Stack Data Structure: Comprehensive Summary

1. Introduction to Stacks

A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle. Elements are added (pushed) and removed (popped) only from one end, called the **top** of the stack. Stacks are used in many applications such as function call management, expression evaluation, backtracking algorithms, and browser history navigation[1].

Key Characteristics

- **LIFO Access:** The most recently added element is the first to be removed
- **Top Pointer/Reference:** Tracks the current top element
- **Homogeneous Data:** All elements are of the same type
- **Time Complexity:** Push, pop, and peek operations are O(1)

---

2. Stack Implementation Using Arrays

2.1 Array-Based Stack Structure

Attributes:

- **arr[]** - A fixed-size array to store stack elements
- **top** - An integer variable that points to the index of the top element
  - **Initial value:** top = -1 (when stack is empty)
  - **After first push:** top = 0
  - **Increments on push:** top++
  - **Decrements on pop:** top--
- **capacity** - The maximum size of the array (fixed at creation time)

2.2 Array Implementation Code (C++)

```
#include <iostream>
using namespace std;

class StackArray {
private:
int* arr;
int top;
int capacity;

public:
StackArray(int size) {
capacity = size;
arr = new int[capacity];
```

top = -1;
}

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

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

void push(int value) {
    if (isFull()) {
        cout << "Stack Overflow!\n";
        return;
    }
    arr[++top] = value;
    cout << value << " pushed to stack\n";
}

int pop() {
    if (isEmpty()) {
        cout << "Stack Underflow!\n";
        return -1;
    }
    return arr[top--];
}

int peek() {
    if (isEmpty()) {
        cout << "Stack is empty!\n";
        return -1;
    }
    return arr[top];
}

void print() {
    if (isEmpty()) {
```

```
        cout << "Stack is empty!\n";
        return;
    }
    cout << "Stack elements: ";
    for (int i = 0; i <= top; i++) {
        cout << arr[i] << " ";
    }
    cout << "\n";
}

~StackArray() {
    delete[] arr;
}
```

};

```
// Example usage
int main() {
StackArray stack(5);
stack.push(10);
stack.push(20);
stack.push(30);
stack.print(); // Output: 10 20 30
cout << "Top: " << stack.peek() << "\n"; // Output: 30
cout << "Popped: " << stack.pop() << "\n"; // Output: 30
stack.print(); // Output: 10 20
return 0;
}
```

2.3 Array Implementation Advantages

- **Cache-Friendly:** Array elements are contiguous in memory, leading to better CPU cache performance[1]
- **Fast Access:** Direct index-based access to elements (O(1) average case)
- **Memory Efficient:** No extra memory overhead for pointers
- **Simple Implementation:** Straightforward logic with minimal code complexity

2.4 Array Implementation Disadvantages

- **Fixed Size:** Capacity is predetermined; cannot grow dynamically[1]
- **Memory Waste:** If stack size is unknown, declaring a large array wastes memory
- **Overflow Risk:** Stack overflow occurs when trying to push beyond capacity
- **Costly Resizing:** If more space is needed, array must be resized (usually doubled), which is O(n) operation

---

3. Stack Implementation Using Linked Lists

3.1 Linked List-Based Stack Structure

Attributes:

- **Node Structure:**
  - **data** - The value stored in the node
  - **next** - A pointer/reference to the next node in the stack
- **top** (or head) - A pointer to the topmost node
  - **Initial value:** top = NULL/nullptr (when stack is empty)
  - **Points to:** The most recently added node
  - **Updates on push:** New node becomes the new top
  - **Updates on pop:** top pointer shifts to the previous node

3.2 Linked List Implementation Code (C++)

#include <iostream>
using namespace std;

class StackLinkedList {
private:
struct Node {
int data;
Node* next;
Node(int value) : data(value), next(nullptr) {}
};

```
Node* top;
```

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

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

void push(int value) {
    Node* newNode = new Node(value);
    newNode->next = top;
    top = newNode;
    cout << value << " pushed to stack\n";
}

int pop() {
```

```
    if (isEmpty()) {
        cout << "Stack Underflow!\n";
        return -1;
    }
    Node* temp = top;
    int value = temp->data;
    top = top->next;
    delete temp;
    return value;
}

int peek() {
    if (isEmpty()) {
        cout << "Stack is empty!\n";
        return -1;
    }
    return top->data;
}

void print() {
    if (isEmpty()) {
        cout << "Stack is empty!\n";
        return;
    }
    cout << "Stack elements: ";
    Node* temp = top;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << "\n";
}

~StackLinkedList() {
    while (!isEmpty()) {
        pop();
```

```
    }
  }
```

};

// Example usage
int main() {
StackLinkedList stack;
stack.push(10);
stack.push(20);
stack.push(30);
stack.print(); // Output: 30 20 10
cout << "Top: " << stack.peek() << "\n"; // Output: 30
cout << "Popped: " << stack.pop() << "\n"; // Output: 30
stack.print(); // Output: 20 10
return 0;
}

3.3 Linked List Implementation Advantages

- **Dynamic Size:** Stack grows and shrinks as needed; no fixed capacity[1]
- **No Overflow:** As long as memory is available, stack can expand indefinitely
- **Memory Efficient:** Memory is allocated only when needed
- **Flexible:** Scales well for unpredictable data sizes

3.4 Linked List Implementation Disadvantages

- **Memory Overhead:** Extra memory required for storing pointers (typically 4-8 bytes per node)[1]
- **Cache Unfriendly:** Nodes are scattered in memory; poor cache locality
- **Slower Access:** Accessing elements requires traversing from the top
- **More Complex Code:** Requires manual memory management and pointer handling

---

4. Comparison: Array vs Linked List Stack

Attribute	Array-Based	Linked List-Based
Memory Allocation	Fixed at creation	Dynamic (as needed)
Size Limitation	Fixed capacity	Unlimited (limited by available memory)
Push/Pop Time	O(1) amortized	O(1)
Peek Time	O(1)	O(1)
Memory Overhead	None for data	Extra for pointers per node
Cache Performance	Cache-friendly	Cache-unfriendly
Overflow Risk	Yes (if full)	No
Resizing Cost	O(n) when doubled	No resizing needed
Underflow Handling	Checked at top = -1	Checked at top = NULL
Best Use Case	Known fixed size	Unknown/variable size
Implementation Complexity	Simple	Moderate
Memory Waste	Possible if unused	Minimal (one pointer per element)

Table 1: Detailed Comparison of Array vs Linked List Stack Implementations

---

5. Attribute Differences in Detail

5.1 Top Pointer/Index

Array Version:

- Stores an **integer index** (0 to capacity-1)
- Top = -1 means empty
- Top = 4 means elements from index 0 to 4 are in use
- Direct calculation of used space: (top + 1) elements

Linked List Version:

- Stores a **pointer/reference** to a Node
- Top = NULL means empty
- Top points to the actual node object
- Must traverse to find stack size

5.2 Capacity Management

Array Version:

- **Fixed capacity** set during initialization
- Cannot exceed declared size
- Checking: isFull() returns `(top == capacity - 1)`
- Overflow must be handled explicitly

Linked List Version:

- **No predefined capacity**
- Grows with each push (within system memory limits)
- No isFull() method needed
- Overflow only occurs if system runs out of memory

5.3 Memory Overhead

Array Version:
Memory used = capacity × sizeof(int)
Example: Stack of 100 integers = 100 × 4 bytes = 400 bytes

Linked List Version:
Memory used = (number of elements) × (sizeof(int) + sizeof(pointer))
Example: Stack with 100 elements = 100 × (4 + 8) = 1200 bytes
Note: Extra 800 bytes for pointers, but no wasted space if stack is small

5.4 Empty State Representation

Array Version:

- Empty condition: `top == -1`
- Simple integer comparison

Linked List Version:

- Empty condition: `top == nullptr` (or NULL)
- Pointer null check

5.5 Deletion and Cleanup

Array Version:

- Single `delete[]` for the entire array
- Destructors called for array elements automatically

Linked List Version:

- Must delete each node individually in the destructor
- Requires traversal from top to bottom
- Risk of memory leaks if not handled carefully

---

6. Real-World Applications

1. **Function Call Stack:** Manages recursive function calls and local variables[1]
2. **Expression Evaluation:** Evaluates postfix expressions and checks balanced parentheses
3. **Browser History:** Back button in web browsers stores visited pages
4. **Undo/Redo Operations:** Text editors maintain stacks of changes
5. **Depth-First Search (DFS):** Graph traversal algorithms use stacks
6. **Tower of Hanoi:** Classic recursive algorithm that uses stacks

---

7. When to Use Which Implementation

Use **Array Implementation** when:

- Maximum stack size is known and fixed[1]
- Memory is limited and you want minimal overhead
- Cache performance is critical
- Implementation simplicity is important
- The application is performance-sensitive (e.g., embedded systems)

Use **Linked List Implementation** when:

- Stack size is unknown or highly variable[1]
- Memory allocation must be dynamic
- Frequent resizing of array would be costly
- Preventing overflow is critical
- Flexibility is more important than raw performance

---

8. Conclusion

Both array-based and linked list-based stack implementations provide O(1) push, pop, and peek operations[1]. The choice between them depends on:

- **Known vs. unknown capacity**
- **Fixed vs. variable size requirements**
- **Memory constraints vs. performance needs**
- **Implementation complexity tolerance**

For **fixed, predictable sizes**, arrays offer better cache performance and memory efficiency. For **dynamic, unpredictable sizes**, linked lists provide unlimited growth without overflow risk. In modern applications, both are widely used depending on the specific use case[1].

References

[1] GeeksforGeeks & Multiple Sources. (2024). Stack Implementation using Arrays and Linked Lists. https://www.geeksforgeeks.org/dsa/

[2] FullStackPrep. (2025, June 21). Array Stack vs Linked List Stack in .NET: Real-World Comparison. https://www.fullstackprep.dev/articles/dsa/array/array-vs-linkedlist-stack

[3] Fiveable. (2025, August 21). Implementation of Stacks and Queues using Arrays and Linked Lists. https://fiveable.me/data-structures/unit-3/implementation-stacks-queues-arrays-linked-lists/

[4] Stack Overflow. (2023, October 30). Which is more efficient: stacks using array or stacks using LinkedList? https://stackoverflow.com/questions/77393453/

[5] Purdue University. (n.d.). Stacks, Queues, and Linked Lists. https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap3.pdf