PES UNIVERSITY

CELEBRATING 50 YEARS

Data Structures

Dilip Kumar Maripuri

Computer Applications

Data Structures

Session : Linked Lists: Linked List as an ADT, Types of Linked Lists, Construction of a Node and introduction to operations

Dilip Kumar Maripuri

Computer Applications

Data Structures

Overview of Pointers

- Pointers in C are powerful constructs that allow the manipulation of memory directly.
- They store the memory address of another variable, allowing for more efficient and flexible handling of data.
  - **Declaration:** A pointer is declared by placing an asterisk (*) before the pointer's name, indicating that it is a pointer variable.
  - **Initialization:** A pointer must be assigned the address of a variable using the address-of operator (&).
  - **Dereferencing:** Dereferencing a pointer is done using the asterisk (*) operator, which provides access to the value stored at the address contained in the pointer.

Data Structures

Overview of Pointers

PES UNIVERSITY

- **Types of Pointers**
  - **Null Pointer**

    A pointer that is assigned a `NULL` value points to nothing.

    It is often used to indicate that a pointer is not initialized yet or no memory has been assigned.

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

  - **Void Pointer**

    A generic pointer type that can point to any data type.

    However, it must be cast to another pointer type before dereferencing.

    ```
    void *ptr;
    int a = 5;
    ptr = a;
    ```

Data Structures

Overview of Pointers

PES UNIVERSITY

- **Types of Pointers**
  - **Wild Pointer**

    A pointer that has not been initialized properly, pointing to some random memory location.

    Dereferencing it leads to undefined behavior.

    ```
    int *ptr;
    ```
  - **Dangling Pointer**

    A pointer that still points to a memory location that has been freed or deleted.

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

Data Structures

Overview of Pointers

PES UNIVERSITY

- **Types of Pointers**
  - **Constant Pointer**

    A pointer whose value cannot be changed after initialization.

    It always points to the same address.

    ```
    int a = 10;

    int *const ptr = a;
    ```
  - **Pointer to Constant**

    A pointer that points to a constant value, so the value it points to cannot be modified.

    ```
    const int a = 5;

    const int *ptr = &a;
    ```

Overview of Pointers

- The size of a pointer in C depends on the architecture and the compiler implementation, rather than the data type it points to.
  - In a 32-bit architecture, pointers are typically 4 bytes (32 bits) in size.
  - In a 64-bit architecture, pointers are usually 8 bytes (64 bits).
  - Different compilers and operating systems may have different implementations that affect the size of a pointer.

Data Structures

Overview of Pointers

PES UNIVERSITY

```c
#include <stdio.h>

int main() {
    int *intPointer;
    char *charPointer;
    void *voidPointer;

    printf("Size of int pointer: %zu bytes\n", sizeof(intPointer));
    printf("Size of char pointer: %zu bytes\n", sizeof(charPointer));
    printf("Size of void pointer: %zu bytes\n", sizeof(voidPointer));

    return 0;
}
```

```
maripuri@maripuri:~$ ./a.out
Size of int pointer: 8 bytes
Size of char pointer: 8 bytes
Size of void pointer: 8 bytes
maripuri@maripuri:~$
```

Data Structures

Overview of Pointers

PES UNIVERSITY

- **Dynamic Memory Allocation**
  - Dynamic memory allocation in C allows programmers to allocate memory during the runtime of the program.
  - This is crucial for creating flexible and efficient applications that can adapt to varying data sizes and requirements.
  - C provides four essential functions in the **stdlib.h** library for dynamic memory management: `malloc()` / `calloc()` / `realloc` / `free()`

Data Structures

Overview of Pointers

PES UNIVERSITY

- **Key Functions from `stdlib.h` for Memory Management**
  - `malloc()`:
    - **Purpose** : Allocates a specified number of bytes of memory.
    - **Syntax** : `void* malloc(size_t size)`
    - **Context of Usage** : Use when the size of the memory block is known and initialization to zero is not required.

Data Structures

Overview of Pointers

PES UNIVERSITY

- **Key Functions from `stdlib.h` for Memory Management**
  - `calloc()`:
    - **Purpose** : Allocates memory for an array of elements of a specified size and initializes all bytes to zero.
    - **Syntax** : `void* calloc(size_t nitems, size_t size)`
    - **Context of Usage** : Ideal for allocating memory for arrays where initialized values are needed.

Data Structures

Overview of Pointers

PES UNIVERSITY

- **Key Functions from `stdlib.h` for Memory Management**
  - `realloc()`:
    - **Purpose** : Resizes a previously allocated memory block.
    - **Syntax** : `void* void* realloc(void* ptr, size_t size)`
    - **Context of Usage** : Use when modifying the size of an existing memory block, such as expanding or reducing it.

Data Structures

Overview of Pointers

PES UNIVERSITY

- **Key Functions from `stdlib.h` for Memory Management**
  - `free()`:
    - **Purpose** : RFrees the allocated memory, releasing it back to the system.
    - **Syntax** : `void free(void* ptr)`
    - **Context of Usage** : Essential for preventing memory leaks by releasing memory when it is no longer needed.
    - To avoid issues with dangling pointers, it is a good practice to manually set ptr to NULL after calling free.
    - Setting ptr to NULL after freeing the memory makes ptr a null pointer, which is safer because dereferencing a null pointer is more predictable

Data Structures

Overview of Pointers

PES UNIVERSITY

- **Important Considerations :**
  - Remember to always deallocate memory with free() to avoid memory leaks.
  - Be cautious with pointer usage after freeing memory to avoid dangling pointers.
  - It is a good practice to set pointers to NULL after freeing.

Data Structures

Overview of Pointers

PES UNIVERSITY

```
int *arr = (int *)malloc(5 * sizeof(int));
if (arr == NULL)    {
    printf("Memory allocation failed using malloc.\n");
    return 1;       }

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

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

Data Structures

Overview of Pointers

PES UNIVERSITY

```c
// Using calloc to allocate memory for an array of 5 integers
int *arr2 = (int *)calloc(5, sizeof(int));
if (arr2 == NULL) {
    printf("Memory allocation failed using calloc.\n");
    free(arr); // Free previously allocated memory
    return 1;
}

// Printing the calloc array (should be initialized to zero)
printf("Array allocated using calloc: ");
for (int i = 0; i < 5; i++)
    printf("%d ", arr2[i]);
```

Data Structures

Overview of Pointers

PES UNIVERSITY

```
// Using realloc to resize the malloc array to hold 10 integers
arr = (int *)realloc(arr, 10 * sizeof(int));
if (arr == NULL) {
    printf("Memory reallocation failed using realloc.\n");
    free(arr2);
    return 1;
}

for (int i = 5; i < 10; arr[i++]=i+1);

for (int i = 0; i < 10; i++)
    printf("%d ", arr[i]);
```

```
// Freeing the allocated memory
free(arr);
free(arr2);

printf("Memory successfully freed.\n");
```

Data Structures

Overview of Pointers

PES UNIVERSITY

Aspect	`malloc`	`calloc`	`realloc`
Purpose	Allocates memory of a specified size.	Allocates memory for an array and sets it to zero.	Resizes a previously allocated memory block.
How it Works	Takes a size in bytes and returns a pointer.	Takes element count and size, returns pointer to zero-initialized memory.	Takes a pointer and a new size, reallocates memory, and returns a pointer.

Data Structures

Overview of Pointers

PES UNIVERSITY

Aspect	`malloc`	`calloc`	`realloc`
Initialization	Memory is uninitialized (garbage values).	Memory is initialized to zero.	New memory is uninitialized, existing memory is preserved.
Pitfalls	Forgetting to initialize can cause bugs.	Initialization can slow down large allocations.	If reallocation fails, original memory may be lost.

Data Structures

Overview of Pointers

PES UNIVERSITY

Aspect	Static Memory Allocation	Dynamic Memory Allocation
Definition	Memory allocation at compile time.	Memory allocation at runtime.
When Used	Used when the amount of memory needed is known and constant.	Used when memory needs vary or are not known in advance.
Memory Location	Often allocated in the stack segment of memory.	Allocated in the heap segment of memory.

Data Structures

Overview of Pointers

PES UNIVERSITY

Aspect	Static Memory Allocation	Dynamic Memory Allocation
Lifetime	The allocated memory lasts for the lifetime of the program or function.	The allocated memory lasts until it is explicitly freed.
Size Flexibility	Fixed size; cannot be altered during runtime.	Flexible; can be resized using functions like `realloc()`.
Memory Manage-ment	Managed by the compiler.	Managed by the programmer, requiring explicit allocation and deallocation.

Data Structures

Overview of Pointers

PES UNIVERSITY

Aspect	Static Memory Allocation	Dynamic Memory Allocation
Example Functions/ Keywords	Variables declared with static storage duration, arrays of fixed size.	malloc(), calloc(), realloc(), free() in C.
Efficiency	Generally faster as memory is allocated at compile time.	Slower compared to static allocation due to runtime management.
Risk of Errors	Lower risk of errors like memory leaks.	Higher risk of errors like memory leaks and dangling pointers.

Data Structures

Overview of Pointers

PES UNIVERSITY

Aspect	Static Memory Allocation	Dynamic Memory Allocation
Usage in Programs	Suitable for programs with known, fixed memory requirements.	Ideal for programs requiring flexibility in memory allocation.
Control Over Memory	Less control; the system handles allocation and deallocation.	More control; the programmer explicitly manages memory.

Data Structures
Linked Lists

PES UNIVERSITY

- A linked list is a fundamental data structure in computer science, used to store collections of elements.
- Unlike arrays, elements in a linked list (called ”nodes”) are not stored in contiguous memory locations.
- A linked list is an ordered collection of data in which each element contains the location of the next element or elements.
- In a linked list, each element contains two parts: data and one or more links.
  - The data part holds the application data — the data to be processed.
  - Links are used to chain the data together. They contain pointers that identify the next element or elements in the list.

Data Structures

Why Linked Lists?

PES UNIVERSITY

- **Dynamic Size :** The number of elements is not known in advance or changes frequently.
- **Efficient Insertions/Deletions :** They allow for efficient insertions and deletions of elements at any position, without the need to reorganize the entire data structure (unlike arrays).
- **Memory Efficiency :** They can be more memory-efficient, as they do not pre-allocate space (unlike static arrays) and only allocate memory as new elements are added.

Data Structures
Comparisons !!!

PES UNIVERSITY

Criteria	Arrays	Dynamic Arrays	Linked Lists
Memory Allocation	Static (fixed size)	Dynamic (resizable)	Dynamic (per element)
Memory Utilization	Can be inefficient	More efficient than arrays	Optimal (no pre-allocation)
Element Access	Constant time (O(1))	Constant time (O(1))	Linear time (O(n))
Insertions / Deletions	Slow (O(n))	Average (O(n) on resizing)	Fast (O(1) at specific positions)
Memory Layout	Contiguous	Contiguous	Non-contiguous (pointer-based)

Data Structures
Comparisons !!!

PES UNIVERSITY

Criteria	Arrays	Dynamic Arrays	Linked Lists
Size Flexibility	Fixed	Resizable	Extremely flexible
Overhead	Minimal	Moderate (due to resizing)	Higher (additional memory for pointers)

Data Structures
Linked Lists

PES UNIVERSITY

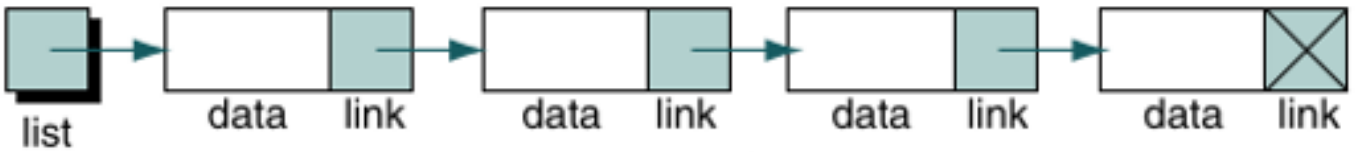
- We can use a linked list to create linear and non-linear structures.
  - In linear linked lists, each element has only zero or one successor.
  - In non-linear linked lists, each element can have zero, one, or more successors.
- The major advantage of the linked list over the array is that data are easily inserted and deleted.
- It is not necessary to shift elements of a linked list to make room for a new element or to delete an element.

Data Structures

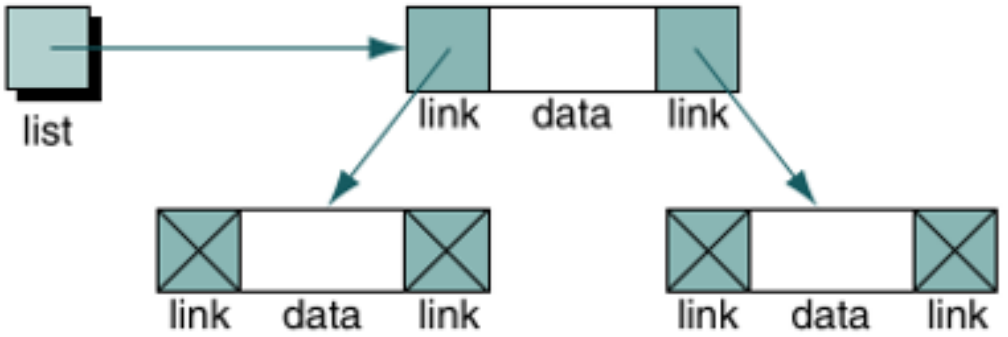
Types of Linked Lists

- **Singly Linked List**: Each node contains data and a pointer to the next node in the sequence.
- **Doubly Linked List**: Each node contains data, a pointer to the next node, and a pointer to the previous node.
- **Circular Linked List**: A variation of a singly linked list where the last node points back to the first node.
- **Circular Doubly Linked List**: A doubly linked list where the last node points back to the first node, and the first node points to the last node.
- **Skip List**: A layered linked list with additional forward pointers to allow fast traversal of the list. (List of Lists)

Linked Lists



(a) Linear list



(b) Non-linear list

(c) Empty list

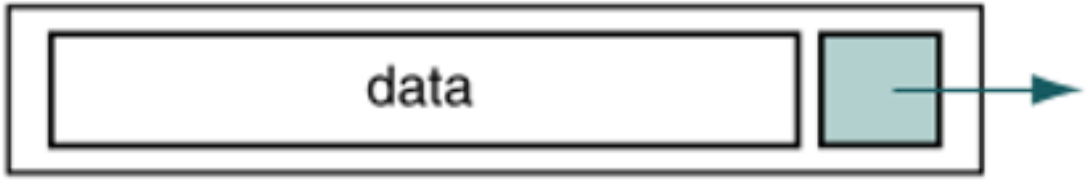
Src: Data Structures: A Pseudocode Approach with C,Richard F. Gilberg & Behrouz A. Forouzan

Data Structures

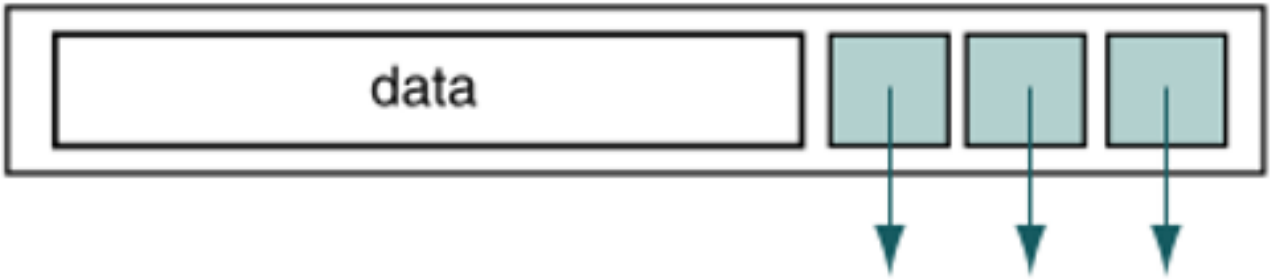
Linked Lists - Nodes

PES UNIVERSITY

(a) Node in a linear list



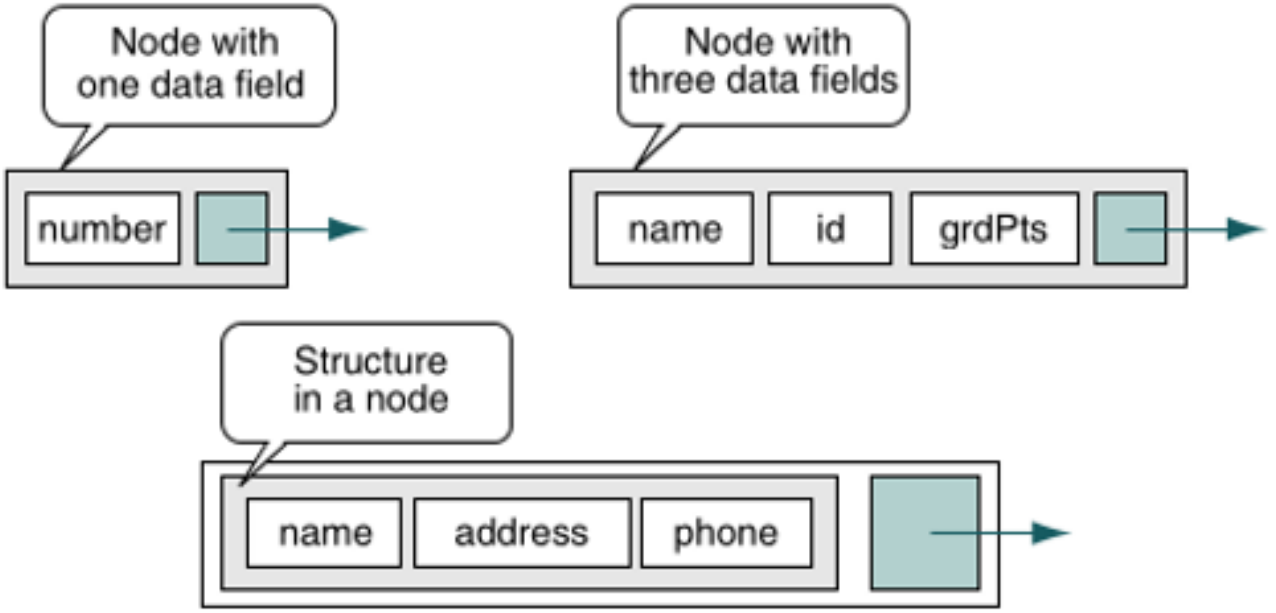
(b) Node in a non-linear list



Src: Data Structures: A Pseudocode Approach with C,Richard F. Gilberg & Behrouz A. Forouzan

Data Structures

Node



Src: Data Structures: A Pseudocode Approach with C,Richard F. Gilberg & Behrouz A. Forouzan

Data Structures

ADT for Linked Lists

- **Data Structure**: Node containing data and a Pointer to the next node
- **Primitive Operations:**
  - **Create:** Create an empty list, Create a node and initialize it
  - **Insert:** Insert nodes at the beginning, end, or at a specific position.
  - **Delete:** Delete a node by key, Delete a node by position, Delete the first node, Delete the last node.
  - **Update:** Update the value of a node by key.
  - **Search:** Search for nodes by key or position.
  - **Count:** Count the total number of nodes in the list.
  - **Display:** Print the entire list.
  - **Reverse:** Reverse the order of nodes.
  - **Destroy:** Delete all nodes in the list.

ADT for Linked Lists

- **1. Data Node Definition:**

```
typedef struct Node {
int data; struct Node *link; } *NODE;
 // Generic Data Type
typedef struct Node {
int data; struct Node *link; } *NODE;
```

Data Structures

ADT for Linked Lists

PES UNIVERSITY

- **2. ADT primitive operations :**
  - **NODE** CreateList()
  - **NODE** CreateNode(NODE HEADER, int Data);
  - **NODE** Insert_Front(NODE HEADER, int Data);
  - **NODE** Insert_Rear(NODE HEADER, int Data);
  - **NODE** Insert_Position(NODE HEADER, int Data);
  - **NODE** Insert_Order(NODE HEADER, int Data);

Note : All the functions take HEADER as parameter and return the same (in case of any modification to the HEADER)

Data Structures

ADT for Linked Lists

PES UNIVERSITY

- **2. ADT primitive operations :**
  - **NODE** Delete_First(NODE HEADER);
  - **NODE** Delete_Last(NODE HEADER);
  - **NODE** Delete_Position(NODE HEADER);
  - **NODE** Delete_Content(NODE HEADER);
  - **void** Display(NODE HEADER);
  - **int** Search(NODE HEADER);
  - **int** Find_Replace(NODE HEADER, int find, int replace);
  - **NODE** Reverse_List(NODE HEADER);
  - **NODE** Sort_List(NODE HEADER);
  - **void destroyList(Node head);**

Node Definition

```
1  # include<stdio.h>
2  # include<stdlib.h>
3
4  // Data type is constant
5  typedef struct mynode{
6      int data;
7      struct mynode *link;
8      } *NODE;
9
10 // Data type is variable
11 typedef struct mynode{
12     void *dataptr;
13     struct mynode *link;
14     } *NODE;
```

We are defining a NODE user defined data type

Node Creation and Initialization

```
1
2  NODE create_node(int data)
3  {
4      NODE temp = (NODE) malloc (sizeof(NODE));
5      if (temp != NULL)
6      {
7          temp->data = data;
8          temp->link = NULL;
9      }
10     else
11         printf("\n\t Unable to create a Node");
12     return temp;
13 }
```

Thank You

Dilip Kumar Maripuri
Associate Professor
Department of Computer Applications
dilip.maripuri@pes.edu
8073212026