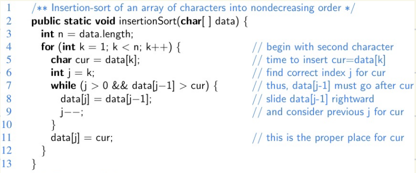
# Fundamental Data Structures

## 1. Arrays

- specified maximum capacity
- initially null entries

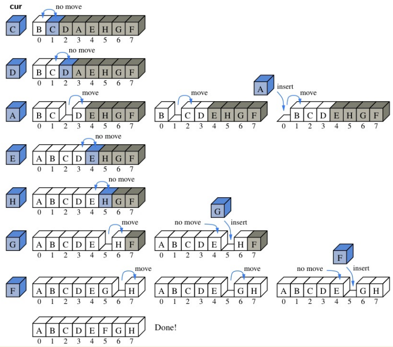
### 1.1. Insertion-Sort

- one element at a time



```
/** Insertion-sort of an array of characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
  int n = data.length;
  for (int k = 1; k < n; k++) {                 // begin with second character
    char cur = data[k];                          // time to insert cur=data[k]
    int j = k;                                   // find correct index j for cur
    while (j > 0 && data[j−1] > cur) {           // thus, data[j-1] must go after cur
      data[j] = data[j−1];                       // slide data[j-1] rightward
      j−−;                                       // and consider previous j for cur
    }
    data[j] = cur;                               // this is the proper place for cur
  }
}
```

**Code Fragment 3.6:** Java code for performing insertion-sort on a character array.



### 1.2 Pseudorandom Number Generator

$next = (a * cur + b) \% n$

$n \approx 2^{48}$ (java.util.Random)

seed – initial input in a generator in which the next value depends on the previous one(s)

### 1.3. Cryptography

- science of secret messages

$$\begin{tikzcd}\text{plaintext} \arrow[r, bend left, "\text{encryption}"] & \text{ciphertext} \arrow[l, bend left, "\text{decryption}"]\end{tikzcd}$$

Caesar cipher (shift letters)

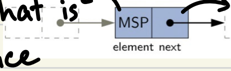
### 1.4. Two-Dimensional Array

array[#rows][#columns] -> matrix

## 2. Singly Linked Lists

– a collection of nodes that collectively form a linear sequence

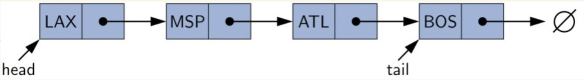
a reference to an object that is an element of the sequence



a reference to the next node of the list

head – a reference to the first node of the list

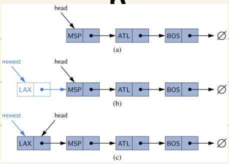
tail – last node of the list

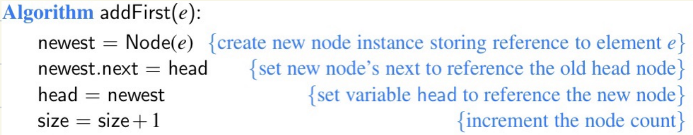


traversing

link/pointer hopping

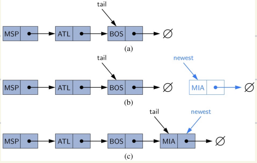
## 2.1. Inserting an Element at the Head of a Singly Linked List

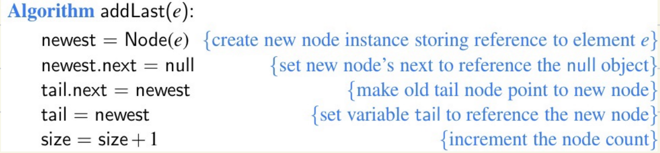




```
Algorithm addFirst(e):
    newest = Node(e)    {create new node instance storing reference to element e}
    newest.next = head        {set new node's next to reference the old head node}
    head = newest                   {set variable head to reference the new node}
    size = size + 1       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

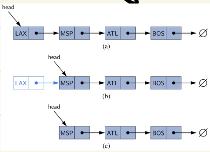
## 2.2. Inserting an Element at the Tail of a Singly Linked List

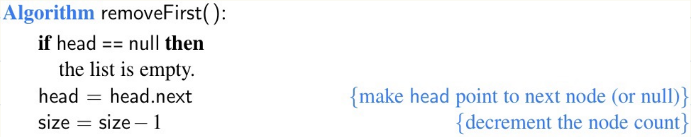




```
Algorithm addLast(e):
    newest = Node(e)   {create new node instance storing reference to element e}
    newest.next = null          {set new node's next to reference the null object}
    tail.next = newest                   {make old tail node point to new node}
    tail = newest             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

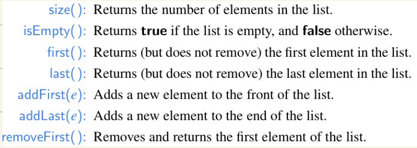
## 2.3. Removing an Element from a Singly Linked List





```
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}
```

## 2.4. Implementation



size( ): Returns the number of elements in the list.

isEmpty( ): Returns **true** if the list is empty, and **false** otherwise.

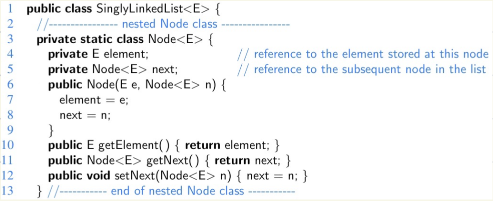
first( ): Returns (but does not remove) the first element in the list.

last( ): Returns (but does not remove) the last element in the list.

addFirst(e): Adds a new element to the front of the list.

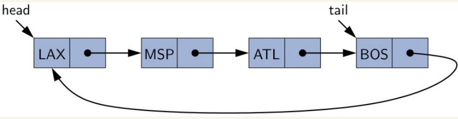
addLast(e): Adds a new element to the end of the list.

removeFirst( ): Removes and returns the first element of the list.

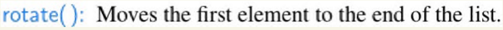


```java
public class SinglyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;            // reference to the element stored at this node
    private Node<E> next;         // reference to the subsequent node in the list
    public 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 -----------
```

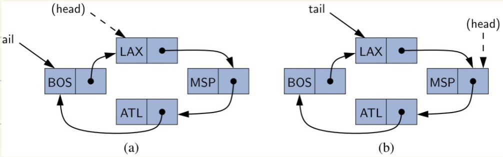
## 3. Circularly Linked Lists



- cyclic order
- the next reference of the tail node is set to refer back to the head of the list



rotate( ): Moves the first element to the end of the list.

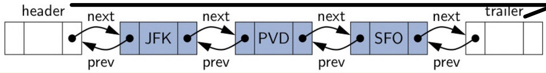


(a)

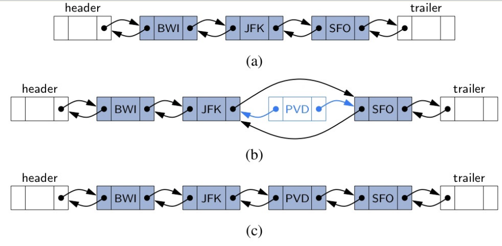
(b)

## 4. Doubly Linked Lists

- each node keeps an explicit reference to the node before it and a reference to the node after it



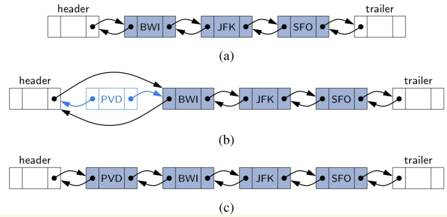
sentinels (guards) - do not store elements of the primary sequence



(a)

(b)

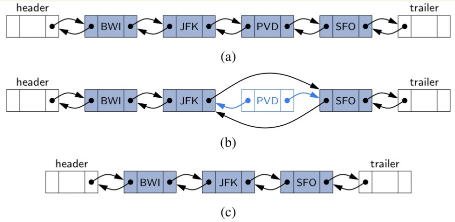
(c)



(a)

(b)

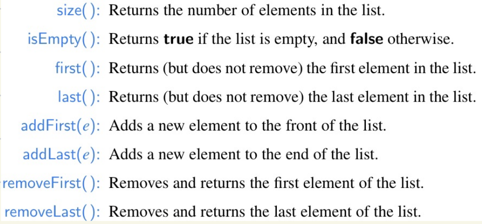
(c)



(a)

(b)

(c)



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.

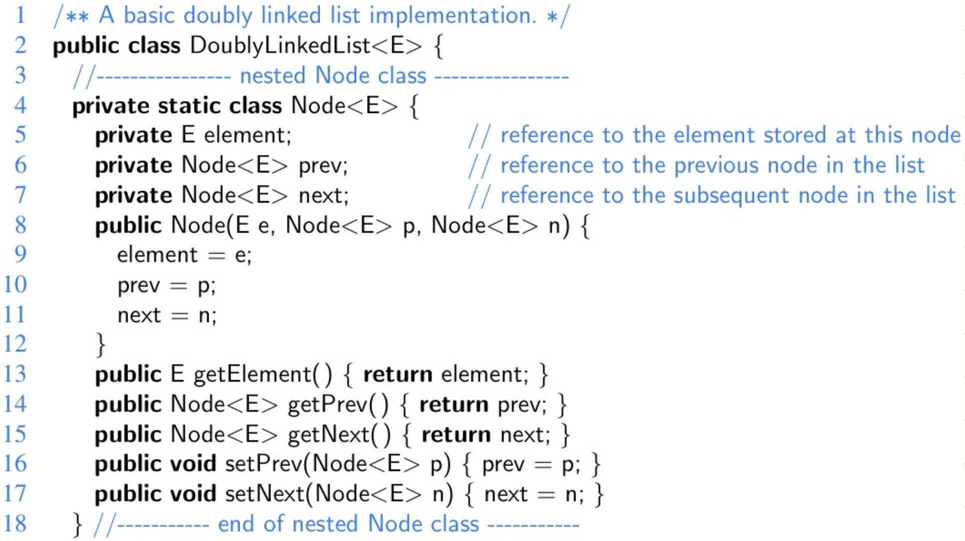
last( ): Returns (but does not remove) the last element in the list.

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.

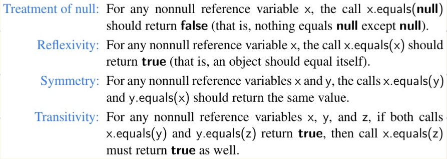
removeLast( ): Removes and returns the last element of the list.



```
/** 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 -----------
```

## 5. Equivalence Testing

### equivalence relation

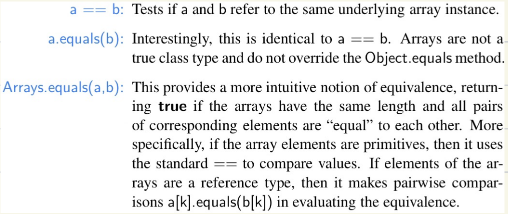


Treatment of null: For any nonnull reference variable x, the call x.equals(**null**) should return **false** (that is, nothing equals **null** except **null**).

Reflexivity: For any nonnull reference variable x, the call x.equals(x) should return **true** (that is, an object should equal itself).

Symmetry: For any nonnull reference variables x and y, the calls x.equals(y) and y.equals(x) should return the same value.

Transitivity: For any nonnull reference variables x, y, and z, if both calls x.equals(y) and y.equals(z) return **true**, then call x.equals(z) must return **true** as well.

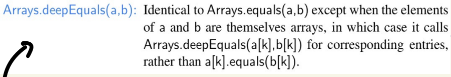


a == b: Tests if a and b refer to the same underlying array instance.

a.equals(b): Interestingly, this is identical to a == b. Arrays are not a true class type and do not override the Object.equals method.

Arrays.equals(a,b): This provides a more intuitive notion of equivalence, returning **true** if the arrays have the same length and all pairs of corresponding elements are "equal" to each other. More specifically, if the array elements are primitives, then it uses the standard == to compare values. If elements of the arrays are a reference type, then it makes pairwise comparisons a[k].equals(b[k]) in evaluating the equivalence.

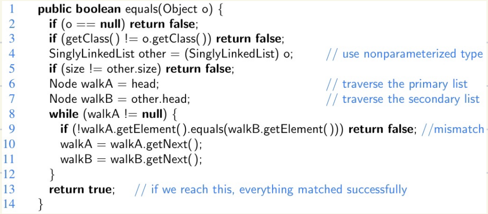
### compound objects



Arrays.deepEquals(a,b): Identical to Arrays.equals(a,b) except when the elements of a and b are themselves arrays, in which case it calls Arrays.deepEquals(a[k],b[k]) for corresponding entries, rather than a[k].equals(b[k]).

↑ 2D array

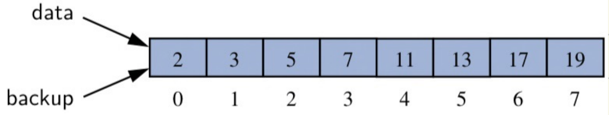
SSL ⟶

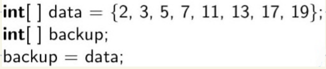


```
public boolean equals(Object o) {
  if (o == null) return false;
  if (getClass() != o.getClass()) return false;
  SinglyLinkedList other = (SinglyLinkedList) o;       // use nonparameterized type
  if (size != other.size) return false;
  Node walkA = head;                                   // traverse the primary list
  Node walkB = other.head;                             // traverse the secondary list
  while (walkA != null) {
    if (!walkA.getElement().equals(walkB.getElement())) return false; //mismatch
    walkA = walkA.getNext();
    walkB = walkB.getNext();
  }
  return true;    // if we reach this, everything matched successfully
}
```

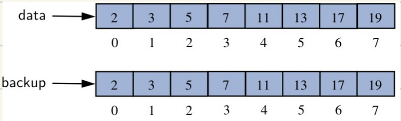
## 6. Cloning Data Structures

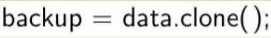
shallow copy – the value of each field of the new object is assigned to the corresponding field of the existing object





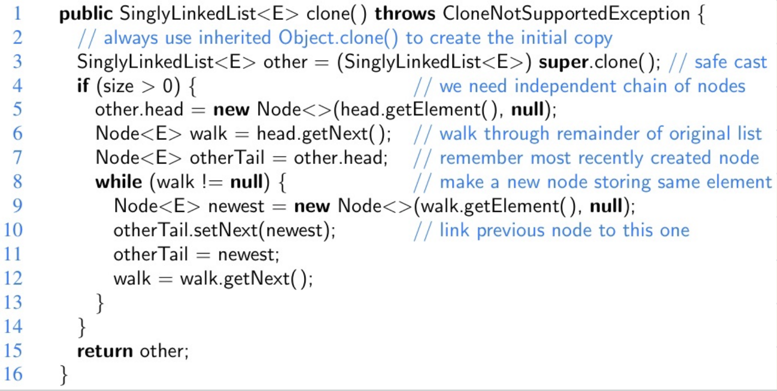
```
int[ ] data = {2, 3, 5, 7, 11, 13, 17, 19};
int[ ] backup;
backup = data;
```





```
backup = data.clone();
```

shallow copy



```
public SinglyLinkedList<E> clone() throws CloneNotSupportedException {
  // always use inherited Object.clone() to create the initial copy
  SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone(); // safe cast
  if (size > 0) {                          // we need independent chain of nodes
    other.head = new Node<>(head.getElement(), null);
    Node<E> walk = head.getNext();         // walk through remainder of original list
    Node<E> otherTail = other.head;        // remember most recently created node
    while (walk != null) {                 // make a new node storing same element
      Node<E> newest = new Node<>(walk.getElement(), null);
      otherTail.setNext(newest);           // link previous node to this one
      otherTail = newest;
      walk = walk.getNext();
    }
  }
  return other;
}
```