Data Structures and Algorithms [1]

BITS-Pilani K. K. Birla Goa Campus

[1] Material for the presentation taken from Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Fourth Edition*;

- Dynamic sets : must be able to add and delete items (i.e. must be dynamic).

Ch. 10 : Elementary Data Structures

- Dynamic sets : must be able to add and delete items (i.e. must be dynamic).
- Max Priority Queue

Ch. 10 : Elementary Data Structures

- Dynamic sets : must be able to add and delete items (i.e. must be dynamic).
- Max Priority Queue
  - Extract-Maximum

- Dynamic sets : must be able to add and delete items (i.e. must be dynamic).
- Max Priority Queue
  - Extract-Maximum
  - Insert-Key

- Dynamic sets : must be able to add and delete items (i.e. must be dynamic).
- Max Priority Queue
  - EXTRACT-MAXIMUM
  - INSERT-KEY
  - Worst-case time complexity : $O(\lg n)$

Ch. 10 : Elementary Data Structures

- Dynamic sets : must be able to add and delete items (i.e. must be dynamic).
- Max Priority Queue
  - EXTRACT-MAXIMUM
  - INSERT-KEY
  - Worst-case time complexity : $O(\lg n)$
- Data structures for dynamic sets : Stacks, Queues and Linked list.

- Stack : last-in, first-out (LIFO)

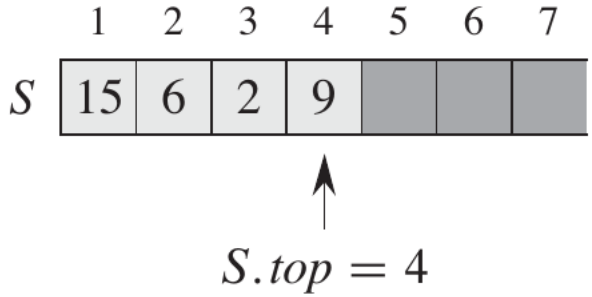
- Stack : last-in, first-out (LIFO)
- There are many ways of implementing stacks.

Elementary data structure: Stack

- PUSH and POP

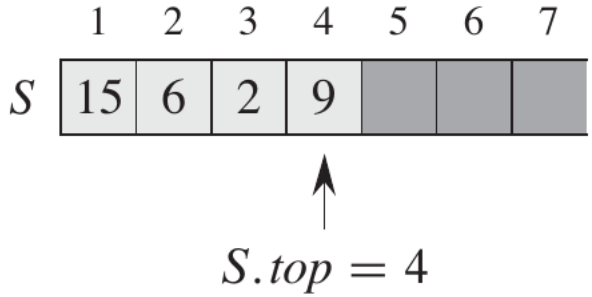
Elementary data structure: Stack

- PUSH and POP



Elementary data structure: Stack

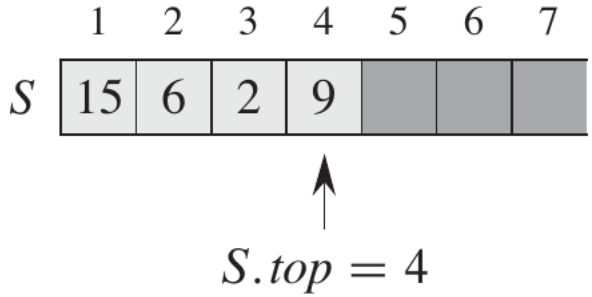
- PUSH and POP



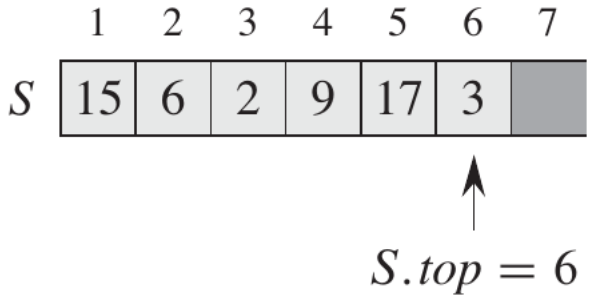
- PUSH(S,17) , PUSH(S,3)

Elementary data structure: Stack

- PUSH and POP

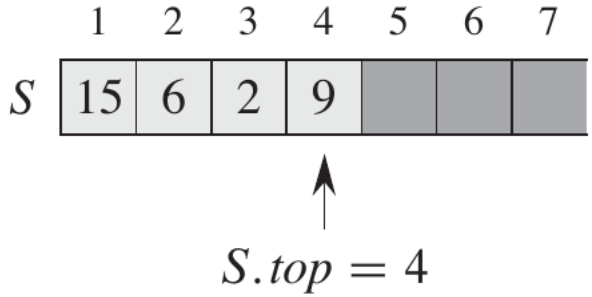


- PUSH(S,17) , PUSH(S,3)

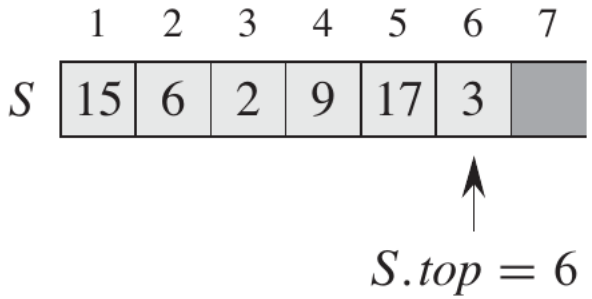


Elementary data structure: Stack

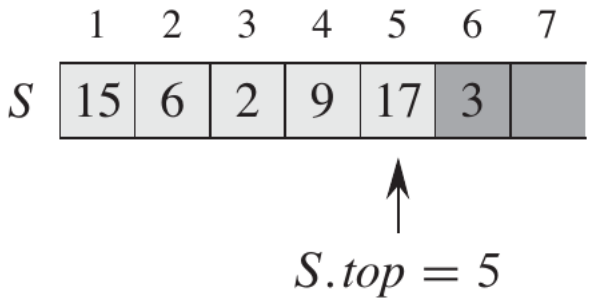
- PUSH and POP



- PUSH(S,17) , PUSH(S,3)



- POP(S)



Stack operation pseudocode

```
STACK-EMPTY(S)
1  if S.top == 0
2        return TRUE
3  else return FALSE
```

Stack operation pseudocode

```
STACK-EMPTY(S)
1  if S.top == 0
2        return TRUE
3  else return FALSE

PUSH(S, x)
1  if S.top == S.size
2        error "overflow"
3  else S.top = S.top + 1
4        S[S.top] = x
```

Stack procedure pseudocode

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  if S.top == S.size
2       error "overflow"
3  else S.top = S.top + 1
4       S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

Stack operations

- Time complexity of stack operations?

Stack operations

- Time complexity of stack operations?
- Stack operations can lead to overflows and underflows.

Balanced Parentheses

- Input : [( )] { } { [( ) ( )] ( ) }

Balanced Parentheses

- Input : [( )] { } { [( ) ( )] ( ) }
  Output : True

Balanced Parentheses

- Input : [( )] { } { [( ) ( )] ( ) }

  Output : True
- Input : { ( ) [}]

Balanced Parentheses

- Input : [( )] { } { [( ) ( )] ( ) }

  Output : True
- Input : { ( ) [}]

  Output : False

Balanced Parentheses

- Input : [( )] { } { [( ) ( )] ( ) }

  Output : True
- Input : { ( ) [}]

  Output : False
- Can we use a Stack to solve this problem?

Balanced Parentheses

- Input : [( )] { } { [( ) ( )] ( ) }
  Output : True
- Input : { ( ) [}]
  Output : False
- Can we use a Stack to solve this problem?
- Use the `std::stack` container adapter or the `deque` template class.

Elementary data structure: Queue

- Queue : first-in, first-out (FIFO)

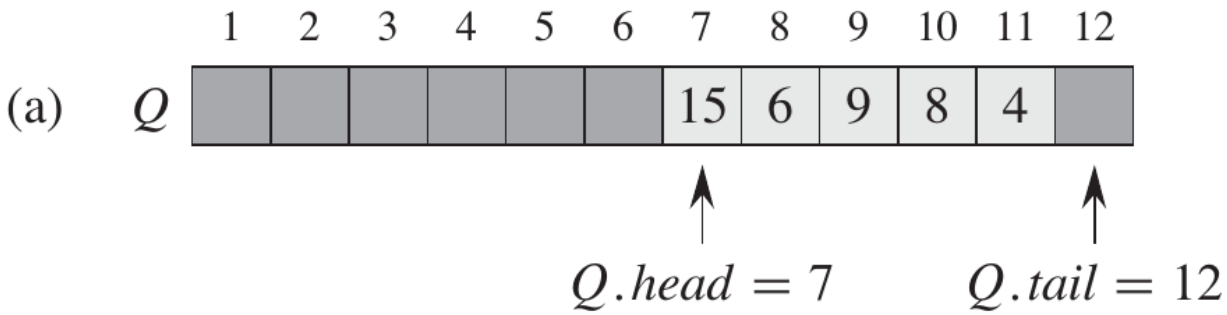
Elementary data structure: Queue

- Queue : first-in, first-out (FIFO)
- There are many ways of implementing queues.

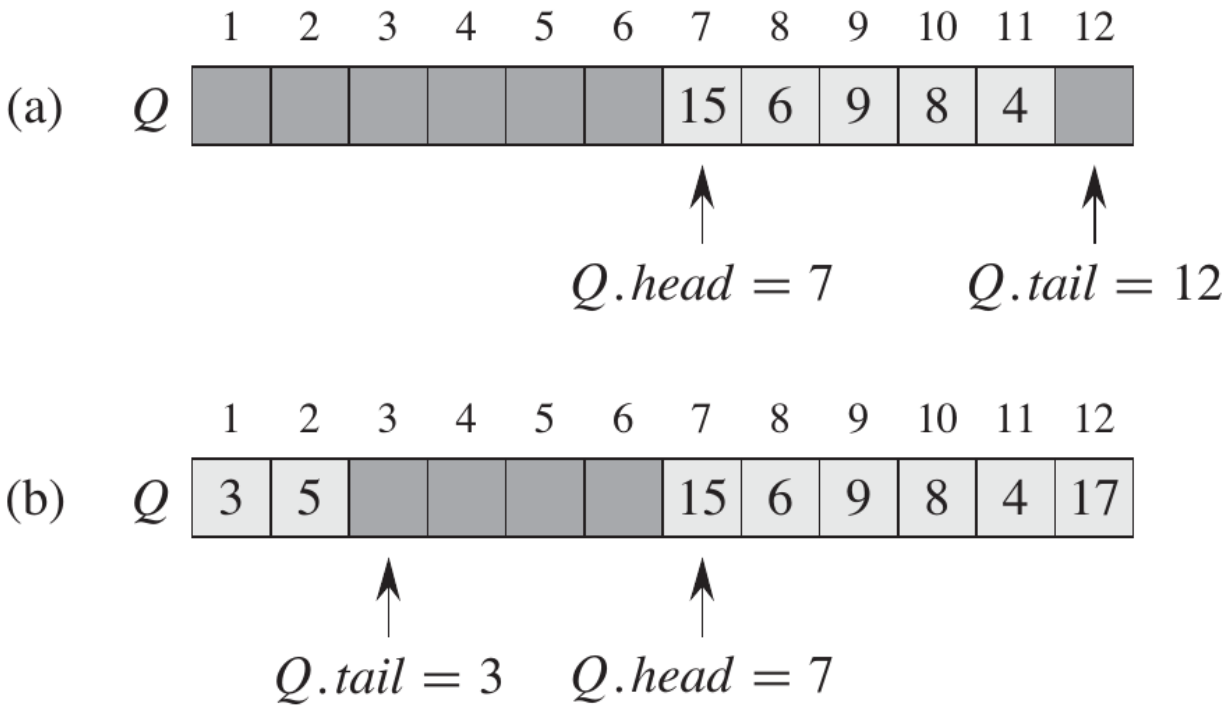
Elementary data structure: Queue

- Queue : first-in, first-out (FIFO)
- There are many ways of implementing queues.
- ENQUEUE and DEQUEUE

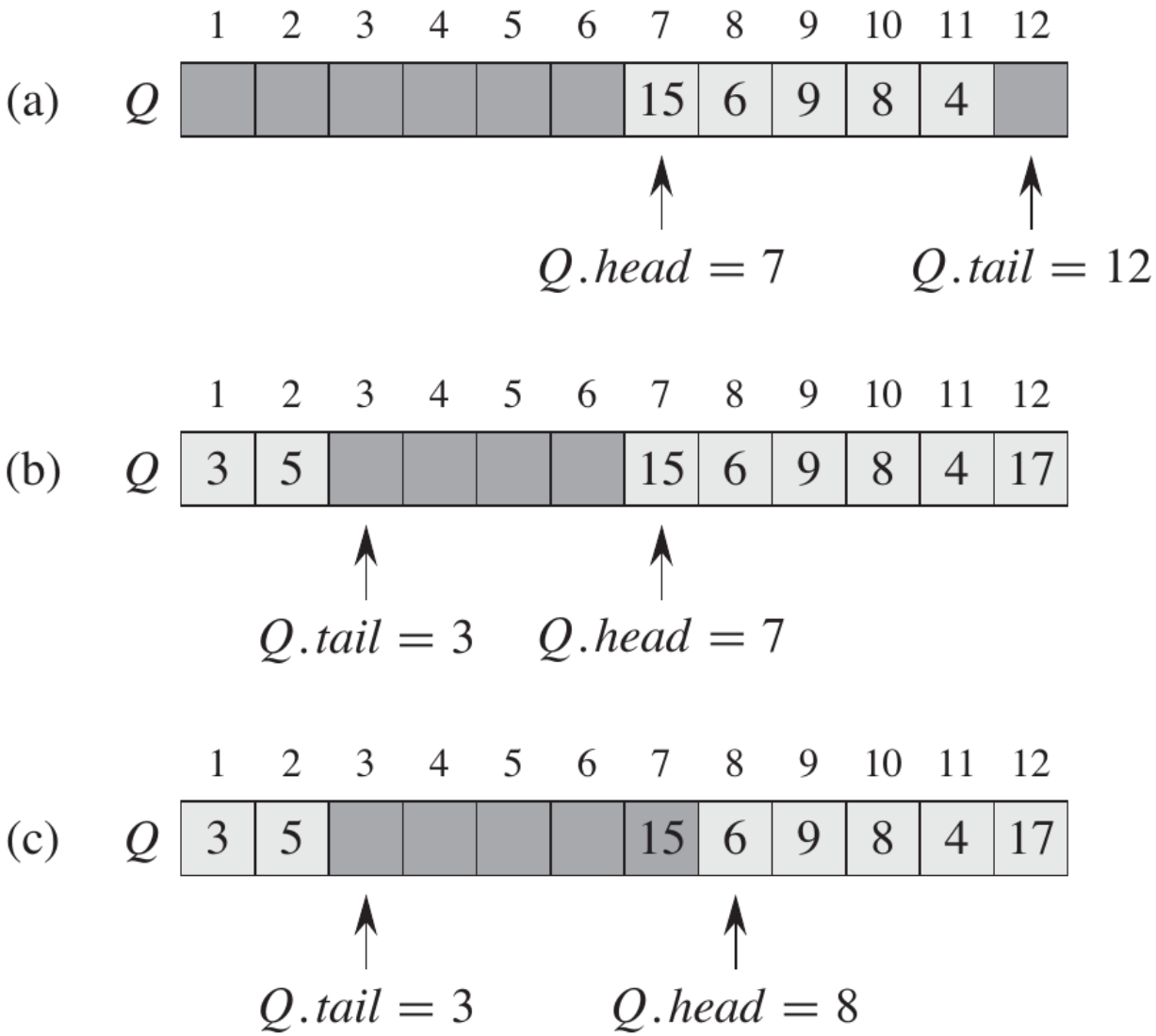
Queue array implementation



Queue array implementation



Queue array implementation



Enqueue and Dequeue

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.size
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

Enqueue and Dequeue

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.size
3        Q.tail = 1
4  else Q.tail = Q.tail + 1

DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.size
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Queue operations

- Each operation can be performed in $O(1)$ time.

Queue operations

- Each operation can be performed in $O(1)$ time.
- Use the `std::queue` container adapter or the `deque` template class.

Queue operations

- Each operation can be performed in $O(1)$ time.
- Use the `std::queue` container adapter or the `deque` template class.
- Queue can be used to perform Breadth-first-search in Graphs.

Minimum operations

- Starting from integer 1 we can use the following three operations to reach any positive integer:

Minimum operations

- Starting from integer 1 we can use the following three operations to reach any positive integer:
  1. Add one to the integer
  2. Multiply the integer by 2
  3. Multiply the integer by 10

Minimum operations

- Starting from integer 1 we can use the following three operations to reach any positive integer:
  1. Add one to the integer
  2. Multiply the integer by 2
  3. Multiply the integer by 10
- We need to find the *minimum* number of operations needed to reach integer x starting from integer 1.

Minimum operations

- Starting from integer 1 we can use the following three operations to reach any positive integer:
  1. Add one to the integer
  2. Multiply the integer by 2
  3. Multiply the integer by 10
- We need to find the *minimum* number of operations needed to reach integer x starting from integer 1.

  Input : 31

  Output : 4

  Explanation : $(((1 \times 2) + 1) \times 10) + 1 = 31$.

Linked lists

- A data structure in which objects are ordered linearly using a pointer.

Linked lists

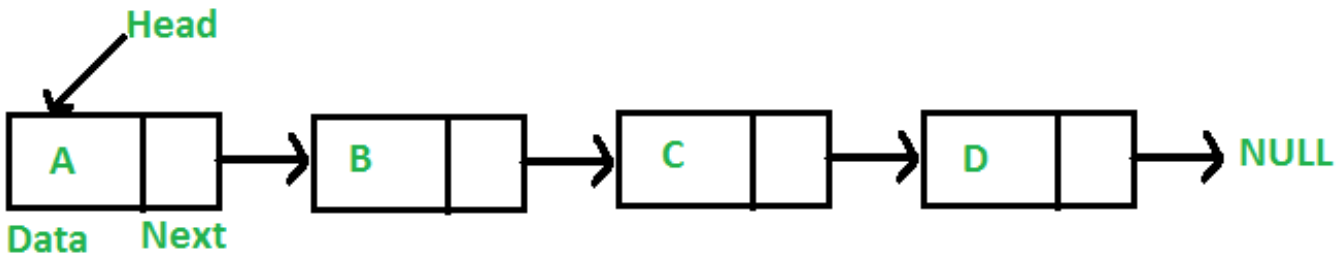
- A data structure in which objects are ordered linearly using a pointer.

```
// A linked list node
struct Node {
    int data;
    struct Node* next;
};
```

Linked lists

- A data structure in which objects are ordered linearly using a pointer.

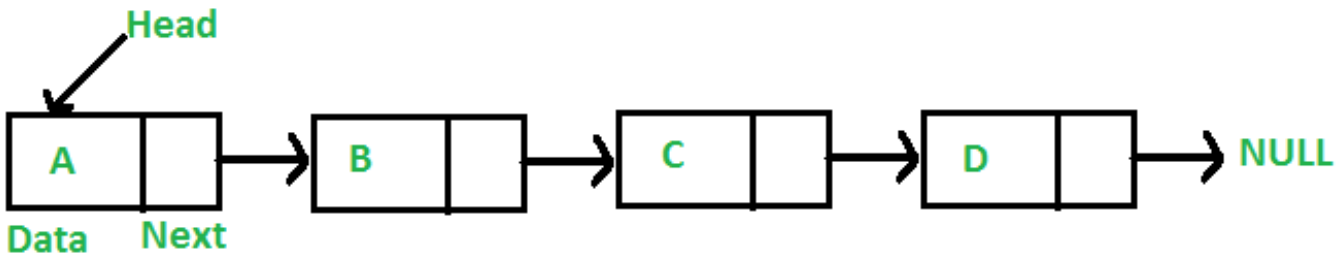
```
// A linked list node
struct Node {
    int data;
    struct Node* next;
};
```



Linked lists

- A data structure in which objects are ordered linearly using a pointer.

```
// A linked list node
struct Node {
    int data;
    struct Node* next;
};
```



- Linked list can support all the operations on a dynamic set.

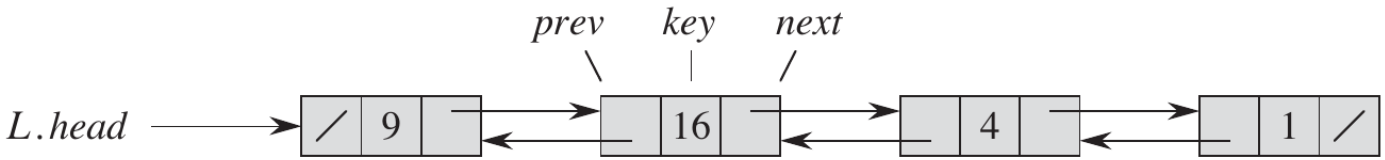
C++ program

- C++ program for singly linked list

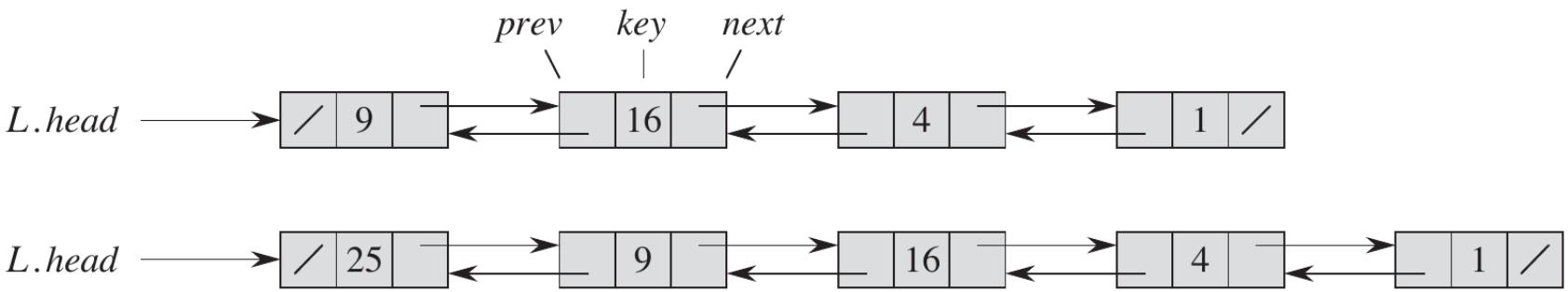
C++ program

- C++ program for singly linked list
- insertNode function (L. 109)

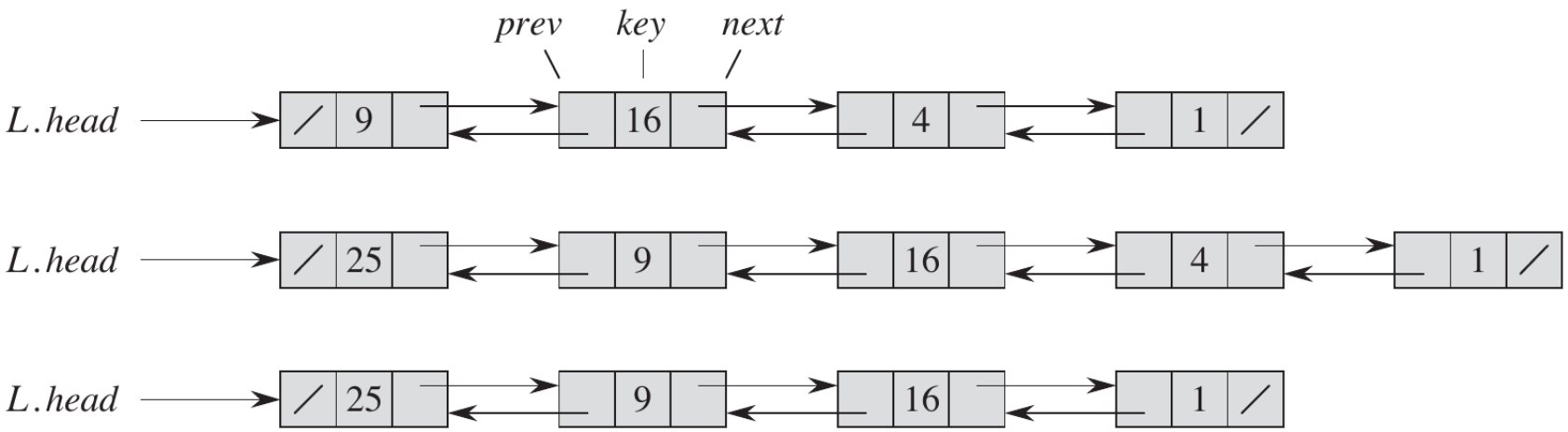
Doubly Linked List



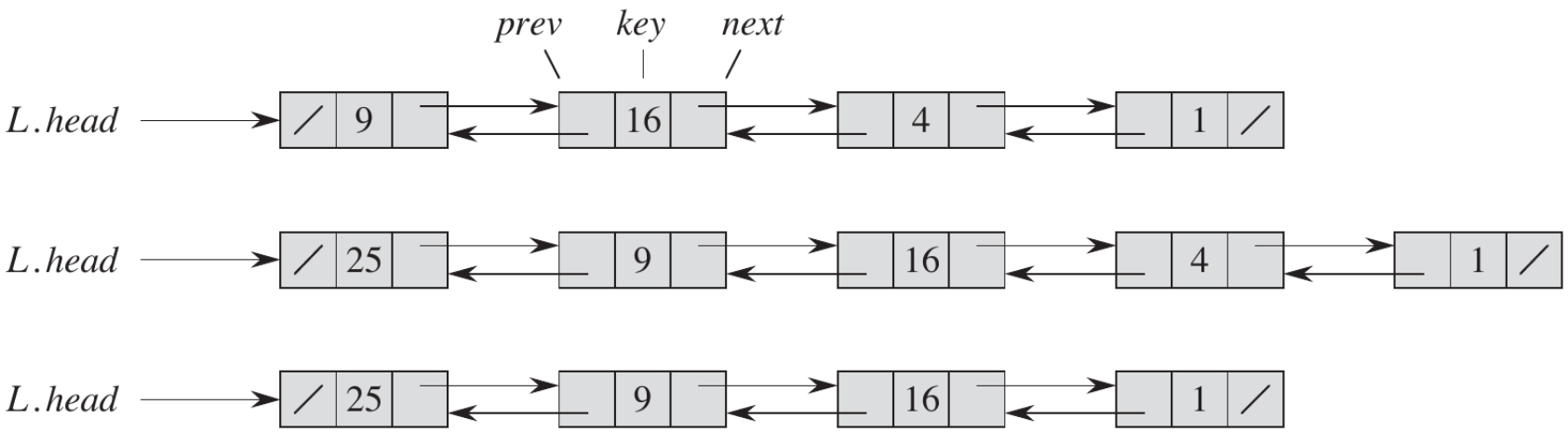
Doubly Linked List



Doubly Linked List

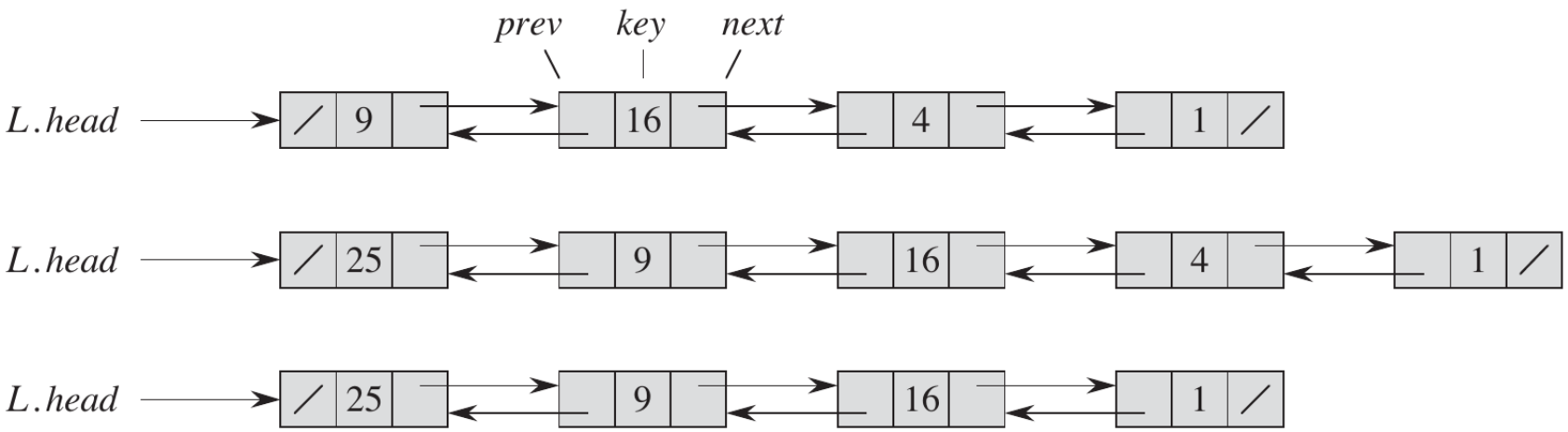


Doubly Linked List



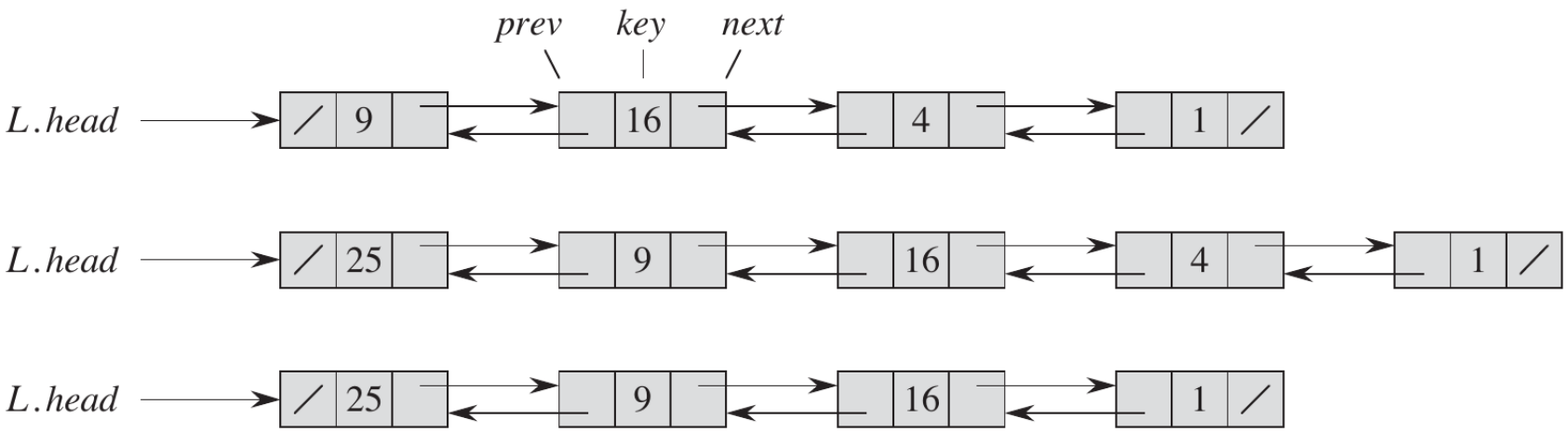
- NIL: prev of head element, and next of tail element

Doubly Linked List



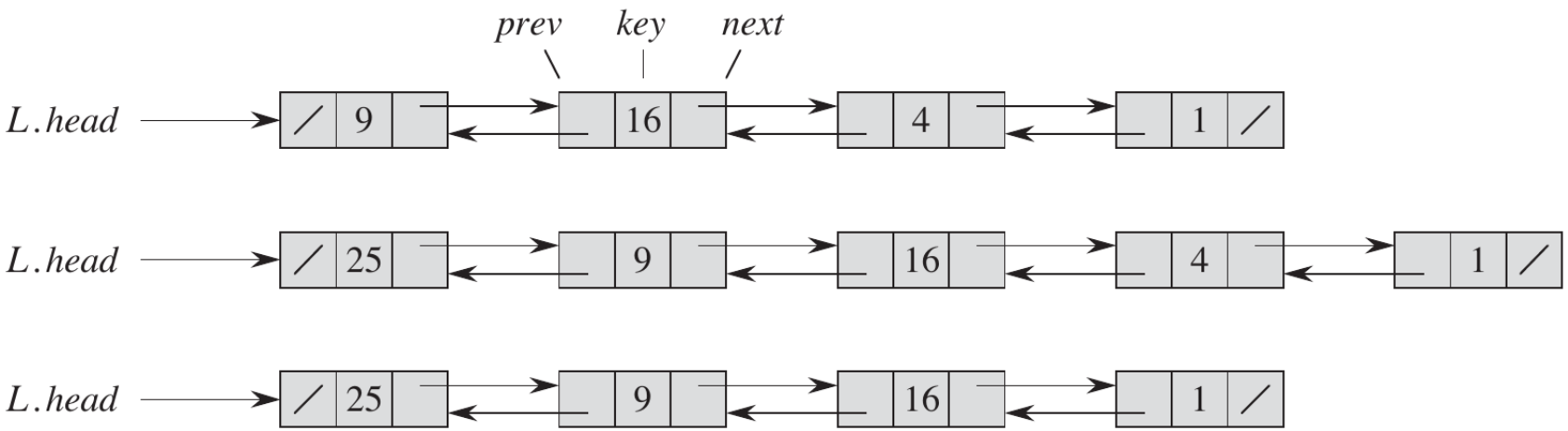
- NIL: prev of head element, and next of tail element
- Singly linked list does not have a prev pointer

Doubly Linked List



- NIL: prev of head element, and next of tail element
- Singly linked list does not have a prev pointer
- Circular linked list

Doubly Linked List



- NIL: prev of head element, and next of tail element
- Singly linked list does not have a prev pointer
- Circular linked list
- We will assume that we are working with an unsorted, doubly linked list.

List-Search

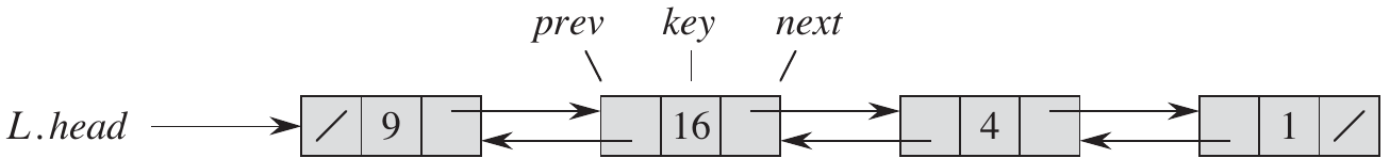
```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3        x = x.next
4  return x
```

List-Search

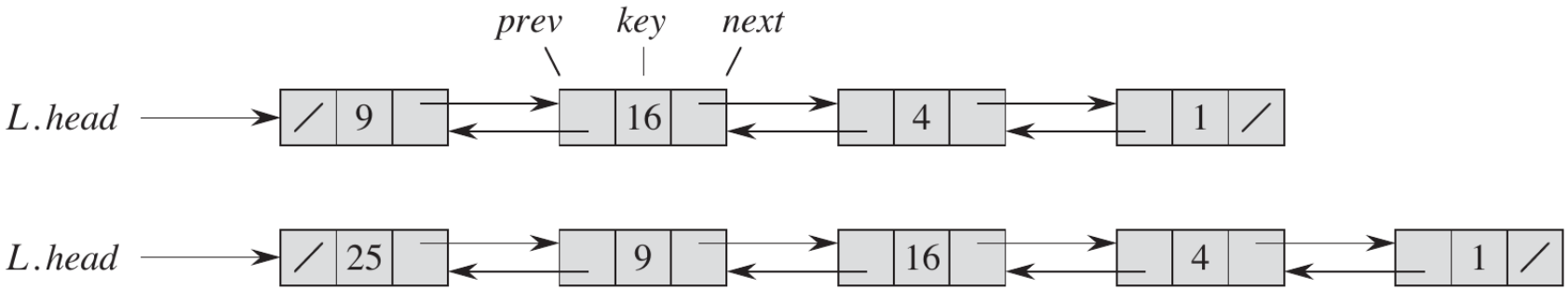
```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3        x = x.next
4  return x
```

- List-Search takes $\Theta(n)$ time in the worst case.

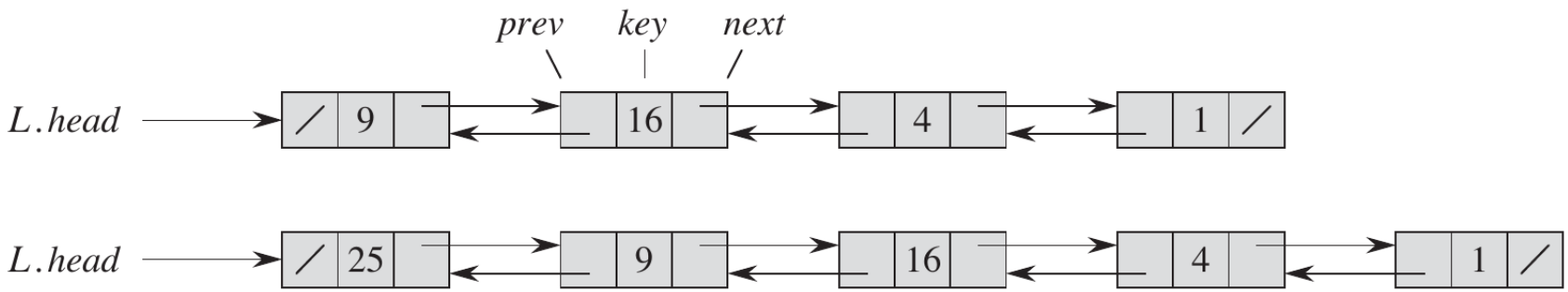
List-Prepend



List-Prepend

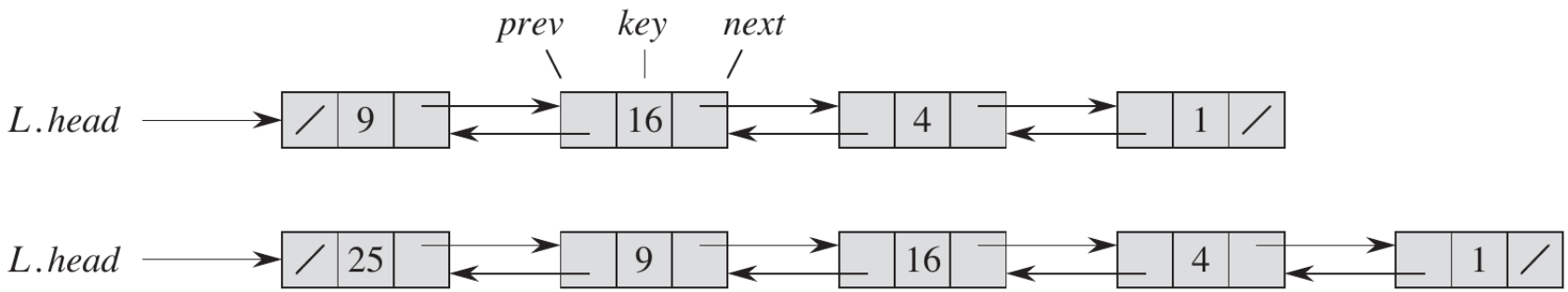


List-Prepend



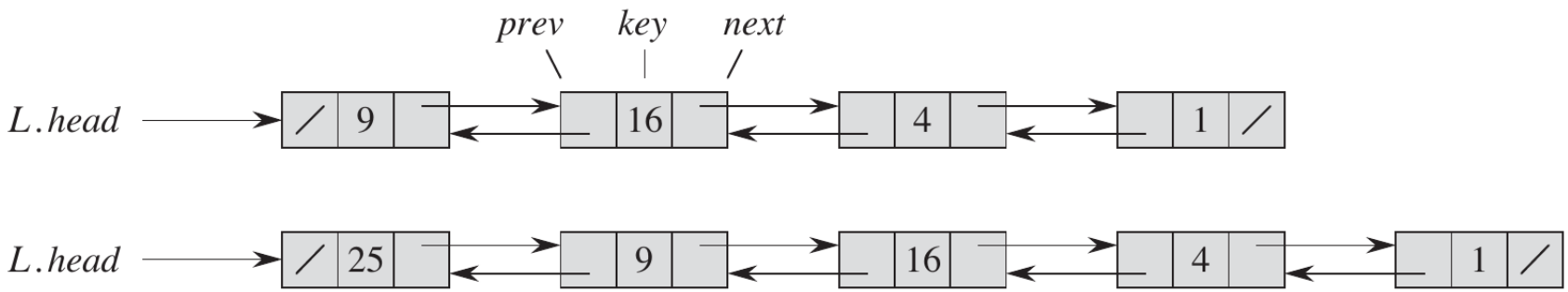

```
List-Prepend(L, x)
1   x.next = L.head
2   x.prev = NIL
3   if L.head ≠ NIL
4         L.head.prev = _____
5   L.head = _____
```

List-Prepend




```
List-Prepend(L, x)
1   x.next = L.head
2   x.prev = NIL
3   if L.head ≠ NIL
4       L.head.prev = x
5   L.head = x
```

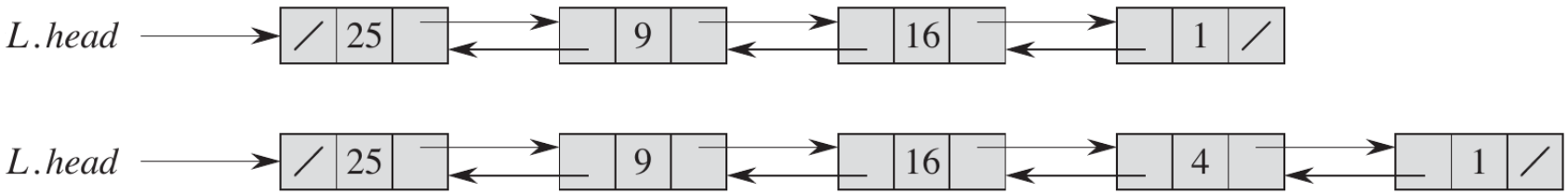
List-Prepend



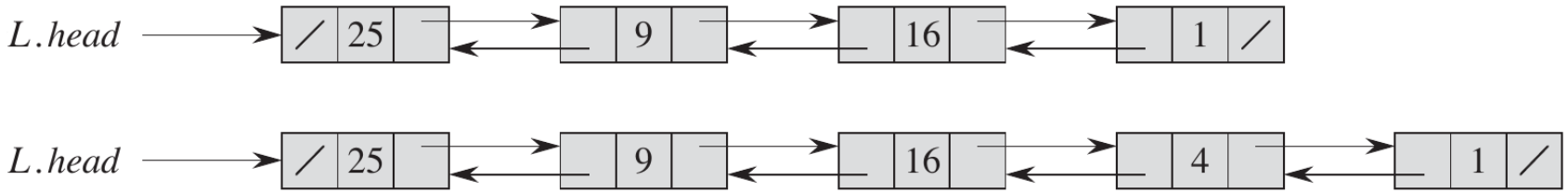

```
List-Prepend(L, x)
1   x.next = L.head
2   x.prev = NIL
3   if L.head ≠ NIL
4         L.head.prev = x
5   L.head = x
```

List-Prepend takes $O(1)$ time

List-Insert



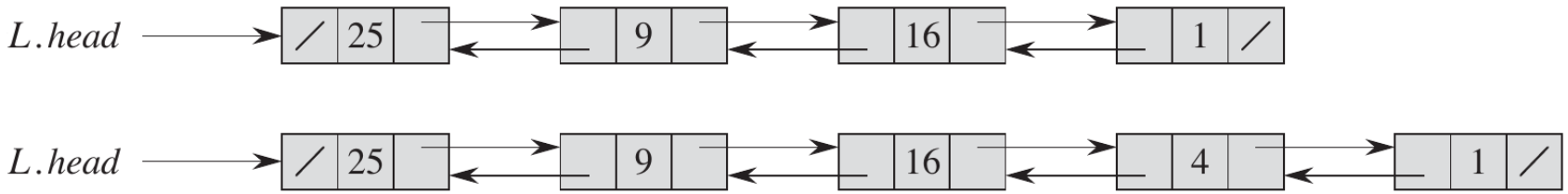
List-Insert



Insert node x after node y

```
LIST-INSERT(x, y)
1   x.next = y.next
2   x.prev = y
3   if y.next ≠ NIL
4         y.next.prev = _____
5   y.next = _____
```

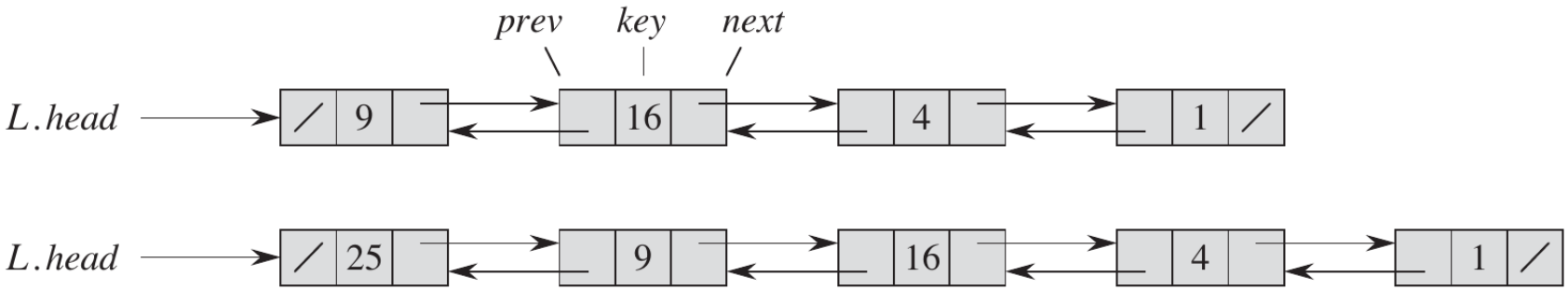
List-Insert



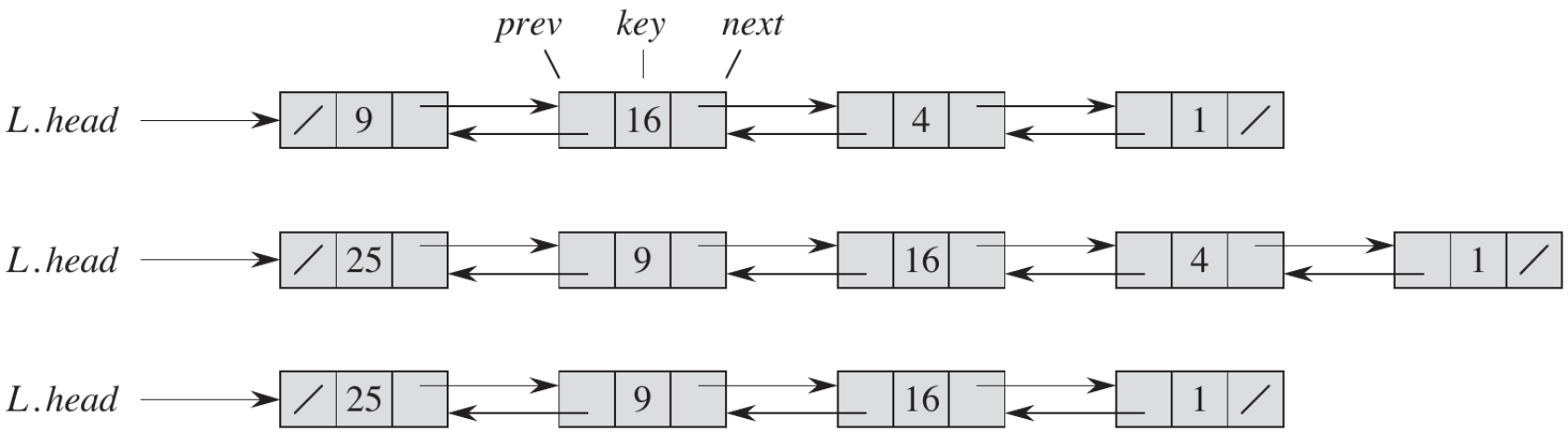
Insert node x after node y

```
LIST-INSERT(x, y)
1   x.next = y.next
2   x.prev = y
3   if y.next ≠ NIL
4         y.next.prev = x
5   y.next = x
```

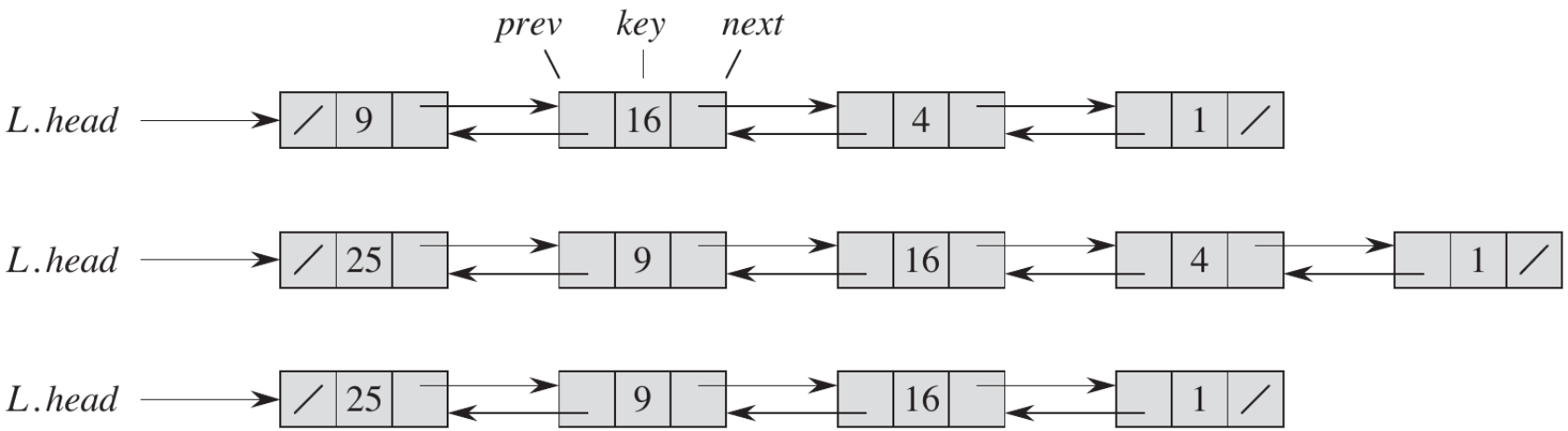
List-Delete



List-Delete



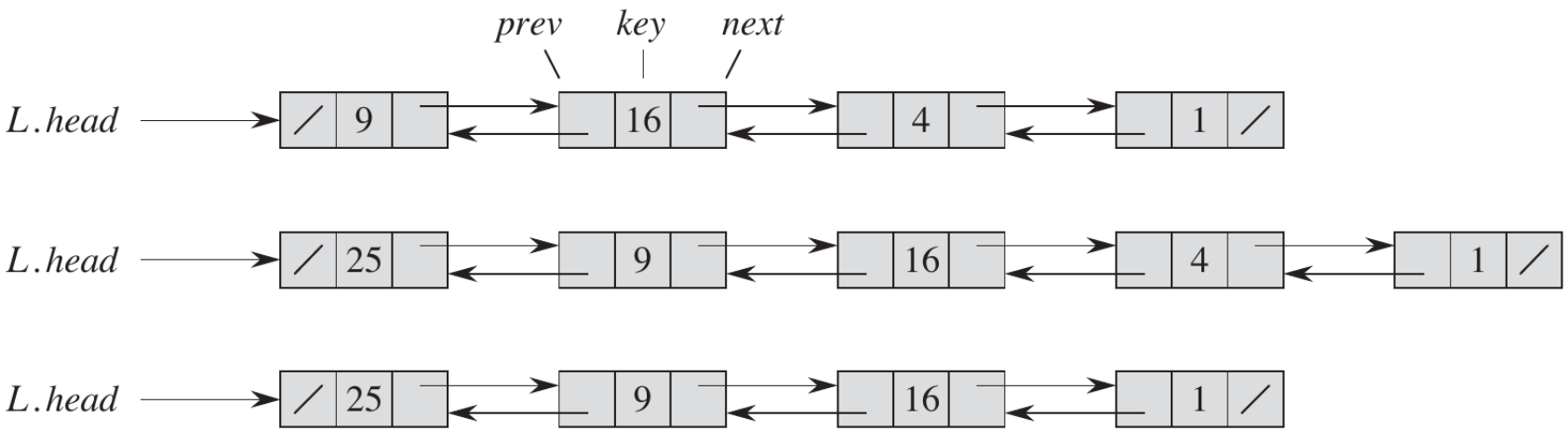
List-Delete



List-Delete(L, x)

1 **if** $x.prev \neq$ NIL
2 $\quad x.prev.next =$ _____
3 **else** $L.head = x.next$
4 **if** $x.next \neq$ NIL
5 $\quad x.next.prev =$ _____

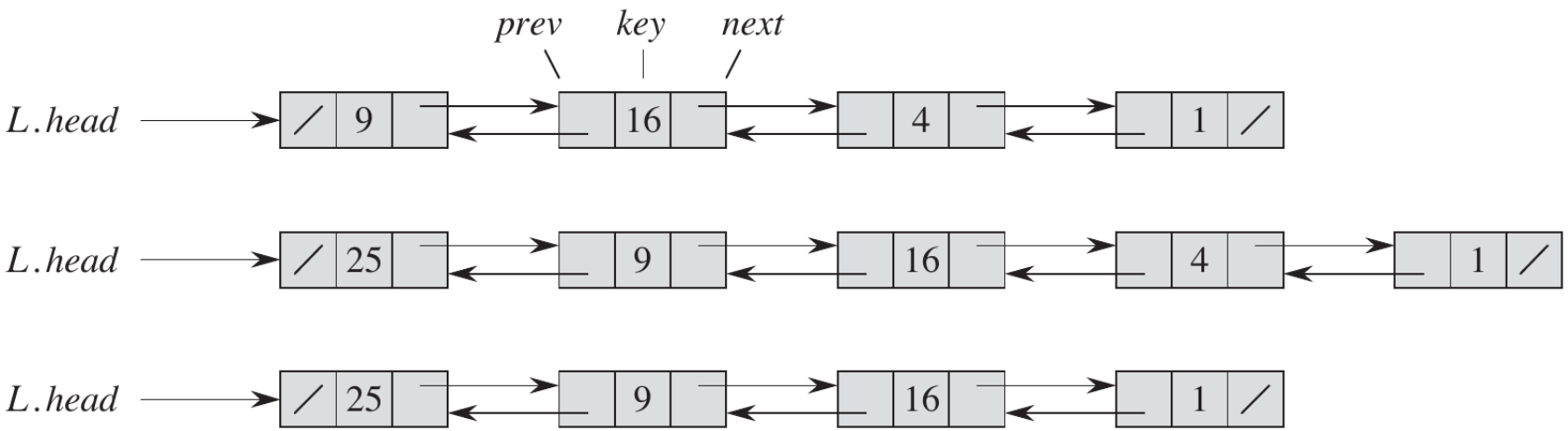
List-Delete



List-Delete(L, x)

```
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

List-Delete

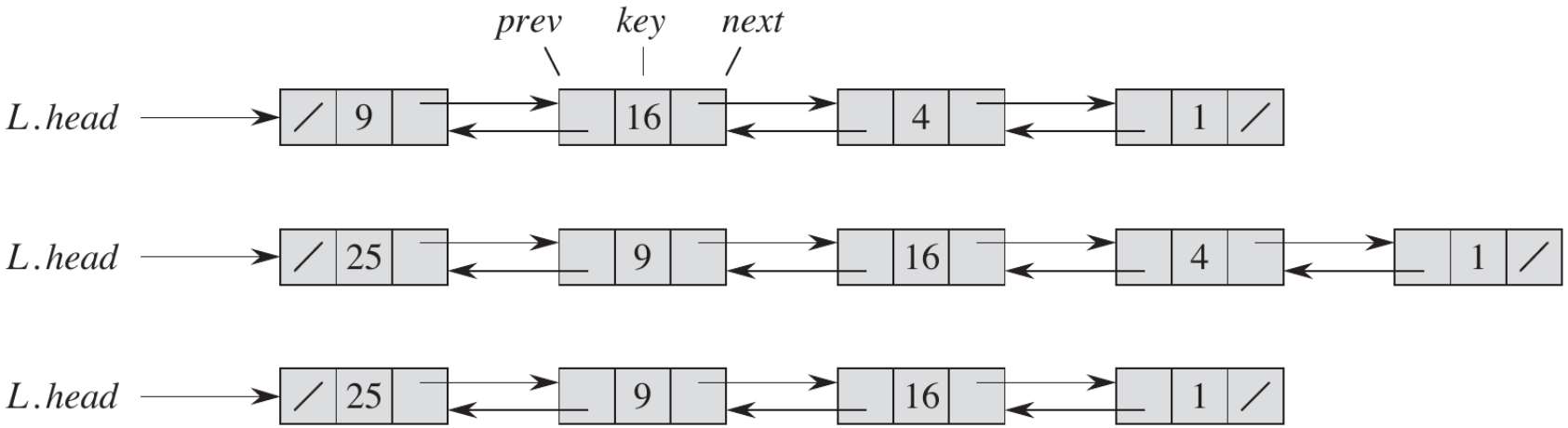


List-Delete(L, x)

```
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

List-Delete takes $O(1)$ time.

List-Delete



List-Delete(L, x)

```
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

List-Delete takes $O(1)$ time. Deleting an element with a given key would take $\Theta(n)$ time.