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.

c C

```c
#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

Initialization and Declaration

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.

```c
struct Employee emp1, emp2;

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

A self-referential structure, also known as a recursive structure, is a data structure in which each instance of the structure contains a pointer or reference to another instance of the same type. In other words, the structure contains a member that refers to objects of the same structure type.

Structure and Union difference

Feature	Structures	Unions
Declaration	`struct` keyword followed by a structure tag	`union` keyword followed by a union tag
Memory Usage	Each member has its own memory space	Members share the same memory space
Size Calculation	Size is the sum of sizes of all members	Size is the size of the largest member
Access	Members accessed individually	Only one member can be accessed at a time
Usage	Used when multiple pieces of data are needed	Used when data needs to be overlapped or shared between members
Memory Overlap	Members do not overlap	Members may overlap if they're of different sizes

Write syantax and examples

Pointer

A pointer is a variable that stores the memory address of another variable. Pointers are used to access and manipulate data indirectly through memory addresses.

Declaration: `data_type *pointer_name;`

Example: `int i, *pi; // i is an integer variable, pi is a pointer to an integer`

Operators:

1. & (Address Operator): The unary operator & is used to obtain the memory address of a variable.
   `pi = &i; // pi stores the address of i`
2. - (Indirection/Dereference Operator): The *operator is used to access the value stored at the memory address pointed to by the pointer.*
   `pi = 10; // Assigns 10 to the integer variable i`

Null Pointer:

- A null pointer is a pointer that does not point to any valid memory location.
- In C, the value NULL (defined as 0) represents a null pointer.
- Dereferencing a null pointer can lead to a segmentation fault or undefined behavior.
  `int *ptr = NULL; // ptr is a null pointer`

Dangers of Pointers:

1. Accessing out-of-range memory: Attempting to access memory locations that are not allocated to the program or not within the valid range can lead to undefined behavior or program crashes.
   `int arr[5] = {1, 2, 3, 4, 5}; int *p = arr; printf("%d", *(p + 6)); // Accessing out-of-range memory`
2. Dereferencing a null pointer: Dereferencing a null pointer (attempting to access the value at address 0) can cause a segmentation fault or undefined behavior.
   `int *ptr = NULL; *ptr = 10; // Dereferencing a null pointer (Segmentation Fault)`
3. Type casting between pointer types: Improper type casting between pointer types can lead to unexpected results or memory corruption.
   `int *pi = malloc(sizeof(int)); float *pf = (float*)pi; // Type casting between pointer types`
4. Incorrect return type for functions: If a function's return type is omitted, it defaults to int, which can be interpreted as a pointer type, leading to unexpected behavior.

Pointers are dangerous

Pointer can be very dangerous if they are misused. The pointers are dangerous in the following situations:

1. Pointer can be dangerous when an attempt is made to access an area of memory that is either out of range of the program or that does not contain a pointer reference to a legitimate object.

```c
int main() {
    int *p;
    int pa = 10;
    p = &pa;
    printf("%d", *p); // Output = 10
    printf("%d", *(p+1)); // Accessing memory which is out of range
}
```

2. It is dangerous when a NULL pointer is de-referenced, because on some computers it may return 0 and permitting execution to continue, or it may return the result stored in location zero, so it may produce a serious error.
3. Pointer is dangerous when using explicit type casts in converting between pointer types.

```c
int *pi = malloc(sizeof(int));
float *pf = (float*)pi; // Type casting between pointer types
```

4. In some systems, pointers have the same size as the `int` type. Since `int` is the default type specifier, some programmers omit the return type when defining a function. The return type defaults to `int`, which can later be interpreted as a pointer. This has proven to be a dangerous practice on some computers, and the programmer should define explicit types for functions.

Dynamic Allocation

Dynamic memory allocation refers to the process of manually managing memory allocation and deallocation during the runtime of a program using standard library functions. This allows for more flexibility and efficient memory usage compared to static memory allocation. Here's an explanation of each function used for dynamic memory allocation in C:

1. `malloc()` (Memory Allocation):
   - This function is used to allocate a block of contiguous memory of a specified size.
   - The syntax is: `ptr = (data_type *) malloc(size_in_bytes);`
   - `malloc()` returns a pointer to the first byte of the allocated memory block.
   - If the allocation is successful, the returned pointer can be used to access and manipulate the allocated memory.
   - If the allocation fails (e.g., due to insufficient memory), `malloc()` returns a null pointer.
   - Example: `int *ptr = (int *) malloc(10 * sizeof(int));` allocates memory for an array of 10 integers.
2. `calloc()` (Contiguous Allocation):
   - This function is used to allocate a block of contiguous memory for an array of elements and initializes all bytes to zero.
   - The syntax is: `ptr = (data_type *) calloc(num_elements, element_size);`
   - `calloc()` returns a pointer to the first element of the allocated memory block.
   - If the allocation is successful, the returned pointer can be used to access and manipulate the allocated memory.
   - If the allocation fails (e.g., due to insufficient memory), `calloc()` returns a null pointer.
   - Example: `float *ptr = (float *) calloc(25, sizeof(float));` allocates memory for an array of 25 floats, all initialized to zero.
3. `realloc()` (Re-allocation):
   - This function is used to resize (increase or decrease) the memory block previously allocated with `malloc()` or `calloc()`.
   - The syntax is: `ptr = realloc(ptr, new_size);`
   - `realloc()` returns a pointer to the reallocated memory block, which may be different from the original pointer if the memory needs to be moved.

- If the reallocation is successful, the returned pointer can be used to access and manipulate the reallocated memory.
- If the reallocation fails (e.g., due to insufficient memory), `realloc()` returns a null pointer, and the original memory block remains unchanged.
- Example: `ptr = realloc(ptr, 20 * sizeof(int));` resizes the memory block pointed to by `ptr` to accommodate an array of 20 integers.

4. `free()` (Memory Deallocation):
   - This function is used to deallocate (release) the memory block previously allocated with `malloc()`, `calloc()`, or `realloc()`.
   - The syntax is: `free(ptr);`
   - After calling `free()`, the memory pointed to by `ptr` is released and can be reused by the system.
   - Attempting to access the memory after calling `free()` can lead to undefined behavior or a segmentation fault.
   - It is important to call `free()` on dynamically allocated memory when it is no longer needed to avoid memory leaks.
   - Example: `free(ptr);` releases the memory block pointed to by `ptr`.

Dynamic memory allocation is essential when the size of the required memory is not known at compile-time or when the memory requirements change during program execution. It provides flexibility and efficient memory usage but also introduces the risk of memory leaks and other memory-related errors if not used correctly.

String operations

int strlen(char* str)	length of the char array
char* strcat(char* dest, char* src)	concatenate dest and src strings return result in dest
char* strcat(char* dest, char* src, int n)	concatenate dest and n characters from src strings return result in dest
char* strcpy(char* dest, char* src)	copy src into dest ; return dest
char* strncpy(char* dest, char* src, int n)	copy n characters from src string into dest ; return dest
char* strcmp(char* str1, char* str2)	compares two strings . returns <0 if str1<str2, returns 0 if str1=str2, returns >0 if str1>str2
char* strcmpi(char* str1, char* str2) or char* stricmp(char* str1, char* str2)	compares two strings . returns <0 if str1<str2, returns 0 if str1=str2, returns >0 if str1>str2 **it ignores case.**
char* strcmp(char* str1, char* str2)	compare first n characters of two strings . returns <0 if str1<str2, returns 0 if str1=str2, returns >0 if str1>str2
char* strchr(char*s, int c)	returns pointer to the first occurrence of c in s; return NULL if not present.
char* strrchr(char*s, int c)	returns pointer to the last occurrence of c in s; return NULL if not present.
char* strstr(char* s, char* pat)	Return pointer to start of pat in s
char* strtok(char* s, char delimiters)	Return a token from s, token is surrounded by delimiters
char* strspn(char* s, char* spanset)	Scan s for characters in spanset , return length of span
char* strcspn(char* s, char* spanset)	Scan s for characters not in spanset , return length of span
strlwr()	converts from upper case to lower case
struper()	converts from upper case to lower case
strrev()	reverses a string
strset(char* s, char c)	It sets all characters in string to character c
strnset(char* s, char c)	It sets first n characters in string to character c

---

Polynomial

A polynomial is a mathematical expression consisting of variables (also known as indeterminates or unknowns) and coefficients, combined using addition, subtraction, multiplication, and non-negative integer exponents. It typically takes the form:

The degree of a polynomial is the highest power of the variable in the polynomial expression.

Sparse Matrix

A sparse matrix is a type of matrix that contains mostly zero elements. In contrast to dense matrices, which have a majority of non-zero elements, sparse matrices have a high proportion of zero entries relative to the total number of elements.