Linked Lists

CS 16: Solving Problems with Computers I
Lecture #16

Ziad Matni
Dept. of Computer Science, UCSB

FINAL EXAM IS COMING! Dec 12th!

- Material: ***Everything*** we've done
  - Homework, Labs, Lectures, Textbook
- **Tuesday, 12/12** in this classroom
- **Starts at 4:00pm **SHARP** (come early)**
- **Ends at 7:00pm **SHARP****
- **BRING YOUR STUDENT IDs WITH YOU!!!**
- Closed book: no calculators, no phones, no computers
- Only 1 sheet (double-sided ok) of written notes
  - Must be no bigger than 8.5" x 11"
  - **You have to turn it in with the exam**
- **You will write your answers on the exam sheet itself.**

DSP Students: Put in your requests TODAY!

Final Exam Preparation

- Your TA office hours
- Your prof’s office hours
- Exam prep questions (will post them on Piazza by the weekend)
- Exam review session with TAs next Thursday eve
  – Details to-be-announced later

Lecture Outline

- **Linked Lists (Ch. 13.1)**
  - We will cover everything in this section thru page XXX

- We are not covering **Ch. 13.2** section!

Pointers and Linked Lists

- Definition of Linked Lists:
  Linear collection of data elements, called ***nodes***, each pointing to the *next* node by means of a pointer

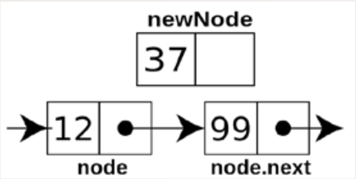
- List elements can easily be **inserted** or **removed** *without* reorganization of the entire structure (unlike arrays)

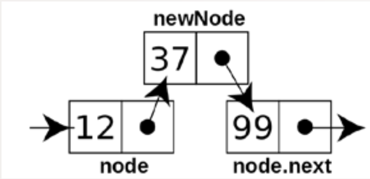
- Data items in a linked list do not have to be stored in one large memory block (again, unlike arrays)

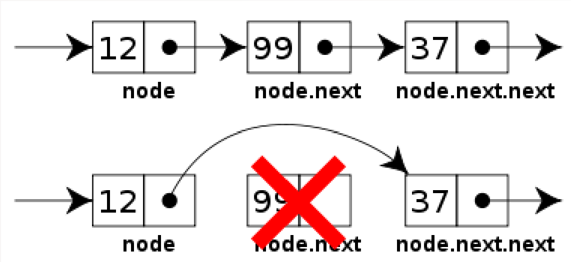
Linked Lists

- You can build a list of “nodes” which are made up of variables and pointers to create a chain.

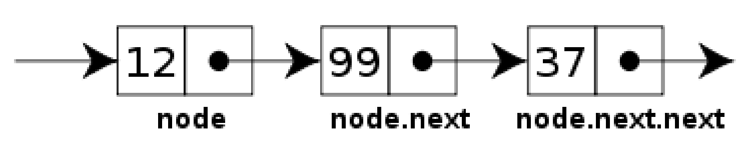
- Adding and deleting nodes in the link can be done by “re-routing” pointer links.







Nodes

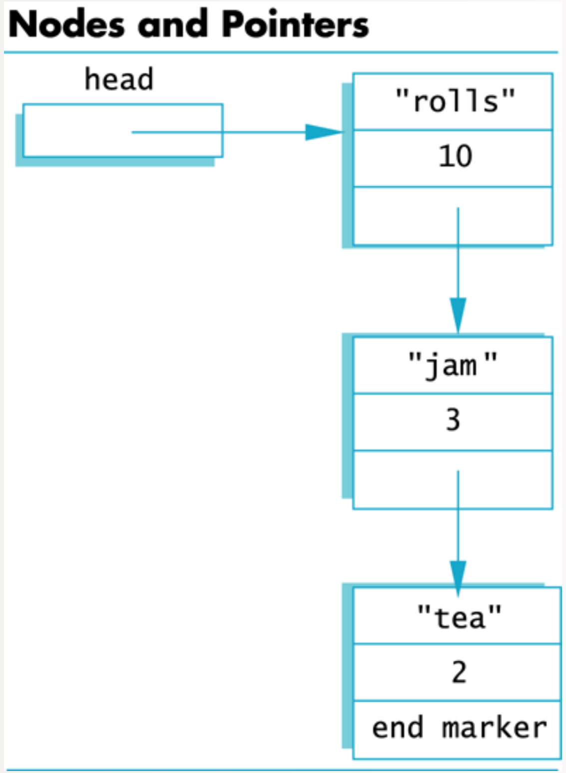


- The boxes in the previous drawing represent the **nodes** of a linked list
  - Nodes contain the data item(s) **and** a pointer that can point to another node of the same type
  - The pointers **point to an entire node**, not an individual item that might be in the node

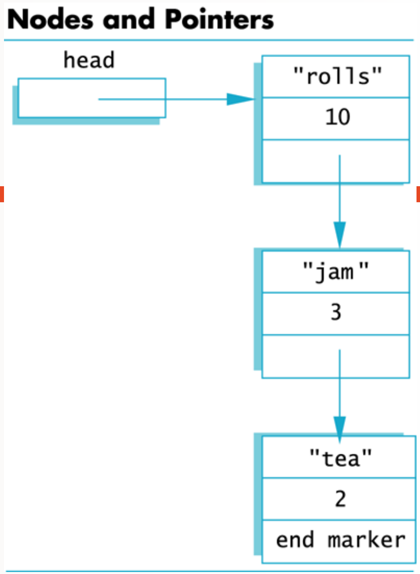
- The arrows in the drawing represent pointers

Nodes and Pointers – An Illustrated Example

(shown as Display 13.1 in the textbook)



Implementing Nodes



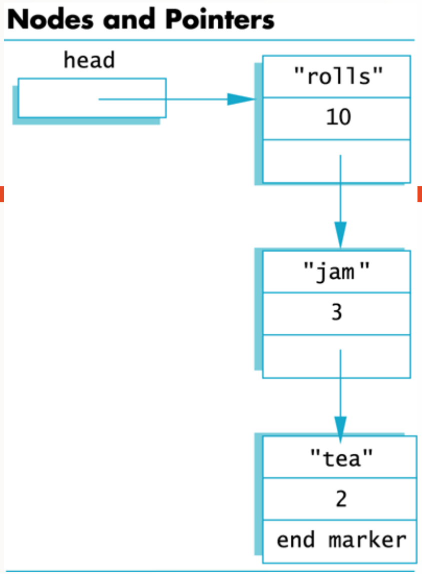
- Nodes are implemented in C++ as **structs** or **classes**
- *Example*: A structure to store two data items and a pointer to another node of the same type, along with a type definition might be:

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

typedef ListNode* ListNodePtr;
```

This circular definition is allowed in C++

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;
```

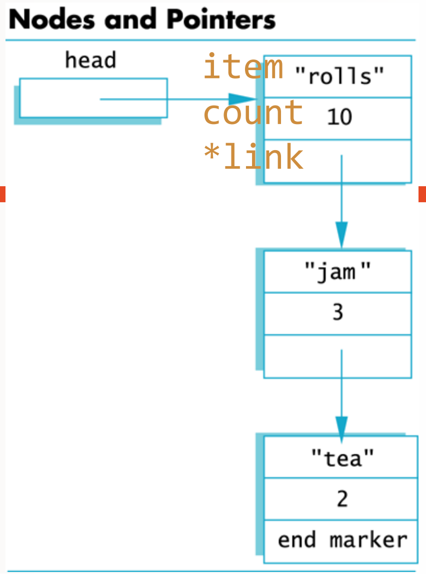
```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
typedef ListNode* ListNodePtr;
ListNodePtr head;
```

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;`

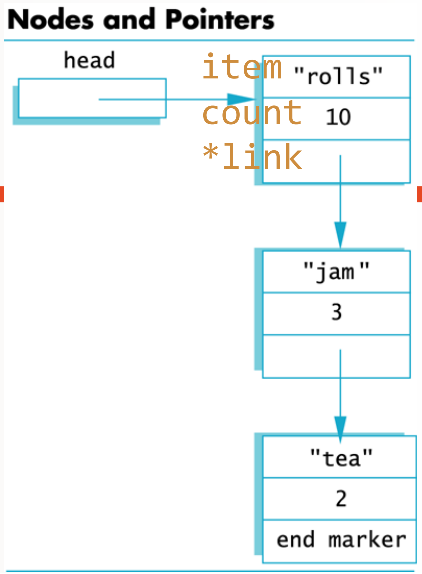
- **head** is a pointer variable to a node, so ***head** is the node that **head** points to

- 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;
```

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

```
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

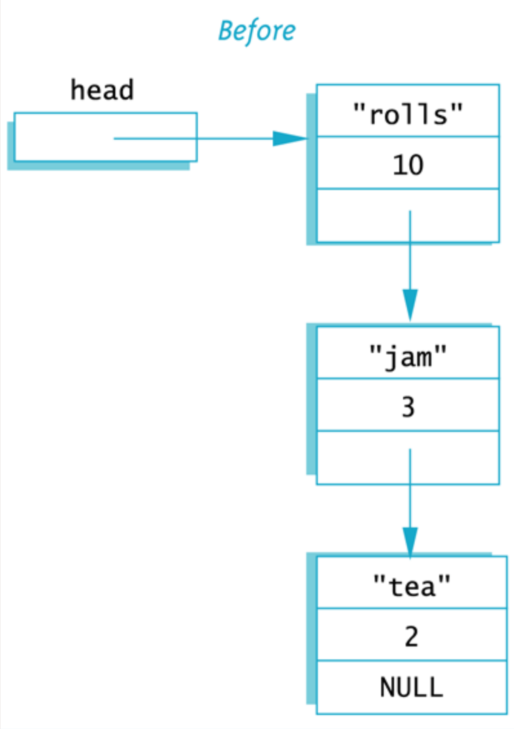
- The value of a pointer that has nothing to point to is NULL
  - The value of NULL is 0

NULL

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

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

Accessing Node Data

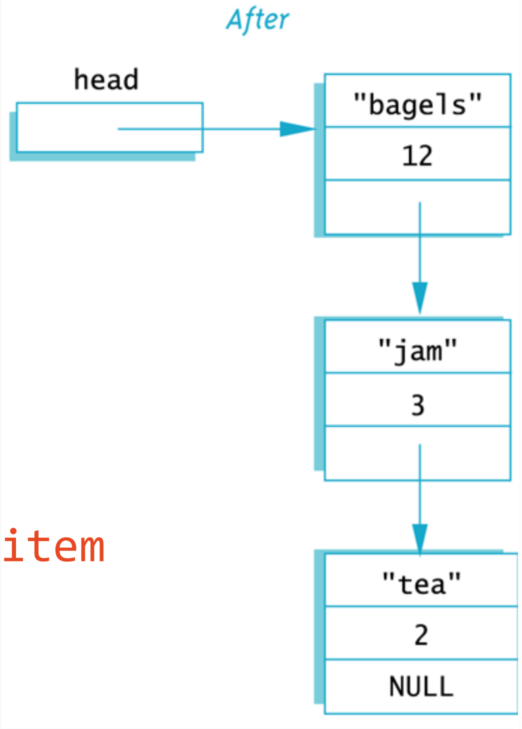



```
head->count = 12;
head->item = “bagels”;

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

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

cout << head->link->link->item
//prints “tea”
```



Linked Lists in a Nutshell

- The diagram in Display 13.2 depicts a linked list

- A linked list is a list of nodes in which each node has a member variable that is a pointer that points to the next node in the list
  - The first node is called the **head**
  - The pointer variable **head**, points to the first node
    - The pointer named **head** is not the head of the list…it points to the head of the list
  - The last node contains a pointer set to **NULL**

nullptr

- The fact that the constant NULL is actually the number 0 leads to an ambiguity problem.

Consider the overloaded function below:

```
void func(int *p);
void func(int i);
```

Which function will be invoked if we call `func(NULL)`?

- To avoid this, **C++11** has a new constant, **`nullptr`**.
  It is not the integer zero, but a literal constant used to represent a null pointer.
- Use **NULL** in your work for now, but understand the concept of **nullptr** also…

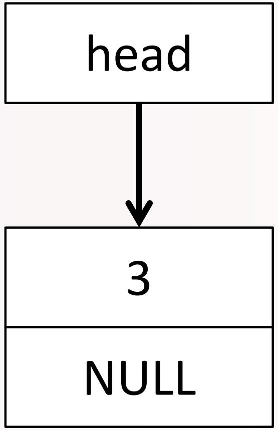
Building a Linked List

```
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

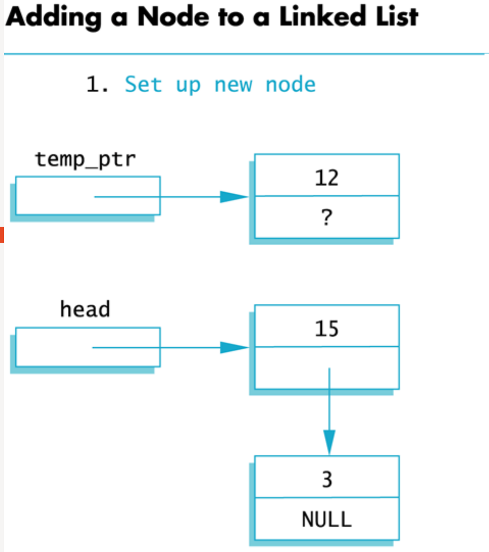
- **head_insert** will create a new node with **the_number**
  - First, we will copy the_number into a new node
  - Then, this new node will be inserted in the list as the new head node

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**

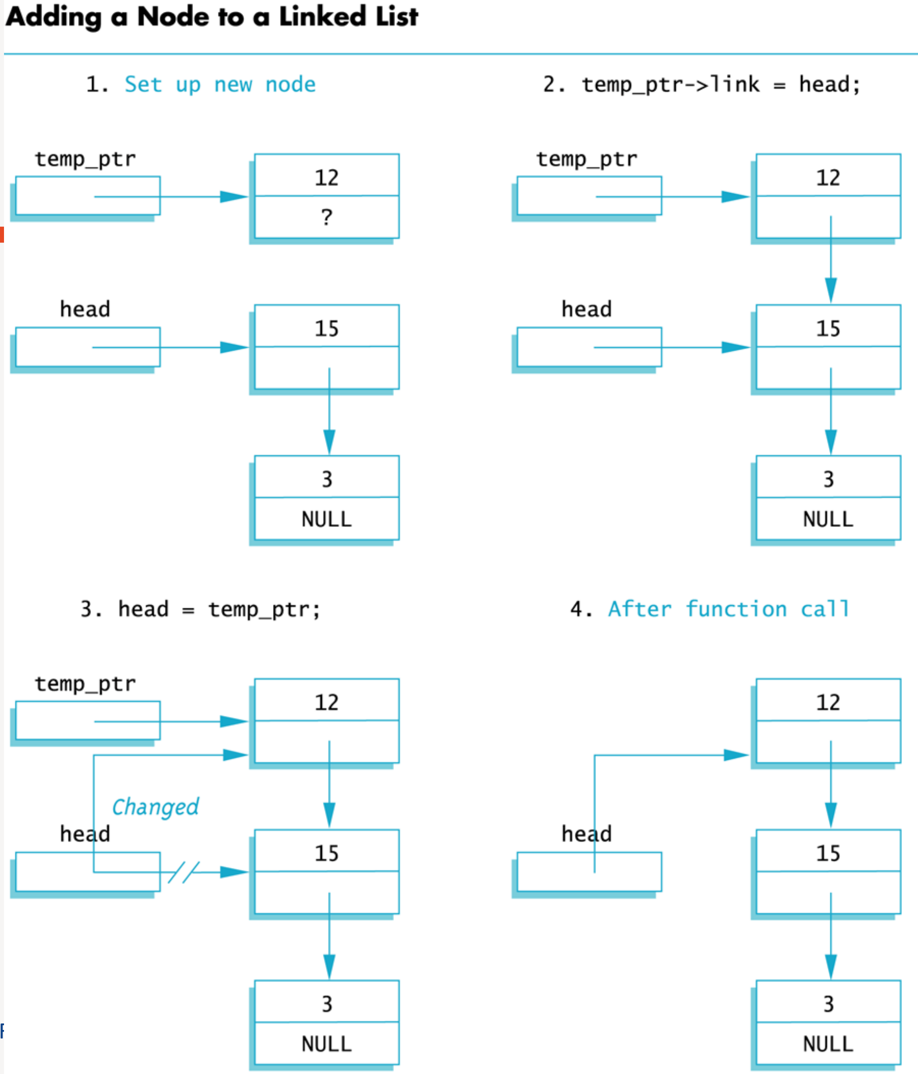
Pseudocode for **head_insert**

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 = nullptr;

    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;

}
```

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 (nullptr)
  - 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;
};
```

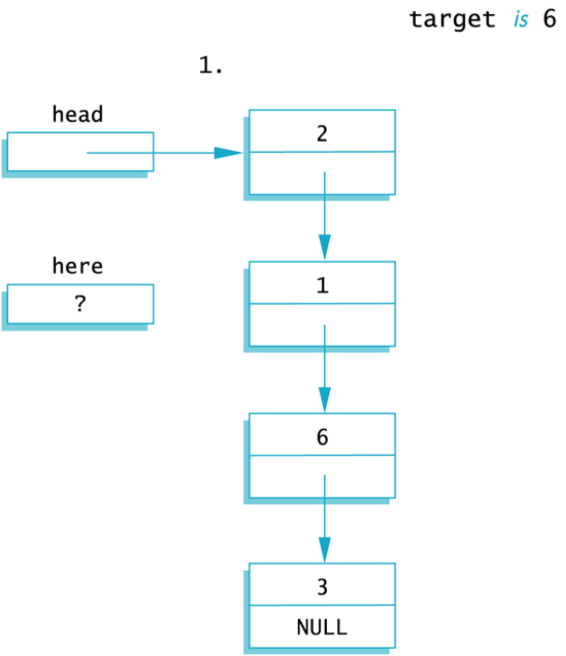
- The pseudocode for search requires that pointer **here** *step through the list*

- 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*

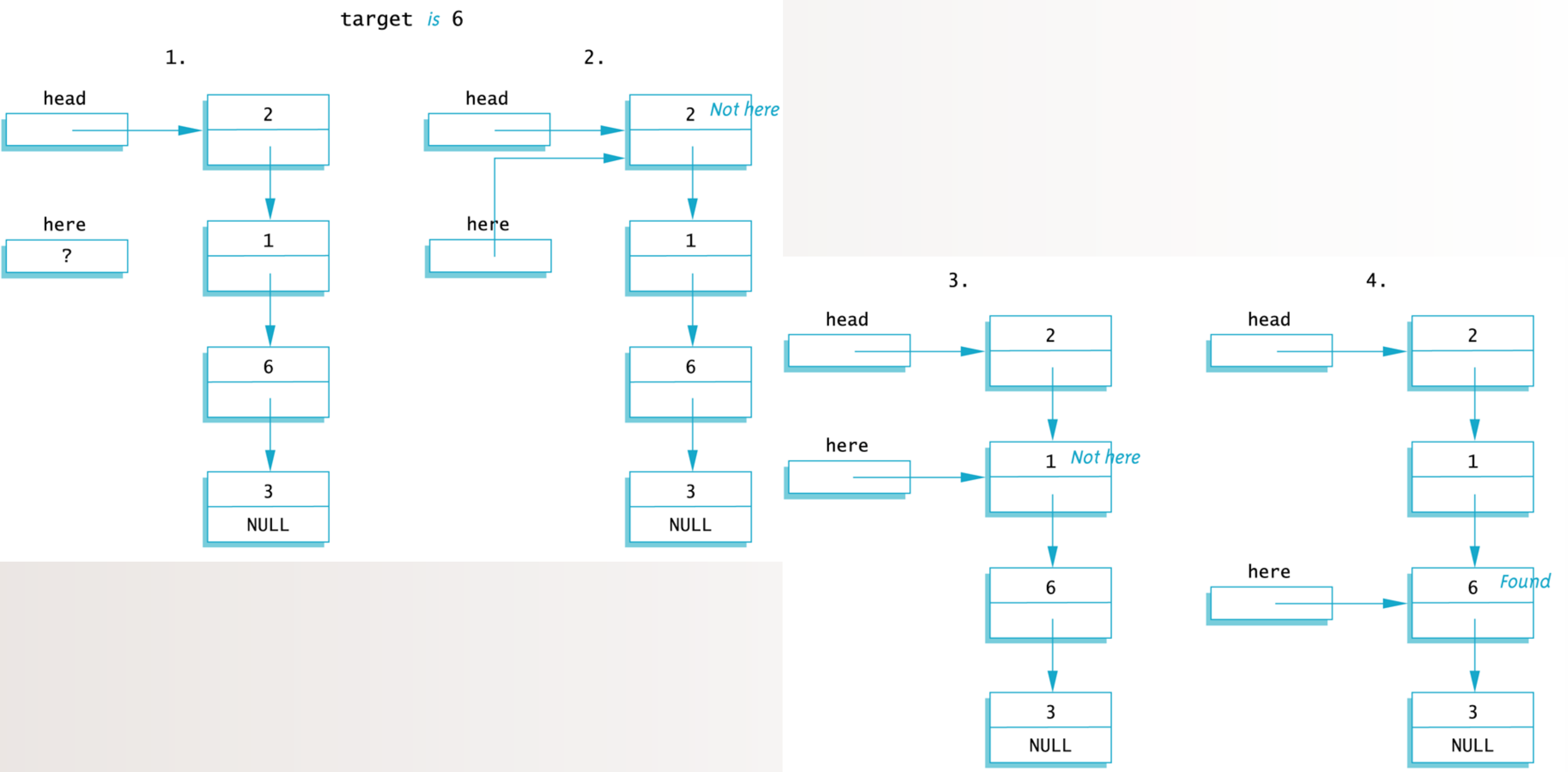
- To make here point to the next node, make the assignment:

`here = here->link;`

Searching a Linked List



Searching a Linked List



Translating **search** to C++

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

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

Typedef Node* NodePtr;
NodePtr search(NodePtr head, int target);

int main()
{
    …
    …
    someptr = search(head, 6);
    …
    return 0; }
```

```
NodePtr search(NodePtr head, int target)
{
    NodePtr here = head;

    if (here == NULL)
        return NULL;
    else
    {
//go thru the linked list and look for target
        while ((here->data != target) &&
                        (here->link != NULL))
            here = here->link;

//the while loop stopped b/c it either
// found target or it found nothing
        if (here->data == target)
            return here;
        else
            return NULL;
    }
}
```

Writing Code That Goes Thru a LL

```
//let's say you have a LL already defined…
Node *temp = new Node;
temp = head;
while(temp != NULL)
   {
      cout << temp->data << endl;
      temp = temp->next;
   }
```

Other Functions We Might Create for LLs…

- Insert node at the head
- Print out all the values in the LL
- Search the LL for a target

- Insert node *at the end* of LL
- Insert node *anywhere* in the LL
- Delete a node according to some target value criteria
- Sort an LL according to some target value criteria

etc…

YOUR TO-DOs

- ❑ HW 9 due Thu. 12/7
- ❑ Lab 9 due Wed. 12/6 by noon

- ❑ Read Ch. 14 on **Recursion** for Tuesday

- ❑ Visit Prof's and TAs' office hours if you need help!
- ❑ Smile! *And make people wonder why the heck you're smiling*

</LECTURE>