5.4. WHAT IS A SOCIAL NETWORK

A social network is a heterogeneous and multi-relational information set described by a graph. The graph is generally very large, with nodes corresponding to objects and edges corresponding to connections describing relationships or connections between objects. Both nodes and connections have attributes. Objects can have class labels. Links can be one-directional and are not needed to be binary.

A social network is a heterogeneous and multi-relational information set described by a graph. The graph is generally very large, with nodes corresponding to objects and edges corresponding to connections describing relationships or connections between objects. Both nodes and connections have attributes. Objects can have class labels. Links can be one-directional and are not needed to be binary

Characteristics of Social Networks

There are the following characteristics of social network which are as follows −

- **Densification power-law** − It was considered that as a network evolves, the number of degrees increases linearly in the multiple nodes. This was called the constant average degree hypothesis. But, extensive experiments have displayed that, viceversa, networks become denser over time with the average degree increasing (and therefore, the number of edges increasing super linearly in the number of nodes).The densification follows the densification power law (or growth power-law), which defines
    - **Shrinking diameter** − It has been experimentally shown that the efficient diameter tends to reduce as the network increases. This contradicts an earlier understanding that the diameter slowly increases as a service of network size.
    - Consider a citation web, where nodes are papers and a citation from one paper to another is denoted by a directed edge. Outlinks of a node, v (defining the papers cited by v), are “frozen” at the moment it combines the graph. The decreasing distances among pairs of nodes consequently occur to be the result of subseque
    - **Heavy-tailed out-degree and in-degree distributions** − The multiple out-degrees for a node tend to follow a heavy-tailed distribution by observing the power law, $1/n^a$, where n is the rank of the node in the order of decreasing out-degrees and generally, 0

The smaller the value of a the heavier the tail. This phenomenon is defined in the preferential connection model, where each new node connects to an existing network by a fixed number of out-links, following a rich-get-richer rule. The

indegrees also follow a heavy-tailed distribution, although its influence is more skewed than the out-degrees distribution.

The following table describes some of the famous social networking services provided over web and mobile:

S.N.	Service Description
1.	**Facebook** Allows to share text, photos, video etc. It also offers interesting online games.
2.	**Google+** It is pronounced as Google Plus. It is owned and operated by Google.
3.	**Twitter** Twitter allows the user to send and reply messages in form of tweets. These tweets are the small messages, generally include 140+ characters.
4.	**Faceparty** Faceparty is a UK based social networking site. It allows the users to create profiles and interact with each other using forums messages.
5.	**Linkedin** Linkedin is a business and professional networking site.
6.	**Flickr** Flickr offers image hosting and video hosting.
7.	**Ibibo** Ibibo is a talent based social networking site. It allows the users to promote ones self and also discover new talent.
8.	**Whatsapp** It is a mobile based messaging app. It allows to send text, video, and audio messages
9.	**Line** It is same as whatsapp. Allows to make free calls and messages.
10.	**Hike** It is also mobile based messager allows to send messages and exciting emoticons.

Where Social Networking Helps

Following are the areas where social networking has become most popular:

Online Marketing

Website like facebook allows us to create a page for specific product, community or firm and promiting over the web.

Online Jobs

Website like linkedin allows us to create connection with professionals and helps to find the suitable job based on ones specific skills set.

Online News

On social networking sites, people also post daily news which helps us to keep us updated.

Chatting

Social networking allows us to keep in contact with friends and family. We can communicate with them via messages.

Share Picture, Audio and video

One can share picture, audio and video using social networking sites

MEMORY MANAGEMENT IN C

One of the important characteristics of C is that the compiler manages how the memory is allocated to the variables declared in the code. Once the compiler allocates the required bytes of memory, it cannot be changed during the runtime.

The compiler employs static memory allocation approach. However, there are times where you may need to allocate the memory on demand, during the runtime. Read this chapter to understand how dynamic memory management works in C.

Functions for Dynamic Memory Management in C

The C programming language provides several functions for dynamic memory allocation and management. These functions can be found in the **<stdlib.h> header file**.

Function	Description
void ***calloc**(int num, int size);	This function allocates an array of num elements each of which size in bytes will be size.
void **free**(void *address);	This function releases a block of memory block specified by address.
void ***malloc**(size_t size);	This function allocates an array of num bytes and leave them uninitialized.
void ***realloc**(void *address, int newsize);	This function re-allocates memory extending it up to newsize.

Memory in C

Understanding how memory works in C is important. When you create a basic variable, C will automatically reserve space for that variable. An int variable for example, will typically occupy 4 bytes of memory, while a double variable will occupy 8 bytes of memory.

You can use the sizeof operator to find the size of different types:

```
int myInt;
float myFloat;
double myDouble;
char myChar;

printf("%zu\n", sizeof(myInt));      // 4 bytes
printf("%zu\n", sizeof(myFloat));    // 4 bytes
printf("%zu\n", sizeof(myDouble));   // 8 bytes
printf("%zu\n", sizeof(myChar));     // 1 byte
```

C Allocate Memory

The process of reserving memory is called allocation. The way to allocate memory depends on the type of memory.

C has two types of memory: Static memory and dynamic memory.

Static Memory

Static memory is memory that is reserved for variables **before** the program runs. Allocation of static memory is also known as *compile time* memory allocation.

C automatically allocates memory for every variable when the program is compiled.

For example, if you create an integer array of 20 students (e.g. for a summer semester), C will reserve space for 20 elements which is typically 80 bytes of memory (20 * 4):

```
int students[20];
printf("%zu", sizeof(students)); // 80 bytes
```

Dynamic Memory

Dynamic memory is memory that is allocated **after** the program starts running. Allocation of dynamic memory can also be referred to as *runtime* memory allocation.

Unlike with static memory, you have full control over how much memory is being used at any time. You can write code to determine how much memory you need and allocate it.

Dynamic memory does not belong to a variable, it can only be accessed with pointers.

To allocate dynamic memory, you can use the malloc() or calloc() functions. It is necessary to include the <stdlib.h> header to use them.
The malloc() and calloc() functions allocate some memory and return a pointer to its address.

```
int *ptr1 = malloc(size);
int *ptr2 = calloc(amount, size);
```

The malloc() function has one parameter, *size*, which specifies how much memory to allocate, measured in bytes.

The calloc() function has two parameters:

- *amount* - Specifies the amount of items to allocate

- *size* - Specifies the size of each item measured in bytes

Stack Memory

For completeness, it is worth mentioning stack memory. Stack memory is a type of dynamic memory which is reserved for variables that are declared inside functions. Variables declared inside a function use stack memory rather than static memory.

When a function is called, stack memory is allocated for the variables in the function. When the function returns the stack memory is freed.

It is good to be aware of stack memory to be able to handle the memory usage of nested function calls and recursion. Recursion that repeats itself too many times may take up too much stack memory. When that happens it is called a **stack overflow**.

Access Dynamic Memory

Dynamic memory behaves like an array, with its data type specified by the type of the pointer.

As with arrays, to access an element in dynamic memory, refer to its **index number**:

```
ptr[0] = 12;
```

You can also dereference the pointer to access the first element:

```
*ptr = 12;
```

Example

```
// Allocate memory
int *ptr;
ptr = calloc(4, sizeof(*ptr));

// Write to the memory
*ptr = 2;
ptr[1] = 4;
ptr[2] = 6;

// Read from the memory
```

```
printf("%d\n", *ptr);
printf("%d %d %d", ptr[1], ptr[2], ptr[3]);
```

1. Memory Layout of a C Program

When a C program runs, memory is typically divided into several regions:

```
+------------------+
| Command Line Args|
+------------------+
| Stack            |
| (local variables)|
+------------------+
| Free Memory      |
+------------------+
| Heap             |
| (dynamic memory) |
+------------------+
| Data Segment     |
| (global/static)  |
+------------------+
| Text Segment     |
| (program code)   |
+------------------+
```

2. Text Segment (Code Segment)

Contains the executable instructions of the program.

Example:

```
#include <stdio.h>

int main() {
    printf("Hello");
    return 0;
}
```

The machine code generated for `main()` is stored in the text segment.

Characteristics

- Read-only
- Shared among processes when possible
- Contains executable instructions

3. Data Segment

Stores global and static variables.

Initialized Data Segment

```
int x = 10;
static int y = 20;
```

Stored in initialized data area.

Uninitialized Data Segment (BSS)

```
int a;
static int b;
```

Stored in BSS (Block Started by Symbol).

Characteristics:

- Exists throughout program execution
- Created before `main()`
- Destroyed after program termination

4. Stack Memory

Used for:

- Function calls
- Local variables
- Parameters
- Return addresses

Example:

```
void fun() {
    int x = 10;
}
```

`x` is stored on the stack.

Stack Growth

```
main()
 |
 +--> fun()
        |
        +--> local variables
```

When function returns:

```
Stack frame removed automatically
```

Advantages

- Fast allocation
- Automatic deallocation
- No memory leaks

Disadvantages

- Limited size
- Stack overflow possible

Example:

```
void recurse() {
    recurse();
}
```

May cause:

```
Stack Overflow
```

5. Heap Memory

Used for dynamic memory allocation.

Managed by programmer.

Example:

```
int *ptr = malloc(sizeof(int));
```

Memory allocated from heap.

Characteristics

- Large memory area
- Allocation at runtime
- Manual deallocation required

6. Dynamic Memory Allocation Functions

C provides four functions from:

```
#include <stdlib.h>
```

malloc()

Allocates memory block.

Syntax:

```
ptr = malloc(size);
```

Example:

```
int *ptr;

ptr = (int *)malloc(sizeof(int));
```

Memory:

```
Heap:
+------+
| ???? |
+------+
```

Contents are garbage values.

calloc()

Allocates and initializes memory to zero.

Syntax:

```
ptr = calloc(n, size);
```

Example:

```
int *ptr;

ptr = (int *)calloc(5, sizeof(int));
```

Memory:

```
0 0 0 0 0
```

Difference

malloc:

```
Garbage values
```

calloc:

```
Zero initialized
```

realloc()

Changes size of allocated block.

Syntax:

```
ptr = realloc(ptr, new_size);
```

Example:

```
ptr = realloc(ptr, 10 * sizeof(int));
```

Before:

```
5 integers
```

After:

```
10 integers
```

Use when memory requirements change dynamically.

free()

Releases heap memory.

Syntax:

```
free(ptr);
```

Example:

```
free(ptr);
ptr = NULL;
```

Always set pointer to NULL after freeing when practical.

7. Memory Allocation Example

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *arr;

    arr = malloc(5 * sizeof(int));

    for(int i=0;i<5;i++)
        arr[i] = i + 1;

    for(int i=0;i<5;i++)
        printf("%d ", arr[i]);

    free(arr);

    return 0;
}
```

Output:

```
1 2 3 4 5
```

8. Memory Leak

Occurs when allocated memory is never released.

Example:

```
void fun()
{
    int *ptr = malloc(100);

    return;
}
```

Problem:

```
Memory allocated
Memory never freed
```

Result:

```
Memory Leak
```

Correct:

```
void fun()
{
    int *ptr = malloc(100);

    free(ptr);
}
```

9. Dangling Pointer

Pointer referencing freed memory.

Example:

```
int *ptr = malloc(sizeof(int));

free(ptr);
```

Now:

```
ptr → invalid memory
```

Danger:

```
*ptr = 100;
```

Undefined behavior.

Solution:

```
free(ptr);
ptr = NULL;
```

10. Wild Pointer

Uninitialized pointer.

Example:

```
int *ptr;
```

Contains random address.

Danger:

```
*ptr = 10;
```

May crash.

Correct:

```
int *ptr = NULL;
```

11. NULL Pointer

Points to no valid memory location.

Example:

```
int *ptr = NULL;
```

Useful for:

- Initialization
- Error checking
- Preventing accidental access

Check:

```
if(ptr != NULL)
{
    *ptr = 10;
}
```

12. Double Free Error

Example:

```
int *ptr = malloc(sizeof(int));

free(ptr);
free(ptr);
```

Problem:

```
Same memory released twice
```

Can corrupt heap.

Solution:

```
free(ptr);
ptr = NULL;
```

13. Buffer Overflow

Writing beyond allocated memory.

Example:

```
int *arr = malloc(5 * sizeof(int));

arr[10] = 100;
```

Problem:

```
Access outside allocated block
```

Consequences:

- Crash
- Data corruption
- Security vulnerabilities

14. Fragmentation

Internal Fragmentation

Allocated memory larger than required.

Example:

```
Need 18 bytes
Allocated 32 bytes
```

Unused memory wasted.

External Fragmentation

Free memory exists but scattered.

Example:

```
[Free][Used][Free][Used][Free]
```

Large contiguous block unavailable.

15. Memory Management Techniques

First Fit

Allocate first suitable block.

```
Free Blocks:
100 500 200

Request:
150

Allocate from 500
```

Fast allocation.

Best Fit

Allocate smallest suitable block.

```
Free:
100 500 200

Request:
150

Allocate from 200
```

Less wastage but slower search.

Worst Fit

Allocate largest available block.

```
Free:
100 500 200

Request:
150

Allocate from 500
```

Leaves larger free spaces.

16. Garbage Collection vs C

Languages like:

- Java
- C#
- Python

Use garbage collectors.

C:

```
No automatic garbage collection
```

Programmer responsibilities:

1. Allocate memory.
2. Use memory.
3. Free memory.

17. Static vs Dynamic Memory

Feature	Static	Dynamic
Allocation Time	Compile Time	Runtime
Memory Area	Stack/Data Segment	Heap
Size	Fixed	Flexible
Speed	Fast	Slower
Control	Automatic	Manual

Example Static:

```
int arr[100];
```

Example Dynamic: