. You find the two pieces you want to disconnect, and you make the one before point to the one after, effectively removing the piece you want.

1. Delete from beginning

This is similar to taking the first piece from a line of puzzle pieces.

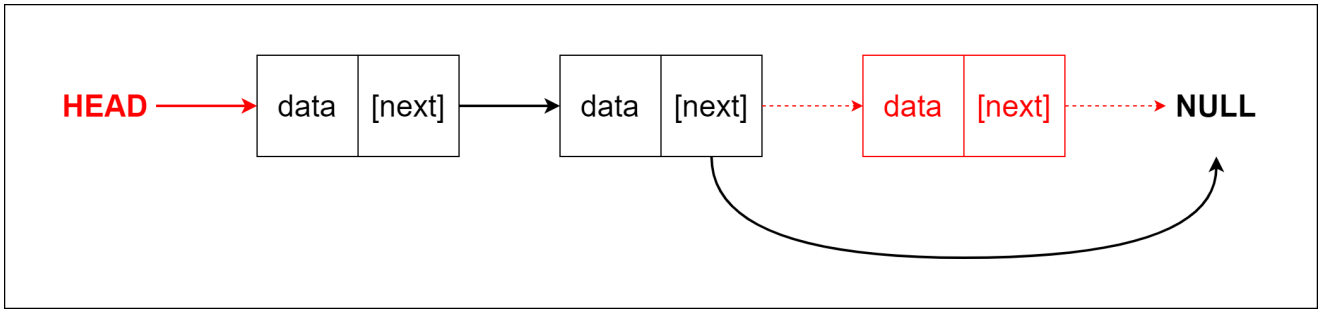


Here's how you can do it using code:

```
// Create a temporary node and point it to the first node
struct node *temp = head;

// Change the head to point to the second node
head = head->next;

// Now you can free the memory of the original first node
free(temp);
```

2. Delete from end

It's like taking the last piece from a line of puzzle pieces.



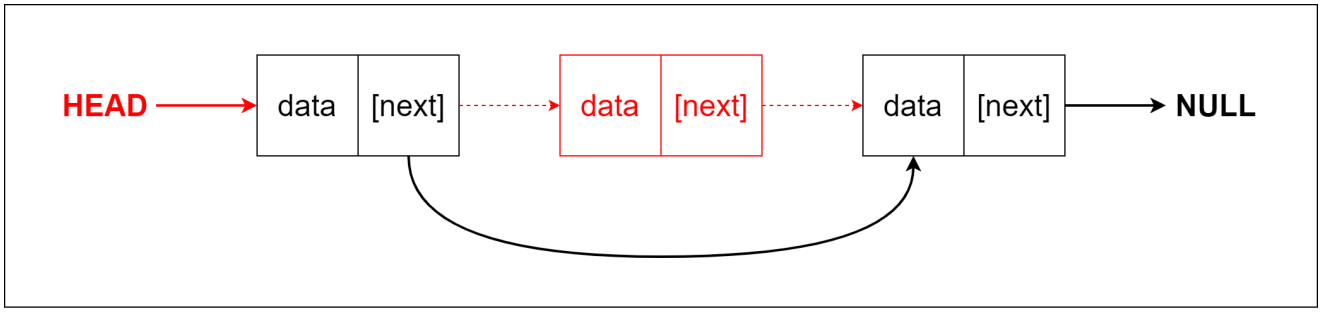
Here's how you can do it using code:

```
// Create a temporary node and start from the head
struct node *temp = head;

// Traverse to the second-to-last element
while(temp->next->next != NULL) {
        temp = temp->next;
}

// Store the last node
struct node *lastNode = temp->next;

// Set the next pointer of the second-to-last node to null
temp->next = NULL;

// Now you can free the memory of the last node
free(lastNode);
```

Think of it as carefully removing the last piece from a puzzle and then making sure the previous piece doesn't point to it anymore.

3. Delete from middle

It's like taking out a specific piece from a row of puzzle pieces.



Here's how you can do it using code:

```
// Create a temporary node and start from the head
struct node *temp = head;

// Traverse to the element just before the one to be deleted
for(int i = 2; i < position; i++) {
        if(temp->next != NULL) {
                temp = temp->next;
        }
}

// Store the node to be deleted
struct node *nodeToDelete = temp->next;

// Adjust the next pointers to exclude the node from the chain
temp->next = temp->next->next;
```

```
// Now you can free the memory of the deleted node
free(nodeToDelete);
```

Think of it like carefully removing a specific puzzle piece from the middle and then making sure the pieces on either side connect seamlessly.

Search an Element on a Linked List

Imagine you're looking for a particular toy in a row of toys. Here's how you'd do it in code:

```
// Search a node
bool searchNode(struct Node** head_ref, int key) {
  // Start from the beginning of the list
  struct Node* current = *head_ref;

  // Keep going until you reach the end of the list
  while (current != NULL) {
    // Check if the data in the current node matches the key you're looking for
    if (current->data == key) return true;

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

  // If you didn't find the key in any node, return false
  return false;
}
```

Think of it like going through a line of toys, checking each one to see if it's the one you're looking for. If you find it, great! If not, you keep moving down the line until you've checked all the toys.

Sort Elements of a Linked List

Let's talk about sorting a linked list, like arranging your toys in order. We'll use a simple sorting method called Bubble Sort, and here's how it works in code:

```
// Sort the linked list
void sortLinkedList(struct Node** head_ref) {
  struct Node *current = *head_ref, *index = NULL;
  int temp;

  // First, check if the list is empty
```

```
  if (head_ref == NULL) {
    return;
  } else {
    // Now, let's start sorting
    while (current != NULL) {
      // 'index' will point to the node next to 'current'
      index = current->next;

      // For each node, compare it with the next nodes
      while (index != NULL) {
        // If the data in the current node is greater than the next node
        if (current->data > index->data) {
          // Swap their data
          temp = current->data;
          current->data = index->data;
          index->data = temp;
        }
        // Move to the next index
        index = index->next;
      }
      // Move to the next current node
      current = current->next;
    }
  }
}
```

Think of it like comparing toys and swapping their positions to get them in the right order. We keep doing this until we've gone through the whole line of toys and they're all arranged from the smallest to the largest.

LinkedList Operations in C++

```
// Linked list operations in C++

#include <stdlib.h>

#include <iostream>
using namespace std;

// Create a node
struct Node {
  int data;
  struct Node* next;
};

void insertAtBeginning(struct Node** head_ref, int new_data) {
  // Allocate memory to a node
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
```

```
  // insert the data
  new_node->data = new_data;
  new_node->next = (*head_ref);

  // Move head to new node
  (*head_ref) = new_node;
}

// Insert a node after a node
void insertAfter(struct Node* prev_node, int new_data) {
  if (prev_node == NULL) {
  cout << "the given previous node cannot be NULL";
  return;
  }

  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  new_node->data = new_data;
  new_node->next = prev_node->next;
  prev_node->next = new_node;
}

// Insert at the end
void insertAtEnd(struct Node** head_ref, int new_data) {
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  struct Node* last = *head_ref; /* used in step 5*/

  new_node->data = new_data;
  new_node->next = NULL;

  if (*head_ref == NULL) {
  *head_ref = new_node;
  return;
  }

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

  last->next = new_node;
  return;
}

// Delete a node
void deleteNode(struct Node** head_ref, int key) {
  struct Node *temp = *head_ref, *prev;

  if (temp != NULL && temp->data == key) {
  *head_ref = temp->next;
  free(temp);
  return;
  }
  // Find the key to be deleted
  while (temp != NULL && temp->data != key) {
  prev = temp;
  temp = temp->next;
```

```
  }

  // If the key is not present
  if (temp == NULL) return;

  // Remove the node
  prev->next = temp->next;

  free(temp);
}

// Search a node
bool searchNode(struct Node** head_ref, int key) {
  struct Node* current = *head_ref;

  while (current != NULL) {
  if (current->data == key) return true;
  current = current->next;
  }
  return false;
}

// Sort the linked list
void sortLinkedList(struct Node** head_ref) {
  struct Node *current = *head_ref, *index = NULL;
  int temp;

  if (head_ref == NULL) {
  return;
  } else {
  while (current != NULL) {
    // index points to the node next to current
    index = current->next;

    while (index != NULL) {
    if (current->data > index->data) {
      temp = current->data;
      current->data = index->data;
      index->data = temp;
    }
    index = index->next;
    }
    current = current->next;
  }
  }
}

// Print the linked list
void printList(struct Node* node) {
  while (node != NULL) {
  cout << node->data << " ";
  node = node->next;
  }
```

```
}

// Driver program
int main() {
  struct Node* head = NULL;

  insertAtEnd(&head, 1);
  insertAtBeginning(&head, 2);
  insertAtBeginning(&head, 3);
  insertAtEnd(&head, 4);
  insertAfter(head->next, 5);

  cout << "Linked list: ";
  printList(head);

  cout << "\nAfter deleting an element: ";
  deleteNode(&head, 3);
  printList(head);

  int item_to_find = 3;
  if (searchNode(&head, item_to_find)) {
  cout << endl << item_to_find << " is found";
  } else {
  cout << endl << item_to_find << " is not found";
  }

  sortLinkedList(&head);
  cout << "\nSorted List: ";
  printList(head);
}
```