By: Ashutosh Pawan
Faculty
Department of BCA
H. D. Jain College, Ara

Stack Definition

A stack, linear data structure, is a collection of similar type of elements called nodes. Each node in the stack is divided into two parts, where the first part stores the information and the other part stores the address of next node in the stack. There is a limitation about the operations that can be performed on stack. The elements of stack can only be accessed (inserted/ removed) from one side only. This side is sometimes referred as 'top'. The insertion of an element at stack top is referred as '**push()**' and the removal of element from stack top is referred as '**pop()**'. Since the element inserted in the **last** remains at the top of the stack it will be the **first** to be removed. Hence, stack is also known as LIFO data structure. The other operations that can be performed on the stack are: '**peek()**' that displays only the topmost element in the stack. While implementing a stack using linked list another operation should be performed before exiting the program that confirms the deletions of all remaining elements in the stack. This operation is termed as '**destroy()**'.

The data structure can thus be declared as:

```
struct  <StackTypeName>

{

<informationfield>;

<address_field>;

};
```

Or for convenience the programmer can also use the syntax to declare the data structure of stack as represented below:

```
typedef  struct  <StackTypeName>

{

<informationfield>;

<address_field>;

}<NewStackTypeName>;
```

The benefit of implementing the stack in linked implementation is that its size will grow or shrink according to the operations push() and pop() performed on the stack. Also no overflow or underflow condition is required.

After declaring the data structure a stack pointer should be declared and initialized with NULL address to indicate the empty stack. This stack pointer, if the data structure is declared using 'typedef', can be declared as:

<NewStackTypeName> *<StackPointer>;

The assignment of NULL address to represent an empty stack can be done as:

<StackPointer>=(<NewStackTypeName>*) NULL;

The memory allocation for a node to be added can be done in 'C' as:

<NewStackTypeName> *<NewNodePointer>=(<NewStackTypeName>*)
malloc(sizeof(<NewStackTypeName>));

Push (stackpointer,data) operation on stack implemented in Linked implementation

Algorithm

begin

STEP 1: if <NewNodePointer>=(<NewStackTypeName>*) NULL then PROMPT "Memory Allocation Error"

STEP 2: <NewNodePointer> -> <informationfield>:=data;

STEP 3: <NewNodePointer> -> <addressfield> := <StackPointer>;

STEP 4: <StackPointer> := <NewNodePointer>;

end

Pop (stackpointer) operation on stack implemented in Linked implementation

Algorithm

begin

STEP 1: (<NewStackTypeName>) *<NodePointer> := <StackPointer>;

STEP 2: if <StackPointer> = (<NewStackTypeName>*) NULL then PROMPT "Empty Stack"

STEP 3: PROMPT information of the node;

STEP 4: <StackPointer>:= <NodePointer> -> <addressfield>;

STEP 5: free (<NodePointer>);

end

Peek (stackpointer) operation on stack implemented in Linked implementation

Algorithm

begin

STEP 1: if <StackPointer> = (<NewStackTypeName>*) NULL then PROMPT "Empty Stack"

STEP 2: PROMPT information of the node;

end

Destroy (stackpointer) operation on stack implemented in Linked implementation

Algorithm

begin

STEP 1: (<NewStackTypeName>) *<NodePointer> := <StackPointer>;

STEP 2: if <StackPointer> = (<NewStackTypeName>*) NULL then PROMPT “Empty Stack”

STEP 3: PROMPT information of the node;

STEP 4: <StackPointer>:= <NodePointer> -> <addressfield>;

STEP 5: free (<NodePointer>)

STEP 6: if <StackPointer> <> (<NewStackTypeName>*) NULL then Repeat **STEP 3**;

end

Types of Stack

Register Stack:

Being the memory element present in memory unit, it can handle only a small amount of data as its size is limited compared to the memory.

Memory Stack:

It can handle a large amount of data as its height is flexible.

Applications of Stack Data Structure:

1. Conversion of expression in or from infix, prefix and postfix notation
2. Evaluation of expression
3. Checking balanced parenthesis and curly braces
4. Function calls and recursion
5. Undo/ Redo operations
6. Browser history
7. Backtracking algorithms

etc.