Stack

A diagram of a push button

Description automatically generated

**What is** Stack **Data Structure?**

A stack is a fundamental data structure in computer science that follows the Last-In-First-Out (LIFO) principle. It is an abstract data type that represents a collection of elements, where elements can be inserted and removed only from one end, called the "top" of the stack.

The stack has two main operations:

1. Push: This operation adds an element to the top of the stack. The new element becomes the top, and any existing elements are pushed down one position.
2. Pop: This operation removes the top element from the stack. The element that was below the top becomes the new top, and the size of the stack decreases by one.

In addition to these two primary operations, stacks often provide other auxiliary operations, such as:

* Peek or Top: This operation returns the value of the top element without removing it from the stack.
* IsEmpty: This operation checks whether the stack is empty or not.
* Size: This operation returns the number of elements currently in the stack.

Stacks can be implemented using various data structures, such as arrays or linked lists. The choice of implementation depends on the programming language and the specific requirements of the application.

Stacks are widely used in programming and computer science, and they have numerous applications, including:

* Function call stack: Stacks are used to manage the order of function calls and their return addresses.
* Expression evaluation: Stacks are used to evaluate arithmetic expressions, parse and evaluate postfix expressions, and implement algorithms like infix to postfix conversion.
* Undo/Redo functionality: Stacks can be used to implement undo and redo operations in text editors or other applications.
* Backtracking algorithms: Stacks can be used to store and manage the state of a search or traversal algorithm, enabling backtracking when needed.
* Memory management: Stacks are used in memory allocation and deallocation, such as managing the call stack and local variables in programming languages.

Overall, the stack data structure provides a simple and efficient way to manage data in a Last-In-First-Out manner, making it a valuable tool in many areas of computer science and programming.

* **Why there is a need for Stack Data Structure?**

There are several reasons why the stack data structure is needed in computer science and programming:

1. Managing function calls: Stacks are essential for managing function calls in programming languages. When a function is called, its execution context, including the return address and local variables, is pushed onto the stack. This allows the program to keep track of the current function and resume execution from the correct point after the function returns.
2. Expression evaluation: Stacks are used to evaluate arithmetic expressions, parse and evaluate postfix expressions, and implement algorithms like infix to postfix conversion. Stacks provide an efficient way to handle operators and operands, ensuring the correct order of operations.
3. Memory management: Stacks play a crucial role in memory management, particularly in managing the call stack and local variables. Each function call adds a new frame to the call stack, and local variables are allocated within that frame. When a function returns, its frame is removed from the stack, freeing up the associated memory.
4. Undo/Redo functionality: Stacks are commonly used to implement undo and redo operations in applications like text editors or graphic design software. Each action performed by the user is pushed onto the stack, allowing for easy reversal of actions or redoing previously undone actions.
5. Backtracking algorithms: Many search and traversal algorithms use stacks to manage the state and facilitate backtracking. Stacks allow the algorithm to remember previous states and explore alternative paths when needed. Backtracking algorithms are used in areas such as graph traversal, puzzle solving, and constraint satisfaction problems.
6. Recursive algorithms: Recursive algorithms heavily rely on stacks to manage the recursive function calls. Each recursive call pushes a new frame onto the stack, allowing the algorithm to process subproblems and eventually return the results.
7. Parsing and language processing: Stacks are often used in parsing and language processing tasks, such as syntax analysis and building abstract syntax trees. Stacks help track the structure of the program and ensure correct parsing and interpretation.

In summary, the stack data structure is needed because it provides a simple and efficient way to manage data in a Last-In-First-Out manner. Its properties and operations make it invaluable in various domains, including function calls, expression evaluation, memory management, undo/redo functionality, backtracking algorithms, recursive algorithms, and parsing and language processing.

* **Types of Stack Data Structure**

**1. Array Stack:**

* Implements a stack using a fixed-size array.
* Simple and efficient for small or fixed-size stacks.
* Can encounter overflow if the stack exceeds the array's capacity.

**2. Linked Stack:**

* Uses a linked list to implement a stack.
* Flexible, as it can grow dynamically without any fixed size limit.
* Might have slightly slower operations compared to array stacks due to pointer overhead.

**3. Heap Stack:**

* Uses memory allocated on the heap to create a stack.
* Can grow dynamically, but memory management is more complex.

There are several types of stack data structures, each with its own characteristics and variations. Here are some commonly used types:

1. **Array-based stack:** An array-based stack uses an underlying array to store the elements. The top of the stack is represented by an index pointing to the last inserted element. This type of stack has a fixed size determined by the size of the array. When the stack is full, it cannot accommodate any more elements unless it is resized.
2. **Dynamic array-based stack:** A dynamic array-based stack is an extension of the array-based stack. It starts with a small initial array size and dynamically resizes the array when needed. When the stack is full, a new larger array is created, and the elements from the old array are copied to the new array. This allows the stack to accommodate a variable number of elements.
3. **Linked list-based stack:** A linked list-based stack uses a linked list to store the elements. Each element in the stack is represented by a node that contains the value and a reference to the next node. The top of the stack is represented by the head of the linked list. This type of stack can dynamically grow or shrink as elements are inserted or removed.
4. **Bounded stack:** A bounded stack is a stack with a maximum capacity limit. It can only hold a specific number of elements, and attempting to push more elements when the stack is full results in an overflow condition. Bounded stacks are useful in situations where memory or resource constraints need to be enforced.

These are some of the common types of stack data structures. Each type has its advantages and use cases, and the choice of which type to use depends on the specific requirements of the application or problem at hand.

* **Internal Implementation of X Data Structure:   
  Look at the code…**
* Operations on Stack Data Structure with Asymptotic Complexity and Complexity Analysis:

The stack data structure supports several fundamental operations. Here are the most common operations performed on a stack, along with their asymptotic complexities and complexity analysis:

1. Push operation:

- Description: Adds an element to the top of the stack.

- Complexity: O(1) - constant time complexity.

- Complexity analysis: The push operation simply involves adding an element to the top of the stack. It does not depend on the size of the stack, so the time complexity remains constant.

2. Pop operation:

- Description: Removes the top element from the stack.

- Complexity: O(1) - constant time complexity.

- Complexity analysis: The pop operation involves removing the top element from the stack. Similar to the push operation, it does not depend on the size of the stack, resulting in constant time complexity.

3. Peek or Top operation:

- Description: Returns the value of the top element without removing it.

- Complexity: O(1) - constant time complexity.

- Complexity analysis: The peek operation only involves accessing the top element of the stack, which can be done in constant time.

4. IsEmpty operation:

- Description: Checks whether the stack is empty or not.

- Complexity: O(1) - constant time complexity.

- Complexity analysis: The isEmpty operation simply checks if the stack has any elements. It does not require iterating through the stack or performing any complex computations, resulting in constant time complexity.

5. Size operation:

- Description: Returns the number of elements in the stack.

- Complexity: O(1) - constant time complexity.

- Complexity analysis: The size operation typically involves maintaining a count of the elements in the stack. Incrementing or decrementing this count when elements are pushed or popped can be done in constant time.

It's important to note that the above complexity analysis assumes that the underlying data structure used to implement the stack (e.g., array-based or linked list-based) has efficient operations for adding and removing elements from one end.

In summary, the core operations on a stack—push, pop, peek, isEmpty, and size—have a constant time complexity of O(1). This makes stacks highly efficient for managing data in a Last-In-First-Out fashion.

* **Implementation of Operations on X Data Structure:**

**Look at the code ….**

* **Properties of Stack Data Structure:**
* **The stack data structure has several properties that define its behavior and characteristics. Here are the key properties of a stack:**
* **1. Last-In-First-Out (LIFO) order: The primary property of a stack is that it follows the Last-In-First-Out order. This means that the last element inserted into the stack is the first one to be removed. Elements are added and removed from only one end, called the top of the stack.**
* **2. Single access point: A stack has a single access point, which is the top of the stack. All insertions and removals occur at this end. Other elements in the stack cannot be directly accessed or modified without first removing the elements above them.**
* **3. No random access: Unlike arrays or lists, stacks do not support random access to elements. You can only access the top element of the stack. To access other elements, you need to remove the elements on top until you reach the desired element.**
* **4. Fixed or dynamic size: Depending on the implementation, a stack can have a fixed or dynamic size. In a fixed-size stack, there is a pre-defined maximum capacity, and once the stack is full, it cannot accommodate additional elements. In a dynamic-size stack, the stack can grow or shrink dynamically as elements are added or removed.**
* **5. Efficient insertion and removal: Stacks provide efficient insertion and removal operations. Adding an element to the top of the stack (push operation) and removing the top element (pop operation) both have a constant time complexity, typically O(1), making them very efficient.**
* **6. Limited functionality: Stacks have a limited set of operations compared to other data structures. The primary operations are push (add an element to the top), pop (remove the top element), peek (access the top element without removing it), isEmpty (check if the stack is empty), and size (get the number of elements in the stack).**
* **7. Recursive structure: Stacks naturally lend themselves to recursive structures. Recursive algorithms and function calls often use stacks to manage the recursive calls and track the execution context.**
* **8. Stack overflow and underflow: When pushing elements onto a stack, there is a possibility of a stack overflow if the stack is full and cannot accommodate more elements. On the other hand, when popping elements from the stack, there is a possibility of a stack underflow if the stack is empty and there are no elements to remove.**
* **These properties make the stack data structure well-suited for managing data in a Last-In-First-Out manner and enable efficient operations for adding and removing elements. Stacks find applications in various domains, including function calls, expression evaluation, memory management, and backtracking algorithms.Applications of X Data Structure:**

**The LinkedList data structure finds applications in various scenarios where dynamic size, efficient insertion and deletion operations, and sequential access are important. Some common applications of LinkedLists include:**

**1. Implementing Stacks and Queues: LinkedLists provide an excellent foundation for implementing stack and queue data structures. Stacks can be implemented using a singly linked list, where elements are added and removed from the head (top) in constant time. Queues, on the other hand, can be implemented using a doubly linked list, where elements are added at the tail and removed from the head in constant time.**

**2. Undo/Redo Functionality: LinkedLists are commonly used to implement the undo and redo functionality in applications where users can perform a series of actions and then revert or redo them. Each action is stored as a node in the LinkedList, allowing efficient undo and redo operations by traversing the list back and forth.**

**3. Symbol Tables: Symbol tables, or associative arrays, store key-value pairs, and LinkedLists can be used to implement them efficiently. Each node in the list represents a key-value pair, and sequential access enables efficient searching, insertion, and deletion by key.**

**4. Polynomial Manipulation: LinkedLists are often used to represent polynomials for mathematical computations. Each node in the list represents a term of the polynomial, containing the coefficient and the exponent. LinkedLists allow efficient addition, multiplication, and evaluation of polynomials.**

**5. Graph Algorithms: LinkedLists are used in various graph algorithms, such as representing adjacency lists. In an adjacency list, each vertex of a graph is associated with a LinkedList that contains the adjacent vertices. This representation allows efficient traversal of graph edges and is commonly used in algorithms like breadth-first search (BFS) and depth-first search (DFS).**

**6. Music and Video Playlists: LinkedLists are suitable for implementing playlists in music and video players. Each node in the list represents a song or video, and the sequential access allows for playing the items in the desired order. Insertion and deletion operations enable adding or removing items from the playlist dynamically.**

**7. Hash Table Collision Resolution: In hash table implementations, when two or more elements map to the same hash value (collision), LinkedLists can be used as a collision resolution strategy. Each bucket in the hash table contains a LinkedList, and elements with the same hash value are stored in the corresponding list. This approach allows for efficient insertion and retrieval of elements.**

**8. File System Implementations: LinkedLists can be used to implement file systems' directory structures. Each node represents a file or directory, and the references link the nodes together, forming the hierarchical structure. Sequential access enables efficient traversal of directories and listing of files.**

**These are just a few examples of the applications of LinkedLists. The flexibility, ease of modification, and efficient insertion and deletion operations make LinkedLists a valuable choice in various data manipulation scenarios.**

* **Advantages of Stack Data Structure:**

**The stack data structure offers several advantages that make it useful and applicable in various scenarios. Here are the key advantages of using a stack:**

**1. Simple and intuitive: The stack data structure has a simple and intuitive interface, with only a few fundamental operations such as push, pop, peek, isEmpty, and size. This simplicity makes it easy to understand, implement, and use.**

**2. Efficient operations: Stacks provide efficient operations for adding and removing elements. Both the push and pop operations have a constant time complexity, typically O(1). This efficiency allows for fast insertion and removal of elements, making stacks suitable for time-sensitive applications.**

**3. Memory management: Stacks play a crucial role in managing memory in programming languages. The call stack, implemented as a stack data structure, is responsible for managing function calls and local variables. Each function call pushes a new frame onto the stack, and when a function returns, its frame is removed, freeing up the associated memory. This automatic memory management simplifies the programming process and reduces the risk of memory leaks.**

**4. Function call management: Stacks are essential for managing function calls in programming languages. They allow for proper execution context management, including storing return addresses, local variables, and function parameters. This enables the program to keep track of the current function and resume execution from the correct point after a function returns.**

**5. Expression evaluation: Stacks are widely used in expression evaluation, parsing, and conversion algorithms. They provide a natural and efficient way to handle operators and operands, ensuring the correct order of operations. Stacks are particularly useful in evaluating arithmetic expressions, converting between different expression notations (e.g., infix to postfix), and parsing complex expressions.**

**6. Backtracking algorithms: Many backtracking algorithms, such as depth-first search, rely on stacks to manage the state and facilitate backtracking. Stacks allow the algorithm to remember previous states, explore alternative paths, and easily backtrack when needed. This makes stacks a valuable tool in solving problems that require exhaustive search or exploration of solution spaces.**

**7. Undo/Redo functionality: Stacks are commonly used to implement undo and redo functionality in applications. Each action performed by the user is pushed onto the stack, allowing for easy reversal of actions or redoing previously undone actions. Stacks provide a natural way to manage the sequence of user actions and maintain the state history.**

**8. Recursive algorithms: Stacks are closely tied to recursive algorithms. Recursive function calls rely on stacks to manage the recursive context, keeping track of function calls and their local variables. Stacks provide an elegant way to handle recursion, making it easier to implement and understand recursive algorithms.**

**These advantages highlight the versatility and usefulness of the stack data structure in various domains, including memory management, function call management, expression evaluation, backtracking algorithms, and implementing undo/redo functionality. The simplicity, efficiency, and intuitive nature of stacks make them a valuable tool for solving a wide range of problems.**

* **Disadvantages of Stack Data Structure**[:](https://docs.google.com/document/d/117SPrcFF80B2fxxYIisA5cTEdbVoxfgHV1gcGiOCn2w/edit" \l "heading=h.mtw228jwq389)

**While the stack data structure offers many advantages, it also has some limitations and disadvantages. Here are the key disadvantages of using a stack:**

**1. Limited functionality: Stacks have a limited set of operations compared to other data structures. They primarily support push, pop, peek, isEmpty, and size operations. This limited functionality can be a drawback when more complex operations, such as searching for a specific element or accessing elements at arbitrary positions, are required.**

**2. Lack of random access: Stacks do not support random access to elements. You can only access the top element of the stack, and to access other elements, you need to remove elements from the top until you reach the desired element. This lack of random access can be inefficient and impractical for certain applications that require frequent access to elements at arbitrary positions.**

**3. Dynamic memory management: In some implementations, stacks require dynamic memory management. When using a linked list-based stack, each element is allocated individually, potentially leading to memory fragmentation. Moreover, dynamic resizing of an array-based stack can be costly in terms of time and memory, as it involves creating a new larger array and copying elements from the old array.**

**4. Limited capacity: Stack implementations that have a fixed size, such as array-based stacks, have a limited capacity. Once the stack is full, it cannot accommodate additional elements unless it is resized or emptied. This limitation can be problematic if the number of elements exceeds the predetermined capacity.**

**5. Stack overflow and underflow: Stack overflow occurs when there is an attempt to push elements onto a full stack, leading to an error condition. Stack underflow occurs when there is an attempt to pop an element from an empty stack, again resulting in an error condition. These scenarios need to be carefully handled to prevent program crashes or unexpected behavior.**

**6. Non-reversible order: Since stacks follow the Last-In-First-Out (LIFO) order, reversing the order of elements in a stack is not possible using stack operations alone. Reversing the order would require additional data structures or algorithms, which can add complexity and overhead to the solution.**

**7. Space inefficiency: In some scenarios, stacks may not be the most space-efficient data structure. For example, if there are many elements in the stack, but only a few of them are needed at any given time, the stack may occupy more memory than necessary. This can be a concern in memory-constrained environments.**

**It's important to consider these disadvantages when deciding whether to use a stack data structure. While stacks excel in certain situations, such as managing function calls, expression evaluation, and backtracking algorithms, they may not be the optimal choice for every scenario.**

Diff between Stack and linkedList:

Stack and LinkedList are both data structures commonly used in computer science and programming. While there are some similarities between them, there are also significant differences. Here's a comparison between Stack and LinkedList:

Stack:

- Stack is an abstract data type that follows the Last-In-First-Out (LIFO) principle.

- It has a single access point, known as the top of the stack, where elements are added and removed.

- Only the top element of the stack can be accessed or modified directly.

- Stack operations include push (add an element to the top), pop (remove the top element), peek (access the top element without removing it), isEmpty (check if the stack is empty), and size (get the number of elements in the stack).

- The underlying implementation of a stack can vary, but it is often implemented using an array or a linked list.

- Stack operations, such as push and pop, have a constant time complexity of O(1) since they involve adding or removing elements from one end.

LinkedList:

- LinkedList is a data structure consisting of nodes, where each node contains a value and a reference to the next node.

- It allows efficient insertion and removal of elements at any position in the list.

- Unlike a stack, a linked list does not follow a specific order like LIFO or FIFO.

- LinkedList operations include insertion (add an element at a specific position), deletion (remove an element from a specific position), search (find an element), and traversal (visit each element in the list).

- LinkedList operations have a time complexity of O(n) for average and worst-case scenarios, where n is the number of elements in the list. This is because finding the position for insertion or deletion may require traversing the list from the beginning or a specific index.

- LinkedLists can be singly linked (each node points to the next node) or doubly linked (each node has references to both the next and previous nodes).

In summary, a Stack is a specific type of data structure that follows the LIFO principle and provides a limited set of operations for adding and removing elements from the top only. On the other hand, a LinkedList is a more general-purpose data structure that allows insertion and removal at any position and supports traversal and searching. LinkedList operations have a time complexity of O(n), while Stack operations have a constant time complexity of O(1).

Diff between arrayList and Stack:

ArrayList and Stack are both data structures used in computer science and programming, but they have different characteristics and usage scenarios. Here's a comparison between an ArrayList and a Stack:

ArrayList:

- ArrayList is a dynamic array-based data structure that can grow or shrink in size as elements are added or removed.

- It provides random access to elements, meaning you can access any element in the list directly using its index.

- Elements in an ArrayList are stored in contiguous memory locations, allowing for efficient memory access.

- ArrayList allows duplicates and maintains the order of elements based on their insertion.

- It provides a variety of operations, including adding elements at the end (append), inserting elements at a specific position, removing elements by index or value, searching for elements, and iterating over the elements.

- The time complexity for accessing an element by index is O(1), while adding or removing elements at the end has an average time complexity of O(1). However, inserting or removing elements at arbitrary positions has a time complexity of O(n) since it may require shifting subsequent elements.

Stack:

- Stack is an abstract data type that follows the Last-In-First-Out (LIFO) principle.

- It has a single access point, known as the top of the stack, where elements are added and removed.

- Only the top element of the stack can be accessed or modified directly.

- Stack operations include push (add an element to the top), pop (remove the top element), peek (access the top element without removing it), isEmpty (check if the stack is empty), and size (get the number of elements in the stack).

- Stacks are typically implemented using an array or a linked list, but the underlying implementation is not visible to the user.

- The time complexity for push and pop operations in a stack is O(1), as they involve adding or removing elements from one end only.

Differences:

1. Data Structure Type:

- ArrayList: It is a dynamic array-based data structure.

- Stack: It is an abstract data type that can be implemented using an array or a linked list.

2. Access Pattern:

- ArrayList: Provides random access to elements using their index.

- Stack: Provides access to only the top element and follows the LIFO order.

3. Ordering:

- ArrayList: Maintains the order of elements based on their insertion.

- Stack: Follows the Last-In-First-Out (LIFO) order, meaning the last element inserted is the first one to be removed.

4. Operations:

- ArrayList: Supports a variety of operations, including adding, inserting, removing, searching, and iterating over elements.

- Stack: Provides a limited set of operations, such as push, pop, peek, isEmpty, and size.

5. Time Complexity:

- ArrayList: Accessing an element by index is O(1), while inserting or removing elements at arbitrary positions is O(n).

- Stack: Push and pop operations have a constant time complexity of O(1).

In summary, an ArrayList is a dynamic array-based data structure that provides random access to elements and supports various operations. On the other hand, a Stack is an abstract data type that follows the LIFO principle, allowing access to only the top element and providing a limited set of operations. ArrayList is more versatile and suitable for scenarios where random access and flexible operations are required, while Stack is specialized for LIFO-based applications.

ArrayStack VS LinkedStack

ArrayStack and LinkedStack are two common implementations of the stack data structure, but they differ in how they store and access elements.

1. ArrayStack:

- ArrayStack uses a fixed-size array to store elements.

- It allocates memory for the array during initialization.

- The elements are stored in contiguous memory locations, allowing for efficient random access.

- Adding or removing elements from the top of the stack (push and pop operations) has a time complexity of O(1).

- However, if the array becomes full, and a new element needs to be added, the array must be resized, which may require creating a new array and copying elements over. This operation has a time complexity of O(n) as it involves copying all the elements.

- ArrayStack is a good choice when the maximum number of elements is known in advance, or when there is a limit on the stack size.

2. LinkedStack:

- LinkedStack uses a linked list structure to store elements.

- Each element (node) in the linked list contains a reference to the next element in the stack.

- LinkedStack doesn't have a fixed size and can dynamically grow or shrink as needed.

- Adding or removing elements from the top of the stack has a time complexity of O(1) since it only involves updating the references.

- LinkedStack doesn't require resizing operations like ArrayStack, so it avoids the overhead associated with resizing.

- However, LinkedStack uses extra memory to store the references to the next elements.

- LinkedStack is a good choice when the maximum number of elements is unknown or when the stack size needs to change frequently.

In summary, ArrayStack provides better random access performance and is more memory-efficient if the maximum size is known in advance. On the other hand, LinkedStack provides flexibility in size and avoids the resizing overhead. The choice between them depends on the specific requirements of your application.

