```
for (i = 0; i < 10; i++)
{
    printf("Hello world\n");
}
```

ideal timecomplexity

O(1) -> best complexity

O(1 * ~~10~~) --> TC : O(1)

O(10)

If there is a constant value then igore it.

+, -, / , * =>

1, log n, n,

n = 20

count = 5

```
for (i = 1; i <= n; i += 2)
{
    count++;
}
```

i = 1 + 2 = 3

i = 3 + 2 = 5

i = 5 + 2 = 7

i = 7 + 2 = 9

i = 9 + 2 = 11

n/2

n -> variable apply rule 2

TC : O(1 * n/~~2~~) rule 1

TC : O(n)

```
p = 0

for ( i = 1; p <= n; i++)
{
    p = p + i;
}
```

n = 15

i	p	
1	0 + 1	= 1
2	1 + 2	= 3
3	1+2+3	= 6
4	1+2+3+4	= 10
5	1+2+3+4+5+....+k ?	

3rd rule, When there are different values/time complexitie always consider the worst case (or) Highest degree

p = k(k+1) / 2

= (k^2 + k) / ~~2~~

= k^2 ~~+ k~~

p = k ^ 2

p > n => k^2 > n

=> k^2 = n

=> k = root(n)

TC : O(root(n))

```
for ( i = n / 2; i <= n; i ++)
{
        for (j = 1; j <= n; j++)
        {
                stmt;
        }
}
```

$\longrightarrow$ n / 2

$\longrightarrow$ n

= O(n/2 * n)
= O(n^2 / 2)
= O(n^2)

```
for ( i = 1; i * i <= n; i++)
{
        k++;
}
```

i ^ 2 <= n

i ^ 2 = n

i = root(n)

Non-linear loops

```
for (i = 1; i < n; i *= 2)
{
        stmt;
}
```

i = i * 2

2 ^ k = n

$k = \log_2 n$

TC: O(log n)

n = 20 ?

i	i < 20	i *= 2	
1	1 < 20	i = 1 * 2	2 ^ 0
2	2 < 20	i = 2 * 2	2 ^ 1
4	4 < 20	i = 4 * 2	2 ^ 2
8	8 < 20	i = 8 * 2	2 ^ 3
16	16 < 20	i = 16 * 2	2 ^ 4
32	32 < 20		2 ^ k

```
for (i = n; i > 0; i /= 2)
{
        stmt;
}
```

2 ^ k = n

k = log n

TC : 0(log n)

n = 16

i		i = i / 2
16	2 ^ 4	16 / 2
8	2 ^ 3	8 / 2
4	2 ^ 2	4 / 2
2	2 ^ 1	2/2
1	2 ^ 0	1/2

```
i = 0;

while (i < n)
{
        stments;
        i++;
}
```

```
for (i = 0; i < n; i++)
{
        stmt;
}
```

TC : O(n)

Recursive function:

```
fun(num)
{
        if (num >= 1)
        {
                printf(num);
                fun(num-1);
        }
}
```

fun(3)

4

5

n   n + 1

TC : O($\cancel{n + 1}$)

TC : O(n)

n = 20

fun(4)

4

fun(3)

3

fun(2)

2

fun(1)

1

fun(0)

fun(n)   => n + 1

10

```
for (int i = 0; i < n / 2; i++)
{
        for (j = 0; j + n / 2 < n; j++)
        {
                for (k = 1; k < n; k *= 2)
                {
                        c++;
                }
        }
}
```

O(n/2)

O(n/2)

O(logn)

```
for(i=0;i<n;i++)
{
        for(j=1;j<n;j=j*2)
        {
                //statement
        }
}
```

O(n)

O(log n)

TC : O(n * log n)

TC : O(n log n)

O(n/~~2~~ * n/2 * logn)

O(n * n * log n)

O(n^2 logn)

2   4   1   5   7   9

key = 2

key = 19

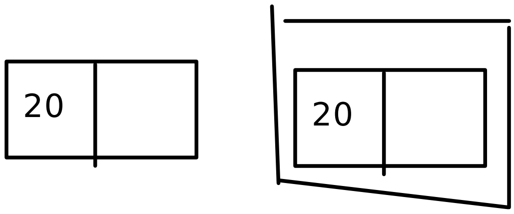
O(1) => if data found at the 0th index

O(n) => if the data not found / if the data found at the last index

O(n) => Average case

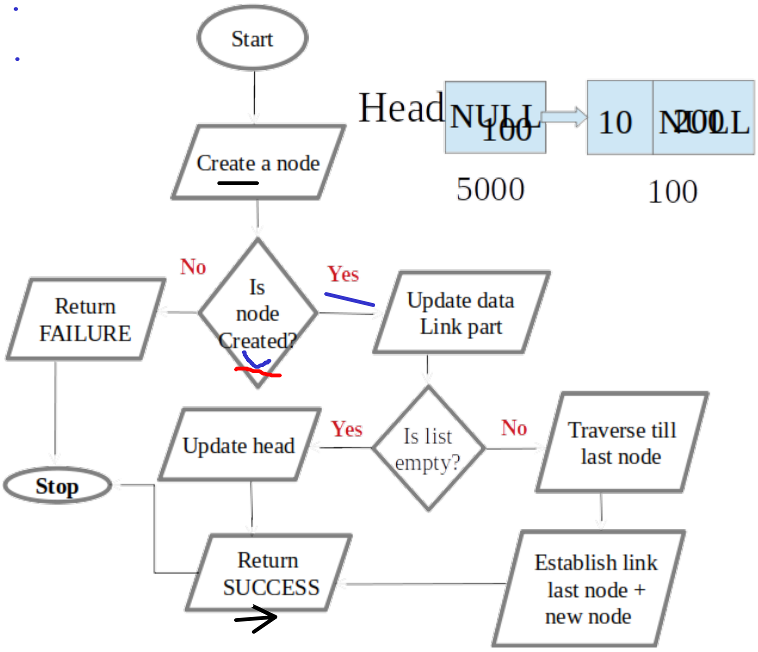
```
typedef struct node {
        int data;
        struct node * link;
    } Slist_t;
  typedef struct node     Slist_t;

int * ptr = &integer;

char *ptr = &char_var;

double *ptr;

void *
```

```
struct node * var;

var = malloc(sizeof(struct node)) ?
```



%d
%c
%f

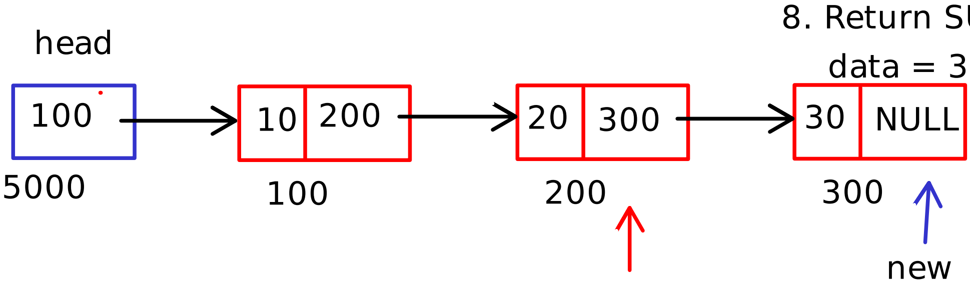
Slist_t (or) Slist



Psuedo code:

1. new ← Memalloc(sizeof(Slist_t))
2. if (new = NULL)
   Return FAILURE
3. new -> data ← data
4. new -> link ← NULL
5. if (head = NULL)
   head ← new
   Return SUCCESS
6. temp ← head
6. while (temp -> link != NULL)
   {
      temp ← temp -> link
   }
7. temp -> link ← new
8. Return SUCCESS

data = 30



Linked List:

Why Linked list?

Static Memory allocation

Array

Dynamic memory allocation

495?

Static Memory allocation:

char name[500];

0	1	2	3	4
A	b	h	i	\0

Abhi

Nithya

Disadvantage:

1. Shortage of memory
2. Wastage of memory

Dynamic memory allocation:

1. malloc
2. calloc
3. realloc

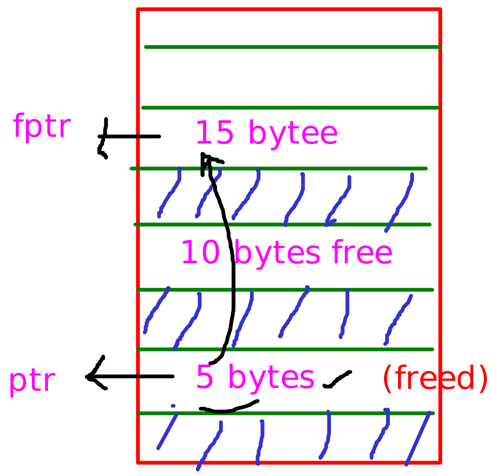
Used allocate the memory

Extending or shrinking the memory

char *ptr = malloc(5);

10 more bytes

fptr = realloc(ptr, 15);



1000s

Disadvantage:

1. coping the old data to new memory, takes more time.

Linked list