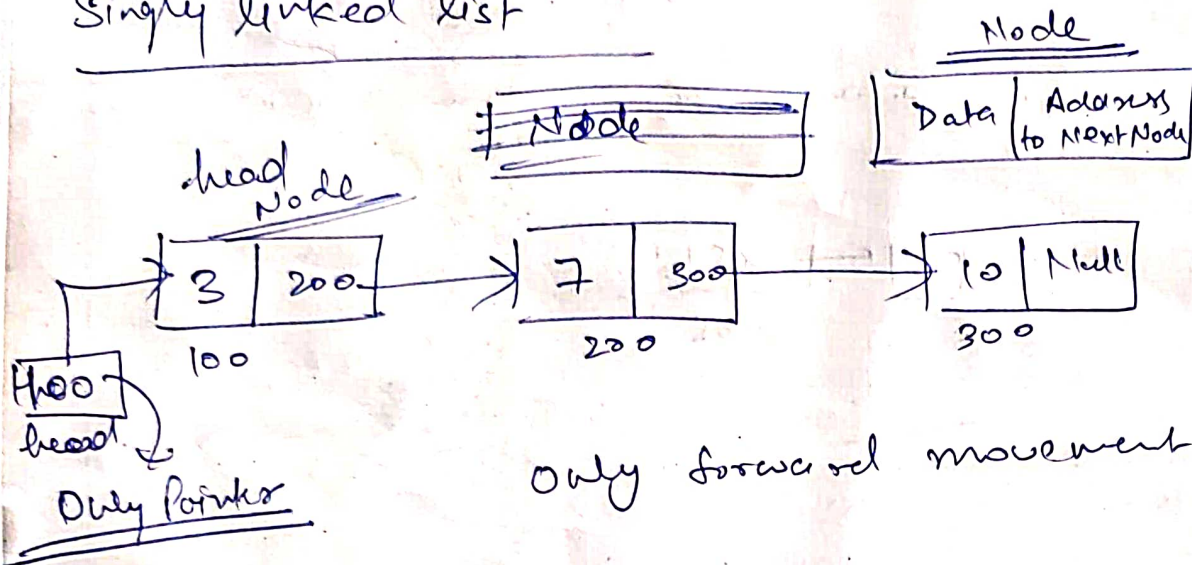


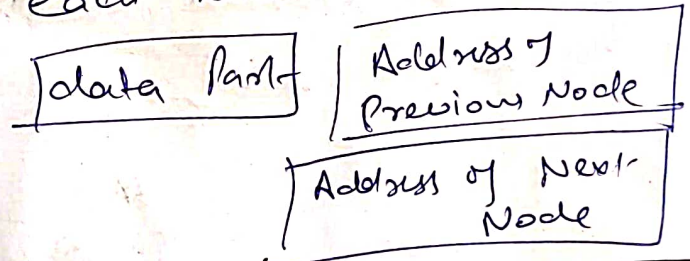
Types of linked list

Singly linked list



Doubly linked list

each node will have

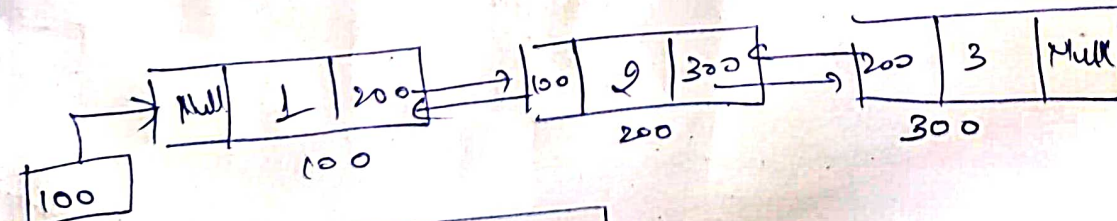
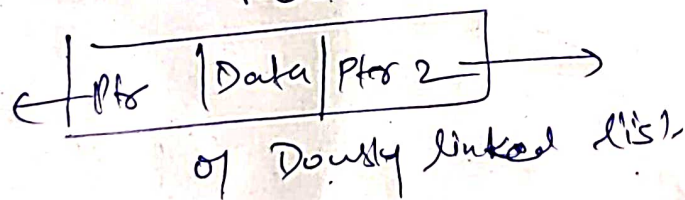


struct node

```

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

Node



struct Node

```

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

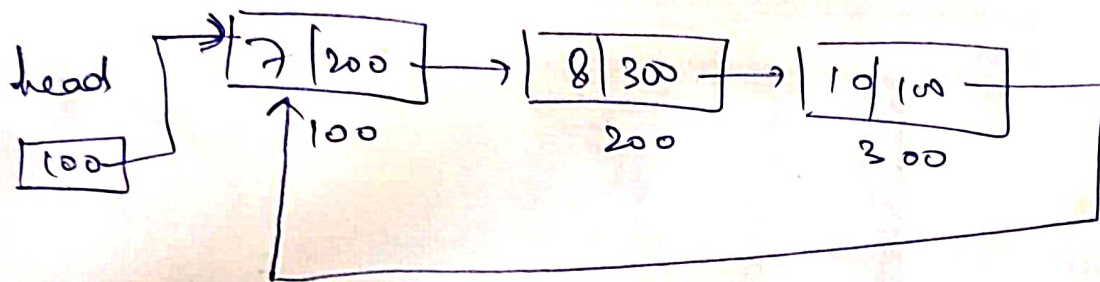
Representation of
Doubly linked list
in C

SLY 10:05 = 11:00

001	12	41	353
002	13	44	355
003	14	45	357
004	15	46	359
005	16	47	361
006	17	48	363
007	18	49	365
008	19	50	367
009	20	51	369
010	21	52	371
011	22	53	373
012	23	54	375
013	24	55	377
014	25	56	379
015	26	57	381
016	27	58	383
017	28	59	385
018	29	60	387
019	30	61	389
020	31	62	391
021	32	63	393
022	33	64	395
023	34	65	397
024	35	66	399
025	36	67	401
026	37	68	403
027	38	69	405
028	39	70	407
029	40	71	409
030	41	72	411
031	42	73	413
032	43	74	415
033	44	75	417
034	45	76	419
035	46	77	421
036	47	78	423
037	48	79	425
038	49	80	427
039	50	81	429
040	51	82	431
041	52	83	433
042	53	84	435
043	54	85	437
044	55	86	439
045	56	87	441
046	57	88	443
047	58	89	445
048	59	90	447
049	60	91	449
050	61	92	451
051	62	93	453
052	63	94	455
053	64	95	457
054	65	96	459
055	66	97	461
056	67	98	463
057	68	99	465
058	69	100	467
059	70	101	469
060	71	102	471
061	72	103	473
062	73	104	475
063	74	105	477
064	75	106	479
065	76	107	481
066	77	108	483
067	78	109	485
068	79	110	487
069	80	111	489
070	81	112	491
071	82	113	493
072	83	114	495
073	84	115	497
074	85	116	499
075	86	117	501
076	87	118	503
077	88	119	505
078	89	120	507
079	90	121	509
080	91	122	511
081	92	123	513
082	93	124	515
083	94	125	517
084	95	126	519
085	96	127	521
086	97	128	523
087	98	129	525
088	99	130	527
089	100	131	529
090	101	132	531
091	102	133	533
092	103	134	535
093	104	135	537
094	105	136	539
095	106	137	541
096	107	138	543
097	108	139	545
098	109	140	547
099	110	141	549
100	111	142	551

③ Circular linked list

is only single linked list- only with one difference that the last node's Address part will point back to ~~head~~ first node.



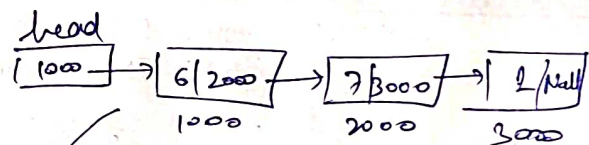
Array Vs linked list

① Cost of Accessing an element

0	1	2	3	4
6	7	0	1	5
100	104	108	112	116

$$a[2] = 100 + 4 \times 2 = 108$$

taking $O(1)$ time



Since only sequential access is possible < unlike array contiguous allocation is not there to access a particular element it will take $O(\frac{n}{2})$ time $\approx O(n)$ [depends on no. of nodes in linked list]

② Memory Requirement & Memory Utilization

0	1	2	3	4	5	6
6	7	1	-	-	-	-

~~total size required~~
total memory allotted = $7 \times 4 = 28$ bytes.

~~for~~ up to 7 data it will still require = 28 bytes

data + Address
Ex: $4 + 4 = 8$ bytes for one node

\Rightarrow total size = $8 \times 3 = 24$ bytes

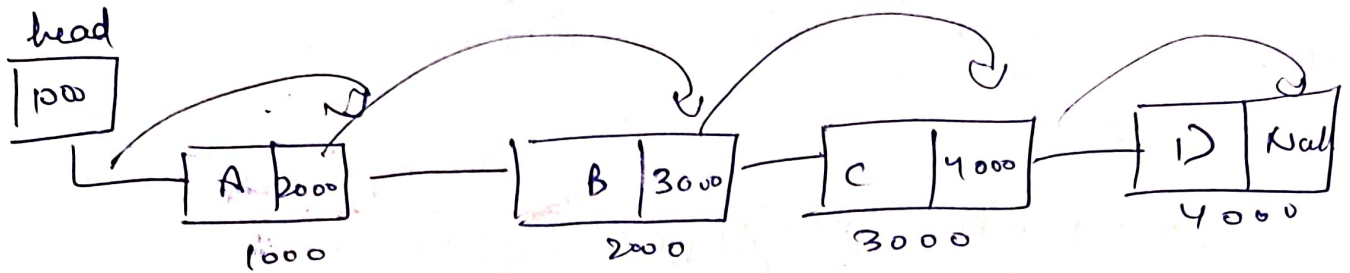
but in case of 7 nodes linked list would require $8 \times 7 = 56$ bytes.

Hence Memory requirement is less in Array but Memory utilization is efficient in linked list.

(3)

Traversal in linked list

the process of visiting each node of the list once is called traversing.

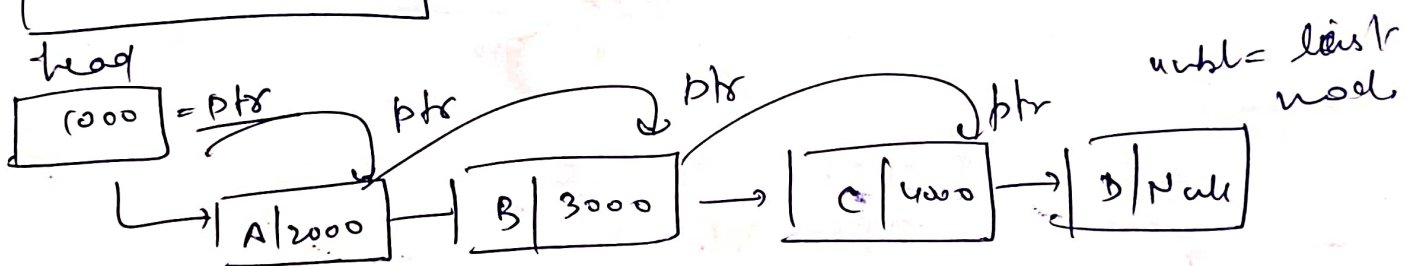


to traverse the link list we need to have one pointer before our first node, coz we can't use head pointer for traversal.

head ← head → next will lose 1

head = 2000 then first node will be lost.

struct node * ptr



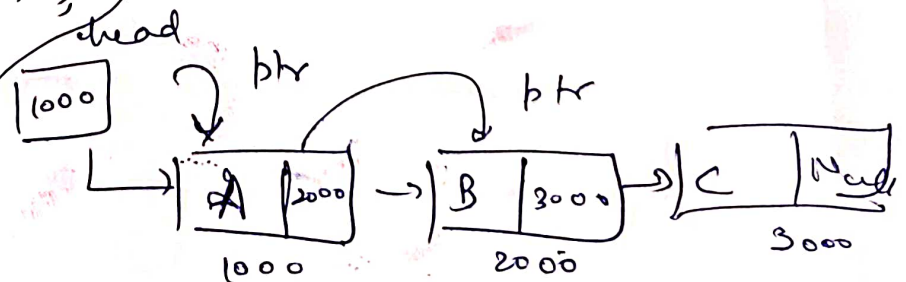
traverse(struct node * head)

```

{ if (head == null)
  printf("linked list is empty");
  struct node * ptr;
  ptr = head; (ptr = 1000)
  while (ptr != null)
  { ptr = ptr → next;
  }
}

```

3.



malloc() in Programming

Syntax of malloc()

→ void * malloc (size of ());

<stdlib.h>

return void pointer

malloc(2)

blocks of 2 bytes
is available.

~~include~~

#include <stdio.h>

#include <stdlib.h>

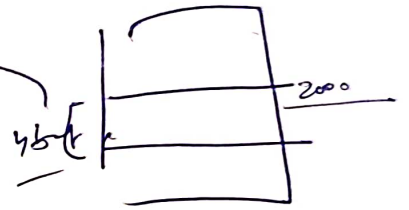
void main()

int * p;

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

*p = 100

p = (int *) malloc (10 * size of (int));



Inserting a node at the beginning

insertbeg (struct node * head, int info) → info of new node to be inserted.

{ struct node * new;

new = (struct node *) malloc (size of (struct node))

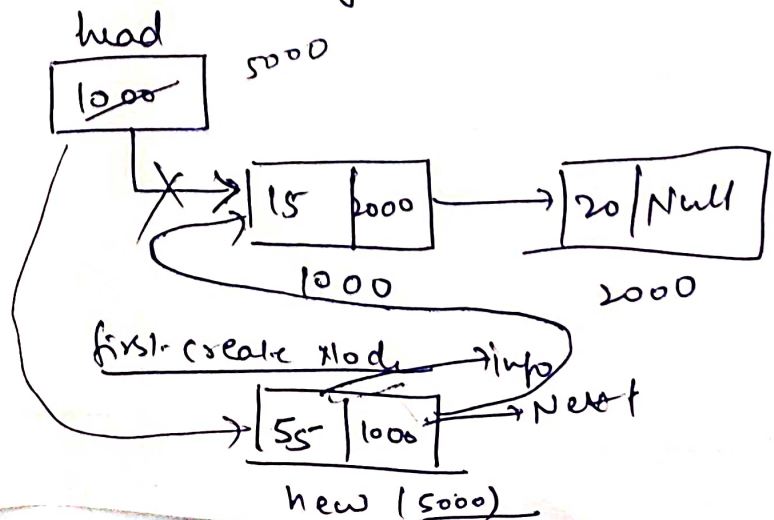
new → data = info;

new → next = head;

head = new;

return head;

}



Inserting a node at the end of linked list (5)

insert_end (struct node * head, int info)

{

struct node * ptr, * new;

new = (struct node *) malloc (sizeof (struct node));

new → data = info;

new → next = null;

ptr = head;

if (ptr != null)

{ while (ptr → next != null)

ptr = ptr → next;

ptr → next = new;

}

else

head = new;

return head;

}

Inserting after a specific node in a linked list

6

insert_after (struct node * head, int x, int info)

{ struct node * ptr, * new;

new = (struct node *) malloc (size of (struct node));

new → data = info;

ptr = head;

while (ptr → data != ~~22~~ 22 ptr != null)

{ ptr = ptr → next;

}

if (ptr → data == x)

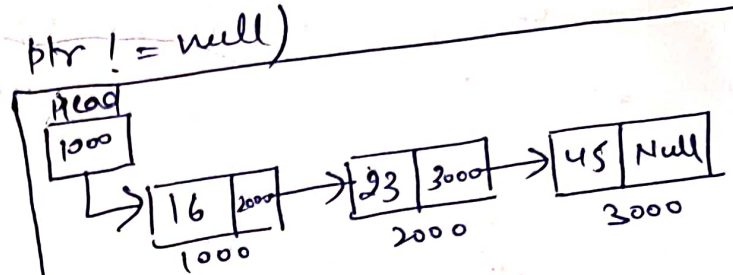
{ new → next = ptr → next;

ptr → next = new;

}

return head;

}



Deletion from the Beginning of the linked list

del-first (struct node * head)

{ struct node * ptr

if (head == null)

printf ("list is empty");

else

{ ptr = head;

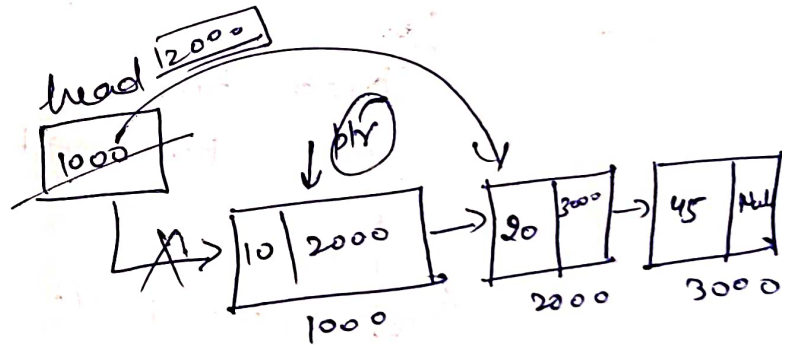
head = head → Next;

free (ptr);

}

return head;

}



(7) Deletion from the end of linked list

```
del - last ( struct node * head )
```

```
{
    struct node * ptr, * prep;
    if ( head == null )
        printf ( "list is empty" );
    elseif ( head -> next == null )
    {
        free ( head );
        head = null;
    }
    else
    {
        ptr = head;
        while ( ptr -> next != null )
        {
            prep = ptr;
            ptr = ptr -> next;
        }
        prep -> next = null;
        free ( ptr );
    }
    return head;
}
```

3- Deletion after a given node of the linked list (7)

delete_after (struct node * head, int key)

```
{
    struct node * ptr1, * ptr2;
    ptr1 = head;
    while (ptr1 -> next != null)
    {
        if (ptr1 -> data == key)
        {
            ptr2 = ptr1 -> next;
            ptr1 -> next = ptr2 -> next;
            free(ptr2);
            break;
        }
        ptr1 = ptr1 -> next;
    }
    return head;
}
```

