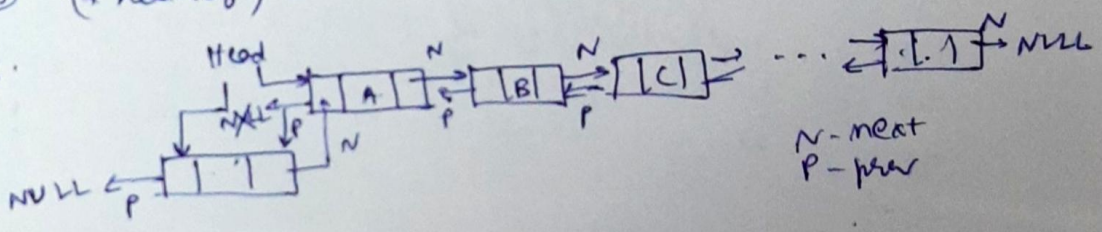
Assignment-2

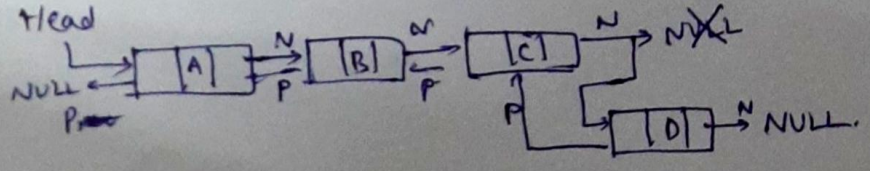
1) we can use doubly linked list for this. Let us see one by one.

. Insertion at the beginning - First allocate node, then put in the data. Make next of new node as head and previous as NULL. change previous of head node to new node. Move the head to point to the new node.

```
void push (struct Node** head_ref, string data)
{
① struct Node* new_node = (struct Node*) malloc (sizeof(struct Node));
② new_node → data = ndata;
③ new_node → next = (*head_ref);
   new_node → prev = NULL;
④ if ((*head_ref) != NULL).
      (*head_ref) → prev = new_node;
⑤ (*head_ref) = new_node;
}.
```

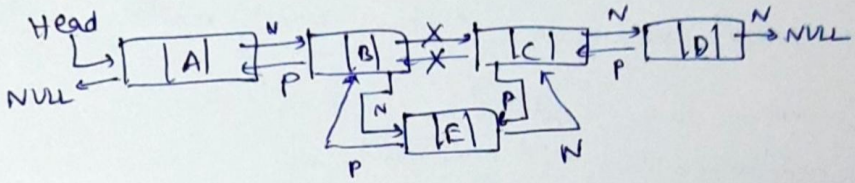


. Insertion at the end - First allocate node, then put in the data. This new node is going to be the last node, so make next of it as NULL. If the linked list is empty, then make the new node as head. Else traverse till the last node. change the next of last node. Make last node as previous of new node.

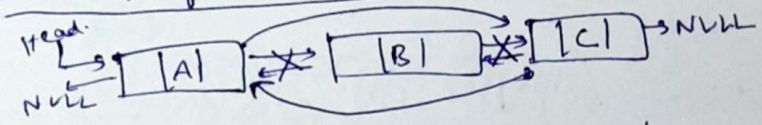


Pseudo code will similar as earlier.

- Insertion at median location (middle) - Since its in the middle, the previous node will not be NULL. Similar to the previous 2 methods, first allocate new node and then put in the data. Make next of new node as next of prev node. The prev. node is given to us, thus DLL will take O(1) time for insertion. (its the middle one). Make the next of prev. node as new node. Make prev node as previous of new node. Change the previous of new node's next node. The below diagram explains it better.



- Deletion of second last element -



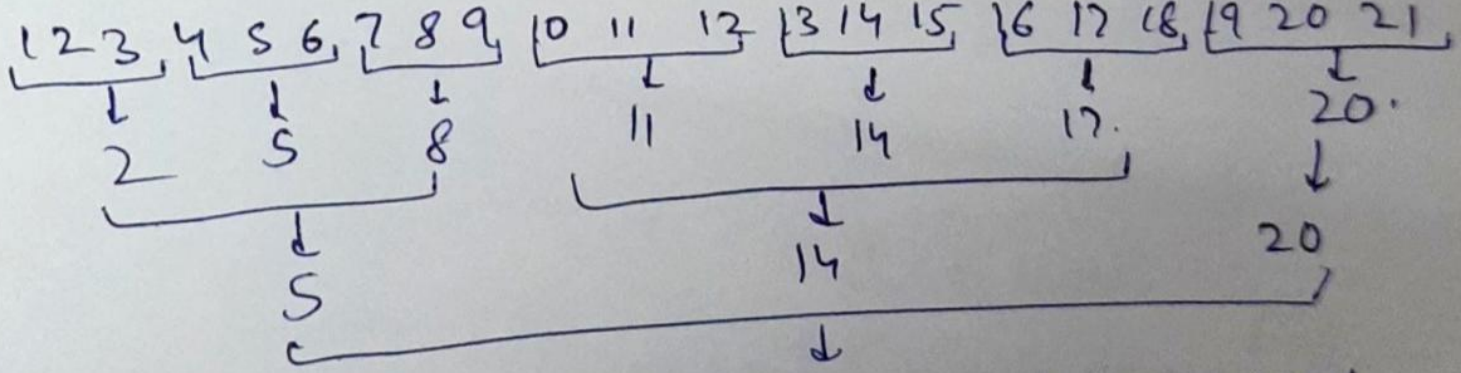
For the second last element, considering length of DLL > 2, the previous node will not be NULL. Traverse the DLL backwards and get the ~~node~~ prev. and next node of second last element (B in this case) ~~Update that next and prev node~~, ~~to the element's~~ previous node of last ~~last~~ ~~no~~ element and the next node of third last element. ~~First~~ Then update the nodes as follows -

a) point next node of third last element to last node.

b) point prev node of last element to third last element.

Since, we do not have to traverse the whole DLL, the time complexity is O(1).

3) Consider 21 elements.



We can see it as a recursive procedure, so for each division, we are skipping. $12 \times \left(\frac{3^n}{2} + 1\right)$ elements. (Integral division i.e $7/3 = 2$)

You can check this pattern from the above example.

For sorting an array having 3 elements, we need, 3 ~~computations~~ comparasions. so,

no. of computation = 3 × 1 × 1 × $[n/3]$

- 3 → sorting array.
- 1 → finding median from sorted array.
- 1 → finding median for sorted arrays.
- $[n/3]$ → no. of chunks.

" = n. for first time.

" = $3 \times n/9$. for second time.

" = $3 \times n/27$ for third time.

Total no. of computations = $n + n/9 + n/27 + \ldots$ upto some finite no. of times.

" $\leq n + n/3 + n/9 + \ldots$ upto infinite no.

" $\leq n(1 + 1/3 + 1/9 + \ldots)$

$\leq n\left(\frac{1}{1-1/3}\right)$

" $\leq \frac{3n}{2} = O(n)$.

2) Let $T(n)$ denote the running time. The median of 7 elements can be found in 14 comparisons.

∴ for finding medians of the subgroups created, we have. $\frac{14n}{7}$ i.e $2n$ comparisons.

Now, we use select algorithm over the ~~same~~ input of size $n/7$, which would take $T(n/7)$ comparisons

In next step, we split input into two subgroups S and L. This can be done in $n-1$ comparison. As the median of medians has $\left[\frac{1}{2}\left(\frac{n}{7}\right) - 2\right]$ below and among itself.

∴ size of S (~~same~~ and L. are $n - 4\left[\frac{1}{2}\left(\frac{n}{7}\right) - 2\right]$

$= n - \frac{2n}{7} + 8$.

$= \frac{5n}{7} + 8$. (∵ group size is 7).

Now, we apply another round of Select algorithm over the element which are present in S or in L. On doing this, we get the recurrence relation as -

$$T(n) \leq 2n + T(n/7) + n-1 + T(5n/7 + 8)$$

4). The idea is that we can use the concept of counting sort. Here is the algorithm.

1) Compute the length of array and then find $\sqrt{\text{length of array}}$. Time $O(\sqrt{n})$ where n is length of array.

2) Find the maximum element (maxa) in the array and create an array of length (maxa + 1) i.e. 1 + maximum of array. Then, initialise it to zero. Name this new array as fre. This is O(n).

3) Loop through all the elements in the given array and store the frequency of the element in fre. This is also O(n). This can be done by the following code —

```
int fre[maxa+1] = {0};
for (int i=0; i<n; i++)
{
    fre[arr[i]]++;
}
```

where arr is the array.

4) Loop over the array fre until we reach the k^{th} element. i.e. O(n)

5) Print that value.

Code for the 4th step —

```
for (int num = 1; num ≤ maxa; num++)
{
    if (fre[num] > 0)
    {
        smallest = smallest + fre[num];
    }
    if (smallest >= k).
    {
        print num;
    }
}.
```

So, total time complexity is O(n).

5) Algorithm for sorting n elements in $O(n \lg n)$ time —

a) For an array A, $n \geq 1$ elements, swap the median element found by blackbox, say A[me] with the middle element of A, thus creating a left and right half of the arrays.

b) Then, swap the elements in the left half, that are larger than A[mid] with elements in the right half that are smaller or equal to A[mid].

c) This subdivides the original array into 2 distinct subarrays of about half the size that each need to be sorted.

d) Apply this algorithm recursively on each subarray.

Pseudo code —

medianSort (A, left, right)

1) if (left < right) then
2) find median value A[me] in A[left, right]
3) mid = $\lfloor$(right + left)/2$\rfloor$
4) swap A[mid] and A[me].
5) for left = 0 to mid - 1 do
6) if (A[i] > A[mid]), then
7) - find $A[k] \leq A[mid]$ where $k > mid$.
8) swap A[i] and A[k].
9) medianSort (A, left, mid - 1)
10) medianSort (A, mid+1, right)

end

Time complexity —

step (2) - O(n).
step (5→8) - O(n).
no. of recursion = lg n.
So, $T(n) = (O(n) + O(n)) \cdot \lg n$.
$= O(n \lg n)$

(take example from binary tree formed from this also)