هيكلة البيانات

المرحلة الثانية

ا. م. د. عبدالناصر يونس احمد

مدرسو العملي:

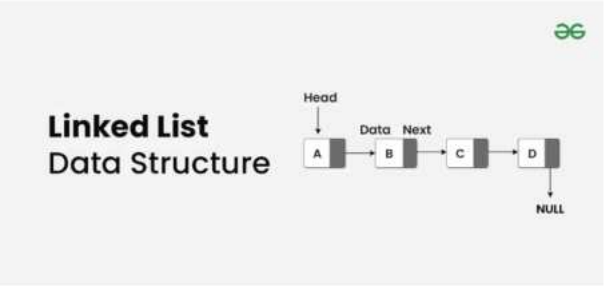
م. اسراء عبدالسلام

م. رغد حازم

Linked Lists (1)

Introduction to Linked List

Linked List is basically **chains of nodes** where each node contains information such as **data** and a **pointer to the next node** in the chain. It is a popular data structure with a wide range of real-world applications. Unlike Arrays, Linked List elements are not stored at a contiguous location. In the linked list there is a **head pointer**, which points to the first element of the linked list, and if the list is empty then it simply points to null or nothing.



Basic Terminologies of Linked List

- **Head:** The Head of a linked list is a pointer to the first node or reference of the first node of linked list. This pointer marks the beginning of the linked list.
- **Node:** Linked List consists of a series of nodes where each node has two parts: **data** and **next pointer**.
- **Data:** Data is the part of node which stores the information in the linked list.
- **Next pointer:** Next pointer is the part of the node which points to the next node of the linked list.

Importance of Linked List

- **Dynamic Data structure:** The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
- **Ease of Insertion/Deletion:** The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
- **Efficient Memory Utilization:** As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
- **Implementation:** Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

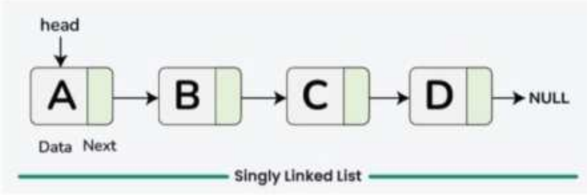
The common variations of linked lists are Singly, Doubly, Singly Circular and Doubly Circular.

Singly Linked List

A **singly linked list** is a fundamental data structure in computer science and programming, it consists of **nodes** where each node contains a **data** field and a **reference** to the next node in the node. The last node points to **null**, indicating the end of the list. This linear structure supports efficient insertion and deletion operations, making it widely used in various applications. In this tutorial, we'll explore the node structure, understand the operations on singly linked lists (traversal, searching, length determination, insertion, and deletion), and provide detailed explanations and code examples to implement these operations effectively.

Understanding Node Structure

In a singly linked list, each node consists of two parts: data and a pointer to the next node. The data part stores the actual information, while the pointer (or reference) part stores the address of the next node in the sequence. This structure allows nodes to be dynamically linked together, forming a chain-like sequence.



Singly Linked List

n this representation, each box represents a node, with an arrow indicating the link to the next node. The last node points to NULL, indicating the end of the list.

In most programming languages, a node in a singly linked list is typically defined using a class or a struct.

```
// Definition of a Node in a singly linked list
struct Node {

   // Data part of the node
   int data;

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

```
    // Constructor to initialize the node with data
    Node(int data)
    {
        this->data = data;
        this->next = nullptr;
    }
};
```

In this example, the Node class contains an integer data field (**data**) to store the information and a pointer to another Node (**next**) to establish the link to the next node in the list.

Operations on Singly Linked List

- **Traversal**
- **Searching**
- **Length**
- **Insertion:**
  - Insert at the beginning
  - Insert at the end
  - Insert at a specific position
- **Deletion:**
  - Delete from the beginning
  - Delete from the end
  - Delete a specific node

Let's go through each of the operations mentioned above, one by one.

Traversal in Singly Linked List

Traversal involves visiting each node in the linked list and performing some operation on the data. A simple traversal function would print or process the data of each node.

Step-by-step approach:

- Initialize a pointer current to the head of the list.
- Use a while loop to iterate through the list until the current pointer reaches NULL.
- Inside the loop, print the data of the current node and move the current pointer to the next node.

Below is the function for traversal in singly Linked List:

```
// C++ Function to traverse and print the elements of the linked

// list

void traverseLinkedList(Node* head)

{
    // Start from the head of the linked list
    Node* current = head;

    // Traverse the linked list until reaching the end
    // (nullptr)
    while (current != nullptr) {

        // Print the data of the current node
        cout << current->data << " ";

        // Move to the next node
        current = current->next;
    }

    cout << std::endl;
}
```

Searching in Singly Linked List

Searching in a Singly Linked List refers to the process of looking for a specific element or value within the elements of the linked list.

Step-by-step approach:

1. Traverse the Linked List starting from the head.
2. Check if the current node's data matches the target value.
   - If a match is found, return **true**.
3. Otherwise, Move to the next node and repeat steps 2.
4. If the end of the list is reached without finding a match, return **false**.

Below is the function for searching in singly linked list:

```
// Function to search for a value in the Linked List
bool searchLinkedList(struct Node* head, int target)
{
    // Traverse the Linked List
    while (head != nullptr) {

        // Check if the current node's
        // data matches the target value
        if (head->data == target) {
            return true; // Value found
        }

        // Move to the next node
        head = head->next;
    }

    return false; // Value not found
}
```

Finding Length in Singly Linked List

Finding Length in Singly Linked List refers to the process of determining the total number of nodes in a singly linked list.

Step-by-step approach:

- Initialize a counter **length** to 0.
- Start from the head of the list, assign it to current.
- Traverse the list:
  - Increment **length** for each node.
  - Move to the next node (**current = current->next**).
- Return the final value of **length**.

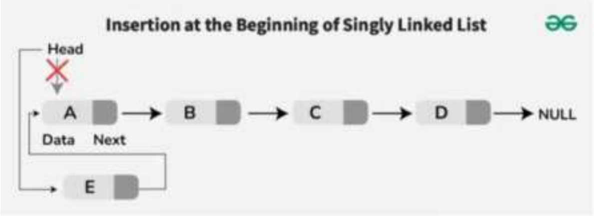
Below is the function for finding length in Singly Linked List:

```
// C++ function to find the length of the linked list
int findLength(Node* head)
{
    // Initialize a counter for the length
    int length = 0;

    // Start from the head of the list
    Node* current = head;

    // Traverse the list and increment the length for each
    // node
    while (current != nullptr) {
        length++;
        current = current->next;
    }

    // Return the final length of the linked list
    return length;
}
```

Insertion in Singly Linked List

Insertion is a fundamental operation in linked lists that involves adding a new node to the list. There are several scenarios for insertion:

a. **Insertion at the Beginning of Singly Linked List**:



Insert a Node at the Front/Beginning of Linked List

Step-by-step approach:

- Create a new node with the given value.
- Set the **next** pointer of the new node to the current head.
- Move the head to point to the new node.
- Return the new head of the linked list.

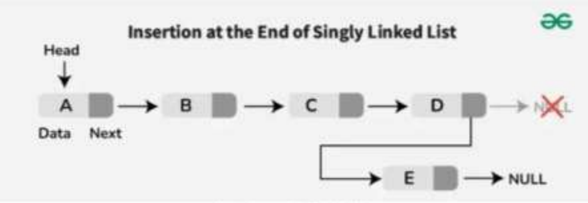
Below is the function for insertion at the beginning of singly linked list:

```
// C++ function to insert a new node at the beginning of the
// linked list
Node* insertAtBeginning(Node* head, int value)
```

```
{
    // Create a new node with the given value
    Node* newNode = new Node(value);

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

    // Move the head to point to the new node
    head = newNode;

    // Return the new head of the linked list
    return head;
}
```

b. Insertion at the End of Singly Linked List:

To insert a node at the end of the list, traverse the list until the last node is reached, and then link the new node to the current last node-



Insertion at end of Linked List

Step-by-step approach:

- Create a new node with the given value.
- Check if the list is empty:
  - If it is, make the new node the head and return.
- Traverse the list until the last node is reached.
- Link the new node to the current last node by setting the last node's next pointer to the new node.

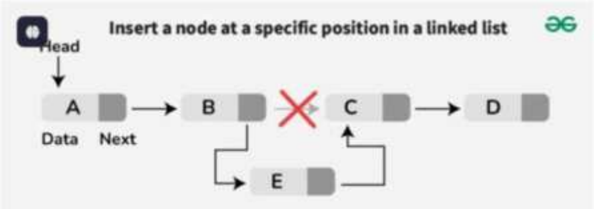
Below is the function for insertion at the end of singly linked list:

```
// C++ Function to insert a node at the end of the linked
// list
```

```
Node* insertAtEnd(Node* head, int value)
{
    // Create a new node with the given value
    Node* newNode = new Node(value);

    // If the list is empty, make the new node the head
    if (head == nullptr)
        return newNode;

    // Traverse the list until the last node is reached
    Node* curr = head;
    while (curr->next != nullptr) {
        curr = curr->next;
    }

    // Link the new node to the current last node
    curr->next = newNode;
    return head;
}
```

c. Insertion at a Specific Position of the Singly Linked List:

To insert a node at a specific position, traverse the list to the desired position, link the new node to the next node, and update the links accordingly.



We mainly find the node after which we need to insert the new node. If we encounter a NULL before reaching that node, it means that the given position is invalid.

Below is the function for insertion at a specific position of the singly linked list:

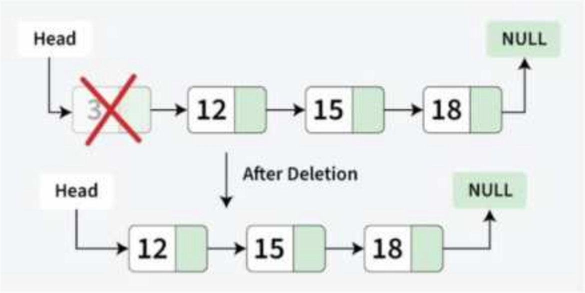
```
// Function to insert a Node at a specified position
// without using a double pointer
Node* insertPos(Node* head, int pos, int data)
{
    if (pos < 1) {
        cout << "Invalid position!" << endl;
        return head;
    }

    // Special case for inserting at the head
    if (pos == 1) {
        Node* temp = new Node(data);
        temp->next = head;
        return temp;
    }

    // Traverse the list to find the node
    // before the insertion point
    Node* prev = head;
    int count = 1;
    while (count < pos - 1 && prev != nullptr) {
        prev = prev->next;
        count++;
    }

    // If position is greater than the number of nodes
    if (prev == nullptr) {
        cout << "Invalid position!" << endl;
        return head;
    }

    // Insert the new node at the specified position
    Node* temp = new Node(data);
    temp->next = prev->next;
    prev->next = temp;

    return head;
}
```

Deletion in Singly Linked List

Deletion involves removing a node from the linked list. Similar to insertion, there are different scenarios for deletion:

a. Deletion at the Beginning of **Singly Linked List**:

To delete the first node, update the head to point to the second node in the list.



Deletion at beginning in a Linked List

Steps-by-step approach:

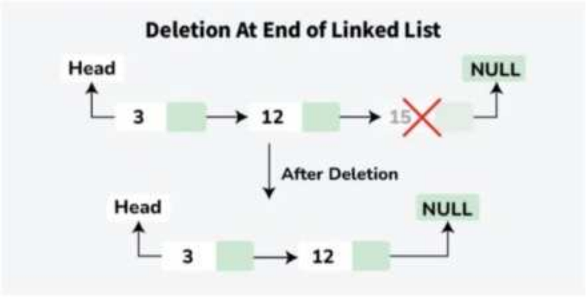
- Check if the head is **NULL**.
    - If it is, return **NULL** (the list is empty).
- Store the current head node in a temporary variable **temp**.
- Move the head pointer to the next node.
- Delete the temporary node.
- Return the new head of the linked list.

Below is the function for deletion at the beginning of singly linked list:

```
// C++ Function to remove the first node of the linked
// list
Node* removeFirstNode(Node* head)
{
    if (head == nullptr)
        return nullptr;

    // Move the head pointer to the next node
    Node* temp = head;
    head = head->next;

    delete temp;

    return head;
}
```

b. Deletion at the End of Singly Linked List:

To delete the last node, traverse the list until the second-to-last node and update its next field to None.



Deletion at the end of linked list

Step-by-step approach:

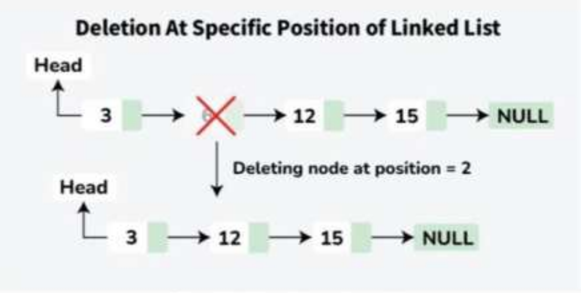
- Check if the head is **NULL**.
  - If it is, return NULL (the list is empty).
- Check if the head's **next** is **NULL** (only one node in the list).
  - If true, delete the head and return **NULL**.
- Traverse the list to find the second last node (**second_last**).
- Delete the last node (the node after **second_last**).
- Set the **next** pointer of the second last node to **NULL**.
- Return the head of the linked list.

Below is the function for deletion at the end of singly linked list:

```
// C++ Function to remove the last node of the linked list
Node* removeLastNode(Node* head)
{
    if (head == nullptr)
        return nullptr;

    if (head->next == nullptr) {
        delete head;
        return nullptr;
    }

    // Find the second last node
    Node* second_last = head;
    while (second_last->next->next != nullptr)
        second_last = second_last->next;

    // Delete last node
    delete (second_last->next);

    // Change next of second last
    second_last->next = nullptr;
    return head;
}
```

c. Deletion at a Specific Position of Singly Linked List:

To delete a node at a specific position, traverse the list to the desired position, update the links to bypass the node to be deleted.


Deletion At Specific Position of Linked List
Head
3
12
15
NULL
Deleting node at position = 2
Head
3
12
15
NULL

Delete a Linked List node at a given position

Step-by-step approach:

- Check if the list is empty or the position is invalid, return if so.
- If the head needs to be deleted, update the head and delete the node.
- Traverse to the node before the position to be deleted.
- If the position is out of range, return.
- Store the node to be deleted.
- Update the links to bypass the node.
- Delete the stored node.

Below is the function for deletion at a specific position of singly linked

```
// C++ function to delete a node at a specific position
Node* deleteAtPosition(Node* head, int position)
{
    // If the list is empty or the position is invalid
    if (head == nullptr || position < 1) {
        return head;
    }

    // If the head needs to be deleted
    if (position == 1) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return head;
    }

    // Traverse to the node before the position to be
    // deleted
    Node* current = head;
    for (int i = 1; i < position - 1 && current != nullptr;
         i++) {
        current = current->next;
    }

    // If the position is out of range
    if (current == NULL || current->next == nullptr) {
        return;
    }

    // Store the node to be deleted
    Node* temp = current->next;

    // Update the links to bypass the node to be deleted
    current->next = current->next->next;

    // Delete the node
    delete temp;

    return head;
}
```

Linked Lists (2)

Importance of Linked List

- **Dynamic Data structure:** The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
- **Ease of Insertion/Deletion:** The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
- **Efficient Memory Utilization:** As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
- **Implementation:** Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

Applications, Advantages and Disadvantages of Linked List

A Linked List is a linear data structure that is used to store a collection of data with the help of nodes. The common variations of linked lists are Singly, Doubly, Singly Circular and Doubly Circular.

Advantages of Linked Lists (or Most Common Use Cases):

- Linked Lists are mostly used because of their effective insertion and deletion.
- Insertion and deletion at any point in a linked list take O(1) time. Whereas in an array data structure, insertion / deletion in the middle takes O(n) time.
- This data structure is simple and can be also used to implement a stack, queues, and other abstract data structures.
- Implementation of Queue data structures : Simple array implementation is not efficient at all. We must use circular array to efficiently implement which is complex.
- Linked List might turn out to be more space efficient compare to arrays in cases where we cannot guess the number of elements in advance.

Applications of Linked Lists:

- Linked Lists can be used to implement stacks, queue, and other types of data structures
- Dynamic memory allocation in operating systems and compilers (linked list of free blocks).
- Manipulation of polynomials
- Arithmetic operations on long integers.
- In operating systems, they can be used in Memory management, process scheduling (for example circular linked list for round robin scheduling) and file system.
- Algorithms that need to frequently insert or delete items from large collections of data.
- LRU cache, which uses a doubly linked list to keep track of the most recently used items in a cache.

Applications of Linked Lists in real world:

- The list of songs in the music player are linked to the previous and next songs.
- In a web browser, previous and next web page URLs can be linked through the previous and next buttons (Doubly Linked List)
- In image viewer, the previous and next images can be linked with the help of the previous and next buttons (Doubly Linked List)

Disadvantages of Linked Lists:

Linked lists are a popular data structure in computer science, but like any other data structure, they have certain disadvantages as well. Some of the key disadvantages of linked lists are:

- **Slow Access Time:** Accessing elements in a linked list can be slow, as you need to traverse the linked list to find the element you are looking for, which is an O(n) operation.
- **Pointers or References:** Linked lists use pointers or references to access the next node, which can make them more complex to understand and use compared to arrays.
- **Higher overhead:** Each node in a linked list requires extra memory to store the reference to the next node.
- **Cache Inefficiency:** Linked lists are cache-inefficient because the memory is not contiguous.

Advantages of Linked List over arrays :

- **Efficient insertion and deletion**. : Insertion and deletion at any point in a linked list take O(1) time. Whereas in an array data structure, insertion / deletion in the middle takes O(n) time.
- **Implementation of Queue and Deque** : It is easier to implement Queue and Dequeue than with arrays or circular arrayas.
- **Space Efficient in Some Cases :** More space efficient where we cannot guess the number of elements in advance.
- **Circular List with Deletion/Addition :** Circular Linked Lists are useful to implement CPU round robin scheduling or similar requirements in the real world because of the quick deletion/insertion in a circular manner.

Advantages of Arrays over Linked List :

- **Random Access**. : We can access ith item in O(1) time (only some basic arithmetic required using base address). In case of linked lists, it is O(n) operation due to sequential access**.**

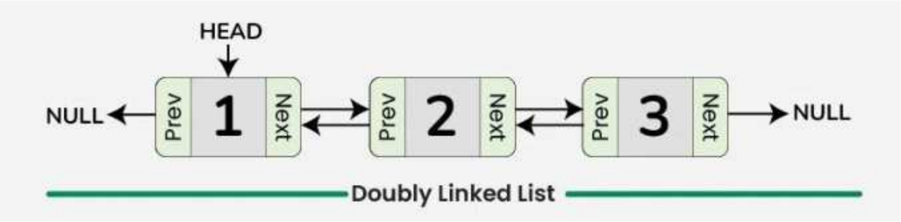
- **Cache Friendliness** : Array items (Or item references) are stored at contiguous locations which makes array cache friendly (Please refer Spatial locality of reference for more details)
- **Easy to use :** Arrays are relatively very easy to use and are available as core of programming languages
- **Less Overhead :** Unlike linked list, we do not have any extra references / pointers to be stored with every item.

Doubly Linked List

A doubly linked list is a more complex data structure than a singly linked list, but it offers several advantages. The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous node and a pointer to the next node. This allows for quick and easy insertion and deletion of nodes from the list, as well as efficient traversal of the list in both directions.

What is a Doubly Linked List?

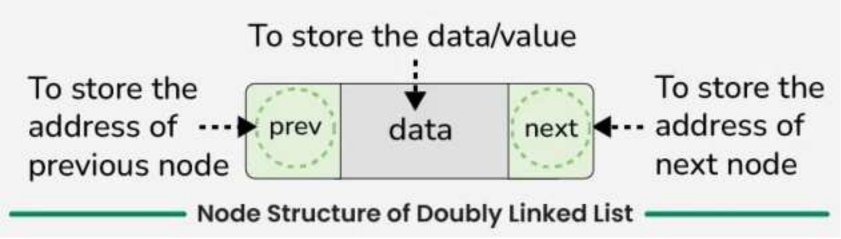
A doubly linked list is a data structure that consists of a set of nodes, each of which contains a value and two pointers, one pointing to the previous node in the list and one pointing to the next node in the list. This allows for efficient traversal of the list in both directions, making it suitable for applications where frequent insertions and deletions are required.



Representation of Doubly Linked List in Data Structure

In a data structure, a doubly linked list is represented using nodes that have three fields:

Data
A pointer to the next node (next)
A pointer to the previous node (prev)


To store the data/value
To store the address of previous node
prev
data
next
To store the address of next node
Node Structure of Doubly Linked List

Node Definition

Here is how a node in a Doubly Linked List is typically represented:

```
struct Node {

   // To store the Value or data.
   int data;

   // Pointer to point the Previous Element
   Node* prev;

   // Pointer to point the Next Element
   Node* next;

   // Constructor
   Node(int d) {
     data = d;
     prev = next = nullptr;
   }
};
```

Each node in a Doubly Linked List contains the data it holds, a pointer to the next node in the list, and a pointer to the previous node in the list. By linking these nodes together through the next and prev pointers, we can traverse the list in both directions (forward and backward), which is a key feature of a Doubly Linked List.

Operations on Doubly Linked List

Traversal in Doubly Linked List

Searching in Doubly Linked List

Finding Length of Doubly Linked List

Insertion in Doubly Linked List:

Insertion at the beginning of Doubly Linked List

Insertion at the end of the Doubly Linked List

Insertion at a specific position in Doubly Linked List

Deletion in Doubly Linked List:

Deletion of a node at the beginning of Doubly Linked List

Deletion of a node at the end of Doubly Linked List

Deletion of a node at a specific position in Doubly Linked List

Let's go through each of the operations mentioned above, one by one.

Traversal in Doubly Linked List

To Traverse the doubly list, we can use the following steps:

a. Forward Traversal:

- Initialize a pointer to the head of the linked list.
- While the pointer is not null:
    - Visit the data at the current node.
    - Move the pointer to the next node.

b. Backward Traversal:

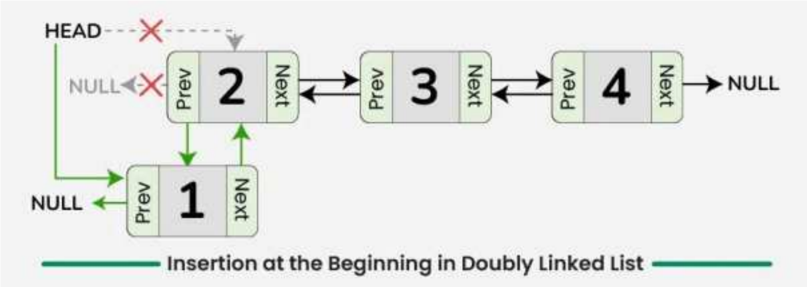
- Initialize a pointer to the tail of the linked list.
- While the pointer is not null:
    - Visit the data at the current node.
    - Move the pointer to the previous node.

Finding Length of Doubly Linked List

To find the length of doubly list, we can use the following steps:

- Start at the head of the list.
- Traverse through the list, counting each node visited.
- Return the total count of nodes as the length of the list.

Insertion at the Beginning in Doubly Linked List

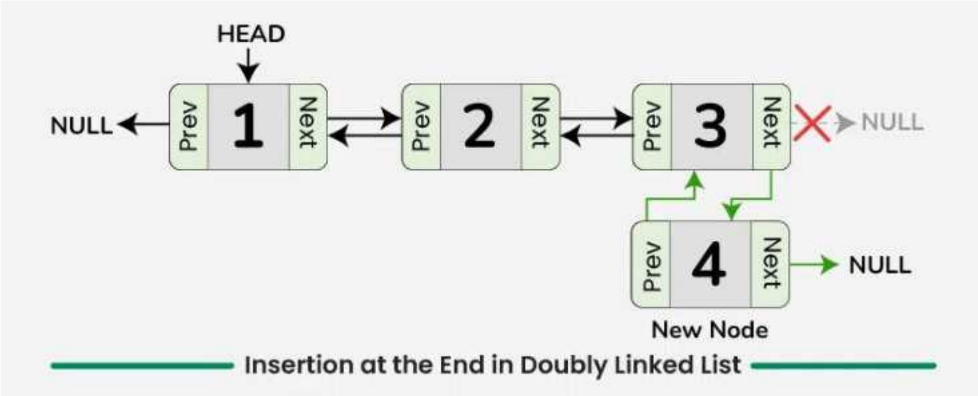


Insertion at the Beginning in Doubly Linked List

To insert a new node at the beginning of the doubly list, we can use the following steps:

- Create a new node, say new_node with the given data and set its previous pointer to null, new_node->prev = NULL.
- Set the next pointer of new_node to current head, new_node->next = head.
- If the linked list is not empty, update the previous pointer of the current head to new_node, head->prev = new_node.
- Return new_node as the head of the updated linked list.

Insertion at the End of Doubly Linked List



Insertion at the End in Doubly Linked List

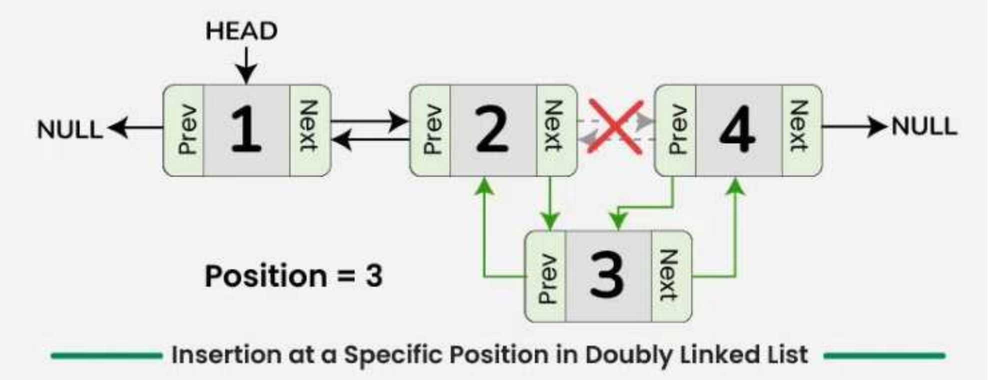
To insert a new node at the end of the doubly linked list, we can use the following steps:

- Allocate memory for a new node and assign the provided value to its data field.
- Initialize the next pointer of the new node to nullptr.
- If the list is empty:

    - Set the previous pointer of the new node to nullptr.
    - Update the head pointer to point to the new node.
- If the list is not empty:
    - Traverse the list starting from the head to reach the last node.
    - Set the next pointer of the last node to point to the new node.
    - Set the previous pointer of the new node to point to the last node.

Insertion at a Specific Position in Doubly Linked List

To insert a node at a specific Position in doubly linked list, we can use the following steps:

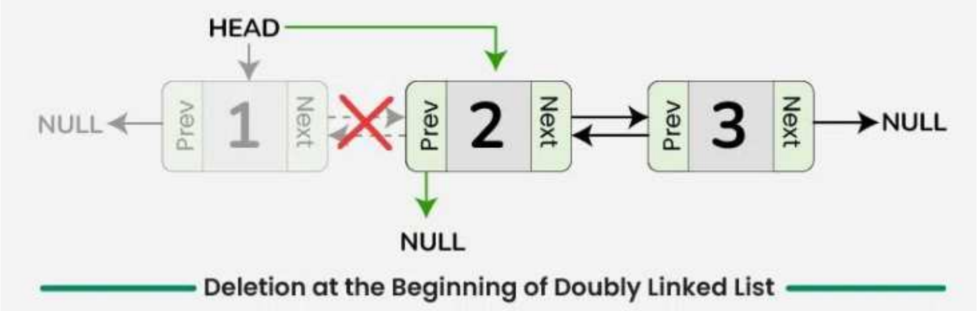


Insertion at a Specific Position in Doubly Linked List

To insert a new node at a specific position,

- If position = 1, create a new node and make it the head of the linked list and return it.
- Otherwise, traverse the list to reach the node at position – 1, say curr.
- If the position is valid, create a new node with given data, say new_node.
- Update the next pointer of new node to the next of current node and prev pointer of new node to current node, new_node->next = curr->next and new_node->prev = curr.
- Similarly, update next pointer of current node to the new node, curr->next = new_node.
- If the new node is not the last node, update prev pointer of new node's next to the new node, new_node->next->prev = new_node.

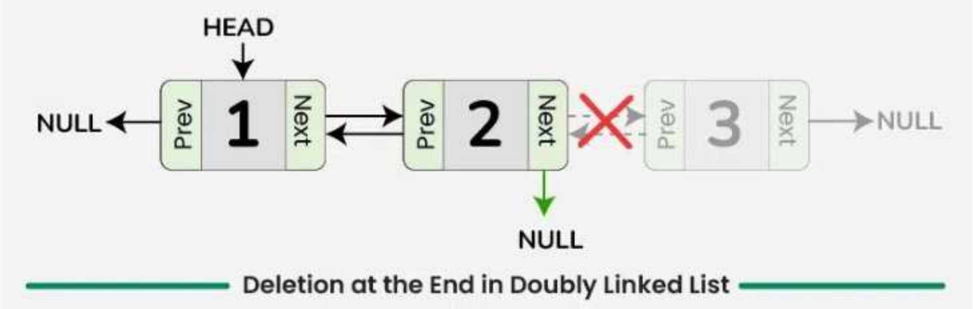
Deletion at the Beginning of Doubly Linked List



To delete a node at the beginning in doubly linked list, we can use the following steps:

- Check if the list is empty, there is nothing to delete. Return.
- Store the head pointer in a variable, say temp.
- Update the head of linked list to the node next to the current head, head = head->next.
- If the new head is not NULL, update the previous pointer of new head to NULL, head->prev = NULL.

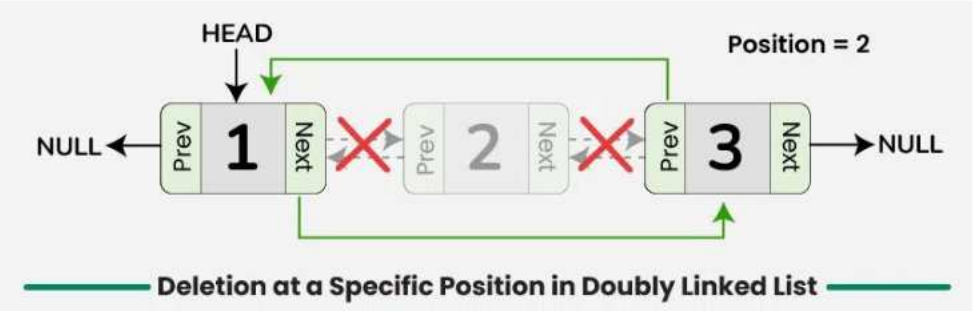
Deletion at the End of Doubly Linked List



To delete a node at the end in doubly linked list, we can use the following steps:

- Check if the doubly linked list is empty. If it is empty, then there is nothing to delete.
- If the list is not empty, then move to the last node of the doubly linked list, say curr.
- Update the second-to-last node's next pointer to NULL, curr->prev->next = NULL.
- Free the memory allocated for the node that was deleted.

Deletion at a Specific Position in Doubly Linked List



To delete a node at a specific position in doubly linked list, we can use the following steps:

- Traverse to the node at the specified position, say **curr**.
- If the position is valid, adjust the pointers to skip the node to be deleted.
  - If curr is not the head of the linked list, update the next pointer of the node before curr to point to the node after curr, **curr->prev->next = curr-next**.
  - If curr is not the last node of the linked list, update the previous pointer of the node after curr to the node before curr, **curr->next->prev = curr->prev**.
- Free the memory allocated for the deleted node.

Advantages of Doubly Linked List

Efficient traversal in both directions: Doubly linked lists allow for efficient traversal of the list in both directions, making it suitable for applications where frequent insertions and deletions are required.

Easy insertion and deletion of nodes: The presence of pointers to both the previous and next nodes makes it easy to insert or delete nodes from the list, without having to traverse the entire list.

Can be used to implement a stack or queue: Doubly linked lists can be used to implement both stacks and queues, which are common data structures used in programming.

Disadvantages of Doubly Linked List

More complex than singly linked lists: Doubly linked lists are more complex than singly linked lists, as they require additional pointers for each node.

More memory overhead: Doubly linked lists require more memory overhead than singly linked lists, as each node stores two pointers instead of one.

Applications of Doubly Linked List

Implementation of undo and redo functionality in text editors.

Cache implementation where quick insertion and deletion of elements are required.

Browser history management to navigate back and forth between visited pages.

Music player applications to manage playlists and navigate through songs efficiently.

Implementing data structures like Deque (double-ended queue) for efficient insertion and deletion at both ends.If the position is valid, adjust the pointers to skip the node to be deleted.

If curr is not the head of the linked list, update the next pointer of the node before curr to point to the node after curr, curr->prev->next = curr-next.

If curr is not the last node of the linked list, update the previous pointer of the node after curr to the node before curr, curr->next->prev = curr->prev.

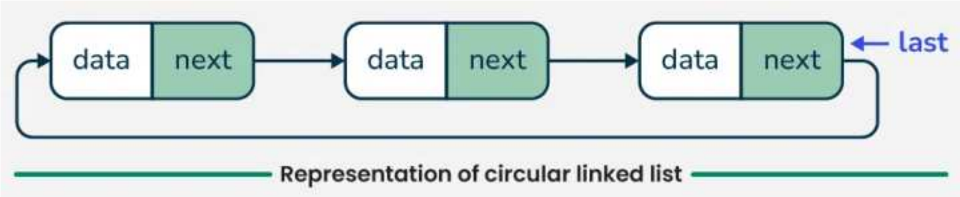
Free the memory allocated for the deleted node.

Circular Linked List

A Circular Linked List is a variation of a Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

Singly Linked List as Circular

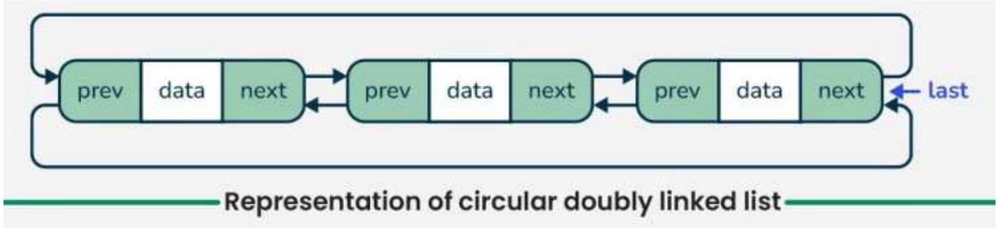
In a singly linked list, the next pointer of the last node points to the first node.



Representation of circular linked list

Doubly Linked List as Circular

In a doubly linked list, the next pointer of the last node points to the first node, and the previous pointer of the first node points to the last node making the circular in both directions.



Representation of circular doubly linked list

Basic Operations

Following are the important operations supported by a circular list.

- **insert** − Inserts an element at the start of the list.
- **delete** − Deletes an element from the start of the list.
- **display** − Displays the list.

Insertion Operation

The insertion operation of a circular linked list only inserts the element at the start of the list. This differs from the usual singly and doubly linked lists as there is no particular starting and ending points in this list. The insertion is done either at the start or after a particular node (or a given position) in the list.

Algorithm

```
1. START
```

```
2. Check if the list is empty
3. If the list is empty, add the node and point the head
   to this node
4. If the list is not empty, link the existing head as
   the next node to the new node.
5. Make the new node as the new head.
6. END
```

Definition of the node:

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

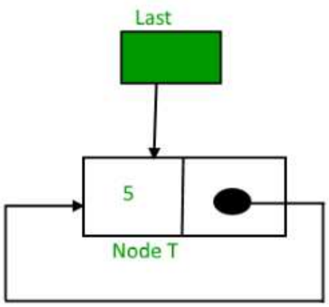
Why have we taken a pointer that points to the last node instead of the first node?

For the insertion of a node at the beginning, we need to traverse the whole list. Also, for insertion at the end, the whole list has to be traversed. If instead of the start pointer, we take a pointer to the last node, then in both cases there won't be any need to traverse the whole list. So insertion at the beginning or at the end takes constant time, irrespective of the length of the list.

<u>Insertion in an empty List:</u>

Initially, when the list is empty, the *last* pointer will be NULL.

After inserting node T,



After insertion, T is the last node, so the pointer *last* points to node T. And Node T is the first and the last node, so T points to itself.

Below is the implementation of the above operation:

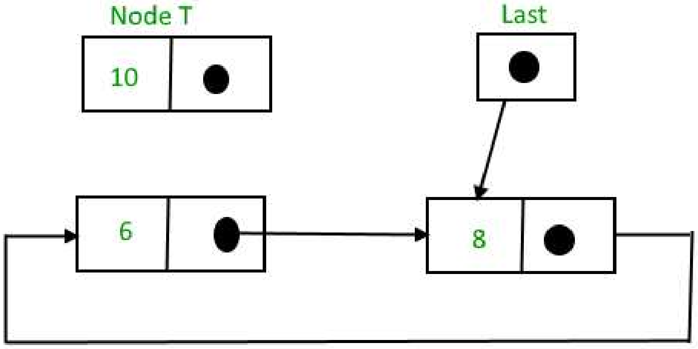
```
// C++ program for the above operation
#include<iostream>
Using namespace std;
struct Node{
  int data;
   Node   *next;
};

struct Node* addToEmpty(struct Node* last, int data)
{
    // This function is only for empty list
    if (last != NULL)  //  It is not an empty list
return last;

    // Creating a node dynamically.
    struct Node* temp  = new Node();

    // Assigning the data.
    temp->data = data;
    last = temp;
    // Note : list was empty. We link single node  to itself.
    temp->next = last;

    return last;
}
```

```
int main()
{
Node *last=NULL, *addednode;
int data=7;
cout<<addToEmpty( last, data);
return 0;
}
```

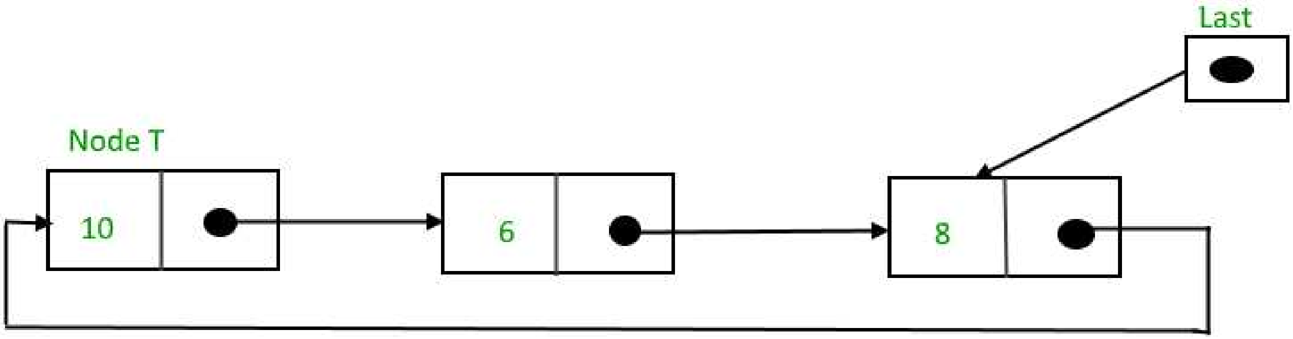
Insertion at the beginning of the list

To insert a node at the beginning of the list, follow these steps:

- Create a node, say T
- Make T -> next = last -> next
- last -> next = T



After insertion,



Below is the implementation of the above operation:

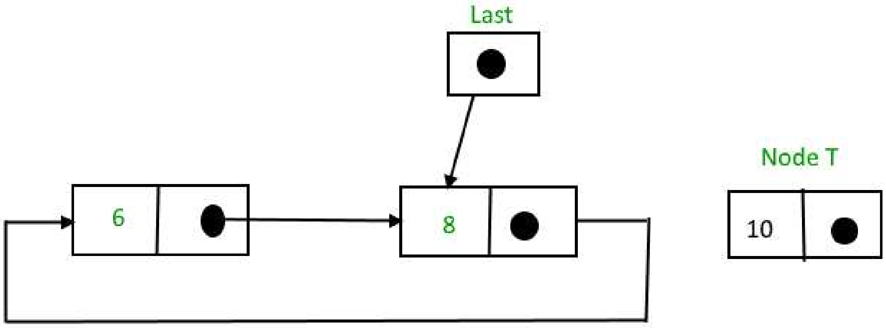
```
// C++ function for the above operation
struct Node* addBegin(struct Node* last, int data)
{
   if (last == NULL)
      return addToEmpty(last, data);
```

```
    // Creating a node dynamically.
    struct Node* temp  = new Node();

    // Assigning the data.
temp->data = data;

    // Adjusting the links.
    temp->next = last->next;
    last->next = temp;
    return last;
 }
```

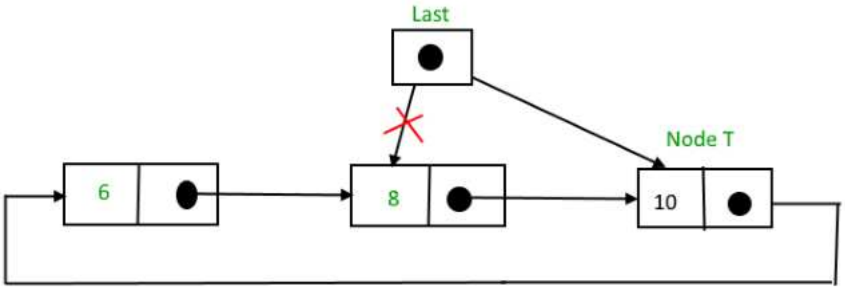
Insertion at the end of the list

To insert a node at the end of the list, follow these steps:

- Create a node, say T
- Make T -> next = last -> next
- last -> next = T
- last = T



Ater insertion



Below is the implementation of the above operation:

// C++ function for the above operation

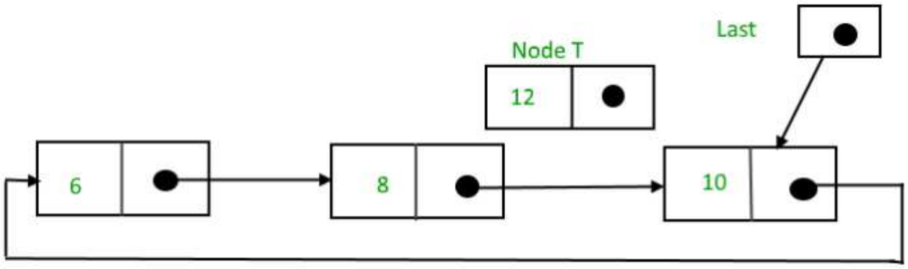
```
struct Node* addEnd(struct Node* last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    // Creating a node dynamically.
    struct Node* temp  = new Node();

    // Assigning the data.
    temp->data = data;

    // Adjusting the links.
    temp->next = last->next;
    last->next = temp;
    last = temp;

    return last;
}
```

Insertion in between the nodes

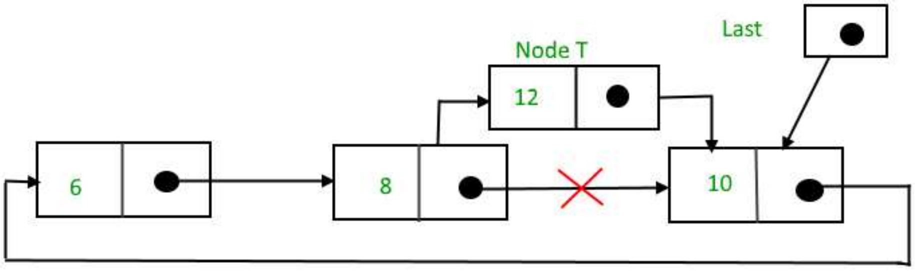
To insert a node in between the two nodes, follow these steps:

- Create a node, say T.
- Search for the node after which T needs to be inserted, say that node is P.
- Make T -> next = P -> next;
- P -> next = T.

Suppose 12 needs to be inserted after the node that has the value 8,



After searching and insertion,



Below is the implementation of the above operation:
// C++ function for the above operation

```
struct Node* addAfter(struct Node* last, int data, int item)
{
    if (last == NULL)
        return NULL;

    struct Node *temp, *p;
    p = last->next;

    // Searching the item.
    do {
        if (p->data == item) {
            // Creating a node dynamically.
            Temp= new Node();

            // Assigning the data.
            temp->data = data;

            // Adjusting the links.
            temp->next = p->next;

            // Adding newly allocated node after p.
            p->next = temp;

            // Checking for the last node.
            if (p == last)
                last = temp;

            return last;
        }
        p = p->next;
    } while (p != last->next);

    cout << item << " not present in the list." << endl;
    return last;
}
```

Deletion Operation

The Deletion operation in a Circular linked list removes a certain node from the list. The deletion operation in this type of lists can be done at the beginning, or a given position, or at the ending.

Delete at the beginning

```
struct node * deleteFirst(){

  //save reference to first link
  struct node *tempLink = last;
  if(last->next == last) {
      last = NULL;
    return tempLink;
  }

  //mark next to first link as first
last = last->next;

  //return the deleted link
  return tempLink;
}
```

Displaying the List

The Display List operation visits every node in the list and prints them all in the output.

Algorithm

```
1. START
2. Walk through all the nodes of the list and print them
3. END
```

```
void traverse(struct Node* last)
{
    struct Node* p;

    // If list is empty, return.
    if (last == NULL) {
        cout << "List is empty." << endl;
        return;
    }

    // Pointing to first Node of the list.
    p = last->next;

    // Traversing the list.
    do {
        cout << p->data << " ";
        p = p->next;
    } while (p != last->next);
}
```

Advantages of Circular Linked Lists

1. In circular linked list, the last node points to the first node. There are no null references, making traversal easier and reducing the chances of encountering null pointer exceptions.
2. We can traverse the list from any node and return to it without needing to restart from the head, which is useful in applications requiring a circular iteration.
3. Circular linked lists can easily implement circular queues, where the last element connects back to the first, allowing for efficient resource management.
4. In a circular linked list, each node has a reference to the next node in the sequence. Although it doesn’t have a direct reference to the previous node like a doubly linked list, we can still find the previous node by traversing the list.

Disadvantages of Circular Linked Lists

1. Circular linked lists are more complex to implement than singly linked lists.
2. Traversing a circular linked list without a clear stopping condition can lead to infinite loops if not handled carefully.

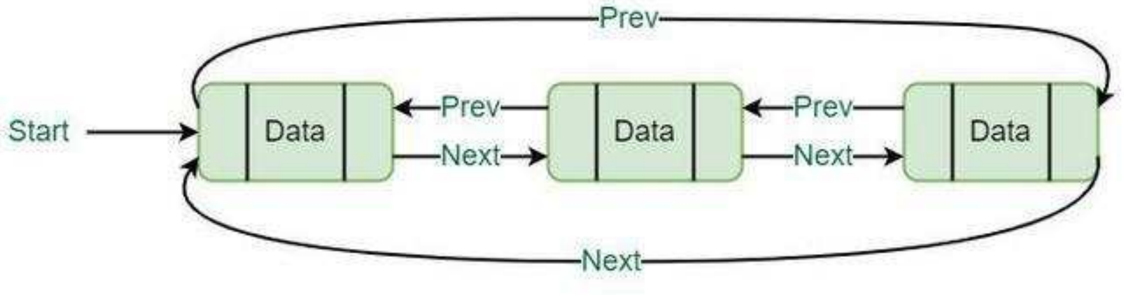
3. Debugging can be more challenging due to the circular nature, as traditional methods of traversing linked lists may not apply.

Applications of Circular Linked Lists

1. It is used for time-sharing among different users, typically through a Round-Robin scheduling mechanism.
2. In multiplayer games, a circular linked list can be used to switch between players. After the last player’s turn, the list cycles back to the first player.
3. Circular linked lists are often used in buffering applications, such as streaming data, where data is continuously produced and consumed.
4. In media players, circular linked lists can manage playlists, this allowing users to loop through songs continuously.
5. Browsers use circular linked lists to manage the cache. This allows you to navigate back through your browsing history efficiently by pressing the BACK button.

Circular Doubly Linked List

*A **circular doubly linked list** is defined as a circular linked list in which each node has two links connecting it to the previous node and the next node.*



Circular Doubly Linked LIst

Characteristics of Circular Doubly Linked List :

A circular doubly linked list has the following properties:

- **Circular**: A circular doubly linked list's main feature is that it is circular in design.
- **Doubly Linked**: Each node in a circular doubly linked list has two pointers – one pointing to the node before it and the other pointing to the node after it.
- **Header Node**: At the start of circular doubly linked lists, a header node is frequently used.

Applications of Circular Doubly Linked List :

Circular doubly linked lists are used in a variety of applications, some of which include:

- **Implementation of Circular Data Structures:** Circular doubly linked lists are extremely helpful in the construction of circular data structures like circular queues and circular buffers.
- **Implementing Undo-Redo Operations**: Text editors and other software programs can use circular doubly linked lists to implement undo-redo operations.

- **Music Player Playlist**: Playlists in music players are frequently implemented using circular doubly linked lists. Each song is kept as a node in the list in this scenario, and the list can be circled to play the songs in the order they are listed.
- **Cache Memory Management**: To maintain track of the most recently used cache blocks, circular doubly linked lists are employed in cache memory management.

Advantages of Circular Doubly Linked List :

Circular doubly linked lists in Data Structures and Algorithms (DSA) have the following benefits:

- **Efficient Traversal**: A circular doubly linked list's nodes can be efficiently traversed in both ways, or forward and backward.
- **Insertion and deletion**: A circular doubly linked list makes efficient use of insertion and deletion operations. The head and tail nodes are connected because the list is circular, making it simple to add or remove nodes from either end.
- **Implementation of Circular Data Structures**: The implementation of circular data structures like circular queues and circular buffers makes extensive use of circular doubly linked lists.

Disadvantages of Circular Doubly Linked List :

Circular doubly linked lists have the following drawbacks when used in DSA:

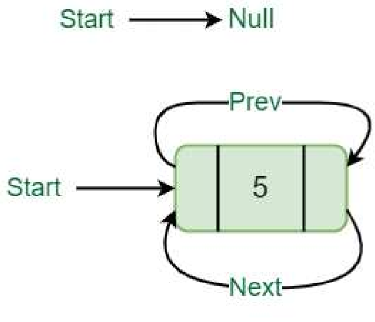
- **Complexity**: Compared to a singly linked list, the circular doubly linked list has more complicated operations, which can make it more difficult to develop and maintain.
- **Cost of Circularity**: In some circumstances, the list's circularity may result in additional overhead. For instance, it may be challenging to tell whether the traversal of the list has completely circled the object and returned to its beginning place.
- **More Complex to Debug**: Circular doubly linked lists can be more difficult to debug than single-linked lists because the circular nature of the list might introduce loops that are challenging to find and repair.

Structure of the node

```
// Structure of a Node
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
```

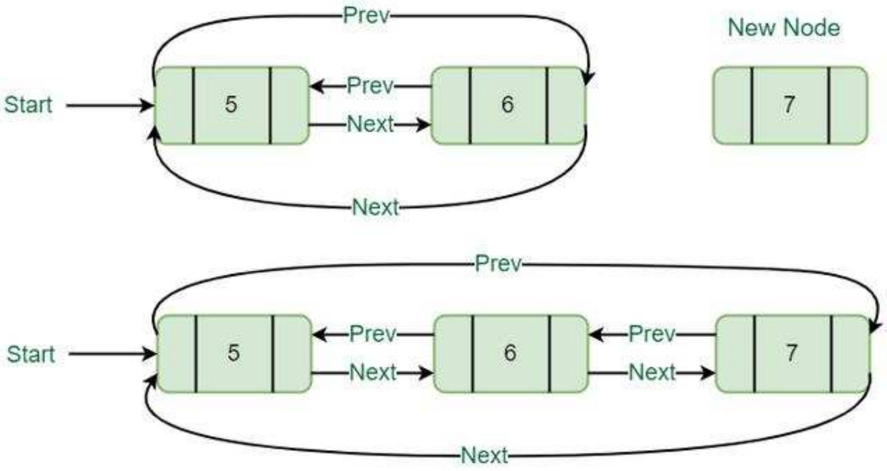
Insertion in Circular Doubly Linked List:

1. Insertion at the end of the list or in an empty list:



Insertion in an empty list

2. List initially contains some nodes, *Start* points to the first node of the List:

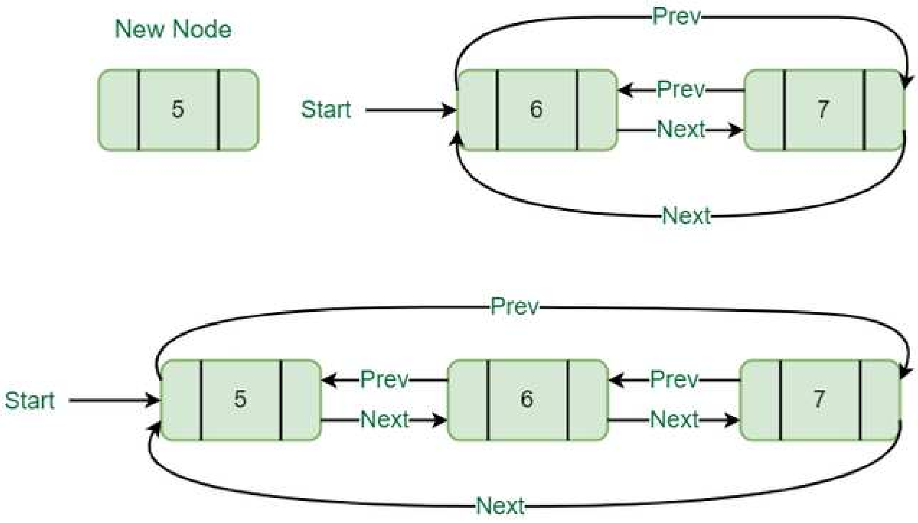


Insertion at the end of list

Below is the implementation of the above operations:

```
// Function to insert at the end
Node * insertEnd(struct Node* start, int value)
{
    // If the list is empty, create a single node
    // circular and doubly list
    if (start == NULL) {
        struct Node* new_node = new Node;
        new_node->data = value;
        new_node->next = new_node->prev = new_node;
        start = new_node;
        return start;
    }

    // If list is not empty

    /* Find last node */
    Node* last = start->prev;

    // Create Node dynamically
    struct Node* new_node = new Node;
    new_node->data = value;

    // Start is going to be next of new_node
    new_node->next = start;

    // Make new node previous of start
    start->prev = new_node;

    // Make last previous of new node
    new_node->prev = last;
    // Make new node next of old last
    last->next = new_node;

    return start;
}
```

3. Insertion at the beginning of the list:

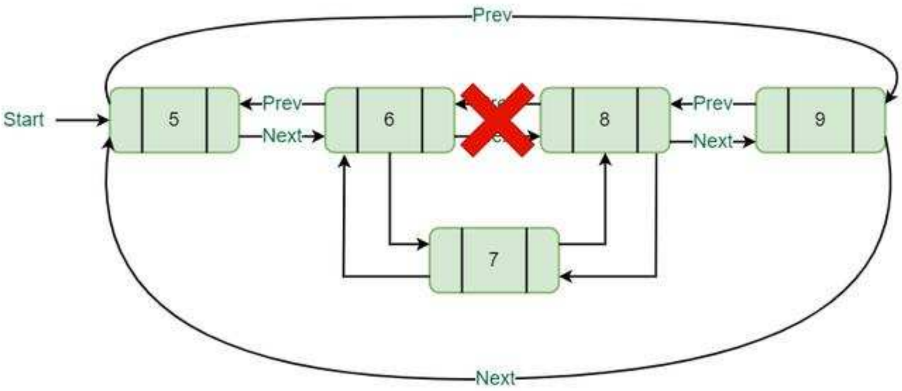


Insertion at the beginning of the list

Below is the implementation of the above operation:

```
// Function to insert Node at the beginning
// of the List,
Node * insertBegin(struct Node* start, int value)
{
    // Pointer points to last Node
    struct Node* last = start->prev;

    struct Node* new_node = new Node;
    new_node->data = value; // Inserting the data

    // setting up previous and next of new node
    new_node->next = start;
    new_node->prev = last;
    // Update next and previous pointers of start
    // and last.
    last->next = start->prev = new_node;
    // Update start pointer
    start = new_node;
    return start;
}
```

4. Insertion in between the nodes of the list:



Insertion in between other nodes

Below is the implementation of the above operation:

```
// Function to insert node with value as value1.
// The new node is inserted after the node with
// with value2
Node * insertAfter(struct Node* start, int value1,
    int value2)
{
    struct Node* new_node = new Node;
    new_node->data = value1; // Inserting the data

    // Find node having value2 and next node of it
    struct Node* temp = start;
    while (temp->data != value2)
        temp = temp->next;
    struct Node* next = temp->next;
    // insert new_node between temp and next.
    temp->next = new_node;
    new_node->prev = temp;
    new_node->next = next;
    next->prev = new_node;

    return start;
}

void display(struct Node* start)
{
    struct Node* temp = start;

    cout<<"\nTraversal in forward direction \n";
    while (temp->next != start) {
        cout<< temp->data<<”   ”;
        temp = temp->next;
    }
    Cout<< temp->data<<”   ”;

    Cout<<” \nTraversal in reverse direction \n”;
    Node* last = start->prev;
    temp = last;
    while (temp->prev != last) {
        cout<< temp->data<<”  ”;
        temp = temp->prev;
    }
    cout<< temp->data<<”  ”;
}
```

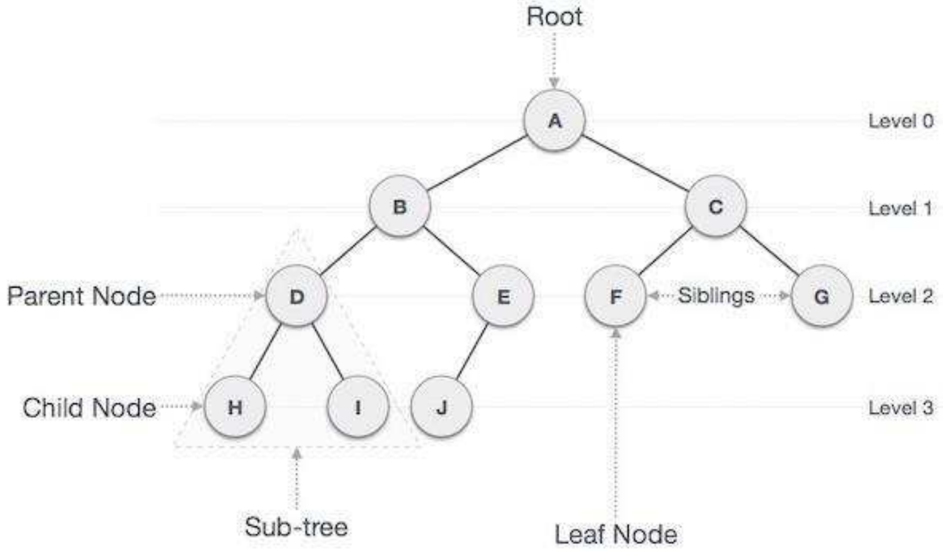
```
/* Driver program to test above functions*/
#include<iostream>
using namespace std;
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
int main()
{
    /* Start with the empty list */
    struct Node* start = NULL;

        // Insert 5. So linked list becomes 5->NULL
        start=insertEnd(start, 5);

       //  Insert 4 at the beginning. So linked
        // list becomes 4->5
        start=insertBegin(start, 4);
    //
    //    // Insert 7 at the end. So linked list
    //    // becomes 4->5->7
        start=insertEnd(start, 7);
    //
    //    // Insert 8 at the end. So linked list
    //    // becomes 4->5->7->8
        start= insertEnd(start, 8);
    //
    //    // Insert 6, after 5. So linked list
    //    // becomes 4->5->6->7->8
        start= insertAfter(start, 6, 5);
    //
    //    cout<<"Created circular doubly linked list is: ";
    display(start);

    return 0;
}
```

Output

```
Created circular doubly linked list is:
Traversal in forward direction
4 5 6 7 8
Traversal in reverse direction
8 7 6 5 4
```

Tree Data Structure

A tree is a non-linear abstract data type with a hierarchy-based structure. It consists of nodes (where the data is stored) that are connected via links. The tree data structure stems from a single node called a root node and has subtrees connected to the root.



Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
-

- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.
-

Why Tree is considered a non-linear data structure?

The data in a tree are not stored in a sequential manner i.e., they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a non-linear data structure.

Importance for Tree Data Structure:

One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer: The DOM model of an HTML page is also tree where we have html tag as root, head and body its children and these tags, then have their own children.

Properties of Tree Data Structure:

Number of edges: An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges. There is only one path from each node to any other node of the tree.

Depth of a node: The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.

Height of a node: The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.

Height of the Tree: The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.

Degree of a Node: The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

Types of Trees

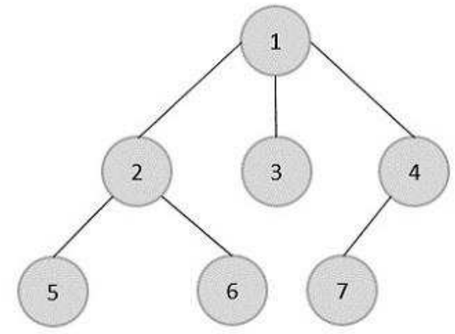
There are three types of trees −

- General Trees
- Binary Trees
- Binary Search Trees

General Trees

General trees are unordered tree data structures where the root node has minimum 0 or maximum ‘n’ subtrees.

The General trees have no constraint placed on their hierarchy. The root node thus acts like the superset of all the other subtrees.



General Tree Data Structure

Binary Trees

Binary Trees are general trees in which the root node can only hold up to maximum 2 subtrees: left subtree and right subtree. Based on the number of children, binary trees are divided into three types.

Full Binary Tree

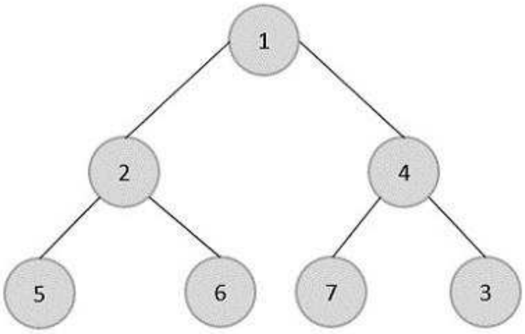
- A full binary tree is a binary tree type where every node has either 0 or 2 child nodes.

Complete Binary Tree

- A complete binary tree is a binary tree type where all the leaf nodes must be on the same level. However, root and internal nodes in a complete binary tree can either have 0, 1 or 2 child nodes.

Perfect Binary Tree

- A perfect binary tree is a binary tree type where all the leaf nodes are on the same level and every node except leaf nodes have 2 children.

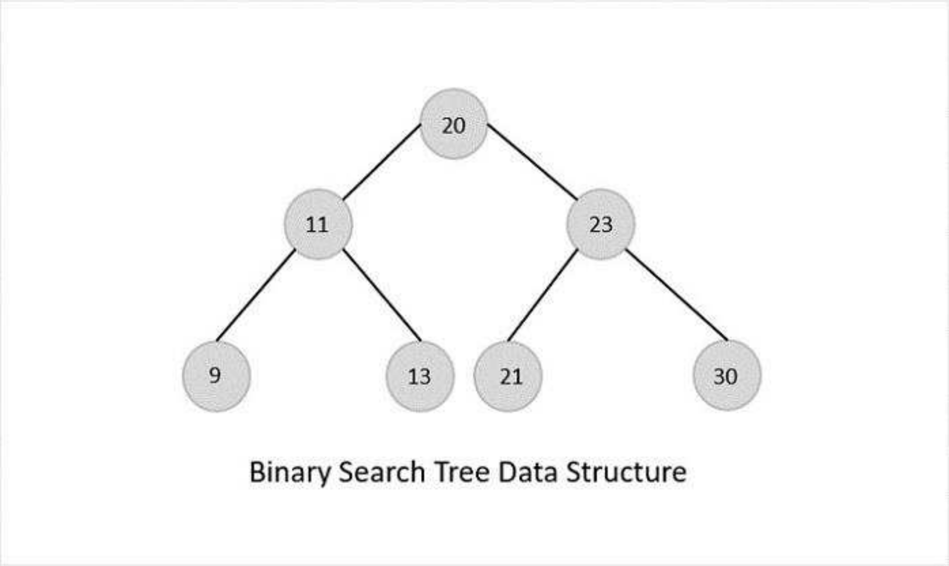


Binary Tree Data Structure

Binary Search Trees

Binary Search Trees possess all the properties of Binary Trees including some extra properties of their own, based on some constraints, making them more efficient than binary trees.

The data in the Binary Search Trees (BST) is always stored in such a way that the values in the left subtree are always less than the values in the root node and the values in the right subtree are always greater than the values in the root node, i.e. left subtree < root node ≤ right subtree.



Balanced Binary Search Trees

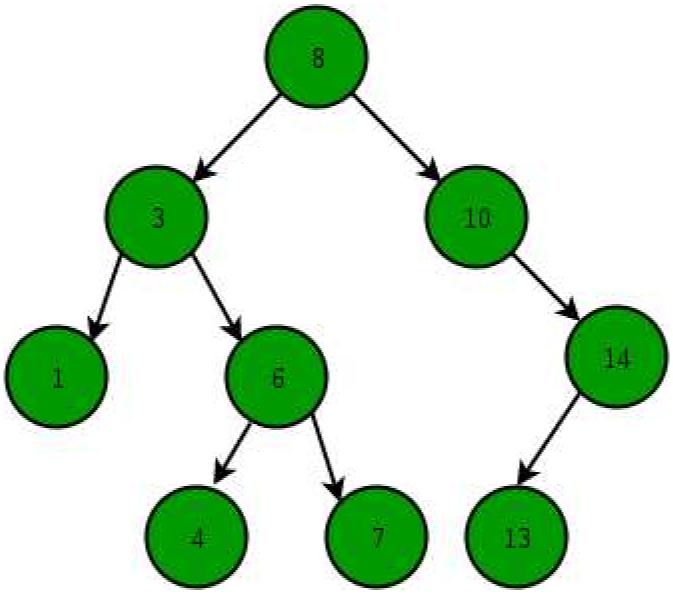
Consider a Binary Search Tree with ‘m’ as the height of the left subtree and ‘n’ as the height of the right subtree. If the value of (m-n) is equal to 0,1 or -1, the tree is said to be a **Balanced Binary Search Tree**.

The trees are designed in a way that they self-balance once the height difference exceeds 1. Binary Search Trees use rotations as self-balancing algorithms. There are four different types of rotations: Left Left, Right Right, Left Right, Right Left.

Introduction to Binary Search Tree

Binary Search tree is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- This means everything to the left of the root is less than the value of the root and everything to the right of the root is greater than the value of the root. Due to this performing, a binary search is very easy.
- The left and right subtree each must also be a binary search tree.
  There must be no duplicate nodes(BST may have duplicate values with different handling approaches)



Handling approach for Duplicate values in the Binary Search tree:

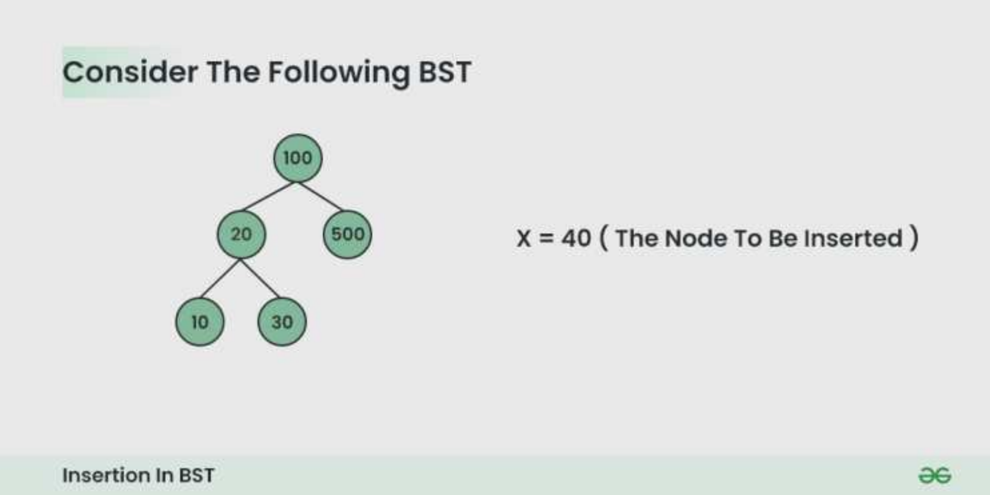
- Cannot allow the duplicated values at all.
- Must follow a consistent process throughout i.e either store duplicate value at the left or store the duplicate value at the right of the root, but be consistent with your approach.
- Can keep the counter with the node and if we found the duplicate value, then we can increment the counter

Insertion in Binary Search Tree (BST)

- Check the value to be inserted (say **X**) with the value of the current node (say **val**) we are in:
  - If **X** is less than **val** move to the left subtree.
  - Otherwise, move to the right subtree.
- Once the leaf node is reached, insert **X** to its right or left based on the relation between **X** and the leaf node's value.

Follow the below illustration for a better understanding:

Illustration:

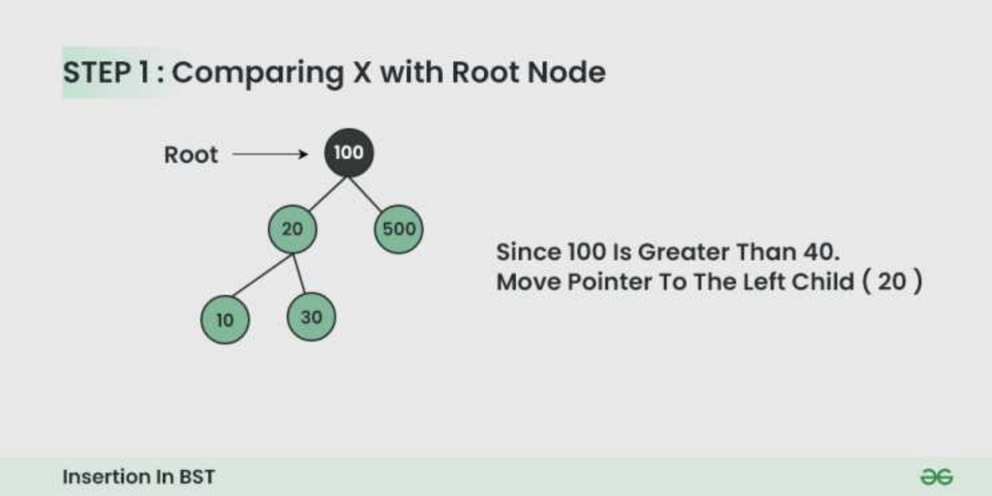


Consider The Following BST

X = 40 (The Node To Be Inserted)

Insertion In BST

Insertion in BST

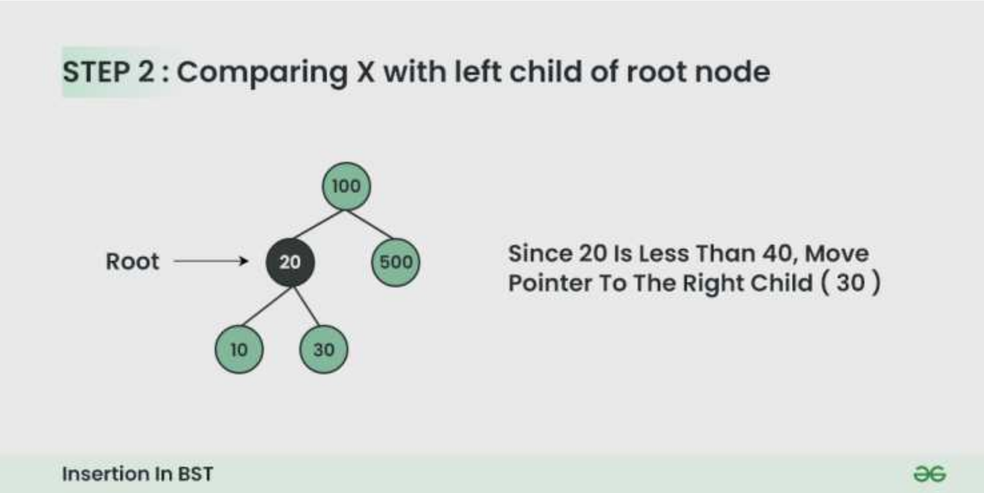


STEP 1 : Comparing X with Root Node

Since 100 Is Greater Than 40.
Move Pointer To The Left Child (20)

Insertion In BST

Insertion in BST

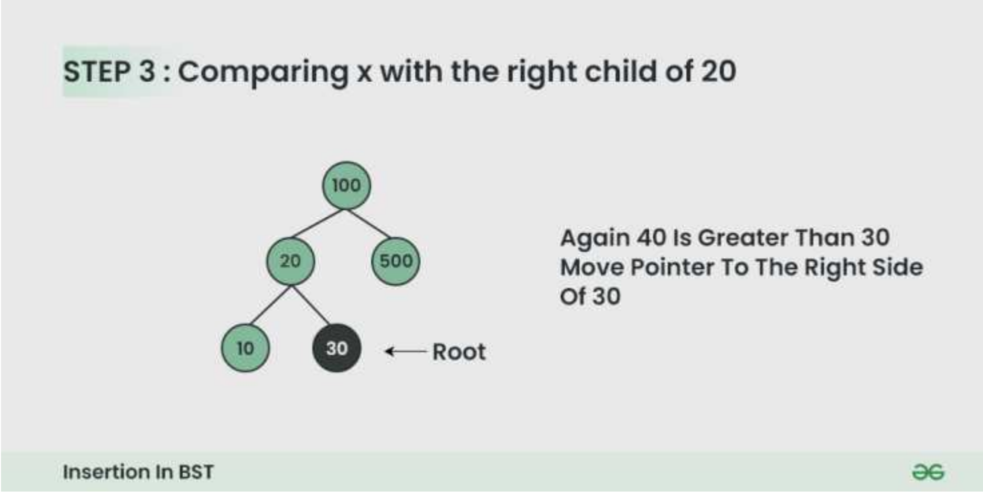


STEP 2 : Comparing X with left child of root node

Since 20 Is Less Than 40, Move Pointer To The Right Child (30)

Insertion In BST

Insertion in BST

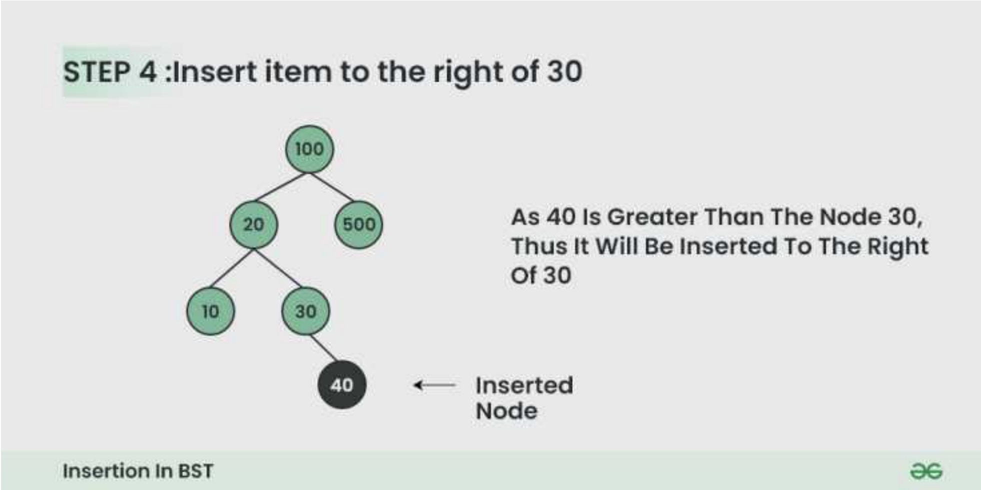


STEP 3 : Comparing x with the right child of 20

Again 40 Is Greater Than 30 Move Pointer To The Right Side Of 30

Insertion In BST

Insertion in BST



STEP 4 :Insert item to the right of 30

As 40 Is Greater Than The Node 30, Thus It Will Be Inserted To The Right Of 30

Insertion In BST

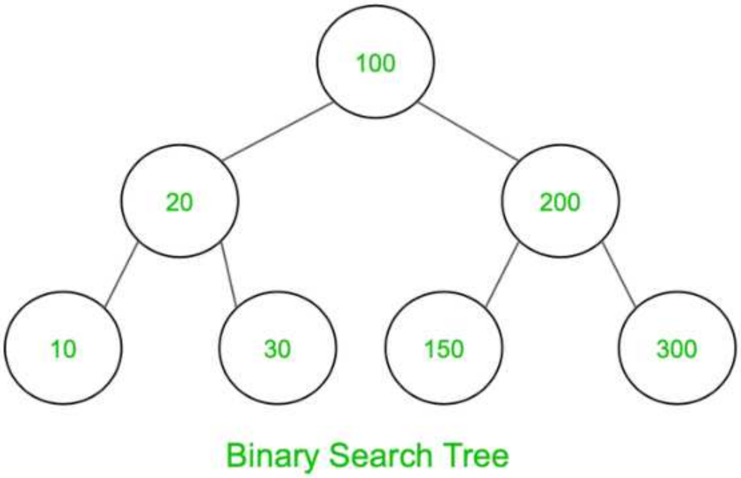
Binary Search Tree (BST) Traversals – Inorder, Preorder, Post Order

-

Tree Traversal techniques include various ways to visit all the nodes of the tree, exactly once in a certain order.
There are multiple tree traversal techniques that decide the order in which the nodes of the tree are to be visited. These are defined below:

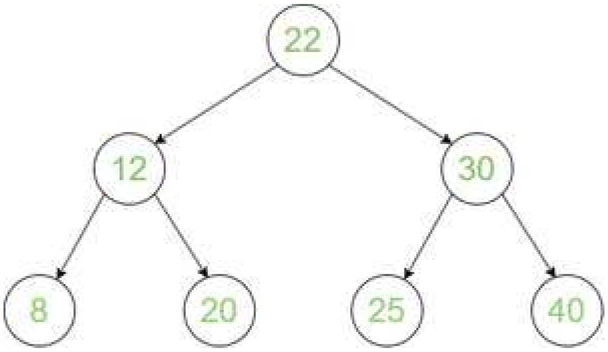
- Depth First Search or DFS
  - Inorder Traversal
  - Preorder Travers
  - Postorder Traversal
- Level Order Traversal or Breadth First Search or BFS

Given a Binary search tree, The task is to print the elements in inorder, preorder, and postorder traversal of the Binary Search Tree.
Input:



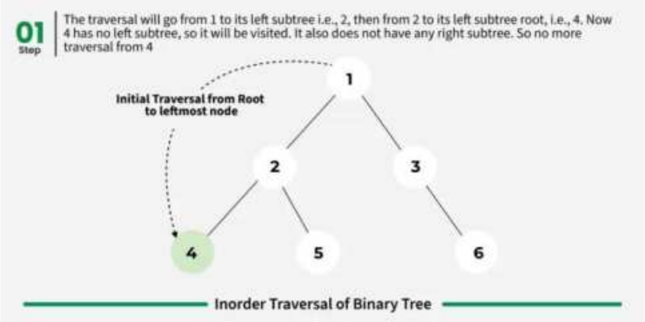
A Binary Search Tree

Output:
Inorder Traversal: 10 20 30 100 150 200 300
Preorder Traversal: 100 20 10 30 200 150 300
Postorder Traversal: 10 30 20 150 300 200 100
Input:



Binary Search Tree

Output:
Inorder Traversal: 8 12 20 22 25 30 40
Preorder Traversal: 22 12 8 20 30 25 40
Postorder Traversal: 8 20 12 25 40 30 22

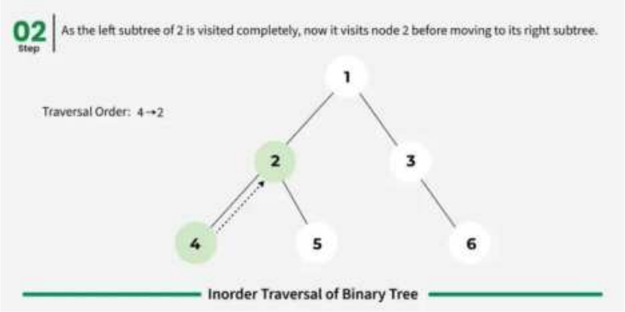


01 Step

The traversal will go from 1 to its left subtree i.e., 2, then from 2 to its left subtree root, i.e., 4. Now 4 has no left subtree, so it will be visited. It also does not have any right subtree. So no more traversal from 4

Initial Traversal from Root to leftmost node

Inorder Traversal of Binary Tree

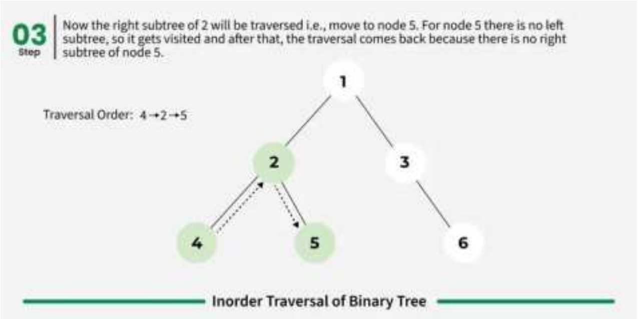


02 Step

As the left subtree of 2 is visited completely, now it visits node 2 before moving to its right subtree.

Traversal Order: 4→2

Inorder Traversal of Binary Tree

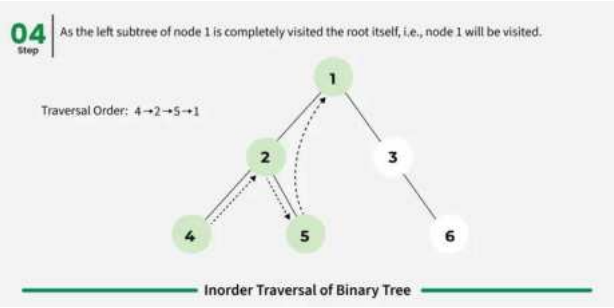


03 Step

Now the right subtree of 2 will be traversed i.e., move to node 5. For node 5 there is no left subtree, so it gets visited and after that, the traversal comes back because there is no right subtree of node 5.

Traversal Order: 4→2→5

Inorder Traversal of Binary Tree

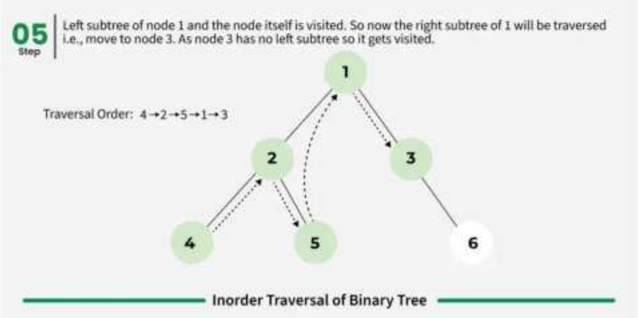


04 Step

As the left subtree of node 1 is completely visited the root itself, i.e., node 1 will be visited.

Traversal Order: 4→2→5→1

Inorder Traversal of Binary Tree

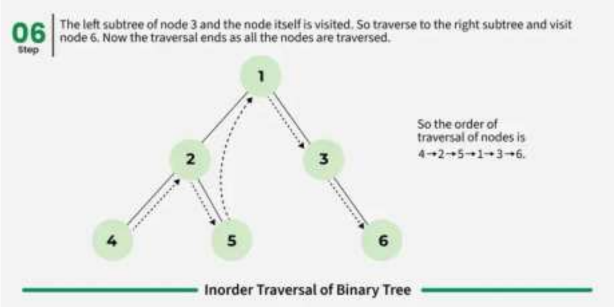


05 Step

Left subtree of node 1 and the node itself is visited. So now the right subtree of 1 will be traversed i.e., move to node 3. As node 3 has no left subtree so it gets visited.

Traversal Order: 4→2→5→1→3

Inorder Traversal of Binary Tree



06 Step

The left subtree of node 3 and the node itself is visited. So traverse to the right subtree and visit node 6. Now the traversal ends as all the nodes are traversed.

So the order of traversal of nodes is 4→2→5→1→3→6.

Inorder Traversal of Binary Tree

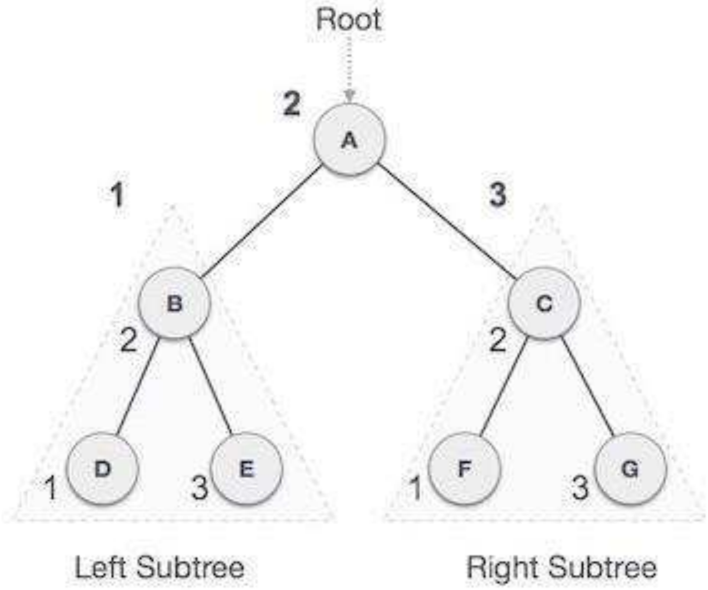
Inorder Traversal

Below is the idea to solve the problem:

*At first traverse **left subtree** then visit the **root** and then traverse the **right subtree**.*
Follow the below steps to implement the idea:

- Traverse left subtree
- Visit the root and print the data.
- Traverse the right subtree

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order. To get the decreasing order visit the right, root, and left subtree.



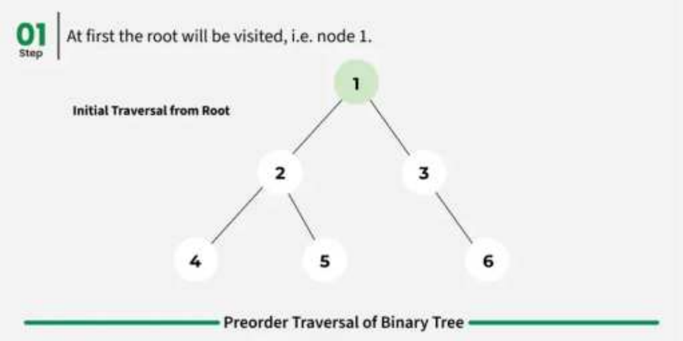
We start from **A**, and following in-order traversal, we move to its left subtree **B**.**B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be −
D → B → E → A → F → C → G

Preorder Traversal:

Below is the idea to solve the problem:

*At first visit the **root** then traverse **left subtree** and then traverse the **right subtree**.*
Follow the below steps to implement the idea:

- Visit the root and print the data.
- Traverse left subtree
- Traverse the right subtree

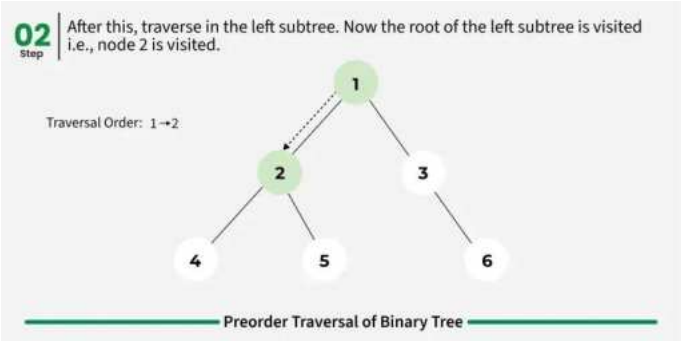


01 Step

At first the root will be visited, i.e. node 1.

Initial Traversal from Root

Preorder Traversal of Binary Tree

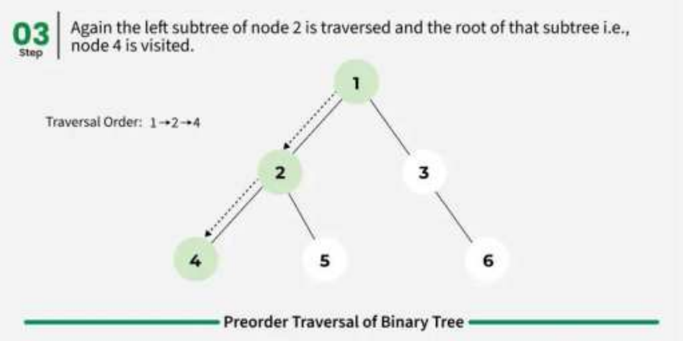


02 Step

After this, traverse in the left subtree. Now the root of the left subtree is visited i.e., node 2 is visited.

Traversal Order: 1→2

Preorder Traversal of Binary Tree

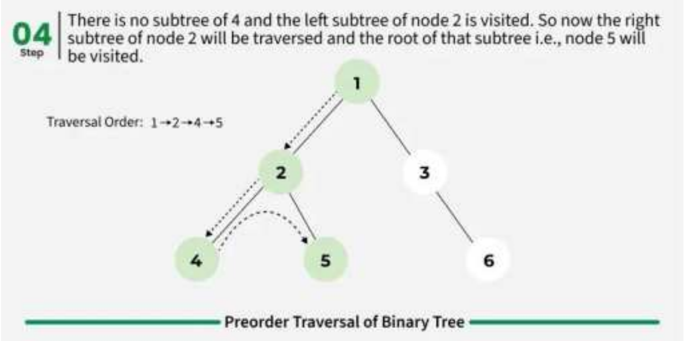


03 Step

Again the left subtree of node 2 is traversed and the root of that subtree i.e., node 4 is visited.

Traversal Order: 1→2→4

Preorder Traversal of Binary Tree

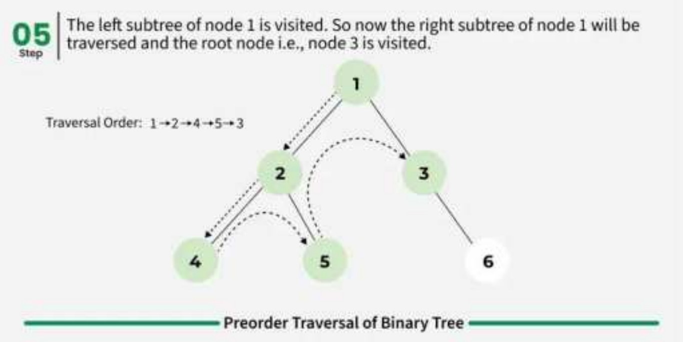


04 Step

There is no subtree of 4 and the left subtree of node 2 is visited. So now the right subtree of node 2 will be traversed and the root of that subtree i.e., node 5 will be visited.

Traversal Order: 1→2→4→5

Preorder Traversal of Binary Tree

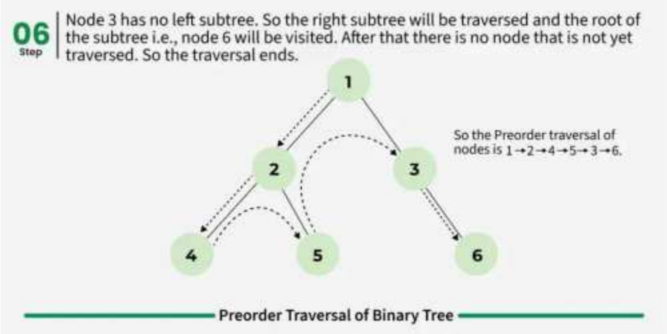


05 Step

The left subtree of node 1 is visited. So now the right subtree of node 1 will be traversed and the root node i.e., node 3 is visited.

Traversal Order: 1→2→4→5→3

Preorder Traversal of Binary Tree

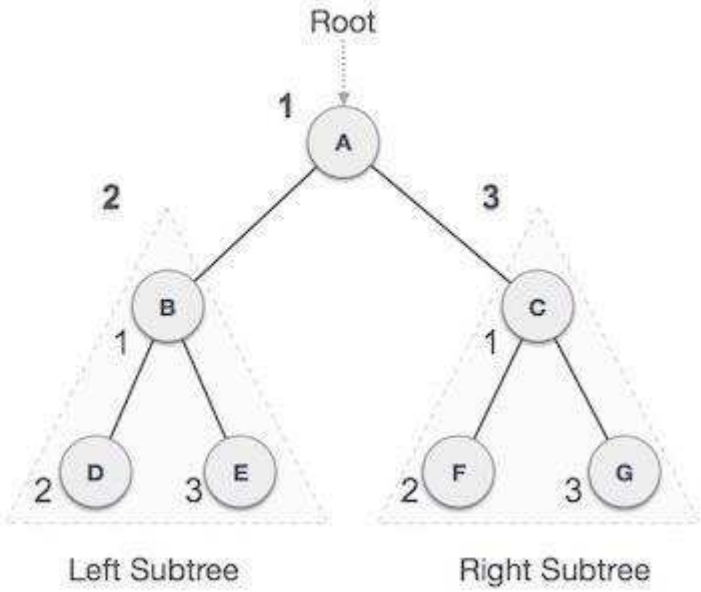


06 Step

Node 3 has no left subtree. So the right subtree will be traversed and the root of the subtree i.e., node 6 will be visited. After that there is no node that is not yet traversed. So the traversal ends.

So the Preorder traversal of nodes is 1→2→4→5→3→6.

Preorder Traversal of Binary Tree



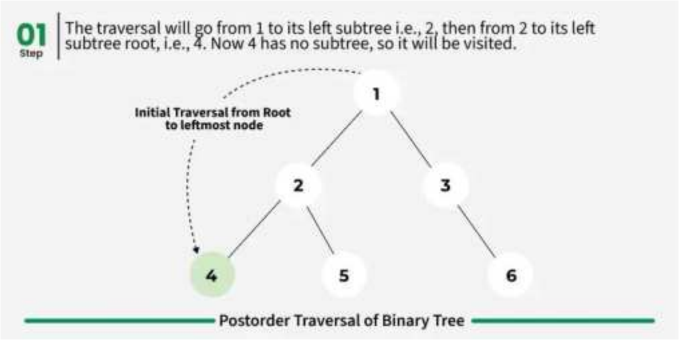
We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −
A → B → D → E → C → F → G

Postorder Traversal:

Below is the idea to solve the problem:

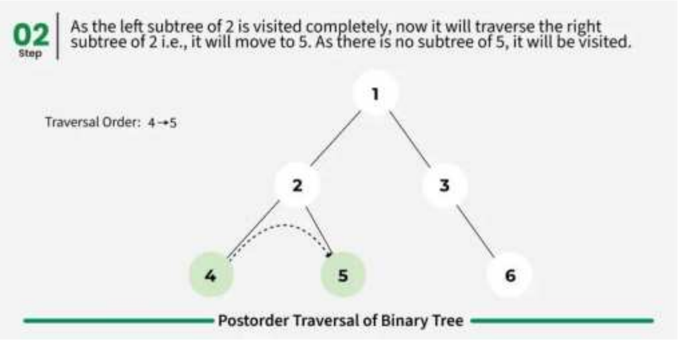
*At first traverse **left subtree** then traverse the **right subtree** and then visit the **root**.*
Follow the below steps to implement the idea:

- Traverse left subtree
- Traverse the right subtree
- Visit the root and print the data.



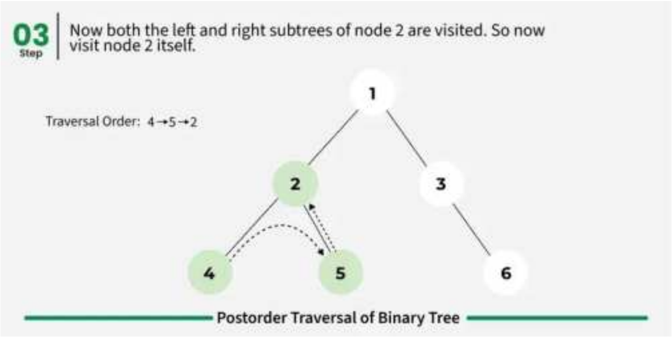
01 Step — The traversal will go from 1 to its left subtree i.e., 2, then from 2 to its left subtree root, i.e., 4. Now 4 has no subtree, so it will be visited.

Initial Traversal from Root to leftmost node
1
2
3
4
5
6
Postorder Traversal of Binary Tree



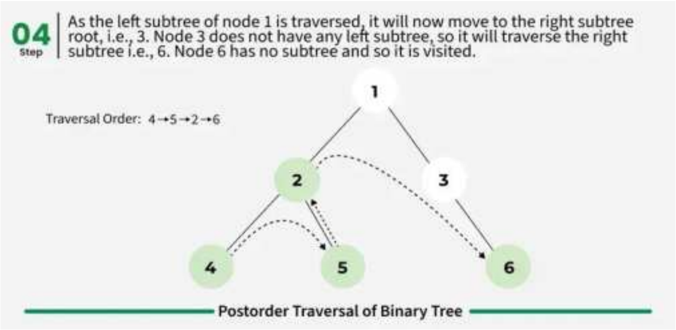
02 Step — As the left subtree of 2 is visited completely, now it will traverse the right subtree of 2 i.e., it will move to 5. As there is no subtree of 5, it will be visited.

Traversal Order: 4→5



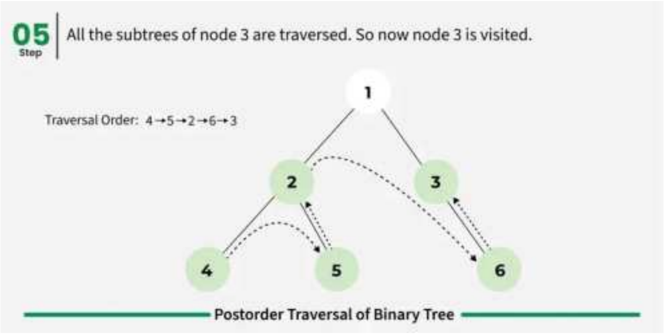
03 Step — Now both the left and right subtrees of node 2 are visited. So now visit node 2 itself.

Traversal Order: 4→5→2



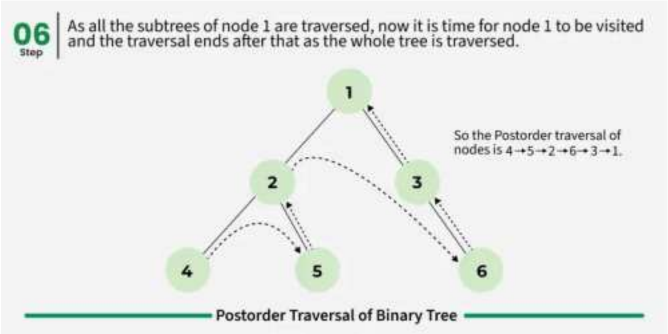
04 Step — As the left subtree of node 1 is traversed, it will now move to the right subtree root, i.e., 3. Node 3 does not have any left subtree, so it will traverse the right subtree i.e., 6. Node 6 has no subtree and so it is visited.

Traversal Order: 4→5→2→6



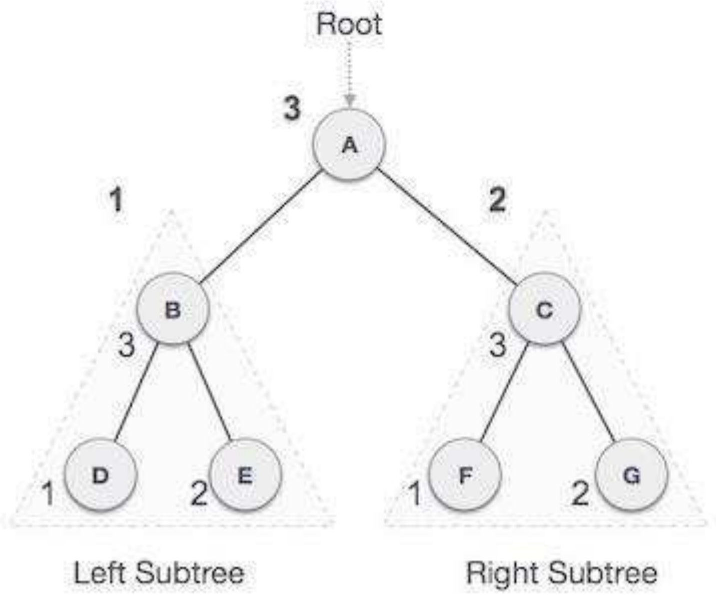
05 Step — All the subtrees of node 3 are traversed. So now node 3 is visited.

Traversal Order: 4→5→2→6→3



06 Step — As all the subtrees of node 1 are traversed, now it is time for node 1 to be visited and the traversal ends after that as the whole tree is traversed.

So the Postorder traversal of nodes is 4→5→2→6→3→1.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Searching in Binary Search Tree (BST)

Given a **BST**, the task is to search a node in this **BST**.

*For searching a value in BST, consider it as a sorted array. Now we can easily perform search operation in BST using the **Binary Search Algorithm***

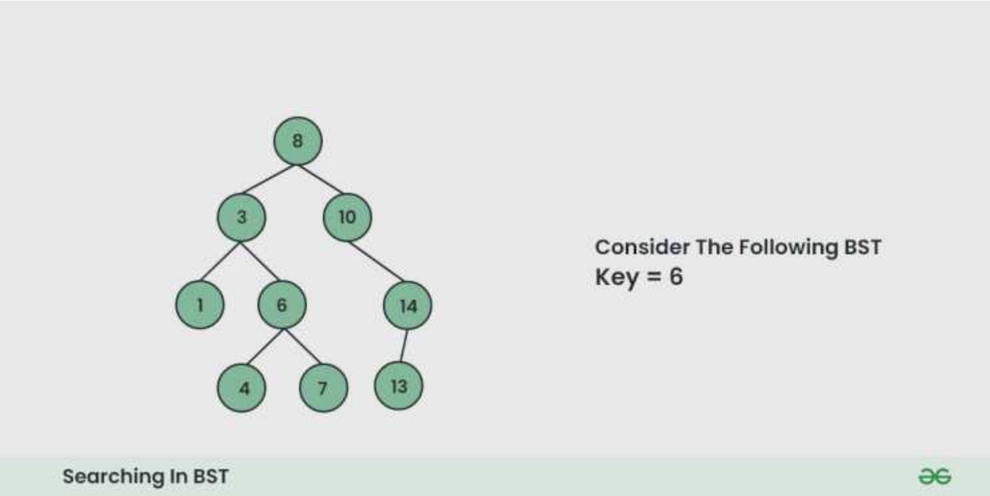
Algorithm to search for a key in a given Binary Search Tree:

Let's say we want to search for the number **X,** We start at the root. Then:

- We compare the value to be searched with the value of the root.
    - If it's equal we are done with the search if it's smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
- Repeat the above step till no more traversal is possible
- If at any iteration, key is found, return True. Else False.

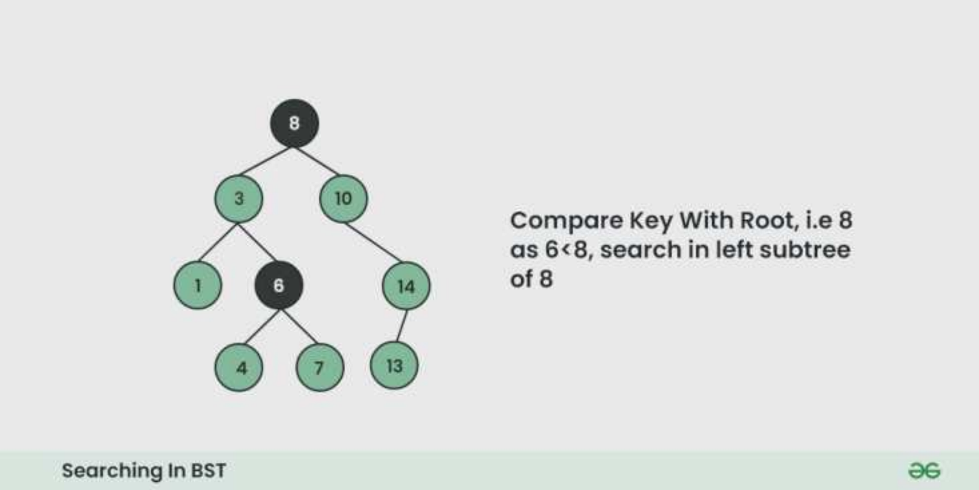
Illustration of searching in a BST:

See the illustration below for a better understanding:



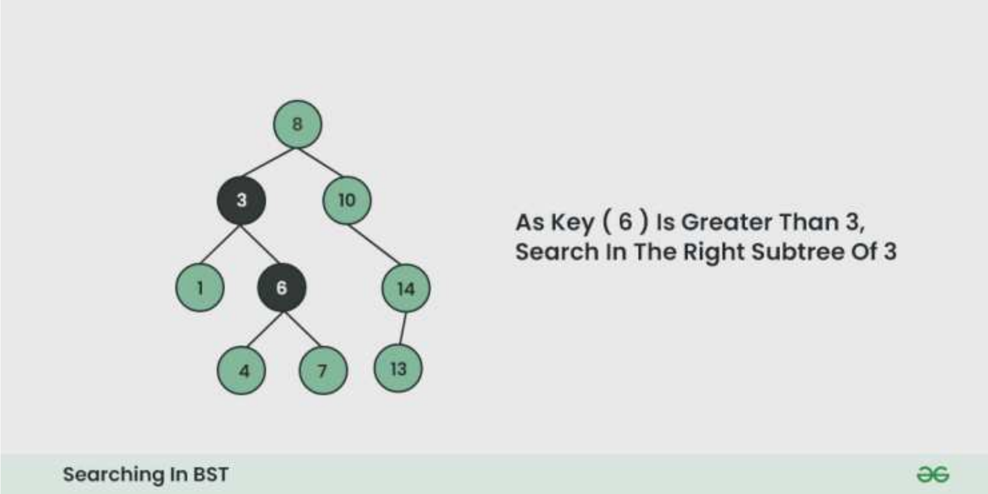
Consider The Following BST
Key = 6

Searching In BST



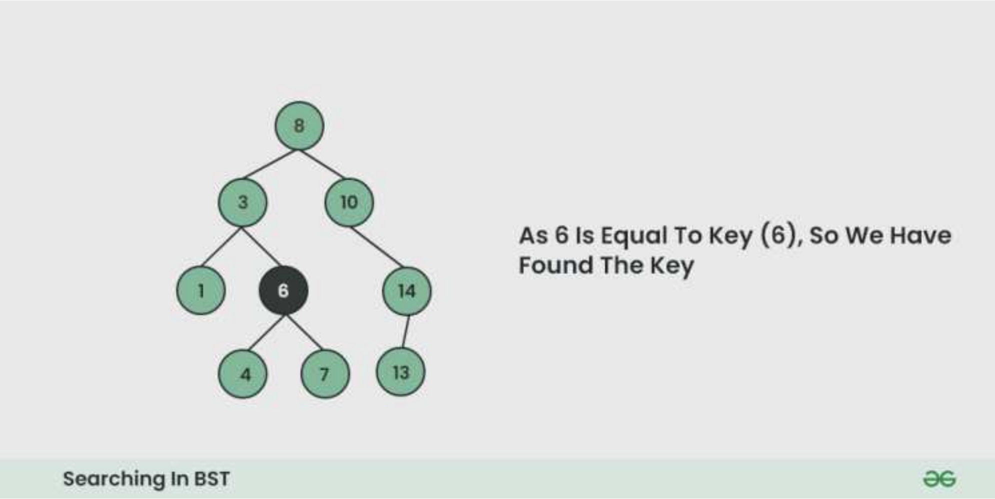
Compare Key With Root, i.e 8
as 6<8, search in left subtree
of 8

Searching In BST



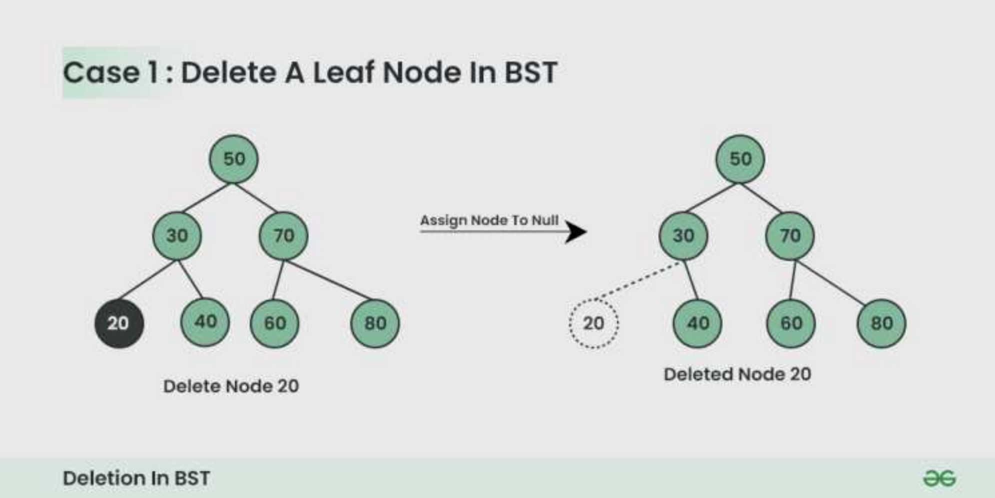
As Key (6) Is Greater Than 3,
Search In The Right Subtree Of 3

Searching In BST



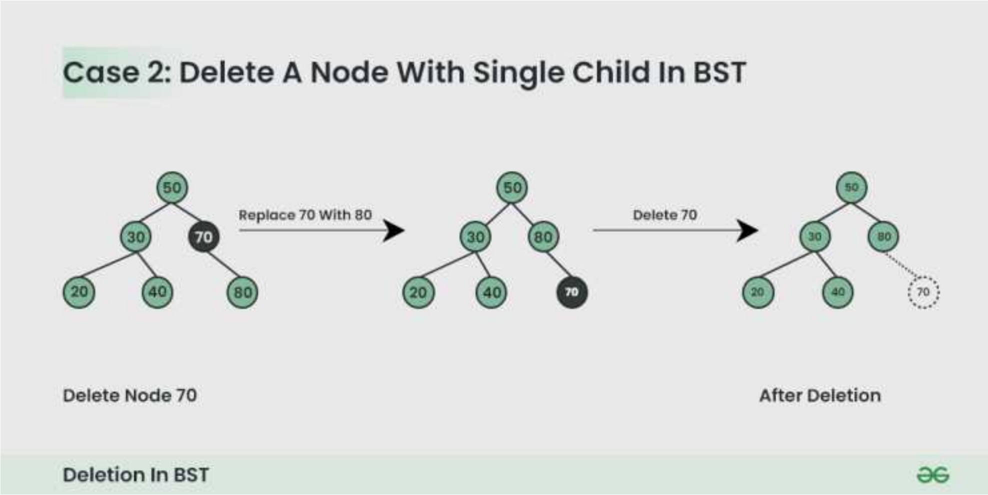
Deletion in Binary Search Tree (BST)

Given a **BST**, the task is to delete a node in this **BST**, which can be broken down into 3 scenarios:
Case 1. Delete a Leaf Node in BST



Deletion in BST

Case 2. Delete a Node with Single Child in BST
Deleting a single child node is also simple in BST. ***Copy the child to the node and delete the node***.



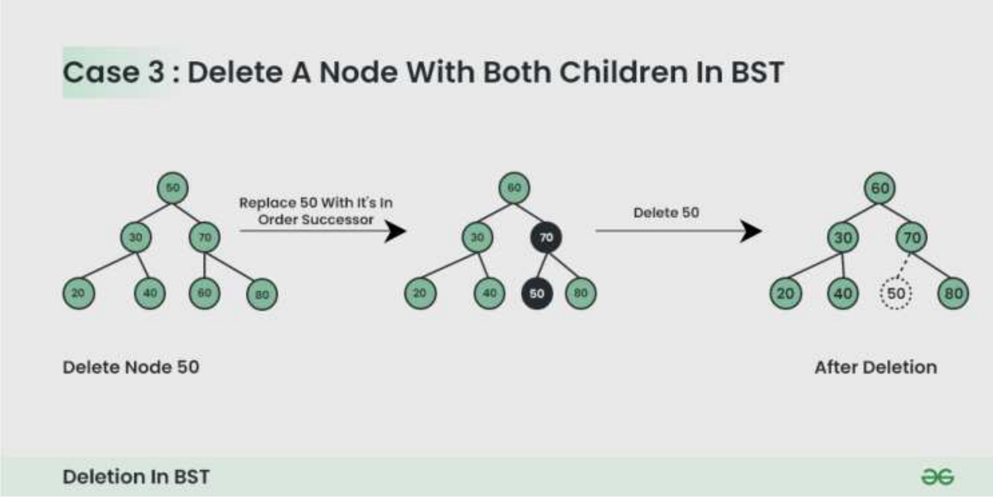
Deletion in BST

Case 3. Delete a Node with Both Children in BST

Deleting a node with both children is not so simple. Here we have to **delete the node is such a way, that the resulting tree follows the properties of a BST.**

The trick is to find the inorder successor of the node. Copy contents of the inorder successor to the node, and delete the inorder successor.

Note: Inorder predecessor can also be used.



Deletion in Binary Tree

Note: Inorder successor is needed only when the right child is not empty. In this particular case, the inorder successor can be obtained by finding the minimum value in the right child of the node.

Advantages of Binary Search Tree:

- BST is fast in insertion and deletion when balanced. It is fast with a time complexity of O(log n).
- BST is also for fast searching, with a time complexity of O(log n) for most operations.

- BST is efficient. It is efficient because they only store the elements and do not require additional memory for pointers or other data structures.
- We can also do range queries – find keys between N and M (N <= M).
- BST code is simple as compared to other data structures.
- BST can automatically sort elements as they are inserted, so the elements are always stored in a sorted order.
- BST can be easily modified to store additional data or to support other operations. This makes it flexible.

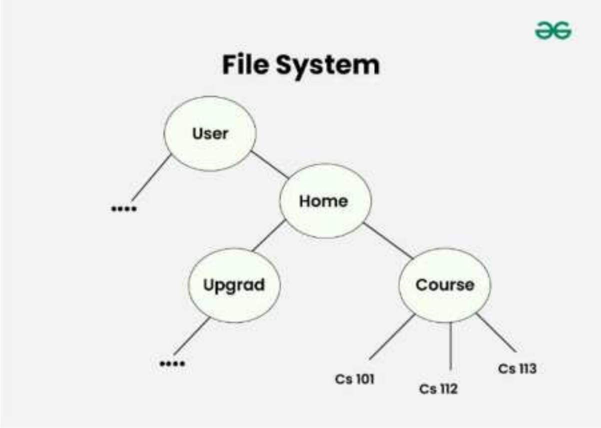
Disadvantages of Binary Search Tree:

- The main disadvantage is that we should always implement a balanced binary search tree. Otherwise the cost of operations may not be logarithmic and degenerate into a linear search on an array.
- They are not well-suited for data structures that need to be accessed randomly, since the time complexity for search, insert, and delete operations is O(log n), which is good for large data sets, but not as fast as some other data structures such as arrays or hash tables.
- A BST can be imbalanced or degenerated which can increase the complexity.
- Do not support some operations that are possible with ordered data structures.
- They are not guaranteed to be balanced, which means that in the worst case, the height of the tree could be O(n) and the time complexity for operations could degrade to O(n).

Applications of tree data structure

Uses of Tree Data Structure:

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.



Hierarchical Structure: trees store information that naturally forms a hierarchy. For example, the file system on a computer: The DOM model of an HTML page is also tree where we have html tag as root, head and body its children and these tags, then have their own children. DNS System is also example where we have natural hierarchy.

Searching Efficiency: For example, in a binary search tree, searching for a value takes time proportional to the logarithm of the number of elements, which is much faster than searching in a linear data structure like an array or a linked list.

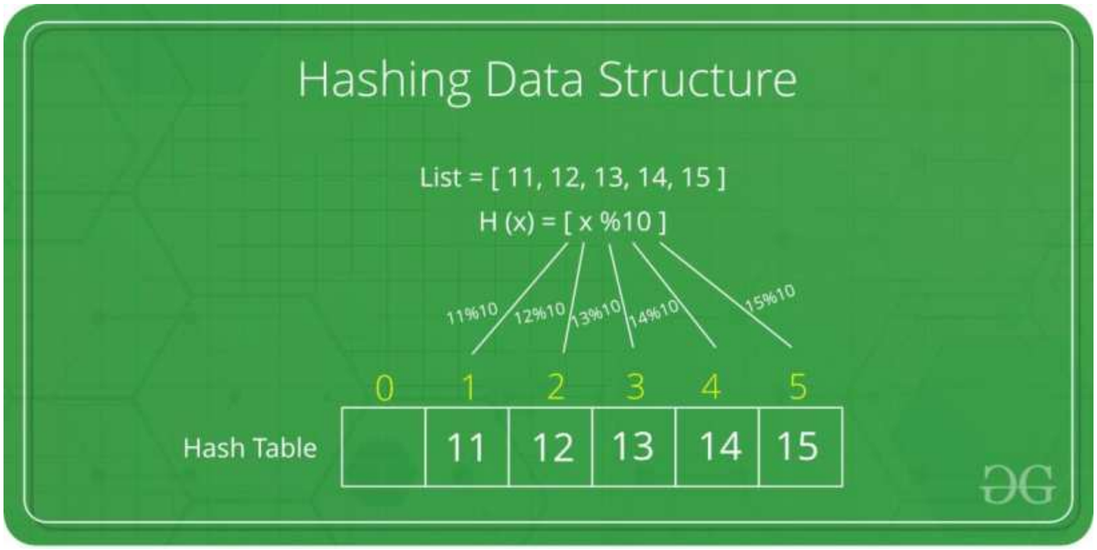
Dynamic Data: Trees are dynamic data structures, which means that they can grow and shrink as needed.

Efficient Insertion and Deletion: Trees provide efficient algorithms for inserting and deleting data.

Easy to Implement: Trees are relatively easy to implement, especially when compared to other data structures like graphs.

Hashing in Data Structure

Hashing is a technique used in data structures to store and retrieve data efficiently. It involves using a **hash function** to map data items to a fixed-size array which is called a **hash table**.



Hashing Data Structure

A **hash table** is also known as a hash map. It is a data structure that stores **key-value pairs**. It uses a **hash function** to map **keys** to a fixed-size array, called a **hash table**. This allows in faster **search**, **insertion**, and **deletion** operations.

The **hash function** is a function that takes a **key** and returns an **index** into the **hash table**. The goal of a hash function is to distribute keys evenly across the hash table, minimizing collisions (when two keys map to the same index).

Common hash functions include:

- **Division Method:** Key % Hash Table Size
- **Multiplication Method:** (Key * Constant) % Hash Table Size
- **Universal Hashing:** A family of hash functions designed to minimize collisions

Applications of Hashing

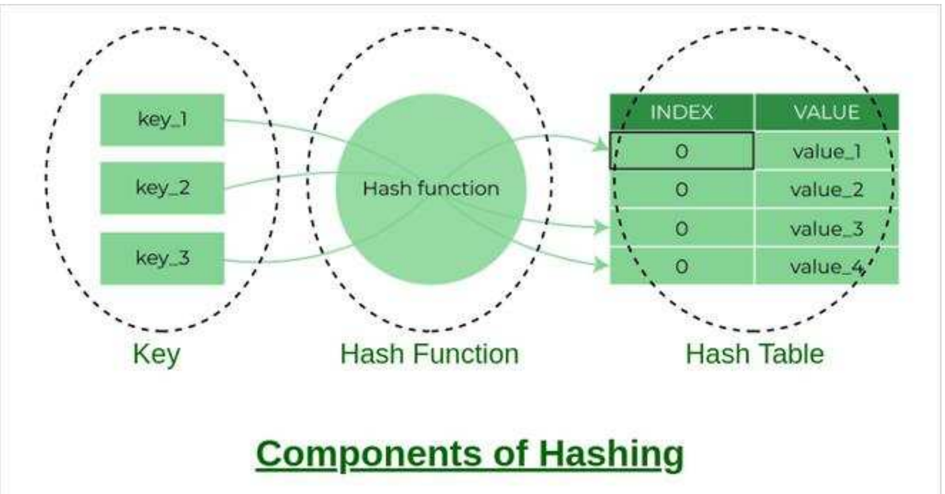
Hash tables are used in a wide variety of applications, including:

- **Databases:** Storing and retrieving data based on unique keys
- **Caching:** Storing frequently accessed data for faster retrieval
- **Symbol Tables:** Mapping identifiers to their values in programming languages
- **Network Routing:** Determining the best path for data packets

Components of Hashing

There are majorly three components of hashing:

1. **Key:** A **Key** can be anything string or integer that 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:** The **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 a data structure that maps keys to values using a hash function. Hash stores the data in an associative manner in an array where each data value has its unique index.

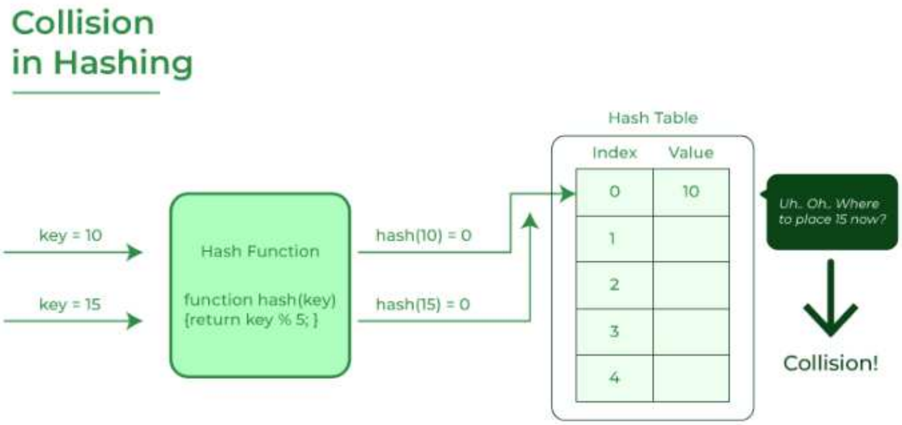


INDEX	VALUE
0	value_1
0	value_2
0	value_3
0	value_4

Components of Hashing

Collision in hashing

The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.



Collision in Hashing

Advantages of Hashing in Data Structures

- **Key-value support:** Hashing is ideal for implementing key-value data structures.
- **Fast data retrieval:** Hashing allows for quick access to elements with constant-time complexity.
- **Efficiency:** Insertion, deletion, and searching operations are highly efficient.
- **Memory usage reduction:** Hashing requires less memory as it allocates a fixed space for storing elements.
- **Scalability:** Hashing performs well with large data sets, maintaining constant access time.
- **Security and encryption:** Hashing is essential for secure data storage and integrity verification.

Handling of Collisions

There are mainly two methods to handle collision:

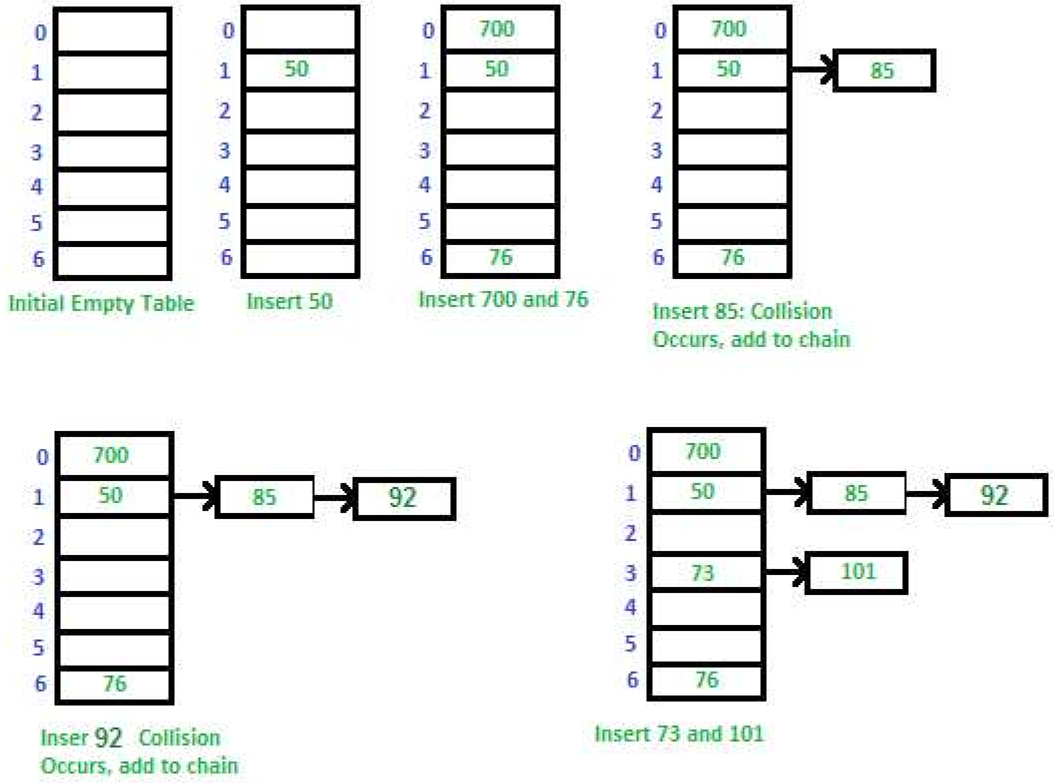
- Separate Chaining
- Open Addressing

Separate Chaining:

The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

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 7**" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101



Initial Empty Table

Insert 50

Insert 700 and 76

Insert 85: Collision Occurs, add to chain

Inser 92 Collision Occurs, add to chain

Insert 73 and 101

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.

Data Structures For Storing Chains:

1. **Linked lists**
2. **Dynamic Sized Arrays** (Vectors in C++, ArrayList in Java, list in Python)
3. **Self Balancing BST** (AVL Trees, Red-Black Trees)

SORTING

A **Sorting Algorithm** is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Insertion sort

Insertion sort is a very simple method to sort numbers in an ascending or descending order.

It is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element that is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$, where **n** is the number of items.

Insertion Sort Algorithm

To sort an array of size N in ascending order:

- Iterate from arr[1] to arr[N] over the array.
- Compare the current element (key) to its predecessor.
- If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Working of Insertion Sort algorithm

Consider an example: arr[]: ***{12, 11, 13, 5, 6}***

12	**11**	**13**	**5**	**6**

First Pass:

Initially, the first two elements of the array are compared in insertion sort.

12 **11** 13 5 6

Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
So, for now 11 is stored in a sorted sub-array.

11 **12** 13 5 6

Second Pass:

Now, move to the next two elements and compare them

11 **12** **13** 5 6

Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

Now, two elements are present in the sorted sub-array which are **11** and **12**
Moving forward to the next two elements which are 13 and 5

11 12 **13** **5** 6

Both 5 and 13 are not present at their correct place so swap them

11 12 5 **13** 6

After swapping, elements 12 and 5 are not sorted, thus swap again

11 5 **12** 13 6

Here, again 11 and 5 are not sorted, hence swap again

5 **11** 12 13 6

Here, 5 is at its correct position

Fourth Pass:

Now, the elements which are present in the sorted sub-array are **5, 11** and **12**
Moving to the next two elements 13 and 6

5 11 12 **13** **6**

Clearly, they are not sorted, thus perform swap between both

5 11 12 **6** **13**

Now, 6 is smaller than 12, hence, swap again

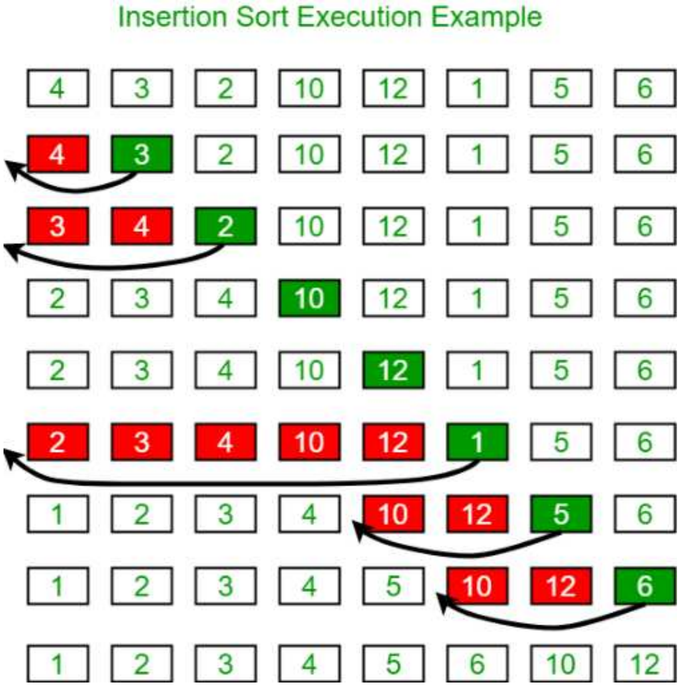
5 11 **6** **12** 13

Here, also swapping makes 11 and 6 unsorted hence, swap again

5 6 11 12 13

Finally, the array is completely sorted.

Illustrations:



Implementation of Insertion Sort Algorithm

Since insertion sort is an in-place sorting algorithm, the algorithm is implemented in a way where the key element – which is iteratively chosen as every element in the array – is compared with it consequent elements to check its position. If the key element is less than its successive element, the swapping is not done. Otherwise, the two elements compared will be swapped and the next element will be chosen as the key element.

```
// C++ program for insertion sort

#include <iostream>
using namespace std;

// Function to sort an array using
// insertion sort
void insertionSort(int arr[], int n)
```

```
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key,
        // to one position ahead of their
        // current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// A utility function to print an array
// of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int N = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, N);
    printArray(arr, N);

    return 0;
}
```

Output

```
5 6 11 12 13
```

Time Complexity: O(N^2)
Auxiliary Space: O(1)

Complexity of insertion sort

Time Complexity of Insertion Sort

- The **worst-case** time complexity of the Insertion sort is **O(N^2)**
- The **average case** time complexity of the Insertion sort is **O(N^2)**
- The time complexity of the **best case** is **O(N)**.

Space Complexity of Insertion Sort

The auxiliary space complexity of Insertion Sort is **O(1)**

Characteristics of Insertion Sort

- This algorithm is one of the simplest algorithms with a simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets that are already partially sorted.