Linked Lists

Linked Lists

A linked list is a linear data structure where each element is a separate object, known as a node Each node contains some data and points to the next node in the structure, forming a sequence. The nodes may be at different memory locations, unlike arrays where all the elements are stored continuously.

Linked List

Linked List

Lasindi [Publidomain]

The linked list can be used to store data similar to arrays but with several more advantages.

Think of it as a friend circle. If person A knows person B and person B knows person C, person C could be reached from person A through this linked connection. Each person can be seen as a Node who knows the link to the next person.

Advantages over Arrays

The 2 advantages of a linked list over an array are:

Not fixed in size: A linked list is not fixed in size. The memory locations to store the nodes are allocated dynamically when each node is created. There is no wastage of memory for unused locations. In comparison, an array can only be defined once of a specific size, and then further cannot be extended or shrunk down accordingly.

Efficient Insertion and Deletion: A quick manipulation of the links between the nodes allows for a constant time taken for insertion and deletion. In contrast, one has to move over all the memory locations while dealing with arrays so that they are in order.

Disadvantages over Arrays

There are some disadvantages of using linked lists when compared to arrays though.

Only sequential access: As the data is linked together through nodes, any node can only be accessed by the node linking to it, hence it is not possible to randomly access any node. One has to go through the links searching for the element required.

Memory Usage of each node: The nodes that hold the data

need extra memory to hold the pointer to the next node. Each element hence takes slightly more memory than an array.

Creating a Linked List

For the linked list to be created, we need to define a node first depending on the type of linked list we want to create. Each type of list has specific properties and its own merits regarding the list operations.

Types of Linked Lists

A linked list is designed depending on its use. The 3 most common types of a linked list are:

Singly Linked List

Doubly Linked List

Circular Linked List

Singly Linked List

This is the most common type of linked list, where each node has one pointer to the next node in the sequence. This means that the list can only be traversed from the beginning to the end in one direction. To access the last element, it is always required to traverse the whole list to the end.

Singly Linked List

Singly Linked List

Lasindi [Publidomain]

The last node always points to NULL in a singly-linked list. This specifies that the list has ended with no more nodes to traverse to. Every time a loop traverses through the array, it checks for this NULL condition to know if the end of the linked list is there.

Implementation

Defining the Node

The Node contains 2 parts, one that is the data itself and the other which references the next node in the sequence. For simplicity, we will consider a Node where the data is a single integer. The data is not just limited to one value, one can define any number of pieces of information to be stored in each node.

In C, the node is defined as a structure This type of a structure is called a self-referential structure where one

member of the structure points to the structure of its kind.

```
struct Node
{
int data;
struct Node *next;
} *head = NULL;
```

A new Node is created first with the desired variable name. We will call this newNode for now.

```
struct Node *newNode;
```

The data stored in this Node can be accessed by using the arrow character (->) to the data member of the structure.

```
newNode->data
```

Similarly, the link to the next Node in the list can be accessed by using the arrow character to the *next member of the structure.

```
newNode->next
```

About the head pointer node

The head node is used to point to the first node in a linked list. This is used to keep track of the list beginning and helps during the traversing operations.

Operations in a Linked List

The few basic operations in a linked list including adding, deleting and modifying.

Creating an empty list

An empty list has to be created before performing any other operations. The head variable is created and assigned NULL. This will be used as the starting point to our linked list.

Adding to the end of the list

New data can be added to the end of the linked list by creating a new Node with the data to be used, traversing to the end of the list and then appending this data to the end.

```
void insertAtEnd(int value)
{
```

```
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode->next = NULL;
if(head == NULL)
head = newNode;
else
{
struct Node *temp = head;
while(temp->next != NULL)
temp = temp->next;
temp->next = newNode;
    }
printf("\nNode inserted successfully at end\n");
}
```

Adding to the beginning of the list

New data can be added to the beginning of the linked list by creating a new Node with the data to be used, replacing the head pointer to the new node and modifying the connections.

```
void insertAtBeginning(int value)
```

```
{
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
if(head == NULL)
{
newNode->next = NULL;
head = newNode;
    }
else
{
newNode->next = head;
head = newNode;
    }
printf("\nNode inserted successfully at beginning\n");
}
```

Adding to a specific position of the list

New data can be added at any position in the list by traversing to that position using a loop, creating a new Node and then manipulating the links to insert it at that position.

```
void insertPosition(int value, int pos)
{
int i = 0;
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
if(head == NULL)
{
newNode->next = NULL;
head = newNode;
    }
else {
struct Node *temp = head;
for (i = 0; i < pos - 1; i++) {
temp = temp-> next;
        }
newNode->next = temp->next;
temp->next = newNode;
    }
printf("\nNode inserted successfully\n");
```

}

Deletion from the end of the list

New data can be added to the end of the linked list by creating a new Node with the data to be used, traversing to the end of the list and then appending this data to the end.

```
void removeEnd()
{
if(head == NULL)
{
printf("\nList is Empty\n");
    }
else
{
struct Node *temp1 = head,*temp2;
if(head->next == NULL)
head = NULL;
else
{
while(temp1->next != NULL)
```

```
{
temp2 = temp1;
temp1 = temp1->next;
        }
temp2->next = NULL;
      }
free(temp1);
printf("\nNode deleted at the end\n\n");
    }
}
```

Deletion from the beginning of the list

New data can be added to the beginning of the linked list by creating a new Node with the data to be used, replacing the head pointer to the new node and replacing the connections.

```
void removeBeginning()
{
if(head == NULL)
printf("\n\nList is Empty");
else
{
```

```
struct Node *temp = head;
if(head->next == NULL)
{
head = NULL;
free(temp);
        }
else
{
head = temp->next;
free(temp);
printf("\nNode deleted at the beginning\n\n");
        }
    }
}
```

Deletion from a specific position of the list

New data can be deleted at any position in the list by traversing to that position using a loop, deleting the required Node and then manipulating the links to make the list continuous.

```
void removePosition(int pos)
```

```
{
int i,flag = 1;
if (head==NULL)
printf("List is empty");
else {
struct Node *temp1 = head, *temp2;
if (pos == 1) {
head = temp1->next;
free(temp1);
printf("\nNode deleted\n\n");
        }
else {
for (i = 0; i < pos - 1; i++)
{
if (temp1 -> next != NULL) {
temp2 = temp1;
temp1 = temp1 -> next;
              }
else {
flag = 0;
```

```
break;
                }
            }
if (flag) {
temp2 -> next = temp1 -> next;
free(temp1);
printf("\nNode deleted\n\n");
            }
else {
printf("Position exceeds number of elements in linked list.
Please try again");
            }
        }
    }
}
```

Searching in a linked list

An element can be searched in a list by going through each element and checking it against the required element. As the element Nodes can only be accessed linearly, only a linear search can be performed in the case.

This is one of the disadvantages of a linked list regarding random access of elements.

```
void search(int key)
{
while (head != NULL)
{
if (head->data == key)
{
printf("The key is found in the list\n");
return;
        }
head = head->next;
    }
printf("The Key is not found in the list\n");
}
```

Doubly Linked List

A doubly linked list has 2 pointers, one pointing to the next node and one to the previous node. This allows for moving in any direction while traversing the list, which may be useful in

certain situations.

Doubly Linked List

Doubly Linked List

Lasindi [Publidomain]

The implementation and details are here: Link to Doubly linked list

Circular Linked List

A circular linked list is like a regular one except for the last element of the list pointing to the first. This has the advantage of allowing to go back back to the first element while traversing a list without starting over.

Circular Linked List

Circular Linked List

Lasindi [Publidomain]

Complexity of operations:

It is not possible to have a constant access time in linked list operations. The data required may be at the other end of the list and the worst case may be to traverse the whole list to get

it.

Access

The elements of a linked list are only accessible in a sequential manner. Hence to access any element, we have to iterate through each node one by one until we reach the required element. The time complexity is hence O(n).

Insertion

Insertion in a linked list involves only manipulating the pointers of the previous node and the new node, provided we know the location where the node is to be inserted. Thus, the insertion of an element is O(1).

Deletion

Similar to deletion, deletion in a linked list involves only manipulating the pointers of the previous node and freeing the new node, provided we know the location where the node is to be deleted. Thus, the deletion of an element is O(1).

Applications of a Linked List