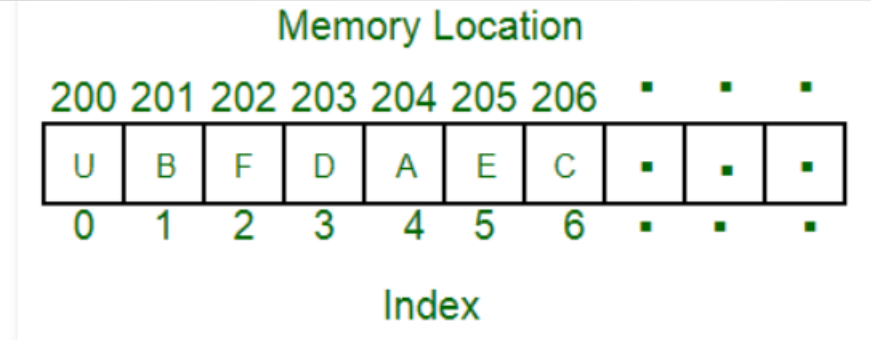

AMIT LEARNING
SESSION 8 DATA STRUCTURE
AMIT

# CONTENTS

# 1 WHAT IS Data structure

## 1.1 Types of Data structure

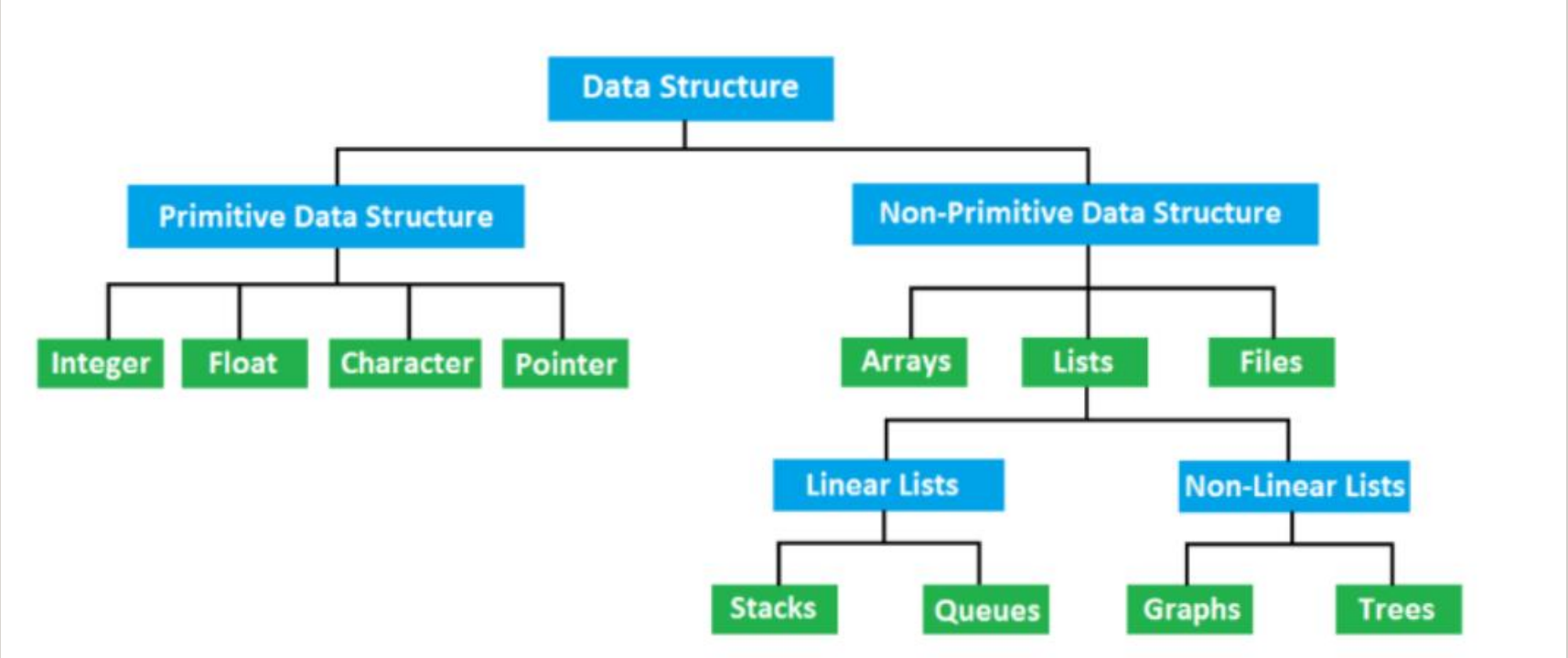
- What is **data structure:**
  - is a particular way of organizing data in a computer so that it can be used effectively.
  - For example, we can store a list of items having the same data-type using the *array* data structure.



# 1 WHAT IS Data structure

## 1.1 Types of Data structure

- Types of **data structure:**

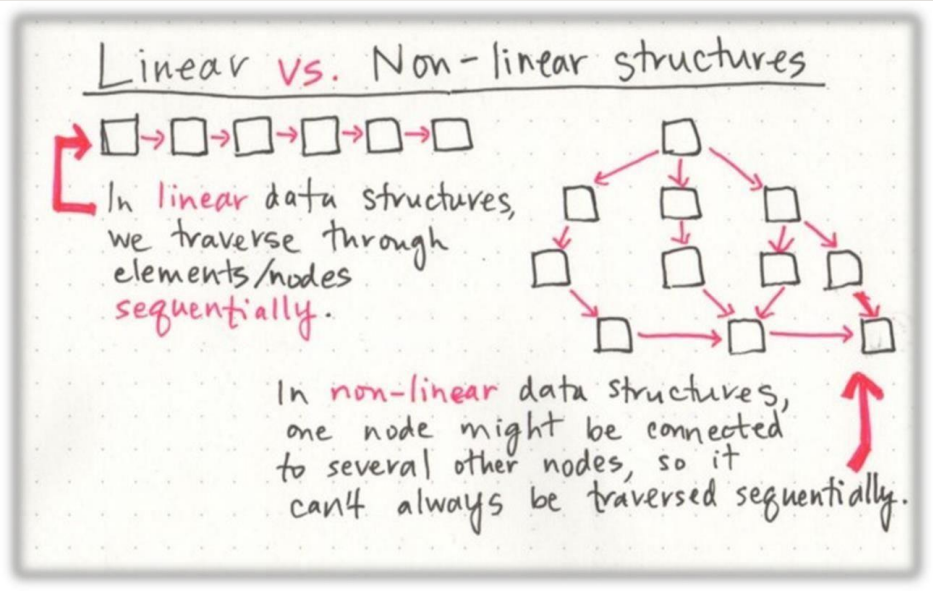


# 1 WHAT IS Data structure

## 1.1 Types of Data structure

- Linear data structures
  - which means that there is sequence and an order to how they are constructed sequentially
  - Examples: arrays and linked lists.

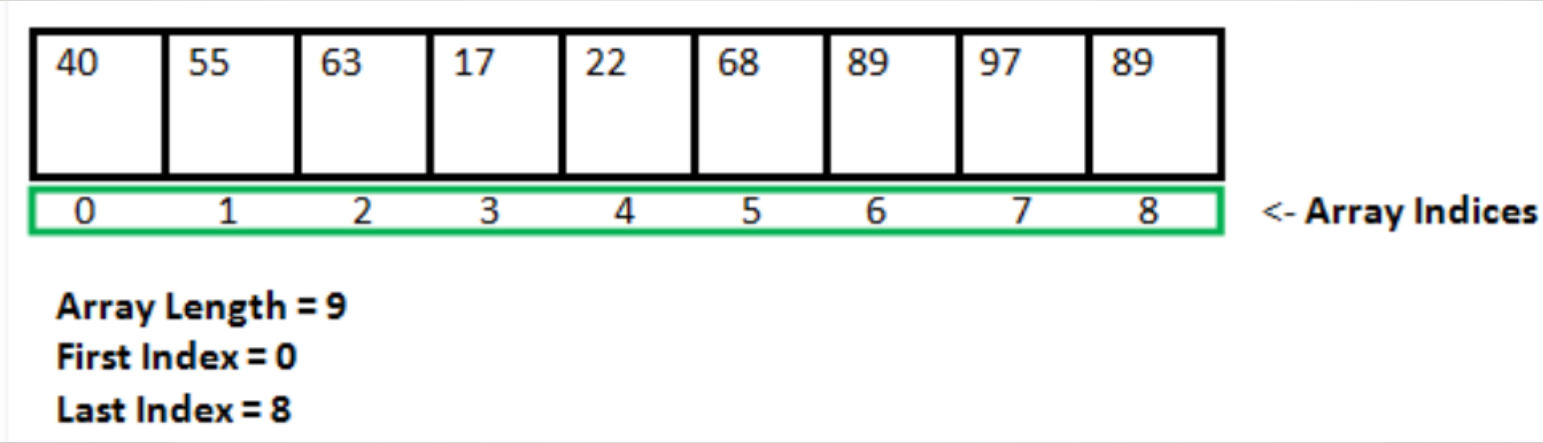
- Non-linear data structures
  - items don't have to be arranged in order, which means that we could arrange the data structure non sequentially.
  - Examples: Trees and graphs



# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

- 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.
- 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 a 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.



# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

- C **Dynamic Memory Allocation** :can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime,store data in heap
- 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:

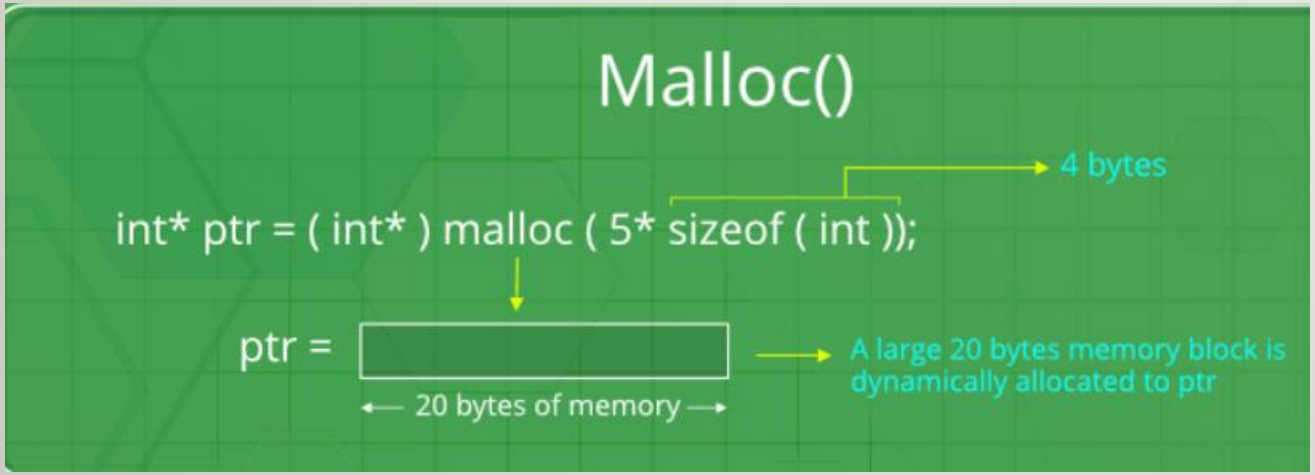
- malloc().
- calloc().
- free().
- realloc().

| Function | Use of Function |
|---|---|
| malloc() | Allocates requested size of bytes and returns a pointer first byte of allocated space |
| calloc() | Allocates space for an array elements, initializes to zero and then returns a pointer to memory |
| free() | dellocate the previously allocated space |
| realloc() | Change the size of previously allocated space |

# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

- malloc():
  - **“malloc”** or **“memory allocation”** method in C 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. It initializes each block with default garbage value.
  - the memory which is allocated by the malloc has the garbage data.
  - If space is insufficient, allocation fails and returns a NULL pointer.




**Syntax:**

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

# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

### ➢ EXAMPLE

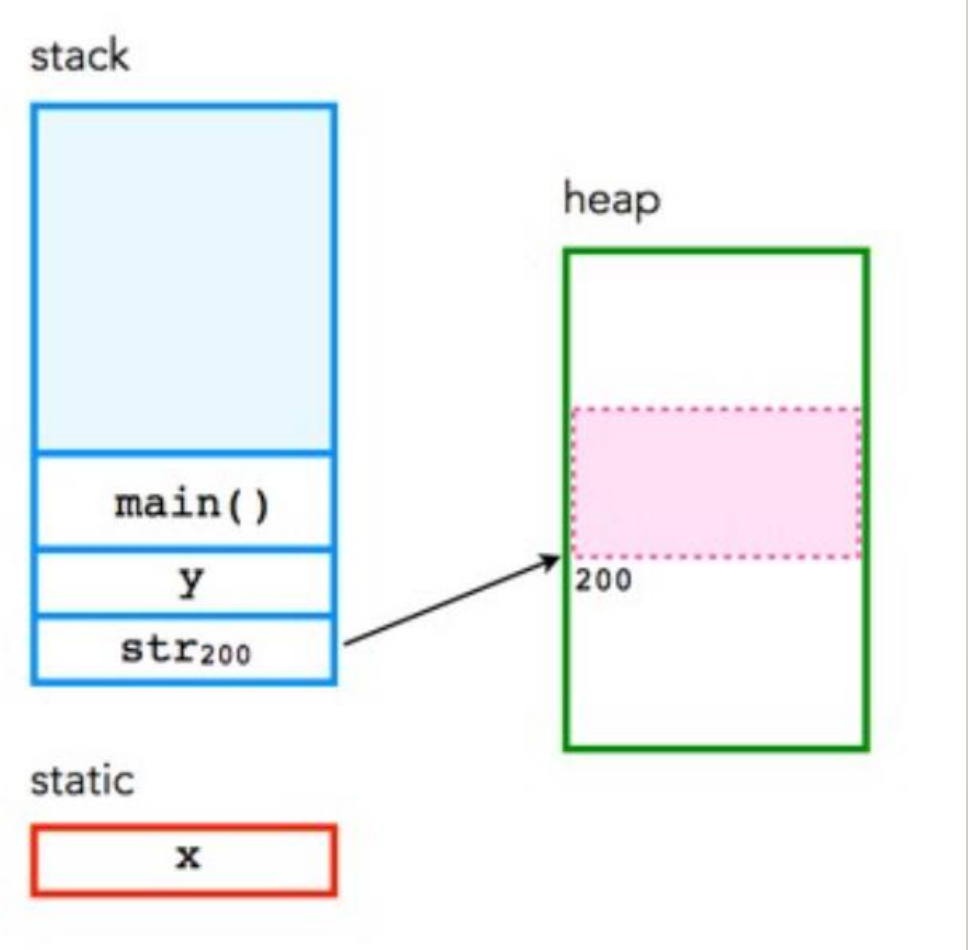
```c
#include <stdio.h>
#include <stdlib.h>

int x;

int main(void)
{
    int y;
    char *str;

    y = 4;
    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c\n", str[0]);
    free(str);
    return 0;
}
```



# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

CODE

```c
int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // 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;
        }
```

# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

### CODE

```
        // 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,
```

### DESCRIPTION

- Allocate memory using dynamic memory allocation MALLOC().

AMIT

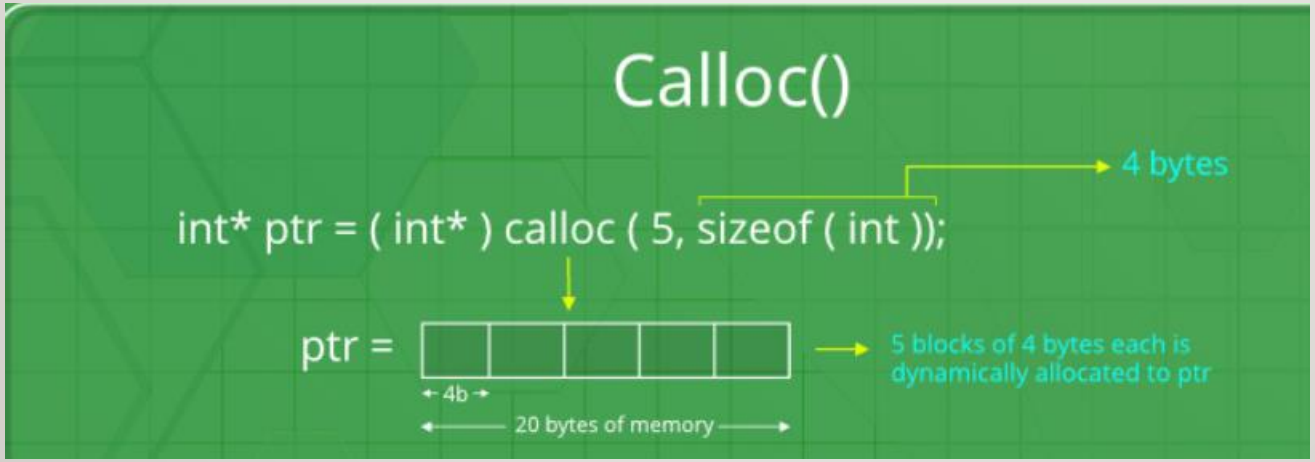
# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

- **calloc:**
- **"contiguous allocation"** method in C 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);
```



# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

CODE

```c
int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // 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 calloc 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
```

# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

### CODE

```
        // 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,
```

### DESCRIPTION

- Allocate memory using dynamic memory allocation.Calloc()

AMIT

# 1 WHAT IS Data structure

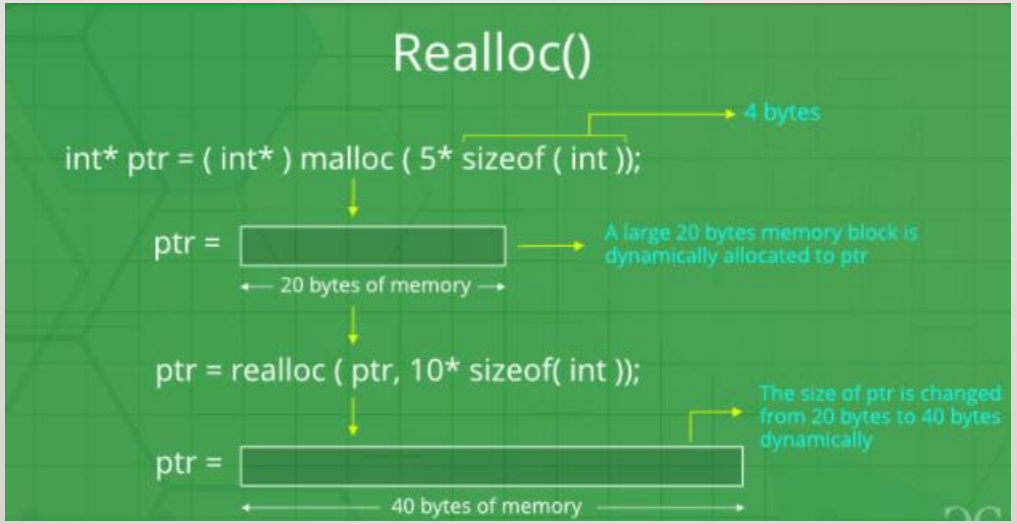
## 1.2 Dynamic memory allocation

- **“realloc”** :
- **“re-allocation”** method in C 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**. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

**Syntax:**

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

where ptr is reallocated with new size 'newSize'.
```



# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

CODE

```c
int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // 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]);
        }
```

# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

### CODE

```
    // 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,
```

### DESCRIPTION

- Allocate memory using dynamic memory allocation.Calloc()

AMIT

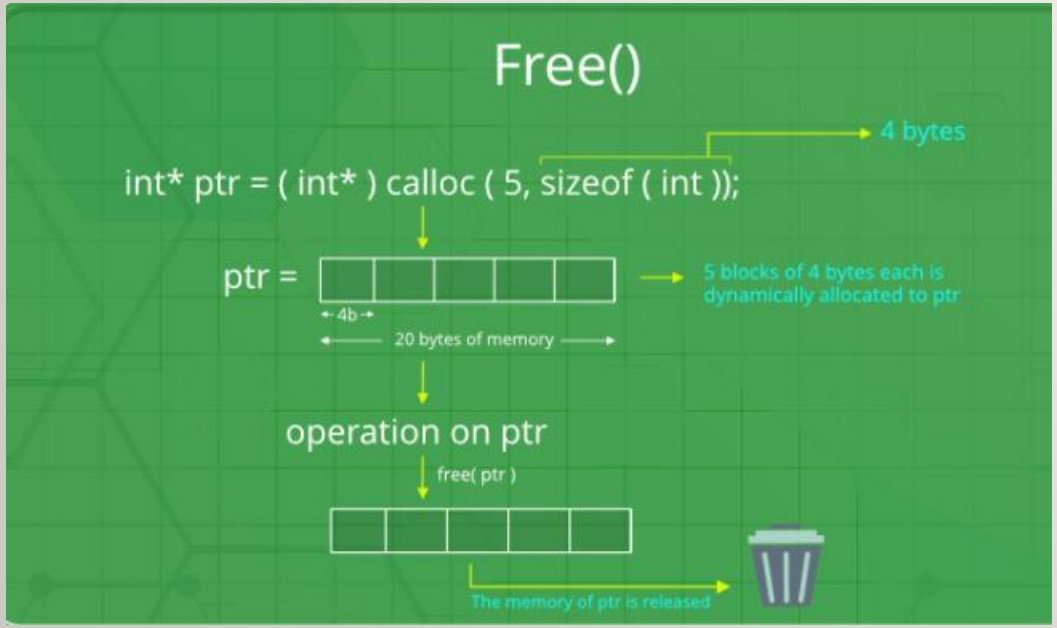
# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation

- **“free”** :
- method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is 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);
```



# 1 WHAT IS Data structure

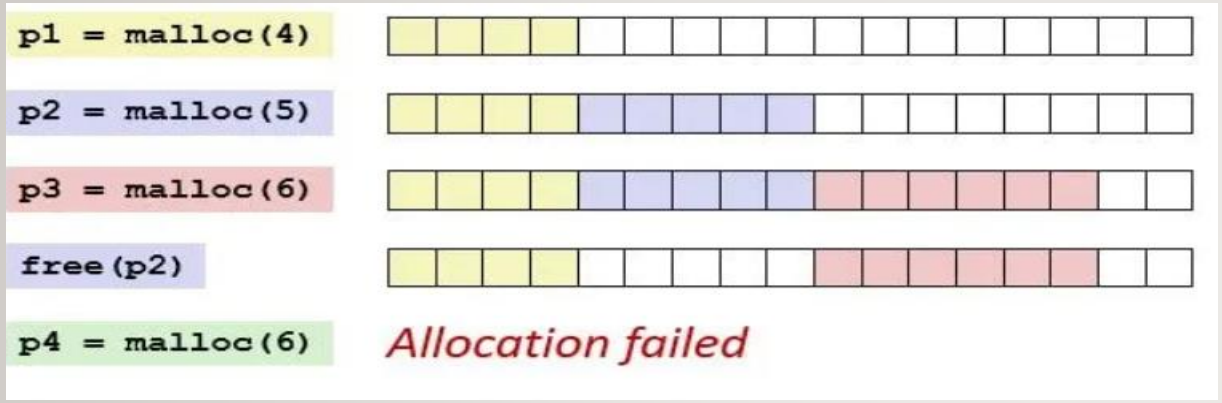
## 1.2 Dynamic memory allocation

- **What is the memory leak in C?**
  - memory leak occurs when you allocate a block of memory using the memory management function and forget to release it.
- **What is dynamic memory fragmentation?**
  - he memory management function gives the guaranteed that allocated memory would be aligned with the object. The fundamental alignment is less than or equal to the largest alignment that's supported by the implementation without an alignment specification.
  - One of the major problems with dynamic memory allocation is fragmentation, basically, fragmentation occurred when the user does not use the memory efficiently
  - Consider a scenario where the program has 3 contiguous blocks of memory and the user frees the middle block of memory. In that scenario, you will not get a memory, if the required block is larger than a single block of memory. See the below Image,

AMIT

# 1 WHAT IS Data structure

## 1.2 Dynamic memory allocation



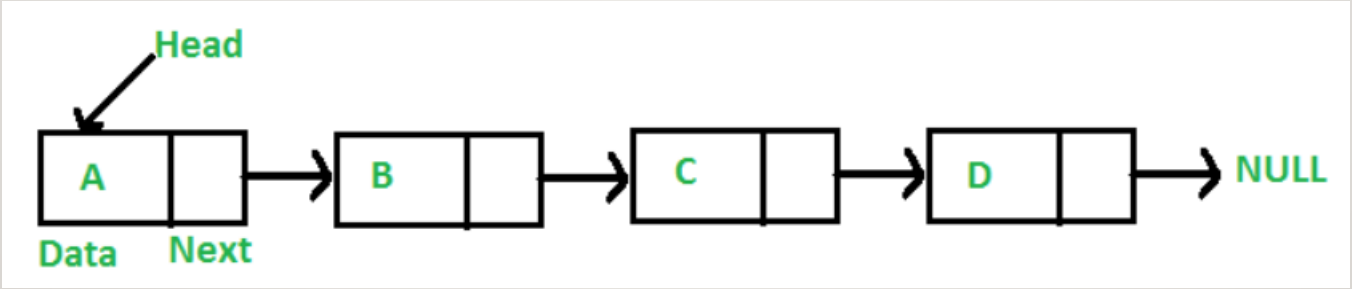
# 1 WHAT IS Data structure

## 1.3 What is linked list

- A linked list:

- is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:

- In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.



# 1 WHAT IS Data structure

## 1.4 types of linked list

- types of linked list:
  1. Singly Linked List


  2. Circular Linked List


  3. Doubly Linked List


AMIT

1. Singly Linked List:

- We created a simple linked list with 3 nodes
- **Linked List Operations:**
  - **Traverse**
  - **Insert**
  - **Delete**

➢ Node is represented as:

```
struct node {
    int data;
    struct node *next;
}
```

# 1 WHAT IS Data structure

## 1.5 Singly linked list

- create singly linked list by three-member and Traverse

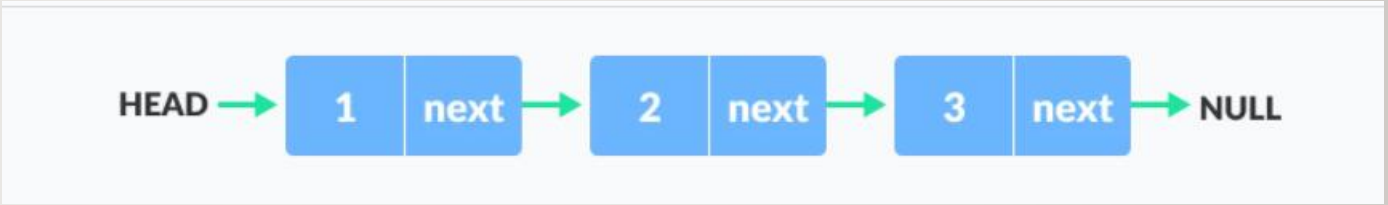
```c
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```



# 1 WHAT IS Data structure

## 1.5 Singly linked list

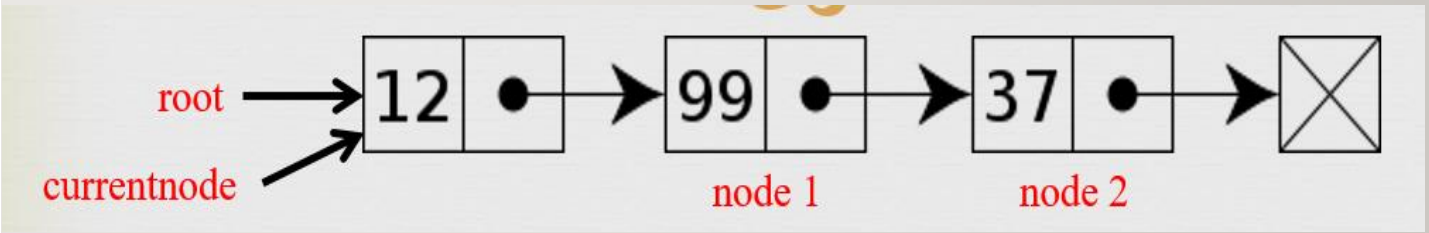
CODE

```
#define   NULL (void*)0
struct node
{
  int data;
  struct node *next;
};
void linkedlist(void);


int main()
{
/* Initialize nodes */
struct node *head;
struct node*cuurent;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = (struct node*)malloc(sizeof(struct node));
two = (struct node*)malloc(sizeof(struct node));
three = (struct node*)malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data=3;
```




AMIT

# 1 WHAT IS Data structure

## 1.5 Singly linked list

### CODE

```
/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
cuurent=head;
while(cuurent!=NULL)
{
     printf("cuurent->data:%d\n",cuurent->data);
    cuurent=cuurent->next;

}

     return 0;
}
```

### OUTPUT

```
cuurent->data:1
cuurent->data:2
cuurent->data:3
```

### DESCRIPTION

- Print data in single linked list.

# 1 WHAT IS Data structure

## 1.5 Singly linked list

### CODE

```
/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
cuurent=head;
while(cuurent!=NULL)
{
     printf("cuurent->data:%d\n",cuurent->data);
    cuurent=cuurent->next;

}

     return 0;
}
```

### OUTPUT

```
cuurent->data:1
cuurent->data:2
cuurent->data:3
```

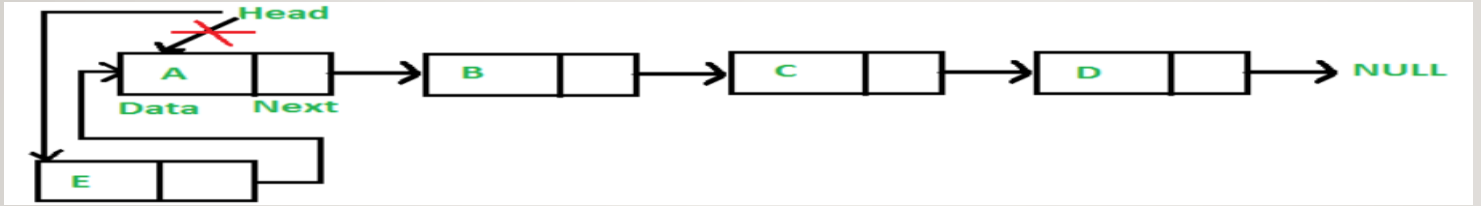
### DESCRIPTION

- Print data in single linked list.

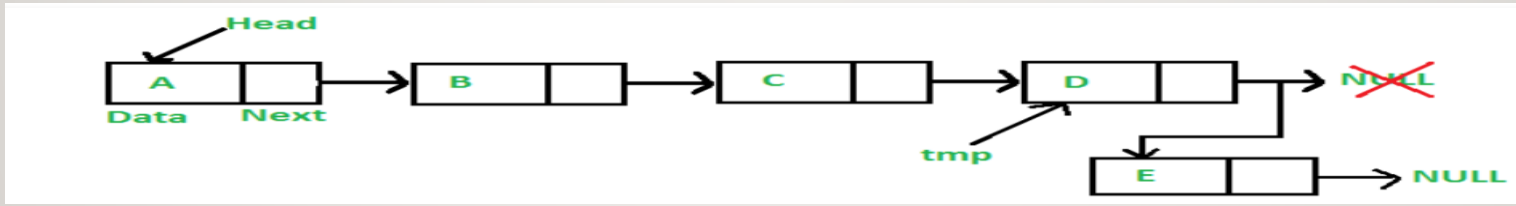
1 WHAT IS Data structure

1.5 Singly linked list

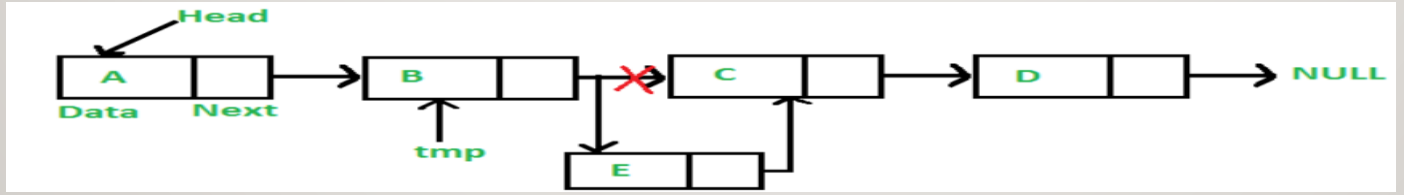
- **Linked List Operations:**
  - **Insert:**
    1. **Insert at the beginning**




    1. **Insert at the End**




    1. **Insert at the Middle**



# 1 WHAT IS Data structure

## 1.5 Singly linked list

CODE

```
struct node
{
  int data;
  struct node *next;
};
void linkedlist(void);


int main()
{
/* Initialize nodes */
struct node *head;
struct node*cuurent;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;
struct node *newNode;

/* Allocate memory */
one = (struct node*)malloc(sizeof(struct node));
two = (struct node*)malloc(sizeof(struct node));
three = (struct node*)malloc(sizeof(struct node));
newNode =(struct node*) malloc(sizeof(struct node));
```

# 1 WHAT IS Data structure

## 1.5 Singly linked list

### CODE

```
/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;
head=one;
newNode->next = head;
head = newNode;
cuurent=head;
while(cuurent!=NULL)
{
    printf("cuurent->data:%d\n",cuurent->data);
   cuurent=cuurent->next;

}

    return 0;
}
```

### OUTPUT

```
cuurent->data:4
cuurent->data:1
cuurent->data:2
cuurent->data:3
```

### DESCRIPTION

- **Insert at the beginning**.

# 1 WHAT IS Data structure

## 1.5 Singly linked list

CODE

```
struct node
{
  int data;
  struct node *next;
};
void linkedlist(void);


int main()
{
/* Initialize nodes */
struct node *head,*newNode;
struct node*cuurent;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;


/* Allocate memory */
one = (struct node*)malloc(sizeof(struct node));
two = (struct node*)malloc(sizeof(struct node));
three = (struct node*)malloc(sizeof(struct node));
newNode =(struct node*) malloc(sizeof(struct node));
```

AMIT

# 1 WHAT IS Data structure

## 1.5 Singly linked list

### CODE

```
/* Assign data values */
one->data = 1;
two->data = 2;
three->data=3;
newNode->data = 4;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;
newNode->next = three->next;
three->next=newNode;
head=one;

cuurent=head;
while(cuurent!=NULL)
{
    printf("cuurent->data:%d\n",cuurent->data);
   cuurent=cuurent->next;

}

    return 0;
}
```

### OUTPUT

```
cuurent->data:1
cuurent->data:2
cuurent->data:3
cuurent->data:4
```

### DESCRIPTION

- **Insert node at the end**.

# 1 WHAT IS Data structure

## 1.5 Singly linked list

CODE

```
struct node
{
   int data;
   struct node *next;
};
void linkedlist(void);


int main()
{
/* Initialize nodes */
struct node *head,*newNode,*temp,*current;
struct node*cuurent;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;
int position;


/* Allocate memory */
one = (struct node*)malloc(sizeof(struct node));
two = (struct node*)malloc(sizeof(struct node));
three = (struct node*)malloc(sizeof(struct node));
newNode =(struct node*) malloc(sizeof(struct node));
```

# 1 WHAT IS Data structure

## 1.5 Singly linked list

### CODE

```
two->data = 2;
three->data=3;
newNode->data = 4;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;
head=one;

temp = head;
current=head;
scanf("%d",&position);


for(int i=2; i < position; i++) {
    if(temp->next != NULL) {
        temp = temp->next;
    }
}
newNode->next = temp->next;
temp->next = newNode;

while(current!=NULL)
{
    printf("current->data:%d\n",current->data);
    current=current->next;
}
```

### OUTPUT

```
enter position:2
current->data:1
current->data:4
current->data:2
current->data:3
```

### DESCRIPTION

- **Insert node at the middle**.

# 1 WHAT IS Data structure

## 1.5 Singly linked list

### CODE

```
void deleteNode( int key) {

  struct Node *temp = head, *prev;

  if (temp != NULL && temp->item == key) {
    head = temp->next;
    free(temp);
    return;
  }
  // Find the key to be deleted
  while (temp != NULL && temp->item != key) {
    prev = temp;
    temp = temp->next;
  }
```

```
// Driver program
int main() {


  insertAtEnd( 1);
  insertAtBeginning(2);
  insertAtBeginning( 3);
  insertAtEnd( 4);
  insertAfter(head->next, 5);

  printf("Linked list: ");
  printList(head);

  printf("\nAfter deleting an element: ");
  deleteNode(3);
  printList(head);
}
```

### OUTPUT

```
Linked list:  3  2  5  1  4
After deleting an element:  2  5  1  4
Process returned 0 (0x0)   execution time : 0.014 s
Press any key to continue.
```

### DESCRIPTION

- **Delete at any node**

# 1 WHAT IS Data structure

## 1.5 Singly linked list

### CODE

```c
#include <stdio.h>
#include <stdlib.h>

// Create a node
struct Node {
  int item;
  struct Node* next;
};
 struct Node* head = NULL;

void insertAtBeginning (int data) {

  // Allocate memory to a node
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));


  // insert the item
  new_node->item = data;
  new_node->next = head;
  // Move head to new node
  head= new_node;
}
```

### DESCRIPTION

- **Insert at beginning node**

# 1 WHAT IS Data structure

## 1.5 Singly linked list

**CODE**

```
// Insert a node after a node
void insertAfter(struct Node* node, int data) {
  if (node == NULL) {
    printf("the given previous node cannot be NULL");
    return;
  }

  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  new_node->item = data;
  new_node->next = node->next;
  node->next = new_node;
}

void insertAtEnd( int data) {

  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  struct Node* last = head;

  new_node->item = data;
  new_node->next = NULL;

  if (head == NULL) {
    head = new_node;
    return;
  }
```

```
  while (last->next != NULL)
    last = last->next;

  last->next = new_node;
  return;
}
```

**DESCRIPTION**

- **Insert after node.**

# 1 WHAT IS Data structure

## 1.5 Singly linked list

**CODE**

```
void deleteNode( int key) {

  struct Node *temp = head, *prev;

  if (temp != NULL && temp->item == key) {
    head = temp->next;
    free(temp);
    return;
  }
  // Find the key to be deleted
  while (temp != NULL && temp->item != key) {
    prev = temp;
    temp = temp->next;
  }

  // If the key is not present
  if (temp == NULL) return;

  // Remove the node
  prev->next = temp->next;

  free(temp);
}
```

**DESCRIPTION**

- **Delete node.**

# 1 WHAT IS Data structure

## 1.5 Singly linked list

**CODE**

```
// Print the linked list
void printList(struct Node* node) {
  while (node != NULL) {
    printf(" %d ", node->item);
    node = node->next;
  }
}
```

**DESCRIPTION**

- **Print all nodes.**

# 1 WHAT IS Data structure

## 1.6 Array Vs. Linked list

| ARRAY | LINKED LIST |
| --- | --- |
| Array is a collection of elements of similar data type. | Linked List is an ordered collection of elements of same type, which are connected to each other using pointers. |
| Array supports **Random Access**, which means elements can be accessed directly using their index, like `arr[0]` for 1st element, `arr[6]` for 7th element etc.<br><br>Hence, accessing elements in an array is **fast** with a constant time complexity of `O(1)`. | Linked List supports **Sequential Access**, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element.<br><br>To access **nth** element of a linked list, time complexity is `O(n)`. |
| In an array, elements are stored in **contiguous memory location** or consecutive manner in the memory. | In a linked list, new elements can be stored anywhere in the memory.<br><br>Address of the memory location allocated to the new element is stored in the previous node of linked list, hence formaing a link between the two nodes/elements. |

# 1 WHAT IS Data structure

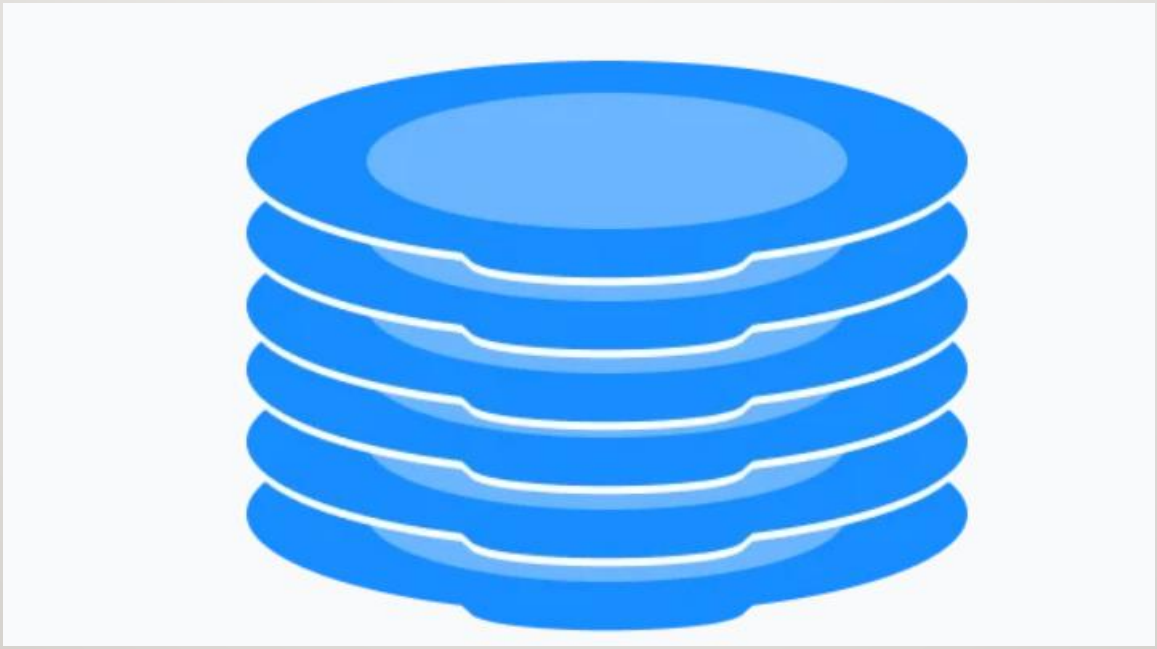
## 1.6 Array Vs. Linked list

| | |
|---|---|
| In array, **Insertion and Deletion** operation takes more time, as the memory locations are consecutive and fixed. | In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list.<br><br>Insertion and Deletion operations are **fast** in linked list. |
| Memory is allocated as soon as the array is declared, at **compile time**. It's also known as **Static Memory Allocation**. | Memory is allocated at **runtime**, as and when a new node is added. It's also known as **Dynamic Memory Allocation**. |
| In array, each element is independent and can be accessed using it's index value. | In case of a linked list, each node/element points to the next, previous, or maybe both nodes. |
| Array can be **single dimensional**, **two dimensional** or **multidimensional** | Linked list can be **Linear(Singly) linked list**, **Doubly linked list** or **Circular linked list** linked list. |
| Size of the array must be specified at time of array decalaration. | Size of a Linked list is variable. It grows at runtime, as more nodes are added to it. |
| Array gets memory allocated in the **Stack** section. | Whereas, linked list gets memory allocated in **Heap** section. |

# 1 WHAT IS Data structure

## 1.7 Stack

- A stack is a useful data structure in programming. It is just like a pile of plates kept on top of each other
- Think about the things you can do with such a pile of plates
  - Put a new plate on top
  - Remove the top plate
- If you want the plate at the bottom, you must first remove all the plates on top. Such an arrangement is called **Last In First Out** - the last item that is the first item to go out.

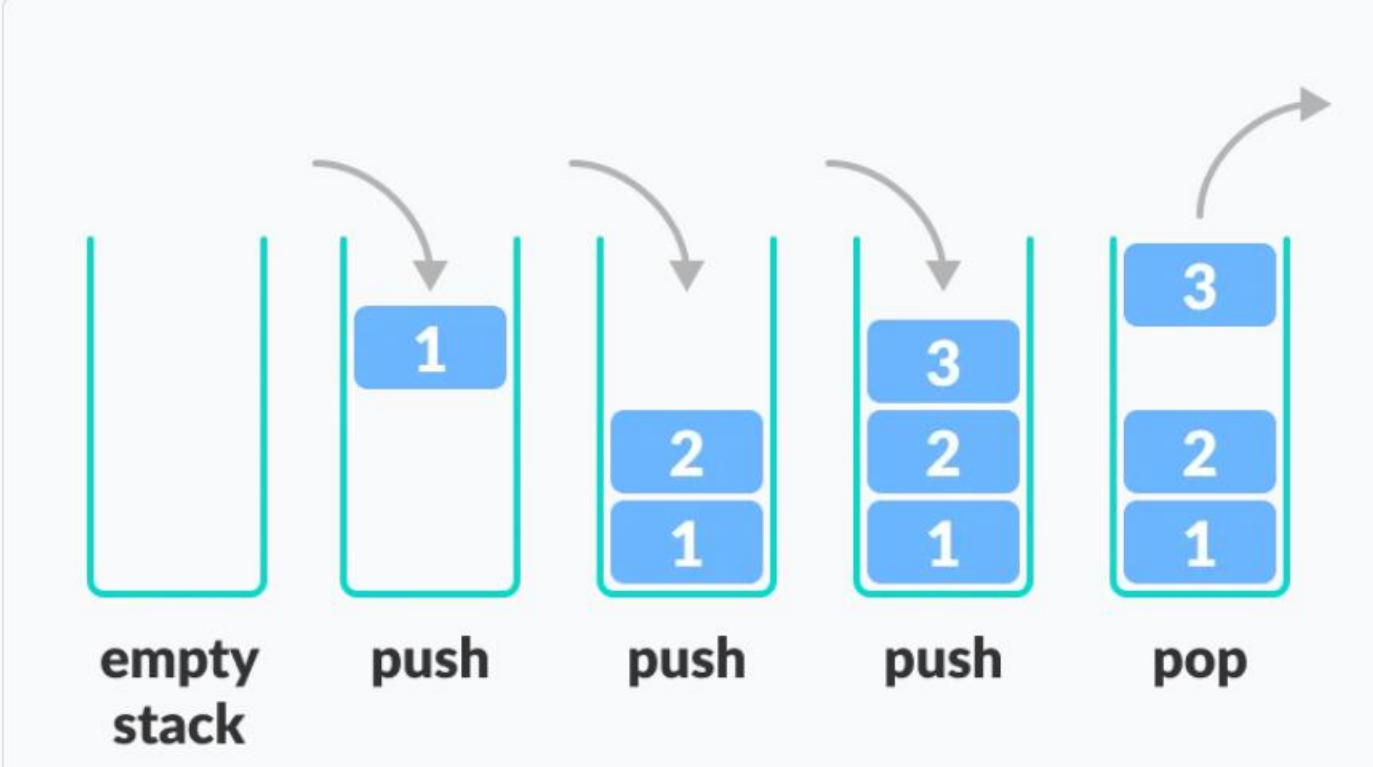


AMIT

1 WHAT IS Data structure

1.7 Stack

- **LIFO Principle of Stack**
  - In programming terms, putting an item on top of the stack is called **push** and removing an item is called **pop**.



1 WHAT IS Data structure

1.7 Stack

## ➢ Basic Operations of Stack

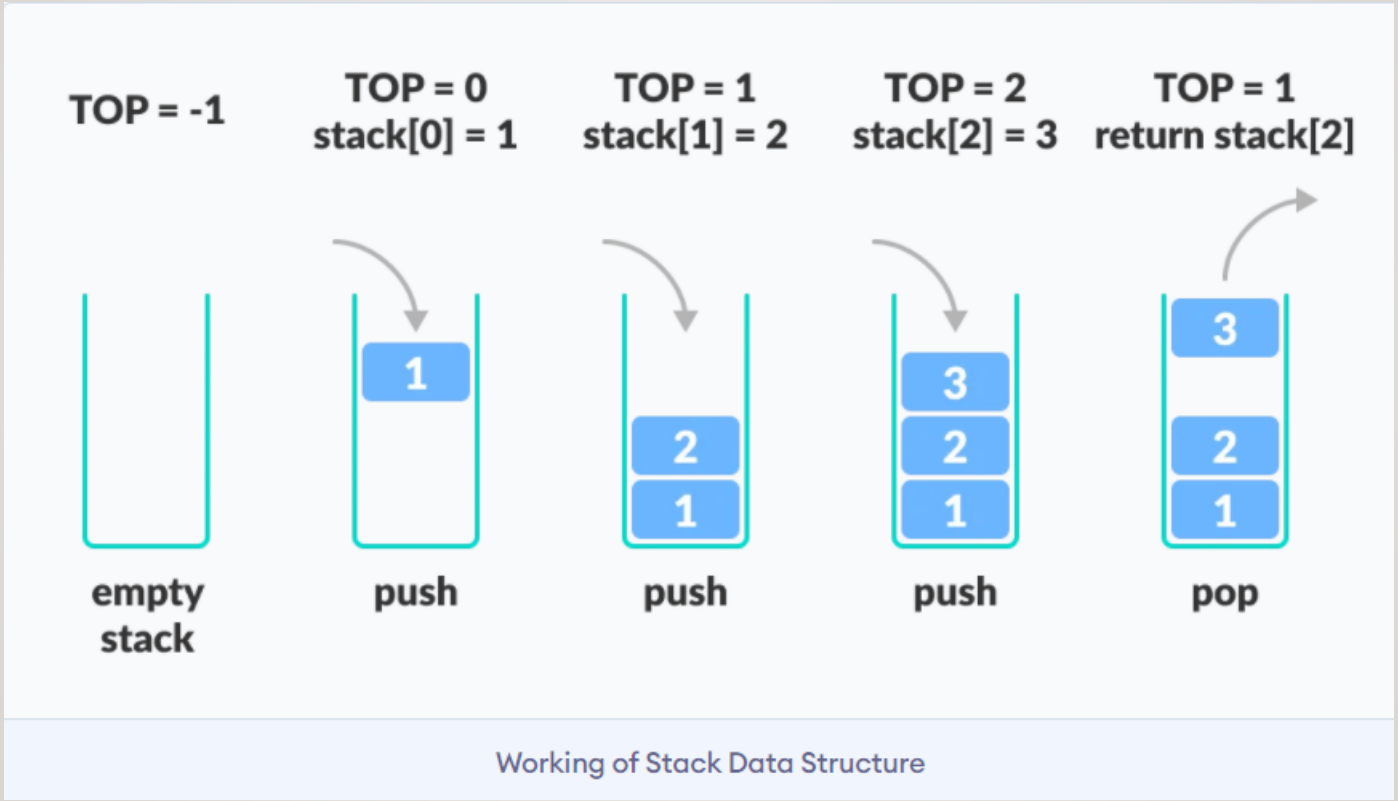
- A stack is an object (an abstract data type - ADT) that allows the following operations:
  - **Push**: Add an element to the top of a stack
  - **Pop**: Remove an element from the top of a stack
  - **IsEmpty**: Check if the stack is empty
  - **IsFull**: Check if the stack is full
  - **Peek**: Get the value of the top element without removing it

## ➢ Working of Stack Data Structure:

- The operations work as follows:
  1. A pointer called TOP is used to keep track of the top element in the stack.
  2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing Top=-1.
  3. On pushing an element, we increase the value of Top and place the new element in the position pointed to by Top
  4. On popping an element, we return the element pointed to by Top and reduce its value.
  5. Before pushing, we check if the stack is already full
  6. Before popping, we check if the stack is already empty

# 1 WHAT IS Data structure

## 1.7 Stack




Working of Stack Data Structure

# 1 WHAT IS Data structure

## 1.7 Stack

### CODE

```
// Driver code
int main() {
  int ch;
  st *s = (st *)malloc(sizeof(st));

  createEmptyStack(s);

  push(s, 1);
  push(s, 2);
  push(s, 3);
  push(s, 4);

  printStack(s);

  pop(s);

  printf("\nAfter popping out\n");
  printStack(s);
}
```

### DESCRIPTION

- Implement stack

AMIT

# 1 WHAT IS Data structure

## 1.7 Stack

### CODE

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 10

int count = 0;

// Creating a stack
struct stack {
  int items[MAX];
  int top;
};
typedef struct stack st;

void createEmptyStack(st *s) {
  s->top = -1;
}

// Check if the stack is full
int isfull(st *s) {
  if (s->top == MAX - 1)
    return 1;
  else
    return 0;
}
```

### DESCRIPTION

- Implement stack

# 1 WHAT IS Data structure

## 1.7 Stack

### CODE

```c
// Check if the stack is empty
int isempty(st *s) {
  if (s->top == -1)
    return 1;
  else
    return 0;
}

// Add elements into stack
void push(st *s, int newitem) {
  if (isfull(s)) {
    printf("STACK FULL");
  } else {
    s->top++;
    s->items[s->top] = newitem;
  }
  count++;
}
```

### DESCRIPTION

- Implement stack

AMIT

# 1 WHAT IS Data structure

## 1.7 Stack

### CODE

```c
// Remove element from stack
void pop(st *s) {
  if (isempty(s)) {
    printf("\n STACK EMPTY \n");
  } else {
    printf("Item popped= %d", s->items[s->top]);
    s->top--;
  }
  count--;
  printf("\n");
}

// Print elements of stack
void printStack(st *s) {
  printf("Stack: ");
  for (int i = 0; i < count; i++) {
    printf("%d ", s->items[i]);
  }
  printf("\n");
}
```

### DESCRIPTION

- Implement stack

AMIT

# 1 WHAT IS Data structure

## 1.8 Queue

- A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

- Queue follows the **First In First Out (FIFO)** rule - the item that goes in first is the item that comes out first.

- **Basic Operations of Queue:**
  - **Enqueue**: Add an element to the end of the queue
  - **Dequeue**: Remove an element from the front of the queue
  - **IsEmpty**: Check if the queue is empty
  - **IsFull**: Check if the queue is full
  - **Peek**: Get the value of the front of the queue without removing it


FIFO Representation of Queue

AMIT

# 1 WHAT IS Data structure

## 1.8 Queue

- Working of Queue

Queue operations work as follows:

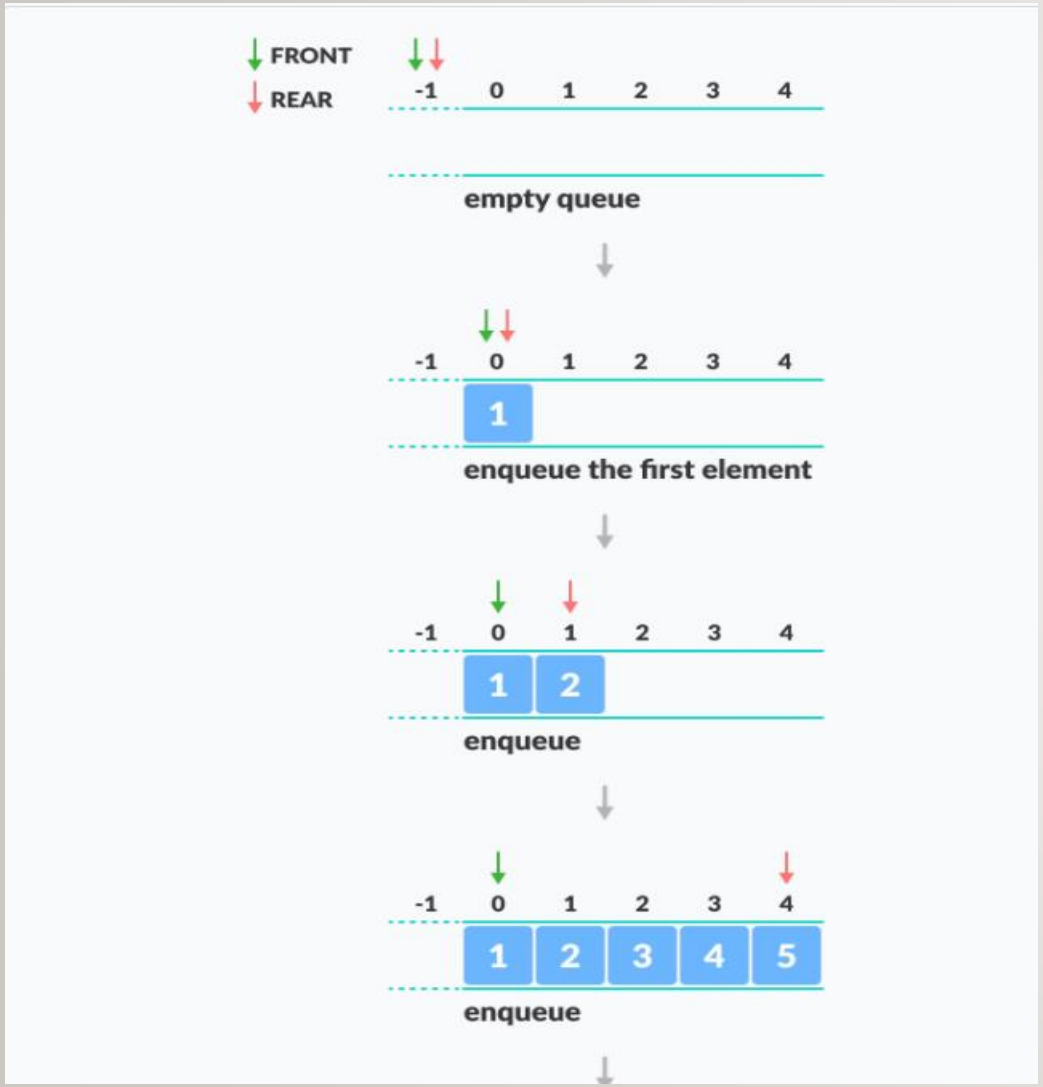
- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last element of the queue
- initially, set value of FRONT and REAR to -1

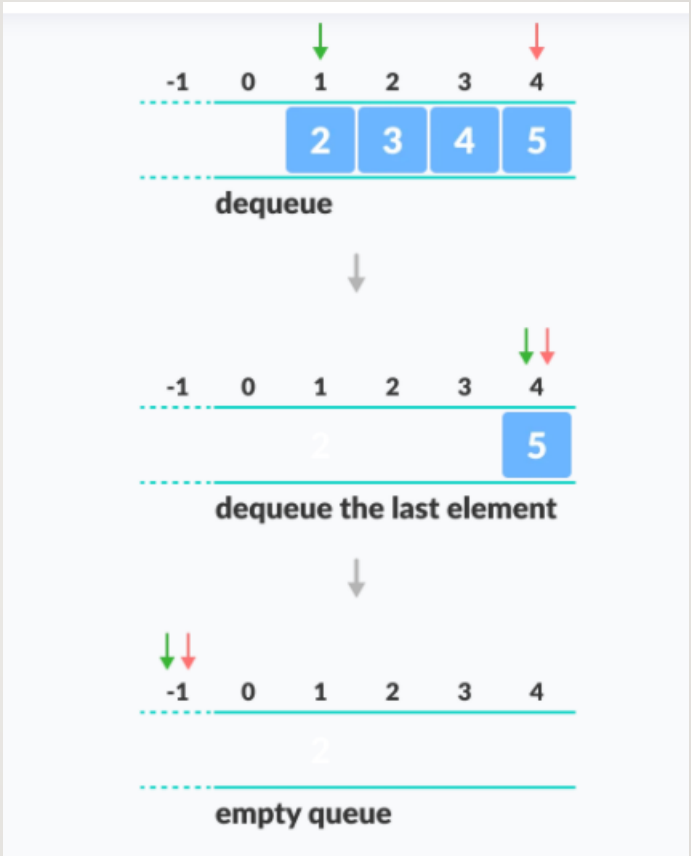
- Enqueue Operation:
  - check if the queue is full
  - for the first element, set the value of FRONT to 0
  - increase the REAR index by 1
  - add the new element in the position pointed to by REAR
- Dequeue Operation:
  - check if the queue is empty
  - return the value pointed by FRONT
  - increase the FRONT index by 1
  - for the last element, reset the values of FRONT and REAR to -1

AMIT

# 1 WHAT IS Data structure

## 1.8 Queue



# 1 WHAT IS Data structure

## 1.8 Stack

### CODE

```c
#define SIZE 5

void enQueue(int);
void deQueue();
void display();

int items[SIZE], front = -1, rear = -1;

int main() {
  //deQueue is not possible on empty queue
  deQueue();

  //enQueue 5 elements
  enQueue(1);
  enQueue(2);
  enQueue(3);
  enQueue(4);
  enQueue(5);

  // 6th element can't be added to because the queue is full
  enQueue(6);

  display();

  //deQueue removes element entered first i.e. 1
  deQueue();

  //Now we have just 4 elements
  display();
```

### DESCRIPTION

- Implement Queue

AMIT

# 1 WHAT IS Data structure

## 1.8 Stack

### CODE

```
void enQueue(int value) {
  if (rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else {
    if (front == -1)
      front = 0;
    rear++;
    items[rear] = value;
    printf("\nInserted -> %d", value);
  }
}

void deQueue() {
  if (front == -1)
    printf("\nQueue is Empty!!");
  else {
    printf("\nDeleted : %d", items[front]);
    front++;
    if (front > rear)
      front = rear = -1;
  }
}
```

### DESCRIPTION

- Implement Queue.

AMIT

# 1 WHAT IS Data structure

## 1.8 Stack

### CODE

```c
// Function to print the queue
void display() {
  if (rear == -1)
    printf("\nQueue is Empty!!!");
  else {
    int i;
    printf("\nQueue elements are:\n");
    for (i = front; i <= rear; i++)
      printf("%d  ", items[i]);
  }
  printf("\n");
}
```

### DESCRIPTION

- Implement Queue.