CS 2124: DATA STRUCTURES
Spring 2024

Fifth Lecture

Topics: **Queues,** Linked Lists

Topics

- Queues
- Queue (Operations)
  - peek()
  - isfull()
  - isempty()
  - Enqueue ()
  - Dequeue ()
- Queue Implementation
  - Array
  - Switch
  - Pointers
- Types of Queues
- Application of Queue
- Advantages and Disadvantages of Queues
- Issues in the applications of Queue

Assignment

- How canvas checks for plagiarism ?
- Using AI based tools or google
- Adding descriptions about your program
  - In your PDF files (under the screenshot output)
- Assignment submission time frame
  - 1- week
- Preparing for Quiz and Exam

Queue

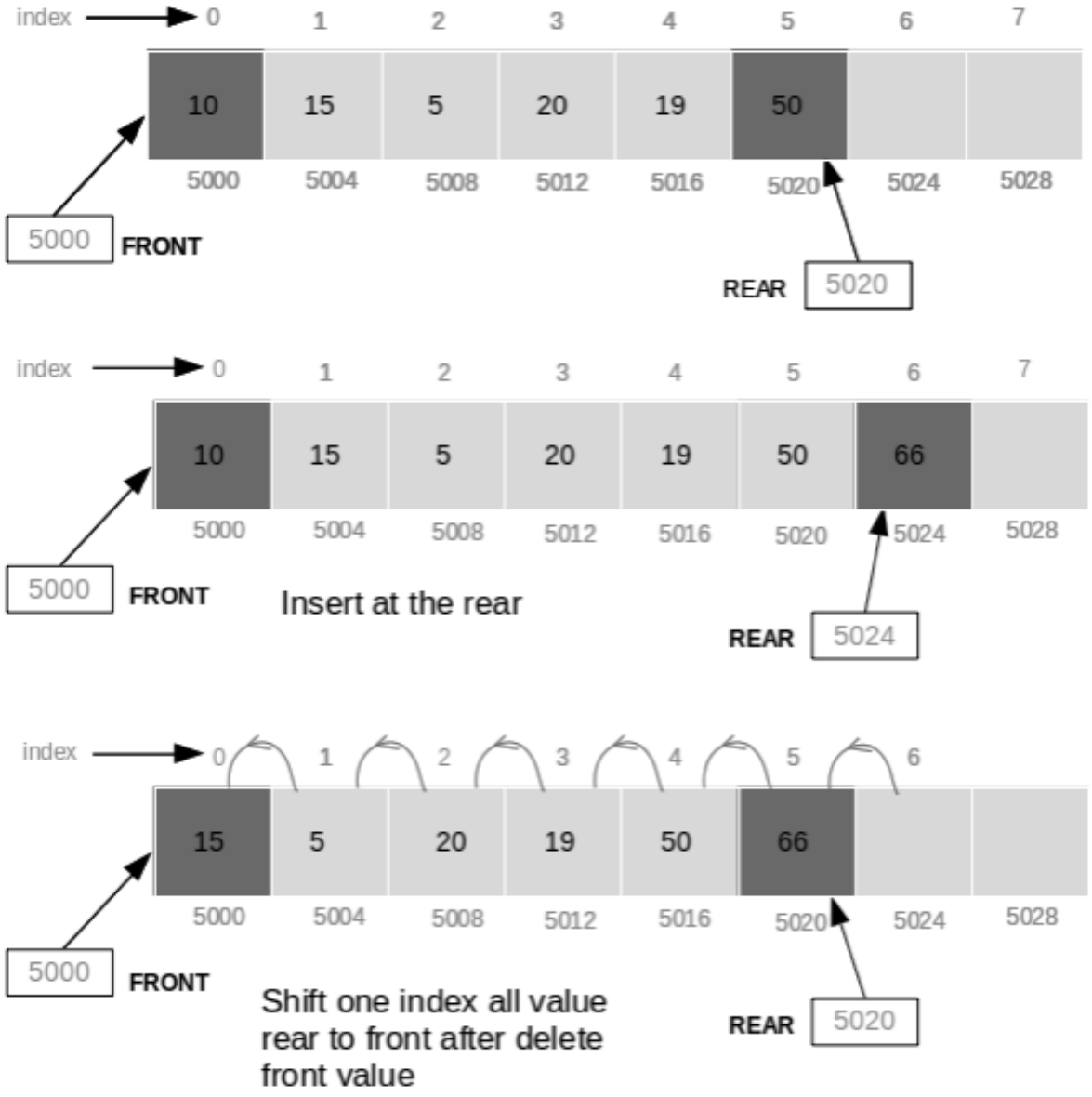
- Queue is an abstract data structure, somewhat similar to Stacks.
- Unlike stacks, a queue is open at both its ends.
- One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).
- Queue follows First-In-First-Out (FIFO) methodology, i.e., the data item stored first will be accessed/exit first.

Queue

- As in stacks, a queue can also be implemented using:
  - Arrays
  - Linked-lists
  - Pointers
  - Structures.

- For the sake of simplicity, we shall implement queues using one-dimensional array without using pointers (initially).

- Later, we will see other implementations too.



Queue (Operations)

- Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. The basic operations associated with queues are
  - **enqueue()** − add (store) an item to the queue.
  - **dequeue()** − remove (access) an item from the queue.

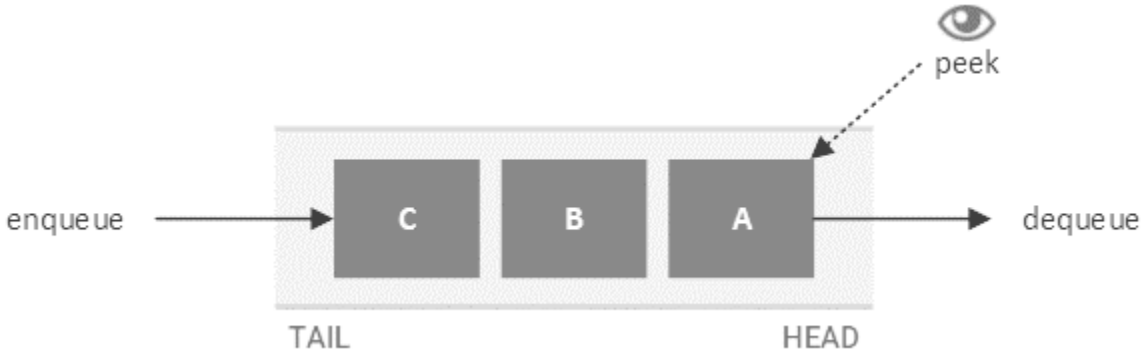
- Few more functions are required to make the above-mentioned queue operation efficient. These are
  - **peek()** − Gets the element at the front of the queue without removing it.
  - **isfull()** − Checks if the queue is full.
  - **isempty()** − Checks if the queue is empty.

Queue (Operations)

peek()

- This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows:

- Algorithm
  1. begin procedure peek
  2. return queue[front]
  3. end procedure



Question: What is wrong in this algorithm?

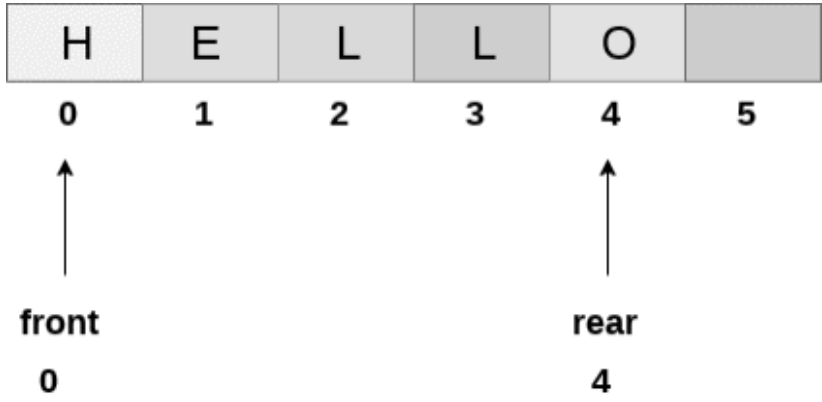
- *We did not checked if Queue is empty!*

Queue (Operations)

isfull()

- To check if Queue is full we can use two approaches:
  1. Check if Queue size equals MAXSIZE
  2. (In case of Pointer) Check for the rear pointer to reach at MAXSIZE to determine that the queue is full.

- **Algorithm**
  1. begin procedure isfull
  2. if rear equals to MAXSIZE
  3. return true
  4. else
  5. return false
  6. endif
  7. end procedure



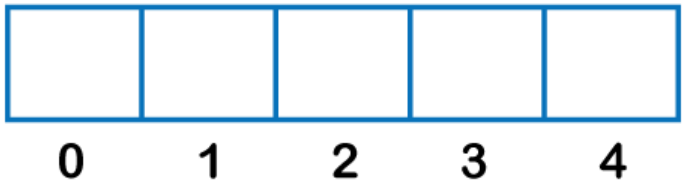
Queue MAXSIZE = 6

Queue (Operations)

isempty()

- If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

- Algorithm
  1. begin procedure isempty
  2. if front is less than MIN OR front is greater than rear OR front is equal to rear
  3. return true
  4. else
  5. return false
  6. endif
  7. end procedure



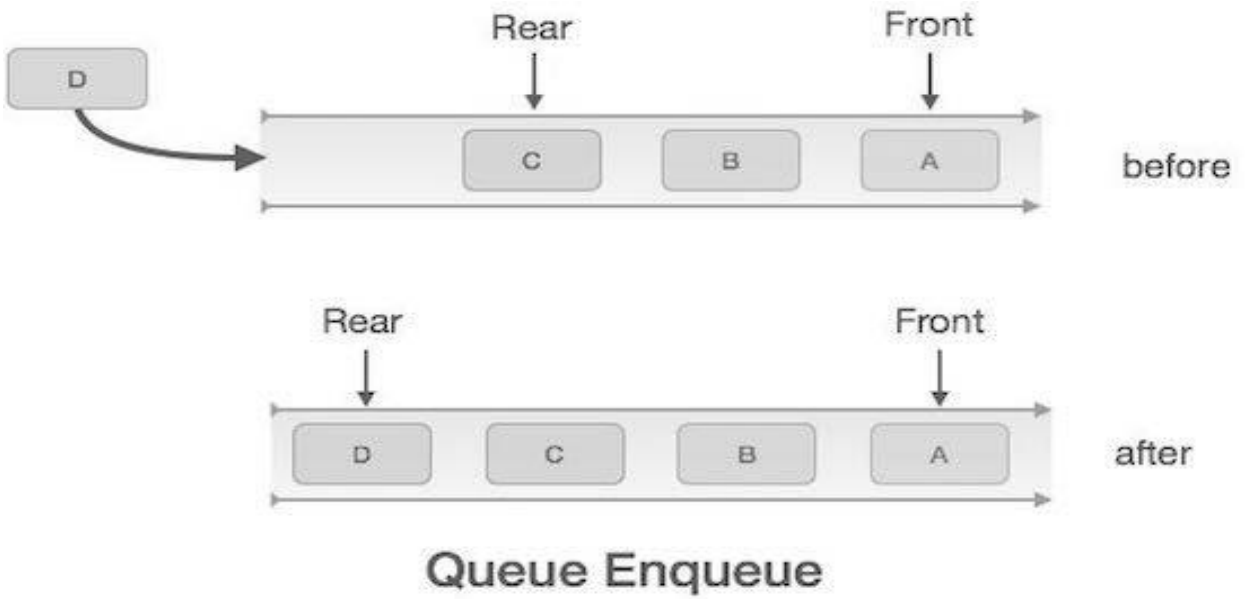
Front = -1

Rear = -1

Question: which is the best option to check if queue isempty()?

- Check if front == rear

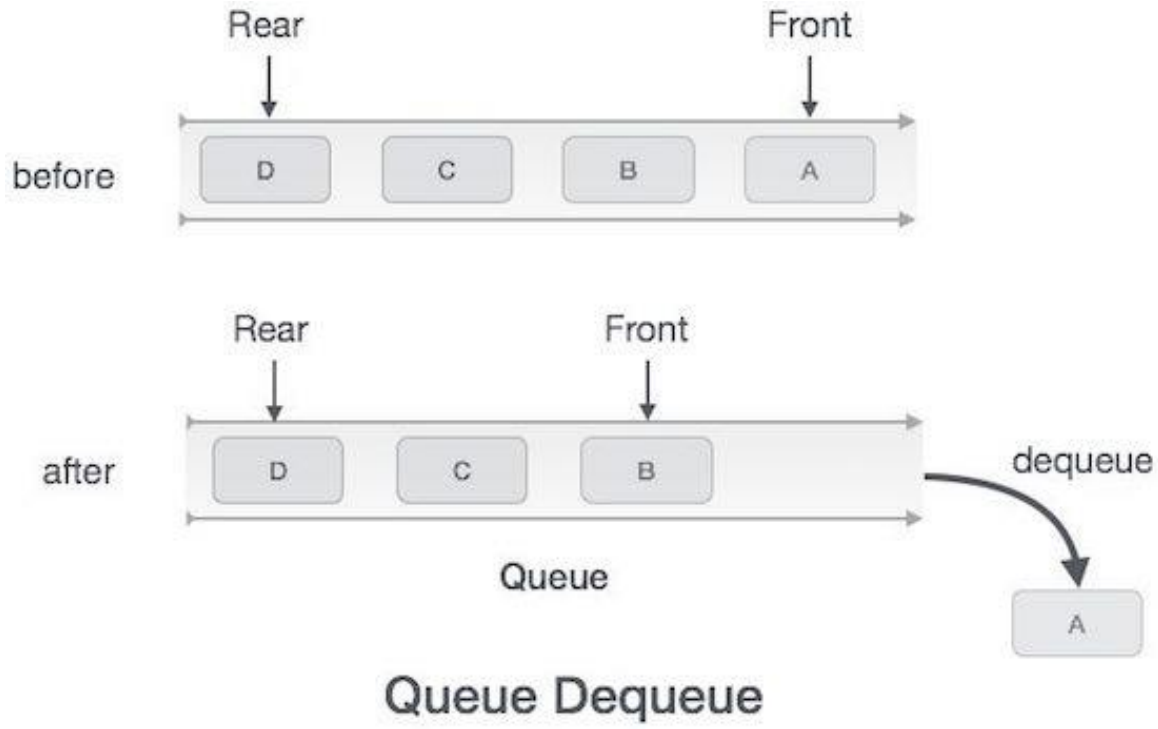
Queue (Operations)



- **Enqueue**
- Queues maintain two data pointers or index values for keeping track of front and rear.

- The following steps should be taken to enqueue (insert) data into a queue −
  - **Step 1** − Check if the queue is full.
  - **Step 2** − If the queue is full, produce overflow error and exit.
  - **Step 3** − If the queue is not full, increment rear pointer to point the next empty space.
  - **Step 4** − Add data element to the queue location, where the rear is pointing.
  - **Step 5** − return success.

Queue (Operations)



- **Dequeue**
- This process involved removing the data in the front of the queue and then updating the value of the front pointer or variable.
- Algorithm:
  - **Step 1** – Check if the queue is empty.
  - **Step 2** – If the queue is empty, produce underflow error and exit.
  - **Step 3** – If the queue is not empty, access the data where **front** is pointing.
  - **Step 4** – Increment **front** pointer to point to the next available data element.
  - **Step 5** – Return success.

Question: Can this idea be improved?

- Maybe after dequeue mover all elements forward to avoid empty space

Queue

- **Use memory safe programming languages:**
  - The most common reason why buffer overflow attacks work is because applications fail to manage memory allocations and validate input from the client or other processes.
  - Applications developed in C or C++ should avoid dangerous standard library functions that are not bounds-checked, such as `gets`, `scanf` and `strcpy`.
  - Instead, they should use libraries or classes that were designed to securely perform string and other memory operations

- **Validate data.**
  - Mobile and web applications developed in-house should always validate any user input and data from untrusted sources to ensure they are within the bounds of what is expected and to prevent overly long input values.

Buffer Overflow Attack on iOS

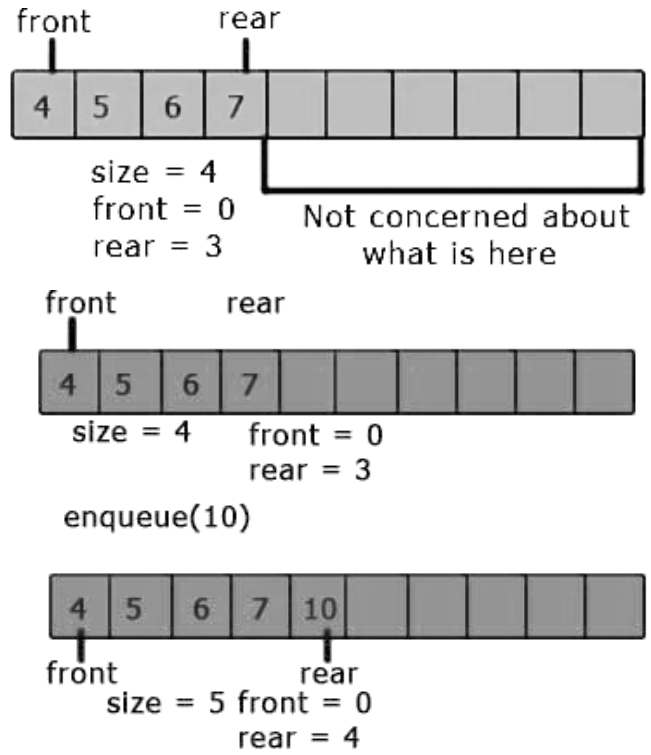
- In January 2021, Apple released a new iOS version including patches for not one, but three newly identified vulnerabilities: two in Safari's browser engine and one in the kernel, the core of iPads and iPhones operating system.
- All of them were being actively exploited by cybercriminals giving them full access to data and devices.

- Hackers used a buffer overflow attack.
- Example: Say, your organization's website uses OpenSSL, an open-source library to implement the SSL and transport layer security (TLS) protocols. Unfortunately, you're leaving the door open to hackers if you forget to patch OpenSSL against the Heartbleed bug (CVE-2014-0160).

Source: Link & Link

Queue

- #include <stdio.h>
- #define MAXSIZE 10
- int queue[MAXSIZE];
- int front = -1;
- int rear = -1;
- int size = -1;



We can use three pointers/variables to implement the queue using an array, i.e. ‘size’, ‘front’ and ‘rear’.

1. The ‘front’ and ‘rear’ will simply store the front and rear elements respectively.
2. While ‘size’ is use to store the current size of the queue.

The values of ‘front’, ‘rear’ and ‘size’ is -1 because the queue is not yet initialized.

enqueue

We just need to add an element at the index ‘rear+1’ and increase the value of the ‘rear’ and size by 1 for the enqueue operation.

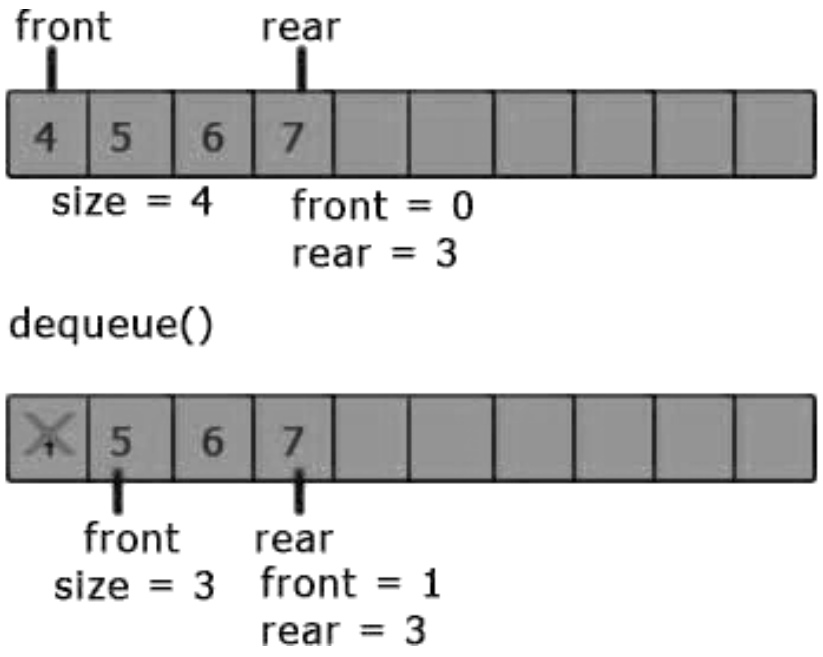
Enqueue

```
void enqueue(int value)
{
    if(size<MAXSIZE)
    {
        if(size<0)
        {
            queue[0] = value;
            front = rear = 0;
            size = 1;
        }
        else if(rear == MAXSIZE-1)
        {
            queue[0] = value;
            rear = 0;
            size++;
        }
        else
        {
            queue[rear+1] = value;
            rear++;
            size++;
        }
    }
    else
    {
        printf("Queue is full\n");
    }
}
```

- if(size<MAXSIZE): We are just making sure that the queue is not full
- if(size<0): The list is not initialized yet. We will initialize the list by making the both ‘front’ and ‘rear’ 0 and ‘size’ 1 and then we will give the value to the first element of the list.
- queue[0] = value; (i.e. int value = <int>)
- front = rear = 0;
- size = 1;
- else if(rear == MAXSIZE-1): This is the case when the ‘rear’ is the last element of the array. We will add a new element at the index 0.
- queue[0] = value;
- rear = 0;
- size++;
- else
- {
- queue[rear+1] = value;
- rear++;
- size++;
- }
- This is the normal case and we have coded accordingly. We are giving the value to the element with index ‘rear+1’ and then increasing both ‘rear’ and ‘size’ by 1.

Dequeue

- First check if the Queue is empty. After that the ‘dequeue’ operation is very simple using the array. We just need to decrease the ‘size’ by 1 and increase the ‘front’ by 1. That’s it.




```c
int dequeue()
{
    if(size<0)
    {
        printf("Queue is empty\n");
    }
    else
    {
        size--;
        front++;
    }
}
```

```c
#include <stdio.h>
#define MAXSIZE 10
int queue[MAXSIZE];
int front = -1;
int rear = -1;
int size = -1;
int isempty()
{
    return size<0;
}
int isfull()
{
    return size == MAXSIZE;
}
void enqueue(int value)
{
    if(size<MAXSIZE)
    {
        if(size<0)
        {
            queue[0] = value;
            front = rear = 0;
            size = 1;
        }
        else if(rear == MAXSIZE-1)
        {
            queue[0] = value;
            rear = 0;
            size++;
        }
        else
        {
            queue[rear+1] = value;
            rear++;
            size++;
        }
    }
    else
    {
        printf("Queue is full\n");
    }
}
int dequeue()
{
    if(size<0)
    {
        printf("Queue is empty\n");
    }
    else
    {
        size--;
        front++;
    }
}

int Front()
{
    return queue[front];
}
```

```c
void display()
{
    int i;
    if(rear>=front)
    {
        for(i=front;i<=rear;i++)
        {
            printf("%d\n",queue[i]);
        }
    }
    else
    {
        for(i=front;i<MAXSIZE;i++)
        {
            printf("%d\n",queue[i]);
        }
        for(i=0;i<=rear;i++)
        {
            printf("%d\n",queue[i]);
        }
    }
}
```

Main function on next slide

Continue >>

Queue

```
84  int main()
85  {
86      enqueue(4);
87      enqueue(8);
88      enqueue(10);
89      enqueue(20);
90      display();
91      dequeue();
92      printf("After dequeue\n");
93      display();
94      enqueue(50);
95      enqueue(60);
96      enqueue(70);
97      enqueue(80);
98      dequeue();
99      enqueue(90);
100     enqueue(100);
101     enqueue(110);
102     enqueue(120);
103     printf("After enqueue\n");
104     display();
105     return 0;
106 }
```

Output ?

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
int intArray[MAX];
int front = 0;
int rear = 0;
int itemCount = 0;
int peek() {
   return intArray[front];
}
bool isEmpty() {
   return itemCount == 0;
}
bool isFull() {
   return itemCount == MAX;
}
int size() {
   return itemCount;
}
void insert(int data) {
   if(!isFull()) {
      if(rear == MAX-1) {
         rear = -1;
      }
      intArray[++rear] = data;
      itemCount++;
   }
}
int removeData() {
   int data = intArray[front++];
   if(front == MAX) {
      front = 0;
   }
   itemCount--;
   return data;
}
int main() {
printf("<Name, abc123, SP-2024>\n");
   insert(2);
   insert(0);
   insert(2);
   insert(4);
if(isFull()) {
      printf("Queue is full!\n");
   }
   printf("Elements in Queue: ");
   for(int i=0;i<5;i++)
   printf("%d,",intArray[i]);
   int num = removeData();
    printf("\n Element removed: %d\n",intArray[0]);
   insert(17);
   insert(18);
printf("Element at front: %d\n",peek());
printf("Queue Size (Array index 0-4): %d \n", size());
   printf("Queue Sequence:  ");
    while(!isEmpty()) {
      int n = removeData();
      printf("%d ",n);
         }
}
```

Output?

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
int peek() {
   return intArray[front];
}
bool isEmpty() {
   return itemCount == 0;
}
bool isFull() {
   return itemCount == MAX;
}
int size() {
   return itemCount;
}
void insert(int data) {
   if(!isFull()) {
      if(rear == MAX-1) {
         rear = -1;
      }
      intArray[++rear] = data;
      itemCount++;
   }
}
```

```c
int removeData() {
   int data = intArray[front++];
   if(front == MAX) {
      front = 0;
   }
   itemCount--;
   return data;
}
int main() {
printf("<Name, abc123, SP-2024>\n");
   insert(2);
   insert(0);
   insert(2);
   insert(4);
if(isFull()) {
      printf("Queue is full!\n");
   }
   printf("Elements in Queue: ");
   for(int i=0;i<5;i++)
   printf("%d,",intArray[i]);
   int num = removeData();
    printf("\n Element removed: %d\n",intArray[0]);
   insert(17);
   insert(18);
printf("Element at front: %d\n",peek());
printf("Queue Size (Array index 0-4): %d \n", size());
   printf("Queue Sequence:  ");
    while(!isEmpty()) {
      int n = removeData();
      printf("%d ",n);
         }
}
```

Output?

Queue (Another example – Array)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
int intArray[MAX];
int front = 0;
int rear = -1; //after MAX
int itemCount = 0;
int peek() {
   return intArray[front];
}
bool isEmpty() {
   return itemCount == 0;
}
bool isFull() {
   return itemCount == MAX;
}
int size() {
   return itemCount;
}
void insert(int data) {
   if(!isFull()) {
      if(rear == MAX-1) {
         rear = -1;
      }
      intArray[++rear] = data;
      itemCount++;
   }
}
int removeData() {
   int data = intArray[front++];
   if(front == MAX) {
      front = 0;
   }
   itemCount--;
   return data;
}
int main() {
    printf("<Name, abc123, SP-2024>\n");
   /* insert 4 items, front : 0 , rear  : 3 */
   insert(2);
   insert(0);
   insert(2);
   insert(3);
   // index : 0 1 2 3
   // queue : 2 0 2 3
   insert(15);
/* insert 1 items */
   // index : 0 1 2 3 4
   // queue : 2 0 2 3 15
    if(isFull()) {
      printf("Queue is full!\n");
   }
   // remove one item
   int num = removeData();
    printf("Element removed: %d\n",num);
/* Remove 1 item */
   // index : 0 1 2 3
   // queue : 0 2 3 15
   // insert more items
   insert(16);
/* Insert 1 Iterm */
   // index : 0 1 2 3  4
   // queue : 0 2 3 15 16
// As queue is full, elements will not be inserted.
   insert(17);
   insert(18);
   // index : 0 1 2 3  4
   // queue : 0 2 3 15 16
   printf("Element at front: %d\n",peek());
printf("Queue Size (Array index 0-4): %d \n", size());
   printf("Queue Sequence:  ");
    while(!isEmpty()) {
      int n = removeData();
      printf("%d ",n);
         }
}
```

Output?

Queue (Another example – Switch Case)

```c
#include <stdio.h>
#include <stdlib.h>
# define SIZE 100
void enqueue();
void dequeue();
void show();
int inp_arr[SIZE];
int Rear = - 1;
int Front = - 1;
int main()
{
    int ch;
    while (1)
    {
printf(" 1.Enqueue \n 2.Dequeue \n");
printf(" 3.Display the Queue\n 4.Exit\n");
printf("Enter your choice of operations : ");
scanf("%d", &ch);
    switch (ch)
       {
        case 1: enqueue();    break;
        case 2: dequeue();    break;
        case 3: show();       break;
        case 4: exit(0);
        default:
        printf("Incorrect choice \n");
        }    }   }
```

```c
void enqueue()
{
    int insert_item;
    if (Rear == SIZE - 1)
       printf("Overflow \n");
    else
    {
    if (Front == - 1)
    Front = 0;
printf("Element to be inserted\n : ");
    scanf("%d", &insert_item);
    Rear = Rear + 1;
    inp_arr[Rear] = insert_item;
    }  }
```

```c
void dequeue()
{
if (Front == - 1 || Front > Rear)
    {
    printf("Underflow \n");
    return ;
    }
    else
    {
printf("Delete Element: %d\n", inp_arr[Front]);
    Front = Front + 1;
    }  }
```

Queue (Another example – Switch Case)

```
57  void show()
58  {
59      if (Front == - 1)
60          printf("Empty Queue \n");
61      else
62      {
63      printf("Queue: \n");
64      for (int i = Front; i <= Rear; i++)
65          printf("%d ", inp_arr[i]);
66          printf("\n");
67      } }
```

Queue (Another example – Switch Case)

```c
#include <stdio.h>
#include <stdlib.h>
#define n 5 // To define the queue size
// The queue is created and front and back are initialised
int queue[n];
int back  = 0;
int front = 0;
int enqueue(int data);
int dequeue();
void print();
```

```c
int enqueue(int data)
{
    // Checks if queue is full
    if (back==n)
    {
        return 0;
    }
    queue[back] = data;
    back = back + 1;
    return 1;
}
```

```c
int dequeue()
{
    // Checks if queue is empty
    if (front==back)
    {
        return 0;
    }
    else
    {
        int data = queue[front];
        front = front + 1;
        return data;
    }
}
```

```c
void print()
{
    if(front!=back)
    {
        for(int i=front;i<back;i++)
        {
            printf("%d ",queue[i]);
        }
    }
}
```

```c
int main()
{
    int ch, data;
    while (1) // A loop to run the program while the user wants
    {
    printf("1. Enqueue \n2. Dequeue \n3. Print \n0. Quit\n");
        printf("Give your choice: ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
                printf("Enter number to enqueue: ");
                scanf("%d", &data);
                if (enqueue(data))
                    printf("Enqueue operation successful");
                else
                    printf("Queue is full");
                break;
            case 2:
                data = dequeue();
                if(data)
                    printf("Data => %d", data);
                else
                    printf("Queue is empty");
                break;
            case 3:
                print();
                break;
            case 0:
                exit(0);
            default:
                printf("Invalid choice");
        } printf("\n");
    }
}
```

Queue (Another example - Pointers)

```c
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node *next;
 };
 struct queue {
    struct node *head;
    struct node *tail; };
void enqueue(struct queue *q, int data)
{
    struct node *new_node = malloc(sizeof(struct node));
    new_node->data = data;
    new_node->next = NULL;
 if (q->tail == NULL)
{
    q->head = new_node;
    q->tail = new_node;
}
else
{
    q->tail->next = new_node;
    q->tail = new_node;
} }
int dequeue(struct queue *q)
{
    if (q->head == NULL)
{
return -1; // queue is empty
}
    int data = q->head->data;
    struct node *temp = q->head;
    q->head = q->head->next;
if (q->head == NULL)
 {
    q->tail = NULL;
 }
free(temp); //free memory
return data;
}
int main()
{
    struct queue q;
    q.head = NULL;
    q.tail = NULL;
        enqueue(&q, 1); //go to Line 10
        enqueue(&q, 2); //go to Line 10
        enqueue(&q, 3); //go to Line 10
printf("%d\n", dequeue(q)); //go to Line 25
printf("%d\n", dequeue(q)); //go to Line 25
printf("%d\n", dequeue(q)); //go to Line 25
}
```

Output?

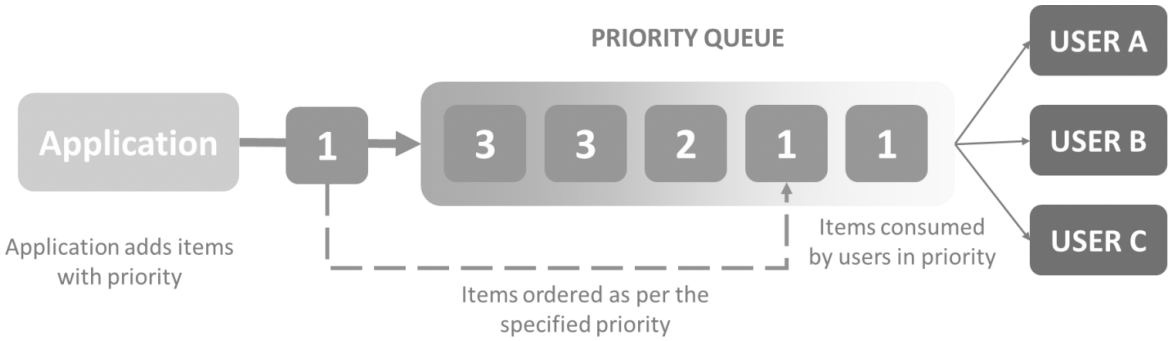
Queue (Another example - Pointers)

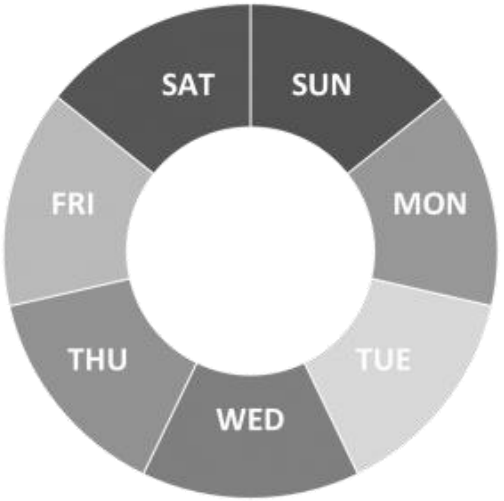
```c
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node *next;
 };
 struct queue {
    struct node *head;
    struct node *tail; };
void enqueue(struct queue *q, int data)
{
    struct node *new_node = malloc(sizeof(struct node));
    new_node->data = data;
    new_node->next = NULL;
 if (q->tail == NULL)
{
    q->head = new_node;
    q->tail = new_node;
}
else
{
    q->tail->next = new_node;
    q->tail = new_node;
} }
int dequeue(struct queue *q)
{
    if (q->head == NULL)
{
return -1; // queue is empty
}
    int data = q->head->data;
    struct node *temp = q->head;
    q->head = q->head->next;
if (q->head == NULL)
 {
    q->tail = NULL;
 }
free(temp); //free memory
return data;
}
int main()
{
    struct queue q;
    q.head = NULL;
    q.tail = NULL;
        enqueue(&q, 1); //go to Line 10
        enqueue(&q, 2); //go to Line 10
        enqueue(&q, 3); //go to Line 10
printf("%d\n", dequeue(&q)); //go to Line 25
printf("%d\n", dequeue(&q)); //go to Line 25
printf("%d\n", dequeue(&q)); //go to Line 25
}
```

Output?

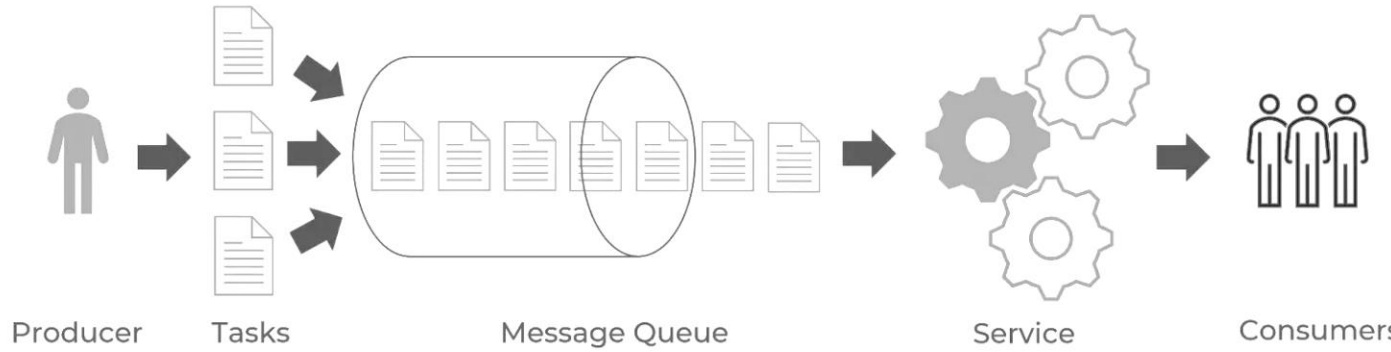
Types of Queues in Data Structure

- There are four different types of queues in data structures:
  1. Simple Queue
  2. Circular Queue
  3. Priority Queue (will be discussed next week)
  4. Double-Ended Queue (Deque)





Application of Queue



- A queue data structure is generally used in scenarios where the FIFO approach (First In First Out) has to be implemented. The following are some of the most common applications of the queue in data structure:
    1. Managing requests on a single shared resource such as CPU scheduling and disk scheduling (Task Scheduling, Resource Allocation)
    2. Handling hardware or real-time systems interrupts (Message Buffering, Message Buffering)
    3. Handling website traffic (Traffic Management, Network protocols, Web servers)
    4. Routers and switches in networking
    5. Maintaining the playlist in media players
    6. In printing systems, queues are used to manage the order in which print jobs are processed.
    7. Queues can be used to handle batch processing jobs, such as data analysis or image rendering (Batch Processing)

Advantages/Disadvantages of Queue

- Queues can have items of any data types.
- You can have a queue of queues, a queue of int, a queue of strings, a queue of arrays, or queue of an object of any type.
- The queue is not readily searchable (You have to start from the end and might have to maintain another queue).
- Adding or deleting elements from the middle of the queue is complex as well.
- Many of the queues are implemented using pointers, so in that respect, many programmers have the notion that they are difficult to create, maintain, and manipulate.

Question: Is searching easy or difficult on Queues?

So if you have some data, which later you would want to be searchable, then using a queue might not be a good idea.

Issues in the applications of Queue

1. Queue overflow
2. Queue underflow
3. Blocking queues (queue may become blocked if it is full or empty. This can cause delays in processing or deadlock)
4. Priority inversion: In some applications, a higher priority element can get stuck behind a lower priority element in the queue. This can lead to priority inversion and result in reduced performance.
5. Synchronization issues: In concurrent applications, multiple threads may access the same queue simultaneously. This can lead to synchronization issues like race conditions, deadlocks, and livelocks.
6. Memory management: In some implementations, a queue may allocate and deallocate memory frequently, leading to memory fragmentation and reduced performance