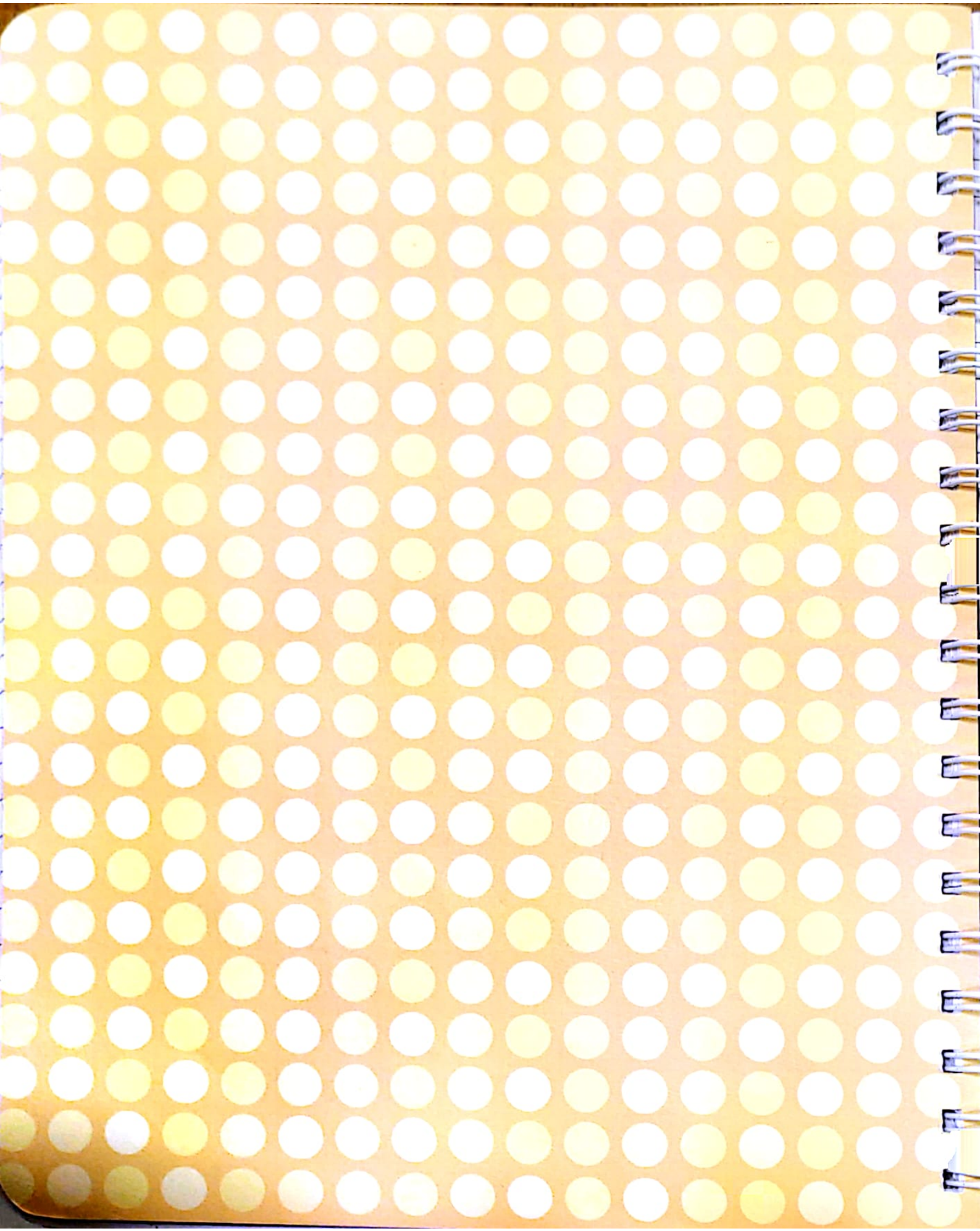


Data Structures

- Combination of 4 bits ⇒ nipple
- 2^x different things can be given unique binary code using 'x' bits

2's compliment notation

unsigned int works on this funda.

9 ⟶ 0000 1001

⇒ −9 — take compliment of all + add 1

1111 0110

+1

1111 0111 ⟶ This is '−9' in 2's compliment notation.

So for −0 ?

0000 0000

destroyed (1) 1111 1111

+1

0000 0000

Most significant bit is 0 ⟶ +ve number

1 ⟶ −ve number.

Range of 8 bit signed number.

-2^7 to $2^7 - 1$

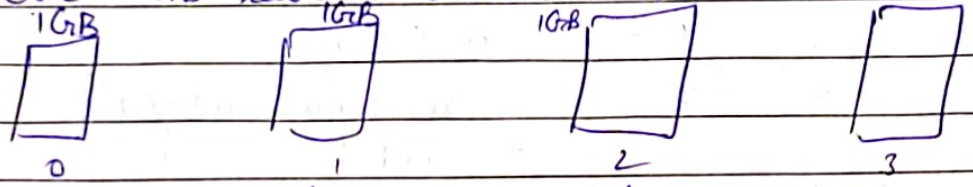
0111 1111 ⟶ 127.

1000 0000 ⟶ −128.

see it from 2's complement notation

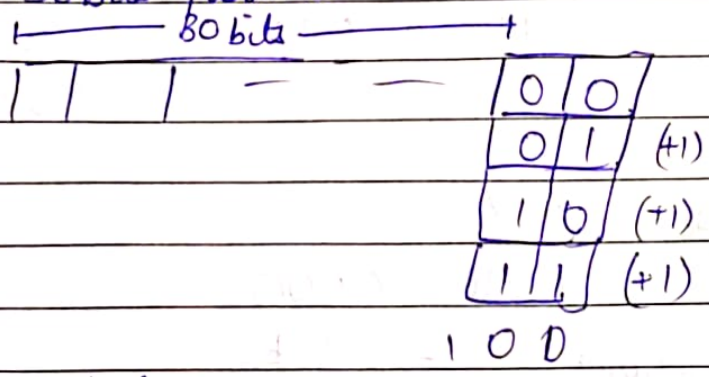
Q. Why Padding is impt? (32 bit).

Our 4GB ram is divided into 4 banks. 1GB.



Now when address starts with 00 it takes

30 bits from 0



If we start from bank 1, on 4th byte address changes so we need to manually change that. This causes delay. Thats why padding is needed.

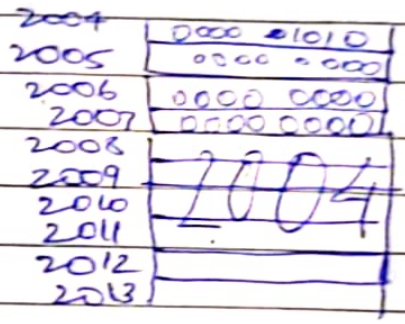
Q.

Intial intel processors, 8080, 8085, etc are ISA.

__/__/____
1010

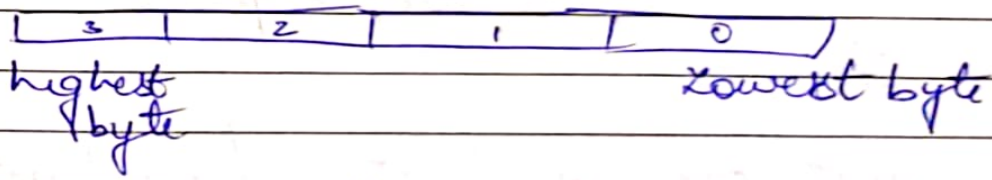
Local variables → stack memory.
Global variable → Heap memory.

```
int x;
int *xptr;
  x = 10;
  xptr = &x;
int **aptr = &xptr;
```



Little Endian

Lowest byte is stored at lower address.
eg for x - (32bit)
10 = 00 — — — — 0000 1010



⇒ 2004 gets 0000 1010

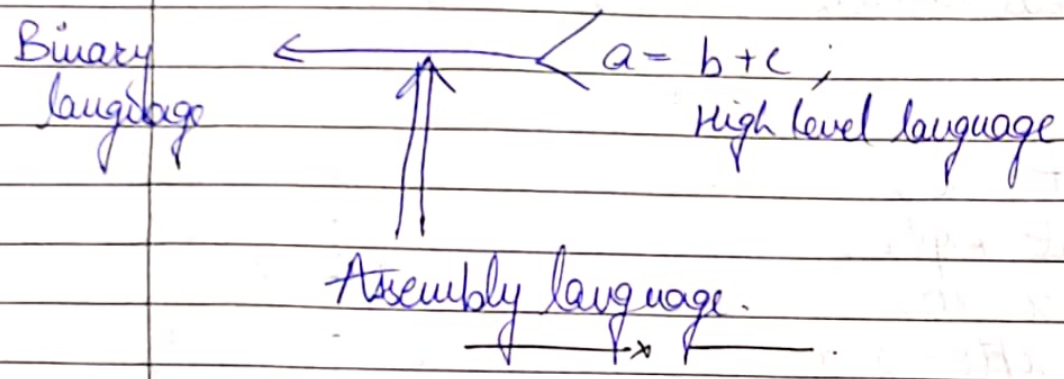
Big Endian

higher order byte is stored at lower address.
Internet follows this.

Why do we use pointers?
Pointers helps in indirect addressing. (Dynamic memory allocation).

__/__/____

Extra.



Self referential Structure.

A Structure which has a pointer to same structure as a data type i.e. a structure which points to itself

eg

```
struct cde
{
    int data;
    struct cde * ptr;
};
```

```
struct cde mystruct;
struct cde * sptr;
```

```
mystruct.data = 10;
sptr -> data = 10;
```

* Memory Allocation

⇒ Static memory allocation

Memory allocated during compilation.

⇒ Dynamic memory allocation.

Memory allocated during the time of ~~allocation~~ execution.

Default value of global = 0.
Default value of local variable → garbage.

C++ → using 'new' & 'delete' keywords.
C → malloc()

NOTE: default return type of malloc() is (void *)

⇒ ~~malloc(10 * sizeof(int));~~
~~This statement requests~~

int * xptr;
>> xptr = (int *) malloc (sizeof (int))
typecast.

malloc basically returns a pointer which can access memory of sizeof(int)

Q. What is a data structure?
It is the way to organize the data in the memory of computers so that we can access it in an efficient manner.
Types: i) Linear eg queue, stack.
ii) Non-linear eg tree

* Queue.

Operations that can be performed.
i. Enqueue: Adding a new element
ii. Dequeue: Removing an element.

Queue works on First in First out system. (FIFO)

→ Logical AND (&&) | anything ≠ 0 (return value) is true.

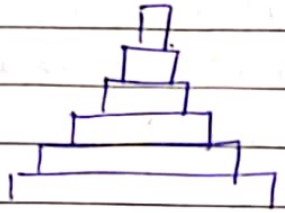
x	y	x && y
1	2	This gives true

Bitwise AND (& &)

(x & y)

0000 - 1 - 0001
0000 - - - 0010
0000 - - - 0000 → 0 → false

* Stacks



Stack works on Last in first out (LIFO) Principle.

Operations that can be performed.

i. • Push: adding in a element.
ii) Pop: removing an element.

⇒ Ways of implementation

Static memory Allocation — Array

Dynamically growing. by allocating memory at runtime eg. linked list

Q. What is the disadvantage of dynamic memory allocation?

i) Complex
ii) Allocated memory is non-contiguous. So the new elements have to be linked with already present elements of structure.

__/__/__

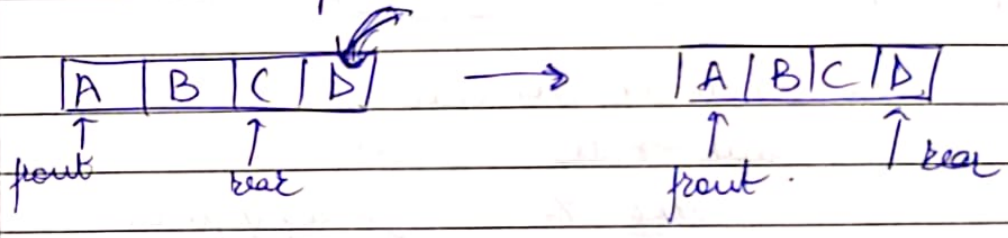
* Static Size Implementation.

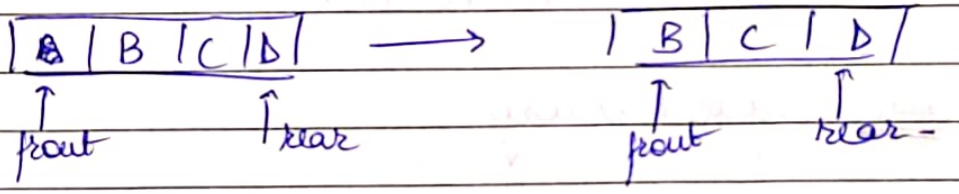
Queue

Create 2 variables: front, rear.

→ 'front' is updated when element is deleted.

→ 'rear' is updated when new element is added.





⇒ We need the following things to be addressed:-

1) 'Max' element?
2) We can't delete an element from empty queue.
3) We can't insert element in full queue.

- if front = 0, rear = -1] empty queue.

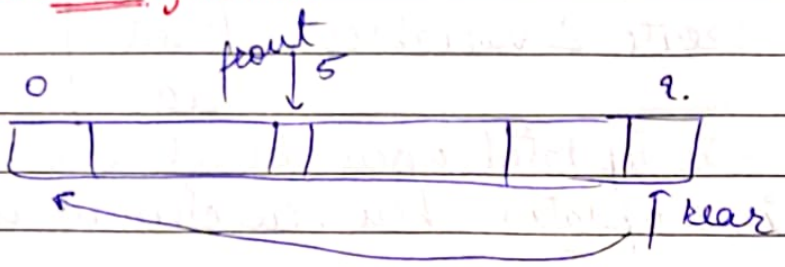
⇒ Exceptions

i) At time of declaration: front = -1, rear = -1

__/__/____

ii) On removing an element when length = 1 for deletion we can't update front.

* Circular Queues.



so when we do rear++;
rear → 10.
Take '%' → size of queue.
i.e. 10 % (10)

⇒ Code: Static Memory

```
#include <stdio.h>
#define MAX 10
int q[MAX];
int front = -1, rear = -1;

int isFull()
{
    if ((front == rear+1) || (front == 0 && rear == MAX-1))
    {
        return 1;
    }
    else
        return 0;
}
```

__/__/____

```
int isEmpty()
{
    if (front == -1)
        return 1
    return 0;
}
void enqueue(int ele)
{
    if (isfull())
        printf("Queue is full \n");
    else
    {
        if (front == -1)
            front++;
        rear++; rear = rear % MAX;
        q[rear] = ele;
        printf("Element added -> %d \n", ele);
    }
}

int dequeue()
{
    int ele;
    if (isempty())
    {
        printf("Queue is empty");
        return (-1);
    }
```

On linux OS, every successful execution of a process must return 0 to shell process. When we do return 0; in int main() is returned to shell. __/__/____

```
        else
        {
            ele = q[front];
            if (front == rear)
                front = rear = -1
            else
            {
                front = (front+1) % MAX;
            }
            printf ("Removed element is %d \n",
                                    ele);
        return (elm);
        }
}

void display()
{       int i;
    if (rear >= front)
        for (i= front; i<= rear; i++)
            printf ("%d ", q[front]);

    else
    {
        for (i=front; i<MAX; i++)
            printf ("%d \n", q[i]);
        for (i=0; i<= rear; i++)
            printf ("%d \n", q[i];
    }
}
```

tenenbaum
__/__/____

* OR.

```
if (isEmpty())
        printf("Empty Queue\n");
else
{
    printf("Front = %d \n", front);
    for (i=front; i != rear; i=(i+1)%MAX)
        printf("%d %t", q[i]);
    printf("%d", q[rear]);
}
```

look at condition & increment ⟹.

* Stack.
  Only 1 logical operator maintained i.e. top.

⇒ Push().
- check if top == MAX-1
- if not, top++;
  stack[top] = element.

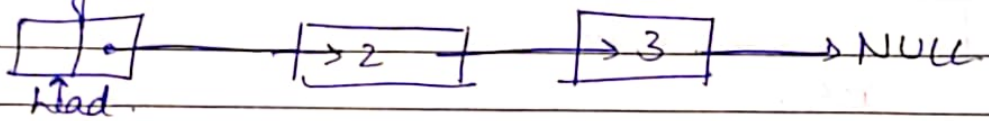
⇒ Pop()
- Check if (top == -1) i.e. stack is empty.
- else otherwise delel = stack[top];
  top--;
- return(delel)

__/__/____

* Linked List

It is a collection of objects linked Together by references from one objects to another. By convention we call these node.

→ Each node contains one or more data fields and a reference to the next node. The last node in the list points to NULL reference to indicate the end of list.

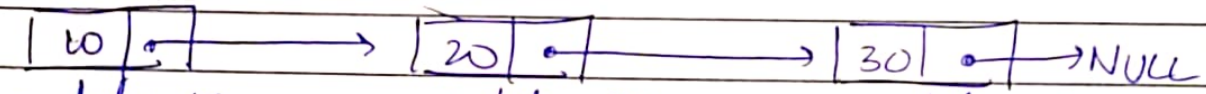


Q. Why is a node called self referential?

Node is called a self referential as it contain a pointer refering to itself.

Code..

```
struct node
{   int data;
    struct node * next;
};
struct node a, b, c;
```



a.data = 10 | b.data = 20 | c.data = 30

a.next = &b | b.next = &c | c.next = NULL

malloc(n * sizeof(int)) __/__/__
realloc.

Or

alternate: Dynamic Allocation.

```
struct node *ap, *bp, *cp;
/* 4 bytes allocated for each pointer */

ap = (struct node *) malloc (sizeof(struct node));
bp = "  ";
cp = "  ";

ap -> data = 10;        ap -> next = bp;
OR
*ap.data = 10;
bp -> data = 20;        bp -> next = cp;
cp -> data = 30;        cp -> next = NULL;
```

⇒ In a linked list, we have a variable 'head' pointing at begining of linked list.

```
struct node * head = ap;
```

Operations

Printing

```
void print (struct node *head)
{
    struct node * current;
    current = head;
    while (current != NULL)
    {
```

```
        printf("value is %d", current->data);
        current = current->next;
    }
}
```

Q. Count the number of nodes using given function prototype.

```
int count (struct node * head)
{
    struct node * p;
    int ctr = 0;
    p = head;
    while (p != NULL)
    {
        p++;
        p = p->next;
    }
    return (p);
}
```

~~Adding elements into linked list~~

Q. Add elements (n) read from keyboard & make linked list.

```
#include <stdio.h>
struct node
{
    int data;
    struct node * next;
};
```

Q.

__/__/____

```
int main()
{
    int n, i, val;
    struct node *head = NULL, *newnode, *current;
    printf("Enter value of n");
    scanf("%d", &n);
for(i=0; i<n; i++)
{
    newnode = (struct node *) malloc(sizeof(struct node));
    printf("Enter value to be inserted \n");
    scanf("%d", &val);
    newnode -> data = val;
    newnode -> next = NULL;
    if(head == NULL)
    {
        head = newnode;
        current = head;
    }
    else
    {
        current -> next = newnode;
        current = newnode;
    }
return 0;
}
```

Write insert function for:-

i) inserting element at end

2) insert data in sorted order.
3) insert data in sorted order.
4) insert at some index (first node = 0).

1) void insert (struct node *head)
{
    struct node *curr, *newnode;
    int val;
    printf("Enter value \n");
    scanf("%d", &val);
    cur = head;
    ~~while (cur->next != null);~~
    while (curr->next != null)
    {
        curr ~~curr~~ = curr->next;
    }
    ~~curr curr~~ newnode->data = val;
    curr->next = newnode;
    newnode->next = NULL;
}

2) void ~~data~~ insertbeg (struct node *head)
{
    struct node *newnode;
    int val;
    printf("Enter value");
    scanf("%d", &val);
    newnode->data = val;
    newnode->next = head;
    head = newnode;
}

__/__/____

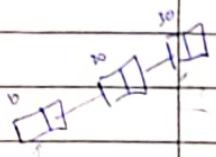
```
void insertind (struct node * head)
{
    struct node * cur, ptr, *newnode;
    int val, in;  int i = 0;
    printf("Enter index");
    scanf "%d" & in);
    printf("Enter value");
    scanf("%d", &val);
    cur = head;
    for (i=0; i<(in-1); i++)
    {
        cur = cur → next;
    }
    ptr = curs → next;
    curs → next = newnode;
    newnode → data = val;
    newnode → next = ptr;
}
```

i<2

Deleting in a linked list.

```
struct node * delete (struct node * head, int val)
{
    struct node * current, * prev;
    current = head;
    prev = current;
    while (current != NULL)
    {
```

```
if (current->data == val)
{  if (current == head)
        head = head->next;
   else
      {  prev->next = current->next;
         ~~prev->next = current->next;~~
         ~~current->next = NULL;~~
      current->next = NULL;    // not required
      free (current);
      }
   return head;
}
else
   {
      prev = current;
      current = current->next;
   }
}return head;
}.
```



Cons of linked list.

1) Time complexity

for searching an element:-

		'Big O' notation
i) Best case	(at i=0)	O(1)
ii) Worst case	(at i=N)	O(N)
iii) Average case		O(N/2)

__/__/____

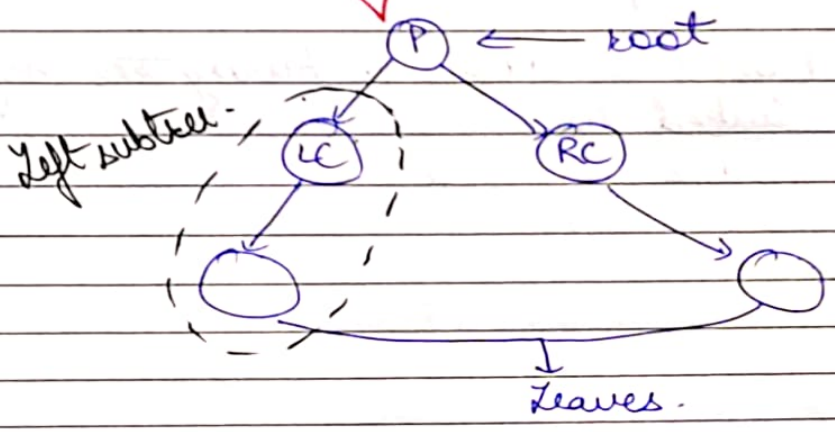
N - 2
N + 2
2N
N/2
> O(N)

So in linked list (Any case)

a) Inserting element at end = O(N) operation
b) Deletion of element → O(N) operation.
c) Searching of element = O(N) operation.

Hence this does not have efficient ~~~~ data arrangement.
So we study tree.

Tree.
If a node can have only 2 (max) child node it is called Binary Tree.



Binary Seach Tree.

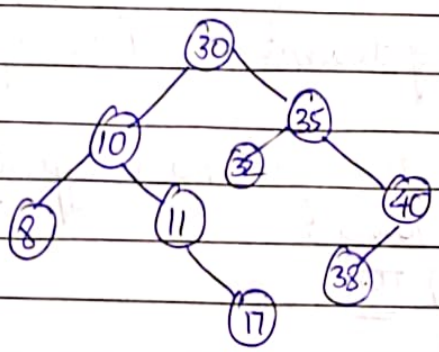
__/__/____

follows the property :-

a) data value in the left child will be less than parent
b) data value in the right child will be more than parent
c) Equal values maybe taken in left child.

Q Make a binary tree of

30, 10, 11, 8, 35, 17, 40, 38, 32



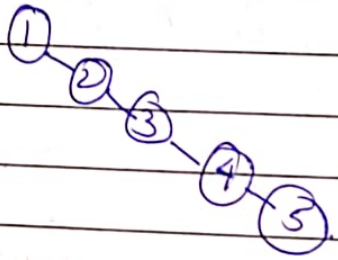
step ∝ levels.

for insertion operation = $O(\log_2 N)$

for search operation = $O(\log_2 N)$

NOTE: Based on data, a binary tree might form linked list.

1, 2, 3, 4, 5

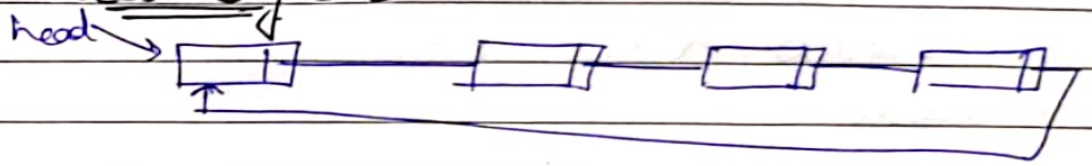


__/__/____

Linked List Variants

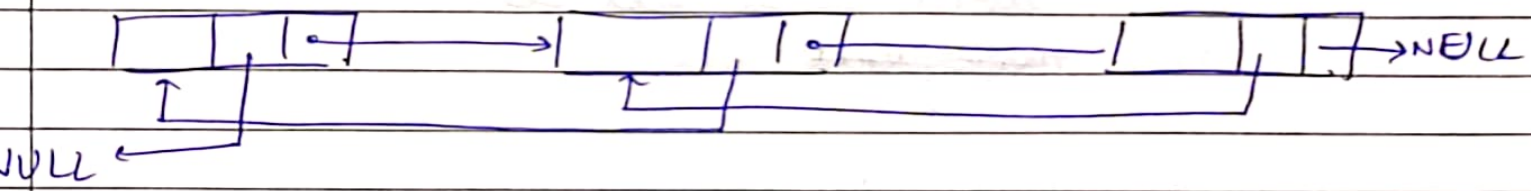
Singly linked list — Circularly linked list
doubly linked list

Circularly L.L.



Doubly linked list.

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



NOTE:

For insertion using call by reference.

```
void insertend(struct node **headref, int val)
{
    struct node *newnode, *current;
    current = (*headref);
```

int *a;
*a = ?;

char *str = "Hello";

__/__/____

→ By this we don't have to return newhead.
→ call this function as:
insertend(&head, 10);

```
#include <stdio.h>
struct node
{
    int data;
    struct node * next;
};
int main()
{
    struct node * head = NULL;
    insertfront1(head, 10);
    insertfront_a(head, 20);
    ~~insertfront~~
}

void insertfront_a (struct node ** headref, int val)
{
    struct node * newnode;
    newnode = (struct node *) malloc (sizeof(struct));
    newnode -> data = val;
    newnode -> next = (*headref);
    * headref = newnode;
}
```

~~void~~ struct node * insert_b (struct node * headf, int val)

{

struct node * newnode;
newnode = (struct node *) malloc (sizeof (struct node));
newnode → data = val;
if (headf == NULL)
{ newnode → next = NULL
headf = newnode;
return headf;
}

}

Doubly Linked List.

```
struct node
{       int data;
        struct node * prev, * next;
};
```

~~insert at~~ end Insertion at end.

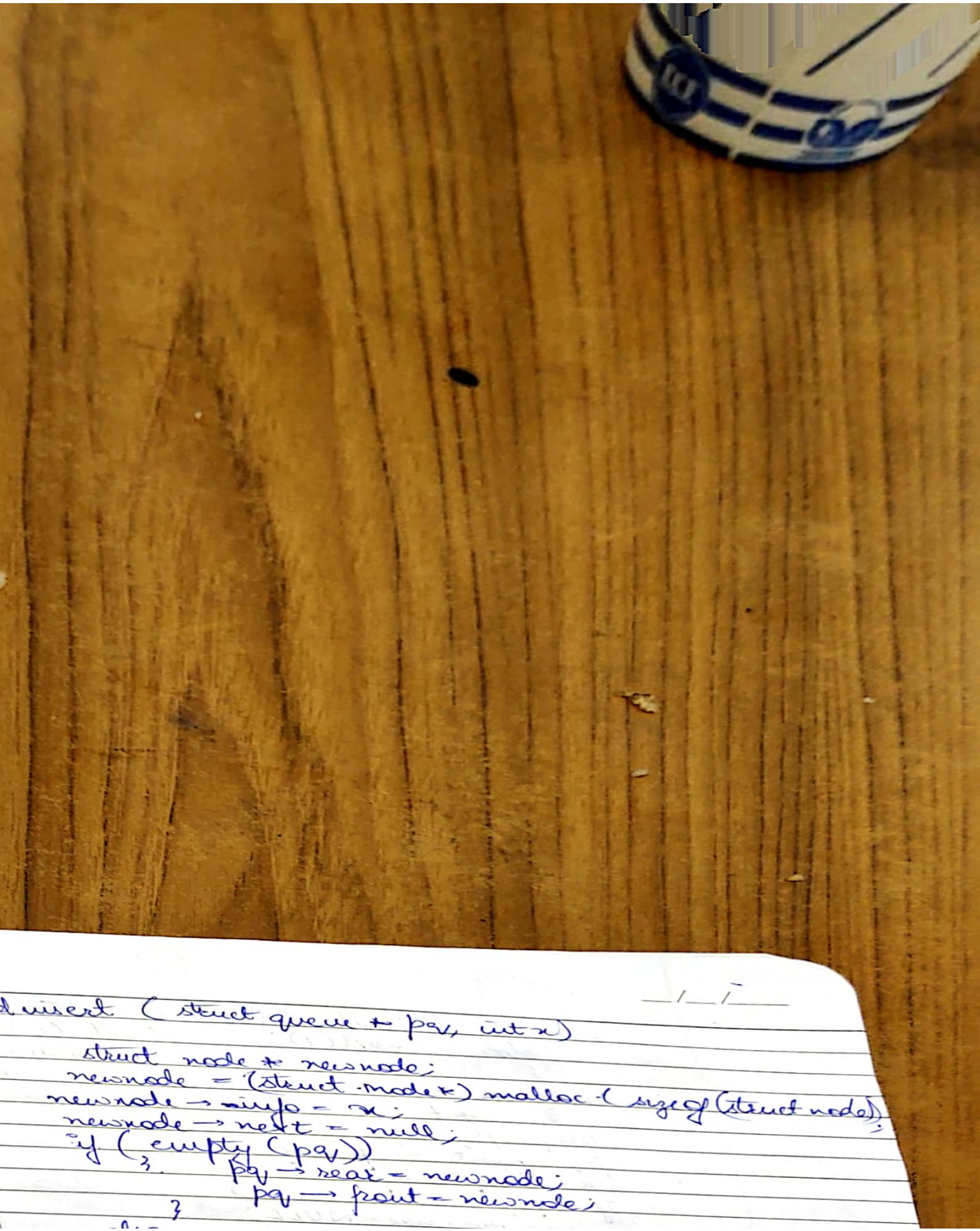
```
void insertatend (struct node **headref, int val)
{
        struct node * newnode, *current
        newnode = (struct node *) malloc (sizeof (struct node));
        newnode → data = val;
        newnode → next = NULL;
```

__/__/____

NULL;
node;

;
next != NULL)
urrent → next;
= current.

→ void
?.



```
insert ( struct queue * pq, int x)
    struct node * newnode;
    newnode = (struct node *) malloc ( sizeof (struct node));
    newnode → info = x;
    newnode → next = null;
    if ( empty (pq))
    {       pq → rear = newnode;
            pq → front = newnode;
    }
```

```
    ?   printf("Queue is empty");
    else   exit(1);   // not (0) as (0) means program terminated normally. convention.
    {
    else
    {   if
    temp = pq->front;
    pq->front = pq->temp->next;
    x = temp->info;
    if (pq->next = NULL)
        pq->rear = NULL;
    free(temp);
    return(x);
}

void display (struct queue *pq)
{
    if (empty(pq))
        printf("Queue is empty");
    else
    {   struct node * tmp;
        tmp = pq->front;
        while (temp->next != NULL)
        {
            printf("%d \t", temp->info);
            temp = temp->next;
        }
    }
}
```

priorities are sorted

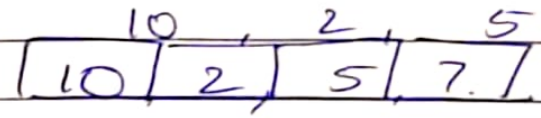
Scanned by CamScanner

__/__/____

Priority Queue

Assign a priority (key) to each data element. which can be implemented using sorted sequence or unsorted sequence

eg

Unsorted → 

Sorted → Asc → 10 → 2,10 → 2,5,10 → 2,5,7,10

Desc → 10 → 10,2 → 10,5,2 → 10,7,5,7

Main Operations

a) Insert b) Search c) Remove.

In unsorted Queue,

Insert → O(1)

Search → O(n)

Remove → O(1).

In priority Quee,

(c) Remove: Elements are removed which have highest priority

less value of key ✓ higher priority

Priority need not be unique.

Since we will have to traverse for if to find smallest key,

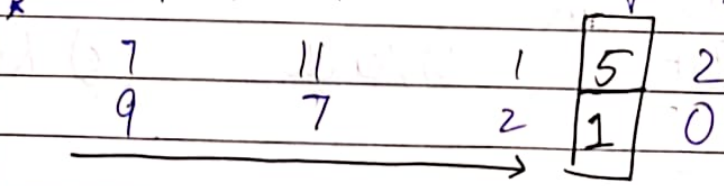
O(n). (unsorted)

a) Insert

Ascending: key → R

We insert & elements like unsorted but priorities are sorted.

	Unsorted	Sorted
Search	O(n)	O(n)
Insert	O(1) always front	O(n)* depends on priority of new element
Remove	O(n) find highest priority	O(1). highest priority



insert 5 [key=1].

A better Data structure than this is Heap.

⇒ Heap.

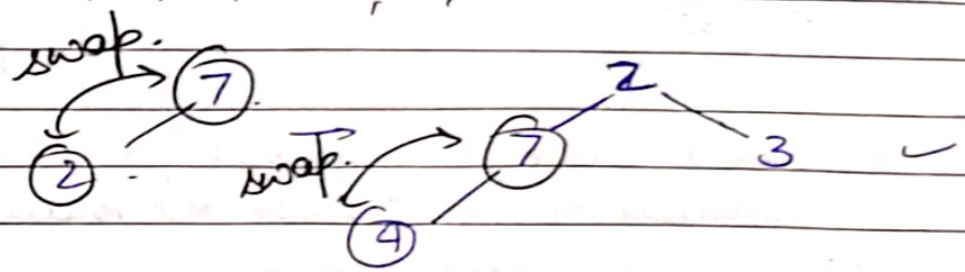
- A binary tree (not a binary search tree)
- Types:
  1) Min Heap
     value (or key if exists) of parent node is less than both left & right child node.
  2) Max heap
- Heap is always complete binary tree.

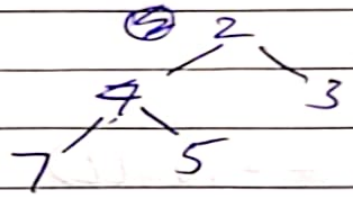
#

__/__/____

eg 7, 2, 3, 4, 5, 11, 10, 9.

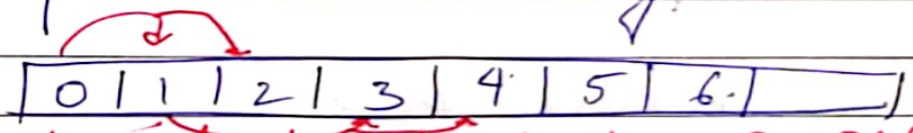
Min heap





so on.

Implemented in array.



element at index k has 2 child 2k+1 & 2k+1.

Time Complexity

Insert → O(log n)

Usage of priority Queues in Data Structures.

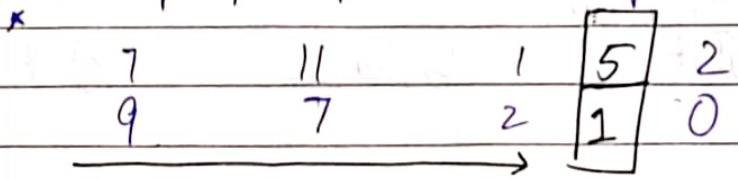
Implementation of priority Queues using Linked List. (sorted)

```
# include < stdio.h >
# include < stdlib >
{
    int data;
    int priority;
    struct node * front next    we don't need
};
```

Ascending ↓: key → R.

We insert & elements like unsorted but priorities are sorted.

	Unsorted	Sorted
Search	O(n)	O(n)
Insert	O(1) always front	O(n)* depends on priority of new element
Remove.	O(n) find highest priority	O(1). highest priority



insert 5 [key=1].

A better Data structure than this is Heap.

⇒ Heap.

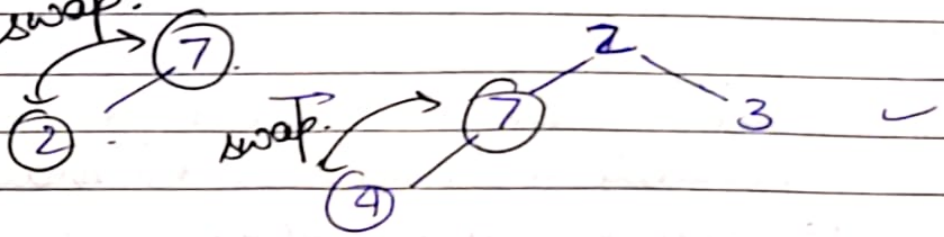
- A binary tree (not a binary search tree)
- Types:
  1) Min Heap
     value (or key if exists) of parent node is less than both left & right child node.
  2) Max heap
- Heap is always complete binary tree.

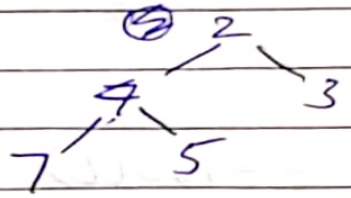
#

__/__/____

eg 7, 2, 3, 4, 5, 11, 10, 9.

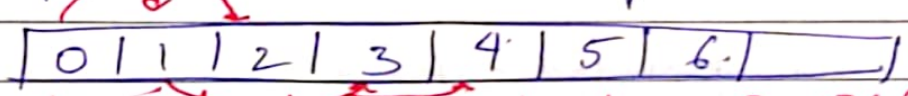
Min heap





so on.

Implemented in array.



element at index k has 2 child 2k+1 & 2k+1.

Time Complexity

Insert → O(log n)

Usage of priority Queues in Data Structures.

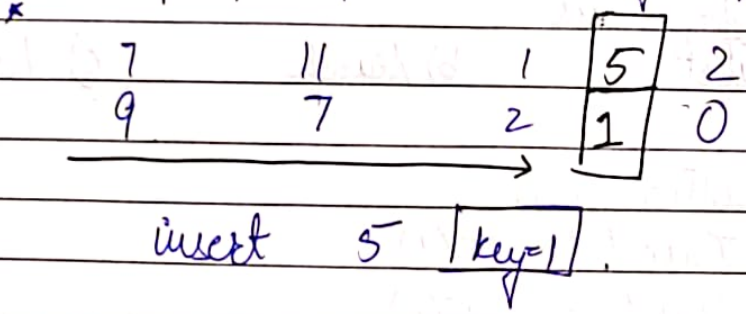
Implementation of priority Queues using Linked List. (Sorted)

```
# include < stdio.h >
# include < stdlib >
?
    int data;
    int priority;
    struct node * ~~front~~ next        we don't need
};
```

Ascending: key → R

We insert & elements like unsorted but priorities are sorted.

	Unsorted	Sorted
Search	O(n)	O(n)
Insert	O(1) always front	O(n)* depends on priority of new element
Remove	O(n) find highest priority	O(1). highest priority



insert 5 [key=1].

A better Data structure than this is Heap.

⇒ Heap.

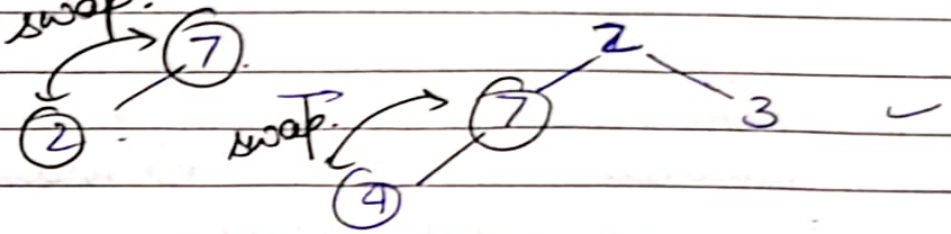
- A binary tree (not a binary search tree)
- Types:
  1) Min Heap
     value (or key if exists) of parent node is less than both left & right child node.
  2) Max heap
- Heap is always complete binary tree.

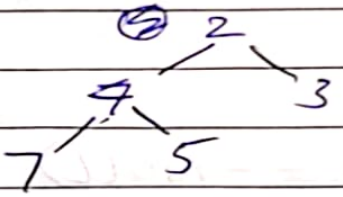
#

__/__/____

eg 7, 2, 3, 4, 5, 11, 10, 9.

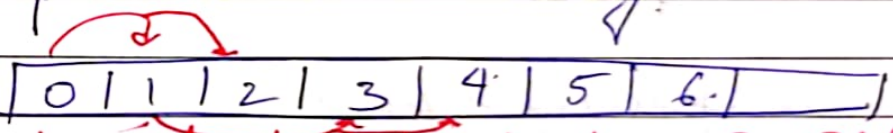
Min heap





so on.

Implemented in array.



element at index k has 2 child 2k+1 & 2k+1.

Time Complexity

Insert → O(log n)

Usage of priority Queues in Data Structures.

Implementation of priority Queues using Linked List. (Sorted)

```
#include <stdio.h>
#include <stdlib>
{
    int data;
    int priority;
    struct node * front next
};
```

```
struct node * insertitem(struct node **head, int val
                         , int prt)
{
    struct node *newnode, *current;
    newnode = (struct node *) malloc (sizeof(struct node));
    newnode->data = val;
    newnode->priority = prt;
    newnode->next = NULL;
    if (*head == NULL)
    {
        newnode->next = NULL;
        *head = newnode;
    }
    if (*head->priority > prt)
    {
        newnode->next = *head;
        *head = newnode;
    }
    else
    {
        current = *head;
        while((current->next == NULL) && current->next->
                                        priority < prt)
        current = current->next;
    }
    newnode->next = current->next;
    current->next = newnode;
}
```

__/__/____

* Applications of Linked List in solving problem.

Q. Josephus Problem

There is a army surrounded by enemy army → require additional support

To request one man must go

Only 1 horse.

Problem: Which one can go?

Constraints:

- There will be a number n
- There is a person, from which n number of person will be counted.
- The place where this n ends, that person will be eliminated from set on which same policy will be applied. Starting from next person.
- At end 1 person will remain. He would go.

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>.

void insert (struct node **cq, char *sn);
void display (struct node **cq);
void remove_after (struct node **cq, char *sn);

int main ()
```

```
}.
    char name[40];
    const char * end = "end";
    int i, n;
    struct node * list = NULL;
    printf("Enter n \n");
    scanf("%d", &n);
    printf("Enter name (end to stop)");
    scanf("%s", name);
    while (strcmp(name, end) != 0)
    {
        insert(&list, name);
        printf("Enter name (end to stop)\n");
        scanf("%s", name);
    }
    display(&list);
    printf("The order in which soldiers are removed: \n");
    while (list != list->next)
    {
        for(i=1, i<n; i++)
        {
            list = list->next;
        }.
        removeafter(&list, name);
        printf("Removed soldier is %s \n", name);
    }.
    printf("The soldier who goes is %s \n",
                                list->sdname);
    free(list);
```

```
    return(0);
}

void insert(struct node **cq, char *sn)
{
    struct node *newnode;
    newnode=(struct node *)malloc(sizeof(struct node));
    strcpy(newnode->sname, sn);
    if((*cq)==NULL)
    {
        *cq=newnode;
    }
    else
        newnode->next=(*cq)->next;
    *cq->next=newnode;
    (*cq)=newnode;
    return;
}

void display(struct node **cq, char *sn)
{
    struct node *current;
    if((*cq)==NULL)
    {
        printf("Empty Queue");
        return;
    }
```

```
    current = *cq;
    do
    {
        printf("Name is: %s \n", current->sdname);
        current = current->next;
    } while (current != (*cq));
    return;
}
void removeafter (struct node **cq, char *sq)
{
    /* This function removes a node after the node
    whose address is passed to this function in cq
    and the value of node getting removed is filled in
    sn */
    struct node *temp;
    temp = *cq->next;
    *cq->next = temp->next;
    strcpy (*sq, temp->sdname);
    free (temp);
}
```

alternate (more professional approach.

```
    if ((*cq == NULL) || (*cq != *cq->next))
    {
        printf("Deletion void \n");
        return;
    }
```

char *sq is call by reference for string & so use strcpy.

```
else
{   struct node * temp;
    temp = *cq -> next;
strcpy (*sq->, temp->sdata);
    *cq -> next = temp -> next;
    free(temp);
}
```

Addition of Integers of arbitary length.

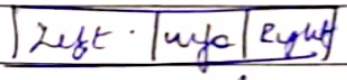
Addition of Long integers using doubly linked list.

→ max 5 digits per node (circularly).

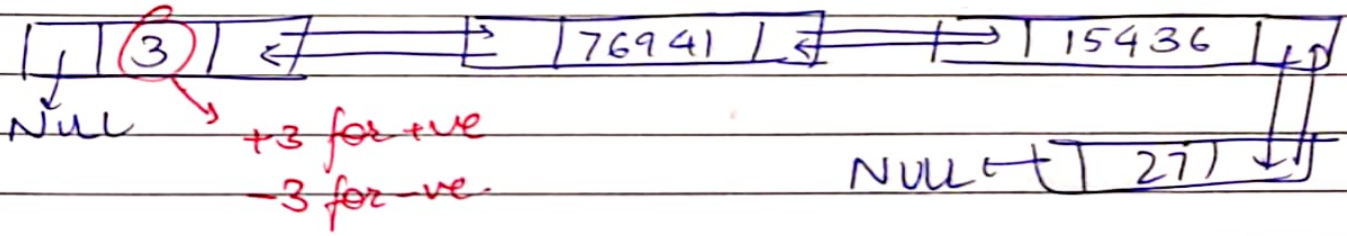
→ There will be a special node header, which info serves 2 purpose.

:→ sign of info (+ve or -ve)

+ magnitude of info tells the number of nodes representing the long int.



27 | 54 | 367 | 6941



+3 for +ve
-3 for -ve.

Code.

compabs ⇒ compairs absolute value of 2 integers
returns 1 if argument 1 > argument 2.

```
int compabs (struct node *p, struct node *q)
```

```
}
    if ( abs(p->info) > abs(q->info))
            return(1);
    if ( abs(p->info) < abs(q->info))
            return (-1).

    struct node *r, *s;
// If counts are equal.
    r = p->left;
    s = q->left;

while ( r != p)
{
    if ((r->info) > (s->info))
            return(1);
    if ((r->info) < (s->info))
            return(-1);
    s = s->left;
    r = r->left
}
 return(0);
}
```

adddiff → adding numbers of different signs
addsame → " " " same "

~~7 9 3 4 1~~
~~2 3 5 7 1~~

→ ~~Absolutely~~ Absolute value of first is not less than second
→ Returns a pointer to a list representing the sum of integers
→ We must be careful to terminate the leading 0's from the sum.

```
struct node * adddiff (struct node *p, struct node *q)
{
    int count;
    struct node *pptr, *qptr, *r, *s, *zeroptr;
    long int huntho = 100000L;
    long int borrow, diff;
    int zeroflag;
    count = 0;
    borrow = 0;
    zeroflag = FALSE;
// generate head node for sum.
    r = (struct node *) malloc sizeof (struct node);
    r->left = r->right = r;
    pptr = p->right;
    qptr = q->right;
    while (qptr != q)
    {
        diff = (pptr->info) - borrow - (qptr->info);
        if (diff >= 0)
            borrow = 0;
        else
        {   diff = diff + huntho;
            borrow = 1;
        }
```

// L enforces it to allocate memory as long.
it won't if it can be stored in int

// this huntho is for the

/*

```
unsigned int a,b,x,y;
x=3;
y=4;
a=x-y;
b=y-x;
if(a>0)
   printf("*");
else
   printf("#");
if(b>0)
   printf("*");
else
   printf("#");
}
```

OUTPUT

* *

in unsigned int.

-1 → (Max limit) - 1

-2 → (Max limit) - 2. i.e. +ve.

*/

/*

Suppose number is 34 - borrow - 54

in computer 2's complement system:

⇒ 34 + (-54)

```
0010   0010
1100   1010
1110   1100
```

```
  34
  54
 -20
```

34 — 2

54 — 1

Now. Lets add 100 in binary.

```
1110   1100
0110   0100
10100  0000   ⟶ 80.
```

```
 234
+154
 80.
```

human signifies borrow

*/.

// writing loop again.
while (qptr -> info != q)
{
    diff = (pptr -> info) - borrow - (qptr -> info);
    if (diff >= 0)
        borrow = 0;
    else
    {
        diff = diff + hundred;
        borrow += 1;
    }
// generate a new node & insert it to left of header.
    insertleft(&r, diff);
    count += 1;
//test for zero node
    if (diff == 0)
    {
        if (zeroflag == FALSE)
        {
            zeroptr = r -> left;
            zeroflag = TRUE;
        }
        else
            zeroflag = FALSE;
    }
    pptr = pptr -> right;
    qptr = qptr -> right;

```
} // end while.

// traverse the remaining p.
while (pptr != p)
{
    diff = (pptr -> info) - borrow;
    if (diff >= 0)
        borrow = 0;
    else
    {
        diff = diff + number
        borrow = 1;
    }.
    insert left (&r, diff);
    count = count + 1;
    if (diff == 0)
    {
        if (zeroflag == FALSE)
        {   zeroptr = r -> left;
            zeroflag = TRUE;
        }.
        else
            zeroflag = FALSE
    }.
    pptr = pptr -> right;
}.
// deleting leading zeros.
```

```
if (zeroflag == TRUE)
{
    while (zeroptr != s)
    {
        s = zeroptr;
        zeroptr = zeroptr -> right;
        delete (s, &diff);
        count = count - 1;
    }
}
// insert count & sign
// into the header.
if (p->info > 0)
        s -> info = count;
else
    s -> info = -count;
return (s);

} // end add diff.
```

```
} // end while

// traverse the remaining p.
while (pptr != p)
{
    diff = (pptr -> info) - borrow;
    if (diff >= 0)
        borrow = 0;
    else
    {
        diff = diff + num[base];
        borrow = 1;
    }
    insert left (&r, diff);
    count = count + 1;
    if (diff == 0)
        if (zeroflag == FALSE)
        {
            zeroptr = r -> left;
            zeroflag = TRUE;
        }
        else
            zeroflag = FALSE
    pptr = pptr -> right;
}
// deleting leading zeros.
```

```
if (zeroflag == TRUE)
{
    while (zeroptr != r)
    {
        s = zeroptr;
        zeroptr = zeroptr -> right;
        delete (s, &diff);
        count = count - 1;
    }
}
// insert count & sign
// into the header.
if (p->info > 0)
        r -> info = count;
else
    r -> info = -count;
return (r);

} // end add diff.
```