Data Structure :→ A data structure is a systematic way of organising and accessing data in the memory of a computer.

Jese ek dukaandar ki dukaan mai bhut saara maal ho toh agar woh usse acche se organised way mai rakhta hai toh access time kaam lagega or efficiency zayada rahega but agar dhang se nhi rakhta to aap hi sochie kitna probems aa sakti hai, same rule applies to datastructures. :)

Structure :→ A set of rules that can hold the data together

eg:→ dictionary, maps

Data Structure → It's a way to store and organise data in a computer so that it can be used efficiently or (It's a way of storing data in an organised way so that it is easy to use and handle).

We study data structures in 2 ways.

① Mathematical / Logical models
or
Abstract data types (ADTs) :→ having a general or abstract view
or
they are entities which define data and operations but no operations.

② Concrete Implementation.

Some of the datastructures are →

Arrays
Linked list
Stacks
queue
Tree
Graph. etc. ...

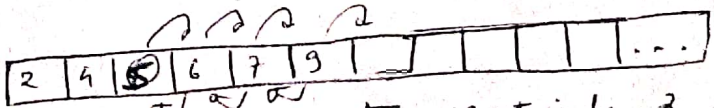
---

1) List as abstract data type

→ A list is a collection of similar objects

So Array is the implementation of this Abstract data type.

We have chosen more type



If I want to enter 10 at index 2 then I need to move all elements one towards right and if I want to remove 5 I will have to move elements one towards left.

now I say to make such list

① empty list has size 0
② insert
③ remove
④ Count the elements
⑤ Read / modify element at a position
⑥ Specify data type.

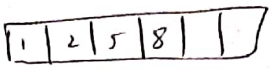
It's not possible to extend an existing array. So when array is full we will create a longer array and copy full data of previous array into it and free up before for previous array

Q) By how much should we increase the size of new array?
→ this whole process of creating a new array and copying it data is time consuming
★ So the strategy is whenever the array is full we create an array of double the size of the previous one

Cost (in time)

① Access - for Read/Write element at an index
↳ It will take constant time
As in array the elements are arranged in continuous blocks of memory starting with the base block.
↳ O(1)

② Insertion:



Time ∝ nos of elements in the array
So
O(n)

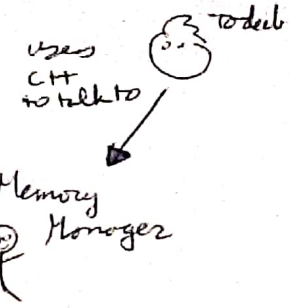
★ Note Inserting an element at the end of an empty array takes constant time i.e O(1) but if array is full we need to create a new array and copy the data of previous array into it which will take time proportional to size of array so it will be O(n).

③ Removing :→
Time ∝ n
So O(n)

Conclusion :→ So if we need to Access or Write some data at particular location then its convinient or easy but for Insertion and Removing its not feasible if array is very large. and copying and creating a new array is hectic if array is growing with time and alot of time array will be empty which will occupy space
★ So use of array dynamically is not efficient in terms of memory.

This leads us to think can we have a datastructure that will give us a datastructure to use list dynamically.
yes we have one i.e. Linked - list

Introduction to linked-list.

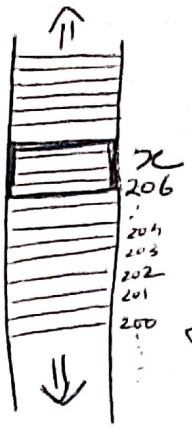


So if todeb wants to tell memory manager to store integer type he writes

`int x;`

memory manager sees it and reserves or gives 4 bytes of memory to this as integer variable takes 4 bytes of memory.

and looks for 4 byte store



consider them as single bits of memory.

1 byte = 8 bits
= 4 nibble

2^{10} bytes = 1 kb
2^{10} kb = 1 Mb
2^{10} Mb = 1 Gb
2^{10} Gb = 1 Tb

So address of block of memory is address of 1st byte of memory i.e. here address of x is 206

So now ~~other~~ todeb can store any data in this block he wants

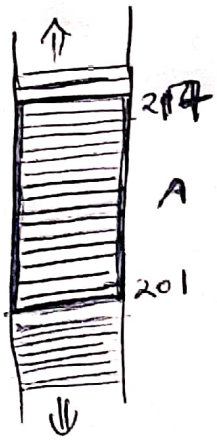
eg: x = 8

now todeb thinks to store a list of int values say he want 4 values to be stored.

so he tells the memory manager for integer Array

`int A[4];`

So memory manager books 16 byte in memory for A



So if todeb gives value to 3rd element say

`A[3] = 2;`

memory manager knows where to put this

$201 + 3 \times 4 = 213$

address of A, 3rd index, 4 bytes for int.

so Application takes constant time -

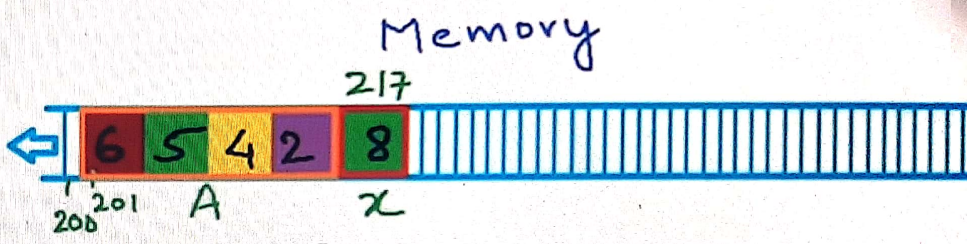
But if the block is full and todeb wants to store more things then memory manager says hey I dont have more space and the adjacent space is booked for other things 😐 See pic in Camscanner.

so todeb creates entirely new array A

Introduction to linked list

Albert

```
int x;
x = 8;
int A[4];
```

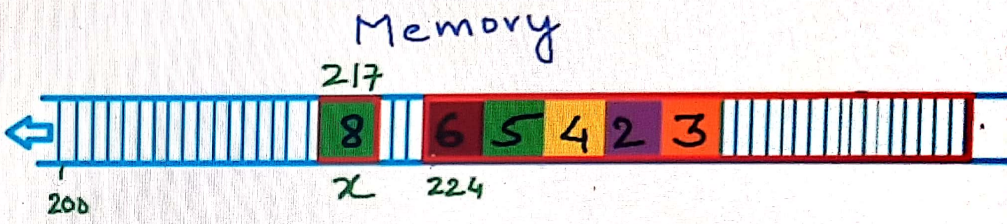


Memory Manager

Introduction to linked list

Albert

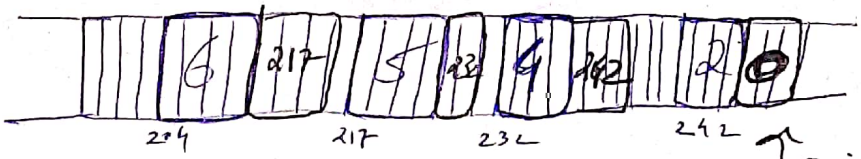
```
int x;
x = 8;
int A[4];
```



Memory Manager

So to solve this problem Todeeb can use Linked-list.

So what todeeb can do is he can store one unit or one element at a time at seperate memory locations.



So 1st he stores 6 say at address 204
now stores 5 say at address 217
Similarly he stores for numbers 4 and 2

0 is used to mark the end of the list i.e. there is no link to the next block now.

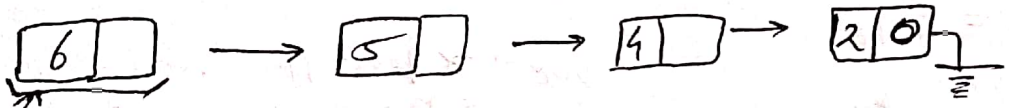
So instead of getting one continuous block of memory, todeeb gets this disjoint non-continuous blocks of memory so he needs to store or give some info to link these blocks and tell which one is the starting point or 1st one.

So we can have two parts stored in memory i.e. 1st) the value or data
2nd) the address of other block.

So Todeeb will need to store or tell memory manager to store 2 values

① the value of the element which hold it value. (variable) 4 bytes
② the pointer variable which holds the address of next block in the list. 4 bytes
⇒ 8 bytes.

This is the linked list data structure



① So there is no wastage of memory although we are taking some extra byte to store address but that's reasonable.

So we always keep address of the head node with us which in turn give us access of the complete list.

So the only way to traverse the linked list is from the start i.e. asking the address of the 2nd node from 1st, 3rd from 2nd and so on,

So can add a new element, delete element, access or read data.

Note.

Unlike arrays we can't access any of the element in constant time because linked-list traverse from start to the element searched

So Time ∝ no of elements or T∝n or O(n)

Delete → O(n)

Insert → O(n) → So we don't shifts here like arrays.

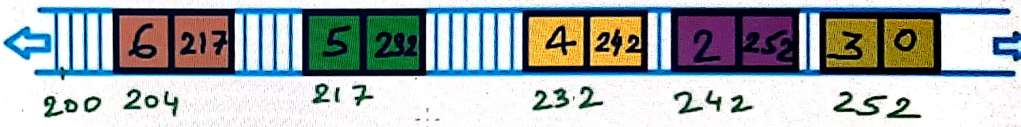
Introduction to linked list

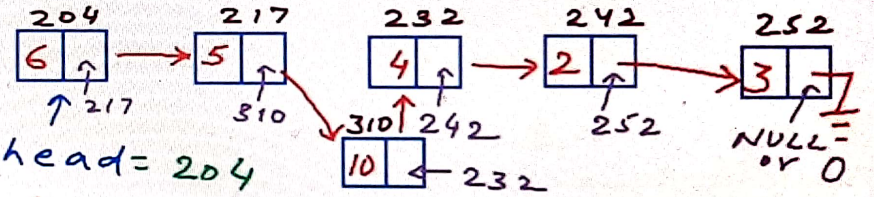
Albert

6, 5, 4, 2, 3

```
Struct Node
{  int data; //4 bytes
   Node* next; //4 bytes
}
```

Memory





head = 204

Access to elements: $T \propto n$, $O(n)$

Insertion: $O(n)$

Pointers

Pointers points to a memory location

So a Pointer is used to access the information at a particular memory location.

Declaration of a pointer variable

datatype* Identifier;

eg: int* a; / int *a;

↑ ↗ Both way we can write

Two types of pointers

1) Typed pointers: It points to a specific type of data

for eg:

int* → It points to only integer data

double* → It points only to double data

Class Emp* → It points only to Employee data

So Int can't point to double data

double can't point to int and so on

2) Untyped pointers:- It can point to any data [Generic pointer]

eg: void* ——→ It can points to any type of data.

& *

↑ address operator ↑ pointer variable

Address operator: returns the address of a particular variable

Pointer operator:- It returns the value which is inside the specified address.

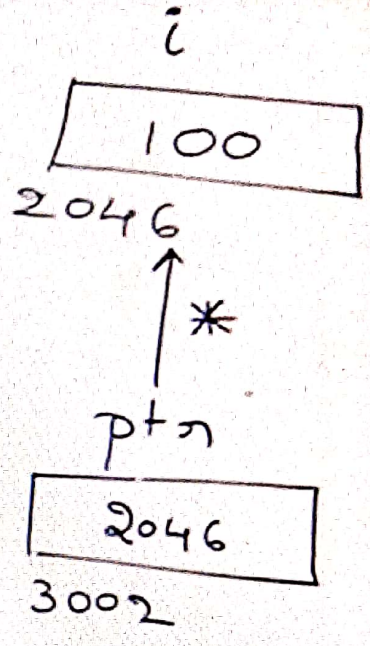
eg: see in next pic Consumer.

So basically pointer is mukhbir → It depends on him what info to give, he can give correct info also or can manipulate info too.

Isko pata batana padega jiski mukhbiri karni hoo

utorial

```
void main()
{
    int  i = 100;
    int* ptr;
    ptr = &i;
    printf("%d", i);      ---> 100
    printf("%d", ptr);    ---> 2046
    printf("%d", &i);     ---> 2046
    printf("%d", &ptr);   ---> 3002
    printf("%d", *ptr);   ---> 100
    printf("%d", *(&i));  ---> 100
}
```



int a = 32 | 101 ptr = 101

char ch = 'A' → 'a'

char cho = 'A' &cho

Q:

```
#include <iostream>
using namespace std;
int main()
{ int a = 32, *ptr = &a;
  char ch = 'A', &cho = ch;
  cho += a;
  *ptr += ch;
  cout << a << ", " << ch << endl;
  return 0;
}
```

This is a Reference - So whatever happens to reference happens to ch also or the thing it is refering to

Outputs
cho = a
*ptr = 32 + 97 = 129

129, a

cho = cho + a
↑ = 'A' + 32
char type = 65 + 32
= 97
cho = 'a'
so ch = 'a' as

ASCII A-Z 65-90 a-z 97-122

```
{ int x = 5;
  int *ptr = &x;
  cout << "Address of " << x << " is " << ptr;
} return 0;
```

→ Output)

Address of 5 is 0x

dalal kee pass abhi sirf address hai usko abhi humne activate nhi kiya.

but if we do this

cout << "The value at Address " << ptr << " is " << *ptr

```
{ const int i = 20;
  const int* const ptr = &i;
  (*ptr)++;
  int j = 15;
  ptr = &j;
  cout << *ptr;
  return 0;
}
```

→ declaring const before means abb hum i kii value poore program mai khi change nhi kr sakte.

→ yee i ko change karna chahta hai but wo kr nhi sakta As i.e 20 → 21 but compilation error will occur.

→ yee statement sahi hai kii abb j ko point kare ptr but last yee kr nhi pata as ptr is const so again compilation error

Harddisk is slow as compared to Ram

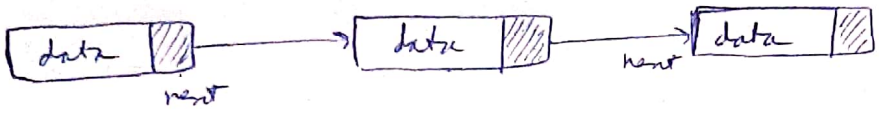
Q structure? Q) Why data structure →

It's a smart working i.e dealing with Ram and then moving it to harddisk because Ram is not very large space

By default malloc returns value in char
So it's a must to convert to the desired datatype and must use <stdlib.h>
eg:

int *p = (int*) malloc (sizeof (int));

#



Creating a linked list
Traverse a linked list

Array VS linked list

→ So there is no such thing that which data structure is better, It all depends on what frequent operation you want to perform on the data structure, based on that we choose the data structure.

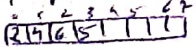
Array

1) Cost of accessing an element — Constant time

or O(1) 200 + i×4

2) Memory Usage — As it is of fixed size memory expansion is difficult

eg:

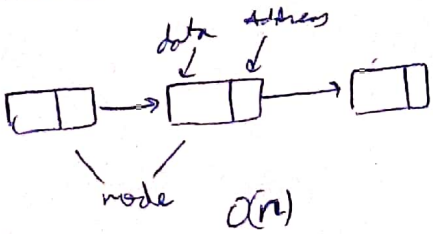


7×4 = 28 bytes to store 4 elements but

and 4×4 = 16 bytes to store 4 elements but memory left is wasted i.e 12 bytes wasted

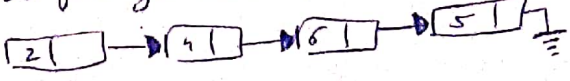
So 7×16 = 112 bytes

linked list



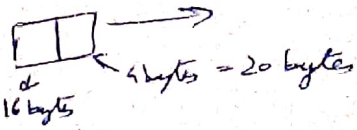
O(n)

No unused memory
Extra memory for pointer variables
So for large data linked list is preferable



8 bytes do 8×4 = 32 bytes to store 4 elements

If data is large then eg:



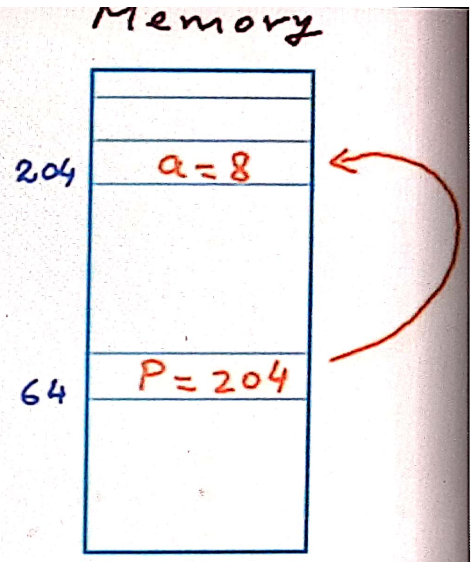
4 × 20 = 80 bytes.

Pointers - variables that store address of another variable

P → address

*P → value at address

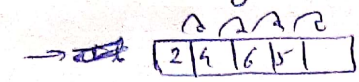
```
int a;
int *p;
⇒ P = &a;
  a = 5;
⇒ Print P     // 204
  Print &a    // 204
  Print &P    // 64
⇒ print *p    // 5   ⇒ dereferencing
⇒ *P = 8
  Print a     // 8
```

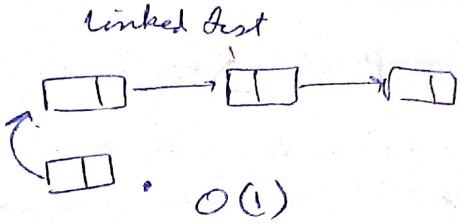


Array

Linked List

Insertion) and Deletion.





- at beginning
  - O(n) → We need to shift
  - O(1)

- at end
  - O(1) → If array is not full
  - O(n) → We need to create new array and copy its elements there
  - O(n) → Traversing the whole list and creating a new node and editing the links.

- at middle
  - O(n) → We need to shift
  - O(n)

Ease to use) ✓

✗ → It is more prone to error
eg: segmentation fault
memory leak. etc

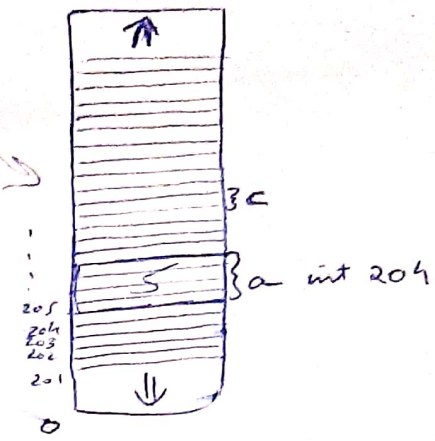
Linked list Implementation in C/C++

Pointers

When we talk of memory, we refer to RAM.

Memory (RAM)

```
int a;
char c;
a=5;
```



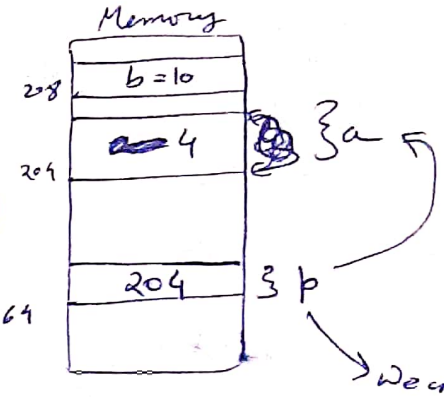
Can we operate on address of these variables so answer is yes using pointers.

Pointer variable → stores address of another variable.

We dont know address format so we store using & sign

eg: address of a

p = 204 ← but 204 is our address but in reality address is a hexadecimal nos so its difficult to write hence



as a is int p is also int so they both take 4 bytes of memory each

We can also modify p

eg p = 208 now it point to b.

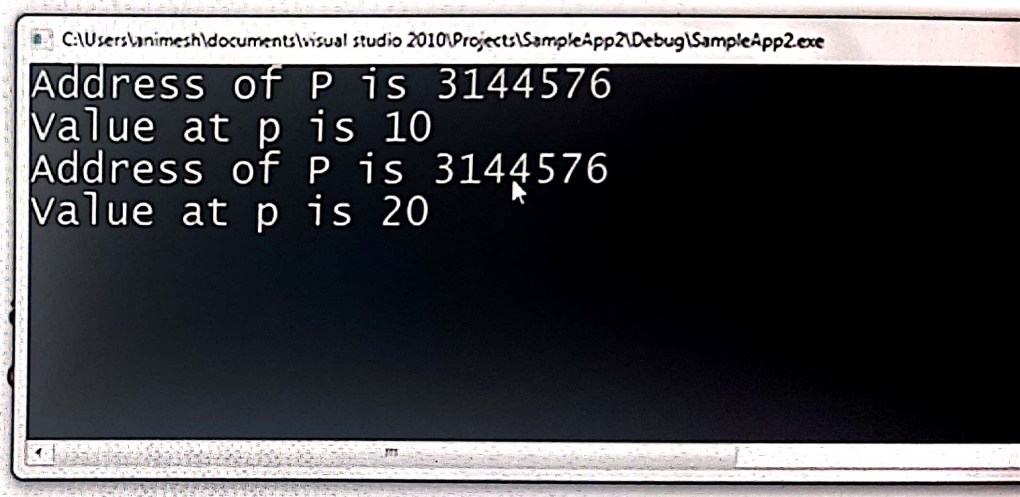
int p = &a

to make p a pointer all we do is put a * astrik sign before p or after int

eg: int *p = &a

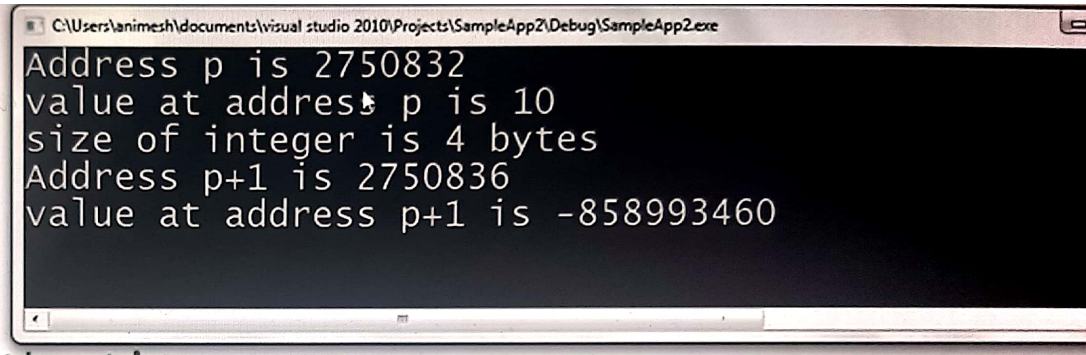
Simply p means pointer variable which holds address and on cout will just give the address stored in it but as soon as we put on * sign before p it make it to see the value inside that address and gives access to that value also.

```c
#include<stdio.h>
int main()
{
    int a;  int *p;
    a = 10;
    p = &a; // &a =
    printf("Address
    printf("Value at
    int b = 20;
    *p = b; // Will the address in p change to point b
    printf("Address of P is %d\n",p);
    printf("Value at p is %d\n",*p);
}
```

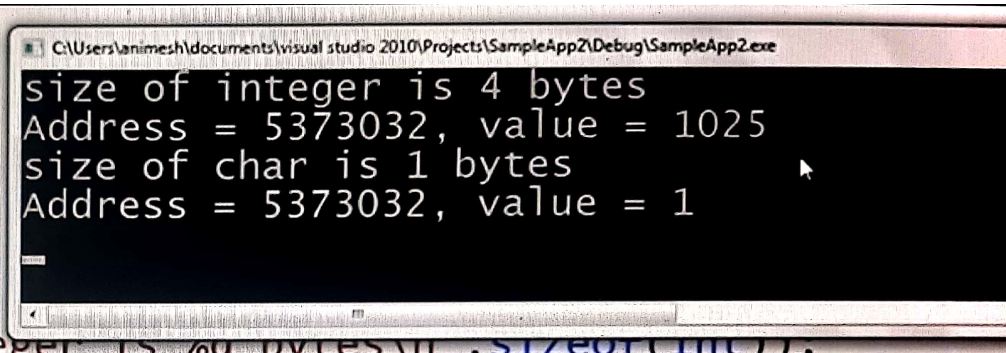


```c
#include<stdio.h>
int main()
{
    int a = 10;
    int *p;
    p = &a;
    // Pointer arithmetic
    printf("Address p is %d\n",p); // p is 2002
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address p+1 is %d\n",p+1); // p+1 is 2006
}
```

```c
#include<stdio.h>
int main()
{
    int a = 10;
    int *p;
    p = &a;
    // Pointer arithmetic
    printf("Address p is %d\n",p);
    printf("value at address p is %d\n",*p);
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address p+1 is %d\n",p+1);
    printf("value at address p+1 is %d\n",*(p+1));
}
```



```c
#include<stdio.h>
int main()
{
    int a = 1025;
    int *p;
    p = &a;
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address = %d, value = %d\n",p,*p);
    char *p0;
    p0 = (char*)p; // typecasting
    printf("size of char is %d bytes\n",sizeof(char));
    printf("Address = %d, value = %d\n",p0,*p0);
}
```



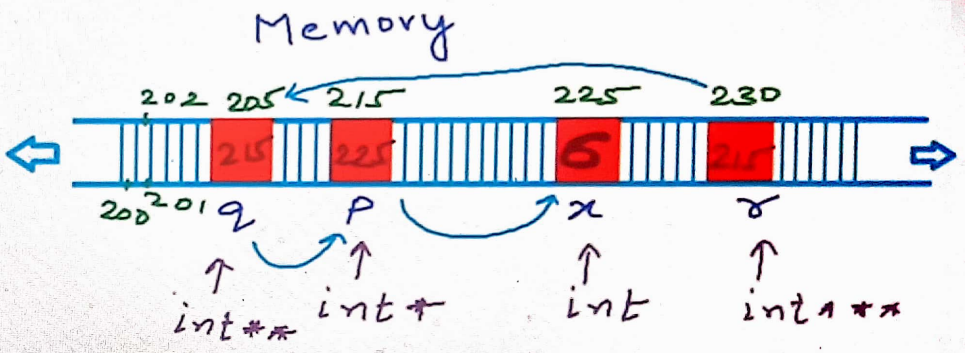
```c
#include<stdio.h>
int main()
{
    int a = 1025;
    int *p;
    p = &a;
    printf("size of int
    printf("Address = %d, value = %d\n",p,*p);
    printf("Address = %d, value = %d\n",p+1,*(p+1));
    char *p0;
    p0 = (char*)p; // typecasting
    printf("size of char is %d bytes\n",sizeof(char));
    printf("Address = %d, value = %d\n",p0,*p0);
    printf("Address = %d, value = %d\n",p0+1,*(p0+1));
    // 1025 = 00000000 00000000 00000100 00000001
```

```
C:\Users\animesh\documents\visual studio 2010\Projects\SampleApp2\Debug\SampleApp2.exe
size of integer is 4 bytes
Address = 4456036, value = 1025
Address = 4456040, value = -858993460
size of char is 1 bytes
Address = 4456036, value = 1
Address = 4456037, value = 4
```

```c
#include<stdio.h>
int main()
{
    int a = 1025;
    int *p;
    p = &a;
    printf("size of integer is %d bytes\n",sizeof(int));
    printf("Address = %d, value = %d\n",p,*p);
    // Void pointer - Genric pointer
    void *p0;
    p0 = p;
    printf("Address = %d",p0);
}
```

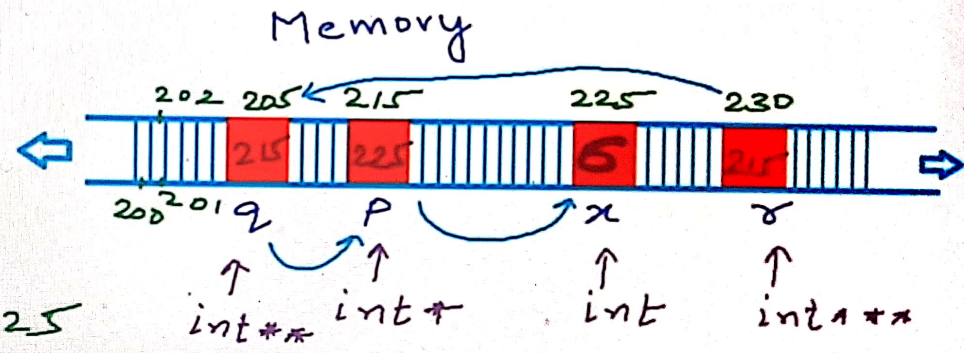
```c
#include<stdio.h>
int main()
{
    int x = 5;
    int* p = &x;
    *p = 6;
    int** q = &p;
    int*** r = &q;
```

Pointer to pointer



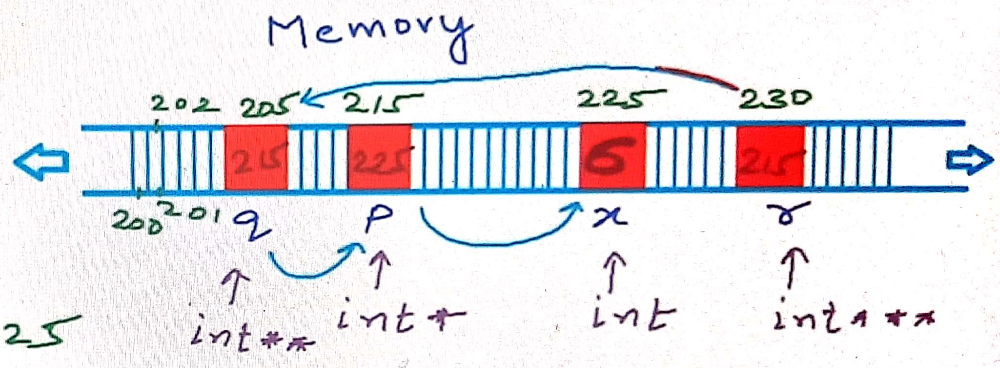
Pointer to pointer

```c
#include<stdio.h>
int main()
{
    int x = 5;
    int* p = &x;
    *p = 6;
    int** q = &p;
    int*** r = &q;
    printf("%d\n",*p); // 6
    printf("%d\n",*q); // 225
    printf("%d\n",*(*q)); // 6
}
```

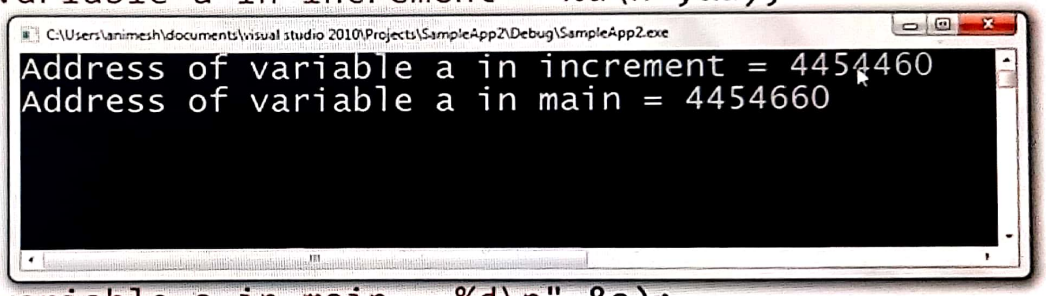


Pointer to pointer

```
nclude<stdio.h>
t main()

  int x = 5;
  int* p = &x;
  *p = 6;
  int** q = &p;
  int*** r = &q;
  printf("%d\n",*p); // 6
  printf("%d\n",*q); // 225
  printf("%d\n",*(*q)); // 6
  printf("%d\n", *(*r)) ; // 225
  printf("%d\n",*(*(*r))); // 6
}
```

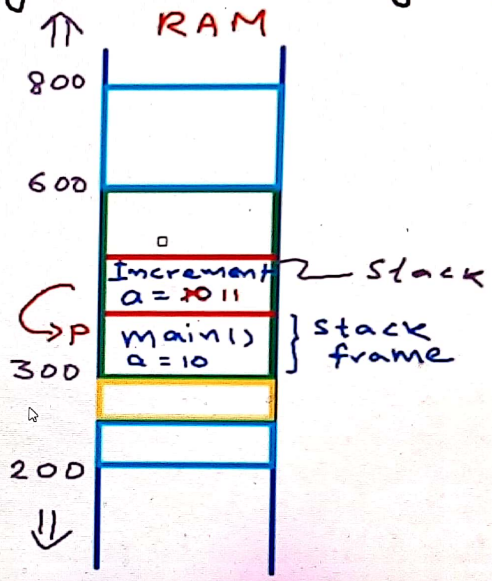


```c
#include<stdio.h>
void Increment(int a)
{
    a = a+1;
    printf("Address of variable a in increment = %d\n",&a);
}
int main()
{
    int a;
    a = 10;
    Increment(a);
    printf("Address of variable a in main = %d\n",&a);
    //printf("a = %d",a);
}
```



Pointers as function arguments - Call by reference

```c
#include<stdio.h>
void Increment(int a)
{
    a = a+1;
}
int main()
{
    int a;
    a = 10;
    Increment(a);
    printf("a = %d",a);
}
```



Pointers as function arguments - Call by reference

```
#include<stdio.h>
void Increment(int a)
{
    a = a+1;
}
int main()
{
    int a;
    a = 10;
    Increment(a);
    printf("a = %d",a);
}
```

x

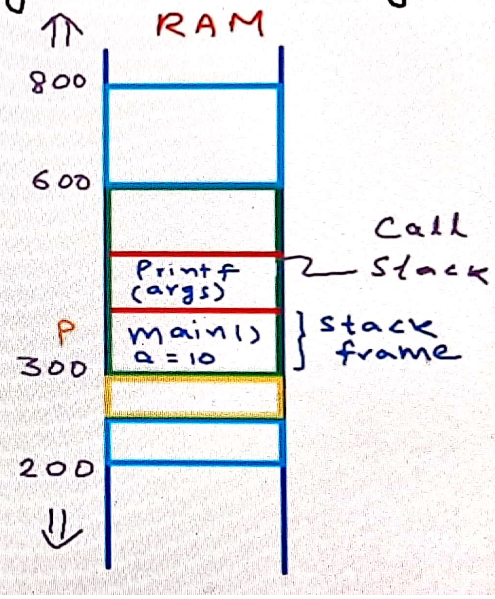
called function

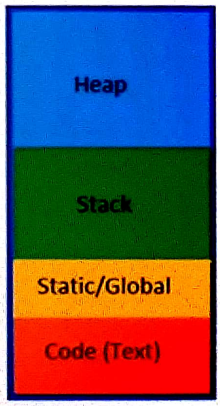
formal argument

calling function

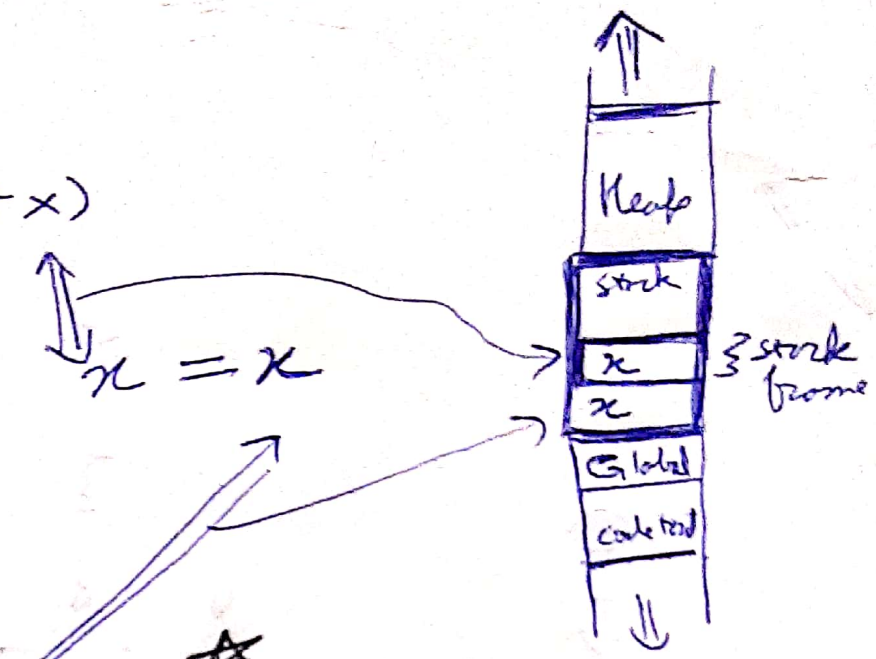
Actual argument

a → a
a → x





```
#include <stdio.h>
void Increment (int x)
{
   x = x+1
}
int main()
{
  int x
   x = 10
   Increment (x);
   cout << "Value of x is :=" << x;
}
```



value of this x is mapped to the above x

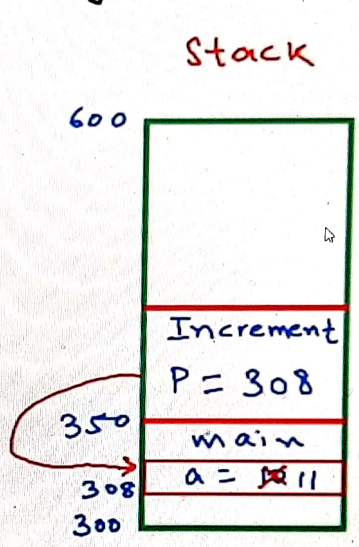
Output is 10 not 11 why

So when we make such a function call where we basically map one variable to another. i.e. the value in one variable gets copied to another variable then such a function call is called as

Call by value.

Pointers as function arguments - Call by reference

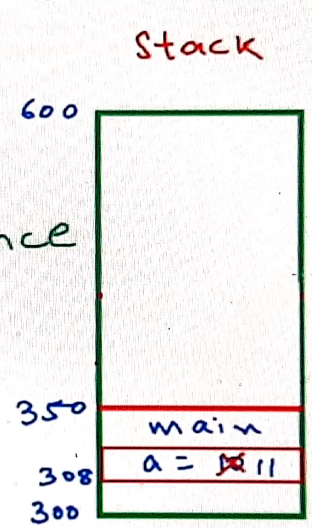
```c
#include<stdio.h>
void Increment(int *p)
{
    *p = (*p) + 1;
}
int main()
{
    int a;
    a = 10;
    Increment(&a);
    printf("a = %d",a);
}
```



Pointers as function arguments - Call by reference

```c
#include<stdio.h>
void Increment(int *p)
{
    *p = (*p) + 1;
}
int main()
{
    int a;
    a = 10;
    Increment(&a);
    printf("a = %d",a);
}
```

Call by reference



Call by Reference :→ When use pointers as function arguments we we are actually using Call By Reference.

Or

Such a function call in which instead of passing the value of variable, we pass the address of the variable so that we have a reference to the variable and we can dereference it and perform some operations is called Call by Reference.

So Call by Reference save us a alot of memory because instead of creating a copy of large and complex data type, if we just use a reference to it and using reference will also cost us some memory but that is feasible then we are saved from creating a copy of complex data type.

Pointers and Arrays

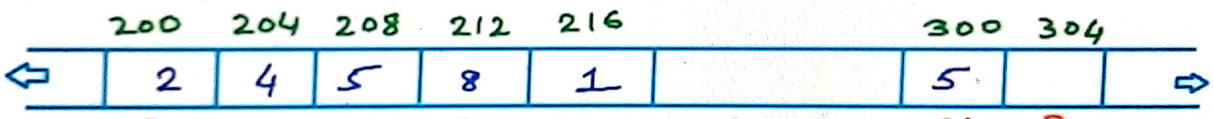
int A[5]

A[0]
A[1]
A[2]
A[3]
A[4]

int → 4 bytes

A → 5 × 4 bytes
= 20 bytes

int A[5]
int *p
P = &A[0]
Print P // 200
Print *p // 2



Print P+2 //208
Print *(P+2) // 5

Pointers and Arrays

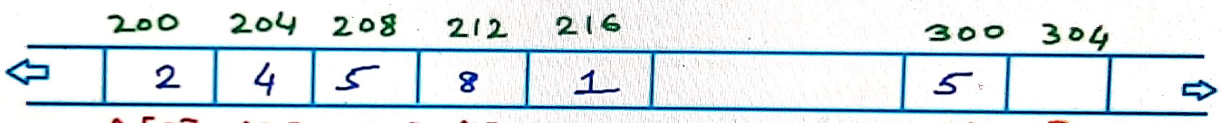
int A[5]

A[0]
A[1]
A[2]
A[3]
A[4]

int → 4 bytes

A → 5 × 4 bytes
= 20 bytes

A gives us base address



Element at index i –

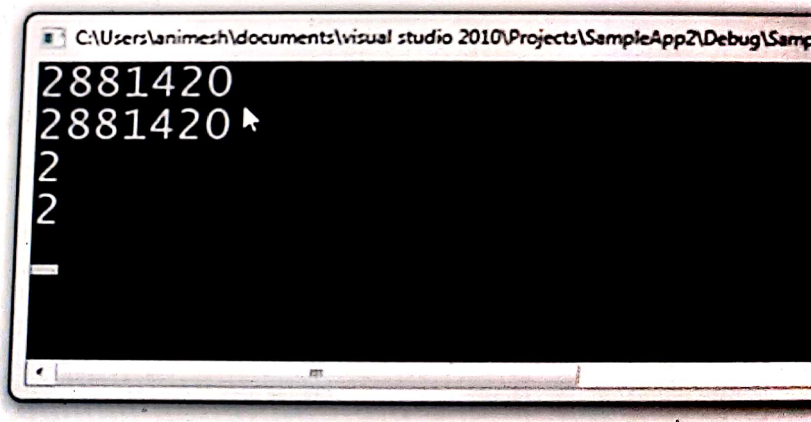
Address – &A[i] or (A + i)

Value – A[i] or *(A+i)

int A[5]
int *p
P = A
Print A // 200
Print *A // 2

Print A+1 // 204
Print *(A+1) // 4

```c
// Pointers and Arrays
#include<stdio.h>
int main()
{

    int A[] ={2,4,5,8,1};
    printf("%d\n",A);
    printf("%d\n",&A[0]);
    printf("%d\n",A[0]);
    printf("%d\n",*A);

}
```



```c
#include<stdio.h>
int SumOfElements(int* A, int size)// "int* A" or "int A[]"  ..it's the same..
{
    int i, sum = 0;
    printf("SOE - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
    for(i = 0;i< size;i++)
    {
        sum+= A[i]; // A[i] is *(A+i)
    }
    return sum;
}
int main()
{
    int A[]= {1,2,3,4,5};
    int size = sizeof(A)/sizeof(A[0]);
    int total = SumOfElements(A,size); // A can be used for &A[0]
    printf("Sum of elements = %d\n",total);
    printf("Main - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
}
```

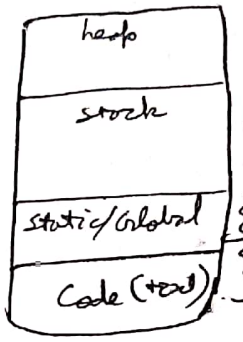
Arrays as function arguments.

For execution of function calls we use Stack part of memory.

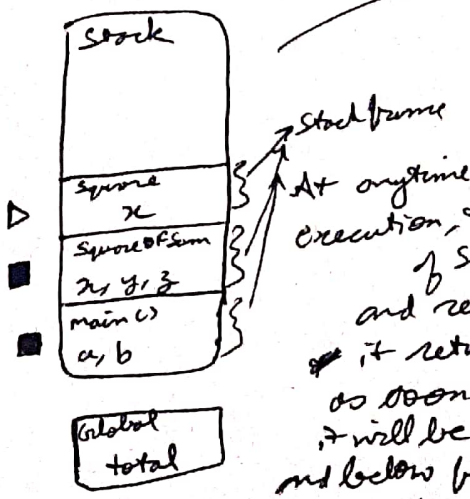
Arrays are always passed as reference.

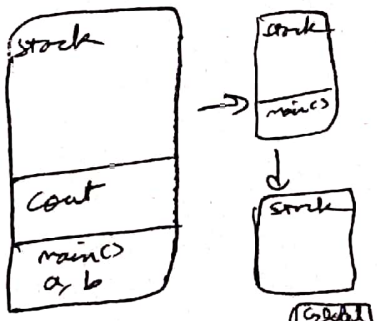
Pointers and dynamic memory.

```
#include <stdio.h>
int total;
int Square (int x)
{
    return x*x;
}
int SquareOfSum (int x, int y)
{
    int z = Square(x,y);
    return z;
}
int main()
{
    int a=4, b=8
    total = SquareOfSum(a,b);
    cout << "The square of Sum of two nos is = " << total;
}
```



So what happens is :-

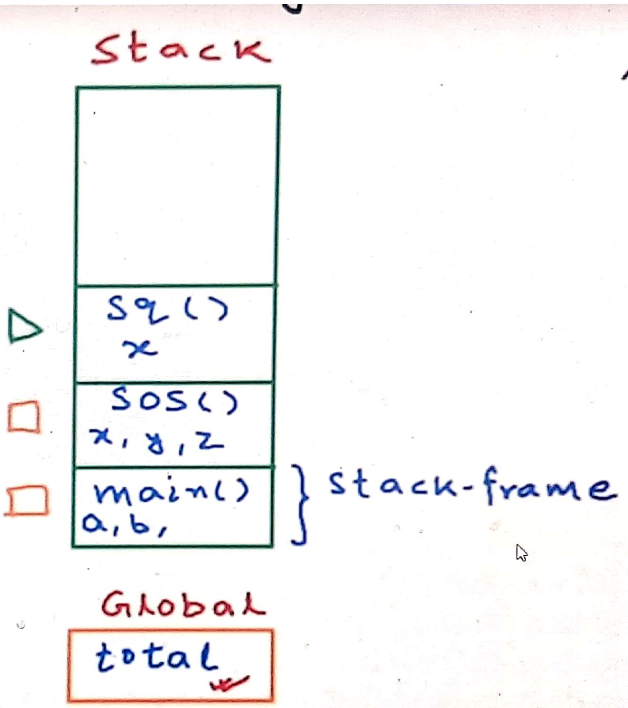


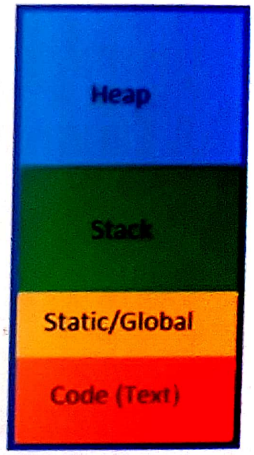


At anytime during the program execution, the function at top level stack frame is executed and rest are paused until it returns some value as soon as it returns something it will be cleared from Stack memory and below function will resume and as soon as it returns something some process

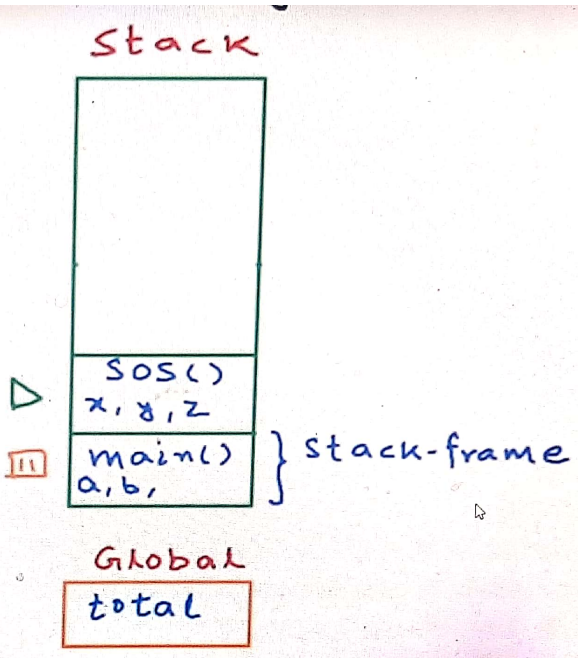
now main will call cout and cout gets executed then it gets cleared now main gets cleared

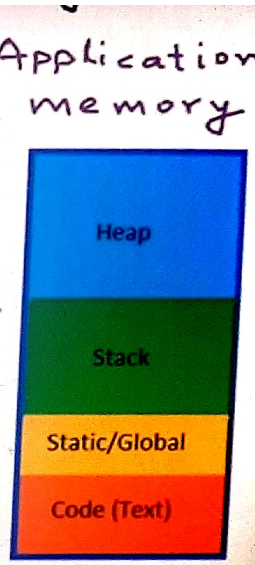
```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



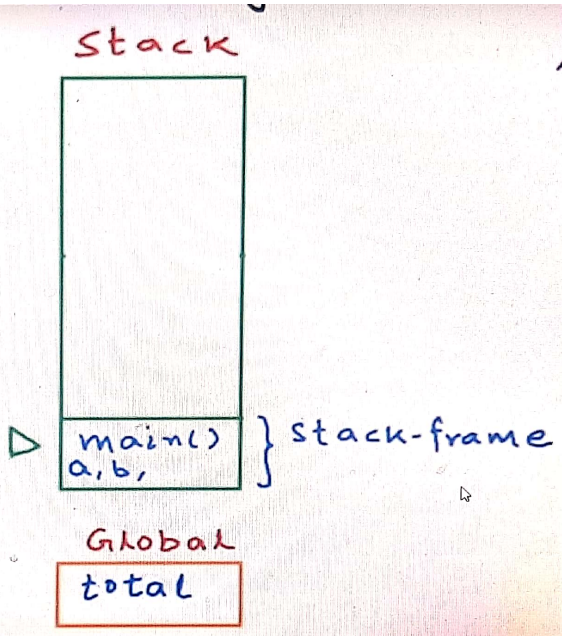


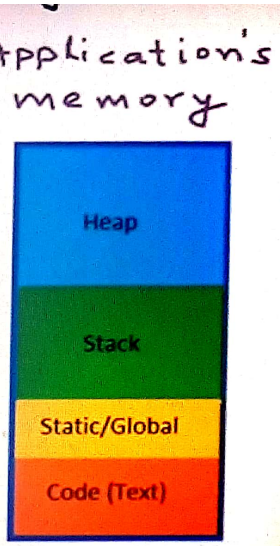
```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



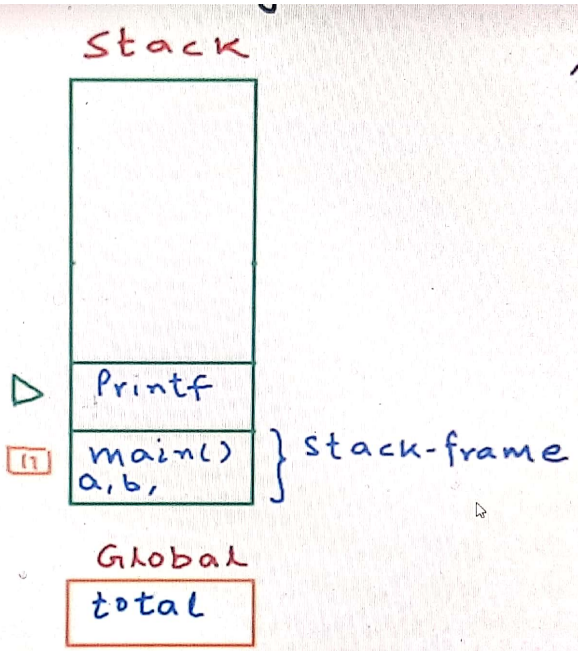


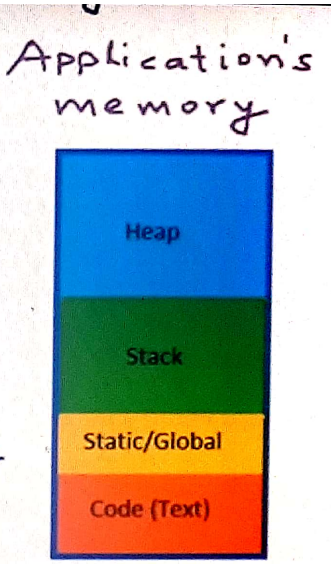
```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



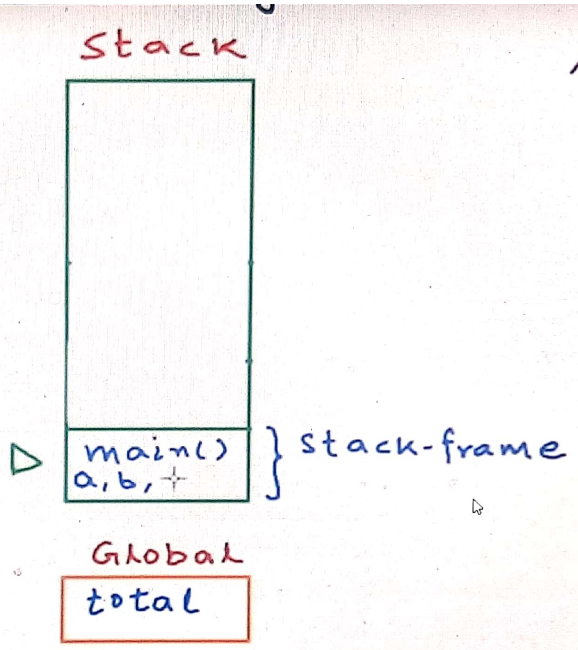


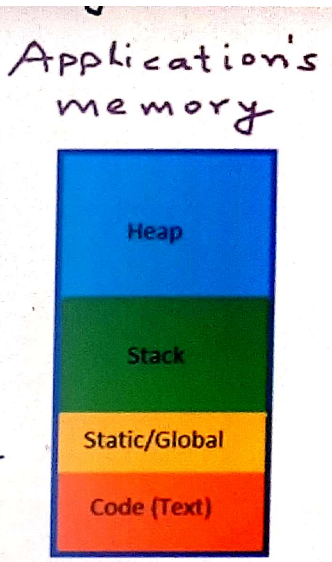
```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```



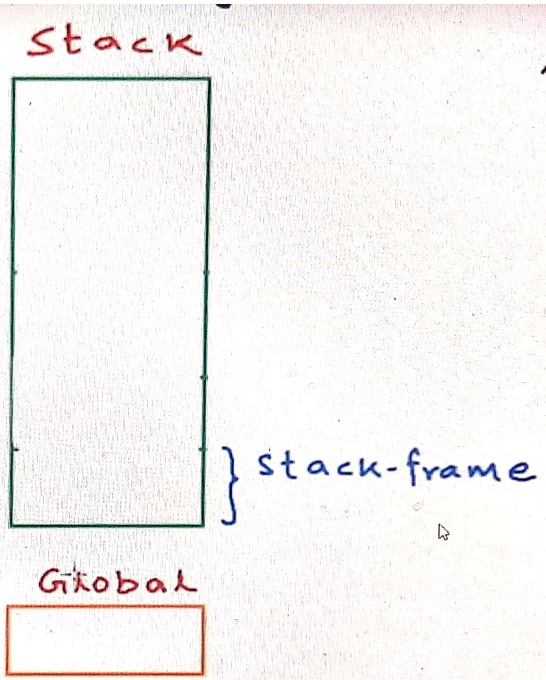


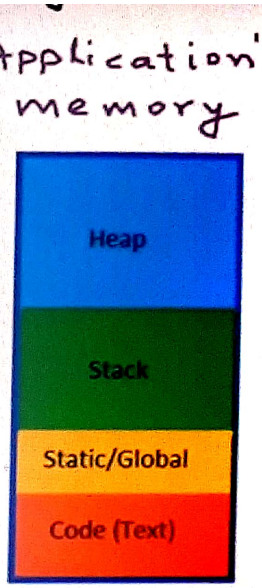
```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```





```c
#include<stdio.h>
int total;
int Square(int x)
{
    return x*x;  // x^2
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z;  // (x+y)^2
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}
```

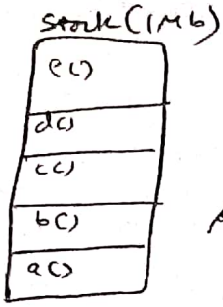




Note:-

When our program starts, our operating system allocates some amount of reserved space to stack. say eg: 1Mb but the actual allocation of local variables / stack frame happens during runtime and if our call stack grows beyond the reserved memory of stack

eg:



A method a() calls b() calls c() ... until stack is full. and we exhaust the whole space reserved for stack and this situation is called Stack Overflow and our program crashes.

Limitation of Using Stack)

1) So Application can't request more memory for stack during run-time so at runtime we can't dynamically adjust the stack memory

2) When a function is called, it is pushed on top of a stack frame and when it finish, it gets poped or removed from the stack so it is not possible to manipulate the scope of a variable

3) If we need to declare a large data type like array as local variable then we need to know its size before runtime i.e at Compile time

So for all these ~~problems~~ requirements i.e. allocating large chunks of memory or keeping variable in the memory till the time we want we have heap. This is a abstract view of heap

Unlike stack Application's heap is not fixed. Its size could vary during the lifetime of the application and there is no set rule for allocation and deallocation of memory. The programmer can have total control. A heap can grow until you drain all your systems memory. So, its a dangerous situation and we need to be careful while using heap.

We also call heap as Free pool of memory / Free Store of memory

Heap is also called as Dynamic Memory

Because its size changes dynamically at Run-time
and its usage is called Dynamic Memory Allocation.

[Heap is also a datastructure name
but this heap (memory one) is different from that heap (datastructure)]

[Stack is also a dataStructure but the stack segment of the memory is actually an implementation of stack datastructure]
but heap is not an implementation of heap datastructure.

To use Dynamic Memory in C we need to know 4 functions.

1) malloc
2) calloc
3) realloc
4) free

} They can also be used in C++ as C++ is just an extention of C

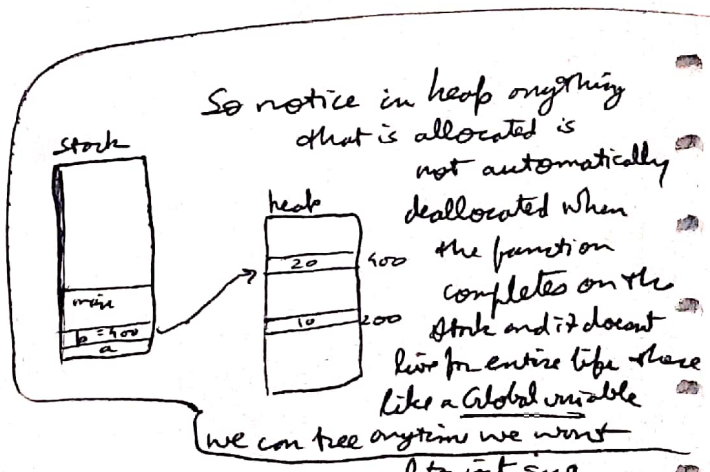
For C++

new
delete

---

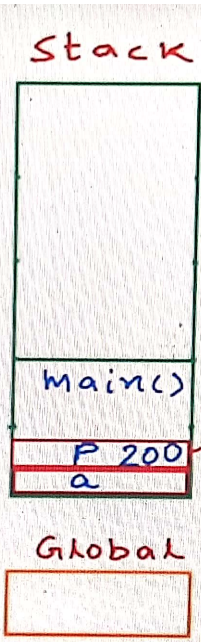
eg:

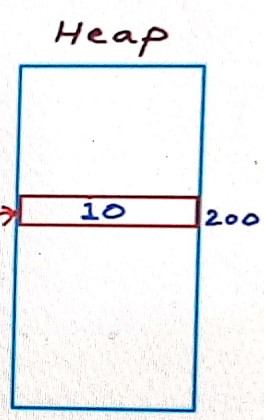
```
#include <studio.h>
#include <stdlib.h>
int main()
{ int a;      // goes on stack part of memory
  int *p;     // goes on stack part
  p = (int *) malloc (sizeof(int)) ;  // they give me size of memory equal to int size
                                      malloc return type is void that's why typecasting
  *p = 10;    // now value given in heap.
free(p); → p= (int *) malloc (sizeof(int)); // again new memory will be allocated in heap and now address in p changes
  *p=20;  # now value given in this new block is 20 - but the old block is still there
            in heap, so we need to clear it. so we need to call free before
            creating new block.
```

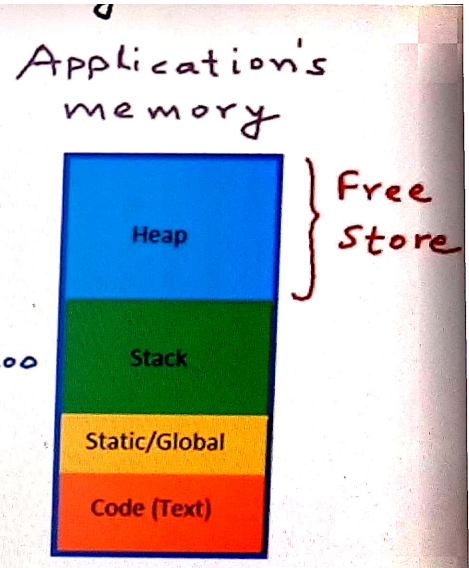


So notice in heap anything that is allocated is not automatically deallocated when the function completes on the stack and it doesn't live for entire life there like a Global variable
we can free anytime we want

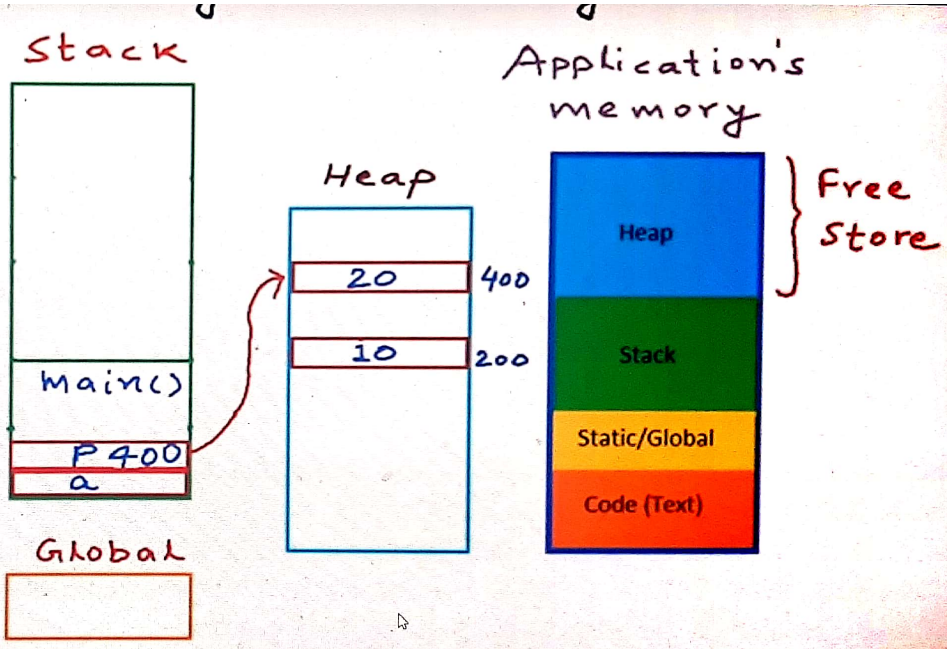
```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
  int a; // goes on stack
  int *p;
  p = (int*)malloc(sizeof(int));
  *p = 10;
```







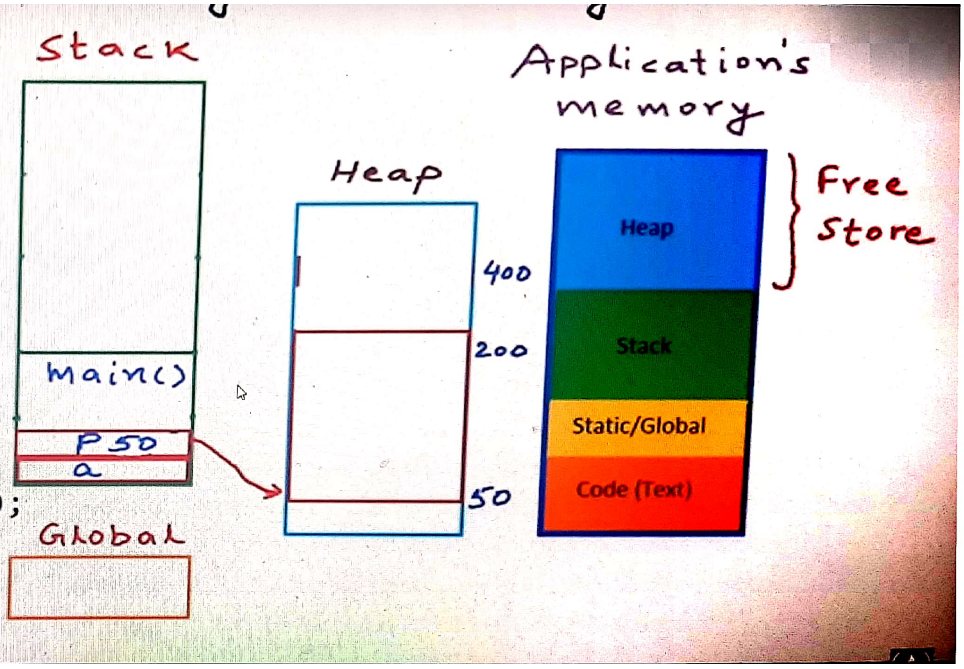
```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int a; // goes on stack
   int *p;
   p = (int*)malloc(sizeof(int));
   *p = 10;
   p = (int*)malloc(sizeof(int));
   *p = 20;
```



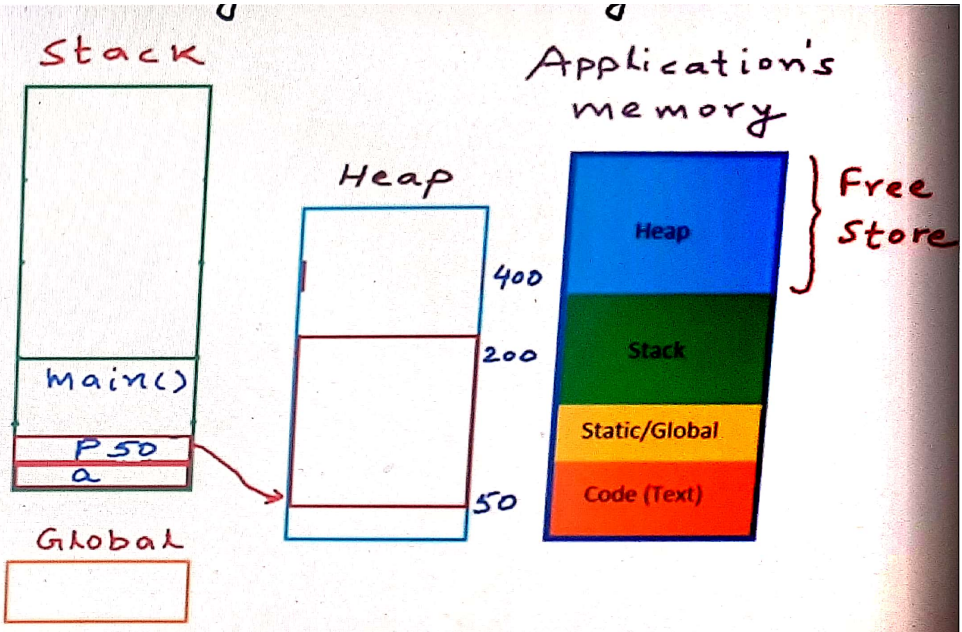
```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int a; // goes on stack
   int *p;
   p = (int*)malloc(sizeof(int));
  *p = 10;
   free(p);
 p = (int*)malloc(20*sizeof(int));
}
```

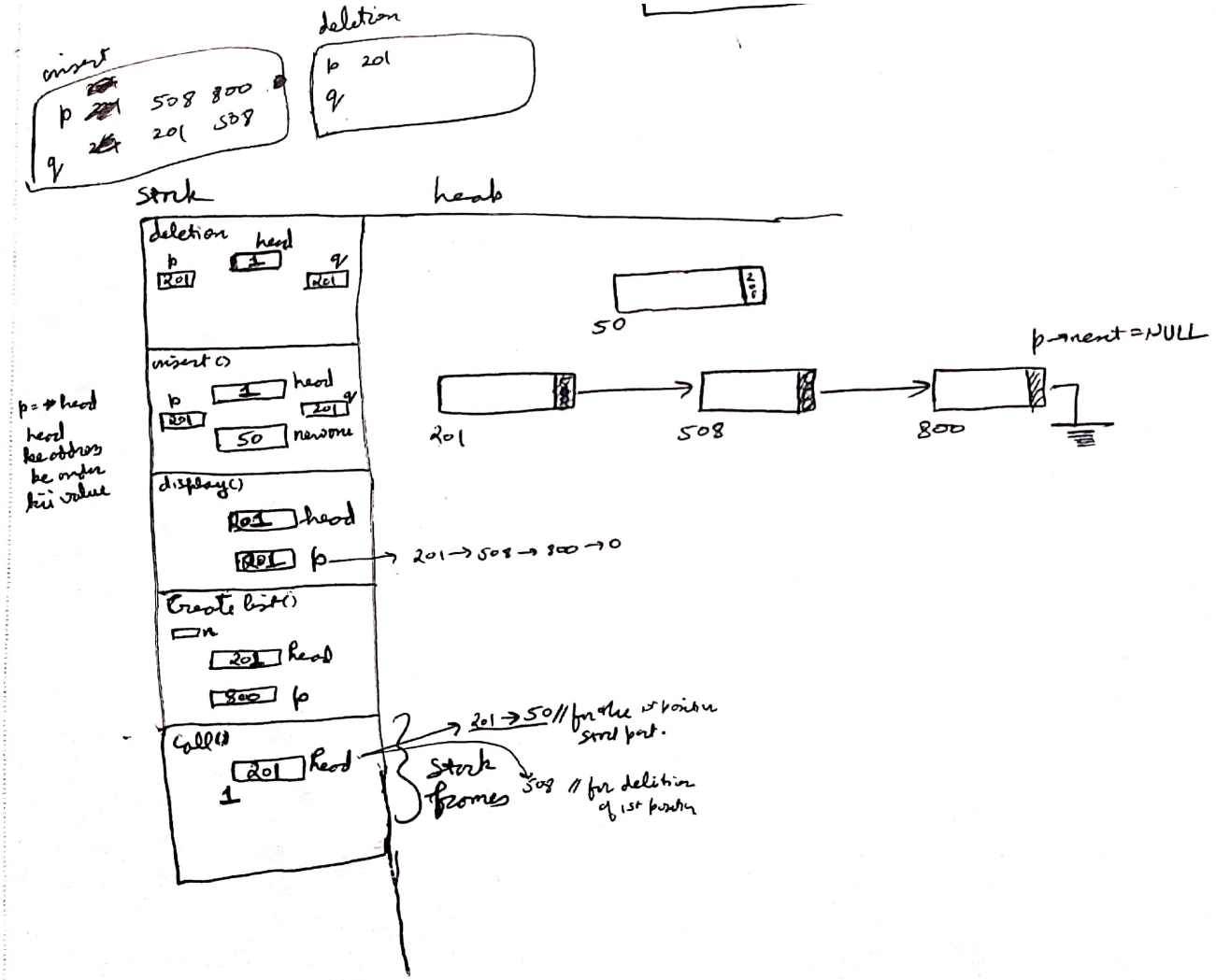
P[0], P[1], P[2]

*p *(p+1)



```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a; // goes on stack
    int *p;
    p = new int;
    *p = 10;
    delete p;
    p = new int[20];
    delete[] p;
}
```





head** → head value → 201 head value

head=1 head* → 201 head** → chg, chg vale etc.

Introduction to Stacks

Stack ADT :⇒ (Abstract datatype)

The items of stack are inserted or removed from the top

This also called → Last-In-First-Out (LIFO).

→ A list with the restriction that insertion and deletion can be performed only from one end called the top.

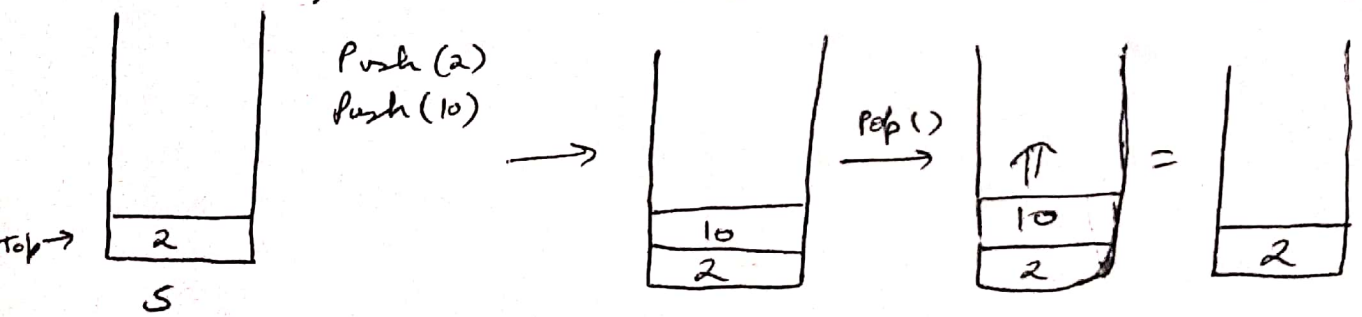
Operations

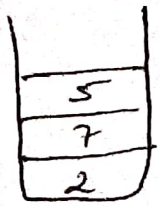
1) Push(x) → Insertion is called push in Stack.
2) Pop() → It is removing the most recent item from the Stack.
3) Top() → It simply returns the element at top of the Stack.
4) IsEmpty() → It checks whether the Stack is empty or not.
   ↳ returns boolean value

These all operations can be performed in constant time. or O(1).

* We can push or pop only one element at a time.

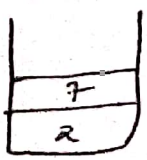
* The stack is empty right now, so we can't pop anything.





Push(7)
Push(5)
Top() → output → 5
IsEmpty() → false

pop() →



pop() →

pop() →

* We can pop() until the stack is empty.

Applications

- Function Calls / Recursion.
- Undo Operation.
- Balanced Paranthesis.
  {(1)}

Implementation of Stacks

So if we can add this property i.e. inserting and deletion from one end in a list then we can achieve Stacks datastructures.

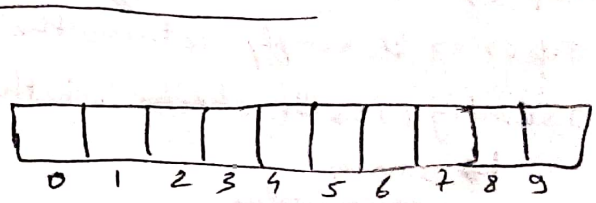
Lists (i.e Arrays and Linked List)

1) Array Implementation.

eg: If I wanna create a Stack of integers in Array

* In an empty stack, top is assumed to be -1

Pseudo Code

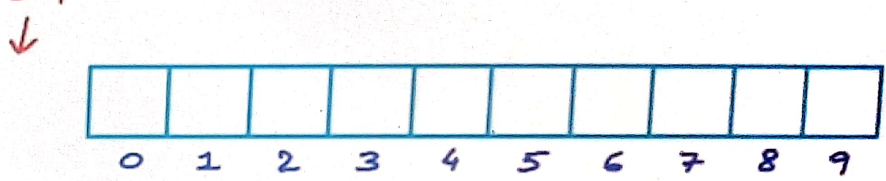



```
int A[10]   //Define int array.
top → -1    // take a variable give its value as -1

Push(value) {      //Define a push function
    top → top + 1      // Increment value of top by 1.
    A[top] = value     // Give the value.
}
```

Implementation of stacks

Array implementation

```
int A[10]
top ← -1   //empty stack
```

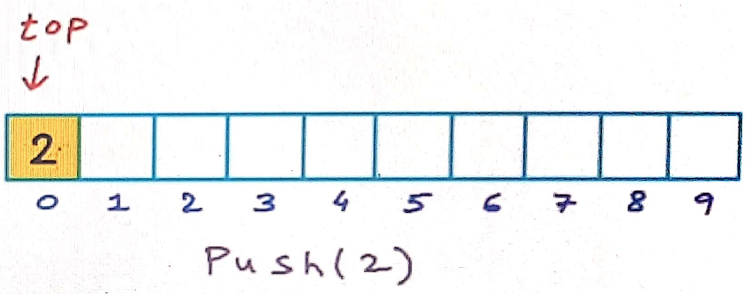



```
Push(x)
{
  top ← top+1
  A[top] ← x
}
```

Implementation of stacks

Array implementation

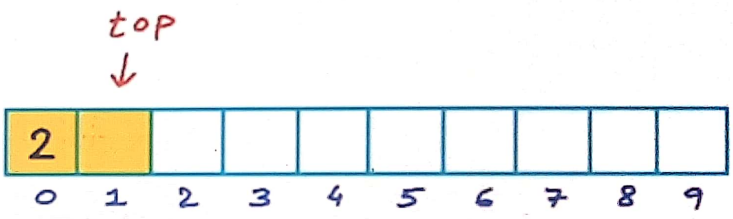
```
int A[10]
top ← -1   //empty stack

Push(x)
{
⇒  top ← top+1
   A[top] ← x
}
```



Implementation of stacks

Array implementation

```
int A[10]
top ← -1   //empty stack

Push(x)
{
   top ← top+1
⇒  A[top] ← x
}
```

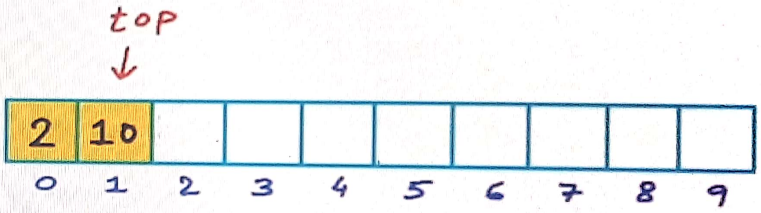


Push(2)
Push(10)

Implementation of stacks

Array implementation

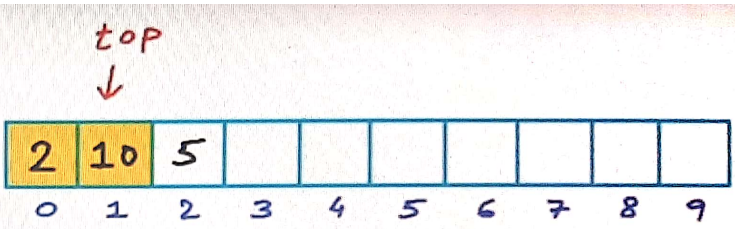
```
int A[10]
top ← -1   //empty stack

Push(x)
{
   top ← top+1
⇒  A[top] ← x
}
```



Push(2)
Push(10)

Array implementation

```
int A[10]
top ← -1   //empty stack
Push(x)
{
  top ← top+1
  A[top] ← x
}
PoP()
{
  top ← top-1
}
```



Push(2)
Push(10)
Push(5)
PoP()

Implementation of Stacks

So if we can add this property i.e. inserting and deletion from one end in a list then we can achieve Stacks datastructures.

Lists (i.e Arrays and Linked List)

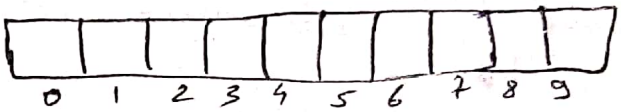
1) Array Implementation.

eg: If I wanna create a Stack of integers in Array

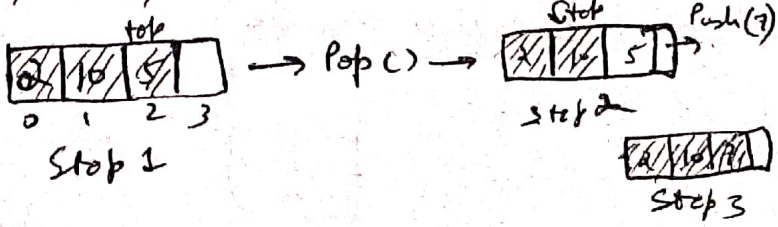
* In an empty stack, top is assumed to be -1

Pseudo Code

```
int A[10]   //Define int array.
top -> -1   // take a variable give its value as -1

Push(x) {       //Define a push function
    top -> top + 1      // Increment value of top by 1.
    A[top] = value      // Give the value.
}

Pop ( ) {
    top -> top - 1
}

top ( )
{
    return A[top]
}

IsEmpty() { if (top == -1)
              return true
            else return false }
```



Note:-> We will simply decrement the top position and we don't care about the element in previous top as our new stack will be reduced and anyway when again increase our stack, the value will be overwritten.

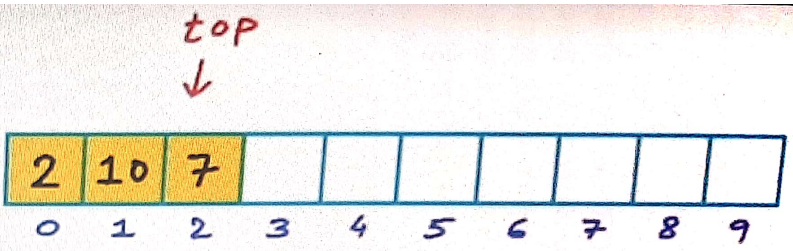


Push() and Pop() will take constant Time i.e O(1)

* Note:-> We can only push in an array stack until it is full. and further push will cause an overflow and this is the problem with array based implementation

Array implementation

```
int A[10]
top ← -1   //empty stack
Push(x)
{
  top ← top+1
  A[top] ← x
}
PoP()
{
  top ← top-1
}
```



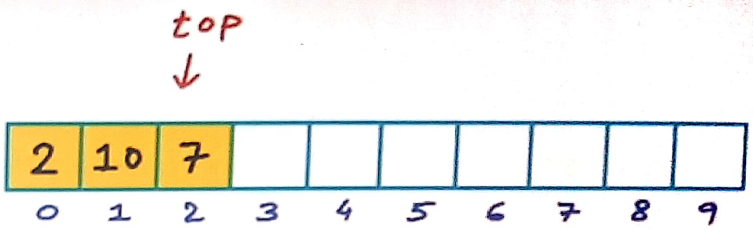
Overflow
↳ create a larger (twice the / smaller) array. copy all elements in new array.

Cost - O(n)
where n = no. of elements in stack

Push(2)
Push(10)
Push(5)
PoP()
Push(7)

Array implementation

```
int A[10]
top ← -1   //empty stack
Push(x)
{
  top ← top+1
  A[top] ← x
}
PoP()
{
  top ← top-1
}
```



Overflow
↳ create a larger (twice the / smaller) array. copy all elements.

Push - O(1) - best case
O(n) - Worst case
⇒ O(1) - Average case

c.n
O(n) for n pushes

Push(2)
Push(10)
Push(5)
PoP()
Push(7)

```c
// Stack - Array based implementation.
#include<stdio.h>
#define MAX_SIZE 101
int A[MAX_SIZE];
int top = -1;
void Push(int x) {
    if(top == MAX_SIZE -1) {
        printf("Error: stack overflow\n");
        return;
    }
    A[++top] = x;
}
int main() {

}
```

GNC, GCC

↓
compiler to add with Dev C++

#Queue ADT

First - In - First - Out

In queue, Insertion occurs from tail or rear end and removal/~~deletion~~ occur from the other end i.e. front/head.

Def
A Queque is a list or collection with the restriction or constraint that insertion can be performed at one end (rear) and deletion can be performed at other end. (front).

Operations:->

1) EnQueue(x) or Push(x) → It is used to insert an element at tail or rear end of queue.

2) Dequeue() or Pop() → It is used to remove or delete an element at front end of queue

3) Front() or Peek() :→ It just returns the element at front of the queue.

4) IsEmpty() :→ To check whether the queue is empty or not.

Introduction to Queues

Queue ADT

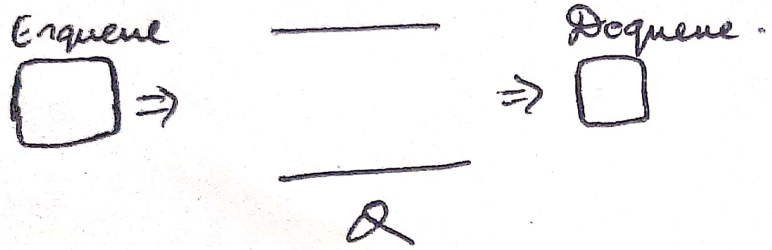
A list or collection with the restriction that insertion can be performed at one end (rear) and deletion can be performed at other end (front).

Operations

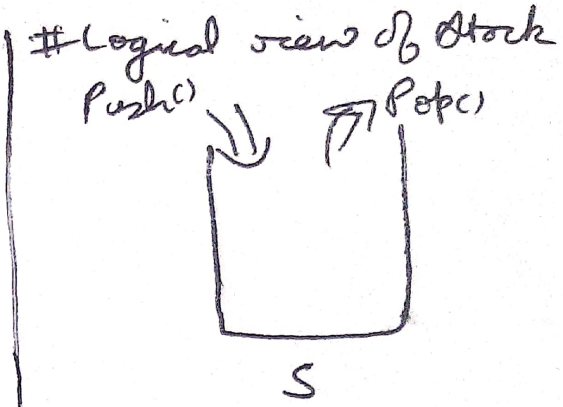
(1) EnQueue(x)
(2) Dequeue()
(3) front()
(4) IsEmpty()

} constant time or O(1)

#Logical view of queue



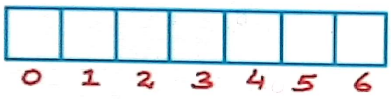
#Logical view of stack

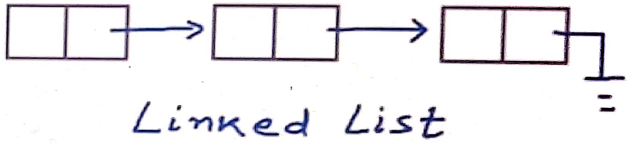


Introduction to Trees

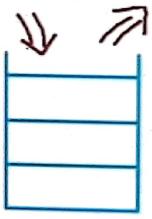
Linear data structures:

Array

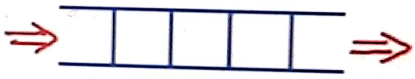




Linked List



Stack



Queue

Introduction to Trees.

* Linear data structures :-> In linear data structure data is stored in a sequencial manner i.e. these type of datastructures have a logical start and a logical end. and data is stored one after the other.

eg: Stacks, queues, Arrays, linked list etc.

Q) How to decide which datastructure to be Chosen?

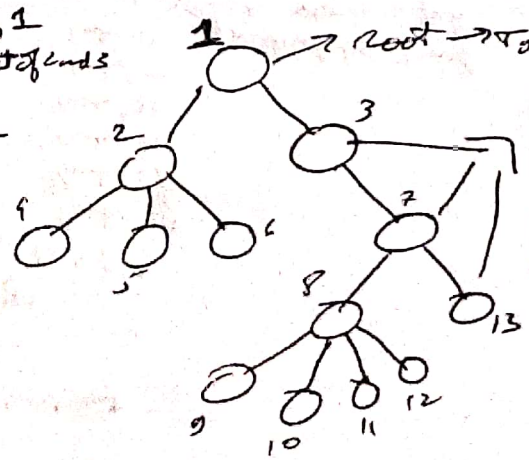
Ans) 1st look for these things

1. What needs to be stored.
2. Cost of Operations. / time
   eg: If we want to search data frequently then we may choose in Array in sorted order because time is O(1)
3. Memory Consumption:
4. Ease of Implementation.

Tree :- It is used to store heirarchical data eg: employees in an organisation and it is non linear datastructure.

or

A collection of entities called as node linked together to simulate or show heirarchy.



2,3 are children of 1 and 1 is the parent of 2,3

4,5,6 are children of 2 and node 2 is parent of 4,5 and 6.

Children of same parents are called siblings

so 2,3 are siblings
4,5,6 are siblings
7 has no sibling
8,13 are siblings 9,10,11,12 are siblings

1 → root → topmost node of the Tree

nodes

Any node which doesn't have a child is called leaf node
eg: 4,5,6,13,9,10,11,12 are leaf nodes

Each node contains data and this data could be of any type. and link to other nodes that maybe referred as children.

→ In linear data structures each data block occupies some memory.

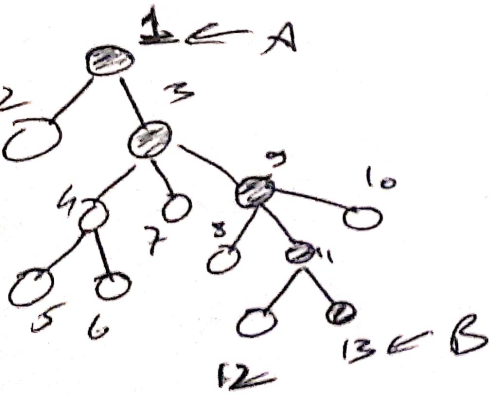
but

In non-linear data structures, its not the case

Look at tree every node has different nos of nodes attached to it
so obviously the memory consumption is different for each node.
Even in Binary tree also the same case applies.

↑ Akashish Sir ?

Tree is unidirectional i.e we can go from 1 to 2 but not 2 to 1 ×



Suppose if we can go from A to B

A = 1 B = 13

then

1, 3, 9, 11 are ancestors of B/13

or

B is decendent of 1, 3, 9, 11

Flag? in programming.

Flag is a variable, It is similar to usual variable but we use flag to show that whether we have achieved some task or not.

so we take its value as 1 or 0, i.e bool values

1 → for completion

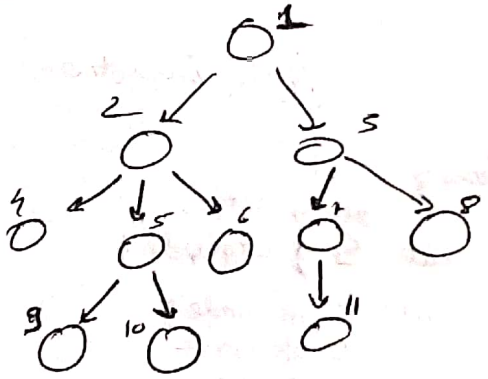
0 → for incomplete work.

we can also take int as 1 and 0 as integers but we prefer bool because it takes less space

bool → 1 byte
int → 4 byte

Choose wisely

Q) 1) What are the common ancestors of 4 and 9



2) Are 6 and 7 cousins?

Ans) 1) 1 and 2 are common ancestors.

2) Cousin :- Nodes having same grandparents are called as cousins.

So yes 6 and 7 are cousins as their grandparent is 1.

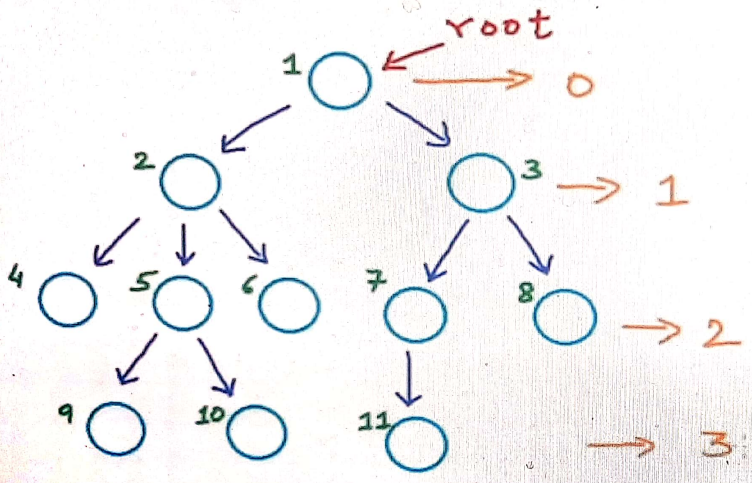
Tree :- It could be also called as recursive data structure.

↓

Reducing something in self similar manner.

A tree with N nodes will exactly have (N-1) edges.

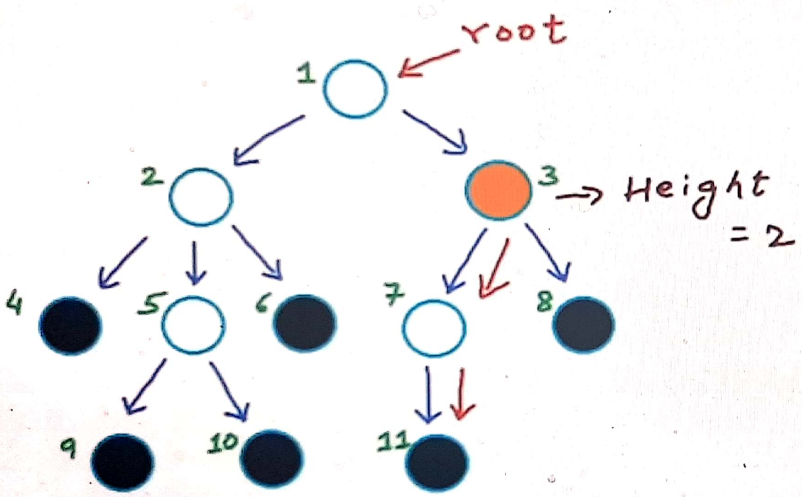
Introduction to Trees



Depth and Height

Depth of x =
length of path from root to x
OR
No. of edges in path from root to x

Introduction to Trees



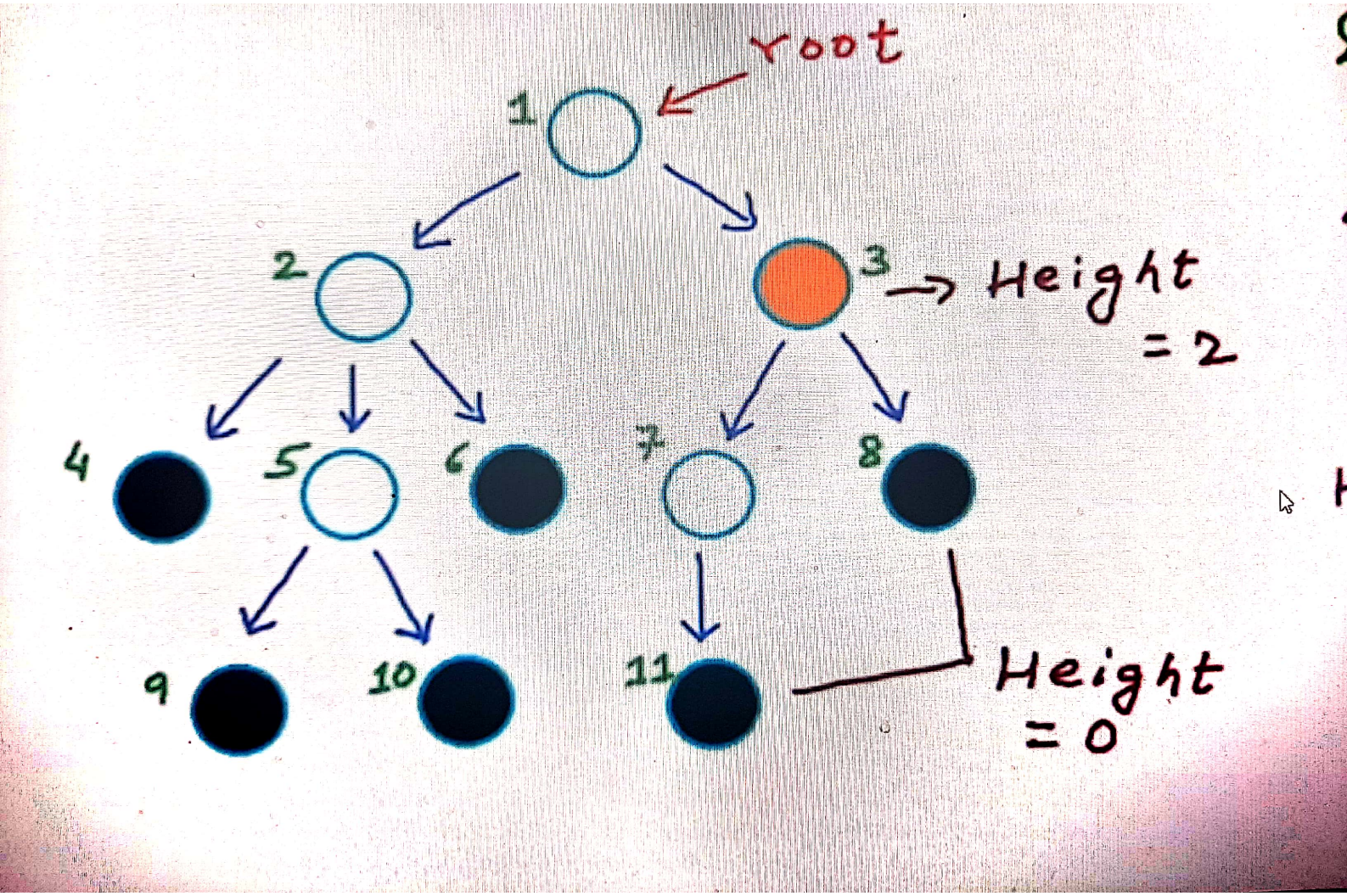
Depth and Height

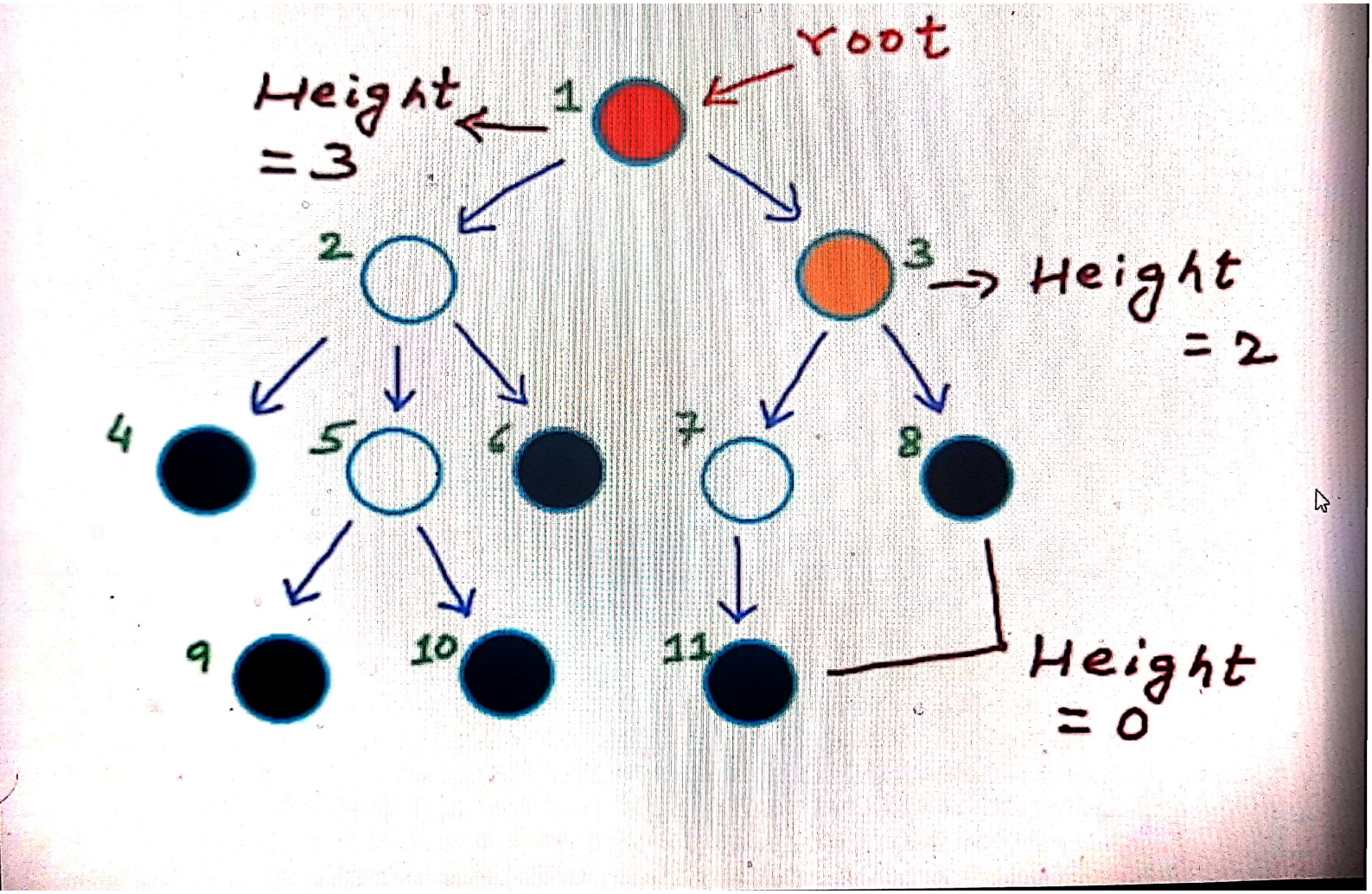
Depth of x =

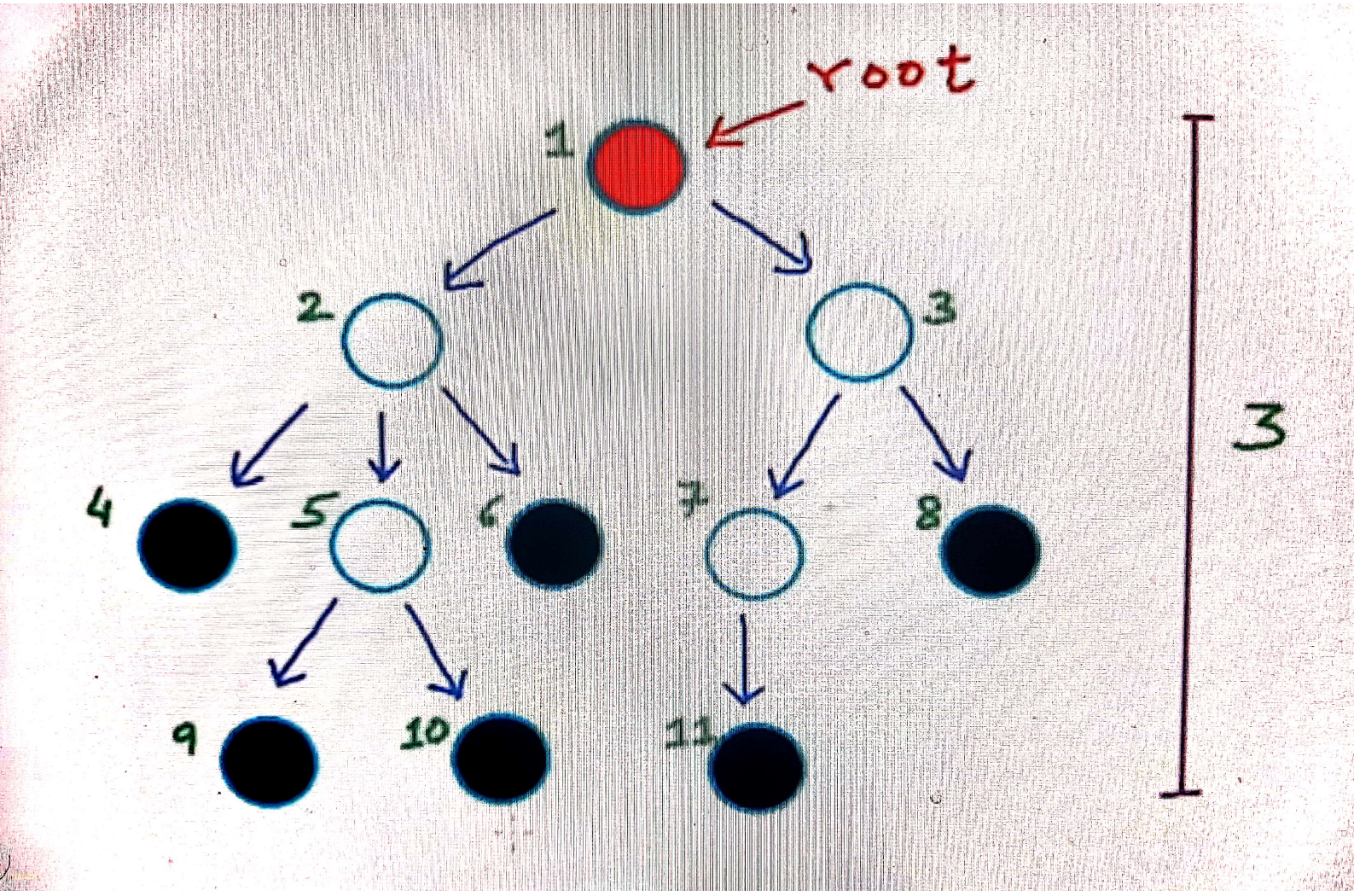
No. of edges in path from root to x

Height of x =

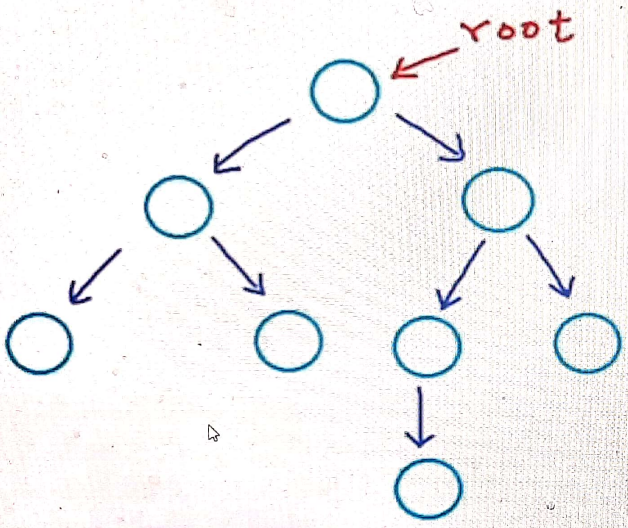
No. of edges in longest path from x to a leaf


root
1
2
3
Height
= 2
4
5
6
7
8
9
10
11
Height
= 0


root
Height
= 3
1
2
3
→ Height
= 2
4
5
6
7
8
9
10
11
Height
= 0


root
1
2
3
4
5
6
7
8
9
10
11
3

Introduction to Trees

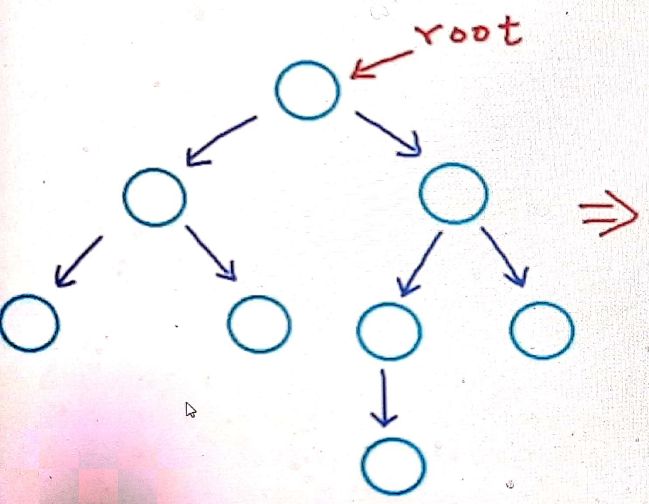


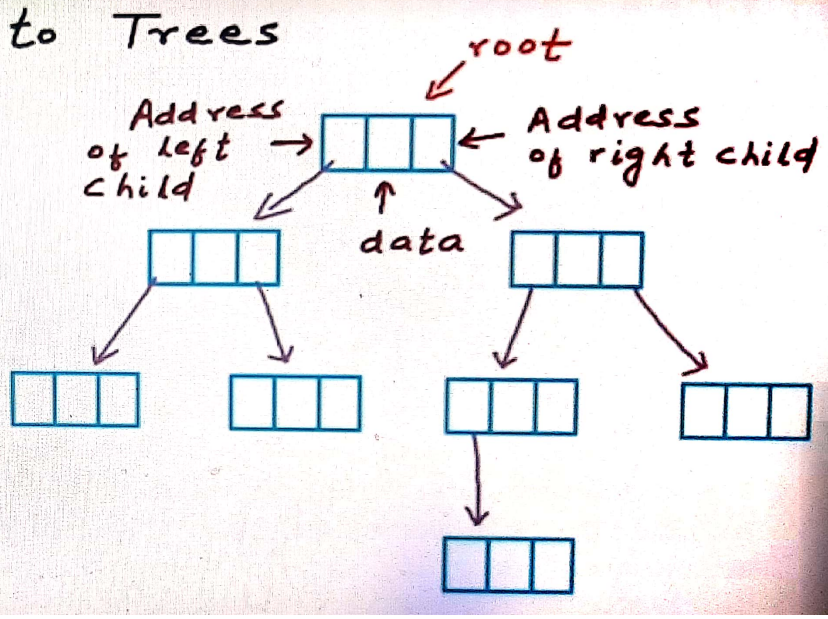
Binary Tree

↓

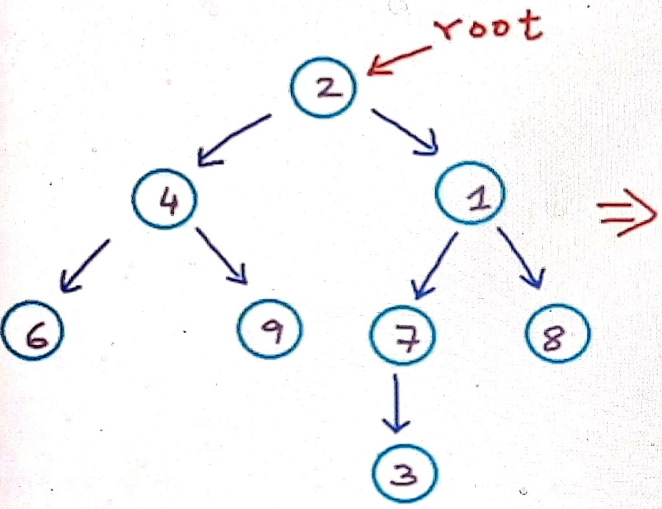
a tree in which each node can have at most 2 children

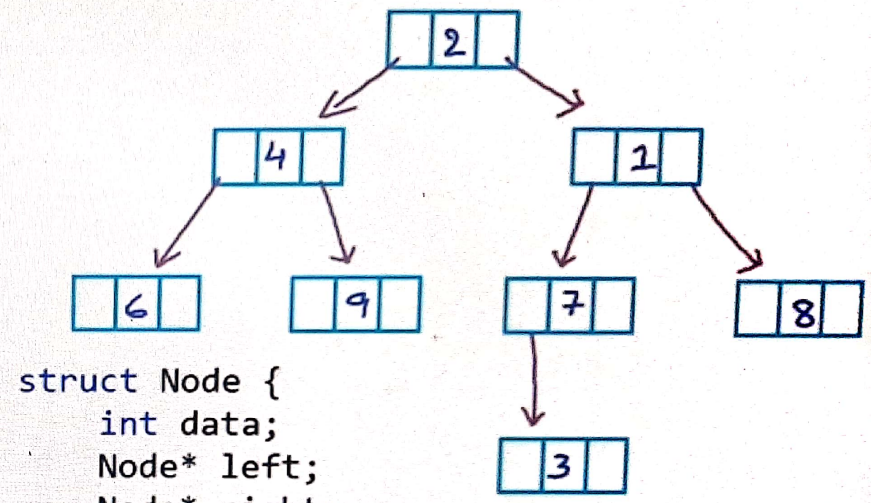
Introduction to Trees





Introduction to Trees






```
struct Node {
    int data;
    Node* left;
    Node* right;
};
```

Introduction to Trees

Applications:

1) Storing naturally hierarchical data → eg:- file system

2) Organize data for quick search, insertion, deletion
   → eg:- Binary Search trees

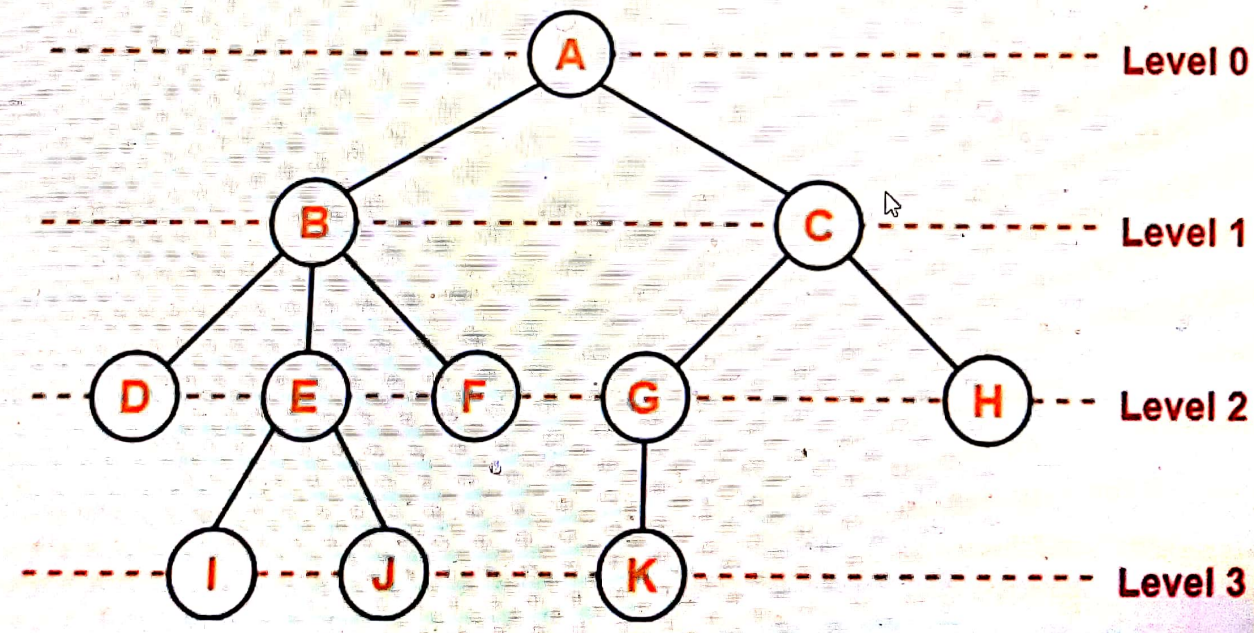
3) Trie → dictionary

4) Network Routing algorithm

9. Level-

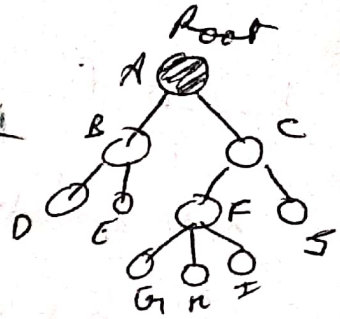
- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.

Example-



It's not a child to anyone.

Root → The topmost node of the Tree
or
from where tree originates is called Root



Edge → A connecting link between any two node is called edge.

Parent

A is parent to B and C so B and C are siblings.
and B is a Child to A
C is a Child to A

Siblings → Nodes belonging to same parent.

Degree :→ No of Children of a node

B → degree of B is 2
A → degree of A is 2
D → degree of D is 0
F → " " F is 3

Highest degree in fig. is degree of Tree.
So tree degree is 3.

Internal Nodes :→ The nodes which has atleast one child is called and Internal node - except root.

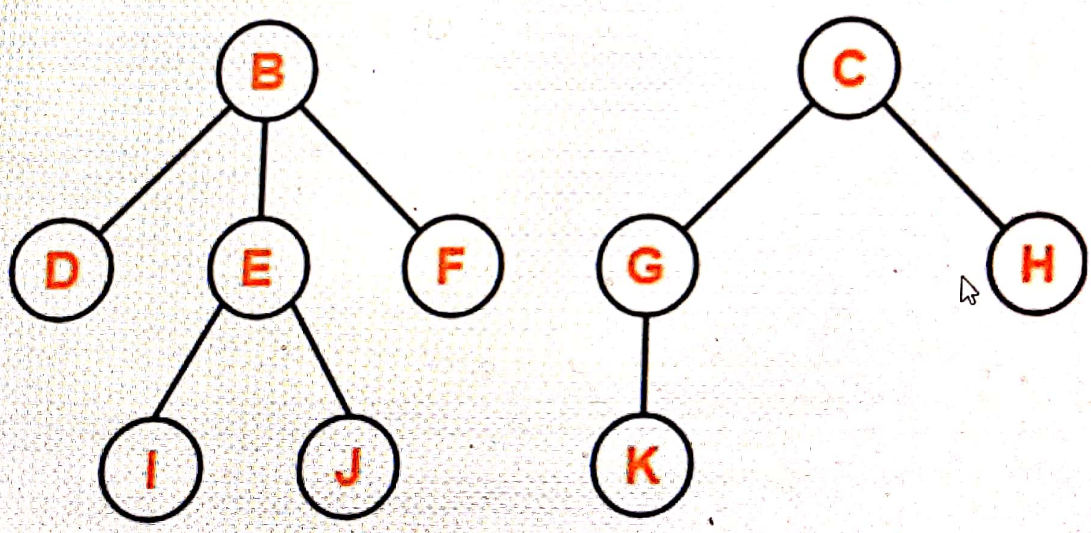
B, C, F are Internal Nodes.
Or All nodes between leaf and Root are Internal Nodes.

Terminal Nodes / Leaf Nodes :→ Those nodes which don't have any child.

13. Forest-

A forest is a set of disjoint trees.

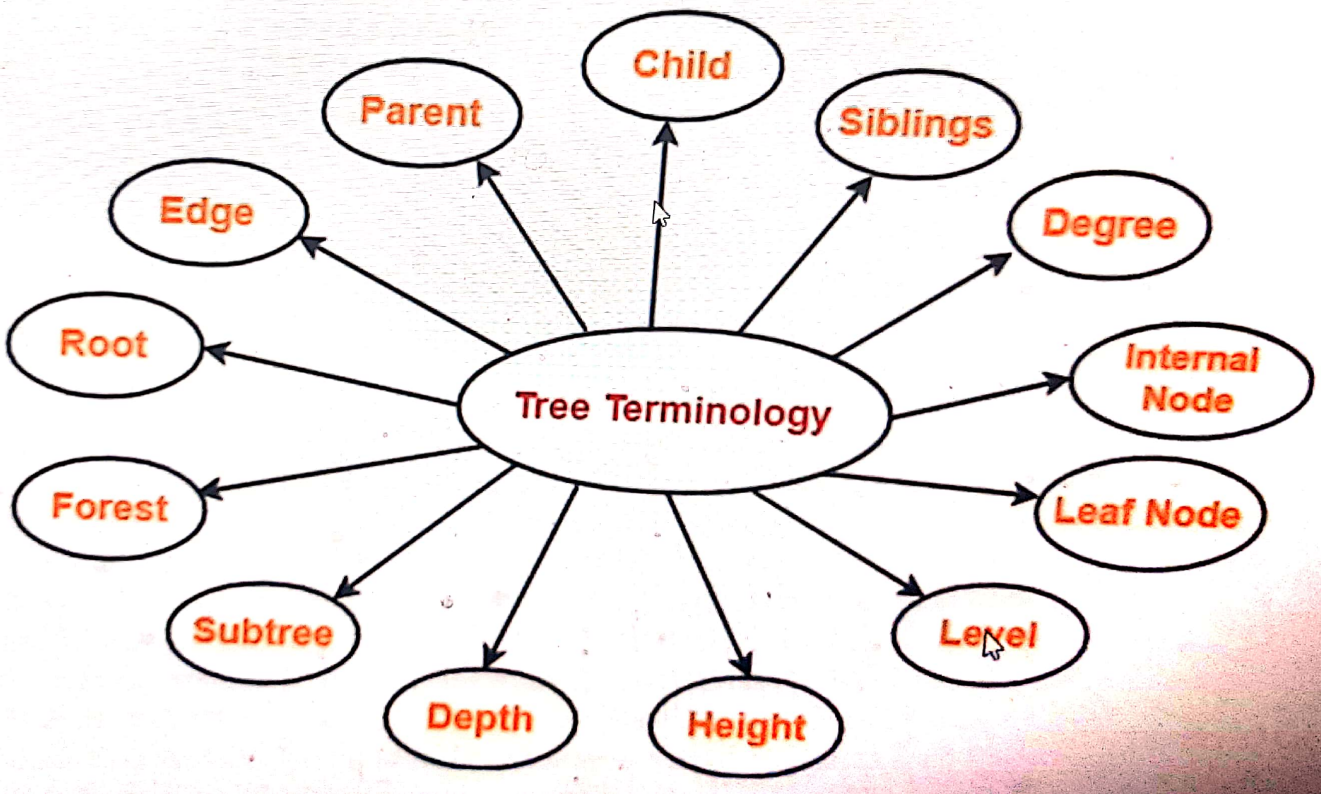
Example-



Forest

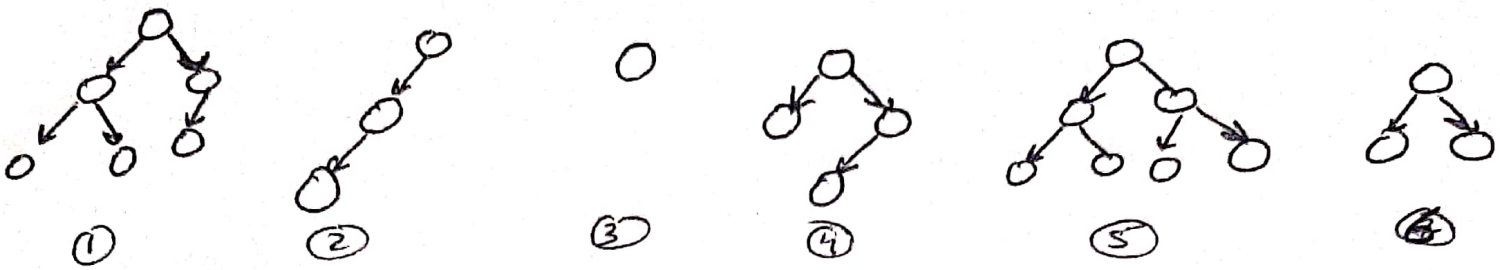
Tree Terminology-

The important terms related to tree data structure are-



Binary Tree

Each node can have almost two children.



These all are Binary Trees and fig (5) and (6) are types of Strict Binary Tree.

★ Note:→ Max no. of nodes at level $i = 2^i$

Maximum nos of nodes in a binary tree with height h

$$2^0 + 2^1 + \cdots + 2^h$$

$$\Rightarrow 2^{h+1} - 1$$

a) What is the height of a perfect binary tree with n nodes.

Ans ⇒

$$n = 2^{h+1} - 1$$

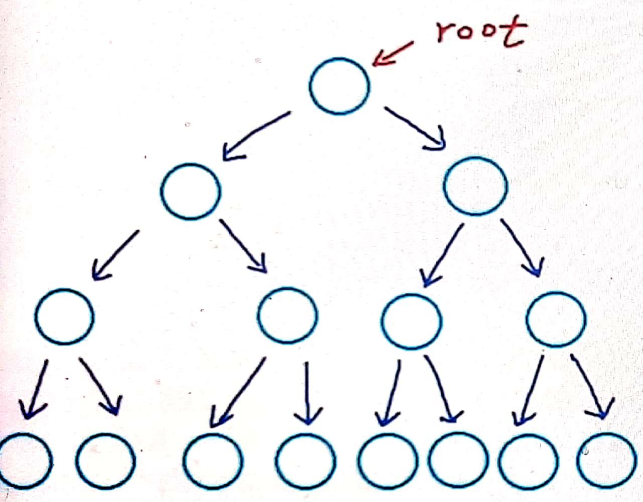
$$(n+1) = 2^{h+1}$$

$$\log(n+1) = \log 2 \times (h+1)$$

$$\frac{\log(n+1)}{\log(2)} = h+1$$

★ $$h = \log_2(n+1) - 1$$

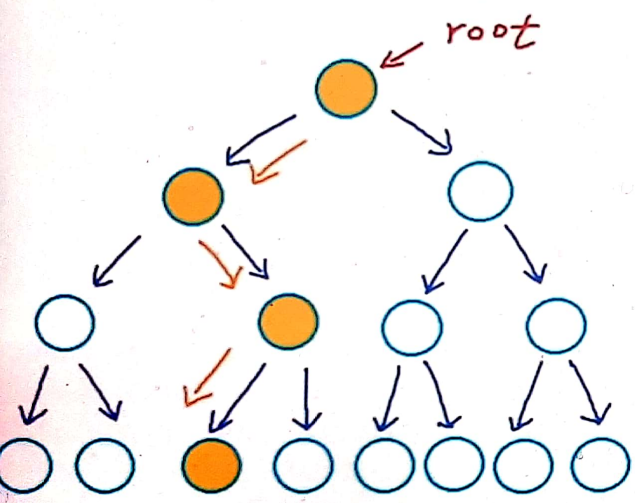
Binary Tree



Balanced binary tree
↳ difference between height of left and right subtree for every node is not more than k (mostly 1)

Height → no. of edges in longest path from root to a leaf

Binary Tree



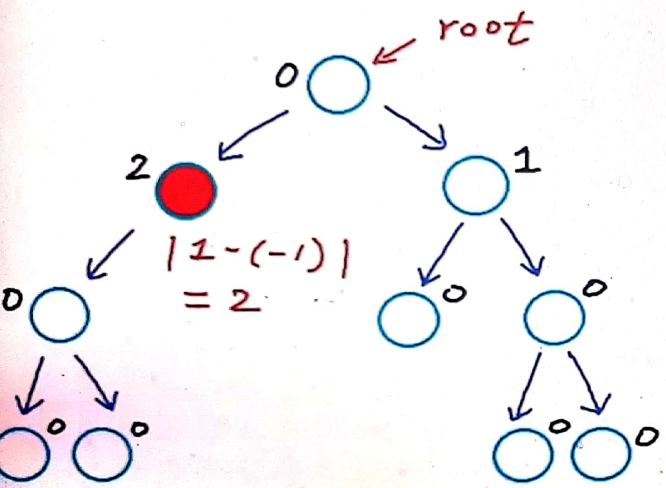
Balanced binary tree
↳ difference between height of left and right subtree for every node is not more than k (mostly 1)

Height → no. of edges in longest path from root to a leaf

Height of an empty tree = -1

Height of tree with 1 node = 0

Binary Tree



Balanced binary tree
↳ difference between height of left and right subtree for every node is not more than k (mostly 1)

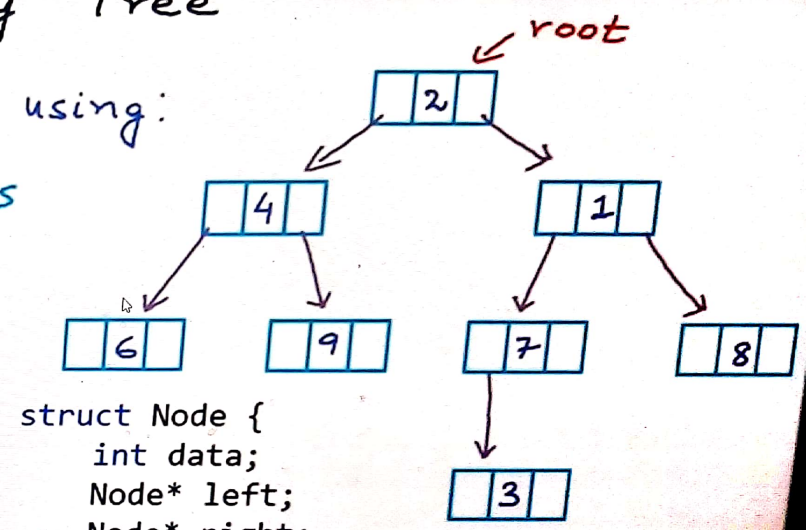
diff = | $h_{left} - h_{right}$ |

Binary Tree

We can implement binary tree using:

a) dynamically created nodes

b) arrays



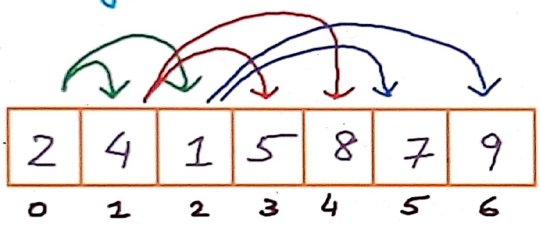

```
struct Node {
    int data;
    Node* left;
    Node* right;
};
```

Binary Tree

We can implement binary tree using:

a) dynamically created nodes

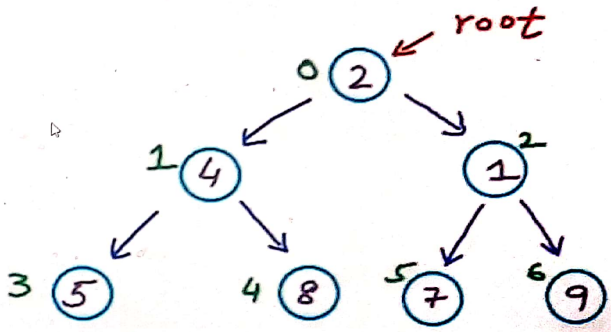
b) arrays



for node at index i,

left-child-index = 2i+1

right-child-index = 2i+2



Binary Search Tree

	Array (unsorted)	Linked List	Array (sorted)	BST (balanced)
Search(x)	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$
Insert(x)	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$
Remove(x)	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$

For n records, $\log_2 n$ comparisons

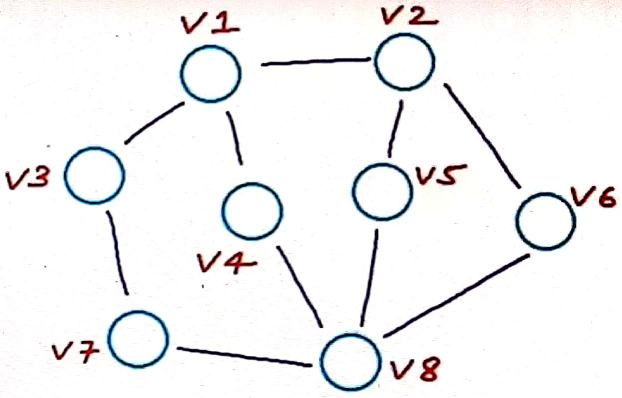
if 1 comparison = 10^{-6} sec

$n = 2^{31} \Rightarrow 31 \times 10^{-6}$ sec

Graph:

A graph G is an ordered pair of a set V of vertices and a set E of edges.

$$G = (V, E)$$



$$V = \{v1, v2, v3, v4, v5, v6, v7, v8\}$$

$$E = \{\{v1,v2\}, \{v1,v3\}, \{v1,v4\}, \{v2,v5\}, \{v2,v6\}, \{v3,v7\}, \{v4,v8\}, \{v7,v8\}, \{v5,v8\}, \{v6,v8\}\}$$

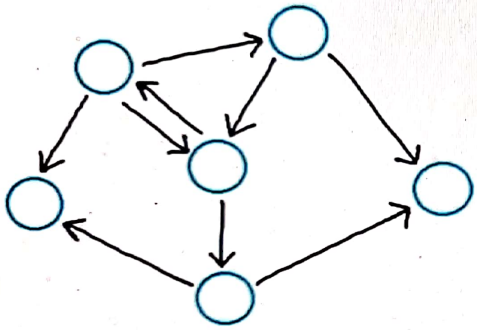
Edges:

directed
(u, v)

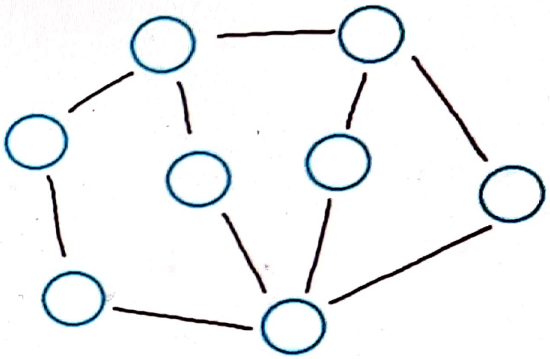
undirected
$\{u, v\}$

Introduction to Graphs

Directed vs Undirected

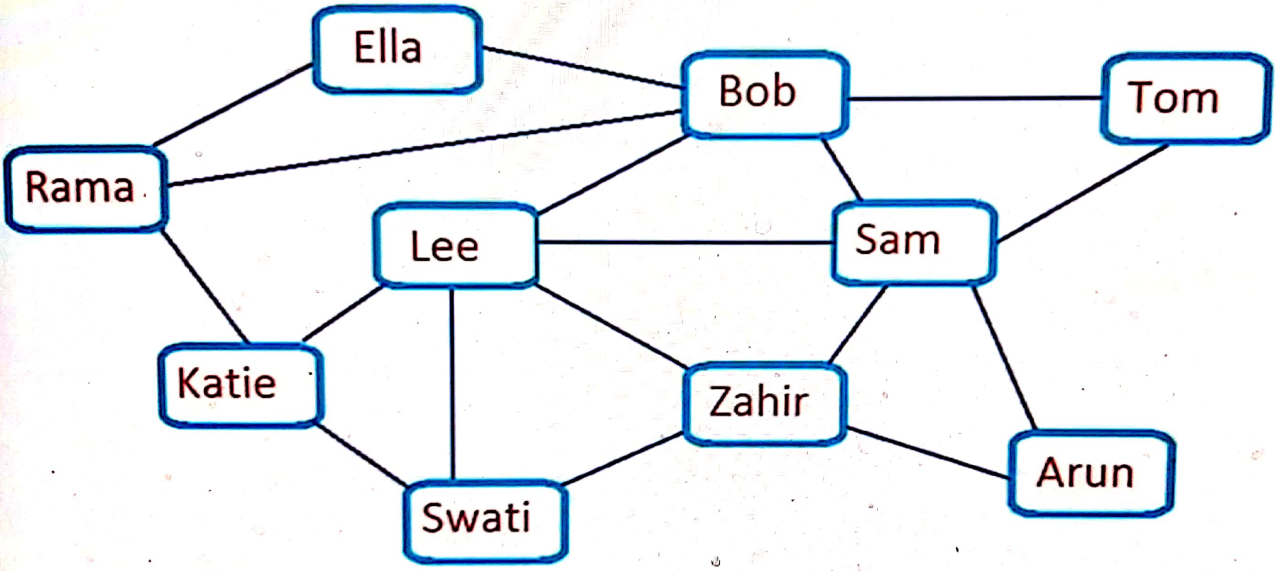


a directed graph
or
Digraph



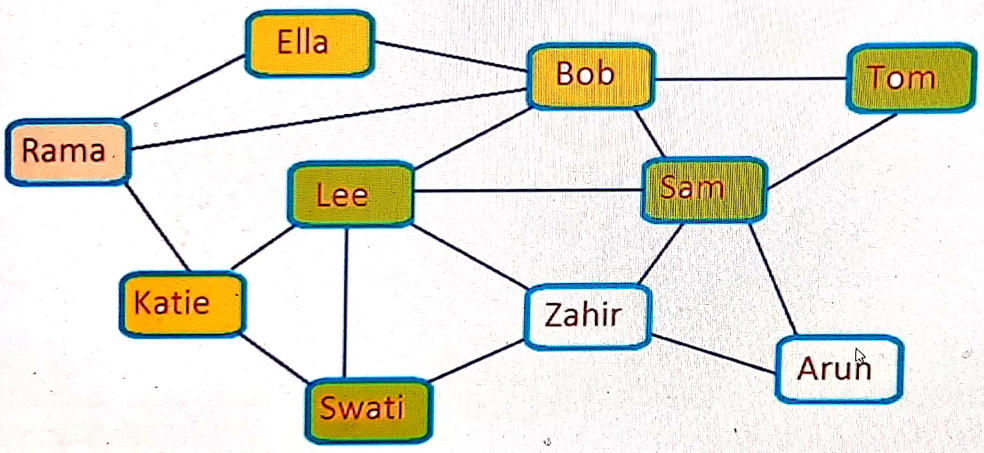
an undirected graph

Introduction to Graphs

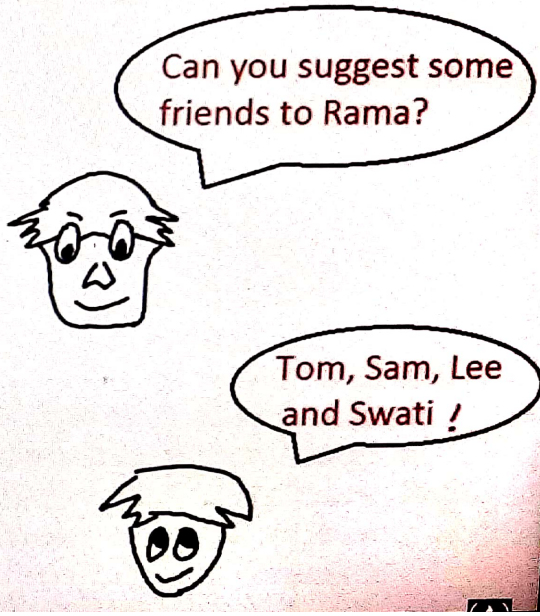


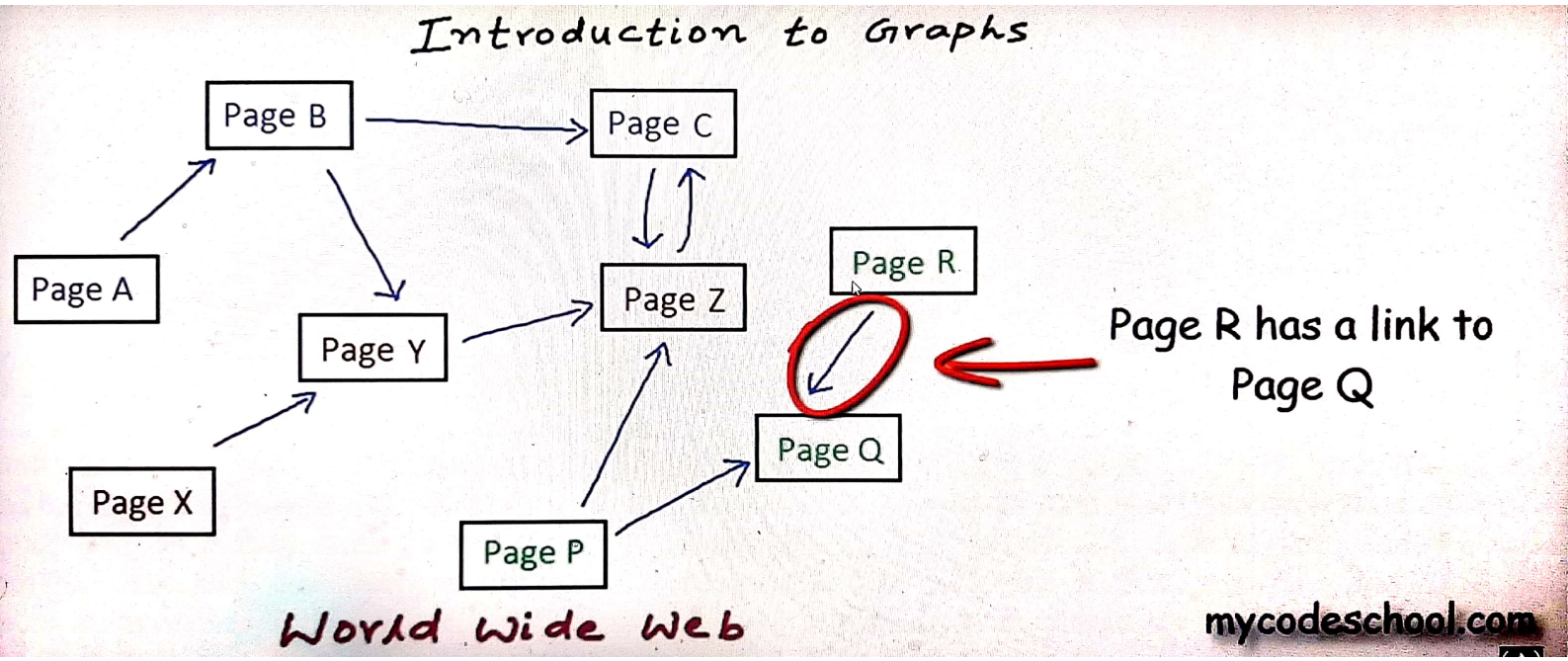
Social Network
(facebook)

Introduction to Graphs



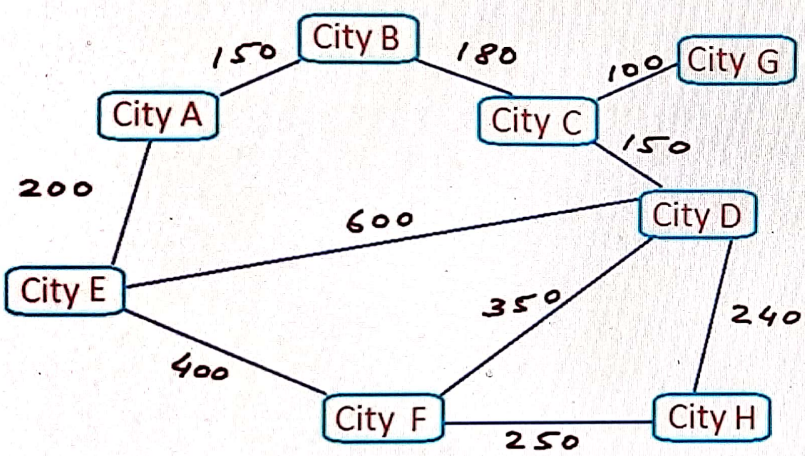
Social Network
(facebook)




Introduction to Graphs
Page A
Page B
Page C
Page X
Page Y
Page Z
Page P
Page R
Page Q
Page R has a link to Page Q
World Wide Web
mycodeschool.com

Introduction to Graphs

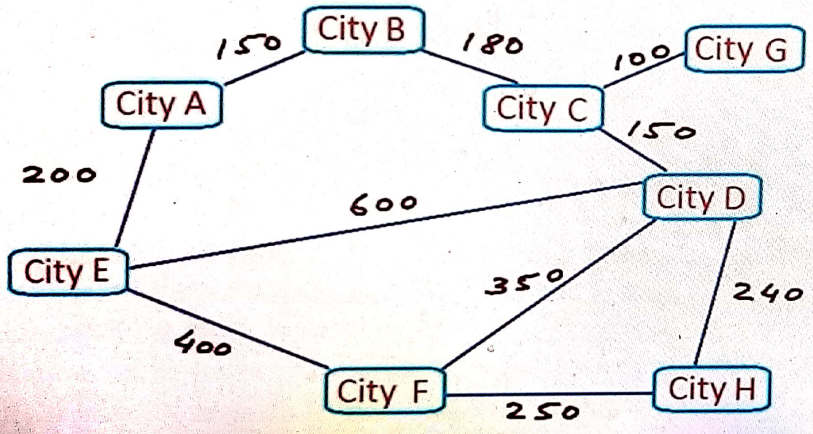
Weighted vs Unweighted



← Intercity road network

Introduction to Graphs

Weighted vs Unweighted



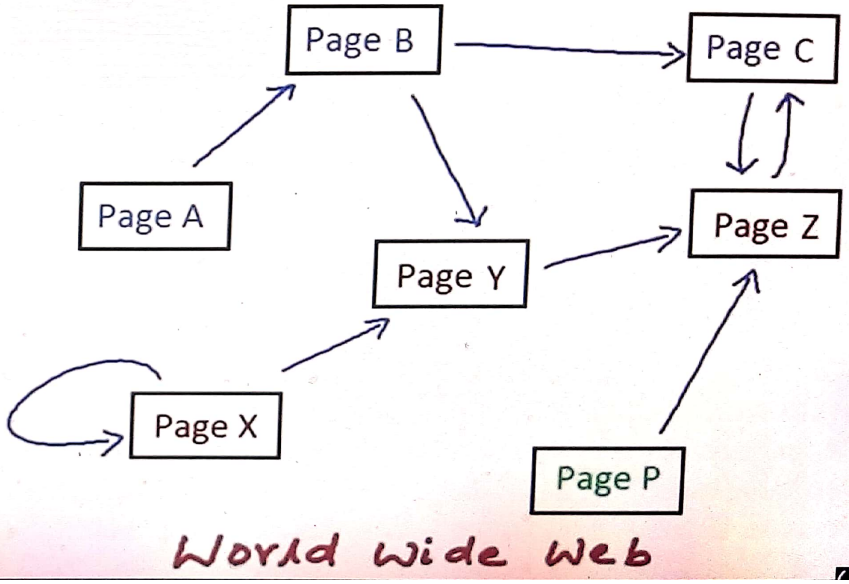
Unweighted graph

↳ a weighted graph with all edges having weight = 1 unit

← Intercity road network

Properties of Graphs

Self-loop:

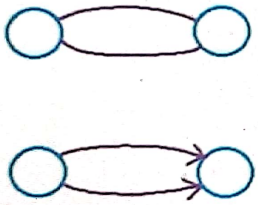


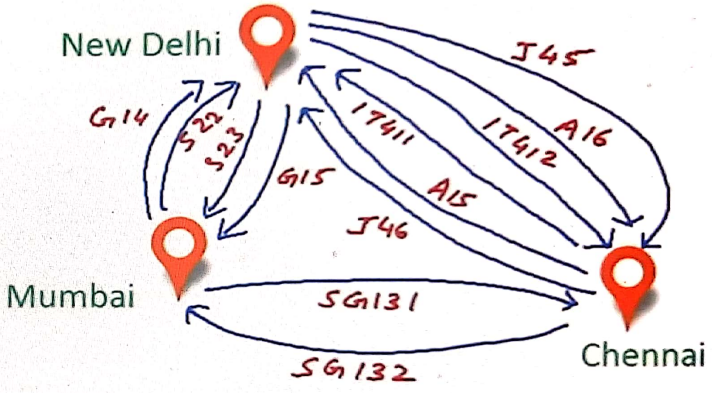
World Wide Web

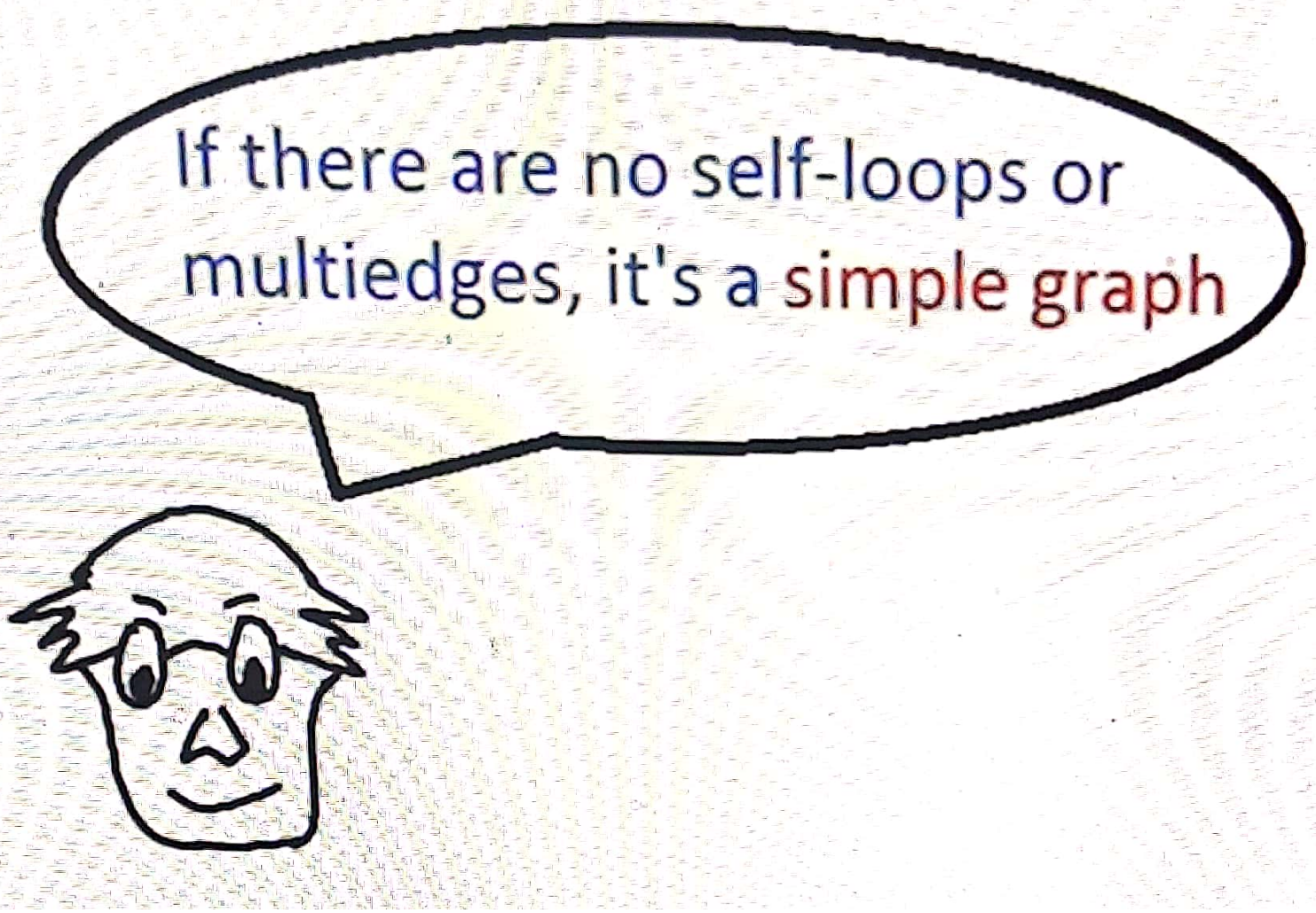
Properties of Graphs

Self-loop:

Multiedge:






If there are no self-loops or multiedges, it's a simple graph

Properties of Graphs

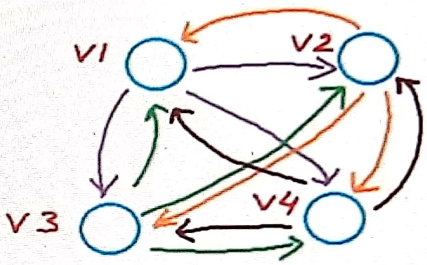
Number of edges:

if $|V| = n$

then,

$0 \leq |E| \leq n(n-1)$, if directed

$0 \leq |E| \leq \frac{n(n-1)}{2}$, if undirected



↳ assuming no self-loop or multiedge

Properties of Graphs

Number of edges:

if $|V| = n$

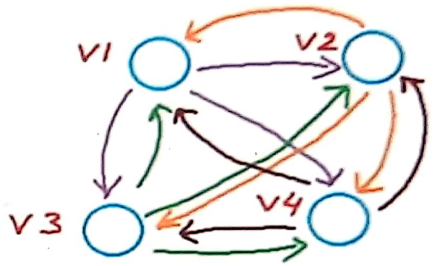
then,

$0 \leq |E| \leq n(n-1)$, if directed

$0 \leq |E| \leq \frac{n(n-1)}{2}$, if undirected

Dense → too many edges

Sparse → too few edges



if $|V| = 10$, $|E| \leq 90$

if $|V| = 100$, $|E| \leq 9900$

Graph Representation

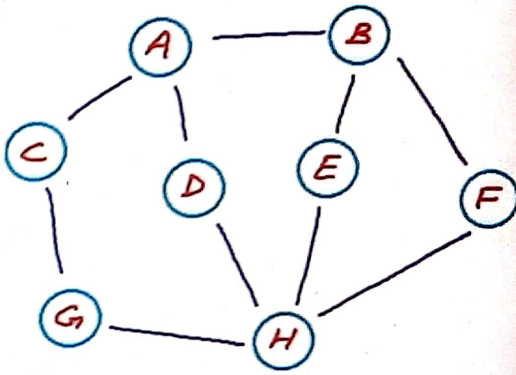
Vertex List

A
B
C
D
E
F
G
H

Edge List

A	B
A	C
A	D
B	E
B	F
C	G
D	H
E	H
F	H
G	H

```
struct Edge
{
    char *startVertex;
    char *endVertex;
};

        OR

class Edge
{
   string startVertex;
   string endVertex;
};
```



← Struct Edge EdgeList[MAX_SIZE]

Graph Representation

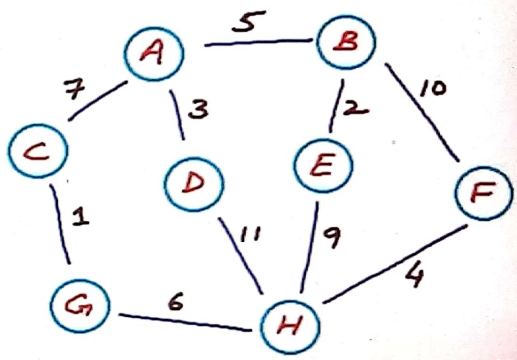
Vertex List

A
B
C
D
E
F
G
H

Edge List

A	B	5
A	C	7
A	D	3
B	E	2
B	F	10
C	G	1
D	H	11
E	H	9
F	H	4
G	H	6

```
struct Edge
{
    char *startVertex;
    char *endVertex;
    int weight;
};

      OR

class Edge
{
   string startVertex;
   string endVertex;
   int weight;
};
```



← Struct Edge EdgeList[MAX_SIZE]

#Hashing :→ It is a technique to uniquely identify a specific object from a group of similar objects

UBUNTU
FLASK
MySQL/Database
Twitter bootstrap
Digital Ocean
NGINX

In Hashing → longer keys are converted into small keys by using hash functions.
These key values are then stored in a datastructure called as hash table.

eg:

key	Value
Cuba	Havana
England	London
France	Paris
Spain	Madrid
Switzerland	Berne

1) Simply count the number of characters
So Cuba has length of 4 and place it at this index
England has " " 7 and place at this index
France " " 6 " "

1) Search O(1)
2) Insert O(1)
3) Delete O(1)

Direct Access table not works fine
Because, it can't handle large values.

Hashing not Value
1) When we ~~key~~ to use for sorted data as everything here is stored in a random manner.

#Hash Function Works
1) → Should always map a large key to some small key.
2) → Should generate values from 0 to n-1 if n is size of Array/hash table
3) → Should be fast
4) → Should be uniform

Examples of hash function
1) keykey = large key % m

m → ideally choose prime number as chances of collision is less
Bad value of m → power of 2
10 etc.

2) For string eg: str[] = "abcd"
str[0] * x^0 + str[1] * x^1 + ...

ASCII code value.

x is prime nos.

Hashing.

1) Direct hashing :→ Whatever the number is, we'll place it to the corresponding index/part of the Array / hash table.

Disadvantage:

1) the keys are limited to location range
2) We can't store large values.

2) Subtraction Method :-

$h(k) = k - m$ m is a chosen integer.

eg: My range is 0–99 and the k=4 I choose 4

Disadvantage

1) We can't store numbers lesser than that given the number

and

eg: $h = k - 4$

the range is limited to

4 to 103

Or

the keys here also are limited to location range.

3) Division Method.

Modulo division method

$h(k) = k \bmod m$

or

$k \% m$

Range of location we get → 0 to m−1

Collision Resolution technique.

1) Linear Probing :-

$L = H(k)$ if collision comes at L

then

$L_{new} = (L_{old} + 1) \bmod m$ until you find next free location and if you can't find that means there is no location free now.

2

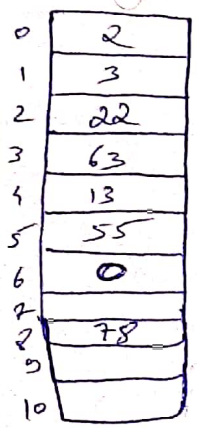
$L_{new} = (L + i) \bmod m.$

Q) The keys 22, 78, 63, 2, 13, 3, 55 and 5 are inserted into an initially empty hash table of length 10 using open addressing with hash function $h(k) = k \bmod 10$ and linear probing. What is the ① resultant hash table ? and ② total no of collisions

length = 10

$h(k) = k \bmod 10$

A →



$22 \% 10 = 2$

$78 \% 10 = 8$

$63 \% 10 = 3$

$2 \% 10 = 2 \rightarrow$ "

$13 \% 10 = 3 \rightarrow L_{new} = (L+i) = (3+1) = 4$

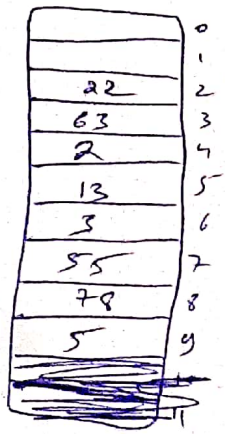
$3 \% 10 = 3 \rightarrow L_{new} = (L+i) = (3+1) = 4$

$55 \% 10 = 5$

$5 \% 10 = 5 \rightarrow \therefore \rightarrow 6^{th}$ index

Note :- The problem with Linear Probing is that if we want to insert any value 52 in this table then its index will come 2 which is already occupied and with linear probing it will face so many collision until it gets index 0.

A →



$22 \% 10 = 2$ ✓

$78 \% 10 = 8$ ✓

$63 \% 10 = 3$ ✓

$2 \% 10 = 2$ collision so $L_{new} = 2+1 = 3$ again collision so $L_{new} = 3+1 = 4$

$13 \% 10 = 3 \rightarrow 4 \rightarrow 5$ ✓

$3 \% 10 = 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$ ✓

$55 \% 10 = 5 \rightarrow 6 \rightarrow 7$

$5 \% 10 = 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$ ✓

total no of Collision = 13