**ARRAYS**

**WHAT IS AN ARRAY**

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value,

**Declaring Arrays**

* Array variables are declared identically to variables of their data type, except that the variable name is followed by one pair of square [ ] brackets for each dimension of the array.
* Uninitialized arrays must have the dimensions of their rows, columns, etc. listed within the square brackets.
* Dimensions used when declaring arrays in C must be positive integral constants or constant expressions.
  + In C++, dimensions must still be positive integers, but variables can be used, so long as the variable has a positive value at the time the array is declared.

**TYPES OF ARRAYS**

## One-Dimensional Arrays

## Multi-Dimensional Arrays

## One-Dimensional Arrays

## A standard way of initializing arrays is as follow

## 

## An example includes

## int a[5] = {1,2,3,4,5}; The array store a total of 5 elements

## Multi-Dimensional Arrays

These multi-dimensional arrays are again of two types. They are:

* **Two dimensional array**

An example can be illustrated below, it stores a total of **6 elements**



## Three dimensional array

## An example is illustrated below, it stores a total of 24 elements

## 

**Applications of Array Data Structure:**

**Storing and accessing data:** Arrays are used to store and retrieve data in a specific order. For example, an array can be used to store the scores of a group of students, or the temperatures recorded by a weather station.

**Sorting:** Arrays can be used to sort data in ascending or descending order. Sorting algorithms such as bubble sort, merge sort, and quicksort rely heavily on arrays.

**Searching:** Arrays can be searched for specific elements using algorithms such as linear search and binary search.

**Insertion:** Insertion in an array is the process of including one or more elements in an array. The insertion can be done at the beginning, at the end and at any index of an array.

**Matrices:** Arrays are used to represent matrices in mathematical computations such as matrix multiplication, linear algebra, and image processing.

**Stacks and queues:** Arrays are used as the underlying data structure for implementing stacks and queues, which are commonly used in algorithms and data structures.

**Graphs:** Arrays can be used to represent graphs in computer science. Each element in the array represents a node in the graph, and the relationships between the nodes are represented by the values stored in the array.

**Dynamic programming:** Dynamic programming algorithms often use arrays to store intermediate results of sub-problems in order to solve a larger problem.

**Advantages of array data structure:**

**Efficient access to elements:** Arrays provide direct and efficient access to any element in the collection.

Accessing an element in an array is an O(1) operation, meaning that the time required to access an element is constant and does not depend on the size of the array.

**Fast data retrieval:** Arrays allow for fast data retrieval because the data is stored in contiguous memory locations. This means that the data can be accessed quickly and efficiently without the need for complex data structures or algorithms.

**Memory efficiency:** Arrays are a memory-efficient way of storing data. Because the elements of an array are stored in contiguous memory locations, the size of the array is known at compile time. This means that memory can be allocated for the entire array in one block, reducing memory fragmentation.

**Versatility:** Arrays can be used to store a wide range of data types, including integers, floating-point numbers, characters, and even complex data structures such as objects and pointers.

**Easy to implement:** Arrays are easy to implement and understand, making them an ideal choice for beginners learning computer programming.

**Compatibility with hardware:** The array data structure is compatible with most hardware architectures, making it a versatile tool for programming in a wide range of environments.

**Complexity:** Arrays offer O(1) search if we know the index.

**Disadvantages of array data structure:**

**Fixed size:** Arrays have a fixed size that is determined at the time of creation. This means that if the size of the array needs to be increased, a new array must be created and the data must be copied from the old array to the new array, which can be time-consuming and memory-intensive.

**Memory allocation issues:** Allocating a large array can be problematic, particularly in systems with limited memory. If the size of the array is too large, the system may run out of memory, which can cause the program to crash.

**Insertion and deletion issues:** Inserting or deleting an element from an array can be inefficient and time-consuming because all the elements after the insertion or deletion point must be shifted to accommodate the change.

**Wasted space:** If an array is not fully populated, there can be wasted space in the memory allocated for the array. This can be a concern if memory is limited.

**Limited data type support:** Arrays have limited support for complex data types such as objects and structures, as the elements of an array must all be of the same data type.

**Lack of flexibility:** The fixed size and limited support for complex data types can make arrays inflexible compared to other data structures such as linked lists and trees.

### **Complexities of Array Operations**

Here are some examples of the complexity of an array.

**Merge Sort**

Time Complexity: O(n log n)

Space Complexity: O(n)

**Quick Sort**

Time Complexity: O(n log n)

Space Complexity: O(log n)

#### **Binary Search**

Time Complexity: O(log n)

Space Complexity: O(1)

**Linear search**

Time complexity: O(n)

Space complexity: O(1)

## Binary Trees

A binary tree is a sort of tree data structure in which each node has a maximum of two offspring, known as the left child and the right child. It is a hierarchical structure that arranges elements or nodes in a certain hierarchy.

Each node in a binary tree might have zero, one, or two offspring. The root node of the tree is the highest node, and it is from this node that the tree branches out into subtrees. The left and right child pointers connect the nodes in the subtrees to their parent nodes.

### **Operations of Binary Trees**

Binary trees are compatible with several techniques that make tree manipulation and traversal efficient. Here are a few regular operations on binary trees:

* **Insertion:**  In order to insert a new node into a binary tree, the correct position must be determined based on the node’s value, and the relevant connections must be made. The node is put in a way that preserves the tree’s ordering attribute if it is a binary search tree.
* **Deletion:**    A binary tree’s deletion of a node entails removing the node from the tree while maintaining the structure of the binary tree. The deletion operation could include altering the tree structure, such as moving child nodes around or elevating successor/predecessor nodes, depending on the node’s position and the intended deletion approach.
* **Searching:**     A binary tree can be searched for a certain value by going from root to leaf and comparing the target value to the values in each node. If the tree is a binary search tree, the search process can be made more efficient by comparing the target value to the values of the nodes and then traversing either the left or right subtree depending on the comparison outcome.
* **Traversals:** A binary tree can be traversed by going through each node in turn. There are various traversal techniques, such as: Visit the left subtree first, followed by the current node, and then the right subtree. For binary search trees, this traversal generates nodes in ascending order. Pre-order Visit the current node first, followed by the left subtree, and then the right subtree. This traversal is frequently used to evaluate expressions or duplicate the tree structure. Visit the left subtree first, then the right subtree, and finally the current node in the post-order traversal. When deleting nodes or carrying out other operations that call for processing child nodes before the parent node, this traversal is helpful. Visiting nodes at each level from left to right, beginning at the root, is known as level-order traversal. This traversal, which frequently makes use of a queue, examines the tree level by level.
* **Measurement of height:**   The length of the longest path from a binary tree’s root to a leaf node determines the tree’s height. A tree’s overall structure and balance can be evaluated by computing its height. A tree with no nodes (the root) has a height of 1, while a tree with one node has a height of 0.
* **Size estimation:**   The total number of nodes in a binary tree determines the size of the tree. By adding the sizes of the left and right subtrees together and adding one (for the root), it can be determined recursively.

### **Applications of Binary Trees**

Due to their adaptability and effective operations, binary trees have a wide range of uses in computer science and programming. Following are a few typical uses for binary trees:

1. Binary Search Trees (BST)
2. Expression Trees
3. Huffman Coding
4. Decision Trees
5. File System Indexing
6. Game Trees
7. Syntax Tree Parsing

These are just a few of the many uses that binary trees have. They are a fundamental data structure utilized in many different domains, including data structures, algorithms, artificial intelligence, and software development. This is due to their hierarchical structure, effective search capabilities, and adaptability.

### **Properties of Binary Trees**

Multiple significant characteristics of binary trees define their structure and behavior. Effective use of binary trees requires an understanding of these features. Here are a few crucial characteristics of binary trees:

1. Node Structure
2. Root Node
3. Parent and child Nodes
4. Leaf Nodes­­
5. Internal Nodes
6. Subtrees
7. Balanced Vs Unbalanced
8. Full Binary Trees
9. Complete Binary Trees

**BINARY SEARCH TREE (BST)**

## AVL (Adelson-Velskii and Landis) Trees

Binary search trees (BSTs) that self-balance and rotate automatically are known as AVL trees. It bears the names of its creators, Adelson-Velskii and Landis. Height-balanced AVL trees guarantee that every node’s left and right subtrees have height differences between them of no more than one.

Keeping search, insertion, and deletion operations effective is the basic objective of balancing an AVL tree. By maintaining the tree’s balance, the temporal complexity of these operations stays logarithmic, resulting in optimal performance.

### **Operations of AVL(BST) Trees**

* **Insertion:**When adding a new node to an AVL tree, rotations may be necessary to keep the tree balanced. The rotations can be left, right, or double.
* **Deletion:**    Like insertion, deletion can throw off the AVL tree’s equilibrium. To get back into balance and keep the AVL property, rotations are done.
* **Searching:**    When conducting a search, the target value is compared to the values of each node in the AVL tree, and the left or right subtree is then traversed as necessary.

**Structure:**

* In a binary search tree, each node has at most two children: a left child with a value less than the parent and a right child with a value greater than the parent.
* The left and right subtrees of each node are also binary search trees.

**Properties:**

* The binary search tree property ensures that for any given node, all nodes in its left subtree have values less than its own, and all nodes in its right subtree have values greater than its own.

### **Applications of AVL Trees**

* **Databases:**To facilitate effective data retrieval and maintenance, AVL trees are frequently employed in database indexing systems.
* AVL trees are frequently used in compiler symbol tables for quick look up and storage of identifiers.
* For effective route management in computer networks, AVL trees can be utilized in routing algorithms.
* **Caches:**To store frequently accessed data and improve cache lookup, AVL trees are used in cache implementations.

### **Properties of AVL Trees**

* **Balance Factor:**Each node in an AVL tree has a balancing factor, which is the difference between the heights of its left and right subtrees. The balance factor, which can range from -1 to 1, indicates whether the node is balanced, slightly left- or right-heavy, respectively.
* **Height:**The length of the longest path from a node to a leaf determines the node’s height. An empty tree is said to have a height of 0.
* **Balance Condition:**In an AVL tree, the balancing factor for each node must fall between [-1, 1].

**Advantages:**

1. **Efficient Search:** Binary Search Trees provide efficient search operations with a time complexity of O(log n) on average, making them suitable for applications where quick retrieval is crucial.
2. **Ordered Structure:** BSTs maintain an ordered structure, making it easy to perform operations like finding the minimum or maximum element, or traversing elements in sorted order.
3. **Simple Implementation:** Binary Search Trees have a straightforward implementation, making them accessible for various applications that require ordered data storage.

**Disadvantages:**

1. **Degenerate Structure:** In certain scenarios, BSTs can degenerate into a linked list, resulting in degraded performance (O(n) time complexity for search operations).
2. **Sensitivity to Input Order:** The efficiency of a BST depends on the order of insertions, and poorly ordered input can lead to a skewed tree, affecting search times.
3. **Lack of Balance:** Self-balancing techniques (e.g., AVL trees) are needed to maintain optimal performance in dynamic scenarios.

**EXPRESSION TREE**

An expression tree is a tree data structure used to represent mathematical expressions. In this tree, each node represents an operand or an operator in the expression. The leaves of the tree are the operands (constants or variables), and the internal nodes represent operators.

**Purpose:**

* Expression trees are used to represent mathematical expressions in a tree-like structure.
* They are commonly used for evaluating mathematical expressions efficiently.

**Structure:**

* Expression trees represent expressions where operators are internal nodes, and operands are leaf nodes.
* The tree’s structure mirrors the order of operations, ensuring proper evaluation.

**Properties:**

* Nodes in an expression tree can represent not just numbers but also variables or more complex sub-expressions.

**Operations**

* **Construction:**

**Build Tree**: Create an expression tree from a given mathematical expression.

* **Traversal:**

**In-order** **Traversal**: Visit the nodes of the tree in the order (left operand, operator, right operand), which corresponds to the original infix expression.

**Preorder Traversal**: Visit the nodes in the order (operator, left operand, right operand).

**Post-order** **Traversal**: Visit the nodes in the order (left operand, right operand, operator).

* **Evaluation**:

**Evaluate Tree:** Compute the result of the mathematical expression represented by the tree.

* **Simplification**:

**Simplify Tree:** Simplify the expression tree by evaluating constant subexpressions.

* **Conversion**:

**Infix to Postfix**: Convert the expression from infix notation to postfix notation using the expression tree.

**Infix to Prefix**: Convert the expression from infix notation to prefix notation.

* **Analysis**:

**Height of Tree**: Determine the height (depth) of the expression tree.

**Number of Nodes**: Count the total number of nodes in the tree.

**Identify Variables/Constants**: Extract and list the variables or constants present in the expression.

* **Derivation**:

**Differentiation**: Derive the expression with respect to a specific variable, producing a new expression tree representing the derivative.

* **Substitution:**

**Substitute Values**: Replace variables in the expression with specific numerical values.

**Advantages of Expression Trees:**

1. **Hierarchical Representation:** Expression trees provide a hierarchical and structured representation of mathematical expressions, making it easier to understand and manipulate complex mathematical formulas.
2. **Efficient Evaluation:** Once constructed, expression trees can be efficiently evaluated. By traversing the tree in a systematic way, mathematical expressions can be computed rapidly.
3. **Facilitates Optimization:** Expression trees offer a foundation for optimizing mathematical expressions. Common subexpressions can be identified and computed only once, reducing redundant computations.

**Disadvantages of Expression Trees:**

1. **Complex Construction:** Constructing expression trees from raw mathematical expressions can be complex and resource-intensive, especially for intricate or nested mathematical formulas.
2. **Memory Overhead:** Expression trees can consume additional memory compared to a simple linear representation of an expression. This overhead is due to the storage of tree nodes and pointers.
3. **Limited Applicability:** While expression trees are well-suited for representing mathematical expressions, they may not be the most efficient or practical representation for all types of computations or data structures. In some cases, other data structures may be more suitable.

**HUFFMAN TREE**

A Huffman tree is a binary tree used in Huffman coding, a compression algorithm. It assigns variable-length binary codes to symbols (e.g., characters), with shorter codes for more frequent symbols. The tree is constructed so that shorter codes are closer to the root, optimizing compression for frequently occurring symbols. This method is widely employed in file compression algorithms to reduce data size without loss of information.

**Purpose:**

* Huffman trees are primarily used for data compression, specifically in Huffman coding algorithms.
* The goal is to minimize the total length of the encoded data by assigning shorter codes to more frequent symbols.

**Structure:**

* Huffman trees are binary trees where each leaf node represents a symbol (like a character) and has an associated frequency.
* Internal nodes do not store symbols; they represent the merging of two nodes with the lowest frequencies.
* The tree is constructed in a way that frequently occurring symbols have shorter binary codes.

**Properties:**

* The binary codes assigned to symbols in a Huffman tree are prefix-free, meaning no code is a prefix of another.
* This property allows for unambiguous decoding of the encoded data.

**Operation:**

1. **Construction of Huffman Tree:**

* **Frequency Counting:** Determine the frequency of each symbol (e.g., characters in a text file) in the input data.
* **Priority Queue:** Create a priority queue or min-heap based on the symbol frequencies.
* **Tree Building:** Repeatedly extract the two nodes with the lowest frequencies from the priority queue, combine them into a new internal node with a frequency equal to the sum of the two nodes, and insert the new node back into the priority queue. Continue until only one node (the root) remains.

1. **Encoding:**

* **Traversing the Tree:** Starting from the root, traverse the Huffman tree to each leaf node, assigning binary codes (0 or 1) based on the path taken. Left branches typically represent 0, and right branches represent 1.
* **Code Generation:** Generate a unique binary code for each symbol based on the traversal path from the root to its corresponding leaf.

1. **Decoding:**

* **Traversing the Tree:** Start at the root and traverse the Huffman tree by following the encoded binary sequence. For each 0, move to the left child; for each 1, move to the right child.
* **Leaf Node Detection:** When a leaf node is reached, decode the corresponding symbol and restart the traversal from the root for the next encoded sequence.

**Advantages of Huffman Coding:**

1. **Efficient Compression:** Huffman coding provides efficient compression by assigning shorter codes to more frequently occurring symbols, reducing the overall size of the compressed data.
2. **Variable-Length Codes:** Unlike fixed-length codes, Huffman codes are variable in length, allowing for more flexibility in representing symbols and achieving better compression ratios.
3. **Lossless Compression:** Huffman coding is a lossless compression technique, meaning that the original data can be perfectly reconstructed from the compressed data, ensuring no loss of information.

**Disadvantages of Huffman Coding:**

1. **Variable-Length Codes Overhead:** While variable-length codes are advantageous for compression, they introduce overhead in terms of encoding and decoding complexity compared to fixed-length codes.
2. **Need for Codebook Transmission:** The compressed data requires a codebook to decode, which needs to be transmitted along with the compressed file. This additional overhead might be a disadvantage in certain applications.
3. **Symbol Frequency Dependency:** The effectiveness of Huffman coding relies on the accurate estimation of symbol frequencies. If the frequency distribution changes, the efficiency of the Huffman code may decrease, potentially impacting compression performance.

**HASHING AND HASH TABLES**

Hashing and hash tables are closely related concepts in computer science, but they are not the same thing.

Hashing is the process of converting an input value (such as a string of characters or a number) into a fixed-size output value called a hash value or hash code. The hash function is used to generate the hash value, and it is designed to be deterministic, meaning that the same input value will always produce the same hash value. Hash functions are used in a variety of applications, including cryptography, data integrity, and data storage.

A hash table is a data structure that uses hashing to store key-value pairs. The hash function is used to map a key to a location in the hash table, where the corresponding value is stored. This makes it possible to retrieve the value associated with a key very quickly, even if the hash table is very large. Hash tables are used in a variety of applications, including dictionaries, caches, and symbol tables.

* **Simple definition of Hashing**

The process of converting an input value into a fixed-size output value called a hash value

**Purpose**

To generate a hash value that can be used to identify or verify data

**Application**

Cryptography, data integrity, data storage

* **Simple Hash Table definition**

A data structure that uses hashing to store key-value pairs

**Purpose**

To store and retrieve key-value pairs efficiently

**Application**

Dictionaries, caches, symbol tables

**Advantages of Hashing and Hash Tables:**

1. **Fast Retrieval:** Hashing provides fast retrieval of data. When properly implemented, the time complexity for searching, inserting, and deleting elements in a hash table is typically O(1), making it efficient for large datasets.
2. **Constant Time Average Case:** In an ideal scenario, where collisions are minimized, the average time complexity for basic operations on a well-designed hash table remains constant, providing consistent and efficient performance.
3. **Space Efficiency:** Hash tables can be space-efficient compared to other data structures. They allow for dynamic resizing, enabling the allocation of memory based on the actual number of elements in the table.

**Disadvantages of Hashing and Hash Tables:**

1. **Collision Handling:** Collisions, where two different keys hash to the same location, must be addressed. Resolving collisions often involves additional operations (such as chaining or open addressing), which can impact performance.
2. **Hash Function Dependency:** The effectiveness of a hash table relies heavily on the quality of the hash function. A poorly designed hash function can lead to a higher rate of collisions, undermining the efficiency of the table.
3. **Deterministic Hashing:** Hashing introduces a deterministic aspect, meaning that identical inputs will always produce the same hash value. In certain scenarios, this predictability may be exploited for malicious purposes, such as denial-of-service attacks or hash collisions in security-sensitive applications. Techniques like salting are used to mitigate these risks.

**Hash functions**

A hash function is a mathematical function that takes an input (or 'message') and produces a fixed-size string of characters, which is typically a hash value or hash code. The primary purpose of a hash function is to uniquely represent data in a way that facilitates efficient retrieval, storage, or comparison of information.

**Characteristics of a Hash Function**

1. **Deterministic:** A hash function is deterministic, meaning that for a given input, it will always produce the same hash value.
2. **Fixed Output Size:** The output of a hash function has a fixed size, regardless of the size of the input. This is crucial for creating consistent-length hash codes, making it easier to manage and compare hash values.
3. **Efficient Computation:** Hash functions are designed for quick computation. The time complexity of a good hash function is generally proportional to the size of the input.
4. **Uniform Distribution:** A good hash function aims for a uniform distribution of hash values across the range of possible outputs. This helps minimize the likelihood of collisions, where different inputs produce the same hash value.
5. **Irreversibility:** Hash functions are designed to be one-way, meaning it should be computationally infeasible to reverse the process and retrieve the original input from its hash value. This property is crucial for security applications.
6. **Avalanche Effect:** A small change in the input should produce a significantly different hash value. This property ensures that similar inputs don't result in similar hash codes, enhancing the security and effectiveness of the hash function.

Hash functions have various applications, including data integrity verification, password storage, digital signatures, and hash tables used in data structures like hash maps. The choice of an appropriate hash function depends on the specific requirements of the application.