Stacks

Week 7 Lecture 1

- In this session, you will learn to:
  - Identify the features of a stack
  - Implement stacks
  - Apply stacks to solve programming problems

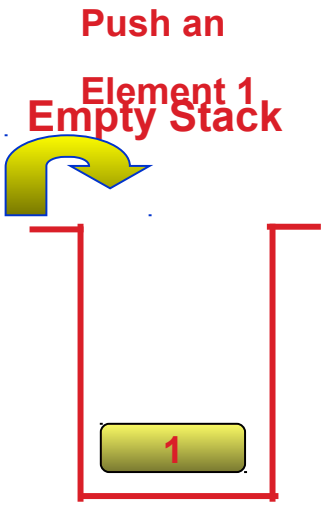
Defining a Stack

- What is a Stack ?

- A stack is a collection of data items that can be accessed at only one end, called top.
- Items can be inserted and deleted in a stack only at the top.
- The last item inserted in a stack is the first one to be deleted.
- Therefore, a stack is called a Last-In-First-Out (LIFO) data structure.

Identifying the Operations on Stacks

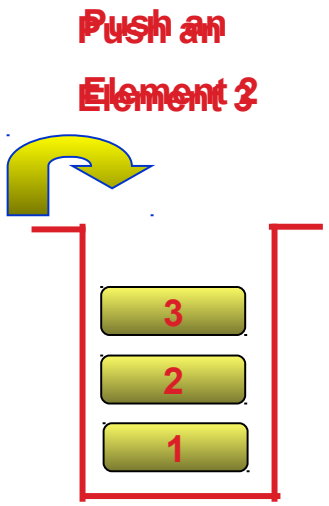
- There are two basic operations that are performed on stacks:
  - PUSH
  - POP



- **PUSH**: It is the process of inserting a new element on the top of a stack.

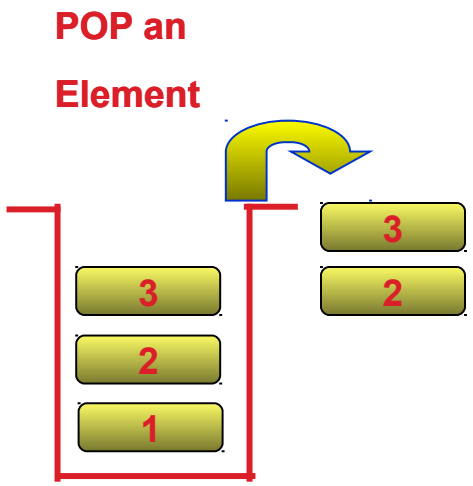
Identifying the Operations on Stacks (Contd.)

- **PUSH**: It is the process of inserting a new element on the top of a stack.



Identifying the Operations on Stacks (Contd.)

- **POP**: It is the process of deleting an element from the top of a stack.



Just a minute

- Elements in stacks are inserted and deleted on a _____ basis.

- Answer:
  - LIFO

Just a minute

- List down some real life examples that work on the LIFO principle.
- Answer:
  - **Pile of books**: Suppose a set of books are placed one over the other in a pile. When you remove books from the pile, the topmost book will be removed first. Similarly, when you have to add a book to the pile, the book will be placed at the top of the pile.
  - **Pile of plates**: The first plate begins the pile. The second plate is placed on the top of the first plate and the third plate is placed on the top of the second plate, and so on. In general, if you want to add a plate to the pile, you can keep it on the top of the pile. Similarly, if you want to remove a plate, you can remove the plate from the top of the pile.
  - **Bangles in a hand**: When a person wears bangles, the last bangle worn is the first one to be removed.

Implementing a Stack Using an Array

- IMPLEMENTATION OF STACK

stack can be implemented using

1)arrays (STATIC IMPLEMENTATION)

2)linked lists.(DYNAMIC IMPLEMENTATION)

- To implement a stack using an array:
  - Declare an array:

```
int Stack[5]; // Maximum size needs to be specified in
              // advance
```

  - Declare a variable, **top to hold** the index of the topmost element in the stacks:

```
int top;
```

  - Initially, when the stack is empty, set:

    top = –1

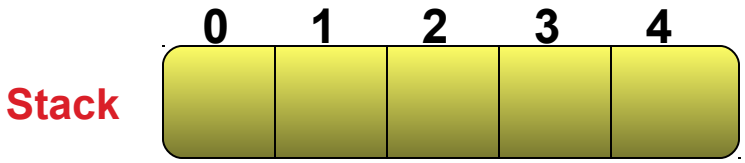
Implementing a Stack Using an Array (Contd.)

- Let us now write an algorithm for the PUSH operation.

Initially:

top = – 1

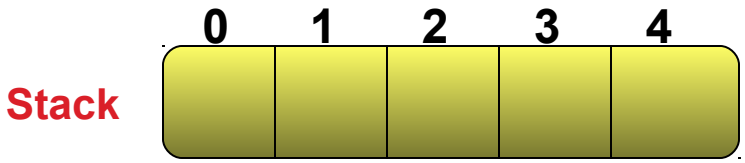
PUSH an element 3



Implementing a Stack Using an Array (Contd.)

top = – 1

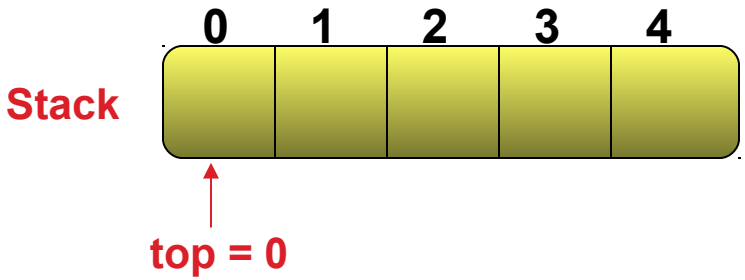
PUSH an element 3



Implementing a Stack Using an Array (Contd.)

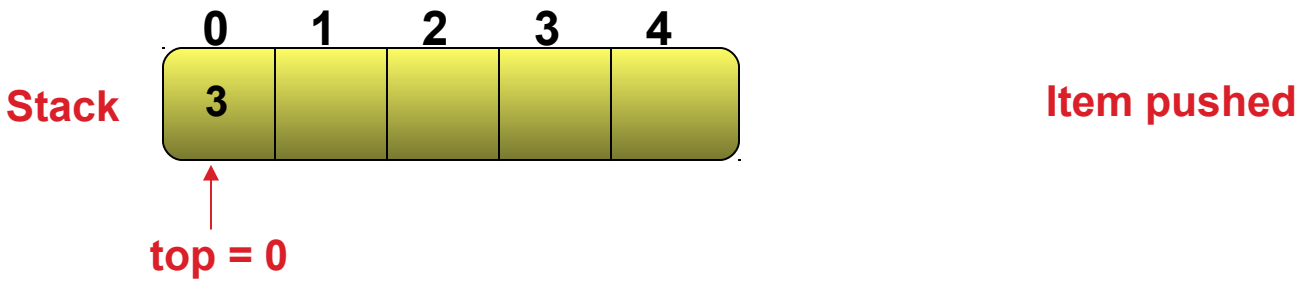
top = 0

PUSH an element 3



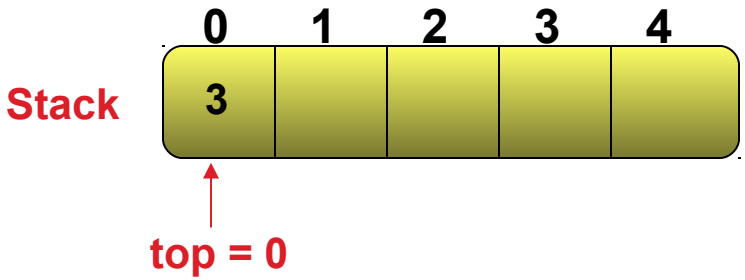
Implementing a Stack Using an Array (Contd.)

PUSH an element 3



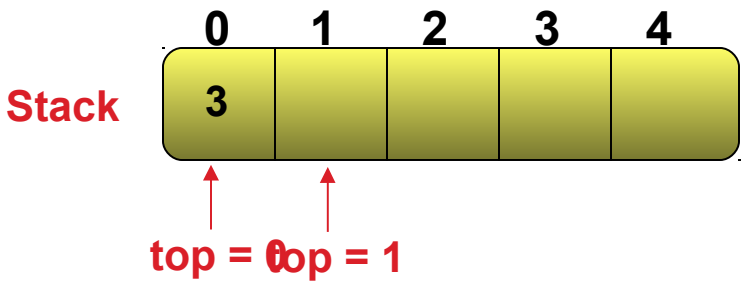
Implementing a Stack Using an Array (Contd.)

PUSH an element 8



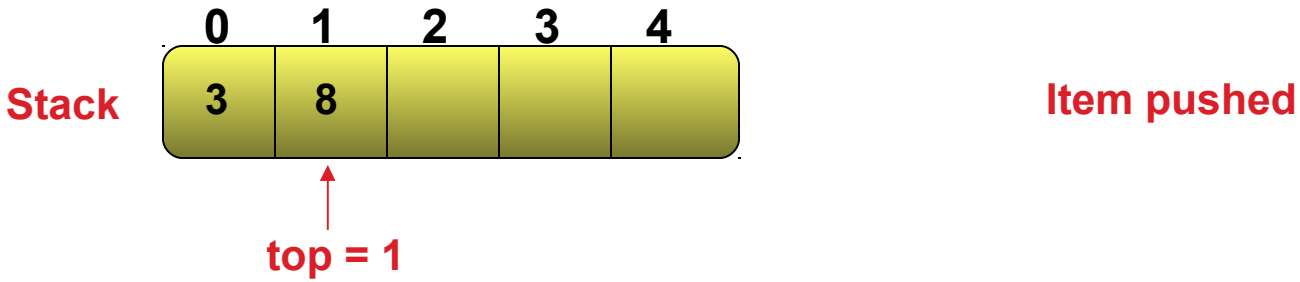
Implementing a Stack Using an Array (Contd.)

PUSH an element 8



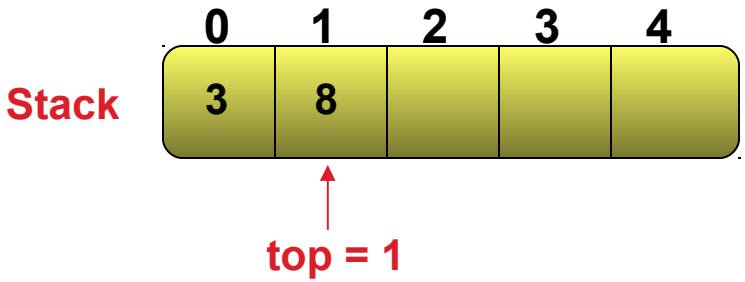
Implementing a Stack Using an Array (Contd.)

PUSH an element 8



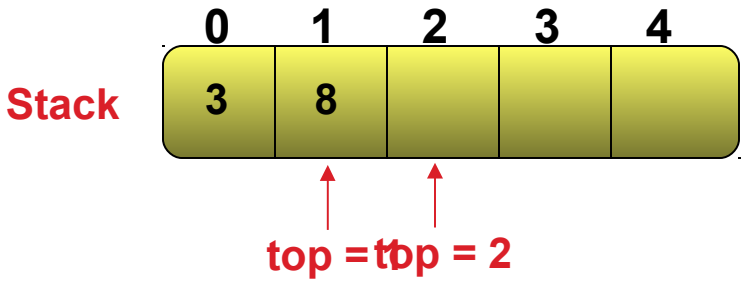
Implementing a Stack Using an Array (Contd.)

PUSH an element 5



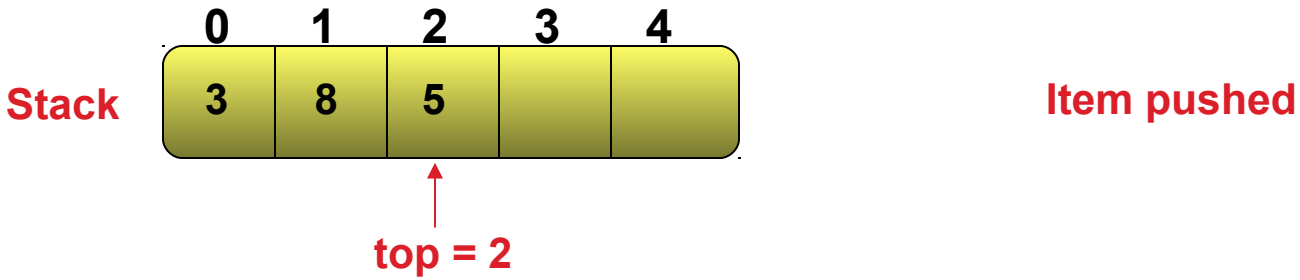
Implementing a Stack Using an Array (Contd.)

PUSH an element 5



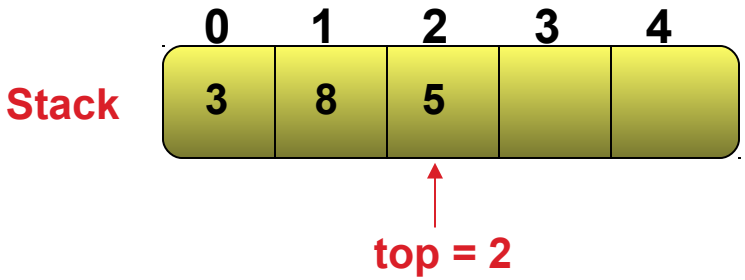
Implementing a Stack Using an Array (Contd.)

PUSH an element 5



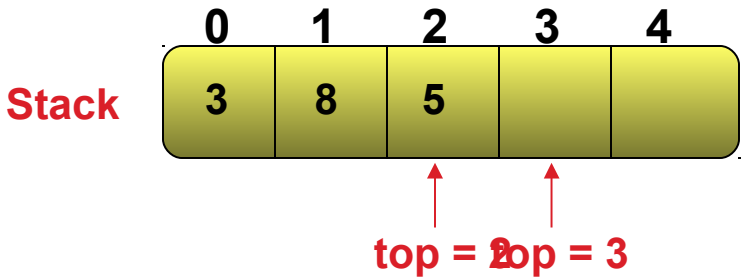
Implementing a Stack Using an Array (Contd.)

PUSH an element 1



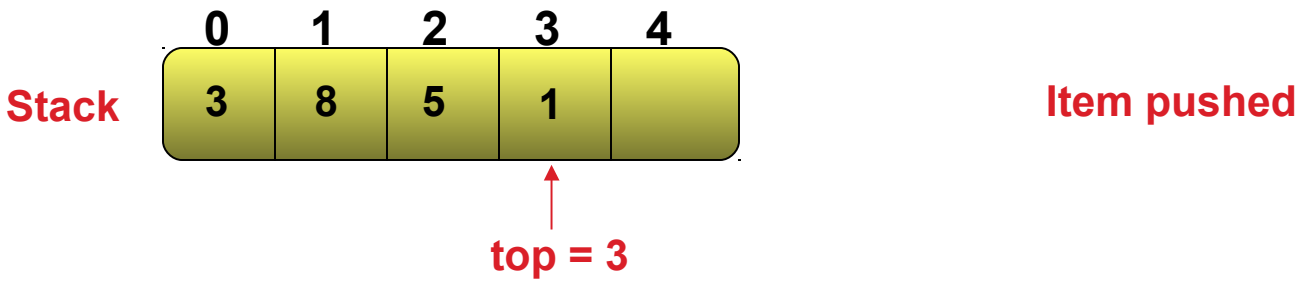
Implementing a Stack Using an Array (Contd.)

PUSH an element 1



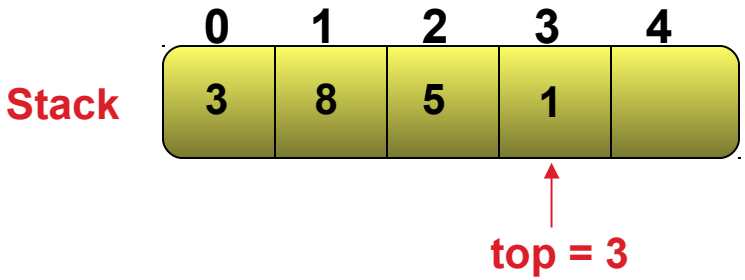
Implementing a Stack Using an Array (Contd.)

PUSH an element 1



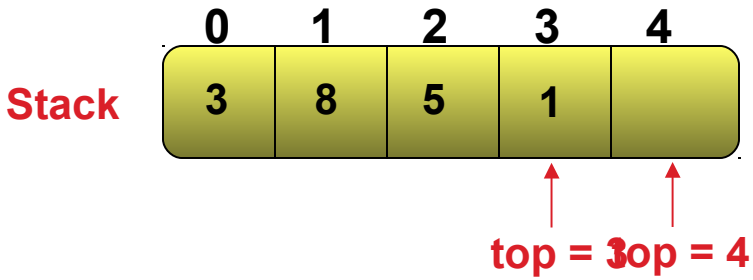
Implementing a Stack Using an Array (Contd.)

PUSH an element 9



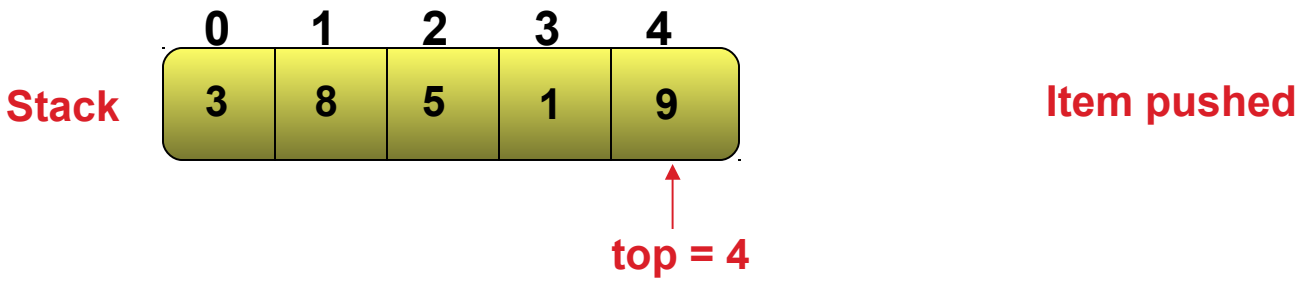
Implementing a Stack Using an Array (Contd.)

PUSH an element 9



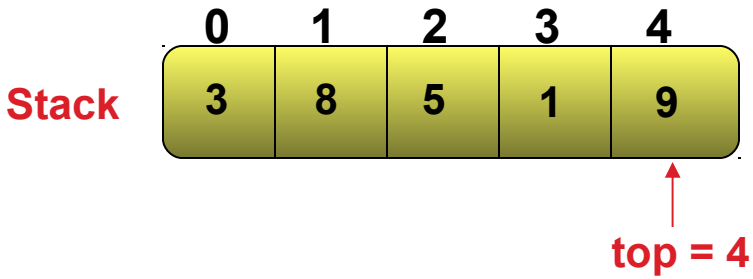
Implementing a Stack Using an Array (Contd.)

PUSH an element 9



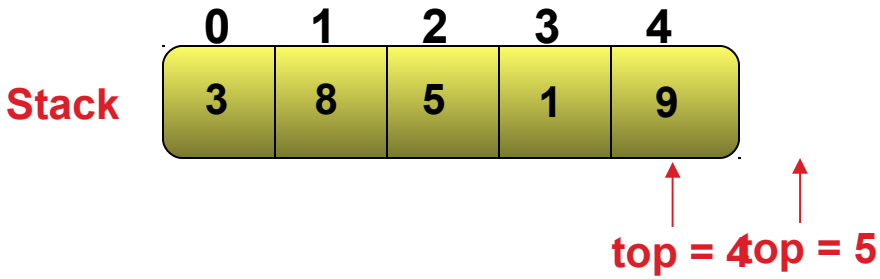
Implementing a Stack Using an Array (Contd.)

PUSH an element 2



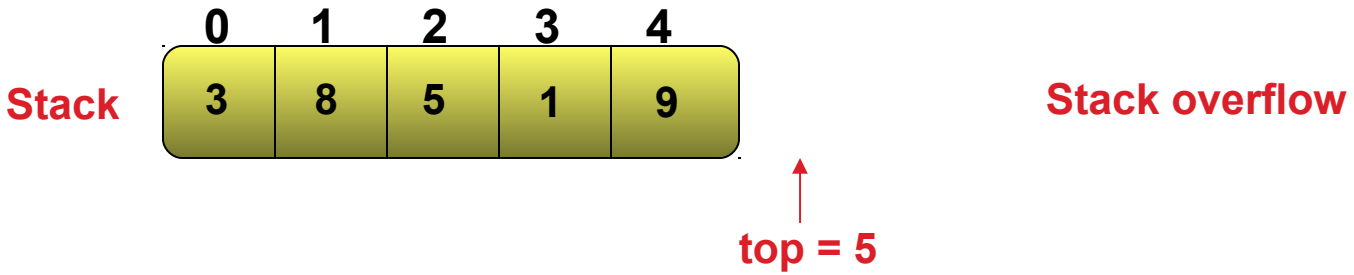
Implementing a Stack Using an Array (Contd.)

PUSH an element 2



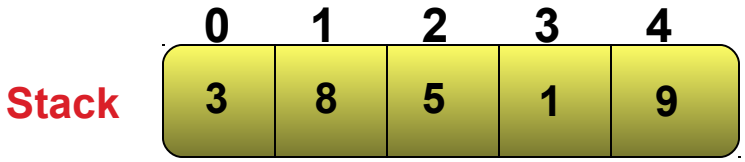
Implementing a Stack Using an Array (Contd.)

PUSH an element 2



Implementing a Stack Using an Array (Contd.)

- The stack has been implemented in an array of size 5.
- Therefore, you cannot store more than 5 elements in the stack.
- To avoid the stack overflow, you need to check for the stack full condition before pushing an element into the stack.



ALGORITHM TO PUSH AN ELEMENT INTO STACK

TOP=-1 TO REPRESENT STACK IS EMPTY

Algorithm PUSH(STACK[N],TOP,ITEM)

{

//STACK is an array of size N ,ITEM is element to be inserted, TOP represents position.

1. if (TOP == N-1)
    1.1 Print " Stack Overflow "
2. else
    2.1 TOP=TOP+1
    2.2 Enter ITEM
    2.3 STACK[TOP] = ITEM

}

- **Write an algorithm to implement the POP operation on a stack.**

```
Algorithm POP(STACK[N],TOP,ITEM)
   {
    1. if( TOP == – 1)
          1.1 Print “Stack Empty”
     2. else
          2.1  ITEM = STACK[TOP]
          2.2  TOP = TOP-1
    }
```

Just a minute

- In a stack, data can be stored and removed only from one end of the stack called the _____ of the stack.

- Answer:
  - top

Applications of Stacks

- Some of the applications of stacks are:
  - Implementing function calls
  - Maintaining the UNDO list for an application
  - Checking the nesting of parentheses in an expression
  - Evaluating expressions

Implementing Function Calls

Implementing function calls:

- Consider an example. There are three functions, F1, F2, and F3. Function F1 invokes F2 and function F2 invokes F3, as shown.

Implementing Function Calls (Contd.)

```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2(x);
1103         print(x);
     }
     void F2(int x)
     {
             x = x + 5;
1120         F3(x);
1121         print(x);
1122 }
     void F3(int x)
     {
             x = x * 2;
1140         print x;
1141 }
```

Assuming these instructions at the given locations in the memory.

Implementing Function Calls (Contd.)

```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2(x);
1103         print(x);
     }
     void F2(int x)
     {
             x = x + 5;
1120         F3(x);
1121         print(x);
1122 }
     void F3(int x)
     {
             x = x * 2;
1140         print x;
1141 }
```

The execution starts from function F1

Implementing Function Calls (Contd.)

```
                void F1()
                {
1100                    int x;
1101                    x = 5;
1102                    F2(x);
1103                    print(x);
                }
                void F2(int x)
                {
                        x = x + 5;
1120                    F3(x);
1121                    print(x);
1122            }
                void F3(int x)
                {
                        x = x * 2;
1140                    print x;
1141            }
```

Implementing Function Calls (Contd.)

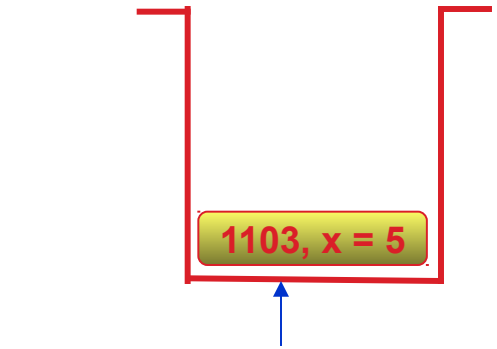
```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2(x);
1103         print(x);
     }
     void F2(int x)
     {
             x = x + 5;
1120         F3(x);
1121         print(x);
1122 }
     void F3(int x)
     {
             x = x * 2;
1140         print x;
1141 }
```

x = 5

Implementing Function Calls (Contd.)

```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2(x);
1103         print(x);
     }
     void F2(int x)
     {
             x = x + 5;
1120         F3(x);
1121         print(x);
1122 }
     void F3(int x)
     {
             x = x * 2;
1140         print x;
1141 }
```

x = 5



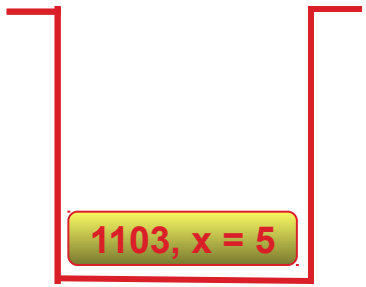
Address and the local variable of F1

Implementing Function Calls (Contd.)

```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2(x);
1103         print(x);
     }
     void F2(int x)
     {
             x = x + 5;
1120         F3(x);
1121         print(x);
1122 }
     void F3(int x)
     {
             x = x * 2;
1140         print x;
1141 }
```

x = 10

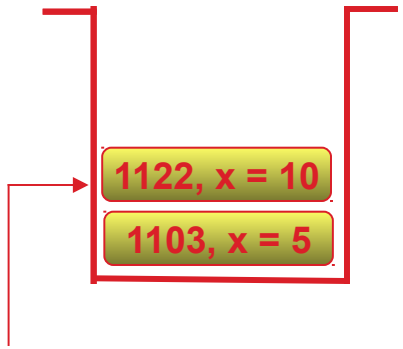
x = 5



Implementing Function Calls (Contd.)

```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2(x);
1103         print(x);
     }
     void F2(int x)
     {
             x = x + 5;
1120         F3(x);
1121         print(x);
1122 }
     void F3(int x)
     {
             x = x * 2;
1140         print x;
1141 }
```

x = 10



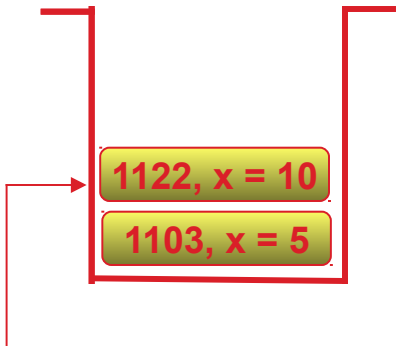
Address and the local variable of F2

Implementing Function Calls (Contd.)

```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2(x);
1103         print(x);
     }
     void F2(int x)
     {
             x = x + 5;
1120         F3(x);
1121         print(x);
1122 }
     void F3(int x)
     {
             x = x * 2;
1140         print x;
1141 }
```

x = 20

x = 10



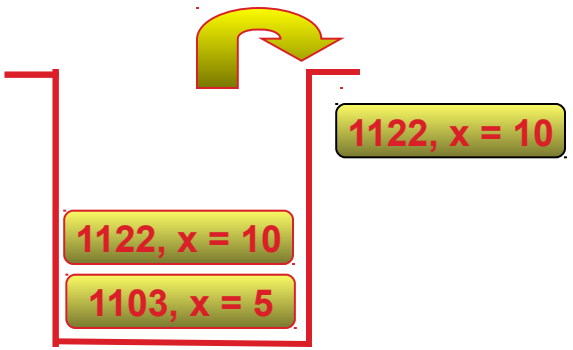
Address and the local variable of F2

Implementing Function Calls (Contd.)

```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2(x);
1103         print(x);
     }
     void F2(int x)
     {
             x = x + 5;
1120         F3(x);
1121         print(x);
1122 }
     void F3(int x)
     {
             x = x * 2;
1140         print x;
1141 }
```

x = 20

x = 10

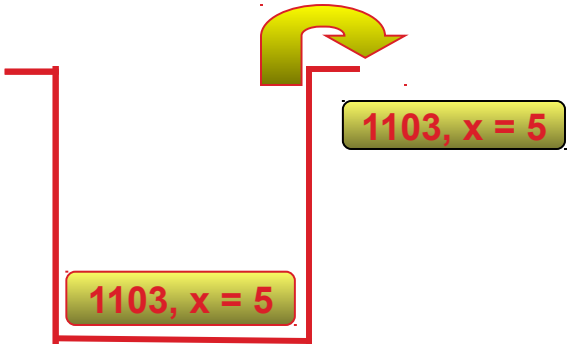


Implementing Function Calls (Contd.)

```
     void F1()
     {
1100          int x;
1101          x = 5;
1102          F2(x);
1103          print(x);
     }
     void F2(int x)
     {
              x = x + 5;
1120          F3(x);
1121          print(x);
1122  }
     void F3(int x)
     {
              x = x * 2;
1140          print x;
1141  }
```

x = 5

x = 10

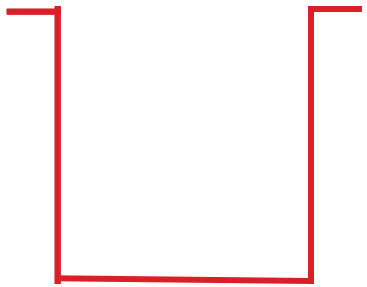


20 10

Implementing Function Calls (Contd.)

```
     void F1()
     {
1100         int x;
1101         x = 5;
1102         F2(x);
1103         print(x);
     }
     void F2(int x)
     {
             x = x + 5;
1120         F3(x);
1121         print(x);
1122 }
     void F3(int x)
     {
             x = x * 2;
1140         print x;
1141 }
```

x = 5



20 10 5

Maintaining the UNDO list for an application:

- Consider that you made some changes in a Word document. Now, you want to revert back those changes. You can revert those changes with the help of an UNDO feature.

- The UNDO feature reverts the changes in a LIFO manner. This means that the change that was made last is the first one to be reverted.

- You can implement the UNDO list by using a stack.

Checking the nesting of parentheses in an expression

You can do this by checking the following two conditions:

- The number of left parenthesis should be equal to the number of right parenthesis.
- Each right parenthesis is preceded by a matching left parenthesis.

Implementing Stacks

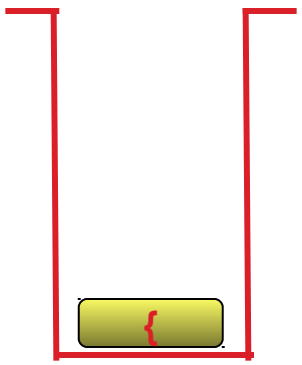
- You need to develop a method to check if the parentheses in an arithmetic expression are correctly nested.
- How will you solve this problem?
- You can solve this problem easily by using a stack.

Implementing Stacks (Contd.)

{(a + b) × (c + d)]}

- Consider an example.
- Suppose the expression is:

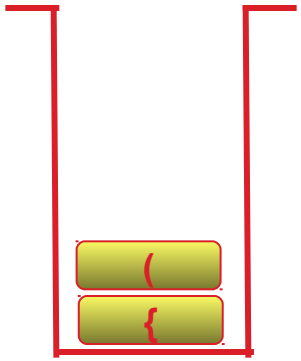
  {(a + b) × (c + d)]}
- Scan the expression from left to right.
- The first entry to be scanned is '{', which is a left parenthesis.
- Push it into the stack.



Implementing Stacks (Contd.)

{(a + b) × (c + d)]}

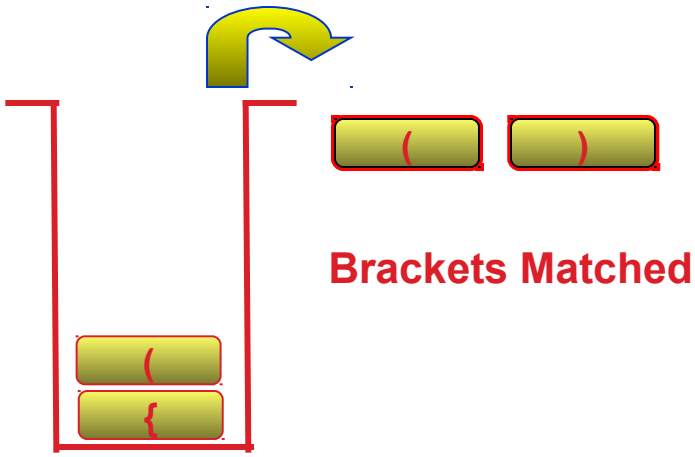
- The next entry to be scanned is '(', which is a left parenthesis.
- Push it into the stack.
- The next entry is 'a', which is an operand. Therefore, it is discarded.
- The next entry is '+', which is an operator. Therefore, it is discarded.
- The next entry is 'b', which is an operand. Therefore, it is discarded.



Implementing Stacks (Contd.)

- The next entry to be scanned is ‘)’, which is a right parenthesis
- POP the topmost entry from the stack.
- Match the two brackets.

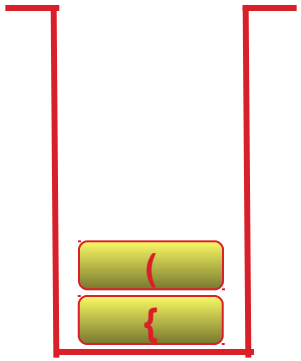
{(a + b) × (c + d)]}



Implementing Stacks (Contd.)

{(a + b) × (c + d)]}

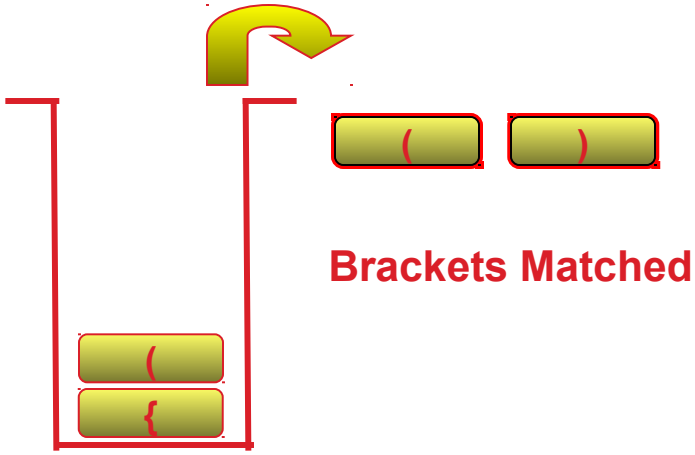
- The next entry to be scanned is '×', which is an operator. Therefore, it is discarded.
- The next entry to be scanned is '(', which is a left parenthesis
- Push it into the stack
- The next entry to be scanned is 'c', which is an operand. Therefore it is discarded
- The next entry to be scanned is '+', which is an operator. Therefore it is discarded
- The next entry to be scanned is 'd', which is an operand. Therefore it is discarded



Implementing Stacks (Contd.)

- The next entry to be scanned is ‘)’, which is a right parenthesis.
- POP the topmost element from the stack.
- Match the two brackets.

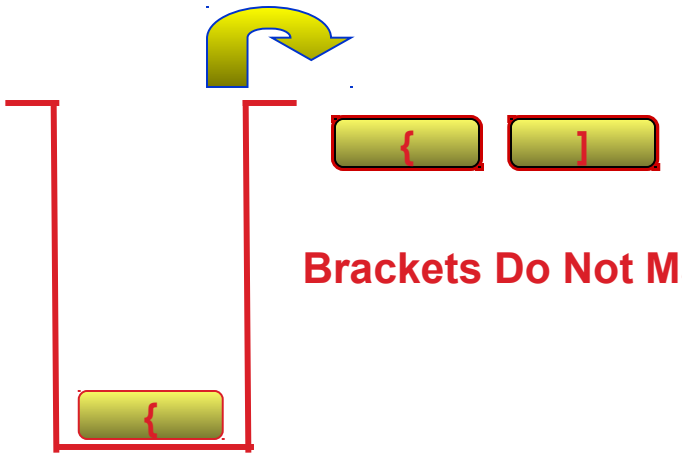
{(a + b) × (c + d)]}



Implementing Stacks (Contd.)

- The next entry to be scanned is ‘]’, which is a right parenthesis.
- POP the topmost element from the stack.
- Match the two brackets.

{(a + b) × (c + d)]}



Brackets Do Not Match

The Expression is INVALID

Algorithm for Balanced Symbol Checking

- Make an empty stack
- read symbols until end of file
  - if the symbol is an opening symbol push it onto the stack
  - if it is a closing symbol do the following
    - if the stack is empty report an error
    - otherwise pop the stack. If the symbol popped does not match the closing symbol report an error
- At the end of the file if the stack is not empty report an error

Evaluating an expression by using stacks:

- Stacks can be used to solve complex arithmetic expressions.
- The evaluation of an expression is done in two steps:
  - Conversion of the infix expression into a postfix expression.
  - Evaluation of the postfix expression.

In this session, you learned that:

- A stack is a collection of data items that can be accessed at only one end, called top. The last item inserted in a stack is the first one to be deleted.
- A stack is called a LIFO data structure.
- There are two operations that can be performed on stacks. They are:
  - **PUSH**
  - **POP**
- Stacks can be implemented by using both arrays and linked lists
- Stacks are used in many applications. Some of the application domains of stacks are as follows:
  - Implementing function calls
  - Maintaining the UNDO list for an application
  - Checking the nesting of parentheses in an expression
  - Evaluating expressions