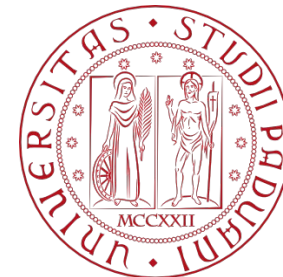


COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

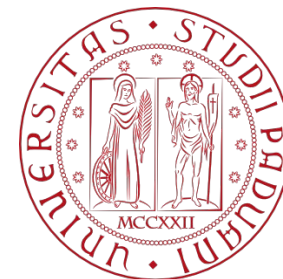
luigi.rizzo@unipd.it

October 2024-January 2025



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Queues, Stacks, Trees



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Agenda



- Stacks
- Queues
- Trees

What is Stack?

Stack is a linear data structure that follows a particular order in which the operations are performed. The order is LIFO (Last In First Out). LIFO implies that the element that is inserted last, comes out first.

A stack is a linear data structure in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.

To implement the stack, it is required to maintain the pointer to the top of the stack, which is the element last inserted and the next/first to be extracted because we can access the elements only on the top of the stack.

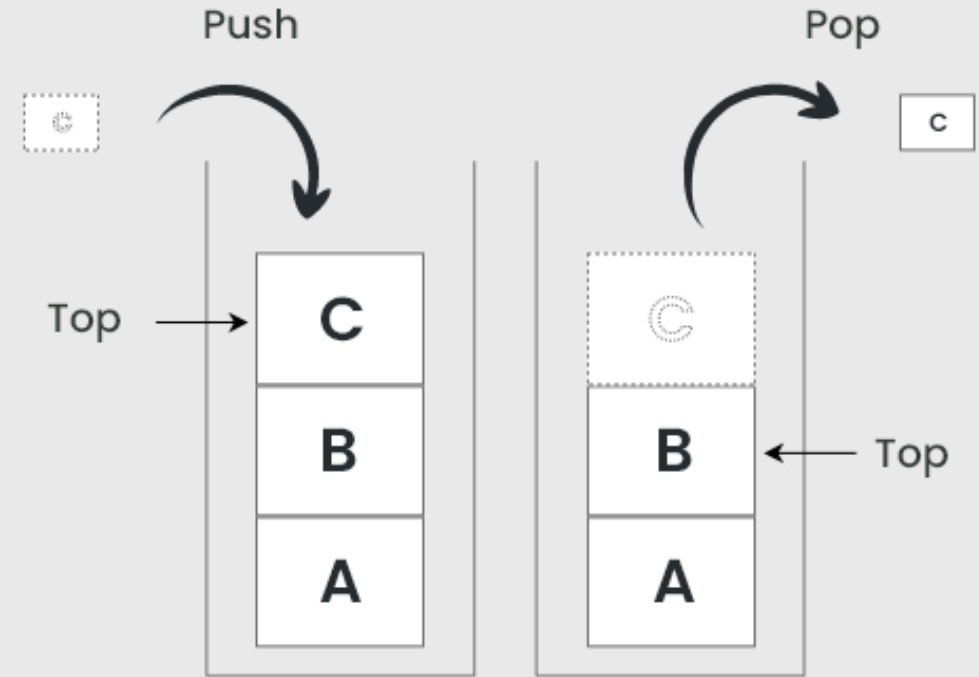
There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. It can be simply seen to follow LIFO (Last In First Out) order.

Stack – LIFO strategy



This strategy states that the element that is inserted last will come out first.

Stack Data Structure

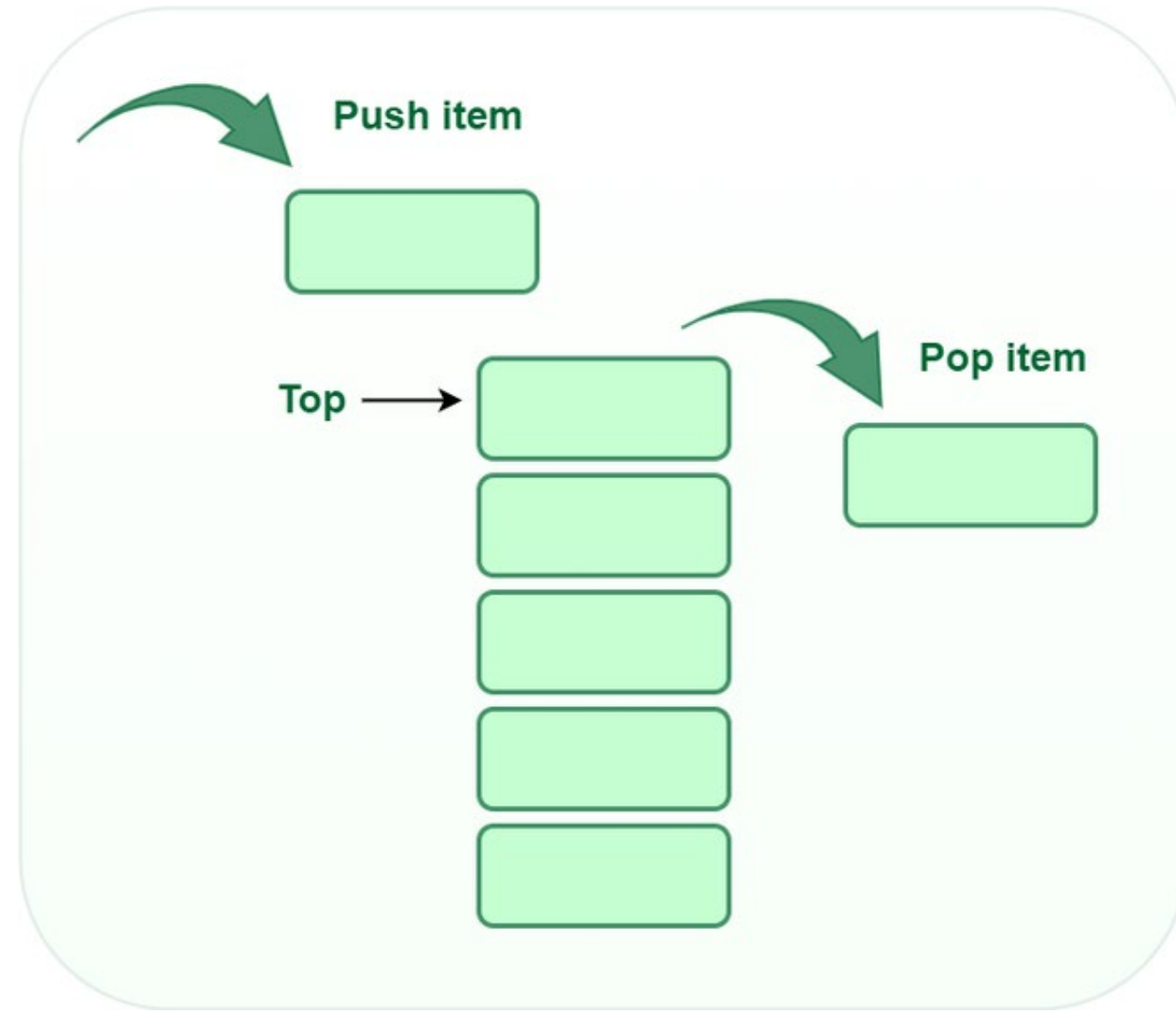


Basic operation on Stack



In order to make manipulations in a stack, there are certain operations.

- `push()` to insert an element into the stack
- `pop()` to remove an element from the stack
- `top()` Returns the top element of the stack.
- `isEmpty()` returns true if stack is empty else false.
- `size()` returns the size of stack.



- Push:
 - Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.
- Pop:
 - Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- Top:
 - Returns the top element of the stack.
- isEmpty:
 - Returns true if the stack is empty, else false.

Time complexity: $O(1)$

- Fixed Size Stack:
 - As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.
- Dynamic Size Stack:
 - A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Implementation of Stack



A stack can be implemented using an array or a linked list.

In an array-based implementation, the push operation is implemented by incrementing the index of the top element and storing the new element at that index. The pop operation is implemented by decrementing the index of the top element and returning the value stored at that index.

In a linked list-based implementation, the push operation is implemented by creating a new node with the new element and setting the next pointer of the new node to the current top node and the top pointer to the new node. The pop operation is implemented by setting the top pointer to the next pointer of the current top node and returning the value of the current top node.

Implementing Stack using arrays



```
#include <limits.h>
```

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

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

```
// A structure to represent a stack
```

```
struct Stack {
```

```
    int top;
```

```
    unsigned int capacity;
```

```
    int* array;
```

```
};
```

Implementing Stack using arrays



// function to create a stack of given capacity. It initializes size of stack as 0

struct Stack createStack(unsigned int capacity)*

{

struct Stack stack = (struct Stack*)malloc(sizeof(struct Stack));*

stack->capacity = capacity;

stack->top = -1;

stack->array = (int)malloc(stack->capacity * sizeof(int));*

return stack;

}

Implementing Stack using arrays



// Stack is full when top is equal to the last index

```
bool isFull(struct Stack* stack)  
{  
    return(stack->top == stack->capacity - 1);  
}
```

// Stack is empty when top is equal to -1

```
bool isEmpty(struct Stack* stack)  
{  
    return(stack->top == -1);  
}
```

Implementing Stack using arrays



// Function to add an item to stack. It increases top by 1

```
int push(struct Stack* stack, int item)  
{  
    if (isFull(stack))  
        return(-1);  
    stack->array[++stack->top] = item;  
    printf("%d pushed to stack\n", item);  
    return(0);  
}
```

Implementing Stack using arrays



// Function to remove an item from stack. It decreases top by 1

```
int pop(struct Stack* stack)  
{  
    if (isEmpty(stack))  
        return(INT_MIN);  
    return stack->array[stack->top--];  
}
```

Implementing Stack using arrays



```
// Function to return the top from stack without removing it  
int top(struct Stack* stack)  
{  
    if (isEmpty(stack))  
        return(INT_MIN);  
    return stack->array[stack->top];  
}
```

Implementing Stack using arrays



// Program to test functions defined in the previous slides

```
int main()  
{  
    struct Stack* stack = createStack(100);  
  
    push(stack, 10);  
    push(stack, 25);  
    push(stack, 5);  
    push(stack, 15);  
    printf("%d popped from stack\n", pop(stack));  
    printf("%d at the top of the stack\n", top(stack));  
    return(0);  
}
```


Implementing Stack using arrays



- Advantages of array implementation:
 - Easy to implement.
 - Memory is saved as pointers are not involved.
- Disadvantages of array implementation:
 - It is not dynamic i.e., it doesn't grow and shrink depending on needs at runtime.
 - The total size of the stack must be defined beforehand.

Implementing Stack using linked lists



```
#include <limits.h>
```

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

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

```
// A structure to represent a stack
```

```
struct StackNode {
```

```
    int data;
```

```
    struct StackNode* next;
```

```
};
```

```
int isEmpty(struct StackNode* head)
```

```
{
```

```
    return(!head);
```

```
}
```

Implementing Stack using linked lists



```
void push(struct StackNode** head, struct StackNode* newNode)  
{  
    newNode->next = *head;  
    *head = newNode;  
    printf("%d pushed to stack\n", newNode->data);  
}
```

Implementing Stack using linked lists



```
int pop(struct StackNode** head)
{
    if (isEmpty(* head))
        return(INT_MIN);
    struct StackNode* tmp = * head;
    * head = (* head)->next;
    int popped = tmp->data;
    free(tmp);

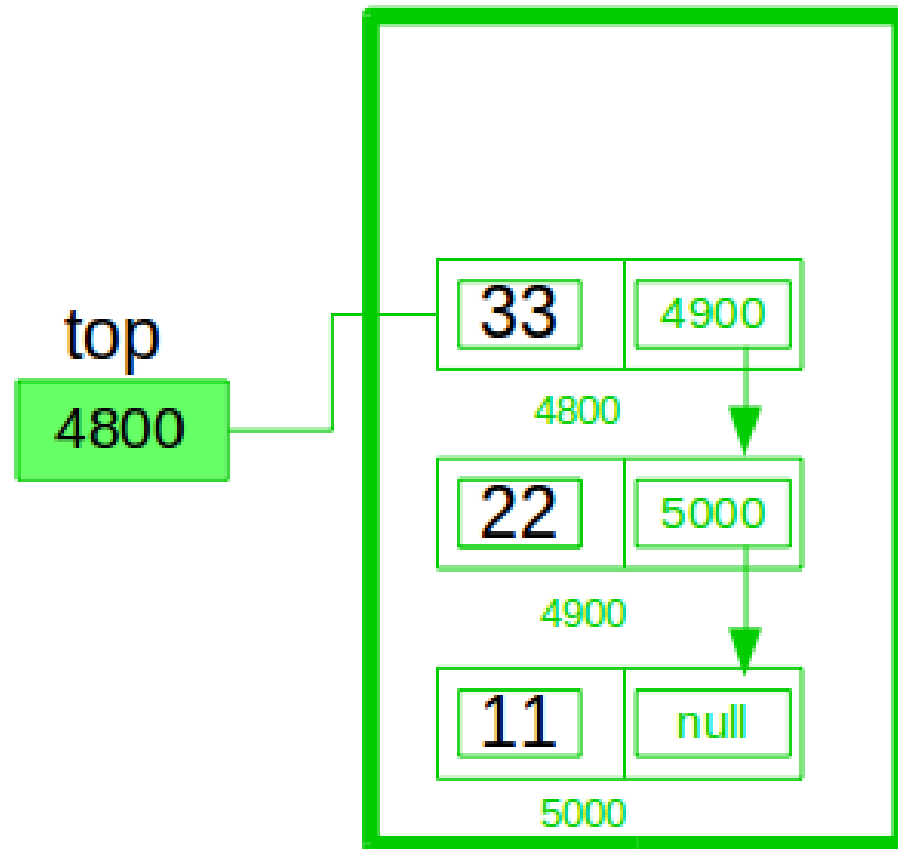
    return(popped);
}
```

Implementing Stack using linked lists



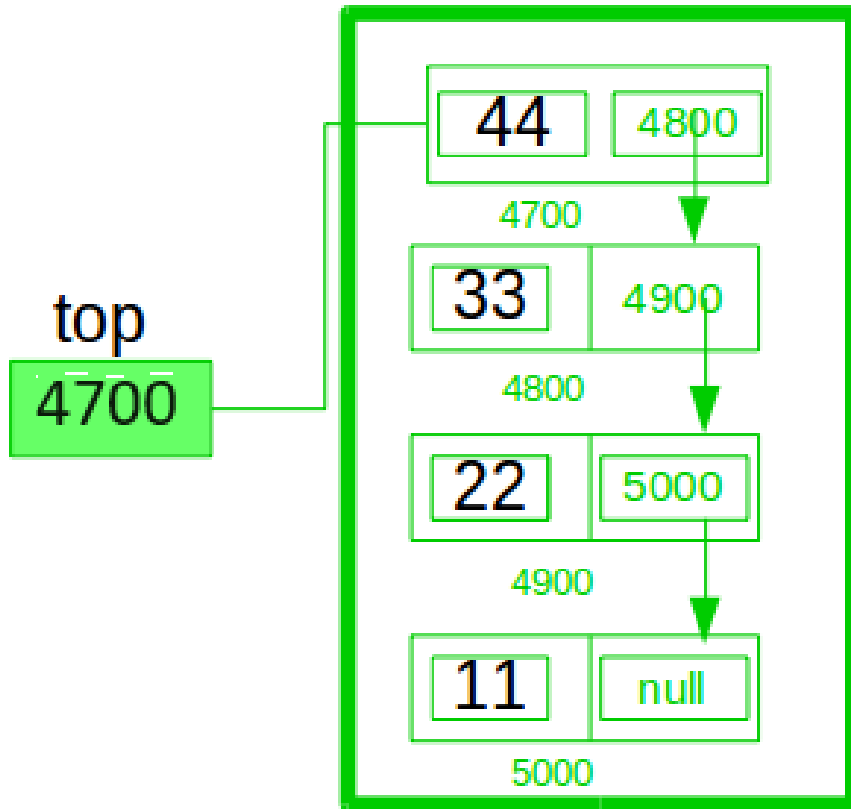
```
int top(struct StackNode* head)
{
    if (isEmpty(head))
        return(INT_MIN);
    return(head->data);
}
```

Implementing Stack using linked lists



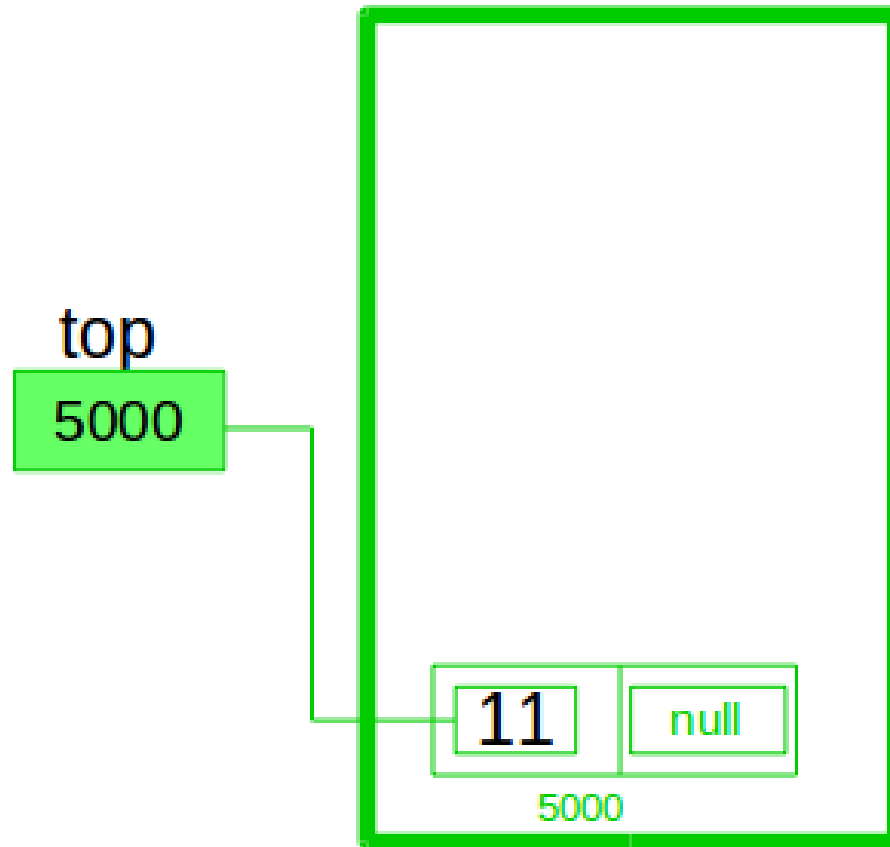
Initial Stack Having Three element
And top have address 4800

Implementing Stack using linked lists



First create a tmp node and
assign 44 into data field then
put top in next field then assign
tmp into top

Implementing Stack using linked lists



```
Pop three elements  
tmp = top;  
top = top->next;  
free(tmp)
```


Implementing Stack using linked lists



- Advantages of Linked List implementation:
 - The linked list implementation of a stack can grow and shrink according to the needs at runtime.
 - It is used in many virtual machines like JVM.
- Disadvantages of Linked List implementation:
 - Requires extra memory due to the involvement of pointers.
 - Random accessing is not possible in stack.

Write a program that reads in a sequence of characters and prints them in reverse order. Use a stack managed by a linked list.

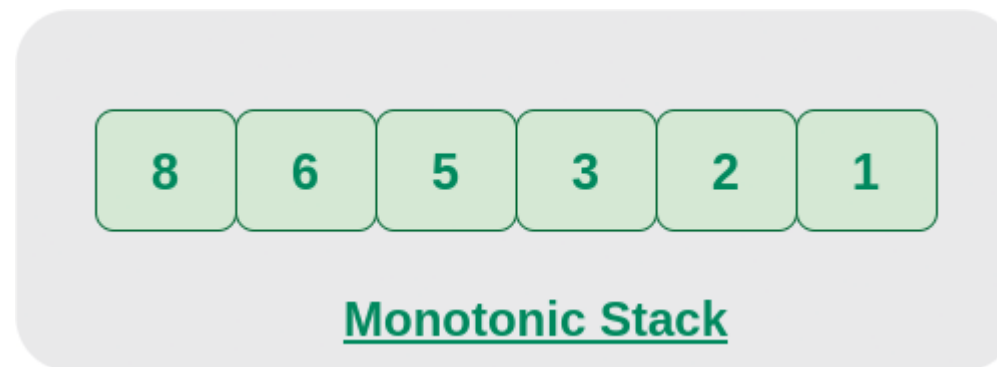
esicrexe ysae

- listInizialization
- createNode
- push
- push

What is a Monotonic Stack?

A monotonic stack is a stack whose elements are monotonically increasing or decreasing. It contains all qualities that a typical stack has and its elements are all monotonic decreasing or increasing.

The monotonic stack maintains monotonicity while popping elements when a new item is pushed into the stack.



A stack is called a monotonic stack if all the elements starting from the bottom of the stack is either in increasing or in decreasing order.

There are 2 types of monotonic stacks:

- Monotonic Increasing Stack
- Monotonic Decreasing Stack

Monotonic Increasing Stack is a stack in which the elements are in increasing order from the bottom to the top of the stack.

Example: 1, 3, 10, 15, 17

Monotonic Decreasing Stack is a stack in which its elements are in decreasing order from the bottom to the top of the stack.

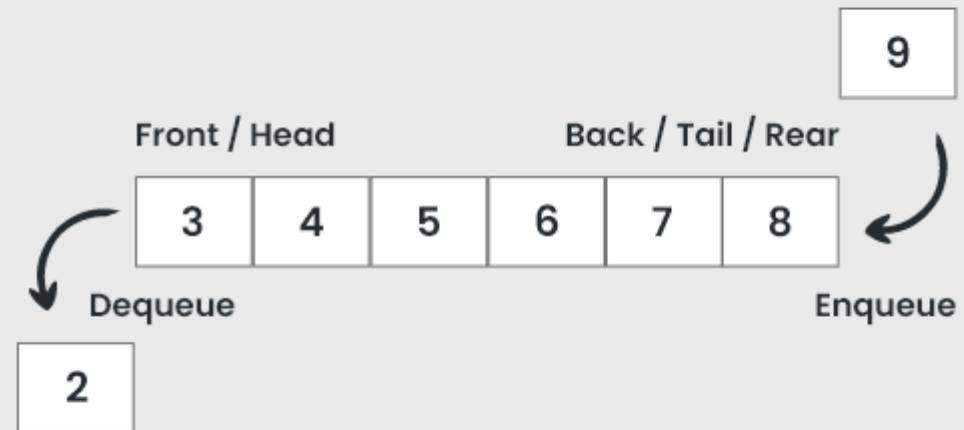
Example: 17, 14, 10, 5, 1

What is Queue Data Structure?

A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order is the one on which the operation is first performed.

Queue Data Structure

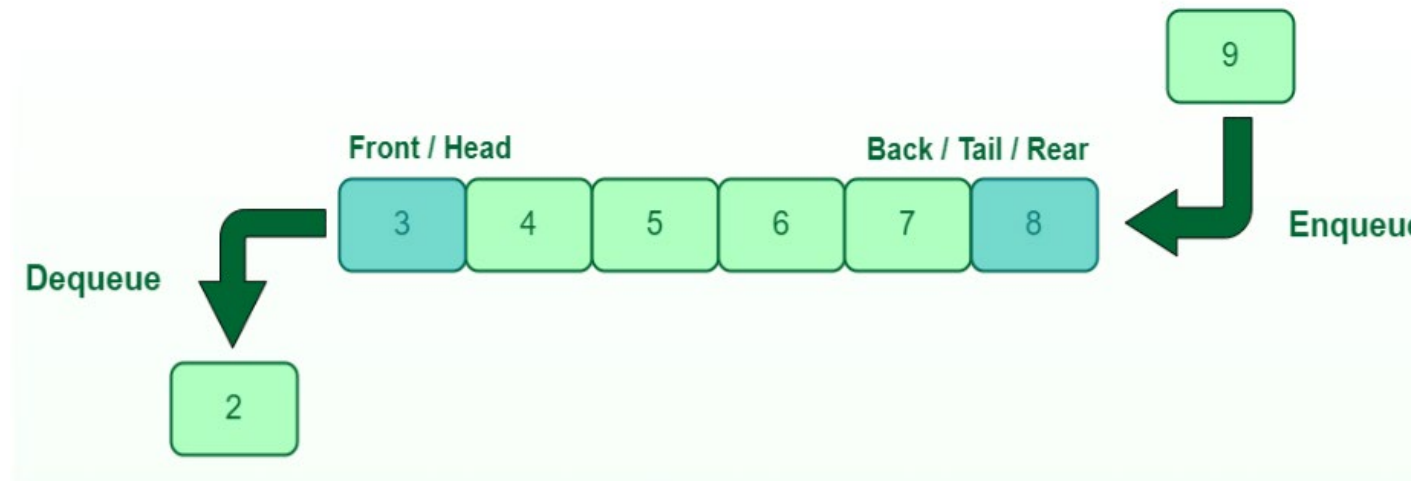


Queue data structure



FIFO Principle of Queue:

A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front or head of the queue, similarly, the position of the last entry in the queue, that is, the one most recently added, is called the rear or tail of the queue. See the below figure.

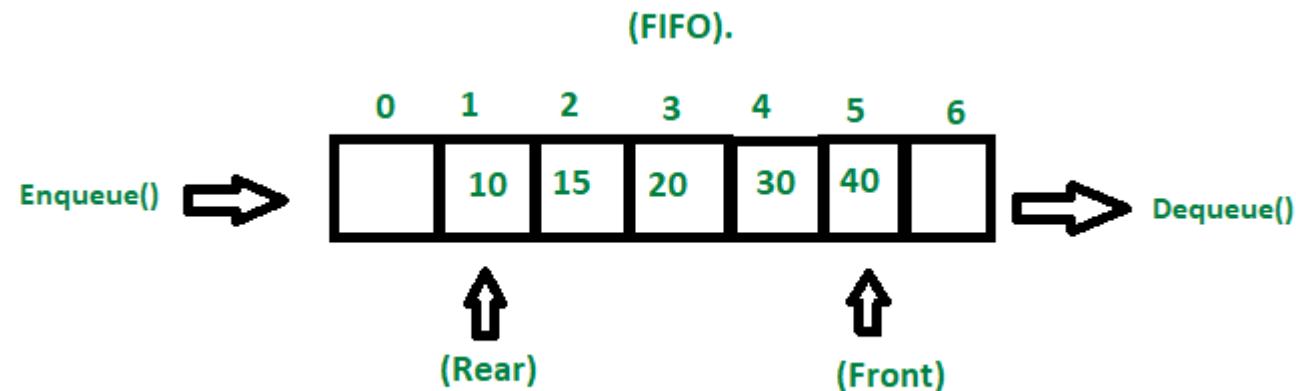


Queue data structure



A queue is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

We define a queue to be a list in which all additions to the list are made at one end, and all deletions/extractions from the list are made at the other end. The element which is first pushed into the order, the delete operation is first performed on that.



Array Representation of Queue



Queues can be represented in an array. Variables used in such case are

- Queue: the name of the array storing queue elements.
- Front: the index where the first element is stored in the array representing the queue.
- Rear: the index where the last element is stored in an array representing the queue.

Array Representation of Queue



```
// A structure to represent a queue  
struct Queue {  
    int front, rear, size;  
    unsigned int capacity;  
    int* array;  
};
```

Array representation of Queue



```
// function to create a queue of given capacity initializing size of queue as 0  
struct Queue* createQueue(unsigned int capacity)  
{  
    struct Queue* queue  
        = (struct Queue*)malloc(sizeof(struct Queue));  
    queue->capacity = capacity;  
    queue->front = queue->size = 0;  
    queue->rear = capacity - 1;  
    queue->array = (int*)malloc(queue->capacity * sizeof(int));  
    return(queue);  
}
```

Array representation of Queue



// Queue is full when size becomes equal to the capacity

```
int isFull(struct Queue* queue)
{
    return (queue->size == queue->capacity);
}
```

// Queue is empty when size is 0

```
int isEmpty(struct Queue* queue)
{
    return (queue->size == 0);
}
```

enqueue() operation in queue adds (or stores) an element to the end of the queue.

The following steps should be taken to enqueue (insert) data into a queue:

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

Array representation of Queue



// Function to add an item to the queue changing rear and size

```
int enqueue(struct Queue* queue, int item)
```

```
{
```

```
    if (isFull(queue))
```

```
        return(-1);
```

```
    queue->rear = (queue->rear + 1) % queue->capacity;
```

```
    queue->array[queue->rear] = item;
```

```
    queue->size = queue->size + 1;
```

```
    printf("%d enqueued to queue\n", item);
```

```
    return(0);
```

```
}
```

Array representation of Queue



dequeue() operation removes (or accesses) the first element from the queue. The following steps are taken to perform the dequeue operation:

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

Array representation of Queue



// Function to remove an item from queue changing front and size

```
int dequeue(struct Queue* queue)
```

```
{
```

```
    if (isEmpty(queue)) {
```

```
        printf("\nQueue is empty\n");
```

```
        return(INT_MIN);
```

```
    }
```

```
    int item = queue->array[queue->front];
```

```
    queue->front = (queue->front + 1) % queue->capacity;
```

```
    queue->size = queue->size - 1;
```

```
    return (item);
```

```
}
```

Array representation of Queue



front() operation returns the element at the front end without removing it.

```
// Function to get front of queue  
int front(struct Queue* queue)  
{  
    if (isempty(queue))  
        return(INT_MIN);  
    return (queue->arr[queue->front]);  
}
```

Array representation of Queue



rear() operation returns the element at the rear end without removing it.

```
// Function to get rear of queue  
int rear(struct Queue* queue)  
{  
    if (isEmpty(queue))  
        return(INT_MIN);  
    return (queue->array[queue->rear]);  
}
```

Array representation of Queue



```
int main()
{
    struct Queue* queue = createQueue(1000);

    enqueue(queue, 16);
    enqueue(queue, 22);
    enqueue(queue, 30);
    enqueue(queue, 13);

    printf("%d dequeued from queue\n", dequeue(queue));
    printf("Front item is %d\n", front(queue));
    printf("Rear item is %d\n", rear(queue));

    return 0;
}
```

Linked lists representation of Queue



A queue can also be represented using following entities:

- Linked-lists
- Pointers
- Structures

```
// A linked list (LL) node to store a queue entry  
struct QNode {  
    int data;  
    struct QNode* next;  
};
```

Linked lists representation of Queue



/ The queue, front stores the front node of LL and rear stores the last node of LL */*

```
struct Queue {  
    struct QNode *front, *rear;  
};
```

// A utility function to create a new LL node.

```
struct QNode* newNode(int k)  
{  
    struct QNode* tmp = (struct QNode*)malloc(sizeof(struct QNode));  
    tmp->data = k;  
    tmp->next = NULL;  
    return(tmp);  
}
```

Linked lists representation of Queue



// A utility function to create an empty queue

```
struct Queue* createQueue()  
{  
    struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));  
    q->front = q->rear = NULL;  
    return(q);  
}
```

Linked lists representation of Queue



```
// The function to add a data k to q
void enQueue(struct Queue* q, int k)
{
    // Create a new LL node
    struct QNode* tmp = newNode(k);

    // If queue is empty, then new node is front and rear both
    if (q->rear == NULL) {
        q->front = q->rear = tmp;
        return;
    }

    // Add the new node at the end of queue and change rear
    q->rear->next = tmp;
    q->rear = tmp;
}
```


Linked lists representation of Queue



```
// Function to dequeue a key from given queue q
struct QNode* deQueue(struct Queue* q)
{
    // If queue is empty, return NULL.
    if (q->front == NULL)
        return(NULL);

    // Store previous front and move front one node ahead
    struct QNode* tmp = q->front;

    q->front = q->front->next;

    // If front becomes NULL, then change rear also as NULL
    if (q->front == NULL)
        q->rear = NULL;

    return(tmp);
}
```

Linked lists representation of Queue

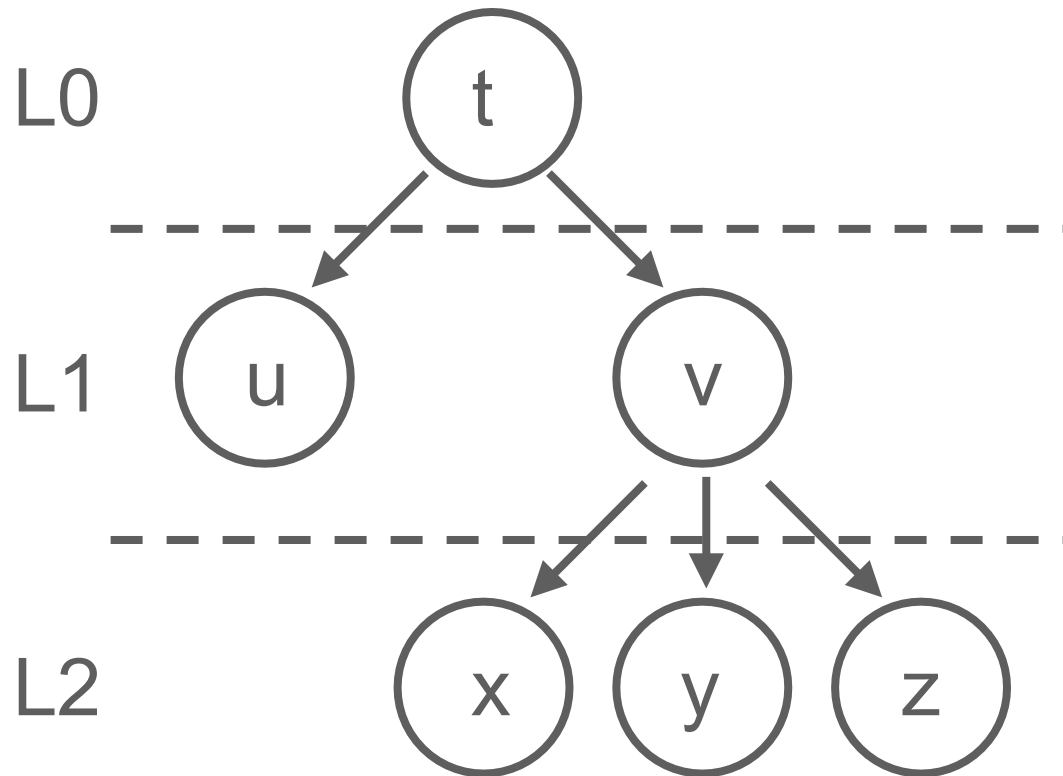


```
int main()
{
    struct Qnode* tmp = NULL;
    struct Queue* q = createQueue();
    enqueue(q, 10);
    enqueue(q, 20);
    tmp = dequeue(q);
    tmp = dequeue(q);
    enqueue(q, 30);
    enqueue(q, 40);
    enqueue(q, 50);
    tmp = dequeue(q);
    printf("Queue Front : %d \n", ((q->front != NULL) ? (q->front)->data : -1));
    printf("Queue Rear : %d", ((q->rear != NULL) ? (q->rear)->data : -1));
    return(0);
}
```

Write a program that reads a sequence of characters and prints them in reading order. Use a queue managed by a linked list.

- queueInizialization
- createNode
- enqueue
- dequeue

- Tree is a set of elements (nodes) on which a "descent" relationship is defined with two properties:
 - There is a single node, called the root, which has no predecessors
 - Every other node has a unique predecessor
- Nodes that have no successors are called leaves
- The tree is divided into levels and the nodes that are neither the root nor a leaf are called intermediate nodes

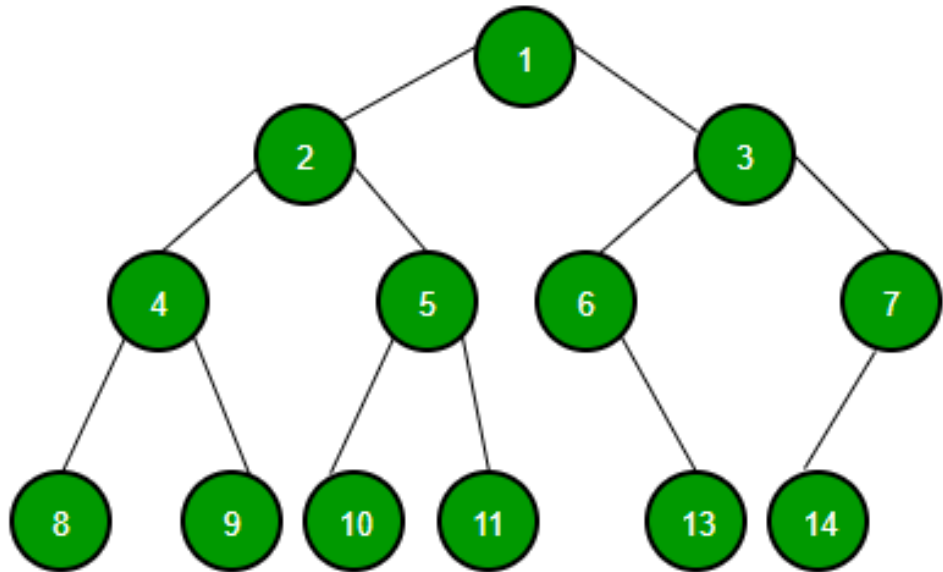


- t is the root node
- v is an intermediate node
- u, x, y, z are leaves
- The tree is divided into three levels and therefore has a depth of 2
- Node v achieves the maximum output degree (equal to 3)

- In the characterization of a tree the following are relevant:
 - Exit degree of a node = number of its direct successors
 - Depth = distance of a node from the root (which by definition has depth 0)
- By "extension", the depth of a tree is given by the max depth of the nodes that compose it
- The set of nodes at the same depth forms a level
- A tree is said to be balanced when at any node the depth of the subtrees differs by at most 1
 - Efficiency: Balanced trees ensure $O(\log n)$ operations for search, insertion, and deletion.
 - Rotations: For AVL and Red-Black trees, rotations help restore balance after operations.
 - Usage: Balanced trees are widely used in databases, compilers, and file systems for efficient data organization and retrieval.

- The same operations performed on a list can be performed on a tree (with some differences)
- On a tree there is not a single terminal node, but multiple leaves
 - The insertion into the queue must be qualified with a selection criterion of which leaf is the target of the operation
 - Insertion at the head (or in an intermediate position) tends to degenerate the tree towards a sequential shape; so, insertions are typically made on leaves

- Binary Tree is defined as a tree data structure where each node has at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary tree is represented by a pointer to the topmost node (commonly known as the “root”) of the tree. If the tree is empty, then the value of the root is NULL. Each node of a Binary Tree contains the following parts:

- 1.Data
- 2.Pointer to left child
- 3.Pointer to right child

- Basic operations on binary tree data structures
 - Inserting an element.
 - Removing an element.
 - Searching for an element.
 - Traversing the tree.

- Auxiliary operations on binary tree data structures
 - Finding the height of the tree
 - Finding the level of a node of the tree
 - Finding the size of the entire tree.

We are interested in a particular type of binary tree data structures.

- A binary search tree is a tree data structure that allows the user to store elements in a sorted manner. It is called a binary tree because each node can have a maximum of two children and is called a search tree because we can search for a number in $O(\log(n))$ time.

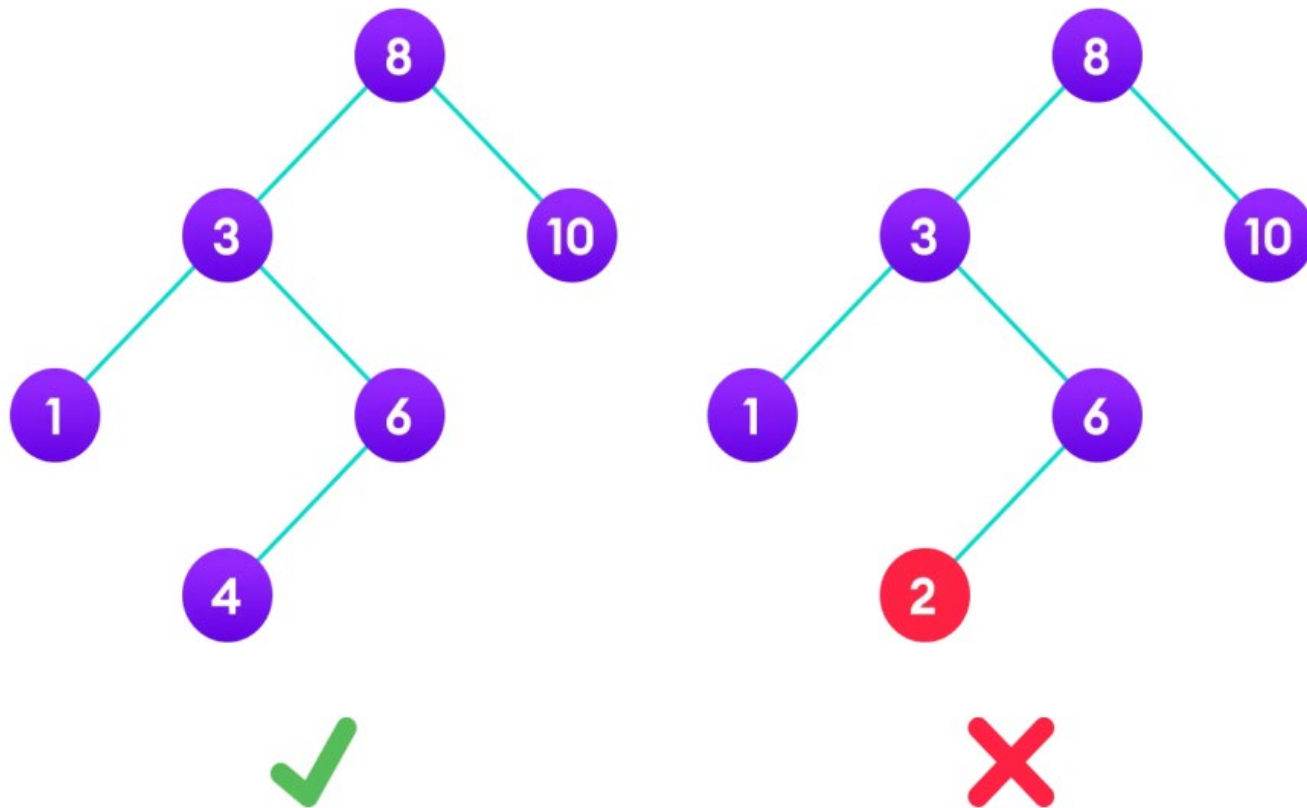
The properties of a binary search tree are:

- All nodes in the left subtree have a value less than the root node
- All nodes in the right subtree have a value more than the root node
- Both subtrees of each node are also binary search trees, i.e. they have the above two properties

Binary search tree data structures



The diagram below demonstrates two binary search trees, one follows all the properties of a binary search tree, while the other violates the rule.



The binary tree on the right isn't a binary search tree because the right subtree of node 3 contains a value smaller than it.

Binary tree data structures



In C, we can represent a binary tree node using structures. Below is an example of a tree node with integer data.

// Structure of each node of the tree

```
struct node {  
    int data;  
    struct node* left_child;  
    struct node* right_child;  
};
```

In search, we have to find a specific element in the data structure. This searching operation becomes simpler in binary search trees because here elements are stored in sorted order.

Algorithm for searching an element in a binary search tree is as follows:

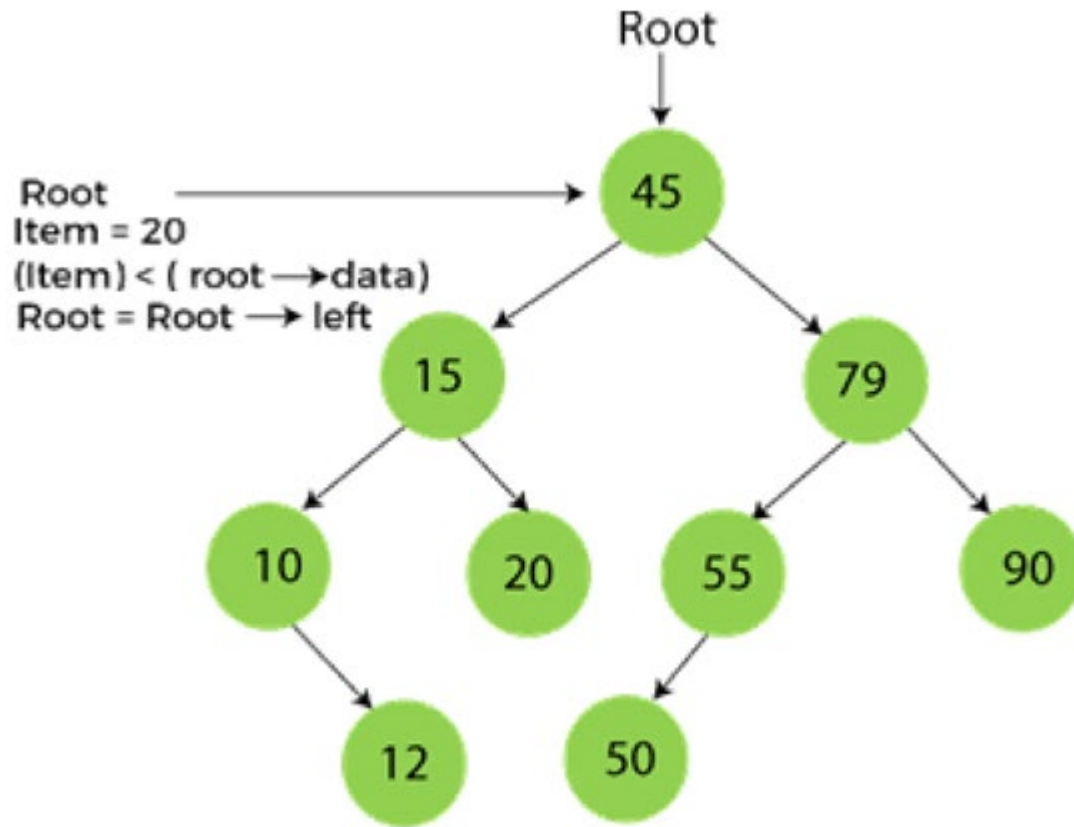
1. Compare the element to be searched with the root node of the tree.
2. If the value of the element to be searched is equal to the value of the root node, return the root node.
3. If the value does not match, check whether the value is less than the root element or not and if it is then traverse the left subtree.
4. If larger than the root element, traverse the right subtree.
5. If the element is not found in the whole tree, return NULL.

Search operation

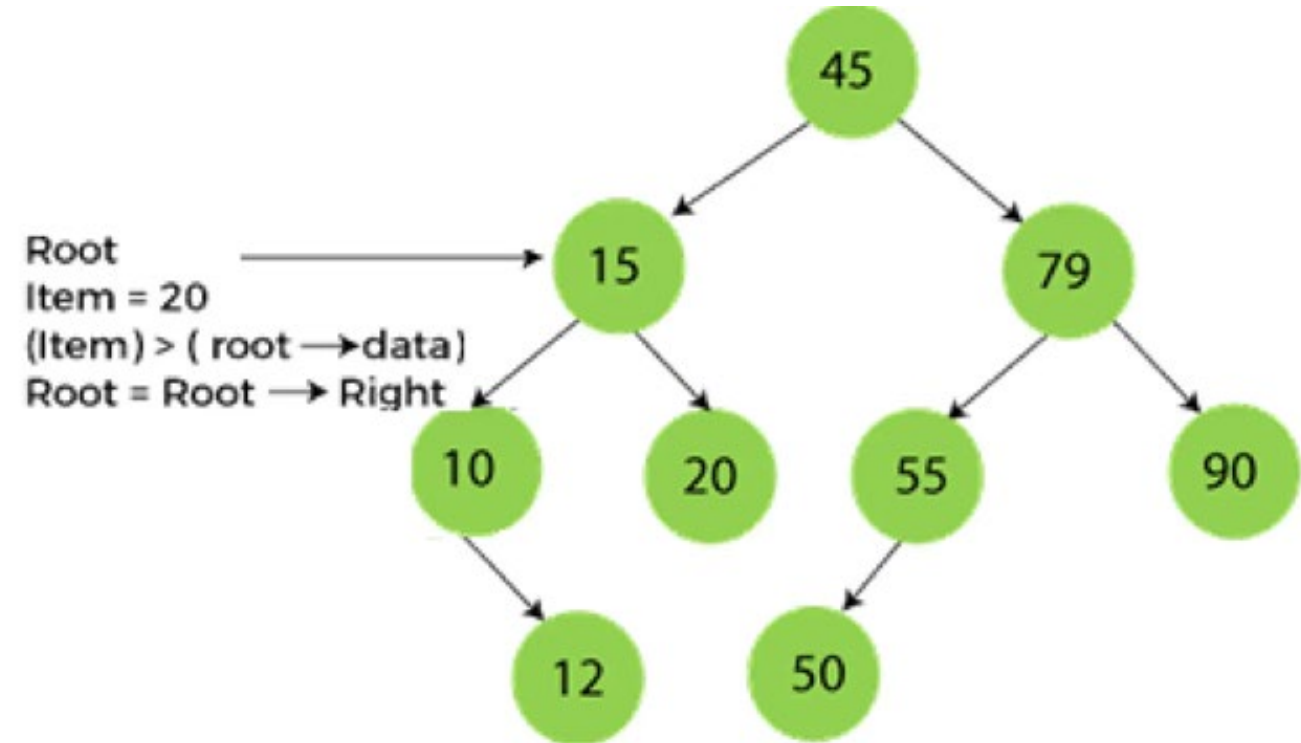


Let's search for an item having a value of 20.

Step 1:



Step 2:



Search operation



Search for an item having a value of 20.

Step 3:

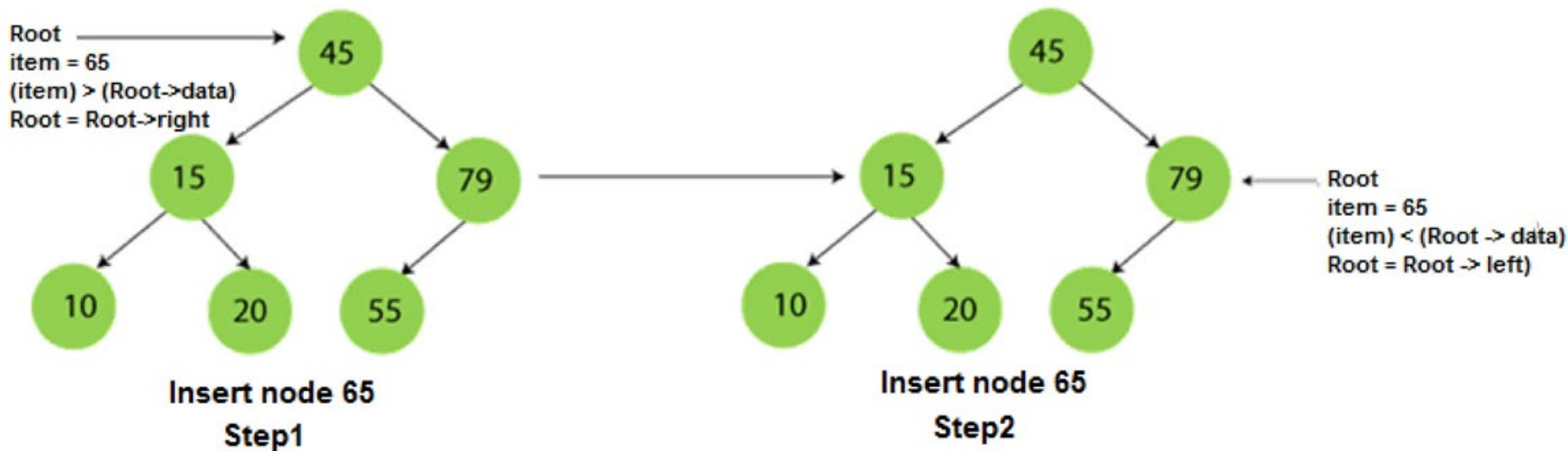


```
// searching operation  
struct node* search(struct node * root, int x) {  
    if (root == NULL || root->data == x) //if root->data is x then the element is found  
        return(root);  
    else if (x > root->data) // x is greater, so we will search the right subtree  
        return(search(root->right_child, x));  
    else //x is smaller than the data, so we will search the left subtree  
        return(search(root->left_child, x));  
}
```


Insert operation

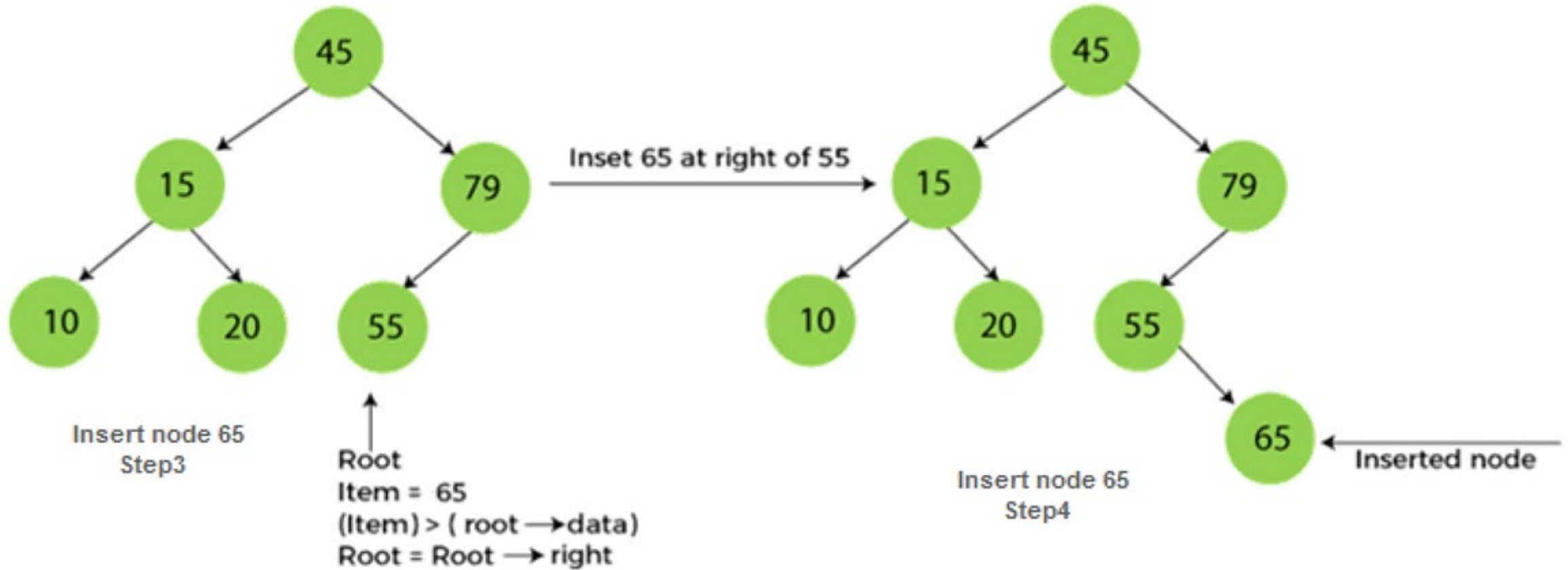


Inserting an element in a binary search tree is always done at the leaf node. To perform insertion in a binary search tree, we start our search operation from the root node, if the element to be inserted is less than the root value or the root node, then we search for an empty location in the left subtree, else, we search for the empty location in the right subtree.



Let's insert an item having a value of 65.

Insert operation



Binary tree data structures



```
/* newNode() allocates a new node with the given data and NULL left and right pointers. */  
struct node* newNode(int data)  
{  
    // Allocate memory for new node  
    struct node* node = (struct node*)malloc(sizeof(struct node));  
  
    // Assign data to this node  
    node->data = data;  
  
    // Initialize left and right children as NULL  
    node->left_child = NULL;  
    node->right_child = NULL;  
    return(node);  
}
```

Insert operation



```
// insertion  
struct node* insert(struct node * root, int x) {  
    //searching for the place to insert  
    if (root == NULL)  
        return newNode(x);  
    else if (x > root->data) // x is greater. Should be inserted to the right  
        root->right_child = insert(root->right_child, x);  
    else // x is smaller and should be inserted to left  
        root->left_child = insert(root->left_child, x);  
    return(root);  
}
```

In the deletion operation, we have to delete a node from the binary search tree in a way that does not violate its properties. Deletion can occur in three possible cases:

1. Node to be deleted is the leaf node
2. Node to be deleted has a single child node
3. The node to be deleted has two children

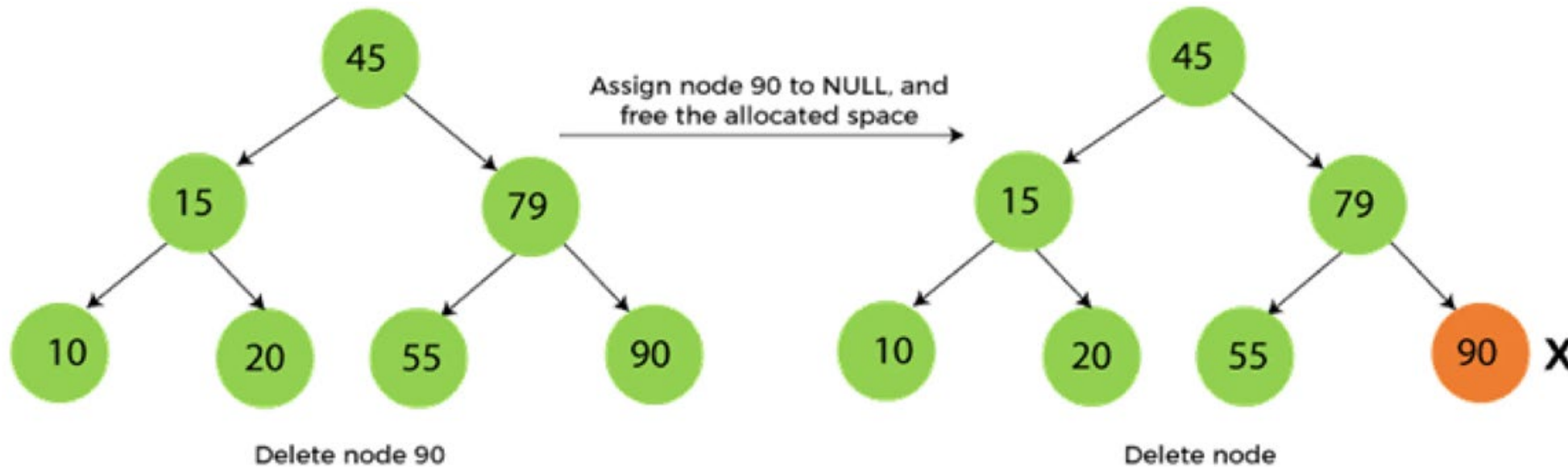
Deletion operation



Node to be deleted is the leaf node.

This is the simplest case of deleting a node in a binary search tree. Here, we will replace the leaf node with NULL and free the allocated space.

Let's try to delete the node having a value of 90.



Node to be deleted has a single child node

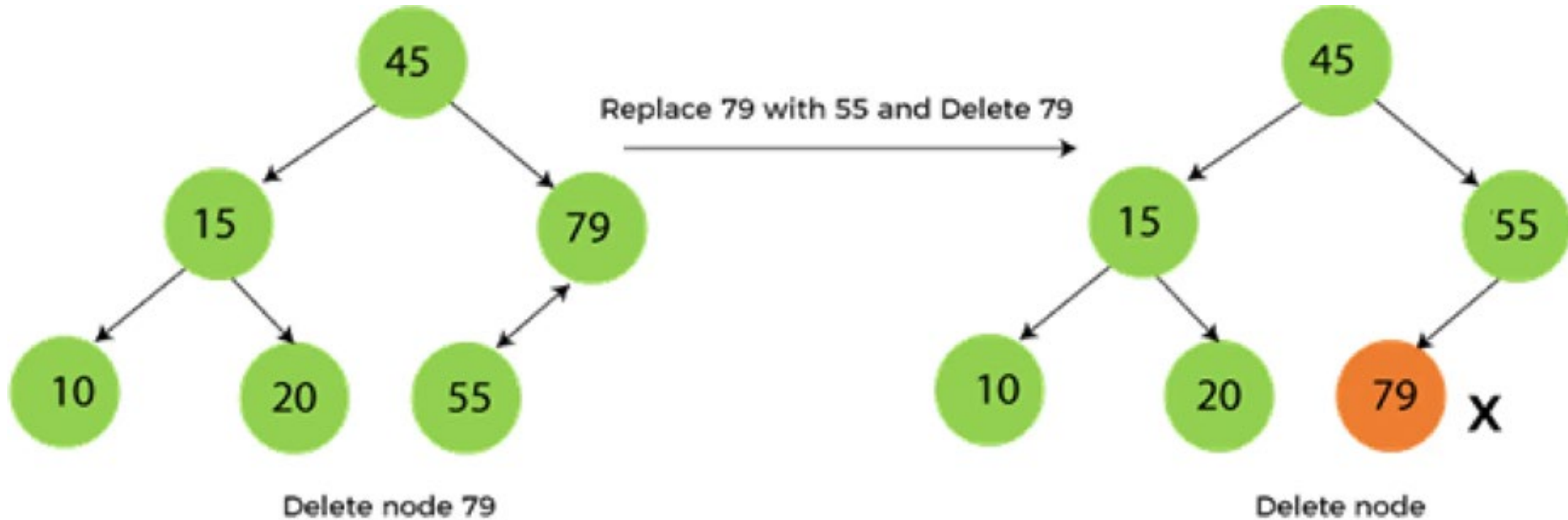
In this case, we will replace the target node with its child and then delete that replaced child node. This means that the child node will now contain the value to be deleted. So, we will just replace the child node with NULL and free up the allocated space.

In the next slide example, we have to delete a node having a value of 79, and this node to be deleted has only one child, so it will be replaced with its child 55.

Deletion operation



Node to be deleted has a single child node



The node to be deleted has two children

This case is complex as compared to the previous two. We follow the below steps to delete the node:

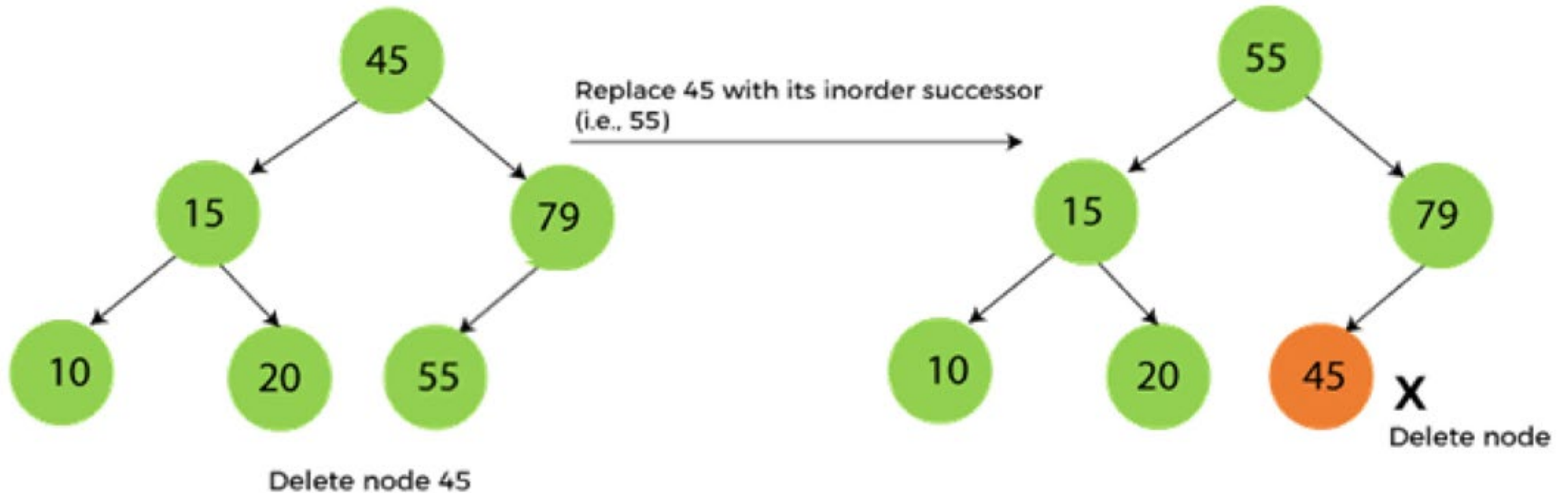
- ❑ We will find the inorder successor of the target node, which has to be deleted
- ❑ Now, replace this node with the successor until the target node is at the leaf
- ❑ Replace the target node with NULL and free up the allocated space

In the following example, we have to delete node 45, which is the root node, so first, it will be replaced with its in-order successor, which is 55. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

Deletion operation



In this example, we have to delete node 45, which is the root node, so first, it will be replaced with its in-order successor, which is 55. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



Deletion operation



// deletion

```
struct node* delete(struct node * root, int x) {  
    //searching for the item to be deleted  
    if (root == NULL)  
        return(NULL);  
    if (x > root->data)  
        root->right_child = delete(root->right_child, x);  
    else if (x < root->data)  
        root->left_child = delete(root->left_child, x);  
    else {  
        //No Child node  
        if (root->left_child == NULL && root->right_child == NULL) {  
            free(root);  
            return(NULL);  
        }  
    }  
}
```

```
//One Child node  
else if (root->left_child == NULL || root->right_child == NULL) {  
    struct node *tmp;  
    if (root->left_child == NULL)  
        tmp = root->right_child;  
    else  
        tmp = root->left_child;  
    free(root);  
    return(tmp);  
}
```

Deletion operation



```
//Two Children  
else {  
    struct node *tmp = find_minimum(root->right_child);  
    root->data = tmp->data;  
    root->right_child = delete(root->right_child, tmp->data);  
}  
}  
return(root);  
}
```

Find inorder successor



```
//function to find the minimum value in a node  
struct node* find_minimum(struct node * root) {  
    if (root == NULL)  
        return(NULL);  
    else if (root->left_child != NULL)  
        // node with minimum value will have no left child  
        return find_minimum(root->left_child);  
    // left most element will be minimum  
    return(root);  
}
```

// Inorder Traversal

```
void inorder(struct node *root) {  
    if (root != NULL) // checking if the root is not null  
    {  
        inorder(root->left_child); // traversing left child  
        printf(" %d ", root->data); // printing data at root  
        inorder(root->right_child); // traversing right child  
    }  
}
```