Programming Abstraction in C++

Eric S. Roberts and Julie Zelenski

Stanford University
2010

Chapter 10. Efficiency and Data Representation

Outline

Outline

Introduction

Goal Use editor buffer as an example to illustrate how the choice of data representation affects the efficiency of applications.

Method Use a low-level built-in structure, such as character array, so the operations are visible and thus easier to assess efficiency.

Lesson Find options and evaluate the trade-offs. A good design demands compromise.

Important The external behavior of an editor buffer (Table 10-1, p. 340) must remain the same while implementation changes.

Operations

- Move the cursor forward one position
  `moveCursorForward()`
- Move the cursor backward one position
  `moveCursorBackward()`
- Jump the cursor to the beginning (before the first character)
  `moveCursorToStart()`
- Move the cursor to the end (after the last character)
  `moveCursorToEnd()`
- Insert a character at the current cursor position
  `insertCharacter(char ch)`
- Delete the character just after the cursor position
  `deleteCharacter()`
- Desplay the content of the buffer
  `display()`

Interface design

Constructor

```
EditorBuffer()
```

Destructor

```
~EditorBuffer()
```

Interface design

Constructor

```
EditorBuffer()
```

Destructor

```
~EditorBuffer()
```

Commands:

`F`: move forward
`B`: move backward
`J`: jump to beginning
`E`: jump to end
`Ixxx`: insert characters xxx
`D`: delete
`Q`: quit editor

Interface design (cont.)

The interface, the public section, Figure 10-1, p. 343-344.
Study

- Documentation
- Style (boilerplate, class definition)
- The public method prototypes
- The private section is included from a file `bufpriv.h`

Interface design (cont.)

The interface, the public section, Figure 10-1, p. 343-344.
Study

- Documentation
- Style (boilerplate, class definition)
- The public method prototypes
- The private section is included from a file `bufpriv.h`

Now that you have the interface, you can write an application program solely based on it, without knowing the implementation. The application program should not be affected when implementation changes.

Command-driven editor

Figure 10-2, p. 346

Pattern: command-driven editor

```
int main() {
    EditorBuffer buffer;
    while (true) {
        cout << "*";
        string cmd = GetLine();
        if (cmd != "") ExecuteCommand(buffer, cmd);
        buffer.display();
    }
    return 0;
}
```

Command-driven editor

Figure 10-2, p. 346

Pattern: command-driven editor

```
int main() {
    EditorBuffer buffer;
    while (true) {
        cout << "*";
        string cmd = GetLine();
        if (cmd != "") ExecuteCommand(buffer, cmd);
        buffer.display();
    }
    return 0;
}
```

A shell program is similar.

Command-driven editor (cont.)

```
void ExecuteCommand(EditorBuffer & buffer, string line) {
    switch (toupper(line[0]) {
      case 'I': for (int i = 1; i < line.length(); i++) {
                    buffer.insertCharacter(line[i]);
                }
                break;
      case 'D': buffer.deleteCharacter(); break;
      case 'F': buffer.moveCursorForward(); break;
      case 'B': buffer.moveCursorBackward(); break;
      case 'J': buffer.moveCursorToStart(); break;
      case 'E': buffer.moveCursorToEnd(); break;
      case 'Q': exit(0);
      default: cout << "Illegal command" << endl; break;
    }
}
```

Outline

Private data representation

Buffer A character array of fixed capacity, which can be extended like the dynamic `CharStack`.
Current length of the buffer.

cursor Position, the index of the character that immediately follows the cursor.

Private data representation

Buffer A character array of fixed capacity, which can be extended like the dynamic `CharStack`.
Current length of the buffer.

cursor Position, the index of the character that immediately follows the cursor.

Private instance variables:

```
char *array;
int capacity;
int length;
int cursor;
```

Implementing the methods

Moving cursor operations are straightforward.

Constructor and destructor, Figure 10-3, p. 349

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

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

moveCursorToEnd

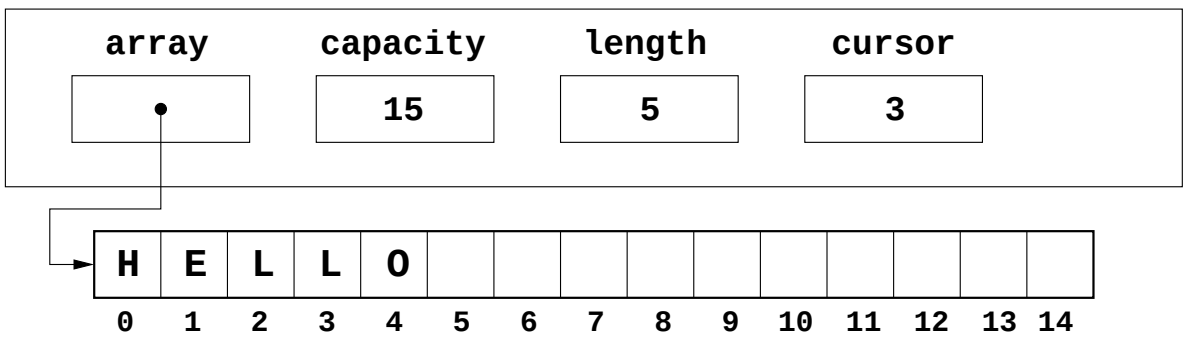
```
void EditorBuffer::moveCursorToEnd() {
    cursor = length;
}
```

The size of the array must be at least `length + 1`.

Insertion

```
buffer.insertCharacter('X');
```

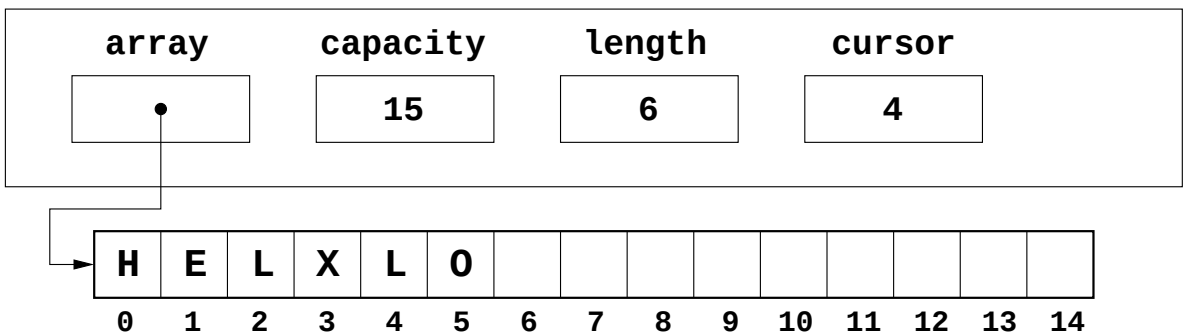
Before



Insertion (cont.)

```
buffer.insertCharacter('X');
```

After



Insertion (cont.)

Figure 10-3, p. 350

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

`expandCapacity` (p. 351) is similar to `CharStack` counterpart (p.326).

Insertion (cont.)

Figure 10-3, p. 350

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

`expandCapacity` (p. 351) is similar to `CharStack` counterpart (p.326).

`deleteCharacter` (p. 350) is similar.

Assessing complexity

Problem size N: buffer length

Operation count:

The operations (comparison, addition, assignment) in moving the cursor are independent of N (constant). No loops.

Assessing complexity

The operations of copying characters (assignment) in insertion and deletion are dependent of the buffer length. A loop. In the worst cases, inserting a character in the beginning or deleting a character in the beginning requires copying the entire buffer.

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

Assessing complexity (cont.)

Function	Complexity
`moveCursorForward`	$O(1)$
`moveCursorBackward`	$O(1)$
`moveCursorToStart`	$O(1)$
`moveCursorToEnd`	$O(1)$
`insertCharacter`	$O(N)$
`deleteCharacter`	$O(N)$

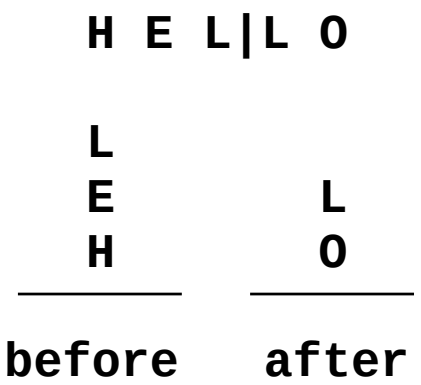
Note. When we do a lot of insertions and deletions and the buffer is large, it gets slow.

Outline

Using stacks

Idea:

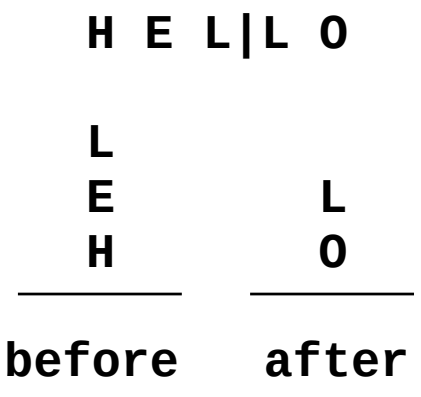
Breaking the buffer at the cursor boundary.
Two stacks: One contains the characters that precede the cursor; One contains the characters after the cursor.



Using stacks

Idea:

Breaking the buffer at the cursor boundary.
Two stacks: One contains the characters that precede the cursor; One contains the characters after the cursor.



Private data representation

```
CharStack before;
CharStack after;
```

Implementing the methods

Figure 10-4, pp. 355-356

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

`moveCursorBackward` is similar. No loops.

Implementing the methods

Figure 10-4, pp. 355-356

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

`insertCharacter` is similar. No loops.

Implementing the methods (cont.)

`moveCursorToStart` requires a loop.

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

Worst case: When the cursor is currently at the end, we have to move (assignment) the entire buffer.

Implementing the methods (cont.)

`moveCursorToStart` requires a loop.

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

Worst case: When the cursor is currently at the end, we have to move (assignment) the entire buffer.

`moveCursorToEnd` is similar. A loop.

Assessing complexity

Table 10-3, p. 354.

Function	Complexity
`moveCursorForward`	$O(1)$
`moveCursorBackward`	$O(1)$
`moveCursorToStart`	$O(N)$
`moveCursorToEnd`	$O(N)$
`insertCharacter`	$O(1)$
`deleteCharacter`	$O(1)$

Outline

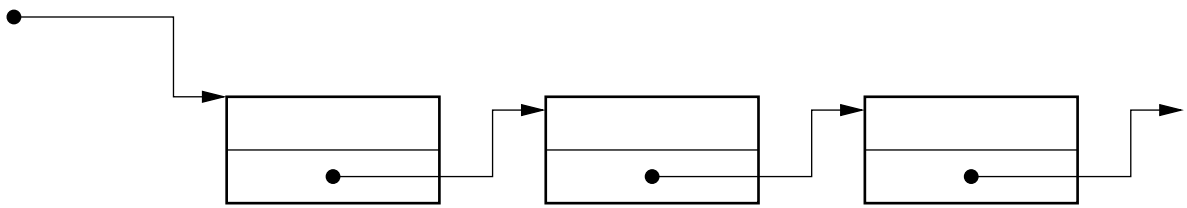
Linked list

What is a linked list? Linked by what?

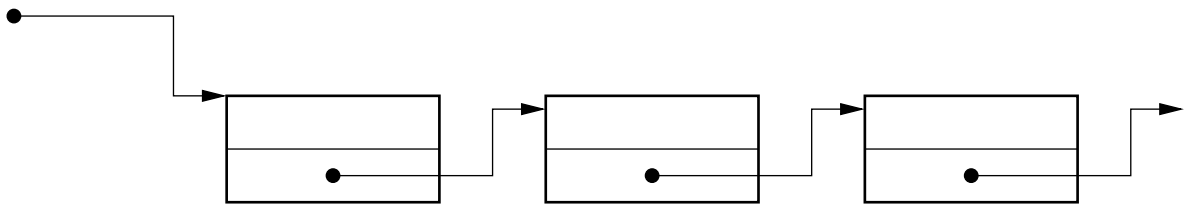
Linked list

What is a linked list? Linked by what?

A list of cells linked by pointers.



Cell structure



The structure of a cell:

```
struct cellT {
    char ch;
    cellT *link;
}
```

A recursive type.

Private data

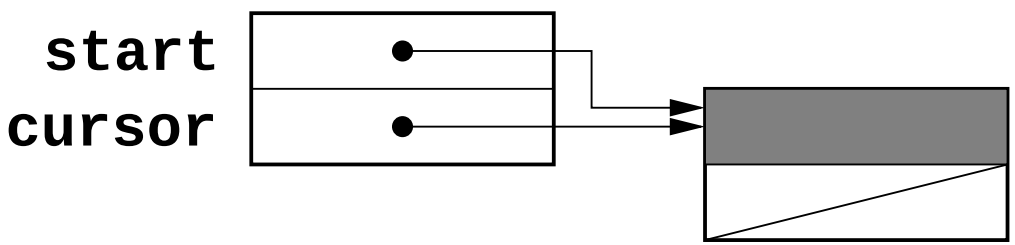
```
struct cellT {
    char ch;
    cellT *link;
};
cellT *start;
cellT *cursor;
```

Use the special pointer value `NULL` to signify the end of a list.

Use a dummy cell at the beginning of a list. (Why?)

Constructor

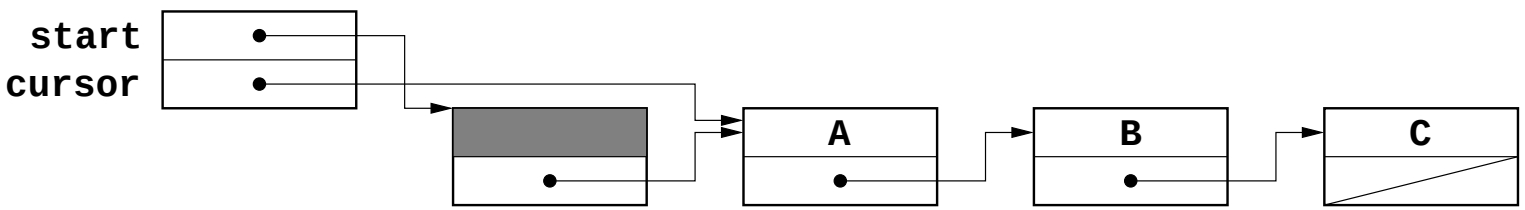
```
EditorBuffer::EditorBuffer() {
    // initialize dummy cell
    start = new cellT;
    start->link = NULL;
    // initialize cursor
    cursor = start;
}
```



A dummy cell.

Example

A|B C

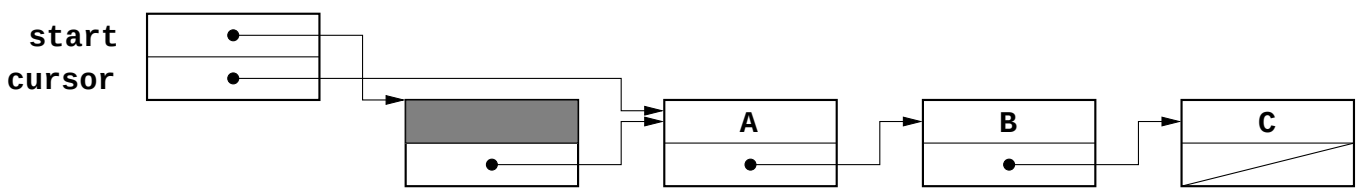


The cursor pointer points to the cell containing the character that immediately *precedes* the cursor.

Simple things first

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

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



Move the cursor backward

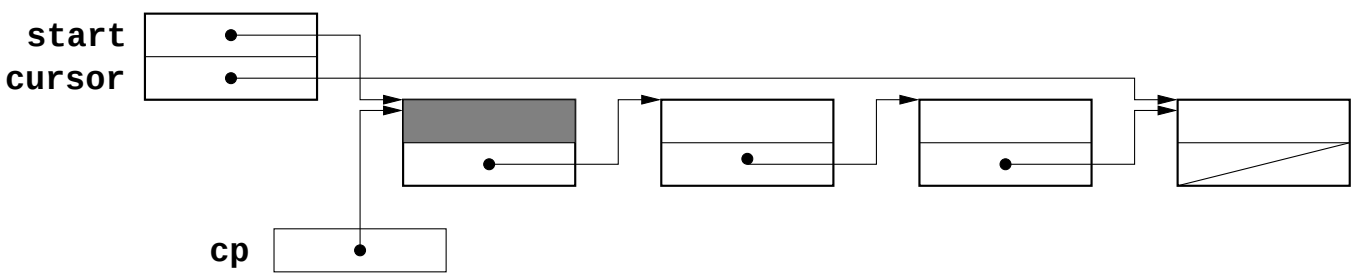
Search from the start for the pointer pointing to the cell whose `link` field equals `cursor`.

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

Requires a loop.

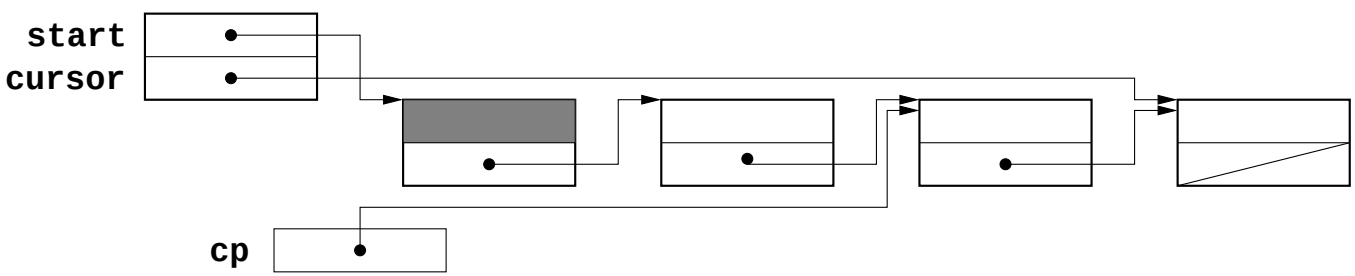
Move the cursor backward (cont.)

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



Move the cursor backward (cont.)

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



Move the cursor to the end

`moveCursorToEnd` is similar to `moveCursorBackward`.

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

Two patterns

Two linked list patterns

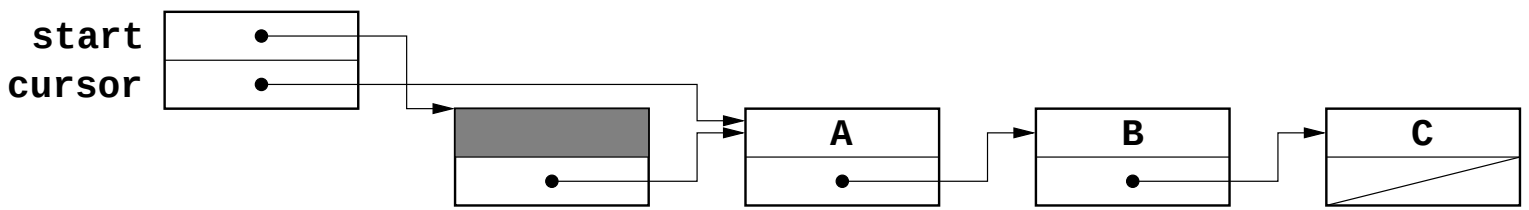
```
for (cp = start; cp->link != cursor; cp = cp->link) {
    ... code using cp ...
}


for (cellT *cp = start; cp != NULL; cp = cp->link) {
    ... code using cp ...
}
```

Example. `display`, p. 370.

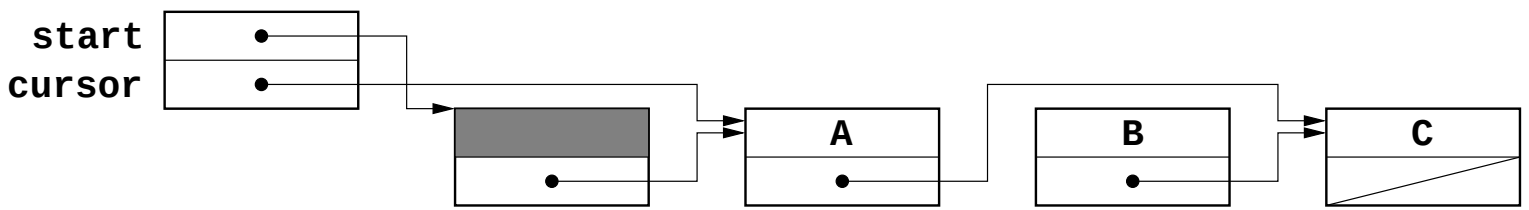
Deletion

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



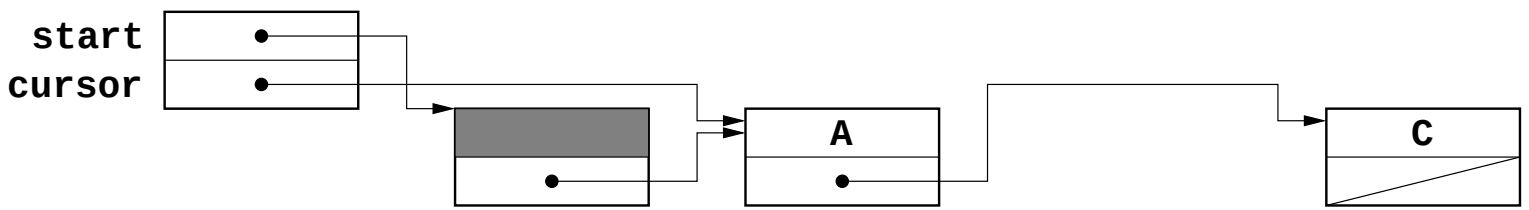
Deletion (cont.)

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



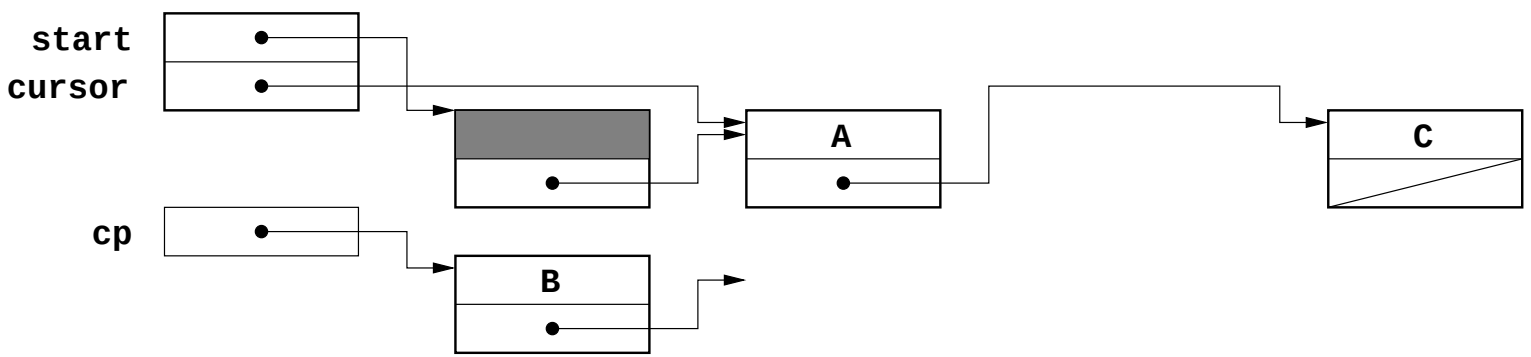
Deletion (cont.)

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



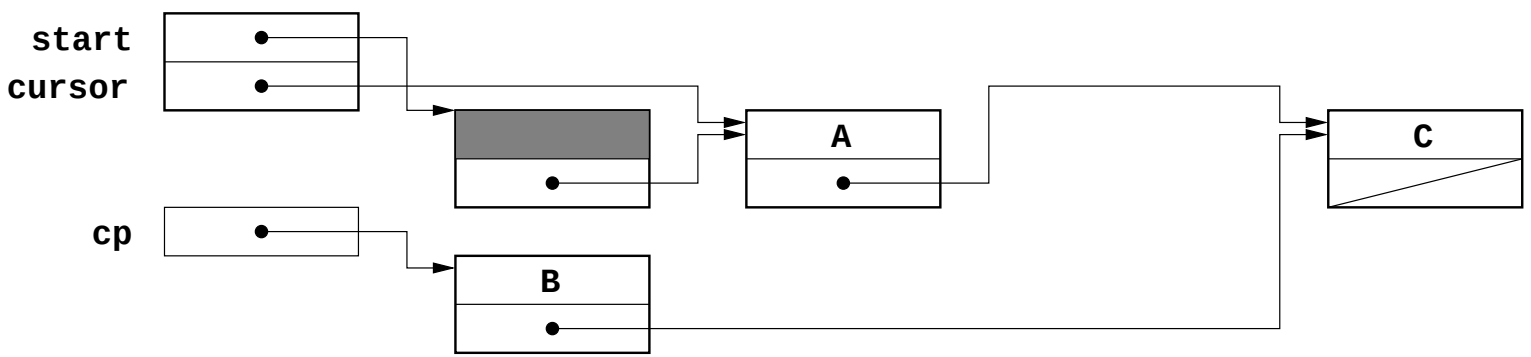
Insertion

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



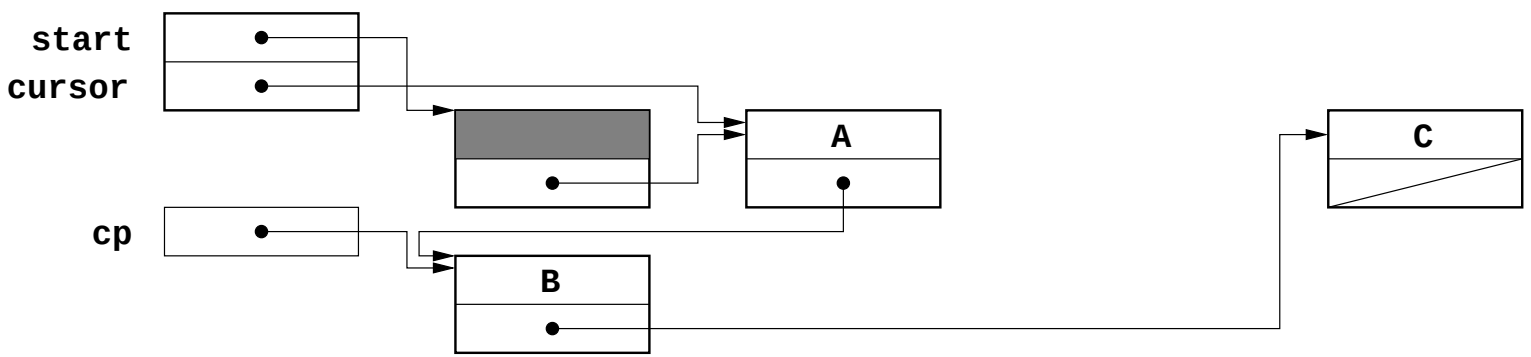
Insertion (cont.)

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



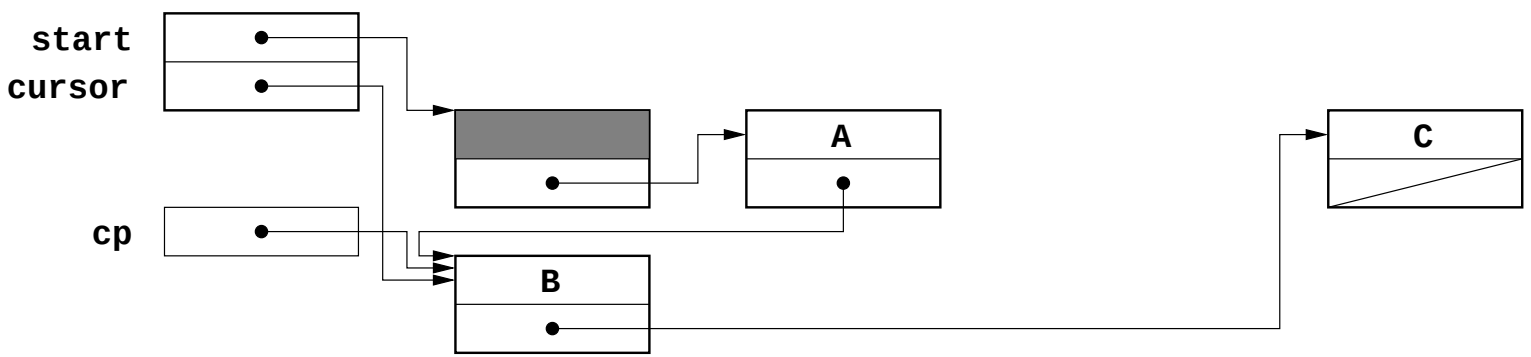
Insertion (cont.)

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



Insertion (cont.)

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



Destructor

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

Note.

- `cp` points to the cell to be deleted.
- Save the `link` field of the current cell in `next` before deleting the cell.

Relative efficiency

Table 10-4, p. 367.

Function	Array	Stack	List
`moveCursorForward`	$O(1)$	$O(1)$	$O(1)$
`moveCursorBackward`	$O(1)$	$O(1)$	$O(N)$
`moveCursorToStart`	$O(1)$	$O(N)$	$O(1)$
`moveCursorToEnd`	$O(1)$	$O(N)$	$O(N)$
`insertCharacter`	$O(N)$	$O(1)$	$O(1)$
`deleteCharacter`	$O(N)$	$O(1)$	$O(1)$

Relative efficiency

Table 10-4, p. 367.

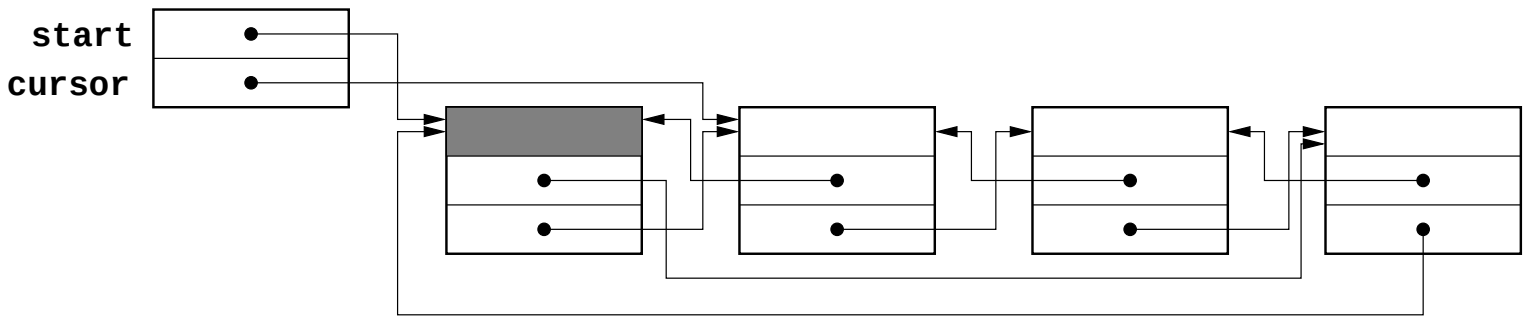
Function	Array	Stack	List
`moveCursorForward`	$O(1)$	$O(1)$	$O(1)$
`moveCursorBackward`	$O(1)$	$O(1)$	$O(N)$
`moveCursorToStart`	$O(1)$	$O(N)$	$O(1)$
`moveCursorToEnd`	$O(1)$	$O(N)$	$O(N)$
`insertCharacter`	$O(N)$	$O(1)$	$O(1)$
`deleteCharacter`	$O(N)$	$O(1)$	$O(1)$

Question

If the cursor pointer pointed to the cell containing the character immediately after the cursor, how would it affect the efficiency?

Doubly linked list

To alleviate the problem of going backwards or to the end in linked list, we can used a circular doubly linked list with a dummy cell.




```
struct cellT {
    char ch;
    cellT *prev;
    cellT *next;
}
```

Time-space trade-offs

In the doubly linked list implementation, all the operations have $O(1)$ complexity, however, it takes at least nine bytes for each character, about ten times the space in the array implementation.

Time-space trade-offs

In the doubly linked list implementation, all the operations have $O(1)$ complexity, however, it takes at least nine bytes for each character, about ten times the space in the array implementation.

A hybrid method: A doubly linked list of arrays, where each array represents a line.

Time-space trade-offs

In the doubly linked list implementation, all the operations have $O(1)$ complexity, however, it takes at least nine bytes for each character, about ten times the space in the array implementation.

A hybrid method: A doubly linked list of arrays, where each array represents a line.

A good design demands compromise.