Unit II
Linear Data Structure

Unit II: Linear Data Structure **(07 Hrs.)**
Array, Stack, Queue, Linked-list and its types, Various Representations, Operations & Applications of Linear Data Structures

2.1 Arrays
Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation
Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

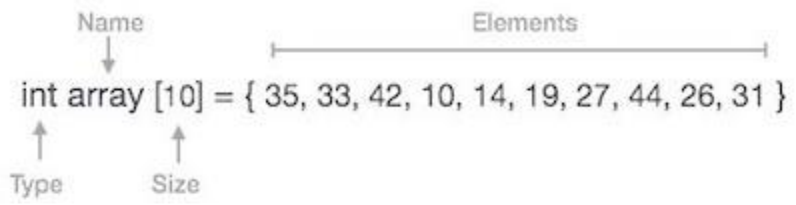


Figure 2.1

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

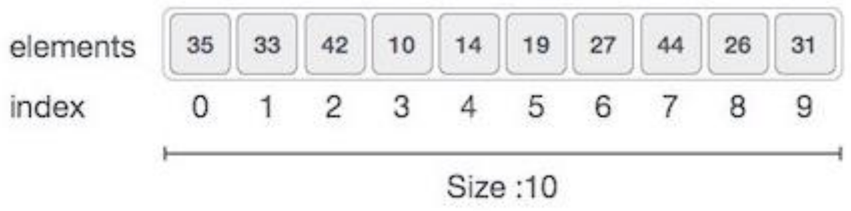


Figure 2.2

As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 27

Basic Operations
Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

Data Type	Default Value
bool	false
char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

Figure 2.3

Time complexity of Array

Operation	Average Case	Worst Case
Read	O(1)	O(1)
Insert	O(n)	O(n)
Delete	O(n)	O(n)
Search	O(n)	O(n)

Figure 2.4

Advantages and disadvantages of Arrays

Advantages

1. Reading an array element is simple and efficient. As shown in the above table, the read time of array is O (1) in both best and worst cases. This is because any element can be instantly read using indexes (base address calculation behind the scene) without traversing the whole array.
2. Array is a foundation of other data structures. For example other data structures such as LinkedList, Stack, Queue etc. are implemented using array.
3. All the elements of an array can be accessed using a single name (array name) along with the index, which is readable, user-friendly and efficient rather than storing those elements in different-2 variables.

Disadvantages

1. While using array, we must need to make the decision of the size of the array in the beginning, so if we are not aware how many elements we are going to store in array, it would make the task difficult.
2. The size of the array is fixed so if at later point, if we need to store more elements in it then it can't be done. On the other hand, if we store less number of elements than the declared size, the remaining allocated memory is wasted.

<u>**Address Calculation in single (one) Dimension Array:**</u>

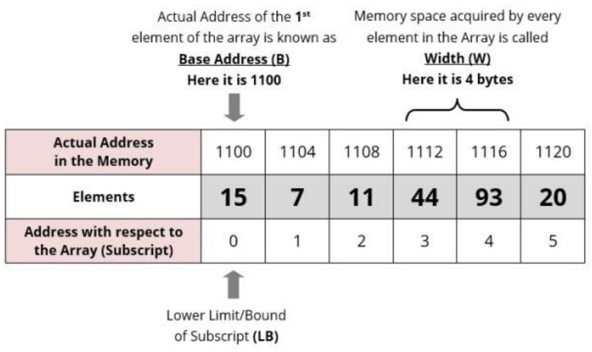

Actual Address of the 1st element of the array is known as Base Address (B) Here it is 1100
Memory space acquired by every element in the Array is called Width (W) Here it is 4 bytes
Actual Address in the Memory
1100 1104 1108 1112 1116 1120
Elements
15 7 11 44 93 20
Address with respect to the Array (Subscript)
0 1 2 3 4 5
Lower Limit/Bound of Subscript (LB)

Figure 2.5

Array of an element of an array say “A[I]” is calculated using the following formula:
Address of A [I] = B + W * (I – LB)
Where,
B = Base address
W = Storage Size of one element stored in the array (in byte)
I = Subscript of element whose address is to be found
LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)
Example:
Given the base address of an array **B[1300…..1900]** as 1020 and size of each element is 2 bytes in the memory. Find the address of **B[1700]**.
Solution:
The given values are: B = 1020, LB = 1300, W = 2, I = 1700
Address of A [I] = B + W * (I – LB)
= 1020 + 2 * (1700 – 1300)
= 1020 + 2 * 400
= 1020 + 800
= 1820 **[Ans]**

2D Array
2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.
However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.
How to declare 2D Array
The syntax of declaring two dimensional array is very much similar to that of a one dimensional array, given as follows.

```
int arr[max_rows][max_columns];
```

However, it produces the data structure which looks like following.

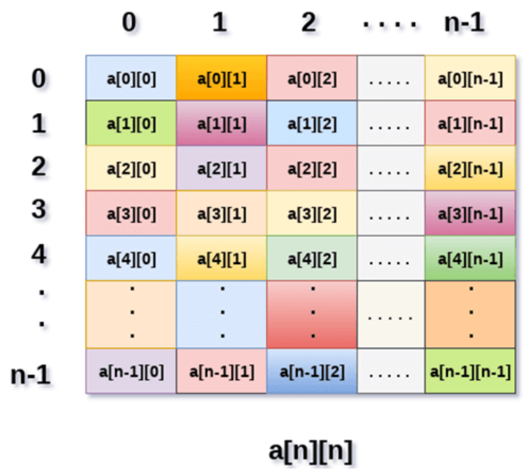


Figure 2.6

Above image shows the two dimensional array, the elements are organized in the form of rows and columns. First element of the first row is represented by a[0][0] where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

How do we access data in a 2D array

Due to the fact that the elements of 2D arrays can be random accessed. Similar to one dimensional arrays, we can access the individual cells in a 2D array by using the indices of the cells. There are two indices attached to a particular cell, one is its row number while the other is its column number.
However, we can store the value stored in any particular cell of a 2D array to some variable x by using the following syntax.

int x = a[i][j]

where i and j is the row and column number of the cell respectively.
We can assign each cell of a 2D array to 0 by using the following code:

```
for ( int i=0; i<n ;i++)
{
    for (int j=0; j<n; j++)
    {
        a[i][j] = 0;
    }
}
```

Figure 2.7

Initializing 2D Arrays

We know that, when we declare and initialize one dimensional array in C programming simultaneously, we don't need to specify the size of the array. However this will not work with 2D arrays. We will have to define at least the second dimension of the array. The syntax to declare and initialize the 2D array is given as follows.

int arr[2][2] = {0,1,2,3};

The number of elements that can be present in a 2D array will always be equal to (**number of rows * number of columns**).

Address Calculation in Double (Two) Dimensional Array:

While storing the elements of a 2-D array in memory, these are allocated contiguous memory locations. Therefore, a 2-D array must be linearized so as to enable their storage. There are two alternatives to achieve linearization: Row-Major and Column-Major.

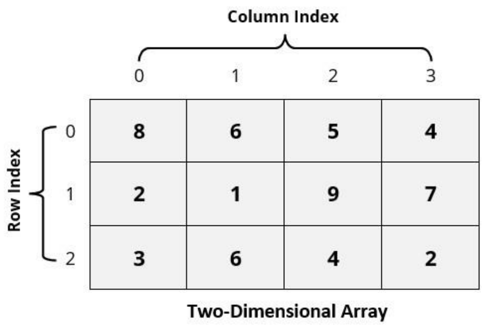


Figure 2.8

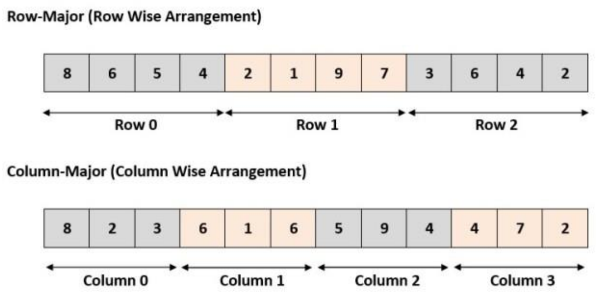


Figure 2.9

Address of an element of any array say “**A[I][J]**” is calculated in two forms as given: (1) Row Major System (2) Column Major System

<u>Row Major System:</u>

The address of a location in Row Major System is calculated using the following formula:

Address of A [I][J] = B + W * [N * (I – Lr) + (J – Lc)]

<u>Column Major System:</u>

The address of a location in Column Major System is calculated using the following formula:

Address of A [I][J] Column Major Wise = B + W * [(I – Lr) + M * (J – Lc)]

Where,

B = Base address
I = Row subscript of element whose address is to be found
J = Column subscript of element whose address is to be found
W = Storage Size of one element stored in the array (in byte)
Lr = Lower limit of row/start row index of matrix, if not given assume 0 (zero)
Lc = Lower limit of column/start column index of matrix, if not given assume 0 (zero)
M = Number of row of the given matrix
N = Number of column of the given matrix

Examples:

Q 1. An array X [-15……….10, 15……………40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20].

Solution:

As you see here the number of rows and columns are not given in the question. So they are calculated as:

Number or rows say **M = (Ur – Lr) + 1** = [10 – (- 15)] +1 = 26

Number or columns say **N = (Uc – Lc) + 1** = [40 – 15)] +1 = 26

(i) Column Major Wise Calculation of above equation

The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, M = 26

Address of A [I][J] = B + W * [(I – Lr) + M * (J – Lc)]

= 1500 + 1 * [(15 – (-15)) + 26 * (20 – 15)] = 1500 + 1 * [30 + 26 * 5] = 1500 + 1 * [160] = 1660 **[Ans]**

(ii) Row Major Wise Calculation of above equation

The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, N = 26

Address of A [I][J] = B + W * [N * (I – Lr) + (J – Lc)]

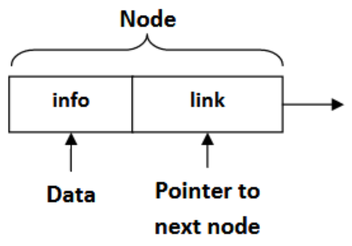
= 1500 + 1* [26 * (15 – (-15))) + (20 – 15)] = 1500 + 1 * [26 * 30 + 5] = 1500 + 1 * [780 + 5] = 1500 + 785

= 2285 **[Ans]**

2.2 Linked-list

Linked list is a linear data structure. It is a collection of data elements, called nodes pointing to the next node by means of a pointer.

In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.



```
// C Structure to represent a node
struct node
{
      int info
      struct node *link
};
```

Figure 2.10

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

How to add elements to linked list

You can add elements to either beginning, middle or end of linked list.

Add to beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = item;
newNode->next = head;
head = newNode;
```

Add to end

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = item;
newNode->next = NULL;
struct node *temp = head;
while(temp->next != NULL){
  temp = temp->next;
}
temp->next = newNode;
```

Add to middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = item;
struct node *temp = head;
for(int i=2; i < position; i++) {
    if(temp->next != NULL) {
        temp = temp->next;
    }
}
newNode->next = temp->next;
temp->next = newNode;
```

How to delete from a linked list

You can delete either from beginning, end or from a particular position.

Delete from beginning

- Point head to the second node

```
head = head->next;
```

Delete from end

- Traverse to second last element
- Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
  temp = temp->next;
}
temp->next = NULL;
```

Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
    if(temp->next!=NULL) {
        temp = temp->next;
    }
}
temp->next = temp->next->next;
```

Types of Linked List

Following are the various types of linked list.

- **Singly Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Singly Linked List

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

Linked list contains a link element called **first** and each link carries a **data item**. Entry point into the linked list is called the **head of the list.**

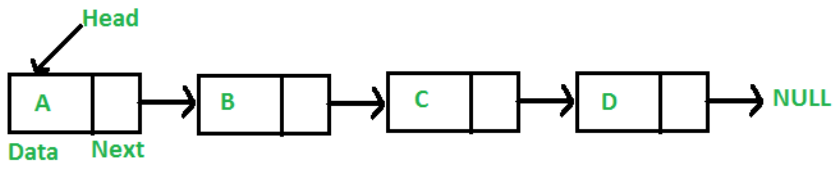


Figure 2.11

Complexity

Data Structure	**Time Complexity**								**Space Compleity**
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	θ(n)	O(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	**Operation**	**Description**
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	**Operation**	**Description**
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Doubly Linked List

We add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.

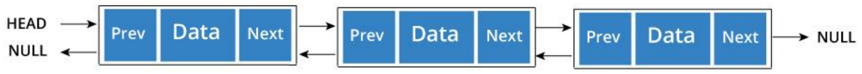


Figure 2.12

A node is represented as

```
struct node {
        int data;
        struct node *next;
        struct node *prev;
    }
```

Operations on doubly linked list

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Circular Linked List

A circular linked list is a variation of linked list in which the last element is linked to the first element. This forms a circular loop. It does not contain null pointers like singly linked list.We can traverse only in one direction that is forward direction.It has the biggest advantage of time saving when we want to go from last node to first node, it directly points to first node.

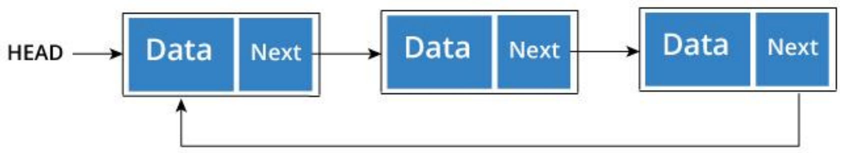


Figure 2.13

A circular linked list can be either singly linked or doubly linked.

- for singly linked list, next pointer of last item points to the first item
- In doubly linked list, prev pointer of first item points to last item as well.

A good example of an application where circular linked list should be used is a timesharing problem solved by the operating system.

Operations on Circular singly linked list:

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

DIFFERENTIATE BETWEEN ARRAY AND LINK LIST

ARRAY	LINKED LIST
Array is a collection of elements of similar data type.	Linked List is an ordered collection of elements of same type, which are connected to each other using pointers.
Array supports **Random Access**, which means elements can be accessed directly using their index, like arr[0] for 1st element, arr[6] for 7th element etc. Hence, accessing elements in an array is **fast** with a constant time complexity of `O(1)`.	Linked List supports **Sequential Access**, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element. To access **nth** element of a linked list, time complexity is `O(n)`.
In an array, elements are stored in **contiguous memory location** or consecutive manner in the memory.	In a linked list, new elements can be stored anywhere in the memory. Address of the memory location allocated to the new element is stored in the previous node of linked list, hence formaing a link between the two nodes/elements.
In array, **Insertion and Deletion** operation takes more time, as the memory locations are consecutive and fixed.	In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list. Insertion and Deletion operations are **fast** in linked list.
Memory is allocated as soon as the array is declared, at **compile time**. It's also known as **Static Memory Allocation**.	Memory is allocated at **runtime**, as and when a new node is added. It's also known as **Dynamic Memory Allocation**.

In array, each element is independent and can be accessed using it's index value.	In case of a linked list, each node/element points to the next, previous, or maybe both nodes.
Array can **single dimensional**, **two dimensional** or **multidimensional**	Linked list can be **Linear(Singly)**, **Doubly** or **Circular** linked list.
Size of the array must be specified at time of array decalaration.	Size of a Linked list is variable. It grows at runtime, as more nodes are added to it.
Array gets memory allocated in the **Stack** section.	Whereas, linked list gets memory allocated in **Heap** section.

2.3 Stack

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.
A stack is a **recursive** data structure.

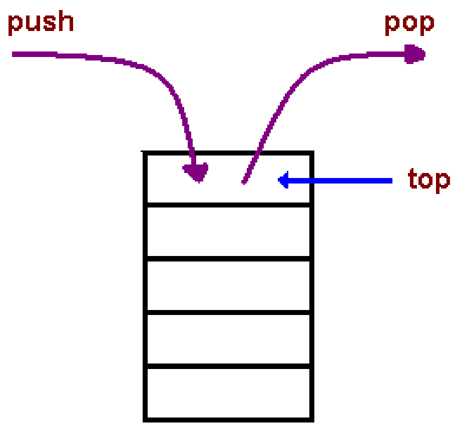
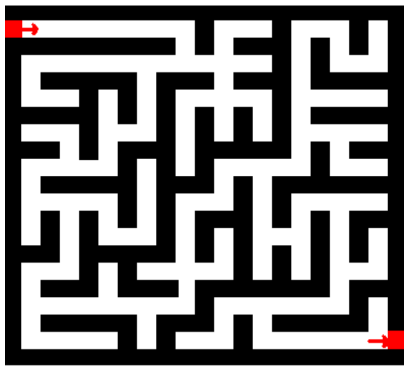


Figure 2.14

Applications

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.



Backtracking. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit? Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

Figure 2.15

Implementation of Stack Data Structure

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.

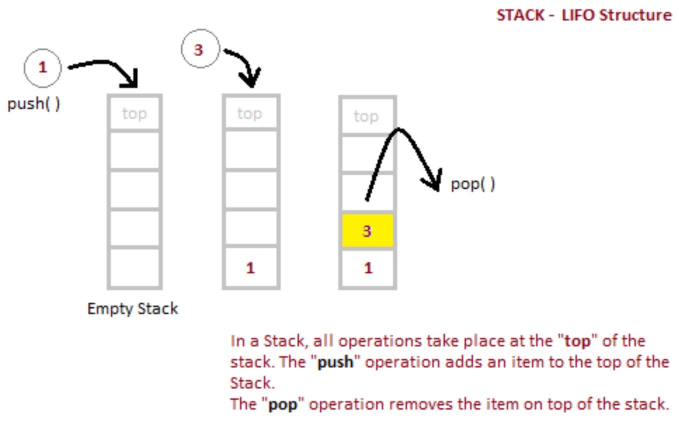

STACK - LIFO Structure
1
push( )
3
top
top
top
pop( )
3
1
1
Empty Stack
In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.
The "pop" operation removes the item on top of the stack.

Figure 2.16

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

Algorithm for PUSH operation

1. Check if the stack is **full** or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element.

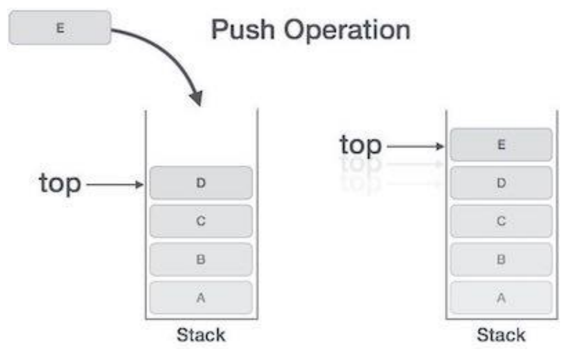


Figure 2.17

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Algorithm for POP operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.

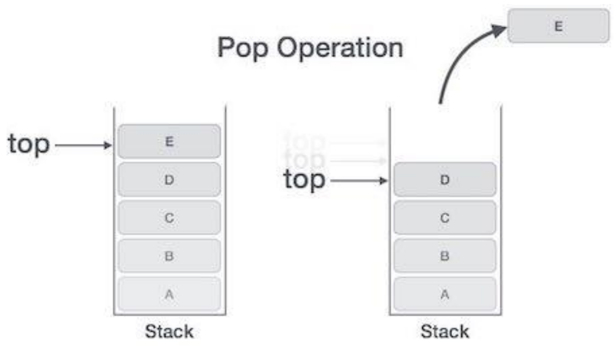


Figure 2.18

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Stack ADT

Structure Stack is
Objects: a finite ordered list with zero or more elements.
Functions:

for all stack ϵ Stack, item ϵ elements, max_stack_size ϵ positive integer

```
Stack CreateS (max_stack_size) ::=
                        Create an empty stack whose maximum size is max_stack_size
Boolean IsFull( stack, max_stack_size) ::=
                        if ( number of element in stack == max_stack_size)
                        return TRUE
                        else return FALSE
Stack Add ( stack, item) ::=
                        if (IsFull (stack)) stack_full
                        else insert item into top of stack and return
```

```
Boolean IsEmpty ( stack) ::=
                    if (stack== CreateS (max_stack_size))
                    return TRUE
                    else return FALSE
Element Delete(stack) ::=
                    if (IsEmpty (stack)) return
                    else remove and return the item on the top of the stack.
```

2.4 Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
2. Queue is referred to be as First In First Out list.
3. For example, people waiting in line for a rail ticket form a queue.

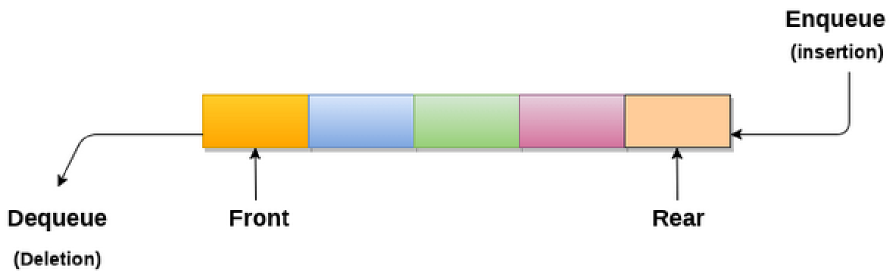


Figure 2.19

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.
If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.
Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

```
• Step 1: IF REAR = MAX - 1
    Write OVERFLOW
    Go to step
    [END OF IF]
• Step 2: IF FRONT = -1 and REAR = -1
    SET FRONT = REAR = 0
    ELSE
    SET REAR = REAR + 1
    [END OF IF]
• Step 3: Set QUEUE[REAR] = NUM
• Step 4: EXIT
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.
Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

```
• Step 1: IF FRONT = -1 or FRONT > REAR
      Write UNDERFLOW
      ELSE
      SET VAL = QUEUE[FRONT]
      SET FRONT = FRONT + 1
      [END OF IF]
• Step 2: EXIT
```

Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Complexity

Data Structure	**Time Complexity**								**Space Complexity**
	Average				**Worst**				
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Queue ADT

```
Structure Queue is
Objects: a finite ordered list with zero or more elements.
functions:
for all queue ∈ Queue, item ∈ elements, max_queue_size ∈ positive integer
Queue CreateQ (max_queue_size) ::=
        Create an empty queue whose maximum size is max_queue_size
Boolean IsFullQ( queue, max_queue_size) ::=
        if ( number of element in queue == max_queue_size)
        return TRUE
        else return FALSE
Queue AddQ ( queue, item) ::=
        if (IsFullQ (queue)) queue_full
        else insert item at rear of queue and return queue
Boolean IsEmptyQ ( queue) ::=
        if (queue== CreateQ (max_queue_size))
        return TRUE
        else return FALSE
Element DeleteQ(queue) ::=
        if (IsEmpty (queue)) return
        else remove and return the item at front of queue.
```

Various Representation of Linear Data Structure

There are two ways to represent a linear data structure in memory,

- Static memory allocation
- Dynamic memory allocation

Static memory allocation- In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.

Example of Static Data Structures: Array

Dynamic memory allocation-

In Dynamic data structure the size of the structure in not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.

Example of Dynamic Data Structures: Linked List

Below we have a pictorial representation showing how consecutive memory locations are allocated for array, while in case of linked list random memory locations are assigned to nodes, but each node is connected to its next node using **pointer**.

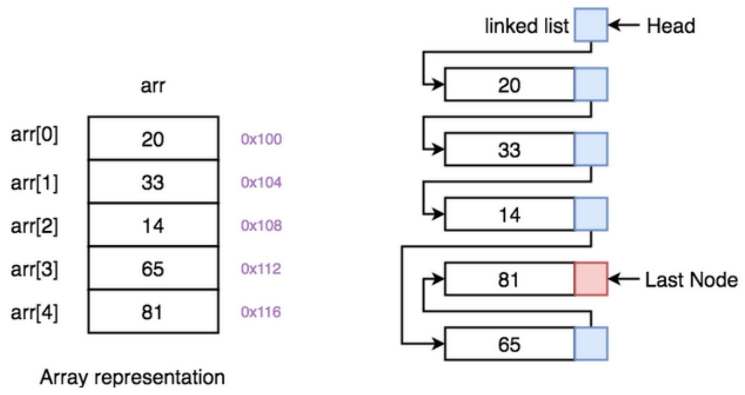


Figure 2.20

On the left, we have **Array** and on the right, we have **Linked List**.

Operations on linear data Structure

The basic operations that are performed on data structures are as follows:

Insertion: Insertion means addition of a new data element in a data structure.

Deletion: Deletion means removal of a data element from a data structure if it is found.

Searching: Searching involves searching for the specified data element in a data structure.

Traversal: Traversal of a data structure means processing all the data elements present in it.

Sorting: Arranging data elements of a data structure in a specified order is called sorting.

Merging: Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

Applications of linear data structure

- Linked list are usually a basic dynamic data structure which implements queues and stacks
- In web-browsers, where it creates a linked list of web-pages visited, so that when you check history (traversal of a list) or press back button, the previous node's data is fetched.
- To reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- An "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack. - Undo/Redo stacks in Excel or Word.
- Language processing : -compiler's syntax check for matching braces is implemented by using stack.

Support for recursion -Activation records of method calls