RULE OF THREE
LINKED LISTS CONTD

Problem Solving with Computers-II

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

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

Read the syllabus. Know what's required. Know how to get help.

CLICKERS OUT – FREQUENCY AB

Questions you must ask about any data structure:

- **What operations does the data structure support?**

  A linked list supports the following operations:

  1. Insert (a value)
  2. Delete (a value)
  3. Search (for a value)
  4. Min
  5. Max
  6. Print all values
- **How do you implement the data structure?**
- **How fast is each operation?**

Linked-list as an Abstract Data Type (ADT)

```
class IntList {
public:
    IntList();                  // constructor
    ~IntList();                 // destructor
    // other methods
private:
    // definition of Node structure
    struct Node {
        int info;
        Node *next;
    };
    Node *head; // pointer to first node
};
```

Code related to linked list ADT:

`https://ucsb-cs24-s18.github.io/lectures/lect07/`

Memory Leaks

- Data created on the heap with **new** must be deleted using the keyword **delete**
- **Code has a memory leak if**
  - Data on the heap is never deleted or
  - Pointer to the data is lost

- Use valgrind to detect leaks

```
./valgrind –leak-check = full <name of executable>
```

- Code that results in a leak

```
void foo(){
    int*p = new int;
}
```

RULE OF THREE

If a class defines one (or more) of the following it should probably explicitly define all three:

1. Copy constructor
2. Copy assignment
3. De-constructor

1. What is the behavior of default copy-constructor, copy-assignment and deconstructor (taking linked lists as example)?
2. When and why do we need to overload these methods?
3. What is the desired behavior of the overloaded methods for linked-lists?

De-constructor: Default behavior

```
void foo(){
   IntList ll;
   ll.insert(100);
   ll.insert(50);
    ll.insert(75);

}
```

```
class IntList{
public:
     IntList(){head = tail = nullptr;}
     void insert(int value);
private:
     //Definition of struct Node
     //not shown here
     Node* head;
     Node* tail;
};
```

Does the above code result in a memory leak?

A. Yes

B. No

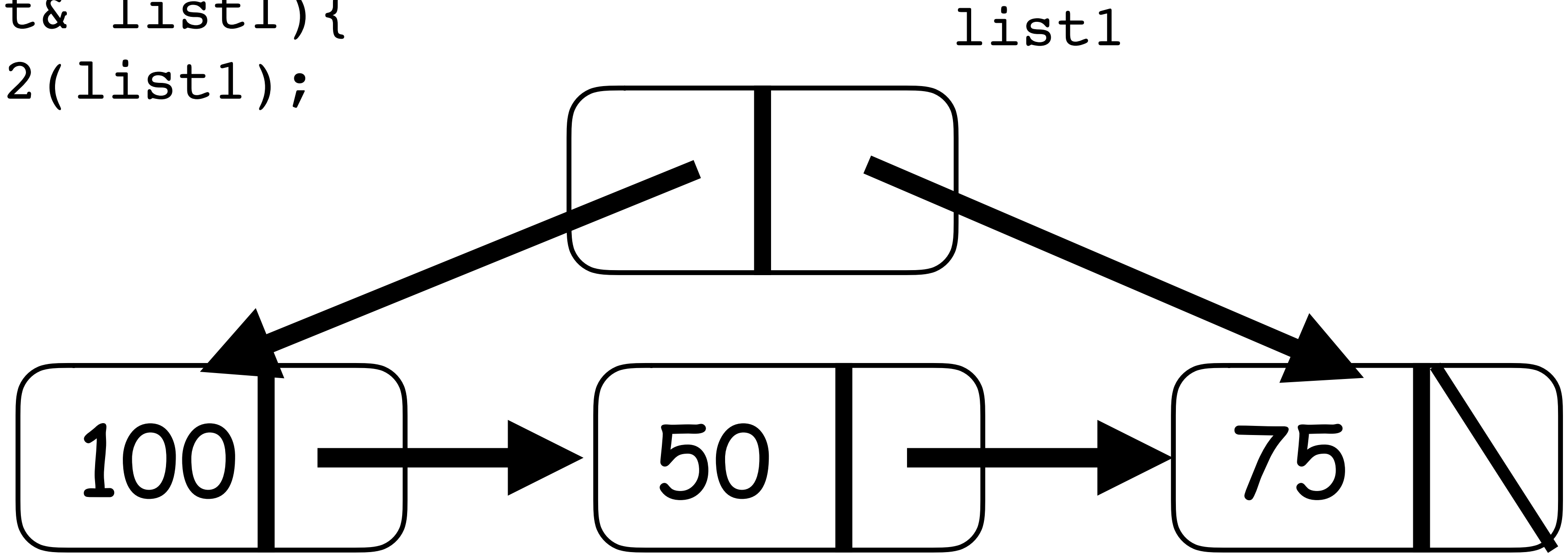
De-constructor: Default behavior

```
void foo(){
    IntList ll;
    ll.insert(100);
    ll.insert(50);
     ll.insert(75);
}
```

Copy constructor: Default behavior

```
void foo(Intlist& list1){
   IntList list2(list1);

}
```



Copy assignment

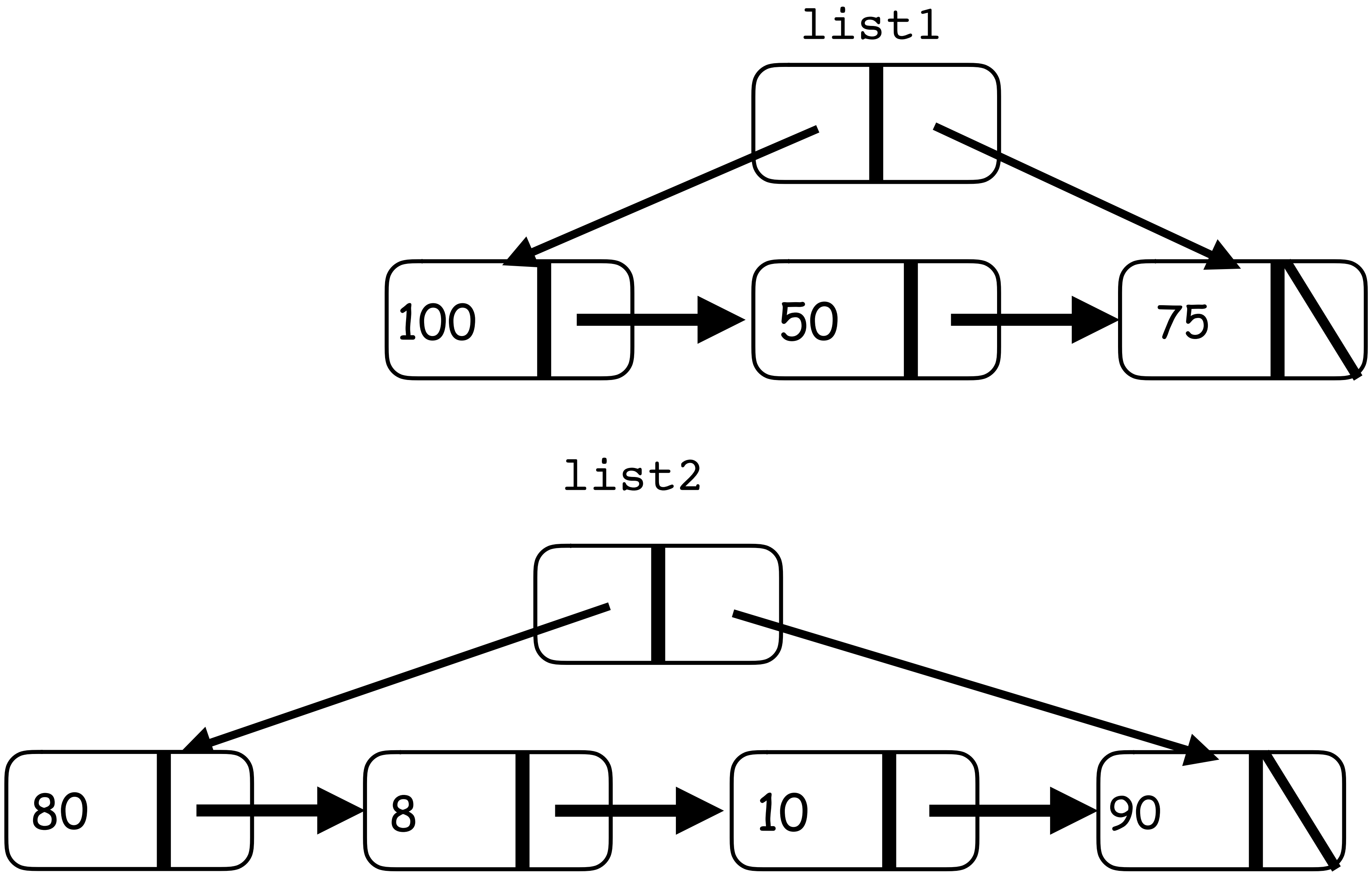
```
IntList list1, list2;  //default constructors called

list1 = list2; //Copy assignment is called
```

- The copy assignment should result in list1 having a copy of the data of list2
- A class always has a default copy assignment which may be overloaded
- Why overload the copy assignment?

Copy assignment: Default behavior

```
list2 = list1;
```



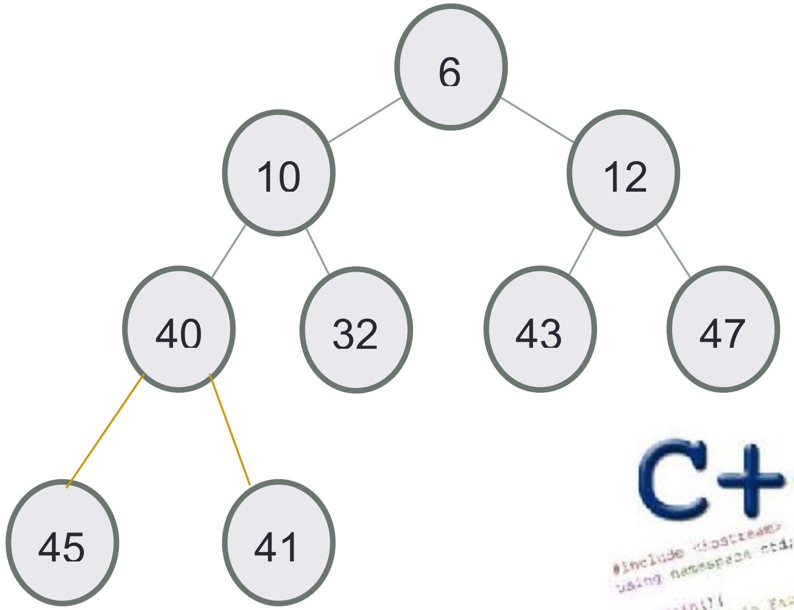
Value semantics: Copy assignment and copy constructor

Value semantics means passing objects to functions by value. The methods invoked are:

- Copy assignment
- Copy constructor

RECURSION

Problem Solving with Computers-I



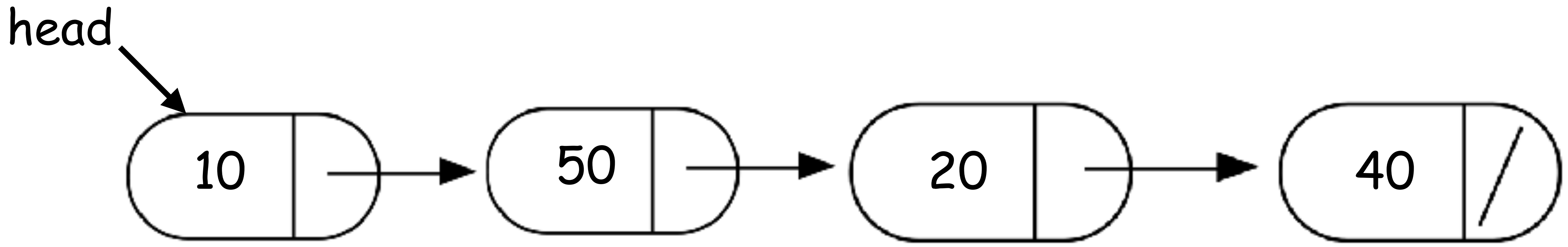
```
#include <iostream>
using namespace std;

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

Let recursion draw you in….

- Many problems in Computer Science have a recursive structure…
- Identify the “recursive structure” in these pictures by describing them

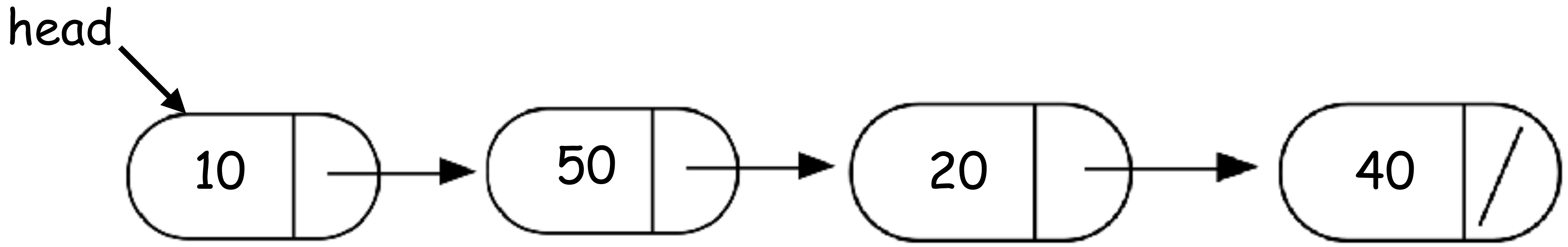
Recursive description of a linked list



- A non-recursive description of the linked list:
  A linked list is a chain of nodes

- A recursive description of a linked-list:
  A linked list is a node, followed by a smaller linked list

Sum all the elements in a linked list



- A recursive description of a linked-list:
  A linked list is a node, followed by a smaller linked list

  Sum of all the elements in a linked list is:

  Value of the first node +

  Sum of the all the elements in the *rest* of the 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

For example

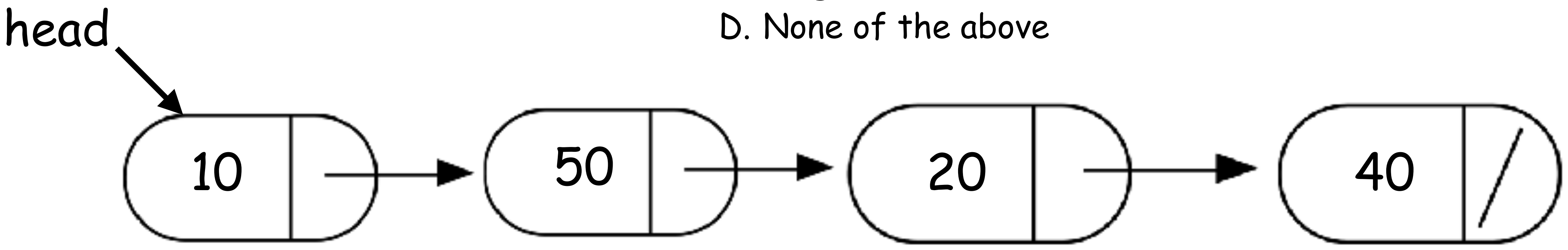
```
int IntList::sum(){
   return sumHelper(head); //sumHelper is the helper
   //function that performs the recursion.

}
```

Let’s code it up

What happens when we execute this code on the example linked list?

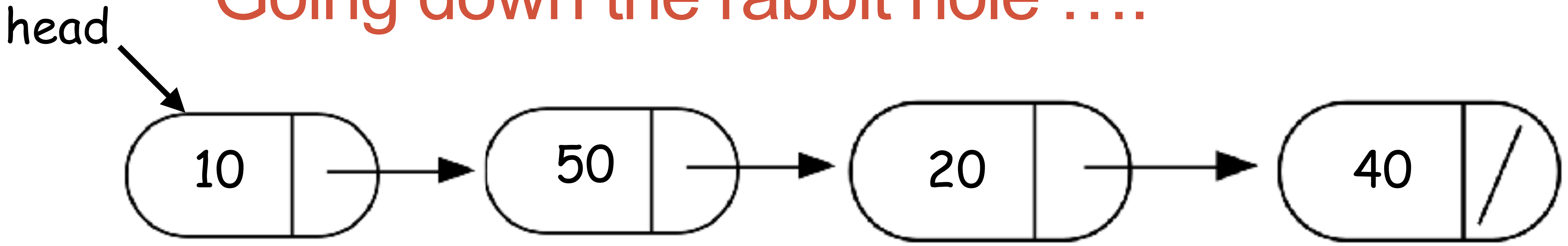
A. Returns the correct sum (120)
B. Program crashes with a segmentation fault
C. Program runs forever
D. None of the above



```
int IntList::sumHelper(Node* h){

    double result = h->value + sum(h->next);
    return result;
}
```

Going down the rabbit hole ….

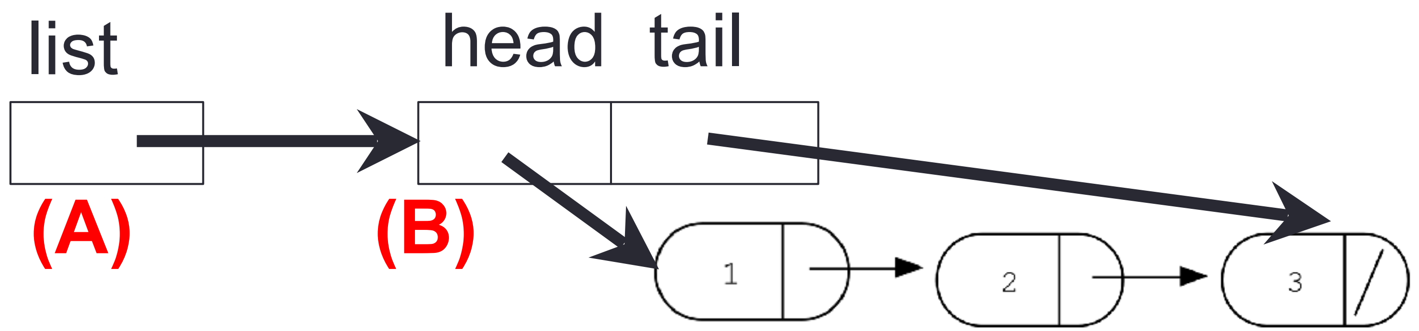


```
int IntList::sumHelper(Node* h){
      // Solve the smallest version of the problem
      // THE BASE CASE!!
    if(!h) return 0;
      // Go deeper into the rabbit hole!!
      // THE RECURSIVE CASE:
    double result = h->value + sumHelper(h->next);
      // Come out of the rabbit hole
    return result;
  }
```

Deleting the list

```
int deleteList(LinkedList * list){
   delete list;
}
```

Which data objects are deleted when the above function is called on the linked list shown below:



(C) All nodes of the linked list

(D) B and C

(E) All of the above

Does this result in a memory leak?

Next time

- Run time analysis