Lecture 22
Queue

November 02, 2021
Tuesday

QUEUE FUNDAMENTALS

- A Queue is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front.
- In Queue both ends are used.
  - One for adding new elements.
  - One for removing the existing elements.
- The last element has to wait for the elements preceding it on the queue to be removed.
- In simpler words it follows **First In First Out (FIFO)** order.

QUEUE VISUALIZATION

QUEUE OPERATIONS

- A queue can be defined in terms of operations
  - that can change its status
  - that can check its status
- A Queue operations are similar to stack operations.
  - *clear ( );*
  - *isEmpty ( );*
  - *push (element)* now becomes *enqueue (element);*
  - *pop ( )* will be called *dequeue ( );*
  - *peek ( );*

Linked List Implementation of Queue

IMPLEMENTATION | CLEAR ( )

```
void clear ( )
 {
        if ( !isEmpty ( ) )
        {
                while (front != NULL)
                {
                        temp = front;
                        front = front->prev;
                        delete temp;
                        temp = NULL;
                }
        }
}
```

clear ( );

- clears the Queue, all elements are deleted.

IMPLEMENTATION | ENQUEUE ( )

```
void enqueue ( int element ) {
        Node *temp = new Node ( element );
        if ( temp ) {
                if ( rear == NULL ) {
                        front = rear = temp;
                        return ;
                }
                rear -> prev = temp;
                rear = temp;
        } else
                cout << "Queue is FULL..!";
}
```

enqueue (element);

- Adds an element to the rear side of the Queue

IMPLEMENTATION | DEQUEUE ( )

```
int dequeue ( ) {
        if ( front ) {
                int value = front -> info;
                Node *temp = front;
                front = front -> prev;

                if ( front == NULL )
                        rear = NULL;

                delete temp;
                return value;

        } else {    cout << "Queue is Empty..!";
                    return -1;       }
}
```

dequeue ( );

- Removes an element from the front of Queue

IMPLEMENTATION | PEEK ( )

```
int peek ( ) {
        int value = -1;
        if ( front ) {
                value = front -> info;
        }
         return value;
}
```

peek ( );

- Returns the value of front without removing it from the Queue.

isEmpty ( )

isEmpty ( );

Check to see if the

Queue is empty?

```
int isEmpty ( )
{
        return front == NULL;
}
```

Array Implementation of Queue

IMPLEMENTATION | CLEAR ( )

```
void clear ( )
 {
        if ( !isEmpty ( ) )
        {
                delete [ ] queue;
                front = rear = -1;
        }
}
```

clear ( );

- clears the Queue, all elements are deleted.

IMPLEMENTATION | ENQUEUE ( )

```
void enqueue ( int element ) {
      if ( queue ) {
            if ( rear < length) {
                  queue [++rear] = element;
                  if ( front  ==  -1)
                        front = rear;
                  return ;
            }
      } else
            cout << "Queue is FULL..!";
}
```

enqueue (element);

- Adds an element to the rear side of the Queue

IMPLEMENTATION | DEQUEUE ( )

```
int dequeue ( ) {
        int value = -1;
        if ( queue ) {
                if (front != -1) {
                        value = queue [ front ];
                        for (int i = 0; i <= rear; i++)
                                queue [ i ] = queue [ i + 1];
                        rear--;
                }
        } else {    cout << "Queue is Empty..!";     }
        return value;
}
```

dequeue ( );

- Removes an element from the front of Queue

IMPLEMENTATION | PEEK ( )

```
int peek ( ) {
        int value = -1;
        if ( queue ) {
                if ( front != -1 )
                value = queue [ front];
        }
         return value;
}
```

peek ( );

- Returns the value of front without removing it from the Queue.

isEmpty ( )

isEmpty ( );

Check to see if the

Queue is empty?

```
int isEmpty ( )
{
        return front == -1;
}
```

ARRAYS LIMITATIONS

- We have to move n-1 items on each dequeue.
- If we don’t move n-1 items,
  - front will simply move forward and space will be wasted.
  - Since, we cannot add data at this end.

Circular Array

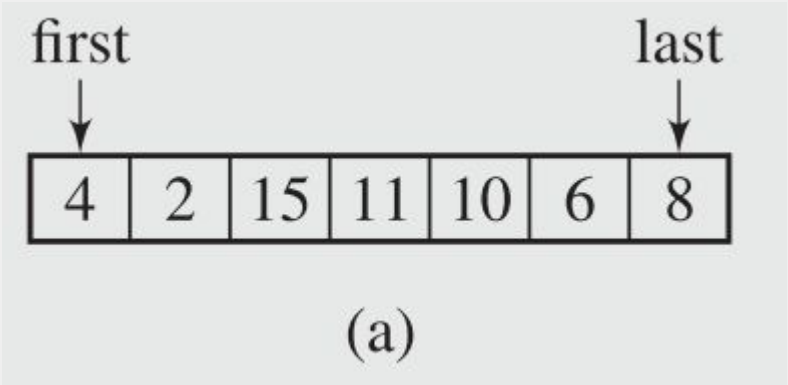
CIRCULAR ARRAY

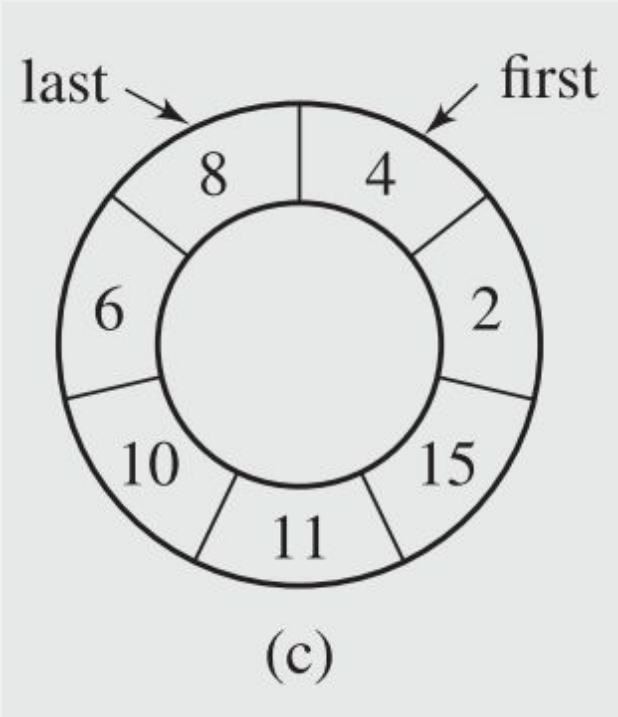
- We don’t move n-1 items and simply increment the front index.
  - The empty cells can be used to enqueue new elements.
  - Since, we cannot add data at this end.

ARRAYS LIMITATIONS

- We have to move n-1 items on each dequeue.
- If we don’t move n-1 items,
  - front will simply move forward and space will be wasted.
  - In this way the end of the queue can occur at the beginning of the array
- This can be better visualized as Circular Array

ARRAYS LIMITATIONS

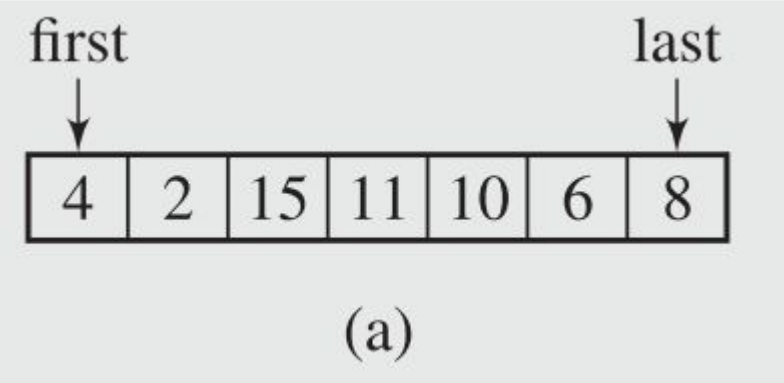


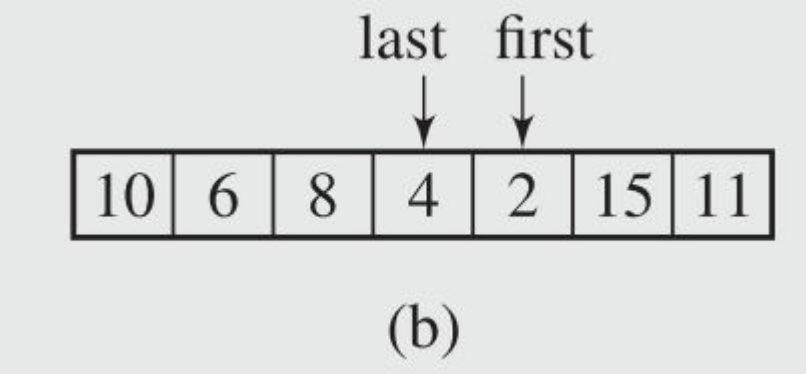


CIRCULAR QUEUE FULL

- Since, we are using a normal array, we have to decide when Queue will be full.
  - The queue is full if the first element immediately precedes the last element in the counterclockwise direction.
  - The queue is full if either the first element is in the first cell AND the last element is in the last cell.

ARRAYS LIMITATIONS





IMPLEMENTATION | ENQUEUE ( )

```
void enqueue ( int element ) {

    if ( ( front == 0 && rear == size - 1) || ( rear == front - 1) )
        cout << " Queue is Full…";
    else if ( front == -1 ) {
        front = rear = 0;
        queue [ front ] = el;
    } else {
        queue [++rear ] = el;
    }

}
```

IMPLEMENTATION | DEQUEUE ( )

```
int dequeue ( int element ) {
        int value = -1;
        if ( front == -1 ) {
                cout<< " Queue is Empty…!";
                return value;
        }
        value = queue [ front ];     queue [front ] = -1;
        if (front == rear ){
                front = rear = -1;
        } else if ( front == size -1 ){      front = 0;      }
        else front++;
        return value;
}
```