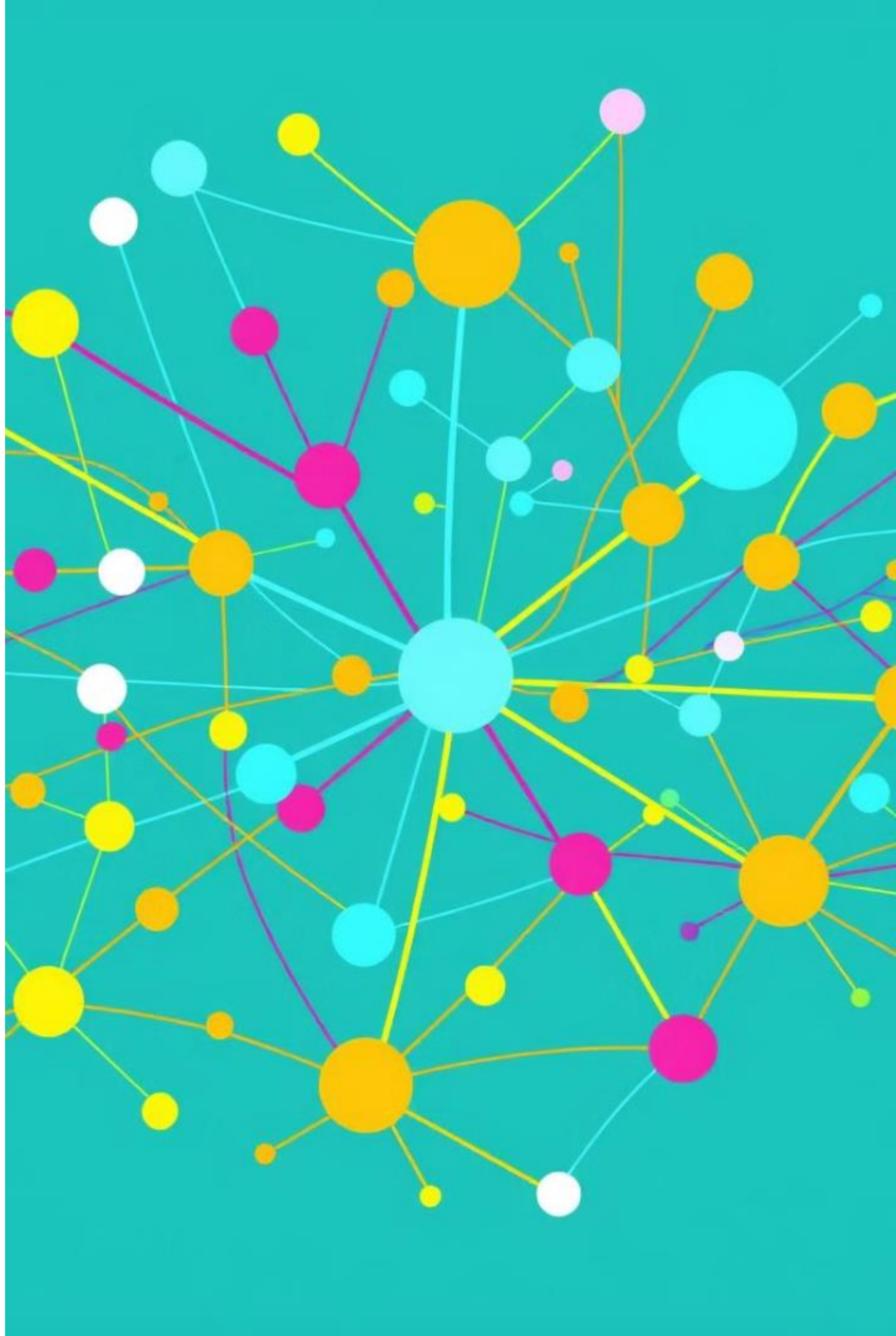
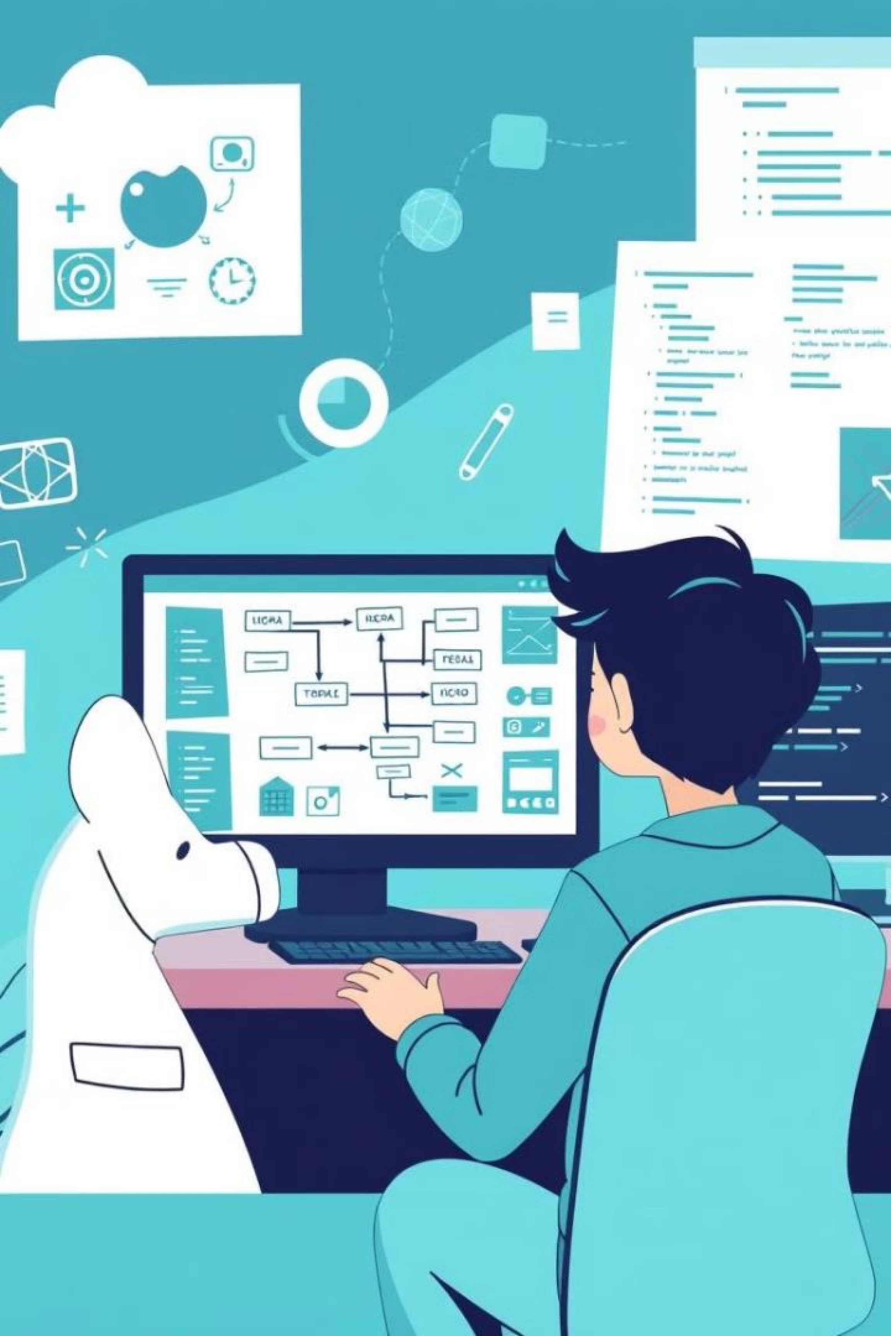
Create a design specification for data structures, explaining the valid operations that can be carried out on the structures (P1)

Data structures are fundamental to computer science, serving as blueprints for organizing and storing data. They underpin software development, enabling efficient data access, manipulation, and management.





Identifying Appropriate Data Structures

Data Type

Consider the type of data you're dealing with: numbers, text, or complex objects.

Operations

Determine the operations you need to perform on the data: insertion, deletion, search, or sorting.

Memory Usage

Evaluate the memory efficiency required for your application, considering the volume and complexity of data.

Array: Operations and Use Cases

1 **Insertion**

Adding elements to an array can be performed at specific indices.

2 **Deletion**

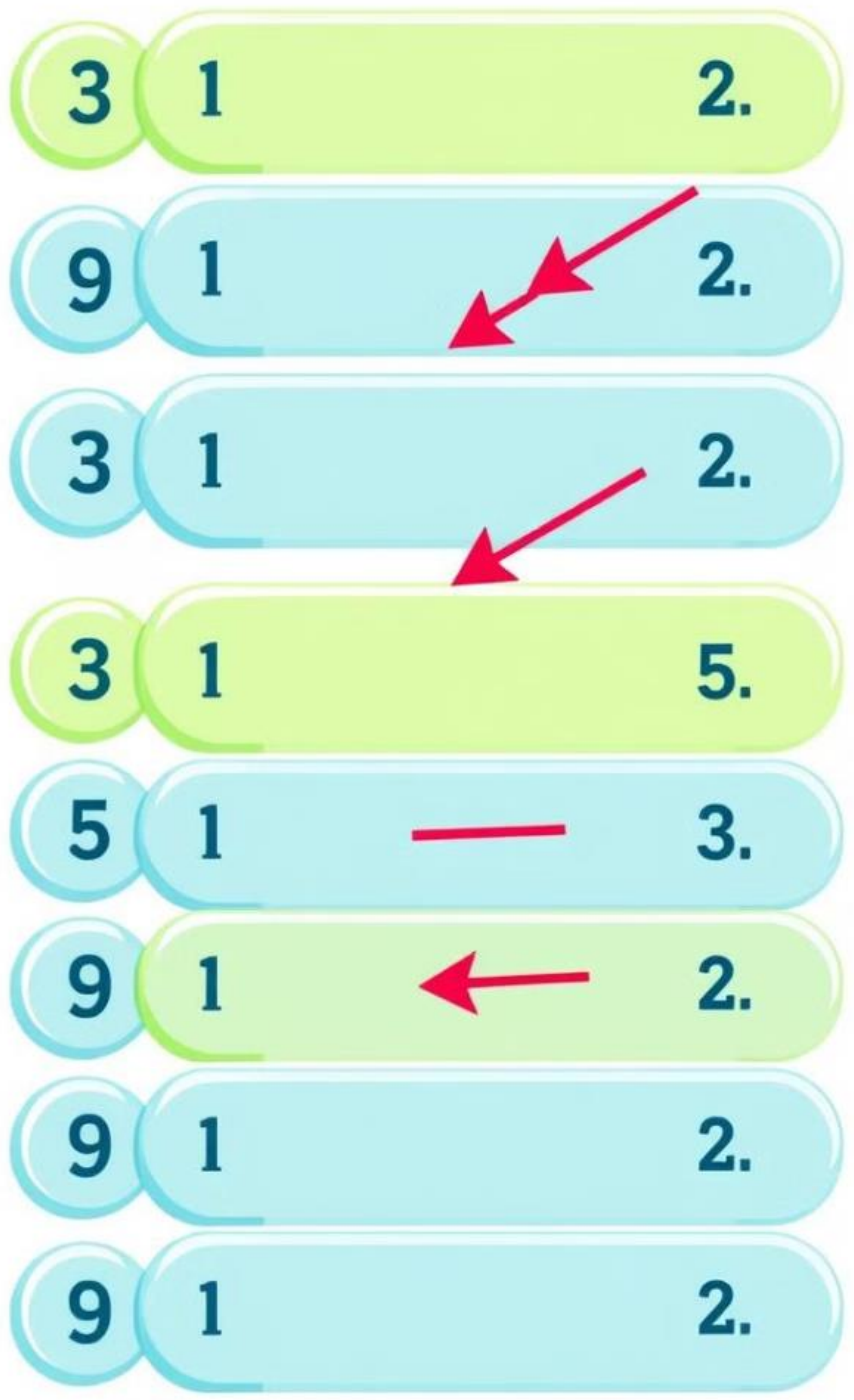
Removing elements from an array involves shifting elements after the deleted element.

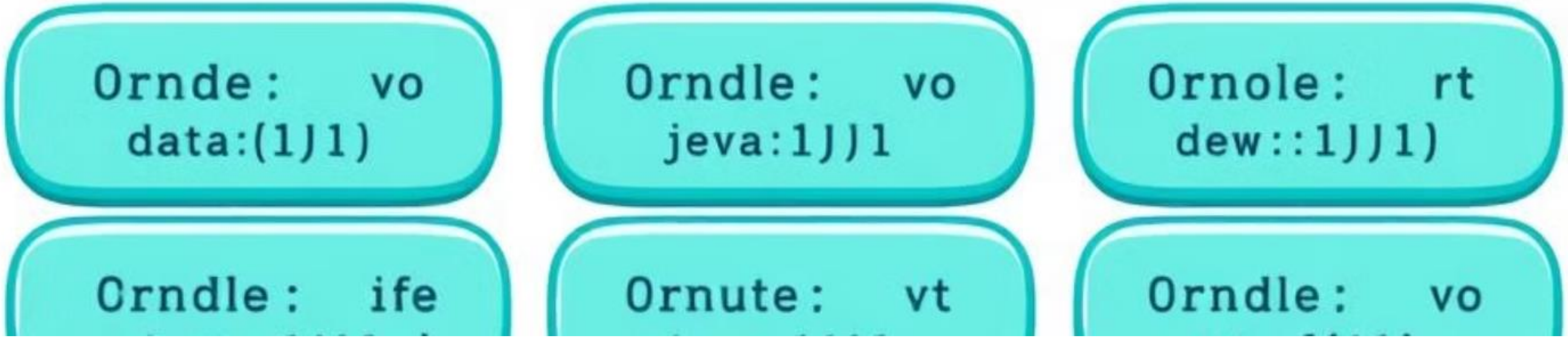
3 **Search**

Searching for a specific element in an array is done by iterating through the elements.

4 **Use Cases**

Arrays are commonly used in storing lists, implementing stacks and queues, and performing matrix operations.





Linked List: Operations and Use Cases

1 **Insertion**

Elements can be inserted at the beginning, end, or any specific position in a linked list.

2 **Deletion**

Deleting elements involves updating pointers to remove the element from the list.

3 **Traversal**

Iterating through the list to access elements is done by following the pointers.

Stack: Operations and Use Cases

1. **Push**

   Adding an element to the top of the stack.

2. **Pop**

   Removing the top element from the stack.

3. **Peek**

   Viewing the top element without removing it.

Queue: Operations and Use Cases

Enqueue

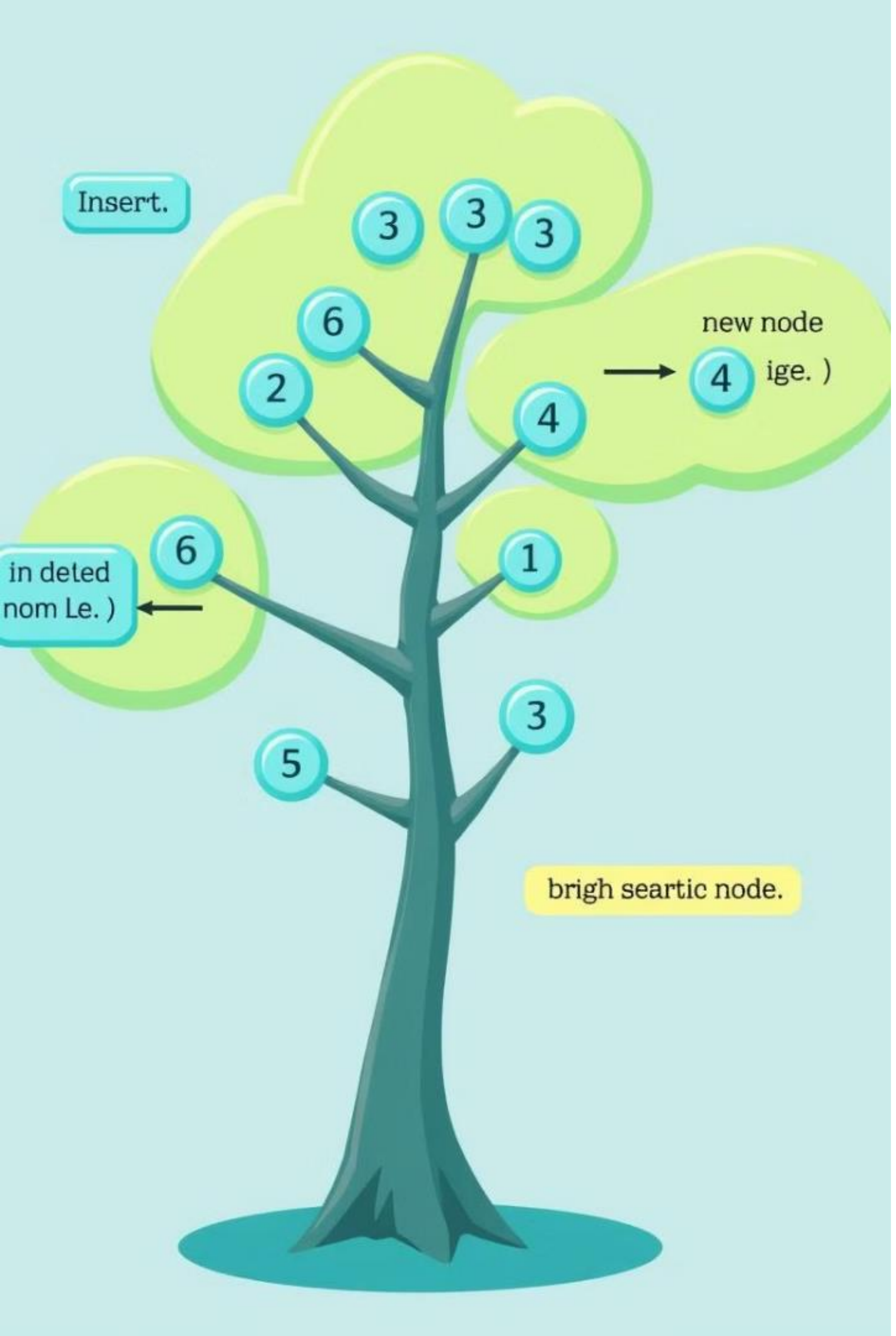
Adding an element to the rear of the queue.

Dequeue

Removing the front element from the queue.

Peek

Viewing the front element without removing it.



Binary Tree: Operations and Use Cases

Operation	Description
Insertion	Adding a new node to the tree, maintaining the binary tree property.
Deletion	Removing a node from the tree while preserving the binary tree structure.
Search	Locating a specific node in the tree based on its value.

Hash Table: Operations and Use Cases

Insertion

Adding key-value pairs to the hash table.

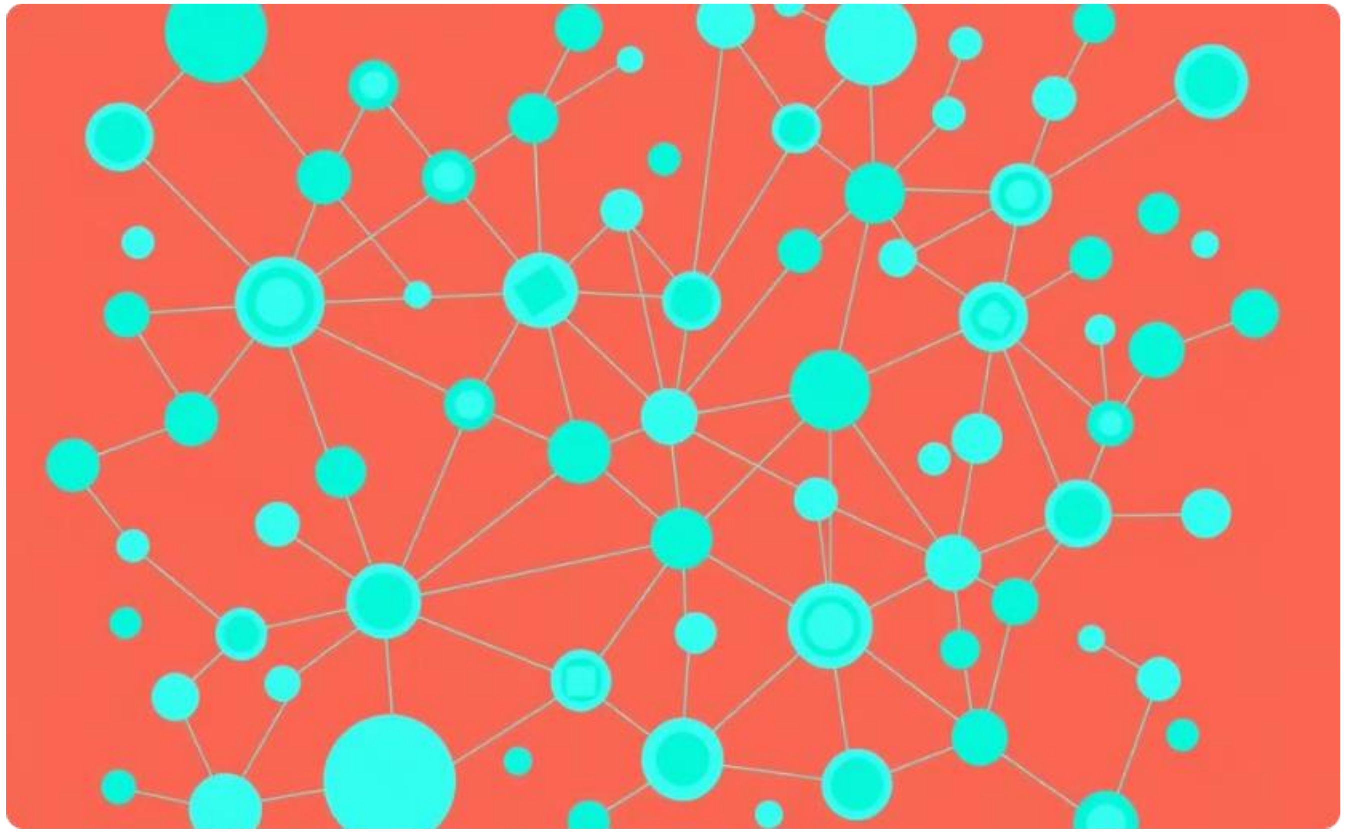
Deletion

Removing a key-value pair from the hash table.

Search

Retrieving the value associated with a specific key.

Graph: Operations and Use Cases

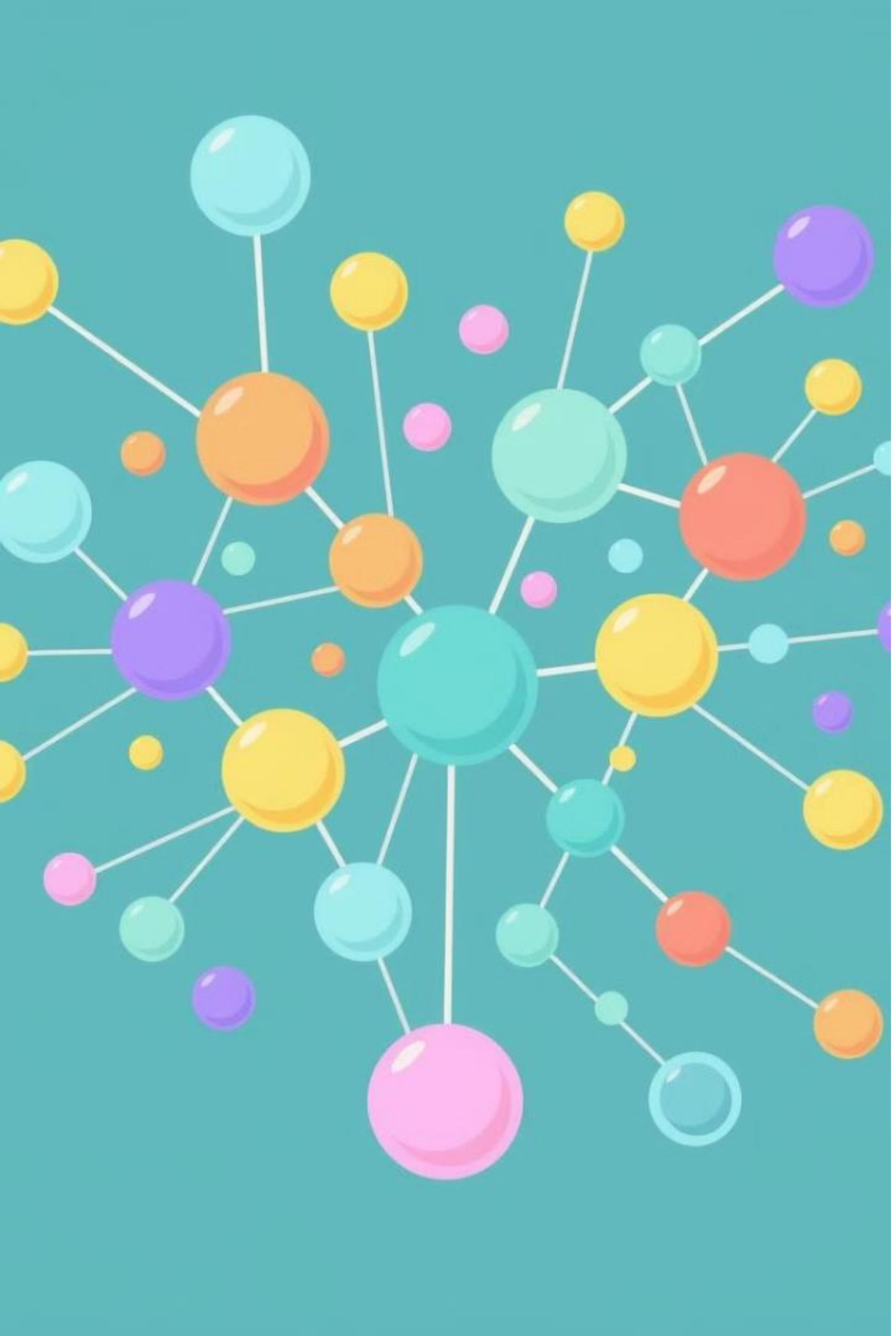


Social Networks

Graphs model connections between users in social networks.

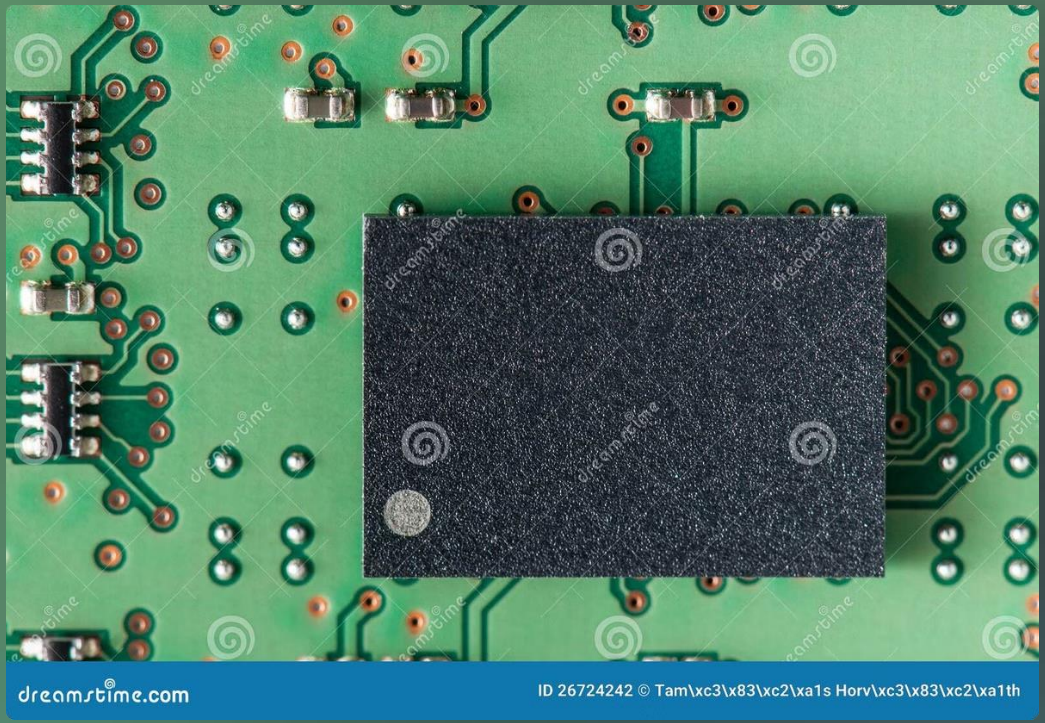
Navigation Systems

Graphs represent road networks, enabling efficient route planning.



Conclusion and Recommendations

Understanding data structures is crucial for efficient data management and algorithm design. Choose the appropriate data structure based on your specific requirements and data characteristics.



Determine the operations of a memory stack and how it is used to implement function calls in a computer (P2)

A stack is a fundamental data structure in computer science. It's used to store information during program execution. This presentation will delve into its operations and how it implements function calls.

Define a Memory Stack

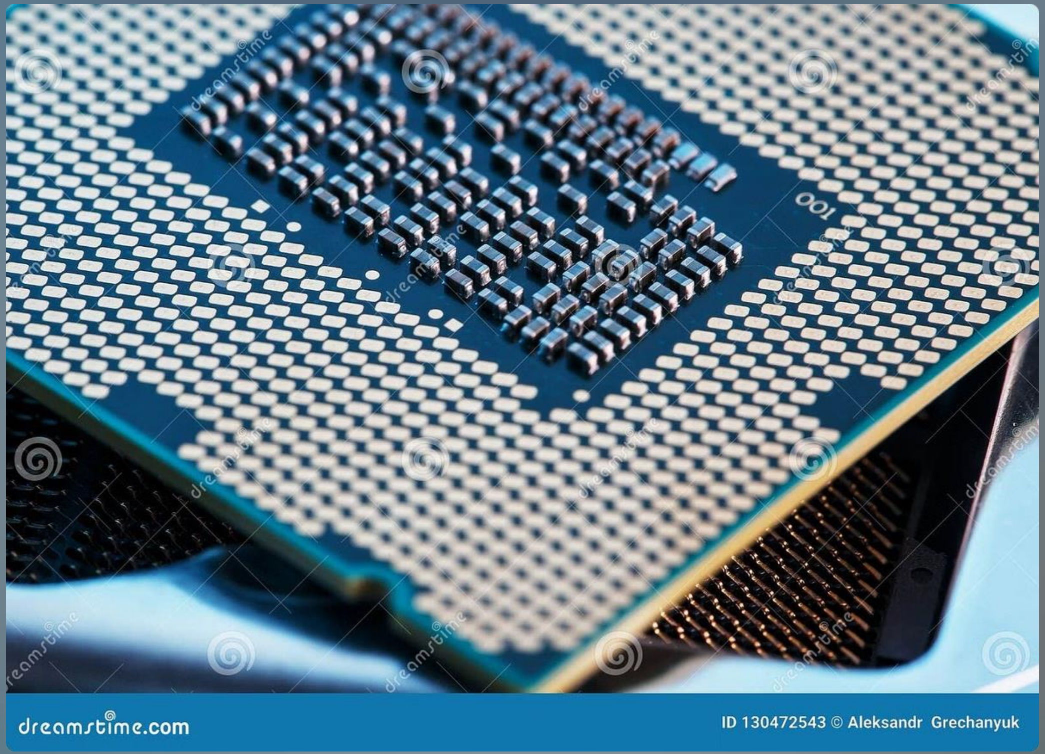
A stack is a linear data structure, conceptually like a stack of plates. The Last-In, First-Out (LIFO) principle applies, where the last element added to the stack is the first one retrieved.

Data Organization

A stack stores data sequentially, with a fixed memory address for the top element.

LIFO Access

Elements can only be accessed or removed from the top of the stack, following the LIFO principle.



The role of the Stack Pointer (SP) register

The Stack Pointer (SP) is a dedicated register in the CPU that keeps track of the top of the stack. It holds the memory address of the current top element.

1

Push Operation

When a new element is added, SP is decremented to point to the next available memory location.

2

Pop Operation

When an element is removed, SP is incremented to point to the new top of the stack.

Push and Pop operations in a stack

The push operation adds a new element to the top of the stack. The pop operation removes the top element from the stack.

Push

1. SP is decremented

2. The new element is stored at the memory location pointed to by SP.

Pop

1. The element at the memory location pointed to by SP is retrieved.

2. SP is incremented.

Implementing function calls using a stack

Function calls are essential for modularity in programming. They involve temporarily switching execution to a different part of the code.

1. **Function Call**

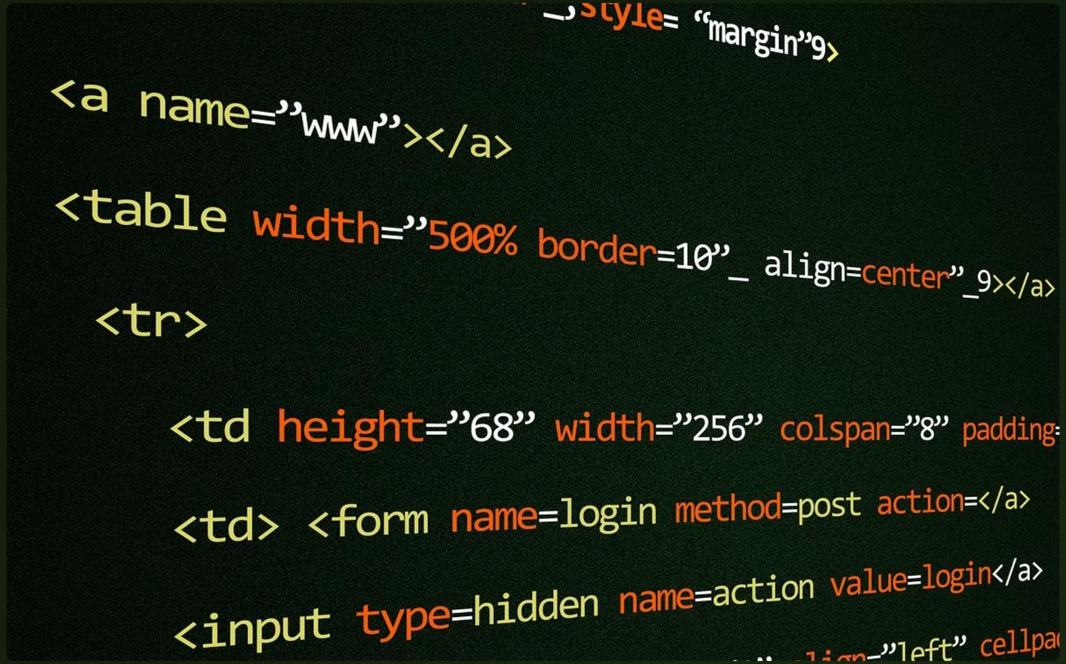
   The current state of the program, including variables and program counter, is pushed onto the stack.

2. **Function Execution**

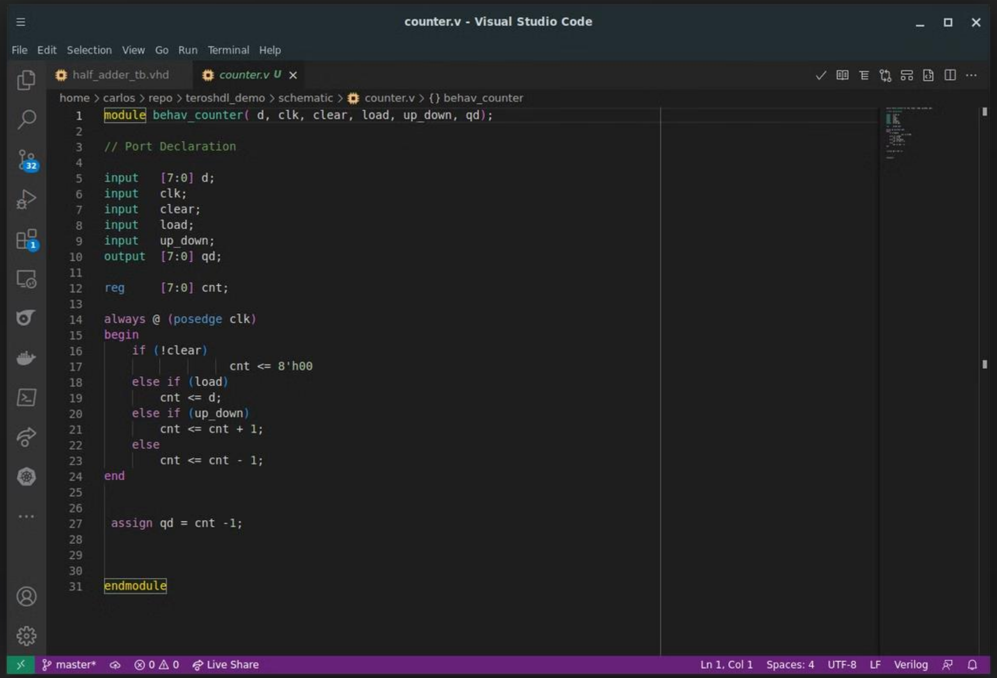
   Control is transferred to the called function, and the function's local variables and parameters are pushed onto the stack.

3. **Function Return**

   Upon function completion, the stack is unwound, restoring the previous program state. The return value is stored on the stack.



```
style= “margin”9>
<a name=”www”></a>
<table width=”500% border=10”_ align=center”_9></a>
  <tr>
     <td height=”68” width=”256” colspan=”8” padding
     <td> <form name=login method=post action=</a>
     <input type=hidden name=action value=login</a>
```


counter.v - Visual Studio Code
File Edit Selection View Go Run Terminal Help
half_adder_tb.vhd
counter.v U
home > carlos > repo > teroshdl_demo > schematic > counter.v > {} behav_counter
master*
Live Share
Ln 1, Col 1 Spaces: 4 UTF-8 LF Verilog

```verilog
module behav_counter( d, clk, clear, load, up_down, qd);

// Port Declaration

input   [7:0] d;
input   clk;
input   clear;
input   load;
input   up_down;
output  [7:0] qd;

reg     [7:0] cnt;

always @ (posedge clk)
begin
    if (!clear)
                cnt <= 8'h00
    else if (load)
        cnt <= d;
    else if (up_down)
        cnt <= cnt + 1;
    else
        cnt <= cnt - 1;
end


 assign qd = cnt -1;



endmodule
```

Storing return addresses on the stack

When a function is called, the address of the instruction following the call needs to be stored. This is the return address.

1

Return Address

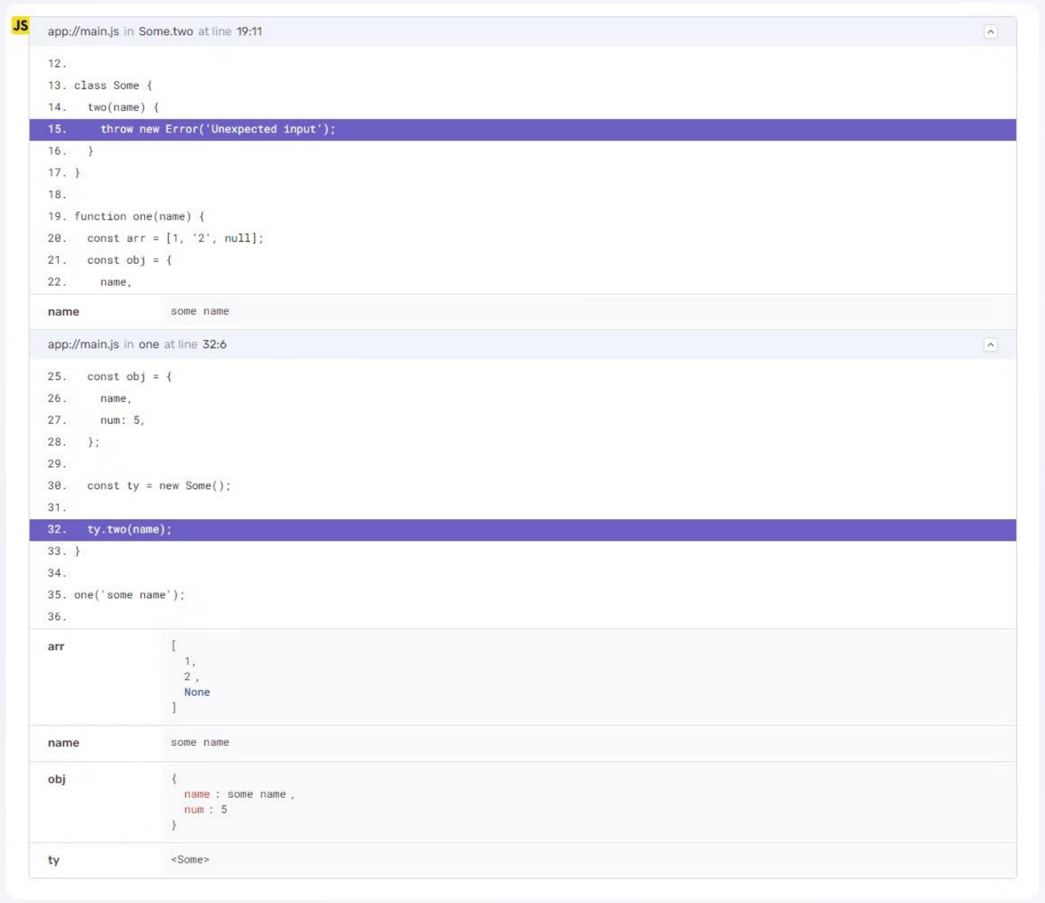
The return address is pushed onto the stack before the function's code is executed.

2

Function Completion

When the function finishes, the return address is popped from the stack, and execution continues at the instruction following the call.

Managing local variables on the stack



app://main.js in Some.two at line 19:11

```

class Some {
  two(name) {
    throw new Error('Unexpected input');
  }
}

function one(name) {
  const arr = [1, '2', null];
  const obj = {
    name,
```

```
name        some name
```

app://main.js in one at line 32:6

```
  const obj = {
    name,
    num: 5,
  };

  const ty = new Some();

  ty.two(name);
}

one('some name');

```

```
arr         [
              1,
              2 ,
              None
            ]
name        some name
obj         {
              name : some name ,
              num : 5
            }
ty          <Some>
```

Each function has its own scope for variables that are only accessible within its definition. These are local variables.

Push	Local variables are allocated space on the stack when a function is called.
Pop	The space for local variables is deallocated when the function returns.

Handling function parameters and return values

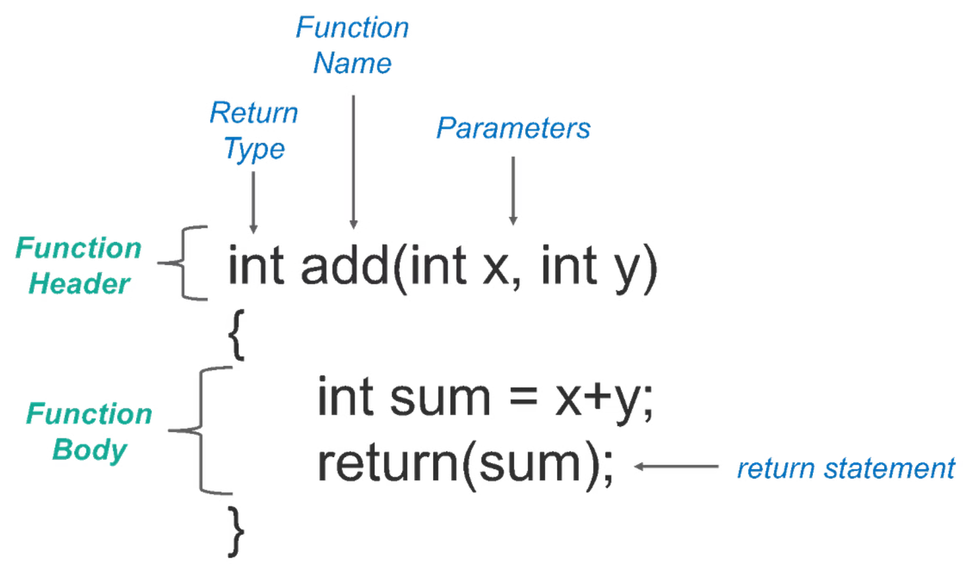
When a function is called, the arguments passed to it are called parameters. The result of a function is the return value.

Parameters

Parameters are pushed onto the stack before the function is called. They can be accessed by the function.

Return Value

The return value is stored on the stack when the function completes. It can be retrieved by the calling function.



```
int add(int x, int y)
{
    int sum = x+y;
    return(sum);
}
```

Nested function calls and the stack

Functions can call other functions, leading to nested function calls. The stack is essential for managing this process.

1. **Outer Function Call**

   The outer function's state is pushed onto the stack.

2. **Inner Function Call**

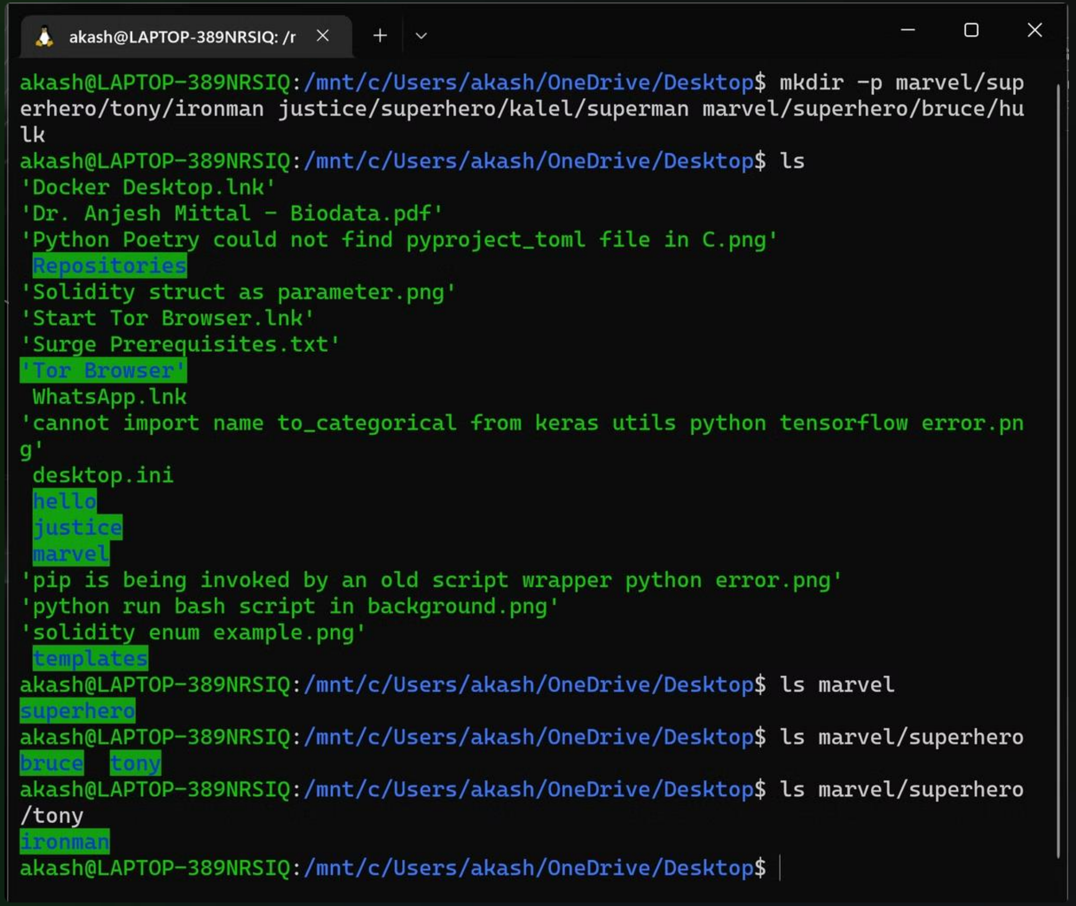
   The inner function's state is pushed onto the stack, further extending the stack.

3. **Return from Inner Function**

   The inner function's state is popped from the stack, restoring the outer function's context.

4. **Return from Outer Function**

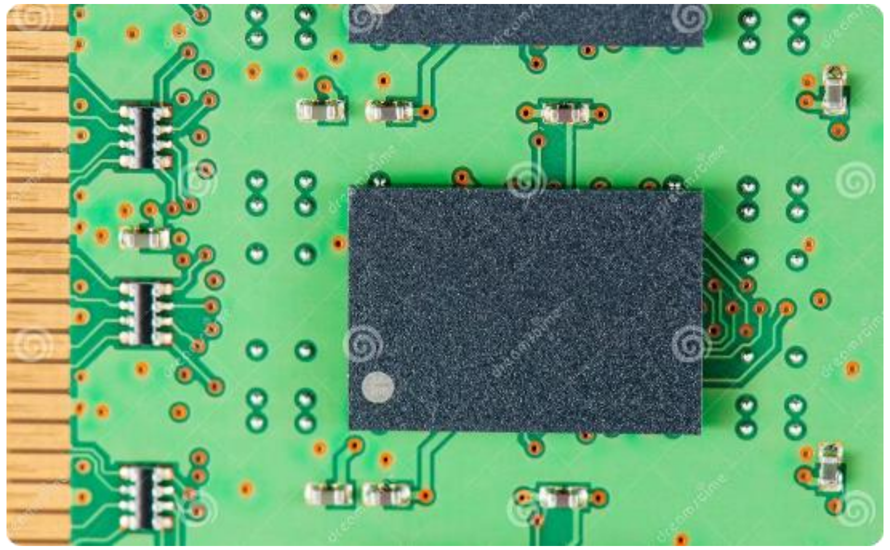
   The outer function's state is popped from the stack, restoring the program's original state.



```
akash@LAPTOP-389NRSIQ:/mnt/c/Users/akash/OneDrive/Desktop$ mkdir -p marvel/sup
erhero/tony/ironman justice/superhero/kalel/superman marvel/superhero/bruce/hu
lk
akash@LAPTOP-389NRSIQ:/mnt/c/Users/akash/OneDrive/Desktop$ ls
'Docker Desktop.lnk'
'Dr. Anjesh Mittal - Biodata.pdf'
'Python Poetry could not find pyproject_toml file in C.png'
 Repositories
'Solidity struct as parameter.png'
'Start Tor Browser.lnk'
'Surge Prerequisites.txt'
'Tor Browser'
 WhatsApp.lnk
'cannot import name to_categorical from keras utils python tensorflow error.pn
g'
 desktop.ini
 hello
 justice
 marvel
'pip is being invoked by an old script wrapper python error.png'
'python run bash script in background.png'
'solidity enum example.png'
 templates
akash@LAPTOP-389NRSIQ:/mnt/c/Users/akash/OneDrive/Desktop$ ls marvel
superhero
akash@LAPTOP-389NRSIQ:/mnt/c/Users/akash/OneDrive/Desktop$ ls marvel/superhero
bruce  tony
akash@LAPTOP-389NRSIQ:/mnt/c/Users/akash/OneDrive/Desktop$ ls marvel/superhero
/tony
ironman
akash@LAPTOP-389NRSIQ:/mnt/c/Users/akash/OneDrive/Desktop$
```

Conclusion: Importance of the memory stack in computer architecture

The stack is a critical component in computer architecture. It facilitates modularity, function calls, and program execution.

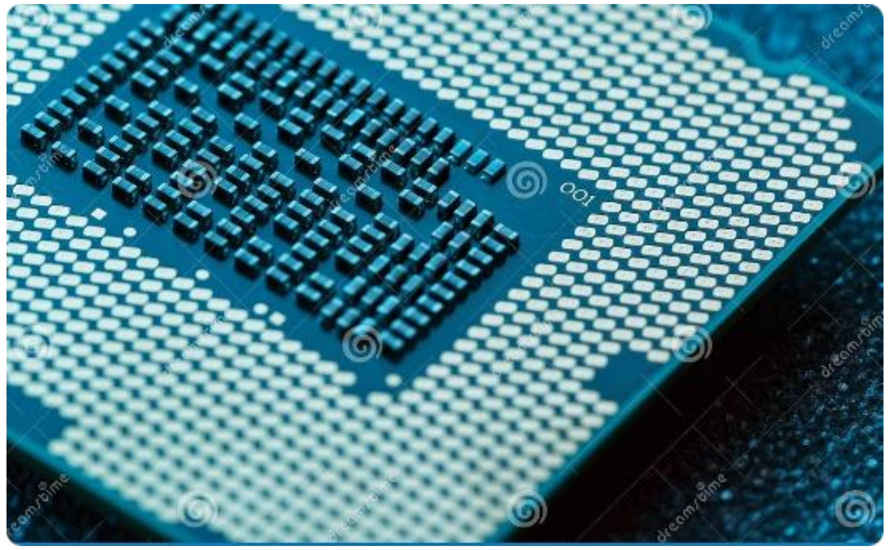


Stack Operations

Efficiently storing and retrieving information for various program operations.

Function Calls

Enabling seamless function calls and returns, facilitating modular programming.



Memory Management

Managing memory efficiently, providing temporary storage for local variables and function parameters.