B. Tech. I CSE (Sem-2)
Data Structures
CS102

Stack: A Data Structure

*Some slides are kept with logical or syntax error. So, if you are absent in theory class, please also refer the slides provided at the end of the presentation.

Dr. Rupa G. Mehta
Dr. Dipti P. Rana
Department of Computer Science and Engineering
SVNIT, Surat

Outline…

- Stack
  - Basic principles
  - Operation of stack
  - Stack using Array
  - Stack using Linked List
  - Applications of stack

Basic Idea

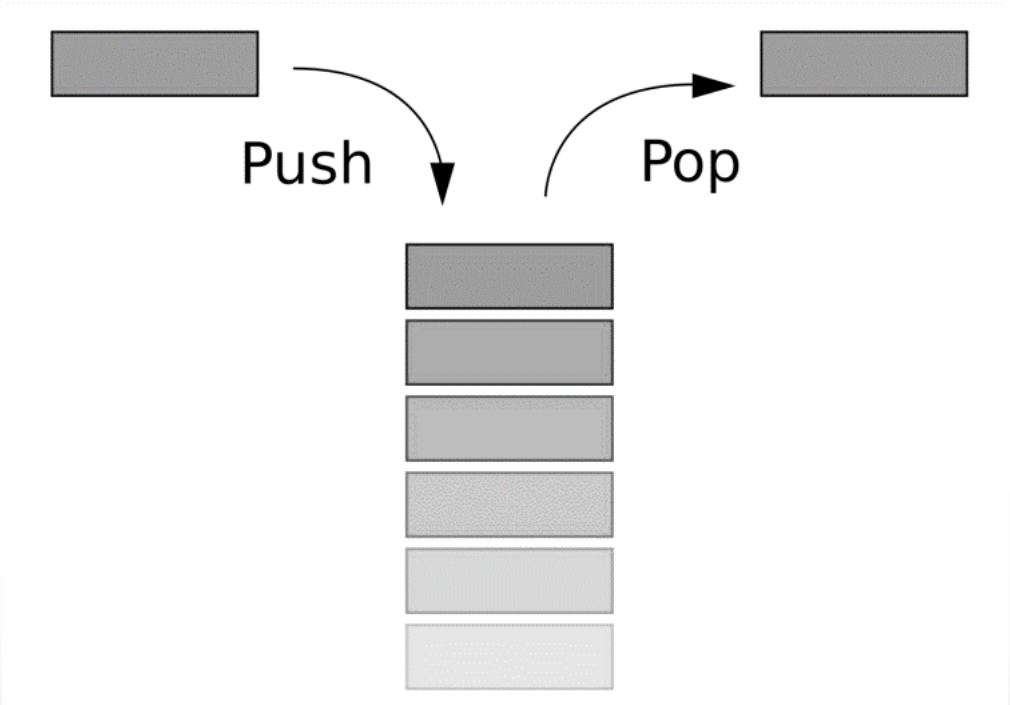
- A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the top of the stack
- It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

- Commonly used in most programming languages

Basic Idea

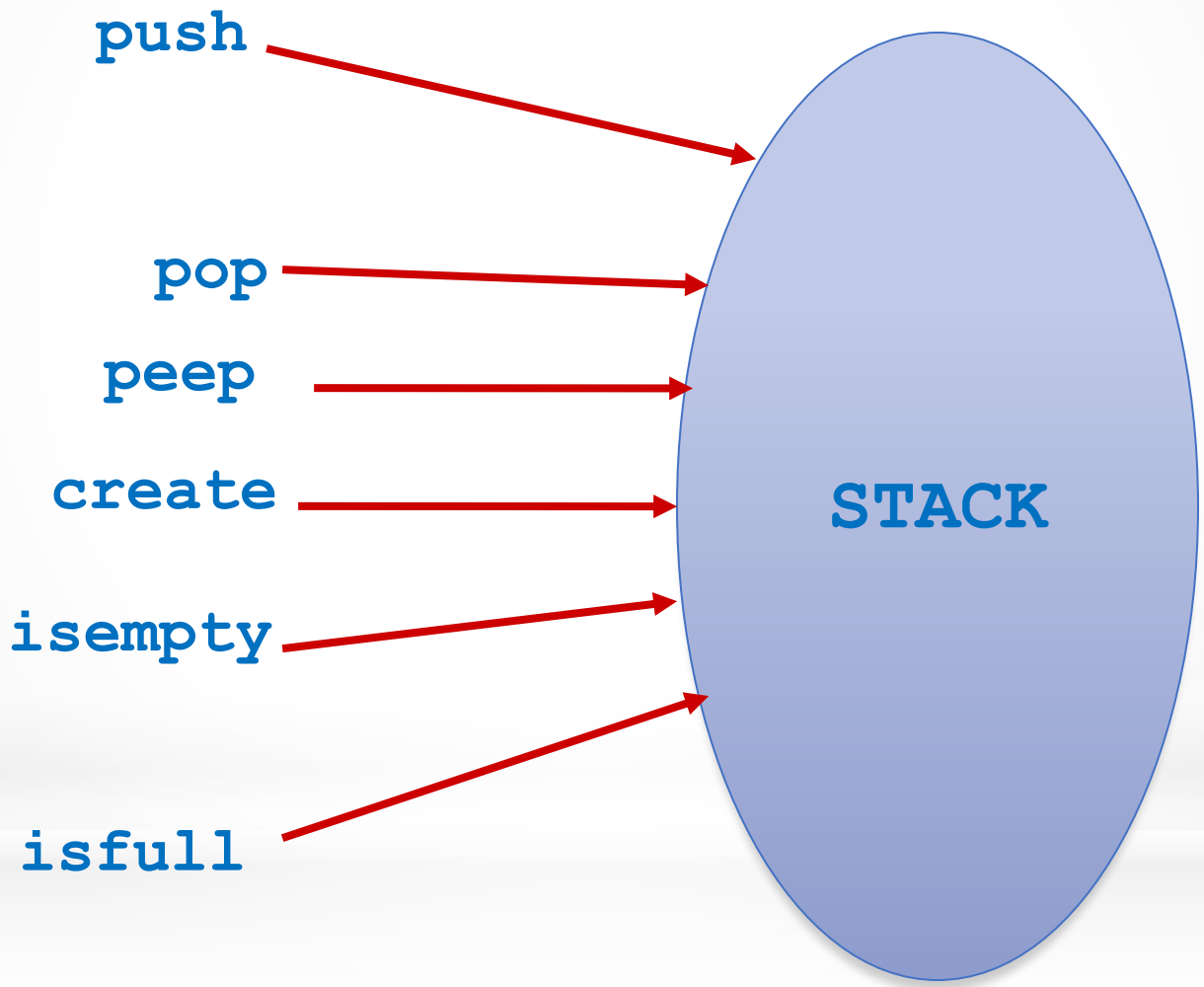
- A stack is a linear data structure which satisfies the following properties at any time:
  - Allocations and de-allocations are performed in a last-in-first-out (LIFO) manner.
    - i.e. amongst all existing entries at any time, the last entry to have been allocated is the first entry to be de-allocated
  - only the last entry is accessible

Stack Representation



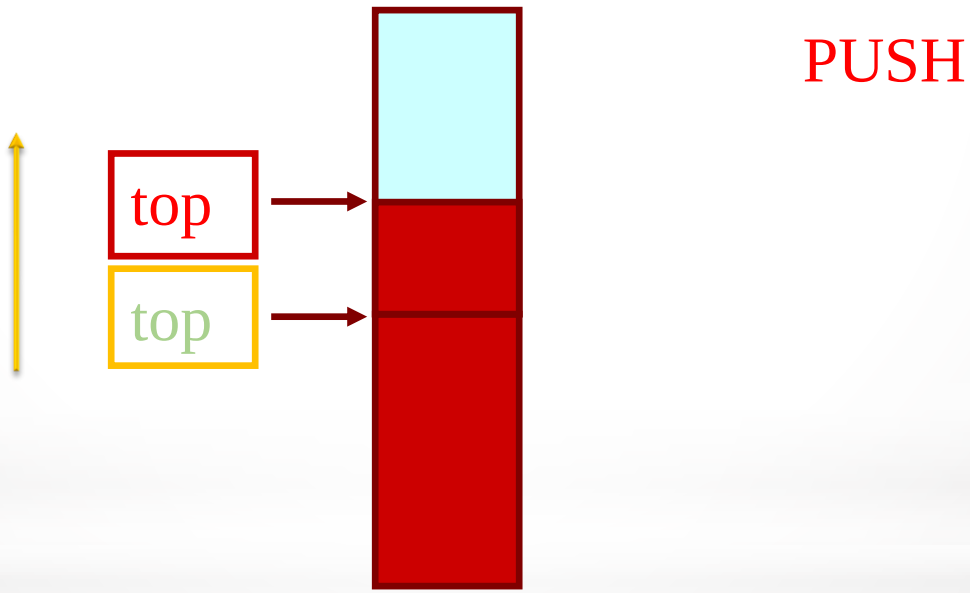
- Can be implemented by means of Array, Structure, Pointers and Linked List.
- Stack can either be a fixed size or dynamic.

Stack Operations



Stack using Array

Push using Stack



Push using Stack

- Before executing the **PUSH** operation, one must test if there is a room in the stack for new item or not
- Stack-full condition is called **"overflow"**

Push Operation

PUSH(STACK, TOP, MAXSTK, ITEM)

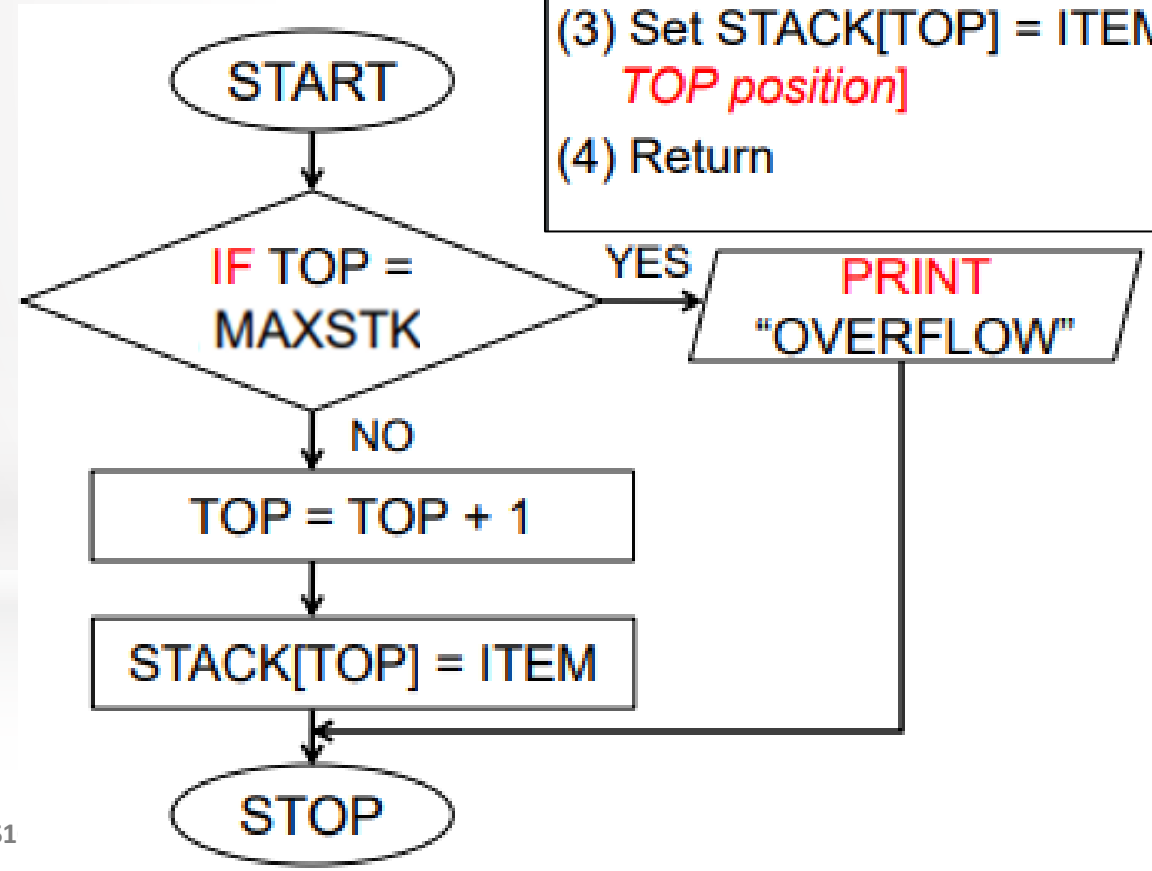
This procedure pushes an ITEM onto a stack

(1) If TOP = MAXSTK, then: Print: OVERFLOW, and Return [*stack already filled ?*]

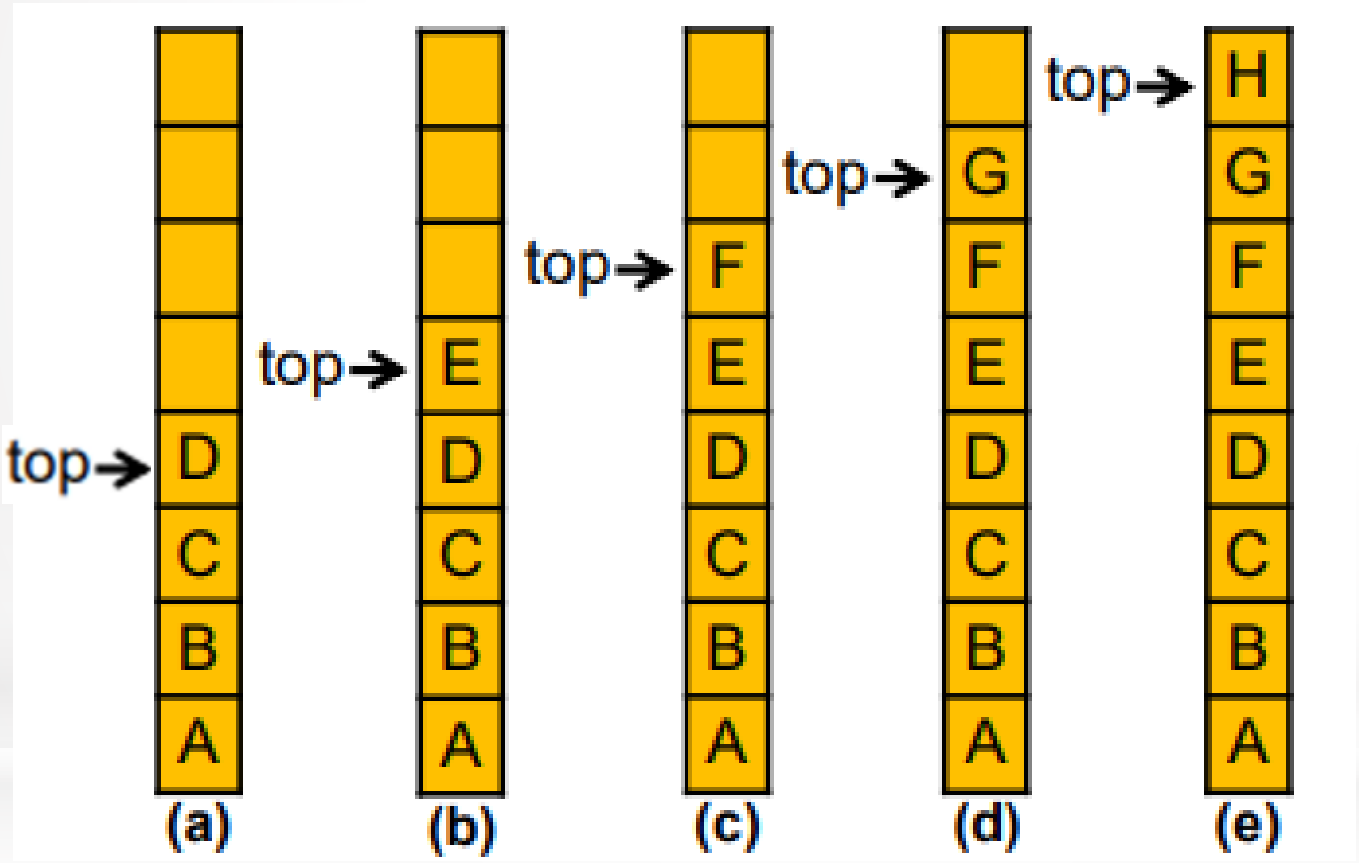
(2) Set TOP = TOP+1 [*increases TOP by 1*]

(3) Set STACK[TOP] = ITEM [*Inserts ITEM in new TOP position*]

(4) Return

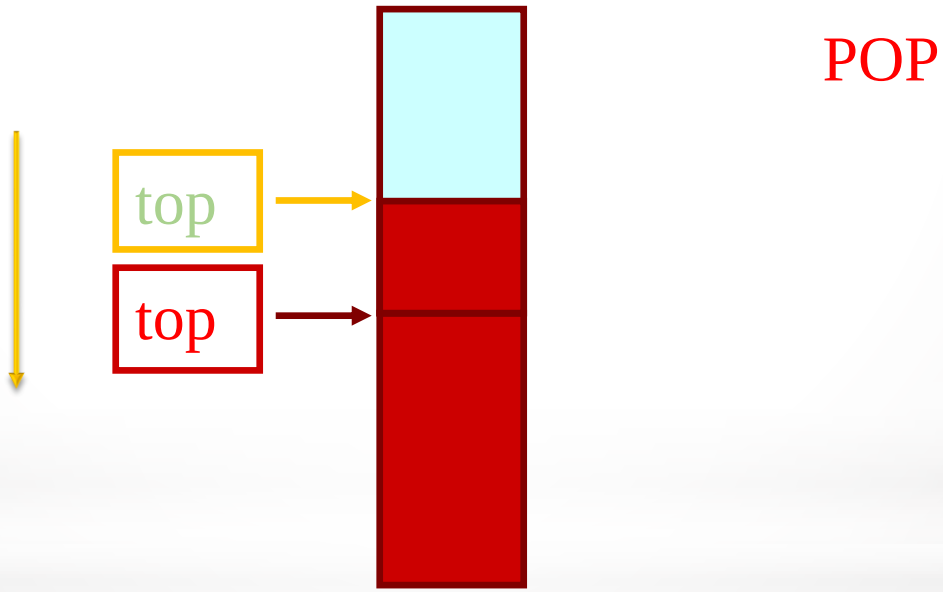


Push using Stack



Stack Overflow for I

Pop using Stack



Pop using Stack

- Before executing the POP operation, one must test whether there is an element in the stack to be deleted or not
- Empty stack condition is called **"underflow"**

Pop using Stack

POP(STACK, TOP, ITEM)

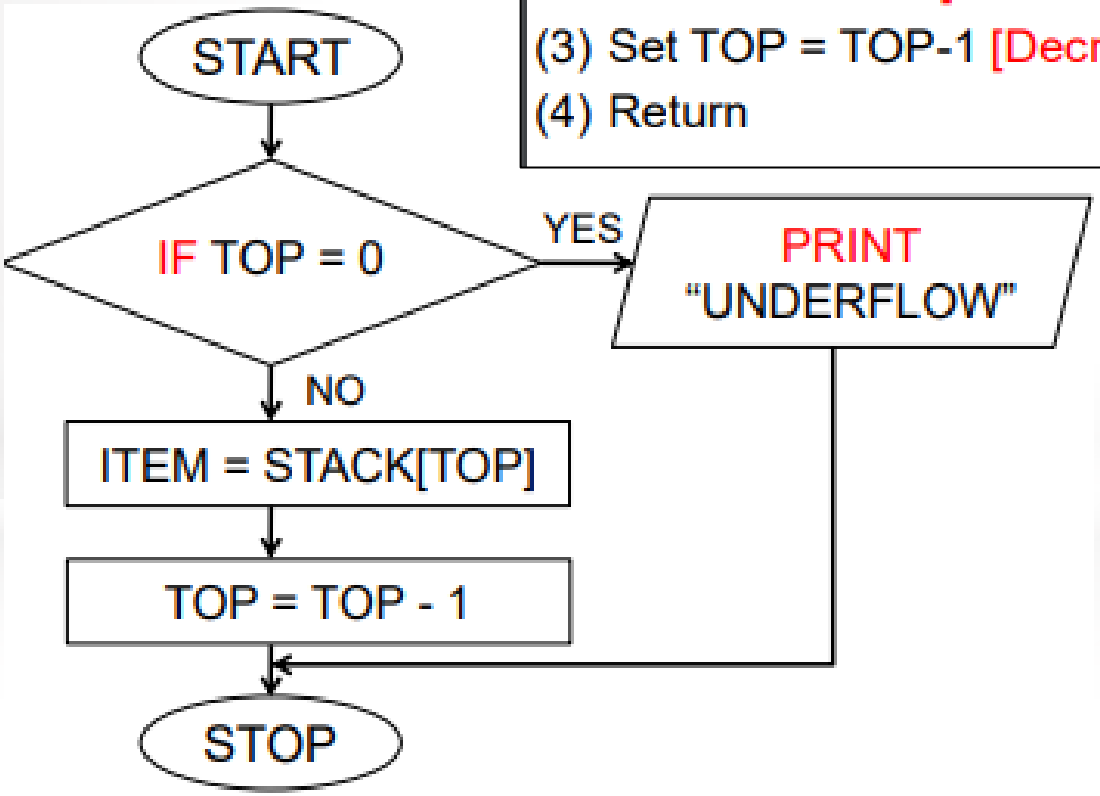
This procedure deletes the top element of stack and assigns it to the variable ITEM

(1) If TOP = 0, then: Print: UNDERFLOW, and Return [*stack empty ?*]

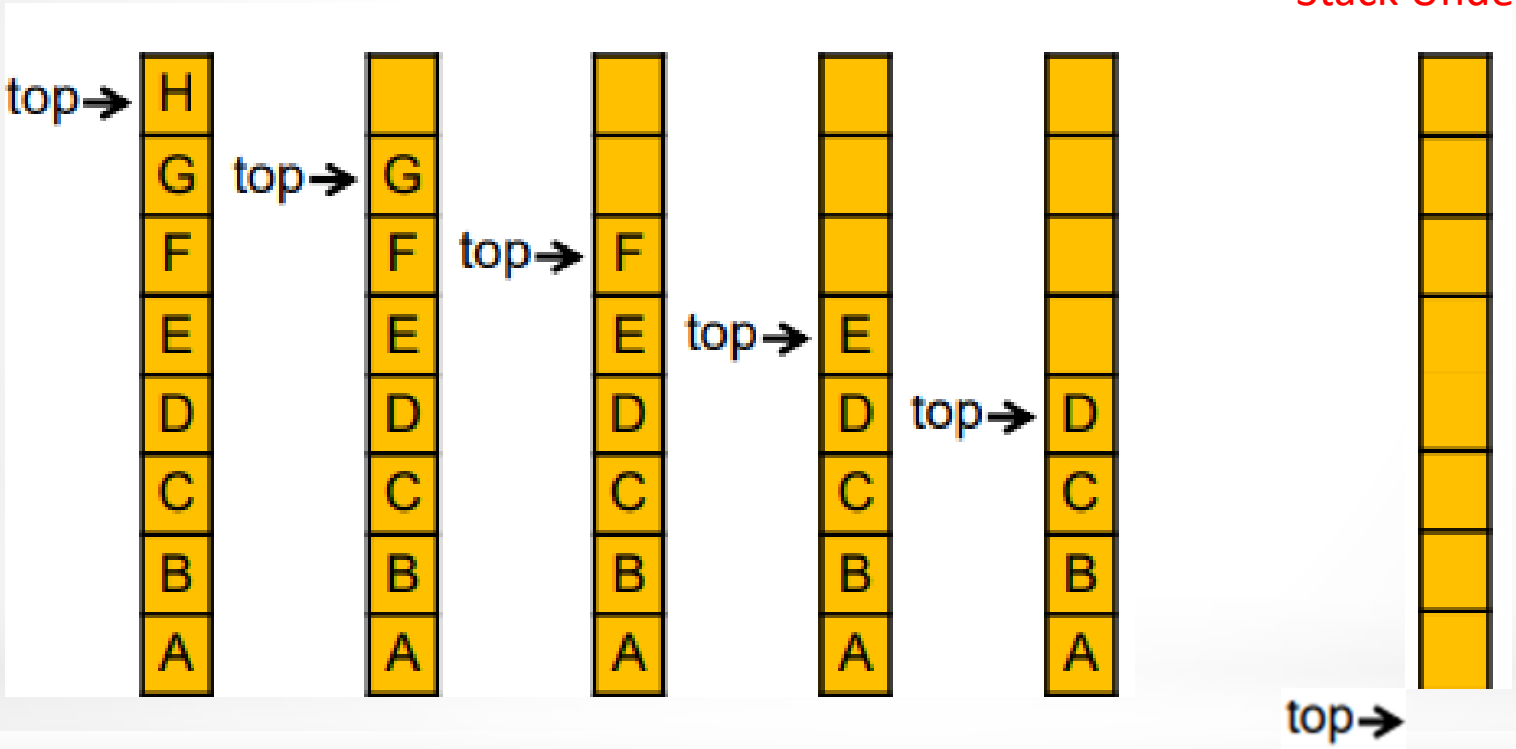
(2) Set ITEM = STACK[TOP] [Assign TOP element to ITEM]

(3) Set TOP = TOP-1 [Decreases *TOP by 1*]

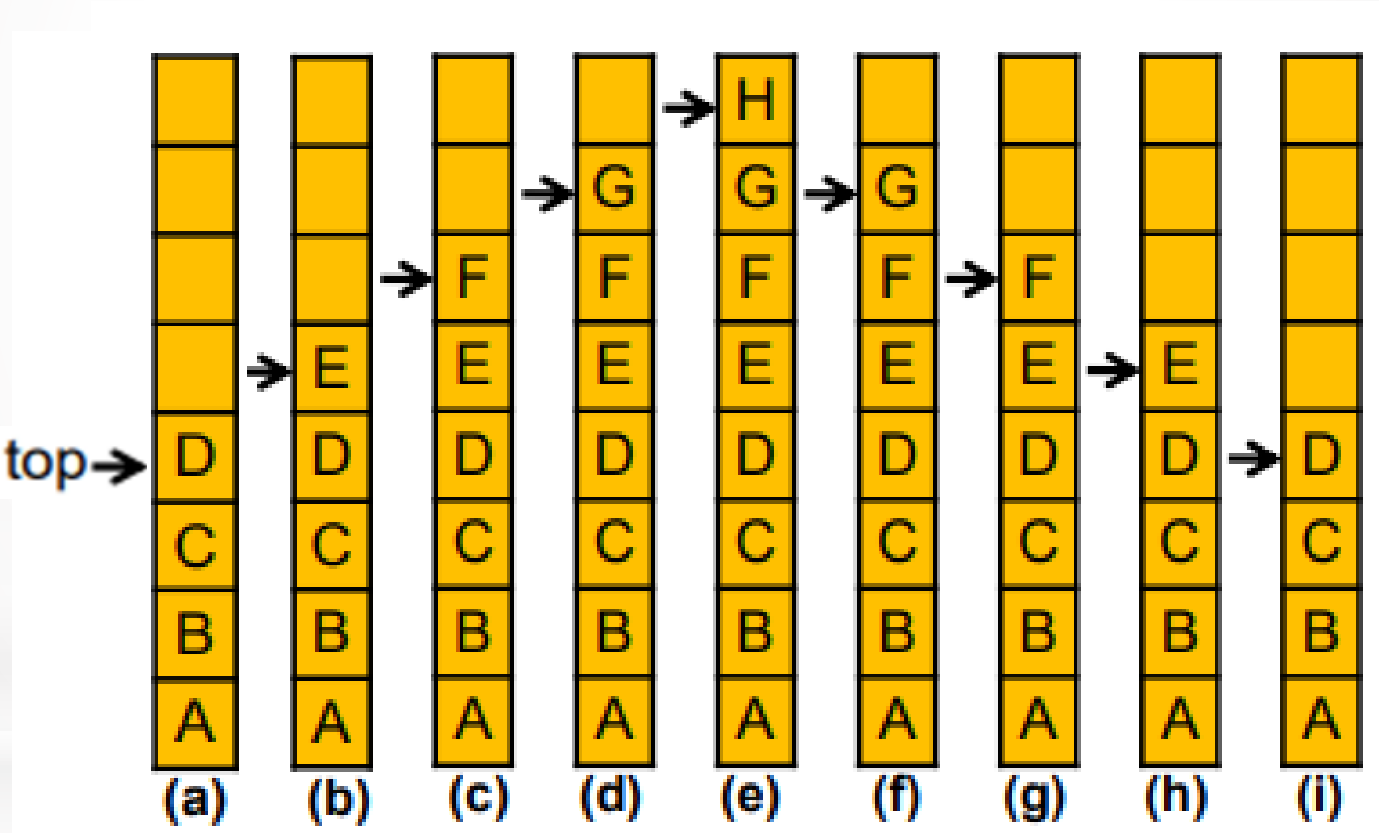
(4) Return



Pop using Stack



Operations on Stack



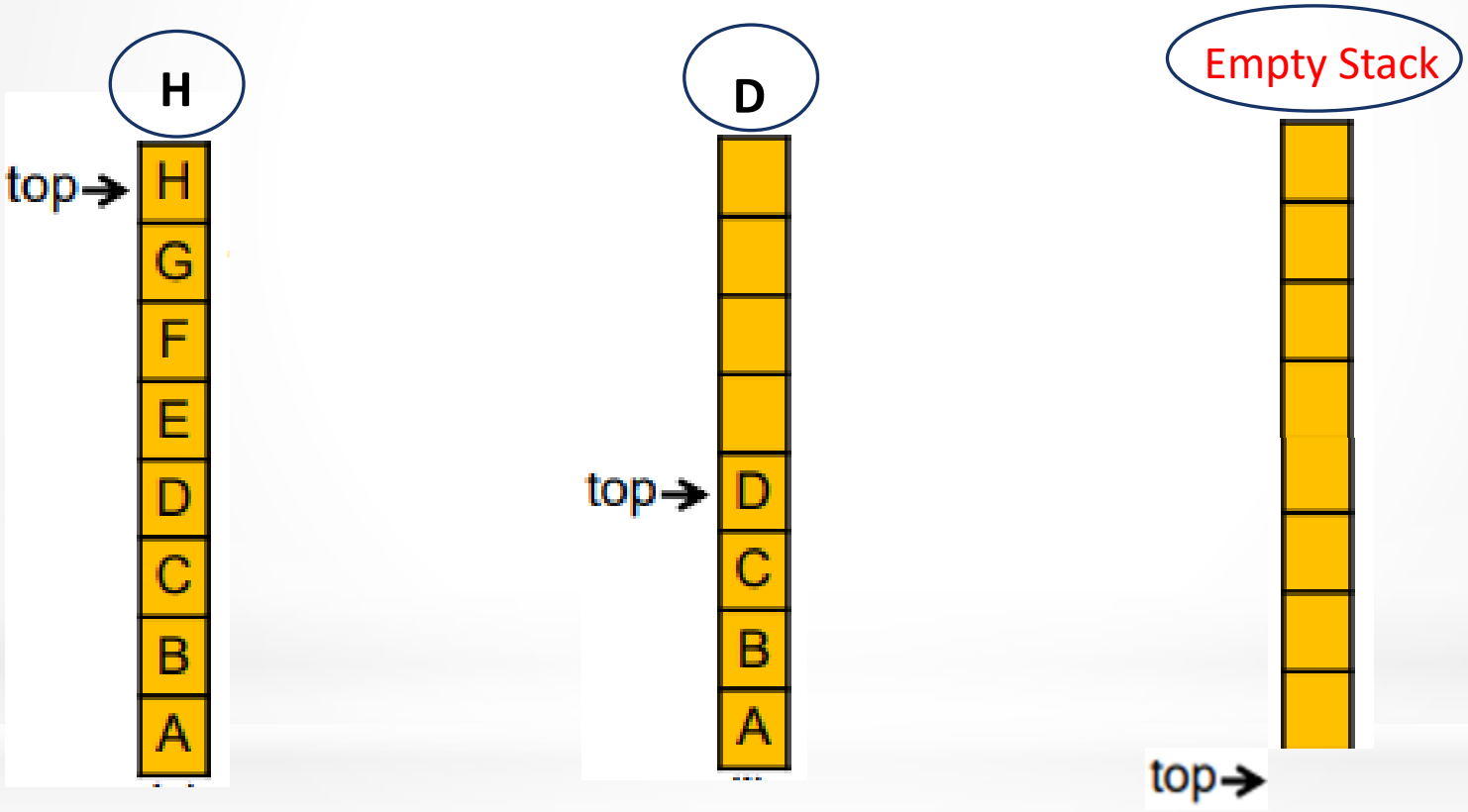
Peek Operation

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack

```
IF TOP = 0  then
    Print: Stack is Empty
    Return
Else
    Print STACK[TOP]
    Return
```

- The Peek operation will return 5, as it is the value of the topmost element of the stack.

Peep using Stack

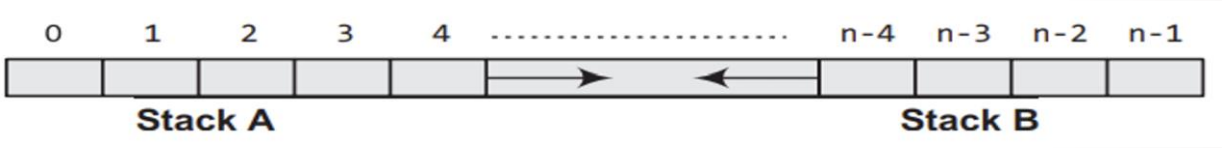


Multiple Stacks

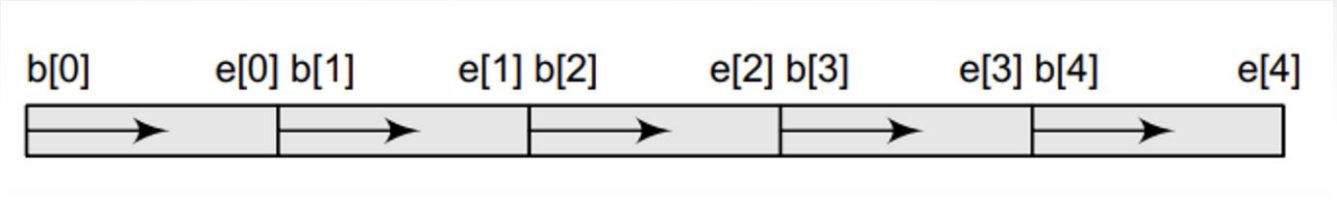
- The size of the array must be known in advance.
- If the stack is allocated less space, then frequent OVERFLOW conditions will be encountered.
- To deal with this problem, the code will have to be modified to reallocate more space for the array.
- In case we allocate a large amount of space for the stack, it may result in sheer wastage of memory. Thus, there lies a trade-off between the frequency of overflows and the space allocated.
- So, a better solution to deal with this problem is to have multiple stacks or to have more than one stack in the same array of sufficient size.

Multiple Stacks

- An array STACK[n] is used to represent two stacks, Stack A and Stack B.
- The value of n is such that the combined size of both the stacks will never exceed n.



- While operating these stacks, it is important to note one thing—Stack A will grow from left to right, whereas Stack B will grow from right to left at the same time.
- Extending this concept to multiple stacks, a stack can also be used to represent n number of stacks in the same array.
- That is, if we have a STACK[n], then each stack I will be allocated an equal amount of space bounded by indices b[i] and e[i].



Basic Idea of Array-based Stack

- In the array implementation, we would:
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the “top” of the stack.
    - Contains the array index of the “top” element.

Performance and Limitations (Array-based implementation of stack)

- Performance
  - Let n be the number of elements in the stack
  - The space used is O(n)
  - Each operation runs in time O(1)
- Limitations
  - The maximum size of the stack must be defined a priori, and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Performance and Limitations (Array-based implementation of stack)

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  1. Incremental strategy
     - Increase the size by a constant c
  2. Doubling strategy
     - Double the size

Stack using Call by Reference

STACK: Last-In-First-Out (LIFO)

- `void push (stack *s, int element);`

  /* Insert an element in the stack */
- `int pop (stack *s);`

  /* Remove and return the top element */
- `void create (stack  *s);`

  /* Create a new stack */
- `int isempty (stack *s);`

  /* Check if stack is empty */
- `int isfull (stack *s);`

  /* Check if stack is full */

Assumption: stack contains integer elements!

Declaration

```
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int top;
};
typedef struct lifo stack;
stack s;
```

ARRAY

Stack Creation

```
void create (stack *s)
{
   s->top = -1;

   /* s->top points to
      last element
      pushed in;
      initially -1 */
}
```

ARRAY

Pushing an element into stack

```
void push (stack *s, int element)
  {
   if (s->top == (MAXSIZE-1))
   {
     printf ("\n Stack overflow");
         exit(-1);
     }
     else
     {
         s->top++;
         s->st[s->top] = element;
     }
  }
```

ARRAY

Popping an element from stack

```
int pop (stack *s)
  {
     if (s->top == -1)
     {
        printf ("\n Stack underflow");
        exit(-1);
     }
     else
     {
        return (s->st[s->top--]);
     }
  }
```

ARRAY

Checking for stack empty

```
int isempty (stack *s)
{
    if (s->top == -1)
            return 1;
    else
            return (0);
}
```

ARRAY

Checking for Stack Full

```
int isfull (stack *s)
{
    if (s->top == ?)
            return 1;
    else
            return (0);
}
```

ARRAY

Example: A Stack using an Array

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo stack;

void main() {
  stack A, B;
  create(&A);
  create(&B);
  push(&A,10);
  printf("\n%d top=%d Stack A [top]=%d", sizeof(A), A.top, A.st[A.top]);
  push(&A,20);
   printf("\n%d top=%d Stack A [top]=%d", sizeof(A), A.top, A.st[A.top]);
  push(&A,30);
   printf("\n%d top=%d Stack A [top]=%d", sizeof(A), A.top, A.st[A.top]);
  push(&B,100);
   printf("\n%d top=%d Stack B [top]=%d", sizeof(B), B.top, B.st[B.top]);
  push(&B,5);
   printf("\n%d top=%d Stack B [top]=%d", sizeof(B), B.top, B.st[B.top]);

  printf ("%d %d", pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(&B))
    printf ("\n B is empty");
  return;
}
```

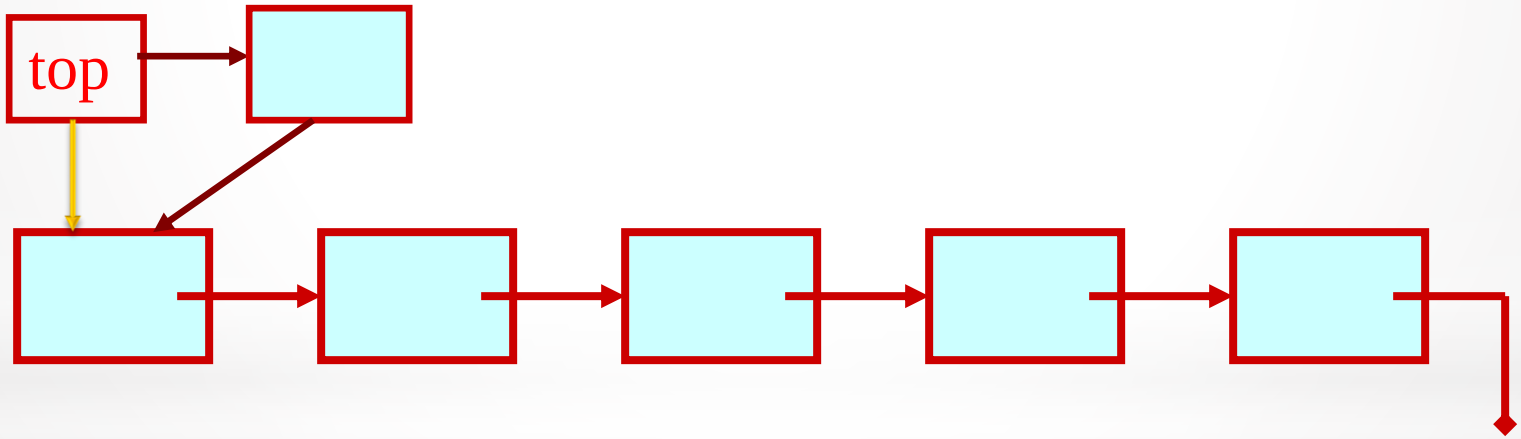
```
404 top=0 Stack A [top]=10
404 top=1 Stack A [top]=20
404 top=2 Stack A [top]=30
404 top=0 Stack B [top]=100
404 top=1 Stack B [top]=5
30 5
 B is empty
```

Applications of Stack

Stack using Linked List

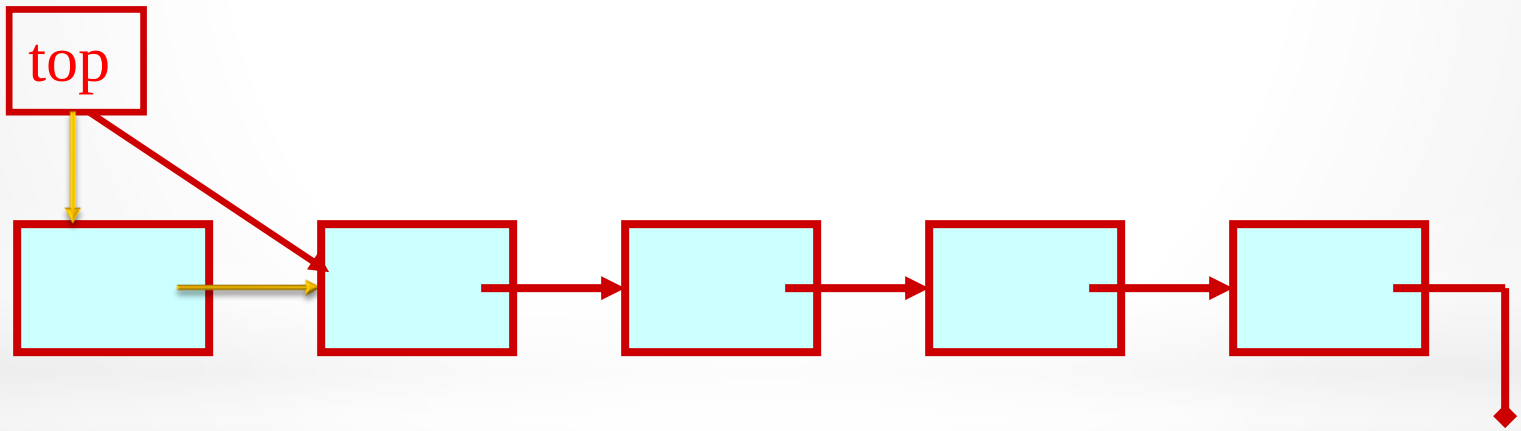
Push using Linked List

PUSH OPERATION



Pop using Linked List

POP OPERATION



Basic Idea

- In the array implementation, we would:
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the “top” of the stack.
    - Contains the array index of the “top” element.
- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable top points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Declaration

```
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo stack;
stack s;
```

ARRAY

```
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo stack;

stack *top;
```

LINKED LIST

Stack Creation

```
void create (stack *s)
{
   s->top = -1;

   /* s->top points to
      last element
      pushed in;
      initially -1 */
}
```

ARRAY

```
void create (stack **top)
{
   *top = NULL;

   /* top points to NULL,
      indicating empty
      stack              */
}
```

LINKED LIST

Pushing an element into stack

```
void push (stack *s, int
element)
  {
   if (s->top == (MAXSIZE-1))
   {
     printf ("\n Stack
overflow");
         exit(-1);
     }
     else
     {
         s->top++;
         s->st[s->top] =
element;
     }
  }
```

ARRAY

```
void push (stack **top, int element)
{
    stack *new;

    new = (stack *)malloc (sizeof(stack));
     if (new == NULL)
     {
       printf ("\n Stack is full");
        exit(-1);
     }

     new->value = element;
     new->next = *top;
     *top = new;
}
```

LINKED LIST

Popping an element from stack

```
int pop (stack *s)
  {
     if (s->top == -1)
     {
        printf ("\n Stack
underflow");
        exit(-1);
     }
     else
     {
        return (s->st[s->top-
-]);
     }
  }
```

ARRAY

```
int pop (stack **top)
{
    int t;
    stack *p;

    if (*top == NULL)
    {
       printf ("\n Stack is empty");
       exit(-1);
    }
    else
    {
       t = (*top)->value;
       p = *top;
       *top = (*top)->next;
       free (p);
       return t;
    }
}
```

LINKED LIST

Checking for stack empty

```
int isempty (stack *s)
{
   if (s->top == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
   if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Checking for Stack Full

```
int isempty (stack *s)
{
   if (s->top == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
   if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Example: A Stack using an Array

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo stack;

main() {
  stack A, B;
  create(&A);
  create(&B);
  push(&A,10);
  push(&A,20);
  push(&A,30);
  push(&B,100);
  push(&B,5);

  printf ("%d %d", pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(&B))
    printf ("\n B is empty");
  return;
}
```

Example: A Stack using Linked List

```
#include <stdio.h>
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo stack;

main() {
  stack *A, *B;
  create(&A);
  create(&B);
  push(&A,10);
  push(&A,20);
  push(&A,30);
  push(&B,100);
  push(&B,5);

  printf ("%d %d", pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(B))
    printf ("\n B is empty");
  return;
}
```

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Validate XML
  - Program execution stack
  - Expression evaluation-infix to postfix, infix to prefix
  - Parsing
  - Simulation of Recursion
  - Function call
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Infix and Postfix Notations

- Infix: operators placed between operands

  A+B*C

- Postfix: operands appear before their operators

  ABC*+

- There are no precedence rules to learn in postfix notation, and parentheses are never needed

Infix to Postfix

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

A + B * C → (A + (B * C)) → (A + (B C *)) → A B C * +

A + B * C + D → ((A + (B * C)) + D) → ((A + (B C*))+ D) → ((A B C *+) + D) → A B C * + D +

Infix to postfix conversion

- Use a stack for processing operators (push and pop operations).
- Scan the sequence of operators and operands from left to right and perform one of the following:
  - output the operand,
  - push an operator of higher precedence,
  - pop an operator and output, till the stack top contains operator of a lower precedence and push the present operator.

The algorithm steps

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Infix to Postfix Conversion

Requires operator precedence information

Operands:

Add to postfix expression.

Close parenthesis:

pop stack symbols until an open parenthesis appears.

Operators:

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator.

End of input:

Pop all remaining stack symbols and add to the expression.

Infix to Postfix Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

Postfix notation is also called as Reverse Polish Notation (RPN)

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

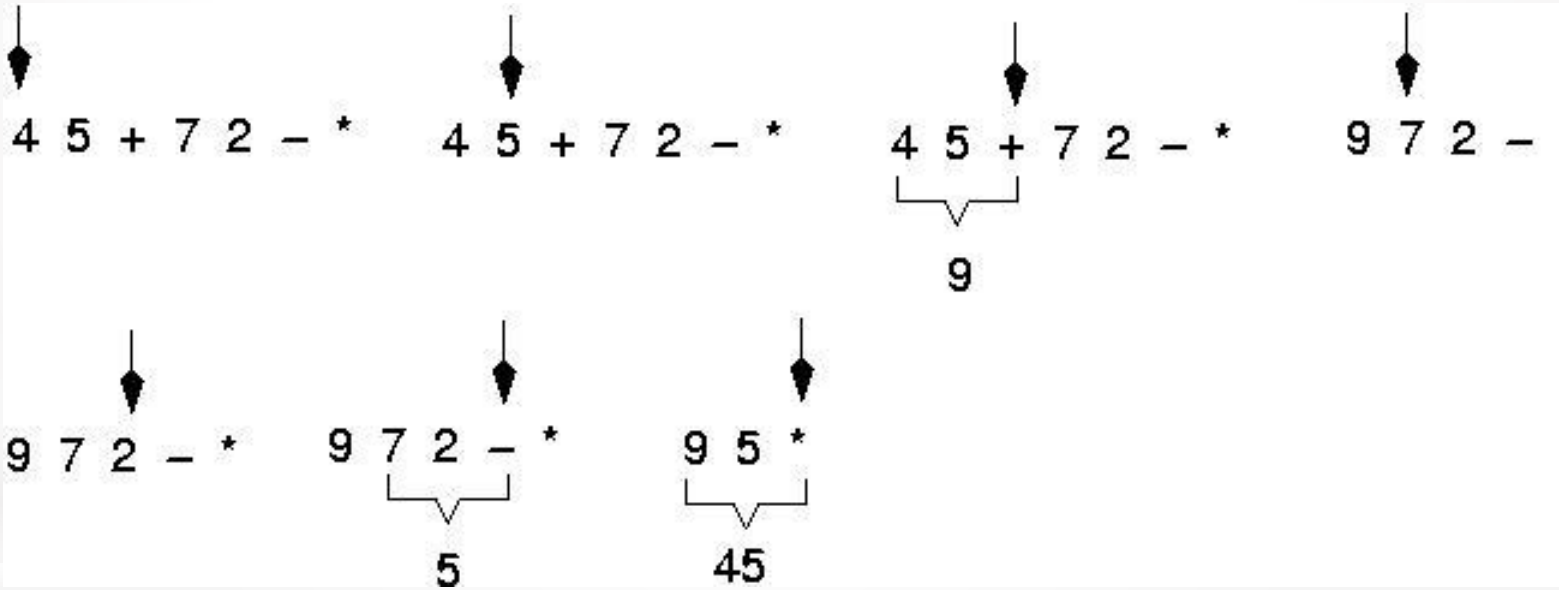
Example: Postfix Expressions

- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.

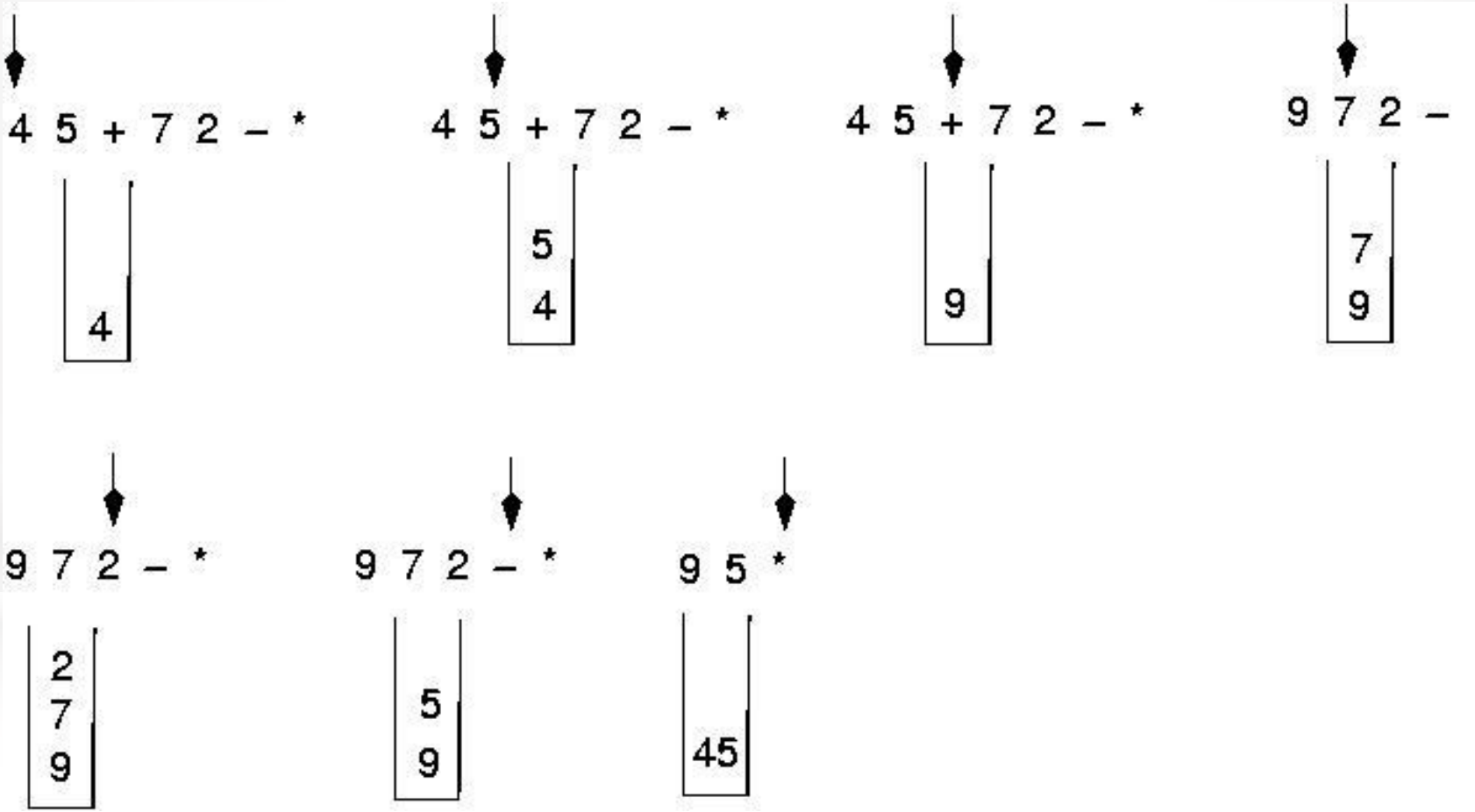
infix: 2+5 *postfix*: 2 5 +

- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never needed!!

Example: Postfix Expressions (Cont.)



Postfix expressions: Algorithm using stacks (cont.)



Postfix expressions: Algorithm using stacks (cont.)

```
WHILE more input items exist
        Get an item
        IF item is an operand
    stack.Push(item)
  ELSE
    stack.Pop(operand2)
    stack.Pop(operand1)
    Compute result
    stack.Push(result)
stack.Pop(result)
```

- Write the body for a function that replaces each copy of an item in a stack with another item. Use the following specification.

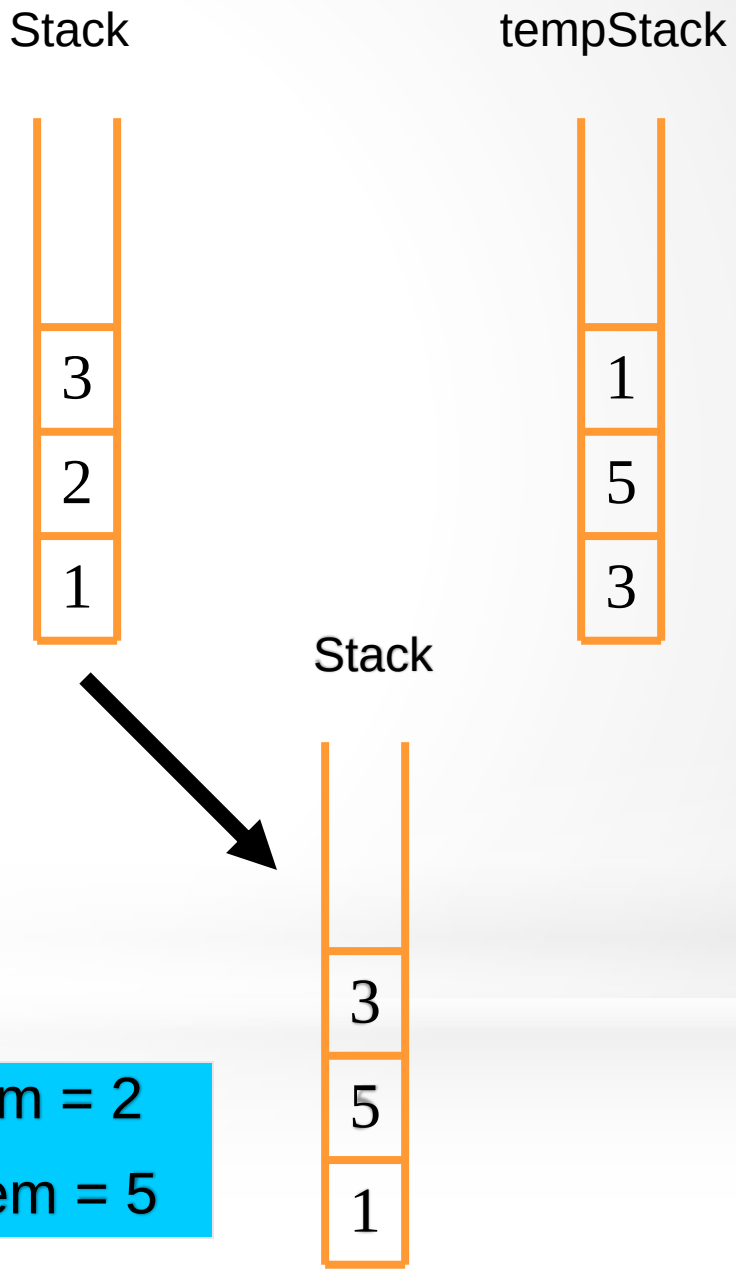
- **ReplaceItem(StackType& stack, ItemType oldItem, ItemType newItem)**

- ***Function***: Replaces all occurrences of oldItem with newItem.
- ***Precondition***: stack has been initialized.
- ***Postconditions***: Each occurrence of oldItem in stack has been replaced by newItem.

- (You may use any of the functions of the StackType).

```
{
  ItemType item;
  StackType tempStack;

  while (!Stack.IsEmpty())  {
          Stack.Pop(item);
          if (item==oldItem)
            tempStack.Push(newItem);
          else
            tempStack.Push(item);
  }
  while (!tempStack.IsEmpty()) {
          tempStack.Pop(item);
          Stack.Push(item);
  }
}
```



References

- BOOKS:
  - A.M. Tenenbaum, “Data Structures using C”, Prentice Hall
  - S. Lipschutz, “Theory and Problems of Data Structures”, McGraw Hill
- ONLINE RESOURCES
  - Microsoft MSDN library

ExTRA

ExTRA

- Longest Valid Parentheses Substring
- Last Updated : 16 Sep, 2024
- Given a string str consisting of opening and closing parenthesis ‘(‘ and ‘)‘. Find the length of the longest valid parenthesis substring.

- Examples:
- Input: (((()
- Output : 2
- Explanation : Longest Valid Parentheses Substring is ().

- Input:)()())
- Output : 4
- Explanation: Longest Valid Parentheses Substring is ()() .
- The idea is to use a stack-based approach to track the indices of unmatched parentheses. It determines the length of valid (well-formed) parentheses substrings by keeping track of the position of the last unmatched closing parenthesis or the starting position of a valid substring.

- Follow the steps below to solve the problem:

- For every opening parenthesis, we push its index onto the stack.
- For every closing parenthesis, we pop the stack.
- If the stack becomes empty after popping, it means we’ve encountered an unmatched closing parenthesis, so we push the current index to serve as a base for the next potential valid substring.
- If the stack is not empty, we calculate the length of the valid substring by subtracting the index at the top of the stack from the current index.
- A variable maxLength keeps track of the maximum length of valid parentheses encountered during the traversal.

Two Stacks in One Array

- Create a data structure twoStacks that represent two stacks. Implementation of two Stacks should use only one array, i.e., both stacks should use the same array for storing elements.

- Following functions must be supported by twoStacks.
  - push1(int x) –> pushes x to first stack
  - push2(int x) –> pushes x to second stack
  - pop1() –> pops an element from first stack and return the popped element
  - pop2() –> pops an element from second stack and return the popped element

- Implementation of twoStack should be space efficient.
- Implement two stacks in an array by Dividing the space into two halves:

- The idea to implement two stacks is to divide the array into two halves and assign two halves to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[(n/2) + 1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

- Follow the steps below to solve the problem:

- To implement push1():
- First, check whether the top1 is greater than 0
- If it is then add an element at the top1 index and decrement top1 by 1
- Else return Stack Overflow
- To implement push2():
- First, check whether top2 is less than n – 1
- If it is then add an element at the top2 index and increment the top2 by 1
- Else return Stack Overflow
- To implement pop1():
- First, check whether the top1 is less than or equal to n / 2
- If it is then increment the top1 by 1 and return that element.
- Else return Stack Underflow
- To implement pop2():
- First, check whether the top2 is greater than or equal to (n + 1) / 2
- If it is then decrement the top2 by 1 and return that element.
- Else return Stack Underflow

https://www.geeksforgeeks.org/implement-two-stacks-in-an-array/

ExTRA