Array ADT

This C++ file is a fantastic, hands-on demonstration of various essential algorithms performed on arrays. It's structured as a collection of common problems and their solutions, making it an ideal study guide for mastering array manipulations. The foundation of the entire file is a custom struct array, which acts as a simple Abstract Data Type (ADT) to bundle an array with its **total size** (capacity) and its current **length** (number of elements). This is a core concept in data structures.

The file covers a wide spectrum of operations. It begins with the fundamentals: **adding** (append, insert), **removing** (delete), and **searching** for elements, showcasing both the basic **Linear Search** and the more efficient **Binary Search** for sorted arrays (in both iterative and recursive forms). It then moves on to common utility functions like getting/setting values and calculating aggregates (max, min, sum, avg).

From there, the code explores more complex algorithms. It demonstrates array manipulation techniques like **reversing** and **shifting**. It also delves into algorithms for sorted arrays, such as inserting an element while maintaining order and merging two sorted arrays. This naturally leads to set operations like finding the **Union**, **Intersection**, and **Difference** between two arrays.

Finally, the file tackles a variety of common interview-style problems. It provides multiple, distinct approaches for finding missing elements, finding duplicate elements, and finding pairs of elements that add up to a target sum. A key theme here is comparing the efficiency of different solutions: brute-force vs. two-pointer techniques for sorted data, and the powerful use of a **hash map** (simulated with an auxiliary array) to solve problems on unsorted data in linear time.

**Demo**

This initial code sets up a basic program to demonstrate the custom array structure. It allows the user to define the array's size dynamically at runtime, allocate memory for it on the heap, and then fill it with a specified number of elements.

* struct array{ int \*A; ... }; This defines the **blueprint** for our array ADT. The key part is int \*A, a pointer that will hold the address of the dynamically allocated array on the heap.
* arr.A = (int \*) malloc(arr.size\*(sizeof(int))); This is the C-style way to **allocate memory on the heap**. It requests a block of memory large enough to hold arr.size integers and stores the starting address of this block in the pointer arr.A.
* scanf("%d", &arr.size); This line reads an integer from the user's input and stores it in the size member of the arr structure, making the array's capacity dynamic.

**Append & Insert operations**

This section demonstrates two fundamental ways to add elements to an array: append adds an element to the very end, while insert places an element at a specific index, requiring other elements to be shifted.

* void append(struct array \*arr, int x){ ... } The append function is simple. It places the new value x at the next available position, which is indexed by the current length, and then increments the length.
* void insert(struct array \*arr, int index, int x){ ... } The insert function is more complex. It must first make space for the new element.
  + for(int i = arr->length; i>index; i--){ arr->A[i] = arr->A[i-1]; } This loop is the core of the insertion. It starts from the end of the array and **shifts every element** from the target index one position to the right.
  + arr->A[index] = x; Once the space is created, the new value x is placed at the desired index.

**Delete operation**

This code shows how to remove an element from a specific index in the array. Since arrays are contiguous blocks of memory, you can't just create a "hole." Instead, you must shift all subsequent elements to the left to cover the deleted element's spot.

* for(int i = index; i<arr->length-1; i++){ arr->A[i] = arr->A[i+1]; } This loop is the key to the deletion. It starts at the index to be deleted and copies the value from the next element (i+1) into the current position (i), effectively **shifting all elements left** by one spot.
* arr->length--; After shifting, the logical length of the array is reduced by one to reflect the removal.

**Linear Search**

This section implements a **Linear Search**, which sequentially checks each element of an array until it finds a target value or reaches the end. It also includes an optimization to make future searches for the same element faster.

* for(int i = 0; i<arr->length; i++){ ... } This is the standard **sequential search**. The loop iterates from the first element to the last.
* if(arr->A[i] == x){ ... } Inside the loop, this condition checks if the current element matches the target x.
* swap(&arr->A[i], &arr->A[i-1]); This is an optimization called **Transposition**. After an element is found, it's swapped with the element before it. This makes it slightly quicker to find again if it's searched for frequently. Another common optimization (commented out) is Move-to-Front, where the found element is swapped with the very first element.

**Binary search**

This code demonstrates **Binary Search**, a highly efficient algorithm for finding an element in a **sorted** array. It works by repeatedly dividing the search interval in half. Two versions are provided: an iterative one using a loop and a recursive one.

* mid = (l+h)/2; This is the core of the algorithm. It calculates the **middle index** of the current search range (from low l to high h).
* if(key<arr.A[mid]){ h = mid-1; } else{ l = mid+1; } This logic **eliminates half** of the remaining search space in each step. If the key is smaller than the middle element, the search continues in the left half; otherwise, it continues in the right half.
* return RBinarySearch(arr, l, mid-1, key); This line shows the **recursive** implementation. Instead of using a while loop, the function calls itself with the updated search range (l to mid-1 or mid+1 to h).

**Get, Set, Max, Min, Sum, Avg operations**

This section contains a suite of common utility functions for performing basic operations and calculations on the array data.

* int get(struct array arr, int index){ ... } This function **retrieves** the value at a given index. It includes a check to ensure the index is valid.
* int set(struct array \*arr, int index, int value){ ... } This function **updates** the value at a given index with a new value.
* int max(struct array arr){ ... } This function finds the **largest element** by iterating through the array and keeping track of the maximum value seen so far.
* float avg(struct array arr){ ... } This function calculates the **average** of the array elements. It cleverly reuses the sum() function to get the total and then divides by the length. The (float) cast is important to ensure the division results in a decimal value.

**Reversing an array**

This code demonstrates two different methods for reversing the order of elements in an array.

* void reverse(struct array \*arr){ ... } This is the first method, which uses an **auxiliary array** (B).
  + B[i] = arr->A[j]; It copies elements from the original array A into B in reverse order.
  + arr->A[i] = B[i]; Finally, it copies the reversed elements from B back into A. This method is simple to understand but requires extra memory.
* void swapping(struct array \*arr){ ... } This is the second, more efficient method, which performs an **in-place reversal** using swapping.
  + for(int k = 0; k<arr->length/2; k++){ ... } The loop only needs to run up to the middle of the array.
  + temp = arr->A[i]; arr->A[i] = arr->A[j]; arr->A[j] = temp; It swaps the first element with the last, the second with the second-to-last, and so on, until the two pointers (i and j) meet in the middle.

**Left shifting of an array**

This section implements a circular left shift (also known as a left rotation). Each element is moved one position to the left, and the first element is moved to the end of the array.

* int a = arr->A[0]; First, the value of the **first element is saved** in a temporary variable so it isn't lost.
* arr->A[i] = arr->A[j]; This loop **shifts** each element one position to the left by copying the value from its right neighbor.
* arr->A[arr->length-1] = a; Finally, the saved first element is placed at the **end of the array**, completing the rotation.

**Inserting an element in sorted array, Checking if the array is sorted or not, Negative numbers on left side**

This code block contains three distinct and useful algorithms.

* void insertSort(struct array \*arr, int x){ ... } This function inserts an element into a **sorted array** while maintaining the sorted order.
  + while(i>=0 && arr->A[i] > x){ arr->A[i+1] = arr->A[i]; i--; } It starts from the end and shifts all elements that are greater than the new element x one position to the right. This creates a space at the correct sorted position.
* bool isSorted(struct array arr){ ... } This function **checks if an array is sorted** in non-decreasing order. It iterates through the array and returns false as soon as it finds an element that is greater than the one following it.
* void negativeCheck(struct array \*arr1){ ... } This function **partitions the array** so that all negative numbers are on the left side and all positive numbers are on the right.
  + while(i<j){ ... } It uses a **two-pointer approach**. Pointer i moves from the left, skipping over negatives, and pointer j moves from the right, skipping over positives. When they both stop, they have found a positive number on the left and a negative number on the right, which are then swapped.

**Merging two sorted arrays**

This function takes two sorted arrays and merges them into a single, new sorted array. This is a core component of the Merge Sort algorithm.

* struct array \*arr3 = (struct array \*)malloc(sizeof(struct array)); A new array structure arr3 is **dynamically allocated** on the heap to store the merged result.
* while(i<arr1.length && j<arr2.length){ ... } This is the main loop. It uses three index pointers (i for arr1, j for arr2, and k for arr3). It repeatedly compares the elements at arr1[i] and arr2[j] and copies the smaller of the two into arr3.
* while (i < arr1.length) { ... } After the main loop finishes, one of the arrays might still have elements left. These final two while loops are responsible for **copying any remaining elements** into arr3.

**Union, Intersection and Difference of 2 arrays**

This section implements three fundamental **set operations** for sorted arrays. The logic is similar to merging but with different conditions for adding elements to the result array.

* struct array \*ArrayUnion( ... ){ ... } This function computes the **Union**. It includes all elements from both arrays but ensures that common elements are included only once.
  + else{ arr3->A[k++] = arr1.A[i++]; j++; } This else block handles the case where elements are equal. It copies the element once and increments both i and j to avoid duplicates.
* struct array \*ArrayIntersection( ... ){ ... } This function computes the **Intersection**. It includes only the elements that are present in *both* arrays.
  + else if(arr1.A[i] == arr2.A[j]){ arr3->A[k++] = arr1.A[i++]; j++; } This is the key condition. An element is added to the result array only when it's found to be equal in both input arrays.
* struct array \*ArrayDifference( ... ){ ... } This function computes the **Difference** (arr1 - arr2). It includes only the elements that are present in arr1 but *not* in arr2.
  + if(arr1.A[i]<arr2.A[j]){ arr3->A[k++] = arr1.A[i++]; } This condition adds an element from arr1 to the result only if it's smaller than the current element in arr2, meaning it can't possibly exist in the rest of arr2.

**Finding single missing number in a series array**

This code shows two methods for finding a single missing element in a sorted array that is supposed to be a contiguous sequence of numbers.

* void SingleMissing1(int arr[]){ ... } This method works when the sequence starts from **1**.
  + int sum = (10\*(10+1))/2; It uses the mathematical formula for the sum of the first n natural numbers to find what the sum *should* be.
  + printf("The missing number is: %d\n", sum-realSum); It then subtracts the actual sum of the array elements from the expected sum. The result is the missing number.
* void SingleMissing2(int arr[]){ ... } This method works for any sequence, even if it doesn't start at 1.
  + int diff = arr[0] - 0; It calculates the expected difference between an element's value and its index (value - index).
  + if(arr[i]-i != diff){ ... } It iterates through the array, and the first element that does not have this expected difference reveals the location of the missing number.

**Finding multiple missing numbers in a series array**

This code expands on the logic from SingleMissing2 to find all the missing numbers in a sequence, not just the first one.

* while(diff<arr[i]-i){ printf("%d ", diff+i); diff++; } When a discrepancy is found (i.e., arr[i]-i != diff), this inner while loop runs. It keeps printing the expected missing numbers (diff+i) and incrementing diff until the expected difference matches the actual difference again. This effectively "fills in" all the gaps.

**Finding missing elements in unsorted array using hashmap**

This section demonstrates how to find missing elements in an **unsorted** array by using a simple hash map (also known as a frequency map or hash table).

* int arr2[12] = {0}; An auxiliary array arr2 is created and initialized to all zeros. This array will act as our **hash map**. The index of arr2 will correspond to a number, and the value at that index will tell us if the number is present.
* arr2[arr[i]]++; This is the **hashing step**. The code iterates through the input array arr. For each element arr[i], it uses that element's value as an index into the hash map arr2 and increments the count at that index. This "marks" the number as present.
* if(arr2[i] == 0){ printf("%d ", i); } Finally, the code loops through the hash map arr2. Any index i where the value is still 0 corresponds to a number that was never present in the original input array.

**Finding duplicate values and counting no of times it occured**

This code finds and counts consecutive duplicate values in a **sorted** array.

* if(arr[i] == arr[i+1]){ ... } The outer loop iterates through the array and checks if an element is the same as the one immediately following it. This is the starting point of a sequence of duplicates.
* while(arr[j] == arr[i]) j++; Once a duplicate is found, this inner while loop continues moving forward as long as the elements are the same. The pointer j ends up at the position just after the sequence of duplicates.
* printf("%d appeared %d times\n", arr[i], j-i); i=j-1; The number of occurrences is calculated by j-i. The outer loop's counter i is then updated to j-1 to skip over the duplicates that have already been counted.

**Finding duplicate values and counting no of times it occured using hashmap**

This section uses the hash map technique to find and count duplicates. This approach works for both **sorted and unsorted** arrays.

* h[arr[i]]++; The code iterates through the input array arr and uses each element's value as an index into the hash map h, incrementing the count at that index. This builds a **frequency count** for every number.
* if(h[i]>1){ printf("%d appeared %d times\n", i, h[i]); } Finally, it iterates through the hash map h. Any index i where the count is greater than 1 represents a number that was a duplicate in the original array.

**Finding duplicate values and counting no of times it occured in unsorted array**

This code provides a simple, brute-force method to find duplicates in an **unsorted** array without using extra space for a hash map.

* for(int i=0; i<9; i++){ ... } The outer loop picks an element.
* for(int j=i+1; j<10; j++){ if(arr[i] == arr[j]){ ... } } The inner loop compares the chosen element with all subsequent elements in the array to find matches.
* arr[j] = -1; This is a clever trick. When a duplicate is found, its value is changed to -1 (or any other value outside the expected range of data). This prevents it from being counted again in a later iteration of the outer loop.

**Finding duplicate values and counting no of times it occured in unsorted array using hashmap**

This is another example of the powerful hash map technique, specifically applied to finding duplicates in an unsorted array. The logic is identical to the previous hash map example for duplicates.

* h[arr[i]]++; Builds a **frequency map** where h[value] = count.
* if(h[i] > 1){ ... } Iterates through the frequency map to find which numbers appeared more than once.

**Finding a pair with sum K (a+b=k)**

This code finds pairs of numbers in an **unsorted** array that add up to a target value k (in this case, 10). This is a brute-force approach with O(n²) time complexity.

* for(int i = 0; i<9; i++){ ... } The outer loop selects the first number of the potential pair.
* for(int j = i+1; j<10; j++){ ... } The inner loop checks all subsequent numbers to see if they form a pair with the number selected by the outer loop.
* if(arr[i]+arr[j]==10){ ... } If the sum of the two numbers equals the target, the pair is printed.

**Finding a pair with sum K (a+b=k) using hashmap**

This section provides a much more efficient, O(n) solution to the pair-sum problem for **unsorted** arrays using a hash map.

* if(h[10-arr[i]] != 0 && 10-arr[i]>0){ ... } This is the key insight. For each element arr[i], the code calculates the **complement** it needs to reach the sum (i.e., 10 - arr[i]). It then checks the hash map to see if this complement has been encountered *before*. If h[complement] is not zero, it means the complement exists in the array, and a pair has been found.
* h[arr[i]]++; After checking for a pair, the current element arr[i] is added to the hash map. This ensures it can be found as a complement for future elements.

**Finding a pair with sum K (a+b=k) in sorted array**

This code demonstrates a highly efficient, O(n) solution to the pair-sum problem that works specifically for **sorted** arrays.

* int i = 0; int j = 9; It uses a **two-pointer approach**. One pointer i starts at the beginning (the smallest element), and another pointer j starts at the end (the largest element).
* while(i<j){ ... } The pointers move towards each other until they cross.
  + if(arr[i] + arr[j] == sum){ ... } If the sum of the elements at the pointers equals the target, a pair is found. Both pointers are then moved to look for other potential pairs.
  + else if(arr[i] + arr[j] > sum){ j--; } If the sum is too large, the larger element at j must be replaced by a smaller one, so j is decremented.
  + else i++; If the sum is too small, the smaller element at i must be replaced by a larger one, so i is incremented.

**Finding min and max in single scan**

This final section shows an efficient algorithm to find both the minimum and maximum values in an array in just a single pass.

* int min = arr[0]; int max = arr[0]; The min and max variables are **initialized** with the value of the first element.
* for(int i = 1; i<10; i++){ ... } The loop then iterates through the *rest* of the array, starting from the second element.
* if(arr[i] < min){ min = arr[i]; } else if(arr[i] > max){ max = arr[i]; } For each element, it's compared first to the current min. If it's smaller, min is updated. If it's not smaller, it's then compared to the current max. If it's larger, max is updated. This avoids redundant comparisons and finds both values in one go.