CSCI 2270: Data Structures

Recitation #5 (Section 101)

Office Hours

- Name: Himanshu Gupta
  Email: himanshu.gupta@colorado.edu

- **Office Hours - 10am to 2pm on Mondays in ECAE 128**
  - In case that doesn't work for you, shoot me an email. We will figure something out that works for both of us.
  - Also, you can attend any TA's office hours. Timings are available on moodle in calendar.

- Need extra office hours on this Friday for Assignment 4?

Logistics

- Midterm 1 on **Feb 21, 2020 from 5pm to 7pm**
  - Details about location available on Moodle

- Conflict makeup exam
  - On Feb 19, 2020 from 5pm to 7pm
  - Please ensure that you have filled the form (link to the form)
  - **Deadline to fill the form - Feb 13, 2020 11:59pm**

- Special Accommodation
  - On Feb 21, 2020 from 5pm to 7pm
  - Please ensure that you have filled the form (link to the form)
  - **Deadline to fill the form - Feb 14, 2020 11:59pm**

Logistics

- Practice Midterm 1 is out
  - Solve it. We will discuss it in the next recitation.

- Next recitation - Reviewing midterm material
  - Any specific topic that you guys want me to review properly?
  - Shoot me an email if you want me to cover something specific.

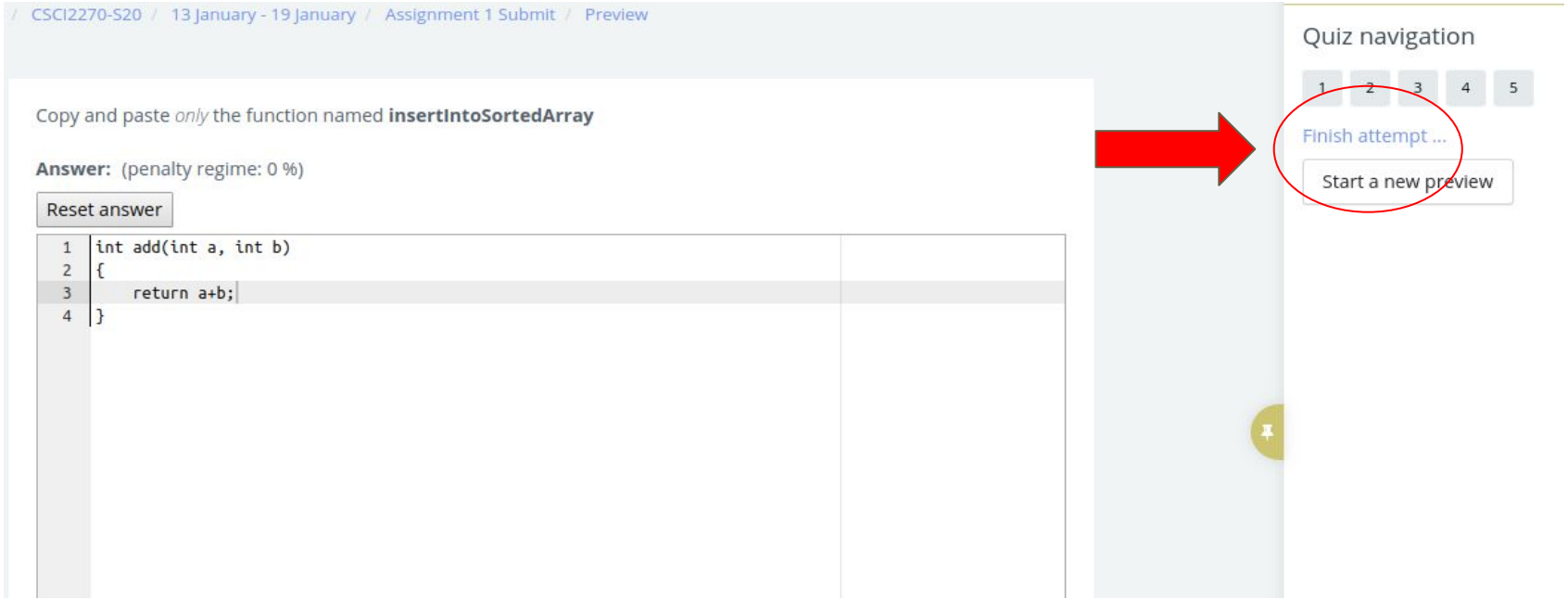
- **Assignment 4 is due on Sunday, February 16 2020, 11:59 PM. GOOD LUCK!**
  - Any questions on that?

Logistics

- **Fill out FCQs**
  - You will have an email in your inbox.
  - Search your inbox with the keyword “FCQ” or “Rajshree Shrestha”

- **Your feedback is really valuable. It helps me evaluate myself.**
  - **Constructive criticism is always welcome!**

Please click on “Finish Attempt” after you are done!



Any questions on Logistics?

Today's Agenda

- Review (20 ~ 30 mins)
  - Assignment 4 (hints)
  - Stacks
  - Array and Linked List Implementation of a Stack
  - Queues
  - Array and Linked List Implementation of a Queue

- Exercise
  - Silver Problem - Complete the enqueue and dequeue operations for linked list implementation of a queue.
  - Gold Problem - Check if parentheses is balanced in an input string

Assignment 4

Assignment 4

- readjustNetwork()

Assignment 4

void readjustNetwork(int startIndex, int endIndex);

- ➔ Manipulate **next** pointers to readjust the linked list. Here, ***startIndex*** is index of a node from starting. Similarly ***endIndex*** is index of a node from beginning. The function will send the chunk of the link list between start index and end index at the end of the linked list. Consider the node at head as index 0.

  For example, if you have linked list like this: "`A -> B -> C -> D -> E-> NULL`", and ***startIndex=1 and endIndex=3***, then the linked list after readjustNetwork should be "`A -> E -> B -> C -> D-> NULL`".

  If you have linked list like this: "`A -> B -> C -> D -> NULL`", and ***startIndex=0 and endIndex=2***, then the linked list after readjustNetwork should be "`D-> A -> B -> C -> NULL`". Here, "D" is the new head.
- ➔ If the linked list is empty, print "*Linked List is Empty*".
- ➔ If ***endIndex*** is bigger than the number of nodes in the linked list or smaller than ***0***, then print "*Invalid end index*".
- ➔ ***endIndex*** should be lesser than the index of the last element in the linked list. Otherwise print "*Invalid end index*".
- ➔ If ***startIndex*** is bigger than the number of nodes in the linked list or smaller than ***0***, then print "*Invalid start index*".
- ➔ If ***startIndex > endIndex*** print "Invalid indices".

What's a stack?

What's a stack?

- It is just a pile of elements that is properly arranged.

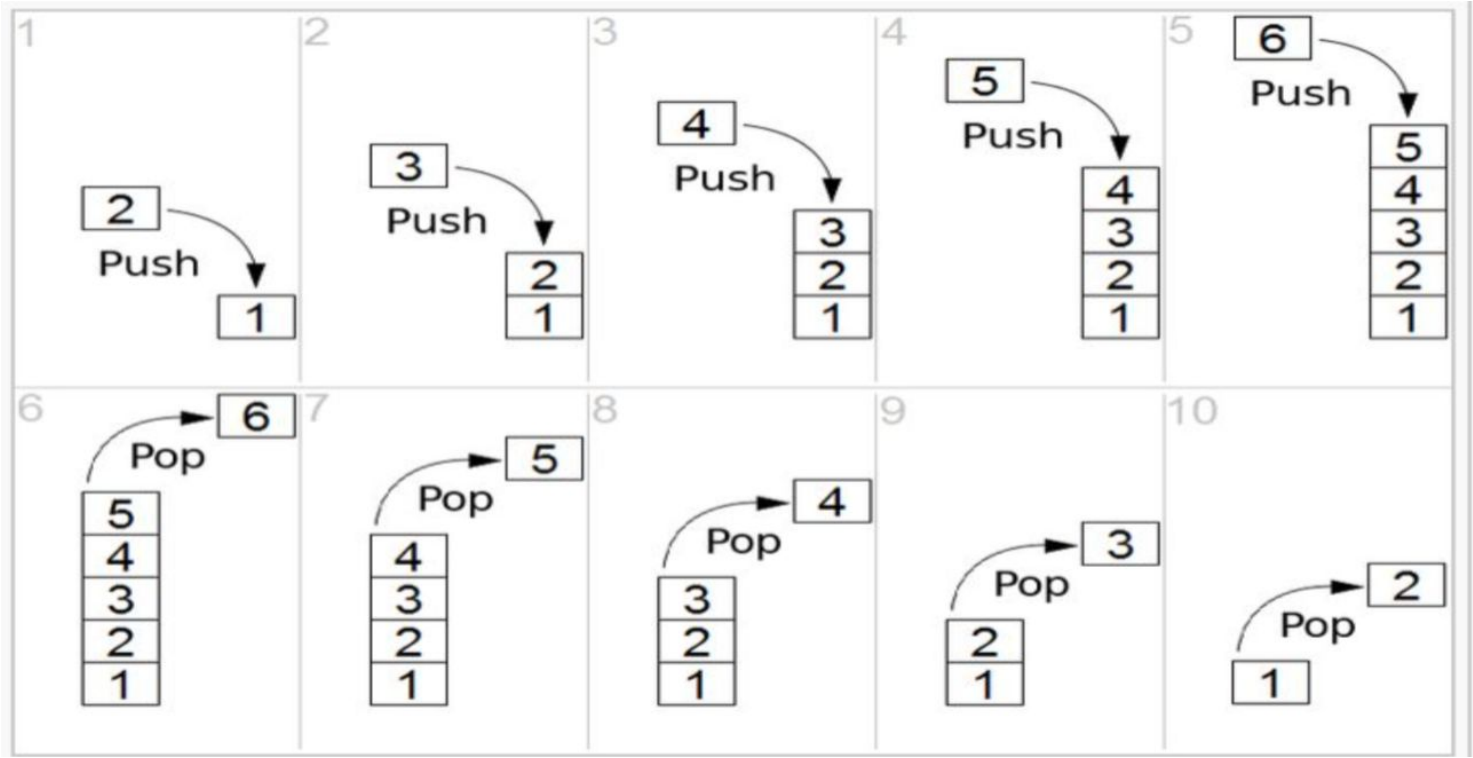
Stacks

- Formal definition - It is a linear data structure which follows a particular order in which the elements are inserted or removed.
  - Can only perform operations from one end. Generally called “TOP”

- LIFO - Last In First Out
  - The element that was inserted last in a stack will be removed first from the stack.
  - This also implies that the element that was inserted first in a stack will be removed last from the stack.

- Typical stack operations:
  - Push - Insert an element into your stack
  - Pop - Remove an element into your stack
  - isFull - Is the stack full?
  - isEmpty - Is the stack empty?
  - Peek - What’s the most recent value that was entered?

Stacks

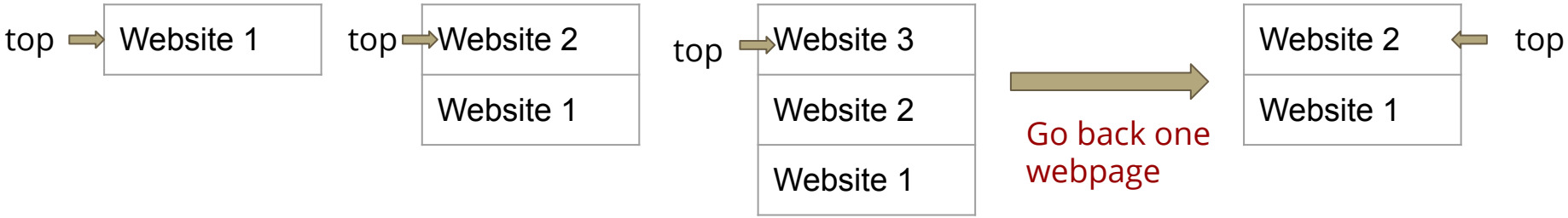

1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
6
Pop
5
4
3
2
1
7
5
Pop
4
3
2
1
8
4
Pop
3
2
1
9
3
Pop
2
1
10
2
Pop
1

Stacks

- Any cool application of stacks?
  - (Hint: We make use of it everyday on our browsers)

Stacks

- Any cool application of stacks?
  - (Hint: We make use of it everyday on browsers)
- Well, how do you think a browser keeps track of the web pages you have visited?
  - Click on the back button and it pops the last visited website from your stack



Stacks

- How does a stack look like?
- **top** always used to refer to the topmost element in a stack.

```
class Stack
{
      private:
            top  //top of the stack
            data //stack data (in array or list)
      public:
            init()
            push(value)
            pop()
            peek()
            isFull()
            isEmpty()
}
```

Array Implementation of a Stack

- Elements are stored in an array but while performing push and pop operations, ensure LIFO order is followed.

- Initialize variable **top** to 0

- **top** of the stack refers to the index of the array where the next element will be added. All the insert/delete operations occur using **top**
  - When stack is empty, top = 0
  - When stack is full, top = maximum size of the array
  - Overflow error when top> maximum size of the array

Array Implementation of a Stack

- **isFull()**

```
bool stack::isFull()
{
        return top == max_size;
}
```

```
//If max_size is 10, top can only take values from 0 to 9
//As soon as top becomes 10, we know the array can't store more elements
```

Array Implementation of a Stack

- **isEmpty()**

```
bool stack::isEmpty()
{
        return top == 0;
}
```

```
//If top==0, then there are no elements in your stack
```

Array Implementation of a Stack

- **peek()**

```
int stack::peek(){
if (!isEmpty())
        return arr[top-1];
else{
        cout<<"Stack is empty"<<endl;
        exit(EXIT_FAILURE);
}
```

//Trying to peek when the stack is empty is an edge case and should be covered,

Array Implementation of a Stack

- **push()**

```
void stack::push(int x)
{
    if (isFull()){
        cout << " Stack Overflow"<<endl;
        return;
    }
    cout << "Inserting " << x << endl;
    data[top] = x;   //Store x at position “top” in array data
    top++;           //Increment “top” by one
    return;
}
```

Array Implementation of a Stack

- **pop()**

```
int stack::pop()
{
        if ( isEmpty() ){
                cout << "Stack UnderFlow"<<endl;
                return -1;
        }
        cout << "Removing " << peek() << endl;
        //decrease stack size by 1 and return the popped element
        top = top-1;
        return data[top];
}
```

Linked List Implementation of a Stack

- Why is there a linked list implementation?
  - because now the size can grow and shrink according to the needs at runtime. (unlike arrays of constant size)

- **top** here is a pointer that always points to the head of the linked list.

- Push() in the Linked list implementation of a stack:
  Every new element is inserted at the head of the list. So, every new element is pointed by the top pointer.

- Pop() in the Linked list implementation of a stack:
  To remove an element, simply remove the node pointed by the top pointer, and make top point to the next node in the list.

Linked List Implementation of a Stack

- Push()

  Equivalent to adding a new node at the beginning of a linked list.

Push() in linked list implementation of a stack

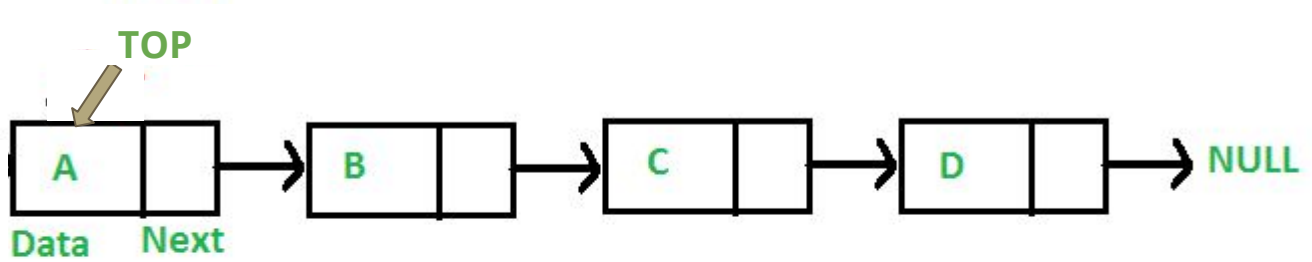
```
//Create a new node
Node* newNode = new Node;
newNode->key = newKey;
//Make it point to the current top (head) of the LL
newNode->next = top;
//Make your top point to the new node
top = newNode;



//Cover the edge case when the stack is empty
```

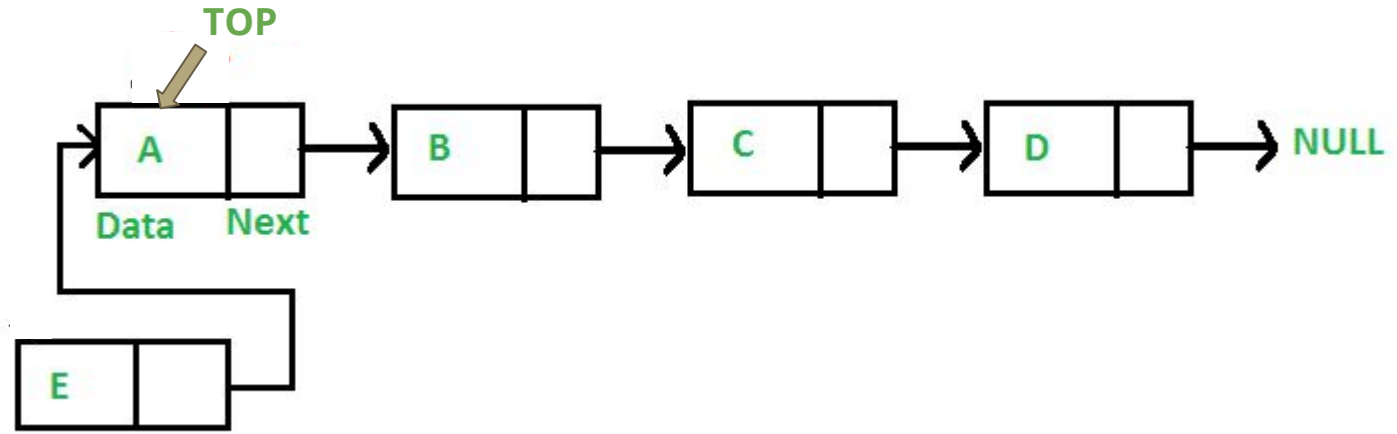
Push() in linked list implementation of a stack

STEP 1



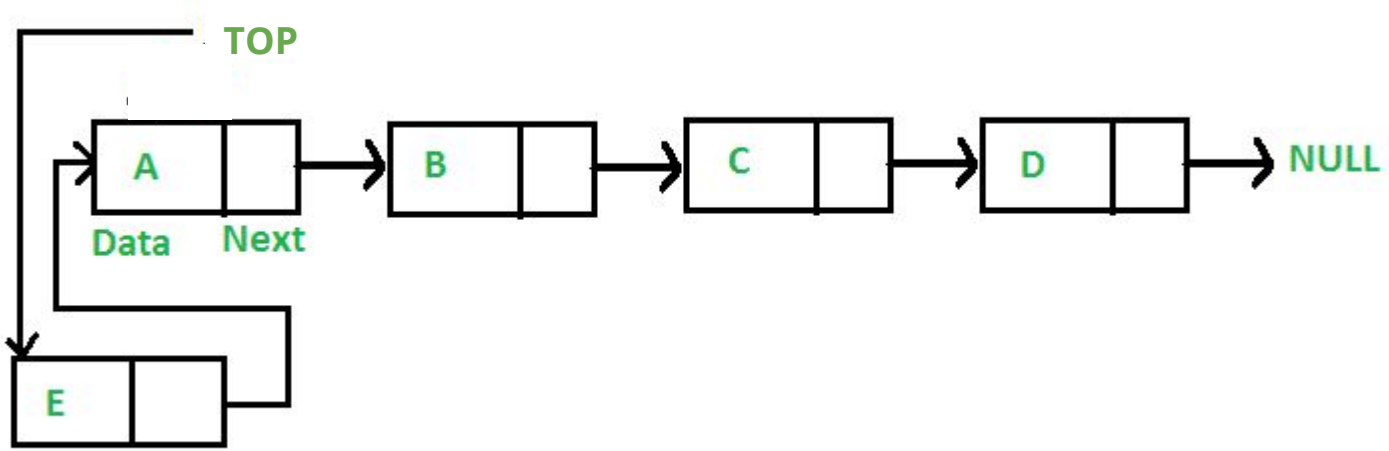
Push() in linked list implementation of a stack

STEP 2



Push() in linked list implementation of a stack

STEP 3



Linked List Implementation of a Stack

- Pop()

  Equivalent to deleting at the beginning of a linked list.

Pop() in linked list implementation of a stack

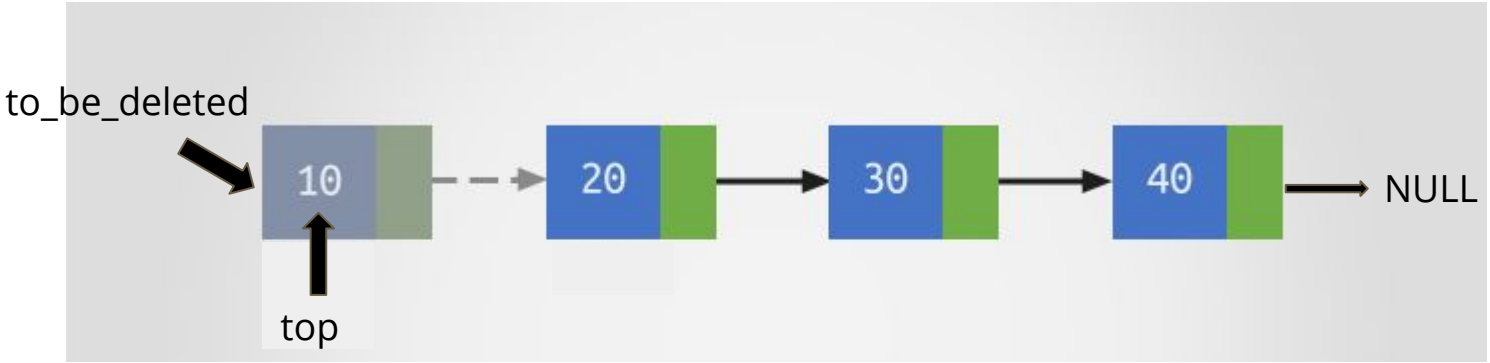
```
//Create a new pointer and make it point to the first node in the LL
Node* to_be_deleted = top;
//Make your head point to next node of the LL (this is the 2nd node in LL )
top = top->next;
//delete your to_be_deleted node
delete to_be_deleted;



//Cover the edge case when the stack is empty
```

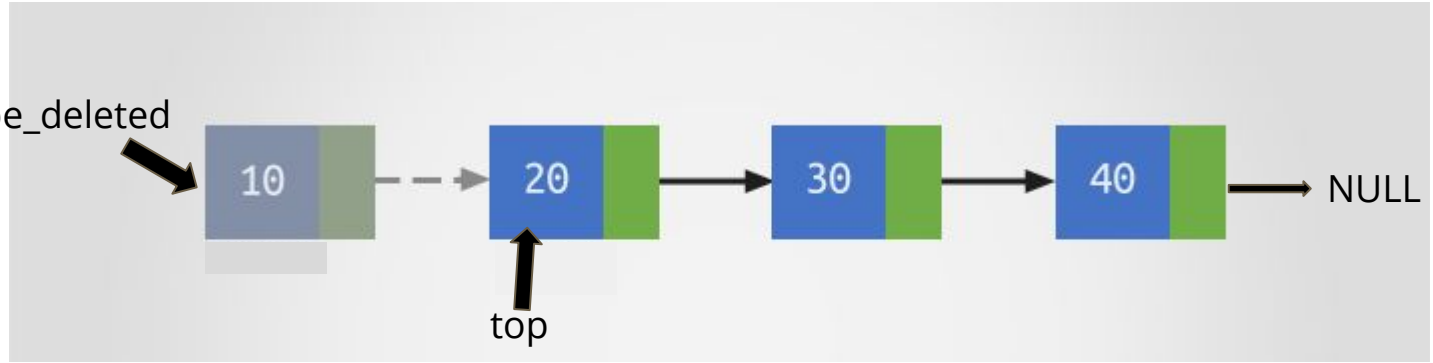
Pop() in linked list implementation of a stack

STEP 1



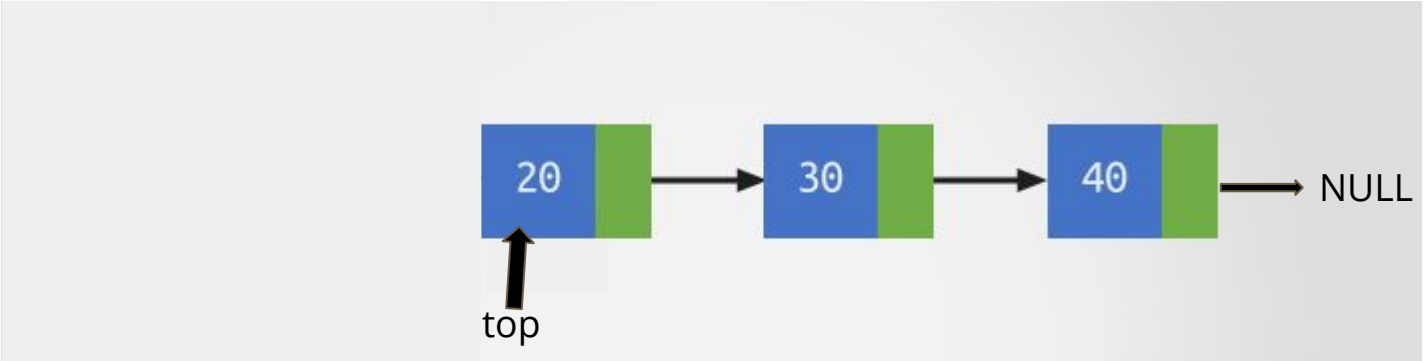
Pop() in linked list implementation of a stack

STEP 2



Pop() in linked list implementation of a stack

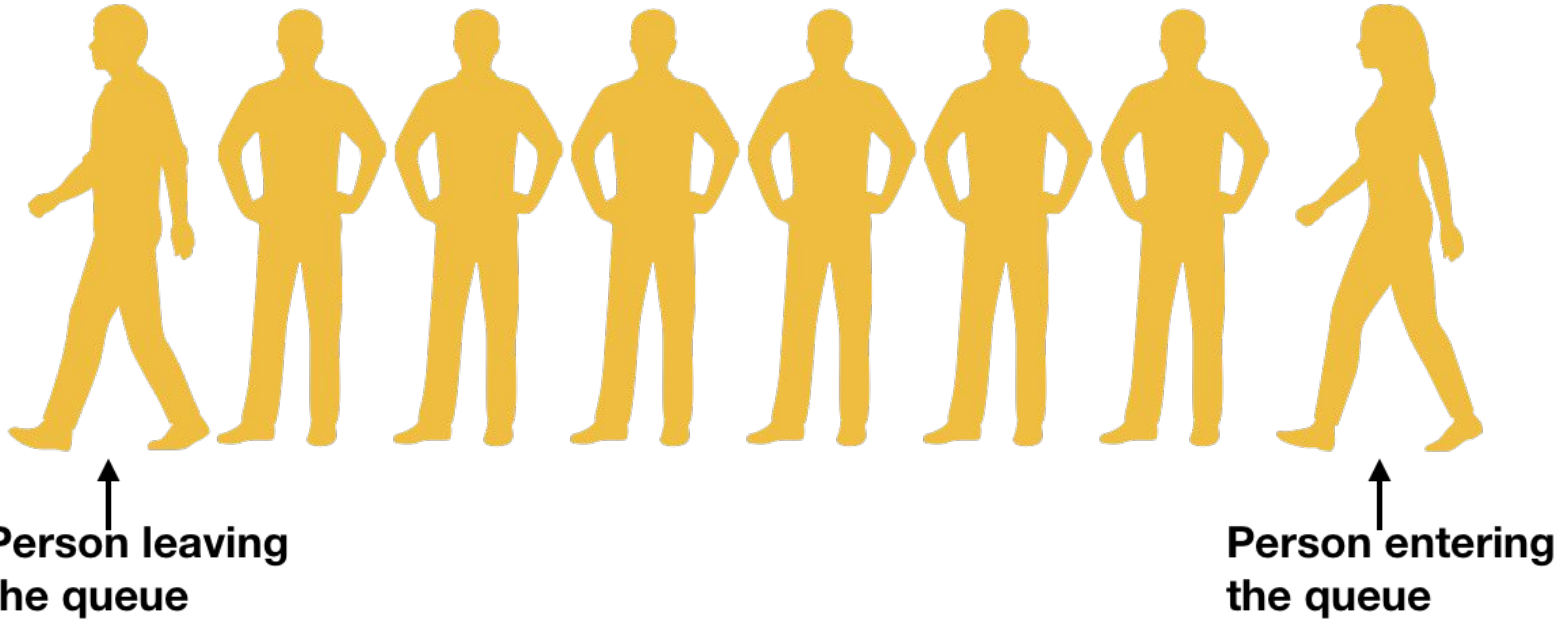
STEP 3



Any questions?

What is a queue?

What is a queue?



Queue

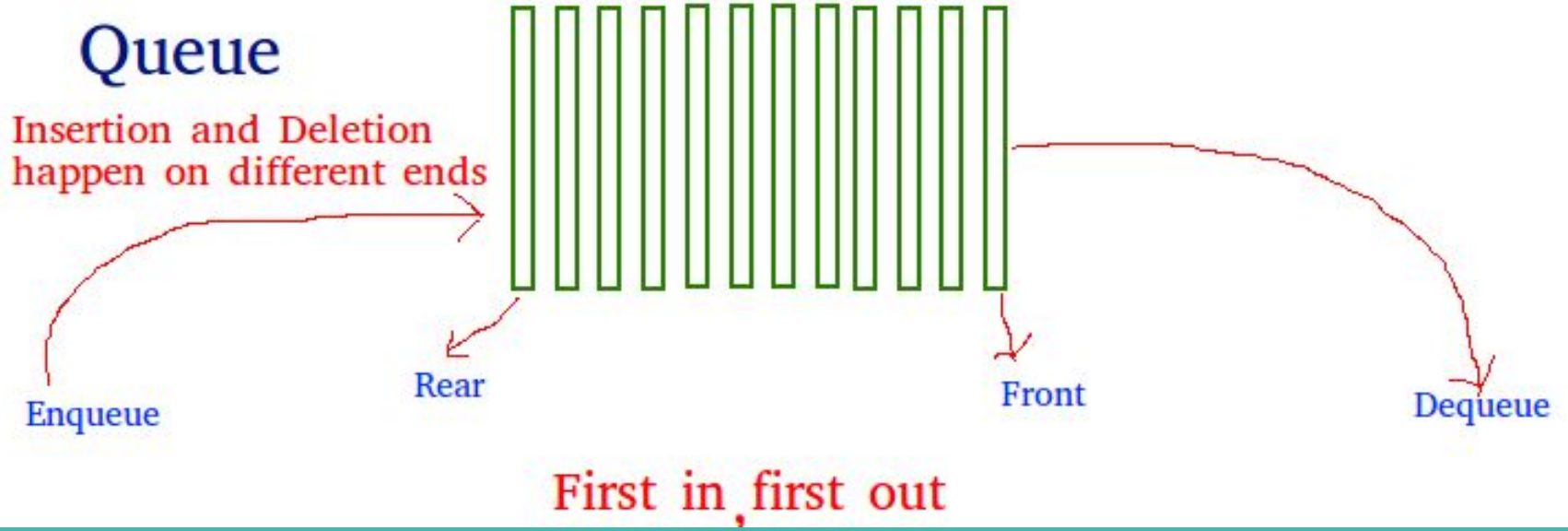
- Formal definition - It is a linear data structure which follows a particular order in which the elements are inserted or removed.
  - Can perform operations from two ends.
  - Generally called (“FRONT” and “REAR”) or (“FRONT” and “END”) or (“HEAD” and “TAIL”)

- FIFO - First In First Out
  - The element that was inserted first in a queue will be removed first from the queue.
  - This also implies that the element that was inserted last in a queue will be removed last from the queue.

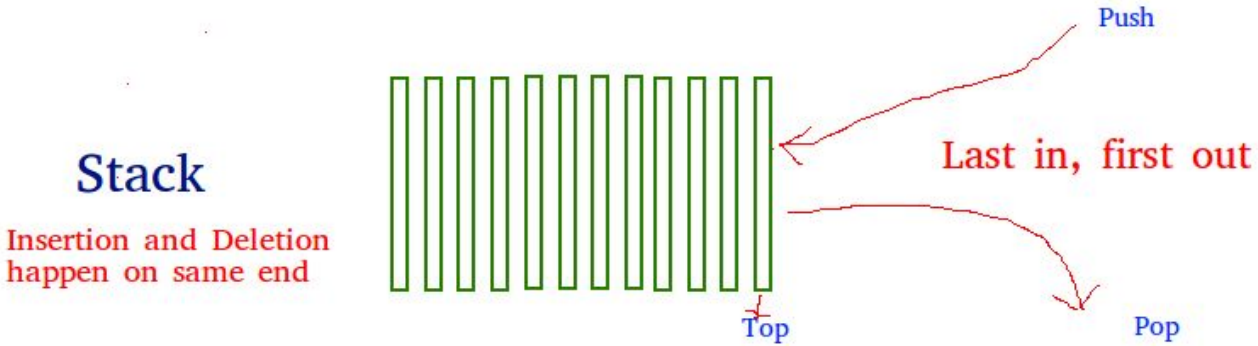
- Typical queue operations:
  - Enqueue - Insert an element in your queue
  - Dequeue - Remove an element from your queue
  - Peek - Returns the front element present in the queue without dequeuing it.
  - isFull - Is the queue full?
  - isEmpty - Is the queue empty?

Queue

- Insertion occurs at the Rear end and Deletion occurs at the Front end
- You remove from the front (head) of the queue.
- You insert at the rear (tail) of the queue.



Stack vs Queue



First in, first out

Queues

- How does a queue look like?
- Element always removed using the **front** end of the queue. (Enqueue)
- Element always added using the **rear** end of the queue. (Dequeue)

```
class Queue
{
      private:
            front
            rear
            queueSize
            capacity
      public:
           init()
           enqueue(value)
           dequeue()
           peek()
           isFull()
           isEmpty()
}
```

Array implementation of a Queue

- Elements are stored in an array but while performing enqueue and dequeue operations, ensure FIFO order is followed.
- Initialize variable **front** to 0 and **rear** to 0
- All the enqueue operations occur using **rear** and all the dequeue operations occur using **front**

Array Implementation of a Queue

- **isFull()**

```
bool queue::isFull()
{
    return (queueSize == capacity);
}
```

```
//If capacity is 10, queueSize can only take values from 0 to 9
//As soon as queueSize becomes 10, the array can't store more elements
```

Array Implementation of a Queue

- **isEmpty()**

```
bool queue::isEmpty()
{
        return queueSize == 0;
}
```

//If queueSize==0, then there are no elements in your queue

Array Implementation of a Queue

- **enqueue()**

```
void enqueue::push(int x)
{
    if (isFull())
        return;
    rear = (rear + 1) % capacity;
    data[rear] = item;
    queueSize = queueSize + 1;
    cout << item << " enqueued to queue\n";
    return;
}
```

Array Implementation of a Queue

- dequeue()

```
int queue::dequeue()
{
   if (isEmpty()){
      cout<<”Already Empty””
      return INT_MIN;
   }
   int item = data[front];
   front = (front + 1) % capacity;
   queueSize = queueSize - 1;
   return item;
}
```

Linked List Implementation of a Queue

- **front** here is a pointer that always points to the head of the linked list.
- **rear** here is a pointer that always points to the last node of the linked list.
- Enqueue() in the Linked list implementation of a queue:
  Every new element is inserted at the end of the linked list. So, every new element is pointed by the **rear** pointer.
- Dequeue() in the Linked list implementation of a stack:
  Element is removed from the beginning of the linked list. To remove an element, simply remove the node pointed by the **front** pointer, and make **front** point to the next node in the list.

Linked List Implementation of a Queue

- Enqueue()

1. Equivalent to adding a new node at the end of a linked list.

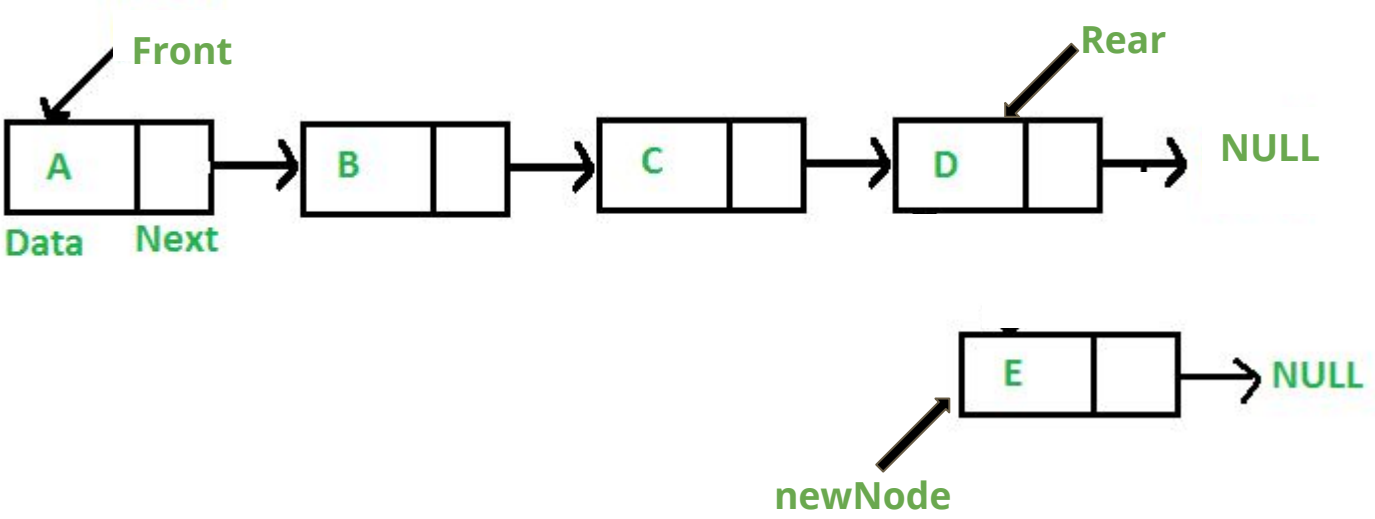
2. However, you now have a pointer to the last node of your linked list **(rear)** This makes the “adding a new node problem at the end” a much easier problem. You don’t need to traverse the linked list from the start node to the end node.

Enqueue() in linked list implementation of a queue

```
void enqueue(int x) {
        //Create a new LL node
        Node* newNode = new Node;
        newNode->key = newKey;
        newNode->next = NULL;
        // If queue is empty, then new node is both “front” and “rear”
        if (rear == NULL && front==NULL ) {
                front = rear = newNode;
                return;
        }
        // Add the new node at the end of queue and change rear
        rear->next = newNode;
        rear = newNode;
}
```

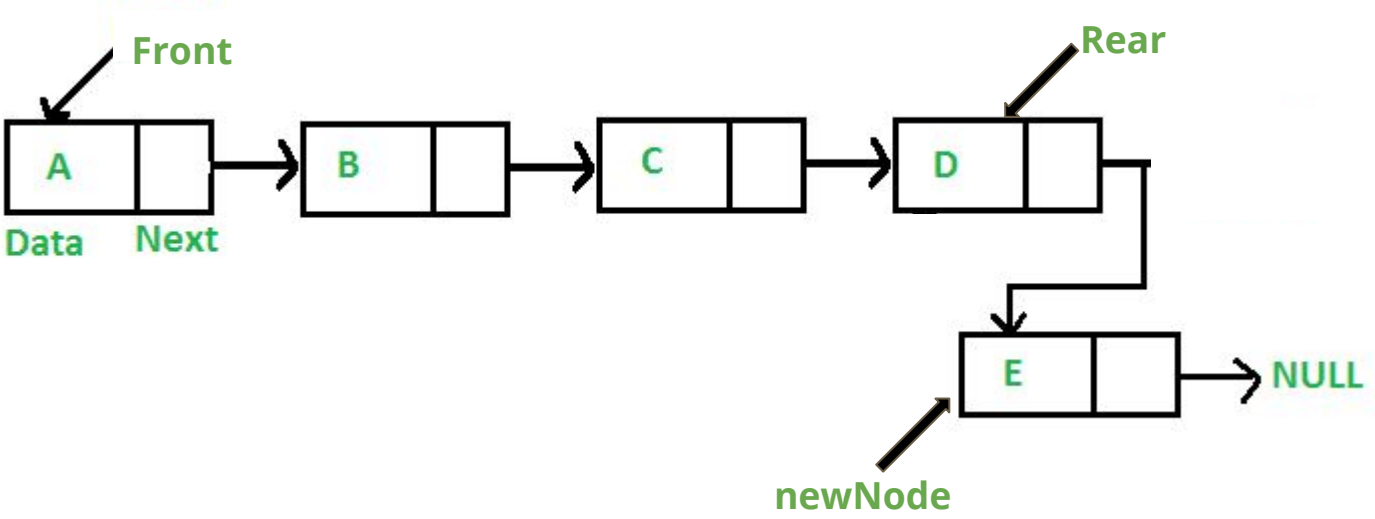
Enqueue() in linked list implementation of a queue

STEP 1



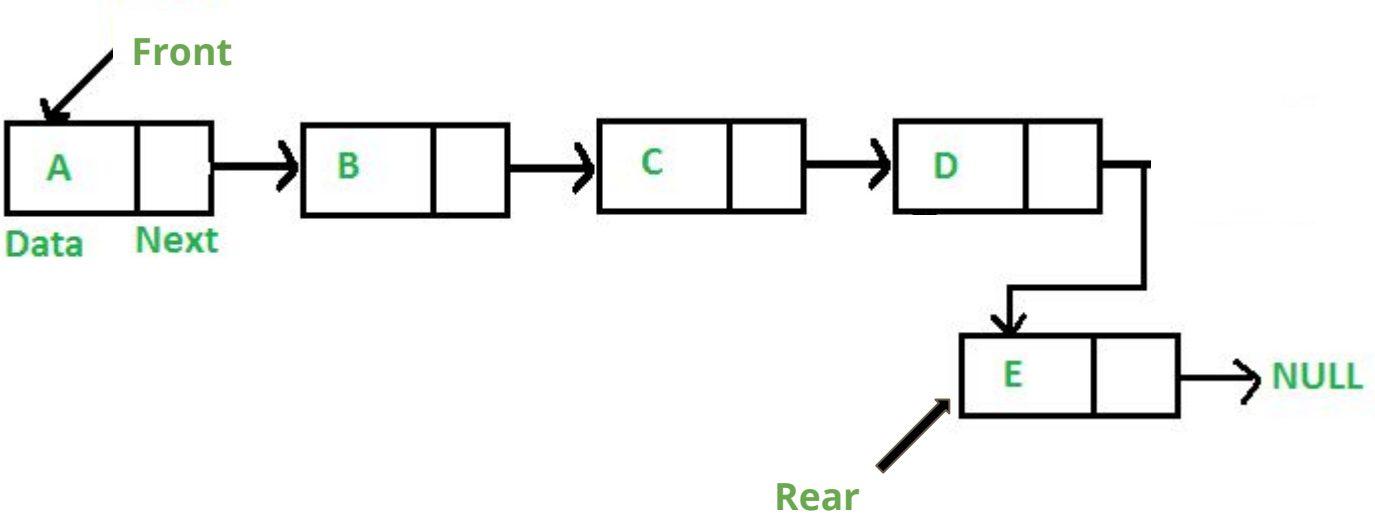
Enqueue() in linked list implementation of a queue

STEP 2



Enqueue() in linked list implementation of a queue

STEP 3



Linked List Implementation of a Queue

- Dequeue()

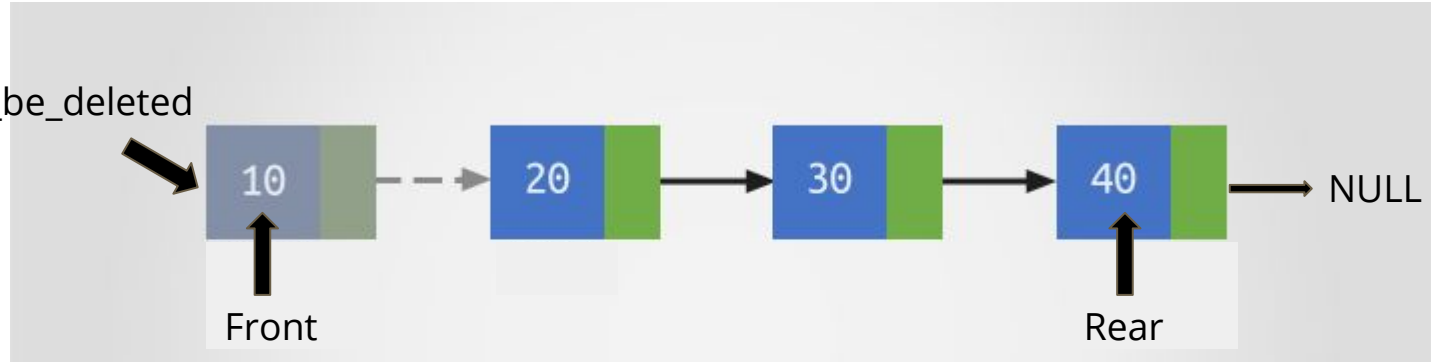
  Equivalent to deleting at the beginning of a linked list.

Dequeue() in linked list implementation of a queue

```
void deQueue() {
        // If queue is empty, return.
        if (front == NULL && rear==NULL)
                return;
        // Store previous front and move front one node ahead
        Node* to_be_deleted = front;
        front = front->next;
        // If front becomes NULL, then change rear also to NULL
        if (front == NULL)
                rear = NULL;
        delete (temp);
        return;
}
```

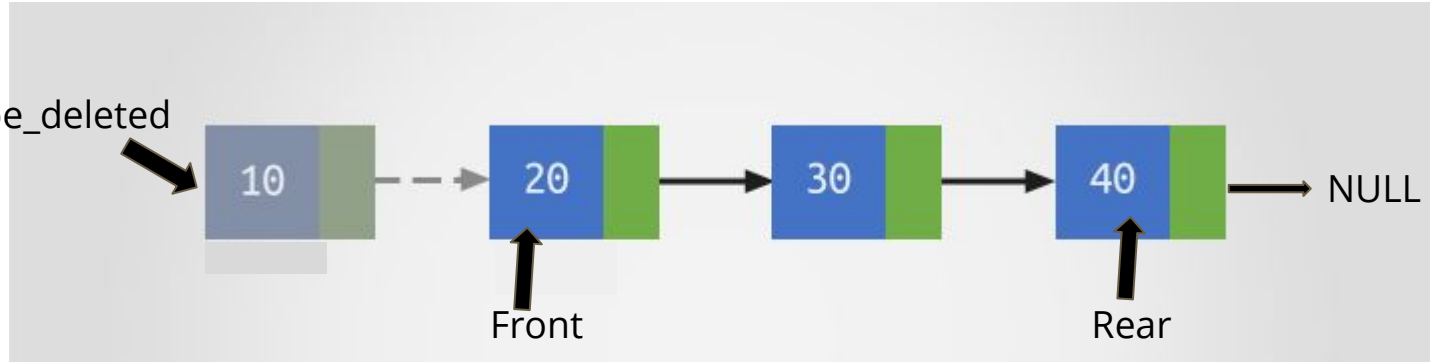
Dequeue() in linked list implementation of a queue

STEP 1



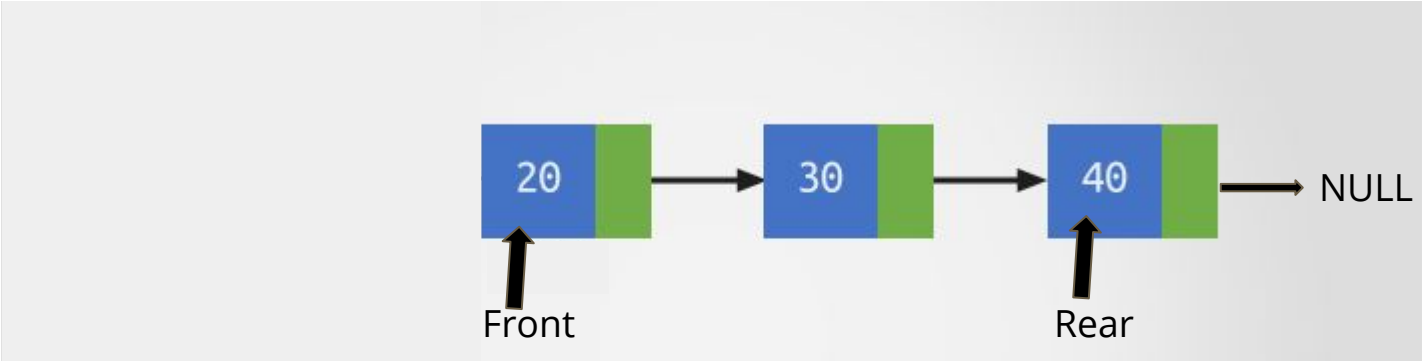
Dequeue() in linked list implementation of a queue

STEP 2



Dequeue() in linked list implementation of a queue

STEP 3



Any questions?

Few common errors

- Always check if your Queue is currently empty or not.
  - How to do that?
  - Check if rear == NULL && front==NULL
    - If True, that means it is empty

- **REMEMBER: “front” pointer always points to the first node in the linked list ALL THE TIME and we delete from the “front” end. “rear” pointer always points to the last node in the linked list ALL THE TIME and we add from the “rear” end.**

Exercise (Silver Problem)

- This problem is mandatory.
- Definition of the QueueLL class is present in QueueLL.hpp
- You will be implementing the QueueLL::enqueue(int key) and QueueLL::dequeue() function in QueueLL.cpp
- Compile both DriverQueue.cpp and QueueLL.cpp together for successful compilation.
  - **g++ -std=c++11 DriverQueue.cpp QueueLL.cpp -o rec5**
  - **./rec5**

Expected Output

```
himanshu@Mercury:~/Downloads/Lab 5/Lab5-20200209T224357Z-001/Lab5$ g++ QueueLL.cpp DriverQueue.cpp
himanshu@Mercury:~/Downloads/Lab 5/Lab5-20200209T224357Z-001/Lab5$ ./a.out
(1) Queue empty? yes.
(2) Enqueuing 1, 2, 3
(3) Queue empty? no.
(4) Peeked key: 1
(5) Dequeuing everything
  - dequeue: 1
  - dequeue: 2
  - dequeue: 3
```

Exercise (Gold Problem)

- This problem is not mandatory but you are highly encouraged to solve it.
- You will be implementing the isValid() function in Driver.cpp
- Once again, compile both Driver.cpp and StackLL.cpp together for successful compilation.
  - **g++ -std=c++11 Driver.cpp StackLL.cpp -o rec5**
  - **./rec5**

Exercise (Gold Problem)

Input: *exp = "[()]{}{[()()]()}"*

Output: *Balanced*

Input: *exp = "[(])"*

Output: *Not Balanced*

Exercise (Gold Problem)

- **How can you solve this?**
  - You can store the opening brackets in a stack.
  - Everytime you encounter a closed bracket, peek/pop an element from your stack. If the returned element is the corresponding opening bracket, then keep repeating. If the returned element is not the corresponding opening bracket, then that means it is not balanced.