**EXPERIMENT 1**

**AIM:** Implement sparse matrix using array. Description of program:

1. Read a 2D array from the user.
2. Store it in the sparse matrix form, use array of structures.
3. Print the final array.

**ALGORITHM:**

1. Get a sparse matrix inputted from a user.
2. Loop through the sparse matrix for i,j index
3. Set element = sparse matrix[i][j]
4. If element != 0, store the value of i, j and element in an array of structure

[End of Step 2 loop]

1. Print the array of structure using a loop
2. Exit

A sparse matrix is a matrix which has a high number of 0 elements. This causes a lot of space to be wasted by storing the same value.

So for storing a sparse matrix we use this algorithm where, we traverse through the sparse matrix and find out the nonzero elements. We then input the non zero elements with their row and columns in new array.This method is called 3 tuple notation.

Since we are storing 3x data for non zero elements, it is preferred to convert sparse matrix to 3 tuple notation when the non zero to zero ratio is less than 1/3.

Example:

1. Take a sparse matrix A =
2. Take a struct(row, column, value) array B
3. Loop through the matrix taking i, j index
4. i = 0, j = 0 , element = A[0][0] = 2
5. element != 0, store struct(0, 0, 2) in B
6. I = 0, j = 1, element = A[0][1] = 0
7. Element == 0, skip
8. I = 1, j = 0, element = A[1][0] = 0
9. Element == 0, skip
10. I = 1, j = 1, element = A[1][1] = 1
11. Element != 0, store struct(1, 1, 1) in B
12. Print B
13. Exit

**CODE:**

// Header files

#include <stdio.h>

#include <stdlib.h>

// Struct that will store our non zero data

struct DataElement

{

int value;

int row;

int column;

};

// Prototyping the functions

int \*input2D\_matrix1D(int \*, int \*);

struct DataElement \*shredder(int[], int, int, int \*);

void printCompactMatrix(struct DataElement \*, int);

// Driver code

int main()

{

// Getting a user inputted 2D sparse array that is passed as 1D through the function

int rows, columns;

int \*sparse\_matrix1D = input2D\_matrix1D(&rows, &columns);

// Converting the sparse array into a compact array (array of structure)

int size;

struct DataElement \*compact\_matrix = shredder(sparse\_matrix1D, rows, columns, &size);

// Printing the compact array

printCompactMatrix(compact\_matrix, size);

return 0;

}

struct DataElement \*shredder(int sparse\_matrix1D[], int rows, int columns, int \*compact\_size)

{

// This function converts sparse matrix to compact matrix. Returns struct array and array size.

int(\*sparse\_matrix)[rows] = (int(\*)[rows])sparse\_matrix1D; // Converts 1D array into 2D array using typecasting

// Calculates the number of non-zero elements in the matrix

int filled\_count = 0;

for (int i = 0; i < rows; i++)

for (int j = 0; j < columns; j++)

if (sparse\_matrix[i][j] != 0)

filled\_count++;

// Creates a struct array with length as of the number of non-zero elements

struct DataElement \*compact\_matrix = (struct DataElement \*)malloc(filled\_count \* sizeof(struct DataElement));

// Add structs to the struct array

filled\_count = 0;

for (int i = 0; i < rows; i++)

for (int j = 0; j < columns; j++)

if (sparse\_matrix[i][j] != 0)

{

compact\_matrix[filled\_count].value = sparse\_matrix[i][j];

compact\_matrix[filled\_count].row = i;

compact\_matrix[filled\_count].column = j;

filled\_count++;

}

// Returns the array and its size

\*compact\_size = filled\_count;

return compact\_matrix;

}

int \*input2D\_matrix1D(int \*rows, int \*columns)

{

// This function takes input from the user and creates an array from that

printf("Enter the number of Rows: ");

scanf("%d", rows); // It also returns the value back because pointer

printf("Enter the number of Columns: ");

scanf("%d", columns);

// Creates an array with length as if the 2D matrix was converted to 1D row wise

// This is done since you cannot return a variable length multidimensional array

int \*matrix = (int \*)malloc((\*rows) \* (\*columns) \* sizeof(int));

for (int i = 0; i < \*rows; i++)

for (int j = 0; j < \*columns; j++)

{

printf("Enter the number at position (%d,%d): ", i, j);

scanf("%d", matrix + i \* (\*columns) + j);

}

return matrix;

}

void printCompactMatrix(struct DataElement \*matrix, int size)

{

// Prints the compact matrix using a for loop

printf("\nCompact Matrix \n");

for (int i = 0; i < size; i++)

printf("Element No. %d, Value: %d Row: %d Column: %d \n", i, matrix[i].value, matrix[i].row, matrix[i].column);

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Enter the number of Rows: 3

Enter the number of Columns: 3

Enter the number at position (0,0): 0

Enter the number at position (0,1): 0

Enter the number at position (0,2): 0

Enter the number at position (1,0): 0

Enter the number at position (1,1): 2

Enter the number at position (1,2): 4

Enter the number at position (2,0): 0

Enter the number at position (2,1): 0

Enter the number at position (2,2): 1

Compact Matrix

Element No. 0, Value: 2 Row: 1 Column: 1

Element No. 1, Value: 4 Row: 1 Column: 2

Element No. 2, Value: 1 Row: 2 Column: 2

**EXPERIMENT 2**

**AIM:** Create a linked list with nodes having information about a student and perform:

1. Insert a new node at a specified position.
2. Delete a node with the roll number of a student specified.
3. Reversal of that linked list.

**ALGORITHM:**

**INSERTING NEW NODE AT SPECIFIC POSITION**

1. [check for overflow]

Ptr = (struct node \*) malloc (sizeof(struct node));

If ptr = null

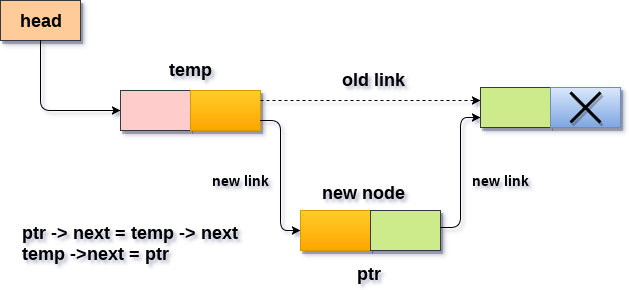
Print overflow   
exit

1. Ptr->data = item;
2. Set temp = head
3. Set i = 0
4. Repeat until i = desired node index
5. Temp = temp → next
6. i = i +1
7. If temp = null

Print "desired node not present"  
 Exit

[ end of loop]

1. Ptr → next = temp → next
2. Temp → next = ptr
3. Exit



**DELETING A SPECIFIC NODE**

1. [check for underflow]

If head = null

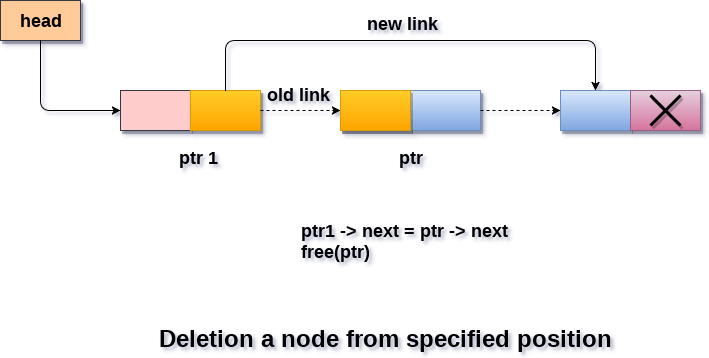
Print underflow   
Exit

1. Set ptr = head
2. Set i = 0
3. Repeat until i = desired node index
4. Set ptr1= ptr
5. ptr = ptr → next
6. i= i +1
7. If ptr = null

Print "desired node not present"  
 Exit

[ end of loop]

1. Ptr1 → next =ptr → next
2. Free(ptr)
3. Exit



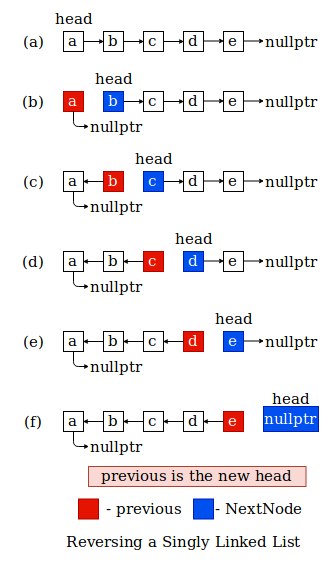
**REVERSAL OF LINKED LIST**

**Iterative method**

1. If head = NULL or head->next = NULL, return head and exit
2. Set Curr = head, sec = head-> next, third = head->next->next
3. Curr->next = NULL
4. While sec != NULL
5. Third = sec->next
6. Sec->next = curr
7. Curr = sec
8. Sec = third

[End of loop]

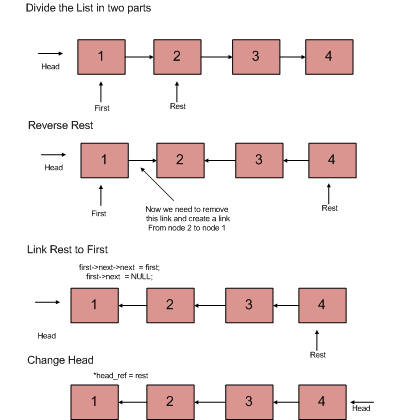
1. Head = curr
2. Exit



**Recursive Method**

**Algorithm reverse\_recursion(node)**

1. If head = NULL or head->next = NULL, return head and exit
2. New\_head = reverse\_recursion(head)
3. Head ->next->next = head
4. Head->next = NULL
5. Return New\_head and Exit



**CODE:**

#include <stdio.h> // Header files

#include <stdlib.h>

// Struct that actually holds the information

struct Student

{

int roll\_no;

char name[30];

char class\_sec[5];

};

// Node struct of the linked list

struct Node

{

struct Student student;

struct Node \*next;

};

// Prototyping the functions

struct Node \*createDummyLinkedList();

struct Node \*reverseLoop(struct Node \*);

struct Node \*reverseRecursion(struct Node \*);

struct Node \*deleteNode(struct Node \*, int);

struct Node \*insertNode(struct Node \*, struct Node \*, int);

void printLL(struct Node \*);

// Driver Code

int main()

{

// Gets a dummy list for us to perform operations on

struct Node \*head = createDummyLinkedList();

// Prints the dummy list

printf("Created Linked list: \n\n");

printLL(head);

// Creating a new node called "node"

struct Node \*node = (struct Node \*)malloc(sizeof(struct Node));

// Check overflow

if (node == NULL)

{

printf("OVERFLOW\n");

return 1;

}

struct Student student = {14463, "Ritesh Singh", "11-B"};

node->student = student;

// Inserting node at second position and printing the new list

head = insertNode(head, node, 2);

printf("\nLinked list after insterting a student data at index 2: \n\n");

printLL(head);

// Deleting the node with roll number as 14461 and printing the new list

deleteNode(head, 14461);

printf("\nLinked list after deleting a student with roll number 14461: \n\n");

printLL(head);

// Reversing the list using a while loop and printing the new list

head = reverseLoop(head);

printf("\nReversed Linked list through loop: \n\n");

printLL(head);

// Reversing the list using recursion and printing the new list

head = reverseRecursion(head);

printf("\nReversed Linked list through recursion: \n\n");

printLL(head);

return 0;

}

struct Node \*reverseLoop(struct Node \*head)

{

// Reverses the loop using while loop and returns the new head

// Checks the list for empty or 1 element

if ((head == NULL) || (head->next == NULL))

return head;

struct Node \*currentnode = head;

struct Node \*secondnode = head->next;

struct Node \*thirdnode = head->next->next;

currentnode->next = NULL;

// This loop reverses the

// Going on reversing the direction of every element as traversing through it

while (secondnode != NULL)

{

thirdnode = secondnode->next;

secondnode->next = currentnode;

currentnode = secondnode;

secondnode = thirdnode;

}

return currentnode; // returning the new head

}

struct Node \*reverseRecursion(struct Node \*head)

{

// Reverses the list using recursion

// Checks if the list is empty or 1D

if ((head == NULL) || (head->next == NULL))

return head;

// It gets the list from n+1 element to the end and points its tail to nth element

// and returns the list from n to end

struct Node \*new\_head = reverseRecursion(head->next);

head->next->next = head;

head->next = NULL;

return new\_head;

}

struct Node \*deleteNode(struct Node \*head, int roll\_no)

{

// Deletes a node with the given roll number

struct Node \*currentnode = head;

// checks for if the list is empty (UNDERFLOW)

if (currentnode == NULL)

return currentnode;

// checks for if the head node needs to be deleted

if (currentnode->student.roll\_no == roll\_no)

{

struct Node \*secondnode = currentnode->next;

free(currentnode);

return secondnode;

}

// Traversing through the list

while (currentnode->next != NULL)

{

// chceking if next node has the roll number

if (currentnode->next->student.roll\_no == roll\_no)

{

// re-pointing the current node to the third node and deleting the next node

struct Node \*secondnode = currentnode->next;

currentnode->next = secondnode->next;

free(secondnode);

break;

}

// continues the traversing

currentnode = currentnode->next;

}

return head;

}

struct Node \*insertNode(struct Node \*head, struct Node \*node, int index)

{

struct Node \*lastnode = NULL;

struct Node \*nextnode = head;

// Loop till the given index elements

for (int i = 0; i < index; i++)

{

// break If reached end

if (nextnode == NULL)

{

break;

}

// continue the loop with increment

lastnode = nextnode;

nextnode = nextnode->next;

}

// if node inserted at beginning

if (lastnode == NULL)

{

node->next = head;

head = node;

}

// inserting node at index

lastnode->next = node;

node->next = nextnode;

// returning head

return head;

}

struct Node \*createDummyLinkedList()

{

// Creating a dummy list to do our operations on

// Allocating memory for our variables

struct Node \*head = (struct Node \*)malloc(sizeof(struct Node));

struct Node \*second = (struct Node \*)malloc(sizeof(struct Node));

struct Node \*third = (struct Node \*)malloc(sizeof(struct Node));

// Check Overflow

if (head == NULL || second == NULL || third == NULL)

{

printf("OVERFLOW\n");

return NULL;

}

// linking the list

head->next = second;

second->next = third;

third->next = NULL;

// Creating information

struct Student student1 = {14460, "Sunil Shetty", "6-B"};

struct Student student2 = {14461, "Rajiv Gandhi", "7-F"};

struct Student student3 = {14462, "Ajay Devgan", "12-D"};

// Assigning information

head->student = student1;

second->student = student2;

third->student = student3;

// Returning head

return head;

}

void printLL(struct Node \*head)

{

// Printing the list by traversing

struct Node \*currentnode = head;

while (currentnode != NULL)

{

printf("Roll No: %d Name: %s Class: %s \n", currentnode->student.roll\_no, currentnode->student.name, currentnode->student.class\_sec);

currentnode = currentnode->next;

}

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Created Linked list:

Roll No: 14460 Name: Sunil Shetty Class: 6-B

Roll No: 14461 Name: Rajiv Gandhi Class: 7-F

Roll No: 14462 Name: Ajay Devgan Class: 12-D

Linked list after insterting a student data at index 2:

Roll No: 14460 Name: Sunil Shetty Class: 6-B

Roll No: 14461 Name: Rajiv Gandhi Class: 7-F

Roll No: 14463 Name: Ritesh Singh Class: 11-B

Roll No: 14462 Name: Ajay Devgan Class: 12-D

Linked list after deleting a student with roll number 14461:

Roll No: 14460 Name: Sunil Shetty Class: 6-B

Roll No: 14463 Name: Ritesh Singh Class: 11-B

Roll No: 14462 Name: Ajay Devgan Class: 12-D

Reversed Linked list through loop:

Roll No: 14462 Name: Ajay Devgan Class: 12-D

Roll No: 14463 Name: Ritesh Singh Class: 11-B

Roll No: 14460 Name: Sunil Shetty Class: 6-B

Reversed Linked list through recursion:

Roll No: 14460 Name: Sunil Shetty Class: 6-B

Roll No: 14463 Name: Ritesh Singh Class: 11-B

Roll No: 14462 Name: Ajay Devgan Class: 12-D

**EXPERIMENT 3**

**AIM:** Create doubly linked list with nodes having information about an employee and perform Insertion at front of doubly linked list and perform deletion at end of that doubly linked list.

**ALGORITHM:**

**INSERTING NEW NODE AT START**

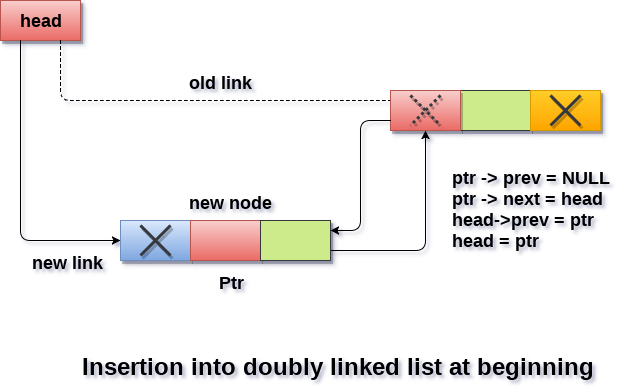
1. [check for overflow]

Ptr = (struct node \*) malloc (sizeof(struct node));

If ptr = null

Print overflow   
exit

1. Ptr->data = item
2. Ptr-> prev = NULL
3. Ptr -> next = head
4. Head -> prev = ptr
5. Head = ptr
6. Exit



**DELETING A NODE AT THE END**

1. [check for underflow]

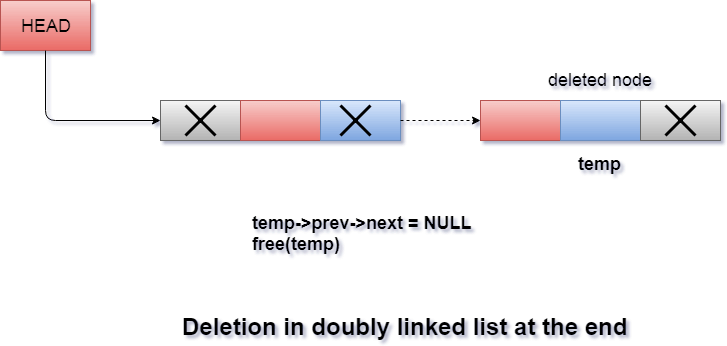
If head = null

Print underflow   
Exit

1. Set temp = head
2. Repeat until temp ->next != NULL // Last node
3. temp = temp→ next

[ end of loop]

1. temp -> prev -> next = NULL
2. Free(temp)
3. Exit



**CODE:**

#include <stdio.h> // Header files

#include <stdlib.h>

// Struct that actually holds the information

struct Employee

{

int employee\_id;

char name[30];

char department[10];

};

// Node struct of the linked list

struct Node

{

struct Employee employee;

struct Node \*next;

struct Node \*prev;

};

// Prototyping the functions

struct Node \*createDummyLinkedList();

struct Node \*insertStartNode(struct Node \*, struct Node \*);

struct Node \*deleteLastNode(struct Node \*);

void printLL(struct Node \*);

// Driver code

int main()

{

// Gets a dummy list for us to perform operations on

struct Node \*head = createDummyLinkedList();

// Prints the dummy list

printf("Created Linked list: \n\n");

printLL(head);

// Creating a new node called "node"

struct Node \*node = (struct Node \*)malloc(sizeof(struct Node));

// Check Overflow

if (node == NULL)

{

printf("OVERFLOW\n");

return 1;

}

struct Employee employee = {14463, "Ritesh Singh", "IT"};

node->employee = employee;

// Inserting a Node at the starting

head = insertStartNode(head, node);

printf("\nLinked list after insterting a employee data at starting: \n\n");

printLL(head);

// Deleting a Node at the ending

head = deleteLastNode(head);

printf("\nLinked list after deleting a employee at the end: \n\n");

printLL(head);

return 0;

}

struct Node \*createDummyLinkedList()

{

// Creating a dummy list to do our operations on

// Allocating memory for our variables

struct Node \*head = (struct Node \*)malloc(sizeof(struct Node));

struct Node \*second = (struct Node \*)malloc(sizeof(struct Node));

struct Node \*third = (struct Node \*)malloc(sizeof(struct Node));

// Check for overflow

if (head == NULL || second == NULL || third == NULL)

{

printf("OVERFLOW\n");

return NULL;

}

// linking the list

head->next = second;

head->prev = NULL;

second->next = third;

second->prev = head;

third->next = NULL;

third->prev = second;

// Creating information

struct Employee emp1 = {14460, "Sunil Shetty", "HR"};

struct Employee emp2 = {14461, "Rajiv Gandhi", "Accounts"};

struct Employee emp3 = {14462, "Ajay Devgan", "Admin"};

// Assigning information

head->employee = emp1;

second->employee = emp2;

third->employee = emp3;

// Returning head

return head;

}

struct Node \*deleteLastNode(struct Node \*head)

{

// This function deletes the node at the ending

struct Node \*currentnode = head;

// checks for if the list is empty (UNDERFLOW)

if (currentnode == NULL)

return head;

// Traverses to the end of the list

while (currentnode->next != NULL)

currentnode = currentnode->next;

// Breaking the prev nodes link to this node

currentnode->prev->next = NULL;

// Freeing this memory

free(currentnode);

return head;

}

struct Node \*insertStartNode(struct Node \*head, struct Node \*node)

{

// This function inserts node at the starting of the list

node->prev = NULL;

node->next = head;

head->prev = node;

return node;

}

void printLL(struct Node \*head)

{

// Printing the list by traversing

struct Node \*currentnode = head;

while (currentnode != NULL)

{

printf("Employee ID: %d Name: %s Department: %s \n", currentnode->employee.employee\_id, currentnode->employee.name, currentnode->employee.department);

currentnode = currentnode->next;

}

}

**OUPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Created Linked list:

Employee ID: 14460 Name: Sunil Shetty Department: HR

Employee ID: 14461 Name: Rajiv Gandhi Department: Accounts

Employee ID: 14462 Name: Ajay Devgan Department: Admin

Linked list after insterting a employee data at starting:

Employee ID: 14463 Name: Ritesh Singh Department: IT

Employee ID: 14460 Name: Sunil Shetty Department: HR

Employee ID: 14461 Name: Rajiv Gandhi Department: Accounts

Employee ID: 14462 Name: Ajay Devgan Department: Admin

Linked list after deleting a employee at the end:

Employee ID: 14463 Name: Ritesh Singh Department: IT

Employee ID: 14460 Name: Sunil Shetty Department: HR

Employee ID: 14461 Name: Rajiv Gandhi Department: Accounts

**EXPERIMENT 4**

**AIM:** Create circular linked list having information about a college and perform Insertion at front perform Deletion at end.

**ALGORITHM:**

**INSERTING NEW NODE AT START**

1. [check for overflow]

Ptr = (struct node \*) malloc (sizeof(struct node));

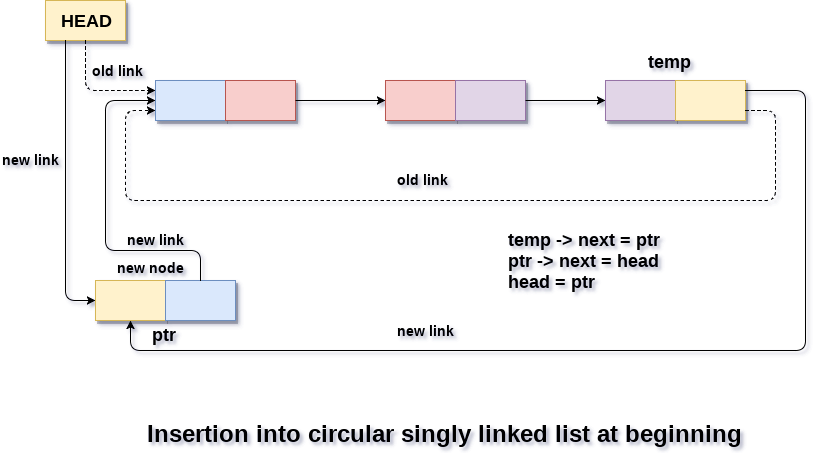
If ptr = null

Print overflow   
exit

1. Ptr->data = item
2. Set temp = head
3. Repeat until temp ->next != NULL // Last node
4. temp = temp→ next

[ end of loop]

1. Ptr -> next = head
2. temp -> next = ptr
3. Head = ptr
4. Exit



**DELETION AT THE END:**

1. [check for underflow]

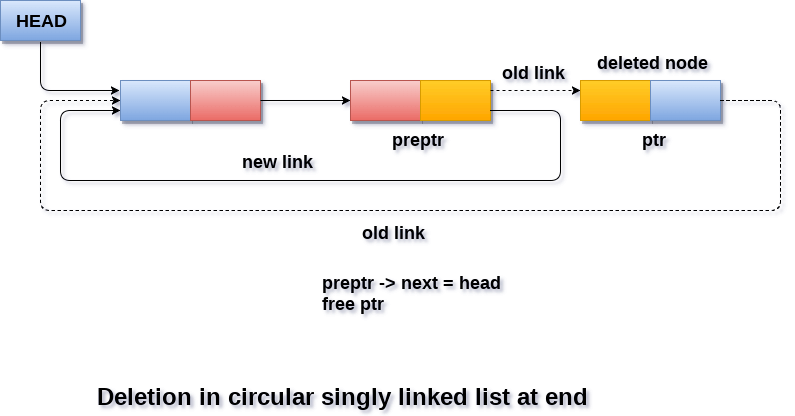
If head = null

Print underflow   
Exit

1. Set preptr = head
2. Repeat until preptr ->next ->next != head //second last node
3. preptr = preptr ->next

[ end of loop]

1. Free(preptr->next) // Last Node
2. Preptr->next = head
3. Exit



**CODE:**

#include <stdio.h> // Header files

#include <stdlib.h>

// Struct that actually holds the information

struct College

{

int college\_id;

char name[30];

};

// Node struct of the linked list

struct Node

{

struct College college;

struct Node \*next;

};

// Prototyping the functions

struct Node \*createDummyLinkedList();

struct Node \*deleteLastNode(struct Node \*);

struct Node \*insertStartNode(struct Node \*, struct Node \*);

void printLL(struct Node \*);

int main()

{

// Gets a dummy list for us to perform operations on

struct Node \*head = createDummyLinkedList();

// Prints the dummy list

printf("Created Linked list: \n\n");

printLL(head);

// Creating a new node called "node"

struct Node \*node = (struct Node \*)malloc(sizeof(struct Node));

// Check Overflow

if (node == NULL)

{

printf("OVERFLOW\n");

return 1;

}

struct College college = {14463, "IITD"};

node->college = college;

// Inserting a Node at the starting

head = insertStartNode(head, node);

printf("\nLinked list after insterting a college data at starting: \n\n");

printLL(head);

// Deleting a Node at the ending

head = deleteLastNode(head);

printf("\nLinked list after deleting a college at the end: \n\n");

printLL(head);

return 0;

}

struct Node \*deleteLastNode(struct Node \*head)

{

// This function deletes the node at the ending

struct Node \*temp = head;

// checks for if the list is empty (UNDERFLOW)

if (temp == NULL)

return head;

// Traverses to the second last of the list

while (1)

{

if (temp->next->next == head)

{

free(temp->next);

temp->next = head;

break;

}

temp = temp->next;

}

return head;

}

struct Node \*insertStartNode(struct Node \*head, struct Node \*node)

{

// This function inserts node at the starting of the list

node->next = head;

struct Node \*temp = head;

// Traverses to the end of the list to change the last link

while (1)

{

if (temp->next == head)

{

temp->next = node;

break;

}

temp = temp->next;

}

return node;

}

struct Node \*createDummyLinkedList()

{

// Creating a dummy list to do our operations on

// Allocating memory for our variables

struct Node \*head = (struct Node \*)malloc(sizeof(struct Node));

struct Node \*second = (struct Node \*)malloc(sizeof(struct Node));

struct Node \*third = (struct Node \*)malloc(sizeof(struct Node));

// Check Overflow

if (head == NULL || second == NULL || third == NULL)

{

printf("OVERFLOW\n");

return NULL;

}

// linking the list

head->next = second;

second->next = third;

third->next = head;

// Creating information

struct College college1 = {14460, "MSIT"};

struct College college2 = {14461, "NSUT"};

struct College college3 = {14462, "DTU"};

// Assigning information

head->college = college1;

second->college = college2;

third->college = college3;

// Returning head

return head;

}

void printLL(struct Node \*head)

{

// Printing the list by traversing

struct Node \*currentnode = head;

if (head == NULL)

return;

do

{

printf("College ID: %d Name: %s \n", currentnode->college.college\_id, currentnode->college.name);

currentnode = currentnode->next;

} while (currentnode != head);

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Created Linked list:

College ID: 14460 Name: MSIT

College ID: 14461 Name: NSUT

College ID: 14462 Name: DTU

Linked list after insterting a college data at starting:

College ID: 14463 Name: IITD

College ID: 14460 Name: MSIT

College ID: 14461 Name: NSUT

College ID: 14462 Name: DTU

Linked list after deleting a college at the end:

College ID: 14463 Name: IITD

College ID: 14460 Name: MSIT

College ID: 14461 Name: NSUT

**EXPERIMENT 5**

**AIM:** Implement two stacks in a using single array.

**ALGORITHM:**

Two stacks in an array by Starting from endpoints

Starting States = top1 = -1, top2 = array size

**PUSH1:**

1. If top2 - top 1 <= 1 // Space not available in array

Print Overflow

exit

1. Top1 = top1 +1
2. Array[top1] = item
3. Exit

**POP1:**

1. If top1 <= -1

Print Underflow

exit

1. Item = array[top1]
2. Top1 = top1 -1
3. exit

**PUSH2:**

1. If top2 - top 1 <= 1 // Space not available in array

Print Overflow

exit

1. Top2 = top2 - 1
2. Array[top2] = item
3. exit

**POP2:**

1. If top2 >= array\_size

Print Underflow

exit

1. Item = array[top2]
2. Top2 = top2 +1
3. exit

Implemented 2 stacks in 1 array by dividing the array into 2 parts.

Pop function removes the top element from the stack.

Push function inserts the top element into the stack.

Pop1() and push1() work on the left half of the array.

Pop2() and push2() work on the right half of the array.

Top1 and top2 are taken as the end points of the array.

**EXAMPLE:**

Array[3], top1 = -1, top2 = 3

* Push1 (6)

1. Top2-top1 = 4 > 1 // Space available
2. Top1 = -1 +1 = 0
3. Array[0] = item = 6
4. Exit

Array = [6,,]

* Push2(10)

1. Top2 – top1 = 3 > 1 // Space avaiable
2. Top2 = 3-1 = 2
3. Array[2] = item = 10
4. Exit

Array = [6,,10]

* Push1 (4)

1. Top2-top1 = 2 > 1 // Space available
2. Top1 = 0 +1 = 1
3. Array[1] = item = 4
4. Exit

Array = [6,4,10]

* Push2(15)

1. Top2 – top1 = 1 == 1 // Space not avaiable
2. Print Overflow
3. Exit

Array = [6,4,10]

* Pop2()

1. Top2 = 2 < array\_size // not underflow
2. Item = array[2]
3. Top2 = 2+1 = 3
4. Exit

Item = 10, Array = [6,4,10] // 10 can be overwritten

* Pop2()

1. Top2 = 3 == array\_size
2. Print Underflow
3. Exit

* Push1(5)

1. Top2-top1 = 2 > 1 // Space available
2. Top1 = 1 +1 = 2
3. Array[2] = item = 5
4. Exit

Array = [6,4,5]

**CODE:**

#include <stdio.h> // Header files

// Defining stack variables

int array[100];

int top1 = -1, bottom2 = 100, size = 100;

// Top 1 is top of stack 1, Bottom 2 is top of the inverted stack 2

// Prototyping the functions

void push1(int);

void push2(int);

int pop1();

int pop2();

void print\_stack\_1();

void print\_stack\_2();

// Driver Code

int main()

{

// Printing the stacks at the start of program

printf("\nStacks at the start.\n");

print\_stack\_1();

print\_stack\_2();

// Pushing 2 elements into stack 1 and printing

push1(1);

push1(2);

printf("\nStack 1 after pushing 2 elements.\n");

print\_stack\_1();

// Pushing 2 elements into stack 2 and printing

push2(3);

push2(4);

printf("\nStack 2 after pushing 2 elements.\n");

print\_stack\_2();

// Popping elements from stack 1

printf("\nPopping element from stack 1: ");

printf("%d \n", pop1());

printf("Popping element from stack 1: ");

printf("%d \n", pop1());

printf("Popping element from stack 1: ");

printf("%d \n", pop1());

printf("\nStack 1 after popping.\n");

print\_stack\_1();

// Popping elements from stack 2

printf("\nPopping element from stack 2: ");

printf("%d \n", pop2());

printf("Popping element from stack 2: ");

printf("%d \n", pop2());

printf("Popping element from stack 2: ");

printf("%d \n", pop2());

printf("\nStack 2 after popping.\n");

print\_stack\_2();

return 0;

}

void push1(int element)

{

// Function to push element into stack 1

if (bottom2 - top1 > 1) // Checks if space available

{

top1++;

array[top1] = element;

}

else

printf("Stack Overflow\n");

}

void push2(int element)

{

// Function to push element into stack 2

if (bottom2 - top1 > 1) // Checks if space available

{

bottom2--;

array[bottom2] = element;

}

else

printf("Stack Overflow\n");

}

int pop1()

{

// Function to pop and return element from stack 1

// Checks if elements present

// returns -1 if Underflow

if (top1 > -1)

{

int element = array[top1];

top1--;

return element;

}

else

{

printf("Stack Underflow\n");

return -1;

}

}

int pop2()

{

// Function to pop and return element from stack 2

// Checks if elements present

// returns -1 if Underflow

if (bottom2 < size)

{

int element = array[bottom2];

bottom2++;

return element;

}

else

{

printf("Stack Underflow\n");

return -1;

}

}

void print\_stack\_1()

{

// Function to print elements of stack 1

printf("Stack1: \n");

for (int i = top1; i >= 0; i--)

{

printf("%d\n", array[i]);

}

}

void print\_stack\_2()

{

// Function to print elements of stack 2

printf("Stack2: \n");

for (int i = bottom2; i <= size - 1; i++)

{

printf("%d\n", array[i]);

}

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Stacks at the start.

Stack1:

Stack2:

Stack 1 after pushing 2 elements.

Stack1:

2

1

Stack 2 after pushing 2 elements.

Stack2:

4

3

Popping element from stack 1: 2

Popping element from stack 1: 1

Popping element from stack 1: Stack Underflow

-1

Stack 1 after popping.

Stack1:

Popping element from stack 2: 4

Popping element from stack 2: 3

Popping element from stack 2: Stack Underflow

-1

Stack 2 after popping.

Stack2:

**EXPERIMENT 6**

**AIM:** Create a stack and perform Push, Pop, Peek and Traverse operations on the stack using Linked list.

**ALGORITHM:**

**PUSH:**

1. [check for overflow]

Ptr = (struct node \*) malloc (sizeof(struct node));

If ptr = null

Print overflow   
exit

1. Ptr->data = item
2. Ptr -> next = top
3. top = ptr
4. Exit

**POP:**

1. [check for underflow]

If top = null

Print underflow   
Exit

1. Set ptr = top
2. Top = ptr ->next
3. Item = ptr->data
4. Free(ptr)
5. Exit

**PEEK:**

1. [check for underflow]

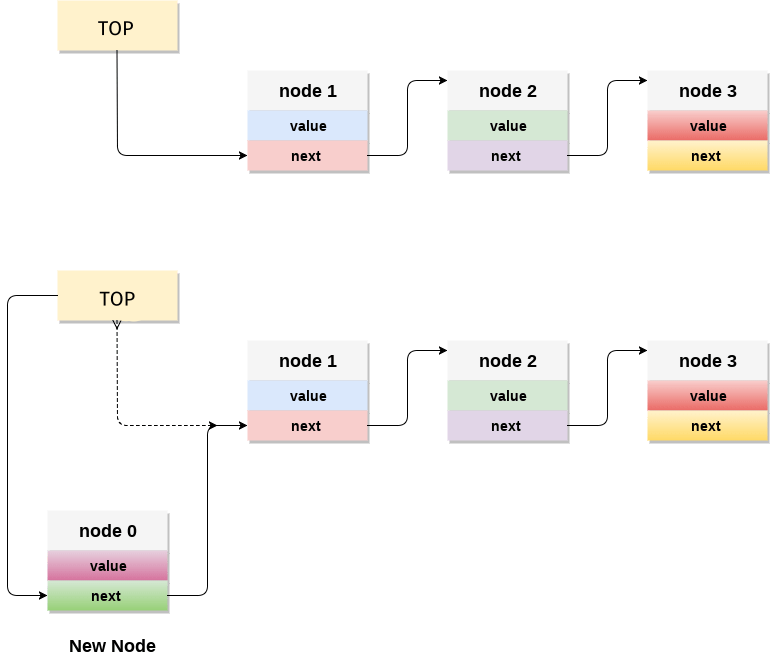
If top = null

Print underflow and exit

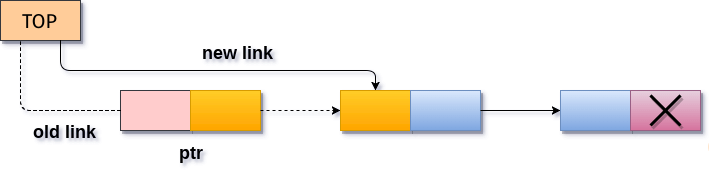
1. Else item = top -> data
2. Exit

**TRAVERSE:**

1. Ptr = top
2. While ptr ! = Null
3. Print ptr->data
4. Ptr = ptr->next
5. exit

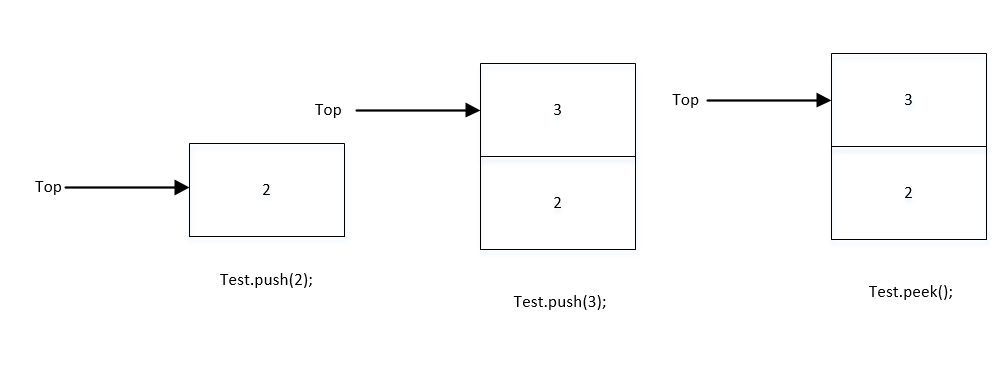


**PUSHING TO THE STACK**



**POPPING FROM THE STACK**

**PUSH, POP AND PEEK ON THE STACK**



**CODE:**

#include <stdio.h> // Header files

#include <stdlib.h>

// Node struct of the linked list

struct Node

{

int data;

struct Node \*next;

};

// Top of the stack

struct Node \*top = NULL;

// Prototyping the functions

void push(int);

int pop();

int peek();

void display();

// Driver code

int main()

{

// Printing the stack at the start of program

printf("\nStack at the start:\n");

display();

// Pushing 2 elements into stack and printing

push(1);

push(2);

printf("\nStack after pushing 2 elements:\n");

display();

// Peeking stack and printing the stack to see changes due to peeking

printf("\nPeeking stack: ");

printf("%d \n", peek());

printf("\nStack after peeking:\n");

display();

// Popping stack and printing the stack to see changes due to popping

printf("\nPopping element from stack: ");

printf("%d \n", pop());

printf("\nStack after popping:\n");

display();

printf("\nPeeking stack: ");

printf("%d \n", peek());

printf("Popping element from stack: ");

printf("%d \n", pop());

printf("Peeking stack: ");

printf("%d \n", peek());

printf("Popping element from stack: ");

printf("%d \n", pop());

printf("\nStack after popping:\n");

display();

return 0;

}

void push(int element)

{

// Function to push element into stack

// Create new node

struct Node \*newNode = (struct Node \*)malloc(sizeof(struct Node));

if (newNode == NULL)// Check Overflow

{

printf("Overflow\n");

return;

}

// Set data to the node

newNode->data = element;

newNode->next = NULL;

// replacing the top

newNode->next = top;

top = newNode;

}

int pop()

{

// Function to pop and return element from stack

// Checks if elements present

// Returns -1 if underflow

if (top == NULL)

{

printf("Underflow\n");

return -1;

}

else

{

struct Node \*temp = top;

int element = top->data; // to store data of top node

top = top->next;

free(temp); // deleting the node

return element;

}

}

int peek()

{

// Function to peek and return element from stack 1

// Checks if elements present

if (top == NULL)

{

printf("Underflow\n");

return -1;

}

else

return top->data;

}

void display()

{

// Printing the list by traversing

struct Node \*currentnode = top;

while (currentnode != NULL)

{

printf("%d\n", currentnode->data);

currentnode = currentnode->next;

}

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Stack at the start:

Stack after pushing 2 elements:

2

1

Peeking stack: 2

Stack after peeking:

2

1

Popping element from stack: 2

Stack after popping:

1

Peeking stack: 1

Popping element from stack: 1

Peeking stack: Underflow

-1

Popping element from stack: Underflow

-1

Stack after popping:

**EXPERIMENT 7**

**AIM:** Create a stack and perform Push, Pop, Peek and Traverse operations on the stack using Linked list.

**ALGORITHM:**

**ENQUEUE:**

1. [check for overflow]

Ptr = (struct node \*) malloc (sizeof(struct node));

If ptr = null

Print overflow   
exit

1. Ptr->data = item
2. If rear = NULL
3. rear = front = ptr

[End of If]

1. Else rear -> next = ptr
2. rear = ptr

[End of else]

1. Exit

**DEQUEUE:**

1. [check for underflow]

If front = null

Print underflow   
Exit

1. Set ptr = front
2. front = ptr ->next
3. Item = ptr->data
4. Free(ptr)
5. If front = NULL
6. rear = NULL

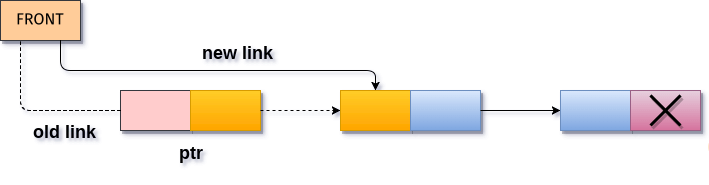
[End of if]

1. Exit

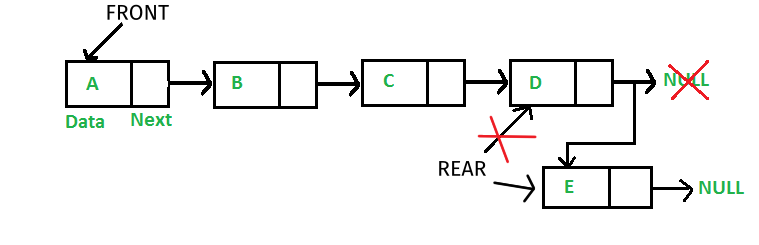
**TRAVERSE:**

1. Ptr = bottom
2. While ptr ! = Null
3. Print ptr->data
4. Ptr = ptr->next
5. exit

**DEQUEUE**



**ENQUEUE**



**CODE:**

#include <stdio.h> // Header files

#include <stdlib.h>

// Node struct of the linked list

struct Node

{

int data;

struct Node \*next;

};

// rear and front of the queue, entry from rear and exit from front

struct Node \*front = NULL;

struct Node \*rear = NULL;

// Prototyping the functions

void enqueue(int);

int dequeue();

void display();

// Driver code

int main()

{

// Printing the queue at the start of program

printf("\nQueue at the start:\n");

display();

// enqueuing 2 elements into queue and printing

enqueue(1);

enqueue(2);

printf("\nQueue after enqueuing 2 elements:\n");

display();

// dequeuing 1 element from queue and printing

printf("\ndequeuing element from Queue: ");

printf("%d \n", dequeue());

printf("\nQueue after dequeuing:\n");

display();

// dequeuing all the elements from queue and printing

printf("dequeuing element from Queue: ");

printf("%d \n", dequeue());

printf("dequeuing element from Queue: ");

printf("%d \n", dequeue());

printf("\nQueue after dequeuing:\n");

display();

return 0;

}

void enqueue(int element)

{

// Function to enqueue element into queue

// Create new node

struct Node \*newNode = (struct Node \*)malloc(sizeof(struct Node));

if (newNode == NULL) // Check Overflow

{

printf("Overflow\n");

return;

}

// Set data to the node

newNode->data = element;

newNode->next = NULL;

// replacing the rear

if (front == NULL)

front = rear = newNode;

else

{

rear->next = newNode;

rear = newNode;

}

}

int dequeue()

{

// Function to dequeue and return element from queue

// Checks if elements present

// Returns -1 if underflow

if (front == NULL)

{

printf("Underflow\n");

return -1;

}

else

{

struct Node \*temp = front;

int element = front->data; // to store data of rear node

front = front->next;

if (front == NULL)

rear = front;

free(temp); // deleting the node

return element;

}

}

void display()

{

// Printing the list by traversing

struct Node \*currentnode = front;

while (currentnode != NULL)

{

printf("%d ,", currentnode->data);

currentnode = currentnode->next;

}

printf("\n");

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Queue at the start:

Queue after enqueuing 2 elements:

1 ,2 ,

dequeuing element from Queue: 1

Queue after dequeuing:

2 ,

dequeuing element from Queue: 2

dequeuing element from Queue: Underflow

-1

Queue after dequeuing:

**EXPERIMENT 8**

**AIM:** Implement Experiment-2 using liked list.

**ALGORITHM:**

**INSERTING AT SPECIFIC POSITION:**

N is the count of elements in the list

Starting state, N = 0

1. If N>= array\_size,

Print Overflow and exit

1. For i = N-1 till i = index to be inserted at
2. Swap array[i] and array[i+1]
3. I = i - 1

[ End of loop]

1. N = N+1
2. Array[index] = item
3. exit

**DELETING AT SPECIFIC POSITION**

1. If N<= 0,

Print Underflow and exit

1. For i = index till i < N -1
2. Swap array[i] and array[i+1]
3. I = i + 1

[ End of loop]

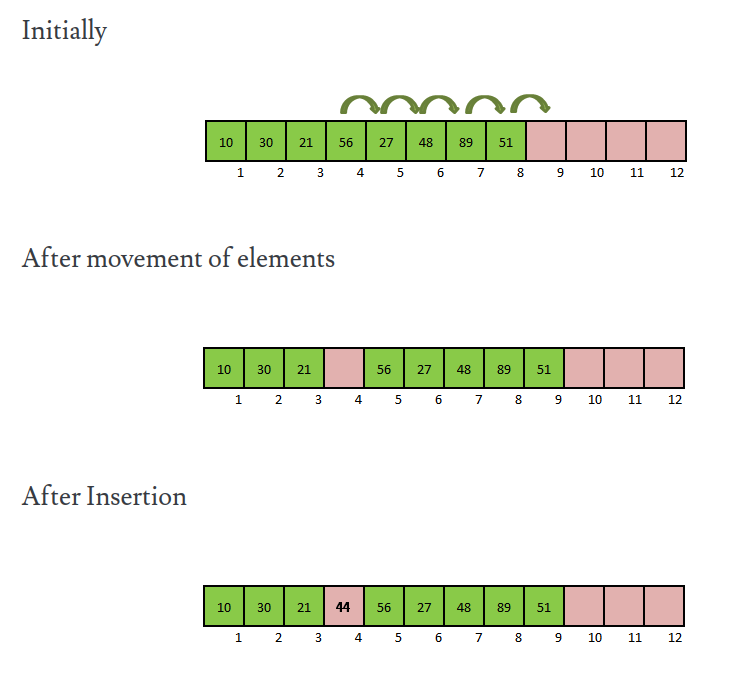
1. N = N-1
2. exit

**REVERSAL OF THE LIST**

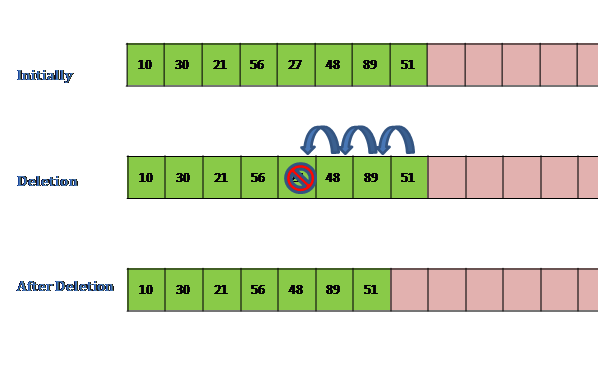
1. For i = 0 till i < N/2
2. I = i + 1
3. Swap array[i] and array[n - 1 - i]

[ End of loop]

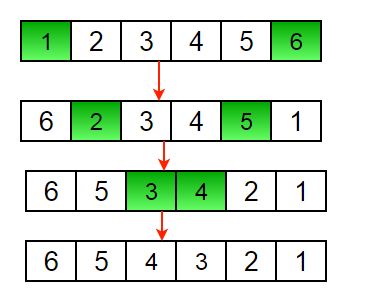
1. Exit



**INSERTION IN ARRAY**



**DELETION IN ARRAYS**



**REVERSAL IN ARRAY**

**CODE:**

#include <stdio.h> // Header files

#include <stdlib.h>

// Struct that actually holds the information

struct Student

{

int roll\_no;

char const \*name;

char const \*class\_sec;

};

// Macro for swapping 2 numbers

#define swap(a, b) \

{ \

struct Student \*temp = a; \

a = b; \

b = temp; \

}

// Prototyping the functions

struct Student \*\*createDummyList(int, int \*);

void reverse(struct Student \*\*, int);

void delete(struct Student \*\*, int, int \*);

void insert(struct Student \*\*, struct Student \*, int, int, int \*);

void printList(struct Student \*\*, int, int);

int main()

{

// Keeping track of elements and size of the array

int no\_of\_elements = 0, size = 10;

// Gets a dummy list for us to perform operations on

struct Student \*\*list = createDummyList(size, &no\_of\_elements);

// Prints the dummy list

printf("Created list: \n\n");

printList(list, size, no\_of\_elements);

// creating a new student

struct Student \*student = (struct Student \*)malloc(sizeof(struct Student));

// Check overflow

if (student == NULL)

{

printf("OVERFLOW\n");

return 1;

}

// assigning values to the student node

student->roll\_no = 14463;

student->name = "Ritesh Singh";

student->class\_sec = "11-B";

// Inserting node at second position and printing the new list

insert(list, student, 2, size, &no\_of\_elements);

printf("\nList after insterting a student data at index 2: \n\n");

printList(list, size, no\_of\_elements);

// Deleting the node with roll number as 14461 and printing the new list

delete(list, 14461, &no\_of\_elements);

printf("\nList after deleting a student with roll number 14461: \n\n");

printList(list, size, no\_of\_elements);

// Reversing the list and printing the new list

reverse(list, no\_of\_elements);

printf("\nReversed list through loop: \n\n");

printList(list, size, no\_of\_elements);

return 0;

}

void reverse(struct Student \*\*list, int n)

{

// Reversing the array

for (int i = 0; i < n / 2; i++)

swap(list[i], list[n - 1 - i]);

}

void insert(struct Student \*\*list, struct Student \*student, int index, int size, int \*n)

{

// Inserting element into array

// If no space available

if (\*n >= size)

{

printf("List Overflow");

return;

}

// Copying the elements around and create space

for (int i = \*n - 1; i >= index; i--)

swap(list[i], list[i + 1]);

// increase the count of no of elements in array

\*n = \*n + 1;

list[index] = student;

}

void delete(struct Student \*\*list, int roll\_no, int \*n)

{

// Removing element from array

// If no element in list

if (\*n <= 0)

{

printf("List Underflow");

return;

}

int index;

// Finding the index of the roll number

for (int i = 0; i < \*n; i++)

if (list[i]->roll\_no == roll\_no)

{

index = i;

break;

}

// Copying the elements over the index

for (int i = index; i < \*n - 1; i++)

swap(list[i], list[i + 1]);

// Decreasing the count

\*n = \*n - 1;

}

struct Student \*\*createDummyList(int size, int \*n)

{

// Creating a dummy list to do our operations on

// Allocating memory for our variables

struct Student \*student1 = (struct Student \*)malloc(sizeof(struct Student));

struct Student \*student2 = (struct Student \*)malloc(sizeof(struct Student));

struct Student \*student3 = (struct Student \*)malloc(sizeof(struct Student));

// Check Overflow

if (student1 == NULL || student2 == NULL || student3 == NULL)

{

printf("OVERFLOW\n");

return NULL;

}

char \*name;

// Creating information

student1->roll\_no = 14460;

student1->name = "Sunil Shetty";

student1->class\_sec = "6-B";

student2->roll\_no = 14461;

student2->name = "Rajiv Gandhi";

student2->class\_sec = "7-F";

student3->roll\_no = 14462;

student3->name = "Ajay Devgan";

student3->class\_sec = "12-D";

struct Student \*\*list = (struct Student \*\*)malloc(size \* sizeof(struct Student \*));

list[0] = student1;

list[1] = student2;

list[2] = student3;

\*n = 3;

return list;

}

void printList(struct Student \*\*list, int size, int n)

{

// if count is greater than size for some reason print till the size only

if (n > size)

n = size;

for (int i = 0; i < n; i++)

printf("Roll No: %d Name: %s Class: %s \n", list[i]->roll\_no, list[i]->name, list[i]->class\_sec);

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Created list:

Roll No: 14460 Name: Sunil Shetty Class: 6-B

Roll No: 14461 Name: Rajiv Gandhi Class: 7-F

Roll No: 14462 Name: Ajay Devgan Class: 12-D

List after insterting a student data at index 2:

Roll No: 14460 Name: Sunil Shetty Class: 6-B

Roll No: 14461 Name: Rajiv Gandhi Class: 7-F

Roll No: 14463 Name: Ritesh Singh Class: 11-B

Roll No: 14462 Name: Ajay Devgan Class: 12-D

List after deleting a student with roll number 14461:

Roll No: 14460 Name: Sunil Shetty Class: 6-B

Roll No: 14463 Name: Ritesh Singh Class: 11-B

Roll No: 14462 Name: Ajay Devgan Class: 12-D

Reversed list through loop:

Roll No: 14462 Name: Ajay Devgan Class: 12-D

Roll No: 14463 Name: Ritesh Singh Class: 11-B

Roll No: 14460 Name: Sunil Shetty Class: 6-B

**EXPERIMENT 9**

**AIM:** Create a Binary Tree and perform Tree traversals (Preorder, Postorder, Inorder) using the concept of recursion.

**ALGORITHM:**

**TRAVERSALS:**

**Algorithm Inorder-traversal(root)**

1. If root ≠ NULL then // Not empty tree
2. Inorder-traversal(root ->left)
3. Print root ->data
4. Inorder-traversal(root ->right)

[ end of if ]

1. Exit

**Algorithm Preorder-traversal(root)**

1. If root ≠ NULL then
2. Print root ->data
3. Preorder-traversal(root ->left)
4. Preorder-traversal(root ->right)

[ end of if ]

1. Exit

**Algorithm Postorder-traversal(root)**

1. If root ≠ NULL then
2. Postorder-traversal(root ->left)
3. Postorder-traversal(root ->right)
4. Print root ->data

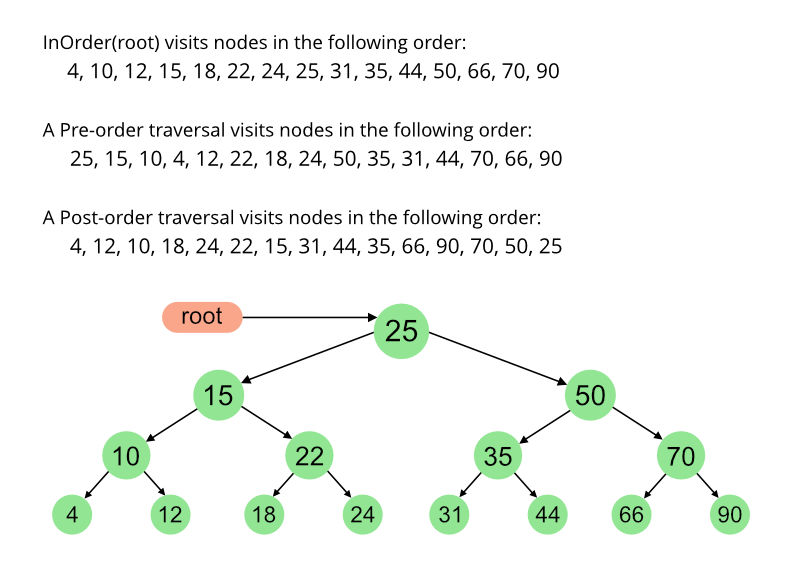
[ end of if ]

1. Exit

A Binary tree can be traversed through three basic algorithms: inorder, preorder, and postorder traversals

* **Inorder traversal:** Nodes from the left sub tree get visited first, followed by the root node and right sub tree.
* **Preorder traversal**: The root node gets visited first, followed by left and right sub trees.
* **Postorder traversal:** Nodes from the left sub tree get visited first, followed by the right sub tree, and finally the root.

**EXAMPLE:**



**CODE:**

#include <stdio.h> // Header files

#include <stdlib.h>

// Node struct of the binary tree

struct Node

{

int data;

struct Node \*left;

struct Node \*right;

};

// Prototyping the functions

void Inorder(struct Node \*);

void Preorder(struct Node \*);

void Postorder(struct Node \*);

struct Node \*dummyData();

// Driver Code

int main()

{

// The Binary tree representation as ASCII art

char binarytree\_rep[40] = " 1\n / \\\n 2 3\n / \\\n4 5";

printf("Binary Tree:\n%s", binarytree\_rep);

// Gets the Dummy binary tree to perform operations on

struct Node \*root = dummyData();

// Printing the Inorder traversal of binary tree

printf("\nInorder traversal:\n");

Inorder(root);

// Printing the Preorder traversal of binary tree

printf("\nPreorder traversal:\n");

Preorder(root);

// Printing the Postorder traversal of binary tree

printf("\nPostorder traversal:\n");

Postorder(root);

return 0;

}

struct Node \*newNode(int data)

{

// Function to create a new node with the data

struct Node \*node = (struct Node \*)malloc(sizeof(struct Node));

if (node == NULL)

{

printf("Overflow");

return node;

}

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

struct Node \*dummyData()

{

// Creating a dummy tree to do our operations on

// Linking our binary tree

struct Node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

return root;

}

void Inorder(struct Node \*node)

{

// Function to traverse through the binary tree in Inorder

// If tree is empty

if (node == NULL)

return;

// Inorder traversing through left subtree and then

// printing the root value

// and then Inorder traversing through right subtree

Inorder(node->left);

printf("%d ,", node->data);

Inorder(node->right);

}

void Preorder(struct Node \*node)

{

// Function to traverse through the binary tree in Preorder

// If tree is empty

if (node == NULL)

return;

// printing the root value then

// Inorder traversing through left subtree and then

// Inorder traversing through right subtree

printf("%d ,", node->data);

Preorder(node->left);

Preorder(node->right);

}

void Postorder(struct Node \*node)

{

// Function to traverse through the binary tree in Postorder

// If tree is empty

if (node == NULL)

return;

// Inorder traversing through left subtree and then

// Inorder traversing through right subtree and then

// printing the root value

Postorder(node->left);

Postorder(node->right);

printf("%d ,", node->data);

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Binary Tree:

1

/ \

2 3

/ \

4 5

Inorder traversal:

4 ,2 ,5 ,1 ,3 ,

Preorder traversal:

1 ,2 ,4 ,5 ,3 ,

Postorder traversal:

4 ,5 ,2 ,3 ,1 ,

**EXPERIMENT 10**

**AIM:** Implement insertion, deletion and traversals (inorder, preorder and postorder) on binary search tree with the information in the tree about the details of an automobile (type, company, year of make).

**ALGORITHM:**

**TRAVERSALS:**

**Algorithm Inorder-traversal(root)**

1. If root ≠ NULL then // Not empty tree
2. Inorder-traversal(root ->left)
3. Print root ->data
4. Inorder-traversal(root ->right)

[ end of if ]

1. Exit

**Algorithm Preorder-traversal(root)**

1. If root ≠ NULL then
2. Print root ->data
3. Preorder-traversal(root ->left)
4. Preorder-traversal(root ->right)

[ end of if ]

1. Exit

**Algorithm Postorder-traversal(root)**

1. If root ≠ NULL then
2. Postorder-traversal(root ->left)
3. Postorder-traversal(root ->right)
4. Print root ->data

[ end of if ]

1. Exit

**INSERTION INTO BST:**

**Algorithm Insert(root, node)**

1. If root = NULL
2. Root = node
3. Exit

[ End of if]

1. If node.value > root.value

Insert(root->right,node)

1. Else

Insert(root->left,node)

1. Exit

**DELETION IN BST:**

**Algorithm Delete(root, value)**

1. If root = NULL // Empty tree
2. Return cannot delete and exit

[End of if]

1. If value > root.value

Delete(root->right,value) and exit

1. Else if value < root.value

Delete(root->left,value) and exit

1. Else // root has the value
2. If root->left = NULL // Left sub tree is empty

Old root = root

Set root as right subtree

Free(old root)

exit

1. Else If root -> right = NULL // Right sub tree is empty

Old root= root

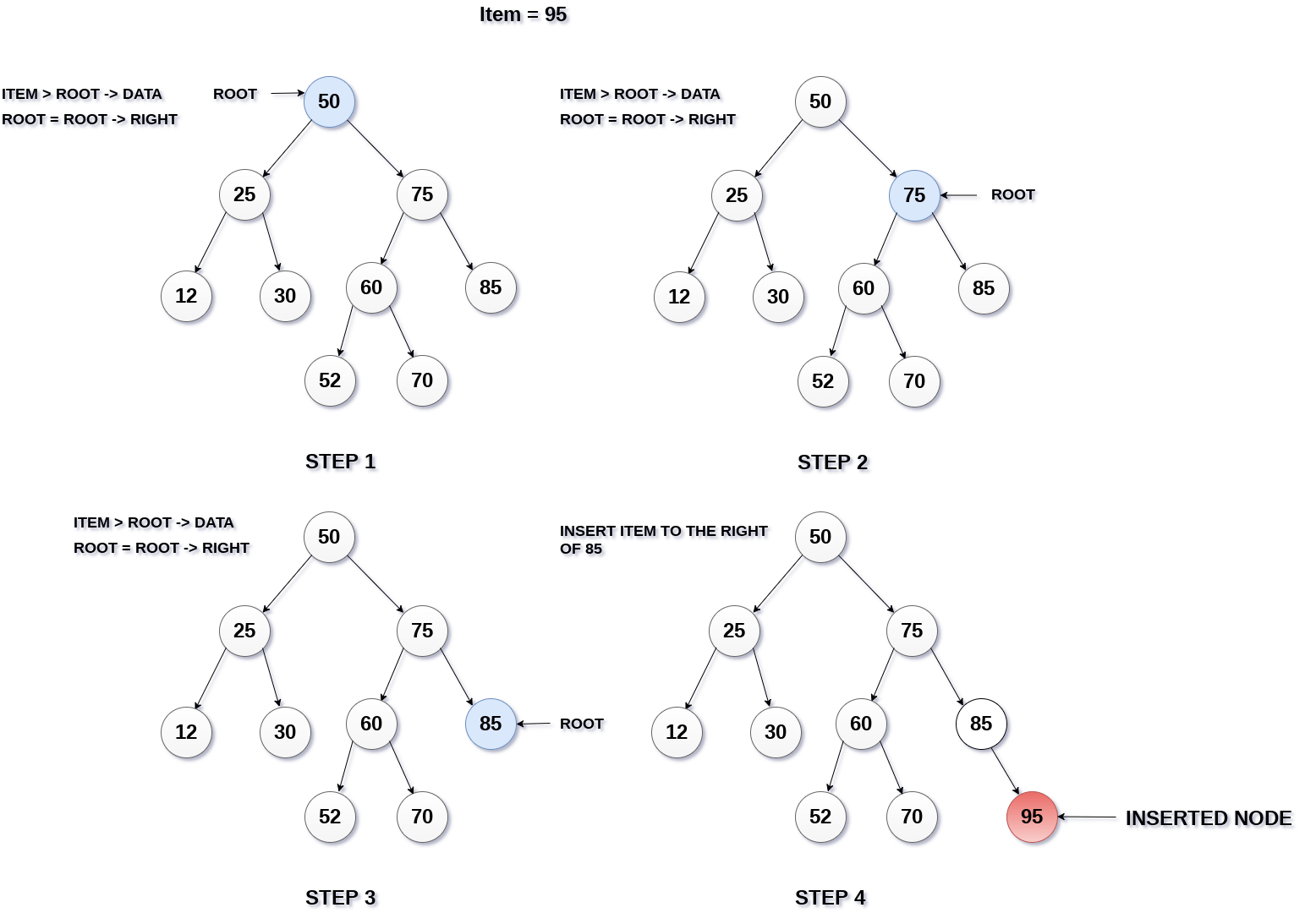
Set root as left subtree

Free(old root)

exit

1. Else, Largest = node with largest value in left subtree
2. Set root as largest node
3. Delete(root->left, largest.value) // Deleting the old largest node
4. Exit

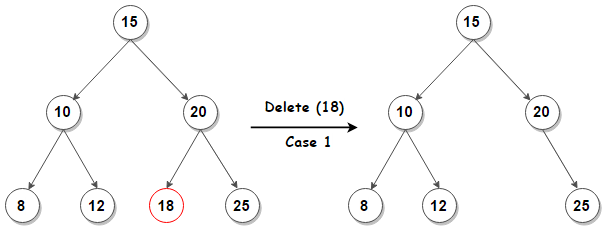
**EXAMPLE:**



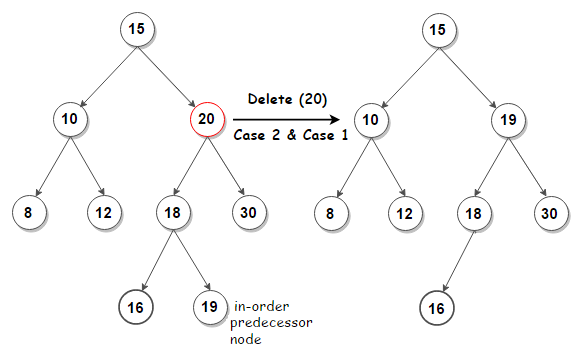
**INSERTION IN BINARY SEARCH TREE**

**DELETION IN BINARY TREE**

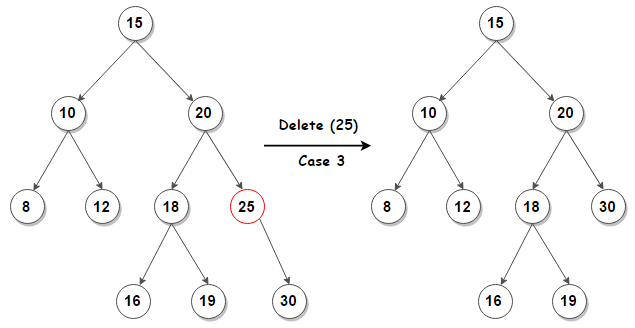
**Case 1**: Deleting a node with no children:



**Case 2:** Deleting a node with two children:



**Case 3:** Deleting a node with one child:



**CODE:**

#include <stdio.h> // Header files

#include <stdlib.h>

// Struct Automobile to store data about automobile

struct Automobile

{

char const \*type;

char const \*company;

int year\_of\_make;

};

// Node struct of the BST

struct Node

{

struct Automobile automobile;

struct Node \*left;

struct Node \*right;

};

// Prototyping the functions

struct Node \*dummyData();

struct Node \*newNode(struct Automobile);

void insert(struct Node \*\*, struct Node \*);

struct Node \*largest(struct Node \*);

int delete(struct Node \*\*, int);

void Inorder(struct Node \*);

void Preorder(struct Node \*);

void Postorder(struct Node \*);

// Driver code

int main()

{

// Gets a dummy BST for us to perform operations on

struct Node \*root = dummyData();

printf("Created BST:\n");

Preorder(root); // Displaying the BST using Preorder traversal

// Creating an automobile data and inserting it into the BST

struct Automobile automobile = {"Mercedes-Benz", "300 CE", 1993};

insert(&root, newNode(automobile));

printf("\nBST After inserting a Node:\n");

Preorder(root);

// Removing the car with year of make as 1994

delete(&root, 1994);

printf("\nBST After Removing a Car with year of make as 1994:\n");

Preorder(root);

// Printing the Inorder traversal of BST

printf("\nInorder traversal:\n");

Inorder(root);

// Printing the Preorder traversal of BST

printf("\nPreorder traversal:\n");

Preorder(root);

// Printing the Postorder traversal of BST

printf("\nPostorder traversal:\n");

Postorder(root);

return 0;

}

struct Node \*newNode(struct Automobile automobile)

{

// Function to create a new node with the automobile data

struct Node \*node = (struct Node \*)malloc(sizeof(struct Node));

if (node == NULL)

{

printf("Overflow");

return node;

}

node->automobile = automobile;

node->left = NULL;

node->right = NULL;

return node;

}

void insert(struct Node \*\*root, struct Node \*node)

{

// Function to insert a node into the BST

// Inserting on the basis of year of make

// If tree empty

if (\*root == NULL)

{

\*root = node;

return;

}

// If year of make of node greater than the root's year of make

if (node->automobile.year\_of\_make > (\*root)->automobile.year\_of\_make)

insert(&((\*root)->right), node); // Insert in right subtree

else

insert(&((\*root)->left), node); // Insert in left subtree

}

struct Node \*largest(struct Node \*root)

{

// Function for finding the largest node in the tree

// Used in the removing node from a BST function

// Checking the largest on the basis of year of make

int largest\_yom = root->automobile.year\_of\_make;

struct Node \*largest\_node = root;

if (root->left != NULL)

{

struct Node \*largest\_left = largest(root->left);

if (largest\_yom < largest\_left->automobile.year\_of\_make)

{

largest\_yom = largest\_left->automobile.year\_of\_make;

largest\_node = largest\_left;

}

}

if (root->right != NULL)

{

struct Node \*largest\_right = largest(root->right);

if (largest\_yom < largest\_right->automobile.year\_of\_make)

{

largest\_yom = largest\_right->automobile.year\_of\_make;

largest\_node = largest\_right;

}

}

// Returning the largest node

return largest\_node;

}

int delete(struct Node \*\*root, int year\_of\_make)

{

// Function to delete a node from the BST

// Deleting on the basis of year of make

// Returns 0 if deletion failed

// Returns 1 if deletion successful

// If tree empty

if (\*root == NULL)

return 0;

// If year of make of node greater than the root's year of make

if (year\_of\_make > (\*root)->automobile.year\_of\_make)

return delete(&((\*root)->right), year\_of\_make); // deletion in the right subtree

// If year of make of node leseer than the root's year of make

else if (year\_of\_make < (\*root)->automobile.year\_of\_make)

return delete(&((\*root)->left), year\_of\_make); // deletion in the left subtree

else // If year of make of node is equal to the root's year of make

{ // root is the node to be deleted

if ((\*root)->left == NULL) // if left subtree empty

{

// Set right subtree as root

struct Node \*temp = \*root;

\*root = (\*root)->right;

free(temp);

return 1;

}

else if ((\*root)->right == NULL) // if right subtree empty

{

// Set left subtree as root

struct Node \*temp = \*root;

\*root = (\*root)->left;

free(temp);

return 1;

}

else // Both subtrees not empty

{

// Putting the largest node from left subtree as root

struct Node \*largest\_left = largest((\*root)->left);

(\*root)->automobile = largest\_left->automobile;

// Deleting the largest node in the left subtree

return delete(&((\*root)->left), largest\_left->automobile.year\_of\_make);

}

}

}

struct Node \*dummyData()

{

// Creating a dummy tree to do our operations on

struct Node \*root = NULL;

struct Automobile a1, a2, a3;

a1.company = "BMW";

a1.type = "1 Series";

a1.year\_of\_make = 2013;

a2.company = "Audi";

a2.type = "100";

a2.year\_of\_make = 1994;

a3.company = "FIAT";

a3.type = "124 Spider";

a3.year\_of\_make = 2019;

// Inserting the automobiles into the BST

insert(&root, newNode(a1));

insert(&root, newNode(a2));

insert(&root, newNode(a3));

return root;

}

void Inorder(struct Node \*node)

{

// Function to traverse through the binary tree in Inorder

// If tree is empty

if (node == NULL)

return;

// Inorder traversing through left subtree and then

// printing the root value

// and then Inorder traversing through right subtree

Inorder(node->left);

printf("Company: %s, Type: %s, Year:%d\n", node->automobile.company, node->automobile.type, node->automobile.year\_of\_make);

Inorder(node->right);

}

void Preorder(struct Node \*node)

{

// Function to traverse through the binary tree in Preorder

// If tree is empty

if (node == NULL)

return;

// printing the root value then

// Inorder traversing through left subtree and then

// Inorder traversing through right subtree

printf("Company: %s, Type: %s, Year:%d\n", node->automobile.company, node->automobile.type, node->automobile.year\_of\_make);

Preorder(node->left);

Preorder(node->right);

}

void Postorder(struct Node \*node)

{

// Function to traverse through the binary tree in Preorder

// If tree is empty

if (node == NULL)

return;

// Inorder traversing through left subtree and then

// Inorder traversing through right subtree and then

// printing the root value

Postorder(node->left);

Postorder(node->right);

printf("Company: %s, Type: %s, Year:%d\n", node->automobile.company, node->automobile.type, node->automobile.year\_of\_make);

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Created BST:

Company: BMW, Type: 1 Series, Year:2013

Company: Audi, Type: 100, Year:1994

Company: FIAT, Type: 124 Spider, Year:2019

BST After inserting a Node:

Company: BMW, Type: 1 Series, Year:2013

Company: Audi, Type: 100, Year:1994

Company: 300 CE, Type: Mercedes-Benz, Year:1993

Company: FIAT, Type: 124 Spider, Year:2019

BST After Removing a Car with year of make as 1994:

Company: BMW, Type: 1 Series, Year:2013

Company: 300 CE, Type: Mercedes-Benz, Year:1993

Company: FIAT, Type: 124 Spider, Year:2019

Inorder traversal:

Company: 300 CE, Type: Mercedes-Benz, Year:1993

Company: BMW, Type: 1 Series, Year:2013

Company: FIAT, Type: 124 Spider, Year:2019

Preorder traversal:

Company: BMW, Type: 1 Series, Year:2013

Company: 300 CE, Type: Mercedes-Benz, Year:1993

Company: FIAT, Type: 124 Spider, Year:2019

Postorder traversal:

Company: 300 CE, Type: Mercedes-Benz, Year:1993

Company: FIAT, Type: 124 Spider, Year:2019

Company: BMW, Type: 1 Series, Year:2013**EXPERIMENT 11**

**AIM:** Implement Selection Sort, Bubble Sort, Insertion sort, Merge sort, Quick sort, and Heap Sort using array as a data structure.

**ALGORITHM:**

**SELECTION SORT:**

1. for i = 0 till i < list\_size - 1
2. Set smallest\_element\_index = i;
3. for j = i till j < list\_size
4. if (array[j] < array[smallest\_element\_index])
5. smallest\_element\_index = j;

[End of if]

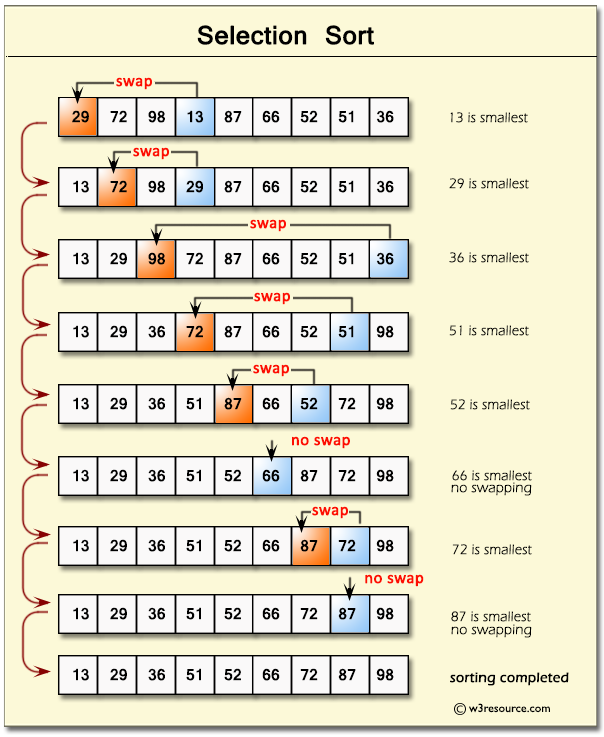
1. j= j+1

[End of for j loop]

1. if i != smallest\_element\_index
2. swap array[i] and array[smallest\_element\_index]
3. I= I+1

[End of for i loop]

1. Exit



**BUBBLE SORT:**

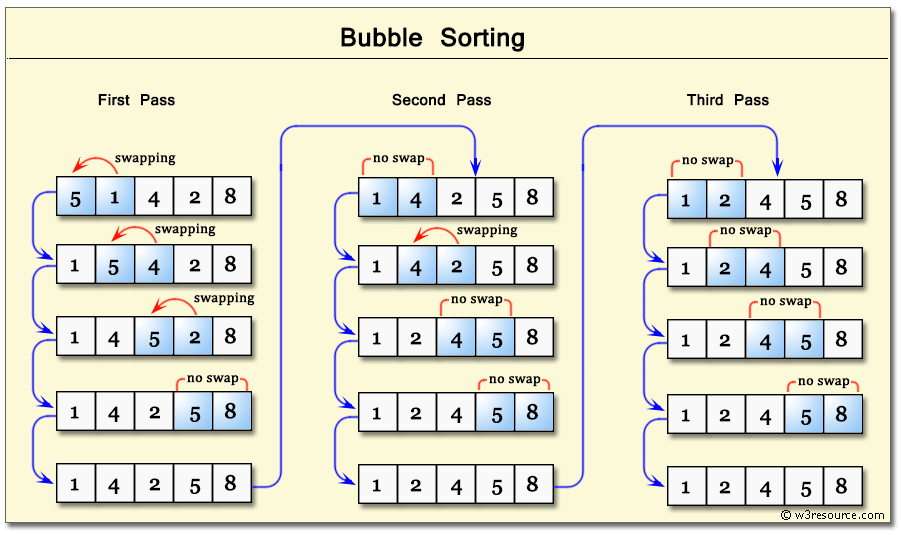
1. For i = 0 till i < list\_size - 1
2. For j = 0 till j < list\_size – i - 1
3. If array[j] > array[j+1]
4. Swap array[j] and array[j+1]
5. [end of if]
6. J = j+1

[end of for j loop]

1. I= i+1

[end of for i loop]

1. Exit



**INSERTION SORT:**

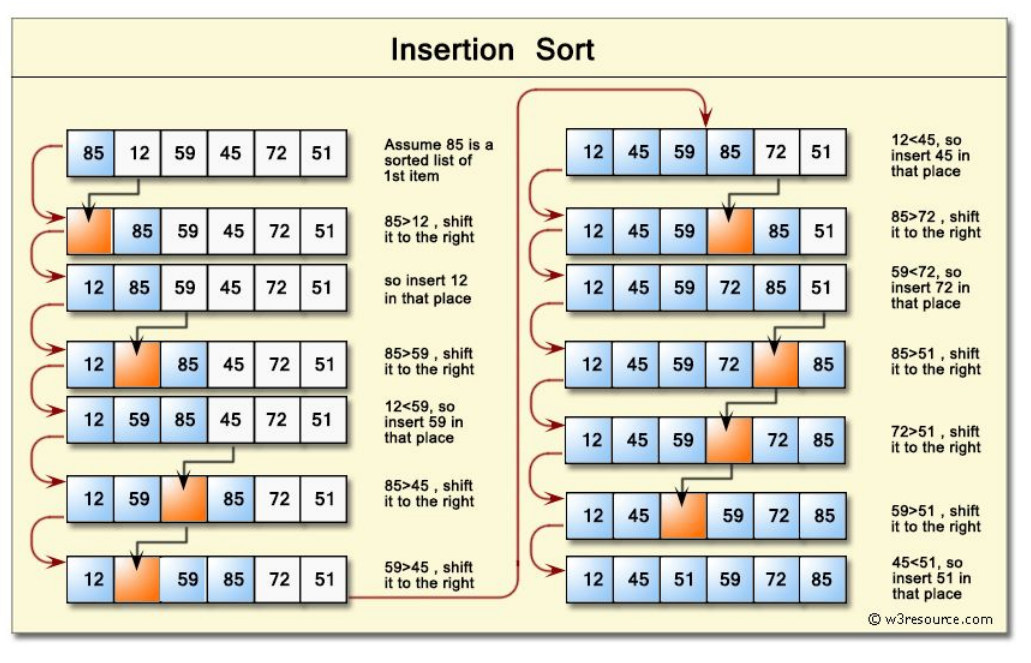
1. For i = 1 till i < list\_size
2. For j = i till j > 0
3. If array[j] < array[j-1]
4. Swap array[j] and array[j-1]
5. [end of if]
6. J = j-1

[end of for j loop]

1. I= i+1

[end of for i loop]

1. Exit



**MERGE SORT:**

**Algorithm merge\_sort(array, size)**

[check for overflow]

1. Returner\_array = (struct node \*) malloc (sizeof(int) \* size);
2. If Returner\_array= null
3. Print overflow   
   exit
4. If size = 1
5. Returner array[0] = array[0]
6. Return returner array and exit

[End of if]

1. Set mid = size/2
2. Sorted first half = merge\_sort(array, mid)
3. Sorted second half = merge\_sort(array + mid, size – mid)

// Merging

1. Set first\_half\_index, second\_half index,
2. While first\_half\_index < mid and second\_half\_index < size – mid
3. if (sorted\_first\_half[first\_half\_index]<= sorted\_second\_half[second\_half\_index])
4. append sorted\_first\_half[first\_half\_index] in returner\_array
5. first\_half\_index = first\_half\_index + 1

[End of if]

1. else append sorted\_second\_half[second\_half\_index] in returner\_array
2. second\_half\_index = second\_half\_index +1

[End of else]

[End of loop]

1. If first\_half\_index < mid\_pt
2. Append remaining elements of sorted first half to returner array

[End of if]

1. If second\_half\_index < size - mid\_pt
2. Append remaining elements of sorted second half to returner array

[End of if]

1. Free sorted first half and sorted second half
2. Return returner array
3. Exit



**MERGE SORT**

**HEAP SORT:**

1. Non\_leaf\_members = size/2 -1
2. For i = non leaf members till i >=0
3. Heapify(array,size, i)
4. I = i-1

[End of loop]

1. For i = size -1 till i>0
2. Swap array[0] and array[i]
3. Heapify(array, i, 0)
4. I = i-1

[end of loop]

**Algorithm heapify(array,size, index)**

1. Root = array[index]
2. Largest = largest( array[index] , array [2 \* index + 1]/ array[2 \* index + 2])
3. if(Root != Largest)
4. Swap(Root, Largest)
5. Heapify(array, size, largest\_index)

[End of if}

1. Exit



**HEAP SORT**

**QUICK SORT:**

1. Quick\_sort\_recursion(array, 0, size - 1)
2. Exit

**Algorithm quick\_sort\_recursion(array, low, high)**

1. If low < high
2. Pivot = partition(array, low, high)
3. If pivot > low
4. quick\_sort\_recursion(array, low, pivot - 1)

[end of if]

1. If pivot < high
2. quick\_sort\_recursion(array, pivot+1, high)

[end of if]

[end of if]

**Algorithm partition(array, low, high)**

1. Pivot = find\_pivot(array, low, high)
2. While true
3. While low <= high and value at low is lesser than value of pivot
4. Low = low+1

[ End of loop]

1. While low <= high and value at high is greater than value of pivot
2. high = high - 1

[ End of loop]

1. If low > high
2. If high < pivot
3. Swap array[low] and array [ pivot]
4. Pivot = low

[end of if]

1. Else
2. Swap array[high] and array [ pivot]
3. Pivot = high

[End of else]

1. Break the loop

[end of if]

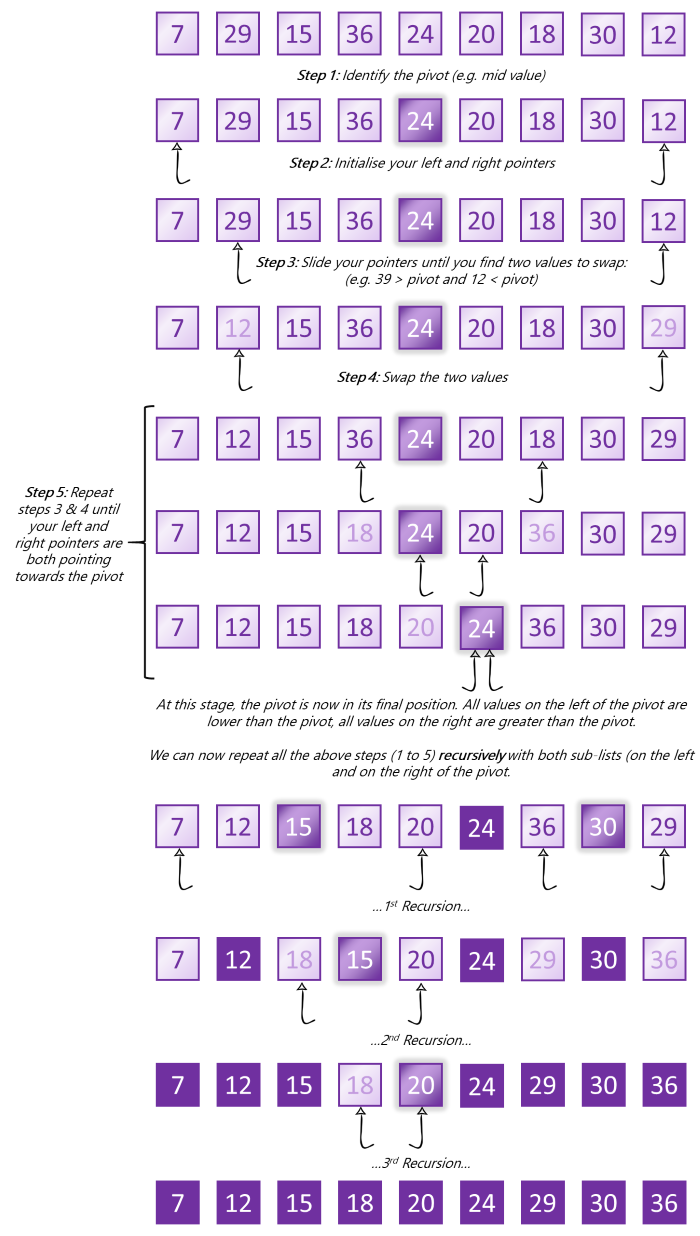
1. Swap array[high] and array[low]

[End of loop]

1. Return pivot
2. exit

**Algorithm find\_pivot(array, low, high)** // median of 3 method

1. Take high, low and middle element of the array
2. Find median of the three numbers
3. Set array[middle] = median
4. Set array[low] = smaller of the other two remaining numbers
5. Set array[high} = the last number
6. Return middle and exit



**CODE:**

#include <stdio.h> // Header files

#include <stdlib.h>

// Macro to swap 2 numbers

#define swap(a, b) \

{ \

int temp = a; \

a = b; \

b = temp; \

}

// Prototyping the functions

void selection\_sort(int[], int);

void bubble\_sort(int[], int);

void insertion\_sort(int[], int);

void quick\_sort(int[], int);

void heap\_sort(int[], int);

int \*merge\_sort(int[], int);

// Driver Code

int main()

{

// Original list to be sorted and its size

int array[] = {-82, 94, -42, -2, -84, 89, -49, 45, -12, 100, -35, -45, 77, -47, 16, 31, -65, 58, 99, -63, -51, -88, -78, 25, 90, -15, 91, -43, 84, -34};

int size = 30;

// Names cache for printing

char sorting\_methods[5][15] = {"Selection", "Bubble", "Insertion", "Quick", "Heap"};

// Copying the original list, since some functions perform swaps on original list

int copies[5][30];

for (int i = 0; i < 5; i++)

for (int j = 0; j < 30; j++)

copies[i][j] = array[j];

// Sorting the copies

selection\_sort(copies[0], size);

bubble\_sort(copies[1], size);

insertion\_sort(copies[2], size);

quick\_sort(copies[3], size);

heap\_sort(copies[4], size);

// Merge sort Creates new array

int \*merge\_sorted\_list = merge\_sort(array, size);

// Printing the original list

printf("Original List which was to be sorted: \n");

for (int j = 0; j < 30; j++)

printf("%d, ", array[j]);

printf("\n");

// Printing the sorted lists except merge sort

for (int i = 0; i < 5; i++)

{

printf("\nSorting the list using %s sort:\n", sorting\_methods[i]);

for (int j = 0; j < 30; j++)

printf("%d, ", copies[i][j]);

printf("\n");

}

// Printing the merge sort lisst

printf("\nSorting the list using Merge sort:\n");

for (int j = 0; j < 30; j++)

printf("%d, ", merge\_sorted\_list[j]);

return 0;

}

//====================== SELECTION SORT======================

void selection\_sort(int array[], int size)

{

// Selects the smallest elements and brings it to the starting

int smallest\_element\_index;

for (int i = 0; i < size - 1; i++)

{

smallest\_element\_index = i;

for (int j = i; j < size; j++)

if (array[j] < array[smallest\_element\_index])

smallest\_element\_index = j;

if (i != smallest\_element\_index)

swap(array[i], array[smallest\_element\_index]);

}

}

//====================== BUBBLE SORT======================

void bubble\_sort(int array[], int size)

{

// This function sorts the array using bubble sort

// Bubble sort bringss the biggest element to the end

for (int i = 0; i < size - 1; i++)

for (int j = 0; j < size - i - 1; j++)

if (array[j] > array[j + 1])

swap(array[j], array[j + 1]);

}

//====================== INSERTION SORT======================

void insertion\_sort(int array[], int size)

{

// Insertion sort divides the list into sorted and unsorted part

// brings each element from unsorted part into sorted part and

// move it to its position in the sorted list

for (int i = 1; i < size; i++)

for (int j = i; j > 0; j--)

if (array[j] < array[j - 1])

swap(array[j], array[j - 1]);

}

//====================== HEAP SORT======================

void heapify(int array[], int size, int index)

{

// Heapify converts an array to a max heap

// Finds the largest of elements at indices i, 2i + 1, 2i + 2

int largest = index;

int left = 2 \* index + 1;

int right = 2 \* index + 2;

if ((left < size) && (array[left] > array[largest]))

largest = left;

if ((right < size) && (array[right] > array[largest]))

largest = right;

// if value at index wasn't the largest

if (index != largest)

{

// swap value largest and index

swap(array[index], array[largest]);

// heapify largest index (which is not largest now)

heapify(array, size, largest);

}

}

void heap\_sort(int array[], int size)

{

// Heap sort converts the array into max heap and then systemetically removes the largest elements from heap

// and create a new sorted list

// leaf members for heap are from index size/2 and above

// Not includig non leaf members to increase speed

int non\_leaf\_members = size / 2 - 1;

for (int i = non\_leaf\_members; i >= 0; i--)

heapify(array, size, i); // convert array to max heap

// remove max element and heapify the remaining elements

for (int i = size - 1; i > 0; i--)

{

swap(array[0], array[i]);

heapify(array, i, 0);

}

}

//====================== MERGE SORT======================

int \*merge\_sort(int array[], int size)

{

// Merge sort works by dividing the list into 2 lists,

// sorting the sub lists and merging them back to form the sorted final list

// merge sort creates a new array to store the sorted list

// Returns NULL if overflow

// allocating space to returner array which is the new array

int \*returner\_array = (int \*)malloc(sizeof(int) \* size);

// Overflow

if (returner\_array == NULL)

{

printf("Overflow\n");

return NULL;

}

// If the array size is 1, list is already sorted

if (size == 1)

{

\*returner\_array = \*array;

return returner\_array;

}

// dividing the list into 2 parts

int mid\_pt = size / 2;

// sorting the two halves by recursion

int \*sorted\_first\_half = merge\_sort(array, mid\_pt);

int \*sorted\_second\_half = merge\_sort(array + mid\_pt, size - mid\_pt);

// If either of the halves are null means overflow detected

if (sorted\_first\_half == NULL || sorted\_second\_half == NULL)

return NULL;

// MERGING

int first\_half\_index = 0, second\_half\_index = 0, final\_list\_index = 0;

// While loop till either halves are exhausted

while (first\_half\_index < mid\_pt && second\_half\_index < size - mid\_pt)

{

// Going through elements of both lists and adding the elements into the returner array sequentially

if (sorted\_first\_half[first\_half\_index] <= sorted\_second\_half[second\_half\_index])

{

returner\_array[final\_list\_index] = sorted\_first\_half[first\_half\_index];

first\_half\_index++;

}

else

{

returner\_array[final\_list\_index] = sorted\_second\_half[second\_half\_index];

second\_half\_index++;

}

final\_list\_index++;

}

// One half is exhausted, in other some elements are left

// Add elements of first half if its not empty

while (first\_half\_index < mid\_pt)

{

returner\_array[final\_list\_index] = sorted\_first\_half[first\_half\_index];

first\_half\_index++;

final\_list\_index++;

}

// Add elements of second half if its not empty

while (second\_half\_index < size - mid\_pt)

{

returner\_array[final\_list\_index] = sorted\_second\_half[second\_half\_index];

second\_half\_index++;

final\_list\_index++;

}

// Free the memory allocated by the halves

free(sorted\_first\_half);

free(sorted\_second\_half);

return returner\_array;

}

//====================== QUICK SORT======================

int find\_pivot(int array[], int low, int high)

{

// Finding the pivot for quick sort

// This function uses the median of 3 to find the pivot

int pivot = (low + high) / 2;

// code for shifting the smallest of first last and middle elements to first index,

// largest to last index, and the median to middle index

if (array[low] > array[high])

swap(array[low], array[high]);

if (array[low] > array[pivot])

swap(array[low], array[pivot]);

if (array[pivot] > array[high])

swap(array[pivot], array[high]);

// returning the median

return pivot;

}

int partition(int array[], int low, int high)

{

// Partition is done by taking a pointer to the left most element and one at right most element

// swap right and left if left is higher than pivot and right is smaller than pivot

// replace the pivot with their position when they coincide or cross

// keeping the copies to remember the bounds of array

int low\_copy = low, high\_copy = high;

// getting the pivot

int pivot = find\_pivot(array, low, high);

while (1)

{

// while low is smaller than pivot and low and high didnt cross

while ((low <= high) && (array[low] <= array[pivot]))

low++;

// while high is smaller than pivot and low and high didnt cross

while ((low <= high) && (array[high] >= array[pivot]))

high--;

// here either low and high crossed or high is smaller than pivot and low is greater than pivot

// if high and low crossed

if (low > high)

{

// if they met at left end

if (high < low\_copy)

high = low;

// if they met at right end

if (low > high\_copy)

low = high;

// if they met at left of pivot

if (high < pivot)

{

swap(array[low], array[pivot]);

pivot = low;

}

// if they met at right of pivot

else

{

swap(array[high], array[pivot]);

pivot = high;

}

break;

}

// if high is smaller than pivot and low is greater than pivot

swap(array[high], array[low]);

}

// return the new position of pivot after swap

return pivot;

}

void quick\_sort\_recursion(int array[], int low, int high)

{

// This function recursively partitions the array into 2 parts and applies quick sort on both parts

if (low < high)

{

int pivot = partition(array, low, high);

// if statement for cases pivot == low or pivot == high

if (pivot > low)

quick\_sort\_recursion(array, low, pivot - 1);

if (pivot < high)

quick\_sort\_recursion(array, pivot + 1, high);

}

}

void quick\_sort(int array[], int size)

{

// Uses the recursion function to sort the array using quicksort algorithm

quick\_sort\_recursion(array, 0, size - 1);

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Original List which was to be sorted:

-82, 94, -42, -2, -84, 89, -49, 45, -12, 100, -35, -45, 77, -47, 16, 31, -65, 58, 99, -63, -51, -88, -78, 25, 90, -15, 91, -43, 84, -34,

Sorting the list using Selection sort:

-88, -84, -82, -78, -65, -63, -51, -49, -47, -45, -43, -42, -35, -34, -15, -12, -2, 16, 25, 31, 45, 58, 77, 84, 89, 90, 91, 94, 99, 100,

Sorting the list using Bubble sort:

-88, -84, -82, -78, -65, -63, -51, -49, -47, -45, -43, -42, -35, -34, -15, -12, -2, 16, 25, 31, 45, 58, 77, 84, 89, 90, 91, 94, 99, 100,

Sorting the list using Insertion sort:

-88, -84, -82, -78, -65, -63, -51, -49, -47, -45, -43, -42, -35, -34, -15, -12, -2, 16, 25, 31, 45, 58, 77, 84, 89, 90, 91, 94, 99, 100,

Sorting the list using Quick sort:

-88, -84, -82, -78, -65, -63, -51, -49, -47, -45, -43, -42, -35, -34, -15, -12, -2, 16, 25, 31, 45, 58, 77, 84, 89, 90, 91, 94, 99, 100,

Sorting the list using Heap sort:

-88, -84, -82, -78, -65, -63, -51, -49, -47, -45, -43, -42, -35, -34, -15, -12, -2, 16, 25, 31, 45, 58, 77, 84, 89, 90, 91, 94, 99, 100,

Sorting the list using Merge sort:

-88, -84, -82, -78, -65, -63, -51, -49, -47, -45, -43, -42, -35, -34, -15, -12, -2, 16, 25, 31, 45, 58, 77, 84, 89, 90, 91, 94, 99, 100,

**EXPERIMENT 12**

**AIM:** Perform Linear Search and Binary Search on an array. Description of program:

1. Read an array of type integer.
2. Input element from user for searching.
3. Search the element by passing the array to a function and then returning the position of the element from the function else return -1 if the element is not found.
4. Display the position where the element has been found.

**ALGORITHM:**

**LINEAR SEARCH:**

1. Get the number to be searched inputted from the user and SET it to item.
2. Loop through the array with index i
3. If item == array[i] , element is present in the array,

print the element is present,

exit

[End of Step 2 loop]

1. Else element is not present in the array,

Print the element is not present

Exit

The working behind a linear search is that we traverse through each element in the list and check if that’s the element we needed to search for.

Advantage: Easy to implement

Disadvantage: High computational order

**BINARY SEARCH:**

1. Get the number to be searched inputted from the user and SET it to item.
2. Sort the array.
3. Set lo = 0, hi = size of array - 1 // first and last element
4. Loop till hi –lo >1 //they are not coinciding or consecutive
5. Set mid =( hi + lo)//2
6. If array[mid] < item, lo = mid+1 //element in second half of list
7. Else hi = mid // element in first half of list

[End of Step 4 loop]

1. if item == array[hi] or array[low], element is present in the array,

Print the element is present

Exit

1. Else element is not present in the array,

Print the element is not present

Exit

The working behind binary search is to have a sorted list and then dividing the list into equal halves and checking in which half the item lies in and continue to divide it further until we are homed on one/two element(s) in the list.

The basic steps to perform Binary Search are:

* Begin with the mid element of the whole array as a search key.
* If the value of the search key is equal to the item then return an index of the search key.
* Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
* Otherwise, narrow it to the upper half.
* Repeatedly check from the second point until the value is found or the interval is empty.

EXAMPLE:

A = [1,2,3,4,5,6,7]

To find 3 in list A

Linear Search:

1. Item = 3
2. Loop through A with index i
3. I = 0, a[0] = 1, 1!=item
4. I = 1, a[1] = 2, 2!=item
5. I = 2, a[2] = 3, 3==item, print item present and exit

Binary search

1. Item = 3
2. Sort array A, A is already sorted
3. Lo = 0, hi = size of A -1 = 7-1 = 6
4. Loop till hi - lo > 1
5. Mid = (hi – lo)//2 = 3
6. A[mid] = A[3] = 4, 4>item, therefore hi =mid =3
7. Hi – lo = 3 > 1, loop continues
8. Mid = (hi-lo)//2 = 1
9. A[mid] = A[1] = 2, 2<item, therefore lo =mid+1 =2
10. Hi – lo = 1 not > 1, loop ends
11. (A[2] == 3) or (A[3] ==3} is true
12. Print element is present in the list

**CODE:**

// Header files

#include <stdio.h>

#include <stdlib.h>

// Prototyping the functions

void bubble\_sort(int \*, int, int \*\*);

int linear\_search(int[], int, int);

int binary\_search(int[], int, int);

int \*input\_list(int \*);

int main()

{

// Getting the list inputted

int size, \*list;

list = input\_list(&size);

// Getting the element to be searched from the user

int search\_element;

printf("Input the search element: ");

scanf("%d", &search\_element);

// Doing searches using functions

int linear\_index = linear\_search(list, size, search\_element);

int binary\_index = binary\_search(list, size, search\_element);

if (linear\_index >= 0)

printf("Found the element using linear search at index %d.\n", linear\_index);

else

printf("Could not find the element using linear search.\n", linear\_index);

if (binary\_index >= 0)

printf("Found the element using binary search at index %d. \n", binary\_index);

else

printf("Could not find the element using binary search. \n", binary\_index);

return 0;

}

int linear\_search(int list[], int size, int search\_element)

{

// This function uses linear search to find the element in list

for (int i = 0; i < size; i++) // Traversing through the list

{

if (list[i] == search\_element) // If element found

return i;

}

// If foor loop is Completed and element not found

return -1;

}

int binary\_search(int list[], int size, int search\_element)

{

// This function searches for an element using binary search

int \*index\_list;

bubble\_sort(list, size, &index\_list); // binary search uses a sorted list

int lo = 0, hi = size - 1; // first and last index

int mid;

// while indexes not coinciding or consecutive

while (hi - lo > 1)

{

int mid = (hi + lo) / 2;

if (list[mid] < search\_element)

{ // element is in the second half of the list

lo = mid + 1;

}

else

{ // element is in the first half of the list

hi = mid;

}

}

int lo\_actual\_index = index\_list[lo];

int hi\_actual\_index = index\_list[hi];

free(index\_list);

// Loop finished with finding the element

if (list[lo] == search\_element)

return lo\_actual\_index;

if (list[hi] == search\_element)

return hi\_actual\_index;

// Loop finished without finding the element

else

return -1;

}

void bubble\_sort(int list[], int size, int \*\*index\_list)

{

// This function sorts the list using bubble sort

int \*local\_index\_list = (int \*)malloc(size \* sizeof(int));

for (int i = 0; i < size; i++)

{

local\_index\_list[i] = i;

}

\*index\_list = local\_index\_list;

for (int i = 0; i < size - 1; i++)

{

for (int j = 0; j < size - i - 1; j++)

{

if (list[j] > list[j + 1])

{

int swap\_var = list[j];

list[j] = list[j + 1];

list[j + 1] = swap\_var;

swap\_var = local\_index\_list[j];

local\_index\_list[j] = local\_index\_list[j + 1];

local\_index\_list[j + 1] = swap\_var;

}

}

}

}

int \*input\_list(int \*size)

{

// This function inputs the list from user

int \*array;

printf("Input the number of elements in the list: ");

scanf("%d", size);

// Allocating the memory to the array

array = (int \*)malloc(\*size \* sizeof(int));

// Putting the elements into the array

for (int i = 0; i < \*size; i++)

{

printf("Input element no %d of the list: ", i + 1);

scanf("%d", array + i);

}

return array;

}

**OUTPUT:**

* **Test Case 1:** Element in the list

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Input the search element: 10

Found the element 10 at position 3 using linear search.

Found the element 10 at position 3 using binary search.

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Input the search element: 30

Found the element 30 at position 4 using linear search.

Found the element 30 at position 4 using binary search.

* **Test Case 2:** Element not in the list

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Input the search element: 50

Could not find the element 50 in the list using linear search.

Could not find the element 50 in the list using binary search.

**EXPERIMENT 13**

**AIM:** Implement the searching using hashing method.

**ALGORITHM:**

**HASHING:**

1. Set integer\_key = 0
2. For each character in key
3. Integer\_key = integer\_key + character // Fold Hashing
4. Rotate address 12 bits right // Rotation Hashing

[end loop]

1. Address = integer\_key % list size // Modulo Hashing
2. Return address, integer\_key
3. Exit

**EXAMPLE:**

Take a key “Hat” for example,

List\_size = 307

1. Integer key = (00000000 00000000 00000000 00000000b)
2. For ‘H’ in “Hat”
3. Integer\_key = integer\_key + H(01001000b)=(00000000 00000000 00000000 01001000b)
4. Rotate integer\_key by 12 bits,
5. Integer\_key = (0000 01001000 00000000 00000000 0000b)
6. For ‘a’ in “Hat”
7. Integer\_key = integer\_key +a(01100001b) =(0000 01001000 00000000 0000 01100001b)
8. Rotate integer\_key by 12 bits,
9. Integer\_key = ( 0000 01100001 0000 01001000 00000000b)
10. For ‘t’ in “Hat”
11. Integer\_key = integer\_key +t(01110100b)= (0000 01100001 0000 01001000 01110100b)
12. Rotate integer\_key by 12 bits,
13. Integer\_key =(1000 01110100 0000 01100001 0000 0100b)
14. Integer\_key = 2269143300
15. address = 2269143300 % 307 = 78

**INSERTING VALUE INTO HASHED LIST:**

**Algorithm insert(key, address)**

1. If list[address] ->is\_filled = 0 or list[address]->key = key
2. List[address]->is\_filled = 1,
3. List[address]->key = key,
4. List[address]->data = item

[end of if]

Else, // Collision

1. address = (address + hash(key))% list\_size // Double hashing
2. Insert(key, address)

[End of else]

1. Exit

**EXAMPLE:**

Take key “Hat” and lets assume another key alpha has same integer\_key = 2269143300

Hash for double hashing is defined as hash(integer\_key) = integer\_key/ list\_size

First input data 210 at “Hat”,

1. Address = 78
2. List[78] -> is\_filled ==0 so,
3. List[78]-> is\_filled =1
4. List[78]->key = “Hat”
5. List[78]->data = 210

Now input data 425 at alpha

1. Address = 78
2. List[78] ->is\_filled == 1 and list[address]->key != alpha so,
3. Address = address + hash = (78 + 2269143300/307)%307 = 7391424%307 = 92
4. List[92] ->is\_filled ==0 so,
5. List[92]-> is\_filled =1
6. List[92]->key = alpha
7. List[92]->data = 425

**SEARCHING VALUE FROM HASHED LIST:**

**Algorithm search(key, address)**

1. If list[address] ->is\_filled = 0
2. Return no data and exit

[End of if]

1. if list[address]->key = key
2. Return List[address]->data and exit

[end of if]

Else, // Collision

1. address = (address + hash(key))% list\_size // Double hashing
2. search(key, address)

[End of else]

1. Exit

**EXAMPLE:**

Search data at alpha

1. Address = 78
2. List[78] ->is\_filled == 1 and list[address]->key != alpha so,
3. Address = address + hash = (78 + 2269143300/307)%307 = 7391424%307 = 92
4. List[92] ->is\_filled == 1 and list[address]->key == alpha so,
5. Return List[92] = 425

**CODE:**

#include <stdio.h> // Header Files

#include <stdlib.h>

// MACROS for hashing

#define modulo\_hash(key, list\_size) (key % list\_size)

#define offset\_hash(key, list\_size) (key / list\_size)

// List Element struct whose array will create our list

struct ListElement

{

char const \*key;

int is\_filled;

int data;

};

// Prototyping the functions

unsigned int insert(char const \*, int, struct ListElement \*, int);

int search(char const \*, struct ListElement \*, int);

struct ListElement \*createEmptyList(int size);

// Driver Code

int main()

{

// Creating a list of size 307 (prime number)

int size = 307;

struct ListElement \*list = createEmptyList(size);

// Inserting the values 10 at key "Saksham",

// 12 at "Gupta" and 14 at "DS file"

unsigned int key1 = insert("Saksham", 10, list, size);

unsigned int key2 = insert("Gupta", 12, list, size);

unsigned int key3 = insert("DS file", 14, list, size);

// Searching and printing the values at the keys

printf("Searching in the hashed list key \"Saksham\":\n%d\n", search("Saksham", list, size));

printf("Searching in the hashed list key \"Gupta\":\n%d\n", search("Gupta", list, size));

printf("Searching in the hashed list key \"DS file\":\n%d\n", search("DS file", list, size));

return 0;

}

unsigned int string\_to\_key(char const \*key\_string)

{

// This function converts the string to a number key using various hashing methods

unsigned int key = 0;

unsigned int key\_copy;

int i = 0;

while (key\_string[i] != '\0')

{

key\_copy = key; // ROTATION HASHING

key >>= 12;

key\_copy <<= sizeof(int) \* 8 - 12;

key += key\_copy;

key += key\_string[i]; // FOLD HASHING

i++;

}

return key;

}

void insert\_recursion(const char \*key\_string, unsigned int key\_int, int hashed\_key, int data, struct ListElement \*list, int size)

{

// This function inserts the given value at given hash key

// If the hashed key address is not filled or is filled by the same string key

if (list[hashed\_key].is\_filled == 0 || list[hashed\_key].key == key\_string)

{

// Fill the adress

list[hashed\_key].is\_filled = 1;

list[hashed\_key].key = key\_string;

list[hashed\_key].data = data;

return;

}

else

{

// COLLISION

// Resolution by douuble hashing method

hashed\_key = hashed\_key + offset\_hash(key\_int, size);

insert\_recursion(key\_string, key\_int, hashed\_key, data, list, size);

}

}

unsigned int insert(char const \*key, int data, struct ListElement \*list, int size)

{

// This function inserts the given value at given string key

// Converts the string key to integer key

unsigned int key\_int = string\_to\_key(key);

// Uses a recursion function to store the value at the given key

// Uses the hash key as modulo hash

insert\_recursion(key, key\_int, modulo\_hash(key\_int, size), data, list, size);

return key\_int;

}

int search\_recursion(const char \*key\_string, unsigned int key\_int, int hashed\_key, struct ListElement \*list, int size)

{

// This function searches the value at given hash key

// Returns -1 if no data

// If no data return -1

if(list[hashed\_key].is\_filled == 0)

return -1;

// If the hashed key address is filled by the same string key

if (list[hashed\_key].key == key\_string)

// return data

return list[hashed\_key].data;

else

{

// COLLISION

// Resolution by douuble hashing method

hashed\_key = hashed\_key + offset\_hash(key\_int, size);

return search\_recursion(key\_string, key\_int, hashed\_key, list, size);

}

}

int search(char const \*key, struct ListElement \*list, int size)

{

// This function searches the value at the given string key

unsigned int key\_int = string\_to\_key(key);

// Uses a recursion function to search the value at the given string key

// Uses the hash key as modulo hash since stored the same

return search\_recursion(key, key\_int, modulo\_hash(key\_int, size), list, size);

}

struct ListElement \*createEmptyList(int size)

{

// Allocate the memory for the list of given size

struct ListElement \*list = (struct ListElement \*)malloc(size \* (sizeof(struct ListElement)));

// Set All elements as not filled

for (int i = 0; i < size; i++)

list[i].is\_filled = 0;

return list;

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Searching in the hashed list key "Saksham":

10

Searching in the hashed list key "Gupta":

12

Searching in the hashed list key "DS file":

14

**EXPERIMENT 14**

**AIM:** Create a graph and perform DFS and BFS traversals.

**ALGORITHM:**

**GRAPH STRUCTURE**



**DEPTH FIRST SEARCH TRAVERSAL:**

1. If vertex = Null , exit // Empty Graph
2. Else, set ptr = vertex
3. While ptr != NULL
4. Ptr->processed = 0 // Go through every vertex and set as unprocessed
5. Ptr = ptr->next

[End of loop]

1. depth-first-recursion(vertex)
2. Exit

**Algorithm depth-first-recursion(vertex)**

1. If vertex-> processed = 1, return
2. Else print vertex->data
3. Vertex->processed = 1
4. set ptr = vertex->firstArc
5. While ptr != NULL
6. depth-first-recursion(ptr->destination)
7. Ptr = ptr->nextArc

[End loop]

1. Exit

**BREADTH FIRST SEARCH TRAVERSAL:**

1. If vertex = Null , exit // Empty Graph
2. Else, set ptr = vertex
3. While ptr != NULL
4. Ptr->processed = 0 // Go through every vertex and set as unprocessed
5. Ptr = ptr->next

[End of loop]

1. Ptr = vertex
2. While ptr != NULL // Go through every vertex
3. If ptr->processed = 0
4. Enqueue ptr
5. Ptr->processed = 1
6. While queue not empty
7. Set print\_vertex = dequeue
8. Print print\_vertex->data
9. set arcptr = print\_vertex->firstArc
10. While arcptr != NULL // Go through every arc
11. If arcptr->destination->processed = 0
12. Enqueue arcptr->destination
13. arcptr->destination->processed = 1

[End of if]

1. Arcptr = arcptr->nextArc

[End of arc ptr while loop]

[End of empty queue loop]

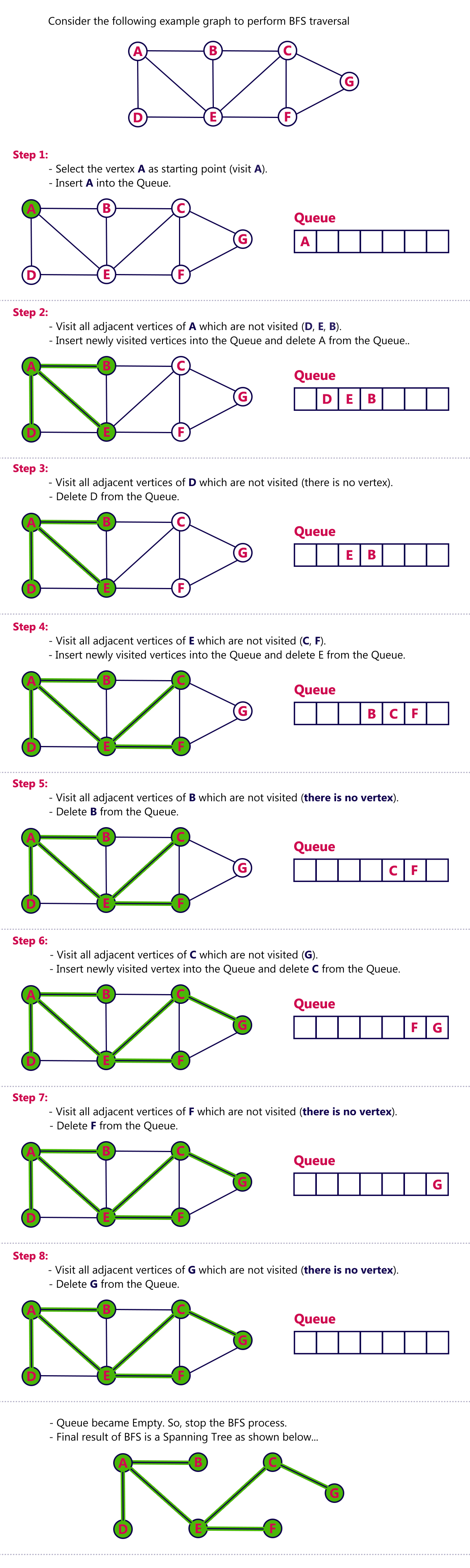
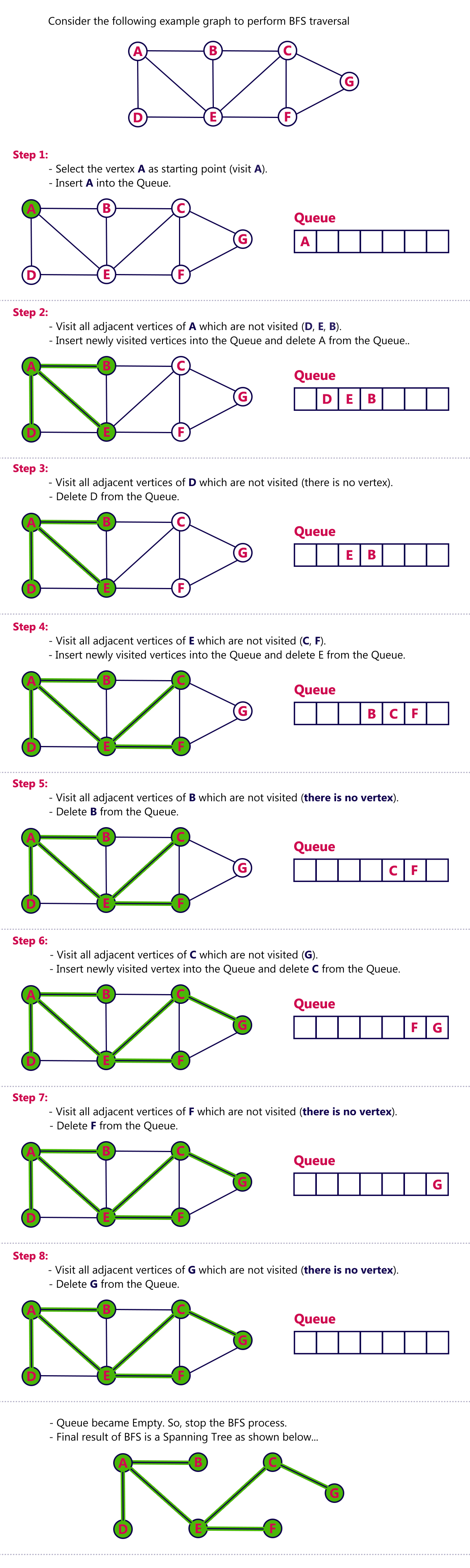
[End of if]

1. Ptr = ptr->nextVertex

[End of ptr while loop]

1. Exit

**DEPTH FIRST SEARCH TRAVERSAL**



**CODE:**

#include <stdio.h> // Header files

#include <stdlib.h>

// For Queue

struct Node

{

struct graphVertex \*vertex;

struct Node \*next;

};

struct Node \*front = NULL;

struct Node \*rear = NULL;

// Struct to store the other end of the vertex's edge

struct graphArc

{

struct graphVertex \*destination; // The other end's vertex

struct graphArc \*nextArc; // The next edge's arc

};

// The Vertex element whose linked list represents the graph

struct graphVertex

{

struct graphVertex \*nextVertex; // Next vertex in the list

int data; // Data of the vertex

int processed; // If the data has been read already( For cycle graphs )

struct graphArc \*firstArc; // First edge's arc

};

// Prototyping the functionis

void depth\_first\_traversal(struct graphVertex \*);

void breadth\_first\_traversal(struct graphVertex \*);

struct graphVertex \*dummyGraph();

// Driver code

int main()

{

// The Graph representation as ASCII art

char graph\_representation[25] = "0 - 1\n| / | \\\n4 - 3 - 2";

printf("Graph:\n%s", graph\_representation);

// Gets the Dummy Graph to perform operations on

struct graphVertex \*vertex = dummyGraph();

// Printing the DFS traversal of the graph

printf("\nDFS traversal:\n");

depth\_first\_traversal(vertex);

// Printing the BFS traversal of the graph

printf("\nBFS traversal:\n");

breadth\_first\_traversal(vertex);

return 0;

}

// enqueue element into queue

void enqueue(struct graphVertex \*vertex)

{

// Function to enqueue an element into the queue

// Allocating memory to the new node in the queue

struct Node \*newNode = (struct Node \*)malloc(sizeof(struct Node));

// Overflow

if (newNode == NULL)

{

printf("Overflow\n");

return;

}

// Assigning values to the node

newNode->vertex = vertex;

newNode->next = NULL;

// replacing the rear

if (front == NULL)

front = rear = newNode;

else

{

rear->next = newNode;

rear = newNode;

}

}

struct graphVertex \*dequeue()

{

// Function to dequeue and return element from queue

// Checks if elements present

// Returns NULL if underflow

if (front == NULL)

{

printf("Underflow\n");

return NULL;

}

else

{

struct Node \*temp = front;

struct graphVertex \*element = front->vertex; // to store data of rear node

front = front->next;

if (front == NULL)

rear = front;

free(temp); // deleting the node

return element;

}

}

struct graphVertex \*addGraphEdge(struct graphVertex \*vertex, int a, int b)

{

// Function to add 2 vertexes which form an edge into the graph

// Vertexes with data a and b

struct graphVertex \*a\_vertex = NULL, \*b\_vertex = NULL;

// Loop variable ptr to loop through the graph

struct graphVertex \*ptr = vertex;

// If not empty graph check if a vertex and b vertex exist

if (ptr != NULL)

{

while (1)

{

if (ptr->data == a)

a\_vertex = ptr;

if (ptr->data == b)

b\_vertex = ptr;

if (a\_vertex != NULL && b\_vertex != NULL)

break;

if (ptr->nextVertex == NULL)

break;

ptr = ptr->nextVertex;

}

}

// Ptr is pointing at the end of the list

// a\_vertex is the vertex in graph which has data a or NULL if doesnt exit

// b\_vertex is the vertex in graph which has data b or NULL if doesnt exit

// If a\_vertex not in list

if (a\_vertex == NULL)

{

// Create a\_vertex

struct graphVertex \*a\_temp = (struct graphVertex \*)malloc(sizeof(struct graphVertex));

if (a\_temp == NULL) // Check Overflow

{

printf("Overflow\n");

return vertex;

}

// Assigning value to a\_vertex (temp)

a\_temp->nextVertex = NULL;

a\_temp->data = a;

a\_temp->processed = 64;

a\_temp->firstArc = NULL;

// Assigning temp to the main

a\_vertex = a\_temp;

// If list is empty

if (ptr == NULL)

ptr = a\_vertex;

// Else append at the end

else

{

ptr->nextVertex = a\_vertex;

ptr = ptr->nextVertex;

}

}

// If b\_vertex not in list

if (b\_vertex == NULL)

{

// Create b\_vertex

struct graphVertex \*b\_temp = (struct graphVertex \*)malloc(sizeof(struct graphVertex));

if (b\_temp == NULL) // Check Overflow

{

printf("Overflow\n");

return vertex;

}

// Assigning value to b\_vertex(temp)

b\_temp->nextVertex = NULL;

b\_temp->data = b;

b\_temp->processed = 32;

b\_temp->firstArc = NULL;

// Assigning temp to the main

b\_vertex = b\_temp;

// Appending at the end

ptr->nextVertex = b\_vertex;

ptr = ptr->nextVertex;

}

// Arcs with data a and b

// a\_arc is in b\_vertex and b\_arc is in a\_vertex

struct graphArc \*a\_arc = NULL, \*b\_arc = NULL;

// Loop variables to loop through the edges of vertices

struct graphArc \*aptr = a\_vertex->firstArc, \*bptr = b\_vertex->firstArc;

// If not empty check if b\_arc exists in a\_vertex

while (aptr != NULL)

{

if (aptr->destination == b\_vertex)

{

b\_arc = aptr;

break;

}

if (aptr->nextArc == NULL)

break;

aptr = aptr->nextArc;

}

// If not empty check if a\_arc exists in b\_vertex

while (bptr != NULL)

{

if (bptr->destination == a\_vertex)

{

a\_arc = bptr;

break;

}

if (bptr->nextArc == NULL)

break;

bptr = bptr->nextArc;

}

// aptr is pointing at the end of the edges of a\_vertex

// bptr is pointing at the end of the edges of b\_vertex

// a\_arc is the arc in b\_vertex which has data a or NULL if doesnt exit

// b\_arc is the arc in a\_vertex which has data b or NULL if doesnt exit

// If a\_arc not in edges

if (a\_arc == NULL)

{

// Create a\_arc

struct graphArc \*a\_arc\_temp = (struct graphArc \*)malloc(sizeof(struct graphArc));

if (a\_arc\_temp == NULL) // Check Overflow

{

printf("Overflow\n");

return vertex;

}

// Assigning value to a\_arc (temp)

a\_arc\_temp->nextArc = NULL;

a\_arc\_temp->destination = a\_vertex;

// Assigning temp to the main

a\_arc = a\_arc\_temp;

// If no edges

if (bptr == NULL)

b\_vertex->firstArc = a\_arc\_temp;

// Else append at the end

else

bptr->nextArc = a\_arc\_temp;

}

if (b\_arc == NULL)

{

// Create b\_arc

struct graphArc \*b\_arc\_temp = (struct graphArc \*)malloc(sizeof(struct graphArc));

if (b\_arc\_temp == NULL) // Check Overflow

{

printf("Overflow\n");

return vertex;

}

// Assigning value to b\_arc (temp)

b\_arc\_temp->nextArc = NULL;

b\_arc\_temp->destination = b\_vertex;

// Assigning temp to the main

b\_arc = b\_arc\_temp;

// If no edges

if (aptr == NULL)

a\_vertex->firstArc = b\_arc\_temp;

// Else append at the end

else

aptr->nextArc = b\_arc\_temp;

}

// If graph is empty

if (vertex == NULL)

return a\_vertex;

// Else return original list

else

return vertex;

}

struct graphVertex \*dummyGraph()

{

// This function creates Dummy data for us to work on

struct graphVertex \*vertex = addGraphEdge(NULL, 0, 1);

addGraphEdge(vertex, 0, 4);

addGraphEdge(vertex, 1, 3);

addGraphEdge(vertex, 1, 4);

addGraphEdge(vertex, 4, 3);

addGraphEdge(vertex, 3, 2);

addGraphEdge(vertex, 1, 2);

// Returns the graph

return vertex;

}

void depth\_first\_recursion(struct graphVertex \*vertex)

{

// If already printed return

if (vertex->processed == 1)

return;

// Else process

printf("%d, ", vertex->data);

vertex->processed = 1;

// Loop through the edges of the vertex

struct graphArc \*ptr = vertex->firstArc;

while (ptr != NULL)

{

// Recursively process the nodes by looping through arcs

depth\_first\_recursion(ptr->destination);

ptr = ptr->nextArc;

}

}

void depth\_first\_traversal(struct graphVertex \*vertex)

{

// This function is to traverse through the graph using DFS

// If graph is empty, return

if (vertex == NULL)

return;

// Loop through the vertices and set them as unprocessed

struct graphVertex \*ptr = vertex;

while (ptr != NULL)

{

ptr->processed = 0;

ptr = ptr->nextVertex;

}

// Use recursion to proecess the nodes

depth\_first\_recursion(vertex);

}

void breadth\_first\_traversal(struct graphVertex \*vertex)

{

// This function is to traverse through the graph using DFS

// If graph is empty, return

if (vertex == NULL)

return;

// Loop through the vertices and set them as unprocessed

struct graphVertex \*ptr = vertex;

while (ptr != NULL)

{

ptr->processed = 0;

ptr = ptr->nextVertex;

}

// Place the pointer back to the starting and loop through the vertices again

ptr = vertex;

while (ptr != NULL)

{

// If vertex not processed

if (ptr->processed == 0)

{

// enqueue the vertex into queue for processing

enqueue(ptr);

ptr->processed = 1;

// Loop for Deleting from queue and processing the vertex

while (front != NULL)

{

// Deleting from queue and processing the vertex

struct graphVertex \*print = dequeue();

printf("%d, ", print->data);

// Loop through the edges of the vertex

struct graphArc \*aptr = print->firstArc;

while (aptr != NULL)

{

// If vertex not processed

if (aptr->destination->processed == 0)

{

// enqueue the vertex into queue for processing

enqueue(aptr->destination);

aptr->destination->processed = 1;

}

aptr = aptr->nextArc;

}

}

}

ptr = ptr->nextVertex;

}

}

**OUTPUT:**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Graph:

0 - 1

| / | \

4 - 3 - 2

DFS traversal:

0, 1, 3, 4, 2,

BFS traversal:

0, 1, 4, 3, 2,