Presentation

Assignment Final

Data Structure & Algorithms

Student Name: Phan Hoai Nam
Student ID: BH01294
Class: SE06304
Assessor Name: Dinh Van Dong

Introduction

In this presentation, we will delve into the design specification process for data structures by examining the valid operations through practical examples. We will start by exploring the Stack Abstract Data Type (ADT), detailing its design and operations, and comparing it with a concrete implementation of a First In First Out (FIFO) queue. This comparison will emphasize the behavioral and usage differences between a stack, which adheres to the Last In First Out (LIFO) principle, and a FIFO queue. Additionally, we will analyze two different sorting algorithms, discussing their design specifications and operations on data structures to achieve ordered data. Our evaluation will focus on various sorting algorithms for managing student information within our application, covering their strengths and weaknesses in terms of efficiency and suitability for our requirements. Benchmarking results will provide empirical evidence of performance differences, supplemented by theoretical analysis to support our findings. This comprehensive review aims to identify the most effective algorithm to enhance the application's performance and reliability.

BTEC

Stack ADT (Abstract Data Type)

Definition

A linear data structure that follows the Last In First Out (LIFO) principle. Elements are added and removed from the same end, known as the “top“ of the stack.

The Operations

· Push(x): Add an element x to the top of the stack.
· Pop(): Remove and return the top element of the stack.
· Peek(): Return the top element without removing it.
· IsEmpty(): Check if the stack is empty.
· IsFull(): Check if the stack is full (for fixed-size stacks).

BTEC

Stack ADT (Abstract Data Type)

Specify Input Parameters

· Push(x): x is the element to be added.
· Pop(): No input parameters.
· Peek(): No input parameters.
· IsEmpty(): No input parameters.
· IsFull(): No input parameters.

Time and Space Complexity

· Push(x): O(1) time, O(1) space.
· Pop(): O(1) time, O(1) space.
· Peek(): O(1) time, O(1) space.
· IsEmpty(): O(1) time, O(1) space.
· IsFull(): O(1) time, O(1) space.

BTEC

Pre- and Post-conditions

· Push(x):
  - Pre-condition: The stack is not full.
  - Post-condition: Element x is added to the top of the stack.

· Pop():
  - Pre-condition: The stack is not empty.
  - Post-condition: The top element is removed and returned.

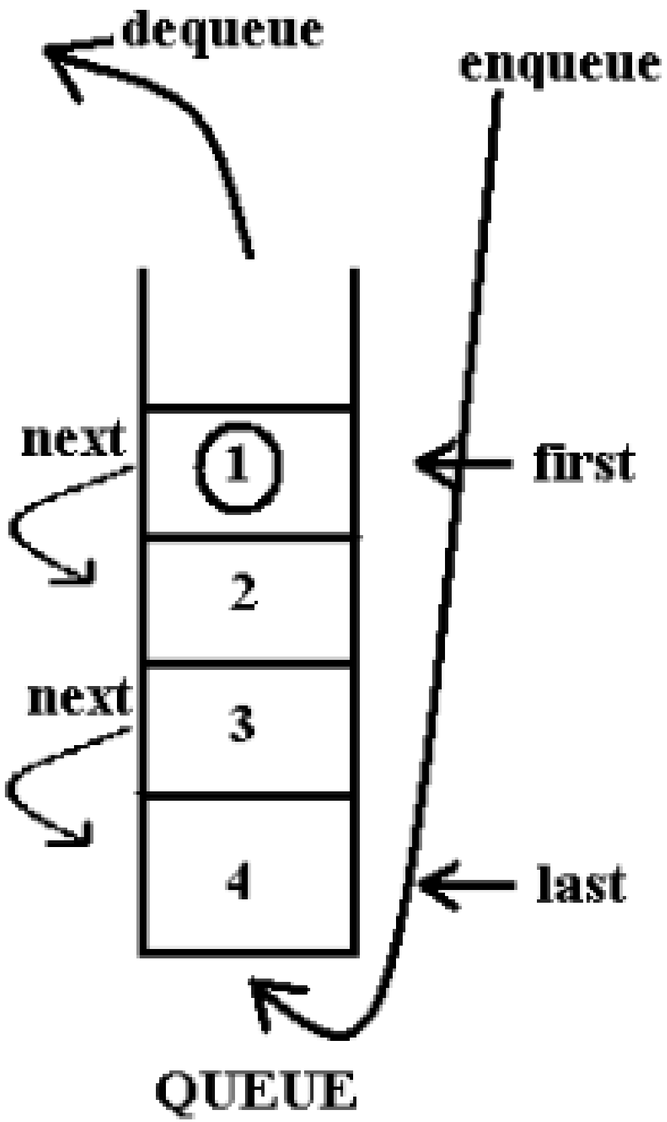
· Peek():
  - Pre-condition: The stack is not empty.
  - Post-condition: The top element is returned without being removed.

· IsEmpty():
  - Pre-condition: None.
  - Post-condition: Returns true if the stack is empty, false otherwise.

· IsFull():
  - Pre-condition: None.
  - Post-condition: Returns true if the stack is full, false otherwise.

Queue

A FIFO queue is a fundamental data structure that follows the First In, First Out principle, meaning the first element added to the queue will be the first one to be removed. This structure is crucial in various real-world applications like scheduling processes in operating systems, handling requests in web servers, and managing tasks in print queues.



Introduction

Queue is a data structure where the first element added is the first one to be removed. It ensures that elements are processed in the order they were added.

Define the Structure

Queue can be implemented using arrays or linked lists. The primary operations are enqueue (add to the rear) and dequeue (remove from the front).

Array-Based Implementation

In an array-based implementation, we use a circular buffer to efficiently manage the queue. This approach allows the queue to wrap around to the beginning of the array when the end is reached.

Pros:

- Simple to implement.
- Provides direct access to elements via indexing.

Cons:

- Fixed size, which can lead to overflow if not managed properly.
- Inefficient use of space if the queue size is not optimal.

Linked List-Based Implementation

In a linked list-based implementation, we use nodes to represent elements, with each node pointing to the next node in the queue. This dynamic structure allows the queue to grow and shrink as needed.

Pros:

- Dynamic size, which can grow or shrink as needed.
- Efficient memory usage.

Cons:

- Slightly more complex to implement.
- Additional memory overhead for storing pointers.

BTEC

Sorting Algorithms

Selection Sort and Quick Sort

- Selection Sort: A simple comparison-based sorting algorithm that divides the input list into two parts: a sorted part and an unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the sorted part.

- Quick Sort: An efficient divide-and-conquer sorting algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

BTEC

Performance of Two Sorting Algorithms

Selection Sort

- Algorithm: Iteratively finds the minimum element from the unsorted part and swaps it with the first unsorted element.
- Time Complexity: O(n^2) for all cases.
- Space Complexity: O(1), as it sorts in place.
- Stability: Not stable, because it swaps elements that may not preserve the original order of equal elements.

Quick Sort

- Algorithm: Divides the array into smaller sub-arrays around a pivot and recursively sorts the sub-arrays.
- Time Complexity: O(n log n) on average, but O(n^2) in the worst case (e.g., when the smallest or largest element is always picked as the pivot).
- Space Complexity: O(log n) due to the recursion stack.
- Stability: Not stable by default, but can be made stable with modifications.

BTEC

Selection Sort Method

- Method: sortStudentsBySelectionSort
- Loop through students: Iterate over the list of students.
- Find the maximum: For each student, find the student with the highest marks in the remaining list.
- Swap: Swap the student with the highest marks to the current position.

```
public void sortStudentsBySelectionSort() {  1 usage
    int n = students.size();
    for (int i = 0; i < n - 1; i++) {
        int maxIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (students.get(j).marks > students.get(maxIndex).marks) {
                maxIndex = j;
            }
        }
        // Swap
        Student temp = students.get(maxIndex);
        students.set(maxIndex, students.get(i));
        students.set(i, temp);
    }
}
```

Quicksort Method

Method: sortStudentsByQuickSort

- Initialize: Call quickSort(0, students.size() - 1).

Method: quickSort

- Recursive sorting: If the sublist has more than one element:
- Partition the list and get the pivot index.
- Recursively sort the sublists before and after the pivot.

```
public void sortStudentsByQuickSort() {  1 usage
    quickSort( low: 0,  high: students.size() - 1);
}

private void quickSort(int low, int high) {  3 usages
    if (low < high) {
        int pi = partition(low, high);
        quickSort(low,  high: pi - 1);
        quickSort( low: pi + 1, high);
    }
}
```

BTEC

Quicksort Method

Method: partition

- Select pivot: Choose the last element as the pivot.
- Partition: Reorganize the list so students with marks greater than or equal to the pivot are on the left.
- Swap pivot: Place the pivot in its correct position.

```
private int partition(int low, int high) {  1 usage
    double pivot = students.get(high).marks;
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (students.get(j).marks >= pivot) {  // Change
            i++;
            // Swap
            Student temp = students.get(i);
            students.set(i, students.get(j));
            students.set(j, temp);
        }
    }
    // Swap
    Student temp = students.get(i + 1);
    students.set(i + 1, students.get(high));
    students.set(high, temp);
```

BTEC

Algorithms

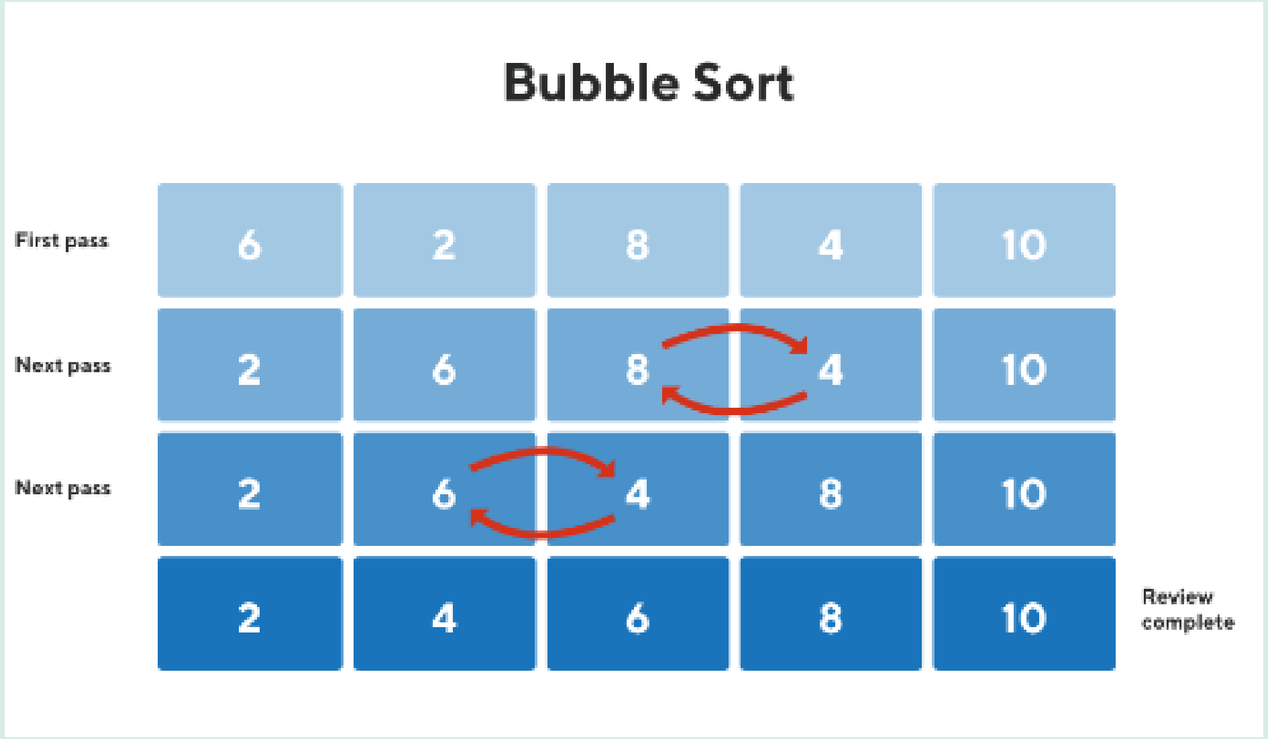
Algorithms Description

Bubble Sort

Description: Bubble Sort is a straightforward sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Algorithm Steps:

1. Compare the first two elements of the list.
2. Swap them if they are in the wrong order.
3. Move to the next pair of elements and repeat the process until the end of the list is reached.
4. Repeat the entire process for the entire list until no swaps are needed in a complete pass.



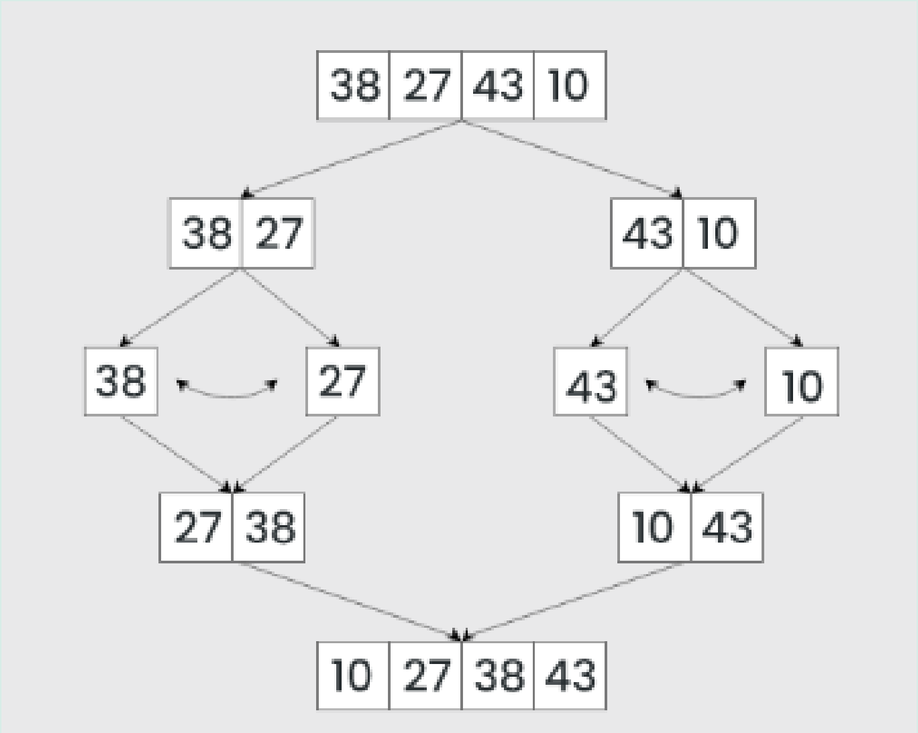
Algorithms Description

Merge Sort

Description: Merge Sort is an efficient, stable, and comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It divides the list into halves, sorts each half recursively, and then merges the sorted halves to produce a sorted list.

Algorithm Steps:

1. Divide the list into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves into a single sorted list.



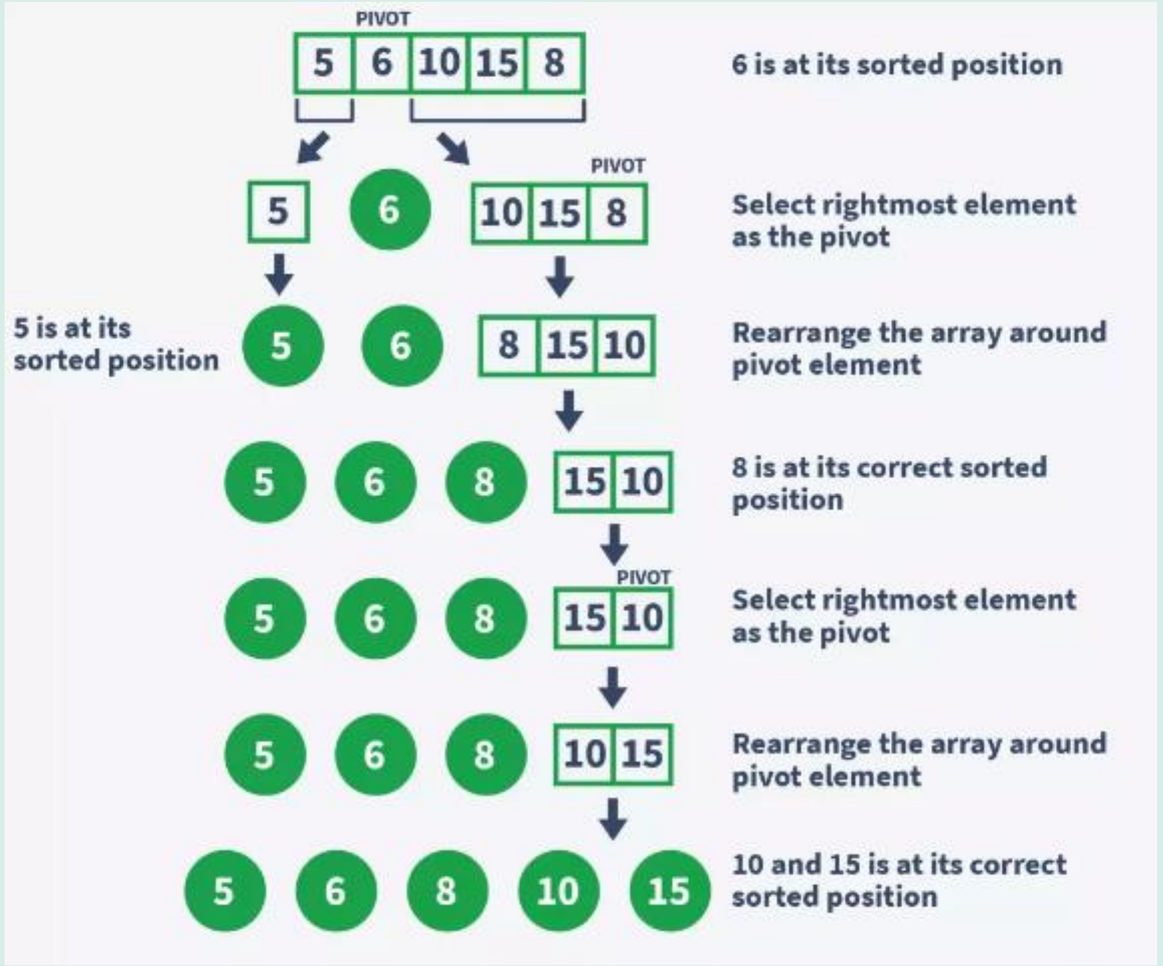
Algorithms Description

Quick Sort

Description: Quick Sort is an efficient, in-place, comparison-based sorting algorithm. It selects a 'pivot' element, partitions the array around the pivot, and recursively sorts the partitions.

Algorithm Steps:

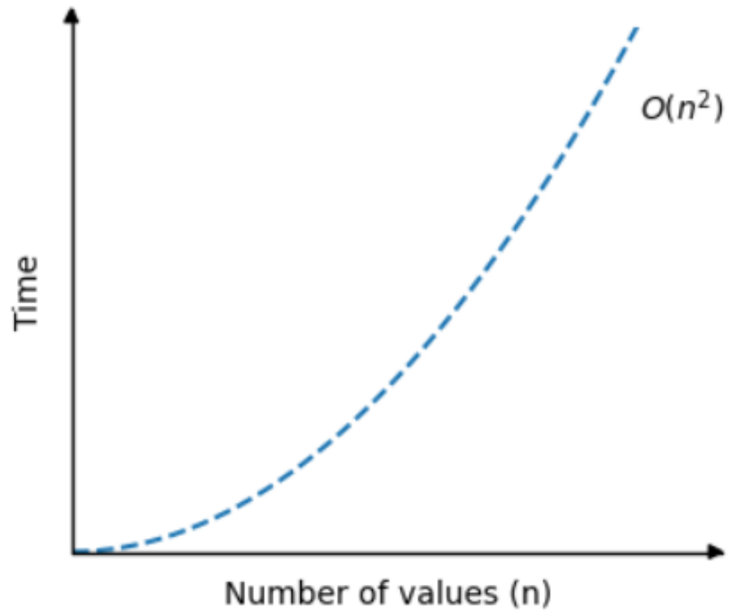
1. Choose a pivot element from the list.
2. Partition the list into two sublists: elements less than the pivot and elements greater than the pivot.
3. Recursively apply Quick Sort to the sublists.



Performance

Bubble Sort

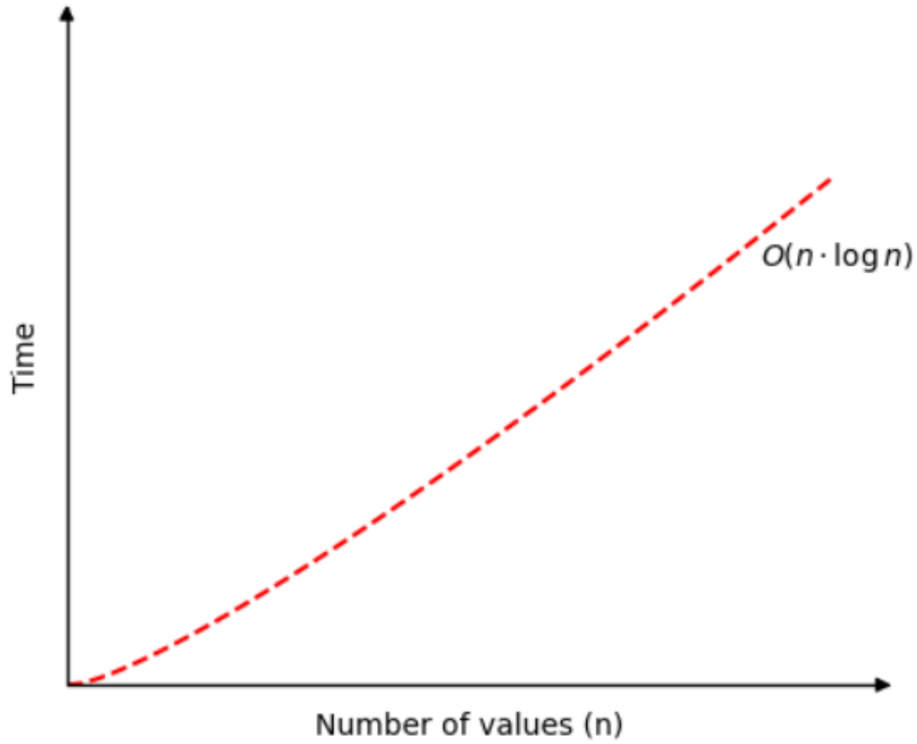
- Time Complexity: O(n^2) in the worst and average cases, as it requires n-1 passes through the list with n comparisons each.
- Space Complexity: O(1), since it sorts in place and requires only a constant amount of additional memory.



Performance

Merge Sort

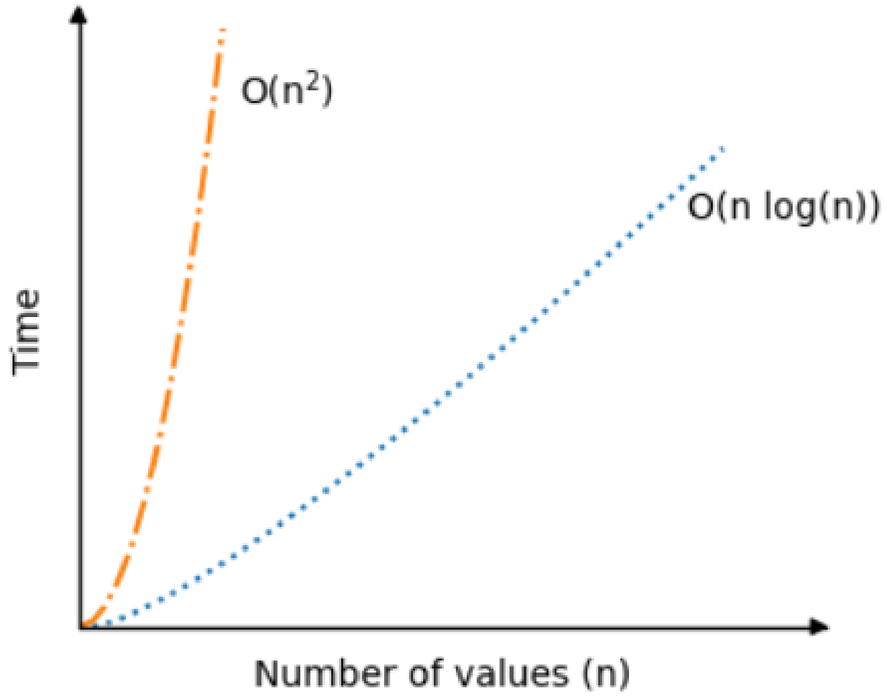
- Time Complexity: O(n log n) for all cases, as it always divides the list in half and requires log n levels of recursion with n comparisons at each level.
- Space Complexity: O(n), as it requires additional space for the temporary arrays used in merging.



Performance

Quick Sort

- Time Complexity: Average-case is O(n log n); worst-case is O(n^2), usually avoided with good pivot selection strategies (e.g., random pivot, median-of-three).
- Space Complexity: O(log n) in the best case with tail recursion; O(n) in the worst case due to the stack depth.



Use Cases and Avoidance

Bubble Sort:

- Use For: Teaching and very small datasets.
- Avoid For: Large datasets due to poor performance.

Merge Sort:

- Use For: Large datasets needing stable sorting, such as linked lists or external sorting.
- Avoid For: Memory-constrained environments.

Quick Sort:

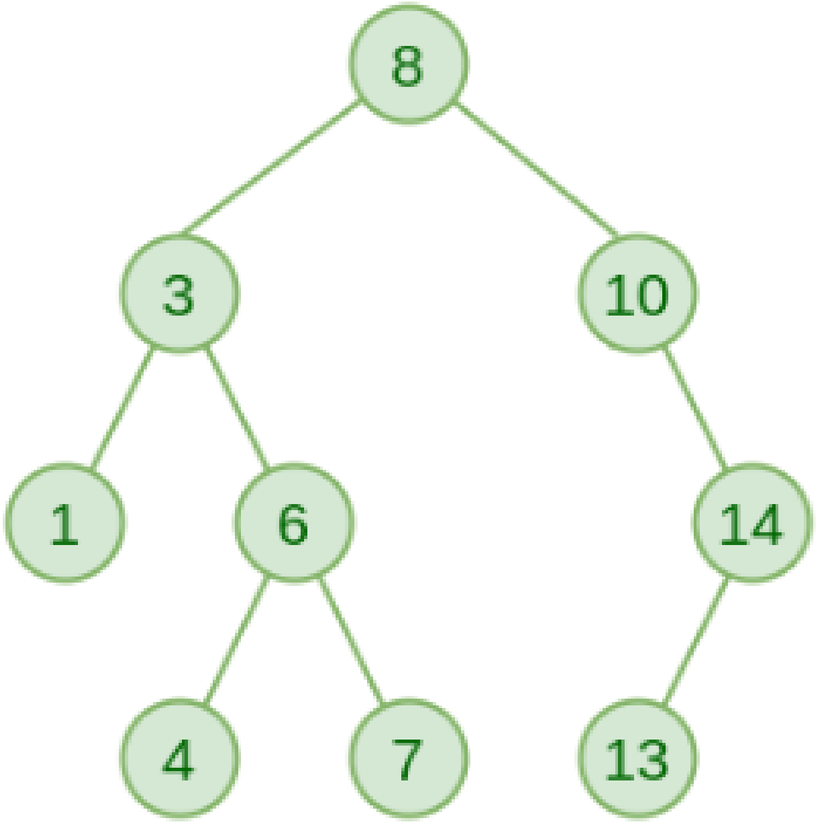
- Use For: Large, unsorted datasets with good pivot selection strategies.
- Avoid For: Nearly sorted datasets to prevent worst-case performance.

Binary Search Tree (BST) and Prim’s/Dijkstra’s Algorithms

BTEC

Binary Search Tree

Description: A Binary Search Tree (BST) is a node-based data structure in which each node has at most two children, referred to as the left and right child. Each node contains a key, and the left subtree of a node contains only nodes with keys less than the node's key, while the right subtree contains only nodes with keys greater than the node's key.



Properties of BST:

- Left Subtree: All nodes have values less than the parent node.
- Right Subtree: All nodes have values greater than the parent node.
- No Duplicate Nodes: Each node has a unique value.

Operations on BST:

- Insertion: Add a node in a manner that preserves the BST property.
- Deletion: Remove a node and rearrange the tree to maintain the BST property.
- Search: Find a node with a specific value.
- Traversal: Visit all nodes in a specific order (inorder, preorder, postorder).

Advantages of BST:

- Efficient Searching: Average-case time complexity for search, insert, and delete operations is O(log n), making it efficient for lookups.
- Dynamic Set Operations: Supports insertion and deletion operations efficiently.
- Sorted Order: Inorder traversal of a BST gives nodes in ascending order, which can be useful for operations that need sorted data.

Disadvantages of BST:

- Unbalanced Trees: The performance degrades to O(n) in the worst case when the tree becomes unbalanced (e.g., if elements are inserted in sorted order).
- Overhead: Additional memory overhead due to the storage of pointers for each node.
- Complexity: Implementation and maintenance can be more complex compared to simpler data structures like arrays or linked lists.

Use Cases:

- Database Indexing: Efficient lookup, insertion, and deletion of records.
- Network Router Algorithms: For quick route lookup.
- Dynamic Set Operations: Such as maintaining a sorted list of elements with frequent insertions and deletions.

Avoidance:

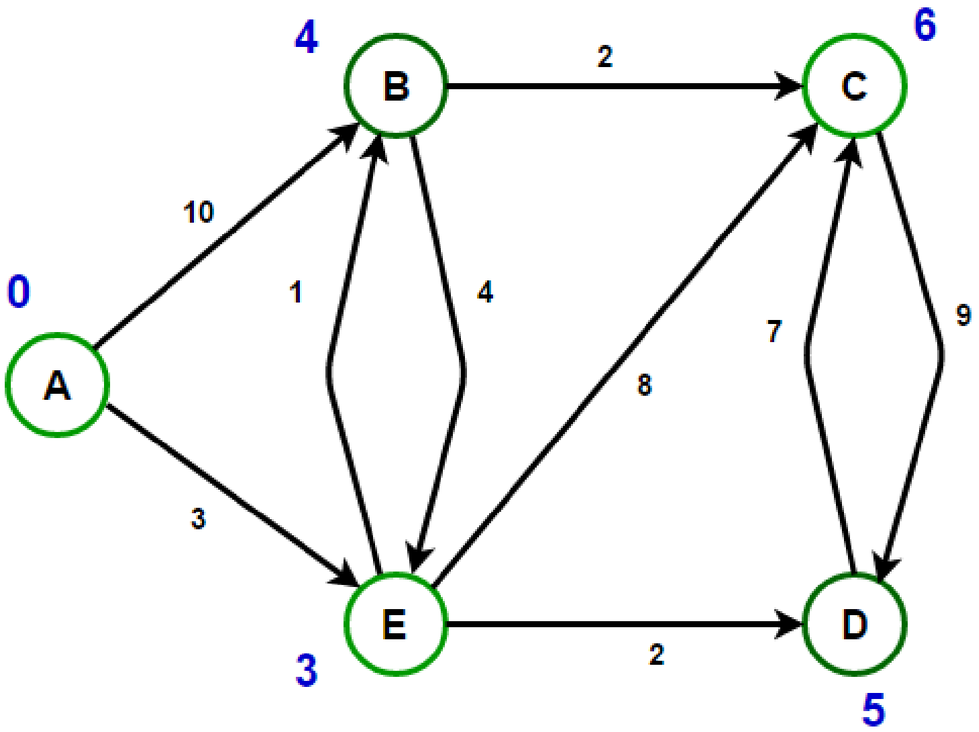
- Highly Unbalanced Data: When the input data is highly skewed, leading to unbalanced trees. In such cases, self-balancing trees like AVL trees or Red-Black trees might be preferred.

Dijkstra’s Algorithms

Description: Dijkstra’s Algorithm finds the shortest paths from a single source vertex to all other vertices in a weighted graph. It works only for graphs with non-negative weights.

Steps:

1. Initialize the distance to the source vertex to 0 and all other vertices to infinity.
2. Set the source vertex as the current vertex.
3. For the current vertex, update the distances to its adjacent vertices.
4. Mark the current vertex as visited.
5. Select the unvisited vertex with the smallest distance as the new current vertex.
6. Repeat steps 3-5 until all vertices are visited or the smallest distance among unvisited vertices is infinity.

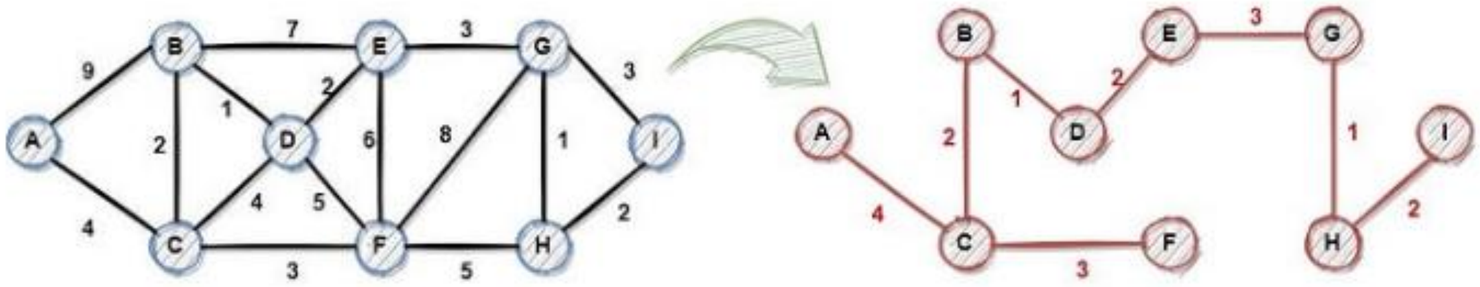


Prim’s Algorithms

Description: Prim’s Algorithm is used to find the Minimum Spanning Tree (MST) for a weighted undirected graph. The MST is a subset of the graph’s edges that connect all vertices without any cycles and with the minimum possible total edge weight.

Steps:

1. Initialize the MST with a starting vertex.
2. Find the edge with the minimum weight that connects a vertex in the MST to a vertex outside the MST.
3. Add this edge to the MST.
4. Repeat steps 2 and 3 until all vertices are included in the MST.



Advantages/Disadvantages and Time Comflexity

Prim's Algorithms

Time Complexity: O(V^2) with adjacency matrix representation or O(E log V) with adjacency list and a priority queue.

Advantages:

- Efficient for dense graphs.
- Simple to implement.

Disadvantages:

- Less efficient for sparse graphs compared to Kruskal's algorithm.

Dijkstra's Algorithms

Time Complexity: O(V^2) with an adjacency matrix or O(E log V) with an adjacency list and a priority queue.

Advantages:

- Efficient for graphs with non-negative weights.
- Provides the shortest path from the source to all other vertices.

Disadvantages:

- Cannot handle graphs with negative weight edges.
- Inefficient for very large graphs with dense connectivity.

Project

ADT

```java
class Student {
    private int id;
    private String name;
    private double marks; // Assuming marks is a double for more precision

    public Student(int id, String name, double marks) {
        this.id = id;
        this.name = name;
        this.marks = marks;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public double getMarks() {
        return marks;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", marks=" + marks +
                '}';
    }
}
```

Implement the ADT

```java
class Student {
    private int id;
    private String name;
    private double marks; // Assuming marks is a double for more precision

    public Student(int id, String name, double marks) {
        this.id = id;
        this.name = name;
        this.marks = marks;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public double getMarks() {
        return marks;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", marks=" + marks +
                '}';
    }
}
```

Array Stack

```java
import java.util.EmptyStackException;

class ArrayStack {
    private int maxSize;
    private Student[] stackArray;
    private int top;

    public ArrayStack(int size) {
        this.maxSize = size;
        this.stackArray = new Student[maxSize];
        this.top = -1;
    }

    public void push(Student student) {
        if (top == maxSize - 1) {
            System.out.println("Stack is full. Cannot add " + student);
        } else {
            stackArray[++top] = student;
            System.out.println("Added " + student + " to the stack.");
        }
    }

    public Student pop() {
        if (top == -1) {
            System.out.println("Stack is empty. Cannot pop.");
            throw new EmptyStackException();
        } else {
            return stackArray[top--];
        }
    }

    public Student peek() {
        if (top == -1) {
            System.out.println("Stack is empty. Nothing to peek.");
            throw new EmptyStackException();
        } else {
            return stackArray[top];
        }
    }

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

Class Main

```java
import java.util.ArrayList;
import java.util.Scanner;


public class Main {
    private static ArrayStack studentStack;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Nhập số lượng sinh viên cần quản lý
        System.out.println("Enter the number of students in the class: ");
        int numberOfStudents = scanner.nextInt();
        studentStack = new ArrayStack(numberOfStudents);

        while (true) {
            // Menu chính
            System.out.println("\n===== Student Management System =====");
            System.out.println("1. Add Student");
            System.out.println("2. Edit Student");
            System.out.println("3. Delete Student");
            System.out.println("4. Display All Students");
            System.out.println("5. Search Student by ID");
            System.out.println("6. Sort Students by Marks");
            System.out.println("7. Exit");
            System.out.print("Choose an option: ");
            int choice = scanner.nextInt();
            scanner.nextLine(); // Xóa bộ đệm

            switch (choice) {
                case 1:
                    addStudent(scanner);
                    break;
                case 2:
                    editStudent(scanner);
                    break;
                case 3:
                    deleteStudent();
```

Class Main 2

```
                break;
            case 4:
                displayAllStudents();
                break;
            case 5:
                searchStudent(scanner);
                break;
            case 6:
                sortStudentsBubbleSort();
                break;
            case 7:
                System.out.println("Exiting program. Goodbye!");
                return;
            default:
                System.out.println("Invalid option. Please try again.");
        }
    }
}

// Thêm sinh viên vào stack
private static void addStudent(Scanner scanner) {
    System.out.print("Enter Student ID: ");
    int id = scanner.nextInt();
    scanner.nextLine(); // Xóa bộ đệm
    System.out.print("Enter Student Name: ");
    String name = scanner.nextLine();
    System.out.print("Enter Marks: ");
    double marks = scanner.nextDouble();

    Student student = new Student(id, name, marks);
    studentStack.push(student);
}

// Sửa thông tin sinh viên
private static void editStudent(Scanner scanner) {
    System.out.print("Enter the Student ID to edit: ");
    int id = scanner.nextInt();
    scanner.nextLine(); // Xóa bộ đệm
    ArrayList<Student> tempStack = new ArrayList<>();

    boolean found = false;
    while (!studentStack.isEmpty()) {
        Student student = studentStack.pop();
        if (student.getId() == id) {
            found = true;
            System.out.print("Enter new Name: ");
            String newName = scanner.nextLine();
            System.out.print("Enter new Marks: ");
            double newMarks = scanner.nextDouble();
            student = new Student(id, newName, newMarks);
        }
        tempStack.add(student);
    }
```

Class Main 3

```
        // Đẩy lại vào stack
        for (int i = tempStack.size() - 1; i >= 0; i--) {
            studentStack.push(tempStack.get(i));
        }

        if (found) {
            System.out.println("Student updated successfully.");
        } else {
            System.out.println("Student ID not found.");
        }
    }

    // Xóa sinh viên
    private static void deleteStudent() {
        if (studentStack.isEmpty()) {
            System.out.println("Stack is empty. No student to delete.");
        } else {
            Student removedStudent = studentStack.pop();
            System.out.println("Removed: " + removedStudent);
        }
    }

    // Hiển thị tất cả sinh viên
    private static void displayAllStudents() {
        if (studentStack.isEmpty()) {
            System.out.println("No students in the stack.");
            return;
        }

        ArrayList<Student> tempStack = new ArrayList<>();
        while (!studentStack.isEmpty()) {
            Student student = studentStack.pop();
            tempStack.add(student);
            System.out.println(student + " -> Rank: " +
etRanking(student.getMarks()));
        }

        // Đẩy lại vào stack
        for (int i = tempStack.size() - 1; i >= 0; i--) {
            studentStack.push(tempStack.get(i));
        }
    }

    // Tìm kiếm sinh viên theo ID
    private static void searchStudent(Scanner scanner) {
        System.out.print("Enter the Student ID to search: ");
        int id = scanner.nextInt();

        ArrayList<Student> tempStack = new ArrayList<>();
        boolean found = false;
```

Class Main 4

```
        while (!studentStack.isEmpty()) {
            Student student = studentStack.pop();
            tempStack.add(student);

            if (student.getId() == id) {
                System.out.println(student + " -> Rank: " +
etRanking(student.getMarks()));
                found = true;
            }
        }

        // Đẩy lại vào stack
        for (int i = tempStack.size() - 1; i >= 0; i--) {
            studentStack.push(tempStack.get(i));
        }

        if (!found) {
            System.out.println("Student not found.");
        }
    }


// Sắp xếp sinh viên theo điểm sử dụng Bubble Sort
    private static void sortStudentsBubbleSort() {
        ArrayList<Student> tempStack = new ArrayList<>();

        // Lấy tất cả sinh viên từ stack và lưu vào tempStack
        while (!studentStack.isEmpty()) {
            tempStack.add(studentStack.pop());
        }

        // Bubble Sort
        for (int i = 0; i < tempStack.size() - 1; i++) {
            for (int j = 0; j < tempStack.size() - 1 - i; j++) {
                // So sánh điểm của 2 sinh viên
                if (tempStack.get(j).getMarks() < tempStack.get(j + 1).getMarks()) {
                    // Hoán đổi nếu sinh viên hiện tại có điểm thấp hơn sinh viên
tiếp theo
                    Student temp = tempStack.get(j); // Khai báo temp để lưu trữ sinh
viên tạm thời
                    tempStack.set(j, tempStack.get(j + 1)); // Đổi vị trí
                    tempStack.set(j + 1, temp); // Đổi vị trí
                }
            }
        }

        // Đưa lại tất cả sinh viên vào stack sau khi đã sắp xếp
        for (int i = tempStack.size() - 1; i >= 0; i--) {
            studentStack.push(tempStack.get(i));
        }

        System.out.println("Students sorted by marks using Bubble Sort.");
```

Project-review

```
C:\Users\hieunguyen\.jdks\openjdk-23\bin\java.exe "-javaagent:
Enter the number of students in the class:
4

===== Student Management System =====
1. Add Student
2. Edit Student
3. Delete Student
4. Display All Students
5. Search Student by ID
6. Sort Students by Marks
7. Exit
Choose an option: 1
Enter Student ID: 1
Enter Student Name: hieu
Enter Marks: 5
Added Student{id=1, name='hieu', marks=5.0} to the stack.
```

Project-review

```
===== Student Management System =====
1. Add Student
2. Edit Student
3. Delete Student
4. Display All Students
5. Search Student by ID
6. Sort Students by Marks
7. Exit
Choose an option: 2
Enter the Student ID to edit: 1
Enter new Name: Tien
Enter new Marks: 8
Added Student{id=1, name='Tien', marks=8.0} to the stack.
Added Student{id=2, name='Bao', marks=4.0} to the stack.
Added Student{id=3, name='Hieu', marks=9.0} to the stack.
Added Student{id=4, name='Hung', marks=3.0} to the stack.
Added Student{id=5, name='Phong', marks=7.0} to the stack.
Student updated successfully.
```

Project-review

```
===== Student Management System =====
1. Add Student
2. Edit Student
3. Delete Student
4. Display All Students
5. Search Student by ID
6. Sort Students by Marks
7. Exit
Choose an option: 3
Removed: Student{id=3, name='Hieu', marks=9.0}
```

Project-review

```
===== Student Management System =====
1. Add Student
2. Edit Student
3. Delete Student
4. Display All Students
5. Search Student by ID
6. Sort Students by Marks
7. Exit
Choose an option: 4
Student{id=4, name='nguyen', marks=9.0} -> Rank: Excellent
Student{id=3, name='huong', marks=7.0} -> Rank: Good
Student{id=2, name='hung', marks=6.0} -> Rank: Medium
Student{id=1, name='hieu', marks=5.0} -> Rank: Medium
Added Student{id=1, name='hieu', marks=5.0} to the stack.
Added Student{id=2, name='hung', marks=6.0} to the stack.
Added Student{id=3, name='huong', marks=7.0} to the stack.
Added Student{id=4, name='nguyen', marks=9.0} to the stack.
```

Project-review

```
===== Student Management System =====
1. Add Student
2. Edit Student
3. Delete Student
4. Display All Students
5. Search Student by ID
6. Sort Students by Marks
7. Exit
Choose an option: 5
Enter the Student ID to search: 3
Student{id=3, name='huong', marks=7.0} -> Rank: Good
Added Student{id=1, name='hieu', marks=5.0} to the stack.
Added Student{id=2, name='hung', marks=6.0} to the stack.
Added Student{id=3, name='huong', marks=7.0} to the stack.
Added Student{id=4, name='nguyen', marks=9.0} to the stack.
```

Project-review

```
===== Student Management System =====
1. Add Student
2. Edit Student
3. Delete Student
4. Display All Students
5. Search Student by ID
6. Sort Students by Marks
7. Exit
Choose an option: 6
Added Student{id=1, name='hieu', marks=5.0} to the stack.
Added Student{id=2, name='hung', marks=6.0} to the stack.
Added Student{id=3, name='huong', marks=7.0} to the stack.
Added Student{id=4, name='nguyen', marks=9.0} to the stack.
Students sorted by marks using Bubble Sort.
Student{id=4, name='nguyen', marks=9.0} -> Rank: Excellent
Student{id=3, name='huong', marks=7.0} -> Rank: Good
Student{id=2, name='hung', marks=6.0} -> Rank: Medium
Student{id=1, name='hieu', marks=5.0} -> Rank: Medium
```

Project-review

```
===== Student Management System =====
1. Add Student
2. Edit Student
3. Delete Student
4. Display All Students
5. Search Student by ID
6. Sort Students by Marks
7. Exit
Choose an option: 7
Exiting program. Goodbye!
```

Conclusion

Mastering algorithms like Bubble Sort, Merge Sort, and Quick Sort, along with data structures such as Binary Search Trees (BSTs), is essential for optimizing software performance. Additionally, understanding algorithms like Prim's and Dijkstra's underscores the importance of efficiency in network problems. These tools and techniques not only enhance software performance but also improve scalability and maintainability, which are crucial for high-quality software development. Equally important is the ability to create precise design specifications for data structures and algorithms, as demonstrated by examining the stack ADT, FIFO queue, and various sorting algorithms. This knowledge is fundamental for defining valid operations, steps, efficiency, and use cases, ultimately leading to more effective problem-solving and software solutions.

BTEC

Thanks For Watching !