Abstract Data Types (ADTs)
(Part I)

Outline

- Abstract Data Types (ADTs)
  - Definition
  - ADTs vs Data Structures
  - Client vs Implementation
  - Benefits
- Examples of ADTs
  - Stack
    - Implementation using array and linked list
    - Application: Postfix Notation
  - Queue
    - Array and Linked List
    - Application: Breadth-first search on a graph
    - Priority queue

Definition of ADT

- ADTs are high-level descriptions of a set of **operations and their expected behaviors** on a collection of data.
  – Example: Stack

Operations	Behavior
Push	The newly added element is on the top.
Pop	The most recently added element is removed.

- They don't specify how operations are implemented.
  – Example: a Stack ADT doesn’t specify whether an array or a linked list is used for implementation.

ADTs vs Data Structures

- Difference between ADTs and data structures
  - ADT provide a high-level description for operations;
  - Data structure is its concrete implementation.
- ADTs can be implemented using various data structures and algorithms.
- Different implementations offer trade-offs in time and space complexity.

Client vs Implementation

- **Client**: A program or module that uses services provided by an ADT.
- It interacts with the ADT through the defined interfaces.
- **Implementation**: The concrete realization of an ADT using data structures and algorithms.
- It involves making design decisions and ensuring correct functionality and efficiency.

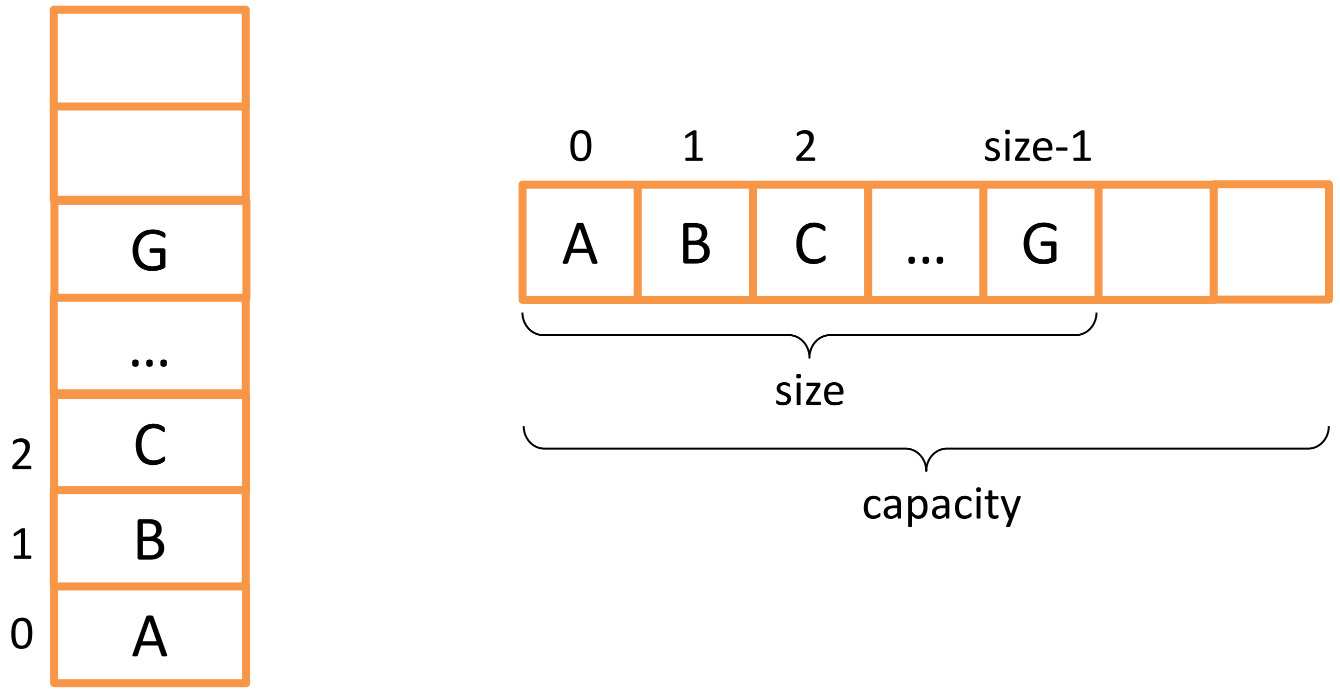
Benefits of ADTs

- Modularity
  - Separation of the client code and the implementation code
  - It enhances maintainability and simplifies debugging.
- Flexibility
  - Multiple implementations allow for adaptability, accommodating different scenarios and requirements.
  - Changes to the implementation do not affect the client code as long as the interface remains consistent.

Stack

- The stack is an ADT that adopts a **LIFO** (last-in first-out) policy.
- Two basic operations:
  - Add (push): New item added to the top
  - Remove (pop): top item (most recently added item) is removed

Stack Implementation using Array



Stack Implementation using Array

Implementation

```
class Stack1 {
   private:
     char *items;
     int size;    // Current size of the stack
     int capacity;// Maximum size of the stack

   public:
     Stack1(int capacity){
       this->capacity = capacity;
       items = new char [capacity];
       size = 0;
     }

     ~Stack1(){
       delete [] items;
     }

     void Push(char item){
        if (size>=capacity)
           return;
        items[size++] = item;
     }

     char Pop(void){
        return items[--size];
    }

     bool IsEmpty(void){
        return (size == 0);
     }
};
```

Client

```
int main(void){
   const int SIZE = 10;
   Stack1 stack(SIZE);

   const char *p = "ABCDEFG";
   for (unsigned i = 0; i < strlen(p); ++i)
      stack.Push(p[i]);

   while (!stack.IsEmpty())
      cout << stack.Pop();

   cout << endl;
   return 0;
}
```

Can we modify the above implementation to accept any data type?

Stack Implementation using Array (Modified)

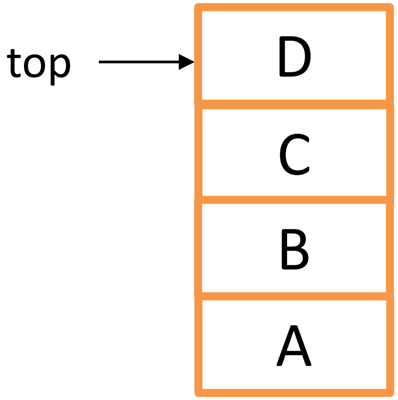
```
class Stack1 {
   private:
     char *items;
     int size;     // Current size of the stack
     int capacity;// Maximum size of the stack

   public:
     Stack1(int capacity){
       this->capacity = capacity;
       items = new char [capacity];
       size = 0;
     }

     ~Stack1(){
       delete [] items;
     }

     void Push(char item){
        if (size>=capacity)
           return;
        items[size++] = item;
     }

     char Pop(void){
        return items[--size];
    }

     bool IsEmpty(void){
        return (size == 0);
     }
};
```

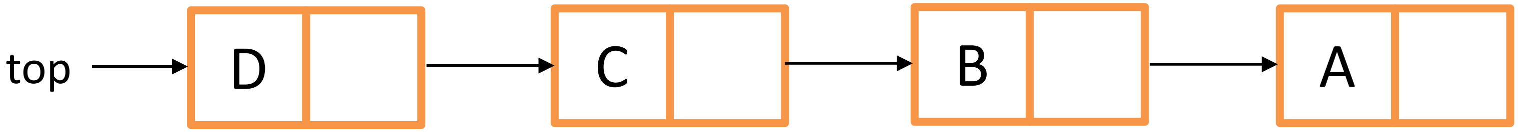
```
template <typename T>
class Stack1 {
   private:
     T *items;
     int size;     // Current size of the stack
     int capacity;// Maximum size of the stack

   public:
     Stack1(int capacity){
       this->capacity = capacity;
       items = new T [capacity];
       size = 0;
     }

     ~Stack1(){
       delete [] items;
     }

     void Push(T item){
        if (size>=capacity)
           return;
        items[size++] = item;
     }

     T Pop(void){
        return items[--size];
    }

     bool IsEmpty(void){
        return (size == 0);
     }
};
```

Stack Implementation using Array

- Advantages
  - Simple implementation
  - Constant time access: $O(1)$
  - Better cache locality
- Disadvantages
  - Fixed size with maximum capacity
  - Cost of Resizing: $O(n)$
  - Wasted memory: If the array size is greater than the actual number of elements in the stack
  - Not Suitable for applications where the size of the stack is not known in advance or fluctuates significantly

Stack Implementation using Linked List





Stack Implementation using Linked List

```
template <typename T>
class Node {
    public:
        T data;
        Node* next;
        Node(T value){
            data = value;
            next = 0;
        }
};
```

```
template <typename T>
class Stack2 {
    private:
        Node<T>* top;
        int size;
        int capacity;

    public:
        Stack2(int capacity) {
            top = 0;
            size = 0;
            this->capacity = capacity;
        }

        ~Stack2() {
       // Traverse the list to delete each item
            while (top) {
                Node<T>* temp = top;
                top = top->next;
                delete temp;
            }
        }
```

Stack Implementation using Linked List

```
        void Push(T value) {
            Node<T>* newNode = new Node<T>(value);
            newNode->next = top;
            top = newNode;
            ++size;
        }

        T Pop() {
            T poppedValue = top->data;
            Node<T>* temp = top;
            top = top->next;
            delete temp;
            --size;
            return poppedValue;
        }

        bool IsEmpty() const {
            return top == 0;
        }
};
```

The same client code can be used.

Stack Implementation using Linked List

- Advantages
  - Dynamic size
  - No Preallocation
  - Easy resizing: $O(1)$
- Disadvantages:
  - Memory overhead due to the storage of pointers
  - Poor cache locality

Choice of Stack Implementation

- The choice of implementing a stack depends on the specific requirements of the application. E.g.,
  - If your stack size is known and relatively fixed, an array-based implementation may be more suitable.
  - If the size of the stack varies dynamically and you want to avoid preallocation, a linked list may be more suitable.
  - If efficient memory access and cache locality are crucial, an array-based implementation might be more suitable.

Stack Application: Evaluating Postfix Expressions

- Arithmetic expressions usually use infix notation where operators are between operands: 3+4, 5*7+2
  - The order of operations is determined by the precedence of operators.
  - Parentheses are used to determine the order of evaluation: a*(b+c)

- Postfix notation has the operators after the operands:
  - 34+ = 3 + 4
  - 57*2+ = 5*7 + 2
  - abc+* = a*(b+c)
  - Operations are conducted in the order from left to right.
  - No need for parentheses as the order evaluation is explicit.

Examples

- Infix notation, with parenthesis
  - 5 * ((9 + 8) * (4 + 6) + 7) = 885
- Infix notation, without parenthesis
  - 5 * 9 + 8 * 4 * 6 + 7 = 244
- Postfix notation
  - 598+46+*7+* = 885

- Infix notation, with parenthesis
  - 5 * 9 + (8 * 4) * (6 + 7) = 461
- Postfix notation
  - 59*84* 67+*+ = 461

- Note: Operands appear in the same order in infix and postfix expressions.

Convert Infix to Postfix by Hand

- Fully parenthesize the expression.
  - Enclose each operator and its operands with a pair of parenthesis.

$$5 * 9 + (8 * 4) * (6 + 7)$$
$$= ((5 * 9) + ((8 * 4) * (6 + 7)))$$

- Move each of the operators immediately to the right of their respective right parentheses.

59*84* 67+*+

- Exercise: Convert
  - (3 + 6) * (2 - 4) + 7

How to evaluate a postfix expression?

- A stack is the perfect data structure to implement this paradigm.
- Algorithm for evaluating postfix expressions :
  - For each token in the postfix expression:
    - If it is an operand, push it onto the stack.
    - If it is an operator,
      1. Pop two operands from the stack: operand1 and operand2
      2. Perform the arithmetic: operand1 operator operand2
      3. Push the result of the arithmetic onto the stack
  - When there is no more token, the answer is on the top of the stack (it will be the only item on the stack.)

Example: Evaluate a Postfix Expression

- Evaluate 34+6*

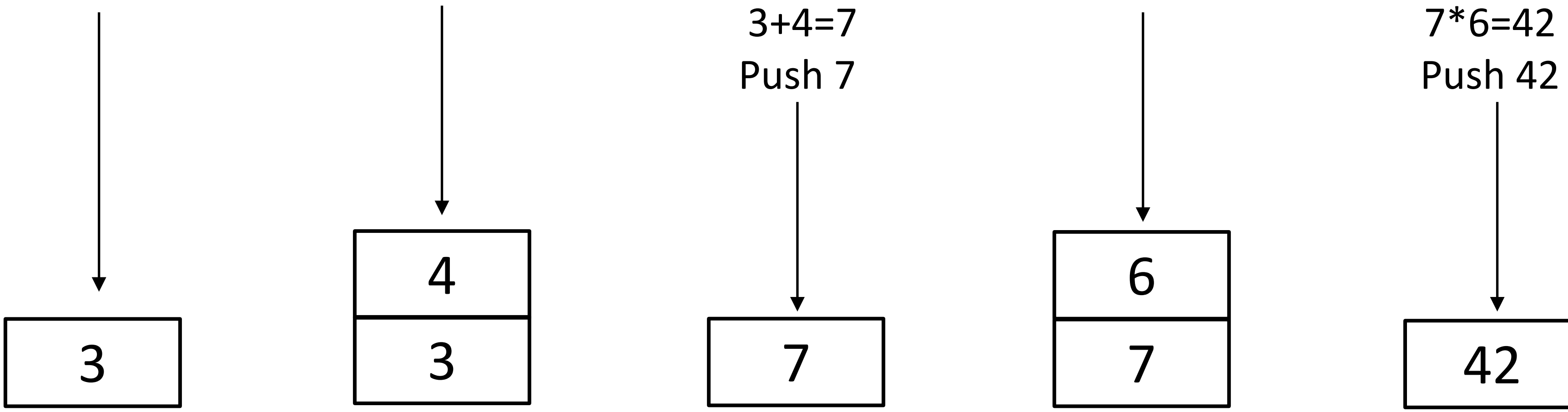
Token = ‘3’
Push 3

Token = ‘4’
Push 4

Token = ‘+’
Pop twice
3+4=7
Push 7

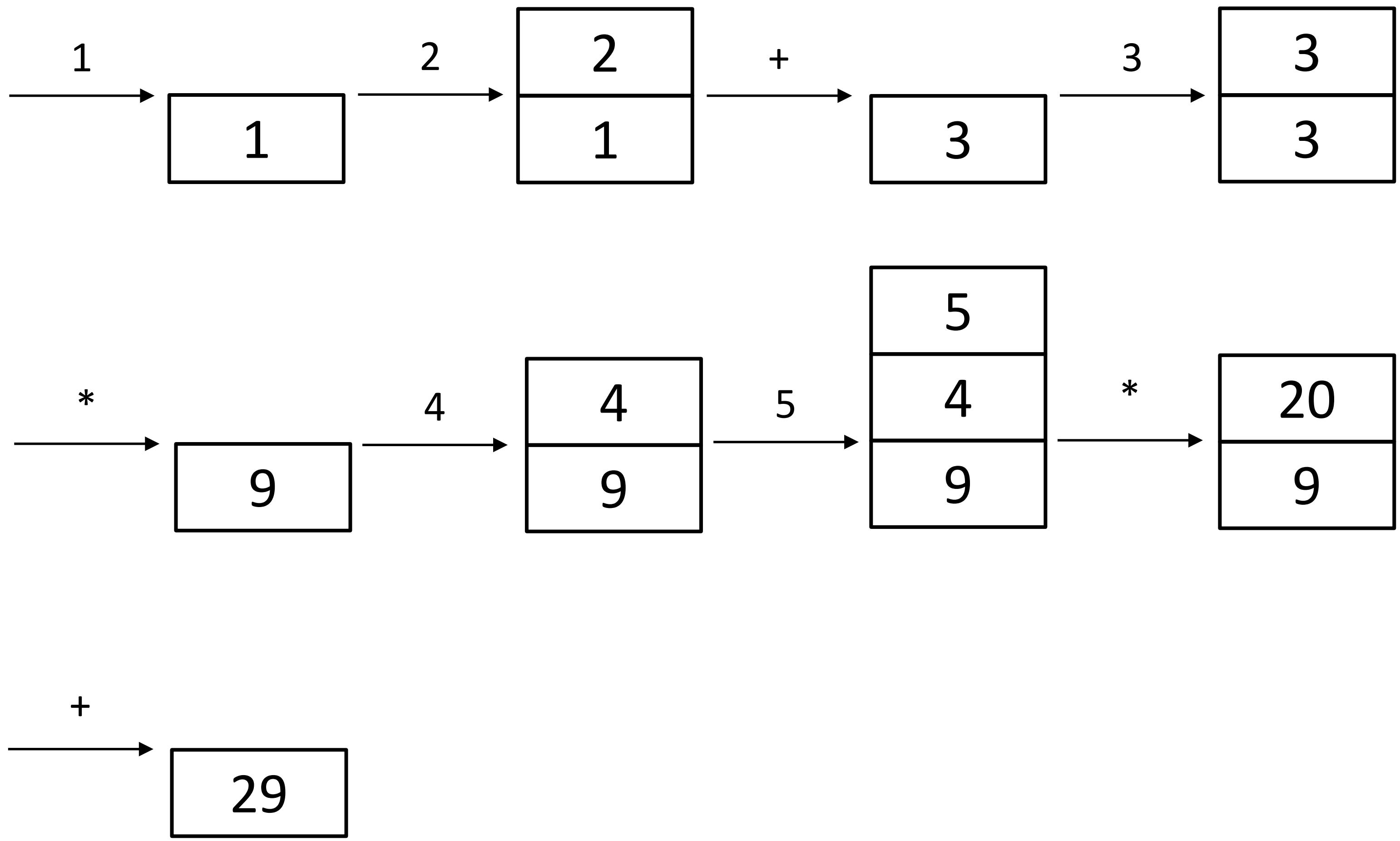
Token = ‘6’
Push 6

Token = ‘*’
Pop twice
7*6=42
Push 42



Example: Evaluate a Postfix Expression

- Evaluate 12+3*45*+



Implementation: Evaluate Postfix Expressions

```
int Evaluate_Postfix(const char * postfix)
{
        Stack<int> stack(strlen(postfix));

        while(*postfix)
        {
              char token = *postfix;

              if(token == '+')
                   // Pop two values, add them, push the result back onto the stack
                    stack.Push(stack.Pop() + stack.Pop());
              else if(token == '*')
                    stack.Push(stack.Pop() * stack.Pop());
              else if (token >= '0' && token <= '9')
                    // Convert the character to its corresponding integer value
                    // and push it onto the stack.
                    stack.Push(token - '0');
              postfix++;
        }
        return stack.Pop();
}
```

```
void main (void)
{
  char postfix [256];

  cout << "Enter the operations" << endl;
  cin.width(256);
  cin >> postfix;
  cout << postfix << " = " << Evaluate_Postfix(postfix) << endl;
}
```

```
598+46+*7+* = 885

34+ = 7

34+7* = 49

12*3*4*5*6* = 720
```

Implementation: Evaluate Postfix Expression

- If we want to modify the above function to support subtraction and division, does the code below work?

```
else if(token == '-')
    stack.Push(stack.Pop()-stack.Pop());
else if(token == '/')
    stack.Push(stack.Pop()/stack.Pop());
```

- Note: You'll need to pay attention to the order of operands.
- Try it with the postfix expression `28*4/56*+8-`
  - Infix: `2*8/4+5*6-8 = 26`