RULE OF THREE
LINKED LISTS CONTD

Problem Solving with Computers-II

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

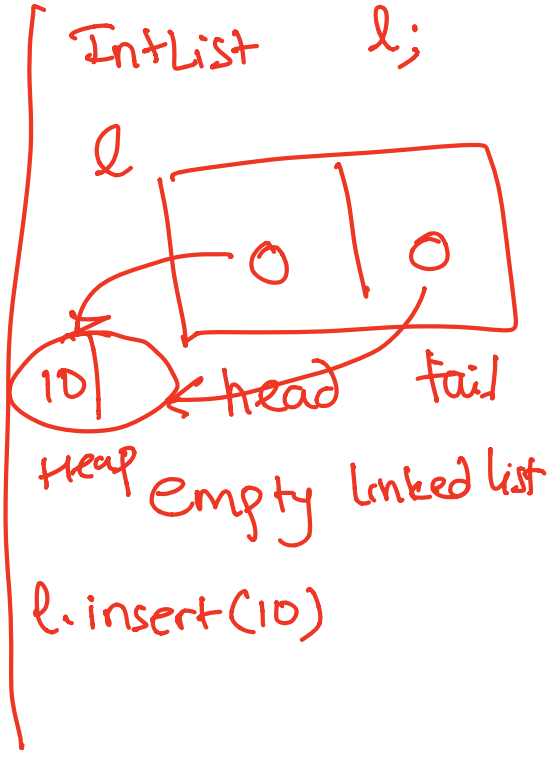
CLICKERS OUT – FREQUENCY AB

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
};   Node + tail
```



Code related to linked list ADT:

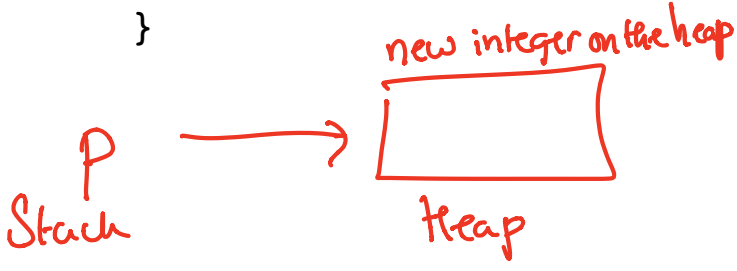
https://ucsb-cs24-s18.github.io/lectures/lect07/

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

- Code that results in a leak

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




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

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;
};
```

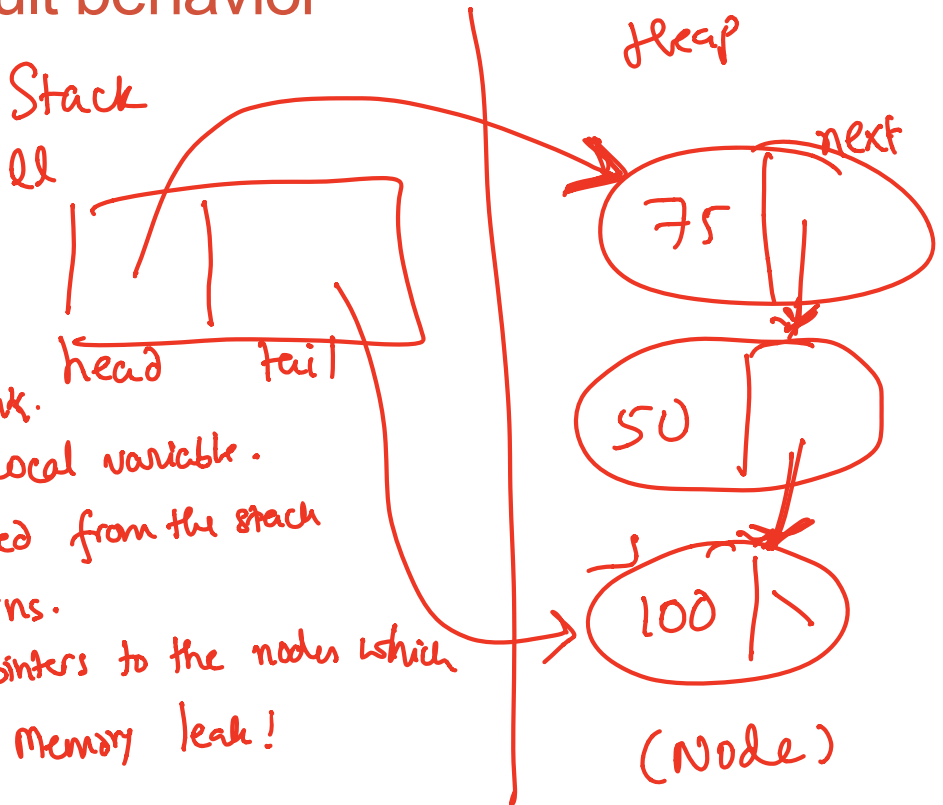
Does the above code result in a memory leak?

A. Yes (See next slide for why)

B. No

De-constructor: Default behavior

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



This function has a memory leak.
The reason is that ll is a local variable.
ll is automatically removed from the stack when the function returns.
This results in losing the pointers to the nodes which are on the heap causing a memory leak!

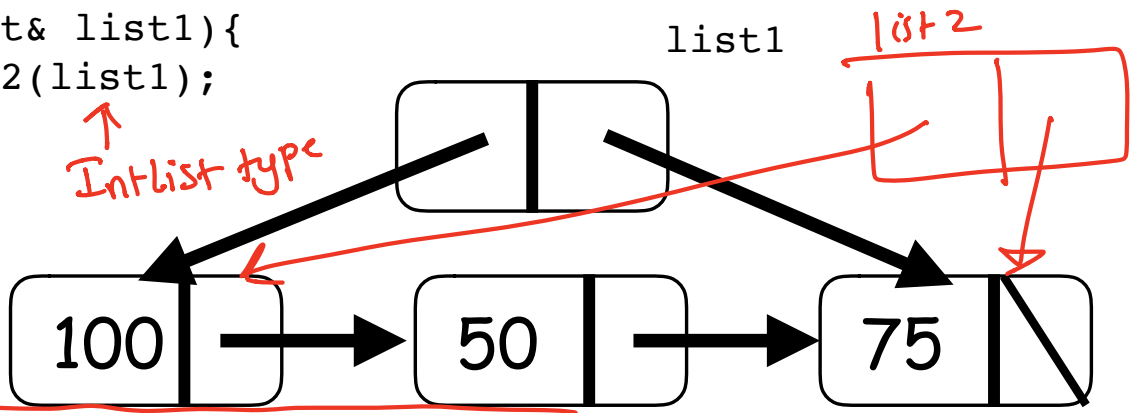
Copy constructor: Default behavior

OUTCOME OF DEFAULT COPY CONSTRUCTOR

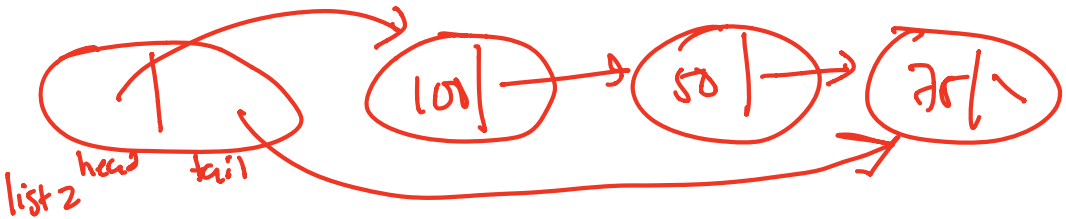
```
void foo(Intlist& list1){
   IntList list2(list1);

}
```

Intlist type



DESIRED OUTCOME : DEEP COPY OF LIST1

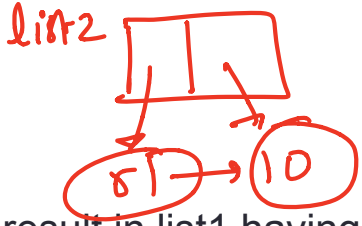


The default copy constructor only copies list 1's head & tail pointers to list 2's pointers (shallow copy)

Copy assignment

```
IntList list1, list2;  //default constructors called
// Some code that adds nodes to list1 & list 2
list1 = list2; //Copy assignment is called
```

Copies the head and tail of list2 into list1



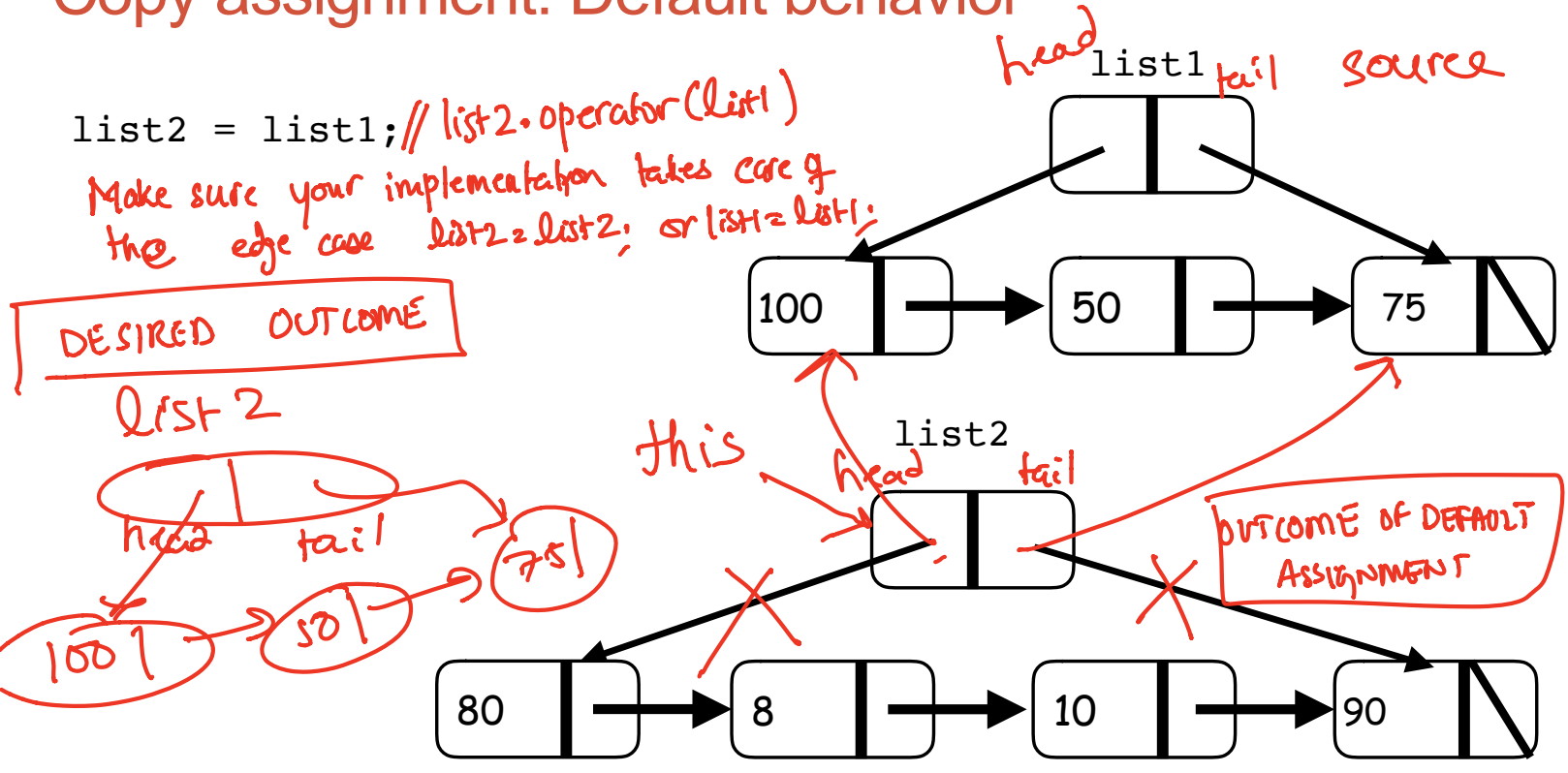
- The copy assignment should result in list1 having a copy of the data of list2
- A class always has a default copy assignment which may be overloaded
- Why overload the copy assignment? The default version just copies the member variables of one object into the other. Not what we want in the case of a linked list.

Copy assignment: Default behavior

```
list2 = list1;
```

// list2.operator(list1)

Make sure your implementation takes care of the edge case list2 = list2; or list1 = 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

Next time

- ~~Run time analysis~~ Recursion