DATA STRUCTURES AND ALGORITHMS

LE QUANG NGUYEN-BH01434

1

A stack ADT, a concrete data structure for a First In First out (FIFO) queue

What is DSA

Data Structures and Algorithms (DSA) is a fundamental part of Computer Science that teaches you how to think and solve complex problems systematically. Using the right data structure and algorithm makes your program run faster, especially when working with lots of data.

What is ADT?

An Abstract Data Type (ADT) is a model for data where only the allowed operations and their effects are defined, without specifying how those operations are implemented. In other words, an ADT tells us what we can do with the data, but not how it's done. This allows the actual details, like data storage or specific algorithms, to be flexible and hidden, focusing instead on the functionality provided.

Different between Stack and Queue

Stack and Queue

Definition

Stack

A data structure that follows Last In, First Out (LIFO) principle. The last item added is the first one removed.

Queue

A data structure that follows First In, First Out (FIFO) principle. The first item added is the first one removed.

Stack and Queue

Main Operations

Stack

- Push: Adds an element to the top of the stack.
- Pop: Removes and returns the top element of the stack.
- Peek (or Top): Returns the top element of the stack without removing it.
- IsEmpty: Checks if the stack is empty.
- Size: Returns the number of elements in the stack.

Queue

- Enqueue: Adds an element to the end (rear) of the queue.
- Dequeue: Removes and returns the front element of the queue.
- Front (or Peek): Returns the front element of the queue without removing it.
- IsEmpty: Checks if the queue is empty.
- Size: Returns the number of elements in the queue.

Stack and Queue

Use cases

Stack

- Function Call Management: The call stack in programming languages keeps track of function calls and returns.
- Expression Evaluation: Used in parsing expressions and evaluating postfix or prefix notations.
- Backtracking: Helps in algorithms that require exploring all possibilities, such as maze solving and depth-first search.

Queue

- Task Scheduling: Operating systems use queues to manage tasks and processes.
- Breadth-First Search (BFS): In graph traversal algorithms, queues help in exploring nodes level by level.
- Buffering: Used in situations where data is transferred asynchronously, such as IO buffers and print spooling.

Stack and Queue

Use cases

Stack

- Function Call Management: The call stack in programming languages keeps track of function calls and returns.
- Expression Evaluation: Used in parsing expressions and evaluating postfix or prefix notations.
- Backtracking: Helps in algorithms that require exploring all possibilities, such as maze solving and depth-first search.

Queue

- Task Scheduling: Operating systems use queues to manage tasks and processes.
- Breadth-First Search (BFS): In graph traversal algorithms, queues help in exploring nodes level by level.
- Buffering: Used in situations where data is transferred asynchronously, such as IO buffers and print spooling.

Stack and Queue

Complexity

Stack

- Push: O(1)
- Pop: O(1)
- Peek: O(1)

Queue

- Enqueue: O(1)
- Dequeue: O(1)
- Front: O(1)
- Rear: O(1)

Stack and Queue

Implementation

Stack

Can be implemented using arrays or linked lists.

Queue

Can be implemented using arrays, linked lists, or circular buffers.

Ways to implement Stack and Queue

Stack

Array-Based Stack

- Fixed-size array: Use a static array with a maximum capacity. This approach has a limited size but is efficient in terms of time complexity.
- Dynamic array (Resizable array): Use a dynamically resizable array

Stack

Linked List-Based Stack

- Implement using a singly or doubly linked list. Each new element is added to the head of the list for constant-time insertion and deletion.

Queue

Array-Based Queue

- Fixed-size circular buffer: Use an array with two pointers for front and rear. When the rear reaches the end, wrap around to the beginning.
- Dynamic array: For flexible size, a resizable array with shifting may be used, though less efficient due to resizing and shifting overhead.

Queue

Linked List-Based Queue

- Implement using a singly or doubly linked list. The front pointer handles dequeue operations, while the rear pointer handles enqueue operations.

2 Two sorting algorithms.

Selection sort

Definition

- Selection Sort is a straightforward comparison-based sorting algorithm. It works by dividing the input list into two segments: a sorted segment at the beginning and an unsorted segment at the end. The algorithm repeatedly selects the smallest (or largest) element from the unsorted segment and swaps it with the first element of that segment, effectively growing the sorted segment until the entire list is sorted. The simplicity of Selection Sort makes it easy to implement, but it is generally inefficient for large datasets.

Bubble sort

Definition

- Bubble Sort is another simple comparison-based sorting algorithm that sorts a list by repeatedly stepping through the list and comparing adjacent elements. If two elements are in the wrong order, they are swapped. This process is repeated until the list is sorted, hence the name "bubble," as larger elements "bubble" to the top of the list. Bubble Sort is intuitive and easy to understand but is also inefficient for large lists due to its repeated comparisons and swaps.

Selection sort

Time Complexity

- Best Case: O(n^2)
- Average Case: O(n^2)
- Worst Case: O(n^2)

Selection Sort always scans the remaining unsorted elements to find the minimum value, leading to a quadratic time complexity in all scenarios, regardless of the initial order of the input list.

Bubble sort

Time Complexity

- Best Case: O(n) (when the list is already sorted)
- Average Case: O(n^2)
- Worst Case: O(n^2)

In the best case, Bubble Sort can achieve linear time complexity if it includes a flag to detect whether any swaps were made during a pass. If no swaps are made, the list is sorted, and the algorithm can terminate early.

Selection sort

Space Complexity

- Space Complexity: O(1)

Selection Sort is an in-place sorting algorithm, requiring only a constant amount of extra space for its operations, regardless of the input size.

Bubble sort

Space Complexity

- Space Complexity: O(1)

Bubble Sort is also an in-place algorithm, using a constant amount of additional space to perform the sorting operations.

Selection sort

Stability

- Not stable in its default implementation because it involves selecting the minimum element and placing it in the current sorted position, which may move equal elements out of their initial order.
- Example: If we have an array [4a, 3, 4b, 2], after selection sorting, we may end up with [2, 3, 4b, 4a], where the original order of 4a and 4b is not preserved.

Bubble sort

Stability

- Stable because adjacent elements are swapped only if they are in the wrong order. This way, equal elements retain their initial order relative to each other.
- Example: For the same array [4a, 3, 4b, 2], bubble sort will yield [2, 3, 4a, 4b], maintaining the original order of 4a and 4b.

Selection sort

Use Cases

Suitable for small datasets or educational purposes due to its simplicity

Bubble sort

Use Cases

Suitable for small datasets or when stability is required, but generally inefficient for larger datasets.

3 Shortest path algorithms

What is Dijkstra algorithms

Dijkstra algorithms is a popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist Edsger W. Dijkstra in 1956.

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

Optimization

Optimized for finding the shortest path between a single source node and all other nodes in a graph with non-negative edge weights

Relaxation

Dijkstra’s algorithm uses a greedy approach where it chooses the node with the smallest distance and updates its neighbors

Time Complexity

Dijkstra’s algorithm has a time complexity of O(V^2) for a dense graph and O(E log V) for a sparse graph, where V is the number of vertices and E is the number of edges in the graph.

Technique

Dijkstra’s algorithm is a single-source shortest path algorithm that uses a greedy approach and calculates the shortest path from the source node to all other nodes in the graph.

Application

Dijkstra’s algorithm is used in many applications such as routing algorithms, GPS navigation systems, and network analysis

Negative Weights

Dijkstra’s algorithm does not work with graphs that have negative edge weights, as it assumes that all edge weights are non-negative.

Student management system

Function

- Add Student: Allows the user to add a new student to the stack.
- Edit Student Marks: Updates the marks of a student with a given ID. If the student is not found, it notifies the user.
- Delete Student: Removes a student with a specified ID from the stack
- Display All Students: Prints all students currently stored in the stack.
- Sort Students by Marks: Allows sorting by marks using a chosen algorithm (Bubble Sort, Quick Sort, or Merge Sort) in ascending or descending order.
- Search Student by ID: Searches for a student in the stack by their ID and displays the result.

Student class

Student

- Represents the student entity.
- Purpose: Stores individual student details like ID, name, and marks.
- Key Methods:
- getId(), getName(), getMarks(), setMarks(): Manage student attributes.
- getRank(): Determines a student's rank based on marks.
- toString(): Provides a formatted string representation of a student.

Node class

Node

- Represents a single node in the stack.
- Purpose: Links a Student object with the next node in the stack.
- Attributes:
  - student: Holds the student object.
  - next: Points to the next node in the stack.

StudentStack class

StudentManagement

- Handles the management operations on the stack, such as adding, editing, deleting, searching, and sorting students.
- Purpose: Acts as the business logic layer.
- Key Methods:
  - Add/Delete/Update:
    - addStudent(Student student): Adds a new student to the stack.
    - editStudent(int id, double newMarks): Updates marks for a student by ID.
    - deleteStudent(int id): Removes a student by ID.
  - Search:
    - searchStudentById(int id): Finds and returns a student by ID.
  -

StudentManagement class

StudentManagement

- Sort:
  - sortStudentsByMarks(int algorithmChoice, boolean ascending): Sorts students by marks using the selected sorting algorithm (Bubble Sort, Quick Sort, or Merge Sort).
- Display:
  - displayAllStudents(): Prints all students.

Main class

Main

- Purpose: Provides the user interface and controls the flow of the program.
- Key Features:
  - Displays a menu of options for the user.
  - Allows the user to interact with the StudentManagement system to manage students.
  - Handles user input and ensures it is validated before processing.

Interface

```
Student Management System
1. Add Student
2. Edit Student Marks
3. Delete Student
4. Display All Students
5. Sort Students by Marks
6. Search Student by ID
7. Exit
```

Add student

```
Enter your choice: 1
Enter ID: 1
Enter Name: Huy
Enter Marks (0 to 10): 11
Marks must be between 0 and 10.
Enter Marks (0 to 10): 10
Student added.
```

```
Enter your choice: 1
Enter ID: 2
Enter Name: nguyen
Enter Marks (0 to 10): 6
Student added.
```

```
Enter your choice: 1
Enter ID: 3
Enter Name: truong
Enter Marks (0 to 10): 9
Student added.
```

Display all student

```
Enter your choice: 4
List of Students:
Student id = 3, name='truong' Marks: 9.0' Rank: Excellent
Student id = 2, name='nguyen' Marks: 6.0' Rank: Medium
Student id = 1, name='Huy' Marks: 10.0' Rank: Excellent
```

Sort student by marks

```
Enter your choice: 5
Choose a sorting algorithm:
1. Bubble Sort
2. Quick Sort
3. Merge Sort
Enter your choice: 1
1. Sort by Ascending Order
2. Sort by Descending Order
```

```
Enter your choice: 1
Students sorted successfully.
Students after sorting:
Student id = 2, name='nguyen' Marks: 6.0' Rank: Medium
Student id = 3, name='truong' Marks: 9.0' Rank: Excellent
Student id = 1, name='Huy' Marks: 10.0' Rank: Excellent
```

Update student

```
Enter your choice: 2
Enter ID of the student to edit: 2
Enter new marks (0 to 10): 5
Student updated.
```

```
Enter your choice: 2
Enter ID of the student to edit: 7
Enter new marks (0 to 10): 1
Student not found.
```

Search student by id

```
Enter your choice: 6
Enter the ID of the student to search: 1
Found: Student id = 1, name='Huy' Marks: 10.0' Rank: Excellent
```

Delete student

```
Enter your choice: 3
Enter ID of the student to delete: 2
Student deleted: Student id = 2, name='nguyen' Marks: 5.0' Rank: Medium
```