

Ch 5

Linked Lists

- A Node Class for Linked Lists
- A Linked List Toolkit
- The Bag Class with a Linked List
- Programming Project: The Sequence Class
- Efficiency Considerations

The Node Class

Elements of Containers Implemented as Linked
Structures

The NULL Pointer

- Use the **NULL** pointer for the link field of the final node of a linked list.
- An empty list's head pointer should be **NULL**.
- Null pointers may be assigned to any pointer variable:

```
node *head_ptr;  
head_ptr = NULL;
```

The Node Constructor

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

```
node(  
    const value_type& data = value_type(),  
    const node *link = NULL);
```

The `value_type` constructor is even valid for primitives.

Indirect Selection Operator

One way to access a member through an object pointer is to dereference the pointer and apply the membership operator:

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

Another way is to simply apply the indirect selection operator:

```
cout << head_ptr->data();
```

More on Pointers & Constants

- A pointer to a constant may not modify the data to which it points.
- A constant pointer may not be redirected to different data.
- Pointers to constants may only activate constant member functions.

A Bad Example

This member function is unsafe:

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

It would allow the following:

```
node *np = head_ptr->link();  
np->set_data(9.2);
```

Which modifies not the first, but the second node!

The Fix

Since C++ allows constant pointers to only access constant member functions, one can write a version of `link()` that returns a pointer to a constant object:

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

This would be supplied in addition to the non-constant version:

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

Node Class Definition

```
class node {  
    public:  
        typedef double value_type;  
    node(  
        const value_type& init_data = value_type( ),  
        node* init_link = NULL) {  
        data_field = init_data;  
        link_field = init_link; }  
  
    void set_data(const value_type& new_data) {  
        data_field = new_data; }  
    void set_link(node* new_link) {  
        link_field = new_link; }  
  
    value_type data( ) const {  
        return data_field; }  
    const node* link( ) const {  
        return link_field; }  
    node* link( ) {  
        return link_field; }  
    private:  
        value_type data_field;  
        node* link_field;  
};
```

The Linked List Toolkit

- The prototypes are contained in the node class' interface file.
- The functions include
 - computing the length of the list,
 - insertion at the head and general insertion,
 - deletion at the head and general deletion,
 - searching by target and position,
 - copying,
 - clearing the list.

The Toolkit Prototypes

```
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);
```

Implementation: Length & Insert

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

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

void list_insert(node* previous_ptr, const node::value_type& entry) {
    node *insert_ptr;
    insert_ptr = new node(entry, previous_ptr->link( ));
    previous_ptr->set_link(insert_ptr);
}
```

Implementation: Remove and Clear

```
void list_head_remove(node*& head_ptr) {
    node *remove_ptr;
    remove_ptr = head_ptr;
    head_ptr = head_ptr->link( );
    delete remove_ptr;
}

void list_remove(node* previous_ptr) {
    node *remove_ptr;
    remove_ptr = previous_ptr->link( );
    previous_ptr->set_link( remove_ptr->link( ) );
    delete remove_ptr;
}

void list_clear(node*& head_ptr) {
    while (head_ptr != NULL)
        list_head_remove(head_ptr);
}
```

Implementation: Searching

```
node* list_locate(node* head_ptr, size_t position) {
    node *cursor;
    size_t i;
    assert (0 < position);
    cursor = head_ptr;
    for (i = 1; (i < position) && (cursor != NULL); i++)
        cursor = cursor->link( );
    return cursor;
}

const node* list_locate(const node* head_ptr, size_t position) {
    const node *cursor;
    size_t i;
    assert (0 < position);
    cursor = head_ptr;
    for (i = 1; (i < position) && (cursor != NULL); i++)
        cursor = cursor->link( );
    return cursor;
}
```

Implementation: Copying

```
void list_copy(const node* source_ptr, node*& head_ptr, node*& tail_ptr) {
    head_ptr = NULL;
    tail_ptr = NULL;

    if (source_ptr == NULL)
        return ;

    list_head_insert(head_ptr, source_ptr->data( ));

    tail_ptr = head_ptr;
    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( ); }
}
```

The Bag Class

Implemented as a Linked List

Bag Class Invariant

- The items are stored in a linked list.
- The head pointer of the list is stored in the member variable **head_ptr**.
- The total number of items in the list is stored in the member variable **many_nodes**.

Bag Class interface

```
class bag {  
  public:  
    typedef std::size_t size_type;  
    typedef node::value_type value_type;  
  
    bag( );  
  
    bag(const bag& source);  
    void operator =(const bag& source);  
    ~bag( );  
  
    size_type erase(const value_type& target);  
    bool erase_one(const value_type& target);  
    void insert(const value_type& entry);  
    void operator +=(const bag& addend);  
  
    size_type size( ) const {  
      return many_nodes; }  
    size_type count(const value_type& target) const;  
    value_type grab( ) const;  
  private:  
    node *head_ptr;  
    size_type many_nodes;  
};  
  
bag operator +( const bag& b1, const bag& b2);
```

Implementation:

Construction & Destruction

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

bag::bag(const bag& source) {
    node *tail_ptr;
    list_copy(source.head_ptr, head_ptr, tail_ptr);
    many_nodes = source.many_nodes;
}

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

Implementation: Count & Insert

```
bag::size_type bag::count(const value_type& target) const {
    size_type answer = 0;
    const node *cursor = list_search(head_ptr, target);

    while (cursor != NULL) {
        ++answer;
        cursor = cursor->link( );
        cursor = list_search(cursor, target); }

    return answer;
}

void bag::insert(const value_type& entry) {
    list_head_insert(head_ptr, entry);
    ++many_nodes;
}
```

Implementation: Erase

```
size_type bag::erase(const value_type& target) {
    size_type answer = 0;
    node *target_ptr;
    target_ptr = list_search(head_ptr, target);
    while (target_ptr != NULL)
        target_ptr->set_data( head_ptr->data( ) );
        target_ptr = target_ptr->link( );
        target_ptr = list_search(target_ptr, target);
        list_head_remove(head_ptr);
        --many_nodes;
        ++answer; }
    return answer;
}

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

Implementation: Grab

```
bag::value_type bag::grab( ) const {
    size_type i;
    const node *cursor;

    assert(size( ) > 0);

    i = (rand( ) % size( )) + 1;
    cursor = list_locate(head_ptr, i);
    return cursor->data( );
}
```

Implementation: Operators

```
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; }
}

void bag::operator =(const bag& source) {
    node * tail_ptr;
    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;
}

bag operator +(const bag& b1, const bag& b2) {
    bag answer;
    answer += b1;
    answer += b2;
    return answer;
}
```

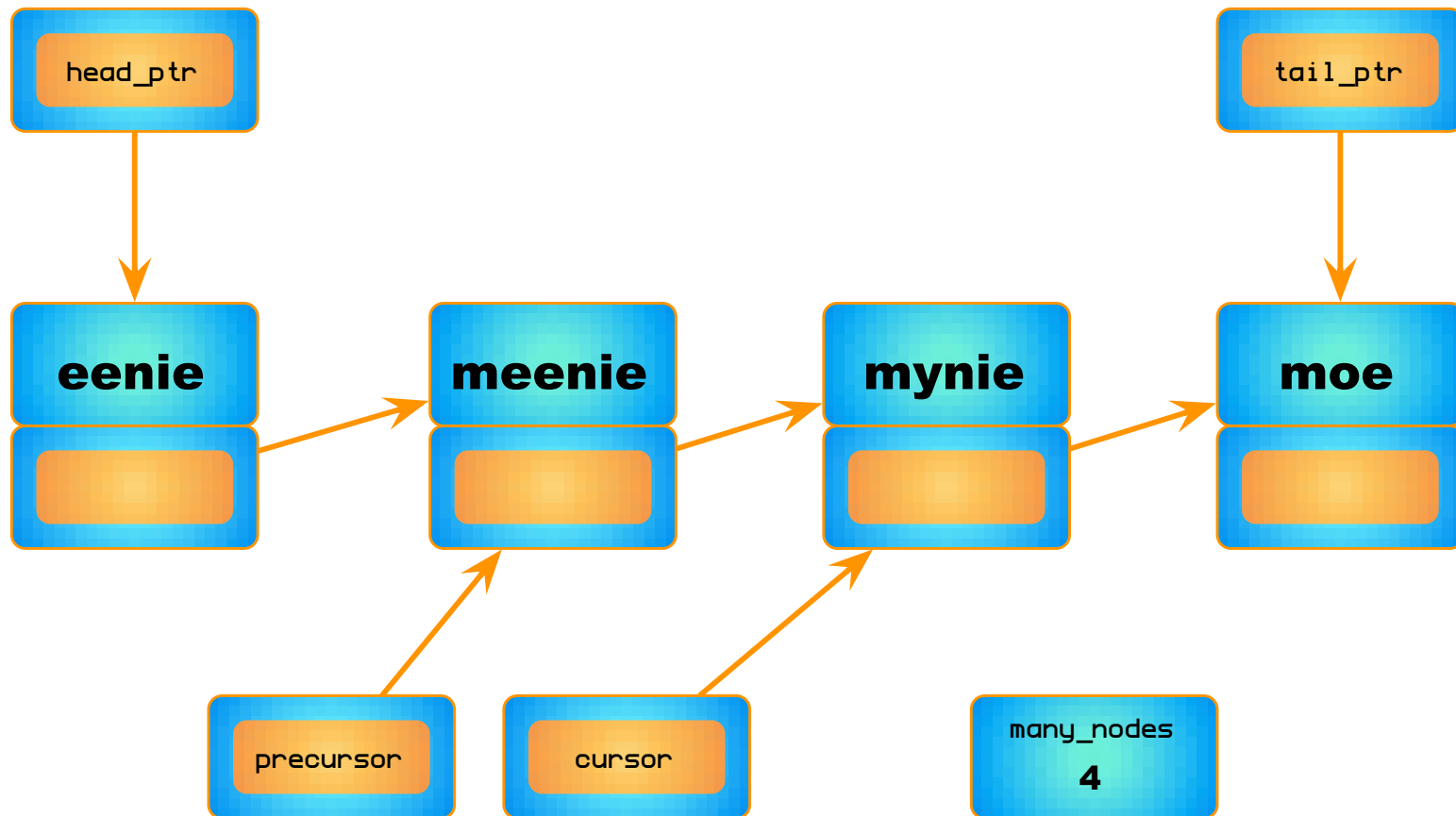
The Sequence Class

Implemented Using a Linked List

Private Member Variables

- A `head_ptr` to point to the beginning of the sequence.
- A `tail_ptr` to facilitate the attach function if there is no current node.
- A `cursor` to point to the current node.
- A `precursor` to point to the node before the current item and facilitate the insert function.
- A counter, `many_nodes`, to keep track of the size (cardinality) of the sequence.

An Example Sequence



Design Considerations

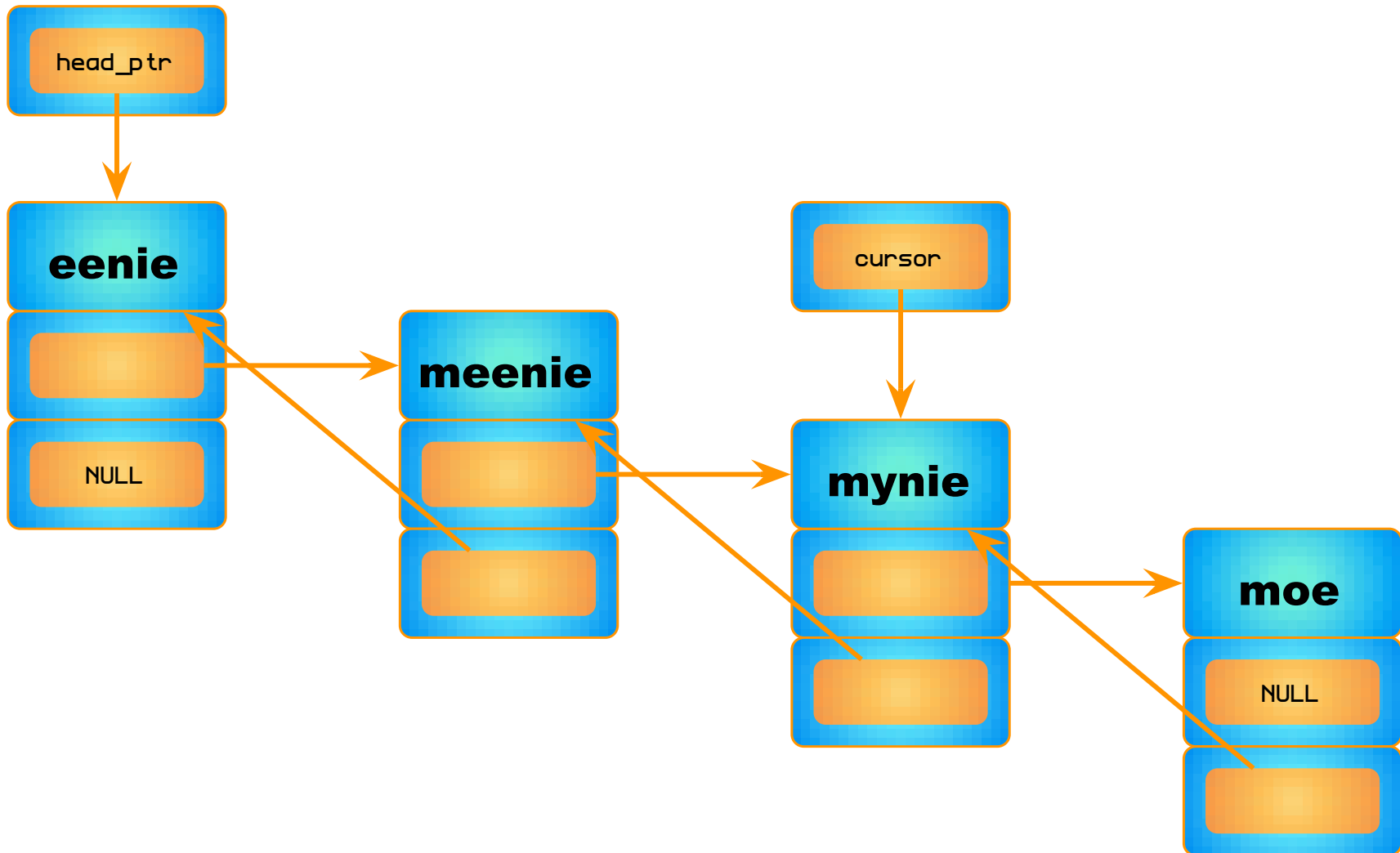
- Start by coding the interface file.
- Establish and document the class invariant.
- Document those functions that allocate dynamic memory.
- Override the automatic copy constructor and assignment operator. Write a destructor.

Assignment and Copying

- Check for self-assignment.
- If the source has no current item, use `list_copy()` and set `cursor` and `precursor` to `NULL`.
- If source's `cursor` is at the first item, copy with `list_copy()`, set `precursor` to `NULL` and `cursor` to `head_ptr`.
- If source's `cursor` is after the first item, copy in two pieces: from `head_ptr` to `precursor`, then from `cursor` to `tail_ptr`.
- Set `many_nodes` of the destination appropriately.

Efficiency Considerations

Doubly-Linked Lists



Arrays and Linked Lists

- Arrays support random access in constant time, linked lists in linear time.
- For insert and delete at a cursor, linked lists support constant time, arrays linear time.
- If bidirectional movement is needed in a linked list, implement as a doubly-linked list.
- If frequent resizing is necessary, avoid array implementations.

Project D

Complete Project 12e from Chapter 5 of the text. Directions are on page 277.

- Name your **class** `keyed_bag3`. You will submit its definition in `keyed_bag3.h` and the implementation in `keyed_bag3.cpp`.
- Use the **node class** interface file [available here](#) instead of the one from the book. The *linked list toolkit* from Chapter 5 of the text will work unmodified. Copy all of the source files into the same directory as the rest of your project. Be careful about using appropriate namespaces. You will not submit the authors' files.
- Include member functions whose prototypes are:

```
keyed_bag3();
keyed_bag3(const keyed_bag3& source);
~keyed_bag3();
keyed_bag3& operator=(const keyed_bag3& source);
void insert(const value_type& entry, int key);
bool erase(int key);
value_type get(int key) const;
size_type size() const;
size_type count(const value_type& target) const;
bool has_key(int key) const;
```

- If an attempt is made to `insert()` an entry with a duplicate key, the existing entry should be replaced by one with the new value.
- Do not create a **namespace** for your **class**.
- Test your **class** thoroughly with your own interactive test program similar to the one presented on pp. 133-135 of the text. You will not submit the test program.

```
class record_type {  
    public:  
        typedef double value_type;  
        void setKey(int k) { key = k; }  
        int getKey() const { return key; }  
        void setData(value_type d) { data = d; }  
        value_type getData() const { return data; }  
        bool operator ==(record_type rv) const {  
            return (key == rv.key) && (data == rv.data); }  
    private:  
        int key;  
        value_type data;  
};
```