Manipulating 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_ptr;
   node* tail_ptr;
   node* current;
...
};
```

Count of list elements

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

```
int LinkedList::list_length();
// Precondition: None
// Postcondition: A count of the nodes in
// the list is returned
```

- The function maintains a counter that is incremented as each `node` is traversed, and a pointer used to indicate the current `node` in the traversal.

Implementation of `list_length()`

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

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

Inserting at head of list

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

```
void LinkedList::list_head_insert(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_ptr = new node(entry, head_ptr);
// The following is required if a tail pointer is
// used. It deals with adding to an empty list
    if (tail == NULL) {tail = head_ptr;}
}
```

Inserting, but not at the head of the list

- Such as insertion required a pointer to the `node` just before the location of the new `node`
  - We store a pointer to this `node` in `current`
- The steps involved are
  - Create a new instance of `node`
  - Store the data item
  - Store `current->link()`
  - Point `current` to the new `node` instance
  - Update `current` so that next item is inserted after the new node.

```
void LinkedList::list_insert(const node::value_type& entry)
// Precondition: current points to the node just before
// the insertion position
// Postcondition: A new node is 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_link(current->link());
    current->set_link(add_ptr);
    if (current == tail) {tail = current->link();}
    current = current->link();
}
```

Comments

- In the function `list_insert()` 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
  - This will remove the node that `add_ptr` points to, *which is the node you just created!*
  - Only call `delete` if you want to reduce the number of nodes in the list
- Similarly, students sometimes use `current = new node();` when setting up a pointer for list traversal
  - This creates a new instance of `node`, unnecessary for traversal
  - Do not use `new` unless 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_ptr; current != NULL;
                 current = current->link())
    {
        if (target == current->data())
            return true;
    }
    return false;
}
```

Removing a node from a linked list

- 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
- So, in that case, we need to implement a doubly-linked list, which is the choice of 10 out of 10 C++ programmers.

Moving backwards in linked lists

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

```
private:
    Object data;
    node* next;
    node* previous;
```

  - Allowing forward and backward traversal
    - But that makes insertion and removal of `Node` instances that little bit more complex

Moving backwards in linked lists

- You will need to create additional support methods, such as `link_back()`, which returns the pointer to the previous node, or `previous`.
- The previous of the head node is `NULL`.
- In your assignments, you should implement doubly-linked lists.
- It is more difficult to implement than singly-linked lists, but way, way more flexible.

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 overwriting, a crucial pointer*

Removing the head node

- The head `Node` is easy to find because a pointer to it is stored in `head_ptr`
- 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_ptr`
    - Making sure we cope correctly with the single `Node` list situation
  - Use the pointer stored in the temporary variable to return the occupied space to the free heap

```
void LinkedList::list_head_remove()
// Precondition: the list is not empty
// Postcondition: The first Node is removed and
//                returned to the heap
{
   Node* temp_ptr;
   temp_ptr = head_ptr;
   head_ptr = head_ptr->link();
   if (head_ptr != NULL) {head_ptr->set_link_back(NULL);}
   else {tail_ptr = NULL;} // list is empty, update tail
   delete temp_ptr; // Free the Node's space
}
```

Removing an internal node

- With this one we must be careful with the pointers

```
void LinkedList::list_node_remove()
{
// Precondition: current points to
// the Node to be removed
// Postcondition: The Node pointed to by
// current before is gone; current points to
// the next element in the list
  Node* temp_ptr;
  temp_ptr = current->link_back();
  temp_ptr->set_link(current->link());
  temp_ptr = current->link();
  temp_ptr->set_previous(current->link_back());
  delete current;
  current = temp_ptr;
}
```

Removing all nodes in a linked list

- This is achieved by repeatedly removing the head node until there are none left

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

Removing the tail node

- Do it yourself, just to see if you got the idea.

Copying a linked list

- The function `list_copy()` takes a linked list pointed to by `source_ptr` and creates a copy of the list with head `head_ptr` and tail `tail_ptr`

```
void LinkedList::list_copy(LinkedList& source) {
   list_clear(); // avoids memory leak
   // Deal with an empty source list
   if (source.head_ptr == NULL) return;
   // Set up the head Node for the new list
   source.start(); // sends current to the head
   list_head_insert(source->get_current_data());
   tail_ptr = head_ptr;
   current = tail_ptr;
   source.advance();
   // Now copy across the remaining data
   while (source.current() != NULL) {
      list_insert(source.current()); // works
                      // with current == tail;
   }
}
```

Implementing a "Bag" with a linked list

- A bag is similar to a set, but accepts duplicates of items. The order of the items is not relevant.
- The `LBag` implementation *needs* a single piece of member data, namely an instance of `LinkedList` named, say `list`.

Type of data items

- 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 the bag, linked list and the nodes.
  - Otherwise how could we use the bag to store items?
- This is achieved as follows:

```
#include "node.h"
…
class Lbag
{
public:
   typedef LinkedList::value_type value_type;
// Programmer must make sure correct type
// is defined in class Node
   …
```

Prototype of `Lbag`

```
#ifndef ALEX_LBAG
#define ALEX_LBAG
#include <cstlib>
#include "LinkedList.h"
namespace Alex_SENGX120
{
   class Lbag
   {
   public:
      typedef LinkedList::value_type value_type;
      // Constructors
      Lbag();
      Lbag(const Lbag& source);
      // Destructor
      ~Lbag();
      // Mutating member functions
      int erase(const value_type& target);
      bool erase_one(const value_type& target);
      void insert(const value_type& entry);
      void operator +=(const Lbag& addend);
      void operator =(const Lbag& source);
      // Query member functions
      int size() const;
      int count(const value_type& target) const;
   private:
      LinkedList list;
   };
   Lbag operator +(const Lbag& b1, const Lbag& b2);
}
#endif
```

Rules for Dynamic Memory Usage in a Class

- Note that the new `Lbag` class does not use dynamic memory, 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()`.
- Best option is 2
- Why?

```
Lbag::~Lbag() {}

LinkedList::~LinkedList() {
    list_clear();
}
```

Constructors for Lbag

- The default constructor is easy
  - Uses the default constructor for `LinkedList`, i.e. simply set `head_ptr`, `tail_ptr` 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" will be executed within the class `LinkedList`, which will have to have an overloaded copy operator that uses list_copy.
- Let's see how.

Overloading the 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 `list_node_remove()`. 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 `list_head_remove()` in LinkedList.
- OK, enough! This is getting too confusing!
- When things get confusing it is time to rethink your design. Why not merge `list_head_remove()` and `list_node_remove()` in the same method?
- And while you are at that, merge `list_tail_remove()` as well into `list_node_remove()`.
- What about `list_insert()` and `list_head_insert`?

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_ptr and tail_prt are both NULL
   while (head_ptr != NULL)
      list_head_remove();
}
```

- That will become:

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

Implementation of `erase_one()`

- Code changes all the time.
- Sometimes you might think you made the correct design decisions, but then, when 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 principles of encapsulation, it should be fine.
  - A class should never manipulate another class directly, without the use of public methods of the other class.
- It is just that some Linked List 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_search(target);
   if (found) {
      list_node_remove();
      return true;
   }
   return false;
}
```

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

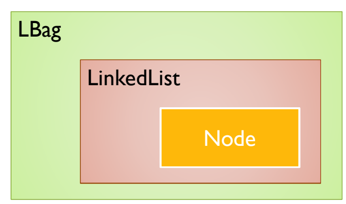
Questions

- 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` needs to update the counter.

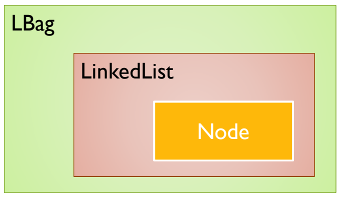
- In your assignments, you can have a counter either in linked list or in the class that uses it.

Wrapping around classes




```
class LinkedList
...
private:
      node* head_ptr;
      node* tail_ptr;
      node* current;
...
```

Wrapping around classes




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

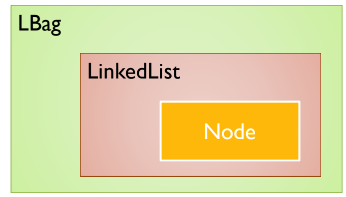
Implementation:

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

Note that LBag does not access the linked list directly. It is always through a Linked List member method. In addition LBag should not even have an `#include "Node.h"`

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 this is done correctly, the outer classes become easy to implement, and nearly "error-free". Implementation requires just a few lines of code.
- The same Linked List can now be used to implement Queues, Stacks and any other "array-like" data structure