UNIVERSITI TEKNOLOGI MALAYSIA

DSPD1733: DATA STRUCTURES AND ALGORITHMS

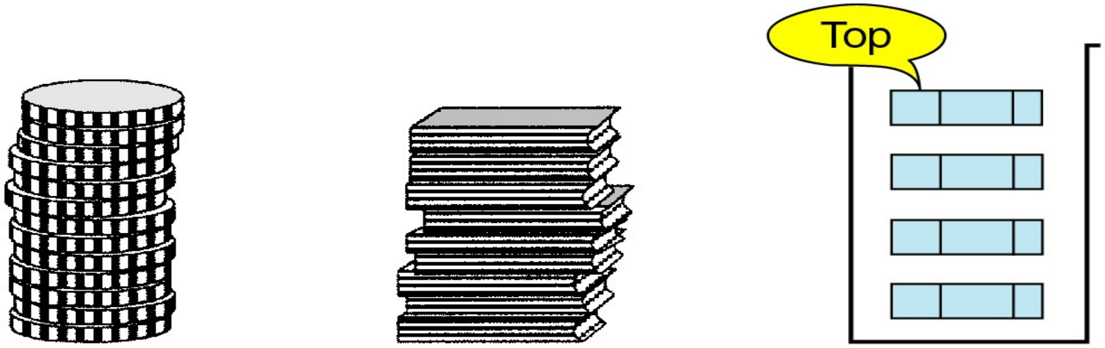
Pusat Pengajian Diploma, SPACE UTM Kuala Lumpur

CHAPTER 4:

STACK

- Stack Concepts
- Stack Operations: Push, Pop, Stack Top,
- Array Based Stacks and Linked List Stacks.
- Stack Applications: Reverse Data and Postponement,
- Infix To Postfix Transformation, Evaluating Postfix Expressions, Reverse Data.

Stack Concepts



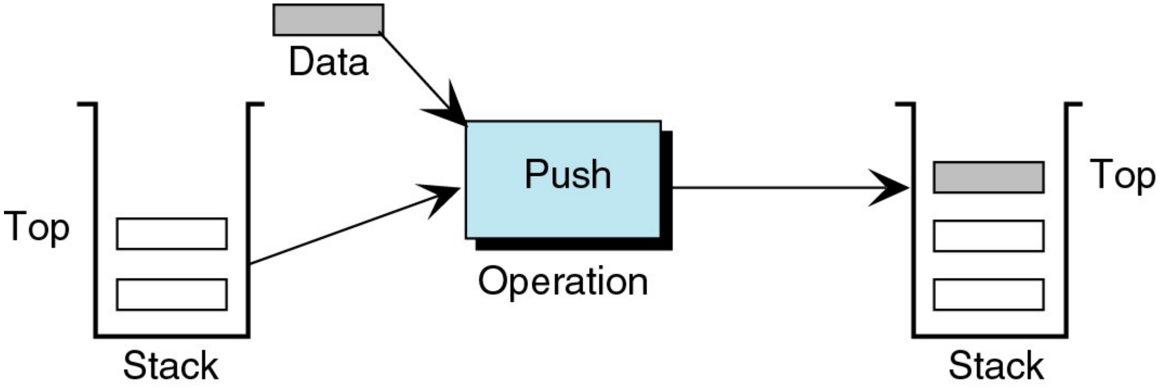
Stack of coins

Stack of books

Computer stack

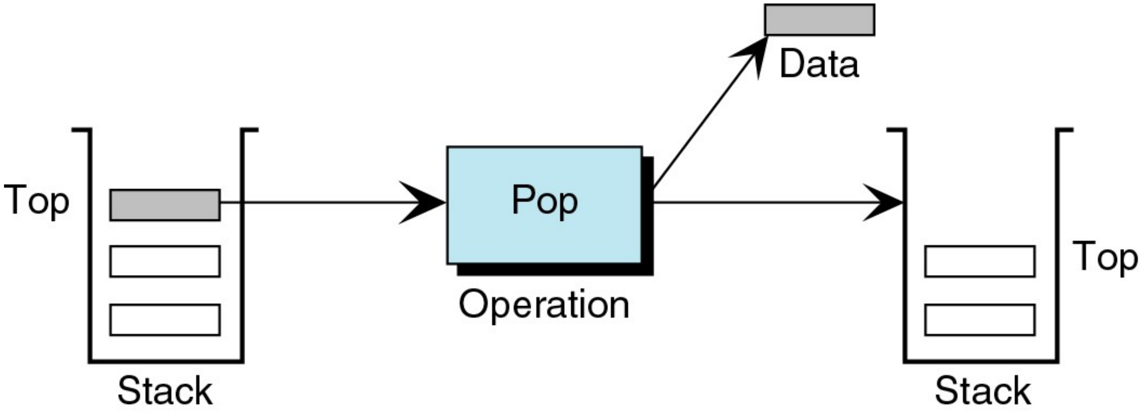
- Is a last in first out (LIFO) data structure
- All insertions and deletions are restricted to one end, called the top.
- Basic Stack Operation:
  - Push
  - Pop
  - Stack Top

Stack Operation : Push



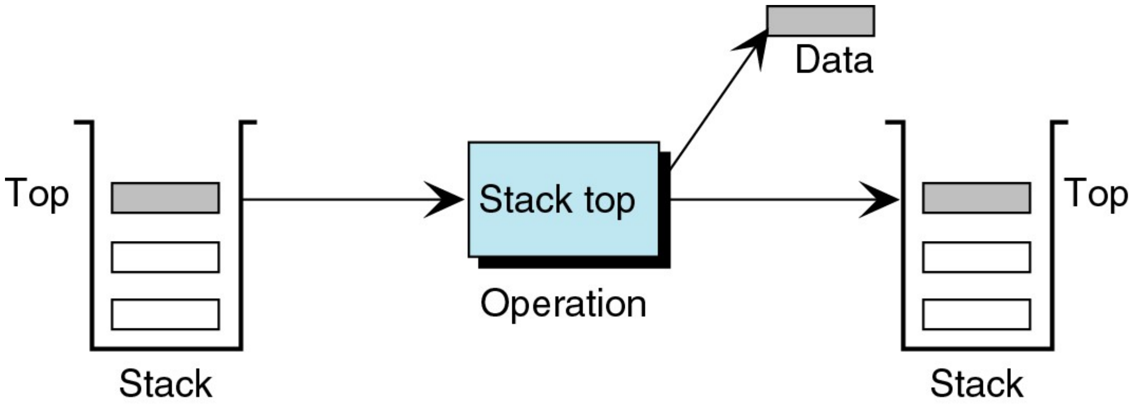
- Adds an item at the top of the stack.
- After push, the new item becomes the top.
- Must ensure that there is space for the new item.
- If there is not enough space, the stack is in an overflow state, the item cannot be added.

Stack Operation : Pop



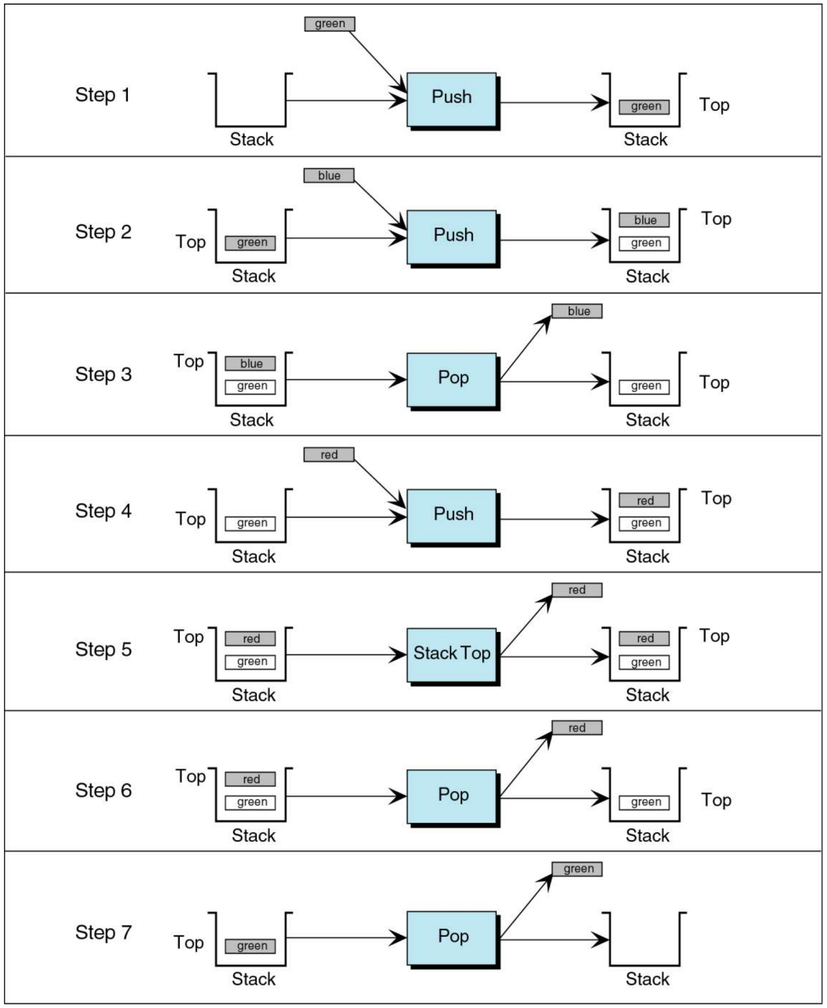
- Remove the item at the top of the stack and return it to the user.
- After removed the top item, the next older item in the stack becomes the top.
- When the last item in the stack is deleted, the stack must set to its empty state.
- If pop is called when the stack is empty, then the stack is in an underflow state.

Stack Operation : Stack Top

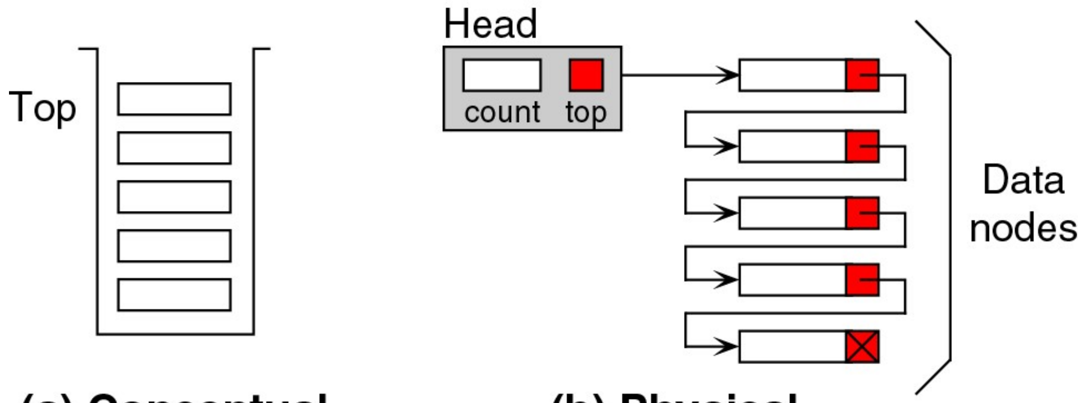


- Stack top copies the item at the top of the stack and return the top element to the user but does not delete it.
- Stack top can result in underflow if the stack is empty.

Stack Example



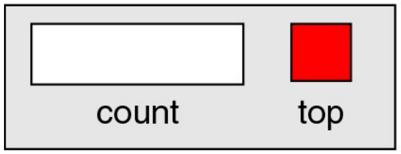
Stack Linked List Implementation



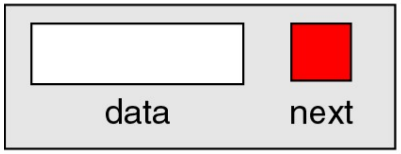
(a) Conceptual (b) Physical

- To implement the linked list stack, need two different structures:
    - Head – contain metadata and a pointer to the top of the stack.
    - Data node – contain data and a next pointer to the next node in the stack.

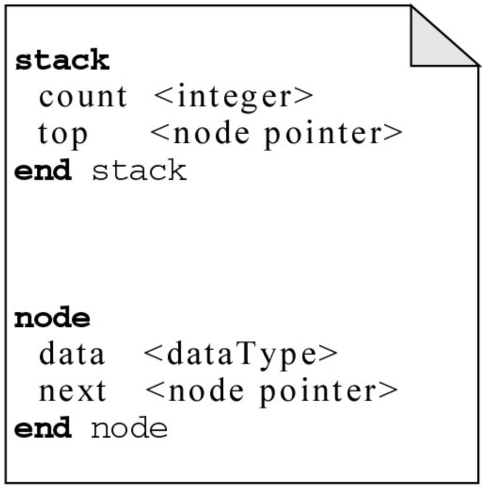
Stack head and Stack Data Node



Stack head structure



Stack node structure

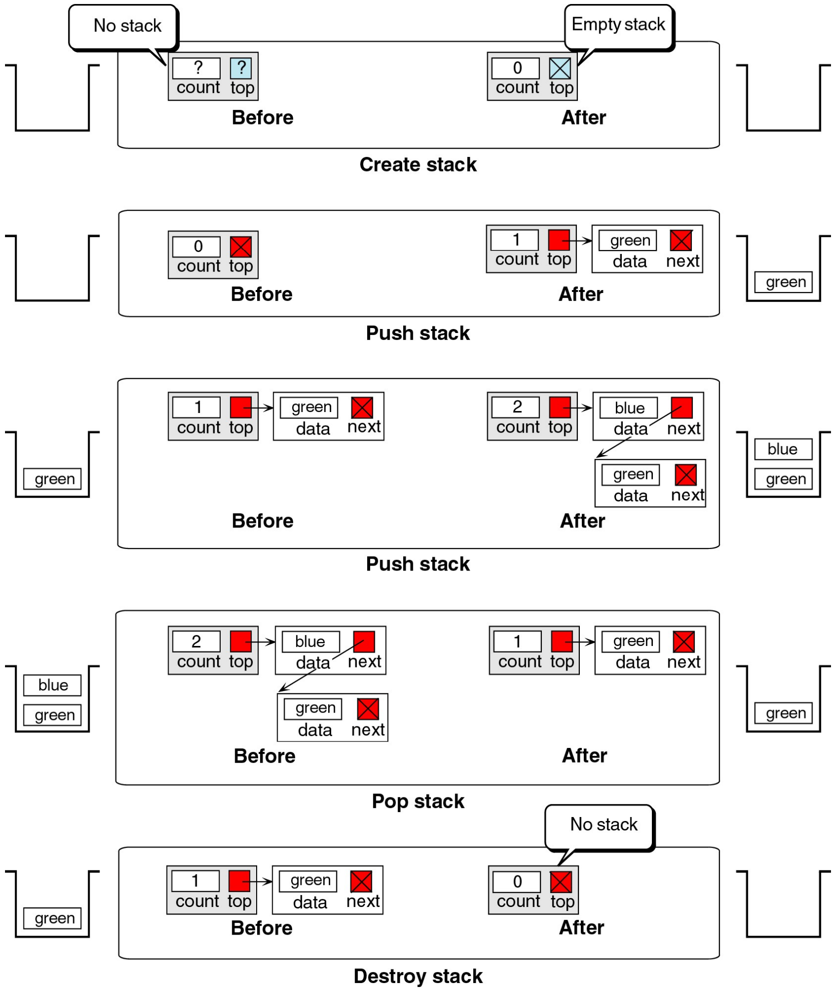


```
stack
  count  <integer>
  top    <node pointer>
end stack


node
  data   <dataType>
  next   <node pointer>
end node
```

- Head for a stack requires two attributes:
  - a top pointer
  - a count of the number of elements in the stack
- Stack data node looks like any linked list node.

Linked List Stack Operations



Create stack

Push stack

Push stack

Pop stack

Destroy stack

Stack Algorithms : Create Stack

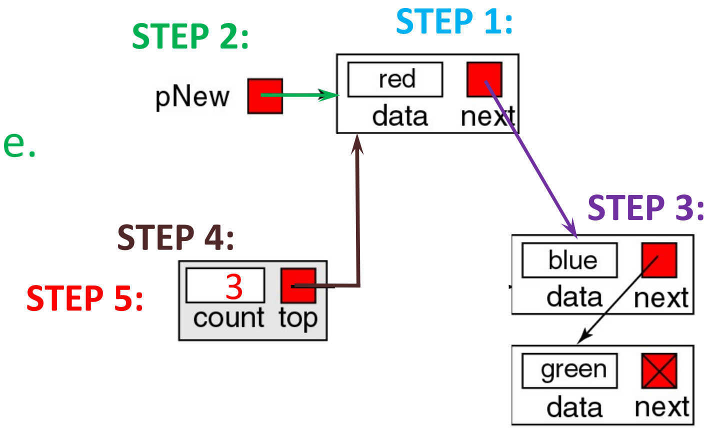
- Initialize the metadata for the stack structure.

```
Algorithm createStack (ref stack  <metadata>)
Initialize metadata for a stack.
   Pre          stack is a metadata structure passed by reference
   Post         metadata initialized

    stack.head = null
    stack.count = 0
    return
end createStack
```

Stack Algorithms : Push Stack

- Insert an element into the stack.
- Steps to push data into a stack:
  1. Find a memory for the node by allocate memory from dynamic memory.
  2. Assign the data to the stack node.
  3. Set the next pointer to point to the node currently indicated as the stack top.
  4. Update the stack top pointer.
  5. Add 1 to the stack count field.
- pNew pointer is used to indentified the data to be inserted into the stack.



Stack Algorithms : Push Stack (continue)

- To develop the insertion algorithms, need to analyze three different stack condition:
  1. Insertion into empty stack.
  2. Insertion into a stack with data.
  3. Insertion into a stack when available memory is exhausted.

Stack Algorithms : Push Stack (continue)

- When insert into a stack that contains a data
  - The new node’s next pointer is set to point to the node currently at the top.
  - The stack’s top pointer is set to point to the new node.
- When insert into a an empty.
  - The new node’s pointer is set to null.
  - The stack’s top pointer is set to point to the new node (since the stack’s pointer is null, it can be used to set the new node’s next pointer to null)
- Stack overflow : depends on the application.

Stack Algorithms: Push Stack

Algorithm pushStack(ref stack <metadata>,
val data <dataType>)

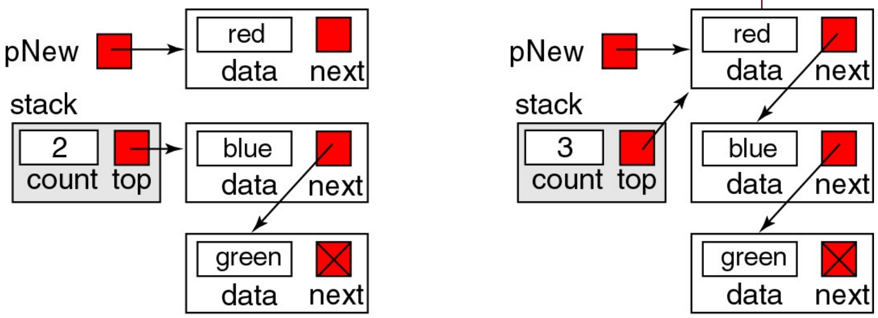
Insert (push) one item into the stack.

Pre stack is metadata structure to a valid stack
data contain data to be pushed into stack

Post data have been pushed in stack

Return true if successful, false if memory overflow

```
1   If (stack full)
    1   success = false
2   else
    1   allocate (newPtr)
    2   newPtr ->data = data
    3   newPtr ->next = stack.top
    4   stack.top = newPtr
    5   stack.count = stack.count + 1
    6   success = true
3   end if
4   return success
end pushStack
```



(a) Before **(b) After**

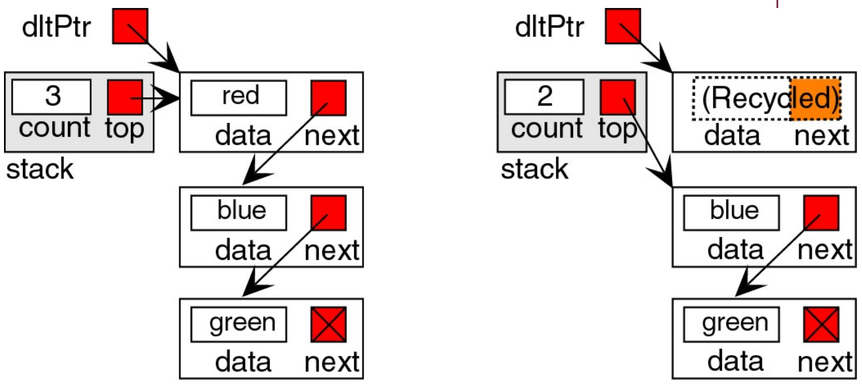
Stack Algorithms : Pop Stack

- Sends the data in the node at the top of the stack back to the calling algorithm.
- It then deletes and recycles the node (returns it to memory).
- Count is adjusted by subtracting 1.
- The algorithms returns to the caller.
- If the pop was successful, it returns true.
- If the stack is empty when pop is called, it returns false.

Stack Algorithms: Pop Stack

```
Algorithm   popStack( ref stack      <metadata>,
                      ref  dataOut   <dataType>)
This algorithm pops the item on the top of the stack and returns it to the user.
   Pre      stack is metadata structure to a valid stack
            dataOut is a reference variable to receive the data
   Post     data have been return to the calling algorithm
   Return   true if successful, false if underflow

1     If (stack empty)
         1     success = false
2     else
         1     dltPtr = stack.top
         2     dataOut = stack.top->data
         3     stack.top= stack.top->next
         4     stack.count = stack.count – 1
         5     recycle (dltPtr)
         6     success = true
3     end if
4     return success
end popStack
```



(a) Before

(b) After

Stack Algorithms : Stack Top

- Sends the data in the node at the top of the stack back to the calling module without deleting the top node.
- It allows the user to see what will be deleted when the stack is popped.
- If the stack top was successful, it returns true.
- If the stack is empty when stack top is called, it returns false.

Stack Algorithms: Stack Top

Algorithm stackTop(val stack <metadata>,
ref dataOut <dataType>)

This algorithm retrieves the data from the top of the stack without changing the stack.

Pre stack is metadata structure to a valid stack
dataOut is a reference variable to receive the data

Post data have been return to the calling algorithm

Return true if successful, false if underflow

```
1   if (stack empty)
    1   success = false
2   else
    1   dataOut = stack.top->data
    2   Success = true
3   end if
4   return success
end stackTop
```

Other Stack Operations

- Empty stack – determine whether the stack is empty.
- Full stack – determine whether there is not enough space or memory to allocate new data.
- Stack count – returns the number of elements currently in the stack.
- Destroy stack –delete all data in the stack

Stack Algorithms: Empty Stack

```
Algorithm   emptyStack ( val stack  <metadata> )
Determine if stack is empty and returns a Boolean.
   Pre      stack is metadata structure to a valid stack
   Post     return stack status
   Return     Boolean, true: stack empty, false: stack contain
  data


1    if (stack not empty)
     1    result = false
2    else
     1    result = true
3    end if
4    return result
end emptyStack
```

Stack Algorithms: Full Stack

```
Algorithm   fullStack ( val stack   <metadata> )
Determine if stack is full and returns a Boolean.
   Pre      stack is metadata structure to a valid stack
   Post     return stack status
   Return   Boolean, true: stack full, false: memory available

1    if (memory available)
     1    result = false
2    else
     1    result = true
3    end if
4    return result
end fullStack
```

Stack Algorithms: Stack Count

```
Algorithm   stackCount ( val stack   <metadata> )
Returns the number of elements currently in stack.
    Pre      stack is metadata structure to a valid stack
    Post     return stack count
    Return   integer count of number of elements in stack

    return stack.count

end stackCount
```

Stack Algorithms: Destroy Stack

```
Algorithm   destroyStack ( ref stack   <metadata> )
This algorithm releases all nodes back to the dynamic
memory.
    Pre      stack is metadata structure to a valid stack
    Post     stack empty and all nodes recycled

1   Loop while (stack.top not null)
      1   temp = stack.top
      2   Stack.top = stack.top->next
      3   Recycle (temp)
2   end loop
3   stack.count = 0
4   return
end destroyStack
```

Stack Application: Reversing Data

- Reversing data requires that a given set of data be reordered so that the first and the last elements are exchanged, with all the positions between the first and last being relatively exchanged also.
- Example: 1, 2, 3, 4 becomes 4, 3, 2, 1

Reverse a List Algorithm

Algorithm reverseNumber

This program reverse a list of integers read from the keyboard by pushing them into a stack and retrieving them one by one.

```
1     createStack (stack)
2     prompt (Enter a number)
3     Read (number)
      Fill stack
4     loop (not end of data AND not fullStack(stack))
   1     pushStack (stack, number)
   2     prompt (Enter next number: <EOF> to stop)
   3     read (number)
5     end loop
      Now print numbers in reverse
6     loop (not emptyStack(stack))
   1     popStack (stack, dataOut)
   2     print (dataOut)
7     end loop
end reverseNumber
```

Stack Application: Postponement application

- Application that requires the use of data to be postponed for a while
- Two stack postponement application
  - Infix to postfix transformation
  - Postfix expression evaluation

Infix to Postfix Transformation

- An arithmetic expression can be represented in three different formats:
  - Infix : **a + b** (the operator comes between the operands)
  - Postfix : **a b +** (the operator comes after the operands)
  - Prefix : **+ a b** (the operator comes before the operands)
- In Infix notation, need to use parentheses to control evaluation of the operators.
- In postfix and prefix notations, no need parentheses, each provides only one evaluation rule.

Infix to Postfix: Manual Transformation

- Fully parenthesize the expression using any explicit parentheses and the arithmetic precedence – multiply and divide before add and subtract.
- Change all infix notation in each parenthesis to postfix notation, starting from the innermost expression. Conversion to postfix notation is done by moving the operator to the location of the expression's closing parenthesis.
- Remove all parentheses.
- Example:

A+B*C

Step 1 result in **(A+(B*C))**

Step 2 moves the multiply operator after C

(A+(BC*))

And then moves the addition operator to between the last two closing parentheses. This change is made because the closing parenthesis for the plus is the last parentheses.

(A (BC*) +)

Finally, **step 3** removes the parentheses.

A BC* +

Transformation Infix to Postfix using Stack Algorithm

- As in infix expressions use a precedence rule to determine how to group the operands and operators in an expression.
- Rules to follow when using stack:
  - Need to push an operator into the stack.
  - If its priority is higher than the operator at the top of the stack, keep push it into the stack.
  - If the operator at the top of the stack has a higher priority that the current operator, it is popped and placed in the output expression.

Priority 2: * /
Priority 1: + -
Priority 0: (

	Infix	Stack	Postfix
(a)	A+B*C–D/ E		
(b)	+B*C–D/ E		A
(c)	B*C–D/ E	+	A
(d)	*C–D/ E	+	AB
(e)	C–D/ E	* +	AB
(f)	–D/ E	* +	ABC
(g)	D/ E	–	ABC*+
(h)	/ E	–	ABC*+D
(i)	E	/ –	ABC*+D
(j)		/ –	ABC*+DE
(k)			ABC*+DE/–

Convert Infix to Postfix Algorithm

```
Algorithm   inToPostFix (val formula <string>)
Convert infix formula to postfix.
Pre    formula is infix notation that has been
       edited to ensure that there are no
       syntactical errors
Post   postfix formula has been formatted as
       a string
Return   postfix formula
1   createStack (stack)
2   set postfix to null string
3   looper = 0
4   loop (looper < sizeof formula)
    1   token = formula [looper]
    2   if (token is open parenthesis)
        1   pushStack (stack, token)
    3   elseif (token is close parenthesis)
        1   popStack (stack, token)
        2   loop (token not open parenthesis)
            1   concatenate token to postFix
            2   popStack(stack, token)
        3   end loop
    4   elseif  (token is operator)
     Test priority of token to token at top of stack
        1   stackTop (stack, topToken)
        2   loop (not emptyStack (stack) AND
            priority(token) <= priority(topToken))
            1   popStack (stack, tokenOut)
            2   concatenate tokenOut to postFix
            3   stackTop (stack, topToken)
        3   end loop
        4   pushStack (stack, token)
    5   else
    Character is operand
        1   concatenate token to postFix
    6   end if
    7   looper = looper +1
5   end loop
    Input formula empty. Pop stack to postFix
6   loop (not emptyStack(stack))
    1   popStack (stack, token)
    2   concatenate token to postFix
7   end loop
8   return postFix
end inToPostFix
```

Convert Infix to Postfix: Example

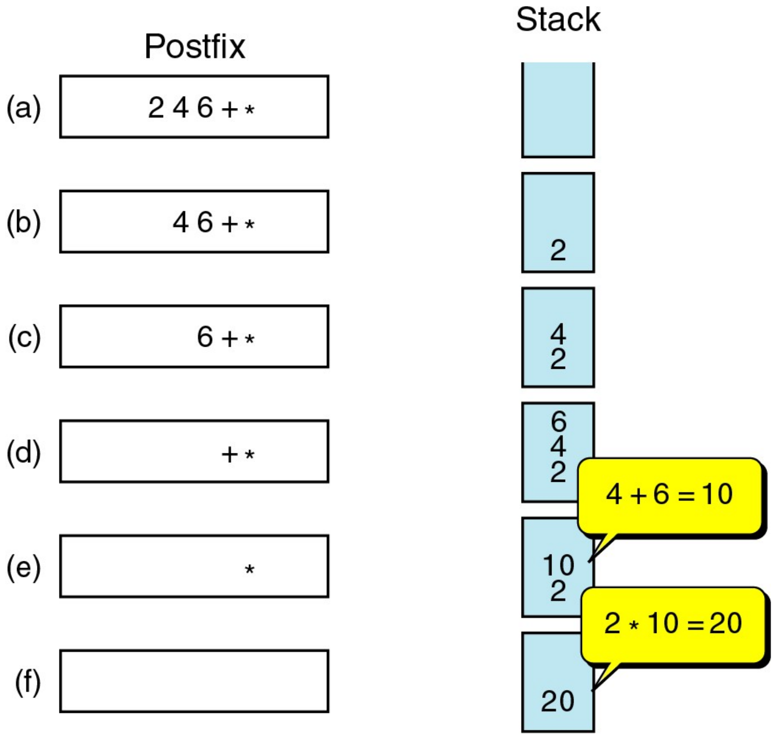
INPUT BUFFER	OPERATOR STACK	OUTPUT STRING
A * B – (C + D) + E	Empty	Empty
* B – (C + D) + E	Empty	A
B – (C + D) + E	*	A
– (C + D) + E	*	A B
– (C + D) + E	Empty	A B *
(C + D) + E	-	A B *
C + D) + E	- (	A B *
+ D) + E	- (	A B * C
D) + E	- (+	A B * C
) + E	- (+	A B * C D
+ E	-	A B * C D +
+ E	Empty	A B * C D + -
E	+	A B * C D + -
	+	A B * C D + - E
	Empty	A B * C D + - E +

Postfix expression evaluation

- Using stack postponement to evaluate the postfix expressions.
- Example: Given postfix expression below,

  A B C + *

  - Assume that A is 2, B is 4 and C is 6
  - What is the expression value?
- Solving by
  1. postpone the use of operands by push into the stack
  2. when an operator is found, pop the two operands at the top of the stack and perform the operation
  3. Then push the value back into the stack to be used later



Evaluation of PostFix Expressions Algorithm

```
Algorithm   postFixEvaluate (val  expr <string>)
This algorithm evaluates a postfix expression and returns its value.
Pre     a valid expression
Post    postfix value computed
Return    value of expression
1   exprSize = length of string
2   createStack (stack)
3   index = 0
4   loop (index  < exprSize)
    1    if  (expr [index] is operand)
         1    pushStack (stack, expr [index] )
    2    else
         1    popStack (stack, operand2)
         2    popStack (stack, operand1)
         3    operator = expr [index]
         4    value = calculate (operand1, operator,operand2)
         5    pushStack (stack, value)
    3    end if
    4    index = index + 1
5   end loop
6    popStack (stack, result)
7     return result
end postFixEvaluate
```

Advantages of a Stack

Simple to Implement:

Stacks are straightforward to implement using arrays or linked lists.

Efficient Data Access:

Push, pop, and peek operations are executed in constant time, i.e., **O(1)**.

Memory Management:

Stacks are widely used in memory allocation for managing the call stack, ensuring efficient execution of recursive functions and nested calls.

Helps in Reversing Data:

Stacks can easily reverse data sequences by pushing items into the stack and popping them out.

Supports Recursive Programming:

The call stack automatically manages function calls and return addresses.

Versatile Applications:

Stacks are essential in parsing expressions, syntax checking, balancing parentheses, and implementing algorithms like Depth-First Search (DFS).

Data Safety:

With controlled access (only through the top), stacks help maintain the integrity of data.

Real-World Examples of Stack Usage

Web Browsers:

The "Back" and "Forward" navigation in a browser uses stacks to maintain visited pages.

Text Editors:

Undo/Redo functionality is implemented using stacks.

Recursive Algorithms:

DFS in graph traversal and solving puzzles like Sudoku or the Tower of Hanoi.

Programming Languages:

The call stack in programming languages stores function calls, parameters, and local variables.

CONCLUSION

The **purpose** of a stack in data structure is to provide a simple and efficient mechanism for storing and managing data that adheres to the **Last In, First Out (LIFO)** principle.

It is designed to handle scenarios where the most recently added item needs to be accessed or removed first

Stacks are foundational to many algorithms and applications in computer science.

LABSKILL 2: Stack (5M) - individually

Write a program for reusable code that can storing and manage a collection of data.

Problem Statement:

Design a data structure that can be reuse to create a stack for storing a collection value type integer. Create 2 program using the same header file. First program is to convert decimal value to binary using stack. Second program is to reverse arrangement order a list of numbers using stack.

Output for first program:

```
Enter decimal value: 19

Binary value: 1 0 0 1 1
```

Output for second program:

```
Enter numbers: 63 75 78 89 92

Reverse list: 92 89 78 75 63
```

Hints for Data Structure Design:

1.Header Files:

- Define a DATA type that implement like int data type. Use typedef and define it in the header file (DATA.h)
- Define Node structure to be use with stack structure. Use struct and define it in the header file (NODE.h)
- Create a Stack class that have suitable attributes and operations to implement Stack objects which need to be define it in the header file (STACK.h)

2.Program File:

- Create main that can produce program such as the output shown.
- Must make use all the structure define in the header files.

www.utm.my

Quiz 1

```
pushStack  (s1, 3);
pushStack  (s1, 5);
pushStack  (s1, 7);
pushStack  (s1, 9);
pushStack  (s1, 11);
pushStack  (s1, 13);
While (!emptyStack (s1)) {
          popStack   (s1, x);
          pushStack  (s2, x);
}  //end while
```

Imagine there are two empty stacks of integers, s1 and s2.

Draw a picture of each stack after the above operations.

Quiz 2

- Using manual transformation, write the following infix expressions in their postfix and prefix forms.

  i. D – B + C

  ii. (A - 2 * (B + C) – D * E) * F

Quiz 3

- Change the following infix expressions to postfix expression using the algorithmic method (a stack).
  - i. A * B + C * D
  - ii. (A – 2) * (B + C) – (D * E) * F

Quiz 4

- If the value of A is 2, B is 3, C is 4 and D is 5. Determine the value of the following postfix expression.
  1. A B * C – D +
  2. A B C + * D –