COEN 79

OBJECT-ORIENTED PROGRAMMING AND ADVANCED DATA STRUCTURES

DEPARTMENT OF COMPUTER ENGINEERING, SANTA CLARA UNIVERSITY

BEHNAM DEZFOULI, PHD

Linked Lists

Slide Set: 5

Learning Objectives

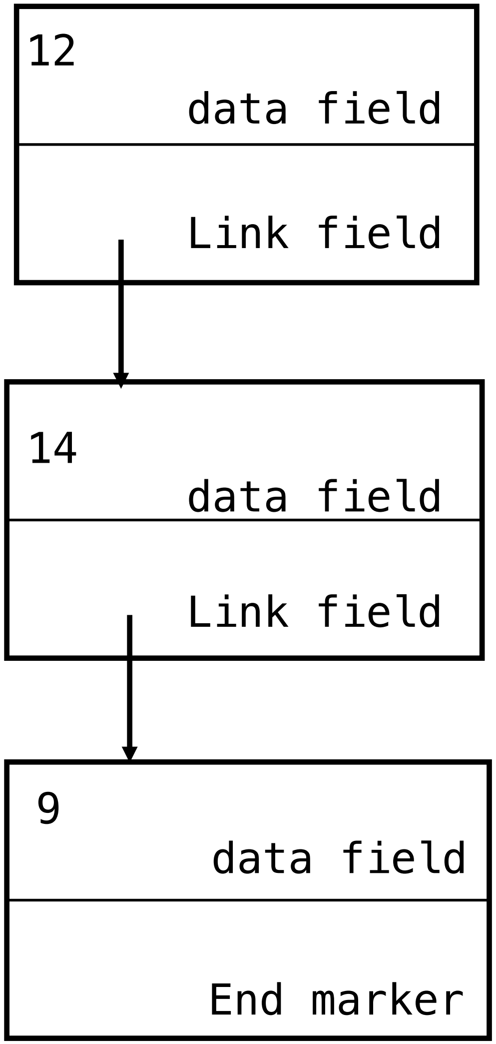
- Design, implement, and test functions to manipulate nodes in a linked list
- Design, implement, and test **data structures that use linked lists** to store a collection of elements
- Analyze problems that can be solved with linked lists
- Propose alternatives to simple linked lists, such as doubly 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's vector, list, and deque classes

A Fundamental Node Class for Linked Lists

Declarations for Linked Lists

- A **linked list** is a sequence of items arranged one after another, with each item connected to the next by a **link**
- Each **node** is a combination of:
  - A piece of data (like a double number)
  - A link to the next node
- A new class for a node:

```
class node
{
public:
    typedef double value_type;
    ...
private:
    value_type  data_field;
    node  *link_field;
};
```

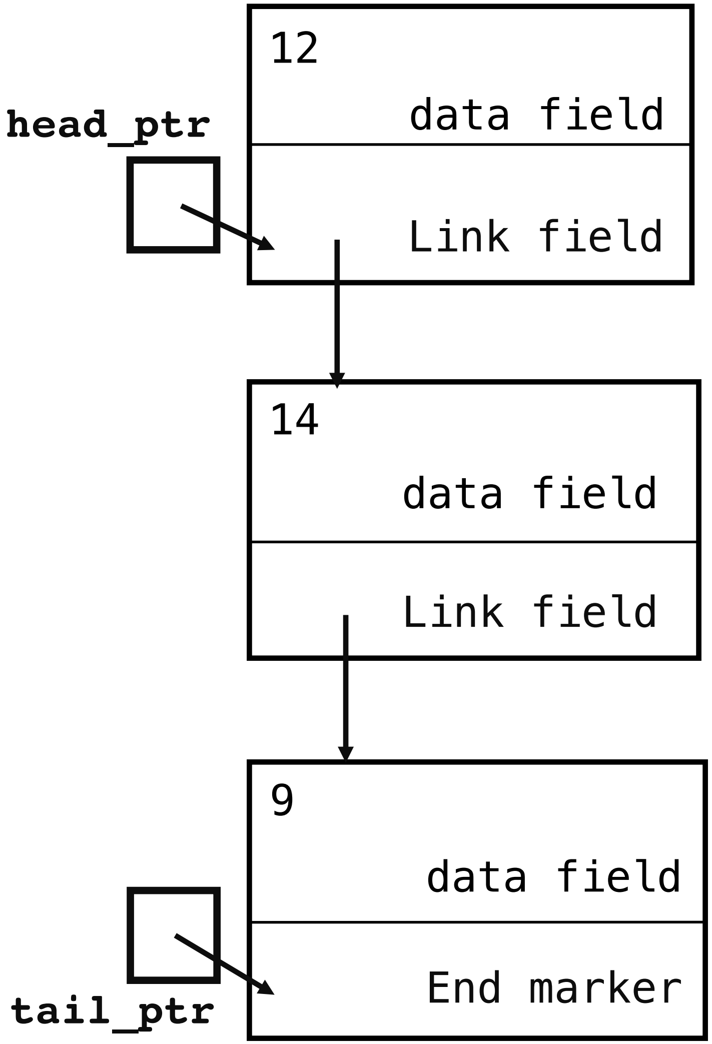


A Fundamental Node Class for Linked Lists

Head Pointers, Tail Pointers

- A linked list is **accessed** through one or more *pointers* to nodes
- A pointer to the first node, **head**, is called the **head pointer**
- Sometimes we maintain a pointer to the last node in a linked list
- The last node is the **tail** of the list, and a pointer to the last node is the **tail pointer**

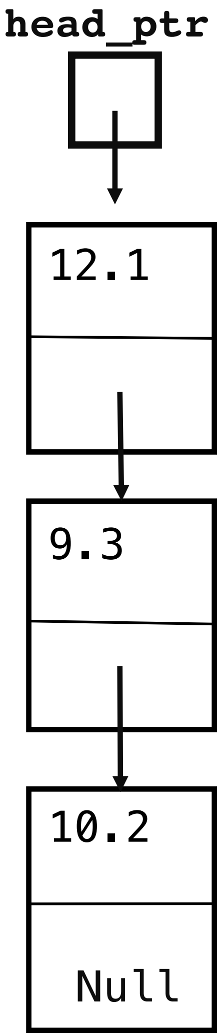
```
node *head_ptr;
node *tail_ptr;
```



A Fundamental Node Class for Linked Lists

The Null Pointer

- The null pointer (`NULL`): Used for any pointer that does not point anywhere
- There are two common situations where the null pointer is used:
  - Use the null pointer for the link field of the final node of a linked list
  - When a linked list does not yet have any nodes, use the null pointer for the value of the head pointer and tail pointer



❑Example: An empty linked list

```
node *head_ptr;
head_ptr = NULL; // Uses the constant NULL from cstdlib
```

A Fundamental Node Class for Linked Lists

The Node Constructor

- The node **constructor** has parameters to initialize both the data and link fields:

```
node( const value_type& init_data = value_type( ),
                        const node* init_link = NULL );
```

- This notation means **the parameter named `init_data` has a default argument that is created by the `value_type`'s default constructor**
- In our case, `value_type` is defined as `double`, so the `init_data` parameter will be `0` if a default argument is needed
- The `init_link` parameter will be NULL if a default argument is needed

A Fundamental Node Class for Linked Lists

The Node Constructor (Cont'd)

```
p = new node;
q = new node(4.9);
r = new node(1.6, p);
```

- What is the result?

A Fundamental Node Class for Linked Lists

The Node Member Functions

- The node has five public member functions for setting and retrieving the data and link fields:

```
class node
 {
  public:

  // TYPEDEF
  typedef double value_type;

  // CONSTRUCTOR
  node( const value_type& init_data = value_type( ),
                                               node* init_link = NULL )
     {
        data_field = init_data;
        link_field = init_link;
     }
```

The Node Member Functions (Cont'd)

```
// 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; }
```

A value parameter that is pointer

```
// 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;
};
```

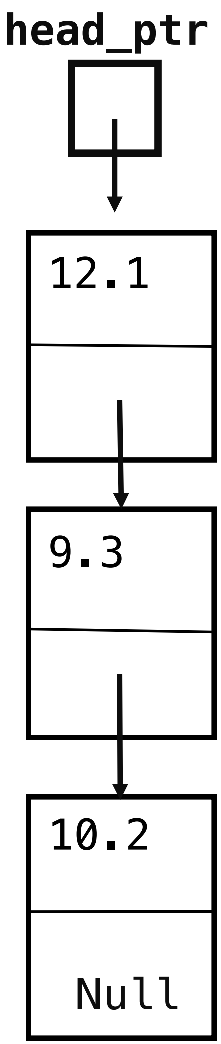
The Member Selection Operator

- As with any object, you can access the public member functions of `*head_ptr`

❑ Example: The following prints `12.1`

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

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



The –> Operator

- The symbol "–>" is called the **member selection operator** or **component selection operator**
- **If p is a pointer to a class, and m is a member of the class, then `p–>m` means the same as `(*p).m`**

❑**Example**

`head_ptr->data()` is the syntax for activating the data function of the node pointed to by `head_ptr`

Clarifying the `const` Keyword

- Consider this pointer to a node: `node *p;`
- We can allocate a node for `p` to point to (through using `p = new node`) and then activate any of the member functions (`p->set_data()` or `p->data()`)
- Now consider this pointer parameter declared using `const` keyword: `const node *c;`
- **The `const` keyword means that the pointer `c` cannot be used to change the node**
  - **Pointer c cannot be used to activate non-constant member functions**
  - The pointer `c` can move and point to many different nodes, but we are forbidden from using `c` to change any of those nodes that `c` points to
- Note that:
  - We cannot assign a const pointer to a non-const pointer
  - We can assign a non-const pointer to a const pointer

Clarifying the const Keyword (Cont'd)

```
node* node_ptr1 = new node;
node* node_ptr2 = new node;

const node* const_node_ptr1 = node_ptr1;

const_node_ptr1->set_data();        Invalid

const_node_ptr1 = node_ptr2;        Valid
```

- **A const pointer cannot be used for activating non-const member functions**
- A new value can be assigned to a const pointer

Clarifying the `const` Keyword (Cont'd)

- If you want to create **a pointer that can be set once during its definition and never changed to point to a new object**, then put the word `const` after the `*`
  - Example: `node *const c = &first;`

```
node* node_ptr1 = new node;
node* node_ptr2 = new node;

node* const const_node_ptr1 = node_ptr1;

const_node_ptr1->set_data(5);     Valid

const_node_ptr1 = node_ptr2;      Invalid
```

A Rule For A Node's Constant Member Functions

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

❑ Example: The purpose of the `link` member function is to obtain a copy of a node's link field

- At first glance, this sounds like a constant member function, since retrieving a member variable does not change an object
- So we might write this:

```
node* link( ) const { return link_field; }
```

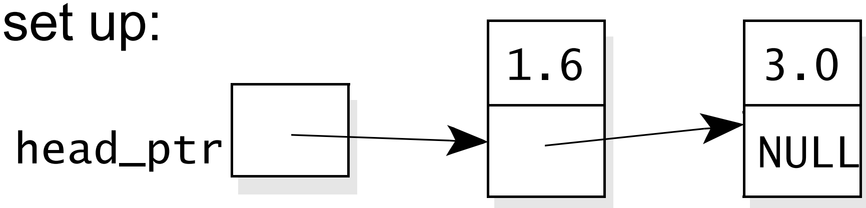
A Fundamental Node Class for Linked Lists

A Rule For A Node's Constant Member Functions (Cont'd)

```
node* link( ) const { return link_field; }
```

- This implementation does compile, but it **violates our programming tip about constant member functions**

❑Example: Suppose we have this list set up:

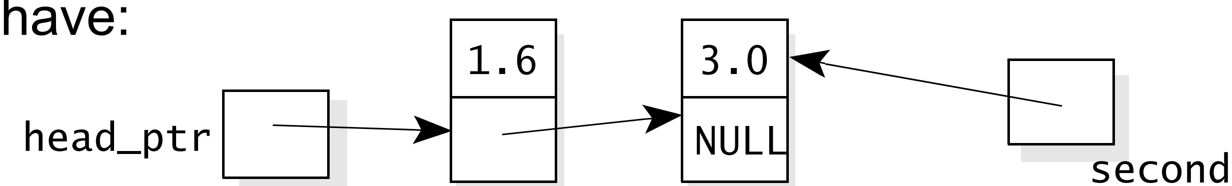


- Using the constant member function, `link`, we can execute two statements that change the data in one of the nodes:

1:

```
node *second = head_ptr->link( );
```

- After this statement we have:



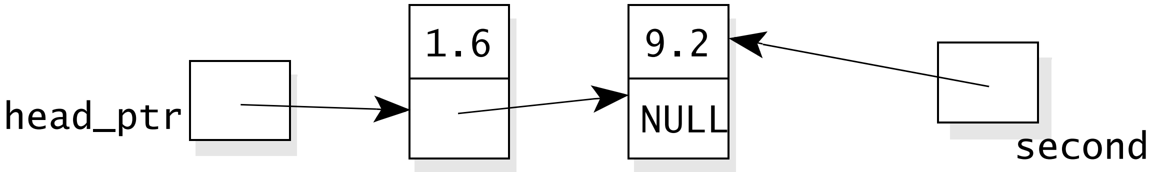
A Rule For A Node’s Constant Member Functions (Cont’d)

- The variable `second` is just an ordinary pointer to a node, it is not a pointer to a constant node, so we can activate any of its member functions:

2:

```
second->set_data(9.2);
```

- After this statement the data in the second node is now 9.2:



- **The node’s constant member functions should never provide a result that we can later use to change any part of the linked list**

A Fundamental Node Class for Linked Lists

A Rule For A Node's Constant Member Functions (Cont'd)

- It makes sense to implement link as a non-constant member function:
  - Making the function non-constant **provides better accuracy about how the function's results might be used**

- `Link` implementation as a non-constant member function:

```
node* link( ) { return link_field; }
```

- **This solution has another problem:**
  - Suppose that `c` is defined as `const node *c`
  - **With the above non-constant link implementation, we could never activate `c->link( )`, because it can only activate const functions**

A Fundamental Node Class for Linked Lists

A Rule For A Node's Constant Member Functions (Cont'd)

- **The final solution:**

```
const node* link( ) const { return link_field; }
```

- This **second version** is a constant member function:
  - **`c->link( )` can be used, even if `c` is declared with the `const` keyword**
  - **The return value from the `const` version of the function cannot be used to change any part of the linked list**
    - The compiler converts the `link_field` to the type `const node*` for the `const` version of the function

A Rule For A Node's Constant Member Functions (Cont'd)

- **When the return value of a member function is a pointer to a node, you should generally have two versions:**
  1. **A `const` version that returns a `const node*`**
  2. **An ordinary version that returns an ordinary pointer to a node**

- When both a const and a non-const version of a function are present:
  - The **compiler automatically chooses the correct version** depending on whether the function was activated by a constant node (such as `const node *c`) or by an ordinary node

A Linked-List Toolkit

Linked-List Toolkit

- We plan to design data structures that use linked lists to store their items
- Storing and retrieving items from a linked list is more work than using an array
  - **We don't have the handy indexing mechanism (such as `data[i]`) to read or write elements**
  - The class requires **extra functions** to build and manipulate the list
- Many container classes might need these same **extra functions**:
  - **We should write a collection of linked-list functions once and for all**, allowing any programmer to use the functions in the implementation of a container class

- **We will create a small toolkit of fundamental linked-list functions** to allow a container class to store elements in a linked list with a simplicity and clarity that is similar to using an array

Header File

- **We include the prototypes of the functions that manipulate linked list in the header file of the node class**

Definition of node class

Prototype of the toolkit functions (Note: These are non-member functions)

```
#ifndef SCU_COEN79_NODE1_H
#define SCU_COEN79_NODE1_H
#include <cstdlib> // Provides size_t and NULL

namespace scu_coen79_5
{
    class node
   {
   public:
                    // TYPEDEF
                    typedef double value_type;

                    // CONSTRUCTOR
                    node(
                        const value_type& init_data = value_type( ),
                        node* init_link = NULL
                    )
                    { data_field = init_data; link_field = 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 retreive
                    // the current link:
                    const node* link( ) const { return link_field; }
                    node* link( )             { return link_field; }

    private:
                    value_type data_field;
                    node* link_field;
   };

    // 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);
}

#endif
```

Linked-List Toolkit

Header File

❑Example

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

- The `list_length` function:
  - Computes the number of nodes in a linked list
  - `head_ptr`, is a pointer to the first node of the linked list
  - Note: If a function does not plan to change the list, then the parameter should be a `const node*`

- The functions should generally **be capable of handling an empty list**
  - **The ability to handle an empty list is one of the reasons why list manipulation functions are generally not node member functions**
  - **Why?**

Implementation of `list_length` Function

```
size_t list_length(const node* head_ptr)
// Precondition: head_ptr is the head pointer of a linked list.
// Postcondition: The value returned is the number of nodes in the
// linked list.
// Library facilities used: cstdlib
{
    const node *cursor;
    size_t answer;
    answer = 0;
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link( ))
        ++answer;

    return answer;
}
```

- Why `cursor` is declared as a `const`?

Parameters for Linked Lists

1. Parameters that are pointers with the const keyword

- Example: The `list_length` function has such a parameter:

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

- The function uses the head pointer to access the list's nodes, but the function does not change any part of the list

- **A pointer to a constant node should be used when the function needs access to the linked list and the function will not make any changes to any of the list's nodes**

Parameters for Linked Lists (Cont'd)

2. Value parameters that are pointers to a node. The second sort of node pointer parameter is a value parameter without the `const` keyword

- Example: One of the toolkit's functions will add a new node after a specified node in the list:

```
void list_insert(node* p, const node::value_type& entry)
// Precondition: p points to a node in a linked list.
// Postcondition: A new node containing the given entry has been
// added after the node that p points to.
```

- **A node pointer should be a value parameter when the function needs access to the linked list, and the function might change the linked list, but the function does not need to make the pointer point to a new node**

Parameters for Linked Lists (Cont'd)

3. Reference parameters that are pointers to a node. Sometimes a function must make a pointer point to a new node

❑ Example: Add a new node at the front of a linked list:

```
void list_head_insert(node*& head_ptr, const node::value_type&
                                                          entry);
// Precondition: head_ptr is the head pointer of a linked list.
// Postcondition: A new node containing the given entry has been
// added at the head of the list; head_ptr now points to the head of
// the new, longer linked list.
```

- **The `head_ptr` is a reference parameter, since the function creates a new head node and makes the head pointer point to this new node**

- **When the function needs access to the linked list and the function makes the pointer point to a new node, this change to the pointer will make the actual argument point to a new node**

A Function to Insert at the Head of a Linked List

`list_head_insert` function adds a new node at the head of a linked list

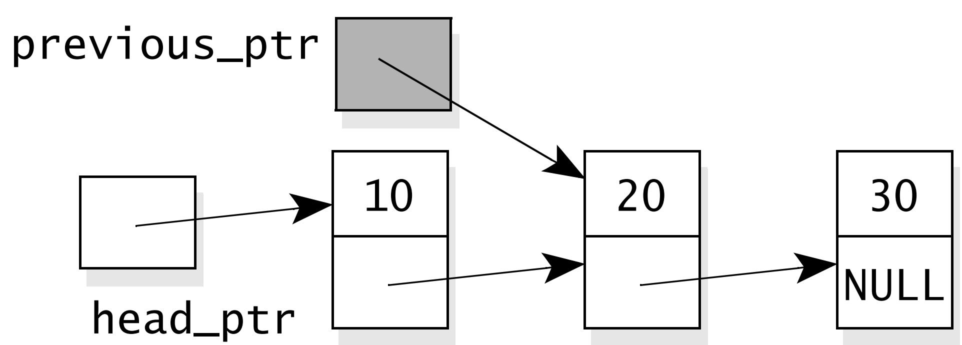
```
void list_head_insert(node*& head_ptr, const node::value_type& entry)
// Precondition: head_ptr is the head pointer of a linked list.
// Postcondition: A new node containing the given entry has been added
// at the head of the linked list; head_ptr now points to the head of
// the new, longer linked list. NOTE: If there is insufficient dynamic
// memory for a new node, then bad_alloc is thrown before changing the
// list.

{
    head_ptr = new node(entry, head_ptr);
}
```

- Show how the function works?
- Does it work when `head_ptr` is NULL?

Inserting a New Node that is Not at the Head

- **Whenever an insertion is not at the head, the insertion process requires a pointer to the node that is just *before* the intended location of the new node**
- We use the name `previous_ptr` for the pointer to the node that is just before the location of the new node
- To insert an item after the 20, we would first have to set up `previous_ptr`:



- Once a program has calculated `previous_ptr`, the insertion can proceed

Inserting a New Node that is Not at the Head (Cont'd)

```
void list_insert (node* previous_ptr, const node::value_type& entry);
// Precondition: previous_ptr points to a node in a linked list.
// Postcondition: A new node containing the given entry has been
// added after the node that previous_ptr points to.
// NOTE: If there is insufficient dynamic memory for a new node, then
// bad_alloc is thrown before changing the list.
```

- Notice that `previous_ptr` is a value parameter that is a pointer to a node
- This allows us to change the list (by inserting a new node), but we will not make `previous_ptr` point to a new node

Inserting a New Node that is Not at the Head (Cont’d)

- Four steps:
  1. Allocate a new node pointed to by a local variable called `insert_ptr`
  2. Place the new entry in the data field of the new node
  3. Make the `link_field` of the new node point to the node after the new node’s location (or `NULL` if there are no nodes after the new location)
  4. Make `previous_ptr->link_field` point to the new node that we just created

Inserting a New Node that is Not at the Head (Cont'd)

```
void list_insert(node* previous_ptr, const node::value_type& entry)

// Precondition: previous_ptr points to a node in a linked list.
// Postcondition: A new node containing the given entry has been added
// after the node that previous_ptr points to.
// NOTE: If there is insufficient dynamic memory for a new node, then
// bad_alloc is thrown before changing the list.

{
    node *insert_ptr;
    insert_ptr = new node;
    insert_ptr->set_data(entry);
    insert_ptr->set_link( previous_ptr->link( ) );
    previous_ptr->set_link(insert_ptr);
}
```

- `insert_ptr` is no longer needed?
- should I write the statement delete `insert_ptr` at the end of the `list_insert` function?

Copying a Linked List

- `list_copy` function makes a copy of a linked list, providing both head and tail pointers for the newly created copy

```
void list_copy (const node* source_ptr, node*& head_ptr,
                                                  node*& tail_ptr);
// Precondition: source_ptr is the head pointer of a linked list.
// Postcondition: head_ptr and tail_ptr are the head and tail
// pointers for a new list that contains the same items as the list
// pointed to by source_ptr.
// NOTE: If there is insufficient dynamic memory to create the new
// list, then bad_alloc is thrown.
```

Copying a Linked List (Cont'd)

- The pseudocode is:
  1. Set `head_ptr` and `tail_ptr` to NULL
  2. If (`source_ptr == NULL`), then return with no further work
     - Note: The head and tail pointers have already been set to NULL
  3. Allocate a new node for the head node of the new list that we are creating, make both `head_ptr` and `tail_ptr` point to this new node, and copy data from the head node of the original list to our new node
  4. Perform the followings:
     - Make `source_ptr` point to the second node of the original list, then the third node, then the fourth node, and so on until we have traversed all of the original list
     - At each node that `source_ptr` points to, add one new node to the tail of the new list, and move the tail pointer forward to the newly added node, as follows:
       - `list_insert(tail_ptr, source_ptr->data( ));`
       - `tail_ptr = tail_ptr->link();`

Function for Copying a Linked List

```
void list_copy(const node* source_ptr, node*& head_ptr,
                                             node*& tail_ptr)
// Precondition: source_ptr is the head pointer of a linked list.
// Postcondition: head_ptr and tail_ptr are the head and tail pointers
// for a new list that contains the same items as the list pointed to
// by source_ptr.

{
    head_ptr = NULL;
    tail_ptr = NULL;

    // Handle the case of the empty list.
    if (source_ptr == NULL)
        return;

    // Make the head node for the newly created list, and put data
    // in it.
    list_head_insert(head_ptr, source_ptr->data( ));
    tail_ptr = head_ptr;
```

- **Note that `head_ptr` and `tail_ptr` are defined in the function that calls `list_copy`**
- **These pointers are passed by reference**

Function for Copying a Linked List (Cont'd)

```
    // Copy the rest of the nodes one at a time, adding at the tail of
    // new list.
    source_ptr = source_ptr->link( );

    while (source_ptr != NULL)
    {
        list_insert(tail_ptr, source_ptr->data( ));
        tail_ptr = tail_ptr->link( );
        source_ptr = source_ptr->link( );
    }
}
```

Linked-List Toolkit

Searching for an Item in a Linked List

- `search` function returns a pointer to a node that contains a specified item
- First version:

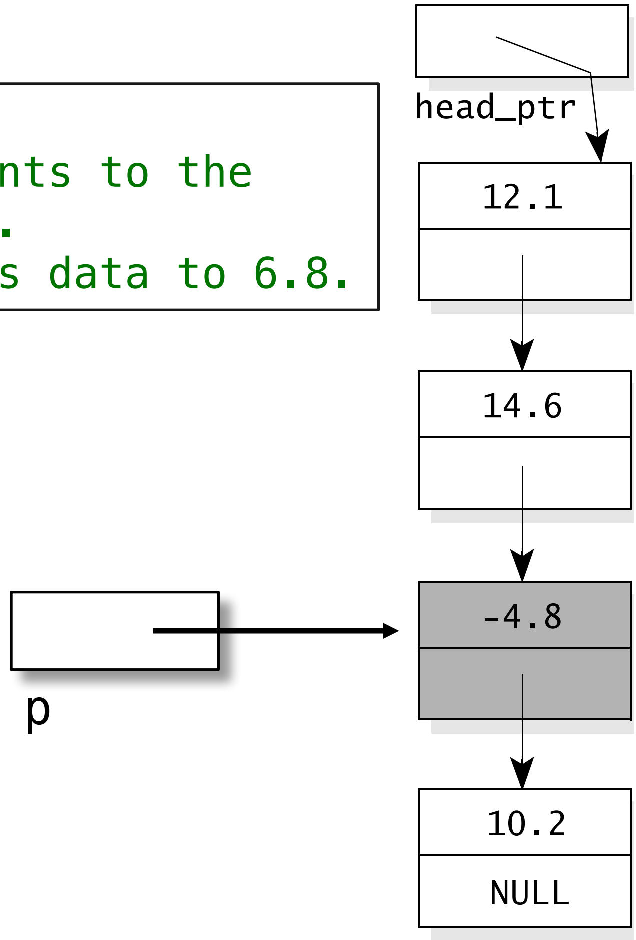
```
node* list_search (node* head_ptr, const node::value_type& target);
// Precondition: head_ptr is the head pointer of a linked list.
// Postcondition: The return value is a pointer to the first node
// containing the specified target in its data field. If there is
// no such node, the null pointer is returned.
```

- This version has a `node*` parameter, and its return value is also a `node*`
- The return value of this function could be used to change the list

Searching for an Item in a Linked List (Cont'd)

- Example: We could execute these statements to find a pointer to the shaded node and change its data to 6.8

```
node* p;
p = list_search(head_ptr, -4.8);   // p now points to the
                                   // -4.8 node.
p->set_data(6.8);                  // Change p's data to 6.8.
```



Implementation of Search Function

```
node* list_search(node* head_ptr, const node::value_type& target)

// Precondition: head_ptr is the head pointer of a linked list.
// Postcondition: The return value is a pointer to the first node
// containing the specified target in its data field. If there is no
// such node, the null pointer is returned.
// Library facilities used: cstdlib

{
    node *cursor;

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

    return NULL;
}
```

- What happens if `head_ptr` is declared as const?

Implementation of Search Function

- **Second version:**

```
const node* list_search
               (const node* head_ptr, const node::value_type& target)
// Precondition: head_ptr is the head pointer of a linked list.
// Postcondition: The return value is a pointer to the first node
// containing the specified target in its data field. If there is no
// such node, the null pointer is returned.
// Library facilities used: cstdlib
{

 const node *cursor;

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

 return NULL;

}
```

The return value from this version of the function is `const node*`

- The compiler prevents us from using the returned pointer to change the list

The compiler determines the right function to use

When to Provide Two Versions for a Function

- When **a nonmember function has a parameter that is a pointer to a node, and the return value is also a pointer to a node**, you should often have two versions:
  - First version: the parameter and return value are both **`node*`**
  - Second version: the parameter and return values are both **`const node*`**

Linked-List Toolkit

Finding a Node by its Position in a Linked List

- Our toolkit has another function that finds a node by its position and returns a pointer to a node in a linked list:

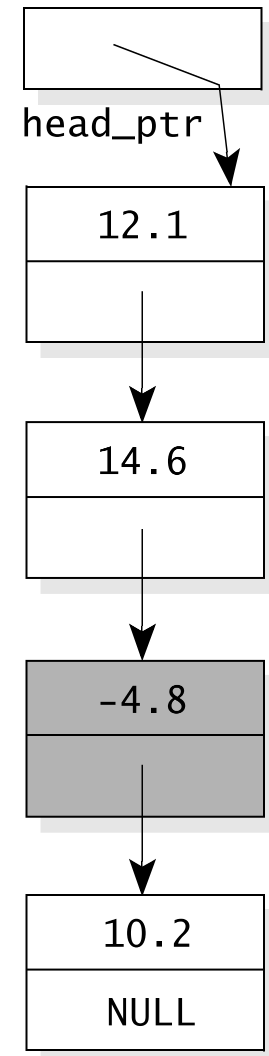
```
node* list_locate(node* head_ptr, size_t position);
// Precondition: head_ptr is the head pointer of a linked
// list, and position > 0.
// Postcondition: The pointer returned points to the node at
// the specified position in the list. (The head node is
// position 1, the next node is position 2, and so on.) If
// there is no such position, then the null pointer is
// returned.
```

- A node is specified by giving its position in the list, with the head node at position 1, the next node at position 2, …

Finding a Node by its Position in a Linked List (Cont'd)

- Example: Function `list_locate(head_ptr, 3)` will return a pointer to the shaded node
  - Notice that the first node is number 1 (not number 0 as in an array)
  - The specified position might also be larger than the length of the list, in this case, the function returns the null pointer

```
cursor = head_ptr;
for (i = 1; (i < position) && (cursor != NULL); ++i)
    cursor = cursor->link( );
```



Removing a Node from the Head of a Linked List

- Our toolkit has functions to remove nodes from a linked list
- The first removal function removes the head node

```
void list_head_remove(node*& head_ptr)
// Precondition: head_ptr is the head pointer of a linked list, with
// at least one node.
// Postcondition: The head node has been removed and returned to the
// heap; head_ptr is now the head pointer of the new, shorter linked
// list.
{
    node *remove_ptr;
    remove_ptr = head_ptr;
    head_ptr = head_ptr->link( );
    delete remove_ptr;
}
```

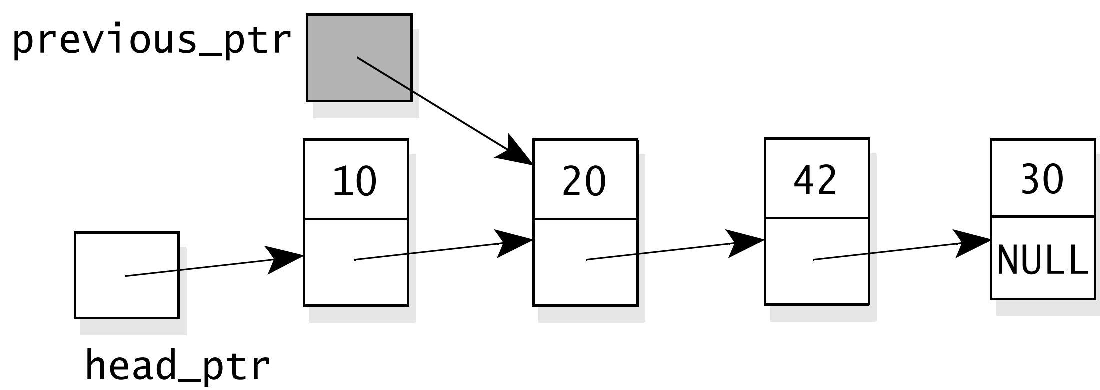
- **The head pointer is a reference parameter, since the function makes the head pointer point to a different node**

Linked-List Toolkit

Removing a Node that is Not at the Head

- Our second removal function **removes a node that is not at the head of a linked list**
- To remove a midlist node, we must set up a pointer to the node that is just before the node that we are removing

❑Example: To remove the 42 from the following list, we would need to set up `previous_ptr` as shown here:



- This is because the `link` field of the previous node must be reassigned

Removing a Node that is Not at the Head (Cont'd)

```
void list_remove(node* previous_ptr)
// Precondition: previous_ptr points to a node in a linked list,
// and this is not the tail node of the list.
// Postcondition: The node after previous_ptr has been removed from
// the linked list.
{
    node *remove_ptr;
    remove_ptr = previous_ptr->link( );
    previous_ptr->set_link( remove_ptr->link( ) );
    delete remove_ptr;
}
```

- The steps:
  1. Set a pointer named `remove_ptr` to point to the condemned node (This is the node after the one pointed to by `previous_ptr`)
  2. Set the link field of `previous_ptr` to point to the node after the condemned node (or NULL if the condemned node is the tail node)
  3. `delete remove_ptr`
     - (This returns the condemned node to the heap)

Clearing a Linked List

- Our final function removes all the nodes from a linked list, returning them to the heap

```
void list_clear(node*& head_ptr)
// Precondition: head_ptr is the head pointer of a linked list.
// Postcondition: All nodes of the list have been deleted, and
// head_ptr is now NULL.
{
    while (head_ptr != NULL)
        list_head_remove(head_ptr);
}
```

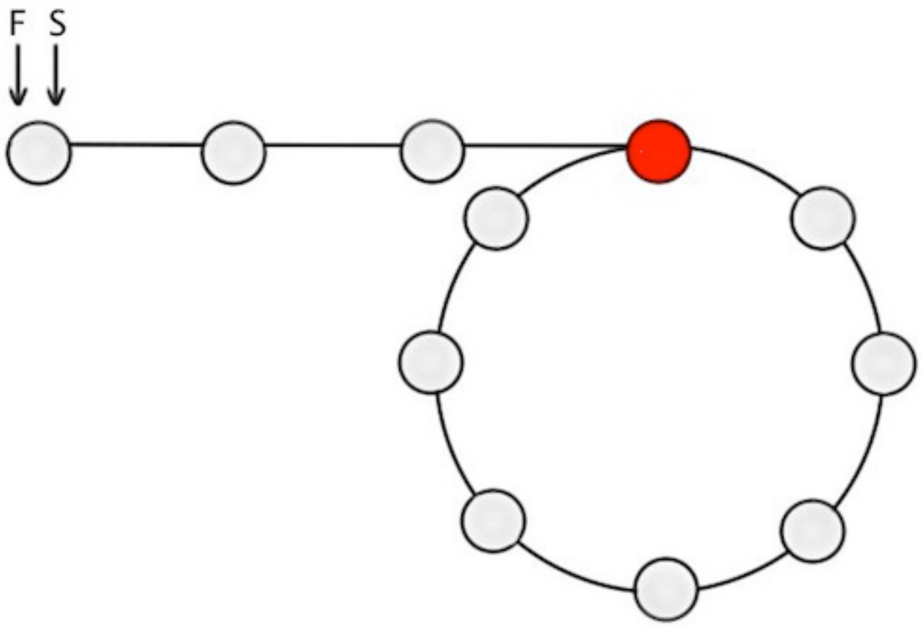
- The head pointer is a reference parameter, so that when the function returns, the actual head pointer in the calling program will be null

Loop Detection

- Given a circular linked list, **we want to find the node at the beginning of the loop** (see the red node below)
- We use the **Floyd's cycle-finding algorithm**

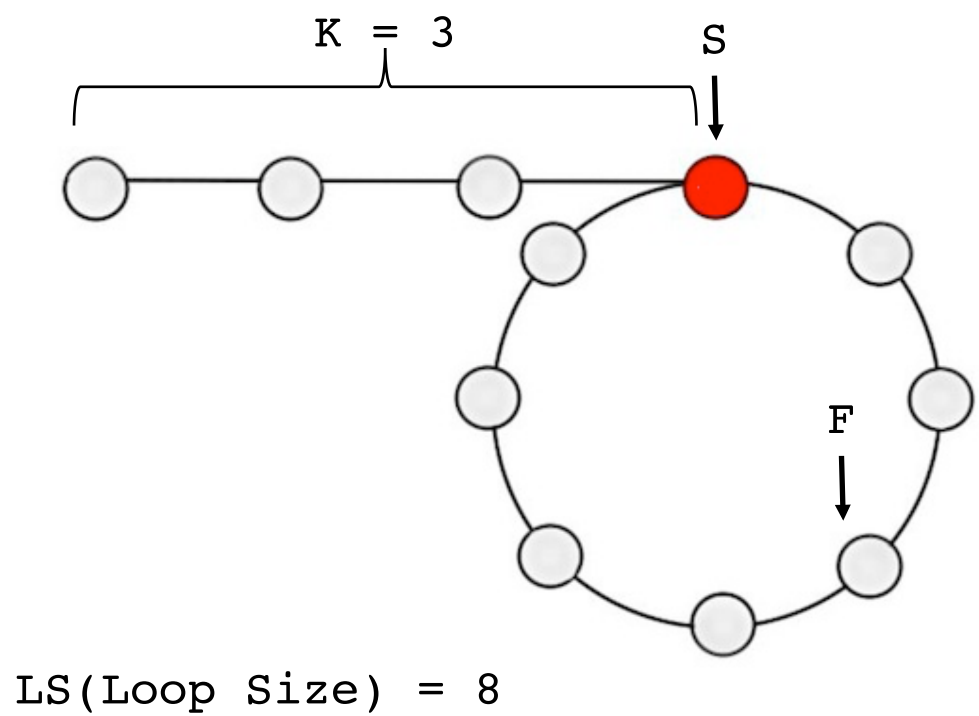
`S: slow_runner`
Moves at rate 1 step

`F: fast_runner`
Moves at rate 2 steps



Loop Detection

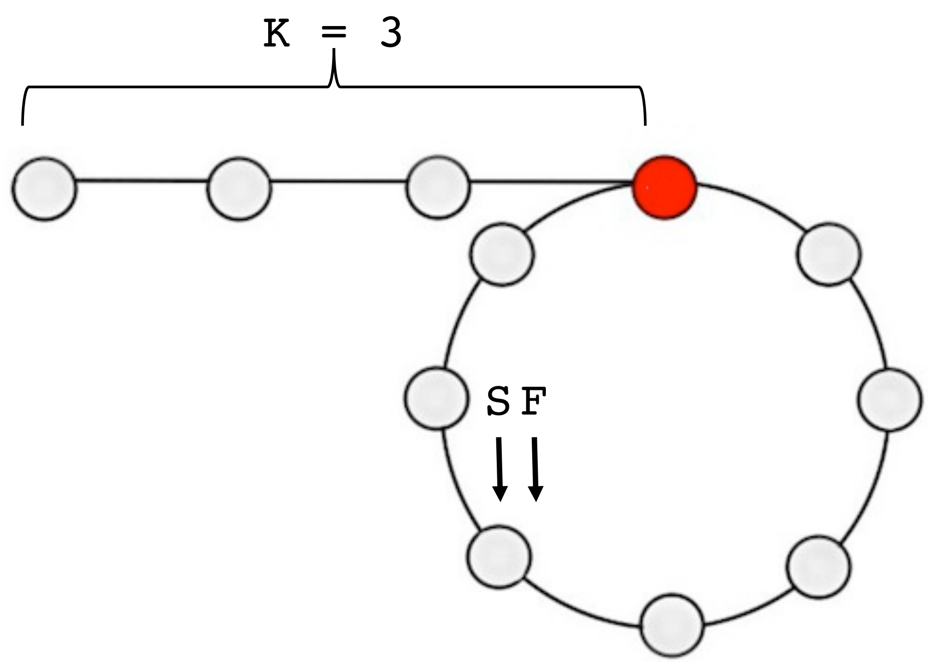
- Assume the non-loop part has size `k`
- After `k` steps of `slow_runner`, `fast_runner` has moved for `2k` steps, and since the first `k` steps are in the non-loop part, it has traversed `k` steps in the loop
- In fact, the distance of `fast_runner` from the start of the loop is `k mod LS`
- In our example, we have:
  `3 mod 8 = 3`
- And the number of steps to the start of the loop (where `slow_runner` point to) is
  `LS – (k mod LS)`
- `In our example:`
  `LS – 3 = 5`



Loop Detection

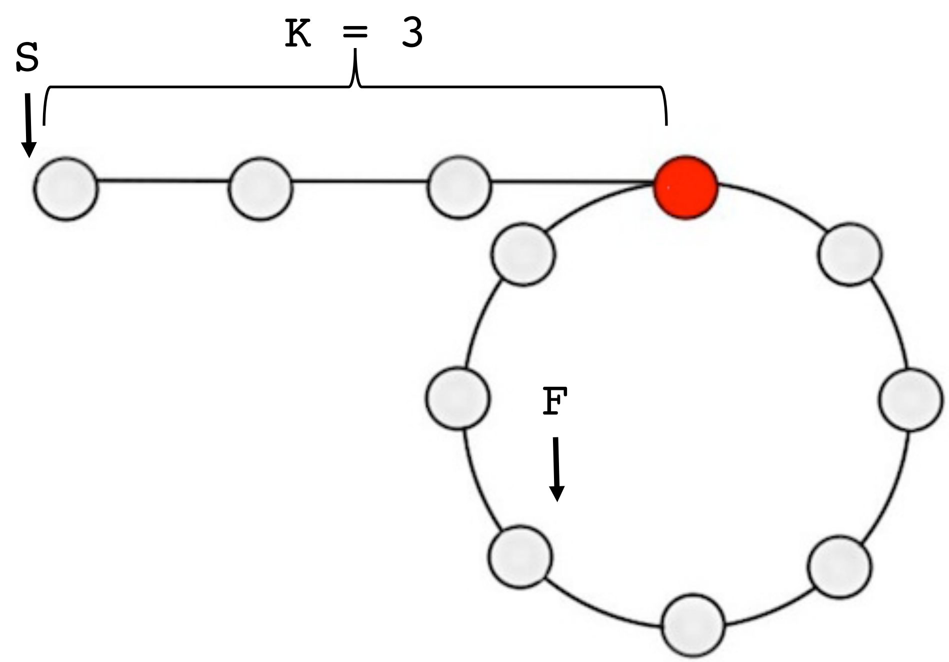
- `fast_runner` moves two steps at a time, and `slow_runner` moves one step at a time
- Since the speed difference of two pointers is 1, then after 5 steps (`LS – (k mod LS)`) the pointers meet
- This means that the `slow_runner` has moved for 5 steps (`LS – (k mod LS)`)
- The number of steps left to the start of the loop is
  `LS – (LS – (k mod LS))`
  `= k mod LS`

`In our example:`
`3 mod 8 = 3`



Loop Detection

- Since we don't know what is k, we make the `slow_runner` point to the list head, and then both pointers move 1 step at a time
- The start of the loop is where the two pointers meet



Loop Detection

- **The pseudocode is:**

1. Declare two pointers `slow_runner` and `fast_runner`, where both point to the head of the linked list

2. `while (fast_runner != NULL && fast_runner -> link() != NULL)`
   - `slow_runner` moves one step at a time, and `fast_runner` moves two steps at a time
   - Break the loop if `slow_runner == fast_runner`

3. Return NULL if the two pointers did not meet (i.e., `fast_runner == NULL || fast_runner -> link() == NULL)`

4. `while (slow_runner != fast_runner)`
   - Both pointers move one step at a time.

Using the Linked-List Toolkit

- **Node Class + Toolkit**
  - Allow a container class to store elements on a basic linked list with the simplicity and clarity of using an array
- Any programmer can use our node and the toolkit
  - The programmer defines the `value_type` according to his or her need
  - Places the `#include "node.h”` in the program

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

- We will implement the new bag by storing the items in a linked list
- The class will have **two private member variables:**
  - **A head pointer** that points to the head of a linked list that contains the items of the bag
  - **A variable** that keeps track of the length of the list
    - Why we use a variable to keep track of the size of the list instead of using the `list_length` function?

Our Third Bag — Class Definition (Cont’d)

```
#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
};
```

The Bag Class with a Linked List

How to Make the `bag's value_type` Match the `node's value_type` (Cont'd)

- **If we need a different type of item**, we can change **`node::value_type`** to the required new type and recompile
  - ❑Example: Suppose we want a bag of strings using the Standard Library's string class (from `<string>`)
- In order to obtain the bag of strings, the start of our **node** definition will be

```
#include <string>
class node
{
public:
    typedef std::string value_type;
    ...
```

- In this case, the `node::value_type` is defined as the `string` class, so that `bag::value_type` will also be a string

The Bag Class with a Linked List

Our Third Bag — Header File Implementation

```
#ifndef SCU_COEN79_BAG3_H
#define SCU_COEN79_BAG3_H

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

namespace scu_coen79_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( );
```

Our Third Bag — Header File Implementation (Cont'd)

```
// 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;
value_type grab( ) const;
```

Our Third Bag — Header File Implementation (Cont'd)

```
    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 (indicating the empty list) 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;
 }
```

As our implementation does not have a private member for tail pointer, we need to define a local pointer

The Third Bag Class — Implementation (Cont'd)

- **Overloading the assignment operator.** It needs to change an existing bag so that it is the same as some other bag
- **Note:** When the assignment operator begins, the bag already has a linked list, and this linked list must be returned to the heap

```
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;

  }
```

To check for self assignment

To return the existing linked list to the heap

we want the bag to be valid before calling `list_copy`

The Assignment Operator Causes Trouble with Linked Lists

- When a data structure uses a linked list, the assignment operator is important
- Two important aspects:
  - We need to check for "**self-assignment**" such as `b = b`
    - The easiest way to handle self-assignment is to check for it at the start of the assignment operator and simply return with no work if self-assignment is discovered
  - In addition, before calling a function that allocates dynamic memory, **make sure that the invariant of your class is valid**

The Third Bag Class — Implementation (Cont'd)

- **The destructor**
  - Our documentation, which is meant for other programmers, never mentioned a destructor
  - However, a destructor is needed because our particular implementation uses dynamic memory
- To return all dynamic memory to the heap, we use `list_clear`:

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

Returns all nodes to the heap and sets `head_ptr` to NULL

- 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'd)

- **The `erase_one` member function.**
- There are two approaches to implement this function:
  1. Using the toolkit's removal function:
     - Use:
       - `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)

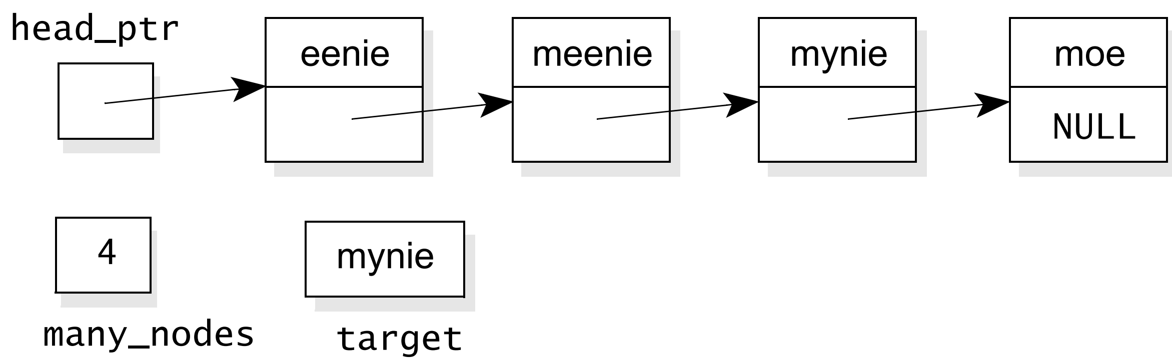
  2. Using `list_search` to obtain a pointer to the node that contains the item to be deleted
     - **We chose this approach because it made better use of `list_search`**

The Third Bag Class — Implementation (Cont'd)

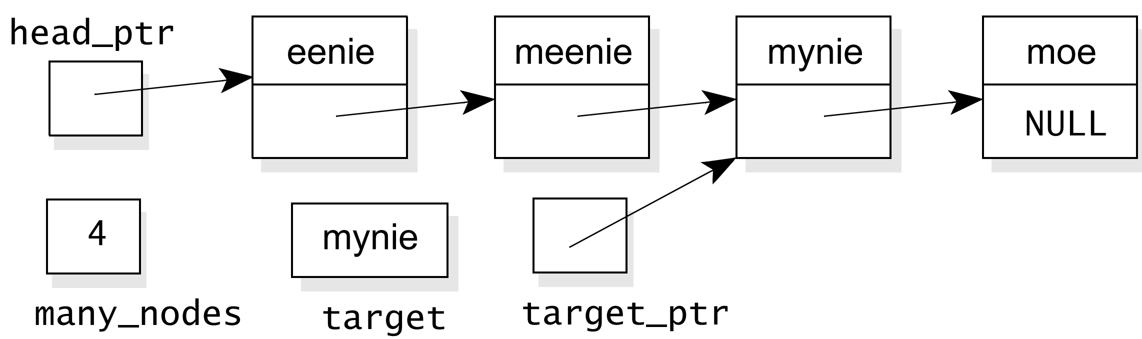
- **The `erase_one` member function using `list_search`**

❑Example

Suppose our target is the string `mynie`



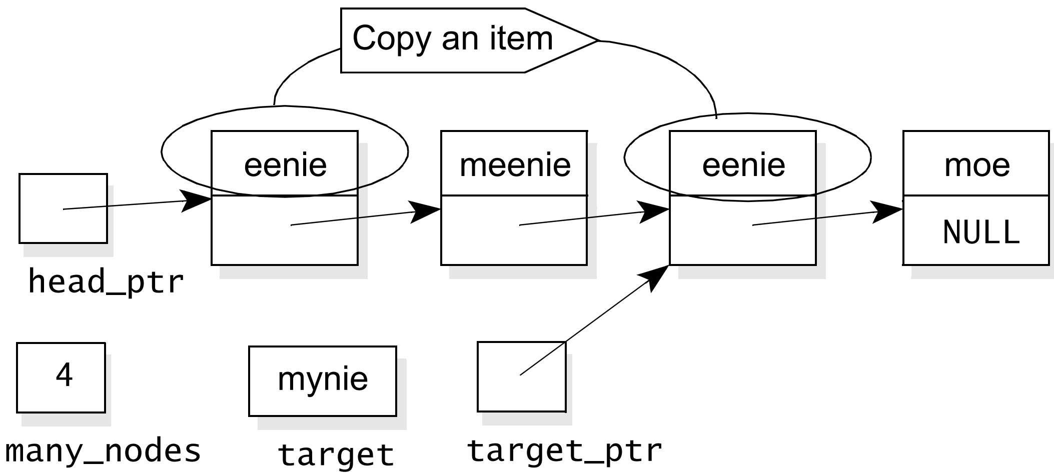
- `target_ptr` points to the node that contains our target
- `target_ptr = list_search(head_ptr, target)`



The Third Bag Class — Implementation (Cont'd)

- We remove the target from the list with two more steps:

1. 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:

2. Use `list_head_remove` to remove the head node (that is, one of the copies of `eenie`)

The Third Bag Class — Implementation (Cont'd)

- 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'd)

- **The `count` member function.** Counts the occurrences of the target and returns the answer
- Two possible approaches:
  - Step through the linked list one node at a time
  - Use `list_search` to find the first occurrence of the target, then use `list_search` again to find the next occurrence, and so on until we have found all occurrences of the target
    - **We chose this approach because it made better use of `list_search`**

The Third Bag Class — Implementation (Cont'd)

```
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'd)

- **The `grab` member function.**

```
value_type grab( ) const;
// Precondition: size( ) > 0.
// Postcondition: The return value is a randomly selected item from
// the bag.
```

- The implementation starts by generating a random integer between 1 and the size of the bag
- The random integer can then be used to select a node from the bag, and we will return the data from the selected node

The Third Bag Class — Implementation (Cont'd)

- The trick is to generate a random integer between 1 and the size of the bag

```
i = some random integer between 1 and the size of the bag;
cursor = list_locate(head_ptr, i);
return cursor->data( );
```

- The `rand` function from the C++ Standard Library can help:

```
int rand( );
// Postcondition: The return value is a non-negative pseudorandom
// integer
```

The Bag Class with a Linked List

The Third Bag Class — Implementation (Cont'd)

```
bag::value_type bag::grab( ) const
{
  size_type i;
  const node *cursor;    // Use const node* since we don't change
                         // the nodes

  assert(size( ) > 0);

  i = (rand( ) % size( )) + 1;

  cursor = list_locate(head_ptr, i);

  return cursor->data( );
}
```

- **Returns a pseudo-random integral number in the range between 0 and RAND_MAX**
- **RAND_MAX is a constant defined in <cstdlib>**

The Bag Class with a Linked List

The Third Bag Class — Implementation (Cont'd)

operator +=

- The implementation starts by making a copy of the linked list of the addend
- The copy is then 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;
    }
 }
```

Dynamic Arrays
vs.
Linked Lists
vs.
Doubly Linked Lists

Dynamic Arrays vs. Linked Lists vs. Doubly Linked Lists

- Many classes can be implemented with either dynamic arrays or linked lists
- **Which approach is better?**

Dynamic Arrays vs. Linked Lists vs. Doubly Linked Lists

Dynamic Array

- **Arrays are better at random access**
- The term **random access** refers to examining or changing an arbitrary element that is **specified by its position** in a list

❑ For example:
  - What is the 2nd item in the list?
  - Change the item at position 16 to a 4
- **These are constant time operations for an array (or dynamic array)**
- In a linked list, a search for an item must begin at the head and will take $O(n)$ time

Dynamic Arrays vs. Linked Lists vs. Doubly Linked Lists

Dynamic Array

- **Resizing can be inefficient for a dynamic array, because:**
  - New memory must be allocated
  - The items are then copied from the old memory to the new memory
  - The old memory is deleted
- When the eventual capacity is unknown and a program must continually adjust the capacity, a linked list has advantages
- If a class is frequently adjusting its size, then a linked list may be better than a dynamic array

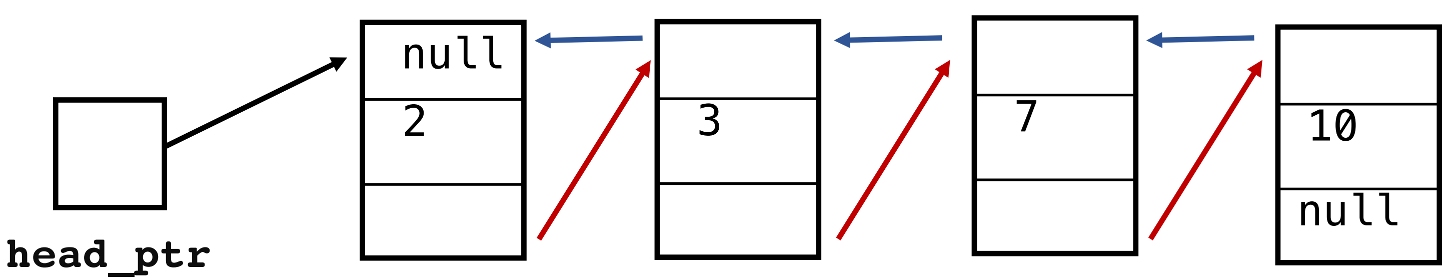
Dynamic Arrays vs. Linked Lists vs. Doubly Linked Lists

Linked lists

- **Linked lists are better at insertions/deletions at a cursor**
- If class operations take place at a cursor, then a linked list is better than a dynamic array
- Insertions and deletions at a cursor are generally linear time ($O(n)$) for an array (since items that are after the cursor must be shifted)
- But these operations are constant time operations ($O(1)$) for a linked list

Doubly Linked Lists

- **Doubly linked lists are better for a two-way cursor**
- Sometimes list operations require a cursor that can move forward and backward through a list
  - A **two-way cursor**
- **Doubly linked list** is like a simple linked list, **except that each node contains two pointers**: one pointing to the next node and one pointing to the previous node



Dynamic Arrays vs. Linked Lists vs. Doubly Linked Lists

Doubly Linked Lists (Cont'd)

- A possible definition for a doubly linked list of items

```
class dnode
{
public:
    typedef _____ value_type;
    ...

private:
    value_type data_field;
    dnode *link_fore;
    dnode *link_back;
};
```

- `link_back` field points to the previous node
- `link_fore` points to the next node in the list

Dynamic Arrays vs. Linked Lists vs. Doubly Linked Lists

A Class for a Node in a Doubly Linked List

A node class with double links

```
#ifndef SCU_COEN79_DNODE_H
#define SCU_COEN79_DNODE_H
#include <cstdlib>    // Provides size_t and NULL
namespace scu_coen79_5
{
    class dnode
    {
    public:

        // TYPEDEF
        typedef double value_type;

        // CONSTRUCTOR
        dnode( const value_type& init_data = value_type( ),
                 dnode* init_fore = NULL, dnode* init_back = NULL )
            {
                data_field = init_data;
                link_fore = init_fore;
                link_back = init_back;
            }
```

A Class for a Node in a Doubly Linked List (Cont'd)

```
// Member functions to set the data and link fields:
void set_data(const value_type& new_data)
                       { data_field = new_data; }
void set_fore(dnode* new_fore)
                       { link_fore = new_fore; }
void set_back(dnode* new_back)
                       { link_back = new_back; }


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


// Two slightly different member functions to retrieve each
// current link:
const dnode* fore( ) const { return link_fore; }
dnode* fore( )             { return link_fore; }
const dnode* back( ) const { return link_back; }
dnode* back( )             { return link_back; }
```

Dynamic Arrays vs. Linked Lists vs. Doubly Linked Lists

A Class for a Node in a Doubly Linked List (Cont'd)

```
    private:
        value_type data_field;
        dnode *link_fore;
        dnode *link_back;
    };

}

#endif
```

Guidelines for Choosing Between a Dynamic Array and a Linked List ☆

Case	Data Structure
Frequent random access operations	Use a dynamic array
Operations occur at a cursor	Use a linked list
Operations occur at a two-way cursor	Use a linked list
Frequent resizing may be needed	Use a linked list

Summary

Summary

- A **linked list** consists of nodes
  - Each **node** contains data and a pointer to the next node in the list
- A linked list is accessed through a **head pointer** that points to the *head node*
- A linked list may have a **tail pointer** that points to the last node
- A **doubly linked list** has nodes with two pointers: one to the next node and one to the previous node
  - Enables a cursor to move forward and backward
- Containers can be implemented in many different ways, such as by using a dynamic array or using a linked list
- Arrays are better at **random access**
- Linked lists are better at **insertions/removals at a cursor**

References

1) Data Structures and Other Objects Using C++, Michael Main, Walter Savitch, 4th Edition

2) Data Structures and Algorithm Analysis in C++, by Mark A. Weiss, 4TH Edition

3) C++: Classes and Data Structures, by Jeffrey Childs

4) http://en.cppreference.com

5) http://www.cplusplus.com

6) https://isocpp.org

Copyright Notice