REVIEW POINTERS, DYNAMIC MEMORY
LINKED LISTS
RULE OF THREE

Problem Solving with Computers-II

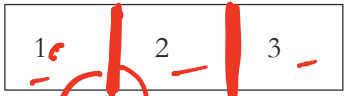
Frequency AC

Have you implemented a linked-list before?

A. Yes
B. No

Linked Lists

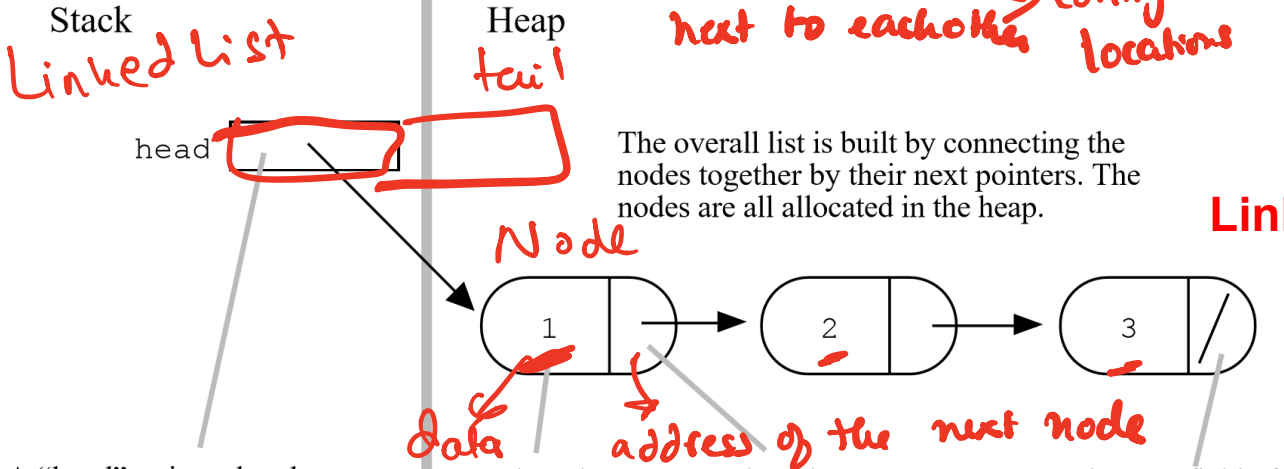
The Drawing Of List {1, 2, 3}



Array List

next to each other

contiguous memory locations



The overall list is built by connecting the nodes together by their next pointers. The nodes are all allocated in the heap.

Linked List

A "head" pointer local to BuildOneTwoThree() keeps the whole list by storing a pointer to the first node.

Each node stores one data element (int in this example).

Each node stores one next pointer.

The next field of the last node is NULL.

What is the key difference between these?

Pointers

- Pointer: A variable that contains the address of another variable
- Declaration: *type* * pointer_name;

```
int* p; // p stores the address of an int
```

What is output of the following code?

```
cout<<*p;
```

A. Random number
B. Undefined behavior
C. Null value

Dereferencing an uninitialized pointer will likely result in a seg fault

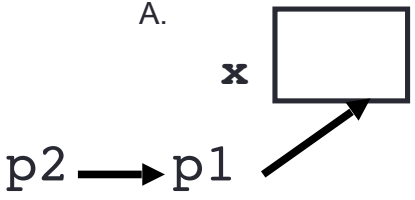
How do we initialize a pointer?

Review: Pointer assignment

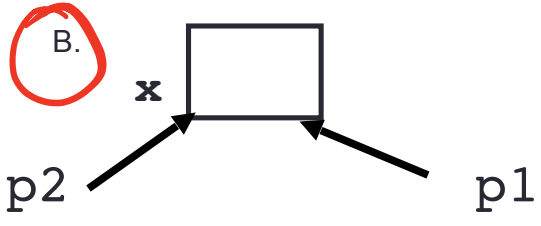
```
int *p1, *p2, x;
p1 = &x;
p2 = p1;
```

Q: Which of the following pointer diagrams best represents the outcome of the above code?

A.



B.



C. Neither, the code is incorrect

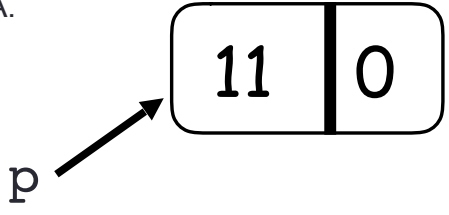
Review: Pointers to structs

```
Node x = {10, 0};
Node *p = &x;
p->data = p->data +1;
p = p->next;
```

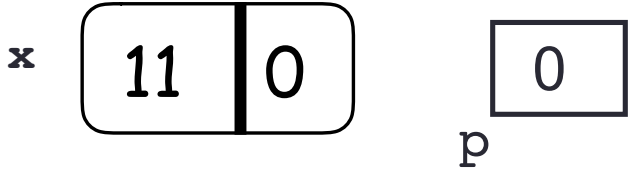
```
struct Node {
    int data;
    Node *next;
};
```

Q: Which of the following pointer diagrams best represents the outcome of the above code?

A.



B.



C. Neither, the code is incorrect

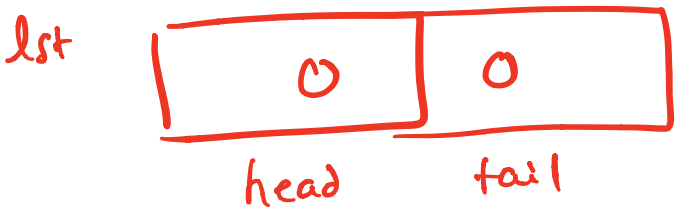
Create a two node list

- Define an empty list
- Add a node to the list with data = 10

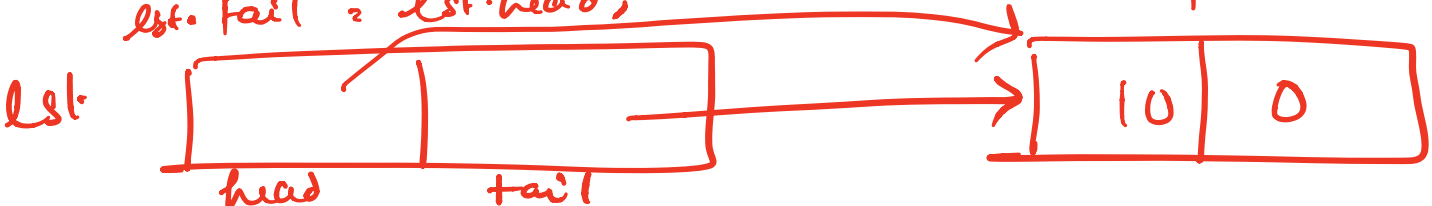
```
struct Node {
      int data;
      Node *next;
};
```

struct LinkedList{
    Node* head;
    Node* tail;
};

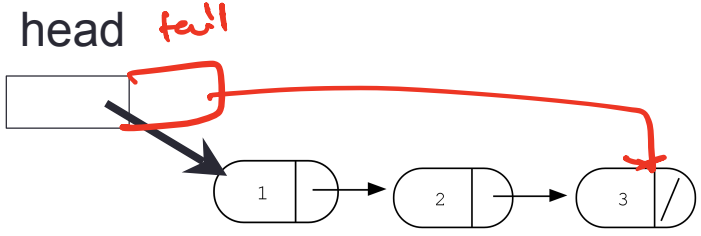
LinkedList lst = {0,0};



lst.head = new Node{10,0};
lst.tail = lst.head;



Iterating through the list

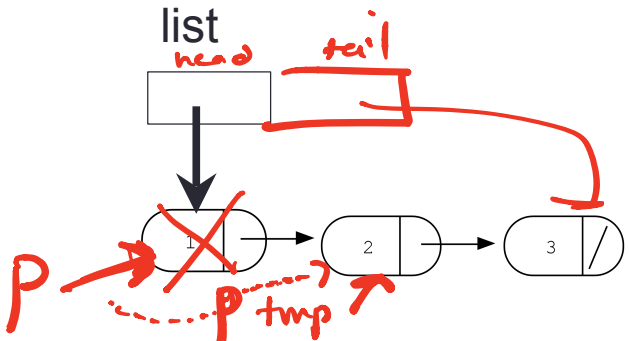



```
void printElements(LinkedList& list) {
   /* Print the values in the list */
        Node *p = list.head;
        while (p) {
             cout << p->data;
             p = p->next;
        }
}
```

Clear the list

```
Node* clearList(LinkedList& list) {
   /* Free all the memory that was created on the heap*/
        Node *p = list.head;
        while (p){
            Node * tmp = p->next;
            delete p;
            p = tmp
        }

}
```



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 front or at the rear of the list
  2. Delete (a value)
  3. Search (for a value)
  4. Min
  5. Max
  6. 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();                   // constructor
    ~LinkedList();                  // destructor
    // other methods
private:
    // definition of Node
    struct Node {
        int info;
        Node *next;
    };
    Node* head; // pointer to first node
    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 (taking linked lists as our running example)?
2. Is the default behavior the outcome we desire ?
3. If not, how should we overload these operators?

Behavior of default

Assume that your implementation of LinkedList uses the default destructor, copy constructor, copy assignment

```
void test_defaults(){
    LinkedList l1;
    l1.append(1);
    l1.append(2);
    l1.append(5);
    l1.print();
}
```

What is the expected behavior of the above code?

A. Compiler error

B. Memory leak

C. Code is correct, output: 1 2 5

D. None of the above

Behavior of default copy constructor

Assume that your implementation of LinkedList uses the overloaded destructor, default: copy constructor, copy assignment

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 the outcome we desire ?
* How do we change it?
```

Behavior of default copy assignment

Assume that your implementation of LinkedList uses the overloaded destructor, copy constructor, default copy assignment

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

```
void test_default_1(LinkedList& l1){
   LinkedList l2;
   l2 = l1;
}
* What is the default behavior?
```

Behavior of default copy assignment

Assume that your implementation of LinkedList uses the overloaded destructor, default: copy constructor, copy assignment

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

```
void test_default_2(LinkedList& l1){
   // Use the copy assignment
   LinkedList l2;
   l2.append(10);
   l2.append(20);
   l2 = l1;
}
* What is the default behavior?
```

Behavior of default copy assignment

Assume that your implementation of LinkedList uses the overloaded destructor, copy constructor, default copy assignment

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

```
void test_default_assignment(LinkedList& l1){
    // Use the copy assignment
    LinkedList l2;
    l2.append(10);
    l2.append(20);
    l2 = l1;
    l1 = l1;
}
* What is the default behavior?
```

Next time

- GDB
- Recursion