STACK ADT

- A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- Stack has one end, whereas the Queue has two ends (front and rear).
- It contains only one pointer top pointer pointing to the topmost element of the stack.
- Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.
- In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

Working of Stack

- Stack works on the LIFO pattern. As we can observe in figure 3.20, there are five memory blocks in the stack; therefore, the size of the stack is 5.
- Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.
- Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.
- When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

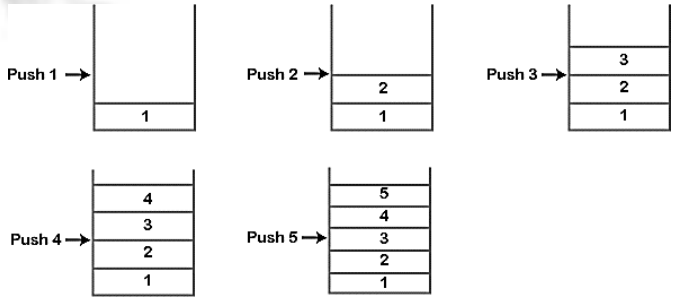


Fig. 3.20: Working principle of a Stack

Operations on Stack

The following are some common operations implemented on the stack:

- push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- isEmpty(): It determines whether the stack is empty or not.
- isFull(): It determines whether the stack is full or not.'
- peek(): It returns the element at the given position.
- count(): It returns the total number of elements available in a stack.
- change(): It changes the element at the given position.
- display(): It prints all the elements available in the stack

PUSH operation

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1**, and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the **max** size of the stack.

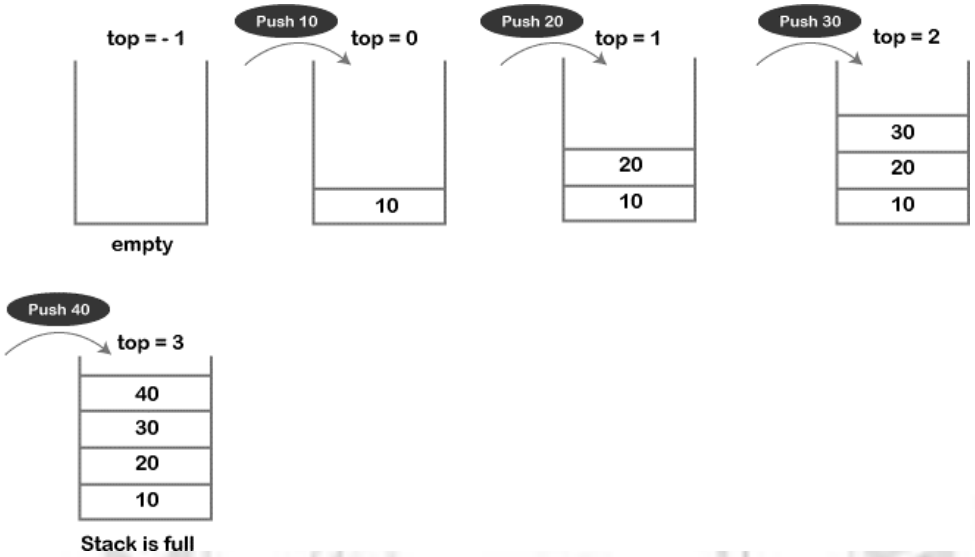
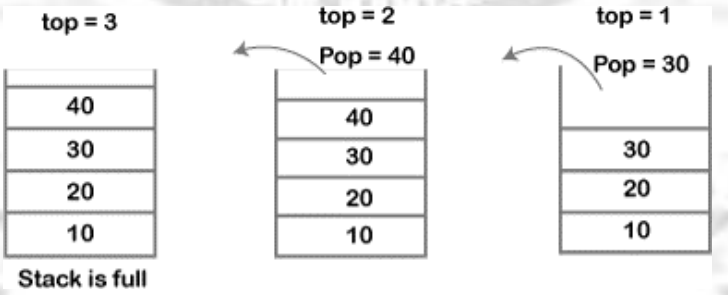


Fig. 3.21 PUSH Operation in Stack

POP operation

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the underflow condition occurs.
- If the stack is not empty, we first access the element which is pointed by the top
- Once the pop operation is performed, the top is decremented by 1, i.e., top=top-1.



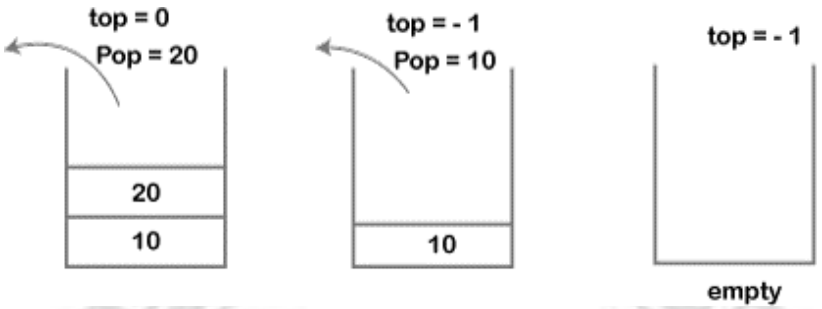


Fig. 3.22 POP Operation in Stack

Applications of Stack

- Balancing of symbols: Stack is used for balancing a symbol.
- String reversal: Stack is also used for reversing a string
- UNDO/REDO: It can also be used for performing UNDO/REDO operations.
- Recursion: The recursion means that the function is calling itself again.
- DFS(Depth First Search): This search is implemented on a Graph, and Graphuses the stack data structure.
- Backtracking: In order to come at the beginning of the path to create a new path, use the stack data structure
- Expression conversion: Stack can also be used for expression conversion.
  - Infix to prefix
  - Infix to postfix
  - Prefix to infix
  - Prefix to postfix
  - Postfix to infix
- Memory management: The stack manages the memory. The memory is assignedin the contiguous memory blocks.

IMPLEMENTATION OF STACK

- Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size.

Array implementation of Stack

- In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Let's see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

- Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.
  1. Increment the variable Top so that it can now refere to the next memory location.
  2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.
- Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm 3.15

```
begin
   if top = n then stack full
   top = top + 1
   stack (top) : = item;
end
```

Time Complexity: O (1)

3.8.1.1.1 Implementation of push algorithm in C language

```
void push (int val,int n) //n is size of the stack
{
   if (top == n )
   printf("\n Overflow");
   else
   {
   top = top +1;
   stack[top] = val;
```

```
    }
}
```

Deletion of an element from a stack (Pop operation)

- Deletion of an element from the top of the stack is called pop operation.
- The value of the variable top will be incremented by 1 whenever an item is deleted from the stack.
- The top most element of the stack is stored in another variable and then the top is decremented by 1.
- The operation returns the deleted value that was stored in another variable as the result.
- The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm 3.16

```
begin
   if top = 0 then stack empty;
   item := stack(top);
   top = top - 1;
end;
```

Time Complexity: O (1)

3.8.1.2.1 Implementation of POP algorithm using C language

```
int pop ()
{
   if(top == -1)
   {
      printf("Underflow");
      return 0;
   }
   else
   {
```

```
        return stack[top - - ];
    }
}
```

Visiting each element of the stack (Peek operation)

- Peek operation involves returning the element which is present at the top of the stack without deleting it.
- Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm 3.17

```
PEEK (STACK, TOP)
    Begin
      if top = -1 then stack empty
      item = stack[top]
      return item
    End
```

Time complexity: O(n)

Implementation of Peek algorithm in C language

```
int peek()
{
  if (top == -1)
  {
    printf("Underflow");
    return 0;
  }
  else
  {
    return stack [top];
  }
}
```

Linked list implementation of stack

- Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.
- In linked list implementation of stack, the nodes are maintained non-contiguously in the memory.
- Each node contains a pointer to its immediate successor node in the stack.
- Stack is said to be overflown if the space left in the memory heap is not enough to create a node.
- The top most node in the stack always contains null in its address field.

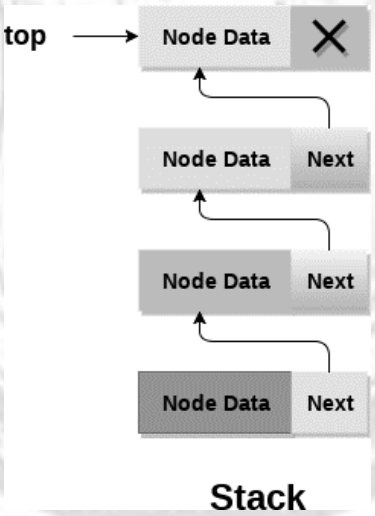


Fig. 3.23: Linked List implementation of Stack

Adding a node to the stack (Push operation)

- Adding a node to the stack is referred to as push operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.
  - Create a node first and allocate memory to it.
  - If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.

- If there are some nodes in the list already, then add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new nodeand make the new node, the starting node of the list.

  Time Complexity: O (1)

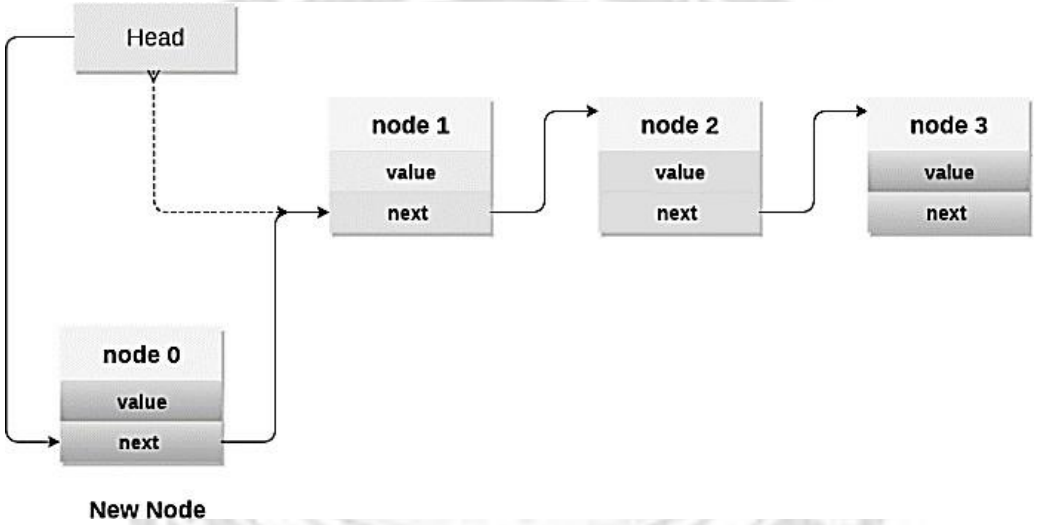


Fig 3.24 Adding new Node to Stack

3.8.2.1.1 Implementation of PUSH in C Language Program

```
void push ()
{
   int val;
   struct node *ptr =(struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("not able to push the element");
   }
   else
   {
      printf("Enter the value");
      scanf("%d",&val);
```

```
        if(head==NULL)
        {
           ptr->val = val;
           ptr -> next = NULL;
           head=ptr;
        }
        else
        {
           ptr->val = val;
           ptr->next =
           head;head=ptr;
        }
        printf("Item pushed");
     }
}
```

Deleting a node from the stack (POP operation)

- Deleting a node from the top of stack is referred to as pop operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation.
- In order to pop an element from the stack, do the following steps:
  - **Check for the underflow condition**: The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
  - **Adjust the head pointer accordingly**: In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

    Time Complexity: O (n)

3.8.2.2.1 Implementation of POP in C Language Program

```
void pop()
{
   int item;
   struct node *ptr;
   if (head == NULL)
   {
      printf("Underflow");
   }
   else
   {
      item = head->val;
      ptr = head;
      head = head->next;
      free(ptr);
      printf("Item popped");
   }
}
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this do the following steps.

- Copy the head pointer into a temporary pointer.
- Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

  Time Complexity: O (n)

3.8.2.3.1 Implementation of Display in C Language Program

```
void display()
{
   int i;
```

```
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
       printf("Stack is empty\n");
    }
    else
    {
       printf("Printing Stack elements \n");
       while(ptr!=NULL)
       {
          printf("%d\n",ptr->val);
          ptr = ptr->next;
       }
    }
}
```

APPLICATIONS OF STACK

- A Stack is a widely used linear data structure in modern computers in which insertions and deletions of an element can occur only at one end, i.e., top of the Stack. It is used in all those applications in which data must be stored and retrieved in the last.
- Following are the various Applications of Stack in Data Structure:
  - Evaluation of Arithmetic Expressions
  - Balancing Symbols
  - Processing Function Calls
  - Backtracking
  - Reverse a Data

Evaluation of Arithmetic Expressions

- A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.
- In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis".

Example 3.3: A + (B - C)

To evaluate the expressions, one needs to be aware of the standard precedence rules for arithmetic expression. The precedence rules for the five basic arithmetic operators are:

Operators	Associativity	Precedence
^ exponentiation	Right to left	Highest followed by *Multiplication and /division
*Multiplication, /division	Left to right	Highest followed by + addition and - subtraction
+ addition, - subtraction	Left to right	Lowest

Evaluation of Arithmetic Expression requires two steps:

- First, convert the given expression into special notation.
- Evaluate the expression in this new

notation.Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

- Infix Notation
- Prefix Notation
- Postfix Notation

Infix Notation

The infix notation is a convenient way of writing an expression in which each operator is placed between the operands. Infix expressions can be parenthesized or un parenthesized depending upon the problem requirement.

Example 3.4: A + B, (C - D) etc.

All these expressions are in infix notation because the operator comes between the operands.

Prefix Notation

The prefix notation places the operator before the operands. This notation was introduced by the Polish mathematician and hence often referred to as polish notation.

Example 3.5: + A B, -CD etc.

All these expressions are in prefix notation because the operator comes before the operands.

Postfix Notation

The postfix notation places the operator after the operands. This notation is just the reverse of Polish notation and also known as Reverse Polish notation.

Example 3.6: AB +, CD+, etc.

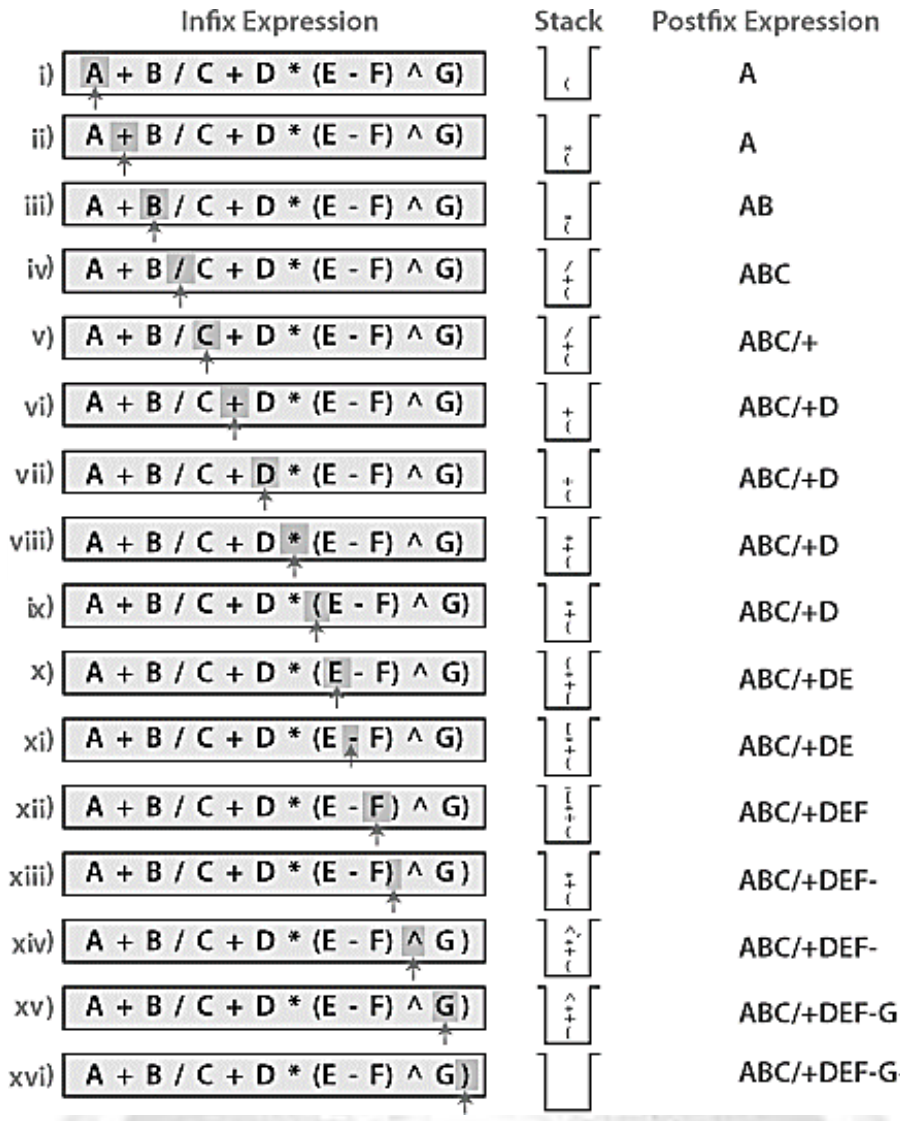
All these expressions are in postfix notation because the operator comes after the operands.

Table 3.3 illustrates the conversion of Arithmetic Expression into various Notations

Table 3.3: Conversion of Arithmetic Expression into various Notations

Infix Notation	Prefix Notation	Postfix Notation
A * B	* A B	AB*
(A+B)/C	/+ ABC	AB+C/
(A*B) + (D-C)	+*AB - DC	AB*DC-+

Let's take the example of Converting an infix expression into a postfix expression



In the above example, the only change from the postfix expression is that the operator is placed before the operands rather than between the operands.

Evaluating Postfix expression

- Stack is the ideal data structure to evaluate the postfix expression because the top element is always the most recent operand. The next element on the Stack is the second most recent operand to be operated on.
- Before evaluating the postfix expression, the following conditions must be checked. If any one of the conditions fails, the postfix expression is invalid.

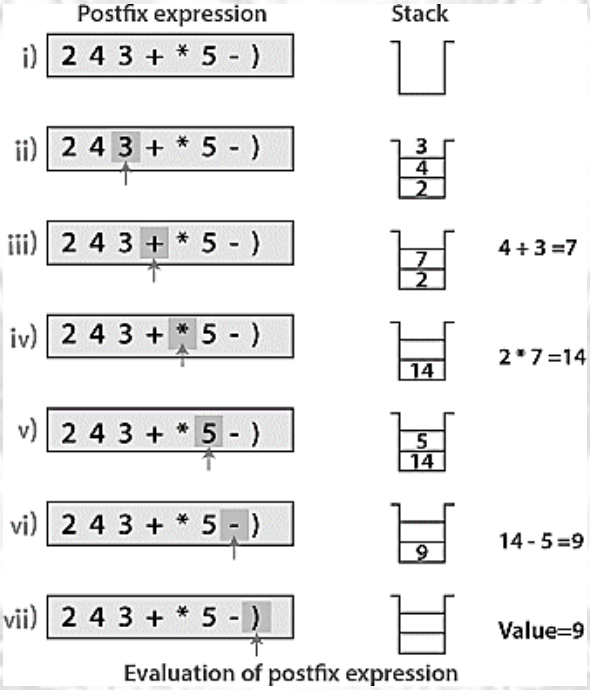
- When an operator encounters the scanning process, the Stack must contain a pair of operands or intermediate results previously calculated.
- When an expression has been completely evaluated, the Stack must contain exactly one value.

Example 3.7

Now let us consider the following infix expression 2 * (4+3) - 5.

Its equivalent postfix expression is 2 4 3 + * 5.

The following step illustrates how this postfix expression is evaluated.



Evaluation of postfix expression

Balancing Symbols

- Stacks can be used to check if the given expression has balanced symbols or not.
- The algorithm is very much useful in compilers.
- Each time parser reads one character at a time.
- If the character is an opening delimiter like '(' , '{' or '[' then it is PUSHED in to the stack.
- When a closing delimiter is encountered like ')' , '}' or ']' is encountered, the stack is popped.

- The opening and closing delimiter are then compared.
- If they match, the parsing of the string continues.
- If they do not match, the parser indicates that there is an error on the

line.A linear time O(n) algorithm based on stack can be given as:-

Create a stack.

while (end of input is not reached) {

If the character read is not a symbol to be balanced, ignore it.

If the character is an opening delimiter like (, { or [, PUSH it into the stack.

If it is a closing symbol like) , } ,] , then if the stack is empty report an error, otherwise POP the stack.

If the symbol POP-ed is not the corresponding delimiter, report an error.

At the end of the input, if the stack is not empty report an error.

Example 3.8

EXAMPLE	Valid?	Description
(A+B) + (C-D)	Yes	The expression is having balanced symbol
((A+B) + (C-D)	No	One closing brace is missing.
((A+B) + [C-D])	Yes	Opening and closing braces correspond
((A+B) + [C-D]]	No	The last brace does not correspond with the first opening brace.
(A+B) + (C-D)	Yes	The expression is having balanced symbol

For tracing the algorithm let us assume that the input is () (() [()])

Input Symbol	Operation	Stack	Output
(	Push (	(	
)	Pop (Test if (and A[i] match? YES		
(	Push (	(	

(	Push (	((	
)	Pop (Test if (and A[i] match? YES	(	
[	Push [	([	
(	Push (	([(	
)	Pop) Test if (and A[i] match? YES	)	
]	Pop [Test if [and A[i] match? YES	(	
)	Pop (Test if (and A[i] match? YES		
	Test if Stack is Empty? YES		TRUE

Processing Function Calls

- Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.

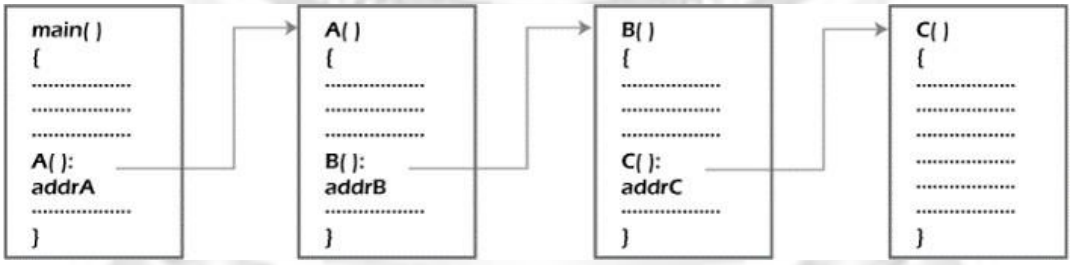


Fig. 3.24: Invoking Function Calls

- When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed.

- Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.
- Consider addrA, addrB, addrC be the addresses of the statements to which control is returned after completing the function A, B, and C, respectively.

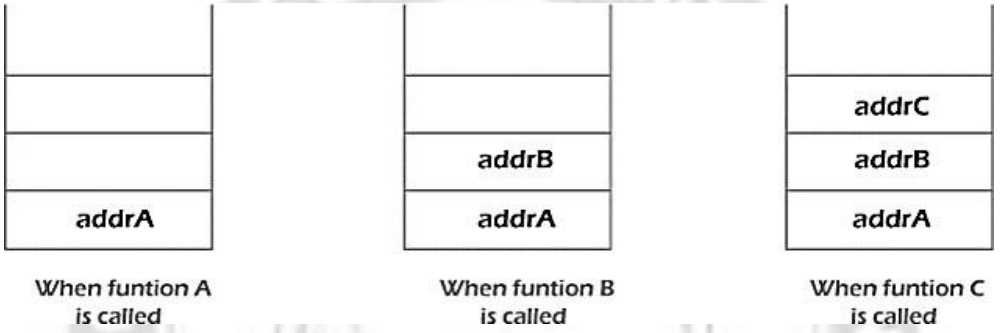


Fig. 3.25: Different states of stack

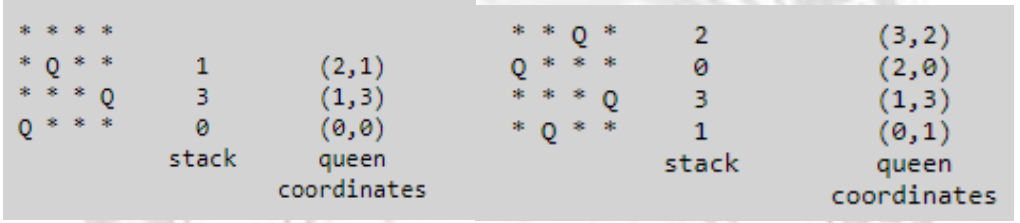
- Figure 3.26 shows that return addresses appear in the Stack in the reverse order in which the functions were called. After each function is completed, the pop operation is performed, and execution continues at the address removed from the Stack.
- Thus the program that calls several functions in succession can be handled optimally by the stack data structure. Control returns to each function at a correct place, which is the reverse order of the calling sequence.

Backtracking

- Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.
- Let's see how Stack is used in Backtracking in the N-Queens Problem
- For the N-Queens problem, one way we can do this is given by the following:
  - For each row, place a queen in the first valid position (column), and then move to the next row
  - If there is no valid position, then one backtracks to the previous row and try the next position

- If one can successfully place a queen in the last row, then a solution is found. Now backtrack to find the next solution

- We can use a stack to indicate the positions of the queens. Importantly, notice that we only have to put the column positions of the queens on the stack. We can determine each queen's coordinates given only the stack. We simply combine the position of an element in the stack (the row) with the value of that element (the column) for each queen.
- Two examples of this are shown below:



- Starting with a queen in the first row, first column (represented by a stack containing just "0"), we search left to right for a valid position to place another queen in the next available row.
- If we find a valid position in this row, we push this position (i.e., the column number) to the stack and start again on the next row.
- If we don't find a valid position in this row, we backtrack to the previous row -- that is to say, we pop the col position for the previous row from the stack and search for a valid position further down the row.
- Note, when the stack size gets to n, we will have placed n queens on the board, and therefore have a solution.
- Of course, there is nothing that requires there be only one solution. To find the rest, every time a solution is found, we can pretend it is not a solution, backtrack to the previous row, and proceed to find the next solution.
- Ultimately, every position in the first row will be considered. When there are not more valid positions in the first row and we need to backtrack, that's our cue that there are no more solutions to be found. Thus, we may stop searching when we try to pop from the stack, but can't as it is empty.
- Putting all this into pseudo-code form, we have the following algorithm...

```
Create empty stack and set current position to 0
    Repeat {
      loop from current position to the last position until valid position found //current row
      if there is a valid position {
        push the position to stack, set current position to 0 // move to next row
      }
      if there is no valid position {
        if stack is empty, break // stop search
        else pop stack, set current position to next position // backtrack to previous row
      }
      if stack has size N { // a solution is found
        pop stack, set current position to next position // backtrack to find next solution
      }
    }
```

Reverse a Data

- To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements. For example, suppose we have a string “welcome”, then on reversing it would be “emoclew”.
- There are different reversing applications:
  - Reversing a string
  - Converting Decimal to Binary

Reverse a String

- A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the last in first out property of the Stack, the first character of the Stack is on the bottom of the Stack and the last

character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.

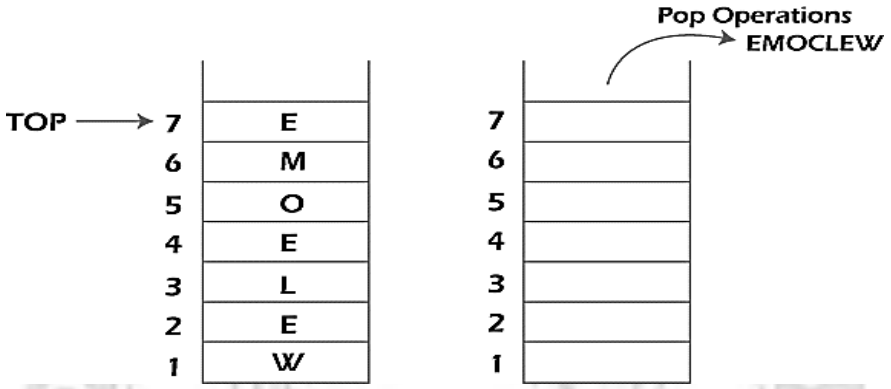


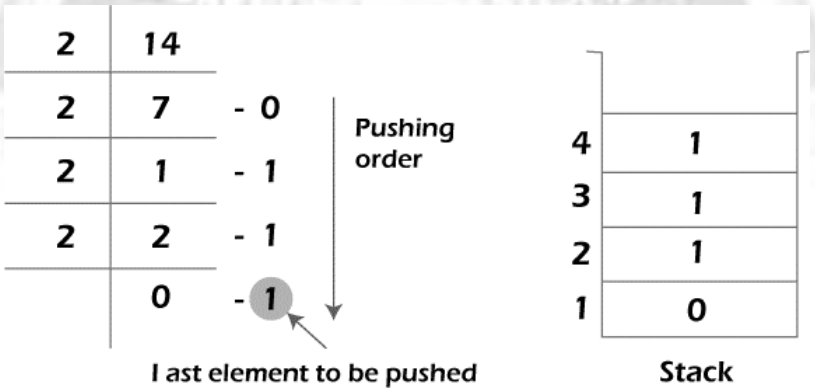
Fig. 3.26: Reverse a String using Stack

Converting Decimal to Binary

- Although decimal numbers are used in most business applications, some scientific and technical applications require numbers in either binary, octal, or hexadecimal. A stack can be used to convert a number from decimal to binary/octal/hexadecimal form.
- For converting any decimal number to a binary number, we repeatedly divide the decimal number by two and push the remainder of each division onto the Stack until the number is reduced to 0. Then we pop the whole Stack and the result obtained is the binary equivalent of the given decimal number.

Example: 3.9:

Converting 14 number Decimal to Binary:



- In the above example, on dividing 14 by 2, we get seven as a quotie'nt and one asthe reminder, which is pushed on the Stack. On again dividing seven by 2, we getthree as quotient and 1 as the reminder, which is again pushed onto the Stack. This process continues until the given number is not reduced to 0. When we pop off the Stack completely, we get the equivalent binary number 1110.