Data Structure Module 1

Data Structure Classification

Data may be organized in many different ways. The logical or mathematical model of a particular organization of data is called a data structure.

Data structures are generally classified into two main categories:

1. Primitive Data Structures: Primitive data structures are the fundamental data types supported by a programming language. These include basic data types such as integer, float, character, and boolean. They are also known as simple data types as they consist of indivisible values.
2. Non-Primitive Data Structures: Non-primitive data structures are created using primitive data structures. Examples include complex numbers, linked lists, stacks, trees, and graphs.

   Non-Primitive Data Structures are further classified into:

   A. Linear Data Structures: A data structure is linear if its elements form a sequence or a linear list. Linear data structures can be represented in memory in two ways:

   a. Using sequential memory locations (arrays)
   b. Using pointers or links (linked lists)

   Common examples of linear data structures are arrays, queues, stacks, and linked lists.

   B. Non-Linear Data Structures: A data structure is non-linear if the data is not arranged in a sequence. Insertion and deletion are not possible in a linear fashion. These structures represent hierarchical relationships between elements.

   Examples of non-linear data structures are trees and graphs.

Data Structure Operations

1. Traversing: Accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "visiting" the record.)
2. Searching: Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.
3. Inserting: Adding a new node/record to the structure.
4. Deleting: Removing a node/record from the structure.

The following two operations are used in special situations:

1. Sorting: Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as Aadhaar number or bank account number)
2. Merging: Combining the records in two different sorted files into a single sorted file.

Array

An Array is defined as an ordered set of similar data items stored in consecutive memory locations.

- All data items in an array are of the same type.

- Each data item can be accessed using the array name and a unique index value.
- An array is a set of pairs <index, value>, where each index has an associated value.
  <index, value> pairs could be:
  < 0 , 25 > list[0] = 25
  < 1 , 15 > list[1] = 15

Array Declaration in C:

A one-dimensional array in C is declared by adding brackets [] to the variable name.
Example: int list[5], plist[5];

- list[5] defines 5 integers, with indexes from 0 to 4 (list[0], list[1], ..., list[4]).
- plist[5] defines an array of 5 pointers to integers.

Array Implementation:

- When an array list[5] is declared, the compiler allocates 5 consecutive memory locations, each large enough to hold an integer.
- The address of the first element (list[0]) is called the Base Address.
- To compute the address of list[i], the compiler uses:
  address(list[i]) = Base Address + i * sizeof(int)

Difference between `int *list1;` and `int list2[5];` in C is:

`int *list1;`

- `list1` is a pointer variable that can store the address of an integer.
- It does not allocate any memory for storing an integer value.
- It just holds the address of an integer value stored somewhere else in memory.

`int list2[5];`

- `list2` is an array of 5 integers.
- It allocates contiguous memory locations to store 5 integer values.
- The name `list2` can be used to access the base address (address of `list2[0]`) of the array.
- `int *list1;` declares a pointer that can point to an integer value.
- `int list2[5];` declares an array that can store 5 integer values.
- The main difference is that `list1` holds the address of an integer, while `list2` is an array that has memory allocated to store 5 integer values directly.

Array as Parameters to Function

- In C, when an array is passed as a parameter to a function, the array name decays into a pointer to the first element of the array.
- Arrays are passed by reference (as pointers) to functions in C.
- The array size must be passed as a separate argument, or accessed as a global variable.
- Inside the function, the array is accessed using the pointer notation `arr[i]`.
- The function can modify the array elements, but it cannot resize or reallocate the array.

```
#include <stdio.h>
#define MAX_SIZE 5
```

```c
float calculateSum(float arr[], int size) {
    float sum = 0.0;
    for (int i = 0; i < size; i++) {
        sum += arr[i]; // Accessing array elements
    }
    return sum;
}

int main() {
    float numbers[MAX_SIZE] = {1.2, 3.4, 5.6, 7.8, 9.0};
    float result = calculateSum(numbers, MAX_SIZE); // Passing array and size
    printf("The sum of the array elements is: %.2f\n", result);
    return 0;
}
```

- In the `main` function, we declare an array `numbers` of size `MAX_SIZE` (which is defined as 5) and initialize it with some float values.
- Inside the `calculateSum` function:
  - The parameter `arr` is now a pointer to the first element of the `numbers` array passed from `main`.
  - The parameter `size` holds the value `MAX_SIZE` (5), which is the size of the array.
  - The function uses a `for` loop to iterate over the elements of the array, accessing them through the pointer `arr` and the array subscript notation `arr[i]`. Since `arr` holds the address of the first element, `arr[i]` accesses the subsequent elements in the array.

Structure

In C programming language, a structure is a way to group diverse data elements of different data types under a single name. It allows for the organization and storage of related data items as a single unit. The members of a structure can be of different data types such as integers, floats, characters, or even other structures.

struct structure_name
{
data_type member1;
data_type member2;
...
data_type memberN;
};
typedef struct struct_name Type_name;

- `struct` is the keyword used to define a structure.
- `structure_name` is the name given to the structure.
- `member1`, `member2`, ..., `memberN` are the names of the individual data elements or members within the structure.
- `data_type` specifies the data type of each member, such as `int`, `float`, `char`, etc.
- `typedef` is the keyword used to define a new type name or alias for the structure.
- `Type_name` is the user-defined data type name or alias for the structure.

Example:

```c
struct Employee {
    char name[50];
    int age;
    float salary;
```

```
};
```

In the above example, a structure named `Employee` is defined with three members:

- `name`: a character array of size 50 to store the employee's name.
- `age`: an integer to store the employee's age.
- `salary`: a float to store the employee's salary.

```
struct Employee emp1, emp2;

strcpy(emp1.name, "Rahul Sharma");
emp1.age = 35;
emp1.salary = 75000.0;
```