Linked Lists

The Bag Class

Data Structures

There are many possible ways to implement a given data structure

So far, we've seen a bag class implemented using:

- a static array
- a dynamic array

Now we'll it implemented using a linked list

How will the specification change?

- the specification is for the user of the class, and they don't need to know that the bag is implemented using a linked list
- one notable change, though, is that we no longer need to worry about the capacity of the bag, since linked lists can easily grow and shrink by adding or removing nodes

Updated Specification

Updated Specification

Type definitions and member constants:

```
    // data type of items in the bag
    typedef _________ value_type;


    // value_type must be a built-in type, or support:
    //  - instantiation via a default constructor
    //  - instantiation via a copy constructor
    //  - assignment operator    (x = y)
    //  - equality operator      (x == y)
```

Updated Specification

Type definitions and member constants:

```
// data type of variables that track a bag's size
typedef _________ size_type;
```

Updated Specification

Constructors:

```
    // creates an empty bag
    bag();


    // postcondition:
    //   the bag is initialized as an empty bag
```

Updated Specification

Modification member functions:

```
// removes all copies of @target from the bag
// returns the number of elements removed
size_type erase(const value_type& target);
```

```
// postcondition:
//   all copies of target have been removed from the bag
//   return value is the number of elements removed
```

Updated Specification

Modification member functions:

```
// removes a single copy of @target from the bag
// returns true if a value was removed; false otherwise
bool erase_one(const value_type& target);
```

```
// postcondition:
//   if @target was in the bag, then one copy is removed
//   otherwise, the bag remains unchanged
//   return value is true if a value was removed,
//     or false otherwise
```

Updated Specification

Modification member functions:

```
// inserts a new copy of @entry into the bag
void insert(const value_type& entry);


// postcondition:
//   a new copy of @entry has been added to the bag
```

Updated Specification

Modification member functions:

```
// inserts a copy of each item in @addend into the bag
void operator +=(const bag& addend);


// postcondition:
//   each item in addend has been added to the bag
```

Updated Specification

Constant member functions:

```
    // returns the total number of items in the bag
    size_type size() const;


    // postcondition:
    //   return value is the number of items in the bag
```

Updated Specification

Constant member functions:

```
    // returns the total number of occurrences of @target
    size_type count(const value_type& target) const;

    // postcondition:
    //    return value is the number of times @target occurs
    //      in the bag
```

Updated Specification

Non-member functions:

```
// returns a new bag that is the union of @b1 and @b2
bag operator +(const bag& b1, const bag& b2);


// postcondition:
//    the bag returned is the union of b1 and b2
```

Updated Specification

Value semantics:

```
// bag objects may be:
//   assigned using operator =
//   copied via the copy constructor
```

Updated Specification

Dynamic memory usage:

```
// If there is insufficient dynamic memory, then the
// following functions throw bad_alloc:
//   constructors
//   insert
//   operator +=
//   operator +
//   operator =
```

Updated Specification

That's it!

- notice that we no longer have a reserve function or a DEFAULT_CAPACITY constant anymore
- also, the dynamic memory specification changed to reflect the functions that allocate memory

Updated Implementation

Updated Implementation

Starting the header file:

```
// bag.h header file

// specification documentation

#pragma once

#include <cstdlib>

#include "Node.h" // include Node class


namespace CS262 {

    class bag {

        // bag class declaration

    };

}
```

Updated Implementation

Starting the implementation file:

```
// bag.cpp implementation file

// INVARIANT (coming soon)

#include "bag.h"

#include <algorithm>

#include <cassert>

using namespace std;


namespace CS262 {

    // member implementations

}
```

Updated Implementation

Type definitions and member constants:

```
class bag {
    public:
        typedef Node::value_type value_type;
        typedef std::size_t size_type;

    private:
        Node* head;
        size_type node_count;
};
```

Updated Implementation

Here's the updated private section:

```
private:
    Node* head;                 // start of the list
    size_type node_count; // number of nodes in list
```

Compare that with the dynamic array version:

```
private:
    value_type* data;
    size_type capacity;
    size_type used;
```

Updated Implementation

Here's the updated private section:

```
private:
    Node* head;                 // start of the list
    size_type node_count; // number of nodes in list
```

Compare that with the static array version:

```
private:
    value_type data[CAPACITY];
    size_type used;
```

Updated Implementation

Here's the updated private section:

```
private:
    Node* head;                 // start of the list
    size_type node_count; // number of nodes in list
```

Our implementation now uses a linked list...

- items will be stored in nodes instead of in arrays
- of course, we need to keep track of the head of the list
- though not technically necessary, we will also track the current size of the list to avoid having to recount the number of nodes every time

Updated Implementation

Document invariant in implementation file:

```
// bag.cpp implementation file
// INVARIANT for bag class:
// 1. The items in the bag are stored in a linked list
// 2. The head pointer of the list is stored in the
//     member variable head.
// 3. The total number of items in the list is stored in
//     the member variable node_count
```

Updated Implementation

The constructor:

```
// creates an empty bag
bag() : head(NULL), node_count(0) { }
```

Updated Implementation

The size method:

```
// returns the total number of items in the bag
size_type size() const { return node_count; }
```

Updated Implementation

The erase_one function prototype:

```
// removes a single copy of @target from the bag
// returns true if a value was removed; false otherwise
bool erase_one(const value_type& target);
```

This function should:

- locate the value in the list, if it exists
- swap the value of the head node with this one, then delete the head node (think about it)
- each node is dynamically allocated, so it needs to be deleted
- update the value of node_count

Updated Implementation

The erase_one function implementation:

```
bool bag::erase_one(const value_type& target) {
    Node* n = list_search(head, target);

    if (n == NULL) return false;
    n->set_data( head->data() );
    list_head_remove(head);

    node_count--;
    return true;
}
```

Updated Implementation

The erase function prototype:

```
// removes all copies of @target from the bag
// returns the number of elements removed
size_type erase(const value_type& target);
```

This function should:

- perform the same logic as the erase_one method, but for each target value in the list
- each node is dynamically allocated, so it needs to be deleted
- update the value of node_count

Updated Implementation

The erase function implementation:

```
bag::size_type bag::erase(const value_type& target) {
    size_type removed_count = 0;
    Node* target_ptr = list_search(target_ptr, target);

    while (target_ptr != NULL) {
        target_ptr->set_data( head->data() );
        target_ptr = target_ptr->link();
        target_ptr = list_search(target_ptr, target);
        node_count++;
        removed_count++;
    }
    return removed_count;
}
```

Updated Implementation

The insert function prototype:

```
// inserts a new copy of @entry into the bag
void insert(const value_type& entry);
```

This function should:

- insert a new node at the beginning of the list
- update the value of node_count

Updated Implementation

The insert function implementation:

```
void insert(const value_type& entry) {
    list_head_insert(head, entry);
    node_count++;
}
```

Updated Implementation

The count function prototype:

```
// returns the total number of occurrences of @target
size_type count(const value_type& target) const;
```

This function should:

- count the number of nodes whose data is equal to target

Updated Implementation

The count function implementation:

```
    // returns the total number of occurrences of @t
    bag::size_type bag::count(const value_type& t) const {
        size_type num_nodes = 0;
        const Node* ptr = head;
        while ((ptr = list_search(ptr, t)) {
            num_nodes++;
            ptr = ptr->link()l
        }
        return num_nodes;
}
```

Updated Implementation

The operator += function prototype:

```
// inserts a copy of each item in @addend into the bag
void operator +=(const bag& addend);
```

This function should:

- copy the items from the addend bag into the current bag
- update the value of node_count

Updated Implementation

The operator += function implementation:

```
// inserts a copy of each item in @addend into the bag
void bag::operator +=(const bag& addend) {
    const Node* n;
    for (n = addend.head; n != NULL; n = n->link()) {
        list_head_insert(head, n->data());
    }

    node_count += addend.node_count;
}
```

Updated Implementation

The operator + function prototype:

```
// returns a new bag that is the union of @b1 and @b2
bag operator +(const bag& b1, const bag& b2);
```

This function should:

- create a new bag whose contents are the union of the two arguments
- can simply create a new bag and simply use operator +=

Updated Implementation

The operator + function implementation:

```
    // returns a new bag that is the union of @b1 and @b2
    bag operator +(const bag& b1, const bag& b2);
        bag union;
        union += b1;
        union += b2;

        return union;
}
```

Updated Implementation

The bag class again uses dynamic memory...

We need to implement the big 3:

- copy constructor
- assignment operator
- destructor

Updated Implementation

The copy constructor prototype:

```
// creates a new bag as a copy of @source
bag(const bag& source);
```

The copy constructor must:

- copy all nodes from the source list to the new one
- update the value of node_count

Updated Implementation

The copy constructor implementation:

```
// creates a new bag as a copy of @source
bag::bag(const bag& source) {
    const Node* n;
    for (n = source.head; n != NULL; n = n->link()) {
        list_head_insert(head, n->data());
    }

    node_count = source.node_count;
}
```

Updated Implementation

The assignment operator prototype:

```
// assigns this bag as a copy of @source
bag& operator =(const bag& source);
```

The assignment operator must:

- check for self-assignment
- clear the current list
- copy all nodes from the source list
- update the value of `node_count`
- ideally, it should also return the calling object by reference (for assignment chaining)

Updated Implementation

The assignment operator implementation:

```
bag& bag::operator =(const bag& source) {
    if (this == &source) return *this;

    const Node* n;
    list_clear(head);
    for (n = source.head; n != NULL; n = n->link()) {
        list_head_insert(head, n->data());
    }
    node_count = source.node_count;
    return *this;
}
```

Updated Implementation

The destructor prototype:

```
// frees the memory used by this bag
~bag();
```

The destructor must:

- deallocate any dynamically allocated memory used by the bag

Updated Implementation

The destructor implementation:

```
// frees the memory used by this bag
bag::~bag() {
    list_clear(head);
}
```

Why use a linked list?

Arrays vs Linked Lists

Many classes can use either arrays or linked lists...

- however, there is no "correct" answer
- each has strengths and weaknesses

A comparison:

- arrays are better at random access (O(1) vs O(n) for linked lists)
- linked lists are better at insertions / deletions at arbitrary locations (O(1) vs O(n) for arrays)
- doubly-linked lists are better for bidirectional tasks where insertion and deletion at arbitrary points are frequent
- frequent resizes are inefficient for arrays; linked lists can resize in constant time