Data Structures

Augustin Cosse.

Spring 2021

February 16, 2021

Simple cryptography with character arrays

- An important application characters arrays and strings is cryptography
- Cryptography deals with the process of encryption where a message called plain text is converted intoa scrambled message called the ciphertext.
- Cryptography also studies corresponding ways of performing decryption which consists in turning a ciphertext back into its original plaintext
- the earliest encryption scheme is known as the Caesar cipher which is named after julius Caesar who used the scheme to protect important military messages

Simple cryptography with character arrays

- The Caesar cipher involves replacing each letter in a message with the letter that is a certain number of letters after it in the alphabet.
- In an english message, we might therefore replace each A with D, each B with E, each C with F and so on.
- We then continue this approach all the way up to W which is replaced with Z. Then we let the substitution pattern wrap around so that we replace X with A, Y with B and Z with C.

Simple cryptography with character arrays

- Since strings are immutable, we cannot directly edit an instance if we want to encrypt it.
- Instead, we will have to generate a new string
- Concretely we will create an equivalent array of characters, edit the array to get the excrypted message and then reassemble a new string
- Java has support for conversion between strings and character arrays. Given a string s = "`bird`", the method s.toCharArray() will return the character array A = {'b', 'i', 'r', 'd'}.
- The string `bird` can then be recovered as **new** String(A).

Simple cryptography with character arrays

- If we were to number our letters like array indices, we would then label A as 0, B as 1, and so on
- To set up our encryption scheme, the first step is to define the encoding array. We define this array such that **enc[0]** stores the character to which A is transformed, **enc[1]** stores the character to which B is mapped and so on
- In Java characters are represented in Unicode by integers and characters associated to uppercase are consecutive. If a character c stores the letter 'A', c - 'A' will thus return 0.
- The decryption array can be obtained by shifting the characters in the opposite direction

Simple cryptography with character arrays

- To define a Caesar cipher with a rotation or r character, our constructor will define the associated encryption and decryption arrays
- Note the use of the modulo. If the shift is 26, it should not change anything

```
public class CaesarCipher {
  protected char[ ] encoder = new char[26];
  protected char[ ] decoder = new char[26];
  /** Constructor */
  public CaesarCipher(int rotation) {
    for (int k=0; k < 26; k++) {
        encoder[k]=(char)('A' + (k + rotation)%26);
        decoder[k]=(char)('A' + (k - rotation + 26)%26);
} }
```

Simple cryptography with character arrays

- the **mod** operator returns the remainder after performing the (integer) division. 26 mod 26 is 0, 27 mod 26 is 1

```
public class CaesarCipher {
  protected char[ ] encoder = new char[26];
  protected char[ ] decoder = new char[26];
  /** Constructor */
  public CaesarCipher(int rotation) {
    for (int k=0; k < 26; k++) {
        encoder[k]=(char)('A' + (k + rotation)%26);
        decoder[k]=(char)('A' + (k - rotation + 26)%26);
} }
```

Simple cryptography with character arrays

- On top of the constructor, we can add methods that will encrypt and decrypt the message through a transformation applied by means of the encoder and decoder

```
public String encrypt(String message) {
   return transform(message, encoder);
}
public String decrypt(String secret) {
   return transform(secret, decoder);
}
private String transform(String original, char[ ] code) {
   char[ ] msg = original.toCharArray( );
   for (int k=0; k < msg.length; k++)
      if (Character.isUpperCase(msg[k])) {
         int j = msg[k] - 'A'; // get the shift
         msg[k] = code[j]; // obtain character in code
}
return new String(msg);}
```

Simple cryptography with character arrays

- Finally we add a main to test our encryption/decryption

```
public static void main(String[ ] args) {
    CaesarCipher cipher = new CaesarCipher(3);
    System.out.println("Encryption code = " +
                                new String(cipher.encoder));
    System.out.println("Decryption code = " +
                                new String(cipher.decoder));
    String message = "THE EAGLE IS IN PLAY;
                               MEET AT JOE'S.";
    String coded = cipher.encrypt(message);
    System.out.println("Secret: " + coded);
    String answer = cipher.decrypt(coded);
    System.out.println("Message: " + answer); }}
```

Two dimensional arrays and games

- Arrays in java are one dimensional. There is nevertheless a way to define two dimensional arrays.
- In Java, two dimensional arrays are usually defined as arrays of arrays. That is we define a two dimensional array as an array in which each cell refers to another array
- In Java, two dimensional arrays are defined as follows

```
int [] [] data = new int[8][10];
```

- The above statement creates a two dimensional "array of arrays" data which is 8 by 10 having 8 rows and 10 columns. That is data is an array of length 8 such that each element of data is an array of length 10 of integers.

Two dimensional arrays and games

- The code below is a valid use of the array data

```
data[i][i+1] = data[i][i] + 3;
j = data.length;
k = data[4].length;
```

- Two-dimensional arrays have many applications in numerical analysis. In the lab, we will explore an application of two dimensional arrays for implementing a simple positional game.

Two dimensional arrays and games

- Tic tac toe is a game played on a 3 by 3 board. Two players 'X' and 'O' alternate in placing their respective marks on the cells of the board starting with player 'X'
- If either player succeeds in getting three of his or her marks in a row, column or diagonal, that player wins.
- To simulate this game, we can use a two dimensional array, **board** to store the board.
- To simplify the code, we will store the current state of the game by associating an entry of the board with 1 ('X'), 0 (no symbol yet) and '-1' ('O')
- To check whether a victory has been achieved, it suffices to keep track of any row, colmn or diagonal that sums up to 3 or -3.

Two dimensional arrays and games

- The game can then be implemented through the following steps

```
public class TicTacToe {
  public static final int X = 1, O = -1; // players
  public static final int EMPTY = 0; // empty cell
  private int board[ ][ ] = new int[3][3]; // board
  private int player; // current player
  /** Constructor */
  public TicTacToe( ) { clearBoard( ); }
  /** Clears the board */
  public void clearBoard( ) {
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        board[i][j] = EMPTY;
     player = X; // the first player is 'X'}
// ....
}
```

Two dimensional arrays and games

- And we can for example add to this class a method to check for a win

```
public class TicTacToe {
// ....

public boolean isWin(int mark) {
  return ((board[0][0] + board[0][1] + board[0][2] == mark*3)
  || (board[1][0] + board[1][1] + board[1][2] == mark*3)
  || (board[2][0] + board[2][1] + board[2][2] == mark*3)
  || (board[0][0] + board[1][0] + board[2][0] == mark*3)
  || (board[0][1] + board[1][1] + board[2][1] == mark*3)
  || (board[0][2] + board[1][2] + board[2][2] == mark*3)
  || (board[0][0] + board[1][1] + board[2][2] == mark*3)
  || (board[2][0] + board[1][1] + board[0][2] == mark*3)); }
}
```

Two dimensional arrays and games

- As well as a method to play a move by specifying a position and a player

```
/** Puts an X or O mark at position i,j. */
public void putMark(int i, int j)
                            throws IllegalArgumentException {
  if ((i < 0) || (i > 2) || (j < 0) || (j > 2))
    throw new IllegalArgumentException("Invalid position");
  if (board[i][j] != EMPTY)
    throw new IllegalArgumentException("position occupied");
  board[i][j] = player;
  player = - player; // switch players
}
```

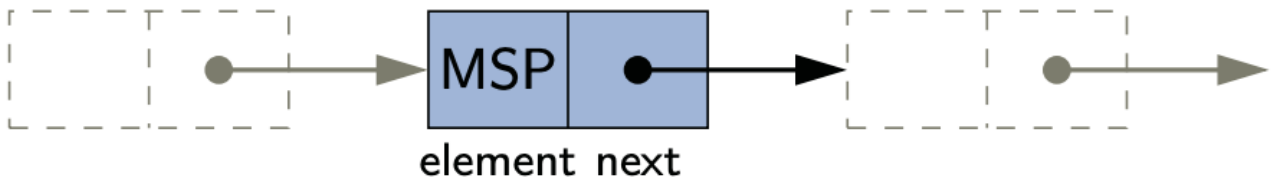
Singly linked lists

- Arrays are great for storing things in certain order but they have drawbacks. The capacity of the array must be fixed when it is created and insertions and deletions at interior positions of an array can be time consuming if many elements must be shifted

- A first alternative to arrays is provided by a data structure known as Linked List

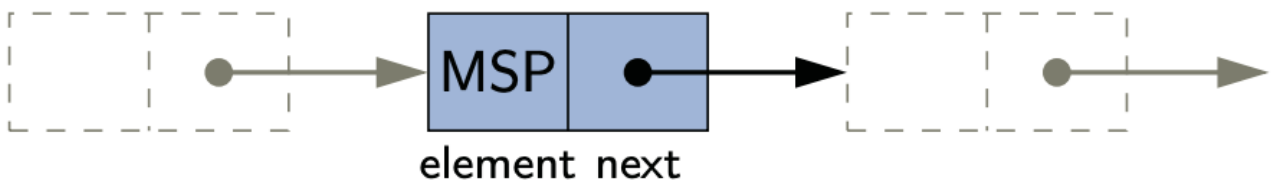
- A linked list in its simplest form is a collection of nodes that collectively form a sequence as show below

- Each node stores a reference to an object that is an element of the sequence as well as a reference to the next node of the list



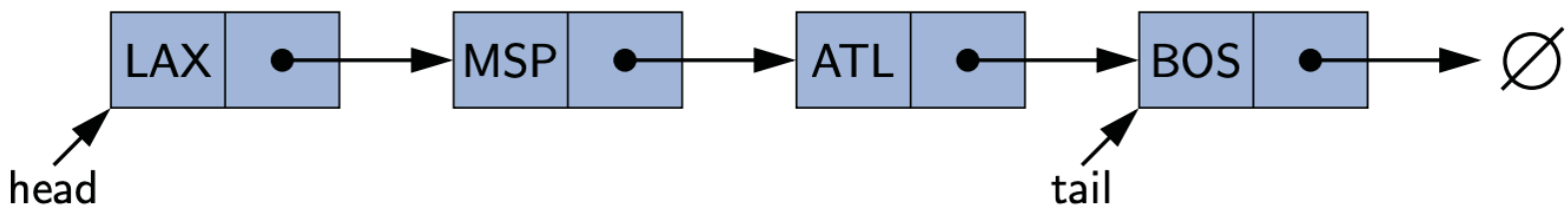
Singly linked lists

- A linked list representation relies on the collaboration of many objects. Minimally 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
- The tail of a list can be found by traversing the linked list, starting from the head and moving from one node to the other by following each node's reference.
- The tail is identified as having **null** as its next reference.



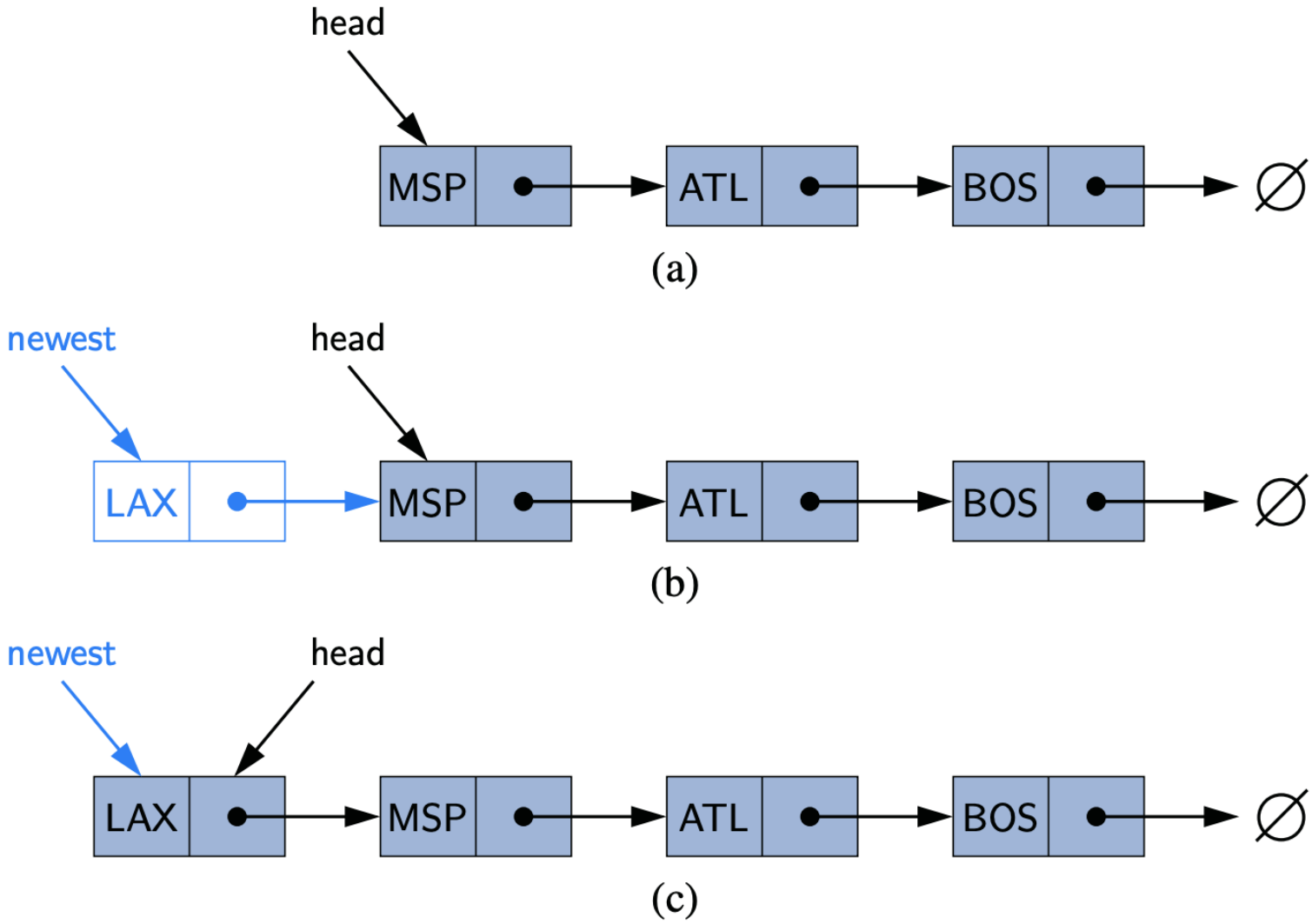
Singly linked lists

- The process is also known as link hopping or pointer hopping
- Storing an explicit reference to the tail node is usually used as way to avoid the traversal.
- In a similar regard, it is also common for a linked list instance to keep a count of the total number of nodes that are comprised in the list (also known as the size of the list).



(Example of a singly linked list whose elements are strings encoding airport names)

Singly linked lists



(Insertion of an element at the head of the list)

Singly linked lists

- An important property of a linked list is that it does not have a predetermined size fixed size. It uses space proportional to its current number of elements
- When using a singly linked list, we can easily insert an element at the head of the list (unlike arrays) with the pseudo code below

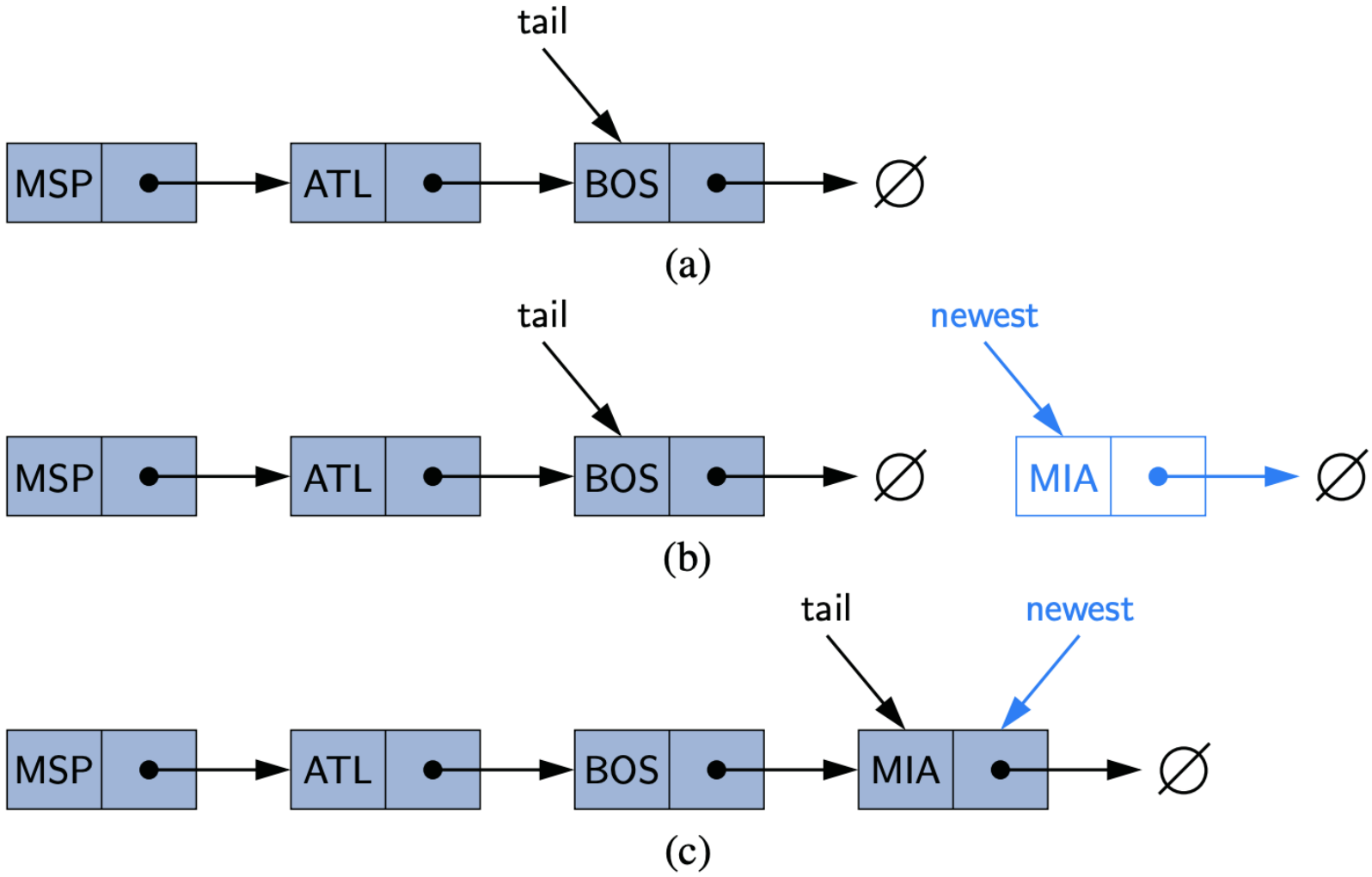
```
newest = Node(e)
// create a new instance storing reference to e
newest.next = head
// set new node's next reference to the old head node
head = newest
// set variable head to reference the new node
size = size+1
// increment node count
```

Singly linked lists

- Similarly, linked list make it easier to insert elements at the tail of the list, provided we keep a reference to the tail node.
- The addition of a tail node can be done by first creating the node, assigning its next reference to the **null**, then assigning the next reference of the tail to point to this new node.

```
newest = Node(e)
// create new node instance storing reference to e
newest.next = null
// set new node's next reference to the null object
tail.next = newest
// make old tail node point to the new node
tail = newest
// set variable tail to reference to the new node
size = size+1
```

Singly linked lists



(Insertion of an element at the tail of the list)

Singly linked lists

- Removing an element at the head of a singly linked list is essentially the reverse operation as inserting a new element at the head.

```
if head == null then
        the list is empty
head = head.next // make head point to the null node
size = size - 1 // decrement node count
```

- Unfortunately, deleting the last node is not as easy. Even if we keep a reference to the last node of the list, in order to delete this node, we must also maintain a reference to the second to last node in the list.
- Yet we cannot access the second to last node in the list from the last as the only reference we have on this last node is a next reference that points to the **null**.

Singly linked lists

- In other words, to access the second to last node in order to remove the last one, we have no other choice than to start from the head and follow all the next references until we reach the node that points to the last node.
- Such a sequence of link-hopping operations can take a long time
- In order for the search to be more efficient, a solution is to make the list doubly linked.

Singly linked lists

- In other words, to access the second to last node in order to remove the last one, we have no other choice than to start from the head and follow all the next references until we reach the node that points to the last node.
- Such a sequence of link-hopping operations can take a long time
- In order for the search to be more efficient, a solution is to make the list doubly linked.

Singly linked lists

- Before discussing doubly linked list, we will first study the concrete implementation of singly linked lists
- Because it does not matter what type of elements are stored in the list, we will use Java's generic framework to define the singly linked list class with a generic framework <E> which is used to represent the user's desired type

Reminders on Generic types

- Recall that there several approaches at writing generic classes and methods.
- The first one is to rely on the **Object** class which in Java is the universal supertype of all objects
- The second approach is to use Java's generic framework

```
public class Pair<A,B> {
A first;
B second;
public Pair(A a, B b) { // constructor
first = a;
second = b;
}
public A getFirst( ) { return first; }
public B getSecond( ) { return second;}}
```

Reminders on Generic types

- From the statement below, we explicitly state that we wish to have String serve in place of type A, and Double serve in place of type B for the pair known as bid
- The actual types for generic programming must be object types, which is why we use the wrapper class Double instead of the primitive type double.
- We can then instantiate the generic class using the lines below

```
Pair <String, Double> bid;
bid = new Pair<>("ORCL", 32.07); // rely on type inf
```

- After the new operator, we provide the name of the generic class followed by an empty set of angle brackets (known as the "diamond"), then the parameters to the constructor.

Reminders on Generic types

- An instance of the generic class is created with the actual type for the formal type parameters determined based upon the original declaration of the variables to which it is assigned (bid in this example).
- This process is known as type inference and was first introduced in Java SE7

```
Pair <String, Double> bid;
bid = new Pair<>("ORCL", 32.07); // rely on type inf
```

Reminders on Generic types

- it is also possible to use a type that existed prior to Java SE 7 in which the generic type parameters are explicitly specified between angle brackets during instantiation
- Using this style, our previous example would be implemented as

```
bid = new Pair<String,Double>("ORCL", 32.07);
```

Reminders on Generic types

- The advantage with the generics framework is that there is no longer any need for explicit narrowing cast from **Object** to a more specific type.

- Since bid was declared with type parameters <String, Double>, the return type of the **getFirst()** Method which appear in the class, will automatically be String and the return type of the **getSecond()** method will be Double.

```
Pair <String, Double> bid;
bid = new Pair<>("ORCL", 32.07); // rely on type inf
```

- Unlike the class style, we can then make the statement (w/o casting)

```
String stock = bid.getFirst();
double price = bid.getSecond();
```

Reminders on Generic types

- There is an important caveat related to generic type and the use of arrays.
- Although Java allows the declaration of an array storing a parametrized type, it does not technically allow the instantiation of new arrays involving those types.
- Fortunately it allows an array defined with a parametrized type to be initialized with a newly created, non parametric array which can be cast to the parametrized type. This idea is illustrated below.

```
Pair<String, Double> [] holdings;
holdings  = new Pair<String, Double>[25];
// illegal -> compile error
holdings = new Pair[25];
holdings[0] = new Pair<>("ORCL", 32.07);
```

Reminders on Generic types

- Another illustration of this impossibility to simultaneously declare and instantiate an array of a parametrized type is given by the class **portfolio** below

```
public class Portfolio<T>{
T[] data;
public portfolio(int capacity){
  data = new T[capacity]; // illegal, compile error
  data = (T[]) new Object[capacity]; //legal but compile warn
}}
```

- In this case, the class can delcare an array of type T[] but it cannot directly instantiate such an array. Instead, a common approach is to instantiate an array of type Object and then make a narrowing cast to type T[].

Generic methods

- Java generics framework also allows us to define generic versions of individual methods
- To do so, it suffices to include a generic formal type among the method modifiers

```
public class GenericDemo{
  public static <T> void reverse(T[] data){
   int low =0, high = data.length - 1;
   while (low<high){
    T temp = data[low];
    data[low++] = data[high];
    data[high--] = temp;

}
}}
```

Generic methods

- Note the use of the <T> type modifier to declare the method to be generic and the use of the type <T > within the method body when declaring the local variable **temp**

- The method can then be used using the syntax **GenericDemo.reverse(books)** with type inference determining the generic type (Note that here declaration and instantiation are simultaneous and done with the generic type.)

```
public class GenericDemo{
  public static <T> void reverse(T[] data){
   int low =0, high = data.length - 1;
   while (low<high){
    T temp = data[low];
    data[low++] = data[high];
    data[high--] = temp;}
}}
```

Generic methods

- By default, when using the a type name such as T in a generic class, or method, a user can specify any type as the actual type of the generic
- A formal parameter type can be restricted by using the **extends** keyword followed by a class or interface
- In that case, only a type that satisfies the the stated condition is allowed to substitute for the parameter. As an example, we can declare a generic **ShoppingCart** class that could only be instantiated with a type that satisfies the Sellable interface

```
public class ShoppingCart<T extends Sellable>
```

- Within the class definition, we will then be allowed to call methods that appear in the interface.

Nested classes

- Another concept that we will use in our implementation of singly linked lists are nested classes
- Java allows a class definition to be nested inside the definition of another class
- The main use for nested classes is when defining a class that is strongly affiliated with another class
- This can help increase encapsulation and reduce name conflicts

```
public class CreditCard{
  private static class Transaction {
/* details omitted*/}
  //...
  Transaction[] history
}
```

Nested classes

- Nested classes are valuable when implementing data structures as an auxiliary class can help navigate a primary data structure.

- The containing class is known as the outer class. The nested class is a member of the outer class and its fully qualified name is **OuterName.NestedName** (CreditCard.Transaction in the example below) although we might refer to it as **Transaction** from within the **CreditCard** class.

```
public class CreditCard{
  private static class Transaction {
/* details omitted*/}
  //...
  Transaction[] history
}
```

Nested classes

- Nested classes help reduce name collisions as it is perfectly acceptable to have other classes named Transaction within other outer classes.
- A nested class has an independent set of modifiers from the outer class. Visibility modifiers (e.g. public private,...) effect whether the nested class definition is accessible beyond the outer class definition.
- For example a private nested class can be used can be used by the outer class only

```
public class CreditCard{
  private static class Transaction {
/* details omitted*/}
  //...
  Transaction[] history}
```

Nested classes

- A nested class can also be designated as either static or non static
- A static nested class is like a traditional class. Its instances have no association with any specific instance of the outer class
- A non static nested class is more commonly known as an inner class in Java. An instance of an inner class can only be created from within a nonstatic method of the outer class and that inner instance becomes associated with the outer instance that creates it
- Each inner instance stores a reference to its associated outer instance accessible from within the inner class methods using the syntax **OuterName.this** as opposed to **this** which refers to the outer instance.

Back to singly linked lists

- In the implementation that we will study, the **Node** class is defined as nested class which provides strong encapsulation and will allow us to differentiate this from forms of nodes we may define for use in other structures.

- Consider the first part of the singly linked list

```
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; }
```

Back to singly linked lists

- Note the nested inner class Node<E>
- Also note how the type defined by the inner class is used inside this class to define the variable **next** and the method **getNext**

```
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; }
```

Back to singly linked lists

- On top of the **node** inner class, we add the variables head and tail to the class as well as accessor methods to return the two

```
public class SinglyLinkedList<E> {
   //------- ..... (see previous slide) --------
 private Node<E> head = null; // head node
 private Node<E> tail = null; // last node
 private int size = 0; // number of nodes
 public SinglyLinkedList( ) { } // initialize list
 // access methods
 public int size( ) { return size; }
 public boolean isEmpty( ) { return size == 0; }
 public E first( ) { // returns 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( );
}
```

Back to singly linked lists

- As we already discussed, we also consider method for the addition of a head or a tail node..

```
public void addFirst(E e) { // adds element e to the front
    head = new Node<>(e, head); // create and link new node
    if (size == 0)
        tail = head; // special case: new node becomes tail
    size++;}
public void addLast(E e) { // adds element e to the end
    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++;
}
```

Back to singly linked lists

- As we already discussed, we also consider method for the addition of a head or a tail node..

```
public E removeFirst( ) { // removes and returns 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;}}
```

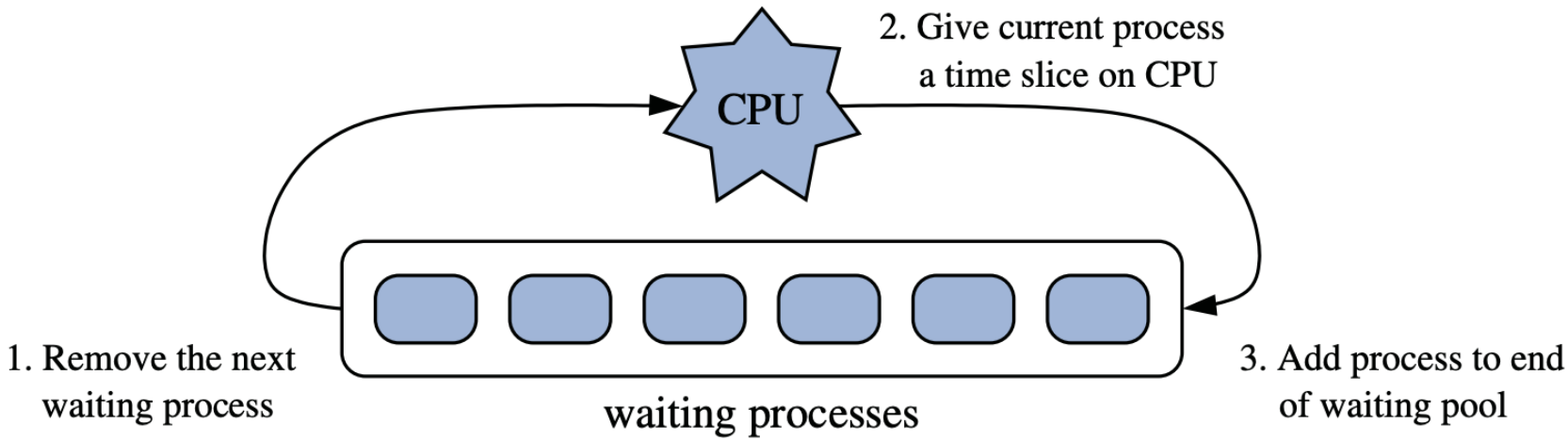
Circularly linked lists

- Linked lists are traditionally viewed as storing a sequence of items in a linear order, from first to last
- There are however many applications in which data can be more naturally viewed as having a cyclic order, with well defined neighboring relationships but no fixed beginning or end
- Examples include multi-player games that are turn based with player A taking a turn, then player B then player C, and so on but then eventually coming back to player A again with the pattern repeating.
- As another example, city buses and subways often run on a continuous loop, making stops in a continuous order but with no designated first or last stop per se.

Round Robin Scheduling

- One of the most important roles of an operating system is to manage the many processes that are active on a computer, including the scheduling of those processes on one or more CPUs
- In order to support the responsiveness of an rbitrary number of concurrent processes, most operating systems allows processes to effectively share use of the CPUs using some form of an algorithm known as Round Robin Scheduling.
- In Round Robin scheduling, a process is given a short turn to execute known as a time slice but it is interrupted when the slice ends, even if its job is not yet completed.
- Each active process is given its own time slice, taking turn in a cyclic order.

Round Robin Scheduling



Round Robin Scheduling

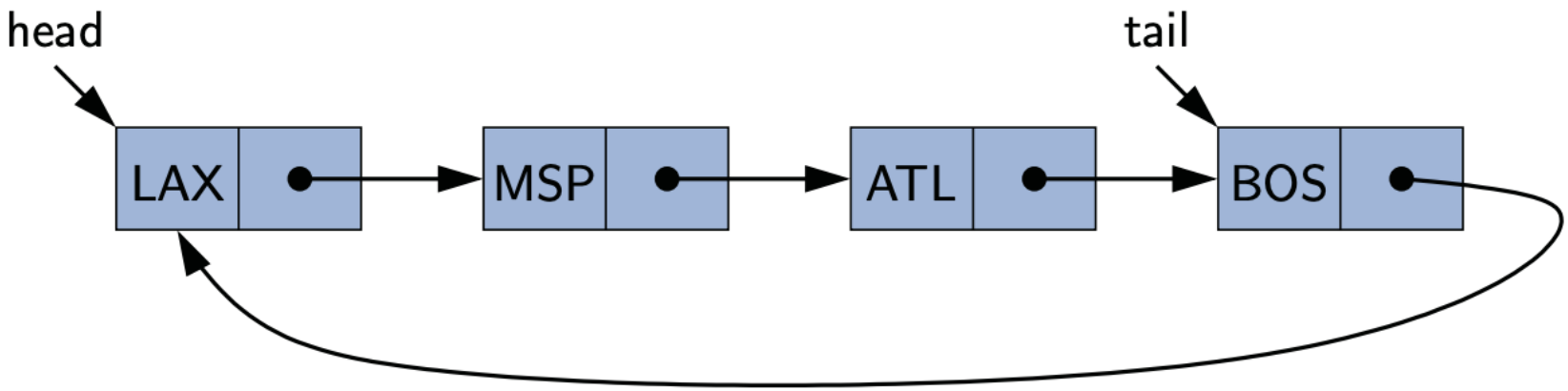
- New processes can be added to the system and processes that complete their work can be removed
- A round robin process could be implemented with a traditional linked list by relying on the following three steps:
  - process p = L.removeFirst()
  - Give a time slice to process p
  - L.addLast(p)
- There are however several drawbacks to these single linked approach:
  - First, it is unnecessarily inefficient to repeatedly throw away a node from one end of the list, only to create a new node for the same element when reinserting it
  - Another disadvantage are multiples updates needed to increment the list size, each time an element is removed/added

Round Robin Scheduling

- As we will see, a more efficient structure can be derived by leveraging circlarly linked lists
- A circularly linked list is essentially a singly linked list in which the **next** reference to the tail node is set to refer back to the first (i.e. head) node of the list (rather than **null**)
- As a consequence, we can keep many of the public behaviors of that we implemented in the singly linked lists
- On top of those, we also need to insert an additional method to change the ordering of the elements (i.e. rotate the last element at the top of the list)

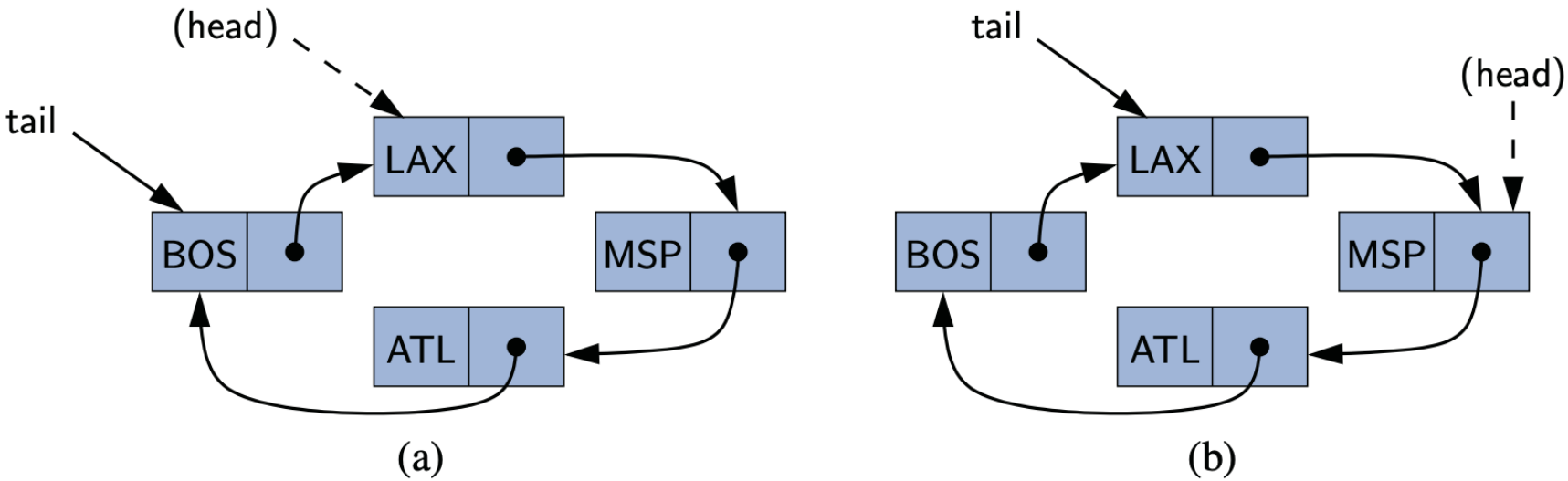
Round Robin Scheduling

- With such lists, we can then implement robin scheduling through the two steps:
  - Give a time slice to process **C.first()**
  - Then use the nex method **C.rotate()**



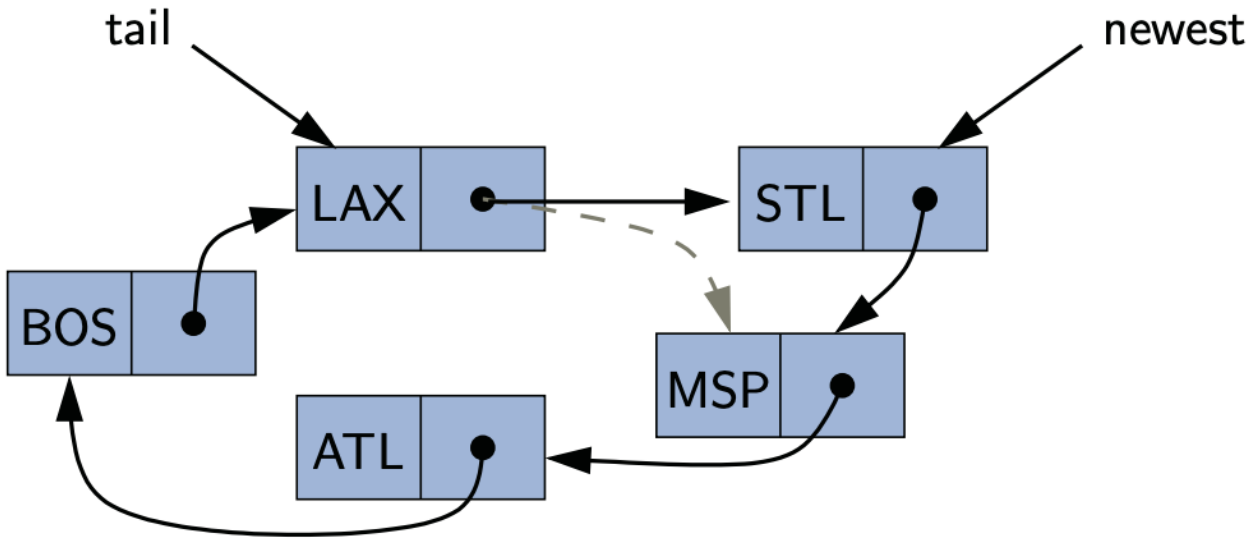
Round Robin Scheduling

- Note that since the tail is now connected to the head, we can only retain a reference to either of those two and use **tail/head.getNext()** to access this node
- If we keep tail in particular, updating a circularly linked list is easy, we simply need to update the identity of the tail by moving it to the node given by **tail.next()** as shown below



Round Robin Scheduling

- As before we will want to add and remove nodes from the list.
- We can for example add a new node right after the tail
- Similarly we can remove the first node by simply bypassing his simplicit head of the list



From Singly Linked to Circularly linked

- As before, we provide the class with a tail node, head node (=tail.next) and methods to access those nodes

```
public class CircularlyLinkedList<E> {
//... (nested class identical to SinglyLinkedList)
// instance variables
    private Node<E> tail = null;
    private int size = 0; // number of nodes
    public CircularlyLinkedList( ) { } // constructs empty list
    // access methods
    public int size( ) { return size; }
    public boolean isEmpty( ) { return size == 0; }
    public E first( ) { // returns first element
        if (isEmpty( )) return null;
        return tail.getNext( ).getElement( );
    }
    public E last( ) { // returns last element
        if (isEmpty( )) return null;
        return tail.getElement( );
    }
```

From Singly Linked to Circularly linked

- We are left with the methods **addFirst**, **addLast** and **rotate**

```
public void rotate( ) { // rotate first element to back
    if (tail != null) // if empty, do nothing
         tail = tail.getNext( ); //  head becomes tail
     }
 public void addFirst(E e) { // adds e to the front
     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 e to the end of the list
     addFirst(e); // insert new element at front
     tail = tail.getNext( ); // new element becomes the tail
 }
```

From Singly Linked to Circularly linked

- Note how the **addLast** function can be designed through the addFirst by just changing the node stored in tail.

```
public void rotate( ) { // rotate first element to back
    if (tail != null) // if empty, do nothing
         tail = tail.getNext( ); //  head becomes tail
     }
 public void addFirst(E e) { // adds e to the front
     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 e to the end of the list
     addFirst(e); // insert new element at front
     tail = tail.getNext( ); // new element becomes the tail
 }
```

From Singly Linked to Circularly linked

- We conclude with the **removeFirst()** method which is implemented by bypassing the current head.

```
public E removeFirst( ) { // removes and returns head
    if (isEmpty( )) return null;
    Node<E> head = tail.getNext( );
    if (head == tail) tail = null;
     else tail.setNext(head.getNext( ));
     size--;
     return head.getElement( );
 }}
```

From Singly Linked to Circularly linked

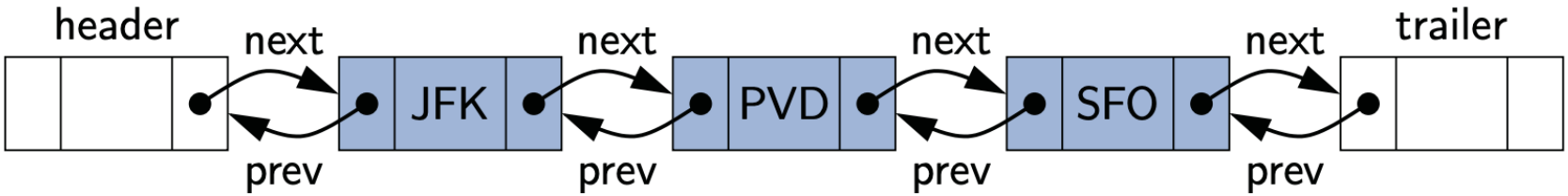
- In a singly linked list, each node maintains a reference to the node that is immediately after it
- There are however limitations that stem from the asymetry of such representation
- As an example of this, we saw that one can efficiently remove a node from the beginning of the list (since it suffices to define our new head as the **.next** node) but that it is much less efficient to remove a node at the end of the list (as we cannot access the second to last element easily)

Doubly linked Lists

- To improve this, we can naturally define a linked list in which each node keeps a reference to the node before it and a reference to the node after it
- Such a structure is known as a Doubly linked List
- Doubly linked lists allow a greater variety of O(1)-time operations including insertion and deletions at arbitrary positions in the list
- On top of the **next** reference, we will now also consider the **prev** reference to refer to a node that is located before the current node

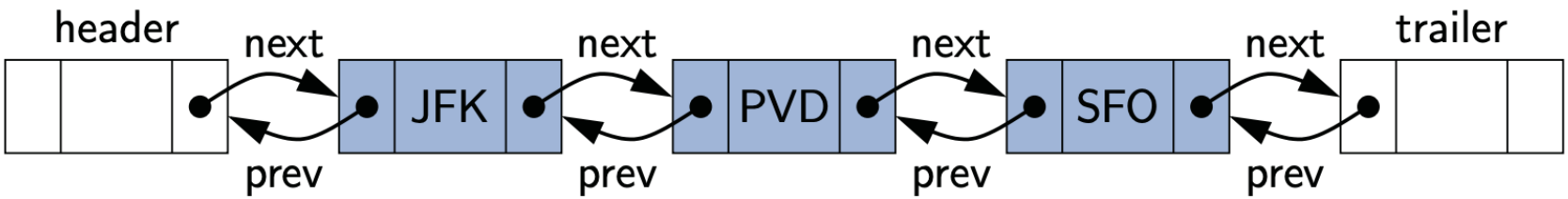
Doubly linked Lists

- We will consider two special nodes at the beginning and end of the list: The header node (beginning) and the trailer node (end).
- The header and the trailer nodes are known as sentinels and they do not store elements of the primary sequence (see below)



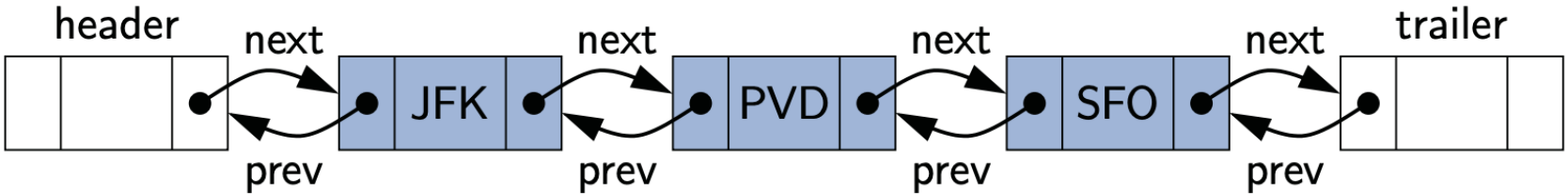
Doubly linked Lists

- When initializing a doubly linked list, we will start by setting the **next** field of the header to refer to the trailer and the **prev** field of the trailer to refer to the header.
- In a non emtpy doubly linked list, the **next** field of the header will be connected to the first node containing the first real element of a sequence while the **prev** field of the trailer will refer to the last real element of the sequence.



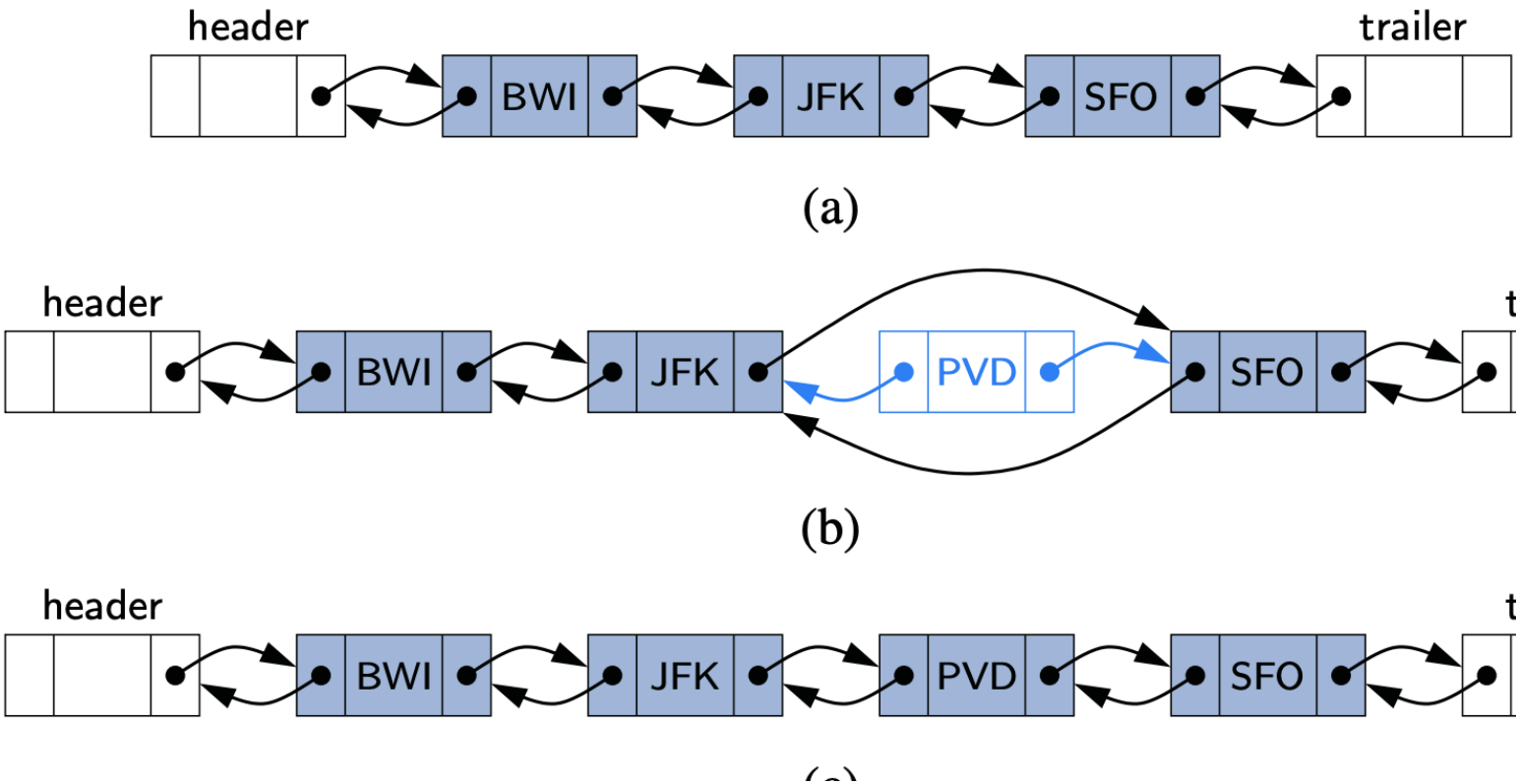
Doubly linked Lists

- Doubly linked lists could be treated without the sentinel nodes. However, this simplifies the operations.
- Every node is always inserted between two existing nodes
- And every node that might be deleted is also guaranteed to have the two **prev** and **next** fields
- This is clearly better than the singly linked list for which we had to treat the case of the insertion of a node into a n empty list separately for example.



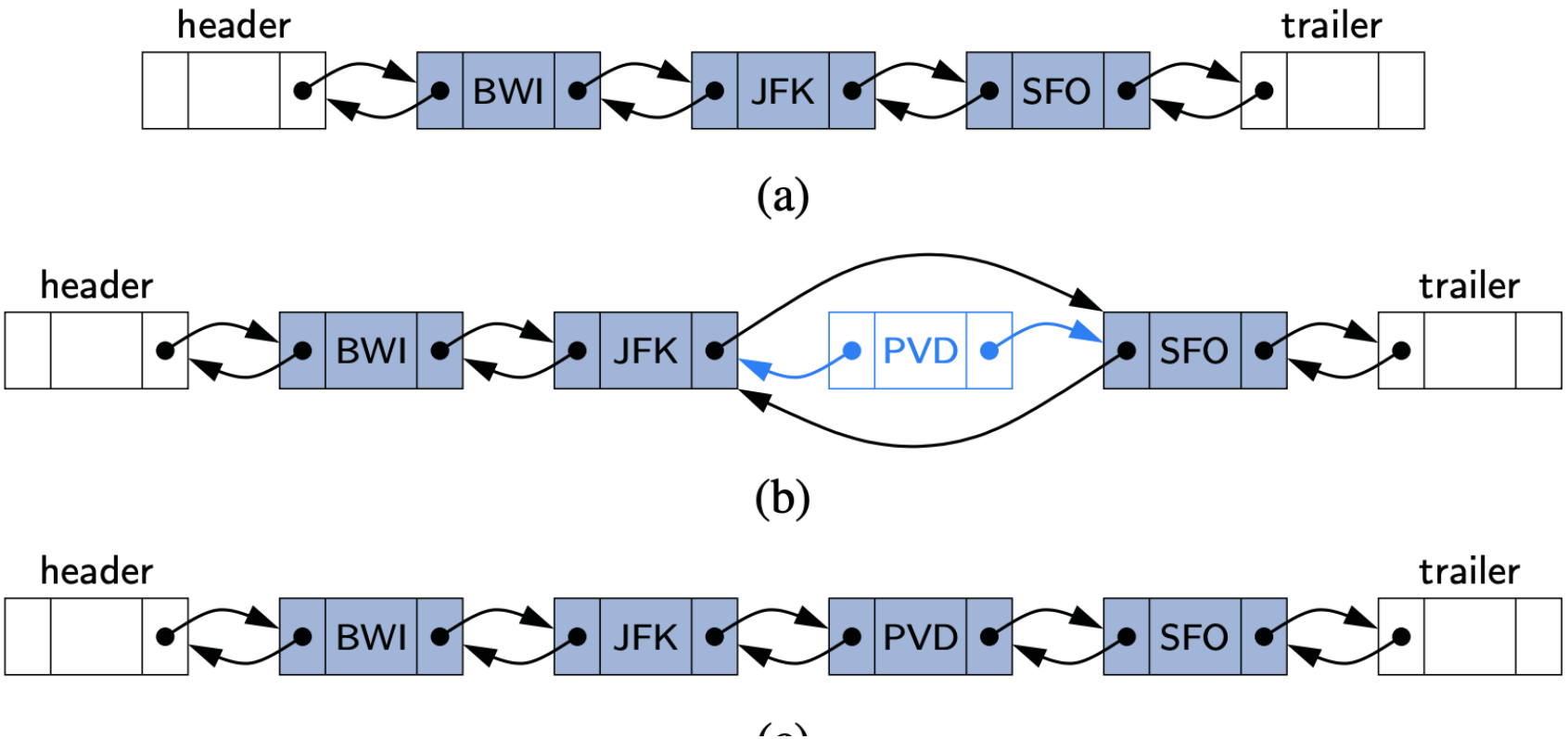
Doubly linked Lists

- Every insertion of a new node in a doubly linked list will take place between two existing nodes



Doubly linked Lists

- Similarly the deletion of a node proceeds by connecting the neighbors of the node that is being deleted



Doubly linked List: concrete implementation

- Concretely, in our implementation of the DoublyLinkedList Class, we will consider the following functions

size()	returns the number of elements in the list
isEmpty()	returns **true** if the list is empty
first	Returns the first element in the list
last	Returns 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
removeLast()	Removes last element

When the list is empty and we are asked to return an element, we will return the **null** reference.

Doubly linked List: concrete implementation

- The inner node class is similar to the SinglyLinked List class except that we now consider a **prev** reference.

```
public class DoublyLinkedList<E> {
    //------ nested Node class ----------
    private static class Node<E> {
        private E element;
        private Node<E> prev;
        private Node<E> next;
        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; }
}
```

Doubly linked List: concrete implementation

- The list is defined with the header, the trailer, the size and a constructor.

```
private Node<E> header; // header sentinel
private Node<E> trailer; // trailer sentinel
private int size = 0; // number of elements in the list
/** Constructor */
public DoublyLinkedList( ) {
header = new Node<>(null, null, null); // create header
trailer = new Node<>(null, header, null);
header.setNext(trailer); }
/** Returns the number of elements in list. */
public int Getsize( ) { return size; }
/** Tests whether the linked list is empty. */
public boolean isEmpty( ) { return size == 0; }
/** Returns first element */
public E first( ) {
if (isEmpty( )) return null;
return header.getNext( ).getElement( ); }
/** Returns last element */
public E last( ) {
if (isEmpty( )) return null;
return trailer.getPrev( ).getElement( ); }
```

Doubly linked List: concrete implementation

- We then add methods for the addition of nodes at the beginning, end or anywhere in the list

```
public void addFirst(E e) {
    addBetween(e, header, header.getNext( ));
}
public void addLast(E e) {
    addBetween(e, trailer.getPrev( ), trailer);
 }
public E removeFirst( ) {
    if (isEmpty( )) return null;
    return remove(header.getNext( )); }
public E removeLast( ) {
    if (isEmpty( )) return null;
    return remove(trailer.getPrev( ));  }
```

Doubly linked List: concrete implementation

- Finally we add the 'addBetween' and the 'remove' methods

```
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( );}
```