**Data Structures and Algorithms**

* **Data Structures**
  + **Array**
  + **Linked List**
  + **Stack**
  + **Queue**
  + **Heap**
  + **Tree**
  + **Graph**
* **Algorithms**
  + **Searching Algorithm**
    - **Linear Search**
    - **Binary Search**
  + **Sorting Algorithm**
    - **Bubble Sort**
    - **Selection Sort**
    - **Insertion Sort**
    - **Quick Sort**
  + **Divide and Conquer Algorithm**
  + **Recursion**
  + **Backtracking**

**Week - 19.12.2022**

**Data Structures**

**– *Array***

An array is a way of storing a group of things that are all the same. They are put together in a certain order, and you can easily get to any of them using a number called an "index."

*int myArray[5] = {10,20,30,40,50}; //Declaration and initialization of an array*

***Operations in Array***

**1-Traverse**

Printing all array elements

*for (int i = 0; i < 5; i++) {*

*printf("myArray[%d] = %d\n", i, myArray[i]); //print the array elements*

*}*

**2-Insertion**

Inserting an element at a specific position

void insert(int pos, int val)

{

for (int i = size; i > pos; i--)

{

a[i] = a[i-1];

}

a[pos] = val;

size++;

}

**3-Deletion**

Deleting an element from a specific position

*void delete(int pos)*

*{*

*for (int i = pos; i < size-1; i++)*

*{*

*a[i] = a[i+1];*

*}*

*size--;*

*}*

**4-Search**

Searching for an element in an array

*int search(int val)*

*{*

*for (int i = 0; i < size; i++)*

*{*

*if (a[i] == val)*

*{*

*return i;*

*}*

*}*

*return -1; //if element is not found*

*}*

**5-Update**

Updating an element at a specific position

*void update(int pos, int val)*

*{*

*a[pos] = val;*

*}*

***Linked List***

A Linked List is a way of storing data in a specific order, similar to an array. Each data collection (called a node) is connected to the next one by a pointer (or previous one). Each node has its own data and a pointer to the next node.

***Simple Linked List***

It is also called single linked list. It is a type of linked list which can only be traversed only one direction. Each pointer of nodes points next node. The last pointer points NULL.

***Doubly Linked List***

Linked list which can be traversed in both directions as either previous or next pointers.

***Circular Linked List***

Last node’s pointer points the head of the linked list.

***Doubly Circular Linked List***

***Representation of Simple Linked List***

Typedef struct Node {

    int data;

    struct Node\* next;

}Node;

**Operations in Linked List**

**1-Insertion**

Inserting a new node at the front, rear or specific position

**void insertFront(int val)**

*{*

*Node\* temp = new Node();*

*temp->data = val;*

*temp->next = head;*

*head = temp;*

*}*

**2-Deletion**

Deleting a node from the front, rear or specific position

**void deleteFront()**

*{*

*Node\* temp = head;*

*head = head->next;*

*delete temp;*

*}*

**3-Search**

Searching for a specific value in a linked list

*bool search(int val)*

*{*

*Node\* temp = head;*

*while (temp != NULL)*

*{*

*if (temp->data == val)*

*{*

*return true;*

*}*

*temp = temp->next;*

*}*

*return false;*

*}*

**4-Update**

Updating the value of a specific node

**void update(int pos, int val)**

*{*

*Node\* temp = head;*

*for (int i = 0; i < pos; i++)*

*{*

*temp = temp->next;*

*}*

*temp->data = val;*

*}*

***Stack***

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. It has two main operations: push, which adds an element to the top of the stack, and pop, which removes the element from the top of the stack.

**Operations in Stack**

**1-Push**

Adding an element to the top of the stack

**2-Pop**

Removing the element from the top of the stack

**3-Peek**

Accessing the element at the top of the stack

**4-isEmpty**

Checking if the stack is empty

*bool isEmpty()*

*{*

*return top < 0;*

*}*

***-Queue***

A queue is a linear data structure that follows the First In First Out (FIFO) principle.

***Operations in Queue***

**1-Enqueue**

Adding an element to the rear of the queue

**2-Dequeue**

Removing the element from the front of the queue

**3-Peek**

Accessing the element at the front of the queue

**4-isEmpty**

Checking if the queue is empty

***-Heap***

A heap is a complete binary tree that satisfies the heap property.

**Operations in Heap**

**1-Insertion**

Inserting a new key in the heap.

**2-Deletion**

Deleting the root of the heap.

**3-Extract Max/Min**

Extracting the maximum/minimum element from a max/min heap.

**4-Increase/Decrease Key**

Increasing or decreasing the value of a specific key in the heap.

***-Tree***

A tree is a data structure that consists of a set of nodes that are connected by edges. It has a root node, which has zero or more child nodes, and so on.

**Operations in Tree**

**1-Insertion**

Inserting a new node in a tree

**2-Deletion**

Deleting a specific node in a tree

**3-Search**

Searching for a specific value in a tree

***-Graph***

A graph is a non-linear data structure that consists of a set of nodes and edges that connect them.

**Operations in Graph**

**1-Add Vertex**

Adding a new vertex (node) to a graph.

**2-Add Edge**

Adding a new edge between two vertices in a graph.

**3-Search**

Searching for a specific vertex or edge in a graph.

**Week – 26.12.2022**

***Searching Algorithm***

**o Linear Search**

Linear search is an algorithm that searches for a specific value in an array by iterating through each element one by one until the desired value is found.

*int linearSearch(int arr[], int n, int x)*

*{*

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

*{*

*if (arr[i] == x)*

*{*

*return i;*

*}*

*}*

*return -1;*

*}*

**o Binary Search**

Binary search is an algorithm that searches for a specific value in a sorted array by repeatedly dividing the search interval in half.

**Sorting Algorithm**

**o Bubble sort**

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares next elements and swaps them if they are in the wrong order.

*void bubbleSort(int arr[], int n)*

*{ for (int i = 0; i < n-1; i++)*

*{*

*for (int j = 0; j < n-i-1; j++)*

*{*

*if (arr[j] > arr[j+1])*

*{ swap(arr[j], arr[j+1]);*

*}*

*}*

*}*

*}*

**o Selection sort**

Selection sort is a comparison based sorting algorithm that repeatedly selects the next smallest element and moves it to its correctt position.

*void selectionSort(int arr[], int n)*

*{*

*int i, j, min\_idx;*

*for (i = 0; i < n-1; i++)*

*{*

*min\_idx = i;*

*for (j = i+1; j < n; j++)*

*{*

*if (arr[j] < arr[min\_idx])*

*{*

*min\_idx = j;*

*}*

*}*

*swap(arr[min\_idx], arr[i]);*

*}*

*}*

**o Insertion sort**

Insertion sort is a sorting algorithm that builds the final sorted list one item at a time by repeatedly inserting the next element into its correct position in the already sorted list.

*void insertionSort(int arr[], int n)*

*{*

*int i, key, j;*

*for (i = 1; i < n; i++)*

*{*

*key = arr[i];*

*j = i-1;*

*while (j >= 0 && arr[j] > key)*

*{*

*arr[j+1] = arr[j];*

*j = j-1;*

*}*

*arr[j+1] = key;*

*}*

*}*

**Divide and Conquer Algorithm**

**o Binary search**

Binary search is a divide and conquer algorithm that searches for a specific value in a sorted array by repeatedly dividing the search interval in half.

**o Merge sort**

Merge sort is a divide and conquer algorithm that sorts an array by dividing it into two halves, sorting each half, and then merging the two sorted halves back together.

**o Quick sort**

Quick sort is a divide-and-conquer algorithm that selects a 'pivot element' from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

**Week – 02.01.2023**

**Contents**

* **Call by value/ Call by reference**
* **Git**
* **array/readme.md 🡪 All the functions were written.**

**Call by value/ Call by reference**

Call by value and call by reference are two ways in which parameters can be passed to a function in programming languages.

*Call by value:*

In call by value, a copy of the value of the argument is passed to the function. This means that any changes made to the parameter within the function will not affect the original argument.

*void addFive(int x)*

*{*

*x = x + 5;*

*}*

*int main()*

*{*

*int num = 10;*

*addFive(num);*

*}*

In above example the value of num variable will not be changed due to passing it by value.

*Call by reference:*

In call by reference, a reference to the memory location of the argument is passed to the function. This means that any changes made to the parameter within the function will affect the original argument.

*void addFive(int \*x)*

*{*

*\*x = \*x + 5;*

*}*

*int main()*

*{*

*int num = 10;*

*addFive(&num);*

*}*

However, in above example the value of num will be changed because it is passed to the function by reference.

In summary, the main difference between call by value and call by reference is that call by value passes a copy of the argument to the function, while call by reference passes a reference to the memory location of the argument.

**Git**

**Setting up a Git repository**

A Git repository is a place where storing the files and tracking their changes were made. To set up a new repository, the *git init* command can be used. This will create a new repository in the current directory.

*git init*

**Setting User Name and Email)**

Before start to use git, configuration of user name and email must be done. This information will be used to identify as the author of the commits were made.

*git config --global user.name "Name"*

*git config --global user.email " email@example.com"*

**Basic Workflow**

**Adding and committing changes**

Once the repository has been set up, adding files and making changes can be done. To add a new file to the repository, the *git add* command is used.

*git add newfile.txt*

Once the changes and added the files have been made, commit them to the repository using the *git commit* command.

*git commit -m "Added a new file"*

**Viewing the commit history**

To view the commit history of the repository, the git log command is used. This will display a list of all the commits made to the repository, along with the author, date, and commit message.

*git log*

**Branching and Merging**

**Creating and switching between branches**

To create a new branch, use the git branch command.

*git branch new-branch*

*git checkout* command is used to switching to an existing branch.

*git checkout new-branch*

**Merging branches**

To merge with main branch,

*git checkout main-branch*

*git merge new-branch*

**Week – 09.01.2023**

**Contents**

* **What is a programming language?**
* **Compilers and Interpreters**
* **Linker/Linking**
* **linked\_list/readme.md 🡪 Some of the functions were written**

**Programming Language**

A programming language is a set of instructions and rules that a computer can understand and execute.

**Compilers and Interpreters**

A compiler is a program that translates the source code of a programming language into machine code (binary code) that can be executed by a computer.

An interpreter is a program that interprets the source code of a programming language, executing each instruction as it reads it. Interpreters are typically slower than compilers.

**Linker/Linking**

A linker is a program that takes the object files generated by a compiler and combines them into a single executable file. A linker's job is to resolve any external references within the object files, such as calls to functions or variables defined in other object files.

Linking is the process of using a linker to combine multiple object files into a single executable file. The linker will also resolve any external references, such as external libraries, that are needed by the program.

In summary, programming languages are the set of instructions and rules that a computer can understand and execute, compilers are a program that translate the source code of a programming language into machine code, and interpreters are a program that interprets the source code of a programming language, executing each instruction as it reads it. Linker and Linking are used to combine multiple object files into a single executable file, resolving any external references and dependencies.

**Week – 16.01.2023**

**Contents**

* **lvalue and rvalue**
* **linked\_list/readme.md 🡪 All of the functions were written (naming of variables, functions were discussed. Some standards were also discussed.)**

**lvalue and rvalue**

lvalue represents a memory location while rvalue represents temporary value that is stored in some memory.

**Lvalue**

Variables as: int x;

Array elements: array[10] is a lvalue

Pointer variables: Since a pointer variable holds the memory address of another variable it is a lvalue: int x; int\* ptr = &x; here &x is a lvalue.

struct S { int m; };

struct S trial; //trial.m and trial are l value

int arr[20]; // arr[10] is an lvalue

int \*p, \*q; //\*p, \*q are lvalue

**rvalue**

Constants like 5, 6 are rvalue.

Variables: int x = 1; int y = 2; int z = x+y; in here x+y is an rvalue

Function calls: int z = add(x,y); add(x,y) is an rvalue

Pointer dereference: y = \*ptr; \*ptr is a rvalue

**Important Notes**

* The assignment operator (=) can only be used on lvalues, not rvalues. Because left side of the assignment operator must be memory location.
* The address-of operator (&) can only be applied to lvalues, as it returns the memory address of the lvalue.
* Pointers can only be assigned to lvalues, because they must refer to a memory location.

**Example**

p = &a; //correct because assignment of address to an l-value

&a = p; //error because &a is an r-value

**Week – 23.01.2023**

**Contents**

* **Stack with Linked List**
* **Stack with Array**
* **Heap and Stack Memory**
* **Time and Space Complexity**
* **Undefined Variables, Unspecified Variables**
* **Queue with Linked List**
* **Queue with Array**
* **Recursion…**

**Stack – Data Structure**

Stack is a data structure that follows last in first out (LIFO) principle. In real life it is suitable to give some examples. Removing the books that are piled up. Because you need to remove the last book in order to remove one by one.

In stack it is important to have pointer points to top element. Because the top element is the last element that is inserted. We only need access to last element of stack.

**Stack Operations:**

* **push()** to insert an element into the stack.
* **pop()** to remove an element from the stack.
* **top()** Returns the top element of the stack.
* **isEmpty()** returns true if stack is empty else false.
* **size()** returns the size of stack.

**Heap and Stack Memory**

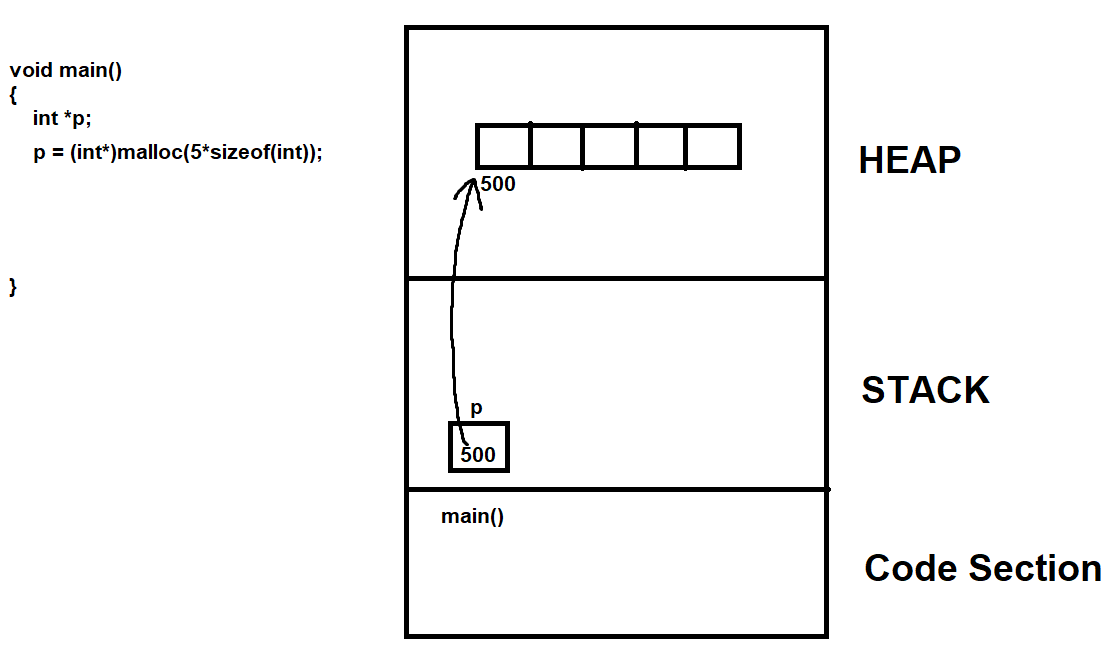
Memory is divided into smaller addressable units that are called as bytes. As it depends on the memory, every byte will have its own address.

Entire memory is not used as single unit, but it is divided into manageable pieces that are called as segments.

Entire memory is divided into three sections and used by a program: Code section, Stack and Heap.

When you want to run a program first the program, the machine code of the program, should be brought inside the main memory. So, the area that is occupied by the program in the main memory called the code section. Once it is loaded the CPU will start the execute the program. And then the program will utilize the remaining memory as divided into Stack and Heap.

Whatever the variables you declare inside your program, or inside a function the memory for those variables will be created inside Stack. So, the memory is allocated inside the stack depends on the variables inside a function. So, this is static memory allocation. The definition of static can be made as how many bytes of memory is required by the function was decided at compile time.

Heap means piling up. The program cannot directly access to heap memory. It can only access by pointers.

It is a good practice that when you don’t need that memory created in heap, you should de-allocate it.

**Time and Space Complexity**

Time complexity of an algorithm can be defined as the amount of time required to execute an algorithm. Similarly, space complexity of an algorithm is a measure of amount of space or memory taken by an algorithm.

**Time Complexity**

**-** The time taken by simple statements is constant,

int i = 0;

i = i + 3;

Time complexity for each can be represented as O(1). Thus: O(2) for both.

**-** for(i=0; i< n; i++)

{

printf(“%d”,i);

}

Time complexity for for loop can be represented as O(n). Since the function printf will be executed n times. If there were 2 nested for loops then the function would be executed n times by first loop and n times by the second loop. Therefore, it would be O(n2).

So, all in all, time complexity is expressed using mathematical notations. These notations represent the time required by an algorithm.

**Undefined** / **Unspecified Variables**

An undefined variable is a variable that has been declared, but does not have a value assigned to it. Attempting to use an undefined variable can result in undefined behavior. Undefined behavior occurs when a program breaks the rules of the language in a way that is not detected until runtime. It's important to initialize variables before using them in the program or to assign a value before using them.

*int x;*

*printf("%d", x); // x is undefined, and using it can result in unexpected output or a fault*

The variable x is declared but not assigned any value, so it's undefined.

Unspecified variable is a variable that has been declared but its value or state is not specified. It can contain any value, and its initial value is not predictable. This can lead to undefined behavior. Unspecified behavior arises when a program follows all the rules but the standard doesn’t completely specify what a particular operation does.

This is not the same as an uninitialized variable, which has not been assigned a value yet. An uninitialized variable will have an indeterminate value, and using it leads to undefined behavior in the program.

*int x;*

*if (condition) {*

*x = 5;*

*}*

*printf("%d", x); //x's value is unspecified, it can be any value can lead to unexpected behavior*

Or

*int x,y;*

*if(&x > &y)*

*{*

*}*

*In above example unspecified behavior would occur. Since it is not possible to be sure whether x has bigger address than y. So the comparison &x > &y is unspecified.*

The variable x is declared but its value is not specified, so it can contain any value, this is an unspecified variable.