BIG FOUR AND THE RULE OF THREE
LINKED LISTS

Problem Solving with Computers-II

GitHub

The Big Four (review)

1. Constructor
2. Destructor
3. Copy Constructor
4. Copy Assignment

Constructor (review)

```
void foo(){
     Complex p;
     Complex* q = new Complex;
     Complex w{10, 5};
}
```

How many times is the constructor called in the above code?

A. Never

B. Once

C. Two times

D. Three times

Destructor (review)

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

The destructor of which of the objects is called after foo() returns?

A. p

B. q

C. *q

D. None of the above

Copy constructor (review)

- In which of the following cases is the copy constructor called?

A. `Complex p1; Complex p2{1, 2};`
B. `Complex p1{1, 2}; Complex p2{p1};`
C. `Complex *p1 = new Complex{2, 3};`
   `Complex p2 = *p1;`
D. B&C
E. A, B & C

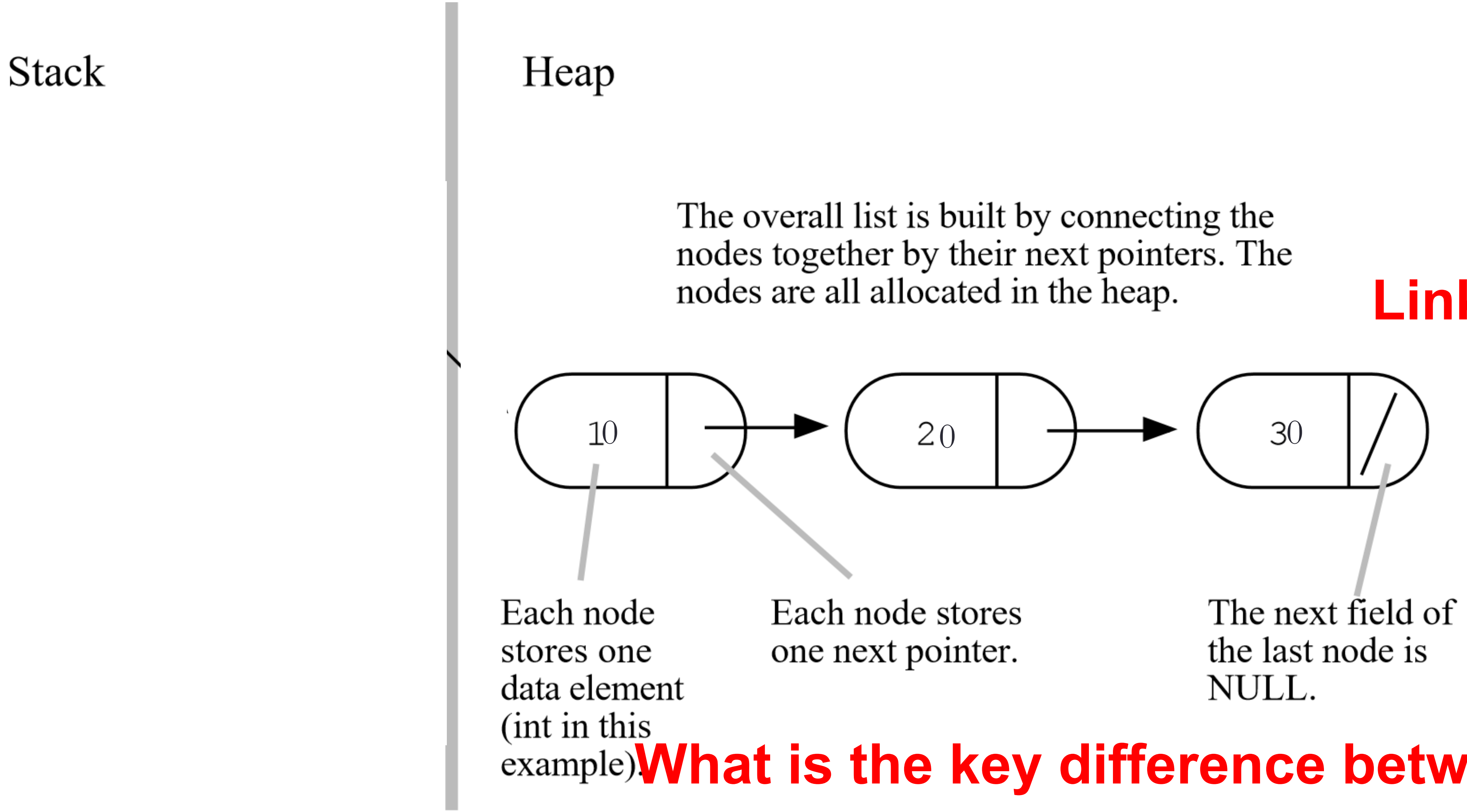
```
double foo(Complex p){
    return p.conjugate(10);
}
int main(){
     Complex q{1, 2};
     foo(q);
}
```

Which of the following special methods is called when passing parameters to foo()?

A. Parameterized constructor
B. Copy constructor
C. Copy assignment
D. Destructor

Linked Lists

Array List



Linked List

What is the key difference between the two?

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 to the head)
  2. Append (a value to the tail)
  3. Delete (a value)
  4. Search (for a value)
  5. Min
  6. Max
  7. Print all values
- **How do you implement each operation?**
- **How fast is each operation?**

Linked-list as an Abstract Data Type (ADT)

```
class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    // other public methods


private:
    struct Node {
        int info;
        Node* next;
    };
    Node* head;
    Node* tail;
};
```

RULE OF THREE

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

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. Prints 10

D. None of the above

Behavior of default copy constructor

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

```
void test_default_copy_constructor(LinkedList& l1){
    // Use the copy constructor to create a copy of l1







}
```

* What is the default behavior?
* Is the default behavior correct ?
* How do we change it?

Assume:

* **Overloaded destructor**
* **Default copy constructor**
* **Default copy assignment**

Behavior of default copy assignment

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

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

```
* What is the default behavior?
```

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.append(10);
   l2.append(20);
   l2 = l1;
   l2.print()
}
```

What is the result of running the above code?
A. Segmentation fault
B. Prints 1 , 2
C. Both A and B
D. None of the above

Assume:

- **Overloaded destructor**
- **Default copy constructor**
- **Default copy assignment**

Behavior of default copy assignment

```
void test_default_assignment_2(){
    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. Segmentation fault
B. Memory leak
C. Both A and B
D. None of the above

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;

}
```

Next time

- Linked Lists contd.
- GDB