

Title: Write a program to evaluate the user input postfix or prefix expression.

Prefix and Postfix expressions can be evaluated faster than an infix expression. This is because we don't need to process any brackets or follow operator precedence rule. In postfix and prefix expressions which ever operator comes before will be evaluated first, irrespective of its priority. Also, there are no brackets in these expressions. As long as we can guarantee that a valid prefix or postfix expression is used, it can be evaluated with correctness. We can convert infix to postfix and can convert infix to prefix using stacks.

To evaluate a postfix expression, we can use a stack. Idea is to iterate the expression from left to right and keep on storing the operands into a stack. Once an operator is received, pop the two topmost elements and evaluate them and push the result in the stack again.

Similarly, to evaluate a prefix expression, we can use a stack. The idea is to iterate through the expression from right to left, and keep pushing the operands into the stack. When an operator is encountered, pop the top two operands from the stack, evaluate the operation, and push the result back onto the stack.

Compiler: Dev C++

Language: C

Source Code:

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

#define MAX 100

char stack[MAX];
int top = -1; // fixed initialization

// Stack operations
void push(char c) {
    if (top >= MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = c;
}

char pop() {
    if (top < 0) {
        printf("Stack Underflow\n");
        return '\0';
    }
    return stack[top--];
}

int precedence(char op) {
    switch (op) {
        case '^': return 3;
        case '*':
        case '/': return 2;
        case '+':
        case '-': return 1;
        default: return 0;
    }
}

int isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^';
}

// Function to reverse a string
void reverse(char str[]) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {
        char tmp = str[i];
        str[i] = str[len - i - 1];
        str[len - i - 1] = tmp;
    }
}
```

```

// Infix to Postfix conversion
void infixToPostfix(char infix[], char postfix[]) {
    int i, j = 0;
    char item;

    printf("\n+-----+-----+-----+\n");
    printf("| Scanned Symbol | Operator Stack | Operand Stack |\n");
    printf("+-----+-----+-----+\n");

    for (i = 0; infix[i] != '\0'; i++) {
        item = infix[i];
        if (item == ' ')
            continue;

        if (isalnum(item)) {
            postfix[j++] = item;
            postfix[j] = '\0';
        } else if (item == '(') {
            push(item);
        } else if (item == ')') {
            while (top != -1 && stack[top] != '(') {
                postfix[j++] = pop();
                postfix[j] = '\0';
            }
            if (top != -1) pop(); // pop '('
        } else if (isOperator(item)) {
            while (top != -1 && precedence(stack[top]) >= precedence(item)) {
                postfix[j++] = pop();
                postfix[j] = '\0';
            }
            push(item);
        }
    }

    char opStack[MAX] = "", operand_Stack[MAX] = "";
    if (top >= 0)
        strncpy(opStack, stack, top + 1);
    else
        strcpy(opStack, " ");
    strcpy(operand_Stack, postfix);
    printf("| %-13c | %-14s | %-14s |\n", item, opStack, operand_Stack);
}

while (top != -1) {
    postfix[j++] = pop();
    postfix[j] = '\0';

    char opStack[MAX] = "", operand_Stack[MAX] = "";
    if (top >= 0)
        strncpy(opStack, stack, top + 1);
    else
        strcpy(opStack, " ");
    strcpy(operand_Stack, postfix);
    printf("| %-13s | %-14s | %-14s |\n", " ", opStack, operand_Stack);
}

```

```

printf("+-----+-----+-----+\n");
}

// Infix to Prefix conversion
void infixToPrefix(char infix[], char prefix[]) {
    char temp[MAX], postfix[MAX] = {0};
    strcpy(temp, infix);
    reverse(temp);

    // Swap parentheses
    for (int i = 0; temp[i] != '\0'; i++) {
        if (temp[i] == '(')
            temp[i] = ')';
        else if (temp[i] == ')')
            temp[i] = '(';
    }

    printf("\nConversion for Prefix (in reverse):\n");
    infixToPostfix(temp, postfix);
    reverse(postfix);
    strcpy(prefix, postfix);
}

// Main driver
int main() {
    printf("\tCompiled by Sarfraj Alam\n");

    char infix[MAX], postfix[MAX] = {0}, prefix[MAX] = {0};

    printf("Enter the infix expression: ");
    scanf("%s", infix);

    while (1) {
        int choice;
        printf("\nChoose conversion:\n");
        printf("1. Infix to Postfix\n");
        printf("2. Infix to Prefix\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                top = -1;
                memset(postfix, 0, sizeof(postfix));
                infixToPostfix(infix, postfix);
                printf("\nPostfix expression: %s\n", postfix);
                break;

            case 2:
                top = -1;
                memset(prefix, 0, sizeof(prefix));
                infixToPrefix(infix, prefix);
                printf("\nPrefix expression: %s\n", prefix);
                break;
        }
    }
}

```

```

case 3:
    printf("Program exited.\n");
    exit(0);

default:
    printf("Invalid choice. Try again.\n");
}

return 0;
}

```

Output:

```

E:\Sarfraj\3rd SEME! × + ▾
Compiled by Sarfraj Alam
Enter the infix expression: A+B/C-D*(E+F)

Choose conversion:
1. Infix to Postfix
2. Infix to Prefix
3. Exit
Enter your choice: 1

+-----+-----+-----+
| Scanned Symbol | Operator Stack | Operand Stack |
+-----+-----+-----+
| A              |                | A             |
| +              | +             | A             |
| B              | +             | AB            |
| /              | +/            | AB            |
| C              | +/            | ABC           |
| -              | -             | ABC/+        |
| D              | -             | ABC/+D       |
| *              | -*            | ABC/+D       |
| (              | -* (          | ABC/+D       |
| E              | -* (          | ABC/+DE      |
| +              | -* (+        | ABC/+DE      |
| F              | -* (+        | ABC/+DEF     |
| )              | -*           | ABC/+DEF+    |
|                | -            | ABC/+DEF+*   |
|                |              | ABC/+DEF+*-  |
+-----+-----+-----+

Postfix expression: ABC/+DEF+*-

Choose conversion:
1. Infix to Postfix
2. Infix to Prefix
3. Exit
Enter your choice: 2

Conversion for Prefix (in reverse):

```

Enter your choice: 2

Conversion for Prefix (in reverse):

Scanned Symbol	Operator Stack	Operand Stack
((
F	(F
+	(+	F
E	(+	FE
)		FE+
*	*	FE+
D	*	FE+D
-	-	FE+D*
C	-	FE+D*C
/	-/	FE+D*C
B	-/	FE+D*CB
+	+	FE+D*CB/-
A	+	FE+D*CB/-A
		FE+D*CB/-A+

Prefix expression: +A-/BC*D+EF

Choose conversion:

1. Infix to Postfix
2. Infix to Prefix
3. Exit

Enter your choice: 3

Program exited.

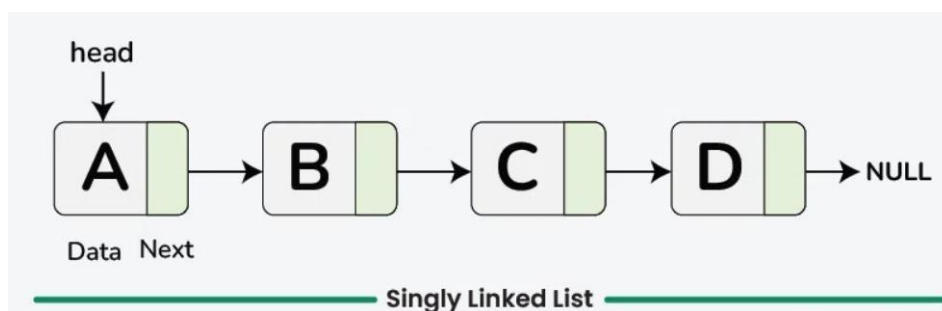
Process exited after 93.09 seconds with return value 0

Press any key to continue . . . |

Title: Write a menu-based program to insert a node at the beginning, at the end and at the specified position in a singly linked list.

Insertion in a linked list involves adding a new node at a specified position in the list. A singly linked list is a fundamental data structure, it consists of nodes where each node contains a data field and a reference to the next node in the linked list. The next of the last node is null, indicating the end of the list. Linked Lists support efficient insertion and deleting operation.

In a singly linked list, each node consists of two parts: data and a pointer to the next node. This structure allows nodes to be dynamically linked together, forming a chain-like sequence.



There are several types of insertion based on the position where the new node is to be added:

- At the beginning
- At the end
- At a specific position

Compiler: Dev C++

Language: C

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

// Define the node structure

typedef struct Node {

    int data;

    struct Node *next;

} Node;

// Function to create a new node

Node *createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed!\n");

        exit(1);

    }

    newNode->data = data;

    newNode->next = NULL;

    return newNode;

}

// Insert at the beginning

void insertAtBeginning(Node **head, int data) {

    Node *newNode = createNode(data);

    newNode->next = *head;

    *head = newNode;

}

// Insert at the end

void insertAtEnd(Node **head, int data) {

    Node *newNode = createNode(data);

    if (*head == NULL) {

        *head = newNode;
```



```

        return;
    }

    Node *current = *head;

    while (current->next != NULL) {

        current = current->next;
    }

    current->next = newNode;
}

// Insert at specific position
void insertAtPosition(Node **head, int data, int position) {

    if (position < 1) {

        printf("Invalid position! Position starts at 1.\n");

        return;
    }

    if (position == 1) {

        insertAtBeginning(head, data);

        return;
    }

    Node *newNode = createNode(data);

    Node *current = *head;

    for (int i = 1; current != NULL && i < position - 1; i++) {

        current = current->next;
    }

    if (current == NULL) {

        printf("Position exceeds list length!\n");

        free(newNode);

        return;
    }

    newNode->next = current->next;

```

```

        current->next = newNode;
    }

// Print the linked list
void printList(Node *head) {
    Node *current = head;

    printf("Linked List: ");

    while (current != NULL) {
        printf("%d -> ", current->data);

        current = current->next;
    }

    printf("NULL\n");
}

// Free the memory
void freeList(Node *head) {
    Node *temp;

    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

// Main function
int main() {

    printf("\tCompiled by Sarfraj Alam\n");

    Node *head = NULL;

    int choice, data, position;

    while (1) {

        printf("\n1. Insert at beginning\n");

        printf("2. Insert at end\n");

        printf("3. Insert at specific position\n");
    }
}

```

```
printf("4. Print the list\n");
```

```
printf("5. Exit\n");
```

```
printf("Enter your choice: ");
```

```
scanf("%d", &choice);
```

```
switch (choice) {
```

```
    case 1:
```

```
        printf("Enter data to insert at beginning: ");
```

```
        scanf("%d", &data);
```

```
        insertAtBeginning(&head, data);
```

```
        break;
```

```
    case 2:
```

```
        printf("Enter data to insert at end: ");
```

```
        scanf("%d", &data);
```

```
        insertAtEnd(&head, data);
```

```
        break;
```

```
    case 3:
```

```
        printf("Enter data: ");
```

```
        scanf("%d", &data);
```

```
        printf("Enter position: ");
```

```
        scanf("%d", &position);
```

```
        insertAtPosition(&head, data, position);
```

```
        break;
```

```
    case 4:
```

```
        printList(head);
```

```
        break;
```

```
    case 5:
```

```
        printf("Exiting...\n");
```

```
        freeList(head);
```

```
        exit(0);
```

default:

```
printf("Invalid choice! Please try again.\n");
```

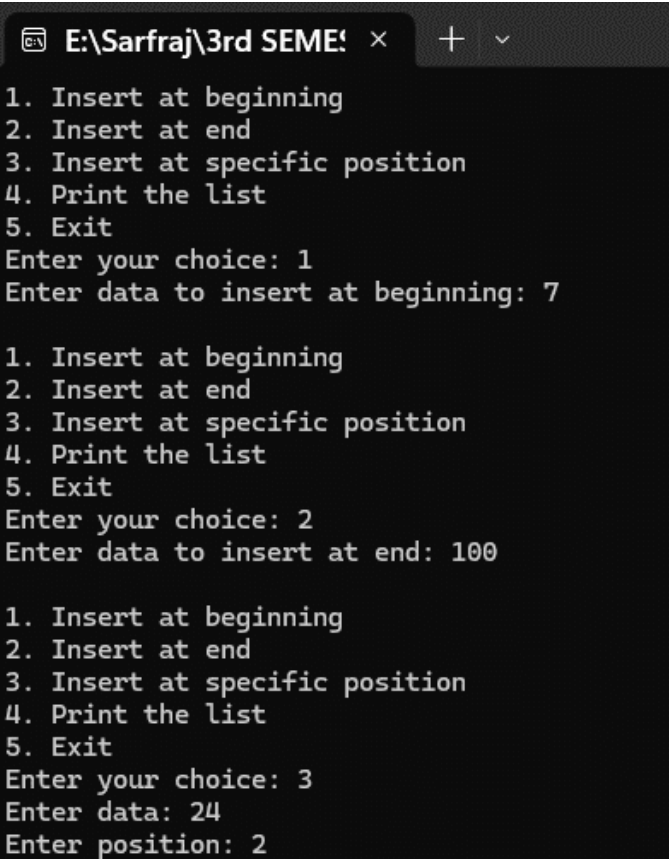
```
}
```

```
}
```

```
return 0;
```

```
}
```

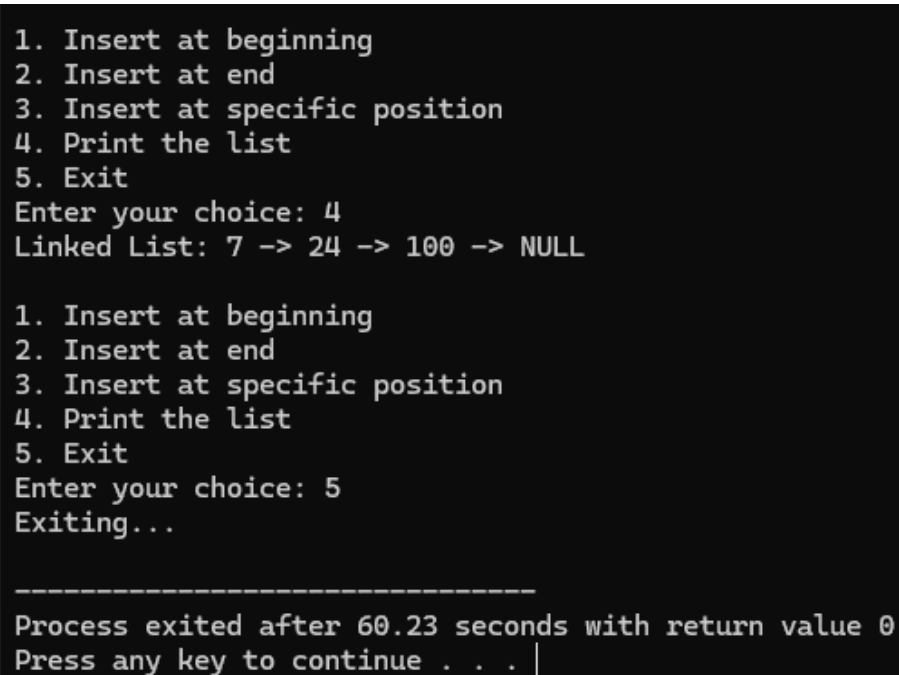
Output:



```
E:\Sarfraj\3rd SEME! x + v
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Print the list
5. Exit
Enter your choice: 1
Enter data to insert at beginning: 7

1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Print the list
5. Exit
Enter your choice: 2
Enter data to insert at end: 100

1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Print the list
5. Exit
Enter your choice: 3
Enter data: 24
Enter position: 2
```



```
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Print the list
5. Exit
Enter your choice: 4
Linked List: 7 -> 24 -> 100 -> NULL

1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Print the list
5. Exit
Enter your choice: 5
Exiting...

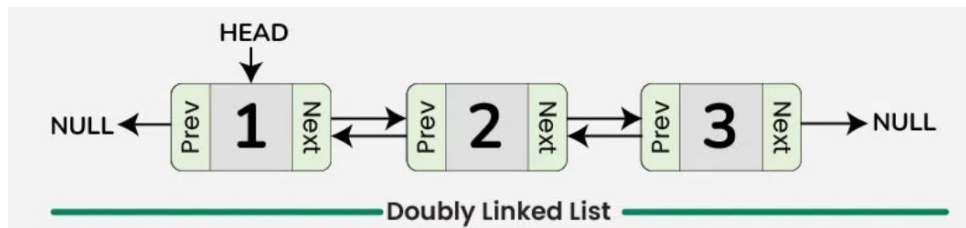
-----
Process exited after 60.23 seconds with return value 0
Press any key to continue . . . |
```

Title: Write a menu-based program to delete a node from the beginning, from the end and from the specified location in doubly linked lists.

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

- At the beginning
- At a specific position
- At the end

Deleting a node in a doubly linked list is very similar to deleting a node in a singly linked list. However, there is a little extra work required to maintain the links of both the previous and next nodes. In this article, we will learn about different ways to delete a node in a doubly linked list.



Compiler: Dev C++

Language: C

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

// Structure for a doubly linked list node
typedef struct Node {
    int data;
    struct Node *prev;
    struct Node *next;
} Node;

// Function to create a new node
Node *createNode(int data) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Function to insert at end (to build the list)
void insertAtEnd(Node **head, int data) {
    Node *newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }

    Node *current = *head;
    while (current->next != NULL) {
        current = current->next;
    }

    current->next = newNode;
```

```

newNode->prev = current;
}

// Function to delete from beginning
void deleteFromBeginning(Node **head) {
    if (*head == NULL) {
        printf("List is already empty!\n");
        return;
    }

    Node *temp = *head;
    *head = (*head)->next;
    if (*head != NULL) {
        (*head)->prev = NULL;
    }

    free(temp);
    printf("Deleted first node successfully.\n");
}

// Function to delete from end
void deleteFromEnd(Node **head) {
    if (*head == NULL) {
        printf("List is already empty!\n");
        return;
    }

    Node *current = *head;
    while (current->next != NULL) {
        current = current->next;
    }

    if (current->prev != NULL) {
        current->prev->next = NULL;
    } else {
        *head = NULL; // Only one node
    }

    free(current);
    printf("Deleted last node successfully.\n");
}

```

```
}
```

```
// Function to delete from specific position
```

```
void deleteFromPosition(Node **head, int position) {
```

```
    if (*head == NULL) {
```

```
        printf("List is empty!\n");
```

```
        return;
```

```
    }
```

```
    if (position < 1) {
```

```
        printf("Invalid position! Position starts at 1.\n");
```

```
        return;
```

```
    }
```

```
    if (position == 1) {
```

```
        deleteFromBeginning(head);
```

```
        return;
```

```
    }
```

```
    Node *current = *head;
```

```
    for (int i = 1; current != NULL && i < position; i++) {
```

```
        current = current->next;
```

```
    }
```

```
    if (current == NULL) {
```

```
        printf("Position exceeds list length!\n");
```

```
        return;
```

```
    }
```

```
    if (current->prev != NULL)
```

```
        current->prev->next = current->next;
```

```
    if (current->next != NULL)
```

```
        current->next->prev = current->prev;
```

```
    free(current);
```

```
    printf("Deleted node at position %d successfully.\n", position);
```

```
}
```

```
// Function to display the list
```



```

void displayList(Node *head) {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }

    printf("Doubly Linked List (forward): ");
    Node *current = head;
    while (current != NULL) {
        printf("%d <-> ", current->data);
        current = current->next;
    }
    printf("NULL\n");

    // Optional backward traversal
    printf("Doubly Linked List (backward): ");
    if (head != NULL) {
        current = head;
        while (current->next != NULL) {
            current = current->next;
        }

        while (current != NULL) {
            printf("%d <-> ", current->data);
            current = current->prev;
        }
        printf("NULL\n");
    }
}

// Function to free the list memory
void freeList(Node *head) {
    Node *temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {

```

```
Node *head = NULL;
```

```
int choice, position;
```

```
// Create a sample list
```

```
insertAtEnd(&head, 10);
```

```
insertAtEnd(&head, 20);
```

```
insertAtEnd(&head, 30);
```

```
insertAtEnd(&head, 40);
```

```
insertAtEnd(&head, 50);
```

```
printf("Initial list:\n");
```

```
displayList(head);
```

```
while (1) {
```

```
    printf("\nDoubly Linked List Operations:\n");
```

```
    printf("1. Delete from beginning\n");
```

```
    printf("2. Delete from end\n");
```

```
    printf("3. Delete from specific position\n");
```

```
    printf("4. Display list\n");
```

```
    printf("5. Exit\n");
```

```
    printf("Enter your choice: ");
```

```
    scanf("%d", &choice);
```

```
    switch (choice) {
```

```
        case 1:
```

```
            deleteFromBeginning(&head);
```

```
            break;
```

```
        case 2:
```

```
            deleteFromEnd(&head);
```

```
            break;
```

```
        case 3:
```

```
            printf("Enter position to delete: ");
```

```
            scanf("%d", &position);
```

```
            deleteFromPosition(&head, position);
```

```
            break;
```

```
        case 4:
```

```
            displayList(head);
```

```
            break;
```

```
        case 5:
```

```
            freeList(head);
```

```

        printf("Program exited. Memory freed.\n");

        return 0;

    default:

        printf("Invalid choice! Please try again.\n");

    }

}

return 0;

}

```

Output:

```

E:\Sarfraj\3rd SEME! x + v
Initial list:
Doubly Linked List (forward): 10 <--> 20 <--> 30 <--> 40 <--> 50 <--> NULL
Doubly Linked List (backward): 50 <--> 40 <--> 30 <--> 20 <--> 10 <--> NULL

Doubly Linked List Operations:
1. Delete from beginning
2. Delete from end
3. Delete from specific position
4. Display list
5. Exit
Enter your choice: 1
Deleted first node successfully.

Doubly Linked List Operations:
1. Delete from beginning
2. Delete from end
3. Delete from specific position
4. Display list
5. Exit
Enter your choice: 2
Deleted last node successfully.

Doubly Linked List Operations:
1. Delete from beginning
2. Delete from end
3. Delete from specific position
4. Display list
5. Exit
Enter your choice: 3
Enter position to delete: 3
Deleted node at position 3 successfully.

Doubly Linked List Operations:
1. Delete from beginning
2. Delete from end
3. Delete from specific position
4. Display list
5. Exit
Enter your choice: 4
Doubly Linked List (forward): 20 <--> 30 <--> NULL
Doubly Linked List (backward): 30 <--> 20 <--> NULL

Doubly Linked List Operations:
1. Delete from beginning
2. Delete from end
3. Delete from specific position
4. Display list
5. Exit
Enter your choice: 5
Program exited. Memory freed.

```

A Binary Search Tree is a binary tree where the value of any node is greater than the left subtree and less than the right subtree. It is a tree-based data structure in which each node has at most two children: a left child and a right child. The BST follows a specific property known as the Binary Search Tree Property, which states that for any node in the tree, the values in its left subtree are less than its value, and the values in its right subtree are greater than its value. This property enables efficient search, insertion, and deletion operations in a BST. It provides an efficient way to store, search, insert, and delete elements in a sorted manner. By following the binary search tree property, these operations can be performed with a time complexity of $O(\log n)$ on average in a balanced BST.

Properties of Binary Search Tree

- All nodes of the left subtree are less than the root node and nodes of the right subtree are greater than the root node.
- The In-order traversal of binary search trees gives the values in ascending order.
- All the subtrees of BST hold the same properties.

Operations of BST

- Insertion---It involves adding a new element to the tree while maintaining the binary search tree property.
- Deletion---It involves removing a specific element from the tree while maintaining the binary search tree property.
- Traversal---It is the process of visiting every node in the BST.
- Search---It involves finding a specific element within the tree.

Compiler: Dev C++

Language: C

Source Code:

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

typedef struct node {
    int key;
    struct node *left, *right;
} s_node;

// Create a new node
s_node *newNode(int item) {
    s_node *temp = (s_node *)malloc(sizeof(s_node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Inorder Traversal
void inorder(s_node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d -> ", root->key);
        inorder(root->right);
    }
}

// Insert a node
s_node *insert(s_node *node, int key) {
    if (node == NULL)
        return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    return node;
}

// Find the node with minimum key (inorder successor)
s_node *minValueNode(s_node *node) {
    s_node *current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

// Delete a node
s_node *deleteNode(s_node *root, int key) {
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
```

```

    root->right = deleteNode(root->right, key);
else {
    // Node with one or no child
    if (root->left == NULL) {
        s_node *temp = root->right;
        free(root);
        return temp;
    } else if (root->right == NULL) {
        s_node *temp = root->left;
        free(root);
        return temp;
    }

    // Node with two children: Get inorder successor
    s_node *temp = minValueNode(root->right);
    root->key = temp->key;
    root->right = deleteNode(root->right, temp->key);
}
return root;
}

// Search a node
s_node *search(s_node *root, int key) {
    if (root == NULL || root->key == key)
        return root;
    if (key < root->key)
        return search(root->left, key);
    return search(root->right, key);
}

// Free all nodes of the tree
void freeTree(s_node *root) {
    if (root) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}

// Main function
int main() {
    printf("\tCompiled by Sarfraj Alam\n");

    s_node *root = NULL;
    s_node *srh;
    int choice, key;

    while (1) {
        printf("\nEnter your choice\n1. Insertion\n2. Deletion\n3. Traversal (Inorder)\n4. Search\n5. Exit\n");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the key to be inserted: ");

```

```
scanf("%d", &key);  
root = insert(root, key);  
break;
```

case 2:

```
printf("Enter the key to be deleted: ");  
scanf("%d", &key);  
root = deleteNode(root, key);  
break;
```

case 3:

```
printf("Inorder Traversal: ");  
inorder(root);  
printf("NULL\n");  
break;
```

case 4:

```
printf("Enter the key to search: ");  
scanf("%d", &key);  
srh = search(root, key);  
printf(srh ? "Key found\n" : "Key not found\n");  
break;
```

case 5:

```
freeTree(root);  
printf("Exiting...\n");  
exit(0);
```

default:

```
printf("Wrong choice!! Try again...\n");
```

```
}  
}
```

```
return 0;  
}
```

Output:

```
E:\Sarfraj\3rd SEME! x + v E:\Sarfraj\3rd SEME! x + v
Compiled by Sarfraj Alam

Enter your choice
1. Insertion
2. Deletion
3. Traversal (Inorder)
4. Search
5. Exit
1
Enter the key to be inserted: 45

Enter your choice
1. Insertion
2. Deletion
3. Traversal (Inorder)
4. Search
5. Exit
1
Enter the key to be inserted: 78

Enter your choice
1. Insertion
2. Deletion
3. Traversal (Inorder)
4. Search
5. Exit
1
Enter the key to be inserted: 17

Enter your choice
1. Insertion
2. Deletion
3. Traversal (Inorder)
4. Search
5. Exit
1
Enter the key to be inserted: 7

Enter your choice
1. Insertion
2. Deletion

Enter your choice
1. Insertion
2. Deletion
3. Traversal (Inorder)
4. Search
5. Exit
2
Enter the key to be deleted: 78

Enter your choice
1. Insertion
2. Deletion
3. Traversal (Inorder)
4. Search
5. Exit
3
Inorder Traversal: 7 -> 17 -> 45 -> NULL

Enter your choice
1. Insertion
2. Deletion
3. Traversal (Inorder)
4. Search
5. Exit
4
Enter the key to search: 17
Key found

Enter your choice
1. Insertion
2. Deletion
3. Traversal (Inorder)
4. Search
5. Exit
5
Exiting...

-----
Process exited after 90.33 seconds with return value 0
Press any key to continue . . . |
```


Title: Write a program to calculate shortest path using Dijkstra's Algorithm

It is an algorithm to find the shortest path from single source to all other vertices. This algorithm uses the weights of the edges to find the path that minimizes the total distance (weight) between the source node and all other nodes.

Since the shortest path can be calculated from single source vertex to all the other vertices in the graph, Dijkstra's algorithm is also called single-source shortest path algorithm. The output obtained is called shortest path spanning tree.

Algorithm:

Step 1: Initialize: ○ Set the distance to the source node as 0.

- Set all other distances to infinity.
- Keep a priority queue (or min-heap) of all nodes with their current shortest distance.

Step 2: Process Nodes: ○ Pick the node with the smallest distance from the queue.

- For each neighbor, calculate the distance from the source through the current node. ○ If this new distance is smaller, update the neighbor's distance.
- Repeat until all nodes have been visited or the shortest path is found.

Step 3: Terminate when all nodes have been processed or the target node is reached (if searching a path to a specific node).

Time Complexity:

$O(E \cdot \log V)$,

where E is the number of edges and V is the number of vertices.

Compiler: Dev C++

Language: C

Source Code:

```
#include <stdio.h>

#include <stdbool.h>

#include <limits.h> // for INT_MAX


#define V 6 // number of vertices


// Function to find the vertex with minimum distance not yet included in shortest path tree
int miniDist(int distance[], bool Tset[]) {
    int minimum = INT_MAX, index = -1;

    for (int k = 0; k < V; k++) {
        if (!Tset[k] && distance[k] <= minimum) {
            minimum = distance[k];
            index = k;
        }
    }
    return index;
}


// Dijkstra's Algorithm using adjacency matrix
void DijkstraAlgo(int graph[V][V], int src) {
    int distance[V]; // Distance from source to each vertex
    bool Tset[V];    // True if vertex is included in shortest path tree

    // Initialize distances and Tset
    for (int k = 0; k < V; k++) {
        distance[k] = INT_MAX;
        Tset[k] = false;
    }

    distance[src] = 0; // Distance to source is 0
```

```

// Find shortest path for all vertices

for (int count = 0; count < V - 1; count++) {

    int m = miniDist(distance, Tset);

    if (m == -1)

        break; // No more reachable vertices

    Tset[m] = true;

    for (int j = 0; j < V; j++) {

        // Update distance[j] if m is connected and results in shorter path
        if (!Tset[j] && graph[m][j] && distance[m] != INT_MAX &&
            distance[m] + graph[m][j] < distance[j]) {
            distance[j] = distance[m] + graph[m][j];
        }
    }
}

// Print the results

printf("Vertex\t\tDistance from Source\n");

for (int k = 0; k < V; k++) {

    char vertex = 'A' + k;

    if (distance[k] == INT_MAX)

        printf("%c\t\t\t8\n", vertex);

    else

        printf("%c\t\t\t%d\n", vertex, distance[k]);

}

}

int main() {

    printf("\tCompiled by Sarfraj Alam\n");

    int graph[V][V] = {

        {0, 1, 2, 0, 0, 0},

```

```

        {1, 0, 0, 5, 1, 0},
        {2, 0, 0, 2, 3, 0},
        {0, 5, 2, 0, 2, 2},
        {0, 1, 3, 2, 0, 1},
        {0, 0, 0, 2, 1, 0}
    };

    DijkstraAlgo(graph, 0); // Starting from vertex A (index 0)

    return 0;
}

```

Output:

```

E:\Sarfraj\3rd SEME! x + v
Compiled by Sarfraj Alam
Vertex      Distance from Source
A           0
B           1
C           2
D           4
E           2
F           3

-----
Process exited after 0.04283 seconds with return value 0
Press any key to continue . . . |

```

Title: Write a program to calculate minimum spanning tree using Prim's algorithm

It is another algorithm to find the minimum spanning tree. It is a greedy algorithm that is used to find the minimum spanning tree for a particular weighted undirected graph. It finds the subset of the edges of a graph which can include each vertex and adds the value of the edges to minimize them. It starts with a single node and evaluates all adjacent nodes that connect the present edges in every evaluation stage.

Principle of Prim's Algorithm:

Prim's algorithm works by maintaining two sets of vertices:

1. Vertices included in the MST.
2. Vertices not yet included in the MST.

At each step, it selects the minimum weight edge that connects a vertex in the MST to a vertex outside the MST and adds it to the MST.

Time Complexity:

$O(V^2)$, If the input graph is represented using an adjacency list, then the time complexity of Prim's algorithm can be reduced to $O(E * \log V)$ with the help of a binary heap.

Applications of Prim's Algorithm

Following are some common applications of Prim's Algorithm:

- Network Design: Used in networking to minimize the costs in telecommunication and computer networks.
- Transportation: Used for planning road, highway, and railway networks with minimal total length.
- Computer Graphics: Used in image segmentation and 3D mesh generation in an optimized way.
- Game Development: Used for generating natural-looking mazes and terrains with minimal cost.
- VLSI Design: Used for reducing wiring lengths in very large-scale integration circuits.

Compiler: Dev C++

Language: C

Source Code:

```
#include <stdio.h>

#include <stdbool.h>

#include <string.h>


#define INF 9999999

#define V 5 // number of vertices


// Graph represented using adjacency matrix
int G[V][V] = {

    {0, 9, 75, 0, 0},

    {9, 0, 95, 19, 42},

    {75, 95, 0, 51, 66},

    {0, 19, 51, 0, 31},

    {0, 42, 66, 31, 0}

};


int main() {

    printf("\tCompiled by Sarfraj Alam\n");


    int no_edge = 0; // Number of edges in MST

    bool selected[V]; // Array to track selected vertices

    memset(selected, false, sizeof(selected));

    selected[0] = true; // Start from vertex 0


    int x, y; // To store edge endpoints


    printf("Edge\t: Weight\n");


    while (no_edge < V - 1) {

        int min = INF;
```

```

x = 0;

y = 0;


for (int i = 0; i < V; i++) {

    if (selected[i]) {

        for (int j = 0; j < V; j++) {

            if (!selected[j] && G[i][j]) { // Check for edge

                if (G[i][j] < min) {

                    min = G[i][j];

                    x = i;

                    y = j;

                }

            }

        }

    }

}

printf("%d -> %d\t: %d\n", x, y, G[x][y]);

selected[y] = true;

no_edge++;

}


return 0;

}

```

Output:

```

E:\Sarfraj\3rd SEME! × + ▾
Compiled by Sarfraj Alam
Edge      : Weight
0 -> 1    : 9
1 -> 3    : 19
3 -> 4    : 31
3 -> 2    : 51

-----
Process exited after 0.09343 seconds with return value 0
Press any key to continue . . .

```

Title: Write a program to search the user input key in the linear search.

Linear search, also known as sequential search, is a simple algorithm for finding an element within a list. It sequentially checks each element of the list until a match is found or the entire list has been searched. In Linear Search, we iterate over all the elements of the array and check if the current element is equal to the target element. If we find any element to be equal to the target element, then return the index of the current element. Otherwise, if no element is equal to the target element, then return -1 as the element is not found.

Time Complexity:

- Best Case: In the best case, the key might be present at the first index. So, the best-case complexity is $O(1)$
- Worst Case: In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is $O(N)$, where, N is the size of the list.
- Average Case: $O(N)$

Advantages of Linear Search Algorithm:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.
- It is a well-suited algorithm for small datasets.

Disadvantages of Linear Search Algorithm:

- Linear search has a time complexity of $O(N)$, which in turn makes it slow for large datasets.
- Not suitable for large arrays.

Compiler: Dev C++

Language: C

Source Code:

```
#include <stdio.h>

// Linear search function
int search(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x)
            return i; // return index of the found element
    }
    return -1; // return -1 if not found
}

int main() {
    printf("\tCompiled by Sarfraj Alam\n");
    printf("\t*****LINEAR SEARCH ALGORITHM*****\n\n");

    int n;
    printf("Enter size of array: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    int x;
    char choice;

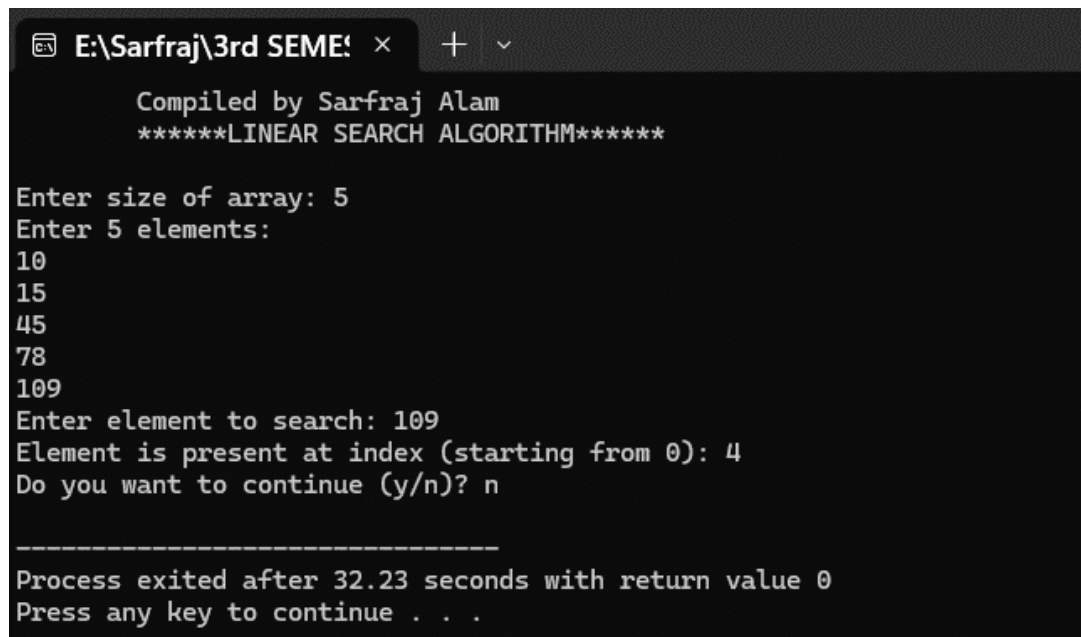
    do {
        printf("Enter element to search: ");
        scanf("%d", &x);

        int result = search(arr, n, x);
        if (result == -1)
            printf("Element is not present in the array.\n");
        else
            printf("Element is present at index (starting from 0): %d\n", result);

        printf("Do you want to continue (y/n)? ");
        scanf(" %c", &choice); // space before %c to catch newline character
    } while (choice == 'y' || choice == 'Y');
```

```
    return 0;  
}
```

Output:



```
E:\Sarfraj\3rd SEME! x + v  
Compiled by Sarfraj Alam  
*****LINEAR SEARCH ALGORITHM*****  
  
Enter size of array: 5  
Enter 5 elements:  
10  
15  
45  
78  
109  
Enter element to search: 109  
Element is present at index (starting from 0): 4  
Do you want to continue (y/n)? n  
  
-----  
Process exited after 32.23 seconds with return value 0  
Press any key to continue . . .
```

Title: Write a program to search the user input key in the list using binary search.

It is divided and conquer based algorithm where the search domain is divided into half after each recursive call. It calculates the midpoints and compare with the key. Thus, reducing half of the search domain. It uses recursive function. It is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log N)$.

Conditions to apply Binary Search Algorithm in a Data Structure

To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure should take constant time.

Algorithm

Below is the step-by-step algorithm for Binary Search:

Divide the search space into two halves by finding the middle index "mid".

- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
- If the key is smaller than the middle element, then the left side is used for next search.
- If the key is larger than the middle element, then the right side is used for next search. • This process is continued until the key is found or the total search space is exhausted.

Time Complexity

- Best Case: $O(1)$
- Average Case: $O(\log N)$
- Worst Case: $O(\log N)$

Compiler: Dev C++

Language: C

Source Code:

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

int number[50];

// Binary Search Function
int binarySearch(int A[], int l, int r, int key) {
    if (l <= r) {
        int mid = (l + r) / 2;
        if (A[mid] == key)
            return mid;
        else if (A[mid] < key)
            return binarySearch(A, mid + 1, r, key);
        else
            return binarySearch(A, l, mid - 1, key);
    }
    return -1;
}

int main() {
    printf("\tCompiled by Sarfraj Alam\n");
    printf("\t*****BINARY SEARCH ALGORITHM*****\n\n");

    int total;
    printf("Enter the total number of elements (max 50): ");
    scanf("%d", &total);

    if (total > 50 || total <= 0) {
        printf("Invalid number of elements.\n");
        return 1;
    }

    printf("\n\nEnter %d elements in ascending order:\n", total);
    for (int i = 0; i < total; i++) {
        scanf("%d", &number[i]);
    }

    char choice;
    do {
        int temp;
        printf("\nEnter the number to search in the array: ");
        scanf("%d", &temp);

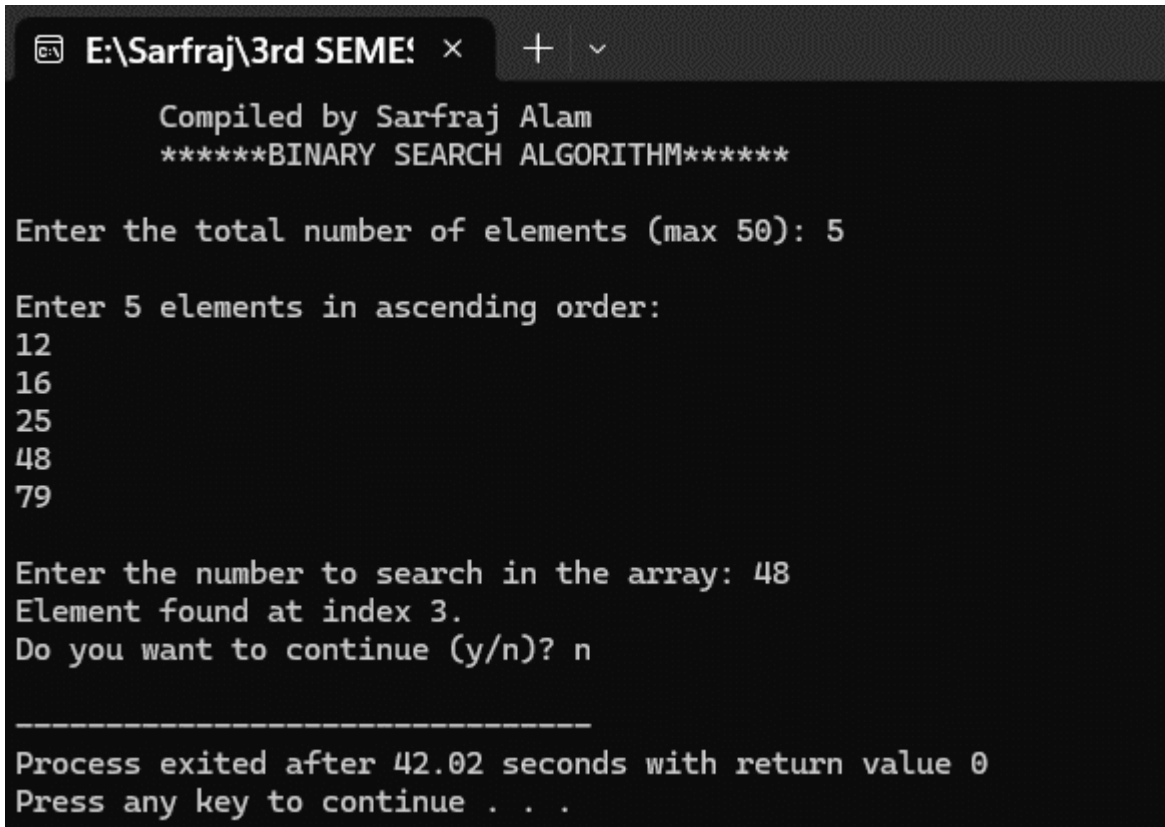
        int index = binarySearch(number, 0, total - 1, temp);
        if (index != -1)
            printf("Element found at index %d.\n", index);
        else
            printf("Element not found in the array.\n");
    } while (choice != 'q');
```

```
printf("Do you want to continue (y/n)? ");
scanf(" %c", &choice); // space before %c to consume newline

} while (choice == 'y' || choice == 'Y');

return 0;
}
```

Output:



```
E:\Sarfraj\3rd SEME! x + v
Compiled by Sarfraj Alam
*****BINARY SEARCH ALGORITHM*****

Enter the total number of elements (max 50): 5

Enter 5 elements in ascending order:
12
16
25
48
79

Enter the number to search in the array: 48
Element found at index 3.
Do you want to continue (y/n)? n

-----
Process exited after 42.02 seconds with return value 0
Press any key to continue . . .
```

Title: Write a program to implement double hashing and quadratic hashing in C.

Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access. Hashing involves mapping data to a specific index in a hash table (an array of items) using a hash function. It enables fast retrieval of information based on its key.

Double Hashing

Double hashing is a collision resolution technique used in hash tables. It works by using two hash functions to compute two different hash values for a given key. The first hash function is used to compute the initial hash value, and the second hash function is used to compute the step size for the probing sequence. Double hashing has the ability to have a low collision rate, as it uses two hash functions to compute the hash value and the step size. This means that the probability of a collision occurring is lower than in other collision resolution techniques such as linear probing or quadratic probing.

Quadratic probing

Quadratic probing is an open-addressing scheme where we look for the i^2 'th slot in the i 'th iteration if the given hash value x collides in the hash table. We have already discussed linear probing implementation.

How Quadratic Probing is done?

Let $\text{hash}(x)$ be the slot index computed using the hash function.

- If the slot $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1*1) \% S$.
- If $(\text{hash}(x) + 1*1) \% S$ is also full, then we try $(\text{hash}(x) + 2*2) \% S$.
- If $(\text{hash}(x) + 2*2) \% S$ is also full, then we try $(\text{hash}(x) + 3*3) \% S$.
- This process is repeated for all the values of i until an empty slot is found.

Compiler: Dev C++

Language: C

Source Code:

```
#include <stdio.h>

#include <stdlib.h>


#define TABLE_SIZE 10

#define DOUBLE_HASHING 1

#define QUADRATIC_PROBING 2


int hash_table[TABLE_SIZE] = {0};


// Hash functions

int hash1(int key) {
    return key % TABLE_SIZE;
}

int hash2(int key) {
    return 7 - (key % 7);
}


// Insertion function

void insert(int method) {
    int key, index, i;

    printf("\nEnter a value to insert into the hash table: ");
    scanf("%d", &key);

    index = hash1(key);
    if (hash_table[index] == 0) {
        hash_table[index] = key;
        return;
    }

    if (method == DOUBLE_HASHING) {
        int increment = hash2(key);
        for (i = 1; i < TABLE_SIZE; i++) {
```

```

        index = (hash1(key) + i * increment) % TABLE_SIZE;

        if (hash_table[index] == 0) {
            hash_table[index] = key;
            return;
        }
    }
} else if (method == QUADRATIC_PROBING) {
    for (i = 1; i < TABLE_SIZE; i++) {
        index = (hash1(key) + i * i) % TABLE_SIZE;
        if (hash_table[index] == 0) {
            hash_table[index] = key;
            return;
        }
    }
}

printf("\nElement cannot be inserted. Table might be full.\n");
}

```

// Search function

```

void search(int method) {
    int key, index, i;

    printf("\nEnter element to search: ");
    scanf("%d", &key);

    index = hash1(key);
    if (hash_table[index] == key) {
        printf("Value is found at index %d\n", index);
        return;
    }

    if (method == DOUBLE_HASHING) {
        int increment = hash2(key);
        for (i = 1; i < TABLE_SIZE; i++) {
            index = (hash1(key) + i * increment) % TABLE_SIZE;

```



```

        if (hash_table[index] == key) {

            printf("Value is found at index %d\n", index);

            return;

        }

    }

} else if (method == QUADRATIC_PROBING) {

    for (i = 1; i < TABLE_SIZE; i++) {

        index = (hash1(key) + i * i) % TABLE_SIZE;

        if (hash_table[index] == key) {

            printf("Value is found at index %d\n", index);

            return;

        }

    }

}

printf("Value is not found in the hash table.\n");

}

```

// Display function

```

void display() {

    printf("\n Hash Table\n");

    printf("-----\n");

    printf(" | Index | Value | \n");

    printf("-----\n");

    for (int i = 0; i < TABLE_SIZE; i++) {

        printf(" | %2d | %3d | \n", i, hash_table[i]);

    }

    printf("-----\n");

}

```

```

int main() {

    printf("\tCompiled by Sarfraj Alam\n");

    int choice, method = DOUBLE_HASHING;

    while (1) {

```

```
printf("\nChoose Hashing Method:\n");  
  
printf("1. Double Hashing\n");  
  
printf("2. Quadratic Probing\n");  
  
printf("3. Continue with current method\n");  
  
printf("4. Exit\n");  
  
printf("Enter choice: ");  
  
int modeChoice;  
  
scanf("%d", &modeChoice);
```

```
if (modeChoice == 1) {  
    method = DOUBLE_HASHING;  
} else if (modeChoice == 2) {  
    method = QUADRATIC_PROBING;  
} else if (modeChoice == 4) {  
    exit(0);  
}
```

```
printf("\n--- Operations Menu ---\n");  
  
printf("1. Insert\n");  
  
printf("2. Display\n");  
  
printf("3. Search\n");  
  
printf("4. Exit\n");  
  
printf("Enter your choice: ");  
  
scanf("%d", &choice);
```

```
switch (choice) {  
    case 1:  
        insert(method);  
        break;  
    case 2:  
        display();  
        break;  
    case 3:  
        search(method);  
        break;
```

```

        case 4:

            exit(0);

        default:

            printf("Invalid choice. Try again!\n");

    }

}

return 0;

}

```

Output:

E:\Sarfraj\3rd SEME! × + ▾

Compiled by Sarfraj Alam

```

Choose Hashing Method:
1. Double Hashing
2. Quadratic Probing
3. Continue with current method
4. Exit
Enter choice: 1

--- Operations Menu ---
1. Insert
2. Display
3. Search
4. Exit
Enter your choice: 1

Enter a value to insert into the hash table: 48

Choose Hashing Method:
1. Double Hashing
2. Quadratic Probing
3. Continue with current method
4. Exit
Enter choice: 1

--- Operations Menu ---
1. Insert
2. Display
3. Search
4. Exit
Enter your choice: 1

Enter a value to insert into the hash table: 58

Choose Hashing Method:
1. Double Hashing
2. Quadratic Probing
3. Continue with current method
4. Exit
Enter choice: 2

--- Operations Menu ---
1. Insert
2. Display
3. Search
4. Exit
Enter your choice: 1

```

E:\Sarfraj\3rd SEME! × + ▾

```

4. Exit
Enter your choice: 1

Enter a value to insert into the hash table: 68

Choose Hashing Method:
1. Double Hashing
2. Quadratic Probing
3. Continue with current method
4. Exit
Enter choice: 3

--- Operations Menu ---
1. Insert
2. Display
3. Search
4. Exit
Enter your choice: 2

Hash Table
-----
| Index | Value |
-----
| 0      | 0      |
| 1      | 0      |
| 2      | 0      |
| 3      | 58     |
| 4      | 0      |
| 5      | 0      |
| 6      | 0      |
| 7      | 0      |
| 8      | 48     |
| 9      | 68     |
-----

Choose Hashing Method:
1. Double Hashing
2. Quadratic Probing
3. Continue with current method
4. Exit
Enter choice: 4

-----
Process exited after 62.08 seconds with return value 0
Press any key to continue . . .

```

Title: Write a program to sort the user input data ascending or descending order using bubble sort.

Bubble Sort is a comparison based simple sorting algorithm that works by comparing the adjacent elements and swapping them if the elements are not in the correct order. It is an in-place and stable sorting algorithm that can sort items in data structures such as arrays and linked lists. It is an in-place and stable sorting algorithm (i.e. the relative order of the elements remains the same after sorting) that can sort items in data structures such as arrays and linked lists. It performs $n-1$ passes of the array/linked list and in each pass the largest unsorted element is moved to its correct position. It repeatedly compares adjacent elements and swaps them if they are in the wrong order. The algorithm iterates through the list multiple times until no more swaps are needed, resulting in a sorted sequence. During each iteration, the largest unsorted element "bubbles" up to its correct position, hence the name "Bubble Sort."

How Bubble Sort Works

1. Pass-Through the List: In each iteration, the algorithm starts from the beginning of the array and compares each pair of adjacent elements.
2. Swapping Elements: If the current element exceeds the next element, they are swapped. This way, the largest element in the unsorted portion "bubbles up" to its correct position at the end of the list.
3. Multiple Iterations: The process is repeated for the remaining unsorted portion of the list. After
 - i. each iteration, one less element (the last element) must be considered as it's already sorted.
4. Completion: The algorithm stops when no more swaps are needed during a full pass through the
 - i. array, meaning the list is sorted.

Time Complexity: $O(n^2)$

Compiler: Dev C++

Language: C

Source Code:

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

```
// Bubble Sort Function
```

```
void bubbleSort(int A[], int n, int choice) {
```

```
    for (int i = 0; i < n - 1; i++) {
```

```
        printf("Pass %d:\n", i + 1);
```

```
        for (int k = 0; k < n; k++) {
```

```
            printf("%d\t", A[k]); // Show array before sorting this pass
```

```
        }
```

```
        printf("\n");
```

```
        // Bubble sort logic
```

```
        for (int j = 0; j < n - i - 1; j++) {
```

```
            if ((choice == 1 && A[j] > A[j + 1]) || (choice == 2 && A[j] < A[j + 1])) {
```

```
                int temp = A[j];
```

```
                A[j] = A[j + 1];
```

```
                A[j + 1] = temp;
```

```
            }
```

```
        // Show intermediate array after each comparison and possible swap
```

```
        for (int k = 0; k < n; k++) {
```

```
            printf("%d\t", A[k]);
```

```
        }
```

```
        printf("\n");
```

```
    }
```

```
    printf("\n");
```

```
}
```

```
printf("\nThe sorted data is:\n");

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

    printf("%d\t", A[i]);

}

printf("\n");

}

int main() {

    printf("\t*****Compiled by Jonash Chataut*****\n");

    printf("\t*****BUBBLE SORT ALGORITHM*****\n\n");


    int terms;

    printf("Enter the number of terms to be sorted: ");

    scanf("%d", &terms);


    int numbers[terms];

    printf("Enter the numbers to be sorted:\n");

    for (int i = 0; i < terms; i++) {

        scanf("%d", &numbers[i]);

    }


    int choice;

    printf("\nChoose sorting order:");

    printf("\n1. Ascending");

    printf("\n2. Descending\n");

    scanf("%d", &choice);


    if (choice == 1 || choice == 2)

        bubbleSort(numbers, terms, choice);
```

```
else
```

```
printf("\nInvalid option selected.\n");
```

```
return 0;
```

```
}
```

Output:

```
E:\Sarfraj\3rd SEME! x + v
*****Compiled by Jonash Chataut*****
*****BUBBLE SORT ALGORITHM*****

Enter the number of terms to be sorted: 5
Enter the numbers to be sorted:
14
2
78
4
56

Choose sorting order:
1. Ascending
2. Descending
1
Pass 1:
14      2      78      4      56
2       14     78      4      56
2       14     78      4      56
2       14     4       78     56
2       14     4       56     78

Pass 2:
2       14     4       56     78
2       14     4       56     78
2       4      14     56     78
2       4      14     56     78

Pass 3:
2       4      14     56     78
2       4      14     56     78
2       4      14     56     78

Pass 4:
2       4      14     56     78
2       4      14     56     78

The sorted data is:
2       4      14     56     78

-----
Process exited after 27.93 seconds with return value 0
Press any key to continue . . .
```

Title: Write a program to sort the user input data ascending or descending order using selection sort.

Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

1. First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and swap it with the second element.
3. We keep doing this until we get all elements moved to correct position.

Advantages of Selection Sort

- Easy to understand and implement, making it ideal for teaching basic sorting concepts.
- Requires only a constant $O(1)$ extra memory space.
- It requires less number of swaps (or memory writes) compared to many other standard algorithms. Only [cycle sort](#) beats it in terms of memory writes. Therefore it can be simple algorithm choice when memory writes are costly.

Disadvantages of the Selection Sort

- Selection sort has a time complexity of $O(n^2)$ makes it slower compared to algorithms like [Quick Sort](#) or [Merge Sort](#).
- Does not maintain the relative order of equal elements which means it is not stable.

Time Complexity: $O(n^2)$, as there are two nested loops:

- One loop to select an element of Array one by one = $O(n)$
- Another loop to compare that element with every other Array element = $O(n)$

Therefore overall complexity = $O(n) * O(n) = O(n*n) = O(n^2)$

Compiler: Dev C++

Language: C

Source Code:

```
#include <stdio.h>

void selectionSort(int A[], int n, int choice) {

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

        int min_index = i; // Assume first element is min/max

        printf("Pass %d:\n", i + 1);

        // Print current state of array before sorting this pass

        for (int k = 0; k < n; k++) {

            printf("%d\t", A[k]);

        }

        printf("\n");

        // Find the min or max index based on choice

        for (int j = i + 1; j < n; j++) {

            if ((choice == 1 && A[j] < A[min_index]) || (choice == 2 && A[j] > A[min_index])) {

                min_index = j;

            }

        }

        // Swap if a new min/max was found

        if (min_index != i) {

            int temp = A[min_index];

            A[min_index] = A[i];

            A[i] = temp;

        }

    }

    // Print the final sorted array

    printf("\nThe sorted data is:\n");

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

        printf("%d\t", A[i]);

    }

}
```

```
printf("\n");  
  
}  
  
int main() {  
  
    printf("\tCompiled by Sarfraj Alam\n");  
  
    printf("\t*****SELECTION SORT*****\n\n");  
  
    int terms;  
  
    printf("Enter the number of terms to be sorted: ");  
  
    scanf("%d", &terms);  
  
    int numbers[terms];  
  
    printf("Enter the numbers to be sorted: \n");  
  
    for (int i = 0; i < terms; i++) {  
        scanf("%d", &numbers[i]);  
    }  
  
    int choice;  
  
    printf("\nChoose sorting order:\n");  
  
    printf("1. Ascending\n");  
  
    printf("2. Descending\n");  
  
    printf("Enter your choice: ");  
  
    scanf("%d", &choice);  
  
    printf("\n");  
  
    if (choice == 1 || choice == 2)  
        selectionSort(numbers, terms, choice);  
  
    else  
        printf("Invalid option selected.\n");  
  
    return 0;  
}
```

Output:

```
E:\Sarfraj\3rd SEME! x + v
Compiled by Sarfraj Alam
*****SELECTION SORT*****

Enter the number of terms to be sorted: 5
Enter the numbers to be sorted:
14
48
1
56
89

Choose sorting order:
1. Ascending
2. Descending
Enter your choice: 2

Pass 1:
14    48    1    56    89
Pass 2:
89    48    1    56    14
Pass 3:
89    56    1    48    14
Pass 4:
89    56    48    1    14

The sorted data is:
89    56    48    14    1

-----
Process exited after 20.24 seconds with return value 0
```

Title: Write a program to sort the user input data ascending or descending order using insertion sort.

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

- We start with the second element of the array as the first element is assumed to be sorted.
- Compare the second element with the first element if the second element is smaller then swap them. • Move to the third element, compare it with the first two elements, and put it in its correct position
- Repeat until the entire array is sorted.

Advantages

- Simple and easy to implement.
- Stable sorting algorithm.
- Efficient for small lists and nearly sorted lists.Space-efficient as it is an in-place algorithm.

Disadvantages

- Inefficient for large lists.
- Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.

Time Complexity

- Best case: $O(n)$, If the list is already sorted, where n is the number of elements in the list.
- Average case: $O(n^2)$, If the list is randomly ordered
- Worst case: $O(n^2)$, If the list is in reverse order

Compiler: Dev C++

Language: C

Source Code:

```
#include <stdio.h>

void insertionSort(int arr[], int n) {

for (int i = 1; i < n; i++) {

    int key = arr[i];

    int j = i - 1;


    // Display the array before insertion

    printf("Array before inserting %d: ", key);

    for (int k = 0; k < n; k++) {

        printf("%d ", arr[k]);

    }

    printf("\n");


    // Shift elements greater than key to the right

    while (j >= 0 && arr[j] > key) {

        printf("Shifting %d to the right\n", arr[j]);

        arr[j + 1] = arr[j];

        j--;

    }


    // Insert the key at correct position

    arr[j + 1] = key;


    // Display the array after insertion

    printf("Array after inserting %d: ", key);

    for (int k = 0; k < n; k++) {

        printf("%d ", arr[k]);

    }

    printf("\n\n");

}

}
```

```
int main() {

    printf("\tCompiled by Sarfraj Alam\n");

    printf("\t*****INSERTION SORT*****\n\n");

    int arr_size;

    printf("Enter the size of the array: ");

    scanf("%d", &arr_size);


    int arr[arr_size];

    printf("Enter the elements of the array:\n");

    for (int i = 0; i < arr_size; i++) {

        scanf("%d", &arr[i]);

    }


    printf("\nGiven array is:\n");

    for (int i = 0; i < arr_size; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n\n");


    // Sort the array using insertion sort

    insertionSort(arr, arr_size);


    printf("Sorted array is:\n");

    for (int i = 0; i < arr_size; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

    return 0;

}
```

Output:

```
E:\Sarfraj\3rd SEME! x + v
Compiled by Sarfraj Alam
*****INSERTION SORT*****

Enter the size of the array: 5
Enter the elements of the array:
48
78
1
56
25

Given array is:
48 78 1 56 25

Array before inserting 78: 48 78 1 56 25
Array after inserting 78: 48 78 1 56 25

Array before inserting 1: 48 78 1 56 25
Shifting 78 to the right
Shifting 48 to the right
Array after inserting 1: 1 48 78 56 25

Array before inserting 56: 1 48 78 56 25
Shifting 78 to the right
Array after inserting 56: 1 48 56 78 25

Array before inserting 25: 1 48 56 78 25
Shifting 78 to the right
Shifting 56 to the right
Shifting 48 to the right
Array after inserting 25: 1 25 48 56 78

Sorted array is:
1 25 48 56 78

-----
Process exited after 27.66 seconds with return value 0
```