UNIT-III

Linked List

* Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
* A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
* The last node of the list contains pointer to the null.



Uses of Linked List

* The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
* list size is limited to the memory size and doesn't need to be declared in advance.
* Empty node can not be present in the linked list.
* We can store values of primitive types or objects in the singly linked list.

## Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

|  |  |  |
| --- | --- | --- |
| **Data Structure** | **Time Complexity** | **Space Compleity** |
|  | **Average** | **Worst** | **Worst** |
|  | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |  |
|

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Insertion at beginning](https://www.javatpoint.com/insertion-in-singly-linked-list-at-beginning) | It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list. |
| 2 | [Insertion at end of the list](https://www.javatpoint.com/insertion-in-singly-linked-list-at-end) | It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario. |
| 3 | [Insertion after specified node](https://www.javatpoint.com/insertion-in-singly-linked-list-after-specified-node) | It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. . |

Singly Linked List | θ(n) | θ(n) | θ(1) | θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |

## Complexity

## Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

### Node Creation

1. struct node
2. {
3. **int** data;
4. struct node \*next;
5. };
6. struct node \*head, \*ptr;
7. ptr = (struct node \*)malloc(sizeof(struct node \*));

### Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following

categories.

### Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Deletion at beginning](https://www.javatpoint.com/deletion-in-singly-linked-list-at-beginning) | It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers. |
| 2 | [Deletion at the end of the list](https://www.javatpoint.com/deletion-in-singly-linked-list-at-end) | It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios. |
| 3 | [Deletion after specified node](https://www.javatpoint.com/deletion-in-singly-linked-list-after-specified-node) | It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list. |
| 4 | [Traversing](https://www.javatpoint.com/traversing-in-singly-linked-list) | In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list. |
| 5 | [Searching](https://www.javatpoint.com/searching-in-singly-linked-list) | In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. . |

Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



In C, structure of a node in doubly linked list can be given as :

1. struct node
2. {
3. struct node \*prev;
4. **int** data;
5. struct node \*next;
6. }

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Operations on doubly linked list

**Node Creation**

1. struct node
2. {
3. struct node \*prev;
4. **int** data;
5. struct node \*next;
6. };
7. struct node \*head;

All the remaining operations regarding doubly linked list are described in the following table.

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Insertion at beginning](https://www.javatpoint.com/insertion-in-doubly-linked-list-at-beginning) | Adding the node into the linked list at beginning. |
| 2 | [Insertion at end](https://www.javatpoint.com/insertion-in-doubly-linked-list-at-the-end) | Adding the node into the linked list to the end. |
| 3 | [Insertion after specified node](https://www.javatpoint.com/insertion-in-doubly-linked-list-after-specified-node) | Adding the node into the linked list after the specified node. |
| 4 | [Deletion at beginning](https://www.javatpoint.com/deletion-in-doubly-linked-list-at-beginning) | Removing the node from beginning of the list |
| 5 | [Deletion at the end](https://www.javatpoint.com/deletion-in-doubly-linked-list-at-the-end) | Removing the node from end of the list. |
| 6 | [Deletion of the node having given data](https://www.javatpoint.com/deletion-in-doubly-linked-list-after-the-specified-node) | Removing the node which is present just after the node containing the given data. |
| 7 | [Searching](https://www.javatpoint.com/searching-in-doubly-linked-list) | Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null. |
| 8 | [Traversing](https://www.javatpoint.com/traversing-in-doubly-linked-list) | Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc. |

# Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

## Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

## Operations on Circular Singly linked list:

### Insertion

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Insertion at beginning](https://www.javatpoint.com/insertion-in-circular-singly-list-at-beginning) | Adding a node into circular singly linked list at the beginning. |
| 2 | [Insertion at the end](https://www.javatpoint.com/insertion-in-circular-singly-linked-list-at-end) | Adding a node into circular singly linked list at the end. |

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Deletion at beginning](https://www.javatpoint.com/deletion-in-circular-singly-linked-list-at-beginning) | Removing the node from circular singly linked list at the beginning. |
| 2 | [Deletion at the end](https://www.javatpoint.com/deletion-in-circular-singly-linked-list-at-end) | Removing the node from circular singly linked list at the end. |
| 3 | [Searching](https://www.javatpoint.com/searching-in-circular-singly-linked-list) | Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null. |
| 4 | [Traversing](https://www.javatpoint.com/traversing-in-circular-singly-linked-list) | Visiting each element of the list at least once in order to perform some specific operation. |

### Deletion & Traversing

# Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

## Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

**Time Complexity : o(1)**



### C implementation :

* 1. **void** push ()
	2. {
	3. **int** val;
	4. struct node \*ptr =(struct node\*)malloc(sizeof(struct node));
	5. **if**(ptr == NULL)
	6. {
	7. printf("not able to push the element");
	8. }
	9. **else**
	10. {
	11. printf("Enter the value");
	12. scanf("%d",&val);
	13. **if**(head==NULL)
	14. {
	15. ptr->val = val;
	16. ptr -> next = NULL;
	17. head=ptr;
	18. }
	19. **else**
	20. {
	21. ptr->val = val;
	22. ptr->next = head;
	23. head=ptr;
	24.
	25. }
	26. printf("Item pushed");
	27.
	28. }
	29. }

## Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

* 1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
	2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

**Time Complexity : o(n)**

### C implementation

* 1. **void** pop()
	2. {
	3. **int** item;
	4. struct node \*ptr;
	5. **if** (head == NULL)
	6. {
	7. printf("Underflow");
	8. }
	9. **else**
	10. {
	11. item = head->val;
	12. ptr = head;
	13. head = head->next;
	14. free(ptr);
	15. printf("Item popped");
	16.
	17. }
	18. }

## Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

* 1. Copy the head pointer into a temporary pointer.
	2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

**Time Complexity : o(n)**

### C Implementation

* 1. **void** display()
	2. {
	3. **int** i;
	4. struct node \*ptr;
	5. ptr=head;
	6. **if**(ptr == NULL)
	7. {
	8. printf("Stack is empty\n");
	9. }
	10. **else**
	11. {
	12. printf("Printing Stack elements \n");
	13. **while**(ptr!=NULL)
	14. {
	15. printf("%d\n",ptr->val);
	16. ptr = ptr->next;
	17. }
	18. }
	19. }

### Menu Driven program in C implementing all the stack operations using linked list :

* 1. #include <stdio.h>
	2. #include <stdlib.h>
	3. **void** push();
	4. **void** pop();
	5. **void** display();
	6. struct node
	7. {
	8. **int** val;
	9. struct node \*next;
	10. };
	11. struct node \*head;
	12.
	13. **void** main ()
	14. {
	15. **int** choice=0;
	16. printf("\n\*\*\*\*\*\*\*\*\*Stack operations using linked list\*\*\*\*\*\*\*\*\*\n");
	17. printf("\n----------------------------------------------\n");
	18. **while**(choice != 4)
	19. {
	20. printf("\n\nChose one from the below options...\n");
	21. printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
	22. printf("\n Enter your choice \n");
	23. scanf("%d",&choice);
	24. **switch**(choice)
	25. {
	26. **case** 1:
	27. {
	28. push();
	29. **break**;
	30. }
	31. **case** 2:
	32. {
	33. pop();
	34. **break**;
	35. }
	36. **case** 3:
	37. {
	38. display();
	39. **break**;
	40. }
	41. **case** 4:
	42. {
	43. printf("Exiting....");
	44. **break**;
	45. }
	46. **default**:
	47. {
	48. printf("Please Enter valid choice ");
	49. }
	50. };
	51. }
	52. }
	53. **void** push ()
	54. {
	55. **int** val;
	56. struct node \*ptr = (struct node\*)malloc(sizeof(struct node));
	57. **if**(ptr == NULL)
	58. {
	59. printf("not able to push the element");
	60. }
	61. **else**
	62. {
	63. printf("Enter the value");
	64. scanf("%d",&val);
	65. **if**(head==NULL)
	66. {
	67. ptr->val = val;
	68. ptr -> next = NULL;
	69. head=ptr;
	70. }
	71. **else**
	72. {
	73. ptr->val = val;
	74. ptr->next = head;
	75. head=ptr;
	76.
	77. }
	78. printf("Item pushed");
	79.
	80. }
	81. }
	82.
	83. **void** pop()
	84. {
	85. **int** item;
	86. struct node \*ptr;
	87. **if** (head == NULL)
	88. {
	89. printf("Underflow");
	90. }
	91. **else**
	92. {
	93. item = head->val;
	94. ptr = head;
	95. head = head->next;
	96. free(ptr);
	97. printf("Item popped");
	98.
	99. }
	100. }
	101. **void** display()
	102. {
	103. **int** i;
	104. struct node \*ptr;
	105. ptr=head;
	106. **if**(ptr == NULL)
	107. {
	108. printf("Stack is empty\n");
	109. }
	110. **else**
	111. {
	112. printf("Printing Stack elements \n");
	113. **while**(ptr!=NULL)
	114. {
	115.

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

1. Ptr = (struct node \*) malloc (sizeof(struct node));

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

1. ptr -> data = item;
2. **if**(front == NULL)
3. {
4. front = ptr;
5. rear = ptr;
6. front -> next = NULL;
7. rear -> next = NULL;
8. }

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

1. rear -> next = ptr;
2. rear = ptr;
3. rear->next = NULL;

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

* **Step 1:** Allocate the space for the new node PTR
* **Step 2:** SET PTR -> DATA = VAL
* **Step3:** IF FRONT = NULL
SET FRONT = REAR = PTR
SET FRONT -> NEXT = REAR -> NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR
SET REAR -> NEXT = NULL
[END OF IF]
* **Step 4:** END

C Function

1. **void** insert(struct node \*ptr, **int** item; )
2. {
3.
4.
5. ptr = (struct node \*) malloc (sizeof(struct node));
6. **if**(ptr == NULL)
7. {
8. printf("\nOVERFLOW\n");
9. **return**;
10. }
11. **else**
12. {
13. ptr -> data = item;
14. **if**(front == NULL)
15. {
16. front = ptr;
17. rear = ptr;
18. front -> next = NULL;
19. rear -> next = NULL;
20. }
21. **else**
22. {
23. rear -> next = ptr;
24. rear = ptr;
25. rear->next = NULL;
26. }
27. }
28. }

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

1. ptr = front;
2. front = front -> next;
3. free(ptr);

The algorithm and C function is given as follows.

Algorithm

* **Step 1:** IF FRONT = NULL
Write " Underflow "
Go to Step 5
[END OF IF]
* **Step 2:** SET PTR = FRONT
* **Step 3:** SET FRONT = FRONT -> NEXT
* **Step 4:** FREE PTR
* **Step 5:** END

C Function

1. **void** delete (struct node \*ptr)
2. {
3. **if**(front == NULL)
4. {
5. printf("\nUNDERFLOW\n");
6. **return**;
7. }
8. **else**
9. {
10. ptr = front;
11. front = front -> next;
12. free(ptr);
13. }
14. }

Menu-Driven Program implementing all the operations on Linked Queue

1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5. **int** data;
6. struct node \*next;
7. };
8. struct node \*front;
9. struct node \*rear;
10. **void** insert();
11. **void** delete();
12. **void** display();
13. **void** main ()
14. {
15. **int** choice;
16. **while**(choice != 4)
17. {
18. printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
19. printf("\n=================================================================\n");
20. printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
21. printf("\nEnter your choice ?");
22. scanf("%d",& choice);
23. **switch**(choice)
24. {
25. **case** 1:
26. insert();
27. **break**;
28. **case** 2:
29. delete();
30. **break**;
31. **case** 3:
32. display();
33. **break**;
34. **case** 4:
35. exit(0);
36. **break**;
37. **default**:
38. printf("\nEnter valid choice??\n");
39. }
40. }
41. }
42. **void** insert()
43. {
44. struct node \*ptr;
45. **int** item;
46.
47. ptr = (struct node \*) malloc (sizeof(struct node));
48. **if**(ptr == NULL)
49. {
50. printf("\nOVERFLOW\n");
51. **return**;
52. }
53. **else**
54. {
55. printf("\nEnter value?\n");
56. scanf("%d",&item);
57. ptr -> data = item;
58. **if**(front == NULL)
59. {
60. front = ptr;
61. rear = ptr;
62. front -> next = NULL;
63. rear -> next = NULL;
64. }
65. **else**
66. {
67. rear -> next = ptr;
68. rear = ptr;
69. rear->next = NULL;
70. }
71. }
72. }
73. **void** delete ()
74. {
75. struct node \*ptr;
76. **if**(front == NULL)
77. {
78. printf("\nUNDERFLOW\n");
79. **return**;
80. }
81. **else**
82. {
83. ptr = front;
84. front = front -> next;
85. free(ptr);
86. }
87. }
88. **void** display()
89. {
90. struct node \*ptr;
91. ptr = front;
92. **if**(front == NULL)
93. {
94. printf("\nEmpty queue\n");
95. }
96. **else**
97. {   printf("\nprinting values .....\n");
98. **while**(ptr != NULL)
99. {
100. printf("\n%d\n",ptr -> data);
101. ptr = ptr -> next;
102. }
103. }
104. }

What do you mean by Linked list?

A linked list is a linear data structure consisting of elements called nodes where each node is composed of two parts: an information part and a link part, also called the next pointer part.

**Linked list is used in a wide variety of applications such as**

* Polynomial Manipulation representation
* Addition of long positive integers
* Representation of sparse matrices
* Addition of long positive integers
* Symbol table creation
* Mailing list
* Memory management
* Linked allocation of files
* Multiple precision arithmetic etc

**Compare singly and doubly linked list:**



Advantages of Linked Lists

 • They are a dynamic in nature which allocates the memory when required.

• Insertion and deletion operations can be easily implemented.

 • Stacks and queues can be easily executed.

• Linked List reduces the access time.

• Disadvantages of Linked Lists

• The memory is wasted as pointers require extra memory for storage.

 • No element can be accessed randomly; it has to access each node sequentially.

• Reverse Traversing is difficult in linked list.