

# CSCI 2270

## Data Structures and Algorithms

### Lecture 8

Elizabeth White  
[elizabeth.white@colorado.edu](mailto:elizabeth.white@colorado.edu)

Office hours: ECCS 128

Wed 1-2pm

Thurs 2-3pm

# Admin

Lab next week: make your array sorted and make sure you test contains()

HW1 this weekend, due a week from Monday: Singly Linked List

Next week, practice questions will post for the exam.  
(NB: I will not post solutions for these.)

# Pseudocode

INSERTION-SORT( $A$ )

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

# Linked Lists

Lists are made up of nodes:

```
struct node {  
    int data;  
    node* next;    // pointer to next node  
};
```

We usually create pointers to list nodes, on the heap, using new.

```
node* my_first_node = new node();  
my_first_node->data = 3;  
my_first_node->next = nullptr;
```

# Linked Lists

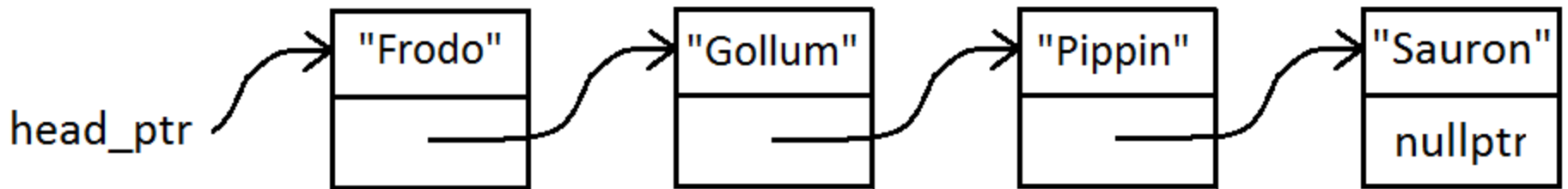
We store a pointer to the first node in the list.

If the list is empty, this pointer is a special value, called `nullptr`.

`nullptrs` aren't considered to point to any valid address.

If the list has only one node, then that node's next pointer is `nullptr`

If the list has more than one node, then the first node's next pointer gives the address of the second node, and we can keep tracing the list out node by node until we hit a `nullptr`.



# Print a list

If list is not empty, we'll print the nodes in sorted order by walking the list (constant reference!)

```
void print_list(const node*& head_ptr)
{
    const node* cursor = head_ptr;
    while (cursor != nullptr)
    {
        cout << cursor->data << " ";
        cursor = cursor->next;
    }
    cout << endl;
}
```

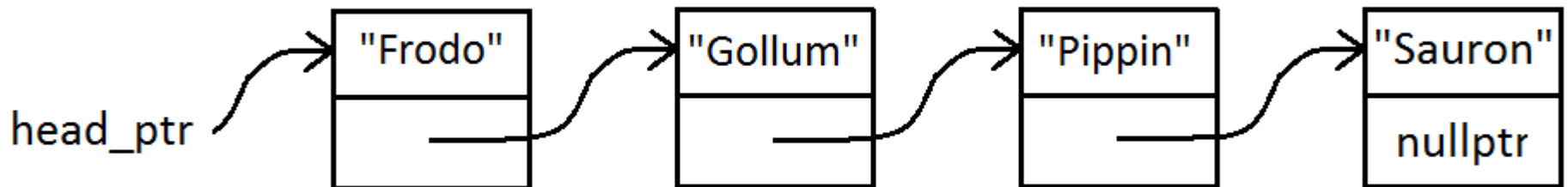
# Add to list

```
void add_node(node*& head_ptr, const int& payload)
```

If list is empty, set head\_ptr to a new node you create

If list is not empty,

1. you need to add the node at the front ("Bilbo")
2. you need to add the node in the middle ("Rosie")
3. you need to add the node at the end ("Tom Bombadil")



Adding is done by reconnecting the pointers

# Remove from list

```
bool remove_node(node*& head_ptr, const int& target)
```

If list is not empty,

1. you need to remove the node at the front (“Frodo”)
2. you need to remove a node in the middle (“Pippin”)
3. you need to remove the node at the end (“Sauron”)

Case 1: delete “Frodo” and update pointer to head

```
node* destructo = head_ptr;
```

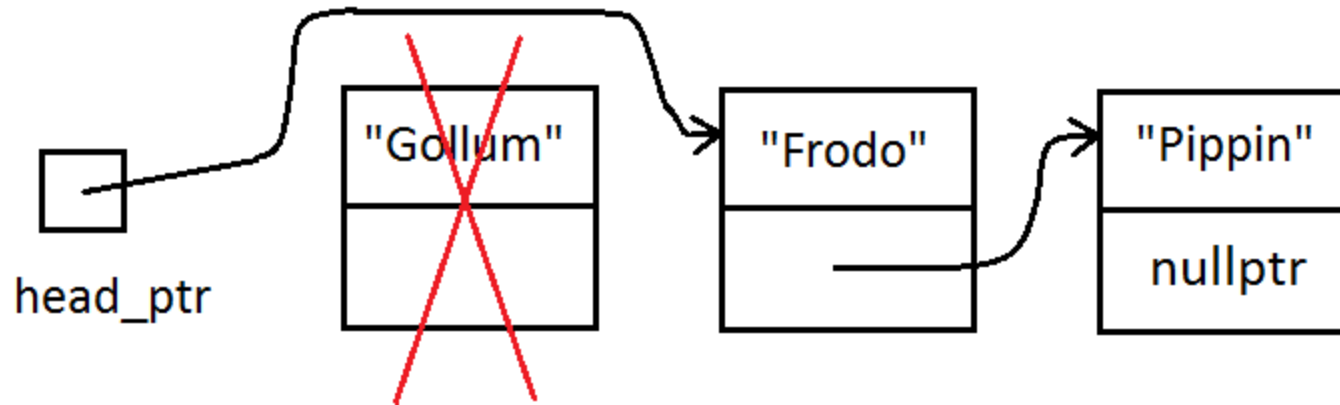
```
head_ptr = head_ptr->next;
```

```
delete destructo;
```



# Remove from list head

```
bool remove_node(node*& head_ptr, const int& target)
```



Case 1: delete “Gollum” and update pointer to head

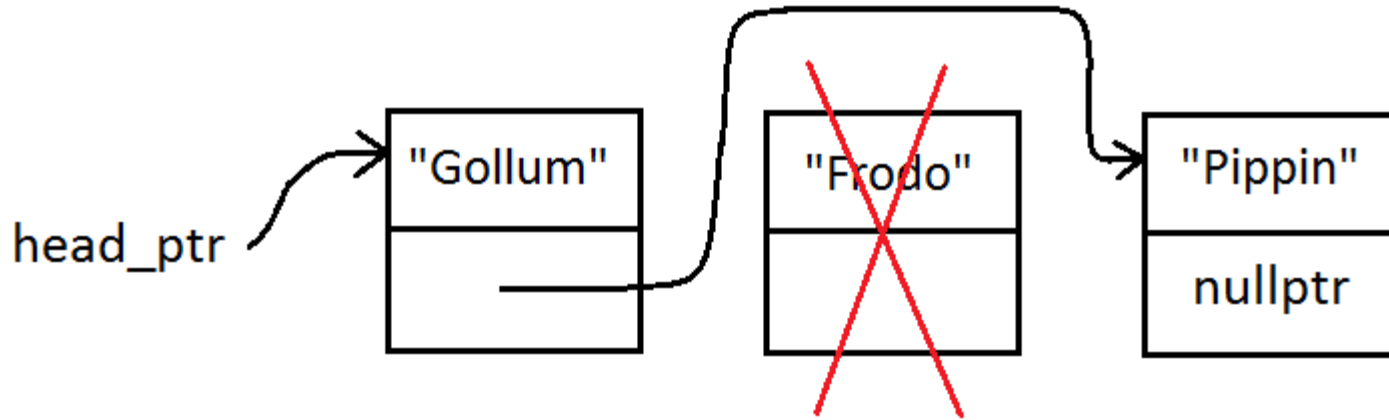
```
node* destructo = head_ptr;
```

```
head_ptr = head_ptr->next;
```

```
delete destructo;
```

# Remove from list middle

Case 2: delete "Frodo"



make pointer to node before deleted one;

call this node\* prev\_node

make pointer to node to delete

node\* destructo = prev\_node->next;

set prev\_node->next = prev\_node->next->next;

delete the node

delete destructo;

# Clearing a linked list

Need to delete each node without losing pointer to following ones...