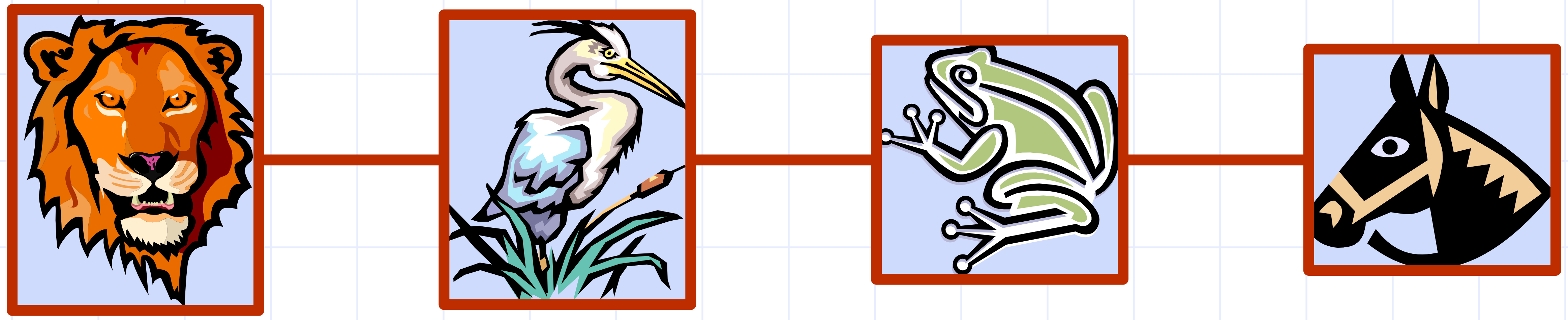
Linked Lists



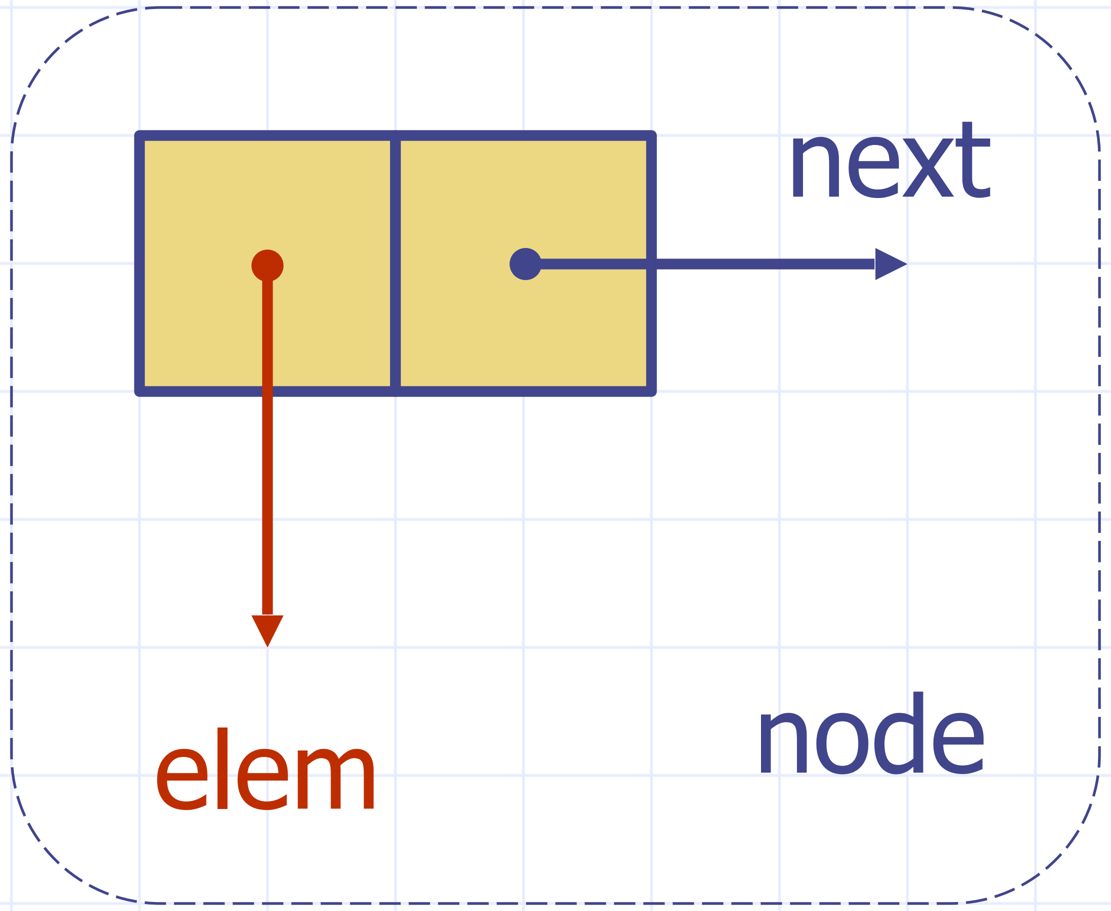
Linked Lists

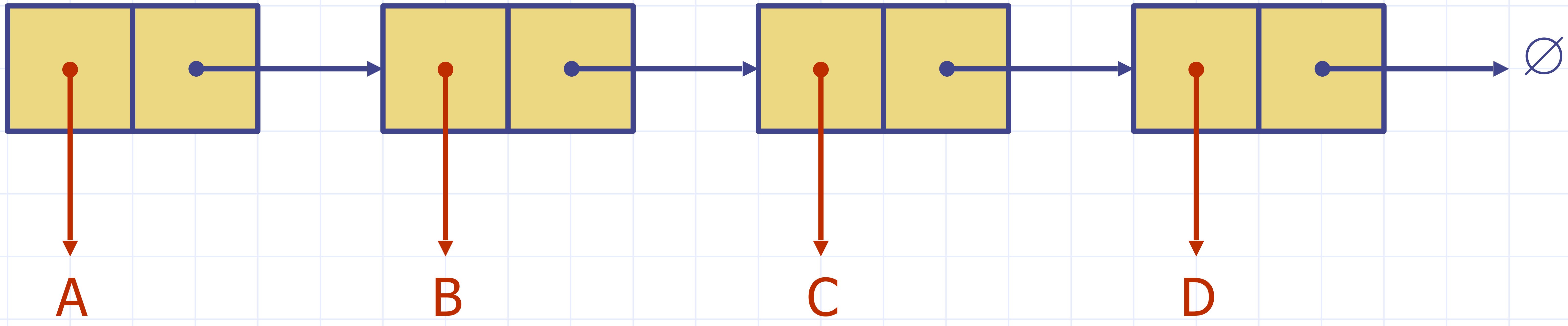
- Linked lists are a data structure that grows to exactly the size needed
  - "Nodes" of data are only added as needed

- With carefully coding, they can shrink to exactly the size needed
  - requires careful "garbage collection" of unused nodes

Singly Linked List

- A singly linked list is a data structure consisting of a sequence of nodes
- Each node stores
  - element (or pointer to)
  - pointer to the next node





Linked Lists

- Linked lists are usually described by tracking a “head” pointer.
  - points to first node in the list
- Sometimes a “tail” pointer is also tracked
  - points to the last node in the list.

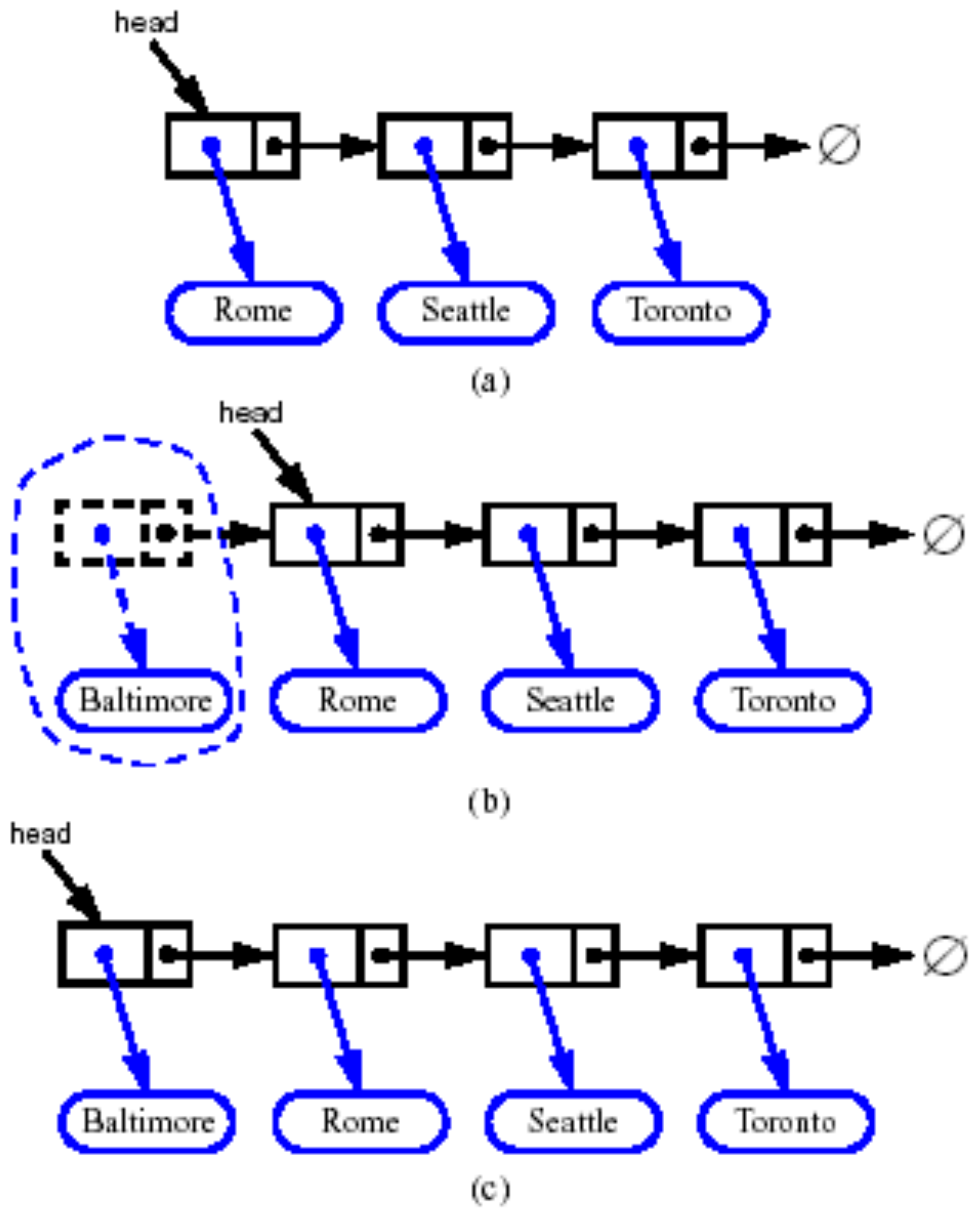
Linked List Traversal

- goal: examine each node in the list:

```
Algorithm traverse(head):
    nodePtr = head
    while (nodePtr != null)
        examine (nodePtr -> element) // i.e. process node
        nodePtr = nodePtr -> next
```

Inserting at the Head

1. Allocate a new node
2. Store element in new node
3. Have new node point to old head (see (b) in figure)
4. Update head to point to new node (see (c) in figure)



Inserting at the Head

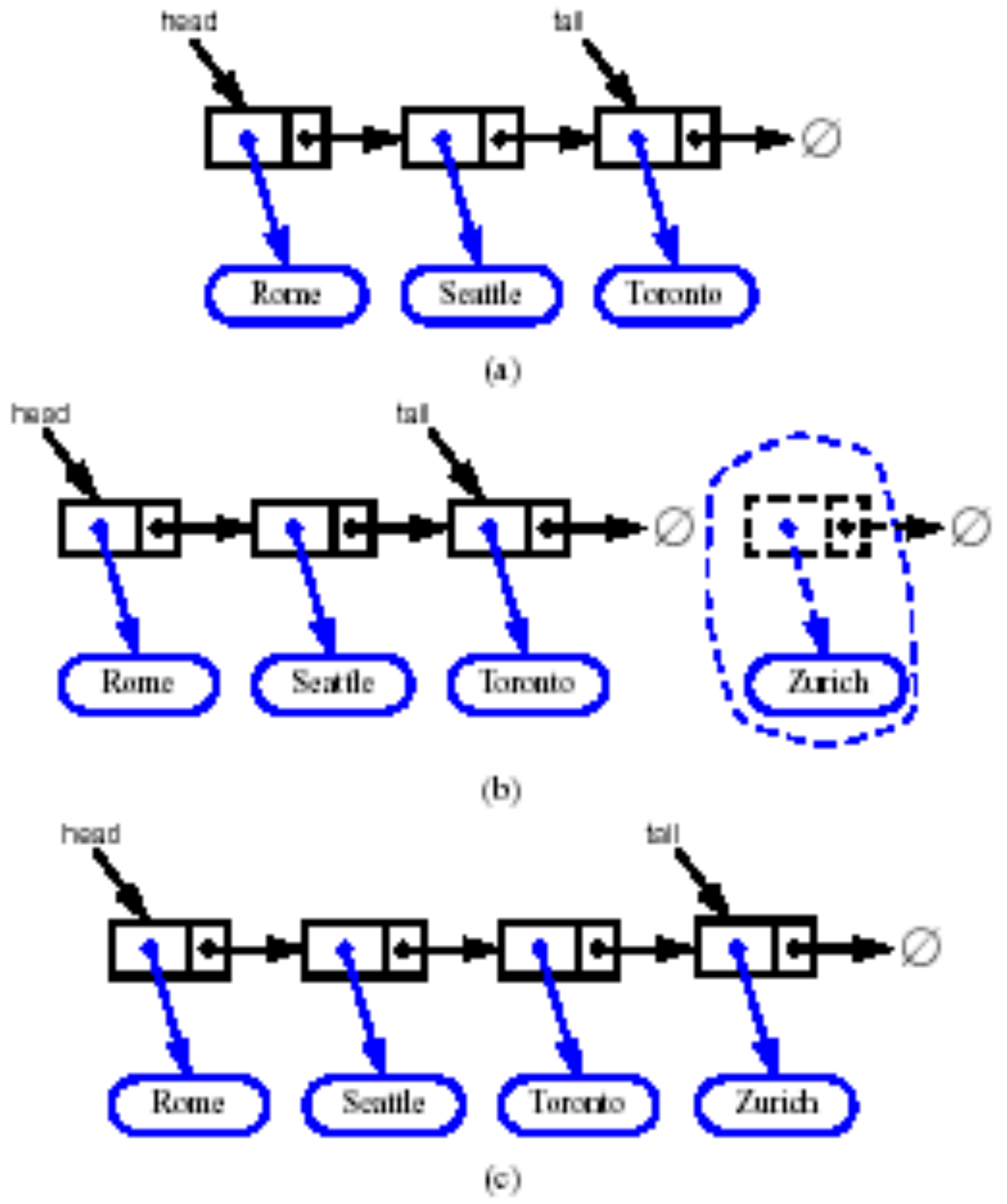
- goal: insert new node at front of the list
  - note: modifies head of list!

Algorithm insertAtHead(head, newValue):

```
nodePtr = new Node(newValue)
nodePtr -> next = head
head = nodePtr
```

Inserting at the Tail

1. Allocate a new node
2. Store new element in new node
3. Have new node point to null (see (b) in figure)
4. Have tail->next point to new node (see (c) in figure)
5. Update tail to point to new node



Inserting at the Tail

- goal: insert new node at end of the list
  - modifies tail of list!
  - only works if list is non-empty to start with

Algorithm insertAtTail(tail, newValue):

```
nodePtr = new Node(newValue)
nodePtr -> next = null
tail -> next = nodePtr
tail = nodePtr
```

Removing at the Head

Update head to point to next node in the list

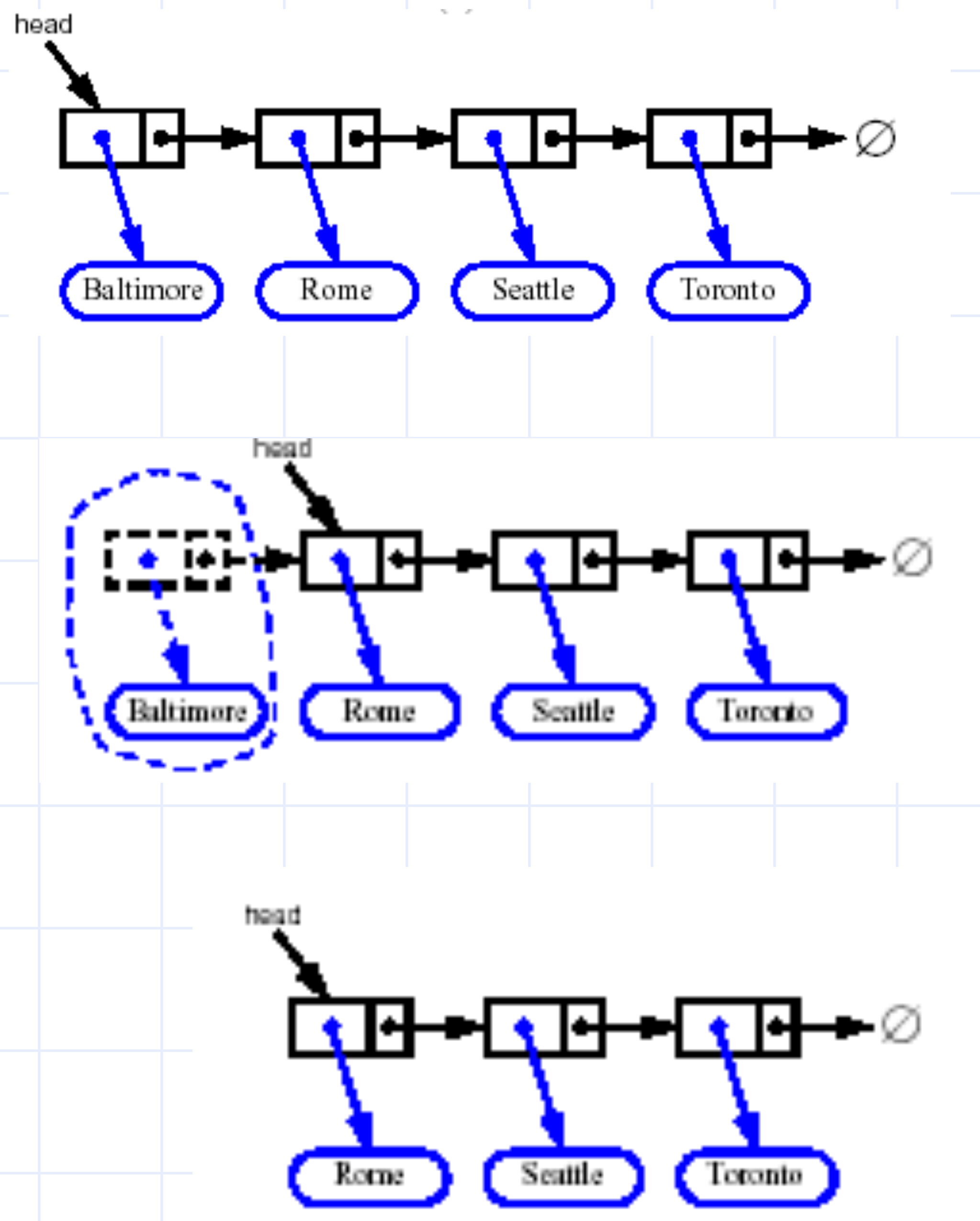
Algorithm removeHead(head):

nodePtr = head

head = head -> next;

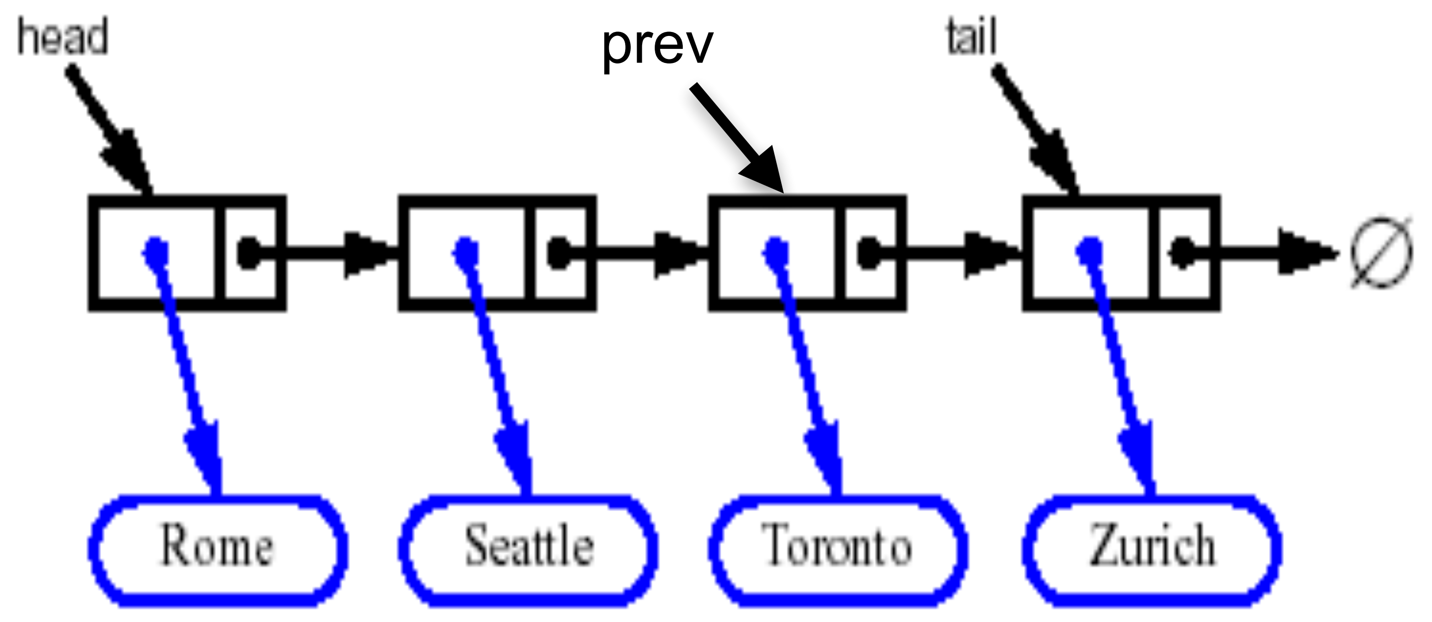
nodePtr -> next = null

return nodePtr



Removing at the Tail

- first, find node before the tail (by list traversal).
- set tail to node just found
- requires two nodes in list



Algorithm removeTail(tail):

answer = tail

prev = findPrev(tail)

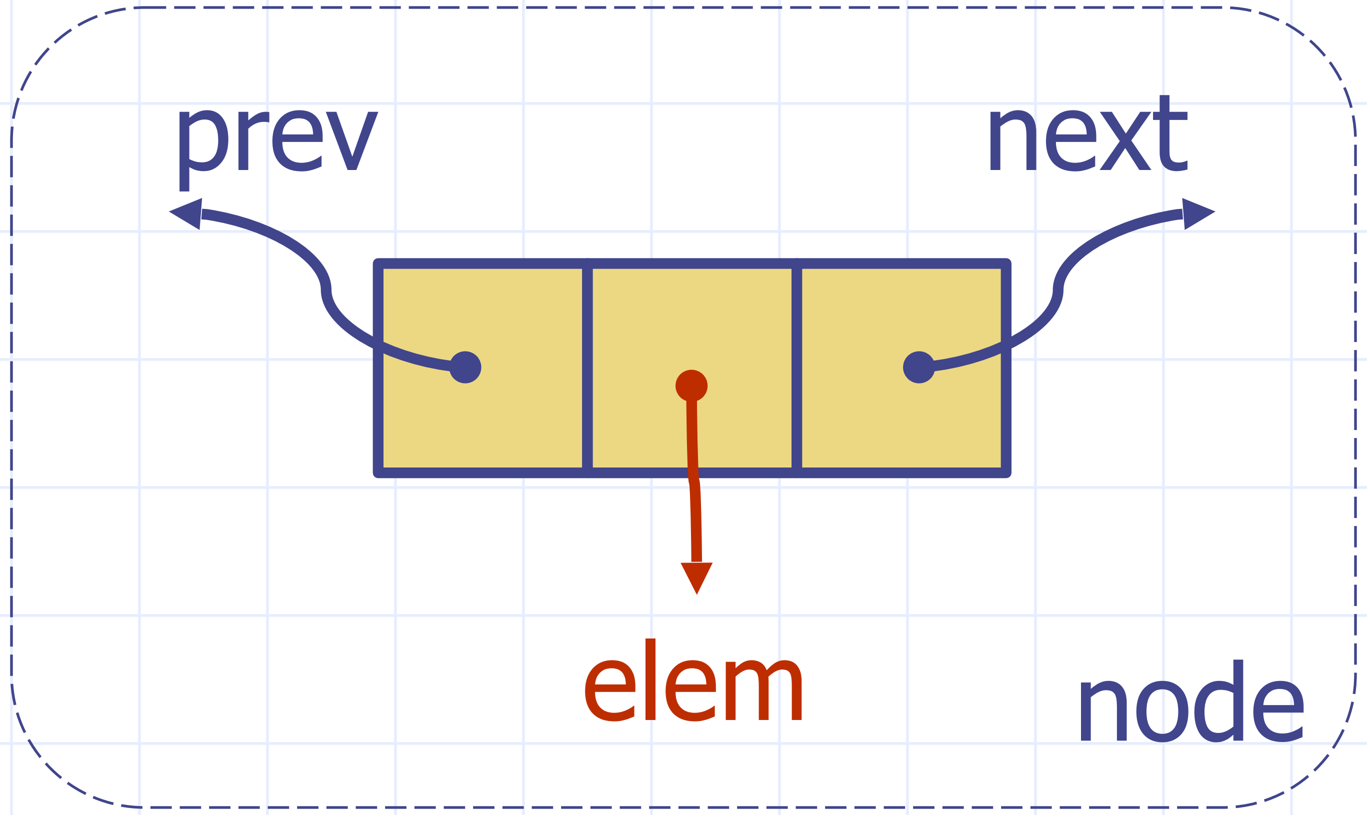
tail = prev

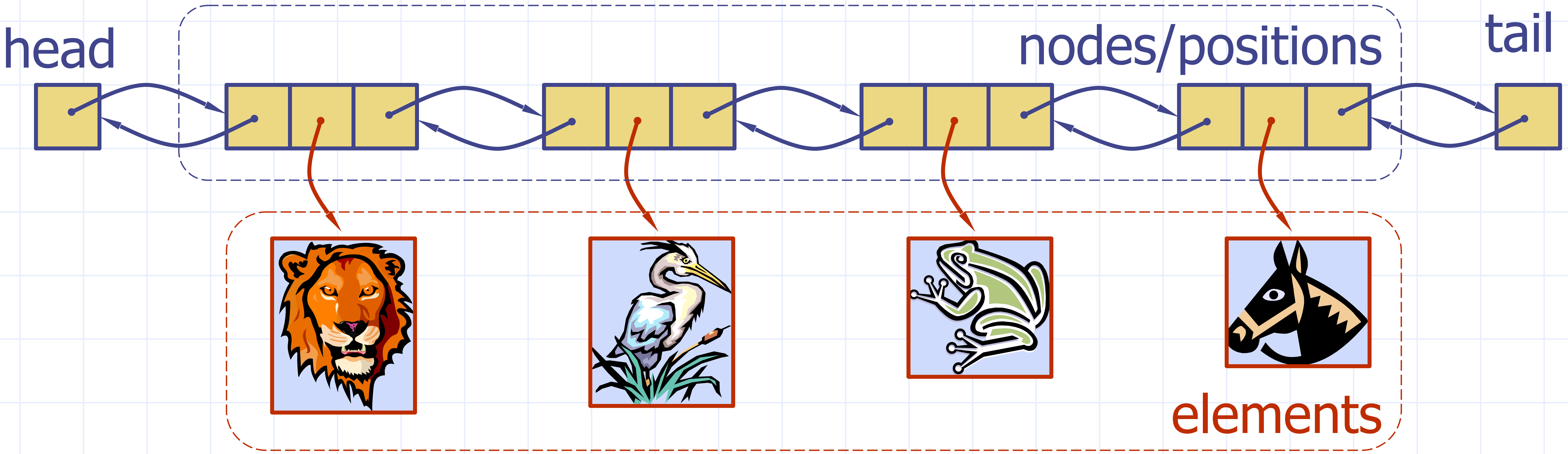
tail -> next=null

return answer

Doubly Linked Lists

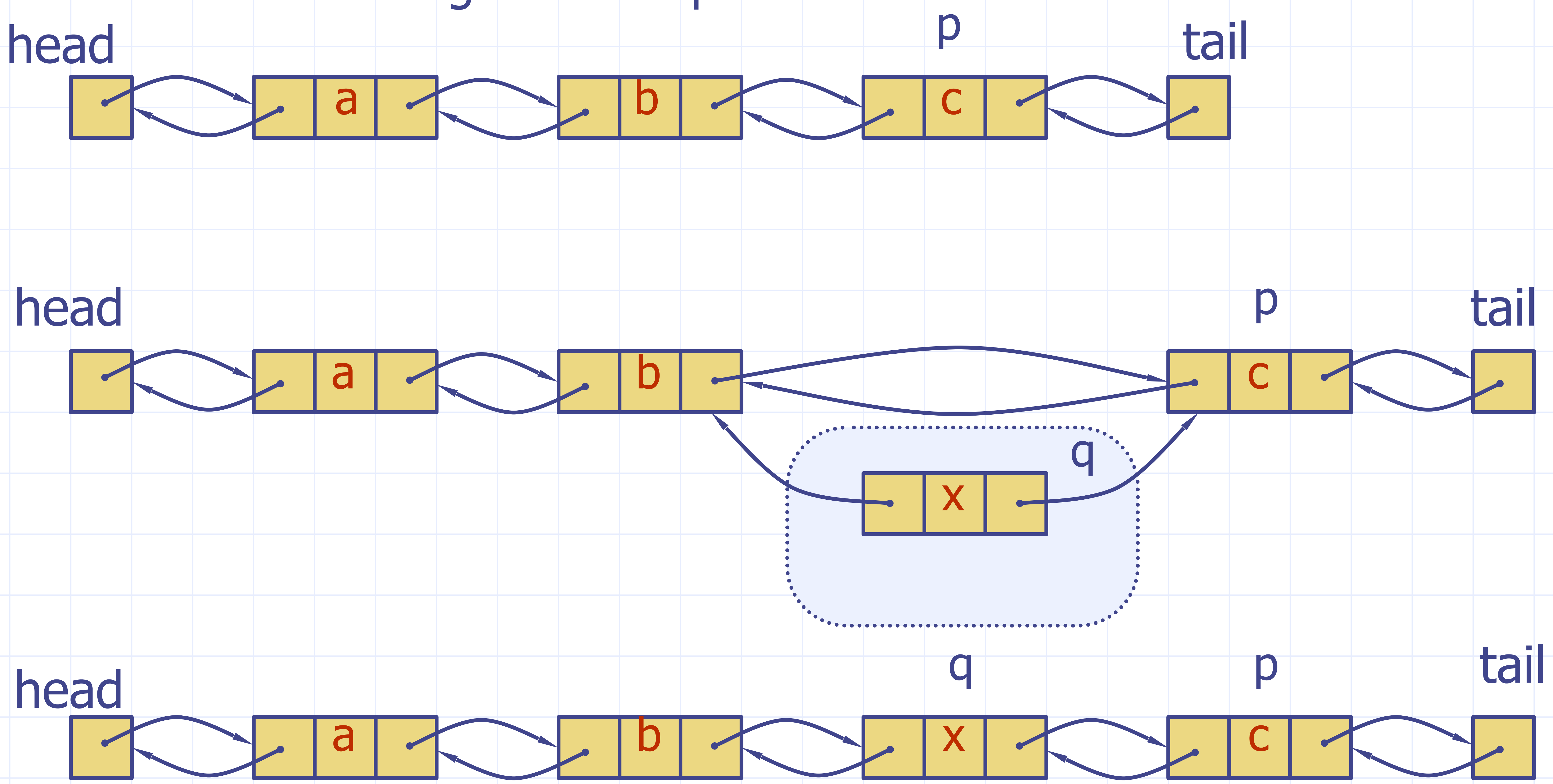
- A doubly linked list node keeps both next and previous links
- has a head pointer …
- and a tail pointer





Insertion

- Consider inserting x before p:

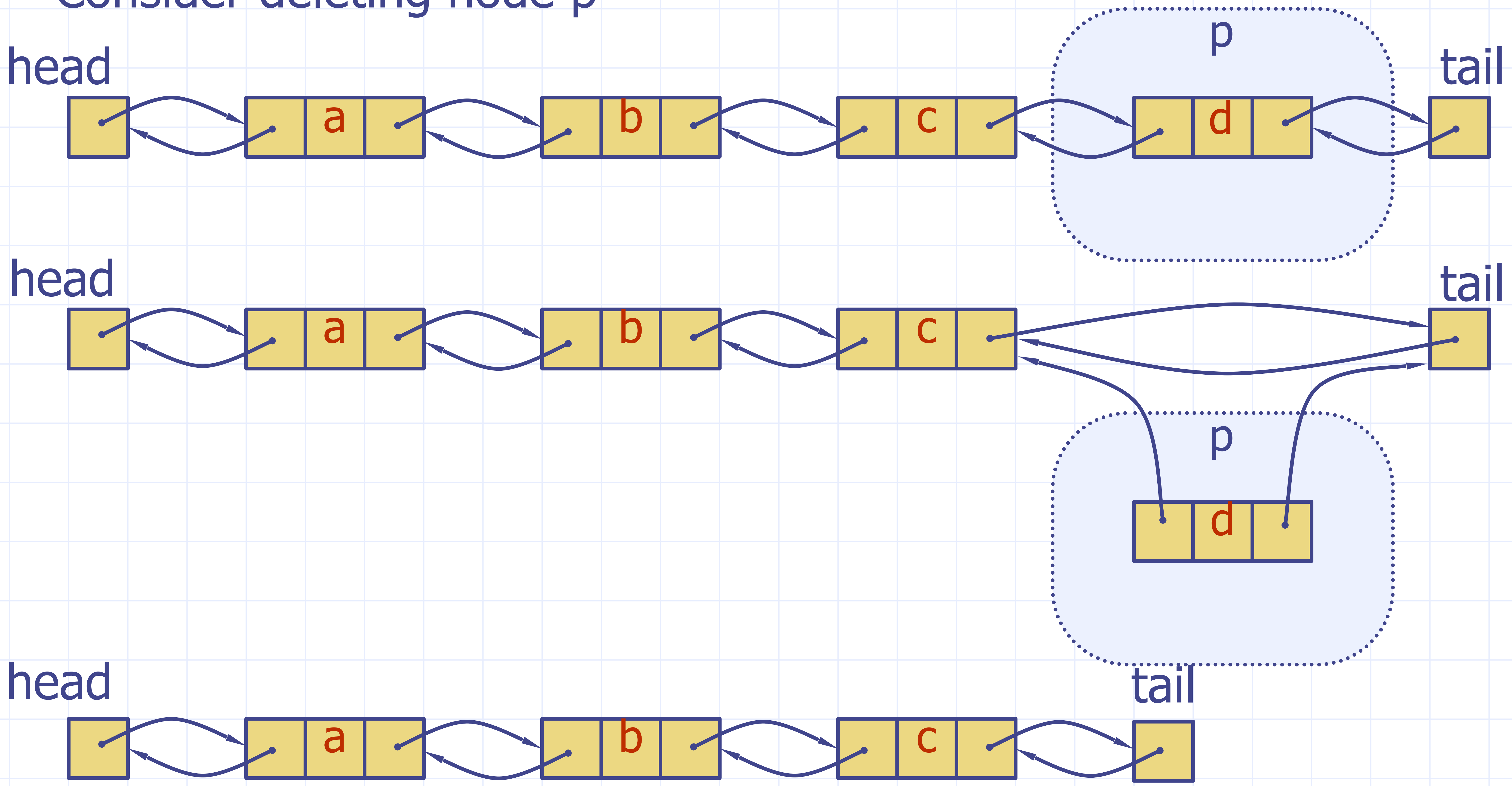


Insertion Algorithm

```
Algorithm insert(p, newValue):  //insert newValue before p
    newNode = new Node(newValue)
    before = p –> prev

    newNode –> next = p

    p→prev = newNode
    newNode –> prev = before

    before -> next = newNode
```

Removing a Node

- Consider deleting node p



Removal Algorithm

Algorithm remove(p):

before = p→prev

after = p→next

before→next = after

after→prev = before

Memory Leaks

- Where does a deleted node go?
  - without further action in C++, it is a “memory leak”
  - a “memory leak” is unused memory without any references to it in the running program.
  - Java automatically “sweeps” such leaks back up for future use via “garbage collection”
  - C++ does **<u>no</u>** automatic garbage collection!!

The C++ delete operator

- Want to release storage back to C++?
  - use the delete command:

```
int *ptr = new int;
// … work with storage at ptr for a while …
delete ptr;  // restores storage to C++
```

  - special case for when pointer is really an array:

```
int *ptr = new int[100];
// … work with storage at ptr for a while …
delete [] ptr; // restores entire array
```

- Extremely dangerous to look at contents of ptr after delete
  - C++ has likely allocated that memory to something else!