Stack

Stack

- A stack is also an ordered collection of elements like arrays, but it has a special feature that deletion and insertion of elements can be done only from one end called the top of the stack (TOP)

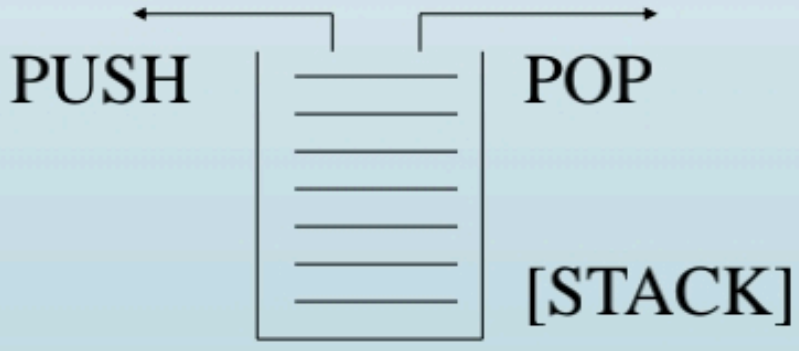
- Due to this property it is also called as last in first out type of data structure (LIFO).

Stack

- It could be through of just like a stack of plates placed on table in a party, a guest always takes off a fresh plate from the top and the new plates are placed on to the stack at the top.
- It is a non-primitive data structure.
- When an element is inserted into a stack or removed from the stack, its base remains fixed where the top of stack changes.

Stack

- Insertion of element into stack is called PUSH and deletion of element from stack is called POP.
- The bellow show figure how the operations take place on a stack:



Objectives

- Use stack to evaluate postfix expression
- Use stack to evaluate prefix expression

Specifications of a Stack

- Organizes entries according to order added
- All additions added to one end of stack
  - Added to “top”
  - Called a “push”
- Access to stack restricted
  - Access only top entry
  - Remove called a “pop”

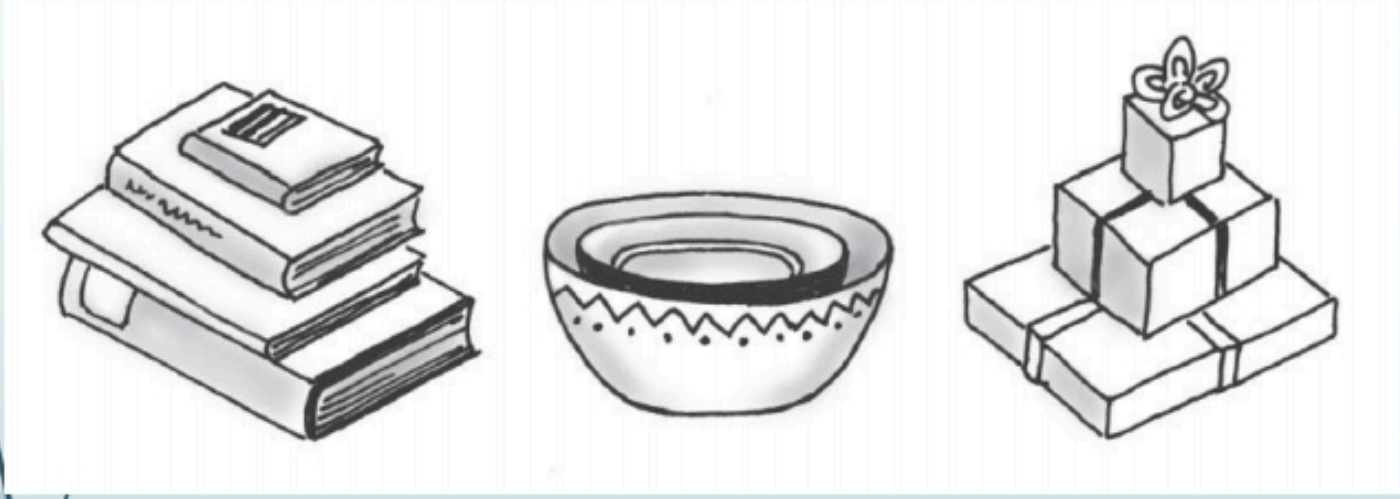


Figure 5-1. Some familiar stacks

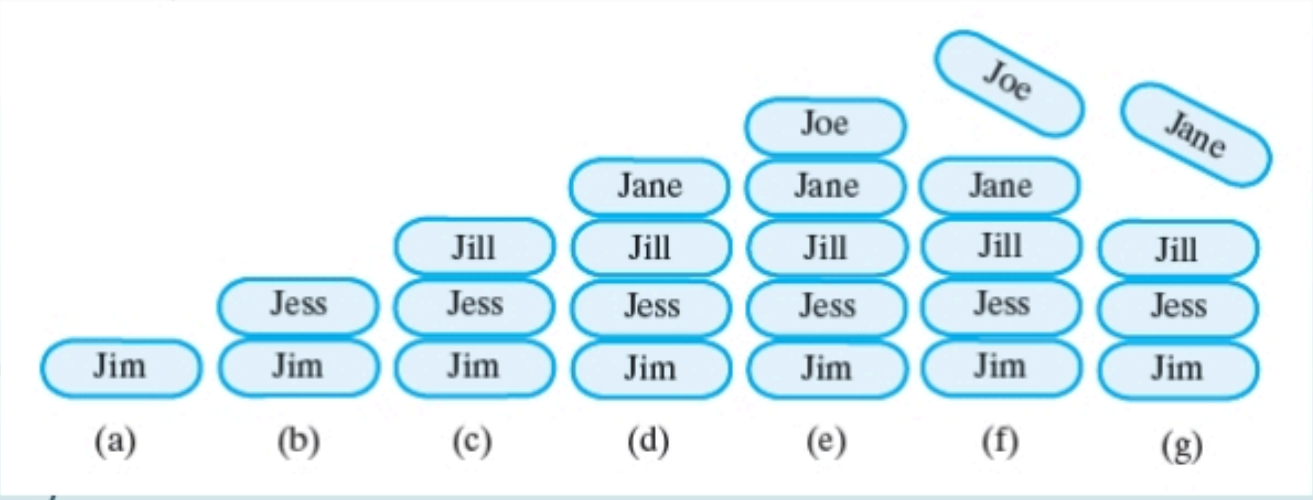


Figure 5-2 A stack of strings after (a) `push` adds Jim; (b) `push` adds Jess; (c) `push` adds Jill; (d) `push` adds Jane; (e) `push` adds Joe; (f) `pop` retrieves and removes Joe; (g) `pop` retrieves and removes Jane

Using a Stack to Process Algebraic Expressions

- Algebraic expressions composed of
  - Operands (variables, constants)
  - Operators (+, -, /, *, ^)
- Operators can be unary or binary
- Different precedence notations
  - Infix a + b
  - Prefix + a b
  - Postfix a b +

Using a Stack to Process Algebraic Expressions

- Precedence must be maintained
  - Order of operators
  - Use of parentheses (must be balanced)
- Use stacks to evaluate parentheses usage
  - Scan expression
  - Push symbols
  - Pop symbols

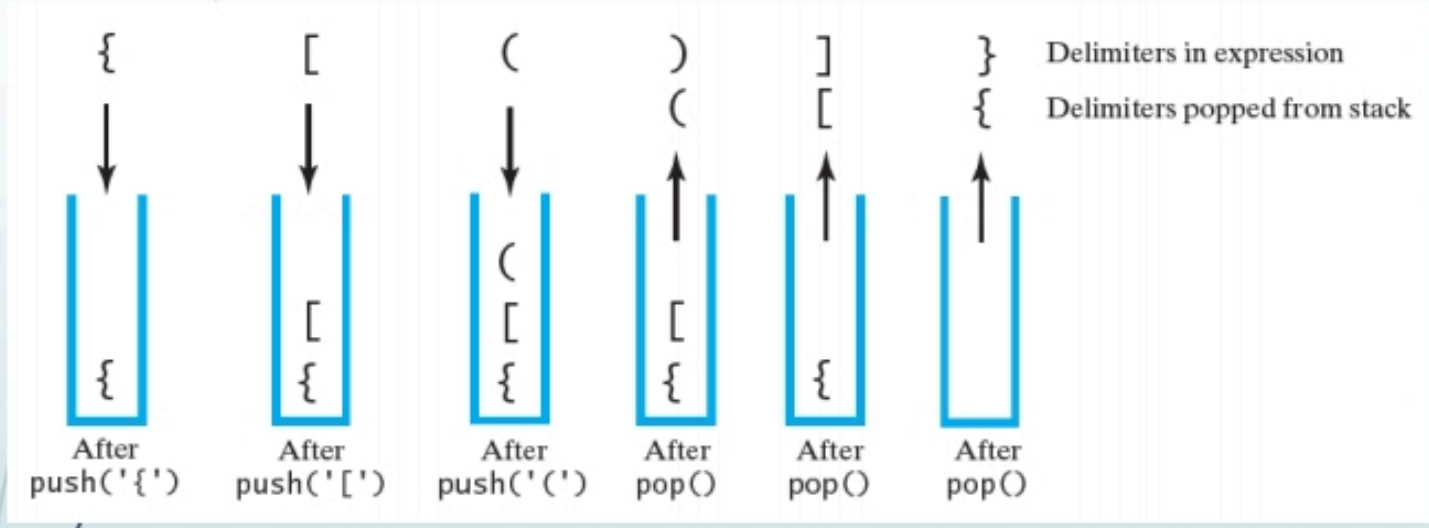


Figure 5-3 The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }

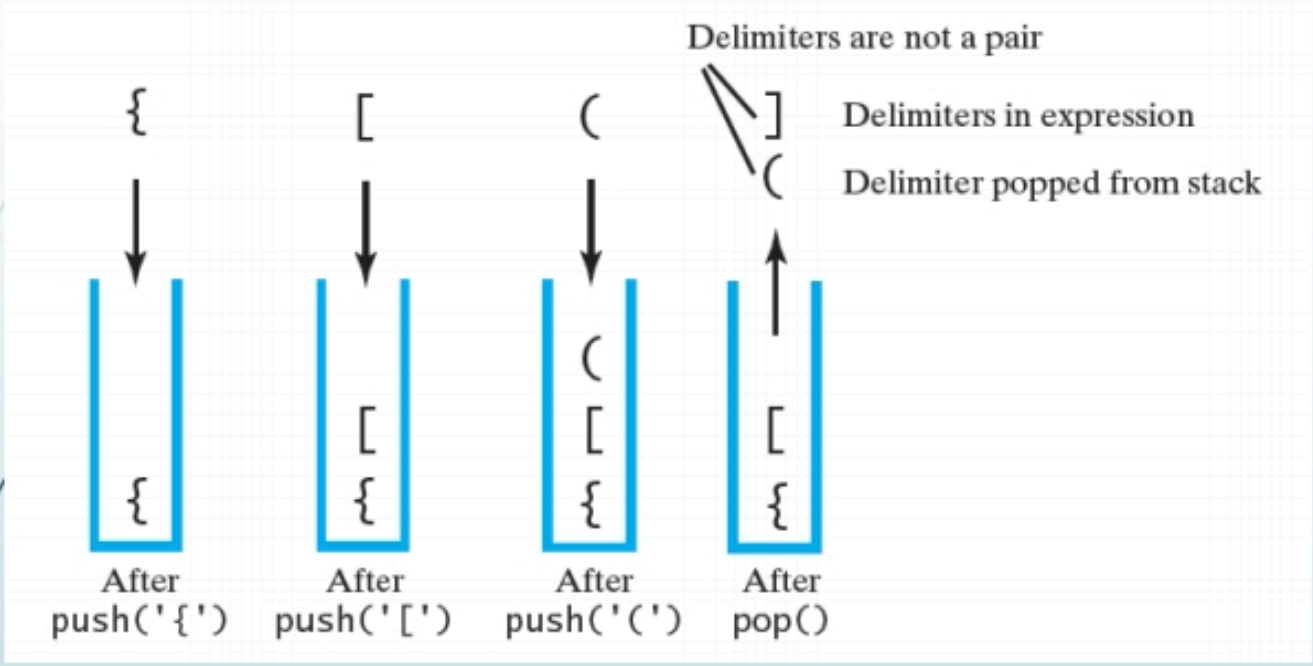


Figure 5-4 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ]) }

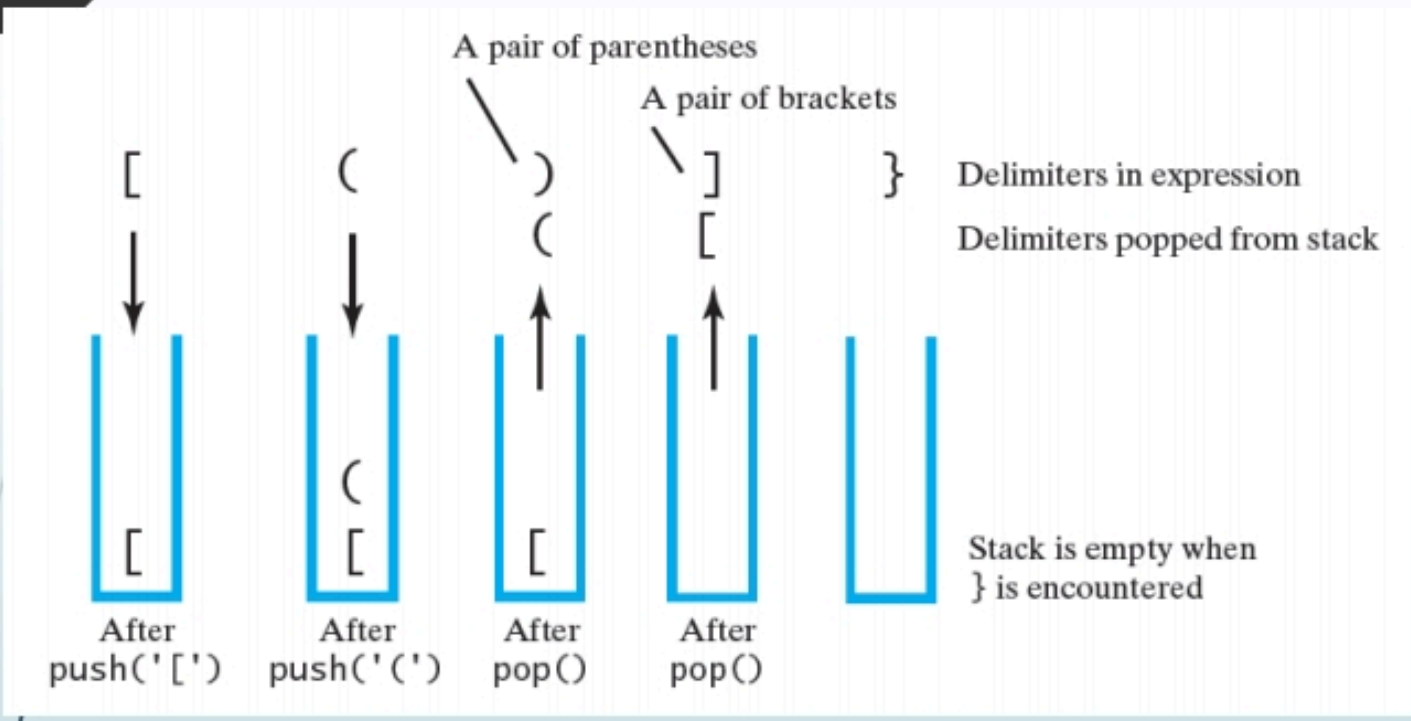


Figure 5-5 The contents of a stack during the scan of an expression that contains the unbalanced delimiters [( )] }

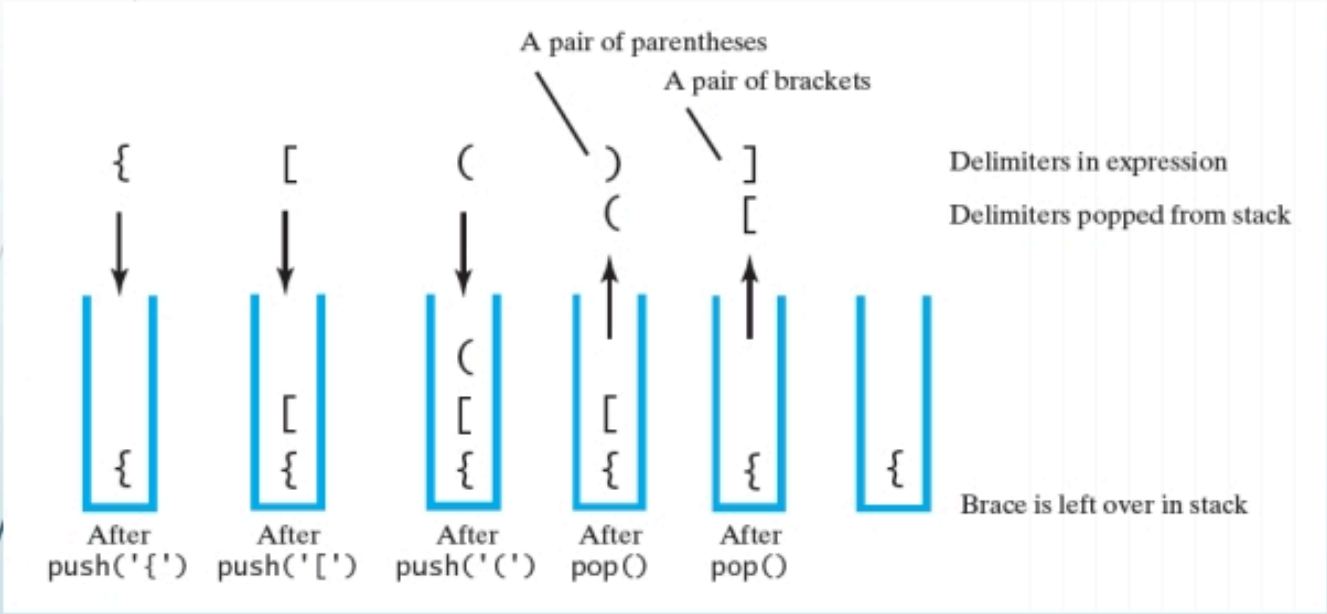


Figure 5-6 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( )]

Infix to Postfix

- Manual algorithm for converting infix to postfix
  (a + b) * c
  - Write with parentheses to force correct operator precedence ((a + b) * c)
  - Move operator to right inside parentheses
    ((a b +) c *)
  - Remove parentheses
    a b + c *

Infix to Postfix

- Algorithm basics
  - Scan expression left to right
  - When operand found, place at end of new expression
  - When operator found, save to determine new position

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Figure 5-7 Converting the infix expression a + b * c to postfix form

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
–	*a*	–
b	*a b*	–
+	*a b* –	
	a b –	+
c	*a b* – *c*	+
	a b – *c* +	

Figure 5-8 Converting an infix expression to postfix form: a - b + c

Infix to Postfix Conversion

1. Operand
   - Append to end of output expression
2. Operator ^
   - Push ^ onto stack
3. Operators +, -, *, /
   - Pop from stack, append to output expression
   - Until stack empty or top operator has lower precedence than new operator
   - Then push new operator onto stack

Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*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 –*	* (+ (
	a b / c d e –	* (+
)	*a b / c d e – +*	* (
	a b / c d e – +	*
	*a b / c d e – + **	

FIGURE 5-9 The steps in converting the infix expression a / b * (c + (d - e)) to postfix form

A*(B+C)

Move	Current token	Output	Stack
1	A	A	empty
2	*	A	*
3	(	A	*(
4	B	A B	*(
5	+	A B	*(+
6	C	A B C	*(+
7	)	A B C +	*
		ABC+*	empty

A+B+C

Move	Current token	Output	Stack
1	A	A	empty
2	+	A	+
3	B	AB	+
4	+	A B+	+
5	C	A B+C	+
6		A B+C+	empty

A-B+C

Move	Current token	Output	Stack
1	A	A	empty
2	-	A	-
3	B	AB	-
4	+	A B-	+
5	C	A B-C	+
6		A B-C+	empty

(A+B)*(C-D)

Move	Current token	Output	Stack
1	(		(
1	A	A	(
2	+	A	(+
3	B	AB	(+
4	)	A B+	
5	*	A B+	*
6	(	A B+	*(
7	C	AB+C	*(
8	-	AB+C	*(-
9	D	AB+CD	*(-
10	)	AB+CD-	*
11		**AB+CD-***	empty

A+B*C/(E-F)

Move	Current token	Output	Stack
1	A	A	Empty
2	+	A	+
3	B	AB	+
4	*	A B	+*
5	C	A BC	+*
6	/	A B C*	+/
7	(	A B C *	+/(
8	E	ABC*E	+/(
9	-	ABC*E	+/(-
10	F	ABC*EF	+/(-
11	)	ABC*EF-	+/
13		**ABC*EF-/+**	empty

Problems

1. (A*B)+(C-D)/(E+F)
2. C-D/(E+F)
3. A*B-C/(E+F)
4. (A/(B-C)*D+E)
5. A+B*C/D-F+A^E
6. A*(B+C*D)
7. A-(B+C*D)/E

Convert Infix Expression into Prefix Expression

Consider an infix expression: (A – B / C) * (A / K – L)

- **Step 1:** Reverse the infix string. Note that while reversing the string you must interchange left and right parenthesis.
  (L – K / A) * (C / B – A)
- **Step 2:** Obtain the corresponding postfix expression of the infix expression obtained as a result of Step 1.
- The expression is: (L – K / A) * (C / B – A)
- Therefore, [L – (K A /)] * [(C B /) - A]
  = [LKA/-] * [CB/A-]
  = L K A / - C B / A - *
- **Step 3:** Reverse the postfix expression to get the prefix expression
- Therefore, the prefix expression is * - A / B C - / A K L

Convert Postfix Expression into Infix Expression

- **1.** While there are input symbol left
- **2.** Read the next symbol from input.
- **3.** If the symbol is an operand

  Push it onto the stack.
- **4.** Otherwise,

  the symbol is an operator.
- **5.** If there are fewer than 2 values on the stack

  Show Error /* input not sufficient values in the expression */

Convert Postfix Expression into Infix Expression

- **6.** Else

  Pop the top 2 values from the stack.

  Put the operator, with the values as arguments and form a string.

  Encapsulate the resulted string with parenthesis.

  Push the resulted string back to stack.

- **7.** If there is only one value in the stack

  That value in the stack is the desired infix string.

- **8.** If there are more values in the stack

Convert Prefix Expression into Infix Expression

- **1.** The reversed input string is completely pushed into a stack.

  prefixToInfix(stack)

- **2.** IF stack is not empty
- **a.** Temp -->pop the stack
- **b.** IF temp is a operator

  Write a opening parenthesis to output

  prefixToInfix(stack)

  Write temp to output

  prefixToInfix(stack)

  Write a closing parenthesis to output

Convert Prefix Expression into Infix Expression

c. ELSE IF temp is a space -->prefixToInfix(stack)

d. ELSE

Write temp to output

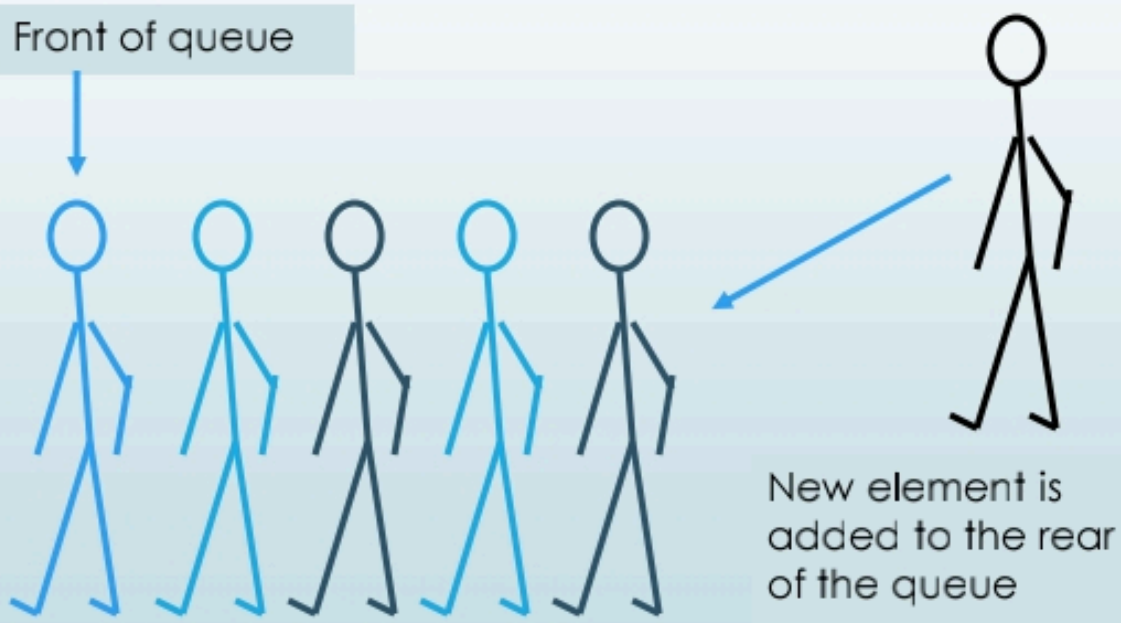
IF stack.top NOT EQUAL to space -->prefixToInfix(stack)

QUEUE

- Queue is the **linear data structure** type which is used to organize the data.
- It is used for temporary storage of data values.
- A new element is added at one end called **rear end**
- The existing element deleted from the other end is called **front end**.
- **First-in-First-out** property.

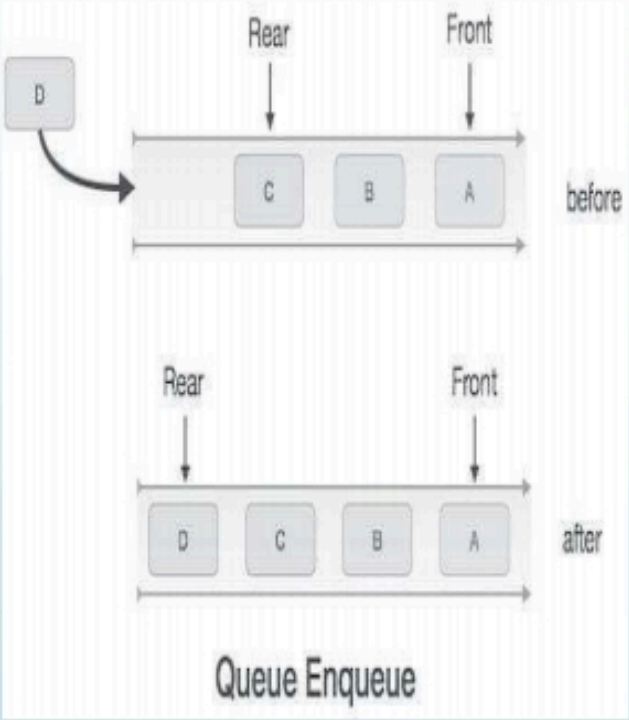
Conceptual View of a Queue

Adding an element (Enqueue)



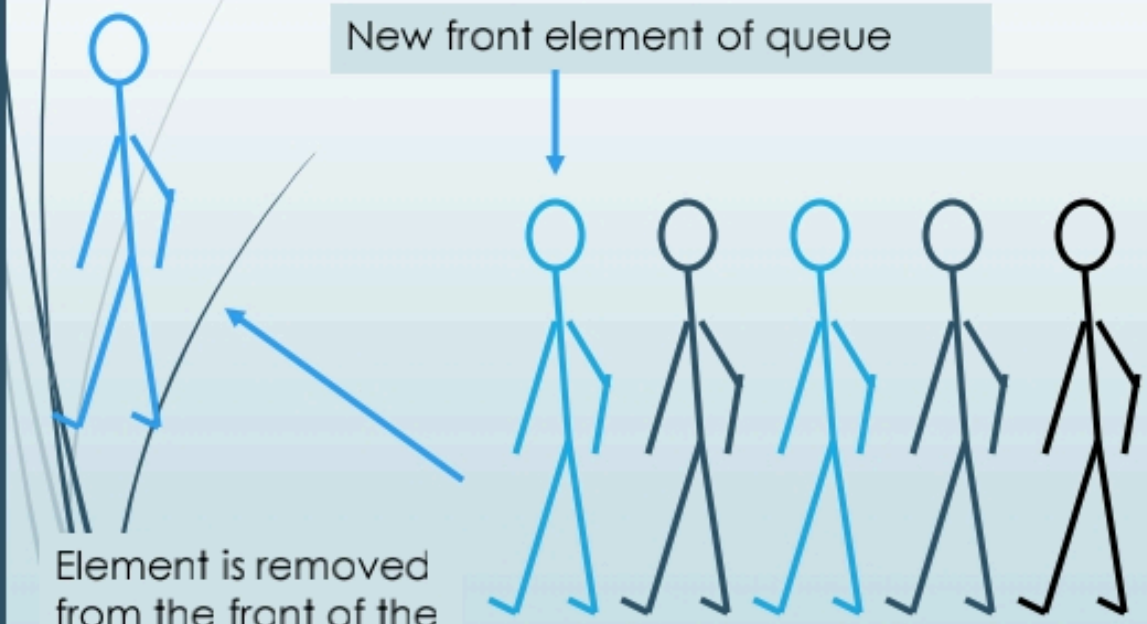
Enqueue Operation

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



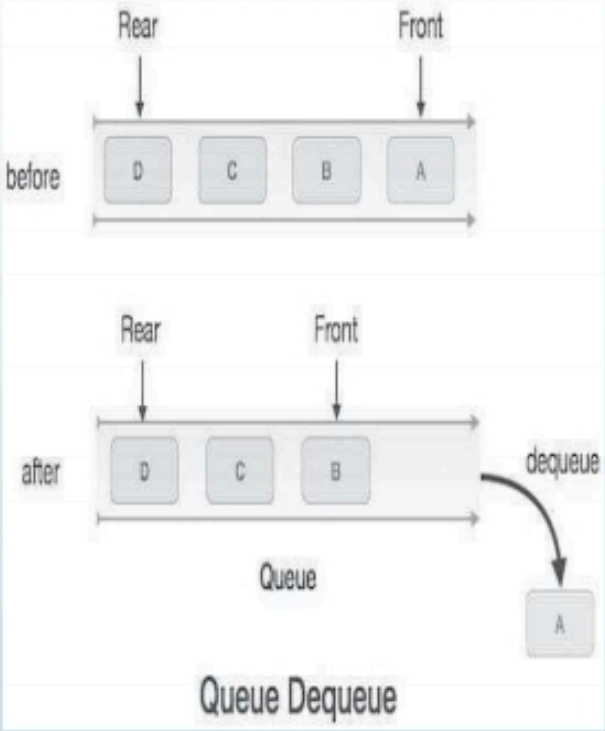
Conceptual View of a Queue

Removing an element (Dequeue)



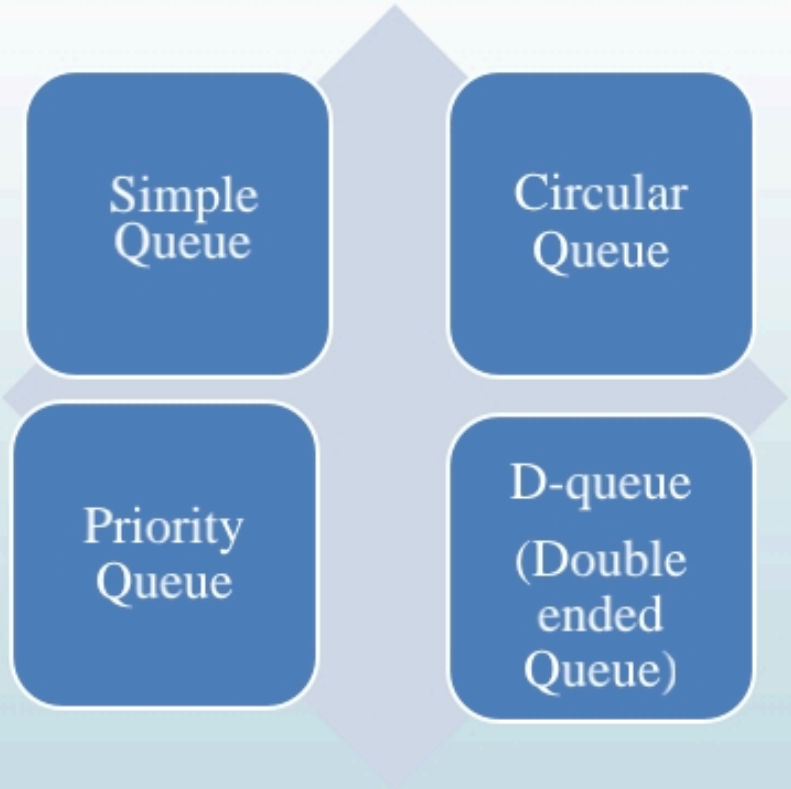
Dequeue Operation

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



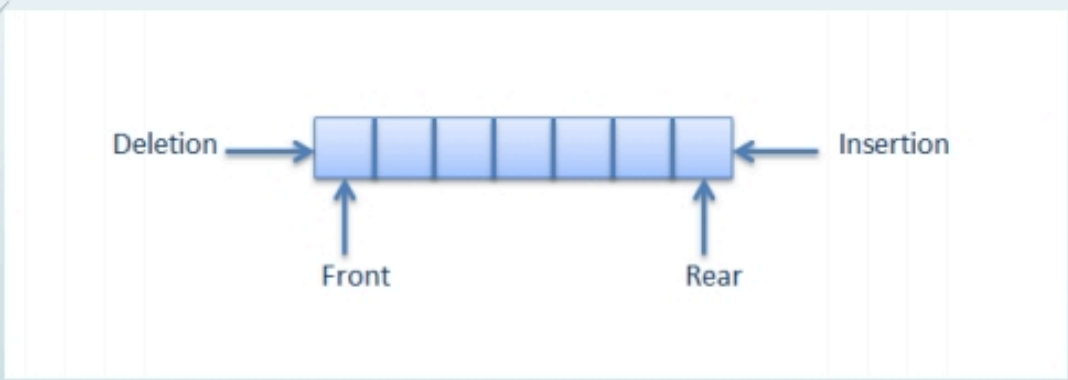
Queue Dequeue

Type of queue



Simple Queue

- Simple queue defines the simple operation of queue in which insertion occurs at the rear of the list and deletion occurs at the front of the list.



Circular Queue

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as **Ring Buffer.**
- It is an abstract data type.
- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue

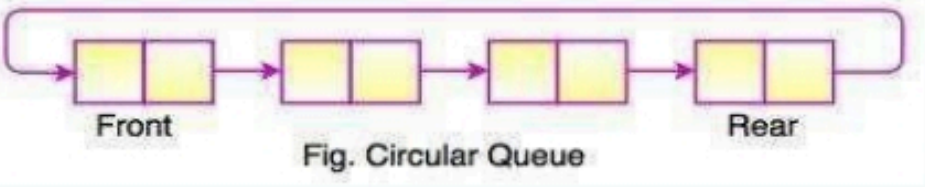


Fig. Circular Queue

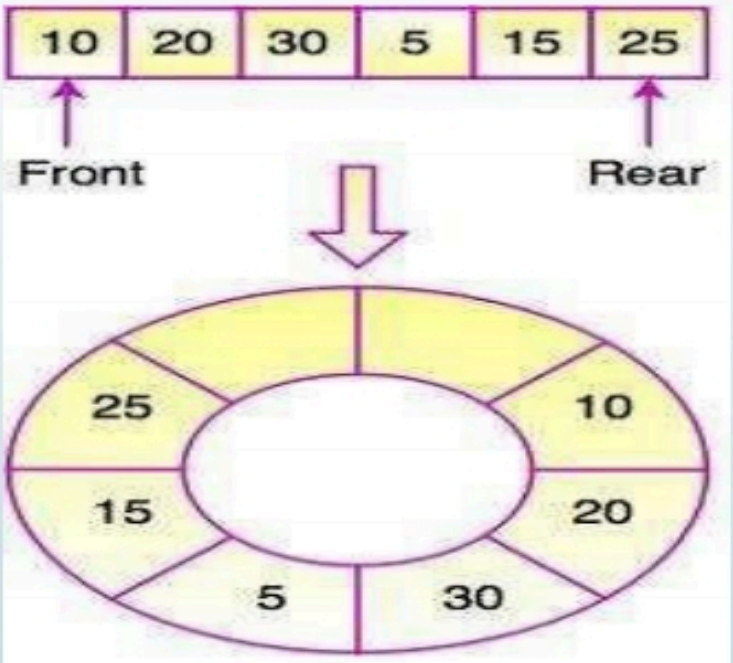
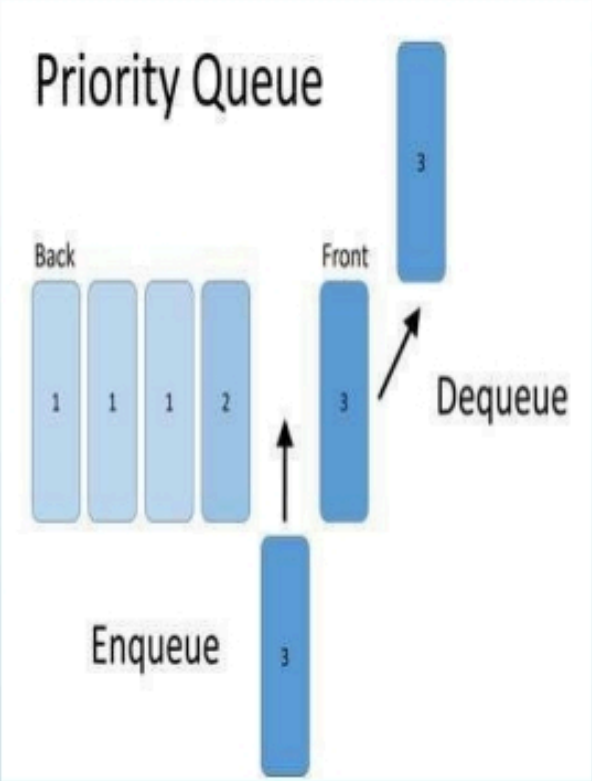


Fig. Circular Queue

Priority Queue

- Priority queue contains data items which have some preset priority. While removing an element from a priority queue, the data item with the highest priority is removed first.
- In a priority queue, insertion is performed in the order of arrival and deletion is performed based on the priority.



D-queue (Double ended Queue)

- In Double Ended Queue, insert and delete operation can be occur at both ends that is front and rear of the queue

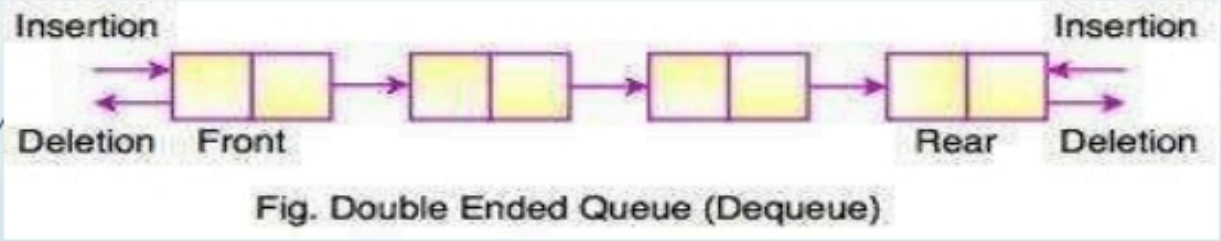


Fig. Double Ended Queue (Dequeue)

Operations on Queue

- **Enqueue()** – add (store) an item to the queue.
- **Dequeue()** – remove (access) an item from the queue.
- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty

LINEAR QUEUE Vs CIRCULAR QUEUE

- A linear data structure that stores data as a sequence of element similar to a real world queue.
- Possible to enter new items from the rear end and remove the items from the front.
- Requires more memory.
- Less efficient.

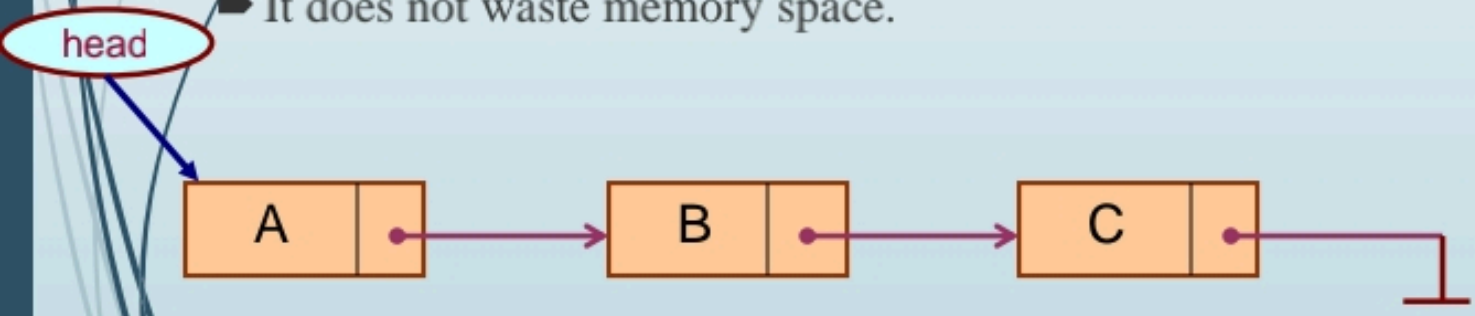
- A linear data structure in which the last item connects back to the first item forming a circle.
- Possible to enter and remove elements from any position.
- Requires less memory.
- More efficient.

Application of Queue

- Queue is useful in CPU scheduling, Disk scheduling.
- When data is transferred asynchronously between two processes. Queue is used for synchronization. Examples : IO Buffers, pipes, file IO, etc.
- In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

Linked List

- A linked list is a data structure which can change during execution.
  - Successive elements are connected by pointers.
  - Last element points to NULL.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.



Linked List

- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.).
- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.

Illustration: Insertion

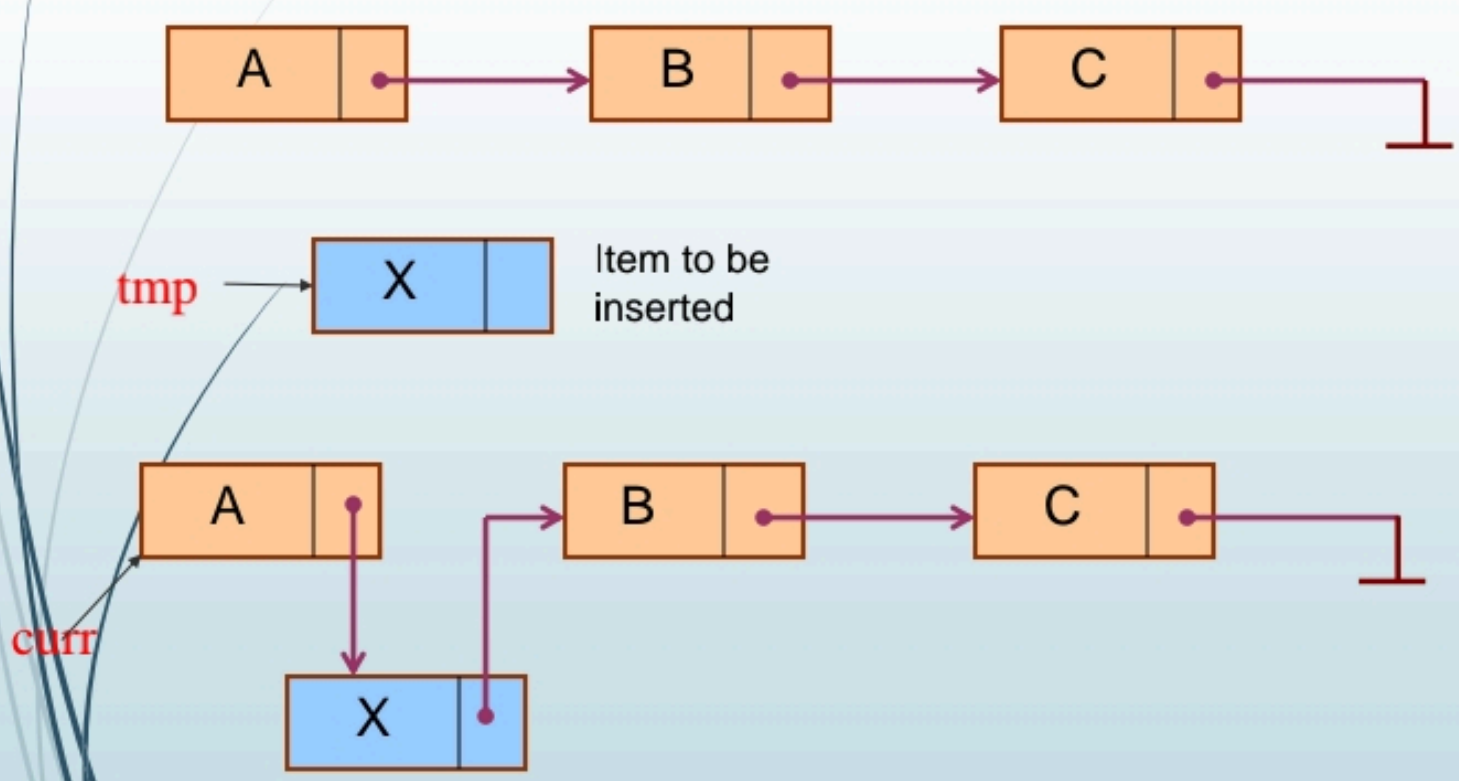
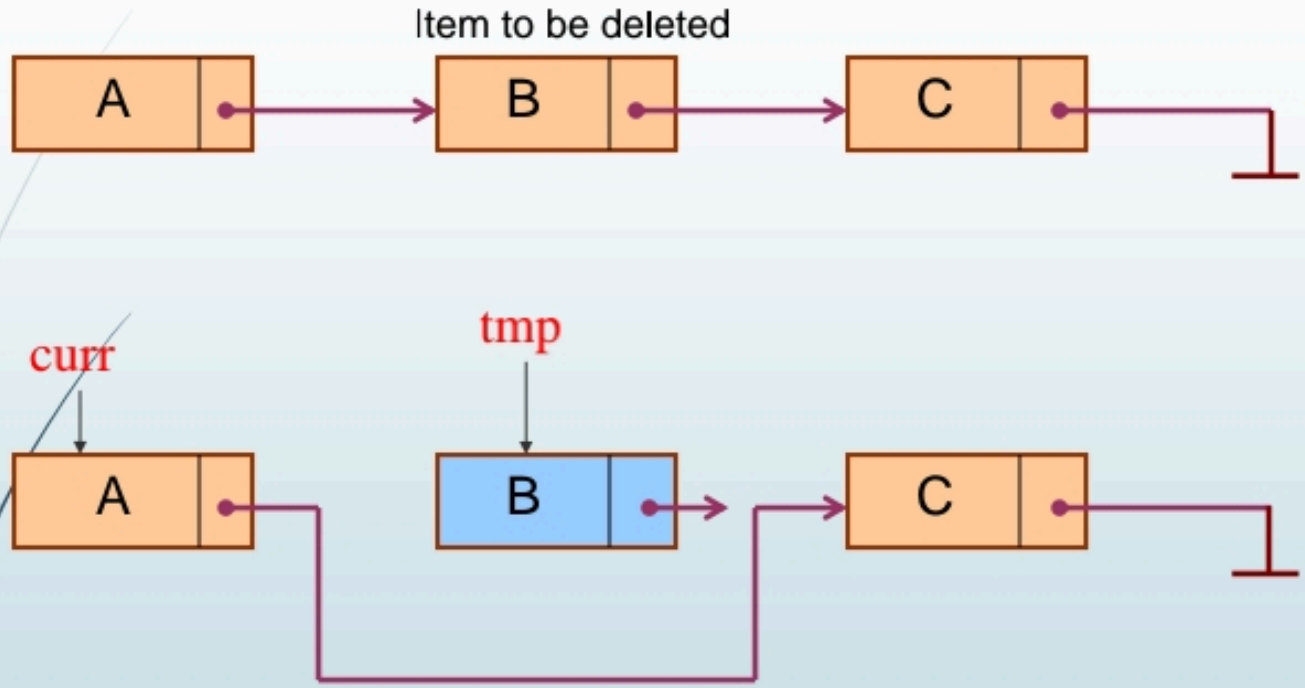


Illustration: Deletion



In essence ...

- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

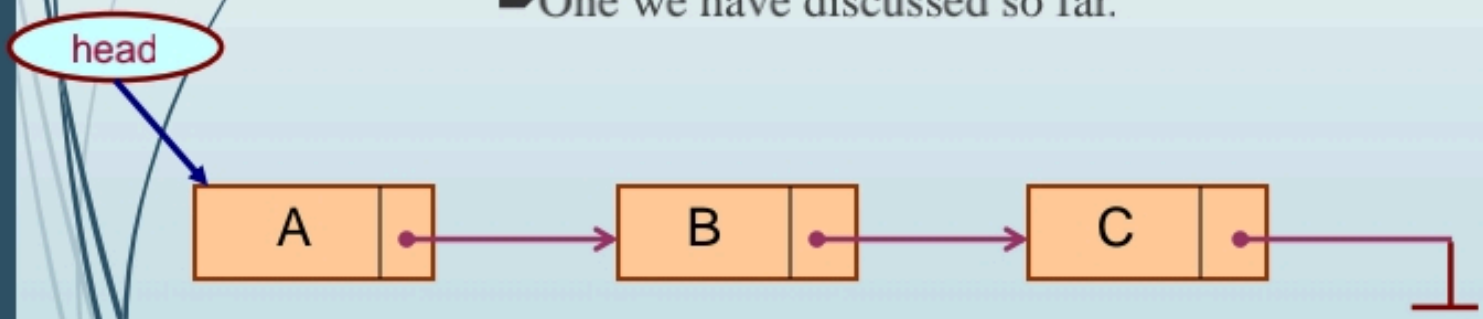
Array versus Linked Lists

- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.

- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

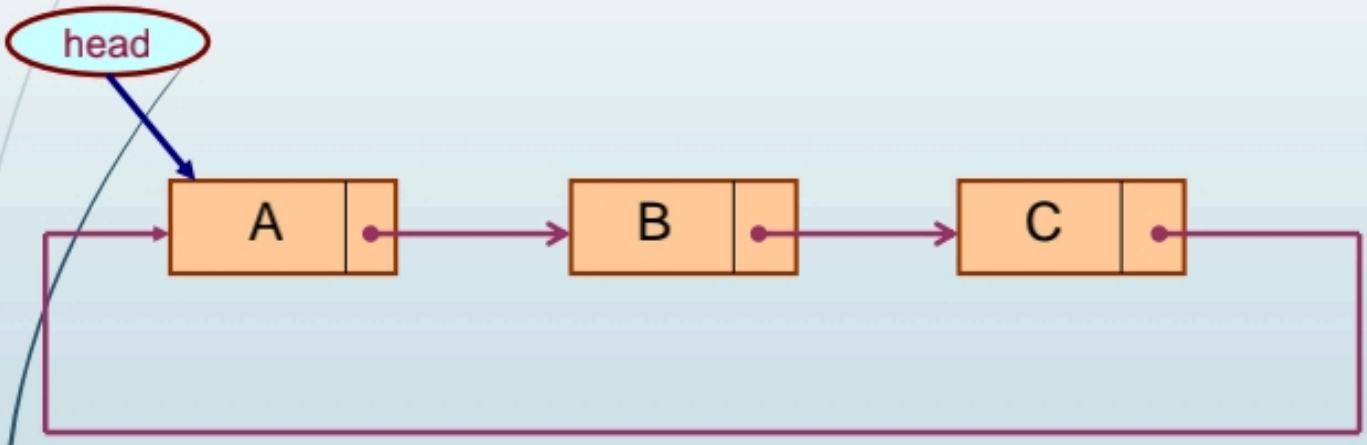
Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
  - Linear singly-linked list (or simply linear list)
    - One we have discussed so far.



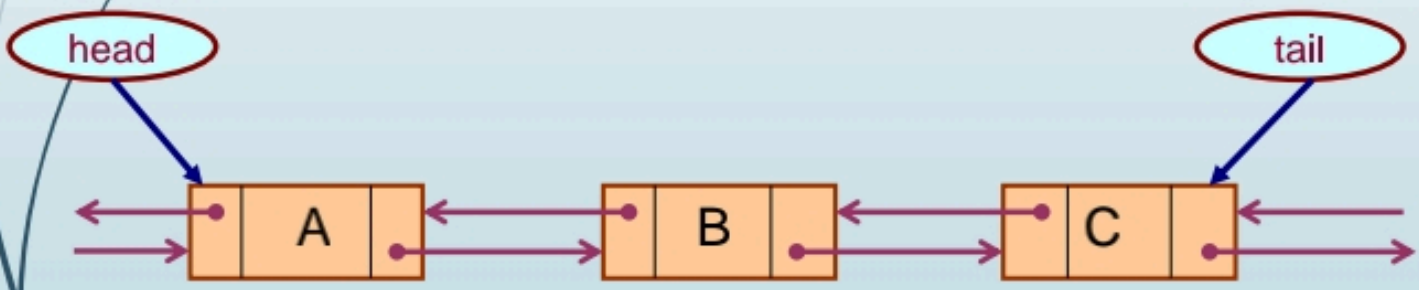
Circular linked list

- The pointer from the last element in the list points back to the first element.



Doubly linked list

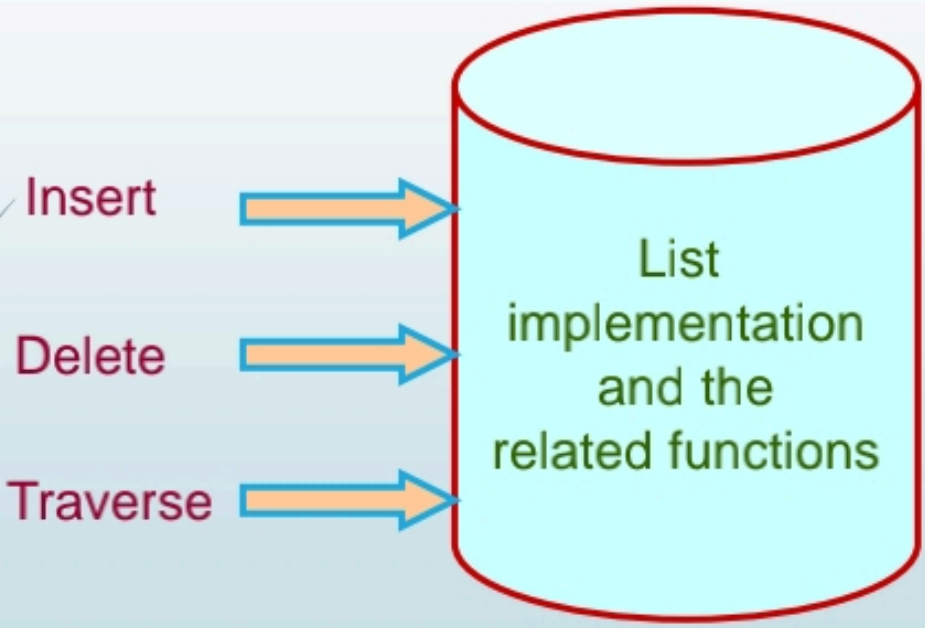
- Doubly linked list
  - Pointers exist between adjacent nodes in both directions.
  - The list can be traversed either forward or backward.
  - Usually two pointers are maintained to keep track of the list, *head* and *tail*.



Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

Conceptual Idea

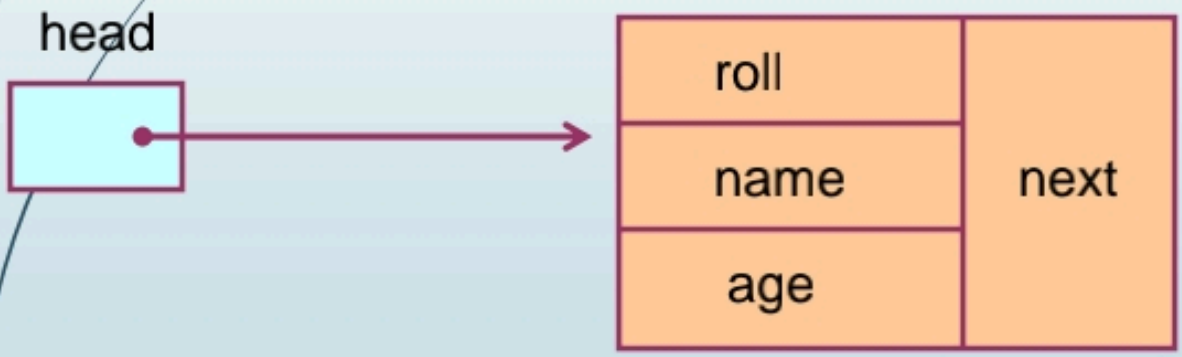


Creating a List

How to begin?

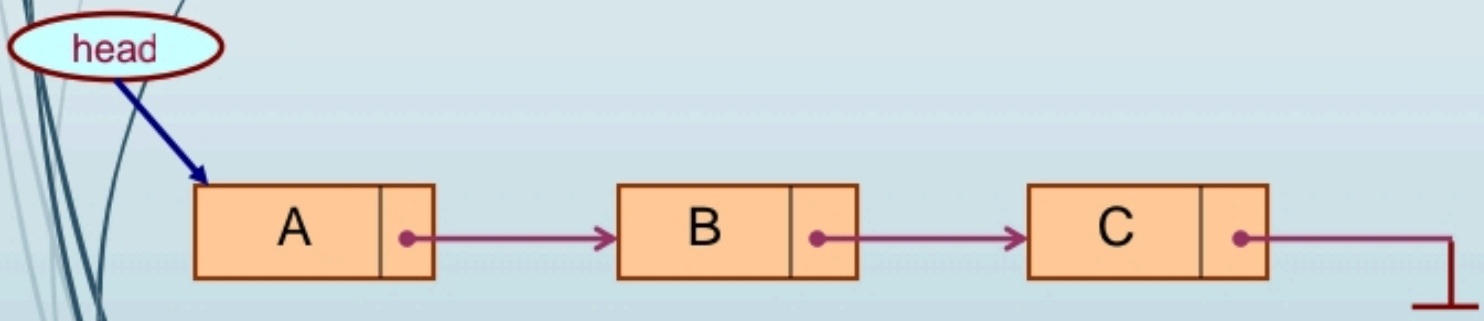
- To start with, we have to create a node (the first node), and make head point to it.

```
head = (node *) malloc(sizeof(node));
```



Contd.

- If there are n number of nodes in the initial linked list:
  - Allocate n records, one by one.
  - Read in the fields of the records.
  - Modify the links of the records so that the chain is formed.



Traversing the List

What is to be done?

- Once the linked list has been constructed and *head* points to the first node of the list,
  - Follow the pointers.
  - Display the contents of the nodes as they are traversed.
  - Stop when the *next* pointer points to NULL.

Inserting a Node in a List

How to do?

- The problem is to insert a node *before a specified node*.
  - Specified means some value is given for the node (called *key*).
  - In this example, we consider it to be `roll`.
- Convention followed:
  - If the value of roll is given as *negative*, the node will be inserted at the *end* of the list.

Contd.

- When a node is added at the beginning,
  - Only one next pointer needs to be modified.
    - *head* is made to point to the new node.
    - New node points to the previously first element.
- When a node is added at the end,
  - Two next pointers need to be modified.
    - Last node now points to the new node.
    - New node points to NULL.
- When a node is added in the middle,
  - Two next pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.

Deleting a node from the list

What is to be done?

- Here also we are required to delete a specified node.
  - Say, the node whose roll field is given.

- Here also three conditions arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.