Motivation

→ Computers do store, retrieve, and process a large amount of data.

→ If the data is stored in well organized way on storage media and in computer's memory, it can be accessed quickly for processing and the user is provided fast response.

What / Why is Data Structure?

→ Data Structure is a way of collecting and organizing data is such a way that we can perform operations on these data in a effective way. In simple language, Data Structure are structures programmed to store ordered data, so that various operations can be performed on it easily.

Data Structure, Algorithm, Function.

→ A data structure should be seen as a logical concept that must address two fundamental concerns.

→ 1) How the data will be stored.
2) What operations will be performed on it?

→ Data Structure is a scheme for data organization

→ The functional defination of a data structure should be independant of it's implementation

→ The functional defination of a data structure is known as ADT (Abstract Data Type), which is independant of implementation.

→ The implementation part is left to the developers who decide which technology better suits to their project needs

Eg: a stack ADT is a structure which supports operations such as push() and pop()

→ A stack can be implemented in a number of ways, like using array or linked list.

→ Along with data structures, in real life, problem solving is done with the help of Algorithms

→ An algorithm is a step-by-step process, to solve a problem.

→ finiteness: The algorithm must always terminate after a finite number of steps.

→ definiteness: Each step much be precisely defined.

→ input: An algorithm has zero or more inputs, taken from a specified set of objects.

→ output: An algorithm has one or more outputs, which have a specified relation to the inputs.

→ effectiveness: It's operations must be basic enough to be able to be done exactly in a finite length of time, for example, somebody using pencil and paper

→ In programming, algorithms are implemented in form of methods or functions or routines

→ To get a problem solved we not only want algorithm but also an efficient algorithm

→ One criteria of efficiency is time taken by the algorithm, another could be memory it takes at run time.

→ We can have more than one algorithm for the same problem

→ The best algorithm is the one which has a fine balance between time taken and memory consumption.

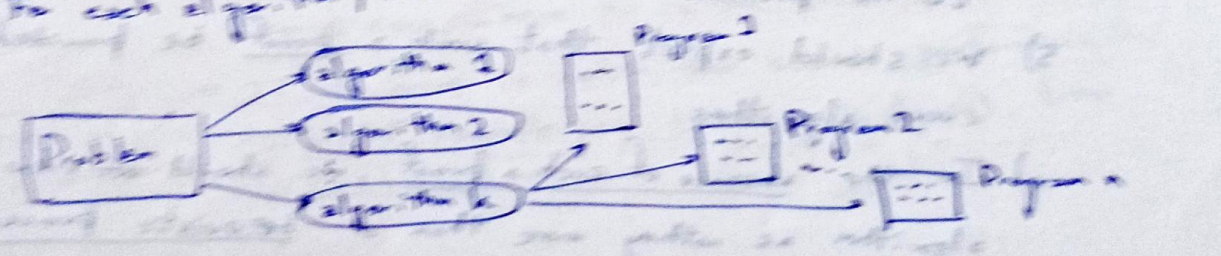
→ Sadly, Best never exists ☹

→ We usually give more priority to the time taken by the algorithm rather than the memory it consumes

Problems vs Algorithms vs Programs

→ For each problem or class of problems there may be different algorithms

→ For each algorithm, there may be different implementations (Programs)



Expressing algorithms

→ Natural Language - usually verbose and ambiguous

→ Flow Charts - avoids most (if not all) issues of ambiguity

→ Pseudo-Codes - also avoids most issues of ambiguity; resembles common elements of programming languages; no particular agreement on syntax

→ Programming Language - tend to require expressing low level details that are not necessary for a high level understanding

Testing Correctness

→ How do we know whether an algorithm is actually correct?

1) The logical analysis of the problem we performed in order to design the algorithm should give us confidence that we have identified a valid procedure for finding a solution.

2) We can test the algorithm by choosing different sets of input values and checking to see if the resulting does, in fact, work.

3) But, no matter how much testing we do, unless there are only a finite number of possible input values for the algorithm, testing can never prove that the algorithm produces correct results in all cases.

4) We can attempt to construct a formal, mathematical proof that, if the algorithm is given valid input values then the results obtained from the algorithm must be a solution to the problem

5) We should expect that such a proof be provided for every algorithms

6) In the absence of such a proof, we should ~~wi~~ view the algorithm as nothing more than a heuristic procedure, which may or may not yield the expected results.

Classification of Data Structures

→ Data structures can be broadly classified in two categories

→ Linear Data Structure: Arrays, linked lists, stack and queue

→ Hierarchical Data Structure: trees, graphs and heaps.

→ Every Data Structure has it's own strengths and weaknesses

→ Every data structure specially suits to specific problem depending upon the operations performed and the data organization

Arrays

→ Arrays are by far the most common data structure used to store data.

→ Arrays are generally statically implemented data structure by some programming languages.

→ The size of this data structure must be known at compile time and cannot be altered at run time.

→ Although, few programming languages implement arrays as objects and give the programmer a way to alter the size of them at run time.

→ Disadvantages

→ Inserting an item to an array

→ Deleting an item to an array

→ The time taken by insert operation depends on how big the array is, and at which position the item is being inserted.

→ Advantages

→ Searching → If the array is unsorted the search operation is also proved costly and takes O(N) time in worst case, where N is the size of the array

→ If the array is sorted then search performance is improved magically an takes O(log N) time in worst case.

Linked List

→ Linked List data structure provides better memory management than arrays

→ Linked List is allocated memory at run time, so there is no waste of memory

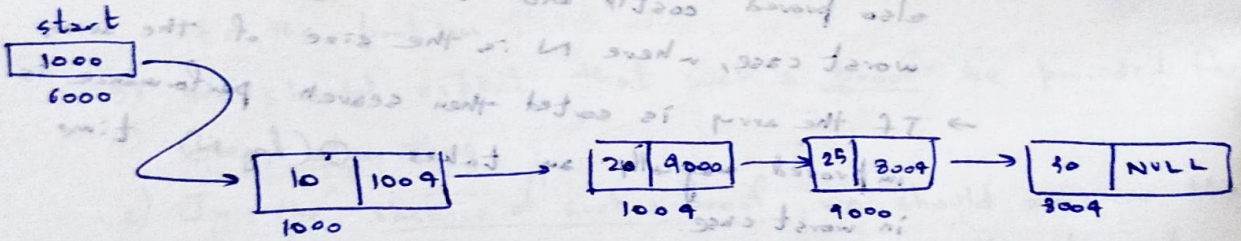
→ Advantages

→ Linked List is proved to be a useful data structure when the no. of elements to be stored is not known ahead of time.

→ Disadvantages

→ Performance wise linked list is slower than array because there is not direct access to linked list elements

→ There are many flavours of linked list: linear, circular, doubly and doubly circular.

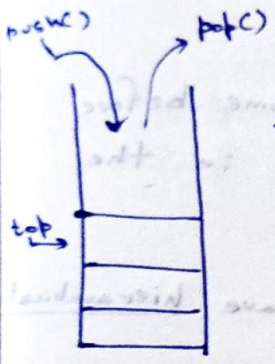


Stack

→ Stack is a last-in-first-out strategy data structure.
→ The element stored in last will be removed first.

→ Stack has specific but very useful applications;

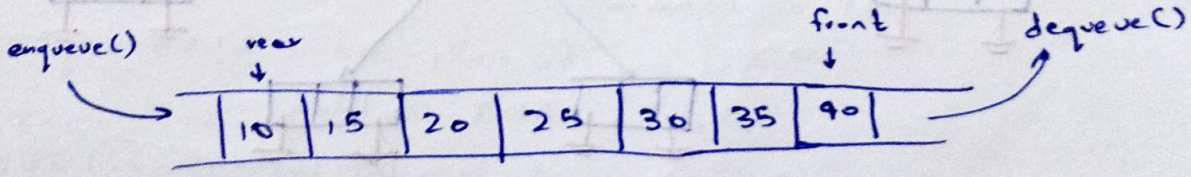
→ Solving recursion → recursive calls are place onto a stack, and removed from there once they are processed.



→ Evaluating Post-Fix expressions
→ Solving Tower of Hanoi
→ Back tracking
→ Depth-First Search
→ Converting a decimal number into a binary number.

Queue

→ Queue is a first-in-first-out data structure
→ The element that is added to the queue data structure first, will be removed from the queue first.
→ Dequeue, priority queue, and circular queue are the variants of queue data structure.
→ Queue has following application uses:
→ Access to shared resources (Eg Printer)
→ Multiprogramming (ready queue)
→ Message queue



Trees

→ Tree is a hierarchical data structure

→ The very top element of a tree is called the root of the tree

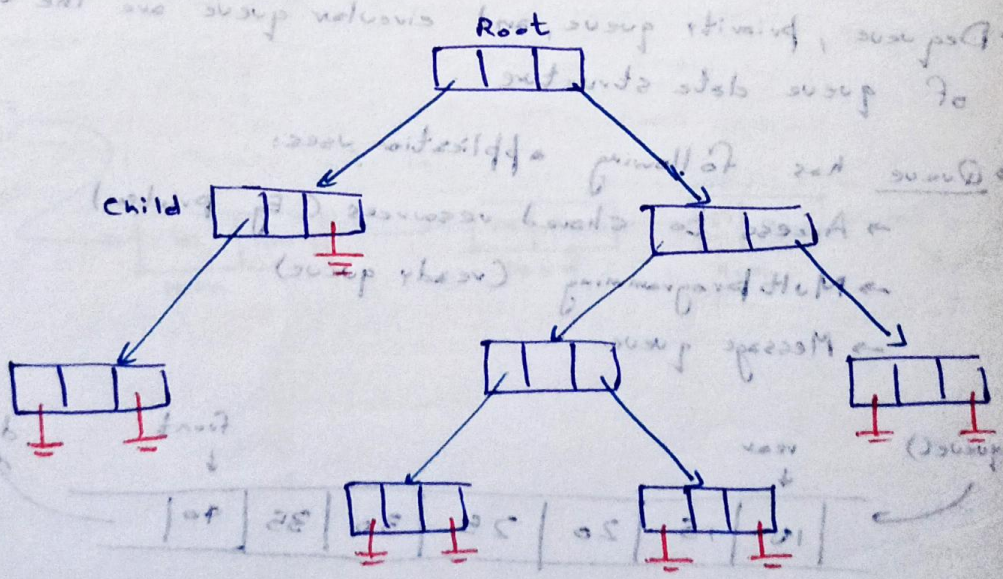
→ Except the root element, every element in a tree has a parent, and zero or more children

→ All the elements in the left sub-tree come before the root in sorting order, and all those in the right sub-tree comes after the root

→ Tree is most useful data structure when you have hierarchical information to store.

Eg: Directory structure of a file system.

→ Some variants of tree data structure are Red-Black tree, threaded Binary tree, AVL tree, etc.



Heap

→ **Heap** is a **Binary Tree** that stores a collection of keys by satisfying Heap property

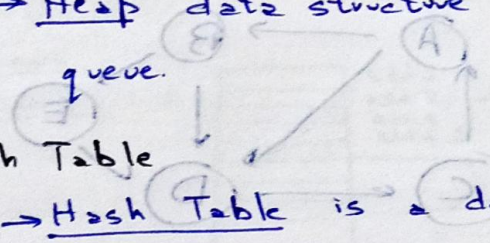
→ **Heap Property**:

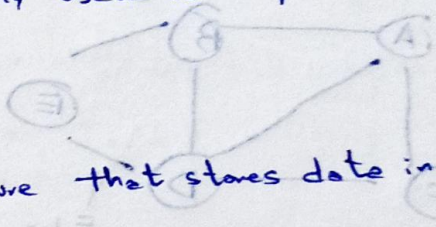
→ There are two flavors of heap data structure (Max Heap and ~~Mean~~ min Heap)

→ For **max heap**, each node should be greater than or equal to each of it's children.

→ For **min heap**, each node should be less than or equal to each of it's children.

→ **Heap** data structure are usually used to implement priority queue.





Hash Table

→ **Hash Table** is a data structure that stores data in form of key-element pairs

→ A key is a non-null value which is mapped to an element

→ The element is accessed on the basis of the key associated with it

→ **Hash Table** is a useful data structure for implementing **dictionary**

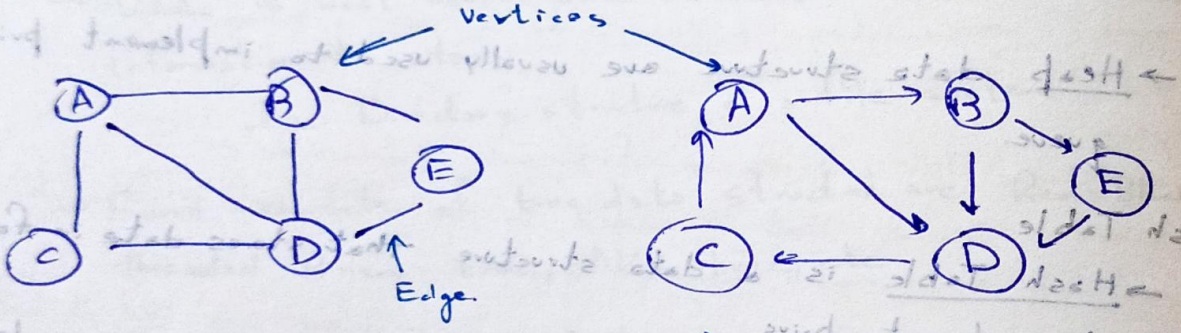
Graph

→ Graph is a networked data structure that connects a collection of nodes called vertices, by connections, called edges

→ An edge can be seen as a path or communication link between two nodes.

→ edges can be either directed or undirected

→ In a directed path, you can move in one direction only, while in an undirected path, the movement is possible in both directions.



a) Undirected graph b) Directed Graph

Abstract Data Types and Data Structures

→ often, these terms are used as synonyms, but it's better to think of them this way:

→ An abstract Data Type (ADT) represents a particular set of behaviors.

→ You can formally define (i.e, using maths logic) what an ADT is/does.

→ Eg. a stack is a list implements a LIFO policy on addition/delition.

→ A data structure is more concrete. Typically, it is a technique or strategy for implement ADT

Eg: using Linked List or Array to implemen stack

→ Going one level lower, we get into particulars of programming languages and libraries

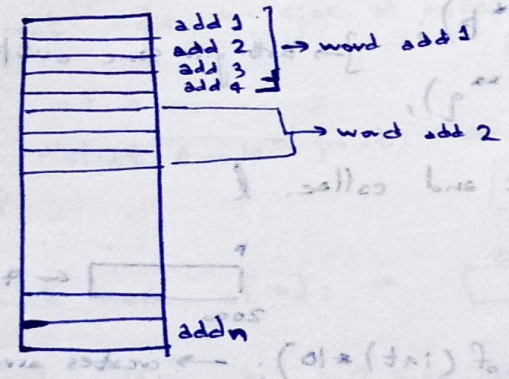
Eg: use java.lang.Vector or java.Util.Stack or a C library from STL

→ Some common ADT's that all trained programmers know about:
  → Stack, queue, priority queue, dictionary, sequence, set

→ Some common data structures used to implement ADT's:
  → array, linked lists, hash table (open, closed, circular hash)
  → trees (binary search tree, heaps, AVL trees, 2-3 trees, tries, red/black tree, B-tree)

~~Pointers in C~~

Memory Structure.



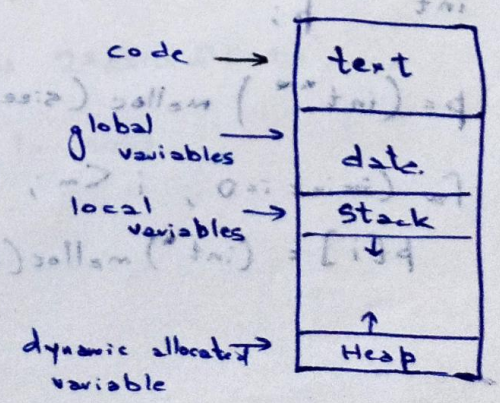
two type of memory: Byte Addresable: each byte has got one address
word addressable: each word has got one address

word = 4 byte

Parts of Executable Program

→ address is unsigned integer.
  hence size is always 4 bytes

Eg: int * p;
  float * q; → both have 4 byte

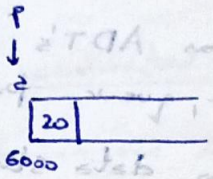


Pointers in C

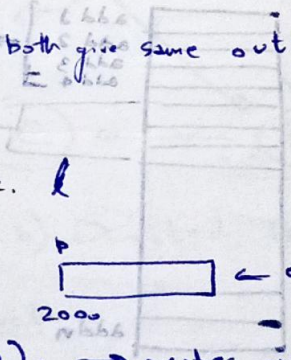
```
int a;
int *p;
p = &a;
```

`*` → value
`&` → address

```
printf("%d", *p)
```



Eg:
```
int a = 20;
int *p;
int **q;   → pointer of a pointer.
p = &a;
q = &p;
printf("%d\n", *p);
printf("%d\n", **q);
```
→ both give same output = 20



Dynamic Array using malloc and callac.

Eg:
```
int *p;
p = (int*) malloc (sizeof(int) * 10);
```
→ creates array of 10 integers (40 byte)

malloc in stdlib.h

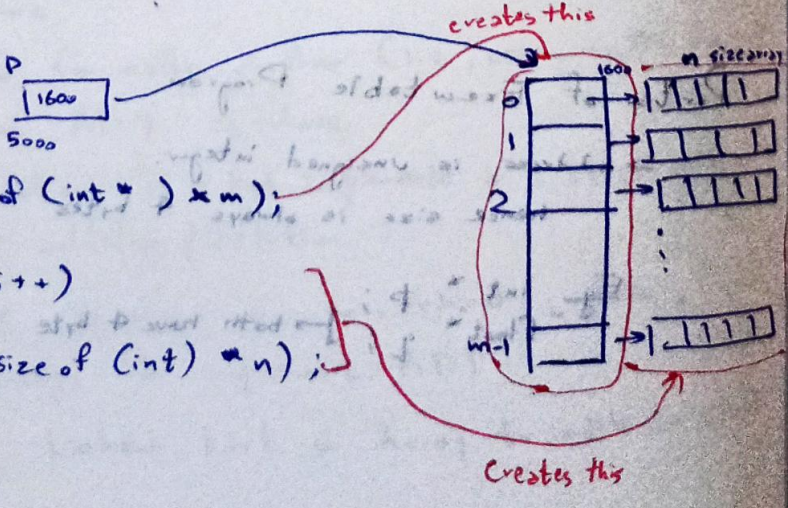
(returns void pointers)

(typecast to int*)

here p is in stack as it's local but malloc memory is from heap

2D array dynamic (m x n)

```
int **p;
p = (int**) malloc (sizeof(int*) * m);
for (int i=0; i<m; i++)
    p[i] = (int*) malloc (sizeof(int) * n);
```



Arithmetic Operation on Pointers

→ pointers can be incremented or decremented

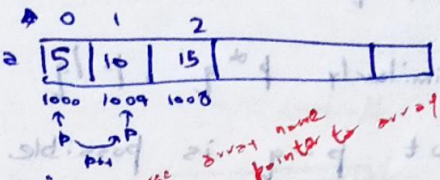
Eg int *p;

p++; → goes to next element

Eg: int a[10];

int *p;

p = &a[0]; or p = a;



p++; → p points to a[1]

p--; → p points to a[0]

Eg: struct node ← size of node = 8 byte

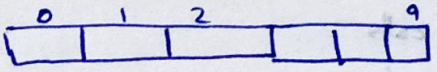
{

int a;

struct node *point; ← it's size is still 4 byte

}

struct node a[10]; →



80 byte size

struct node *p; → size 4 byte

p = a; → p points to a[0]

p++; → p points to a[1]

Eg: int *p; (let p point to 1000)

p = p+2; (p now points to 1008)

↓ not treated as 2 but as 2*(sizeof(int))

p = p-1; (p now points to 1009)

Q) Can we add two pointers?

Eg `int *p;`
`int *q;`

is p+q possible → No

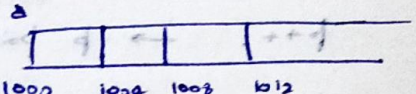
→ Similarly p*q, p/q is not possible

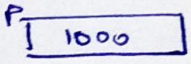
→ But p-q is possible.

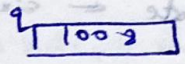
Eg: `int a[10]`

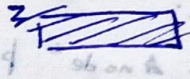
`int *p, *q;`

`p = &a[0];`
`q = &a[2];`









~~int~~

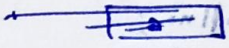
q - p = 2

Function calls

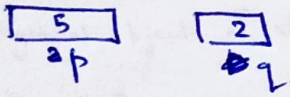
Eg
```
void swap (int a, int b)   ← call by value, change here do not affect actual parameters
{
    temp = a;
    a = b;
    b = temp;
}

void main ()
{
    a = 2;
    b = 5;
    swap (a, b);   ← This will not work
}
```

Call by Pointers or Address

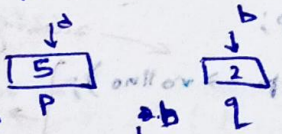
```
void swap (int* a, int *b)   ← as this is using pointers, changes have affects actual parameters
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

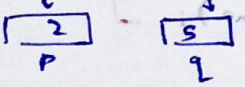


Eg in the function call swap(&p, &q)

I) 

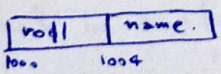
(actual parameter)

II) function used. : 

III) End result : 

Structure

```
struct Students
{
    int roll;
    char name[20];
};

void main()
{
    struct Student s1;
    struct Studen s2;
}
```



→ not necessarily continous with s1

Linked List Eg

```
struct node
{   int data;          → 4 byte.
    struct node* next; → pointer (address) → 4 byte
};
```

Eg: struct node
{
int data;
struct node next;
};

→ This is not defined yet
→ compiler can't calculate it's size
→ if we do sizeof(node) → ∞ loop will form
→ Hence Compiler will give error

Structure Pointer

Eg: struct student
{
int rollno;
char name[4];
};

void main()
{
struct student * p;
printf("%d", p → rollno);
printf("%s", p → name);
}

← accessing variables through pointer
→ No memory allocated, will give error

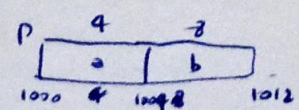
Eg: p → roll = 20
→ not possible independently as p doesn't have allocated memory
→ we have to use malloc() or calloc() to allocate memory
→ uses heap for storing memory

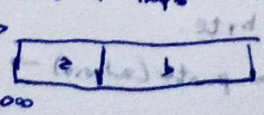
Eg: char *s;
strcpy(s, "hellow world") → will have to allocate memory or else will give segmentation fault.

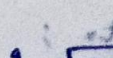
Eg: struct st1
{
int a; (4)
float b; (8)
};

struct st1 p, *q;
→ uses stack to store memory

q = &p







~~Eg typedef~~

Typedef

typedef existing_type new_name;

~~Eg~~ typedef int intvar;

→ now int a; and intvar a; are same

→ now int *a; and intvar *a; are same

~~Eg~~ struct student
{

};
st;
↑ of global object of st defined of type student

~~Eg~~ ~~do~~ typedef int* intptr;

→ now intptr is same as int*

void pointers

~~Eg~~
```
void main()
{
    void *p;
    int *p;
    float *q;

    p=q   → will give error as p and q are of different type
    v=p   → ~~this~~ this is called
}
```

[void pointer is type compatible with any other pointer
void pointers are called generic pointers

→ Void pointers can't be dereferenced

~~Eg~~
```
int *p;
int a;
void *q;

p=&a;
printf("%d",*p);   → OK
q=p                → OK
printf("%d",*q)    → not ok as q has no type.
printf("%d",*((int*)q));  → OK, as (int*)q is of type int*
```

Eg
```
void main()
{
    int a = 10;
    float b = 12.5;
    void *ptr;
    ptr = &a;
    printf("%d", *(int*)ptr);
    ptr = &b;
    printf("%d", *(float*)ptr);
}
```

Pointer to functions

declaration of pointer to function

Eg `void (*pf)(int);`

- `void` → return type
- `(int)` → parameters of a function

Eg: `void *pf(int)` → this returns `void *`, and is different

Eg:
```
void test(int a)
{
    if (a%2 == 0)
        printf("even");
    else
        printf("odd");
}

void (*pf)(int);
pf = test   → assignment of pf to test
pf(10)      → pf will be called.
```

Eg: int sum (int a, int b)
{
return (a+b);
}

void (*pf) (int)

pf = sum; → not possible as pf take one int, but sum take (int, int)

int (*abc) (int, int)

abc = sum → now OK

Eg int mult (int a, int b)
return a*b;

abc = mult; → pointer to function → Possible as type are matching

abc = sum;

(abc) (10, 5); → give 15

abc = mult

(abc) (10, 5); → give 50

Usecase example :→ sort function only applied on fundamental data type.
→ function pointer can be used here to sort user defined date type using the same sort function.

Q-Sort ← looking for more help

Eg: void qsort (void *base, size_t nitems, size_t size, int (*compare) (const void *, const void *)); → can be used to compare anything

both functions

Eg: int comp-int (const void * a, const void * b)
{
return [*((int *)a) - *((int*) b)]

qsort (

qsort (a, 100, 4, comp);

Eg int comp-struct (void* a , void* b)
return [(((struct st*) a) → rollno) - (((struct st*) b) → rollno)]

int (*compare) (const void*, const void*)

compare = comp-struct

qsort(~~a, int~~ int a[10] = { 1,3,4,5,6,7,8,9,0,2 };
int size = sizeof(a) / sizeof(a[0]);

~~qsort (a, 10, 4, compare);~~

qsort ((void*)a , size, sizeof(a[0]), compare);

~~printf~~
for(int i=0; i<10; i++)
printf("%d \n");

Arrays

→ finite set ordered set of homogeneous elements
(finite: has definite size; ordered: elements are arranged in an order; homogeneous: same data type)

→ two type:
i) Base Type → type of element (data type)
ii) Index Type → index of an array → Integer → 0 index → C++, 1 index → Matlab
→ Name

Size / length = $U_b - l_b + 1$
(U_b: upper bound, l_b: lower bound)

C / C++ : a[i]
(a: name of array, i: index position)

Three thing to ~~remember~~ specify array
i) name of array
ii) Data type
iii) index
(int a[10];)