1. Array

- **Description**: A fixed-size collection of elements of the same type.
- **Functions**:
  - Access elements using indices (e.g., `arr[i]`).
  - Traverse using loops.
- **Best Use Cases**:
  - When the size is known at compile-time and does not change.
  - When fast access is required.

2. Vector

- **Description**: A dynamic array that can resize itself automatically.
- **Functions**:
  - `push_back(value)`: Adds an element to the end.
  - `pop_back()`: Removes the last element.
  - `size()`: Returns the number of elements.
  - Random access using indices.
- **Best Use Cases**:
  - When you need dynamic sizing.
  - When frequently adding/removing elements at the end.

3. Stack

- **Description**: A LIFO (Last In, First Out) data structure.
- **Functions**:
  - `push(value)`: Adds an element to the top.
  - `pop()`: Removes the top element.
  - `top()`: Returns the top element without removing it.

cpp Copy code

```cpp
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;

    // Adding elements
    s.push(1);
    s.push(2);
    s.push(3);

    // Accessing and removing elements
    while (!s.empty()) {
        std::cout << s.top() << " "; // Output: 3 2 1
        s.pop();
    }

    return 0;
}
```

- **Best Use Cases**:
  - Function call management (recursion).
  - Undo mechanisms in applications.

4. Queue

```cpp
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;

    // Adding elements
    q.push(1);
    q.push(2);
    q.push(3);

    // Accessing and removing elements
    while (!q.empty()) {
        std::cout << q.front() << " "; // Output: 1 2 3
        q.pop();
    }

    return 0;
}
```

- **Description**: A FIFO (First In, First Out) data structure.
- **Functions**:
  - `enqueue(value)`: Adds an element to the back.
  - `dequeue()`: Removes the front element.
  - `front()`: Returns the front element without removing it.
- **Best Use Cases**:
  - Task scheduling.
  - Print job management.

Characteristics of a Priority Queue

- Elements are ordered based on priority rather than the order of insertion.
- The highest (or lowest, depending on the implementation) priority element is always at the front.
- It is commonly implemented using a binary heap.

```cpp
#include <iostream>
#include <queue>
#include <vector>

int main() {
    // Create a max priority queue (default behavior)
    std::priority_queue<int> maxHeap;

    // Inserting elements
    maxHeap.push(10);
    maxHeap.push(30);
    maxHeap.push(20);

    std::cout << "Max-Heap Elements: ";
    // Displaying elements in priority order
    while (!maxHeap.empty()) {
        std::cout << maxHeap.top() << " "; // Output: 30 20 10
        maxHeap.pop();
    }
    std::cout << std::endl;

    // Create a min priority queue
    std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;

    // Inserting elements
    minHeap.push(10);
    minHeap.push(30);
    minHeap.push(20);

    std::cout << "Min-Heap Elements: ";
    // Displaying elements in priority order
    while (!minHeap.empty()) {
        std::cout << minHeap.top() << " "; // Output: 10 20 30
        minHeap.pop();
    }
    std::cout << std::endl;

    return 0;
}
```

5. Set

```cpp
#include <iostream>
#include <set>

int main() {
    std::set<int> mySet;

    // Adding elements
    mySet.insert(1);
    mySet.insert(2);
    mySet.insert(1); // Duplicate, will be ignored

    // Accessing elements
    for (const int& val : mySet) {
        std::cout << val << " "; // Output: 1 2
    }

    return 0;
}
```

- **Description**: A collection of unique elements, implemented as a balanced binary tree.
- **Functions**:
  - `insert(value)`: Adds a unique element.
  - `erase(value)`: Removes an element.
  - `find(value)`: Checks if an element exists.
- **Best Use Cases**:
  - When uniqueness is required.
  - When fast search, insert, and delete operations are needed.

7. Unordered Map

- **Description**: A collection of key-value pairs stored in a hash table.
- **Functions**:
  - `insert(key, value)`: Adds a key-value pair.
  - `erase(key)`: Removes the key-value pair by key.
  - `find(key)`: Finds the value associated with a key.
- **Best Use Cases**:
  - When fast access is required without maintaining order.

8. Linked List

- **Description**: A collection of nodes where each node contains data and a pointer to the next node.
- **Functions**:
  - `push_front(value)`: Adds an element at the beginning.
  - `push_back(value)`: Adds an element at the end.
  - `pop_front()`: Removes the first element.
  - `pop_back()`: Removes the last element.
- **Best Use Cases**:
  - When frequent insertions and deletions are required.
  - When the size of the data structure is unknown.

```cpp
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<int, std::string> myMap;

    // Adding key-value pairs
    myMap[1] = "one";
    myMap[2] = "two";

    // Accessing values by key
    for (const auto& pair : myMap) {
        std::cout << pair.first << " -> " << pair.second << std::endl; // Output: 
    }

    return 0;
}
```

Algorithms

1. A* Search Algorithm
2. Binary Tree Traversals (Inorder, Preorder, Postorder)
3. Quick select
4. Kadane’s Algorithm
5. Flood Fill Algorithm
6. Lee Algorithm

1. Sorting Algorithms

- **Bubble Sort**: Simple but not efficient. Good for understanding sorting basics.
- **Selection Sort**: Basic and easy to implement.
- **Insertion Sort**: Efficient for small datasets.
- **Merge Sort**: Divide-and-conquer approach; efficient for larger datasets.
- **Quick Sort**: Faster on average than Merge Sort but not stable.
- **Heap Sort**: Uses a heap data structure; efficient and in-place.

2. Searching Algorithms

- **Linear Search**: Simple and works for unsorted data.
- **Binary Search**: Works on sorted data; divide-and-conquer.
- **Interpolation Search**: Optimized for uniformly distributed sorted data.

3. Graph Algorithms

- **Breadth-First Search (BFS)**: Level-order traversal; shortest path in unweighted graphs.
- **Depth-First Search (DFS)**: Explores as far as possible along each branch.
- **Dijkstra’s Algorithm**: Shortest path in weighted graphs.
- **Floyd-Warshall Algorithm**: All-pairs shortest paths.
- **Kruskal’s Algorithm**: Minimum spanning tree (MST).
- **Prim’s Algorithm**: Another MST approach.
- **Bellman-Ford Algorithm**: Shortest path with negative weights.

4. Dynamic Programming (DP) Algorithms

- **Fibonacci Sequence**: Simple example of overlapping subproblems.

- **Longest Common Subsequence (LCS)**: Finds the longest subsequence common to two sequences.
- **Knapsack Problem**: Optimal selection of items with weight and value constraints.
- **Coin Change Problem**: Minimum number of coins to make a given amount.
- **Matrix Chain Multiplication**: Optimal way to parenthesize a matrix product.

5. Divide and Conquer

- **Binary Search**: Core of divide-and-conquer.
- **Merge Sort**: Combines divided parts efficiently.
- **Quick Sort**: Divide, conquer, and combine.

6. Greedy Algorithms

- **Huffman Encoding**: Compression algorithm.
- **Activity Selection Problem**: Maximizes the number of activities.
- **Job Scheduling Problem**: Optimizes job completion based on deadlines and profits.

7. Backtracking Algorithms

- **N-Queens Problem**: Place N queens on an NxN chessboard.
- **Sudoku Solver**: Fills the grid based on rules.
- **Subset Sum Problem**: Finds subsets that match a given sum.

8. String Algorithms

- **KMP Algorithm**: Pattern searching.
- **Rabin-Karp Algorithm**: Efficient for multiple pattern searches.
- **Z Algorithm**: String matching.
- **Trie Construction**: Efficient prefix matching.

9. Miscellaneous Important Algorithms

- **Union-Find (Disjoint Set)**: For connected components and Kruskal's MST.
- **Topological Sorting**: For directed acyclic graphs (DAG).
- **Sliding Window Technique**: Optimizes certain problems with fixed or variable window sizes.
- **Two-Pointer Technique**: Efficient for searching pairs/triples in sorted arrays.