**Module - 1 (Computer Basics)**

**3. Data Structures**

Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

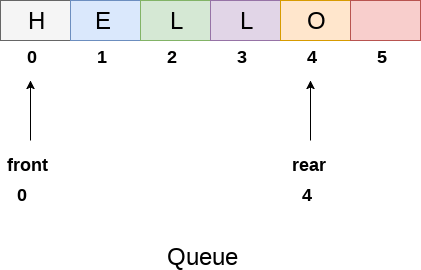
Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

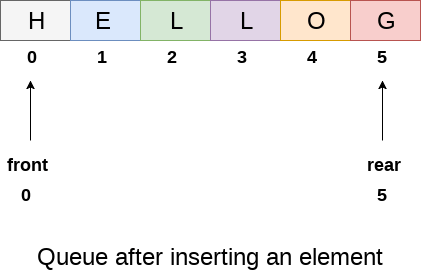
[**next →**](https://www.javatpoint.com/linked-list-implementation-of-queue)[**← prev**](https://www.javatpoint.com/data-structure-queue)

Array representation of Queue

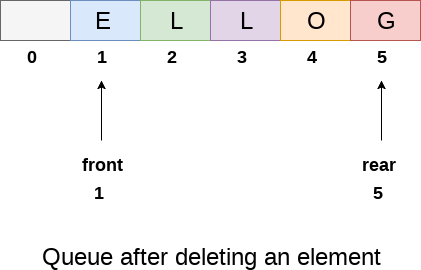
We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

* **Step 1:** IF REAR = MAX - 1  
  Write OVERFLOW  
  Go to step  
  [END OF IF]
* **Step 2:** IF FRONT = -1 and REAR = -1  
  SET FRONT = REAR = 0  
  ELSE  
  SET REAR = REAR + 1  
  [END OF IF]
* **Step 3:** Set QUEUE[REAR] = NUM
* **Step 4:** EXIT

C Function

1. **void** insert (**int** queue[], **int** max, **int** front, **int** rear, **int** item)
2. {
3. **if** (rear + 1 == max)
4. {
5. printf("overflow");
6. }
7. **else**
8. {
9. **if**(front == -1 && rear == -1)
10. {
11. front = 0;
12. rear = 0;
13. }
14. **else**
15. {
16. rear = rear + 1;
17. }
18. queue[rear]=item;
19. }
20. }

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

* **Step 1:** IF FRONT = -1 or FRONT > REAR  
  Write UNDERFLOW  
  ELSE  
  SET VAL = QUEUE[FRONT]  
  SET FRONT = FRONT + 1  
  [END OF IF]
* **Step 2:** EXIT

C Function

1. **int** delete (**int** queue[], **int** max, **int** front, **int** rear)
2. {
3. **int** y;
4. **if** (front == -1 || front > rear)
6. {
7. printf("underflow");
8. }
9. **else**
10. {
11. y = queue[front];
12. **if**(front == rear)
13. {
14. front = rear = -1;
15. **else**
16. front = front + 1;
18. }
19. **return** y;
20. }
21. }

Stack

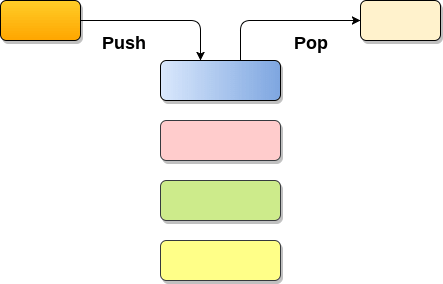
1. Stack is an ordered list in which, insertion and deletion can be performed only at one end that is called **top**.
2. Stack is a recursive data structure having pointer to its top element.
3. Stacks are sometimes called as Last-In-First-Out (LIFO) lists i.e. the element which is inserted first in the stack, will be deleted last from the stack.

Applications of Stack

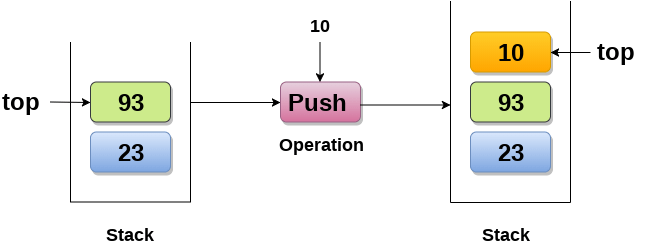
1. Recursion
2. Expression evaluations and conversions
3. Parsing
4. Browsers
5. Editors
6. Tree Traversals

Operations on Stack

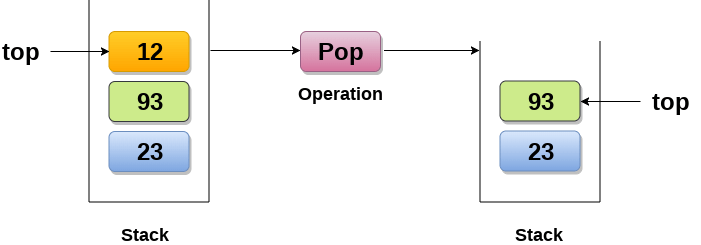
There are various operations which can be performed on stack.



**1. Push :** Adding an element onto the stack



**2. Pop :** Removing an element from the stack

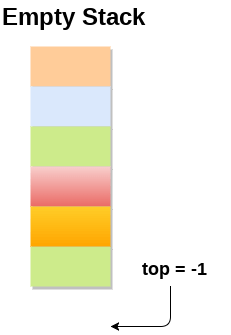


**3. Peek :** Look all the elements of stack without removing them.

How the stack grows?

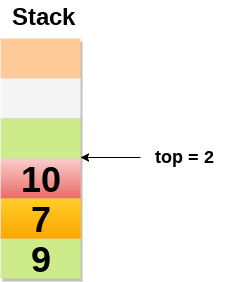
**Scenario 1 : Stack is empty**

The stack is called empty if it doesn't contain any element inside it. At this stage, the value of variable top is -1.



**Scenario 2 : Stack is not empty**

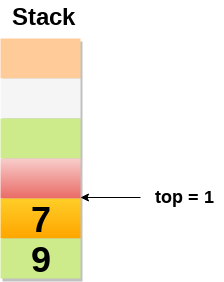
Value of top will get increased by 1 every time when we add any element to the stack. In the following stack, After adding first element, top = 2.



**Scenario 3 : Deletion of an element**

Value of top will get decreased by 1 whenever an element is deleted from the stack.

In the following stack, after deleting 10 from the stack, top = 1.



Top and its value :

|  |  |
| --- | --- |
| **Top position** | **Status of stack** |
| -1 | Empty |
| 0 | Only one element in the stack |
| N-1 | Stack is full |
| N | Overflow |

# Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

### Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

**Algorithm:**

1. begin
2. **if** top = n then stack full
3. top = top + 1
4. stack (top) : = item;
5. end

**Time Complexity : o(1)**

### implementation of push algorithm in C language

1. **void** push (**int** val,**int** n) //n is size of the stack
2. {
3. **if** (top == n )
4. printf("\n Overflow");
5. **else**
6. {
7. top = top +1;
8. stack[top] = val;
9. }
10. }

### Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

**Algorithm :**

1. begin
2. **if** top = 0 then stack empty;
3. item := stack(top);
4. top = top - 1;
5. end;

**Time Complexity : o(1)**

### Implementation of POP algorithm using C language

1. **int** pop ()
2. {
3. **if**(top == -1)
4. {
5. printf("Underflow");
6. **return** 0;
7. }
8. **else**
9. {
10. **return** stack[top - - ];
11. }
12. }

### Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

**Algorithm :**

PEEK (STACK, TOP)

1. Begin
2. **if** top = -1 then stack empty
3. item = stack[top]
4. **return** item
5. End

**Time complexity: o(n)**

### Implementation of Peek algorithm in C language

1. **int** peek()
2. {
3. **if** (top == -1)
4. {
5. printf("Underflow");
6. **return** 0;
7. }
8. **else**
9. {
10. **return** stack [top];
11. }
12. }

**C program**

1. #include <stdio.h>
2. **int** stack[100],i,j,choice=0,n,top=-1;
3. **void** push();
4. **void** pop();
5. **void** show();
6. **void** main ()
7. {
9. printf("Enter the number of elements in the stack ");
10. scanf("%d",&n);
11. printf("\*\*\*\*\*\*\*\*\*Stack operations using array\*\*\*\*\*\*\*\*\*");
13. printf("\n----------------------------------------------\n");
14. **while**(choice != 4)
15. {
16. printf("Chose one from the below options...\n");
17. printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
18. printf("\n Enter your choice \n");
19. scanf("%d",&choice);
20. **switch**(choice)
21. {
22. **case** 1:
23. {
24. push();
25. **break**;
26. }
27. **case** 2:
28. {
29. pop();
30. **break**;
31. }
32. **case** 3:
33. {
34. show();
35. **break**;
36. }
37. **case** 4:
38. {
39. printf("Exiting....");
40. **break**;
41. }
42. **default**:
43. {
44. printf("Please Enter valid choice ");
45. }
46. };
47. }
48. }
50. **void** push ()
51. {
52. **int** val;
53. **if** (top == n )
54. printf("\n Overflow");
55. **else**
56. {
57. printf("Enter the value?");
58. scanf("%d",&val);
59. top = top +1;
60. stack[top] = val;
61. }
62. }
64. **void** pop ()
65. {
66. **if**(top == -1)
67. printf("Underflow");
68. **else**
69. top = top -1;
70. }
71. **void** show()
72. {
73. **for** (i=top;i>=0;i--)
74. {
75. printf("%d\n",stack[i]);
76. }
77. **if**(top == -1)
78. {
79. printf("Stack is empty");
80. }
81. }

# 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.

DS Linked List

## 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.

## Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

## 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.

DS Singly Linked List

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.

## 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 \*));

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.

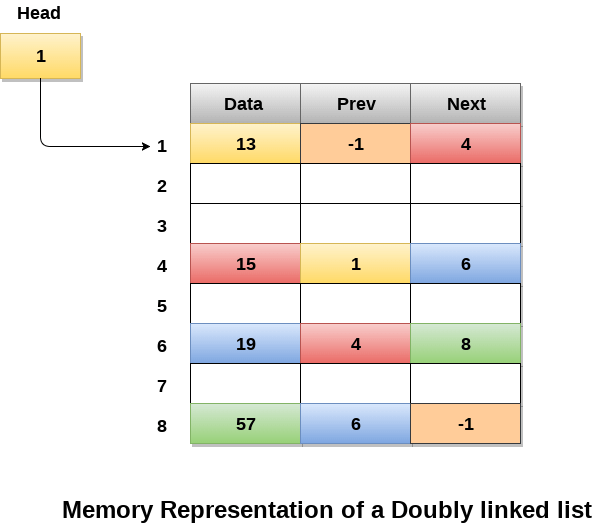
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains 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;

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.

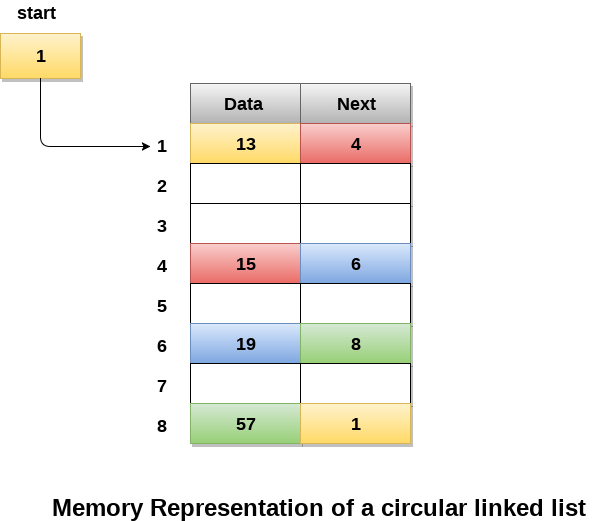


Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and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

# Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An array is collection of items stored at continuous memory locations.  
[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2015/05/Arrays.png)

As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example, if there is situation where only 5 elements are needed to be entered in this array. In this case the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.

Take another situation. In this there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred as **Dynamic Memory Allocation**.

Therefore, **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

Lets see each of them in detail.

### malloc()

**“malloc”** or **“memory allocation”** method is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

**Syntax:**

**ptr = (cast-type\*) malloc(byte-size)**

For Example:

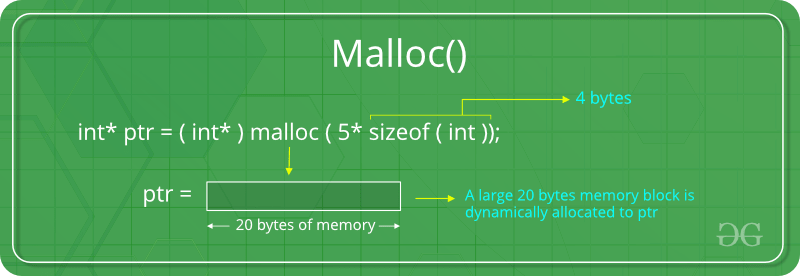
**ptr = (int\*) malloc(100 \* sizeof(int));**

Since the size of int is 4 bytes,

this statement will allocate 400 bytes of memory.

And, the pointer ptr holds the address

of the first byte in the allocated memory.



If the space is insufficient, allocation fails and returns a NULL pointer.

**Example:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    int main()  {        // This pointer will hold the      // base address of the block created      int\* ptr;      int n, i, sum = 0;        // Get the number of elements for the array      n = 5;      printf("Enter number of elements: %d\n", n);        // Dynamically allocate memory using malloc()      ptr = (int\*)malloc(n \* sizeof(int));        // Check if the memory has been successfully      // allocated by malloc or not      if (ptr == NULL) {          printf("Memory not allocated.\n");          exit(0);      }      else {            // Memory has been successfully allocated          printf("Memory successfully allocated using malloc.\n");            // Get the elements of the array          for (i = 0; i < n; ++i) {              ptr[i] = i + 1;          }            // Print the elements of the array          printf("The elements of the array are: ");          for (i = 0; i < n; ++i) {              printf("%d, ", ptr[i]);          }      }        return 0;  } |

**Output:**

Enter number of elements: 5

Memory successfully allocated using malloc.

The elements of the array are: 1, 2, 3, 4, 5,

### calloc()

**“calloc”** or **“contiguous allocation”** method is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

**Syntax:**

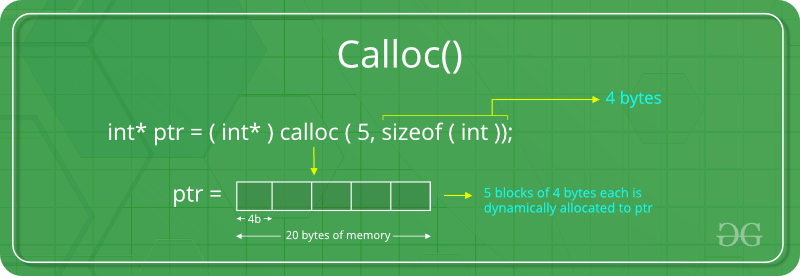
**ptr = (cast-type\*)calloc(n, element-size);**

For Example:

**ptr = (float\*) calloc(25, sizeof(float));**

This statement allocates contiguous space in memory

for 25 elements each with the size of float.



If the space is insufficient, allocation fails and returns a NULL pointer.

**Example:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    int main()  {        // This pointer will hold the      // base address of the block created      int\* ptr;      int n, i, sum = 0;        // Get the number of elements for the array      n = 5;      printf("Enter number of elements: %d\n", n);        // Dynamically allocate memory using calloc()      ptr = (int\*)calloc(n, sizeof(int));        // Check if the memory has been successfully      // allocated by malloc or not      if (ptr == NULL) {          printf("Memory not allocated.\n");          exit(0);      }      else {            // Memory has been successfully allocated          printf("Memory successfully allocated using calloc.\n");            // Get the elements of the array          for (i = 0; i < n; ++i) {              ptr[i] = i + 1;          }            // Print the elements of the array          printf("The elements of the array are: ");          for (i = 0; i < n; ++i) {              printf("%d, ", ptr[i]);          }      }        return 0;  } |

**Output:**

Enter number of elements: 5

Memory successfully allocated using calloc.

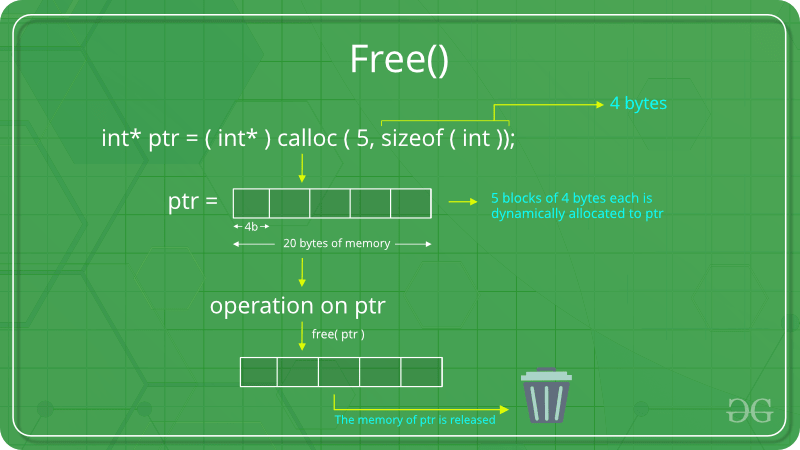
The elements of the array are: 1, 2, 3, 4, 5,

### free()

**“free”** method is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() are not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

**Syntax:**

**free(ptr);**



**Example:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    int main()  {        // This pointer will hold the      // base address of the block created      int \*ptr, \*ptr1;      int n, i, sum = 0;        // Get the number of elements for the array      n = 5;      printf("Enter number of elements: %d\n", n);        // Dynamically allocate memory using malloc()      ptr = (int\*)malloc(n \* sizeof(int));        // Dynamically allocate memory using calloc()      ptr1 = (int\*)calloc(n, sizeof(int));        // Check if the memory has been successfully      // allocated by malloc or not      if (ptr == NULL || ptr1 == NULL) {          printf("Memory not allocated.\n");          exit(0);      }      else {            // Memory has been successfully allocated          printf("Memory successfully allocated using malloc.\n");            // Free the memory          free(ptr);          printf("Malloc Memory successfully freed.\n");            // Memory has been successfully allocated          printf("\nMemory successfully allocated using calloc.\n");            // Free the memory          free(ptr1);          printf("Calloc Memory successfully freed.\n");      }        return 0;  } |

**Output:**

Enter number of elements: 5

Memory successfully allocated using malloc.

Malloc Memory successfully freed.

Memory successfully allocated using calloc.

Calloc Memory successfully freed.

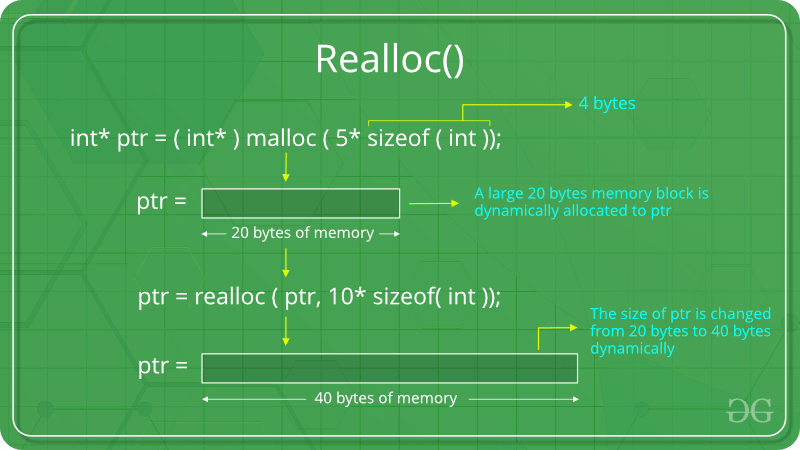
### realloc()

**“realloc”** or **“re-allocation”** method is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.

**Syntax:**

**ptr = realloc(ptr, newSize);**

where ptr is reallocated with new size 'newSize'.



If the space is insufficient, allocation fails and returns a NULL pointer.

**Example:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    int main()  {        // This pointer will hold the      // base address of the block created      int\* ptr;      int n, i, sum = 0;        // Get the number of elements for the array      n = 5;      printf("Enter number of elements: %d\n", n);        // Dynamically allocate memory using calloc()      ptr = (int\*)calloc(n, sizeof(int));        // Check if the memory has been successfully      // allocated by malloc or not      if (ptr == NULL) {          printf("Memory not allocated.\n");          exit(0);      }      else {            // Memory has been successfully allocated          printf("Memory successfully allocated using calloc.\n");            // Get the elements of the array          for (i = 0; i < n; ++i) {              ptr[i] = i + 1;          }            // Print the elements of the array          printf("The elements of the array are: ");          for (i = 0; i < n; ++i) {              printf("%d, ", ptr[i]);          }            // Get the new size for the array          n = 10;          printf("\n\nEnter the new size of the array: %d\n", n);            // Dynamically re-allocate memory using realloc()          ptr = realloc(ptr, n \* sizeof(int));            // Memory has been successfully allocated          printf("Memory successfully re-allocated using realloc.\n");            // Get the new elements of the array          for (i = 5; i < n; ++i) {              ptr[i] = i + 1;          }            // Print the elements of the array          printf("The elements of the array are: ");          for (i = 0; i < n; ++i) {              printf("%d, ", ptr[i]);          }            free(ptr);      }        return 0;  } |

**Output:**

Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10

Memory successfully re-allocated using realloc.

The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

Binary Tree Data Structure

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

Tree Traversals (Inorder, Preorder and Postorder)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



*Example Tree*

Depth First Traversals:  
(a) Inorder (Left, Root, Right) : 4 2 5 1 3  
(b) Preorder (Root, Left, Right) : 1 2 4 5 3  
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5  
Please see [this](https://www.geeksforgeeks.org/level-order-tree-traversal/)post for Breadth First Traversal.

**Inorder Traversal (**[**Practice**](https://practice.geeksforgeeks.org/problems/inorder-traversal/1)**):**

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)

2. Visit the root.

3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder  
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.  
Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

**Preorder Traversal (**[**Practice**](https://practice.geeksforgeeks.org/problems/preorder-traversal/1)**):**

Algorithm Preorder(tree)

1. Visit the root.

2. Traverse the left subtree, i.e., call Preorder(left-subtree)

3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder  
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see <http://en.wikipedia.org/wiki/Polish_notation> to know why prefix expressions are useful.  
Example: Preorder traversal for the above given figure is 1 2 4 5 3.

**Postorder Traversal (**[**Practice**](https://practice.geeksforgeeks.org/problems/postorder-traversal/1)**):**

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)

2. Traverse the right subtree, i.e., call Postorder(right-subtree)

3. Visit the root.

|  |
| --- |
| // C program for different tree traversals  #include <iostream>  using namespace std;    /\* A binary tree node has data, pointer to left child  and a pointer to right child \*/  struct Node  {      int data;      struct Node\* left, \*right;      Node(int data)      {          this->data = data;          left = right = NULL;      }  };    /\* Given a binary tree, print its nodes according to the  "bottom-up" postorder traversal. \*/  void printPostorder(struct Node\* node)  {      if (node == NULL)          return;        // first recur on left subtree      printPostorder(node->left);        // then recur on right subtree      printPostorder(node->right);        // now deal with the node      cout << node->data << " ";  }    /\* Given a binary tree, print its nodes in inorder\*/  void printInorder(struct Node\* node)  {      if (node == NULL)          return;        /\* first recur on left child \*/      printInorder(node->left);        /\* then print the data of node \*/      cout << node->data << " ";        /\* now recur on right child \*/      printInorder(node->right);  }    /\* Given a binary tree, print its nodes in preorder\*/  void printPreorder(struct Node\* node)  {      if (node == NULL)          return;        /\* first print data of node \*/      cout << node->data << " ";        /\* then recur on left sutree \*/      printPreorder(node->left);        /\* now recur on right subtree \*/      printPreorder(node->right);  }    /\* Driver program to test above functions\*/  int main()  {      struct Node \*root = new Node(1);      root->left             = new Node(2);      root->right         = new Node(3);      root->left->left     = new Node(4);      root->left->right = new Node(5);        cout << "\nPreorder traversal of binary tree is \n";      printPreorder(root);        cout << "\nInorder traversal of binary tree is \n";      printInorder(root);        cout << "\nPostorder traversal of binary tree is \n";      printPostorder(root);        return 0;  } |

**Output:**

Preorder traversal of binary tree is

1 2 4 5 3

Inorder traversal of binary tree is

4 2 5 1 3

Postorder traversal of binary tree is

4 5 2 3 1

## Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

## Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

In a tree, all nodes share common construct.

## BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

* **Insert** − Inserts an element in a tree/create a tree.
* **Search** − Searches an element in a tree.
* **Preorder Traversal** − Traverses a tree in a pre-order manner.
* **Inorder Traversal** − Traverses a tree in an in-order manner.
* **Postorder Traversal** − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

## Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

### Algorithm

If root is NULL

then create root node

return

If root exists then

compare the data with node.data

while until insertion position is located

If data is greater than node.data

goto right subtree

else

goto left subtree

endwhile

insert data

end If

### Implementation

The implementation of insert function should look like this −

void insert(int data) {

struct node \*tempNode = (struct node\*) malloc(sizeof(struct node));

struct node \*current;

struct node \*parent;

tempNode->data = data;

tempNode->leftChild = NULL;

tempNode->rightChild = NULL;

//if tree is empty, create root node

if(root == NULL) {

root = tempNode;

} else {

current = root;

parent = NULL;

while(1) {

parent = current;

//go to left of the tree

if(data < parent->data) {

current = current->leftChild;

//insert to the left

if(current == NULL) {

parent->leftChild = tempNode;

return;

}

}

//go to right of the tree

else {

current = current->rightChild;

//insert to the right

if(current == NULL) {

parent->rightChild = tempNode;

return;

}

}

}

}

}

## Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

### Algorithm

If root.data is equal to search.data

return root

else

while data not found

If data is greater than node.data

goto right subtree

else

goto left subtree

If data found

return node

endwhile

return data not found

end if

The implementation of this algorithm should look like this.

struct node\* search(int data) {

struct node \*current = root;

printf("Visiting elements: ");

while(current->data != data) {

if(current != NULL)

printf("%d ",current->data);

//go to left tree

if(current->data > data) {

current = current->leftChild;

}

//else go to right tree

else {

current = current->rightChild;

}

//not found

if(current == NULL) {

return NULL;

}

return current;

}

}

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

* The left sub-tree of a node has a key less than or equal to its parent node's key.
* The right sub-tree of a node has a key greater than to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left\_subtree (keys) ≤ node (key) ≤ right\_subtree (keys)

## Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

## Basic Operations

Following are the basic operations of a tree −

* **Search** − Searches an element in a tree.
* **Insert** − Inserts an element in a tree.
* **Pre-order Traversal** − Traverses a tree in a pre-order manner.
* **In-order Traversal** − Traverses a tree in an in-order manner.
* **Post-order Traversal** − Traverses a tree in a post-order manner.

## Node

Define a node having some data, references to its left and right child nodes.

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

## Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

### Algorithm

struct node\* search(int data){

struct node \*current = root;

printf("Visiting elements: ");

while(current->data != data){

if(current != NULL) {

printf("%d ",current->data);

//go to left tree

if(current->data > data){

current = current->leftChild;

} //else go to right tree

else {

current = current->rightChild;

}

//not found

if(current == NULL){

return NULL;

}

}

}

return current;

}

## Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

### Algorithm

void insert(int data) {

struct node \*tempNode = (struct node\*) malloc(sizeof(struct node));

struct node \*current;

struct node \*parent;

tempNode->data = data;

tempNode->leftChild = NULL;

tempNode->rightChild = NULL;

//if tree is empty

if(root == NULL) {

root = tempNode;

} else {

current = root;

parent = NULL;

while(1) {

parent = current;

//go to left of the tree

if(data < parent->data) {

current = current->leftChild;

//insert to the left

if(current == NULL) {

parent->leftChild = tempNode;

return;

}

} //go to right of the tree

else {

current = current->rightChild;

//insert to the right

if(current == NULL) {

parent->rightChild = tempNode;

return;

}

}

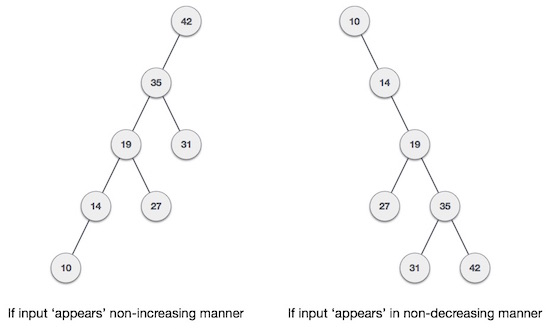
}

}

}

**AVL TREE**

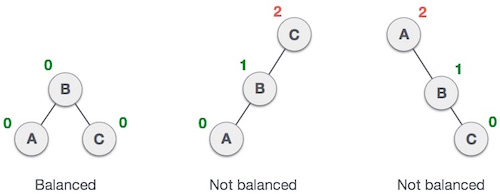
What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



It is observed that BST's worst-case performance is closest to linear search algorithms, that is Ο(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced −



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

***BalanceFactor*** = height(left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

## AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations −

* Left rotation
* Right rotation
* Left-Right rotation
* Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

### Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

## Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

### Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

|  |  |
| --- | --- |
| **State** | **Action** |
| Right Rotation | A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation. |
| Left Rotation | We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**. |
| Left Rotation | Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree. |
| Right Rotation | We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree. |
| Balanced Avl Tree | The tree is now balanced. |

### Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

|  |  |
| --- | --- |
| **State** | **Action** |
| Left Subtree of Right Subtree | A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2. |
| Subtree Right Rotation | First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**. |
| Right Unbalanced Tree | Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation. |
| Left Rotation | A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**. |
| Balanced AVL Tree | The tree is now balanced. |

**GRAPH**

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

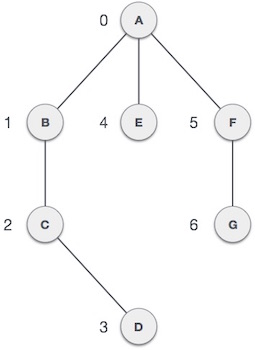
V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

* **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
* **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
* **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
* **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph −

* **Add Vertex** − Adds a vertex to the graph.
* **Add Edge** − Adds an edge between the two vertices of the graph.
* **Display Vertex** − Displays a vertex of the graph.