Table of Contents

1. ARRAY IMPLEMENTATION IN C 2

1.1. Display Operation 5

1.2. Insert Operations 6

1.3. Delete Operations 10

1.4. Search Operation 14

1.5. Sorting Operation 15

1.6. Reversing Operation 17

2. STACK IMPLEMENTATION IN C 18

2.1. Display Operation 21

2.2. Check Operations 22

2.3. Peek Operation 24

2.4. Push Operation 25

2.5. Pop Operation 27

3. SIMPLE LINK LIST IMPLEMENTATION IN C 31

3.1. Display Operation 36

3.2. Insert Operations 37

3.3. Delete Operations 45

3.4. Search Operations 50

3.5. Sorting Operation 53

3.6. Reversing Operation 53

1. ARRAY IMPLEMENTATION IN C

Arrays can be implemented in C in two ways;

1) The static array which means once created, its size can't be changed.
2) The pointer array created using pointer notation. We will be using the pointer array because it allows the array to be dynamic (change size) unlike the static normal array.

➢ Array Operations:

- Display Array
- Insert element into Array
- Delete element from Array
- Search element in Array
- Sort Array
- Reverse Array

➢ Function Headings:

```
void printArray(int array[], int size){⋯}
void insertFront(int *array, int *size, int element){⋯}
void insertEnd(int *array, int *size, int element){⋯}
void insertPos(int *array, int *size, int index, int element){⋯}
void deleteFront(int *array, int *size){⋯}
void deleteEnd(int *array, int *size){⋯}
void deletePos(int *array, int *size, int index){⋯}
int searchElem(int array[], int size, int element){⋯}
void sortArray(int *array, int size, int order){⋯}
int *reverseArray(int *array, int size){⋯}
```

Here are the function headings for the different operations. We will see more about them later.

➢ Array declaration (creation):

```c
    int *myArray;   // creating a dynamic array using pointer notation
    printf("How many items do you want to initialise your array with: ");
    scanf("%d",&size);
//  allocating memory to myArray.
    myArray = (int *)malloc(sizeof(int) * size);
    printf("Enter array elements:\n");
    for (i = 0; i<size; i++){
        printf("Elem %d:\t",i+1);
        scanf("%d",&myArray[i]);  // getting each element of the array
    }
    printf("Array successfully created: ");
    printArray(myArray, size);    // function call to print array
```

- This code creates and pointer array named “myArray” for storing the address of integers, gets the array size from the user and allocates memory of that size to the pointer using the malloc function.
- The user then initializes the array by entering a number of elements which is equal to the size and then prints the array (see output far below)
- Now, our array has being created and we can start carrying out operations on it.

```
How many items do you want to initialise your array with: 5
Enter array elements:
Elem 1: 45
Elem 2: 8
Elem 3: 0
Elem 4: 9
Elem 5: 3
```

➢ Options:

- A do while loop is used to carry out the different options shown above as long as the user wants to.
- A switch case control structure is used to perform the operations according to the option the user chooses.
- After carrying out an operation, the user is asked to continue. If the key pressed is ‘y’ or ‘Y’ (for yes), the loop repeats again, giving the user to carry out the same or another operation. The loop exits and programs exist if the user presses any other key.

```c
    do{   // array operations
    printf("\nOPERATIONS\n============\n\n");
    printf("1. Display array\n\n");
    printf("INSERTION OPERATIONS\n");
    printf("2. Insert element at beginning\n");
    printf("3. Insert element at end\n");
    printf("4. Insert element at given index\n\n");
    printf("DELETION OPERATIONS\n");
    printf("5. Delete first element\n");
    printf("6. Delete last element\n");
    printf("7. Delete element at given index\n\n");
    printf("8. Search an element\n");
    printf("9. Sort array\n");
    printf("10.Reverse array\n\n");
    printf("Select an option: ");
    scanf("%d", &option);
    switch(option){
```

```c
  printf("\nDo you want to continue? (y or n): ");
  choice = getchar();   // getting user's choice to continue
  system("cls");    // clears the screen . use sytem("clear"), sytem("cls) or cl
    } while(choice == 'y' || choice == 'Y');  // loop continues if the choice is
    return 0;
}
```

```
OPERATIONS
============

1. Display array

INSERTION OPERATIONS
2. Insert element at beginning
3. Insert element at end
4. Insert element at given index

DELETION OPERATIONS
5. Delete first element
6. Delete last element
7. Delete element at given index

8. Search an element
9. Sort array
10.Reverse array

Select an option:
```

1.1. DISPLAY OPERATION

➢ **Function Definition**

```c
void printArray(int array[], int size){
  if (size == 0){
    printf("Empty Array\n");
    return;
  }
  int i;
  printf("|");
  for (i = 0; i < size; i++){
    printf(" %d |", array[i]);  // printing array elements
  }
  printf("\tSize:\t%d\n",size);
}
```

- The function prints "Empty Array" if the size is 0. That is if the array is empty. Else it prints its values in a row with each value separated by a horizontal bar.
- At the end of the array, its size is displayed.
- The display function is called after every operation including after initialization in order to confirm if the operations are successful. For example, after initialization, the array is;

```
How many items do you want to initialise your array with: 5
Enter array elements:
Elem 1: 45
Elem 2: 8
Elem 3: 0
Elem 4: 9
Elem 5: 3
Array successfully created: | 45 | 8 | 0 | 9 | 3 |      Size:   5
```

1.2. INSERT OPERATIONS.

1.2.1.Inserting at the beginning of the array

➢ **Function Definition**

```
22 void insertFront(int *array, int *size, int element){
23   int i;
24   if (*size == 0){  // assigning element to array if it is empty
25     *array = element;
26     (*size)++;
27     return;
28   }
29   (*size)++;    // incrementing the size by one
30   array = (int *)realloc(array, sizeof(int) * (*size)); // reallocating increased
31   for (i = (*size) - 1; i > 0; i--){
32     array[i] = array[i-1];      // shifting elements
33   }
34   array[0] = element;
35 }
```

- If the array's size is empty, the element to be inserted is assigned to the array. Since the array is actually a pointer, this stores the element in the first location of the array. The size is then incremented (line 24 – 28).
- When the array already contains elements, the size is first incremented, then the incremented size is used to re-allocate (increase) memory of the array (line 29 and 30). So now one empty space is left at the back of the array.
- The for loop (line 31 – 33) is use to shift the elements one cell to the back (since the last cell is empty after reallocation). The element is inserted at the first position (index 0).

➢ **Function Call**

```
        case 2: printf("Enter data to be inserted: ");
            scanf("%d", &data);
            insertFront(myArray, &size, data);  // insertFront function call
            printf("Array: ");
            printArray(myArray, size);
            break;
```

To insert at the front, select option 2.

➢ Console Output

```
Select an option: 2
Enter data to be inserted: 70
Array: | 70 | 45 | 8 | 0 | 9 | 3 |      Size:   6

Do you want to continue? (y or n):
```

70 has been inserted at the beginning and other elements shifted one cell to the back. New size is 5.

1.2.2.Inserting at the end of the array

➢ Function Definition

```c
36 void insertEnd(int *array, int *size, int element){
37   if (*size == 0){
38     *array = element;
39     (*size)++;
40     return;
41   }
42   (*size)++;
43   array = (int *)realloc(array, sizeof(int) * (*size)); // reallocating increase
44   if (*size > 0){
45     array[(*size)-1] = element;
46   }
47   else  array[0] = element;
48 }
```

- The size is incremented and memory reallocated to the array as for inserting at the head. But this time, the element is inserted at the [new] last position of the array (line 47).

➢ Function call

```c
      case 3: printf("Enter data to be inserted: ");
            scanf("%d", &data);
            insertEnd(myArray, &size, data);  // insertEnd function call
            printf("Array: ");
            printArray(myArray, size);
            break;
```

Select option 3 to insert an element at the back.

➢ Console Output

```
Select an option: 3
Enter data to be inserted: 6
Array: | 70 | 45 | 8 | 0 | 9 | 3 | 6 |  Size:   7

Do you want to continue? (y or n):
```

6 has been inserted at the end of the array. New size is 7

1.2.3.Inserting at an index (position)

➢ Function Definition

```
49 void insertPos(int *array, int *size, int index, int element){
50   int i;
51   if (index < 0 || index > *size){  // can't insert if the index passed in is negative or grea
52     printf("Index out of range\n");
53     return;
54   }
55   (*size)++;
56   array = (int *)realloc(array, sizeof(int) * (*size)); // reallocating increased space to the
57   if(*size == 1){
58     array[0] = element;
59     return;
60   }
61   for (i = (*size)-1; i >= 0; i--){
62     if (i >= index){        // shifting elements until the index is reached
63       array[i+1] = array[i];
64     }
65     if (i == index){    // inserts element when index is reached
66       array[i] = element;
67     }
68   }
69 }
```

- If the index is less than 0 (negative) and greater than the size, a message is printed displaying; “Index out of range”.
- The size is incremented and memory reallocated to the array.
- A for loop (line 61-67) is used to shift elements one cell to the back starting from the position of the index. This creates space at the index where the element is then inserted (line 66)

➢ Function Call

```c
    case 4: printf("Enter data to be inserted: ");
          scanf("%d", &data);
          printf("Enter index (should be from 0 to size): "); // insertPos function call
          scanf("%d", &index);
          insertPos(myArray, &size, index, data);
          printf("Array: ");
          printArray(myArray, size);
          break;
```

Select option 4.

➢ Console Output

For a valid index

```
Select an option: 4
Enter data to be inserted: 900
Enter index (should be from 0 to size): 3
Array: | 70 | 45 | 8 | 900 | 0 | 9 | 3 | 6 |     Size:   8

Do you want to continue? (y or n):
```

900 is inserted at index 3 (indexing starts from 0).

For a negative index

```
Select an option: 4
Enter data to be inserted: 89
Enter index (should be from 0 to size): -1
Index out of range
Array: | 70 | 45 | 8 | 900 | 0 | 9 | 3 | 6 |     Size:   8

Do you want to continue? (y or n):
```

89 can’t be inserted because the index is negative.

For an index greater than the size

```
Select an option: 4
Enter data to be inserted: 53
Enter index (should be from 0 to size): 9
Index out of range
Array: | 70 | 45 | 8 | 900 | 0 | 9 | 3 | 6 |     Size:   8

Do you want to continue? (y or n): _
```

53 can’t be inserted because the index is greater than the size.

1.3. Delete Operations

1.3.1. Delete at the beginning of array

➢ **Function Definition**

```
70 void deleteFront(int *array, int *size){
71   int i;
72   if(*size == 0){
73     printf("Array is Empty\n");
74     return;
75   }
76   (*size)--;  // decrementing the size
77   for (i = 0; i < *size; i++){
78     array[i] = array[i+1];    // shifting elements to the left
79   }
80   if (*size == 0) free(array);  // free memory if array is empty
81   else array = (int *)realloc(array, sizeof(int) * (*size));  // reallocating reduced spa
82 }
```

- If the size is 0, then the array is empty and can’t delete (72 – 75).
- The size is decremented.
- The for loop (77 – 79) is used to move items starting from the second, one cell in front. Therefore, the second overwrites the first, thereby removing it from the array.

➢ **Function Call**

```
    case 5: deleteFront(myArray, &size);   // Function call
        printf("Array: ");
          printArray(myArray, size);
        break;
```

Select option 5.

➢ **Console Output**

```
Select an option: 5
Array: | 45 | 8 | 900 | 0 | 9 | 3 | 6 | Size:   7

Do you want to continue? (y or n):
```

70 (index 0) is deleted. Size is now 7.

1.3.2. Delete at end of array

➢ Function Definition

```c
void deleteEnd(int *array, int *size){
  if(*size == 0){
    printf("Array is Empty\n");
    return;
  }
  (*size)--;
  if (*size == 0) free(array);  // free memory if array is empty
  else array = (int *)realloc(array, sizeof(int) * (*size));  // reallocating reduced spa
}
```

- If the size is 0, then the array is empty and the function exits.
- The size is then decremented by one and used to reallocate (reduce) the arrays memory. This deletes the last element.
- If the new decremented size is 0, the array is freed.

➢ Function Call

```c
    case 6: deleteEnd(myArray, &size);  // Function call
        printf("Array: ");
          printArray(myArray, size);
        break;
```

Select option 6

➢ Console Output

```
Select an option: 6
Array: | 45 | 8 | 900 | 0 | 9 | 3 |      Size:   6

Do you want to continue? (y or n):
```

6 is deleted (last element)

1.3.3. Delete at an index

➢ **Function Definition**

```c
92  void deletePos(int *array, int *size, int index){
93    int i;
94    if(*size == 0){
95      printf("Array is Empty\n");
96      return;
97    }
98    if (index < 0 || index > (*size)-1){  // can't insert if the index passed in is negative
99      printf("Index out of range\n");
100     return;
101   }
102   (*size)--;
103   for (i = index; i < (*size); i++){
104       array[i] = array[i+1];    // shifting elements to the left once the index is reache
105   }
106   if (*size == 0) free(array);  // free memory if array is empty
107   else array = (int *)realloc(array, sizeof(int) * (*size));  // reallocating reduced spa
108 }
```

- If the index is negative or greater then the (size-1), then it is out of range and cannot delete.
- The size is decremented.
- Using the for loop (line 103 – l05), the elements starting from the index are shifted one cell to the front. Therefore, the element at the index is overwritten by the one after.
- The array is reallocated (reduced) memory using the decremented size.
- If the size is 0, therefore the array is now empty and so it is freed instead.

➢ **Function Call**

```c
      case 7: printf("Enter index: ");
              scanf("%d", &index);
              deletePos(myArray, &size, index); // Function Call
              printf("Array: ");
              printArray(myArray, size);
              break;
```

Select option 7.

➢ Console Output

Deleting at a valid index

```
Select an option: 7
Enter index: 4
Array: | 45 | 8 | 900 | 0 | 3 | Size:   5

Do you want to continue? (y or n): _
```

9 deleted. New size is 5.

Deleting at a negative index

```
Select an option: 7
Enter index: -1
Index out of range
Array: | 45 | 8 | 900 | 0 | 3 | Size:   5

Do you want to continue? (y or n): _
```

Deleting at an index greater than or equal to size

```
Select an option: 7
Enter index: 7
Index out of range
Array: | 45 | 8 | 900 | 0 | 3 | Size:   5

Do you want to continue? (y or n): _
```

1.4. Search Operation

➢ Function Definition

```c
int searchElem(int array[], int size, int element){
  int i;
  for (i = 0; i < size; i++){
    if (array[i] == element)  return i+1;
  }
  return -1;
}
```

- Returns i+1 once found else -1 if not found.

➢ Function Call

```c
    case 8: printf("Enter element: ");
            scanf("%d",&data);
            int pos = searchElem(myArray, size, data);  // Function Call
            if (pos == -1)  printf("Element is not found\n");
            else printf("Element found at position %d\n", pos);
            printf("Array: ");
            printArray(myArray, size);
            break;
```

- Select option 8
- If the pos (position) return is -1 (that is item is not found), a message is displayed; “Element not found”. Else the position of the element found is displayed.

➢ Console Output

When element exist in the array

```
Select an option: 8
Enter element: 3
Element found at position 5
Array: | 45 | 8 | 900 | 0 | 3 | Size:   5

Do you want to continue? (y or n):
```

When element doesn't exist in the array

```
Select an option: 8
Enter element: 15
Element is not found
Array: | 45 | 8 | 900 | 0 | 3 | Size:   5

Do you want to continue? (y or n):
```

1.5. Sorting Operation

➢ **Function Definition for swap**

```c
void swap(int *num1, int *num2){
  int temp;
  temp = *num1;
  *num1 = *num2;
  *num2 = temp;
}
```

➢ **Function Definition**

```c
void sortArray(int *array, int size, int order){
  int i, j;
  for (i = 0; i < size; i++){
    for (j = 0; j < size-1; j++){
      if (order == 0){  // swap in accending order if order is 0
        if (array[j] > array[j+1])  swap(&array[j], &array[j+1]);
      } // else swap in descending order.
      else if (array[j] < array[j+1]) swap(&array[j], &array[j+1]);
    }
  }
}
```

- If the order passed is 0, the array is sorted in ascending order. Else descending order.

- **Function Call**

```c
    case 9: printf("Enter order (0 for ascend and 1 for descend): ");
            scanf("%d", &data);
            sortArray(myArray, size, data);
            printf("Array sorted: ");
            printArray(myArray, size);
            break;
```

Select option 9.

- **Console Output**

Ascending

```
Select an option: 9
Enter order (0 for ascend and 1 for descend): 0
Array sorted: | 0 | 3 | 8 | 45 | 900 |  Size:   5

Do you want to continue? (y or n):
```

Descending (any value apart from 0 will sort the array in descending order)

```
Select an option: 9
Enter order (0 for ascend and 1 for descend): 1
Array sorted: | 900 | 45 | 8 | 3 | 0 |  Size:   5

Do you want to continue? (y or n):
```

```
Select an option: 9
Enter order (0 for ascend and 1 for descend): 8
Array sorted: | 900 | 45 | 8 | 3 | 0 |  Size:   5

Do you want to continue? (y or n):
```

1.6. Reversing Operation

➢ Function Definition

```c
int *reverseArray(int *array, int size){
  int i;
  // creating a new dynamic array and allocatng space to it
  int *new_array = (int *)malloc(sizeof(int)*size);
  for (i = 0; i < size; i++){
    // copying elements to new array from the last to first
    new_array[i] = array[size-i-1];
  }
  return new_array;
}
```

- The function creates a new array and copies the values of the array to the new array starting from the back to front, that is in reverse. It then returns the new array.

➢ Function Call

```c
    case 10:myArray = reverseArray(myArray, size);
            printf("Array: ");
            printArray(myArray, size);
            break;
    default:printf("Invalid Option\n");
```

Select option 10.

➢ Console Output

```
Select an option: 10
Array: | 0 | 3 | 8 | 45 | 900 | Size:   5

Do you want to continue? (y or n):
```

Entering any other number (option)

```
Select an option: 11
Invalid Option

Do you want to continue? (y or n):
```

2. STACK IMPLENTATION IN C

➢ Stack Operations:

- Display
- Check if its Empty
- Check if its Full
- Peek
- Push
- Pop

➢ Function Headings

```c
void printStack(struct stack_structure stack){...}
int isEmpty(struct stack_structure stack){...}
int isFull(struct stack_structure stack){...}
char *peek(struct stack_structure *stack){...}
void push(struct stack_structure *stack, char item[10]){...}
char *pop(struct stack_structure *stack){...}
```

➢ Stack Properties

- Items: Could be an array or linked list.
- Size: Maximum number or elements stack can hold
- Pointer: Keeps track of top item. Or the number or items currently in the stack

➢ Stack Structure

```c
// stack structure and properties. stack holds string items
struct stack_structure{
  // array of strings can hold maximum of 100 strings each of maximu
    char items[100][20];
    // keeps track of the top-most element i.e the number of element
    int pointer;
    int size;
};
```

- The stack structure has the following properties;
    - **Items:** which is a list of a maximum of 100 strings. Each string has a maximum length of 20 characters.
    - **Pointer:** keeps track of the position of the last (top) item. It shows the number of items already in stack. It is 0 when stack is empty and equals to the stack's size when the stack is full.
    - **Size:** for the maximum number of items the stack can hold.

➢ Stack declaration (creation).

```c
62 int main(){
63     struct stack_structure myStack; //creating myStack of type, stack_structure
64     int option, num, i;
65     char choice, *data = (char *)malloc(sizeof(char)*20);
66     printf("Enter stack size: ");
67     scanf("%d",&myStack.size);
68     myStack.pointer = 0;    // setting pointer to 0 since stack is empty
69     printf("Stack successfully created\n");
70     printStack(myStack);
```

- "myStack" is created (63).
- The size is entered by the user.
- The pointer is set to 0 (since stack is still empty).

- The newly created stack is display.

```
Enter stack size: 5
Stack successfully created
Stack:
                  | (empty)
                  | (empty)
                  | (empty)
                  | (empty)
                  | (empty)

Size is 5 and pointer is 0
```

➢ Options.

```c
    do{
        printf("STACK OPERTATIONS\n\n");
        printf("1. Push in to stack\n");
        printf("2. Pop out of stack\n");
        printf("3. Peek stack\n");
        printf("4. Check if stack is full\n");
        printf("5. Check if stack is empty\n");
        printf("6. Display stack\n\n");
        printf("Enter option: ");
        scanf("%d",&option);
        switch(option){⋯}
        getchar();      // sucks up the enter key pressed during the previous scanf
        printf("Do you want to continue (y or n):");
        choice = getchar();         // gets a character from the keyboard
        system("cls");      // clears the screen.
    }   while(choice == 'y' || choice == 'Y');
    return 0;
}
```

- A do while loop is used to carry out operations as many times as the user wants. After each operation, the user is asked whether or not to continue. If the user enters ‘y’ or ‘Y’, then the loop repeats else the program ends.
- A switch case control structure is used to perform the operations depending fully on the user’s option.

```
STACK OPERTATIONS

1. Push in to stack
2. Pop out of stack
3. Peek stack
4. Check if stack is full
5. Check if stack is empty
6. Display stack

Enter option: _
```

2.1. Displaying the stack.

➢ **Function Definition**

```c
void printStack(struct stack_structure stack){
    int i;
    printf("Stack:\n");
    for (i = stack.size - 1; i >= 0; i--){
      if (i == stack.pointer-1) printf("  pointer--> ");  // prints an arrow po
      else printf("             ");
        if (i >= stack.pointer) printf("\t| (empty)");  // print empty where th
        else printf("\t| %s",stack.items[i]);       // else prints the item
        printf("\n");
    }
    printf("\nSize is %d and pointer is %d\n",stack.size, stack.pointer);
    printf("\n");
}
```

- Stacks are considered to be like a pile of items; a pile of books or plates.
- The function starts printing from the top (last) position to the first (bottom).
- While the stack's pointer is not reached (that is positions are empty, since items are found only from the position of the pointer and below), "(empty)" is printed, else the value is printed.
- Where the stack's pointer is reached, and arrow is printed pointing to the top element.
- The pointer position and stack size are displayed beneath.

➢ **Function Call**

```
            case 6: printStack(myStack);
                    break;
```

Select option 6 to display stack.

➢ **Console Output**

```
Stack:
                | (empty)
                | (empty)
                | (empty)
                | (empty)
                | (empty)

Size is 5 and pointer is 0
```

The stack is currently empty. We will carry out other operations and view the stack.

2.2. Checking if the stack is empty

➢ **Function Definition**

```
int isEmpty(struct stack_structure stack){
    if (stack.pointer == 0) return 1;
    else return 0;
}
```

- All the function does is check if the stack pointer is 0. If it is, it means no item can be found in the stack. It returns 1 (true) else 0 (false).

➢ **Function Call**

```
            case 5: if (isEmpty(myStack)) printf("Stack is empty\n");
                    else printf("Stack is not empty\n");
                    break;
```

Select option 5.

- **Console Output**

```
Enter option: 5
Stack is empty
Do you want to continue (y or n):
```

2.3. Check if stack is full

- **Function Definition.**

```c
int isFull(struct stack_structure stack){
    if (stack.pointer == stack.size ) return 1;
    else return 0;
}
```

- If stack pointer equals the stack size, then it is full.

- **Function Call**

```c
            case 4: if (isFull(myStack)) printf("Stack is full\n");
                    else printf("Stack is not full\n");
                    break;
```

Select option 4.

- **Console Output**

```
Enter option: 4
Stack is not full
Do you want to continue (y or n):
```

2.4. Peeking the stack

➢ **Function Definition.**

```
char *peek(struct stack_structure *stack){
    if (isEmpty(*stack)) return NULL; // returns NULL when the
    return stack->items[stack->pointer - 1];
}
```

- The peek function simply returns the top item from the stack.
- If the stack is empty, it returns NULL.

➢ **Function Call**

```
                case 3: data = peek(&myStack);
                        printf("Top item is %s\n",data);
                        printStack(myStack);
                        break;
```

Select option 3.

➢ **Console Output**

```
Enter option: 3
Top item is (null)
Stack:
                | (empty)
                | (empty)
                | (empty)
                | (empty)
                | (empty)

Size is 5 and pointer is 0

Do you want to continue (y or n):
```

We will try peeking again once items are pushed into the stack.

Stack is currently empty

2.5. Pushing into stack

➢ Function Definition

```c
void push(struct stack_structure *stack, char item[10]){
  // the stack gets full when pointer equals stack size
    if (isFull(*stack)){
        printf("Can't push %s. Stack is full\n", item);
    }
    else{
        stack->pointer ++;  // increments pointer before pushing item
        strcpy(stack->items[stack->pointer - 1], item);
    }
}
```

- The function checks if the stack is full. If it is, it displays; “Can’t push…” else it increments the pointer and adds the item to the pointer position.

➢ Function Call

```c
        case 1: printf("How many items do you want to push: ");
            scanf("%d",&num);
            getchar();
            for (i = 0; i < num; i++){
              if (isFull(myStack)){
                printf("Stack is already full, can't push again\n");
                break;
        }
        printf("Enter item: ");
                  gets(data);
                  push(&myStack, data);       // function call
                  printf("%s successfully pushed\n\n", data);
                }
                printStack(myStack);
                break;
```

- Select option 1.
- The user enters the number of items to be pushed. Each item is then entered and pushed into the stack.
- While pushing, it the stack is full, the loop breaks and the message displays.

➢ Console Output

```
Enter option: 1
How many items do you want to push: 3
Enter item: item1
item1 successfully pushed

Enter item: item2
item2 successfully pushed

Enter item: item3
item3 successfully pushed

Stack:
                 | (empty)
                 | (empty)
  pointer-->     | item3
                 | item2
                 | item1

Size is 5 and pointer is 3
```

- Three items are pushed successfully. Notice how the pointer points to the top element. Pointer now is 3

```
Enter option: 1
How many items do you want to push: 4
Enter item: item4
item4 successfully pushed

Enter item: item5
item5 successfully pushed

Stack is already full, can't push again
Stack:
  pointer-->     | item5
                 | item4
                 | item3
                 | item2
                 | item1

Size is 5 and pointer is 5
```

- Trying to push 4 more items but only 2 more are pushed. Since at this point, the stack is full. Pointer now equals the size.

Peeking the Stack

```
Enter option: 3
Top item is item5
Stack:
  pointer-->     | item5
                 | item4
                 | item3
                 | item2
                 | item1

Size is 5 and pointer is 5

Do you want to continue (y or n):
```

Top item is item5

Checking if it is full

```
Enter option: 4
Stack is full
Do you want to continue (y or n):
```

Checking if it is empty

```
Enter option: 5
Stack is not empty
Do you want to continue (y or n):
```

2.6. Popping from stack

➢ Function Definition

```c
char *pop(struct stack_structure *stack){
  // stack is empty when pointer equals 0
    if (isEmpty(*stack)) {
        printf("Can't Pop Item. Stack is Empty\n");
        return NULL;
    }
    else {
      char *item;
      item = (char *)malloc(sizeof(char)*20);
        item = stack->items[stack->pointer - 1];
        stack->pointer --;    // decrements pointer since top value has been removed
        return item;
    }
}
```

- Works just like peek, only difference being that, it decrements the pointer by one, thereby reducing the elements by one. (removing the last element).

➢ Function Call

```c
            case 2: printf("How many items do you want to pop: ");
                scanf("%d",&num);
                for (i = 0; i < num; i++){
                  if (isEmpty(myStack)){
                    printf("Stack is already empty, can't push empty\n");
                    getchar();
                    getchar();
                    break;
            }
                      data = pop(&myStack);   // function call
                      printf("%s successfully popped\n", data);
                    }
                    puts("");
                printStack(myStack);
                    break;
```

- The user enters the number of items to be popped out. If while popping, the stack becomes empty, the message is displayed and the loop breaks.

- **Console Output**

```
Enter option: 2
How many items do you want to pop: 4
item5 successfully popped
item4 successfully popped
item3 successfully popped
item2 successfully popped

Stack:
                | (empty)
                | (empty)
                | (empty)
                | (empty)
  pointer-->    | item1

Size is 5 and pointer is 1

Do you want to continue (y or n):
```

Peeking the stack

```
Enter option: 3
Top item is item1
Stack:
                | (empty)
                | (empty)
                | (empty)
                | (empty)
  pointer-->    | item1

Size is 5 and pointer is 1

Do you want to continue (y or n):
```

Checking if it is full

```
Enter option: 4
Stack is not full
Do you want to continue (y or n):
```

Checking if it is empty

```
Enter option: 5
Stack is not empty
Do you want to continue (y or n):
```

Popping out two more items

```
Enter option: 2
How many items do you want to pop: 3
item1 successfully popped
Stack is already empty, can't pop empty
```

- We try popping out 3 more items but the stack has just item1 left. So, it pops out item 1 and displays a message that it can’t pop again since it is already empty.

3. SIMPLE LINKED LIST IN C

➢ Linked List Operations

- Display List
- Insert node
- Delete node
- Search node
- Sort List
- Reverse List

➢ Function Headers

```c
struct node_structure *init_list(struct node_structure *list, char data[20]){…}
int list_len(struct node_structure *head){…}
void print_list(struct node_structure *head){…}
void insert_head(struct node_structure *head, char data[20]){…}
void insert_end(struct node_structure *head, char data[20]){…}
void insert_index(struct node_structure *head, char data[20], int index){…}
void insert_after_id(struct node_structure *head, char data[20], int after_id){…}
void delete_head(struct node_structure *head){…}
void delete_end(struct node_structure *head){…}
void delete_id(struct node_structure *head, int id){…}
void delete_index(struct node_structure *head, int index){…}
int search_id(struct node_structure *head, int id){…}
void print_node_index(struct node_structure *head, int index){…}
void sort_list(struct node_structure *head){…}
struct node_structure *reverse_list(struct node_structure *head){…}
```

➢ Node Properties

- ID: used to uniquely identify each node;
- Data: whatever the node stores. Could be any type of variable, or a structure.
- Next: a pointer node, which can point either to another (the next) node or to NULL.

➢ Node Structure

```
struct node_structure{
    int id;
    char data[20];
    struct node_structure *next;  // pointer for next node
};
```

➢ List declaration

```
int main(){
    struct node_structure *myList = NULL;
```

- myList is created. It has the properties from the node structure. It is given the NULL value.

➢ Options

```
    do{
        puts("*****LIST OPERATIONS*****");
        puts("     ===============\n");
        puts("1.  Create Linked list");
        puts("2.  Display link list\n");
        puts("**** INSERTING ****");
        puts("3.  Insert at head");
        puts("4.  Insert at end");
        puts("5.  Insert at index");
        puts("6.  Insert after an ID\n");
        puts("**** DELETING ****");
        puts("7.  Delete head");
        puts("8.  Delete end");
        puts("9. Delete index");
        puts("10. Delete ID\n");
        puts("11. Search node(ID)");
        puts("12. Display node(Index)");
        puts("13. Display node(ID)");
        puts("14. Sort List(ID)");
        puts("15. Reverse List\n");
        printf("Enter option: ");
        scanf("%d",&option);
        getchar();
        puts("\n");
        switch(option){
```

```
        getchar();
        printf("Do you want to continue(y or n): ");
        choice = getchar();
        system("cls");
    }
    while(choice == 'y' || choice == 'y');
    return 0;
}
```

- A do while loop is used to carry out operations as many times as the user wants. After each operation, the user is asked whether or not to continue. If the user enters ‘y’ or ‘Y’, then the loop repeats else the program ends.
- A switch case control structure is used to perform the operations depending fully on the user’s option.

```
*****LIST OPERATIONS*****
     ===============

1.   Create Linked list
2.   Display link list

**** INSERTING ****
3.   Insert at head
4.   Insert at end
5.   Insert at index
6.   Insert after an ID

**** DELETING ****
7.   Delete head
8.   Delete end
9.  Delete index
10.  Delete ID

11.  Search node(ID)
12.  Display node(Index)
13.  Display node(ID)
14.  Sort List(ID)
15.  Reverse List

Enter option: _
```

LENGTH OF LIST (NUMBER OF NODES)

```c
int list_len(struct node_structure *head){
    int i = 0;
    struct node_structure *node = head;
    while (node != NULL){
        // increments i until the node is NULL. i.e the end
        i++;
        node = node->next; // move to the next node
    }
    return i;
}
```

- The while loop is used to move from one loop to the next until it reaches the last node which points to null. The number of times it moves equals the number of nodes
- The length is returned.
- This function is used throughout the program.

INITIALISING LIST (ADDING FIRST NODE)

```
struct node_structure *init_list(struct node_structure *list, char data[20]){
    list = (struct node_structure *)malloc(sizeof(struct node_structure));  // creati
    // filling the new list
    strcpy(list->data, data);
    list->id = 1;
    list->next = NULL;
    printf("List Initialised\n");
    return list;
}
```

- Once the list is created, this function can be used to initialize it.
- First. Memory is allocated to the list; the first information or data is copied to the node. Its ID is 1 since it's the first node.
- List is ended by a NULL;
- This function will be used below.

➢ Function call

```
        case 1:
            puts("CREATING LINK LIST\n");
            printf("Enter Item to initialise list with: ");
            gets(item);
            myList = init_list(myList, item); // function call
            puts("List Initialised");
            print_list(myList);
            break;
```

Option 1. Check output below

3.1. DISPLAY OPERATION

➢ **Function Definition**

```c
void print_list(struct node_structure *head){
    if (head == NULL){
        printf("List is Empty\n");
        return;
    }
    struct node_structure *node = head;
    while (node != NULL){
        // print node data while node isn't NULL (hasn't reach the end)
        printf("No: %.2d\n", node->id);
        printf("data: %s\n", node->data);
        printf("Next\n");
        printf("    |\n");
        printf("    v\n");
        node = node->next;
    }
    printf(" NULL\t\t");
    printf("No of Nodes: %d\n\n",list_len(head));
}
```

- If the head (first node) of the list is NULL, then the list is empty.
- A new node is created which is use to go through the list starting from the head.
- The data for each node is printed as it loops through.
- An arrow is used to point to the next. At the end, NULL is printed to show that, that is the end. The number of nodes is displayed by calling the list_len function (See output below)

➢ **Function Call**

```c
        case 2:
            print_list(myList); // function call
            break;
```

Option 2.

- **Console Output**

```
Enter option: 2


List is Empty
```

The list is currently empty.

Initializing the list

```
Enter option: 1


CREATING LINK LIST

Enter Item to initialise list with: student 01
List Initialised
No: 01
data: student 01
Next
    |
    v
 NULL          No of Nodes: 1
```

List has been initialized and printed. List now has one node.

We will be displaying the list after each operation.

3.2. INSERT OPERATIONS

3.2.1. Inserting at the head (beginning)

➢ **Function Definition**

```
49 void insert_head(struct node_structure *head, char data[20]){
50     if (head == NULL){
51         head = init_list(head, data); // initialises list if it is empty
52         return;
53     }
54     struct node_structure *node = (struct node_structure *)malloc(sizeof(struct node_structure));
55     // copying data of head to node.
56     strcpy(node->data, head->data);
57     node->id = head->id;
58     node->next = head->next;
59     // copying passed data to head.
60     strcpy(head->data, data);
61     head->next = node;    // pointing head to node
62     // head id equals length of list (number of nodes) during time of creation/insertion.
63     // this makes each node have a unique id
64     head->id = list_len(head);
65     printf("Node (%s) added at head\n", data);
66 }
```

- If the list is empty (head is NULL), then the list is initialized with the data. (50 – 53)
- A new node is created. The content of head is copied to the node including its next node. So, the node and the head now point to the same node. (54 – 58)
- The data to be inserted is use to replace that in head. The head then points to the node and the head ID equals the length of the list which is 2. (60 – 64)

➢ **Function call**

```
        case 3:
            printf("Enter node's data: ");
            gets(item);
            insert_head(myList, item);  // function call
            print_list(myList);
            break;
```

Option 3.

- **Console Output**

```
Enter option: 3


Enter node's data: student 02
Node (student 02) added at head
No: 02
data: student 02
Next
    |
    v
No: 01
data: student 01
Next
    |
    v
 NULL           No of Nodes: 2
```

3.2.2. Inserting at the end.

- **Function definition**

```
67 void insert_end(struct node_structure *head, char data[20]){
68     if (head == NULL){
69         head = init_list(head, data); // initialise list if it is empty
70         return;
71     }
72     struct node_structure *node = head;
73     // looping to the last node
74     while (node->next != NULL){
75         node = node->next;
76     }
77     node->next = (struct node_structure *)malloc(sizeof(struct node_structure)); // al
78   //copying data to the next of ther last node
79     strcpy(node->next->data, data);
80     node->next->next = NULL;
81     node->next->id = list_len(head);
82     printf("Node (%s) added at head\n", data);
83 }
```

- The list is transverse down to last node using the while loop (72 – 76)
- Space is allocated for a node after the last where the data is then inserted. (77-81)

➢ Function call

```c
        case 4:
            printf("Enter node's data: ");
            gets(item);
            insert_end(myList, item); // function call
            print_list(myList);
            break;
```

Option 4.

➢ Console output

```
Enter option: 4


Enter node's data: student 03
No: 02
data: student 02
Next
     |
     v
No: 01
data: student 01
Next
     |
     v
No: 03
data: student 03
Next
     |
     v
 NULL              No of Nodes: 3

Node (student 03) added at head
```

3.2.3. Inserting

```
84  void insert_index(struct node_structure *head, char data[20], int index){
85      struct node_structure *newnode;
86      int i = 0;
87      if (index == 0){
88          // inserts node at head if index is 0
89          insert_head(head, data);
90      }
91      else if (index == list_len(head)){
92          // insert node at end if index equals the length of the list
93          insert_end(head, data);
94      }
95      else if (index > 0 && index < list_len(head)){
96          // looping to the specified index
97          struct node_structure *node = head;
98          while (i < index - 1){
99              node = node->next;  // going to the next node
100             i++;
101         }
102         // creating the new node
103         newnode = (struct node_structure *)malloc(sizeof(struct node_structure));
104         strcpy(newnode->data, data);
105         // inserting new node
106         newnode->next = node->next;
107         node->next = newnode;
```

```
108         newnode->id = list_len(head);
109         printf("Node (%s) added at [%d]\n", data, index);
110     }
111     else{
112         printf("Error inserting %s node, index out of range\n", data);
113     }
114 }
```

- If the index is 0, it is inserted at the head using the insert_head() function (87 – 9)
- If the index equals the list length, insertion takes place at the end by calling insert_end() (91 – 94)
- Else if the index is between 0 and the list length, it moves from one node to the other. i.e to the (index-1)th were it then inserts (94 -101)
- A new node is created and the data to be inserted is copied into the node. The new node is inserted into the transverse position. (103 – 110)
- If the index is negative or greater than the length, a message is displayed.

➢ Function call

```c
        case 5:
            printf("Enter node's data: ");
            gets(item);
            printf("Enter index: ");
            scanf("%d", &index);
            insert_index(myList, item, index);  // function call
            print_list(myList);
```

Option 5

➢ Console output

Valid index

```
Enter node's data: student 04
Enter index: 2
Node (student 04) added at [2]
No: 02
data: student 02
Next
    |
    v
No: 01
data: student 01
Next
    |
    v
No: 04
data: student 04
Next
    |
    v
No: 03
data: student 03
Next
    |
    v
 NULL              No of Nodes: 4
```

Invalid index (out or range)

```
Enter node's data: student 05
Enter index: 5
Error inserting student 05 node, index out of range
```

3.2.4. Inserting after an ID

```
115 void insert_after_id(struct node_structure *head, char data[20], int after_id){
116     struct node_structure *newnode;
117     struct node_structure *node = head;
118     newnode = (struct node_structure *)malloc(sizeof(struct node_structure));
119     while (node->id != after_id){
120         // searching for specified id
121         if (node->next == NULL){
122             printf("Node not found\n");
123             return;
124         }
125         node = node->next;
126     }
127   // copying data to the newnode
128     strcpy(newnode->data, data);
129     // inserting newnode to list
130     newnode->next = node->next;
131     node->next = newnode;
132     newnode->id = list_len(head);
133     printf("Node (%s) added after (%d)\n", data, after_id);
134 }
```

- The list is transverse until the passed is found. If it isn't found, the function exits (117 – 126)
- A new node is created and the data is copied to it then inserted after the passed id (128 – 132)

➢ Function call

```c
        case 6:
            printf("Enter node's data: ");
            gets(item);
            printf("Enter ID to insert after: ");
            scanf("%d", &id);
            insert_after_id(myList, item, id);  // function call
            print_list(myList);
            break;
```

Option 6.

➢ Console output

Where ID exist

```
Enter node's data: student 06
Enter ID to insert after: 4
Node (student 06) added after (4)
No: 02
data: student 02
Next
    |
    v
No: 01
data: student 01
Next
    |
    v
No: 04
data: student 04
Next
    |
    v
No: 05
data: student 06
Next
    |
    v
No: 03
data: student 03
Next
    |
    v
 NULL           No of Nodes: 5
```

Id does not exist

```
Enter node's data: student 07
Enter ID to insert after: 8
Node not found
No: 02
data: student 02
Next
    |
    v
No: 01
data: student 01
Next
    |
    v
No: 04
data: student 04
Next
    |
    v
No: 05
data: student 06
Next
    |
    v
No: 03
data: student 03
Next
    |
    v
 NULL           No of Nodes: 5
```

3.3. DELETE OPERATIONS

3.3.1. Delete head

➢ **Function definition.**

```c
void delete_head(struct node_structure *head){
    struct node_structure *oldhead = head;
    printf("Node (%s) deleted\n", head->data);
    *head = *head->next;  // setting new head to the next node of the old head
    oldhead = NULL;   // freeing the old head
}
```

- The oldhead node is given the address of the head node. The head node now equals the next node (2nd node).
- The oldhead equals NULL to memory

➢ **Function call** (option 7)

```c
            case 7:
                delete_head(myList);  // function call
                print_list(myList);
                break;
```

➢ **Console Output**

```
Node (student 02) deleted
No: 01
data: student 01
Next
    |
    v
No: 04
data: student 04
Next
    |
    v
No: 05
data: student 06
Next
    |
    v
No: 03
data: student 03
Next
    |
    v
 NULL          No of Nodes: 4
```

3.3.2. Delete at end

- **Function definition**

```c
void delete_end(struct node_structure *head){
    struct node_structure *node = head;
    // looping to node before the last node
    while(node->next->next != NULL){
        node = node->next;
    }
    printf("Node (%s) deleted\n", node->next->data);
    node->next = NULL; // setting last node to NULL;
}
```

- Transverse to the second last node and sets the last node to null

- **Function call (option 8)**

```c
        case 8:
            delete_end(myList);   // function call
            print_list(myList);
            break;
```

- **Console output**

```
Node (student 03) deleted
No: 01
data: student 01
Next
    |
    v
No: 04
data: student 04
Next
    |
    v
No: 05
data: student 06
Next
    |
    v
 NULL            No of Nodes: 3
```

node 3 deleted

3.3.3. Delete index

- **Function definition**

```
165 void delete_index(struct node_structure *head, int index){
166     struct node_structure *node = head;
167     int i = 0;
168     while (node != NULL){
169         if (i == index-1){
170             struct node_structure *deletenode;
171             deletenode = node->next;  // holds the node to be deleted
172             node->next = node->next->next; // sets the leftnode to point to the ne
173             printf("Node (%s) deleted\n", deletenode->data);
174             free(deletenode);
175         }
176         node = node->next;
177         i++;
178     }
179 }
```

- The list is transverse to the (index-1)th node.
- The delete node is the next node. The node is detached from the list and freed. (171-174).

- **Function call (option 9)**

```
        case 9:
            printf("Enter index: ");
            scanf("%d",&index);
            delete_index(myList, index);  // function call
            print_list(myList);
            break;
```

- **Console output**

```
Enter index: 1
Node (student 04) deleted
No: 01
data: student 01
Next
    |
    v
No: 05
data: student 06
Next
    |
    v
 NULL              No of Nodes: 2
```

3.3.4. Delete ID

- **Function definition**

```c
150 void delete_id(struct node_structure *head, int id){
151     struct node_structure *deletenode, *node = head;
152     while (node->next->id != id){
153         // searching for specified id
154         if (node->next == NULL){
155             printf("Node not found\n");
156             return;
157         }
158         node = node->next;
159     }
160     deletenode = node->next;
161     node->next = node->next->next;
162     printf("Node (%s) deleted\n", deletenode->data);
163     free(deletenode); // freeing node
164 }
```

- The list is transverse while searching for the id. If ID isn't found, the function exits (152 – 157)
- Else once the Id is found, the node (delete node) is detached and free, (160 – 163

- **Function call (option 10)**

```c
        case 10:
            printf("Enter ID: ");
            scanf("%d",&id);
            delete_id(myList, id);  // function call
            print_list(myList);
            break;
```

- **Console output**

ID which exist

```
Node (student 06) deleted
No: 01
data: student 01
Next
    |
    v
 NULL          No of Nodes: 1

Do you want to continue(y or n): _
```

Inserting more nodes.

4. SEARCHING OPERATIONS

4.1. Searching by ID

➢ **Function definition**

```
int search_id(struct node_structure *head, int id){
    struct node_structure *node = head;
    int i;
    while(node!= NULL){
        if (node->id == id){
            return i+1;
        }
        node = node->next;
        i++;
    }
    return -1;
}
```

➢ **Function call (option 11)**

```
        case 11:
            printf("Enter node id: ");
            scanf("%d",&id);
            pos = search_id(myList, id);  // function call
            if (pos == -1)  puts("Node not found\n");
            else printf("Node found at pos [%d]\n", pos);
            break;
```

➢ **Console output**

ID which exist.

```
Enter option: 11


Enter node id: 3
Node found at pos [4]
Do you want to continue(y or n):
```

ID which does not exist

```
Enter option: 11


Enter node id: 11
Node not found

Do you want to continue(y or n):
```

4.2. Displaying node at Index

- **Function definition**

```c
void print_node_index(struct node_structure *head, int index){
    // looping to the specified index
    if (index >= 0 && index < list_len(head)){
        struct node_structure *node = head;
        int i = 0;
        while (i < index){
            node = node->next;  // going to the next node
            i++;
        }
        printf("No: %.2d\n", node->id);
        printf("data: %s\n\n", node->data);
    }
    else{
        printf("Index out of range\n");
    }
```

- **Function call (option 12)**

```c
        case 12:
            printf("Enter Index: ");
            scanf("%d",&index);
            print_node_index(myList, index);  // function call
            print_list(myList);
            break;
```

- **Console output**

The program won't display the node if the index is not within the range.

```
Enter Index: 3
No: 03
data: student 03
```

4.3. Displaying node with ID (option 13)

```
350             case 13:
351                 printf("Enter ID: ");
352                 scanf("%d",&id);
353                 pos = search_id(myList, id);  // function call
354                 print_node_index(myList, pos-1);  // function call
355                 print_list(myList);
356                 break;
```

- First, the ID is search using the search_id() (353)
- The return value is to print the at that position-1.

- **Console output**

```
Enter ID: 3
No: 03
data: student 03
```

5. SORT OPERATION

➢ Function definition

```
208 struct node_structure *sort_list(struct node_structure *head){
209   struct node_structure *node, *current = head;
210   char temp_data[20];
211   int temp_id;
212   // swapping nodes based on id
213   // bubble sort
214   while (current != NULL){
215     node = current->next;
216     while (node != NULL){
217       if (current->id > node->id){
218         // swapping data
219         strcpy(temp_data, current->data);
220         strcpy(current->data, node->data);
221         strcpy(node->data, temp_data);
222         // swapping ids
223         temp_id = current->id;
224         current->id  = node->id;
225         node->id = temp_id;
226       }
227       node = node->next; // moving to next node
228     }
229     current = current->next;
230   }
231   puts("List Sorted");
232   return head;
233 }
```

- The list is sorted in a bubble sort manner. The ID's of the current and node (next) are compared before swapping. (217 – 230)
- The function returns the (sorted) list

- **Function call (option 14)**

```
            myList = sort_list(myList); // function call
            print_list(myList);
              break;
```

- **Console output**

```
List Sorted
No: 01
data: student 01
Next
    |
    v
No: 02
data: student 02
Next
    |
    v
No: 03
data: student 03
Next
    |
    v
No: 04
data: student 05
Next
    |
    v
 NULL           No of Nodes: 4
```

List reversed

6. REVERSE OPERSTION

➢ Function definition

```c
struct node_structure *reverse_list(struct node_structure *head){
  struct node_structure *newlist = (struct node_structure *)malloc(sizeof(struct node_structure));
  // initialising newlist head with list head
  strcpy(newlist->data, head->data);
  newlist->id = head->id;
  newlist->next = NULL;
  struct node_structure *node = head->next;
  // insert head nodes to the head of newlist while looping through head.
  while(node != NULL){
    insert_head(newlist, node->data);
    newlist->id = node->id;
    node = node->next;
  }
  puts("List Reversed");
  return newlist;
}
```

- A new list is created. The list is initialized with the values of the head of the old list.
- As the old list is transverse downwards, nodes are inserted into the front of the new node thereby in reverse order.
- The new list is then returns,

➢ Function call (option 15)

```c
            case 15:
                myList = reverse_list(myList);  // function call
                print_list(myList);
                break;
```

- **Console output**

```
List Reversed
No: 04
data: student 05
Next
    |
    v
No: 03
data: student 03
Next
    |
    v
No: 02
data: student 02
Next
    |
    v
No: 01
data: student 01
Next
    |
    v
 NULL           No of Nodes: 4
```

References

(n.d.). In M. M. Cezar, Data Structures.

CUED - C++ Tutorial. (n.d.). Retrieved from UNIVERSITY OF CAMBRIDGE.

Gookin, D. (2014). Beginnning C programming for dummies.

Journaldev. (n.d.). Retrieved from Stack in C programming.

Linked LIst in C: How to Imlememnt a linked list in C. (n.d.). Retrieved from Edureka.