Data structures

By: Nguyen Trong Nghia

Table of Contents

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

Identify the Data Structures

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. Different basic and advanced types of data structures are used in almost every program or software system that has been developed. So we must have good knowledge of data structures.

Data structures are an integral part of computers used for the arrangement of data in memory. They are essential and responsible for organizing, processing, accessing, and storing data efficiently. But this is not all.

01

Linear and Nonlinear.

Linear structures arrange data in a linear sequence, such as found in an array, list, or queue.

02

Static and Dynamic.

As the term implies, static structures consist of fixed, permanent structures and sizes at compile time. The array reserves a set amount of reserve memory set up by the programmer ahead of time.

03

Homogenous and Non-Homogenous

Homogenous data structures consist of the same data element type, like element collections found in an array. In non-homogenous structures, the data don't have to be the same type, such as structures

ADT

Definiton

A Stack Abstract Data Type (ADT) is a collection of elements that supports two main operations: push (adding an element to the top of the stack) and pop (removing the top element from the stack). The Stack ADT follows the Last-In-First-Out (LIFO) principle, where the last element added to the stack is the first one to be removed.

Benefits of ADTs

Abstraction

ADTs help hide implementation details and only expose the operations that can be performed on the data. This allows developers to focus on using the ADT without worrying about how it is implemented.

Code Reusability

ADTs are designed to solve specific problems, making them easily reusable in various applications. For example, an ADT for a linked list can be used in multiple scenarios without changing the code.

Code Reusability

When using ADTs, changes to the implementation of the ADT do not affect other parts of the code that use the ADT. This makes it easier to maintain and upgrade the system.

Modularity and Independence

ADTs promote modularity in software design, where each component performs a specific task and can be developed, tested, and maintained independently from other components.

Stack ADT

Definiton

A collection of elements with two principal operations: push (add an element) and pop (remove the most recently added element).

Operations Supported by Stack ADT

push(x): Add element x to the top of the stack.

pop(): Remove and return the top element of the stack. Throws an exception if the stack is empty.

peek(): Return (but do not remove) the top element of the stack. Throws an exception if the stack is empty.

isEmpty(): Return true if the stack is empty, otherwise false.

Example of “push” function

```
// Push an element onto the stack
public void push(T data) {
    Node<T> newNode = new Node<>(data);
    newNode.next = top;
    top = newNode;
    size++;
}
```

```
// Pop an element from the stack
public T pop() {
    if (isEmpty()) {
        throw new EmptyStackException();
    }
    T data = top.data;
    top = top.next;
    size--;
    return data;
```

Example of “pop” function

Example of Emty function

```java
// Check if the stack is empty
public boolean isEmpty() {
    return top == null;
}
```

```java
// Get the number of elements in the stack
public int size() {
    return size;
}
```

Example of Size() function

Specify Input Parameters(2 Types)

Primitive Data Types:

int, char, double, boolean, etc.

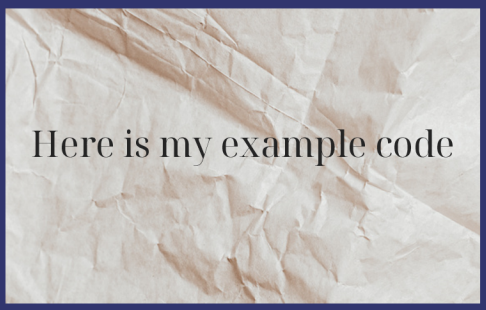
Reference Data Types

String, Arrays, Objects, etc.

Primitive Data Types

```java
public class PrimitiveTypeExample {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
         int number = 10;
        System.out.println("Before method call: " + number);
        changePrimitive(number);
        System.out.println("After method call: " + number);
    }

    public static void changePrimitive(int num) {
        num = 20;
        System.out.println("Inside method: " + num);
    }
    }
```

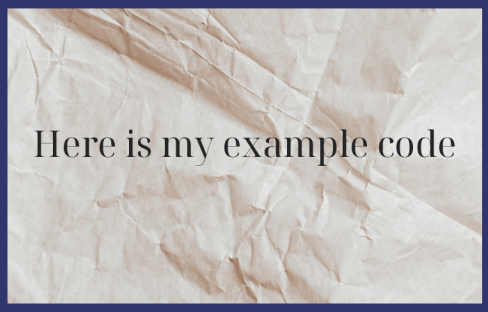
```
Before method call: 10
Inside method: 20
After method call: 10
BUILD SUCCESSFUL (total time: 0 seconds)
```



Reference Data Type Parameters

```java
public class ReferenceDataTypeParameters {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        int[] numbers = {10, 20, 30};
        System.out.println("Before method call: " + numbers[0]);
        changeReference(numbers);
        System.out.println("After method call: " + numbers[0]);
    }

    public static void changeReference(int[] nums) {
        nums[0] = 40;
        System.out.println("Inside method: " + nums[0]);
    }
}
```

```
utput - Reference Data Type Parameters (run) ×
run:
Before method call: 10
Inside method: 40
After method call: 40
BUILD SUCCESSFUL (total time: 0 seconds)
```



Pre-conditions may include expectations about the arguments (except for type, which the compiler guarantees).

Pre-conditions

A pre-condition to a function is a condition that must be true before entering the function—no matter what.

Post-conditions and loop invariants should depend only on whether the function is implemented correctly.

Post-conditions

A post-condition to a function is a condition that must be true before leaving the function—no matter what.

Example

```
void write_sqrt( double x)

//  Precondition:  x >= 0.
//  Postcondition:  The square root of x has
//  been written to the standard output.

. . .
```

This is an example of a small function which simply writes the square root of a number. The number is given as a parameter to the function, called x. For example, if we call write_sqrt(9), then we would expect the function to print 3 (which is the square root of 9

What needs to be true in order for this function to successfully carry out
its work? Since negative numbers don't have a square root, we need to
ensure that the argument, x, is not negative. This requirement is expressed in the precondition:
Precondition: x >= 0.
The postcondition is simply a statement expressing what work has been
accomplished by the function. This work might involve reading or writing data, changing the values of variable parameters, or other actions.
Notice that the information shown on this slide is enough for you to use
the function. You don't need to know what occurs in the function body.

Time Complexity

Time Complexity is a concept in computer science that describes the amount of time an algorithm takes to run as a function of the length of the input. It's a crucial aspect of algorithm analysis as it helps understand how efficiently an algorithm performs, particularly as the size of the input data increases.

Space Complexity

Space Complexcity is a term in computer science used to describe the amount of memory space required by an algorithm to run as a function of the length of the input. It is an important metric for understanding how efficient an algorithm is in terms of memory usage, especially in environments where memory resources are limited.

Types of Space Complexity:

Constant Space (O(1)

The algorithm uses a fixed amount of memory space regardless of the input size. For example, an algorithm that swaps two numbers.

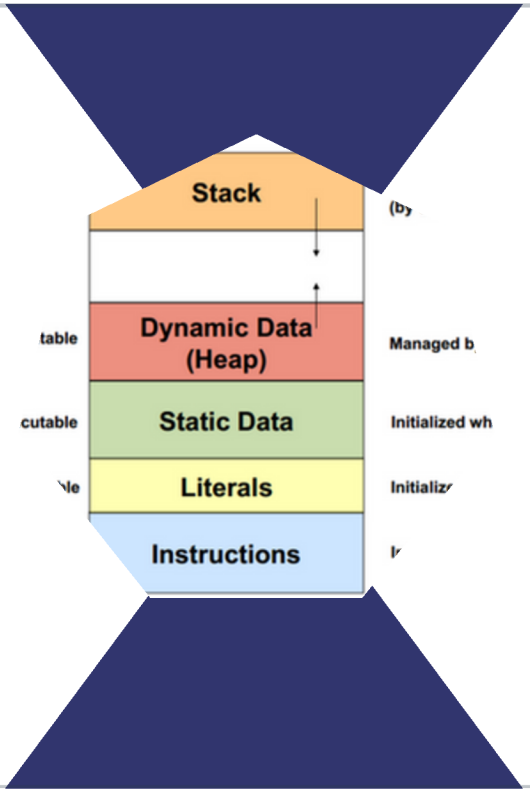
Linear Space (O(n))

The memory required grows linearly with the input size. An example is creating a list of 'n' elements.

Quadratic Space (O(n^2))

The space requirement grows quadratically with the input size, commonly seen in algorithms that create two-dimensional arrays based on the input size.

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



Definiton of Memory Stacks

Stack Memory in Java is used for static memory allocation and the execution of a thread. It contains primitive values that are specific to a method and references to objects referred from the method that are in a heap.
Access to this memory is in Last-In-First-Out (LIFO) order. Whenever we call a new method, a new block is created on top of the stack which contains values specific to that method, like primitive variables and references to objects.
When the method finishes execution, its corresponding stack frame is flushed, the flow goes back to the calling method, and space becomes available for the next method.

They follow the Last In, First Out (LIFO) principle, meaning the last item added to the stack is the first one to be removed. Here are the primary operations associated with memory stacks:

- Push: This operation adds an item to the top of the stack.
- Pop: This operation removes the item from the top of the stack.
- Peek or Top: This operation returns the item currently at the top of the stack without removing it.
- IsEmpty: This operation checks if the stack is empty.
- Size: This operation returns the number of items currently in the stack.

```java
public static void main(String[] args) {

    // TODO code application logic here
    Stack<Integer> stack = new Stack<>();

    // Pushing elements onto the stack
    stack.push(10);
    stack.push(20);
    stack.push(30);


       // Lấy phần tử từ đỉnh của stack và loại bỏ nó (pop)
    int phanTuPop = stack.pop();
    System.out.println("Phần tử pop ra: " + phanTuPop);

    // Peeking at the top element of the stack   // Xem phần tử đầu của stack mà không loại bỏ nó (peek)
    int topElement = stack.peek();
    System.out.println("Top element: " + topElement);

    // Checking if the stack is empty  // Xem phần tử đầu của stack mà không loại bỏ nó (peek)
    boolean isEmpty = stack.isEmpty();
    System.out.println("Is stack empty? " + isEmpty);

    // Getting the size of the stack
    int size = stack.size();
    System.out.println("Size of stack: " + size);
```

stackexample.StackExample > main > phanTuPop >

Output - StackExample (run) ×

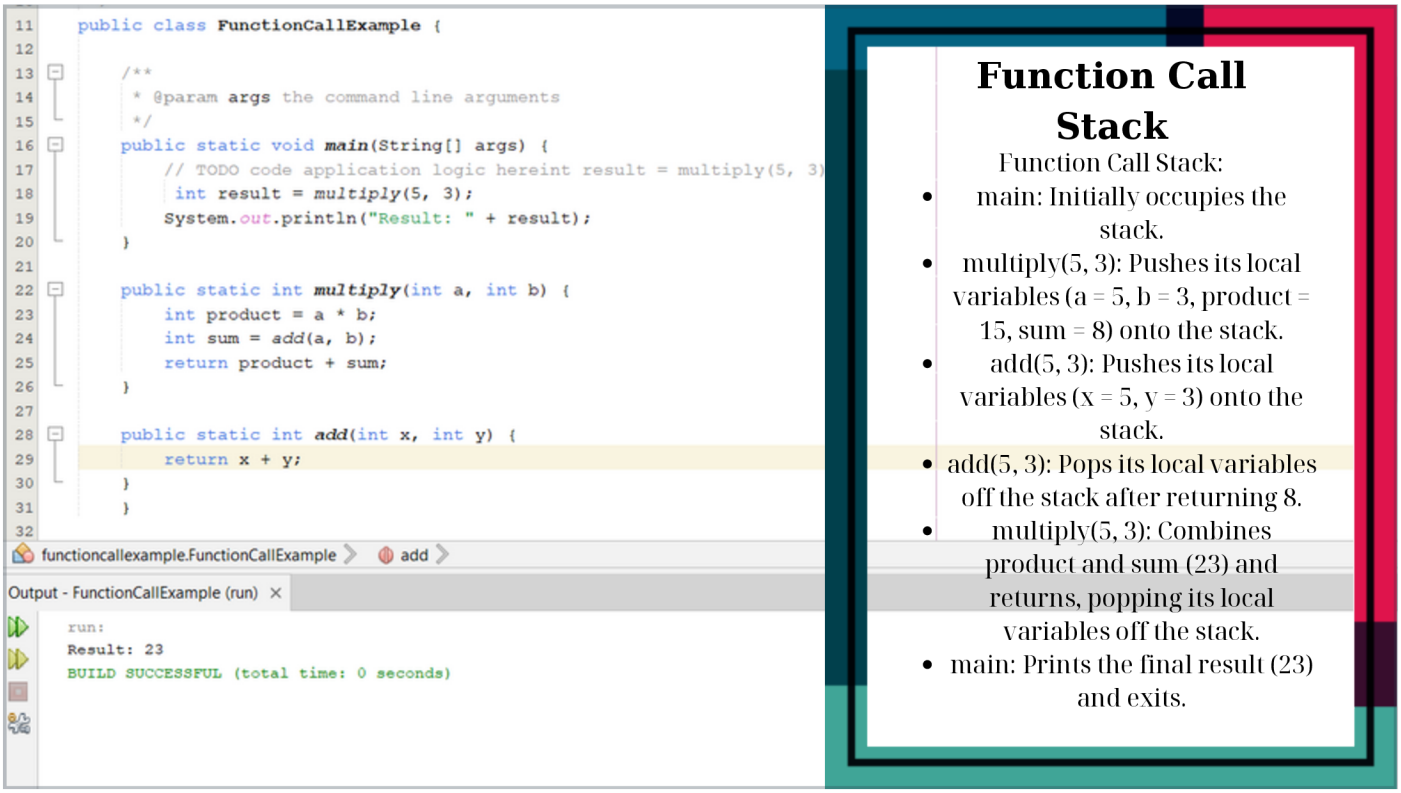
```
run:
Phần tử pop ra: 30
Top element: 20
Is stack empty? false
Size of stack: 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Push: The push method adds elements to the top of the stack (stack.push(10) adds 10, stack.push(20) adds 20, and so on).
- Pop: The pop method removes and returns the element at the top of the stack (stack.pop() removes and returns 30 in this case).
- Peek or Top: The peek method returns the element at the top of the stack without removing it (stack.peek() returns 20 without removing it).
- IsEmpty: The isEmpty method checks if the stack is empty (stack.isEmpty() returns false if there are elements in the stack).
- Size: The size method returns the number of elements in the stack (stack.size() returns 2 after pushing 10, 20, and 30 onto the stack)

Here is my OutPut

FUNCTION CALL IMPLEMENTATION

In Java, function calls are managed using the call stack, which is a specific use case of the stack data structure. When a function is called, its execution context (local variables, parameters, return address, etc.) is pushed onto the stack. When the function completes, its context is popped off the stack, allowing the program to return to the previous execution point


public class FunctionCallExample {
/**
* @param args the command line arguments
*/
public static void main(String[] args) {
// TODO code application logic hereint result = multiply(5, 3)
int result = multiply(5, 3);
System.out.println("Result: " + result);
}
public static int multiply(int a, int b) {
int product = a * b;
int sum = add(a, b);
return product + sum;
}
public static int add(int x, int y) {
return x + y;
}
}
functioncallexample.FunctionCallExample
add
Output - FunctionCallExample (run)
run:
Result: 23
BUILD SUCCESSFUL (total time: 0 seconds)
Function Call Stack
Function Call Stack:
main: Initially occupies the stack.
multiply(5, 3): Pushes its local variables (a = 5, b = 3, product = 15, sum = 8) onto the stack.
add(5, 3): Pushes its local variables (x = 5, y = 3) onto the stack.
add(5, 3): Pops its local variables off the stack after returning 8.
multiply(5, 3): Combines product and sum (23) and returns, popping its local variables off the stack.
main: Prints the final result (23) and exits.

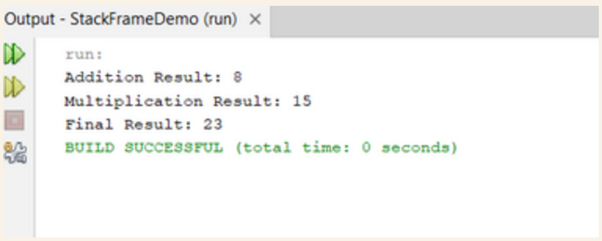
Stack Frames

Stack is one of the segments of application memory that is used to store the local variables, function calls of the function. Whenever there is a function call in our program the memory to the local variables and other function calls or subroutines get stored in the stack frame. Each function gets its own stack frame in the stack segment of the application's memory.

Features

- The memory allocated for a function call in the stack lives only the time when the function is executing once the function gets completed we can't access the variables of that function.
- Once the calling function completes its execution its stack frame is removed and the thread of execution of called function resumes from that position where it was left.
- Stack is used for storing function calls so in the case when we are using a lot of recursive calls in our program the stack memory gets exhausted by the function calls or subroutines which may result in stack overflow because the stack memory is limited.
- Each stack frame maintains the Stack Pointer (SP), and the frame pointer (FP). Stack pointer and frame pointer always point to the top of the stack. It also maintains a program counter (PC) which points to the next instruction to be executed.

```java
public class StackFrameDemo {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        int result = calculate(5, 3);
        System.out.println("Final Result: " + result);
    }

    public static int calculate(int a, int b) {
        int sum = add(a, b);
        int product = multiply(a, b);
        return sum + product;
    }

    public static int add(int x, int y) {
        int sum = x + y;
        System.out.println("Addition Result: " + sum);
        return sum;
    }
```



Stack Frames:

- main: Initially occupies the stack.
- calculate(5, 3): Creates a new stack frame with local variables (a = 5, b = 3).
  - Calls add(5, 3) and multiply(5, 3).
- add(5, 3): Creates a new stack frame with local variables (x = 5, y = 3).
  - Computes sum = 8 and returns 8.
- multiply(5, 3): Creates a new stack frame with local variables (x = 5, y = 3).
  - Computes product = 15 and returns 15.
- calculate(5, 3): Combines sum = 8 and product = 15, returns 23.
- main: Prints "Final Result: 23".

Key Points:

- Each method invocation creates a new stack frame.
- Stack frames hold local variables, parameters, and return addresses.
- When a method completes, its stack frame is popped off the stack.

The Importance

Memory Management: Stack frames play a crucial role in efficient memory management. They ensure that each method call receives its own set of local variables and parameters without interference from other method calls. This segregation helps in organizing and managing memory allocation effectively, especially in environments where memory is limited or needs to be optimized.

Function Invocation and Control Flow: Stack frames facilitate the correct sequencing of function calls and their respective returns. When a method is invoked, its stack frame is pushed onto the call stack, allowing the program to remember where to return after the method completes its execution. This mechanism supports structured programming paradigms and ensures that program control flow follows a predictable path.

The Importance

Recursive Algorithms: Stack frames are essential for implementing recursive algorithms. Each recursive call creates a new stack frame, allowing the algorithm to maintain separate instances of local variables and parameters for each recursive call. This enables algorithms like recursive traversal of data structures (e.g., trees) or divide-and-conquer strategies (e.g., quicksort) to function correctly without interference between recursive instances.

Exception Handling: In languages with stack-based exception handling mechanisms (such as Java's try-catch-finally blocks), stack frames are critical for managing exceptions. When an exception occurs, the call stack helps trace back to the appropriate exception handler, ensuring proper error reporting and recovery.

FIFO Queue: Concrete Data Structure

FIFO (First-In-First-Out) is a fundamental concept in data structures and computer science, describing a way to manage and process elements in a collection. The FIFO principle dictates that the first element added to the collection will be the first one to be removed, much like a queue in real life (e.g., a line at a checkout counter).

Structure of FIFO Queue

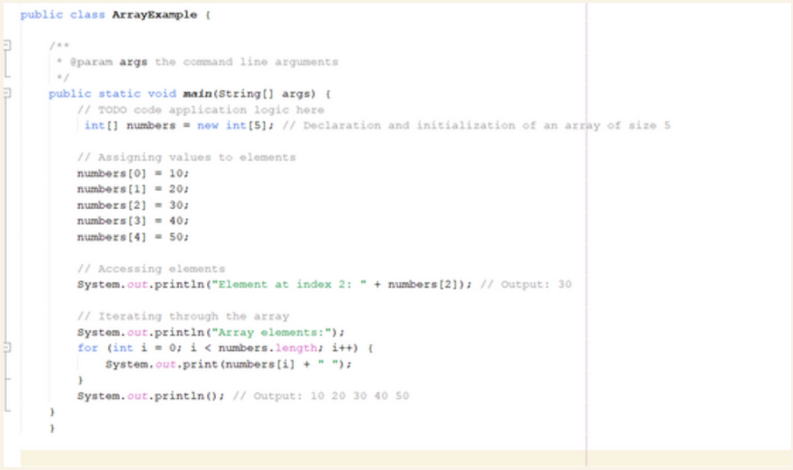
Characteristics

- Order Preservation: Elements are processed in the same order they are added, maintaining a sequential flow of operations.
- Addition and Removal: Elements are enqueued (added) to the end of the queue and dequeued (removed) from the front.
- Behavior: It exhibits a first-come-first-served (FCFS) or first-come-first-processed behavior, ideal for scenarios requiring sequential processing of tasks or data.

Implementation Details

- enqueue(element): Adds an element to the end of the queue.
- dequeue(): Removes and returns the element at the front of the queue.
- peek(): Returns the element at the front of the queue without removing it.
- isEmpty(): Checks if the queue is empty.
- isFull(): Checks if the queue is full (if implemented with a fixed size).

Array-Based Implementation



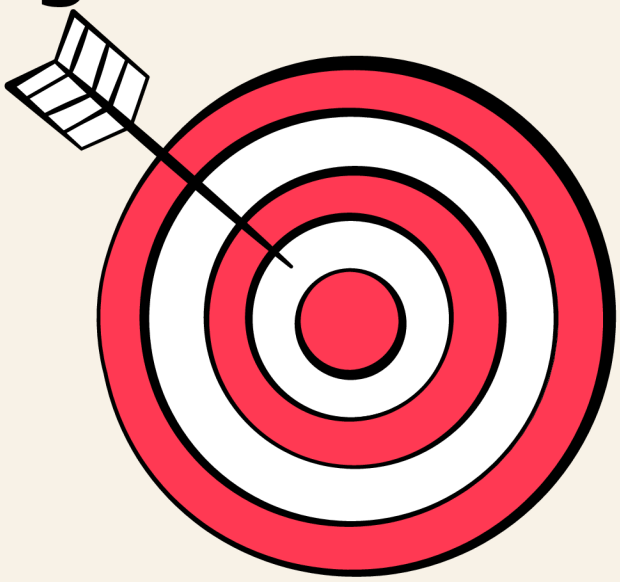
```
public class ArrayExample {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        int[] numbers = new int[5]; // Declaration and initialization of an array of size 5

        // Assigning values to elements
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
        numbers[4] = 50;

        // Accessing elements
        System.out.println("Element at index 2: " + numbers[2]); // Output: 30

        // Iterating through the array
        System.out.println("Array elements:");
        for (int i = 0; i < numbers.length; i++) {
            System.out.print(numbers[i] + " ");
        }
        System.out.println(); // Output: 10 20 30 40 50
    }
    }
```

An array-based implementation refers to the utilization of arrays, a fundamental data structure in programming, to create and manage collections of elements. Arrays provide a contiguous block of memory where elements of the same data type are stored sequentially

Dijkstra's Algorithm

Definition

With Dijkstra's Algorithm, you can find the shortest path between nodes in a graph. Particularly, you can find the shortest path from a node (called the "source node") to all other nodes in the graph, producing a shortest-path tree.



Explain Dijkstra's Algorithm Use For

Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.

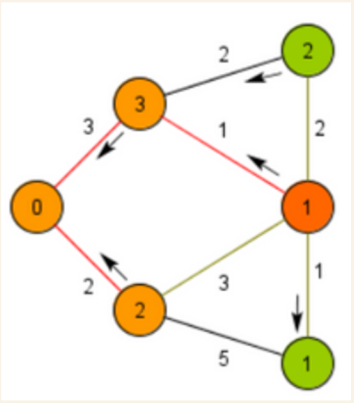
The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.

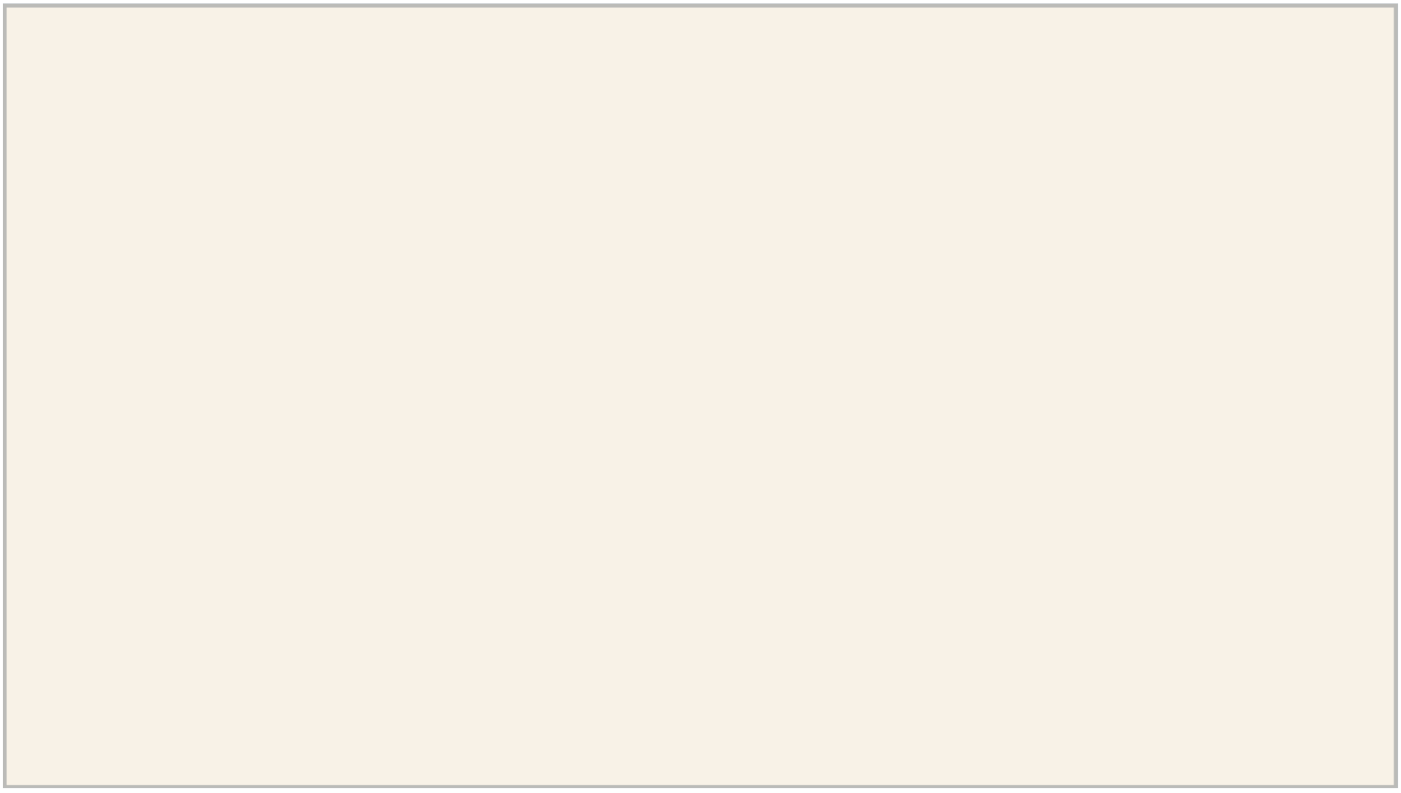
Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.

The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Prim-Jarnik Algorithm

The Prim-Jarnik Algorithm, also known simply as Prim's Algorithm, is a minimum spanning tree algorithm. It finds the subset of edges that forms a tree including every vertex, where the total weight of all the edges in the tree is minimized. This algorithm is particularly useful in the design of networks, such as computer, telecommunication, and transportation networks.





Conclusion

Summary: ADTs provide a structured approach to data management, enhancing software design and development.
Importance of ADTs,
Example implementations
Benefits of using well-defined ADTs in software development
The concrete data structure for a First in First out (FIFO) queue.


Thank You