CSCI 2270
Data Structures and Algorithms
Lecture 18

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 112/128
Wed 11:30am-1:30pm
Thurs 9am-11am

Administrivia

Colloquium talk Thursday:

"Automatically Proving Program Termination (and more)"
Byron Cook
Microsoft Research Cambridge and University College London
ECCR 265
Thursday, October 24
3:30-4:30 PM

ABSTRACT: Over the past 10 years, new techniques have been developed that allow us to automatically prove termination (and other related liveness properties) of non-trivial programs. This lecture will describe my work on the TERMINATOR program termination/liveness prover and its application to industrial software as well as pharmaceutical research.

Administrivia

Exam grading in process

Begin singly linked lists today, to prepare for HW4

HW4: store polynomial coefficients as doubly linked list

After the next 2-3 weeks, we'll pause the C++ content and move back to Java

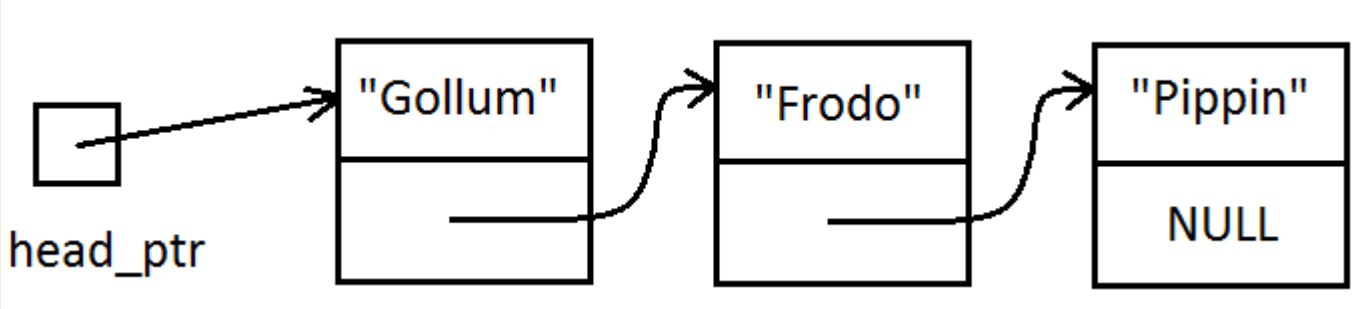
Linked list Node class

Linked lists depend on a Node class

```
template<class ItemType>
class Node
{
private:
  ItemType  item;                 // A data item
  Node<ItemType>* next;           // Pointer to next node
 …
```

Linked list Node class

```
template<class ItemType>
class Node
{
private:
  ItemType  item;                 // A data item
  Node<ItemType>* next;           // Pointer to next node
 …
```

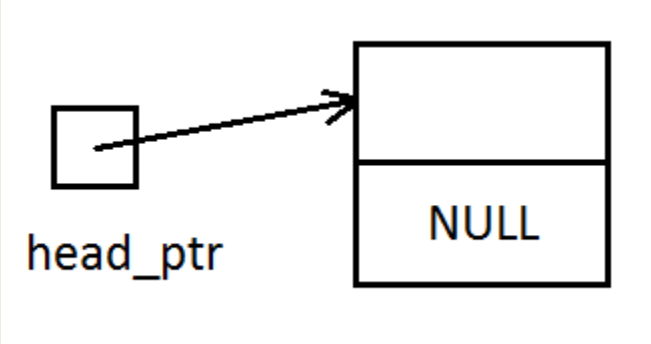


Linked list Node constructors

```
template<class ItemType>
class Node
{
public:
        // 3 constructors
        Node();
        Node(const ItemType& anItem);
        Node(const ItemType& anItem, Node<ItemType>*
nextNodePtr);
  …
```

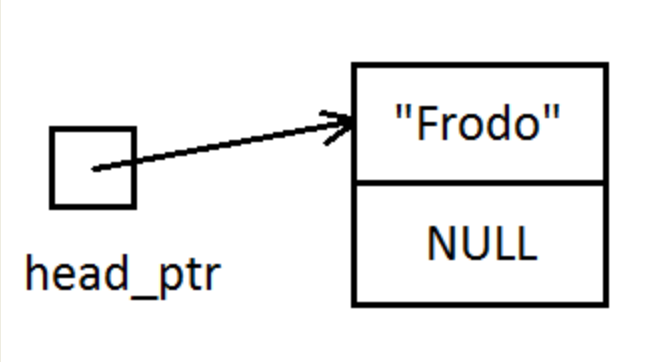
Default constructor example

```
Node<string>*head_ptr;     // Node* used in lists, not Node
head_ptr = new Node();     // head_ptr points to Node on heap
```



Item constructor example

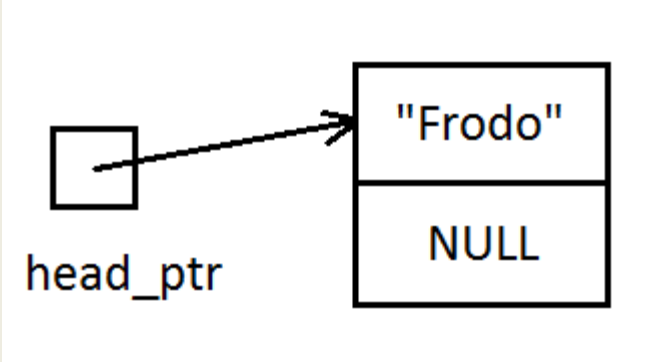
```
Node<string>*head_ptr;
head_ptr = new Node<string>("Frodo");
```



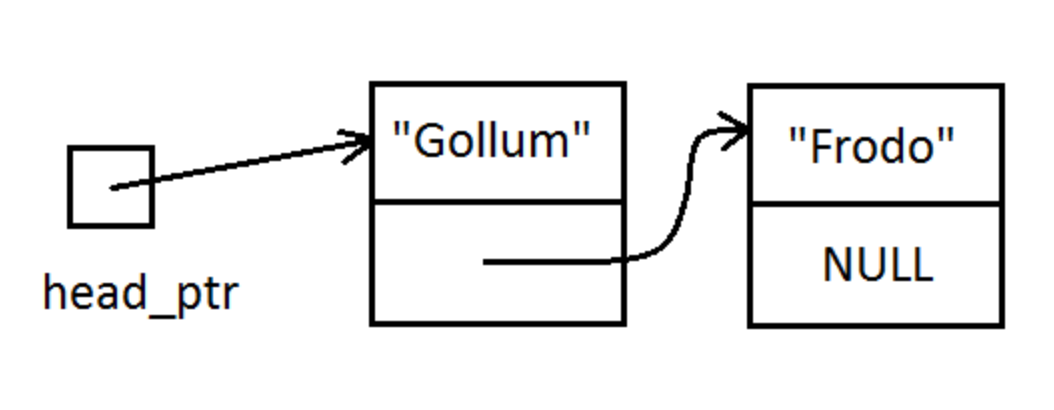
Link constructor example

Node<string>*head_ptr;

head_ptr = new Node<string>("Frodo");



head_ptr = new Node<string>("Gollum", head_ptr);

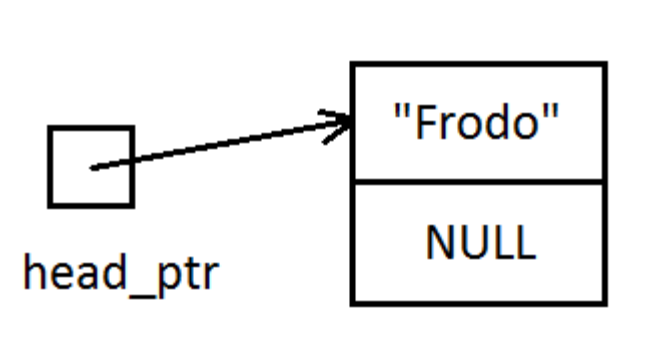


Get and set methods

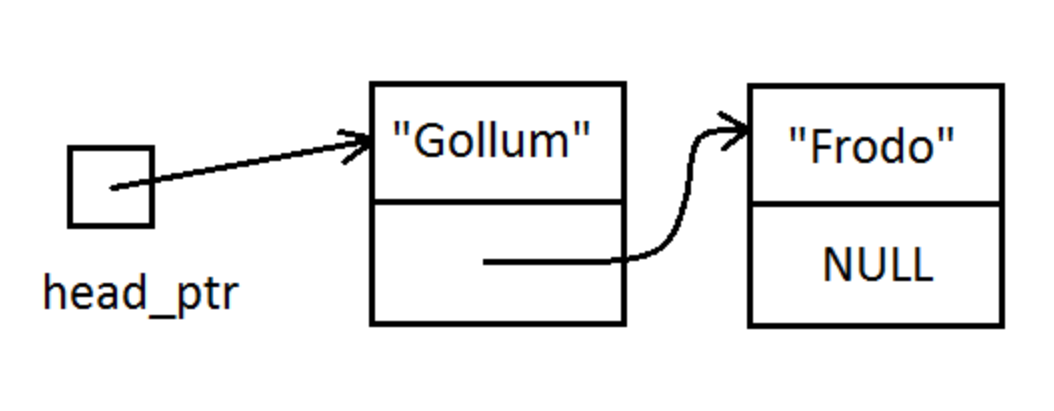
```
template<class ItemType>
class Node
{
public:
        …
        void setItem(const ItemType& anItem);
        void setNext(Node<ItemType>* nextNodePtr);
        ItemType getItem() const ;
        Node<ItemType>* getNext() const ;
} // end Node
```

Linked list Bag: add() method

Add method: can use link constructor



Add "Gollum". Why is it easier to do this at the front of the list?



Linked list bag: contains() method

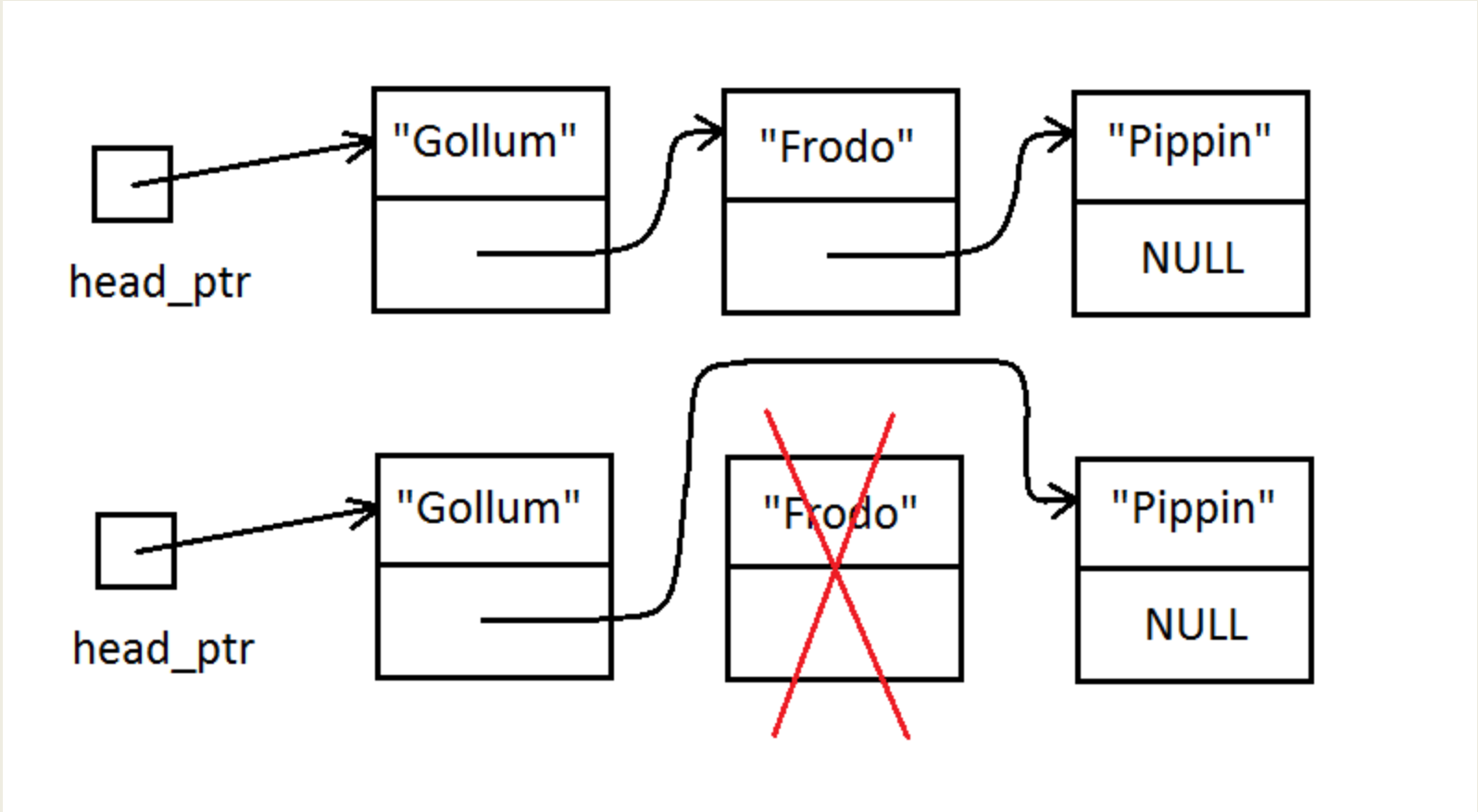
Contains() loops over all the nodes

```
Node<ItemType>* curr = head_ptr;
bool found = false;
while (curr != NULL && !found)
{
        if (curr->getItem() == anItem)
                found = true;
        else
                curr = curr->getNext();
}
```

Linked list Bag's remove() method

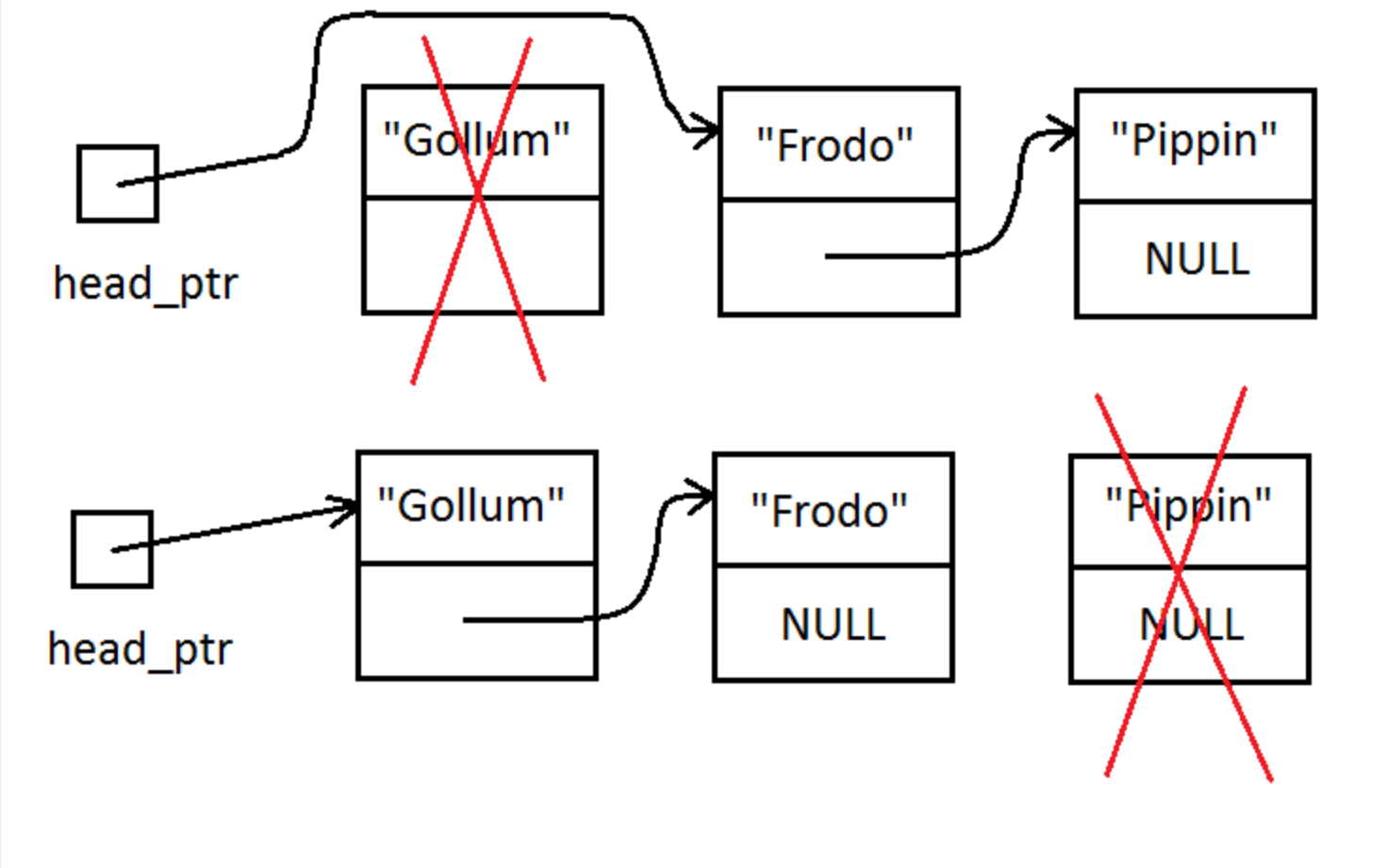
Remove() re-routes the pointers around the removed Item

Must delete the removed node as well



2 other remove() cases

Remove() may target the first or last Node in the list



Linked list pitfalls

Node<ItemType>* curr = head_ptr;

If head_ptr == NULL then the list is empty; check for this case

If curr == NULL then this segfaults:

curr = curr->getNext();

Each Node is created with new, must be destroyed by delete

else, memory leaks

Linked lists vs. arrays

Arrays are faster for random access []

- Easy to get the 8th element in the array
- List has to count to the 8th element from the head_ptr

Lists can be faster for add/remove, especially when sorted

- List re-routes pointers around the added/removed item
- Array has to shift all items up or down to preserve order

If you plan to add and remove a lot from the Bag, use a list.

If you plan to look up a lot of items in the Bag, use an array.