

# Lists, Queues, Stacks, Dequeues

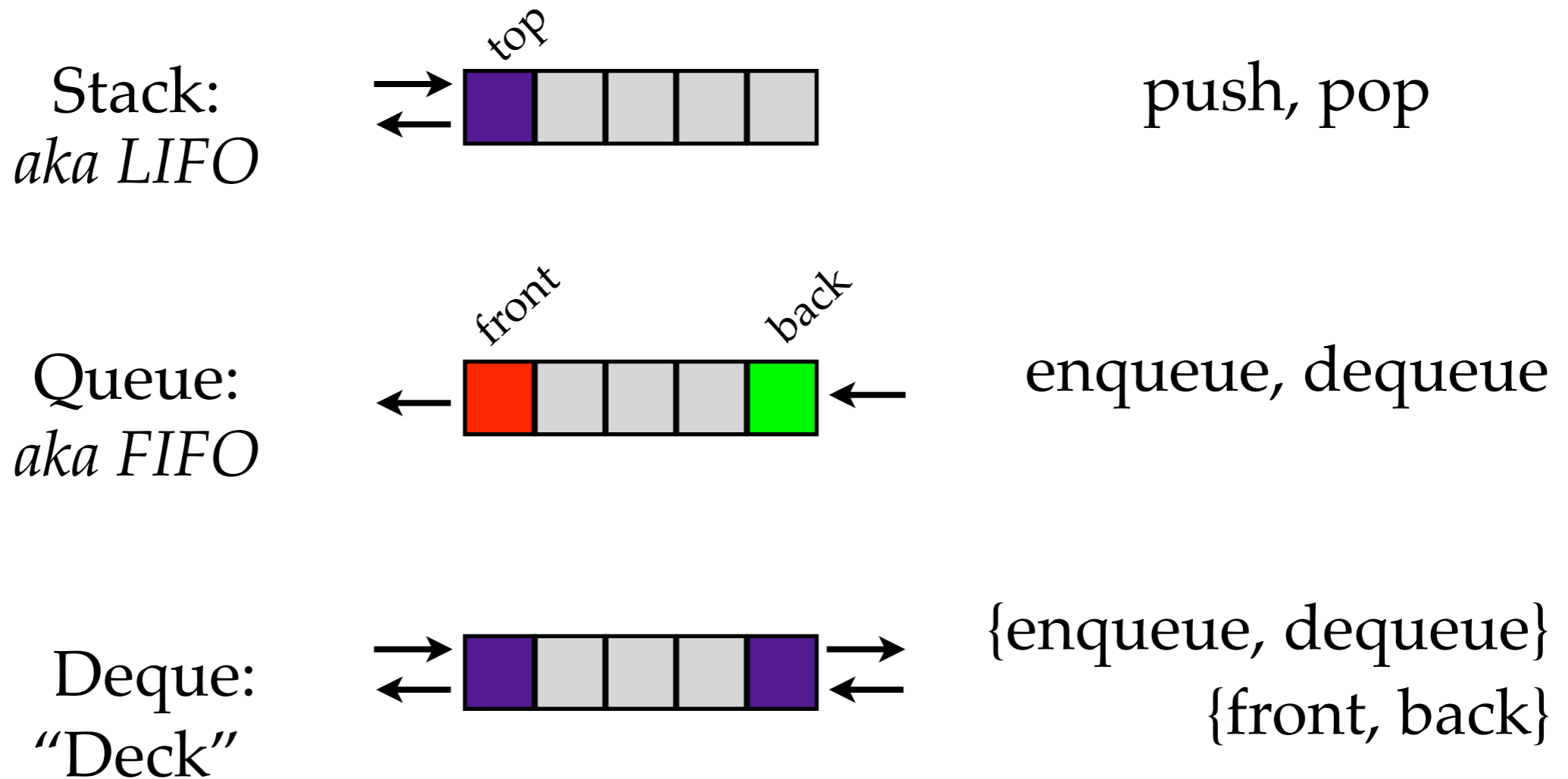
Lecture 2: CMSC 420

# General List ADT

- The simplest (and often most useful) data structure is a linear list: only care about linear ordering of items.
- Supports the following operations:
  - *get(L, i)* – return the  $i^{\text{th}}$  item in  $L$
  - *insert(L, k, i)* – insert  $k$  before item at index  $i$
  - *delete(L, i)* – delete item at index  $i$
  - *length(L)* – return number of item in  $L$
  - *split(L, k)* – split  $L$  into 3 lists: items before  $k$ ,  $\{k\}$ , items after  $k$
  - *copy(L)* – return a copy of list  $L$
  - *find(L, k)* – find item with key  $k$
  - *sort(L)* – sort list  $L$
- Implementation will depend on which subset of these operations are needed.

# Stacks, Queues, Deques

Special cases of List ADT: restrict *insert*, *delete*, and *get* to a subset of the ends of the list:



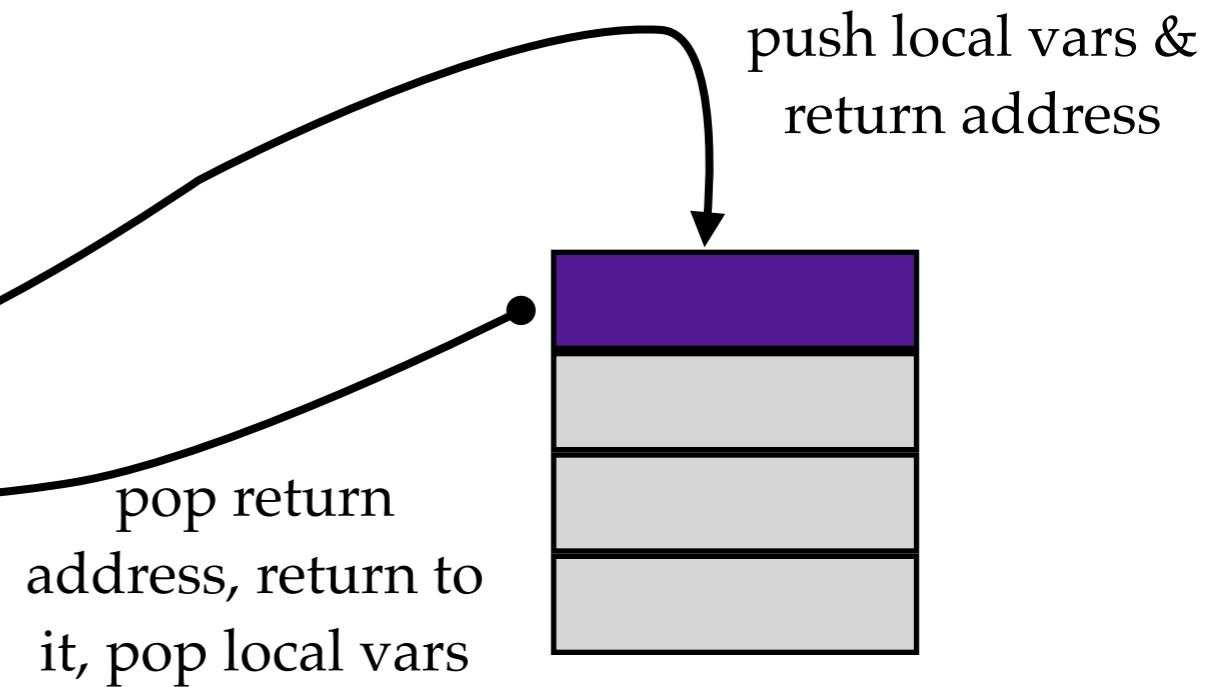
*input-restricted deque*: enqueue only one end  
*output-restricted deque*: dequeue only one end

# General List ADT

- Stacks useful to save subproblems to solve later.
- Queues for processing events.
- Deques for more general needs (e.g. undo adding an event to a queue).

# Stacks & Recursion

```
int treeHeight(Node * T) {  
    if(!T) return 0;  
    return 1 + max(  
        treeHeight(T->left),  
        treeHeight(T->right)  
    );  
}
```



Thus, can eliminate recursion by replacing it with an explicitly managed stack.

Another common example:

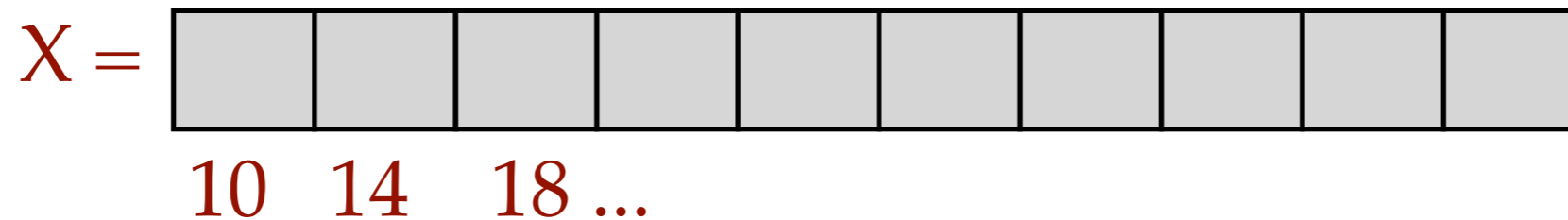
```
void drawWindow(W) {  
    saveGraphicsState();  
    // call complicated drawing routines  
    restoreGraphicsState();  
}
```



As long as save and restore calls are balanced, will work out.

# Sequential Allocation

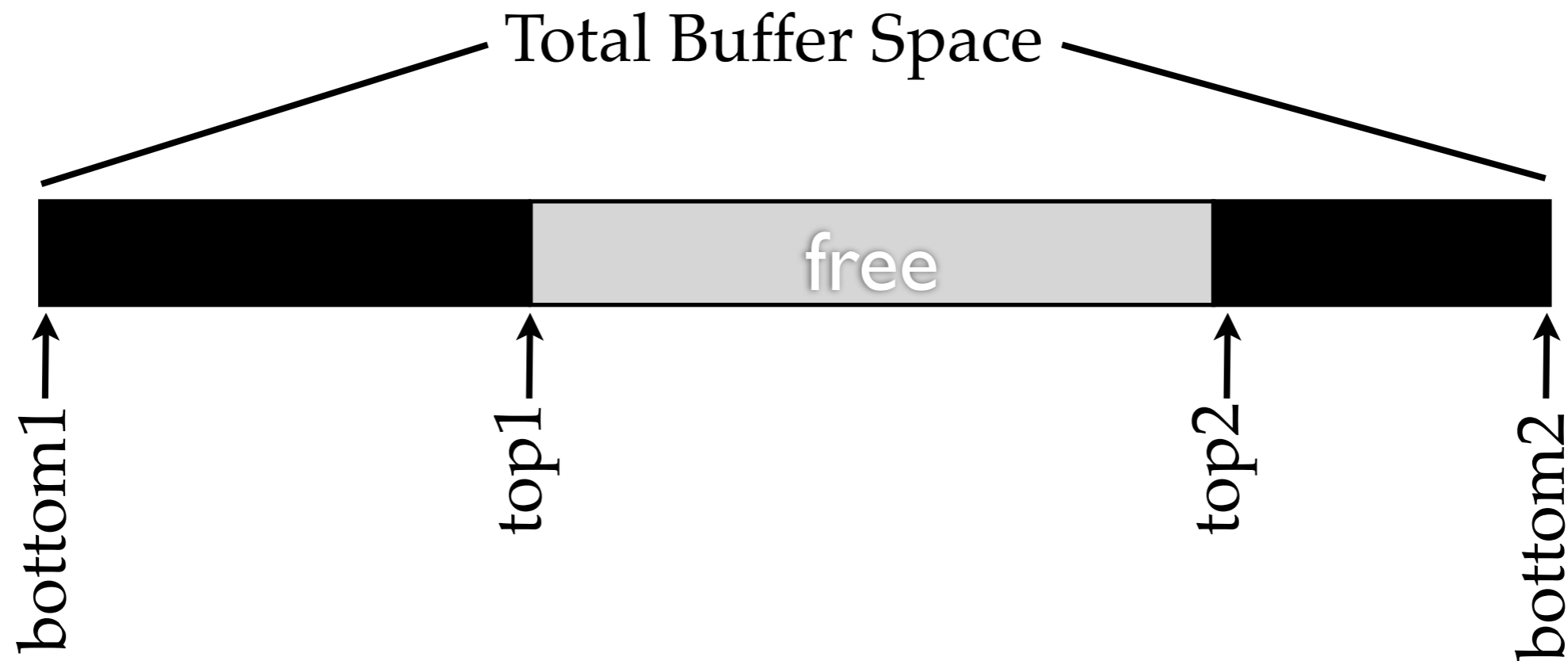
```
int X[n];
```



$X[1], X[2], X[3], \dots, X[n]$

- “**int X[n]**” allocates a block **sizeof(int)\*n** bytes.
- Logically,  $X[0]$  is the first item,  $X[1]$  the second,  $X[n-1]$  is the last.
- Address of  $i^{\text{th}}$  item:  $\text{ADDR}(X[i]) = X + i * \text{sizeof}(\text{int})$
- Natural way to implement stacks, queues, etc., especially if their max size is known in advance.

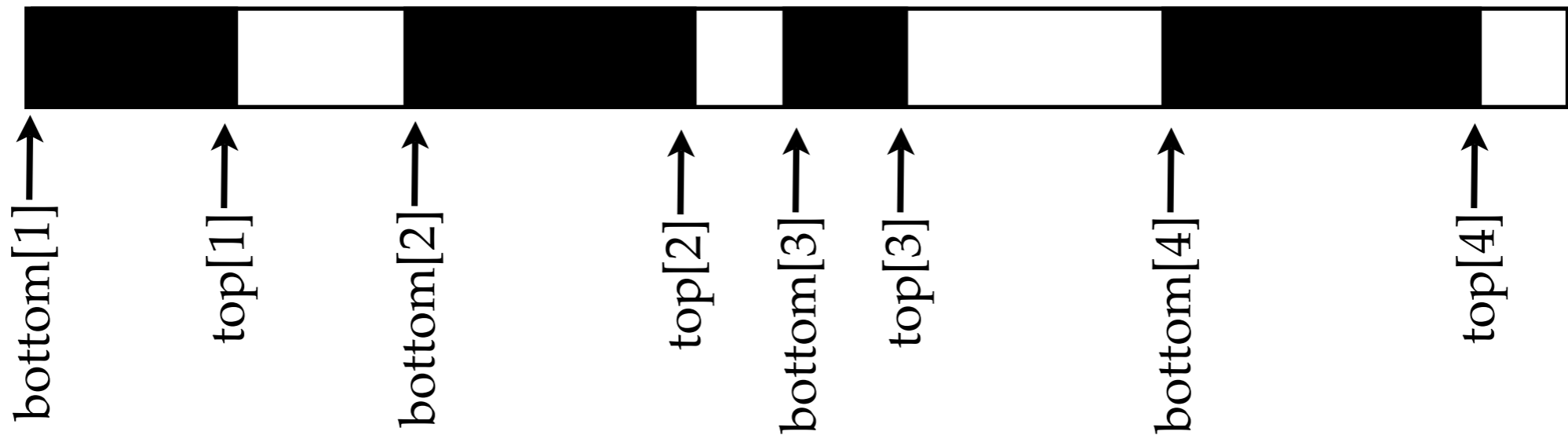
# Sequential Allocation – Two Stacks



- Properties:

- Stacks don't "move"
- OVERFLOW occurs only when sum of sizes of stacks exceeds buffer space.

## Sequential Allocation – More Than Two Stacks

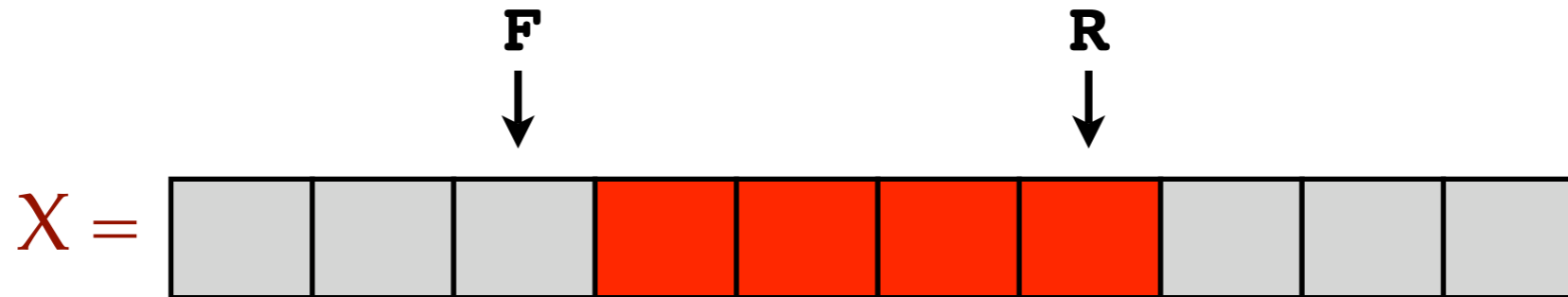


- If  $top[i] == bottom[i+1]$ , there's no room to push anything onto stack  $i$ .
- In this case, check each  $j > i$  to find smallest  $j$  such that  $top[j] < bottom[j+1]$ . If such a  $j$  exists, shift everything between  $bottom[i+1]$  to  $top[j]$  up by 1.
- If no such  $j$  exists, find gap between stacks numbered  $< i$ .



# Sequential Allocation – Ring Buffer

- Natural implementation of queue in sequential buffer:



**push:**  $X[++R] = k;$

**pop:**  $k = X[++F];$  if ( $R==F$ )  $R=F=0;$

- What if  $R > F$  always? Pretend  $X$  is circular:

**push:**

$R = (R+1) \% n;$

$X[R] = k;$

**pop:**

$F = (F+1) \% n;$

$X[F] = k;$

if ( $R==F$ )  $R=F=n;$

# Sequential Allocation – Ring Buffer

- Why isn't this "creeping" a problem in a bank line?

# Sequential Searching

$X[1], X[2], X[3], \dots, X[n]$

- Basic sequential searching:

```
for(i = 0; i < n; i++)  
    if(X[i] == K) break;
```

- Expected cost of...

- **success** =  $2n/2$  comparisons
- **failure** =  $2n$  comparisons

- Remove factor of 2 by using a **sentinel**:

```
X[n+1] = K  
for(i = 0; ; i++)  
    if(X[i] == K) break;
```

- Must be able to cheaply add item to end of list.

## Why does success take $n/2$ comparisons?

$X[1], X[2], X[3], \dots, X[n]$

- Assume we're equally likely to ask for any key.
  - Probability we ask for  $i^{\text{th}}$  key is  $1/n$ .
  - Expected number of comparisons is

$$\sum_{i=1}^n i(1/n) = \frac{1}{n} \sum_{i=1}^n i = \frac{1}{n} \left( \frac{n(n+1)}{2} \right) = (n+1)/2$$

# Sorted Sequential Searching

- Can stop earlier if list is in sorted order:

```
for(i = 0; i < n; i++)  
    if(X[i] >= K) break;  
if(X[i] != K) return FAIL;
```

- Expected cost of...
  - **success** =  $2n/2$  comparisons
  - **failure** =  $2n/2$  comparisons
- Why do this instead of binary search?
  - May not have random access into list
  - Exploits locality

## Use access probabilities to order information

- Put most accessed information first, least used last.
- Violates the idea of worst case analysis.
- But often very useful in practice.

# Use access probabilities to order information

items:  $x_1$   $x_2$   $x_3$   $x_4$  ...  
probability of access:  $p_1$   $p_2$   $p_3$   $p_4$  ...

- Expected # of comparisons over many searches:

$$C_n = 1p_1 + 2p_2 + 3p_3 + \dots + np_n$$

- But don't usually know the  $p_i$

# Self-organizing Lists

- Heuristics to improve repeated access to same elements.
- **Move to Front:**
  - ✳ *When an item is accessed, move it to the front of the list.*
  - Expected number of comparisons is  $< 2$  times the optimal ordering.
  - Rearranges list quickly.
  - A rare, low-probability access effects many subsequent searches.
- **Transpose:**
  - ✳ *When item  $i$  is accessed,  $\text{swap}(i-1, i)$ .*
  - Expected number of comparisons less than Move-to-Front.
  - But adapts slower.



# Self-organizing Lists

- Probably not a good as the search trees will discuss in a few classes, but
  - Simple code change can lead to good speed up.
  - Fast insert time, so if lookups are rare relative to inserts, might be faster.
  - Low space overhead.

# Sets

- A set  $S$  is a collection of objects from some universe  $U$ .
- Only interested in whether an object  $u$  is in  $S$ .
- Item operations:
  - $insert(S, u)$
  - $delete(S, u)$
  - $is\_member(S, u)$
- Operations on sets:
  - $S = S_1$  **intersect**  $S_2$ :  $u$  in  $S$  iff  $u$  in  $S_1$  and  $S_2$
  - $S = S_1$  **union**  $S_2$ :  $u$  in  $S$  if  $u$  in  $S_1$  or  $u$  in  $S_2$
  - $S = S_1 - S_2$  (sometimes written  $S_1 \setminus S_2$ ):  $u$  in  $S$  if  $u$  in  $S_1$  but not in  $S_2$
  - $S = S_1$  **symmetric difference**  $S_2 = u$  in  $S$  if  $u$  in  $S_1$  but not  $S_2$  or  $u$  in  $S_2$  but not  $S_1$ .

# Sets – Example application

The House of Representatives shall be composed of Members chosen every second Year by the People of the several States, and the Electors in each State shall have the Qualifications requisite for Electors of the most numerous Branch of the State Legislature.

No Person shall be a Representative who shall not have attained to the Age of twenty five Years, and been seven Years a Citizen of the United States, and who shall not, when elected, be an Inhabitant of that State in which he shall be chosen.

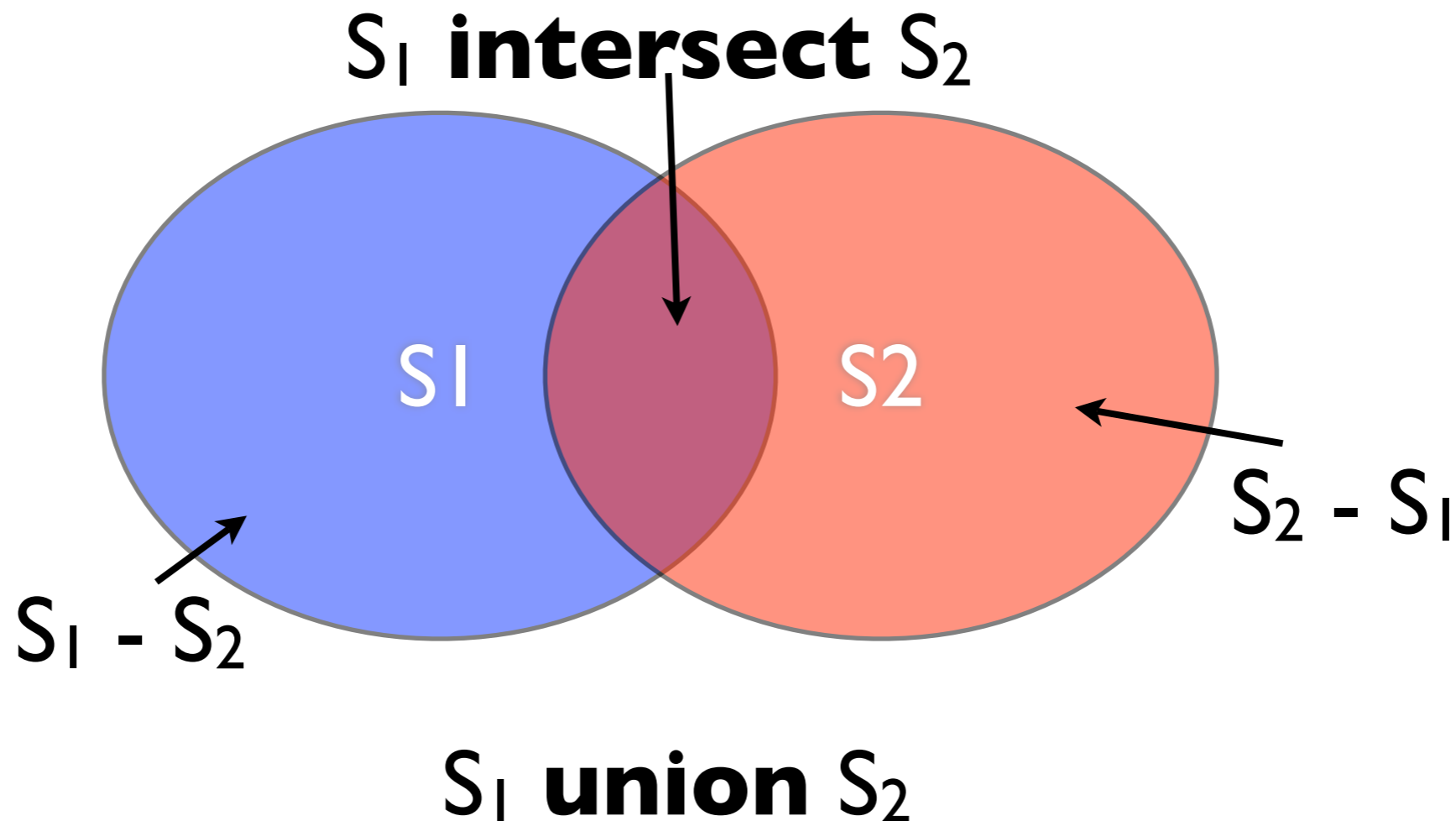
Universe  $U = \{\text{English words}\}$

|                 |   |
|-----------------|---|
| house           | 1 |
| representatives | 1 |
| people          | 1 |
| democracy       | 0 |
| dictator        | 0 |
| privacy         | 0 |
| electors        | 1 |
| ...             |   |

Compare documents based on sets of words they contain.

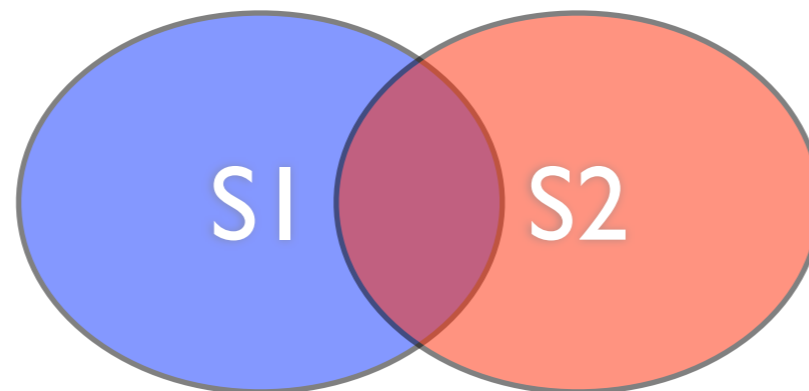
# Symmetric Difference

- $S = S_1$  symmetric difference  $S_2 = u$  in  $S$  if  $u$  in  $S_1$  but not  $S_2$  or  $u$  in  $S_2$  but not  $S_1$ .
- Symmetric difference
  - =  $(A - B)$  union  $(B - A)$
  - =  $(A \text{ union } B) - (A \text{ intersect } B)$



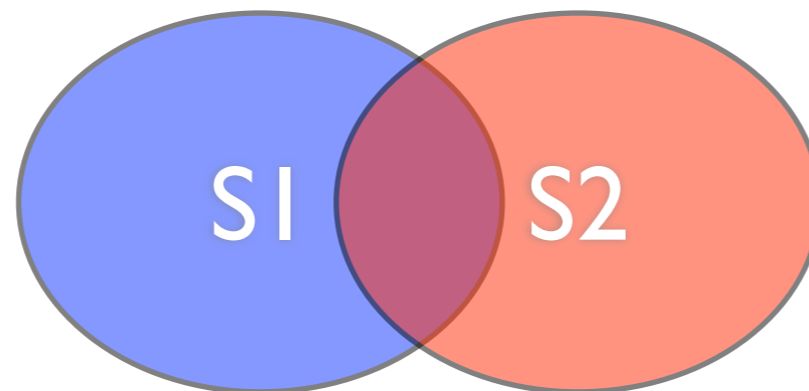
# Bit Vectors

- If universe  $U$  is small enough, can represent a set by a **bit vector**, with one dimension for each possible element in  $U$ .
  - $s1 = 001001010101001001$
  - $s2 = 000101010000111111$
- Intersection = bitwise AND ( $s1 \& s2$  in C/C++)
- Union = bitwise OR ( $s1 | s2$  in C/C++)
- Complement = bitwise NEGATION ( $\sim s1$  in C/C++)
- Set difference ( $s1 - s2$ )???
- Symmetric difference???



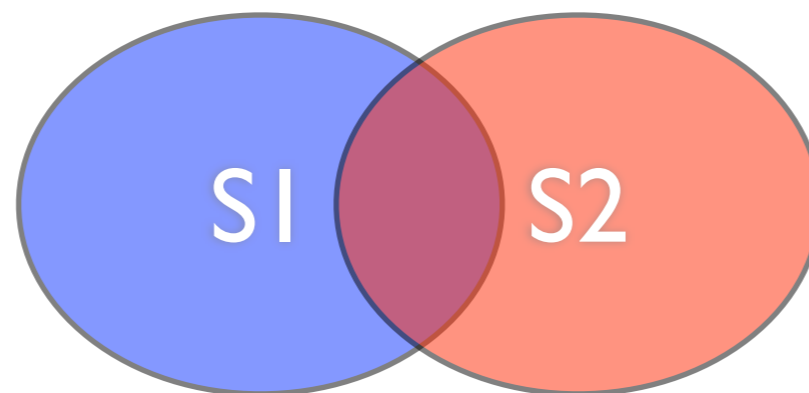
# Bit Vectors

- If universe  $U$  is small enough, can represent a set by a **bit vector**, with one dimension for each possible element in  $U$ .
  - $s1 = 001001010101001001$
  - $s2 = 000101010000111111$
- Intersection = bitwise AND ( $s1 \& s2$  in C/C++)
- Union = bitwise OR ( $s1 | s2$  in C/C++)
- Complement = bitwise NEGATION ( $\sim s1$  in C/C++)
- Set difference ( $s1 - s2$ )???  **$A \& \sim B$**
- Symmetric difference???



# Bit Vectors

- If universe  $U$  is small enough, can represent a set by a **bit vector**, with one dimension for each possible element in  $U$ .
  - $s1 = 001001010101001001$
  - $s2 = 000101010000111111$
- Intersection = bitwise AND ( $s1 \& s2$  in C/C++)
- Union = bitwise OR ( $s1 | s2$  in C/C++)
- Complement = bitwise NEGATION ( $\sim s1$  in C/C++)
- Set difference ( $s1 - s2$ )???  **$A \& \sim B$**
- Symmetric difference???  **$A \wedge B$**

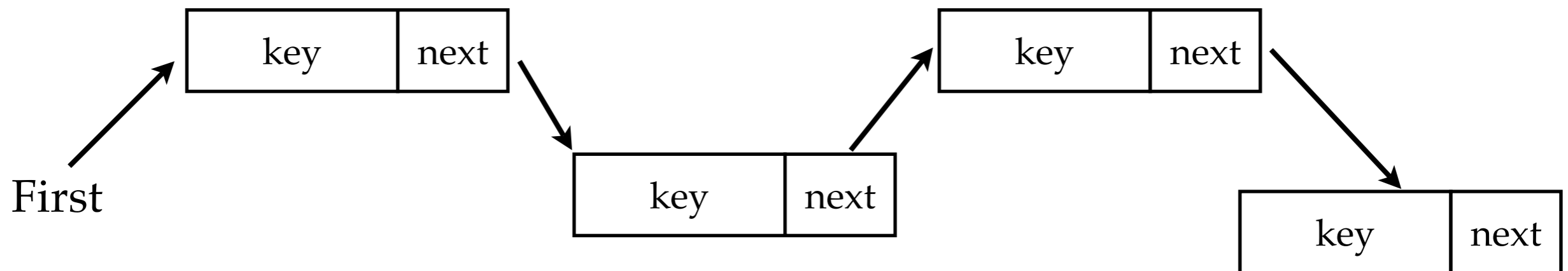


# Linked Allocation

- Records need not be located in adjacent memory.
- Each node has at least 2 fields:



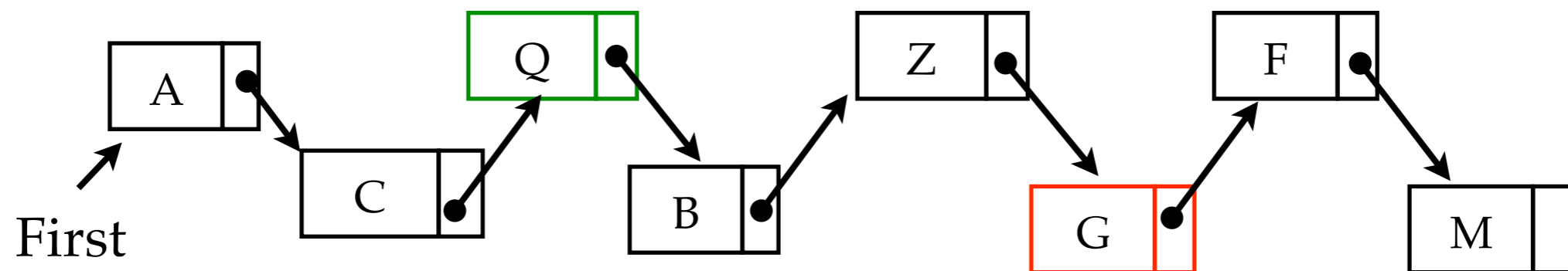
- Need to keep a pointer to the first node:





# Zig-zag scan

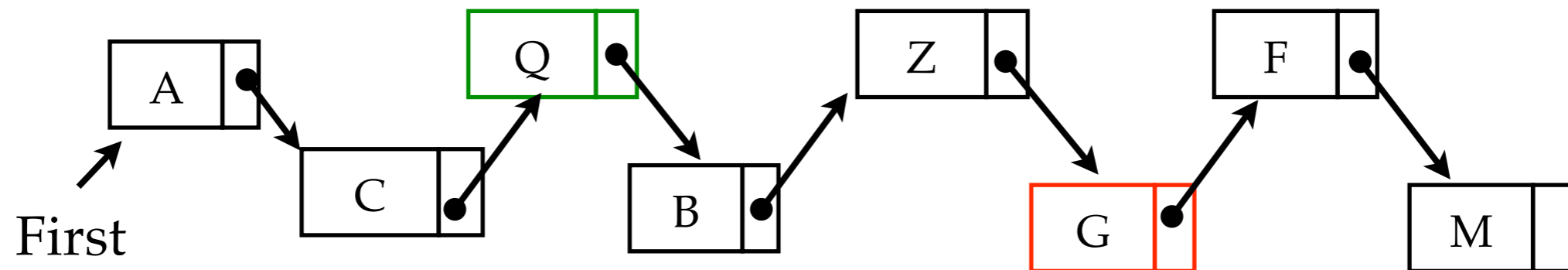
- Suppose L is a singly linked list.
- **Problem:** Find the node 3 nodes before the one containing "G".



- Any thoughts about how to solve this problem?

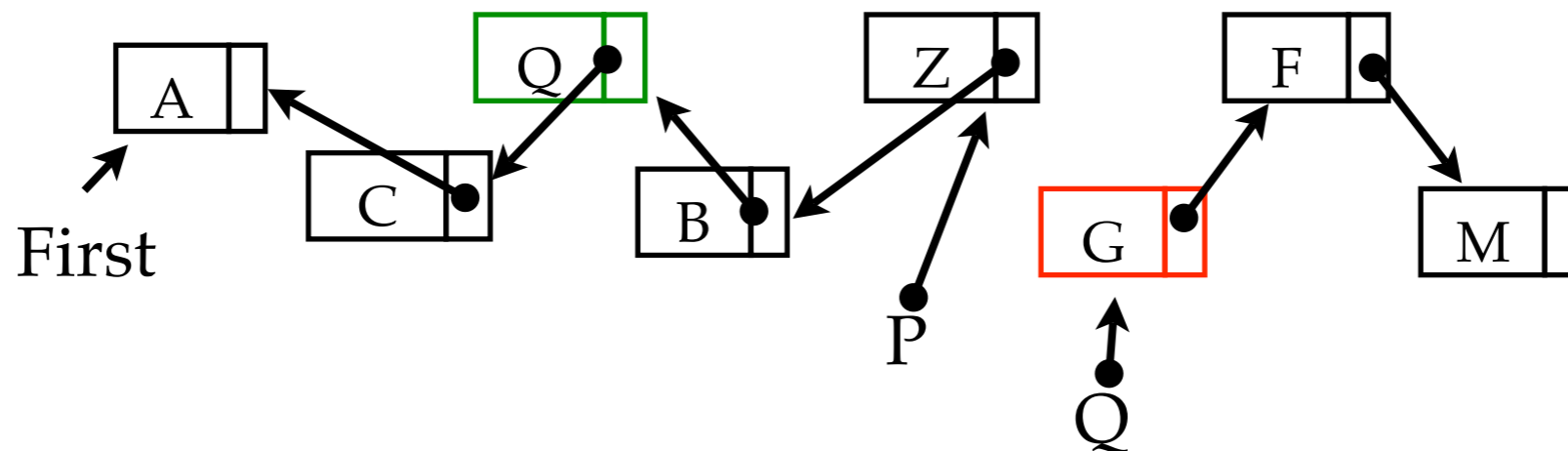
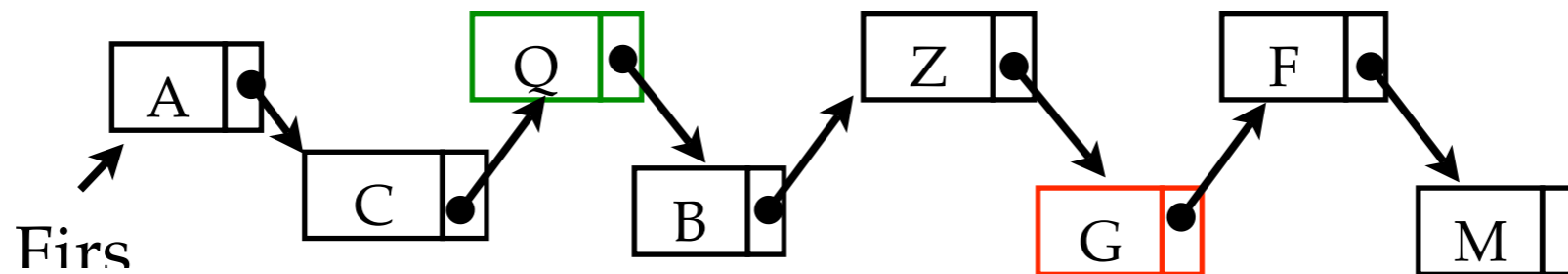
# Zig-zag scan

- Walk down list, pushing node addresses onto a stack.
- When you find "G", pop 3 addresses off the stack.
- Extra storage:  $O(1)$  per node & only used when needed.



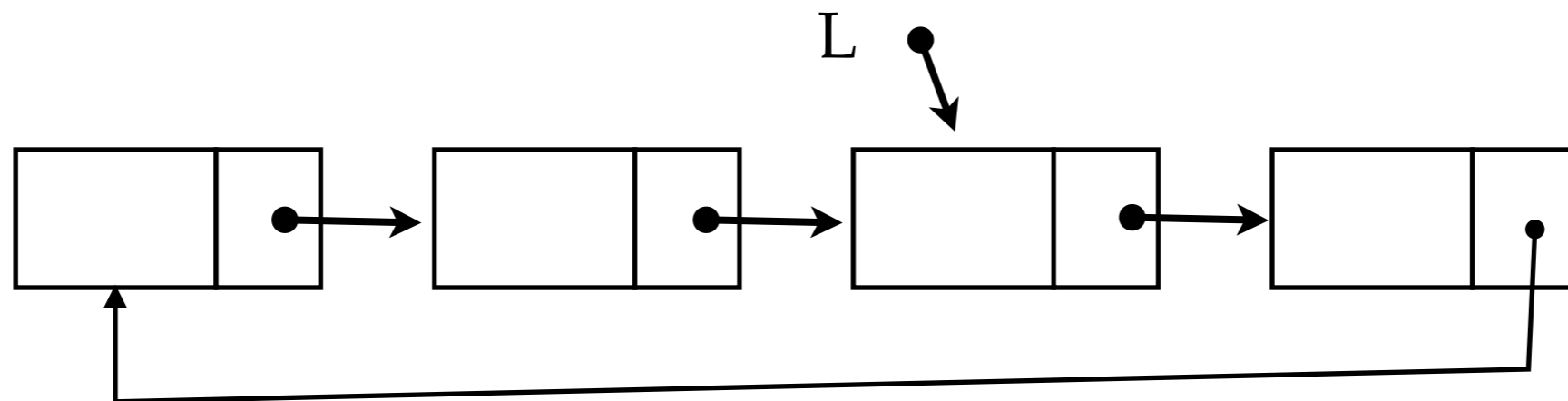
# Link inversion

- What if we want to use only  $O(1)$  space to solve that problem?
- As we walk down the list with two pointers  $P$ ,  $Q$ , we reverse the NEXT pointers to point to the previous node:



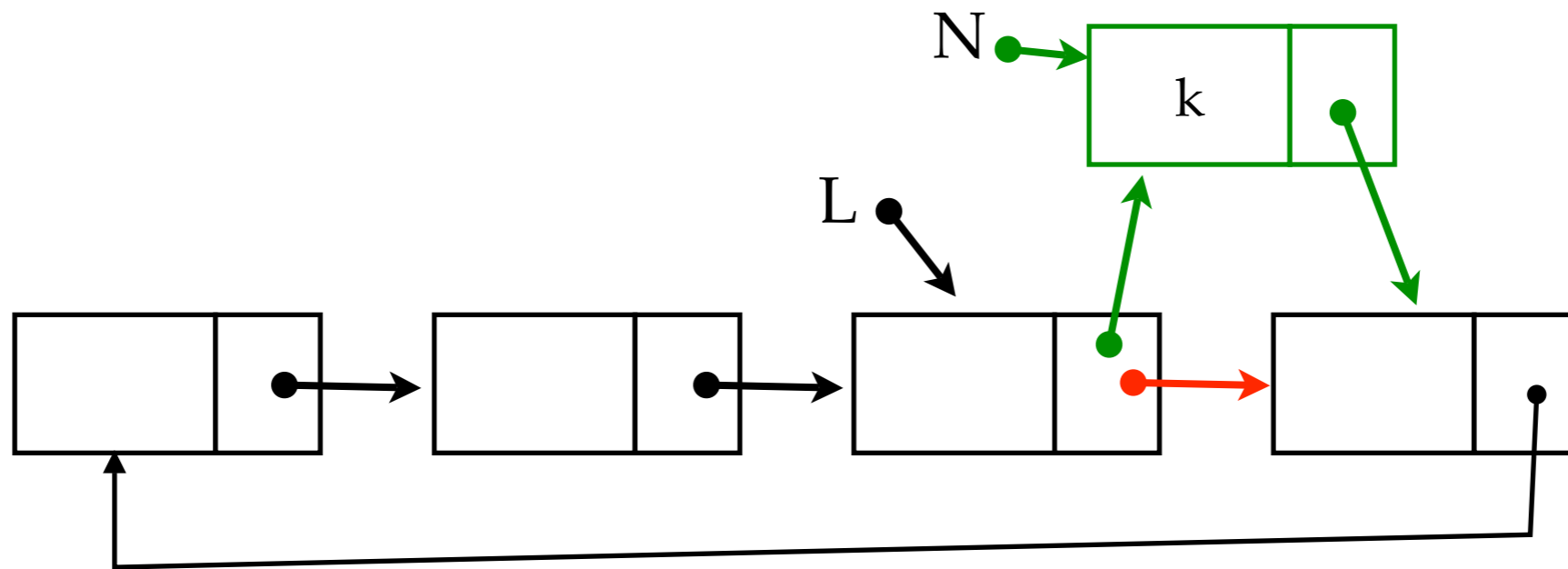
# Circularly Linked List

- “Last” node points to the “first” node.
- No longer need to think of any node as the beginning.
- Pointer to any node gives access to the whole list.



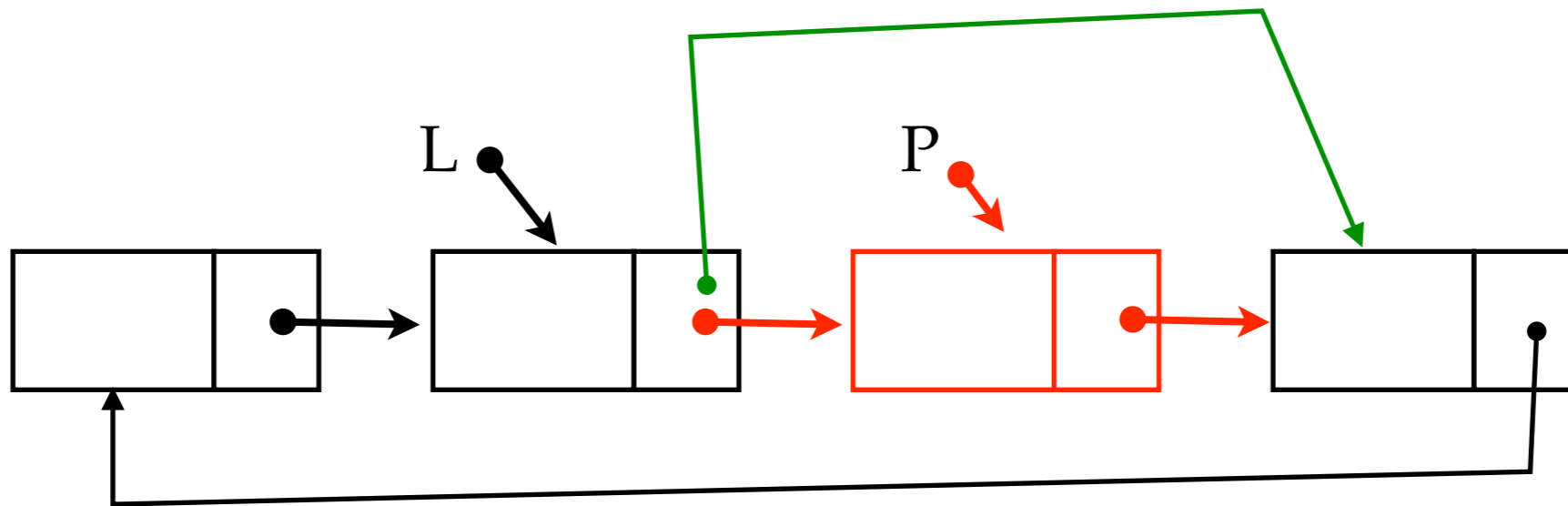
# Circularly Linked List – Insert

```
void insert(Node * L, int k) {  
    Node * N = new Node(k);    /* create new node N */  
    if(L == NULL) {  
        L = N->next = N;  
    } else {  
        N->next = L->next;  
        L->next = N;  
    }  
}
```



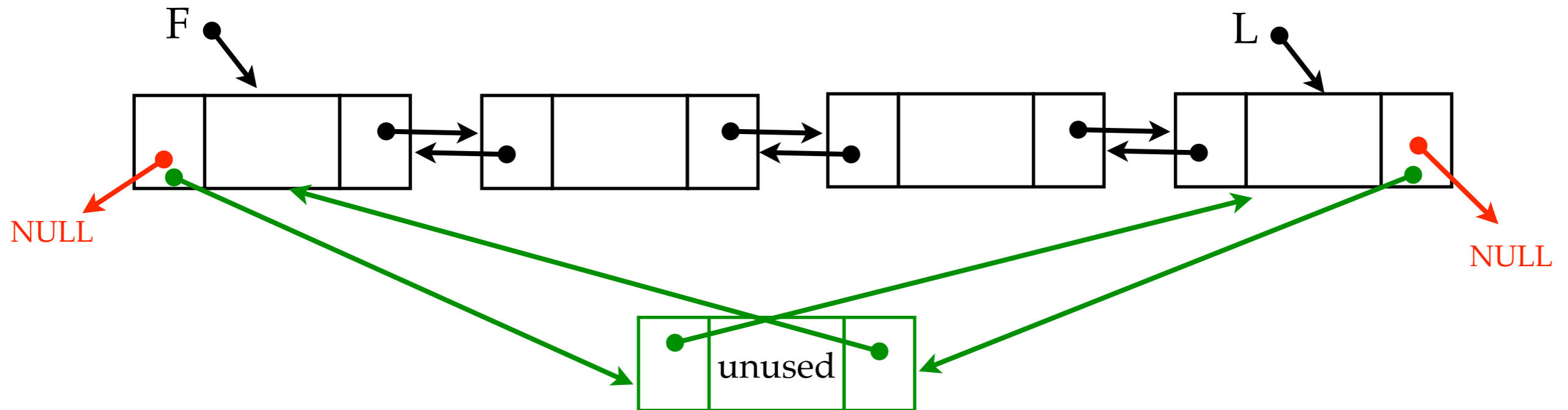
# Circularly Linked List – Delete

```
Node * delete(Node * L) {  
    if(L == NULL) {  
        /* ERROR: underflow! */  
    } else {  
        Node * P = L->next;  
        L->next = P->next;  
        if(L==P) L = NULL; /* list had only one node */  
        delete P;  
    }  
    return L;  
}
```



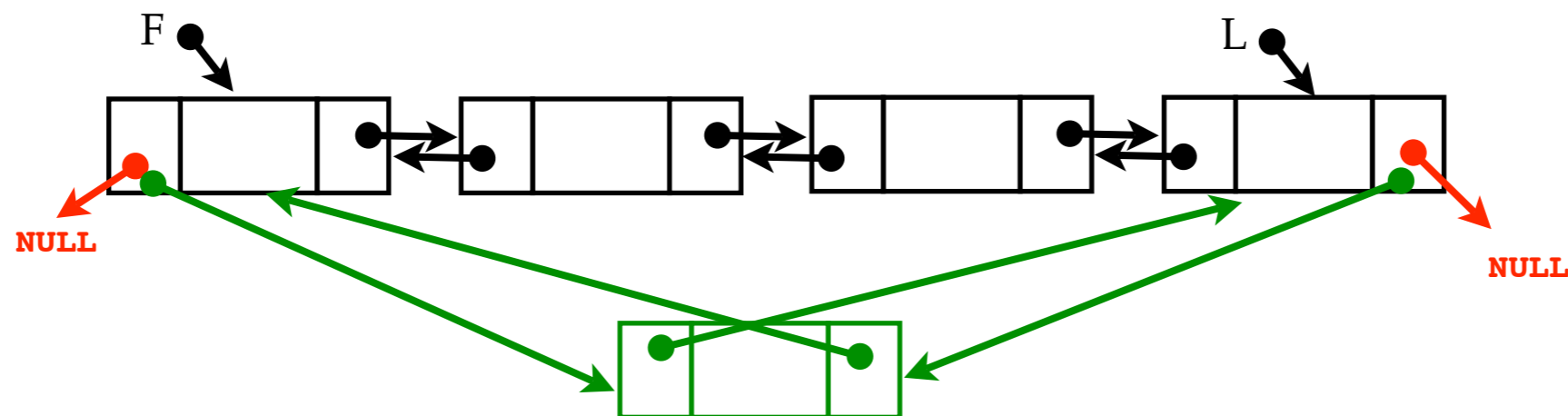
# Doubly Linked List

- Next and Prev pointers for each node.
- If F is the first node, and L is the last node, either:
  - $\text{Prev}(F) == \text{Next}(L) == \text{NULL}$ , or
  - $\text{Prev}(F) == \text{Next}(L) == \text{HEADER}$
  - With HEADER option,  $\text{Prev}(\text{Next}(X)) == \text{Next}(\text{Prev}(X)) == X$



# Doubly Linked List


- Advantages:
  - iterate forward or backward easily,
  - delete any node, given pointer to it in  $O(1)$  time
- Disadvantages:
  - Extra space needed for twice as many pointers.
  - Trick to get around this limitation: XOR





# XOR Implementation of Doubly Linked Lists

- In normal doubly linked list, each pointer value occurs twice:
  - $\text{Next}(\text{Prev}(X)) = \text{Prev}(\text{Next}(X))$
  - $2n$  pointers, but only  $n$  items...
- Recall: if  $a, b$  are 0 or 1, then  $a \text{ XOR } b = 1$  if exactly one of  $a$  or  $b$  is 1, but 0 otherwise.
- If  $(a \text{ XOR } b) = c$ , then  $(c \text{ XOR } b) = a$ :

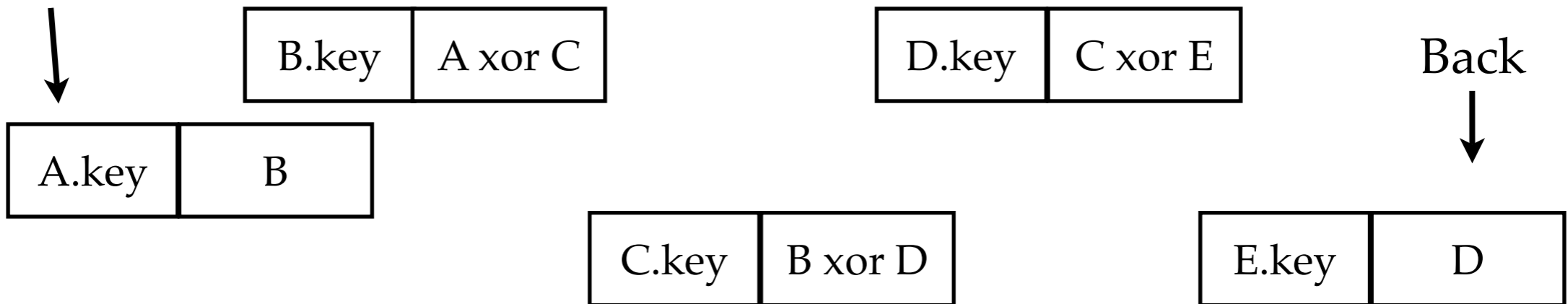
$$\begin{array}{r} \phantom{\text{XOR}} \quad 11111 \ 00000 \\ \text{XOR} \quad 01001 \ 00100 \\ \hline \phantom{\text{XOR}} \quad 10110 \ 00100 \end{array}$$


$$\begin{array}{r} \phantom{\text{XOR}} \quad 10110 \ 00100 \\ \text{XOR} \quad 01001 \ 00100 \\ \hline \phantom{\text{XOR}} \quad 11111 \ 00000 \end{array}$$

# XOR Implementation of Doubly Linked Lists

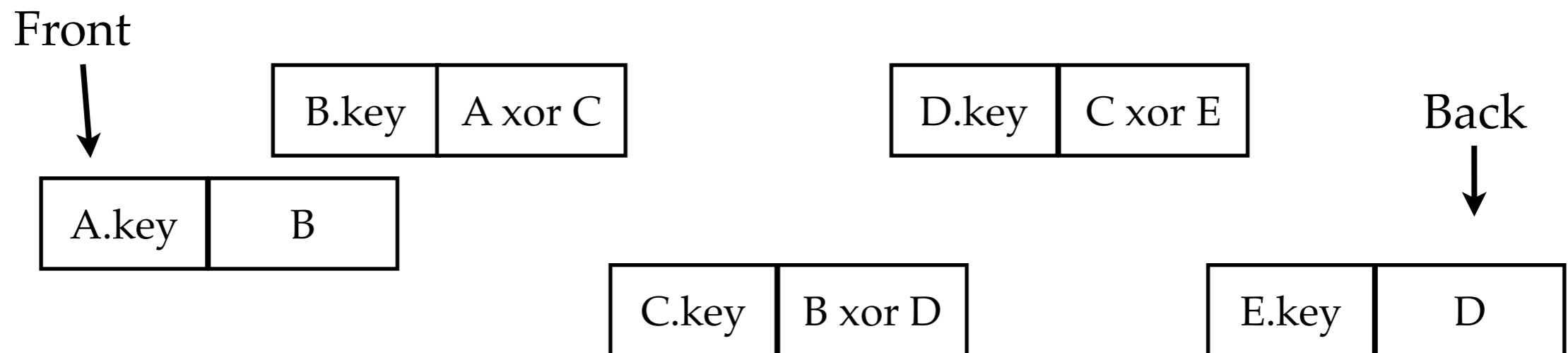
- If  $(a \text{ XOR } b) = c$ , then  $(c \text{ XOR } b) = a$ .
- ➔ Can encode a two pointers  $a, b$  as  $c = (a \text{ XOR } b)$ . You can recover  $a$  and  $b$ :
  - $a = c \text{ XOR } b$ ,
  - $b = c \text{ XOR } a$
- So, can store 2 pointers in one location.
- Use single pointer field to hold Prev XOR Next:

Front



# More on XOR Doubly Linked Lists

- Can only be used if nodes are always accessed by walking from one end to the middle.
- May be difficult to implement in languages like Java that discourage pointer arithmetic.



# Comparison of Linked vs. Sequential

- **Linked:**

- Extra storage required
- Better use of fragmented memory
- Insertion / deletion at middle is easier
- Joining lists easier
- NEXT operation requires pointer dereference

- **Sequential:**

- Next and Previous are implicit (less storage)
- Can take advantage of locality
- Random access
- NEXT operation probably faster