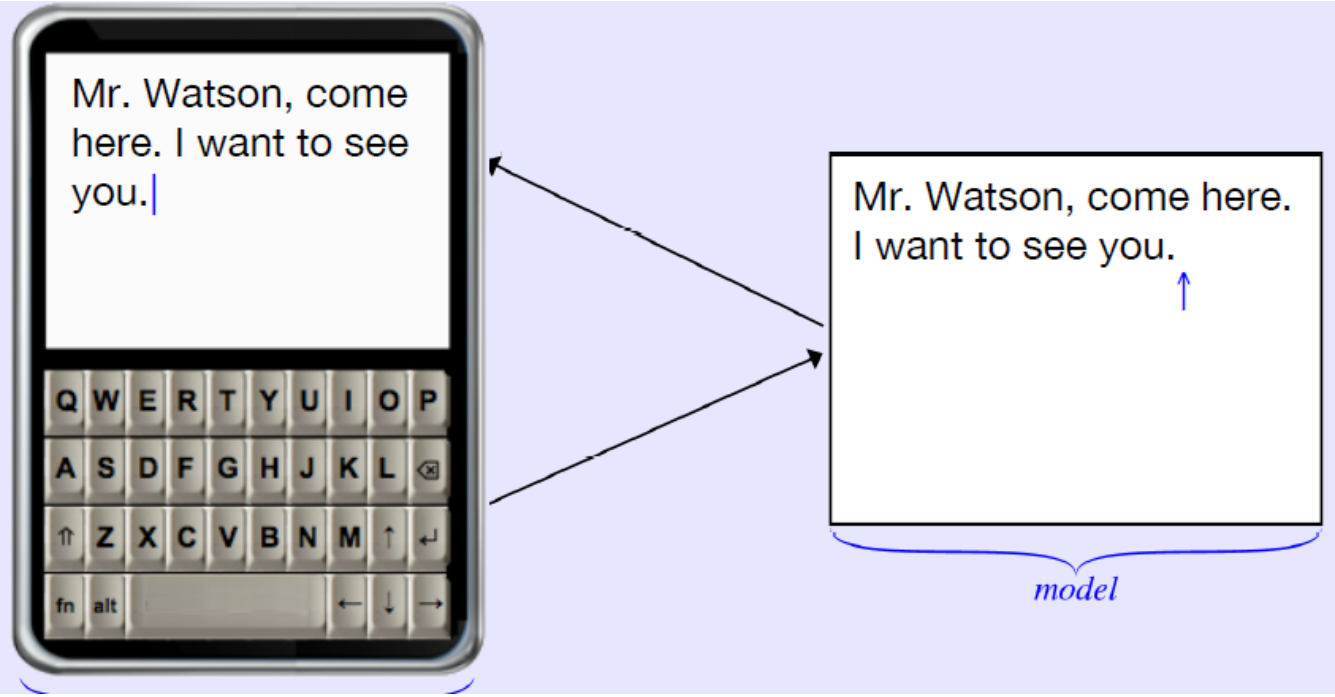
List-based Buffer Implementation

Laiyan Ding
117010053@link.cuhk.edu.cn

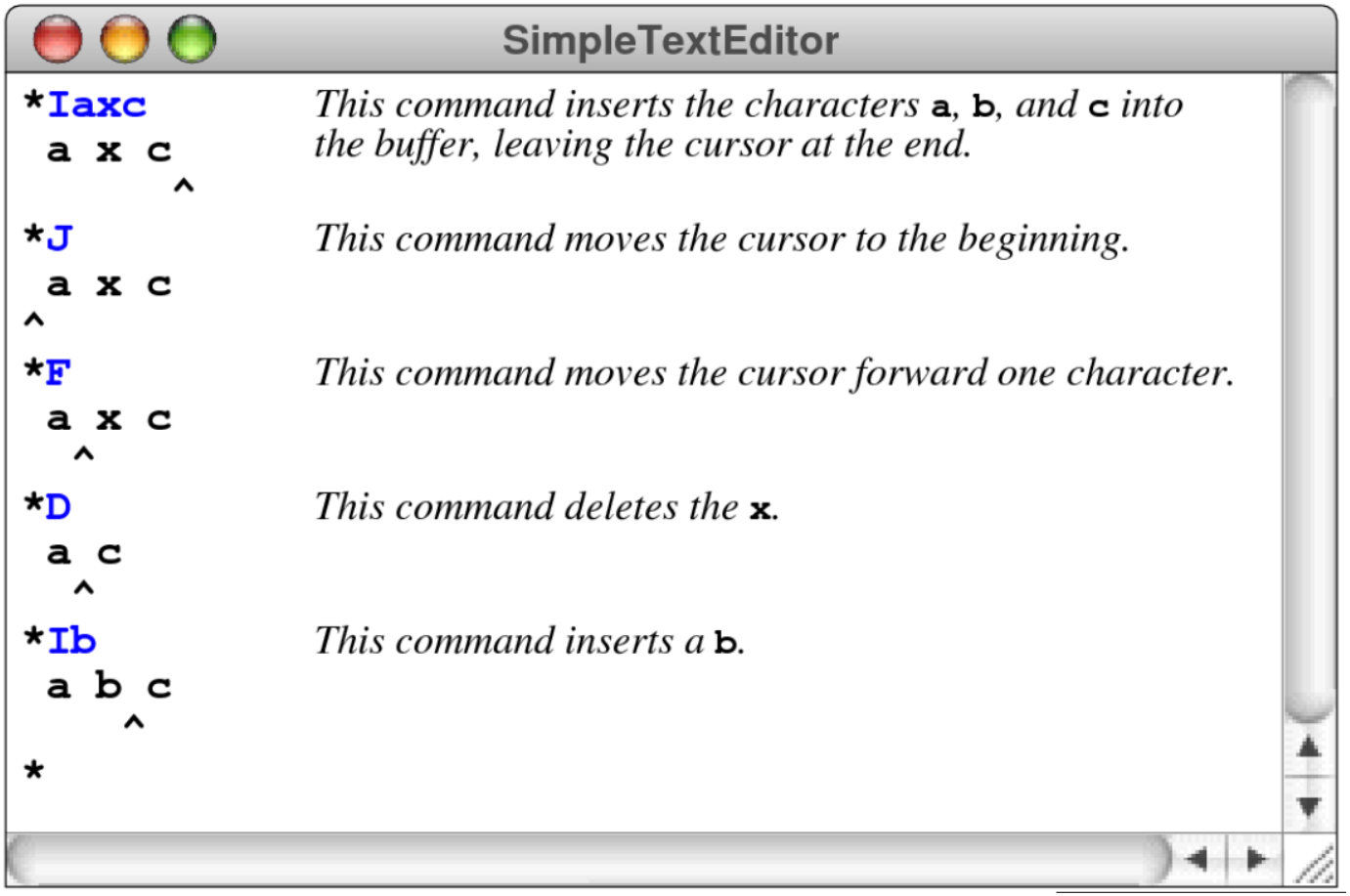
Editor application

- The strategy of editor buffer is widely used in the application of text editing.
- e.g., use mobile phone to edit messages which will be sent later.



A simple text editor

- Develop the buffer abstraction in the context of this command-driven style instead of a complicated one.



A simple text editor

- Commands

F	Moves the editing cursor forward one character position.
B	Moves the editing cursor backward one character position.
J	Jumps to the beginning of the buffer.
E	Moves the cursor to the end of the buffer.
I*xxx*	Inserts the characters *xxx* at the current cursor position.
D	Deletes the character just after the current cursor position.
H	Prints out a help message listing the commands.
Q	Quit the editor program.

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.

The Two-Stack Model

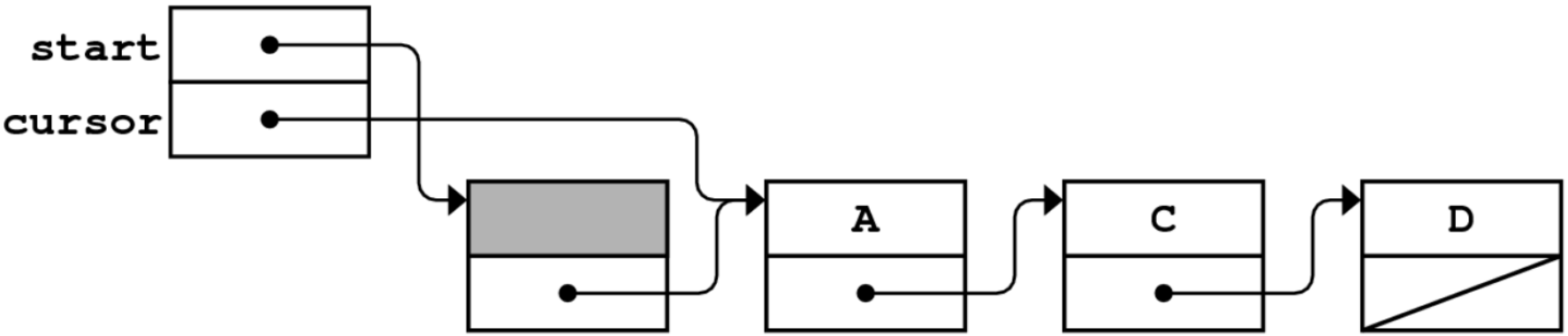
- In the two-stack implementation of the **`EditorBuffer`** class, the characters in the buffer are stored in one of two stacks. The characters before the cursor are stored in a stack called **`before`** and the characters after the cursor are stored in a stack called **`after`**. Characters in each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

  a b c|d e

  the characters would be stored like this:

Implementation of text editor

- Linked list based implementation(Chapter 13, page 24)
  - ✓ cell ;
  - ✓ start ;
  - ✓ cursor ;
  - ✓ use a dummy cell so that all possible cursor positions can be represented



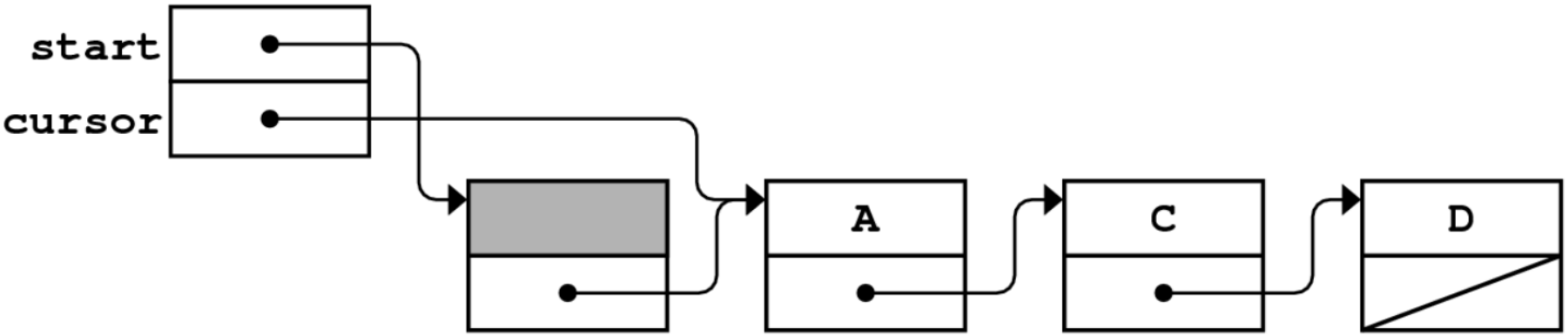
1. Insert Character
2. Delete Character

Insert Character

- Insert the letter `B` into a buffer that currently contains

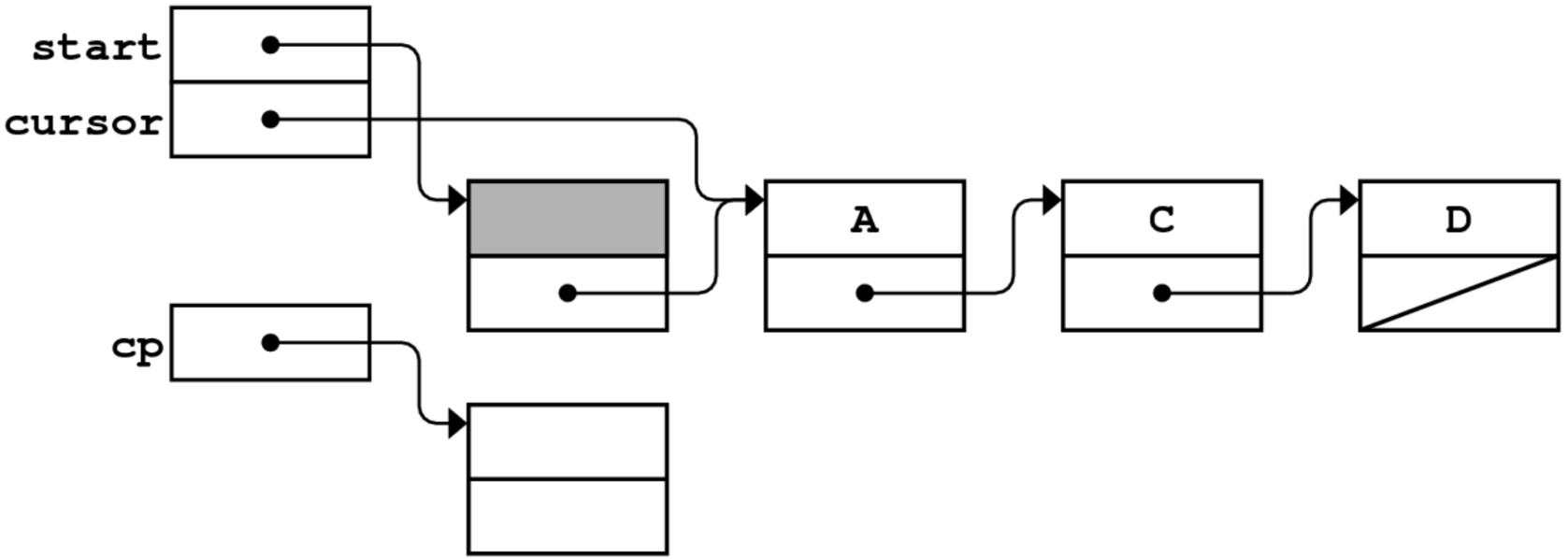
```
A C D
  ^
```

- The cursor is between the `A` and the `C` as shown.
- The situation prior to the insertion looks like this:



Insert Character

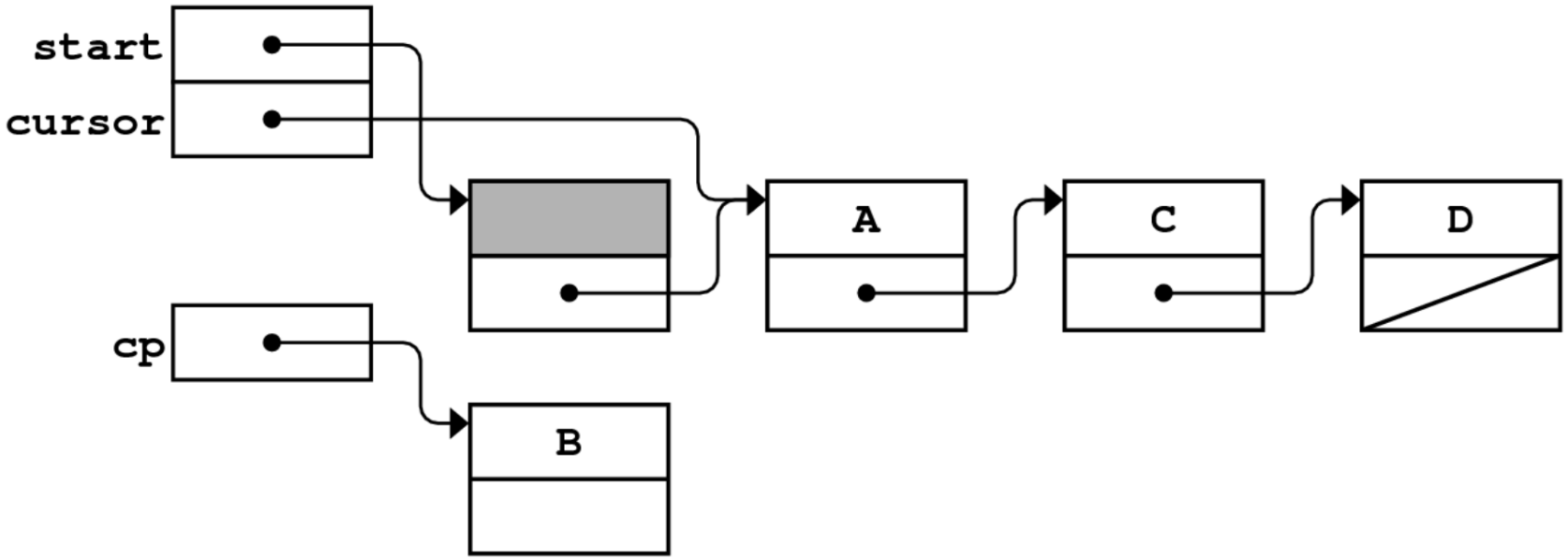
① Allocate a new cell and store a pointer to it in the variable `cp`




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

Insert Character

② Store the character `B` into the `ch` field of the new cell

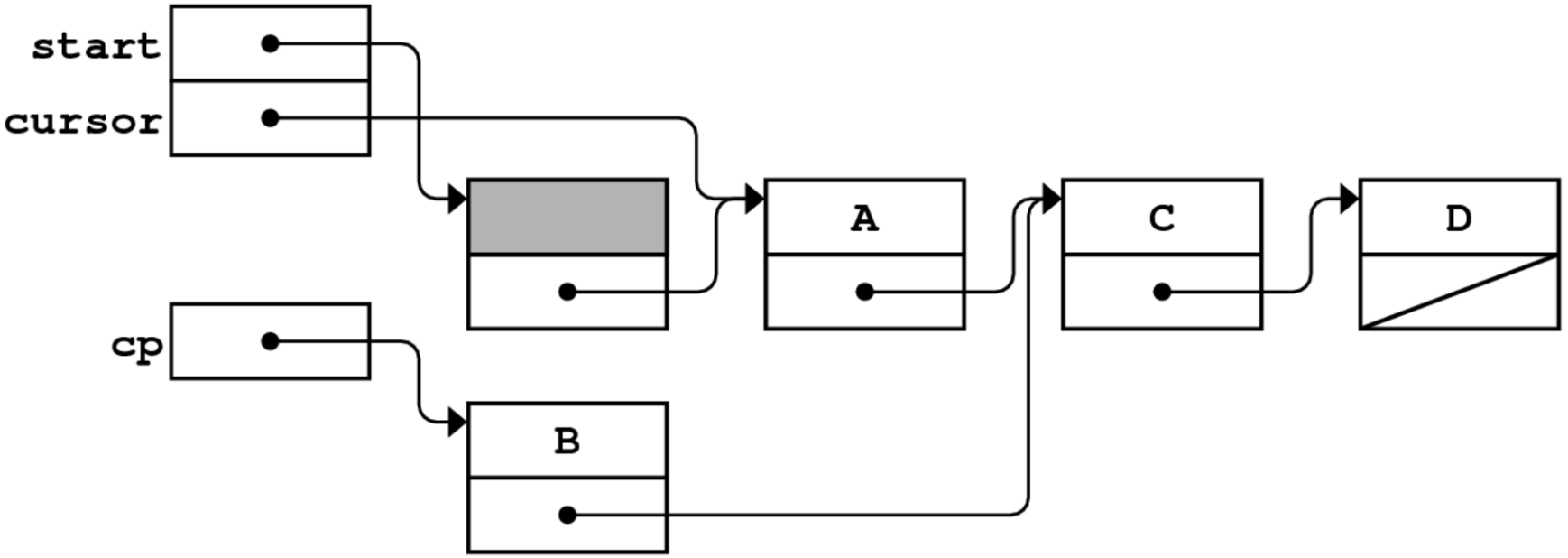



```
void EditorBuffer::insertCharacter(char ch) {
    Cell *cp = new Cell;
    cp->ch = ch;
    cp->link = cursor->link;
    cursor->link = cp;
    cursor = cp;
}
```

Insert Character

③ Copy the `link` field from the cell whose address appears in the `cursor` field into the `link` field of the new cell.

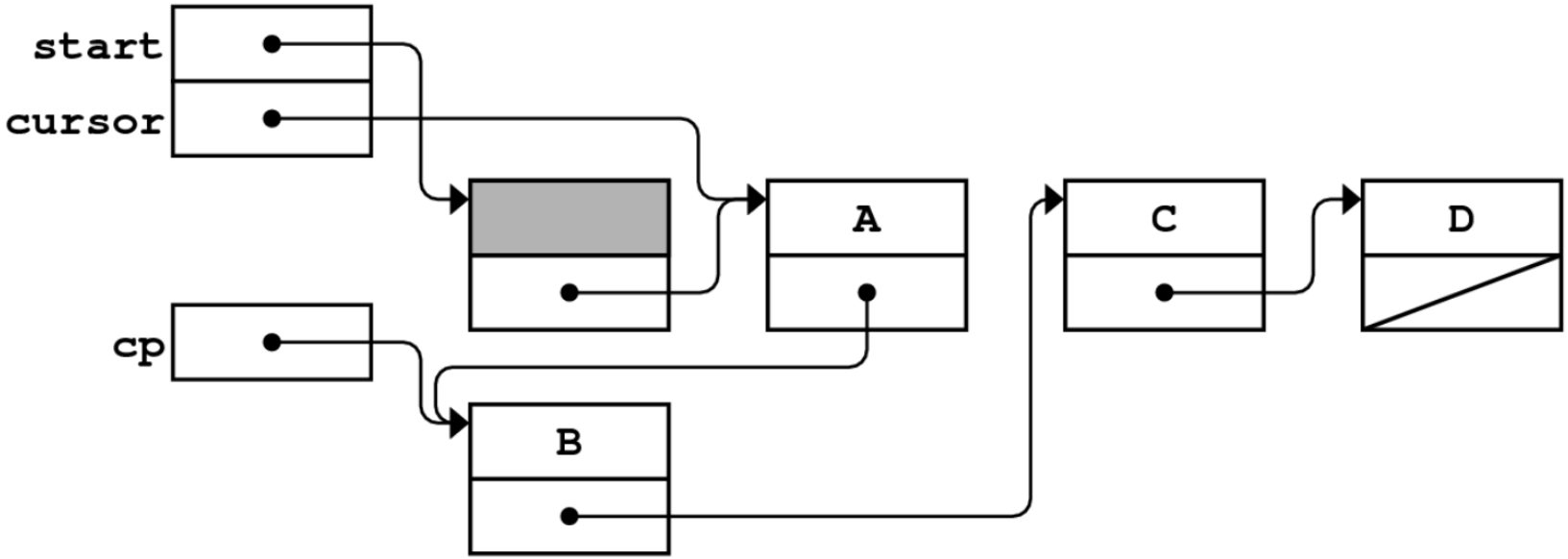
- That `link` field points to the cell containing `C`.




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

Insert Character

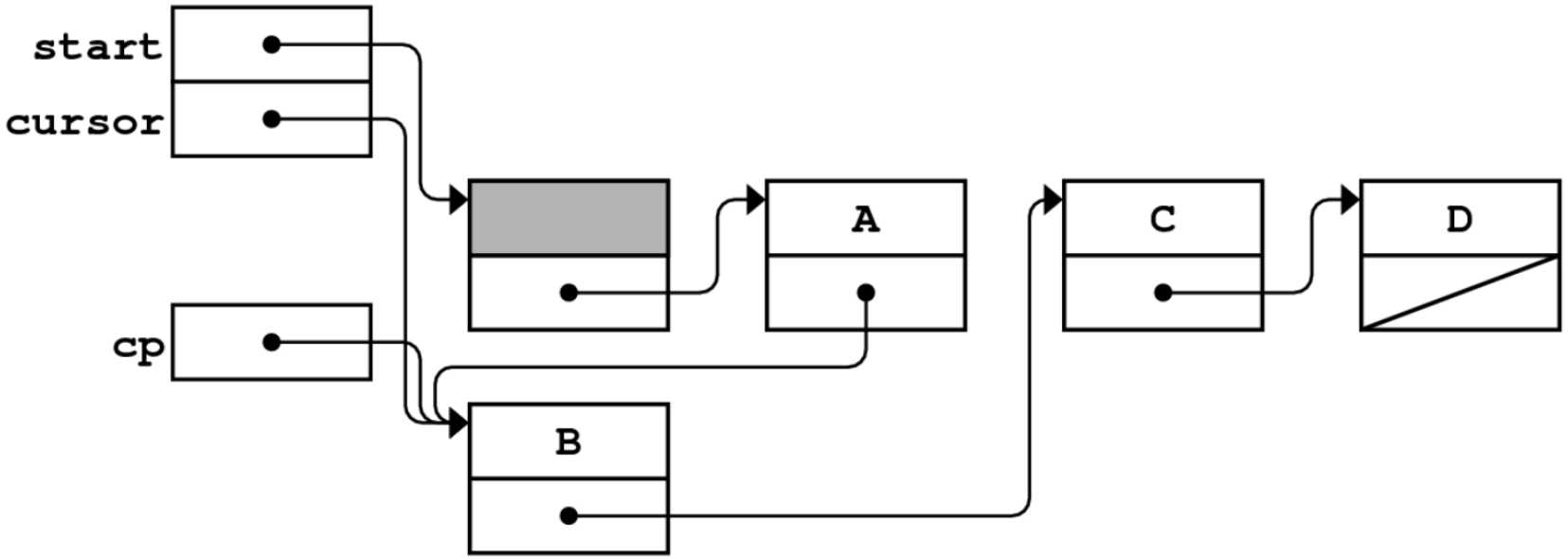
④ Change the `link` field in the current cell addressed by the `cursor` so that it points to the newly allocated cell




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

Insert Character

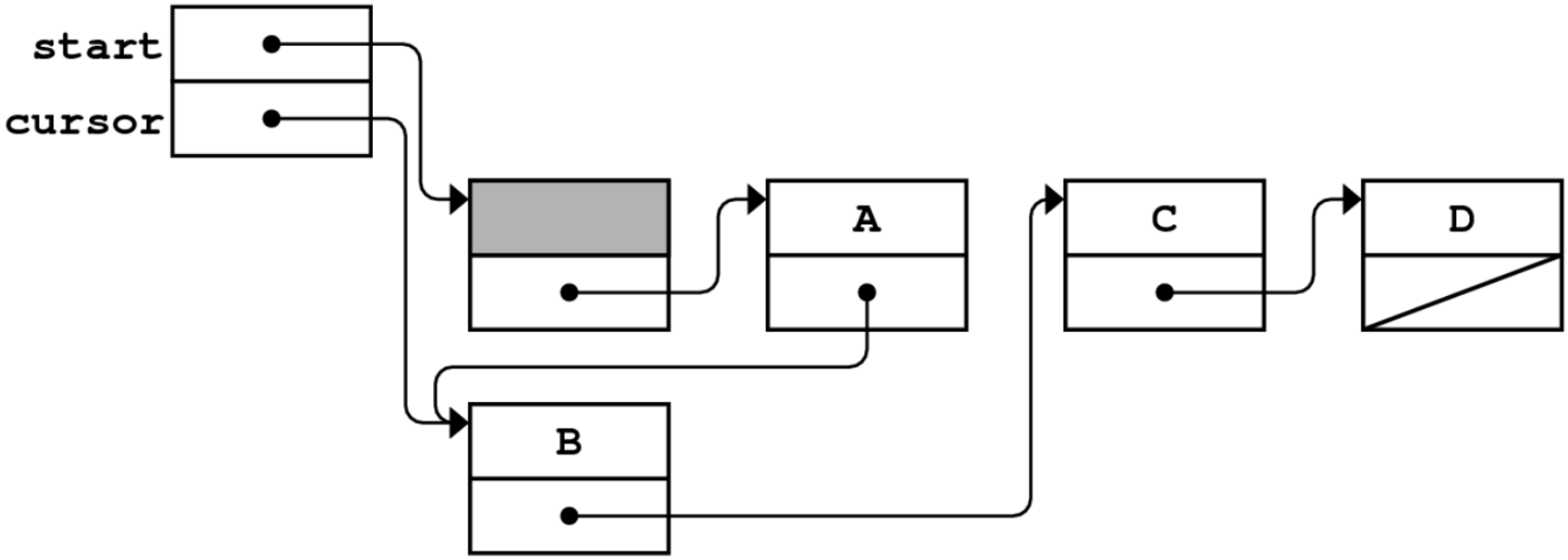
⑤ Change the `cursor` field in the buffer structure so that it also points to the new cell




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

Insert Character

- When the program returns from the `insertCharacter` method, the temporary variable `cp` is released.



- Now, buffer contents becomes

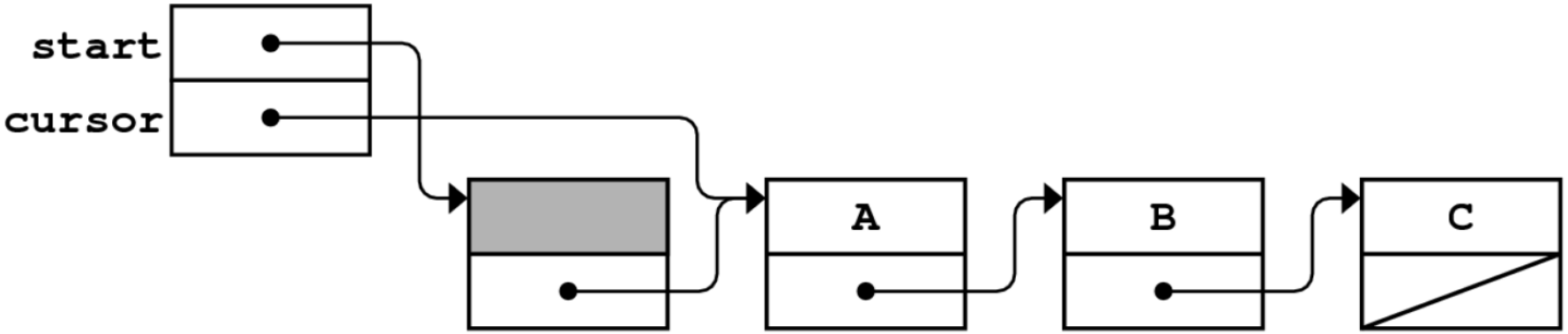
A B C D
^

Delete Character

- Delete the letter `B` in the buffer

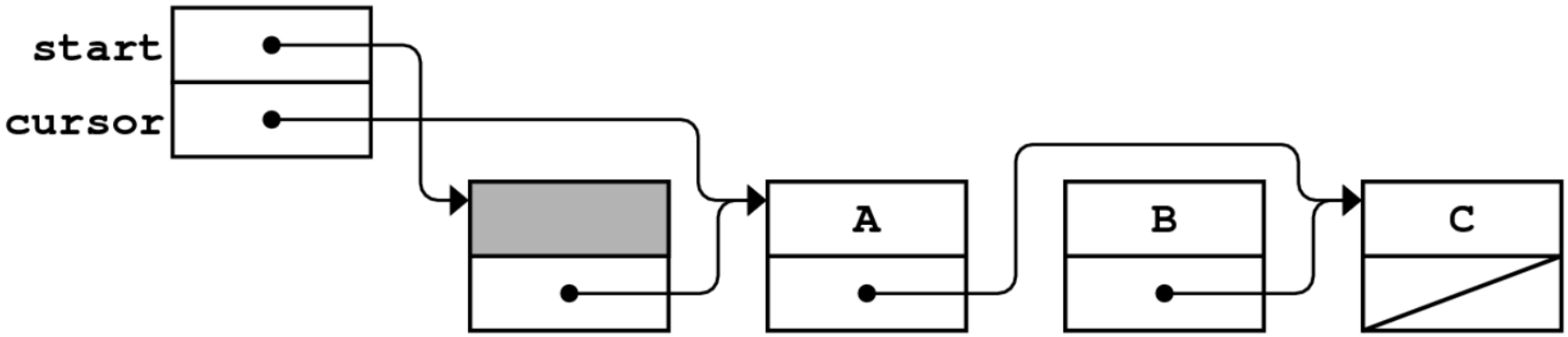
```
A B C
  ^
```

- The cursor is between the `A` and the `B` as shown.
- The situation prior to the insertion looks like this:



Delete Character

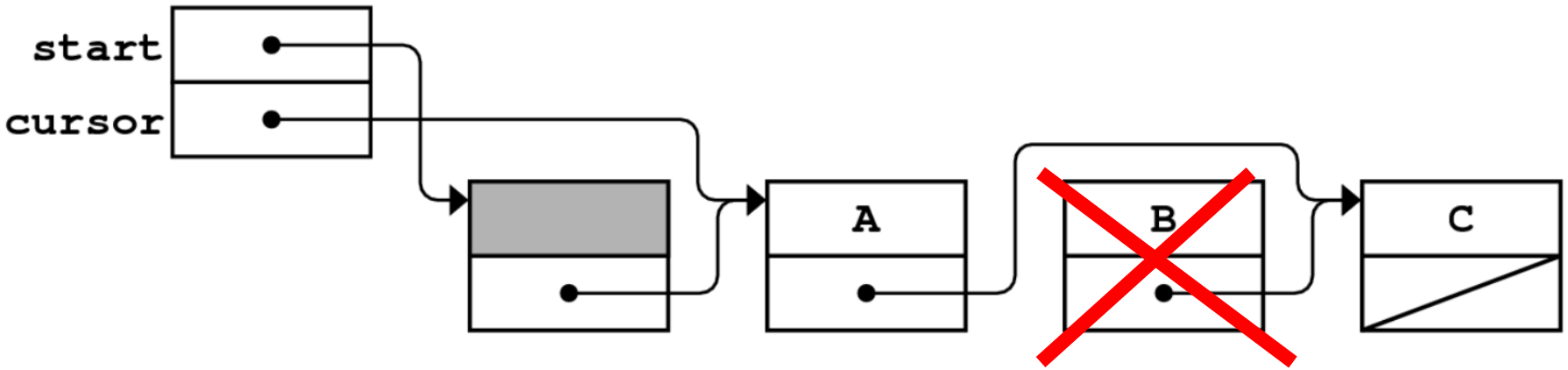
① Change the `link` field of the cell containing `A` so that it points to the next character further on




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

Delete Character

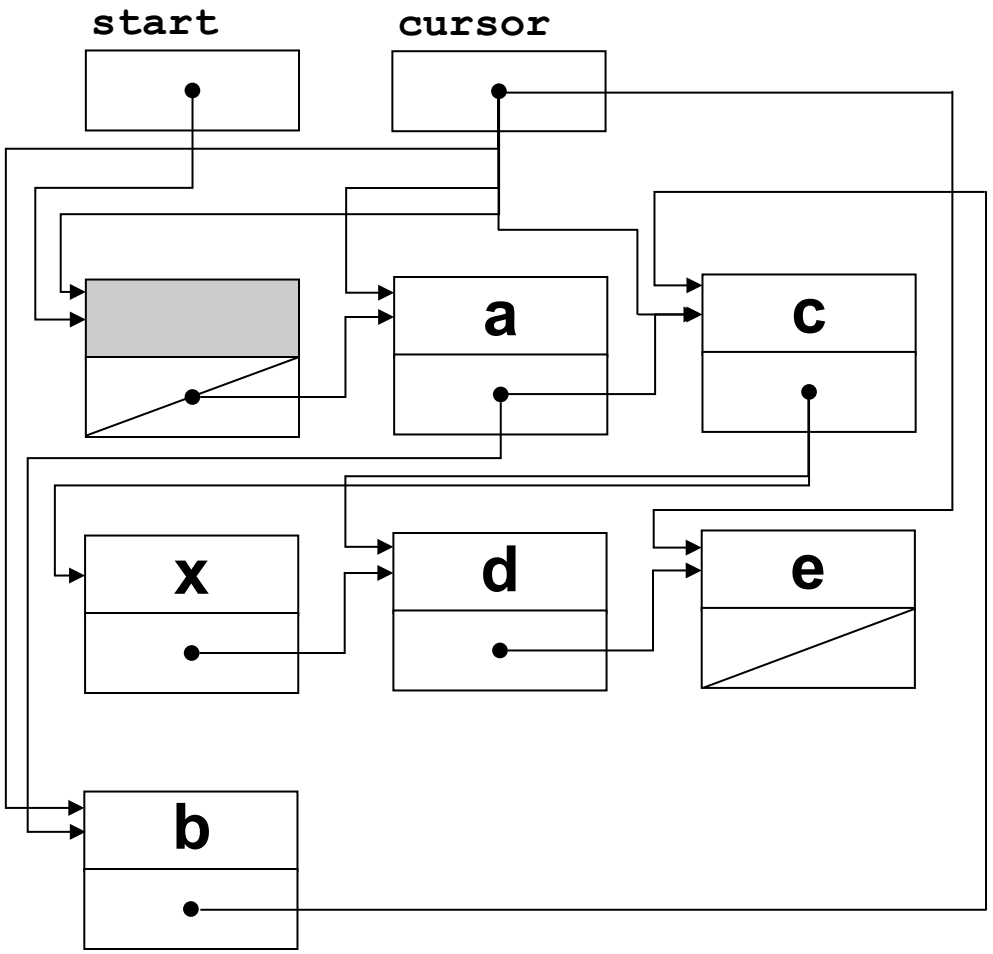
② Free the cell containing `B`.




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

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
      ^
*
```



Any question?