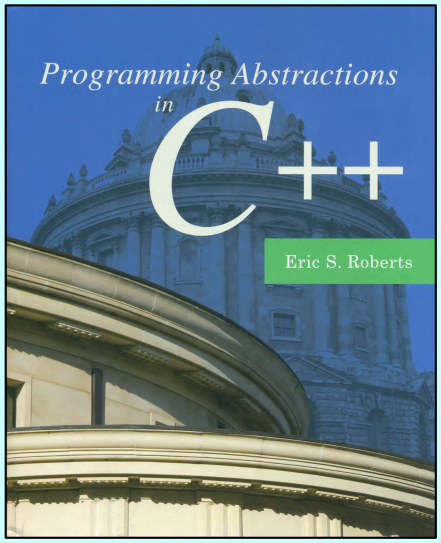
CHAPTER 13

Efficiency and Representation

Time granted does not necessarily coincide with time that can be most fully used.

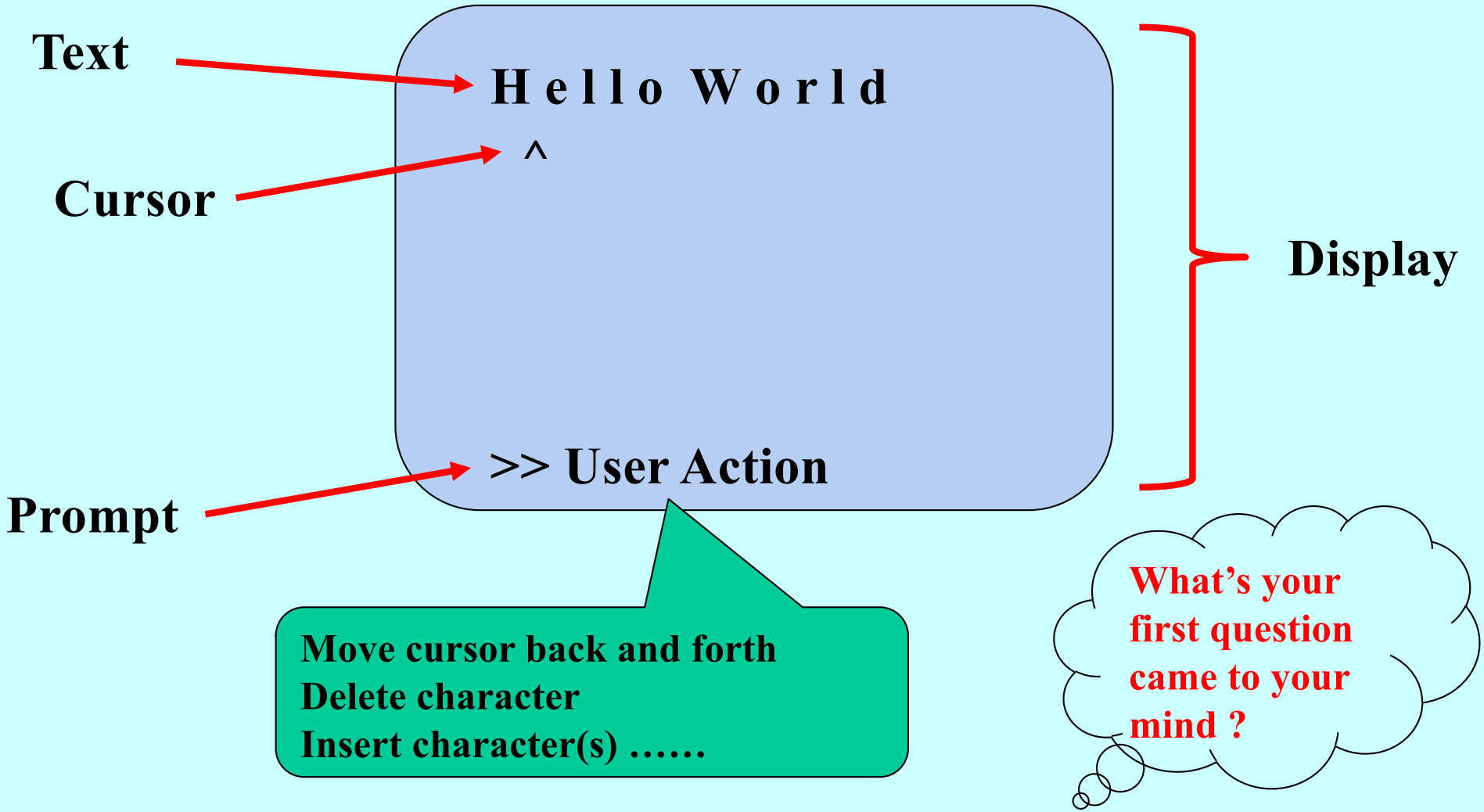
—Tillie Olsen, *Silences,* 1965



A Simple Text Editor

- Most editors today follow the WYSIWYG principle, which is an acronym for "what you see is what you get" that keeps the screen updated so that it shows the current document.
- Before WYSIWYG editors existed, most editors used a command-line model. To edit a document, you enter commands that consist of a letter, possibly along with some additional data.
- Rather than showing the contents of the editor all the time, command-line editors showed the contents only when the user asked for them.

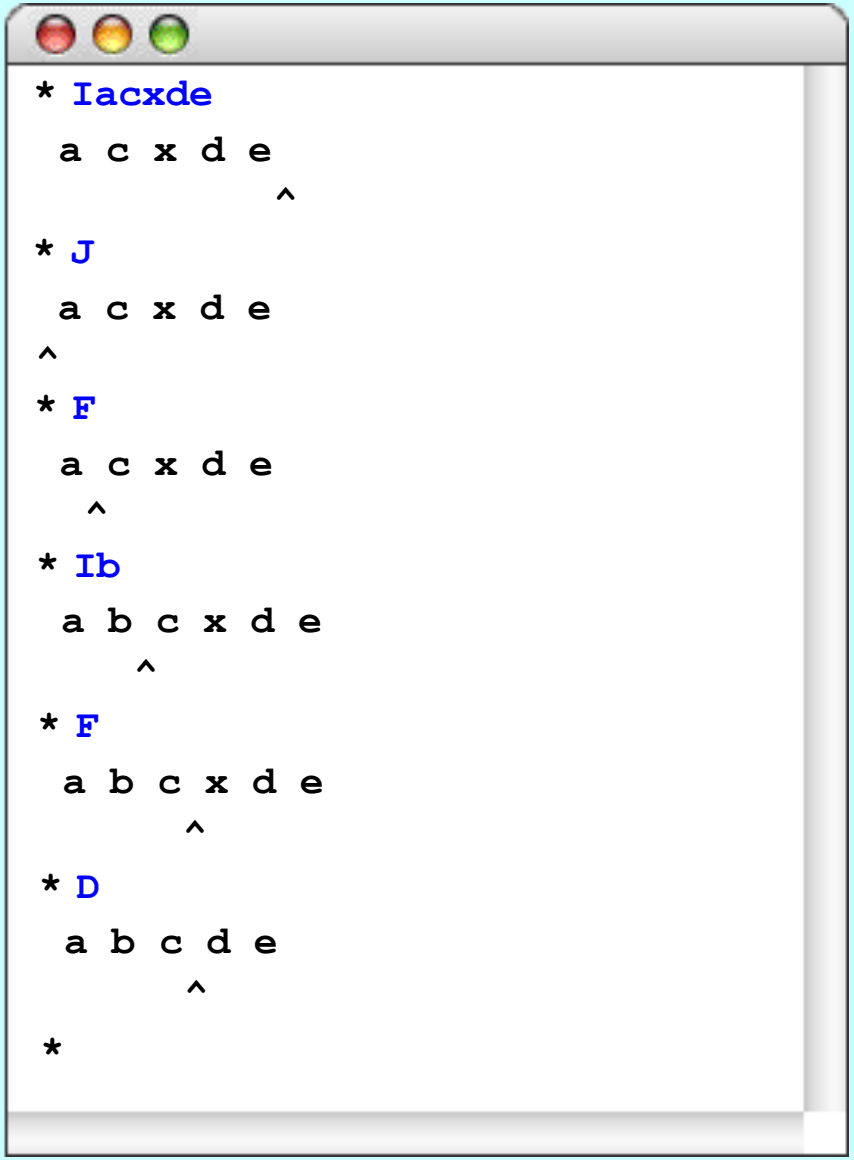
Text Editor – User Interface



Requirements - The Editor Commands

I*text*	Inserts the *text* following the **I** into the buffer.
J	Jumps the current point (the ***cursor***) to the beginning.
E	Moves the cursor to the end of the buffer.
F	Moves the cursor forward one character.
B	Moves the cursor backward one character.
D	Deletes the character after the cursor.
H	Prints a help message listing the commands.
Q	Quits from the editor.

Simple Editor - Illustration



Simple Editor - Spec

Moves the cursor forward one character (does nothing if it's at the end).
Moves the cursor backward one character (does nothing if it's at the beginning).
Moves the cursor to the beginning of the buffer.
Moves the cursor to the end of the buffer.
Inserts the character **`ch`** at the cursor position and advances the cursor past it.
Deletes the character after the cursor, if any.
Refreshes screen with the update
Prompts user for the command …….. etc

Text Editor - Components

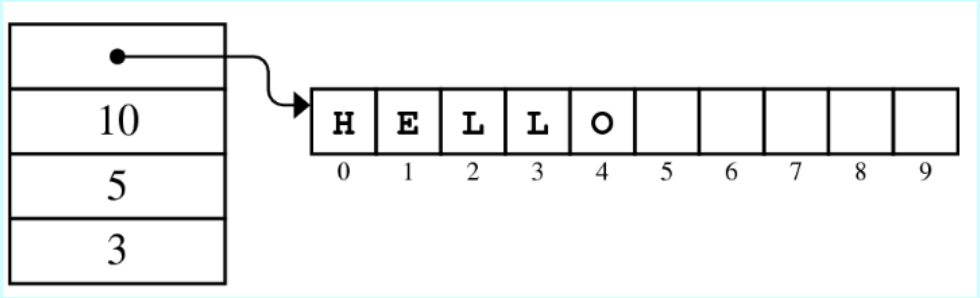
Data Structure

Display

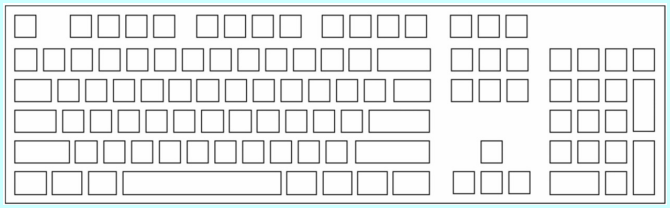
H E L L O

^

>> User Action



Processing Logic



Keyboard

How do these components relate to each other ??

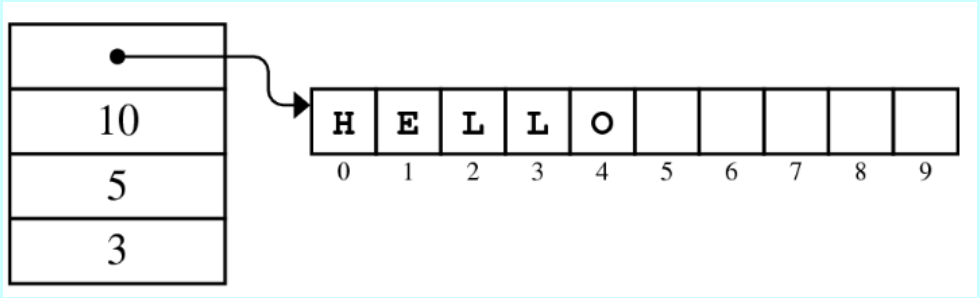
Text Editor - Design

Data Structure

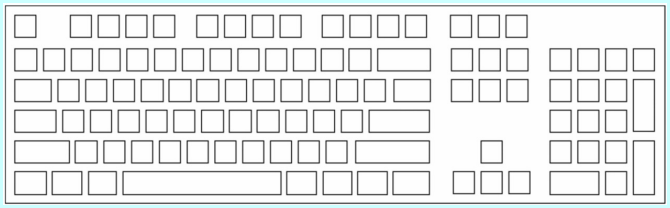
Display

H E L L O
^

>> User Action



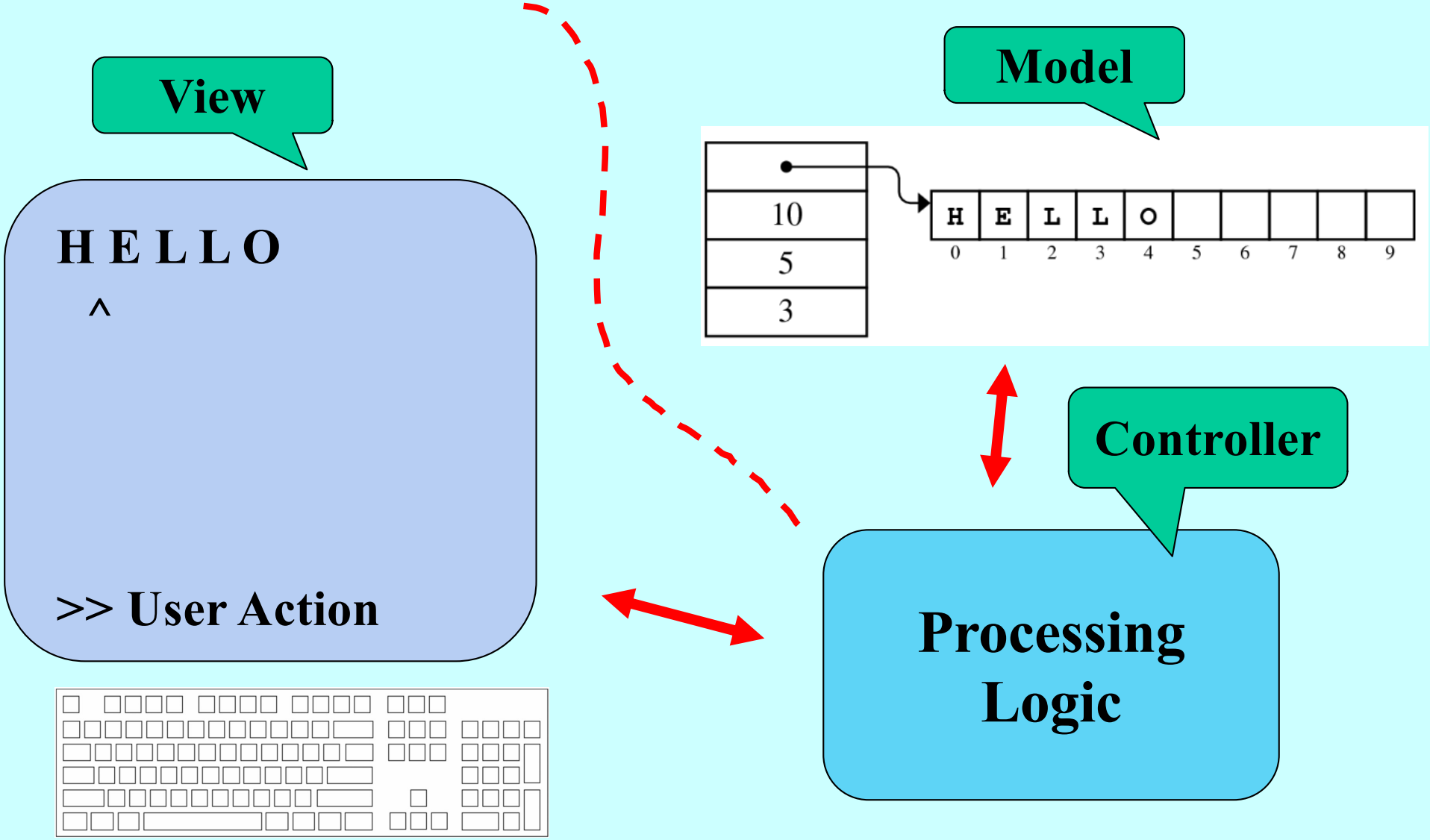
Processing Logic



Keyboard

What's bad about this design ?

Architecture Design - Model View Controller (MVC)



Text Editor – MVC Design

View

Controller

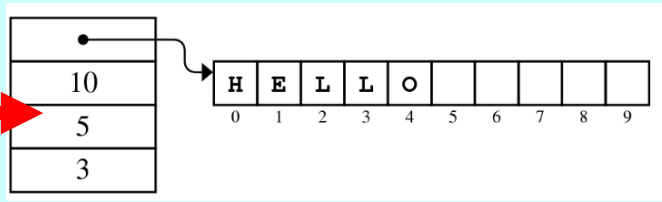
Model

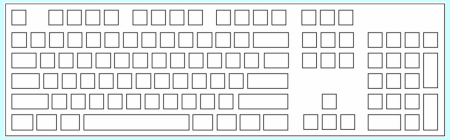
H E L L O

^

>> User Action

Coodinator





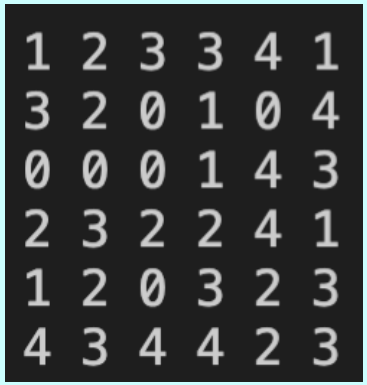
refreshDisplay
.....

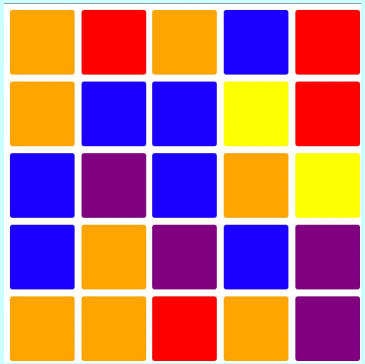
Prompt User
Processing the command
Update the Model
Coordindate Model and UI

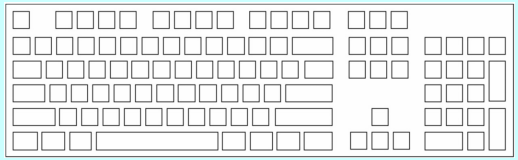
moveCursorForward
moveCursorBackward
moveCursorToStart
moveCursorToEnd
insertCharacter
deleteCharacter
.....

CSC1002

View

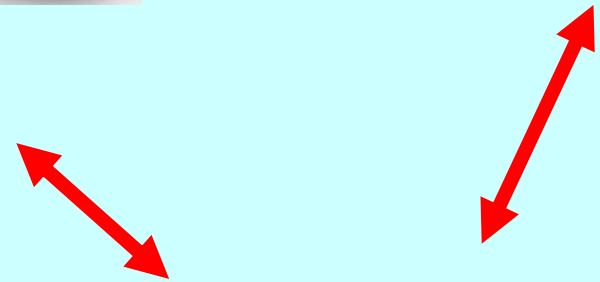






Model

```
g_dim = 5
g_tile_sz = 60
g_game = []
g_colors = ( "red", "yellow", "purple", "orange", "blue")
g_numbers = ( 0, 1, 2, 3 )

g_screen = None
g_squares = []
g_choices = []
g_color_picked = None
g_square_picked = None
```



Controller

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.

Simple Editor – Processing Command

Can you label model, view and controller ?ç

```
/*
 * Function: executeCommand
 * Usage: executeCommand(buffer, line);
 * ------------------------------------
 * Executes the co                    line on the editor buffer.
 */

void executeCommand(EditorBuffer & buffer, string line) {
   switch (toupper(line[0])) {
    case 'I': foreach (char ch in line) {
                 buffer.insertCharacter(ch);
              }
              displayBuffer(buffer);
              break;
    case 'D': buffer.deleteCharacter(); displayBuffer(buffer); break;
    case 'F': buffer.moveCursorForward(); displayBuffer(buffer); break;
    case 'B': buffer.moveCursorBackward(); displayBuffer(buffer); break;
    case 'J': buffer.moveCursorToStart(); displayBuffer(buffer); break;
    case 'E': buffer.moveCursorToEnd(); displayBuffer(buffer); break;
    case 'H': printHelpText(); break;
    case 'Q': exit(0);
    default:  cout << "Illegal command" << endl; break;
   }
}
```

Controller

line.substr(1)

Model

View

Simple Editor - Display

What does this imply ??

What does this function hide ??

```
/*
 * Function: displayBuffer
 * Usage: displayBuffer(buffer);
 * ----------------------------
 * Displays the state of the buffer including the position of the cursor.
 */

void displayBuffer(EditorBuffer & buffer) {
   string str = buffer.getText();
   for (int i = 0; i < str.length(); i++) {
      cout << " " << str[i];
   }
   cout << endl;
   cout << string(2 * buffer.getCursor(), ' ') << "^" << endl;
}
```

IO Stream !!!

Simple Editor - Display

```
/*
 * Function: displayBuffer
 * Usage: displayBuffer(buffer);
 * ----------------------------
 * Displays the state of the buffer including the position of the cursor.
 */

void displayBuffer ( const string & str, int cursor ) {
   string str = buffer.getText();
   for (int i = 0; i < str.length(); i++) {
      cout << " " << str[i];
   }
   cout << endl;
   cout << string(2 * cursor , ' ') << "^" << endl;
}
```

Where Do We Go From Here?

- Our goal from this point is to implement the **`EditorBuffer`** class in three different ways (i.e., different underlying data structures) and to compare the algorithmic efficiency of the various options. These representations are:

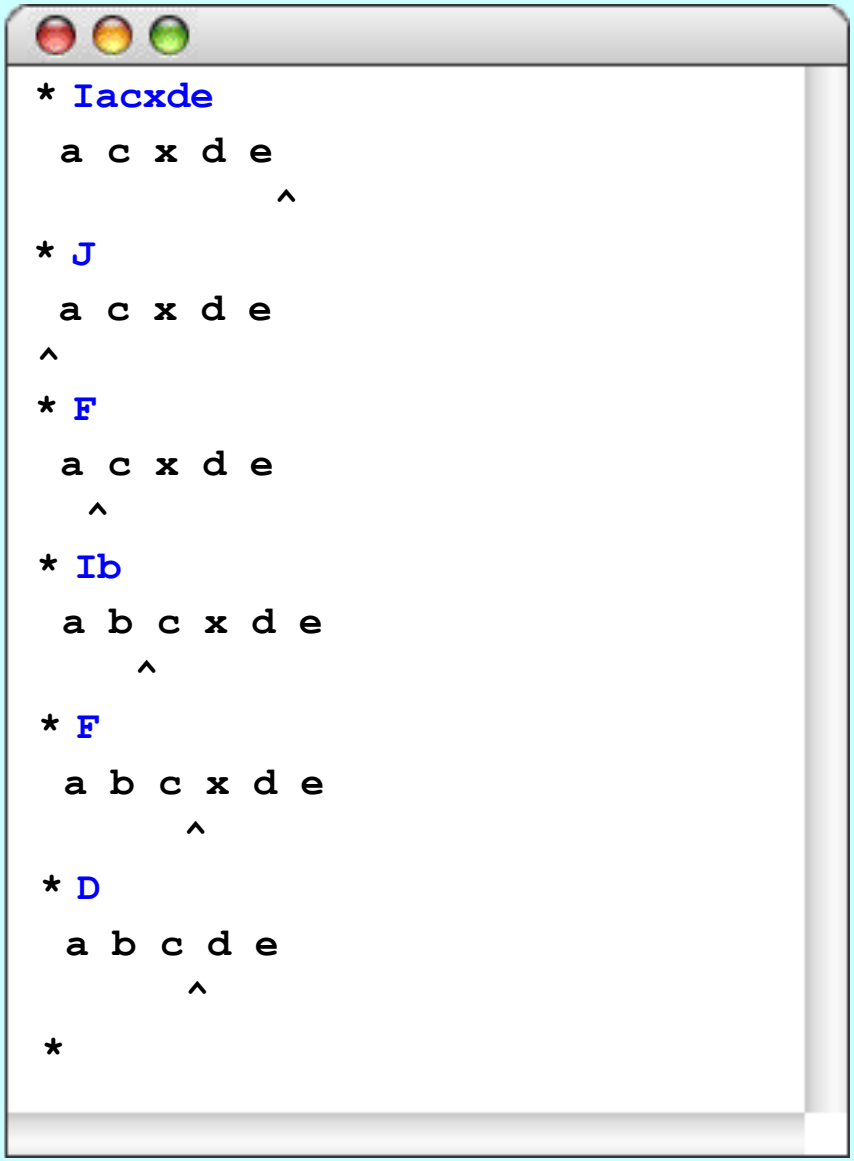
  1. A simple ***array model*** using dynamic allocation.
  2. A ***linked-list model*** that uses pointers to indicate the order.
  3. A ***two-stack model*** that uses a pair of character stacks.

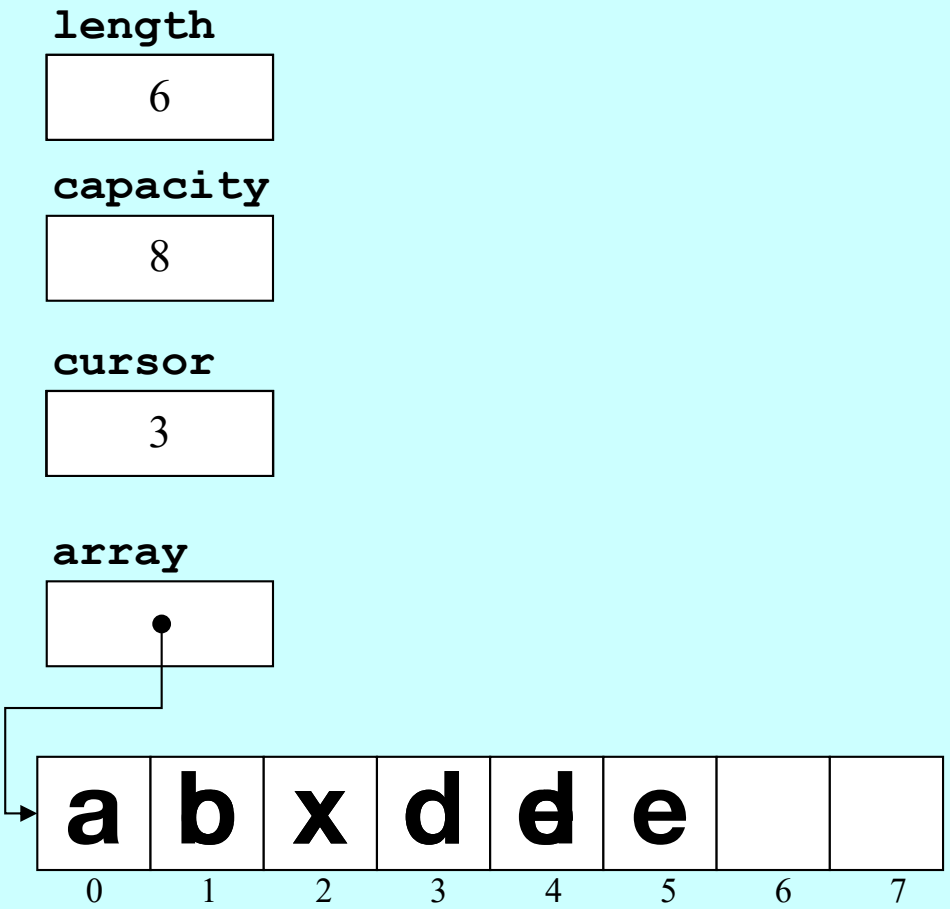
- For each model, we'll calculate the complexity of each of the six fundamental methods in the **`EditorBuffer`** class. Some operations will be more efficient with one model, others will be more efficient with a different underlying representation.

The Array Model

- Conceptually, the simplest strategy for representing the editor buffer is to use an array for the individual characters.
- To ensure that the buffer can contain an arbitrary amount of text, it is important to allocate the array storage dynamically and to expand the array whenever the buffer runs out of space.
- The array used to hold the characters will contain elements that are allocated but not yet in use, which makes it necessary to distinguish the ***allocated size*** (`capacity`) of the array from its ***effective size*** (`length`).
- In addition to the size and capacity information, the data structure for the editor buffer must contain an additional integer variable that indicates the current position of the cursor. This variable can take on values ranging from 0 up to and including the length of the buffer.

Array Editor Simulation





Private Data for Array-Based Buffer

```
private:

/*
 * Implementation notes: Buffer data structure
 * -------------------------------------------
 * In the array-based implementation of the buffer, the characters in the
 * buffer are stored in a dynamic array.  In addition to the array, the
 * structure keeps track of the capacity of the buffer, the length of the
 * buffer, and the cursor position.  The cursor position is the index of
 * the character that follows where the cursor would appear on the screen.
 */

/* Constants */

   static const int INITIAL_CAPACITY = 10;

/* Instance variables */

   char *array;           /* Dynamic array of characters     */
   int capacity;          /* Allocated size of that array    */
   int length;            /* Number of character in buffer   */
   int cursor;            /* Index of character after cursor */

/* Private method prototype */

   void expandCapacity();
```

Array-Based Buffer Implementation

```
/*
 * File: buffer.cpp (array version)
 * --------------------------------
 * This file implements the EditorBuffer class using an array representation.
 */

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

/*
 * Implementation notes: Constructor and destructor
 * ------------------------------------------------
 * The constructor initializes the private fields.  The destructor
 * frees the heap-allocated memory, which is the dynamic array.
 */

EditorBuffer::EditorBuffer() {
   capacity = INITIAL_CAPACITY;
   array = new char[capacity];
   length = 0;
   cursor = 0;
}

EditorBuffer::~EditorBuffer() {
   delete[] array;
}
```

Array-Based Buffer Implementation

```
/*
 * Implementation notes: moveCursor methods
 * ----------------------------------------
 * The four moveCursor methods simply adjust the value of the
 * cursor instance variable.
 */

void EditorBuffer::moveCursorForward() {
   if (cursor < length) cursor++;
}

void EditorBuffer::moveCursorBackward() {
   if (cursor > 0) cursor--;
}

void EditorBuffer::moveCursorToStart() {
   cursor = 0;
}

void EditorBuffer::moveCursorToEnd() {
   cursor = length;
}
```

Array-Based Buffer Implementation

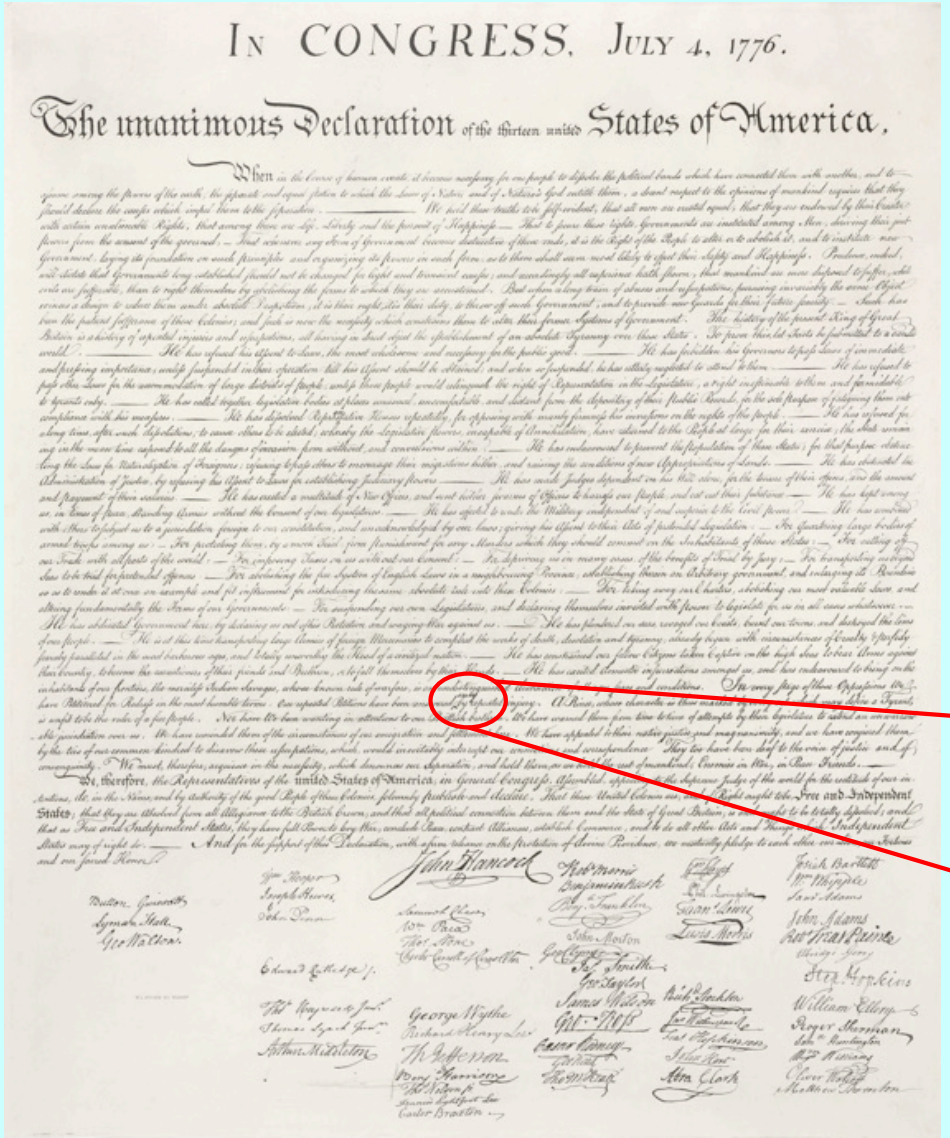
```
/*
 * Implementation notes: insertCharacter and deleteCharacter
 * ---------------------------------------------------------
 * Each of the functions that inserts or deletes characters
 * must shift all subsequent characters in the array, either
 * to make room for new insertions or to close up space left
 * by deletions.
 */

void EditorBuffer::insertCharacter(char ch) {
   if (length == capacity) expandCapacity();
   for (int i = length; i > cursor; i--) {
      array[i] = array[i - 1];
   }
   array[cursor] = ch;
   length++;
   cursor++;
}

void EditorBuffer::deleteCharacter() {
   if (cursor < length) {
      for (int i = cursor+1; i < length; i++) {
         array[i - 1] = array[i];
      }
      length--;
   }
}
```

Array-Based Buffer Implementation

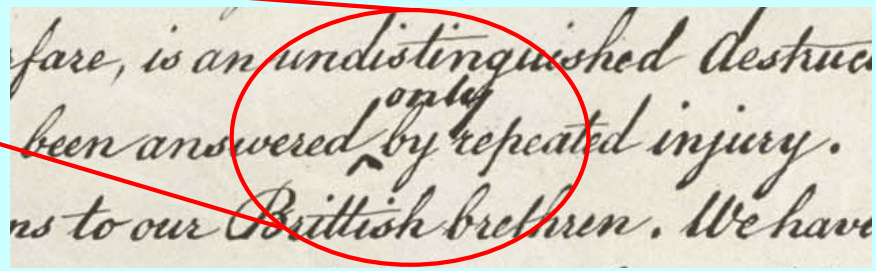
```
/* Simple getter methods: getText, getCursor */

string EditorBuffer::getText() const {
   return string(array, length);
}

int EditorBuffer::getCursor() const {
   return cursor;
}

/*
 * Implementation notes: expandCapacity
 * ------------------------------------
 * This private method doubles the size of the array whenever the old one
 * runs out of space.  To do so, expandCapacity allocates a new array,
 * copies the old characters to the new array, and then frees the old array.
 */

void EditorBuffer::expandCapacity() {
   char *oldArray = array;
   capacity *= 2;
   array = new char[capacity];
   for (int i = 0; i < length; i++) {
      array[i] = oldArray[i];
   }
   delete[] oldArray;
}
```

```
string(const char* s, size_t n);
```

Insertion in Fixed Text


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 the nation's founding (Declaration of Independence). Of the British outrages, you've written that "our repeated petitions have been answered by repeated injury."

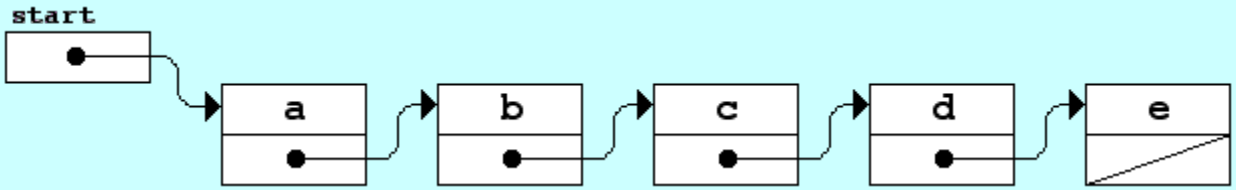
Now someone wants you to add the word "only" after "answered" in the finished text. What do you do?

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

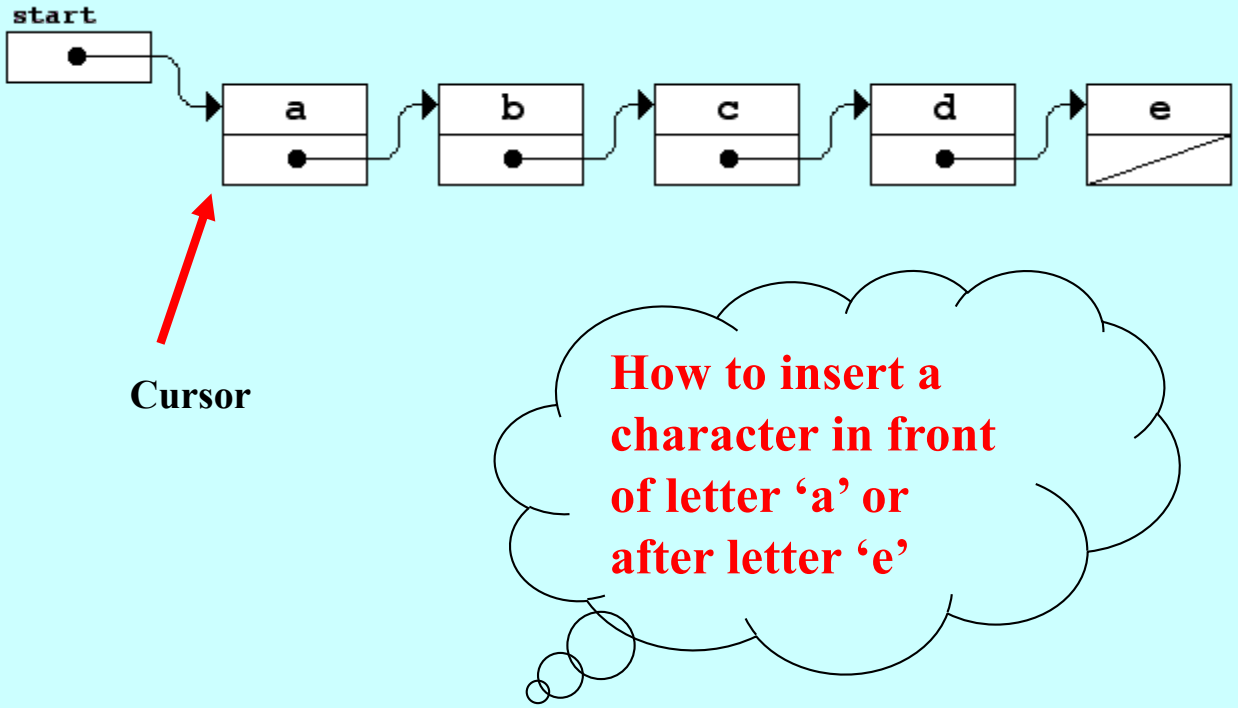
  `a b c d e`

  is modeled conceptually like this:



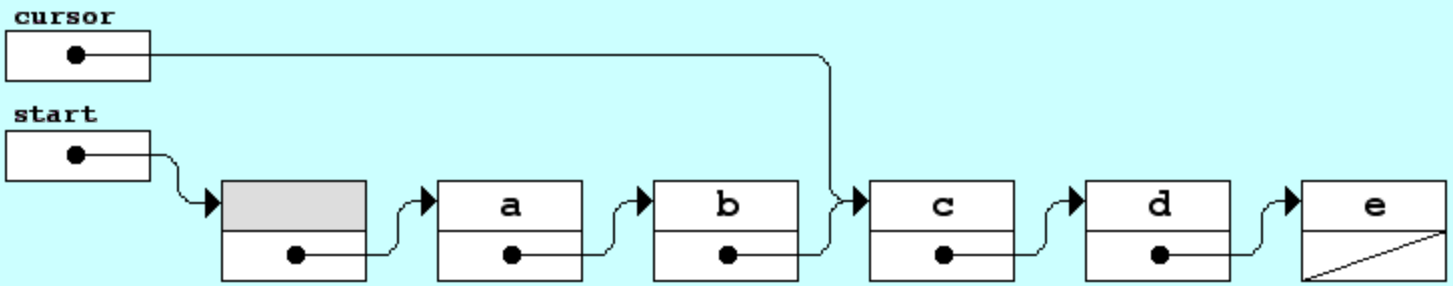
- The diagonal line through the last link pointer is used in list diagrams to indicate the **`NULL`** value that marks the end of the list.

List-Based Buffers – Cursor & Insert



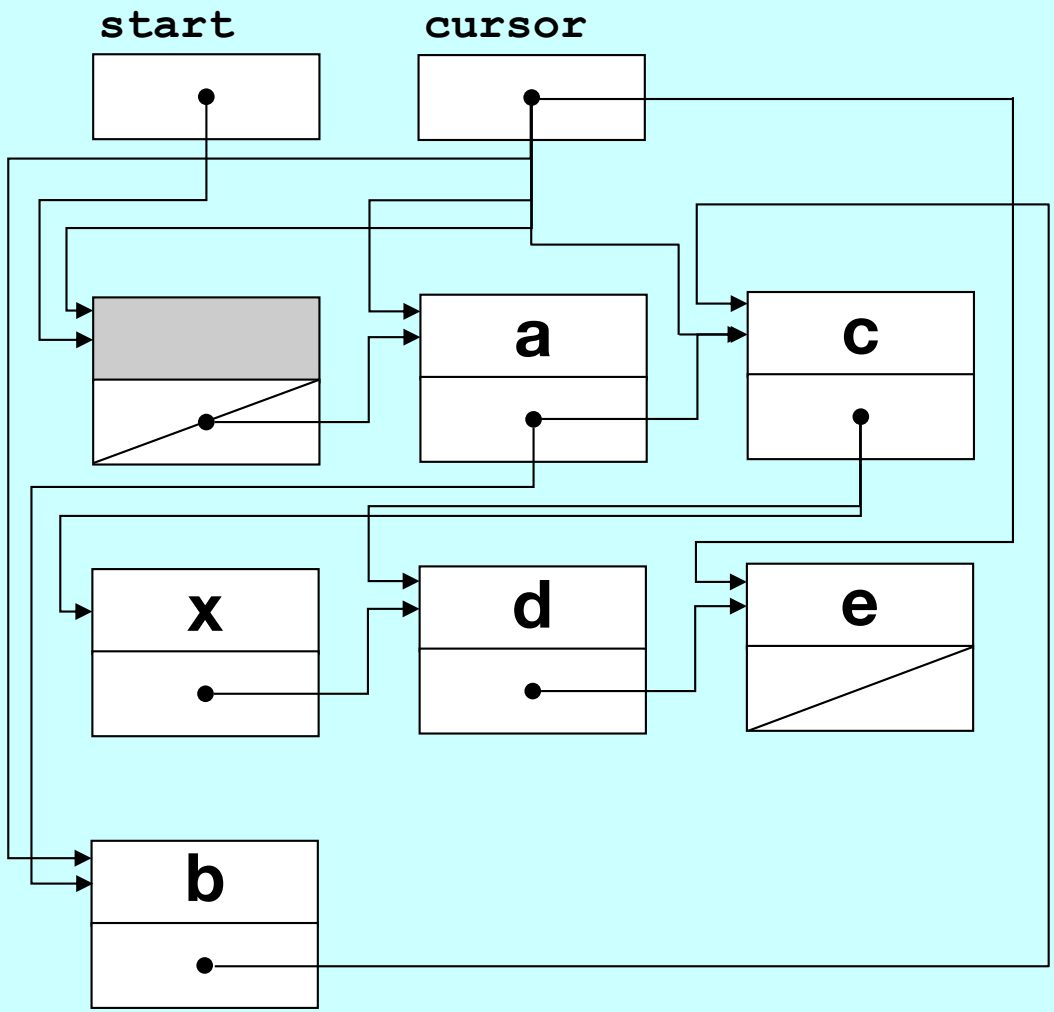
Representing the Cursor

- 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 cell **`c`** and cell **`d`**, the pointer **`cursor`** is pointing to **`c`**. The five-character buffer would look like this:≈



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
      ^
*
```



Private Data for List-Based Buffer

```
private:
/*
 * 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-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

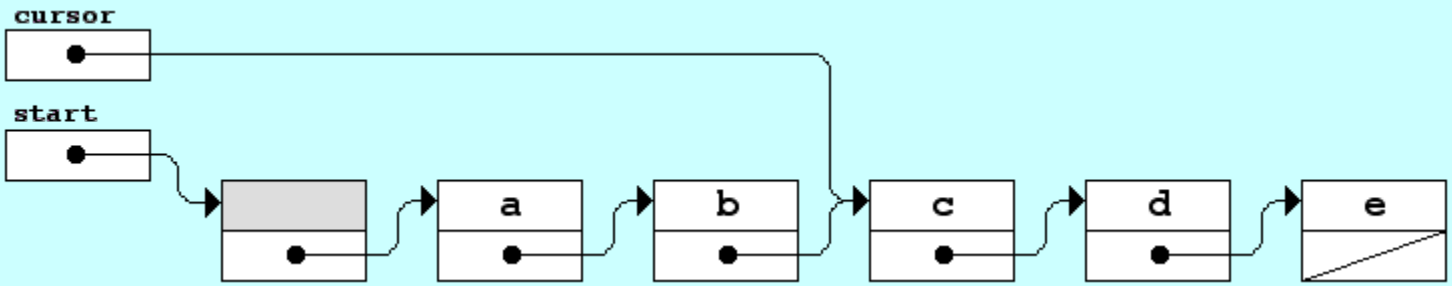
```
void EditorBuffer::moveCursorForward() {
   if (cursor->link != NULL) {
      cursor = cursor->link;
   }
}

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();
   }
}
```

Based Buffer Implementation

```
n notes: insertCharacter, deleteCharacter
----------------------------------------
dvantage of the linked list representation for
 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;
   }
}
```

Exercise: Representing the Cursor

- Can we represent the position of the cursor by pointing to the cell after the insertion point? For instance, if **`cursor`** is pointing to **`c`** in the five-character buffer, not like before, it should now mean the cursor is between cell **`b`** and cell **`c`**:



- First, we should probably move the dummy cell to the end. Anything else?
- Can we achieve the similar complexity on the six fundamental methods?

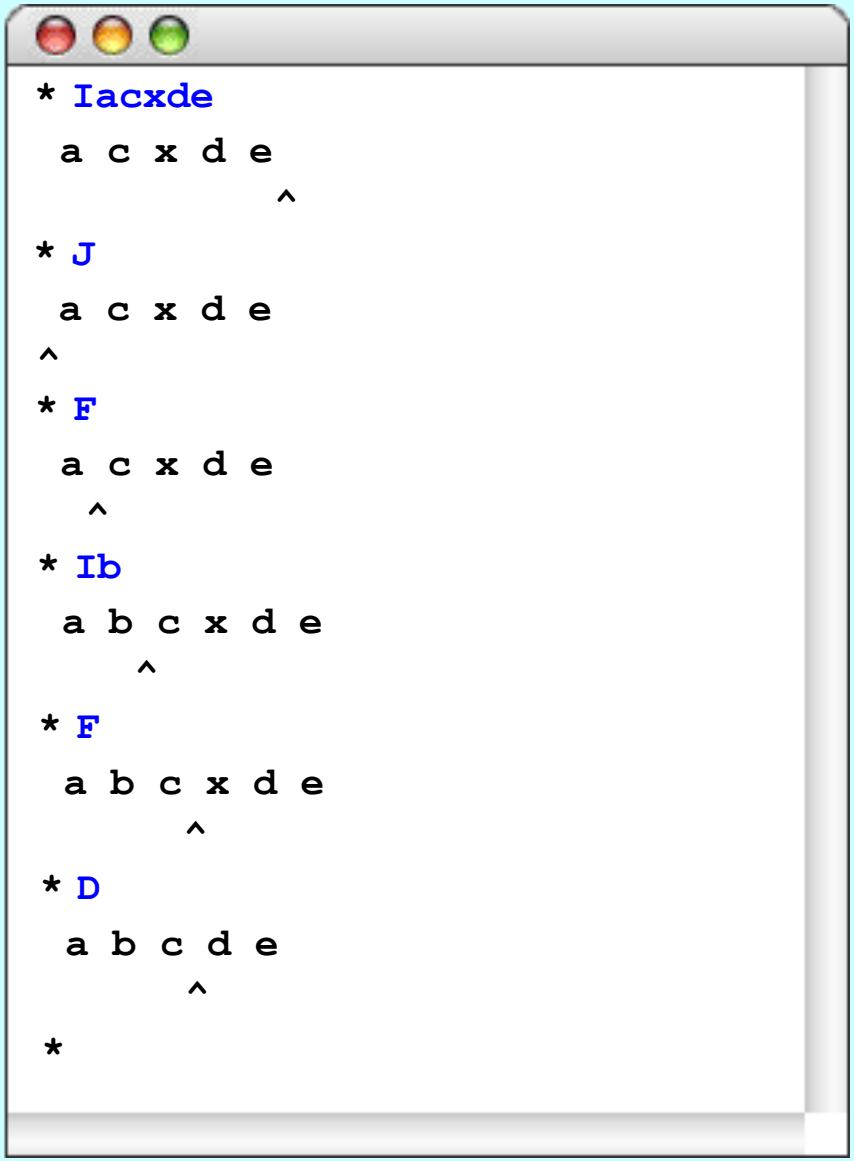
The Two-Stack Model

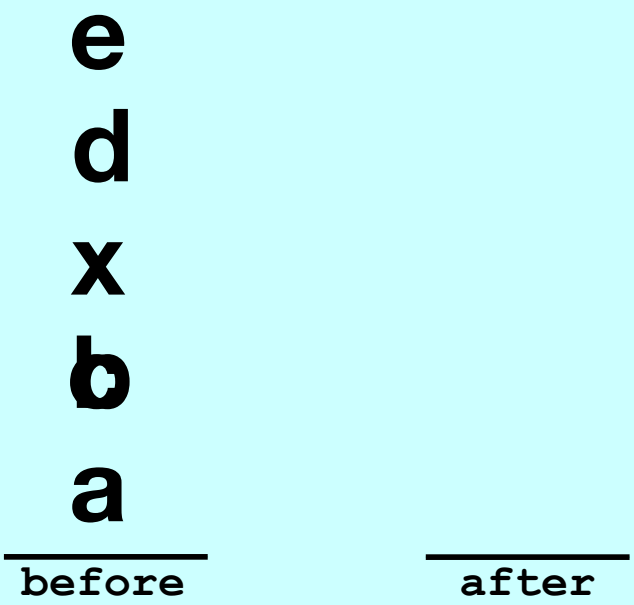
- In the two-stack implementation of the **`EditorBuffer`** class, the characters in the buffer are stored in one of two stacks. The characters before the cursor are stored in a stack called **`before`** and the characters after the cursor are stored in a stack called **`after`**. Characters in each stack are stored so that the ones close to the cursor are near the top of the stack.
- For example, given the buffer contents

  the characters would be stored like this:

Stack Editor Simulation





Private Data for Stack-Based Buffer

```
private:

/*
 * Implementation notes: Buffer data structure
 * -------------------------------------------
 * In the stack-based buffer model, the characters are stored in two
 * stacks.  Characters before the cursor are stored in a stack named
 * "before"; characters after the cursor are stored in a stack named
 * "after".  In each case, the characters closest to the cursor are
 * closer to the top of the stack.  The advantage of this
 * representation is that insertion and deletion at the current
 * cursor position occurs in constant time.
 */

#include "charstack.h"

/* Instance variables */

   CharStack before;    /* Stack of characters before the cursor */
   CharStack after;     /* Stack of characters after the cursor  */
```

Stack-Based Buffer Implementation

```
/*
 * File: buffer.cpp (stack version)
 * --------------------------------
 * This file implements the EditorBuffer class using a pair of
 * stacks to represent the buffer.  The characters before the
 * cursor are stored in the before stack, and the characters
 * after the cursor are stored in the after stack.
 */

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

/*
 * Implementation notes: EditorBuffer constructor/destructor
 * ----------------------------------------------------------
 * In this representation, the implementation of the CharStack class
 * automatically takes care of allocation and deallocation.
 */

EditorBuffer::EditorBuffer() {
   /* Empty */
}

EditorBuffer::~EditorBuffer() {
   /* Empty */
}
```

Stack-Based Buffer Implementation

```
/*
 * Implementation notes: moveCursor methods
 * ----------------------------------------
 * These methods use push and pop to transfer values between the two stacks.
 */

void EditorBuffer::moveCursorForward() {
   if (!after.isEmpty()) {
      before.push(after.pop());
   }
}

void EditorBuffer::moveCursorBackward() {
   if (!before.isEmpty()) {
      after.push(before.pop());
   }
}

void EditorBuffer::moveCursorToStart() {
   while (!before.isEmpty()) {
      after.push(before.pop());
   }
}

void EditorBuffer::moveCursorToEnd() {
   while (!after.isEmpty()) {
      before.push(after.pop());
   }
}
```

Stack-Based Buffer Implementation

```
/*
 * Implementation notes: insertCharacter and deleteCharacter
 * ---------------------------------------------------------
 * Each of the functions that inserts or deletes characters
 * can do so with a single push or pop operation.
 */

void EditorBuffer::insertCharacter(char ch) {
   before.push(ch);
}

void EditorBuffer::deleteCharacter() {
   if (!after.isEmpty()) {
      after.pop();
   }
}
```

Exercise:

```
string getText() const;

int getCursor() const;
```

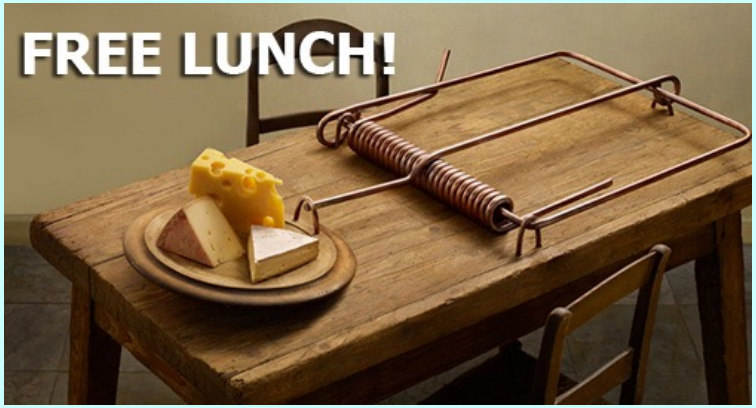
Complexity of the Editor Operations

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

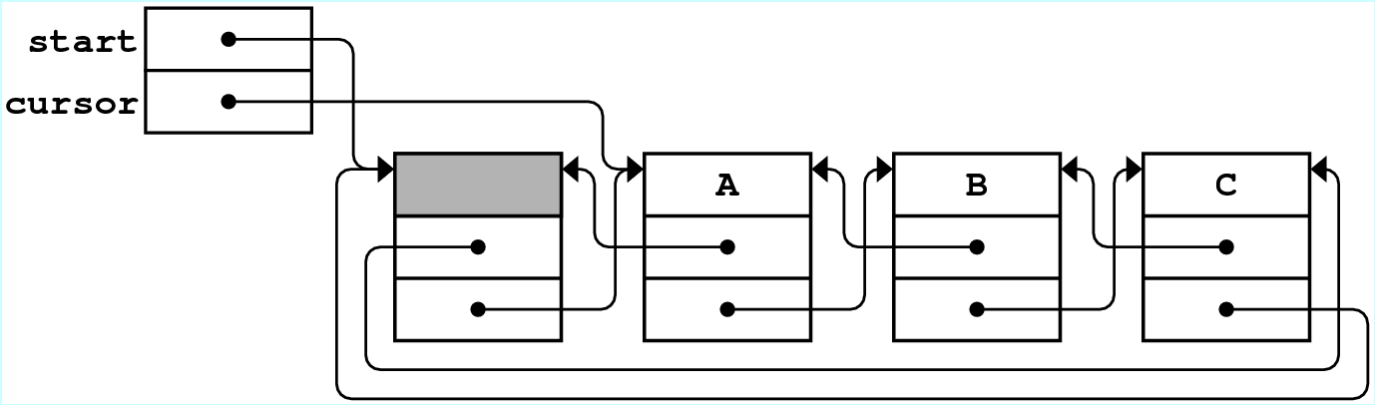
- The complexity of the stack-based operations may not be as straightforward as it appears, because it depends on the complexity of the operations of the **`CharStack`**, such as **`push`** and **`pop`**.
- If the underlying data structure of **`CharStack`** is array, what is the actual complexity of **`push`** (and **`insertCharacter`**)?

Exercise: reduce the complexity

- Is it possible to reimplement the editor buffer so that all six of these operations run in constant time?
- The answer is yes, but “there ain't no such thing as a free lunch”, so what is the price one has to pay?



- Time-space tradeoffs: doubly linked lists



The End