Linked Lists

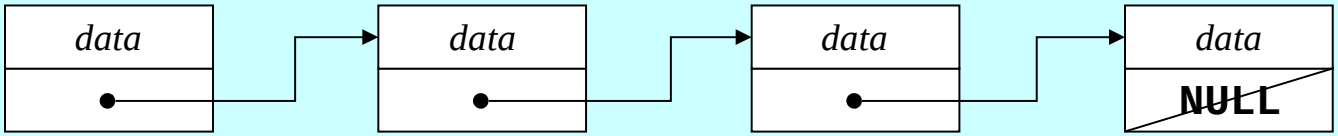
Eric Roberts
CS 106B
February 6, 2015

In Our Last Episode . . .

- On Wednesday, I talked about command-line editors and introduced the **`buffer.h`** interface, which exports a class called **`EditorBuffer`**.
- Our goal for the week is to implement the **`EditorBuffer`** class in three different ways and to compare the algorithmic efficiency of the various options. Those representations are:
  1. A simple *array model*, which I introduced last time.
  2. A *two-stack model* that uses a pair of character stacks.
  3. A *linked-list model* that uses pointers to indicate the order.
- For each model, we'll calculate the complexity of each of the six fundamental methods in the **`EditorBuffer`** class. Some operations will be more efficient with one model, others will be more efficient with a different one.

Linking Objects Together

- Pointers are important in programming because they make it possible to represent the relationship among objects by linking them together in various ways.
- The simplest example of a linked structure (which appears first in Chapter 12 and is used throughout the later chapters) is called a ***linked list***, in which each object in a sequence contains a reference to the one that follows it:

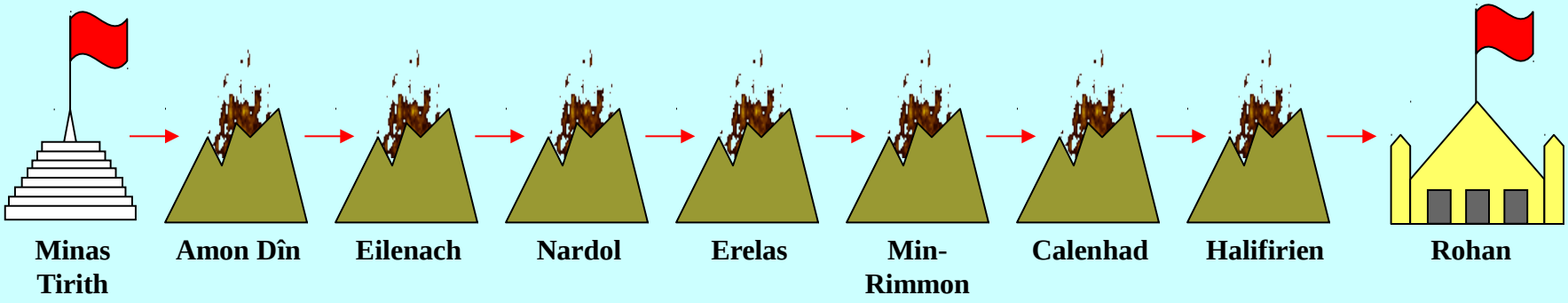


- C++ marks the end of linked list using the constant **`NULL`**, which signifies a pointer that does not have an actual target.
- In diagrams, the **`NULL`** value marking the end of a list is often indicated by drawing a diagonal line across the box.

The Beacons of Gondor

> *For answer Gandalf cried aloud to his horse. "On, Shadowfax! We must hasten. Time is short. See! The beacons of Gondor are alight, calling for aid. War is kindled. See, there is the fire on Amon Dîn, and flame on Eilenach; and there they go speeding west: Nardol, Erelas, Min-Rimmon, Calenhad, and the Halifirien on the borders of Rohan."*
>
> —J. R. R. Tolkien, *The Return of the King*, 1955

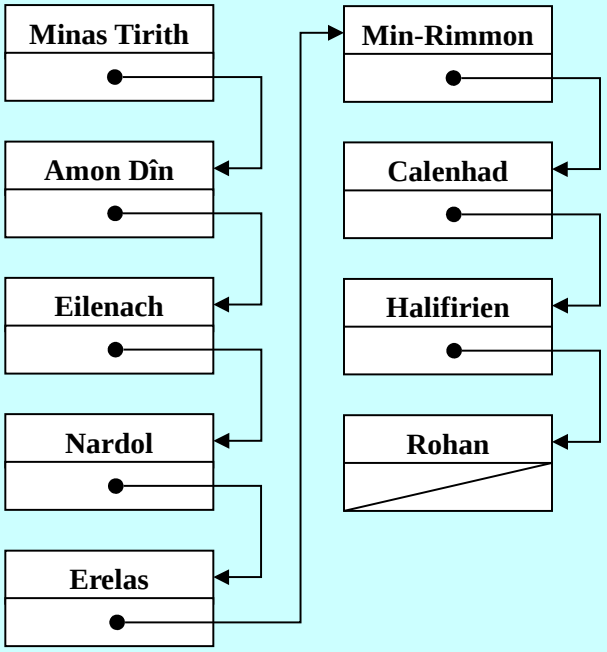
In a scene that was brilliantly captured in Peter Jackson's film adaptation of *The Return of the King*, Rohan is alerted to the danger to Gondor by a succession of signal fires moving from mountain top to mountain top. This scene is a perfect illustration of the idea of message passing in a linked list.



Message Passing in Linked Structures

To represent this message-passing image, you might use a definition such as the one shown on the right.

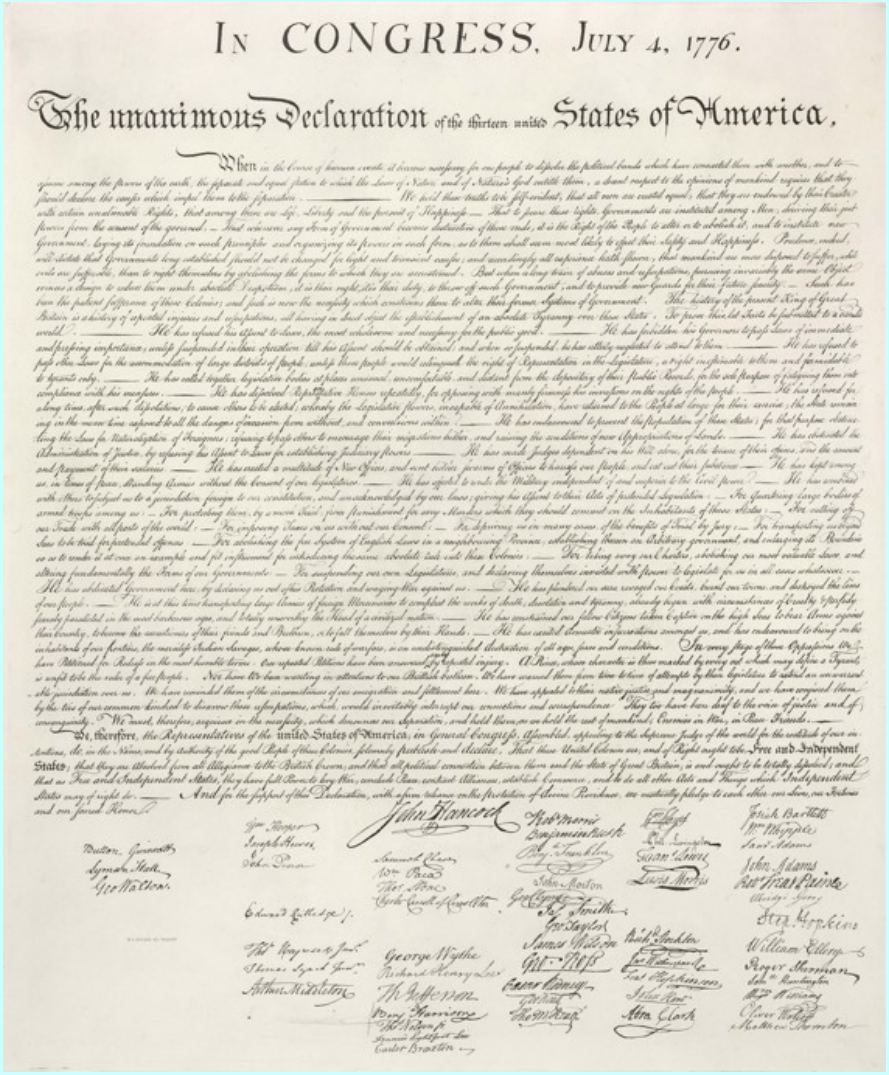
You can then initialize a chain of **`Tower`** structures, like this:



Calling **`signal`** on the first tower sends a message down the chain.

```
struct Tower {
   string name;  /* The name of this tower    */
   Tower *link;  /* Pointer to the next tower */
};

/*
 * Function: createTower(name, link);
 * ----------------------------------
 * Creates a new Tower with the specified values.
 */

Tower *createTower(string name, Tower *link) {
   Tower *tp = new Tower;
   tp->name = name;
   tp->link = link;
   return tp;
}

/*
 * Function: signal(start);
 * ------------------------
 * Generates a signal beginning at start.
 */

void signal(Tower *start) {
   if (start != NULL) {
      cout << "Lighting " << start->name << endl;
      signal(start->link);
   }
}
```

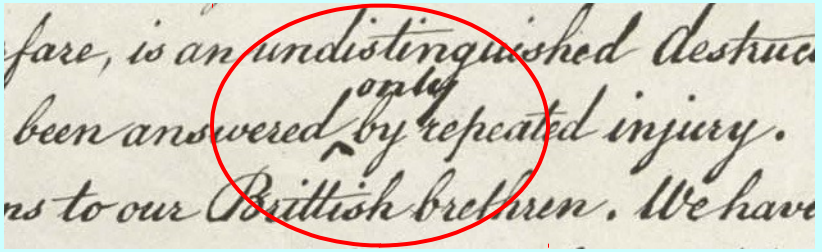
Adding Links to Correct Errors


IN CONGRESS, JULY 4, 1776.

The unanimous Declaration of the thirteen united States of America,

Suppose you're Thomas Jefferson and that you're writing timeless prose to mark our nation's founding. Of the British outrages, you've written that "our repeated petitions have been answered by repeated injury."

Now someone wants you to add the word "only" after "answered" in the finished text. What do you do?

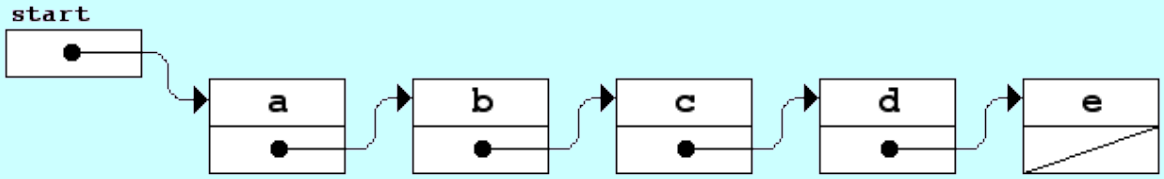


List-Based Buffers

- The list-based model of the **`EditorBuffer`** class uses pointers to indicate the order of characters in the buffer. For example, the buffer containing

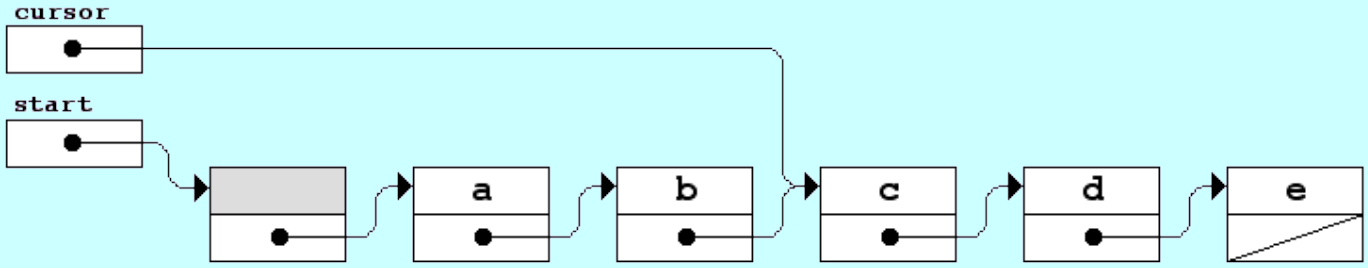
  a b c d e

  is modeled conceptually like this:



Representing the Cursor

- The diagram on the preceding slide did not indicate the cursor position, mostly because doing so is a bit tricky.
- If a list contains five cells, as in this example, there are ***six*** positions for the cursor. Thus, it is impossible to represent all of the possible positions by pointing to some cell.
- One standard strategy for solving this problem is to allocate an extra cell (usually called a ***dummy cell***) at the beginning of the list, and then represent the position of the cursor by pointing to the cell *before* the insertion point. Thus, if the cursor is between the **c** and the **d**, the five-character buffer would look like this:



Methods in the **`EditorBuffer`** Class

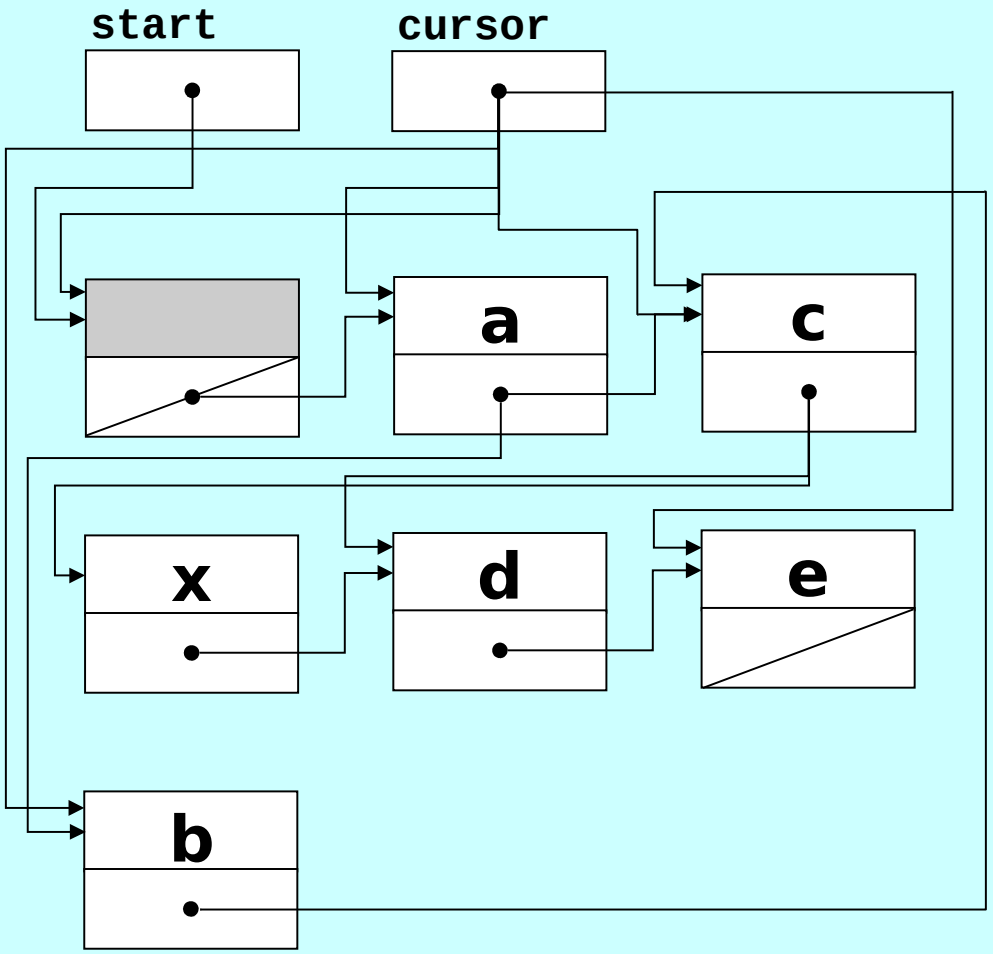
`buffer.moveCursorForward()` Moves the cursor forward one character (does nothing if it's at the end).
`buffer.moveCursorBackward()` Moves the cursor backward one character (does nothing if it's at the beginning).
`buffer.moveCursorToStart()` Moves the cursor to the beginning of the buffer.
`buffer.moveCursorToEnd()` Moves the cursor to the end of the buffer.
`buffer.insertCharacter(ch)` Inserts the character **`ch`** at the cursor position and advances the cursor past it.
`buffer.deleteCharacter()` Deletes the character after the cursor, if any.
`buffer.getText()` Returns the contents of the buffer as a string.
`buffer.getCursor()` Returns the position of the cursor.

List-Based Private Data for Buffer

```
/*
 * Implementation notes
 * --------------------
 * In the linked-list implementation of the buffer, the characters in the
 * buffer are stored in a list of Cell structures, each of which contains
 * a character and a pointer to the next cell in the chain.  To simplify
 * the code used to maintain the cursor, this implementation adds an extra
 * "dummy" cell at the beginning of the list.  The following diagram shows
 * a buffer containing "ABC" with the cursor at the beginning:
 *
 *        +-----+     +-----+     +-----+     +-----+     +-----+
 *  start |  o--+--==>|     |  -->|  A  |  -->|  B  |  -->|  C  |
 *        +-----+  /  +-----+  /  +-----+  /  +-----+  /  +-----+
 * cursor |  o--+--   |  o--+--   |  o--+--   |  o--+--   |  /  |
 *        +-----+     +-----+     +-----+     +-----+     +-----+
 */

   struct Cell {
      char ch;
      Cell *link;
   };

/* Data fields required for the linked-list representation */

   Cell *start;          /* Pointer to the dummy cell      */
   Cell *cursor;         /* Pointer to cell before cursor  */
```

List Editor Simulation

```
* Iacxde
 a c x d e
         ^
* J
 a c x d e
^
* F
 a c x d e
  ^
* Ib
 a b c x d e
    ^
* F
 a b c x d e
      ^
* D
 a b c d e
      ^
*
```



List-Based Buffer Implementation

```
/*
 * File: buffer.cpp (list version)
 * -------------------------------
 * This file implements the EditorBuffer class using a linked
 * list to represent the buffer.
 */

#include <iostream>
#include "buffer.h"
using namespace std;

/*
 * Implementation notes: EditorBuffer constructor
 * ----------------------------------------------
 * This function initializes an empty editor buffer represented as a
 * linked list.  In this representation, the empty buffer contains a
 * "dummy" cell whose ch field is never used.  The constructor must
 * allocate this dummy cell and set the internal pointers correctly.
 */

EditorBuffer::EditorBuffer() {
   start = cursor = new Cell;
   start->link = NULL;
}
```

List-Based Buffer Implementation

```
/*
 * Implementation notes: EditorBuffer destructor
 * ---------------------------------------------
 * The destructor must delete every cell in the buffer.  Note that the loop
 * structure is not exactly the standard for loop pattern for processing
 * every cell within a linked list.  The complication that forces this
 * change is that the body of the loop can't free the current cell and
 * later have the for loop use the link field of that cell to move to
 * the next one.  To avoid this problem, this implementation copies the
 * link pointer before calling delete.
 */

EditorBuffer::~EditorBuffer() {
   Cell *cp = start;
   while (cp != NULL) {
      Cell *next = cp->link;
      delete cp;
      cp = next;
   }
}
```

List-Based Buffer Implementation

```
void EditorBuffer::moveCursorForward() {
   if (cursor->link != NULL) {
      cursor = cursor->link;
   }
}

void EditorBuffer::moveCursorBackward() {
   Cell *cp = start;
   if (cursor != start) {
      while (cp->link != cursor) {
         cp = cp->link;
      }
      cursor = cp;
   }
}

void EditorBuffer::moveCursorToStart() {
   cursor = start;
}

void EditorBuffer::moveCursorToEnd() {
   while (cursor->link != NULL) {
      moveCursorForward();
   }
}
```

List-Based Buffer Implementation

```
/*
 * Implementation notes: insertCharacter, deleteCharacter
 * ------------------------------------------------------
 * The primary advantage of the linked list representation for
 * the buffer is that the insert and delete operations can be
 * performed in constant time by updating pointers instead of
 * moving data.
 */

void EditorBuffer::insertCharacter(char ch) {
   Cell *cp = new Cell;
   cp->ch = ch;
   cp->link = cursor->link;
   cursor->link = cp;
   cursor = cp;
}

void EditorBuffer::deleteCharacter() {
   if (cursor->link != NULL) {
      Cell *oldcell = cursor->link;
      cursor->link = oldcell->link;
      delete oldcell;
   }
}
```

Complexity of the Editor Operations

		array	*stack*	*list*
F	**moveCursorForward()**	$O(1)$	$O(1)$	$O(1)$
B	**moveCursorBackward()**	$O(1)$	$O(1)$	$O(N)$
J	**moveCursorToStart()**	$O(1)$	$O(N)$	$O(1)$
Z	**moveCursorToEnd()**	$O(1)$	$O(N)$	$O(N)$
I	**insertCharacter(ch)**	$O(N)$	$O(1)$	$O(1)$
D	**deleteCharacter()**	$O(N)$	$O(1)$	$O(1)$

- Is it possible to reimplement the editor buffer so that all six of these operations run in constant time?
- The answer is yes, and you will have the opportunity to write the necessary code for next week's section.

The End