

1. ALGORITHM: Traversing a Linear Array.

Let LA is linear array with lower bound LB and upper bound UB. Algorithm traverses LA applying an operation PROCESS to each element of LA.

1. Set $K := LB$
2. Repeat Steps 3 and 4 while $K \leq UB$
3. Apply PROCESS to $LA[K]$.
4. Set $K := K + 1$.
5. Exit.

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

```
void process(int element) {
```

```
    printf("Processing element: %d\n", element);
```

```
}
```

```
int main() {
```

```
    int LA[] = {1, 2, 3, 4, 5}; // Example array
```

```
    int LB = 0;                // Lower Bound (indexing starts from 0)
```

```
    int UB = 4;                // Upper Bound (last index of the array)
```

```
    int K = LB;                // Step 1: Set  $K := LB$ 
```

```
    while (K <= UB) {          // Step 2: Repeat Steps 3 and 4 while  $K \leq UB$ 
```

```
        process(LA[K]);        // Step 3: Apply PROCESS to  $LA[K]$ 
```

```
        K++;                   // Step 4: Set  $K := K + 1$ 
```

```
    }
```

```
    return 0; // Step 5: Exit
```

```
}
```

2.ALGORITHM: INSERT (LA, N, K, ITEM)

Here LA is linear array with N elements and K is positive integer such that $K \leq N$. This algorithm inserts an element ITEM into Kth position in LA.

1. Set $J := N$.
2. Repeat Steps 3 and 4 while $J \geq K$.
3. Set $LA[J + 1] := LA[J]$.
4. Set $J := J - 1$.
5. Set $LA[K] := ITEM$.
6. Set $N := N + 1$.
7. Exit

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

```
int main() {  
    int LA[1000] = {1, 2, 3, 4, 5}; // Example array  
    int N = 5;                       // Initial number of elements in the array  
    int ITEM = 99;                   // Element to be inserted  
    int K = 2;                       // Position at which to insert ITEM (0-based index)  
    int J = N;                       // Step 1: Set J := N  
    while (J >= K) {                 // Step 2: Repeat Steps 3 and 4 while J >= K  
        LA[J + 1] = LA[J];           // Step 3: Set LA[J + 1] := LA[J]  
        J--;                          // Step 4: Set J := J - 1  
    }  
    LA[K] = ITEM;                    // Step 5: Set LA[K] := ITEM  
    N = N + 1;                       // Step 6: Set N := N + 1  
    // Print the updated array to verify the insertion  
    for (int i = 0; i < N; i++) {  
        printf("%d ", LA[i]);  
    }  
    printf("\n");  
    return 0;}
```

3.ALGORITHM: PUSH (STACK, TOP, MAXSTK, ITEM)

This procedure pushes an ITEM onto a Stack.

1. If TOP = MAXSTK, then: Print OVERFLOW, and Return.
2. Set TOP := TOP + 1.
3. Set STACK [TOP] := TEM.
4. Return.

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

```
#define MAXSTK 1000      // Define the maximum size of the stack
```

```
int STACK[MAXSTK];      // Array to hold the stack items
```

```
int TOP = 0;            // Initialize TOP to 0, representing an empty stack
```

```
void push(int ITEM) {    // Function to push an item onto the stack
```

```
    if (TOP == MAXSTK) {
```

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

```
        return;
```

```
    }
```

```
    STACK[TOP] = ITEM;
```

```
    TOP = TOP + 1;
```

```
}
```

```
int main() {
```

```
    // Example usage of push function
```

```
    push(10);
```

```
    push(20);
```

```
    push(30);
```

```
    push(40);
```

```
    for (int i = 0; i < TOP; i++) {
```

```
        printf("%d ", STACK[i]);
```

```
    }
```

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

```
    return 0;}
```

4. ALGORITHM: POP (STACK, TOP, ITEM)

This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. If TOP = 0, then: Print UNDERFLOW, and Return.
2. Set ITEM :=STACK [TOP].
3. Set TOP :=TOP - 1.
4. Return.

```
#include <stdio.h>

#define MAXSTK 100    // Define the maximum size of the stack

int STACK[MAXSTK];    // Array to hold the stack items

int TOP = 0;          // Initialize TOP to 0, representing an empty stack


// Function to push an item onto the stack
void push(int ITEM) {
    if (TOP == MAXSTK) {
        printf("OVERFLOW\n");
        return;
    }
    STACK[TOP] = ITEM;
    TOP = TOP + 1;
}

// Function to pop an item from the stack
int pop() {
    if (TOP == 0) {
        printf("UNDERFLOW\n");
        return -1;      // Assuming -1 indicates underflow
    }
    TOP = TOP - 1;
    return STACK[TOP];
}
```

```
}
```

```
int main() {  
    // Example usage of push and pop functions  
    push(10);  
    push(20);  
    push(30);  
    // Print stack contents before popping  
    printf("Stack before popping: ");  
    for (int i = 0; i < TOP; i++) {  
        printf("%d ", STACK[i]);  
    }  
    printf("\n");  
  
    int poppedItem = pop(); // Pop an item  
    if (poppedItem != -1) {  
        printf("Popped item: %d\n", poppedItem);  
    }  
    // Print stack contents after popping  
    printf("Stack after popping: ");  
    for (int i = 0; i < TOP; i++) {  
        printf("%d ", STACK[i]);  
    }  
    printf("\n");  
    return 0;  
}
```

5.ALGORITHM: QINSERT (QUEUE, N, FRONT, REAR, ITEM)

This procedure inserts an element ITEM into a queue.

1. If FRONT = 1 and REAR = N, or if FRONT = REAR + 1 then:

Write: OVERFLOW, and Return.

2. If FRONT = NULL, then:

Set FRONT :=1 and REAR:=1.

Else if REAR =N, then:

Set REAR:=1.

Else:

Set REAR:=REAR +1.

3. Set QUEUE [REAR]:= ITEM.

4. Return.

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

```
#define N 1000          // Define the maximum size of the circular queue
```

```
int QUEUE[N];          // Array to hold the circular queue elements
```

```
int FRONT = -1, REAR = -1; // Initialize FRONT and REAR to -1, representing an empty queue
```

```
// Function to insert an element into the circular queue
```

```
void enqueue(int ITEM) {
```

```
    if ((FRONT == 0 && REAR == N - 1) || (FRONT == REAR + 1)) {
```

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

```
        return;
```

```
    }
```

```
    if (FRONT == -1) {
```

```
        FRONT = 0;
```

```
        REAR = 0;
```

```
    } else if (REAR == N - 1) {
```

```
        REAR = 0;
```

```

    }
else {
    REAR++;
}
QUEUE[REAR] = ITEM;
}

int main() {
    // Example usage of enqueue function
    enqueue(10);
    enqueue(20);
    enqueue(30);

    // Print queue contents
    printf("Queue contents: ");
    if (FRONT <= REAR) {
        for (int i = FRONT; i <= REAR; i++) {
            printf("%d ", QUEUE[i]);
        }
    } else {
        for (int i = FRONT; i < N; i++) {
            printf("%d ", QUEUE[i]);
        }
        for (int i = 0; i <= REAR; i++) {
            printf("%d ", QUEUE[i]);
        }
    }
    printf("\n");
    return 0;}

```

6.AGORITHM: QDELETE (QUEUE, N, FRONT, REAR, ITEM)

This procedure deletes an element from a queue and assign it

to the variable ITEM.

1. If FRONT =NULL, then:

Write: UNDERFLOW, and Return.

2. Set ITEM := QUEUE [FRONT].

3. If FRONT = REAR, then:

Set FRONT :=NULL and REAR := NULL.

Else if FRONT =N, then:

Set FRONT:= 1.

Else: Set FRONT := FRONT +1.

4. Return.

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

```
#define N 1000 // Define the maximum size of the queue
```

```
int QUEUE[N]; // Array to hold the queue elements
```

```
int FRONT = -1, REAR = -1; // Initialize FRONT and REAR to -1, representing an empty queue
```

```
// Function to delete an element from the queue and return it
```

```
int qdelete() {
```

```
    if (FRONT == -1) {
```

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

```
        return -1; // Return -1 to indicate underflow
```

```
    }
```

```
    int item = QUEUE[FRONT];
```

```
    if (FRONT == REAR) {
```

```
        FRONT = -1;
```

```
        REAR = -1;
```

```
    }
```

```
    else if (FRONT == N - 1) {
```

```
        FRONT = 0;
    } else {
        FRONT = FRONT + 1;
    }
    return item;
}
```

```
int main() {
    // Example usage of qdelete function
    int deletedItem = qdelete();
    if (deletedItem != -1) {
        printf("Deleted item: %d\n", deletedItem);
    }

    // Insert some elements into the queue
    REAR = 0; // Setting REAR to non-empty position
    FRONT = 0;
    QUEUE[REAR] = 10;
    QUEUE[++REAR] = 20;
    QUEUE[++REAR] = 30;

    // Example usage of qdelete function after insertion
    deletedItem = qdelete();
    if (deletedItem != -1) {
        printf("Deleted item: %d\n", deletedItem);
    }

    // Print queue contents
```

```
printf("Queue contents after deletion: ");  
if (FRONT == -1) {  
    printf("EMPTY");  
} else if (FRONT <= REAR) {  
    for (int i = FRONT; i <= REAR; i++) {  
        printf("%d ", QUEUE[i]);  
    }  
} else {  
    for (int i = FRONT; i < N; i++) {  
        printf("%d ", QUEUE[i]);  
    }  
    for (int i = 0; i <= REAR; i++) {  
        printf("%d ", QUEUE[i]);  
    }  
}  
printf("\n");  
  
return 0;  
}
```

7.Algorithm: (Deleting from a Linear Array) DELETE(LA, N, K, ITEM)

Here LA is a linear array with N elements and K'is a positive integer such that

$K \leq N$. This algorithm deletes the Kth element from LA.

1. Set ITEM := LA[K].

2. Repeat for J = K to N -1:

[Move J + 1st element upward.] Set LA[J] := LA[J + 1].

[End of loop.].

3. [Reset the number N'of elements in LA.] Set N: = N - 1.

4. Exit.

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

```
// Function to delete the Kth element from LA
```

```
void deleteElement(int LA[], int* N, int K) {
```

```
    if (K < 1 || K > *N) {
```

```
        printf("Invalid position. Cannot delete.\n");
```

```
        return;
```

```
    }
```

```
    int ITEM = LA[K - 1]; // Save the Kth element
```

```
    // Move elements from K+1 to N one position upward
```

```
    for (int J = K; J < *N; ++J) {
```

```
        LA[J - 1] = LA[J];
```

```
    }
```

```
    // Reduce the number of elements in LA
```

```
    (*N)--;
```

```
    printf("Deleted element: %d\n", ITEM);
```

```
}
```

```
int main() {  
    int LA[] = {10, 20, 30, 40, 50}; // Example linear array  
    int N = 5; // Number of elements in LA  
    int K; // Position of the element to delete  
  
    printf("Enter the position (1 to %d) to delete: ", N);  
    scanf("%d", &K);  
  
    deleteElement(LA, &N, K);  
  
    printf("Updated array after deletion:\n");  
    for (int i = 0; i < N; ++i) {  
        printf("%d ", LA[i]);  
    }  
    printf("\n");  
  
    return 0;  
}
```

8.Algorithm : (Linear Search) LINEAR(DATA, N, ITEM, LOC)

Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC := 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set DATA[N + 1] := ITEM.

2. [Initialize counter.] Set LOC := 1.

3. [Search for ITEM.]

Repeat while DATA[LOC] != ITEM:

Set LOC := LOC + 1.

[End of loop.]-

4. [Successful?] If LOC = N + 1, then: Set LOC := 0,

5. Exit.

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

```
// Function to perform linear search
```

```
int linearSearch(int DATA[], int N, int ITEM) {
```

```
    DATA[N] = ITEM; // Insert ITEM at the end of DATA
```

```
    int LOC = 1; // Initialize counter
```

```
    while (DATA[LOC] != ITEM) {
```

```
        LOC++; // Search for ITEM
```

```
    }
```

```
    if (LOC == N + 1) {
```

```
        LOC = 0; // Unsuccessful search
```

```
    }
```

```
    return LOC;
```

```
}
```

```
int main() {  
    int N;  
    printf("Enter the number of elements in the array: ");  
    scanf("%d", &N);  
  
    int DATA[N + 1]; // Extra space for ITEM  
    printf("Enter %d elements: ", N);  
    for (int i = 1; i <= N; ++i) {  
        scanf("%d", &DATA[i]);  
    }  
  
    int ITEM;  
    printf("Enter the item to search: ");  
    scanf("%d", &ITEM);  
  
    int LOC = linearSearch(DATA, N, ITEM);  
  
    if (LOC == 0) {  
        printf("Item %d not found.\n", ITEM);  
    } else {  
        printf("Item %d found at location %d.\n", ITEM, LOC);  
    }  
  
    return 0;  
}
```

9.Algorithm : (Binary Search) BINARY(DATA, LB, UB, ITEM, LOC)

Here DATA is a sorted array with lower bound LB and upper bound UB, and

ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]

Set $BEG := LB$, $END := UB$ and $MID = \text{INT}((BEG + END)/2)$.

2. Repeat Steps 3 and 4 while $BEG \leq END$ and $DATA[MID] \neq ITEM$.

3.If $ITEM < DATA[MID]$, then:

Set $END := MID - 1$.

Else:

Set $BEG := MID + 1$.

[End of If structure.]

4.Set $MID := \text{INT}((BEG + END)/2)$.

[End of Step 2 loop.]

5. If $DATA[MID] = ITEM$, then;

Set $LOC := MID$.

Else:

Set $LOC := \text{NULL}$.

[End of If structure.]

6. Exit.

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

```
// Function to perform binary search
```

```
int binarySearch(int DATA[], int LB, int UB, int ITEM) {
```

```
    int BEG = LB;
```

```
    int END = UB;
```

```
    int MID;
```

```

while (BEG <= END) {
    MID = (BEG + END) / 2;

    if (DATA[MID] == ITEM) {
        return MID; // Item found
    } else if (ITEM < DATA[MID]) {
        END = MID - 1; // Search left half
    } else {
        BEG = MID + 1; // Search right half
    }
}

return -1; // Item not found
}

int main() {
    int N;

    printf("Enter the number of elements in the sorted array: ");
    scanf("%d", &N);

    int DATA[N];

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

    int ITEM;

    printf("Enter the item to search: ");

```

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

int LOC = binarySearch(DATA, 0, N - 1, ITEM);

if (LOC != -1) {
    printf("Item %d found at location %d.\n", ITEM, LOC);
} else {
    printf("Item %d not found.\n", ITEM);
}

return 0;
}
```

10. Algorithm: A text T and patterns P and Q are in memory. This algorithm replaces every occurrence of P in T by Q.

1. [Find index of P.] Set $K := \text{INDEX}(T, P)$.

2. Repeat while $K \neq 0$:

(a) [Replace P by Q.] Set $T := \text{REPLACE}(T, P, Q)$.

(b) [Update index.] Set $K := \text{INDEX}(T, P)$.

[End of loop:]

3. Write: T.

4. Exit.

1. Write a program which reads a line of text in array TEXT and then replace every occurrence of the given word stored in PREV with the new word stored in NEW.

Sample Input: TEXT="This is my first C program", PREV="is", NEW="are"

Sample Output: "This are my first C program"

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

```
int main() {
```

```
    string text;
```

```
    string prev;
```

```
    string NEW;
```

```
    cout << "Insert the main Text: ";
```

```
    getline(cin, text);
```

```
    cout << "Insert the text which has to be replaced: ";
```

```
    getline(cin, prev);
```

```
    cout << "Insert the new text: ";
```

```
    getline(cin, NEW);
```

```
    int r = text.size();
```

```

int s = prev.size();

int max = r - s + 1;

int found, match = 0;


for (int i = 0; i < max; i++) {
    found = 1;
    for (int j = 0; j < s; j++) {
        if (text.substr(i, s) == prev) {
            continue;
        } else {
            found = 0;
        }
    }
    if (found == 1) {
        text.replace(i, s, NEW);
        match = 1;
    }
}


if (match == 0) {
    cout << "No matching found." << endl;
} else {
    cout << "Modified string: " << text << endl;
}


return 0;
}

```

11.Algorithm:

Write a procedure which counts the number NUM of times the word "the" appears in the short story S. (We do not count "the" in "mother," and we assume no sentence ends with the word "the.")

Procedure P3.15 : COUNT(LINE, N, NUM)

1. Set WORD := 'THE' and NUM := 0.

2. [Prepare for the three cases.]

Set BEG := WORD//'D', END := 'D'//WORD and

MID := 'D'//WORD//'D'

3. Repeat Steps 4 through 6 for K = 1 to N:

4.[First case.] If SUBSTRING(LINE[K], 1, 4) = BEG, then:

Set NUM := NUM +1;

5.[Second case.] If SUBSTRING(LINE[K], 77, 4) = END, then:

Set NUM := NUM +1.

6.[General case.] Repeat for J = 2 to 76.

If SUBSTRING(LINE[K], J, 5) = MID, then:

Set NUM := NUM + 1.

[End of If structure.]

[End of Step 6 loop.]

[End of Step 3 loop.]

7. Return.

2. Write a program which reads a line of text in array TEXT and counts the number of occurrence of the given WORD in TEXT.

Sample Input: TEXT="This is my first C program", WORD="is"

Sample Output: 2

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
int main() {
```

```
    string text;
```

```
    string prev;
```

```
    cout << "Insert the main Text: ";
```

```
    getline(cin, text);
```

```
    cout << "Insert the text which has to be replaced: ";
```

```
    getline(cin, prev);
```

```
    int r = text.size();
```

```
    int s = prev.size();
```

```
    int max = r - s + 1;
```

```
    int count = 0, match = 0, found;
```

```
    for (int i = 0; i < max; i++) {
```

```
        found = 1;
```

```
        for (int j = 0; j < s; j++) {
```

```
            if (text.substr(i, s) == prev) {
```

```
                continue;
```

```
            } else {
```

```
                found = 0;
```

```
            }
```

```
        }
```

```
        if (found == 1) {
```

```
            count++;
```

```
            match = 1;
```

```
        }
```

```
}
```

```
if (match == 0) {
```

```
    cout << "No matching found." << endl;
```

```
} else {
```

```
    cout << "Total occurrences: " << count << endl;
```

```
}
```

```
return 0;
```

```
}
```

12. Algorithm to Find the Adjacency Matrix:

1. Initialize an $M \times M$ matrix (adjacency matrix) with all elements set to 0.
2. For each ordered pair (u, v) representing an edge:
 - Set `adj[u][v] = 1` (indicating an edge from node u to node v).
 - If the graph is undirected, also set `adj[v][u] = 1`.
3. The resulting adjacency matrix represents the graph G .

```
#include <iostream>

#include <vector>

using namespace std;

const int MAX_NODES = 100; // Adjust as needed

int main() {
    int M, N; // Number of nodes and edges
    cout << "Enter the number of nodes (M): ";
    cin >> M;
    cout << "Enter the number of edges (N): ";
    cin >> N;

    vector<vector<int>> adj(M, vector<int>(M, 0)); // Initialize adjacency matrix

    cout << "Enter the edges (u v) one by one:\n";
    for (int i = 0; i < N; ++i) {
        int u, v;
        cin >> u >> v;
        adj[u][v] = 1;
        adj[v][u] = 1; // For undirected graph
    }
```

```
// Print the adjacency matrix
cout << "Adjacency Matrix:\n";
for (int i = 0; i < M; ++i) {
    for (int j = 0; j < M; ++j) {
        cout << adj[i][j] << " ";
    }
    cout << endl;
}

return 0;
}
```

13. Algorithm to Find the Location of an Item in a BST:

1. ****Input****:

- Initialize the parallel arrays:
 - `key[]`: Contains the keys (values) of the nodes.
 - `left[]`: Contains indices of the left child nodes.
 - `right[]`: Contains indices of the right child nodes.
- Read the item to be searched (`item`).

2. Initialize variables:

- `index`: Start with the root node (index 0).

3. ****Search Loop****:

- While `index` is not NULL_INDEX:
 - If `key[index]` equals `item`, return `index` (item found).
 - Otherwise:
 - If `item` is less than `key[index]`, set `index` to `left[index]`.
 - Otherwise, set `index` to `right[index]`.

4. ****Output****:

- If the loop terminates without finding the item (i.e., `index` is NULL_INDEX), return NULL_INDEX (item not found).

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

```
#define SIZE 9
```

```
#define NULL_INDEX -1
```

```
// Parallel arrays to store the BST
```

```
int key[SIZE] = {8, 3, 10, 1, 6, 14, 4, 7, 13};  
int left[SIZE] = {1, 3, NULL_INDEX, NULL_INDEX, 6, 8, NULL_INDEX, NULL_INDEX, NULL_INDEX};  
int right[SIZE] = {2, 4, 5, NULL_INDEX, 7, NULL_INDEX, NULL_INDEX, NULL_INDEX, NULL_INDEX};
```

```
// Function to find the location of a given item
```

```
int findLocation(int item) {  
    int index = 0; // Start with the root node  
    while (index != NULL_INDEX) {  
        if (key[index] == item) {  
            return index; // Item found, return its index  
        } else if (item < key[index]) {  
            index = left[index]; // Move to the left child  
        } else {  
            index = right[index]; // Move to the right child  
        }  
    }  
    return NULL_INDEX; // Item not found  
}
```

```
int main() {  
    int item = 6;  
    int location = findLocation(item);  
    if (location != NULL_INDEX) {  
        printf("Item %d found at index: %d\n", item, location);  
    } else {  
        printf("Item %d not found in the BST.\n", item);  
    }  
    return 0;  
}
```

14. Algorithm for Preorder Traversal of Binary Tree (Parallel Arrays):

1. ****Input****:

- Initialize the parallel arrays:
- ``key[]``: Contains the keys (values) of the nodes.
- ``left[]``: Contains indices of the left child nodes.
- ``right[]``: Contains indices of the right child nodes.

2. ****Preorder Traversal Function****:

- Define a function ``preorderTraversal(index)`` that takes an index as an argument.
- If the index is not `NULL_INDEX`:
- Visit the node by printing its key (``key[index]``).
- Recursively call ``preorderTraversal(left[index])`` to traverse the left subtree.
- Recursively call ``preorderTraversal(right[index])`` to traverse the right subtree.

3. ****Main Program****:

- In the ``main()`` function:
- Print "Preorder traversal of the tree: ".
- Call ``preorderTraversal(0)`` to start the traversal from the root node.
- Print a newline.

4. ****Output****:

- The program will print the keys of the nodes in preorder traversal order.

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

```
#define SIZE 9
```

```
#define NULL_INDEX -1
```

```

// Parallel arrays to store the binary tree
int key[SIZE] = {8, 3, 10, 1, 6, 14, 4, 7, 13};
int left[SIZE] = {1, 3, NULL_INDEX, NULL_INDEX, 6, 8, NULL_INDEX, NULL_INDEX, NULL_INDEX};
int right[SIZE] = {2, 4, 5, NULL_INDEX, 7, NULL_INDEX, NULL_INDEX, NULL_INDEX, NULL_INDEX};

// Function to observe preorder traversal of the binary tree
void preorderTraversal(int index) {
    if (index != NULL_INDEX) {
        printf("%d ", key[index]); // Visit node
        preorderTraversal(left[index]); // Traverse left subtree
        preorderTraversal(right[index]); // Traverse right subtree
    }
}

int main() {
    printf("Preorder traversal of the tree: ");
    preorderTraversal(0); // Start traversal from root node
    printf("\n");
    return 0;
}

```

15. ALGORITHM: INSERT END (INFO, LINK, START, AVAIL, ITEM)

This algorithm inserts ITEM as the END node in the List.

1. If AVAIL = NULL, then: Write OVERFLOW and Exit.
2. Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM.
4. Set PTR := START.
5. Repeat step 6 while LINK[PTR] != NULL
Set PTR := LINK[PTR].
6. Set LINK[PTR] := NEW and LINK[NEW] := NULL.
7. Exit.

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Define maximum size of the linked list

int LINK[MAX];

int INFO[MAX];

int AVAIL;

int START = -1; // Assume START is initially NULL (-1 in our case)

// Function to initialize the available list

void initializeAvail() {

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

        LINK[i] = i + 1;

    }

    LINK[MAX - 1] = -1; // End of the available list

    AVAIL = 0;

}

// Function to add an item to the linked list
```

```
void addItem(int ITEM) {  
    if (AVAIL == -1) { // If AVAIL is NULL  
        printf("OVERFLOW\n");  
        return;  
    }
```

```
    // Step 2  
    int newNode = AVAIL;  
    AVAIL = LINK[AVAIL];
```

```
    // Step 3  
    INFO[newNode] = ITEM;
```

```
    // Step 4  
    int PTR = START;
```

```
    // Step 5  
    if (PTR == -1) { // If START is NULL  
        START = newNode;  
        LINK[newNode] = -1;  
    } else {  
        while (LINK[PTR] != -1) {  
            PTR = LINK[PTR];  
        }
```

```
        // Step 6  
        LINK[PTR] = newNode;  
        LINK[newNode] = -1;  
    }
```

```
    // Step 7
```

```
        return;
    }

// Function to print the linked list
void printList() {
    int PTR = START;
    while (PTR != -1) {
        printf("%d -> ", INFO[PTR]);
        PTR = LINK[PTR];
    }
    printf("NULL\n");
}

int main() {
    initializeAvail();

    addItem(10);
    addItem(20);
    addItem(30);

    printList();

    return 0;
}
```

16. (Warshall's Algorithm)

A directed graph G with M nodes is maintained in memory by its adjacency matrix A . This algorithm finds the (Boolean) path matrix P of the graph G .

1. Repeat for $I, J = 1, 2, \dots, M$: [Initializes P .]

If $A[I, J] = 0$, then: Set $P[I, J] := 0$;

Else: Set $P[I, J] := 1$.

[End of loop.]

2. Repeat Steps 3 and 4 for $K = 1, 2, \dots, M$: [Updates P .]

3. Repeat Step 4 for $I = 1, 2, \dots, M$:

4. Repeat for $J = 1, 2, \dots, M$:

Set $P[I, J] := P[I, J] \vee (P[I, K] \wedge P[K, J])$.

[End of loop.]

[End of Step 3 loop.]

[End of Step 2 loop.]

5. Exit.

```
#include <iostream>
```

```
using namespace std;
```

```
const int MAX_NODES = 100; // Adjust as needed
```

```
void warshallAlgorithm(int A[][MAX_NODES], int M) {
```

```
    int P[MAX_NODES][MAX_NODES];
```

```
    // Initialize P with the same values as A
```

```
    for (int i = 0; i < M; ++i) {
```

```
        for (int j = 0; j < M; ++j) {
```

```
        P[i][j] = A[i][j];
    }
}
```

```
// Warshall's Algorithm
for (int k = 0; k < M; ++k) {
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < M; ++j) {
            P[i][j] = P[i][j] || (P[i][k] && P[k][j]);
        }
    }
}
```

```
// Print the path matrix P
cout << "Path Matrix (Boolean matrix):\n";
for (int i = 0; i < M; ++i) {
    for (int j = 0; j < M; ++j) {
        cout << P[i][j] << " ";
    }
    cout << endl;
}
}
```

```
int main() {
    int M; // Number of nodes
    cout << "Enter the number of nodes (M): ";
    cin >> M;

    int A[MAX_NODES][MAX_NODES]; // Adjacency matrix
```

```
cout << "Enter the adjacency matrix (0 or 1):\n";  
for (int i = 0; i < M; ++i) {  
    for (int j = 0; j < M; ++j) {  
        cin >> A[i][j];  
    }  
}  
  
warshallAlgorithm(A, M);  
  
return 0;  
}
```

17. QUICK(A, N, BEG, END, LOC)

Here A is an array with N elements. Parameters BEG and END contain the boundary values of the sublist of A to which this procedure applies. LOC keeps track of the position of the first element A[BEG] of the sublist during the procedure. The local variables LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned.

1. [Initialize.] Set $LEFT := BEG$, $RIGHT := END$ and $LOC := BEG$.

2. [Scan from right to left.]

(a) Repeat while $A[LOC] \leq A[RIGHT]$ and $LOC \neq RIGHT$:

$RIGHT = RIGHT - 1$.

[End of loop.]

(b) If $LOC = RIGHT$, then: Return.

(c) If $A[LOC] > A[RIGHT]$, then:

(i) [Interchange $A[LOC]$ and $A[RIGHT]$.]

$TEMP := A[LOC]$, $A[LOC] := A[RIGHT]$,

$A[RIGHT] := TEMP$.

(ii) Set $LOC := RIGHT$.

(iii) Go to Step.3.

[End of If structure.]

3. [Scan from left to right.]

(a) Repeat while $A[LEFT] \leq A[LOC]$ and $LEFT \neq LOC$:

$LEFT := LEFT + 1$.

[End of loop.]

(b) If $LOC = LEFT$, then: Return.

(c) If $A[LEFT] > A[LOC]$, then

(i) [Interchange $A[LEFT]$ and $A[LOC]$.]

$TEMP := A[LOC]$, $A[LOC] := A[LEFT]$.

$A[LEFT] := TEMP$.

(ii) Set LOC := LEFT.

(iii) Go to Step. 2.

[End of.If structure.]

```
#include <iostream>
```

```
#include <vector>
```

```
// Partition function for quicksort
```

```
int partition(std::vector<int>& arr, int beg, int end) {
```

```
    int pivot = arr[end]; // Choose the last element as the pivot
```

```
    int i = beg - 1; // Index of smaller element
```

```
    for (int j = beg; j < end; ++j) {
```

```
        if (arr[j] <= pivot) {
```

```
            // Swap arr[i+1] and arr[j]
```

```
            std::swap(arr[++i], arr[j]);
```

```
        }
```

```
    }
```

```
    // Swap arr[i+1] and arr[end] (move pivot to its correct position)
```

```
    std::swap(arr[i + 1], arr[end]);
```

```
    return i + 1; // Return the index of the pivot
```

```
}
```

```
// QUICK procedure
```

```
void QUICK(std::vector<int>& arr, int N, int BEG, int END, int& LOC) {
```

```
    int LEFT = BEG;
```

```
    int RIGHT = END;
```

```
    LOC = BEG;
```

```
// Scan from right to left
while (arr[LOC] <= arr[RIGHT] && LOC != RIGHT) {
    RIGHT--;
}
```

```
if (LOC == RIGHT) {
    return; // Already sorted
}
```

```
if (arr[LOC] > arr[RIGHT]) {
    // Interchange A[LOC] and A[RIGHT]
    std::swap(arr[LOC], arr[RIGHT]);
    LOC = RIGHT;
}
```

```
// Scan from left to right
while (arr[LEFT] <= arr[LOC] && LEFT != LOC) {
    LEFT++;
}
```

```
if (LOC == LEFT) {
    return; // Already sorted
}
```

```
if (arr[LEFT] > arr[LOC]) {
    // Interchange A[LEFT] and A[LOC]
    std::swap(arr[LEFT], arr[LOC]);
    LOC = LEFT;
}
```

```

    }

    // Recurse on left and right subarrays
    if (BEG < LOC - 1) {
        QUICK(arr, N, BEG, LOC - 1, LOC);
    }
    if (LOC + 1 < END) {
        QUICK(arr, N, LOC + 1, END, LOC);
    }
}

int main() {
    int n;

    std::cout << "Enter the number of elements: ";
    std::cin >> n;

    std::vector<int> arr(n);
    std::cout << "Enter " << n << " integers: ";
    for (int i = 0; i < n; ++i) {
        std::cin >> arr[i];
    }

    int loc;
    QUICK(arr, n, 0, n - 1, loc);

    std::cout << "Sorted array in ascending order:\n";
    for (int num : arr) {
        std::cout << num << " ";
    }
}

```

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

```
return 0;
```

```
}
```