

Linked Lists

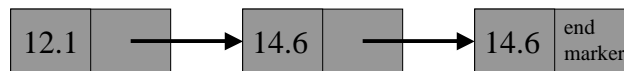
Kruse and Ryba Textbook
4.1 and Chapter 6

Linked Lists

- Linked list of items is arranged in order
- Size of linked list changes as items are inserted or removed
- Dynamic memory allocation is often used in linked list implementation
- Ten fundamental functions are used to manipulate linked lists (see textbook).

Fundamentals

- A linked list is a sequence of items arranged one after another.
- Each item in list is connected to the next item via a *link*



- Each item is placed together with the link to the next item, resulting in a simple component called a *node*.

Declaring a Class for Node

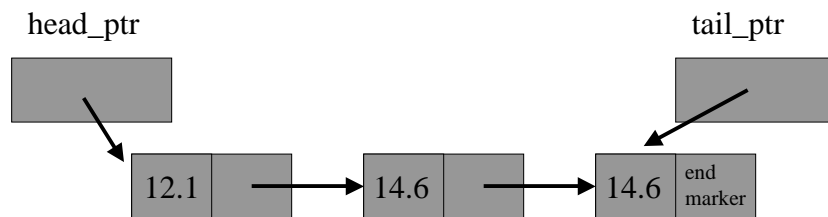
```
struct Node
{
    typedef double Item;
    Item data; // data stored in node
    Node *link; // pointer to next node
};
```

A *struct* is a special kind of class where all members are public. In this case there are two public member variables: data, link.

Whenever a program needs to refer to the item type, we can use the expression `Node::Item`.

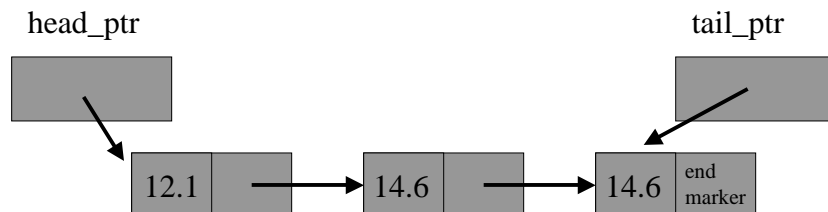
Head Pointers, Tail Pointers

Usually, programs do not actually declare node variables. Instead, the list is accessed through one or more pointers to nodes.



```
Struct Node  
{  
    typedef double Item;  
    Item data;  
    Node *link;  
};
```

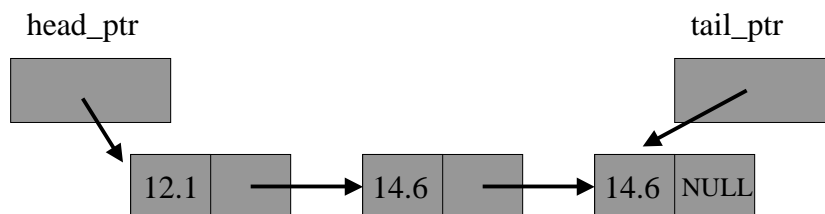
```
Node *head_ptr;  
Node *tail_ptr;
```



Null Pointer

- The final node in the linked list does not point to a next node.
- If link does not point to a node, its value is set to *NULL*.
- *NULL* is a special C++ constant, from the standard library facility `<stdlib.h>`
- *NULL* pointer is often written 0 (zero).

Use of *NULL* pointer in last node of linked list:



Empty List

- When the list is empty, both the head_ptr and tail_ptr are NULL.
- When creating a new linked list, it starts out empty (both tail and head pointers NULL).

```
Node *head_ptr,*tail_ptr;  
head_ptr = NULL;  
tail_ptr = NULL;
```

head_ptr

Null

tail_ptr

Null

- Any linked list functions you write should handle the case of empty list (head and tail pointers NULL).

Member Selection Operator

Suppose a program has built a linked list:



head_ptr is a pointer to a node. How can we get/set the value of the Item inside the node?

Member Selection Operator

One possible syntax:

```
(*head_ptr).data = 4.5;  
cout << (*head_ptr).data;
```

The expression `(*head_ptr).data` means *the data member of the node pointed to by head_ptr*.

Member Selection Operator

Preferred syntax:

```
head_ptr->data = 4.5;  
cout << head_ptr->data;
```

The symbol “->” is considered a single operator.
Reminds you of an arrow pointing to the member.

The expression `head_ptr->data` means *the data member of the node pointed to by head_ptr*.

Two Common Pointer Bugs

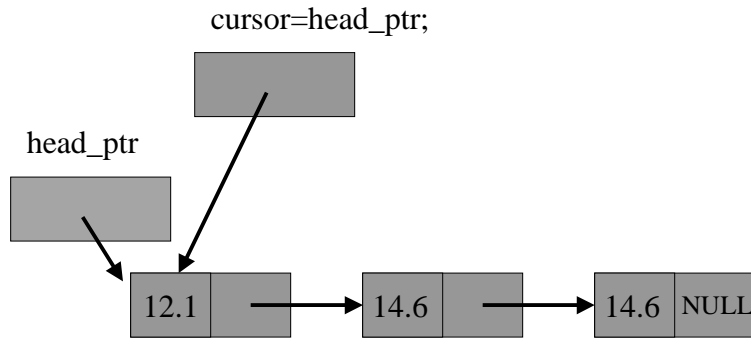
- Attempting to dereference a pointer via *p or p-> when p=NULL.
- Attempting to dereference a pointer via *p or p-> when p is not properly initialized.
- NOTE: this error does not cause a syntax error, but instead causes errors:
 - Bus Error
 - Segmentation violation
 - Address protection violation

Computing the Length of a Linked List

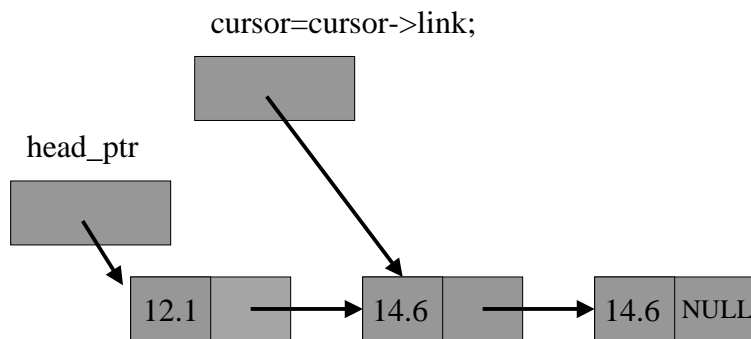
```
size_t list_length(Node * head_ptr)
{
    Node *cursor;
    size_t answer=0;

    for(cursor=head_ptr; cursor != NULL; cursor=cursor->link)
        answer++;
    return answer;
}
```

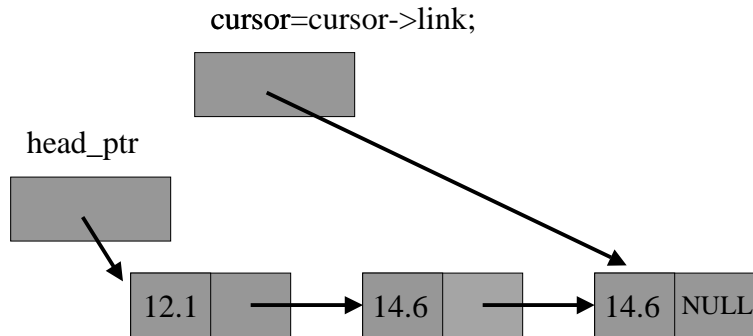
Computing the Length of a Linked List



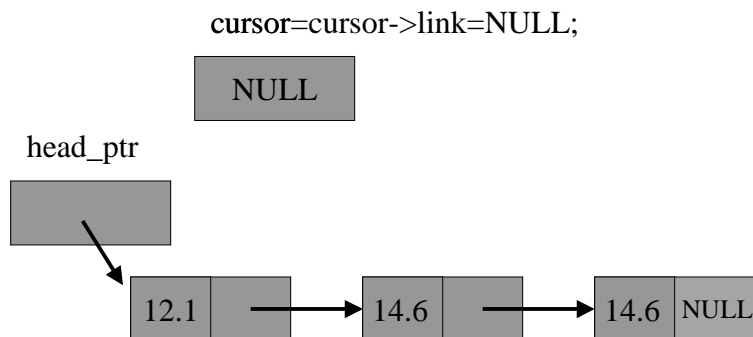
Computing the Length of a Linked List



Computing the Length of a Linked List



Computing the Length of a Linked List



Computing the Length of a Linked List

```
size_t list_length(Node * head_ptr)
{
    Node *cursor;
    size_t answer=0;

    for(cursor=head_ptr; cursor != NULL; cursor=cursor->link)
        answer++;
    return answer;
}
```

Traversing a Linked List

Common pattern in functions that need to traverse a linked list:

```
...
for(cursor=head_ptr; cursor != NULL; cursor=cursor->link)
...
```

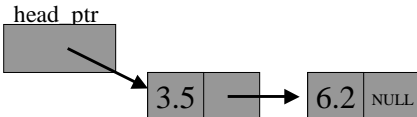
Will this work for an empty list?

Always make sure your functions work in the empty list case!!

Inserting a Node at List Head

```
void list_head_insert(Node* head_ptr, const Node::Item& entry)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: new node is added to front of list containing entry, and
    // head_ptr is set to point at new node.

    Node *insert_ptr;
    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```

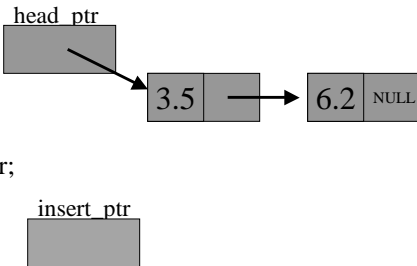


The diagram illustrates the state of a linked list after the insertion. A pointer labeled 'head_ptr' points to a new node containing the value '3.5'. This new node's 'link' field points to the original head of the list, which is a node containing the value '6.2' and 'NULL'.

Inserting a Node at List Head

```
void list_head_insert(Node* head_ptr, const Node::Item& entry)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: new node is added to front of list containing entry, and
    // head_ptr is set to point at new node.

    Node *insert_ptr;
    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```



The diagram illustrates the state of a linked list after the insertion. A pointer labeled 'head_ptr' points to a new node containing the value '3.5'. This new node's 'link' field points to the original head of the list, which is a node containing the value '6.2' and 'NULL'. A separate pointer labeled 'insert_ptr' points to the new node.

Inserting a Node at List Head

```
void list_head_insert(Node* head_ptr, const Node::Item& entry)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: new node is added to front of list containing entry, and
    // head_ptr is set to point at new node.
```

```
    Node *insert_ptr;
    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```

The diagram shows the state of a linked list before and after the insertion. Initially, a box labeled 'head_ptr' has an arrow pointing to a node containing '3.5'. This node's link field points to another node containing '6.2' and 'NULL'. After the function call, a new box labeled 'insert_ptr' has an arrow pointing to a new, empty node. The 'head_ptr' box now has an arrow pointing to this new node, and the original '3.5' node is no longer pointed to by 'head_ptr'.

Inserting a Node at List Head

```
void list_head_insert(Node* head_ptr, const Node::Item& entry)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: new node is added to front of list containing entry, and
    // head_ptr is set to point at new node.
```

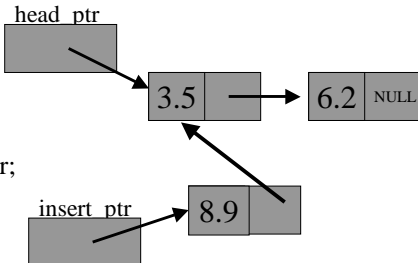
```
    Node *insert_ptr;
    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```

The diagram shows the state of a linked list before and after the insertion. Initially, a box labeled 'head_ptr' has an arrow pointing to a node containing '3.5'. This node's link field points to another node containing '6.2' and 'NULL'. After the function call, a new box labeled 'insert_ptr' has an arrow pointing to a new node containing '8.9'. The 'head_ptr' box now has an arrow pointing to this new node, and the original '3.5' node is no longer pointed to by 'head_ptr'.

Inserting a Node at List Head

```
void list_head_insert(Node* head_ptr, const Node::Item& entry)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: new node is added to front of list containing entry, and
    // head_ptr is set to point at new node.
```

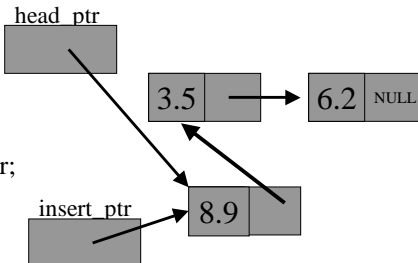
```
    Node *insert_ptr;
    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```



Inserting a Node at List Head

```
void list_head_insert(Node* head_ptr, const Node::Item& entry)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: new node is added to front of list containing entry, and
    // head_ptr is set to point at new node.
```

```
    Node *insert_ptr;
    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```

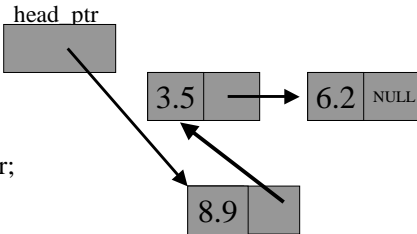


Inserting a Node at List Head

```
void list_head_insert(Node* head_ptr, const Node::Item& entry)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: new node is added to front of list containing entry, and
    // head_ptr is set to point at new node.

    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```



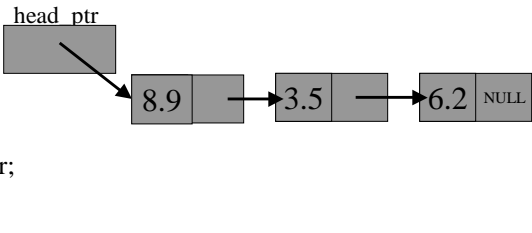
The diagram shows a pointer box labeled 'head_ptr' with an arrow pointing to a new node box containing '8.9'. To the right, there is a linked list starting with a node containing '3.5' and an arrow pointing to a node containing '6.2' and 'NULL'. Another arrow points from the '3.5' node to the '8.9' node, indicating the new list structure after insertion.

Inserting a Node at List Head

```
void list_head_insert(Node* head_ptr, const Node::Item& entry)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: new node is added to front of list containing entry, and
    // head_ptr is set to point at new node.

    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = head_ptr;
    head_ptr = insert_ptr;
}
```



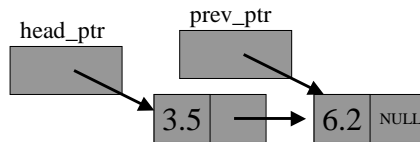
The diagram shows a pointer box labeled 'head_ptr' with an arrow pointing to a new node box containing '8.9'. To the right, there is a linked list starting with a node containing '8.9' which points to a node containing '3.5' which points to a node containing '6.2' and 'NULL'. This represents the state of the list after the insertion operation.

Inserting a Node *not* at List Head

```
void list_insert(Node* previous_ptr, const Node::Item& entry)
{
    // Precondition: previous_ptr is a pointer to a node in a valid linked list
    // Postcondition: new node is added after the node pointed to by
    // previous_ptr

    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = previous_ptr->link;
    previous_ptr->link = insert_ptr;
}
```

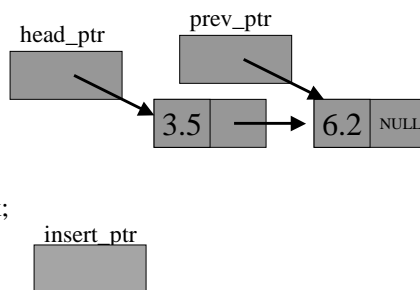


Inserting a Node *not* at List Head

```
void list_insert(Node* previous_ptr, const Node::Item& entry)
{
    // Precondition: previous_ptr is a pointer to a node in a valid linked list
    // Postcondition: new node is added after the node pointed to by
    // previous_ptr

    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = previous_ptr->link;
    previous_ptr->link = insert_ptr;
}
```

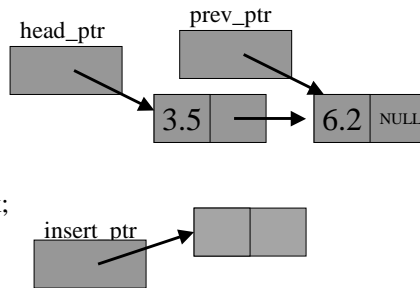


Inserting a Node *not* at List Head

```
void list_insert(Node* previous_ptr, const Node::Item& entry)
{
    // Precondition: previous_ptr is a pointer to a node in a valid linked list
    // Postcondition: new node is added after the node pointed to by
    // previous_ptr

    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = previous_ptr->link;
    previous_ptr->link = insert_ptr;
}
```

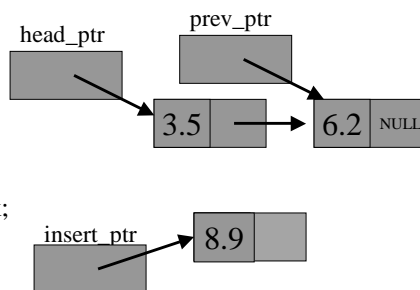


Inserting a Node *not* at List Head

```
void list_insert(Node* previous_ptr, const Node::Item& entry)
{
    // Precondition: previous_ptr is a pointer to a node in a valid linked list
    // Postcondition: new node is added after the node pointed to by
    // previous_ptr

    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = previous_ptr->link;
    previous_ptr->link = insert_ptr;
}
```

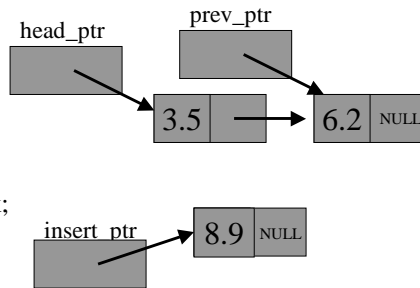


Inserting a Node *not* at List Head

```
void list_insert(Node* previous_ptr, const Node::Item& entry)
{
    // Precondition: previous_ptr is a pointer to a node in a valid linked list
    // Postcondition: new node is added after the node pointed to by
    // previous_ptr

    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = previous_ptr->link;
    previous_ptr->link = insert_ptr;
}
```

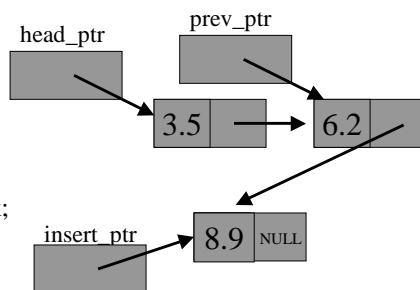


Inserting a Node *not* at List Head

```
void list_insert(Node* previous_ptr, const Node::Item& entry)
{
    // Precondition: previous_ptr is a pointer to a node in a valid linked list
    // Postcondition: new node is added after the node pointed to by
    // previous_ptr

    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = previous_ptr->link;
    previous_ptr->link = insert_ptr;
}
```

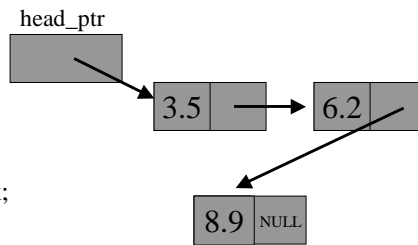


Inserting a Node *not* at List Head

```
void list_insert(Node* previous_ptr, const Node::Item& entry)
{
    // Precondition: previous_ptr is a pointer to a node in a valid linked list
    // Postcondition: new node is added after the node pointed to by
    // previous_ptr

    Node *insert_ptr;

    insert_ptr = new Node;
    insert_ptr->data = entry;
    insert_ptr->link = previous_ptr->link;
    previous_ptr->link = insert_ptr;
}
```



List Search

- Find the first node in a list that contains the specified item.
- Return pointer to that node.

Searching List for Item

```
Node* list_search(Node* head_ptr, const Node::Item& target)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: return value is pointer to first node containing
    // specified target. Returns NULL if no matching node found.

    Node *cursor;
    for(cursor = head_ptr; cursor != NULL; cursor = cursor->link)
        if(target == cursor->data)
            return cursor;
    return NULL;
}
```

Locating n^{th} Node in List

```
Node* list_locate(Node* head_ptr, size_t position)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: return value is pointer to node at specified position
    // first node in list has position=0

    Node *cursor;
    size_t i;

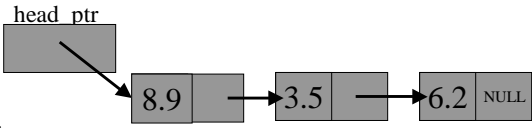
    cursor = head_ptr;
    for(i=0; (i<position) && (cursor != NULL); ++i)
        cursor = cursor->link;

    return cursor;
}
```

Removing a Node at List Head

```
void list_head_remove(Node* head_ptr)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: first node is removed from front of list, and
    // head_ptr is set to point at head_ptr->link. Removed node is deleted

    Node *remove_ptr;
    head_ptr
    remove_ptr = head_ptr;
    head_ptr = head_ptr->link;
    delete remove_ptr;
}
```

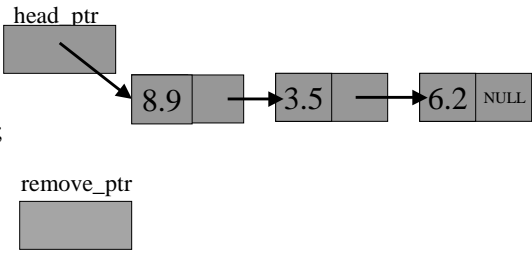


The diagram shows a linked list with three nodes. The first node contains the value 8.9 and points to the second node. The second node contains the value 3.5 and points to the third node. The third node contains the value 6.2 and points to NULL. A pointer labeled head_ptr points to the first node.

Removing a Node at List Head

```
void list_head_remove(Node* head_ptr)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: first node is removed from front of list, and
    // head_ptr is set to point at head_ptr->link. Removed node is deleted

    Node *remove_ptr;
    head_ptr
    remove_ptr = head_ptr;
    head_ptr = head_ptr->link;
    delete remove_ptr;
}
```



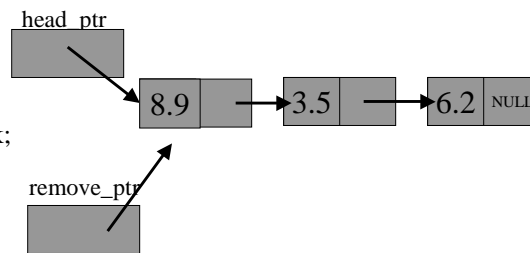
The diagram shows a linked list with three nodes. The first node contains the value 8.9 and points to the second node. The second node contains the value 3.5 and points to the third node. The third node contains the value 6.2 and points to NULL. A pointer labeled head_ptr points to the first node. A pointer labeled remove_ptr also points to the first node.

Removing a Node at List Head

```
void list_head_remove(Node* head_ptr)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: first node is removed from front of list, and
    // head_ptr is set to point at head_ptr->link. Removed node is deleted
```

```
    Node *remove_ptr;
```

```
    remove_ptr = head_ptr;
    head_ptr = head_ptr->link;
    delete remove_ptr;
}
```

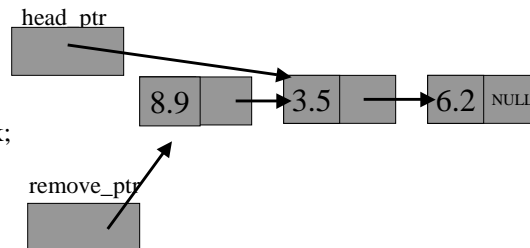


Removing a Node at List Head

```
void list_head_remove(Node* head_ptr)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: first node is removed from front of list, and
    // head_ptr is set to point at head_ptr->link. Removed node is deleted
```

```
    Node *remove_ptr;
```

```
    remove_ptr = head_ptr;
    head_ptr = head_ptr->link;
    delete remove_ptr;
}
```



Removing a Node at List Head

```
void list_head_remove(Node* head_ptr)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: first node is removed from front of list, and
    // head_ptr is set to point at head_ptr->link. Removed node is deleted
```

```
    Node *remove_ptr;
    remove_ptr = head_ptr;
    head_ptr = head_ptr->link;
    delete remove_ptr;
}
```

The diagram shows a linked list with two nodes. The first node contains the value '3.5' and its 'link' field points to the second node. The second node contains the value '6.2' and its 'link' field is 'NULL'. A pointer labeled 'head_ptr' points to the first node. A pointer labeled 'remove_ptr' also points to the first node. An arrow indicates the transition of 'head_ptr' to point to the second node, which is the state after the first node has been removed.

Will it work if head_ptr=NULL?

```
void list_head_remove(Node* head_ptr)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: first node is removed from front of list, and
    // head_ptr is set to point at head_ptr->link. Removed node is deleted
```

```
    Node *remove_ptr;

    remove_ptr = head_ptr;
    head_ptr = head_ptr->link;
    delete remove_ptr;
}
```

Will it work if head_ptr=NULL?

```
void list_head_remove(Node* head_ptr)
{
    // Precondition: head_ptr is a head pointer to a linked list
    // Postcondition: first node is removed from front of list, and
    // head_ptr is set to point at head_ptr->link. Removed node is deleted

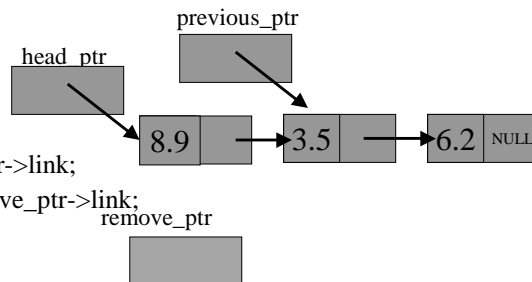
    Node *remove_ptr;
    if(head_ptr == NULL) return;
    remove_ptr = head_ptr;
    head_ptr->link = head_ptr->next;
    delete remove_ptr;
}
```

Removing a *not* Node at List Head

```
void list_remove(Node* previous_ptr)
{
    // Precondition: previous_ptr is a pointer to node in a linked list
    // Postcondition: node is removed from front of list, and
    // removed node is deleted

    Node *remove_ptr;

    remove_ptr = previous_ptr->link;
    previous_ptr->link = remove_ptr->link;
    delete remove_ptr;
}
```

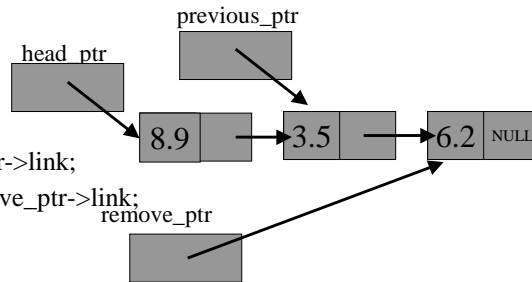


Removing a *not* Node at List Head

```
void list_remove(Node* previous_ptr)
{
    // Precondition: previous_ptr is a pointer to node in a linked list
    // Postcondition: node is removed from front of list, and
    // removed node is deleted

    Node *remove_ptr;

    remove_ptr = previous_ptr->link;
    previous_ptr->link = remove_ptr->link;
    delete remove_ptr;
}
```

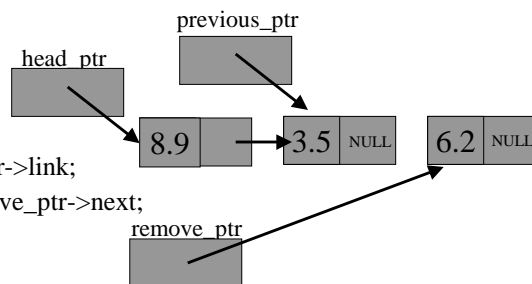


Removing a *not* Node at List Head

```
void list_remove(Node* previous_ptr)
{
    // Precondition: previous_ptr is a pointer to node in a linked list
    // Postcondition: node is removed from front of list, and
    // removed node is deleted

    Node *remove_ptr;

    remove_ptr = previous_ptr->link;
    previous_ptr->link = remove_ptr->next;
    delete remove_ptr;
}
```

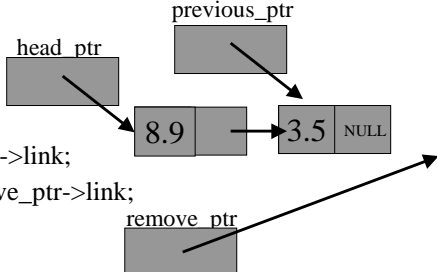


Removing a *not* Node at List Head

```
void list_remove(Node* previous_ptr)
{
    // Precondition: previous_ptr is a pointer to node in a linked list
    // Postcondition: node is removed from front of list, and
    // removed node is deleted

    Node *remove_ptr;

    remove_ptr = previous_ptr->link;
    previous_ptr->link = remove_ptr->link;
    delete remove_ptr;
}
```

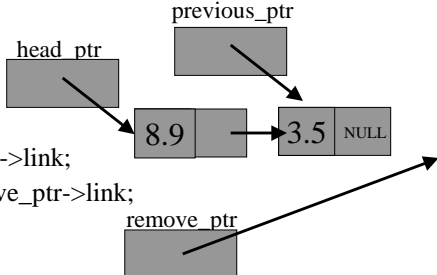


Removing a *not* Node at List Head

```
void list_remove(Node* previous_ptr)
{
    // Precondition: previous_ptr is a pointer to node in a linked list
    // Postcondition: node is removed from front of list, and
    // removed node is deleted

    Node *remove_ptr;

    remove_ptr = previous_ptr->link;
    previous_ptr->link = remove_ptr->link;
    delete remove_ptr;
}
```



Other List Functions

- list clear: *empties a list, deleting all nodes.*
- list copy: *copies a list, and all its nodes.*
- list append: appends one list onto the end of another

Implementations and interfaces may vary, but the basic operations on lists remain more or less the same.

Better implementation: define a list class!! This is object oriented programming after all.