**DATA STRUCTURES**

<Video no. 171>

|  |  |
| --- | --- |
| **Concept: Introduction of Data Structures**   * A data structure is the systematic way to organize data so that it can be used efficiently.   Example: **Arrays**  Instead of creating **multiple variables** of same type. Why not create an array to store all the values.  **Storing strings** is equivalent of storing a sequence of characters. This requires an array.  Some real life examples:  **Ex1: Stack** **data structure** is used in implementing redo and undo feature.  **Ex2: Array** **data structure** is used to store an image as a bitmap.  **Bitmap** images are stored as a series of tiny dots called pixels.  Here, a 2D array of size 37\*40 is enough to store this image.  **Ex3: Graph data structure** is used to store the friendship information on a social networking site  Here, James and Lucy are not friends so there is no edge between them |  |

<Video no. 172>

|  |  |
| --- | --- |
| **Concept: Data Types Vs. Abstract Data Types**  **Q. What is a Data Type?**  1. Defines a certain domain of values.  2. Defines Operations allowed on those values.  **Ex:**  int type   * Takes only integer values * Operations: addition, subtraction, multiplication, bitwise operations etc.   float type   * Takes only floating point values * Operations: addition, subtraction, multiplication, division etc. (bitwise and % operations are not allowed).   **User Defined Data Type:**  The operations and values of user defined data types are not specified in the language itself but are specified by the user.  **Ex: Structure, union and enumeration.**  By using structures, we are defining our own type by combining other data types.  struct point{  int x;  int y;  };  **Abstract Data Type(ADT):**  ADTs are like user defined data types which defines operations on values using functions without specifying what is there inside the function and how the operations are performed.  **Example: Stack ADT**  A stack consists of elements of same type arranged in a sequential order.  **Operations:**  Initialize() – initializing it to be empty  Push () – insert an element into the stack  Pop() – Delete an element from the stack  isEmpty() – check if stack is empty  isFull – check if stack is full   * Think of ADT as a **black box** which hides the inner structure and design of the data type from the user.   There are multiple ways to implement an ADT.  **Ex:**  **A stack ADT can be implemented using array or linked lists.**  **Why ADT?**   * The program which uses data structure is called a **client** program. * It has access to the ADT i.e. interface. * The program which implements the data structure is known as the **implementation**.   **Advantage:**   * Let say, if someone wants to use the stack in the program, then he can simple use push and pop operations without knowing its implementation. * Also, if in future, the implementation of stack is changed from array to linked list, then the client program will work in the same way without being affected.   **Conclusion:**   * **ADT provides Abstraction. (**which means that hiding details from the user, because user doesn’t bother about how that particular thing is implemented, just have to use that thing**)** |  |

<Video no. 173>

|  |  |
| --- | --- |
| **Concept: Data Structure Definition & Advantages**   * A data structure is the systematic way to organize data so that it can be used efficiently.   **Ex:** in order to implement stack ADT, we can use an array data structure or linked list data structure.    **Q. How to know which data structure to use for a particular ADT?**  In reality, different implementations of ADT are compared for time and space efficiency. The one best suited according to the current requirement of the user will be selected.  **Example:**    **Advantages of Data Structures:**  **Efficiency –** proper choice of data structure make program efficient in terms of space and time.  **Reusability –** one implementation can be used by multiple client programs.  **Abstraction –** Data structure is specified by an ADT which provides a level of abstraction. The client program doesn’t have to worry about the implementation details. |  |

<Video no. 174>

|  |  |
| --- | --- |
| **Concept: Types of Data Structure**   1. Linear Data Structure 2. Non Linear Data Structure 3. Linear Data Structures:   A data structure is linear when all the elements are arranged in a liner (sequential) order.   1. Non Linear Data Structures:   A data structure is non-linear when all the elements are not arranged in a liner (sequential) order. There is no linear arrangement of the elements.  **Further Distinction-**  Static Data Structures:  The memory is allocated at compile time (when creating code). Therefore, maximum size is fixed.  Advantage: Fast access  Disadvantage: Slower insertion and deletion  Dynamic Data Structures:  The memory is allocated at run time. Therefore, maximum size is flexible.  Advantage: Slower insertion and deletion  Disadvantage: Fast access |  |

<Video no. 185, 186, 187, 188>

|  |
| --- |
| **Concept: Understanding the Void Pointers.**  Q. What is a void pointer?  - **Void pointer** is a pointer which has no associated data type with it.  - It can point to any data type and can be **typecasted** to any type.  Example:  #include<stdio.h>  int main()  {  int n = 10;  void \*ptr = &n; // pointed to any data of any data type  printf("%d", \*(int\*)ptr); // ptr typecasted first then dereferencing  return 0;  }  **OUTPUT**: 10   * We cannot dereference a void pointer; first we have to typecaste then dereference it.   **Use of Void Pointer:**   * malloc and calloc(built- in functions) function returns a void pointer. Due to this reason, they can allocate a memory for any type of data. * malloc and calloc are used to allocate memory at runtime   Syntax: void\* malloc(size\_t size);  **Concept: Understanding the NULL Pointers.**  Q. What is a Null pointer?  - A **NULL pointer** is a pointer that does not point to any memory location. It represents an **invalid memory location**.  - When a NULL value is assigned to a pointer, then the pointer is considered as NULL pointer.  Example:  int main()  {  int \*ptr = NULL; <---- This is a Null Pointer  return 0;  }  **Use of NULL Pointer:**   1. It is used to initialize a pointer when that pointer isn’t assigned any valid memory address yet   Example:  int main()  {  int \*ptr = NULL;  return 0;  }   1. Useful for handling errors when using malloc function.   Example:  int main()  {  int \*ptr;  ptr = (int\*)malloc(2\*sizeof(int));  if(ptr == NULL)  printf("Memory could not be allocated");  else  printf("Memory allocated successfully");  return 0;  }  **Facts about NULL Pointer:**   1. The value is NULL is 0. We can either use NULL or 0 but this is written in context of pointers and is not equivalent to the integer 0.   Example:  #include<stdio.h>  int main()  {  int \*ptr = NULL;  printf("%d", ptr); //OUTPUT: 0  return 0;  }   1. Size of the NULL pointer depends upon the platform and is similar to the size of the normal pointers.   Example:  #include<stdio.h>  int main()  {  printf("%d", sizeof(NULL)); //OUTPUT: 8  return 0;  }  **Best Practices:**   * It is good practice to initialize a pointer as NULL. * It is a good practice to perform NULL check before dereferencing any pointer to avoid surprises.   **Concept: Understanding the Dangling Pointers.**  Q. What is a Dangling Pointer?  - A **dangling pointer** is a pointer which points to some non-existing memory location.  Example 1:  int main()  {  int \*prt = (int \*)malloc(sizeof(int)); //ptr contains the address of the first byte of that memory location  ...  ...  free(ptr); <---- Memory is now released (deallocating memory allocated by malloc function )  // but the pointer is still pointing to the deallocated memory  prt = NULL; // Now, ptr is **no more dangling**  return 0;  }  Example 2:  #include<stdio.h>  int\* fun()  {  int num = 10; // num variable 🡪 local to function so function finish its executions🡪 variable get vanished  return &num; // returning address of non- exciting memory  }  int main()  {  int \*prt = NULL;  ptr = fun(); // storing the address of ptr pointer --> ptr becomes dangling pointer  printf("%d", \*ptr); // dereferencing it cause segmentation fault(read or write an illegal memory)  return 0;  }    **Concept: Understanding the Wild Pointers.**  Q. What is a Wild Pointer?  - **Wild pointers** are also known as **uninitialized pointers.**  - These pointers usually point to some arbitrary memory location and may cause a program to crash or misbehave.  Example:  int main()  {  int \*p; <------ This is a WILD POINTER (it wildly behaves and may cause a program to crash or misbehave.)  \*p = 10; // this pointer may contain address of some illegal memory location  return 0;  }  **How to avoid wild Pointer?**   1. Initialize them with the address of a known variable.   Example:  int main()  {  int var = 10;  int \*p;  p = &var; <------- NO MORE A WILD POINTER  return 0;  }   1. Explicitly allocate the memory and put the values in the allocated memory   Example:  int main()  {  int \*p = (int \*)malloc(sizeof(int)); // allocated the memory  \*p = 10; // value in the allocated memory (it is legal)  free(p);  return 0;  } |

<Video no. 189, 190, 191, 192, 193>

|  |
| --- |
| **Concept: Basics of Dynamic Memory Allocation.**  **Static Memory Allocation:**   * Memory allocated during compile time (or time when programmer is creation the program) is called static memory. * The memory allocated is fixed and cannot be increased or decreased during run time.   Example:  int main()  {  int arr[5] = {1, 2, 3, 4, 5}; // memory is allocated at compile time and is fixed  }    **Dynamic Memory Allocation:**   * The process of allocating memory at the time of execution is called dynamic memory allocation.   Built in functions:   * malloc() * calloc() * realloc() * free()     **NOTE:**   * Pointers play an important role in dynamic memory allocation. Allocated memory can only be accessed through pointers.   **Concept: Dynamic Memory Allocation using malloc().**  **Q. What is malloc()?**  **-** malloc() is a built-in function declared in the header file <stdlib.h>  - malloc is the short name for ”memory allocation” and is used to dynamically allocate a single large block of contiguous memory according to the size specified.  Syntax: (void\*) malloc(size\_t size);   * malloc function simply allocates a memory block according to the size specified in the heap and on success it return a pointer pointing to the first byte of the allocated memory else return NULL. * size\_t is defined in <stdlib.h> as unsigned int.   Pictorial representation of Heap memory:  Just an imagination, and here some contiguous block of memory has been allocated by malloc and malloc will return a pointer, which is basically a void pointer assuming the pointer is ptr which is pointer pointing to a this particular memory.    **Q. Why malloc() returns void pointer?**  - malloc doesn’t have an idea of what it is pointing to.  - It merely allocates memory requested by the user without knowing the type of data to be stored inside the memory.  - The void pointer can be typecasted to an appropriate type  int \*ptr = (int \* ) malloc(4)  malloc allocates 4 bytes of memory in the heap and the address of the first byte is stored in the pointer ptr  Example:  #include<stdio.h>  #include <stdlib.h>  int main()  {  int i, n;  printf("Enter the number of integers: ");  scanf("%d", &n); // & --> n representing the value and requires the address  int \*ptr = (int \*)malloc(n\*sizeof(int)); //size of n integers -->malloc return a void pointer--> typecaste to an integer pointer    if(ptr == NULL)  {  printf("Memory not available.");  exit(1);  }  for(i=0; i<n; i++)  {  printf("Enter an integer: ");  scanf("%d", ptr + i); //ptr contains the first bytes of memory --> 1000+0 , 1000+1\*4=1004, 1008,...  // ptr --> already giving the address so no need of &  }  for(i=0; i<n; i++)  {  printf("%d ", \*(ptr+i)); //dereferencing it  }  return 0;  }  **OUTPUT:**  Enter the number of integers: 3  Enter an integer: 54  Enter an integer: 42  Enter an integer: 27  54 42 27  **Concept: Dynamic Memory Allocation using calloc().**  **Q. What is malloc()?**  **-** calloc() function is used to dynamically allocate multiple blocks of memory.  - calloc stands for “Clear Allocation”.  It is different from malloc in two ways:   * calloc() needs two arguments instead of just one   Syntax: (void\*) malloc(size\_t n, size\_t size);  size\_t n 🡪 Number of blocks  size\_t size 🡪 Size of each block  Example:     * Memory allocated by calloc is initialized to zero. It is not the case with malloc; malloc initialized with some garbage value.   **NOTE:**  Malloc and calloc both return NULL when sufficient memory is not available in the heap.  **Concept: Dynamic Memory Allocation using realloc().**  **Q. What is realloc()?**  - realloc() function is used to change the size of the memory block without losing the old data.  Syntax: void \*malloc(void \*ptr , size\_t newsize);  void \*ptr 🡪 Pointer to the previously allocated memory  size\_t newsize 🡪 New size   * On failure, realloc return NULL   Example:  int \*ptr = (int \*)malloc(sizeof(int)); //malloc allocates memory for 1 integer 🡪 4 bytes  ptr = (int \*)recallo(ptr, 2\*sizeof(int)); // recalloc the memory 🡪 pointer to previously allocated memory required  // second argument 🡪 new size(one more integer to be allocated 🡪 also consider previously allocated memory as well)   * This will allocate memory space of 2\*sizeof(int). * Also, this function moves the contents of the old block to a new block and the data of the old block is not lost. * We may lose the data when the new size is smaller than the old size. * Newly allocated bytes are uninitialized.   Example:  #include<stdio.h>  #include <stdlib.h>  int main()  {  int i;  int \*ptr = (int \*)malloc(2\*sizeof(int)); //malloc function allocating 8 bytes of memory, also return the address of the 1st byte of memory    if(ptr == NULL)  {  printf("Memory not available.");  exit(1);  }  printf("Enter the two integer: \n");  for(i=0; i<2; i++)  {  scanf("%d", ptr + i);  }    //memory allocation for 2 more integer  ptr = (int \*)realloc(ptr, 4\*sizeof(int)); //ptr--> ointer to the previously allocated memory  if(ptr == NULL)  {  printf("Memory not available.");  exit(1);  }  printf("Enter 2 more integers: \n");  for(i=2; i<4; i++)  {  scanf("%d", ptr + i);  }  //printing the value on the screen  for(i=0; i<4; i++)  {  printf("%d ", \*(ptr + i));  }  return 0;  }  **OUTPUT:**  Enter the two integer:  48  75  Enter 2 more integers:  42  11  48 75 42 11  **Concept: Releasing the dynamically allocated memory using free().**  **Q. What is free()?**  - free() function is used to release the dynamically allocated memory in heap.  Syntax: void free(ptr)   * The memory allocated in heap wil not be released automatically after using the memory the space remains there and can’ be used. * It is the programmer’s responsibility to release the memory after use.   Example :  int main()  {  int \*prt = (int \*)malloc(4\*sizeof(int));  ...  ...  free(ptr); //using free() we deallocate the memory  }  Program Example :  #include<stdio.h>  #include <stdlib.h>  int \*input() it is returning pointer to an integer  {  int \*ptr, i;  ptr = (int \*)malloc(5\*sizeof(int)); //malloc is allocating the memory for 5 integers 🡪 address returns to ptr  printf("Enter 5 numbers: ");  for(i=0; i<5; i++)  scanf("%d", ptr + i);  return ptr; //returning the address of the first byte of the allocated memory back to the main function  }  int main()  {  int i, sum=0;  int \*ptr = input(); // ptr pointer receives the address  for(i=0; i<5; i++)  sum += \*(ptr + i);    printf("sum is: %d", sum);  free(ptr); //releasing the memory ar the end  ptr = NULL; //good practice to assign the null ptr pointer after releasing  return 0;  }  **OUTPUT:**  Enter 5 numbers: 1 2 3 4 5  sum is: 15 |

<Video no. 194,195,196,197,198,199,200>

|  |
| --- |
| **Concept: Structure and Functions (Part 1).**  A **structure** is a user defined data type that can be used to group elements different types into a single type.  Example :  struct student  {  char name[50];  int age;  int roll\_no;  float marks;  };    **Concept: Passing structure member as argument.**  Just like variables, we can pass structure members as arguments to a function.  Example :  #include<stdio.h>  struct student  {  char name[50];  int age;  int roll\_no;  float marks;  };  void print(char name[], int age, int roll, float marks) //declaration of print function --> parameters will receive  {  printf("%s %d %d %.2f", name, age, roll,marks); //printf --> print all the values of the members  }  int main()  {  struct student s1 = {"Nick", 16, 50, 72.5}; //variable s1 of type struct student  print(s1.name, s1.age, s1.roll\_no, s1.marks); //print function --> passing the parameters  }  **OUTPUT**:  Nick 16 50 72.50  **Call by references:**  Instead of passing the copies of the structure members, we can pass their addresses (or references).  Example :  #include<stdio.h>  struct charset{  char s;  int i;  };  void keyvalue(char\* s, int\* i) // declaration of keyvalue function --> pointers receive the addresses  {  scanf("%c %d", s, i);  }  int main()  {  struct charset cs; //variable cs of type charset  keyvalue(&cs.s, &cs.i); //passing the address of struct members to the keyvalue function  printf("%c %d", cs.s, cs.i);  return 0;  }  **OUTPUT**:  A 6  A 6  **Structure and Functions (Part 2).**  **Concept: Passing structure variable as argument.**   * Instead of passing structure members individually, it is good practice to pass a structure variable as an argument. * Unlike arrays, name of the structure variable are not pointers. * It’s a **pass by value**   Example :  #include<stdio.h>  struct point{  int x;  int y;  };  void print(struct point p) // p --> parameter of function; receive the copy of p1  {  printf("%d %d\n", p.x, p.y); // print the values associated with p1 and then p2  }  int main()  {  struct point p1 = {23, 45}; //p1 variable initailzed with x =23 and y= 45  struct point p2 = {56, 90}; //p2 variable initailzed with x =56 and y= 90  print(p1); //calling function 🡪 passing p1 variable as argument (i.e. whole variable)🡪 call by value  print(p2);  return 0;  }  **OUTPUT**:  23 45  56 90  **Structure and Functions (Part 3).**  **Concept: Passing pointers to structures as argument.**  If the size of the structure is very large then passing the copy of the whole structure is not efficient.  BETTER CHOICE: Pass the address of the structure   * Use the arrow operator (->) to access the structure members inside the called function.   Example :  #include<stdio.h>  struct point{  int x;  int y;  };  void print(struct point \*ptr) //address of p1 is received by pointer  {  printf("%d %d\n", ptr->x, ptr->y); // initially, accessing the member value associated with p1  int main()  {  struct point p1 = {23, 45}; //p1 variable initialized with x =23 and y= 45  struct point p2 = {56, 90}; //p2 variable initialized with x =56 and y= 90  print(&p1); // passing addresses of p1 and p2 to print function one by one  print(&p2);  return 0;  }  **OUTPUT**:  23 45  56 90  **Structure and Functions (Part 4).**  **Concept: Returning a structure variable from the function.**  Returning a structure variable from a function is similar to returning a variable from a function.  Example :  #include<stdio.h>  struct point{  int x;  int y;  };  struct point edit(struct point p) //p contains the copy of p1 --> p.x=23 and p.y= 45  {  (p.x)++; //(p.x)++= p.x + 1 = 24  p.y = p.y + 5; // p.y = 50  return p;  }  void print(struct point p) {  printf("%d %d\n", p.x, p.y); // print the member value associated with p1 and p2 variables  }  int main()  {  struct point p1 = {23, 45}; //p1 variable initialized with x =23 and y= 45  struct point p2 = {56, 90}; //p2 variable initialized with x =56 and y= 90  p1 = edit(p1); // calling edit function -->passing the copy of p1 variable --> received by p above  p2 = edit(p2); // similarly as edit(p1)  print(p1); // calling print function --> member value associated with p1  print(p2);  return 0;  }  **OUTPUT**:  24 50  57 95  **Structure and Functions (Part 5).**  **Concept: Returning a pointer to a structure from the function.**  Example :  #include<stdio.h>  #include<stdlib.h>  struct point{  int x;  int y;  };  struct point\* fun(int a, int b) // here initialy, a will recive 2 and b=3  {  struct point \*ptr = (struct point \*)malloc(sizeof(struct point)); // malloc function --> allocate memory at Heap  (fix memory)  ptr->x = a; // ptr is pointing to heap memory --> passing values to heap memory  ptr->y = b+5;  return ptr; // fix memory so can easily return first byte of memory 🡪 after  finishing the function execution; memory remains in place  }  void print(struct point \*ptr)  {  printf("%d %d\n", ptr->x, ptr->y); // print the member value associated with p1 and p2 variables  }  int main()  {  struct point \*ptr1, \*ptr2; // \*ptr1, \*ptr2 --> point struct point data to  ptr1 = fun(2, 3); // calling fun function -->passing two arguments  ptr2 = fun(6, 9);  print(ptr1); print(ptr2);  free(ptr1); //deallocating memory  free(ptr2);  return 0;  }  **OUTPUT**:  2 8  6 14  **Structure and Functions (Part 6).**  **Concept: Passing Array of structures as argument.**  Example :  #include<stdio.h>  #include<stdlib.h>  struct point{  int x;  int y;  };  void print(struct point arr[]) //receiving the arr  {  int i;  for(i=0; i<2; i++)  printf("%d %d\n", arr[i].x, arr[i].y); //arr[0].x = 1, arr[0].y = 2  //arr[1].x = 3, arr[1].y = 4  }  int main()  {  struct point arr[2] = {{1, 2}, {3, 4}}; //declared an array of type struct point --> 2 elements (array of 2 structures)  print(arr); //calling print function --> passing complete arr  return 0;  }  **OUTPUT**:  1 2  3 4  **Structure and Functions (Part 7).**  **Concept: Self Referential structures.**  Self-referential structures are those structures in which one or more pointers points to the structure of the same type.  struct self  {  int p;  struct self \*ptr; //pointer pointing to a struct self itself  };  Example :  #include<stdio.h>  struct code{  int i;  char c;  struct code \*ptr; //ptr pointing to struct code  };  int main()  {  struct code var1;  struct code var2;    var1.i = 65;  var1.c = 'A';  var1.ptr = NULL; // ptr pointer it contains address--> initially containing NULL    var2.i = 66;  var2.c = 'B';  var2.ptr = NULL;    var1.ptr = &var2;  printf("%d %c", var1.ptr->i, var1.ptr->c); //var1.ptr means calling var2 🡪var1 contains address of var2  // var1.ptr->i 🡪 accessing members of var2 = 66  // var1.ptr->c 🡪 accessing members of var2 = B  }    **OUTPUT**:  66 B   * Self-referential structures are useful in LINKED LIST. |

<Video no. 201,202>

|  |  |
| --- | --- |
| **Concept: Introduction to Linked List.**  Ways to maintain a list in memory **:**  **Two ways:**   1. **Array** 2. **Linked list**   **Types of Linked List:**   * **Single linked list:** Navigation is forward only. * **Doubly linked list:** Forward and backward navigation   is possible.   * **Circular linked list:** Last element is linked to the first   element.  **Single linked list:**  A Single linked list is a list made up of nodes that consist of two parts.   * Data * Link   **Concept: Array vs. Single Linked List (in terms of representation).**   * **Store a list of numbers: 23, 54,78, 90**   Array Data structure:  In an array, elements are stored in consecutive memory location.  Array is the SEQUENTIAL representation of list.  Single Linked List:   * Pointer head pointing to first node of list. * Nodes are scattered here and there in memory but they are still connect to each other.   Linked list is the LINKED representation of list. |  |

<Video no. 203,204,205>

|  |  |
| --- | --- |
| **Concept: Creating the Node of a Single Linked List.**  For creating Node of a Single Linked List we use self referential structure.  struct node {  int data;  struct node \*link;  };  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  int main()  {  struct node \*head = NULL;  head = (struct node \*)malloc(sizeof(struct node));  head ->data = 45;  head -> link = NULL;  printf("%d", head ->data);  return 0;  }  OUTPUT: 45  **Concept: Creating a Single Linked List (Part 1).**  Creating a Single Linked List using two nodes:  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  int main()  {  //creating 1st node  struct node \*head = (struct node \*)malloc(sizeof(struct node));  head ->data = 45;  head -> link = NULL;  //creating 2nd node  struct node \*current =(struct node \*)malloc(sizeof(struct node));  current ->data = 98;  current -> link = NULL;  //to connect 1st node to 2nd node  head -> link = current;  return 0;  }  **Concept: Creating a Single Linked List (Part 2).**  Adding 3rd node to above list:  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  int main()  {  //creating 1st node  struct node \*head = (struct node \*)malloc(sizeof(struct node));  head ->data = 45;  head -> link = NULL;  //creating 2nd node  struct node \*current = (struct node \*)malloc(sizeof(struct node));  current ->data = 98;  current -> link = NULL;  head -> link = current;  // reusing current pointer for creating 3rd node  current = (struct node \*)malloc(sizeof(struct node));  current->data = 3;  current -> link = NULL;  //to connect 2nd node to 3rd node  head->link->link = current ;  return 0;  } |  |

<Video no. 206,207,208>

|  |  |
| --- | --- |
| **Concept: Traversing a Single Linked List (Counting the Node).**  Traversing a single linked list means visiting each node of a single linked list until the end node is reached.  Example: Program to count the number of nodes  (not from neso)  #include <stdio.h>  #include<stdlib.h>  //linked list node structure  struct node{  int data;  struct node\* link;  };    struct node\* head;  void insert(int data)  {  /\* Allocate memory\*/  struct node\* current = (struct node\*)malloc(sizeof(struct node));  current->data = data;  current->link = head;  head = current;  }  void print()  {  struct node\*current = head;  int count=0;  /\* Traverse the linked list and maintain the count. \*/  while(current!= NULL)  {  current = current->link;  count++;  }  printf("\n Total no. of nodes is %d",count);  }  int main(){  head = NULL;    insert(84);  insert(42);  insert(27);  /\* calling print function to print the count of node. \*/  print();  }  OUTPUT: Total no. of nodes is 3  **Concept: Traversing a Single Linked List (Printing the Data).**  Example: Program to print the data of linked list  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  void print\_data(struct node \*head)  {  if(head == NULL)  printf("Linked list is empty");  struct node \*ptr = NULL;  ptr = head;  while(ptr != NULL)  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  }  int main()  {  print\_data(head);  }  Output: 45 98 3 |  |
| **Concept: Traversing the List (Array vs. Linked).** | |

<Video no. 209,210,211>

|  |  |  |
| --- | --- | --- |
| **Concept: Single Linked List (Inserting Node at the End**  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  int main()  {  add\_at\_end(head, 67);  }  void add\_at\_end(struct node \*head, int data)  {  struct node \*ptr, \*temp;  ptr = head;  temp = (struct node \*)malloc(sizeof(struct node));  temp -> data = data;  temp -> link = NULL;  while(ptr->link != NULL)  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  ptr->link = temp;  } |  | |
| **Concept: Insertion at the end (Array vs. Linked)- Part1.**  (How much time will they take to insert a new element at end of single linked list?) | | |
| - Reduce time complexity(without traversal)  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  struct node \*add\_at\_end(struct node \*ptr, int data)  {  struct node \*temp = (struct node \*)malloc(sizeof(struct node));  temp -> data = data;  temp -> link = NULL;  ptr->link = temp;  return temp;  }  int main()  {  struct node \*head = (struct node \*)malloc(sizeof(struct node));  head->data = 45;  head->link =NULL;  struct node \*ptr = head;  ptr = add\_at\_end(ptr, 98);  ptr = add\_at\_end(ptr, 3);  ptr = add\_at\_end(ptr, 67);  ptr = head;  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  return 0;  }  - How much time will they take to insert a new element at end of array?  Example: **Case 1**: Array is not full  #include<stdio.h>  int add\_at\_end(int a[], int freePos,int data)  {  a[freePos] = data;  freePos++;  return freePos;  }  int main()  {  int a[10];  int i, n, freePos;  printf("Enter the number of elements: ");  scanf("%d", &n);  for(i=0; i<n; i++);  scanf("%d ",&a[i]);  freePos = n;  freePos = add\_at\_end(a, freePos, 65);  for(i=0; i<freePos; i++)  printf("%d ", a[i]);  return 0;  }  Example: **Case 2**: Array is full  #include<stdio.h>  int add\_at\_end(int a[], int b[], int n, int freePos,int data)  {  int i;  for(i=0; i<n; i++)  b[i] = a[i];  b[freePos] = data;  freePos++;  return freePos;  }  int main()  {  int a[3];  int i, n, freePos;  printf("Enter the number of elements: ");  scanf("%d", &n);  for(i=0; i<n; i++);  scanf("%d ",&a[i]);  int size = sizeof(a)/sizeof(a[0]);  freePos = n;  if(n == size) //Indicates array is full  {  int b[size+2];  freePos = add\_at\_end(a, b, size, freePos, 65);  for(i=0; i<freePos; i++)  printf("%d ", b[i]);  }  return 0;  } | |  |
|  | | |

<Video no. 212,213,214,215>

|  |  |  |
| --- | --- | --- |
| **Concept: Insertion Node at the Beginning of List**  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  struct node \*add\_beg(struct node \*head, int d)  {  struct node \*ptr = (struct node \*)malloc(sizeof(struct node));  ptr -> data = d;  ptr-> link = NULL;  ptr->link = head;  head = ptr;  return head;  }  int main()  {  struct node \*head=(struct node \*)malloc(sizeof(struct node));  head->data = 45;  head->link =NULL;  struct node \*ptr = (struct node \*)malloc(sizeof(struct node));  ptr->data = 98;  ptr->link =NULL;  head->link =ptr;  /\* adding node at beginning\*/  int data = 3;  head = add\_beg(head, data); // pass by value  ptr = head;  while(ptr != NULL)  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  return 0;  }  OUTPUT: 3 45 98  **Concept: Insertion Node at the Beginning of Single linked List(Possible mistake, - 2nd method)**  Example: **version 2.0**  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  struct node \*add\_beg(struct node \*\*head, int d)  {  struct node \*ptr = (struct node \*)malloc(sizeof(struct node));  ptr -> data = d;  ptr-> link = NULL;  ptr->link = \*head;  \*head = ptr;  }  int main()  {  struct node \*head = (struct node \*)malloc(sizeof(struct node));  head->data = 45;  head->link =NULL;  struct node \*ptr = (struct node \*)malloc(sizeof(struct node));  ptr->data = 98;  ptr->link =NULL;  head->link =ptr;  /\* adding node at beginning\*/  int data = 3;  add\_beg(&head, data); //pass by reference  ptr = head;  while(ptr != NULL)  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  return 0;  }  OUTPUT: 3 45 98 | |  |
| **Concept: Insertion Data at the Beginning (Single Linked List vs. Array)**   * Time complexity in adding a data at beginning in linked list. | | |
| * Time complexity in adding a data at beginning of array.   If there are n elements in the array, then n shifts are required.  Example:  #include<stdio.h>  int add\_beg(int arr[], int n, int data)  {  int i;  for(i=n-1; i>=0; i--)  {  arr[i+1] = arr[i];  }  arr[0] = data;  return n+1;  }  int main()  {  int arr[10], data = 10, i, n;  printf("Enter the number of elements: ");  scanf("%d", &n);  printf("Enter the elements: ");  for(i=0; i<n; i++);  scanf("%d ",&arr[i]);  n = add\_beg(arr, n, data);  for(i=0; i<n; i++)  printf("%d ", arr[i]);  return 0;  } |  | |
|  | | |

<Video no. 216,217,218>

|  |  |  |
| --- | --- | --- |
| **Concept: Insertion a Data at Certain Position.**  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  void add\_at\_end(struct node \*head, int data)  {  struct node \*ptr, \*temp;  ptr = head;  temp = (struct node \*)malloc(sizeof(struct node));  temp -> data = data;  temp -> link = NULL;  while(ptr->link != NULL)  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  ptr->link = temp;  }  void add\_at\_pos(struct node\* head, int data, int pos)  {  struct node \*ptr = head;  struct node \*ptr2 = (struct node \*)malloc(sizeof(struct node));  ptr2-> data = data;  ptr-> link = NULL;  pos--;  while(pos != 1)  {  ptr = ptr->link;  pos--;  }  ptr2->link = ptr->link; //doubt 🡪 not executing properly  ptr->link = ptr2;  }  int main()  {  struct node \*head = (struct node \*)malloc(sizeof(struct node));  head->data = 45;  head->link =NULL;  /\* adding node at end(previous videos)\*/  add\_at\_end(head, 98);  add\_at\_end(head, 3);  /\* adding node at certain position\*/  int data = 67, position = 3;  add\_at\_pos(head, data, position);  struct node \*ptr = head;  while(ptr != NULL)  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  return 0;  } | |  |
| **Concept: Insertion a Data at Certain Position (Single Linked List vs. Array) Part1-2.** | | |
| * **Insertion a Data at Certain Position in Array**   Example:  #include<stdio.h>  int add\_at\_pos(int arr[], int arr2[], int n, int data, int pos)  {  int i;  int index = pos-1;  for(i=0; i<=index-1; i++)  {  arr2[i] = arr[i];  }  arr2[index] = data;  int j;  for(i=index, j=index; i<11, j<10; i++, j++) //doubt 🡪 not executing properly  arr2[i] = arr[j];  }  int main()  {  int arr[] = {2, 34, 21, 6, 7, 8, 90, 67, 23, 39};  int pos = 5, data = 27, i;  int size = sizeof(arr)/sizeof(arr[0]);  int arr2[size+1];  add\_at\_pos(arr, arr2, size, data, pos);  for(i=0; i<size+1; i++)  printf("%d ", arr2[i]);  } |  | |
|  | | |

<Video no. 219,220,221, 222, 223, 224, 225>

|  |  |  |  |
| --- | --- | --- | --- |
| **Concept: Single Linked List (Deleting the first Node).**  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  struct node\* del\_first( struct node \*head)  {  if(head == NULL)  printf("List is already empty!");  else  {  struct node \*temp = head;  head = head->link;  free(temp);  }  return head;  }  int main()  {  //creating 1st node  struct node \*head = (struct node \*)malloc(sizeof(struct node));  head ->data = 45;  head -> link = NULL;  //creating 2nd node  struct node \*ptr = (struct node \*)malloc(sizeof(struct node));  ptr ->data = 98;  ptr -> link = NULL;  head -> link = ptr;  // reusing current pointer for creating 3rd node  ptr = (struct node \*)malloc(sizeof(struct node));  ptr->data = 3;  ptr -> link = NULL;  //to connect 2nd node to 3rd node  head->link->link = ptr;  /\* deleting of 1st node\*/  head = del\_first(head);  ptr = head;  while(ptr != NULL)  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  return 0;  }  OUTPUT: 98 3  **Concept: Single Linked List (Deleting the Last Node).**  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  struct node\* del\_last( struct node \*head)  {  if(head == NULL)  printf("List is already empty!");  else if(head->link == NULL)  {  free(head);  head = NULL;  }  else  {  struct node \*temp = head;  struct node \*temp2 = head;  while(temp->link != NULL)  {  temp2 = temp;  temp = temp->link;  }  temp2->link = NULL;  free(temp);  }  return head;  }  int main()  {  //creating 1st node  struct node \*head = (struct node \*)malloc(sizeof(struct node));  head ->data = 45;  head -> link = NULL;  //creating 2nd node  struct node \*ptr = (struct node \*)malloc(sizeof(struct node));  ptr ->data = 98;  ptr -> link = NULL;  head -> link = ptr;  // reusing current pointer for creating 3rd node  ptr = (struct node \*)malloc(sizeof(struct node));  ptr->data = 3;  ptr -> link = NULL;  //to connect 2nd node to 3rd node  head->link->link = ptr;  /\* deleting of last node\*/  head = del\_last(head);  ptr = head;  while(ptr != NULL)  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  return 0;  }  OUTPUT: 45 98  **NOTE:** There is no need to return the head pointer as we are not updating it in the function.  **Concept: Single Linked List (Deleting the Last Node using Single pointer).**  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  void del\_last( struct node \*head)  {  if(head == NULL)  printf("List is already empty!");  else if(head->link == NULL)  {  free(head);  head = NULL;  }  else  {  struct node \*temp = head;  while(temp->link ->link != NULL)  {  temp = temp->link;  }  free(temp->link);  temp->link = NULL;  }  }  int main()  {  //creating 1st node  struct node \*head = (struct node \*)malloc(sizeof(struct node));  head ->data = 45;  head -> link = NULL;  //creating 2nd node  struct node \*ptr = (struct node \*)malloc(sizeof(struct node));  ptr ->data = 98;  ptr -> link = NULL;  head -> link = ptr;  // reusing current pointer for creating 3rd node  ptr = (struct node \*)malloc(sizeof(struct node));  ptr->data = 3;  ptr -> link = NULL;  //to connect 2nd node to 3rd node  head->link->link = ptr;  /\* deleting of last node\*/  del\_last(head);  ptr = head;  while(ptr != NULL)  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  return 0;  }  OUTPUT: 45 98 | | |  |
| **Concept: Deletion at the End (Single Linked List vs. Array).**     * **Time complexity when deleting last element of array.**   Example:  #include<stdio.h>  int main()  {  int arr[] = {2, 34, 21, 6, 7, 8, 90, 67, 23, 39};  int size = sizeof(arr)/sizeof(arr[0]);  int i;  size = size -1;  for(i=0; i<size; i++)  printf("%d ", arr[i]);  return 0;  }  OUTPUT: 2 34 21 6 7 8 90 67 23 | | | |
| **Concept: Deletion at the Beginning (Single Linked List vs. Array).** | | | |
| * **Time complexity when deleting first element of array.**   Example:  #include<stdio.h>  int del\_first(int arr[], int n)  {  int i;  if(n == 0)  {  printf("Array is empty.");  return n;  }  for(i=0; i<=n-1; i++)  {  arr[i] = arr[i+1];  }  return n-1;  }  int main()  {  int arr[8];  int i, n;  printf("Enter the number of elements: ");  scanf("%d", &n);  for(i=0; i<n; i++)  scanf("%d", &arr[i]);    n= del\_first(arr, n);  for(i=0; i<n; i++)  printf("%d ", arr[i]);  }  OUTPUT:  Enter the number of elements: 6  23 2 5 56 32 54  2 5 56 32 54 |  | | |
|  | | | |
| **Concept: Single Linked List (Deleting the Last Node at a Particular Position).**  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  void del\_pos( struct node \*\*head, int position)  {  struct node \*current = \*head;  struct node \*previous = \*head;  if(\*head == NULL)  printf("List is already empty!");  else if(position == 1)  {  \*head = current->link;  free(current);  current == NULL;  }  else  {  while(position != 1)  {  previous = current;  current = current->link;  position--;  }    previous->link = current->link;  free(current);  current = NULL;  }  }  int main()  {  //creating 1st node  struct node \*head = (struct node \*)malloc(sizeof(struct node));  head ->data = 45;  head -> link = NULL;  //creating 2nd node  struct node \*ptr = (struct node \*)malloc(sizeof(struct node));  ptr ->data = 98;  ptr -> link = NULL;  head -> link = ptr;  // reusing current pointer for creating 3rd node  ptr = (struct node \*)malloc(sizeof(struct node));  ptr->data = 3;  ptr -> link = NULL;  //to connect 2nd node to 3rd node  head->link->link = ptr;  /\* deleting of node at particular position\*/  int position = 2;  del\_pos(&head, position);  ptr = head;  while(ptr != NULL)  {  printf("%d ", ptr->data);  ptr = ptr->link;  }  return 0;  }  OUTPUT:45 3  **Concept: Deleting the Entire Single Linked List.**  Example:  #include<stdio.h>  #include<stdlib.h>  struct node{  int data;  struct node \*link;  };  struct node\* del\_list( struct node \*head)  {  struct node\*temp = head;  while(temp!= NULL)  {  temp = temp->link;  free(head);  head = temp;  }  return head;  }  int main()  {  //creating 1st node  struct node \*head = (struct node \*)malloc(sizeof(struct node));  head ->data = 45;  head -> link = NULL;  //creating 2nd node  struct node \*ptr = (struct node \*)malloc(sizeof(struct node));  ptr ->data = 98;  ptr -> link = NULL;  head -> link = ptr;  // reusing current pointer for creating 3rd node  ptr = (struct node \*)malloc(sizeof(struct node));  ptr->data = 3;  ptr -> link = NULL;  //to connect 2nd node to 3rd node  head->link->link = ptr;  /\* deleting of entire list\*/  head = del\_list(head);  if(head == NULL)  printf("linked list deleted successfully");  return 0;  }  OUTPUT: linked list deleted successfully | |  | |

DOUBT – 216, 218