Data Structures and Algorithms (Spring2024)

Week07 (27/28/28-Mar-2024)

M Ateeq,
Department of Data Science, The Islamia University of Bahawalpur.

Stacks

Stacks are a fundamental data structure in computer science, embodying a simple yet powerful organizing principle. At their core, stacks are a collection of elements with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element. This principle is commonly summarized by the acronym LIFO, standing for Last In, First Out. It encapsulates the essence of stack operations, where the last element added to the stack is the first one to be removed.

Definition and Concept

A stack is conceptualized as a linear collection of elements with restrictions on where elements can be added or removed. These restrictions imbue stacks with unique properties and make them suitable for various computational tasks. The operations on a stack are performed at only one end, referred to as the "top" of the stack, in contrast to the "bottom."

The primary operations associated with stacks are:

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the element on the top of the stack, effectively reducing its size by one.
- **Peek** or **Top**: Returns the element at the top of the stack without removing it, allowing a look at the most recent element.

Additional utility operations might include checking if the stack is empty to prevent a pop operation on an empty stack, which would be an erroneous operation.

Characteristics of Stacks (LIFO - Last In, First Out)

The LIFO principle is what differentiates stacks from other data structures. This characteristic means that the most recently added (or "pushed") element is always the first one to be removed (or "popped") from the collection. This principle is critical in various applications, such as parsing expressions in compilers, managing function calls in programming languages, and undo mechanisms in applications, where the most recent action is the first to be reversed.

The LIFO property ensures that the order of operations is preserved in a way that the last unresolved action or the most recent context is always addressed first, facilitating a straightforward implementation of algorithms that require backtracking or sequential processing of nested structures.

Real-world Analogy (e.g., a stack of plates)

A common real-world analogy for a stack is a stack of plates in a cafeteria. Consider a vertical pile of plates where a new plate can be added to the top, and plates can only be removed from the top. When a plate is cleaned and ready to be stacked, it is placed on the top of the pile. When someone needs a plate, they take the top one from the stack. This process does not affect the order of the remaining plates and ensures that the most recently washed plate is used first, demonstrating the LIFO principle in a tangible context.

This analogy helps to intuitively understand the operations and limitations of stacks. Just as it would be impractical to remove a plate from the bottom of the stack without disturbing the others, in computational terms, accessing elements out of order in a stack would violate its fundamental design and operational constraints.

Summary

Stacks, with their simple yet powerful LIFO characteristic, are indispensable in computer science for tasks requiring ordered backtracking, sequential processing, or nested operations. The conceptual clarity of stacks, paralleled with real-world analogies like a stack of plates, helps in understanding their practical applications and limitations.

Basic Operations on Stacks

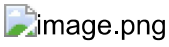
Stacks are dynamic data structures that follow the Last In, First Out (LIFO) principle. This section delves into the fundamental operations that define the functionality of stacks: Push, Pop, Peek (or Top), and isEmpty. Each of these operations plays a critical role in manipulating the stack and accessing its elements.

Push: Adding an Element to the Top of the Stack

The Push operation involves adding a new element to the top of the stack. This is the primary means of inserting data into the stack. Since stacks are LIFO structures, the most recently added element is always positioned at the top, ready to be the first one removed.

Implementation Insight: In a stack implemented using an array, the push operation increments the top index and places the new element at this updated position. In a linked list implementation, it creates a new node and links it to the previous top node.

```
void push(int data) {
    // Check if stack (or underlying storage) is full, then add the element
    Node* newNode = new Node(data);
    newNode->next = top; // Link the new node with the current top
    top = newNode;       // Update top to the new node
}
```



Pop: Removing the Top Element of the Stack

Pop operation removes the element at the top of the stack, effectively reducing its size by one. This operation adheres to the LIFO principle, ensuring that the last element added is the first to be removed.

Implementation Insight: The pop operation checks if the stack is not empty, accesses the top element, updates the top to the next element (or decrements the top index in array-based implementations), and finally returns the popped element.

```
int pop() {
    if (isEmpty()) {
        // Handle underflow or return a sentinel value indicating the stack is empt
y
    }
    int poppedData = top->data; // Retrieve data to return
    Node* temp = top;           // Temporarily store the top node to delete
    top = top->next;            // Move top to the next node
    delete temp;                // Free the memory of the popped node
    return poppedData;
}
```

Peek/Top: Viewing the Top Element Without Removing It

The Peek or Top operation allows observing the element at the top of the stack without removing it. This operation is essential for assessing the most recently added element's value when removing it from the stack is not desired.

Implementation Insight: This operation is straightforward; it simply returns the data held by the top element. It must ensure the stack is not empty to avoid accessing an undefined element.

```
int peek() {
    if (!isEmpty()) {
        return top->data; // Return the top element's data
    }
    // Handle empty stack case appropriately
}
```

isEmpty: Checking if the Stack is Empty

The isEmpty operation checks whether the stack contains any elements. It's a critical safety check that prevents errors such as stack underflow, where an attempt is made to remove an element from an empty stack.

Implementation Insight: In an array-based implementation, this could be checking if the top index is at its initial position. In a linked list, it checks if the top pointer is `nullptr`.

```
bool isEmpty() {
    return top == nullptr; // Returns true if the stack is empty
}
```

Implementation of Stack Using Arrays

Implementing a stack using arrays involves using a fixed-size array along with a top pointer (or index) to keep track of the current stack's top element. This method is one of the simplest and most direct ways to create a stack, leveraging the contiguous memory allocation of arrays for efficient element access. However, it also introduces limitations regarding the stack's maximum size, which must be defined at compile time or initialization.

Core Components

- **Array**: A fixed-size array stores the stack elements. The size of this array determines the maximum number of elements the stack can hold.
- **Top Pointer (Index)**: An integer that tracks the index of the top element in the stack. It is used to add (push) and remove (pop) elements from the stack. Initially, it can be set to -1 to indicate that the stack is empty.

Implementation Details

Initializing the Stack

Upon initialization, the array is allocated with a predetermined size, and the top index is set to -1.

```
class Stack {
    int* arr;
    int top;
    int capacity;

public:
    Stack(int size) {
        arr = new int[size]; // Allocate array of given size
        capacity = size;
        top = -1; // Initialize top to -1, indicating an empty stack
    }
```

Push Operation

The push operation adds an element to the top of the stack. It first checks if the stack is full (to prevent overflow), then increments the top index and stores the new element at this position.

```
    void push(int data) {
            if (top >= capacity - 1) { // Check for stack overflow
                cout << "Stack overflow" << endl;
                return;
            }
            arr[++top] = data; // Place data at the current top position after incremen
    ting
        }
```

Pop Operation

The pop operation removes the element from the top of the stack. It checks if the stack is empty (to prevent underflow), then returns the element at the top index and decrements the top.

```
    int pop() {
            if (top == -1) { // Check for stack underflow
                cout << "Stack underflow" << endl;
                return -1;
            }
            return arr[top--]; // Return the top element and decrement the top index
        }
```

Peek/Top Operation

The peek operation returns the top element without removing it from the stack, first ensuring the stack is not empty.

```
    int peek() {
            if (top != -1) return arr[top];
            cout << "Stack is empty" << endl;
            return -1; // Sentinel value or error code
        }
```

isEmpty Operation

This utility function checks whether the stack is empty by looking at the top index.

```
    bool isEmpty() {
            return top == -1;
        }

        ~Stack() {
            delete[] arr; // Destructor to free allocated memory
        }
    };
```

Advantages and Limitations

- **Advantages**:
  - Simplicity: The implementation is straightforward and easy to understand.
  - Performance: Accessing the top element, pushing, and popping are all performed in constant time, (O(1)).
- **Limitations**:
  - Fixed Capacity: The maximum size of the stack must be known in advance and cannot be changed at runtime, which could either waste memory or limit the stack's usability.
  - No dynamic resizing: Unlike a stack implemented with a linked list, an array-based stack does not grow or shrink dynamically based on current needs.

Implementation of Stack Using Linked Lists

Implementing a stack using linked lists offers a dynamic and flexible approach to stack data structures. Unlike arrays, linked lists do not have a fixed size, allowing the stack to grow and shrink as needed without the overhead of managing capacity or dealing with stack overflow (unless system memory is exhausted).

Core Components

- **Node Structure**: Each element in the stack is represented by a node in the linked list. Each node contains the data and a pointer to the next node down in the stack.
- **Top Pointer**: A pointer to the top node of the stack. The stack operations are performed at this end.

Implementation Details

Defining the Node and Stack Classes

First, define a `Node` class that will represent each element in the stack:

```
class Node {
public:
    int data;
    Node* next;

    Node(int data): data(data), next(nullptr) {}
};
```

Next, define the `Stack` class, which will use the `Node` class for storing data:

```
class Stack {
    Node* top;

public:
    Stack(): top(nullptr) {}

    ~Stack(); // Destructor for cleanup
```

Push Operation

The push operation adds a new element to the top of the stack. In a linked list implementation, this involves creating a new node and linking it to the current top node.

```
void push(int data) {
        Node* newNode = new Node(data);
        newNode->next = top; // New node points to the current top
        top = newNode;       // Top is updated to the new node
    }
```

Pop Operation

The pop operation removes the element from the top of the stack. It checks if the stack is empty to prevent underflow, removes the top node, and returns its data.

```
int pop() {
        if (isEmpty()) {
            std::cout << "Stack underflow" << std::endl;
            return -1; // Indicate error or underflow
        }
        Node* temp = top;    // Store current top
        //int poppedData = top->data; // Data to return
        top = top->next;     // Move top to next node
        delete temp;         // Free memory of old top
        //return poppedData;
    }
```

Peek/Top Operation

The peek operation returns the data of the top element without removing it from the stack. It ensures the stack is not empty before accessing the top node's data.

```
int peek() {
        if (!isEmpty()) return top->data;
        std::cout << "Stack is empty" << std::endl;
        return -1; // Sentinel value or error code
    }
```

isEmpty Operation

This function checks whether the stack is empty by verifying if the top pointer is `nullptr`.

```
bool isEmpty() {
        return top == nullptr;
    }
```

Destructor

To prevent memory leaks, define a destructor to clear the stack when it is no longer needed:

```
~Stack() {
        while (!isEmpty()) {
            pop(); // Pop all elements to free allocated memory
        }
    }
};
```

Advantages and Limitations

- **Advantages**:
  - Dynamic Size: The stack can grow and shrink according to the needs of the program, eliminating the need for a predefined capacity.
  - Memory Efficiency: Memory is allocated and deallocated as elements are pushed and popped, potentially leading to more efficient memory usage compared to a fixed-size array implementation.
- **Limitations**:
  - Overhead: Each node in the linked list requires extra memory for the pointer to the next node, introducing some overhead.
  - Performance: While operations remain (O(1)), dynamic memory allocation and deallocation can be more costly than manipulating indices in an array-based implementation.

Applications of Stacks

Stacks, with their Last In, First Out (LIFO) nature, are not just a theoretical concept but a cornerstone in various practical applications in computer science and software development. Their unique property of storing elements in a sequential order where the last stored (or pushed) item is the first to be retrieved (or popped) makes them ideal for several critical operations, including function calls and recursion, undo mechanisms in applications, syntax parsing, and evaluating expressions.

Function Calls/Recursion

One of the fundamental uses of stacks in computing is to manage function calls within a program. When a function is called, its execution context (including parameters, local variables, and return address) is saved on a call stack. This stack structure ensures that, upon completion of the function, execution returns to the correct location in the program, and the function's context is appropriately managed, especially in the case of nested or recursive function calls.

In recursion, where a function calls itself, a new context for each call is pushed onto the stack. This allows each recursive call to operate with its variables without interfering with other instances of the function. The stack's LIFO nature ensures that the most recent call is resolved first, with each recursive call being popped off the stack after completion until it returns to the base case.

Undo Mechanisms in Text Editors

Stacks play a crucial role in implementing undo mechanisms in software applications, such as text editors or graphic design programs. Every action performed by the user can be pushed onto an "undo stack," where each stack element represents a state or an action. When the user triggers the undo command, the application pops the last action from the stack and reverses it, effectively restoring the previous state.

This stack-based approach can also be extended to support redo operations, where reversed actions are pushed onto a separate "redo stack," allowing users to move forward through the action history by popping from the redo stack.

Syntax Parsing

Compilers and interpreters use stacks for syntax parsing of programming languages. Syntax parsing involves checking, among other things, whether parentheses, brackets, and braces are properly opened and closed in the correct order. A stack is ideal for this purpose: each time an opening symbol is encountered, it is pushed onto the stack; when a closing symbol is seen, the stack is popped, and the popped symbol is checked against the current one to ensure they match. If the stack is empty when a closing symbol is encountered or not empty when the entire input has been processed, the syntax is incorrect.

Evaluating Expressions (Postfix/Prefix)

Stacks are extensively used in the evaluation of arithmetic expressions, especially in postfix (Reverse Polish Notation) or prefix notation. In postfix expression evaluation, the operands are pushed onto a stack as they are read. Upon encountering an operator, the necessary number of operands (typically two) are popped from the stack, the operation is performed, and the result is pushed back onto the stack. This process continues until the expression is fully evaluated, and the final result remains on the stack.

Prefix expressions are evaluated in a similar manner, but the process involves reading the expression from right to left, or reversing the expression and then applying the same algorithm as for postfix evaluation.

- Consider the evaluation of 4 - 5 + (6 * 2)
- In RPN this would be rewritten as 4 5 - 6 2 * +
- RPN evaluation:
  - when a number is seen it is pushed onto the stack
  - when an arithmetic operator is seen then two numbers are popped off the stack, the operator is applied, and the result is pushed on the stack
- the stack would undergo the following transformations:
  - (4)
  - (4 5)
  - (-1) after applying −
  - (-1 6)
  - (-1 6 2)
  - (-1 12) after applying ∗
  - (11) final result after applying +

Summary

The versatility and efficiency of stacks make them an indispensable tool in software development, offering elegant solutions for managing function calls, implementing undo functionalities, parsing syntax, and evaluating expressions. These applications highlight the stack's ability to manage ordered data in a way that is both intuitive and aligned with the operational needs of these tasks, demonstrating the practical importance of stacks in the field of computer science.

Complexity Analysis of Stacks

Analyzing the time and space complexity of basic operations on stacks is essential for understanding their efficiency and scalability. This analysis helps in selecting the appropriate data structure for a given problem based on performance requirements. The complexity of stack operations can vary depending on the underlying implementation (e.g., using arrays or linked lists), but the abstract data type itself offers remarkable efficiency for its operations.

Time Complexity of Basic Operations

- **Push Operation**: The push operation involves adding an element to the top of the stack. Regardless of the underlying implementation (array or linked list), adding an element to the stack does not require traversing the entire data structure. Therefore, the time complexity of the push operation is (O(1)), or constant time.
- **Pop Operation**: Similar to the push operation, popping an element from the stack involves removing the top element, which is directly accessible without the need to traverse other elements. Hence, the pop operation also has a time complexity of (O(1)), or constant time.
- **Peek/Top Operation**: The peek or top operation simply retrieves the element at the top of the stack without removing it. Since this operation does not involve traversal and the top element is directly accessible, its time complexity is (O(1)), or constant time.
- **isEmpty Operation**: Checking whether a stack is empty is a straightforward operation that typically involves checking if the top pointer or index is at its initial value (indicating an empty stack). This operation does not depend on the size of the stack, making its time complexity (O(1)), or constant time.

Space Complexity

The space complexity of a stack is determined by the amount of memory it uses in relation to the number of elements it contains. The analysis here assumes the stack is implemented using either an array or a linked list.

- **Array-based Implementation**: In an array-based implementation, the space complexity is (O(n)), where (n) is the capacity of the array. This is because a contiguous block of memory is allocated to hold the stack's elements, and the size of this block is fixed upon the initialization of the stack. It's worth noting that the allocated memory might not be fully utilized if the stack contains fewer elements than its capacity.
- **Linked List Implementation**: For a linked list implementation, the space complexity is also (O(n)), where (n) is the number of elements in the stack. Each element or node in a linked list contains the data and the pointer(s) (next and possibly prev, in the case of a doubly linked list). Unlike the array-based implementation, there's no pre-allocated, unused memory; memory is allocated dynamically for each element added to the stack. However, the overhead of the additional pointers for each node must be considered.

Summary

Stack operations generally boast a time complexity of (O(1)), making stacks an exceptionally efficient data structure for scenarios where the primary operations involve adding or removing elements from the top of the stack. The space complexity of (O(n)) indicates that the memory usage grows linearly with the number of elements in the stack, which is typical for data structures that store collections of items. The choice between an array-based or linked list-based stack implementation can be based on specific requirements related to space efficiency, the need for dynamic resizing, and the overhead associated with each approach.

Reflection MCQs

1. What does LIFO stand for in the context of a stack?

    - A) Last In, First Out
    - B) Last In, Final Out
    - C) Least In, First Out
    - D) **A) Last In, First Out**
2. Which operation adds an element to the top of a stack?

    - A) enqueue
    - B) **push**
    - C) insert
    - D) add

1. What operation removes the top element from a stack?

    - A) remove
    - B) dequeue
    - C) **pop**
    - D) extract
2. Which operation returns the top element without removing it from the stack?

    - A) top
    - B) peek
    - C) get
    - D) **B) peek (also A) top is acceptable in some contexts)**

1. The time complexity of push, pop, and peek operations in a stack is:

    - A) O(n)
    - B) O(log n)
    - C) **O(1)**
    - D) O(n^2)
2. Which of the following is NOT a direct application of stacks?

    - A) Evaluating postfix expressions
    - B) Undo mechanisms in text editors
    - C) **Queue management**
    - D) Function calls management

1. In a stack implemented using an array, what happens if a push operation is attempted when the array is full?

    - A) The stack automatically resizes
    - B) **An overflow condition occurs**
    - C) The bottom element is removed
    - D) The push operation is ignored
2. Which data structure can be used to implement a stack?

    - A) Array
    - B) Linked list
    - C) Both A and B
    - D) **C) Both A and B**

1. An attempt to pop an element from an empty stack is known as:

    - A) Overflow
    - B) **Underflow**
    - C) Null operation
    - D) Empty pop
2. Which of the following scenarios is an example of a stack application?

    - A) Breadth-first search
    - B) **Managing browser history**
    - C) Round-robin scheduling
    - D) Packet routing

1. Stacks are ideal for:

    - A) First In, First Out (FIFO) operations
    - B) **Last In, First Out (LIFO) operations**
    - C) Priority-based operations
    - D) Non-sequential access
2. How can you check if a stack is empty?

    - A) By checking if top is -1
    - B) By checking if top is null
    - C) **Both A and B (depending on the implementation)**
    - D) By using the size method

1. The operation to insert an element in a stack is called:

    - A) enqueue
    - B) append
    - C) **push**
    - D) insert
2. What does it mean to "peek" at a stack?

    - A) To add an element
    - B) **To view the top element without removing it**
    - C) To remove an element
    - D) To clear the stack

1. In stack terminology, what does "overflow" refer to?

    - A) Deleting from an empty stack
    - B) **Trying to add an element to a full stack**
    - C) Accessing an element that doesn't exist
    - D) A failed search operation
2. Which structure is more suitable for implementing an undo feature in a text editor?

    - A) Queue
    - B) **Stack**
    - C) Array
    - D) Tree

1. What is the space complexity of a stack implemented as a linked list, where n is the number of elements?

    - A) O(1)
    - B) O(log n)
    - C) **O(n)**
    - D) O(n^2)
2. Recursion uses the call stack to:

    - A) Store variables
    - B) Keep track of function calls
    - C) **Both A and B**
    - D) Optimize memory usage

1. In postfix expression evaluation using a stack, what happens when an operator is encountered?
    - A) The operator is pushed onto the stack
    - B) The stack is cleared
    - C) **The top two elements are popped, the operation is applied, and the result is pushed back**
    - D) A new stack is created

Coding Exercise: Implementing a Browser History Feature Using a Stack

In this exercise, you'll create a simplified version of a web browser's history feature using a stack data structure. The application will allow users to visit websites (push URLs onto the stack), go back to the previous page (pop a URL from the stack), and display the current page. To focus on understanding stacks and their applications, you will implement the stack from scratch without using any pre-existing stack templates or libraries.

Step 1: Define the Stack Structure

First, define a basic stack structure using a linked list to store URLs (as strings). Each node in the linked list will represent a web page that has been visited.

```
struct Page {
    string url;
    Page* next;
    Page(string url): url(url), next(nullptr) {}
};
```

Create a class `BrowserHistory` that contains the stack operations and a pointer to the top of the stack.

```
class BrowserHistory {
    Page* top;

public:
    BrowserHistory(): top(nullptr) {}
    ~BrowserHistory();
    void visit(const string& url);
    string goBack();
    string currentPage();
};
```

Step 2: Implement Stack Operations

Implement the methods for the `BrowserHistory` class to perform stack operations.

Visit a Page

Add a new page to the history (push operation).

```
void BrowserHistory::visit(const string& url) {
    Page* newPage = new Page(url);
    newPage->next = top;
    top = newPage;
}
```

Go Back

Go back to the previous page (pop operation) and return the URL of the now-current page.

```
string BrowserHistory::goBack() {
    if (top == nullptr) return "No history available.";
    Page* temp = top;
    top = top->next;
    string url = temp->url;
    delete temp;
    return top ? top->url : "No history available.";
}
```

Current Page

Return the URL of the current page without removing it from the history.

```
string BrowserHistory::currentPage() {
    return top ? top->url : "No current page.";
}
```

Step 3: Destructor

Ensure to free all allocated memory when the `BrowserHistory` instance is destroyed.

```
BrowserHistory::~BrowserHistory() {
    while (top != nullptr) {
        Page* temp = top;
        top = top->next;
        delete temp;
    }
}
```

Step 4: Main Function

Demonstrate the functionality of your browser history feature with a simple main function.

```
int main() {
    BrowserHistory history;
    history.visit("http://example.com");
    history.visit("http://example.com/about");
    cout << "Current Page: " << history.currentPage() << endl; // http://example.co
m/about
    cout << "Going back to: " << history.goBack() << endl; // http://example.com
    cout << "Current Page: " << history.currentPage() << endl; // http://example.co
m
    return 0;
}
```

Make sure that you have:

- **completed all the concepts**
- **ran all code examples yourself**
- **tried changing little things in the code to see the impact**
- **implemented the required programs**

Please feel free to share your problems to ensure that your weekly tasks are completed and you are not staying behind