Why an Unsorted Linked List?

Say you are making a program to hold a list of names for a party so that you can order everyone's food. You only anticipate having 15 people show up max, so you use an array. Of course, you plan for a few people not to show up, because you know how people are. But…the worst possible outcome happens.

EVERYONE shows up.

…

And not just that. One person brought his girl, one person brought their roommate, and another person doesn't have a car, so he had to bum a ride from his friend, who came inside because he wants a plate.

So now you have 3 more people that you need to add to the list that werent accounted for in the original 15. But because your program uses an array, you only have 15 max slots. If only you could add 3 more slots…

So what went wrong?

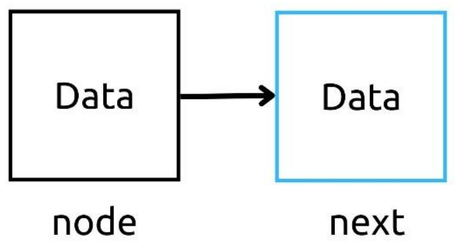
Arrays are great for static programs where the maximum number of elements is static, meaning they are defined at compile time and cannot change during runtime. They are also random access, meaning that there aren't any custom functions needed to access or manipulate an element.

But, in real life situations, it is common to have to change the number of elements during runtime. So a popular Data Structure used to get around this is a linked list.

A linked list is a collection of nodes. Each node points to the location of the next node.

Real simple: Imagine a computer's memory is like a big ass library and you are doing a scavenger hunt. You know that you are looking for a series of books, but you have zero idea where each book actually is. And each book is at a random location in the library, so you can't actually just go down one section and find them all. This seems like a close to impossible task. But, whoever is running your scavenger hunt gives you a piece of paper with the location of your first book: Row 9, Column 7, Book 2. You go to it, and you see that your first book has a sticky note on it that tells you the location of your next book: Row 50, Colum 11, Book 14. You go to the next one, and that one has a sticky note that points to the next book too. Well, you keep going, finding these books with oddly bossy sticky notes until you find a book with a sticky note that says end of search. You then go and get your medal or cookie or whatever from the coordinator.

This my friend is how a linked list works. Each node has a completely random place in memory, but each node points to the location of the next node. In the thought experiment, the piece of paper with the location of the first book is the head pointer. You are a temp pointer, going to the location pointed to by whatever note you are reading. You know you are looking for books, just like a temp pointer points to the addresses of a certain object. You continue to look for books until you find one that says you are finished, just like a linked list function traverses by checking if the current temp points to nullptr.



Each node is a struct that stores data and the location of the next node. They don't have to be a struct; They could be defined using a class, but because structs are public by default, and a node's information should be public so that it can be referenced in the rest of the code, so it's faster to use a define a node using a struct.

Example definition

```
struct node
{
int data;  //Data
node* next; //Creates a pointer to the location of the next node;

node(int Data): data(Data), next(nullptr){};

}
```

So line by line, int data is the payload the node is carrying. In this case, it is an integer called data. Node* next is creating a pointer to the next node object. The last line is a constructor that allows you to initialize the node in one step. You don't need a constructor, but it results in cleaner code because you don't have to write "Node* newNode = new Node(); newNode->data = 5; newNode->next = nullptr;" everytime you want to create a new object.

The next part of a linked list is the LinkedList class. This is used for holding and organizing the member functions. This could also be a struct, however it is most important to learn what parts are private and which are public. The only thing you absolutely need in private is the head definition.

```
class LinkedList

{

node* head = nullptr;

}
```

This creates a pointer called head and initializes it to nullptr. The head pointer serves as the starting point of the linked list, and all other nodes are connected to the list by updating the head pointer or the next pointers of existing nodes.

Linked List Member Functions

Unlike an array where you can insert and delete elements without special functions, a Linked list requires special member functions that traverse through and find, insert or delete elements from a linked list. The most common and necessary functions are:

~Destructor

The destructor is a special function that is called automatically when a linked list object is destroyed. It ensures that all dynamically allocated memory for the list's nodes is properly freed to prevent memory leaks.

Logic
The destructor traverses the list, starting at the head, and deletes each node one by one. A temporary pointer is used to keep track of the current node being deleted while the traversal continues.

Code
```
void ~LinkedList()
{
        node* temp = head;
        while (temp != nullptr)
        {
                node* toDelete = temp;
                temp = temp->next;
                delete toDelete;
        }
}
```

Line by Line

node* temp = head;

- This creates a pointer called temp, which starts at the head of the list. If the list is empty, head points to nullptr, and the loop will not execute.

while (temp != nullptr)

- This loop checks if temp is not equal to nullptr. If true, it continues to delete nodes.

node* toDelete = temp;

- This creates a temporary pointer toDelete that points to the current node that temp is pointing to. This allows the program to delete the current node while keeping track of the next node.

temp = temp->next;

- This moves temp to the next node in the list.

delete toDelete;

- This deletes the current node that toDelete points to, freeing the memory associated with it.

void insertAtHead(int data)

This function inserts a new node at the beginning of the linked list.

Logic

A new node is created with the given data. The new node's next pointer is set to point to the current head of the list. Finally, the head pointer is updated to point to the new node.

Code

```
void insertAtHead(int data)
{
        node* newNode = new node(data);
        newNode->next = head;
        head = newNode;
}
```

Line by Line

node* newNode = new node(data);

- This creates a new node with the given data and dynamically allocates memory for it.

```
newNode->next = head;
```

- This sets the new node's next pointer to point to the current head of the list.

```
head = newNode;
```

- This updates the head pointer to point to the new node, making it the first node in the list.

void insertAtEnd(int data)

This function inserts a new node at the end of the linked list.

Logic

A new node is created with the given data. If the list is empty, the new node becomes the head. Otherwise, the function traverses the list to find the last node and sets its next pointer to the new node.

Code

```
void insertAtEnd(int data)
{
        node* newNode = new node(data);
        if (head == nullptr)
        {
                head = newNode;
                return;
        }
        node* temp = head;
        while (temp->next != nullptr)
        {
                temp = temp->next;
        }
        temp->next = newNode;
}
```

Line by Line

node* newNode = new node(data);

- This creates a new node with the given data and dynamically allocates memory for it.

if (head == nullptr)

- This checks if the list is empty. If the head points to nullptr, the list is empty.

head = newNode;

- If the list is empty, this sets the head pointer to the new node, making it the first and only node in the list.

node* temp = head;

- If the list is not empty, this creates a pointer called temp that starts at the head of the list.

while (temp->next != nullptr)

- This loop traverses the list until temp points to the last node, where the next pointer is nullptr.

temp->next = newNode;

- This sets the next pointer of the last node to the new node, adding it to the end of the list.

void deleteNode(int target)

This function deletes the first node in the linked list with the specified target value.

Logic

The function first checks if the list is empty. If the target is found in the head node, the head is updated to point to the next node, and the head node is deleted. Otherwise, the function traverses the list to find the target node and updates the previous node's next pointer to skip the target node. Finally, the target node is deleted.

Code

```
void deleteNode(int target)
{
        if (head == nullptr)
        {
                return;
        }
        if (head->data == target)
        {
                node* temp = head;
                head = head->next;
                delete temp;
                return;
        }
        node* temp = head;
        while (temp->next != nullptr && temp->next->data != target)
        {
                temp = temp->next;
        }
        if (temp->next == nullptr)
        {
                return;
        }
                node* toDelete = temp->next;
                temp->next = temp->next->next;
                delete toDelete;
}
```

Line by Line

if (head == nullptr)

- This checks if the list is empty. If head points to nullptr, the function exits immediately since there is nothing to delete.

if (head->data == target)

- This checks if the target value is in the head node.

node* temp = head;

- If the target is in the head node, this creates a pointer temp that points to the head node.

head = head->next;

- This updates the head pointer to point to the second node, effectively removing the first node from the list.

delete temp;

- This deletes the original head node, freeing the memory associated with it.

node* temp = head;

- This creates a pointer temp to traverse the list starting from the head.

while (temp->next != nullptr && temp->next->data != target)

- This loop traverses the list to find the node just before the target node. It stops when temp->next points to the target node or when temp->next becomes nullptr.

if (temp->next == nullptr)

- This checks if the target value was not found in the list. If temp->next is nullptr, the function exits without deleting anything.

node* toDelete = temp->next;

- This creates a pointer toDelete that points to the node containing the target value.

```
temp->next = temp->next->next;
```

- This updates the next pointer of the node just before the target node to skip over the target node.

```
delete toDelete;
```

- This deletes the target node, freeing the memory associated with it.

void display()

display is a void function that prints out everything in the list.

Logic

The function checks to see if the list is empty by checking if temp which is points to the location of head is nullptr. If so, if prints out that the list is empty. If temp doesn't point to nullptr, it enters a while loop which prints out the current node's data and traverses to the next node.

Code

```
void display()
{
        node* temp = head;
        if (temp == nullptr)
        {
                cout << “List is empty” <<endl;
                return;
        }
        else
        {
                while(temp != nullptr)
                {
                        cout << temp->data << endl;
                        temp = temp->next;
                }
        }
}
```

Line by Line

node* temp = head;

- This creates a pointer called temp which points at a node object. Currently it is pointing to head, which itself is pointing to nullptr if the list is empty, or the location of a node if the list if full.

if (temp == nullptr)

- This checks to see if the list is empty. If temp which points to head points to nullptr, this means that the list has to be empty because the head has no memory addres.

while (temp != nullptr)

- While loop that checks if temp doesn't equal nullptr. If not, it runs everything inside the loop

temp = temp->next;

- Points temp to the next node;

bool isIn(int data)

isIn is a boolean function that returns true if the given data is in the list. If it is not in the list, the function returns false.

Logic

The function checks to see if the list contains the target value by starting at head and assigning its location to temp. If temp points to nullptr, meaning the list is empty, the loop will not execute, and the function will return false. If temp doesn't point to nullptr, it enters a while loop that checks each node's data to see if it matches the target. If a match is found, the function immediately returns true. Otherwise, it moves temp to the next node by following the next pointer and continues the loop. When temp eventually points to nullptr, indicating the end of the list, and no match has been found, the function returns false.

Code

```
bool isIn(int target)
{
  node* temp = head;
  while (temp != nullptr)
        {
               if (temp->data == target)
               {
                      return true;
               }
               temp = temp->next;
        }

  return false;
```

}

Line by Line

node* temp = head;

- This creates a pointer called temp which points at a node object. Currently it is pointing to head, which itself is pointing to nullptr if the list is empty, or the location of a node if the list if full.

if (temp == nullptr)

- Checks for if temp points to the location of nullptr. If so, list is empty and returns false.

while (temp->data != target)

- This checks for if temp's pointed to node's data is target. If not, the contents of the loop runs.

int size()

size is an int function that returns the number of nodes in a linked list.

Logic

The function begins by creating a pointer called temp, which is set to point to the location of head. If the list is empty, meaning head points to nullptr, temp will also point to nullptr. It also initializes a counter variable called count and sets it to 0. The function then enters a while loop that runs as long as temp does not point to nullptr. Inside the loop, the function increments count by 1 for each node encountered and moves temp to the next node in the list by setting it to temp->next. Once temp points to nullptr, indicating the end of the list, the loop terminates, and the function returns the value of count, which now holds the total number of nodes in the list. If the list is empty, the loop is skipped entirely, and count remains 0, which is returned immediately.

Code

```
int size()
{
	node* temp = head;
	int count = 0;

	while (temp != nullptr)
	{
		count ++;
		temp = temp->next;
	}
		return count;
}
```

Line by Line

node* temp = head;

- This creates a pointer named temp that points to a node object. It is initially set to point to head, which could either point to nullptr if the list is empty or to the first node if the list has elements.

int count = 0;

- This initializes a counter variable named count with a value of zero. The counter will be incremented for each node encountered during traversal, and it will eventually hold the total number of nodes in the list.

```
while (temp != nullptr)
```

- This begins a loop that continues as long as temp does not point to nullptr. If temp points to nullptr, it indicates the end of the list, and the loop stops.

```
    count ++;
```

   - This increments the counter count by one for every node that temp points to during the traversal.

```
    temp = temp->next;
```

   - This moves temp to the next node in the list. The next pointer of the current node is used to find the memory address of the next node. If the current node is the last node, temp is set to nullptr, which ends the loop.

```
return count;
```

- This returns the total value of the count variable, which represents the number of nodes in the list. If the list is empty and head points to nullptr, the loop never runs, and count remains zero.

Linked List Implementation

So now that we know what the individual parts of a linked list are, lets put them together and show what it might look like in an actual c++ file.

```
#include <iostream>

using namespace std;
```

```
struct node

{

int data;

node*next;

node(int Data); data(Data), next(nullptr){};
};


class linkedList

{

private:

node* head = nullptr;


public:

// Destructor

void ~LinkedList()

{

   node* temp = head;

   while (temp != nullptr)

   {

      node* toDelete = temp;

      temp = temp->next;

      delete toDelete;

   }

}


// insertAtHead
```

```
void insertAtHead(int data)
{
  node* newNode = new node(data);
  newNode->next = head;
  head = newNode;
}

// insertAtEnd
void insertAtEnd(int data)
{
  node* newNode = new node(data);
  if (head == nullptr)
  {
    head = newNode;
    return;
  }
  node* temp = head;
  while (temp->next != nullptr)
  {
    temp = temp->next;
  }
  temp->next = newNode;
}

// deleteNode
void deleteNode(int target)
```

```
{
  if (head == nullptr)
  {
    return;
  }
  if (head->data == target)
  {
    node* temp = head;
    head = head->next;
    delete temp;
    return;
  }
  node* temp = head;
  while (temp->next != nullptr && temp->next->data != target)
  {
    temp = temp->next;
  }
  if (temp->next == nullptr)
  {
    return;
  }
  node* toDelete = temp->next;
  temp->next = temp->next->next;
  delete toDelete;
}
```

```
// display
void display()
{
   node* temp = head;
   if (temp == nullptr)
   {
      cout << “List is empty” << endl;
      return;
   }
   else
   {
      while(temp != nullptr)
      {
         cout << temp->data << endl;
         temp = temp->next;
      }
   }
}

// isIn
bool isIn(int target)
{
   node* temp = head;
   while (temp != nullptr)
   {
      if (temp->data == target)
```

```
        {
            return true;
        }
        temp = temp->next;
    }
    return false;
}

// size
int size()
{
    node* temp = head;
    int count = 0;

    while (temp != nullptr)
    {
        count++;
        temp = temp->next;
    }
    return count;
}
};

int main() {
    linkedList list;
    int choice;
```

```
bool running = true;

while (running) {
  cout << "\nMenu:\n";
  cout << "1. Insert at head\n";
  cout << "2. Insert at end\n";
  cout << "3. Delete a node\n";
  cout << "4. Display list\n";
  cout << "5. Check if a value is in the list\n";
  cout << "6. Get the size of the list\n";
  cout << "7. Exit\n";
  cout << "Enter your choice: ";
  cin >> choice;

  switch (choice) {
    case 1: {
      int num;
      cout << "Enter the number to insert at head: ";
      cin >> num;
      list.insertAtHead(num);
      break;
    }
    case 2: {
      int num;
      cout << "Enter the number to insert at end: ";
      cin >> num;
      list.insertAtEnd(num);
      break;
    }
    case 3: {
      int num;
      cout << "Enter the number to delete: ";
      cin >> num;
      list.deleteNode(num);
      break;
    }
    case 4:
      list.display();
      break;
    case 5: {
      int num;
      cout << "Enter the number to check: ";
      cin >> num;
      if (list.isIn(num)) {
          cout << "The number " << num << " is in the list." << endl;
      } else {
          cout << "The number " << num << " is not in the list." << endl;
      }
      break;
    }
    case 6:
      cout << "The size of the list is: " << list.size() << endl;
```

```
            break;
          case 7:
            running = false;
            cout << "Exiting program. Goodbye!" << endl;
            break;
          default:
            cout << "Invalid choice. Please try again." << endl;
        }
    }

    return 0;
}
```

Summary

This program implements an interactive menu-driven application to manage a linked list. The user is presented with a menu of options to perform various operations on the linked list, such as inserting a node at the head or the end, deleting a node, displaying the list, checking if a value exists in the list, and determining the size of the list. The program runs in a continuous loop, allowing the user to execute multiple operations until they choose to exit. Each menu option is handled using a switch statement that calls the corresponding function from the linked list class. The running variable controls the loop, ensuring the program continues to operate until the user selects the exit option.

Logic

1. **Initialization**:
   - A linkedList object is created to manage the linked list operations.
   - The program initializes a choice variable to store the user's menu selection and a running boolean to control the while loop.

2. **Menu Display**:
    - Inside the while loop, the program displays a menu with seven options:
        - Insert a node at the head.
        - Insert a node at the end.
        - Delete a node by value.
        - Display all nodes in the list.
        - Check if a value exists in the list.
        - Get the size of the list.
        - Exit the program.
3. **User Input**:
    - The program prompts the user to enter their choice and stores the value in the choice variable.
4. **Switch Statement**:
    - Based on the user's input, the switch statement directs the program to the appropriate case:
        - **Case 1 (Insert at Head)**: Prompts the user for a value and inserts a new node containing that value at the head of the list.
        - **Case 2 (Insert at End)**: Prompts the user for a value and appends a new node containing that value to the end of the list.
        - **Case 3 (Delete a Node)**: Prompts the user for a value and removes the first node in the list containing that value, if it exists.
        - **Case 4 (Display List)**: Calls the display function to print all nodes in the list.
        - **Case 5 (Check if a Value is in the List)**: Prompts the user for a value and checks if it exists in the list, printing the result.
        - **Case 6 (Get the Size of the List)**: Calls the size function and prints the total number of nodes in the list.
        - **Case 7 (Exit)**: Sets running to false to terminate the loop and prints a goodbye message.

        - **Default**: Handles invalid inputs by notifying the user and returning to the menu.

5. **Program Termination**:
    - The program continues looping until the user selects the exit option, at which point it ends gracefully. The destructor ensures that all dynamically allocated memory is properly freed when the linked list object is destroyed.

*Addendum

Smartass kid: But Christian, can't you just use a vector to dynamically manipulate a list of elements?

Well sure but try using a vector on a linked list test.

But seriously, in a project, a vector is also a good option when dynamically manipulating a list of elements.

Vectors are good for random read access and insertion and deletion in the back (takes amortized constant time), but bad for insertions and deletions in the front or any other position (linear time, as items have to be moved). Vectors are usually laid out contiguously in memory, so traversing one is efficient because the CPU memory cache gets used effectively.

Linked lists on the other hand are good for inserting and deleting items in the front or back (constant time), but not particularly good for much else: For example deleting an item at an arbitrary index in the middle of the list takes linear time because you must first find the node. On the other hand, once you have found a particular node you can delete it or insert a new item after it in constant time, something you cannot do with a vector. Linked lists are also very simple to implement, which makes them a popular data structure.