ASSIGNMENT : DATA STRUCTURES AND ALGORITHMS

ASSIGNMENT

ABSTRACT

There are 20 programs in this assignment.

Prajwal Sundar

```c
// Program 1

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

// Structure of a node
struct Node
{
    int data; // store data
    struct Node * next; // address of next node
};

// Create a node wih given data
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // set data
    node->next = NULL; // by default
    return node;
}

// Push an element into the stack
struct Node * push(struct Node * head, int n)
{
    struct Node * newNode = create(n); // create a new node
    newNode->next = head; // new link
    head = newNode; // change address of head
    return head;
}

// Pop an element from the stack
struct Node * pop(struct Node * head)
{
    struct Node * tmp = head; // temperory pointer
    head = head->next; // bring head address forward
    free(tmp); // free memory
    return head;
}

// Display the stack
void display(struct Node * head)
{
    if (!head) // empty stack
        printf("Your Stack : EMPTY");
    else
    {
        struct Node * ptr = head; // pointer
        while (ptr)
        {
            printf("%d ", ptr->data); // print data
            ptr = ptr->next; // bring pointer forward
        }
    }
}

void main()
{
    printf("Welcome to C Stack as a Linked List !\n\n");

    int flag = 1; // flag variable
    struct Node * head = NULL; // stack
```

```c
    while (flag)
    {
        char ch; // choice
        int n; // number to push

        printf("Enter 'P' to push, 'p' to pop, 'D' to display and 'E' to
exit : ");
        scanf(" %c", &ch);

        switch(ch)
        {
            case 'P': // push operation
                printf("Enter the element you wish to push : ");
                scanf("%d", &n);
                head = push(head, n);
                printf("%d was pushed successfully.", n);
                break;

            case 'p': // pop operation
                if (head)
                {
                    printf("%d was popped successfully.", head->data);
                    head = pop(head);
                }
                else
                    printf("Error : STACK UNDERFLOW");
                break;

            case 'D': // dispaly linked list
                display(head);
                break;

            case 'E': // exit from code
                flag = 0;
                printf("Program Execution Terminated.");
                break;

            default:
                printf("Invalid Choice.");
        }

        printf("\n\n");
    }

    printf("Thank you for using Stack as a Linked List. Bye Bye !");
}
```

```c
// Program 2

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

// Structure of a node
struct Node
{
    int data; // store data
    struct Node * next; // address of next node
};

// Create a node wih given data
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // set data
    node->next = NULL; // by default
    return node;
}

// Enqueue an element into the queue
struct Node * enqueue(struct Node * head, int n)
{
    struct Node * node = create(n); // create new node
    if (!head) // empty queue
        return node;

    struct Node * ptr = head; // pointer
    while (ptr->next)
        ptr = ptr->next; // loop till penultimate node
    ptr->next = node; // enqueue at last

    return head;
}

// Dequeue an element from the queue
struct Node * dequeue(struct Node * head)
{
    struct Node * tmp = head; // temperory pointer
    head = head->next; // bring head address forward
    free(tmp); // free memory
    return head;
}

// Display the queue
void display(struct Node * head)
{
    if (!head) // empty queue
        printf("Your Queue : EMPTY");
    else
    {
        struct Node * ptr = head; // pointer
        while (ptr)
        {
            printf("%d ", ptr->data); // print data
            ptr = ptr->next; // bring pointer forward
        }
    }
}
```

```c
void main()
{
    printf("Welcome to C Queue as a Linked List !\n\n");

    int flag = 1; // flag variable
    struct Node * head = NULL; // queue

    while (flag)
    {
        char ch; // choice
        int n; // number to push

        printf("Enter 'E' to enqueue, 'D' to dequeue, 'd' to display and
'e' to exit : ");
        scanf(" %c", &ch);

        switch(ch)
        {
            case 'E': // enqueue operation
                printf("Enter the element you wish to enqueue : ");
                scanf("%d", &n);
                head = enqueue(head, n);
                printf("%d was enqueued successfully.", n);
                break;

            case 'D': // dequeue operation
                if (head)
                {
                    printf("%d was dequeued successfully.", head->data);
                    head = dequeue(head);
                }
                else
                    printf("Error : QUEUE UNDERFLOW");
                break;

            case 'd': // dispaly linked list
                display(head);
                break;

            case 'e': // exit from code
                flag = 0;
                printf("Program Execution Terminated.");
                break;

            default:
                printf("Invalid Choice.");
        }

        printf("\n\n");
    }

    printf("Thank you for using Queue as a Linked List. Bye Bye !");
}
```

```c
// Program 3

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

// Structure of a node
struct Node
{
    int data; // store data
    struct Node * next; // address of next node
};

// Create a node with given data n
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // set data
    node->next = NULL; // by default
    return node;
}

// Enqueue operation
struct Node * enqueue(struct Node * head, int n)
{
    struct Node * node = create(n); // create a new node
    if (!head) // empty queue
        return node;

    struct Node * ptr = head; // pointer
    while (ptr->next)
        ptr = ptr->next; // loop till last node
    ptr->next = node; // enqueue at last

    return head;
}

// Dequeue operation
struct Node * dequeue(struct Node * head)
{
    if (!head) // queue underflow
        return head;

    struct Node * tmp = head; // temperory pointer
    head = head->next; // bring head address forward
    free(tmp); // free memory

    return head;
}

// Push operation into stack using enqueue
struct Node * push(struct Node * head, int n)
{
    struct Node * node = create(n); // create new node
    struct Node * ptr = head; // pointer

    while (ptr)
    {
        node = enqueue(node, ptr->data); // enqueue
        ptr = ptr->next;
    }
```

```
    head = node; // change head address
    return head;
}

// Pop operation from stack using dequeue
struct Node * pop(struct Node * head)
{
    return dequeue(head); // both are same operations
}

// Display the stack
void display(struct Node * head)
{
    if (!head) // empty stack
        printf("Your Stack : EMPTY");
    else
    {
        struct Node * ptr = head; // pointer
        while (ptr)
        {
            printf("%d ", ptr->data); // print data
            ptr = ptr->next; // bring pointer forward
        }
    }
}

void main()
{
    printf("Welcome to C Stack using Queue !\n\n");

    int flag = 1; // flag variable
    struct Node * head = NULL; // stack

    while (flag)
    {
        char ch; // choice
        int n; // number to push

        printf("Enter 'P' to push, 'p' to pop, 'D' to display and 'E' to
exit : ");
        scanf(" %c", &ch);

        switch(ch)
        {
            case 'P': // push operation
                printf("Enter the element you wish to push : ");
                scanf("%d", &n);
                head = push(head, n);
                printf("%d was pushed successfully.", n);
                break;

            case 'p': // pop operation
                if (head)
                {
                    printf("%d was popped successfully.", head->data);
                    head = pop(head);
                }
                else
                    printf("Error : STACK UNDERFLOW");
                break;
```

```
            case 'D': // dispaly linked list
                display(head);
                break;

            case 'E': // exit from code
                flag = 0;
                printf("Program Execution Terminated.");
                break;

            default:
                printf("Invalid Choice.");
        }

        printf("\n\n");
    }

    printf("Thank you for using Stack using Queue. Bye Bye !");
}
```

```c
// Program 4

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

// Structure of a node
struct Node
{
    int data; // store data
    struct Node * next; // address of next node
};

// Create a node wih given data
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // set data
    node->next = NULL; // by default
    return node;
}

// Push an element into the stack
struct Node * push(struct Node * head, int n)
{
    struct Node * newNode = create(n); // create a new node
    newNode->next = head; // new link
    head = newNode; // change address of head
    return head;
}

// Pop an element from the stack
struct Node * pop(struct Node * head)
{
    struct Node * tmp = head; // temperory pointer
    head = head->next; // bring head address forward
    free(tmp); // free memory
    return head;
}

// Enqueue an element into the queue using push operation
struct Node * enqueue(struct Node * head, int n)
{
    struct Node * tmpHead = NULL; // temperory list
    struct Node * newHead = NULL; // new list
    struct Node * ptr = head;

    while (ptr)
    {
        tmpHead = push(tmpHead, ptr->data); // push
        ptr = ptr->next; // increment pointer
    }

    tmpHead = push(tmpHead, n); // push needed element now
    ptr = tmpHead;

    while (ptr)
    {
        newHead = push(newHead, ptr->data); // push
        ptr = ptr->next; // increment pointer
    }
```

```
    head = newHead; // change address of head
    return head;
}

// Dequeue an element from the queue using pop operation
struct Node * dequeue(struct Node * head)
{
    return pop(head); // both operations are the same
}

// Display the queue
void display(struct Node * head)
{
    if (!head) // empty queue
        printf("Your Queue : EMPTY");
    else
    {
        struct Node * ptr = head; // pointer
        while (ptr)
        {
            printf("%d ", ptr->data); // print data
            ptr = ptr->next; // bring pointer forward
        }
    }
}

void main()
{
    printf("Welcome to C Queue using Stack !\n\n");

    int flag = 1; // flag variable
    struct Node * head = NULL; // queue

    while (flag)
    {
        char ch; // choice
        int n; // number to push

        printf("Enter 'E' to enqueue, 'D' to dequeue, 'd' to display and
'e' to exit : ");
        scanf(" %c", &ch);

        switch(ch)
        {
            case 'E': // enqueue operation
                printf("Enter the element you wish to enqueue : ");
                scanf("%d", &n);
                head = enqueue(head, n);
                printf("%d was enqueued successfully.", n);
                break;

            case 'D': // dequeue operation
                if (head)
                {
                    printf("%d was dequeued successfully.", head->data);
                    head = dequeue(head);
                }
                else
                    printf("Error : QUEUE UNDERFLOW");
                break;
```

```
            case 'd': // dispaly linked list
                display(head);
                break;

            case 'e': // exit from code
                flag = 0;
                printf("Program Execution Terminated.");
                break;

            default:
                printf("Invalid Choice.");
        }

        printf("\n\n");
    }

    printf("Thank you for using Queue using Stack. Bye Bye !");
}
```

```c
// Program 5

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

// Calculate length of a string
int length(char * str)
{
    int c = 0; // count variable
    for (int i = 0; str[i] != '\0'; i++, c++); // increment count
    return c;
}

// Supported Left Brackets
int isLeft(char ch)
{
    switch(ch)
    {
        case '(': case '[': case '{':
            return 1; // positive result
        default:
            return 0; // negative result
    }
}

// Supported Right Brackets
int isRight(char ch)
{
    switch(ch)
    {
        case ')': case ']': case '}':
            return 1; // positive result
        default:
            return 0; // negative result
    }
}

// Return matching left bracket
char leftOf(char ch)
{
    switch(ch)
    {
        case ')': return '(';
        case ']': return '[';
        case '}': return '{';
        default: return '\0';
    }
}

// Check if a string with brackets is balanced
int check(char * str)
{
    int l = length(str); // length of string
    char * S = (char *) malloc(l * sizeof(char)); // stack
    int T = -1; // top pointer

    for (int i = 0; i < l; i++)
    {
        if (isLeft(str[i]))
            S[++T] = str[i]; // push the bracket into the stack
        else if (isRight(str[i]))
```

```
        {
            if (S[T] != leftOf(str[i])) // invalid match
                return 0;
            else
                T--; // pop the matched left bracket
        }
    }

    if (T != -1) // some left brackets are left without a match
        return 0;
    else
        return 1; // perfectly matched
}

void main()
{
    printf("Welcome to C Bracket Balance Validator !\n\n");

    char str [100]; // input string
    printf("Enter an expression with brackets : ");
    scanf("%s", str);

    if (check(str))
        printf("The brackets in your expression are perfectly balanced.");
    else
        printf("The brackets in your expression are not properly
balanced.");

    printf("\n\nThank you for using C Bracket Balancer. Bye Bye !");
}
```

```c
// Program 6

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

// Structure of a node
struct Node
{
    int data; // number
    struct Node * next; // address of next node
};

// Create a node with given data
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(n * sizeof(struct Node));
    node->data = n; // set data
    node->next = NULL; // by default
    return node;
}

// Get middle node of a linked list
void middle(struct Node * head)
{
    if (!head) // empty list
    {
        printf("List is Empty.");
        return;
    }

    struct Node * slow = head; // slow pointer
    struct Node * fast = head; // fast pointer

    // Loop
    while (1)
    {
        if (!fast->next) // odd termination case
        {
            printf("The middle node of your linked list is %d.", slow-
>data);
            return;
        }

        else if (!fast->next->next) // even termination case
        {
            printf("The middle nodes of your linked list are %d and %d.",
slow->data, slow->next->data);
            return;
        }

        else // other cases
        {
            slow = slow->next; // forward by one position
            fast = fast->next->next; // forward by two positions
        }
    }
}

// Push an element into the beginning of a linked list
struct Node * push(struct Node * head, int n)
{
```

```
    struct Node * node = create(n); // create a node
    node->next = head; // new link
    head = node; // change head address to new node
    return head;
}

// Display the linked list
void display(struct Node * head)
{
    if (!head) // empty list
    {
        printf("List is empty.");
        return;
    }

    struct Node * ptr = head; // pointer
    printf("Your Linked List : ");

    while (ptr)
    {
        printf("%d ", ptr->data); // print data
        ptr = ptr->next; // increment pointer position by one
    }
}

// Main function
void main()
{
    printf("Welcome to C Middle Node Locator !\n\n");

    struct Node * head = NULL; // linked list
    int flag = 1;
    while (flag)
    {
        int ch; // choice
        int n; // data

        printf("Enter 'P' to push an element, 'D' to display the linked
list, 'M' to locate middle node and 'E' to exit : ");
        scanf(" %c", &ch);

        switch(ch)
        {
            case 'P':
                printf("Enter the element you want to push : ");
                scanf("%d", &n);
                head = push(head, n);
                printf("%d was succesfully pushed into the linked list.",
n);
                break;

            case 'D':
                display(head);
                break;

            case 'M':
                middle(head);
                break;

            case 'E':
                flag = 0;
```

```
                printf("Program Execution Terminated.");
                break;

            default:
                printf("Invalid Choice.");
        }

        printf("\n\n");
    }

    printf("Thank you for using C Middle Node Locator. Bye Bye !");
}
```

```c
// Program 7

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

// Definition of a node
struct Node
{
    int data; // data
    struct Node * next; // address to next node
};

// Create a node with given data
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // store data
    node->next = NULL; // by default
    return node;
}

// Function to reverse a linked list
struct Node * rev(struct Node * head)
{
    struct Node * prev = NULL;
    struct Node * curr = head;
    struct Node * next = NULL;

    while (curr != NULL)
    {
        next = curr->next; // store next node
        curr->next = prev; // reverse link
        prev = curr;
        curr = next; // bring forward
    }

    head = prev; // new head
    return head; // return head
}

// Push an element (in front) into the linked list
struct Node * push(struct Node * head, int n)
{
    struct Node * node = create(n); // create node
    node->next = head; // push
    head = node; // bring back head pointer
    return head;
}

void display(struct Node * head)
{
    if (!head) // empty linked list
        printf("EMPTY");

    else
    {
        struct Node * ptr = head; // pointer to traverse
        while (ptr != NULL)
        {
            printf("%d ", ptr->data); // print data
            ptr = ptr->next; // update pointer
```

```
        }
    }
}

void main()
{
    printf("Welcome to C Linked Lists !\n\n");

    struct Node * head = NULL;
    int flag = 1;

    while (flag)
    {
        char ch; // choice
        int n; // element to insert

        printf("Enter 'P' to push, 'D' to display, 'R' to reverse and 'E'
to exit : ");
        scanf(" %c", &ch); // get choice from user

        switch(ch)
        {
            case 'P': // push an element into the linked list
                printf("Enter the element you wish to push : ");
                scanf("%d", &n);
                head = push(head, n);
                printf("%d was pushed successfully.", n);
                break;

            case 'D': // display the linked list
                printf("Your Linked List : ");
                display(head);
                break;

            case 'R': // reverse the linked list
                head = rev(head);
                printf("Linked List was reversed successfully.");
                break;

            case 'E': // exit the program loop
                flag = 0;
                printf("Program Execution Terminatied.");
                break;

            default: // invalid choice
                printf("Invalid Choice.");
        }

        printf("\n\n");
    }

    printf("Thank you for using C Linked Lists. Bye Bye !");
}
```

```c
// Program 8

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

// Definition of a node
struct Node
{
    int data; // store data
    struct Node * next; // address of next node
};

// Create a node with given data
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // set data
    node->next = NULL; // by default
    return node;
}

// Check if loop exists
int isLoop(struct Node * head)
{
    if (!head) // base case
        return 0;

    else if (!head->next) // single node
        return 0;

    struct Node * slow = head; // slow pointer
    struct Node * fast = head->next; // fast pointer

    while (fast != NULL)
    {
        if (slow->data == fast->data)
            return 1; // loop found

        slow = slow->next; // update slow pointer by one position
        fast = fast->next; // update fast pointer by one position
        if (fast) fast = fast->next; // update fast pointer by one more
position if possible
    }

    return 0; // no loop found
}

void main()
{
    printf("Welcome to C Linked List Loop Checker !\n\n");

    struct Node * head = create(1);
    head->next = create(2);
    head->next->next = create(3);
    head->next->next->next = create(4);

    printf("Does Loop Exist in the Linked List ? Result is : %d.\n",
isLoop(head));

    head->next->next->next->next = head->next;
    printf("Now Loop is made. Result is : %d.\n\n", isLoop(head));
```

```
    printf("Thank you for using C Linked List Loop Checker. Bye Bye !");
}
```

```c
// Program 9

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

// Definition of a node
struct Node
{
    int data; // data
    struct Node * next; // address of next node
};

// Create a node with given data
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // set data
    node->next = NULL; // by default
    return node;
}

// Push an element into a given linked list
struct Node * push(struct Node * head, int n)
{
    struct Node * node = create(n); // create a node
    node->next = head; // new link
    head = node; // change address of head node
    return head;
}

// Check if linked list is palindrome or not
int check(struct Node * head)
{
    struct Node * rev = NULL; // store reverse of given linked list
    struct Node * ptr = head; // pointer

    while (ptr)
    {
        rev = push(rev, ptr->data); // push in opposite direction
        ptr = ptr->next; // increment pointer
    }

    struct Node * P1 = head;
    struct Node * P2 = rev; // pointers to both lists
    while (P1 && P2)
    {
        if (P1->data != P2->data) // data don't match
            return 0; // not a palindrome

        P1 = P1->next;
        P2 = P2->next; // update pointers
    }

    return 1; // is palindrome
}

// Display the linked list
void display(struct Node * head)
{
    if (!head) // empty list
    {
```

```
        printf("List is empty.");
        return;
    }

    struct Node * ptr = head; // pointer
    printf("Your Linked List : ");

    while (ptr)
    {
        printf("%d ", ptr->data); // print data
        ptr = ptr->next; // increment pointer position by one
    }
}

// Main function
void main()
{
    printf("Welcome to C Palindrome Checker !\n\n");

    struct Node * head = NULL; // linked list
    int flag = 1;
    while (flag)
    {
        int ch; // choice
        int n; // data

        printf("Enter 'P' to push an element, 'D' to display the linked
list, 'C' to check if palindrome and 'E' to exit : ");
        scanf(" %c", &ch);

        switch(ch)
        {
            case 'P':
                printf("Enter the element you want to push : ");
                scanf("%d", &n);
                head = push(head, n);
                printf("%d was succesfully pushed into the linked list.",
n);
                break;

            case 'D':
                display(head);
                break;

            case 'C':
                if (check(head))
                    printf("It is a palindrome.");
                else
                    printf("It is not a palindrome.");
                break;

            case 'E':
                flag = 0;
                printf("Program Execution Terminated.");
                break;

            default:
                printf("Invalid Choice.");
        }

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

```
    }

    printf("Thank you for using C Palindrome Checker. Bye Bye !");
}
```

```c
// Program 10

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

// Structure of a node
struct Node
{
    int data; // number
    struct Node * next; // address of next node
};

// Create a node with given data
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(n * sizeof(struct Node));
    node->data = n; // set data
    node->next = NULL; // by default
    return node;
}

// Perform Pairwise Swap of elements
struct Node * pairwise(struct Node * head)
{
    if (!head || !head->next) // null or single node
        return head;

    struct Node * prev = head;
    struct Node * curr = prev->next;
    struct Node * next = curr->next; // capture three positions

    curr->next = prev;
    prev->next = pairwise(next); // recursive call

    return curr;
}

// Push an element into the beginning of a linked list
struct Node * push(struct Node * head, int n)
{
    struct Node * node = create(n); // create a node
    node->next = head; // new link
    head = node; // change head address to new node
    return head;
}

// Display the linked list
void display(struct Node * head)
{
    if (!head) // empty list
    {
        printf("List is empty.");
        return;
    }

    struct Node * ptr = head; // pointer
    printf("Your Linked List : ");

    while (ptr)
    {
        printf("%d ", ptr->data); // print data
```

```
        ptr = ptr->next; // increment pointer position by one
    }
}

// Main function
void main()
{
    printf("Welcome to C Pairwise Node Swapper !\n\n");

    struct Node * head = NULL; // linked list
    int flag = 1;
    while (flag)
    {
        int ch; // choice
        int n; // data

        printf("Enter 'P' to push an element, 'D' to display the linked
list, 'S' to perform pairwise swap and 'E' to exit : ");
        scanf(" %c", &ch);

        switch(ch)
        {
            case 'P':
                printf("Enter the element you want to push : ");
                scanf("%d", &n);
                head = push(head, n);
                printf("%d was succesfully pushed into the linked list.",
n);
                break;

            case 'D':
                display(head);
                break;

            case 'S':
                head = pairwise(head);
                printf("Pairwise Swap Successful.");
                break;

            case 'E':
                flag = 0;
                printf("Program Execution Terminated.");
                break;

            default:
                printf("Invalid Choice.");
        }

        printf("\n\n");
    }

    printf("Thank you for using C Pairwise Node Swapper. Bye Bye !");
}
```

```c
// Program 11

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

// Structure of a node
struct Node
{
    int data; // store data
    struct Node * next; // address of next node
    int visited; // node is visited or not
};

// Create a node
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // set data
    node->next = NULL; // by default
    node->visited = 0; // not visited by default
}

// Locate the intersection point
int getPoint(struct Node * A, struct Node * B)
{
    struct Node * ptr = A; // pointer to first list
    while (ptr)
    {
        ptr->visited = 1; // mark visited
        ptr = ptr->next; // move pointer forward
    }

    ptr = B; // pointer to second list
    while (ptr)
    {
        if (ptr->visited) // intersection point reached
            return ptr->data;
        ptr = ptr->next; // move pointer forward
    }

    return -1; // no intersection point found
}

// Display a linked list
void display(struct Node * head)
{
    struct Node * ptr = head; // pointer to head
    while (ptr)
    {
        printf("%d ", ptr->data); // print data
        ptr = ptr->next; // increment pointer
    }
}

// Main function
void main()
{
    printf("Welcome to C Linked List Intersection Locator !\n\n");

    struct Node * A = create(3), * ptr = A;
    ptr->next = create(6); ptr = ptr->next;
```

```
    ptr->next = create(9); ptr = ptr->next;
    ptr->next = create(15); ptr = ptr->next;
    struct Node * point = ptr;
    ptr->next = create(30); ptr = ptr->next; // fill first list

    struct Node * B = create(5); ptr = B;
    ptr->next = create(10); ptr = ptr->next;
    ptr->next = point; ptr = ptr->next; // fill second list

    printf("List 1 : ");
    display(A); // print the first list
    printf("\nList 2 : ");
    display(B); // print the second list

    printf("\nThe intersection point of the lists is : %d.\n", getPoint(A,
B));
    printf("\nThank you for using C Linked List Intersection Locator. Bye
Bye !");
}
```

```c
// Program 12

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

// Definition of a tree node
struct Node
{
    int data;
    struct Node * left;
    struct Node * right;
};

// Create a binary tree node with given data
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // set data
    node->left = node->right = NULL; // leaf node by default
    return node;
}

// Check if a given tree is a BST or not
int isBST(struct Node * root, int min, int max)
{
    if (root == NULL) // null reached
        return 1;

    int rootChk = (root->data > min) && (root->data < max); // check if
root satisfies condition
    int leftChk = isBST(root->left, min, root->data); // check if left
subtree is a BST or not
    int rightChk = isBST(root->right, root->data, max); // check if right
subtree is a BST or not

    return (rootChk && leftChk && rightChk); // all 3 conditions must
satisfy
}

int check(struct Node * root)
{
    return isBST(root, INT_MIN, INT_MAX); // default minimum and maximum
limits
}

// Get binary tree as input from the user
struct Node * form()
{
    int n; // root data
    scanf("%d", &n);
    struct Node * root = create(n); // create node with given data

    int l, r; // do left and right subtrees exist or not
    printf("Does %d have a left child ? Enter 0/1 : ", n);
    scanf("%d", &l);
    if (l)
    {
        printf("Enter the left child of %d : ", n);
        root->left = form(); // left subtree
    }
```

```
    printf("Does %d have a right child ? Enter 0/1 : ", n);
    scanf("%d", &r);
    if (r)
    {
        printf("Enter the right child of %d : ", n);
        root->right = form(); // right subtree
    }

    return root;
}

// In-Order Traversal of a Binary Tree
void in(struct Node * root)
{
    if (root)
    {
        in(root->left); // left
        printf("%d ", root->data); // root
        in(root->right); // right
    }
}

void main()
{
    printf("Welcome to C Binary Search Tree Checker !\n\n");

    printf("Enter root value : ");
    struct Node * root = form();
    printf("\n");

    printf("In-Order Traversal : ");
    in(root);

    if (root)
        printf("\nThe tree is a binary search tree.");
    else
        printf("\nThe tree is not a binary search tree.");

    printf("\n\nThank you for using C Binary Search Tree checker. Bye Bye
!");
}
```

```c
// Program 13

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

// Definition of a tree node
struct Node
{
    int data;
    struct Node * left;
    struct Node * right;
};

// Create a binary tree node
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // set data
    node->left = node->right = NULL; // leaf node by default
    return node;
}

// Sort a given array using bubble sort
void sort(int * A, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n-i-1; j++)
        {
            if (A[j] > A[j+1])
            {
                int tmp = A[j];
                A[j] = A[j+1];
                A[j+1] = tmp; // swap adjacent positions
            }
        }
    }
}

// Search for an element in an array
int search(int A[], int l, int n)
{
    for (int i = 0; i < l; i++)
        if (A[i] == n)
            return i; // element found
    return -1; // element not found
}

struct Node * formTree(int pre [], int in [], int l, int L1, int U1, int
L2, int U2)
{
    if (L1 > U1) // null position
        return NULL;

    struct Node * root = create(pre[L1]); // form root
    if (L1 == U1) // leaf node
        return root;

    int P = search(in, l, root->data);
    root->left = formTree(pre, in, l, L1+1, P+L1-L2, L2, P-1);
    root->right = formTree(pre, in, l, P+L1-L2+1, U1, P+1, U2);
```

```
    return root;
}

// Construct a binary tree using pre-order and in-order traversals
struct Node * form(int pre [], int in [], int l)
{
    return formTree(pre, in, l, 0, l-1, 0, l-1);
}

// Print the post-order traversal of a binary tree
void post(struct Node * root)
{
    if (root != NULL)
    {
        post(root->left); // left
        post(root->right); // right
        printf("%d ", root->data); // root
    }
}

void main()
{
    printf("Welcome to C BST from PreOrder Generator !\n\n");

    int n;
    printf("Enter the number of elements in the BST : ");
    scanf("%d", &n);

    int pre [n], in[n]; // arrays to store traversals
    printf("Enter the pre-order traversal of the BST : ");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &pre[i]);
        in[i] = pre[i]; // first copy pre-order as such into in-order
    }

    sort(in, n); // sort the pre-order traversal in ascending order to get
in-order
    struct Node * root = form(pre, in, n);

    printf("In-Order Traversal : ");
    for (int i = 0; i < n; i++)
        printf("%d ", in[i]);

    printf("\nPost-Order Traversal : ");
    post(root);

    printf("\n\nThank you for using C BST from Pre-Order Generator. Bye Bye
!");
}
```

```c
// Program 14

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

// Definition of a node
struct Node
{
    int data;
    struct Node * left;
    struct Node * right;
};

// Create a node
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // set data
    node->left = node->right = NULL; // leaf by default
    return node;
}

// Maximum of 2 variables function
int max(int x, int y)
{
    return (x > y) ? x : y;
}

// Height of a binary tree function
int height(struct Node * root)
{
    if (root == NULL)
        return -1; // to delete an extra edge traversed
    else
        return 1 + max(height(root->left), height(root->right)); // normal
case
}

// Search an element in an array
int search(int * A, int L, int U, int n)
{
    for (int i = L; i <= U; i++)
        if (*(A + i) == n)
            return i; // element found
    return -1; // element not found
}

// Consturct a Tree Using Pre-Order and In-Order Traversals
struct Node * formTree(int pre [], int in [], int L1, int U1, int L2, int
U2)
{
        if (L1 > U1)
                return NULL;

        struct Node * root = create(pre[L1]);
        if (L1 == U1)
                return root;

        int P = search(in, L2, U2, root->data);
        root->left = formTree(pre, in, L1+1, P+L1-L2, L2, P-1);
        root->right = formTree(pre, in, P+L1-L2+1, U1, P+1, U2);
```

```
        return root;
}

struct Node * form(int pre [], int in [], int l)
{
        return formTree(pre, in, 0, l-1, 0, l-1);
}

void main()
{
    printf("Welcome to C Binary Tree Height Evaluator !\n\n");

    int n; // number of nodes
    printf("Enter the number of nodes in your tree : ");
    scanf("%d", &n);

    int pre [n], in [n]; // arrays to store traversals
    printf("Enter the pre-order traversal : ");
    for (int i = 0; i < n; i++)
        scanf("%d", &pre[i]);
    printf("Enter the in-order traversal : ");
    for (int i = 0; i < n; i++)
        scanf("%d", &in[i]);

    printf("Height of the Tree Generated is : %d.", height(form(pre, in,
n)));
    printf("\n\nThank you for using C Binary Tree Height Evaluator. Bye Bye
!");
}
```

```c
// Program 15

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

// Structure of a tree node
struct tNode
{
    int data; // store data
    struct tNode * left; // address of left child
    struct tNode * right; // address of right child
};

// Structure of a queue node
struct qNode
{
    struct tNode * node; // store node
    struct qNode * next; // address to next node
};

// Create a tNode with given data
struct tNode * CtNode(int n)
{
    struct tNode * node = (struct tNode *) malloc(sizeof(struct tNode));
    node->data = n; // set given data
    node->left = node->right = NULL; // by default
    return node;
}

// Create a qNode with a given tNode
struct qNode * CqNode(struct tNode * node)
{
    struct qNode * newNode = (struct qNode *) malloc(sizeof(struct qNode));
    newNode->node = node; // set tNode
    newNode->next = NULL; // by default
    return newNode;
}

// Enqueue function
struct qNode * enqueue(struct qNode * head, struct tNode * node)
{
    struct qNode * newNode = CqNode(node); // create a new qNode
    if (!head) // empty queue
        return newNode;

    struct qNode * ptr = head; // pointer
    while (ptr->next)
        ptr = ptr->next; // loop till last node
    ptr->next = newNode; // enqueue at last

    return head;
}

// Dequeue function
struct qNode * dequeue(struct qNode * head)
{
    if (!head) // empty queue - underflow
        return head;

    struct qNode * tmp = head; // temperory pointer
    head = head->next; // dequeue
```

```
    free(tmp); // free memory

    return head;
}

// Function to perform level order traversal
void levelOrder(struct tNode * root)
{
    struct qNode * head = CqNode(root); // enqueue root
    head->next = CqNode(NULL); // enqueue NULL
    int l = 0; // level 0
    printf("Level 0 : ");

    while (1) // loop elements
    {
        struct tNode * ptr = head->node; // store front element
        head = dequeue(head); // dequeue front position

        if (!head) // queue is empty
            break; // end loop

        if (ptr) // NOT NULL
        {
            printf("%d ", ptr->data); // print data
            if (ptr->left)
                head = enqueue(head, ptr->left); // enqueue left child
            if (ptr->right)
                head = enqueue(head, ptr->right); // enqueue right child
        }

        else // NULL
        {
            head = enqueue(head, NULL); // enquue a NULL position
            printf("\nLevel %d : ", ++l); // print new line for new level
        }
    }
}

// Find the position of an element in an array
int search(int arr [], int L, int U, int n)
{
    for (int i = L; i <= U; i++)
        if (arr[i] == n)
            return i; // element found, return position
    return -1; // element not found
}

// Form a binary tree using given post order and in order traversals
struct tNode * formTree(int post [], int in [], int L1, int U1, int L2, int
U2)
{
    if (L1 > U1)
                return NULL;

        struct tNode * root = CtNode(post[U1]);
        if (L1 == U1)
                return root;

        int P = search(in, L2, U2, root->data);
        root->left = formTree(post, in, L1, P+L1-L2-1, L2, P-1);
        root->right = formTree(post, in, P+L1-L2, U1-1, P+1, U2);
```

```
        return root;
}

struct tNode * form(int post [], int in [], int l)
{
    return formTree(post, in, 0, l-1, 0, l-1); // set default limits
}

// Main function
void main()
{
    printf("Welcome to C Level Order Traversal !\n\n");

    int n; // number of elements
    printf("Enter the number of elements in your tree : ");
    scanf("%d", &n);

    int post [n];
    int in [n]; // array to store traversals
    printf("Enter the post order traversal of your tree : ");
    for (int i = 0; i < n; i++)
        scanf("%d", &post[i]);
    printf("Enter the in order traversal of your tree : ");
    for (int i = 0; i < n; i++)
        scanf("%d", &in[i]);

    printf("\nThe Level Order Traversal of your tree is as follows :-\n");
    levelOrder(form(post, in, n));

    printf("\n\nThank you for using C Level Order Traversal. Bye Bye !");
}
```

```c
// Program 16

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

// Structure of a tree node
struct tNode
{
    int data; // store data
    struct tNode * left; // address of left child
    struct tNode * right; // address of right child
};

// Structure of a stack node
struct sNode
{
    struct tNode * node; // store a tree node
    struct sNode * next; // address of next node
};

// Create a tree node with given data
struct tNode * CtNode(int n)
{
    struct tNode * node = (struct tNode *) malloc(sizeof(struct tNode));
    node->data = n; // set data
    node->left = node->right = NULL; // by default
    return node;
}

// Create a stack node with a given tree node
struct sNode * CsNode(struct tNode * node)
{
    struct sNode * newNode = (struct sNode *) malloc(sizeof(struct sNode));
    newNode->node = node; // set the node
    newNode->next = NULL;
    return newNode;
}

// Push a tree node into the stack
struct sNode * push(struct sNode * head, struct tNode * node)
{
    struct sNode * newNode = CsNode(node); // new node
    newNode->next = head; // set link
    head = newNode; // change address of head
    return head;
}

// Pop a tree node from the stack
struct sNode * pop(struct sNode * head)
{
    struct sNode * tmp = head; // temperory pointer
    head = head->next; // bring forward head
    free(tmp); // free memory
    return head;
}

// Iterative pre-order traversal of a tree
void pre(struct tNode * root)
{
    struct sNode * head = CsNode(root);
```

```
    while (head) // loop while head is not empty
    {
        struct tNode * ptr = head->node; // get top element
        head = pop(head); // pop top element
        printf("%d ", ptr->data);

        if (ptr->right)
            head = push(head, ptr->right); // push right child
        if (ptr->left)
            head = push(head, ptr->left); // push left child
    }
}

// Iterative in-order traversal of a tree
void in(struct tNode * root)
{
    struct sNode * head = NULL; // stack
    struct tNode * ptr = root; // pointer

    while (head || ptr)
    {
        if (ptr) // pointer is NOT NULL
        {
            head = push(head, ptr); // push pointer
            ptr = ptr->left; // go left
        }

        else // pointer is NULL
        {
            ptr = head->node; // point to topmost element
            head = pop(head); // pop topmost element
            printf("%d ", ptr->data);
            ptr = ptr->right; // go right
        }
    }
}

// Iterative post-order traversal of a tree
void post(struct tNode * root)
{
    struct sNode * head = NULL; // stack
    struct tNode * ptr = root; // pointer

    while (head || ptr)
    {
        if (ptr) // pointer is NOT NULL
        {
            if (ptr->right)
                head = push(head, ptr->right); // push right child
            head = push(head, ptr); // push root
            ptr = ptr->left; // go left
        }

        else // pointer is NULL
        {
            ptr = head->node; // point to topmost element
            head = pop(head); // pop topmost element

            if ((ptr->right) && (head) && (ptr->right->data == head->node-
>data))
            {
```

```
                head = pop(head); // pop right child
                head = push(head, ptr); // push root
                ptr = ptr->right; // go right
            }

            else
            {
                printf("%d ", ptr->data); // print data
                ptr = NULL; // set pointer to NULL
            }
        }
    }
}

// Get binary tree as input from the user
struct tNode * form()
{
    int n; // root data
    scanf("%d", &n);
    struct tNode * root = CtNode(n); // create node with given data

    int l, r; // do left and right subtrees exist or not
    printf("Does %d have a left child ? Enter 0/1 : ", n);
    scanf("%d", &l);
    if (l)
    {
        printf("Enter the left child of %d : ", n);
        root->left = form(); // left subtree
    }
    printf("Does %d have a right child ? Enter 0/1 : ", n);
    scanf("%d", &r);
    if (r)
    {
        printf("Enter the right child of %d : ", n);
        root->right = form(); // right subtree
    }

    return root;
}

void main()
{
    printf("Welcome to C Binary Tree Iterative Traversals !\n\n");

    printf("Enter root value : ");
    struct tNode * root = form();
    printf("\n");

    printf("Pre-Order Traversal : ");
    pre(root);
    printf("\nIn-Order Traversal : ");
    in(root);
    printf("\nPost-Order Traversal : ");
    post(root);

    printf("\n\nThank you for using C Binary Tree Iterative Traversals. Bye
Bye !");
}
```

```c
// Program 17

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

// Swap Elements at 2 positions
void swap(int * A, int x, int y)
{
    int tmp = *(A + x);
    *(A + x) = *(A + y);
    *(A + y) = tmp;
}

// Bubble Sort
void Bsort(int * A, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n-i-1; j++)
        {
            // check condition to do swapping
            if (*(A + j) > *(A + j + 1))
                swap(A, j, j+1); // swap adjcent elements
        }
    }
}

// Selection Sort
void Ssort(int * A, int n)
{
    for (int i = 0; i < n; i++)
    {
        int sml = *(A + i), p = i; // smallest element and position
        for (int j = i+1; j < n; j++)
        {
            if (*(A + j) < sml)
            {
                sml = *(A + j);
                p = j; // update smallest element and position
            }
        }
        swap(A, i, p); // swap elements at ith and pth positions
    }
}

// Insertion Sort
void Isort(int * A, int n)
{
    for (int i = 0; i < n; i++)
    {
        int val = *(A + i);
        int j;
        for (j = i; (j > 0) && (*(A + j - 1) > val); j--)
            *(A + j) = *(A + j - 1);
        *(A + j) = val;
    }
}

// Radix Sort
int getMax(int arr [], int n)
{
```

```
    int mx = arr[0]; // maximum element
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i]; // update maximum
    return mx; // return maximum element in array
}

void countSort(int arr [], int n, int exp) // count sort digit by digit
{
    int output[n]; // output array
    int count [10] = { 0 };

    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++; // fill counts
    for (int i = 1; i < 10; i++)
        count[i] += count[i-1]; // fill cumulative counts
    for (int i = n-1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i]; // fill output
        count[(arr[i] / exp) % 10]--; // reduce count
    }
    for (int i = 0; i < n; i++)
        arr[i] = output[i]; // copy back into original array
}

void Rsort(int arr [], int n) // main radix sort code
{
    int m = getMax(arr, n);
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

// Merge Sort
void merge(int * A, int L, int U)
{
    int M = (L + U) / 2;
    int * B = (int *) malloc((U-L+1) * sizeof(int));
    int P1 = L, P2 = M+1, P = 0; // pointers to arrays

    // Compare and insert into new array
    while (P1 <= M && P2 <= U)
    {
        if (*(A + P1) < *(A + P2))
            *(B + (P++)) = *(A + (P1++));
        else
            *(B + (P++)) = *(A + (P2++));
    }

    // Insert remaining elements into new array
    while (P1 <= M)
        *(B + (P++)) = *(A + (P1++));
    while (P2 <= U)
        *(B + (P++)) = *(A + (P2++));

    // Copy values from new array into original array
    int P3 = L; P = 0;
    while (P3 <= U)
        *(A + (P3++)) = *(B + (P++));
}

void M_sort(int * A, int L, int U)
{
```

```
    if (L == U) return; // base case
    int M = (L + U) / 2;
    M_sort(A, L, M); // sort left half
    M_sort(A, M+1, U); // sort right half
    merge(A, L, U); // merge sorted halves
}

void Msort(int * A, int n)
{
    M_sort(A, 0, n-1); // set default limits as 0 to n-1 (whole array)
}

// Quick Sort
int partition(int * A, int L, int U)
{
    int i = L-1;
    int pivot = *(A + U);

    for (int j = L; j < U; j++)
    {
        if (*(A + j) < pivot)
        {
            i++;
            swap(A, i, j); // swap
        }
    }

    swap(A, i+1, U); // final swap
    return (i+1);
}

void Q_sort(int * A, int L, int U)
{
    if (L < U) // L >= U is termination case
    {
        int P = partition(A, L, U); // get correct position of pivot
        Q_sort(A, L, P-1); // sort left of pivot
        Q_sort(A, P+1, U); // sort right of pivot
    }
}

void Qsort(int * A, int n)
{
    Q_sort(A, 0, n-1); // set default limits as 0 to n-1
}

// Heap sort
void heapify(int * arr, int N, int i)
{
    int largest = i;
    int left = (2 * i) + 1; // left child
    int right = (2 * i) + 2; // right child

    if (left < N && arr[left] > arr[largest]) // left is greater than root
        largest = left;
    if (right < N && arr[right] > arr[largest]) // right is greater than
root
        largest = right;

    if (largest != i) // largest is not root
    {
```

```
        swap(arr, largest, i);
        heapify(arr, N, largest); // heapify at child position
    }
}

void Hsort(int * arr, int N)
{
    for (int i = (N/2)-1; i >= 0; i--)
        heapify(arr, N, i); // heapify starting from parent of last leaf

    for (int i = N-1; i >= 0; i--)
    {
        swap(arr, 0, i); // swap root and last element
        heapify(arr, i, 0); // heapify root
    }
}

void main()
{
    printf("Welcome to C Array Sorter !\n");
    printf("7 sorts are available : Bubble (B), Selection (S), Insertion
(I), Radix (R), Merge (M), Quick (Q), Heap (H).\n\n");

    int n; // number of elements
    printf("Enter the number of elements : ");
    scanf("%d", &n);

    int * arr = (int *) malloc(n * sizeof(int));
    printf("Enter the unsorted array : ");
    for (int i = 0; i < n; i++)
        scanf("%d", arr+i);

    char ch; // choice of sort
    printf("Enter your choice of sort : ");
    scanf(" %c", &ch);

    switch(ch)
    {
        case 'B': Bsort(arr, n); break;
        case 'S': Ssort(arr, n); break;
        case 'I': Isort(arr, n); break;
        case 'R': Rsort(arr, n); break;
        case 'M': Msort(arr, n); break;
        case 'Q': Qsort(arr, n); break;
        case 'H': Hsort(arr, n); break;
        default: printf("Invalid Sort."); break;
    }

    printf("Sorted Array : ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    printf("\n\nThank you for using C Array Sorter. Bye Bye !");
}
```

```c
// Program 18

#include <stdio.h>

void main()
{
    char sorts [8] [4] [100] = {
        {"SORT\t", "WORST CASE", "AVERAGE CASE", "BEST CASE"},
        {"Bubble\t", "O(n^2)\t", "O(n^2)\t", "O(n)"},
        {"Insertion", "O(n^2)\t", "O(n^2)\t", "O(n)"},
        {"Selection", "O(n^2)\t", "O(n^2)\t", "O(n)"},
        {"Quick\t", "O(n^2)\t", "O(nlogn)", "O(nlogn)"},
        {"Merge\t", "O(nlogn)", "O(nlogn)", "O(nlogn)"},
        {"Heap\t", "O(nlogn)", "O(nlogn)", "O(nlogn)"},
        {"Radix\t", "O(nd)\t", "O(nd)\t", "O(nd)"}
    };

    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 4; j++)
            printf("%s\t", sorts[i][j]);
        printf("\n");
    }
}
```

```c
// Program 19

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

// Defintion of a node
struct Node
{
    int data;
    struct Node * next;
};

// Create a new node
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // store data
    node->next = NULL;
    return node;
}

// Display the hash table
void display(struct Node ** A, int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("Position %d : ", i);
        if (A[i])
            printf("%d\n", A[i]->data); // print data
        else
            printf("-\n"); // print nothing
    }
}

// Insert a new number into the hash table
void insert(struct Node ** A, int size, int c, int num)
{
    if (c >= size)
    {
        printf("Array is FULL. More elements cannot be accomodated.\n");
        return;
    }

    int key = (num % size);
    while (A[key] != NULL)
        key = (key+1) % size;
    A[key] = create(num);

    printf("%d inserted successfully into the hash table.\n");
}

// Search a number in the hash table
int search(struct Node ** A, int size, int num)
{
    int c = 0;
    int key = num;

    while (c < size)
    {
        key = key % size;
        if (A[key] && A[key]->data == num)
```

```
            return key;
        key++; c++;
    }

    return -1;
}

// Delete a number from the hash table
void delete(struct Node ** A, int size, int num)
{
    int p = search(A, size, num);

    if (p == -1) // element not found
    {
        printf("%d was not found in your hash table.\n", num);
        return;
    }

    else
    {
        free(A[p]); // delete the node at position p
        A[p] = NULL;
        printf("%d was successfully deleted from your hash table.\n", num);
    }
}

// Main function
void main()
{
    printf("Welcome to C Hash Table : Linear Probing \n");

    int size;
    printf("Enter the size of your hash table : ");
    scanf("%d", &size);
    printf("\n");

    struct Node ** A = (struct Node **) malloc(size * sizeof(struct Node
*));
    for (int i = 0; i < size; i++)
        A[i] = NULL;

    int flag = 1;
    int c = 0;
    while (flag)
    {
        char ch;
        int num;

        printf("Enter 'I' to insert, 'D' to delete, 'S' to search, 'd' to
display and 'E' to exit : ");
        scanf(" %c", &ch);

        switch(ch)
        {
            case 'I':
                printf("Enter the number you wish to insert : ");
                scanf("%d", &num);
                insert(A, size, c++, num);
                break;

            case 'D':
```

```
                printf("Enter the number you wish to delete : ");
                scanf("%d", &num);
                delete(A, size, num);
                c--;
                break;

            case 'S':
                printf("Enter the number you wish to search : ");
                scanf("%d", &num);
                int p = search(A, size, num);
                if (p == -1)
                    printf("%d was not found in your hash table.\n", num);
                else
                    printf("%d was found in your hash table at position
%d.\n", num, p);
                break;

            case 'd':
                display(A, size);
                break;

            case 'E':
                flag = 0;
                break;

            default:
                printf("Invalid Choice.\n");
        }

        printf("\n");
    }

    printf("Thank you for using C Hash Table : Linear Probing. Bye Bye !");
}
```

```c
// Program 20

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

// Defintion of a node
struct Node
{
    int data;
    struct Node * next;
};

// Create a new node
struct Node * create(int n)
{
    struct Node * node = (struct Node *) malloc(sizeof(struct Node));
    node->data = n; // store data
    node->next = NULL;
    return node;
}

// Display the hash table
void display(struct Node ** A, int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("Position %d : ", i);
        if (A[i])
            printf("%d\n", A[i]->data); // print data
        else
            printf("-\n"); // print nothing
    }
}

// Insert a new number into the hash table
void insert(struct Node ** A, int size, int c, int num)
{
    if (c >= size)
    {
        printf("Array is FULL. More elements cannot be accomodated.\n");
        return;
    }

    int key = (num % size);
    int n = 0;

    while (A[key] != NULL)
        key = (key + (2 * (n++) + 1)) % size;
    A[key] = create(num);

    printf("%d inserted successfully into the hash table.\n");
}

// Search a number in the hash table
int search(struct Node ** A, int size, int num)
{
    int c = 0;
    int key = num;
    int n = 0;

    while (c < size)
```

```
    {
        key = key % size;
        if (A[key] && A[key]->data == num)
            return key;
        key = (2 * (n++) + 1); c++;
    }

    return -1;
}

// Delete a number from the hash table
void delete(struct Node ** A, int size, int num)
{
    int p = search(A, size, num);

    if (p == -1) // element not found
    {
        printf("%d was not found in your hash table.\n", num);
        return;
    }

    else
    {
        free(A[p]); // delete the node at position p
        A[p] = NULL;
        printf("%d was successfully deleted from your hash table.\n", num);
    }
}

// Main function
void main()
{
    printf("Welcome to C Hash Table : Quadratic Probing \n");

    int size;
    printf("Enter the size of your hash table : ");
    scanf("%d", &size);
    printf("\n");

    struct Node ** A = (struct Node **) malloc(size * sizeof(struct Node
*));
    for (int i = 0; i < size; i++)
        A[i] = NULL;

    int flag = 1;
    int c = 0;
    while (flag)
    {
        char ch;
        int num;

        printf("Enter 'I' to insert, 'D' to delete, 'S' to search, 'd' to
display and 'E' to exit : ");
        scanf(" %c", &ch);

        switch(ch)
        {
            case 'I':
                printf("Enter the number you wish to insert : ");
                scanf("%d", &num);
                insert(A, size, c++, num);
```

```
                break;

            case 'D':
                printf("Enter the number you wish to delete : ");
                scanf("%d", &num);
                delete(A, size, num);
                c--;
                break;

            case 'S':
                printf("Enter the number you wish to search : ");
                scanf("%d", &num);
                int p = search(A, size, num);
                if (p == -1)
                    printf("%d was not found in your hash table.\n", num);
                else
                    printf("%d was found in your hash table at position
%d.\n", num, p);
                break;

            case 'd':
                display(A, size);
                break;

            case 'E':
                flag = 0;
                break;

            default:
                printf("Invalid Choice.\n");
        }

        printf("\n");
    }

    printf("Thank you for using C Hash Table : Quadratic Probing. Bye Bye
!");
}
```