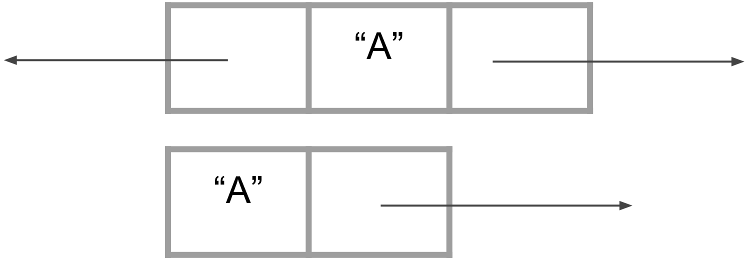
Nodes and Linked Lists

Node

- A node is a basic data structure.
- A node stores:
  - Data
  - One or more pointers to other elements (helps to link nodes together)



Class ListNode

```
public class ListNode{
  private String data;
  private ListNode next;

  public ListNode(String d){ //default next should be null
  }
  public ListNode(String d, ListNode n){ }

  public String toString(){} //Return the string of the data

  public String getData(){} //return the data

  public ListNode next(){} //return the next node

  public String setData(String newdata){} //replace the data, with the newdata, return the original data.

  public void setNext(ListNode n){}

}
```

Pointers Exercise 1

Make a diagram to represent the following code using nodes and pointers (analize one line at the time).

```
ListNode node1 = new ListNode(“a”);

ListNode node2 = new ListNode(“b”);

node1.setNext(node2);

node2.setNext(new ListNode(“c”));

node2 = new ListNode(“d”);

ListNode node3 = new ListNode(“e”, node2);
```

Pointers Exercise 2

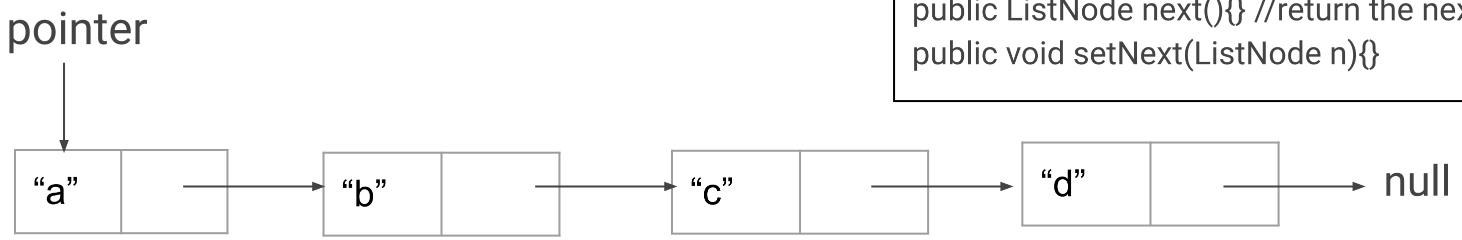
Use the previous diagram and do the following:

```
node2.setNext(node1);

node1 = node3;
```

Pointers Exercise 3

ListNode Class:
public ListNode(String d){}
public ListNode(String d, ListNode n){ }
public ListNode next(){} //return the next node
public void setNext(ListNode n){}

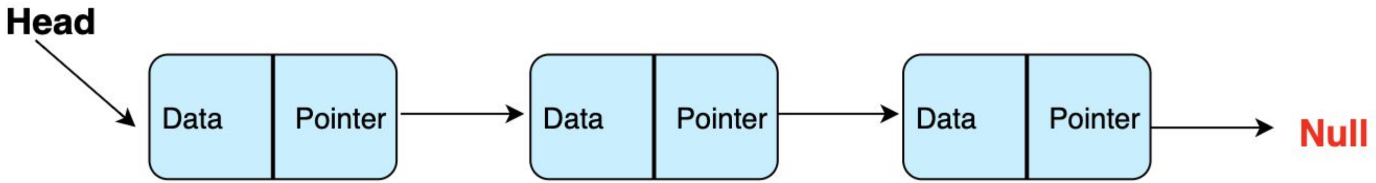


Write a few lines of code to perform the following steps:

1. Create a new `ListNode` variable **x** and instantiate it to a new `ListNode` with value "z" set it to point to the node with the “b” in it.
2. Create a new `ListNode` variable **y** and instantiate it to a new `ListNode` with a value of “e”.
3. Write the code to insert this new `ListNode` **y** between the nodes with the values “b” and the “c”

Linked List

- It is a linear data structure made of a chain of nodes.
- Each node contains a value and a pointer to the next node in the chain.
- It has a Head pointer which points to the first node
- The last element point to null



Linked List

How would you access the linked list chain?

How would you traverse the elements in a linked list?

Class Linked List

How would you access the linked list chain?

We need a node to track the first element of the list.

public ListNode head; // head of the linked list

How would you traverse the elements in a linked list?

Having the first element, we can go over the next elements in the list.

Linked List Characteristics

- The size increases dynamically
- No need to know the size of the element when we create a linked list
- Easy to insert/delete (change pointers)
- Linked list uses extra memory to store links

Types of Link List

Singly: It is a list where each node has data and a reference pointer to its next node.

Doubly: Each node in this list has 3 attributes which are data, next node reference and previous node reference.

Applications of Linked List

- In music players: Your playlist may be created using a linked list.

- Photo gallery applications were you can access the previous/next picture.

- URLs that have previous/next buttons to navigate between pages

Linked List Operations

- Insertion : adds a new element to the linked list
- Deletion : delete existing element form the linked list
- Searching : search for an element by its value in the linked list
- Traversal : traverse all elements starting from head in the linked list

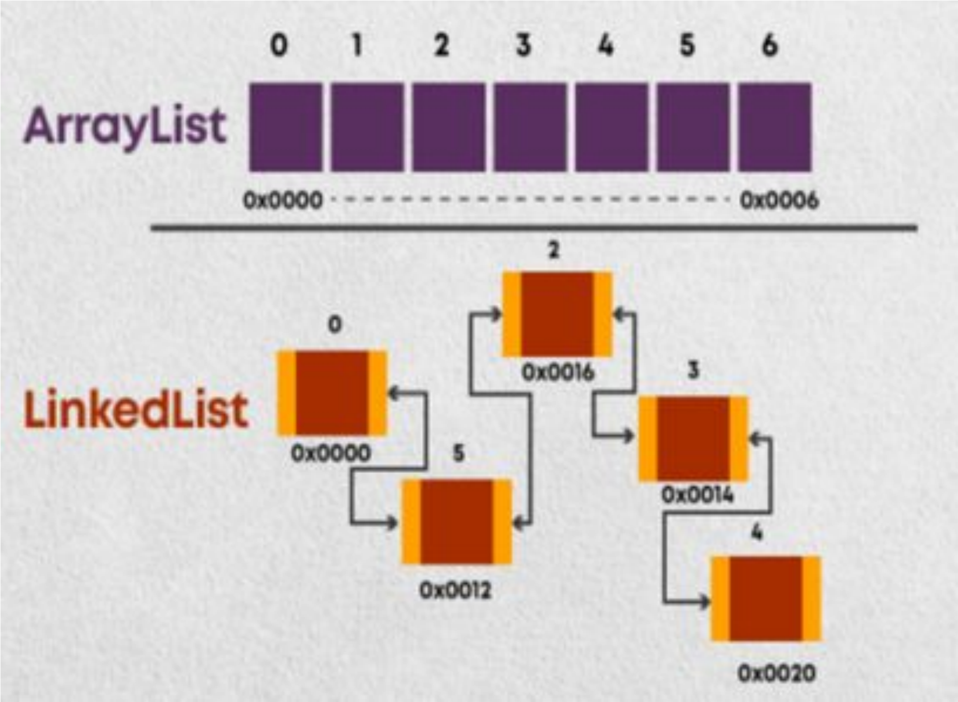
Insert

- Inserting new node at the beginning.
- Inserting new node at the end.
- Inserting new node at random position of the linked list.

Delete

- Deleting node at the beginning.
- Deleting node at the end.
- Deleting node at random position of the linked list.

Difference between ArrayList and LinkedList



Key	ArrayList	LinkedList
Access time	O(1) for random access O(n) for insertion and deletion	O(n) for random access O(1) for insertion and deletion
Memory usage	More memory is used for maintaining the size of the array	Less memory is used since only the elements and pointers are stored
Iteration performance	Fast, since elements are stored in contiguous memory locations	Slower, since elements are not stored in contiguous memory locations
Adding elements	Can be slow if the size of the array needs to be increased to accommodate new elements	Fast, since only pointers need to be updated
Removing elements	Can be slow if elements need to be shifted to fill the gap left by the removed element	Fast, since only pointers need to be updated
Use cases	Best suited for scenarios where random access is required and the list will not be modified frequently	Best suited for scenarios where insertion and deletion are frequent, and random access is not required.