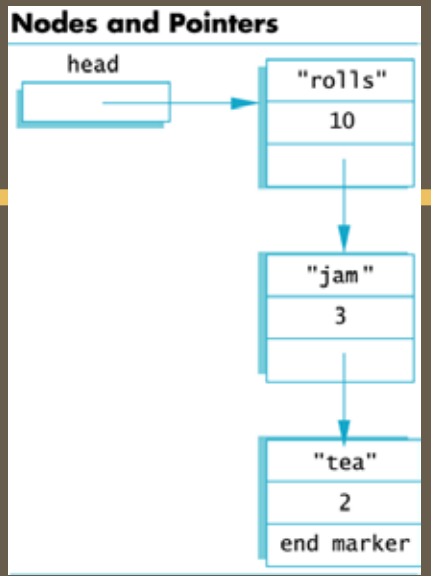
Exercises with Linked Lists

CS 16: Solving Problems with Computers I
Lecture #15

Ziad Matni
Dept. of Computer Science, UCSB

The **head** of a List



- The box labeled head, in Display 13.1, is not a node, but simply a **pointer variable** that points to a node

- Pointer variable head is declared as:

```
ListNodePtr head;
```

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
typedef ListNode* ListNodePtr;
ListNodePtr head;
```

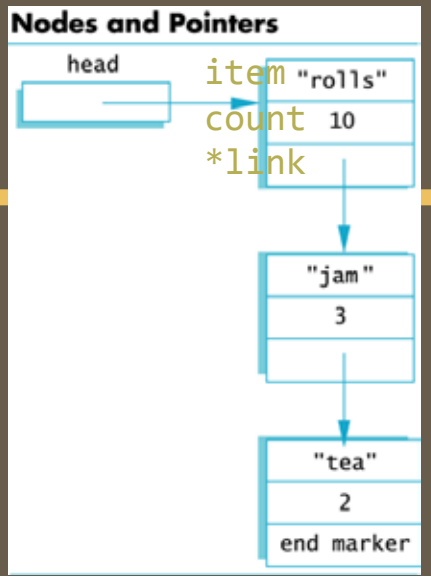
Accessing Items in a Node

- Looking at this example: one way to change the number in the first node from **10** to **12**:

```
(*head).count = 12;
```

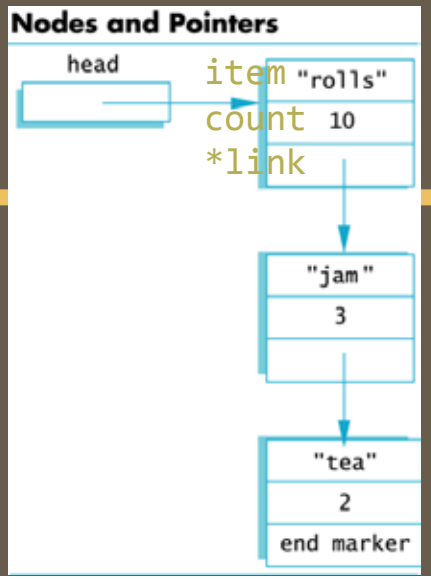
- **head** is a pointer variable to a node, so ***head** is the node that **head** points to

- The parentheses are necessary because the dot operator (.) has higher precedence than the dereference operator (*)




```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
typedef ListNode* ListNodePtr;
ListNodePtr head;
```

The Arrow Operator



- The arrow operator **->** combines the actions of the dereferencing operator ***** and the dot **.** operator

- Specifies a member of a **struct** or object pointed to by a pointer:

```
(*head).count = 12;
```

can be written as

```
head->count = 12;
```

- **The arrow operator is more commonly used than the (*head).*varName* approach**

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
typedef ListNode* ListNodePtr;
ListNodePtr head;
```

NULL

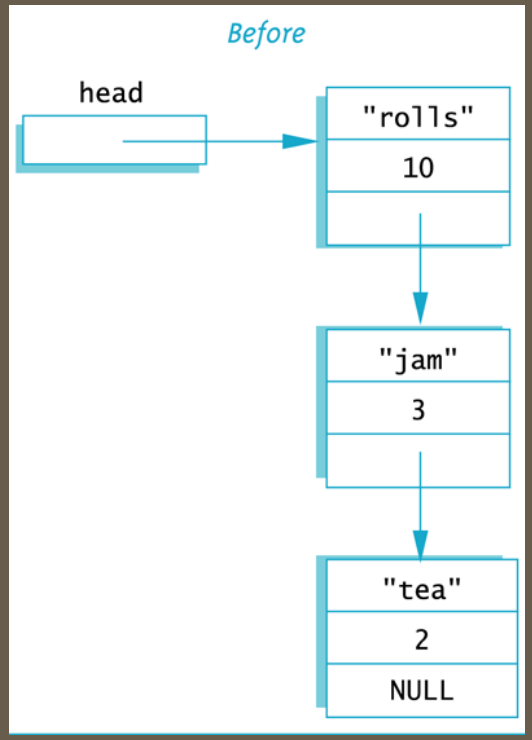
- The pre-defined constant NULL is used as an **end marker** for a linked list
  - A program can step through a list of nodes by following the pointers, but when it finds a node containing NULL, it knows it has come to the end of the list
- The value of a pointer that has nothing to point to is NULL
  - The value of NULL is 0

NULL

- A definition of NULL is found in several libraries, including <iostream>
- Any pointer can be assigned the value NULL:

```
double* there = NULL; // a pointer pointing to nothing
                      // C++ as Zen Buddhism?!
```

Accessing Node Data

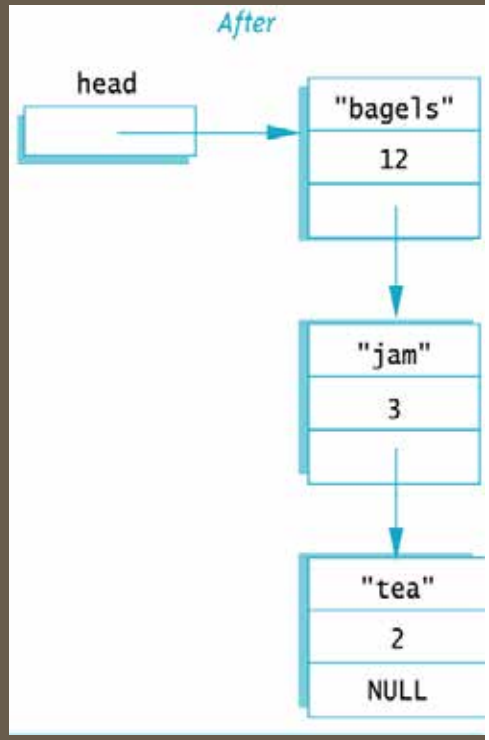



```
head->count = 12;
head->item = "bagels";

cout << head->count;
//prints 12

cout << head->link->count;
//prints 3

cout << head->link->link->item
//prints "tea"
```



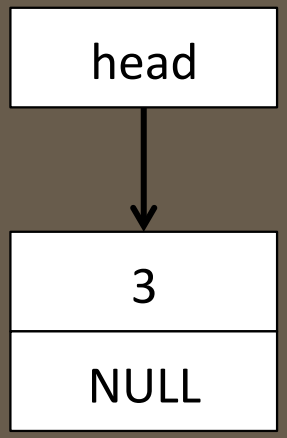
Building a Linked List

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

typedef Node* NodePtr;
NodePtr head;

head = new Node;

head->data = 3;
head->link = NULL;
```



Function **head_insert**

- Let's create a function that **inserts nodes** at the **head** of a list.

```
void head_insert(NodePtr& head, int the_number);
```

  - The first parameter is a **NodePtr** parameter that points to the first node in the linked list
  - The second parameter is the number to store in the list

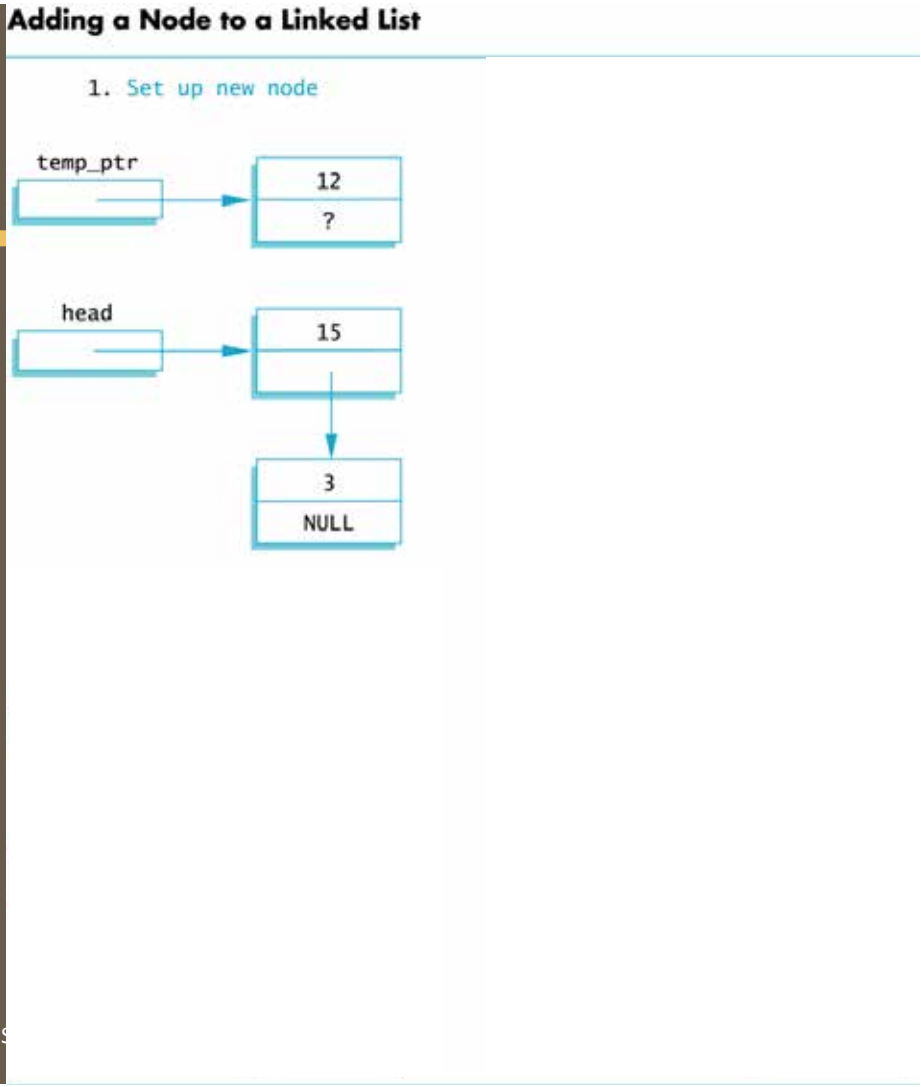
- **head_insert** will create a new node with **the_number**
  - First, we will copy **the_number** into a new node
  - Then, this new node will be inserted in the list as the new head node

Pseudocode for **head_insert**

1. Create a new dynamic variable pointed to by **temp_ptr**
2. Place the data (**the_number**) in the new node called ***temp_ptr**
3. Make **temp_ptr**'s link variable point to the **head** node
4. Make the head pointer point to **temp_ptr**

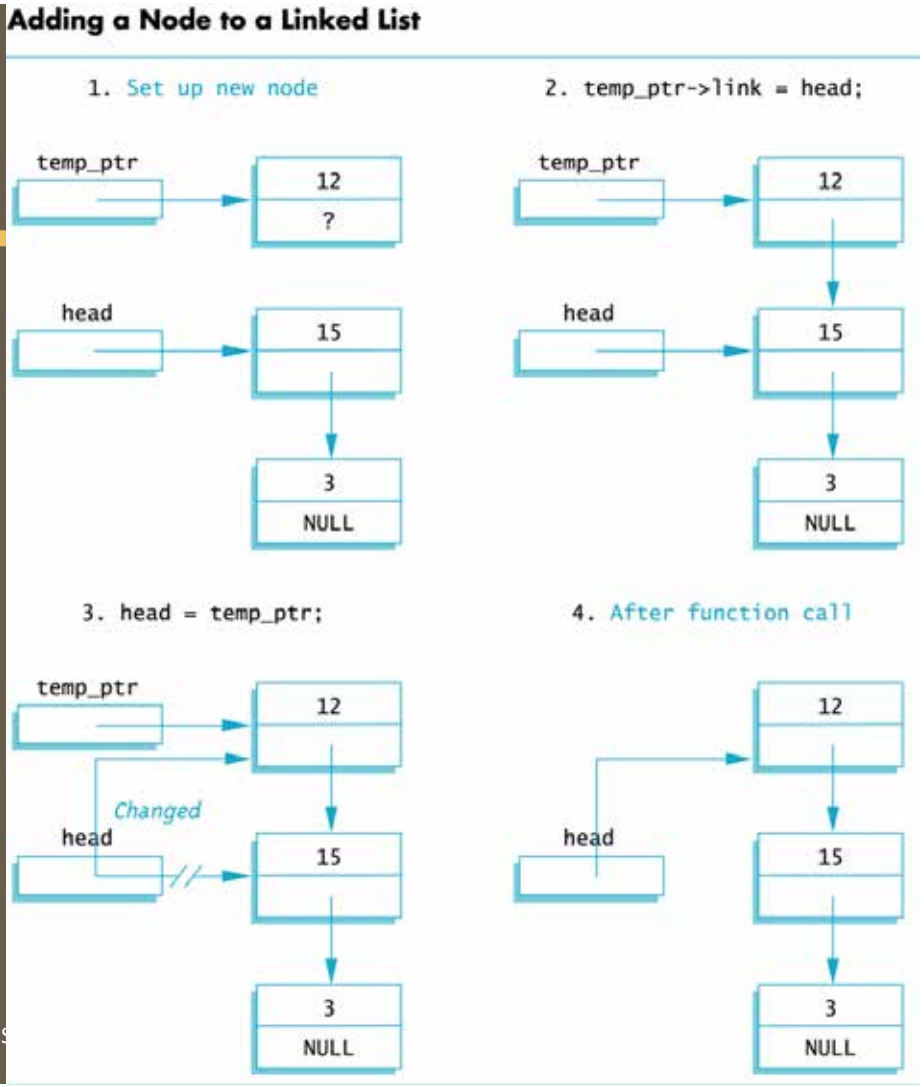
Pseudocode for **head_insert**

1. Create a new dynamic variable pointed to by **temp_ptr**
2. Place the data (**the_number**) in the new node called ***temp_ptr**
3. Make **temp_ptr**'s link variable point to the **head** node
4. Make the head pointer point to **temp_ptr**



Pseudocode for **head_insert**

1. Create a new dynamic variable pointed to by **temp_ptr**
2. Place the data (**the_number**) in the new node called ***temp_ptr**
3. Make **temp_ptr**'s link variable point to the **head** node
4. Make the head pointer point to **temp_ptr**



Translating **head_insert** to C++

```
#include <iostream>
using namespace std;

struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;
void head_insert(NodePtr& head, int the_number);

int main()
{
    NodePtr head;
    head = new Node;

    head->data = 3;
    head->link = NULL;

    head_insert(head, 5);

    return 0; }
```

```
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = head;
    head = temp_ptr;

}
```

Reversing a LL

- What if you wanted to go from Node1 -> Node2 -> Node3
  to Node3 -> Node2 -> Node1 ??

- It helps to think of other pointers showing you current, previous and next nodes
- Repeat the following thru the LL
  - Next becomes what current links to
  - Current then links to previous
  - Previous is now current
  - Current is now next
- Finally make h = previous and you've reversed it!

Memory Leaks

- Nodes that are lost by assigning their pointers a new address are not accessible any longer
- The program has no way to refer to the nodes and cannot delete them to return their memory to the heap (freestore)
- Programs that lose nodes have a memory leak
  - Significant memory leaks can cause system crashes

Searching a Linked List

- To design a function that will **locate** a particular node in a linked list:
  - We want the function to return a pointer to the node so we can use the data if we find it, else it should return NULL
  - The linked list is one argument to the function
  - The data we wish to find is the other argument
  - This declaration should work:

```
NodePtr search(NodePtr head, int target);
```

Function **search** (refined)

- We will use a local pointer variable, named **`here`**, to move through the list checking for the target
  - The only way to move around a linked list is to follow pointers

- We will start with **`here`** pointing to the first node and move the pointer from node to node following the pointer out of each node

Pseudocode for **search**

- Make pointer variable **here** point to the **head node**
- While ((**here** does not point to a node containing target)
  AND (**here** does not point to the last node))

  {

  make **here** point to the next node

  }
- If (**here** points to a node containing the target)

  return **here**;

  else

  return **NULL**;

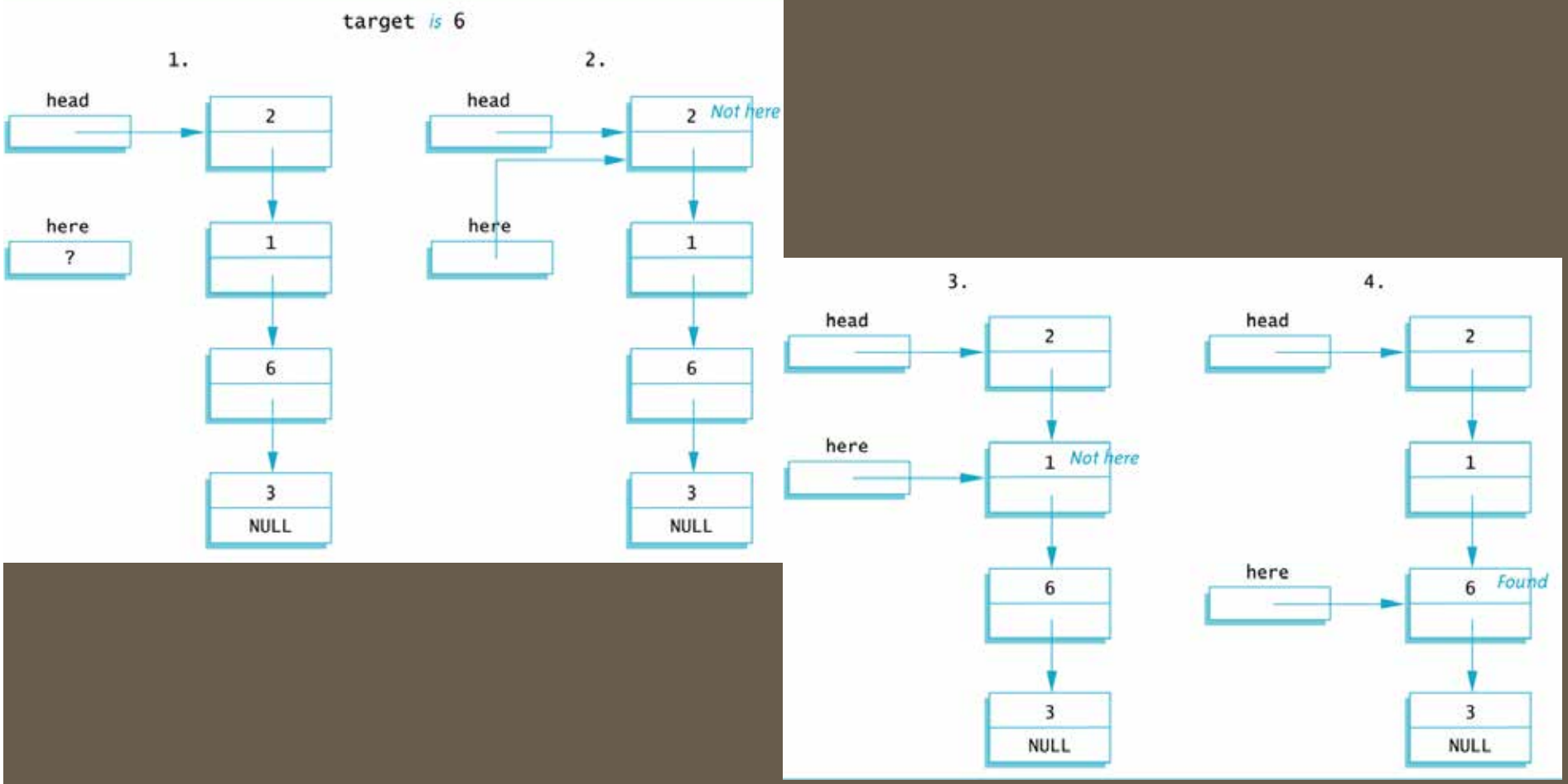
Moving Through the List

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

- The pseudocode for search requires that pointer **here** *step through the list*
- How does **here** follow the pointers from node to node?
  - When **here** points to a node, **here->link** is the *address of the next node*
- To make here point to the next node, make the assignment:

```
here = here->link;
```

Searching a Linked List



YOUR TO-DOs

- Start Lab 9 on Wednesday
- Do HW15 by **Thursday**

- Visit Prof's and TAs' office hours if you need help!

</LECTURE>