(1) Insertion sort on an already sorted array does best case work. it performs one comparision per element & no swaps. → O(n) comparisons & (0) swaps.

Shell sort (with shell's original sequence: gaps = $\frac{n}{2}$, $\frac{n}{4}$ ---1) runs a sequence of gapped insertion-sort passes, for an already sorted array each gapped pass finds that each gapped subarray is already sorted, so the gapped insertion operation do minimal work.

- Shell's early large gaps move elements distances in one pass; that reduces the no. of adjacent swaps later so if the array is almost sorted except for a few elements far from their final position shell moves them quickly, making later passes cheap.

(2) If the keys are multiples of 10 i.e., (10, 20, 30, ---) then $h(k) = k \bmod 10 = 0$ for all those keys.

This results in clustering, where all these keys are stored in the same bucket of the hash table. Instead of being distributed evenly, the keys are concentrated in a single location, leading to large no. of collision.

Due to this extreme clustering, finding a specific key requires searching through all the keys that have been clustered in that single bucket.

The access time to this becomes O(n) where n = no. of input keys.

3] → #include <stdio.h>
#include <stdlib.h>

```
Struct PolyNode {
    int coeff ;
    int exp ;
    Stuct PolyNode * next ; }
Stuct PolyNode * create (int c, int e) {
    Struct PolyNode * node = (Struct PolyNode * ) malloc (sizeof (
        Struct PolyNode)) ;
    node → coeff = c ;
    node → exp = e ;
    node → next = NULL ;
    return node ; }
// Assume 2 polynomials are sorted in decreasing
   // exponent
Stuct PolyNode * add poly. (Stuct PolyNode * P1 ,
    Stuct PolyNode * P2 ) {
    Struct PolyNode dummy ;
    Stuct PolyNode * tail = & dummy ;
    dummy. next = NULL ;
while (P1 ; = NULL && P2 ! = NULL ) {
    if (P2 → exp == P2 → exp) {
```

```
int s = p1 -> coeff + p2 -> coeff;
  if (s != 0) {
     tail -> next = create (s, p1 -> exp);
      tail = tail -> next;
  }
  p1 = p1 -> next;
  p2 = p2 -> next;
} else if (p1 -> exp > p2 -> exp) {
    tail -> next = create (p1 -> coeff, p1 -> exp);
      tail = tail -> next;
       p1 = p1 -> next;
    } else {
        tail -> next = create (p2 -> coeff, p2 -> exp);
        tail = tail -> next;
         p2 = p2 -> next;
  }}
 while (p2 != NULL) {
     tail -> next = create (p2 -> coeff, p2 -> exp);
      tail = tail -> next;
     p2 -> p2 -> next;
      }
    return dummy.next;
}
```

4] - The diff. b/w Singly, Doubly & Circular linked list.

5]

Property	Singly	Doubly	Circular
Structure	node → next only	prev & next	tail is connected to head.
Insertion at head	O(1)	O(1)	O(1) if head is known.
Insertion at tail	O(n)	O(1) if tail is known otherwise O(n)	O(1)
Deletion at head	O(1)	O(1)	O(1)
Traversal	Only forward Traverse	forward & Backward	It iterates continously if condition not given.
Memory overhead	1 pointer & Data	2 pointers & Data	Depend if singly or Double.
Preffered When	Low memory, only Forward traversal, simple stacks	frequent deletion & Bidirectional traversal	Round Robin, Scheduling Buffering.
Example use cases	Implementing adjencecy list in grap, stacks	Implement LRU cache, editor undo Redo, deque	Implement Circular queue round robin.

5] Reverse a singly linked list

Pseudo_code

```
prev = NULL
Current = head
While ( current != NULL) {
        next = current -> next
        current -> next_ prev
        prev = current
        current = next
    }
head = prev.
```

C-Code

```
Struct Node * reverseList (Struct Node **head) {
    Struct Node * prev = NULL, * cur = * head;
        Struct Node * next = NULL ;
        While ( cur != NULL ){
            next = cur -> next ;
            cur -> next = prev;
             prev = cur ;
                cur = next ;
            }
            return prev;
    }
```

6] Quadratic Probing formula:

$h(k,i) = (h'(k) + i^2) \bmod m$ where $h'(k)$ is the initial hash & i.e is the probe number.

for prime table size quadratic probing can guarantee to probe at least half of the slots before repeating. If the table is less than or equal to half full (load factor ≤ 0.5) quadratic probing guarantees that an empty slot will be found, as an empty slot is guaranteed to be visited within first $\frac{m}{2}$ probes.

Ex- If size $m = 11$.

$h(k,i) = (h(k) * i^2) \bmod 11$ the probe sequence starting at $h'(k) = 0$ would be 0, 1, 4, 9, 5, 3, 3, --- (after 11th probe it starts repeating). This sequence only visits 0, 1, 3, 4, 5, 9 which are only 6 not all 11.

7) $m - 3 = 11 - 3 = 8$ (Assuming Base value = 0).

hash fn $\Rightarrow (2 + 3 * k) \bmod 8$

for 43 = $(2 + 3(43)) \bmod 8 = 3$

for 165 = $(2 + 3(165)) \bmod 8 = 1$

for 62 = $(2 + 3(62)) \bmod 8 = 4$

for 123 = $(2 + 3(123)) \bmod 8 = 3$.

By using linear Probing (123) will go no 5 position.

for 142 value would be $(2+3(142)) \bmod 8 = 4$
using Linear Probing

142 will be inserted at 6 position.

8] $h(k) = ((9k^2) + 3k + 17) \bmod 23) \bmod 11$

Collision:

$h_i(k) = (h(k) + i^2) \bmod 11.$

a]

index

$h(21) = 1$
$h(82) = 3$
$h(43) = 9$
$h(87) = 10$
$h(54) = 8$
$h(63) = 2$
$h(72) = 4$

b) $h(99) = 8$

$(8+1) \bmod 11 = 9$
$(8+4) \bmod 11 = 1$
$(8+9) \bmod 11 = 6$

99 will be placed at 6th position.

9] $h_1(k) = (5k^2 + 3k + 7) \bmod 17$

$h_2(k) = 1 + ((13k + 11) \bmod 16)$

after collision:

$(h_1(k) + i h_2(k)) \bmod 17$

$h_1(25) = 11$ → placed at 11

$h_2(25) = 7$

$h_1(47) = 7$ → placed at 7

$h_2(47) = 9$

$h_1(47) = 7 \rightarrow$ placed at 7

$h_2(47) = 9$

$h_1(89) = 14 \rightarrow$ placed at 14

$h_2(89) = 7.$

$h_1(106) = 14$, $h_2(106) = 10$

occupied

$i = 1 \Rightarrow (14 + 10 = 124) \bmod 17$

$= 7$

occupied

$i = 2 \Rightarrow (14 + 20) \bmod 17 = 0$

placed at 0

$h_1(132) = 7 \rightarrow$ occupied

$h_2(132) = 8$

$i = 1 : (7 + 8) \bmod 17 = 15$

placed at 15

0	106
1	205
2	
3	
4	
5	
6	
7	47
8	310
9	
10	
11	25
12	
13	
14	89
15	132
16	

$h_1(205) = 15$ index 15 occupied

$h_2(205) = 3$

$i = 1 \Rightarrow (15 + 3) \bmod 17 = 1$

$h_1(310) = 14 \rightarrow$ occupied

$h_2(310) = 14$

$i = 1 \Rightarrow (14 + 14) \bmod 17 = 11 \rightarrow$ occupied

$i = 2 \Rightarrow (14 + 28) \bmod 17 = 8$

6] Expected no. of probes of unsuccessful search

$$\Rightarrow \frac{1}{1-\alpha} = \frac{1}{1-7/17} \Rightarrow \frac{17}{10} = \underline{\underline{1.7}}$$

{load factor(α) = $\frac{7}{17}$}

10]
```
#include <stdio.h>
#include <stdlib.h>
Struct node {
        int data;
        Struct node * next ; } ;
Struct node * Create (int data) {
        Struct node * new = (struct node *) malloc (sizeg
                        (Struct node)) ;
            new→ data = data;
            new → next = NULL;
            return new ; }
Void insert at begin (struct node ** head, int data) {
        Struct node *new = Create (data);
            if (new == NULL) { printf ("Error in malloc ");
                    return ; }
            if (*head == NULL) {
                    *head = new ;
                        return ; }
                New → next = *head
                    *head = new ; }
```

```
Void print (Struct node *head) {
    Struct node *temp = head;
    While (temp != NULL) {
        printf (" %d ", temp->data);
        temp = temp -> next;
    }

int main() { Struct node *head = NULL;
        insertatbegin (&head, 12);
        insertatbegin (&head, 13);
        print (head);
}
```

b) If malloc fails on the 4th Node the first three allocation succeeded remain valid. Those 3 nodes Continue to exist.

You can still added nodes to the list as the 4th node is not the head.

C-code

11]

```
# include <stdio.h>
# include <stdlib.h>
Struct node {
        int data ;
        Struct node * next;
        Struct node * prev;
}
Struct node * Create (int data) {
Struct node * new = (Struct node *) malloc (sizeof (Struct node) };
        new -> data = data;
        new -> next = NULL;
        new -> prev = NULL;
          return new; }
Void insertatend (Struct node ** head , int data) {
    Struct node *new = create (data);
        if (*head = NULL) {
              * head = new;
                 return ; }
          Struct node * temp = * head ;
        while (temp -> next != NULL) temp=temp -> next;
            temp -> next = new ;
             new -> prev = temp;
Void Print (Struct node * head ) {
      Struct node * temp = head;
          while (temp) { printf ("%. d", temp -> data );
                    temp = temp -> next ;
        }}
```

```
Struct Node * Merge (Struct Node * first, Struct Node * second)
{
    if (! first) return second;
    if (! second) return first;
  Struct node * head = NULL;
  Struct Node * tail = NULL;
   if (first -> data <= second -> data) {
            head = tail = first;
             first = first -> next; }
     else { head = tail = second;
              second = second -> next ; }
     while (first && second) {
         if (first -> data <= second -> data) {
                tail -> next = first;
                first -> prev = tail;
                 tail = first;
                first = first -> next; }
         else { tail -> next = second;
                 second -> prev = tail;
                  tail = second;
                 second = second -> next;
                }
              }
    if (first) { tail -> next = first;
                 first -> prev = tail;
      } else if (second) { tail -> next = second
                           second -> prev = tail;
                         }
                  return head;
                }
```

```
int main() {
    Struct Node * list1 = NULL;
    Struct Node * list2 = NULL;

        insertatend (&list1, 1);
        insertatend (&list1, 4);
        insertatend (&list1, 7);

       insertatend (&list2, 2);
       insertatend (&list2, 3);
       insertatend (&list2, 5);

Struct Node *merged = merge (list1, list2);
      print (merged);

}.
```

13] a Linked list is more suitable when size is highly dynamic & unpredictable as it supports dynamic growth without requiring large contiguous block of memory (using malloc).

Arrays are of fixed sizes. Dynamic resizing is difficult in array.

b] Random accessing is easier in array & it take O(1). while in linked list it takes O(n) where n is the no. of inputs.

Random Insertions & deletion is much easier in linked list & takes O(1) with just update of pointers.

While in Arrays Insertion & deletion takes shifting of elements with O(n) operations.

Therefore Linked list is a better option for Data Buffer System.
