Data Structures

Dilip Kumar Maripuri
Computer Applications

Data Structures

Session : Concept of Stack, Stack as ADT, Implementation using Singly linked list and Dynamic Arrays

Dilip Kumar Maripuri
Computer Applications

- Stacks are a widely used data structure in computer science, operating on the Last In, First Out (LIFO) principle.
- They are abstract data types (ADTs) with defined operations that include pushing (adding an element), popping (removing the top element), peeking (viewing the top element without removing it), and checking if the stack is empty.

- **Basic Concepts of a Stack**
  - **Definition**: A stack is a linear data structure that restricts access to only the most recent element added, known as the "top." This structure enforces a LIFO or FILO (First In, Last Out) access pattern.
  - **Analogy**: The stack structure can be visualized as a stack of plates where you can only take the top plate off or add a new one on top.

- **Operations in a Stack**
  - **Push**: Adds an element to the top of the stack.
  - **Pop**: Removes the top element from the stack.
  - **Peek/Top**: Returns the top element without removing it.
  - **IsEmpty**: Checks if the stack is empty.

Core Operations of a Stack

- **1. Push Operation**
  - **Description**: The push operation adds an element to the top of the stack. In conceptual terms, think of adding a plate on top of a stack; only the top plate can be pushed or popped.
  - **Time Complexity**: $O(1)$ because it involves placing an element directly on the top, regardless of the stack's size.
  - **Post-Condition**: After a push operation, the new element becomes the top of the stack, and the previous elements remain as they are below it.

- **2. Pop Operation**
  - **Description**: The pop operation removes the top element of the stack. Conceptually, this is like removing the top plate from a stack of plates.
  - **Time Complexity**: $O(1)$ since it only involves removing the element at the top position.
  - **Post-Condition**: After popping, the next element in line becomes the top of the stack. If the stack was empty before the operation, it returns an error (stack underflow).

- **3. Peek (Top) Operation**
  - **Description**: The peek or top operation allows you to view the element at the top of the stack without removing it. This can be useful for checking the latest added item or the state of the stack.
  - **Time Complexity**: $O(1)$, as it simply reads the top element.
  - **Post-Condition**: The stack remains unchanged, as the operation only inspects the top element.

- **4. IsEmpty Operation**
  - **Description**: This operation checks whether the stack is empty.
  - **Time Complexity**: $O(1)$, as it simply checks the stack size or the position of the top element.
  - **Post-Condition**: The stack remains unchanged, and the operation returns a boolean indicating whether the stack has any elements.

- **5. Size Operation**
  - **Description**: The size operation returns the number of elements currently in the stack.
  - **Time Complexity**: $O(1)$ if the stack keeps track of its size, or $O(n)$ if it needs to count elements.
  - **Post-Condition**: The stack remains unchanged, providing a count of elements stored within it.

Data Model of a Stack

- The stack data model is a **linear collection of elements** with access limited to one end, termed the ”top.” The data model works as below:
    - **LIFO Principle**: Last-In-First-Out means the last element added is the first to be removed.
    - **Single Access Point**: All operations (push, pop, peek) occur at the ”top” of the stack.
    - **Fixed or Dynamic Size**: Some stack implementations (e.g., array-based) are fixed in size, whereas others (e.g., linked list-based) are dynamic and limited only by system memory.

Summary of Stack Operations

Operation	Description	Time Complexity	Post-Condition
Push	Adds an element to the top of the stack	$O(1)$	New element becomes the top
Pop	Removes the top element from the stack	$O(1)$	Top element is removed, next element becomes the top
Peek/Top	Returns the top element without removing it	$O(1)$	Stack remains unchanged
IsEmpty	Checks if the stack has any elements	$O(1)$	Stack remains unchanged
Size	Returns the number of elements in the stack	$O(1)$ or $O(n)$	Stack remains unchanged

Implementation	Pros	Cons
Array-Based Stack	▶ **Simple and Fast**: Operations like push, pop, and peek have $O(1)$ time complexity. ▶ **Memory Efficient**: No overhead from pointers.	▶ **Fixed Size**: Requires a predefined size, leading to potential stack overflow if the capacity is exceeded. ▶ **Inflexible**: Cannot resize at runtime.

Implementation	Pros	Cons
Dynamic Array-Based Stack	▶ **Resizable**: Automatically resizes when the stack reaches capacity, providing more flexibility.	▶ **Resizing Overhead**: Resizing is costly in terms of time and memory, as elements need to be copied to a larger array. ▶ **Fragmentation**: Frequent resizing may lead to memory fragmentation.

Implementation	Pros	Cons
Linked List-Based Stack	▶ **Flexible Size**: No predefined limit, as nodes are dynamically allocated, allowing the stack to grow as needed. ▶ **Efficient Push and Pop**: Push and pop operations are $O(1)$ without the need for resizing.	▶ **Memory Overhead**: Requires extra memory for pointers in each node. ▶ **Pointer Management**: Managing pointers adds complexity and increases the risk of memory leaks if nodes are not handled correctly.

- Choosing the right implementation depends on the specific needs:
  - **Array-based stacks** are suitable for small, fixed-size stacks.
  - **Dynamic arrays** are effective when resizing is occasionally required.
  - **Linked lists** work well when unlimited growth is needed, but they have higher memory overhead.

Implementation of Stacks using Arrays

```c
#define MAX 10

typedef struct StackADT {
    int stk[MAX];
    int TOP;
} *Stack;

int IsFull(Stack MyStack) {
    return ((MyStack->TOP) == (MAX - 1) ? 1 : 0);
}

int IsEmpty(Stack MyStack) {
    return ((MyStack->TOP) == -1 ? 1 : 0);
}
```

Implementation of Stacks using Arrays - PUSH Operation

```
    void push(Stack MyStack) {
    int element;
    if (IsFull(MyStack))
        printf("\n\t\t Stack Overflow");
    else {
        printf("\n\t\t Enter the Element : ");
        scanf("%d", &element);
        MyStack->stk[++MyStack->TOP] = element;
    }
    display(MyStack);
}
```

Implementation of Stacks using Arrays - POP Operation

```
void pop(Stack MyStack) {
    if (IsEmpty(MyStack))
        printf("\n\t\t Stack Underflow !!!");
    else
        printf("\n\t\t %d is Popped", MyStack->
            stk[(MyStack->TOP)--]);
    display(MyStack);
}
```

Implementation of Stacks using Arrays - Display Operation

```c
void display(Stack MyStack) {
    int i;
    if (IsEmpty(MyStack))
        printf("\n\t\t Empty Stack");
    else {
        printf("\n\t TOP -> |");
        for (i = MyStack->TOP; i >= 0; i--)
            printf("%d | ", MyStack->stk[i]);
    }
}
```

Implementation of Stacks using Arrays - declaration in main()

```
Stack MyStack = (Stack)malloc(sizeof(struct
    StackADT));
MyStack->TOP = -1;
```

Implementation of Stacks using Dynamic Arrays

```c
typedef struct MyStackADT {
    int *array;
    int top;
    int capacity;
} *Stack;

Stack CreateStack(int capacity) {
    Stack NewStack = (Stack)malloc(sizeof(struct MyStackADT));
    if (NewStack == NULL) {
        printf("\n\t\t Insufficient Memory !!! Exiting !!!!");
        exit(EXIT_FAILURE);
    }
```

Implementation of Stacks using Dynamic Arrays

```c
    NewStack->array = (int *)malloc(sizeof(int) * capacity);
    if (NewStack->array == NULL) {
        printf("\n\t\t Insufficient Memory !!! Exiting !!!!");
        exit(EXIT_FAILURE);
    }

    NewStack->top = -1;
    NewStack->capacity = capacity;

    printf("\n\t\t Successfully Created Stack ADT");
    return NewStack;
}
```

Implementation of Stacks using Dynamic Arrays

```c
int Do_Resize(Stack MyStack, int inc_dec) {
    int NewSize = MyStack->capacity + inc_dec;
    int *NewArray = (int *)realloc(MyStack->
        array, sizeof(int) * NewSize);

    if (NewArray == NULL) {
        printf("\n\t\t Insufficient Memory !!!
            Exiting !!!!");
        return 0;
    }

    MyStack->capacity = NewSize;
    MyStack->array = NewArray;
    return 1;
}
```

Implementation of Stacks using Dynamic Arrays

```c
void push(Stack MyStack) {
    int element, status;
    printf("\n\t\t Enter the element to be pushed onto the stack: ");
    scanf("%d", &element);
    status = (MyStack->top == MyStack->capacity - 1) ? Do_Resize(MyStack, 1) : 1;
    if (status)
        MyStack->array[++MyStack->top] = element;
    else
        printf("\n\t\t Stack Overflow !!!");
    display(MyStack);
}
```

Implementation of Stacks using Dynamic Arrays

```c
void pop(Stack MyStack) {
    int element, status;
    if (MyStack->top == -1)
        printf("\n\t\t Stack Underflow !!!");
    else {
        printf("\n\t\t Popped Element is %d",
            MyStack->array[MyStack->top--]);
        status = Do_Resize(MyStack, -1);
    }
    display(MyStack);
}
```

Implementation of Stacks using Dynamic Arrays

```c
void display(Stack MyStack) {
    int i;
    printf("\n\t\t Display() %d", MyStack->top);
    if (MyStack->top == -1)
        printf("\n\t\t Empty Stack !!");
    else {
        printf("\n\t Stack Contents \n\t\t TOP
            -> |");
        for (i = MyStack->top; i >= 0; i--)
            printf(" %d |", MyStack->array[i]);
    }
}
```

Implementation of Stacks using Dynamic Arrays

```c
int peek(Stack MyStack) {
    if (MyStack->top == -1) {
        printf("Stack is Empty\n");
        return INT_MIN;
    }
    return MyStack->array[MyStack->top];
}

void DestroyStack(Stack MyStack) {
    free(MyStack->array);
    free(MyStack);
}

// inside main function'
    Stack MyStack = CreateStack(1); (Initial
        Size is 1)
```

- **Observations**
  - When pushing an item onto a full stack, the stack's capacity is increased by one.
  - When popping an item, the stack's capacity is decreased by one, maintaining just enough space for the current elements plus one extra space.

- **Observations**
  - This approach ensures that the stack is always optimally sized for its current load, but it may lead to frequent memory reallocations if elements are frequently pushed and popped, which can be inefficient.
  - Remember, while this approach is dynamic and adapts to the stack size, frequent resizing could lead to performance overhead due to the cost of memory allocation and deallocation.
    - It's a trade-off between memory efficiency and computational efficiency.

Implementation of Stacks using Linked Lists

```c
NODE push(NODE TOP, int data) {
    return (sll_ins_front(data, TOP));
}

NODE pop(NODE TOP) {
    if (TOP == NULL) {
        printf("Stack Underflow\n");
        return -1;
    }
    int data = TOP->data;
    return (sll_delete_first(TOP));
}
```

Implementation of Stacks using Linked Lists

```c
int peek(NODE TOP) {
    if (TOP == NULL) {
        printf("Stack is Empty\n");
        return -1;
    }
    return TOP->data;
}

void displayStack(NODE TOP) {
    sll_display(TOP);
}
```

- **Balanced Parentheses Checker**
- **Objective**: Write a program that uses a stack to check for balanced parentheses in an expression. **Tasks**
  - Read an expression (string) from the user.
  - Use a stack to track open parentheses.
  - Check for matching closing parentheses for each open parenthesis.

  Extensions
  - Expand the program to check for other types of brackets (e.g., { }, [ ]).
  - Implement error messaging to show the position of the unbalanced parenthesis.

- **Create a Stack with a Min Function**

- **Objective** : Design a stack that, in addition to push and pop, has a function `min` which returns the minimum element.

Tasks

- Implement the stack as usual.
- Ensure the min function operates in $O(1)$ time.

Extensions

- Modify the stack to also have a max function for the maximum element.

- **Data Structure**
  - Let `stack` be the main stack holding all elements.
  - Let `minStack` be an auxiliary stack that tracks the minimum elements.
- **Push Operation**:
  - Push element onto stack.
  - Check if minStack is empty or if element is less than or equal to the top element of minStack.
  - If true, push element onto minStack as well.
  - **Result:** element is added to stack, and minStack is updated if element is the new minimum.

- **Pop Operation:**
  - Check if stack is empty. If true, return "Stack Underflow".
  - Pop the top element from stack and store it in poppedElement.
  - Check if poppedElement is equal to the top element of minStack.
  - If true, pop the top element from minStack as well.
  - **Result:** The top element is removed from stack, and minStack is updated if the minimum element was removed.

- **Min Operation:**
  - Check if minStack is empty. If true, return "Stack is Empty".
  - Return the top element of minStack (this is the minimum element).
  - **Result:** Returns the minimum element in O(1) time.

Thank You

Dilip Kumar Maripuri

Associate Professor

Department of Computer Applications

dilip.maripuri@pes.edu

8073212026