Introduction to Data Structures

Prof. John Smith

Fall 2024

Outline

Introduction

What are Data Structures?

- Data structures organize and store data efficiently
- Enable fast access, insertion, and deletion operations
- Form the foundation of algorithm design

N

> Choosing the right data structure is crucial for program performance!

Arrays

Array Basics

[

Array Definition] An **array** is a contiguous block of memory that stores elements of the same type, accessible by index.

Declaration in C++:

```
// Static array
int arr[5] = {1, 2, 3, 4, 5};

// Dynamic array
int* dynamicArr = new int[10];

// C++ vector (recommended)
std::vector<int> vec = {1, 2, 3, 4, 5};
```

Array Complexity

Operation	Time Complexity	Notes
Access by index	$O(1)$	Direct memory access
Search	$O(n)$	Linear scan required
Insert at end	$O(1)$	Amortized for vectors
Insert at position	$O(n)$	Requires shifting
Delete	$O(n)$	Requires shifting

Key Insight: Arrays excel at random access but struggle with insertions/deletions.

Linked Lists

Linked List Structure

[

Linked List] A **linked list** is a linear data structure where elements (nodes) are connected via pointers.

Node Structure:

```
struct Node {
    int data;          // Data field
    Node* next;        // Pointer to next node

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

Linked List Operations

Insertion at Head

```
void insertAtHead(Node*& head, int value) {
    Node* newNode = new Node(value);
    newNode->next = head;
    head = newNode;
}
```

Complexity Analysis:

- Time: $\overbrace{O(1)}^{\text{constant time}}$
- Space: $O(1)$ auxiliary space

Comparison

Arrays vs. Linked Lists

Arrays

- \+ Fast random access
- \+ Cache-friendly
- \+ Less memory overhead
- \- Fixed size (static)
- \- Expensive insertions

Linked Lists

- \+ Dynamic size
- \+ Efficient insertions
- \+ No waste of space
- \- No random access
- \- Extra memory for pointers

Design Choice

Use arrays for **frequent access**, linked lists for **frequent modifications**.

Practice Problems

Exercise 1: Reverse an Array

Problem: Write a function to reverse an array in-place.

Answers to Exercise 1

Approach: Use two pointers (left and right) and swap elements while moving toward the center.

Time Complexity: $O(n)$
Space Complexity: $O(1)$

Key idea: Swap elements at positions i and $n-1-i$ for $i=0$ to $\lfloor n/2 \rfloor$.

Mathematical Notation Examples

Using custom commands for emphasis:

Array index calculation:

$$\text{address}(A[i]) = \overbrace{\text{base\_address}}^{\text{starting point}} + \underbrace{i \times \text{element\_size}}_{\text{offset}}$$

Amortized analysis:

$$\text{Total cost} = \sum_{i=1}^{n} c_i \leq O(n)$$

Summary

- Arrays provide $O(1)$ **access** but $O(n)$ **insertions**
- Linked lists offer $O(1)$ **insertions** but $O(n)$ **access**
- Choice depends on the application requirements
- Understanding trade-offs is essential for efficient programming

Questions?