LECTURE Four and Five

Dynamic Data Structure

- A dynamic data structure (DDS) refers to **an organization or collection of data in memory that has the flexibility to grow or shrink in size**, enabling a programmer to control exactly how much memory is utilized.
- Dynamic data structures change in size by having unused memory allocated or de-allocated from the heap as needed.
- Their role is to provide the programmer with the flexibility to adjust the memory consumption of software programs.
- Some of the common examples of the dynamic data structure include linked lists, stack and queues.

DYNAMIC DATA STRUCTURES VS. STATIC DATA STRUCTURES

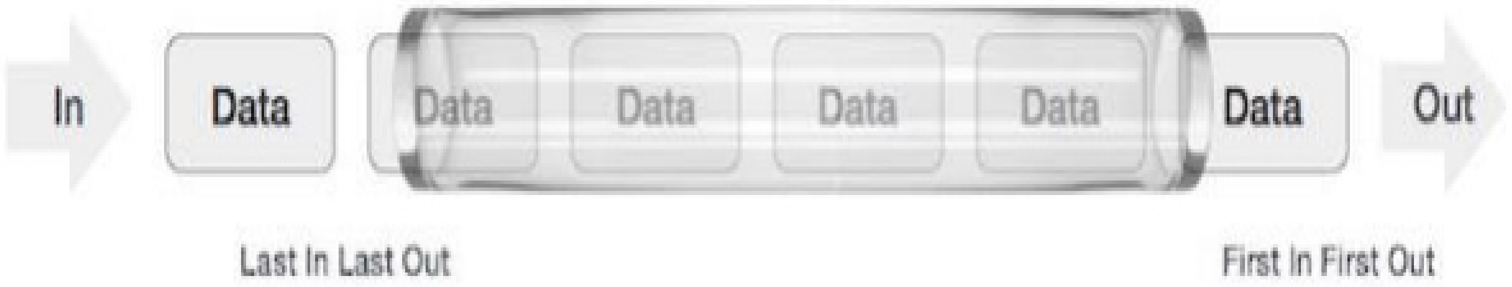
- Dynamic data structures stand in contrast to **static data structures** (SDS), wherein in the case of the latter the size of the structure is fixed.
- Static data structures are ideal for storing a fixed number of data items, but they lack the dynamic data structure s flexibility to consume additional memory if needed or free up memory when possible for improved efficiency.
- As a result, when the number of data items can t be predicted beforehand, a dynamic data structure should be used.
- A potential drawback to using dynamic data structures, though, is that because allocation of memory isn t fixed, there is the possibility for the structure to **overflow** if it exceeds the maximum allowed memory limit or **underflow** if the data structure becomes empty.

QUEUES

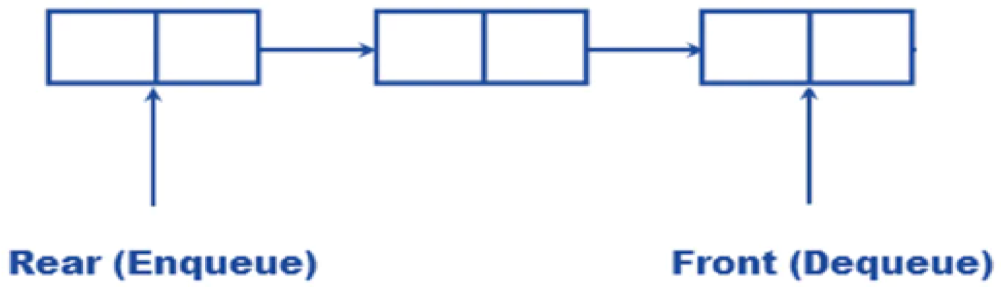
- It is a sequential data structure that follows the FIFO methodology.
- First in Frist out (FIFO)
- A good example of a queue is any line of consumers for a resource where the consumer that came first is served first. Therefore it can be used to develop online delivery applications.
- It can only be modified by the addition of data entities at one end (enqueue) or the removal of data entities at another(dequeue).
- By convention, the end where insertion is performed is called **Rear**, and the end at which deletion takes place is known as the **Front**.

Queue Representation

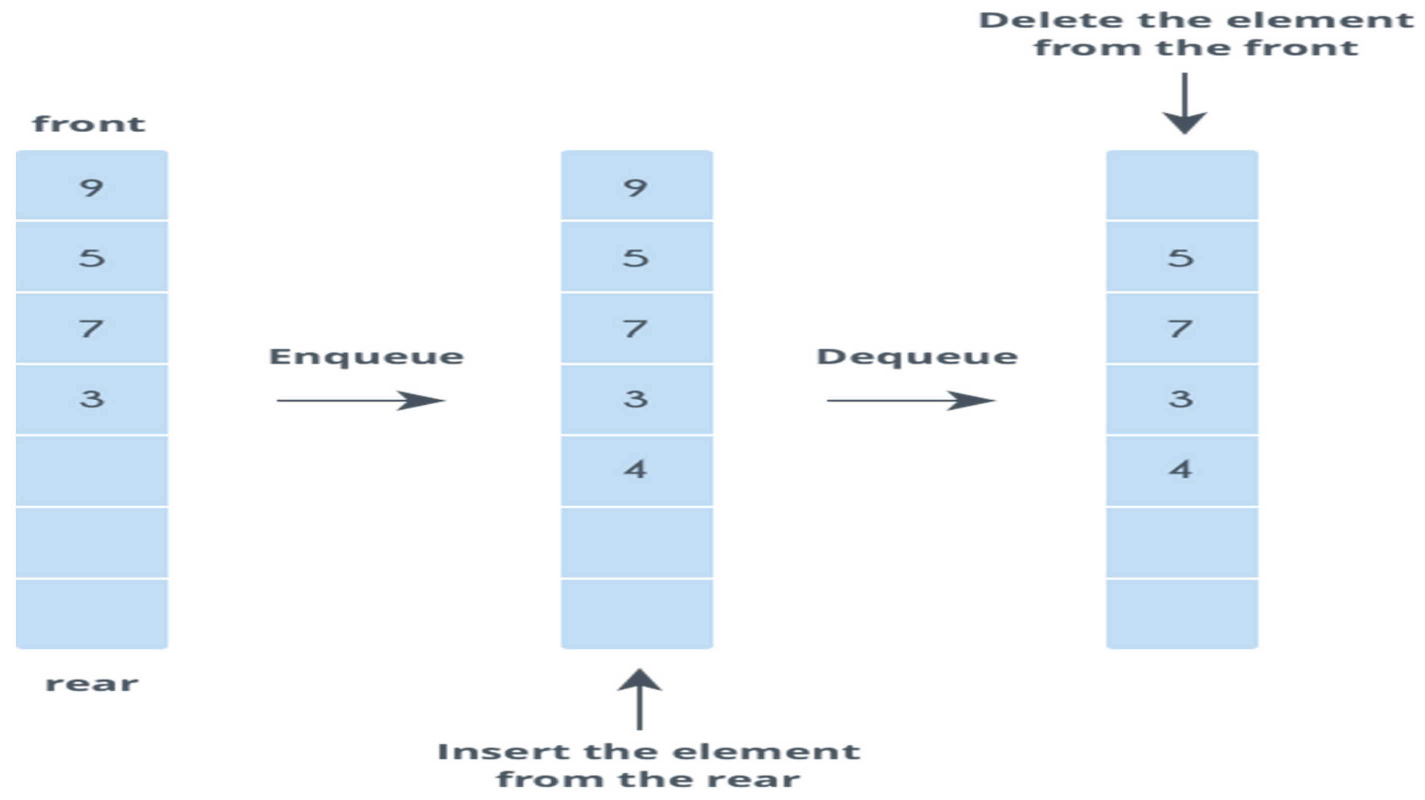
The Principle



The Operation



How a queue works


Delete the element from the front
front
9
5
7
3
rear
Enqueue
9
5
7
3
4
Insert the element from the rear
Dequeue
5
7
3
4

Basic Operations on a Queue

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

peek()

- This function function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −
- Algorithm
- Begin procedure of peek
- Return queue (front)
- End procedure

Implementation of peek() function in C programming language –

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

Isfull() Operation

- As queue is implemented using single dimension, ensure that the rear pointer to reach at MAXSIZE to determine that the queue is full. In case queue is maintained in circular linked-list, the algorithm will differ.
- Algorithm

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

Implementation of isfull() function in C programming language –

- bool isfull() {
- if(rear == MAXSIZE - 1)
- return true;
- else
- return false;
- }

Algorithm of isempty() function –

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

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

Implementation in C

```
bool isempty() {
if(front < 0 || front > rear)
return true;
else
return false;
}
```

Types of Queues: Simple Queue

- Simple queue also known as a linear queue is the most basic version of a queue.
- Here, insertion of an element i.e. the Enqueue operation takes place at the rear end and removal of an element i.e. the Dequeue operation takes place at the front end.

Types of Queues: Priority Queue

- This queue is a special type of queue. Its specialty is that it arranges the elements in a queue based on some priority.
- The priority can be something where the element with the highest value has the priority so it creates a queue with decreasing order of values.
- The priority can also be such that the element with the lowest value gets the highest priority so in turn it creates a queue with increasing order of values.

Types of Queues: Dequeue

- Dequeue is also known as Double Ended Queue.
- As the name suggests double ended, it means that an element can be inserted or removed from both the ends of the queue unlike the other queues in which it can be done only from one end.
- Because of this property it may not obey the First In First Out property

Types of Queues: Circular Queue

- In a circular queue, the element of the queue act as a circular ring. The working of a circular queue is similar to the linear queue except for the fact that the last element is connected to the first element. Its advantage is that the memory is utilized in a better way. This is because if there is an empty space i.e. if no element is present at a certain position in the queue, then an element can be easily added at that position.

Enqueue Operation into a Data Structure Queue

- Two pointers – **front** and **rear** are maintained by Queues and hence the operations are different from that of Stacks.
- The steps to enqueue (insert) data into a queue are -
- **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.

Steps for DEQUEUE

- Check queue is empty or not
- if empty, print underflow and exit
- if not empty, print element at the head and increment head

```c
// C program for array implementation of queue
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
  // A structure to represent a queue
struct Queue {
    int front, rear, size;
    unsigned capacity;
    int* array;      };
// function to create a queue // of given capacity. t initializes size of queue as 0
struct Queue* createQueue(unsigned capacity)
{
    struct Queue* queue = (struct Queue*)malloc(
        sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
      // This is important, see the enqueue
    queue->rear = capacity - 1;
    queue->array = (int*)malloc(
        queue->capacity * sizeof(int));
    return queue;
}
```

```
// Queue is full when size becomes
// equal to the capacity
int isFull(struct Queue* queue)
{
    return (queue->size == queue->capacity); }

// Queue is empty when size is 0
int isEmpty(struct Queue* queue)
{
    return (queue->size == 0);
}
```

```c
// Function to add an item to the queue.
// It changes rear and size
void enqueue(struct Queue* queue, int item)
{
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1)
                  % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    printf("%d enqueued to queue\n", item);
}

// Function to remove an item from queue.
// It changes front and size
int dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1)
                   % queue->capacity;
    queue->size = queue->size - 1;
    return item;
}
//all these 4 slides belong to one program.
```

```c
// Function to get front of queue
int front(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->front];
}

// Function to get rear of queue
int rear(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->rear];
}

// Driver program to test above functions./
int main()
{
    struct Queue* queue = createQueue(1000);

    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    enqueue(queue, 40);

    printf("%d dequeued from queue\n\n",
           dequeue(queue));

    printf("Front item is %d\n", front(queue));
    printf("Rear item is %d\n", rear(queue));

    return 0;
}
```

Applications of Queue

- Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

Applications of Queue

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Queue applications

- Queues are one of the most common of all data processing structures.
- They are found in virtually every operating system and network and in countless other areas.
- For example, queues are used in online business applications such as processing customer requests, jobs and orders.
- In a computer system, a queue is needed to process jobs and for system services such as print spools.

Bus Stop Queue

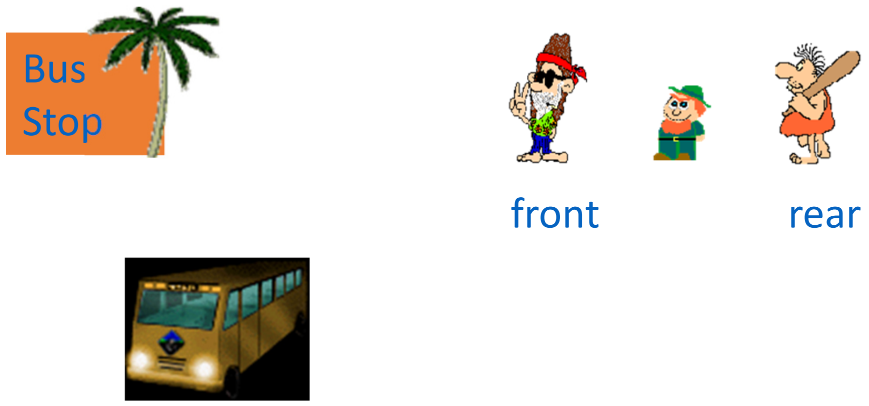
Bus
Stop
front
rear

- Remove a person from the queue

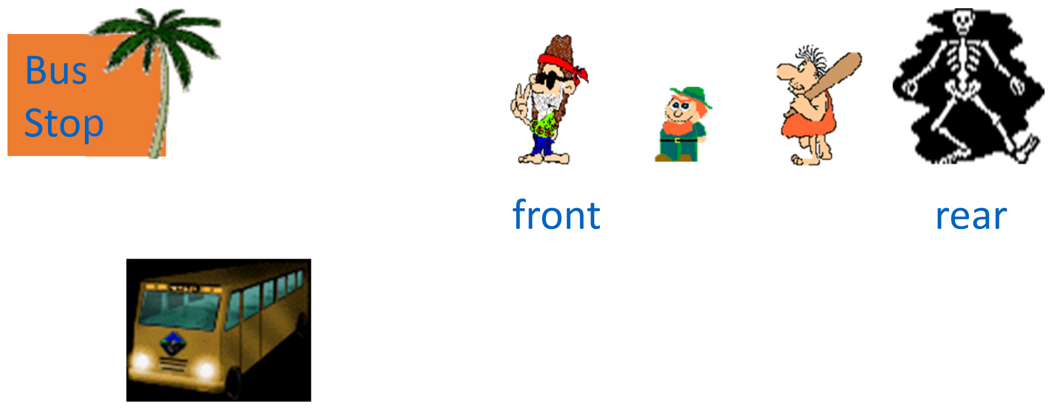
Bus Stop Queue

Bus
Stop
front
rear

Bus Stop Queue

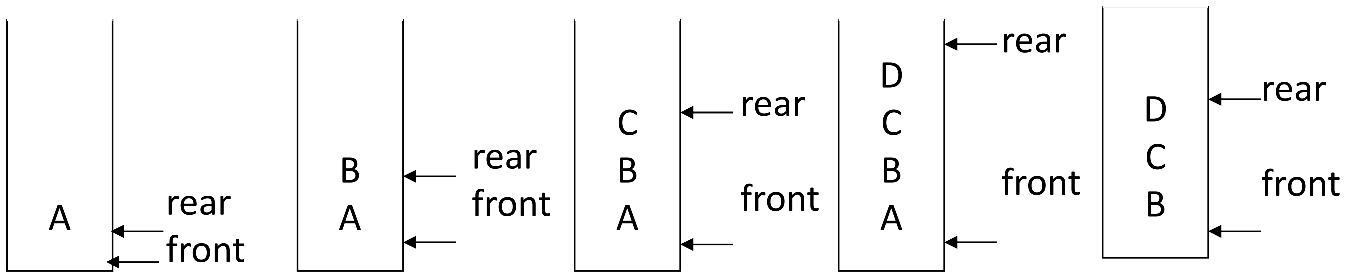


Bus Stop Queue



- Add a person to the queue
- A queue is a FIFO (First-In, First-Out) list.

First In First Out



Queue Example

Operation		***Output***	***Q***
enqueue(5)	–	(5)	
enqueue(3)	–	(5, 3)	
dequeue()	*5*	(3)	
enqueue(7)	–	(3, 7)	
dequeue()	*3*	(7)	
front()	*7*	(7)	
dequeue()	*7*	()	
dequeue()	*“error”*	()	
isEmpty()		*true*	()
enqueue(9)	–	(9)	
enqueue(7)	–	(9, 7)	
size()	*2*	(9, 7)	
enqueue(3)	–	(9, 7, 3)	
enqueue(5)	–	(9, 7, 3, 5)	
dequeue()	*9*	(7, 3, 5)	

Implementing a queue using List

- The list is a built-in data type in Python. We are going to use the list data type to implement a queue in a class.
- We will walk you through different steps to implement the queue data structure from scratch without any modules.

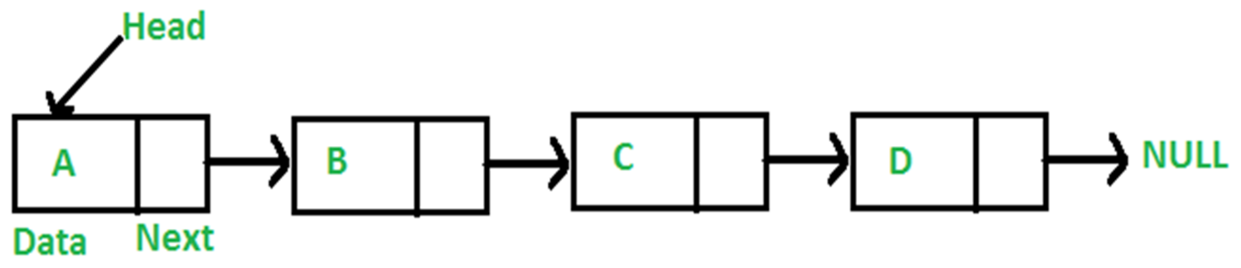
implementing Queue using List in python:

- q=[]
- q.append(10)
- q.append(100)
- q.append(1000)
- q.append(10000)
- print("Initial Queue is:",q)
- print(q.pop(0))
- print(q.pop(0))
- print(q.pop(0))
- print("After Removing elements:",q)

LINKED LISTS

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.

A LINKED LIST



- In simple words, a linked list consists of nodes where each node contains a data field and a reference(link/Pointer) to the next node in the list.
- Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.
- They includes a series of connected nodes.

Each node stores the data and the address of the next

Why Linked List?

- Arrays can be used to store linear data of similar types, but arrays have the following limitations.
- **The size of the arrays is fixed**: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
- **Insertion of a new element / Deletion of a existing element in an array of elements is expensive:** The room has to be created for the new elements and to create room existing elements have to be shifted but in Linked list if we have the head node then we can traverse to any node through it and insert new node at any position.

Examples on insertion and deletion of an element from array

- In a system, if we maintain a sorted list of IDs in an array id[].
  id[] = [1000, 1010, 1050, 2000, 2040].
  And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
- Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved due to this so much work is being done which affects the efficiency of the code.
- Therefore a linked has two advantages over array
  - Dynamic Array.
  - Ease of Insertion/Deletion.

Drawbacks:

- Random access is not allowed. We have to access elements sequentially starting from the first node(head node). So we cannot do binary search with linked lists efficiently with its default implementation.
- Extra memory space for a pointer is required with each element of the list.
- Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation

- A linked list is represented by a pointer to the first node of the linked list. The first node is called the head.
- If the linked list is empty, then the value of the head points to NULL.
- Each node in a list consists of at least two parts:
  - A Data Item (we can store integer, strings or any type of data).
  - Pointer (Or Reference) to the next node (connects one node to another) or An address of another node
- In C, we can represent a node using structures. In Java or C#, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

A linked list in C

```
/ A linked list node
struct Node {
    int data;
    struct Node* next;
};
```

Linked List in Python

```
# Function to initialize the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize next as null
# Linked List class
class LinkedList:

    # Function to initialize the Linked
    # List object
    def __init__(self):
        self.head = None
```