Linked Lists

What is a linked list?

- A linked list in C is a linear data structure that consists of a sequence of nodes.
- Each node contains a value and a pointer to the next node in the sequence.
- Linked lists are dynamic in size, meaning that they can grow or shrink as needed.
- They are also relatively easy to implement and manipulate.

Adv & disadv Of Linked Lists

Advantages

- **Dynamic in size**: Linked lists can grow or shrink in size as needed, without having to preallocate memory. This makes them efficient for storing data that is of unknown size or that changes frequently.
- **Easy to implement**: Linked lists are relatively easy to implement in C, using only a few basic data structures and operations.
- **Flexible**: Linked lists can be used to implement a variety of different data structures, such as stacks, queues, and hash tables.

Disadvantages

- **Slower access**: Accessing an element in a linked list requires traversing the list from the beginning until the desired element is found. This can be slower than accessing an element in an array, which can be accessed directly by its index.
- **More memory usage**: Linked lists require more memory than arrays, because each node in a linked list must store a pointer to the next node. This can be a significant disadvantage for applications that need to store large amounts of data.
- **Susceptible to memory fragmentation**: Linked lists can be susceptible to memory fragmentation, which can occur when memory is allocated and deallocated frequently. This can lead to performance problems and memory leaks.

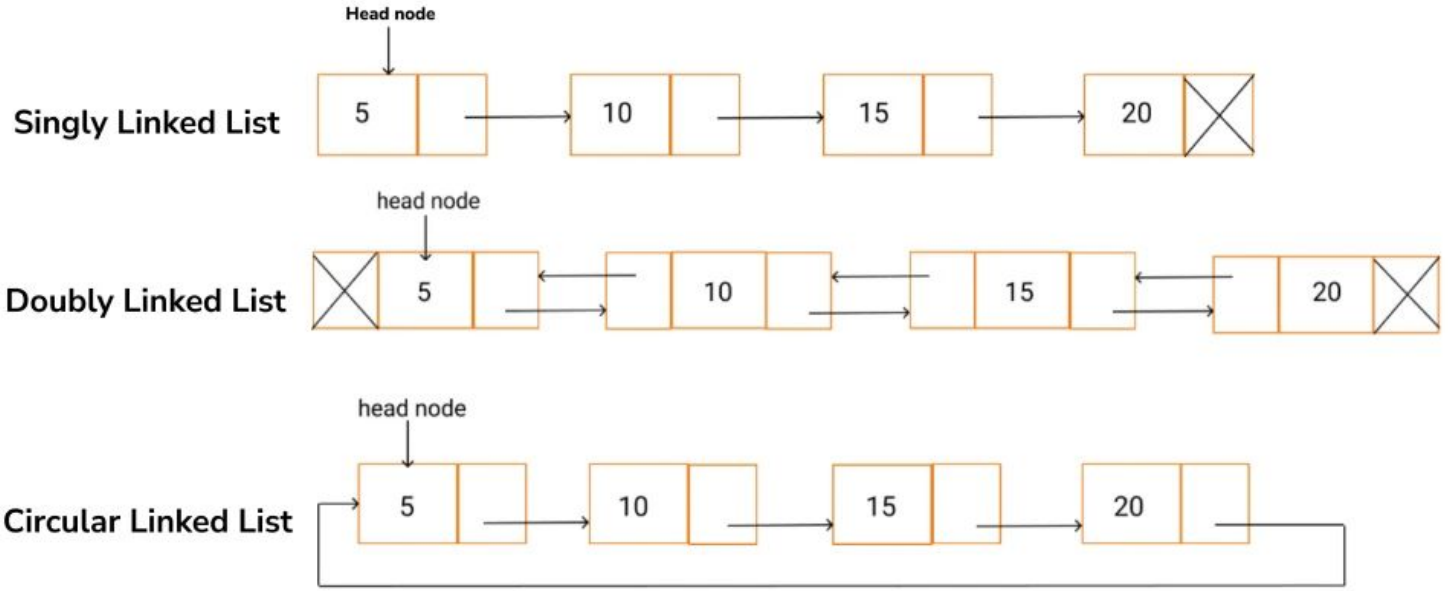
Tips for using Linked Lists in C

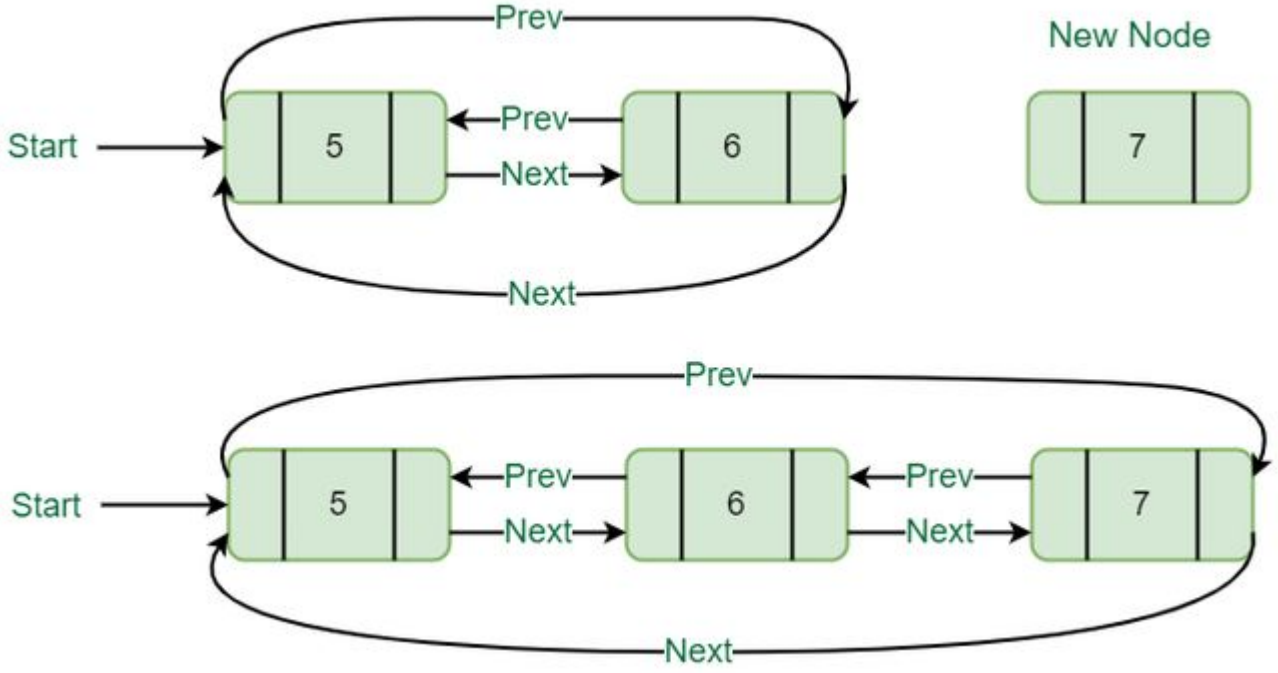
- Use linked lists when you need to store data that is of unknown size or that changes frequently.
- Use linked lists to implement data structures such as *stacks*, *queues*, and *hash tables*.
- Be aware of the performance and memory implications of using linked lists.
- Use a memory management library to help prevent memory fragmentation like jemalloc.

Types of Linked Lists in C

1. **Singly linked list**: A singly linked list is the simplest type of linked list. Each node in a singly linked list contains a value and a pointer to the next node in the list. The last node in the list points to NULL.
2. **Doubly linked list**: A doubly linked list is similar to a singly linked list, but each node in a doubly linked list also contains a pointer to the previous node in the list. This allows for faster traversal of the list in both directions.
3. **Circular linked list**: A circular linked list is a linked list where the last node points back to the first node in the list. This creates a loop, which can be useful for implementing certain data structures, such as queues.
4. **Doubly circular linked list**: A doubly circular linked list is a combination of a doubly linked list and a circular linked list. Each node in a doubly circular linked list contains pointers to both the next and previous nodes in the list, and the last node points back to the first node in the list.

Types of Linked Lists




Prev
New Node
Start
5
Prev
6
Next
7
Next
Prev
Start
5
Prev
6
Prev
7
Next
Next
Next

Linked Lists Operations

1. Insertion

- This involves adding a new node to the list. You can add a node at the beginning (head), at the end (tail), or after a given node.
- When you want to insert a new node into a linked list, you have three options:
    - Insert at the beginning of the list.
    - Insert at a specific position in the list.
    - Insert at the end of the list.

demo

- The `insert()` function takes an integer value as input and creates a new node with that data.
- If the head of the linked list is `NULL`, the new node is set as the head of the linked list.
- Otherwise, the function traverses the linked list until it reaches the end of the list. The new node is then added to the end of the linked list.
- The `print()` function traverses the linked list and prints the data in each node.
- In the `main()` function, we create three new nodes and insert them into the linked list.
- We then call the `print()` function to print the contents of the linked list.

2. Deletion

- This involves removing a node from the list. You can remove a node based on the value in the data field or based on its position in the list.

demo

- The `delete()` function takes an integer value as input and deletes the node in the linked list that contains that data.
- If the node to be deleted is the head of the linked list, the head of the linked list is set to the next node in the list.
- Otherwise, the function traverses the linked list until it finds the node to be deleted. The `previous` pointer is used to keep track of the node before the node to be deleted.
- Once the node to be deleted is found, the `next` pointer of the previous node is set to the next pointer of the node to be deleted.
- The node to be deleted is then freed from memory.
- The `print()` function traverses the linked list and prints the data in each node.
- In the `main()` function, we create three new nodes and insert them into the linked list.
- We then call the `print()` function to print the contents of the linked list.
- Next, we call the `delete()` function to delete the node with the data 20.
- We then call the `print()` function again to print the contents of the linked list.

3. Traversal

- This involves moving through the list, typically starting from the head and following the links to each node until you reach the end (or a specific node).

demo

- The `traverse()` function traverses the linked list and prints the data in each node.
- The function starts at the head of the linked list and follows the next pointers until it reaches the end of the list.
- At each node, the function prints the data in the node.
- The function returns when it reaches the end of the list.
- In the `main()` function, we create three new nodes and insert them into the linked list.
- We then call the `traverse()` function to print the contents of the linked list.

4. Searching

- This involves finding a node with a given value in its data field.

demo

- The `search()` function takes an integer value as input and searches for the node in the linked list that contains that data.
- The function starts at the head of the linked list and follows the next pointers until it finds a node with the matching data, or it reaches the end of the list.
- If the function finds a node with the matching data, it returns a pointer to that node.
- Otherwise, the function returns `NULL`.
- In the `main()` function, we create three new nodes and insert them into the linked list.
- We then call the `search()` function to search for the node containing the data 20.
- If the node is found, we print a message saying that the node was found.
- Otherwise, we print a message saying that the node was not found.

5. Updating

- This involves changing the data of a given node.

demo

- The `update()` function takes two integer values as input: the old data and the new data. The function searches for the node in the linked list that contains the old data. If the node is found, the function updates the node's data to the new data.
- The function starts at the head of the linked list and follows the next pointers until it finds a node with the matching data, or it reaches the end of the list.
- If the function finds a node with the matching data, it updates the node's data to the new data and returns.
- Otherwise, the function returns without updating any nodes.
- In the `main()` function, we create three new nodes and insert them into the linked list.
- We then call the `print()` function to print the contents of the linked list.
- Next, we call the `update()` function to update the node containing the data 20 to 40.
- We then call the `print()` function again to print the contents of the linked list.

Uses of Linked Lists

- **Stacks**: Stacks can be implemented using singly linked lists.
- **Queues**: Queues can be implemented using singly linked lists or doubly linked lists.
- **Hash tables**: Hash tables can be implemented using linked lists to store the elements in each bucket.
- **Graphs**: Graphs can be represented using linked lists to store the edges of the graph.

Point to Remember

Linked lists are dynamic and can grow and shrink during the execution of a program. They do not need a contiguous memory space like arrays, but they do use more memory due to the storage of pointers.

Resources

1. Repo Link https://github.com/betascribbles/LinkedLists
2. TutorialsPoint: This website has a comprehensive tutorial on linked lists in C, covering topics such as creating and manipulating linked lists, traversal, searching, and sorting.
3. GeeksforGeeks: This website has a number of articles on linked lists in C, covering topics such as different types of linked lists, operations on linked lists, and common linked list problems and solutions.
4. Stack Overflow: This website is a great resource for finding answers to specific questions about linked lists in C.
5. The C Programming Language: This book by Dennis Ritchie and Brian Kernighan is considered to be the definitive book on the C programming language. It covers the basics of linked lists in C, as well as more advanced topics.
6. Data Structures and Algorithms in C: This book by Mark Allen Weiss provides a comprehensive introduction to data structures and algorithms, including linked lists.

See you at the next session!