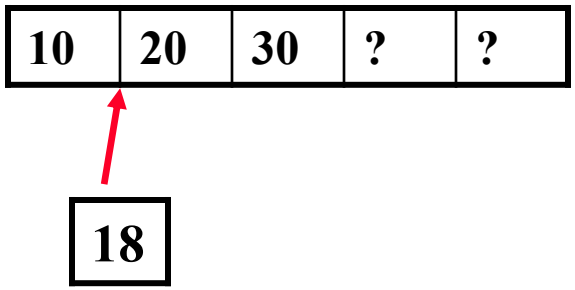
Linked List

Learning Objectives

- Design, implement, and test functions to manipulate node in a **linked list**, including inserting new nodes, removing nodes, searching for nodes, and processing (such as copying) that involves all the nodes of a list
- Design, implement, and test **collection classes that use linked lists** to store a collection of elements, generally using a node class to create and manipulate the linked list
- Analyze problems that can be solved with linked lists and, when appropriate, propose alternatives to simple linked lists, such as doubly linked lists and lists with dummy nodes
- Understand the **trade-offs between dynamic arrays and linked lists** in order to correctly select between the STL vector, list, and deque classes

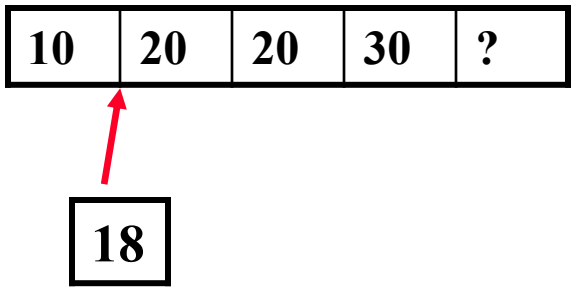
Motivation

- In a sequence using an array, inserting a new item needs to move others back...



Motivation

- In a sequence using an array, inserting a new item needs to move others back...



Motivation

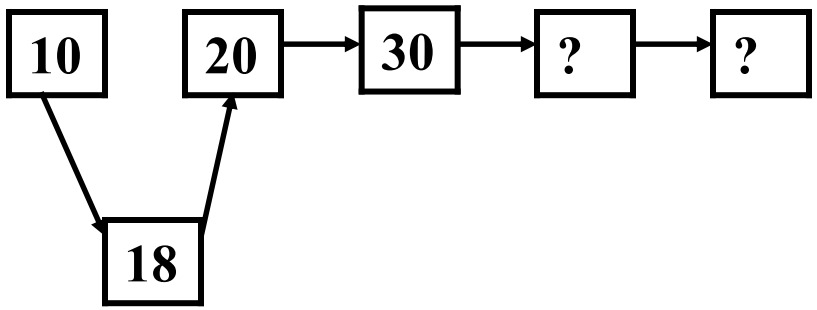
- In a sequence using an array, inserting a new item needs to move others back...

10	18	20	30	?

- So the time complexity of the insert is O(n)

Motivation

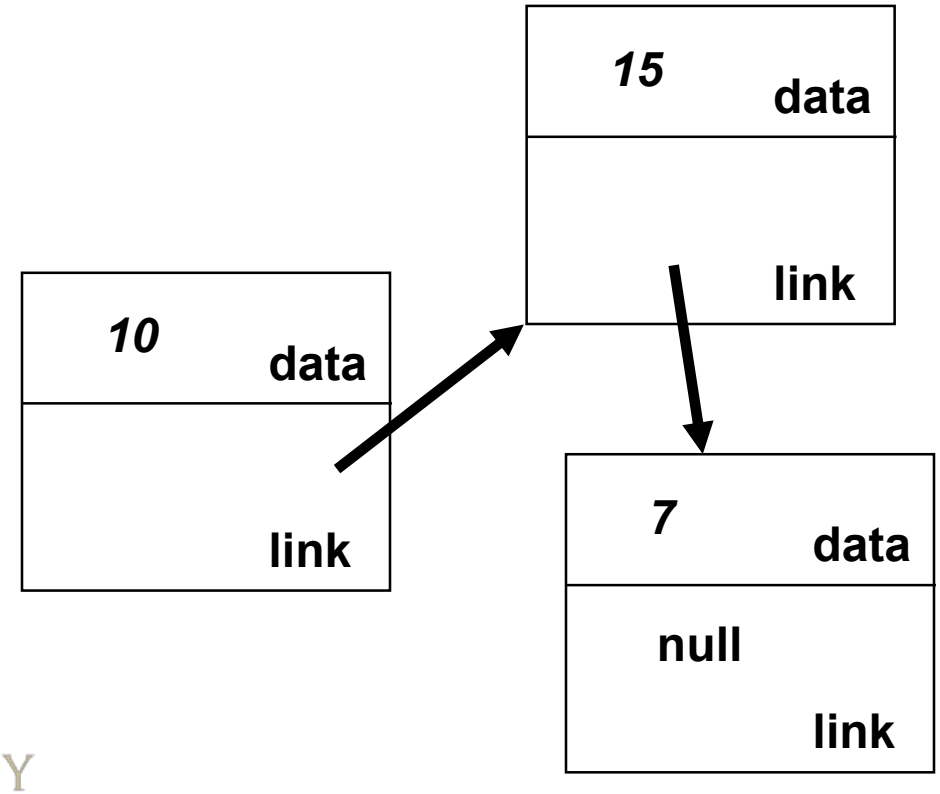
- How can we insert a new item without moving others?
  - Need a new data structure



Declarations for Linked Lists

- Each node in the linked list is a class, as shown here.

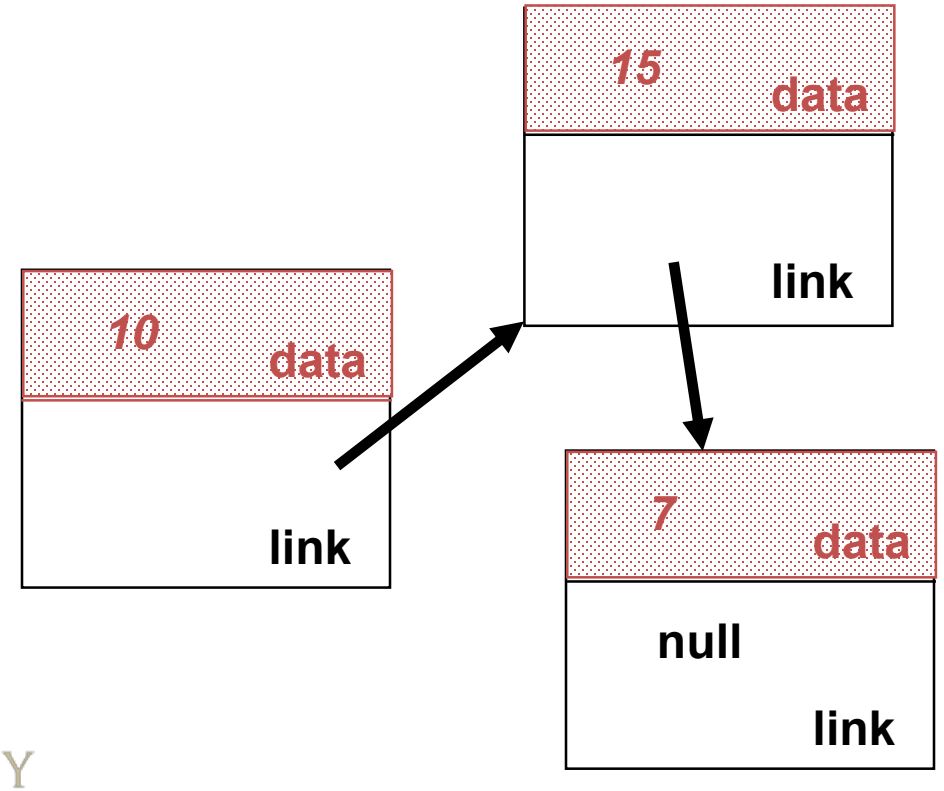
```
class node
{
public:
    typedef int value_type;
    ...
private:
    value_type data;
    node *link;
};
```



Declarations for Linked Lists

- The data portion of each node is a type called **value_type**, defined by a typedef.

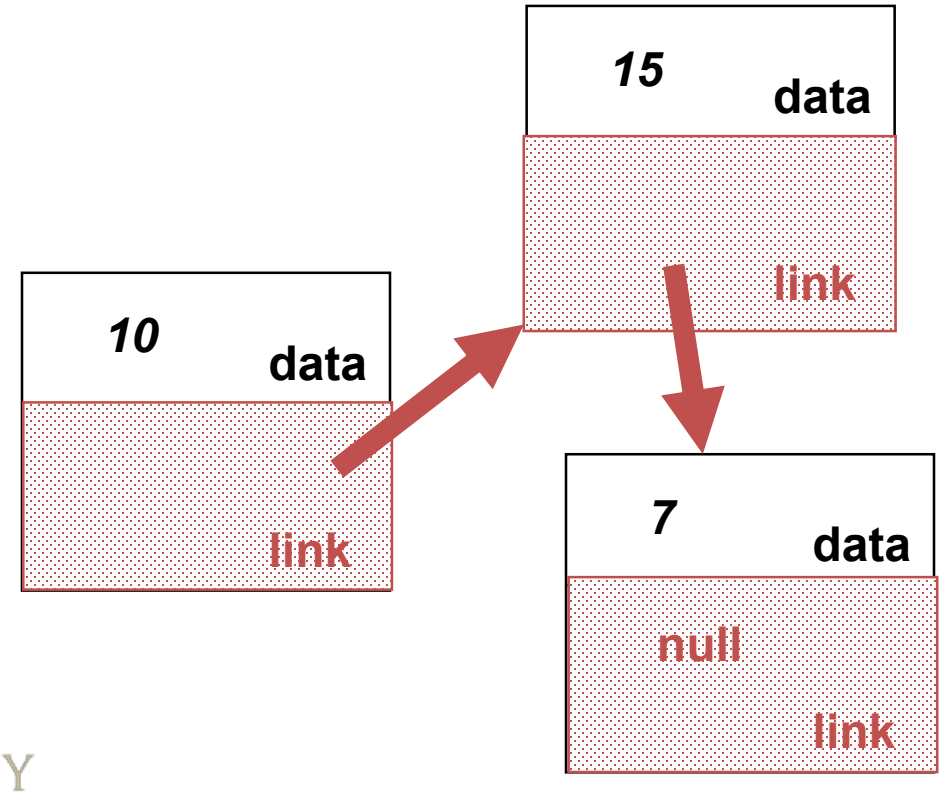
```
class node
{
public:
    typedef int value_type;
    ...
private:
    value_type data;
    node *link;
};
```



Declarations for Linked Lists

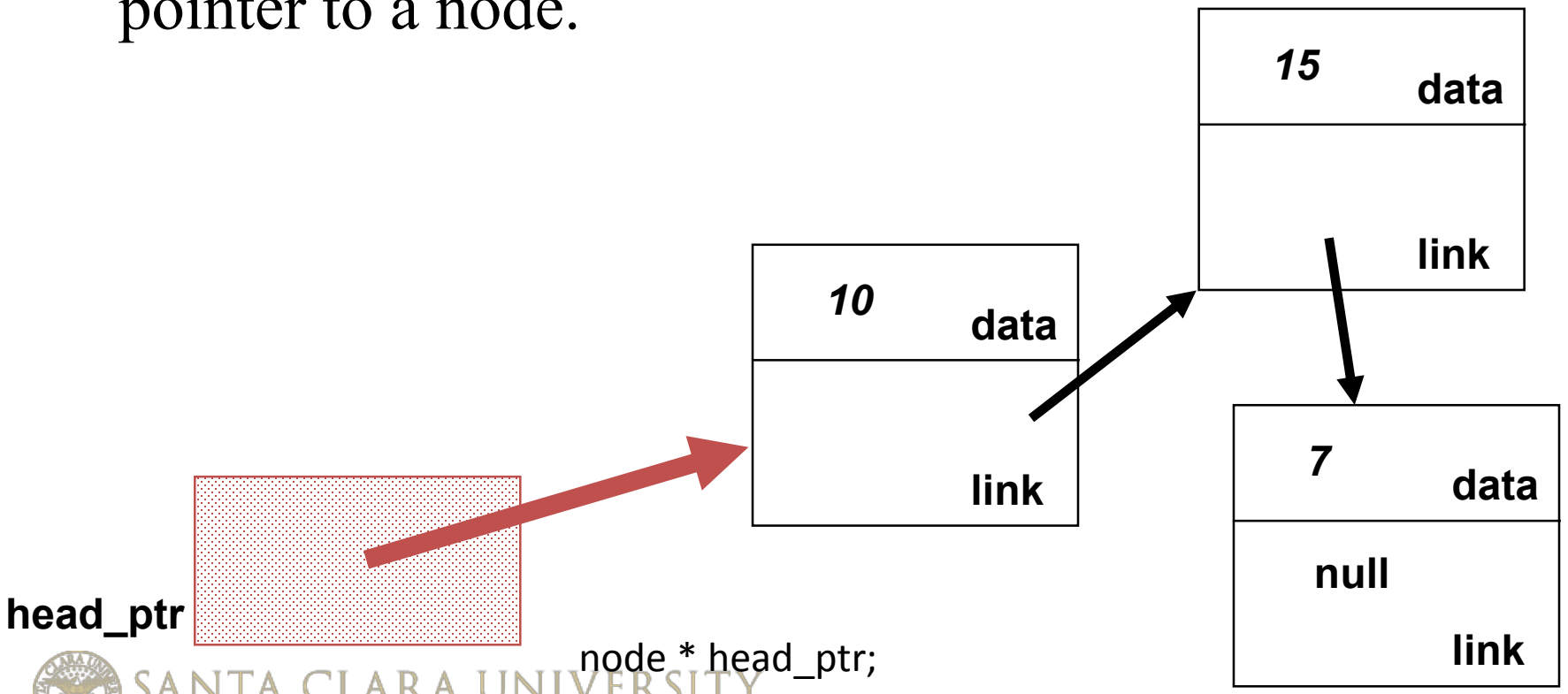
- Each node also contains a link field which is a pointer to another node.

```
class node
{
public:
    typedef int value_type;
    ...
private:
    value_type data;
    node *link;
};
```



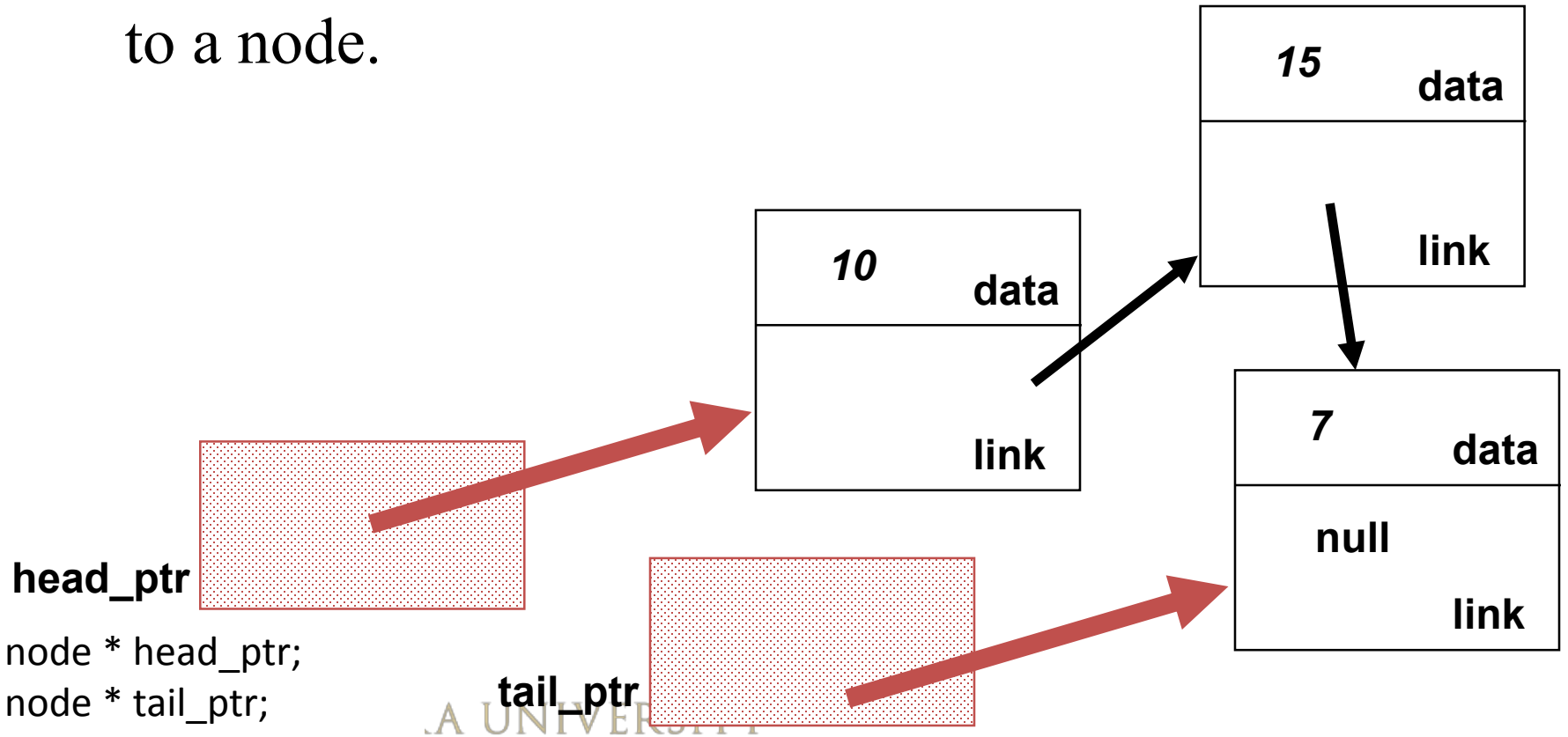
Declarations for Linked Lists

- A program can keep track of the first node by using a pointer variable such as **head_ptr** in this example.
- Notice that head_ptr itself is not a node -- it is a pointer to a node.



Declarations for Linked Lists

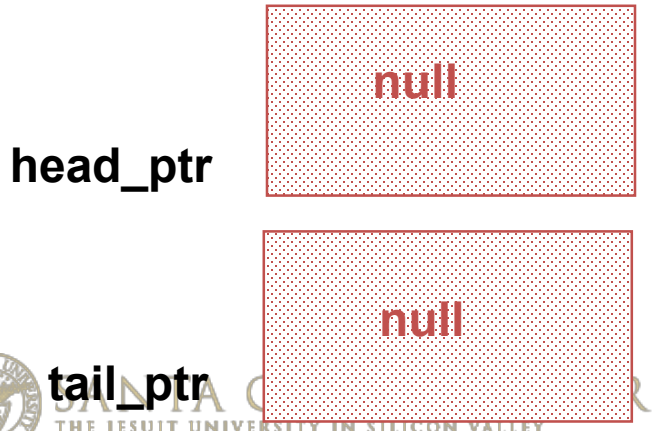
- A program can also keep track of the last node by using a pointer variable such as **tail_ptr**
- Notice that tail_ptr itself is not a node -- it is a pointer to a node.




```
node * head_ptr;
node * tail_ptr;
```

Declarations for Linked Lists

- A program can keep track of the first and the last nodes by using pointer variables such as **head_ptr, tail_ptr**.
- Notice that neither head_ptr nor tail_ptr is a node -- it is a pointer to a node.
- For an empty list, **null** ***is stored*** in both the head and the tail pointers.




```
node * head_ptr;
node * tail_ptr;
head_ptr = NULL;
tail_ptr = NULL;
// NULL can be  used for any pointers!
```

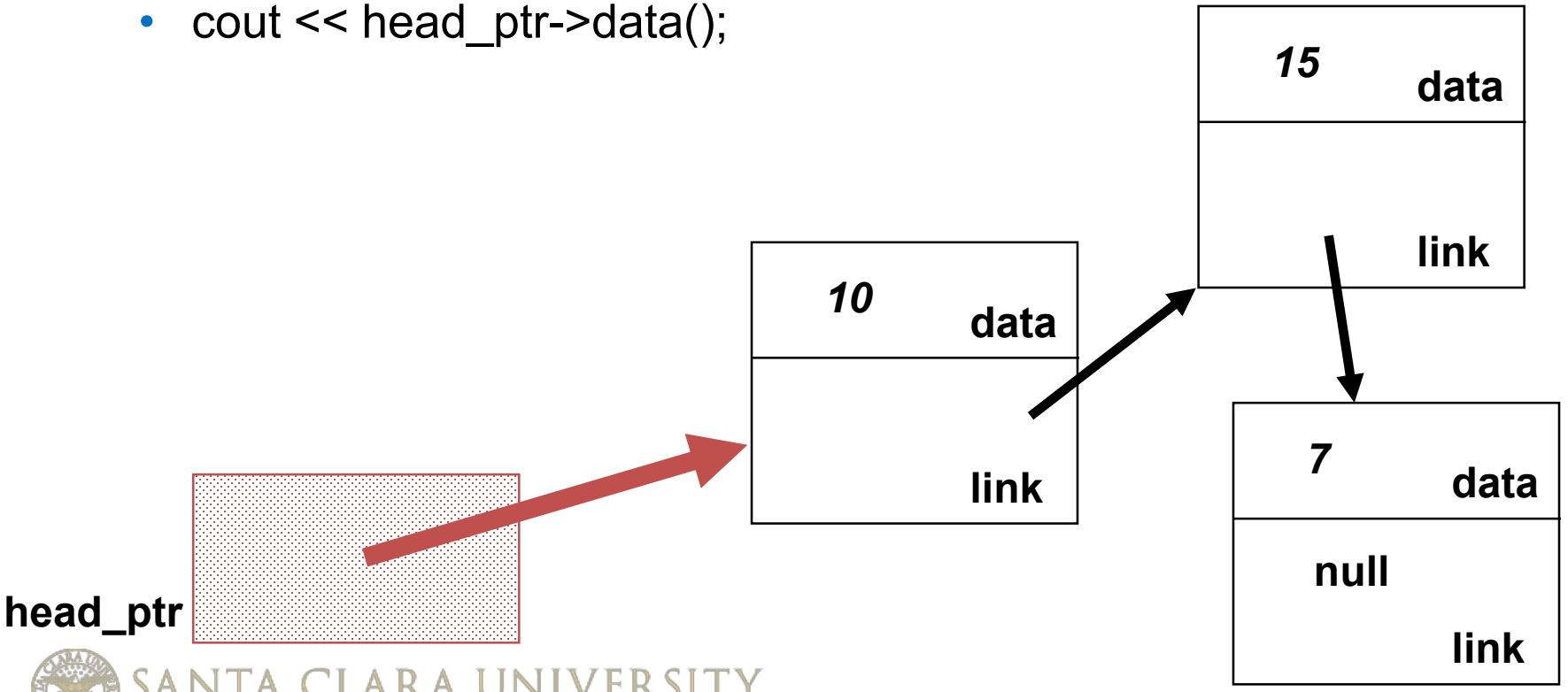
The Complete node Class Definition

- The node class is fundamental to linked lists
- The private member variables
  - data: a value_type variable
  - link: a pointer to the next node
- The member functions include:
  - A constructor
  - Set data and set link
  - Retrieve data and retrieve link

```
class node
{
public:
    // TYPEDEF
    typedef double value_type;

    // CONSTRUCTOR
    node(const value_type& init_data = value_type( ),
            node* init_link = NULL)
            { data = init_data; link = init_link; }

    // Member functions to set the data and link fields:
    void set_data(const value_type& new_data)
                                    { data_field = new_data; }
    void set_link(node* new_link) { link_field = new_link; }

    // Constant member function to retrieve the current data:
    value_type data( ) const { return data_field; }

    // Two slightly different member functions to retrieve
    // the current link:
    const node* link( ) const { return link_field; }
    node* link( )             { return link_field; }

private:
    value_type data_field;
    node* link_field;
};
```

A Small Quiz -

- Suppose a program has built the linked list as shown, and head_ptr is a pointer to a node.
  - What is the data type of *head_ptr?
  - cout << (*head_ptr). data();
  - cout << head_ptr->data();



const node *c;

- The pointer `c` cannot be used to change the node
  - The pointer `c` can move and point to many different nodes
  - Other pointers that point to the same node can change it
  - `c` can activate only constant member functions

```
c->data( )          : ok
c->set_data( )      : error
```

link( )

- Rule for a node's constant member functions
  - A node's constant member function should never provide a result that could later be used to change any part of the linked list

LINKED-LIST TOOLKIT

Linked List Toolkit

- Design Container Classes using Linked Lists
  - The use of a linked list is similar to our previous use of an array in a container class
  - But storing and retrieving needs more work since we do not have handy indexing

=> Linked List Toolbox

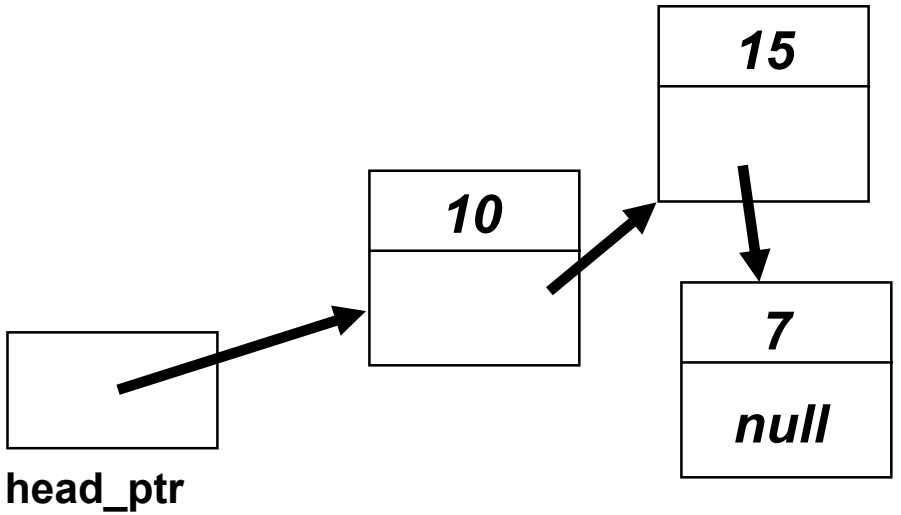
  - using node class

Length of a Linked List

```
size_t list_length(const node* head_ptr);
```

We simply want to compute the **length** of the linked list

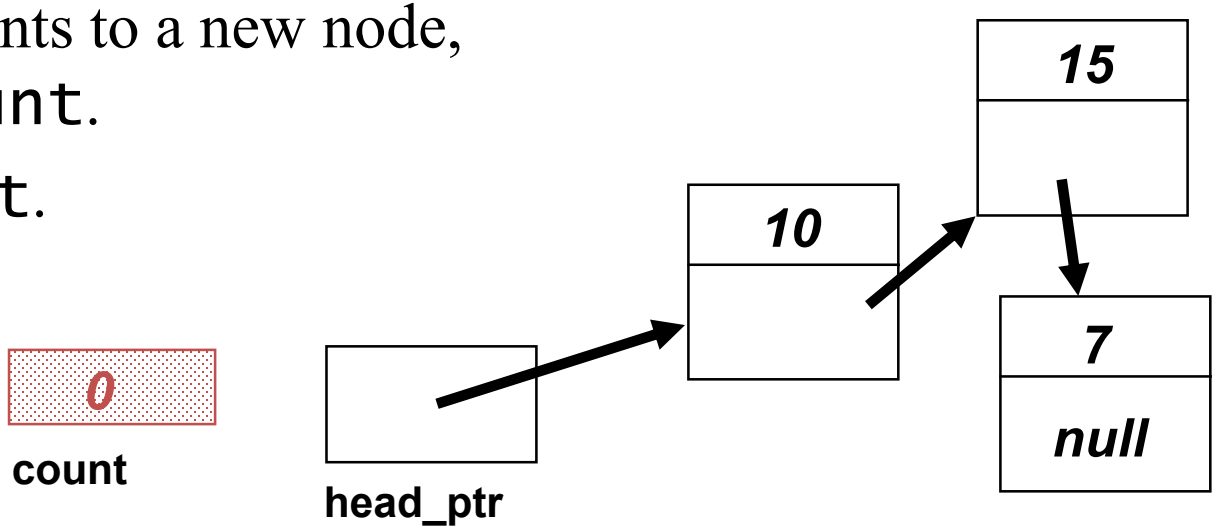
Note that list_length is not a member function of the node class



Pseudo-code of list_length

```
size_t list_length(const node* head_ptr);
```

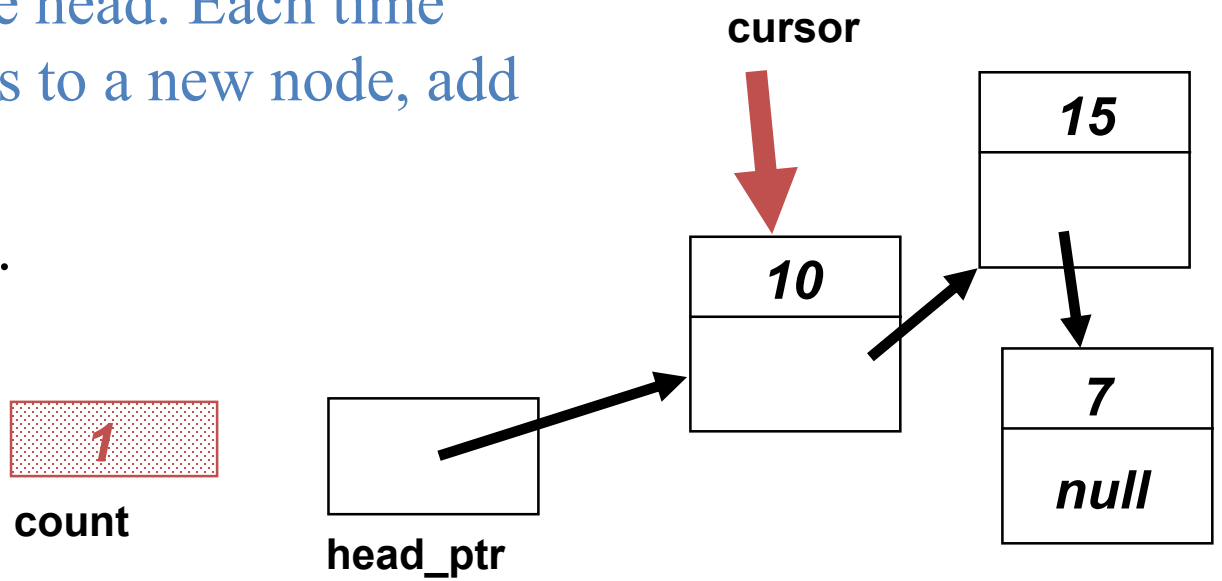
1. Initialize the `count` to zero.
2. Make `cursor` point to each node, starting at the head. Each time `cursor` points to a new node, add 1 to `count`.
3. return `count`.



Pseudo-code of list_length

```
size_t list_length(const node* head_ptr);
```

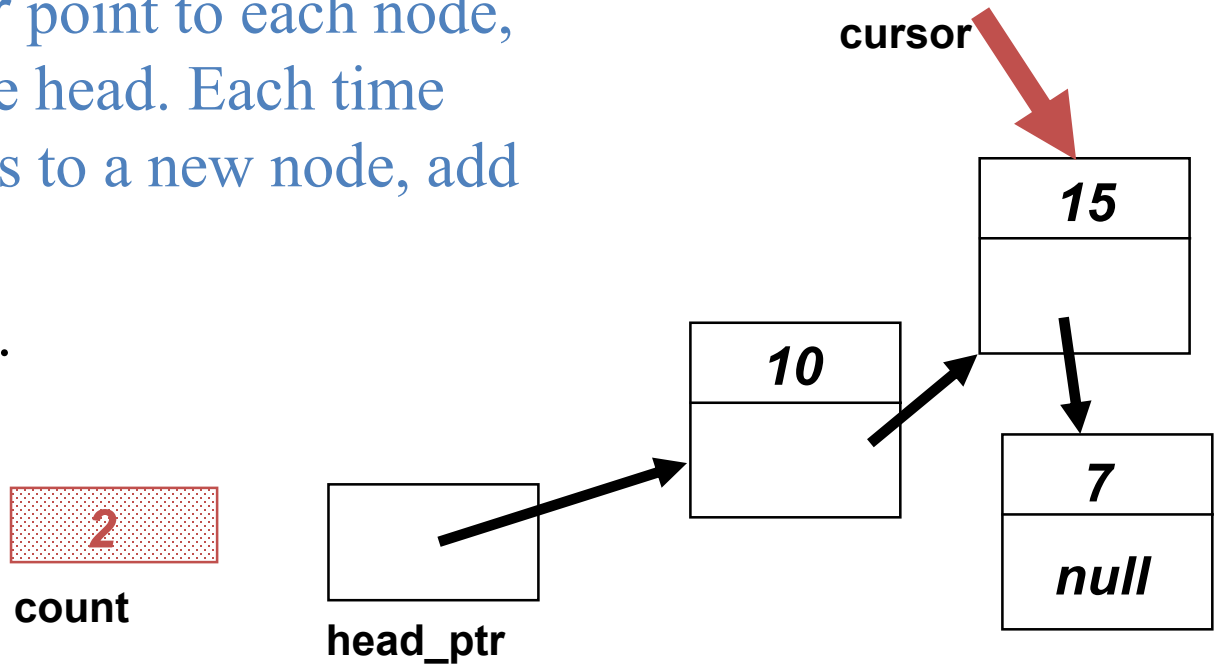
1. Initialize the count to zero.
2. Make cursor point to each node, starting at the head. Each time cursor points to a new node, add 1 to count.
3. return count.



Pseudo-code of list_length

```
size_t list_length(const node* head_ptr);
```

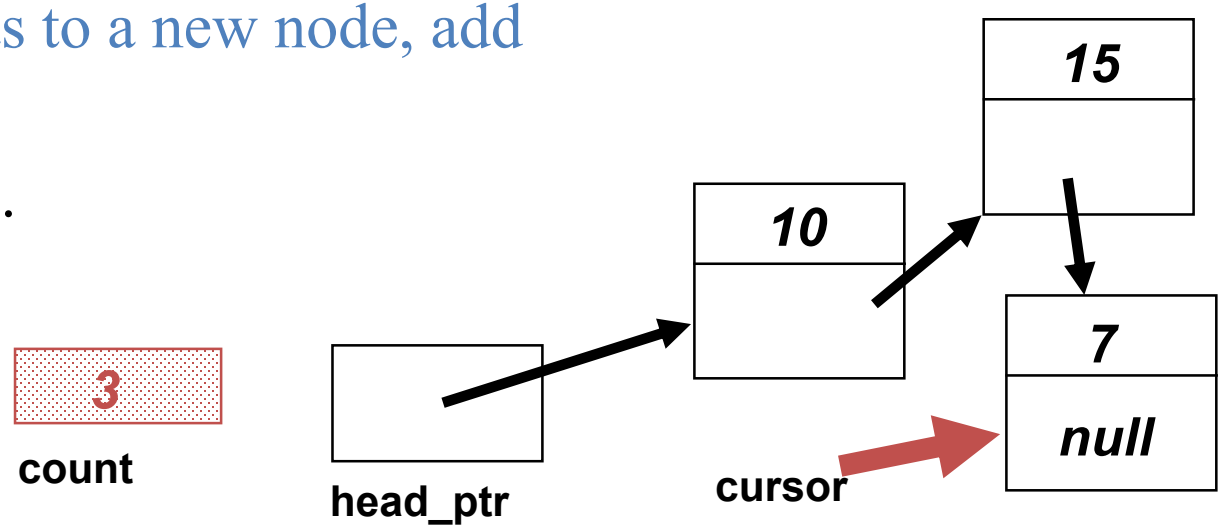
1. Initialize the count to zero.
2. Make cursor point to each node, starting at the head. Each time cursor points to a new node, add 1 to count.
3. return count.



Pseudo-code of list_length

```
size_t list_length(const node* head_ptr);
```

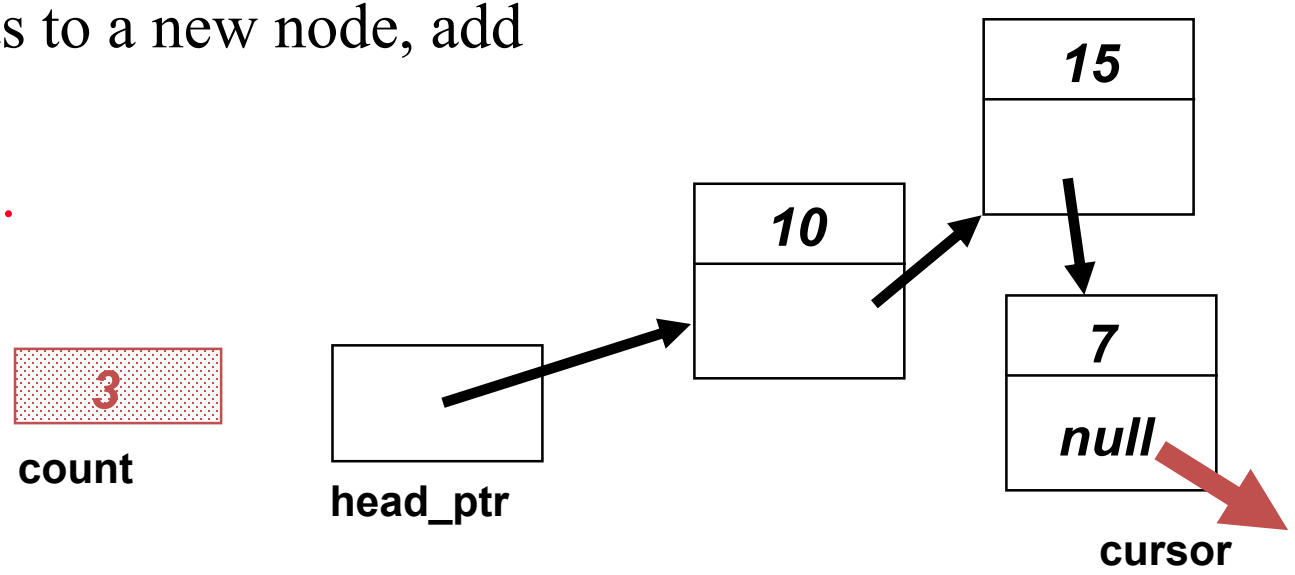
1. Initialize the count to zero.
2. Make cursor point to each node, starting at the head. Each time cursor points to a new node, add 1 to count.
3. return count.



Pseudo-code of list_length

```
size_t list_length(const node* head_ptr);
```

1. Initialize the count to zero.
2. Make cursor point to each node, starting at the head. Each time cursor points to a new node, add 1 to count.
3. return count.



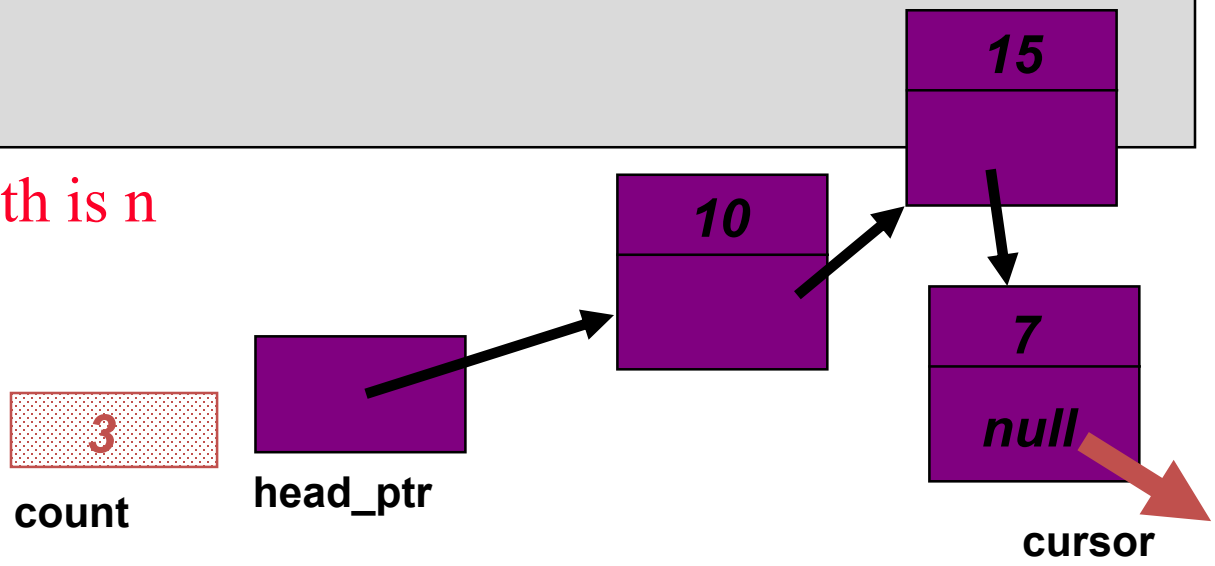
Real code of list_length: List Traverse

```
size_t list_length(const node* head_ptr)
{
    const node *cursor;
    size_t count = 0;
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link())
            count++;
    return count;
}
```

Big-O of list_length

```
size_t list_length(const node* head_ptr)
{
    const node *cursor;
    size_t count = 0;
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link())
            count++;
    return count;
}
```

Big-O: O (n) if length is n

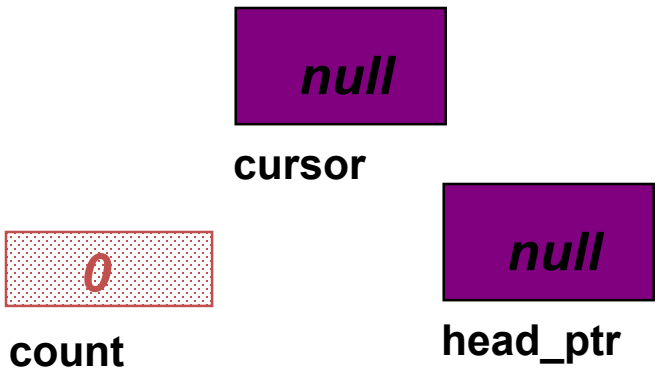


list_length works for an empty list?

```
size_t list_length(const node* head_ptr)
{
    const node *cursor;
    size_t count = 0;
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link())
            count++;
    return count;
}
```

cursor = head_ptr = NULL

count = 0

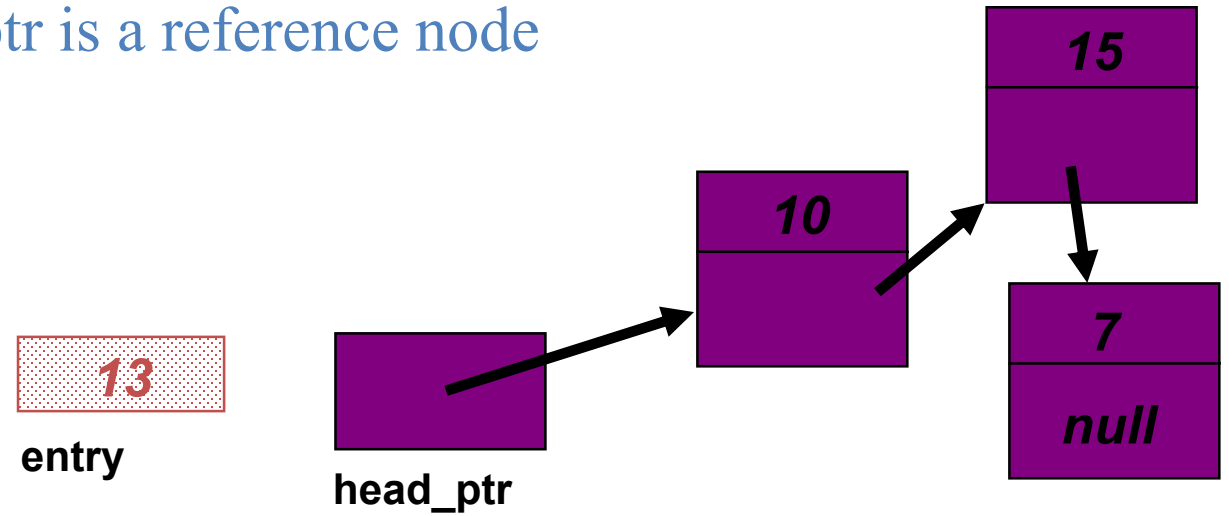


Inserting a node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry);
```

We want to add a new entry, 13, to the **head** of the linked list shown here.

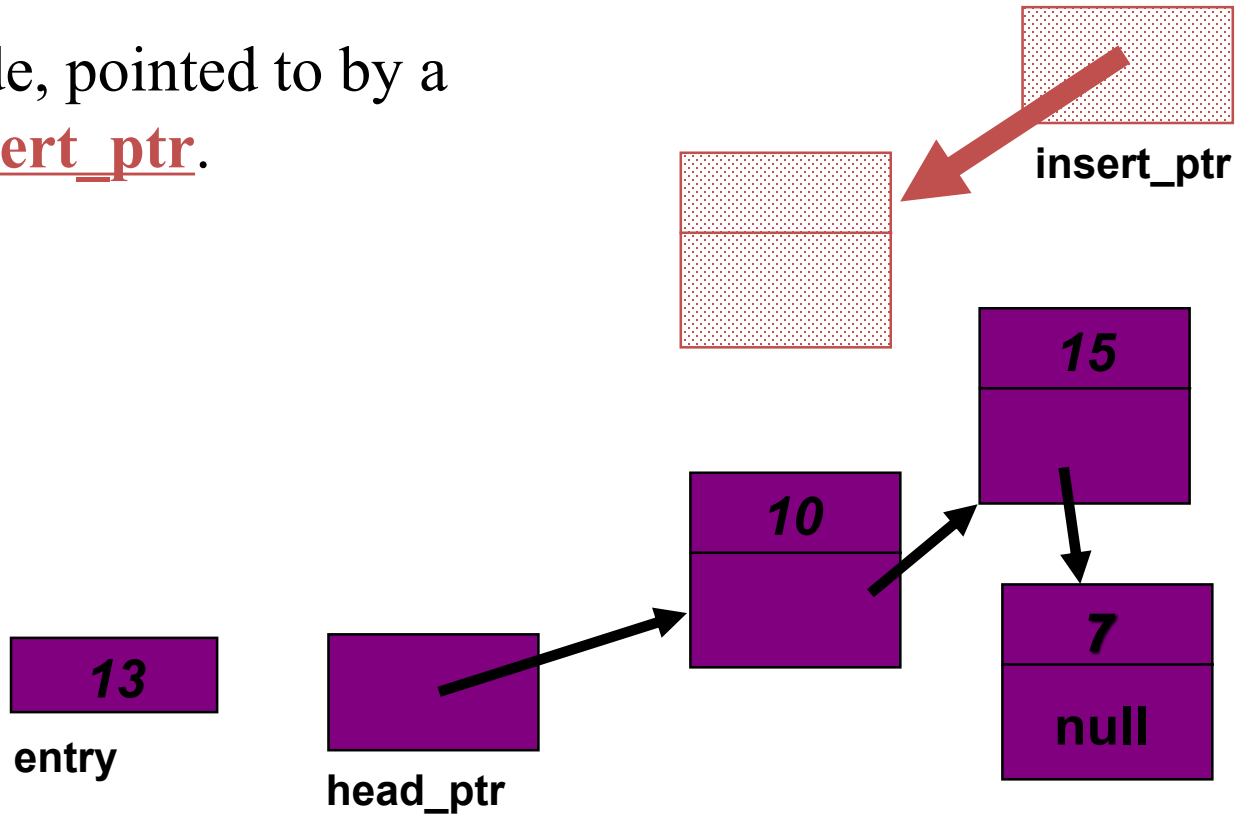
Note that head_ptr is a reference node pointer



Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry);
```

Create a new node, pointed to by a local variable **insert_ptr**.

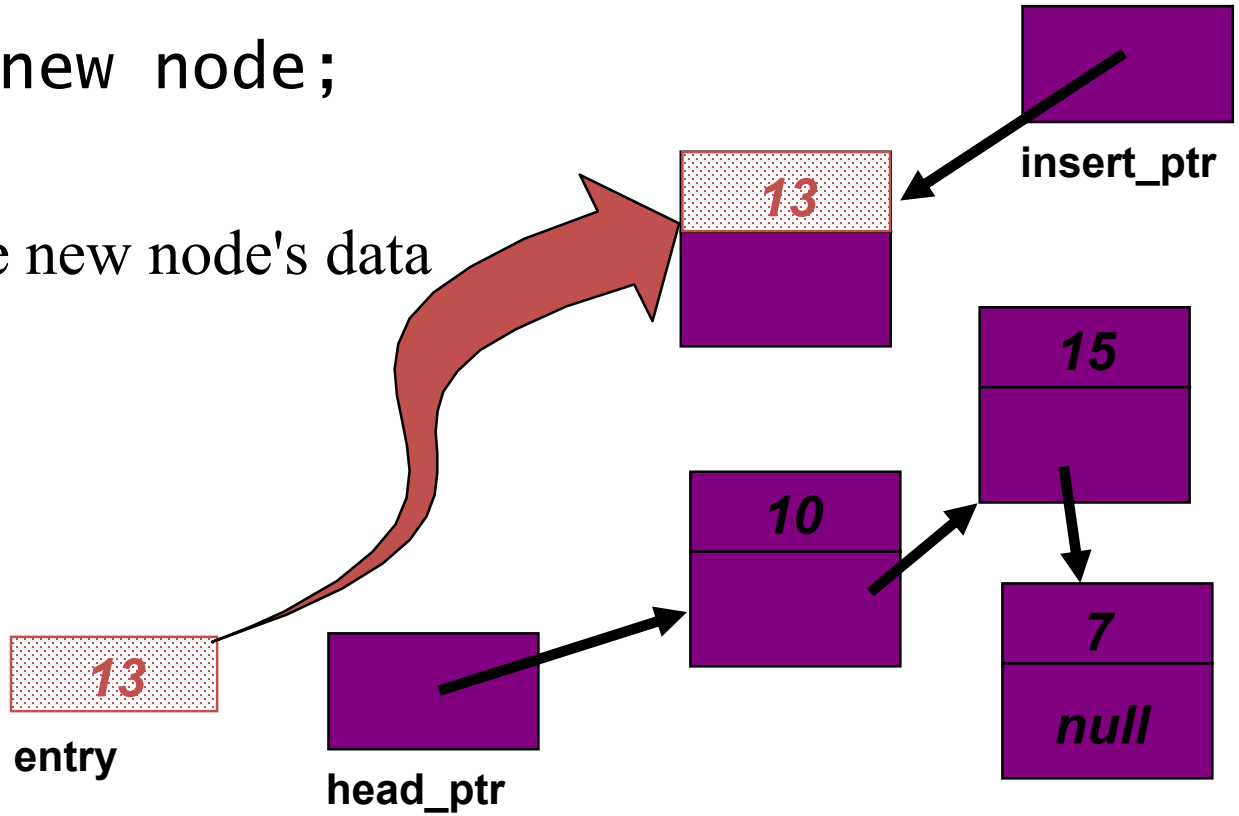


Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry);
```

```
insert_ptr = new node;
```

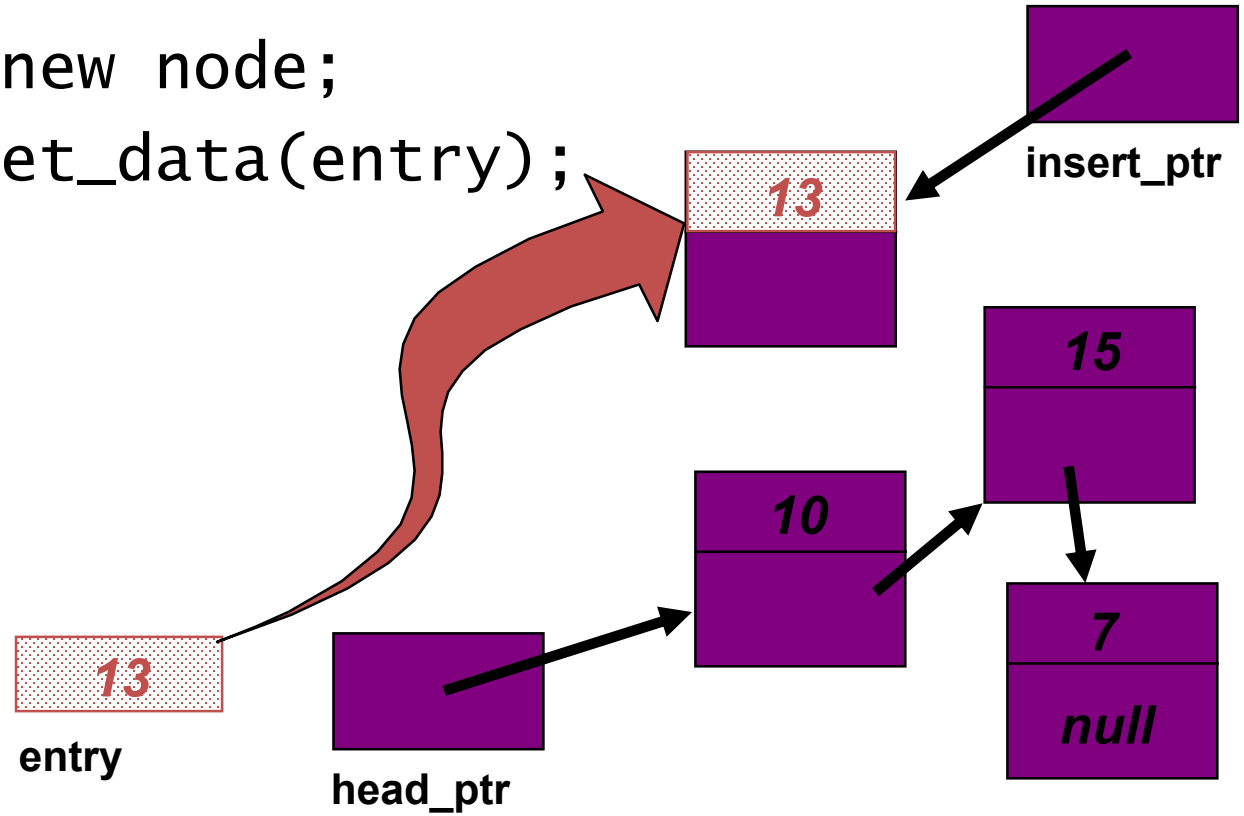
Place the data in the new node's data field.



Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry);
```

```
insert_ptr = new node;
insert_ptr->set_data(entry);
```

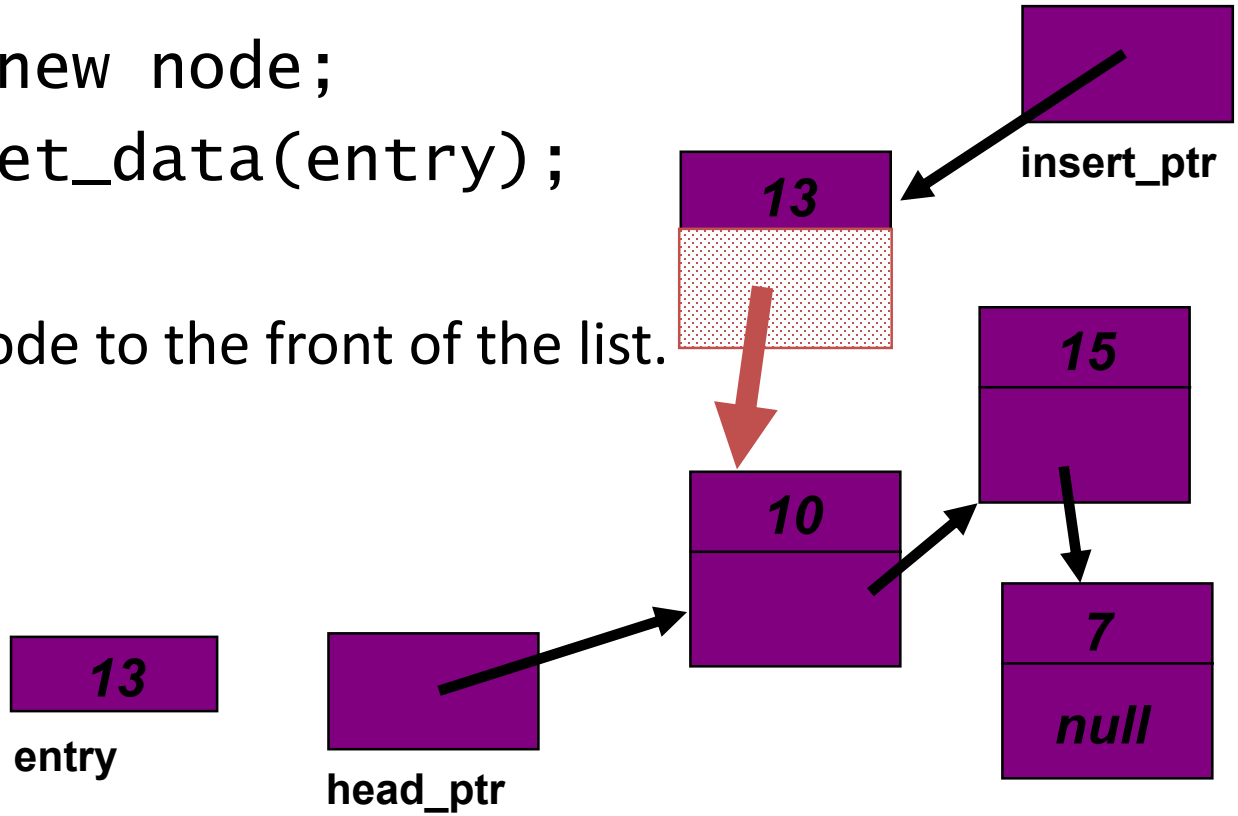


Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry);
```

```
insert_ptr = new node;
insert_ptr->set_data(entry);
```

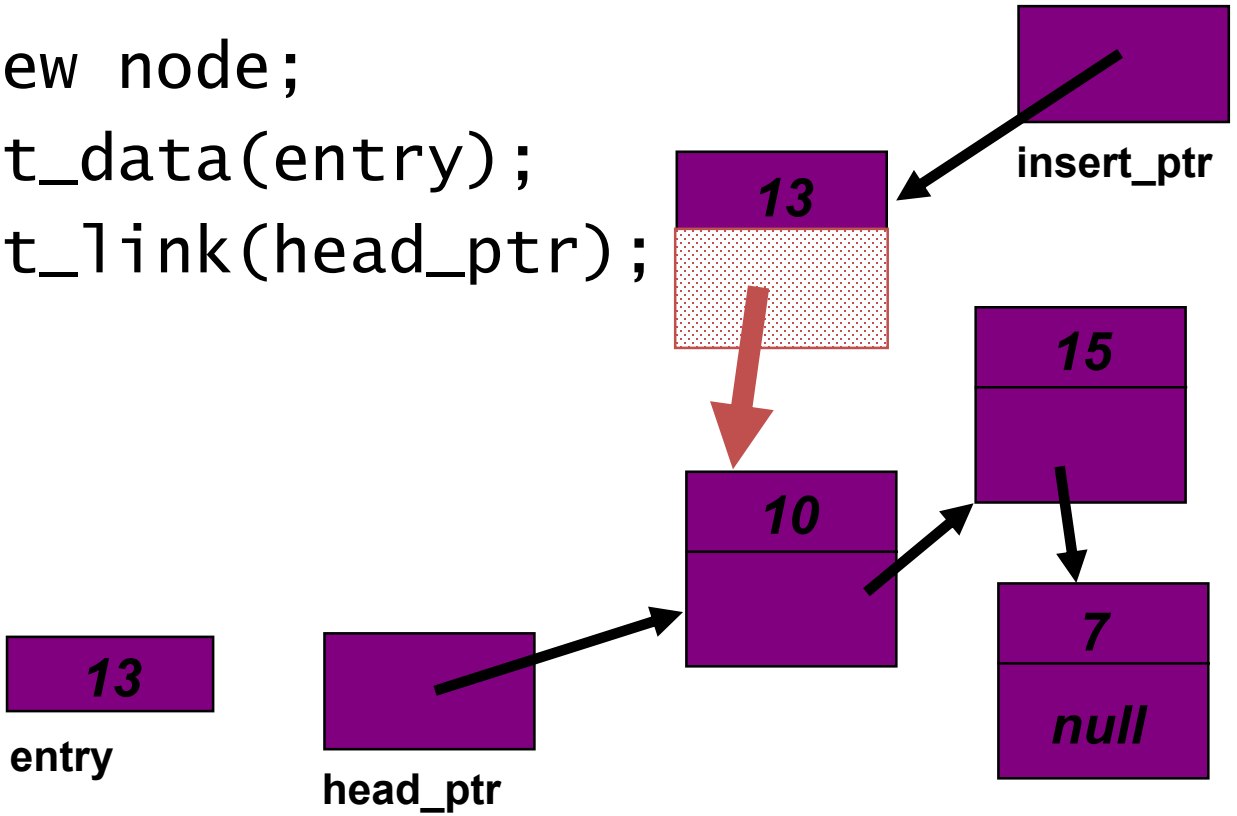
Connect the new node to the front of the list.



Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry);
```

```
insert_ptr = new node;
insert_ptr->set_data(entry);
insert_ptr->set_link(head_ptr);
```

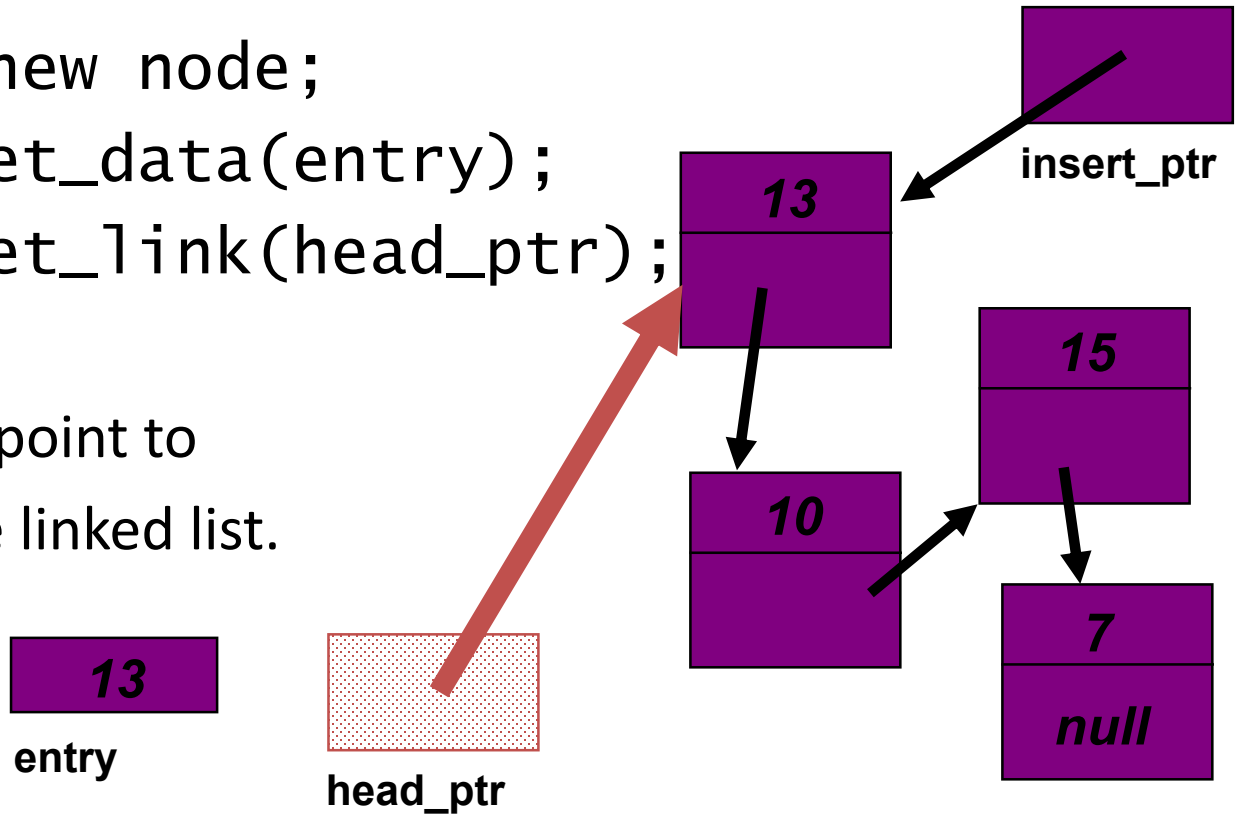


Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry);
```

```
insert_ptr = new node;
insert_ptr->set_data(entry);
insert_ptr->set_link(head_ptr);
```

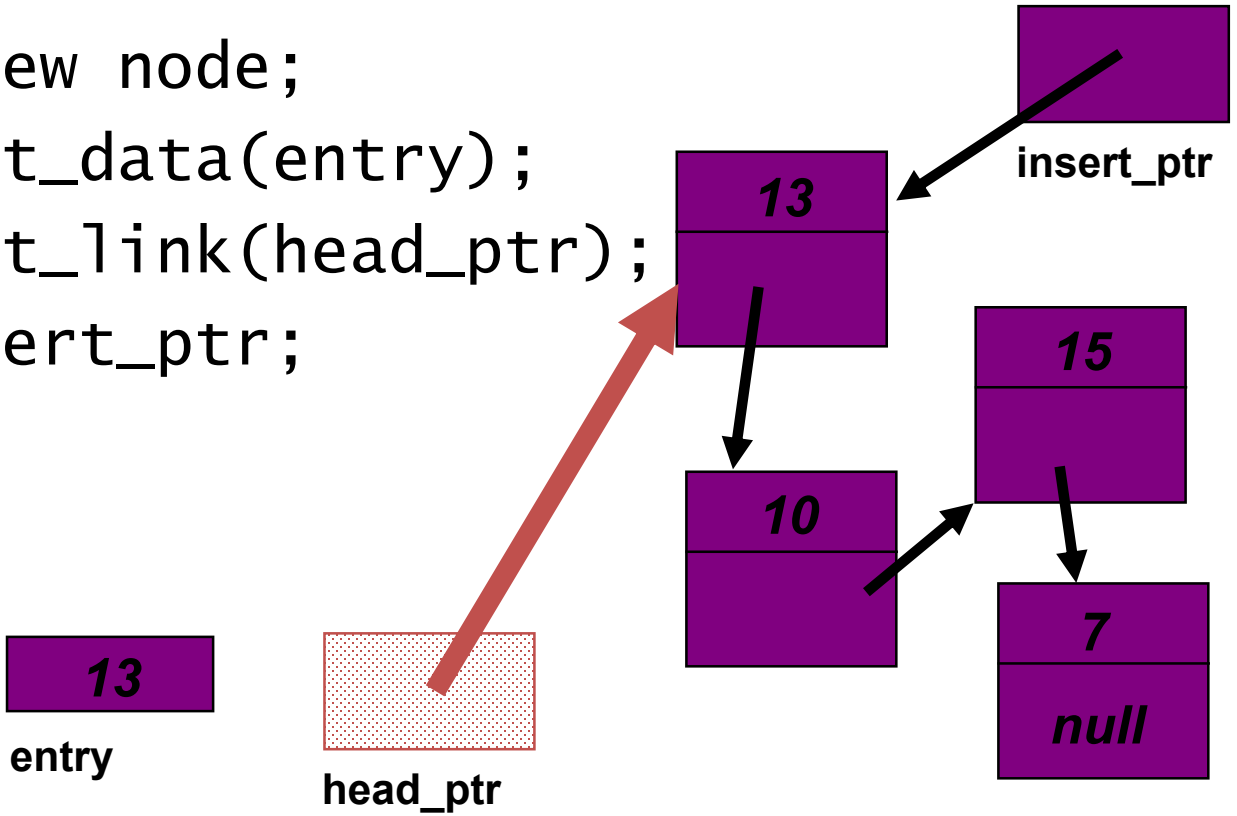
Make the head_ptr point to
the new head of the linked list.



Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry);
```

```
insert_ptr = new node;
insert_ptr->set_data(entry);
insert_ptr->set_link(head_ptr);
head_ptr = insert_ptr;
```

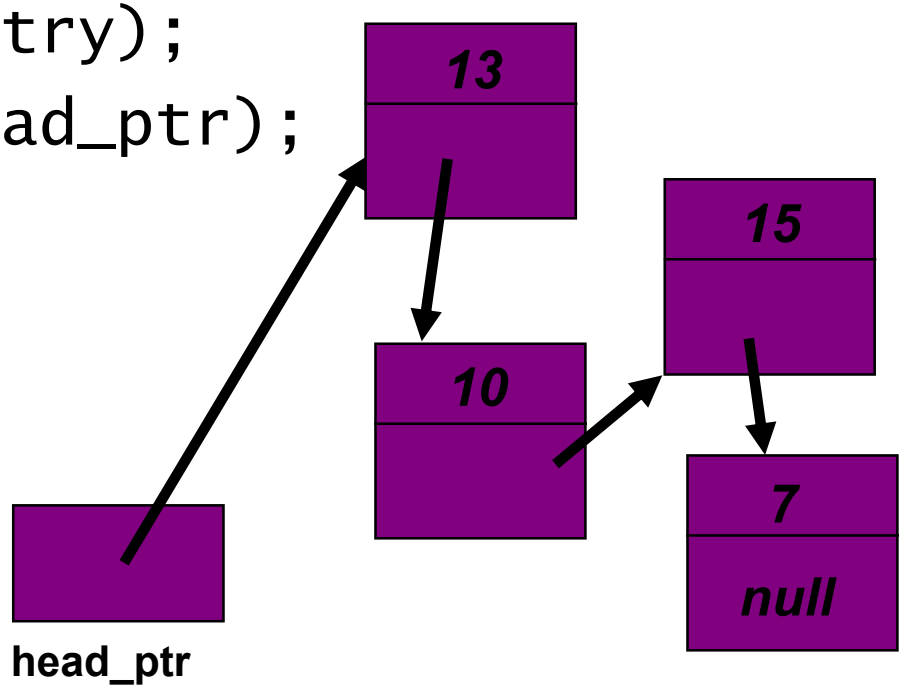


Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry);
```

```
insert_ptr = new node;
insert_ptr->set_data(entry);
insert_ptr->set_link(head_ptr);
head_ptr = insert_ptr;
```

When the function returns, the linked list has a new node at the head, containing 13.



Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry)
{
    node *insert_ptr;

    insert_ptr = new node;
    insert_ptr->set_data(entry);
    insert_ptr->set_link(head_ptr);
    head_ptr = insert_ptr;
}
```

- Big-O?
  - Linked List: O(1)
  - Array: O(n)

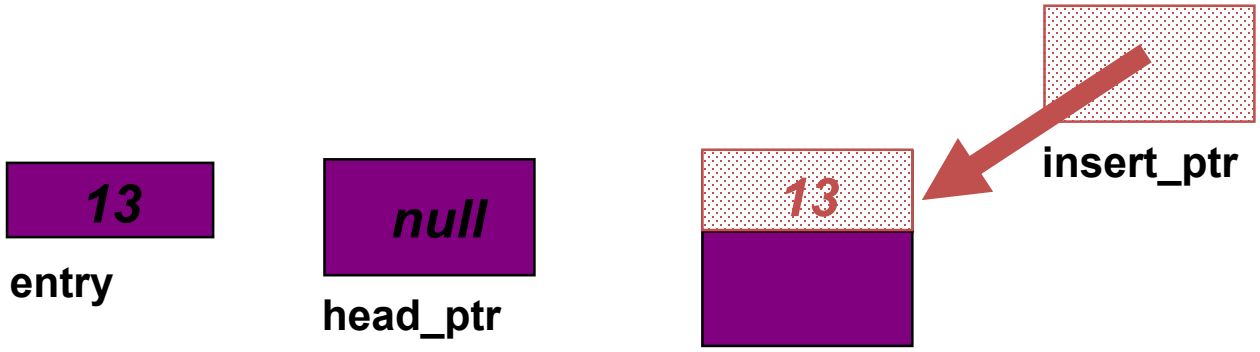
Inserting a Node at the Head

- Does the function work correctly for the empty list ?

```
void list_head_insert(node*& head_ptr, const node::value_type& entry)
{
    node *insert_ptr;

    insert_ptr = new node;
    insert_ptr->set_data(entry);
    insert_ptr->set_link(head_ptr);
    head_ptr = insert_ptr;
}
```

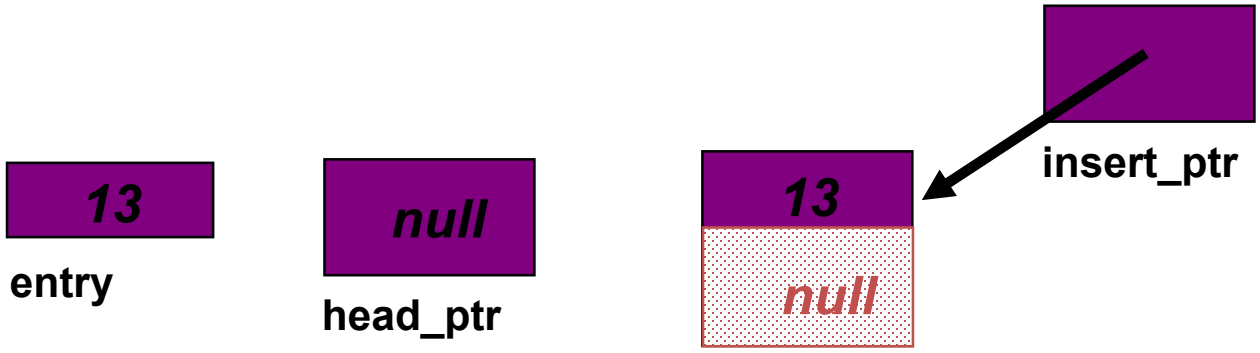
Inserting a Node at the Front

```
void list_head_insert(node*& head_ptr, const node::value_type& entry)
{
    node *insert_ptr;

    insert_ptr = new node;
    insert_ptr->set_data(entry);
    insert_ptr->set_link(head_ptr);
    head_ptr = insert_ptr;
}
```



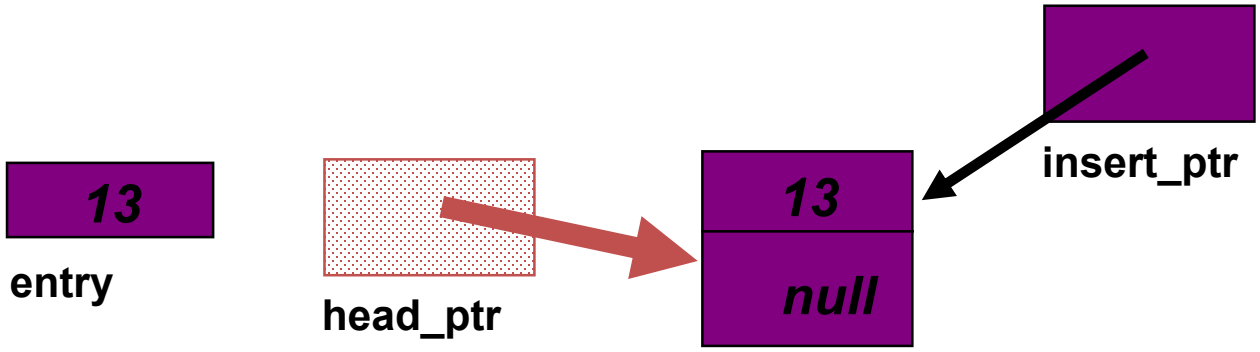
Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry)
{
    node *insert_ptr;

    insert_ptr = new node;
    insert_ptr->set_data(entry);
    insert_ptr->set_link(head_ptr);
    head_ptr = insert_ptr;
}
```



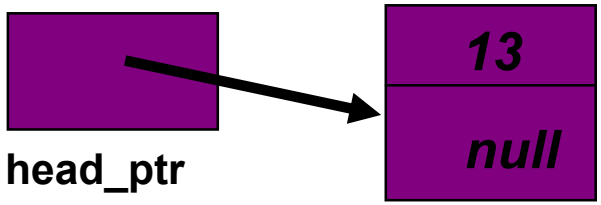
Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry)
{
    node *insert_ptr;

    insert_ptr = new node;
    insert_ptr->set_data(entry);
    insert_ptr->set_link(head_ptr);
    head_ptr = insert_ptr;
}
```



Inserting a Node at the Head

```
void list_head_insert(node*& head_ptr, const node::value_type& entry)
{
    node *insert_ptr;

    insert_ptr = new node;
    insert_ptr->set_data(entry);
    insert_ptr->set_link(head_ptr);
    head_ptr = insert_ptr;
}
```



- Always make sure that your linked list functions work correctly with an empty list

Inserting a Node at the Head

- Can you give an implementation with ONLY a single statement?
- YES, we can use the constructor with parameters!

```
void list_head_insert(node*& head_ptr, const node::value_type& entry)
{
    node *insert_ptr;

    insert_ptr = new node(entry, head_ptr);



    head_ptr = insert_ptr;
}
```

Inserting a Node at the Head

- and assign the return pointer of new directly to the head pointer!

```
void list_head_insert(node*& head_ptr, const node::value_type& entry)
{


    head_ptr = new node(entry, head_ptr);




}
```

Pseudocode for Inserting Nodes

- Nodes are often inserted at places other than the front of a linked list.
- There is a general pseudocode that you can follow for any insertion function. . .

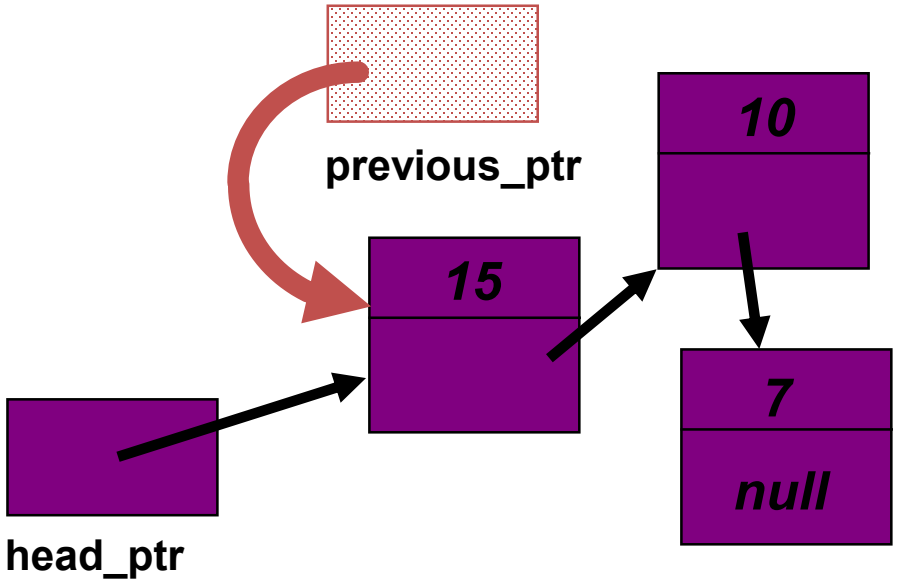
Pseudocode for Inserting Nodes

- Determine whether the new node will be the first node in the linked list. If so, then there is only one step:

```
list_head_insert (head_ptr, entry);
```

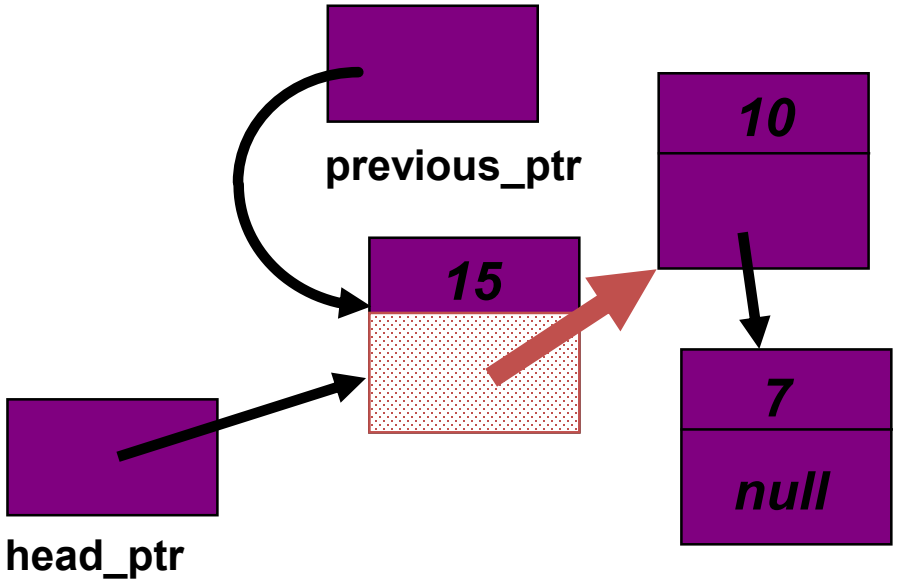
Pseudocode for Inserting Nodes

- Otherwise (if the new node will not be first):
- Start by setting a pointer named **`previous_ptr`** to point to the node which is just **before** the new node's position.
  - In this example, the new node will be the second node



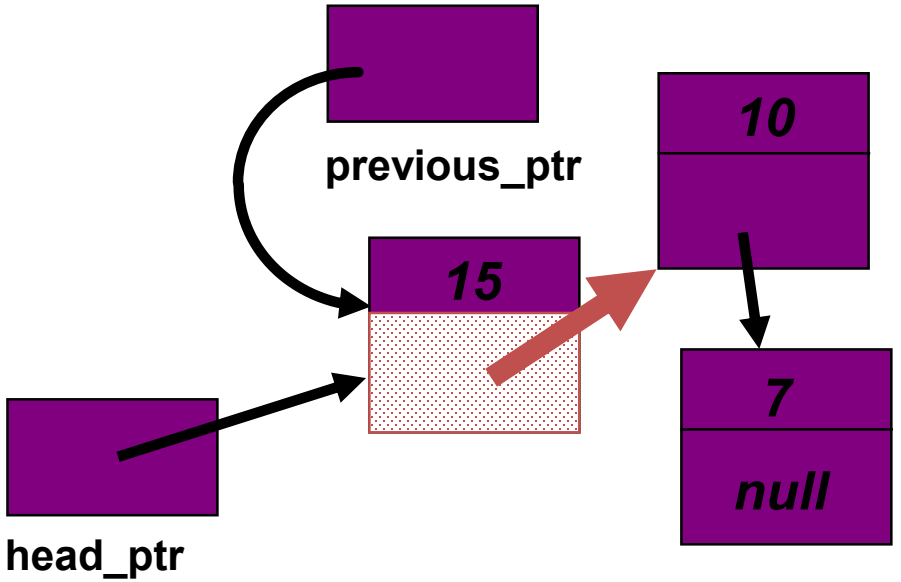
Pseudocode for Inserting Nodes

- Otherwise (if the new node will not be first):
- Start by setting a pointer named **`previous_ptr`** to point to the node which is just **before** the new node's position.
  - In this example, the new node will be the second node
  - Look at the pointer which is in the node *previous_ptr



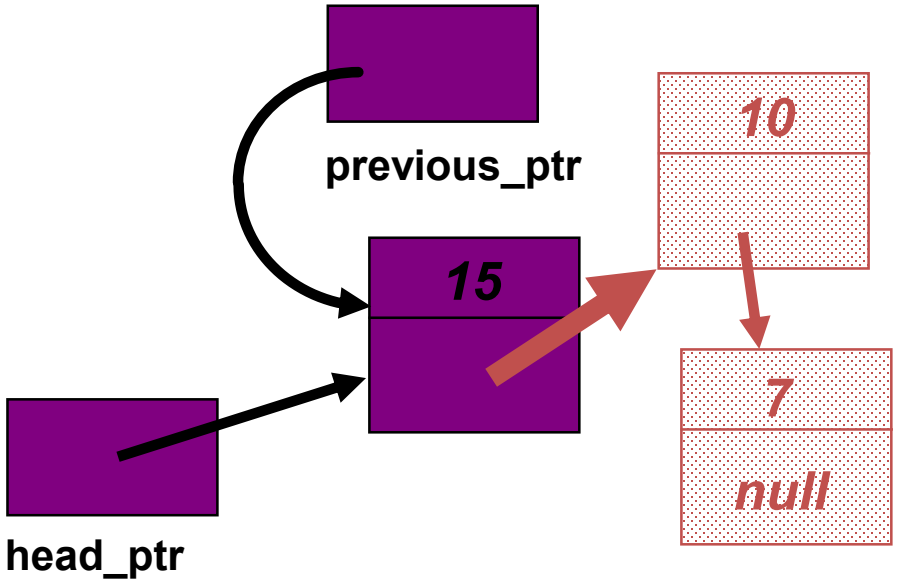
Pseudocode for Inserting Nodes

- Otherwise (if the new node will not be first):
- Start by setting a pointer named **`previous_ptr`** to point to the node which is just **before** the new node's position.
  - In this example, the new node will be the second node
  - Look at the pointer which is in the node *previous_ptr
  - This pointer is called previous_ptr->link



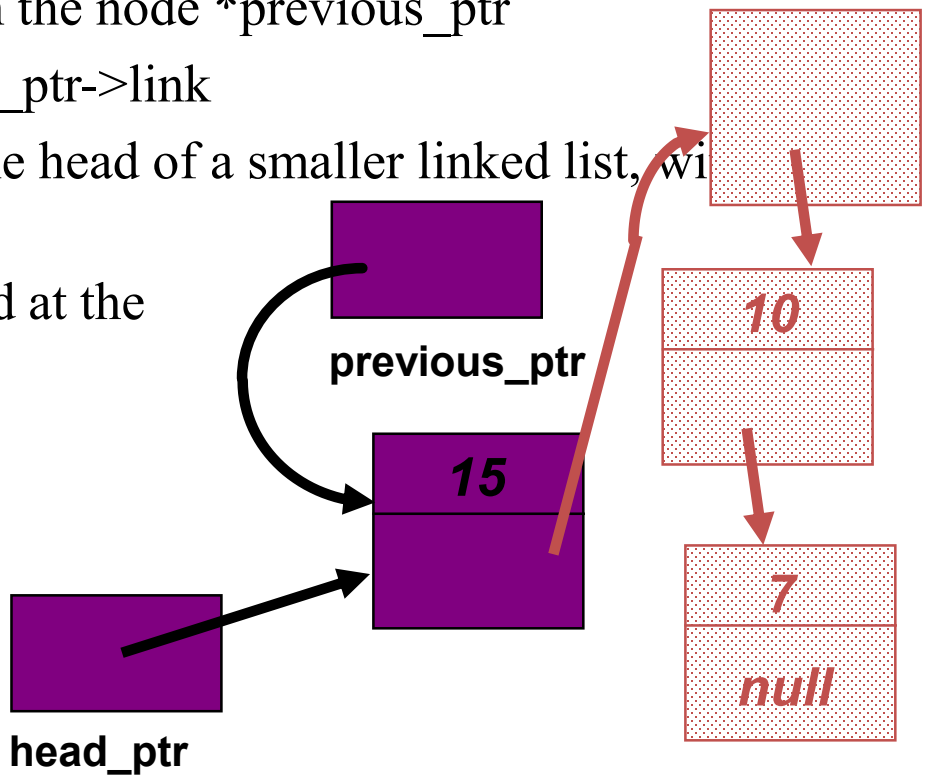
Pseudocode for Inserting Nodes

- Otherwise (if the new node will not be first):
- Start by setting a pointer named **`previous_ptr`** to point to the node which is just **before** the new node's position.
  - In this example, the new node will be the second node
  - Look at the pointer which is in the node *previous_ptr
  - This pointer is called previous_ptr->link
  - previous_ptr->link points to the head of a smaller linked list, with 10 and 7



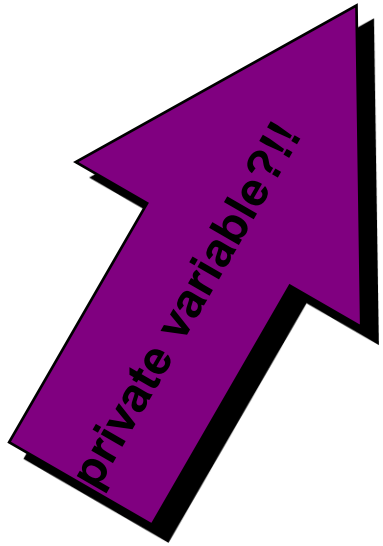
Pseudocode for Inserting Nodes

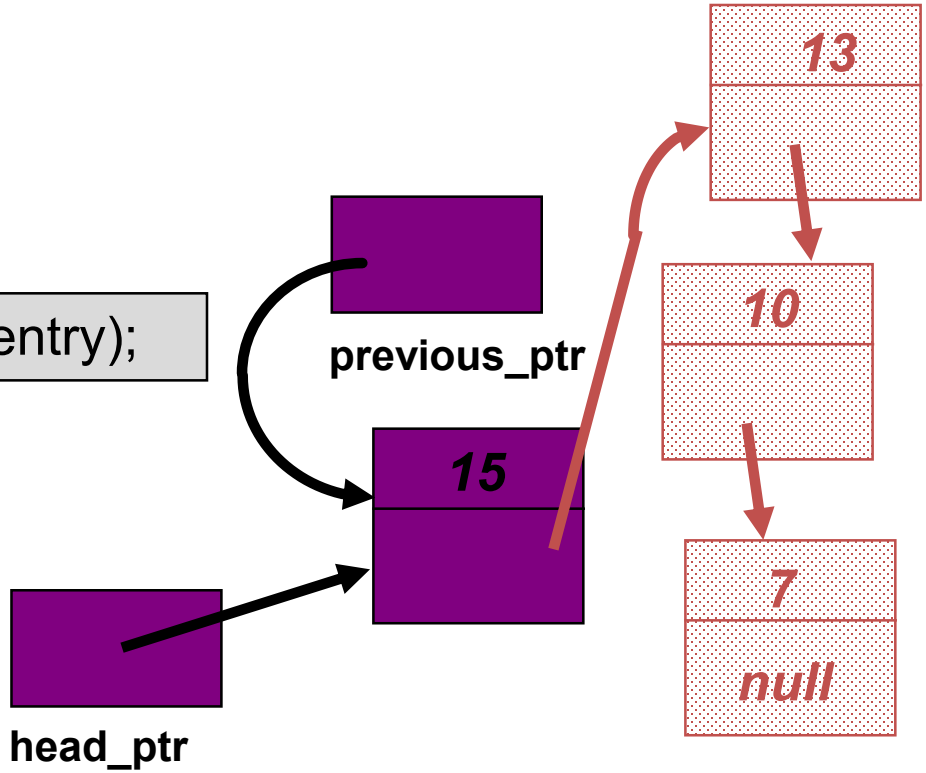
- Otherwise (if the new node will not be first):
- Start by setting a pointer named **`previous_ptr`** to point to the node which is just **before** the new node's position.
  - In this example, the new node will be the second node
  - Look at the pointer which is in the node *previous_ptr
  - This pointer is called previous_ptr->link
  - previous_ptr->link points to the head of a smaller linked list, wi and 7
  - The new node must be inserted at the head of this small linked list.



Pseudocode for Inserting Nodes

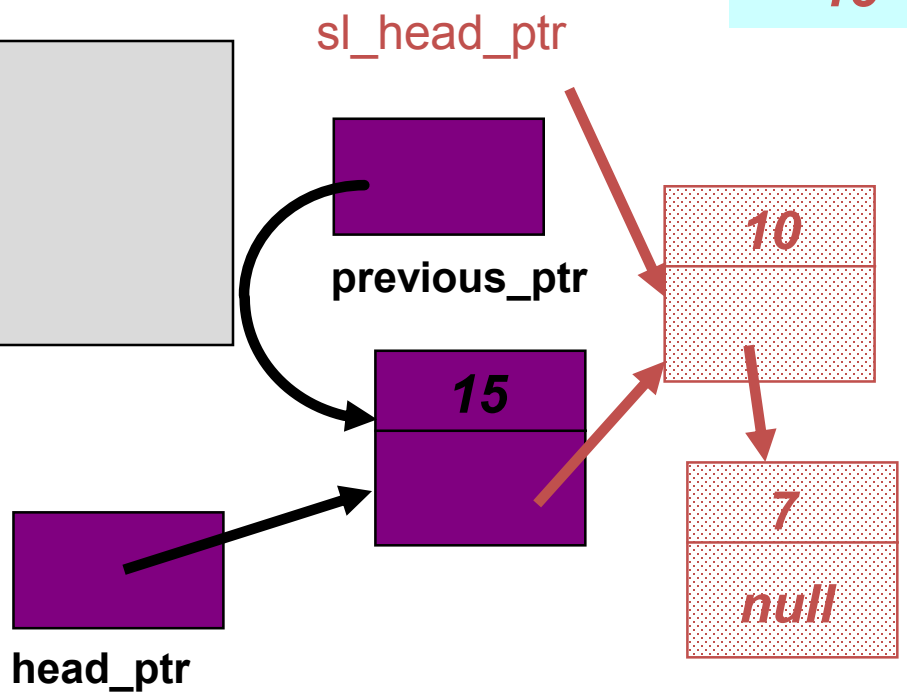
```
list_head_insert(previous_ptr->link, entry);
```





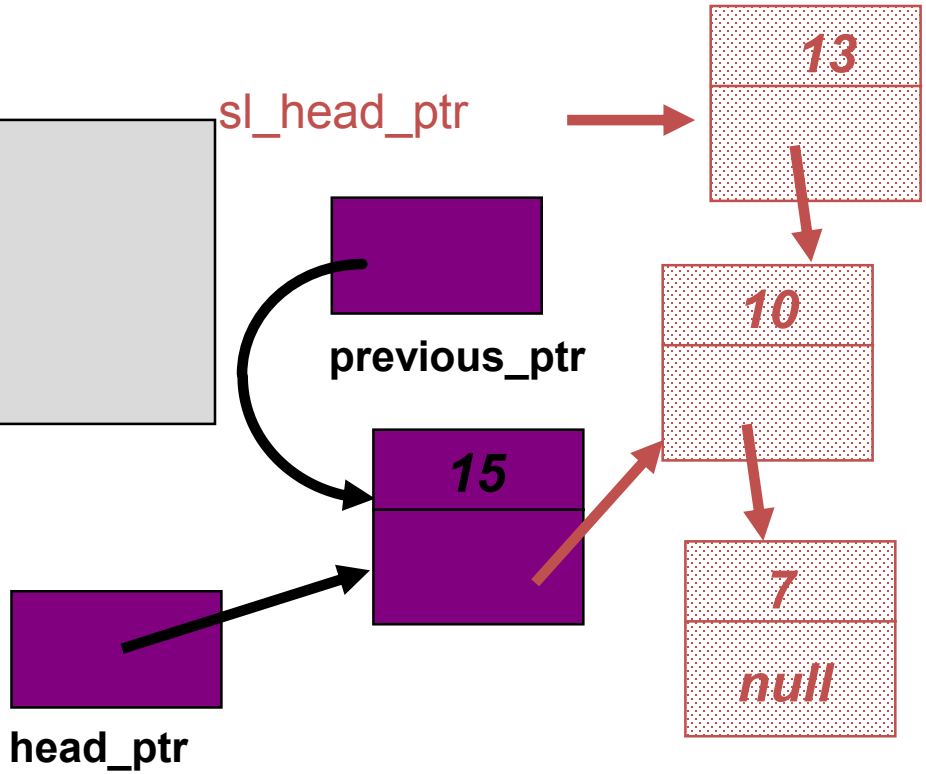
Pseudocode for Inserting Nodes


```
node *sl_head_ptr;
sl_head_ptr = previous_ptr->link();
list_head_insert(sl_head_ptr, entry);
previous_ptr->set_link(sl_head_ptr);
```



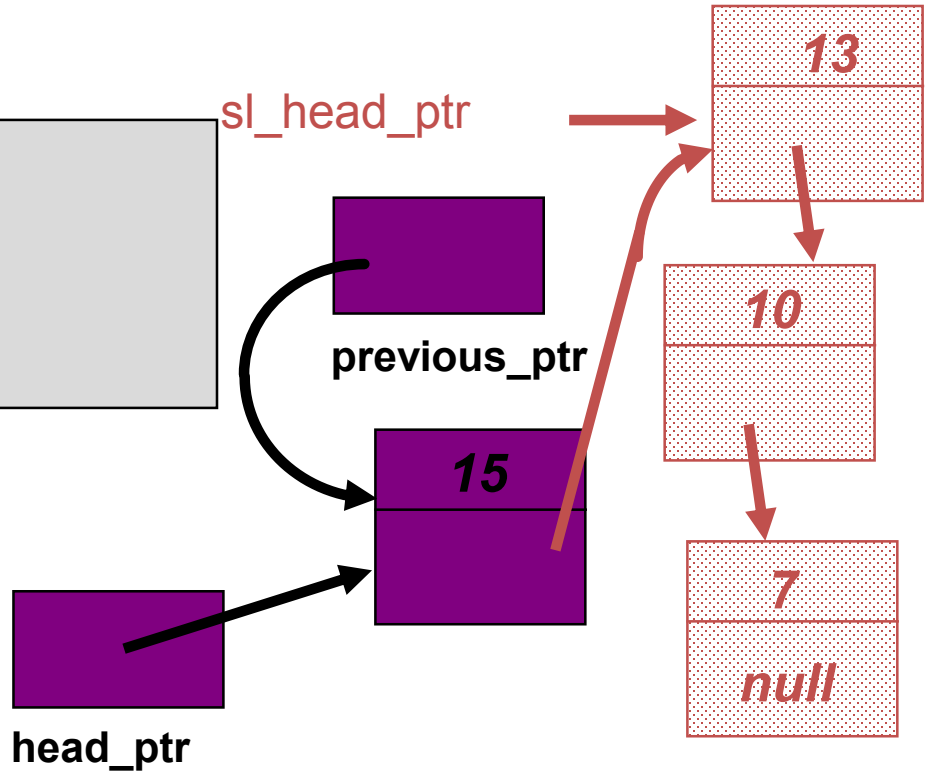
Pseudocode for Inserting Nodes

```
node *sl_head_ptr;
sl_head_ptr = previous_ptr->link();
list_head_insert(sl_head_ptr, entry);
previous_ptr->set_link(sl_head_ptr);
```



Pseudocode for Inserting Nodes

```
node *sl_head_ptr;
sl_head_ptr = previous_ptr->link();
list_head_insert(sl_head_ptr, entry);
previous_ptr->set_link(sl_head_ptr);
```



Pseudocode for Inserting Nodes

- Determine whether the new node will be the first node in the linked list. If so, then there is only one step:

```
list_head_insert(head_ptr, entry);
```

- Otherwise (if the new node will not be first):
  - Set a pointer named previous_ptr to point to the node which is just before the new node's position.

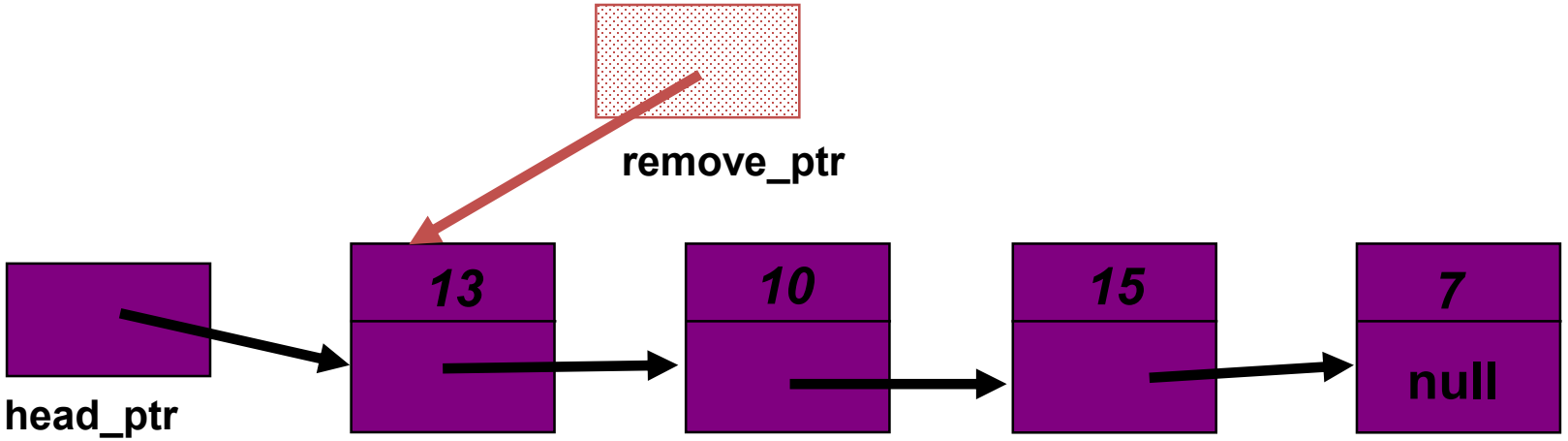
```
node *sl_head_ptr;
sl_head_ptr = previous_ptr->link();
list_head_insert(sl_head_ptr, entry);
previous_ptr->set_link(sl_head_ptr);
```

Pseudocode for Removing Nodes

- Nodes often need to be removed from a linked list.
- As with insertion, there is a technique for removing a node from the front of a list, and a technique for removing a node from elsewhere.
- We'll look at the pseudocode for removing a node from the head of a linked list.

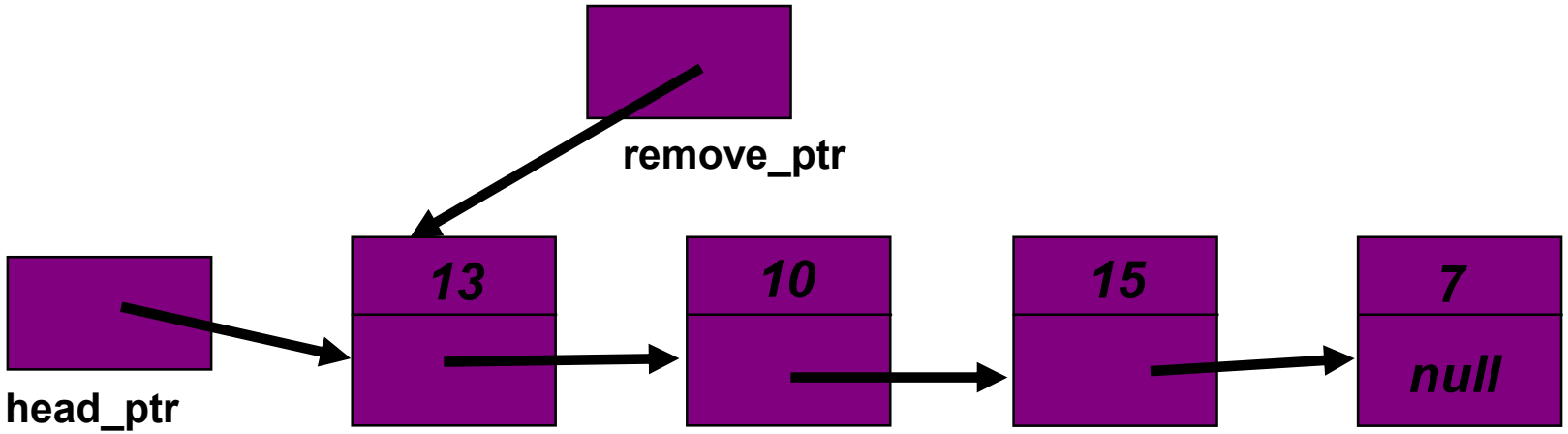
Removing the Head Node

- Start by setting up a temporary pointer named `remove_ptr` to the head node.



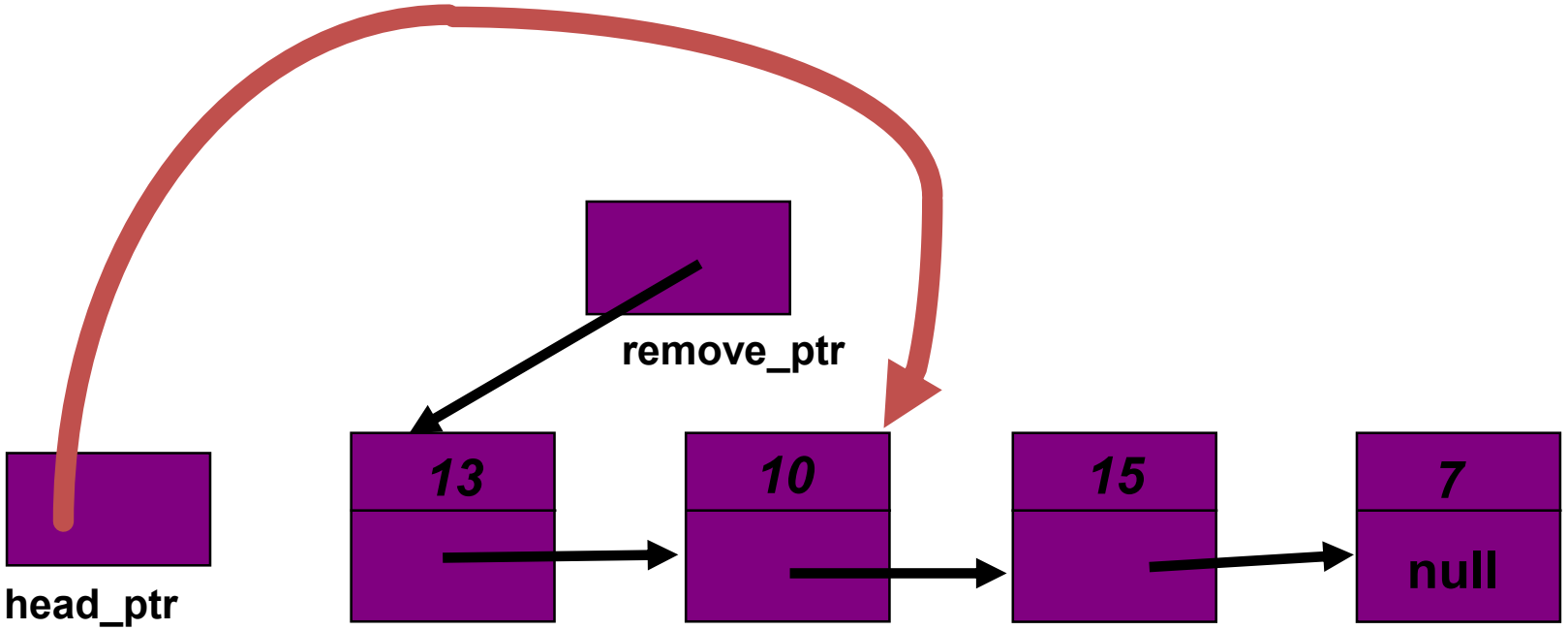
Removing the Head Node

- Set up remove_ptr.
- head_ptr = remove_ptr->link();



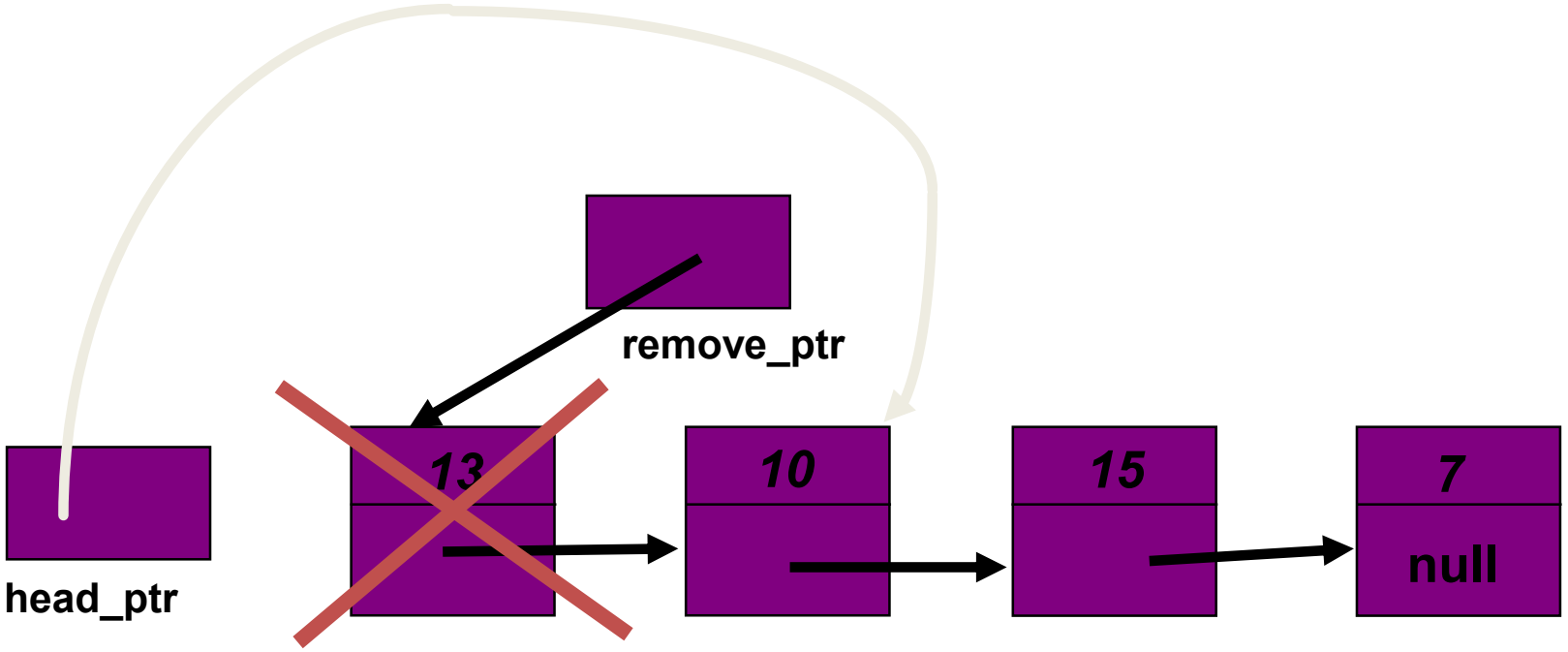
Removing the Head Node

- Set up remove_ptr.
- head_ptr = remove_ptr->link();



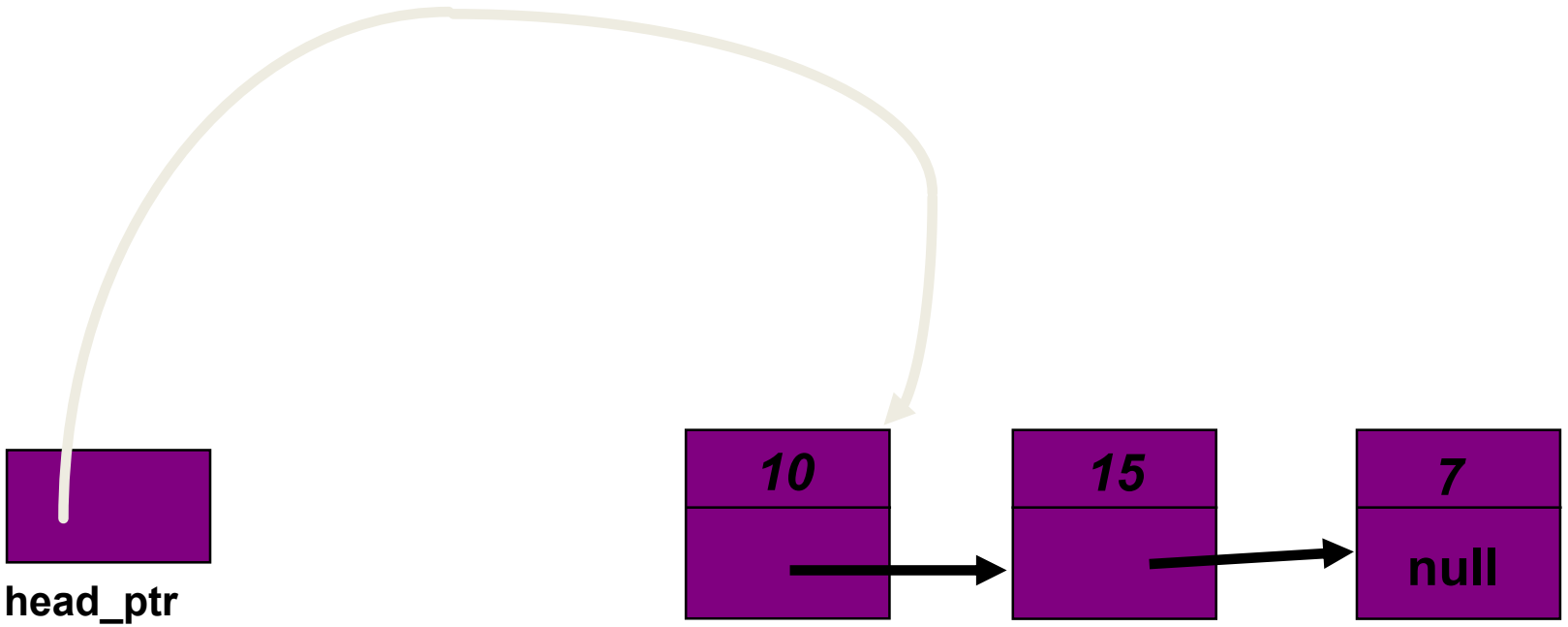
Removing the Head Node

- Set up remove_ptr.
- head_ptr = remove_ptr->link;
- delete remove_ptr; // Return the node's memory to heap.



Removing the Head Node

Here's what the linked list looks like after the removal finishes.



```
// FUNCTIONS for the linked list toolkit
  std::size_t list_length(const node* head_ptr);
  void list_head_insert(node*& head_ptr, const node::value_type& entry);
  void list_insert(node* previous_ptr, const node::value_type& entry);
  node* list_search(node* head_ptr, const node::value_type& target);
  const node* list_search(const node* head_ptr, const node::value_type& target);
  node* list_locate(node* head_ptr, std::size_t position);
  const node* list_locate(const node* head_ptr, std::size_t position);
  void list_head_remove(node*& head_ptr);
  void list_remove(node* previous_ptr);
  void list_clear(node*& head_ptr);
  void list_copy(const node* source_ptr, node*& head_ptr, node*& tail_ptr);
```

Key points you need to know

- Linked List Toolkit uses the node class which has
  - set and retrieve functions
- The functions in the Toolkit are not member functions of the node class
  - length, insert(2), remove(2), search, locate, copy,...
  - compare their Big-Os with similar functions for an array
- They can be used in various container classes, such as bag, sequence, etc.

THE BAG CLASS WITH A LINKED LIST

Our Third Bag — Specification

- Our new bag vs the old bag
  - There is no default capacity and no need for a reserve function that reserves a specified capacity

- Storing the bag's items in a linked list enables us to easily grow and shrink the list by adding and removing nodes from the linked list
- Of course, the programmer who uses the new bag class does not need to know about linked lists
- The documentation of our new header file will mention the use of linked lists

Our Third Bag — Class Definition

```
#include "node1.h"

class bag
{

public:

    // TYPEDEFS
    typedef node::value_type value_type;
        . . .

private:
    node *head_ptr;          // List head pointer
    size_type many_nodes;    // Number of nodes on the list
};
```

bag's value_type Match node's value_type

Our Third Bag — Header File Implementation

```
#ifndef SCU_COEN70_BAG3_H
#define SCU_COEN70_BAG3_H
#include <cstdlib>    // Provides size_t and NULL
#include "node1.h"    // Provides node class

namespace scu_coen70_5
{
    class bag
    {
    public:
        // TYPEDEFS
        typedef std::size_t size_type;
        typedef node::value_type value_type;

        // CONSTRUCTORS and DESTRUCTOR
        bag( );
        bag(const bag& source);
        ~bag( );
```

copy constructor

we can't use the former bag constructor that we have because this constructor is initializing data_field and link_field

Our Third Bag — Header File (cont.)

```
// MODIFICATION MEMBER FUNCTIONS
size_type erase(const value_type& target);
bool erase_one(const value_type& target);
void insert(const value_type& entry);
void operator +=(const bag& addend);
void operator =(const bag& source);

// CONSTANT MEMBER FUNCTIONS
size_type size( ) const { return many_nodes; }
size_type count(const value_type& target) const;
```

Our Third Bag — Header File (cont.)

```
    private:
        // List head pointer
        node *head_ptr;

        // Number of nodes on the list
        size_type many_nodes;
    };

    // NONMEMBER FUNCTIONS for the bag class:
    bag operator +(const bag& b1, const bag& b2);
}
#endif
```

Rules for Dynamic Memory Usage in a Class

- Some of the member variables of the class are pointers
- Member functions allocate and release dynamic memory as needed
- The automatic value semantics of the class is overridden
  - i.e., **The class must implement a copy constructor and an assignment operator that correctly copy one bag to another.**
- The class has a destructor

The Third Bag Class — Implementation

- Constructors
  - The **default constructor** sets `head_ptr` to be the null pointer and sets `many_nodes` to zero
  - The **copy constructor** uses `list_copy( )` to make a separate copy of the source list

```
bag::bag( )
 {
   head_ptr = NULL;
   many_nodes = 0;
 }

bag::bag(const bag& source)
 {
   node *tail_ptr;  // Needed for argument of list_copy

   list_copy(source.head_ptr, head_ptr, tail_ptr);
   many_nodes = source.many_nodes;
 }
```

The Third Bag Class — Implementation (cont.)

- Assignment operator overloading

```
void bag::operator =(const bag& source)
{
     node *tail_ptr; // Needed for argument to list_copy

     if (this == &source)
         return;

     list_clear(head_ptr);
     many_nodes = 0;
     list_copy(source.head_ptr, head_ptr, tail_ptr);
     many_nodes = source.many_nodes;

}
```

The Third Bag Class — Implementation (cont.)

- The destructor
  - Needed because our implementation uses dynamic memory
- Use `list_clear( )` to return all dynamic memory to the heap

```
bag::~bag( )
{
    list_clear(head_ptr);
    many_nodes = 0;
}
```

- Note: The second statement, `many_nodes = 0`, is not necessary, since the bag is not supposed to be used after the destructor has been called

The Third Bag Class — Implementation (cont.)

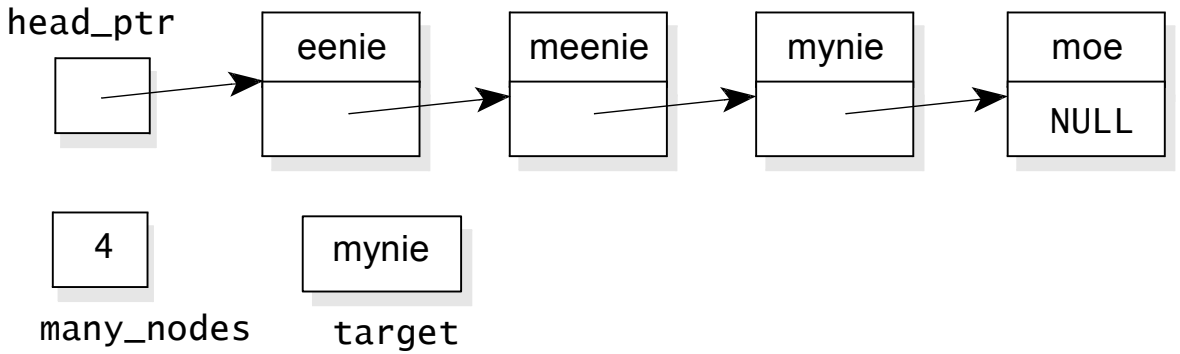
- The erase_one member function.
  - There are two approaches to implement this function:
  - Using the toolkit's removal function:
    - ✓list_head_remove to remove an item at the head of the list
    - ✓list_remove to remove an item that is farther down the line

    Note: list_remove requires a pointer to the node that is just before the item that you want to remove (Note: we cannot use list_search to find this item)
  - Using list_search to obtain a pointer to the node that contains the item to be deleted
  - We chose this approach because it made better use of list_search

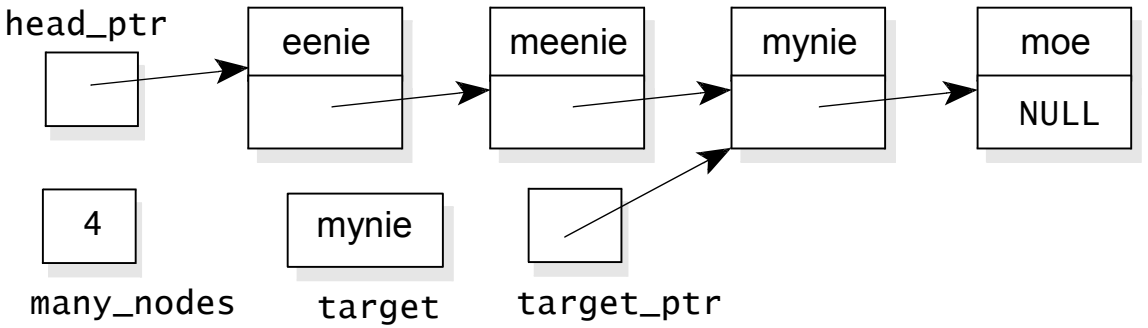
The Third Bag Class — Implementation (cont.)

- The erase_one member function using list_search

Suppose our target is the string `mynie`

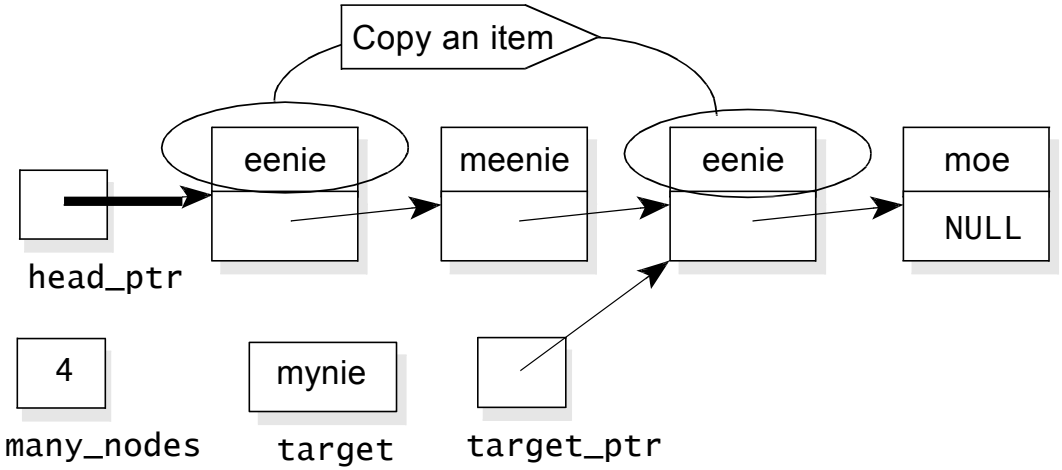


- `target_ptr` points to the node that contains our target
- `target_ptr = list_search(head_ptr, target)`



The Third Bag Class — Implementation (cont.)

- Copy the data from the head node to the target node, as shown here:



- After this step, we have certainly removed the target, but we are left with two `eenies`, so, we proceed to a second step:
- Use `list_head_remove` to remove the head node

The Third Bag Class — Implementation (cont.)

- erase_one function implementation

```
bool bag::erase_one(const value_type& target)
{
    node *target_ptr;

    target_ptr = list_search(head_ptr, target);
    if (target_ptr == NULL)
        return false;      // target is not in the bag

    target_ptr->set_data( head_ptr->data( ) );
    list_head_remove(head_ptr);
    --many_nodes;
    return true;
}
```

Three steps:

1. **Use `list_search` to obtain a pointer to the node that should be deleted**
2. **Copy data from the head node to the target node**
3. **Use `list_head_remove` to remove the head node**

The Third Bag Class — Implementation (cont.)

- The count member function.
  - Counts the occurrences of the target and returns the answer

- Two possible approaches:
  1. Step through the linked list one node at a time
  2. Use list_search to find the first occurrence of the target, then use list_search again to find the next occurrence, and so on
     - ✓We chose this approach because it made better use of list_search

The Third Bag Class — Implementation (cont.)

```
bag::size_type bag::count(const value_type& target) const
{
  size_type answer;

  // Use const node* since we do not intend to change the nodes
  const node *cursor;

  answer = 0;
  cursor = list_search(head_ptr, target);
  while (cursor != NULL)
  {
          // Each time that cursor is not NULL, we have another
          // occurrence of target, so we add one to answer, and
          // move cursor to the next occurrence of the target
         ++answer;
         cursor = cursor->link( );
         cursor = list_search(cursor, target);
   }
   return answer;
}
```

The Third Bag Class — Implementation (cont.)

- operator +=
  - The implementation starts by making a copy of the linked list of the addend
  - The copy is attached at the front of the linked list for the bag that's being added to

```
void bag::operator +=(const bag& addend)
{
  node *copy_head_ptr;
  node *copy_tail_ptr;

  if (addend.many_nodes > 0)
   {
      list_copy(addend.head_ptr, copy_head_ptr, copy_tail_ptr);
      copy_tail_ptr->set_link( head_ptr );
      head_ptr = copy_head_ptr;
      many_nodes += addend.many_nodes;
    }
 }
```

source pointer, newly copy space - head, tail

The Third Bag Class — Implementation (cont.)

- Functions in the bag class

```
bag( );
bag(const bag& source);
~bag( );
size_type erase(const value_type& target);
bool erase_one(const value_type& target);
void insert(const value_type& entry);
void operator +=(const bag& addend);
void operator =(const bag& source);
size_type size( ) const { return many_nodes; }
size_type count(const value_type& target) const;
```

DYNAMIC ARRAYS
VS.
LINKED LISTS
VS.
DOUBLY LINKED LISTS

Dynamic Arrays vs Linked Lists

- Arrays are better at random access
  - O (1) vs. O(n)
- Linked lists are better at insertions/deletions at a cursor
  - O(1) vs O(n)
- Doubly linked lists are better for a two-way cursor
- Resizing can be Inefficient for a Dynamic Array
  - re-allocation, copy, release

Copyright Notice

Part of this lecture was adapted from the slides of Dr. Behnam Dezfouli