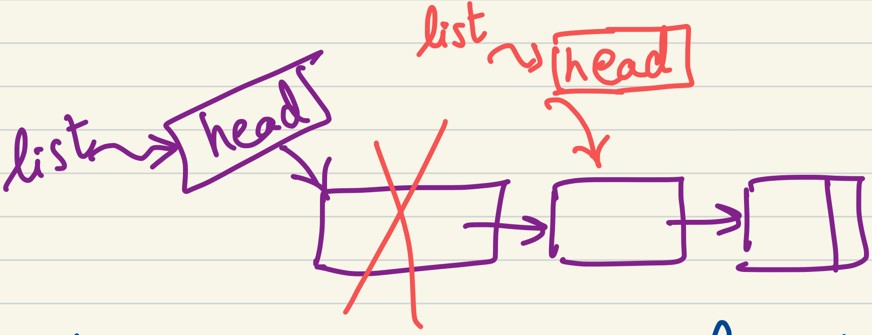
APS 105 Lecture 33 Notes

Last time: insertAtBack, insertInOrderedList, deleteAtFront, deleteAtBack operations on linked lists.

Today: deleteAllNodes, deleteFirstMatch, deleteAllMatches, then we introduce sorting algorithms such as selection sort.



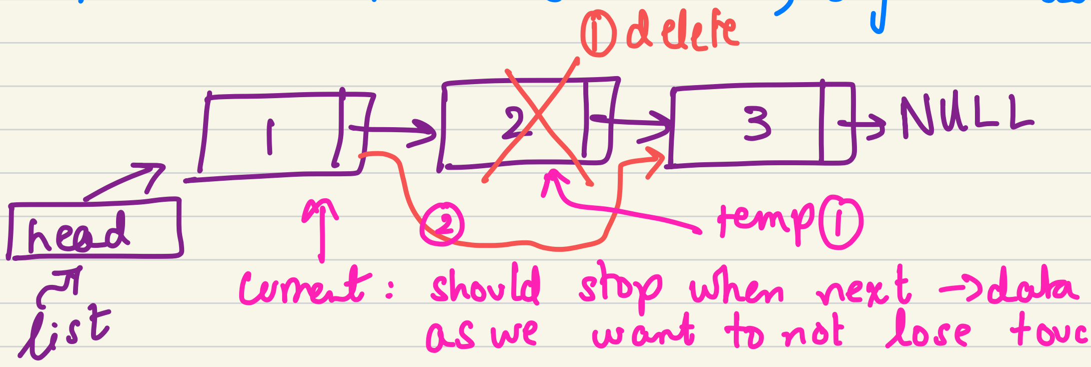
Delete all nodes, return # of nodes you deleted

```
int deleteAllNodes(LinkedList *list){
    int numDeleted = 0;
     while(list -> head != NULL) {

              deleteFront(list);
              numDeleted++;
     }
    list -> head = NULL;
    return numDeleted;
```

← unrequired as deleteFront does this

Delete 1st Node observed, e.g. with data = 2



current : should stop when next ->data is = 2 as we want to not lose touch with previous node

① Node * temp= current ->next ③ free (temp)
② current ->next= temp->next

```
bool deleteFirstMatch( LinkedList * list, int value){
        if(list -> head= = NULL)
                return false;
        if ( list -> head ->data = =value){
                deleteFront (list);
                return true;
        }
    Node * current = list -> head;
    while ( current -> next != NULL &&
            current ->next-> data != value ){
            current = current -> next;
    }
    → current points to node before found node OR
      last node
    if (current ->next != NULL){ Not last node
        Node * temp= current ->next;
        current -> next = temp ->next; →Skip node to be deleted
        free (temp);
        return true; → we deleted the node yay!
    }
    return false;
}
```

bool → true if deleted, false if not deleted

Nothing to be made if list is empty

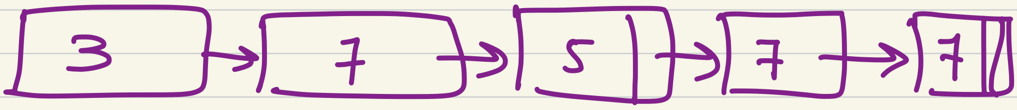
if 1st node is to be deleted

when node not found!

General Case or if no match is found

Delete all matches in a linked list, count # of deleted nodes

Eg.



```
int deleteAllMatches(LinkedList *list, int value){

        int numOfDeleted = 0;
        while (deleteFirstMatch(list, value)){
            numOfDeleted++;
        }

        return numDeleted;
}
```

Sorting Algorithms

- → Sort #, strings, char
- → in any data structure, arrays or linked lists
- → in ascending or descending order
- → we sort "in-place" to avoid creating new list.

Why sorting?

To form a phonebook, dictionary, iPod sorts your playlist

Insertion Sort:

2	9	6	5	1	7

① Sort 1st 2 elements (already sorted)   2, 9

② Sort 3rd element with respect to the previous 2 elements   2 6 9   insert 6 in between 2 and 9

③ Sort 4th element w.r.t. to the previous 3 elements.

2 5 6 9   insert 5 when it's between 2 & 6

④ " 5th " " " " 4 elements

1 2 5 6 9

⑤ " 6th " " " " 5 elements

1 2 5 6 7 9   ✓ Sorted array

I- Every step we "insert" a number to the previously sorted small array — we do so in 1 loop.

II- We have 1 loop to loop over all elements in the array we want to "insert"

```
void  insertSort( int list[], int size ){
        for(int key =1 ; key < size ; key ++)

            int item= list [key];
            int ind= key;
            while (ind>0 && item < list[ind-1]){
                list[ind]=list[ind-1];
                ind--;
            }
            list[ind] = item;
}
```

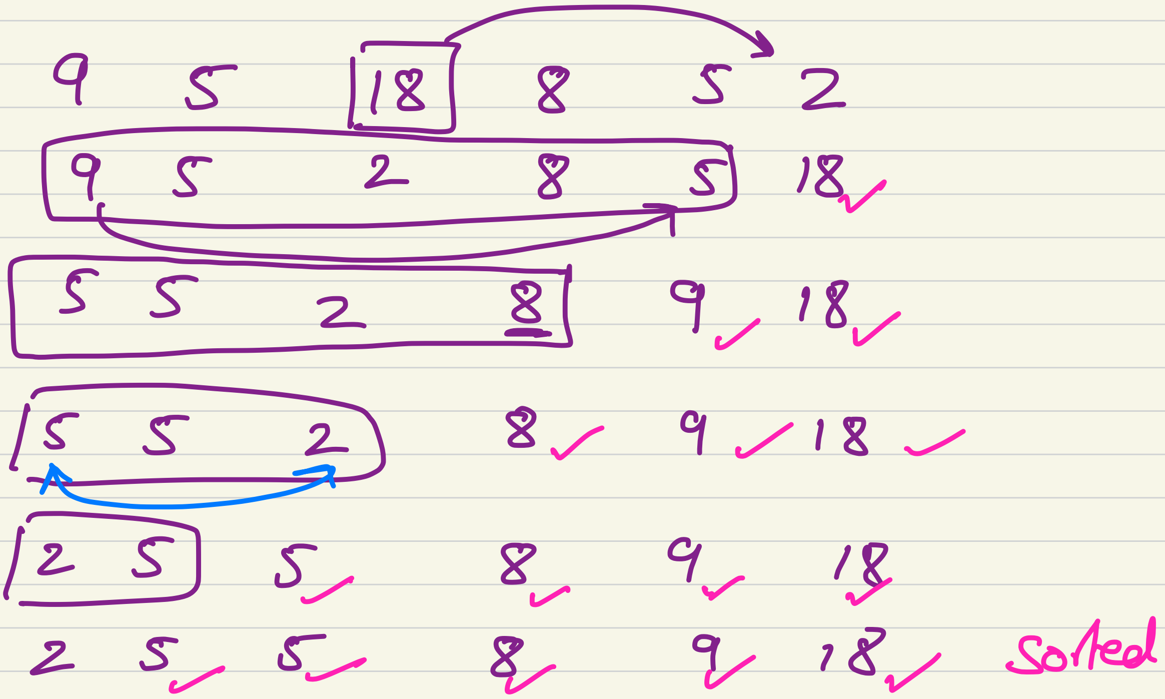
loop over the elements to insert them in the previously sorted array

We insert 1 element into its position in the previously sorted subarray

keep shifting item at list[top] 1 to the left till it fits in its position / or till you see list [i-1] is smaller than i

Selection Sort

- Search entire array to find largest & move it to the end (swap with end).
- Then search for largest element excluding last element, since itis in the correct place



9 5 18 8 5 2

9 5 2 8 5 18 ✓

5 5 2 8 9 ✓ 18 ✓

5 5 2 8 ✓ 9 ✓ 18 ✓

2 5 5 ✓ 8 ✓ 9 ✓ 18 ✓

2 5 ✓ 5 ✓ 8 ✓ 9 ✓ 18 ✓ Sorted

How many times did we look for the largest #?

size of array - 1

How much work in each time we search?

1st time : 6
2nd time : 5
⋮
last time : 2

```
void   selectionSort( int list[], int n){

        int top, large Loc, i;

        for ( int top = n-1; top > 0; top--){

            Large Loc = 0; //assume 1st element is
                                           largest
          for( int i = 1; i<=top; i++)
                 if( list[i] > list[large Loc]){
                          Large Loc = i;
                 }
          }
          //swap largest element found with top.
          to be placed in right place
          int temp = list[top];
          list[top] = list[Large Loc];
          list[Large Loc] = temp;
     }
   return;
}
```