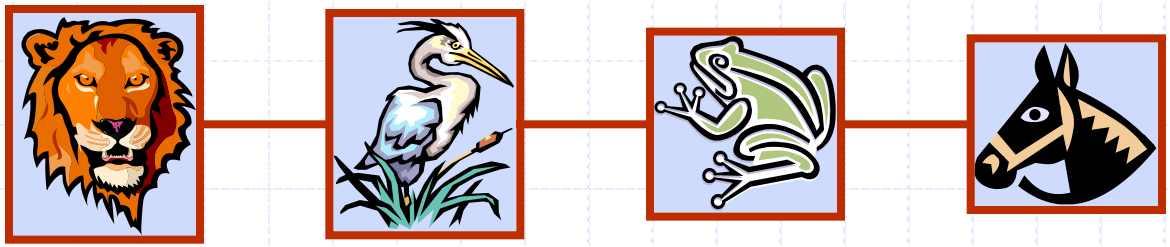
Lists and Sequences

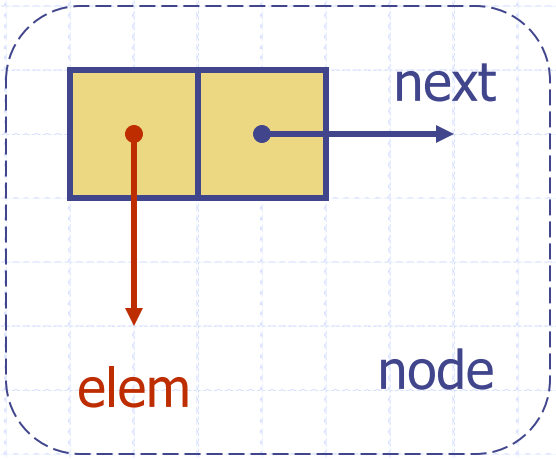


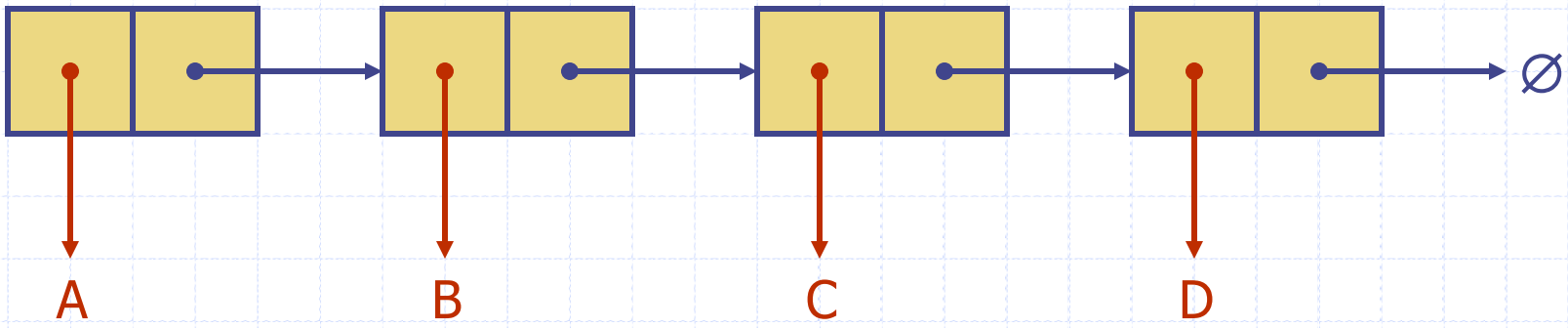
Outline

- Singly linked list
- Position ADT and List ADT
- Doubly linked list
- Sequence ADT
- Implementations of the sequence ADT
- Iterators

Singly Linked List

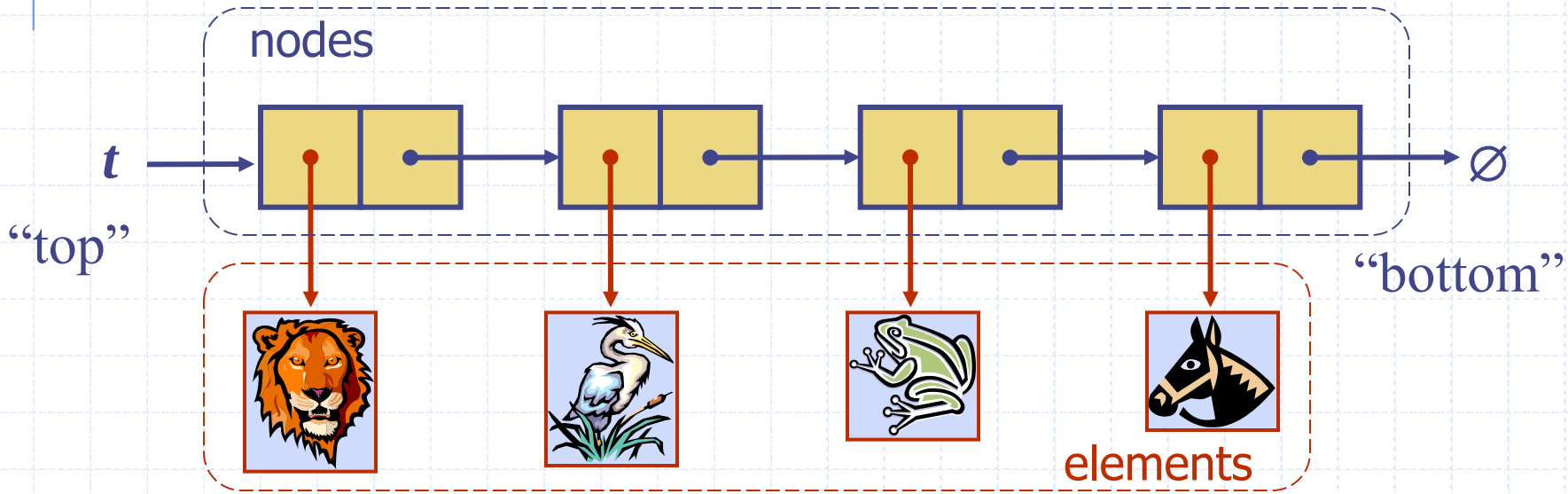
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





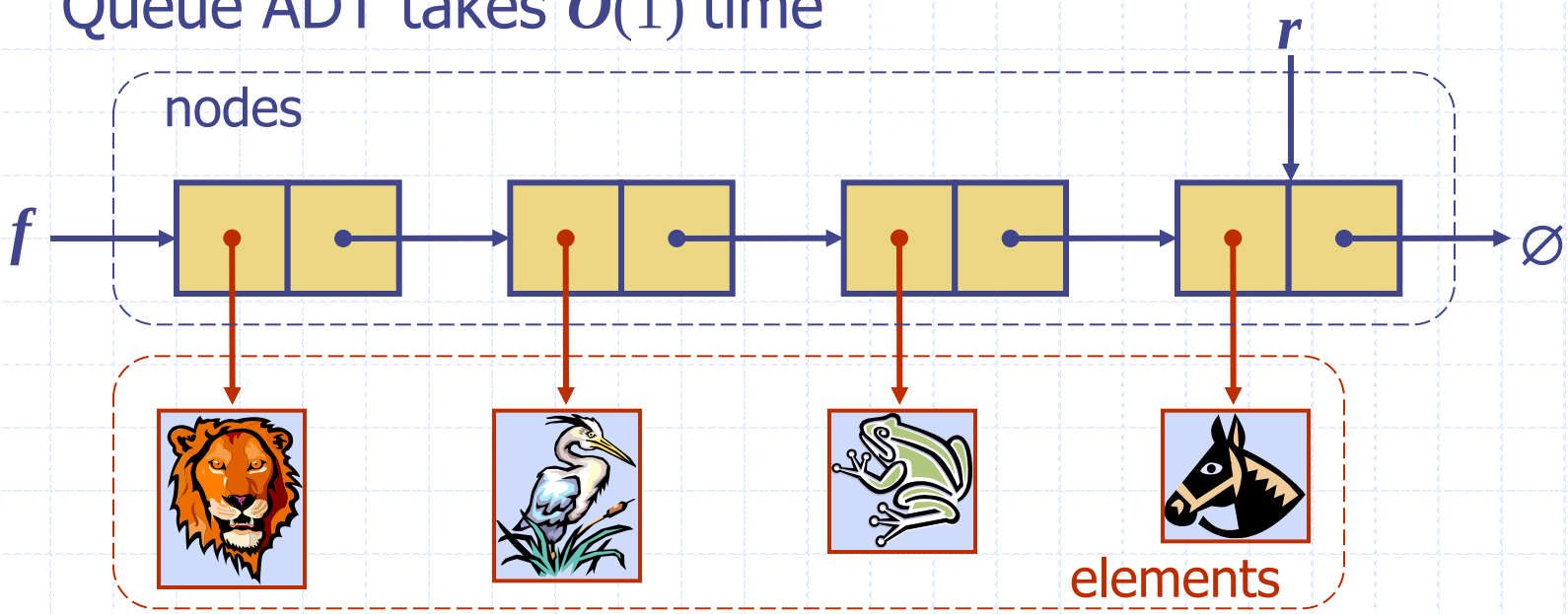
Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



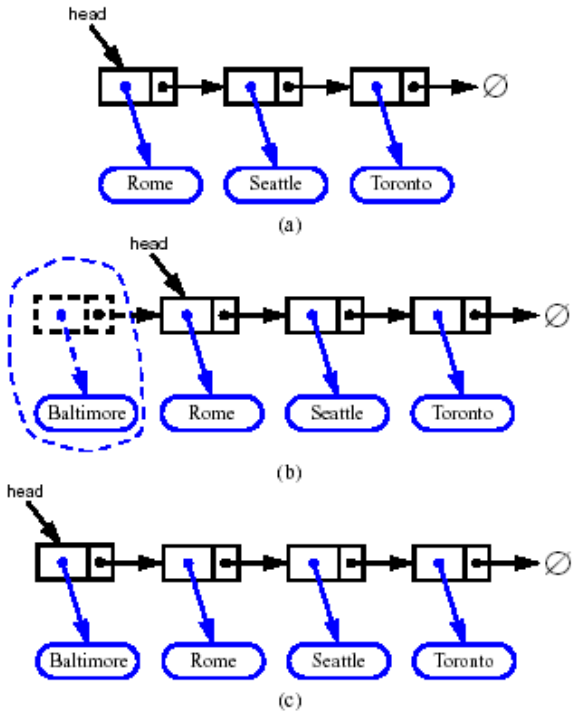
Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



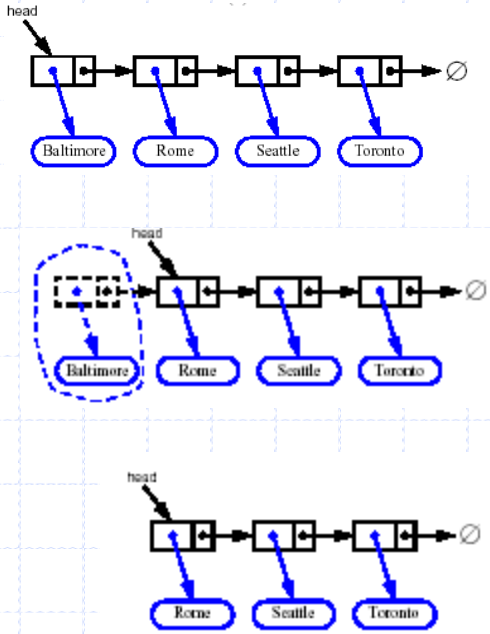
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



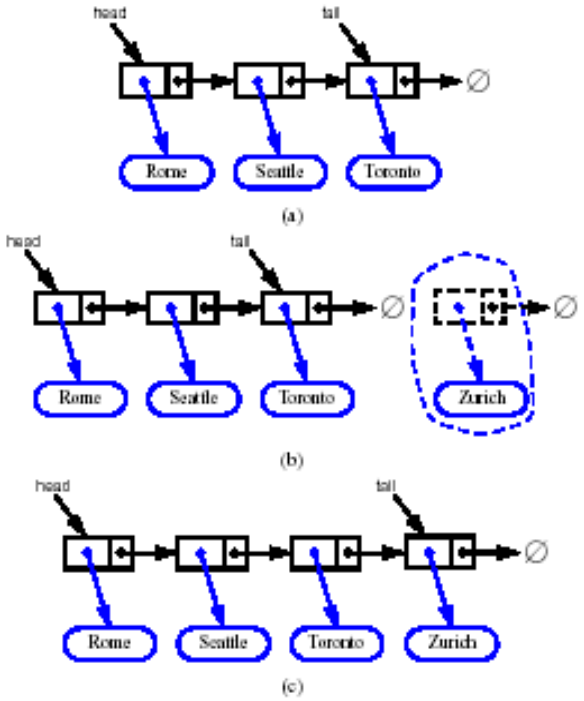
Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



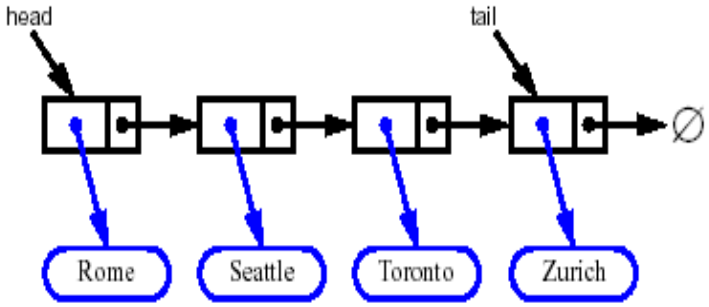
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- A special **null** position refers to no object.
- Positions provide a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- Member functions:
  - Object& element(): returns the element stored at this position
  - bool isNull(): returns true if this is a null position

List ADT

- The List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), isEmpty()
- Query methods:
  - isFirst(p), isLast(p)

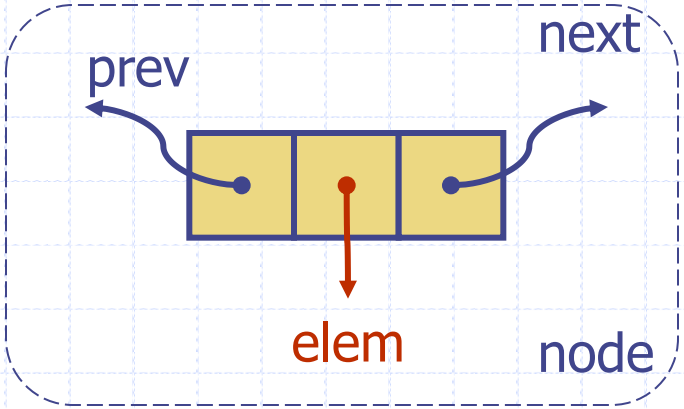
Accessor methods:

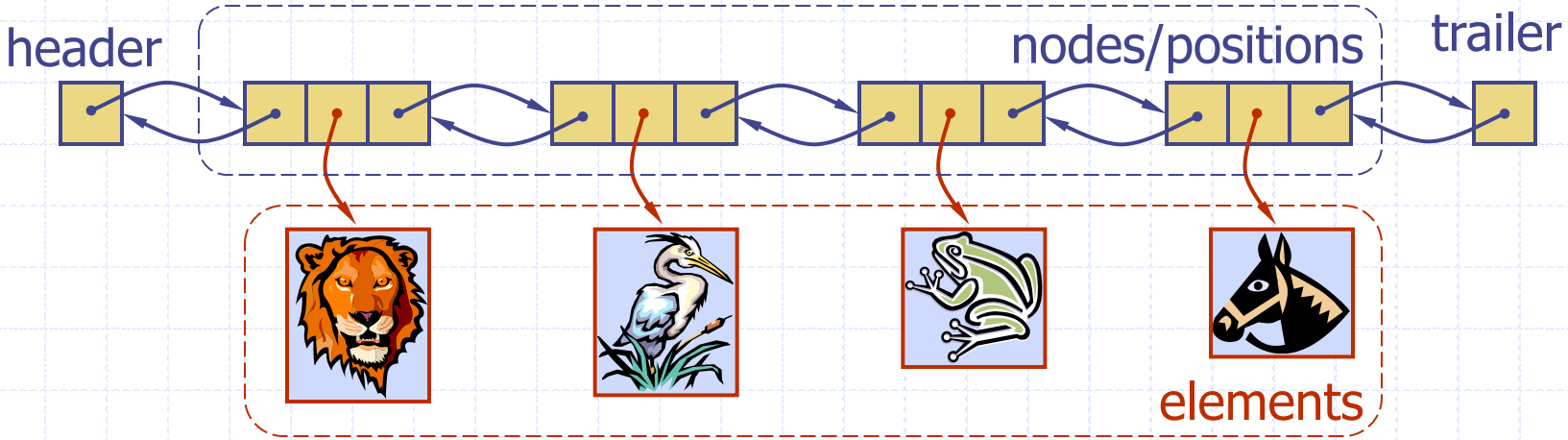
- first(), last()
- before(p), after(p)

- Update methods:
  - replaceElement(p, o), swapElements(p, q)
  - insertBefore(p, o), insertAfter(p, o),
  - insertFirst(o), insertLast(o)
  - remove(p)

Doubly Linked List

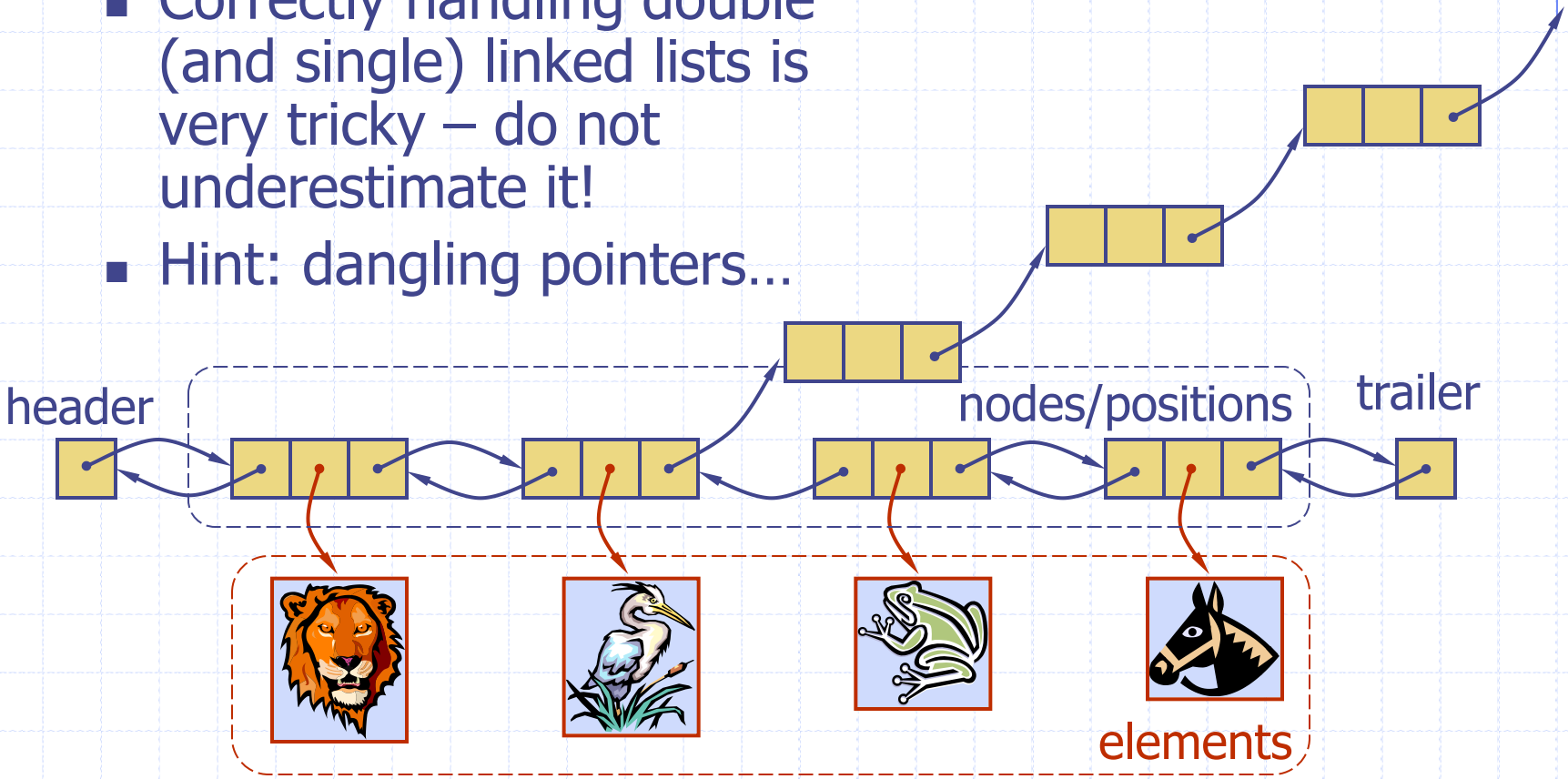
- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- *Optional - special trailer and header nodes*

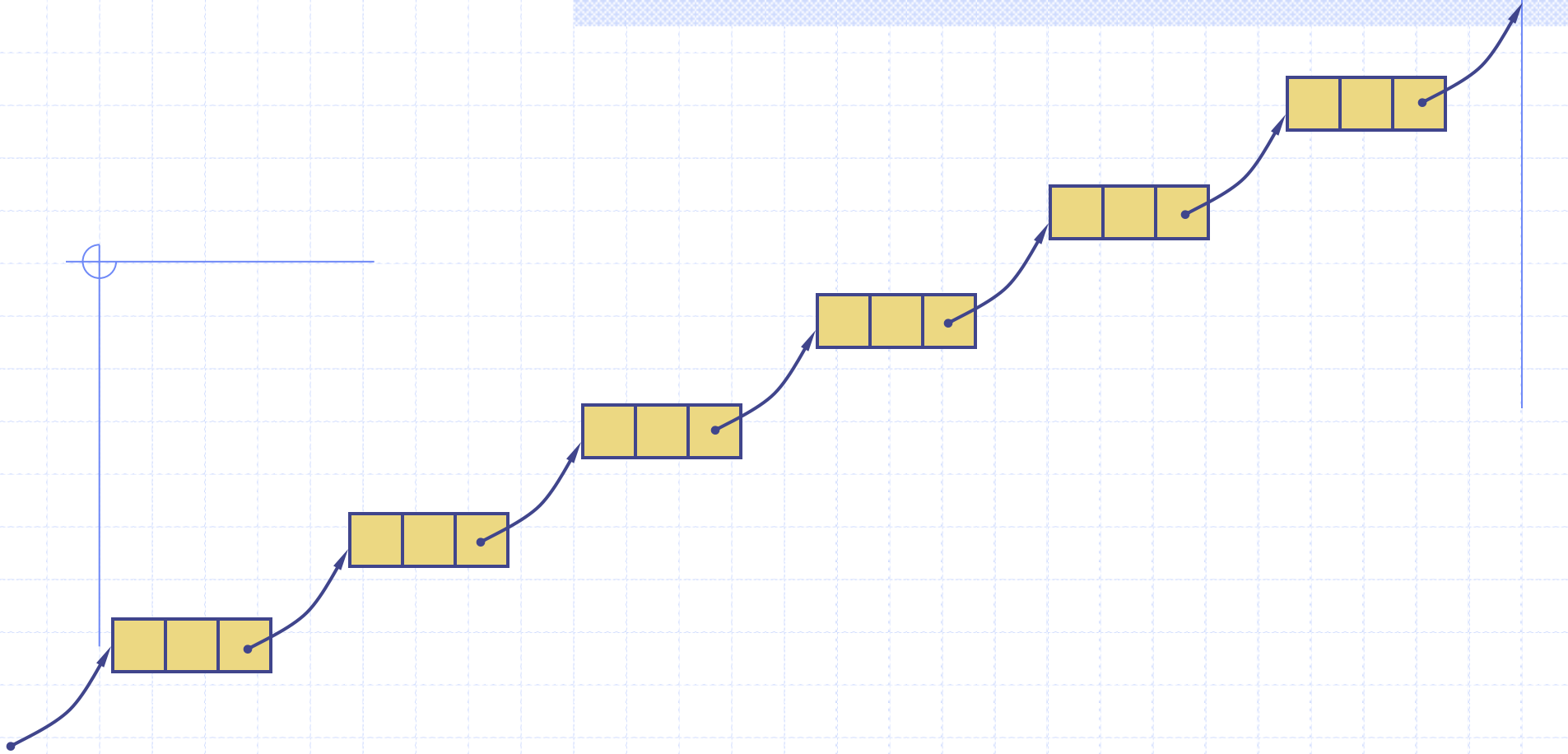




Doubly Linked List

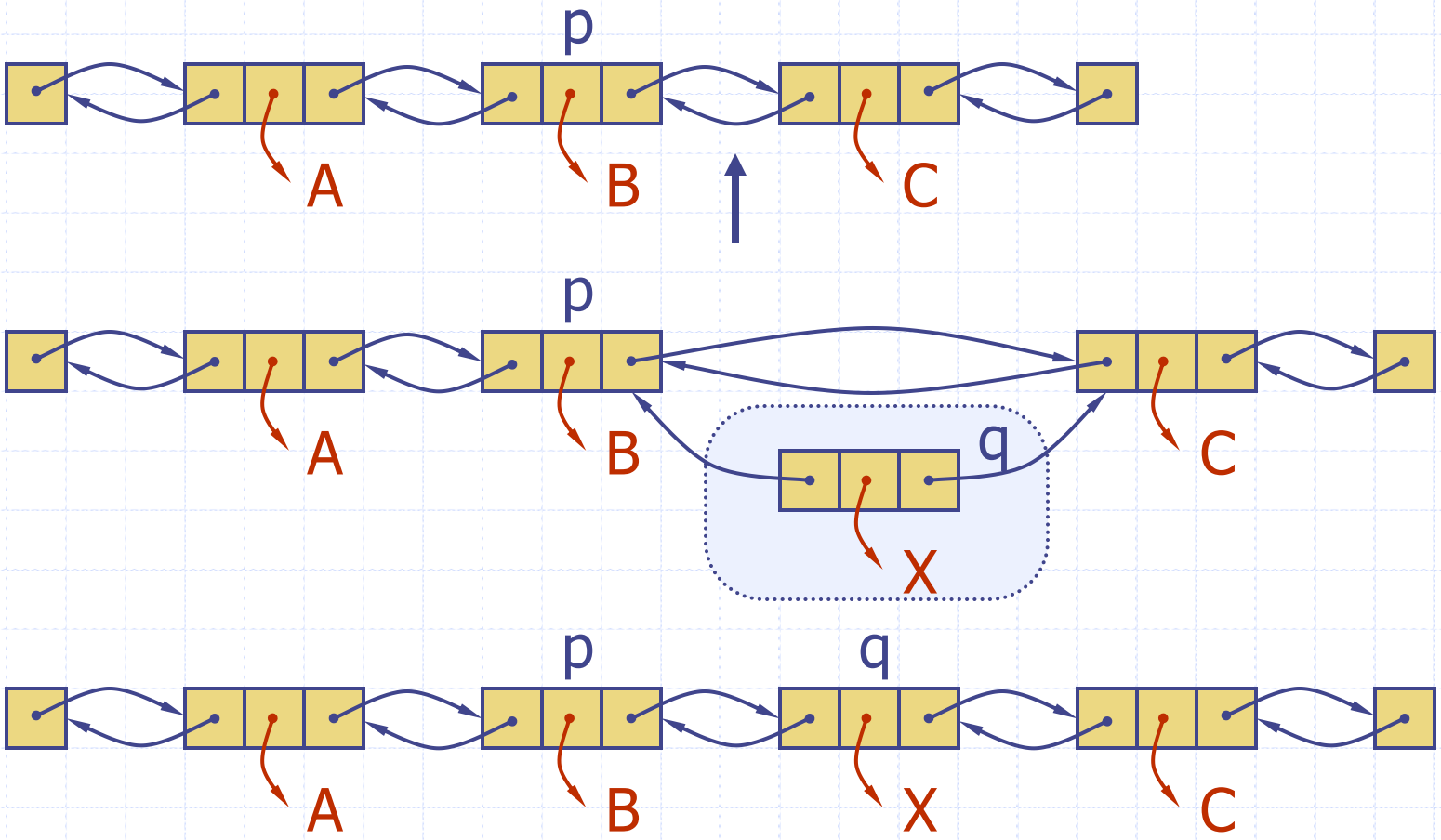
- Very important interjection:
  - Correctly handling double (and single) linked lists is very tricky – do not underestimate it!
  - Hint: dangling pointers…





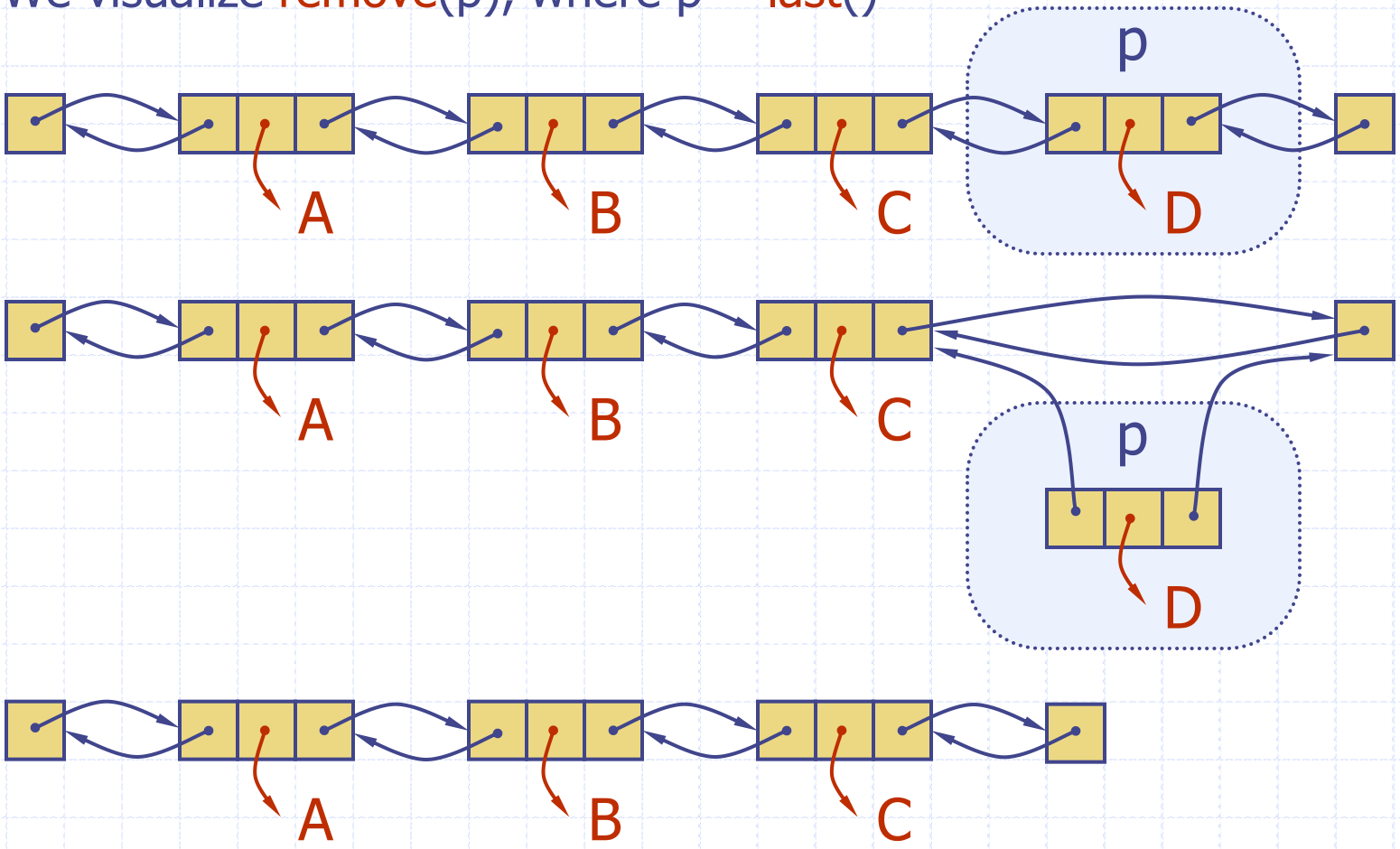
Insertion

- We visualize operation insertAfter(p, X), which returns position q



Deletion

- We visualize remove(p), where p = last()



Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time
  - Operation element() of the Position ADT runs in $O(1)$ time

Sequence ADT

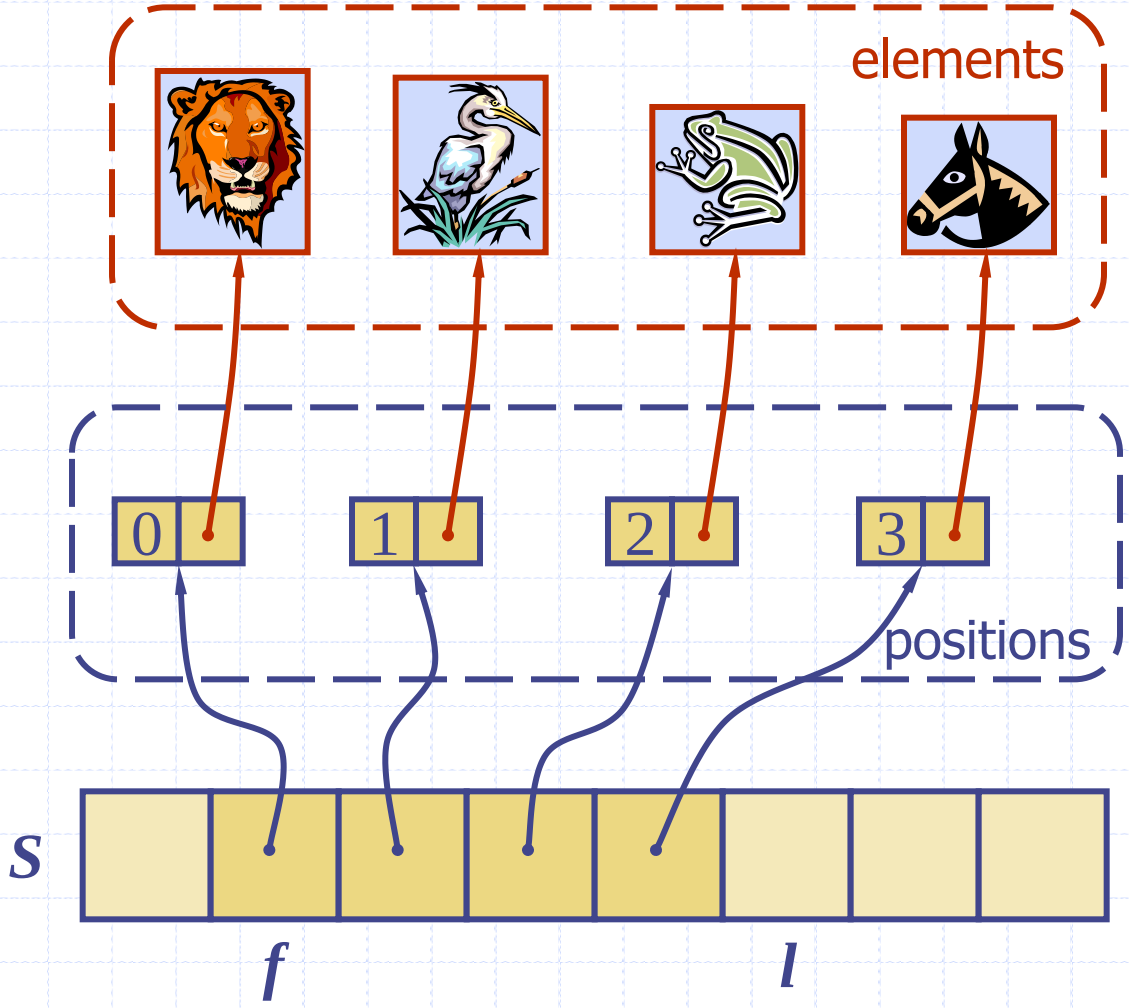
- The Sequence ADT is the union of the Vector and List ADTs
- Elements accessed by
  - Rank, or
  - Position
- Generic methods:
  - size(), isEmpty()
- Vector-based methods:
  - elemAtRank(r), replaceAtRank(r, o), insertAtRank(r, o), removeAtRank(r)
- List-based methods:
  - first(), last(), before(p), after(p), replaceElement(p, o), swapElements(p, q), insertBefore(p, o), insertAfter(p, o), insertFirst(o), insertLast(o), remove(p)
- Bridge methods:
  - atRank(r), rankOf(p)

Applications of Sequences

- The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements
- Direct applications:
  - Generic replacement for stack, queue, vector, or list
  - small database (e.g., address book)
- Indirect applications:
  - Building block of more complex data structures

Array-based Implementation

- We use a circular array storing positions
- A position object stores:
  - Element
  - Rank
- Indices *f* and *l* keep track of first and last positions



Sequence Implementations

Operation	Array	List
size, isEmpty	1	1
atRank, rankOf, elemAtRank	1	n
first, last, before, after	1	1
replaceElement, swapElements	1	1
replaceAtRank	1	n
insertAtRank, removeAtRank	n	n
insertFirst, insertLast	1	1
insertAfter, insertBefore	n	1
remove	n	1

Iterators

- An iterator abstracts the process of scanning through a collection of elements
- Methods of the ObjectIterator ADT:
  - boolean hasNext()
  - object next()
  - reset()
- Extends the concept of position by adding a traversal capability
- May be implemented with an array or singly linked list
- An iterator is typically associated with an another data structure
- We can augment the Stack, Queue, Vector, List and Sequence ADTs with method:
  - ObjectIterator elements()
- Two notions of iterator:
  - snapshot: freezes the contents of the data structure at a given time
  - dynamic: follows changes to the data structure

Comparison of Position, List, and Sequence ADTs

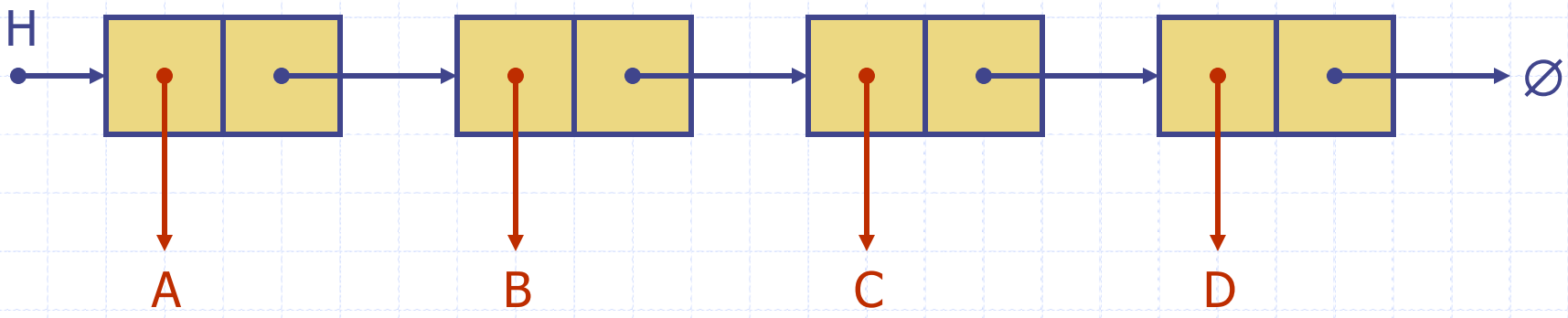
- Position ADT
  - Keeps track of the "position" of an element within the data structure
    - E.g., "element at X", "array-like indexing"
- List ADT
  - Models a sequence of positions storing elements
    - E.g., "first", "last", "isBefore", "linked list"
- Sequence ADT
  - Contains both position and list data
    - E.g., "vector+linked-list"

Linked List Operations

- 4 nodes in all cases
- 1. Swap in singly linked list
  - Swap node 2 and 3
- 2. Swap in double linked list
  - Swap node 2 and 3
- 3. Swap 1st and last in singly linked list
- 4. Swap 1st and last in doubly linked list

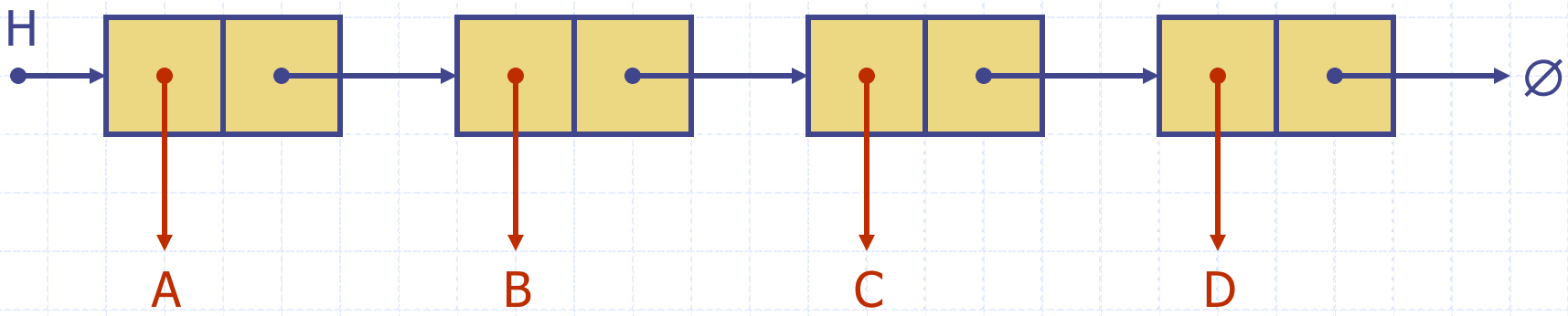
Linked List Operations

- Swap
  - You have the pointer H
  - Node fields are “data” and “next”
  - How do you most efficiently swap B and C using only one auxiliary variable space?



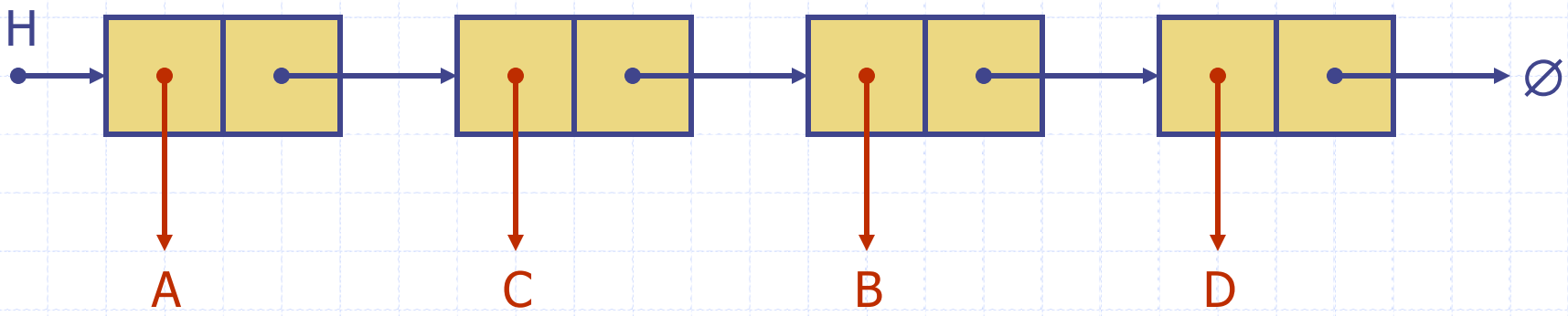
Linked List Operations

- Swap
  - B ← H->next
  - H->next ← B->next
  - B->next ← B->next->next
  - H->next->next ← B



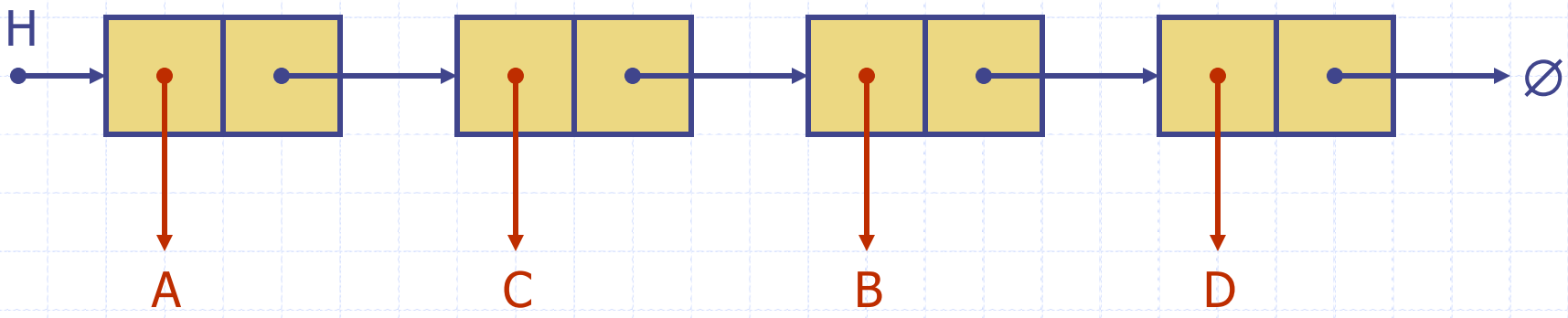
Linked List Operations

- Swap
  - B ← H->next
  - H->next ← B->next
  - B->next ← B->next->next
  - H->next->next ← B



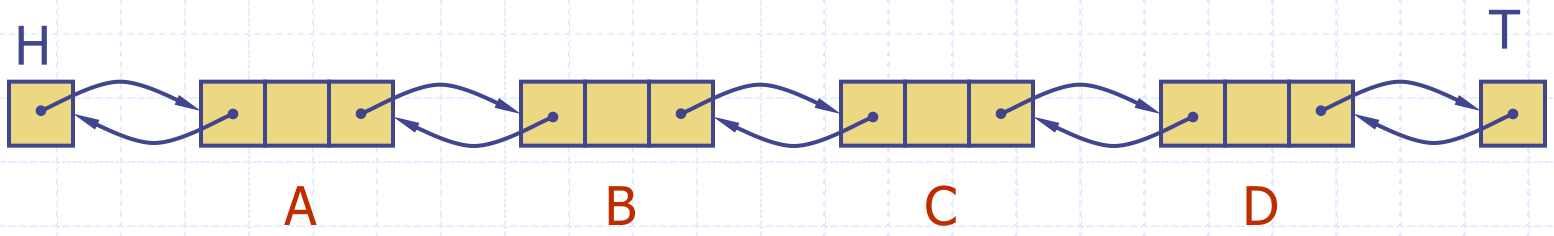
Linked List Operations

- Swap(Node *curr, Node *prev)
  - prev->next ← curr->next
  - curr->next ← curr->next->next
  - prev->next->next ← curr
- What if curr = head?



Doubly Linked List Operations

- Swap
  - How do you most efficiently swap C and D using only one auxiliary variable space?



Doubly Linked List Operations

Swap(Node *curr, Node *next)

curr->prev->next ← next

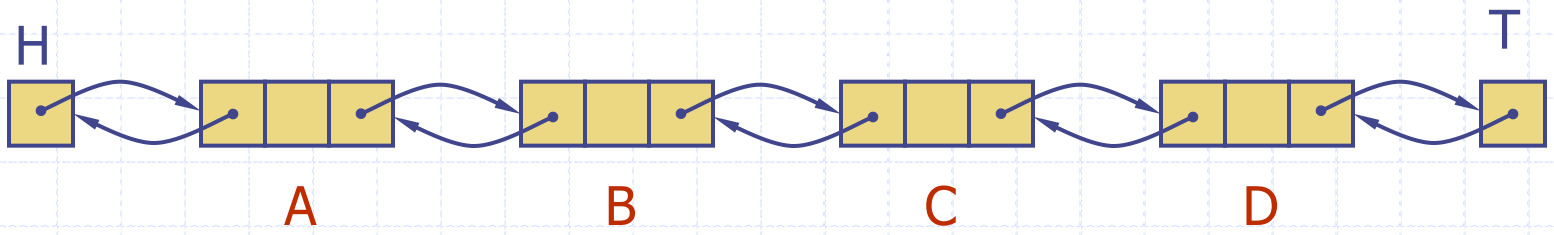
curr->next ← next->next

next->next ← curr

next->prev ← curr->prev

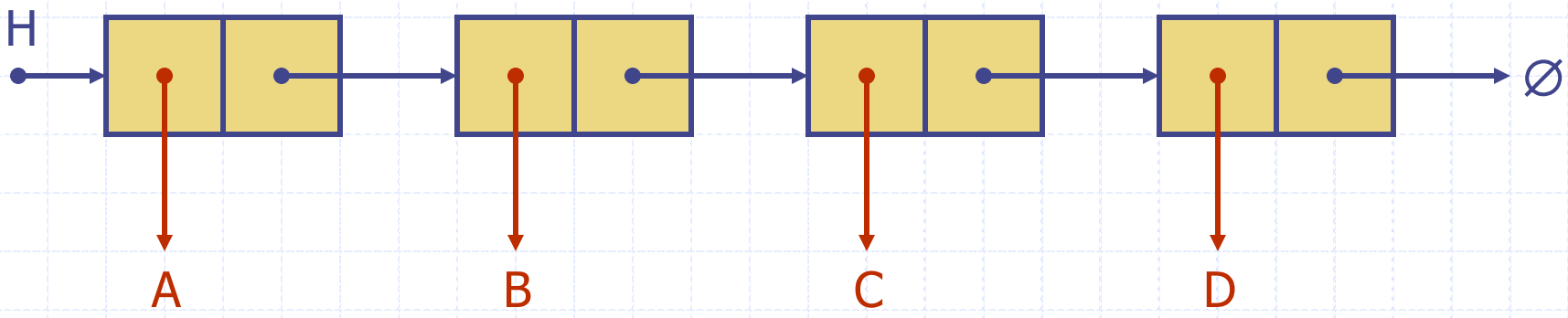
curr->next->prev ← curr

curr->prev ← next



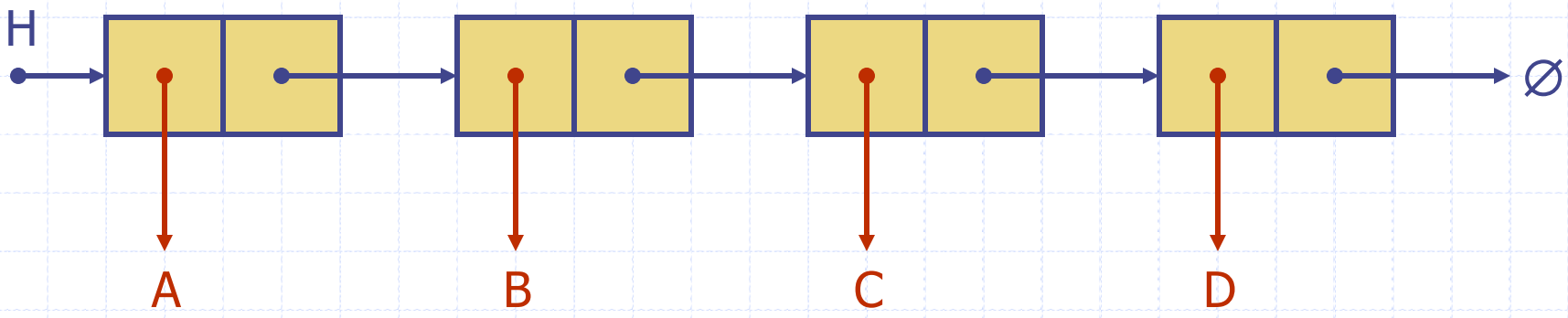
Linked List Operations

- Reverse
  - How do I efficiently "reverse" the order of the linked list?



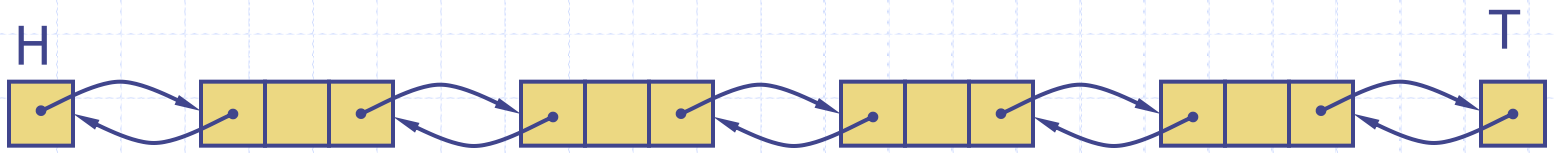
Linked List Operations

```
Reverse(Node *curr, Node *prev)
    if (curr->next = NULL) {
        curr->next ← prev;
        newHead ← curr;
    } else {
        newHead ← Reverse(curr->next, curr);
        curr->next ← prev;
    }
    return (newHead);
```



Doubly Linked List Operations

- Reverse
  - How do I efficiently "reverse" the order of a doubly linked list?



Doubly Linked List Operations

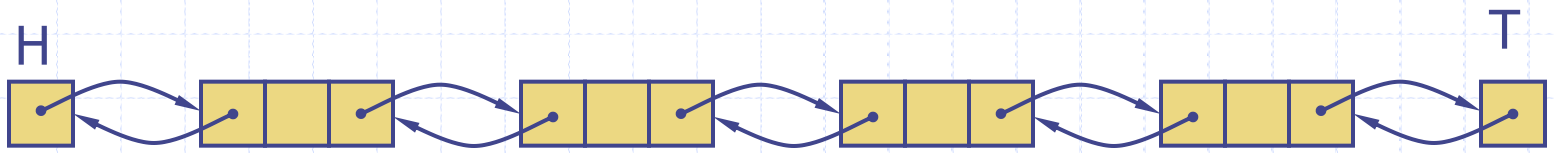
- Can use the previous Reverse() method or if the semantics of next/prev are flexible then…

Reverse(Node *head, Node *tail)

tmp ←head

head ←tail

tail ←tmp



Doubly Linked List Operations

- Can use the previous Reverse() method or if the semantics of next/prev are flexible then…

Reverse(Node *head, Node *tail)

tmp ←head

head ←tail

tail ←tmp