Array

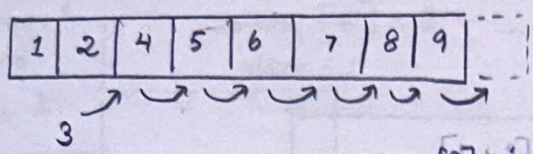
* collection of datatypes - same datatype.

* Inserting:

Bestcase : free O(1)

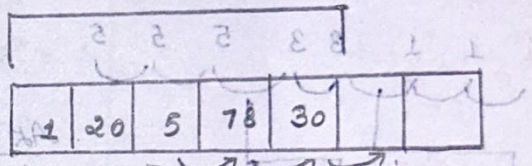
worstcase : occupied : shift (O(N))

Adding element




```
temp = array[i];
array[i] = elem;
elem = array[i+1];
```

5 elements filled



I want to add by index:

size = 10

position = 2

Element = 100

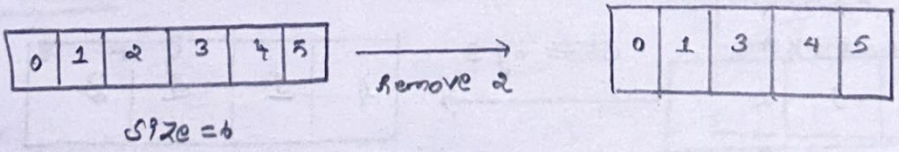
Algorithm : Right shift

temp = array[i]; → Backup for future use

array[i] = elem; → From user (1st loop) - current num for next replacing

elem = temp; → For next replacement.

Remove



(or) - Remove first occurence of 2

Left shift

arr[i] = arr[i+1] → up to shown to n-2 ∴ [arr[n-2] = arr[n-1]

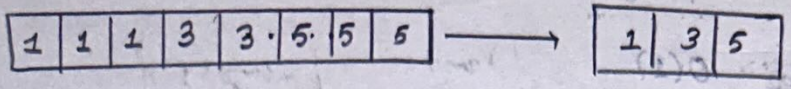
Now after loop: count--;

Time Complexity:

O(1) - last position

O(n) - First position. [N-1 iterations]

No additional memory - Remove duplicates from the sorted array



Sorted: when no repitation: — when repitation:

if (arr[i] == arr[i-1]) → Repeat [0 to n-2]

1 1 1 3 3 5 5 5

* skaark = 1

```
when arr[i] != arr[i+1]
    array[skaark] = array[i+1];
    skaark++;
```

→ working?

1 1 1 3 3 5 5 5

1 3

At last

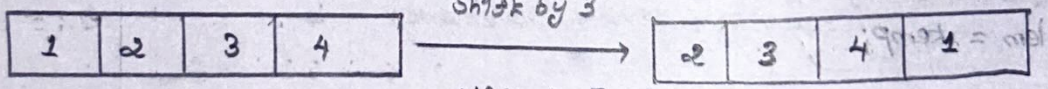
```
array[count++] = array[i];
```

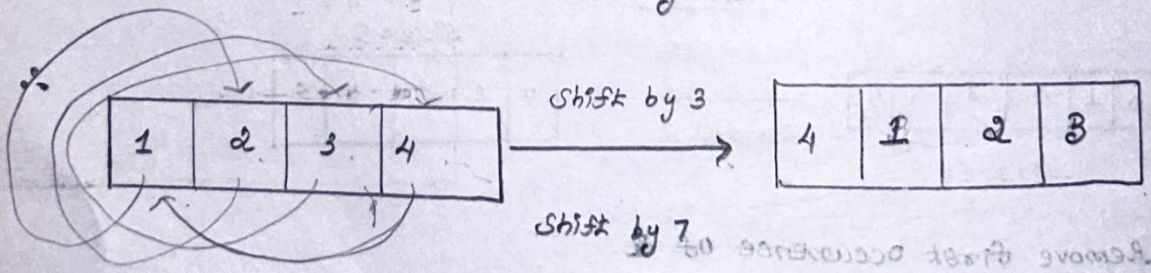
Rotate an array by k positions

say

+ve → Right shift

-ve → Left shift.





Left shift = Size ~ Right shift

concept do Right shift!

when −3 → 4−3 = 1

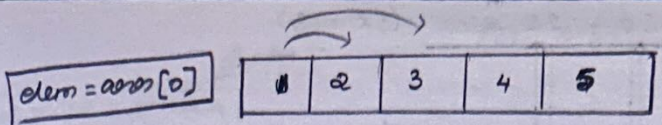
−4 → 4+4 = 0

−5 → means → −1 → 4−1 = 3.

---

1 2 3 4 5 ⇒ K=2 → 4 5 1 2 3

3. 2 3 4 5 K=3 → 3 4 5 1 2



temp = 2
arr[1] = elem;
elem = temp;

temp = 3
arr[2] = elem;
elem = 3;

temp = 4
arr[3] = 3
elem = 4

temp = 5
arr[4] = 4;
elem = 5

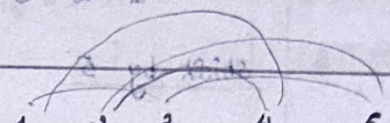
(1 shift)
Start from: 1
up to
< size + 1

temp = 1
arr[0] = 5 ⟶ done.
elem = 1

nshift
start from n
up to
< size + n

Inside: (size + i) ⇒ (i % size)

1 % 5 → 1	2 → 2
2	3 → 3
3	4 → 4
4	5 → 0
5 ⟶ 0	6 → 1



temp = 2
arr[2] = 1
elem = 2

temp

2 % 5 = 2 ——
4 % 5 = 4 ——
6 % 5 = 1 ——
8 % 5 = 3 ——
10 % 5 = 0 ——

K = 2 means

1 2 3 4 5
↓
5 1 2 3 4
↓
4 5 1 2 3

?

2 % 6 = 2
4 % 6 = 4
6 % 6 = 0
8 % 6 = 2
10 % 6 = 4
12 % 6 = 6

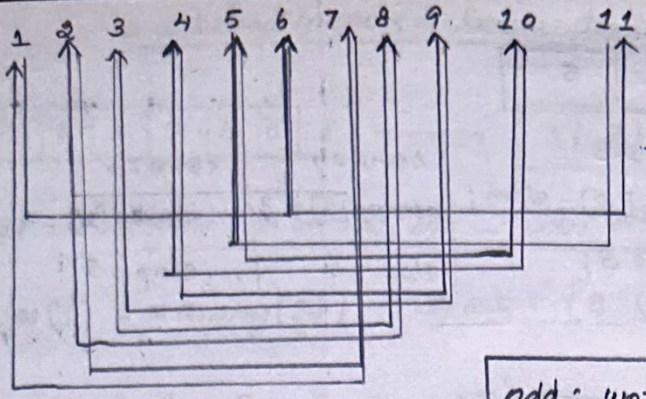
2 % 5 = 2
4 % 5 = 4

Complexity: $O(n) = n^2$

Right Rotate:

```
int temp = arr[n-1], i;
for (int i = n-1; i > 0; i++) {
    arr[i] = arr[i-1];
}
arr[0] = temp;
```

⟶ Call that many times
K = 2 → 2 times.



(5 times)

→ won't work for even (loop)

odd : working!

even: No!

1 2 3 4 5 6 7 8 9 10 11 12

2 % 6 = 2

4 % 6 = 4

6 % 6 = 0 → 0

8 % 6 = 2 → 3 → 2

10 % 6 = 4 → 5 → 4

12 % 6 = 0 → 7 → 6

shift by 5

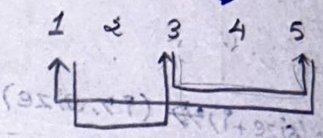
5 % 12 = 5

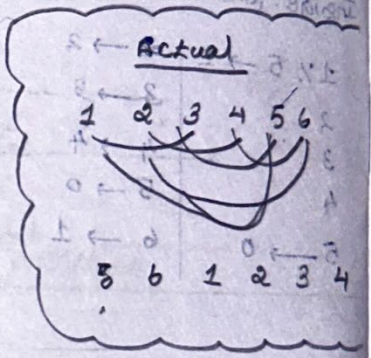
10 % 12 = 10

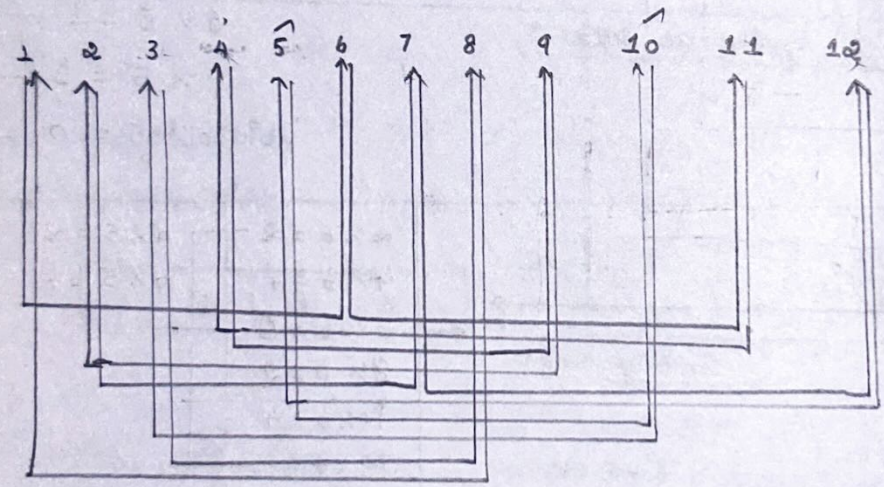
15 % 12 = 3 → 4

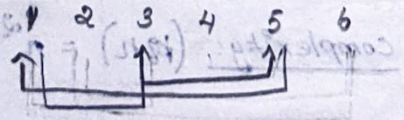
20 % 12 = 8 → 9

25 % 12 = 1 → .









→ Incomplete.

Algo :

2 % 6 = 2

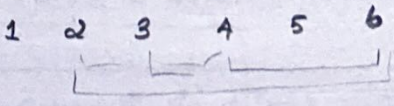
4 % 6 = 4

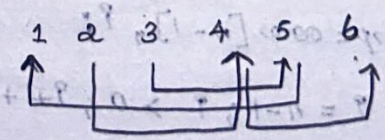
6 % 6 = 0

8 % 6 = 2 → 1

10 % 6 = 4 → 3

12 % 6 = 0 → 5





div by 2

2 % 12 = 2 → 0
4 % 12 = 4 → 0
6 % 12 = 6 → 0
8 % 12 = 8 → 0
10 % 12 = 10 → 0
12 % 12 = 0 → 11th position
14 % 12 = 2 → 1
16 % 12 = 4 → 3
18 % 12 = 6 → 5
20 % 12 = 8 → 7
22 % 12 = 10 → 9
24 % 12 = 0 → 11

14 − (14/12) % 12 = 14 − 1 % 12 = 1
16 − 1 % 12 = 3
⋮

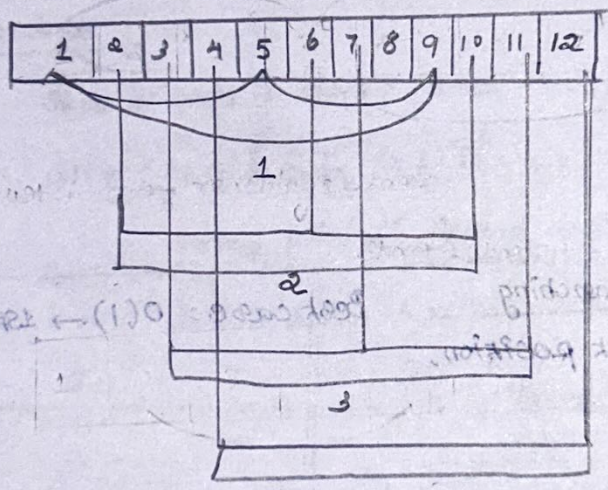
For even

1 2 3 4 5 6

2 % 6 = 2
4 % 6 = 4
6 % 6 = 0 → 5
8 % 6 = 2 → 1
10 % 6 = 4 → 3
12 % 6 = 0 → 5

Not in order! won't work!

Conclusion: wrong algorithm! — order wrong!



stop - when count == n

```
public static void rightshift (int [] array, int n, int num) {
    int count = 0, elem = array[0], temp;
    for (int i = num, start = 0; count != n; ) {
        temp = array[i];
        array[i] = elem;
        if (start == i) {
            start ++;
            i = start + num;
            elem = array[start];
        }
        else {
            i = (i+num) % n;
            elem = temp;
        }
        count ++;
    }
}
```

Reverse method

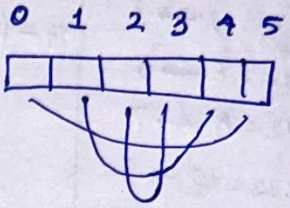
K=2, n=5

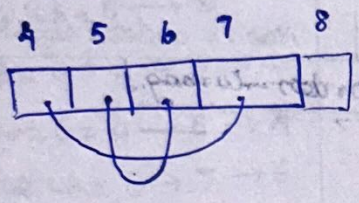
1. Reverse (arr, 0, n-k-1);
2. Reverse (arr, n-k, n-1);
3. Reverse (arr, 0, n-1); → entire array!

Reverse

reverse (array, 0, 5)

$\frac{5+0}{2} = 2$ | $\frac{6+0}{2} = 3$





$\frac{7+4}{2} = 5$ ✓

$\frac{8+4}{2} = 6$ ✓

8+4-4 → 8

8+4-5 → 7

8+4-6 → 6

Searching

Best case: O(1) → 1st position

* Linear search → O(n) → Last position.
* Binary search.

Reverse word

Programming is an art.

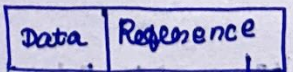
gnimmargorp si na tra.

Linked list

* Linear data structure
* each node → Data → any data type
  → Reference to next node → next node address.

"group - heterogeneous data" → Structure.

every node: Structure. (In C/C++)



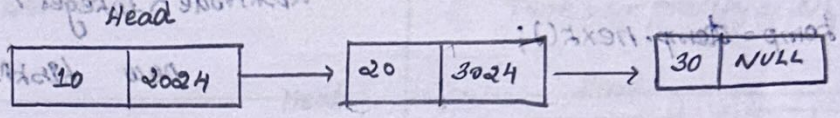
Node → pointer to next node.

```
struct node {
    int data;
    struct node *node;
};
struct node *head, *middle, *last;
head = malloc (sizeof (struct node));
middle = malloc (sizeof (struct node));
last = malloc (sizeof (struct node));
head -> data = 10;
middle -> data = 20;
last -> data = 30;
```

```
head -> node = middle;
middle -> node = last;
last -> node = NULL;
```

"Identify end"

NULL



Print data:

```
struct node *temp = head;
while (temp != NULL){
    printf ("%d", temp -> data);
    temp = temp -> node;
}
```

"C programming"

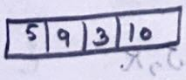
Java

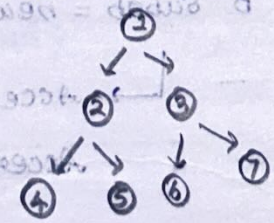
DS: organize data – easy to access & process

DS → Linear
DS → Non linear

Linear: Array, Linked list, stack, queue – single level – continuous memory

Nonlinear: (non sequence) Connected via paths – multi level (Tree, graph)





Time complexity
Space complexity

Time Complexity: Asymptotic analysis: how time & space complexity ↑ as i/p size ↑.

Goal: generic programming

```
class Linked {
    int data;
    Linked next;
}
```

```
class Demo{
    Linked head = new Linked();
    Linked middle = new Linked();
    Linked last = new last();
    head.data = 10;
    middle.data = 20;
    last.data = 30;
    head.next = middle;
    middle.next = last;
    last.next = null;
Linked. temp = head;
    while (temp != null){
        print (temp.data);
        temp = temp.next();
    }
```

Generic

```
public class ListNode <T> {
    private T data;
    Private ListNode <T> next;
}
// Integeer type
```

1) Length of a linked list ?

use

```
ListNode <Integer> obj =
    new ListNode <Integer>
    (15);
```

* Note: generics - only works with reference types

Naming convention:

- T - Type
- K - Key
- N - Number
- V - value
- E - Element

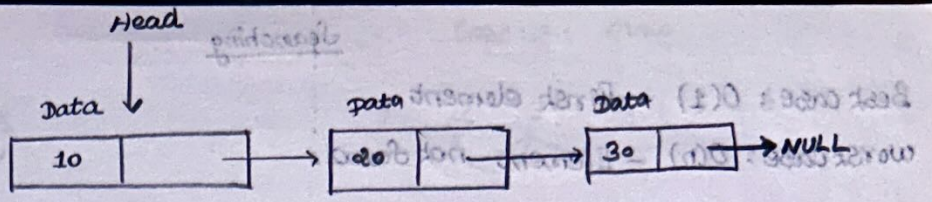
Inheritance Basics

"extends"

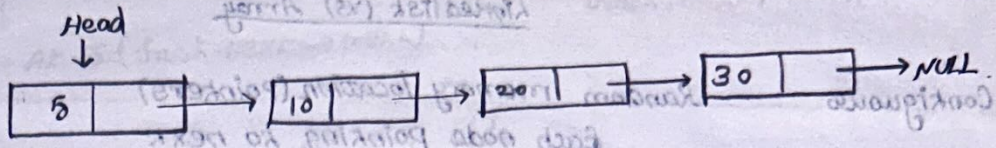
```
class A {
    int i, j;
    void showij () {
        print (i + " " + j);
    }
}
class B extends A {
    int k;
    void showk () {
        print (k);
    }
    Sum (){
        print (i+j+k);
    }
}
```

```
A Superobj = new A();
    └→ Access i, j
B Subob = new B();
    └→ Access i, j, k
    Access showij, showk
```

Add at the Front:



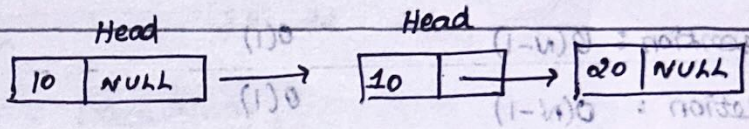
Now:




```
void add first (int val){
    Struct node *newNode = malloc (sizeof (Struct node));
    newNode → data = val;
    newNode → next = head;
    head = newNode;
}
```

Time Complexity: O(1)

Head = Null ⟶

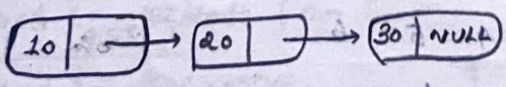


'Add at the End'

```
Node temp = head.getNode();
temp.data = value; temp.next = null;
if (head == NULL)
    return temp;
else {
    Node Samp = head;
    while (Samp.next != NULL){
        Samp = Samp.next;
    }
    Samp.next = temp;
    return head;
}
```

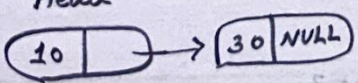
Delete element

Head



Delete 20

Head



Time Complexity

O(1) – Best Case

O(n) – Worst Case.

If → deleting Node is Head

If → deleting Node is NULL.

Searching

Best case : O(1) – First element

Worst case : O(n) – Element not found.

Linked list (vs) Array

* Contiguous – Random memory location (Pointers)
  Each node pointing to next
* Static – Dynamic (memory allocation)
* Stack – Heap
* Fixed size – Expandable / Shrinkable
* Access : Index (O(1)) Traverse (O(n)) : worst case
* Insertion : O(N-1) O(1)
* Deletion : O(N-1) O(1)

Middle node

10 → 20 → 30 → 40 → 50 → NULL 30

10 → 20 → 30 → 40 → NULL 20

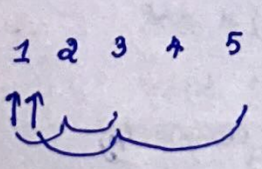
* Find length – divide by 2 – Node at that position returned

Another method

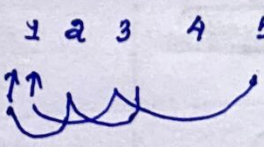
Slow, Fast pointer method

2 pointers – Slow (Slowly moves) – 1 node
Fast (moves Faster) – 2 node

At the end - Slow will be at the middle
Fast will be at the end.



Answer : 3



Answer : 3

This Situation

fast.next == NULL

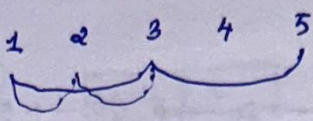
↓

fast = fast.next.next;

↓

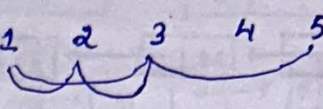
NULL (fast = NULL).

Case 1: odd



This case: At 5 (fast.next → NULL)

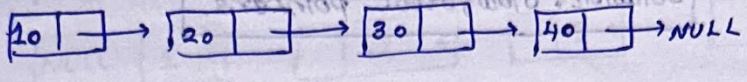
Case: 2: even



fast.next.next → NULL

---

get Nth node



Index : 1 (20)

Index : 3 : (40)

Index : 4 : -1 (Not found).

Time Complexity: O(n) → Worst case. (Index not found)

O(9) → 9(Index) → found.

Reverse Print LinkedList

10 → 20 → 30 → NULL

11 → 22 → 33 → NULL

30 20 10

33 22 11.

Recursively

10 → 20 → 30 → 40

If (temp == null)

return

else

recursive (temp.next);

print (temp.data);

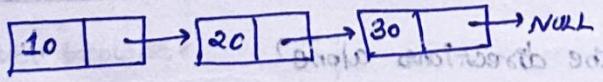
→ disadvantage: stack overflow

Not time eff as compared to Iterative one

'Stack smashing'

---

get Nth Node from the end of Linked list.



10 → 20 → 30 → NULL [N=1 → 30]

10 → 20 → 30 → NULL [N=3 → 10]

[Say: 1 means 3-1 = 2]

method 1: getNode (n - index)

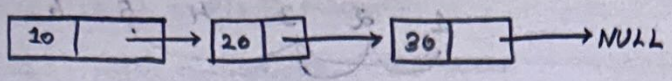
method 2: Recursive.

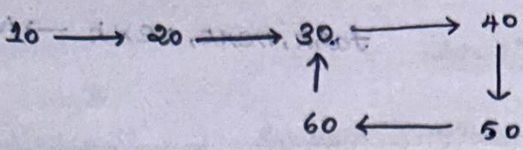
4 3 2 1

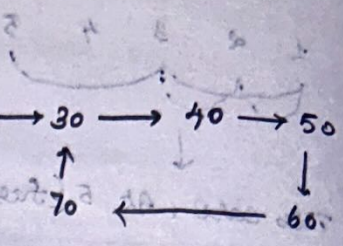
10 20 30 40

```
Static int Count = 1;
If (head == NULL)
        return;
else
    recursive (head.next, index);
    If (Count == index) ——→ print, return
else
    Count++;
```

Is there a loop







Solution: Slow, fast pointers.

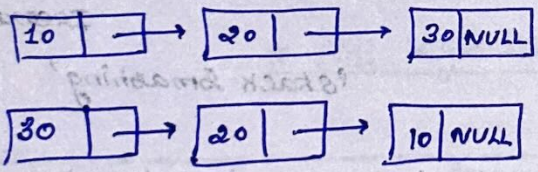
Slow: 1 move
fast: 2 moves

even:	odd:
0, 0	0, 0
1, 2	1, 2
2, 4	2, 4
3, 6	3, 6
4, 4	4, 2
	5, 4
	6, 6

Loop: Slow & fast points to same!

Noloop: fast reaches NULL.

Reverse linked list

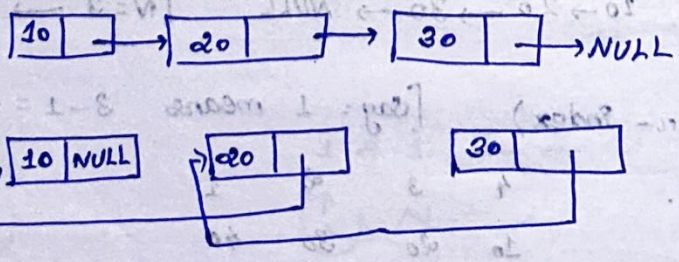


10 → 20 → 30 → NULL
30 → 20 → 10 → NULL

50 → 60 → 70 → NULL
70 → 60 → 50 → NULL

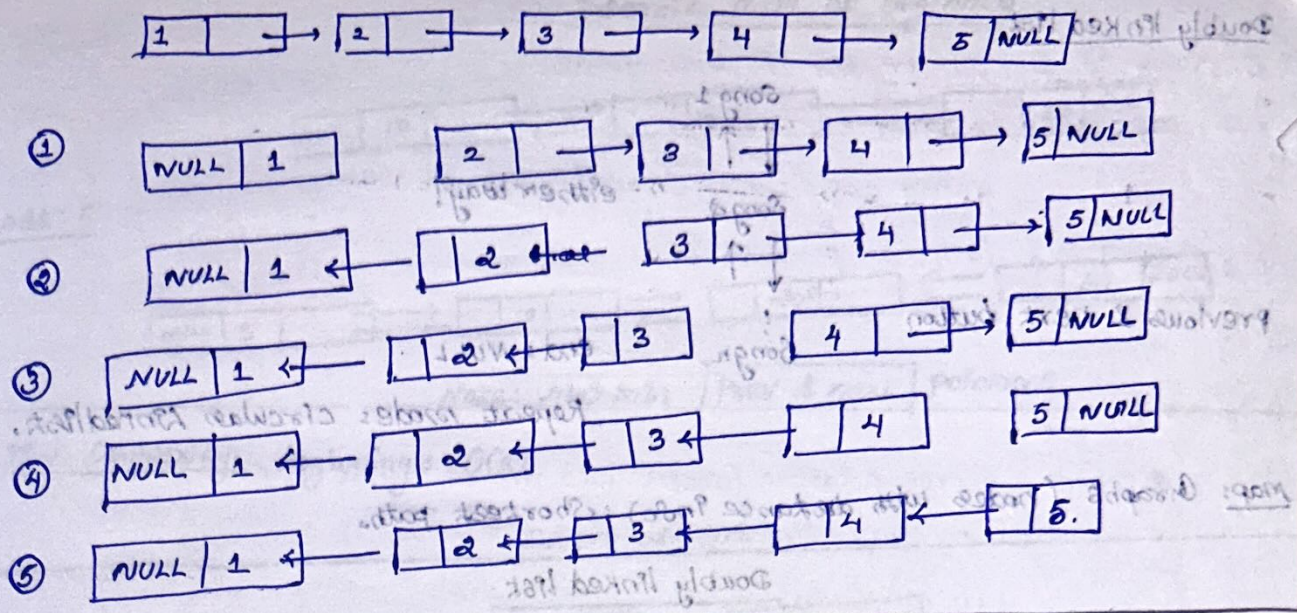
"change the directions alone"

No need to swap data.



★ Starting NULL [current address to next one's address field]

★ current address to next one.



1) Assign NULL, backup the next address to go there before NULL assign.

2) Go to next address, again ① Node address, backup next address.

```
Node *temp = head;
Node *prev = null;
Node hold;
while (temp != NULL) {
    hold = temp.next;
    temp.next = prev;
    prev = temp;
    temp = hold;
}
return prev;
}
```

Why DS?

- ★ web browser
- ★ movie Ticket booking
- ★ Google map.
- ★ Music player

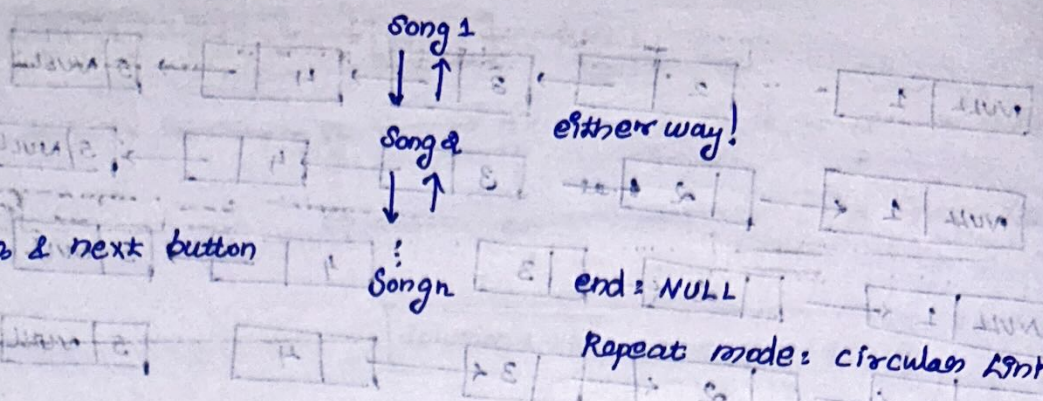
} use DS

- ★ Ticket booking: 2d array.
- ★ Browser: Stack (Back) (Last visited) LIFO
- ★ Queue: counter, (FIFO)
- ★ Linked list: Four Songs (After 1 Song 2 Song 3)

Song 1 → Song 2 → ⋮ → Songn (connected nodes. (Singly linked list)

No backwards ← (only forward.)

Doubly linked list:

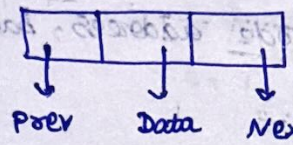


Repeat mode: circular LinkedList.

Map: Graphs (nodes with distance info) : Shortest path.

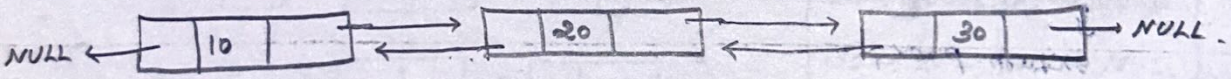
Doubly linked list

* 3 Fields




```
Struct node {
    Struct node *prev;
    int data;
    Struct node *next;
};
```

Bidirectional navigation



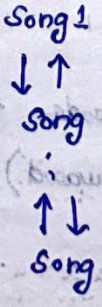
1) Delete, Insert → got easy

* Main disadv: more memory.
* Expensive – maintain 2 pointers always.

Create doubly linked list with data 10,20,30.

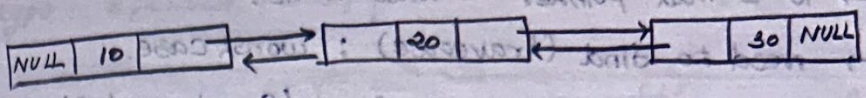
* Forward traversal
* Reverse traversal.

Realtime application (undo, redo) systems

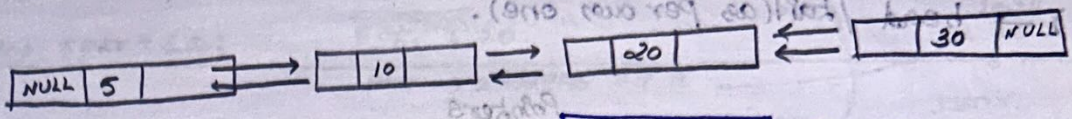


next, prev functions of music / videoplayer.

Insert a Node at beginning



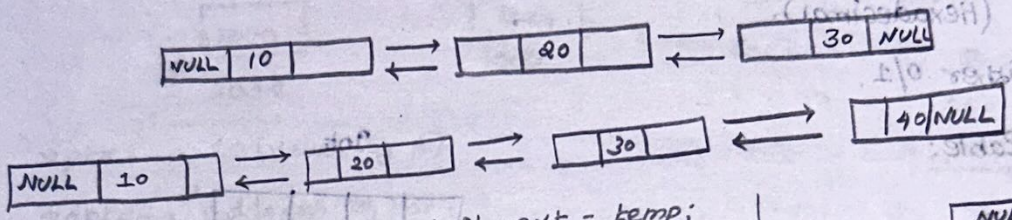
Add : 5



Note: Maintain | Prev & next | pointers.

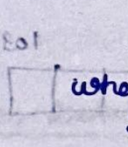
Time Complexity: Beginning : O(1)

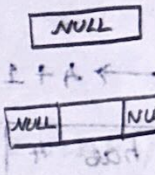
Insert at end



Replace

```
tail.next = temp;
temp.next = null;
temp.prev = tail;
temp.data = value;
return(temp)
```





Can't go to tail.next and change!

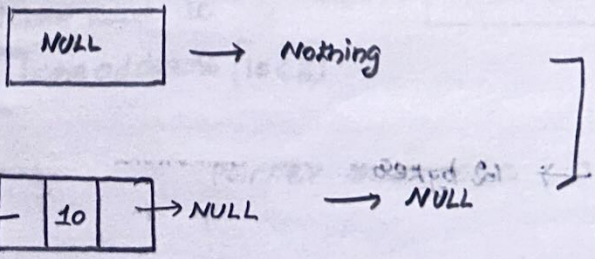
When I Know tail : O(1) → No need to traverse.

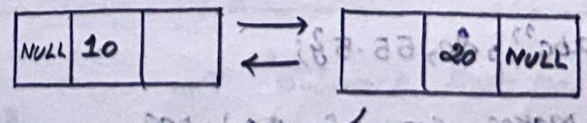
When I don't keep tail pointer : O(n) → each time traverse to get tail!

Search for a node

* Search from head / tail
* Time Complexity: Best case : O(1) – First node
  O(n) – last node / not found.

Delete a doubly linked list node





delete 10 meaning.

head =

return (head.next)

Time Complexity

O(1) → If I had pointer to that node.

O(n) → need to find (traverse) : worst case.

worst case : Not found / found at last

O(1) → head / tail (as per our one).

Pointers

Computer memory:

* Pointer - stores address (access data stored in that location - using pointers)
* Computer memory - linear contiguous nature.

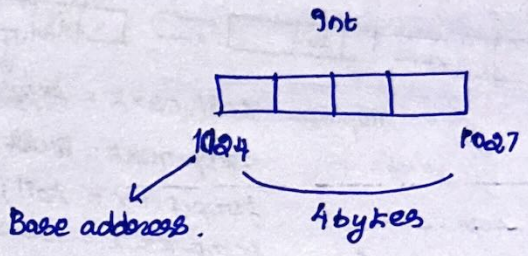
(Hexadecimal)

* Byte = 8 bits — either 0/1.

Address of a variable:

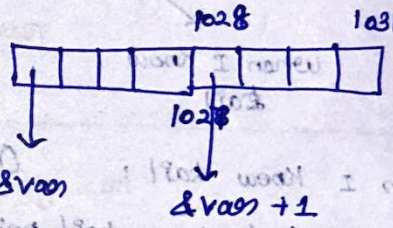
* (&var) + 1 → 4 + 1 = 5.

Say var has 4



&var + 1 → pointer arithmetic →

&var + 5 → arr[5] say in array.



&var + 1 = &var + (1) * size of(var)

Applications: * Embedded devices - Embedded programming

Takes fixed memory → To update status. (use pointer)

"Pointers - really fast"

memory efficient.

```
struct employee {
        char name[4];
        int age;          → 12 bytes
        float weight;
};
struct employee e = {"abc", 20, 55.5};
fun(e); ——————→ makes copy of 12 bytes
```

Fast!

"use pointer - pass address" → save memory.

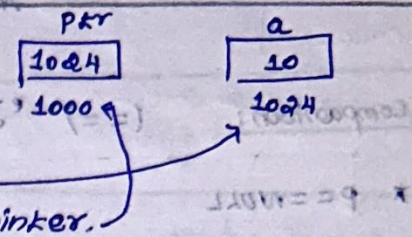
* dynamical memory allocation! (At run time)

datatype *pointer_name;

Single pointers:

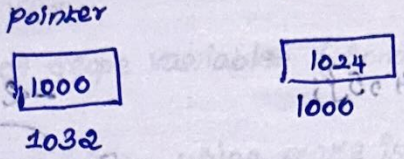
```
int a = 10;
int *ptr = &a;
```

a = 10
*ptr = 10
ptr → address of a
&ptr → address of pointer.



Pointer to pointer:

Double pointer → pointer → variable




```
int a=10;
int *ptr = &a;
int **dptr = &ptr;
```

a → 10
ptr → address of a
dptr → address of ptr

*ptr → 10 (value of a)
*dptr → address of ptr.
**dptr → 10 (value of a)

Call by value (vs) reference:

No change to argument ⟷ Just pass copy (value)
Change (∵ address passed) → call by reference.

Function returning pointer:

return_type function_name (int *, int *)

(&a, &b) say!

int * fun (int *, int *)

```
int * get(){
    int i = 100;
    return (&i);
}
int main(){
    int *ptr = get();
}
```

Sometimes error! (∵ stack destroyed)

Pointer arithmetic:

Increment

```
int *ptr = &i;
ptr++;
```

10 [1024]

*ptr → some value at base address [1028]

Pointer arithmetic → increases by size of datatype of pointer.

'Some'!

ptr = 1024
ptr+1 = 1028
ptr+3 = 1036

'Subtract that no. of blocks' – count!

ptr - 1 = 1020
ptr - 2 = 1016

double - 8 bytes

No pointer → mul, div, Bitwise] operations

Comparison: (==) "same memory location"

* p == NULL
* p == M → Same memory?
* p < q [p pointing to less memory location than q]

Subtract two pointers:

```
arr[5] = {1, 2, 3, 4, 5};
int *p, *q;
p = &arr[0];
q = &arr[2];
```

P - q → -2 → decrement (error)
q - P → +2 → decrement (Some garbage/error).

-2 (Pointer arith)
P - q → [1024 - 1032]
q - p → [1032 - 1024]
→ "Some garbage"
+2 error!

Void pointer: General purpose pointer [doesn't know what it points to]

```
char c = 'a';
int a = 10;
void *ptr = &c;
*ptr = &a;
```

→ Can't dereference
why? : don't know what it points to (type).
char → 1 byte, int → 4 bytes ...

So casting important! (Before dereferencing)!

```
printf("value of q = %d \n", *(int *) ptr);
```

*(char *) ptr

Application:

malloc → Just return void * (generic) - assign it to any datatype.

If no malloc: (generic)

*mallocInt() ..
*mallocFloat() ..

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

→ 4 bytes.

int * | void *

Pointer arithmetic on void pointer: → can't perform until casting!

* Unknown size → std microsoft c
* In gcc: void* → has default size of 1 (can do pointer arithmatic like char*)

Advice: use casting!

Dangling/NULL pointer

* Dangling pointer: pointer points to deleted memory location/freed.
* Doesn't exist.

Ways: 1) out of scope variables (some functions)!

```
int *ptr;
{
  int i = 10;
  ptr = &i;
}
```

i destroyed ← ptr

↓ out of scope

2) Solution: After using make it as NULL

```
{
  int i = 10;
  ptr = &i;
}
ptr = NULL; → Avoid dangling pointers
```

2) Functions! (Return address of local variable)

```
int * get() {
  int i = 100;
  return &i;
}
```

→ dangling pointer! → stack destroyed!

```
void fun() {
  int var = 250;
}
```

```
int *get() {
  int i = 100;
  return &i;
}
```

```
main() {
  int *ptr = get();
  fun();
  *ptr → print
}
```

windows: 250
mac os: 250
linux: segmentation error

But now: windows → error (returning local variable address)

↓

why: Returns recent stack frame!

Deallocating

```
int *p = malloc(8);
free(p); → dangling pointer.
```

Set as NULL

∴ sometimes same memory used by others!

dynamic memory allocation:

* Resize, shrink ↔ Expand.

malloc | calloc | realloc | free.

malloc: (→ static, → dynamic) memory allocation

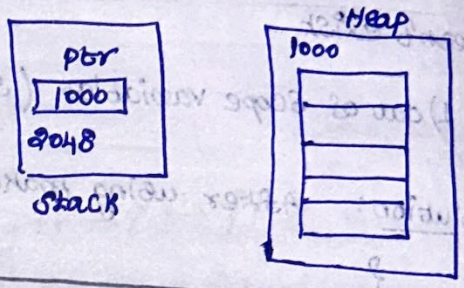
↓

dynamic : heap.

Std lib.h	
malloc	calloc

*ptr = malloc (5 * sizeof(int)); → Returns void * (Implicit Conversion)

Heap full : Returns NULL → Always check for NULL!



---

calloc: "Contiguous allocation"

→ ptr = calloc (5, sizeof(int)); → Initialize with zero.

NULL → Failure

---

malloc vs calloc

* malloc (size in bytes) → No initialize (fast)
* calloc (no. of elem, size of elem) → Initializes with zero

Calloc () = malloc() + memset () → zero.

memset() → fills block of value with particular value.

Realloc — Std lib.h

Resize,

* If the memory area — not created dynamically using malloc / calloc then the behaviour of the realloc is undefined.

↓

Limitation!

realloc (ptr, new size); → more / less than previous!

↳ ptr (need to be resized)

```
char *ptr = malloc(100);
ptr = realloc (ptr, 1000);
```

```
int *ptr;
ptr = malloc (5 * sizeof(int));
ptr = realloc (ptr, 2 * sizeof (int));
ptr = realloc (ptr, 6 * sizeof (int));
```

* unable to expand → new block created ; copy all
* expandable → just resize (no copy)!
* shrink (just shrink).

memory leak

* Heap – won't be deallocated automatically – can't used by other processes
* System crash.

```
free(ptr);
```

using free(ptr); → again → undefined behaviours.

Array & pointer

"more / less same"

10	20	30	40
arr	arr+0	arr+1	arr+2

printing address: %P

print values : *(arr+1);

ptr = arr; ptr can be indexed – if it as an array ptr[0].

use pointer

arr[5] = {1,2,3,4,5}

int *ptr = arr;

*ptr
*(ptr+1)
*(ptr+2)
:

*(ptr+3) = 100;

1d array & pointers

arr
&arr
arr + 1
arr + 2

array is like pointer – not exactly pointer

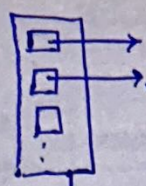
1	2	. . .	7

arr = 0061FF04
&arr = 0061FF04

pointer ptr won't be same as &ptr.

10	20	30	40	50
1000	1004	1008	1012	1016

arr = 1000
&arr = 1000
arr + 1 = 1004 [arr → means address of first value (elem) of array]
&arr + 1 = 1020 [&arr → address of array it self
+1 (means add by size of array (20 bytes)]



→ 1st building aaron

→ 2nd building aaron+1

→ entire building &aaron

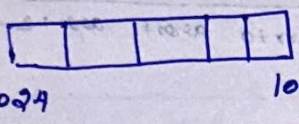
&aaron + 1 → next building.

Pointers to an array & array of pointers

* pointer to an entire array

int (* ptr) [5]; → (array of integers) of size 5.

ptr = &aaron; → must!



ptr = 1024

ptr + 1 → 1044

*ptr goes to array

**ptr goes to 1st element of array

Array of pointers:

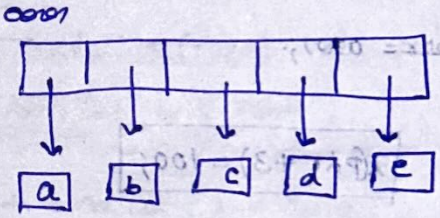
int *aaron [5]; → array of int pointers.

aaron [0] = &a;

aaron [1] = &b;

⋮

aaron [4] = &e;



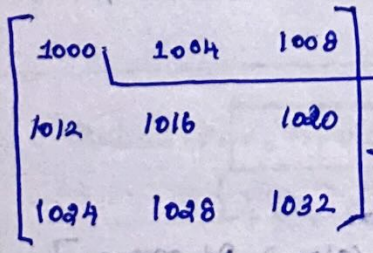
dereference:

*aaron [index] gives value!

∴ aaron [0] means address of a

*(&a) → value of a

2d array & pointers



→ aaron [4 bytes]

→ &aaron [36 bytes]

&aaron → 1000

&aaron + 1 → 1036

&aaron → (int *) [3] [3]

↓

pointer holding int [] [].

aaron → (int *) [3] ⟷ Row alone!

---

aaron[3][3]

*aaron → (row 0, col 0) address

∴ *aaron → pointer to single integer.

∴ *aaron → (int *) [address of 0,0]

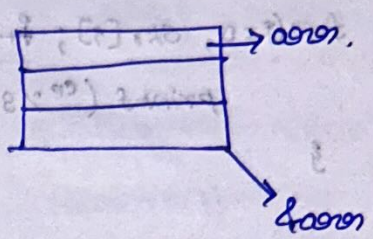
*aaron + 1 → (int *) [address of 0,1]

*(aaron) + 2 → 0,2

**aaron → value of 0,0

**(aaron + 1) → value of 0,1

** aaron + 1 → 0,0 value incremented by 1

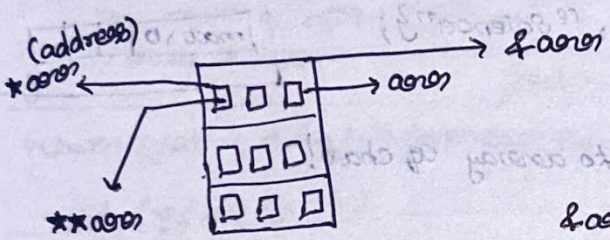


&aaron – 1st 2d array

&aaron + 1 – 2nd "

&aaron → 2d pointer

aaron → 1d pointer [aaron + 1] → next 1d array: row 1.



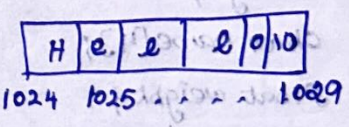
&aaron + 1 → next building!

String & pointer

String → char array with \0 termination

Str[6] = "Hello";

Str → pointer to base address.



Str

1024

Address of each char

Str + i → %p

*(Str + i) → %c (get char.

---

pointer

char *ptr = Str;

Ptr + i → %p

*(ptr + i) → %c

Manipulate: *(ptr + 3) = '0';

HelDO

%s and string

%s and string

printf ("%s", str); → Hello\0

↳ pointer to base address (print upto \0)

printf ("%s", str+1); → ello\0

```
for(i=0; str[i]; i++){
    printf ("%s\n", str+i);
}
```

Hello\0
ello\0
llo\0
lo\0
o \0

Array of pointers to String

char subject [5][20];

subject[0]

H	i	n	d	i	\0	-	-

5 rows, 20 columns

Note: waste of space method: Array of pointers.

char *subject[5] = {"Tamil", ..., "Science"};

↳ Array of char* pointers

pointer pointing to array of char!

only read.

Tamil\0

mat\0

---

print address: %p → subject[i]

%s → subject[i]

Pointer to structure & arrow operator

```
struct employee{
    int age;
    char name[4];
    float weight;
};
```

struct employee e = {"abc", 20, 55.5};

why pointer?

pass to func.! → make copy (not effective)

struct: large!

struct employee * ptr = &e;

(*ptr).name;
(*ptr).age;
] (or)
ptr → name
ptr → age
] dereference then get value.

Passing & return function pointer from function

function pointer: function_name (return type (*ptr_name) (param));

```
func (void (*fp) ( ));
func (int (*fp) (int, int));
```

typedef

```
typedef int (*FP) (int, int);
func (FP fp);
```

```
int sum (int a, int b){
    return (a+b);
}
int execute (int x, int y, FP fp){
    int result = (*fp) (x,y);
    return result;
}
```

Return function pointer

```
main( ){
    execute (10,5, sum);
}
execute (int a, int b, FP fp)
        (or)
    int (*fun) (int, int).
```

```
return type (*fun ()) (parameter)
void (*fun ()) (){
}
```

```
int (*fun() ) (int, int){ }
typedef int (*FP) (int, int);
FP fun(){} ⟶ define function pointer.
```

```
typedef int (*FP) (int,int)
int sum (int a, int b) {
    return (a+b);
}
int mod (int a, int b){
    return (a%b);
}
FP get (char c){
    if (c == '+'){
        return sum;
    }
    else
        return mod;
}
```

```
int main(){
    FP fp;
    fp = get ('+');
    printf ("%d", (*fp)(10,5));
    fp = get ('%');
    printf ("%d", (*fp) (10,5));
}
```

```
typedef int (*FP)(int,int);        Also (correct)

int sum(int a, int b){
    return(a+b);
}
int mod(int a, int b){
    return(a%b);
}                          ⟶ (or) int (*fun)(int, int)
int execute(int x, int y, FP fp){
    return(fp(a,b));        (or) fun(a,b);
}
```

```
typedef int (*FP)(int,int)
FP get(char c){
    if(c=='+'){
        return sum;
    }
    else
        return mod;
}
```

```
int main(){
    FP fp; // can't use fp directly
           since fp is a type!
    printf("%d\n", fp(1,2));
```

↓

3.

```
fn = &sum;
Print(*fn)(12,10);
```

(*fp)(10,5) ⟶ Same as fp(10,5)

```
fun = &sum;
fun(1,2)
```

↓

But use

(*fun)(1,2)

```
fun = sum;
fun(1,2)
```

Without typedef

```
int (*getc(char c))(int,int){
```

⟶ Parameters of return function!

⟶ function parameters

```
}
```

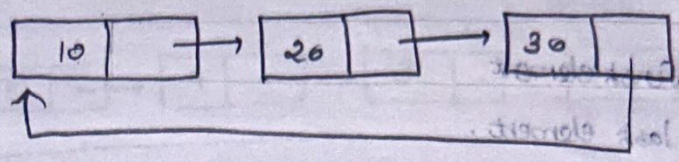
↓

Same as

```
FP getc(char c){ ; }.
```

Function returning struct pointers.

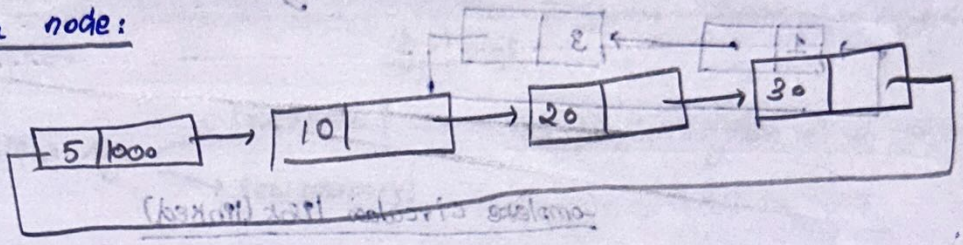
circular linked lists



* Circular singly linked list

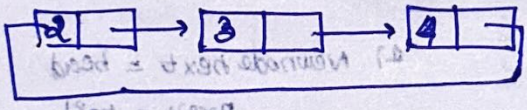
Task: Create circular linked list, print.

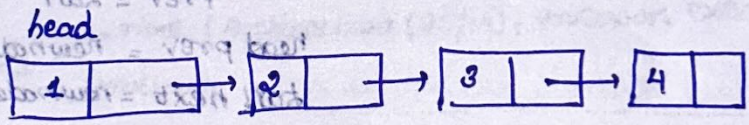
Insert a node:



when end not know: To find end : O(n).

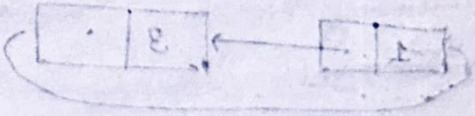
when end known : O(1)





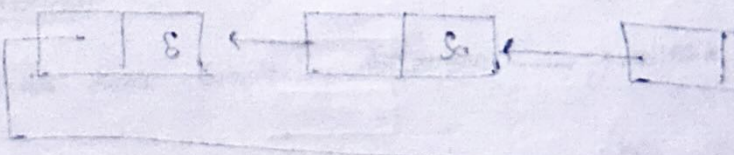
New node: (first index)

1) Alter head
2) New node (prev points to tail)
3) New node next points to previous head
4) Alter previous head's (prev)!



Add at 0th index → Time complexity: O(1)

∴ we have tail (no need traversal)



Insert at end

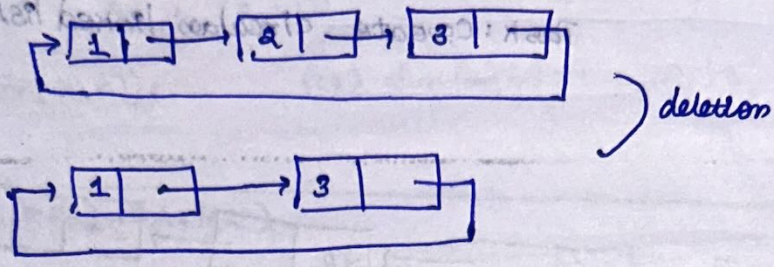
O(1) → we have tail!

Search : best case : O(1) – first element !

worst case : O(n) – last node !

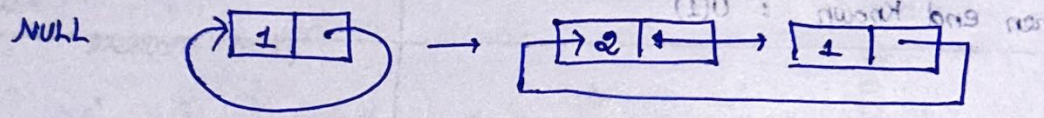
Delete : O(1) – first element

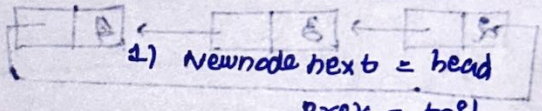
O(n) – last element.



Complete circular link (linked)

Create – Add at last



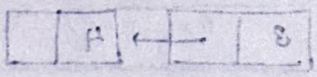


1) Newnode next = head

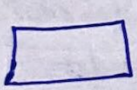
prev = tail

head prev = newnode

tail next = newnode.

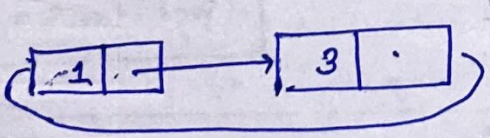


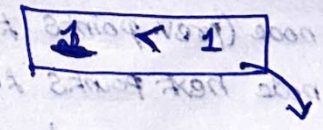
1)



Index : 2 [Join Index at end]

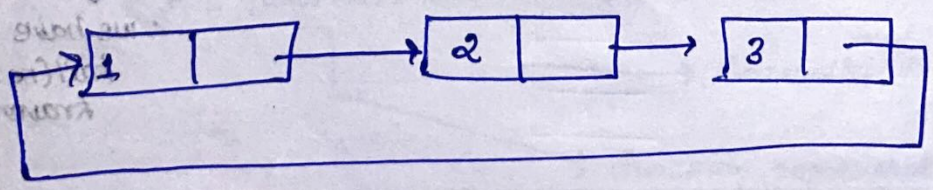
2)



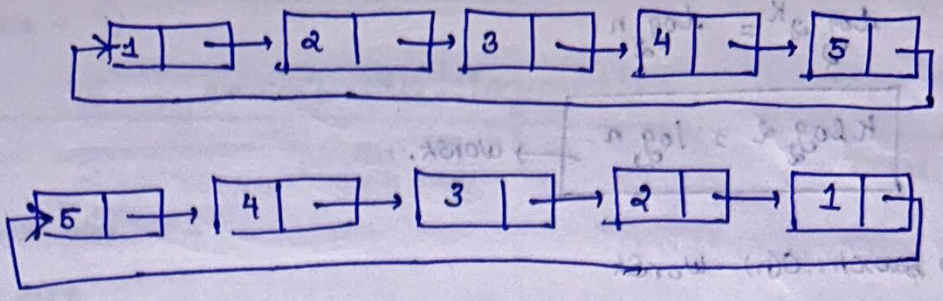


Now we have Index 1.

Join at 1 means



1 < 1



go to prev (changed).

1 prev = next

1 next = prev

Algorithms

Efficient Algo → low space!
→ low memory!

→ simple english

Pseudocode: no code syntax.

```
Swap (a,b)
{
  temp ← a
  a ← b
  b ← temp
}
```

Analyze: → time (Architecture (32/64), processor, execution, memory access time)
→ memory

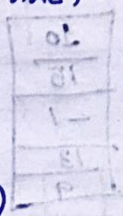
given i/p

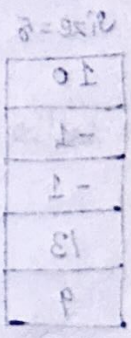
What we take:

Read, write, return, assignment → 1 unit of time.

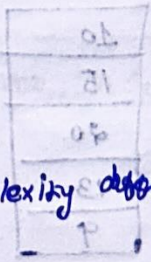
k(x) = 3

(O(1))





O
Θ
Ω



Space & Time Complexities

Small time Complexity difference → greatest impact.

Binary search → Best case: O(1) [Key: middle element]
→ Worst case: $n/2^0, \ldots, n/2^3, \ldots, 1$

$\frac{n}{2^k} = 1$

$K = \log_2 n$

$\log_2 2^K = \log_2 n$

$K \log_2 2 = \log_2 n$ → worst.

Linear search: O(n) – worst

O(1) – best (first element)

Hashing & Collision

* Index page [normal: O(n)] → Linear Search Binary Search: $\log n$

= 3! (worst) 8 elements

Can I make it in Constant time:

Place at exact array index.

f(2) = 2 f(6) = 6	f(x) = x

*place waste.

Number % N (0 to N-1) → Initialize (-1) → not present.

Collision:

* 15 % 7 = 1 [Index element has a number already]

* Avoid collision: Linear probing (closed hashing)

Size = 5

10
-1
-1
13
9

15 % 5 = 0

Already there:

Next index

10
15
-1
13
9

20 % 5 = 0

10
15
20
13
9

Linear Probing:

$f(x) = (x+1) \% size$ [1 = 1, 2, ...]

(Data + 4) → means OK

5 → we can't fit! (anymore!)

Hash table

Initialize with −1

Data % size

Collision: Linear probing!

Linear probing – done ✓

Quadratic probing:

disadvantage: Collision – consecutive elements will fall clustered

Look for a slot!

hard to find next avail index, Search, retrieve.

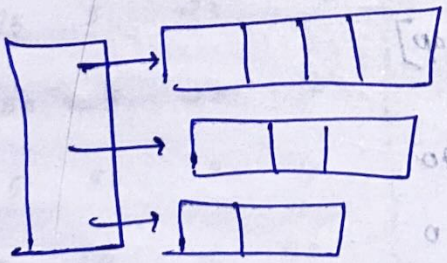
Instead of immediate next slot.

Quadratic probing

$f(x) = (x + i^2)$ % size i = 1, 2, 3, 4, etc..

Linear probing / closed hashing

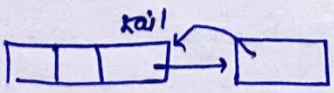
open hashing / seperate chaining → use linked list



Add new node to linked list

---

* In C: Whenever you do – malloc → do free() ✓



Selection Sort

180, 165, 150, 170, 145

swap

165, 180, 150, 170, 145

150, 180, 165, 170, 145

145, 180, 165, 170, 150.

145, 180, 165, 170, 150

145, 165, 180, 170, 150

145, 150, 180, 170, 165

145, 150, 165, 180, 180

$1+2+3+\cdots+N-1$

$O(n^2) \rightarrow \frac{n(n-1)}{2} = \frac{(n^2-n)}{2}$

Inplace $O(1)$

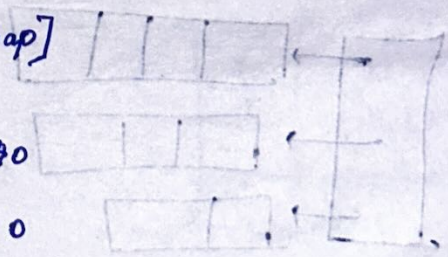
$O(N) \rightarrow$ for array alone — No extra Space!

Bubble Sort

* Similar to movement of bubbles in water
  - Largest bubbles — rise first
  - Smaller one — follows it.

each iteration: largest/smallest gets its actual place.

* Compare adjacent bubbles [largest: swap]



50 20 30 10 40

20 50 30 10 40

20 30 50 10 40

20 30 10 50 40

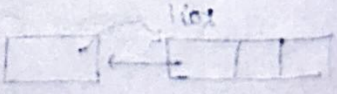
20 30 10 40 50

20 30 10 40

20 30 10 40

20 10 30 40

10 20 30



10 20

50 25 5 20 10

25 50 5 20 10

25 5 50 20 10

25 5 20 50 10

25 5 20 10 | 50

5 25 20 10 | 50

5 20 25 10 | 50

5 20 10 | 25 50

5 10 | 20 25 50

5 10 20 25 50

```
for (i=0; i<n; i++) {
    for (j=0; j<n-i-1; j++){
        if (array[j] > array[j+1]){
            swap
        }
    }
}
```

---

50 25 5 20 10

25 50 5 20 10

25 5 50 20 10

25 5 20 50 10

5 25 20 10 50

5 20 25 10 50

5 20 10 25 50

5 10 20 25 50

when no swap ⟶ Sorted!

10 20 30 40 50

1 ⟷ 2
2 ⟷ 3
3 ⟷ 4
4 ⟷ 5

↳ Fail: No need to swap!

So make a flag! ⟶ Cut unnecessary work!

Best case: Already sorted : $O(n-1) \approx O(n)$ [only 1 is enough

false = flag
↓
end]

Worst case:

$4+3+2+1 = 10$

$\frac{n(n-1)}{2} \approx O(n^2)$

Space Complexity: $O(N)$ — for storing

No extra space.

Insertion Sort

* Efficient: Sorting small no. of elements. (Similar: arrange plain cards)
* Assume: first card — in correct position
* Smaller: above
  Larger: below.

1) Assume correct

40 10 50 30 20

10 40 50 30 20

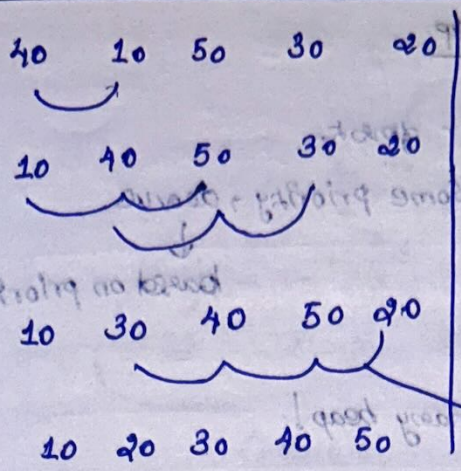
10 40 50 30 40

10 40 30 50 20

10 30 40 50 20.

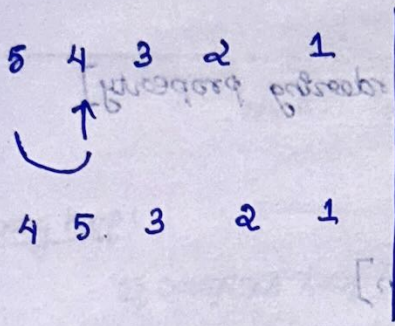
Rearrange.

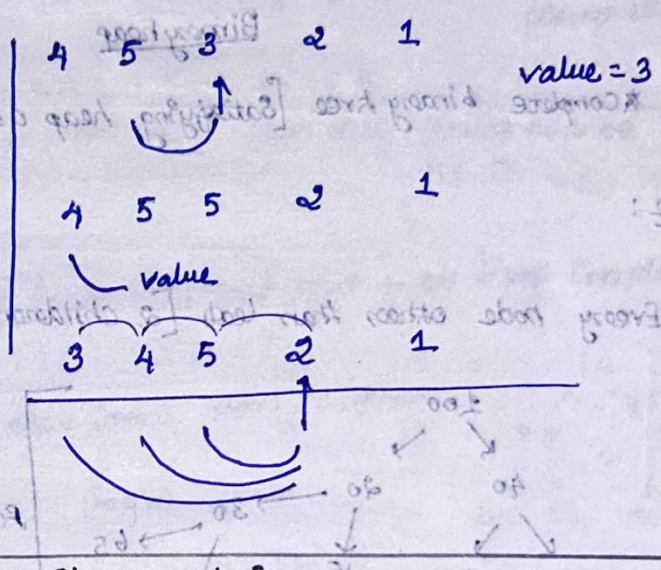
10 20 30 40 50




```
for(int i=0; i<size; i++){
    if(
}
```

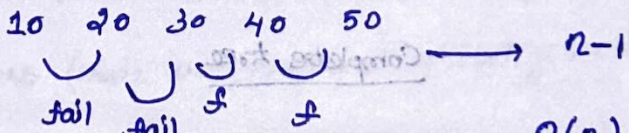
until a small number is detected!





Time Complexity

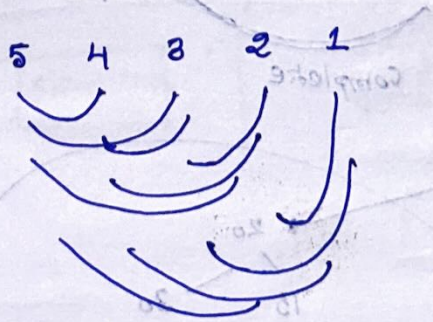
Best case: Sorted



$\longrightarrow n-1$

$O(n)$

Worst case: Reverse Sorted



$1+2+\dots+N-1$

$1+2+3+4 = \frac{n(n-1)}{2}$

$O(n^2)$

No extra space: $O(1)$

Binary heap

queue: person comes first → sees doctor first.

serious: priority! (priority queue) → some priority, dequeue

↓

based on priority!

Efficiently implement priority queue

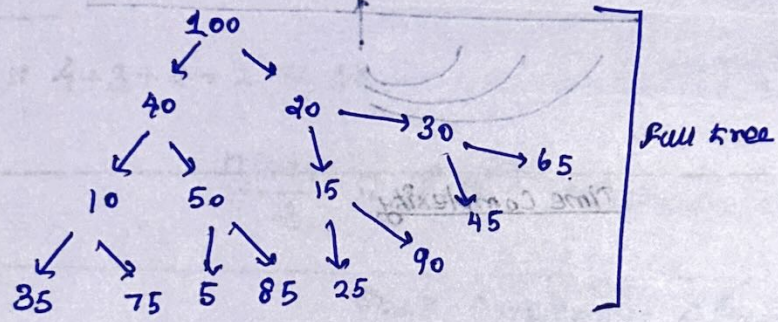
↳ using binary heap!

Binary heap

* Complete binary tree [satisfying heap ordering property]

Full tree:

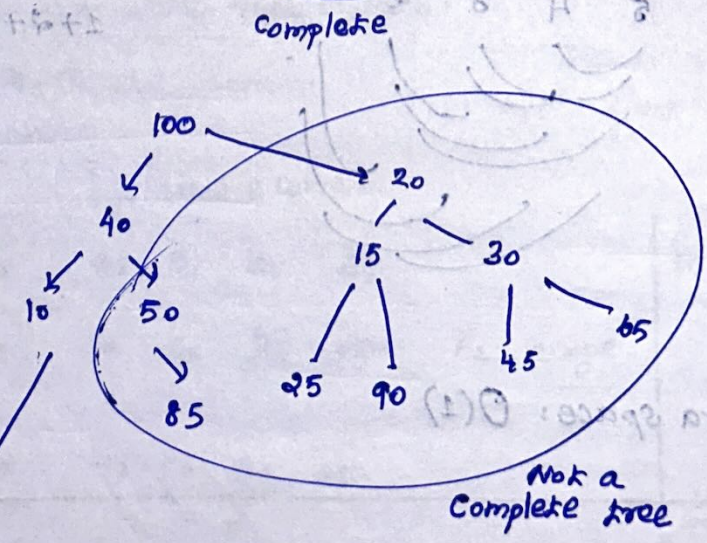
* Every node either has leaf [2 children]



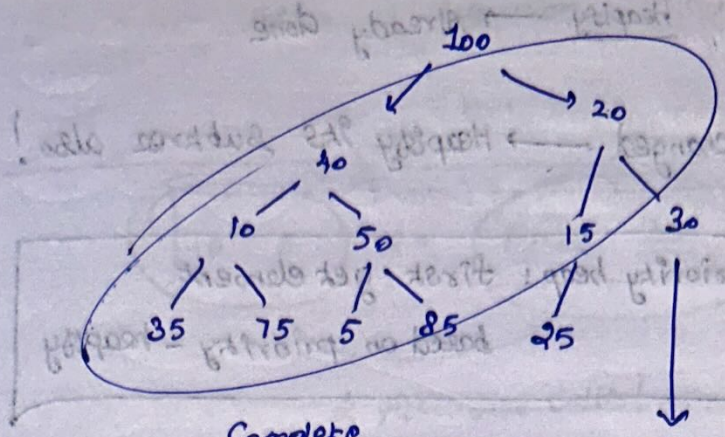
Complete tree

* Full tree having (last level) — may have less no. nodes

1) As left as possible / ~~as right as possible~~



Left side incomplete!



Complete

Only right incomplete ✓

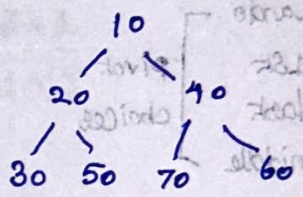
(Complete tree).

Note: Complete tree = last level (must not be vacant).

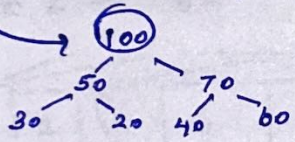
Binary heap:

1) Complete heap! (other than last level – all level Complete).

min heap ⟶ every parent (min) than children



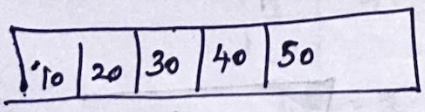
Max heap ⟶ (max) "

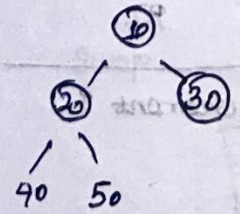


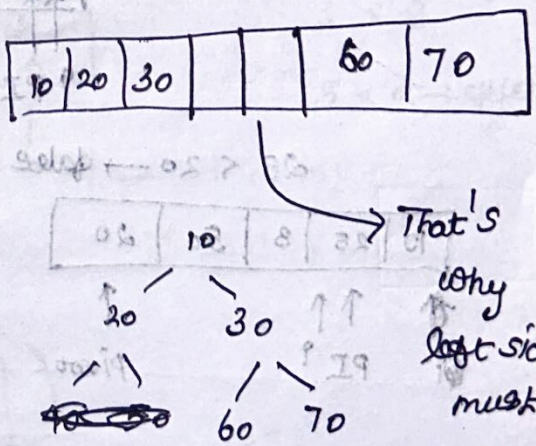
min heap: Access (node with minimum priority node at constant time)

max heap: Access (node with maximum priority node at " ⟶)

Why left side must be complete







That's why left side must be filled

Array – will be Consistent!

Heapify → Already done

Note: If a parent - changed → Heapify its subtree also!

How priority heap: first get element
based on priority - heapify

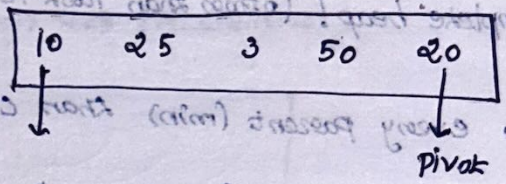
Quicksort

★ Divide & Conquer Algorithm

★ pivot - element selection

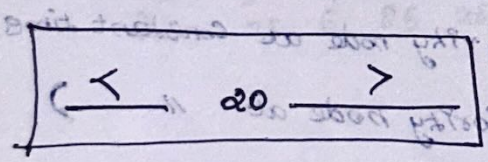
★ Partition

Random
1st
last
middle
} pivot choices

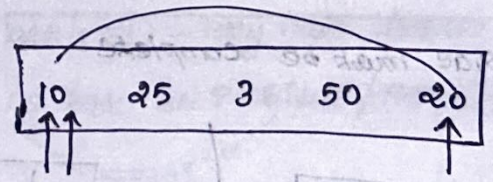


① Pivot

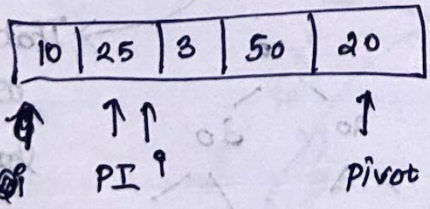
② Partition



10 < 20 (swap i and PI)



25 < 20 → false



swap then
increment
PI

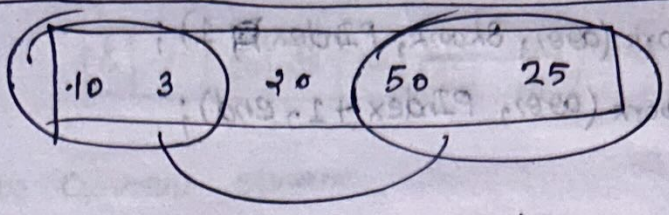
increment i

false → PI can't be incremented

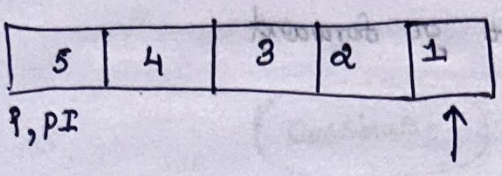
increment i.

3 < 20 → swap array[i] & array[PI]

After 1 iteration → swap array[pivot], array[pI]

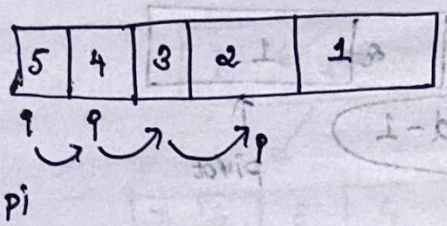


2 recursive calls!



array[i] < pivot

false

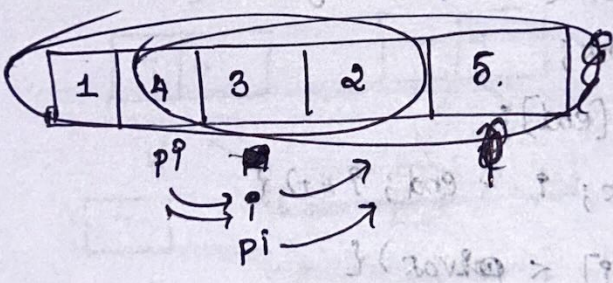


false 4 < 1

false 3 < 1

false 2 < 1

Swap pivot & array[pi]

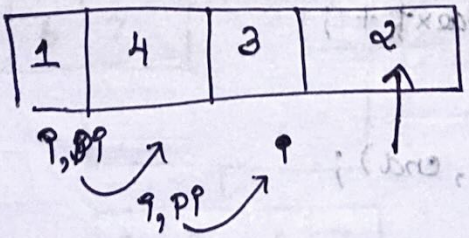


4 < 5 → false

3 < 5 → yes

2 < 5 → yes

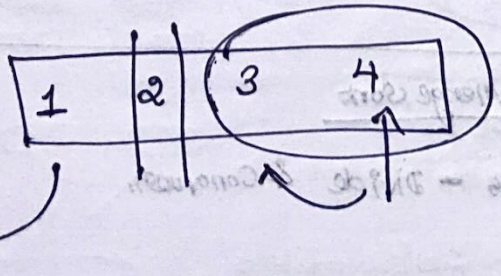
~~Swap pivot & array[pi]~~



1 < 2.

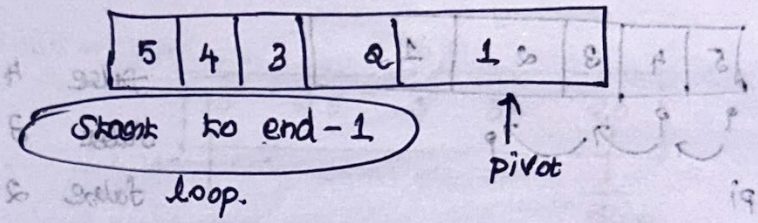
3 < 2 → false.

Swap.



```
public static void quickSort(int[] array, int start, int end){
    PIndex = partition(arr, start, end)
        quickSort(arr, start, PIndex - 1);
        quickSort(arr, PIndex + 1, end);
}
```

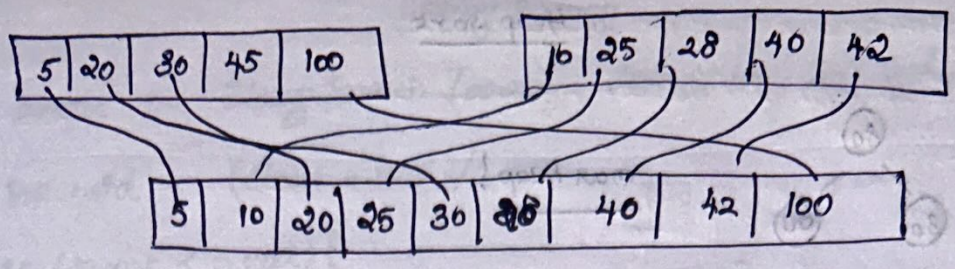
```
public static int partition(int[] array, int start, int end){
    if (start < end) ⟶ don't go forward
}
```




```
public static int partition(int [] array, int start, int end){
    int pIndex = start;
    int pivot = arr[end];
    for (int i = start; i < end; i++){
        if (arr[i] < pivot){
            swap(arr, i, pIndex);
            pIndex++;
        }
    }
    swap(arr, pIndex, end);
    return (pIndex);
}
```

Merge sort

* Sorting two sorted arrays - Divide & Conquer.



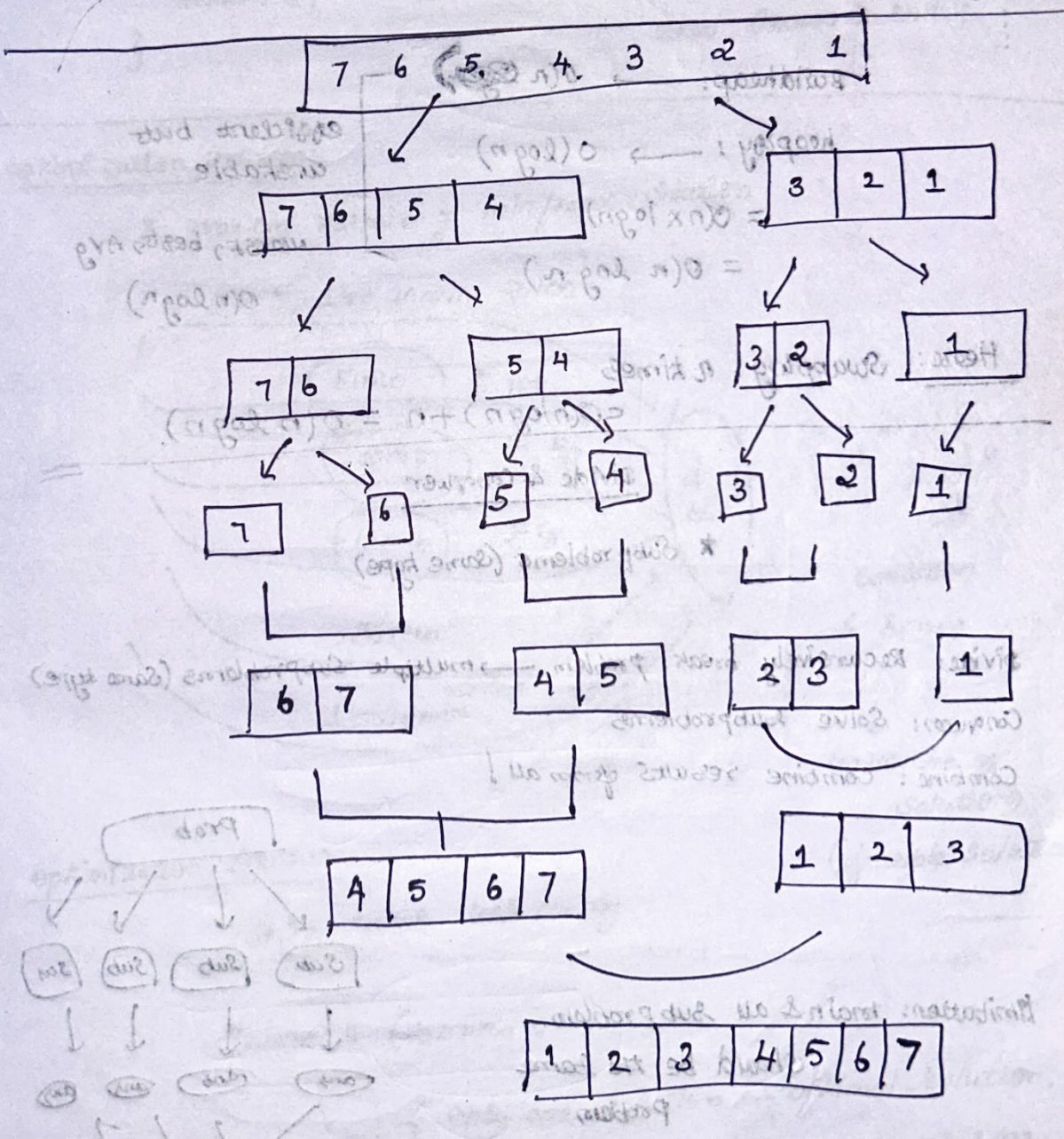
Compare Current element

1 array array[i] < array[j]

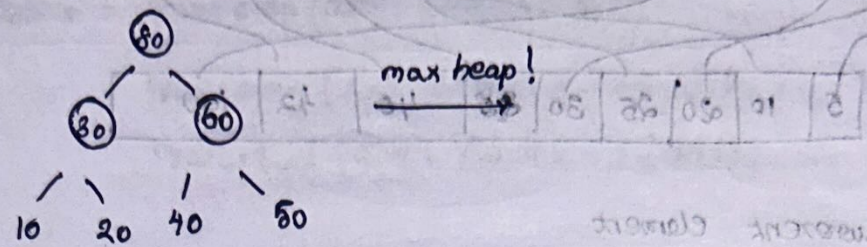
insert i

i++

Continue!



Heap sort



* Swap 1st & last element. (reduce array size by 1)
* Heapify
* Swap 1st & last element
* Heapify
⋮

Buildheap: ⟶ O(n)

heapify: ⟶ O(log n)

= O(n × log n)

= O(n log n)

efficient but unstable

worst, best, Avg

O(n log n)

Heap: Swapping: n times

= O(n log n) + n = O(n log n).

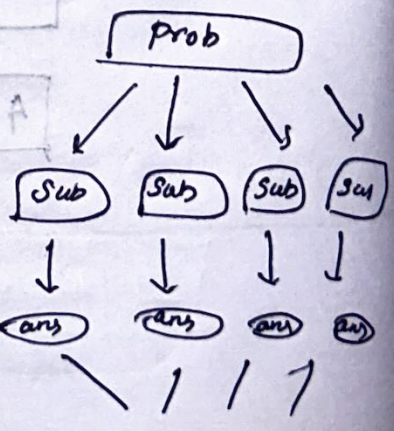
Divide & Conquer

* Subproblems (same type)

Divide: Recursively break problem ⟶ multiple subproblems (same type)

Conquer: Solve subproblems

Combine: Combine results from all!



limitation: main & all subproblem should be the same problem

main problem: Searching - all subproblems searching

Sorting - all subproblems sorting.

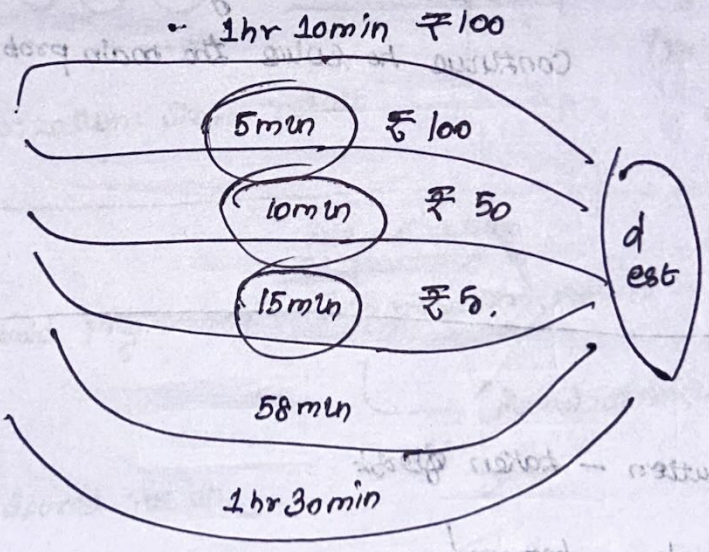
Binary Search

```
public static int BinarySearch (array, int start, int end, int key){
    int mid = (start + end)/2;
    if (start <= end){
        if (arr[mid] == key)
            return 1;
        .
        .
        .
    }
    else {
        return 0;
    }
```

→ Not found when start > end !

Optimization problem:

* problem requiring min/max solution



Condition
≤ 45min

↓

lesser no. of solutions
(feasible solutions)

Optimization Solution:

which takes less money

Route: 4 (15min, ₹5)

↳ Only one solution — optimal solution.
(feasible solution giving max/min result).

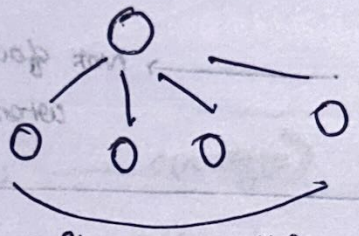
* more than one solution — feasible
* But 1 optimal solutions.

1. Greedy
2. Dynamic
3. Branch & Bound

Dynamic programming

1. Basic method — Solve optimization problem in linear time

Problem:



Store results

Use the stored results

to solve other big subproblem

Continue to solve the main problem.

* Bottom up
* Top down

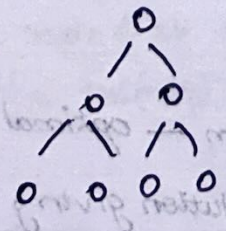
Bottom up:

* Base solution — taken first

from bottom!

Top down:

main problem — broken — baseCase reached (problem solved)



Nth fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21 . . . .

Fib(N) = Fib(N-1) + Fib(N-2).

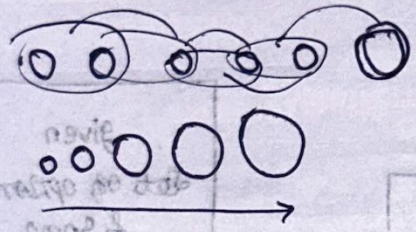
Bottom up

Base case: Fib(0) = 0
Fib(1) = 1

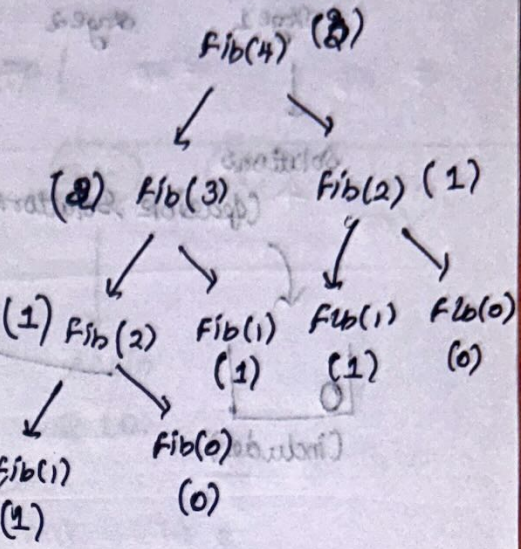
```
if (n==0 || n==1){
    return 1
}
else :
```

fib(0) fib(1) . fib(4)

0	1	1	2	3



Top down



many repitation
(In Top down approach)

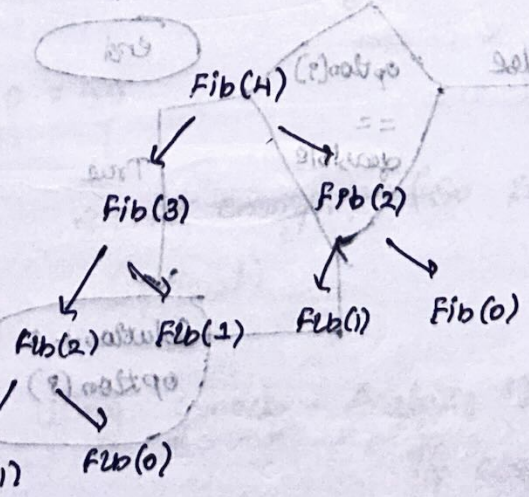
* Memoization: Store result

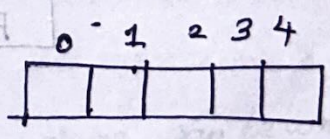
Memoization

In dynamic programming: Recomputation issues in top down approach

→ Memoization (used)

* Use Stored result.



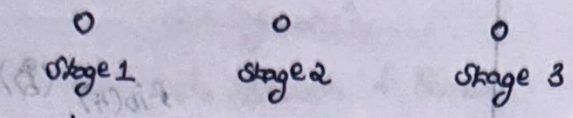


Check there?
(else – go for recursive call).

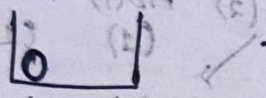
Implement — Use array

Greedy method

★ Problem solved – multiple stages

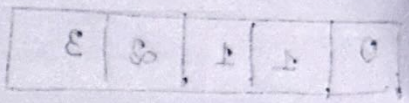


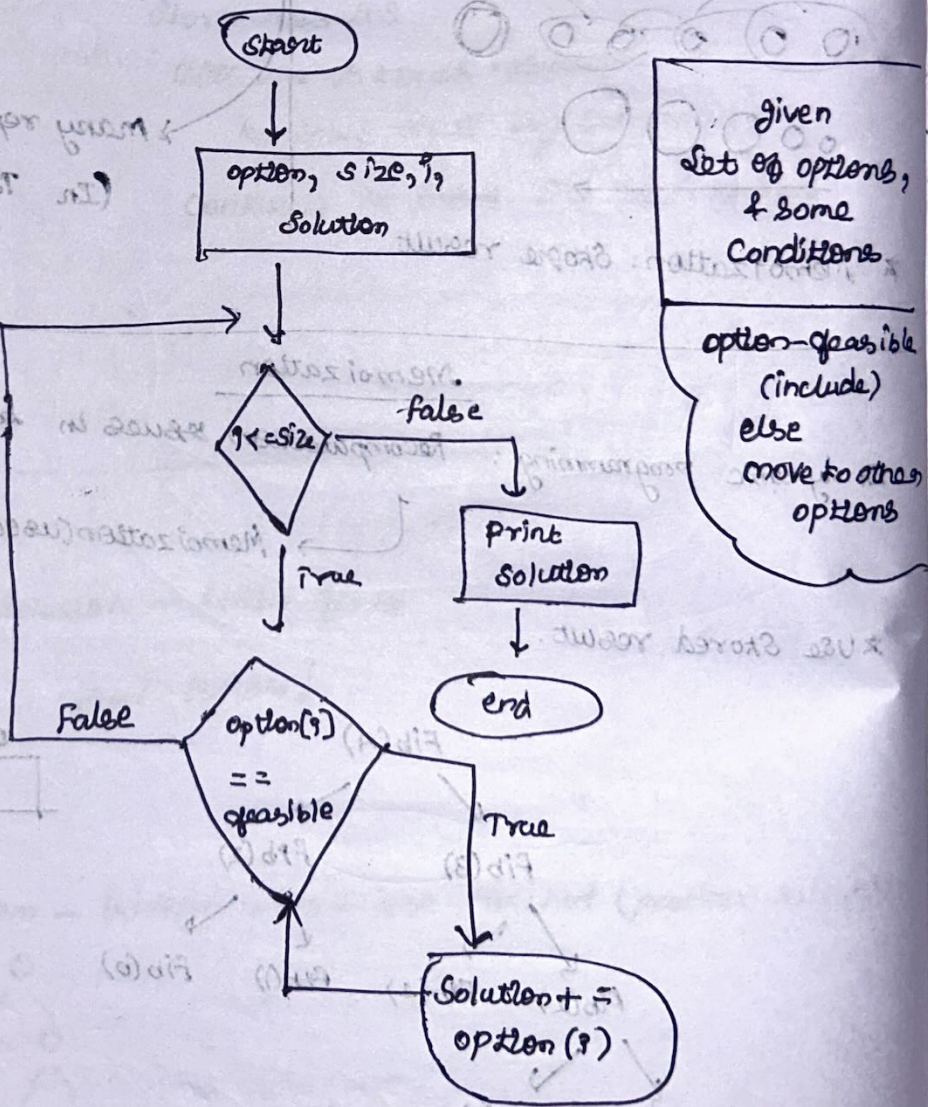
Solutions
(feasible solutions)



From feasible solutions

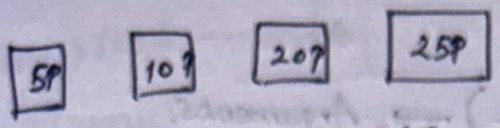
we get optimal solutions.





overall optimal results – achieved by choosing feasible options in each stage.

min Coin exchange problem



Find min no. of coins - satisfy given value.

₹50 → (25 × 4) × 50 = 50 rupees.

200 → coins from ₹25P!

₹110

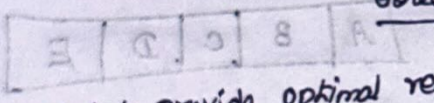
₹25 ₹20 ₹10 ₹5

4 coins ₹100

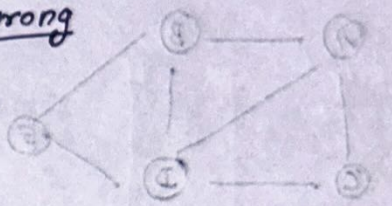
1 Coin ₹10.

5 coins!

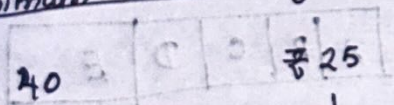
Greedy can be wrong



*may not provide optimal results always.



minimum coin exchange:



₹40 ₹25 ₹20 ₹10 ₹5

25 10 5

3.

Now: 20 + 20 = 40

↓

2 coins enough! (So sometimes can be wrong)

(optimal)

Before: using greedy - Analyze pt for all possible o/p cases

↓

else we should use another approach.

Implement – minimum coin exchange

* may be wrong for some i/p s.

(coins array, size, value) ⟶ Arguments.

↳ sorted.

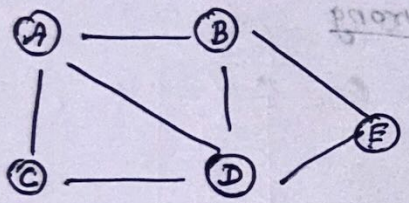
₹ 25 ₹ 20 ₹ 10 ₹ 5. ← ₹110

$110/25 = 4$

Breadth Search Algorithm

Traverse graphs:

* BFS – use queue



visited

A	B	C	D	E

queue

A	B	C	D	E

A, A's neighbours

1) is queue Empty

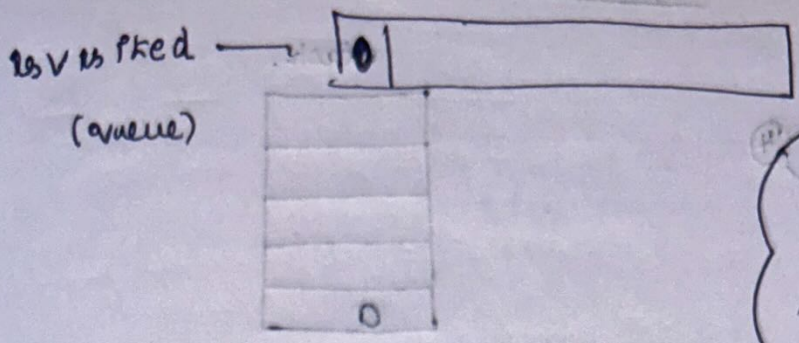
2) dequeue ⟶ First one first out

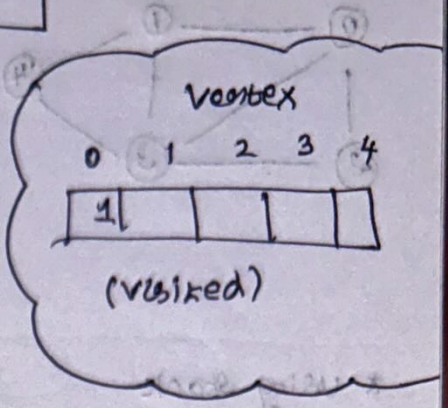
3) Enqueue ⟶ Insert element

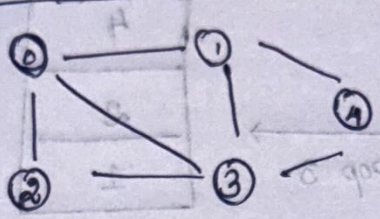
4) simply: queue.h #include

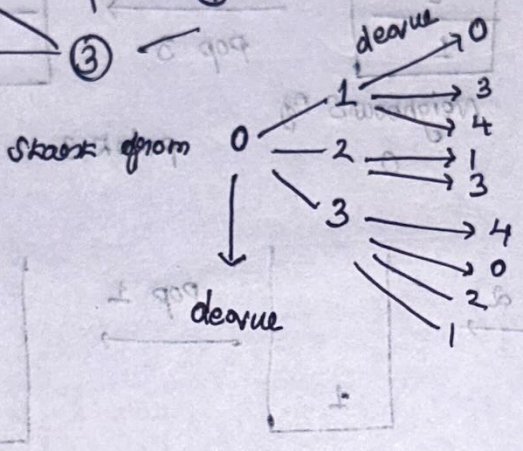
```
Int arr[5][5] = {
                {0, 1, 1, 1, 0},
                {1, 0, 0, 1, 1},
                {1, 0, 0, 1, 0},
                {1, 1, 1, 0, 1},
                {0, 1, 0, 1, 0}
```

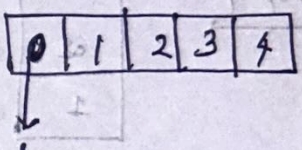
bfs (arr, 0);











use of this approach:

when queue is empty

↳ means traversal done!

(or)

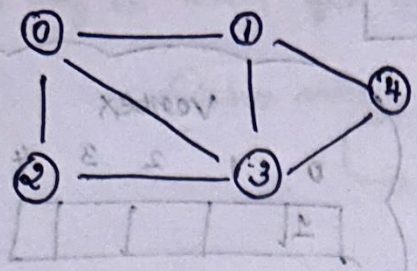
go to all → matrix → n^2

$O(V+E)$

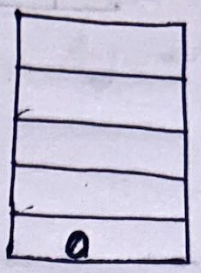
If we didn't use: visited array → Traversal → $O(n)$

Graph → chance of ∞ loop!

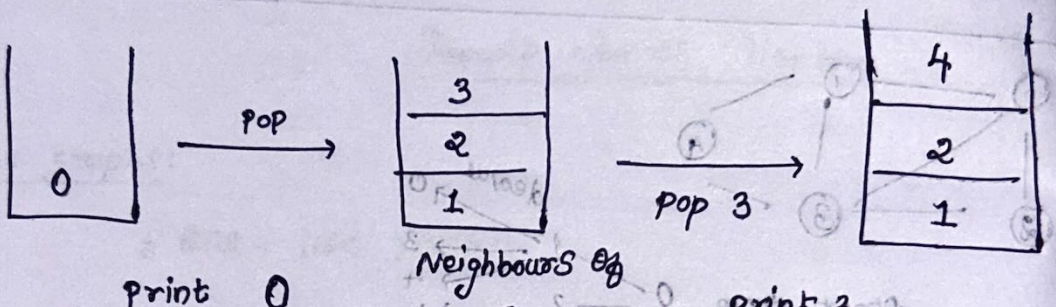
Depth first Search



Stack.



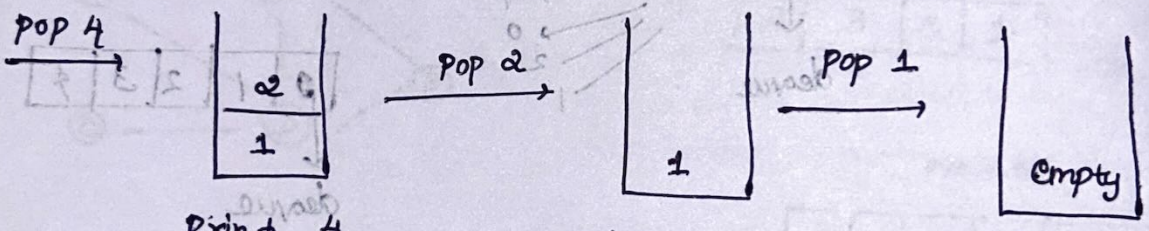
* Using stack
* Same procedure.



print 0

Neighbours of 0

print 3



print 4

print 2

print 1

∴ only push if already not visited!