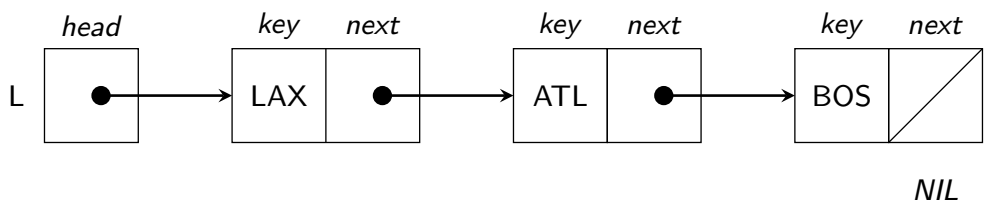
Linked-List

Jianguo Lu



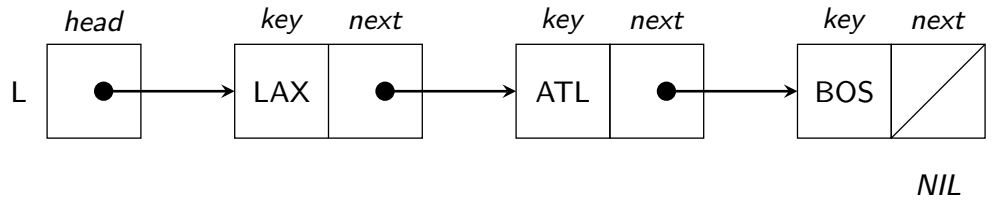
Overview

Drawbacks of Arrays

- Pre-allocate all needed memory up-front
  - waste memory space for cells not used
- Fixed-size–We may not know the size before hand
- One block allocation–empty memory space may be scattered/fragmental
- Not efficient for insert and remove operations–need to shift cells

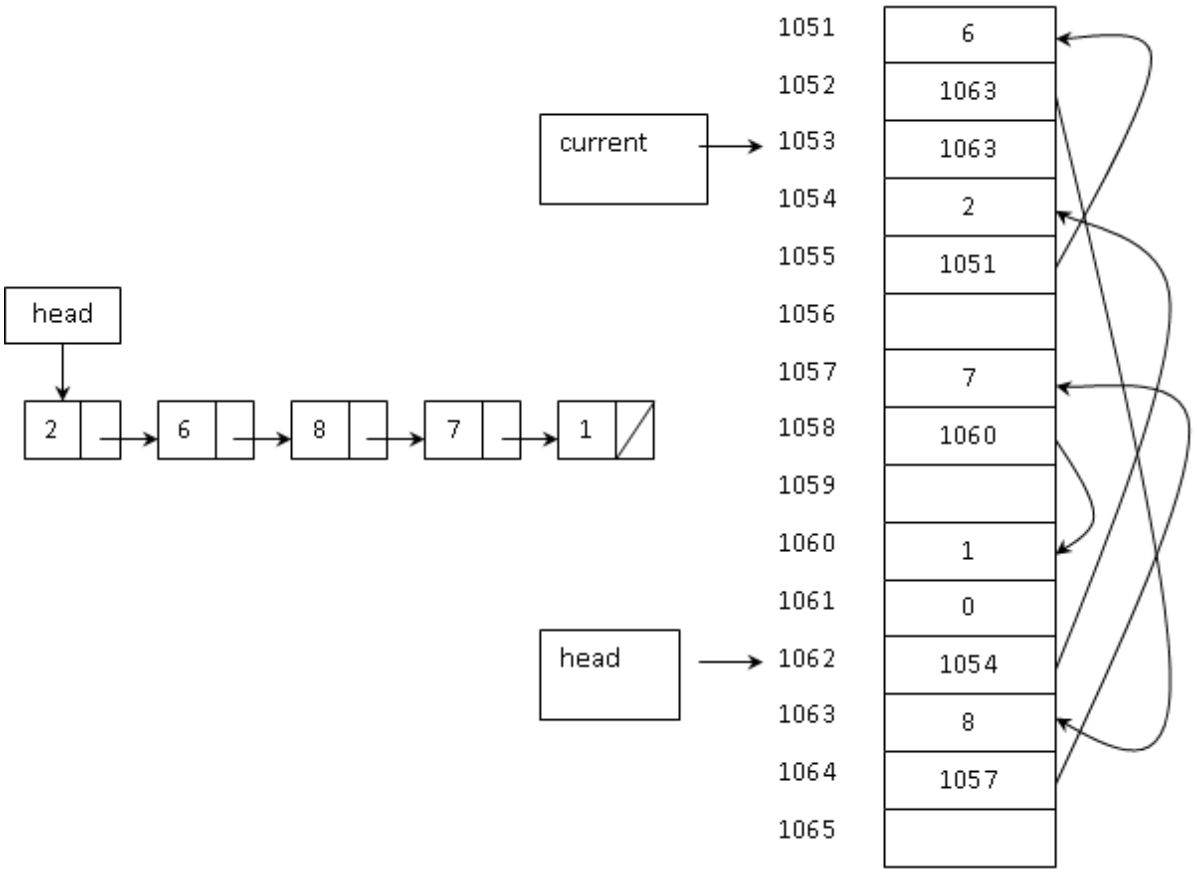
Singly Linked List

Example: Represent list (LAX, ALT, BOS) :



- Each linked list contains a head node which points to the first link in the list
- Singly linked list: contain two slots in each link:
  - *key* slot: contains the content, and
  - *next* slot: points to the next link in the sequence.
- If it is the last node, then *next* will be set to NIL.

Memory allocation of LinkedList

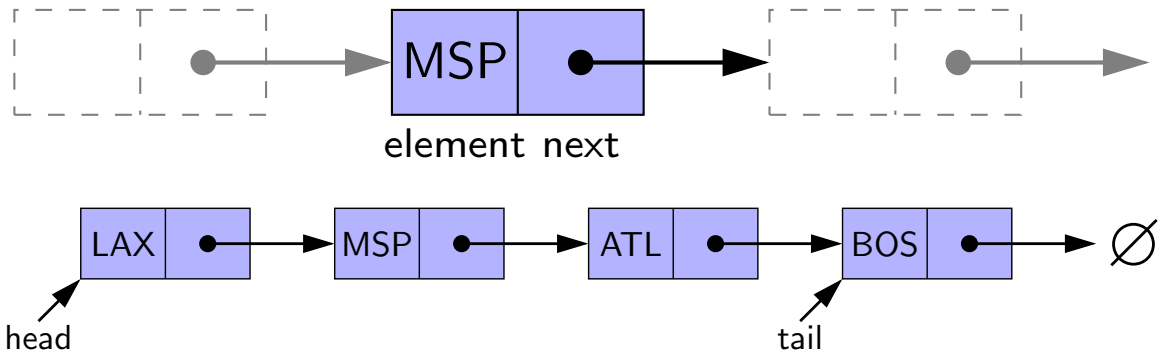


ADT for singly LinkedList

ADT for Singly LinkedList

size( )	:	Returns the number of elements in the list.
isEmpty( )	:	Returns true if the list is empty, and false otherwise.
first()	:	Returns (but does not remove) the first element in the list.
last( )	:	Returns (but does not remove) the last element in the list.
addFirst(e)	:	Adds a new element to the front of the list.
addLast(e)	:	Adds a new element to the end of the list.
removeFirst( )	:	Removes and returns the first element of the list.

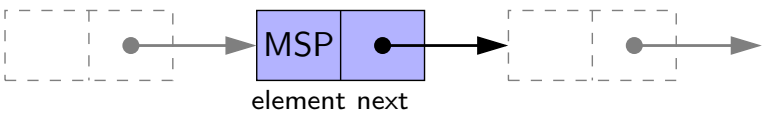
Linked List: Node and List



Java code for Node

```
public class     Node
    private   Object element;
    private   Node next;

    public   Node(Object e,   Node n)   {
         element   =   e;
         next   =   n;
   }
```

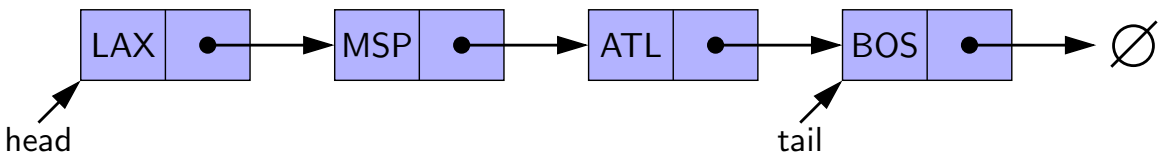


There are always getter and setter methods

```
public   Object getElement()   {return   element;}
public   Node getNext()   {return   next;}
public void setElement(Object newElem)   {element=newElem;   }
public void setNext(Node newNext){next=newNext;}
```

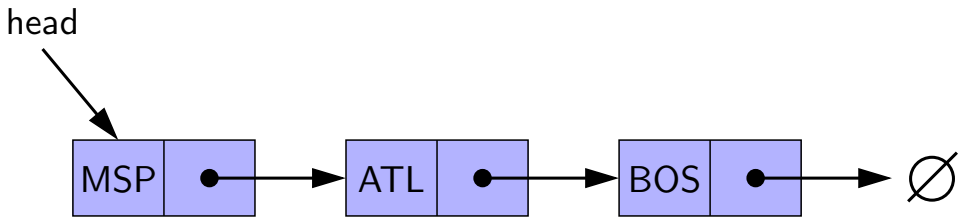
Java code for SinglyLinkedList

```
public class SinglyLinkedList<E> implements Cloneable
  private Node<E> head = null;
  private Node<E> tail = null;
  private int size = 0;
  ...
```

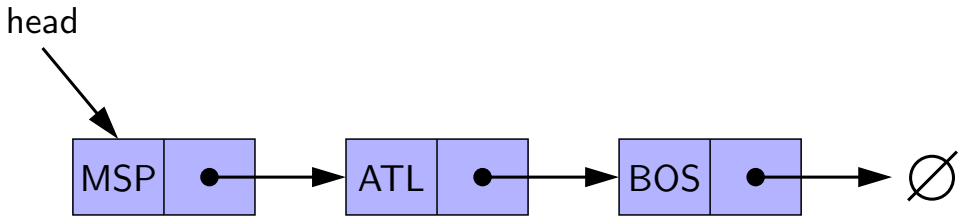


- What else we need to have?
- We need to have methods to operate on the linked list
  - insert at the beginning
  - insert at the tail
  - remove first

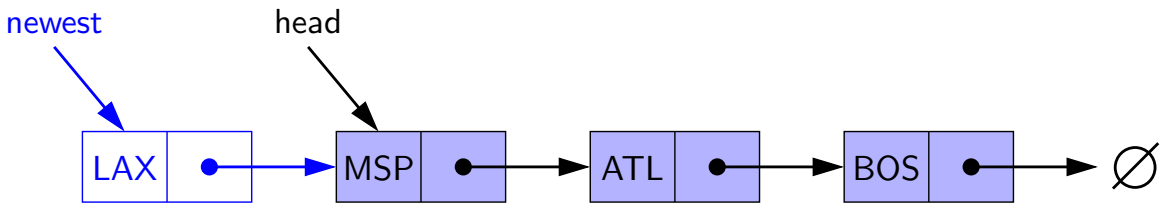
Add at the front: example process



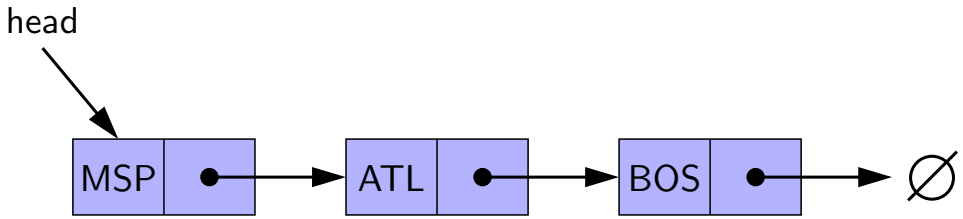
Add at the front: example process



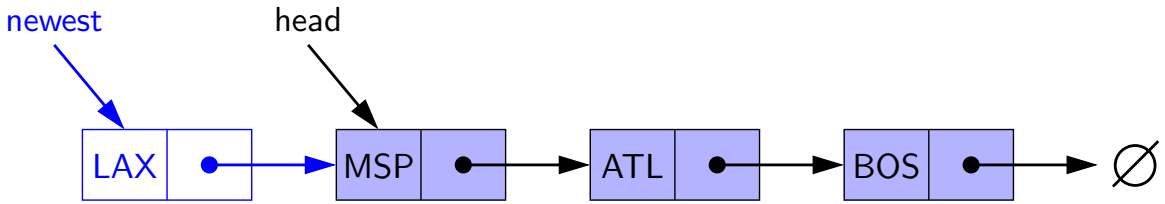
- Allocate a new node
- Insert new element
- Have new node point to old head



Add at the front: example process

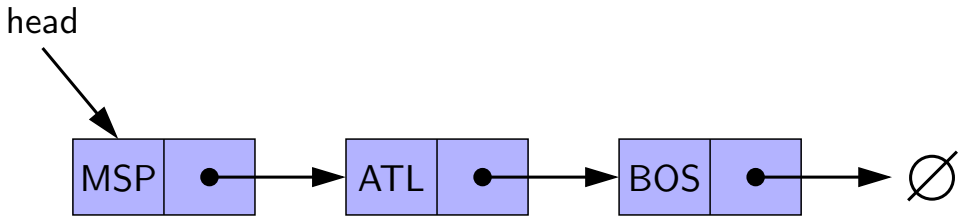


- Allocate a new node
- Insert new element
- Have new node point to old head

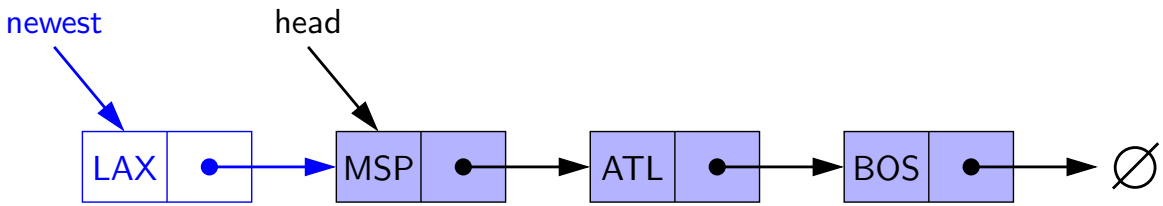


- Update head to point to new node

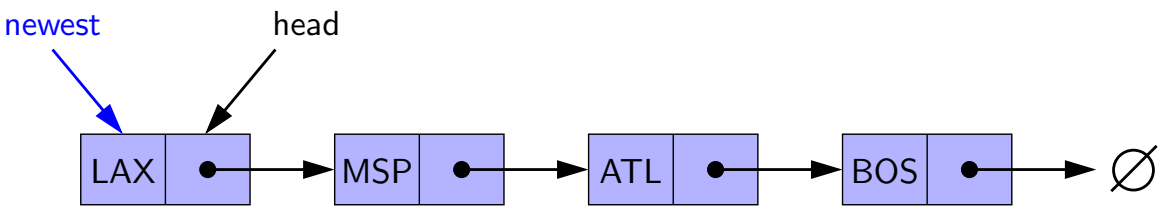
Add at the front: example process



- Allocate a new node
- Insert new element
- Have new node point to old head

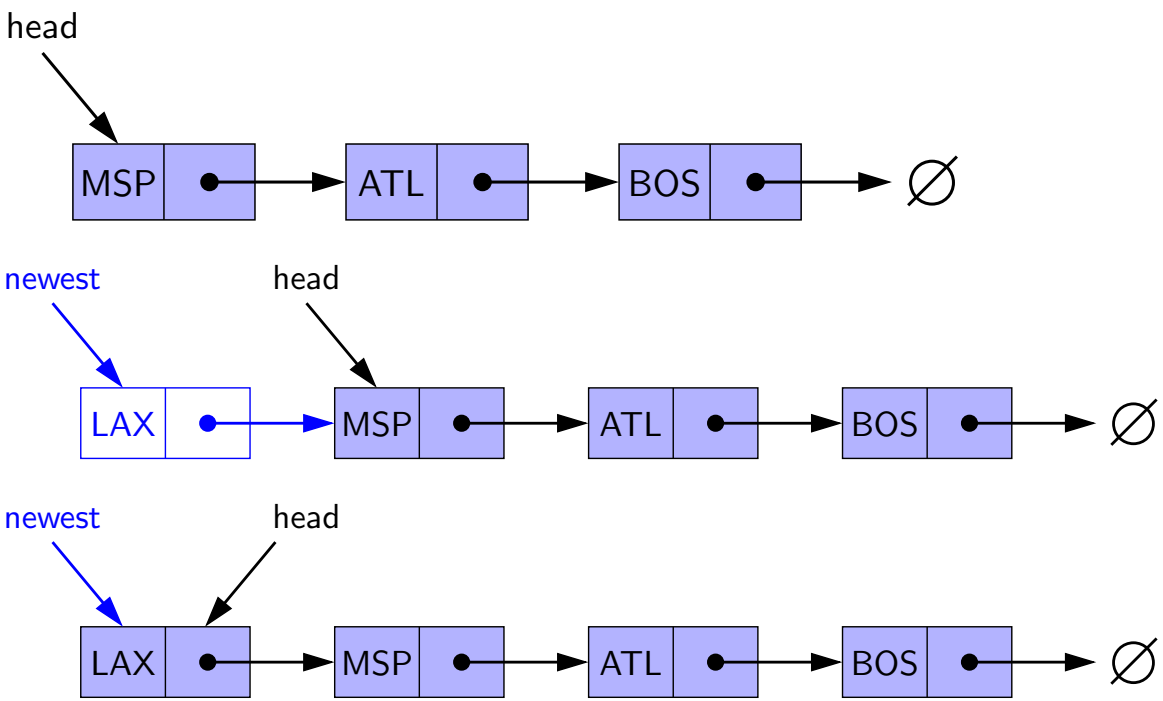


- Update head to point to new node



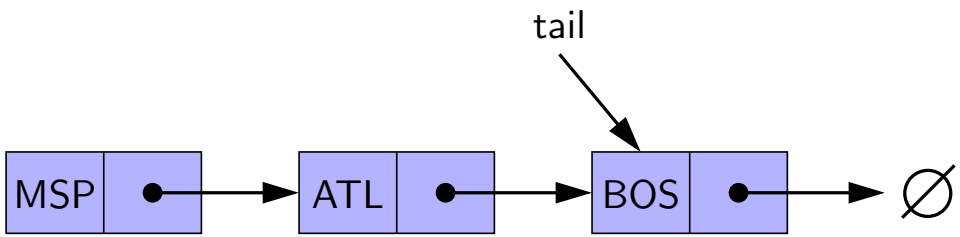
Add at the front: the code

```
public void addFirst(E e) {
  head = new Node<>(e, head);
  size++;
}
```



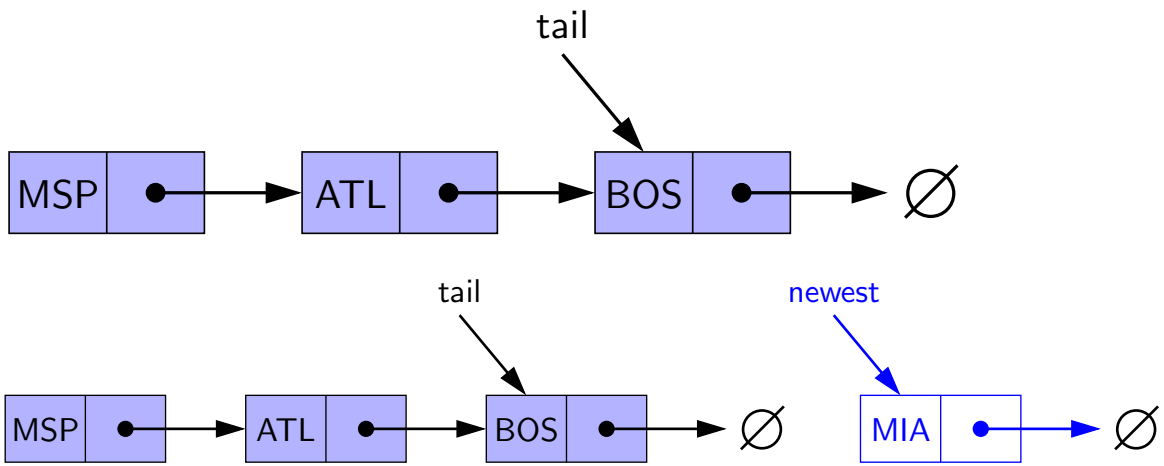
Insert at the tail

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



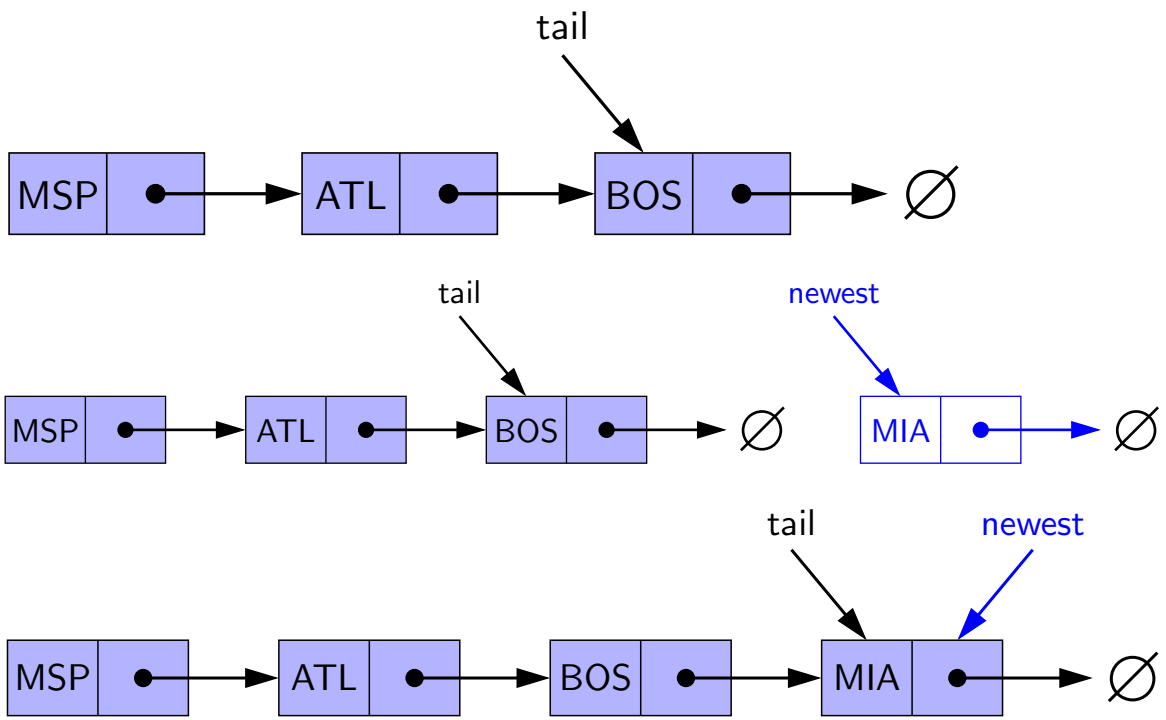
Insert at the tail

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



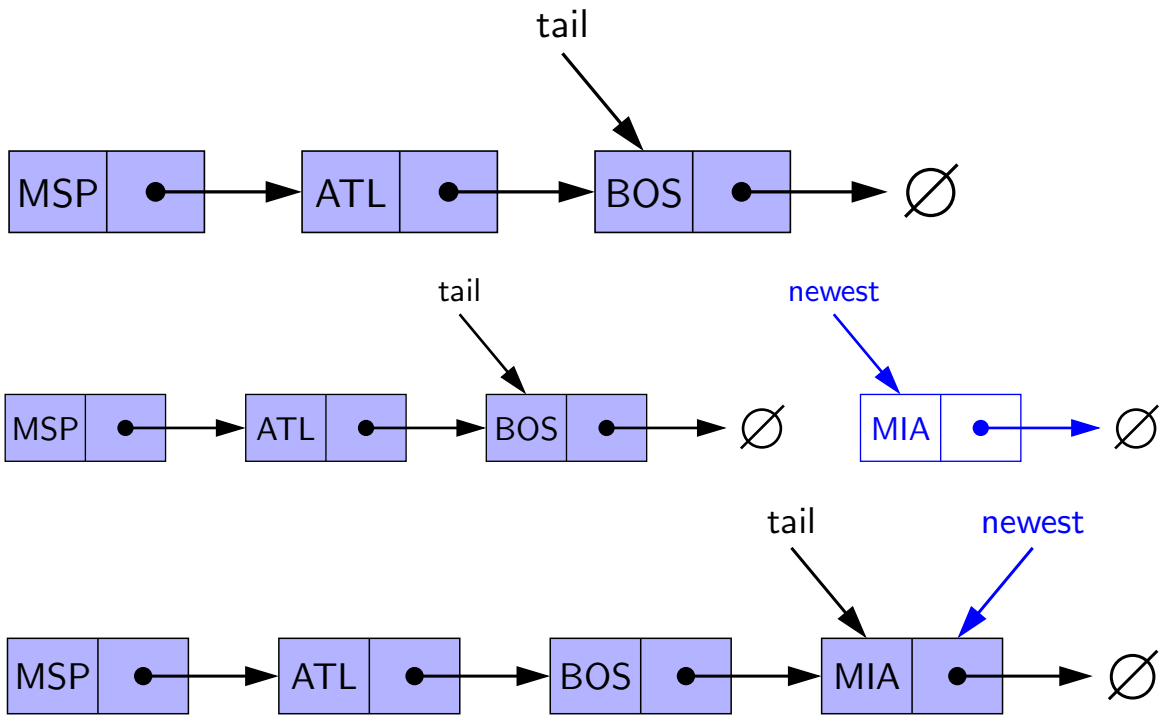
Insert at the tail

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



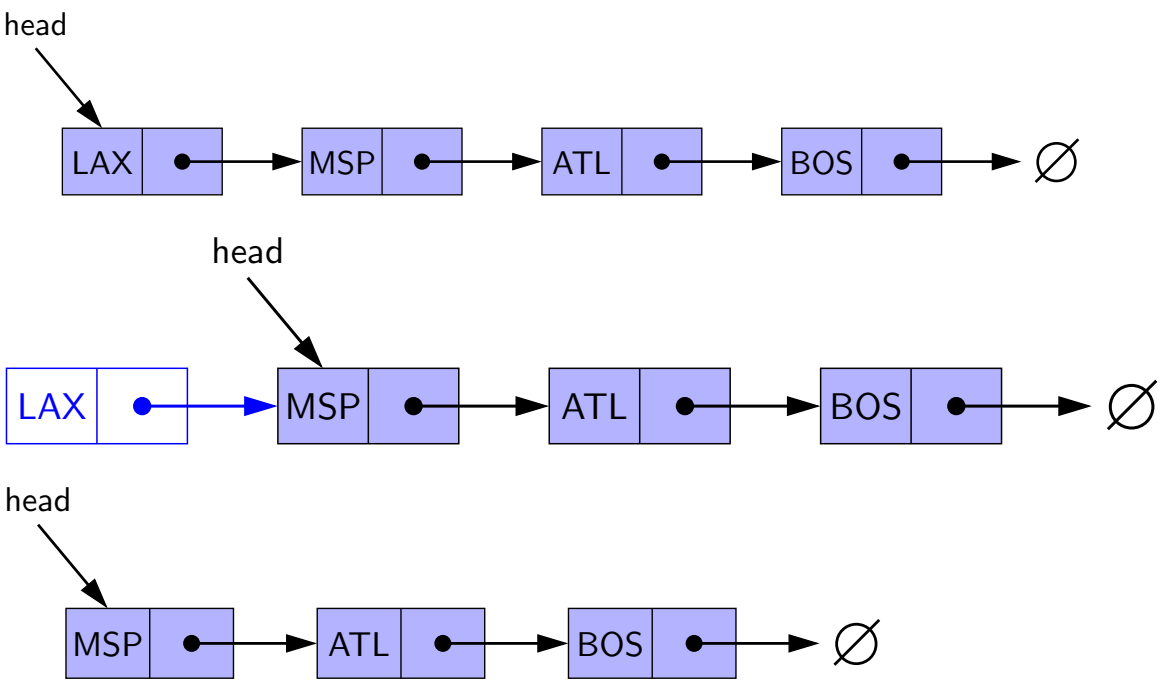
Insert at the tail with Java code

```
public void addLast(E e)
    Node<E> newest = new Node<>(e, null);
   if (isEmpty()) head = newest;
   else tail.setNext(newest);
   tail = newest;
   size++;
```

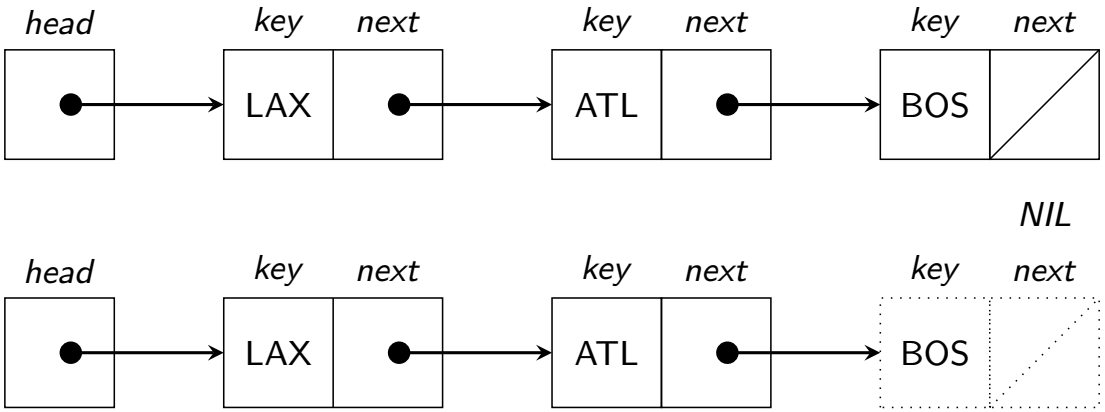


Remove first element

```
public E removeFirst()
    if (isEmpty()) return null;
    E answer = head.getElement();
    head = head.getNext();
    size--;
    if (size == 0)      tail = null;
    return answer;
```



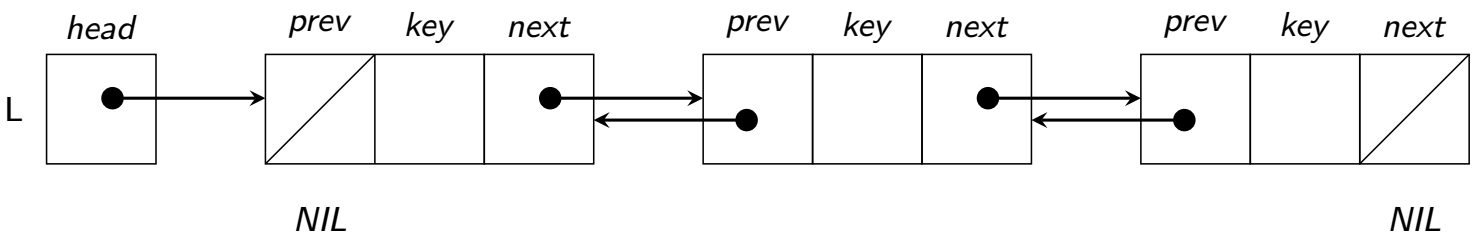
How to remove the last element **efficiently**?



- We can access the tail, but can not access the node before it.
- How to access node ATL?
- We need to add additional links

Doubly linked lists

- Contain three slots in each node.
- One additional *prev* slot, refers to the previous link in the list.
- Two types of linked lists
  - linear: the last node's *next* slot is set to NIL as well as the first link's *prev* slot
  - circular: the last *next* slot points to the first node in the sequence, the *prev* slot of the first link will point to the last node's *next* slot.



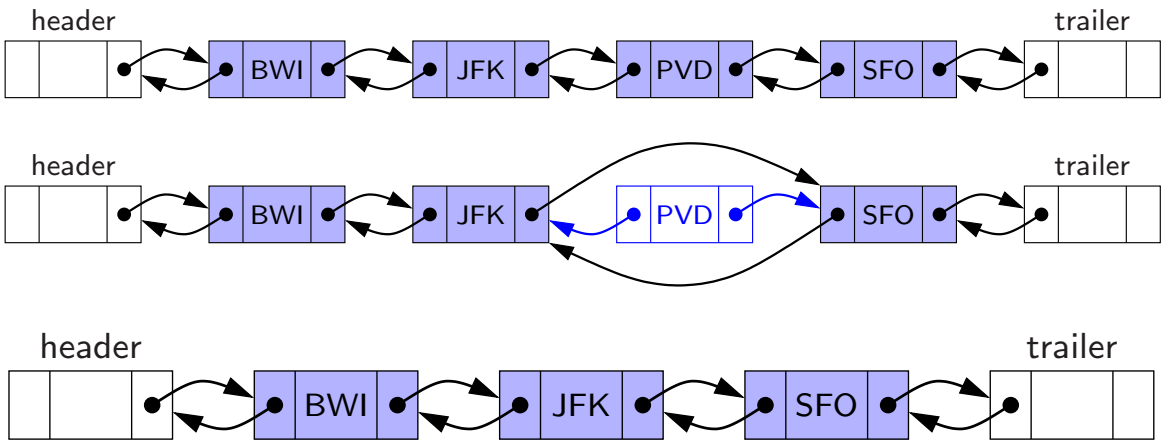
ADT for doubly LinkedList

ADT for doubly LinkedList

size( )	:	Returns the number of elements in the list.
isEmpty( )	:	Returns true if the list is empty, and false otherwise.
first( )	:	Returns (but does not remove) the first element in the list.
last( )	:	Returns (but does not remove) the last element in the list.
addFirst(e)	:	Adds a new element to the front of the list.
removeFirst( )	:	Removes and returns the first element of the list.
addLast(e)	:	Adds a new element to the end of the list.
removeLast( )	:	Removes and returns the last element of the list.

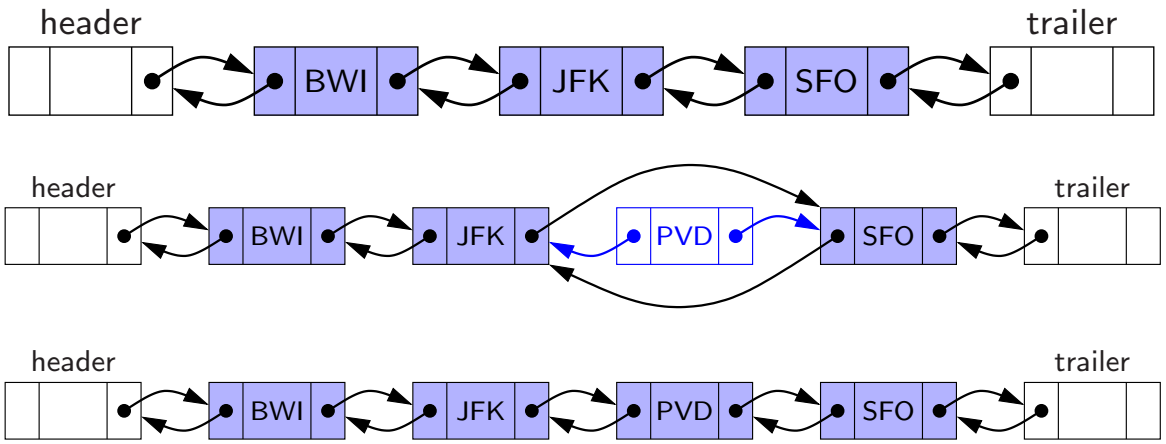
Most methods are the same as in singly LinkedList, except removeLast.

Remove an element




```
private E remove(Node<E> node) {
  Node<E> predecessor = node.getPrev();
  Node<E> successor = node.getNext();
  predecessor.setNext(successor);
  successor.setPrev(predecessor);
  size--;
  return node.getElement();
}
```

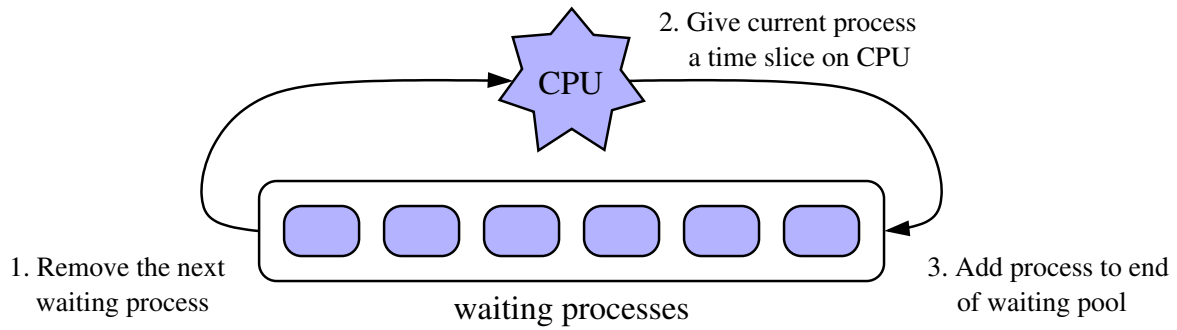
Insert an element between two elements




```
private void addBetween(E e,Node<E> predecessor, Node<E> successor) {
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
}
```

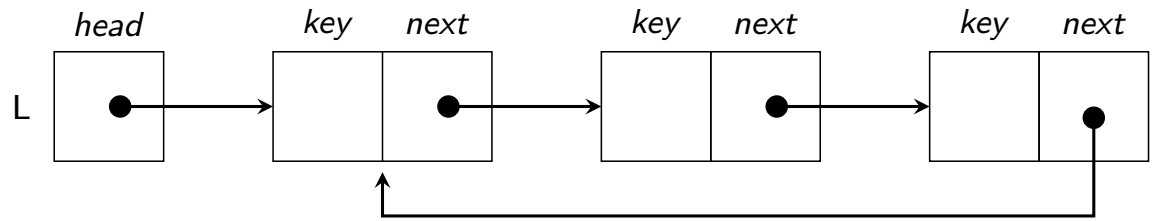
Why circular list

- There are applications where the list is circular
- e.g., processes waiting for CPU

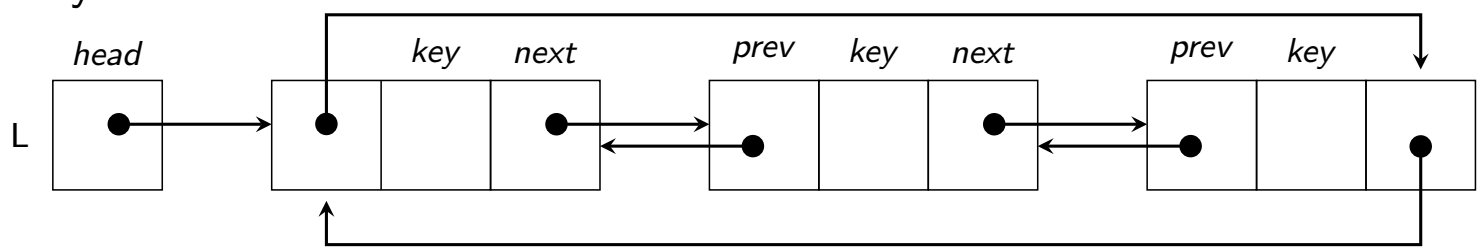


Example of circular list

- tail point to the first element
- Singly-linked circular list:

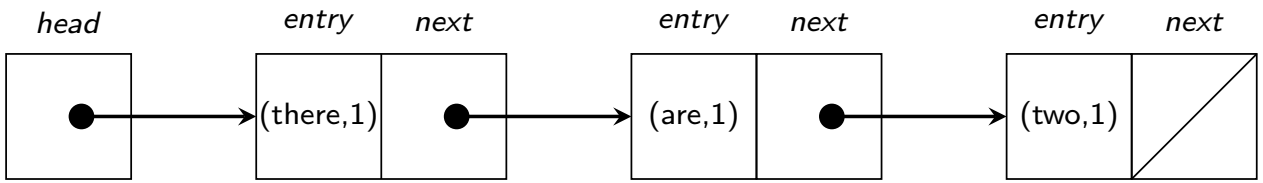


- Doubly-linked circular list:



Word count implemented using Linked List

```
there are two ways of constructing a software design one way
is to make it so simple that there are obviously no deficiencies
and the other way is to make it so complicated that there are no
obvious deficiencies
```



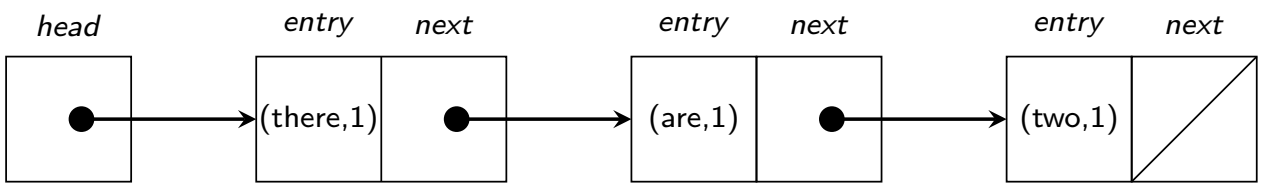
Algorithm 1: Use LinkedList

Input: Array of string tokens
Output: The most frequent word and its frequency

```
1 begin
2     wordFreqList =empty;
3     foreach token in the input do
4         for i =1; i<wordFreqList.length; i++ do
5             if wordFreqList.get(i) equals token then
6                 increment the freq of the word in
                    wordFreqList;
7         if token not in wordFreqList then
8             insert token into the wordFreqList with
                freq=1;
9     Find the most frequent word from wordFreqList;
```

Word count implemented using Linked List

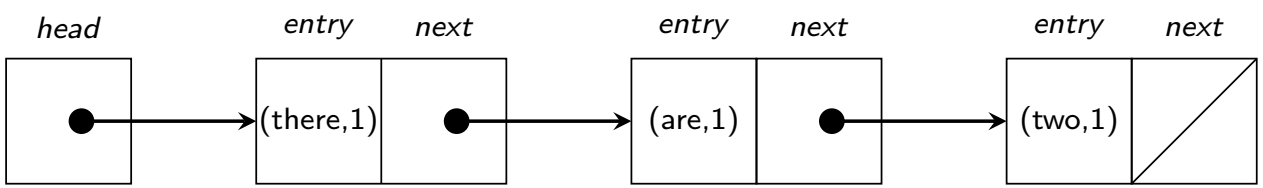
```
LinkedList<Entry<String,Integer>> list=new LinkedList<Entry<String,Integer>>
for (int j = 0; j < tokens.length; j++) {
      String word = tokens[j];
      boolean found = false;
      for (int i = 0; i < list.size(); i++) {
        Entry<String, Integer> e = list.get(i);
        if (word.equals(e.getKey())) {
          e.setValue(e.getValue() + 1);
          list.set(i, e);
          found = true;
          break;
        }
      }
      if (! found)
        list.add(new AbstractMap.SimpleEntry<String, Integer>(word, 1));
    }
```



Note that this is an extremely bad program!

Find the max

```
    int maxCount = 0;
    String maxWord = "";
    for (int i = 0; i < list.size(); i++) {
      int count = list.get(i).getValue();
      if (count > maxCount) {
        maxWord = list.get(i).getKey();
        maxCount = count;
      }
    }
    return new AbstractMap.SimpleEntry<String, Integer>(maxWord, maxCount);
  }
```



Takeaways

- Linked list is more flexible

  Expandable no longer fixed length as in Arrays
  Scattered Memory no longer contiguous block memory allocation
- Not efficient for indexing. Compare the operation that gets the i-th element in Array and in LinkedList
- Has memory overhead
- There are circular LinkedListed and Doubly LinkedList
- Readings: Goodrich et al: p122-137. Lab assignment 1.