Dynamic Arrays, Linear and Binary Search

Objective:

The objective of this experiment to get familiar with

1. Arrays and its role in data structure
2. Storage of data in Row major order and column major order.
3. Safe Arrays
4. Jagged Array
5. Implementation of linear and binary searching techniques.

Introduction:

An array is a series of elements of the same data type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. int arr[5]

For example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). These values can be accessed using the same identifier, with the proper index. Multidimensional arrays can be described as "arrays of arrays". For example, a bi-dimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.

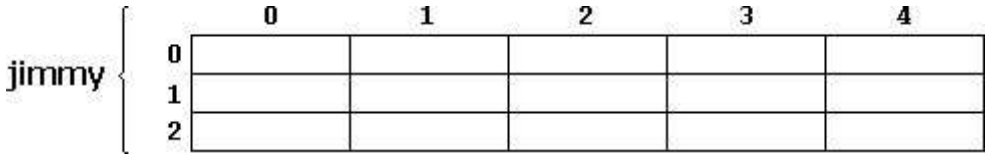


Figure 1.1 2D Arrays

Jimmy represents a bi-dimensional array of 3 per 5 elements of type int as shown in above

Figure 1.1. The C++ syntax for this is: int jimmy[3][5]

In addition, for example, the way to reference the second element vertically and fourth horizontally in an expression would be: jimmy[1][3]

Dynamic Memory Allocation for arrays:

Memory in your C++ program is divided into two parts

1. **The stack** − All variables declared inside the function will take up memory from the stack.

2. **The heap** − this is unused memory and can be used to allocate the memory dynamically during program execution.

A dynamic array is an array with a big improvement, that is, automatic resizing. One limitation of static arrays is that they're fixed size, meaning you need to specify the number of elements your array will hold ahead of time. A dynamic array expands as you add more elements. So you don't need to determine the size ahead of time.

Strengths:

1. **Fast lookups.** Just like static arrays, retrieving the element at a given index takes O(1) time.
2. **Variable size.** You can add as many items as you want, and the dynamic array will expand to hold them.
3. **Cache-friendly**. Just like static arrays, dynamic arrays place items right next to each other in memory, making efficient use of caches.

Weaknesses:

1. **Slow worst-case appends**. Usually, adding a new element at the end of the dynamic array takes O(1) time. But if the dynamic array doesn't have any room for the new item, it'll need to expand, which takes O(n) time.
2. **Costly inserts and deletes.** Just like static arrays, elements are stored adjacent to each other. So adding or removing an item in the middle of the array requires "scooting over" other elements, which takes O(n) time.

Factors impacting the performance of Dynamic Arrays:

The array's initial size and its growth factor determine its performance. Note the following points:

1. If an array has a small size and a small growth factor, it will keep on reallocating memory more often. This will reduce the performance of the array.
2. If an array has a large size and a large growth factor, it will have a huge chunk of unused memory. Due to this, resize operations may take longer. This will reduce the performance of the array.

The new Keyword:

In C++, we can create a dynamic array using the new keyword. The number of items to be allocated is specified within a pair of square brackets. The type name should precede this. The requested number of items will be allocated. Syntax:

```
int *ptr1 = new
int;
int *ptr1 = new int[5];
int *array { new int[10]{}};
int *array { new int[10]{1,2,3,4,5,6,7,8,9,10}};
```

Resizing Arrays:

The length of a dynamic array is set during the allocation time. However, C++ doesn't have a built-in mechanism for resizing an array once it has been allocated. You can, however, overcome this challenge by allocating a new array dynamically, copying over the elements, and then erasing the old array.

Dynamically Deleting Arrays:

A dynamic array should be deleted from the computer memory once its purpose is fulfilled. The delete statement can help you accomplish this. The released memory space can then be used to hold another set of data. However, even if you do not delete the dynamic array from the computer memory, it will be deleted automatically once the program terminates. Syntax:

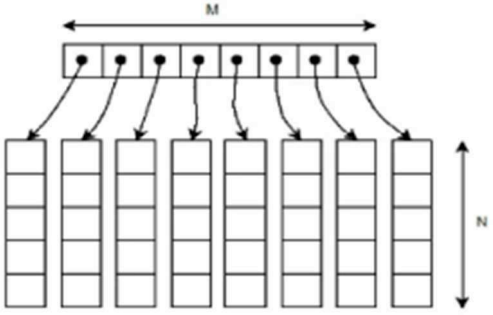
```
delete ptr;
delete [] arr;
```

NOTE: To delete a dynamic array from the computer memory, you should use delete[], instead of delete. The [] instructs the CPU to delete multiple variables rather than one variable. The use of delete instead of delete[] when dealing with a dynamic array may result in problems.
Examples of such problems include memory leaks, data corruption, crashes, etc. Example: Single Dimensional Array:

```
#define N 10
int main(){
        // dynamically allocate memory of size 5 and assign values to allocated memory
int *array{ new int[5]{ 10, 7, 15, 3, 11 } };
        // print the 1D array de allocate memory
}
```

Two Dimensional Array Using Array of Pointers:

We can dynamically create an array of pointers of size M and then dynamically allocate memory of size N for each row as shown below.



Example:

```
// Dynamically Allocate Memory for 2D Array in C++
```

```
int
main(){
//Enter two dimensions N and M, Dynamically allocate memory to both
int** ary = new int*[N];
for(int i = 0; i < N; ++i)
ary[i] = new int[M];
//fill the arrays, print them, deallocate the memory
}
```

Row-Major Order and Column-Major Order

When dealing with multidimensional arrays in programming, particularly 2D arrays, the elements can be stored in memory in two common ways: Row-Major Order and Column-Major Order.

1. Row-Major Order

In **Row-Major Order**, the array elements are stored row by row. This means that all elements of the first row are stored in consecutive memory locations, followed by the elements of the second row, and so on.

Example: Consider a 2D array:

```
css
Copy code
int arr[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

- **Memory Layout in Row-Major Order:**
  - The elements are stored in memory as: `[1, 2, 3, 4, 5, 6, 7, 8, 9]`

2. Column-Major Order

In **Column-Major Order**, the array elements are stored column by column. This means that all elements of the first column are stored in consecutive memory locations, followed by the elements of the second column, and so on.

Example: Consider the same 2D array:

```
css
Copy code
int arr[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

- **Memory Layout in Column-Major Order:**
  - The elements are stored in memory as: `[1, 4, 7, 2, 5, 8, 3, 6, 9]`

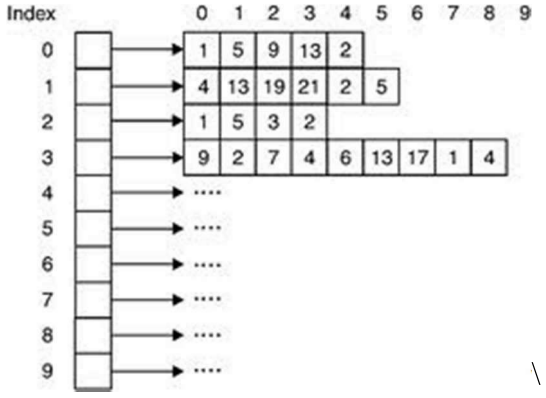
Safe Array:

In C++, there is no check to determine whether the array index is out of bounds. During program execution, an out-of- bound array index can cause serious problems. Also, recall that in C++ the array index starts at 0. Safe array solves the out-of-bound array index problem and allows the user to begin the array index starting at any integer, positive or negative. "Safely" in this context would mean that access to the array elements must not be out of range. ie. the position of the element must be validated prior to access. For example in the member function to allow the user to set a value of the array at a particular location:

```
void set(int pos, Element val){      //set method
        if (pos<0 || pos>=size)
                cout<<"Boundary Error\n";
         else Array[pos] = val;
}
```

Jagged Array

Jagged array is nothing but it is an array of arrays in which the member arrays can be in different sizes.



Example:

```
        int **arr = new int*[3];
         int Size[3], i,j,k;
       for(i=0;i<3;i++){
       cout<<"Row "<<i+1<< " size: ";
       cin>>Size[i];
                arr[i] =new int[Size[i]]; }       for(i=0;i<3;i++){
       for(j=0;j<Size[i];j++){                          cout<<"Enter
row " <<i+1<<" elements: ";                             cin>>*(*(arr
+ i) + j);  }
         }// print the array elements  deallocate memory using delete[] operator
```

Linear Search:

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set. It is the easiest searching algorithm as shown in figure 1.4:

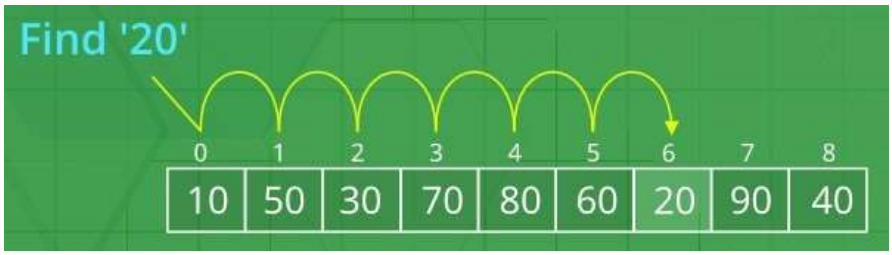


Figure 1.4 Linear

Search Algorithm:

1. Start from the leftmost element of arr[] and one by one compare x with each element of arr[],
2. If x matches with an element, return the index,
3. If x doesn't match with any of the elements, return -1

Code:

```
#include <iostream> using
namespace std;
int search(int arr[], int N, int x){ int i;
  for (i = 0; i < N; i++) if (arr[i] ==
x) return i; return -1;
}
int main(void){
  int arr[] = { 2, 3, 4, 10, 40 }; int x = 10;
  int N = sizeof(arr) / sizeof(arr[0]);
  // Function call
  int result = search(arr, N, x);
  (result == -1)
    ? cout << "Element is not present in array"
    : cout << "Element is present at index " << result; return 0;}
```

Binary Search:

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n) as shown in Figure 1.5.

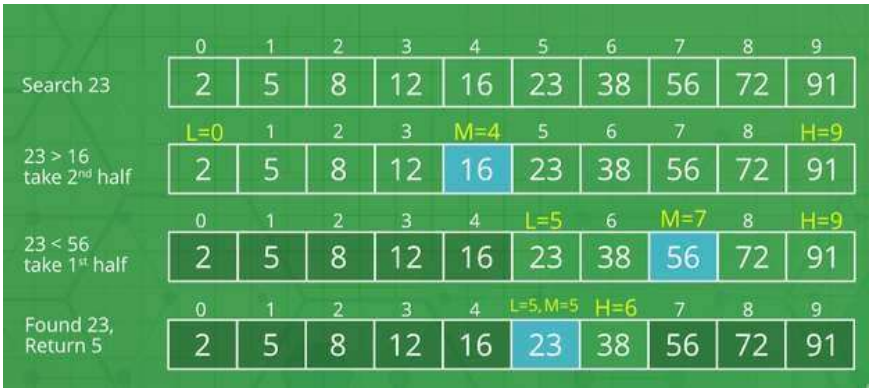


Figure 1.5 Binary Search

Algorithm:

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (x is smaller) recur for the left half.

Data Structure Algorithm & Application (CT-159)
Lab 01

Code: (Iterative Approach)

```
using namespace std;
// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x){     while (l <= r) {
        int m = l + (r - l) / 2;
        // Check if x is present at mid
if (arr[m] == x)             return m;
        // If x greater, ignore left half
if (arr[m] < x)             l = m + 1;
        // If x is smaller, ignore right half
else
            r = m - 1;
    }
    // if we reach here, then element was
    // not present
return -1;
}
int main(void){
    int arr[] = { 2, 3, 4, 10, 40 };
int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
return 0;
}
```

Data Structure Algorithm & Application (CT-159)
Lab 01

Exercise

1. Write a C++ program to copy data of a 2D array in a 1D array using Column Major Order.
2. Write a program to calculate the GPA of students of all subjects of a single semester. Assume all the courses have the same credit hour (let's assume 3 credit hours).

	Data Structure	**Programming for AI**	**Digital Logic Design**	**Probability & Statistics**	**Finance & Accounting**
Ali	3.66	3.33	4.0	3.0	2.66
Hiba	3.33	3.0	3.66	3.0	---
Asma	4.0	3.66	2.66	---	---
Zain	2.66	2.33	4.0	---	---
Faisal	3.33	3.66	4.0	3.0	3.33

3. The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

   For example, for arr = [2,3,4], the median is 3.

   For example, for arr = [2,3], the median is (2 + 3) / 2 = 2.5.

   Implement the MedianFinder class:

   - MedianFinder() initializes the MedianFinder object.
   - void addNum(int num) adds the integer num from the data stream to the data structure.
   - double findMedian() returns the median of all elements so far. Answers within 10-5 of the actual answer will be accepted.

   Example 1:

   Input: ["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
   [[], [1], [2], [], [3], []]
   Output: [null, null, null, 1.5, null, 2.0]
   Explanation
   MedianFinder medianFinder = new MedianFinder();
   medianFinder.addNum(1); // arr = [1] medianFinder.addNum(2); // arr = [1, 2] medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)
   medianFinder.addNum(3); // arr[1, 2, 3] medianFinder.findMedian(); // return 2.0

   Constraints: -105 <= num <= 105

   There will be at least one element in the data structure before calling findMedian. At most 5 * 104 calls will be made to addNum and findMedian.

4. Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1. You must write an algorithm with O(log n) runtime complexity.
   Example 1: Input: nums = [-1,0,3,5,9,12], target = 9, Output: 4
   Explanation: 9 exists in nums and its index is 4
   Example 2: Input: nums = [-1,0,3,5,9,12], target = 2, Output: -1
   Explanation: 2 does not exist in nums so return -1 Constraints:
   1 <= nums.length <= 104

Data Structure Algorithm & Application (CT-159)
Lab 01

   -104 < nums[i], target < 104
   All the integers in nums are unique.
5. nums is sorted in ascending order.You are given an m x n integer matrix with the following two properties: Each row is sorted in non-decreasing order. The first integer of each row is greater than the last integer of the previous row. Given an integer target, return true if target is in matrix or false otherwise. You must write a solution in O(log(m * n)) time complexity.

Example 1:

1	3	5	7
10	11	16	20
23	30	34	60

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3, Output: true
Constraints: m == matrix.length, n == matrix[i].length, 1 <= m, n <= 100, -104 <= matrix[i][j], target <= 104

Lab 01 Evaluation		
Student Name:	**Student ID:**	**Date:**
Rubric	**Marks (25)**	**Remarks by teacher in accordance with the rubrics**
R1		
R2		
R3		
R4		
R5		