CSCI 2270
Data Structures and Algorithms
Lecture 9

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128
Wed 1-2pm
Thurs 2-3pm

Admin

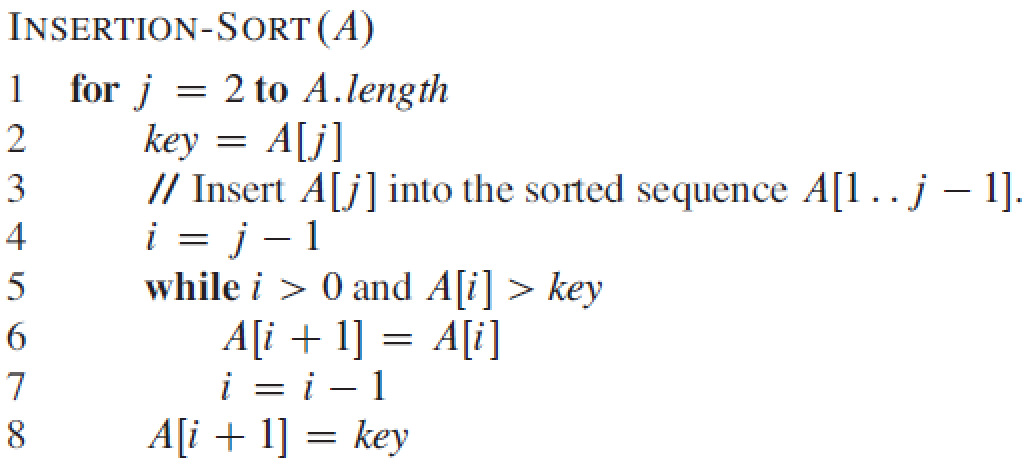
Lab next week: make your array sorted and make sure you test contains()

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($A$)
for $j = 2$ to $A.length$
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

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

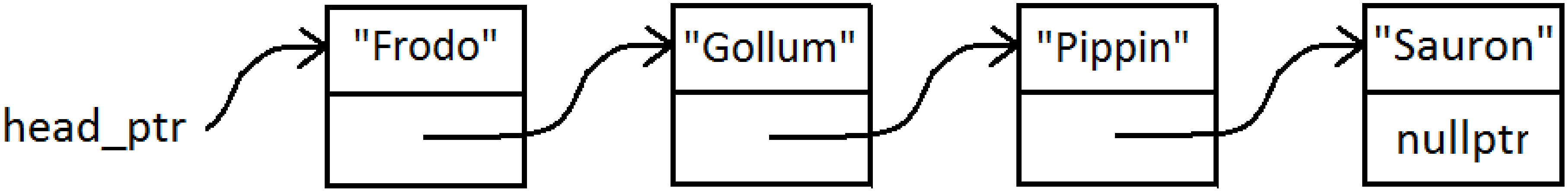
We store a pointer to the first node in the list.

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;
}
```

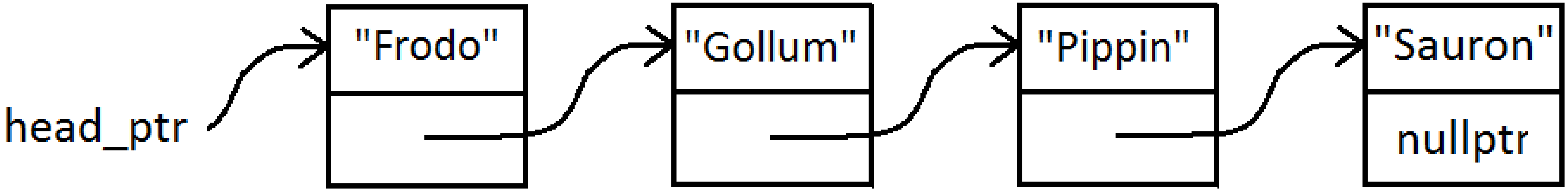
Add to list

```
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

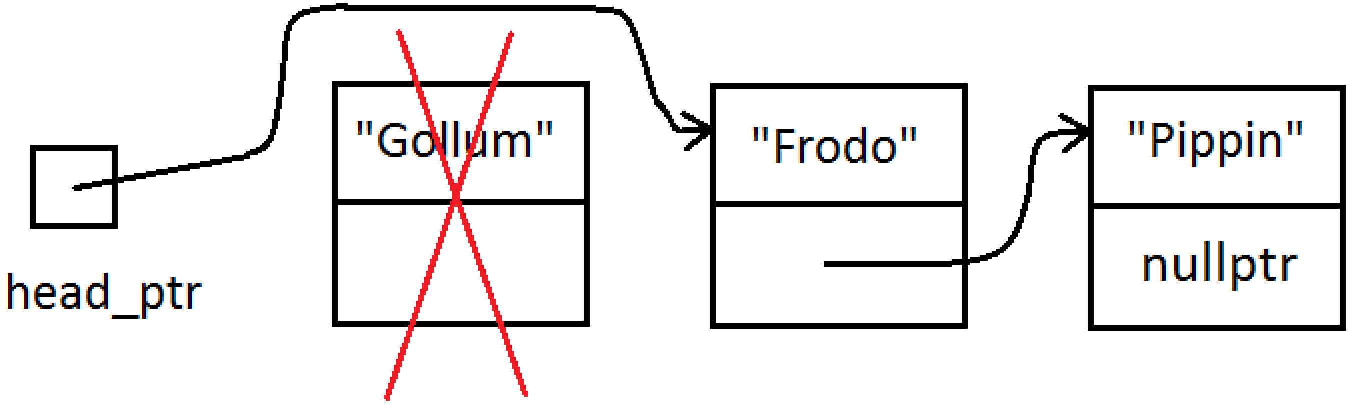
```
bool remove_node(node*& head_ptr, const int& target)
```

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”)

Remove from list head

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

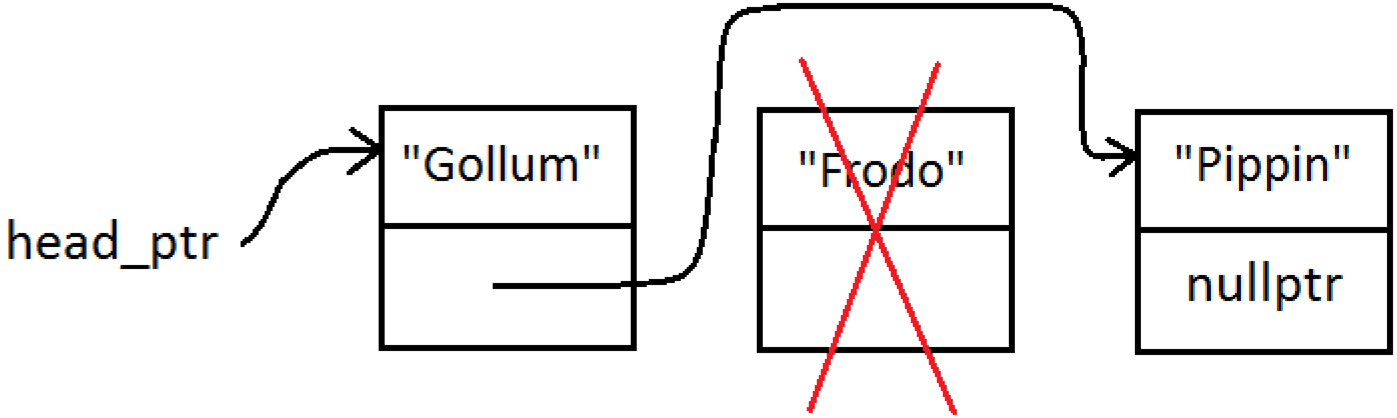


Case 1: delete “Gollum” and update pointer to head

```
node* destructo = head_ptr;
head_ptr = head_ptr->next;
delete destructo;
```

Remove from list middle

Case 2: delete “Frodo” (does this cover case 3 too?)



```
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 the node’s pointer to following ones… that would leak memory

One way… delete first node, redefining head_ptr as the second node, until the list goes away

```
void clear_list(node*& head_ptr)  {
        while (head_ptr != nullptr)  {
                node* destructo = head_ptr;  // node to delete
                head_ptr = destructo->next;
                delete destructo;
        }
}
```

Clearing a linked list

Another way…

```
void clear_list(node*& head_ptr)
{
        if (head_ptr == nullptr) return;
        if (head_ptr->next == nullptr)
        {
                delete head_ptr; head_ptr = nullptr; return;
        }
        clear_list(head_ptr->next);
}
```

Copying a linked list

We need an independent copy of the list (*deep* copy)

```
void copy(const node*& source_ptr, node*& dest_ptr)
{
        // step 1. if dest_ptr is not nullptr, clear this list
        // step 2. if source_ptr is not nullptr, add every item
        //     in the source_ptr list to the dest_ptr_list
}
```

Copying a linked list

What if we had a dependent *shallow* copy,

```
        void copy(const node*& source_ptr, node*& dest_ptr) {
                dest_ptr = source_ptr; }


node* node1 = nullptr;
add_node(node1, 5);
add_node(node1, 8);
add_node(node1, 9);
node* node2 = nullptr;
copy(node1, node2);
add_node(node1, 12);
print_list(node2);              // what prints here?
```

Linked list pitfalls

Falling off the list’s end

Consider a function to check if a linked list’s integers are in sorted order

```
bool check_list(const node* head_ptr)  {
        for (const node* cursor = head_ptr;
                ______________________________;
                cursor = cursor->next)  {
                if (cursor->data > cursor->next->data) return false;
        }
        return true;
}  // what if cursor == nullptr?  What if cursor->next == nullptr?
```

Linked list pitfalls

Another example: print every other element

Easy with array:

```
for (unsigned int q = 0; q < array_size; q += 2)
        cout << arr[q] << " “;
cout << endl;
```

Harder with list: 2 conditions to check, *and order matters*

```
for (const node* cursor = head_ptr;
        _______________________________;
        cursor = cursor->next;
        cout << cursor->data << " “;
cout << endl;
```

How much harder would it be to print every 7th element in a list?