Stack

Stack – Basics

❑ A stack is a linear data structure that follows the **Last In First Out (LIFO)** principle

- the most recently added element is the first one to be removed (observed).
- It is an Abstract Data Type (ADT) with two main operations:

  push: which adds an element to the top of the stack

  pop: which removes the element from the top of the stack and returns it

- A stack can be implemented using an **array** or a **linked list**, and is often used in computer science to store temporary data, such as **function calls**, history or undo/redo operations.

What is an Abstract Data Type (ADT)?

- An abstract data type is a data structure that provides a set of well-defined operations and defines a behaviour for the data it holds, without specifying the implementation details.

- the stack ADT that is discussed in previous slide defines what a stack is and what operations can be performed on a stack (push and pop), and what the behaviour of these operations should be (adding an element to the top, removing an element from the top)…
  - but it does not specify how the stack should be implemented (e.g., using an array, linked list, etc.).
  - The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.

Stack – Applications

- Implementing function calls (i.e., as part of the function call stack)
- Evaluating expressions, such as in converting from infix to postfix
- Balancing symbols, such as in syntax validation of HTML, XML, or programming languages
- Finding path in a maze
- Solving tower of hanoi puzzle.

Slack – Operations

In addition to push and pop, there are several other operations that can be performed on a stack:

- **Peek:** This operation returns the value of the top element of the stack without removing it.
- **Empty** : This operation checks if the stack is empty, and returns a Boolean value indicating whether the stack is empty or not.
- **Full** : This operation checks if the stack is full, and returns a Boolean value indicating whether the stack is full or not.
- **Size**: This operation returns the number of elements currently in the stack.
- **Clear**: This operation removes all elements from the stack, effectively making it empty.
- **Search**: This operation searches for a specific element in the stack and returns its position relative to the top of the stack.
- **Display**: This operation prints the elements of the stack in their current order.

Stack using Array vs Linked List

```python
class Stack:
    def __init__(self, size):
        self.stack = [None] * size
        self.top = -1
        self.size = size

    def push(self, item):
        if self.top < self.size - 1:
            self.top += 1
            self.stack[self.top] = item
        else:
            raise Exception("Stack overflow")

    def pop(self):
        if self.top >= 0:
            item = self.stack[self.top]
            self.top -= 1
            return item
        else:
            raise Exception("Stack underflow")
```

```python
stack = Stack(10)
```

```python
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
        self.top = None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node

    def pop(self):
        if self.is_empty():
            print("Stack underflow")
            return
        item = self.top.data
        self.top = self.top.next
        return item
```

```python
stack = Stack()
```

Stack in Array vs Linked List

```
def peek(self):
    if self.top >= 0:
        return self.stack[self.top]
    else:
        return None

def is_empty(self):
    if self.top == -1:
        return True
    else:
        return False

def size(self):
    return self.top + 1

def clear(self):
    self.top = -1

def search(self, item):
    for i in range(self.top, -1, -1):
        if self.stack[i] == item:
            return self.top - i
    return -1

def display(self):
    for i in range(self.top, -1, -1):
        print(self.stack[i], end=' ')
    print()
```

```
def is_empty(self):
    if self.top is None:
        return True
    else:
        return False

def peek(self):
    if self.is_empty():
        print("Stack is empty")
        return
    return self.top.data

def display(self):
    if self.is_empty():
        print("Stack is empty")
        return
    temp = self.top
    while temp:
        print(temp.data, end=" ")
        temp = temp.next
    print()
```

Balancing Parentheses Using Stack

- The idea is to use a **stack** to **keep track of the parentheses** as they are encountered in an expression.
- If a left parenthesis is encountered, it is pushed onto the stack.
- If a right parenthesis is encountered, it is **popped** off the stack.
- If the stack is empty and a right parenthesis is encountered, the expression is unbalanced.
- If the stack is not empty at the end of the expression, the expression is also unbalanced.

Balancing Parentheses Using Stack

```
def is_balanced(expression):
    stack = Stack()
    for char in expression:
        if char == '(' or char == '[' or char == '{':
            stack.push(char)
        elif char == ')' or char == ']' or char == '}':
            if stack.is_empty():
                return False
            if char == ')' and stack.peek() != '(':
                return False
            if char == ']' and stack.peek() != '[':
                return False
            if char == '}' and stack.peek() != '{':
                return False
            stack.pop()
    return stack.is_empty()
```

- If a right parenthesis is encountered, it is **popped** off the stack.
- If the stack is empty and a right parenthesis is encountered, the expression is unbalanced.
- If the stack is not empty at the end of the expression, the expression is also unbalanced.

Arithmetic Expressions

Infix Notation

- Infix notation is the standard mathematical notation in which operators are written between the operands.
- The main advantage of infix notation is that it is very easy to read and understand.
- not the most efficient way to represent expressions for a computer to process.

"2 + 3" in infix notation.

Arithmetic Expressions

Prefix Notation

- Prefix notation, also known as Polish notation, where the operators precede the operands
- Introduced by Polish logicial **Jan Łukasiewicz** in the 1920s
- it is very easy for a computer to evaluate expressions;
  - because prefix notation eliminates the need for parentheses
  - and the need to know the rules of operator precedence.

Arithmetic Expressions

Postfix Notation

"2 + 3" would be written as "2 3 +" in postfix notation.

- also known as reverse Polish notation (RPN), is a way of representing mathematical expressions where the operators follow the operands.
- Similar benefits as prefix
- Charles L. Hamblin in the 1950s, was interested in developing a notation that was easy to read and evaluate using a computer.

Ease of evaluation:

- RPN expressions can be evaluated easily using a **stack-based algorithm**
- while Polish notation expressions can be evaluated using recursive function calls or by using a stack-based algorithm with more complex rules.

Evaluation of Postfix expressions

“2 + (3 * 1) – 5”

"2 3 1 * + 5 -"

1.Create an empty stack.

2.Read "2": Push "2" onto the stack.
3.Read "3": Push "3" onto the stack.
4.Read "1": Push "1" onto the stack.

5.Read "*": Pop "1" and "3" from the stack, form the expression "(3 * 1)" and push the result back onto the stack.

6.Read "+": Pop "(3 * 1)" and "2" from the stack, form the expression "2 + (3 * 1)" and push the result back onto the stack.

7.Read "5": Push "5" onto the stack.

8.Read "-": Pop "5" and "2 + (3 * 1)" from the stack, form the expression "2 + (3 * 1) - 5" and push the result back onto the stack.

9.The expression has been fully processed, and the result on the top of the stack, "2 + (3 * 1) - 5", is the equivalent infix expression.

- If the element is a operand, push it into the stack
- If the element is an operator, pop two operands for the operator from the stack.
  - Evaluate the operator and push the result back to the stack

Conversions of Infix to Postfix expression

1. Initialize an **empty stack**.
2. Initialize an empty output string (i.e. the postfix).
3. For each character in the input expression:
   a. If the character is an operand, append it to the output string.
   b. If the character is an operator, *pop operators from the stack and append them to the output string* until you encounter an operator with lower precedence or an open parenthesis. Then push the operator onto the stack.
   c. If the character is an open parenthesis, push it onto the stack.
   d. If the character is a close parenthesis, pop operators from the stack and append them to the output string until you encounter an open parenthesis. Then discard the parenthesis.
4. After the input expression has been fully processed, pop any remaining operators from the stack and append them to the output string.
5. The final output string is the postfix expression.

((A+B)*C-D)

Considered	Stack	Postfix
(	(	
((	((	
((A	((	A
((A+	((+	A
((A+B	((+	AB
((A+B)	(	AB+
((A+B)*	(*	AB+
((A+B)*C	(*	AB+C
((A+B)*C-	(-	AB+C*
((A+B)*C-D	(-	AB+C*D
((A+B)*C-D)		AB+C*D-