DS-Unit-2

24 May 2025 06:46

Stack in Data Structure

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle.
Stack has one end, whereas the Queue has two ends (**front and rear**).
It contains only one pointer **top pointer** pointing to the topmost element of the stack.
Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.
In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

- A stack can be implemented using an array or a linked list data structure
- For an array implementation, we need to define a fixed size for the stack initially
- We maintain a 'top' variable that keeps track of the index of the topmost element in the stack
- Initially, 'top' is set to -1, indicating an empty stack
- The following operations can be performed on a stack:
  - **Push Operation:**
    - Check if the stack is full (top == size - 1)
    - If full, return an error (stack overflow)
    - If not full, increment 'top' and insert the new element at the new 'top' index
  - **Pop Operation:**
    - Check if the stack is empty (top == -1)
    - If empty, return an error (stack underflow)
    - If not empty, retrieve the element at the 'top' index
    - Decrement 'top' to remove the topmost element
  - **Peek Operation:**
    - Check if the stack is empty (top == -1)
    - If empty, return an error
    - If not empty, return the element at the 'top' index (without removing it)
  - **isEmpty Operation:**
    - Check if 'top' is equal to -1
    - If yes, return true (stack is empty)
    - If no, return false (stack is not empty)
  - **isFull Operation:**
    - Check if 'top' is equal to size - 1
    - If yes, return true (stack is full)
    - If no, return false (stack is not full)

- **size():** Returns the number of elements currently present in the stack.

- **top():** Accesses and returns the value of the topmost element without removing it, similar to peek(). An alias for peek().
- **display():** Outputs all elements in the stack, typically by traversing from top to bottom.

- **clear():** Removes all elements from the stack, resetting it to an empty state.
- **reverse():** Reverses the order of elements in the stack.
- **contains(item):** Checks if the given 'item' exists in the stack, returning true or false.

Applications of Stack

The following are the applications of the stacks

Balancing of symbols
String reversal
UNDO/REDO
Recursion
DFS(Depth First Search)
Backtracking:
Expression conversion
Memory management

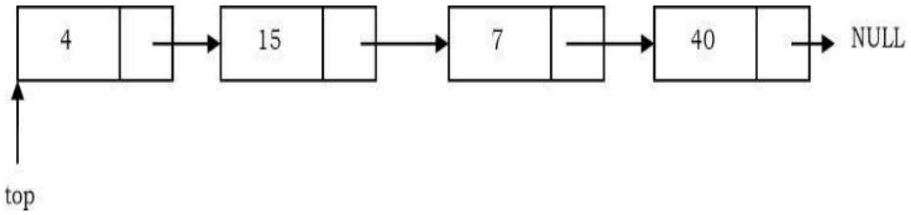
Simple Array Implementation

This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the top element.

The array storing the stack elements may become full. A push operation will then throw a *full stack exception*. Similarly, if we try deleting an element from an empty stack it will throw *stack empty exception.*

Screen clipping taken: 24-05-2025 07:25

Linked List Implementation



The other way of implementing stacks is by using Linked lists. Push operation is implemented by inserting element at the beginning of the list. Pop operation is implemented by deleting the node from the beginning (the header/top node).

Screen clipping taken: 24-05-2025 08:14

Array Implementation

- Operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of n operations (starting from empty stack) – *"amortized"* bound takes time proportional to n.

Linked List Implementation

- Grows and shrinks gracefully.
- Every operation takes constant time O(1).
- Every operation uses extra space and time to deal with references.

Screen clipping taken: 24-05-2025 08:15

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.
    1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
    2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.
4. Copy the head pointer into a temporary pointer.
5. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Following is the various Applications of Stack in Data Structure:

- Evaluation of Arithmetic Expressions
- Backtracking

- Delimiter Checking
- Reverse a Data
- Processing Function Calls

From <https://www.tpointtech.com/application-of-stack-in-data-structure>

==========

Queue in Data Structure

Definition: A queue is an ordered list in which insertions are done at one end (rear) and deletions are done at other end (front).
The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list.
Similar to Stacks, special names are given to the two changes that can be made to a queue.
When an element is inserted in a queue, the concept is called EnQueue, and when an element is removed from the queue, the concept is called DeQueue

DeQueueing an empty queue is called underflow and EnQueuing an element in a full queue is called overflow. Generally, we treat them as exceptions

3 Queue ADT
The following operations make a queue an ADT. Insertions and deletions in the queue must
follow the FIFO scheme. For simplicity we assume the elements are integers.

Main Queue Operations
• EnQueue(int data): Inserts an element at the end of the queue
• int DeQueue(): Removes and returns the element at the front of the queue

Auxiliary Queue Operations
• int Front(): Returns the element at the front without removing it
• int QueueSize(): Returns the number of elements stored in the queue
• int IsEmptyQueueQ: Indicates whether no elements are stored in the queue or not

Applications
Following are some of the applications that use queues.

Direct Applications
• Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
• Simulation of real-world queues such as lines at a ticket counter or any other first come first-served scenario requires a queue.
• Multiprogramming.
• Asynchronous data transfer (file IO, pipes, sockets).
• Waiting times of customers at call center.
• Determining number of cashiers to have at a supermarket.

Indirect Applications
• Auxiliary data structure for algorithms
• Component of other data structures

Implementation
There are many ways (similar to Stacks) of implementing queue operations and some of the commonly used methods are listed below.
• Simple circular array based implementation
• Dynamic circular array based implementation
• Linked list implementation

Implementation of Queue using Array:

==========

Hashing

A hashing algorithm is used to convert an input (such as a string or integer) into a fixed-size output (referred to as a hash code or hash value).
The data is then stored and retrieved using this hash value as an index in an array or hash table.
Hashing is commonly used to create a unique identifier for a piece of data, which can be used to quickly look up that data in a large dataset.

What is a hash Key?

In the context of hashing, a hash key (also known as a hash value or hash code) is a fixed-size numerical or alphanumeric representation generated by a hashing algorithm.

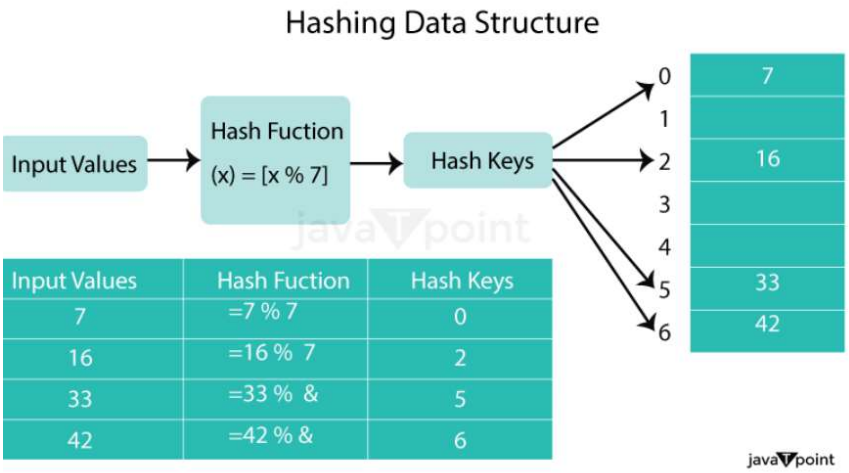
How Hashing Works?

The process of hashing can be broken down into three steps:

- Input: The data to be hashed is input into the hashing algorithm.
- Hash Function: The hashing algorithm takes the input data and applies a mathematical function to generate a fixed-size hash value.
- Output: The hash value is returned, which is used as an index to store or retrieve data in a data structure.

Hashing Algorithms:

- MD5: A widely used hashing algorithm that produces a 128-bit hash value.
- SHA-1: A popular hashing algorithm that produces a 160-bit hash value.
- SHA-256: A more secure hashing algorithm that produces a 256-bit hash value.



Input Values	Hash Fuction	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % &	5
42	=42 % &	6

Screen clipping taken: 07-06-2025 07:32

Hash Function types

1. Division Method
2. Multiplication Method
3. Universal hashing

- **Division method:**
  This method involves dividing the key by the table size and taking the remainder as the hash value. For example, if the table size is 10 and the key is 23, the hash value would be 3 (23 % 10 = 3).
- **Multiplication method:**
  This method involves multiplying the key by a constant and taking the fractional part of the product as the hash value. For example, if the key is 23 and the constant is 0.618, the hash value would be 2 (floor(10*(0.61823 - floor(0.61823))) = floor(2.236) = 2).
- **Universal hashing:**
  This method involves using a random hash function from a family of hash functions. This ensures that the hash function is not biased towards any particular input and is resistant to attacks

Collision Resolution

One of the main challenges in hashing is handling collisions, which occur when two or more input values produce the same hash value.
1.chaining
2.open addressing
3. Double hashing

- Chaining: In this technique, each hash table slot contains a linked list of all the values that have the same hash value. This technique is simple and easy to implement, but it can lead to poor performance when the linked lists become too long.
- Open addressing: In this technique, when a collision occurs, the algorithm searches for an empty slot in the hash table by probing successive slots until an empty slot is found. This technique can be more efficient than chaining when the load factor is low, but it can lead to clustering and poor performance when the load factor is high.
- Double hashing: This is a variation of open addressing that uses a second hash function to determine the next slot to probe when a collision occurs. This technique can help to reduce clustering and improve performance.

From <https://www.tpointtech.com/hashing-in-data-structure>

Hash Table:

A hash table is a data structure that stores data in an array.Typically, a size for the array is selected that is greater than the number of elements that can fit in the hash table. A key is mapped to an index in the array using the hash function.

The hash function is used to locate the index where an element needs to be inserted in the hash table in order to add a new element. The element gets added to that index if there isn't a collision.If there is a collision, the collision resolution method is used to find the next available slot in the array.

The hash function is used to locate the index that the element is stored in order to retrieve it from the hash table. If the element is not found at that index, the collision resolution method is used to search for the element in the linked list (if chaining is used) or in the next available slot (if open addressing is used).

Hash Table Operations

There are several operations that can be performed on a hash table, including:

- Insertion: Inserting a new key-value pair into the hash table.
- Deletion: Removing a key-value pair from the hash table.
- Search: Searching for a key-value pair in the hash table.

Applications of Hashing

Hashing has many applications in computer science, including:

- Databases: Hashing is used to index and search large databases efficiently.
- Cryptography: Hash functions are used to generate message digests, which are used to verify the integrity of data and protect against tampering.
- Caching: Hash tables are used in caching systems to store frequently accessed data and improve performance.
- Spell checking: Hashing is used in spell checkers to quickly search for words in a dictionary.
- Network routing: Hashing is used in load balancing and routing algorithms to distribute network traffic across multiple servers.

Advantages of Hashing:

- Fast Access: Hashing provides constant time access to data, making it faster than other data structures like linked lists and arrays.
- Efficient Search: Hashing allows for quick search operations, making it an ideal data structure for applications that require frequent search operations.
- Space-Efficient: Hashing can be more space-efficient than other data structures, as it only requires a fixed amount of memory to store the hash table.

Hashing

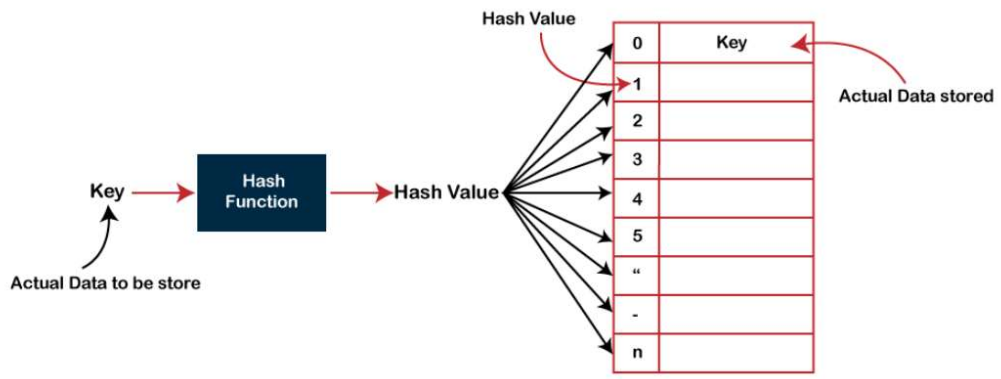
Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1)

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:

From <https://www.tpointtech.com/hash-table>

Index = hash(key)



There are three ways of calculating the hash function:

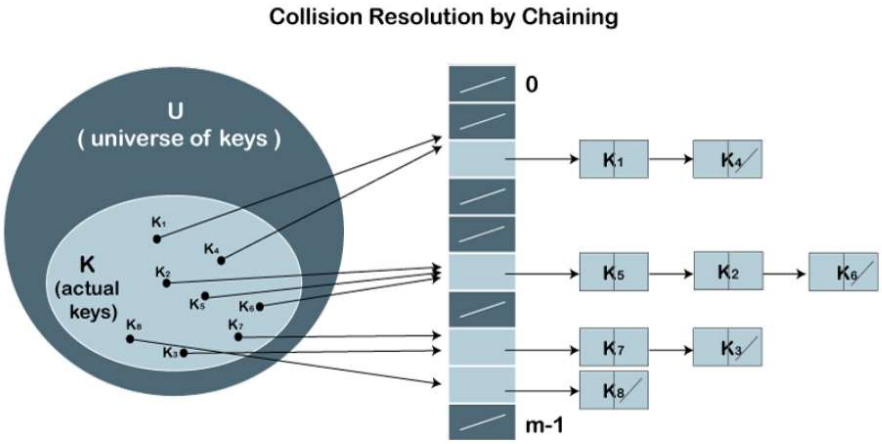
- **Division method**
- **Folding method**
- **Mid square method**

The following are the collision techniques:

- Open Hashing: It is also known as closed addressing.
- Closed Hashing: It is also known as open addressing.

Open Hashing

In Open Hashing, one of the methods used to resolve the collision is known as a chaining method.



Let's first understand the chaining to resolve the collision.

Suppose we have a list of key values

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

In this case, we cannot directly use $h(k) = k_i/m$ as $h(k) = 2k+3$

- The index of key value 3 is:
  index = h(3) = (2(3)+3)%10 = 9
  The value 3 would be stored at the index 9.
- The index of key value 2 is:
  index = h(2) = (2(2)+3)%10 = 7
  The value 2 would be stored at the index 7.
- The index of key value 9 is:
  index = h(9) = (2(9)+3)%10 = 1
  The value 9 would be stored at the index 1.
- The index of key value 6 is:
  index = h(6) = (2(6)+3)%10 = 5
  The value 6 would be stored at the index 5.
- The index of key value 11 is:
  index = h(11) = (2(11)+3)%10 = 5
  The value 11 would be stored at the index 5. Now, we have two values (6, 11) stored at the same index, i.e., 5. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 11 to this list. After the creation of the new list, the newly created list will be linked to the list having value 6.
- The index of key value 13 is:
  index = h(13) = (2(13)+3)%10 = 9
  The value 13 would be stored at index 9. Now, we have two values (3, 13) stored at the same index, i.e., 9. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 13 to this list. After the creation of the new list, the newly created list will be linked to the list having value 3.
- The index of key value 7 is:
  index = h(7) = (2(7)+3)%10 = 7
  The value 7 would be stored at index 7. Now, we have two values (2, 7) stored at the same index, i.e., 7. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 7 to this list. After the creation of the new list, the newly created list will be linked to the list having value 2.
- The index of key value 12 is:
  index = h(12) = (2(12)+3)%10 = 7
  According to the above calculation, the value 12 must be stored at index 7, but the value 2 exists at index 7. So, we will create a new list and add 12 to the list. The newly created list will be linked to the list having a value 7.

From <https://www.tpointtech.com/hash-table>

The calculated index value associated with each key value is shown in the below table:

key	Location(u)
3	((2*3)+3)%10 = 9
2	((2*2)+3)%10 = 7
9	((2*9)+3)%10 = 1
6	((2*6)+3)%10 = 5
11	((2*11)+3)%10 = 5
13	((2*13)+3)%10 = 9
7	((2*7)+3)%10 = 7
12	((2*12)+3)%10 = 7

From <https://www.tpointtech.com/hash-table>

Closed Hashing

In Closed hashing, three techniques are used to resolve the collision:

1. Linear probing
2. Quadratic probing
3. Double Hashing technique

Linear Probing

Linear probing is one of the forms of open addressing.
As we know that each cell in the hash table contains a key-value pair,
so when the collision occurs by mapping a new key to the cell already occupied by another key,
then linear probing technique searches for the closest free locations and adds a new key to that empty cell.
In this case, searching is performed sequentially, starting from the position where the collision occurs till the empty cell is not found.

Let's understand the linear probing through an example.

Consider the above example for the linear probing:
A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

From <https://www.tpointtech.com/hash-table>

Quadratic Probing

quadratic probing is an open addressing technique that uses quadratic polynomial for searching until a empty slot is found.

It can also be defined as that it allows the insertion ki at first free location from **$(u+i_2)\%m$ where i=0 to m-1.**

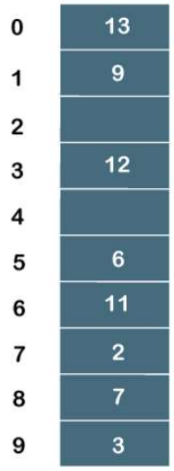
A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3
The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5, respectively. We do not need to apply the quadratic probing technique on these key values as there is no occurrence of the collision.
The index value of 11 is 5, but this location is already occupied by the 6. So, we apply the quadratic probing technique.
When i = 0
Index= $(5+0_2)\%10 = 5$
When i=1
Index = $(5+1_2)\%10 = 6$
Since location 6 is empty, so the value 11 will be added at the index 6.

From <https://www.tpointtech.com/hash-table>

Double Hashing

Double hashing is an open addressing technique which is used to avoid the collisions. When the collision occurs then this technique uses the secondary hash of the key. It uses one hash value as an index to move forward until the empty location is found.
In double hashing, two hash functions are used. Suppose $h_1(k)$ is one of the hash functions used to calculate the locations whereas $h_2(k)$ is another hash function. It can be defined as "insert k_i at first free place from ***(u+v*i)%m*** where i=(0 to m-1)". In this case, u is the location computed using the hash function and v is equal to $(h_2(k)\%m)$.
Consider the same example that we use in quadratic probing.
A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and
$h_1(k) = 2k+3$
$h_2(k) = 3k+1$

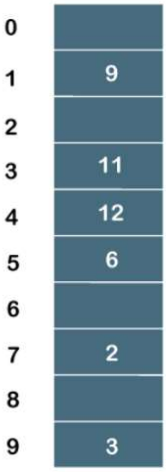
The final hash table would be:



Therefore, the order of the elements is 13, 9, _, 12, _, 6, 11, 2, 7, 3.

Screen clipping taken: 07-06-2025 15:01

The final hash table would be:



The order of the elements is _, 9, _, 11, 12, 6, _, 2, _, 3.

Screen clipping taken: 07-06-2025 15:01

==========

What is a skip list?

A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list.
It allows the process of the elements or data to view efficiently.
In one single step, it skips several elements of the entire list, which is why it is known as a skip list.
The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly.
It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

Skip list structure

It is built in two layers: The lowest layer and Top layer.
The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

Skip List Basic Operations

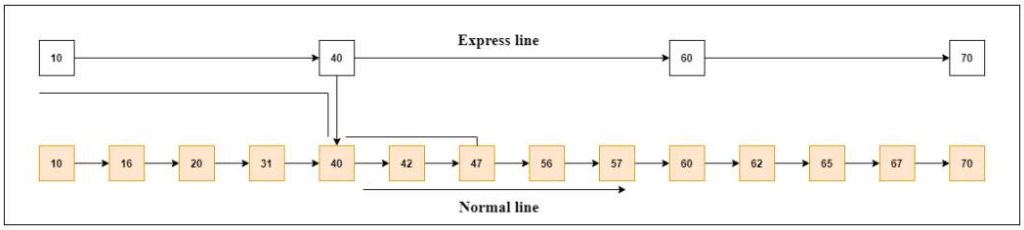
There are the following types of operations in the skip list.

- **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
- **Deletion operation:** It is used to delete a node in a specific situation.
- **Search Operation:** The search operation is used to search a particular node in a skip list

From <https://www.tpointtech.com/skip-list-in-data-structure>

Suppose you want to find 47 in this example. You will start the search from the first node of the express line and continue running on the express line until you find a node that is equal a 47 or more than 47.
You can see in the example that 47 does not exist in the express line, so you search for a node of less than 47, which is 40. Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.

From <https://www.tpointtech.com/skip-list-in-data-structure>



Screen clipping taken: 07-06-2025 15:49

Applications of the Skip list

1. It is used in distributed applications, and it represents the pointers and system in the distributed applications.
2. It is used to implement a dynamic elastic concurrent queue with low lock contention.
3. It is also used with the QMap template class.
4. The indexing of the skip list is used in running median problems.
5. The skip list is used for the delta-encoding posting in the Lucene search.