Karnataka Law Society's

GOGTE INSTITUTE OF TECHNOLOGY

Udyambag Belagavi -590008

Karnataka, India.

A Course Project Report on

STACK ADT

Submitted for the requirements of 3rd semester B.E. in CSE

for **"Data Structures with C(18CS32)"**

Submitted by

Name	USN
Venkatesh G Dhongadi	**2GI19CS175**

Under the guidance of

Prof. JYOTI AMBOJI

Dept of Computer Science

Academic Year 2021 (Odd semester)

Table of contents

Signature

1. Name/title of the project:

STACK ADT

2. Statement of the problem:

Develop and execute a C program to show implementation of Stack ADT. Perform its basic operations

3. Objectives and scope of the project:

- Understand the Stack Data Structure
- Understand basic operations on stack
- Understand the method of defining Stack ADT and Implement the basic operations
- Understand how to import ADT in a program
- Understand how member functions of an ADT are accessed in an application program

4. Hardware and software to be used:

Hardware: A computer used to run the program

Software: Windows OS, Code Blocks 20.03 (with GCC compiler)

5. Methodology:

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc. A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

Stack ADT

The stack ADT implementation in C is straightforward. Rather than store data in each node, we store a pointer to the data. It is the application program's responsibility to allocate memory for the data and pass the address to the stack ADT. Within the ADT, the stack node looks like any linked list node except that it contains a pointer to the data rather than the actual data. Because the data pointer type is unknown, it is stored as a pointer to void. The head node and the data nodes are encapsulated in the ADT. The calling function's only view of the stack is a pointer to the stack structure in the ADT, which is declared as a type definition. We name this stack type STACK. This design is very similar to C's FILE structure.

To create a stack, the programmer defines a pointer to a stack as shown in the following example and then calls the create stack function. The address of the stack structure is returned by create stack and assigned to the pointer in the calling function.

ADT Implementation

It takes more than a data structure to make an abstract data type: there must also be operations that support the stack. We develop the C functions in the sections that follow.

Stack Structure: The stack abstract data type structure is shown in Program. The node structure consists only of a data pointer and a link node pointer. The stack head structure also contains only two elements, a pointer to the top of the stack and a count of the number of entries currently in the stack.

Create Stack: Create stack allocates a stack head node, initializes the top pointer to null, and zeros the count field. The address of the node in the dynamic memory is then returned to the caller. The call to create a stack must assign the return pointer value to a stack pointer.

Push Stack: The first thing that we need to do when we push data into a stack is to find a place for the data. This requires that we allocate memory from the heap using *malloc*. Once the memory is allocated, we simply assign the data pointer to the node and then set the link to point to the node currently indicated as the stack top. We also need to add one to the stack count field.

Pop Stack: Pop stack returns the data in the node at the top of the stack. It then deletes and recycles the node. After the count is adjusted by subtracting 1, the function returns to the caller. Note the way underflow is reported, we set the data pointer to NULL. If the stack is empty, when we return the data pointer we return NULL.

Stack Top: The stack top function returns the data at the top of the stack without deleting the top node. It allows the user to see what will be deleted when the stack is popped.

Empty Stack: Because the calling function has no access to the data structure, it cannot determine if there are data in the stack without actually trying to retrieve them. We therefore provide empty stack, a function that simply reports that the stack has data or that it is empty.

Full Stack: Full stack is one of the most complex of the supporting functions. There is no straightforward way to tell if the next memory allocation is going to succeed or fail. All we can do is try it. But by trying to make an allocation, we use up part of the heap. Therefore, after allocating space for a node, we immediately free it so that it will be there when the program requests memory.

Stack Count: Stack count returns the number of items in the stack. Because this count is stored in the stack head node, we do not need to traverse the stack to determine how many items are currently in it. We simply return the head count.

Destroy Stack: Destroy stack deletes the nodes in a stack and returns a null pointer. It is the user's responsibility to set the stack pointer in the calling area to NULL by assigning the return value to the local stack pointer. Because the stack is implemented as a dynamic data structure in the heap, the memory is also released for reuse.

PROGRAM

```
#include   <stdio.h>

#include  <stdlib.h>

#include<stdbool.h>


// Stack ADT Type Defintions

typedef struct node

{

   void* dataPtr;

   struct node* link;

} STACK_NODE;


typedef struct

{
```

```
    int count;

    STACK_NODE* top;

} STACK;


//prototype declarations

STACK* createStack (void);

bool pushStack (STACK* , void* );

void* popStack (STACK* );

void* stackTop (STACK* );

bool emptyStack (STACK* );

bool fullStack (STACK* );

int stackCount (STACK* );

STACK* destroyStack (STACK* );


STACK* createStack (void)

{

    // Local Definitions

    STACK* stack;

    // Statements

    stack = (STACK*) malloc( sizeof (STACK));
```

```
    if (stack)
    {
        stack->count = 0;
        stack->top = NULL;
    } // if
    return stack;
} // createStack


bool pushStack (STACK* stack, void* dataInPtr)
{
    // Local Definitions
    STACK_NODE* newPtr;
    // Statements
newPtr = (STACK_NODE* ) malloc(sizeof( STACK_NODE));
    if (!newPtr)
        return false;
newPtr->dataPtr = dataInPtr;
newPtr->link = stack->top;
    stack->top = newPtr;
    (stack->count)++;
```

```
    return true;
} // pushStack


void* popStack (STACK* stack)
{
    // Local Definitions
    void* dataOutPtr;
    STACK_NODE* temp;
    // Statements
    if (stack->count == 0)
dataOutPtr = NULL;
    else
    {
        temp = stack->top;
dataOutPtr = stack->top->dataPtr;
        stack->top = stack->top->link;
        free (temp);
        (stack->count)--;
    } // else
    return dataOutPtr;
```

```
} // popStack


void* stackTop (STACK* stack)

{

// Statements

    if (stack->count == 0)

    return NULL;

    else

    return stack->top->dataPtr;

} // stackTop


bool emptyStack (STACK* stack)

{

    // Statements

    return (stack->count = 0);

} // emptyStack


bool fullStack (STACK* stack)

{

    // Local Definitions
```

```
    STACK_NODE* temp;

    // Statements

    if ((temp =(STACK_NODE*)malloc (sizeof(*(stack->top)))))

    {

        free (temp);

        return false;

    } // if

    // malloc failed

    return true;

} // fullStack


int stackCount (STACK* stack)

{

// Statements

    return stack->count;

} // stackCount


STACK* destroyStack (STACK* stack)

{

    // Local Definitions
```

```
    STACK_NODE* temp;

    // Statements

    if (stack)

    {

        // Delete all nodes in stack

        while (stack->top != NULL)

        {

            // Delete data entry

            free (stack->top->dataPtr);

            temp = stack->top;

            stack->top = stack->top->link;

            free (temp);

        } // while

    // Stack now empty. Destroy stack head node.

    free (stack);

    } // if stack

    return NULL;

} // destroyStack


void disp(STACK_NODE*);     //prototype
```

```
int main()

{

   STACK *s;

   s = createStack();

   int *item;

   int *newdata,*top;

   int choice,count;

   while(1) {

printf("\n1: Push\t\t2: Pop\t\t3: Stack Top\t\t4: Stack Count\t\t5: Display\t\t6: Destroy Stack and Exit\nEnter your choice: ");

scanf("%d", &choice);

   switch(choice){

   case 1: item=(int*)malloc(sizeof(int));

         if(fullStack(s)==true)

printf("Stack full\n");

         else{

printf("Enter the item to be pushed: ");

scanf("%d",item);

pushStack(s,item);

         }

         break;
```

```
   case 2: newdata =(int*)popStack(s);
           if(newdata==NULL)
printf("Stack Empty\n");
           else
printf("Popped item is: %d\n", *newdata);
           break;
   case 3: top=(int*)stackTop(s);
           if(top==NULL)
printf("Stack Empty\n");
           else
printf("Item at the top is: %d\n", *top);
           break;
   case 4: if(emptyStack(s))
printf("Stack Empty, count=%d\n",count);
           else{
           count=stackCount(s);
printf("The no.of elements are: %d\n",count);
           }
           break;
```

```
   case 5: disp(s->top);
          break;


   case 6: destroyStack(s);
printf("Stack destroyed\n");
          exit(0);
                // proper termination of the program
   default: printf("Invalid choice\n");
  }// end of switch
} // end of while
   return 0;
} // end of int main


void disp(STACK_NODE *top)
{
   STACK_NODE* temp=top;
   int *k;
printf("\n Stack contents are:\n");
   while(temp){
      k=(int*)temp->dataPtr;
```

```
printf("%d\t",*k);
        temp=temp->link;
    }
}
```

OUTPUT:

```
1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 3
Stack Empty

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 4
Stack Empty, count=0

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 1
Enter the item to be pushed: 10

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 1
Enter the item to be pushed: 20

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 1
Enter the item to be pushed: 30

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 5

 Stack contents are:
30      20      10
1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 4
The no.of elements are: 3

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 2
Popped item is: 30

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 3
Item at the top is: 20

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 2
Popped item is: 20

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 2
Popped item is: 10

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 2
Stack Empty
```

```
1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 1
Enter the item to be pushed: 100

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 1
Enter the item to be pushed: 200

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 1
Enter the item to be pushed: 300

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 4
The no.of elements are: 3

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 5

 Stack contents are:
300     200     100
1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 3
Item at the top is: 300

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 2
Popped item is: 300

1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 8
Invalid choice
1: Push        2: Pop        3: Stack Top        4: Stack Count        5: Display        6: Destroy Stack and Exit
Enter your choice: 6
Stack destroyed
Process returned 0 (0x0)   execution time : 74.817 s
Press any key to continue.
```

6. Conclusion:

This project implements Stack ADT. All the basic operations on stack are performed. We understood how to define all the stack functions. This project successfully imports Stack ADT in program. All member functions of the ADT are accessed in the program. The program is tested for all possible inputs, it also checks for wrong input. The required outputs are successfully generated.

References:

- Data Structures: A Pseudocode Approach with C, Second Edition- Richard F. Gilberg & Behrouz A. Forouzan