



PROGRAM TITLE: Data Structures and Algorithms

UNIT TITLE: DSA

ASSIGNMENT NUMBER: 1

ASSIGNMENT NAME: Data Structures and Algorithms

SUBMISSION DATE: 21/12/2023

DATE RECEIVED: 19/12/2023

TUTORIAL LECTURER: Nguyễn Ngọc Tân

WORD COUNT: 7130

STUDENT NAME: Nguyễn Phúc Nhật Thành

STUDENT ID: BKC 13474

MOBILE NUMBER: 0961581328

|  |
| --- |
| **Summative Feedback:**  **Internal verification:** |

Contents

[I. Examine abstract data types, concrete data structures and algorithms 3](#_Toc154099472)

[1.1 Create a design specification for data structures explaining the valid operations that can be carried out on the structures. 3](#_Toc154099473)

[1.2. Determine the operations of a memory stack and how it is used to implement function calls in a computer 5](#_Toc154099474)

[1.3. Illustrate, with an example, a concrete data structure for a First In First out (FIFO) queue. 6](#_Toc154099475)

[1.4. Compare the performance of two sorting algorithms. 8](#_Toc154099476)

[1.5. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each. 10](#_Toc154099477)

[II. Specify abstract data types and algorithms in a formal notation 11](#_Toc154099478)

[2.1 Using an imperative definition, specify the abstract data type for a software stack 11](#_Toc154099479)

[2.2. Examine the advantages of encapsulation and information hiding when using an ADT 13](#_Toc154099480)

[2.3. Discuss the view that imperative ADTs are a basis for object orientation and, with justification, state whether you agree 14](#_Toc154099481)

[III. Implement complex data structures and algorithms 15](#_Toc154099482)

[3.1. Implement a complex ADT and algorithm in an executable programming language to solve a well-defined problem. 15](#_Toc154099483)

[3.2. Implement error handling and report test results 24](#_Toc154099484)

[3.3. Demonstrate how the implementation of an ADT/algorithm solves a well-defined problem 28](#_Toc154099485)

[3.4. Critically evaluate the complexity of an implemented ADT/algorithm 32](#_Toc154099486)

[IV. Assess the effecttiveness of data structures and algorithms 32](#_Toc154099487)

[4.1. Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm 32](#_Toc154099488)

[4.2. Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example 33](#_Toc154099489)

[4.3. Interpret what a trade-off is when specifying an ADT using an example to support your answer 34](#_Toc154099490)

[4.4. Evaluate three benefits of using implementation independent data structures. 35](#_Toc154099491)

# I. Examine abstract data types, concrete data structures and algorithms

## 1.1 Create a design specification for data structures explaining the valid operations that can be carried out on the structures.

Linked list is a structured database in scientific computing, used to organize and store data in an operational way. Linked lists allow us to add, delete, and access data in an efficient way without limiting the size and number of elements.

- The linked list data structure includes the following components:

Node: Node is the basic component of linked list. Each node contains two parts: data and a reference (or pointer) to the next node in the list (or null object if it's the last node). The data can be of any data type depending on the specific application.

Linked List: A linked list is a collection of nodes linked together in a way that forms a chain. There are two common types of linked lists: Singly linked list and doubly linked list. In a singly linked list, each node has only one link to the next node, while a doubly linked list allows each node to have two links, one to the previous node and one to the next node.

- Some basic operations with linked lists include:

+ Insertion: Add a new node to the linked list, which can be at the beginning of the list, at the end of the list or in any position.

+ Deletion: Delete a node from the linked list, it can be the first node, the last node or a node in any position.

+ Access: Access the data of a node in a linked list by browsing from the beginning to the end of the list.

+ Search: Search for a specific value in a linked list.

- Basic mathematical operations on linked lists help perform operations on adding, deleting, accessing and searching for data in the list. Here are some common operations on linked lists:

Add an element to the list (Insertion):

+ Insert at the Beginning: Creates a new node containing the new data, then updates the new node's reference to point to the first node of the list, making the new node the first node.

+ Insert at the End: Creates a new node containing new data, then iterates from the beginning of the list to the last node and updates the last node's reference to point to the new node.

+ Add at any position (Insert at a Specific Position): Browse from the beginning of the list to the position to insert, then perform the same steps as adding at the beginning of the list.

Delete an element from the list (Deletion):

+ Delete from the Beginning: Updates the reference of the first node in the list to point to the second node, then frees the memory of the deleted node.

+ Delete from the End: Browse from the beginning of the list to the node before the last node, then update the reference of the node before the last node to point to null and free the memory of the last node.

+ Delete node at any position (Delete from a Specific Position): Browse from the beginning of the list to the position to be deleted, then update the reference of the node before the deleted node to point to the node after the deleted node and free the set The node's memory is erased.

Accessing data in the list (Access):

To access data of a node in the list, you browse from the beginning of the list to the node you need to access and retrieve the data value from that node.

Search for data in a list (Search):

To search for a specific value in a list, you browse from the beginning of the list until you find the value or to the end of the list (if not found).

## 1.2. Determine the operations of a memory stack and how it is used to implement function calls in a computer

Stack is a data structure in programming and computer systems used to manage and store information about function calls and local variables in a program. Stack operates according to the "Last In, First Out" (LIFO) mechanism, meaning that the last element put on the stack is the first element removed. In programming, the stack is often used to perform call stack management and to store local variables and return addresses of functions.

Some important points about the memory stack:

+ Function call management (Call Stack): The memory stack helps keep track of function calls in the program. When a function is called, information about that function (including parameters, local variables, and return addresses) is put on the stack. When the function completes, information about that function is removed from the stack.

+ Local Variable Set Management (Local Variables): The stack is also used to store local variable sets declared in a function. This variable can only access the scope of that function and is destroyed at the end of the function.

+ Return address (Return address): The return address of the calling function is usually stored on the stack. After completing this function, the program will return to this return address to continue executing this main program.

+ Memory Segmentation: In some operating systems and computer architectures, the stack memory area is divided separately from the heap memory area (stack and mountain). This helps avoid conflicts and effectively manage memory resources.

+ Automatic memory allocation and release: When you declare a local variable in a function, the memory for this variable is allocated on the stack. When the function ends, this memory is automatically released, helping to avoid memory leaks.

+ Size limit: Stack usually has a fixed size, so it has limited capacity. If you try to use too much memory on the stack (for example, by declaring a large array), it may overflow the stack and cause an error.

+ Managed by the computer system: Stack management is typically performed by the computer system and the compiler/controller. They ensure that additions and deletions of information on the stack are done in the correct order and safely.

Function calls are an important part of computer programming, allowing you to use and reuse source code by calling predefined functions. How to implement function calls in a computer:

+ Function Definition:

The first step is to define the function. You need to specify the function name, return data type (if any), and parameters (if any). This is usually done outside of the function call.

+ Function Call:

When you want to use a function, you call it by the function name and pass the necessary arguments (actual parameters). This will create a function call and send control to the defined function.

+ Function Execution:

The computer then executes the function that was called. During execution, the actual parameters (3 and 5 in the above example) are passed to the function and used to calculate the result.

+ Return Value:

The function can return a result value (if so defined). This result can then be assigned to a variable or used in other mathematical operations.

+ Return to Caller:

After the function has executed and returned results (if any), control returns to the original function call point. The computer continues to execute subsequent statements after the function call.

## 1.3. Illustrate, with an example, a concrete data structure for a First In First out (FIFO) queue.

- Presentation of queues and mathematical operations on queues

The queue is a data structure in the calculator used to store and manage elements in order, similar to queuing at a store checkout. In a queue, the first element added (called "pushing") is the first element removed (called "pushing"). This Multi-function adheres to the "FIFO" (First In - First Out) principle, meaning the element that comes first will be processed first.

Queues are expected to be used in many applications, including setting job schedules, managing network resources, and processing tasks in expected order. This configuration helps ensure that tasks or data are processed in the exact order they arrive.

Queues support some basic mathematical operations to manage data in the queue. Below are the operations commonly performed on queues:

+ Enqueue: Add an element to the end of the queue. This element will become the last element in the queue.

+ Dequeue: Removes the first element from the queue and returns the value of this element. After this operation, the second element will become the first element in the queue.

+ Front (See first element): Returns the value of the first element in the queue without removing it from the queue.

+ Is Empty (Check if the queue is empty): Check if the queue contains any elements. If there are no elements, the queue is considered empty.

+ Size (Size of the queue): Returns the number of elements currently in the queue.

+ Clear: Removes all elements from the queue, making the queue empty.

These operations help manage data in the queue and ensure that elements are processed in the correct order (according to the FIFO principle).

- Implement a concrete example of a queue?

class Queue:  
 def \_\_init\_\_(self):  
 self.items = []  
  
 def enqueue(self, item):  
 self.items.append(item)  
  
 def dequeue(self):  
 if not self.is\_empty():  
 return self.items.pop(0)  
 return None  
  
 def is\_empty(self):  
 return len(self.items) == 0  
  
 def size(self):  
 return len(self.items)  
  
 my\_queue = Queue()  
  
 my\_queue.enqueue(1)  
 my\_queue.enqueue(2)  
 my\_queue.enqueue(3)  
  
 print("Hàng đợi sau khi thêm phần tử: ", my\_queue.items)  
  
 print("Phần tử được lấy ra từ hàng đợi: ", my\_queue.dequeue())  
 print("Hàng đợi sau khi lấy ra một phần tử: ", my\_queue.items)  
  
 print("Hàng đợi có rỗng không? ", my\_queue.is\_empty())  
 print("Kích thước của hàng đợi: ", my\_queue.size())

## 1.4. Compare the performance of two sorting algorithms.

- How to compare the performance of two algorithms?

Comparing the performance of two algorithms is a crucial step in determining which algorithm is more suitable for a specific task. Here are several techniques and considerations to compare the performance of algorithms:

- Time Complexity Analysis:

Analyze the time complexity of both algorithms. Time complexity provides an understanding of how the algorithm's execution time grows concerning the size of the input data.

Compare the Big O notation of both algorithms. The algorithm with the lower Big O notation generally performs better, especially for larger datasets.

- Space Complexity Analysis:

Consider the space complexity of the algorithms. Space complexity refers to the amount of memory space an algorithm uses concerning the input size.

Similar to time complexity, compare the space complexity using Big O notation.

- Empirical Testing:

Implement both algorithms and test them with various input sizes.

Measure the actual execution time and memory usage. This can be done using built-in timers in programming languages or external profiling tools.

Run the algorithms with both small and large datasets to observe their behavior under different conditions.

- Benchmarking:

Use benchmarking tools specific to your programming language. For example, Python has the timeit module, and other languages have similar tools.

Benchmark the algorithms using standardized datasets and measure their execution time.

- Algorithmic Analysis:

Consider the best-case, average-case, and worst-case scenarios for both algorithms.

Evaluate how the algorithms perform in real-world scenarios. Some algorithms might have excellent theoretical performance but might not perform well in practical applications due to constant factors in their execution time.

- Profiling:

Use profiling tools to identify which parts of the algorithms consume the most time and resources.

Optimize the critical sections of the algorithms identified through profiling.

- Consider Special Cases:

Evaluate how algorithms behave with special cases of input data. Some algorithms might perform significantly better or worse with specific types of input.

- Comparative Studies:

Read academic papers and comparative studies about similar algorithms. Researchers often conduct in-depth analyses and provide insights into algorithm performance.

- Real-world Use Cases:

Consider real-world use cases. An algorithm that performs well in a specific application domain might not be the best choice for another domain.

- Community and Expert Opinions:

Engage with the programming community and seek expert opinions. Forums, communities, and discussions can provide valuable insights.

Remember that the choice of the algorithm depends on the specific problem, available resources, and the context of its application. It's essential to consider not only the theoretical performance but also practical aspects and real-world efficiency.

- Compare the performance of two sorting algorithms:

When comparing the performance of two sorting algorithms, we are often interested in two main factors: execution time (time complexity) and memory usage (space complexity). Here are some popular sorting algorithms and a comparison of their performance:

1. Bubble Sort và Insertion Sort:

Time Complexity: Both algorithms have an average and worst-case time complexity of O

Space Complexity: Both of these algorithms only require a constant amount of additional memory, so have a space complexity of O.

So Sánh: Bubble sort and insertion sort are suitable for small or nearly sorted arrays. In these cases, they can be more efficient than other sorting algorithms.

2. Quick Sort và Merge Sort:

Time Complexity: In the average case, quick sort has O(nlogn) complexity and merge sort also has O(nlogn) complexity.

Space Complexity: Quick sort typically requires O(logn) extra memory, while merge sort requires O(n) extra memory.

Comparison: Quick sort is often faster in practice because it requires less memory. However, merge sort is a good choice when it comes to ensuring that execution times are consistent across all data instances.

3. Heap Sort:

Time Complexity: Heap sort has O(nlogn) complexity in the average case and in the worst case.

Space Complexity: Heap sort requires O additional memory.

Comparison: Heap sort is suitable for sorting large amounts of data or when in-place sorting is needed without requiring additional memory.

4. Special Properties:

Stability: A sorting algorithm is stable if it maintains the same position of the same elements in the sorted sequence. Insertion sort and merge sort are stable, while quick sort and heap sort are not necessarily stable.

Adaptability: A sorting algorithm is said to be adaptable if it is more efficient when processing nearly sorted data. Insertion sort is an example of an adaptable algorithm.

When choosing a sorting algorithm, you need to consider the above factors based on the characteristics of the data you are processing and the performance and space requirements. Sometimes, using combination algorithms (e.g. using insertion sort for small elements and quick sort for large elements) can lead to better performance in some situations.

## 1.5. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

Two common algorithms used to find the shortest path in a graph are Dijkstra's Algorithm and the A Algorithm\*. Here's an explanation and specific examples for each algorithm:

Dijkstra's Algorithm:

Dijkstra's Algorithm is used to find the shortest path from a source vertex to all other vertices in a graph with positive edge weights. This algorithm uses a priority queue (usually a priority queue) to explore vertices in order of increasing distance from the source vertex.

Example:



Let's assume the source vertex is A, and we want to find the shortest paths to all other vertices. Dijkstra's Algorithm starts from A and moves to A's neighboring vertices (B and C). Then, it selects the next vertex with the shortest distance (either D or B) and continues this process until all vertices are considered.

A\* Algorithm:

The A\* Algorithm is also used to find the shortest path in a graph but is often used in graph search problems when there's heuristic information available. A\* combines the actual cost (from the start vertex to the current vertex) and an estimation of the remaining cost (heuristic) to the destination.

Example:

Using the same graph, suppose we want to find a path from A to E. A\* uses heuristic information to estimate the remaining cost. In this case, if we know that E is on the far right, A\* will take this information into account when evaluating nodes closer to E.

A\* explores nodes based on priority, considering the total cost from the start node to the current node and the estimated cost from the current node to the destination (denoted as f(n) = g(n) + h(n), where g(n) is the actual cost, and h(n) is the estimated remaining cost).

In this way, A\* examines nodes in the order that minimizes the total expected cost, incorporating both the actual cost and the heuristic estimate of remaining cost to the destination.

Both algorithms have their own advantages and disadvantages and are widely used in practical applications depending on the specific requirements of the problem. Dijkstra's Algorithm ensures finding the shortest path, whereas the A\* Algorithm can be fine-tuned to balance between performance and accuracy based on the heuristic estimate.

# II. Specify abstract data types and algorithms in a formal notation

## 2.1 Using an imperative definition, specify the abstract data type for a software stack

class Stack:  
 def \_\_int\_\_(self):  
 self.items = []  
  
 def is\_empty(self):  
 return len(self.items) == 0  
  
 def push(self, item):  
 self.items.append(item)  
  
 def pop(self):  
 if not self.is\_empty():  
 return self.items.pop()  
 else:  
 return None  
  
 def peek(self):  
 if not self.is\_empty():  
 return self.items[-1]  
 else:  
 return None  
  
 def size(self):  
 return len(self.items)

In this implementation:

\_\_init\_\_: Initializes an empty stack.

is\_empty: Checks if the stack is empty.

push(item): Pushes an item onto the stack.

pop(): Pops and returns the top item from the stack.

peek(): Returns the top item from the stack without removing it.

size(): Returns the number of items in the stack.

- Illustrate how to use this stack with a simple example:

stack = Stack()  
  
 stack.push(1)  
 stack.push(2)  
 stack.push(3)  
  
 print("Top element: ", stack.peek())  
  
 print("Popped element: ", stack.pop())  
 print("Popped element: ", stack.pop())  
  
 print("Is the stack empty?", stack.is\_empty())  
  
 print("Size of the stack: ", stack.size())

When you run this code, it will demonstrate the basic operations of a stack: pushing elements onto the stack, peeking at the top element, popping elements from the stack, checking if the stack is empty, and getting the size of the stack.

## 2.2. Examine the advantages of encapsulation and information hiding when using an ADT

Abstract Data Types (ADTs) play a crucial role in software engineering by providing a high-level description of data and the operations that can be performed on that data. Using ADTs offers several advantages in the context of packaging and information hiding:

+ Encapsulation:

Advantage: ADTs allow bundling data (attributes) and the operations (methods) that manipulate that data into a single unit. This encapsulation hides the internal details of how the data is represented and manipulated.

Benefit: Encapsulation helps in organizing and structuring the code, making it more maintainable, understandable, and modular. It prevents direct access to the internal data, promoting data integrity and security.

+ Modularity:

Advantage: ADTs define clear interfaces for interacting with data structures, specifying the methods and their behavior without revealing the underlying implementation.

Benefit: Modularity enables developers to work on different parts of a system independently. Changes to the implementation of an ADT do not affect the code that uses the ADT, as long as the interface remains consistent. This promotes code reusability and simplifies maintenance.

+ Information Hiding:

Advantage: ADTs allow developers to hide the implementation details of data structures from users of the data types. Users only need to know how to use the provided methods, not how those methods are implemented.

Benefit: Information hiding enhances security and reduces complexity. Users can focus on using the ADTs without being concerned about the internal workings, promoting a clear separation of concerns and minimizing potential errors caused by unintended direct access or modifications.

+ Abstraction:

Advantage: ADTs provide an abstraction layer that simplifies complex data structures into easy-to-understand interfaces, allowing users to work with data without worrying about the complexities of the underlying algorithms.

Benefit: Abstraction simplifies the development process, as programmers can interact with high-level concepts rather than getting bogged down in the intricate details. It enhances productivity and ensures that data structures are used correctly and efficiently.

+ Encourages Proper Usage:

Advantage: By defining clear interfaces and constraints through ADTs, developers are guided in the correct usage of data structures and operations.

Benefit: Proper usage reduces the likelihood of errors and misuse of data structures, leading to more reliable and stable software systems.

In summary, ADTs facilitate a well-organized, modular, and secure approach to software development. By encapsulating data and operations, abstract data types promote information hiding, abstraction, and proper usage, ensuring the development of robust and maintainable software applications.

## 2.3. Discuss the view that imperative ADTs are a basis for object orientation and, with justification, state whether you agree

I agree with the view that Abstract Data Types (ADTs) are commonly used in object-oriented programming (OOP). In fact, ADTs are a fundamental concept in object-oriented design and programming. Object-oriented programming is centered around the idea of encapsulating data and behavior into objects, and ADTs provide a formal way to achieve this encapsulation.

In OOP, classes are used to define ADTs. A class encapsulates data (attributes) and behavior (methods) related to a specific entity or concept. Objects are instances of these classes, and they represent real-world entities in the program. By defining classes and objects, OOP enables the creation and manipulation of ADTs.

The four key principles of OOP—encapsulation, inheritance, polymorphism, and abstraction—align closely with the concept of ADTs:

Encapsulation: Encapsulation in OOP is achieved through classes and objects, which encapsulate data and behavior. ADTs provide the theoretical foundation for encapsulation, allowing the bundling of data and operations.

Inheritance: Inheritance allows one class (subclass) to inherit properties and behavior from another class (superclass). ADTs can be inherited, allowing the creation of specialized ADTs based on existing ones.

Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. ADTs can be polymorphic, allowing objects of different concrete types to be treated uniformly through their common abstract interface.

Abstraction: Abstraction in OOP refers to the process of simplifying complex systems by modeling classes appropriate to the problem and working at the most relevant level of inheritance for a particular aspect of the problem. This concept aligns closely with the notion of ADTs, which provide abstraction through well-defined interfaces and hide implementation details.

In summary, ADTs are a foundational concept in object-oriented programming, enabling the creation of modular, encapsulated, and reusable code. They provide a structured way to define and work with abstract concepts, making them essential in OOP paradigms.

# III. Implement complex data structures and algorithms

## 3.1. Implement a complex ADT and algorithm in an executable programming language to solve a well-defined problem.

Orders must be processed chronologically in an online store, with the earliest orders placed being processed first. Order codes, dates, total amounts, recipient names, phone numbers, receiving addresses, and statuses are all included with each order. Numerous products, each with a product identifier, name, and price, will be included in each order.

The order's processed status will change to processed and the processed order will print on the screen. Until the order list is empty, orders will be handled in order of receipt.

The program may have menus like this:

1. Create invoice

- Interface QueueADT:

public interface QueueADT <T>{  
 void enqueue(T t);  
  
 T denqueue ();  
  
 T peek();  
  
 boolean isEmpty();  
 boolean isFull();  
  
}

- Java QueueADT:

import java.util.ArrayList;  
import java.util.List;  
  
public class QueueADTImpl<T> implements QueueADT<T>{  
  
 private int capacity;  
 private final List<T> item;  
  
 public QueueADTImpl() {  
 item = new ArrayList<>();  
 }  
  
 public QueueADTImpl(int capacity) {  
 this.capacity = capacity;  
 this.item = new ArrayList<>(capacity);  
 }  
  
 public int getCapacity() {  
 return capacity;  
 }  
  
 public void setCapacity(int capacity) {  
 this.capacity = capacity;  
 }  
  
 @Override  
 public void enqueue(T t) {  
 if(!isFull()) {  
 this.item.add(t);  
 }  
 else{  
 throw new RuntimeException("Queue is full");  
 }  
 }  
  
 @Override  
 public T dequeue() {  
 if(!isEmpty()) {  
 T t = item.get(0);  
 item.remove(0);  
 return t;  
 }  
 return null;  
 }  
  
 @Override  
 public T peek() {  
 if(!isEmpty()) {  
 return item.get(0);  
 }  
 return null;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return item.isEmpty();  
 }  
  
 @Override  
 public boolean isFull() {  
 return capacity == item.size();  
 }  
}

- Java Product:

public class Product {  
 private int id;  
 private String name;  
 private long price;  
  
 public Product() {  
 }  
  
 public Product(int id, String name, long price) {  
 this.id = id;  
 this.name = name;  
 this.price = price;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public long getPrice() {  
 return price;  
 }  
  
 public void setPrice(long price) {  
 this.price = price;  
 }  
  
 @Override  
 public String toString() {  
 return "Product{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", price=" + price +  
 '}';  
 }  
}

- Java orderItem:

public class OrderItem {  
 private int id;  
 private Product item;  
 private int quanity;  
  
 public OrderItem(int id){  
 this.id = id;  
 }  
  
 public OrderItem(int id, Product item, int quanity) {  
 this.id = id;  
 this.item = item;  
 this.quanity = quanity;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public Product getItem() {  
 return item;  
 }  
  
 public void setItem(Product item) {  
 this.item = item;  
 }  
  
 public int getQuanity() {  
 return quanity;  
 }  
  
 public void setQuanity(int quanity) {  
 this.quanity = quanity;  
 }  
  
 @Override  
 public String toString() {  
 return "OrderItem{" +  
 "id=" + id +  
 ", item=" + item +  
 ", quanity=" + quanity +  
 '}';  
 }  
}

- Order:

import java.util.ArrayList;  
import java.util.List;  
  
public class Order {  
 private int id;  
 private String orderDate;  
 private long totalAmount;  
 private String customerName;  
 private String customerPhone;  
 private List<OrderItem> orderItems;  
 private boolean status = false;  
  
 public Order(int id) {  
 this.id = id;  
 orderItems = new ArrayList<>();  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getOrderDate() {  
 return orderDate;  
 }  
  
 public void setOrderDate(String orderDate) {  
 this.orderDate = orderDate;  
 }  
  
 public long getTotalAmount() {  
 return totalAmount;  
 }  
  
 public void setTotalAmount(long totalAmount) {  
 this.totalAmount = totalAmount;  
 }  
  
 public String getCustomerName() {  
 return customerName;  
 }  
  
 public void setCustomerName(String customerName) {  
 this.customerName = customerName;  
 }  
  
 public String getCustomerPhone() {  
 return customerPhone;  
 }  
  
 public void setCustomerPhone(String customerPhone) {  
 this.customerPhone = customerPhone;  
 }  
  
 public List<OrderItem> getOrderItems() {  
 return orderItems;  
 }  
  
 public void setOrderItems(List<OrderItem> orderItems) {  
 this.orderItems = orderItems;  
 }  
  
 public boolean isStatus() {  
 return status;  
 }  
  
 public void setStatus(boolean status) {  
 this.status = status;  
 }  
 public void printOrder(){  
 System.*out*.println("Order - " + id);  
 System.*out*.printf("Customer: %s, %s \n", customerName, customerPhone); // Use %s for string formatting  
 if (status) {  
 System.*out*.println("Status: Duyệt thành công");  
 } else {  
 System.*out*.println("Status: Duyệt thất bại");  
 }  
 System.*out*.println("Amount: " + totalAmount);  
 System.*out*.println("Ma SP - Ten SP - Gia - So Luong ");  
  
 for (OrderItem orderItem : orderItems) {  
 System.*out*.printf("%d - %s - %d - %d \n", orderItem.getItem().getId(),  
 orderItem.getItem().getName(), orderItem.getItem().getPrice(), orderItem.getQuantity());  
 }  
 };  
}

- Main:

import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.List;  
import java.util.Scanner;  
  
public class Main {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 QueueADT<Order> orderQueueADT = new QueueADTImpl<>(100);  
 List<Order> orderList = new ArrayList<>();  
 List<Product> availableProducts = new ArrayList<>();  
 availableProducts.add(new Product(1, "Oc Huong", 145000));  
 availableProducts.add(new Product(2, "Oc Mit", 75000));  
 availableProducts.add(new Product(3, "Oc Mong Tay", 99000));  
 availableProducts.add(new Product(4, "Ngao Hoa", 95000));  
 availableProducts.add(new Product(5, "So Huyet", 100000));  
 availableProducts.add(new Product(6, "Chan Ga", 60000));  
 availableProducts.add(new Product(7, "Canh Ga", 25000));  
 availableProducts.add(new Product(8, "Chao Ngao", 120000));  
 availableProducts.add(new Product(9, "Mien Xao", 120000));  
 availableProducts.add(new Product(10, "Lau Oc", 100000));  
 availableProducts.add(new Product(12, "Cut Lon Xao Me", 60000));  
 availableProducts.add(new Product(13, "Nem Chua Ran", 60000));  
 availableProducts.add(new Product(14, "Khoai Tay Chien", 40000));  
 availableProducts.add(new Product(15, "Banh Mi", 4000));  
  
 while (true) {  
 System.*out*.println("--------------------Nha Hang Oc Nam Tu xin chao quy khach--------------------");  
 System.*out*.println("------------------------------Select:------------------------------");  
 System.*out*.println("1. Create Order");  
 System.*out*.println("2. Confirm orders");  
 System.*out*.println("3. Show all orders in the order of increasing price");  
 System.*out*.println("4. Search orders by ID");  
 System.*out*.println("0. Exit");  
 System.*out*.println("----------Enter your selection:----------");  
  
 int choice = scanner.nextInt();  
 scanner.nextLine(); // Consume the newline character  
  
 switch (choice) {  
 case 1:  
 // Tạo hóa đơn  
 Order order = *createOrder*(availableProducts); // Pass the list of available products  
 orderQueueADT.enqueue(order);  
 orderList.add(order);  
 System.*out*.println("Invoice added to queue.");  
 break;  
 case 2:  
 // Duyệt đơn hàng  
 *processOrders*(orderQueueADT);  
 break;  
 case 3:  
 // Hiển thị tất cả các đơn hàng theo thứ tự giá tăng dần  
 *displayOrdersByPrice*(orderList);  
 break;  
 case 4:  
 // Tìm kiếm đơn hàng theo mã  
 System.*out*.println("Enter the order ID to search: ");  
 int orderId = scanner.nextInt();  
 scanner.nextLine(); // Consume the newline character  
 *searchOrderByID*(orderList, orderId);  
 break;  
 case 0:  
 // Thoát  
 System.*out*.println("Exit.");  
 System.*exit*(0);  
 break;  
 default:  
 System.*out*.println("Not a valid choice. Please try again.");  
 }  
 }  
 }  
  
 static Order createOrder(List<Product> availableProducts) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 // Nhập thông tin chung cho đơn hàng  
 System.*out*.println("Enter information for the order:");  
 System.*out*.print("ID orders: ");  
 int orderId = scanner.nextInt();  
 scanner.nextLine(); // Consume the newline character  
 System.*out*.print("Order date: ");  
 String orderDate = scanner.nextLine();  
 System.*out*.print("Customer name: ");  
 String customerName = scanner.nextLine();  
 System.*out*.print("Customer's phone number: ");  
 String customerPhone = scanner.nextLine();  
  
 // Tạo đối tượng Order  
 Order order = new Order(orderId);  
 order.setOrderDate(orderDate);  
 order.setCustomerName(customerName);  
 order.setCustomerPhone(customerPhone);  
  
 // Nhập thông tin cho các sản phẩm trong đơn hàng  
 while (true) {  
 System.*out*.println("--------------------Menu:--------------------");  
 System.*out*.println("Product is in stock:");  
 for (Product product : availableProducts) {  
 System.*out*.println(product.getId() + ". " + product.getName() + " - " + product.getPrice());  
 }  
  
 System.*out*.println("Enter product ID for the order (Enter 0 to finish):");  
 int productId = scanner.nextInt();  
 if (productId == 0) {  
 break; // Người dùng nhập 0 để kết thúc nhập sản phẩm  
 }  
  
 // Find the selected product from the available products  
 Product selectedProduct = null;  
 for (Product product : availableProducts) {  
 if (product.getId() == productId) {  
 selectedProduct = product;  
 break;  
 }  
 }  
  
 if (selectedProduct == null) {  
 System.*out*.println("Invalid product ID. Please try again.");  
 continue;  
 }  
  
 scanner.nextLine(); // Consume the newline character  
  
 System.*out*.print("Amount: ");  
 int quantity = scanner.nextInt();  
  
 // Tạo đối tượng OrderItem  
 OrderItem orderItem = new OrderItem(productId, selectedProduct, quantity);  
  
 // Thêm OrderItem vào Order  
 order.getOrderItems().add(orderItem);  
 }  
  
 // Tính tổng tiền cho đơn hàng  
 long totalAmount = 0;  
 for (OrderItem orderItem : order.getOrderItems()) {  
 totalAmount += orderItem.getItem().getPrice() \* orderItem.getQuantity();  
 }  
 order.setTotalAmount(totalAmount);  
  
 return order;  
 }  
  
  
  
 static void processOrders(QueueADT<Order> orderQueue) {  
 while (!orderQueue.isEmpty()) {  
 Order order = orderQueue.dequeue();  
 if (order != null) {  
 order.setStatus(true);  
 order.printOrder();  
 }  
 }  
 System.*out*.println("All orders in the queue have been processed.");  
 }  
  
 static void displayOrdersByPrice(List<Order> orderList) {  
 // Sắp xếp danh sách đơn hàng theo tổng giá tăng dần  
 Collections.*sort*(orderList, Comparator.*comparingLong*(Order::getTotalAmount));  
  
 // Hiển thị các đơn hàng  
 System.*out*.println("List of orders in ascending price order:");  
 for (Order order : orderList) {  
 order.printOrder();  
 }  
 }  
  
 static void searchOrderByID(List<Order> orderList, int orderId) {  
 for (Order order : orderList) {  
 if (order.getId() == orderId) {  
 System.*out*.println("Order found:");  
 order.printOrder();  
 return;  
 }  
 }  
 System.*out*.println("No orders with ID found " + orderId);  
 }  
}

2. Browse orders

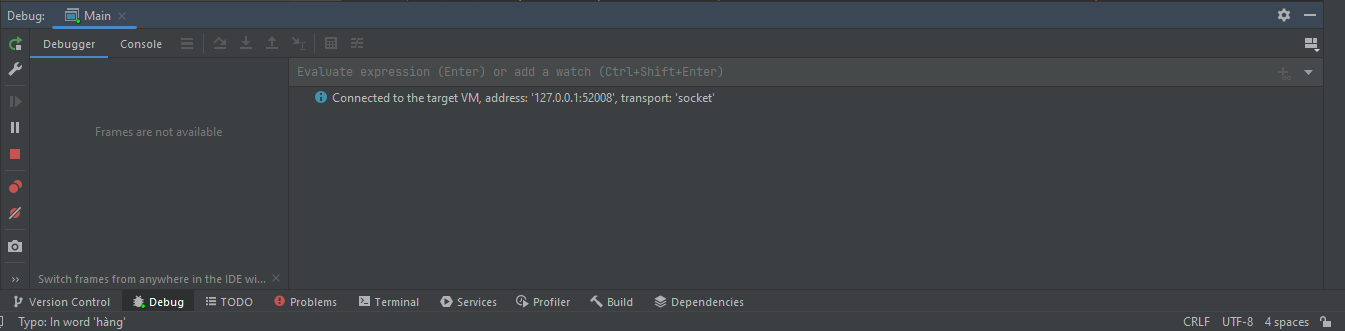
3. Display all orders in ascending price order.

4. Search orders by code

0. Exit

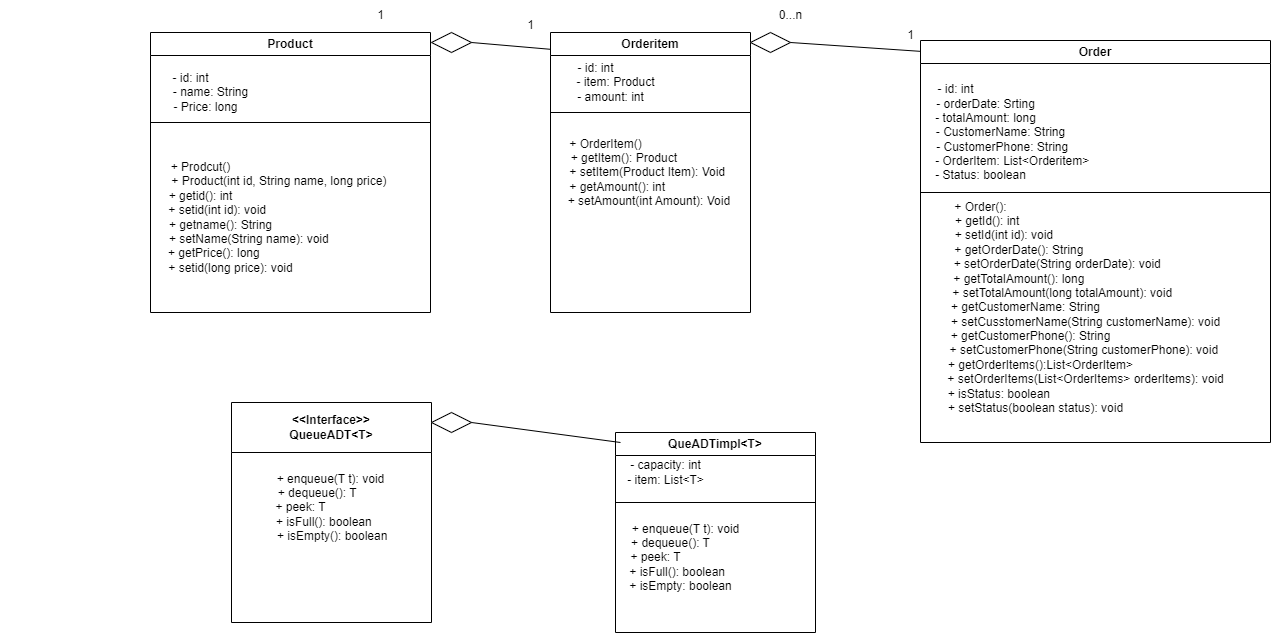
## 3.2. Implement error handling and report test results

import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.List;  
import java.util.Scanner;  
  
public class Main {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 QueueADT<Order> orderQueueADT = new QueueADTImpl<>(100);  
 List<Order> orderList = new ArrayList<>();  
 List<Product> availableProducts = new ArrayList<>();  
 availableProducts.add(new Product(1, "Oc Huong", 145000));  
 availableProducts.add(new Product(2, "Oc Mit", 75000));  
 availableProducts.add(new Product(3, "Oc Mong Tay", 99000));  
 availableProducts.add(new Product(4, "Ngao Hoa", 95000));  
 availableProducts.add(new Product(5, "So Huyet", 100000));  
 availableProducts.add(new Product(6, "Chan Ga", 60000));  
 availableProducts.add(new Product(7, "Canh Ga", 25000));  
 availableProducts.add(new Product(8, "Chao Ngao", 120000));  
 availableProducts.add(new Product(9, "Mien Xao", 120000));  
 availableProducts.add(new Product(10, "Lau Oc", 100000));  
 availableProducts.add(new Product(12, "Cut Lon Xao Me", 60000));  
 availableProducts.add(new Product(13, "Nem Chua Ran", 60000));  
 availableProducts.add(new Product(14, "Khoai Tay Chien", 40000));  
 availableProducts.add(new Product(15, "Banh Mi", 4000));  
  
 while (true) {  
 System.*out*.println("--------------------Nha Hang Oc Nam Tu xin chao quy khach--------------------");  
 System.*out*.println("------------------------------Select:------------------------------");  
 System.*out*.println("1. Create Order");  
 System.*out*.println("2. Confirm orders");  
 System.*out*.println("3. Show all orders in the order of increasing price");  
 System.*out*.println("4. Search orders by ID");  
 System.*out*.println("0. Exit");  
 System.*out*.println("----------Enter your selection:----------");  
  
 int choice = scanner.nextInt();  
 scanner.nextLine(); // Consume the newline character  
  
 switch (choice) {  
 case 1:  
 // Tạo hóa đơn  
 Order order = *createOrder*(availableProducts); // Pass the list of available products  
 orderQueueADT.enqueue(order);  
 orderList.add(order);  
 System.*out*.println("Invoice added to queue.");  
 break;  
 case 2:  
 // Duyệt đơn hàng  
 *processOrders*(orderQueueADT);  
 break;  
 case 3:  
 // Hiển thị tất cả các đơn hàng theo thứ tự giá tăng dần  
 *displayOrdersByPrice*(orderList);  
 break;  
 case 4:  
 // Tìm kiếm đơn hàng theo mã  
 System.*out*.println("Enter the order ID to search: ");  
 int orderId = scanner.nextInt();  
 scanner.nextLine(); // Consume the newline character  
 *searchOrderByID*(orderList, orderId);  
 break;  
 case 0:  
 // Thoát  
 System.*out*.println("Exit.");  
 System.*exit*(0);  
 break;  
 default:  
 System.*out*.println("Not a valid choice. Please try again.");  
 }  
 }  
 }  
  
 static Order createOrder(List<Product> availableProducts) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 // Nhập thông tin chung cho đơn hàng  
 System.*out*.println("Enter information for the order:");  
 System.*out*.print("ID orders: ");  
 int orderId = scanner.nextInt();  
 scanner.nextLine(); // Consume the newline character  
 System.*out*.print("Order date: ");  
 String orderDate = scanner.nextLine();  
 System.*out*.print("Customer name: ");  
 String customerName = scanner.nextLine();  
 System.*out*.print("Customer's phone number: ");  
 String customerPhone = scanner.nextLine();  
  
 // Tạo đối tượng Order  
 Order order = new Order(orderId);  
 order.setOrderDate(orderDate);  
 order.setCustomerName(customerName);  
 order.setCustomerPhone(customerPhone);  
  
 // Nhập thông tin cho các sản phẩm trong đơn hàng  
 while (true) {  
 System.*out*.println("--------------------Menu:--------------------");  
 System.*out*.println("Product is in stock:");  
 for (Product product : availableProducts) {  
 System.*out*.println(product.getId() + ". " + product.getName() + " - " + product.getPrice());  
 }  
  
 System.*out*.println("Enter product ID for the order (Enter 0 to finish):");  
 int productId = scanner.nextInt();  
 if (productId == 0) {  
 break; // Người dùng nhập 0 để kết thúc nhập sản phẩm  
 }  
  
 // Find the selected product from the available products  
 Product selectedProduct = null;  
 for (Product product : availableProducts) {  
 if (product.getId() == productId) {  
 selectedProduct = product;  
 break;  
 }  
 }  
  
 if (selectedProduct == null) {  
 System.*out*.println("Invalid product ID. Please try again.");  
 continue;  
 }  
  
 scanner.nextLine(); // Consume the newline character  
  
 System.*out*.print("Amount: ");  
 int quantity = scanner.nextInt();  
  
 // Tạo đối tượng OrderItem  
 OrderItem orderItem = new OrderItem(productId, selectedProduct, quantity);  
  
 // Thêm OrderItem vào Order  
 order.getOrderItems().add(orderItem);  
 }  
  
 // Tính tổng tiền cho đơn hàng  
 long totalAmount = 0;  
 for (OrderItem orderItem : order.getOrderItems()) {  
 totalAmount += orderItem.getItem().getPrice() \* orderItem.getQuantity();  
 }  
 order.setTotalAmount(totalAmount);  
  
 return order;  
 }  
  
  
  
 static void processOrders(QueueADT<Order> orderQueue) {  
 while (!orderQueue.isEmpty()) {  
 Order order = orderQueue.dequeue();  
 if (order != null) {  
 order.setStatus(true);  
 order.printOrder();  
 }  
 }  
 System.*out*.println("All orders in the queue have been processed.");  
 }  
  
 static void displayOrdersByPrice(List<Order> orderList) {  
 // Sắp xếp danh sách đơn hàng theo tổng giá tăng dần  
 Collections.*sort*(orderList, Comparator.*comparingLong*(Order::getTotalAmount));  
  
 // Hiển thị các đơn hàng  
 System.*out*.println("List of orders in ascending price order:");  
 for (Order order : orderList) {  
 order.printOrder();  
 }  
 }  
  
 static void searchOrderByID(List<Order> orderList, int orderId) {  
 for (Order order : orderList) {  
 if (order.getId() == orderId) {  
 System.*out*.println("Order found:");  
 order.printOrder();  
 return;  
 }  
 }  
 System.*out*.println("No orders with ID found " + orderId);  
 }  
}



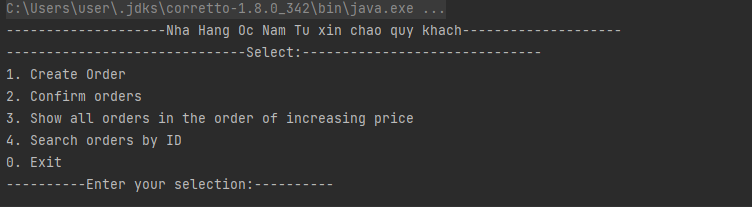
## 3.3. Demonstrate how the implementation of an ADT/algorithm solves a well-defined problem

Class diagram:

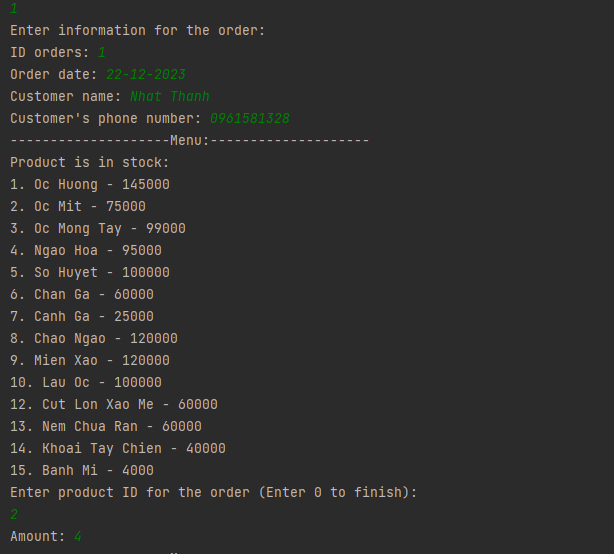


Result:

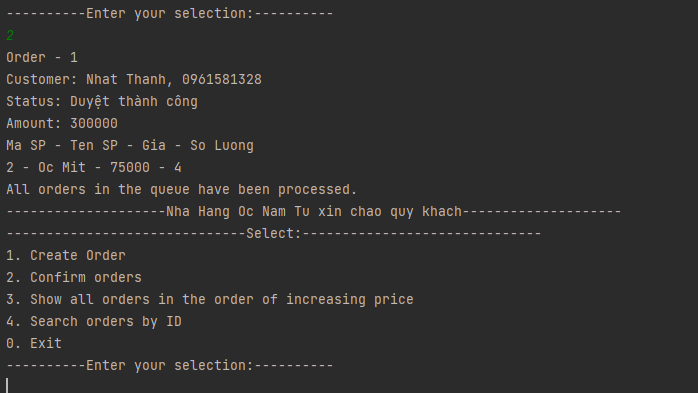
Menu:



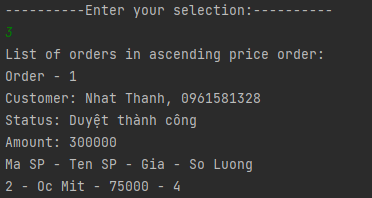
Create Order:



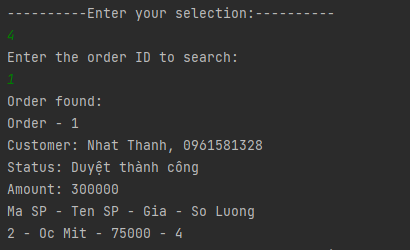
Confirm order:

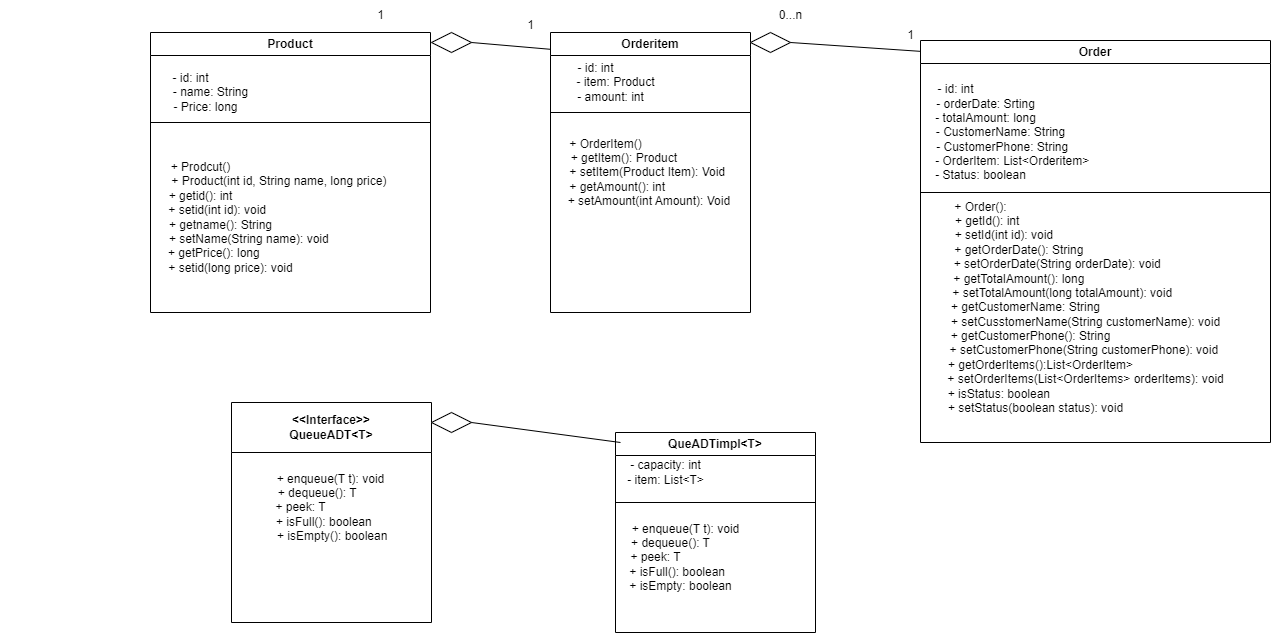


Show list order:



Search ID:





## 3.4. Critically evaluate the complexity of an implemented ADT/algorithm

Performance: Performance is good, but in some cases, some theoretically more efficient algorithms become less practical due to constant factors or implementation details.

# IV. Assess the effecttiveness of data structures and algorithms

## 4.1. Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm

Asymptotic analysis is a method of analyzing the performance of an algorithm when the input approaches infinity. Typically, we use Big O notation to represent the worst-case time and space complexity of an algorithm.

However, when you want to evaluate the effectiveness of an algorithm on different computers, you need to consider many different factors, not just algorithm complexity. Here are some ways to evaluate performance and effectiveness between algorithms on different computers:

Empirical Experiment:

Run algorithms on specific computers and measure execution time and memory usage.

Compare experimental results between algorithms and computers.

Benchmarking:

Use benchmark tools to test performance on many different computers.

Consider using frameworks like JMH (Java Microbenchmarking Harness) for Java, pytest-benchmark for Python, and similar tools for other languages.

Resource Measurement:

Evaluate the memory and CPU usage of the algorithm on different computers.

Use tools like Valgrind, perf (Linux), Instruments (macOS), or Task Manager (Windows) to monitor resources.

Standard Comparison:

Compare the performance of the algorithm with algorithms that have been proven effective or used in similar real-world applications.

Discussion and Quality Assessment:

Discuss with the computer science community and experts in the specific field to evaluate the effectiveness of the algorithm.

## 4.2. Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example

The efficiency of an algorithm can be measured in various ways, but two common metrics are time complexity and space complexity.

Time Complexity:

Time complexity represents the amount of time an algorithm takes to complete as a function of the size of the input.

It is often expressed using Big O notation to describe the upper bound of the growth rate concerning the input size.

Example: Let's consider the time complexity of a simple linear search algorithm. In the worst-case scenario, where the target element is at the end of the list or not present, the time complexity is O(n), where n is the size of the input list.

def linear\_search(arr, target):

for i in range(len(arr)):

if arr[i] == target:

return i

return -1

Space Complexity:

Space complexity measures the amount of memory (space) an algorithm uses as a function of the input size.

Similar to time complexity, it is often expressed using Big O notation.

Example: Consider a function that generates Fibonacci numbers using a recursive approach. The space complexity of this algorithm is O(n) because it requires space for each function call in the call stack.

def fibonacci\_recursive(n):

if n <= 1:

return n

else:

return fibonacci\_recursive(n-1) + fibonacci\_recursive(n-2)

## 4.3. Interpret what a trade-off is when specifying an ADT using an example to support your answer

In the context of specifying an Abstract Data Type (ADT), a trade-off refers to a situation where improving one aspect of the ADT may result in a degradation of another aspect. Designing an ADT often involves making decisions about how to balance competing factors such as simplicity, performance, and flexibility.

Let's illustrate this with an example of a data structure: a List ADT.

Array-based List:

Advantage: Constant-time access to elements using indices (e.g., list[3]).

Trade-off: Insertion or deletion of elements at arbitrary positions may require shifting all subsequent elements.

class ArrayList:

def \_\_init\_\_(self):

self.array = []

def get(self, index):

return self.array[index]

def insert(self, index, element):

self.array.insert(index, element)

def delete(self, index):

del self.array[index]

Linked List:

Advantage: Efficient insertion and deletion of elements at arbitrary positions without shifting.

Trade-off: Access time to elements by index is linear, requiring traversal from the beginning or end.

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class LinkedList:

def \_\_init\_\_(self):

self.head = None

def get(self, index):

current = self.head

for \_ in range(index):

current = current.next

return current.data

def insert(self, index, element):

new\_node = Node(element)

if index == 0:

new\_node.next = self.head

self.head = new\_node

else:

current = self.head

for \_ in range(index - 1):

current = current.next

new\_node.next = current.next

current.next = new\_node

def delete(self, index):

if index == 0:

self.head = self.head.next

else:

current = self.head

for \_ in range(index - 1):

current = current.next

current.next = current.next.next

## 4.4. Evaluate three benefits of using implementation independent data structures.

Implementation-independent data structures provide several benefits that contribute to the flexibility, portability, and maintainability of software systems. Here are three key benefits:

Abstraction and Encapsulation:

Abstraction: Implementation-independent data structures allow developers to work with abstract representations of data, focusing on the logical structure and operations rather than the low-level details.

Encapsulation: The internal details of how the data structure is implemented are hidden from the rest of the program, providing a clear interface for interacting with the data structure. This promotes modularity and reduces dependencies between different parts of the code.

Code Reusability and Portability:

Code Reusability: Implementation-independent data structures can be designed as reusable components. Once defined, they can be easily incorporated into different projects and applications without modification, promoting code reusability.

Portability: Since the interface and functionality of the data structure are independent of the underlying implementation, the code can be easily ported to different platforms or programming languages. This is particularly valuable in multi-platform or cross-language development environments.

Adaptability to Changing Requirements:

Flexibility: By separating the logical structure from the implementation details, implementation-independent data structures provide flexibility to adapt to changing requirements. If the requirements evolve or if a more efficient implementation becomes available, the data structure's interface can remain unchanged while the underlying implementation is modified or replaced.

Scalability: Implementation-independent data structures can be designed to handle a wide range of use cases and scale with changing data sizes. Developers can choose or modify implementations based on the specific needs of the application without affecting the external usage of the data structure.

Ease of Maintenance and Debugging:

Maintenance: Separating the interface from the implementation simplifies maintenance tasks. Developers can update or fix issues in the implementation without affecting the rest of the codebase, reducing the risk of introducing unintended side effects.

Debugging: Implementation-independent data structures can enhance the debugging process. Developers can focus on the specific data structure operations and logic without being concerned about intricate implementation details, making it easier to identify and resolve issues.