perplexity

What is a Data Structure?

A data structure is a specialized format for organizing, managing, and storing data in a computer so that it can be accessed and modified efficiently. It consists of a collection of data values, the relationships between them, and the operations or functions that can be applied to the data. Data structures are essential in programming and computer science because they determine how data is arranged in memory, which impacts the efficiency of algorithms that process that data.[1]

Data structures are broadly classified into two categories:[1]

- **Primitive Data Structures**: Basic types supported by programming languages, such as integers, floats, characters, and pointers.
- **Non-Primitive Data Structures**: More complex structures built from primitive types, further divided into linear (e.g., arrays, linked lists, stacks, queues) and non-linear (e.g., trees, graphs).

Examples include arrays for storing fixed-size collections of similar data, or graphs for representing networks with interconnected nodes.[1]

Areas Where Data Structures Are Applied Extensively

Data structures form the backbone of efficient software and systems design. They are used in various domains to handle data manipulation, storage, and retrieval. Here are some key areas:

- **Operating Systems**: For process scheduling (using queues and priority queues), memory management (linked lists and trees), and file systems (trees for directory structures).
- **Database Management Systems (DBMS)**: To index data (B-trees and hash tables for quick searches), manage records (arrays and linked lists), and optimize queries.
- **Compiler Design**: For symbol tables (hash tables), syntax trees (trees for parsing), and expression evaluation (stacks for postfix notation).
- **Graphics and Simulations**: Graphs for modeling scenes or networks, and queues for rendering pipelines or event simulations.
- **Artificial Intelligence and Machine Learning**: Trees and graphs for decision-making algorithms, heaps for priority-based searches, and arrays for dataset handling.
- **Networking**: Graphs for routing algorithms, queues for packet buffering, and hash tables for caching.
- **Web Development and Search Engines**: Tries for autocomplete features, hash maps for session management, and graphs for page ranking.

- **Numerical Analysis and Scientific Computing**: Matrices (2D arrays) for computations, and sparse data structures for efficient storage of large datasets.

These applications leverage data structures to optimize time and space complexity, making them crucial for real-world problem-solving in computer engineering. If you're working on a specific project, like your Java GUI development, stacks or queues could be useful for undo/redo features or task management.[1]

⁂

Classification of Data Structures

Data structures are generally classified into two main categories: primitive and non-primitive. This classification helps in understanding how data is organized and manipulated in programming and computer systems. Below, I'll break it down with explanations and examples.[4]

Primitive Data Structures

These are the fundamental data types directly supported by programming languages. They represent basic values and are not composed of other data types. Common examples include:[4]

- **Integer**: Used for whole numbers, e.g., `int` in C or Java for values like 5 or -10.
- **Float**: For decimal or floating-point numbers, e.g., `float` for values like 3.14.
- **Character**: For single letters or symbols, e.g., `char` for 'A' or '1'.
- **Pointer**: For storing memory addresses, e.g., in C, a pointer to an integer like `int* ptr`.

These are building blocks for more complex structures and are efficient for simple operations.

Non-Primitive Data Structures

These are more advanced structures created using primitive data types. They focus on organizing groups of homogeneous (same type) or heterogeneous (different types) data items. Non-primitive data structures are further divided into linear and non-linear types based on how elements are arranged.[4]

- **Linear Data Structures**: Elements are stored in a sequential order, allowing traversal in a single run. Examples include:[4]
  - **Array**: A fixed-size collection of similar elements stored in contiguous memory, e.g., `int marks = {90, 85, 78, 92, 88};` for storing student scores.
  - **Linked List**: A dynamic chain of nodes where each node points to the next, e.g., a list of names where insertion/deletion doesn't require shifting elements.
  - **Stack**: Follows Last-In-First-Out (LIFO) principle, e.g., a call stack in programming for function calls or undo operations in editors.
  - **Queue**: Follows First-In-First-Out (FIFO) principle, e.g., a print queue managing jobs in order.

- **Non-Linear Data Structures**: Elements are not in sequential order, allowing multiple paths for traversal. Examples include:[4]
  - **Tree**: A hierarchical structure with nodes connected via edges, e.g., a binary search tree for efficient searching and sorting data like file directories.
  - **Graph**: A network of nodes connected by edges, e.g., a social network graph where nodes are users and edges are friendships.

This classification is foundational in your computer engineering studies, especially for topics like stacks and queues from your attached slides. If you're implementing these in Java, linear structures like arrays are great for starting simple projects.

⁂

Definition of ADT

An Abstract Data Type (ADT) is a high-level description of a data structure that defines its behavior in terms of a set of operations and the possible values it can hold, without specifying the underlying implementation details. It acts as a mathematical model for data types, focusing on what the data structure does rather than how it is implemented in memory or code. This abstraction allows programmers to use the data structure independently of its internal workings, promoting modularity and reusability in software design. For example, an ADT specifies operations like insertion, deletion, and search, along with any constraints (e.g., time complexity or error conditions), but leaves the actual storage (e.g., array-based or linked list-based) to the implementer.[7]

In programming languages like C or Java, ADTs are often implemented using classes or structures, encapsulating data and methods. The concept is crucial in data structures courses, as seen in your attached materials, where it's listed under topics to cover.[7]

Stack ADT

A Stack ADT is a linear data structure that follows the Last-In-First-Out (LIFO) principle, meaning the last element added is the first one removed. It models real-world scenarios like a stack of plates, where you add or remove from the top. Stacks are useful for tasks requiring reversal, backtracking, or recursion, such as expression evaluation, undo mechanisms in editors, or function call management in programming.[7]

Key Operations:

- **Push(item)**: Adds an item to the top of the stack. If the stack is full, it may result in an overflow error.
- **Pop()**: Removes and returns the top item. If the stack is empty, it causes an underflow error.
- **Peek()** (or Top): Returns the top item without removing it.
- **isEmpty()**: Checks if the stack is empty.
- **isFull()**: Checks if the stack is full (relevant for fixed-size implementations).

- **Size()**: Returns the number of elements in the stack.

Example:

Imagine a stack for browser history:

- Push: Add a new page visit.
- Pop: Go back to the previous page.
- Peek: View the current top page without navigating.

Stacks can be implemented using arrays (fixed size) or linked lists (dynamic size). Time complexity for push and pop is typically O(1). Your attached "ADT-Stack-lec3.ppsx" likely covers this in detail, including implementations.[8]

Queue ADT

A Queue ADT is a linear data structure that follows the First-In-First-Out (FIFO) principle, where the first element added is the first one removed. It resembles a real-life queue, like people waiting in line, with elements added at the rear and removed from the front. Queues are essential for scheduling, buffering, and breadth-first search in graphs, such as managing print jobs or task queues in operating systems.[7]

Key Operations:

- **Enqueue(item)** (or Add): Adds an item to the rear of the queue. If full, it may cause an overflow.
- **Dequeue()** (or Remove): Removes and returns the front item. If empty, it results in an underflow.
- **Front()** (or Peek): Returns the front item without removing it.
- **isEmpty()**: Checks if the queue is empty.
- **isFull()**: Checks if the queue is full (for fixed-size queues).
- **Size()**: Returns the number of elements in the queue.

Example:

For a customer service queue:

- Enqueue: A new customer joins the end.
- Dequeue: Serve the customer at the front.
- Front: Check who's next without serving.

Queues can be implemented with arrays (using modular arithmetic for circular queues to optimize space) or linked lists. Basic operations are O(1) time complexity. Refer to your "Queue-ADT-Lec-5-and-6.ppsx" for specific lectures on this.[9]

Both Stack and Queue ADTs build on the non-primitive linear data structures from your Module-1 PDF, emphasizing operations over implementation. If you're implementing these in Java for your

projects, start with array-based versions for simplicity. Let me know if you need code examples! [7]

⁂

Abstract Data Type (ADT)

An ADT is a high-level model defining a data structure's behavior through operations and values, abstracting implementation details for modularity.[10]

Stack ADT

Linear LIFO structure for adding/removing from top. Operations: Push (add), Pop (remove), Peek (view top), isEmpty, isFull, Size. Example: Undo in editors. O(1) time.[11] [10]

Queue ADT

Linear FIFO structure for adding to rear, removing from front. Operations: Enqueue (add), Dequeue (remove), Front (view), isEmpty, isFull, Size. Example: Task scheduling. O(1) time.[12] [10]

⁂

Operations on Data Structures

Data structures enable efficient data management through key operations, which vary by type (e.g., arrays, stacks, queues). Here's a concise overview:[13]

- **Traversal**: Accessing each element sequentially, e.g., looping through an array to print values.
- **Insertion**: Adding an element, e.g., at end/middle of array (shifting needed) or push in stack.
- **Deletion**: Removing an element, e.g., from array position (shifting elements) or pop in stack.
- **Searching**: Locating an element, e.g., linear search in arrays or binary in sorted trees.
- **Merging**: Combining two structures, e.g., copying arrays into a third one.
- **Sorting**: Rearranging elements, e.g., ascending/descending order in arrays or lists.

These ensure efficient access and modification, with time complexities depending on the structure (e.g., O(1) for stack push/pop).[13]

⁂

Infix Expressions

Infix notation is the standard arithmetic expression format where operators are placed between operands. It's human-readable but requires parentheses for precedence and is evaluated using rules like BODMAS/PEMDAS. This is the most common form in everyday math but needs stacks for computer parsing to handle operator priority.

Example:
`(A + B) * C`
Here, `+` is between A and B, and `*` is between the result and C. Evaluation: Add A and B first, then multiply by C.

Postfix Expressions (Reverse Polish Notation - RPN)

Postfix places operators after their operands, eliminating the need for parentheses since evaluation is strictly left-to-right with a stack (push operands, pop for operations). It's efficient for machines, often used in compilers and calculators. Conversions from infix use stacks, as covered in your stack applications slides.[16]

Example:
`A B + C *` (equivalent to `(A + B) * C`)
Evaluation: Push A, push B, add (pop B and A, push result), push C, multiply (pop C and sum, push result).

Prefix Expressions (Polish Notation)

Prefix places operators before their operands, also avoiding parentheses and using stack-based evaluation (right-to-left scan). It's useful in functional programming and some parsers, with similar conversion methods via stacks.

Example:
`* + A B C` (equivalent to `(A + B) * C`)
Evaluation: Scan from right, push C, push B, push A, add (pop A and B, push sum), multiply (pop sum and C, push result).

These notations are key in data structures for expression evaluation and parsing, especially with stacks for conversions. In Java, you could implement a stack to convert infix to postfix for your GUI projects.

⁂

Write a C program to implement Stack using array.

IN SHORT

```
#include <stdio.h>
#define MAX 5 // Maximum stack size

int stack[MAX];
```

```
int top = -1;

// Check if stack is full
int isFull() {
    return top == MAX - 1;
}

// Check if stack is empty
int isEmpty() {
    return top == -1;
}

// Push element to stack
void push(int value) {
    if (isFull()) {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = value;
    printf("%d pushed\n", value);
}

// Pop element from stack
int pop() {
    if (isEmpty()) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}

// Peek top element
int peek() {
    if (isEmpty()) {
        printf("Stack Empty\n");
        return -1;
    }
    return stack[top];
}

// Display stack
void display() {
    if (isEmpty()) {
        printf("Stack Empty\n");
        return;
    }
    printf("Stack: ");
    for (int i = 0; i <= top; i++) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}

int main() {
    push(10);
    push(20);
```

```
    push(30);
    display();
    printf("Popped: %d\n", pop());
    printf("Peek: %d\n", peek());
    display();
    return 0;
}
```

This implements a basic stack using an array with operations like push, pop, peek, and display, as per your stack ADT slides. Compile and run in a C environment like GCC. If you need modifications for your Java projects, let me know![23]

⁂

Write a C program to evaluate a postfix expression using STACK ADT.

IN SHORT

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h> // For isdigit
#include <math.h>  // For pow (if needed, but basic ops here)

#define MAX 100

int stack[MAX];
int top = -1;

// Push to stack
void push(int value) {
    if (top >= MAX - 1) {
        printf("Stack Overflow\n");
        exit(1);
    }
    stack[++top] = value;
}

// Pop from stack
int pop() {
    if (top < 0) {
        printf("Stack Underflow\n");
        exit(1);
    }
    return stack[top--];
}

// Evaluate postfix expression
int evaluatePostfix(char* exp) {
    for (int i = 0; exp[i] != '\0'; i++) {
        if (isdigit(exp[i])) {
            push(exp[i] - '0'); // Convert char to int
        } else {
```

```
            int op2 = pop(); // Second operand
            int op1 = pop(); // First operand
            switch (exp[i]) {
                case '+': push(op1 + op2); break;
                case '-': push(op1 - op2); break;
                case '*': push(op1 * op2); break;
                case '/': push(op1 / op2); break;
                default: printf("Invalid operator\n"); exit(1);
            }
        }
    }
    return pop();
}

int main() {
    char exp[] = "231*+9-"; // Example: (2 + (3*1)) - 9 = -4
    int result = evaluatePostfix(exp);
    printf("Result: %d\n", result);
    return 0;
}
```

This program evaluates a postfix expression using a stack (array-based), handling single-digit operands and basic operators for brevity. It aligns with stack applications in your slides. Assumes valid input; compile and test with GCC. For multi-digit or more ops, extend as needed! [27]

⁂

Applications of Stack Data Structure

Stacks are linear data structures following the Last-In-First-Out (LIFO) principle, making them ideal for scenarios requiring reversal or backtracking. Here are two key applications:

1. **Expression Evaluation and Conversion**
   Stacks are widely used in compilers and calculators for evaluating arithmetic expressions, especially in postfix (Reverse Polish Notation) form. For instance, to evaluate a postfix expression like "2 3 + 4 *", operands are pushed onto the stack, and when an operator is encountered, the top two operands are popped, operated on, and the result is pushed back. This eliminates the need for parentheses and handles precedence efficiently. Stacks also convert infix to postfix by managing operators based on priority (e.g., converting "(A + B) * C" to "A B + C *"). This application is crucial in programming languages for parsing and ensures O(n) time complexity for evaluation. Without stacks, handling nested expressions would be complex and error-prone.[31]
2. **Function Call Management and Recursion**
   Stacks manage function calls in programming, storing return addresses, parameters, and local variables in the call stack. During recursion, like in factorial computation (e.g., fact(n) calls fact(n-1) until base case), each recursive call pushes a new frame onto the stack, and pops occur on returns. This prevents infinite loops via stack overflow detection and enables backtracking in algorithms like depth-first search (DFS) in graphs. In operating systems, it's

essential for interrupt handling. This LIFO nature ensures the most recent call is resolved first, optimizing memory usage in runtime environments.

Applications of Queue Data Structure

Queues follow the First-In-First-Out (FIFO) principle, suitable for ordered processing like buffering. Here are two applications:

1. **Process Scheduling in Operating Systems**
   Queues manage CPU scheduling, where processes are enqueued in a ready queue and dequeued for execution in algorithms like First-Come-First-Served (FCFS). For example, print jobs in a printer spooler are queued, ensuring the first submitted job prints first. Circular queues prevent space wastage in fixed-size implementations, as seen in round-robin scheduling where processes cycle through the queue with time quanta. This ensures fairness and efficiency, with O(1) enqueue/dequeue operations, critical for multitasking OS like Windows or Linux.[32]
2. **Breadth-First Search (BFS) in Graphs**
   Queues facilitate BFS for traversing graphs or trees level by level, enqueuing nodes and dequeuing to explore neighbors. For instance, in finding the shortest path in an unweighted graph (e.g., social network friend recommendations), start from a source, enqueue it, and process levels sequentially. This guarantees O(V + E) time complexity, where V is vertices and E is edges, making it ideal for applications like web crawlers or puzzle solvers (e.g., solving a maze). Queues ensure FIFO order, preventing revisits via visited arrays, unlike stack-based DFS.

⁂

Write a menu-driven program in C to implement QUEUE ADT. (using array)

IN SHORT

```
#include <stdio.h>
#define MAX 5

int queue[MAX];
int front = -1, rear = -1;

// Check if queue is empty
int isEmpty() {
    return front == -1;
}

// Check if queue is full
int isFull() {
    return rear == MAX - 1;
}

// Enqueue element
```

```
void enqueue(int value) {
    if (isFull()) {
        printf("Queue Overflow\n");
        return;
    }
    if (isEmpty()) front = 0;
    queue[++rear] = value;
    printf("%d enqueued\n", value);
}

// Dequeue element
int dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow\n");
        return -1;
    }
    int value = queue[front++];
    if (front > rear) front = rear = -1;
    return value;
}

// Peek front element
int peek() {
    if (isEmpty()) {
        printf("Queue Empty\n");
        return -1;
    }
    return queue[front];
}

// Display queue
void display() {
    if (isEmpty()) {
        printf("Queue Empty\n");
        return;
    }
    printf("Queue: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

int main() {
    int choice, value;
    while (1) {
        printf("\n1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: printf("Enter value: "); scanf("%d", &value); enqueue(value); break;
            case 2: value = dequeue(); if (value != -1) printf("Dequeued: %d\n", value);
            case 3: value = peek(); if (value != -1) printf("Front: %d\n", value); break;
            case 4: display(); break;
            case 5: return 0;
            default: printf("Invalid choice\n");
        }
```

```
    }
}
```

This menu-driven program implements a basic Queue ADT using an array, with operations like enqueue, dequeue, peek, and display. It handles overflow/underflow and is non-circular for simplicity. Compile and run in a C environment. For circular queue, refer to your lecture slides.[35]

⁂

Linked List as an Abstract Data Type (ADT)

An Abstract Data Type (ADT) is a high-level model that defines a data structure in terms of its behavior, operations, and possible values, abstracting away implementation details like memory allocation. This allows focus on "what" the structure does rather than "how" it's built, promoting modularity and reusability in programming.[39]

A Linked List ADT represents a dynamic, linear collection of elements (nodes) where each node contains data and a reference (pointer) to the next node, enabling efficient insertions and deletions without contiguous memory. Unlike arrays, linked lists grow or shrink dynamically, making them ideal for scenarios with unpredictable sizes, such as implementing stacks, queues, or graphs. The ADT specifies operations without mandating how nodes are stored (e.g., singly, doubly, or circular variants).[40]

Key Components of Linked List ADT

- **Node Structure**: Each element (node) holds a value (data) and a link to the next node. In a singly linked list, it's one-way; doubly adds a previous link for bidirectional traversal.[40]
- **Head/Tail Pointers**: A reference to the first (head) and optionally last (tail) node for quick access.

Operations on Linked List ADT

The ADT defines these core operations, typically with O(1) to O(n) time complexity depending on position:[40]

- **Traversal**: Visit each node sequentially, e.g., printing elements from head to tail. Time: O(n).
- **Insertion**: Add a new node at the beginning, end, or after/before a specific node. For example, inserting at the start updates the head pointer.[40]
- **Deletion**: Remove a node from the beginning, end, or after/before a given node, adjusting pointers to maintain links.[40]
- **Search**: Find a node by value, traversing from head. Returns position or null if not found.[40]
- **isEmpty**: Check if the list has no nodes (head is null).
- **Size**: Count nodes by traversal.

Example

Consider a singly linked list ADT for student records:[40]

- Nodes: {Data: Student ID, Next: Pointer to next student}.
- Insert at beginning: New node points to current head; update head.
- Delete last node: Traverse to second-last, set its next to null.

Advantages and Use Cases

As an ADT, linked lists offer flexibility over fixed-size arrays, with no wasted space and easy modifications. They're used in browser history (forward/back navigation via doubly links) or playlists (circular for looping). However, random access is O(n), unlike arrays' O(1).

In C (from your slides), it's implemented with structs and pointers, but the ADT remains implementation-agnostic. For Java projects, use classes for nodes to encapsulate this ADT.[40]

⁂

Types of Linked Lists

Linked lists are dynamic data structures consisting of nodes, each containing data and pointer(s) to other nodes. They allow efficient insertions and deletions compared to arrays. Based on your lecture slides, here are the main types, explained with textual diagrams for clarity.[43] [44] [45]

1. Singly Linked List

This is the simplest type, where each node has data and a single pointer to the next node. Traversal is unidirectional (forward only), starting from the head. It's useful for basic lists but lacks backward navigation.

- **Advantages**: Simple, low memory overhead.
- **Disadvantages**: No reverse traversal; deletion requires previous node access.

Diagram (Textual Representation):

```
Head -> [Data: A | Next] -> [Data: B | Next] -> [Data: C | Next] -> NULL
```

2. Doubly Linked List

Each node has data and two pointers: one to the next node and one to the previous node. This enables bidirectional traversal. It's ideal for applications needing forward/backward movement, like browser history.

- **Advantages**: Easy insertion/deletion from both ends; reverse traversal possible.
- **Disadvantages**: Higher memory usage due to extra pointer.

Diagram (Textual Representation):

```
NULL <- [Prev | Data: A | Next] <-> [Prev | Data: B | Next] <-> [Prev | Data: C | Next] -
```

3. Circular Linked List

A variation of singly linked list where the last node's next pointer points back to the first node (head), forming a circle. No NULL at the end, allowing continuous looping. Useful for round-robin scheduling.

- **Advantages**: Efficient for cyclic operations; no end marker needed.
- **Disadvantages**: Risk of infinite loops if not handled carefully; finding end requires traversal.

Diagram (Textual Representation):

```
Head -> [Data: A | Next] -> [Data: B | Next] -> [Data: C | Next] -> (back to Head)
```

4. Circular Doubly Linked List

Combines doubly linked list with circular structure: last node's next points to first, and first's previous points to last. Enables bidirectional circular traversal. Common in advanced applications like music playlists with loop and reverse.

- **Advantages**: Full bidirectional access in a cycle; efficient for insertions/deletions.
- **Disadvantages**: Most memory-intensive; complex pointer management.

Diagram (Textual Representation):

```
<-> [Prev | Data: A | Next] <-> [Prev | Data: B | Next] <-> [Prev | Data: C | Next] <-> (
```

These types build on the basic linked list ADT, with variations suiting different needs like traversal direction or cyclicity. In C implementations from your slides, they use structs with pointers for nodes. If implementing in Java, use classes for encapsulation.[44] [45] [43]

⁂

Advantages of Linked Lists Over Arrays

Linked lists offer several benefits over arrays, especially in scenarios requiring dynamic data management. These advantages stem from their node-based structure, where elements (nodes) are linked via pointers rather than stored in contiguous memory. Here's a breakdown:[47]

- **Dynamic Size**: Unlike arrays, which have a fixed size declared at creation, linked lists can grow or shrink dynamically by allocating/deallocating nodes at runtime. This eliminates the need for resizing (e.g., no copying elements to a new array) and prevents wasted space or overflow errors when the number of elements is unpredictable.[47]

- **Efficient Insertions and Deletions**: Adding or removing elements in a linked list is faster and more efficient. For example, inserting at the beginning or end is O(1) time complexity by updating pointers, without shifting elements. In arrays, insertions/deletions often require O(n) time due to shifting all subsequent elements. This is highlighted in operations like inserting before/after a node in your slides.[47]
- **No Contiguous Memory Requirement**: Linked lists don't need a single block of contiguous memory; nodes can be scattered across memory. This is advantageous in fragmented memory environments or when large contiguous blocks aren't available, unlike arrays which demand sequential space.[47]
- **Flexibility for Advanced Structures**: Linked lists easily support variations like doubly or circular lists for bidirectional traversal or looping, which arrays can't replicate without extra overhead. They're ideal for implementing other ADTs like stacks (push/pop via head) or queues (enqueue/dequeue via tail), with dynamic growth.[48]
- **Better Memory Utilization in Sparse Data**: For sparse or irregularly sized data, linked lists use memory only for actual elements, avoiding the pre-allocation issues in arrays that might lead to unused slots.

However, linked lists have trade-offs like slower random access (O(n) vs. arrays' O(1)) and extra memory for pointers. In your computer engineering projects, use linked lists for flexible, insertion-heavy tasks like dynamic databases, as per the examples in your lecture materials. If you need code examples in C or Java, let me know![49] [50] [47]

⁂

Linked Representation of Stack

In linked representation, a stack is implemented using a singly linked list where each node contains data and a pointer to the next node. The top of the stack is a pointer (TOP) to the first node; the last node's next is NULL. This allows dynamic size without fixed limits, unlike arrays.[51]

Node Structure (in C):

```
struct Node {
    int data;
    struct Node* next;
};
struct Node* TOP = NULL; // Initially empty
```

Push Operation (Insert at Beginning):

- Allocate a new node, set its data, and point its next to current TOP.
- Update TOP to the new node.
- Time: O(1).
- Example: After push, new node becomes TOP.[51]

Pop Operation (Delete from Beginning):

- If TOP is NULL, underflow.
- Set a temp pointer to TOP, update TOP to TOP→next, free temp.
- Time: O(1).
- Example: Removes and returns TOP's data, updates TOP.[51]

Advantages: No overflow (grows dynamically), efficient memory use. Used in recursion and backtracking.

⁂

Program on STACK as a SLL.

```
#include <stdio.h>
#include <stdlib.h>

// Node structure for Singly Linked List
struct Node {
    int data;
    struct Node* next;
};

struct Node* top = NULL; // Stack top pointer

// Check if stack is empty
int isEmpty() {
    return top == NULL;
}

// Push (Insert at beginning)
void push(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Stack Overflow\n");
        return;
    }
    newNode->data = value;
    newNode->next = top;
    top = newNode;
    printf("%d pushed\n", value);
}

// Pop (Delete from beginning)
int pop() {
    if (isEmpty()) {
        printf("Stack Underflow\n");
        return -1;
    }
    struct Node* temp = top;
    int value = temp->data;
    top = top->next;
```

```
    free(temp);
    return value;
}

// Peek top element
int peek() {
    if (isEmpty()) {
        printf("Stack Empty\n");
        return -1;
    }
    return top->data;
}

// Display stack
void display() {
    if (isEmpty()) {
        printf("Stack Empty\n");
        return;
    }
    struct Node* current = top;
    printf("Stack: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    int choice, value;
    while (1) {
        printf("\n1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: printf("Enter value: "); scanf("%d", &value); push(value); break;
            case 2: value = pop(); if (value != -1) printf("Popped: %d\n", value); break;
            case 3: value = peek(); if (value != -1) printf("Top: %d\n", value); break;
            case 4: display(); break;
            case 5: return 0;
            default: printf("Invalid choice\n");
        }
    }
}
```

This menu-driven C program implements a Stack using a Singly Linked List (SLL), with push (insert at beginning), pop (delete from beginning), peek, and display operations. It handles underflow and dynamic memory allocation, aligning with your lecture slides on linked stack representation. Compile and run in a C environment like GCC. If you need adjustments, let me know![55]

⁂

1. Module-1-Introduction-to-Data-Structures.pdf
2. ADT-Stack-lec3.ppsx

3. Queue-ADT-Lec-5-and-6.ppsx
4. Module-1-Introduction-to-Data-Structures.pdf
5. ADT-Stack-lec3.ppsx
6. Queue-ADT-Lec-5-and-6.ppsx
7. Module-1-Introduction-to-Data-Structures.pdf
8. ADT-Stack-lec3.ppsx
9. Queue-ADT-Lec-5-and-6.ppsx
10. Module-1-Introduction-to-Data-Structures.pdf
11. ADT-Stack-lec3.ppsx
12. Queue-ADT-Lec-5-and-6.ppsx
13. Module-1-Introduction-to-Data-Structures.pdf
14. ADT-Stack-lec3.ppsx
15. Queue-ADT-Lec-5-and-6.ppsx
16. Module-2-Stack-Applications.ppsx
17. ADT-Stack-lec3.ppsx
18. Queue-ADT-Lec-5-and-6.ppsx
19. Module-1-Introduction-to-Data-Structures.pdf
20. Module-2-Priority-Queue-and-dqueue-Lec12-1.ppt
21. Linked-Lists-Lec-14.pptx
22. Module-2-Circular-queue-Lec11.pptx
23. Module-2-Priority-Queue-and-dqueue-Lec12-1.ppt
24. Linked-Lists-Lec-14.pptx
25. Module-2-Stack-Applications.ppsx
26. Module-2-Circular-queue-Lec11.pptx
27. Module-2-Stack-Applications.ppsx
28. Module-2-Priority-Queue-and-dqueue-Lec12-1.ppt
29. Linked-Lists-Lec-14.pptx
30. Module-2-Circular-queue-Lec11.pptx
31. Module-2-Stack-Applications.ppsx
32. Module-2-Circular-queue-Lec11.pptx
33. Module-2-Priority-Queue-and-dqueue-Lec12-1.ppt
34. Linked-Lists-Lec-14.pptx
35. Module-2-Circular-queue-Lec11.pptx
36. Module-2-Priority-Queue-and-dqueue-Lec12-1.ppt
37. Linked-Lists-Lec-14.pptx
38. Module-2-Stack-Applications.ppsx
39. Doubly-linked-list_lec2425.pptx
40. Circular-Linked-List-and-Doubly-linked-list.pptx

41. Linked-Lists-Lec-14.pptx
42. Linked-Stack-and-Queue-Lec18.pptx
43. Linked-Lists-Lec-14.pptx
44. Circular-Linked-List-and-Doubly-linked-list.pptx
45. Doubly-linked-list_lec2425.pptx
46. Linked-Stack-and-Queue-Lec18.pptx
47. Linked-Lists-Lec-14.pptx
48. Linked-Stack-and-Queue-Lec18.pptx
49. Circular-Linked-List-and-Doubly-linked-list.pptx
50. Doubly-linked-list_lec2425.pptx
51. Linked-Stack-and-Queue-Lec18.pptx
52. Linked-Lists-Lec-14.pptx
53. Circular-Linked-List-and-Doubly-linked-list.pptx
54. Doubly-linked-list_lec2425.pptx
55. Linked-Stack-and-Queue-Lec18.pptx
56. Linked-Lists-Lec-14.pptx
57. Circular-Linked-List-and-Doubly-linked-list.pptx
58. Doubly-linked-list_lec2425.pptx