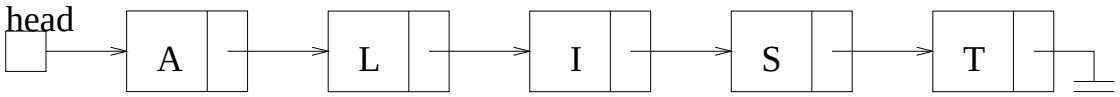
Linked Lists

A **linked list** is a linear set of *nodes* with the following properties:

- Each node has at least two fields, the *key* and the *next*.
- The next field of the ith element "points to" the $(i+1)$th element of the list.
- The first element of the linked list is called the *head*, and the last element the *tail*. The head contains no data, and the next field of the tail points to *NULL*.
- The *key* field contains the data we are actually interested in.
- What exactly is meant by things like "points to" and "NULL" depends on the implementation used.



Linked List Operations

There are several operations that we *must* be able to do on a linked list:

- Insert a node
- Delete a node
- Find an element with key k

There are other operations which might be useful:

- Move an element
- Swap two elements

We will talk about a few of these next, and give one possible way to implement a linked list (in part, anyway).

Linked List: C++ Declaration

```
struct node
  { char key;
    struct node *next; }
struct node *head;
head = new node;
head->next = NULL;
```

- This creates an empty linked list.
- In this implementation, we have the tail point to NULL, which is usually defined to be ‘0’. Thus, if the next field of an element is ‘0’, we know we have reached the tail.
- Here “point to” means that. We really used pointers. We will see shortly that we don’t need to use actual pointers to implement a linked list.

Linked List: JAVA Declaration

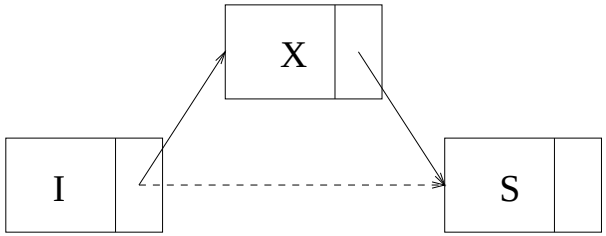
- A JAVA implementation is similar to the C++ one.

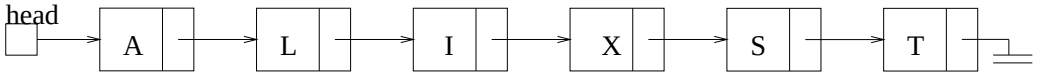
```
public class Node {
    public Node next;
    public char key;
}

public class LinkedList {
    private Node head;
    public LinkedList() {
        head = new Node();
        head.next = NULL;
     }
     // more methods
 }
```

Linked List: Insert

- The simplest insert would place new items at the head or tail. One may also want to insert elements in the middle of the list.
- To insert X between I and S:

```
//C++                       //JAVA
struct node *A;             node A;
A=new node;                 A=new node();
A->key = X;                 A.Key=X;
A->next = I->next;          A.next=I.next;
I->next = A;                I.next=A;
```



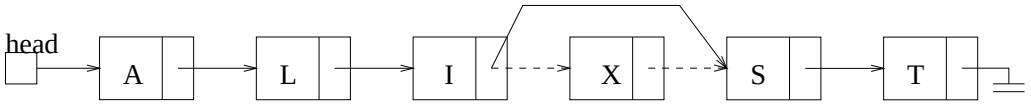


- Only two references need to be changed no matter how long the list is. How does this compare with arrays?

Linked List: Delete

- Deleting a node is very simple. We just need to change one pointer.
- However, we need to know which node points to the node we wish to delete.
- Assuming we know that node i points to node x, we can delete x by:

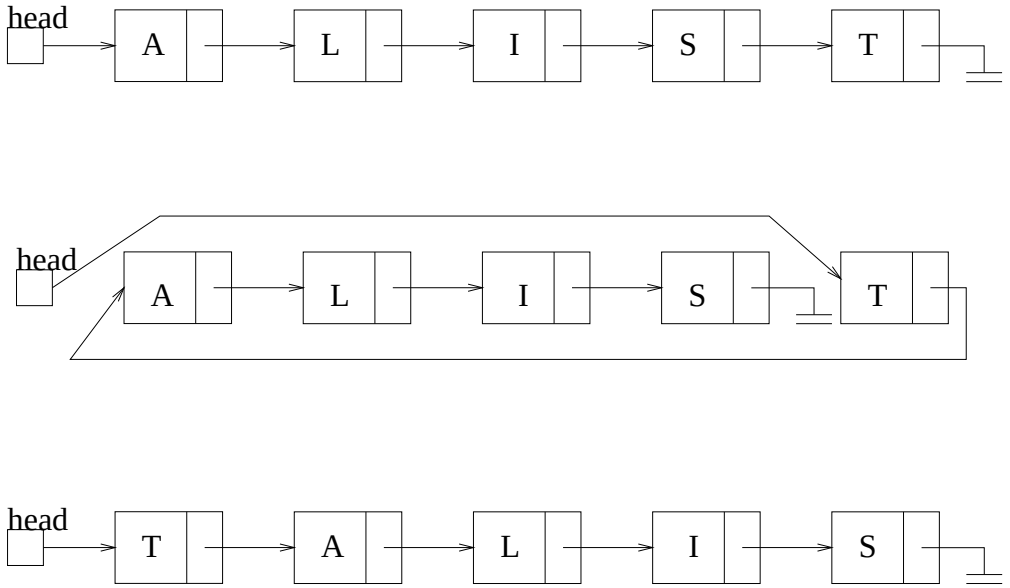
```
// C++                      // JAVA
i->next = x->next;          i.next=x.next;
```



- Only one reference needs to be changed, no matter how long the list is. How does this compare with an array?
- **Note:** There is a slight problem with the C++ code. What is it?

Linked List: Moving a node

- Moving a node consist of a delete operation follow by an insert operation.
- **Example:** Move T from the end of the list to the beginning:



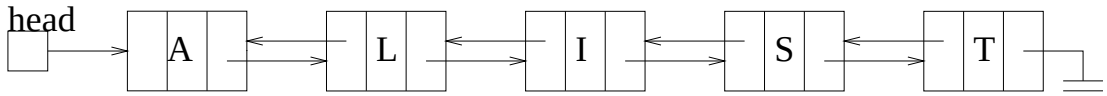
- Again, this requires changing only 3 references, no matter how long the list is. What about with arrays?

Comparison: Linked Lists and Arrays

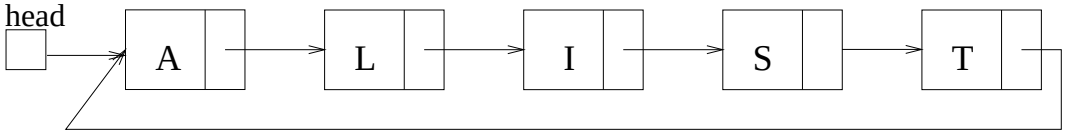
- A linked list can grow and shrink during its lifetime, and its maximum size doesn't need to be specified in advance. In contrast, arrays are always of fixed size.
- We can rearrange, add, and delete items from a linked list with only a constant number of operations. With arrays, these operations are generally linear in the size of the array.
- To find the ith entry of a linked list, we need to follow i pointers, which requires i operations. With an array, this takes only one operation.
- Similarly, it may not be obvious how large a linked list is, whereas we always know the size of an array. (This problem can be eliminated very easily. How?)

Other Linked Lists

- A **doubly linked list** is like a linked list, but each node also has a *previous* field, which points to the nodes predecessor.

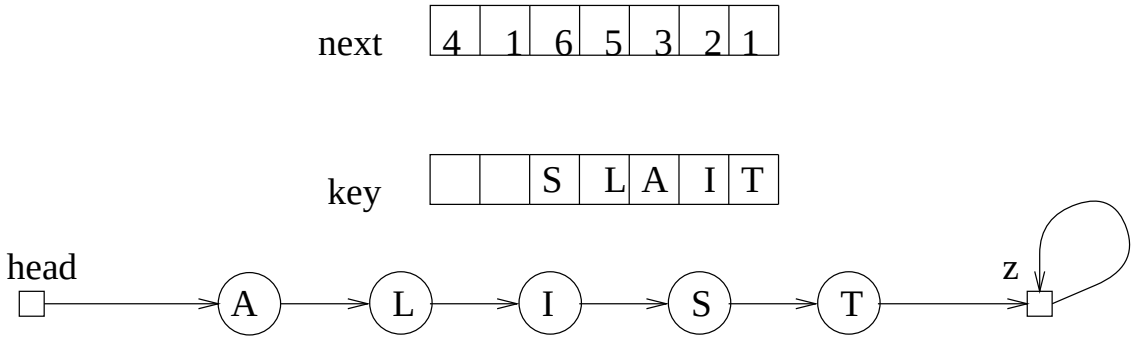


- This can simplify searching, and makes the deletion operation (potentially) easier.
- There is obviously added storage cost, and the number of instructions needed for the various operations approximately doubles.
- **Circular-linked list**:
  - The last node points to the first node.
  - It can be single or doubly linked list.
  - It can be implemented with a fixed or moving “head.”



Linked Lists without Pointers

- Instead of using pointer to implement linked lists, we can use arrays.
- We won't look at this in depth, but it is not too hard to imagine how we could do it.
- There are a few complications in this type of implementation, but they can easily be worked around.
- **Example:**



Linked Lists Summary

- Linked lists are data structures that are in many ways similar to arrays.
- Inserting, deleting or accessing items in linked lists are operations which can be performed.
- Insertion and deletion can be done in constant time.
- Finding an element in a linked list generally takes linear time. This is true whether we are trying to find an element whose value is x, or we are trying to find the ith element on the list.
- It is this last fact that can limit the usefulness of linked lists.