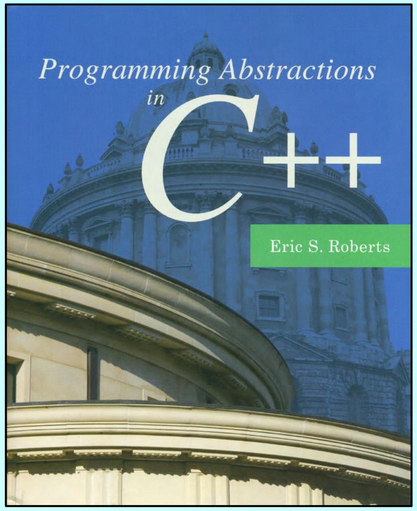
CHAPTER 13

Efficiency and Representation

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

—Tillie Olsen, *Silences,* 1965

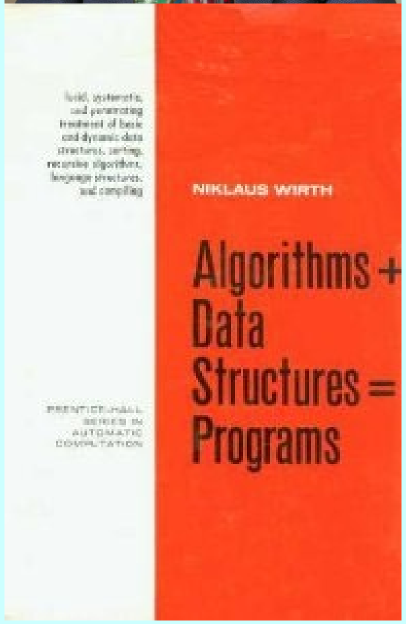


Flashback

What? … is a program

- A computer program is a collection of instructions that can be executed by a computer to perform a specific task.
- Niklaus Emil Wirth:
  - The programming language Pascal (a small, efficient language intended to encourage good programming practices using structured programming and data structuring)
  - Algorithms + Data Structures = Programs

Not only algorithms (what we discussed in the last chapter), but also data structures affect the efficiency of programs.

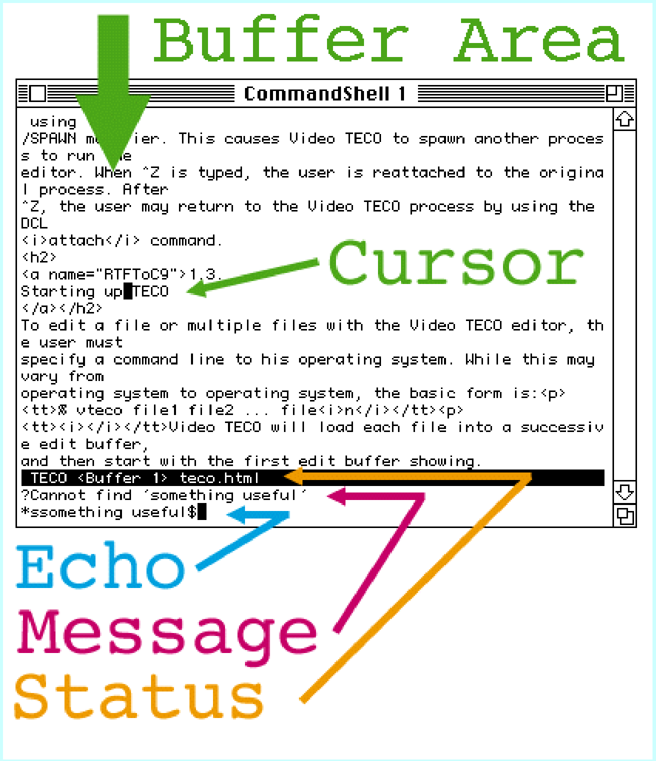

NIKLAUS WIRTH
Algorithms + Data Structures = Programs
PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION

Software Patterns for Editing Text

- 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.

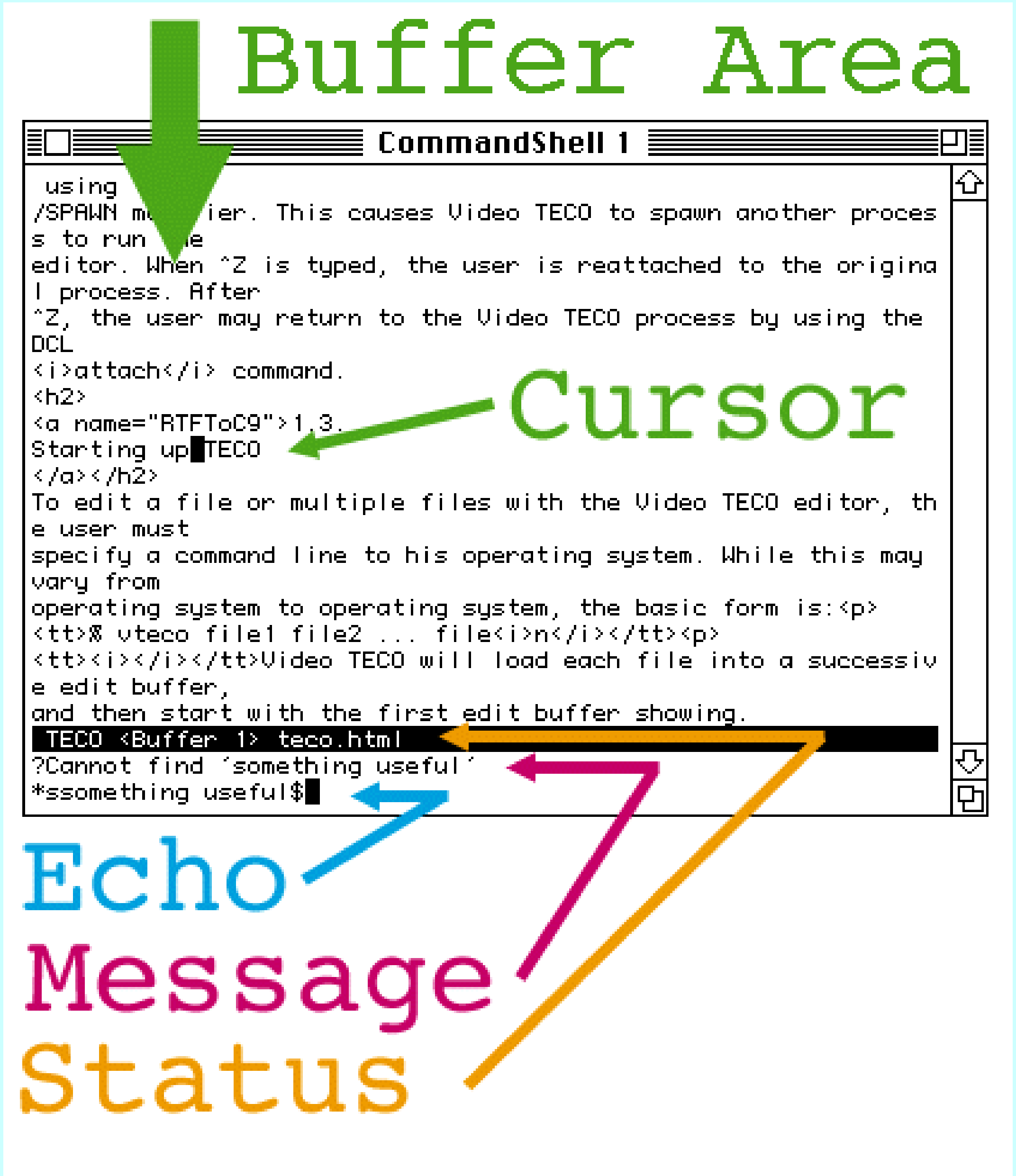
Why Look at Old Editors?

- The editor in the text uses the command-line model because:
  - It’s the best example of how using different data representations can have a profound effect on performance.
  - No modern editor is simple enough to understand in its entirety.
  - Command-line editors, e.g., TECO (Text Editor and COrrector), are historically important as the first extensible editors, and the basis of EMACS, which is still in use today.
  - The strategies used to representing the editor buffer persist in modern editor classes, such as Java’s **`JEditorPane`**.



```
 using
/SPAWN modifier. This causes Video TECO to spawn another proces
s to run the
editor. When ^Z is typed, the user is reattached to the origina
l process. After
^Z, the user may return to the Video TECO process by using the
DCL
<i>attach</i> command.
<h2>
<a name="RTFToC9">1.3.
Starting up█TECO
</a></h2>
To edit a file or multiple files with the Video TECO editor, th
e user must
specify a command line to his operating system. While this may
vary from
operating system to operating system, the basic form is:<p>
<tt>% vteco file1 file2 ... file<i>n</i></tt><p>
<tt><i></i></tt>Video TECO will load each file into a successiv
e edit buffer,
and then start with the first edit buffer showing.
 TECO <Buffer 1> teco.html
?Cannot find 'something useful'
*ssomething useful$█
```

Why Look at Old Editors?



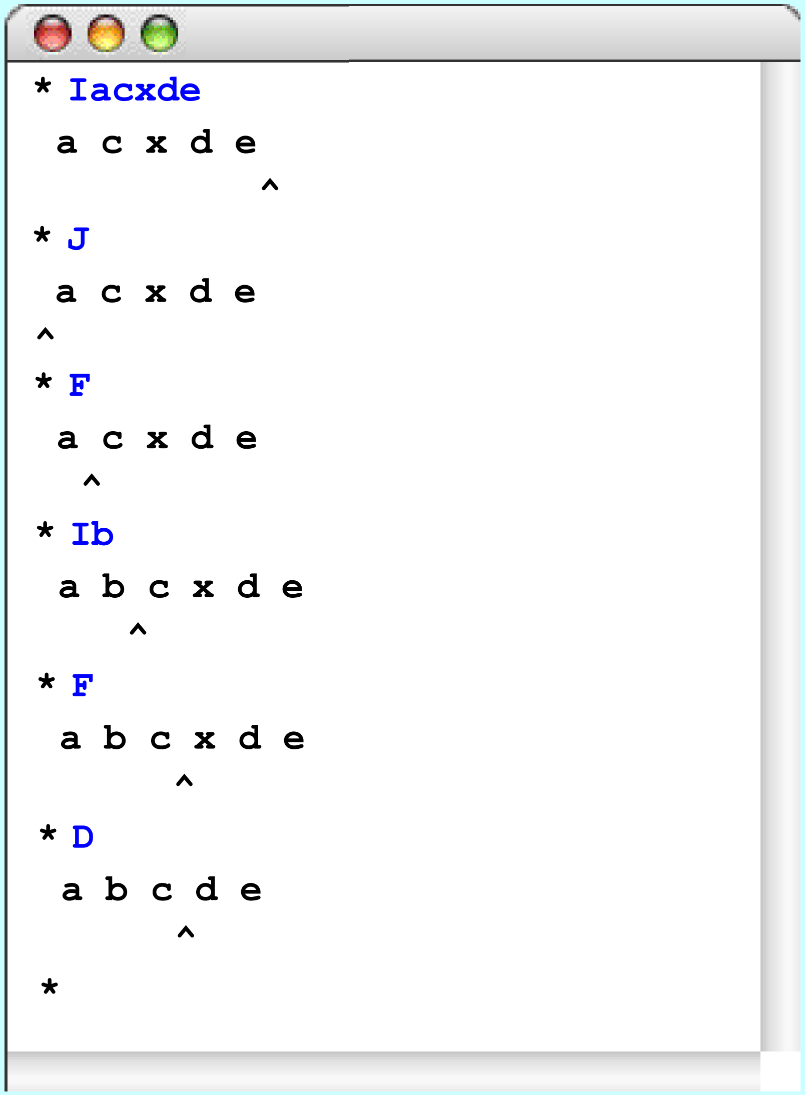
```
 using
/SPAWN m    ier. This causes Video TECO to spawn another proces
s to run   e
editor. When ^Z is typed, the user is reattached to the origina
l process. After
^Z, the user may return to the Video TECO process by using the
DCL
<i>attach</i> command.
<h2>
<a name="RTFToC9">1.3.
Starting up TECO
</a></h2>
To edit a file or multiple files with the Video TECO editor, th
e user must
specify a command line to his operating system. While this may
vary from
operating system to operating system, the basic form is:<p>
<tt>% vteco file1 file2 ... file<i>n</i></tt><p>
<tt><i></i></tt>Video TECO will load each file into a successiv
e edit buffer,
and then start with the first edit buffer showing.
 TECO <Buffer 1> teco.html
?Cannot find 'something useful'
*ssomething useful$
```

The Editor Commands

- Our minimal version of TECO has the following 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 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
      ^
*
```

Simple Editor Simulation

```
/*
 * Function: executeCommand
 * Usage: executeCommand(buffer, line);
 * ------------------------------------
 * Executes the command specified by 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;
   }
}
```

`line.substr(1)`

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.

The **buffer.h** Interface

```
/*
 * Methods: moveCursorForward, moveCursorBackward
 * Usage: buffer.moveCursorForward();
 *        buffer.moveCursorBackward();
 * -----------------------------------
 * These functions move the cursor forward or backward one
 * character, respectively.  If you call moveCursorForward
 * at the end of the buffer or moveCursorBackward at the
 * beginning, the function call has no effect.
 */

   void moveCursorForward();
   void moveCursorBackward();

/*
 * Methods: moveCursorToStart, moveCursorToEnd
 * Usage: buffer.moveCursorToStart();
 *        buffer.moveCursorToEnd();
 * --------------------------------
 * These functions move the cursor to the start or the end of this
 * buffer, respectively.
 */

   void moveCursorToStart();
   void moveCursorToEnd();
```

The **buffer.h** Interface

```
/*
 * Method: insertCharacter
 * Usage: buffer.insertCharacter(ch);
 * ----------------------------------
 * This function inserts the single character ch into this
 * buffer at the current cursor position.  The cursor is
 * positioned after the inserted character, which allows
 * for consecutive insertions.
 */

   void insertCharacter(char ch);

/*
 * Method: deleteCharacter
 * Usage: buffer.deleteCharacter();
 * --------------------------------
 * This function deletes the character immediately after
 * the cursor.  If the cursor is at the end of the buffer,
 * this function has no effect.
 */

   void deleteCharacter();
```

The `buffer.h` Interface

```
/*
 * Method: getText
 * Usage: string str = buffer.getText();
 * -------------------------------------
 * Returns the contents of the buffer as a string.
 */

   std::string getText() const;

/*
 * Method: getCursor
 * Usage: int cursor = buffer.getCursor();
 * ---------------------------------------
 * Returns the index of the cursor.
 */

   int getCursor() const;
```

Private section goes here.

```
}

#endif
```

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:

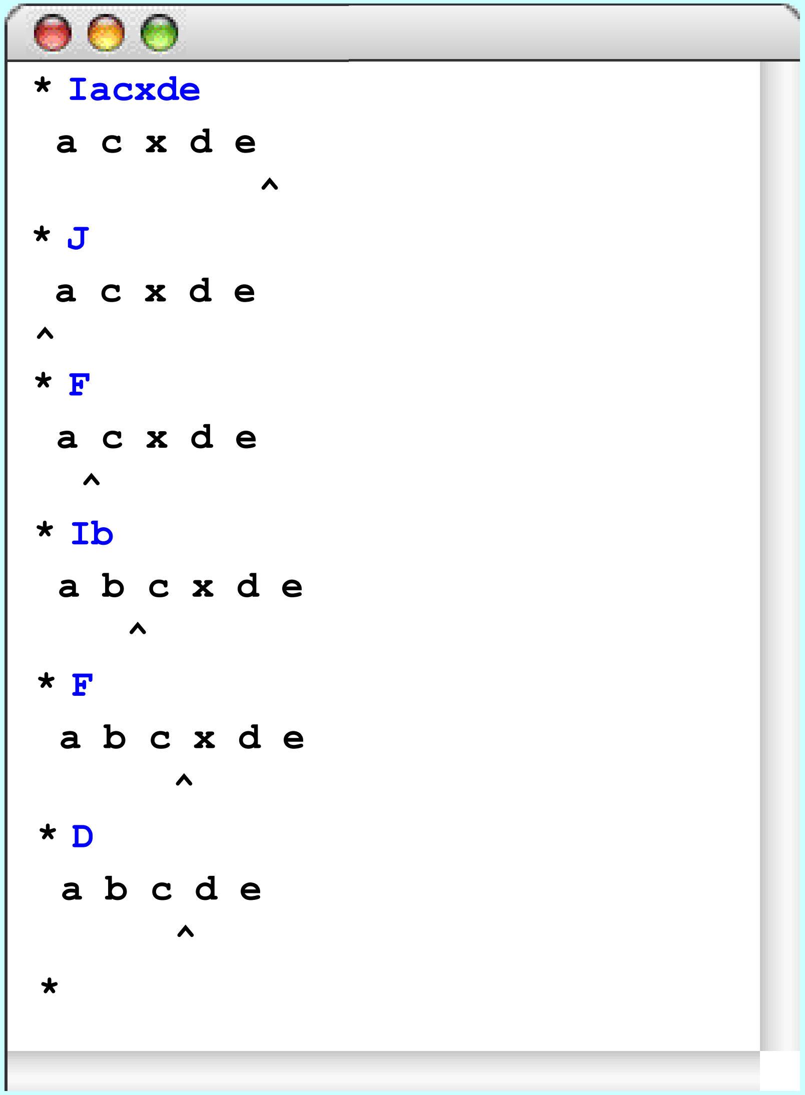
  1. A simple ***array model*** using dynamic allocation.
  2. A ***linked-list model*** that uses pointers to indicate the order.
  3. A ***two-stack model*** that uses a pair of character stacks.

- 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.

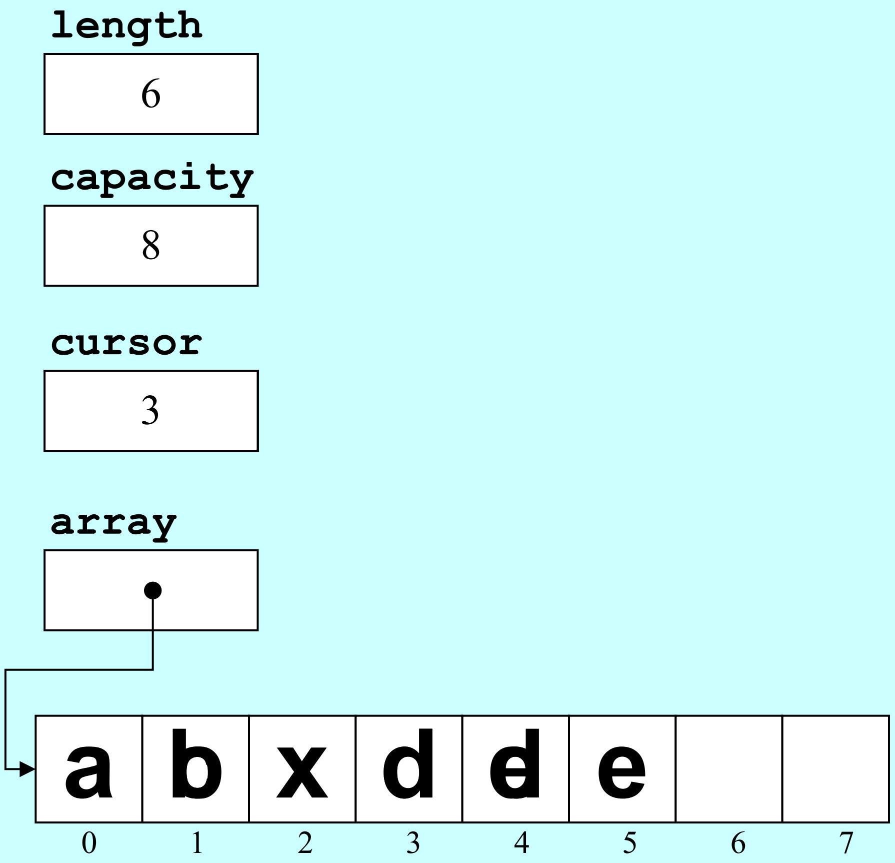
The Array Model

- 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



```
* 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 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

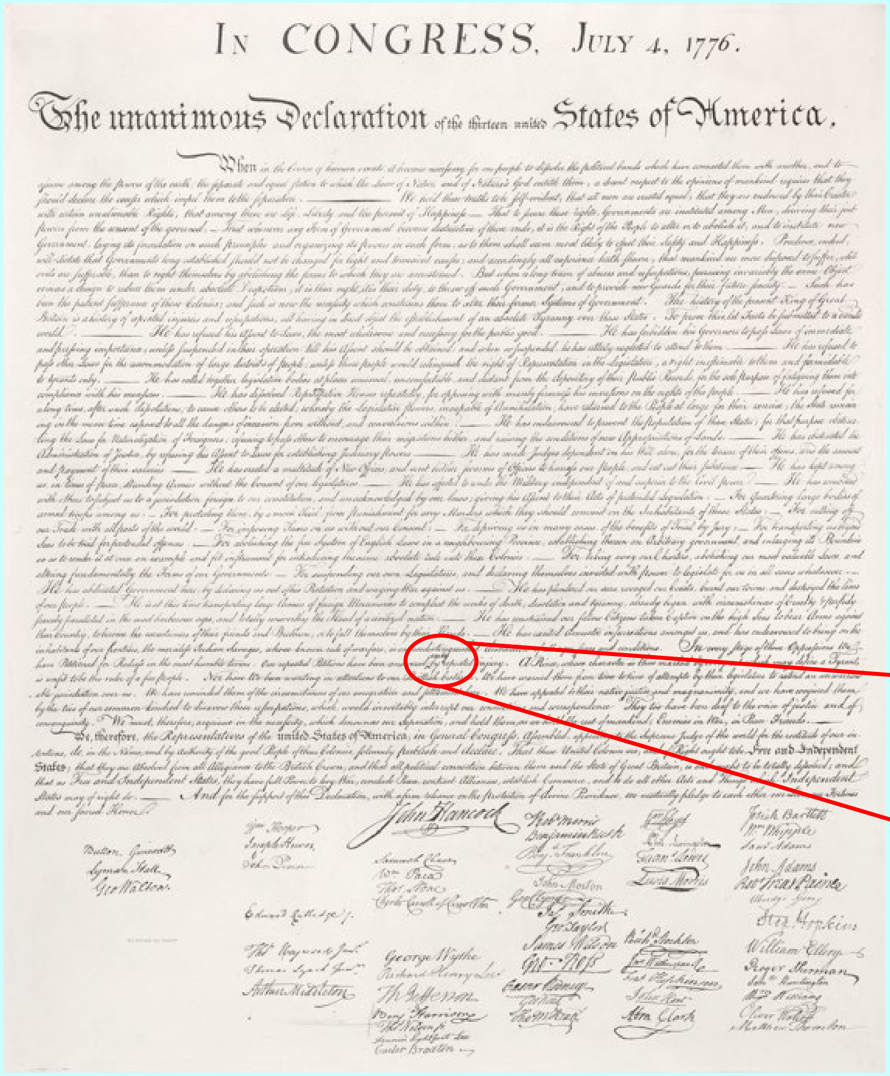
```
/*
 * Implementation notes: insertCharacter and deleteCharacter
 * --------------------------------------------------------
 * Each of the functions that inserts or deletes characters
 * must shift all subsequent characters in the array, either
 * to make room for new insertions or to close up space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



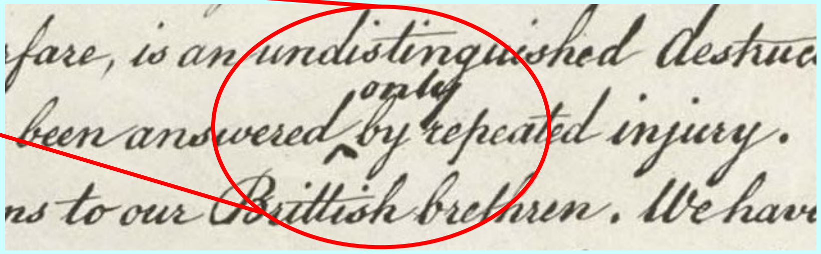
IN CONGRESS, JULY 4, 1776.

The unanimous Declaration of the thirteen united States of America,

[illegible]

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?

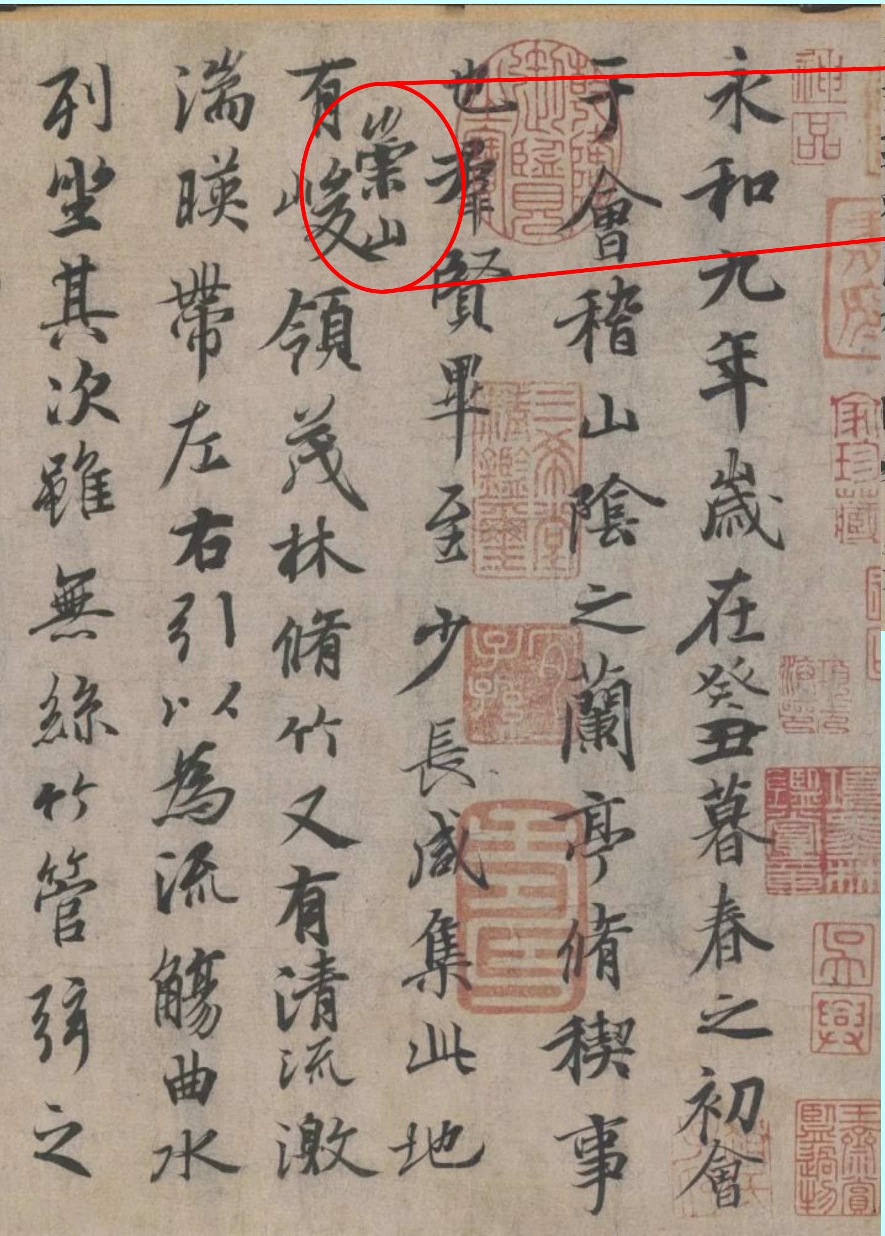


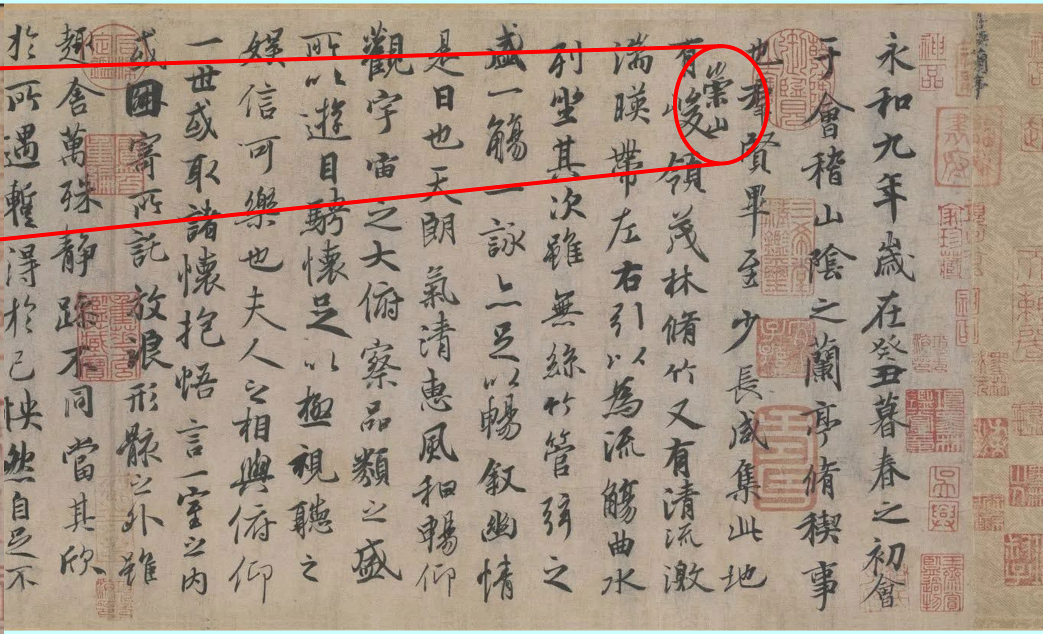
fare, is an undistinguished destruc

been answered $\overset{\text{only}}{\wedge}$ by repeated injury.

ns to our British brethren. We have

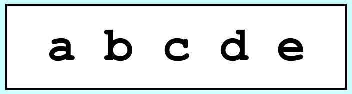
Insertion in Fixed Text


永和九年歲在癸丑暮春之初會
于會稽山陰之蘭亭脩稧事
也羣賢畢至少長咸集此地
有崇山峻領茂林脩竹又有清流激
湍暎帶左右引以為流觴曲水
列坐其次雖無絲竹管弦之

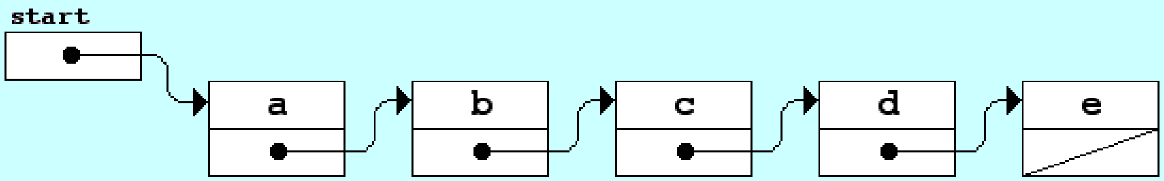

永和九年歲在癸丑暮春之初會
于會稽山陰之蘭亭脩稧事
也羣賢畢至少長咸集此地
有崇山峻領茂林脩竹又有清流激
湍暎帶左右引以為流觴曲水
列坐其次雖無絲竹管弦之
盛一觴一詠亦足以暢敘幽情
是日也天朗氣清惠風和暢仰
觀宇宙之大俯察品類之盛
所以遊目騁懷足以極視聽之
娛信可樂也夫人之相與俯仰
一世或取諸懷抱悟言一室之內
或因寄所託放浪形骸之外雖
趣舍萬殊靜躁不同當其欣
於所遇暫得於己快然自足不

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



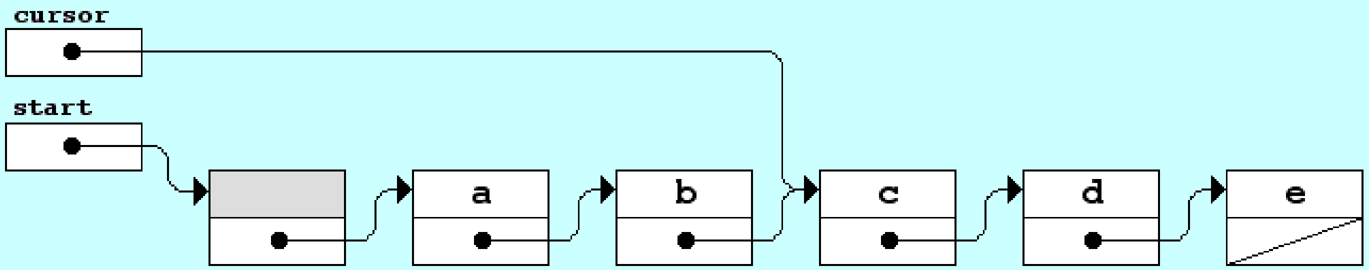
is modeled conceptually like this:



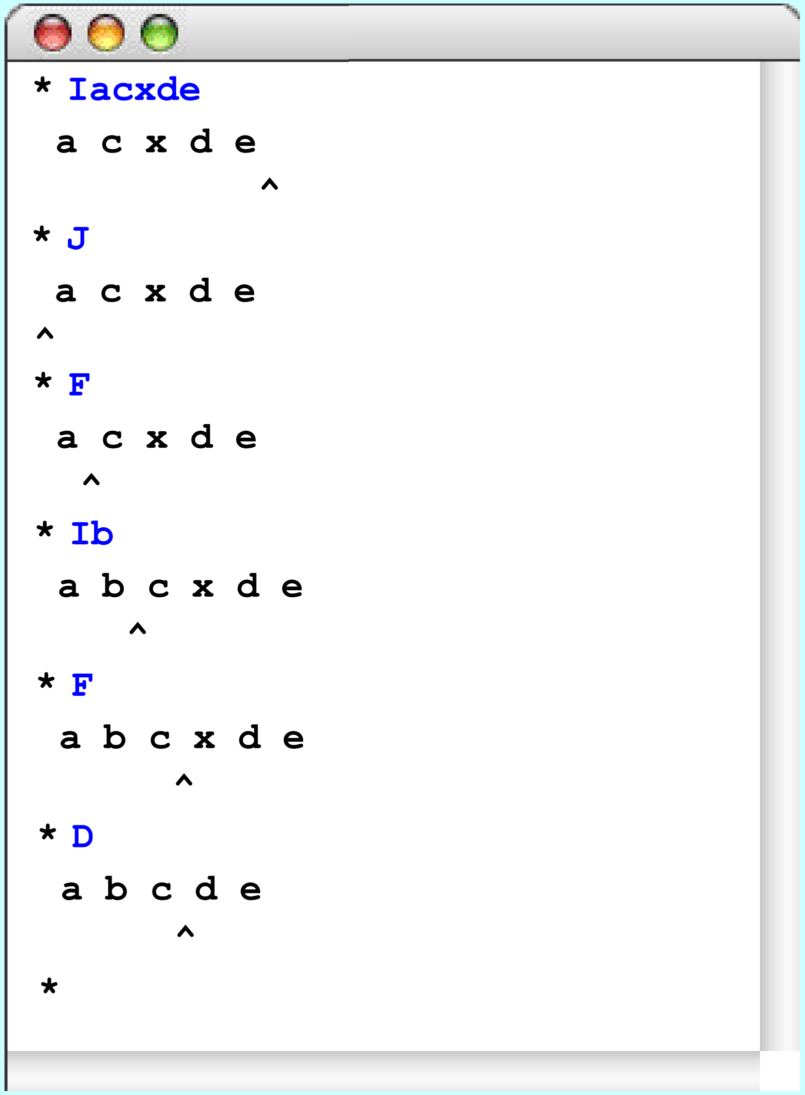
- The diagonal line through the last link pointer is used in list diagrams to indicate the **`NULL`** value that marks the end of the list.

Representing the Cursor

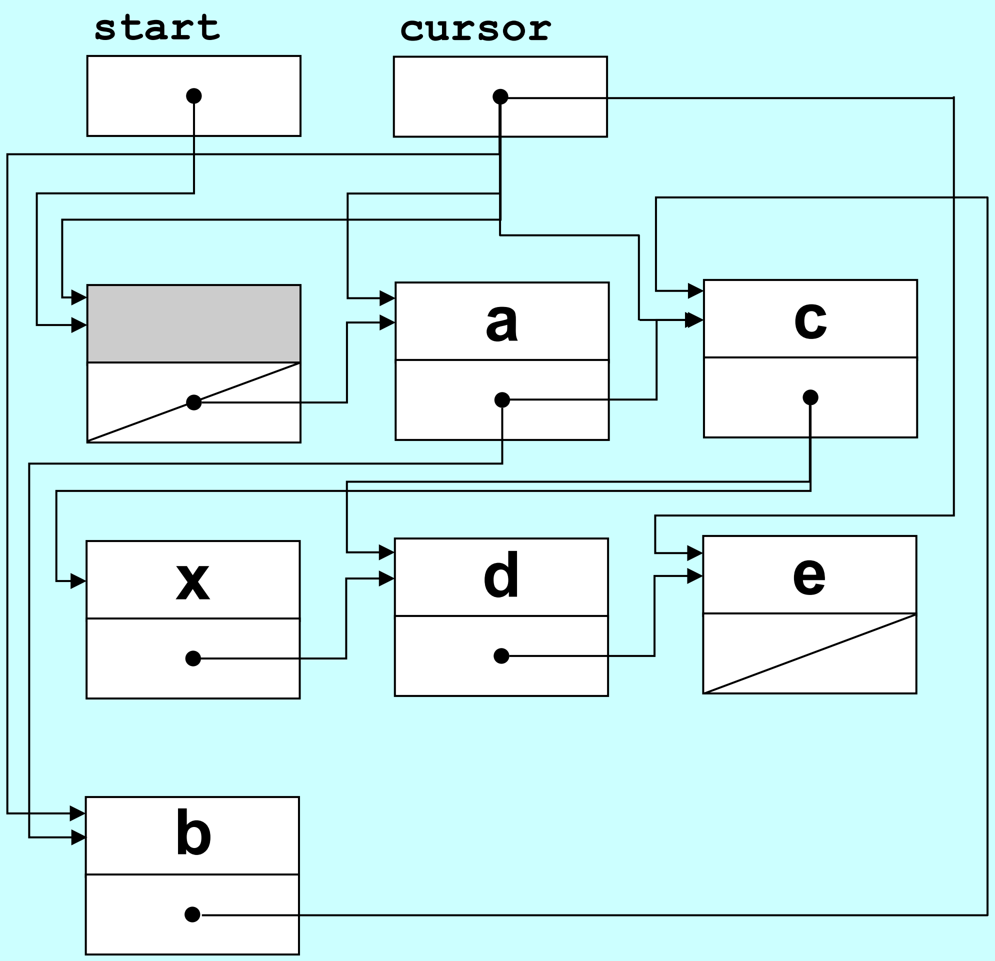
- The diagram on the preceding slide did not indicate the cursor position, mostly because doing so is a bit tricky.
- If a list contains ***five*** cells, as in this example, there are ***six*** positions for the cursor. Thus, it is impossible to represent all of the possible positions by pointing to 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

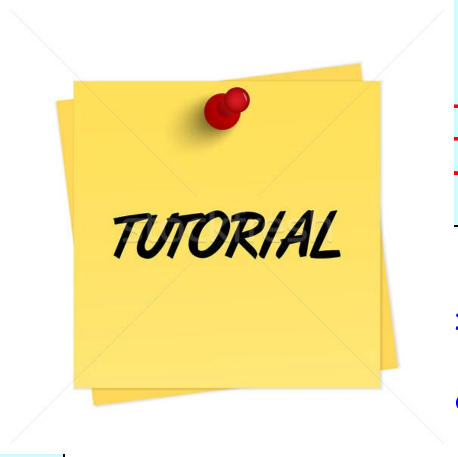
```
/*
 * File: buffer.cpp (list version)
 * -------------------------------
 * This file implements the EditorBuffer class using a linked
 * list to represent the buffer.
 */

#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();
   }
}
```

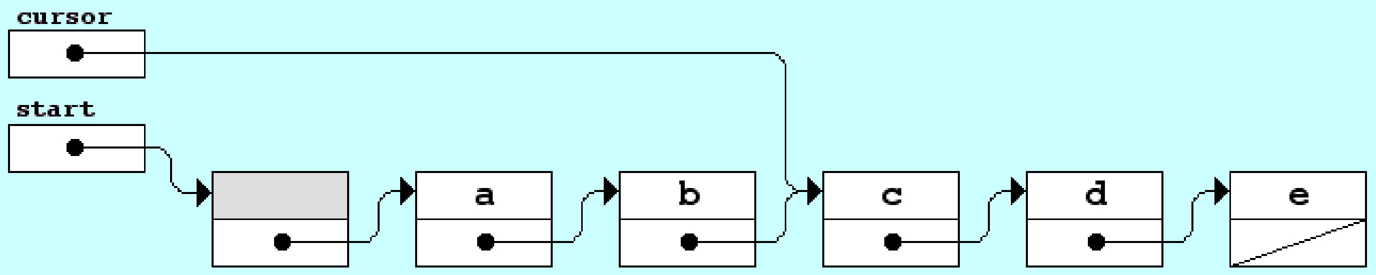


Based Buffer Implementation

```
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`**:

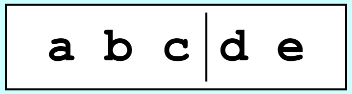


- First, we should probably move the dummy cell to the end. Anything else?
- Can we achieve the similar complexity on the six fundamental methods?

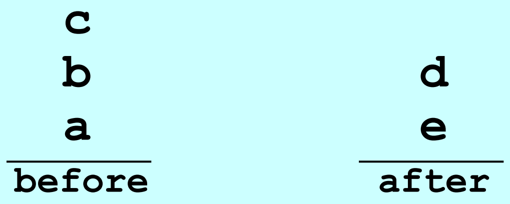
The Two-Stack Model

- 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.

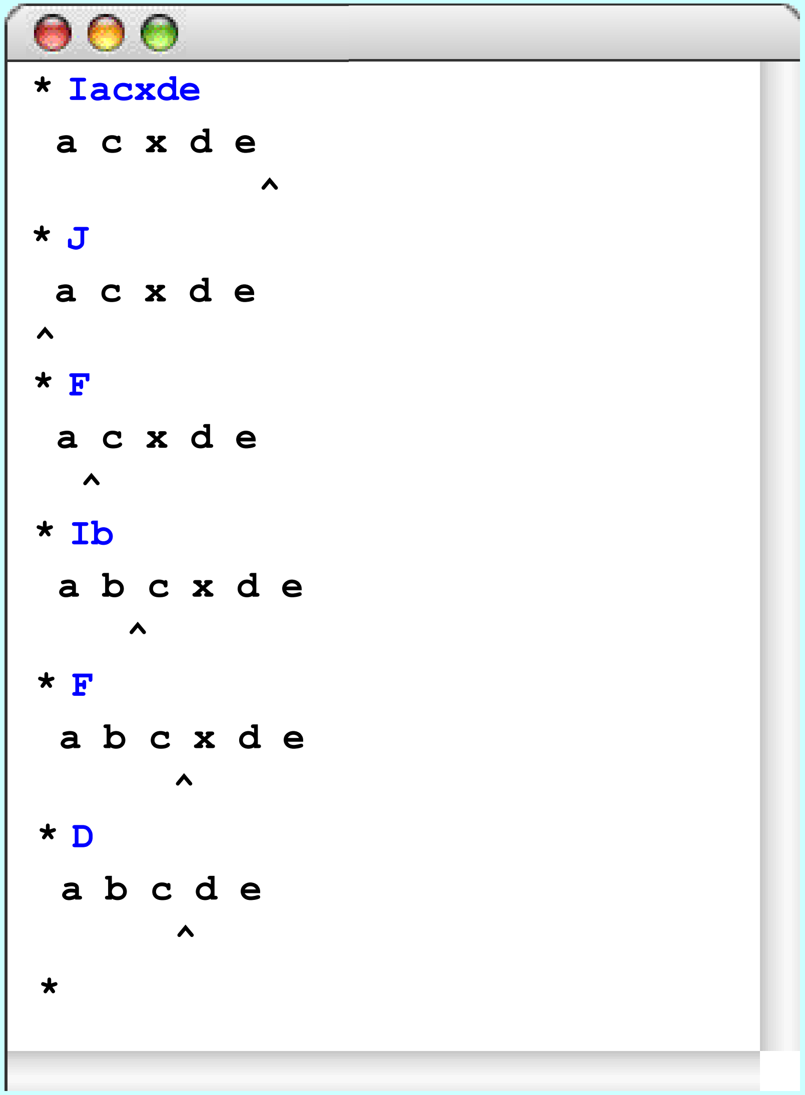
- For example, given the buffer contents



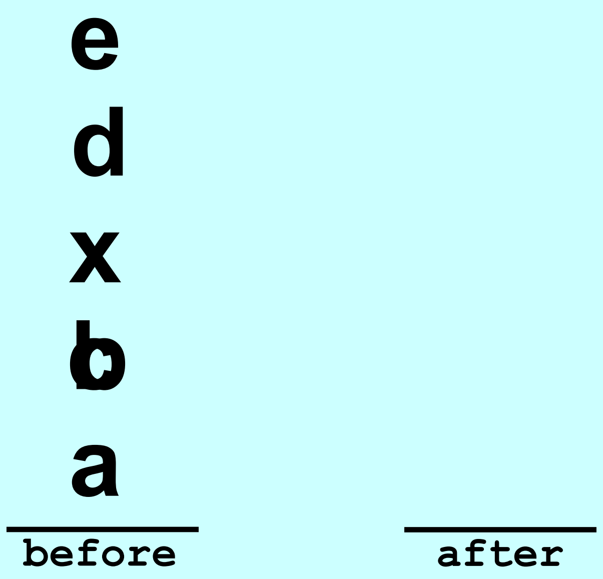
the characters would be stored like this:



Stack Editor Simulation



```
* 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
      ^
*
```



Flashback

Example: The `CharStack` Class

- Write classes that use dynamic allocation
- `CharStack`: a stack of characters

`CharStack cstk;`
Initializes an empty stack.

`cstk.size()`
Returns the number of characters pushed onto the stack.

`cstk.isEmpty()`
Returns **`true`** if the stack is empty.

`cstk.clear()`
Deletes all characters from the stack.

`cstk.push(ch)`
Pushes a new character onto the stack.

`cstk.pop()`
Removes and returns the top character from the stack.

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`)?

Flashback

The `charstackpriv.h` File

```
/*
 * File: charstackpriv.h
 * ---------------------
 * This file contains the private data for the CharStack class.
 */

private:

/* Instance variables */

   char *array;              /* Dynamic array of characters   */
   int capacity;             /* Allocated size of that array  */
   int count;                /* Current count of chars pushed */

/* Private function prototypes */

   void expandCapacity();
```

Flashback

The `charstack.cpp` Implementation

```
/*
 * File: charstack.cpp
 * -------------------
 * This file implements the CharStack class.
 */

void CharStack::push(char ch) {
   if (count == capacity) expandCapacity();
   array[count++] = ch;
}

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

Amortized Analysis

- *Worst-case* analysis of run time complexity is often too pessimistic.
- *Average-case* analysis may be difficult. What is "average"? "Random" might not be "average". Probabilistic analysis is often involved.
- In an ***amortized analysis***, we average the time required to perform a sequence of data-structure operations over all the operations performed.
- With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.
- Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case.

Mathematics of the Doubling Strategy

- The collection classes in this text all use the strategy of doubling their capacity whenever the structure runs out of room. Doing so guarantees that the ***amortized cost*** of inserting elements is $O(1)$.
- If α is the cost of an insertion that doesn't expand the capacity and β is the per-element cost of the linear expansion, the total cost of inserting N items is:

$$\textit{total time} = \alpha N + \beta\left(N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \cdots\right)$$

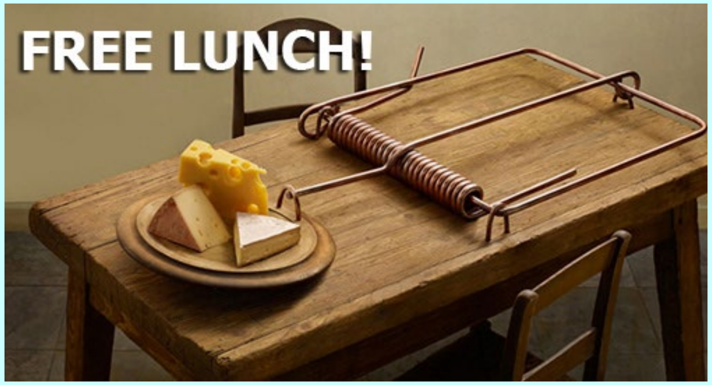
- Dividing by N gives the average time, as follows:

$$\textit{average time} = \alpha + \beta\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right)$$

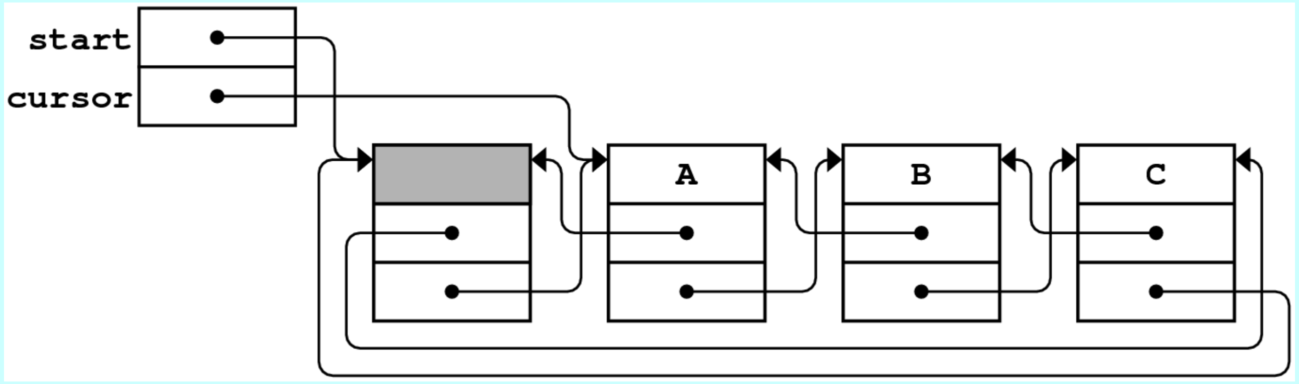
- The sum in parentheses is always less than two, so the average time of the worse case is constant. If deletion happens, the constant will be even lower.

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