
Data
Structures
Fibonacci Heap
FM-Index
Hash Table
Trie
Beap
Splay Tree
Top Tree
Tree
Heap
BST
queue
Soft Heap
Stack
Array
Binary Heap
Soft Heap
Weak Heap
Suffix Tree
Doubly Linked List
Stack
Trie
Beap
Rope
BIT
BST
Tango Tree
array
Queue
Array
Trie
Treap
Tree
Array
Binomial Heap
Trie
ডেটা স্ট্রাকচার শিখি বাংলাতে...
Queue
Beap
Rope
Heap
Treap
queue
BIT
Suffix Array
BST
Tree
Splay Tree
Linked List

DATA STRUCTURE

By
Dr. Mona Mohamed Arafa
Lecture , Information System Department ,
Faculty of Computers & Informatics, Benha University
mona.abdelmonem@fci.bu.edu.eg

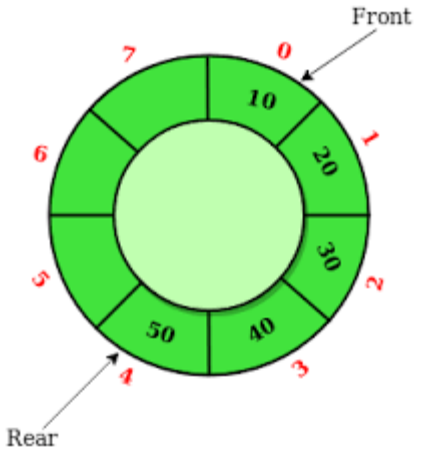
LECTURE QUEUE

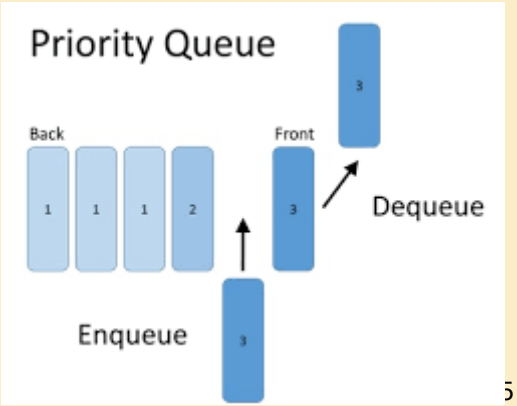
- Introduction to queue
- queue operations
- queue Implementation
- Circular queue

front

rear

Queue



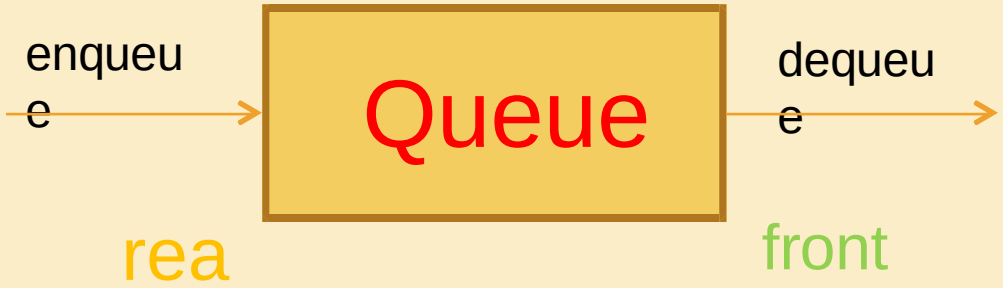


Introduction to queue

- Queue is a linear data structure where the first element is inserted from one end called **REAR** and deleted from the other end called as **FRONT**.
- **Front** points to the **beginning** of the queue and **Rear** points to the **end** of the queue.
- Queue follows the **FIFO (First - In - First Out)** structure.
- According to its FIFO structure, element inserted first will also be removed first.
- In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue), because queue is open at both its ends.

queue operations

- Basic operations: enqueue and dequeue
- enqueue
  - *insert an element at the end of the list(called the rear)*
- dequeue
  - *Delete and return the element at the start of the list(known as the front)*



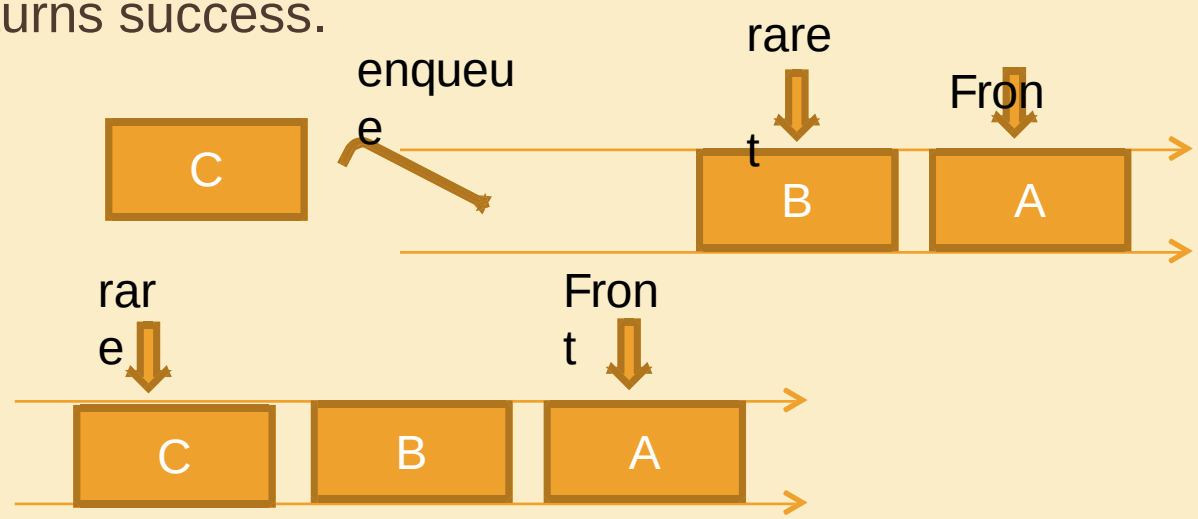
queue operations

Some functionality is added to queue to Check the status of queue

- *isFull() − check if queue is full.*
- *isEmpty() − check if queue is empty.*
- *Front- Front is used to get the front data item from a queue*
- *Rear - Rear is used to get the last item from a queue.*

Queue operations: Enqueue

- The process of putting a new data element onto queue is known as a
  enqueue operation. enqueue operation involves a series of steps −
  - **Step 1** − Checks if the queue is full.
  - **Step 2** − If the queue is full, produces an 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** − Returns success.



queue operations: dequeue

- dequeue operation : accessing the content while removing it from the queue. Dequeue operation may involve the following steps :

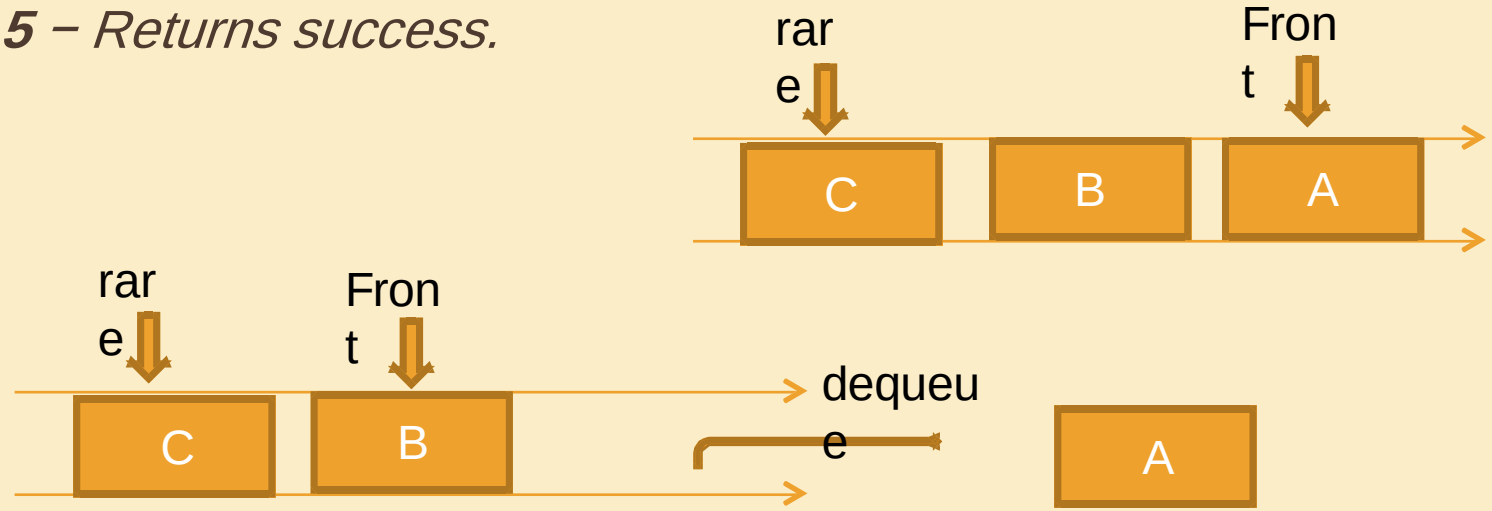
  Step 1 − *Checks if the queue is empty.*

  Step 2 − *If the queue is empty, produces an error and exit.*

  Step 3 − *If the queue is not empty, accesses the data element at which **front** is pointing.*

  Step 4 − *Increment **front** pointer to point to the next available data element..*

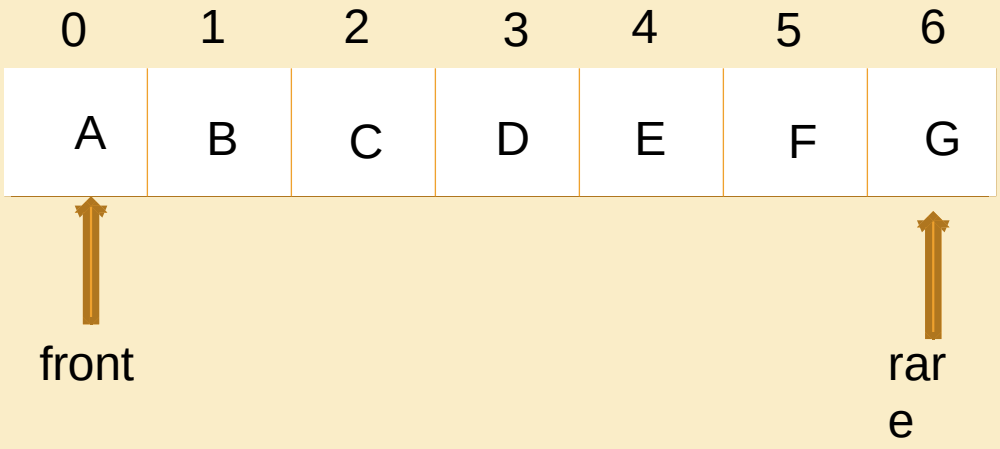
  Step 5 − *Returns success.*



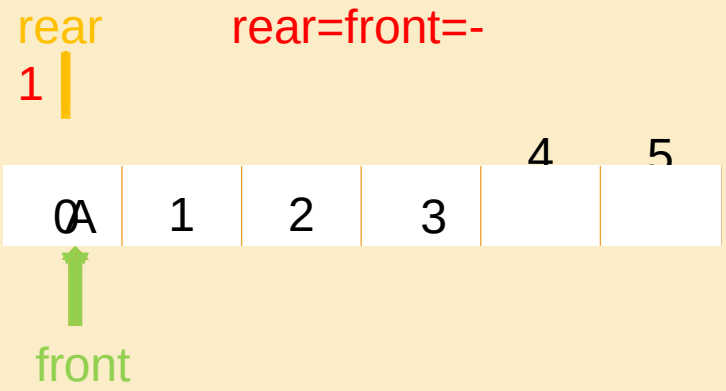
Queue Implementation

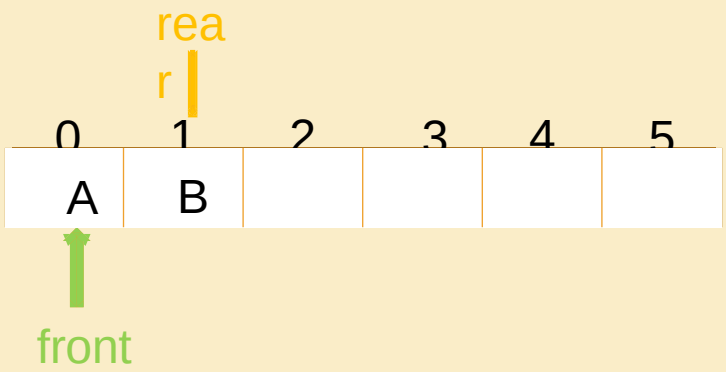
Array-based queue Implementation

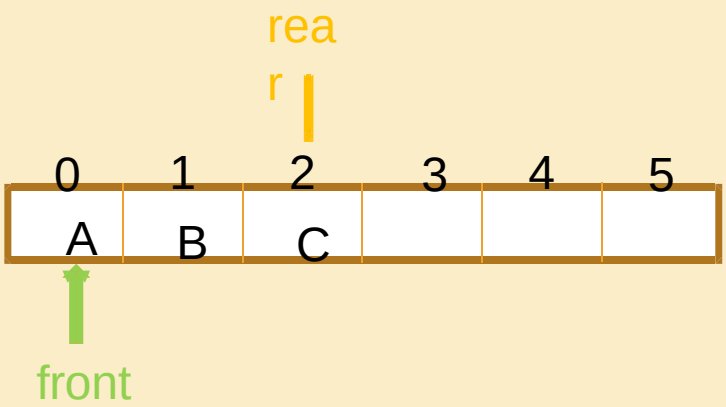
- Array is the easiest way to implement a queue. Queue can be also
  implemented using Linked List

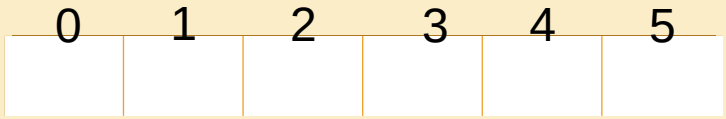


Array-based queue Implementation



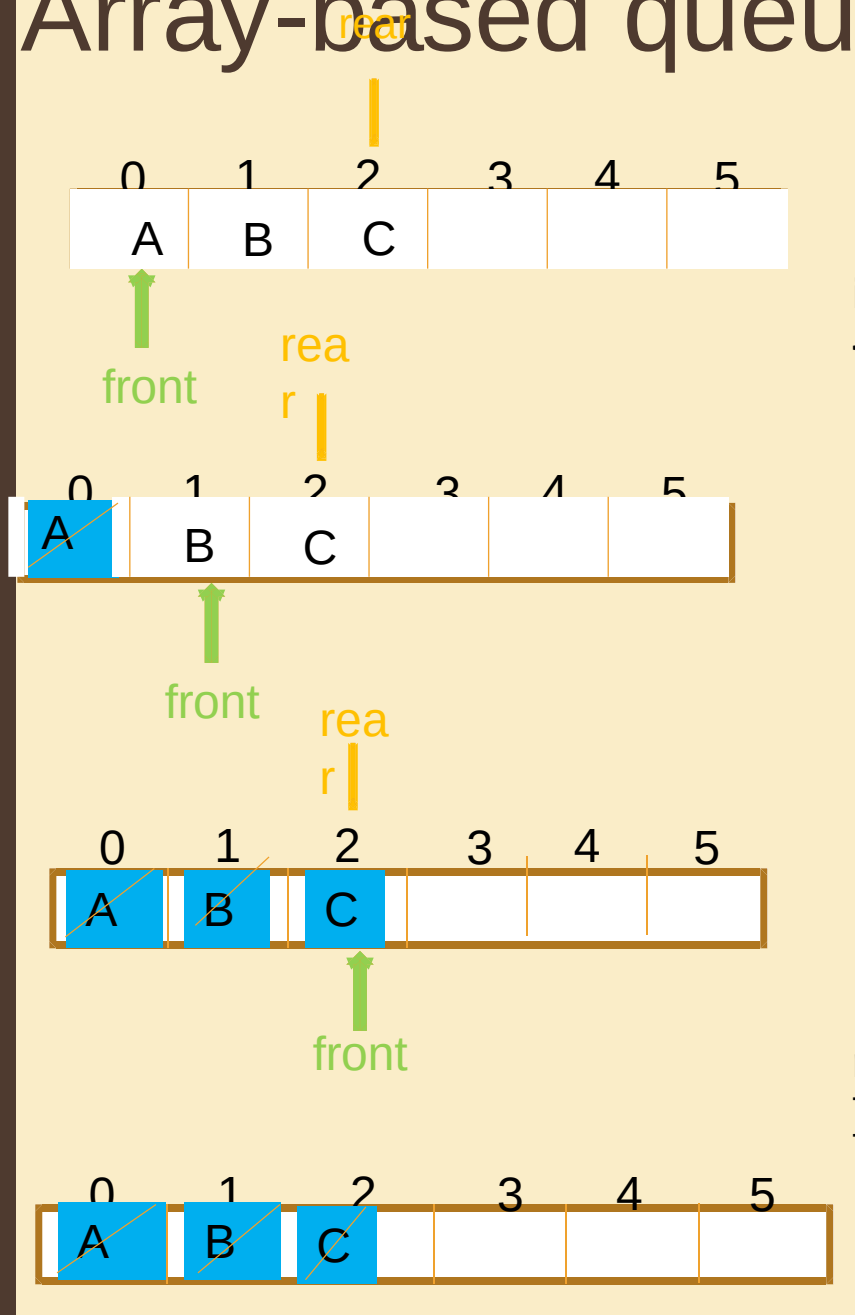







```
void enqueue(char item)
{
        if (is_full())
        {
        cout<< "queue is overflow";
        return;
         }
        else{/*If queue is initially empty*/

                if(front==–1)
                     front=0;
                rear++;
        queue_items[rear]=new_item;
            }

}
```

Array-based queue Implementation




```
int queue::dequeue()
{
        if (is_empty())
        {
                cout<< "queue is underflow";
                exit(0) ;
        }
         else if (front==rear)
        {
                item=queue_items[front];
                front=-1;rear=-1;
        }

        else  {item=queue_items[front];
                front++; }
return item;
}
```

rear=front=-

Array-based queue Implementation

```
#include <iostream>
#define  max_size  100
using namespace std;
class queue
 {
    private:
            int queue_items[max_size];
      public:
           queue() {front=-1,rear=-1;};
            void enqueue(int x);
           int  dequeue();
           int  is_empty();
           int is_full();
           void print_all_elements();
};
```

Array-based queue Implementation

```
//Is_empty


int queue::is_empty()
{
        if(front== -1&&rear== -1)
                return1;
return 0;
}
```

Array-based queue Implementation

```
//Is_fullFunction
int
queue::is_full()
{
        if (rear == max_size-
              1)  return 1;
       return 0;
 }
```

Array-based queue Implementation

```
// enqueue Function
 void queue::enqueue(int new_item)
 {
        if (is_full())
          {
               cout<< "queue is overflow";
               return;
           }
        else{
               if(front==–1)       /*If queue is initially empty*/
                   front=0;
               rear++; queue_items[rear]=new_item;
             }
}
```

Array-based queue Implementation

```
// enqueue Function
int queue::dequeue()
{       int item;
          if (is_empty())
          {
                    cout<< "queue is underflow";
                    exit(0) ;
          }
          else if (front==rear)
          {
                    item=queue_items[front];
                    front=-1;rear=-1;
          }

          else {item=queue_items[front];
                    front++; }
return item;
}
```

Array-based queue Implementation

```
//print all element in the queue
void
queue::print_all_elements()
 {
        if(is_empty())
        {
                cout<<"\n Queue is empty"<<endl; return;
        }
        cout<<"the current items of the queue are:"
        <<endl;
        for(int i=front; i<=rear; i++)
                cout<<queue_items[i]<<"
                ";
```

Array-based queue Implementation

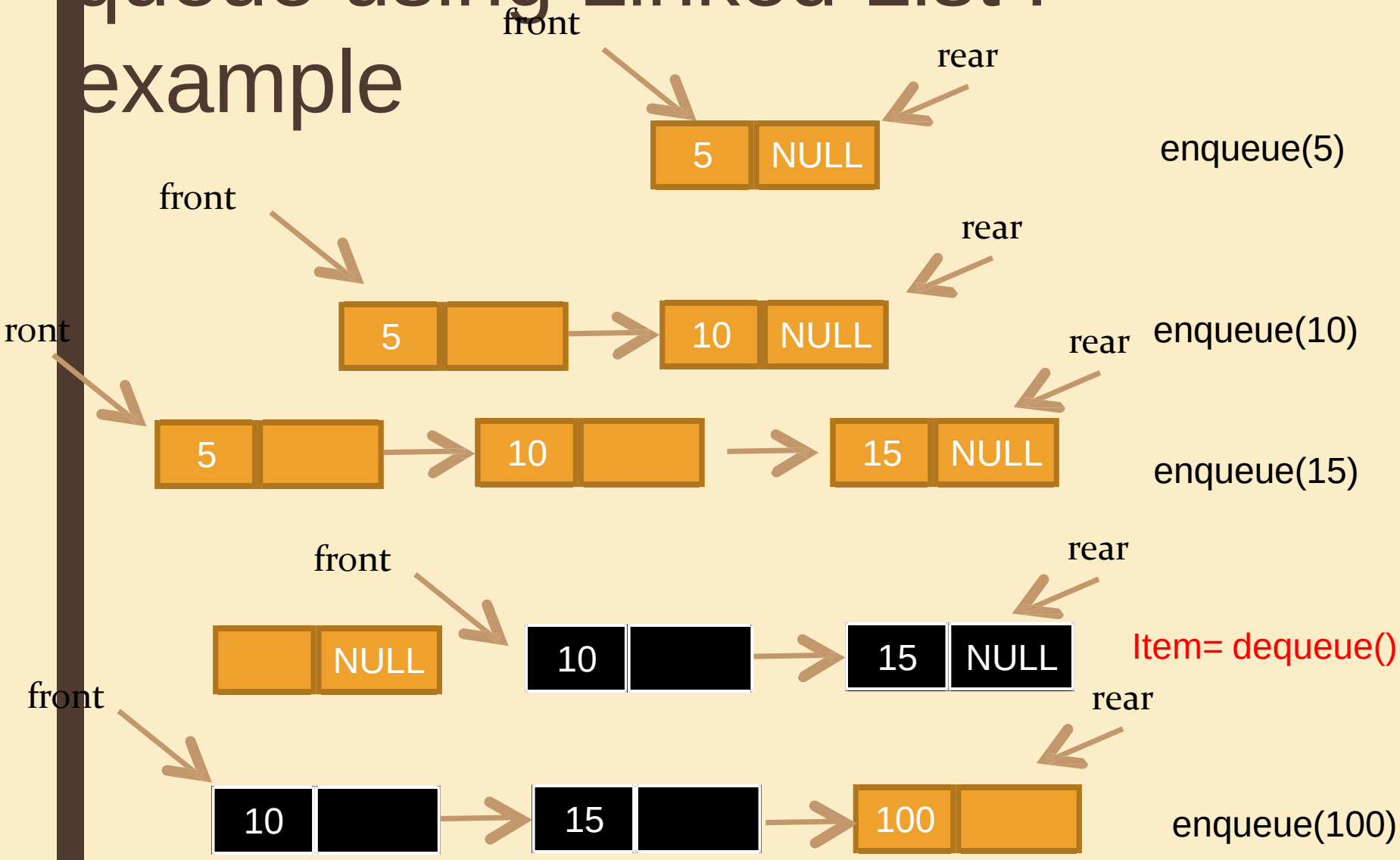
```
// File: Main_program.cpp

// Run of the Main Program

int  main()
{
     queue q;
    q.enqueue(1); q.enqueue(2); q. enqueue(3);  q.enqueue(4);
    q. print_all_elements();
    int val=q. dequeue(); val=q. dequeue();
    q. print_all_elements();
    q. enqueue(5); q. enqueue(6);
    q. print_all_elements();
    }
```

the current items of the queue are 1 2 3 4

the current items of the queue are 3 4

the current items of the queue are

queue using Linked List : example



queue using Linked List

```
class queue
{
private: node *front,*rear;
public:
    queue(){front=NULL;rear=NULL;}
    void enqueue(int value);
    int dequeue();
    void display();
};
```

queue using Linked List :enqueue()

```
Void queue:: enqueue (int value)
        //insert last in linked list
{       \\ insert element at rear (last)
\\ no nodes in list
if (rear == NULL)
{
        rear = new node;
        rear->next = NULL;
        rear->data = value;
        front = rear;
 }
\\ al least on node in list
else
{
node * temp=new node;
temp->data = value;
temp->next = NULL;
rear->next = temp;
rear = temp;
 }

}
```

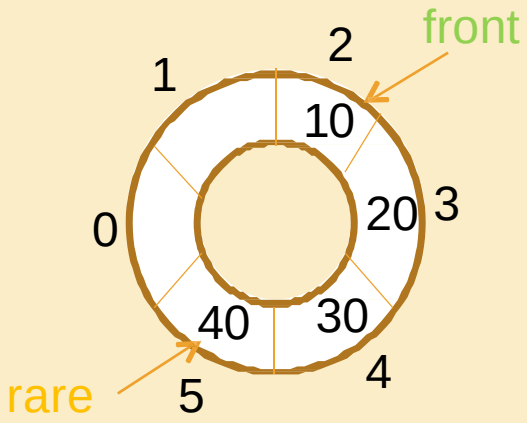
Queue using Linked List

```
int queue:: dequeue()                    //delete first in linked list
{           int item;
          node * temp;
          temp = front;
          if (front == NULL)
                    { cout<<"Underflow"<<endl; exit(0); }
          else if (temp->next == NULL)
              {     item =front->data;
                  front = NULL; rear = NULL;       }
           else
              {
                    item= temp->data;
                    temp = temp->next;
                     front = temp;  }
return item;}
```

Queue using Linked List : display()

```
void display()
{
        node *temp = front;
        if ((front == NULL) && (rear == NULL))
           { cout<<"Queue is empty"<<endl;
                return; }  cout<<"Queue elements
                are: ";
        while (temp != NULL)
                {
                        cout<<temp->data<<"
                        ";  temp = temp->next;
                }
                cout<<endl;
}
```

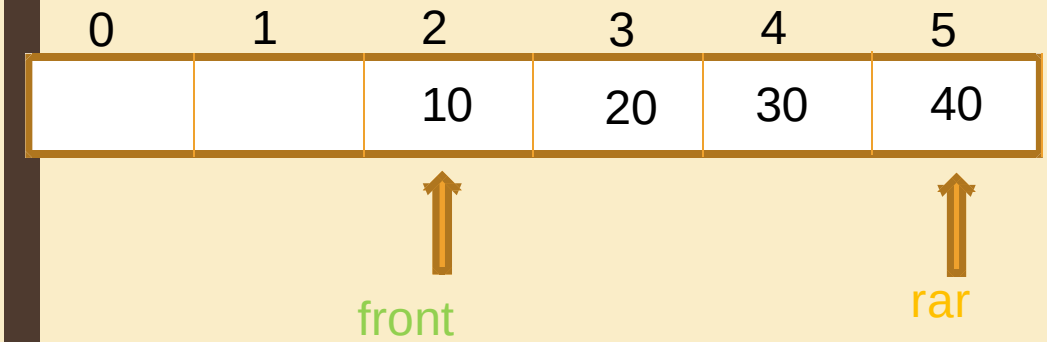
Circular queue

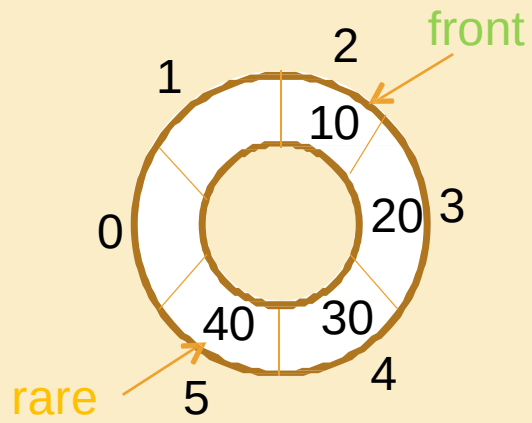


Circular queue

- Circular queue avoids the wastage of space in a regular queue implementation using arrays
- Circular Queue works by the process of circular increment i.e. when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size.

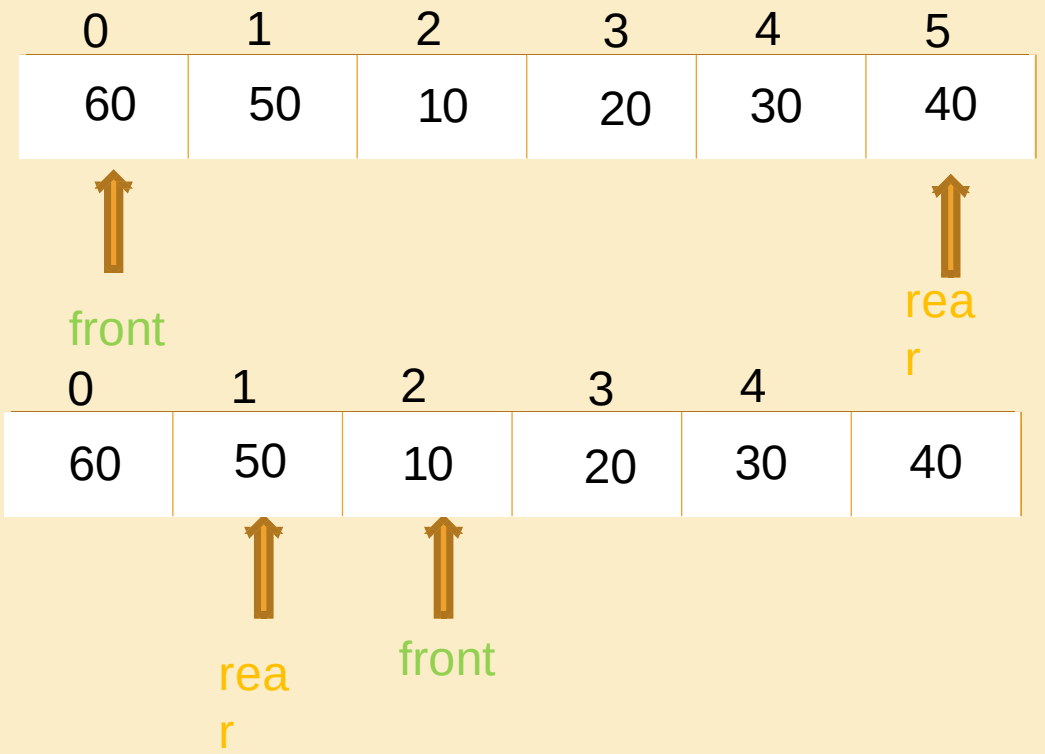
i.e. if rear + 1 == 6 (overflow!), rear = (rear + 1)%6= 0 (start of queue





Circular queue Implementation Array-based

```
bool Cqueue::is_full()
{
  if(front == 0 && rear == max_size - 1) return true;
    if(front == rear + 1)
        return true;
    return false;
}
```



Circular queue Implementation Array-based

```
// enqueue Function of circular queue
 void Cqueue::enqueue(int new_item)
 {
      if(is_full()){
           cout << "Queue is full";
       }
      else {
           if(front == -1) front = 0;
           rear = (rear + 1) % max_size;
          queue_items[rear] = new_item;
               }
}
```

Circular queue Implementation

```
// enqueue Function
int Cqueue::dequeue()
{       int item;
            if (is_empty())
            {
                        cout<< "queue is underflow";
                        return -1 ;
            }
            else if (front==rear)
            {
                        item=queue_items[front];
                        front=-1;rear=-1;
            }

            else {item=queue_items[front];
                        front=(front+1)%max_size; }
return item;
```

Array-based queue Implementation

```
//print all element in the queue
void
Cqueue::print_all_elements()
 {
        if(is_empty())
        {
                cout<<"\n Queue is empty"<<endl; return;
        }
        cout<<"the current items of the queue are:"
        <<endl;  for(int i=front; i!=rear; i=(i+1)%max_size)
                cout<<queue_items[i]<<"
";  cout<<queue_items[i];
        cout<<endl;
}
```


THANK
YOU


Any
Question?