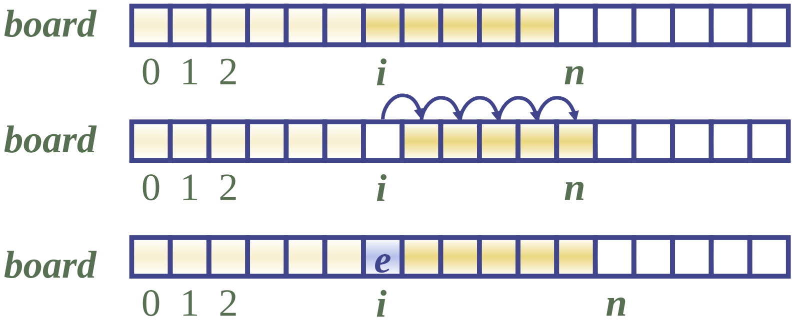
CS121 Data Structures A, C
Linked Lists

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

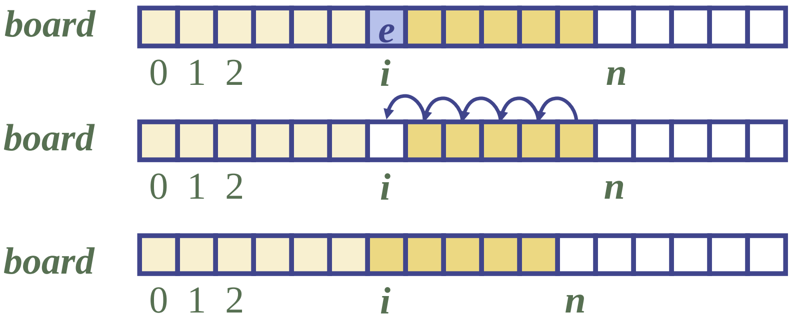
Adding an Array Entry

To add an entry *e* into array *board* at index i, we need to make room for it by shifting forward the $n - i$ entries $board[i], \ldots, board[n-1]$



Removing an Array Entry

To remove the entry e at index i, we need to fill the hole left by e by shifting backward the $n - i - 1$ elements $board[i+1], \ldots, board[n-1]$



Singly Linked Lists

Drawbacks of *array* as an ordered data structure:

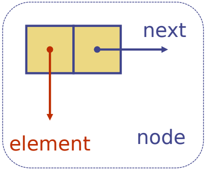
- fixed capacity
- expensive insertions and deletions at interior positions (shifting many elements)

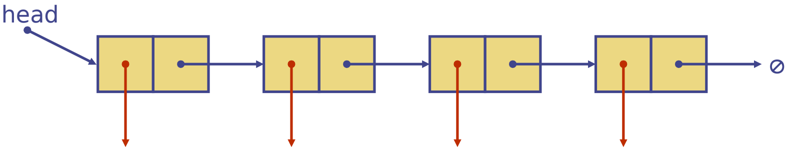
Linked list provides an alternative to an array-based structure.

A linked list is a collection of **nodes** that collectively form a linear sequence.

In a **singly linked list**, each node stores:

- a reference to an object that is an element of the sequence
- a reference to the next node of the list





Linked List Terms

The linked list instance must keep a reference to the first node of the list, known as the **head**.

The last node of the list is known as the **tail**.

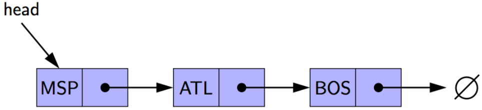
Traversing the linked list—starting at the head and moving from one node to another by following each node's `next` reference.

The tail has **null** as its `next` reference.

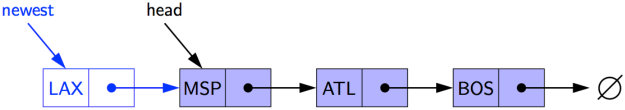
Commonly, a reference to the tail node is also stored, as is the count of the total number of nodes in the list (its **size**).

Inserting at the Head

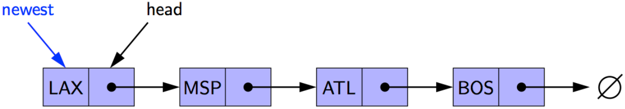
Algorithm `addFirst`(e):
- newest = Node(e) {create new node instance storing reference to element e}
- newest.next = head {set new node's next to reference the old head node}
- head = newest {set variable head to reference the new node}
- size = size + 1 {increment the node count}



(a)



(b)

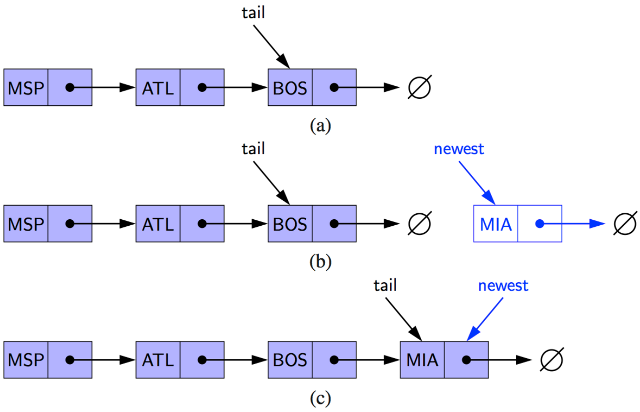


(c)

Inserting at the Tail

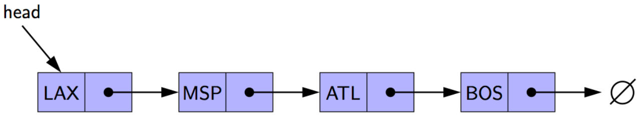
Algorithm `addLast`(e):

newest $=$ Node(e)	{create new node instance storing reference to element e}
newest.next $=$ null	{set new node's next to reference the null object}
tail.next $=$ newest	{make old tail node point to new node}
tail $=$ newest	{set variable tail to reference the new node}
size $=$ size $+$ 1	{increment the node count}

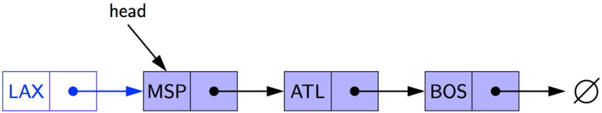


Removing from the Head

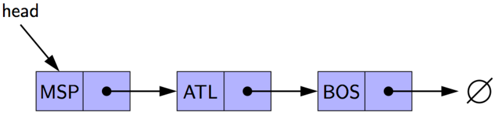
Algorithm `removeFirst`:
  if head == null **then**
    the list is empty.
  head = head.next {make head point to next node (or null)}
  size = size − 1 {decrement the node count}



(a)



(b)

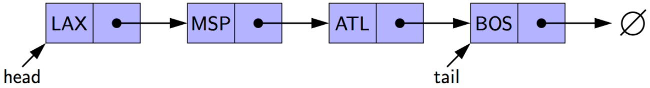


(c)

Removing from the Tail

Removing from the tail of a singly linked list is not efficient!

There is no constant-time way to update the tail to point to the previous node



Interface of a Singly Linked List

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.

Singly Linked List Implementation: Node

```
public class SinglyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;            // reference to the element stored at this node
    private Node<E> next;         // reference to the subsequent node in the list
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    public E getElement() { return element; }
    public Node<E> getNext() { return next; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------
```

Singly Linked List Implementation I

```
public class SinglyLinkedList<E> {
  (nested Node class goes here)
  // instance variables of the SinglyLinkedList
  private Node<E> head = null;              // head node of the list (or null if empty)
  private Node<E> tail = null;              // last node of the list (or null if empty)
  private int size = 0;                     // number of nodes in the list
  public SinglyLinkedList( ) { }            // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {                 // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return head.getElement( );
  }
  public E last( ) {                  // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
```

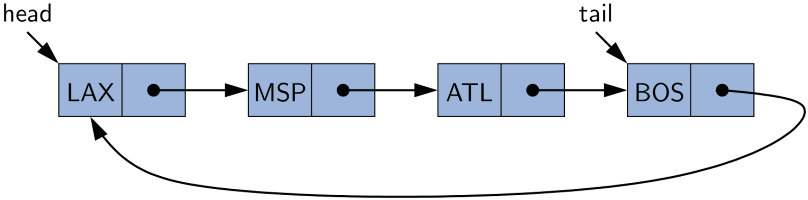
Singly Linked List Implementation II

```
  // update methods
  public void addFirst(E e) {                 // adds element e to the front of the list
    head = new Node<>(e, head);               // create and link a new node
    if (size == 0)
      tail = head;                            // special case: new node becomes tail also
    size++;
  }
  public void addLast(E e) {                  // adds element e to the end of the list
    Node<E> newest = new Node<>(e, null);     // node will eventually be the tail
    if (isEmpty())
      head = newest;                          // special case: previously empty list
    else
      tail.setNext(newest);                   // new node after existing tail
    tail = newest;                            // new node becomes the tail
    size++;
  }
  public E removeFirst() {                    // removes and returns the first element
    if (isEmpty()) return null;               // nothing to remove
    E answer = head.getElement();
    head = head.getNext();                    // will become null if list had only one node
    size--;
    if (size == 0)
      tail = null;                            // special case as list is now empty
    return answer;
  }
}
```

Circularly Linked Lists

There are applications in which data can be viewed as having a **cyclic order**, with well-defined neighbouring relationships, but no fixed beginning or end.

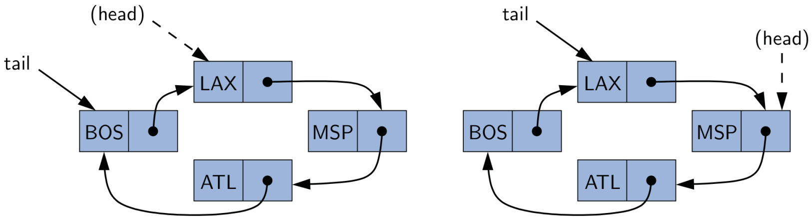
A **circularly linked list** is a singularly linked list in which the `next` reference of the tail node is set to refer back to the head of the list (rather than **null**).



Rotating

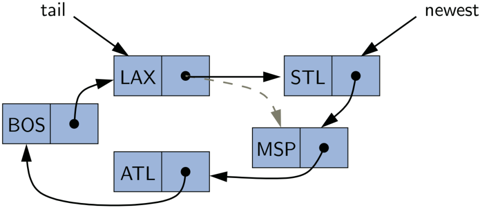
We no longer explicitly maintain the `head` reference. Thus we save a bit on memory usage and make the code simpler and more efficient.

When **rotating** the linked list, we simply advance the `tail` reference to point to the node that follows it.



Adding at the Head/Tail

Add at the head:



Add at the tail: add at the head and immediately rotate

Circularly Linked List Implementation I

```
public class CircularlyLinkedList<E> {
  (nested node class identical to that of the SinglyLinkedList class)
  // instance variables of the CircularlyLinkedList
  private Node<E> tail = null;                  // we store tail (but not head)
  private int size = 0;                         // number of nodes in the list
  public CircularlyLinkedList( ) { }            // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {                  // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return tail.getNext( ).getElement( );       // the head is *after* the tail
  }
  public E last( ) {                   // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
```

Circularly Linked List Implementation II

```
    // update methods
    public void rotate() {              // rotate the first element to the back of the list
      if (tail != null)                 // if empty, do nothing
        tail = tail.getNext();          // the old head becomes the new tail
    }
    public void addFirst(E e) {         // adds element e to the front of the list
      if (size == 0) {
        tail = new Node<>(e, null);
        tail.setNext(tail);             // link to itself circularly
      } else {
        Node<E> newest = new Node<>(e, tail.getNext());
        tail.setNext(newest);
      }
      size++;
    }
    public void addLast(E e) {          // adds element e to the end of the list
      addFirst(e);                      // insert new element at front of list
      tail = tail.getNext();            // now new element becomes the tail
    }
    public E removeFirst() {            // removes and returns the first element
      if (isEmpty()) return null;       // nothing to remove
      Node<E> head = tail.getNext();
      if (head == tail) tail = null;    // must be the only node left
      else tail.setNext(head.getNext()); // removes "head" from the list
      size--;
      return head.getElement();
    }
  }
```

Doubly Linked Lists

Limitations of *singly linked list*:

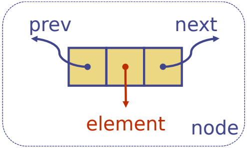
- unable to efficiently delete a node at the tail
- cannot efficiently delete a node from an interior position if given a reference to it (cannot determine the preceding node)

In a **doubly linked list** each node keeps an explicit reference to the node before it and a reference to the node after it.

A doubly linked list can be traversed forward and backward.

In a **doubly linked list**, each node stores:

- a reference to an object that is an element of the sequence
- a reference to the previous node of the list
- a reference to the next node of the list

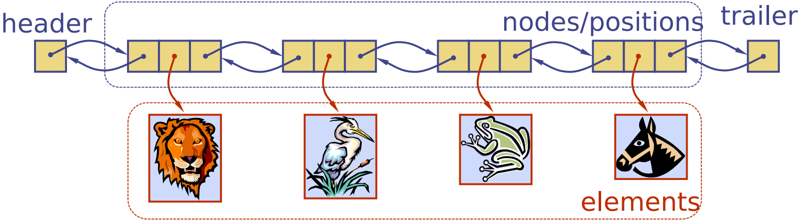


Header and Trailer Sentinels

It helps to add special nodes at both ends of the list:

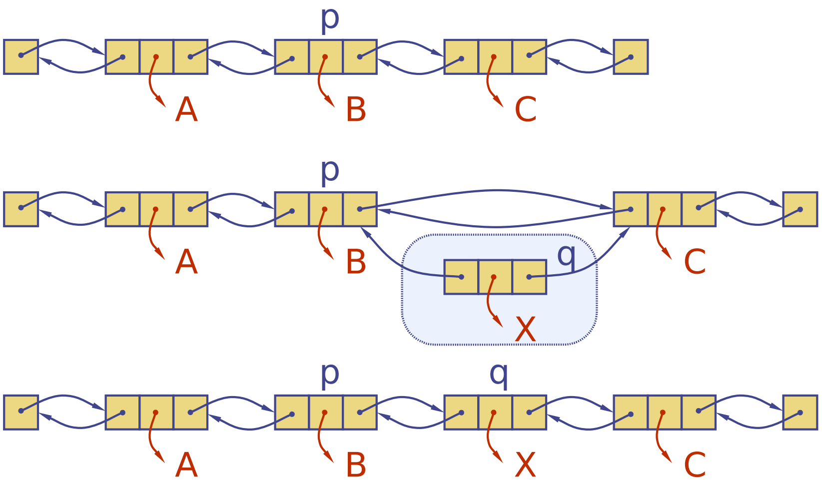
- a **header** node at the beginning of the list
- a **trailer** node at the end of the list

These 'dummy' nodes are known as **sentinels** (or guards). They do not store elements.



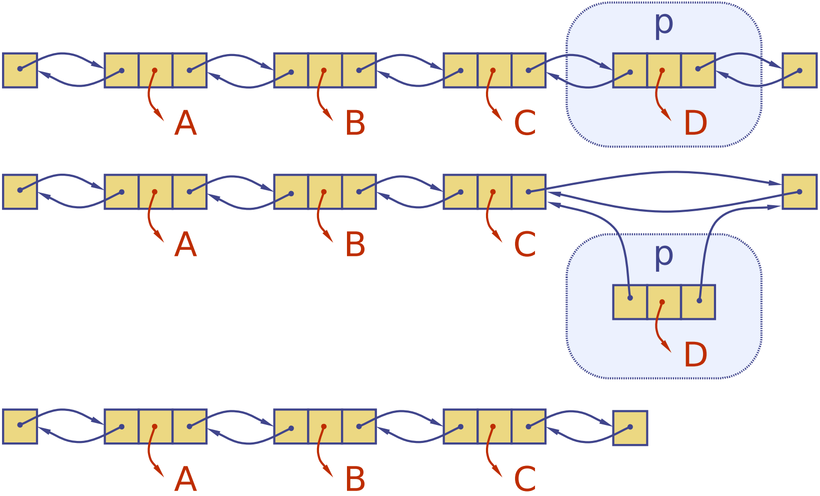
Insertion

Every insertion takes place between a pair of existing nodes.



Deletion

Every deletion takes place between a pair of existing nodes.



Interface of a Doubly Linked List

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.
removeLast()	Removes and returns the last element of the list.

Doubly Linked List Implementation: Node

```
/** A basic doubly linked list implementation. */
public class DoublyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;              // reference to the element stored at this node
    private Node<E> prev;           // reference to the previous node in the list
    private Node<E> next;           // reference to the subsequent node in the list
    public Node(E e, Node<E> p, Node<E> n) {
      element = e;
      prev = p;
      next = n;
    }
    public E getElement() { return element; }
    public Node<E> getPrev() { return prev; }
    public Node<E> getNext() { return next; }
    public void setPrev(Node<E> p) { prev = p; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------

```

Doubly Linked List Implementation I

```
// instance variables of the DoublyLinkedList
private Node<E> header;                          // header sentinel
private Node<E> trailer;                         // trailer sentinel
private int size = 0;                            // number of elements in the list
/** Constructs a new empty list. */
public DoublyLinkedList() {
  header = new Node<>(null, null, null);         // create header
  trailer = new Node<>(null, header, null);      // trailer is preceded by header
  header.setNext(trailer);                       // header is followed by trailer
}
/** Returns the number of elements in the linked list. */
public int size() { return size; }
/** Tests whether the linked list is empty. */
public boolean isEmpty() { return size == 0; }
/** Returns (but does not remove) the first element of the list. */
public E first() {
  if (isEmpty()) return null;
  return header.getNext().getElement();          // first element is beyond header
}
/** Returns (but does not remove) the last element of the list. */
public E last() {
  if (isEmpty()) return null;
  return trailer.getPrev().getElement();         // last element is before trailer
}
```

Doubly Linked List Implementation II

```
// public update methods
/** Adds element e to the front of the list. */
public void addFirst(E e) {
  addBetween(e, header, header.getNext());        // place just after the header
}
/** Adds element e to the end of the list. */
public void addLast(E e) {
  addBetween(e, trailer.getPrev(), trailer);      // place just before the trailer
}
/** Removes and returns the first element of the list. */
public E removeFirst() {
  if (isEmpty()) return null;                    // nothing to remove
  return remove(header.getNext());               // first element is beyond header
}
/** Removes and returns the last element of the list. */
public E removeLast() {
  if (isEmpty()) return null;                    // nothing to remove
  return remove(trailer.getPrev());              // last element is before trailer
}

```

Doubly Linked List Implementation III

```
// private update methods
/** Adds element e to the linked list in between the given nodes. */
private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
  // create and link a new node
  Node<E> newest = new Node<>(e, predecessor, successor);
  predecessor.setNext(newest);
  successor.setPrev(newest);
  size++;
}
/** Removes the given node from the list and returns its 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();
}
} //----------- end of DoublyLinkedList class -----------
```

Summary

Reading

Sections 3.2–3.6 of the main textbook

Questions?