Linked lists (cont.)

Basic functionality of linked lists
How to implement a bag structure

Read chapter 16 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

Basic functionality of linked lists

- Programmers should take great care to ensure that `NULL` pointers are correctly handled
- Functions for manipulating linked lists are usually not node member functions, because they should be able to handle empty lists, for which head and tail are `NULL`

```
class LinkedList
{
...
private:
   node* head;
   node* tail;
   node* current;
...
};
```

Counting the number of elements

- The method `list_length()` returns the number of nodes (and hence stored items) in a linked list

```
int LinkedList::list_length(); // this goes in the .h file
```

- The method maintains a counter that is incremented as nodes are traversed

```
int LinkedList::list_length() // this goes in the .cpp file
// Precondition: None
// Postcondition: A count of the nodes in the list is returned. Uses cstdlib.
{
    int answer = 0;
    for (current = head; current != NULL; current = current->get_next()) {answer++;}
    return answer;
}
```

- Make sure your code works properly for the empty list!

Inserting at the head of the list

- To add an instance of `value_type` at the head of a list pointed to by `head`
  - Create a new node with the data item and head of list specified as parameters
  - Change the value of `head` to point to the new node

```
void LinkedList::add_to_head(const node::value_type& entry)
// Precondition: None
// Postcondition:  A new node storing the supplied entry is
// created and linked in to be the new head of the linked list
{
    head = new node(entry, head);
    if (tail == NULL) {tail = head;} // this is required to deal with
                                     // adding to an empty list.
}
```

Inserting an internal node

- This insertion requires a pointer to the node just before the location of the new node. We store a pointer to this node in `current`. The steps are:
  - Create a new instance of `node` and store the data item
  - Make the `next` of the new node be the `next` of `current`
  - Point the `next` of `current` to the new node instance
  - Update `current` so that next item is inserted after the new node
  - Check if `tail` needs to change

```
void LinkedList::add_current(const node::value_type& entry)
// Precondition: current points to the node just before the insertion position
// Postcondition: A new node containing entry is inserted after the node pointed
// to by current; current points to the new node
{
    node* add_ptr = new node;
    add_ptr->set_data(entry);
    add_ptr->set_next(current->get_next());
    current->set_next(add_ptr);
    if (current == tail) {tail = current->get_next();}
    current = current->get_next();
}
```

Inserting an internal node

- In the function `add_current()` a local variable `add_ptr` is used
- You may be tempted to use `delete add_ptr` to make sure you have not created a memory leak
- Don't do that! This will remove the node that `add_ptr` points to, which is the node you just created!
- Only call `delete` if you need to reduce the number of nodes in the list

- Similarly, students sometimes use `new node();` when setting up a pointer to be used locally in a method
- This creates a new instance of node, which in some cases in unnecessary
- Only use `new` if you need to increase the number of nodes in the list

Searching through a linked list

- This is achieved by traversing the list, checking the item stored at each node

```
boolean LinkedList::list_search(const node::value_type& target)
// Preconditions: None
// Postconditions: Current points to the first node storing the target, and true is
// returned. If not present, current is NULL and false is returned. Uses cstdlib.
{
    for (current = head; current != NULL; current = current->get_next())
    {
        if (target == current->get_data())
            return true;
    }
    return false;
}
```

Removing a node

- Removing a `node` must be done carefully
- It is not possible to remove the `current` node in a singly-linked list, because we need to link the previous node to the next. Instead we need the method before to stop pointing to the previous node to the one we want to delete.
- Or, we can implement a so-called doubly-linked list, which is the choice of 10 out of 10 C++ programmers.

Moving backwards in linked lists

- The ability to move to the node before is very useful
- E.g. when we wanted to remove a node, it was not easily possible to do so after using the original `list_search()`
- The solution is to implement the list using both forward and backward pointers
- Thus the private member data for a `node` becomes:

```
private:
    value_type data;
    node* next;
    node* previous;
    node* current;
```

- Of course, that will require the creation of the methods `get_previous` and `set_previous`, and changes to the constructor of `node`.
- Also, the previous of `head` is NULL

Removing a node from a doubly-linked list

- Removing a `node` must be done carefully
- There are four situations to consider:
  - Removing the head `node`
  - Removing the `node` pointed to by the current `pointer`
  - Removing all nodes
  - Removing the tail `node`
- In all cases, be careful not to lose the rest of list by failing to store, or by overwriting a crucial pointer

Removing the head node

- The head `node` is easy to find because a pointer to it is stored in `head`
- The technique is:
  - Use a temporary variable to point to the about-to-be-removed `node`
  - Store a pointer to what was previously the second `node` as the new `head`
    - Making sure we cope correctly with the single node list situation
  - Use the pointer stored in the temporary variable to free the memory

```
void LinkedList::remove_from_head()
// Precondition: the list is not empty
// Postcondition: The first node is removed and returned to the heap
{
   node* temp;
   temp = head;
   head = head->get_next();
   if (head != NULL) {head->set_previous(NULL);}
   else {tail = NULL;} // list is empty, update tail
   delete temp; // free the node's space
}
```

Removing an internal node

- With this one we must be careful with the pointers

```
void LinkedList::remove_from_current()
{
// Precondition: current points to the node to be removed
// Postcondition: The node pointed to by current before is now gone; current points to
// the next element in the list
  node* temp;
  temp = current->get_previous();
  temp->set_next(current->get_next());
  temp = current->get_next();
  temp->set_previous(current->get_previous());
  delete current;
  current = temp;
}
```

Removing all nodes

- This is achieved by repeatedly removing the head node until there are no nodes left

```
void LinkedList::list_clear()
{
// Precondition: None
// Postcondition: the list is empty and both head and tail are NULL
   while (head != NULL)
      remove_from_head();
}
```

Removing the tail node

- Do this by yourself later, as an exercise.
- Have a look at Visualgo, and see visually how the various linked list operations work, step-by-step: https://visualgo.net/en/list

Copying a linked list

- The function `list_copy()` takes a reference to a linked list `source` and creates a copy of the list with new `head` and `tail` pointers

```
void LinkedList::list_copy(LinkedList& source) {
   list_clear(); // avoids memory leak
   if (source.head == NULL) return; // Deals with an empty source list
   // Copies the first node to the new list
   source.start(); // sends current to the head
   add_to_head(source.get_current_data());
   tail = head;
   current = tail;
   source.forward();
   // Now copies across the remaining data
   while (source.current() != NULL) {list_insert(source.get_current_data());}
}
```

- It is easy to use `list_copy`:

```
LinkedList destLL; // destination linked list
destLL.list_copy(sourceLL)
```

Copying a linked list

- Note the use of additional methods from `LinkedList` not yet described, e.g. `start()`, `get_current_data()`, `forward()`, `current()`.
- Those are methods to help control and use the linked list
  - `start()` makes `current` point to `head`
  - `get_current_data()` returns the item in the `node` pointed to by `current`
  - `forward()` moves `current` one node forward
  - `current()` returns the `current` pointer
- The implementation of a real linked list will require dozens of methods, some of them not mentioned here. You will notice the need for them when you code your first assignment and receive a list of requirements. Other examples are:
  - `void add_to_tail(const value_type&);`
  - `value_type remove_from_tail();`
  - `void end();`
  - `void backward();`

How to use linked lists

How to implement a bag structure with a linked list

Read chapter 16 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

Implementing a "Bag" with a linked list

- A bag is a container similar to a set, but accepts duplicates of items. The order of the items is not relevant.
- We will name this class LBag, as a bag implemented using a linked list.
- The `LBag` implementation needs a single piece of member data, namely an instance of `LinkedList` named, say `list`.

Typedef for LBag

- Previously we used `typedef <type> value_type;` to denote a generic type of object
- That was used in the `node` class.
- However we now need to ensure that `value_type` is the same for `LBag`, `LinkedList` and `node`. Otherwise how could we use the bag to store items?
- This is achieved as follows:
  - In the class `node`: `typedef <type> value_type;`
  - In the class `LinkedList`: `typedef node::value_type value_type;`
  - In the class `LBag`: `typedef LinkedList::value_type value_type;`

.h file for LBag

```
#ifndef ALEX_LBAG
#define ALEX_LBAG
#include <cstdlib>
#include "LinkedList.h"

class LBag
{
public:
   typedef LinkedList::value_type value_type;
   // Constructors
   LBag();
   LBag(LBag& source);
   // Destructor
   ~LBag();
   // Member functions
   int erase(const value_type& target);
   bool erase_one(const value_type& target);
   void insert(const value_type& entry);
   void operator +=(LBag& addend);
   void operator =(LBag& source);
   int size() const;
   int count(const value_type& target);
private:
   LinkedList list;
};
LBag operator +(LBag& b1, LBag& b2);
#endif
```

Rules for Dynamic Memory in a Class

- Note that the `LBag` class does not use dynamic memory directly, but you should always have a destructor.
- The easiest way is to not have any implementation in the destructor of `LBag`. In that case, the destructor of `LBag` will call the destructor of `LinkedList`, which then should call `list_clear()`.

```
LBag::~LBag() {} // empty destructor for LBag

LinkedList::~LinkedList() {list_clear();} // destructor for LinkedList
```

Constructors for LBag

- The default constructor is easy
- Uses the default constructor for `LinkedList`, i.e. simply set `head`, `tail` and `current` to `NULL`.
- The copy constructor is as follows

```
LBag::LBag(LBag& source){list = source.list;}
```

- Notice how simple the constructor is. A single line of code. It only requires the contents of the linked list from `source` to be copied onto the contents of `list`.
- All the "hard-work" is executed within the class `LinkedList`, which will have to have an overloaded copy operator (=) that uses `list_copy()`.
- Let's see how.

Copy operator (=) in LinkedList

```
void LinkedList::operator =(LinkedList& source)
{
   // Check for self-assignment.
   // 'this' is a pointer to the object for which the function
   // is activated (i.e. the LHS)
   if (this == &source) return;
   list_copy(source);
}
```

- Be careful, very careful when copying objects in C++. Remember there are "Shallow" and "Deep" types of copy.

Implementation of erase_one()

- For a bag, this function removes a single instance of the target from the bag. It calls `list_search()` from `LinkedList`, followed by `remove_from_current()`. Again, just a couple of lines of code.
- But what if the item to be removed is stored in the `head` node? There is a method `remove_from_head()` in `LinkedList`. What about the tail node? Do I need to then use `remove_from_tail()`?
- OK, enough! This is getting too confusing!

- When things get confusing it is time to rethink your code design. Why not merge `remove_from_head()`, `remove_from_current()` and `remove_from_tail()` into a single method, named say, `remove_node()`?

- What about `add_to_head()`, `add_current()` and `add_to_tail()`?

Implementation of erase_one()

- See, that will require changing a few methods that were already implemented, e.g. `list_clear()`. It was:

```
void LinkedList::list_clear()
{
   // Precondition: None
   // Postcondition: the list is empty and head and tail are both NULL
   while (head != NULL)
      remove_from_head();
}
```

- That will become:

```
void LinkedList::list_clear()
{
   // Precondition: None
   // Postcondition: the list is empty and head, tail and current are NULL
   current = head;
   while (current != NULL)
      remove_from_current();
}
```

Implementation of erase_one()

- Code changes all the time
- Sometimes you might think you made the correct design decisions, but then, if the use of your data structure gets too awkward and counter-intuitive, it is time to revisit some early design decisions
- Again, there are dozens of ways to implement a linked list, and all are correct
- As long as you don't violate the object-oriented programming principle of encapsulation, it should be fine
- A class should never manipulate another class directly, without the use of public methods from the other class
- It is just that some `LinkedList` class implementations will be easier to use than others

Implementation of erase_one()

```
bool LBag::erase_one(const value_type& target) {
   // Uses cstdlib, node.h
   boolean found = false;
   found = list.list_search(target);
   if (found) {
      list.remove_from_current();
      return true;
   }
   return false;
}
```

- A similar technique can be used to implement `erase`, which removes all instances of `target` from `LBag`.

Questions

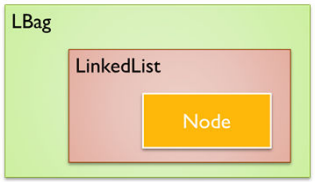
- Should we have a private member variable in `LinkedList` to store the size of the list?
- YES! Why? Efficiency!

- Changes lots of things.
- All methods that modify the number of elements in `LinkedList` now need to update that counter.

- In your assignments, you can have a counter in the `LinkedList`, in the class that uses it, or both.

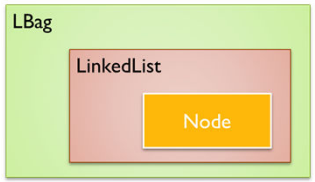
Wrapping around classes

```
class LinkedList
...
private:
      node* head;
      node* tail;
      node* current;
...
```



Wrapping around classes

```
class LBag
...
private:
      LinkedList list;
...
```

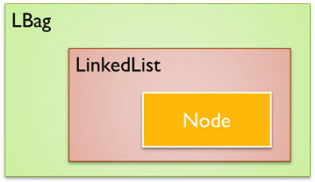


- Implementation

```
void LBag::insert(const value_type& entry) {
   list.start(); // sends current to the head
   list.add_node(entry);
}
```

- Note that `LBag` does not access the linked list directly. It is always through a `LinkedList` member method. Moreover, `LBag` should not even have an `#include "Node.h"` statement
- That is encapsulation and separation of concerns

Wrapping around classes



- Reuse! Reuse! Reuse!
- Note that all the heavy lifting happens in the inner classes. If the design is done correctly, the outer classes become easier and easier to implement, in a process that is nearly "error-free". Implementation requires just a few lines of code.
- The same `LinkedList` can now be used to implement queues, stacks and other "array-like" data structures

DEMO

See you next week!