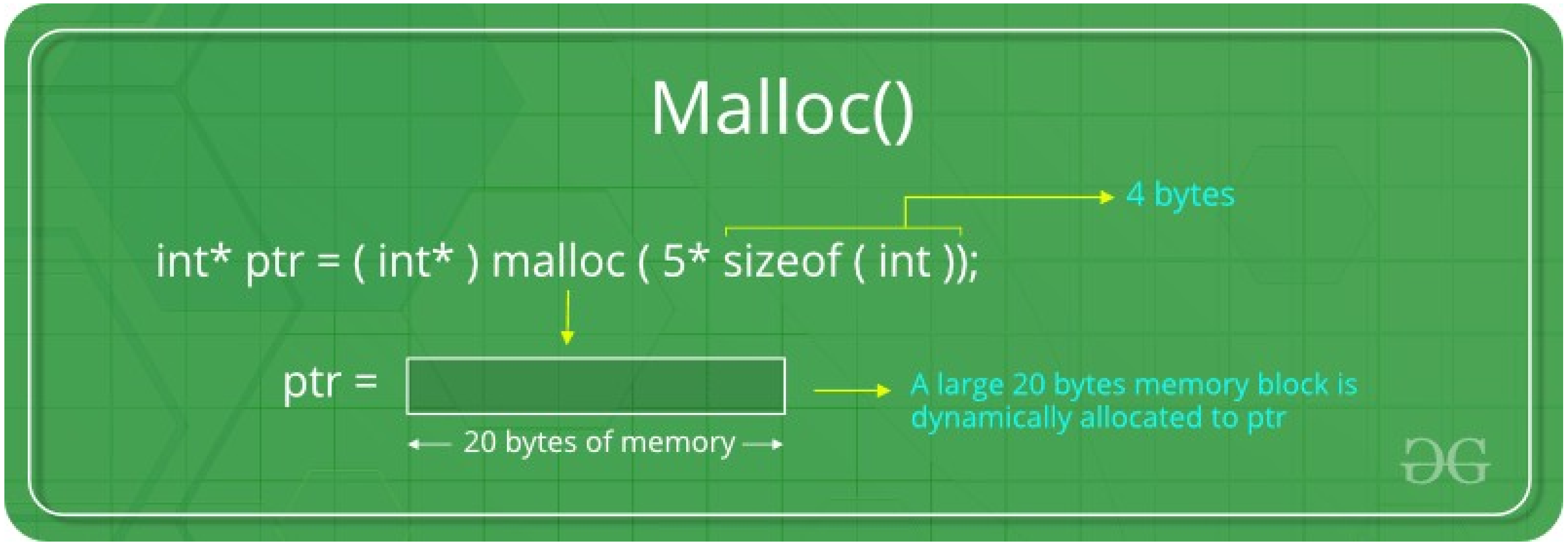
C PROGRAMMING
Lecture 8

1st semester 2023-2024

Dynamic Memory Allocation


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
20 bytes of memory
A large 20 bytes memory block is dynamically allocated to ptr

https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

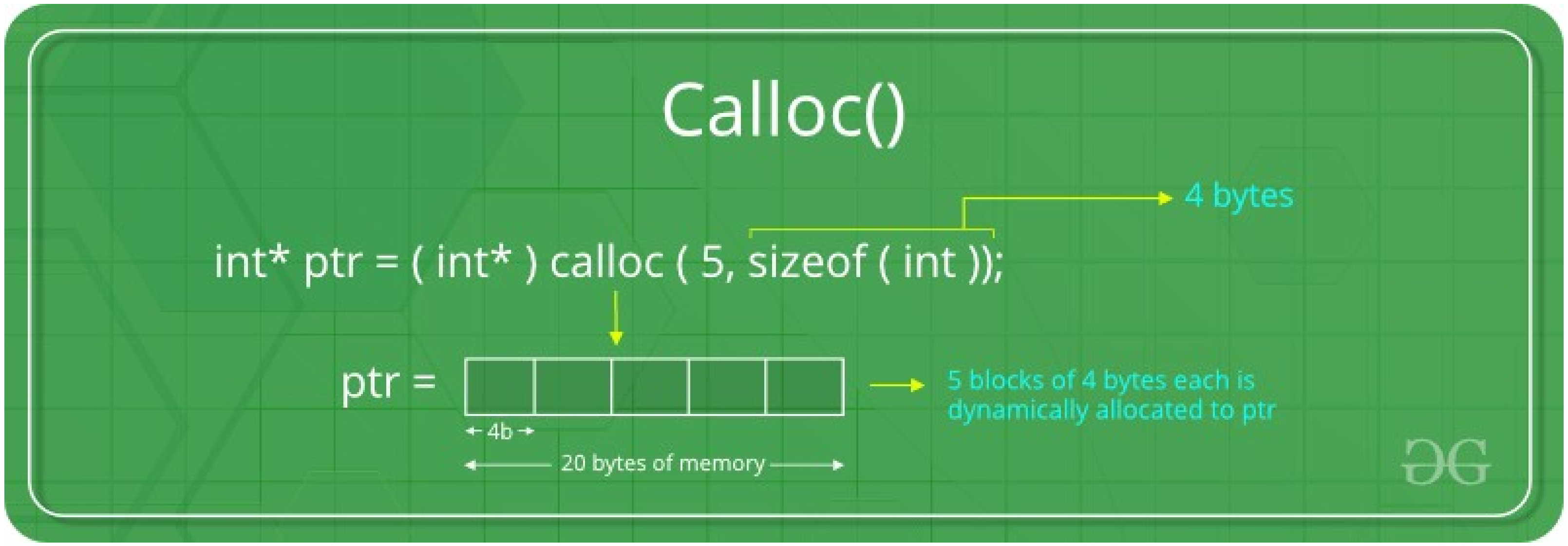
Dynamic Memory Allocation

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

int main()
{

	// This pointer will hold the
	// base address of the block created
	int* ptr;
	int n, i;

	// Get the number of elements for the array
	printf("Enter number of elements:");
	scanf("%d",&n);
	printf("Entered number of elements: %d\n", n);

	// Dynamically allocate memory using malloc()
	ptr = (int*)malloc(n * sizeof(int));

	// Check if the memory has been successfully
	// allocated by malloc or not
	if (ptr == NULL) {
		printf("Memory not allocated.\n");
		exit(0);
	}
```

Dynamic Memory Allocation

```
else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
```

Dynamic Memory Allocation

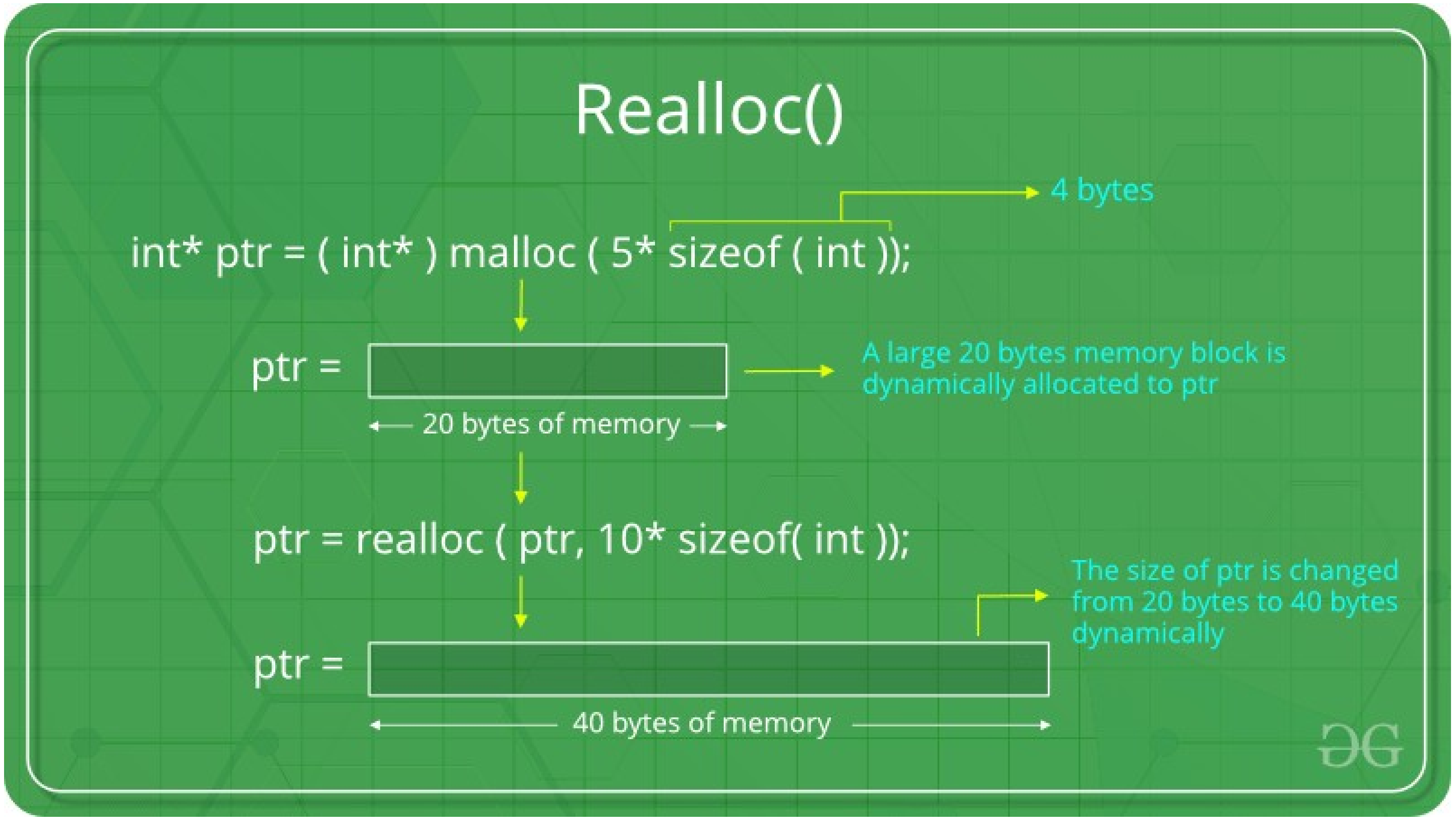

Calloc()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
4b
20 bytes of memory
5 blocks of 4 bytes each is dynamically allocated to ptr

https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Dynamic Memory Allocation

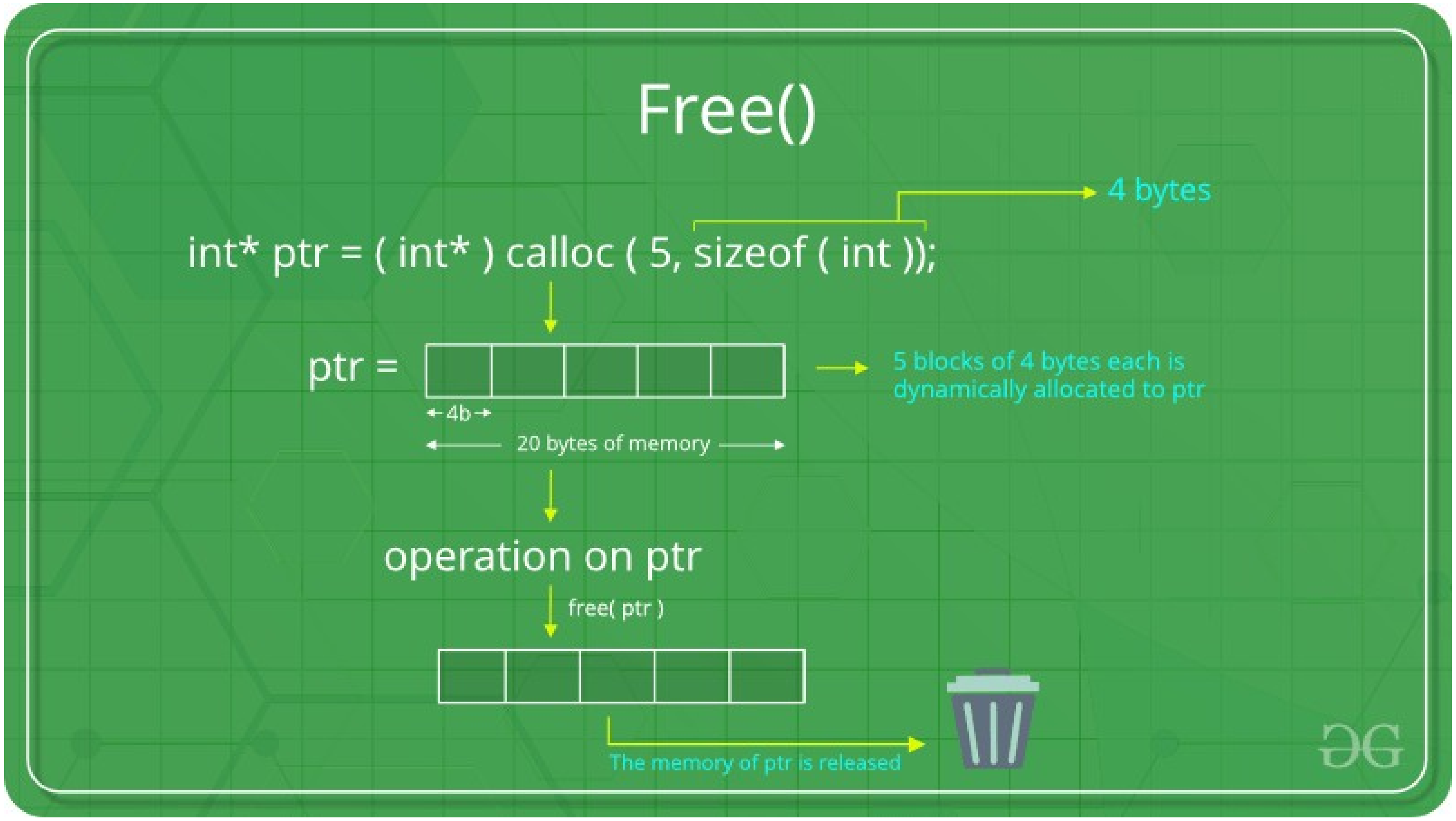
```
int* ptr;
        int n, i;

        // Get the number of elements for the array
        n = 5;
        printf("Enter number of elements: %d\n", n);

        // Dynamically allocate memory using calloc()
        ptr = (int*)calloc(n, sizeof(int));
```

Dynamic Memory Allocation


Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Dynamic Memory Allocation


Free()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory
operation on ptr
free(ptr)
The memory of ptr is released

https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Variable Length Arrays(VLAs)

VLAs allow you to declare arrays whose size is not known until runtime.

VLAs provide more flexibility in handling situations where the array size is determined during the execution of the program.

```
int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int arr[n];
```

Variable Length Arrays(VLAs)

```
#include <stdio.h>

int main()
{

    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter elements: ");

    for (int i = 0; i < n; ++i) {

        scanf("%d", &arr[i]);
    }

      printf("Elements of VLA of Given Size: ");
    for (int i = 0; i < n; ++i) {

        printf("%d ", arr[i]);
    }

    return 0;
}
```

Variable Length Arrays(VLAs)

Automatic Storage Duration: VLAs in C have automatic storage duration, which means they are typically allocated on the stack. This can lead to stack overflow if the array size is too large.

No Size Checking: There is no automatic bounds checking for VLAs. It's the programmer's responsibility to ensure that the array is not accessed out of bounds.

Limited Portability: Not all compilers support VLAs, and they are not part of the C standard before C99. Therefore, code using VLAs may not be portable across different compilers.

Dynamic Memory Allocation: For larger arrays or when the size is not known until runtime, dynamic memory allocation using functions like malloc (and its counterparts) is often a more flexible and safer alternative.

Variable Length Arrays(VLAs)

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

int main() {
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int *array = (int *)malloc(n * sizeof(int));

    // Access and manipulate the array as needed

    // Don't forget to free the allocated memory
    free(array);

    return 0;
}
```

Flexible Array Members (FAM)

FAM is a feature in C that allows you to declare an array as the last member of a structure without specifying its size. This array can then be used to represent variable-length data within the structure. The size of the array is not included in the structure's size, and it is determined dynamically during runtime.

Keep in mind that FAM is specific to the last member of a structure, and you should be careful about accessing the elements beyond the declared size to avoid undefined behavior.

Flexible Array Members (FAM)

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

// Define a structure with a flexible array member
//Flexible Array Members (FAM)
struct MyStruct {
    int length;        // Some other data members
    int data[];        // Flexible array member
};

int main() {
    int n;

    // Get the size of the array from the user
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    // Allocate memory for the structure and the flexible array member
    struct MyStruct *myStruct = malloc(sizeof(struct MyStruct) + n * sizeof(int));
```

Flexible Array Members (FAM)

```
// Check if memory allocation was successful
    if (myStruct == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;  // Exit with an error code
    }

    // Initialize the structure
    myStruct->length = n;

    // Access and manipulate the flexible array member
    for (int i = 0; i < n; ++i) {
        myStruct->data[i] = i * 10;
    }

    // Print the elements of the flexible array member
    printf("Array elements: ");
    for (int i = 0; i < n; ++i) {
        printf("%d ", myStruct->data[i]);
    }

    // Don't forget to free the allocated memory
    free(myStruct);

    return 0;
}
```

Linked Lists

- Linked list: data structure composed by nodes representing a sequence. Each node is composed of useful data and a link to the next node in the sequence. The structure allows operations for insertion or removal of elements from the sequence.
- Compared to arrays, lists are easier to operate when inserting or removing elements as these operations don't need reallocation or reorganization of the list.
- Advantages:
  - Basis for abstract types such as stacks, queues, associative arrays
  - Dynamic structure; memory allocated while the program is running
  - Easy implementation for insert and delete operations
- Disadvantages:
  - Can waste memory because of using pointers that require extra storage space
  - Compared to arrays, nodes are stored at non contiguous locations, increasing access time to specific elements
  - Nodes must be read in order from the beginning as linked lists
  - Difficult to navigate backwards

Linked Lists

A linked list is a data structure that consists of a sequence of elements, where each element points to the next one in the sequence. Unlike arrays, linked lists do not have a fixed size in memory, and their elements (nodes) are not stored in contiguous locations.

Components of a Linked List:

Node:

A node is the basic building block of a linked list.
It contains data and a pointer to the next node in the sequence.
In C, a node structure might look like this:

```
struct Node {
        int data;
        struct Node* next;
              };
```

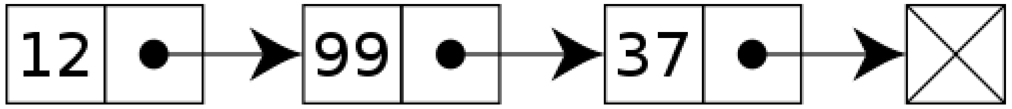
Linked Lists

Head:

The head is a pointer that points to the first node in the linked list.
It is the entry point for accessing the linked list.

Singly Linked List:

Each node points to the next node in the sequence.
The last node typically has a NULL (or nullptr) next pointer.



Linked Lists

Basic Operations:

Insertion:

Adding a new node to the linked list.
Three cases: at the beginning, in the middle, or at the end.

Deletion:

Removing a node from the linked list.
Three cases: at the beginning, in the middle, or at the end.

Traversal:

Visiting each node in the linked list to perform an operation

Linked Lists

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

// Define the structure for a node
struct Node {
    int data;
    struct Node* next;
};
```

Linked Lists

```
/ Function to insert a new node at the beginning of the linked list
struct Node* insertAtBeginning(struct Node* head, int newData) {


    if (newNode == NULL) {
        perror("Error allocating memory");
        exit(EXIT_FAILURE);
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    newNode->data = newData;
    newNode->next = head;
    return newNode;
}
```

Linked Lists

```
// Function to print the elements of the linked list
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
```

Linked Lists

```
// Function to free the memory allocated for the linked list
void freeList(struct Node* head) {
    struct Node* current = head;
    struct Node* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
}
```

Linked Lists

```
int main() {
    struct Node* head = NULL;

    // Insert nodes at the beginning
    head = insertAtBeginning(head, 3);
    head = insertAtBeginning(head, 2);
    head = insertAtBeginning(head, 1);

    // Print the linked list
    printf("Linked List: ");
    printList(head);

    // Free allocated memory
    freeList(head);

    return 0;
```

Delete Linked List

- We need to know the head, node to delete and previous node
- Several cases:
  - Head is NULL
  - Node to delete is the first node
  - Arbitrary node from the list (different from first)

Delete Linked List

```
nodeptr delete(nodeptr head, int value)
{
        if (head == NULL)
                return NULL;
        if (head->num == value) {
                nodeptr tempNextP;
                tempNextP = head->next;
                free(head);
                return tempNextP;
        }
        head->next = delete(head->next, value);
        return head;
}

 scanf("%d",&val);
 printf("\n");
 head=delete(head,val);
```

Other Dynamic Structures

- Trees
- Hash Tables
- Directed Acyclic Graphs
- …