KEMENTERIAN PENGAJIAN TINGGI
JABATAN PENDIDIKAN POLITEKNIK DAN KOLEJ KOMUNITI

POLITEKNIK MALAYSIA
KUALA TERENGGANU

DATA STRUCTURES

AUTHOR

Mohd Sabri Bin Ahmad
Siti Sarah Malini Bt Mohd Hanifa
Rasmaliza Bt Rashid

Cetakan Pertama 2022
©Politeknik Kuala Terengganu

Tidak dibenarkan diterbitkan semula atau ditukar dalam apa jua bentuk dengan cara apa jua sama ada elektronik, mekanikal, fotokopi, rakaman dan sebagainya sebelum mendapat kebenaran bertulis daripada Pengarah Politeknik Kuala Terengganu.

Diterbitkan oleh:
Politeknik Kuala Terengganu
20200 Jalan Sultan Ismail
Kuala Terengganu, Terengganu.

Politeknik Kuala Terengganu

All right reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage or retrieval system, without prior written permission from the publisher, Politeknik Kuala Terengganu

Author:
Mohd Sabri Bin Ahmad
Siti Sarah Malini Bt Mohd Hanifa
Rasmaliza Bt Rashid

Published by:

Politeknik Kuala Terengganu
20200 Jalan Sultan Ismail
Kuala Terengganu, Terengganu

Perpustakaan Negara Malaysia Cataloguing-in-Publication Data

Mohd. Sabri Ahmad, 1982-
DATA STRUCTURES / AUTHOR Mohd Sabri Bin Ahmad, Siti Sarah Malini Bt Mohd Hanifa, Rasmaliza Bt Rashid.
Mode of access: Internet
eISBN 978-967-2240-39-6

1. Data structures (Computer science).
2. Electronic data processing.
3. Government publications--Malaysia.
4. Electronic books

I. Siti Sarah Malini Mohd. Hanifa, 1982-. II. Rasmaliza Rashid, 1979-. III. Title. 005.73

AUTHOR

MOHD SABRI BIN AHMAD

SITI SARAH MALINI BT MOHD HANIFA

RASMALIZA BT RASHID

OVERVIEW

e-Book Data Structures is written to focus on the basic concept of data structures. This e-book cover the specific topic of data structures such as linked list, stacks, queues, trees, sorting and searching. The emphasis is on choosing appropriate data structures and designing correct and efficient algorithms to operate on these data structures.

CHAPTER 1

INTRODUCTION TO DATA STRUCTURE

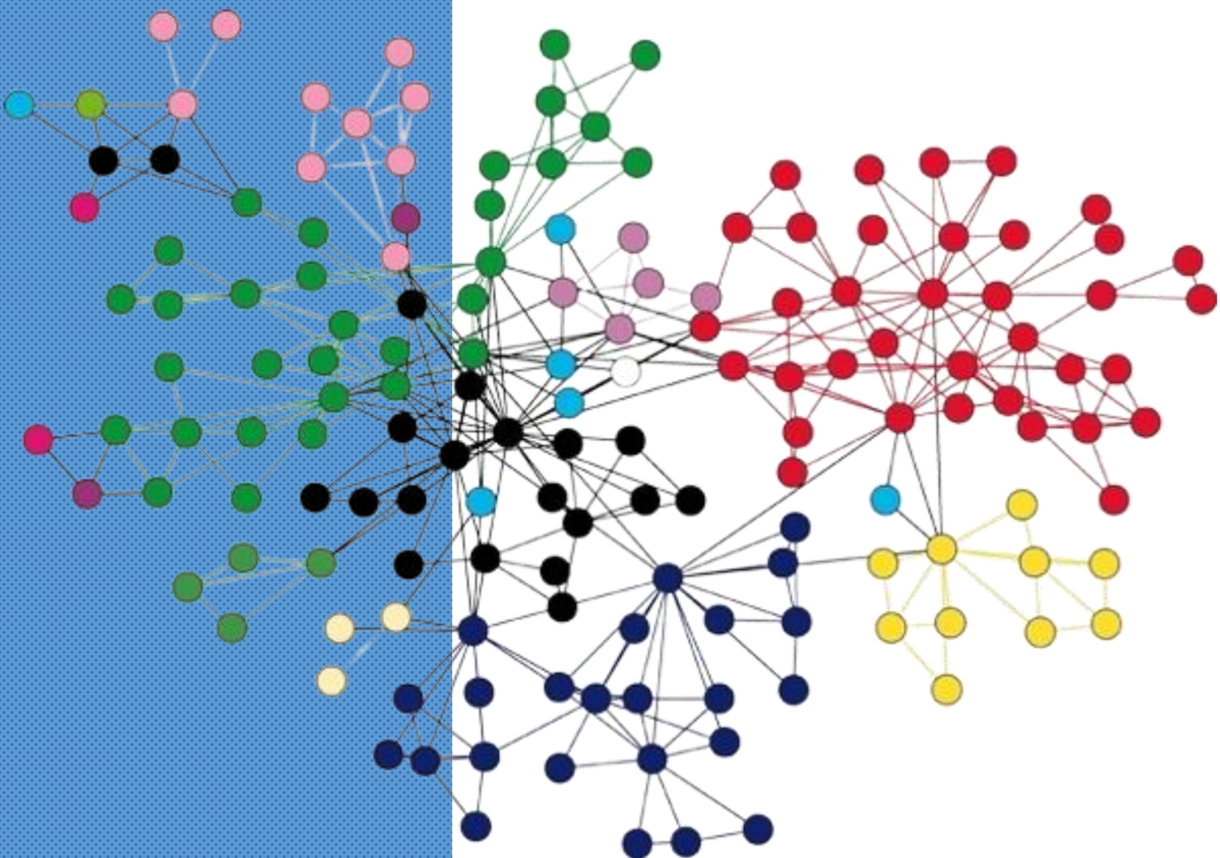
DEFINITION OF DATA STRUCTURE

Data structure is a specialized format for organizing and storing data.

Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

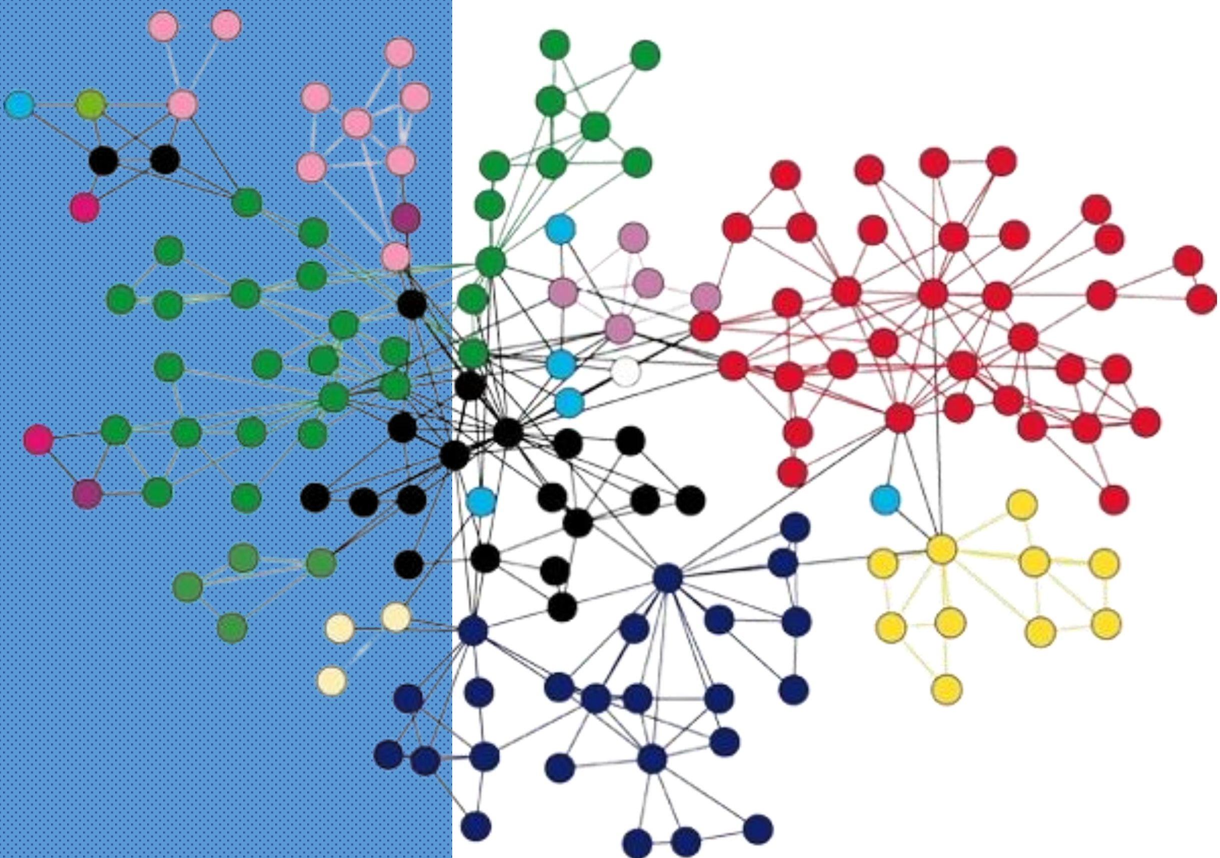
TYPES OF DATA IN DATA STRUCTURE

- Primitive and non-primitive (data type)
- Linear and non-linear (structure)
- Static and dynamic (structure)



DIFFERENCE BETWEEN **PRIMITIVE & NON PRIMITIVE** DATA TYPES

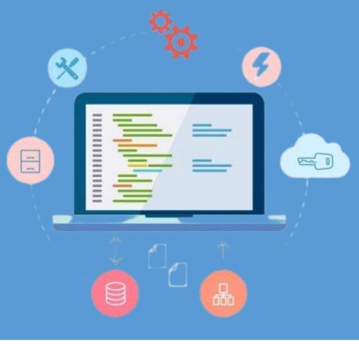
Primitive Data Types	Non-primitive Data Types
• Available in most of the programming languages • Used to represent single values • Integer • Example: ✓ Float and Double ✓ Character ✓ String ✓ Boolean	• Not defined by the programming language but created by the programmer • Used to store a group of values • Example: ✓ Arrays ✓ Structure ✓ Union ✓ Linked list ✓ Stacks ✓ Queue



DIFFERENCE BETWEEN **LINEAR &** **NON LINEAR** DATA TYPES

Linear Data Types	Non-linear Data Types
• Data elements ARE arranged sequentially or linearly • Single level is involved • Are easy to implement because computer memory is arranged in a linear way • Data elements can be traversed in a single run	• Data elements ARE arranged in hierarchically manner • Multiple levels are involved. • Not easy to implement because it utilizes computer memory efficiently • Data elements can't be traversed in a single run only.

Linear Data Types	Non-linear Data Types
• Memory is NOT utilized in an efficient way • Application: ✓ Software development • Example: ✓ Array ✓ Stacks ✓ Queue ✓ Linked List	• Memory is utilized in an efficient way • Applications : ✓ Artificial intelligence and image processing • Example: ✓ Graph ✓ Tree



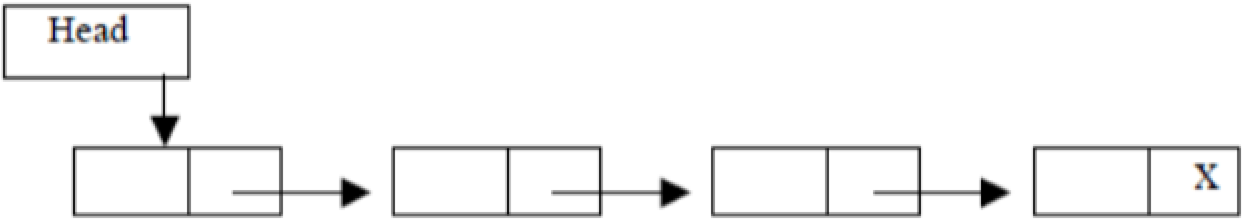
Linear

A[0]	A[1]	A[2]	....	A[9]

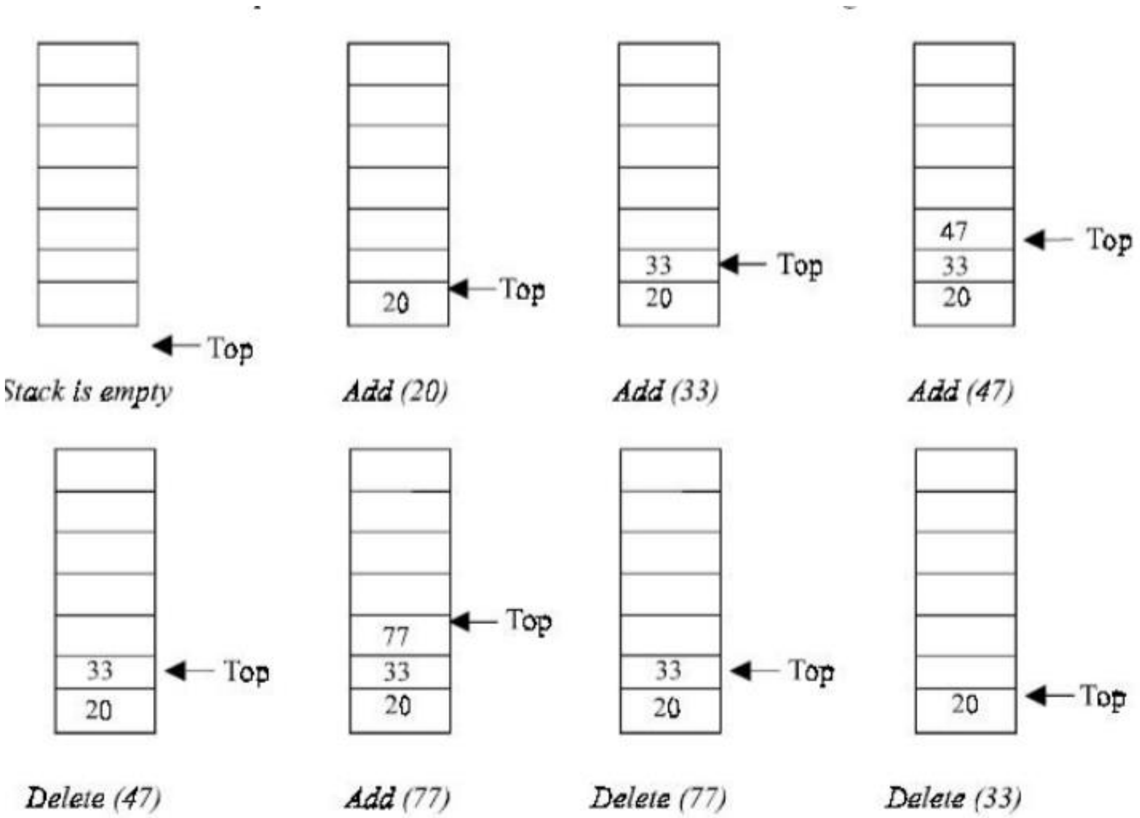
Array

A_{00}	A_{01}	A_{02}						A_{08}	A_{09}
A_{10}	A_{11}								A_{19}
A_{20}									
A_{30}									
									A_{69}
A_{70}								A_{78}	A_{79}
A_{80}	A_{81}						A_{87}	A_{88}	A_{89}
A_{90}	A_{91}	A_{92}				A_{96}	A_{97}	A_{98}	A_{99}

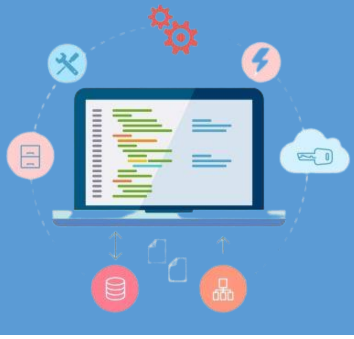
Multidimensional Array



List


Top
Stack is empty
20
Top
Add (20)
33
20
Top
Add (33)
47
33
20
Top
Add (47)
33
20
Top
Delete (47)
77
33
20
Top
Add (77)
33
20
Top
Delete (77)
20
Top
Delete (33)

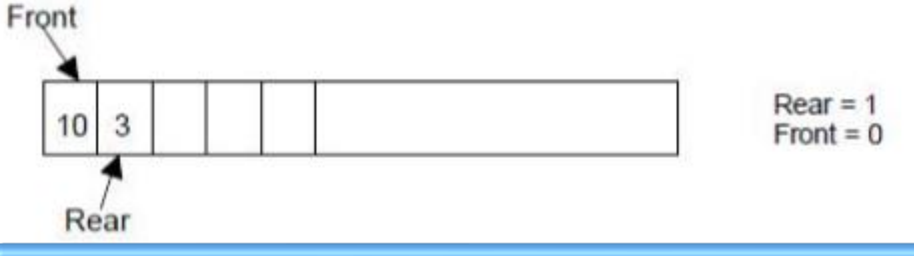
Stack



Linear

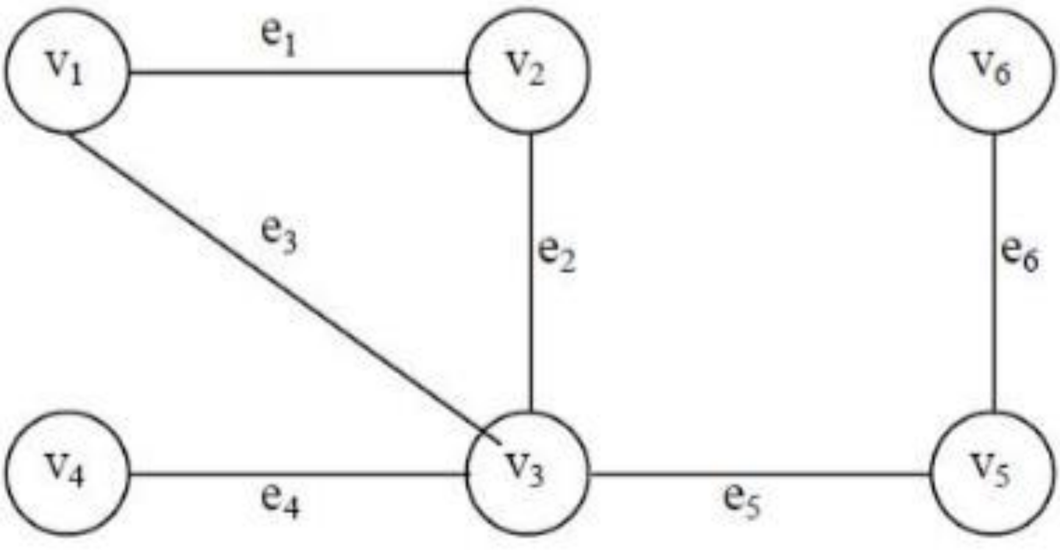
Fig. 4.1. Queue is empty.

Fig. 4.2. push(10)



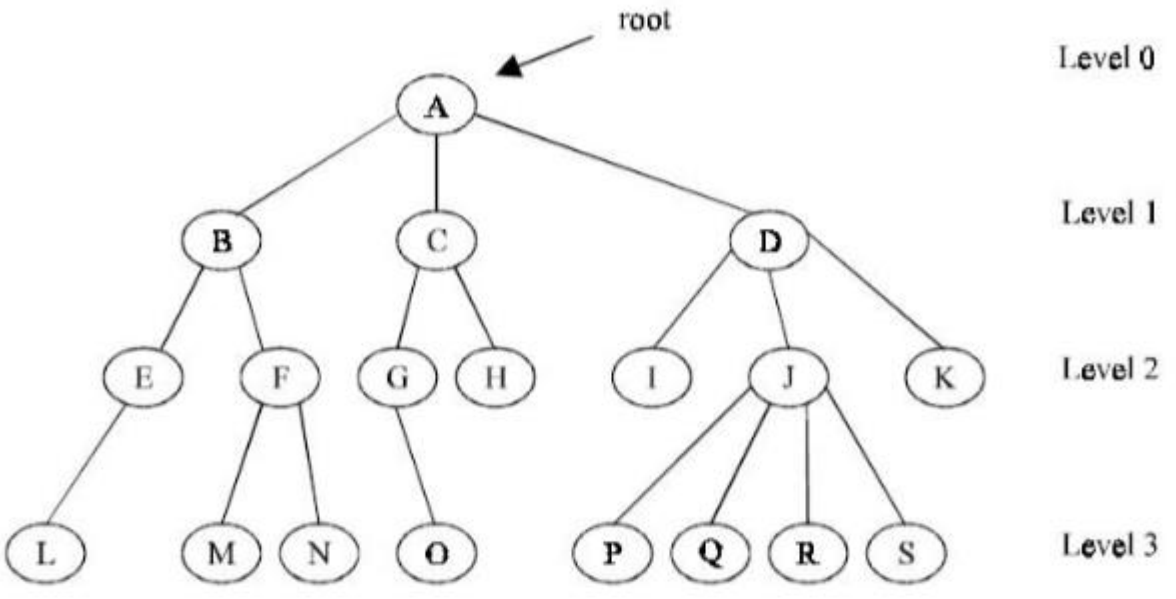
Queue

Non - Linear

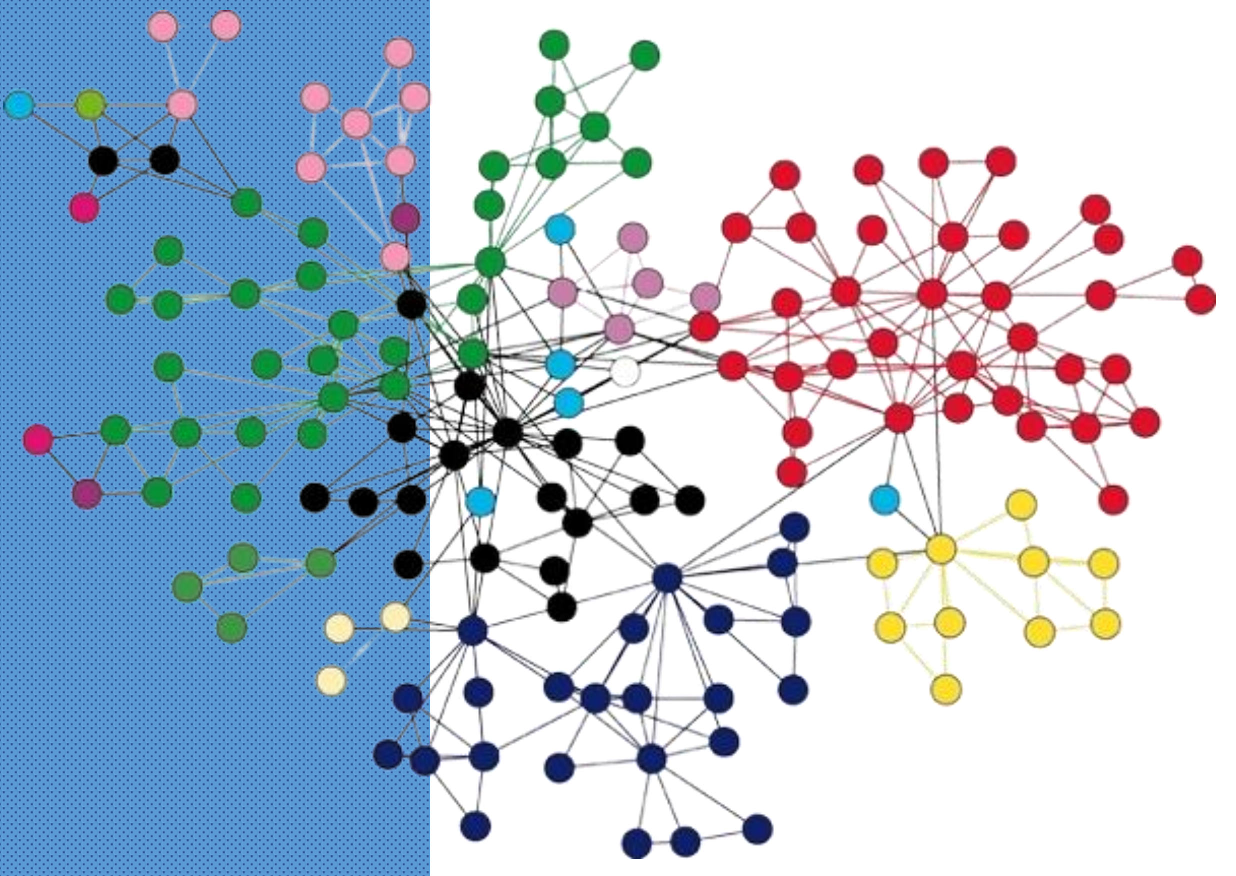


Graph

Non - Linear

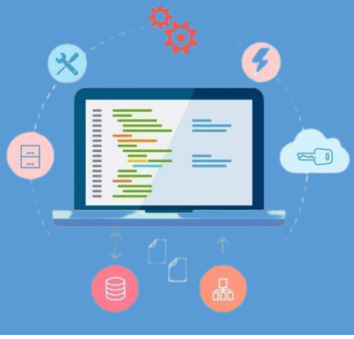


Graph



DIFFERENCE BETWEEN **STATIC &** **DYNAMIC** BEHAVIOUR (STRUCTURE)

Static Behaviour (Structure)	Dynamic Behaviour (Structure)
• The size of the structure is fixed – once created the size cannot be change • Very good for storing a well-defined number of data items • Example: Array	• The data structure is allowed to grow and shrink as the demand for storage arises – size can be change while running • The number of items to be stored is not known before hand, • Need to set a maximum size to help avoid memory collisions • Example: Tree



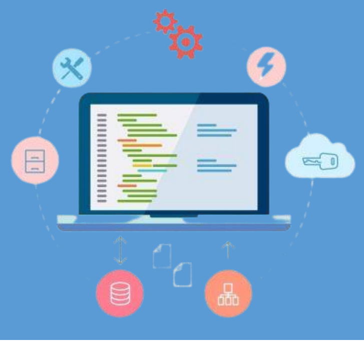
Static Behaviour (Structure)

Advantages Static Behaviour

- Compiler allocates spaces
- Easy to program
- Easy to check overflow
- Allow arrays random access

Disadvantages Static Behaviour

- Have to estimate the size needed
- Memory waste



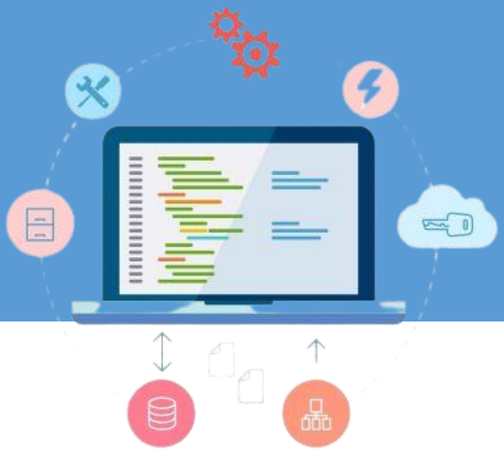
Dynamic Behaviour (Structure)

Advantages Dynamic Behaviour

- Only use what memory is needed
- Efficient use of memory

Disadvantages Dynamic Behaviour

- Hard to program
- Searching is slow



Selection of Data Structure

There are many considerations to be taken into account when choosing the best data structure for a specific program:

- Size of data
- Speed and manner data use
- Data dynamics, as change and edit.
- Size of required storage
- Fetch time of any information from data structure

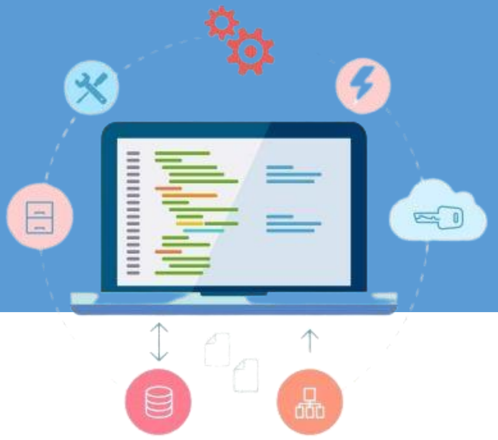
Structure

Structure is a collection of heterogeneous data.

It's create user-defined type.

Structure members are referred by its unique name.

Structure members are accessed by its variable as '.' operator.



General Syntax to Define structure and declare structure

define

```
struct struct_name {
        member_type1 member_name1;
        member_type2 member_name2;
        member_type3 member_name3;
};
```

Example:

"Define and declare a structure type called Book with three members bookName (25 character), bookID and bookPrice in a different data type. "

```
struct Book{
        char bookName[25];
        int bookID;
        float bookPrice;
};
```

However, memory has not been allocated after structure declaration.

To allocate memory of a given structure type

To allocate memory of a given structure type and work with it, we need to create variables of a given structure type.

```
struct Book{
      char bookName[25];
      int bookID;
      float bookPrice;
} Book1;
```

Variable name as Book1

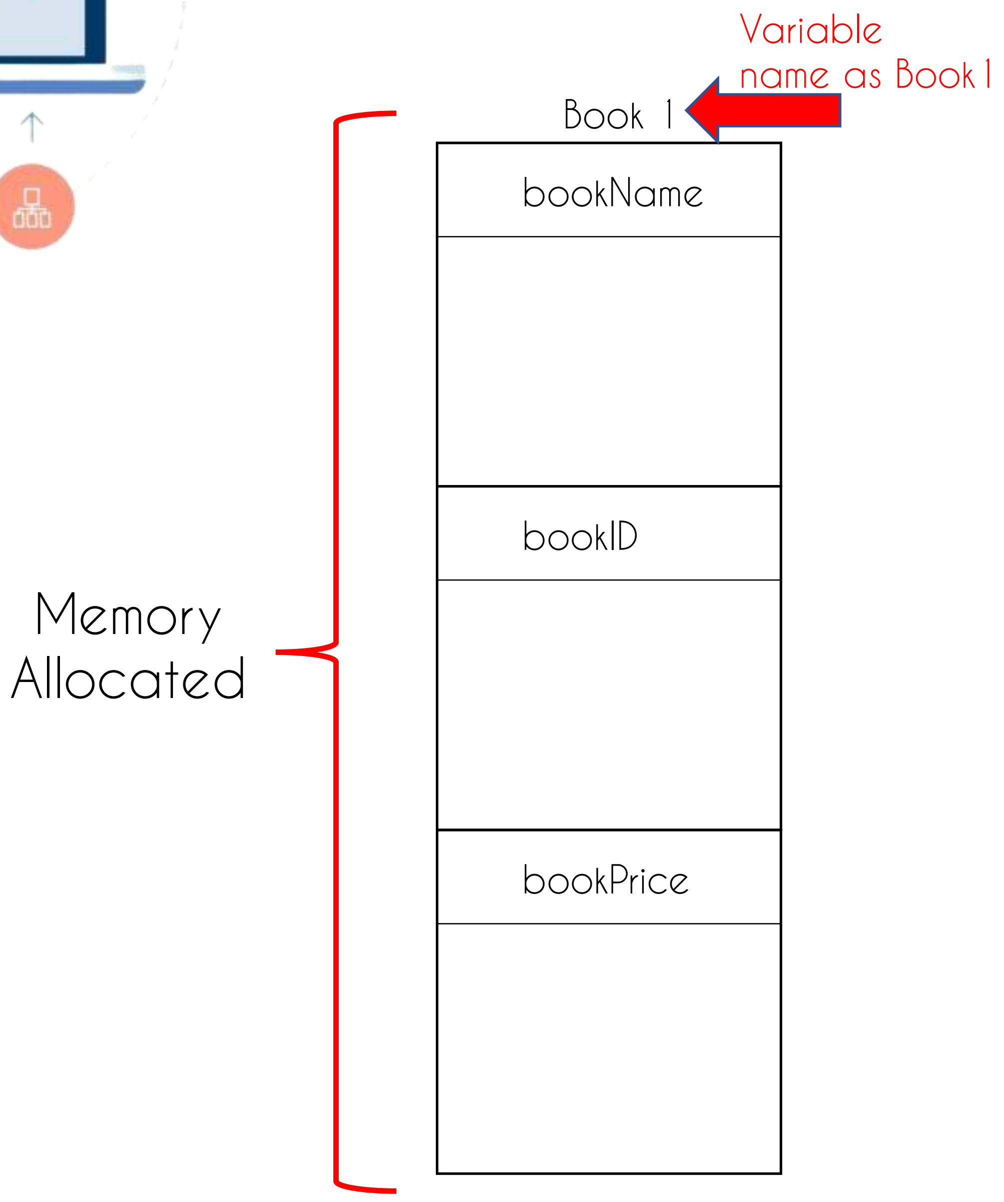
OR

```
struct Book{
      char bookName[25];
      int bookID;
      float bookPrice;
};
Book Book1;
```

Variable name as Book1

Memory is allocated after the declaration of a variable of a structure type.

Memory allocated when variables of a given structure type created



Memory is allocated after the declaration of a variable of a structure type.

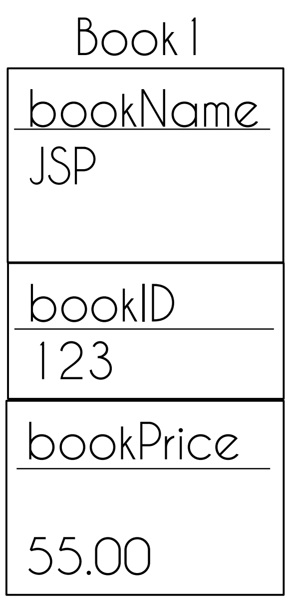
Assigning values into variables member in structure

Assigning value into each variable member in a structure by accessing each member using variables created from the type of structure declared.

```
Book1.bookID = 123;
Book1.bookPrice = 55.00;
```

Structure

To allocate memory of a given structure type and work with it, we need to create variables.



Assign value into variable members in structure:

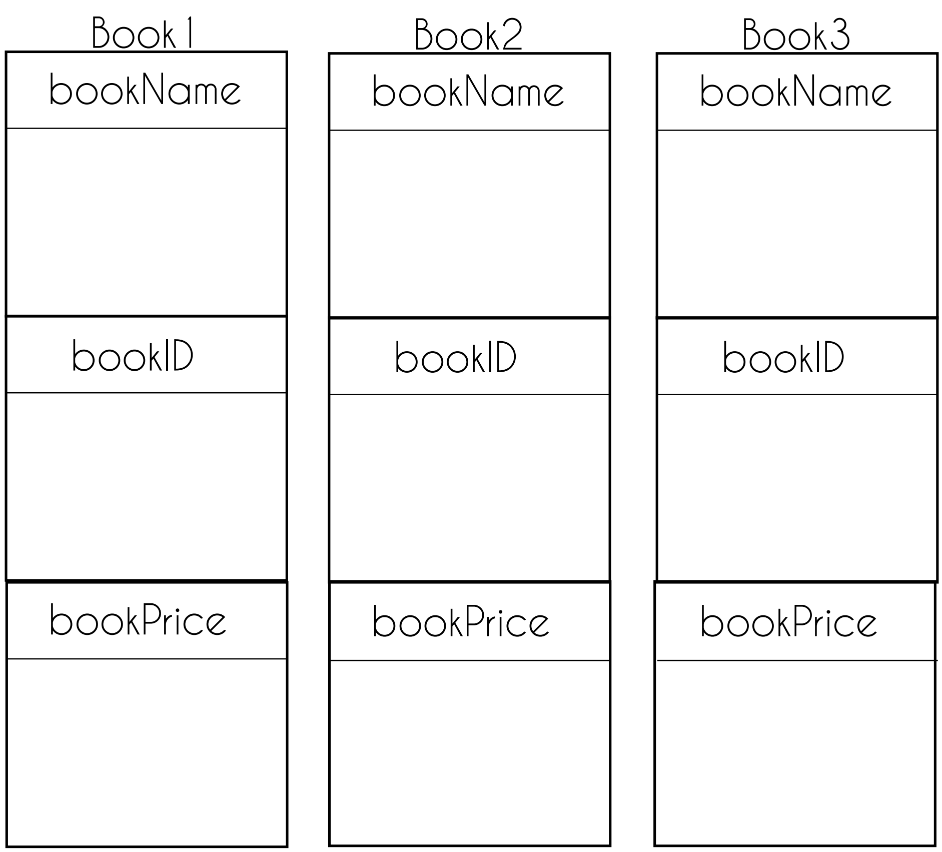
Book1. bookName= “JSP”;

Book1. bookID = 123;

Book1. bookIPrice= 55.00;

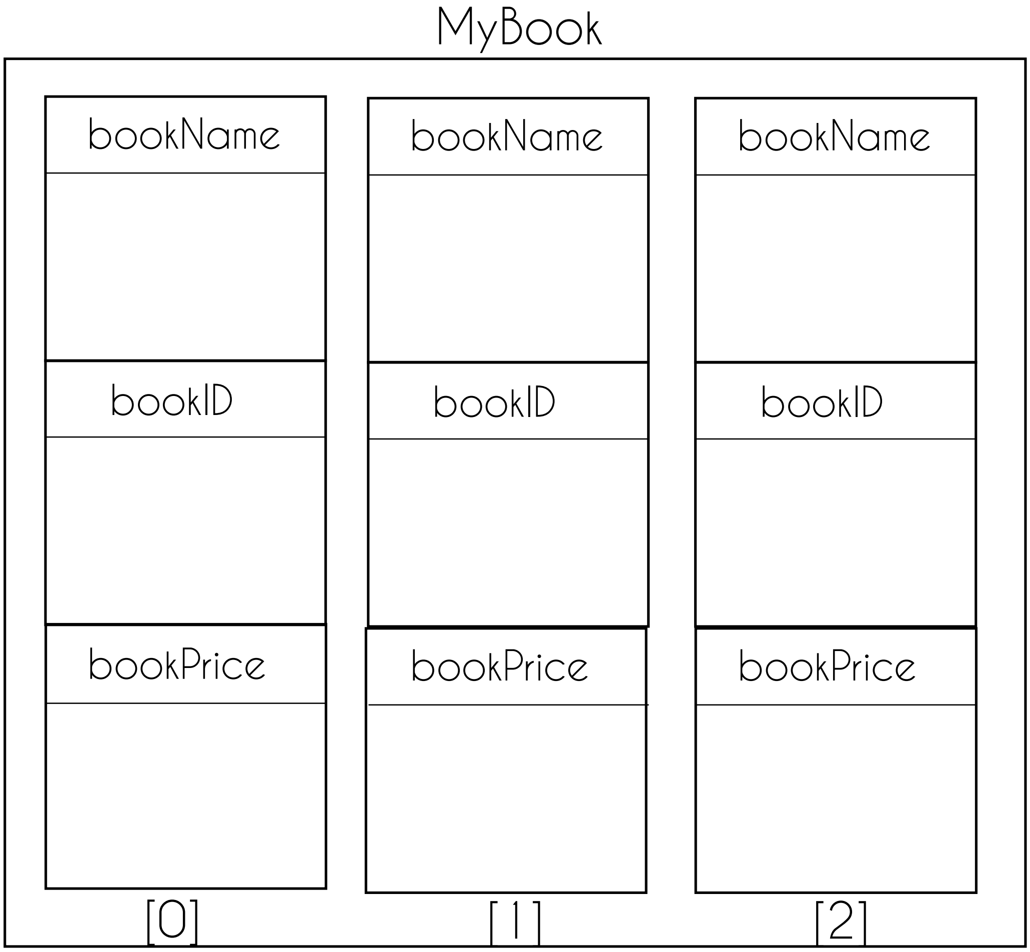
Create variables with the same Structure

```
struct Book{
        char bookName[25];
        int bookID;
        float bookPrice;
   } Book1, Book2, Book3 ;
```



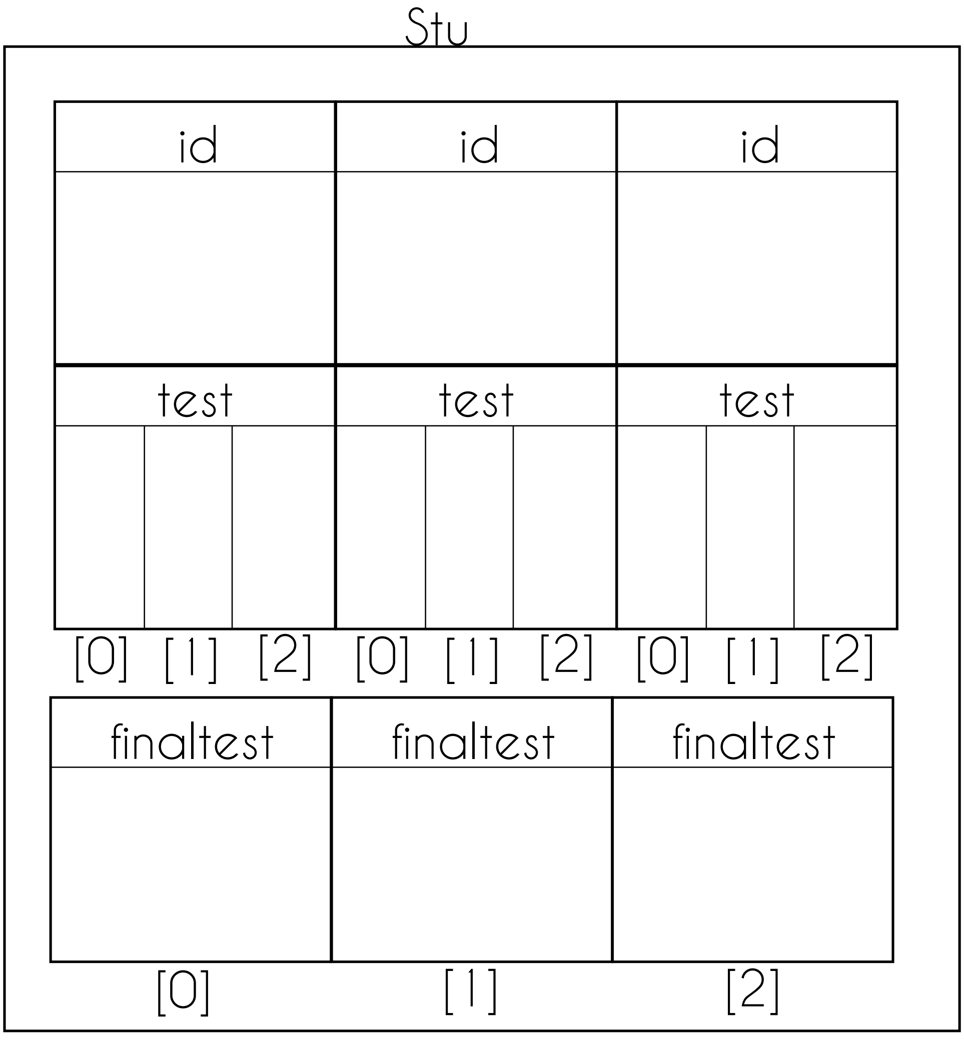
Array As Structure

```
struct Book{
        char bookName[25];
        int bookID;
        float bookPrice;
    }  MyBook[3] ;
```



Array As Structure Member

```
struct Student{
        int id;
        float test[3];
        float finaltest;
    } Stu[3];
```



Disadvantages Of An Array

1. We must know in advance that how many elements are to be stored in array.

2. Array is static structure. It means that array is of fixed size. The memory which is allocated to array can not be increased or reduced.

3. Since array is of fixed size, if we allocate more memory than requirement then the memory space will be wasted. And if we allocate less memory than requirement, then it will create problem.

4. The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.

Activity

a. A tree is a dynamic data structure.

i) State the meaning of the term dynamic when applied to data structure.

....

....

ii) State one disadvantage to programmer of using dynamic data structures compared with static data structures.

....

....

iii) State one type of data structure which must be static.

....

....

Activity

b. Define a data structure named "Pelajar".

....

c. Based on answer in previous question (a), declare the following data members in a structure Pelajar
   i) "nopend" with a character type
   ii) "nama" with a character type
   iii) "umur" with an integer type
   iv) "gpa" with a floating point type

....

....

....

....

d. Based on answer in previous question (b), declare a variable named "objek" using structure type of Pelajar.

....

....

Activity

e. Based on answer in previous question (c), access data members in struct Pelajar using variable "objek" by assigning following values to each data members.

i) "nopend" with a value of your own registration number
ii) "nama" with your own name
iii) "umur" with your own age
iv) "gpa" with your current gpa

....

....

....

....

....

....

CHAPTER 2

LIST & LINKED LIST

LIST

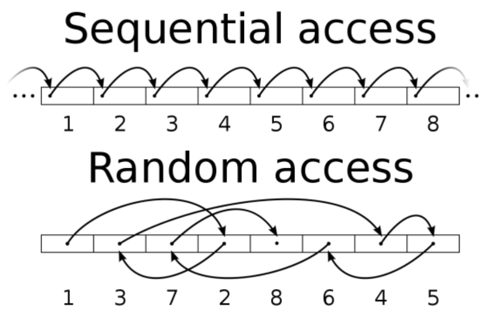
What is List ?

The list is a **collection** of data, elements, components or objects of the same data type.

- List a group of student which will have same data such as name, matric number
- List a group of staff which will have same data such as name, staff number, identity card number.

LIST

A list is a sequential data structure



- ❑ lists are stored sequentially in memory
- ❑ the elements are stored one after the other
- ❑ element data are faster to access
- ❑ addition or deletion of elements data is slow

It differs from the stack and queue data structures in that additions and removals can be made at any position in the list

ILLUSTRATION OF LIST

01

Initialize
Create a new empty List named L with size 5

L

[0]	[1]	[2]	[3]	[4]

02

Add(0,A,L)
adds the value A to list L at position 0

L

[0]	[1]	[2]	[3]	[4]
A				

03

Add(1,B,L)
adds the value B to list L at position 1

L

[0]	[1]	[2]	[3]	[4]
A	B			

04

Add(2,C,L)
adds the value C to list L at position 2

L

[0]	[1]	[2]	[3]	[4]
A	B	C		

05

Add(1,X,L)
adds the value X to list L at position 1 (shifting subsequent elements up)

L

[0]	[1]	[2]	[3]	[4]
A	X	B	C	

ILLUSTRATION OF LIST

06

Set(2,Z,L)

Set(2,Z,L)updates the values at position 2 to be Z

L

[0]	[1]	[2]	[3]	[4]
A	X	Z	C	

07

Remove(Z,L)

Remove value Z

(shifting subsequent elements down)

L

[0]	[1]	[2]	[3]	[4]
A	X	C		

08

Get(2,L)

returns the value of the third element which is C

C

09

IndexOf(X,L)

returns the index of the element with value X, which is 1

1

SHIFTED IN LIST

Shifted up

The time taken to add element near the start of the list take longer than additions near the middle or end list.

L

A	B	C		
[0]	[1]	[2]	[3]	[4]

To add X at index 1, B and C have to shifted up one step forward

L

A	X	B	C	
[0]	[1]	[2]	[3]	[4]

Shifted up

The time taken to add in the list does depend on the size of the list except to add an element at the end of the list.

Shifted down

The time taken to remove element near the start of the list take longer than removing near the middle or end list.

L

A	X	Z	C	
[0]	[1]	[2]	[3]	[4]

After remove Z at index 2, C have to shifted down one step backward

L

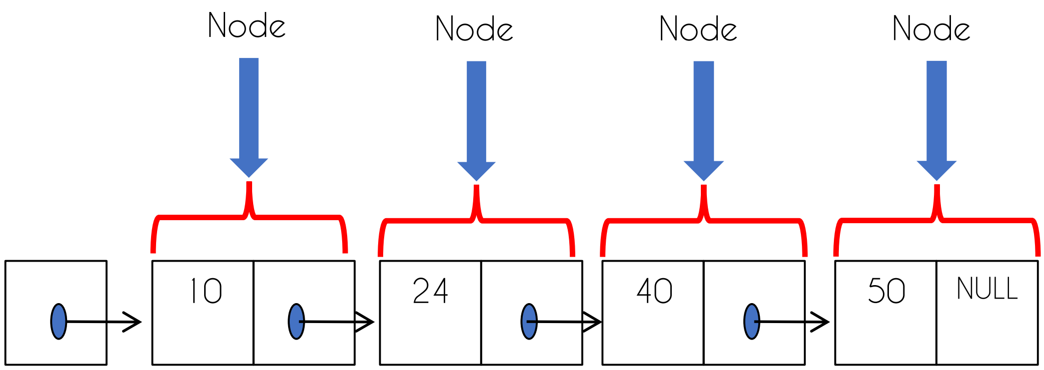
A	X	C		
[0]	[1]	[2]	[3]	[4]

Shifted down

The time taken to remove in the list does depend on the size of the list except to remove an element at the end of the list.

LINKED LIST

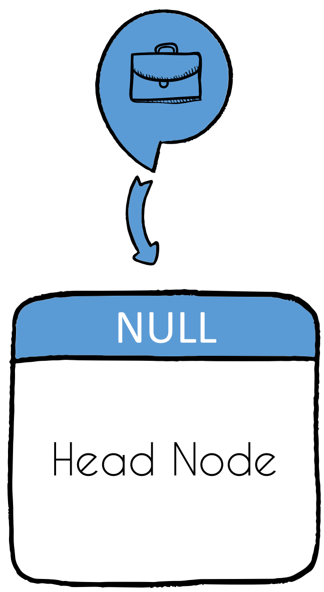
A linked list is a series of connected nodes where each node consists of an element of data and one or more pointers to other nodes.



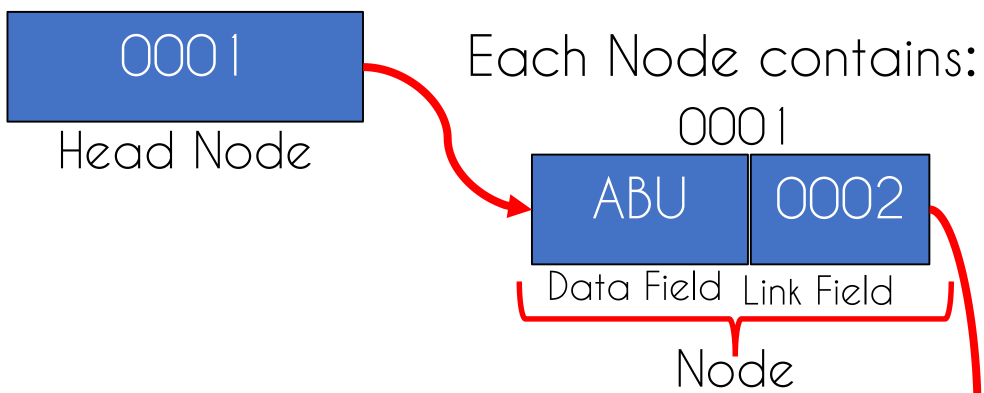
- Linked list consist of at least one head node.
- Head node acts as a pointer to the first node in linked list and contains the address of the first node.
- The most important concept in linked list is the node that point/link to other node.

LINKED LIST

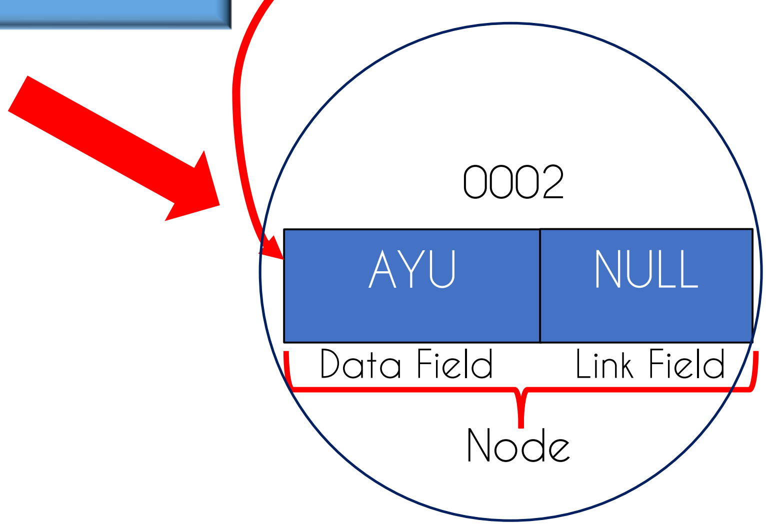
Linked list is said to be empty when it does not contain any node or head node contains the value NULL.

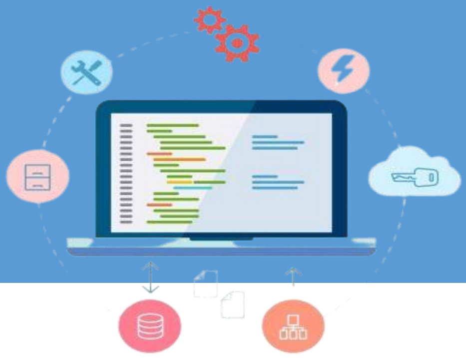


Linked list must consist of at least **one head node**



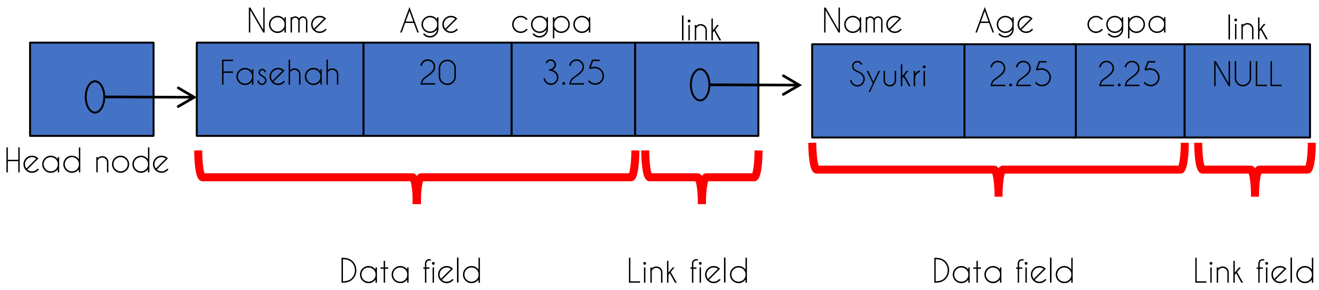
Node containing data AYU is known as **Last Node** in a Linked List

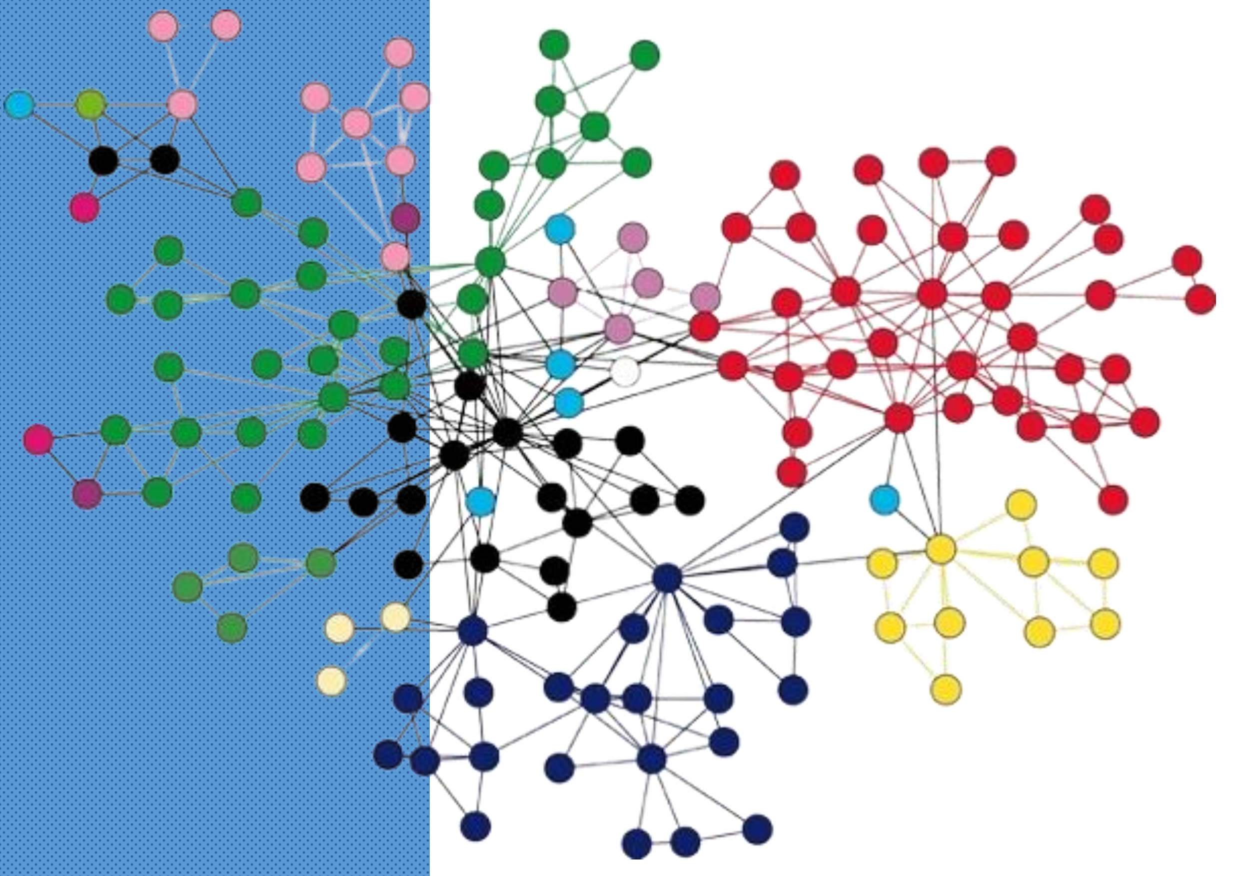




Node In A Linked List

Each Node can contains more than one data:





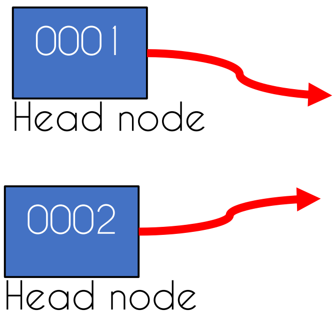
DIFFERENCE LIST & LINKED LIST

List	Linked List
• Elements are stored in linear order, accessible with an index. • Have a fixed size, it is static data structure. • Can access the previous element easily • Insertions and Deletions are not efficient because of shifting element. • Waste of memory if the size of list is bigger than the size of data.	• Elements are stored in linear order, accessible with links. • Do not have a fixed size, it is dynamic data structure. • Cannot access the previous element • Insertions and Deletions are efficient because of no shifting element. • There is no waste of memory.

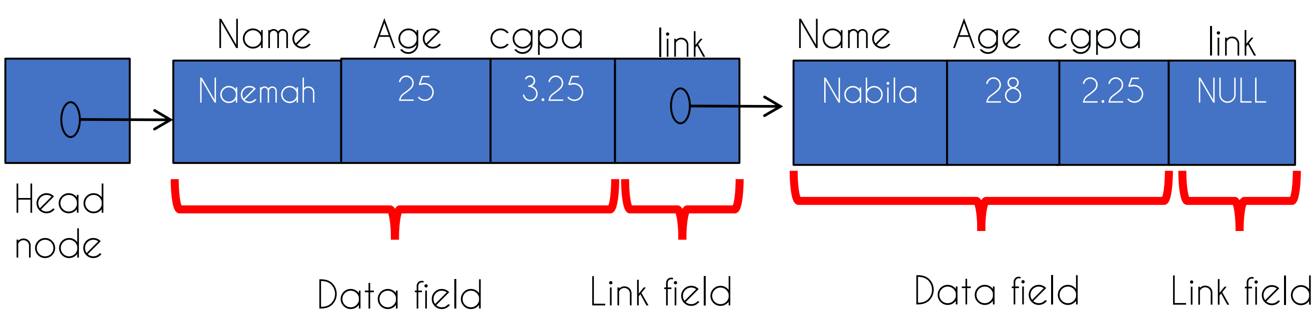
List	Linked List
• Sequential access is faster because elements in contiguous memory locations allocation. • Requires less memory because List only holds actual data and its index	• Sequential access slow because elements not in contiguous memory locations allocation • Requires more memory because each node holds data and reference to next and previous elements.

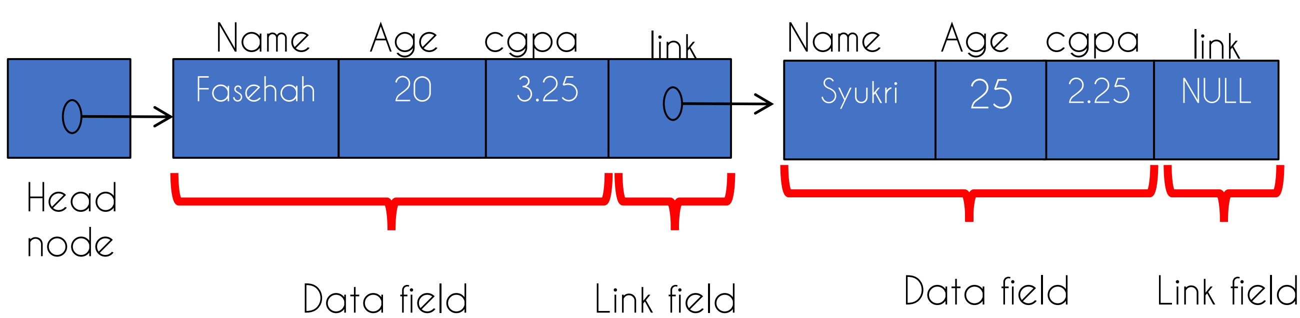
Memory Management In A Linked List

Memory Management Linked list not in contiguous memory locations allocation

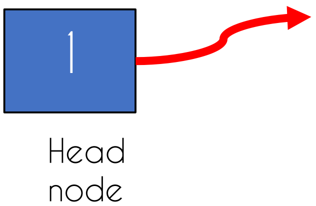


	Name.	Age	cgpa	link
0001	Naemah	25	3.25	0003
0002	Fasehah	20	3.25	0004
0003	Nabila	28	2.25	Null
0004	Syukri	25	2.25	Null

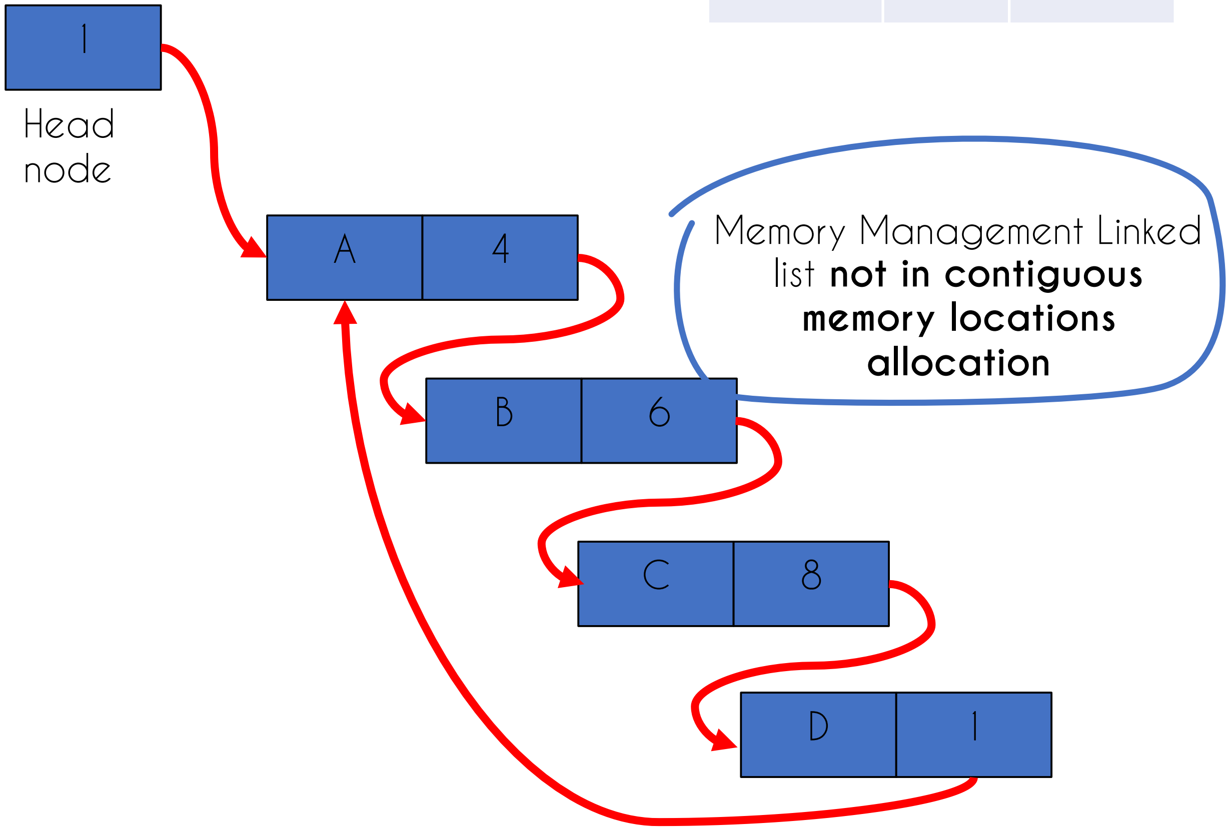




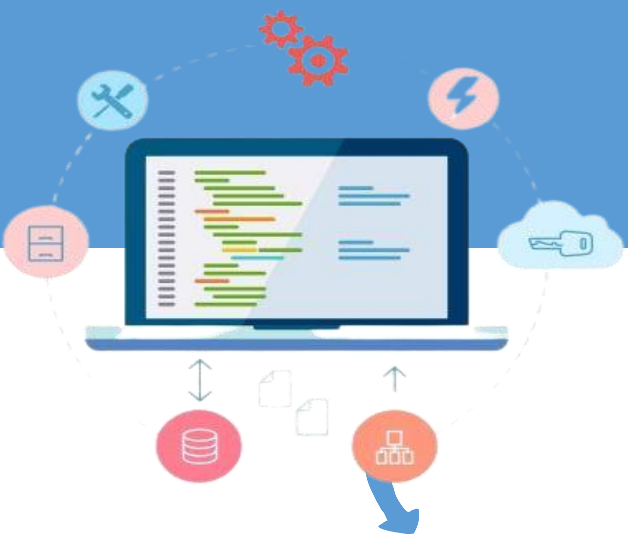
Memory Management In A Linked List



Address	Data Field	Link Field
1	A	4
2		
3		
4	B	6
5		
6	C	8
7		
8	D	1



Advantages Of Linked List



Dynamic Data Structure

It can grow and shrink at runtime by allocating and deallocating memory. There is no need to give initial size of linked list

Insertion and Deletion

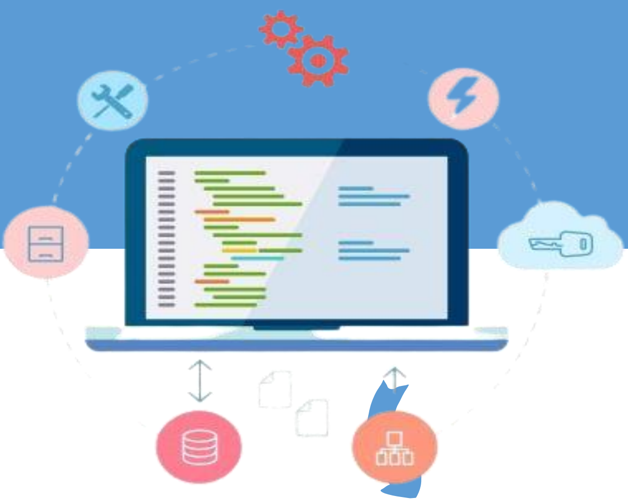
don't have to shift elements after insertion or deletion of an element

No Memory Wastage

memory is allocated only when required

Implementation

Data structures such as stack and queues can be easily implemented using linked list.



Disvantages Of Linked List

Memory Usage

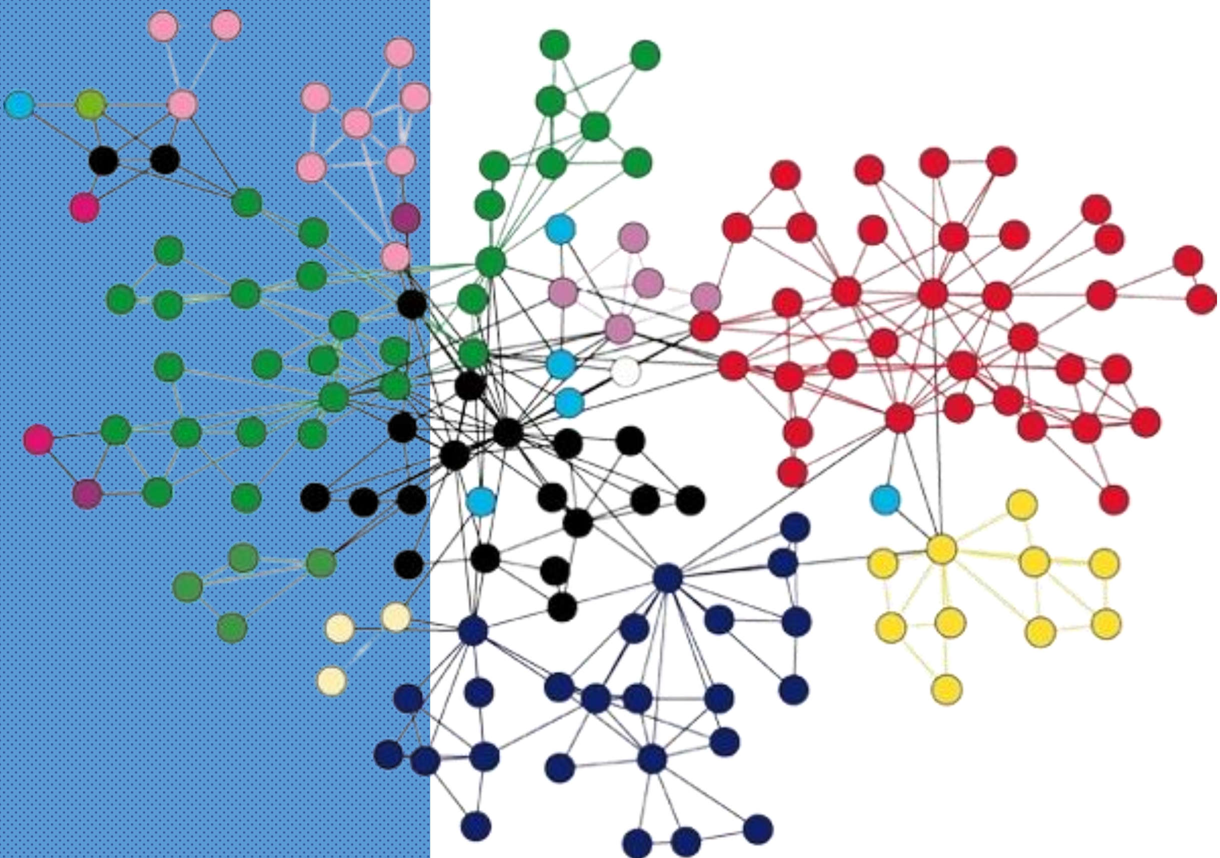
More memory is required to store elements in linked list as compared to array. Because in linked list each node contains a pointer and it requires extra memory for itself.

Traversal

Elements or nodes traversal is difficult in linked list. We can not randomly access any element as we do in array by index. For example if we want to access a node at position n then we have to traverse all the nodes before it. So, time required to access a node is large

Reverse Traversing

In linked list reverse traversing is really difficult. In case of doubly linked list its easier but extra memory is required for back pointer hence wastage of memory.



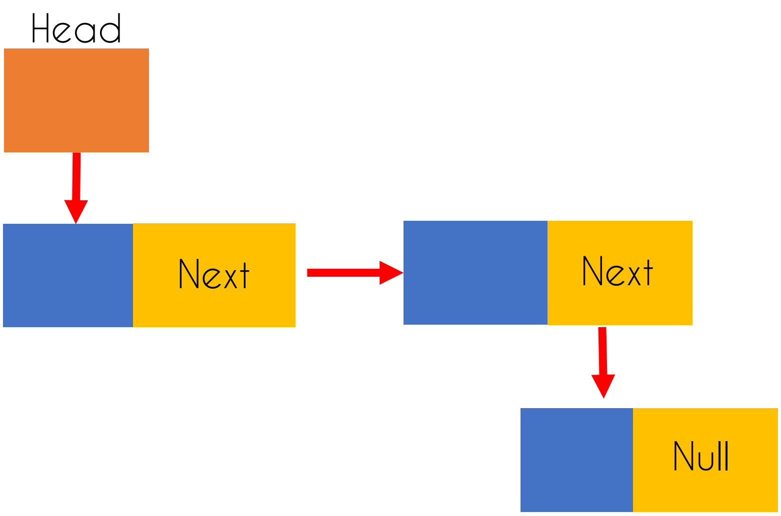
TYPE OF **LINKED LIST**

Types of Linked List

01

Single Linked List

- can be **traversed in only one direction** from head to the last node.
- each node contains **only one link** field pointing the next node in the list.
- last node contain value NULL.

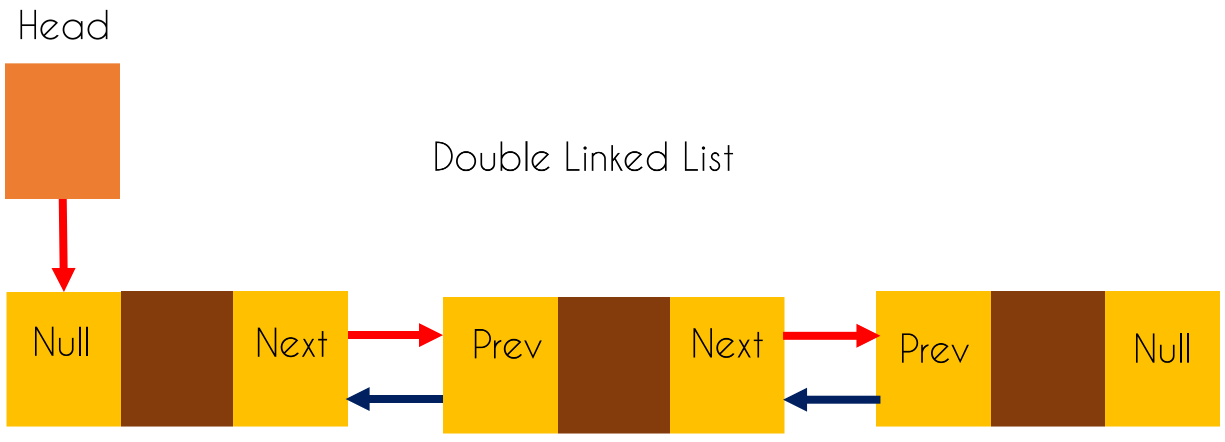


Types of Linked List

02

Double Linked List

- can be **traversed in either forward and backward easily** as compared to Single Linked List.
- each node contains two link field to point to next node and previous node in the linked list.
- **First node** contains value of null in previous link field
- **First node** contains value of null in next link field
- Playlist MP3



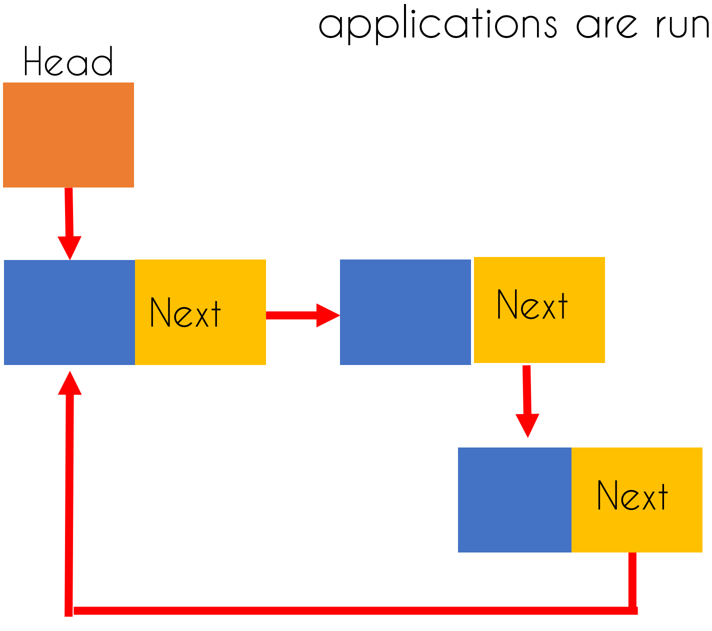
Double Linked List

Types of Linked List

03

Circular Linked List

- all nodes are connected to form a circle
- last node contains the address of the first node in link field.
- how do we know when we have finished traversing the list?
- the real life application where the circular linked list is used is our Personal Computers, where multiple applications are running.

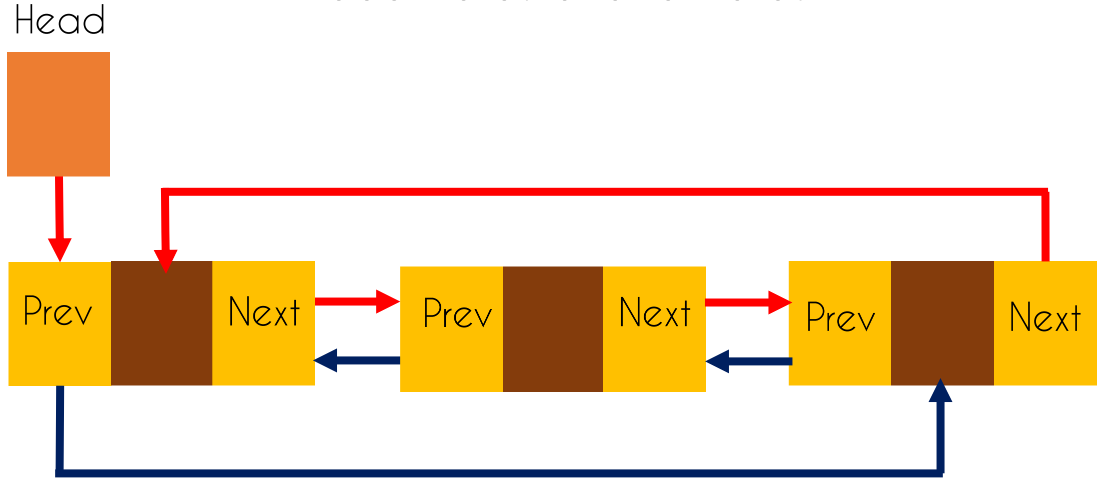


Types of Linked List

Circular Doubly Linked List

04

- the last node of the list contains the address of the first node in next link field
- the first node of the list contains the address of the last node in previous link field
- doesn't contain NULL in any of the node
- convenient to traverse lists backwards and forwards



Activity

1. Draw a new list based on Figure A, after adding value M to list myList at position 1.

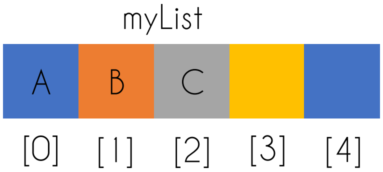


Figure A

2. From the answer in Question (1), explain the movement that occurs to the value of B and C.

3. Draw a new list based on Figure B, after removing value A from a list myList.

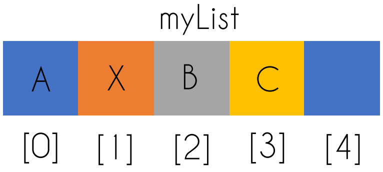


Figure B

4. From the answer in Question (3), explain the movement that occurs to the value of X, B and C.

Activity

5. Illustrate Circular linked list with 5 nodes

6. State THREE (3) types of linked list

7. State THREE (3) differences between list and linked list

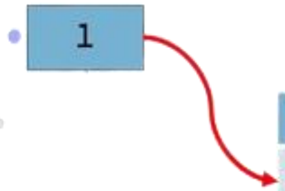
8. Draw a circular linked list based on memory representation of circular linked list in Figure A.

Address	Data	Next
1	13	4
2		
3		
4	15	6
5		
6	19	8
7		
8	57	1

Figure A

Activity

9. Draw a circular double linked list based on memory representation of circular double linked list in Figure B.



Address	Prev	Data	Next
1	4	A	2
2	1	B	3
3	2	C	4
4	3	D	1

Figure B

CHAPTER 3

STACK

INTRODUCTION TO STACK

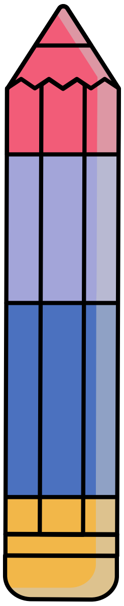
Stack is a **collection of items which is organized in a sequential manner**

Example: stack of books or stack of plates

All **additions and deletions are restricted at one end**, called top

LAST IN FIRST OUT (LIFO) data structure

Implementation Of Stack In Real Life



a person wear **bangles**

the last bangle worn is the first one to be removed

and the first bangle would be the last to be removed

This follows last in first out (LIFO) principle of stack

Batteries in the flashlight :
You can't remove the second battery unless you remove the last in. So the battery that was put in first would be the last one to take out.

This follows the LIFO principle of stack

Implementation Of Stack In Real Life

Layer of Pancake :
When you're placing pancakes on your plate you are going to put them one after another on top of each other. If you want to eat one of the pancakes in the middle of your stack you will first have to eat all the pancakes on top of the one you are trying to get to. This is like a stack data structure where if you want to get to an element in the middle of the stack you first have to remove all of the elements that are on top of it.

This follows the LIFO principle of stack

Cars in a garage :
In order to take out the car that was parked first you need to take out the car that was parked last. So the car that was parked first would be the last to take out.

This follows the LIFO principle of stack

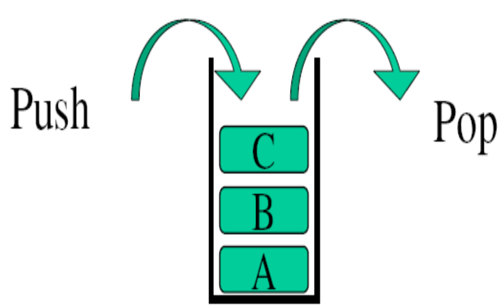
What is Stack

Stack is an abstract data type

Adding an entry on the top (push)

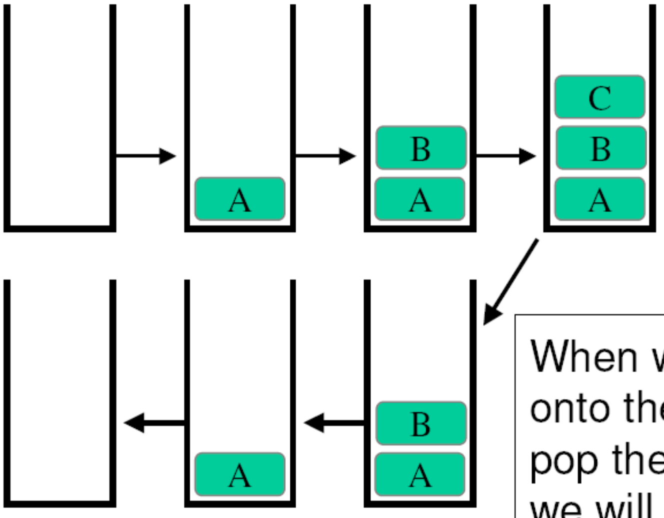
Deleting an entry from the top (pop)

A stack is open at one end (the top) only. You can push entry onto the top, or pop the top entry out of the stack



Last-in First-out (LIFO)

Push A, B, C



The last one pushed in is the first one popped out! (LIFO)

When we push entries onto the stack and then pop them out one by one, we will get the entries in reverse order.

Pop C, B, A

Stack Implementation

Stack is an abstract data structure

Item can be Integer, Double, String, and also can be any data type, such as Employee, Student...

How to implement a general stack for all those types?

We can implement stack using array or linked list.

Stack Implementation Using Array

1. Size of stack is fixed during declaration

2. Item can be pushed if there is some space available, need to check if stack is full

3. Need a variable called, top to keep track the top of a stack

4. Stack is empty when the value of Top is -1

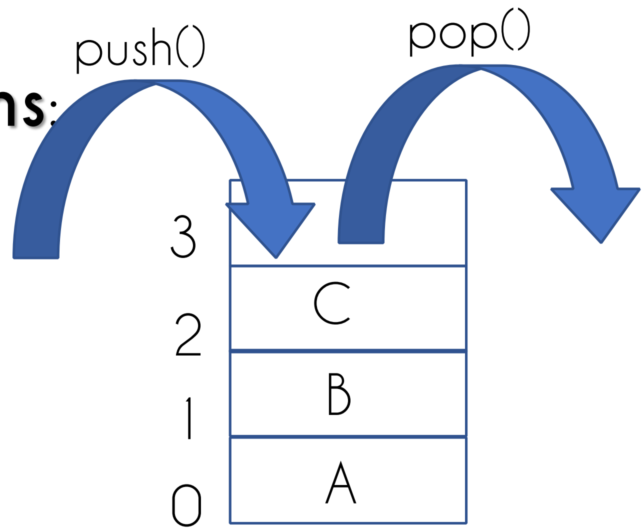
Stack Implementation Using Linked List

1. Size of stack is flexible. Item can be pushed and popped dynamically

2. Need a pointer, called top to point to top of stack

Stack Implementation Using Array

Stack Operations:



- createStack()
- push(item)
- pop( )
- stackTop()

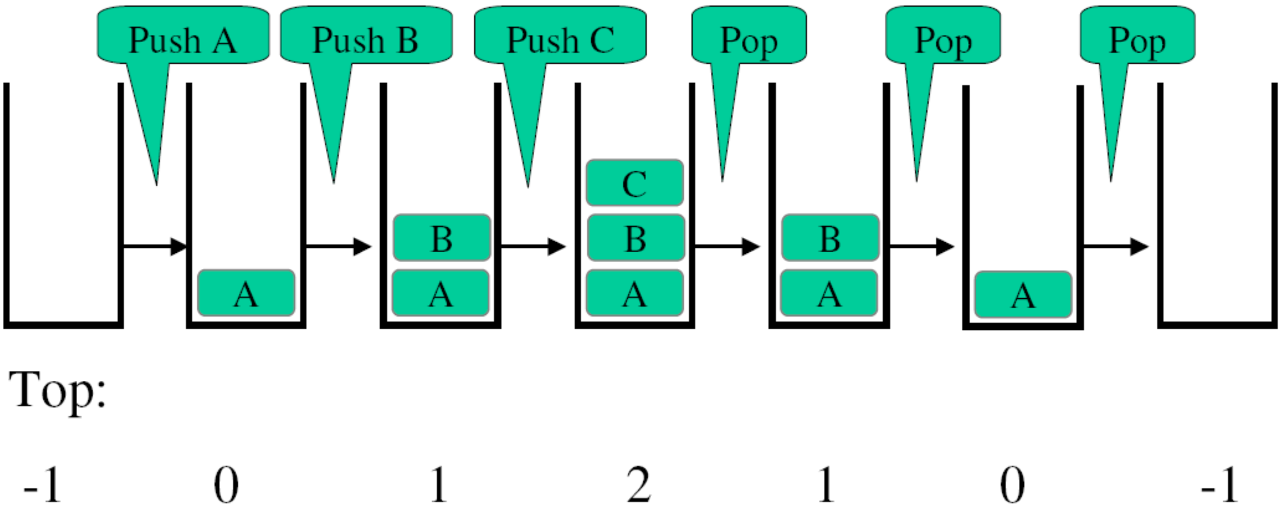
❑ createStack() will allocate fix size of an array and initialize value of variable top is -1

❑ stackTop() refer to **last data inserted to stack**

"Stack can be visualized as array, BUT the operations can be done on top stack only."

Stack Implementation Using Array

Push() and pop() operations



Stack Implementation Using Array

3 things to be considered for stack with array

1

Stack Empty : when top is -1

2

Push operations : To insert data into stack, 2 statements must be used

```
top = top + 1;
stack[top] = data;
```

3

Push operations : To delete data from stack, 2 statements must be used

```
stack[top] = null;
top = top – 1;
```

Stack Implementation Using Linked List

Stack implemented using linked list – number of elements in stack or size of stack is not restricted to certain size

Dynamic memory creation, memory will be assigned to stack when a new node is pushed into stack, and memory will be released when an element being popped from the stack

Stack using linked list implementation can be empty or contains a series of nodes

Stack Implementation Using Linked List

Each node in a stack must contain at least 2 attributes:

i. **data** – to store information in the stack.
ii. **pointer next** (store address of the next node in the stack)

Basic operations for a stack implemented using linked list:

i. **createStack()** – initialize top
ii. **push()** – insert data onto stack
iii. **pop()** – delete data from stack
iv. **stackTop()** – get data at top.

Push and pop operations can only be done at the top ~ similar to add and delete in front of the linked list.

Stack Implementation Using Linked List

Stack Operations:

- createStack()
- push(item)
- pop( )
- stackTop()

head

NULL

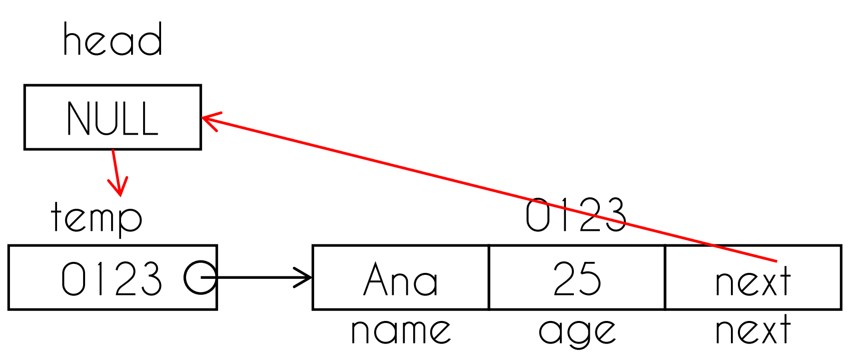
❑ createStack() will create a pointer as a head node with initialization value of null

Push() to empty stack

2 conditions for inserting element in stack

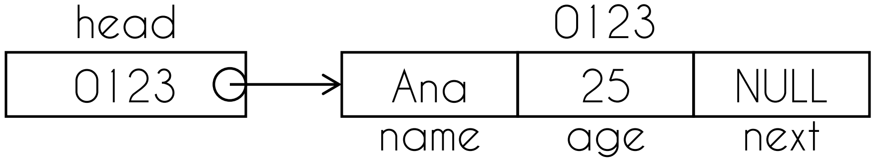
Insert to empty stack	Insert item to non empty stack : stack with value

push() to empty stack



In this situation the new node being inserted, will become the first item in stack.

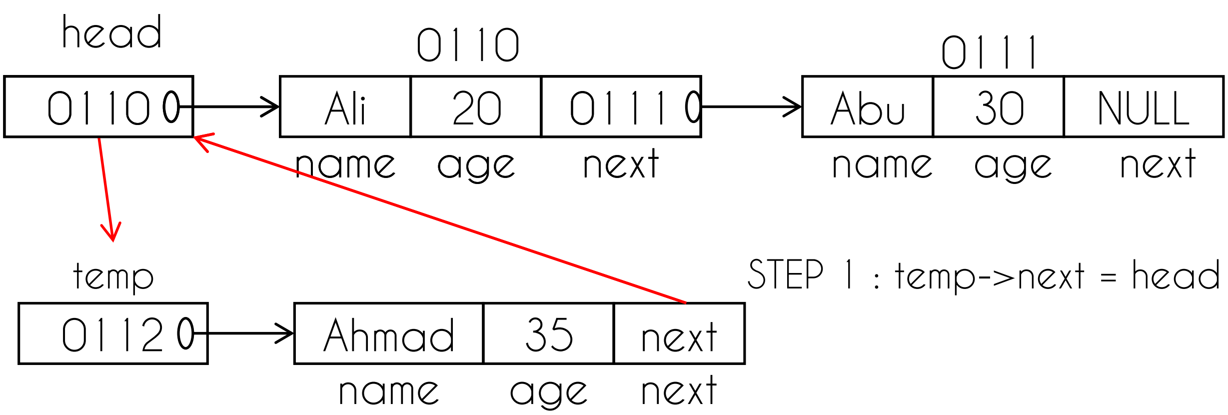
Step 1 : temp->next = head;
Step 2 : head = temp;



Stack Implementation Using Linked List

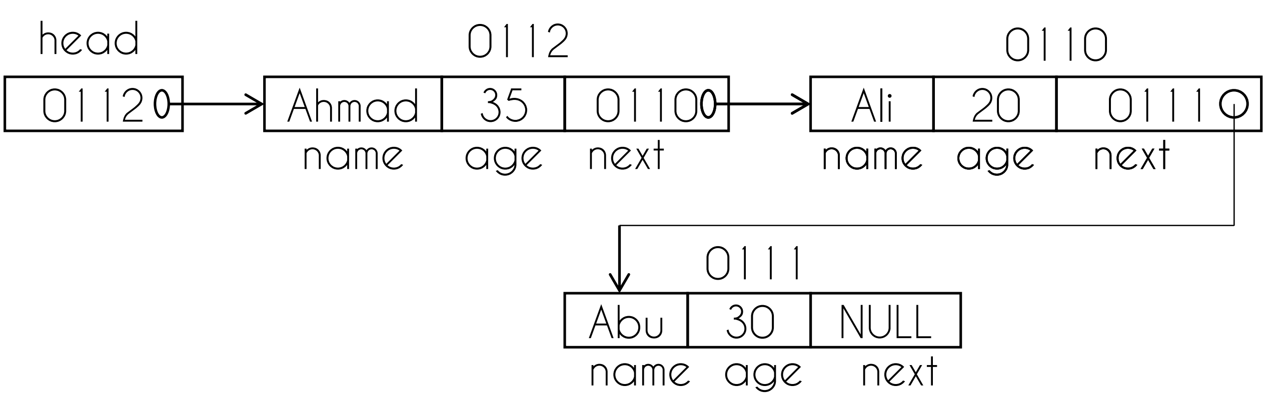
Push() to non-empty st

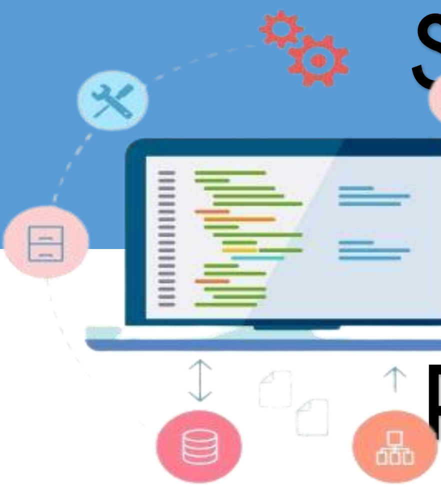
- This operation is similar to inserting element in front of a linked list. The next value for the new element will point to the top of stack and head will point to the new element



Step 1 : temp->next = head;

Step 2 : head = temp;





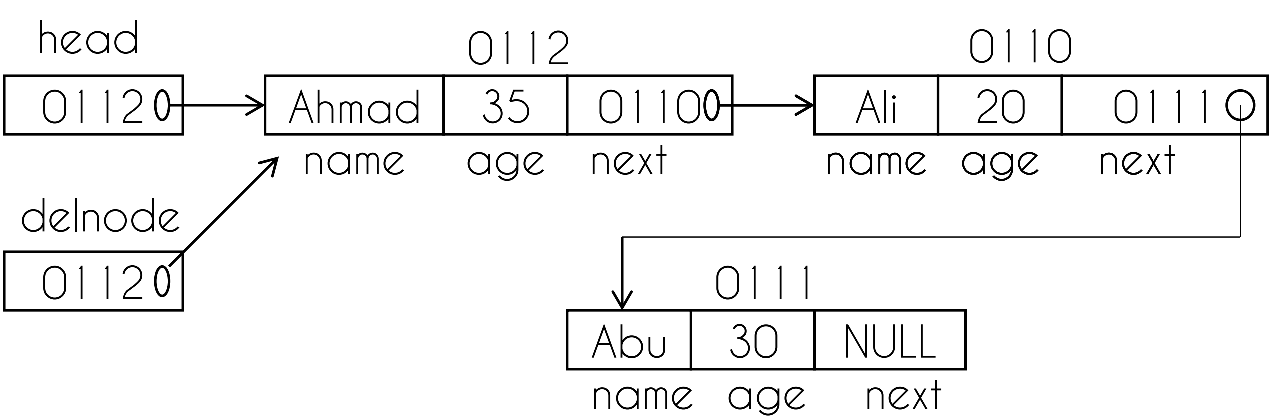
Stack Implementation Using Linked List

Pop() to non-empty st

- Pop operation can only be done to non-empty stack. Before pop() operation can be done, operation must be called in order to check whether the stack is empty or there is item in the stack. If isEmpty() function return true, pop() operation cannot be done.

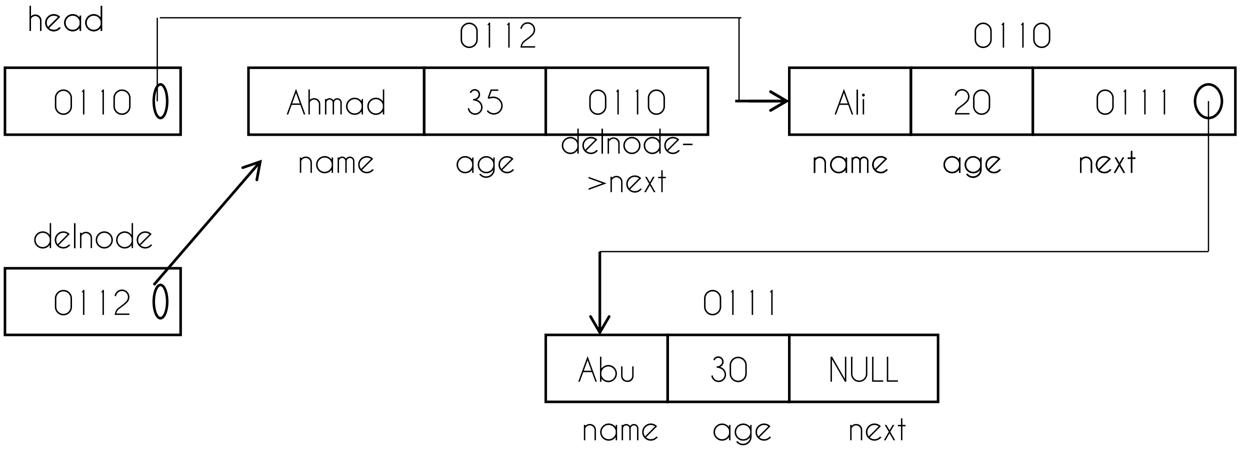
- During pop() operation, an external pointer is needed to point to the delete node. In the figure below, delnode is the pointer variable to point to the node that is going to be deleted.

Step 1 : delnode = head;

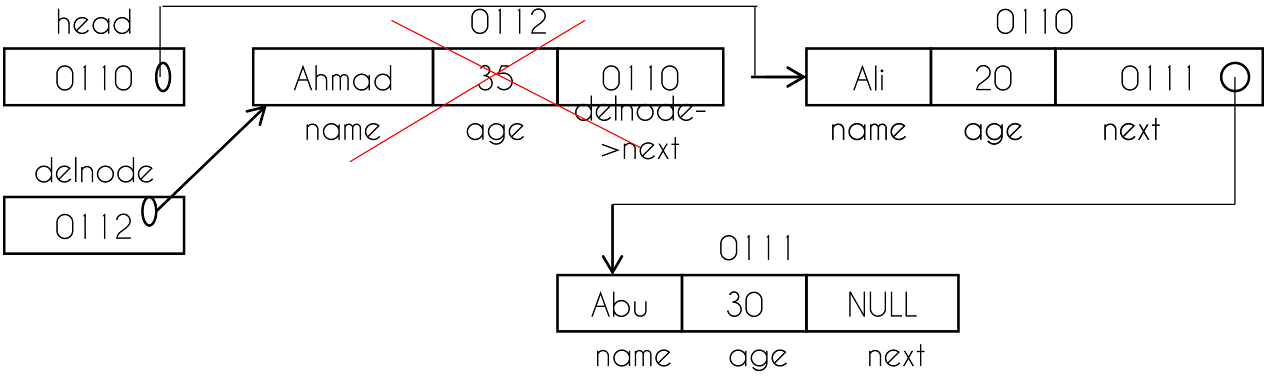


Pop() to non-empty stack

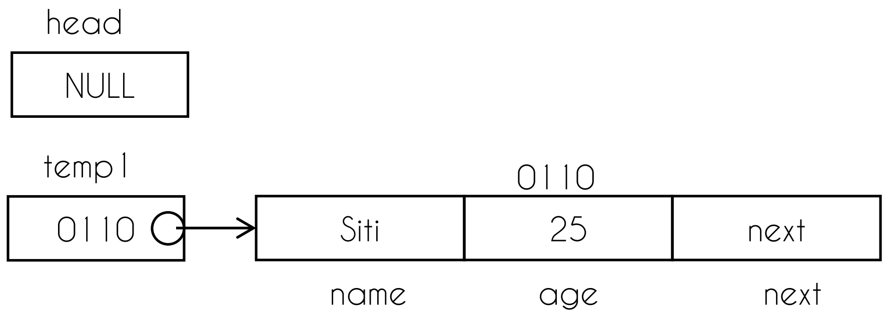
Step 2 : head = delnode -> next;



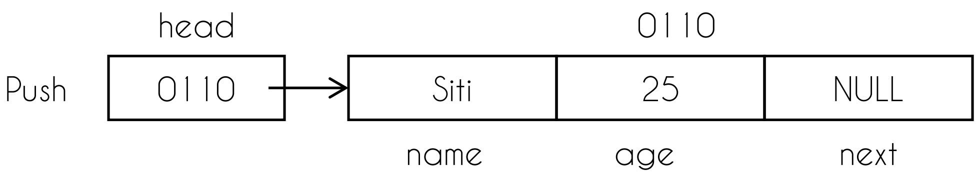
Step 3 : delete delnode;

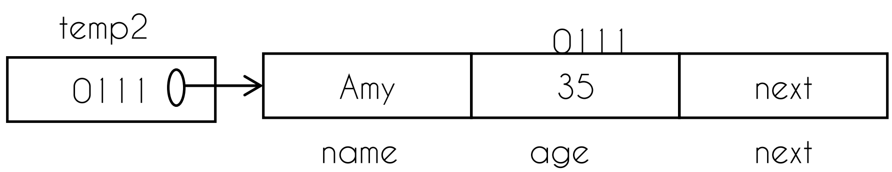


push() operations example:

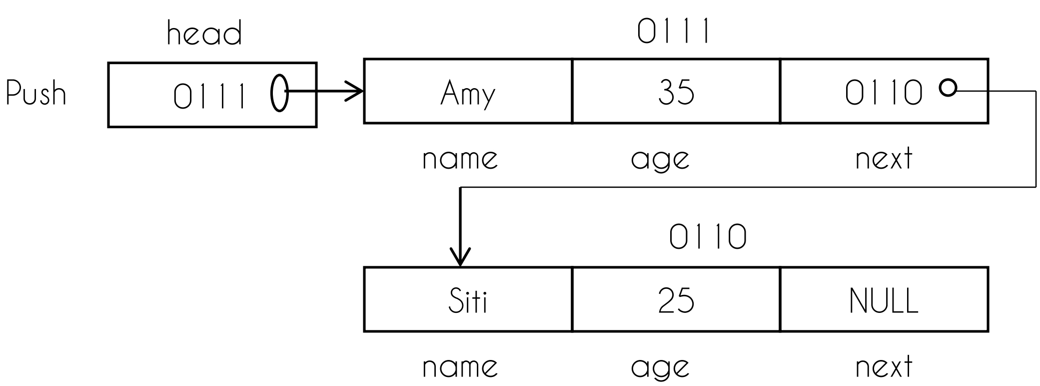


temp1->next = head;
head = temp1;



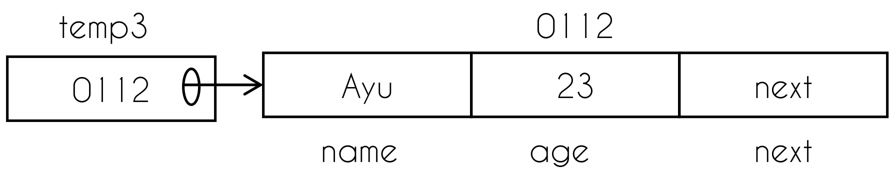


temp2->next = head;
head = temp2;

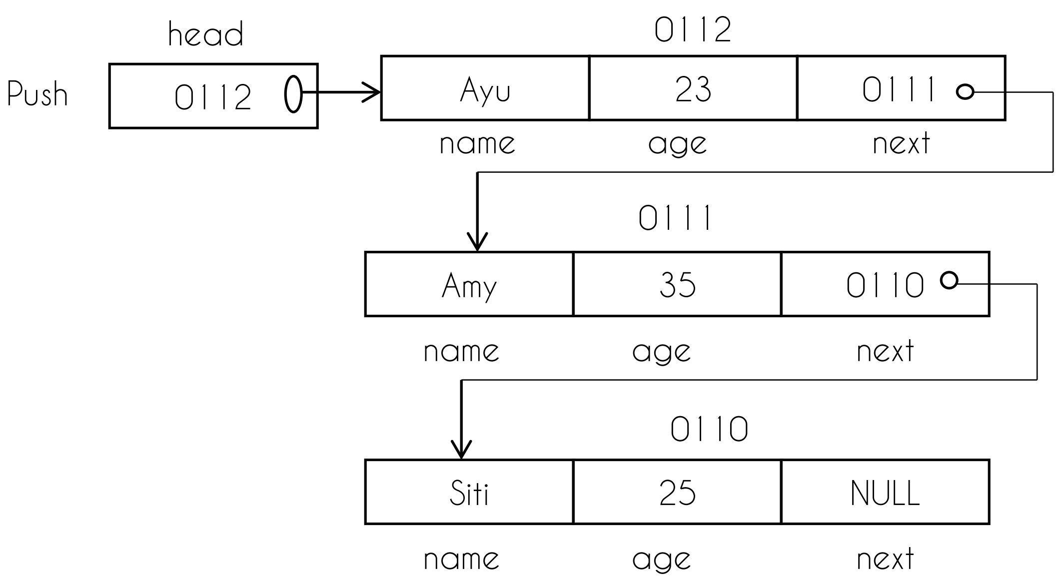


Stack Implementation Using Linked List

push() operations example:

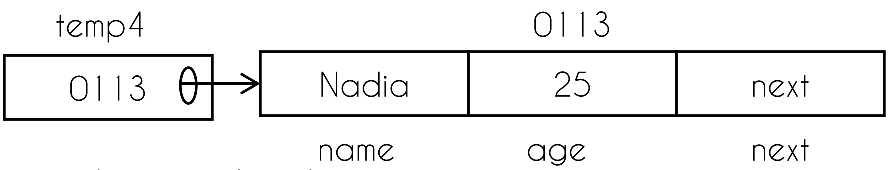



```
temp3->next = head;
head = temp3;
```



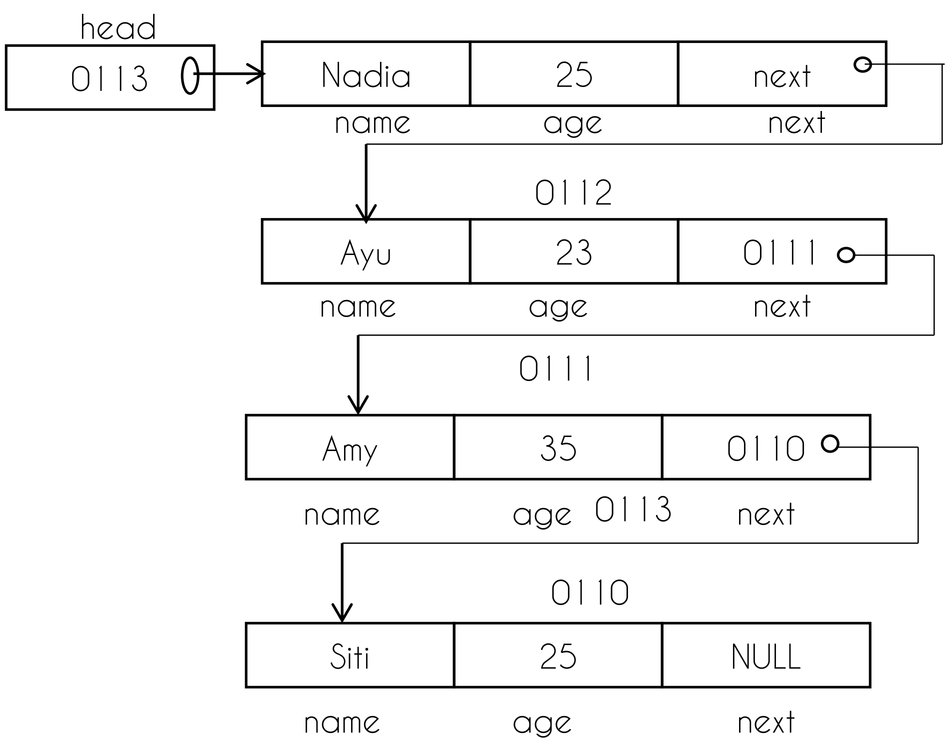
Stack Implementation Using Linked List

push() operations example:

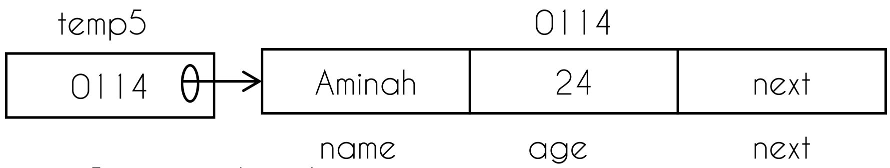


temp4->next = head;
head = temp4;

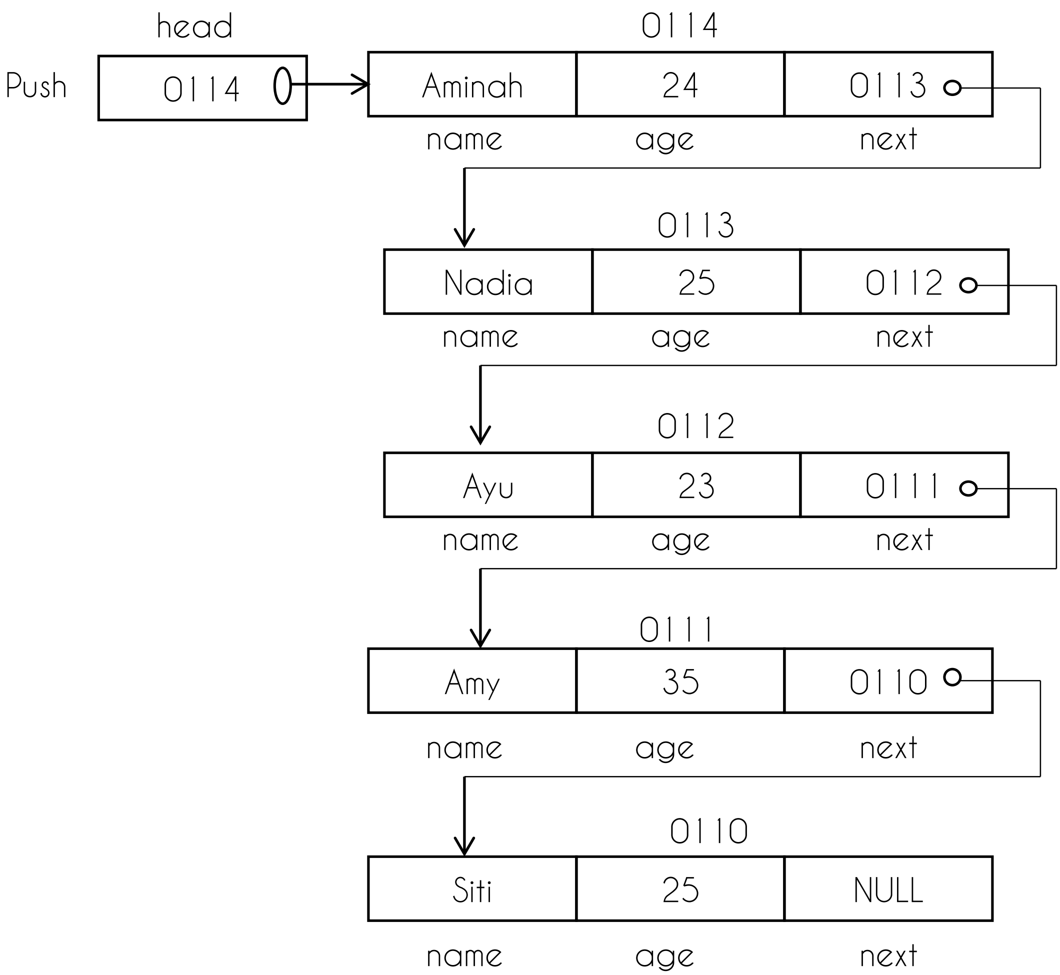
Push



push() operations example:



temp5->next = head;
head = temp5;



Stack Implementation Using Linked List

Step for pop() operations

STEP 1 : create a temporary pointer node named as delnode

STEP 2 : assign the address in pointer head node into a temporary pointer node named as delnode.

```
delnode = head;
```

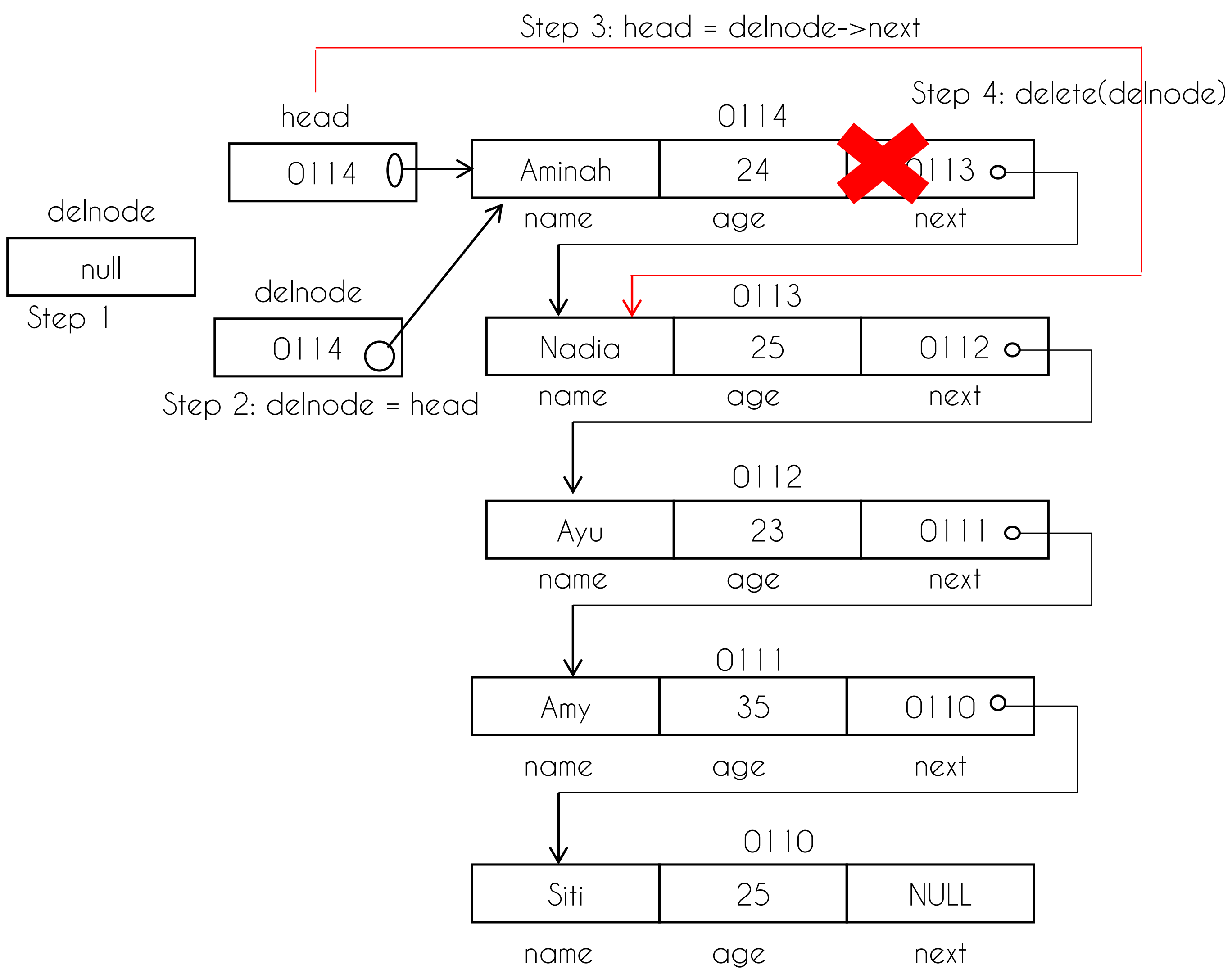
delnode will point to first node in a linked list

STEP 3 : assign the address of second node into pointer head node.

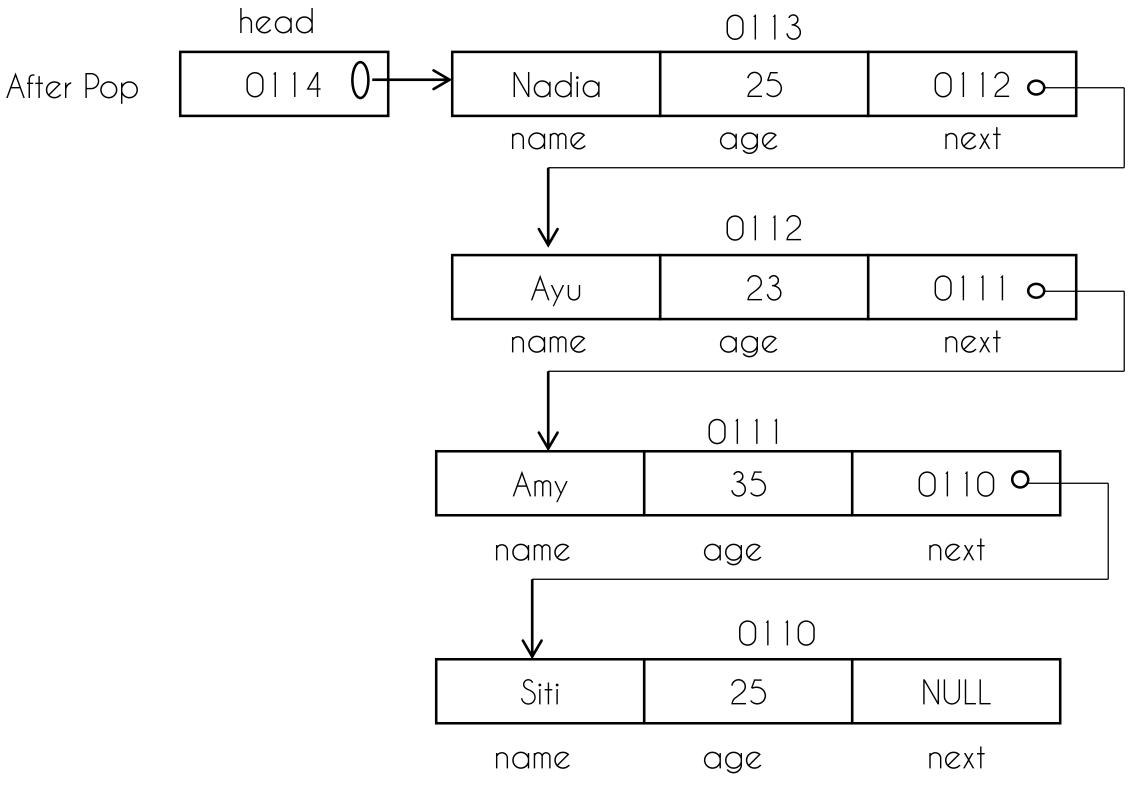
```
head = delnode -> next;
or head = head->next;
```

STEP 4 : delete(delnode);

Stack Implementation Using Linked List



pop() operations



Stack Application Examples

- Check whether parentheses are balanced (open and closed parentheses are properly paired)
- Evaluate Algebraic expressions.
- Creating simple Calculator
- Backtracking (example. Find the way out when lost in a place)

Example 1 Parentheses Balance

- Stack can be used to recognize a balanced parentheses.
- Examples of balanced parentheses.

  (a+b), (a/b+c), a/((b-c)*d)

  Open and closed parentheses are properly paired.
- Examples of not balance parentheses.

  ((a+b)*2 and m*(n+(k/2)))

  Open and closed parentheses are not properly paired.

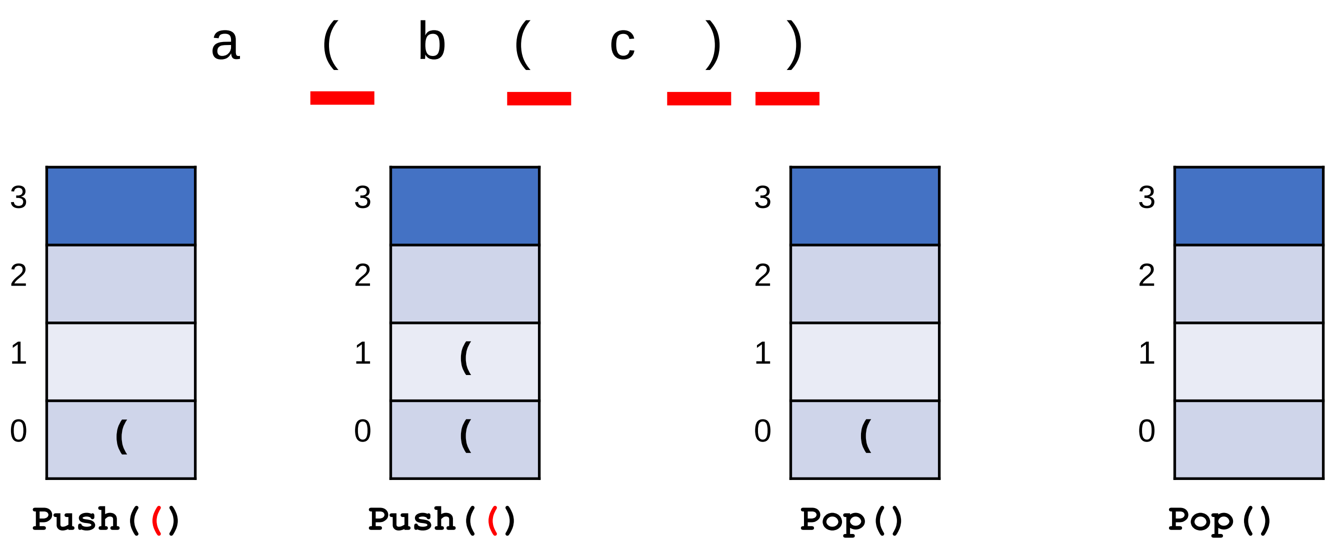
Stack Application Examples

Check for Balanced Parentheses Algorithm

- Every '(' read from a string will be pushed into stack.
- The open parentheses '(' will be popped from a stack whenever the closed parentheses ')' is read from string.
- An expression have balanced parentheses if :
  - ✓ Each time a ")" is encountered it matches a previously encountered "(".
  - ✓ When reaching the end of the string, every "(" is matched and stack is finally empty.
- An expression does NOT have balanced parentheses if :
  - ✓ When there is still ')' in input string, the stack is already empty.
  - ✓ When end of string is reached, there is still '(' in stack.

Stack Application Examples

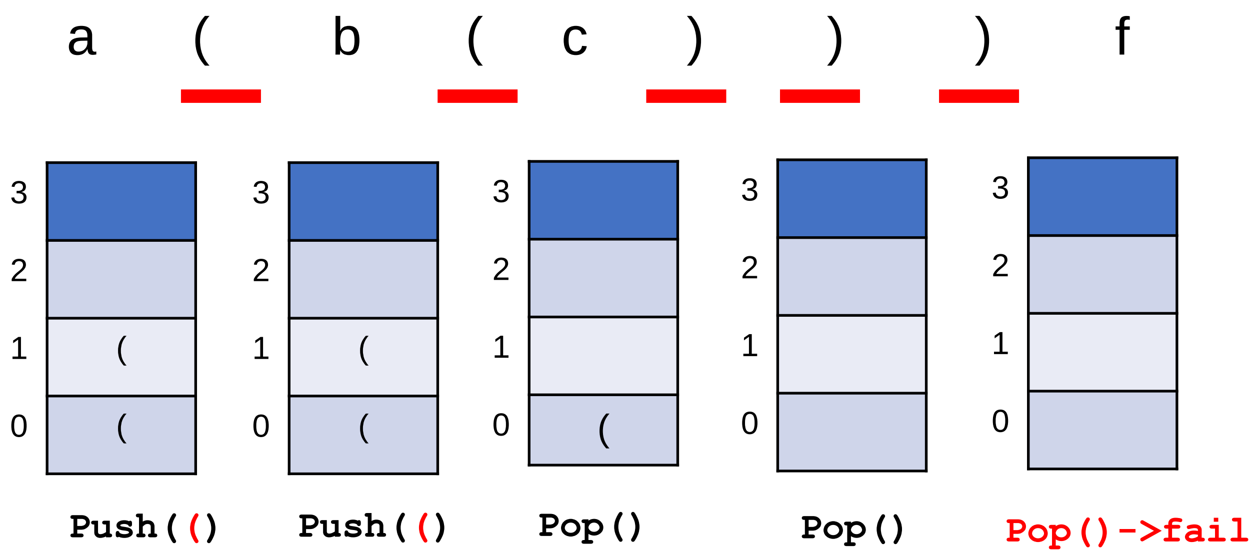
Example for Balance Parentheses



Expression **a(b(c))** have balance parentheses since when end of string is found the stack is empty.

Stack Application Examples

Example for Balance Parentheses



Expression a(b(c))) f does not have balance parentheses => the third) encountered does not has its match, the stack is empty.

Stack Application Examples

Conversion of Infix expression to Postfix expression using Stack data structure

- Infix expressions are hard to parse in a computer program hence it will be difficult to evaluate expressions using infix notation.
- Postfix expressions are used in the computer programs.

A * (B + C)

Symbol	Stack	Postfix
A		A
*	*	A
(	* (	A
B	* (	A B
+	* (+	A B
C	* (+	A B C
)	*	A B C +
	*	A B C + *

Stack Application Examples

A * B ^ C + D

Symbol	Stack	Postfix
A		A
*	*	A
B	*	A B
^	* ^	A B
C	* ^	A B C
+	+	A B C ^ *
D	+	A B C ^ * D
		A B C ^ * D +

3 * 4 + 5

Symbol	Stack	Postfix Expression	Description
3		3	
*	*	3	
4	*	3 4	
+	+	3 4 *	'*' is higher precedence than '+'
5	+	3 4 * 5	
		3 4 * 5 +	

Stack Application Examples

Conversion of Infix expression to Postfix expression using Stack data structure

3 * 4 + 5

Symbol	Stack	Postfix
3		3
*	*	3
4	*	3 4
+	+	3 4 *
5	+	3 4 * 5
		3 4 * 5 +

Stack Application Examples

(A + (B * C – (D / E ^ F) * G) * H)

Symbol	Stack	Postfix
(	(	
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
*	(+ (*	A B
C	(+ (*	A B C
-	(+ (-	A B C *
(	(+ (- (	A B C *
D	(+ (- (	A B C * D
/	(+ (- (/	A B C * D
E	(+ (- (/	A B C * D E
^	(+ (- (/ ^	A B C * D E
F	(+ (- (/ ^	A B C * D E F
)	(+ (-	A B C * D E F ^ /
*	(+ (- *	A B C * D E F ^ /
G	(+ (- *	A B C * D E F ^ / G
)	(+	A B C * D E F ^ / G * -
*	(+ *	A B C * D E F ^ / G * - H
H	(+ *	A B C * D E F ^ / G * - H
)		A B C * D E F ^ / G * - H * +

Evaluate Postfix Expression Using Stack

2 3 1 * + 9 -

Symbol	Stack	Postfix	Description
2	Push	2	
3	Push	2 3	
1	Push	2 3 1	
*	Pop Two Elements & Evaluate	2	3 * 1 = 3
	Push Result (3)	2 3	
+	Pop Two Elements & Evaluate		2 + 3 = 5
	Push Result (5)	5	
9	Push	5 9	
-	Pop Two Elements & Evaluate		5-9=-4
	Push	-4	

A summary of the rules follows:

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Infix, prefix and postfix

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + (b * c)	+ a * b c	a b c * +
(a + b) * c	* + a b c	a b + c *

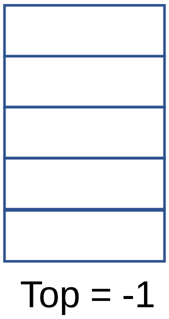
The advantage of using prefix and postfix is that we don't need to use precedence rules, associative rules and parentheses when evaluating an expression.

Activity

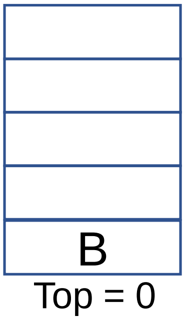
Apply stack implementation using array.

1. stackArray is an array with size of 5. Draw a suitable stack diagram for each statement below:

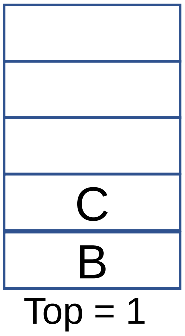
a. createStack;



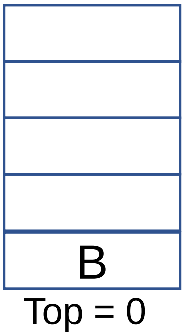
b. push('B');



c. push('C');



d. pop();



Activity

2. "myArray" is an <u>array</u> with a size of 5. Draw a suitable stack diagram for each statement below.

    CreateStack;
    Push ('B');
    Push ('F');
    Pop ( );
    Push ('J');
    Pop ( );
    Push ('M');

3. Converting Infix to Postfix
    a. a + b
    b. a + b * c
    c. a + b * (c - d) / (p - r)

CHAPTER 4

QUEUES

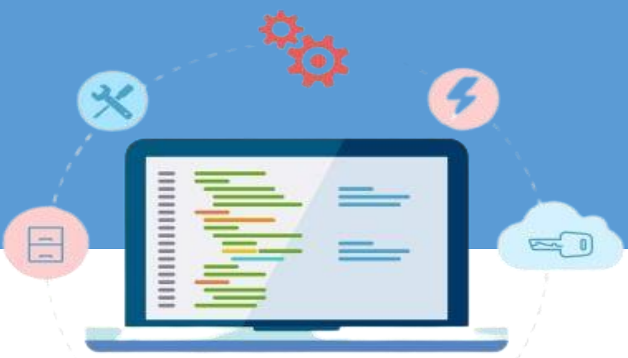
INTRODUCTION TO QUEUE

- New items enter at the back, or rear, of the queue

- Items leave from the front of the queue

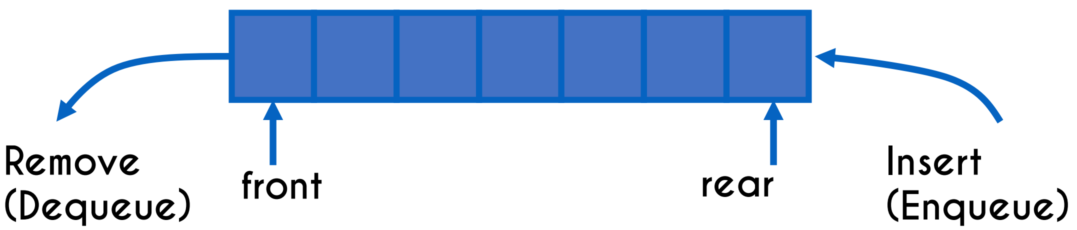
- First-in, first-out (FIFO) property
  - ✓ the first item inserted into a queue is the first item to leave
  - ✓ middle elements are logically inaccessible

- Important in simulation & analyzing the behavior of complex systems



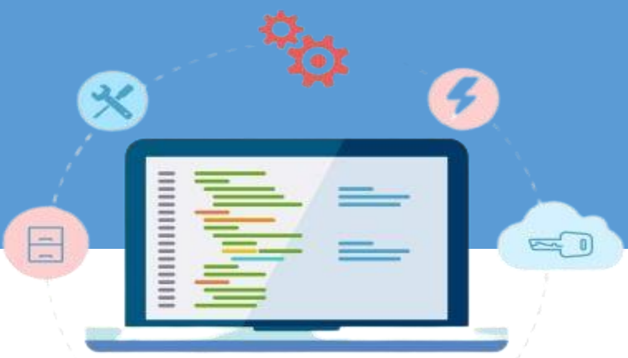
Enqueue and Dequeue

- A queue has a front and a rear.
- Enqueue (Push)
  - ✓ Insert an element at the rear of the queue
- Dequeue (Pop)
  - ✓ Remove an element from the front of the queue



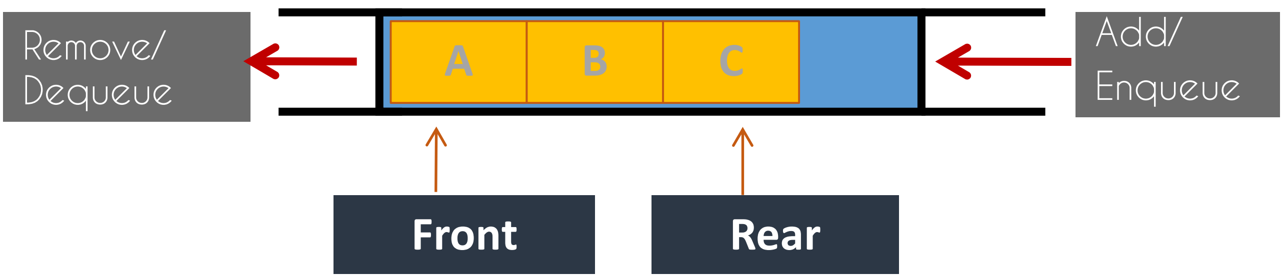
Basic Structure of a Queue:

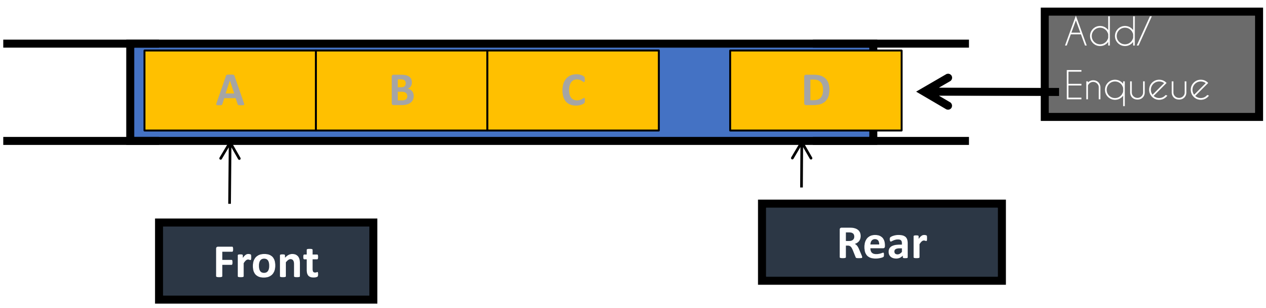
- data structure that hold the queue
- front
- rear



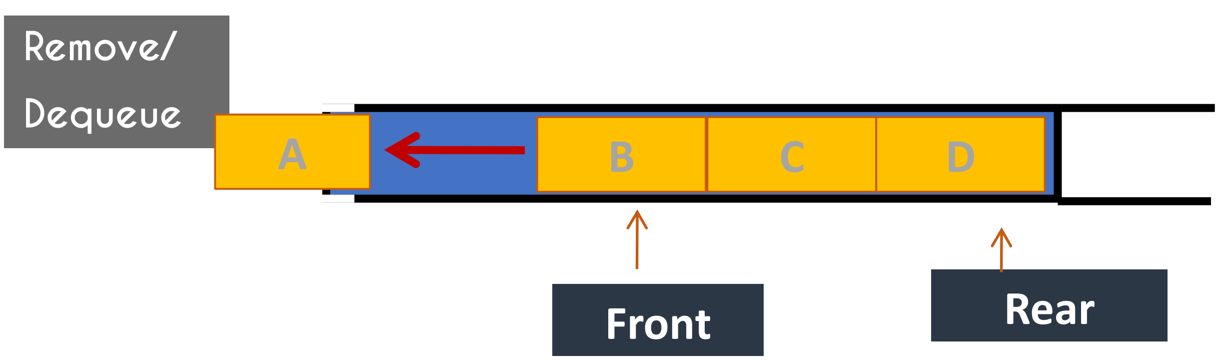
Enqueue and Dequeue

Queue implementation:





Insert D into Queue (enQueue) : D is inserted at rear



Delete from Queue (deQueue) : A is removed

Queue Implementation

Two Types of Queue Implementation:

- Linear implementation (Using Array)
- Circular Array

Queue: Linear Implementation (Using Array)

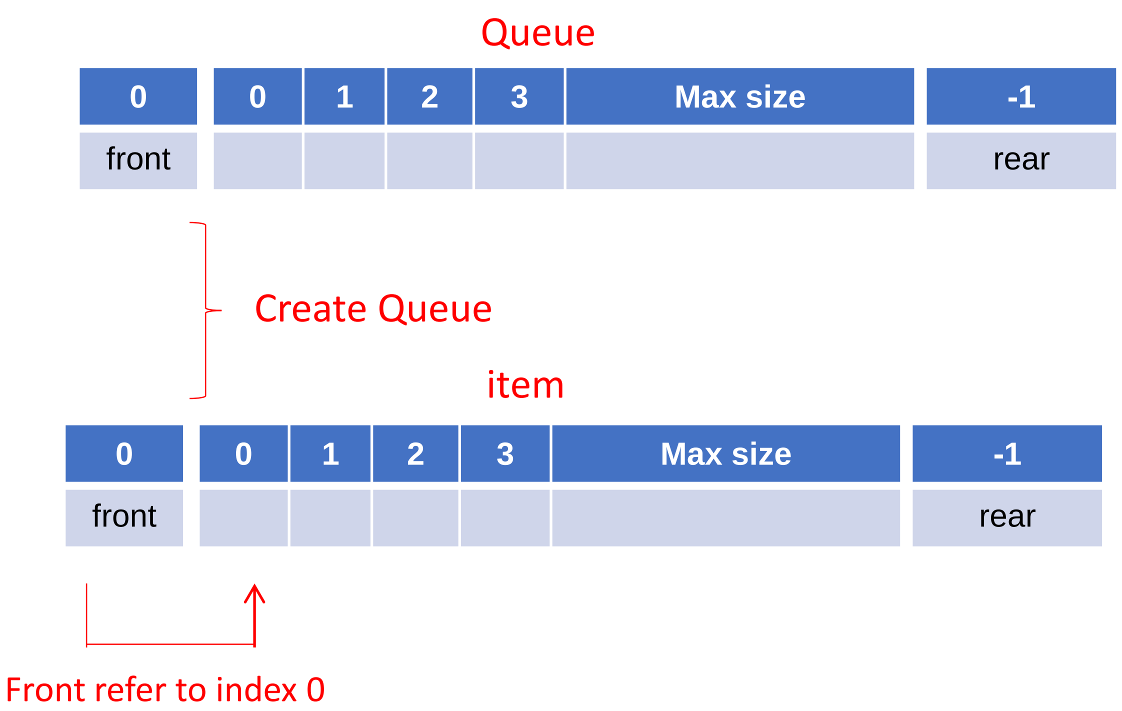
- Number of elements in Queue are fixed during declaration.
- Need **isFull()** operation to determine whether a queue is full or not.

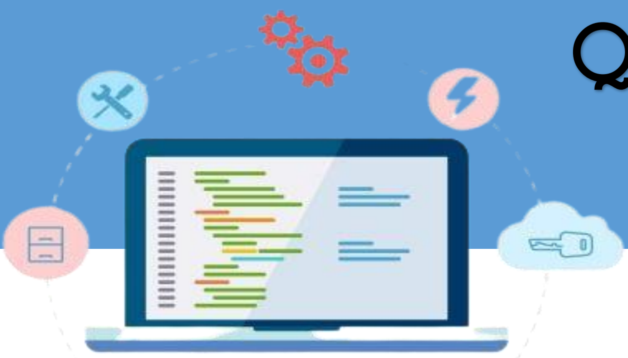
Queue structure need 3 elements:

1. Element to store items in Queue
2. Element to store index at front
3. Element to store index at rear

Create New Queue Operation

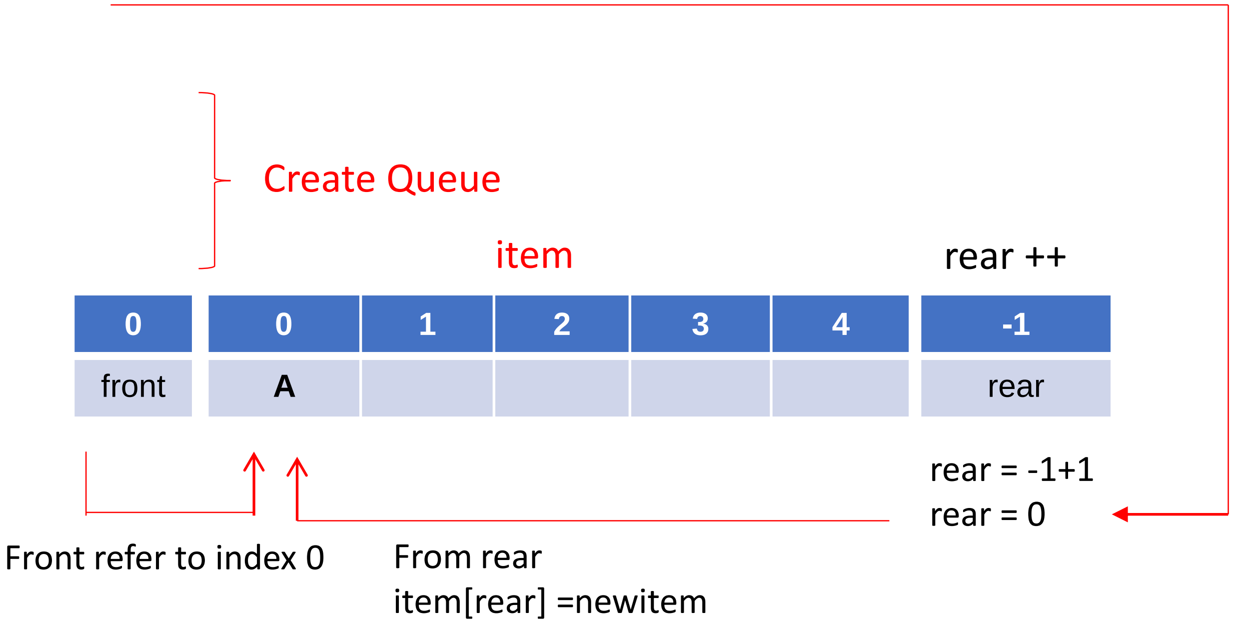
- Declare
  - ✓ front & rear are indexes in the array
  - ✓ Initial condition: front =0 & rear = -1
  - ✓ Size of an array in queue

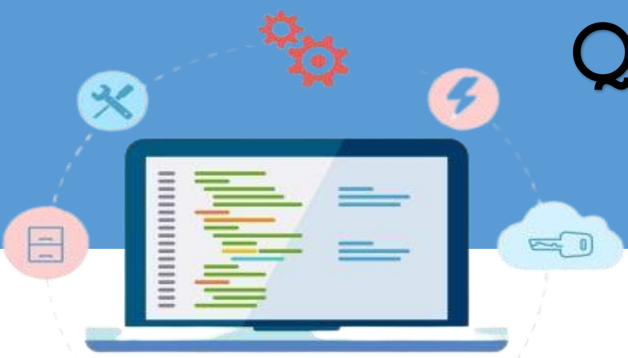




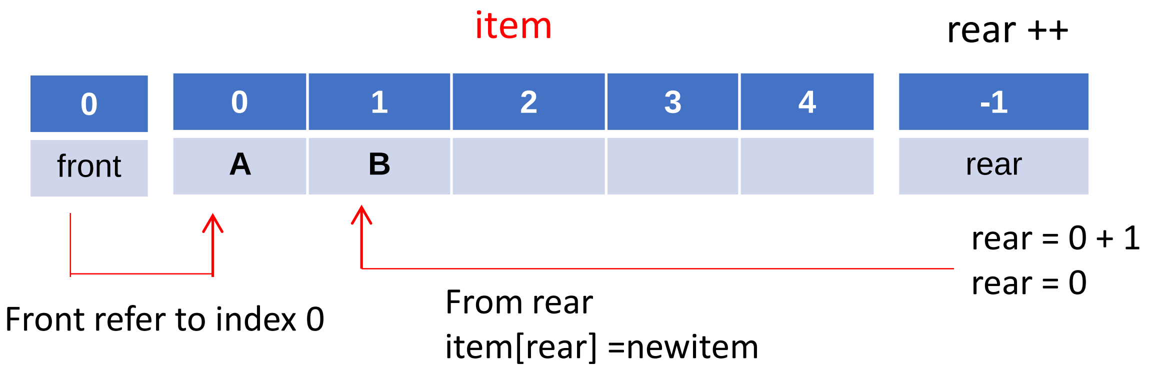
enQueue Operation

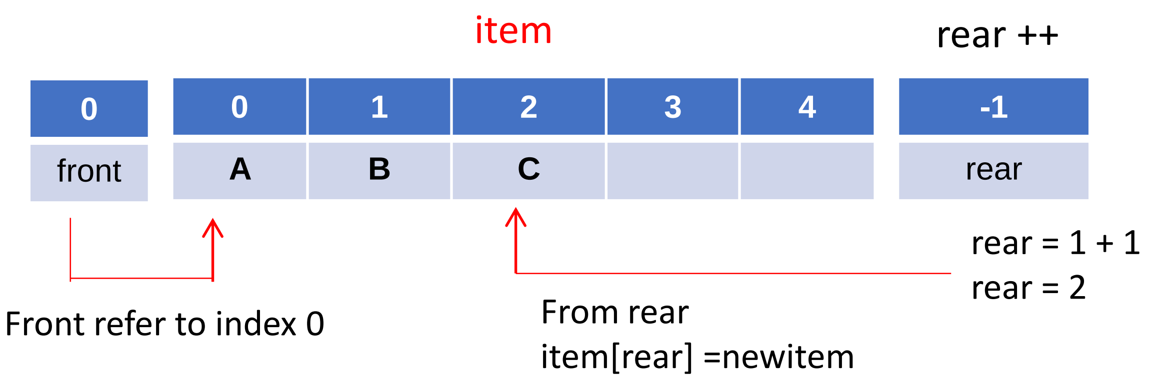
```
void enQueue(){
        cout<<"\n\t##################\n";
        cout<<"\n\t1. enQueue\n";
        //check queue is full
        if(rear == max - 1){
                cout<<"\n\tQueue Is Full, Cannot Add Item In Queue\n";
        }else{
                cout<<"\n\t\tEnter Item:";
                cin>>newitem;
                rear++;
                item[rear]=newitem;
                cout<<endl;
        }
}
```

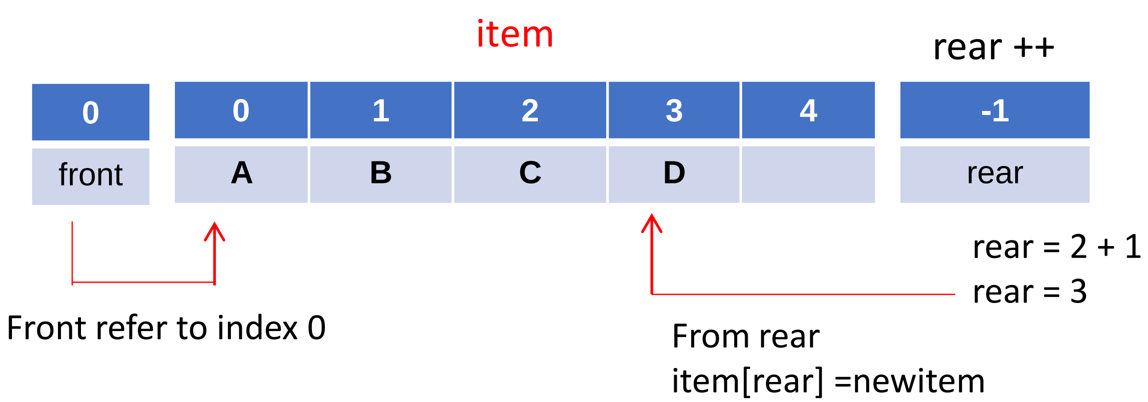


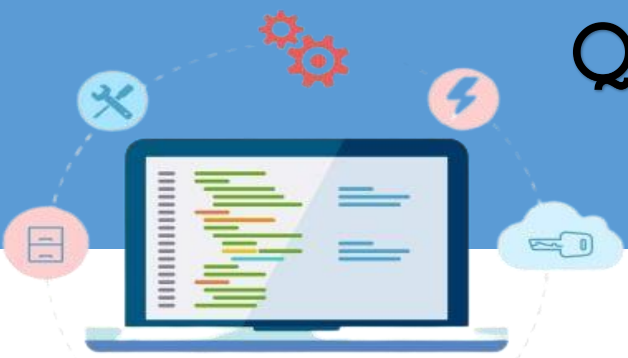


enQueue Operation

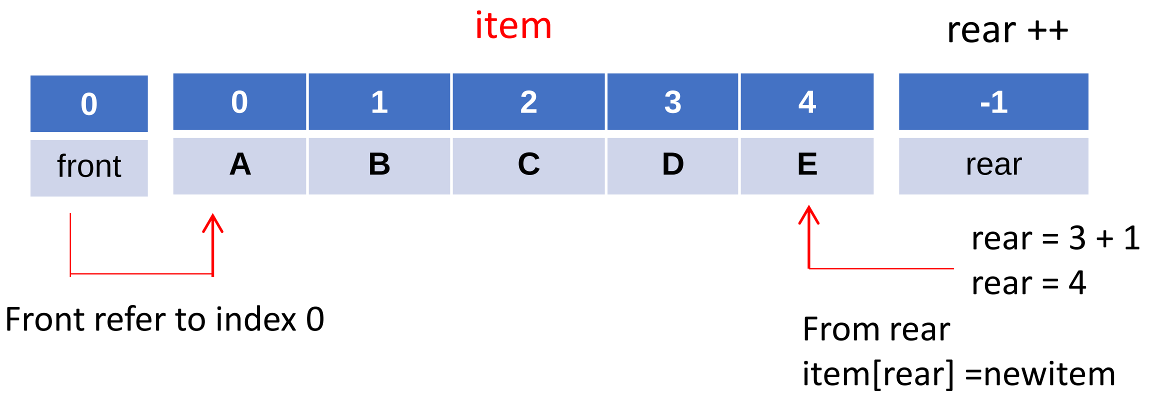


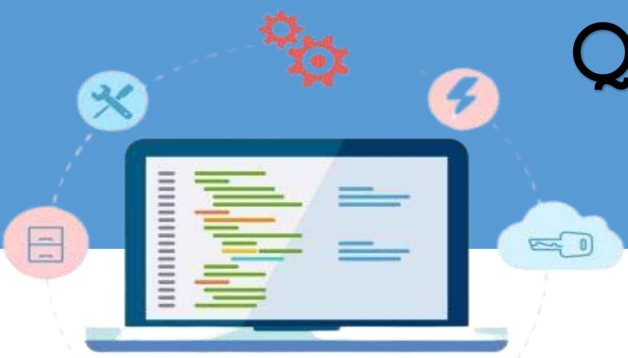






enQueue Operation



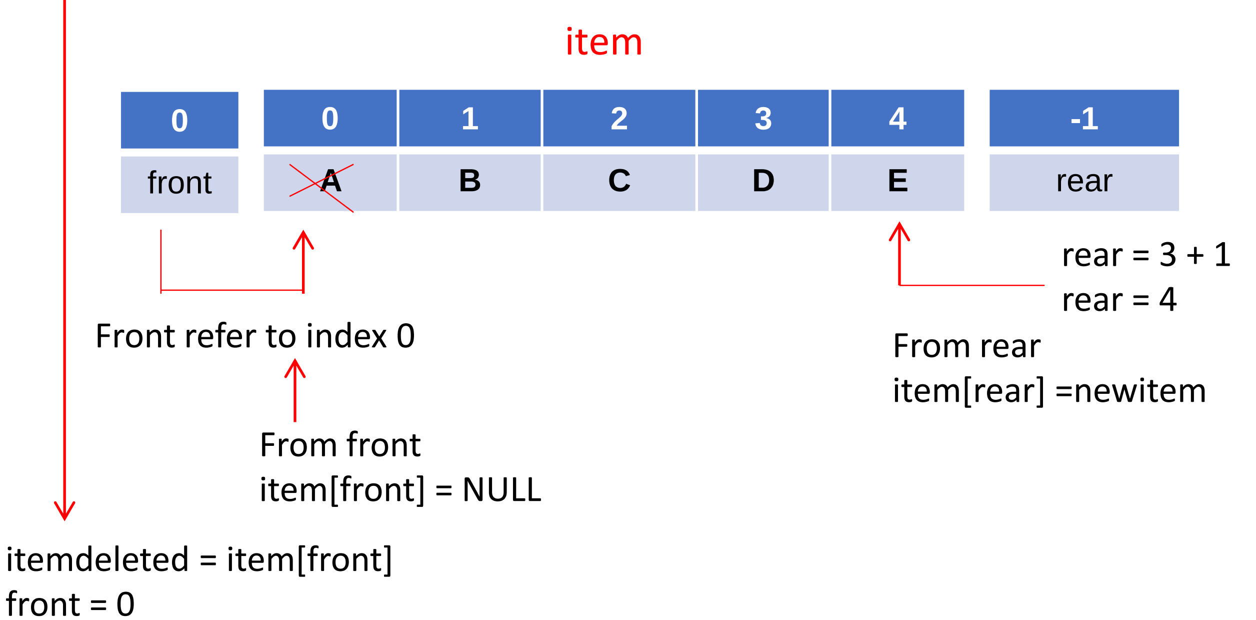


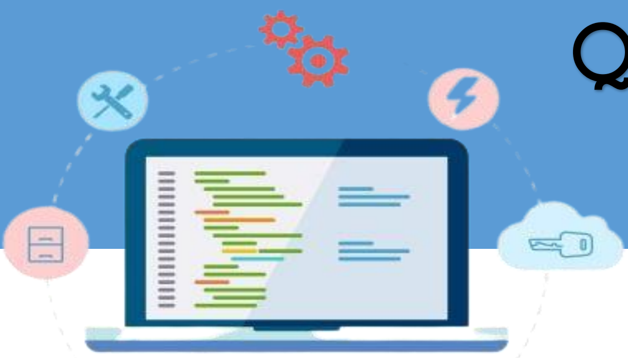
deQueue Operation

```
void deQueue(){
        cout<<"\n\t##################\n";
        cout<<"\n\t2.deQueue\n";
        if(rear < front){
                cout<<"\n\tThere is no data to remove from queue\n";
        }else{
                char itemdeleted;
                itemdeleted=item[front];
                item[front] = NULL;
                cout<<"\n\tItem Remove From Queue:"<<itemdeleted<<endl;
                front++;

        }
        cout<<endl;
}
```

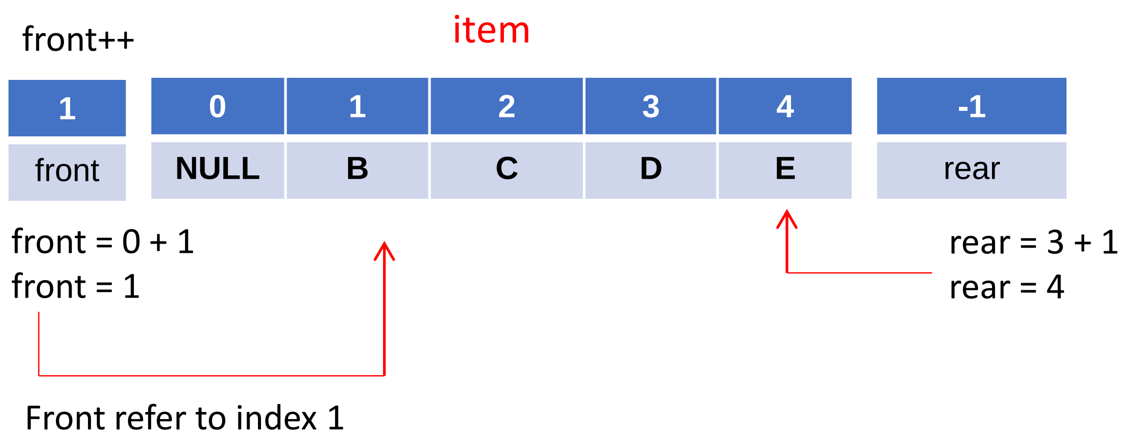
deQueue

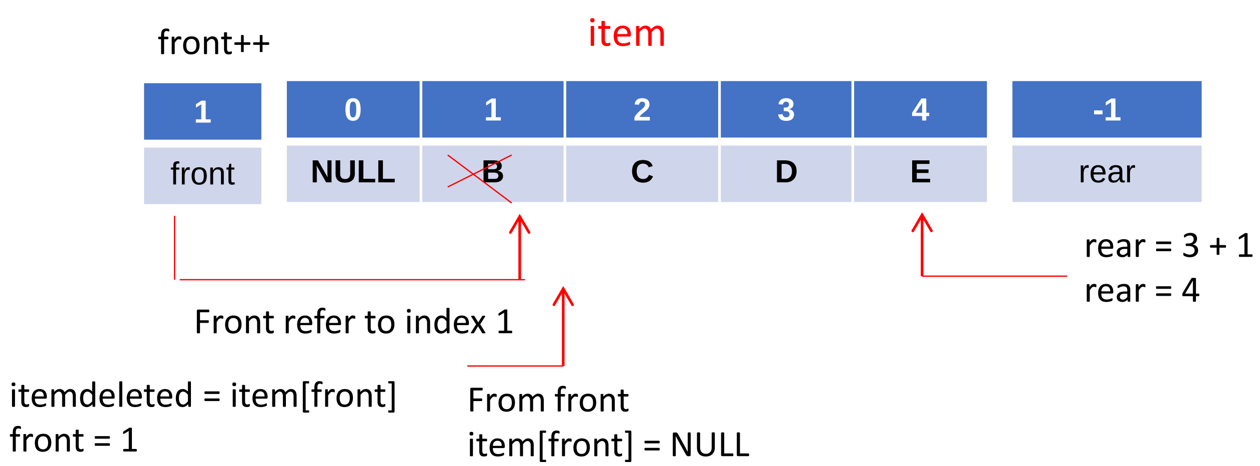


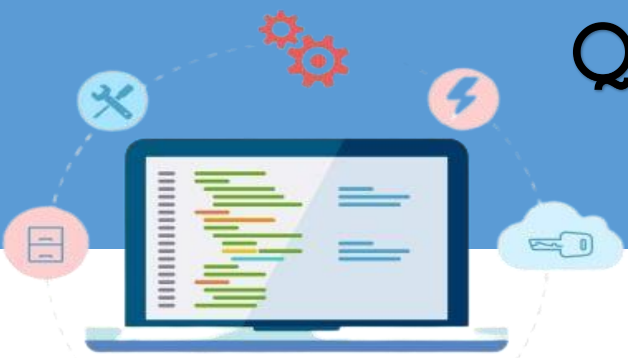


Queue Implementation Using Array

deQueue Operation







deQueue Operation

front++

item

2	0	1	2	3	4	-1
front	NULL	NULL	C	D	E	rear

front = 1 + 1
front = 2

rear = 3 + 1
rear = 4

Front refer to index 2

front++

item

2	0	1	2	3	4	-1
front	NULL	NULL	~~C~~	D	E	rear

rear = 3 + 1
rear = 4

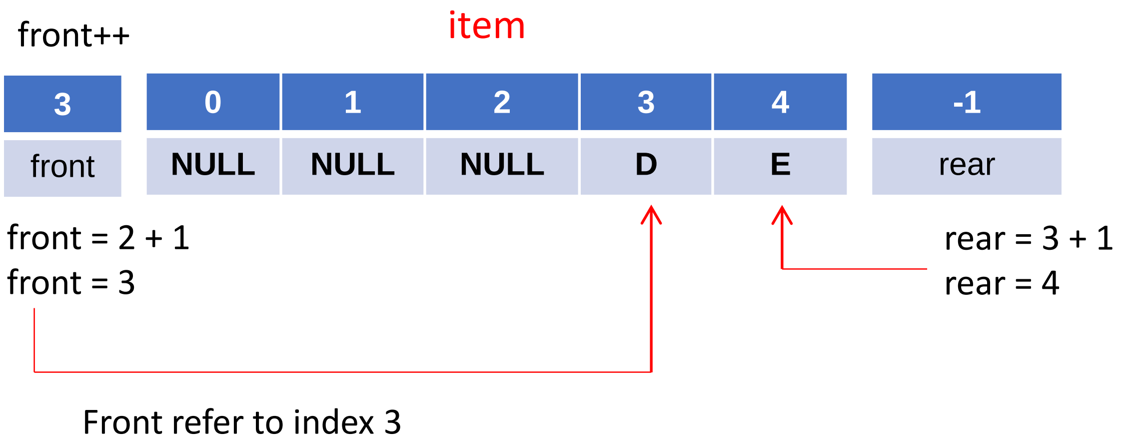
Front refer to index 2

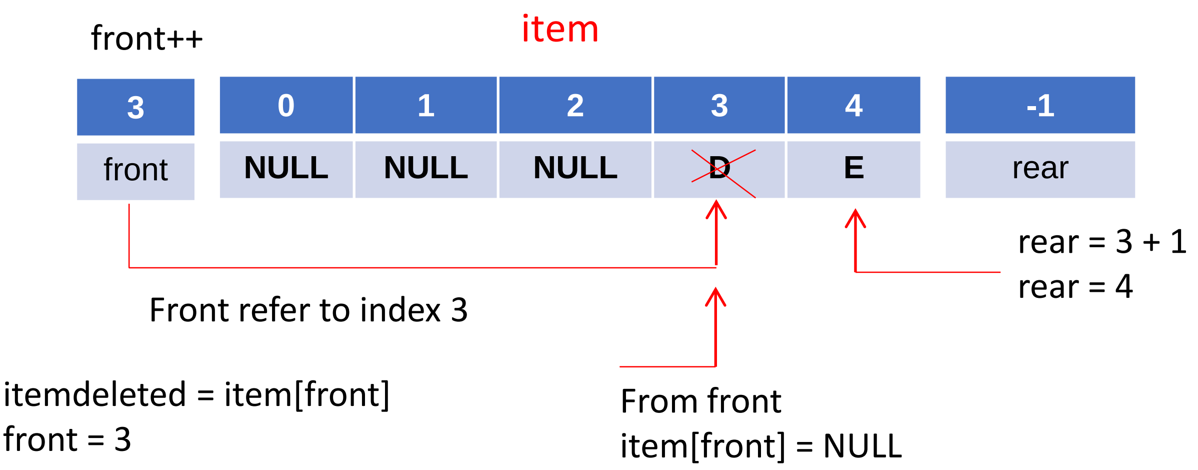
itemdeleted = item[front]
front = 2

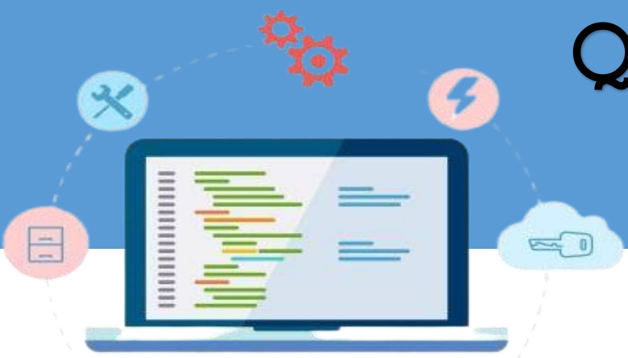
From front
item[front] = NULL

Queue Implementation Using Array

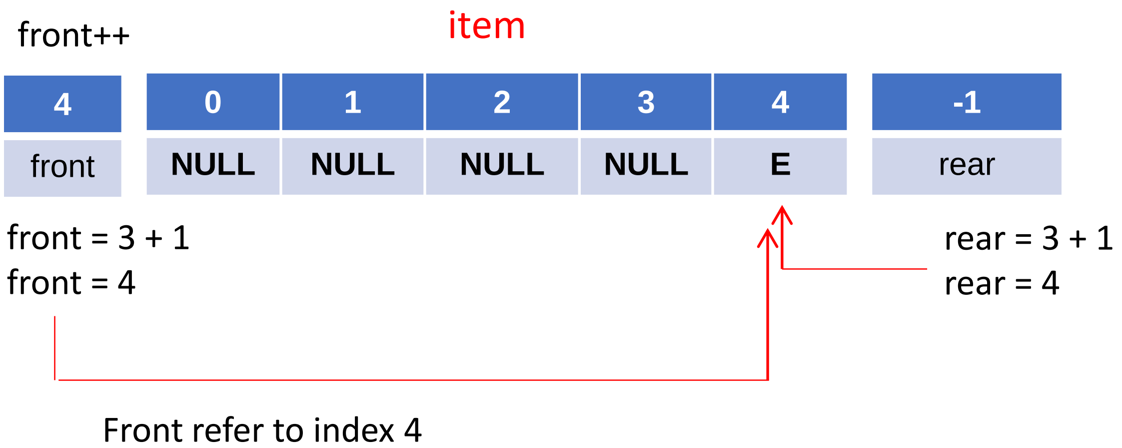
deQueue Operation

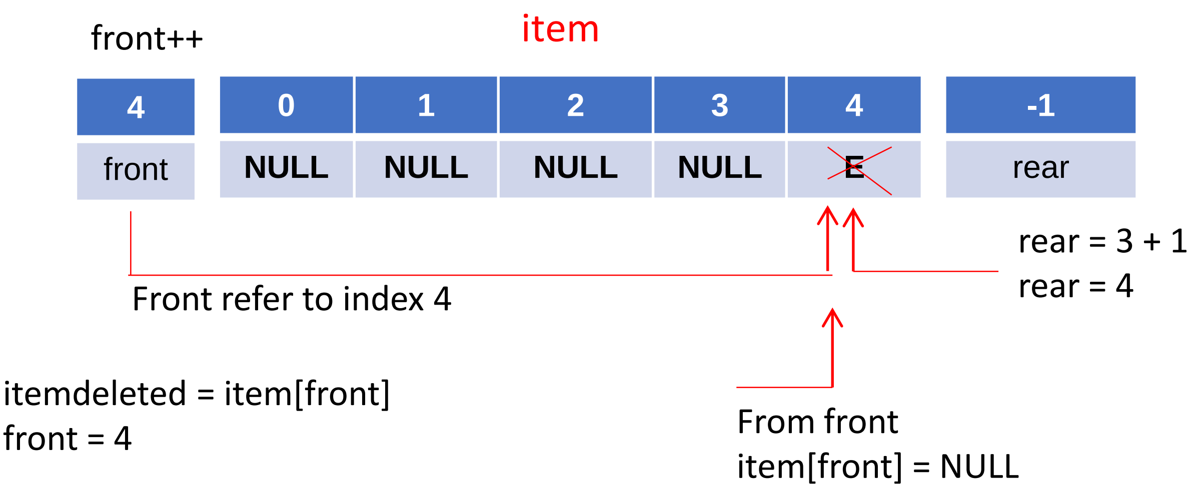






deQueue Operation

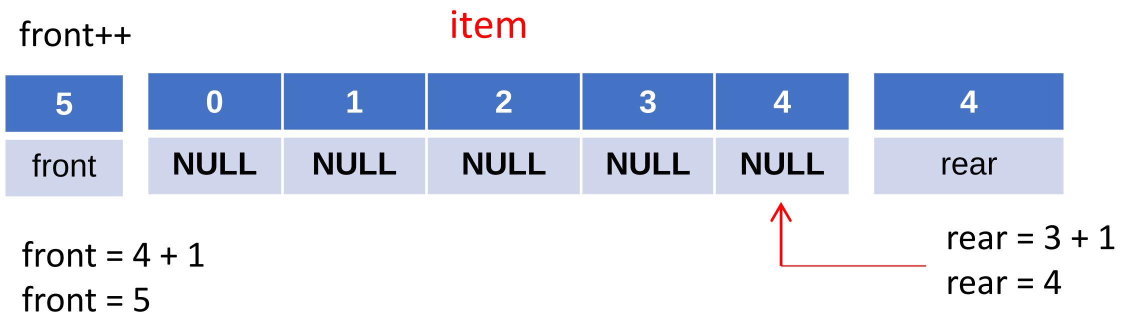




Queue Implementation Using Array

Queue: Linear Implementation (Using Array)

- Problem: Rightward-Drifting:
  - ✓ After a sequence of additions & removals, items will drift towards the end of the array
  - ✓ enQueue operation cannot be performed on the queue below, since rear = max – 1



- Rightward drifting solutions
  - ✓ Shift array elements after each deletion
  - ✓ Shifting dominates the cost of the implementation

Queue : Circular Array

- Use a circular array: When Front or rear reach the end of the array, wrap them around to the beginning of the array

- Problem:
  - ✓ Front & rear can't be used to distinguish between queue-full & queue-empty conditions

Solution

- Use a counter
- Count == 0 means empty queue
- Count == MAX_QUEUE means full queue

Queue : Circular Array

- Number of elements in Queue are fixed during declaration.
- Need **isFull()** operation to determine whether a queue is full or not.

Queue structure need 4 elements

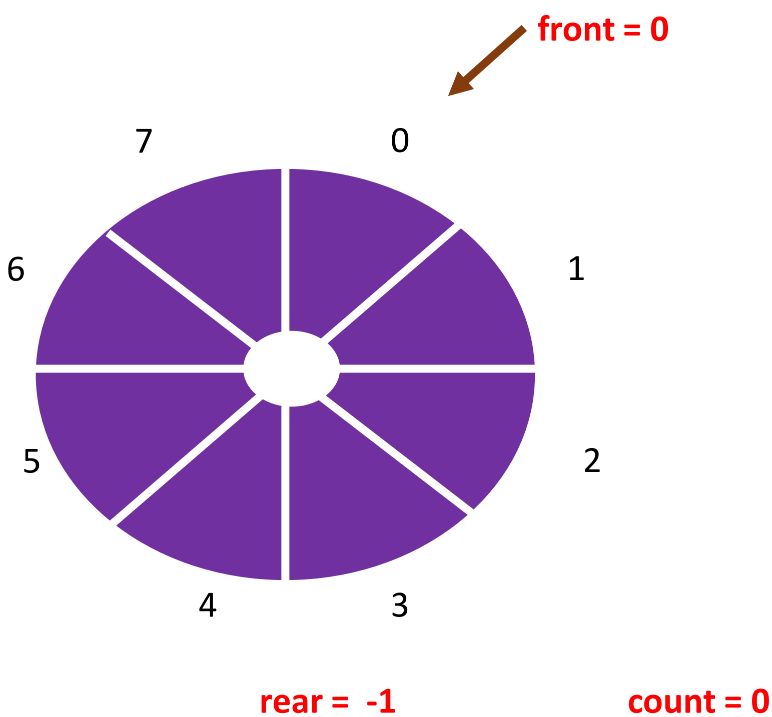
1. Element to store items in Queue
2. Element to store in Queue
3. Element to store index at rear
4. Element to store index in counter

Create Queue Operation

- Declare
  - ✓ front & rear are indexes in the array
  - ✓ count to store index
  - ✓ Initial condition: front =0 , rear = -1, count = 0
  - ✓ Size of an array in queue

Queue: Circular Array

- The Wrap-around effect is obtained by using modulo arithmetic (%-operator)



Queue: Circular Array

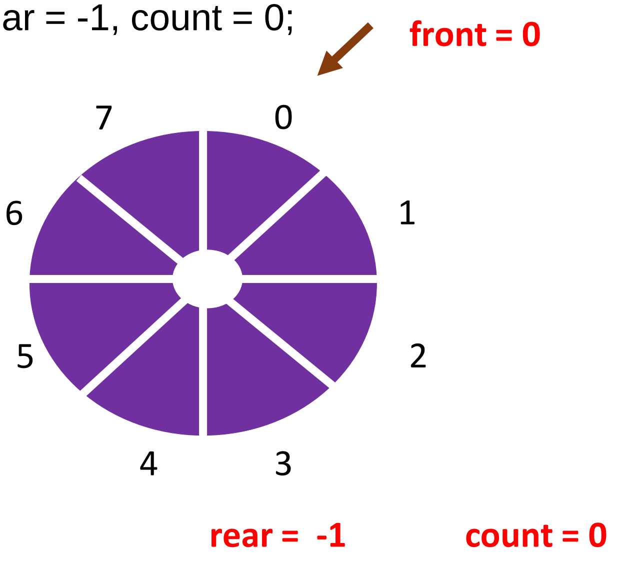
- enQueue
  - ✓ Increment rear, using modulo arithmetic
  - ✓ Insert item
  - ✓ Increment count

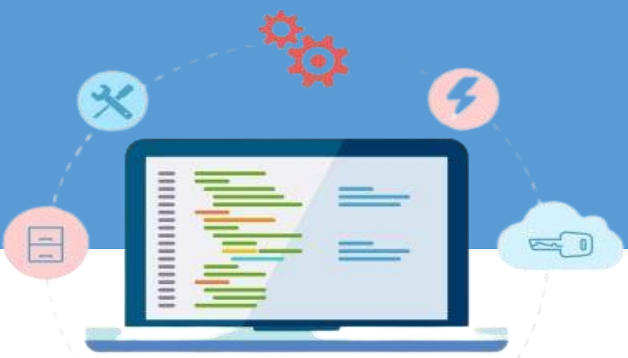
- deQueue
  - ✓ Increment front using modulo arithmetic
  - ✓ Decrement count

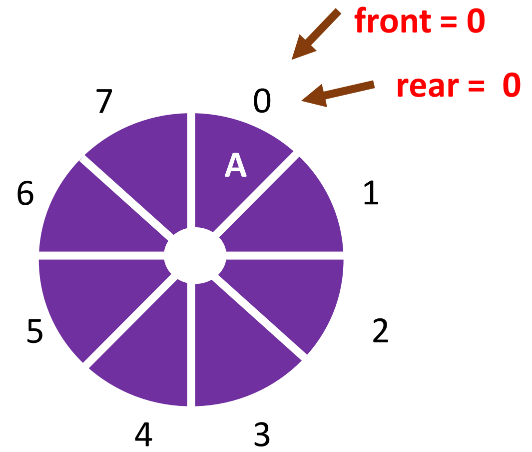
- Disadvantage
  - ✓ Overhead of maintaining a counter

Example Code 2:

```
#include <iostream>
using namespace std;
#define max 8
char queue[max], newitem;
int front = 0, rear = -1, count = 0;
```







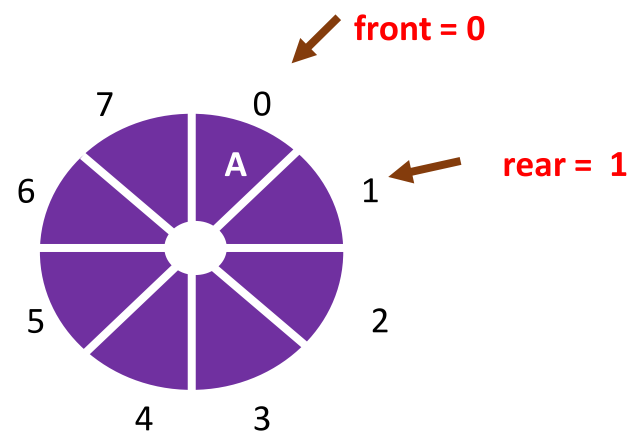
rear = (-1 + 1) % 8
rear = 0 % 8
rear = 0
queue[0] = A

count = 0 + 1
count = 1

0
8√ 0
0
0

count = 1

From previous slide: front = 0, rear = 0, count = 1



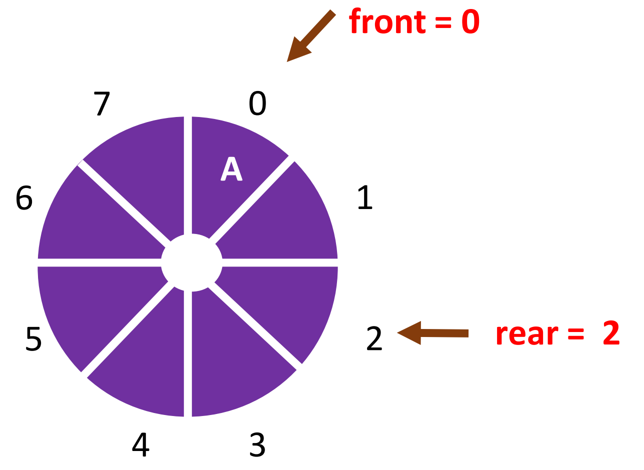
rear = (0 + 1) % 8
rear = 1 % 8
rear = 1
queue[1] = B

count = 1 + 1
count = 2

0
8√ 1
0
1

count = 2

From previous slide: front = 0, rear = 1, count = 2



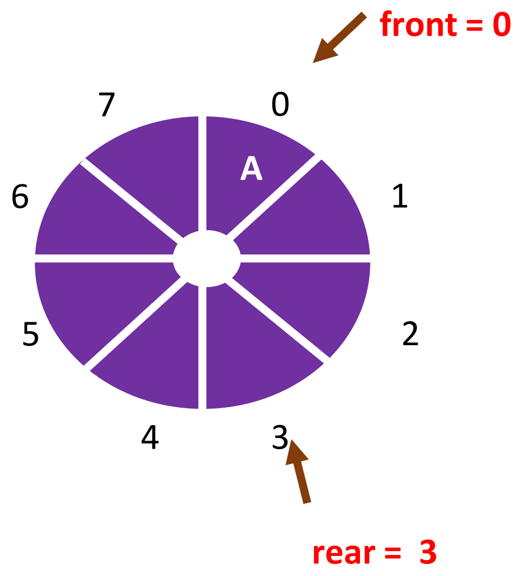
rear = (1 + 1) % 8
rear = 2 % 8
rear = 2
queue[2] = C

count = 2 + 1
count = 3

0
8√ 2
0
2

count = 3

From previous slide: front = 0, rear = 2, count = 3

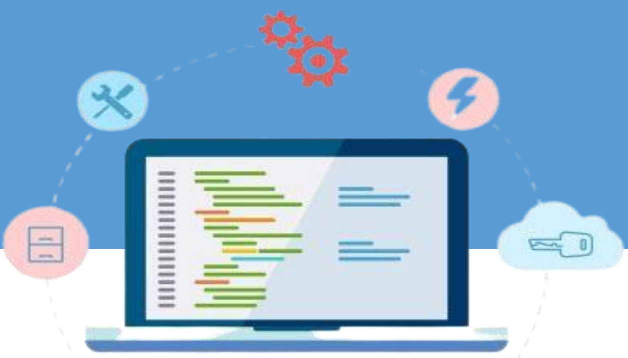


rear = (2 + 1) % 8
rear = 3 % 8
rear = 3
queue[3] = D

count = 3 + 1
count = 4

0
8√ 3
0
3

count = 4



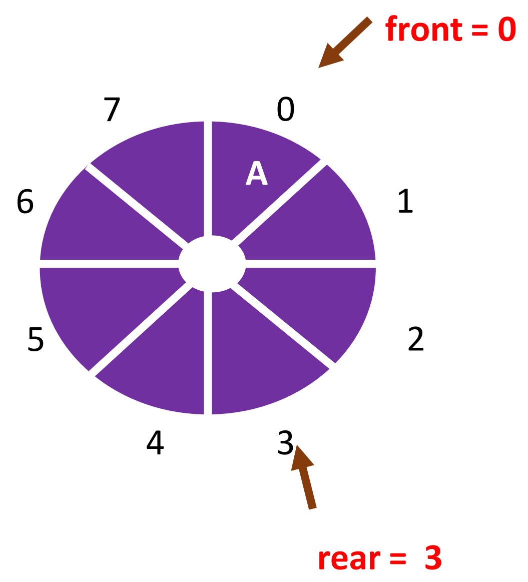
From previous slide: front = 0, rear = 2, count = 3

rear = (2 + 1) % 8
rear = 3 % 8
rear = 3
queue[3] = D

count = 3 + 1
count = 4

0
8√ 3
0
3

count = 4



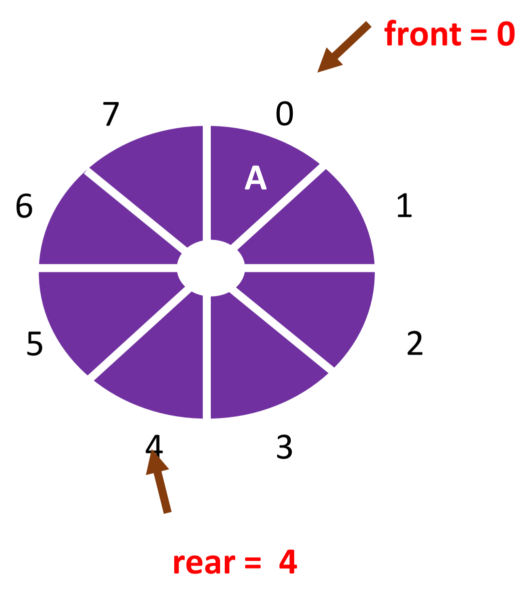
From previous slide: front = 0, rear = 3, count = 4

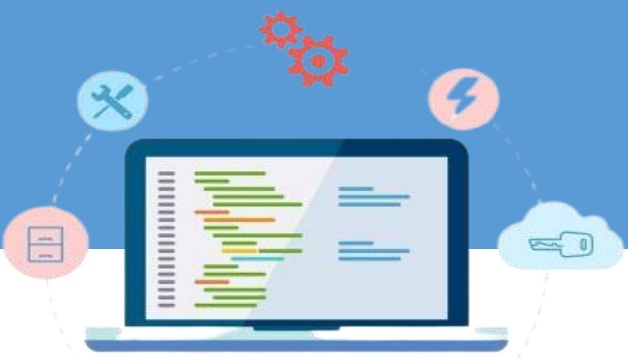
rear = (3 + 1) % 8
rear = 4 % 8
rear = 4
queue[4] = E

count = 4 + 1
count = 5

0
8√ 4
0
4

count = 5



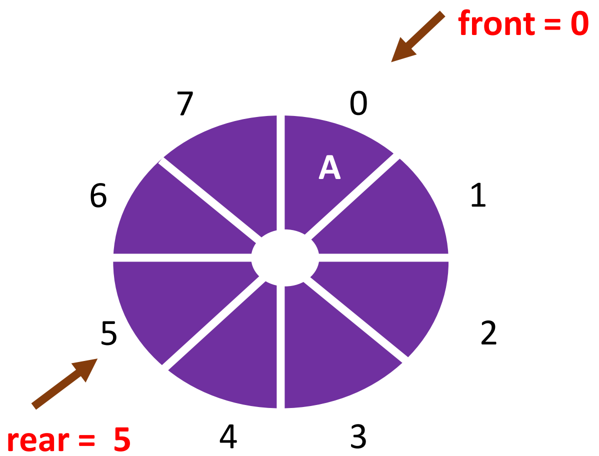


From previous slide: front = 0, rear = 4, count = 5

rear = (4 + 1) % 8
rear = 5 % 8
rear = 5
queue[5] = F

count = 5 + 1
count = 6

0
8√ 5
0
5



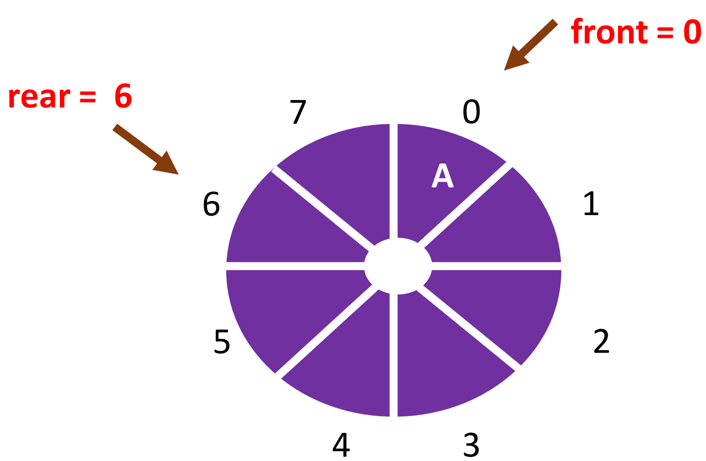
count = 6

From previous slide: front = 0, rear = 5, count = 6

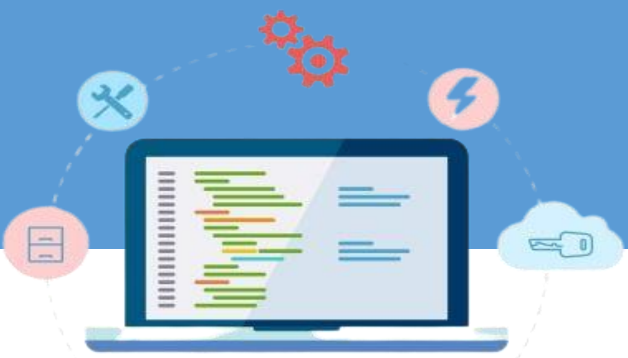
rear = (5 + 1) % 8
rear = 6 % 8
rear = 6
queue[6] = G

count = 6 + 1
count = 7

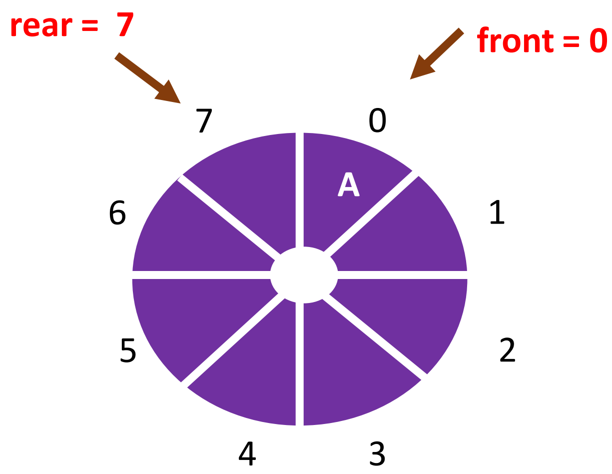
0
8√ 6
0
6



count = 7



From previous slide: front = 0, rear = 6, count = 7

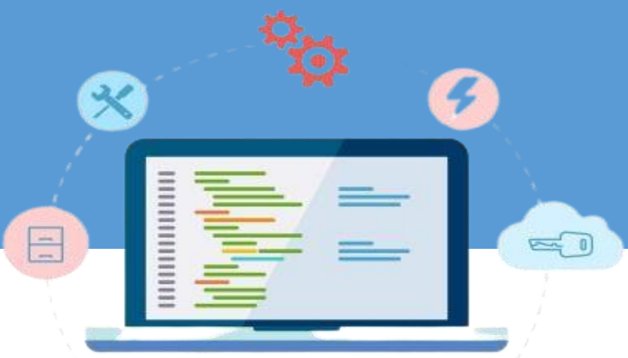


rear = (6 + 1) % 8
rear = 7 % 8
rear = 7
queue[7] = H

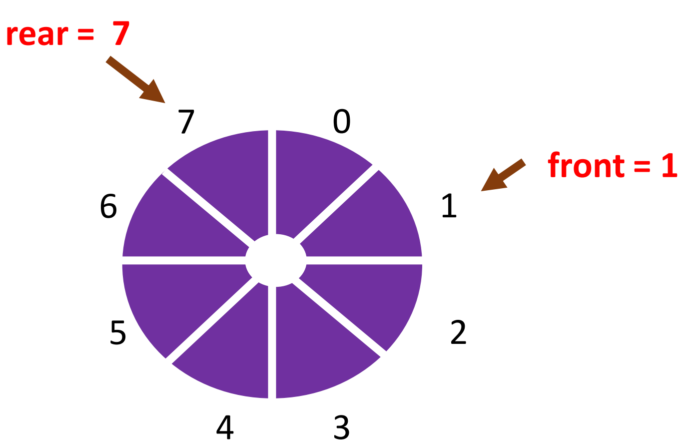
```
   0
8√ 7
   0
   7
```

count = 7 + 1
count = 8

count = 8



```
void deQueue(){
cout<<"\n\t#### deQueue Circular
####\n";
          if(count == 0){
                    cout<<"\n\tQueue Circular Is
Empty, No Data To Be Deleted!!!\n";
          }else{
                    queue[front] = NULL;
                    front=(front + 1) % max;
                    count--;
          }
}
```

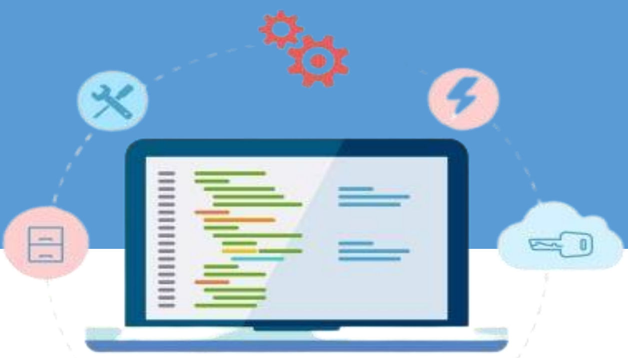


queue[0] = NULL
front = (0 + 1) % 8
front = 1 % 8
front = 1

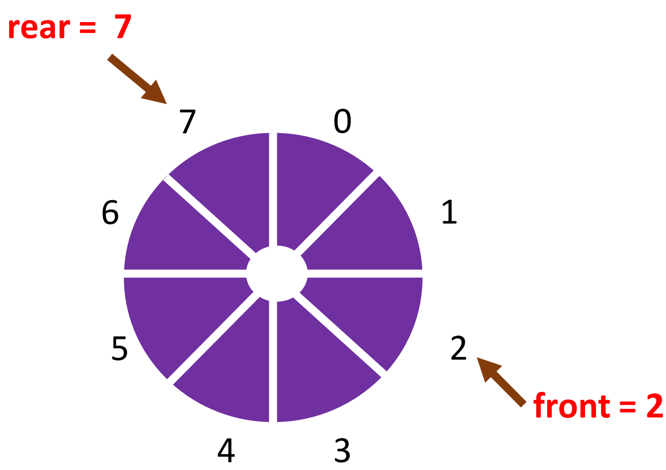
0
8√ 1
0
1

count = 8 - 1
count = 7

count = 7



From previous slide: front = 1, rear = 7 , count = 7



queue[1] = NULL
front = (1 + 1) % 8
front = 2% 8
front = 2

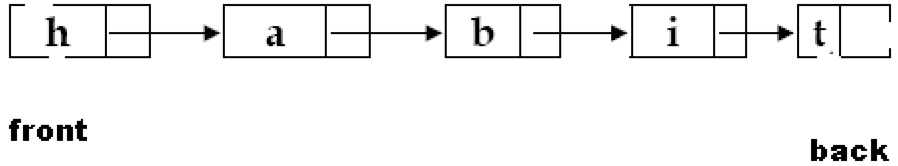
0
8√ 2
0
2

count = 7 - 1
count = 6

count = 6

Queue Implementation Using Linked List

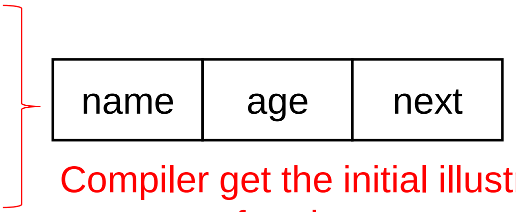
- Pointer-Based Implementation
  - ✓ More straightforward than array-based
  - ✓ Need Two external pointer (Front & rear) which front to trace deQueue operation and rear to trace enQueue operation.



Create Queue Implementation Using Linked List

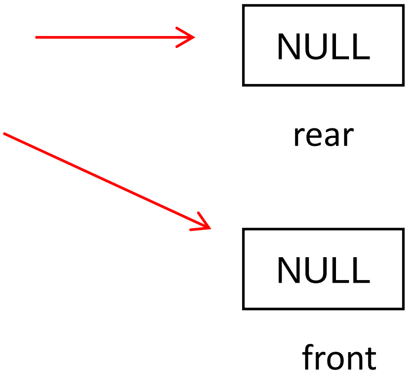
Example Code 1:

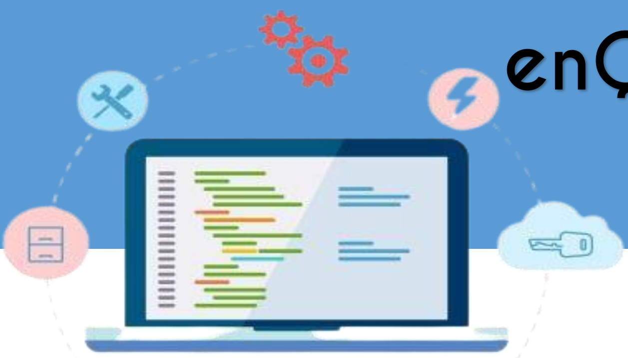
```
#include <iostream>
using namespace std;
struct nodeQueue{
        char name;
        int age;
        nodeQueue *next;
};
```



Compiler get the initial illustrated structure of node

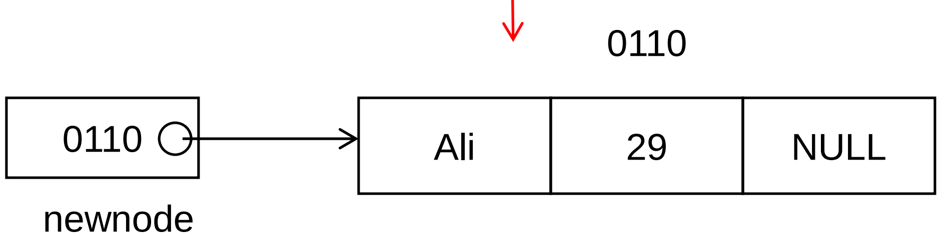
```
nodeQueue *rear = NULL;
nodeQueue *front=NULL;
```

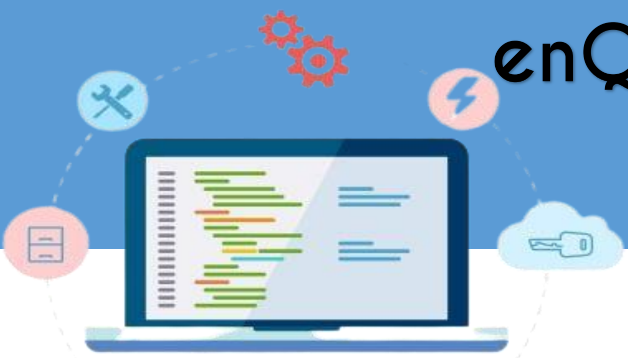




enQueue Implementation Using Linked List

```
void enQueue(){
//create new node
	nodeQueue *newnode;
	newnode = new nodeQueue;
cout<<"\n\t####enQueue####\n";
//assign data field for name and age
	cout<<"Enter Name:";
	cin>>newnode->name;
	cout<<"Enter Age:";
	cin>>newnode->age;
	newnode->next = NULL;
```

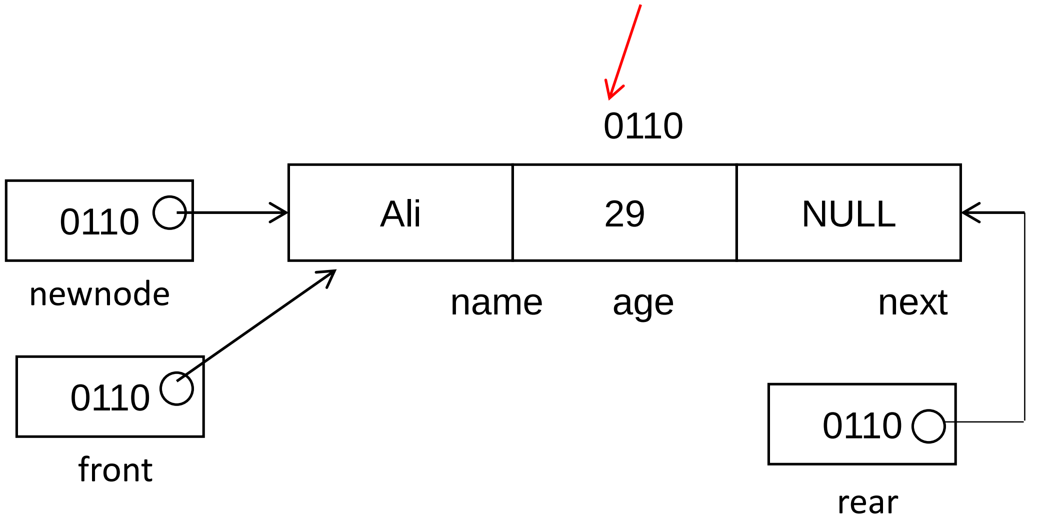




enQueue Implementation Using Linked List

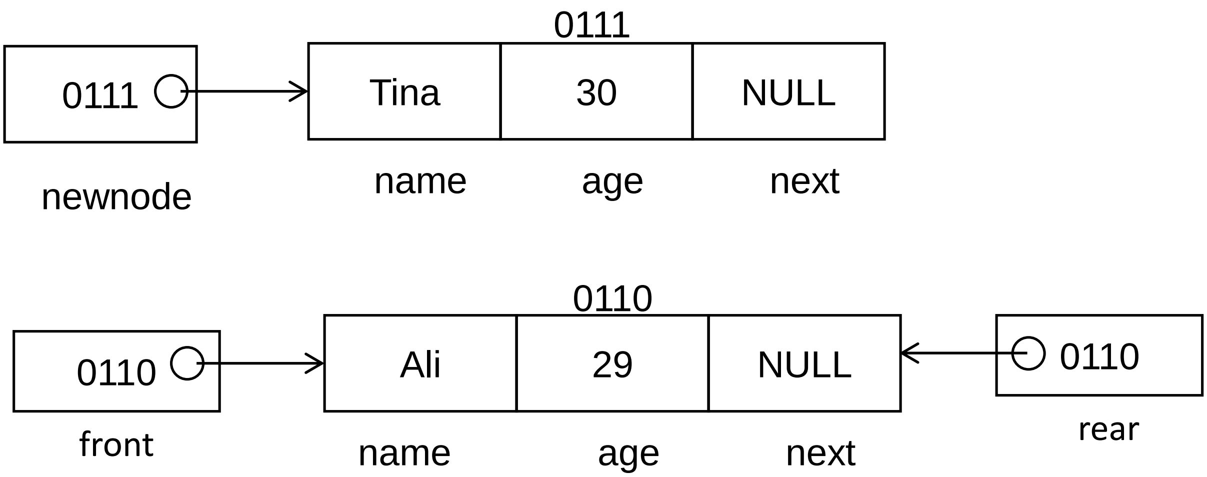
```
//insert newnode into queue
//check whether queue is empty
if((front == NULL) && (rear ==
NULL)){
        front = newnode;
        rear = newnode;
}else{
        rear->next = newnode;
        rear = newnode;
}
```

Insertion to an empty queue



enQueue Implementation Using Linked List

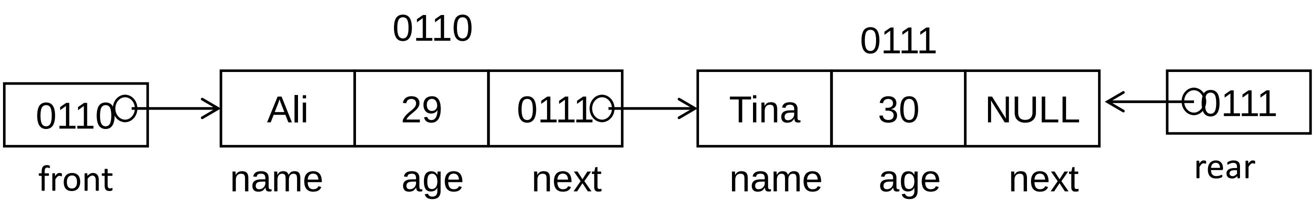
Insertion to a non empty queue




```
rear->next = newnode;
rear=newnode;
```

enQueue Implementation Using Linked List

Insertion to a non empty queue



enQueue Implementation Using Linked List

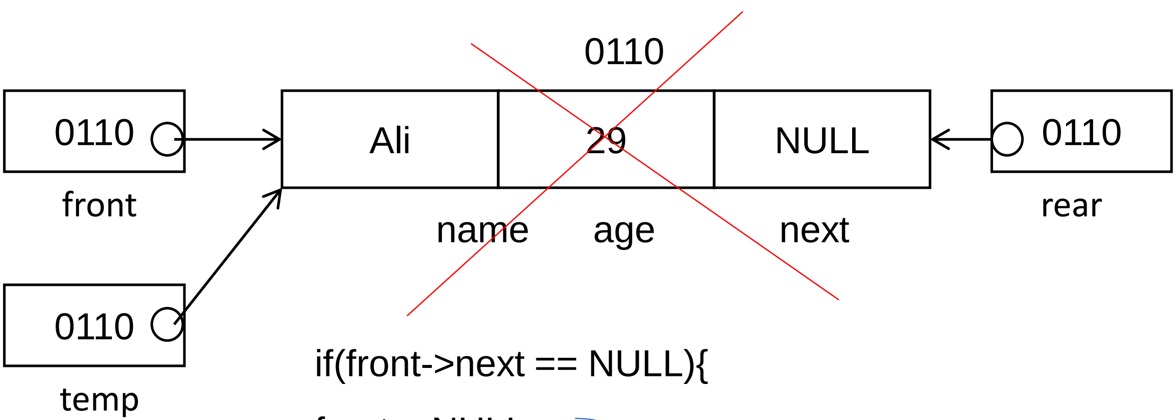
```
void deQueue(){
        cout<<"\n\t####deQueue####\n";
        //check whether queue is empty
        if((front == NULL) && (rear == NULL)){
                cout<<"\n\tQueue Is Empty!!!\n";
        }else{
                nodeQueue *temp;
                temp = front;
                if(front->next == NULL){
                        front = NULL;
                        rear = NULL;
                        delete temp;
                }else{
                        front = front->next;
                        delete temp; } } }
```

If the queue contains one item only

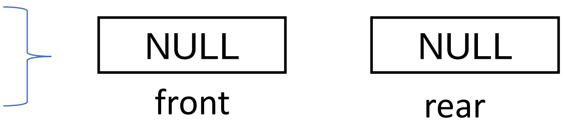
enQueue Implementation Using Linked List

If the queue contains one item only to be deleted

```
nodeQueue *temp;
temp = front;
```



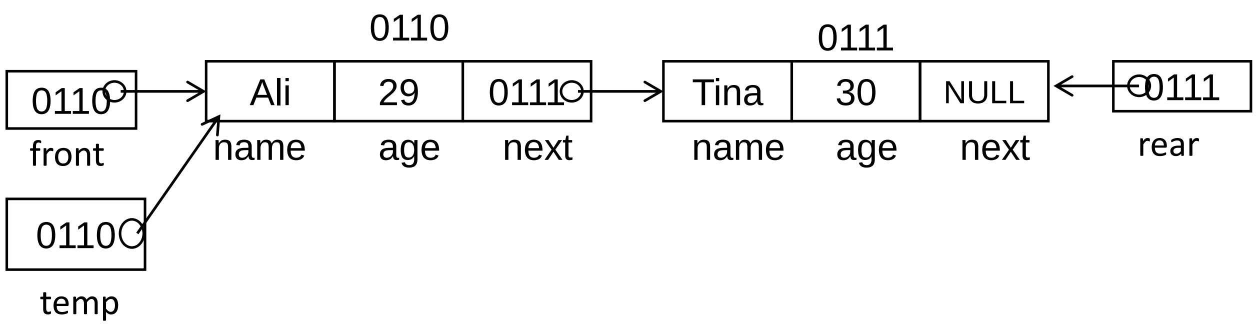

```
if(front->next == NULL){
front = NULL;
rear = NULL;
delete temp;
}else{
…}
```



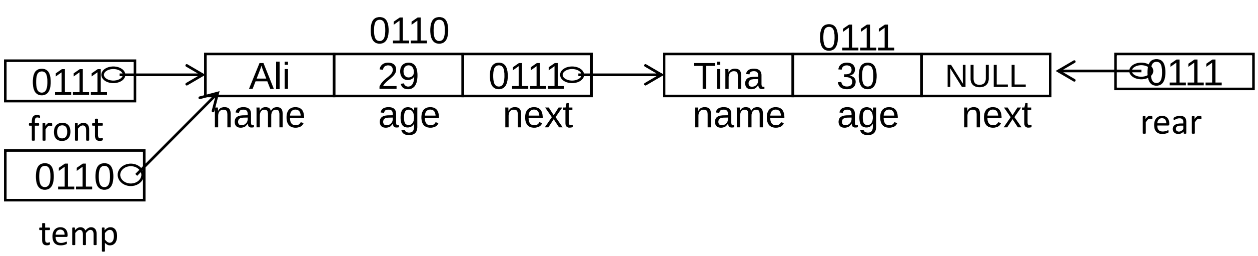
enQueue Implementation Using Linked List

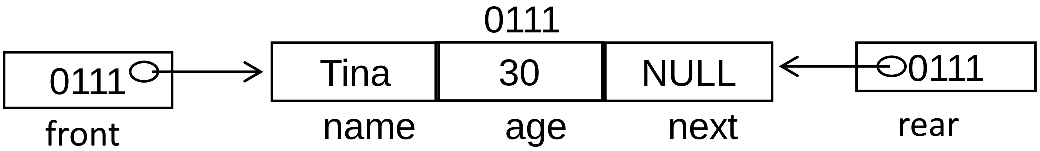
If the queue contains more than one item

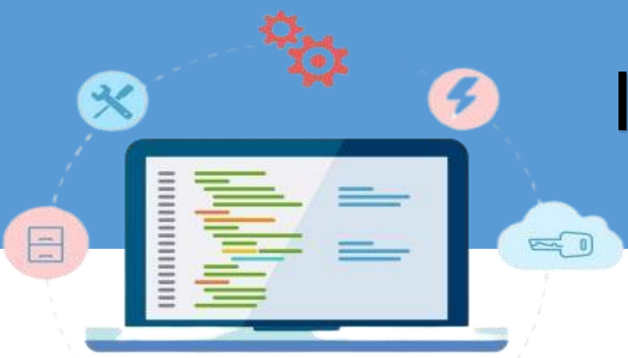
```
nodeQueue *temp;
temp = front;
```




```
…}else{
front = front->next;
delete temp; }
```

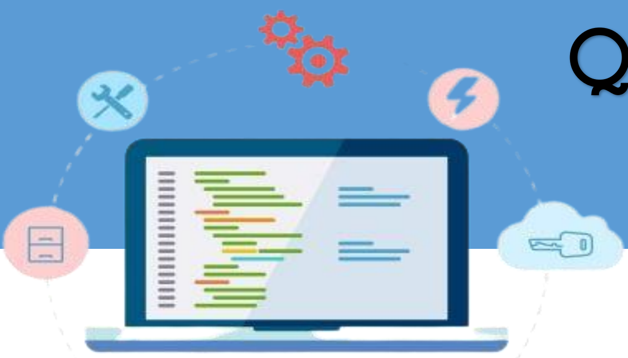






Display Queue Implementation Using Linked List

```
void displayQueue(){
cout<<"\n\t####Display Queue####\n";
if((front == NULL) && (rear == NULL)){
        cout<<"\n\tQueue Is Empty!!!\n";
        cout<<"\n\tfront :"<<front<<"\trear :"<<rear<<endl;
}else{
        nodeQueue *cursor;
        cursor=front;
        cout<<"\n\tThe Elements In Queue Are\n";
        cout<<"\n\tfront :"<<front<<"\trear :"<<rear<<endl;
        int node=1;
        while(cursor){
        cout<<"\n\tNode :"<<node++<<"\tName :"<<cursor->name<<"\tAge
:"<<cursor- >age<<"\tcursor-next:"<<cursor->next<<endl;
        cursor=cursor->next; }  }
```




```
int main()
{
int selection;
menu:
    cout<<"\n\nMenu Selection\n";
    cout<<"\n1\tenQueue\n";
    cout<<"\n2\tdeQueue\n";
    cout<<"\n3\tDisplay Queue\n";
    cout<<"\n\tSelection is:";
    cin>>selection;
        switch(selection){
            case 1:     enQueue();
                displayQueue();
                goto menu;
                break;
            case 2:     deQueue();
                displayQueue();
                goto menu;
                break;
            case 3:     displayQueue();
                goto menu;
                break;
            default:cout<<"\n\tWrong Selection\n";    }
            return 0;
                }
```

Activity

1. A Queue Linear Array name as Q stores int values. Draw a Queue Linear Array to show what Q will look like after each of the following operations is executed. Set the size of an array is 7, a rear=-1 and front=0 before the following operations start. State the changes of rear and front after each of the operation is executed.
   - i. enqueue(Q, 6);
   - ii. enqueue(Q, 12);
   - iii. enqueue(Q, 13);
   - iv. dequeue( );
   - v. dequeue( );
   - vi. enqueue(Q, 19);
   - vii. enqueue(Q, 21);
   - viii. enqueue(Q, 22);
   - ix. dequeue( );
   - x. enqueue(Q, 20);

Q

0	1	2	3	4	5	6
front						

rear=-1

Activity

2. Draw the Circular Queue according to the segment code below:

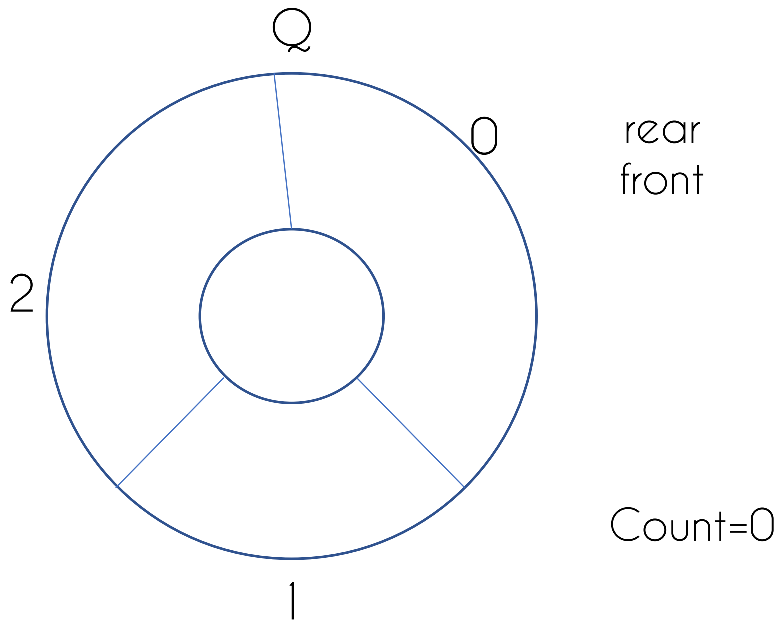
```
struct cQueue
{
 int front,rear,count;
 int cQueue[3];
} cQueue;

void create(cQueue *cq)
{
 cq->front = 0;
 cq->rear = 0;
 cq->count = 0;
}
```

Activity

3. Draw the Circular Queue according to the segment code below:

```
enQueue(Q,A)
enQueue(Q,B)
enQueue(Q,C)
deQueue()
deQueue()
deQueue()
enQueue(Q,D)
enQueue(Q,E)
```



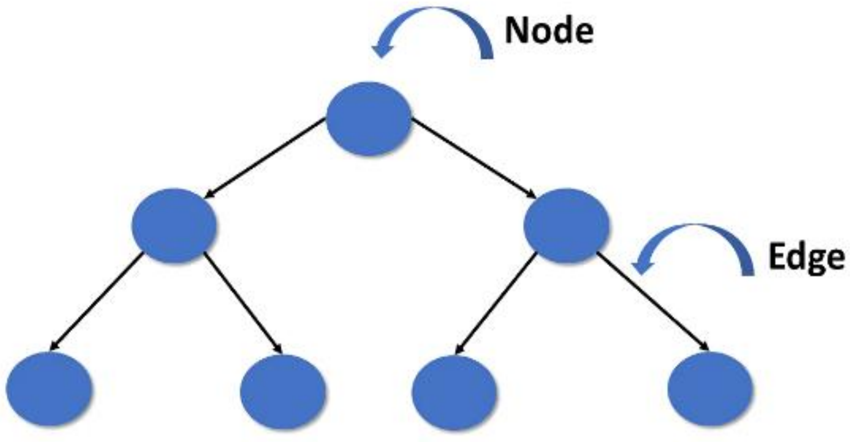
CHAPTER 5

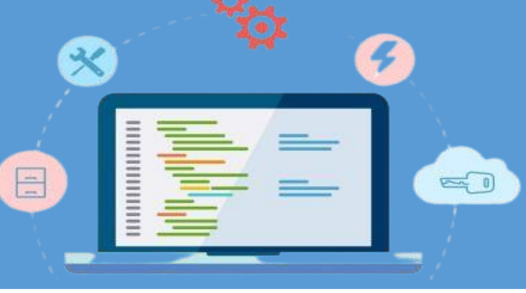
TREES

DEFINITION OF TREE

Trees represent one of the most important types of data structures in computing. They can be implemented in virtually any programming language.

The tree is a nonlinear hierarchical data structure and comprises a collection of entities known as nodes. It connects each node in the tree data structure using "edges", both directed and undirected.

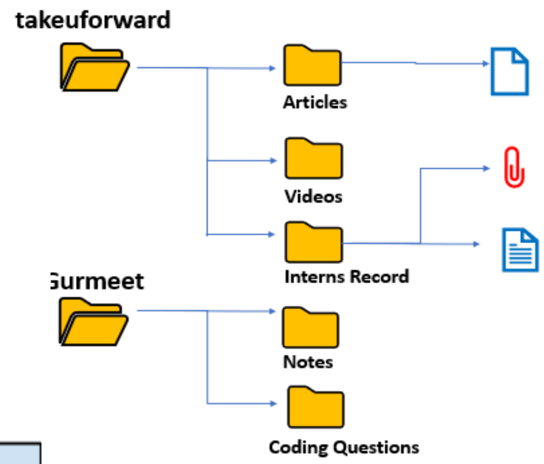


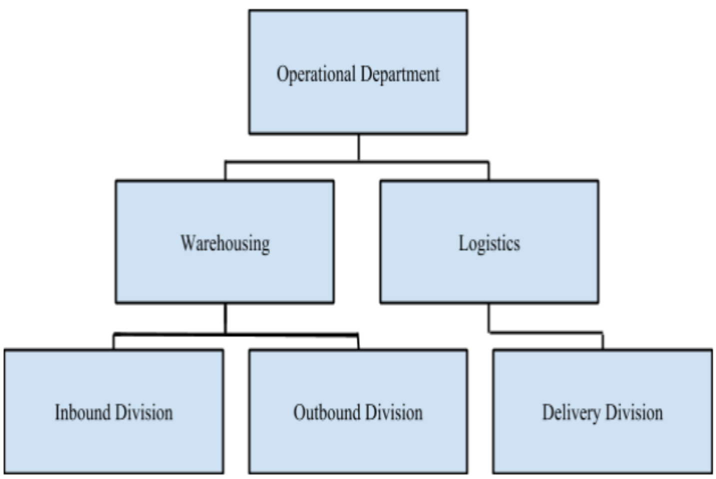


Application Of Tree

The information that we store in our computers is in the form of a hierarchy where every folder has some files stored in it.

Folders hierarchy System in Computers

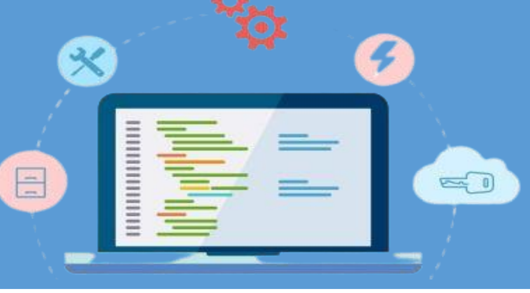




An organization's structure

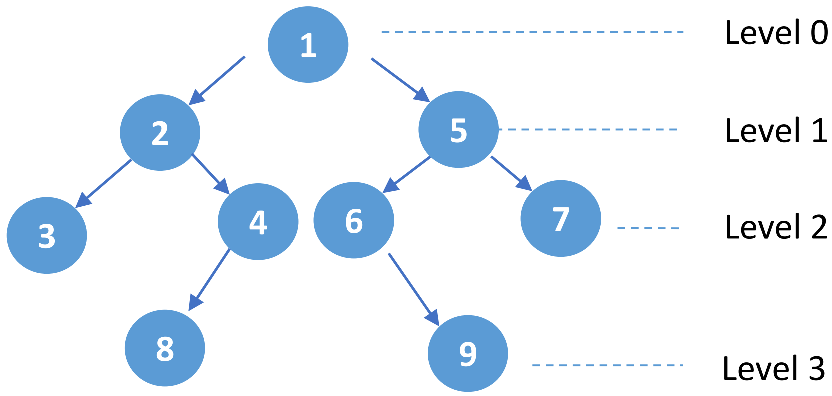
Other Application

1. Store hierarchical data
2. Decision trees
3. In Computer graphics
4. In java virtual machine

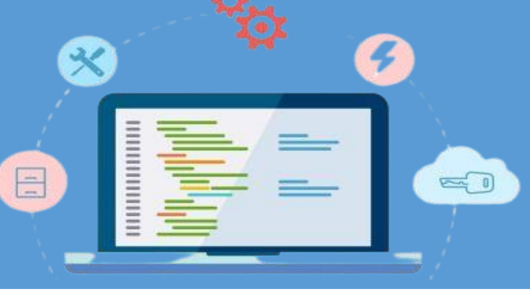


Tree Terminology

Tree is a hierarchical data structure defined as a collection of nodes. Nodes represent value and nodes are connected by edges. A tree has the following properties:

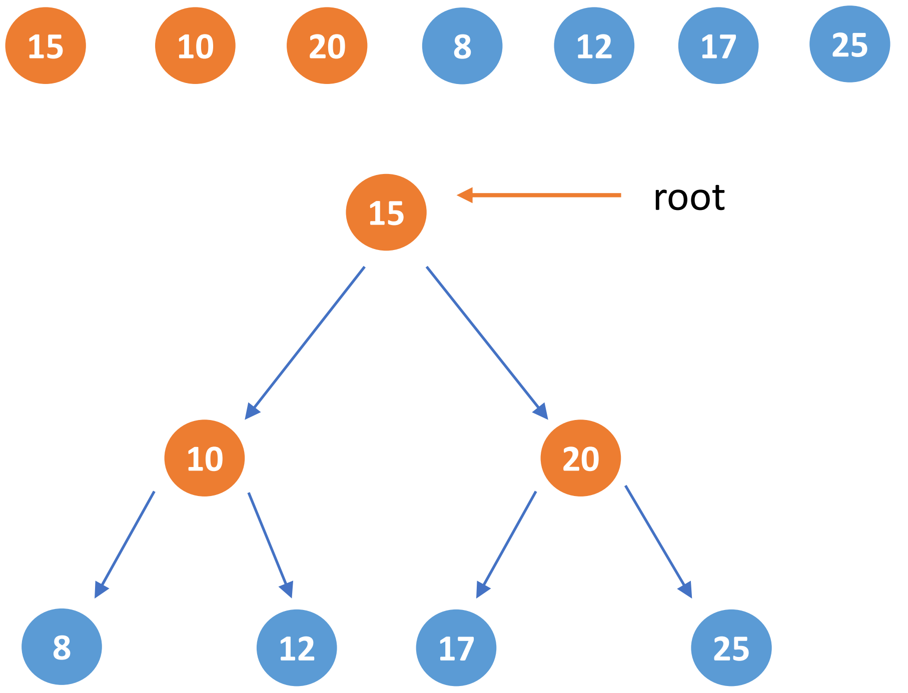


Terminology	Description	Example
Root	Root is a special node in a tree. The entire tree originates from it. It does not have a parent.	1
Parent Node	Parent node is an immediate predecessor of a node	2 is parent of 3 & 4
Child Node	All immediate successors of a node are its children.	3 & 4 are children of 2
Leaf	Node which does not have any child is called as leaf	3,8,9 and 7
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.	Line between 2 & 3 is edge
Siblings	Nodes with the same parent are called Siblings.	3 & 4 are siblings
Path / Traversing	Path is a number of successive edges from source node to destination node.	1-2-3
Degree of Node	Degree of a node represents the number of children of a node	Degree of 2 is 2 and of 6 is 1



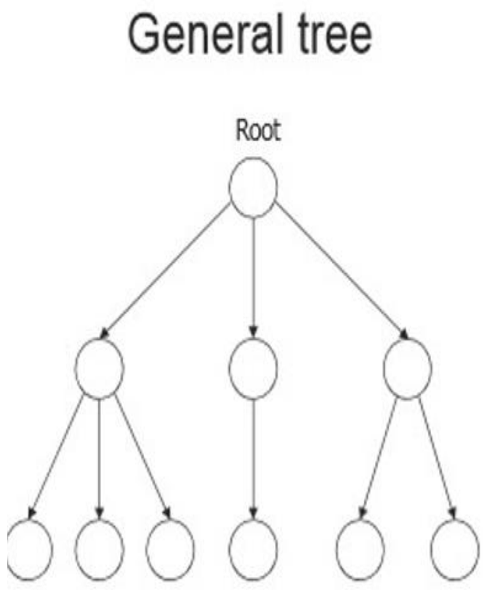
Binary Tree

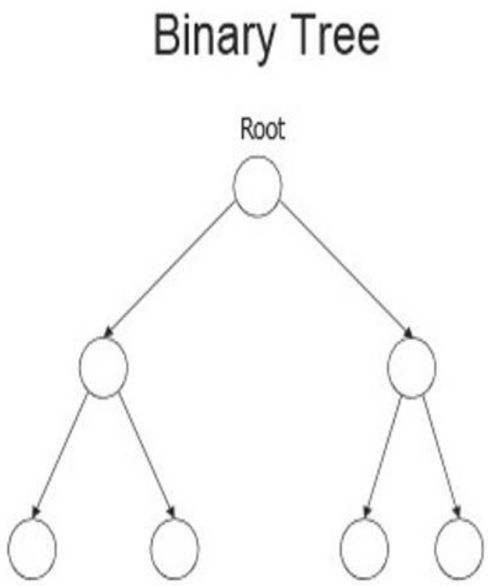
A **binary tree** is a more focused version of a tree data structure. Each node is only allowed to have a maximum of 2 children, a left hand node and a right hand node. The left hand node will generally have a value less than its parent, and the right hand node will have a value greater than its parent.



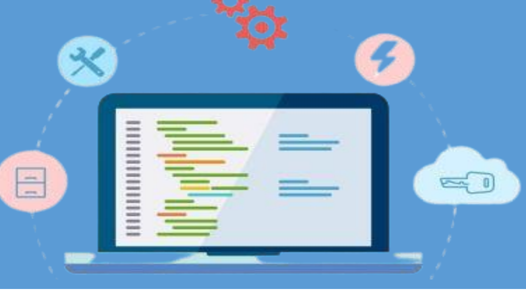
Example

Tree Vs Binary Tree

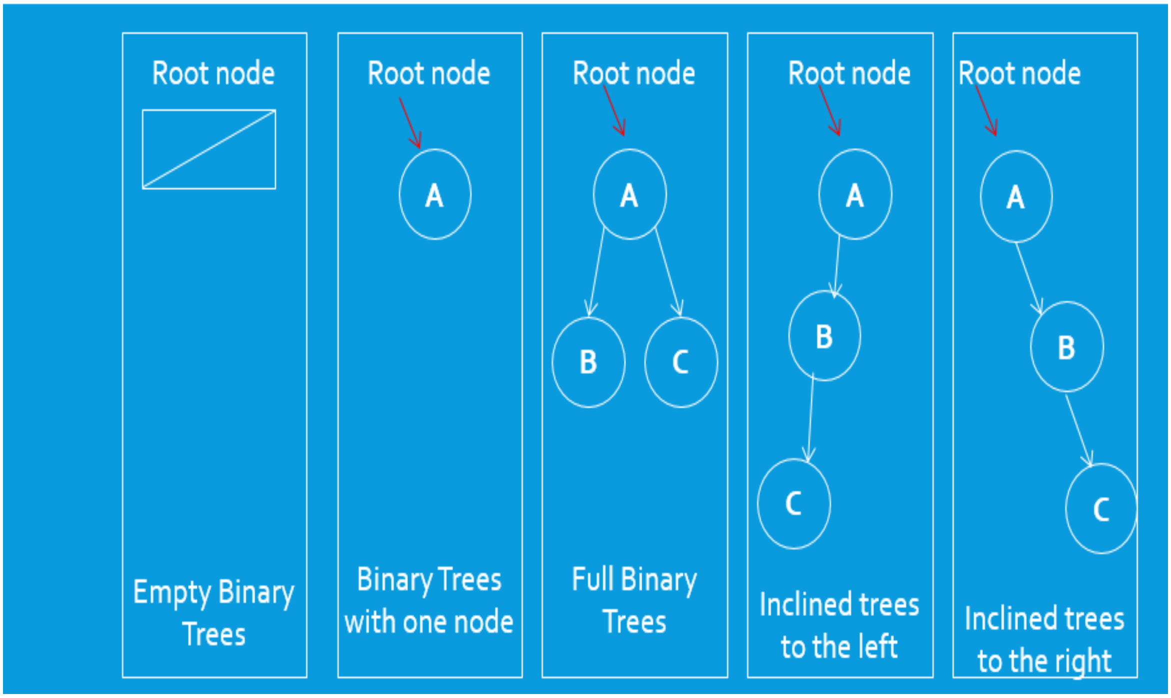




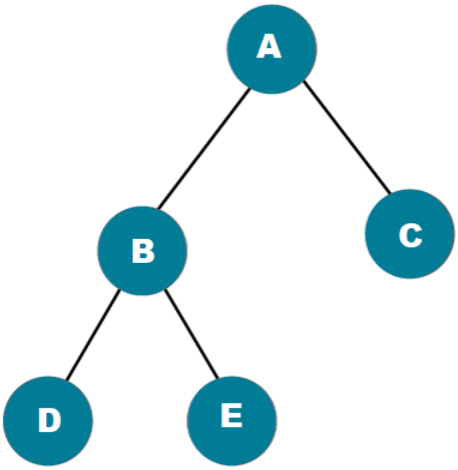
General Tree	Binary Tree
Tree **can't** be empty	Tree can be **empty**
There is no limit on the **degree of node**	Nodes in a binary tree cannot have more than **degree 2**
Subtree of general tree are **not ordered**.	Subtree of binary tree are **ordered**
Each **node** have in-degree **one** and maximum out-degree **n**	Each node have in-degree **one** and maximum out-degree **2**.

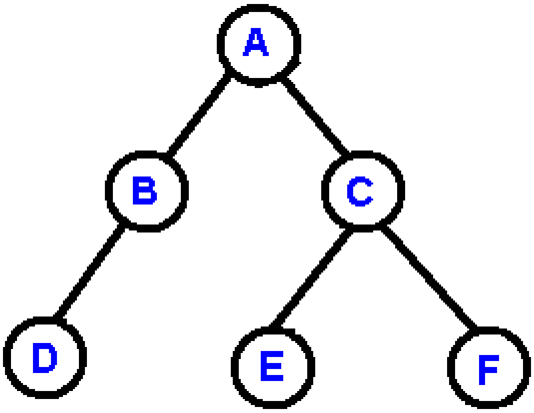


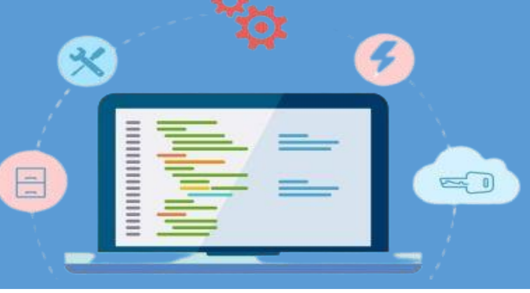
Binary Tree



Example



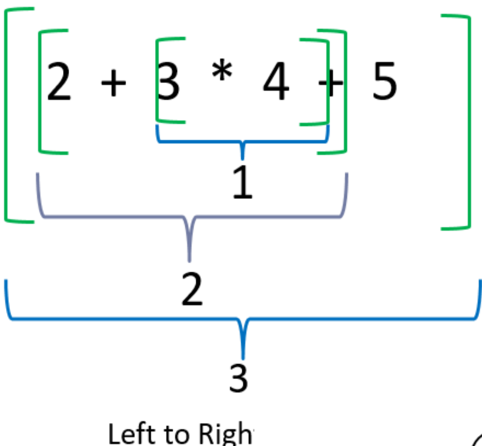


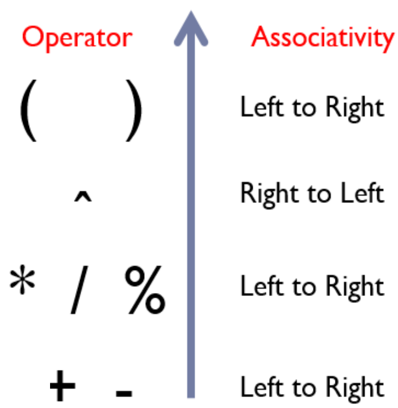


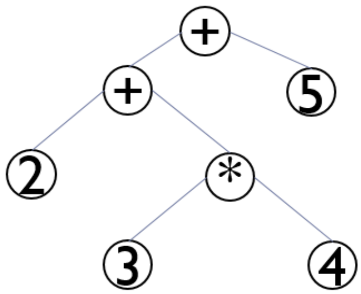
Construct Binary Tree from Arithmetic Expression and vice versa

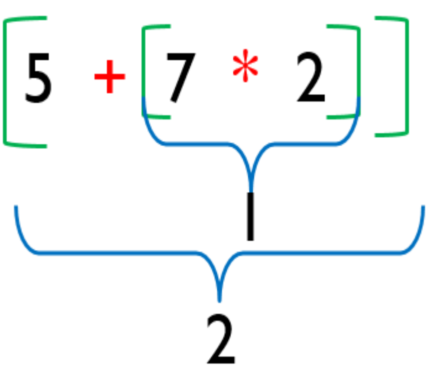
To construct Binary Tree :

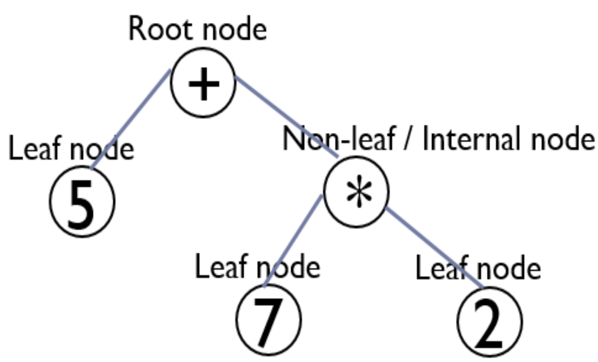
- Each leaf node represents an operand
- Each non-leaf node or internal node represents a single binary operator

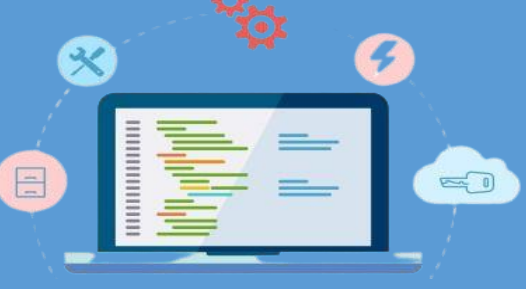










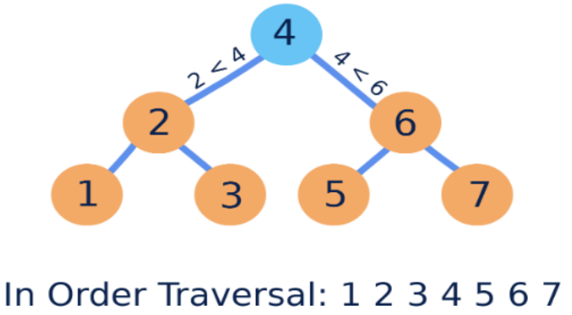


Binary Search Trees

To construct Binary Search Tree :

Definition

A *binary search tree* (BST) is a *binary tree* where every node in the left subtree is less than the root, and every node in t he right subtree is of a value greater than the root.

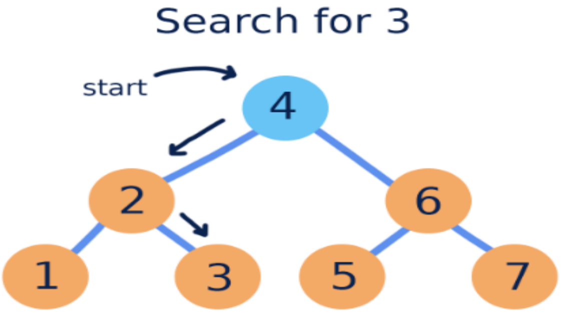


Searching

Binary search trees are called "search trees" because they make searching for a certain value more efficient than in an unordered tree. In an ideal binary search tree, we do not have to visit every node when searching for a particular value.

Here is how we search in a binary search tree:

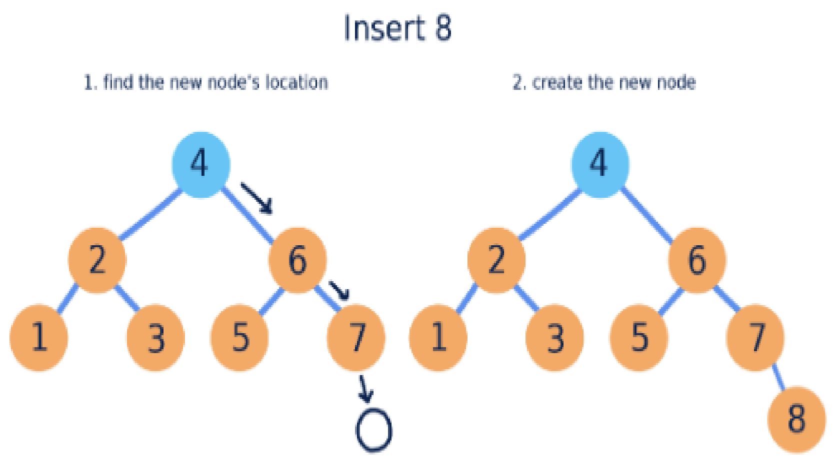
- Begin at the tree's *root node*
- If the value is smaller than the current node, move left
- If the value is larger than the current node, move right



To construct Binary Search Tree :

Inserting

New nodes in a binary search tree are always added at a *leaf* position. Performing a search can easily find the position for a new node.

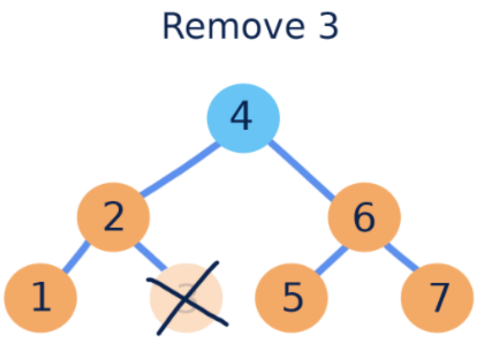


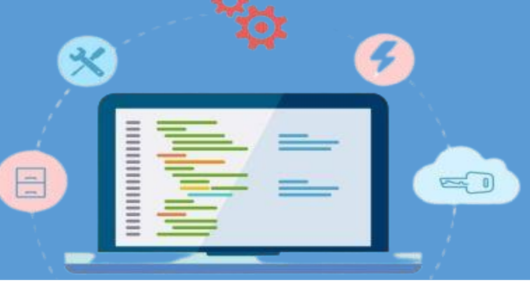
Removing

When removing from a binary search tree, we are concerned with keeping the rest of the tree in the correct order. This means removing is different depending on whether the node we are removing has children.

There are three cases:

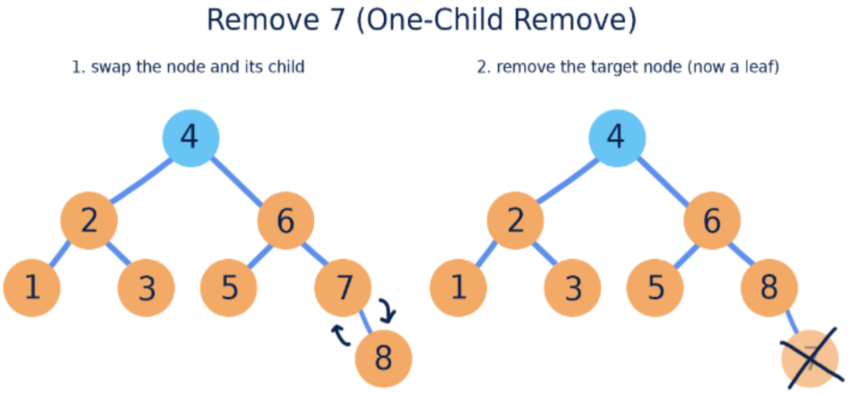
- If the node being removed is a leaf, it can simply be deleted.



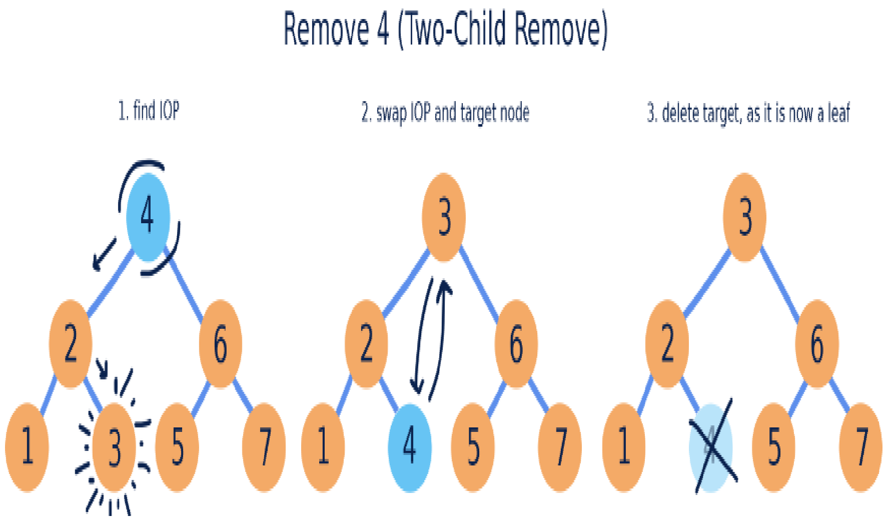


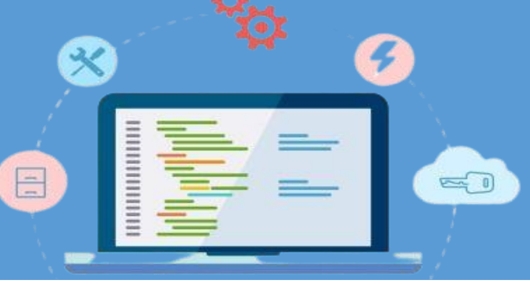
Removing..

- If the node has a single child, (left or right) we must move the child into the position of the node when deleting it..



- If the node has two children, we must first find the *In-Order Predecessor* (IOP): the largest node in our node's left subtree. The IOP is always a leaf node, and can be found by starting at the left subtree's root and moving right. We can then swap the node being removed with its IOP and delete it, as it is now a leaf.



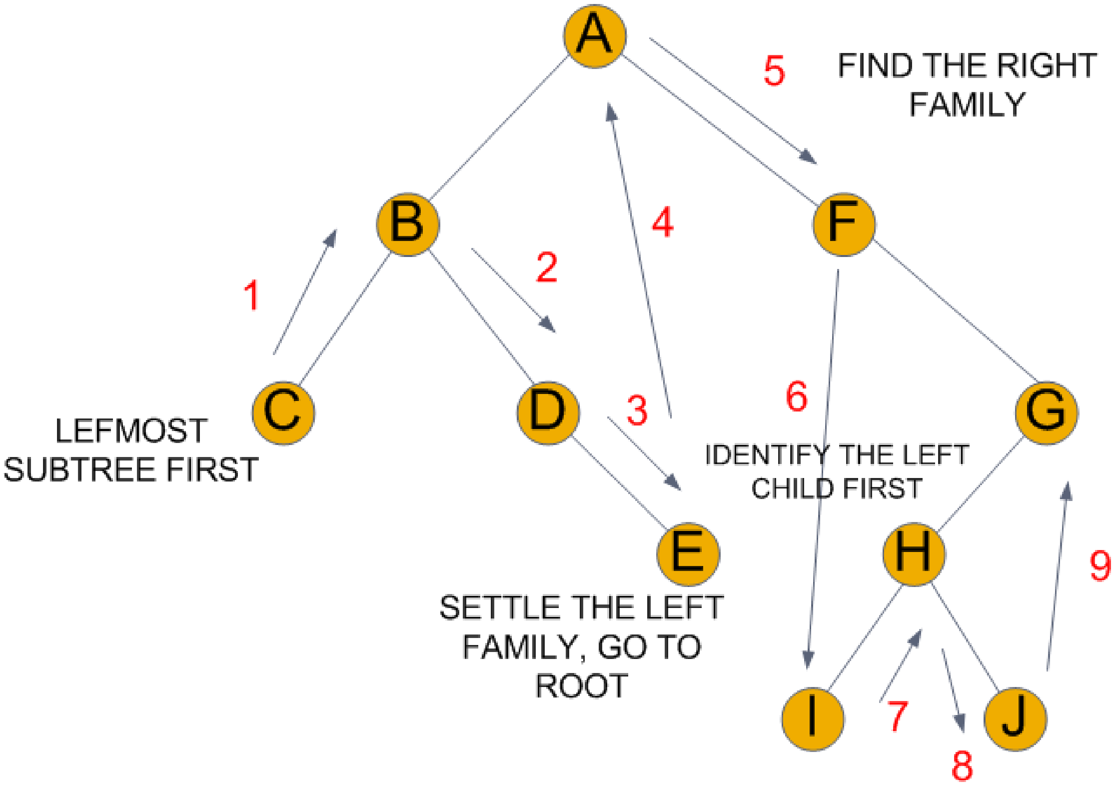


There are three ways which we use to traverse a tree

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

In-order Traversal

- Visit the left sub tree if exist
- Visit Root
- Visit the right sub tree if exist

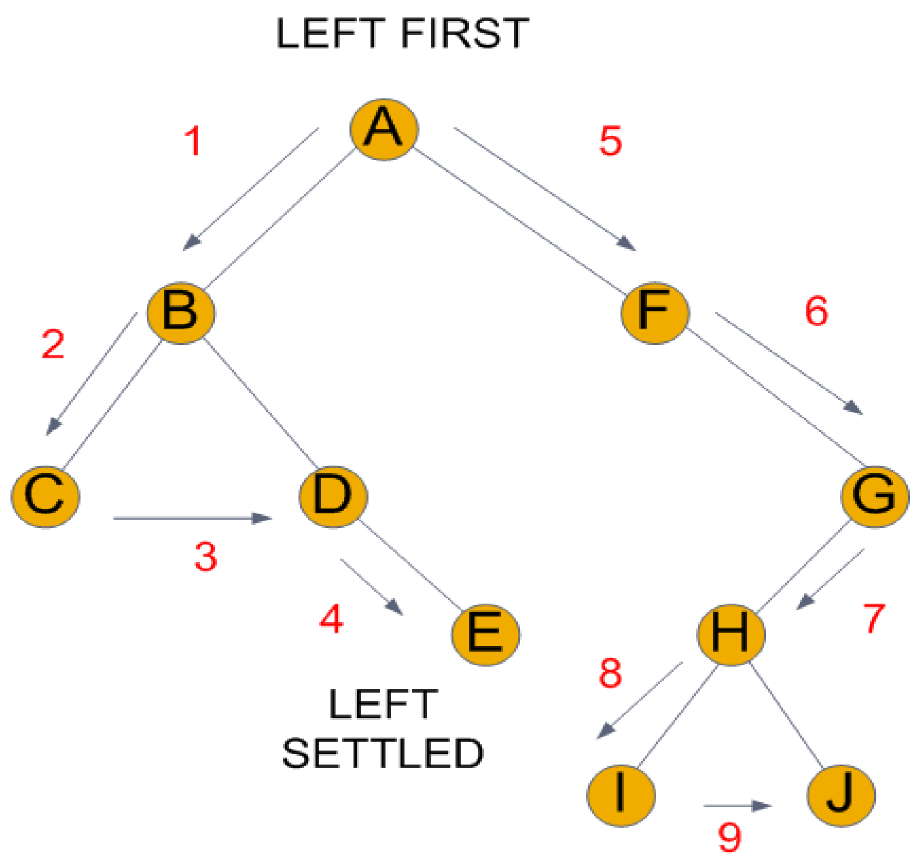


Inorder :

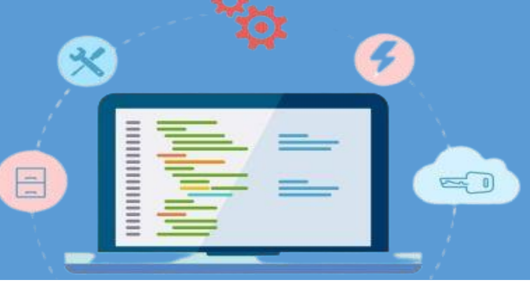
C, B, D, E, A, F, I, H, J, G

Pre-order Traversal

- Visit Root
- Visit Subtrees left to right

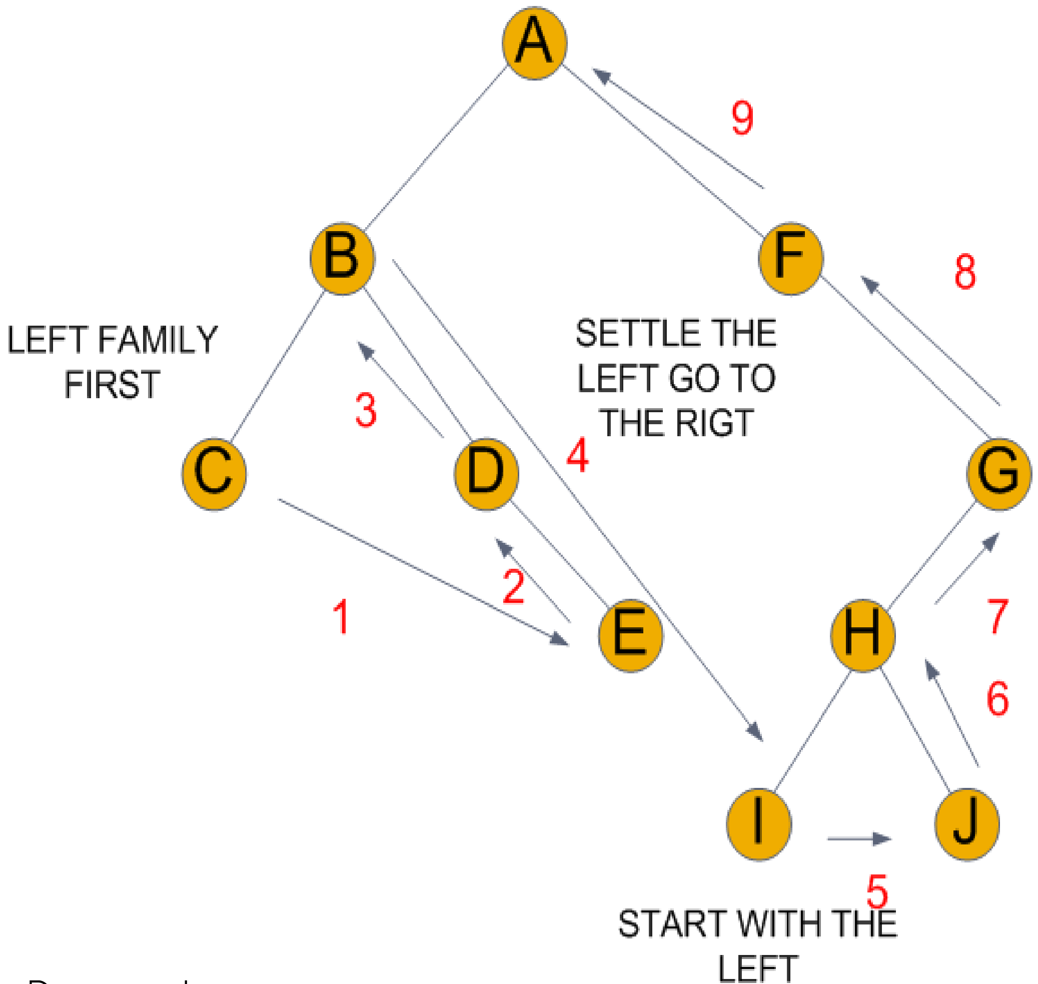


Preorder :
A , B, C, D, E, F, G, H, I, J



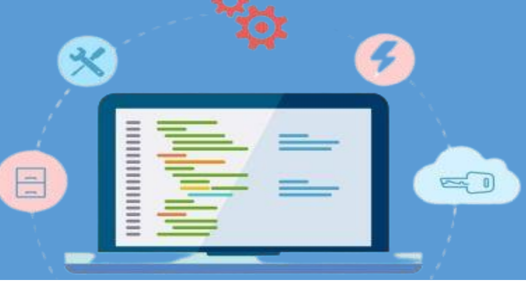
Post-order Traversal

- Visit the left sub tree if exists.
- the right sub tree if exists
- Visit root



Postorder :

C, E, D, B, I, J, H, G, F, A



Conversion

Postfix

Prefix

Postfix, Prefix & Infix

Infix :	(A + B)
Postfix :	AB+
Prefix :	+AB

Operators: A & B   Operands: +

Infix

Operation : Any Expression of algebraic format (Example : A + B)
Operands : A and B or 5 & 6 are operands
Operators : +. –, %,*,/ etc are operators

Activity

Draw a Binary Tree from the Arithmetic Expressions below:

i. A + B * C / (D – E)

ii. A * B + (C – D / E)

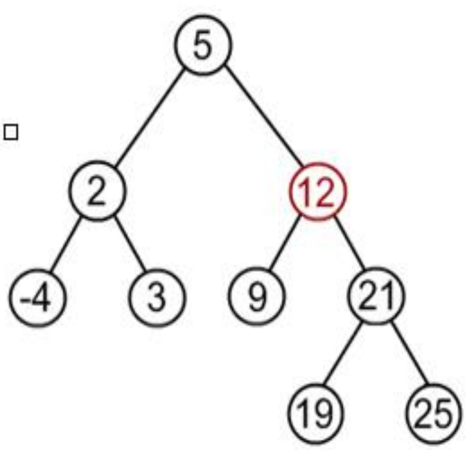
iii. A * B / (5 * C) + 10

Activity

1.

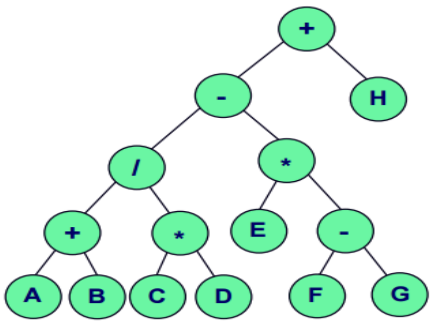
2.

Remove 12 from a BST.

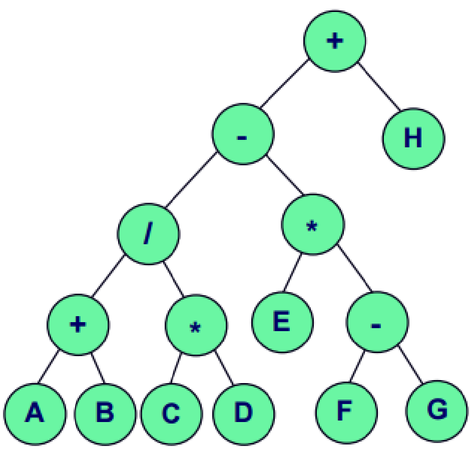


Activity

3. Find PreOrder Traversal

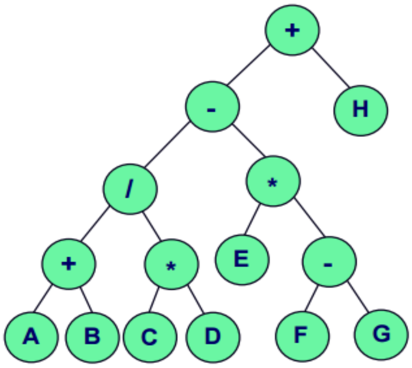


4. Find InOrder Traversal



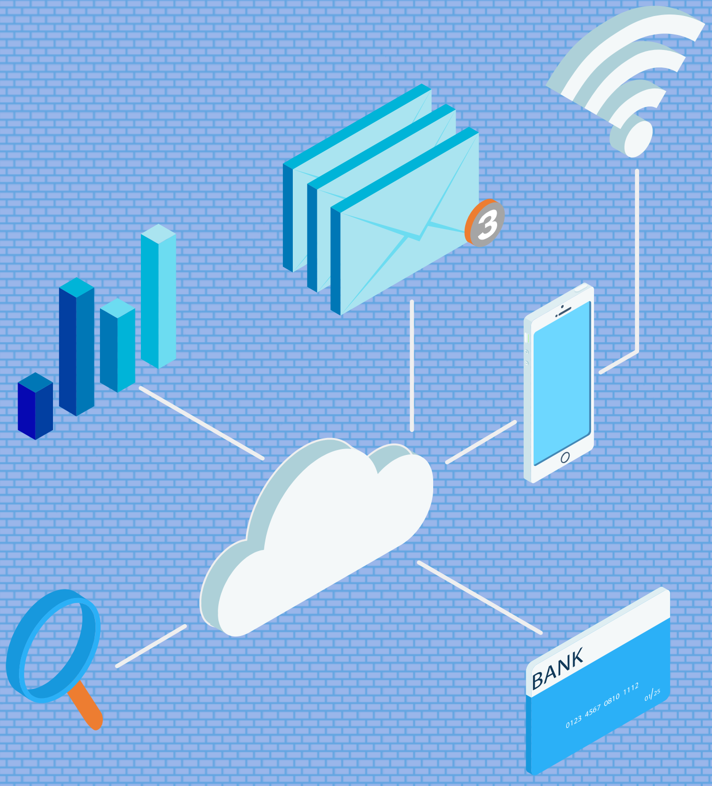
Activity

5. Find PostOrder Traversal



CHAPTER 6

SORTING & SEARCHING



DEFINITION

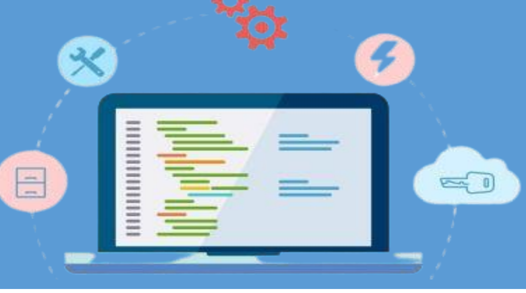
- Sorting refers to **arranging** data in a particular format.
- Particular format
  - ✓ increasing order
  - ✓ decreasing order
- It arranges the data in a sequence which makes searching easier.

The importance of Sorting

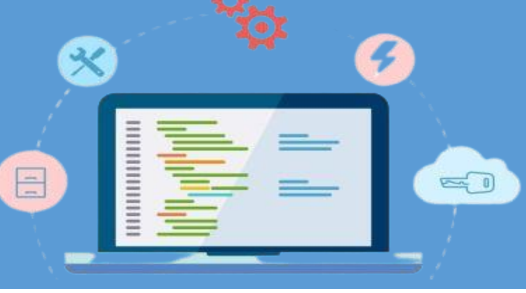
To represent data in more readable formats

Speed up the search process to the data

Simplify the process of understanding and analysis of data collection

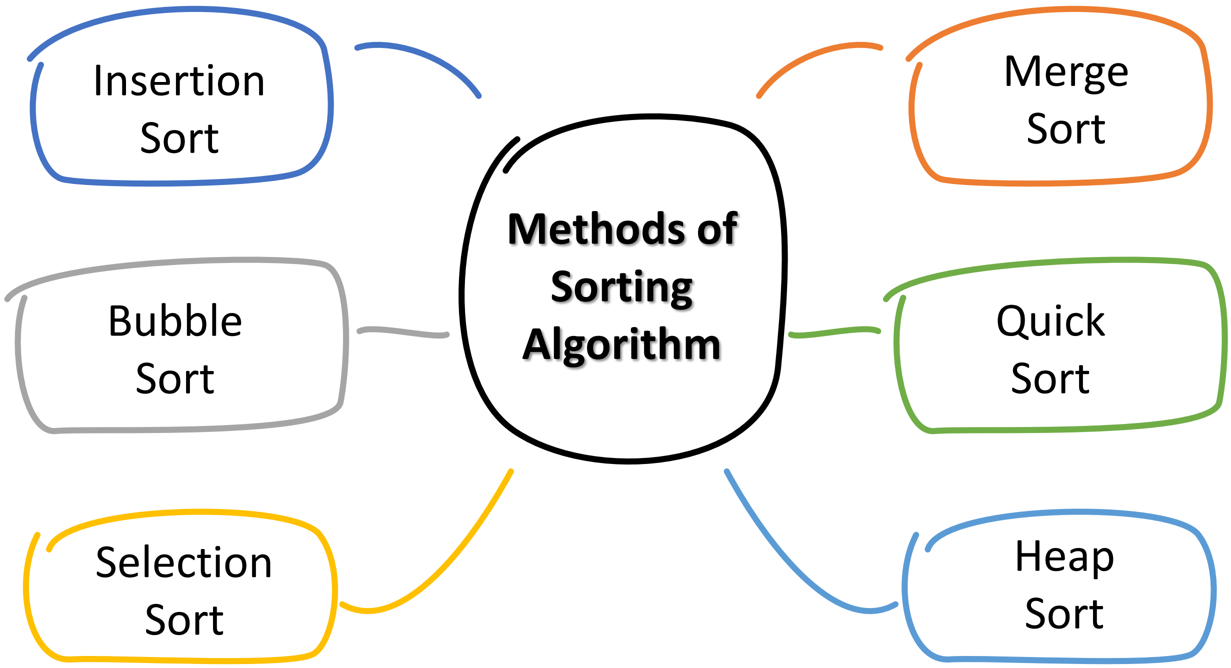


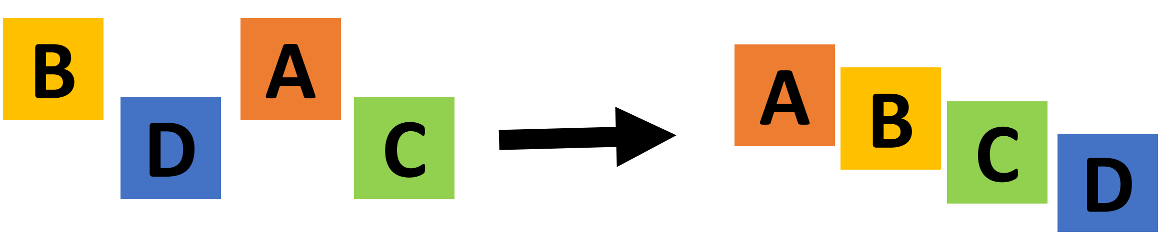
Example Of Sorting In Real-life Scenarios

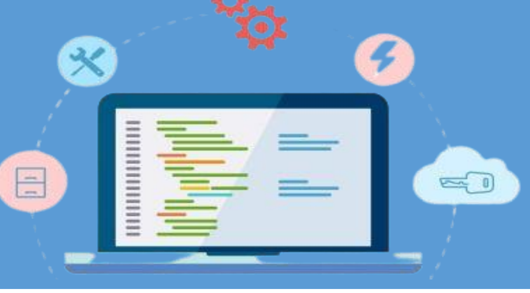


Sorting Algorithm

Sorting technique that is used to sort the data in a sequence order in ascending order or in descending order.



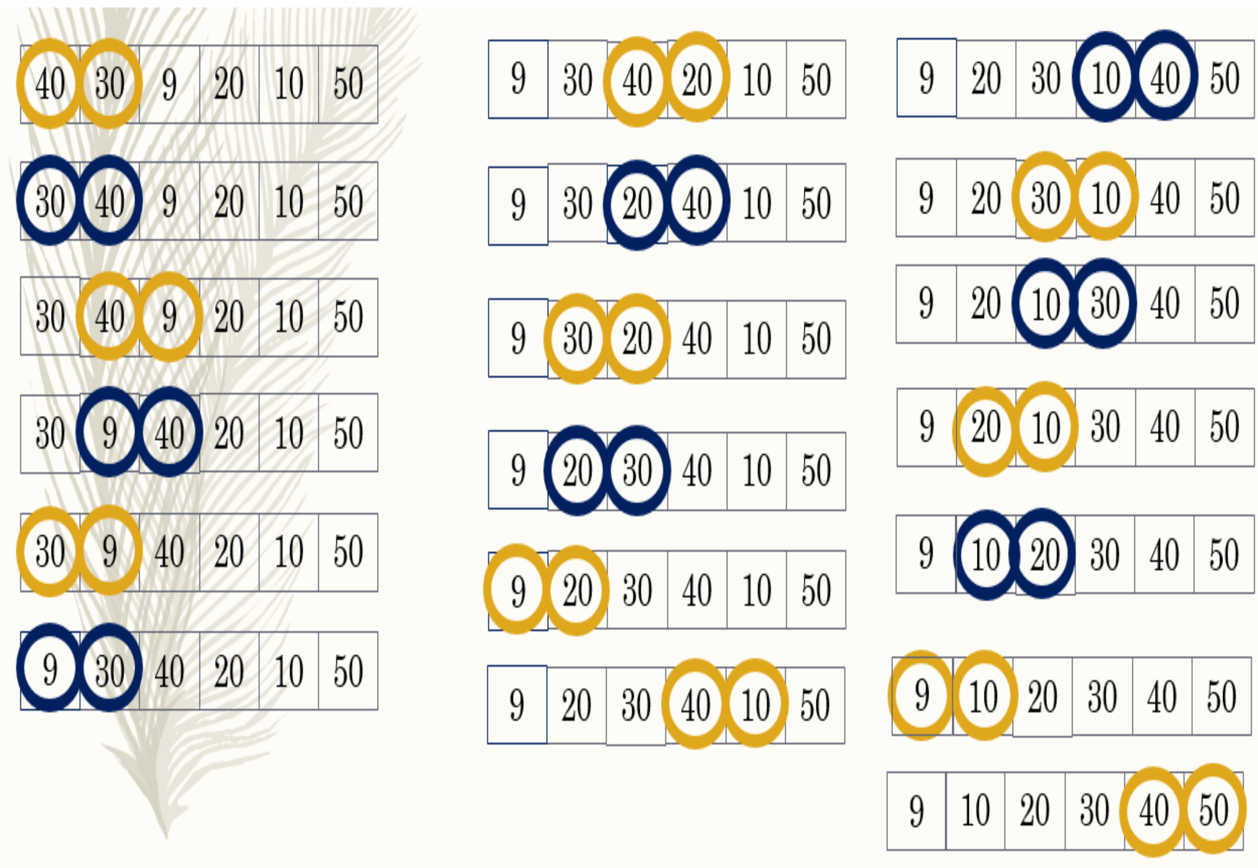


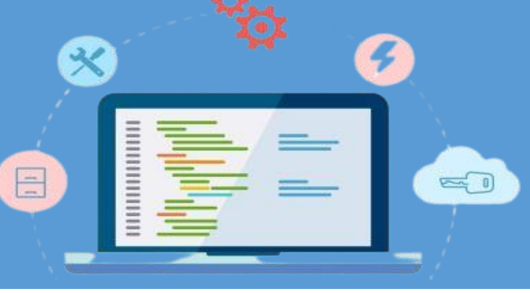


Insertion Sort

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Insertion Sort (in ascending order)

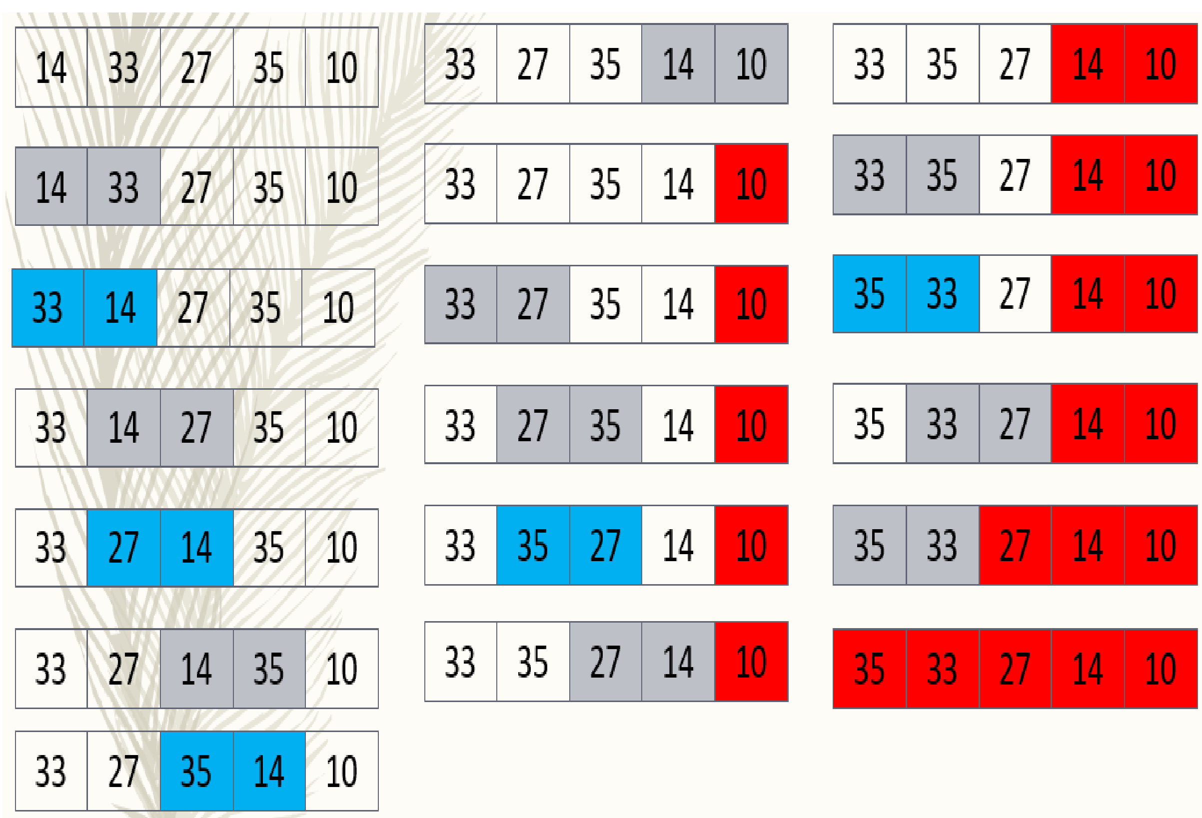


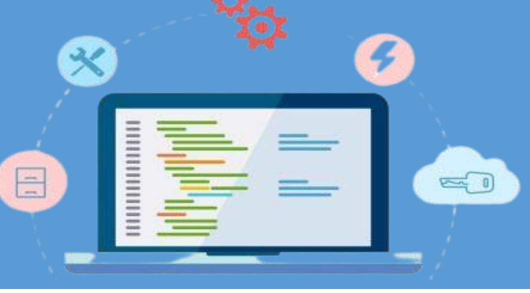


Bubble Sort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent pairs and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. This algorithm starts at the beginning of the array, compares each element with the element immediately to the right of it, and makes a swap if the elements are out of order with each other.

Process Bubble Sort (in descending order)

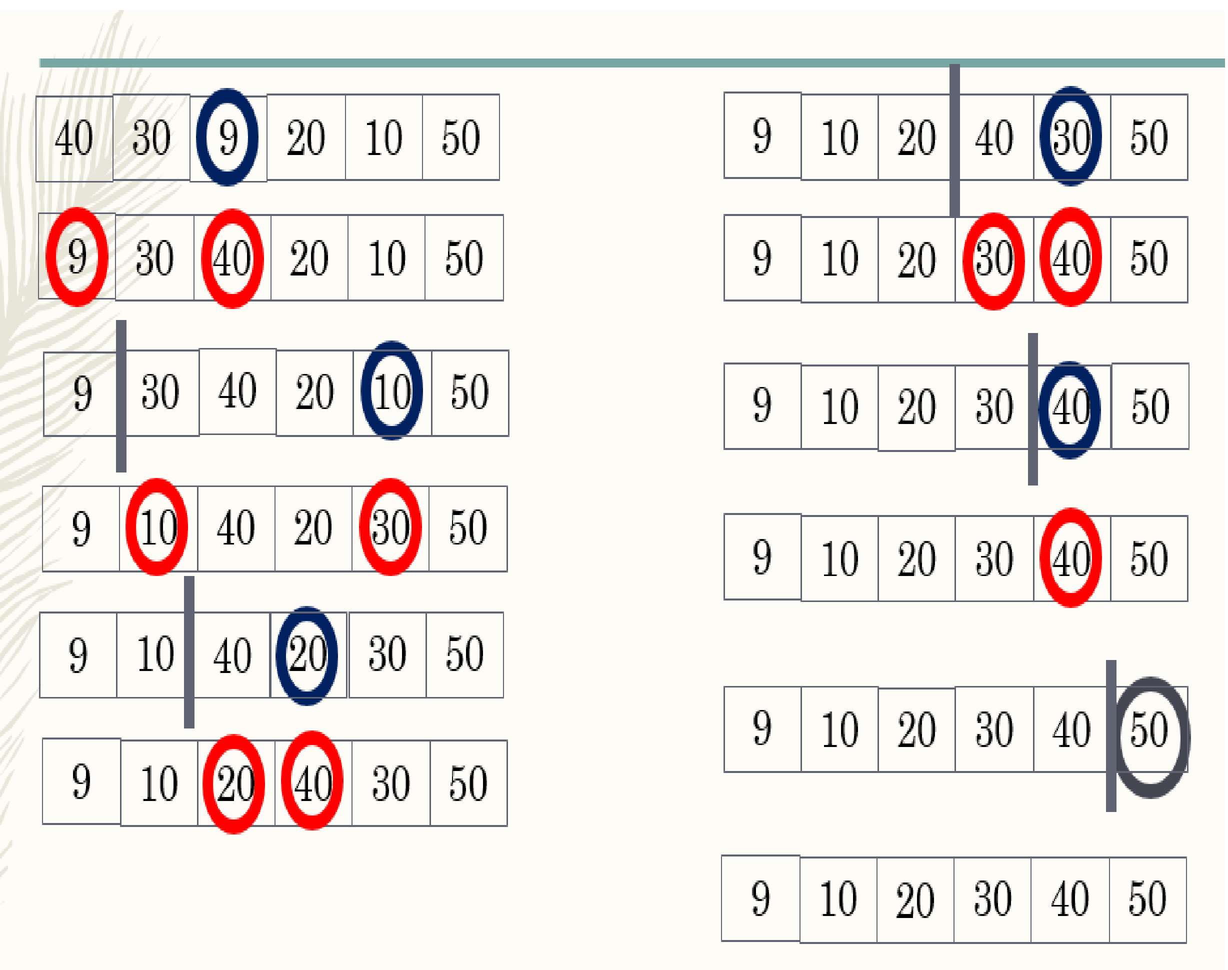


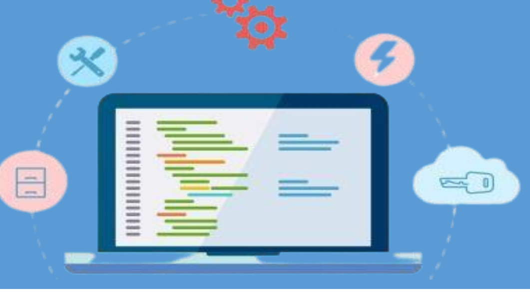


Selection Sort

- Finds the smallest element in the array and exchanges it with the element in the first position.
- Then finds the second smallest element and exchanges it with the element in the second position
- Continues until the entire array is sorted in ascending order

Sort the following numbers in ascending order



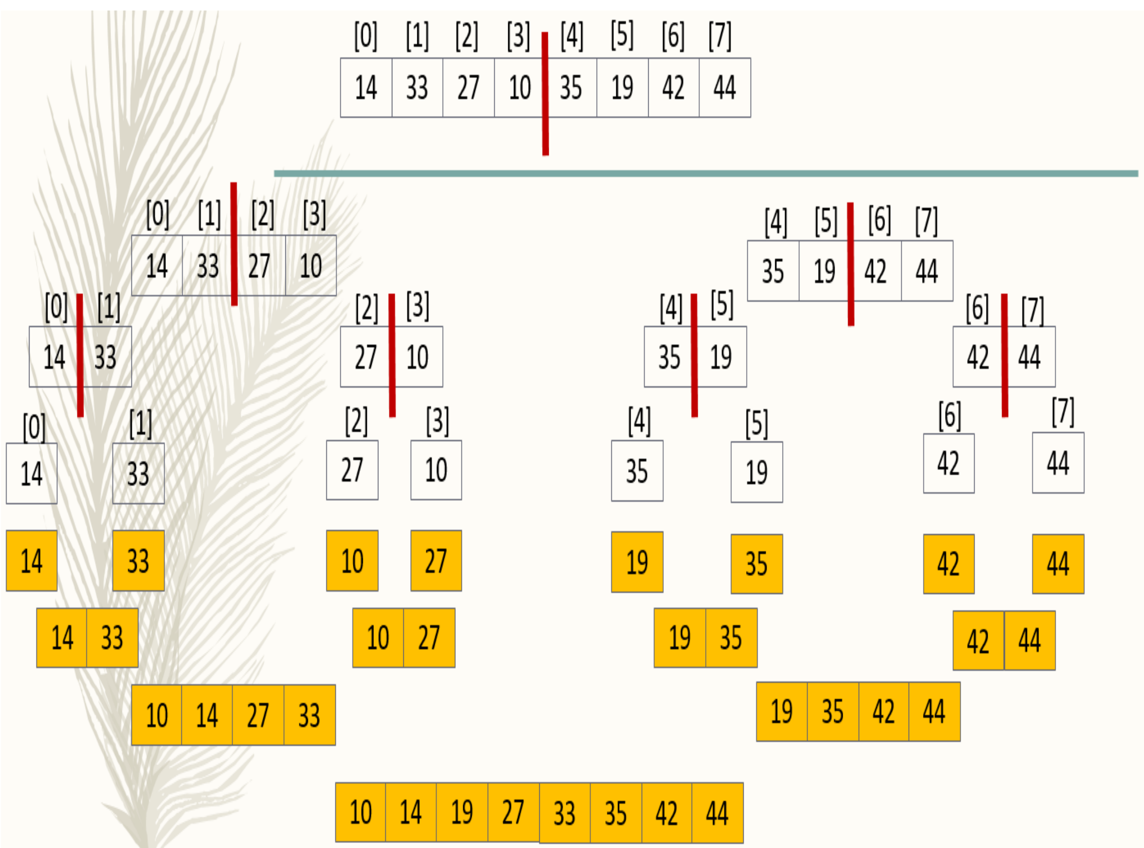


Merge Sort

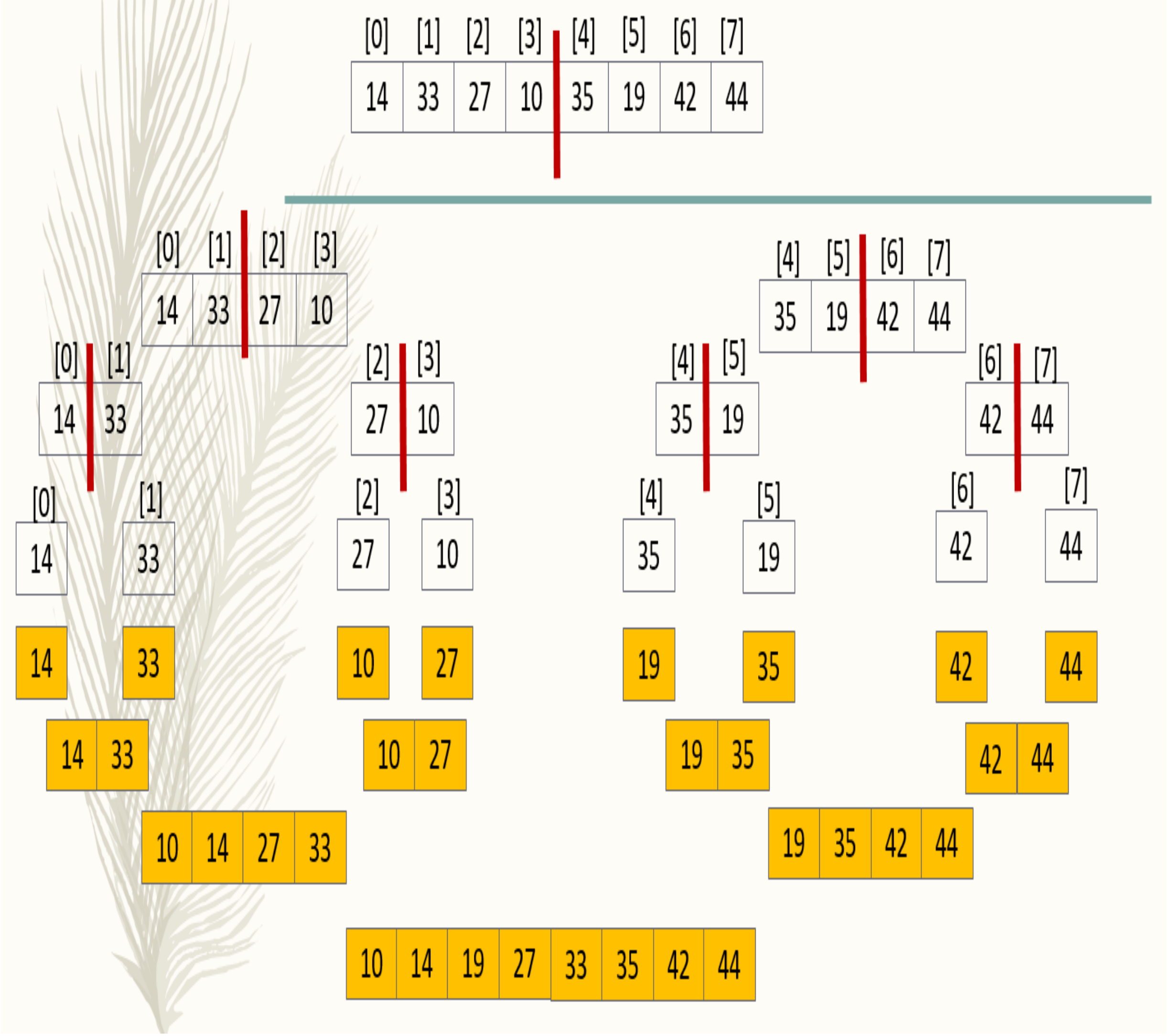
- Dividing the data elements in the array to smaller groups
- Carry out the sorting in the smaller group
- Use divide and conquer approach

Three Steps in merge sort

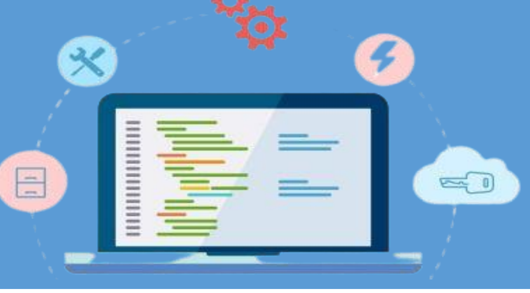
1. Divide – break the problem into sub problems
2. Conquer – sub problems will be solved
3. Merge – combine the solutions for each sub problems to solve the original problem



Example merge sort 1

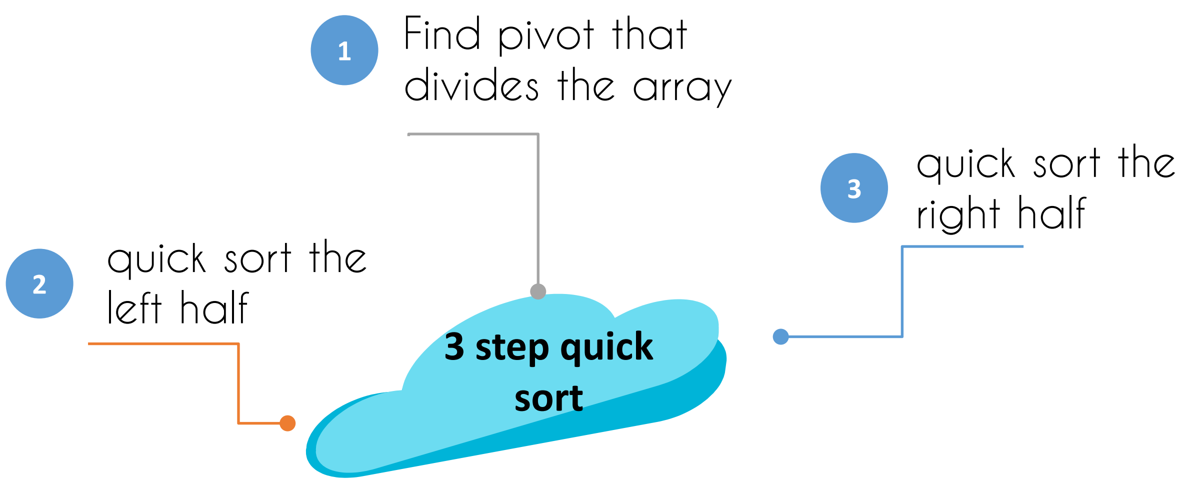


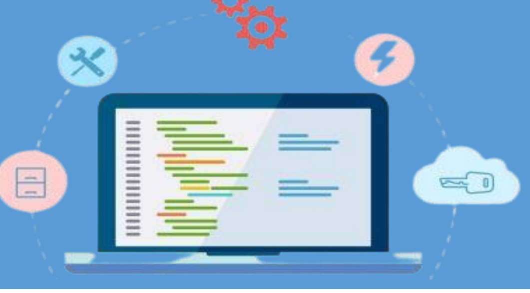
Example merge sort 2

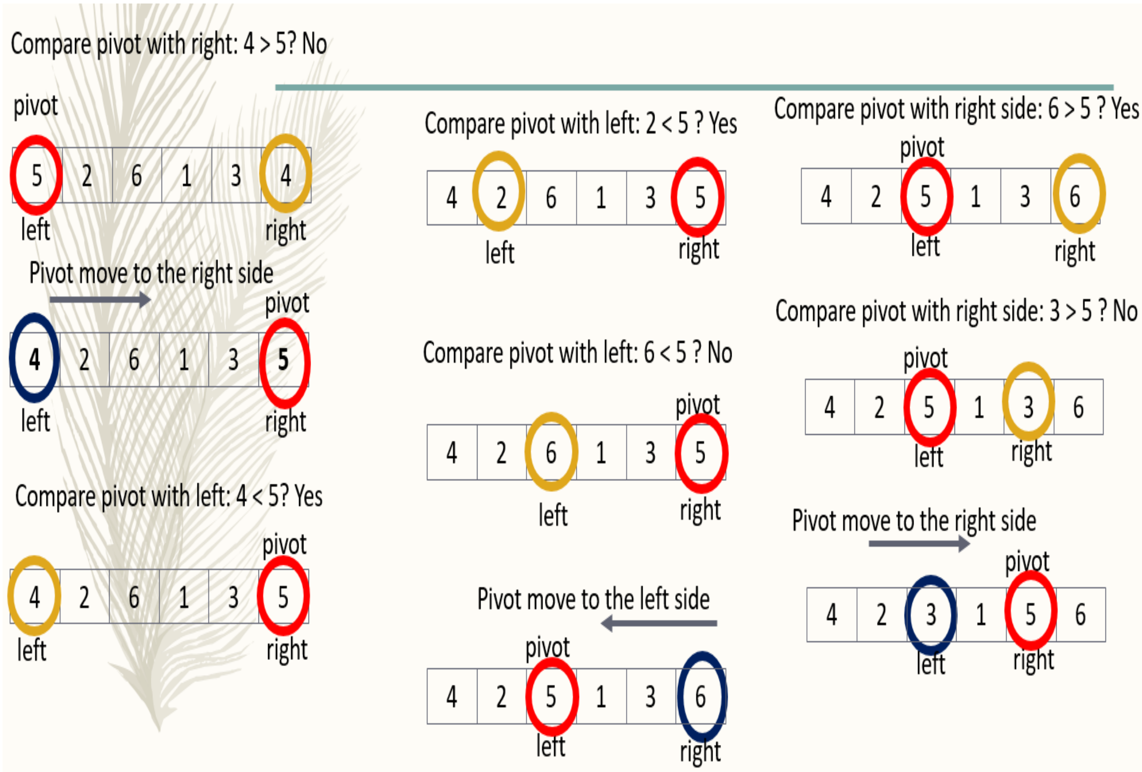


Quick Sort

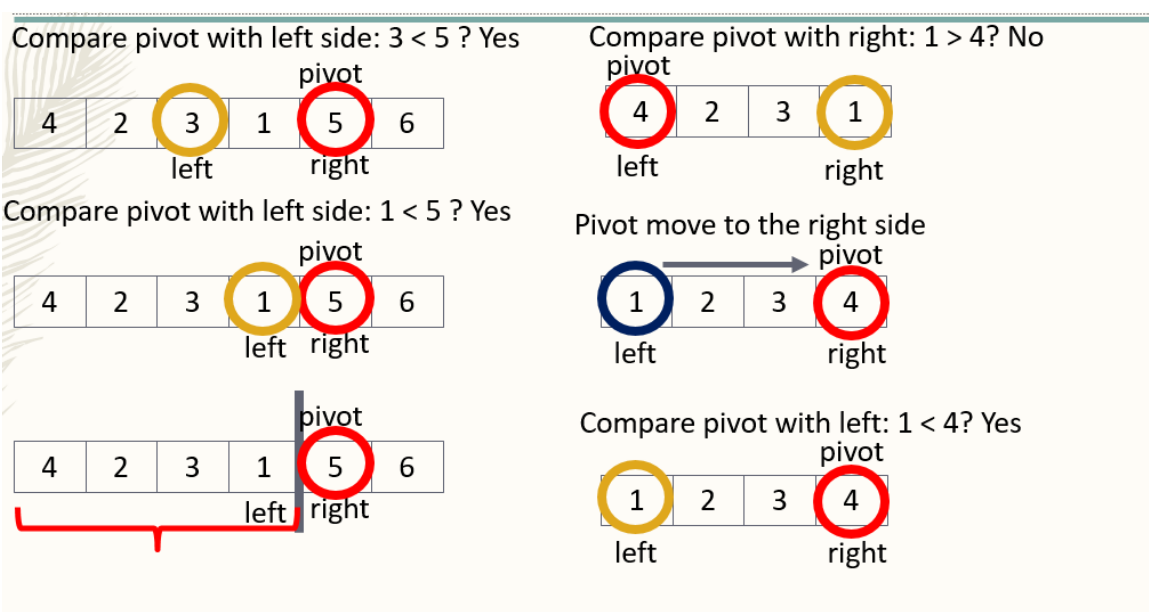
1. Uses the idea of divide an conquer.
2. It finds the element called pivot which is divides the array into two halves in such a way that the elements in the left half are smaller than pivot and elements in the right are greater than pivot.
3. Three steps in quick sort



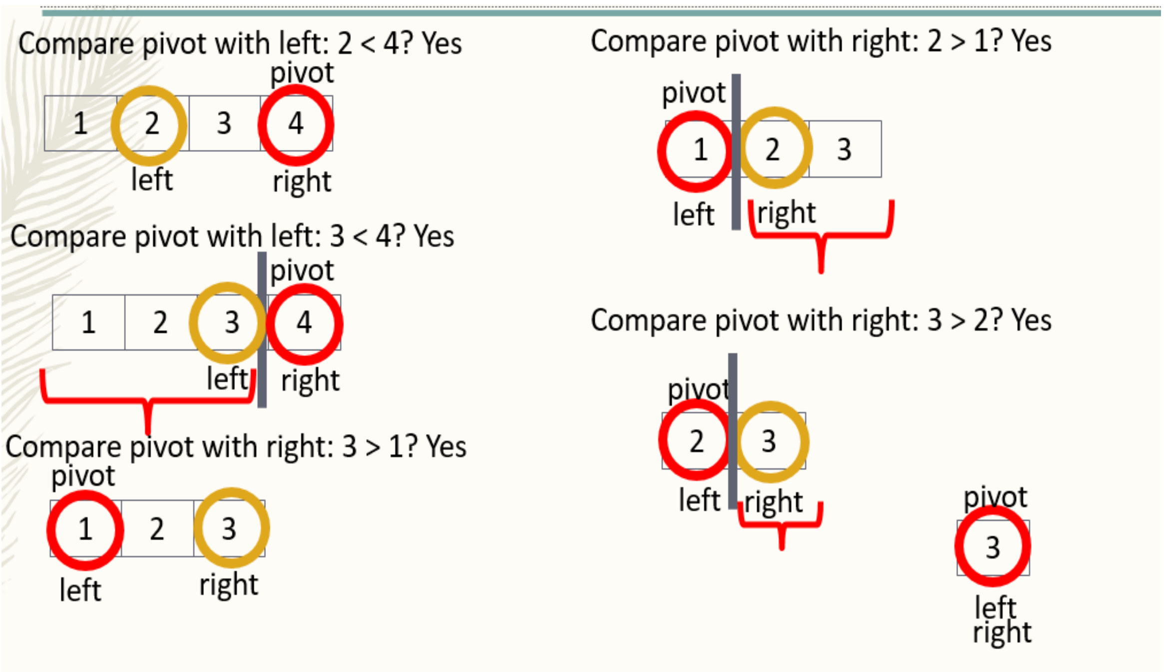


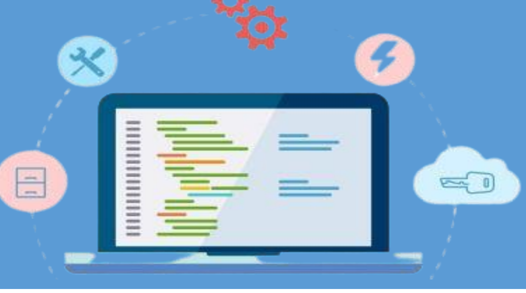


Compare pivot with right: 4 > 5? No
pivot
5 2 6 1 3 4
left
right
Pivot move to the right side
pivot
4 2 6 1 3 5
left
right
Compare pivot with left: 4 < 5? Yes
pivot
4 2 6 1 3 5
left
right
Compare pivot with left: 2 < 5 ? Yes
4 2 6 1 3 5
left
right
Compare pivot with left: 6 < 5 ? No
pivot
4 2 6 1 3 5
left
right
Pivot move to the left side
pivot
4 2 5 1 3 6
left
right
Compare pivot with right side: 6 > 5 ? Yes
pivot
4 2 5 1 3 6
left
right
Compare pivot with right side: 3 > 5 ? No
pivot
4 2 5 1 3 6
left
right
Pivot move to the right side
pivot
4 2 3 1 5 6
left
right



Compare pivot with left side: 3 < 5 ? Yes
pivot
4 2 3 1 5 6
left
right
Compare pivot with left side: 1 < 5 ? Yes
pivot
4 2 3 1 5 6
left
right
pivot
4 2 3 1 5 6
left
right
Compare pivot with right: 1 > 4? No
pivot
4 2 3 1
left
right
Pivot move to the right side
pivot
1 2 3 4
left
right
Compare pivot with left: 1 < 4? Yes
pivot
1 2 3 4
left
right

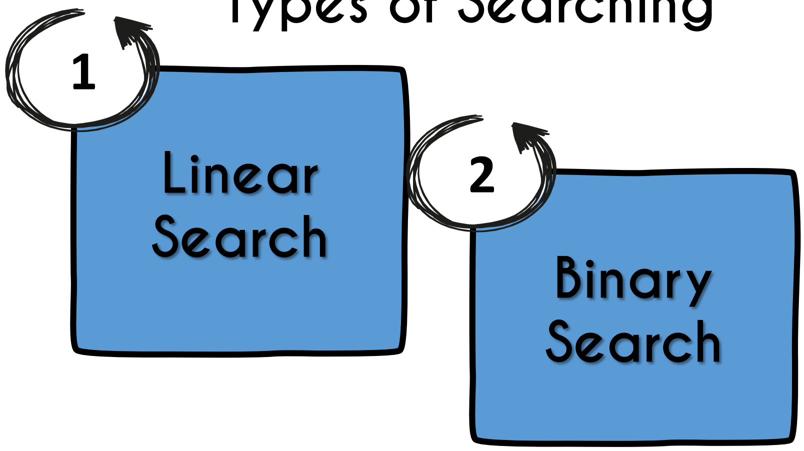

Compare pivot with left: 2 < 4? Yes
pivot
1 2 3 4
left
right
Compare pivot with left: 3 < 4? Yes
pivot
1 2 3 4
left
right
Compare pivot with right: 3 > 1? Yes
pivot
1 2 3
left
right
Compare pivot with right: 2 > 1? Yes
pivot
1 2 3
left
right
Compare pivot with right: 3 > 2? Yes
pivot
2 3
left
right
pivot
3
left
right



Searching

In computer science, a search algorithm, is an algorithm for finding an item with specified properties among a collection of items.

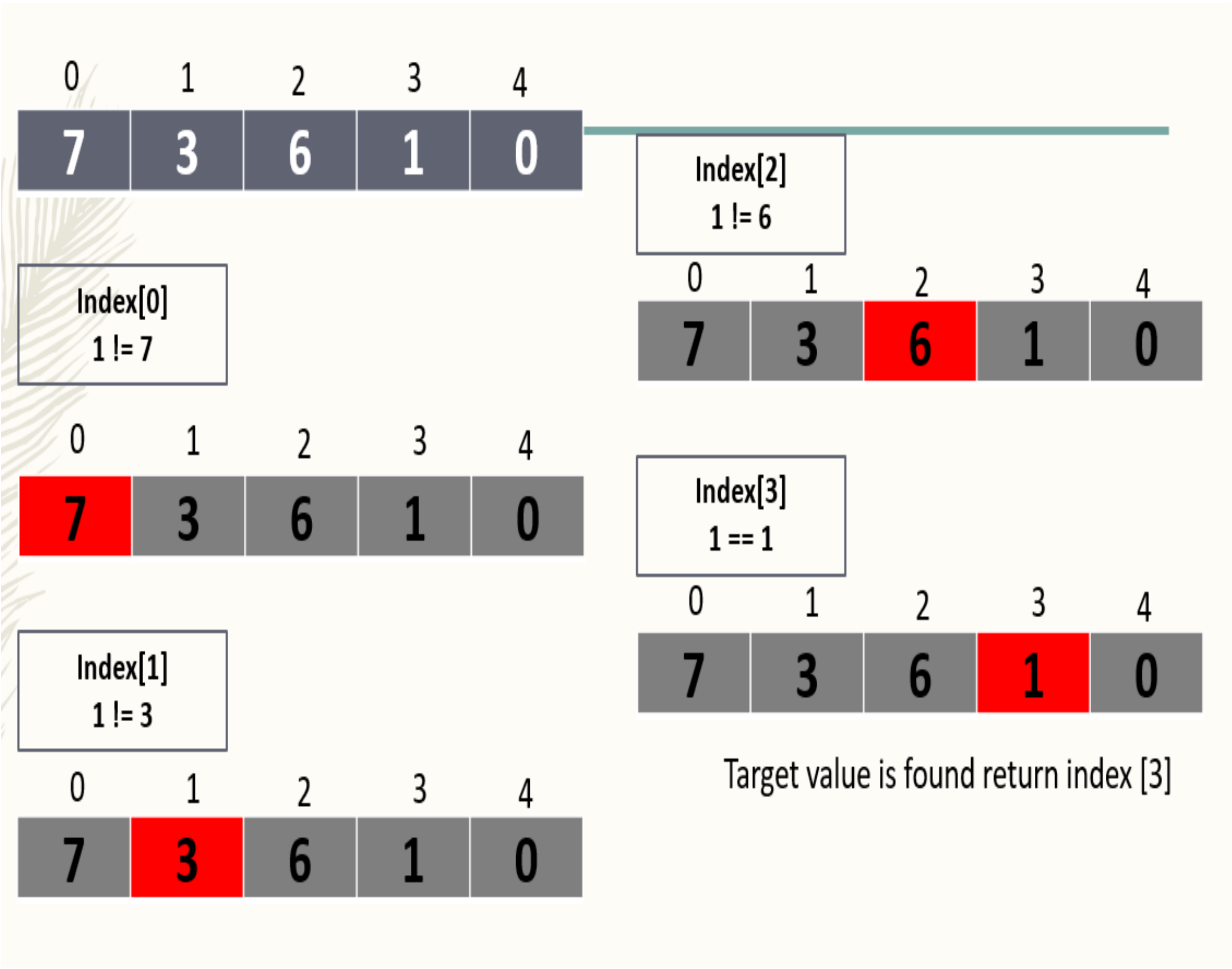
Types of Searching

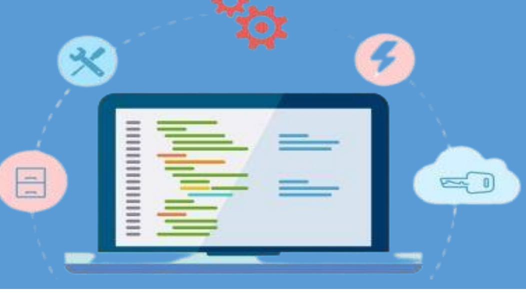


Linear Search

- In computer science, linear search or sequential search is a method for finding a particular value in a list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found
- Linear search is the simplest search algorithm
- Is use to search data when the list is unsorted
- Searching for the key is done one by one from the first element on the list until the key is found or until the last element

Linear Search





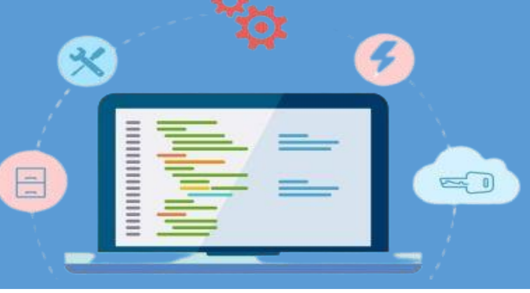
Binary Search

- At each stage, the algorithm compares the input key value with the key value of the middle element of the array. If the keys match, then a matching element has been found so its index, or position, is returned.
- Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right.

Binary Search Implementation

- Algorithm is quite simple. It can be done either recursively or iteratively:
  - ✓Sort the list first
  - ✓get the middle element;
  - ✓if the middle element **equals to** the searched value, the algorithm stops;

- Otherwise, two cases are possible:
  - ✓searched value is **less,** than the middle element. In this case, search the part of the array, before middle element.
  - ✓searched value is **greater,** than the middle element. In this case, search the part of the array, after middle element.



Binary Search

Search value 5 in list below

0	1	2	3	4	5
1	2	3	4	5	6

1. Middle index = (0 + 5) / 2
   Middle index = 2.5 = 2

0	1	2	3	4	5
1	2	3	4	5	6

Middle element = 3

2. Compare value 5 with middle element 3
   a. 5 > 3

0	1	2	3	4	5
			4	5	6

3	4	5
4	5	6

1. Middle index = (3 + 5) / 2
   Middle index = 4

3	4	5
4	5	6

Middle element = 5

2. Compare value 5 with middle element 5
   a. 5 == 5
   Target is found

Activity

1. Show the procedure to sort the items below using selection sort

40	30	9	20	10	50

2. Show the procedure to sort the items below using selection sort.
   a) show selection sort process in ascending order
   b) show selection sort process in descending order

7	4	5	9	8	2	1

3. Show the procedure to sort items below using bubble sort.

Structure

4. Show the procedure to sort the items below using quick sort.

8	1	5	14	4	15	12	6	2	11	10	7	9

REFERENCES

Abstract data type in data structure - javatpoint. www.javatpoint.com. (n.d.). Retrieved August 10, 2022, from https://www.javatpoint.com/abstract-data-type-in-data-structure.

Data Structures. GeeksforGeeks. (2022). Retrieved August 10, 2022, from https://www.geeksforgeeks.org/data-structures/

Data Structure and algorithms tutorial. Tutorials Point. (n.d.). Retrieved August 10, 2022, from https://www.tutorialspoint.com/data_structures_algorithms/index.htm

Eljinini. M. A. (2019). Practical Data Structures with C++, C#, and Java: in English and Arabic. Jordan. (ISBN: 9957674005)

Roughgarden. T. (2020). Algorithms Illuminated (Part 4): Algorithms for NP-Hard Problems. Soundlikeyourself Publishing, LLC. New York. (ISBN: 0999282964)

Subero. A. (2020). Codeless Data Structures and Algorithms: Learn DSA Without Writing a Single Line of Code 1st ed. Edition. Apress. New York. (ASIN: B084V17V3B)

T.Sheela. (2007). Data Structures. Chennai. (ISBN: 81-87721-88-X)

Wengrow. J. (2020). A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills 2nd Edition. Pragmatic Bookshelf.United State. (ISBN: 1680507222)

POLITEKNIK MALAYSIA
KUALA TERENGGANU

e ISBN 978-967-2240-39-6
9 7 8 9 6 7 2 2 4 0 3 9 6
Data Structures