CS1102: Data Structure and Algorithms

Tutorial 2 - Abstract Data Types (solution)

(4, 5 February 2010)

1. a) What is the difference between abstract classes and interfaces?
   Answer:

Abstract class	Interface
Describes similar functionality and data for all the classes that extend it	Describes similar functionality for all classes that implemented
Can have implemented methods and attributes	Only has method signatures (unimplemented); methods are only public
Can have non-constant data fields	All data is constant
Classes can only extend a single abstract class	Classes can implement multiple interfaces
Classes that extend an abstract class need not provide implementation for all abstract methods in their superclass. They will be in turn abstract classes.	Classes that implement an interface must provide implementation for *all* methods defined in the interface.

The decision of when to use abstract classes and when to use interfaces comes with experience. For example, if concepts have common behavior and attributes, then inheritance from an abstract class is useful. See the example of class `Circle` `extending` an abstract class `Shape` in Lecture 3, slide 24. Other classes that extend the abstract class `Shape` could be `Rectangle` and `Triangle`.

When only some functionality is common but the classes cannot be grouped under a single supertype (no other common behavior and data), then an interface is useful. For example, consider the case of concrete classes `Elephant` and `Airplane`. Both can be used to transport objects between `Location`, and as such implement the interface `Transport`. However, they have nothing else in common, and thus cannot inherit the same common abstract class.

```
public class Location{
      private String name;
      ...
      public Location(String name){
            this.name = name; //”this” removes ambiguities
      }

      public String getName(){return name;}
      ...
}

public interface Transport{
      public void move(Object o, Location a, Location b);
}
```

```
public class Elephant implements Transport{//must implement method move
	private String dateOfBirth;
	private String weight;
	private String preferredTrainerName;
	...

	public void move(Object o, Location a, Location b){
		System.out.println("Elephant moving " +o +" from "
		+a.getName()+" to " + b.getName());
	}
	...

}

public class Airplane implements Transport{//must implement method move
	private String maker;
	private int capacity;
	...

	public void move(Object o, Location a, Location b){
		System.out.println("Airplane moving " +o +" from "
		+a.getName()+" to " + b.getName());
	}
	...

}
```

Java talk: when A class inherits from superclass B, we say it A `extends` B. When a class C implements an interface D, we say C `implements` D.

b) What fundamental principle(s) does class StudentA implement? Comparing class StudentA with class StudentB, what are the benefits of said principle(s)?

```
public class StudentA{
      private String name;
      private String matric;
      private double CAP;

      public StudentA(String name, String matric){
            this.name = name;
            this.matric = matric;
      }

      public void setCAP(double val) throws Exception{
            if (val <= 0 || val >5) throw new Exception(“Illegal value!!”);
            CAP = val;
      }

      public double getCAP(){return CAP;}
      ...
}

public class StudentB{
      public  String name;
      public String matric;
      public double CAP;

      public StudentB(String name, String matric){
            this.name = name;
            this.matric = matric;
      }
}

public class Test{
      public static void main(String[] args){
            StudentA s1= new StudentA("John Lee", "HT0851227A");
            try{
                  s1.setCAP(-10);
            }
            catch (Exception exp){
                  exp.printStackTrace();
            }

            StudentB s2 = new StudentB("Jane Smith", "HT051235B");
            s2.CAP = 345;
      }
}
```

Answer:
The principles implemented by class StudentA are *information hiding* and *encapsulation*. ***Information hiding*** makes implementation details, including components of an object, inaccessible. ***Encapsulation*** is the grouping of data and the operations that apply to them to form an aggregate while hiding the implementation of the aggregate. Encapsulation and information hiding are achieved in Java through the use of the class and its public and private sections.

The attribute `CAP` of class `StudentA` is private and thus cannot be set by other objects. Contrary to attribute `CAP` in class `StudentB`, the only way to access attribute `CAP` in object

of type `StudentA` is through the use of `accessor` and `mutator` methods, namely `getCAP` and `setCAP`.

The code is also an example of ***defensive programming***. Here, the writer of class `StudentA` makes sure that no illegal values (negative or greater than 5) can be set to `CAP`. The writer of class `StudentB` has never thought that somebody (i.e. the writer of class `Test`) could attack his code by entering illegal values of `CAP` (i.e. 345). Always assume that the users of your code will make mistakes, sometimes unintentional.

c) Below is the pseudo-code for bubble sort, one of the most straightforward algorithms for sorting an integer array in an ascending order. The algorithm goes through the vector and swaps any adjacent values (`swap(items[i], items[i+1])`) if they are in the incorrect order. Write the pre and post conditions for the pseudo-code. Try to find a loop-invariant. What arrays would you use to test the method once you have implemented it?

```
method void bubbleSort(items[])
//pre-condition:
do
    swapped := false
    for each i in 0 to length(items) - 2 inclusive do: //line 1
      if items[i] > items[i+1] then
        swap(items[i], items[i+1])              //line 2
        swapped := true
      end if
    end for
  while swapped
//post-condition:
end method
```

Answer:
Bubble sort is one of the simplest algorithms for sorting arrays. The algorithm goes through the array until all elements have been sorted. For any adjacent pair of elements, swap them if the one on the left (items[i]) is greater than the one on the right (items[i+1]).

The pre and post conditions are given below. One pre-condition could be that the array has only integer values and that it is non-empty. The straightforward post-condition is that the array is sorted.

```
method void bubbleSort(items[])
//pre-condition: array is of integer items and is non-empty
do
    swapped := false
    for each i in 0 to length(items)- 2 inclusive do: //line 1
      if items[i] > items[i+1] then
        swap(items[i], items[i+1])              //line 2
        swapped := true
      end if
    end for
  while swapped
//post-condition: array is sorted
end method
```

For most algorithms, computing loop invariants is not easy. For this bubble sort algorithm, apply the pseudo-code for the array: items[]: 7 2 8 5 1
2 7 8 5 1 (swap items[0] with items[1] according to line 2)
2 7 8 5 1 no swap
2 7 5 8 1 (swap items[2] with items[3] according to line 2)
2 7 5 1 8 (swap items[3] with items[4] according to line 2)
As it can be seen, at the end of the first pass of the do loop, the largest number is in its place at the end of the array. Similarly, we will see that with each i-th pass of the `do` loop, the largest i numbers will be in place at the end of the array. This is the loop invariant for the bubble sort algorithm.

Determining the values to use for testing requires a lot of experience and a “nose” for determining bugs. When you first test a method, try first with border cases such as negative or zero values or with already sorted arrays. For example, for bubble sort you could try with cases such as: empty array, an already sorted array, an array sorted in descending order, and an array with all values equal.

d) Use the above pseudo-code to quickly modify the code below by implementing the method sort for the vector of `StudentA`, which sorts the students in the array in an ascending order based on their `CAP`.

```
public class StudentA{
      private String name;
      private String matric;
      private double CAP;

      public StudentA(String name, String matric){
            this.name = name;
            this.matric = matric;
      }

      public void setCAP(double val) throws Exception{
            if (val <= 0 || val >5) throw new Exception("Illegal value!!");
            CAP = val;
      }

      public double getCAP(){return CAP;}

      public String toString(){
            return name +"\t(" + matric+"t:\t" + CAP;
      }
      …
}

public class Test{
      public void sort(StudentA[] students){
      //implement here!! How fast can you write this?
     }//end sort


      public static void main(String[] args){
            StudentA[] students = new StudentA[3];
            StudentA s1= new StudentA("John Lee", "HT0851227A");
            StudentA s2 = new StudentA("Jane Doe", "HT0556223Y");
            StudentA s3 = new StudentA("Yixiang Chen", "HT07542324L");
            try{
                  s1.setCAP(3.56);
                  s2.setCAP(4.30);
                  s3.setCAP(4.75);
            }
            catch (Exception exp){
                  exp.printStackTrace();
            }
            students[0] = s1; students[1] = s2; students[2] = s3;

            Test t = new Test();
            t.sort(students);
            for (StudentA student : students){
                  System.out.println(student);
            }
      }
}
```

Answer:

Converting the pseudo-code to cater for `StudentA` objects is straightforward. Instead of using `item[i],` we now use `students[i].getCAP()`. The swap method will use `setCAP().` An implementation is:

```
public void sort(StudentA[] students){
      boolean swapped = false;
      do
      {
           swapped = false;
           for (int i = 0; i < students.length -1; i++){
                 if (students[i].getCAP() > students[i+1].getCAP()){
                       StudentA temp = students[i];
                       students[i] = students[i+1];
                       students[i+1] = temp;
                       swapped = true;
         }//end if
           }//end for
      }while (swapped); //end do
     }//end sort
```

In general, it is better to first write a higher level pseudo-code BEFORE writing the code.

Quick Java tip: Notice how we have used method `toString`() in class `StudentA` to return relevant information about the `StudentA` object. All objects in Java have a `toString`() method. If you implement it to return relevant information as shown in class `StudentA`, it can then be used during `System.out.println` calls to print out the desired information instead of the default one. (Try removing the `toString`() method to see what is printed instead.)

2. Suppose that you are required to code a ContactList ADT using a Java Array as the data structure to store the contacts. The ADT must provide public methods for the programmer to add, edit, and delete a contact. A contact contains a person name and a phone number. Duplicate contacts (same person name and same phone number) are not allowed. You can safely assume that the person name is unique. Explain the following in English (with the aid of diagrams if necessary).

   a) What assumptions would you make? (Hint: a person has more than one phone numbers.)

      Answer:
      There are many possible assumptions, including but not limited to the followings:
      i. A person possibly has more than one phone numbers, so multiple contacts can be allowed for one person, with each contact having same person name and different phone number.
      ii. In the ContactList, contacts can be sorted by person names.

   b) How would you implement the add method based on your assumptions? Assume that the new contact information (person name and phone number) is passed to the method.

      Answer:
      Since duplicate contacts are not allowed, first check whether there is any existing contact that is the same as the new one, i.e. same person name and same phone number. If duplicates are found, throw an exception and new contact is not added.

      If contacts are sorted by person names, search through the array to find the position where new contact should be inserted, shift all the contacts starting from the position till the end of the list to vacate the position and add the new contact to the position.

   c) What would you do when the array is full?

      Answer:
      Create a new array larger than the existing one and copy the contents of the existing array into the new array. For example: create a new array twice the size of the existing one.

   d) How would you implement the edit method based on your assumptions? Assume that both old and new contact information (person name and phone number) is passed to the method.

      Answer:
      Search through the array to find the contact which matches the person name and phone number specified as old contact, and edit it according to the given new contact.

      If person name is changed and contacts are sorted by person names, re-sort should be performed.

e) How would you implement the delete method based on your assumptions? Assume that the contact information (person name and phone number) is passed to the method.

Answer:

Search through the array to find the contact which matches the person name and phone number, set that position to empty and shift all the contacts starting from the next position till the end the list to fill the empty slot.

3. Consider the Java class `ArrayList<E>`. Suppose we define a data type called

   `ArrayList<ArrayList<Integer>>`

   which is an ArrayList of ArrayLists of Integers.
   Implement a method

   `int getMin( ArrayList<ArrayList<Integer>> aList )`

   that returns the minimum of all the integers in the `aList`.

Answer:

This question is about using generic Java ADT classes and its methods.

```
int getMin( ArrayList<ArrayList<Integer>> aList )
// ------------------------------------------------------
// Finds the minimum of the integers in the list of lists: aList.
// Precondition:  aList is a list of lists of integers.
// Postcondition:  The minimum of the integers in aList is returned.
// ------------------------------------------------------
{
      int min = Integer.MAX_VALUE; //maximum integer value

      //Iterates through all the available ArrayLists
      for(ArrayList<Integer> row: aList) //enhanced for loop
      {
            //Iterate through all the available integers
            for(int val : row) //unbox
            {
                  if (val < min) min = val;
            }
      }

      return min;
}
```

4. Consider the following ADT of a simple linked list.

```
class ListNode<E> {
    private E element;
    private ListNode<E> next;

    public ListNode (E item, ListNode <E> n) {
        element = item;
        next = n;
    }

    public E getElement() {
        return this.element;
    }

    public ListNode<E> getNext() {
        return this.next;
    }

    public void setNext(ListNode<E> n) {
        this.next = n;
    }
}

class SimpleLinkedList<E> {
    private ListNode<E> head = null;

    public ListNode<E> getHead() {
        return this.head;
    }

    public void addHead(E item) {
        head = new ListNode<E>(item, head);
    }

    public void insertAfter(E item, E newItem) {
        for(ListNode<E> curr = head;
            curr != null;
            curr = curr.getNext())
        {
            if (curr.getElement().equals(item)) {
                ListNode<E> newNode =
                    new ListNode<E>(newItem, curr.getNext());
                curr.setNext(newNode);
                System.out.println(newItem + " is inserted after " + item);
                return;
            }
        }
        System.out.println("Cannot find " + item);
    }

    public void delete(E item) {
        for(ListNode<E> prev = null, curr = head;
            curr != null;
            prev = curr, curr = curr.getNext())
        {
            if (curr.getElement().equals(item)) {
                if (prev == null) {
                    head = curr.getNext();
```

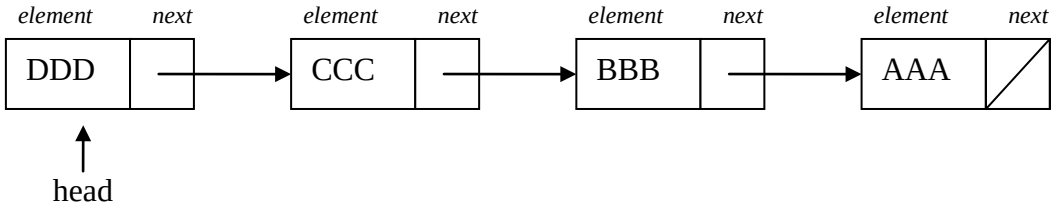
```
                }
                else {
                    prev.setNext(curr.getNext());
                }
                System.out.println(item + " is deleted");
                return;
            }
        }
        System.out.println("Cannot find " + item);
    }

    public void print() {
        for(ListNode<E> curr = head;
            curr != null;
            curr = curr.getNext())
        {
            System.out.print(curr.getElement() + " -- ");
        }
        System.out.println();
    }
}
```

Using the above ADT, we create a simple linked list of Strings:

```
SimpleLinkedList<String> list = new SimpleLinkedList<String>();
list.addHead("AAA");
list.addHead("BBB");
list.addHead("CCC");
list.addHead("DDD");
list.print();
```

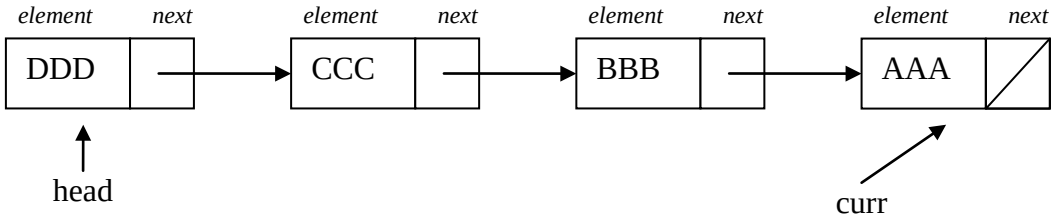
The output of list.print() is DDD -- CCC -- BBB -- AAA --
The following diagram shows the structure of our created list.

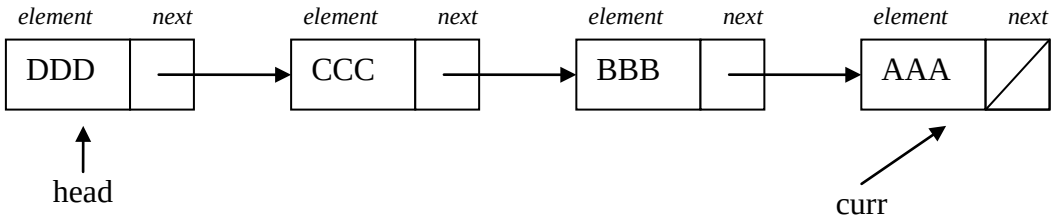


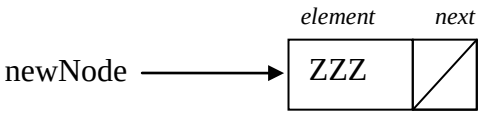
Draw diagrams for the following actions step by step and write the output.

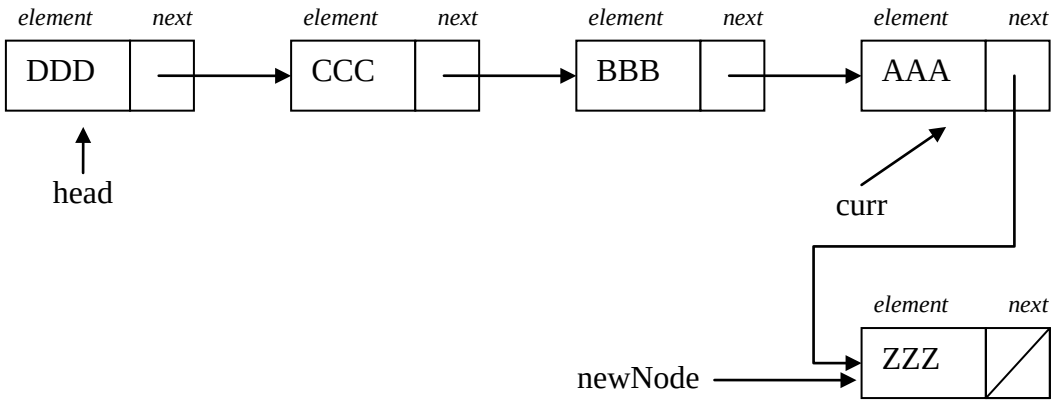
a) `list.insertAfter("AAA", "ZZZ"); list.print();`

Answer:





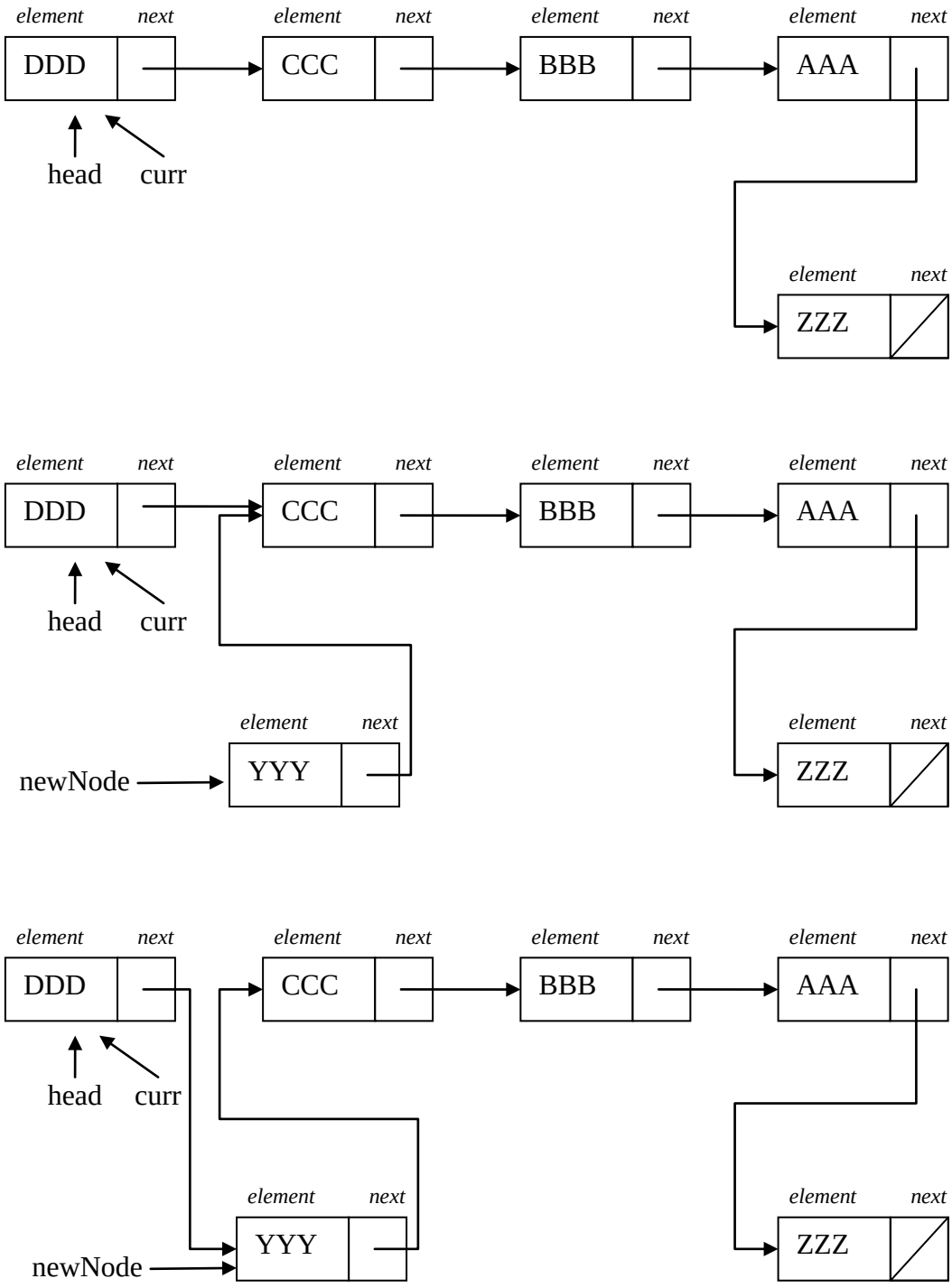




The output is:
ZZZ is inserted after AAA
DDD -- CCC -- BBB -- AAA -- ZZZ –

```
b) list.insertAfter("DDD", "YYY"); list.print();
```

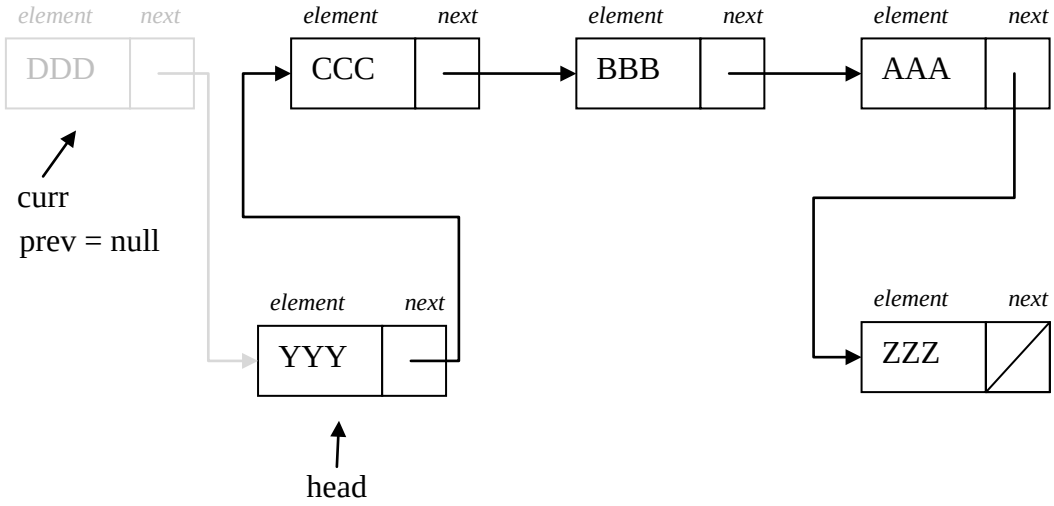
Answer:



The output is:
YYY is inserted after DDD
DDD -- YYY -- CCC -- BBB -- AAA -- ZZZ –

```
c) list.delete("DDD"); list.print();
```

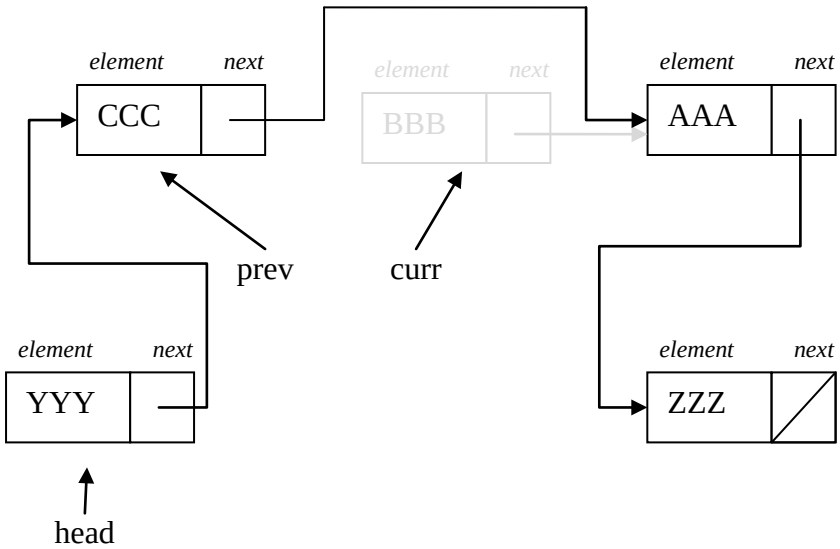
Answer:



The output is:
DDD is deleted
YYY -- CCC -- BBB -- AAA -- ZZZ --

```
d) list.delete("BBB"); list.print();
```

Answer:



The output is:
BBB is deleted
YYY -- CCC -- AAA -- ZZZ --