Data Structures Lab 02

Course: Data Structures (CL2001) **Semester**: Fall 2024
Instructor: Muhammad Nouman Hanif

Note:
- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Data Structure:

A data structure is essentially a specialized format for organizing, processing, retrieving, and storing data. It provides a way to manage and work with data efficiently based on the requirements of different applications and operations.

Data structures can be categorized into various types based on their characteristics and how they organize data. Here's a breakdown:

1. Primitive Data Structures

Primitive data structures are the basic building blocks of data organization. They are directly operated upon by the machine instructions and including Integer, Float, Character, and Boolean.

2. Non-Primitive Data Structures

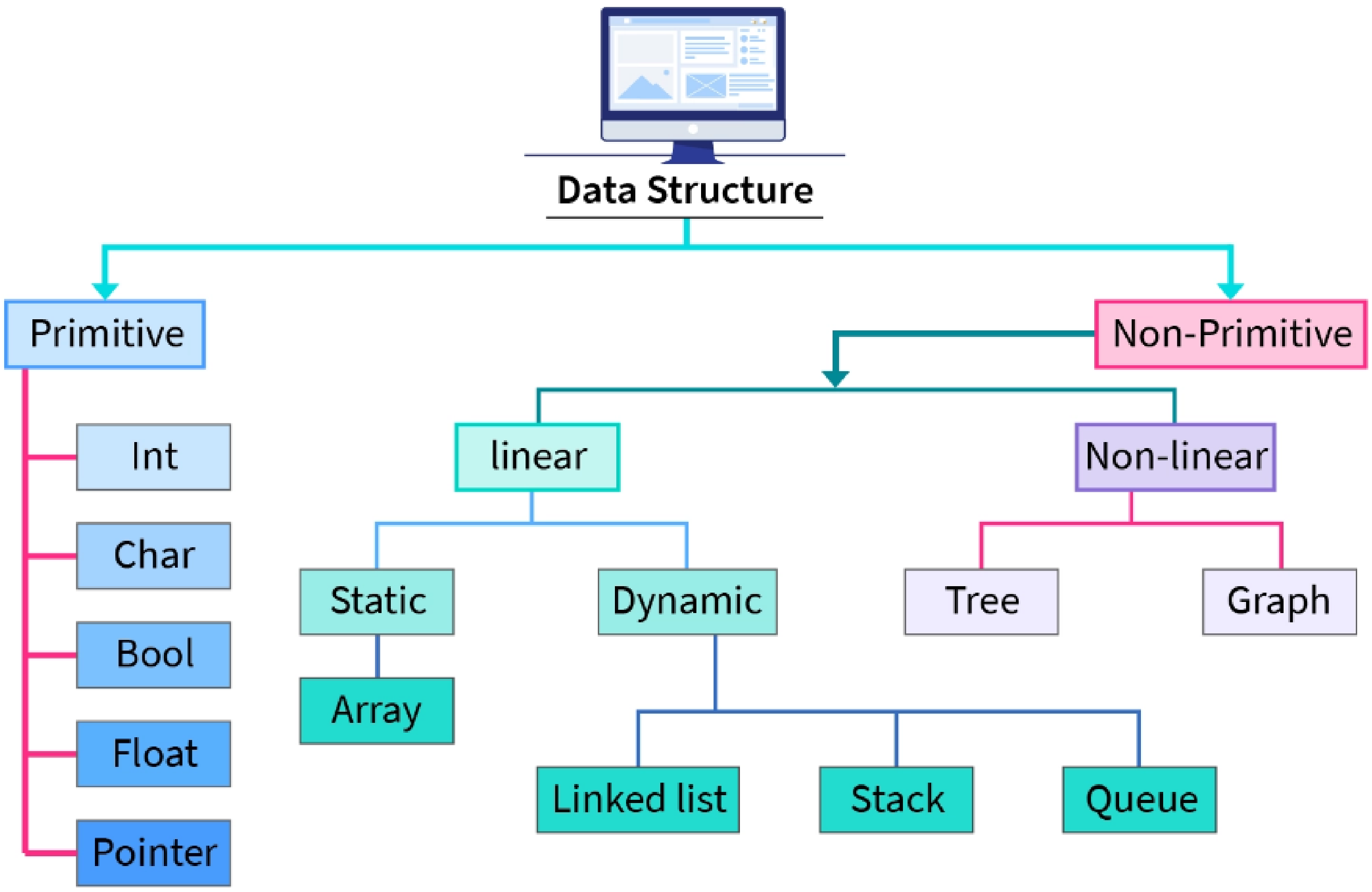
Non-primitive data structures are more complex and are built using primitive data structures. They are categorized into:

Linear Data Structures:

In linear data structures, elements are organized sequentially, and each element is connected to its previous and next element. They are easier to implement and understand but can be less efficient for some operations. Examples including Arrays, Linked Lists, Stacks, and Queues.

Non-Linear Data Structures:

In non-linear data structures, elements are not organized sequentially. They can have hierarchical or interconnected relationships. Examples including Trees, and Graphs.



Arrays:

The simplest type of data structure is a linear (or one-dimensional) array. By a linear array, we mean a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers, usually 1, 2, 3,..., n. If we choose the name A for the array, then the elements of A are denoted by subscript notation i.e. Al, A2, A3 or by the parenthesis notation i.e. A(1), A(2), A(3), …., A(N) or by the bracket notation A[1], A[2], A[3], ..., A[N]. Regardless of the notation, the number K in A[K] is called a subscript and A[K] is called a subscripted variable.

1D & 2D Array:

A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.

Syntax:

```
char name[5];
int mark[5] = {5,11,14,65,85};
int mark[] = {5,11,14,65,85};
```

Like a 1D array, a 2D array is a collection of data cells, all of the same type, which can be given a single name. However, a 2D array is organized as a matrix with a number of rows and columns.

Syntax:

```
     float x[3][4];
     int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
     int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
     int c[2][3] = {1, 3, 0, -1, 5, 9};
```

<u>Dynamic Arrays</u>:

To allocate an array dynamically we use array form of new and delete (new[ ] , delete[ ])

```
#include <iostream>
using namespace std;
int main()
{
    // Initialize a static array
    int array[] = {1, 2, 3};
    cout << array[0];  // Output: 1

    // Allocate a dynamic array using `new`
    Int *dArray = new int[3] {1, 2, 3};

    // Print the values in the dynamic array
    // Output: 2 (accessing the second element using pointer arithmetic)
    cout << *(dArray + 1);
    // Output: 3 (accessing the third element directly)
    cout << dArray[2];
    // Deallocate the dynamic array
    delete[] dArray;
    return 0;
}
```

<u>Dynamic Memory Allocation for arrays:</u>

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 of the program and can be used to allocate the memory dynamically when the program runs.

A dynamic array is an array with a big improvement: **automatic resizing.**

One limitation of 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 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 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 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 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.

Resizing Arrays:

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

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.

<u>Safe Array:</u>

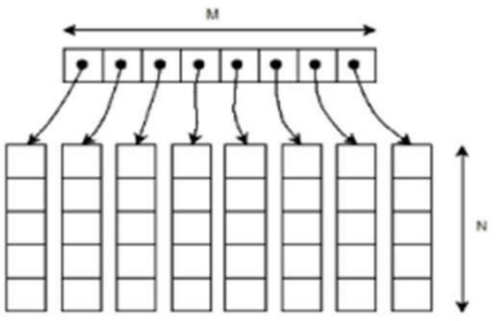
In C++, there is no check to determine whether the array index is out of bounds. During program execution, an out-of-bound array index can cause serious problems. Also, recall that in C++ the array index starts at 0. Safe array solves the out-of-bound array index problem and allows the user to begin the array index starting at any integer, positive or negative.

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){

     if (pos<0 OR pos>=size){    //  if (pos<0 || pos>=size){

        cout<<"Boundary Error\n";
     }

     Else {
               Array[pos] = val;
          }
}
```

Example:

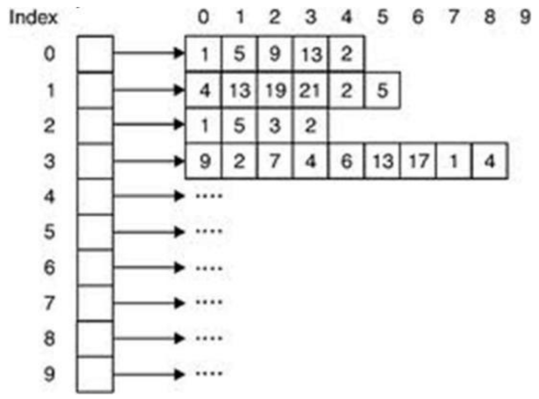
Two-Dimensional Array Using Array of Pointers:
We can dynamically create an array of pointers of size M and then dynamically allocate memory of size N for each row as shown below.



```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main() {
  const int M = 3; // Number of rows
  const int N = 3; // Number of columns

  int** A = new int*[M]; // dynamically create array of pointers of size M
  srand(time(NULL));     // initialize random seed

  for (int i = 0; i < M; i++) {
    A[i] = new int[N]; // dynamically allocate memory of size N for each
row
      for (int j = 0; j < N; j++) {
         A[i][j] = rand() % 100; // assign random values to allocated
memory
      }
  }
  for (int i = 0; i < M; i++) {
      for (int j = 0; j < N; j++) {
          cout << A[i][j] << " "; // print the 2D array
      }
      cout << endl;
  }
  for (int i = 0; i < M; i++) {
      delete[] A[i]; // deallocate memory for each row
  }
  delete[] A; // deallocate memory for the array of pointers
  return 0;
}
```

Jagged Array:

Jagged array is similar to an array but the difference is that it's an array of arrays in which the member arrays can be in different sizes.



Example:

```
    #include <iostream>

    using namespace std;

    int main() {

        int **arr = new int*[3];
        int Size[3];
        int i, j;

        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 << " element " << j + 1 << ": ";
                cin >> arr[i][j];
                //cin>> *(*(arr + i) + j);
            }
        }

        // Print the array elements using loops
        for (i = 0; i < 3; i++) {
            for (j = 0; j < Size[i]; j++) {
                cout << arr[i][j] << " ";
            }
            cout << endl;
        }
        // Deallocate memory
        for (i = 0; i < 3; i++) {
            delete[] arr[i];
        }

        return 0;
    }
```