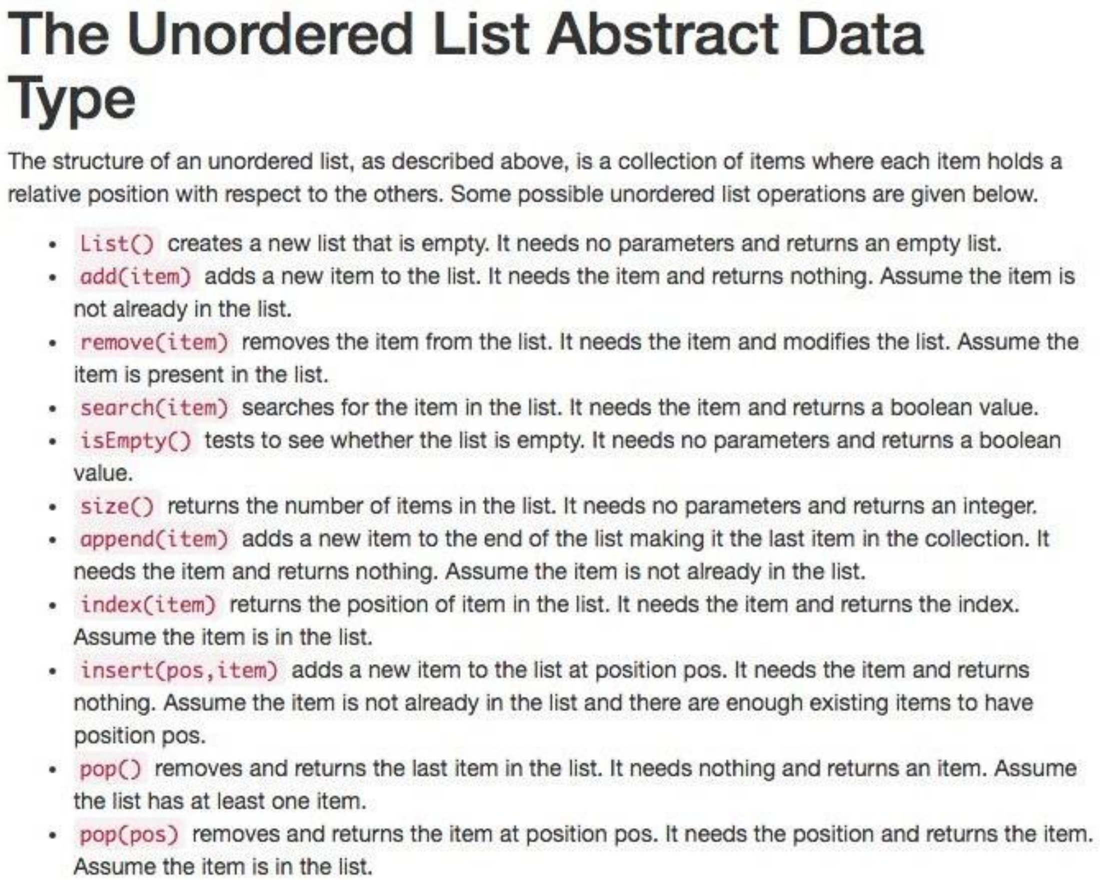
Data Structure Programs

IMPORTANT NOTE - Use Generics to Solve all the Data Structure Programs

1. UnOrdered List

a. Desc -> Read the Text from a file, split it into words and arrange it as Linked List. Take a user input to search a Word in the List. If the Word is not found then add it to the list, and if it found then remove the word from the List. In the end save the list into a file
b. I/P -> Read from file the list of Words and take user input to search a Text
c. Logic -> Create a Unordered Linked List. The Basic Building Block is the Node Object. Each node object must hold at least two pieces of information. One ref to the data field and second the ref to the next node object.
d. O/P -> The List of Words to a File.



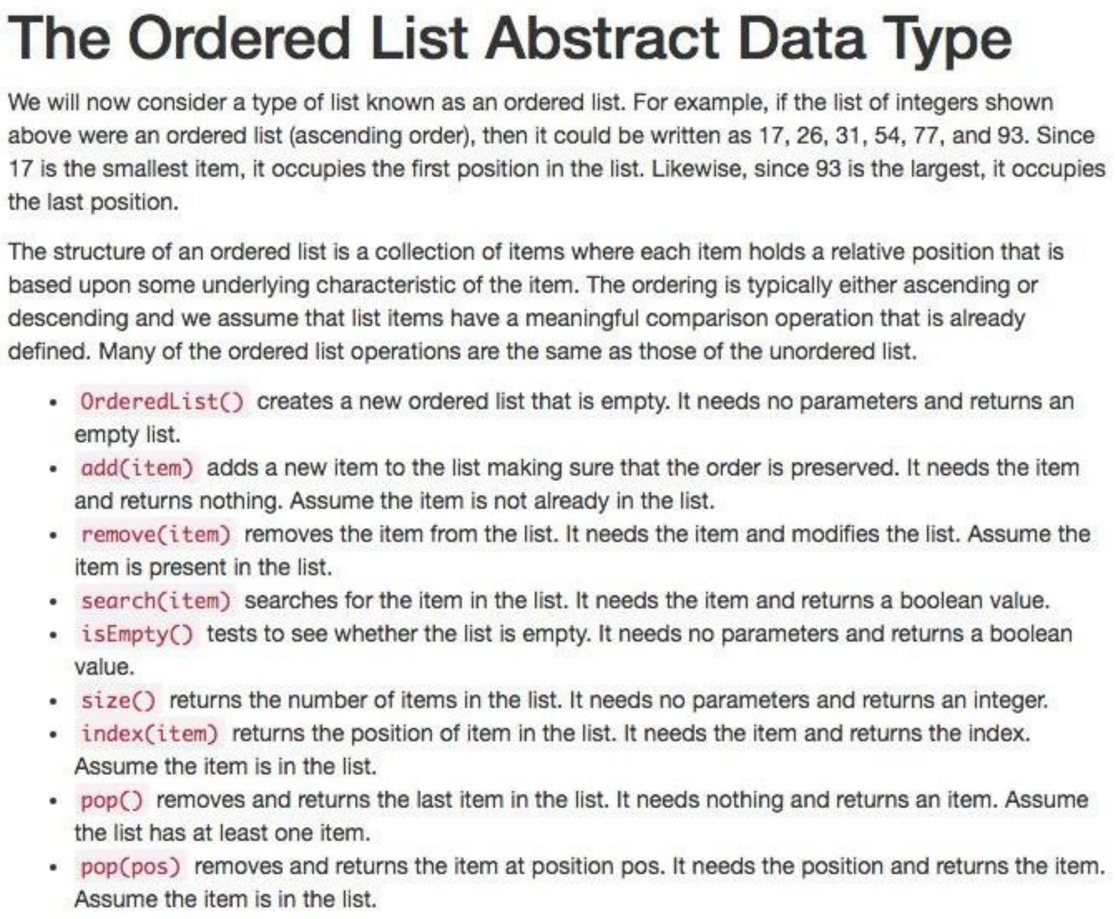
The Unordered List Abstract Data Type

The structure of an unordered list, as described above, is a collection of items where each item holds a relative position with respect to the others. Some possible unordered list operations are given below.

- `List()` creates a new list that is empty. It needs no parameters and returns an empty list.
- `add(item)` adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.
- `remove(item)` removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- `search(item)` searches for the item in the list. It needs the item and returns a boolean value.
- `isEmpty()` tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the list. It needs no parameters and returns an integer.
- `append(item)` adds a new item to the end of the list making it the last item in the collection. It needs the item and returns nothing. Assume the item is not already in the list.
- `index(item)` returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- `insert(pos,item)` adds a new item to the list at position pos. It needs the item and returns nothing. Assume the item is not already in the list and there are enough existing items to have position pos.
- `pop()` removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- `pop(pos)` removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

2. Ordered List

a. Desc -> Read .a List of Numbers from a file and arrange it ascending Order in a Linked List. Take user input for a number, if found then pop the number out of the list else insert the number in appropriate position
b. I/P -> Read from file the list of Numbers and take user input for a new number
c. Logic -> Create a Ordered Linked List having Numbers in ascending order.
d. O/P -> The List of Numbers to a File.



The Ordered List Abstract Data Type

We will now consider a type of list known as an ordered list. For example, if the list of integers shown above were an ordered list (ascending order), then it could be written as 17, 26, 31, 54, 77, and 93. Since 17 is the smallest item, it occupies the first position in the list. Likewise, since 93 is the largest, it occupies the last position.

The structure of an ordered list is a collection of items where each item holds a relative position that is based upon some underlying characteristic of the item. The ordering is typically either ascending or descending and we assume that list items have a meaningful comparison operation that is already defined. Many of the ordered list operations are the same as those of the unordered list.

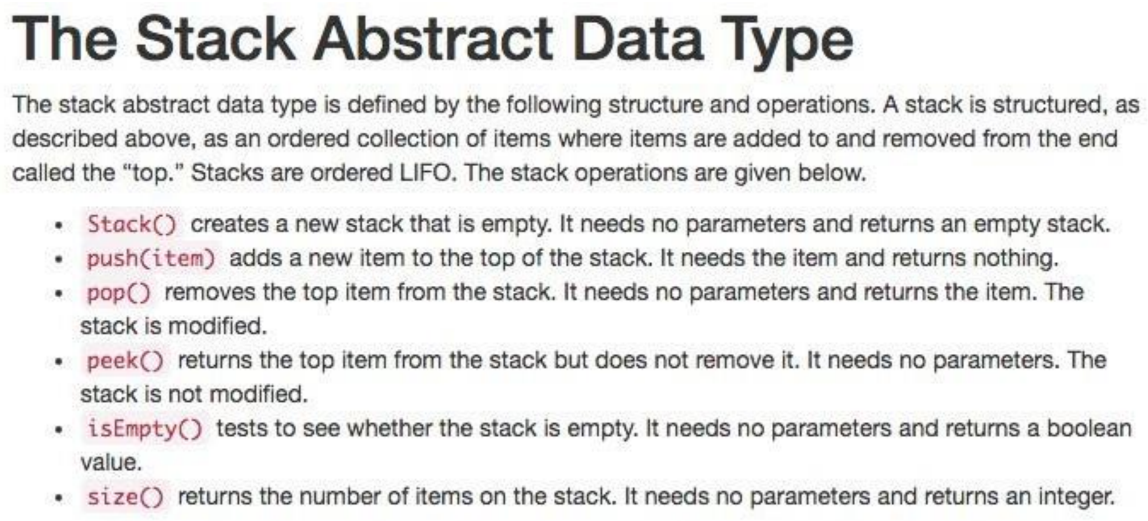
- `OrderedList()` creates a new ordered list that is empty. It needs no parameters and returns an empty list.
- `add(item)` adds a new item to the list making sure that the order is preserved. It needs the item and returns nothing. Assume the item is not already in the list.
- `remove(item)` removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- `search(item)` searches for the item in the list. It needs the item and returns a boolean value.
- `isEmpty()` tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the list. It needs no parameters and returns an integer.
- `index(item)` returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- `pop()` removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- `pop(pos)` removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

3. Simple Balanced Parentheses

a. Desc -> Take an Arithmetic Expression such as (5+6)∗(7+8)/(4+3)(5+6)∗(7+8)/(4+3) where parentheses are used to order the performance of operations. Ensure parentheses must appear in a balanced fashion.
b. I/P -> read in Arithmetic Expression such as (5+6)∗(7+8)/(4+3)(5+6)∗(7+8)/(4+3)
c. Logic -> Write a Stack Class to push open parenthesis “(“ and pop closed parenthesis “)”. At the End of the Expression if the Stack is Empty then the

Arithmetic Expression is Balanced. Stack Class Methods are Stack(), push(), pop(), peak(), isEmpty(), size()

d. O/P -> True or False to Show Arithmetic Expression is balanced or not.



The Stack Abstract Data Type

The stack abstract data type is defined by the following structure and operations. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the "top." Stacks are ordered LIFO. The stack operations are given below.

- `Stack()` creates a new stack that is empty. It needs no parameters and returns an empty stack.
- `push(item)` adds a new item to the top of the stack. It needs the item and returns nothing.
- `pop()` removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- `peek()` returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- `isEmpty()` tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items on the stack. It needs no parameters and returns an integer.

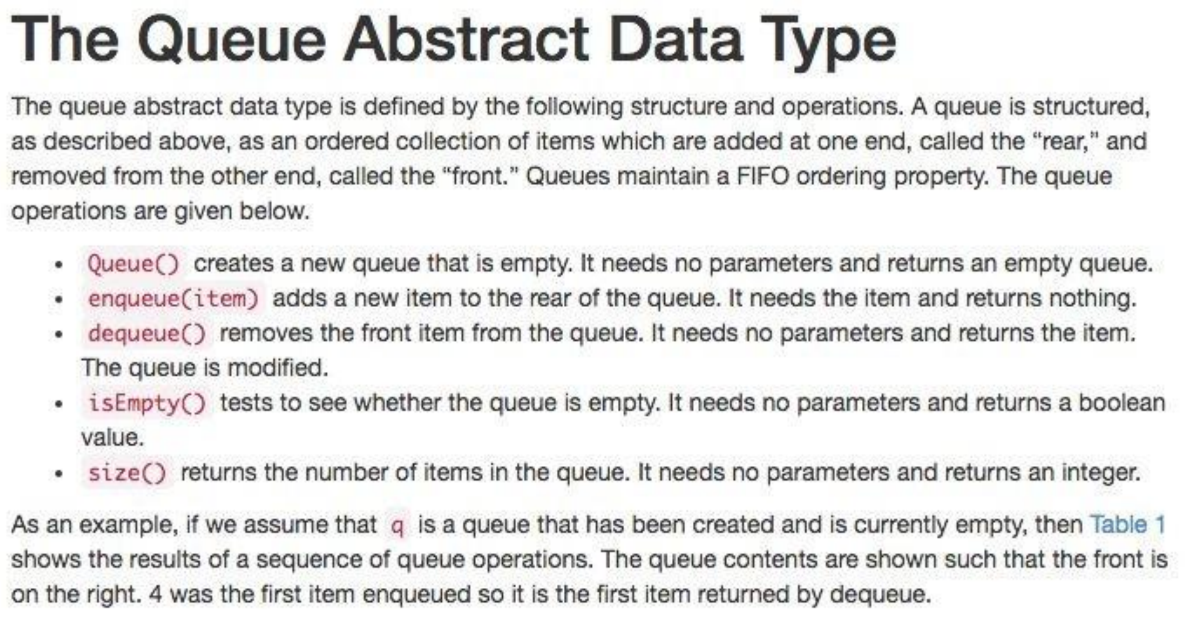
4. Simulate Banking Cash Counter

a. Desc -> Create a Program which creates Banking Cash Counter where people come in to deposit Cash and withdraw Cash. Have an input panel to add people to Queue to either deposit or withdraw money and dequeue the people. Maintain the Cash Balance.

b. I/P -> Panel to add People to Queue to Deposit or Withdraw Money and dequeue

c. Logic -> Write a Queue Class to enqueue and dequeue people to either deposit or withdraw money and maintain the cash balance

d. O/P -> True or False to Show Arithmetic Expression is balanced or not.



The Queue Abstract Data Type

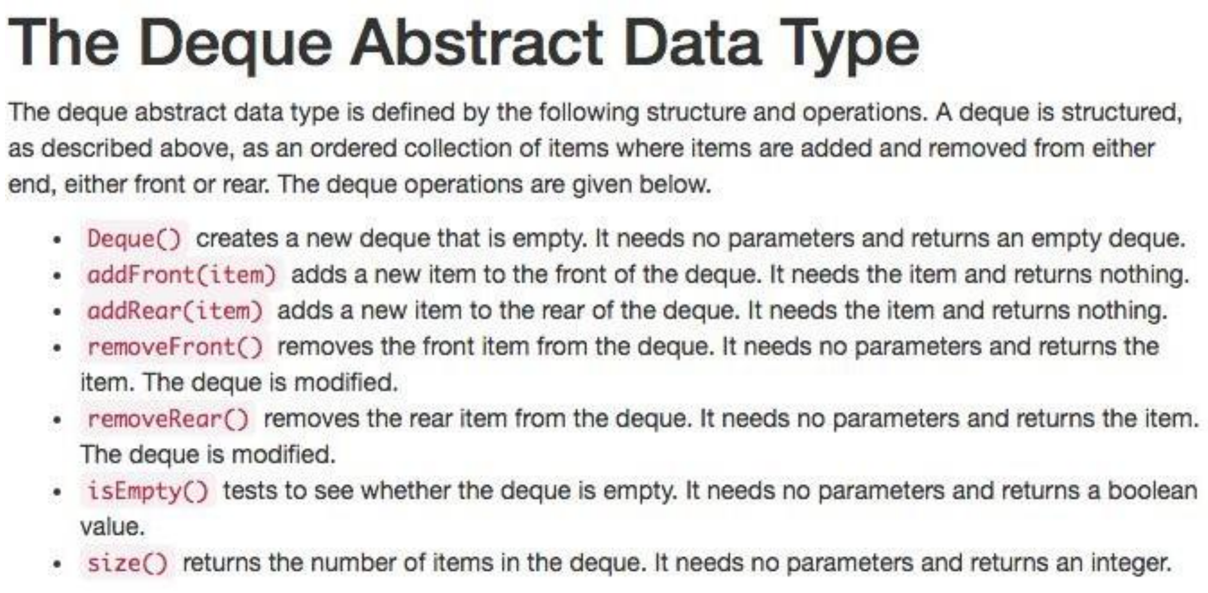
The queue abstract data type is defined by the following structure and operations. A queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front." Queues maintain a FIFO ordering property. The queue operations are given below.

- `Queue()` creates a new queue that is empty. It needs no parameters and returns an empty queue.
- `enqueue(item)` adds a new item to the rear of the queue. It needs the item and returns nothing.
- `dequeue()` removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- `isEmpty()` tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the queue. It needs no parameters and returns an integer.

As an example, if we assume that `q` is a queue that has been created and is currently empty, then Table 1 shows the results of a sequence of queue operations. The queue contents are shown such that the front is on the right. 4 was the first item enqueued so it is the first item returned by dequeue.

5. Palindrome-Checker

a. Desc -> A palindrome is a string that reads the same forward and backward, for example, radar, toot, and madam. We would like to construct an algorithm to input a string of characters and check whether it is a palindrome.
b. I/P -> Take a String as an Input
c. Logic -> The solution to this problem will use a deque to store the characters of the string. We will process the string from left to right and add each character to the rear of the deque.
d. O/P -> True or False to Show if the String is Palindrome or not.



The Deque Abstract Data Type

The deque abstract data type is defined by the following structure and operations. A deque is structured, as described above, as an ordered collection of items where items are added and removed from either end, either front or rear. The deque operations are given below.

- `Deque()` creates a new deque that is empty. It needs no parameters and returns an empty deque.
- `addFront(item)` adds a new item to the front of the deque. It needs the item and returns nothing.
- `addRear(item)` adds a new item to the rear of the deque. It needs the item and returns nothing.
- `removeFront()` removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
- `removeRear()` removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- `isEmpty()` tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the deque. It needs no parameters and returns an integer.

6. Hashing Function to search a Number in a slot

a. Desc -> Create a Slot of 10 to store Chain of Numbers that belong to each Slot to efficiently search a number from a given set of number
b. I/P -> read a set of numbers from a file and take user input to search a number
c. Logic -> Firstly store the numbers in the Slot. Since there are 10 Numbers divide each number by 11 and the reminder put in the appropriate slot. Create a Chain for each Slot to avoid Collision. If a number searched is found then pop it or else push it. Use Map of Slot Numbers and Ordered LinkedList to solve the problem. In the Figure Below, you can see number 77/11 reminder is 0 hence sits in the 0 slot while 26/11 remainder is 4 hence sits in slot 4
d. O/P -> Save the numbers in a file

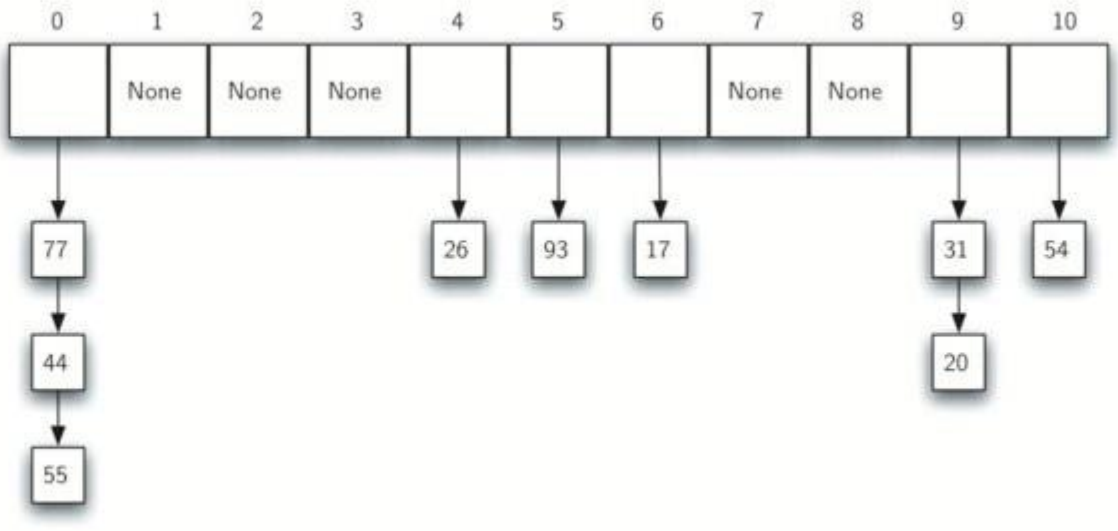


Figure 12: Collision Resolution with Chaining

7. **Number of Binary Search Tree**

   Solve the Problem in the following link https://www.hackerrank.com/challenges/number-of-binary-search-tree.

8. Write a program ***Calendar.java*** that takes the month and year as command-line arguments and prints the Calendar of the month. Store the Calendar in an 2D Array, the first dimension the week of the month and the second dimension stores the day of the week. Print the result as following.

```
% java Calendar 7 2005
 July 2005
 S  M  T  W Th  F  S
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

9. Create the Week Object having a list of WeekDay objects each storing the day (i.e S,M,T,W,Th,..) and the Date (1,2,3..) . The WeekDay objects are stored in a Queue implemented using Linked List. Further maintain also a Week Object in a Queue to finally display the Calendar

   Note - If a particular day has no date then the date is set as empty string and add it to queue.

10. Modify the above program to store the Queue in two Stacks. Stack here is also implemented using Linked List and not from Collection Library
11. Take a range of 0 - 1000 Numbers and find the Prime numbers in that range. Store the prime numbers in a 2D Array, the first dimension represents the range 0-100, 100-200, and so on. While the second dimension represents the prime numbers in that range
12. Extend the Prime Number Program and store only the numbers in that range that are Anagrams. For e.g. 17 and 71 are both Prime and Anagrams in the 0 to 1000 range. Further store in a 2D Array the numbers that are Anagram and numbers that are not Anagram
13. Add the Prime Numbers that are Anagram in the Range of 0 - 1000 in a Stack using the Linked List and Print the Anagrams in the Reverse Order. Note no Collection Library can be used.
14. Add the Prime Numbers that are Anagram in the Range of 0 - 1000 in a Queue using the Linked List and Print the Anagrams from the Queue. Note no Collection Library can be used