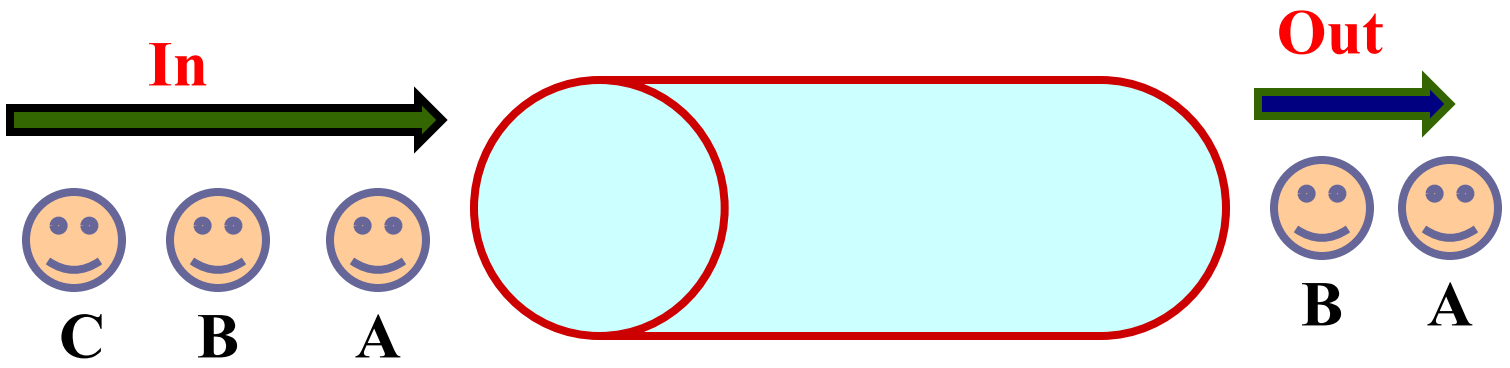
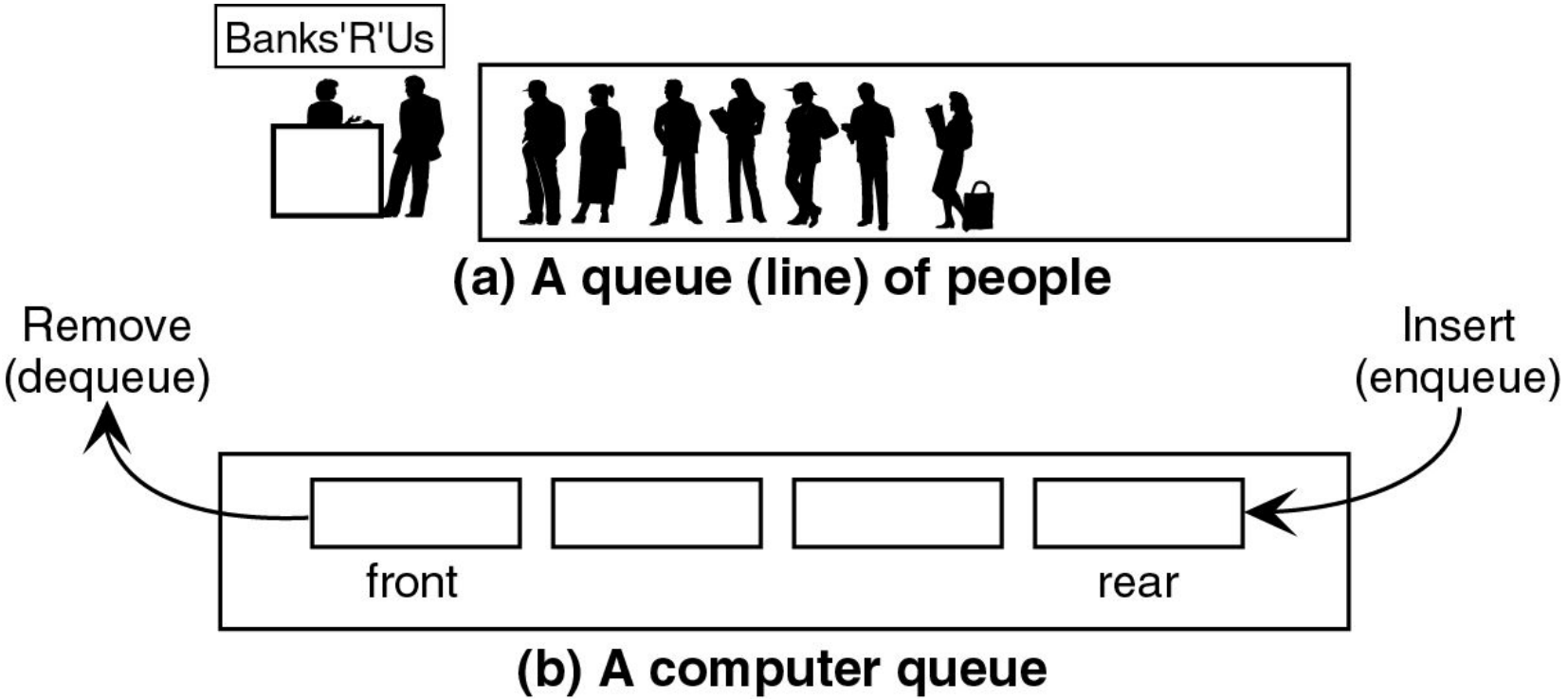
Queues

A First-in First-out (FIFO) List



Also called a QUEUE

Queues in Our Life



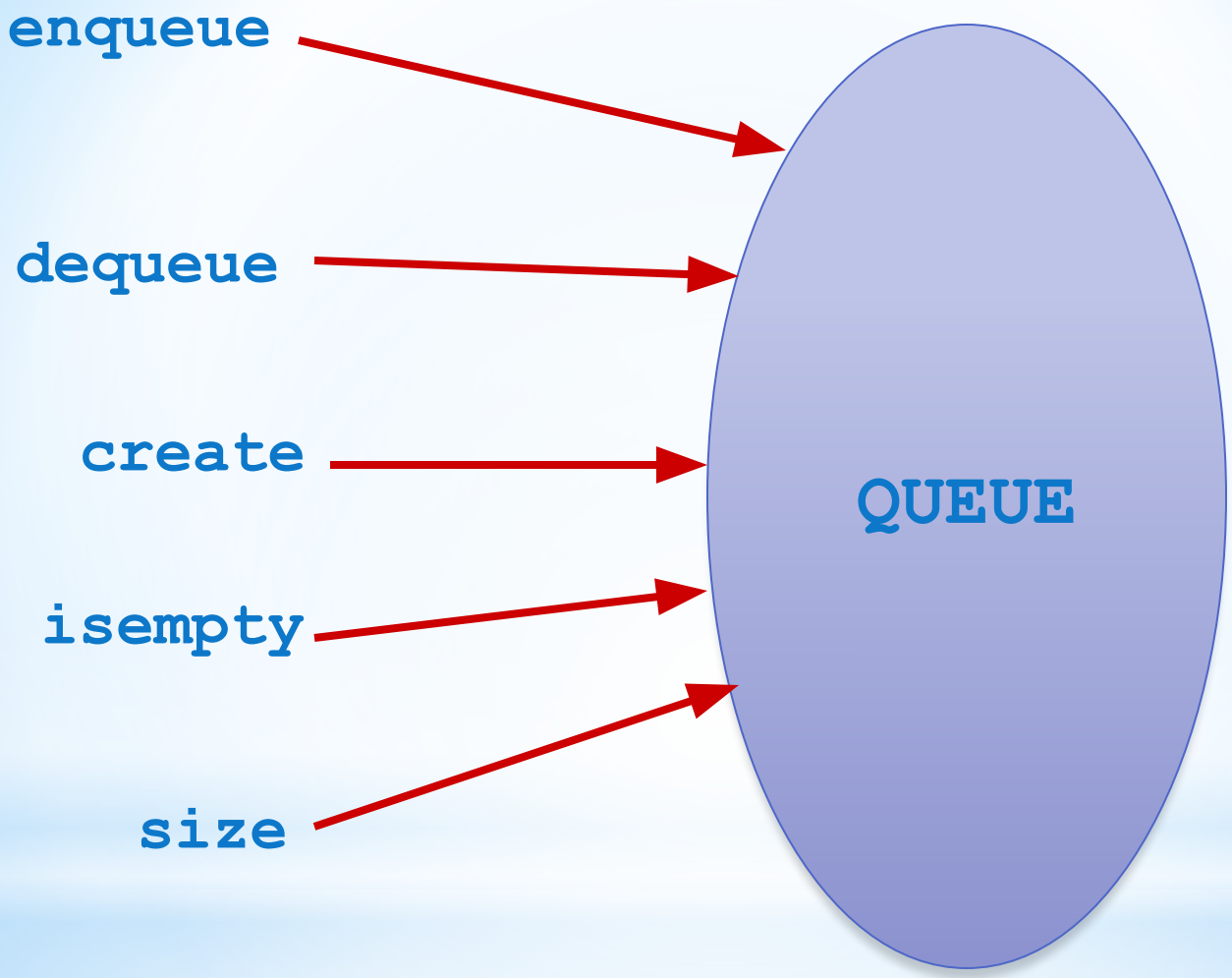
- A queue is a **FIFO** structure: Fast In First Out

Basic Idea

- 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 Representation

- As in stacks, a queue can also be implemented using Arrays (static implementation) and Linked-lists (dynamic implementation).


enqueue
dequeue
create
isempty
size
QUEUE

QUEUE: First-In-First-Out (LIFO)

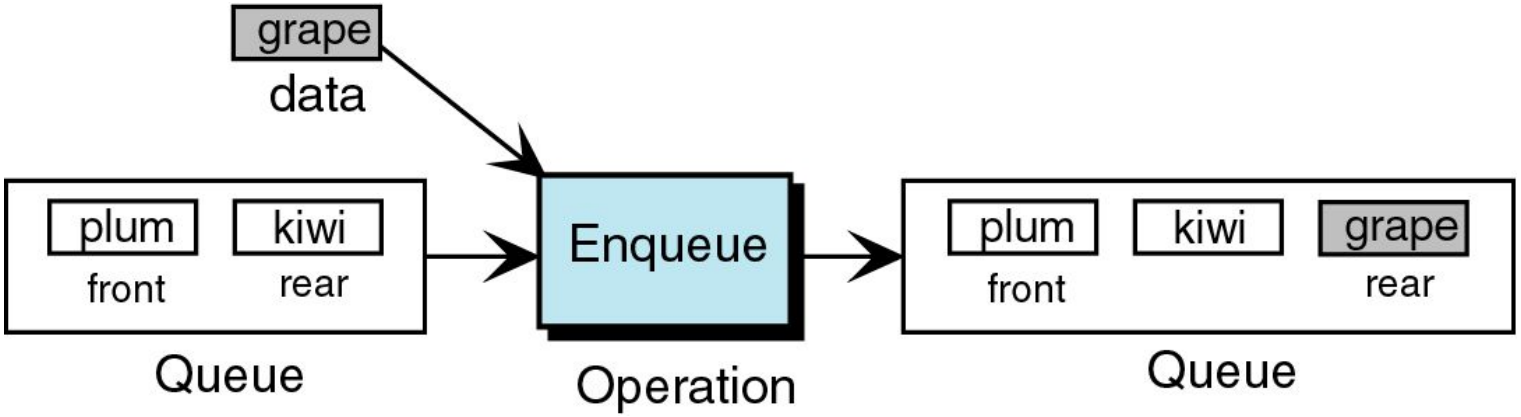
```
void enqueue (queue *q, int element);
                    /* Insert an element in the queue */
int dequeue (queue *q);
                    /* Remove an element from the queue */
queue *create();
                    /* Create a new queue */
int isempty (queue *q);
                    /* Check if queue is empty */
int size (queue *q);
                    /* Return the no. of elements in queue */
```

Assumption: queue contains integer elements!

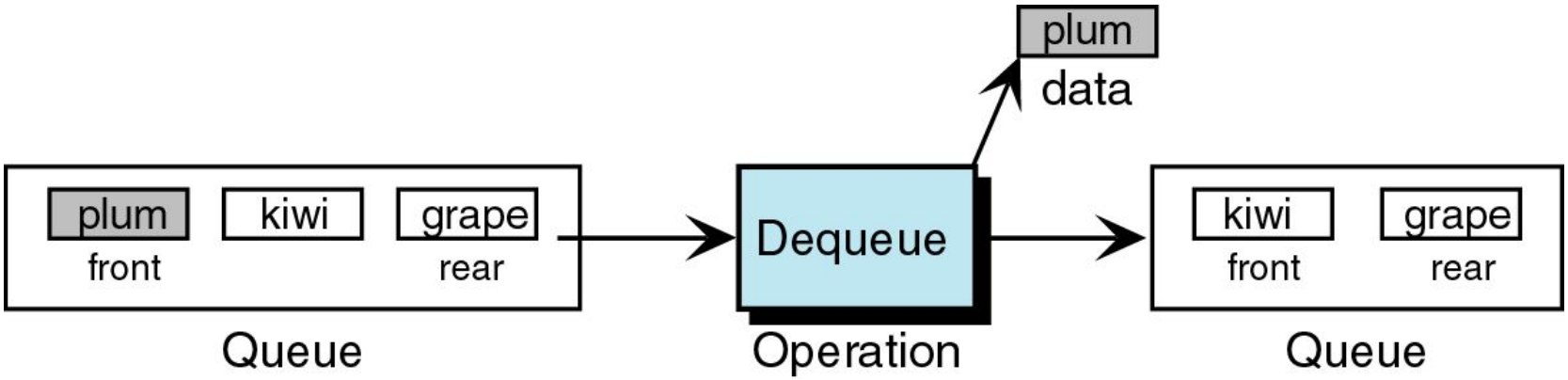
Basic Operations with Queues

- Enqueue - Add an item to the end of queue
  - Overflow
- Dequeue - Remove an item from the front
  - Could be empty
- Queue Front - Who is first?
  - Could be empty
- Queue End - Who is last?
  - Could be empty

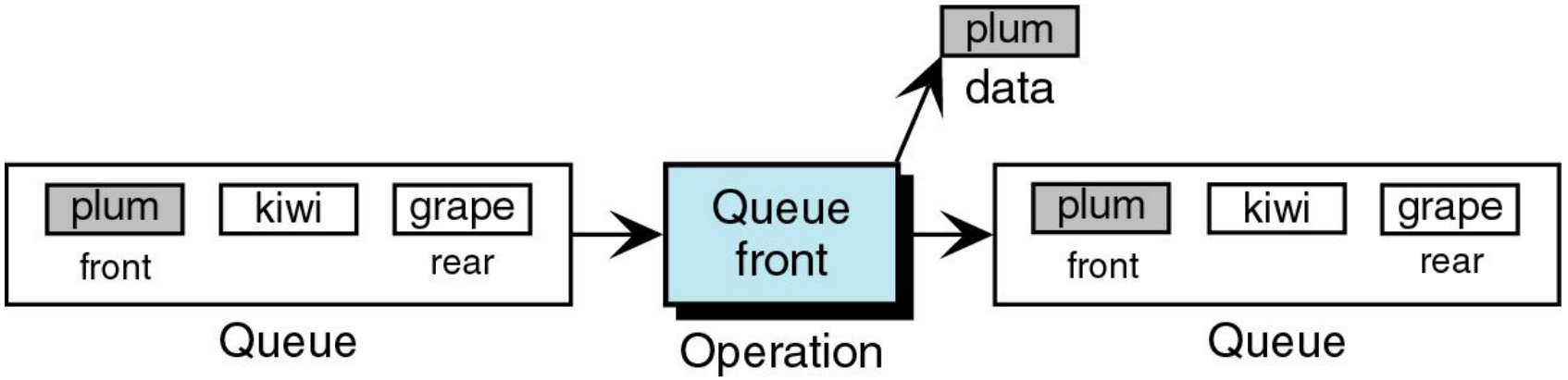
Enqueue


grape
data
plum
kiwi
front
rear
Queue
Enqueue
Operation
plum
kiwi
grape
front
rear
Queue

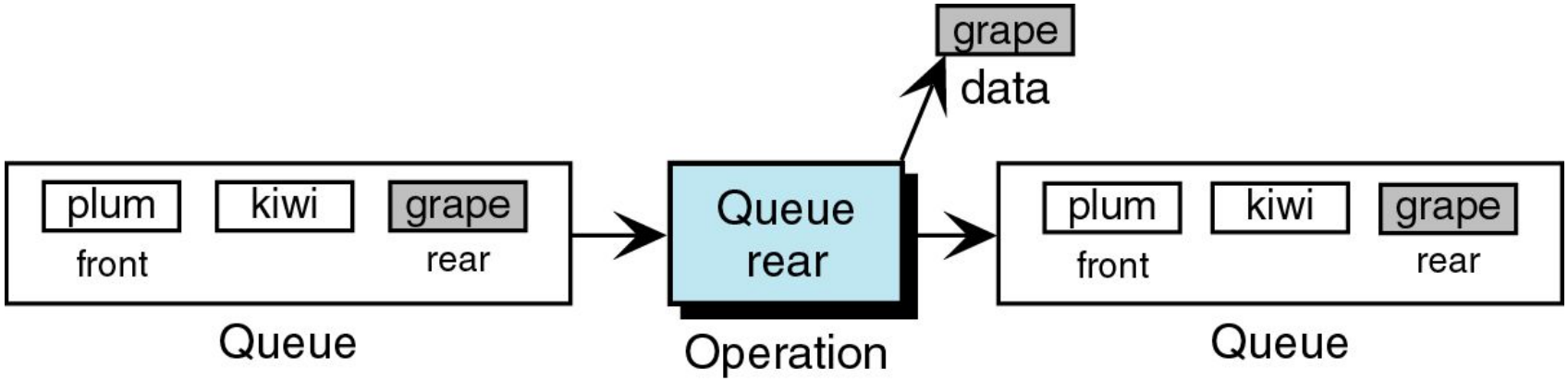
Dequeue

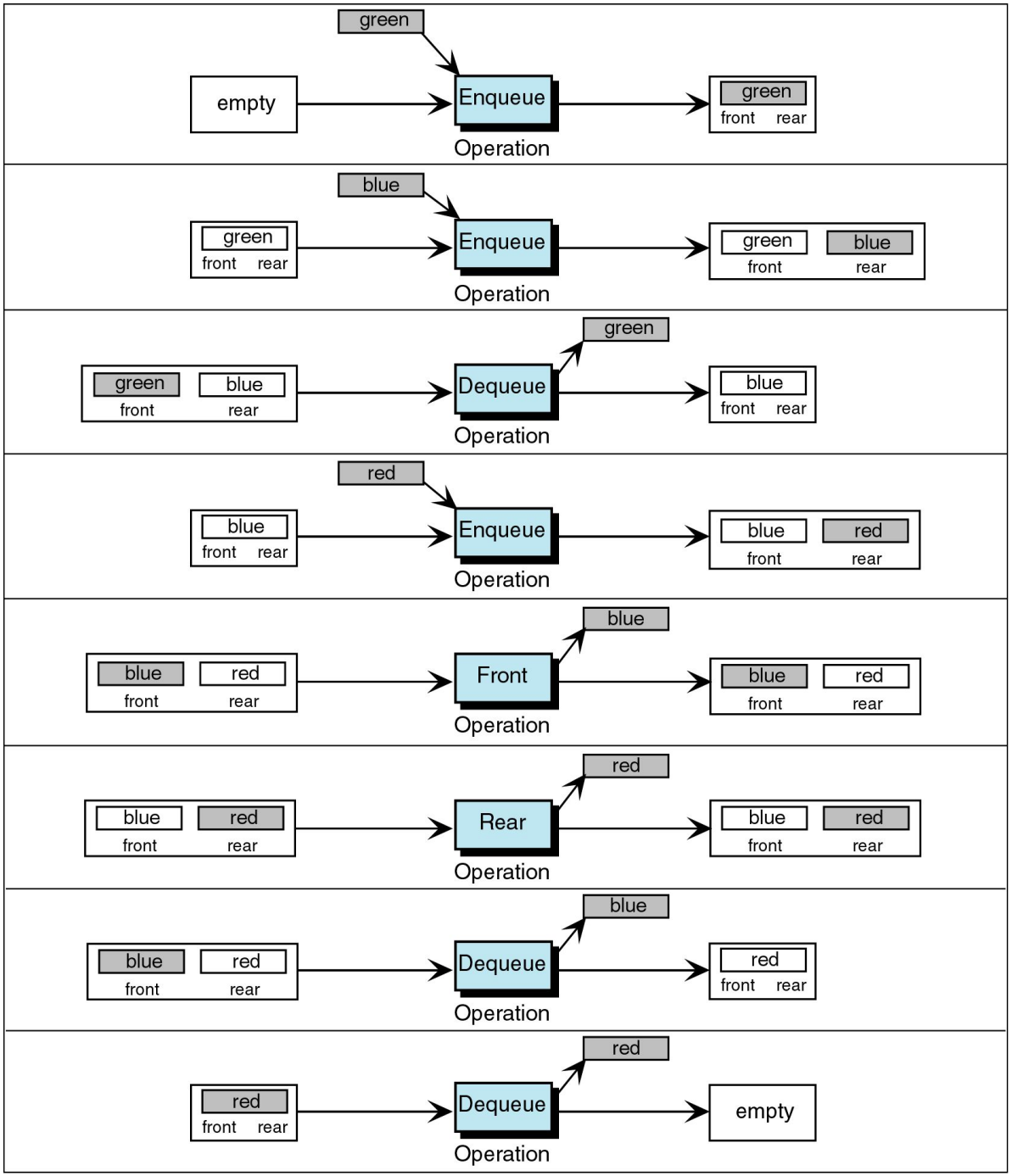

plum
data
plum
kiwi
grape
front
rear
Queue
Dequeue
Operation
kiwi
grape
front
rear
Queue

Queue Front


plum
data
plum
kiwi
grape
front
rear
Queue
Queue front
Operation
plum
kiwi
grape
front
rear
Queue

Queue Rear


grape
data
plum
kiwi
grape
front
rear
Queue
Queue
rear
Operation
plum
kiwi
grape
front
rear
Queue


green
empty
Enqueue
Operation
green
front rear
blue
green
front rear
Enqueue
Operation
green
blue
front rear
green
Dequeue
Operation
blue
front rear
red
blue
front rear
Enqueue
Operation
blue
red
front rear
blue
Front
Operation
blue
red
front rear
red
blue
red
front rear
Rear
Operation
blue
red
front rear
blue
red
front rear
blue
Dequeue
Operation
red
front rear
red
red
front rear
Dequeue
Operation
empty

Q. Draw the queue structure in each case when the following operations are performed on an empty queue.

(a) Add A, B, C, D, E, F
(b) Delete two letters
(c) Add G
(d) Add H
(e) Delete four letters
(f) Add I

Sol:

Intial Queue.

a.) Add A,B,C,D,E,F

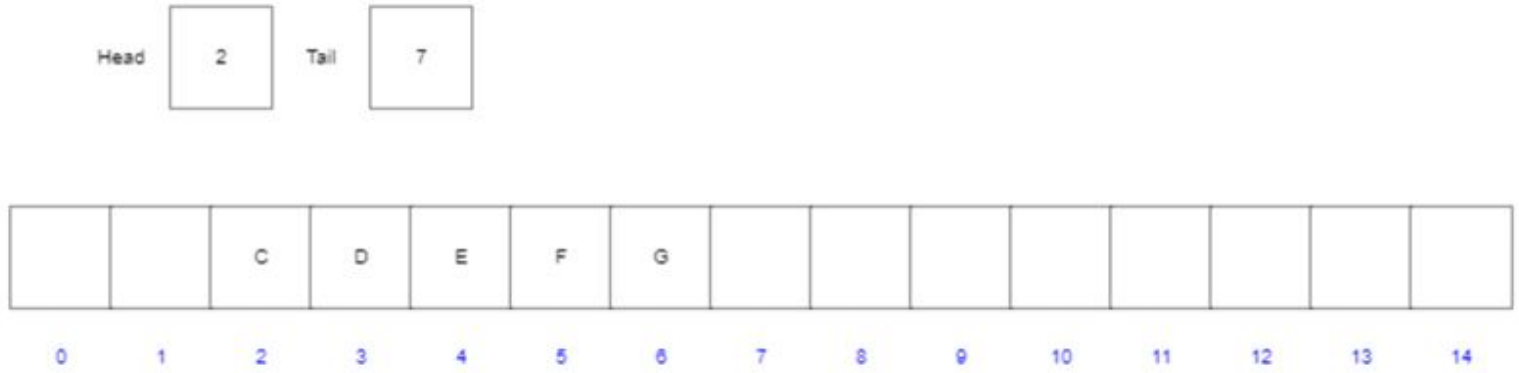
(b) Delete two letters

dequeuing A

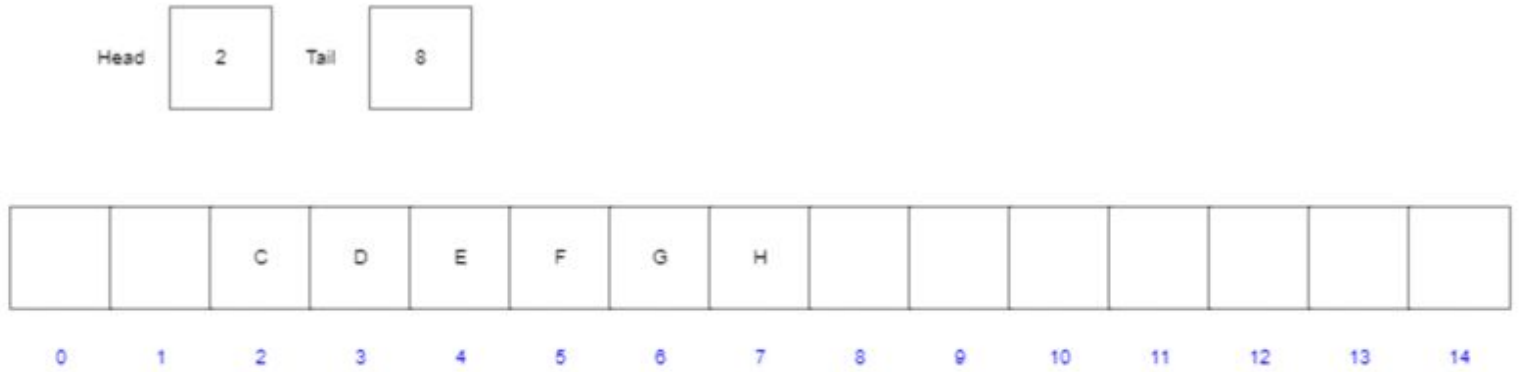
Dequeing B

\

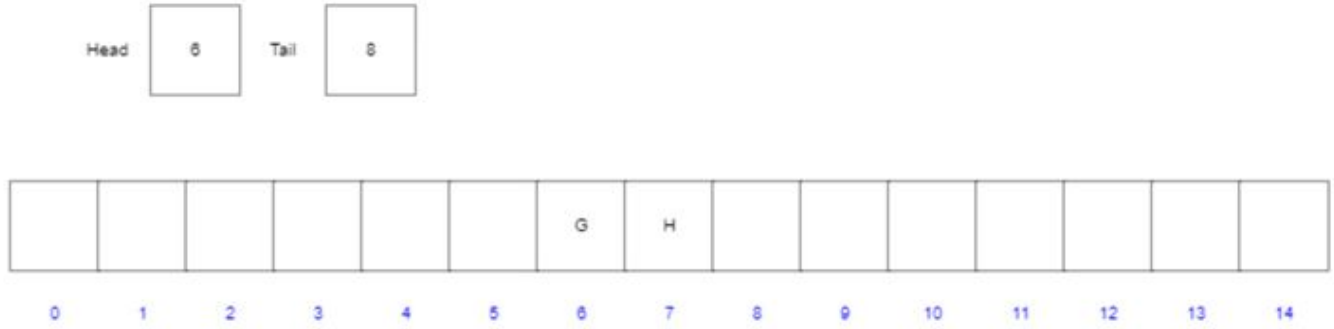
(c) Add G


Head
2
Tail
7
C
D
E
F
G
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

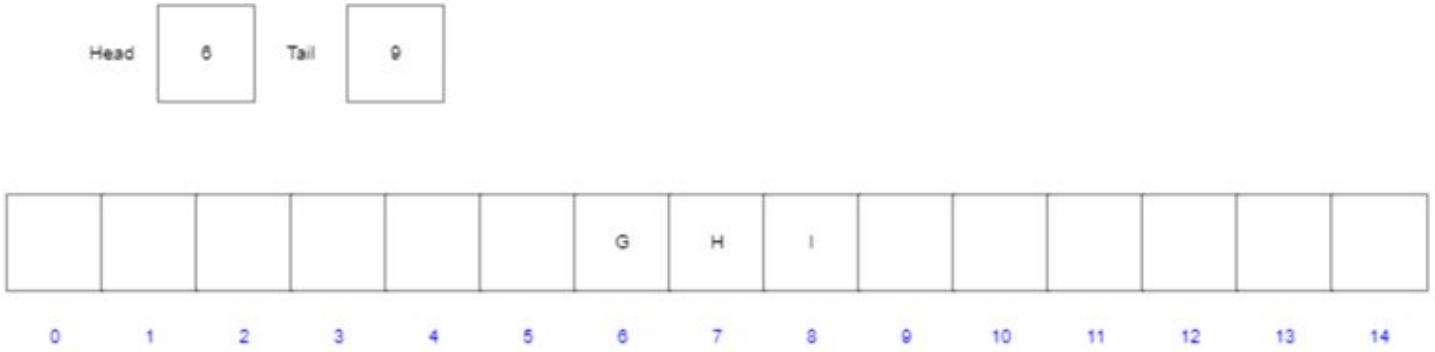
d) Add H


Head
2
Tail
8
C
D
E
F
G
H
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

(e) Delete four letters



(f) Add I



Stacks, Queues, and Linked Lists

- Stacks (Last in/First out List)
  - Operations: Push, Pop, Test Empty, Test Full, Peek, Size
- Queue(First in/First out List)
  - Operations: Insert, Remove, Test Empty, Test Full, Peek, Size
- Linked List(A list of elements connected by pointers)
  - Insert, Delete, Find, Traverse, Size
  - Advantages: can grow, delete/insert with assignments

Stack vs Queue

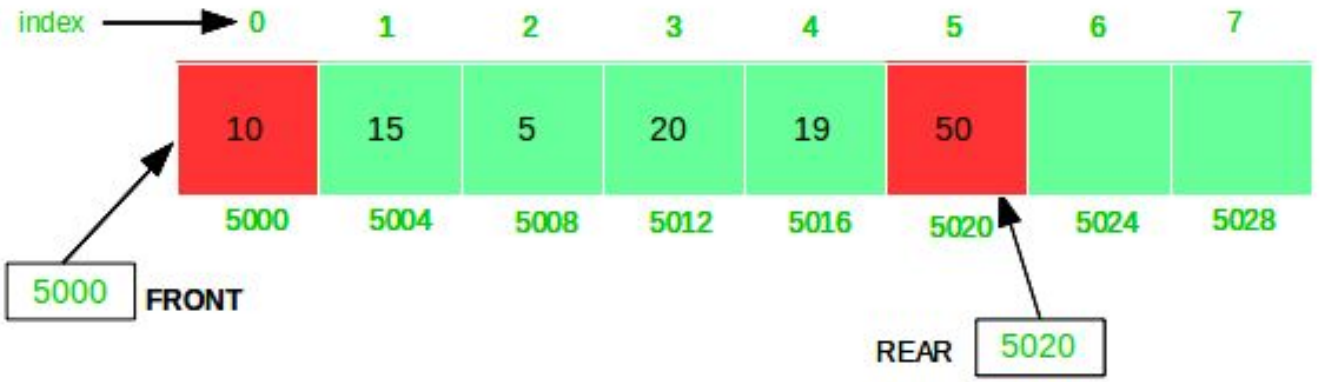
A stack is logically a LIFO type of list.	A queue is logically a FIIO type of list
In a stack insertions and del etions are possible only at one end	In queue insertion is done at one end and deletion is performed at other end
Only one item can be added at a time	only one item can be add at a time
only one item can be deleted at a time	only one item can be deleted at a time
no element other than the top element of stack is visible	no element other than the front and rear element are visible

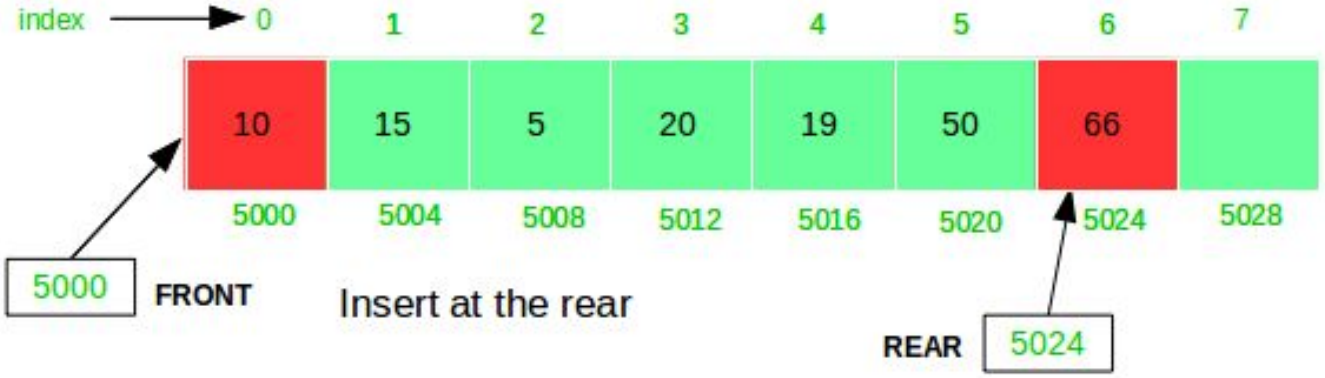
Implementing Queue with Arrays

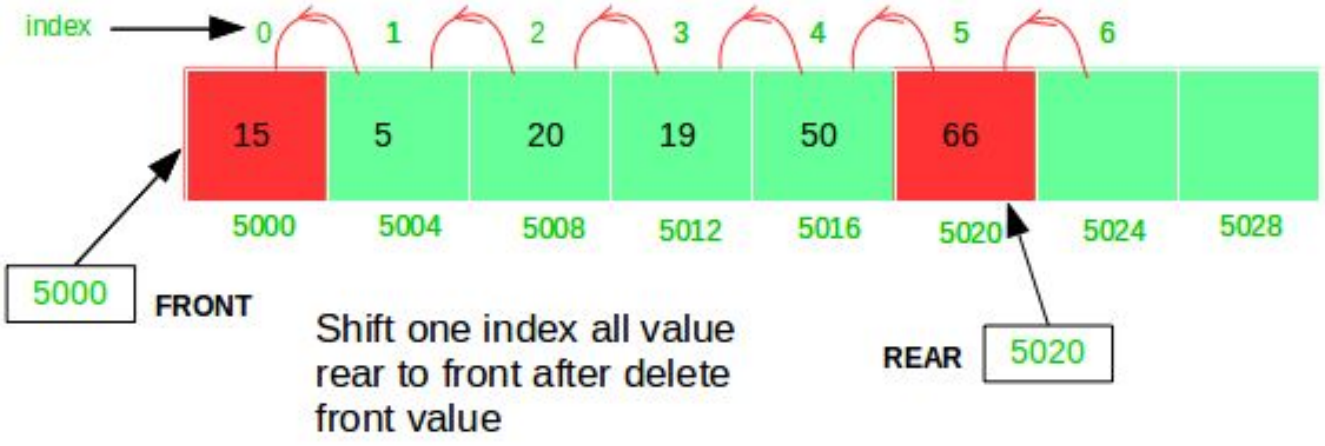
To implement a queue using an array-

- create an array arr of size **n** and
- take two variables **front** and **rear** both of which will be initialized to 0 which means the queue is currently empty.
- Element
  - rear is the index up to which the elements are stored in the array and
  - front is the index of the first element of the array.

Code:
https://towardsdev.com/implementing-a-queue-in-c-using-arrays-a-step-by-step-guide-66146af3f5e1


index
0
1
2
3
4
5
6
7
10
15
5
20
19
50
5000
5004
5008
5012
5016
5020
5024
5028
5000
FRONT
REAR
5020


index
0
1
2
3
4
5
6
7
10
15
5
20
19
50
66
5000
5004
5008
5012
5016
5020
5024
5028
5000
FRONT
Insert at the rear
REAR
5024


index
0
1
2
3
4
5
6
15
5
20
19
50
66
5000
5004
5008
5012
5016
5020
5024
5028
5000
FRONT
Shift one index all value
rear to front after delete
front value
REAR
5020

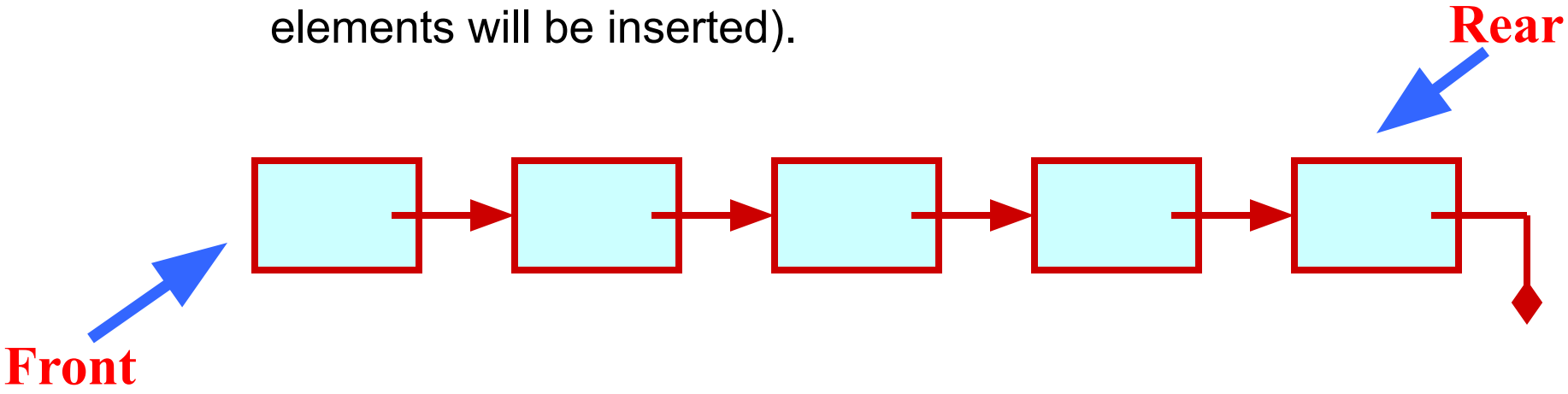
Types of Queues

Types of queue (theory):
https://www.javatpoint.com/ds-types-of-queues

LL implementation of Simple Queue:
https://www.javatpoint.com/linked-list-implementation-of-queue

Queue Implementation Using Linked List

- Basic idea:
  - Create a linked list to which items would be added to one end and deleted from the other end.
  - Two pointers will be maintained:
    - One pointing to the beginning of the list (point from where elements will be deleted).
    - Another pointing to the end of the list (point where new elements will be inserted).



Example :Queue using Linked List

```
struct qnode
{
  int val;
   struct qnode *next;
};

struct queue
{
   struct qnode *qfront, *qrear;
};
typedef struct queue QUEUE;
```

```
void enqueue (QUEUE *q,int element)
{
   struct qnode *q1;
   q1=(struct qnode *)malloc(sizeof(struct qnode));
   q1->val= element;
   q1->next=q->qfront;
   q->qfront=q1;
}
```

Example :Queue using Linked List

```
int size (queue *q)
{
    queue *q1;
    int count=0;
    q1=q;
    while(q1!=NULL)
    {
      q1=q1->next;
      count++;
    }
return count;
}
```

```
int peek (queue *q)
{
   queue *q1;
   q1=q;
   while(q1->next!=NULL)
      q1=q1->next;
return (q1->val);
}
```

```
int dequeue (queue *q)
{
   int val;
   queue *q1,*prev;
   q1=q;
   while(q1->next!=NULL)
   {
      prev=q1;
      q1=q1->next;
   }
   val=q1->val;
   prev->next=NULL;
   free(q1);
return (val);
}
```

Assume:: queue contains integer elements

```
void  enqueue  (queue q,  int element);
                              /* Insert an element in the queue */
int  dequeue (queue q);
                          /* Remove an element from the queue */
queue  *create  ();
                                  /* Create a new queue */
int  isempty  (queue q);
                                  /* Check if queue is empty */
int  size  (queue q);
                    /* Return the number of elements in queue */
```

Creating a queue

front = NULL;

rear = NULL;

Inserting an element in queue

```
void  enqueue  (queue q,  int x)
{
    queue  *ptr;
    ptr  =  (queue *)  malloc (sizeof (queue));

    if  (rear == NULL)                    /* Queue is empty */
        {
            front  =  ptr;    rear  =  ptr;
            ptr->element  =  x;
            ptr->next  =  NULL;
        }
    else                                  /* Queue is not empty
  */
        {
            rear->next  =  ptr;
            ptr ->element  =  x;
            ptr->next  =  NULL;
        }
}
```

Deleting an element from queue

```
int  dequeue  (queue q)
 {
    queue  *old;

    if  (front == NULL)                          /* Queue is empty */
        printf  ("\n Queue is empty");

    else   if  (front == rear)                    /* Single element */
                {
                   k = front->element;
                   free (front);    front = rear = NULL;
                   return (k);
                }
           else
                {
                   k  =  front->element;    old  =  front;
                   front  =  front->next;
                   free (old);
                   return (k);
                }
 }
```

Checking if empty

```
int  isempty  (queue q)
{
    if  (front == NULL)
        return (1);
    else
        return (0);
}
```

Types of Queues

1. Deque

2. Circular Queue

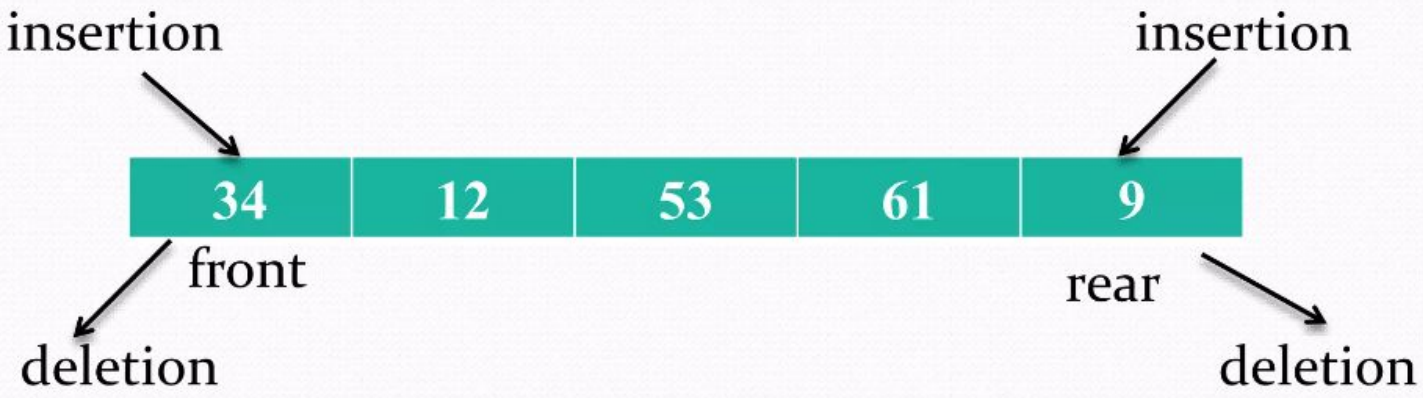
3. Priority Queue

DEQUES

1.Deque stands for *double ended queue*.

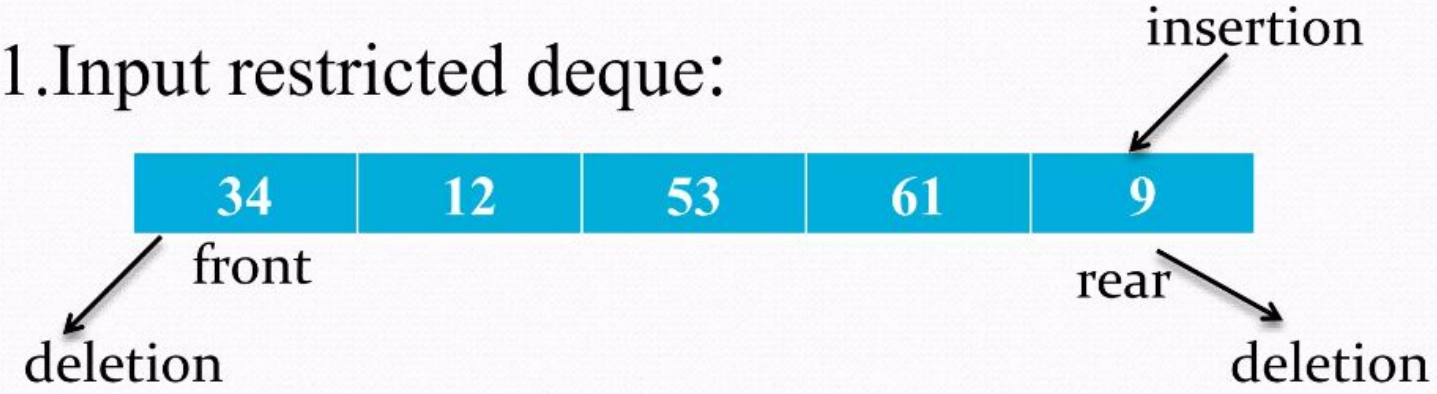
2.Elements can be inserted or deleted at either end.

3. Also known as *head-tail linked list*.



Types Of Deque

1.Input restricted deque:



2. Output restricted deque:

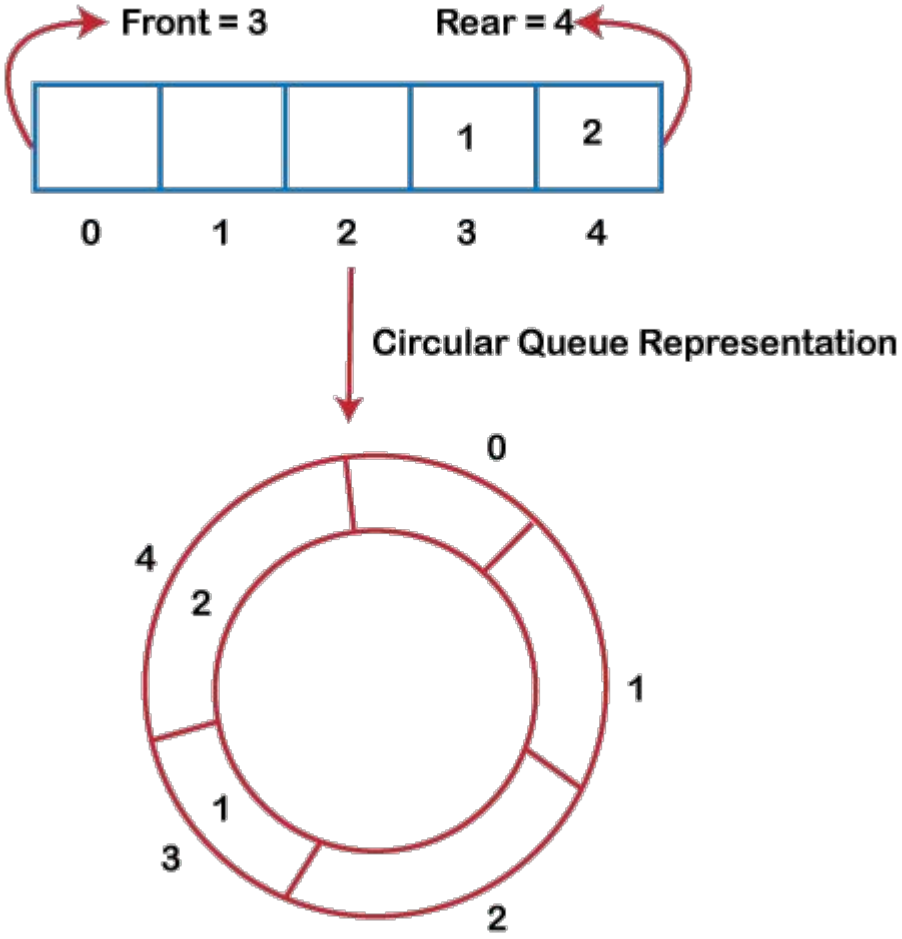
CIRCULAR QUEUES

- Circular queue are used to remove the drawback of simple queue.

- Both the front and the rear pointers wrap around to the beginning of the array.

- It is also called as "***Ring buffer***".

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.



Why is circular queue better than a linear queue?

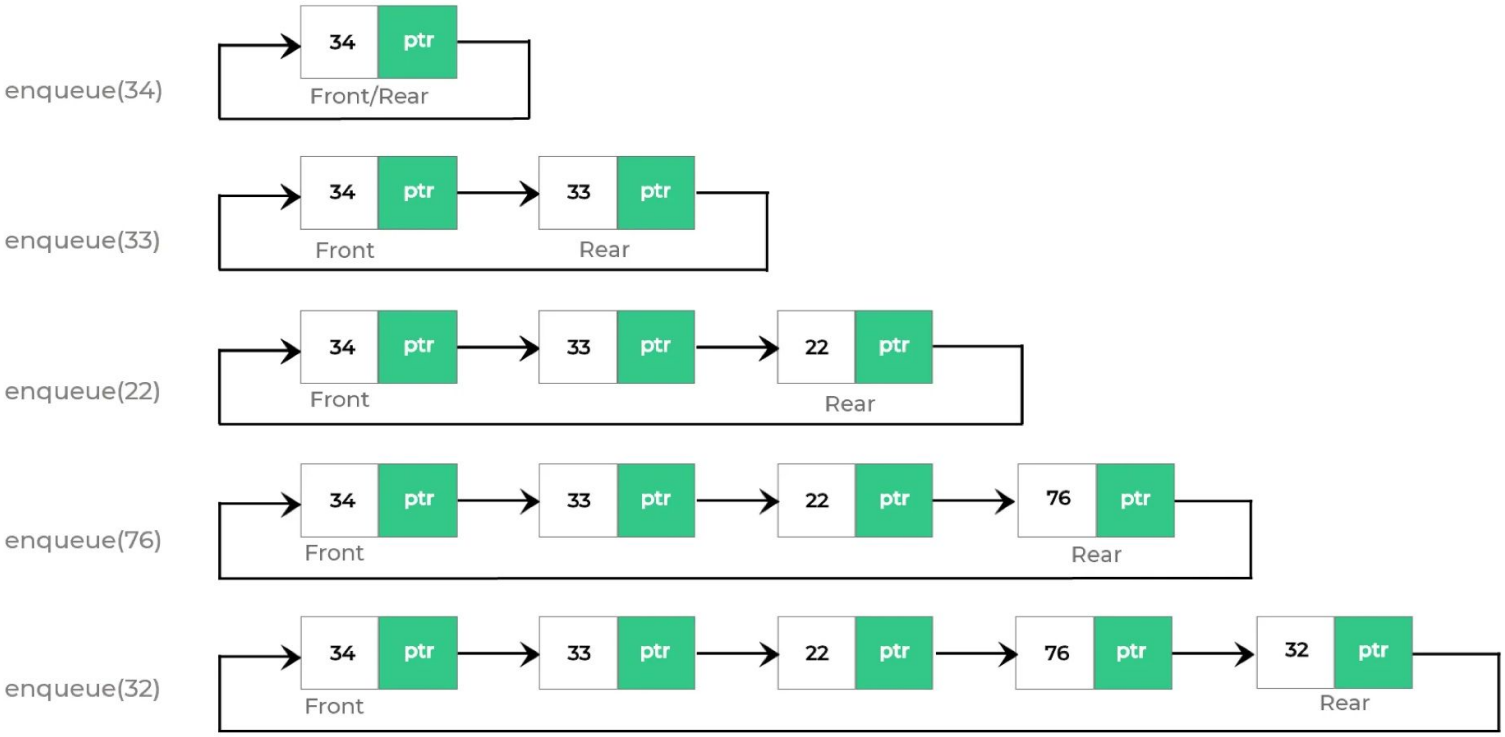
In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such **wastage of memory space** by shifting 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.**

CIRCULAR QUEUE

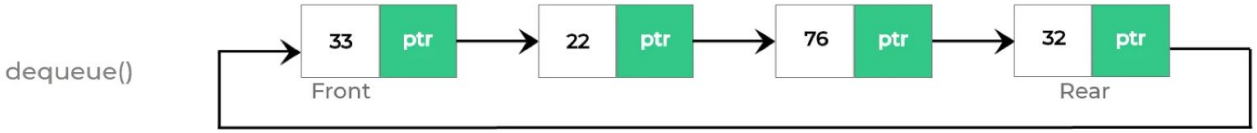
A circular queue is **similar to a linear queue** as it is also **based on the FIFO (First In First Out)** principle except that **the last position is connected to the first position in a circular queue that forms a circle.** It is also known as a ***Ring Buffer***.

Implementation of Circular Queues using Linked List in C

Adding the elements into Queue



Removing the elements from Queue



Implementing Circular Queue using Linked List in C

enqueue(data)

- Create a struct node type node.
- Insert the given data in the new node data section and NULL in address section.
- If Queue is empty then initialize front and rear from new node.
- Queue is not empty then initialize rear next and rear from new node
- New node next initialize from front

dequeue()

- Check if queue is empty or not.
- If queue is empty then dequeue is not possible.
- Else Initialize temp from front.
- If front is equal to the rear then initialize front and rear from null.
- Print data of temp and free temp memory.
- If there is more than one node in Queue then make front next to front then initialize rear next from front.
- Print temp and free temp.

print()

- Check if there is some data in the queue or not.
- If the queue is empty print “No data in the queue.”
- Else define a node pointer and initialize it with front.
- Print data of node pointer until the next of node pointer becomes NULL.

Linked representation of

- Circular queue
- Priority queue
- Deque

Linked Representation of Circular Queue

C CODE:
https://docs.google.com/document/d/10ePZus_Rgye9CY0HemJOLOOf8zWEpkaOkAQIE1hgITw/edit?usp=sharing

https://www.javatpoint.com/circular-queue

https://prepinsta.com/c-program/circular-queue-using-linked-list/

https://www.geeksforgeeks.org/introduction-to-circular-queue/ (theory)

PRIORITY QUEUE

1. It is collection of elements where elements are stored according to the their priority levels.
2. Inserting and removing of elements from queue is decided by the priority of the elements.
3. An element of the higher priority is processed first.
4. Two element of same priority are processed on first-come-first-served basis.

Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority.

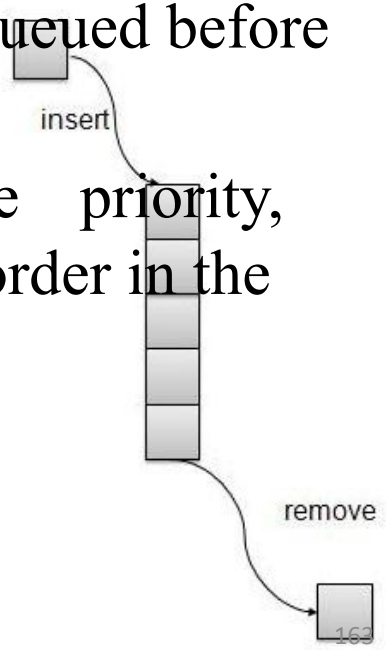
- If elements with the same priority occur, they are served according to their order in the queue.
- Generally, the value of the element itself is considered for assigning the priority.

Priority Queue

- Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference.
- In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa.

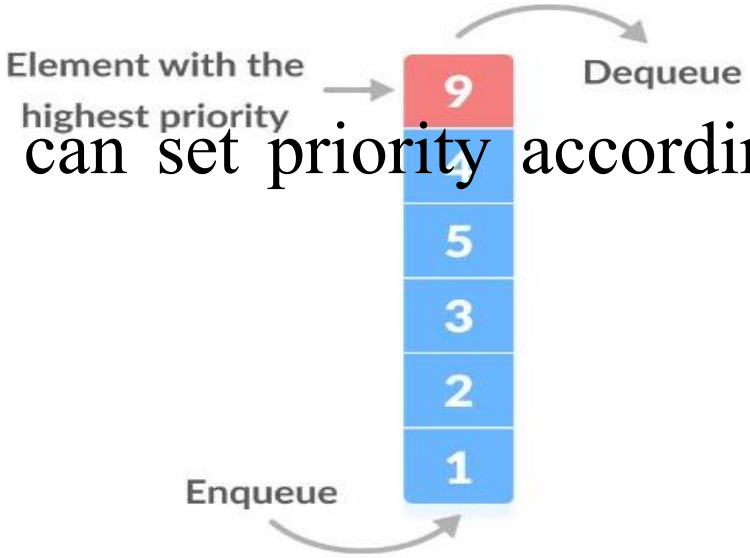
Priority Queue

- Priority Queue is an extensionof queue with following properties.
  - Every item has a priority associated with it.
  - An element with high priority is dequeued before an element with low priority.
  - If two elements have the same priority, they are served according to their order in the queue.



Priority Queue

- The element with the highest value is considered as the highest priority element.
- However, in other case, we can assume the element with the lowest value as the highest priority element.
- In other cases, we can set priority according to our need.



Priority Queue

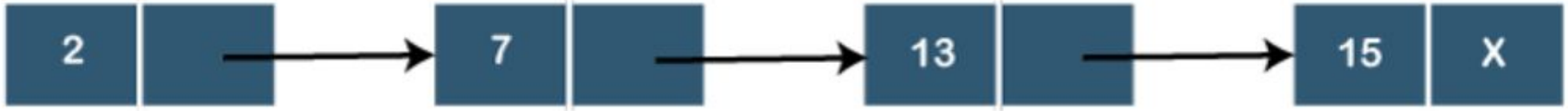
- Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a difference that the logical order of elements in the priority queue depends on the priority of the elements.
- The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue. Thus it is possible that when you enqueue an element at the back in the queue, it can move to front because of its highest priority.

Example: Suppose you have a few assignment from different subjects. Which assignment will you want to do first?

subjects	Due date	priority
DSGT	15 OCT	4
DLD	6 OCT	2
CYB	4 OCT	1
DS	8 OCT	3

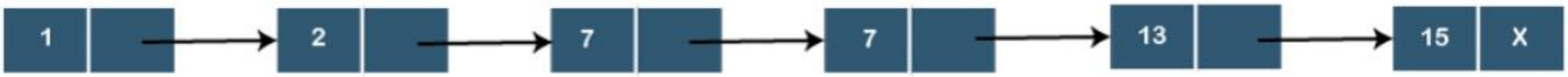
Let's understand through an example.

Consider the below-linked list that consists of elements 2, 7, 13, 15.



Suppose we want to add the node that contains the value 1. Since the value 1 has more priority than the other nodes so we will insert the node at the beginning of the list shown as below:

Now we have to add 7 element to the linked list. We will traverse the list to insert element 7. First, we will compare element 7 with 1; since 7 has lower priority than 1, so it will not be inserted before 7. Element 7 will be compared with the next node, i.e., 2; since element 7 has a lower priority than 2, it will not be inserted before 2.. Now, the element 7 is compared with a next element, i.e., since both the elements have the same priority so they will be served based on the first come first serve. The new element 7 will be added after the element 7 shown as below:



Linked Representation of Priority Queue

Implement Priority Queue using Linked Lists.

- **push():** This function is used to insert a new data into the queue.
- **pop():** This function removes the element with the highest priority from the queue.
- **peek() / top():** This function is used to get the highest priority element in the queue without removing it from the queue.

The list is so created so that the **highest priority element is always at the head of the list.** The list is arranged in **descending order of elements based on their priority.** This allow us to remove the highest priority element quickly. **To insert an element we must traverse the list and find the proper position to insert the node** so that the overall order of the priority queue is maintained.

Algorithm :

PUSH(HEAD, DATA, PRIORITY):

- ***Step 1:*** *Create new node with DATA and PRIORITY*
- ***Step 2:*** *Check if HEAD has lower priority. If true follow Steps 3-4 and end. Else goto Step 5.*
- ***Step 3:*** *NEW -> NEXT = HEAD*
- ***Step 4:*** *HEAD = NEW*
- ***Step 5:*** *Set TEMP to head of the list*
- ***Step 6:*** *While TEMP -> NEXT != NULL and TEMP -> NEXT -> PRIORITY > PRIORITY*
- ***Step 7:*** *TEMP = TEMP -> NEXT*

  [END OF LOOP]
- ***Step 8:*** *NEW -> NEXT = TEMP -> NEXT*
- ***Step 9:*** *TEMP -> NEXT = NEW*
- ***Step 10****: End*

POP(HEAD):

- ***Step 1:*** *Set the head of the list to the next node in the list. HEAD = HEAD -> NEXT.*
- ***Step 2:*** *Free the node at the head of the list*
- ***Step 3:*** *End*

PEEK(HEAD):

- ***Step 1:*** *Return HEAD -> DATA*
- ***Step 2:*** *End*

Linked Representation of Priority Queue

C CODE:
https://docs.google.com/document/d/10ePZus_Rgye9CY0HemJOLOOf8zWEpkaOkAQIE1hgITw/edit?usp=sharing

Linked Representation of Deque

Code in C++:
https://www.geeksforgeeks.org/implementation-deque-using-doubly-linked-list/

Application of Queues

1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2. When data is transferred asynchronously (data not necessarily received at the same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
3. Linear Queue: A linear queue is a type of queue where data elements are added to the end of the queue and removed from the front of the queue. Linear queues are used in applications where data elements need to be processed in the order in which they are received. Examples include printer queues and message queues.
4. Circular Queue: A circular queue is similar to a linear queue, but the end of the queue is connected to the front of the queue. This allows for efficient use of space in memory and can improve performance. Circular queues are used in applications where the data elements need to be processed in a circular fashion. Examples include CPU scheduling and memory management.
5. Priority Queue: A priority queue is a type of queue where each element is assigned a priority level. Elements with higher priority levels are processed before elements with lower priority levels. Priority queues are used in applications where certain tasks or data elements need to be processed with higher priority. Examples include operating system task scheduling and network packet scheduling.
6. Double-ended Queue: A double-ended queue, also known as a deque, is a type of queue where elements can be added or removed from either end of the queue. This allows for more flexibility in data processing and can be used in applications where elements need to be processed in multiple directions. Examples include job scheduling and searching algorithms.

Some common applications of Queue data structure :

1. **Task Scheduling**: Queues can be used to schedule tasks based on priority or the order in which they were received.
2. **Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time.
3. **Batch Processing**: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.
4. **Message Buffering**: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.
5. **Event Handling**: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.
6. **Traffic Management**: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.
7. **Operating systems:** Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.

8. **Network protocols:** Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.
9. **Printer queues :**In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.
10. **Web servers:** Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.
11. **Breadth-first search algorithm:** The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

Useful Applications of Queue

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at the same rate as sent) between two processes. Examples include IO Buffers, pipes, etc.

Applications of Queue in Operating systems:

- Semaphores
- FCFS (first come first serve) scheduling, example: FIFO queue
- Spooling in printers
- Buffer for devices like keyboard
- CPU Scheduling
- Memory management

Applications of Queue in Networks:

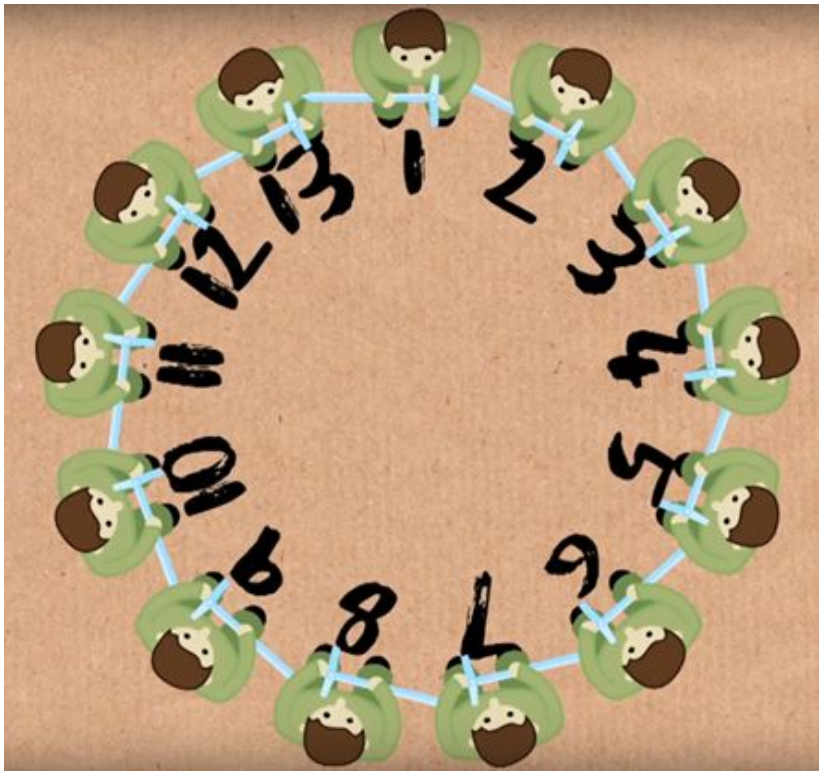
- Queues in routers/ switches
- Mail Queues
- **Variations:** (Deque, Priority Queue, Doubly Ended Priority Queue)

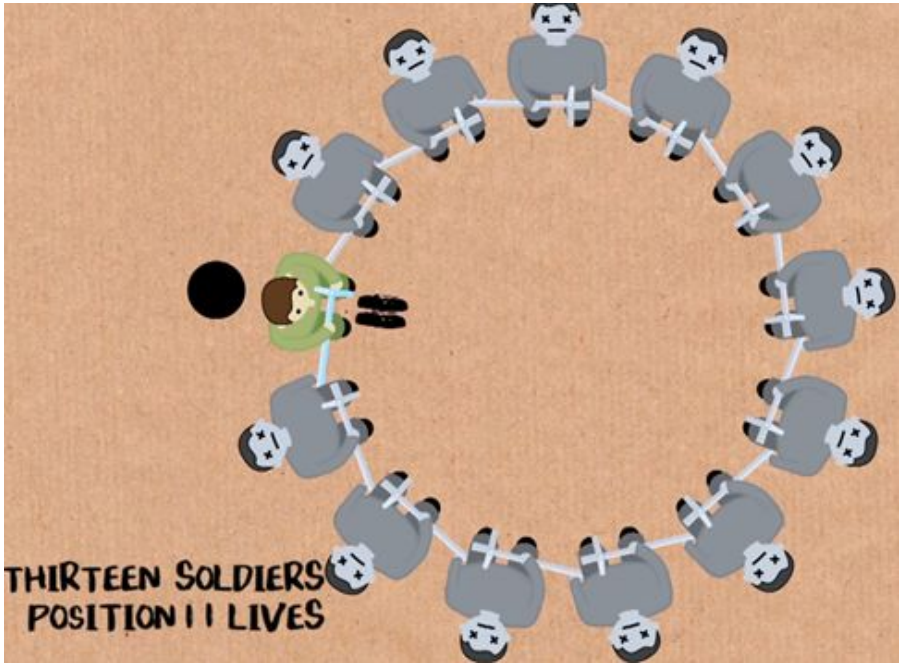
Some other applications of Queue:

- Applied as waiting lists for a single shared resource like CPU, Disk, and Printer.
- Applied as buffers on MP3 players and portable CD players.
- Applied on Operating system to handle the interruption.
- Applied to add a song at the end or to play from the front.
- Applied on WhatsApp when we send messages to our friends and they don't have an internet connection then these messages are queued on the server of WhatsApp.
- Traffic software (Each light gets on one by one after every time of interval of time.)

Application of Queue: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to





- History: N men surrounded by enemies. Preferred dying rather than captured as slaves. Every men killed the next living men until 1 man is left. That guy (Josephus) then surrendered (did not tell this initially 'cos others'd turn on him).

Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	1
2	1
3	3
4	1
5	3
6	5
7	7
8	1
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	1
17	3

*

Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	1
2	1
3	3
4	1
5	3
6	5
7	7
8	1
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	1
17	3

Pattern # 1

Winner always odd!

Makes sense as the first loop kills all the evens

Application: Josephus Problem

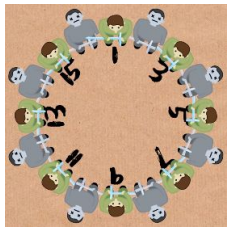
- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

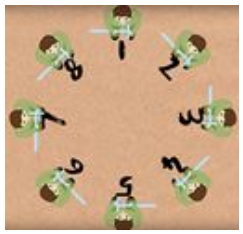
N	W(N)
1	**1**
2	**1**
3	3
4	**1**
5	3
6	5
7	7
8	**1**
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	**1**
17	3

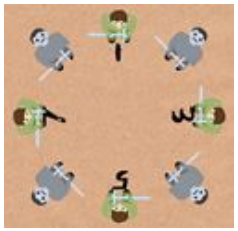
Pattern # 2

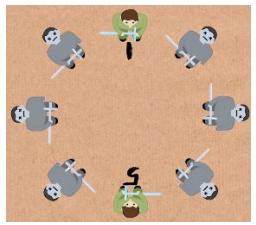
Jump by 2; reset at 2^a for some a!

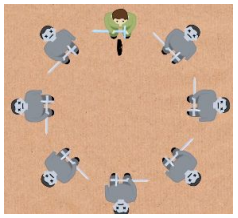
Makes sense: Assume 2^a men in circle. 1 pass removes half of them; at 1; repeat on 2^{a-1} men; so winner is the starting point (1)











*

Application: Josephus Problem

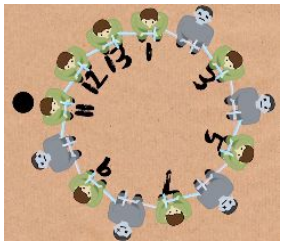
- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	**1**
2	**1**
3	3
4	**1**
5	3
6	5
7	7
8	**1**
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	**1**
17	3

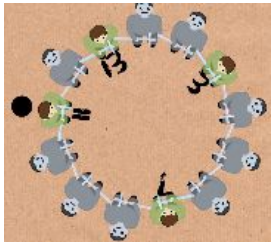
Pattern # 2 (cont'd)

Jump by 2; reset at 2^a for some a!

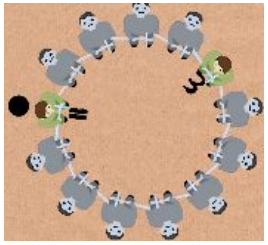
Makes sense: Assume $2^a + b$ men in circle, where a is the biggest possible power; hence $b < 2^a$ (binary notation idea); after b men we are left with 2^a men, whose winner is the starting point (11 for below)



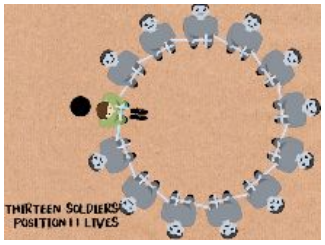
8 left



4 left



2 left



1 left

Application: Josephus Problem

- N men. Circular arrangement. Kill every second. Where to sit to survive?
- List winners/survivors for each N to see a pattern

N	W(N)
1	**1**
2	**1**
3	3
4	**1**
5	3
6	5
7	7
8	**1**
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	**1**
17	3

Pattern # 2 (cont'd)

Jump by 2; reset at 2^a for some a!

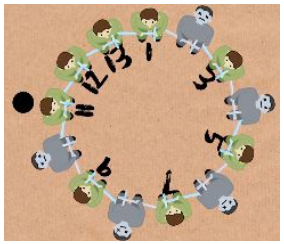
So what is 11 for N=13?

$N = 2^3 + 5$ (a=3, b=5); and $11 = 2*5 + 1$

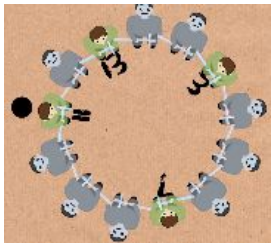
In general, after b steps, we arrive at the position $2*b + 1$ (every 2^{nd} is killed). Hence,

$$W(N) = 2*b + 1$$

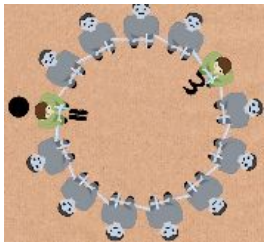
where $N = 2^a + b$ and $b < 2^a$



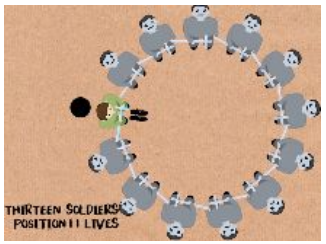
8 left



4 left



2 left



1 left

Application: Josephus Problem

- Based on W(N) = 2*b + 1 where $N = 2^a + b$ and $b < 2^a$ we can write a simple program that returns the winner/survivor index

```
int W(int N) {
  int a = 0;
  while (N > 1) {
     N /= 2;
     a++;
  }
  return 2*(N – pow(2, a)) + 1;
  //or you could compute pow(2, a) in variable V like this:
  //int V = 1; for (int i = 0; i < a; i++) V *= 2;
}
```

Application: Josephus Problem

- Based on W(N) = 2*b + 1 where $N = 2^a + b$ and $b < 2^a$ we can write a simple program that returns the winner/survivor index
- If you do not like math and cannot extract W(N) above, you can write the code using a Queue
- Math gets way complicated for the generic problem where you kill every k^{th} man where k > 1

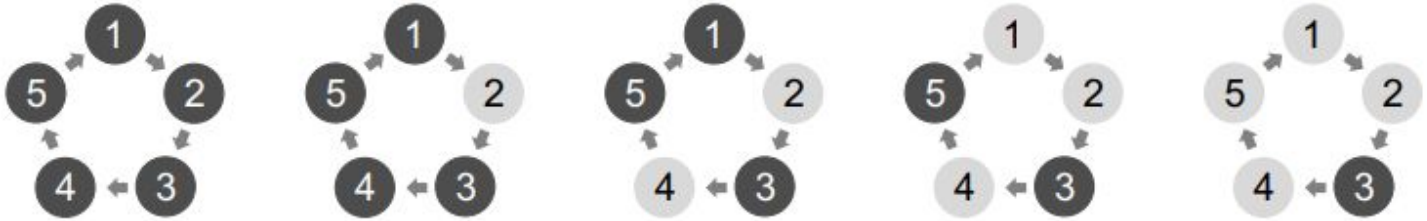
```
int Josephus(Q, k) { //Queue Q is built in advance with e.g., 1, 2, 3, 4, 5, 6.
  while (Q.size() > 1) {
    for (i = 1; i <= k-1; i++) //skip the k-1 men without killing
      Q.enqueue( Q.dequeue() );
    killed = Q.dequeue();
  }
* return Q.dequeue(); } //only one left in the Q, the winner ☺
```

Josephus Problem

Let us see how queues can be used for finding a solution to the Josephus problem.
In Josephus problem, `n` people stand in a circle waiting to be executed. The counting starts at some point in the circle and proceeds in a specific direction around the circle. In each step, a certain number of people are skipped and the next person is executed (or eliminated). The elimination of people makes the circle smaller and smaller. At the last step, only one person remains who is declared the 'winner'.

Therefore, if there are `n` number of people and a number `k` which indicates that `k-1` people are skipped and `k-th` person in the circle is eliminated, then the problem is to choose a position in the initial circle so that the given person becomes the winner.

For example, if there are 5 (`n`) people and every second (`k`) person is eliminated, then first the person at position 2 is eliminated followed by the person at position 4 followed by person at position 1 and finally the person at position 5 is eliminated. Therefore, the person at position 3 becomes the winner.



Try the same process with n = 7 and k =3. You will find that person at position 4 is the winner. The elimination goes in the sequence of 3, 6, 2, 7, 5 and 1.

7. Write a program which finds the solution of Josephus problem using a circular linked list.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
        int player_id;
        struct node *next;
};
struct node *start, *ptr, *new_node;

int main()
{
        int n, k, i, count;
        clrscr();
        printf("\n Enter the number of players : ");
        scanf("%d", &n);
        printf("\n Enter the value of k (every kth player gets eliminated): ");
        scanf("%d", &k);
        // Create circular linked list containing all the players
        start = malloc(sizeof(struct node));
        start->player_id = 1;
        ptr = start;
        for (i = 2; i <= n; i++)
        {
                new_node = malloc(sizeof(struct node));
                ptr->next = new_node;
                new_node->player_id = i;
                new_node->next=start;
                ptr=new_node;
        }
        for (count = n; count > 1; count--)
        {
                for (i = 0; i < k - 1; ++i)
                        ptr = ptr->next;
                ptr->next = ptr->next->next; // Remove the eliminated player from the
circular linked list
        }
        printf("\n The Winner is Player %d", ptr->player_id);
        getche();
        return 0;
}
```

\\getche() function reads a single character from the keyboard and displays immediately on the output screen without waiting for enter key.

Output

```
Enter the number of players : 5
Enter the value of k (every kth player gets eliminated): 2
The Winner is Player 3
```

Any questions?