USN

CMR
INSTITUTE OF TECHNOLOGY

Internal Assessment Test II – April 2024

Sub:	Data Structures						Sub Code:		22MCA13
Date:	25/04/2023	Duration:	90 min's	Max Marks:	50	Sem:	I	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Part.

	PART I	MARKS	OBE	
			CO	RBT
1	Define Queue. Explain the different types of Queue, operations performed on the queue.	[10]	CO3	L2
	OR			
2	Design, develop, and execute a program in C to simulate the working of a linear queue of integers using an array. Provide the following operations: a. Insert b. Delete c. Display	[10]	CO2	L2
	PART II			
3	What is Circular Queue? Write a program to insert and delete an item from circular Queue.	[10]	CO3	L3
	OR			
4	What is Double ended Queues. Explain its type and operations performed on Double ended Queue.	[10]	CO3	L3
	PART III			
5	Explain Static (stack) and Dynamic (heap) memory allocation with neat diagram and also list the differences between static and dynamic memory allocation.	[10]	CO3	L3
	OR			
6	Explain different types of linked list. What are the advantages of linked list over arrays?	[10]	CO4	L3
	PART IV			
7	Explain the different functions used in C language for memory allocation and management with example program.	[10]	CO3	L2
	OR			
8	Explain the various real-time applications of Queue and Linked List. Also list and explain the various operations that can be performed on a linked list.	[10]	CO3	L2
	PART V			
9	Write a program to create a structure named Student. Allocate memory dynamically to this structure and input and print student details	[10]	CO3	L3
	OR			
10	What is Priority Queue? Write a program to simulate the working of priority queue.	[10]	CO2	L3

Solution

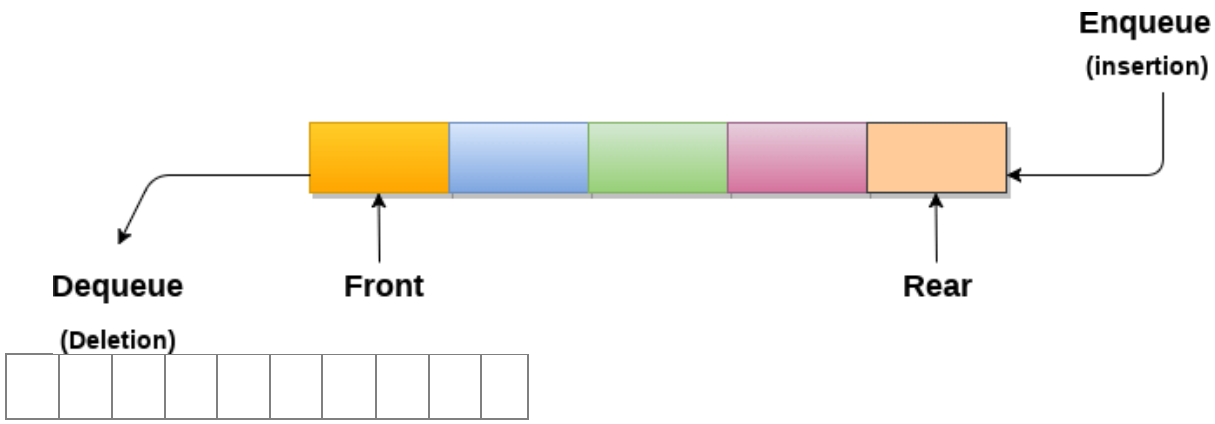
1. Define Queue. Explain the different types of Queue, operations performed on the queue and

Definition

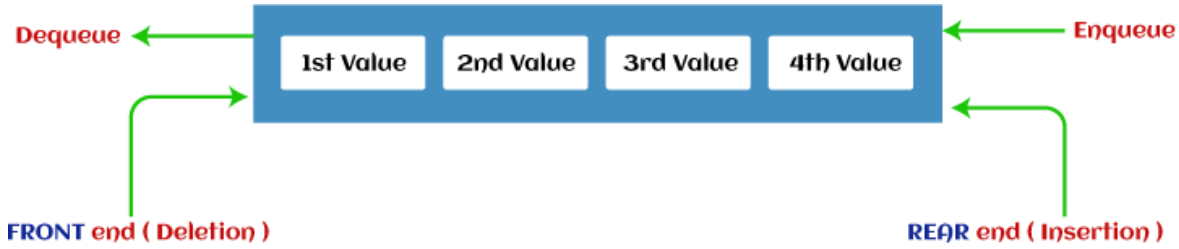
1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

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.



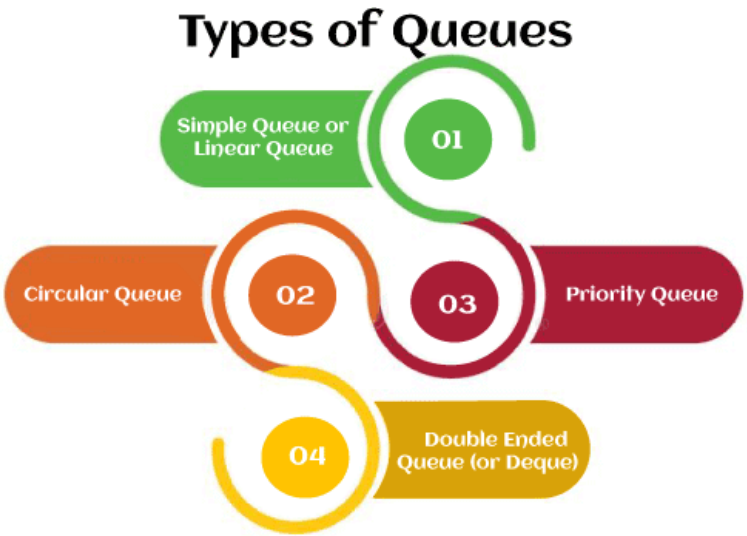
Representation: Queue is the data structure that is similar to the queue in the real world. A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Queue can also be defined as the list or collection in which the insertion is done from one end known as the **rear end** or the **tail** of the queue, whereas the deletion is done from another end known as the **front end** or the **head** of the queue.
The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the last person enters in the queue gets the ticket at last. Similar approach is followed in the queue in data structure.



Now, let's move towards the types of queue.

Types of Queue

There are four different types of queue that are listed as follows -



- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

- **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.

- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
- **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

Simple Queue or Linear Queue

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.


```
void enqueue()
{
```

```
  if(rear == MAX-1) printf("\nQueue
  is full."); else
  {
    printf("\n\nEnter ITEM:");
    scanf("%d", &item);
    if (rear == -1 && front == -1)
    {
      rear = 0;
      front = 0;
    }
    else
      rear++; queue[rear] =
   item;
   printf("\n\nItem inserted: %d", item);
  }
}
void dequeue()
{
  if(front == -1) printf("\n\nQueue is
  empty."); else
  {
    item = queue[front]; if
    (front == rear)
    {
      front = -1;
      rear = -1;
    }
    else
      front++;
    printf("\n\nItem deleted: %d", item);
  }
}
```

2. Queue Implementation using array (Linear Queue)

```
#include <stdio.h>
#include<stdlib.h>
#define MAX 6 void
enqueue(); void
dequeue(); void
display();
int queue[MAX], rear=-1, front=-1, item;

void main()

{

  int ch;
  while(1)
  {

printf("\n\n1. Insert\n2. Delete\n3. Display\n4. Exit\n"); printf("\nEnter
your choice:");
scanf("%d", &ch);
switch(ch)
{

case 1:

enqueue();
break; case 2:
dequeue();
break; case 3:
display();
break; case
4:
exit(0);
default:
printf("\n\nInvalid entry. Please try again...\n");

}
  }

}
void enqueue()

{
```

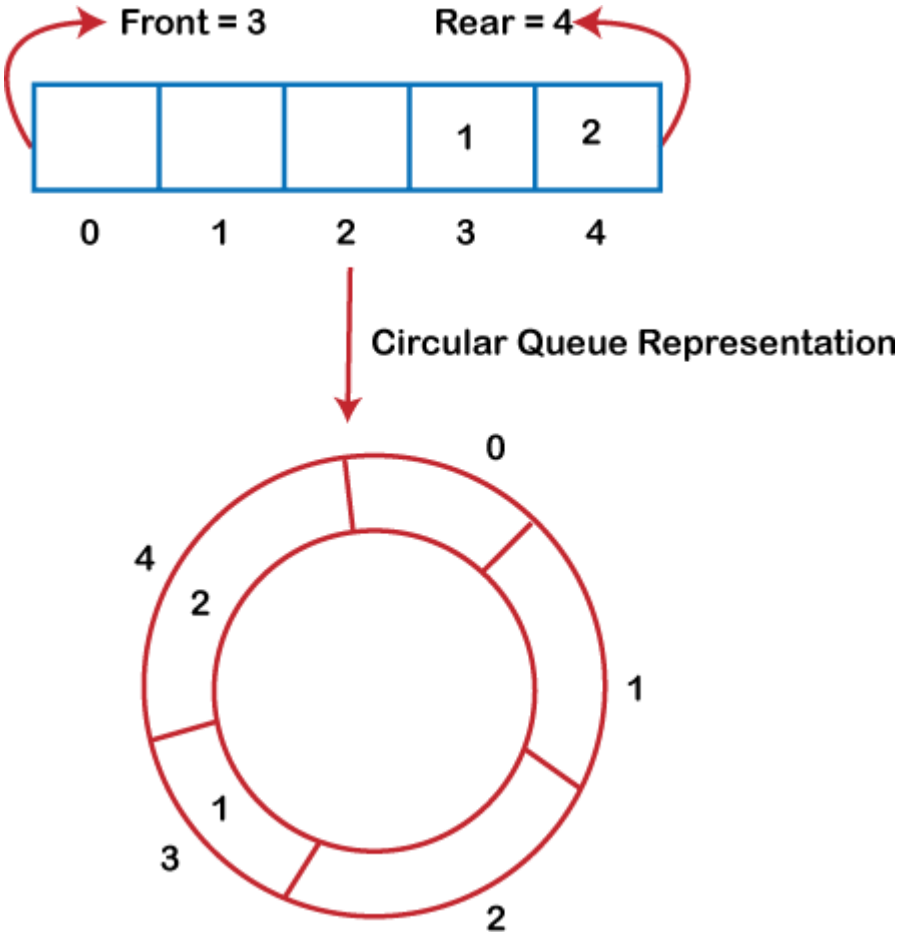
```
  if(rear == MAX-1) printf("\nQueue
  is full."); else
  {
    printf("\n\nEnter ITEM:");
    scanf("%d", &item);
    if (rear == -1 && front == -1)
    {
      rear = 0;
      front = 0;
    }
    else
      rear++; queue[rear] =
   item;
   printf("\n\nItem inserted: %d", item);
  }
}
void dequeue()
{
  if(front == -1) printf("\n\nQueue is
  empty."); else
  {
    item = queue[front]; if
    (front == rear)
    {
      front = -1;
      rear = -1;
    }
    else
      front++;
    printf("\n\nItem deleted: %d", item);
  }
}
void display()
{
  int i;
```

```
  if(front == -1) printf("\n\nQueue is
  empty."); else

  {
    printf("\n\n"); for(i=front;
    i<=rear; i++) printf( "%d",
    queue[i]);
  }
}
```

2. What is Circular Queue? Write a program to insert and delete an item from circular Queue.

Circular Queue:

There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0^{th} position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue:

Implementation of Circular Queue using Array

```
#include <stdio.h>
#include<stdlib.h>
#define MAX 6
void enqueue();
void dequeue();
void display();
int queue[MAX], rear=-1, front=-1, item;


void main()
{
  int ch;
  while(1)
  {
printf("\n\n1. Insert\n2. Delete\n3. Display\n4. Exit\n");
printf("\nEnter your choice:");
scanf("%d", &ch);
switch(ch)
{
case 1:
enqueue()
; break;
case 2:
dequeue()
; break;
case 3:
display();
break;
```

```
case 4:
exit(0);
default:
printf("\n\nInvalid entry. Please try again...\n");
}
  }
}
void enqueue()
{
  if((rear+1)%MAX == front)
  printf("\nQueue is full.");
  else
  {
    printf("\n\nEnter ITEM:");
    scanf("%d", &item);
    if (rear == -1 && front == -1)
    {
      rear = 0;
      front = 0;
    }
    else
 rear=(rear+1)%MAX;
 queue[rear] = item;
   printf("\n\nItem inserted: %d", item);
  }
}
void dequeue()
{
  if(front == -1)
  printf("\n\nQueue is empty.");
  else
  {
    item = queue[front];
    if (front == rear)
    {
      front = -1;
      rear = -1;
    }
    else
      front=(front+1)%MAX;
    printf("\n\nItem deleted: %d", item);
```

```
  }
}
```

```
void display()
{
  int i=front;
  if(front == -1 && rear==-1)
  printf("\n\nQueue is empty.");
  else
 {
    printf("\n\n queue is ");
    while(i!=rear)
    {
    printf( "%d", queue[i]);
     i=(i+1)%MAX;
    }
  printf("%d",queue[rear]);
}
}
```

4. What is Double ended Queues. Explain its type and operations performed on Double ended Queue.

Deque(or double-ended queue)

In this article, we will discuss the double-ended queue or deque. We should first see a brief description of the queue.

What is a queue?

A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Insertion in the queue is done from one end known as the **rear end** or the **tail,** whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the person enters last in the queue gets the ticket at last.

What is a DEQue (or double-ended queue?)

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Representation of deque

Operations performed on deque

There are the following operations that can be applied on a deque –

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque –

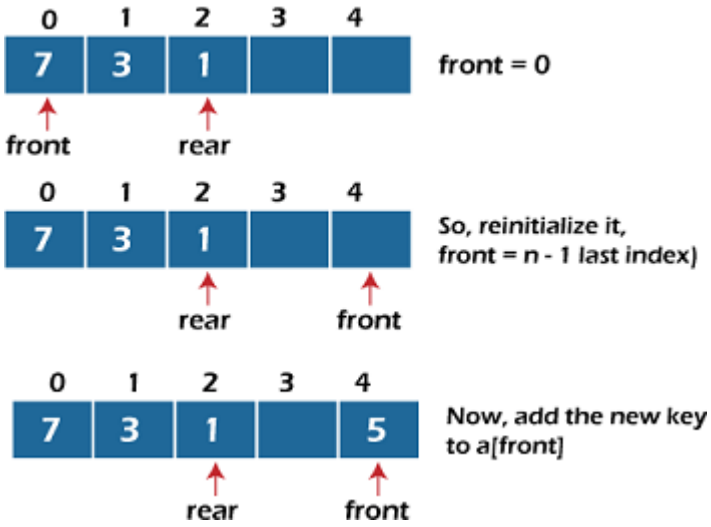
- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions –

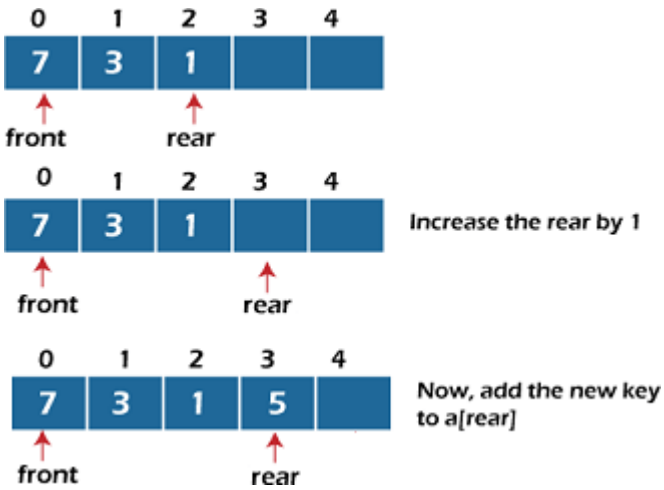
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n – 1**, i.e., the last index of the array.



Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions –

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size – 1), then instead of increasing it by 1, we have to make it equal to 0.



Deletion at the front end

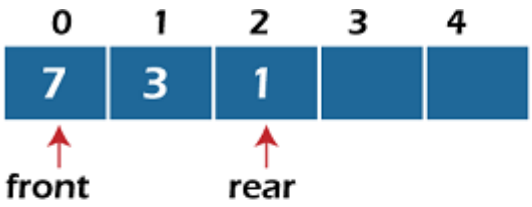
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

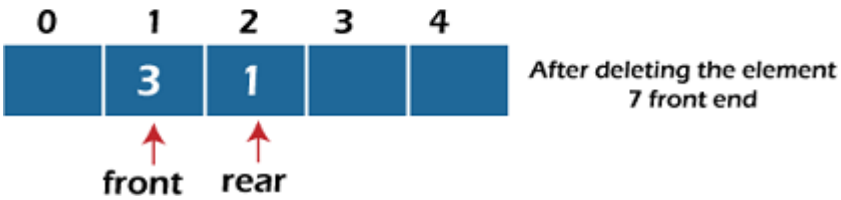
If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions

If the deque has only one element, set rear = -1 and front = -1. Else

if front is at end (that means front = size – 1), set front = 0. Else

increment the front by 1, (i.e., front = front + 1).





Deletion at the rear end

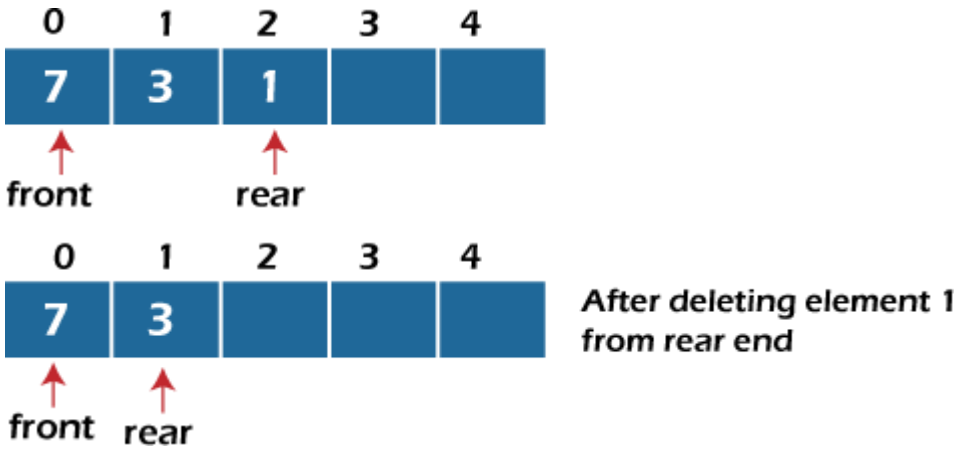
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1. If

rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).



Check empty

This operation is performed to check whether the deque is empty or not. If front = –1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n – 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

5. Explain Static (stack) and Dynamic (heap) memory allocation with neat diagram and also list the differences between static and dynamic memory allocation

Memory Allocation: Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space. The memory allocation is done either before or at the time of program execution. There are two types of memory allocations:

1. Compile-time or Static Memory Allocation
2. Run-time or Dynamic Memory Allocation

Static Memory Allocation:

Static Memory is allocated for declared variables by the compiler. The address can be found using the *address of* operator and can be assigned to a pointer. The memory is allocated during compile time.

Dynamic Memory Allocation:

Memory allocation done at the time of execution (run time) is known as dynamic memory allocation. Functions calloc() and malloc() support allocating dynamic memory. In the Dynamic allocation of memory space is allocated by using these functions when the value is returned by functions and assigned to pointer variables.

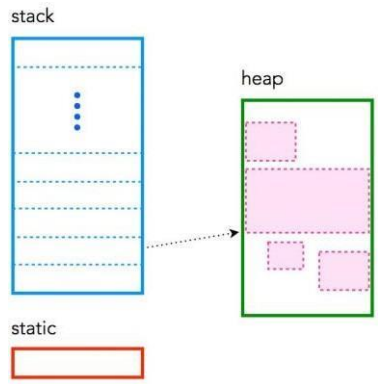
Memory in C – the stack, the heap, and static

C has three different pools of memory.

– **static**: global variable storage, permanent for the entire run of the program.

– **stack**: local variable storage (automatic, continuous memory).

– **heap**: dynamic storage (large pool of memory, not allocated in contiguous order).



Static memory and Stack (Static Memory Allocation)

Static memory persists throughout the entire life of the program, and is usually used to store things like *global* variables, or variables created with the **static** clause. For example:

int theforce;

On many systems this variable uses 4 bytes of memory. This memory can come from one of two places. If a variable is declared *outside* of a function, it is considered global, meaning it is accessible anywhere in the program. Global variables are **static,** and there is only one copy for the entire program. Inside a function the variable is allocated on the stack. It is also possible to force a variable to be static using the **static** clause. For example, the same variable created inside a function using the **static** clause would allow it to be stored in static memory.

static int theforce;

Stack memory

The *stack* is used to store variables used on the inside of a function (including the **main()** function). It's a LIFO, "**L**ast-**I**n,-**F**irst-**O**ut", structure. Every time a function declares a new variable it is "pushed" onto the stack. Then when a function finishes running, all the variables associated with that function on the stack are deleted, and the memory they use is freed up. This leads to the "local" scope of function variables. The stack is a special region of memory, and automatically managed by the CPU – so you don't have to allocate or deallocate memory. Stack memory is divided into successive frames where each time a function is called, it allocates itself a fresh stack frame.

Note that there is generally a limit on the size of the stack – which can vary with the operating system (for example OSX currently has a default stack size of 8MB). If a program tries to put too much information on the stack, **stack overflow** will occur. Stack overflow happens when all the memory in the stack has been allocated, and further allocations begin overflowing into other sections of memory. Stack overflow also occurs in situations where recursion is incorrectly used.

A summary of the stack:

- the stack is managed by the CPU, there is no ability to modify it
- variables are allocated and freed automatically
- the stack it not limitless – most have an upper bound
- the stack grows and shrinks as variables are created and destroyed
- stack variables only exist whilst the function that created them exists

Heap memory (Dynamic Memory Allocation)

The *heap* is the diametrical opposite of the stack. The *heap* is a large pool of memory that can be used dynamically – it is also known as the "free store". This is memory that is not automatically managed – you have to explicitly allocate (using functions such as malloc), and deallocate (e.g. free) the memory. Failure to free the memory when you are finished with it will result in what is known as a *memory leak* – memory that is still "being used", and not available to other processes. Unlike the stack, there are generally no restrictions on the size of the heap (or the variables it creates), other than the physical size of memory in the machine. Variables created on the heap are accessible anywhere in the program.

Oh, and heap memory requires you to use **pointers**.

A summary of the heap:

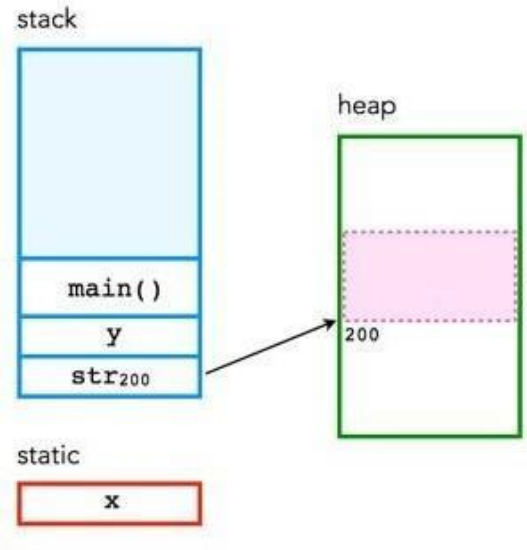
- the heap is managed by the programmer, the ability to modify it is somewhat boundless
- in C, variables are allocated and freed using functions like malloc() and free()
- the heap is large, and is usually limited by the physical memory available
- the heap requires pointers to access it

An example of memory use

Consider the following example of a program containing all three forms of memory:

```
#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;
}
```

The variable **x** is static storage, because of its global nature. Both **y** and **str** are dynamic stack storage which is deallocated when the program ends. The function **malloc()** is used to allocate 100 pieces of of dynamic heap storage, each the size of char, to **str**. Conversely, the function **free()**, deallocates the memory associated with **str**.



Tabular Difference between Static and Dynamic Memory Allocation in C:

S.No	Static Memory Allocation	Dynamic Memory Allocation
1	In the static memory allocation, variables get allocated permanently, till the program executes or function call finishes.	In the Dynamic memory allocation, variables get allocated only if your program unit gets active.
2	Static Memory Allocation is done before program execution.	Dynamic Memory Allocation is done during program execution.
3	It uses stack for managing the static allocation of memory	It uses heap for managing the dynamic allocation of memory
4	It is less efficient	It is more efficient
5	In Static Memory Allocation, there is no memory re-usability	In Dynamic Memory Allocation, there is memory re-usability and memory can be freed when not required
6	In static memory allocation, once the memory is allocated, the memory size can not change.	In dynamic memory allocation, when memory is allocated the memory size can be changed.
7	In this memory allocation scheme, we cannot reuse the unused memory.	This allows reusing the memory. The user can allocate more memory when required. Also, the user can release

		the memory when the user needs it.
8	In this memory allocation scheme, execution is faster than dynamic memory allocation.	In this memory allocation scheme, execution is slower than static memory allocation.
9	In this memory is allocated at compile time.	In this memory is allocated at run time.
10	In this allocated memory remains from start to end of the program.	In this allocated memory can be released at any time during the program.
11	Example: This static memory allocation is generally used for array.	Example: This dynamic memory allocation is generally used for linked list.
12	Example: inti; float f;	p = malloc(sizeof(int));

6. Explain different types of linked list. What are the advantages of linked list over arrays?

Types of Linked list

Linked list is classified into the following types –

- **Singly-linked list -** Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists 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.
- **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), and pointer to the previous node (previous pointer).
- **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.

- **Circular doubly linked list –** Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

Advantages of Linked list

The advantages of using the Linked list are given as follows –

- **Dynamic data structure –** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
- **Insertion and deletion –** Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.
- **Memory efficient –** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- **Implementation –** We can implement both stacks and queues using linked list.

7. Explain the different functions used in C language for memory allocation and management with example program.

The C programming language provides several functions for memory allocation and management. These functions can be found in the <stdlib.h> header file.

Sr.No.	Function & Description
1	void *calloc(int num, int size); This function allocates an array of num elements each of which size in bytes will be size.
2	void free(void *address); This function releases a block of memory block specified by address.
3	void *malloc(size_t size); This function allocates an array of num bytes and leave them uninitialized.
4	void *realloc(void *address, int newsize); This function re-allocates memory extending it upto newsize.

Allocating Memory Dynamically

While programming, if you are aware of the size of an array, then it is easy and you can define it as an array. For example, to store a name of any person, it can go up to a maximum of 100 characters, so you can define something as follows −

```
char name[100];
```

But now let us consider a situation where you have no idea about the length of the text you need to store, for example, you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later, based on requirement, we can allocate memory as shown in the below example −

```
#include<stdio.h
>
#include<stdlib.h
>#i
nclude<string.h>i
nt main(){ char
name[100];
char*description;
strcpy(name,"Raj
");
/* allocate memory dynamically */ description=malloc(200*sizeof(char));

if( description == NULL ){
fprintf(stderr,"Error - unable to allocate required memory\n");
}else{
strcpy( description,"Raj welcome to CMRIT");
}

printf("Name = %s\n", name );
```

When the above code is compiled and executed, it produces the following result.

Name = Raj
Description: Raj Welcome to CMRIT

Same program can be written using calloc(); only thing is you need to replace malloc with calloc as follows −

```
calloc(200,sizeof(char));
```

So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size defined, you cannot change it.

Resizing and Releasing Memory

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function free().

Alternatively, you can increase or decrease the size of an allocated memory block by calling the function realloc().

8. **Explain the various real-time applications of Queue and Linked List. Also list and explain the various operations that can be performed on a linked list.**

Real-Time Applications of Queue

A queue is a linear data structure that follows the First In First Out (FIFO) principle. Here are some real-time applications where queues are prominently used:

Job Scheduling: Operating systems use queues for job scheduling, managing processes that are scheduled to run on the CPU, ordered by time or priority.

Handling Requests: In web servers, queues are used to manage incoming HTTP requests. They help in handling requests in the order they arrive and balancing load.

Data Buffering: Queues are used as buffers in applications like media streaming or data transfer between processes. They manage data packets that need to be processed sequentially.

Print Queue Management: Printers use queues to manage print jobs, ensuring documents are printed in the order

they were sent to the printer.

Call Center Systems: Incoming calls are held in a queue until they can be directed to the next available customer service representative.

Traffic Management: Queue data structures are used in traffic management systems to regulate the flow of traffic and prioritize vehicle movements at intersections.

Real-Time Applications of Linked List

A linked list is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer. Here are some applications:

1. **Dynamic Memory Allocation**: Linked lists are used to implement dynamic memory allocation where the size of the data structure can grow and shrink during runtime.
2. **Implementation of Stacks and Queues**: Linked lists provide a flexible way to implement other data structures like stacks and queues.
3. **Undo Functionality in Applications**: They can be used to implement undo functionality in applications by keeping a list of operations performed.
4. **Image Viewer Applications**: Linked lists can be used to implement features in image viewer applications where users can go to the next or previous image or file.
5. **Music Playlists**: Music player applications use linked lists to manage playlists, where users can easily add, remove, and skip tracks.

Operations on a Linked List

Linked lists support a variety of operations that allow for flexible data management. Here are the primary operations that can be performed on a linked list:

1. **Insertion**:
   - **At the front**: Adds a new node at the beginning of the list.
   - **At the end**: Adds a new node after the last node of the list.
   - **After a given node**: Inserts a new node after a specified node.
2. **Deletion**:
   - **At the front**: Removes the first node of the list.
   - **At the end**: Removes the last node of the list.
   - **By value**: Removes a node containing a specific value.
3. **Search**: Finds a node in the list containing a given value, often returning the node itself or a boolean indicating presence.
4. **Traversal**: Access each element of the linked list. This is typically done starting from the head and moving through each node until the end of the list is reached.
5. **Reverse**: Reverses the order of nodes in the list. This can be done iteratively or recursively.
6. **Sort**: Orders the nodes in the list, usually according to numerical or lexicographical order.
7. **Update**: Modifies the data of one or more nodes in the list.

9. Write a program to create a structure named Student. Allocate memory dynamically to this structure and input and print student details.

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

// Define the structure for Student
typedef struct {
    char name[100];
    int age;
    float gpa;
} Student;

// Function to input student details
void inputStudentDetails(Student *s) {
    printf("Enter student name: ");
```

```
    fgets(s->name, sizeof(s->name), stdin); // Using fgets to include spaces
    printf("Enter student age: ");
    scanf("%d", &s->age);
    printf("Enter student GPA: ");
    scanf("%f", &s->gpa);
}

// Function to print student details
void printStudentDetails(Student *s) {
    printf("\nStudent Details:\n");
    printf("Name: %s", s->name); // fgets includes the newline character
    printf("Age: %d\n", s->age);
    printf("GPA: %.2f\n", s->gpa);
}

int main() {
    Student *student;

    // Dynamically allocate memory for one Student structure
    student = (Student *)malloc(sizeof(Student));
    if (student == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1; // Exit the program with an error code
    }

    // Flush stdin to clear any leftover characters
    fflush(stdin);

    // Input student details
    inputStudentDetails(student);

    // Print student details
    printStudentDetails(student);

    // Free the allocated memory
    free(student);

    return 0; // Successful execution
}
```

10. What is Priority Queue? Write a program to simulate the working of priority queue.

A priority queue is a type of data structure in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue. Typically, the priority queue is implemented using heaps, but it can also be implemented using arrays or linked lists.

```
#include <stdio.h>
#define size 5
int deque[size]; 4.
int f = -1, r = -1;
// insert_front function will insert the value from the front
void insert_front(int x) 7.
{
        if((f==0 && r==size-1) || (f==r+1))
        {
printf("Overflow");
}
else if((f==-1) && (r==-1))
{
f=r=0;

deque[f]=x;
}
```

```
else if(f==0)
{
f=size-1;
deque[f]=x;
}
else
{
f=f-1;
deque[f]=x;
}
}
28
.
// insert_rear function will insert the value from the rear
void insert_rear(int x)
{
if((f==0 && r==size-1) || (f==r+1))
{
printf("Overflow");
}
else if((f==-1) && (r==-1))
{
r=0;
deque[r]=x;
}
else if(r==size-1)
{
r=0;
deque[r]=x;
}
else
{
r++;

deque[r]=x;
}
}
// display function prints all the value of deque.
void display()
{
int i=f;
printf("\nElements in a deque are: "); 59.
            while(i!=r)
            {
                printf("%d ",deque[i]);
                i=(i+1)%size;
```

```
        }
         printf("%d",deque[r]);
 }
// getfront function retrieves the first value of the deque.
void getfront()
{
        if((f==-1) && (r==-1))
        {
            printf("Deque is empty");
        }
        else
        {
printf("\nThe value of the element at front is: %d", deque[f]);
        }
}
```

```
void getrear()
{
          if((f==-1) && (r==-1))
          {
              printf("Deque is empty");
          }
          else
          {
              printf("\nThe value of the element at rear is %d", deque[r]);
          }
}
// delete_front() function deletes the element from the front
void delete_front()
{
         if((f==-1) && (r==-1))
         {
         printf("Deque is empty");
         }
         else if(f==r)
         {
                        printf("\nThe deleted element is %d", deque[f]);
         f=-1;
         r=-1;

         }
         else if(f==(size-1))
         {
                         printf("\nThe deleted element is %d", deque[f]);
         f=0;
         }
         else
         {
```

```
            printf("\nThe deleted element is %d", deque[f]);
            f=f+1;
          }
}

// delete_rear() function deletes the element from the rear
void delete_rear()
{
if((f==-1) && (r==-1))
{
printf("Deque is empty");
}
else if(f==r)
{
          printf("\nThe deleted element is %d", deque[r]);
f=-1;
r=-1;

}
else if(r==0)
{
          printf("\nThe deleted element is %d", deque[r]);
r=size-1;
}
else
{
           printf("\nThe deleted element is %d", deque[r]);
r=r-1;
}
}

int main()
{
insert_front(20);
```

```
insert_front(10);
insert_rear(30);
insert_rear(50);
insert_rear(80);
display(); // Calling the display function to retrieve the values of deque
getfront(); // Retrieve the value at front-end
getrear(); // Retrieve the value at rear-end
delete_front();
delete_rear();
display(); // calling display function to retrieve values after deletion
return 0;
    }
```