Lecture 7
Linked Lists

Instructor: George Wolberg
Department of Computer Science
City College of New York
Motivation

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

| 10 | 20 | 30 | ? | ? |

18
Motivation

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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>20</td>
<td>30</td>
<td>?</td>
</tr>
</tbody>
</table>

18
Motivation

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

| 10 | 18 | 20 | 30 | ? |

- So the Big-O of the insert is O(n)
Motivation

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

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
<th>?</th>
<th>?</th>
</tr>
</thead>
</table>

18

We need a new data structure
Motivation

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

break an array into a linked chain...
Motivation

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

10 -> 20 -> 30 -> ?

and then put the new item into the chain
Motivation

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

But the links (->) need some way to build up
Chapter 5 introduces the often-used data structure of linked lists.

This presentation shows how to implement the most common operations on 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;
};
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;
};
Each node also contains a link field which is a pointer to another node.

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

```
node * head_ptr;
```
A program can also keep track of the last node by using a pointer variable such as \texttt{tail\_ptr} in this example.

Notice that \texttt{tail\_ptr} itself is not a node -- it is a pointer to a node.
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.

```c
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
The node class is fundamental to linked lists. The private member variables:

- **data_field**
- **link_field**

The member functions include:

- A constructor
- Set data and set link
- Retrieve data and retrieve link

```cpp
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 = new_data; }
    void set_link(node* new_link) { link = new_link; }

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

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

private:
    value_type data;
    node* link;
};
```

Why TWO? p. 213-4
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();
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 that handy indexing

=> Linked List Toolbox

- using node class
The Workings of four functions

- This lecture will show four functions:
  - Compute the length of a linked list (code)
  - Insert a new node at the head (code)
  - Insert a node at any location (pseudo-code)
  - Delete a node from the head (pseudo-code)

- Read Section 5.2 for other functions in the Toolbox
  - will be used in container classes bag and sequence
Length of a Linked List

We simply want to compute the **length** of the linked list, for example the one shown here.

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

```c
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`

```c
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**.
size_t list_length(const node* head_ptr)
{
    const node *cursor;
    size_t count = 0;  // step 1
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link())
        count++;  // step 2
    return count;  // step 3
}

1. Initialize the count to zero.
2. 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; // step 1
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link())
        count++; // step 2
    return count; // step 3
}

1. Initialize the count to zero.
2. Each time cursor points to a new node, add 1 to count.
3. return count.
size_t list_length(const node* head_ptr)
{
    const node *cursor;
    size_t count = 0; // step 1
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link())
        count++ ; // step 2
    return count; // step 3
}

1. Initialize the count to zero.
2. Each time cursor points to a new node, add 1 to count.
3. return count.
size_t list_length(const node* head_ptr) {
    const node *cursor;
    size_t count = 0;  // step 1
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link())
        count++;
    // step 2
    return count;  // step 3
}
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; // step 3
}
Does list_length work for an empty list?

```c
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`
The Workings of four functions

- This lecture will show four functions:
  - Compute the length of a linked list (code)
  - Insert a new node at the head (code)
  - Insert a node at any location (pseudo-code)
  - Delete a node from the head (pseudo-code)

- Read Section 5.2 for other functions in the Toolbox
  - will be used in container classes bag and sequence
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
Create a new node, pointed to by a local variable `insert_ptr`.

```cpp
void list_head_insert(node*& head_ptr, const node::value_type& entry);
```
Inserting a Node at the Head

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

- `insert_ptr = new node;`
Inserting a Node at the Head

- `insert_ptr = new node;`
- Place the data in the new node's data field.

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

- insert_ptr = new node;
- ? = entry;

What expression appears on the left side of the assignment statement?
Inserting a Node at the Head

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

- `insert_ptr = new node;`
- `insert_ptr->data = entry;`

But data is a private variable, so cannot be accessed by a non-member function.
Inserting a Node at the Head

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

- `insert_ptr = new node;`
- `insert_ptr->data = entry;`

But data is a private variable, so cannot be accessed by a non-member function.
Inserting a Node at the Head

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

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

Instead, `Set_data` function is used since `data_field` is a private variable of the node class.
Inserting a Node at the Head

```cpp
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(?)`;

**What will be the parameter?**
Inserting a Node at the Head

```cpp
def 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);
```

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

The new node is linked to the node that `head_ptr` is pointing to.
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.
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

```c
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.
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

Linked List: O(1)
- cmp: 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;
}
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;
}
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;
}
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;
}
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

When the function returns, the linked list has one node, containing 13.
Caution!

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

Q: Can you give an implementation with ONLY a single statement?
Inserting a Node at the Head

```cpp
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;
}
```

YES, we can use the constructor with parameters!
Inserting a Node at the Head

```cpp
void list_head_insert(node*& head_ptr, const node::value_type& entry) {
    head_ptr = new node(entry, head_ptr);
}
```

and assign the return pointer of new directly to the head pointer!
Declarations for Linked Lists

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

```cpp
class node {
public:
    typedef int value_type;
    ...
private:
    value_type data;
    node *link;
};
```
The node class is fundamental to linked lists. The private member variables are:

- `data_field`
- `link_field`

The member functions include:

- A constructor
- Set data and set link
- Retrieve data and retrieve link

```cpp
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 = new_data; }
    void set_link(node* new_link) { link = new_link; }

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

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

private:

    value_type data;
    node* link;
};
```
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; // step 3
}
The Workings of four functions

- This lecture will show four functions:
  - Compute the length of a linked list (code)
  - Insert a new node at the head (code)
  - Insert a node at any location (pseudo-code)
  - Delete a node from the head (pseudo-code)
- Read Section 5.2 for other functions in the Toolbox
  - will be used in container classes bag and sequence
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

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

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

A pointer to the head of the list
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):

- Start by setting a pointer named `previous_ptr` to point to the node which is just before the new node's position.
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.
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.

Look at the pointer which is **in the node** `*previous_ptr`.

What is the name of this pointer?
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.
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.

`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.

The new node must be inserted at the head of this small linked list.

Write one C++ statement which will do the insertion.
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.

Write one C++ statement which will do the insertion.

```c
list_head_insert(previous_ptr->link, 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.

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

More precisely, you need to use member function `link()` and have three lines of code.
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.

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

More precisely, you need to use member function link(), and have three lines of code.
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.

More precisely, you need to use member function `link()` , and have three lines of code:

```c
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:
  ```c
  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.
  - Do the following:
    ```c
    node *sl_head_ptr;
    sl_head_ptr = previous_ptr->link();
    list_head_insert(sl_head_ptr, entry);
    previous_ptr->set_link(sl_head_ptr);
    ```
The process of adding a new node in the middle of a list (only the step after previous_ptr has been set) can also be incorporated as a separate function. This function is called `list_insert` in the linked list toolkit of Section 5.2.

Challenge yourself:

- The textbook actually gives you a different implementation (p 235, 4 lines of code)
- Can you implement `list_insert` with just one line of code?
  - Don’t use `list_head_insert`
  - See Self-Test Ex 16
The Workings of four functions

- This lecture will show four functions:
  - Compute the length of a linked list (code)
  - Insert a new node at the head (code)
  - Insert a node at any location (pseudo-code)
  - Delete a node from the head (pseudo-code)

- Read Section 5.2 for other functions in the Toolbox
  - will be used in container classes bag and sequence
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();

Draw the change that this statement will make to the linked list.
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.
It is easy to insert a node at the front of a list. The linked list toolkit also provides a function for inserting a new node elsewhere.

It is easy to remove a node at the front of a list. The linked list toolkit also provides a function for removing a node elsewhere--you should read about this function and the other functions of the toolkit.
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.
Homework...

- Self-Test Exercises (node)
  - 1-12
- Read after class
  - Linked List ToolKit (Section 5.2)
  - Do Self-Test Ex 13 - 25
- Read before the next lecture
  - Section 5.3 - 5.4
- Programming Assignment 4
  - Detailed guidelines online!
Presentation copyright 1997, Addison Wesley Longman,
For use with Data Structures and Other Objects Using C++
by Michael Main and Walter Savitch.


Students and instructors who use Data Structures and Other Objects Using C++ are welcome to use this presentation however they see fit, so long as this copyright notice remains intact.