LINKED LISTS (CONTD)
RULE OF THREE
OPERATOR OVERLOADING

Problem Solving with Computers-II

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

Memory Errors

- Memory Leak: Program does not free memory allocated on the heap.

- Segmentation Fault: Code tries to access an invalid memory location

RULE OF THREE

If a class overload one (or more) of the following methods, it should overload all three methods:

1. Destructor
2. Copy constructor
3. Copy assignment

The questions we ask are:

1. What is the behavior of these defaults?
2. What is the desired behavior ?
3. How should we over-ride these methods?

```
void test_append_0(){
      LinkedList ll;
      ll.append(10);
      ll.print();
}
```

Assume:

* Default destructor

* Default copy constructor

* Default copy assignment

What is the result of running the above code?

A. Compiler error

B. Memory leak

C. Segmentation fault

D. None of the above

Why do we need to write a destructor for LinkedList?

A. To free LinkedList objects

B. To free Nodes in a LinkedList

C. Both A and B

D. None of the above

Behavior of default copy constructor

```
void test_copy_constructor(){
    LinkedList l1;
    l1.append(1);
    l1.append(2);
    LinkedList l2{l1};
    // calls the copy c’tor
    l1.print();
    l2.print();
}
```

Assume:

destructor: overloaded

copy constructor: default

What is the output?

A. Compiler error

B. Memory leak

C. Segmentation fault

D. All of the above

E. None of the above

Behavior of default copy assignment

l1 : 1 -> 2- > 5 -> null

```
void default_assignment_1(LinkedList& l1){
   LinkedList l2;
   l2 = l1;
}
```

`* What is the behavior of the default assignment operator?`

Assume:

* **Overloaded destructor**

* **Default copy constructor**

* **Default copy assignment**

Behavior of default copy assignment

```
void test_default_assignment_2(){
    LinkedList l1, l2;
    l1.append(1);
    l1.append(2)
    l2 = l1;
    l2.print()
}
```

What is the result of running the above code?
A. Prints 1 , 2
B. Segmentation fault
C. Memory leak
D. A &B
E. A, B and C

Assume:

*** Overloaded destructor**

*** Default copy constructor**

*** Default copy assignment**

Behavior of default copy assignment

```
void test_default_assignment_3(){
    LinkedList l1;
    l1.append(1);
    l1.append(2)
    LinkedList l2{l1};
    l2.append(10);
    l2.append(20);
    l2 = l1;
    l2.print()
}
```

What is the result of running the above code?

A. Prints 1 , 2

B. Segmentation fault

C. Memory leak

D. A &B

E. A, B and C

Assume:

*** Overloaded destructor**

*** Overloaded copy constructor**

*** Default copy assignment**

Overloading Binary Comparison Operators

We would like to be able to compare two objects of the class using the following operators

==

!=

and possibly others

```
void isEqual(const LinkedList & lst1, const LinkedList &lst2){
    if(lst1 == lst2)
          cout<<“Lists are equal”<<endl;
    else
         cout<<“Lists are not equal”<<endl;

}
```

Overloading Binary Arithmetic Operators

We would like to be able to add two points as follows

```
LinkedList l1, l2;

//append nodes to l1 and l2;

LinkedList l3 = l1 + l2 ;
```

Overloading input/output stream

Wouldn’t it be convenient if we could do this:

```
LinkedList list;
cout<<list; //prints all the elements of list
```

Overloading Binary Comparison Operators

We would like to be able to compare two objects of the class using the following operators

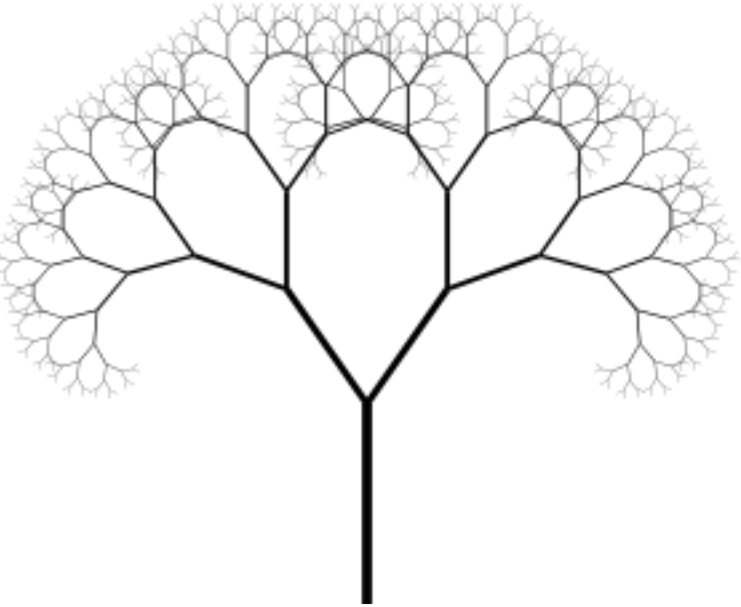
==

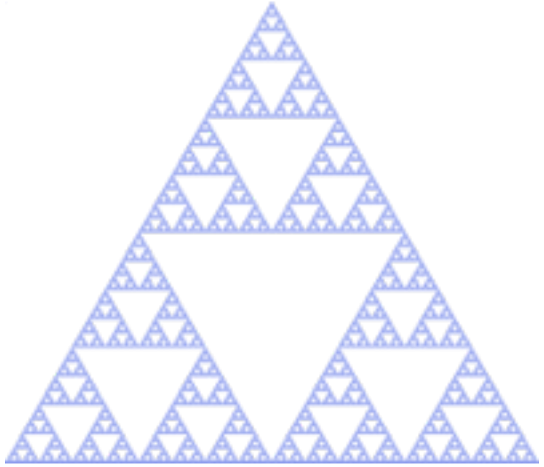
!=

and possibly others

Last class: overloaded == for LinkedList

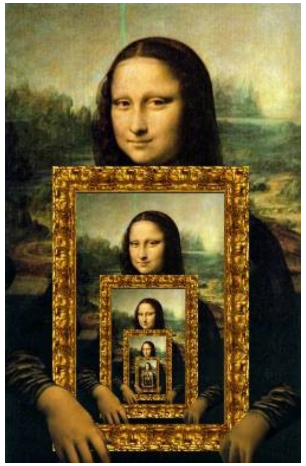
Recursion





Sierpinski triangle

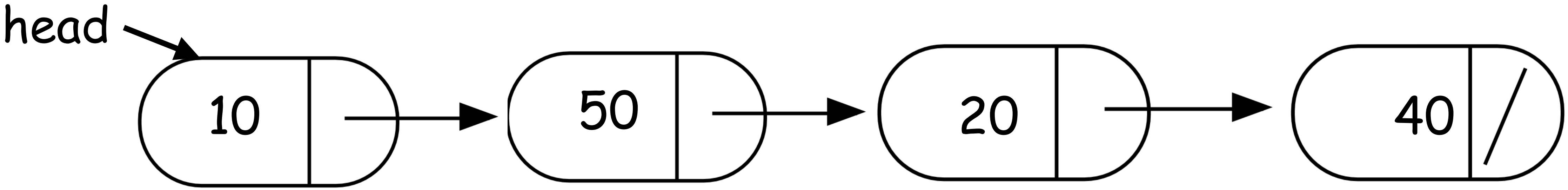
Zooming into a Koch's snowflake



Describe a linked-list recursively

Which of the following methods of LinkedList CANNOT be implemented using recursion?

A. Find the sum of all the values
B. Print all the values
C. Search for a value
D. Delete all the nodes in a linked list
E. All the above can be implemented using recursion



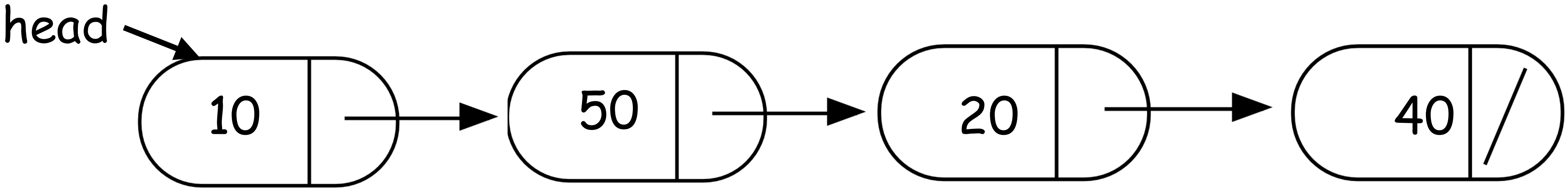
```
int IntList::sum(){

     //Return the sum of all elements in a linked list
  }
```

Helper functions

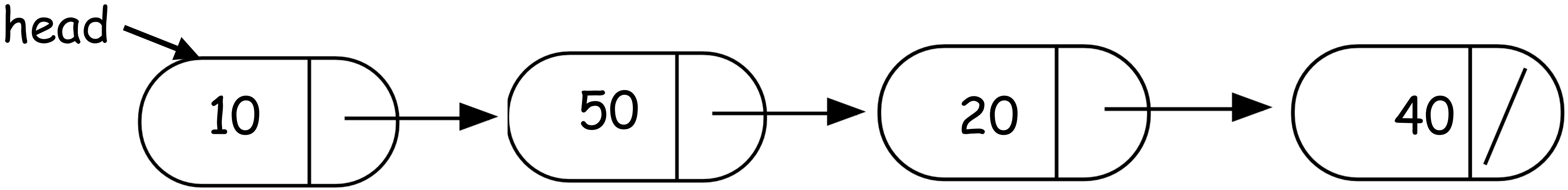
- Sometimes your functions takes an input that is not easy to recurse on
- In that case define a new function with appropriate parameters: This is your helper function
- Call the helper function to perform the recursion
- Usually the helper function is private

For example

```
Int IntList::sum(){

   return sum(head);
    //helper function that performs the recursion.

}
```



```
int IntList::sum(Node* p){









 }
```



```
bool IntList::clear(Node* p){





  }
```

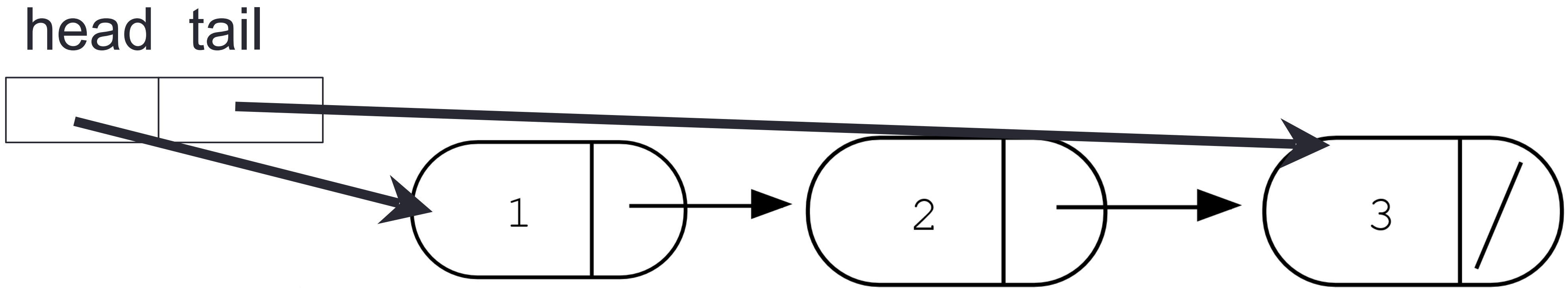
Concept Question

```
LinkedList::~LinkedList(){
   delete head;
}
```

```
class Node {
    public:
       int info;
       Node *next;
};
```

Which of the following objects are deleted when the destructor of Linked-list is called?

(A)



(B): only the first node

(C): A and B

(D): All the nodes of the linked list

(E): A and D

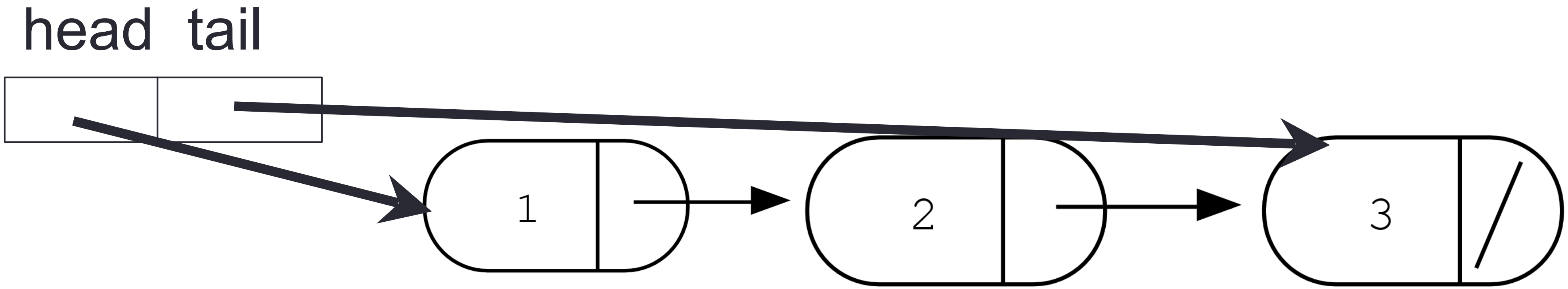
Concept question

```
LinkedList::~LinkedList(){
   delete head;
}
```

```
Node::~Node(){
   delete next;
}
```

Which of the following objects are deleted when the destructor of Linked-list is called?

(A)



(B): All the nodes in the linked-list

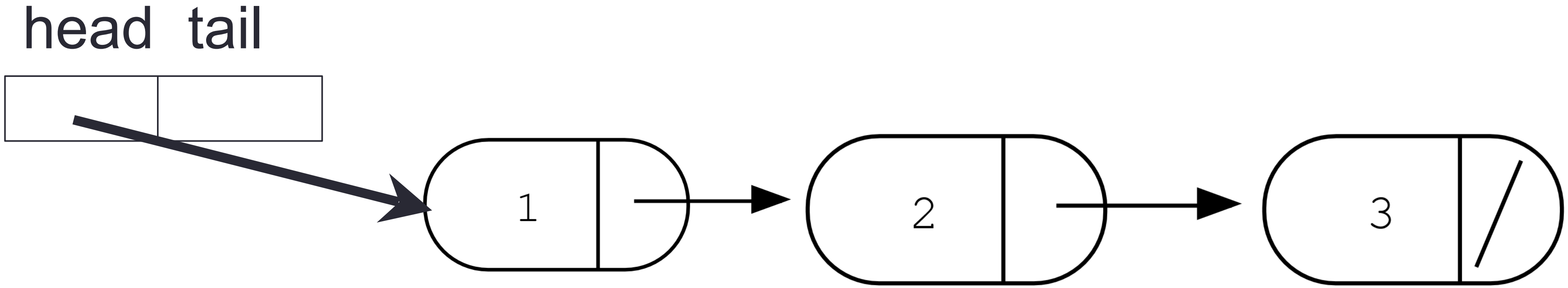
(C): A and B

(D): Program crashes with a segmentation fault

(E): None of the above

```
LinkedList::~LinkedList(){
   delete head;
}
```

```
Node::~Node(){
   delete next;
}
```



CAUTION
THIS SIGN HAS
SHARP EDGES
DO NOT TOUCH THE EDGES OF THIS SIGN
ALSO, THE BRIDGE IS OUT AHEAD

Next time

- Binary Search Trees

Next time

- Recursion + PA01