Data Structures and Algorithms

Linked Lists

Amilcar Aponte

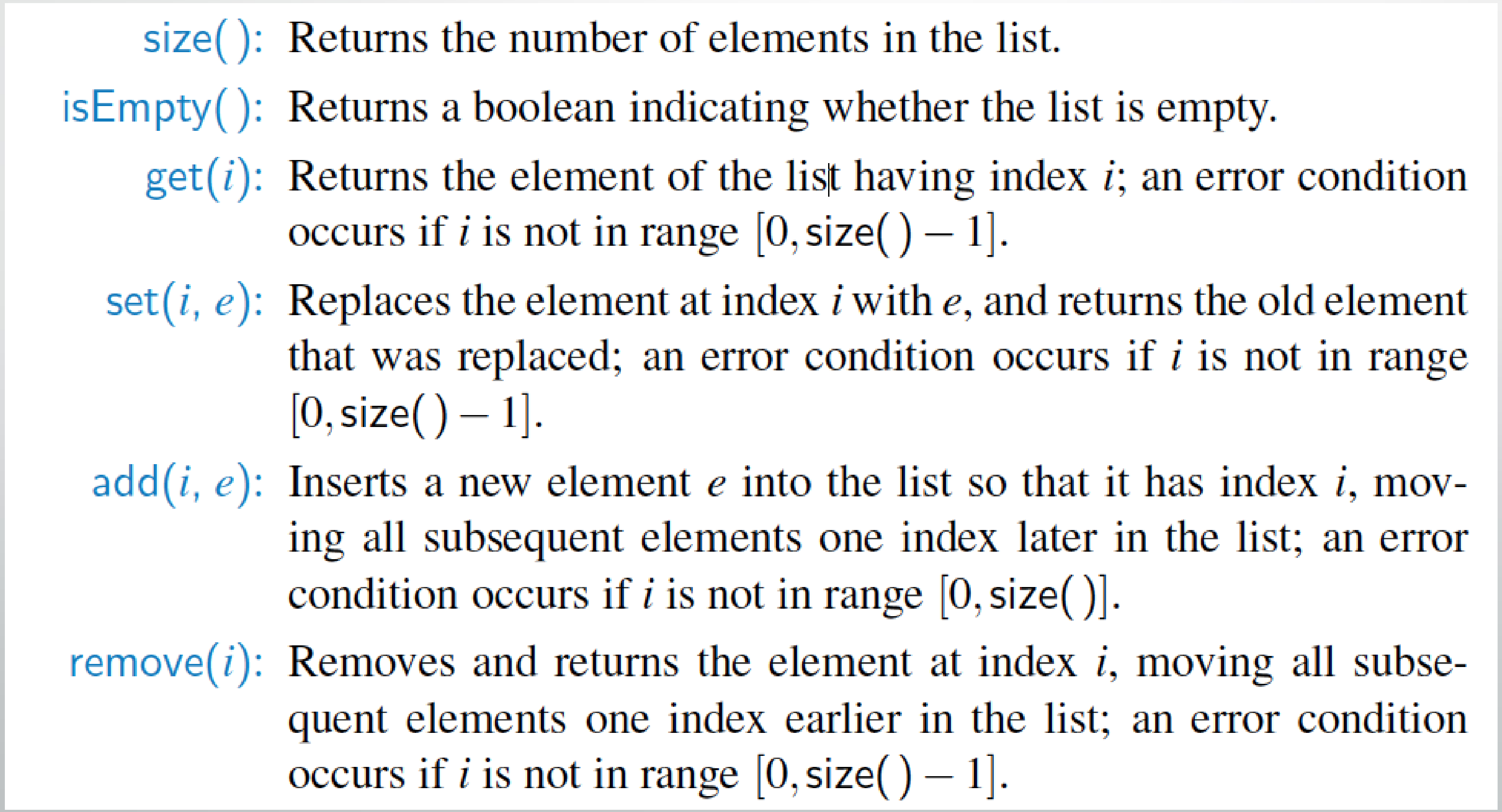
amilcar@cct.ie

Lists

- A way to organize data
- **Examples**
  - To-do list
  - Gift lists
  - Grocery Lists
- Items in list have position – It is an ordered structure
  - May or may not be important
- Items may be added anywhere

The java.util.List ADT

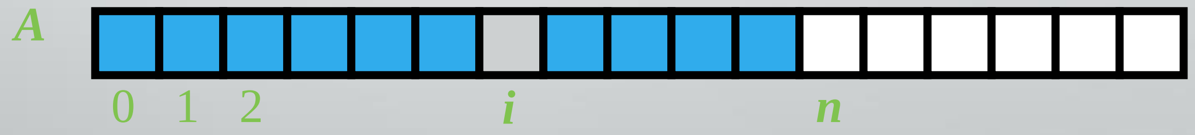
- The java.util.List interface includes the following methods:



size( ):	Returns the number of elements in the list.
isEmpty( ):	Returns a boolean indicating whether the list is empty.
get(i):	Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.
set(i, e):	Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.
add(i, e):	Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}()]$.
remove(i):	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

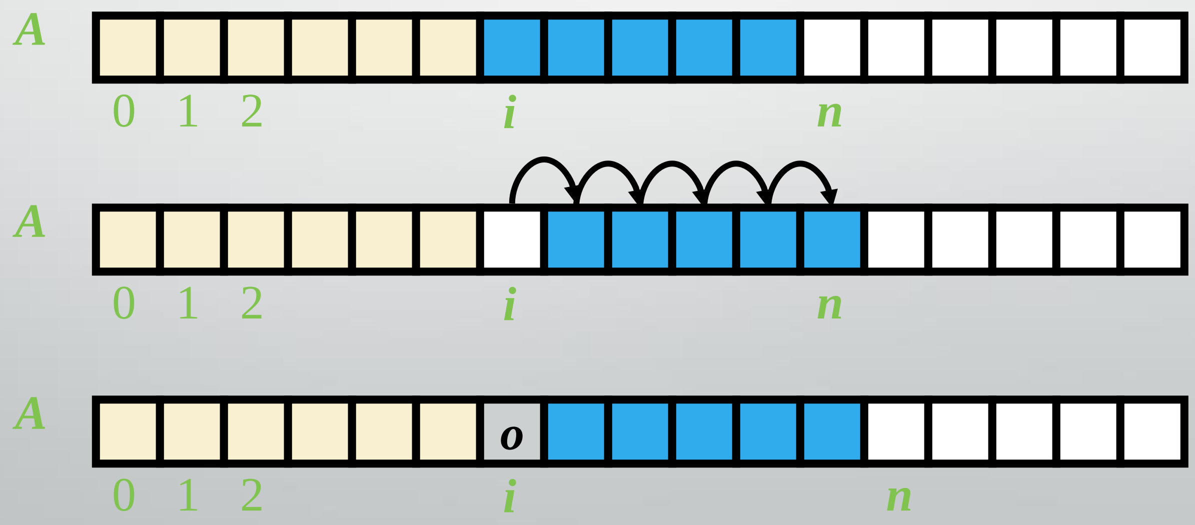
Array Lists

- An obvious choice for implementing the list ADT is to use an array, **A**, where **A[i]** stores (a reference to) the element with index **i**.
- With a representation based on an array **A**, the get(**i**) and set(**i**, **e**) methods are easy to implement by accessing **A[i]** (assuming **i** is a legitimate index).



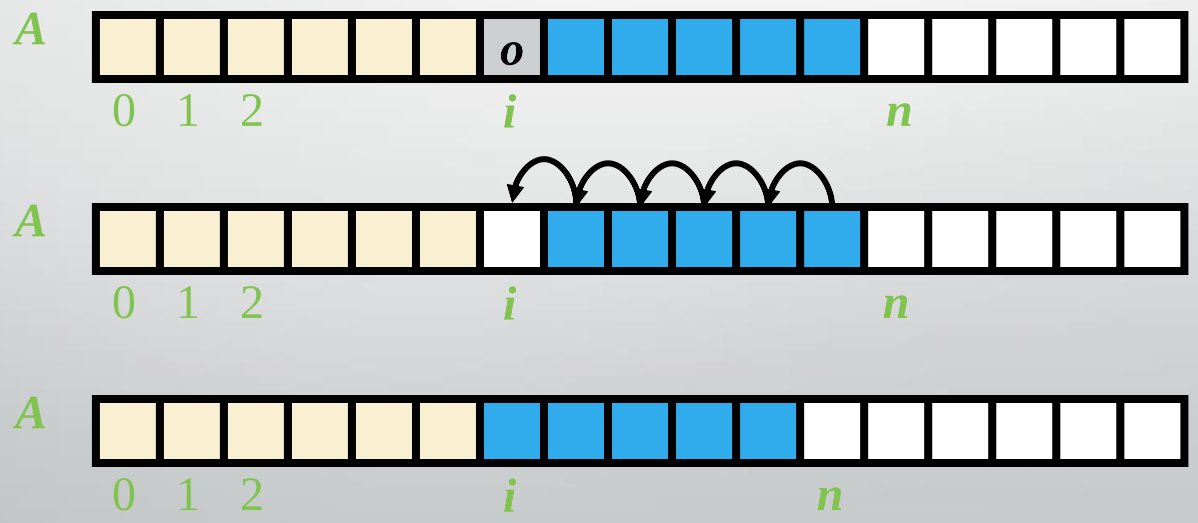
Insertion

- In an operation $\boldsymbol{add}(\boldsymbol{i}, \boldsymbol{o})$, we need to make room for the new element by shifting forward the $\boldsymbol{n} - \boldsymbol{i}$ elements $\boldsymbol{A}[\boldsymbol{i}], \ldots, \boldsymbol{A}[\boldsymbol{n} - 1]$
- In the worst case ($\boldsymbol{i} = 0$)



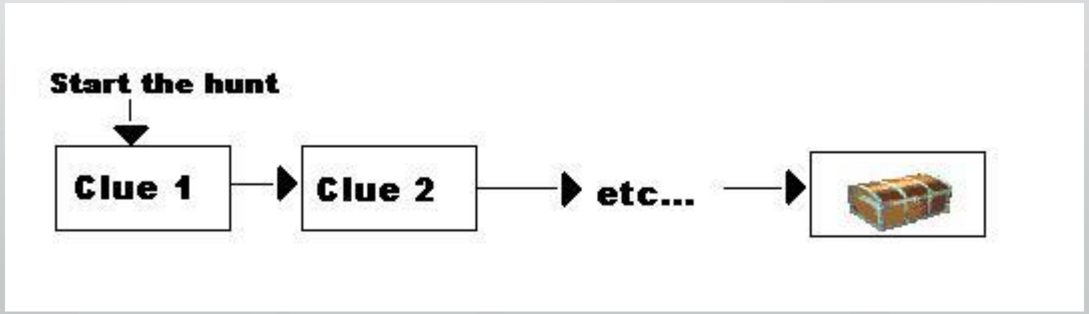
Element Removal

- In an operation ***remove***(i), we need to fill the hole left by the removed element by shifting backward the $\boldsymbol{n - i - 1}$ elements $\boldsymbol{A[i+1], \ldots, A[n-1]}$
- In the worst case ($\boldsymbol{i = 0}$)



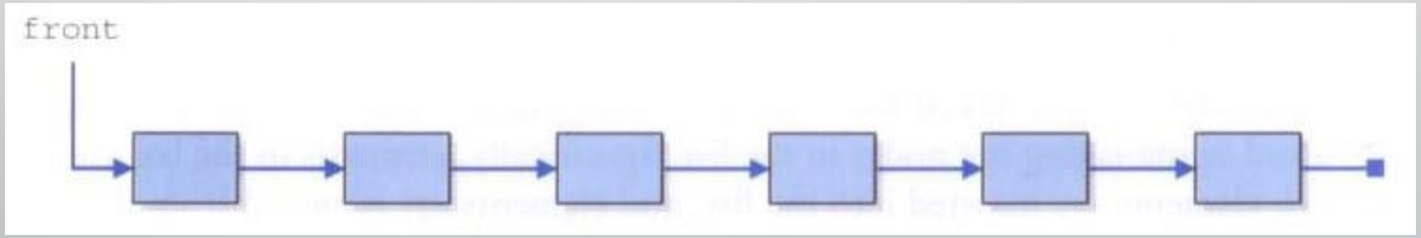
Linked Structures

- An alternative to array-based implementations are *linked structures*
- A linked structure uses object references to create links between objects
- Recall that an object reference variable holds the address of an object



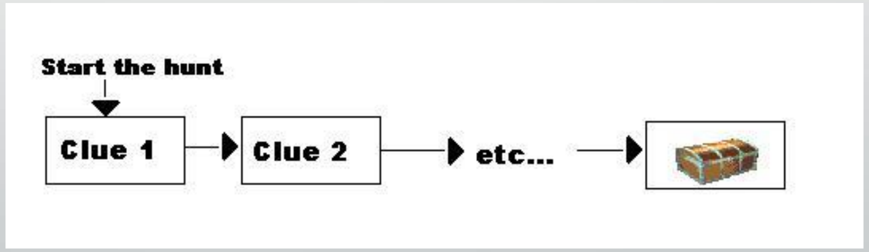
Linked Structures

- A `Person` object, for instance, could contain a reference to another `Person` object
- A series of `Person` objects would make up a *linked list*:



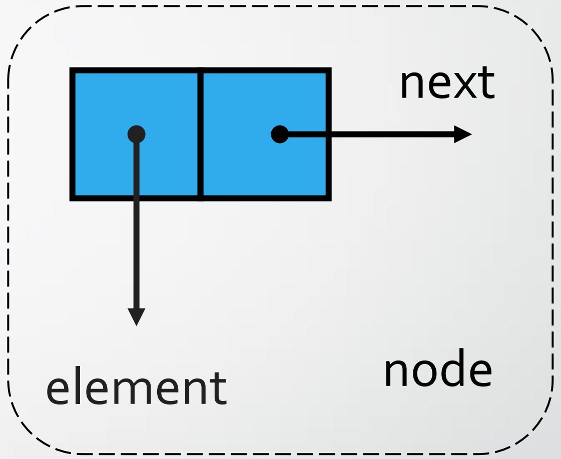
Linked Lists

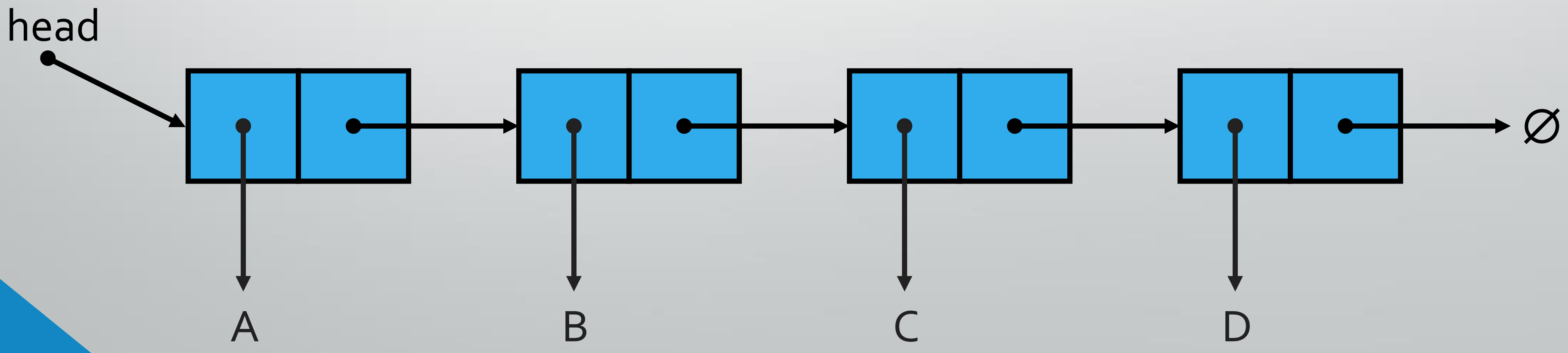
- There are **no index** values built into linked lists
- To access each node in the list you must follow the references from one node to the next



Singly Linked List

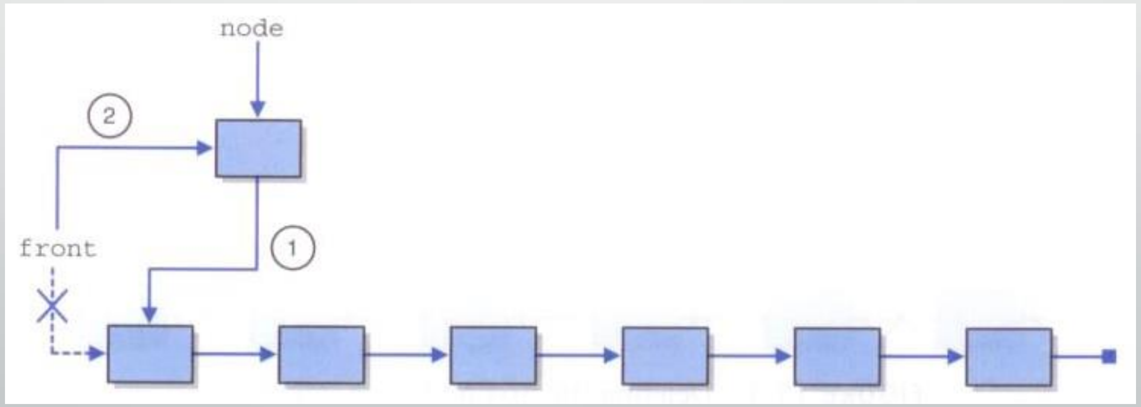
- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - Link (reference or address) to the next node





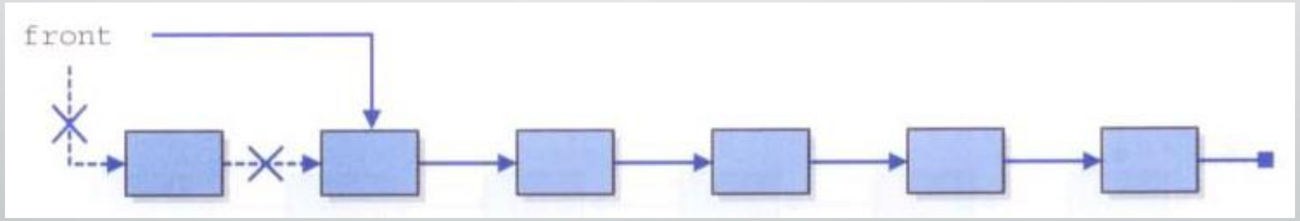
Singly Linked Lists

- Care must be taken to maintain the integrity of the links
- To insert a node at the front of the list, first point the new node to the front node, then reassign the `front` reference



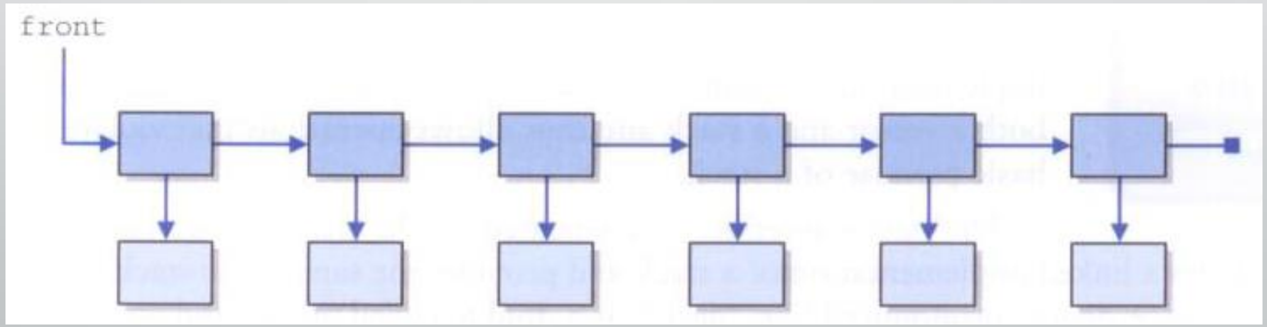
Singly Linked Lists

- To delete the first node, reassign the `front` reference accordingly
- If the deleted node is needed elsewhere, a reference to it must be established before reassigning the `front` pointer



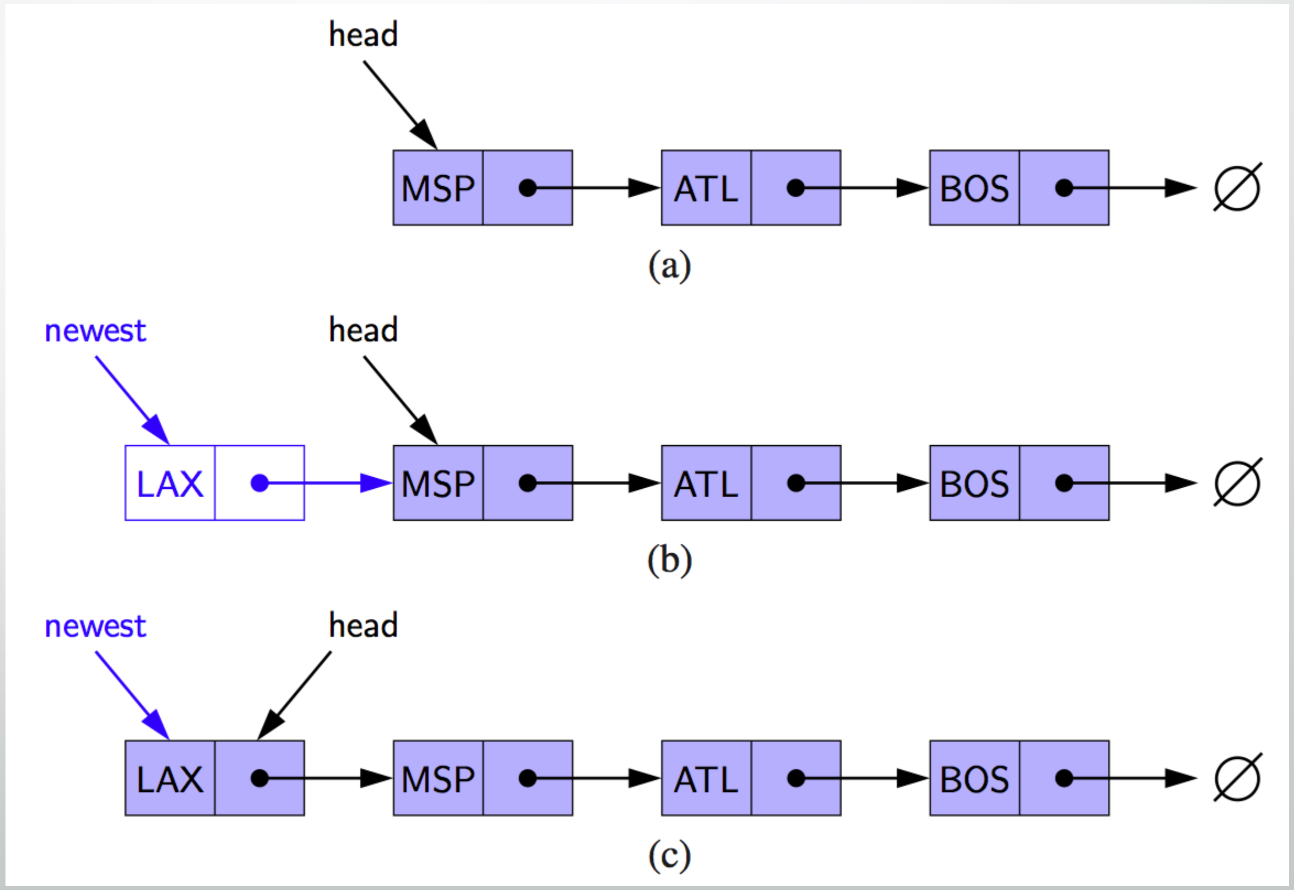
Singly Linked Lists

- So far we've assumed that the list contains nodes that are *self-referential* (`Person` points to a `Person`)
- But often we'll want to make lists of objects that don't contain such references
- **Solution:** have a separate `Node` class that forms the list and holds a reference to the objects being stored



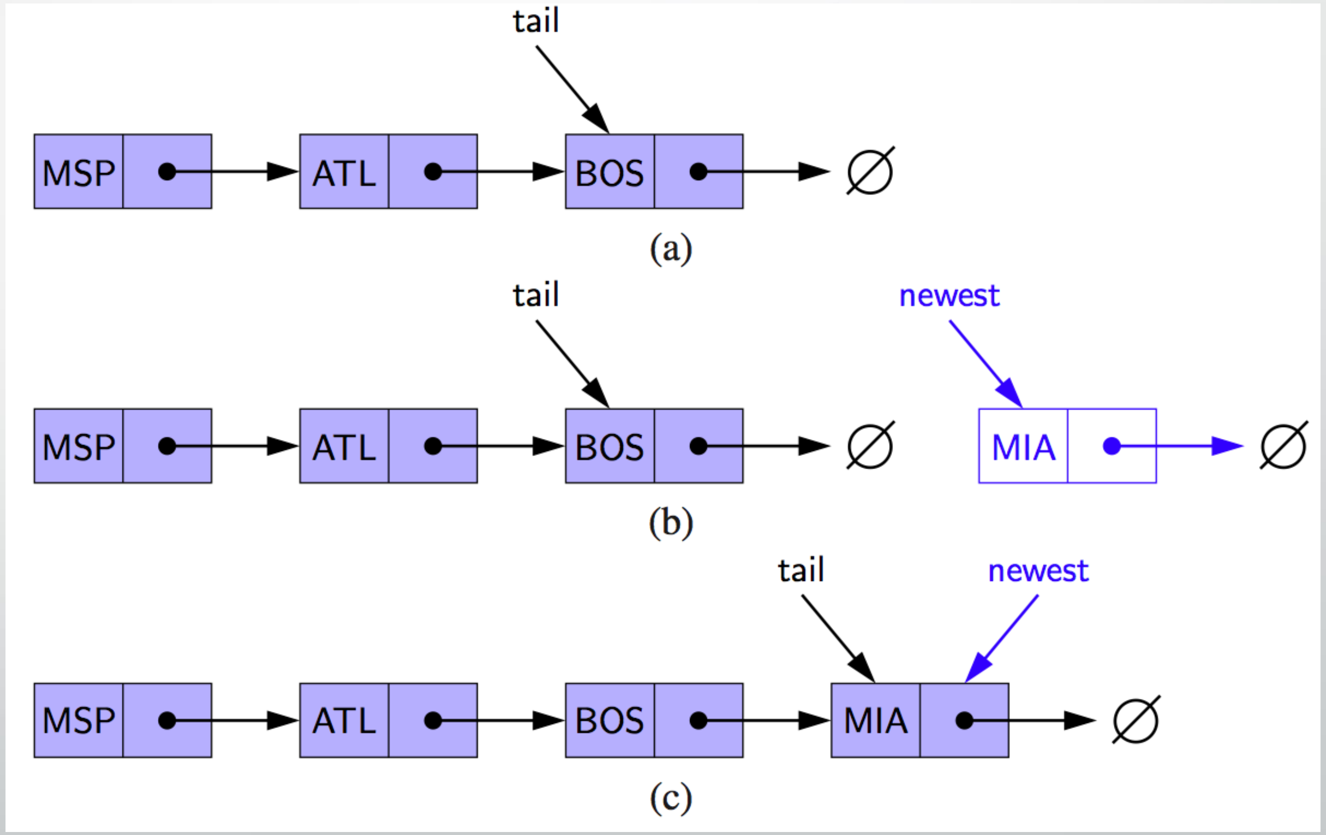
Inserting at the Head

- Allocate new node
- Have new node point to old head
- Update head to point to new node



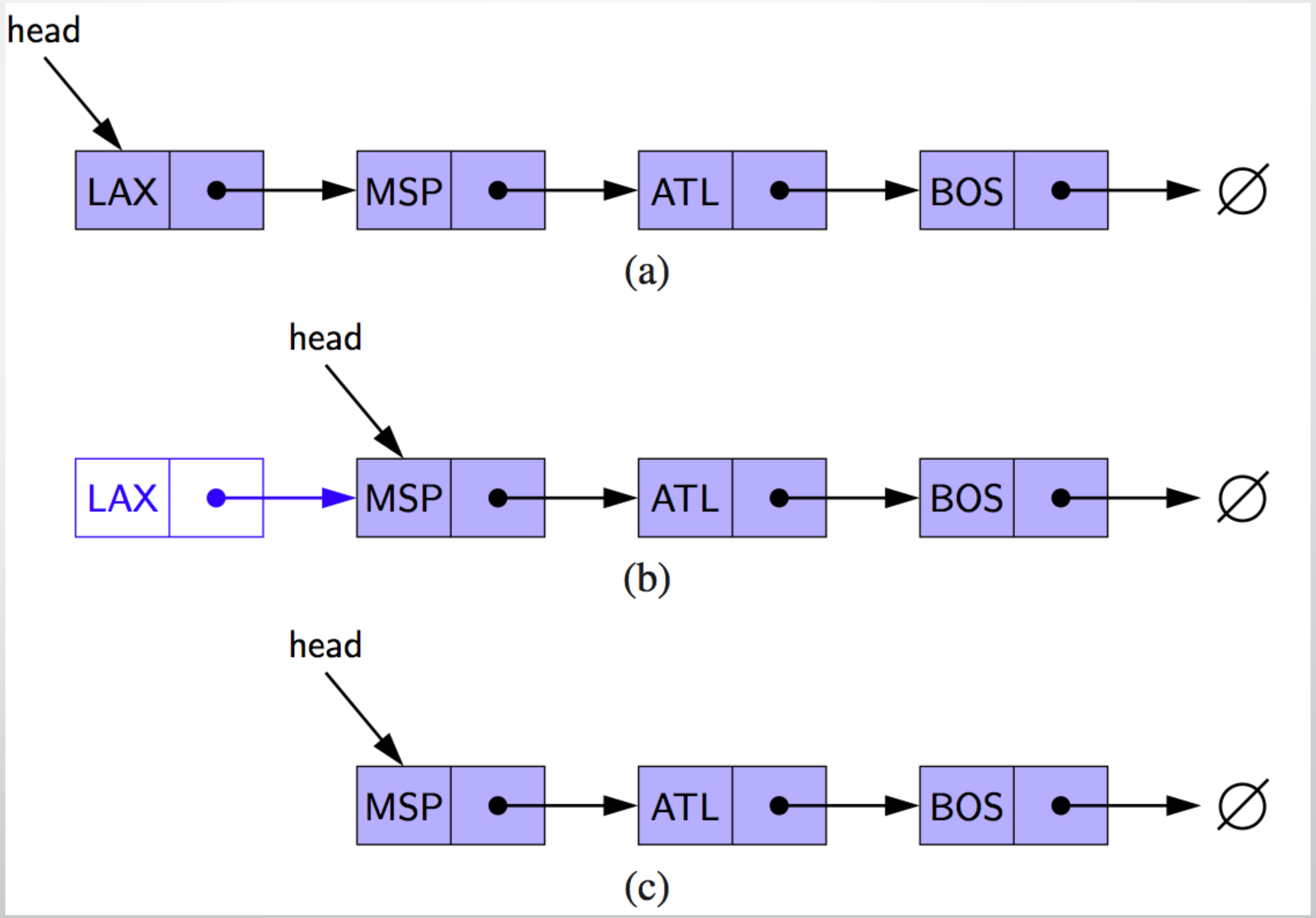
Inserting at the Tail

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



Removing at the Head

- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node



Let’s try to implement this

- What classes do we need?
- What sort of Data are we going to store?
- Who is the client of my linked list?

That’s all folks

- Any question?