1)

These are some of the advantages of linked lists over array:

MEMORY: Array should be stored in contiguous memory locations while it is not necessary to store in contiguous memory locations for linked lists.

INSERTION: As the array elements are stored in contiguous memory locations, we need to create space for insertion of new elements. Where as in linked lists we just have to change the pointer address.

DELETION: In arrays we have to delete the element from the given location and then shift all successive elements up by 1 location. Where as in linked lists we just have to change the pointer address.

i) Arrays are fixed in size while linked lists are dynamic in size.

ii) Deletion in linked list is easy when compared to deletion in array. AS there is no need of moving items in linked list.

iii) Inserting a new element is difficult in arrays when compared to linked list. As the array should be resized before doing it.

iv) Array assumes every element is exactly same size where as in linked list we can store data of different sizes.

==========

2)

a) Given code,

```
        public static void add( List<Integer> lst1, List<Integer> lst2)
{
  for ( Integer x : lst1 )
    lst2.add(0, x);       // add to front
}
```

If an ArrayList is passed for lst1 and lst2. The Big-O running time of the above code is O(N^2).

Because adding ArrayList to the front requires moving the other elements up by 1 location which takes O(N) time and there are N iterations. So in total O(N^2).

b) Given code,

```
        public static void add( List<Integer> lst1, List<Integer> lst2)
  {
    for ( Integer x : lst1 )
      lst2.add(0, x);       // add to front
  }
```

If a LinkedList is passed for lst1 and lst2. The Big-O running time of the above code is O(N).

Because adding LinkedList to the front requires O(constant) with N iterations. So in total O(N).

==========

3)

a) Given code,

```
  public static void erase( List<Integer> lst )
  {
    Iterator<Integer> itr = lst.iterator();

    while ( itr.hasNext() )
    {
      Integer x = itr.next();
      itr.remove();
    }

  }
```

If an ArrayList is passed for lst. The Big-O running time of the above code is O(N^2).

Because removal takes O(constant), shifting takes O(N) and there is a loop of N iterations. So in total O(N^2).

b) Given code,

```
  public static void erase( List<Integer> lst )
```

```
  {
    Iterator<Integer> itr = lst.iterator();

    while ( itr.hasNext() )
    {
      Integer x = itr.next();
      itr.remove();
    }

  }
```

If a LinkedList is passed for lst.The Big-O running time of the above code is O(N).

Because removes take O(constant) and iteration of O(N). Hence in total O(N).

==========

4)

a) Given code,

```
  public static int Count( List<Integer> lst1, List<Integer> lst2)
  {
    Iterator<Integer> itr1 = lst1.iterator();

    int count=0;
    while ( itr1.hasNext() )
    {
      Integer x = itr1.next();
      Iterator<Integer> itr2 = lst2.iterator();
      while ( itr2.hasNext() )
        if ( x.equals( itr2.next()) )
          count++;
    }
```

```
    return count;
  }
```

and lst1 has N items, and lst2 has N items.

If two ArrayLists are passed for lst1 and lst2. The Big-O running time of the above code is O(N^2).

Because the code fragment has just traversal with two loops. One loop inside the other. So in total O(N^2).

b) Given code,

```
  public static int Count( List<Integer> lst1, List<Integer> lst2)
  {
    Iterator<Integer> itr1 = lst1.iterator();

    int count=0;
    while ( itr1.hasNext() )
    {
      Integer x = itr1.next();
      Iterator<Integer> itr2 = lst2.iterator();
      while ( itr2.hasNext() )
        if ( x.equals( itr2.next()) )
          count++;
    }

    return count;
  }
```

and lst1 has N items, and lst2 has N items.

If two Linkedlists are passed for lst1 and lst2. The Big-O running time of the above code is O(N^2).

Because the code fragment has just traversal with two loops. One loop inside the other. So in total O(N^2).

==========

5)

a) Given code,

```
 public static int calc( List<Integer> lst )
    {
      int count = 0;
      int N = lst.size();

      for ( int i=0; i<N; i++)
      {
        if (lst.get(i) > 0)
          sum += lst.get(i);
        else
          sum += lst.get(i) * lst.get(i);
      }
      return sum;
    }
```

If an ArrayList is passed for lst. The Big-O running time of the above code is O(N).

Because the code fragment takes O(constant) for get and has a loop with iteration N i.e O(N). So in total O(N).

b) Given code,

```
 public static int calc( List<Integer> lst )
    {
      int count = 0;
```

```
    int N = lst.size();


    for ( int i=0; i<N; i++)
    {
      if (lst.get(i) > 0)
        sum += lst.get(i);
      else
        sum += lst.get(i) * lst.get(i);
    }
    return sum;
  }
```

If a LinkedList is passed for lst. The Big-O running time of the above code is O(N^2).

Because the code fragment takes O(N) for get and has a loop with iteration N i.e O(N). So in total O(N^2).

==========

6)

a) Given a Java method receives a List<Integer> and reverses the order of the items it contains by removing each item from the front of the list, adding each item to a Stack<Integer>, and then removing the items from the stack and inserting each item to the end of the list.

If an ArrayList is passed. For reversal, first we have to remove the elements from front and rearrange in array takes O(N) also we are doing for N iterations. Hence O(N^2) and for pushing and poping in stack takes O(constant). Also for insertion at the end of the list takes O(N).=>O((constant)N^2 + N) => O(N)

So in total it takes O(N^2).

b) Given a Java method receives a List<Integer> and reverses the order of the items it contains by removing each item from the front of the list, adding each item to a Stack<Integer>, and then removing the items from the stack and inserting each item to the end of the list.

If a LinkedList is passed. For reversal, first we have to remove the elements from front takes O(constant) also we are doing for N iterations O(N) and for pushing and poping in stack takes O(constant). Also for insertion at the end of the list takes O(N).

So in total it takes O(N).

==========

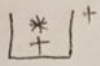
7) Infix to Postfix conversion of a+b*c+(d-e)

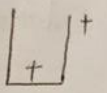
a+b*c+(d-e)

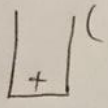
①  o/p: a

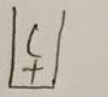
②  o/p: a

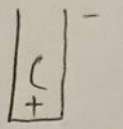
③  o/p: ab

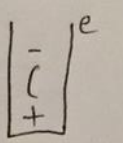
④  o/p: abc

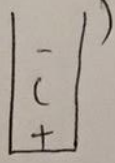
⑤  o/p: abc*

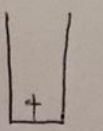
⑥  o/p: abc*+

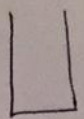
⑦  o/p: abc*+

⑧  o/p: abc*+d

⑨  o/p: abc*+d

⑩  o/p: abc*+de

⑪  o/p: abc*+de-

⑫  o/p: abc*+de-+