Sub : Data Structure

Unit :1

The **Indian Knowledge System (IKS)** offers a fascinating look at how ancient scholars organized vast amounts of data. Centuries before modern computer science formalized "Sorting and Searching," scholars in **Sanskrit Grammar (Vyakarana)** and **Ayurveda** were using remarkably similar logical frameworks to manage linguistic and medical information.

---

1. Sanskrit Grammar: The Paninian "Hashing" Algorithm

Pāṇini's *Aṣṭādhyāyī* (c. 4th Century BCE) is often cited by computer scientists as the world's first formal language system. His methods for organizing the Sanskrit alphabet and rules resemble modern **Data Structures**.

The Shiva Sutras (Pratyahara-sutras)

Panini organized the Sanskrit phonemes into 14 verses called the *Shiva Sutras*.

- **The Technique:** This wasn't just a list; it was a **compressed index**. By taking the first letter of a range and the "marker" (IT-letter) at the end, he created a **Pratyahara** (a shorthand code).
- **Modern Parallel:** This is akin to **Hashing** or **Bitmasking**. Instead of searching through every vowel, a single code like *Ac* instantly points to the entire set of vowels.

Anuvritti: The Recursive Search

In the 4,000 rules of the *Aṣṭādhyāyī*, Panini avoids repetition by using *Anuvritti*.

- **The Technique:** A keyword mentioned in an earlier rule is "carried forward" to subsequent rules until it is explicitly cancelled.
- **Modern Parallel:** This mirrors **Inheritance** in Object-Oriented Programming and **Context-Aware Search** algorithms where the system remembers previous parameters to narrow down current results.

---

2. Ayurveda: Multi-Key Classification & Decision Trees

In Ayurveda, particularly in the *Charaka Samhita*, the classification of substances (*Dravya*) and diseases (*Roga*) follows a rigorous logical hierarchy.

Padartha Vigyan (Categorization)

Ayurveda classifies the universe into six categories (*Padarthas*). To identify a medicinal herb, an Ayurvedic practitioner uses a system of **Multi-level Sorting**:

1. **Rasa** (Taste)
2. **Guna** (Quality)
3. **Virya** (Potency)
4. **Vipaka** (Post-digestive effect)

- **Modern Parallel:** This is essentially **Multi-Key Sorting**. In a database, you might sort by `Last Name` > `First Name` > `ID`. Ayurveda sorts by `Potency` > `Taste` > `Effect` to find the specific "record" (cure) for a "query" (symptom).

Nidan Panchakam (Diagnostic Search)

The process of diagnosis (*Nidana*) is a systematic **Search Algorithm**:

- **Hetu** (Etiology/Cause)
- **Purvarupa** (Prodromal signs)
- **Rupa** (Clinical signs)
- **Upashaya** (Therapeutic test)
- **Samprapti** (Pathogenesis)

This five-step process functions like a **Decision Tree**. The physician starts with a broad set of possibilities and "prunes" the tree branches based on the presence or absence of specific signs until the specific disease is identified.

---

3. Comparison Table: Ancient vs. Modern

Ancient Method	Field	Modern CS Equivalent	Logic
Pratyahara	Sanskrit	**Hashing / Indexing**	Using short keys to represent large data sets.
Anuvritti	Sanskrit	**Recursion / Scope**	Carrying context from one block of code to the next.

Ancient Method	Field	Modern CS Equivalent	Logic
Vichaya	Nyaya/Ayurveda	**Search / Discovery**	Systematic investigation of a "target" (*Lakshya*).
Tantra-yukti	Ayurveda	**System Architecture**	Methodologies for composing and interpreting complex text.

Why it was "Efficient"

Ancient scholars faced a unique constraint: **Memory.** Since knowledge was oral, data structures had to be **lossless but highly compressed**.

The **Sutras** were designed to be "Alpaksaram" (minimalist in vowels). This forced the creation of high-efficiency sorting and searching mechanisms—if the "search" took too long or the "data" was too bulky, the oral tradition would fail. Essentially, the human brain was the CPU, and the Sanskrit language was the optimized machine code.

Would you like to dive deeper into how Pāṇini's rules specifically influenced the development of Backus-Naur Form (BNF) in modern programming?

Indian Knowledge System (IKS) & Data Structures

Ancient Indian scholars developed sophisticated systems for organizing vast amounts of information, ranging from medicinal properties in **Ayurveda** to complex grammatical rules in **Sanskrit**.

1. The Buddhist Numerical Sorting Method

While often categorized under general Indian mathematics, Buddhist logic (especially in texts like the *Abhidharma*) used a system of **numerical categories** to classify mental and physical phenomena.

- **Hierarchical Classification:** Knowledge was grouped into "Vargas" (classes), which resembles modern **Object-Oriented Programming (OOP)** or hierarchical clustering.
- **The Bhāskara II Connection:** Bhāskara II (1114–1185 CE) was a master of the decimal system. His work, the *Līlāvatī*, includes combinations and permutations (*Anka-pasha*), which are the mathematical bedrock for **sorting algorithms**.

2. Ayurveda & Search Algorithms

Ayurveda uses a multi-criteria classification system based on:

- **Doshas:** (Vata, Pitta, Kapha)
- **Gunas:** (Attributes)
- **Rasa:** (Taste)

  This "indexing" allowed ancient practitioners to perform a **logical search** for cures, much like how a modern database uses **multi-key indexing** to retrieve specific records.

🔬 Ancient Concepts vs. Modern Techniques

The table below illustrates how ancient methods parallel modern Data Structures and Algorithms (DSA):

Ancient Indian Concept	Modern Computer Science Equivalent
Varga (Classes)	Data Classification / Clustering
Sutra (Aphorisms)	Compressed Algorithms / Logic
Anka-pasha	Combinatorics / Permutation-based Sorting
Vedic Lineage (Vamshavali)	Tree Data Structures / Genealogy Graphs
Temple Architecture Connectivity	Graph Theory / Connectivity Models

---

Suggested Academic Activities

Based on the text from Vikram University (Ujjain), students are encouraged to explore these "Vedic implementations":

- **Vedic Sorting Implementation:** Develop code for a sorting algorithm inspired by Ayurvedic categorization (sorting by multiple medicinal attributes).
- **Genealogy Trees:** Use **Binary Trees** to model ancient Indian lineage texts (Vedic lineage).
- **Graphviz Modeling:** Model the network connectivity of ancient Indian temples as a **Graph Data Structure**.

Note on Bhāskara II: He is most famous for the **Chakravala Method**, an iterative algorithm to solve indeterminate quadratic equations. This iterative approach is essentially a precursor to **numerical methods** used in modern computing.

Unit :II

Data Structure: Basic concepts, Linear and Non-Linear data structures

- **Linear vs. Non-Linear: * Linear:** Data elements are arranged sequentially (e.g., Arrays, Stacks, Queues).
    - **Non-Linear:** Data elements are attached hierarchically or interconnected (e.g., Trees, Graphs).
- **Recursive Algorithms:** Functions that call themselves to solve smaller sub-problems. This is a key concept in "Vedic" mathematical shortcuts where complex calculations are broken into repetitive steps.
- **Performance Analysis:** Using **Big O Notation** to measure time and space complexity—essentially asking, "How much memory and time does this logic take?"

Arrays & Matrix Representation

1. Array Representation (1D)

A 1D array is a linear collection of elements accessible by a single index.

GeeksforGeeks

- **Declaration:** `data_type array_name[size];` (e.g., `int arr[5];`).
- **Memory:** Elements are stored in a contiguous block, allowing constant-time access `O(1)`.
- **Indexing:** Starts at `0` and ends at `size - 1`.
- **Operations:** Includes traversal, insertion, deletion, and searching.

TutorialsPoint +4

2. Matrix Representation (2D Array)

A matrix is represented as a two-dimensional array, logically organized into rows and columns.

GeeksforGeeks +1

- **Declaration:** `data_type matrix_name[rows][cols];` (e.g., `int mat[3][4];`).
- **Accessing Elements:** Uses two indices: `matrix_name[row_index][column_index]`.
- **Memory Layout:** C++ uses **Row-Major Order**, where all elements of the first row are stored first, followed by the second row, etc..
- **Traversing:** Typically requires nested for-loops.

3. Advanced Representations

- **Dynamic Allocation:** For matrices where the size is unknown at compile-time, use `new` or std::vector (e.g., `vector<vector<int>> matrix;`).

- **Sparse Matrix:** Matrices with many zeros are represented using specialized structures like "triplet representation" (Row, Column, Value) to save memory.
- **Pseudo-Multidimensional Array:** A large 1D array can simulate a 2D matrix using the formula: `index = (i * num_columns) + j`

Performance Analysis

When analyzing the algorithms in your syllabus, you will likely use these standard variables:

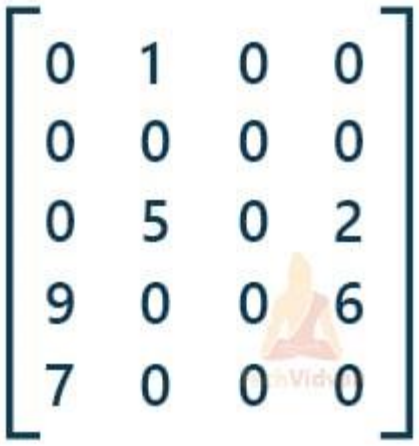
$$T(n) = O(f(n))$$

Complexity	Name	Example from Syllabus
$O(1)$	Constant	Accessing an element in a 1D Array
$O(n)$	Linear	Searching through a Singly Linked List
$O(n^2)$	Quadratic	Basic Bubble Sort or Selection Sort
$O(\log n)$	Logarithmic	Searching in a Binary Search Tree

Sparse Matrix in Data Structure

In programming, we usually represent a 2-D array in the form of a matrix. However, if a matrix has most of its elements equal to zero, then the matrix is known as a sparse matrix. In the case of a sparse matrix, we don't store the zeros in the memory to reduce memory usage and make it more efficient. We only store the non-zero values of the sparse matrix inside the memory.

For example, in the following 5*4 matrix, most of the numbers are zero. Only a few elements are non-zero which makes it a sparse matrix.



Thus, a sparse matrix is a matrix in which the number of zeros is more than the number of non-zero elements. If we store this sparse matrix as it is, it will consume a lot of space. Therefore, we store only non-zero values in the memory in a more efficient way.

Why Sparse Matrix:

There are mainly two reasons for using sparse matrices. These are:

1. Computation time: If we store the sparse matrix in a memory-efficient manner, we can save a lot of computational time to perform operations on the matrix.
2. Storage: When we store only non-zero elements, we can save a lot of memory/space that we can use for storing other data structures or performing other operations.

Memory Representation of Sparse Matrix:

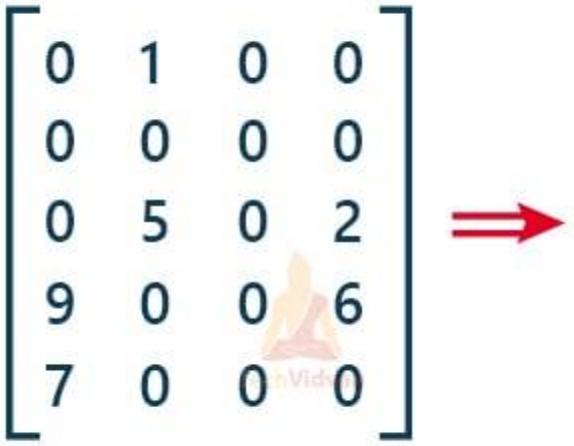
There are two types of representations for sparse matrices. These are based on the type of data structure used to store the sparse matrix. Based on this, the representations are:

1. Array representation
2. Linked list representation

Array Representation:

In an array representation, we make use of arrays to store a sparse matrix. The sparse matrix is stored in a 2-D array having three rows as follows:

1. Row: It stores the index of the row, where we have a non-zero element in the sparse matrix.
2. Column: It stores the index of the column, where we have the non-zero element in the sparse matrix.
3. Value: This variable consists of the actual non-zero value being stored.
For example, the following diagram shows how we can represent a sparse matrix with the help of an array by storing only non-zero elements in the array along with their row number and column number.



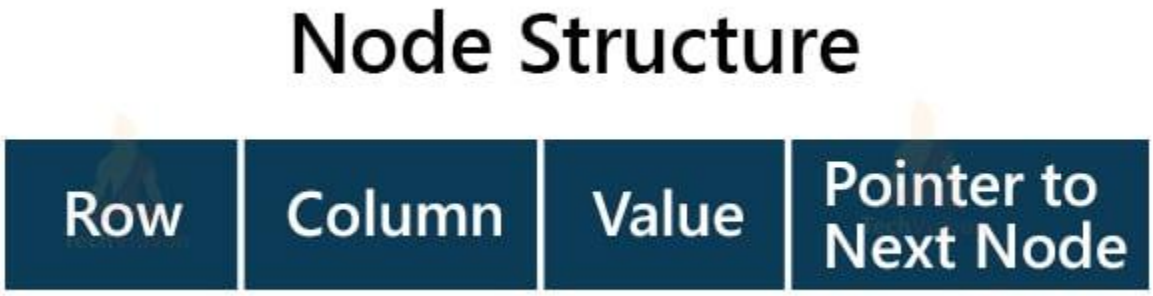
Row	Column	Value
0	1	1
2	1	5
2	3	2
3	0	9
3	3	6
4	0	7

Linked List Representation:

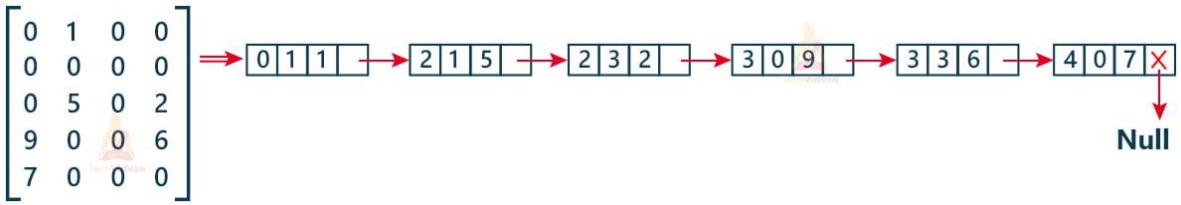
A linked list is a combination of interconnected rows joined in a linear manner. In linked list representation, each node consists of four components:

1. Row: It stores the index of the row, where we have a non-zero element in the sparse matrix.

2. Column: It stores the index of the column, where we have the non-zero element in the sparse matrix.
3. Value: This variable consists of the actual non-zero value being stored.
4. Next node: It is a pointer to store the address of the next connected node.
Thus, we can represent the node as follows:



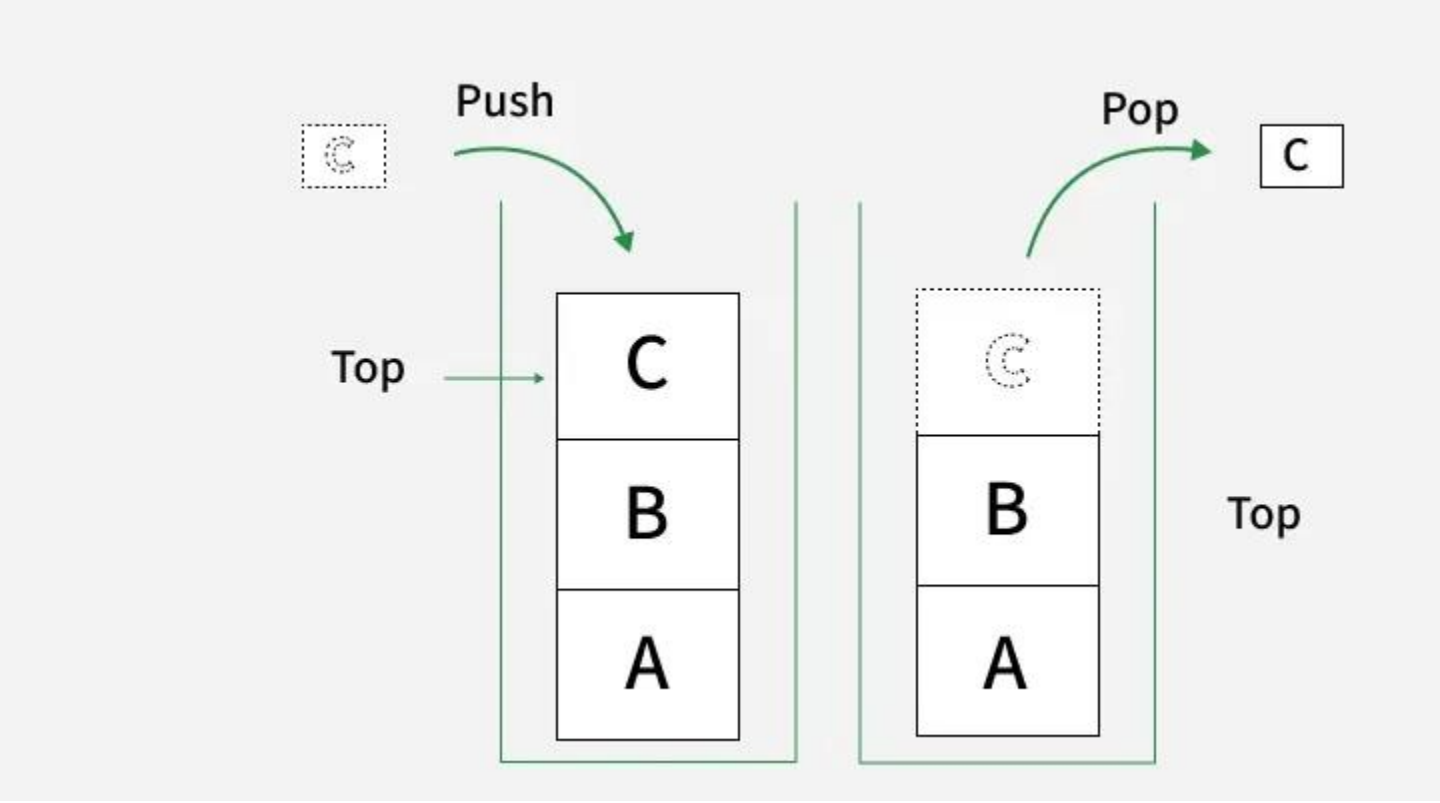
In the following diagram, we can see the linked list representation of a sparse matrix.



Stack Data Structure

- A **Stack** is a linear data structure that follows a particular order in which the operations are performed. The order may be **LIFO(Last In First Out)** or **FILO(First In Last Out)**. **LIFO** implies that the element that is inserted last, comes out first and **FILO** implies that the element that is inserted first, comes out last.
  It behaves like a stack of plates, where the last plate added is the first one to be removed. **Think of it this way:**
  Pushing an element onto the stack is like adding a new plate on top.

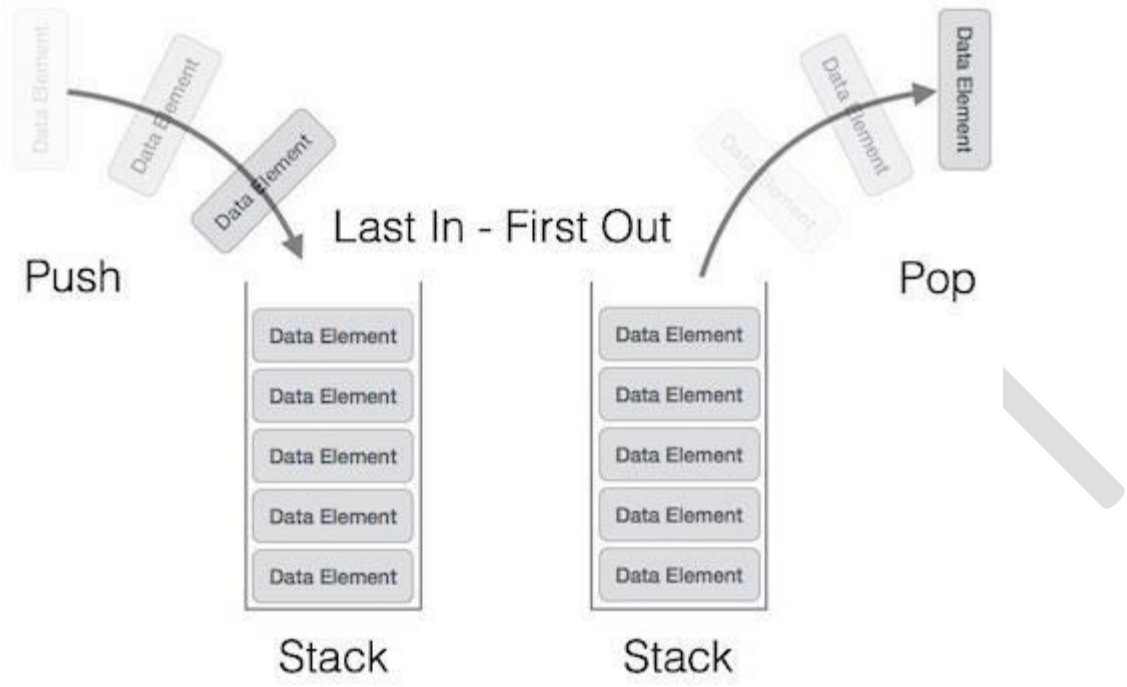
  Popping an element removes the top plate from the stack.



Stack Representation

A stack allows all data operations at one end only. At any given time, we can only access the top element of a stack.

The following diagram depicts a stack and its operations –



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations on Stacks

Stack operations are usually performed for initialization, usage and, de-initialization of the stack ADT.

The most fundamental operations in the stack ADT include: push(), pop(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the stack.

Stack uses pointers that always point to the topmost element within the stack, hence called as the **top** pointer.

Stack Insertion: push()

The push() is an operation that inserts elements into the stack. The following is an algorithm that describes the push() operation in a simpler way.

Algorithm

1. Checks if the stack is full.
2. If the stack is full, produces an error and exit.
3. If the stack is not full, increments top to point next empty space.
4. Adds data element to the stack location, where top is pointing.
5. Returns success.

Example

Following are the implementations of this operation in various programming languages –

C **C++** **Java** **Python**

```
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;

/* Check if the stack is full*/
int isfull(){
   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

/* Function to insert into the stack */
int push(int data){
   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
```

```
    }
}

/* Main function */
int main(){
    int i;
    push(44);
    push(10);
    push(62);
    push(123);
    push(15);
    printf("Stack Elements: \n");

    // print stack data
    for(i = 0; i < 8; i++) {
        printf("%d ", stack[i]);
    }
    return 0;
}
```

Output

```
Stack Elements:

44 10 62 123 15 0 0 0
```

Note – In Java we have used to built-in method **push()** to perform this operation.

Stack Deletion: pop()

The *pop()* is a data manipulation operation which removes elements from the stack. The following pseudo code describes the pop() operation in a simpler way.

Algorithm

```
1. Checks if the stack is empty.
```

2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, accesses the data element at which top is pointing.
4. Decreases the value of top by 1.
5. Returns success.

Example

Following are the implementations of this operation in various programming languages –

C C++ Java Python

```
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;

/* Check if the stack is empty */
int isempty(){
   if(top == -1)
      return 1;
   else
      return 0;
}

/* Check if the stack is full*/
int isfull(){
   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

/* Function to delete from the stack */
int pop(){
   int data;
```

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

/* Function to insert into the stack */
int push(int data){
   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}

/* Main function */
int main(){
   int i;
   push(44);
   push(10);
   push(62);
   push(123);
   push(15);
   printf("Stack Elements: \n");

   // print stack data
   for(i = 0; i < 8; i++) {
      printf("%d ", stack[i]);
   }
   /*printf("Element at top of the stack: %d\n" ,peek());*/
   printf("\nElements popped: \n");
```

```
   // print stack data
   while(!isempty()) {
      int data = pop();
      printf("%d ",data);
   }
   return 0;
}
```

Output

```
Stack Elements:

44 10 62 123 15 0 0 0

Elements popped:

15 123 62 10 44
```

Note – In Java we are using the built-in method pop().

Retrieving topmost Element from Stack: peek()

The *peek()* is an operation retrieves the topmost element within the stack, without deleting it. This operation is used to check the status of the stack with the help of the top pointer.

Algorithm

```
1. START
2. return the element at the top of the stack
3. END
```

Example

Following are the implementations of this operation in various programming languages –

C C++ Java Python

```
#include <stdio.h>
int MAXSIZE = 8;
```

```
int stack[8];
int top = -1;

/* Check if the stack is full */
int isfull(){
   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

/* Function to return the topmost element in the stack */
int peek(){
   return stack[top];
}

/* Function to insert into the stack */
int push(int data){
   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}

/* Main function */
int main(){
   int i;
   push(44);
   push(10);
   push(62);
   push(123);
   push(15);
   printf("Stack Elements: \n");
```

```
    // print stack data
    for(i = 0; i < 8; i++) {
       printf("%d ", stack[i]);
    }
    printf("\nElement at top of the stack: %d\n" ,peek());
    return 0;
}
```

Output

```
Stack Elements:

44 10 62 123 15 0 0 0

Element at top of the stack: 15
```

Verifying whether the Stack is full: isFull()

The *isFull()* operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

```
1. START
2. If the size of the stack is equal to the top position of the
stack,
   the stack is full. Return 1.
3. Otherwise, return 0.
4. END
```

Example

Following are the implementations of this operation in various programming languages –

C C++ Java Python

```
#include <stdio.h>
int MAXSIZE = 8;
```

```
int stack[8];
int top = -1;

/* Check if the stack is full */
int isfull(){
   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

/* Main function */
int main(){
   printf("Stack full: %s\n" , isfull()?"true":"false");
   return 0;
}
```

Output

```
Stack full: false
```

Verifying whether the Stack is empty: isEmpty()

The *isEmpty()* operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1. START
2. If the top value is -1, the stack is empty. Return 1.
3. Otherwise, return 0.
4. END

Example

Following are the implementations of this operation in various programming languages –

C **C++** **Java** **Python**

```
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;

/* Check if the stack is empty */
int isempty() {
   if(top == -1)
      return 1;
   else
      return 0;
}

/* Main function */
int main() {
   printf("Stack empty: %s\n" , isempty()?"true":"false");
   return 0;
}
```

Output

```
Stack empty: true
```

Stack Complete implementation

Following are the complete implementations of Stack in various programming languages –

C **C++** **Java** **Python**

```
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;
/* Check if the stack is empty */
int isempty(){
```

```
   if(top == -1)
      return 1;
   else
      return 0;
}
/* Check if the stack is full */
int isfull(){
   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

/* Function to return the topmost element in the stack */
int peek(){
   return stack[top];
}

/* Function to delete from the stack */
int pop(){
   int data;
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}

/* Function to insert into the stack */
int push(int data){
   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
```

```
        printf("Could not insert data, Stack is full.\n");
    }
}
/* Main function */
int main(){
    push(44);
    push(10);
    push(62);
    push(123);
    push(15);
    printf("Element at top of the stack: %d\n" ,peek());
    printf("Elements: \n");
    // print stack data
    while(!isempty()) {
        int data = pop();
        printf("%d\n",data);
    }
    printf("Stack full: %s\n" , isfull()?"true":"false");
    printf("Stack empty: %s\n" , isempty()?"true":"false");
    return 0;
}
```

Output

```
Element at top of the stack: 15

Elements:

15123

62

10

44

Stack full: false

Stack empty: true
```

Unit :III

Stack

Implement a stack using a linked list with push, pop

```
#include <iostream>    // Include the iostream header for input and output operations

using namespace std;   // Use the std namespace to simplify code

class Node {

public:

    int data;          // Data of the node

    Node* next;        // Pointer to the next node in the linked list

};

class Stack {

private:

    Node* top;          // Pointer to the top of the stack

public:

    Stack() {

        top = NULL;      // Initialize top pointer to NULL (empty stack)

    }

    void push(int x) {

        Node* newNode = new Node();  // Create a new node

        newNode->data = x;           // Set the data of the new node to the input value
```

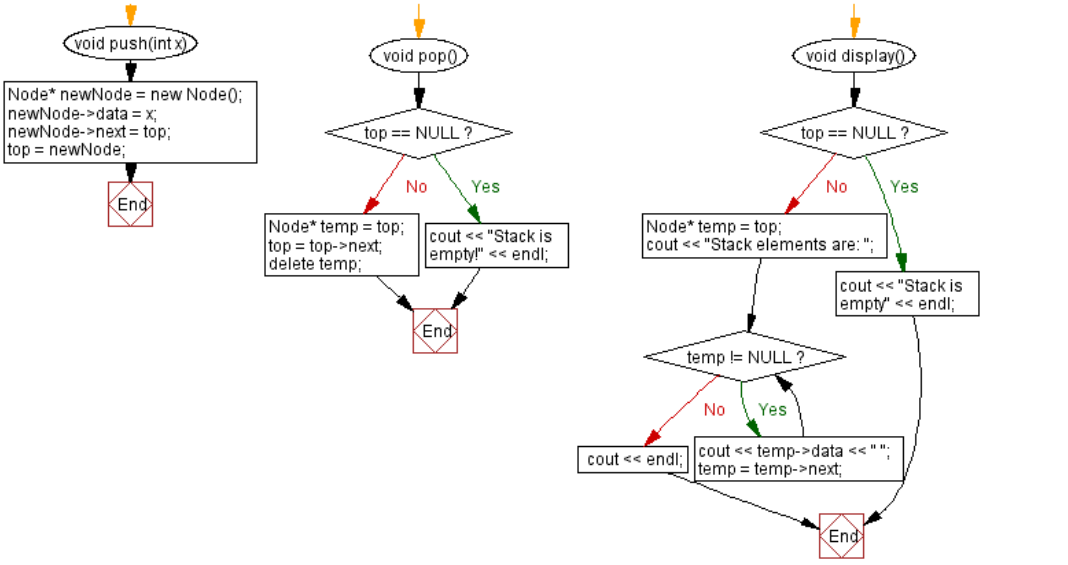
```
        newNode->next = top;          // Set the next pointer of the
new node to the current top

        top = newNode;                // Update the top pointer to the
new node

    }

    void pop() {

        if (top == NULL) {

            cout << "Stack is empty!" << endl;  // If the stack is
empty, display an error message

            return;

        }

        Node* temp = top;             // Temporary pointer to the
current top

        top = top->next;              // Update the top pointer to the
next node

        delete temp;                  // Delete the old top node

    }

    void display() {

        if (top == NULL) {

            cout << "Stack is empty" << endl;  // If the stack is
empty, display an empty message

            return;

        }

        Node* temp = top;             // Temporary pointer to traverse
the stack

        cout << "Stack elements are: ";

        while (temp != NULL) {        // Traverse the stack and print
each element

            cout << temp->data << " ";
```

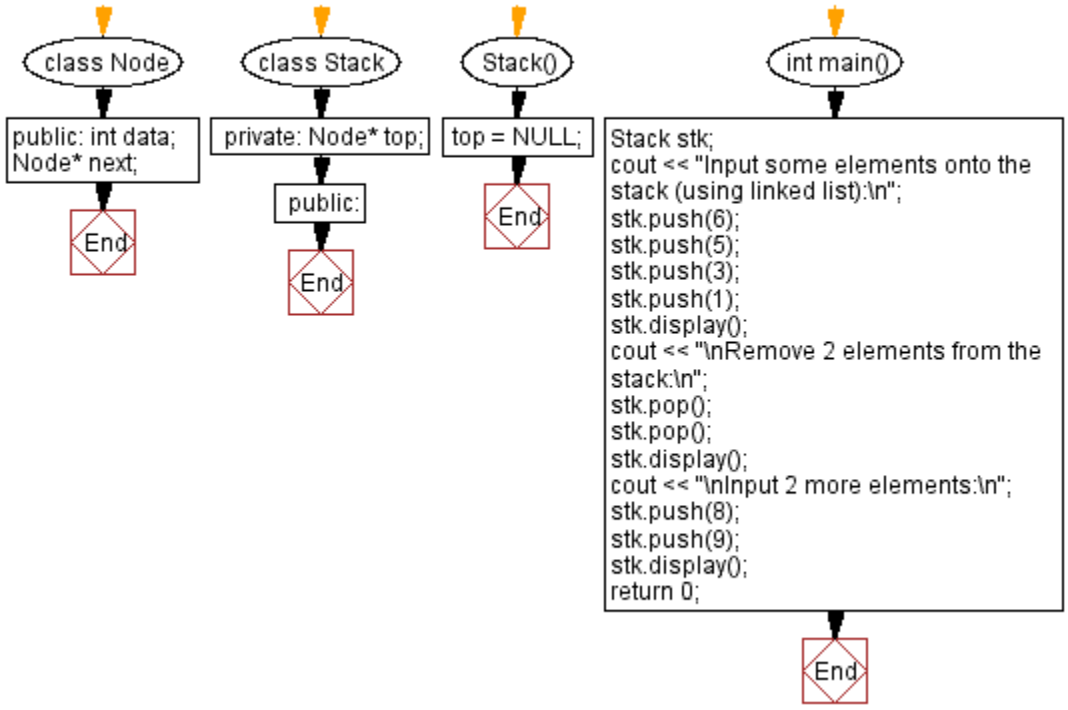
```
            temp = temp->next;
        }
        cout << endl;
    }
};
int main() {
    Stack stk;                      // Create a stack object
    cout << "Input some elements onto the stack (using linked
list):\n";
    stk.push(6);
    stk.push(5);
    stk.push(3);
    stk.push(1);
    stk.display();                  // Display the elements in the
stack
    cout << "\nRemove 2 elements from the stack:\n";
    stk.pop();
    stk.pop();
    stk.display();                  // Display the updated stack
    cout << "\nInput 2 more elements:\n";
    stk.push(8);
    stk.push(9);
    stk.display();                  // Display the final state of the
stack
    return 0;
}
```

Sample Output:

```
Input some elements onto the stack (using linked list):
Stack elements are: 1 3 5 6

Remove 2 elements from the stack:
Stack elements are: 5 6

Input 2 more elements:
Stack elements are: 9 8 5 6
```

Flowchart:





Stack - Linked List Implementation

A stack is a linear data structure that follows the **Last-In-First-Out (LIFO)** principle. It can be implemented using a linked list, where each

element of the stack is represented as a node. The head of the linked list acts as the top of the stack.

Declaration of Stack using Linked List

A stack can be implemented using a **linked list** where we maintain:

- A Node structure/class that contains:
  data → to store the element.
  next → pointer/reference to the next node in the stack.
- A pointer/reference top that always points to the current **top node** of the stack.
  Initially, top = null to represent an empty stack.

// Node structure

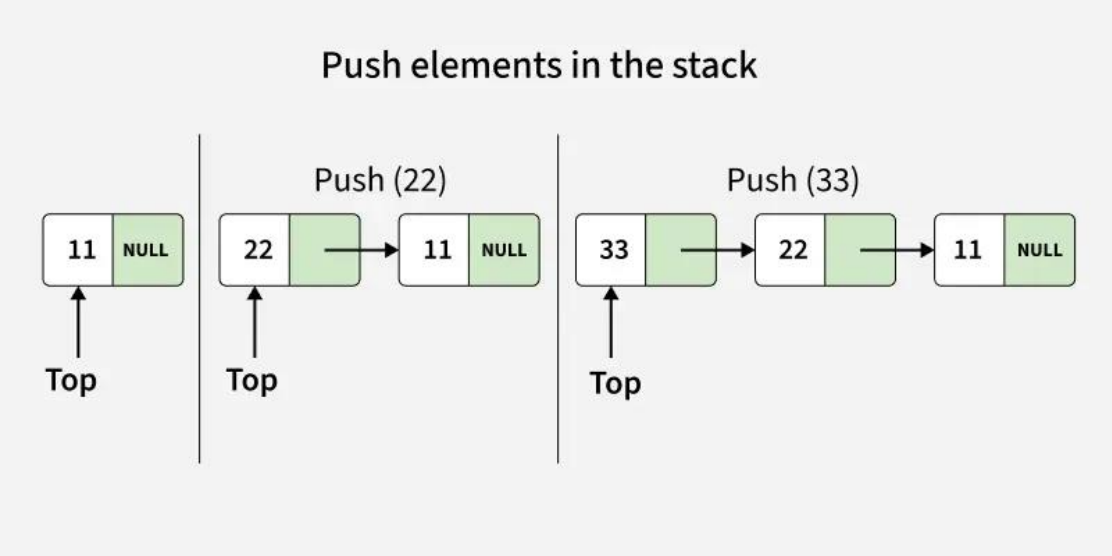
```
class Node {
public:
    int data;
    Node* next;

    Node(int x) {
        data = x;
        next = NULL;
    }
};

// Stack class
class myStack {

    // pointer to top node
    Node* top;

public:
    myStack() {

        // initially stack is empty
        top = NULL;
    }
};
```

Operations on Stack using Linked List

Push Operation

Adds an item to the stack. Unlike array implementation, there is no fixed capacity in linked list. Overflow occurs only when memory is exhausted.

- A new node is created with the given value.
- The new node's next pointer is set to the current top.
- The top pointer is updated to point to this new node.



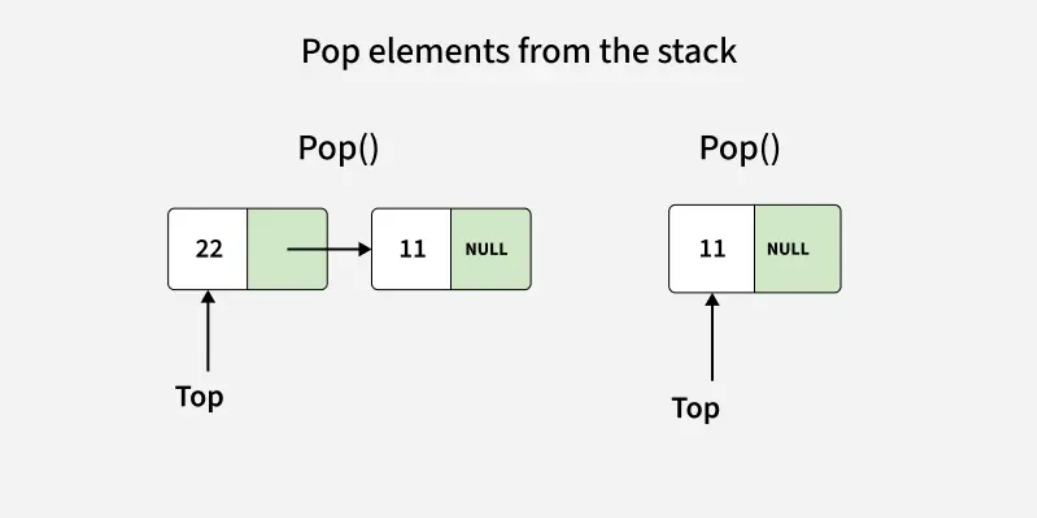

```
void push(int x) {
    Node* temp = new Node(x);
    temp->next = top;
    top = temp;
}
```

Time Complexity: O(1)
Auxiliary Space: O(1)

Pop Operation

Removes the top item from the stack. If the stack is empty, it is said to be an **Underflow condition**.

- Before deleting, we check if the stack is empty (top == NULL).
- If the stack is empty, underflow occurs and deletion is not possible.
- Otherwise, we store the current top node in a temporary pointer.
- Move the top pointer to the next node.
- Delete the temporary node to free memory.




```
int pop() {

    if (top == NULL) {
        cout << "Stack Underflow" << endl;
        return -1;
    }

    Node* temp = top;
    top = top->next;
    int val = temp->data;

    delete temp;
    return val;
}
```

Time Complexity: O(1)
Auxiliary Space: O(1)

Peek (or Top) Operation

Returns the value of the top item without removing it from the stack.

- If the stack is empty (top == NULL), then no element exists.
- Otherwise, simply return the data of the node pointed by top.

```
int peek() {

    if (top == NULL) {
        cout << "Stack is Empty" << endl;
        return -1;
    }
```

```
    return top->data;
}
```

Time Complexity: O(1)
Auxiliary Space: O(1)

isEmpty Operation

Checks whether the stack has no elements.

- If the top pointer is NULL, it means the stack is empty and the function returns true.
- Otherwise, it returns false.

```
bool isEmpty()
{
    return top == NULL;
}
```

Time Complexity: O(1)
Auxiliary Space: O(1)

Stack Implementation using Linked List

```
#include <iostream>
using namespace std;

// Node structure
class Node {
public:
    int data;
    Node* next;

    Node(int x) {
        data = x;
        next = NULL;
    }
};

// Stack implementation using linked list
class myStack {
    Node* top;
```

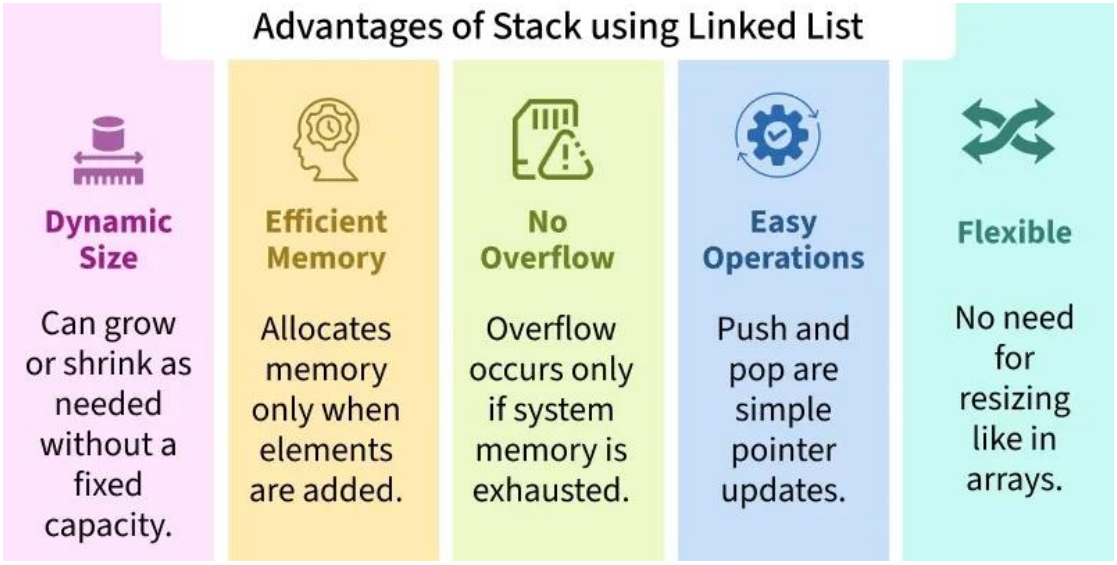
```
    // To Store current size of stack
    int count;

public:
    myStack() {

        // initially stack is empty
        top = NULL;
        count = 0;
    }

    // push operation
    void push(int x) {
        Node* temp = new Node(x);
        temp->next = top;
        top = temp;

        count++;
    }

    // pop operation
    int pop() {
        if (top == NULL) {
            cout << "Stack Underflow" << endl;
            return -1;
        }
        Node* temp = top;
        top = top->next;
        int val = temp->data;

        count--;
        delete temp;
        return val;
    }

    // peek operation
    int peek() {
        if (top == NULL) {
            cout << "Stack is Empty" << endl;
            return -1;
        }
        return top->data;
    }

    // check if stack is empty
    bool isEmpty() {
```

```
        return top == NULL;
    }

    // size of stack
    int size() {
        return count;
    }
};

int main() {
    myStack st;

    // pushing elements
    st.push(1);
    st.push(2);
    st.push(3);
    st.push(4);

    // popping one element
    cout << "Popped: " << st.pop() << endl;

    // checking top element
    cout << "Top element: " << st.peek() << endl;

    // checking if stack is empty
    cout << "Is stack empty: " << (st.isEmpty() ? "Yes" : "No")
<< endl;

    // checking current size
    cout << "Current size: " << st.size() << endl;

    return 0;
}
```

Output

```
Popped: 4

Top element: 3

Is stack empty: No

Current size: 3
```

Advantages of Stack using Linked List



Queue Data Structure | Operations, Types & More (+Code Examples)

A queue is a linear data structure that follows the First-In, First-Out (FIFO) principle, where elements are added at the rear and removed from the front, like a line of people waiting for service.

What Is Queue Data Structure?

A **Queue** is a linear data structure that follows the **First-In, First-Out (FIFO)** principle — the element that is inserted first will be the first one to be removed. Think of it as a line of people waiting to get tickets at a movie counter: the person who joins the line first gets served first, and the newcomers have to wait at the end of the line.

Key Characteristics Of A Queue:

- **Front**: The position where elements are removed from the queue.
- **Rear (or Back)**: The position where elements are added to the queue.
- **FIFO Order**: Elements are processed in the exact sequence they were added.
- **Two Main Operations**:
  - **Enqueue**: Add an element to the rear (end) of the queue.
  - **Dequeue**: Remove an element from the front (beginning) of the queue.
- **Access**: Only the front element can be removed, and only the rear end can be used to insert.

Structure Of A Queue Data Structure:

```
Front --> [10] [20] [30] [40] <-- Rear

          ↑              ↑

       Dequeue        Enqueue
```

Real-Life Analogy: A Queue At A Ticket Counter 🎟️

Imagine you're waiting in line at a movie theater to buy tickets.

- The person who joins the queue **first** is the **first** to buy the ticket (get served).
- Others must wait their turn at the **rear** of the queue.
- No one can "cut the line" — just like how elements in a queue can't skip positions.

Types Of Queues In Data Structures

A Queue is a linear structure, but depending on the specific need or optimization, it can take different forms. Let's explore the most common and useful types of queues:

1. Simple Queue (Linear Queue)

A **simple queue** follows the **First In, First Out (FIFO)** principle strictly.

- Insertion happens at the **rear end.**
- Deletion happens at the **front end.**

Once the queue becomes full, even if some elements are removed from the front, no new elements can be inserted unless the array is shifted, which can be inefficient.

Example: Ticket counters, print queues.

2. Circular Queue

A **circular queue** overcomes the limitation of wasted space in a simple queue.

- In a circular queue, the last position is connected back to the first position, making the queue circular in structure.
- When the rear reaches the end of the array and there is space at the beginning (due to previous dequeues), new elements can be inserted at the start.

Example: Buffer management in computer systems, CPU scheduling.

3. Priority Queue

In a **priority queue**, each element is associated with a priority.

- Elements are dequeued based on their priority, not just the order of insertion.
- Higher priority elements are removed first. If two elements have the same priority, they are served according to their order in the queue (FIFO among same-priority elements).

Example: Emergency room patients in a hospital, task scheduling in operating systems.

4. Double-Ended Queue (Deque)

A **deque** (pronounced "deck") is a **double-ended queue** that allows insertion and deletion from **both the front and rear** ends.

- It can be used as both a **queue (FIFO)** and a **stack (LIFO).**
- There are two variations:
    - **Input-restricted deque:** Insertion allowed at only one end, deletion allowed at both ends.
    - **Output-restricted deque:** Deletion allowed at only one end, insertion allowed at both ends.

Example: Palindrome checking, undo operations in text editors.

Summary Table:

Queue Type	Insertion	Deletion	Special Feature	Use Cases
Linear Queue	Rear	Front	Simple FIFO	Print queues, service counters
Circular Queue	Rear (Circular)	Front (Circular)	Efficient use of storage	OS scheduling, buffering
Priority Queue	Rear (with order)	Front (based on priority)	Elements dequeued based on priority	CPU Scheduling, A* algorithm
Deque	Both Front & Rear	Both Front & Rear	Insertion & deletion from both ends	Undo features, task scheduling

Each type of queue is designed to solve particular problems efficiently and is chosen depending on the needs of the application.

Basic Operations In Queue Data Structure

A queue supports several basic operations that allow it to behave according to the First In, First Out (FIFO) rule. These operations manage inserting, removing, checking, and displaying elements. Here are the main operations, along with their algorithm:

1. Enqueue (Insertion) Operation

The enqueue operation is used to insert a new element into the queue at the rear end. When we enqueue, we first check if the queue is full. If it is not full, we increase the rear index and place the new element at that position. If the queue was initially empty, both front and rear are set to the first element.

Algorithm:

1. Check if the queue is full.
2. If not full, increase rear and insert the element at rear.

Example:

```
void enqueue(int value) {
    if (rear == size - 1) {
        cout << "Queue Overflow!" << endl;
        return;
    }
    if (front == -1) front = 0;
    arr[++rear] = value;
}
```

2. Dequeue (Deletion) Operation

The dequeue operation is used to remove an element from the front end of the queue. Before dequeuing, we check if the queue is empty. If it is not empty, we remove the element pointed to by front and move the front pointer one step ahead. If the queue becomes empty after the operation, front and rear can be reset.

Algorithm:

1. Check if the queue is empty.
2. If not empty, remove the element at front and move front forward.

Example:

```
void dequeue() {
    if (front == -1 || front > rear) {
        cout << "Queue Underflow!" << endl;
        return;
    }
    front++;
}
```

3. Peek / Front Operation

The peek operation is used to view the element at the front of the queue without removing it. Peek checks if the queue is empty. If it is not empty, it returns the element at the front index without modifying the queue. This helps in accessing the next element to be removed.

Algorithm:

1. Check if the queue is empty.
2. If not empty, return the element at front.

Example:

```
int peek() {
    if (front == -1 || front > rear) {
        cout << "Queue is Empty!" << endl;
        return -1;
    }
    return arr[front];
}
```

4. isEmpty Operation

The isEmpty operation checks whether the queue contains any elements or not. The queue is empty if the front is -1 (never initialized) or if the front has moved past the rear. This operation is useful to avoid underflow during dequeue or peek operations.

Algorithm: If front == -1 or front > rear, the queue is empty.

Example:

```
bool isEmpty() {
    return (front == -1 || front > rear);
}
```

5. isFull Operation

The isFull operation checks whether the queue has reached its maximum capacity. In an array-based queue, the queue is full when the rear index reaches size - 1. If it is full, no more elements can be added unless some are removed (or unless a circular queue is used).

Algorithm: If rear == size - 1, the queue is full.

Example:

```
bool isFull() {
    return (rear == size - 1);
}
```

Note: This check is only applicable to **static array-based queues**, not linked list or STL queues which are dynamically resizable.

Time Complexity Overview

Operation	Time Complexity
Enqueue	O(1)
Dequeue	O(1)
Peek / Front	O(1)
isEmpty	O(1)
isFull	O(1)

All these operations are **constant-time** because they either modify a pointer/index or perform a comparison.

Queue Implementation Using Arrays

In a queue, insertion (called enqueue) happens at the rear end, while deletion (called dequeue) happens from the front end.

- When implementing a queue using arrays, we **use two variables: front** to track the start of the queue and **rear** to track the end. Initially, both are **set to -1** to indicate that the queue is empty.
- During **enqueue,** we move the rear pointer forward and insert the element.
- During **dequeue,** we move the front pointer forward to remove elements.
- If the rear reaches the end of the array, the **queue is full** (overflow), and if the front crosses the rear, the **queue is empty** (underflow).
- Arrays provide a simple and efficient way to implement queues when the maximum size is known in advance.

Here is a code example of queue implementation using arrays in C++ programming.

Code Example:

```
#include
using namespace std;

class Queue {
private:
    int front, rear, size;
    int* arr;

public:
    // Constructor
    Queue(int capacity) {
        size = capacity;
        arr = new int[size];
        front = rear = -1;
    }

    // Destructor
```

```
    ~Queue() {
        delete[] arr;
    }

    // Enqueue operation
    void enqueue(int value) {
        if (rear == size - 1) {
            cout << "Queue Overflow! Cannot insert " << value << endl;
            return;
        }
        if (front == -1) front = 0; // First insertion
        arr[++rear] = value;
        cout << value << " enqueued to the queue." << endl;
    }

    // Dequeue operation
    void dequeue() {
        if (front == -1 || front > rear) {
            cout << "Queue Underflow! Cannot dequeue." << endl;
            return;
        }
        cout << arr[front] << " dequeued from the queue." << endl;
        front++;
    }

    // Display operation
    void display() {
        if (front == -1 || front > rear) {
            cout << "Queue is Empty!" << endl;
            return;
        }
        cout << "Queue elements are: ";
        for (int i = front; i <= rear; i++) {
            cout << arr[i] << " ";
```

```
        }
        cout << endl;
    }
};

int main() {
    Queue q(5);

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();

    q.dequeue();
    q.display();

    q.enqueue(40);
    q.enqueue(50);
    q.enqueue(60); // Should cause overflow
    q.display();

    return 0;
}
#include <iostream>
using namespace std;

class Queue {
private:
    int front, rear, size;
    int* arr;

public:
    // Constructor
    Queue(int capacity) {
```

```
        size = capacity;
        arr = new int[size];
        front = rear = -1;
    }

    // Destructor
    ~Queue() {
        delete[] arr;
    }

    // Enqueue operation
    void enqueue(int value) {
        if (rear == size - 1) {
            cout << "Queue Overflow! Cannot insert " << value << endl;
            return;
        }
        if (front == -1) front = 0; // First insertion
        arr[++rear] = value;
        cout << value << " enqueued to the queue." << endl;
    }

    // Dequeue operation
    void dequeue() {
        if (front == -1 || front > rear) {
            cout << "Queue Underflow! Cannot dequeue." << endl;
            return;
        }
        cout << arr[front] << " dequeued from the queue." << endl;
        front++;
    }

    // Display operation
    void display() {
        if (front == -1 || front > rear) {
```

```
            cout << "Queue is Empty!" << endl;
            return;
        }
        cout << "Queue elements are: ";
        for (int i = front; i <= rear; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    Queue q(5);

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();

    q.dequeue();
    q.display();

    q.enqueue(40);
    q.enqueue(50);
    q.enqueue(60); // Should cause overflow
    q.display();

    return 0;
}
```

Output:

```
10 enqueued to the queue.
20 enqueued to the queue.
30 enqueued to the queue.
Queue elements are: 10 20 30
```

```
10 dequeued from the queue.
Queue elements are: 20 30
40 enqueued to the queue.
50 enqueued to the queue.
Queue Overflow! Cannot insert 60
Queue elements are: 20 30 40 50
```

Explanation:

In the above code example-

1. We start by ***including the <iostream> header*** to use input and output streams and declare using namespace std to avoid prefixing std:: everywhere.
2. We ***define a class named Queue*** that represents a simple queue using a dynamic array.
3. Inside the class, we declare ***private data members front, rear, and size*** to track the queue's state, along with a ***pointer arr to dynamically allocate memory*** for the elements.
4. We ***create a constructor Queue(int capacity)*** that initializes the size of the queue, allocates memory for arr, and ***sets both front and rear to -1*** to indicate the queue is initially empty.
5. We ***also provide a destructor ~Queue()*** to release the dynamically allocated memory when the queue object is destroyed, ***avoiding memory leaks.***
6. For inserting elements, we ***define an enqueue function.*** Here, we first check if the ***queue is full by comparing rear with size - 1.*** If full, we ***print an overflow message***. Otherwise, ***if it's the first insertion (front == -1), we set front to 0*** and then insert the ***new value at ++rear.***
7. For removing elements, we ***define a dequeue function.*** We check if the ***queue is empty by verifying if front is -1 or front > rear.*** If so, we print an underflow message. Otherwise, we print the dequeued element and increment front.
8. We ***define a display function*** to show all elements from front to rear. If the queue is empty, we inform the user accordingly.
9. In the ***main() function,*** we ***create a Queue object q*** with a capacity of 5.
10. We enqueue the values 10, 20, and 30 into the queue, and then display the current elements.
11. We ***dequeue one element,*** display the updated queue, and then enqueue 40 and 50.
12. When we attempt to enqueue 60, the ***queue overflows*** because it has ***reached its maximum capacity,*** and we see an overflow message.
13. We ***display the final state of the queue*** before ending the program.

Queue Implementation Using Linked List

A queue can also be implemented using a linked list instead of an array. In a linked list-based queue, each element is stored in a node that contains two parts: the data and a pointer to the next node. This approach allows the queue to grow and shrink dynamically without worrying about fixed size or overflow unless the system runs out of memory.

- We **maintain two pointers: front,** which points to the first node (for deletion), and **rear,** which points to the last node (for insertion).
- To **enqueue** an element, we **create a new node** and link it at the end of the list by updating the rear pointer.
- To **dequeue** an element, we **remove the node** pointed to by front and move the front pointer to the next node.
- If **front becomes NULL,** it means the **queue is empty.**
- Linked list-based queues are flexible and efficient for applications where the maximum number of elements is not known in advance.

Here is a code example of queue implementation using a linked list in C++.

Code Example:

```
#include
using namespace std;

// Node structure
struct Node {
    int data;
    Node* next;
};

// Queue class
class Queue {
private:
    Node* front;
    Node* rear;
```

```
public:
    // Constructor
    Queue() {
        front = rear = nullptr;
    }

    // Enqueue operation
    void enqueue(int value) {
        Node* temp = new Node();
        temp->data = value;
        temp->next = nullptr;

        if (rear == nullptr) { // Queue is empty
            front = rear = temp;
        } else {
            rear->next = temp;
            rear = temp;
        }
        cout << value << " enqueued to the queue." << endl;
    }

    // Dequeue operation
    void dequeue() {
        if (front == nullptr) {
            cout << "Queue Underflow! Cannot dequeue." << endl;
            return;
        }
        Node* temp = front;
        cout << front->data << " dequeued from the queue." << endl;
        front = front->next;

        if (front == nullptr) { // Queue becomes empty
            rear = nullptr;
        }
```

```
        delete temp;
    }

    // Display operation
    void display() {
        if (front == nullptr) {
            cout << "Queue is Empty!" << endl;
            return;
        }
        cout << "Queue elements are: ";
        Node* temp = front;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

// Main function to test the Queue
int main() {
    Queue q;

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();

    q.dequeue();
    q.display();

    q.dequeue();
    q.dequeue();
    q.dequeue(); // Attempting to dequeue from an empty queue
```

```
    return 0;
}
#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* next;
};

// Queue class
class Queue {
private:
    Node* front;
    Node* rear;

public:
    // Constructor
    Queue() {
        front = rear = nullptr;
    }

    // Enqueue operation
    void enqueue(int value) {
        Node* temp = new Node();
        temp->data = value;
        temp->next = nullptr;

        if (rear == nullptr) { // Queue is empty
            front = rear = temp;
        } else {
            rear->next = temp;
```

```
        rear = temp;
    }
    cout << value << " enqueued to the queue." << endl;
}

// Dequeue operation
void dequeue() {
    if (front == nullptr) {
        cout << "Queue Underflow! Cannot dequeue." << endl;
        return;
    }
    Node* temp = front;
    cout << front->data << " dequeued from the queue." << endl;
    front = front->next;

    if (front == nullptr) { // Queue becomes empty
        rear = nullptr;
    }
    delete temp;
}

// Display operation
void display() {
    if (front == nullptr) {
        cout << "Queue is Empty!" << endl;
        return;
    }
    cout << "Queue elements are: ";
    Node* temp = front;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
```

```
    }
};

// Main function to test the Queue
int main() {
    Queue q;

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();

    q.dequeue();
    q.display();

    q.dequeue();
    q.dequeue();
    q.dequeue(); // Attempting to dequeue from an empty queue

    return 0;
}
```

Output:

```
10 enqueued to the queue.
20 enqueued to the queue.
30 enqueued to the queue.
Queue elements are: 10 20 30
10 dequeued from the queue.
Queue elements are: 20 30
20 dequeued from the queue.
30 dequeued from the queue.
Queue Underflow! Cannot dequeue.
```

Explanation:

In the above code example-

1. We begin by ***including the <iostream> header*** for input and output operations and use ***using namespace std*** to simplify our code.
2. We ***define a Node structure that holds two members:*** an ***integer data*** to store the value and a ***pointer next*** to link to the next node.
3. We ***create a Queue class*** that ***uses linked nodes*** instead of an array. Inside the class, we ***keep two pointers, front and rear,*** to represent the start and end of the queue.
4. We ***define a constructor Queue()*** that initializes ***both front and rear to nullptr,*** indicating the queue is initially empty.
5. For inserting elements, we ***implement the enqueue function.*** Here, we ***create a new node,*** assign it the given value, and set its next pointer to nullptr.
6. If the ***queue is empty (rear == nullptr),*** we ***set both front and rear to point to the new node.*** Otherwise, we link the new node at the end of the queue by ***updating rear->next,*** and then move rear to the new node.
7. For removing elements, we ***define the dequeue function.*** If the ***queue is empty (front == nullptr),*** we ***print an underflow message.*** Otherwise, we print the value at the front, move front to the next node, and delete the old front node.
8. After dequeuing, if the ***queue becomes empty (front == nullptr),*** we also ***set rear to nullptr*** to correctly update the state.
9. We ***define a display function to print all elements from front to rear.*** If the queue is empty, we show a message accordingly.
10. In the ***main() function,*** we ***create a Queue object q*** and enqueue the values 10, 20, and 30.
11. We ***display the current elements*** of the queue after inserting these values.
12. We ***dequeue one element*** and display the updated queue.
13. We continue to dequeue all elements, and finally ***attempt another dequeue when the queue is empty,*** which triggers an underflow message.

Advantages Of Queue Data Structure

1. **Orderly Processing (FIFO Behavior):** Queues strictly follow the **First In, First Out (FIFO)** order, making them ideal for situations where elements must be processed in the order they arrive. Example: Print jobs in a printer queue are handled in the order they were submitted.
2. **Efficient Resource Sharing:** In multi-user systems (like operating systems), queues are used to manage tasks efficiently, allowing fair access to shared resources like CPU time, disk drives, etc.

3. **Simplified Scheduling:** Queues help in implementing scheduling algorithms (like round-robin scheduling), making task management straightforward and organized.
4. **Useful in Real-World Applications:** Queues are used everywhere: customer service lines, traffic systems, call centers, and more. Their behavior matches naturally with many real-world processes.
5. **Supports Variants:** Different types like **circular queues**, **priority queues**, and **double-ended queues (deque)** provide flexibility to handle a variety of specific use cases.

Disadvantages Of Queue Data Structure

1. **Fixed Size (in Array Implementation):** In a static array-based queue, the size must be defined in advance. If the queue becomes full, no more elements can be added even if there is unused space (unless using a circular queue).
2. **Wasted Space (Simple Queue):** After many dequeue operations, front elements are removed, but the physical space they occupied is not reused unless special techniques like circular arrays are used.
3. **Sequential Access Only:** Unlike arrays or linked lists where any element can be accessed directly, queues allow access only at the front (for removal) and rear (for insertion).
4. **Complex Implementation for Special Queues:** Implementing advanced types like **priority queues** or **double-ended queues** can be more complex and may require additional data structures like heaps or doubly linked lists.
5. **Overhead in Dynamic Implementation:** In linked list-based queues, managing memory (allocation and deallocation) introduces some overhead compared to simple array-based structures.

Applications Of Queue Data Structure

1. **Job Scheduling in Operating Systems:** Queues are used to schedule processes that are waiting for CPU time. Processes are handled in the order they arrive, following FIFO to ensure fairness.
2. **Handling Requests in Web Servers:** Web servers use queues to manage incoming client requests. Requests are processed one after another to maintain a stable load.
3. **Printer Spooling:** When multiple print jobs are sent to a printer, they are queued up and printed one by one in the order they were received.
4. **Call Center Systems:** Incoming calls are placed in a queue and served based on their arrival time, ensuring that no call is missed or skipped.

5. **Data Buffers (IO Buffers):** Queues are used as buffers in devices like keyboards, hard disks, or networks where data must be stored temporarily and processed in order.
6. **Breadth-First Search (BFS) in Graphs:** The BFS algorithm for traversing or searching tree or graph data structures uses a queue to keep track of the nodes yet to be explored.
7. **Traffic Systems:** Traffic at toll booths, traffic lights, or vehicle queues in congestion management can be modeled using queues to handle vehicles in a fair sequence.
8. **Resource Management in Networks:** In networking, queues manage data packets while they wait to be transmitted over the network, ensuring orderly and efficient communication.
9. **Simulation Systems:** Queues are used in simulations of real-world systems like supermarkets, banks, airports, and hospitals to study and optimize service processes.
10. **Load Balancing:** Tasks can be queued and distributed among servers for load balancing, helping systems handle large amounts of work without becoming overloaded.

Conclusion

The **queue** is a fundamental and widely used data structure that plays a critical role in both real-world systems and computer science applications. By following the **First-In, First-Out (FIFO)** principle, queues ensure orderly processing where elements are served in the exact sequence they arrive.

From simple **linear queues** to more specialized forms like **circular queues**, **priority queues**, and **double-ended queues (deques)**, each type of queue addresses specific needs related to storage efficiency, task scheduling, and resource management. Understanding the basic operations such as **enqueue**, **dequeue**, **peek**, **isEmpty**, and **isFull**, and recognizing their constant time complexity, enables developers to design solutions that are both efficient and reliable.

Whether managing print jobs, handling server requests, or implementing algorithms like breadth-first search (BFS) in graphs, queues offer a structured and predictable way to organize data. A strong grasp of queue concepts forms a foundation for mastering more complex data structures and algorithms.

By integrating queues thoughtfully into your programs, you can ensure better performance, resource handling, and a clearer logic flow — essential qualities in professional software development.

Frequently Asked Questions

Q. What is the main principle behind a queue data structure?

A queue operates on the **First-In, First-Out (FIFO)** principle. This means the first element inserted into the queue is the first one to be removed, just like standing in a line at a ticket counter.

Q. How does a circular queue differ from a linear queue?

In a **linear queue**, once the queue is full, no more elements can be inserted even if there is empty space at the front. In a **circular queue**, the rear end connects back to the front, allowing the available space to be reused efficiently.

Q. What are some real-world applications of queues?

Queues are used in a variety of real-world systems such as:

- Managing processes in operating systems (CPU scheduling).
- Handling requests in web servers.
- Task scheduling in printers.
- Breadth-first search (BFS) algorithms in graphs.

Q. Why is the time complexity of queue operations O(1)?

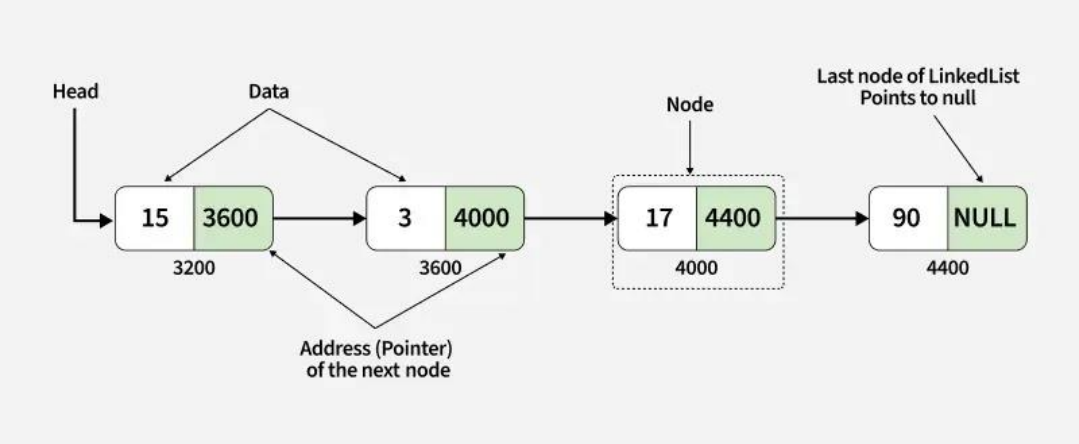
All basic queue operations like **enqueue**, **dequeue**, **peek**, **isEmpty**, and **isFull** only involve simple pointer updates or checks. They do not require shifting or searching elements, which keeps their time complexity constant, i.e., **O(1)**.

Q. What is the role of a priority queue, and how does it differ from a normal queue?

In a **priority queue**, elements are processed based on their priority rather than the order of arrival. Higher-priority elements are served before lower-priority ones. In contrast, a **normal queue** strictly follows the FIFO order without considering any priority.

Linked List Data Structure

- A linked list is a fundamental data structure in computer science. It mainly allows efficient insertion and deletion operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque.
  A linked list is a type of linear data structure individual items are not necessarily at contiguous locations. The individual items are called nodes and connected with each other using links.
  - A node contains two things first is data and second is a link that connects it with another node.
  - The first node is called the head node and we can traverse the whole list using this head and next links.



Linked List:

- **Data Structure:** Non-contiguous
- **Memory Allocation:** Typically allocated one by one to individual elements
- **Insertion/Deletion:** Efficient
- **Access:** Sequential

Array:

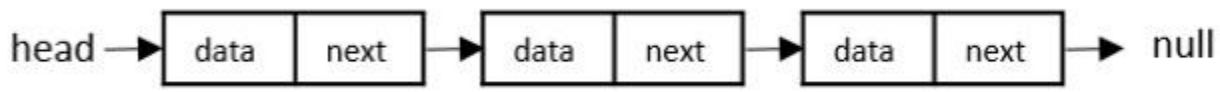
- **Data Structure:** Contiguous
- **Memory Allocation:** Typically allocated to the whole array
- **Insertion/Deletion:** Inefficient
- **Access:** Random

Types of Linked List

Following are the various types of linked list.

Singly Linked Lists

Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.



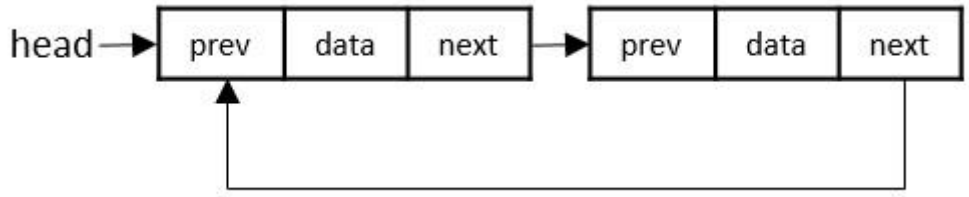
Doubly Linked Lists

Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.

Circular Linked Lists

Circular linked lists can exist in both singly linked list and doubly linked list.

Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.



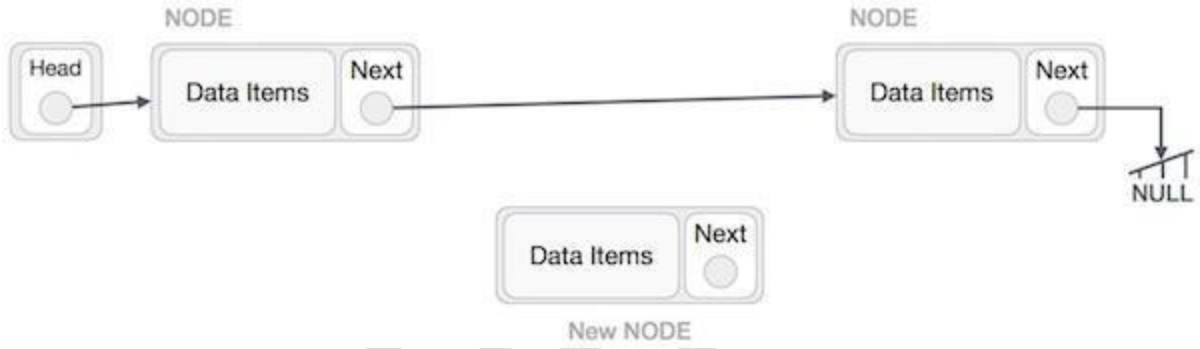
Basic Operations in Linked List

The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key. These operations are performed on Singly Linked Lists as given below –

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

Linked List - Insertion Operation

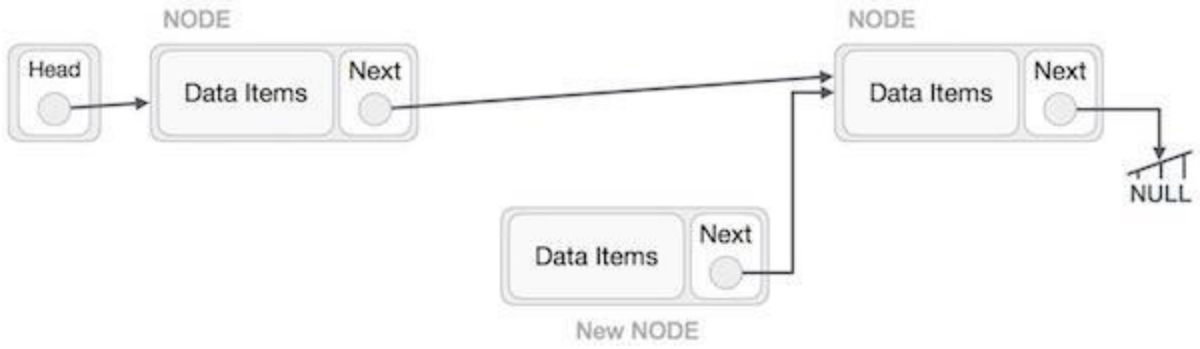
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C –

```
NewNode.next -> RightNode;
```

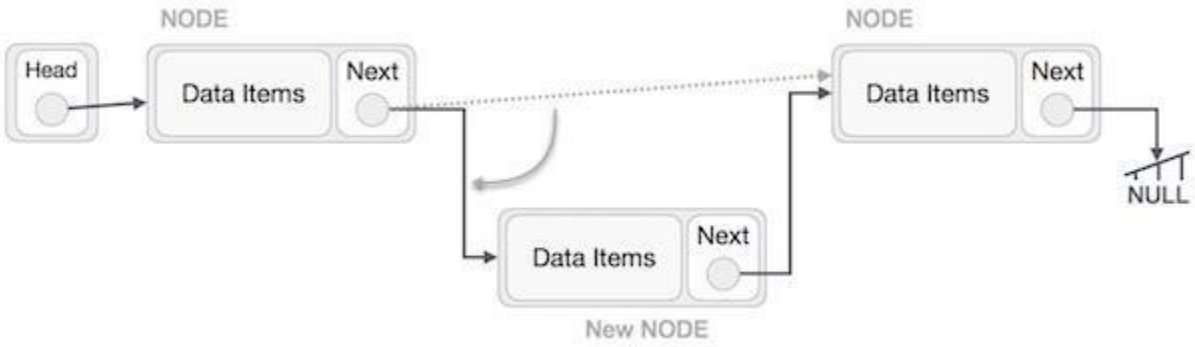
It should look like this –



Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```

This will put the new node in the middle of the two. The new list should look like this –



Insertion in linked list can be done in three different ways. They are explained as follows –

Insertion at Beginning

In this operation, we are adding an element at the beginning of the list.

Algorithm

```
1. START
2. Create a node to store the data
3. Check if the list is empty
4. If the list is empty, add the data to the node and
   assign the head pointer to it.
5. If the list is not empty, add the data to a node and link to the
   current head. Assign the head to the newly added node.
6. END
```

Example

Following are the implementations of this operation in various programming languages –

C C++ Java Python

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void main(){
   int k=0;
   insertatbegin(12);
```

```
    insertatbegin(22);
    insertatbegin(30);
    insertatbegin(44);
    insertatbegin(50);
    printf("Linked List: ");

    // print list
    printList();
}
```

Output

```
Linked List:
[ 50  44  30  22  12 ]
```

Insertion at Ending

In this operation, we are adding an element at the ending of the list.

Algorithm

```
1. START
2. Create a new node and assign the data
3. Find the last node
4. Point the last node to new node
5. END
```

Example

Following are the implementations of this operation in various programming languages –

C C++ Java Python

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
    int data;
    struct node *next;
};
```

```
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void insertatend(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;
   struct node *linkedlist = head;
```

```
    // point it to old first node
    while(linkedlist->next != NULL)
       linkedlist = linkedlist->next;

    //point first to new first node
    linkedlist->next = lk;
}
void main(){
    int k=0;
    insertatbegin(12);
    insertatend(22);
    insertatend(30);
    insertatend(44);
    insertatend(50);
    printf("Linked List: ");

    // print list
    printList();
}
```

Output

```
Linked List:
[ 12 22 30 44 50 ]
```

Insertion at a Given Position

In this operation, we are adding an element at any position within the list.

Algorithm

```
1. START
2. Create a new node and assign data to it
3. Iterate until the node at position is found
4. Point first to new first node
5. END
```

Example

Following are the implementations of this operation in various programming languages –

C **C++** **Java** **Python**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;
```

```
    //point first to new first node
    head = lk;
}
void insertafternode(struct node *list, int data){
    struct node *lk = (struct node*) malloc(sizeof(struct node));
    lk->data = data;
    lk->next = list->next;
    list->next = lk;
}
void main(){
    int k=0;
    insertatbegin(12);
    insertatbegin(22);
    insertafternode(head->next, 30);
    printf("Linked List: ");

    // print list
    printList();
}
```

Output

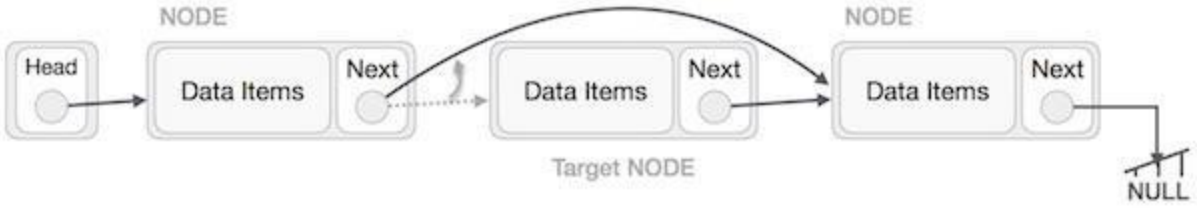
```
Linked List:
[ 22 12 30 ]
```

Linked List - Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

The left (previous) node of the target node now should point to the next node of the target node −

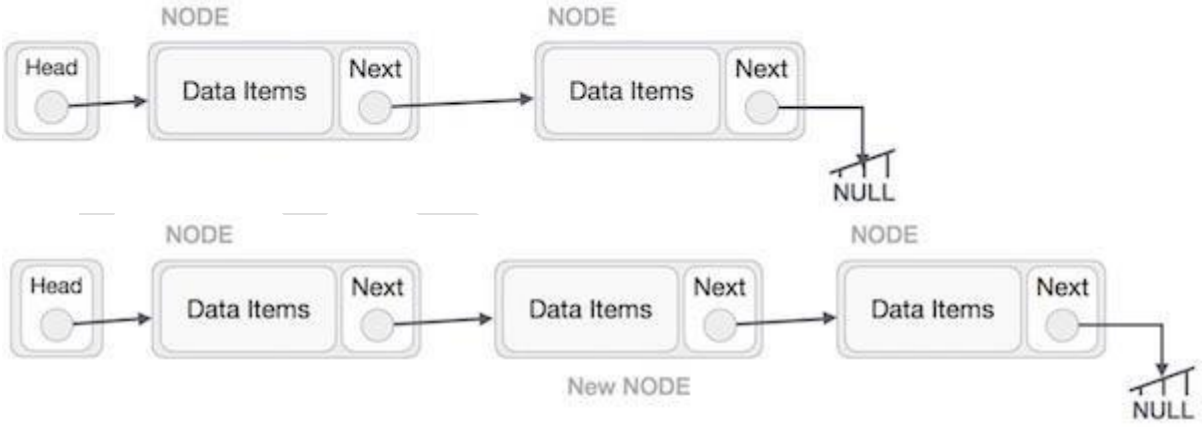
```
LeftNode.next -> TargetNode.next;
```



This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion in linked lists is also performed in three different ways. They are as follows −

Deletion at Beginning

In this deletion operation of the linked, we are deleting an element from the beginning of the list. For this, we point the head to the second node.

Algorithm

```
1. START
2. Assign the head pointer to the next node in the list
3. END
```

Example

Following are the implementations of this operation in various programming languages –

C C++ Java Python

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
```

```
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void deleteatbegin(){
   head = head->next;
}
int main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   insertatbegin(40);
   insertatbegin(55);
   printf("Linked List: ");

   // print list
   printList();
   deleteatbegin();
   printf("\nLinked List after deletion: ");

   // print list
   printList();
}
```

Output

```
Linked List:

[ 55  40  30  22  12 ]

Linked List after deletion:

[ 40  30  22  12 ]
```

Deletion at Ending

In this deletion operation of the linked, we are deleting an element from the ending of the list.

Algorithm

```
1. START
2. Iterate until you find the second last element in the list.
3. Assign NULL to the second last element in the list.
4. END
```

Example

Following are the implementations of this operation in various programming languages –

C C++ Java Python

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");
```

```
    //start from the beginning
    while(p != NULL) {
        printf(" %d ",p->data);
        p = p->next;
    }
    printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

    //create a link
    struct node *lk = (struct node*) malloc(sizeof(struct node));
    lk->data = data;

    // point it to old first node
    lk->next = head;

    //point first to new first node
    head = lk;
}
void deleteatend(){
    struct node *linkedlist = head;
    while (linkedlist->next->next != NULL)
       linkedlist = linkedlist->next;
    linkedlist->next = NULL;
}
void main(){
    int k=0;
    insertatbegin(12);
    insertatbegin(22);
    insertatbegin(30);
    insertatbegin(40);
    insertatbegin(55);
    printf("Linked List: ");
```

```
    // print list
    printList();
    deleteatend();
    printf("\nLinked List after deletion: ");

    // print list
    printList();
}
```

Output

```
Linked List:
[ 55  40  30  22  12 ]
Linked List after deletion:
[ 55  40  30  22 ]
```

Deletion at a Given Position

In this deletion operation of the linked, we are deleting an element at any position of the list.

Algorithm

```
1. START
2. Iterate until find the current node at position in the list.
3. Assign the adjacent node of current node in the list
   to its previous node.
4. END
```

Example

Following are the implementations of this operation in various programming languages –

C C++ Java Python

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
```

```
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void deletenode(int key){
   struct node *temp = head, *prev;
   if (temp != NULL && temp->data == key) {
      head = temp->next;
```

```
        return;
    }

    // Find the key to be deleted
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If the key is not present
    if (temp == NULL) return;

    // Remove the node
    prev->next = temp->next;
}
void main(){
    int k=0;
    insertatbegin(12);
    insertatbegin(22);
    insertatbegin(30);
    insertatbegin(40);
    insertatbegin(55);
    printf("Linked List: ");

    // print list
    printList();
    deletenode(30);
    printf("\nLinked List after deletion: ");

    // print list
    printList();
}
```

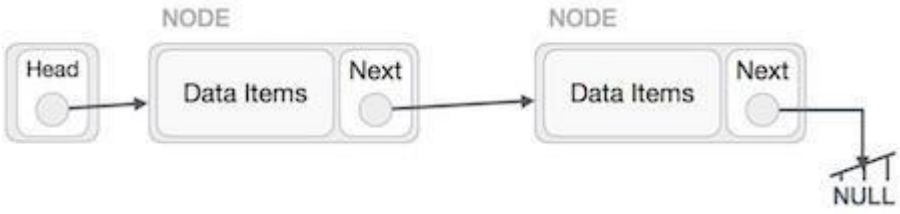
Output

```
Linked List:
[ 55  40  30  22  12 ]
```

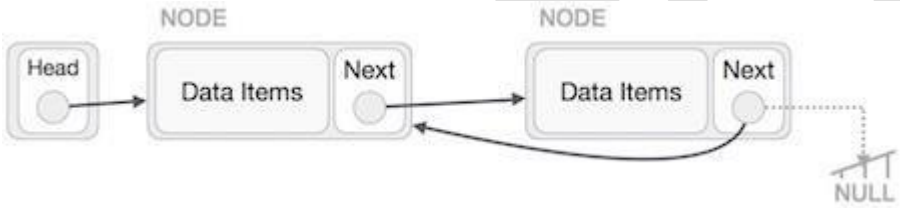
```
Linked List after deletion:

[ 55  40  22  12 ]
```

Linked List - Reversal Operation

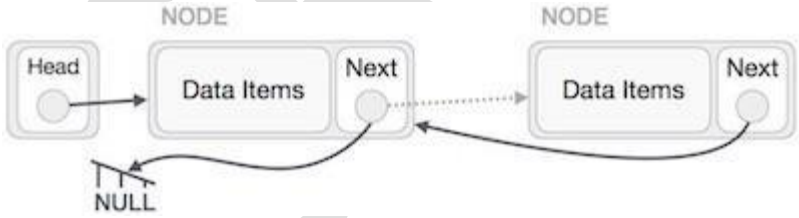
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



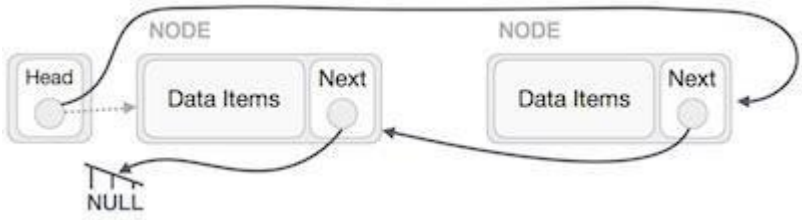
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



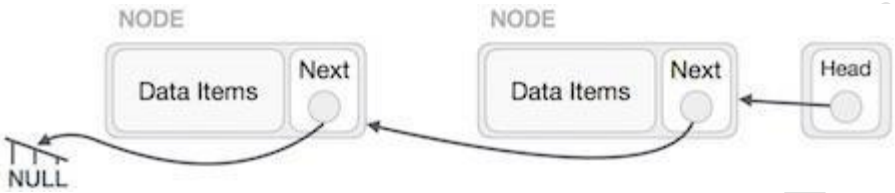
We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



Algorithm

Step by step process to reverse a linked list is as follows –

1. START
2. We use three pointers to perform the reversing:
   prev, next, head.
3. Point the current node to head and assign its next value to
   the prev node.
4. Iteratively repeat the step 3 for all the nodes in the list.
5. Assign head to the prev node.

Example

Following are the implementations of this operation in various programming languages –

C C++ Java Python

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
```

```
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void reverseList(struct node** head){
   struct node *prev = NULL, *cur=*head, *tmp;
   while(cur!= NULL) {
      tmp = cur->next;
      cur->next = prev;
      prev = cur;
      cur = tmp;
```

```
    }
    *head = prev;
}
void main(){
    int k=0;
    insertatbegin(12);
    insertatbegin(22);
    insertatbegin(30);
    insertatbegin(40);
    insertatbegin(55);
    printf("Linked List: ");

    // print list
    printList();
    reverseList(&head);
    printf("\nReversed Linked List: ");
    printList();
}
```

Output

```
Linked List:
[ 55  40  30  22  12 ]
Reversed Linked List:
[ 12  22  30  40  55 ]
```

Linked List - Search Operation

Searching for an element in the list using a key element. This operation is done in the same way as array search; comparing every element in the list with the key element given.

Algorithm

```
1 START
2 If the list is not empty, iteratively check if the list
  contains the key
```

```
3 If the key element is not present in the list, unsuccessful
  search
4 END
```

Example

Following are the implementations of this operation in various programming languages –

C **C++** **Java** **Python**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){
```

```
    //create a link
    struct node *lk = (struct node*) malloc(sizeof(struct node));
    lk->data = data;

    // point it to old first node
    lk->next = head;

    //point first to new first node
    head = lk;
}
int searchlist(int key){
    struct node *temp = head;
    while(temp != NULL) {
        if (temp->data == key) {
            return 1;
        }
        temp=temp->next;
    }
    return 0;
}
void main(){
    int k=0;
    insertatbegin(12);
    insertatbegin(22);
    insertatbegin(30);
    insertatbegin(40);
    insertatbegin(55);
    printf("Linked List: ");

    // print list
    printList();
    int ele = 30;
    printf("\nElement to be searched is: %d", ele);
    k = searchlist(30);
    if (k == 1)
        printf("\nElement is found");
```

```
   else
      printf("\nElement is not found in the list");
}
```

Output

```
Linked List:

[ 55  40  30  22  12 ]

Element to be searched is: 30

Element is found
```

Linked List - Traversal Operation

The traversal operation walks through all the elements of the list in an order and displays the elements in that order.

Algorithm

```
1. START
2. While the list is not empty and did not reach the end of the
list,
   print the data in each node
3. END
```

Example

Following are the implementations of this operation in various programming languages –

C C++ Java Python

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
```

```
};
struct node *head = NULL;
struct node *current = NULL;

   // display the list
   void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

   //insertion at the beginning
   void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   printf("Linked List: ");
```

```
   // print list
   printList();
}
```

Output

```
Linked List:

[ 30  22  12 ]
```

Linked List - Complete implementation

Following are the complete implementations of Linked List in various programming languages −

C **C++** **Java** **Python**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");
   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
```

```
}
//insertion at the beginning
void insertatbegin(int data){
   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;
   // point it to old first node
   lk->next = head;
   //point first to new first node
   head = lk;
}
void insertatend(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;
   struct node *linkedlist = head;

   // point it to old first node
   while(linkedlist->next != NULL)
      linkedlist = linkedlist->next;

   //point first to new first node
   linkedlist->next = lk;
}
void insertafternode(struct node *list, int data){
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;
   lk->next = list->next;
   list->next = lk;
}
void deleteatbegin(){
   head = head->next;
}
void deleteatend(){
   struct node *linkedlist = head;
```

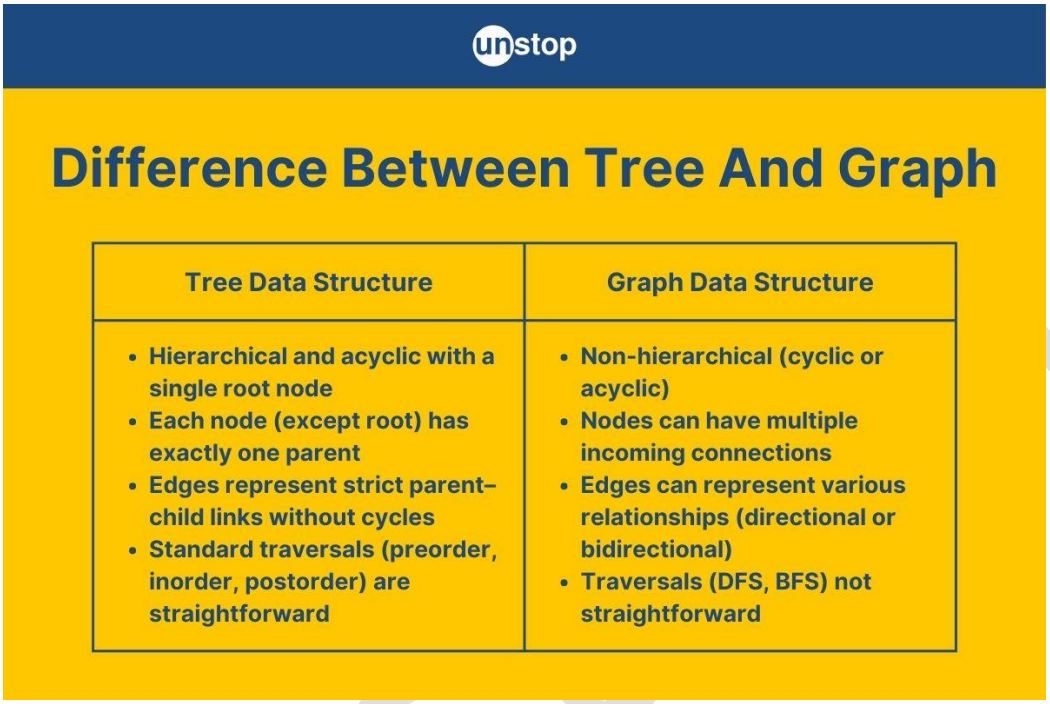
```
    while (linkedlist->next->next != NULL)
        linkedlist = linkedlist->next;
    linkedlist->next = NULL;
}
void deletenode(int key){
    struct node *temp = head, *prev;
    if (temp != NULL && temp->data == key) {
        head = temp->next;
        return;
    }

    // Find the key to be deleted
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If the key is not present
    if (temp == NULL) return;

    // Remove the node
    prev->next = temp->next;
}
int searchlist(int key){
    struct node *temp = head;
    while(temp != NULL) {
        if (temp->data == key) {
            return 1;
        }
        temp=temp->next;
    }
    return 0;
}
void main(){
    int k=0;
    insertatbegin(12);
```

```
    insertatbegin(22);
    insertatend(30);
    insertatend(44);
    insertatbegin(50);
    insertafternode(head->next->next, 33);
    printf("Linked List: ");

    // print list
    printList();
    deleteatbegin();
    deleteatend();
    deleteatend(12);
    printf("\nLinked List after deletion: ");

    // print list
    printList();
    insertatbegin(4);
    insertatbegin(16);
    printf("\nUpdated Linked List: ");
    printList();
    k = searchlist(16);
    if (k == 1)
       printf("\nElement is found");
    else
       printf("\nElement is not present in the list");
}
```

Output

```
Linked List:

[ 50  22  12  33  30  44 ]

Linked List after deletion:

[ 22  33  30 ]

Updated Linked List:
```

```
[ 16  4  22  33  30 ]

Element is found
```



In computer science, data structures are the building blocks that allow us to organize and manage data efficiently. In this article, we explore two of the most fundamental structures, i.e., trees and graphs, and uncover the differences between them. By understanding these concepts, you'll be better equipped to choose the right structure for your application.

Overview Of Trees & Graphs

Trees: A tree is a hierarchical data structure that organizes data in a parent–child relationship.

- It starts with a single root node and branches out into sub-nodes, forming a structure similar to an inverted tree.
- Trees are ideal for representing data with inherent hierarchies, such as organizational charts, file systems, or binary search trees used for fast lookup and sorting.

Graphs: A graph is a flexible, non-linear data structure composed of nodes (vertices) connected by edges.

- Unlike trees, graphs can represent complex relationships where connections are not strictly hierarchical and cycles are allowed.

- Graphs excel in modeling networks such as social networks, transportation systems, or communication networks, where relationships between entities can be many-to-many.

Difference Between Tree And Graph (Comparison Table)

Trees and graphs are both vital data structures, yet they serve different purposes and exhibit distinct characteristics. Understanding the difference between trees and graphs will help you in selecting the right structure based on your application needs.

The table below summarises the key difference between the two data structures:

Aspect	Tree	Graph
Structure	Hierarchical and acyclic with a single root node	Non-hierarchical; can be cyclic or acyclic
Parent–Child Relationship	Each node (except the root) has exactly one parent, ensuring a unique path from the root	Nodes can have multiple incoming connections; no unique path is guaranteed
Edge Characteristics	Edges represent strict parent–child links without cycles	Edges can represent various relationships; may be directional or bidirectional and can form cycles
Traversal Methods	Standard traversals (preorder, inorder, postorder) are straightforward	Traversals (DFS, BFS) require additional mechanisms (e.g., cycle detection)
Memory Representation	Typically implemented with pointers (or arrays for heaps), with moderate overhead	Memory usage varies with representation—adjacency lists for sparse graphs or matrices for dense graphs
Constraints	Must be acyclic and fully connected; each node (except root) has one parent	Fewer structural constraints; graphs can be disconnected, cyclic, or include self-loops
Implementation Complexity	Generally simpler to implement (often with recursive algorithms)	More complex due to arbitrary connectivity and potential cycles

Use Cases	Ideal for hierarchical data like file systems, decision trees, or XML/JSON parsing	Suited for modeling networks such as social networks, transportation, or dependency graphs
Directionality	Implicitly directed from parent to children	Can be directed (digraphs) or undirected, offering greater flexibility
Search and Efficiency	Efficient search in balanced trees (e.g., O(log n) in binary search trees)	Performance depends on the algorithm and graph density; often requires more complex methods

What Is A Tree?

A tree is a hierarchical data structure composed of nodes connected by edges, where each node (except the root) has one parent and zero or more children. This structure naturally represents hierarchical relationships found in many real-world scenarios, such as file systems, organizational charts, and decision-making processes.

Key Features of Trees

- **Hierarchical Structure:** Trees naturally represent data in levels with a single root and subsequent branches.
- **Acyclic Nature:** By definition, trees do not contain cycles, ensuring there is only one unique path from the root to any node.
- **Ordered Relationships:** Trees provide a clear ordering of elements, which makes them useful for tasks like searching and sorting (especially in binary search trees).
- **Versatility in Applications:** Whether representing hierarchical data (like family trees or organizational charts) or supporting efficient algorithms (like tree traversals), trees are fundamental to many computing tasks.

Simple Code Example: Binary Tree Traversal

Below is a short Python snippet that demonstrates an inorder traversal of a simple binary tree:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

```
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.value, end=' ')
        inorder_traversal(root.right)

# Constructing a simple binary tree:
#         4
#        / \
#       2   6
#      / \ / \
#     1  3 5  7

root = Node(4)
root.left = Node(2)
root.right = Node(6)
root.left.left = Node(1)
root.left.right = Node(3)
root.right.left = Node(5)
root.right.right = Node(7)

print("Inorder Traversal of the tree:")
inorder_traversal(root)
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.value, end=' ')
        inorder_traversal(root.right)
```

```
# Constructing a simple binary tree:
# Â  Â  Â  Â  4
#Â  Â  Â  Â  / \
# Â  Â  Â  2 Â  6
#Â  Â  Â  / \ / \
# Â  Â  1Â  3 5Â  7

root = Node(4)
root.left = Node(2)
root.right = Node(6)
root.left.left = Node(1)
root.left.right = Node(3)
root.right.left = Node(5)
root.right.right = Node(7)

print("Inorder Traversal of the tree:")
inorder_traversal(root)
```

In this example, we create a class named Node, wherein we define the structure of each tree node, containing a value and pointers to left and right children.

- Then, we define a function inorder_traversal(,) which uses a recursive approach to traversing the tree in an "inorder" fashion– first visiting the left subtree, then the node itself, followed by the right subtree.
- This concise example illustrates both the hierarchical nature of trees and one of the common ways to process them.

How Is A Tree Represented In Memory?

Trees are typically implemented using one of two main methods:

Approach	Description
Pointer-Based Representation	Each node is a dynamically allocated object or structure that contains the data and pointers (or references) to its child nodes. This method is well-suited for dynamic or irregular tree structures and can handle trees where

	nodes have varying numbers of children (using techniques like left-child right-sibling for general trees).
Array-Based Representation	Nodes are stored sequentially in a contiguous block of memory (an array). In binary trees, parent–child relationships are derived using index calculations (e.g., for a node at index *i*, the left child is at *2i + 1* and the right child at *2i + 2*). This approach works best for complete or nearly complete trees and often improves memory efficiency and cache performance.

For more information on this data structure, its components, and operations, read: What Is Tree Data Structure? Operations, Types & More (+Examples)

What Is A Graph?

A graph is a non-linear data structure that consists of a set of vertices (or nodes) and edges that connect these vertices. Unlike trees, graphs do not enforce a hierarchical structure; instead, they allow for complex relationships, including cycles and multiple connections between nodes. Graphs are ideal for modeling real-world networks such as social networks, transportation systems, or communication networks.

Key Characteristics Of Graphs

- **Vertices (Nodes):** The individual elements or entities in the graph.
- **Edges:** The connections between vertices, which can be directed (showing a one-way relationship) or undirected (showing a two-way relationship).
- **Flexibility:** Graphs can represent a variety of relationship types, including one-to-one, one-to-many, and many-to-many connections.
- **Cycles:** Graphs can contain cycles, meaning you might be able to start at one node and follow a path that eventually loops back to it.

Simple Code Example: Graph Traversal Using DFS

Below is a concise Python snippet that demonstrates a depth-first search (DFS) on a graph represented by an adjacency list:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
```

```
    print(start, end=' ')
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# Graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

print("DFS traversal starting from 'A':")
dfs(graph, 'A')
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=' ')
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# Graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
```

```
    'E': ['F'],
    'F': []
}

print("DFS traversal starting from 'A':")
dfs(graph, 'A')
```

In this example, the graph is represented as a dictionary where each key is a vertex, and its corresponding value is a list of adjacent vertices.

- The dfs function uses recursion to visit each node, ensuring each vertex is processed only once.
- This snippet highlights the flexible, non-hierarchical nature of graphs and provides a practical example of traversing a graph.

Components Of Graph

A graph is built from a few fundamental components that define its structure and behavior:

- **Vertices (Nodes):** The basic units or points in a graph that hold data or represent entities.
- **Edges:** The connections between vertices. Edges illustrate the relationships between nodes. They can be:
    - **Directed:** Indicating a one-way relationship (from one vertex to another).
    - **Undirected:** Representing a two-way, mutual relationship.
- **Weights (Optional):** Some graphs assign weights to edges to represent cost, distance, or capacity, adding an extra dimension for analysis.
- **Adjacency Structure:** How the graph is stored in memory. This can be in the form of:
    - **Adjacency Lists:** Each vertex maintains a list of its neighbors.
    - **Adjacency Matrices:** A 2D array is used where each cell indicates whether a pair of vertices is connected (and possibly the weight of the connection).
- **Subgraphs and Components:** A graph may consist of several connected subgraphs, known as components. In a connected graph, every vertex is reachable from every other vertex; otherwise, the graph is said to be disconnected.

Understanding these components provides the foundation for analyzing more complex graph properties, including the types of edges and the various forms of graphs used in different applications.

For more information of this data structure, its components, and more, read: Graph Data Structure | Types, Algorithms & More (+Code Examples)

Key Differences Between Tree And Graph Explained

While both trees and graphs are fundamental data structures, they differ in several key ways that affect how they are used and implemented. In this section, we will take a deeper look at the differences between tree and graph data structures.

Structural Organization | Difference Between Tree And Graph

Trees: Trees are inherently hierarchical.

- They start with a single root node and branch out into sub-levels.
- Every node (except the root) has exactly one parent, ensuring that there is only one unique path from the root to any given node.
- This structure makes trees especially suitable for representing hierarchies, such as file systems or organizational charts.

Graphs: Graphs are more general and can represent complex relationships.

- They consist of nodes (vertices) and edges that connect any pair of nodes.
- Graphs do not enforce a hierarchical structure; nodes can be interconnected in any way, and there can be multiple paths between nodes.
- This flexibility makes graphs ideal for modeling networks like social networks or transportation systems.

Connectivity and Cycles | Difference Between Tree And Graph

Trees: By definition, trees are acyclic.

- This means that once you traverse from the root to a leaf, you will never revisit a node—ensuring a clear, non-repeating structure.
- The acyclic nature simplifies many operations, such as traversal and search.

Graphs: Graphs may be cyclic or acyclic.

- The presence of cycles means that special care (like using visited markers) is needed during traversals to avoid infinite loops.
- Cycles in graphs enable the representation of more complex interconnections, such as those seen in communication networks or dependency graphs.

Parent–Child Relationship vs. Arbitrary Connections | Difference Between Tree And Graph

Trees: In a tree, every node (aside from the root) has a single parent and a clearly defined set of children. This one-to-many relationship forms a predictable structure that is easy to navigate and manipulate.

Graphs: Graphs allow for arbitrary connections–nodes can have multiple incoming and outgoing edges without a fixed parent-child relationship. This many-to-many connectivity enables the modeling of diverse real-world relationships but can also lead to increased complexity in operations like traversal and search.

Use-Case Scenarios | Difference Between Tree And Graph

Trees: Trees excel in scenarios that require a clear hierarchy and ordered structure. They are widely used in applications like:

- File systems and directory structures.
- Decision trees in machine learning.
- Syntax trees in compilers.
- Binary search trees for efficient sorting and searching.

Graphs: Graphs shine in situations where relationships are complex and not strictly hierarchical. They are used to model:

- Social networks where individuals can be connected in many different ways.
- Road networks and transportation systems.
- Communication and dependency networks.
- Any scenario where cycles and arbitrary relationships are common.

Tree vs. Graphs Practical Applications: A Comparative Look

Both trees and graphs are versatile data structures, but each shines in different contexts. Understanding their practical applications can help you decide which structure best fits a given problem.

Trees In Practice

- **Hierarchical Data Representation:** Trees are naturally suited for data with an inherent hierarchy. For example, file systems, organizational charts, and XML/HTML document structures are commonly represented as trees.

- **Efficient Searching and Sorting:** Binary Search Trees (BSTs) are widely used for fast data retrieval, insertion, and deletion. They enable efficient lookup operations, which are ideal for scenarios like database indexing or maintaining sorted lists.
- **Decision-Making and Parsing:** Decision trees in machine learning help model decision processes, while compilers use Abstract Syntax Trees (ASTs) to parse and analyze source code.

Graphs In Practice

- **Network and Relationship Modeling:** Graphs excel at representing complex relationships where entities are interconnected in non-hierarchical ways. Social networks, communication networks, and transportation systems are prime examples where graphs are used.
- **Routing and Navigation:** Algorithms like Dijkstra's and A* operate on graphs to determine the shortest or most efficient paths. This makes graphs indispensable in mapping applications and network routing.
- **Dependency and Relationship Analysis:** Graphs effectively model dependencies—such as task scheduling, package management systems, or even knowledge graphs—where multiple entities interact in various ways.

Application Area	Tree	Graph
Data Representation	Hierarchical structures (e.g., file systems, org charts)	Complex networks (e.g., social or transportation networks)
Search and Sorting	Binary search trees for fast lookup and sorted order	Not typically used for sorting, but ideal for pathfinding
Decision & Parsing	Decision trees and ASTs in compilers	Useful for modeling interdependencies and relationships
Routing & Navigation	Rarely used	Ideal for implementing routing algorithms (Dijkstra’s, A*)

The choice between trees and graphs in practical applications depends on the nature of your data:

- Use trees when the data is inherently hierarchical and ordered.
- Choose graphs when you need to represent complex, non-linear relationships with potential cycles.

Conclusion

Trees and graphs are both essential data structures with distinct strengths. Trees offer a simple, hierarchical framework that excels in modeling ordered, acyclic relationships, while graphs provide

the flexibility to represent complex, interconnected networks. By understanding the core difference between tree and graph and their practical applications, you can choose the most effective structure for your problem. Whether organizing hierarchical data or mapping intricate networks, selecting the right structure is key to building efficient, scalable systems.

Frequently Asked Questions

Q1. What is the difference between tree and graph search?

The key difference lies in the structure being traversed:

Tree Search	Graph Search
Structure: Trees are hierarchical and acyclic, meaning there is only one unique path from the root to any node.	**Structure:** Graphs can be cyclic and may have multiple paths between nodes. This flexibility means that the structure is less predictable than a tree.
Algorithm Simplicity: Since trees don't contain cycles, tree search algorithms (like preorder, inorder, or postorder traversals) don't need to track visited nodes.	**Cycle Handling:** Graph search algorithms (like depth-first search or breadth-first search) must incorporate mechanisms to track visited nodes, preventing infinite loops and redundant processing.
Redundancy: In trees, duplicate nodes are generally not a concern, as each node appears only once in the structure.	**Complexity:** Due to potential cycles and arbitrary connections, graph search can be more complex and may require additional considerations such as weights or the directionality of edges.

In summary, while tree search is straightforward due to the acyclic, hierarchical nature of trees, graph search demands extra care to handle cycles and multiple paths effectively.

Q2. When should I use a tree over a graph?

Use trees when your data has an inherent hierarchical structure—for example, in file systems, organizational charts, or decision trees. Trees are optimal for operations that require an ordered, acyclic relationship. Graphs are preferable when relationships are complex and not strictly hierarchical, such as in social networks, road maps, or dependency networks.

Q3. What are the common operations performed on trees and graphs?

For trees, common operations include:

- Insertion and Deletion: Adding or removing nodes while preserving the hierarchy.
- Traversals: Inorder, preorder, postorder (for binary trees), or level-order traversals.
- Searching: Efficient lookup in structures like binary search trees (BSTs).

For graphs, typical operations include:

- Search Algorithms: Depth-first search (DFS) and breadth-first search (BFS) with cycle detection.
- Shortest Path: Algorithms like Dijkstra's or A* for finding the best route.
- Connectivity Analysis: Determining connected components or detecting cycles.

Q4. How is a tree represented in memory compared to a graph?

Trees are commonly implemented using either:

- **Pointer-Based Representations:** Each node is an object with data and pointers to its children.
- **Array-Based Representations:** Often used for complete binary trees (e.g., heaps), where relationships are calculated via indices.

Graphs, on the other hand, are typically represented using:

- **Adjacency Lists:** Each node stores a list of its adjacent nodes, ideal for sparse graphs.
- **Adjacency Matrices:** A 2D array where each cell indicates whether a pair of nodes is connected, which works well for dense graphs.

Q5. What is a binary search tree (BST), and how does it differ from a general binary tree?

A BST is a specialized binary tree where each node's left subtree contains only nodes with keys less than the node's key, and the right subtree contains only nodes with keys greater than the node's key. This ordering allows for efficient searching, insertion, and deletion operations (typically in O(log n) time). A general binary tree, however, does not enforce this ordering, which means it may not support efficient search operations.

Q6. Can trees or graphs be used for both searching and sorting?

Trees, especially binary search trees, are particularly well-suited for both searching and sorting due to their ordered structure. Traversing a BST in-order produces sorted output. Graphs are generally used for modeling relationships and connectivity rather than sorting, although graph search algorithms are essential for exploring networks and finding optimal paths.

By now, you must have a clear idea of the difference between tree and graph and what works for you. Do explore more interesting topics:

1. What Is Linear Data Structure? Types, Uses & More (+ Examples)
2. Single Linked List In Data Structure | All Operations (+Examples)
3. Doubly Linked List Data Structure | All Operations Explained (+Codes)
4. Time Complexity Of Algorithms: Types, Notations, Cases, and More
5. DSA Cheatsheet Unlocked: Roadmap, Boot Camp & Must-Know Resources

Hashing in Data Structure

Hashing is a data structure, where we can store the data and look up that data very quickly. Hashing uses a special formula called a hash function to map data to a location in the data structure.

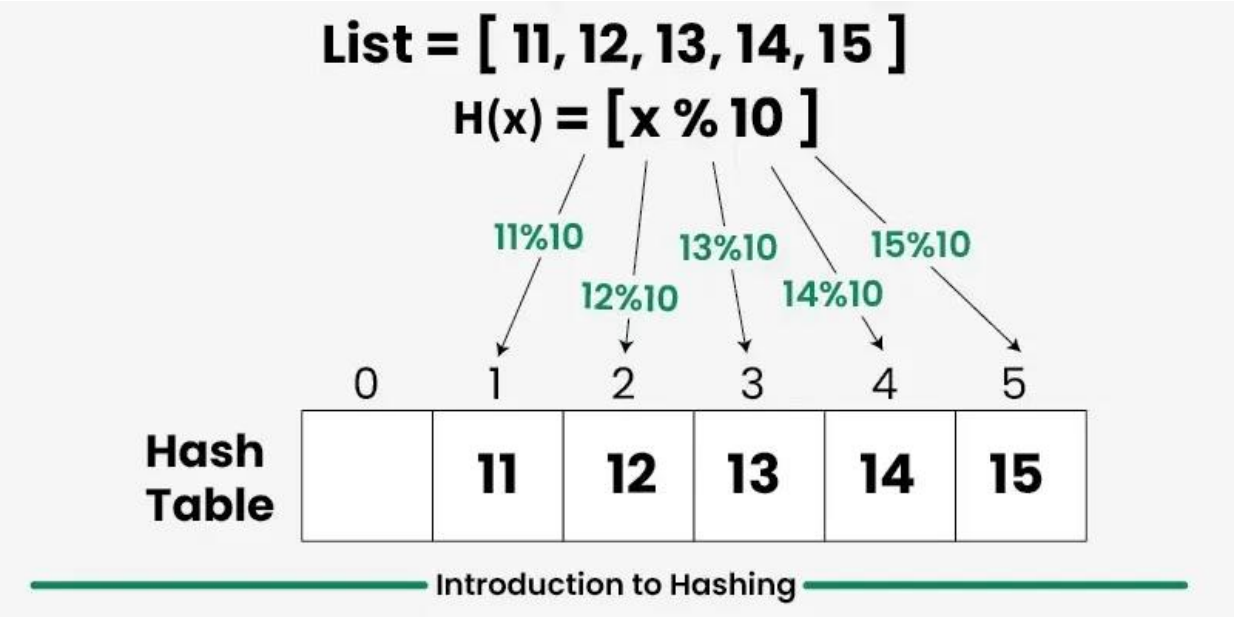
The hash function takes the data as input and returns an index in the data structure where the data should be stored. This allows us to quickly retrieve the data by using the hash function to calculate the index and then looking up the data at that index.

Imagine you have a list of names and you want to look up a specific name in the list. You could use a hash function to map the name to an index in a data structure, such as an array or a hash table. This allows you to quickly find the name in the data structure without having to search through the entire list.

Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access.

- Hashing involves mapping data to a specific index in a hash table (an array of items) using a hash function. It enables fast retrieval of information based on its key.
- The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time on average.
- Hashing is mainly used to implement a set of distinct items (only keys) and dictionaries (key value pairs).

Here's an example of hashing using the modulo method. The hash function H(x) = x % 10 converts any large number into a smaller value between 0 and 9.



What is Hash Function?

A hash function is a mathematical function, which takes an input and returns a fixed size string of bytes. The output of the hash function is called a hash value or hash code. The hash function is designed to be fast and efficient, so that it can quickly calculate the hash value for a given input.

Hash functions have several important properties:

These are deterministic meaning that the same input will always produce the same output.

They can quickly calculate the hash value for a given input.

Hash functions are secure, meaning that it is difficult to reverse-engineer the input from the hash value.

It has a fixed output size, so the hash value is always the same length.

What is Hash Table?

A hash table is a data structure that make use of hash function to map keys to values. It consists of an array of buckets, where each bucket stores a key-value pair.

The hash function is used to calculate the index of the bucket where the key-value pair should be stored. This allows us to quickly retrieve the value associated with a key by using the hash function to calculate the index and then looking up the value at that index.

Properties of Hash Table

Hash tables have several important properties:

They provide fast lookups, insertions, and deletions, with an average time complexity of O(1).

They can store a large amount of data easily, with a space complexity of O(n).

It can handle collisions, where two keys map to the same index, by using techniques like chaining or open addressing.

Hash Table Data Structure Overview

- It is one of the most widely used data structure after arrays.
- It mainly supports search, insert and delete in O(1) time on average which is more efficient than other popular data structures like arrays, Linked List and Self Balancing BST.
- We use hashing for dictionaries, frequency counting, maintaining data for quick access by key, etc.
- Real World Applications include Database Indexing, Cryptography, Caches, Symbol Table and Dictionaries.
- There are mainly two forms of hash typically implemented in programming languages.
  Hash Set : Collection of unique keys (Implemented as Set in Python, Set in JavaScrtipt, unordered_set in C++ and HashSet in Java.
  Hash Map : Collection of key value pairs with keys being unique (Implemented as dictionary in Python, Map in JavaScript, unordered_map in C++ and HashMap in Java)

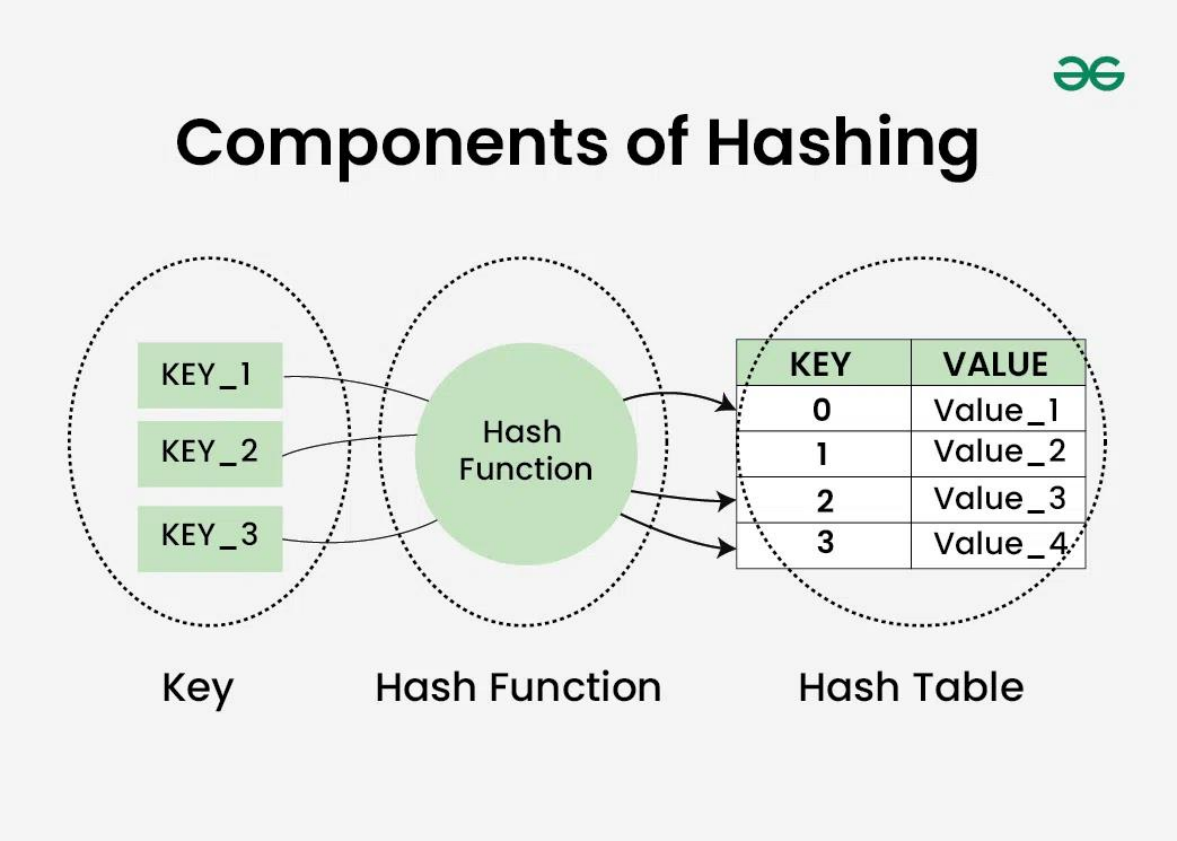
Situations Where Hash is not Used

- Need to maintain sorted data along with search, insert and delete. We use a self balancing BST in these cases.
- When Strings are keys and we need operations like prefix search along with search, insert and delete. We use Trie in these cases.
- When we need operations like floor and ceiling along with search, insert and/or delete. We use Self Balancing BST in these cases.
-

Components of Hashing

There are majorly three components of hashing:

1. **Key:** A Key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** Receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index .
3. **Hash Table:** Hash table is typically an array of lists. It stores values corresponding to the keys. Hash stores the data in an associative manner in an array where each data value has its own unique index.



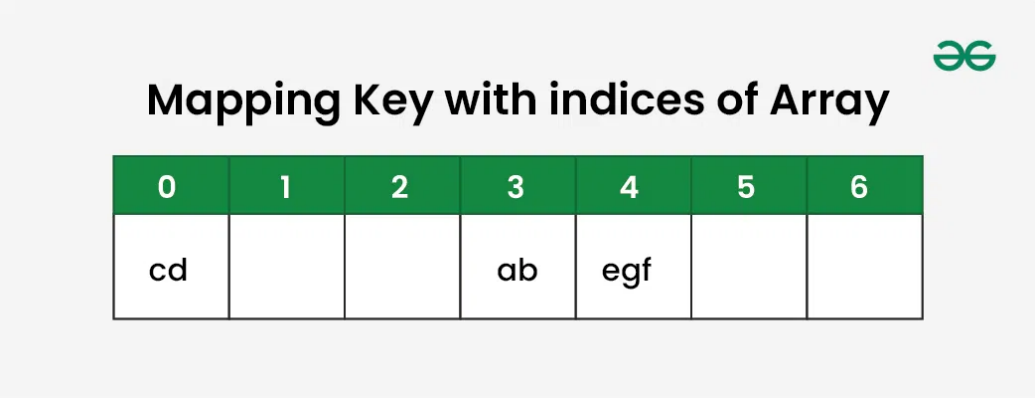
How does Hashing work?

Suppose we have a set of strings {“ab”, “cd”, “efg”} and we would like to store it in a table.

- **Step 1:** We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.
- **Step 2:** So, let's assign
    - “a” = 1,
    - “b”=2, .. etc, to all alphabetical characters.
- **Step 3:** Therefore, the numerical value by summation of all characters of the string:

- *“ab” = 1 + 2 = 3,*
- *“cd” = 3 + 4 = 7 ,*
- *“efg” = 5 + 6 + 7 = 18*

- **Step 4:** Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size** . We can compute the location of the string in the array by taking the **sum(string) mod 7** .
- **Step 5:** So we will then store
    - “ab” in 3 mod 7 = 3,
    - “cd” in 7 mod 7 = 0, and
    - “efg” in 18 mod 7 = 4.



The above technique enables us to calculate the location of a given string by using a simple hash function and rapidly find the value that is stored in that location. Therefore the idea of hashing seems like a great way to store (key, value) pairs of the data in a table.

Collision in Hashing

Collision in hashing occurs when we get similar output values, or rather hash values, for different input values. This can happen due to the limited range of hash values or the nature of the hash function.

Applications of Hashing

Hashing is used in various applications in computer science, such as:

- Storing and retrieving data quickly.
- Implementing data structures like hash tables and hash maps.
- Searching and indexing data efficiently.

- Ensuring data integrity and security.
- Password hashing and encryption.

Separate Chaining Collision Handling Technique in Hashing

Last Updated : 24 Jul, 2025

- Separate Chaining is a collision handling technique. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions. In this article, we will discuss about what is Separate Chain collision handling technique, its advantages, disadvantages, etc.
  There are mainly two methods to handle collision:
  - Separate Chaining
  - Open Addressing

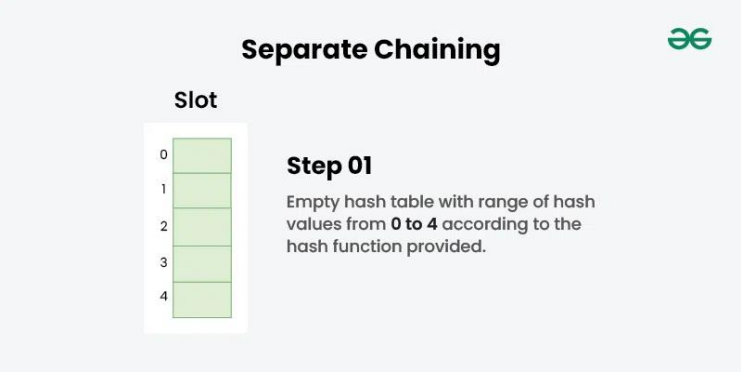
  In this article, only separate chaining is discussed. We will be discussing Open addressing in the next post

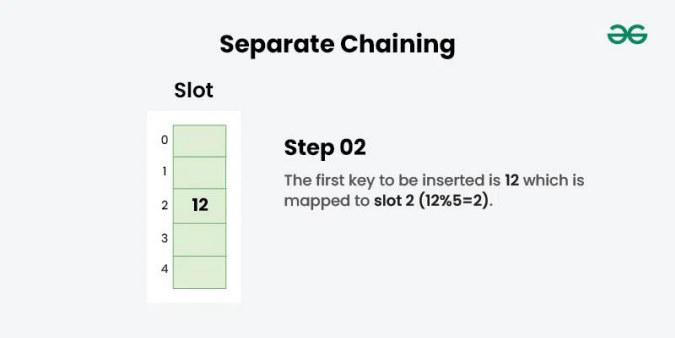
Separate Chaining:

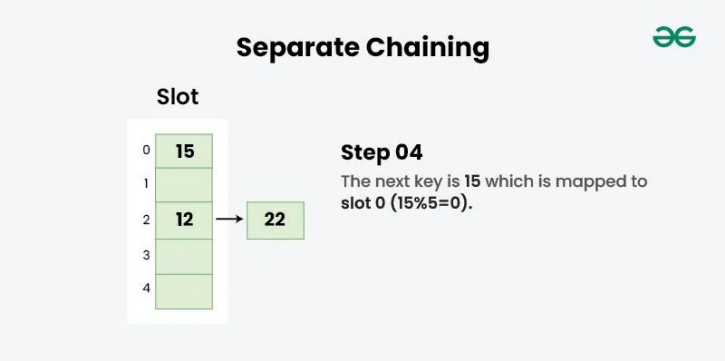
The idea behind separate chaining is to implement the array as a linked list called a chain.

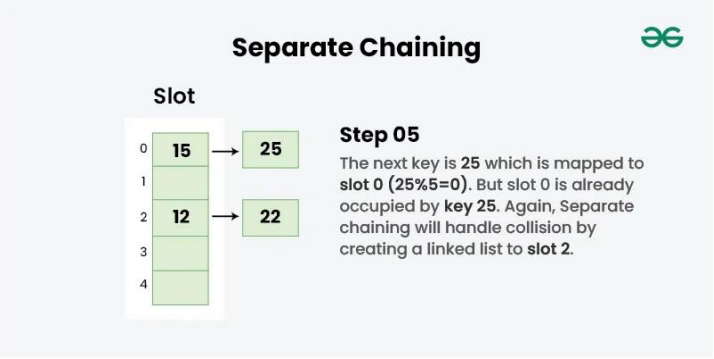
Linked List (or a Dynamic Sized Array) *is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.*
Here, all those elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to K then it means that we have found our entry. If we have reached the end of the linked list and yet we haven't found our entry then it means that the entry does not exist. Hence, the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

Example: Let us consider a simple hash function as "**key mod 5**" and a sequence of keys as 12, 22, 15, 25









.

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.

- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

Performance of Chaining:

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing).

m = Number of slots in hash table
n = Number of keys to be inserted in hash table
Load factor α = n/m
Expected time to search = O(1 + α)
Expected time to delete = O(1 + α)
Time to insert = O(1)
Time complexity of search insert and delete is O(1) if α is O(1)

Data Structures For Storing Chains:

Below are different options to create chains.

- The advantage of linked list implementation is insert is O(1) in the worst case.
- The advantage of array is cache friendliness, but the insert operation can be O(1) in cases when we have to resize the array.
- The advantage of Self Balancing BST is the worst case is bounded by O(Log (len)) for all operations

1. Linked lists

- Search: O(len) where len = length of chain
- Delete: O(len)
- Insert: O(1)
- Not cache friendly

2. Dynamic Sized Arrays (Vectors in C++, ArrayList in Java, list in Python)

- Search: O(len) where len = length of chain
- Delete: O(len)
- Insert: O(1)
- Cache friendly

3. Self Balancing BST (AVL Trees, Red-Black Trees)

- Search: O(log(len)) where len = length of chain
- Delete: O(log(len))

- Insert: O(log(len))
- Not cache friendly

Open Addressing Collision Handling technique in Hashing

- **Open Addressing** is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:
  - ***Insert(k):*** *Keep probing until an empty slot is found. Once an empty slot is found, insert k.*
  - ***Search(k):*** *Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.*
  - ***Delete(k)**: **Delete operation is interesting**. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.*

Different ways of Open Addressing:

1. Linear Probing:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

The function used for rehashing is as follows: rehash(key) = (n+1)%table-size.
For example, The typical gap between two probes is 1 as seen in the example below:
*Let **hash(x)** be the slot index computed using a hash function and **S** be the table size*
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
Example: Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

2. Quadratic Probing

If you observe carefully, then you will understand that the interval between probes will increase proportionally to the hash value. Quadratic probing is a method with the help of which we can solve the problem of clustering that was discussed

above. This method is also known as the **mid-square** method. In this method, we look for the **i^2'th** slot in the **i^{th}** iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.
let hash(x) be the slot index computed using hash function.
*If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S*
*If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S*
*If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S*
Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be $f(i) = i^2$. Insert = 22, 30, and 50.

3. Double Hashing

The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **i^{th}** rotation.
let hash(x) be the slot index computed using hash function.
*If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S*
*If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S*
*If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S*
Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

Comparison of the above three:

Open addressing is a collision handling technique used in hashing where, when a collision occurs (i.e., when two or more keys map to the same slot), the algorithm looks for another empty slot in the hash table to store the collided key.

- In **linear probing**, the algorithm simply looks for the next available slot in the hash table and places the collided key there. If that slot is also occupied, the algorithm continues searching for the next available slot until an empty slot is found. This process is repeated until all collided keys have been stored. Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
- In **quadratic probing**, the algorithm searches for slots in a more spaced-out manner. When a collision occurs, the algorithm looks for the next slot using an

equation that involves the original hash value and a quadratic function. If that slot is also occupied, the algorithm increments the value of the quadratic function and tries again. This process is repeated until an empty slot is found. Quadratic probing lies between the two in terms of cache performance and clustering.

- In **double hashing**, the algorithm uses a second hash function to determine the next slot to check when a collision occurs. The algorithm calculates a hash value using the original hash function, then uses the second hash function to calculate an offset. The algorithm then checks the slot that is the sum of the original hash value and the offset. If that slot is occupied, the algorithm increments the offset and tries again. This process is repeated until an empty slot is found. Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

The choice of collision handling technique can have a significant impact on the performance of a hash table. Linear probing is simple and fast, but it can lead to clustering (i.e., a situation where keys are stored in long contiguous runs) and can degrade performance. Quadratic probing is more spaced out, but it can also lead to clustering and can result in a situation where some slots are never checked. Double hashing is more complex, but it can lead to more even distribution of keys and can provide better performance in some cases.

S.No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.

S.No.	Separate Chaining	Open Addressing
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Note: Cache performance of chaining is not good because when we traverse a Linked List, we are basically jumping from one node to another, all across the computer's memory. For this reason, the CPU cannot cache the nodes which aren't visited yet, this doesn't help us. But with Open Addressing, data isn't spread, so if the CPU detects that a segment of memory is constantly being accessed, it gets cached for quick access.

Performance of Open Addressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

m = Number of slots in the hash table
n = Number of keys to be inserted in the hash table
Load factor α = n/m (< 1)
Expected time to search/insert/delete < 1/(1 - α)
So Search, Insert and Delete take (1/(1 - α)) time

Hashing in Data Structure

Hashing is a data structure, where we can store the data and look up that data very quickly. Hashing uses a special formula called a hash function to map data to a location in the data structure.

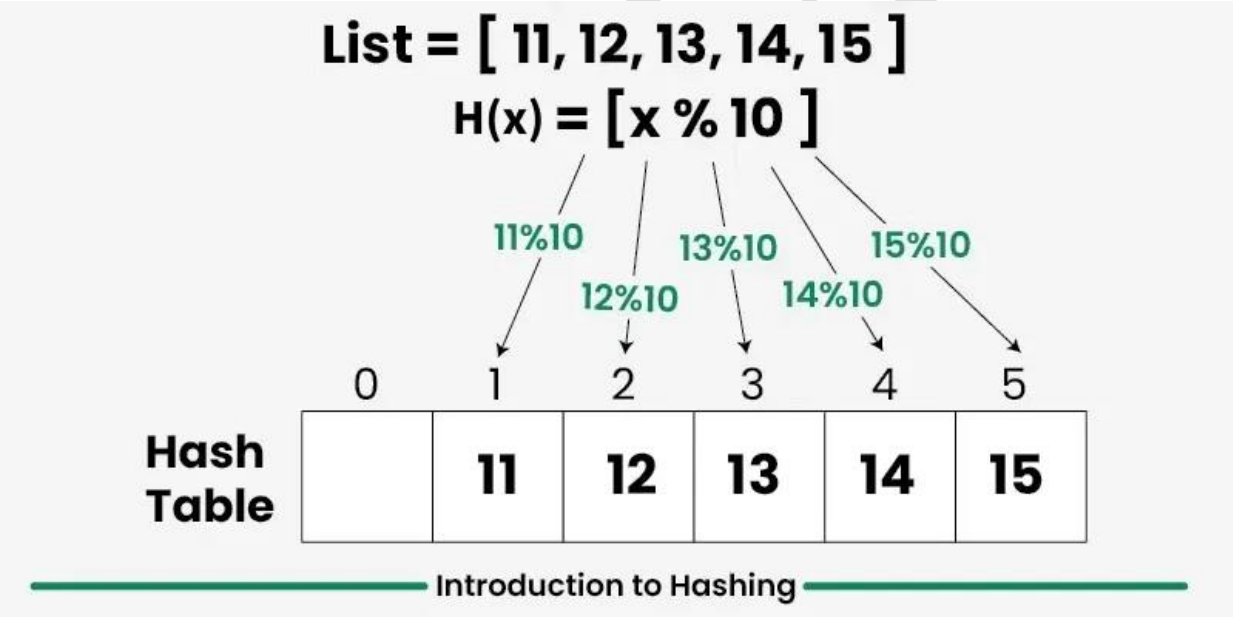
The hash function takes the data as input and returns an index in the data structure where the data should be stored. This allows us to quickly retrieve the data by using the hash function to calculate the index and then looking up the data at that index.

Imagine you have a list of names and you want to look up a specific name in the list. You could use a hash function to map the name to an index in a data structure, such as an array or a hash table. This allows you to quickly find the name in the data structure without having to search through the entire list.

Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access.

- Hashing involves mapping data to a specific index in a hash table (an array of items) using a hash function. It enables fast retrieval of information based on its key.
- The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time on average.
- Hashing is mainly used to implement a set of distinct items (only keys) and dictionaries (key value pairs).

Here's an example of hashing using the modulo method. The hash function H(x) = x % 10 converts any large number into a smaller value between 0 and 9.



What is Hash Function?

A hash function is a mathematical function, which takes an input and returns a fixed size string of bytes. The output of the hash function is called a hash value or hash code. The hash function is designed to be fast and efficient, so that it can quickly calculate the hash value for a given input.

Hash functions have several important properties:

These are deterministic meaning that the same input will always produce the same output.

They can quickly calculate the hash value for a given input.

Hash functions are secure, meaning that it is difficult to reverse-engineer the input from the hash value.

It has a fixed output size, so the hash value is always the same length.

What is Hash Table?

A hash table is a data structure that make use of hash function to map keys to values. It consists of an array of buckets, where each bucket stores a key-value pair.

The hash function is used to calculate the index of the bucket where the key-value pair should be stored. This allows us to quickly retrieve the value associated with a key by using the hash function to calculate the index and then looking up the value at that index.

Properties of Hash Table

Hash tables have several important properties:

They provide fast lookups, insertions, and deletions, with an average time complexity of O(1).

They can store a large amount of data easily, with a space complexity of O(n).

It can handle collisions, where two keys map to the same index, by using techniques like chaining or open addressing.

Hash Table Data Structure Overview

- It is one of the most widely used data structure after arrays.
- It mainly supports search, insert and delete in O(1) time on average which is more efficient than other popular data structures like arrays, Linked List and Self Balancing BST.
- We use hashing for dictionaries, frequency counting, maintaining data for quick access by key, etc.
- Real World Applications include Database Indexing, Cryptography, Caches, Symbol Table and Dictionaries.
- There are mainly two forms of hash typically implemented in programming languages.
  Hash Set : Collection of unique keys (Implemented as Set in Python, Set in JavaScrtipt, unordered_set in C++ and HashSet in Java.
  Hash Map : Collection of key value pairs with keys being unique (Implemented as dictionary in Python, Map in JavaScript, unordered_map in C++ and HashMap in Java)

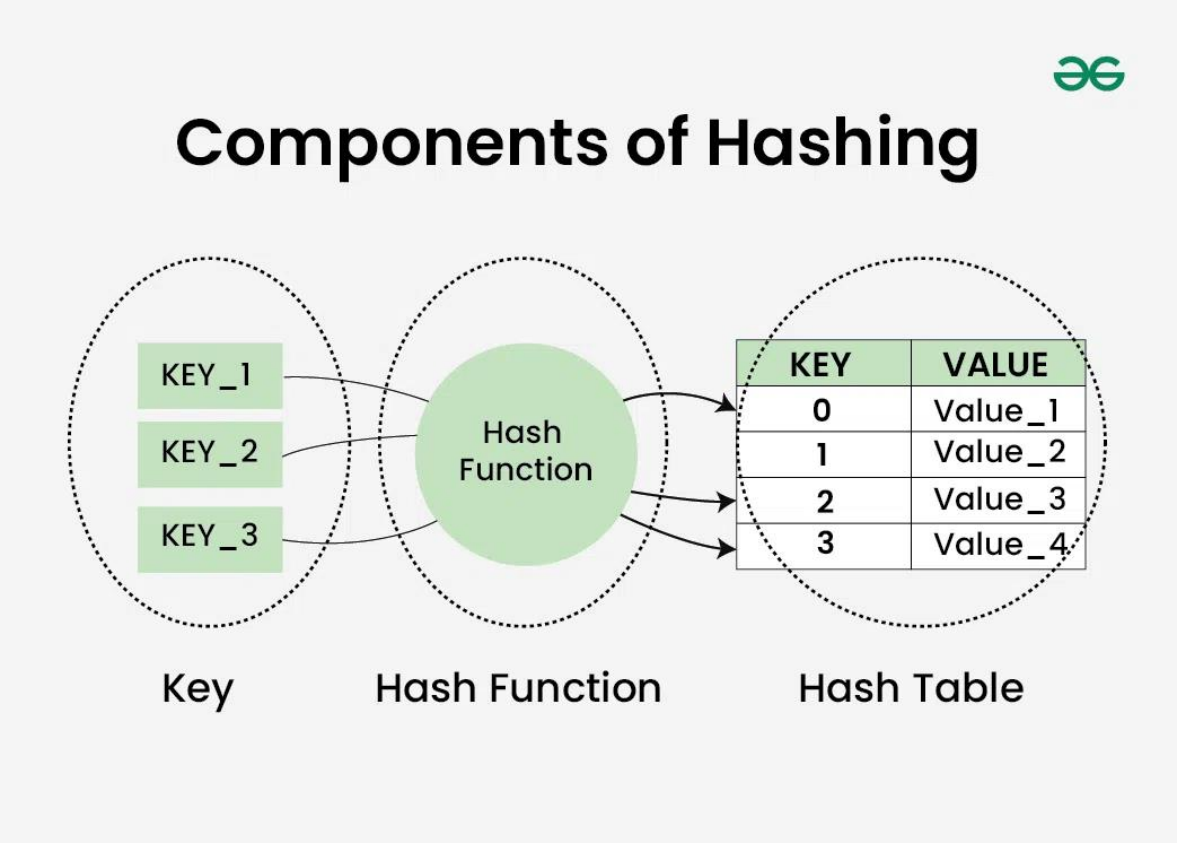
Situations Where Hash is not Used

- Need to maintain sorted data along with search, insert and delete. We use a self balancing BST in these cases.
- When Strings are keys and we need operations like prefix search along with search, insert and delete. We use Trie in these cases.
- When we need operations like floor and ceiling along with search, insert and/or delete. We use Self Balancing BST in these cases.
-

Components of Hashing

There are majorly three components of hashing:

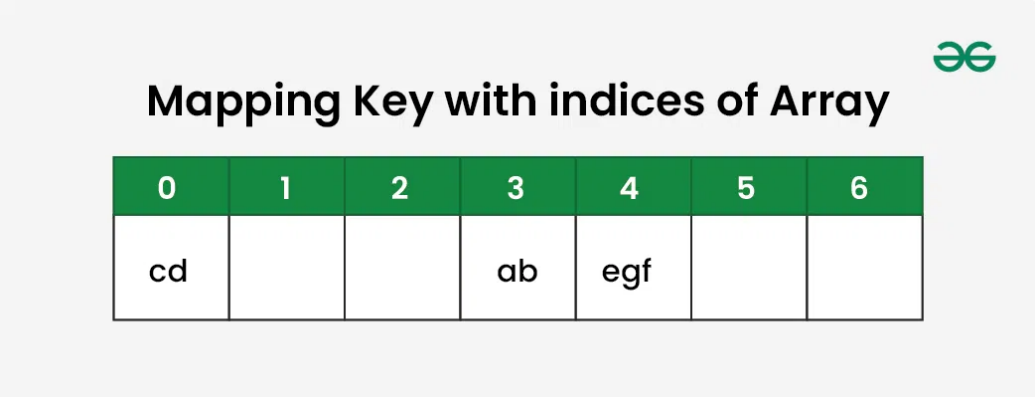
4. **Key:** A Key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
5. **Hash Function:** Receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index .
6. **Hash Table:** Hash table is typically an array of lists. It stores values corresponding to the keys. Hash stores the data in an associative manner in an array where each data value has its own unique index.



How does Hashing work?

Suppose we have a set of strings {“ab”, “cd”, “efg”} and we would like to store it in a table.

- **Step 1:** We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.
- **Step 2:** So, let's assign
    - “a” = 1,
    - “b”=2, .. etc, to all alphabetical characters.
- **Step 3:** Therefore, the numerical value by summation of all characters of the string:
- *“ab” = 1 + 2 = 3,*
- *“cd” = 3 + 4 = 7 ,*
- *“efg” = 5 + 6 + 7 = 18*
- **Step 4:** Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size** . We can compute the location of the string in the array by taking the **sum(string) mod 7** .
- **Step 5:** So we will then store
    - “ab” in 3 mod 7 = 3,
    - “cd” in 7 mod 7 = 0, and
    - “efg” in 18 mod 7 = 4.



The above technique enables us to calculate the location of a given string by using a simple hash function and rapidly find the value that is stored in that location. Therefore the idea of hashing seems like a great way to store (key, value) pairs of the data in a table.

Collision in Hashing

Collision in hashing occurs when we get similar output values, or rather hash values, for different input values. This can happen due to the limited range of hash values or the nature of the hash function.

Applications of Hashing

Hashing is used in various applications in computer science, such as:

- Storing and retrieving data quickly.
- Implementing data structures like hash tables and hash maps.
- Searching and indexing data efficiently.
- Ensuring data integrity and security.
- Password hashing and encryption.

Separate Chaining Collision Handling Technique in Hashing

Last Updated : 24 Jul, 2025

- Separate Chaining is a collision handling technique. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions. In this article, we will discuss about what is Separate Chain collision handling technique, its advantages, disadvantages, etc.
  There are mainly two methods to handle collision:
  - Separate Chaining
  - Open Addressing

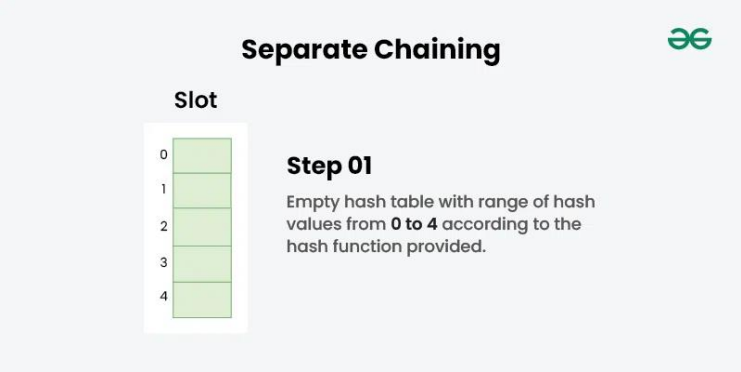
  In this article, only separate chaining is discussed. We will be discussing Open addressing in the next post

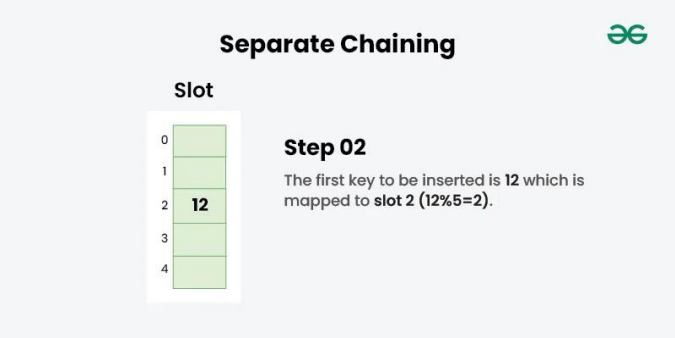
Separate Chaining:

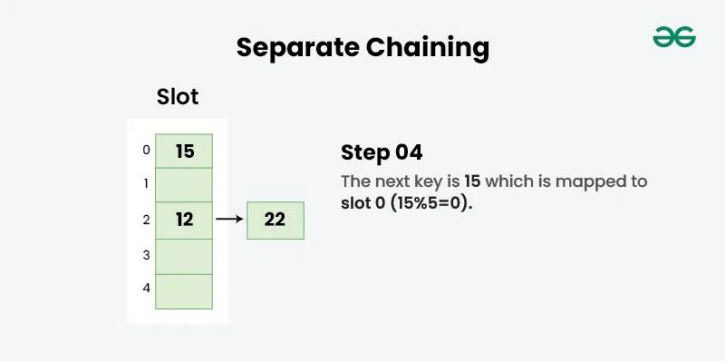
The idea behind separate chaining is to implement the array as a linked list called a chain.

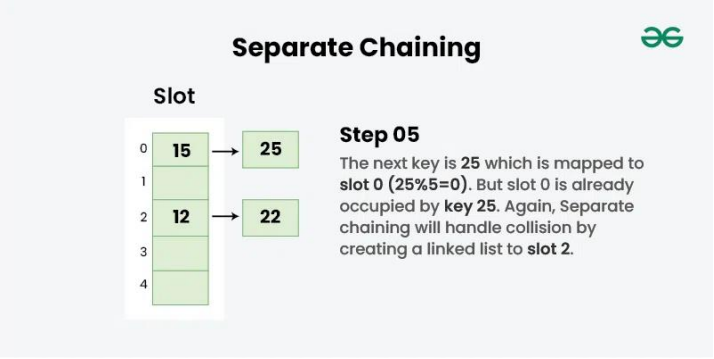
Linked List (or a Dynamic Sized Array) *is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.*
Here, all those elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to K then it means that we have found our entry. If we have reached the end of the linked list and yet we haven't found our entry then it means that the entry does not exist. Hence, the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

Example: Let us consider a simple hash function as "**key mod 5**" and a sequence of keys as 12, 22, 15, 25









.

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.

- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

Performance of Chaining:

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing).

m = Number of slots in hash table
n = Number of keys to be inserted in hash table
Load factor α = n/m
Expected time to search = O(1 + α)
Expected time to delete = O(1 + α)
Time to insert = O(1)
Time complexity of search insert and delete is O(1) if α is O(1)

Data Structures For Storing Chains:

Below are different options to create chains.

- The advantage of linked list implementation is insert is O(1) in the worst case.
- The advantage of array is cache friendliness, but the insert operation can be O(1) in cases when we have to resize the array.
- The advantage of Self Balancing BST is the worst case is bounded by O(Log (len)) for all operations

1. Linked lists

- Search: O(len) where len = length of chain
- Delete: O(len)
- Insert: O(1)
- Not cache friendly

2. Dynamic Sized Arrays (Vectors in C++, ArrayList in Java, list in Python)

- Search: O(len) where len = length of chain
- Delete: O(len)
- Insert: O(1)
- Cache friendly

3. Self Balancing BST (AVL Trees, Red-Black Trees)

- Search: O(log(len)) where len = length of chain
- Delete: O(log(len))

- Insert: O(log(len))
- Not cache friendly

Open Addressing Collision Handling technique in Hashing

- **Open Addressing** is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:
  - ***Insert(k):*** *Keep probing until an empty slot is found. Once an empty slot is found, insert k.*
  - ***Search(k):*** *Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.*
  - ***Delete(k)**: **Delete operation is interesting**. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.*

Different ways of Open Addressing:

1. Linear Probing:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

The function used for rehashing is as follows: rehash(key) = (n+1)%table-size.
For example, The typical gap between two probes is 1 as seen in the example below:
*Let **hash(x)** be the slot index computed using a hash function and **S** be the table size*
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
Example: Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

2. Quadratic Probing

If you observe carefully, then you will understand that the interval between probes will increase proportionally to the hash value. Quadratic probing is a method with the help of which we can solve the problem of clustering that was discussed

above. This method is also known as the **mid-square** method. In this method, we look for the **i^2'th** slot in the **i^{th}** iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.
let hash(x) be the slot index computed using hash function.
*If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S*
*If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S*
*If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S*
Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be $f(i) = i^2$. Insert = 22, 30, and 50.

3. Double Hashing

The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **i^{th}** rotation.
let hash(x) be the slot index computed using hash function.
*If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S*
*If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S*
*If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S*
Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

Comparison of the above three:

Open addressing is a collision handling technique used in hashing where, when a collision occurs (i.e., when two or more keys map to the same slot), the algorithm looks for another empty slot in the hash table to store the collided key.

- In **linear probing**, the algorithm simply looks for the next available slot in the hash table and places the collided key there. If that slot is also occupied, the algorithm continues searching for the next available slot until an empty slot is found. This process is repeated until all collided keys have been stored. Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
- In **quadratic probing**, the algorithm searches for slots in a more spaced-out manner. When a collision occurs, the algorithm looks for the next slot using an

equation that involves the original hash value and a quadratic function. If that slot is also occupied, the algorithm increments the value of the quadratic function and tries again. This process is repeated until an empty slot is found. Quadratic probing lies between the two in terms of cache performance and clustering.

- In **double hashing**, the algorithm uses a second hash function to determine the next slot to check when a collision occurs. The algorithm calculates a hash value using the original hash function, then uses the second hash function to calculate an offset. The algorithm then checks the slot that is the sum of the original hash value and the offset. If that slot is occupied, the algorithm increments the offset and tries again. This process is repeated until an empty slot is found. Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

The choice of collision handling technique can have a significant impact on the performance of a hash table. Linear probing is simple and fast, but it can lead to clustering (i.e., a situation where keys are stored in long contiguous runs) and can degrade performance. Quadratic probing is more spaced out, but it can also lead to clustering and can result in a situation where some slots are never checked. Double hashing is more complex, but it can lead to more even distribution of keys and can provide better performance in some cases.

S.No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.

S.No.	Separate Chaining	Open Addressing
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Note: Cache performance of chaining is not good because when we traverse a Linked List, we are basically jumping from one node to another, all across the computer's memory. For this reason, the CPU cannot cache the nodes which aren't visited yet, this doesn't help us. But with Open Addressing, data isn't spread, so if the CPU detects that a segment of memory is constantly being accessed, it gets cached for quick access.

Performance of Open Addressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

m = Number of slots in the hash table
n = Number of keys to be inserted in the hash table
Load factor α = n/m (< 1)
Expected time to search/insert/delete < 1/(1 - α)
So Search, Insert and Delete take (1/(1 - α)) time