Algorithms and Data Structures 2

10 – Linked lists

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

Outline

- **Dynamic data structures**
- **Singly linked lists**
  - Insertion
  - Deletion
  - Search
  - Recursion
  - MERGE-SORT
- **Doubly linked lists**
  - Operations
- **Circular doubly linked list with a sentinel**
  - Operations

Dynamic data structures

- **Data collections can vary considerably in size at runtime**

- **Disadvantages of arrays**
  - Inserting/deleting a new element requires much of array to be rewritten
  - Array size is fixed, must be estimated before use
  - If only few items held, much of array (hence memory) is wasted

- **Using dynamic data structures (linked data structures)**
  - We don’t need to know how many items to expect
  - We can increase/decrease memory when items added/deleted

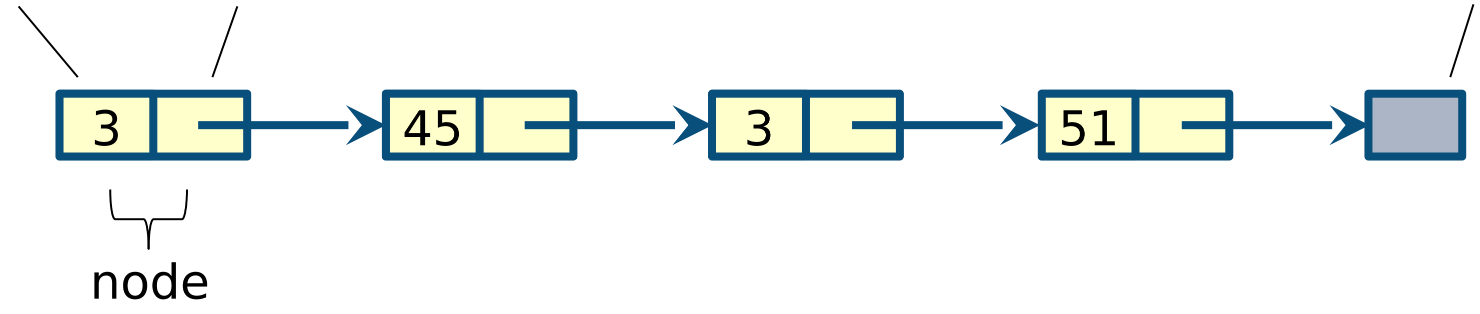
Examples

- **Linked lists**
  - Singly linked lists
  - Doubly linked lists
  - Circular lists
- **Trees**
  - Binary search trees
  - AVL trees
  - Red-black trees
  - B-trees
  - Binomial heaps
  - Fibonacci heaps

Singly linked list

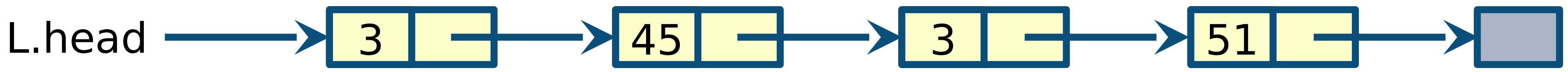
- **Dynamic data structure consisting of a sequence of nodes arranged in a linear order**
  - The order in a linked list is determined by a pointer in each object

- **Each node (or element) of a (singly) linked list L has an attribute key and a pointer attribute next**
  - Given a node x in the list, x.next points to its successor in the linked list
  - If x.next = NIL, x has no successor and is therefore the last element, or tail, of the list



Head pointer

- **An attribute L.head points to the first element of list L**



- **If L.head = NIL, the list is empty**

Insertion

- **Insertion at the head**
  - Allocate a new node
  - Update two pointers
- **Complexity O(1)**

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

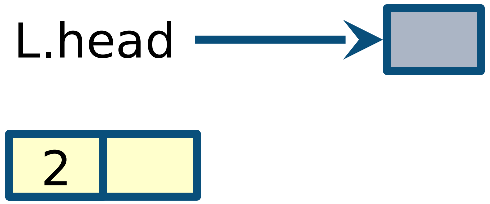
- **Example**
  - Add nodes with keys 2 and 3 (in this order) to an empty list

Insertion

- **Insertion at the head**
  - Allocate a new node
  - Update two pointers
- **Complexity O(1)**

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- **Example**
  - Add nodes with keys 2 and 3 (in this order) to an empty list



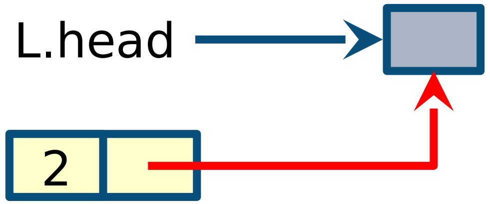
  - Allocate new node for INSERT(L,2)

Insertion

- **Insertion at the head**
  - Allocate a new node
  - Update two pointers
- **Complexity O(1)**

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- **Example**
  - Add nodes with keys 2 and 3 (in this order) to an empty list



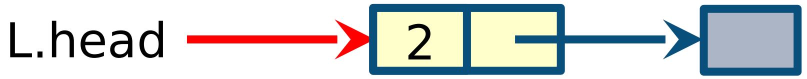
  - Update next

Insertion

- **Insertion at the head**
  - Allocate a new node
  - Update two pointers
- **Complexity O(1)**

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- **Example**
  - Add nodes with keys 2 and 3 (in this order) to an empty list



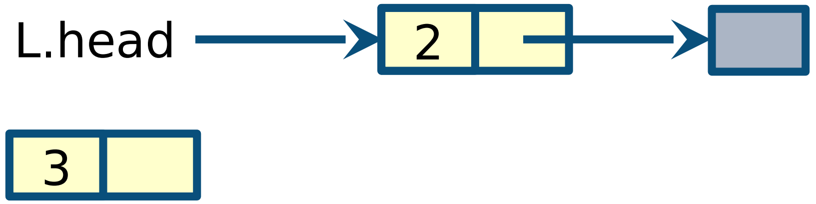
  - Update head

Insertion

- **Insertion at the head**
  - Allocate a new node
  - Update two pointers
- **Complexity O(1)**

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- **Example**
  - Add nodes with keys 2 and 3 (in this order) to an empty list



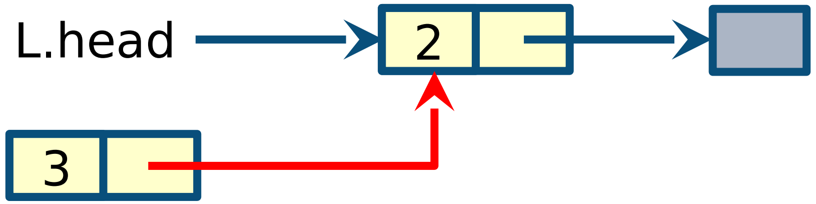
  - Allocate new node for INSERT(L,3)

Insertion

- **Insertion at the head**
  - Allocate a new node
  - Update two pointers
- **Complexity O(1)**

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- **Example**
  - Add nodes with keys 2 and 3 (in this order) to an empty list



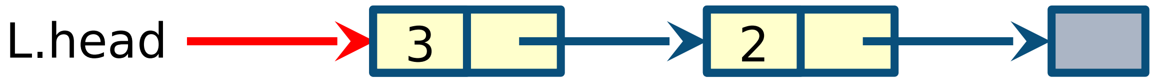
  - Update next

Insertion

- **Insertion at the head**
  - Allocate a new node
  - Update two pointers
- **Complexity O(1)**

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- **Example**
  - Add nodes with keys 2 and 3 (in this order) to an empty list



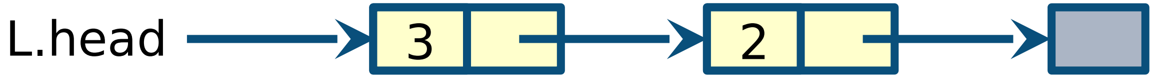
  - Update head

Deletion

- **Deletion at the head**
  - Update L.head
  - Deallocate memory of node being deleted
- **Deallocation is performed by the garbage collector in Java**

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- **Complexity O(1)**
- **Example**
  - Call DELETE three times on the list below

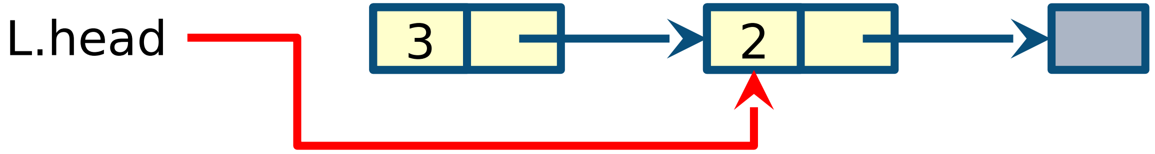


Deletion

- **Deletion at the head**
  - Update L.head
  - Deallocate memory of node being deleted
- **Deallocation is performed by the garbage collector in Java**

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- **Complexity O(1)**
- **Example**
  - Call DELETE three times on the list below



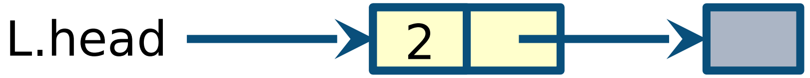
1) L.head is updated

Deletion

- **Deletion at the head**
  - Update L.head
  - Deallocate memory of node being deleted
- **Deallocation is performed by the garbage collector in Java**

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- **Complexity O(1)**
- **Example**
  - Call DELETE three times on the list below



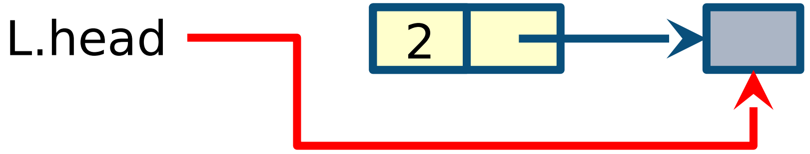
1) 3 is garbage collected

Deletion

- **Deletion at the head**
  - Update L.head
  - Deallocate memory of node being deleted
- **Deallocation is performed by the garbage collector in Java**

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- **Complexity O(1)**
- **Example**
  - Call DELETE three times on the list below



2) L.head is updated

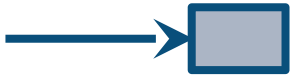
Deletion

- **Deletion at the head**
  - Update L.head
  - Deallocate memory of node being deleted
- **Deallocation is performed by the garbage collector in Java**

- **Complexity O(1)**

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- **Example**
  - Call DELETE three times on the list below

L.head 

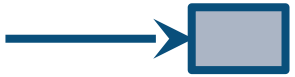
2) 2 is garbage collected

Deletion

- **Deletion at the head**
  - Update L.head
  - Deallocate memory of node being deleted
- **Deallocation is performed by the garbage collector in Java**

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- **Complexity O(1)**
- **Example**
  - Call DELETE three times on the list below

L.head 

3) L.head = NIL

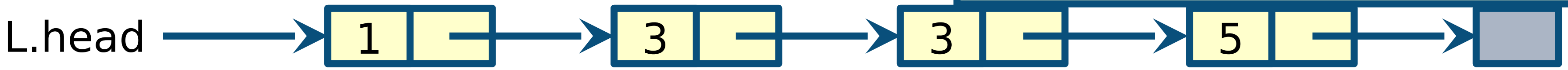
Search

- **Find the first element with key k in list L by a simple linear search**
  - If found, return a pointer to this element
  - If no object with key k appears in the list, then return NIL
- **Iterator pattern**

```
SEARCH(L,k)
  x := L.head
  while x != NIL and x.key != k

    x := x.next
  return x
```

- **Complexity O(n)**
- **Example**
  - Find k=3 in the list below



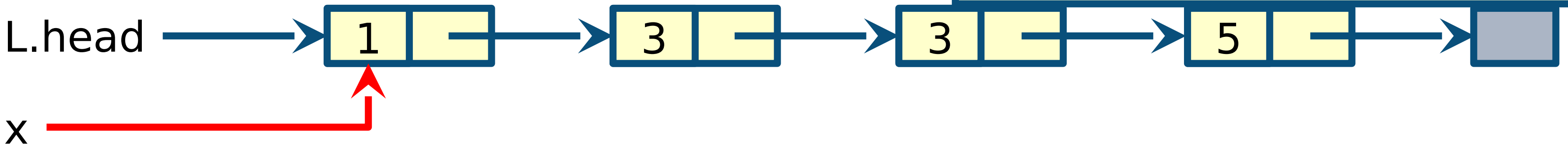
Search

- **Find the first element with key k in list L by a simple linear search**
  - If found, return a pointer to this element
  - If no object with key k appears in the list, then return NIL
- **Iterator pattern**

```
SEARCH(L,k)
  x := L.head
  while x != NIL and x.key != k

    x := x.next
  return x
```

- **Complexity O(n)**
- **Example**
  - Find k=3 in the list below



  - Initialise cursor x

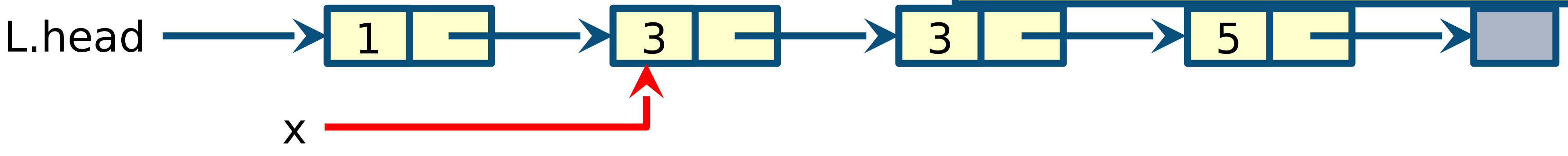
Search

- **Find the first element with key k in list L by a simple linear search**
  - If found, return a pointer to this element
  - If no object with key k appears in the list, then return NIL
- **Iterator pattern**

```
SEARCH(L,k)
  x := L.head
  while x != NIL and x.key != k

    x := x.next
  return x
```

- **Complexity O(n)**
- **Example**
  - Find k=3 in the list below



  - 1!= 3 then we update cursor x

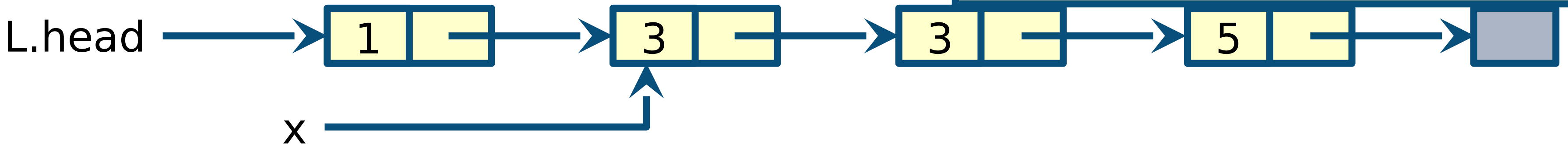
Search

- **Find the first element with key k in list L by a simple linear search**
  - If found, return a pointer to this element
  - If no object with key k appears in the list, then return NIL
- **Iterator pattern**

```
SEARCH(L,k)
  x := L.head
  while x != NIL and x.key != k

    x := x.next
  return x
```

- **Complexity O(n)**
- **Example**
  - Find k=3 in the list below



  - 3=3 then we return cursor x

Recursion on linked lists

- **The structure of linked lists is inherently recursive**
  - Iterators can be implemented by recursive algorithms
- **Example: recursive search**

```
REC-SEARCH(x,k)
  if x = NIL
    return x
  elseif x.key = k
    return x
  else
    return REC-SEARCH(x.next,k)
```

MERGE-SORT for linked lists

- **Sorting algorithm of choice as as it does not heavily rely on fast random access**

- **Recall the divide and conquer strategy to define MERGE-SORT(L)**
  1. If L.head = NIL or there is only one element in L then return
  2. Otherwise, divide L into two halves with SPLIT
  3. Sort the two halves recursively
  4. Merge the sorted halves with MERGE

- **We begin by defining MERGE and SPLIT**

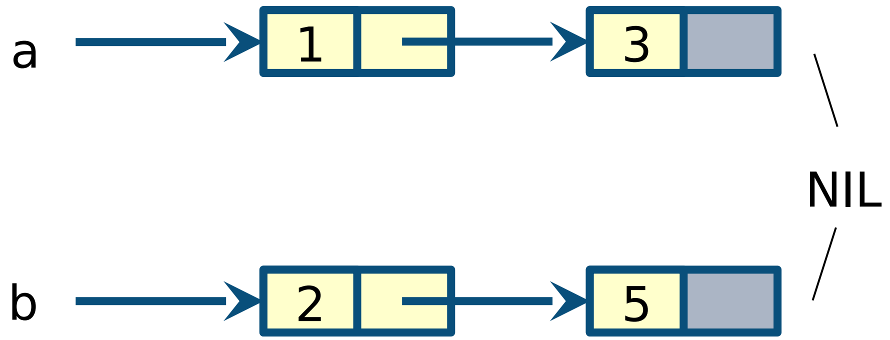
MERGE

- **Input: two sorted linked lists a and b**
- **Output: a sorted linked list with the elements of a and b**

- **Only pointer manipulations**
  - Data is not copied
  - Attribute head is not used

```
MERGE(a,b)
  if a = NIL
    return b
  elseif b = NIL
    return a
  x := NIL
  if a.key ≤ b.key
    x := a
    x.next := MERGE(a.next,b)
  else
    x := b
    x.next := MERGE(a,b.next)
  return x
```

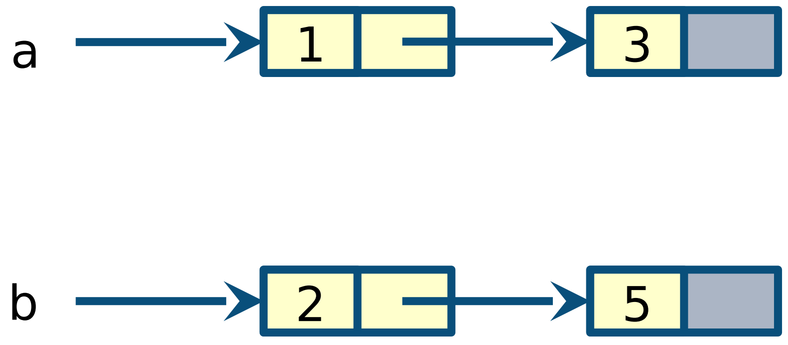
Try MERGE(a,b)




```
MERGE(a,b)
  if a = NIL
    return b
  elseif b = NIL
    return a
  x := NIL
  if a.key ≤ b.key
    x := a
    x.next := MERGE(a.next,b)
  else
    x := b
    x.next := MERGE(a,b.next)
  return x
```

- **Similar to MERGE for arrays**

Try MERGE(a,b)




```
MERGE(a,b)
  if a = NIL
    return b
  elseif b = NIL
    return a
  x := NIL
  if a.key ≤ b.key
    x := a
    x.next := MERGE(a.next,b)
  else
    x := b
    x.next := MERGE(a,b.next)
  return x
```

- **Recursive calls (sketch)**
  - MERGE(a,b) return [1,2,3,5]
  - x=1, MERGE(a.next,b) return [2,3,5]
  - x=2, MERGE(a,b.next) return [3,5]
  - x=3, MERGE(a.next,b) return [5]

SPLIT

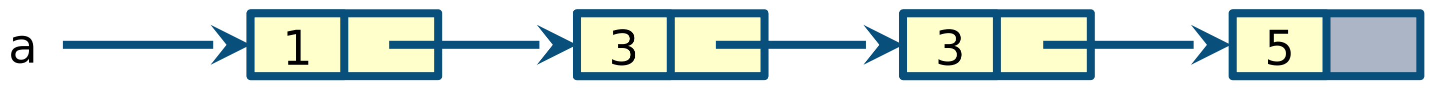
- **Input: list a**
- **Output: the two halves of a**

- **We use two cursors to find the middle element**
  - Slow advances one node per iterations
  - Fast advances two nodes per iteration
  - At the end of the loop, slow is before the midpoint

```
SPLIT(a)
  if a = NIL or a.next = NIL
    return (a,NIL)     // two values
  slow := a
  fast := a.next
  while fast != NIL and fast.next != NIL
    slow := slow.next
    fast := fast.next.next
  mid := slow.next
  slow.next := NIL
  return (a,mid)
```

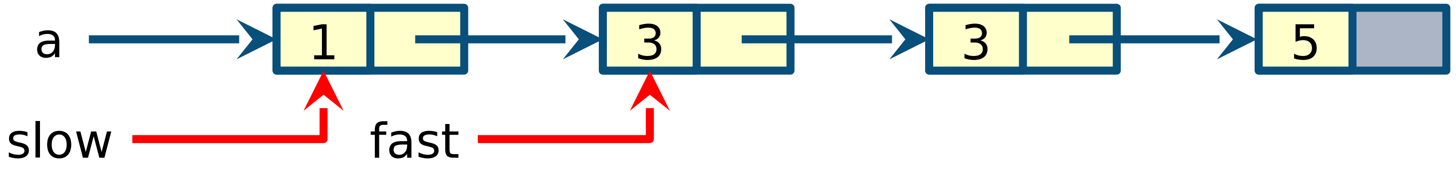
Example execution of SPLIT

```
SPLIT(a)
  if a = NIL or a.next = NIL
    return (a,NIL)     // two values
  slow := a
  fast := a.next
  while fast != NIL and fast.next != NIL
    slow := slow.next
    fast := fast.next.next
  mid := slow.next
  slow.next := NIL
  return (a,mid)
```



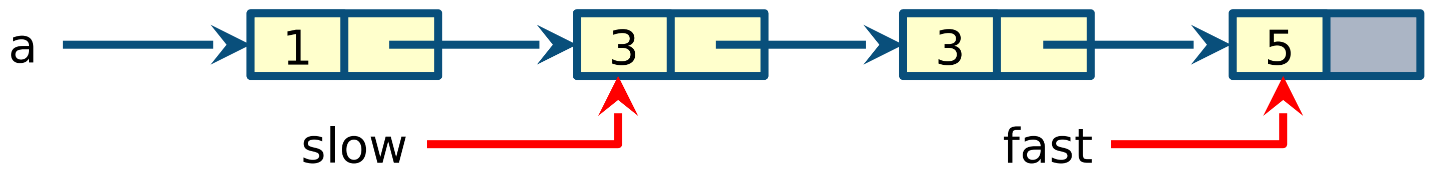
Example execution of SPLIT

```
SPLIT(a)
  if a = NIL or a.next = NIL
    return (a,NIL)     // two values
  slow := a
  fast := a.next
  while fast != NIL and fast.next != NIL
    slow := slow.next
    fast := fast.next.next
  mid := slow.next
  slow.next := NIL
  return (a,mid)
```



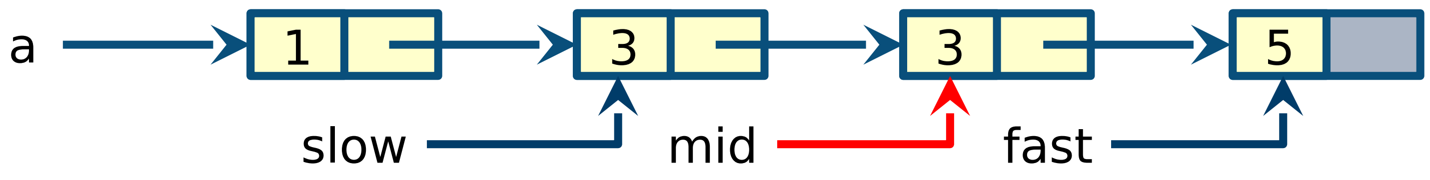
Example execution of SPLIT

```
SPLIT(a)
  if a = NIL or a.next = NIL
    return (a,NIL)     // two values
  slow := a
  fast := a.next
  while fast != NIL and fast.next != NIL
    slow := slow.next
    fast := fast.next.next
  mid := slow.next
  slow.next := NIL
  return (a,mid)
```



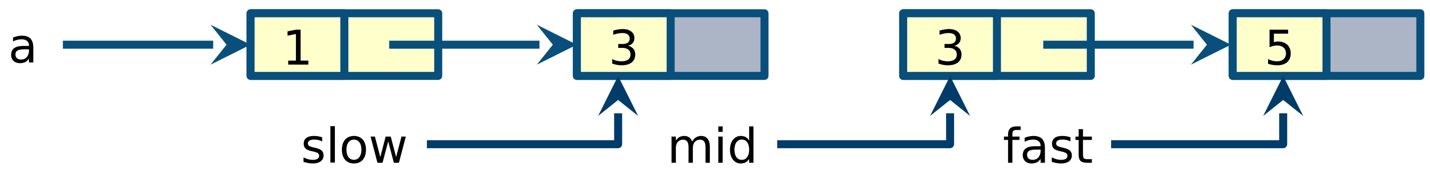
Example execution of SPLIT

```
SPLIT(a)
  if a = NIL or a.next = NIL
    return (a,NIL)     // two values
  slow := a
  fast := a.next
  while fast != NIL and fast.next != NIL
    slow := slow.next
    fast := fast.next.next
  mid := slow.next
  slow.next := NIL
  return (a,mid)
```



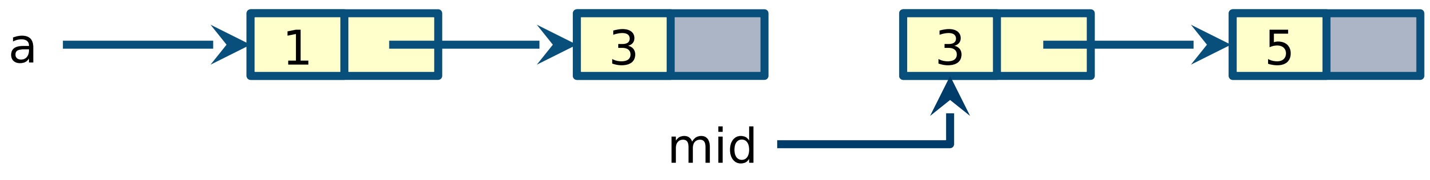
Example execution of SPLIT

```
SPLIT(a)
  if a = NIL or a.next = NIL
    return (a,NIL)     // two values
  slow := a
  fast := a.next
  while fast != NIL and fast.next != NIL
    slow := slow.next
    fast := fast.next.next
  mid := slow.next
  slow.next := NIL
  return (a,mid)
```



Example execution of SPLIT

```
SPLIT(a)
  if a = NIL or a.next = NIL
    return (a,NIL)     // two values
  slow := a
  fast := a.next
  while fast != NIL and fast.next != NIL
    slow := slow.next
    fast := fast.next.next
  mid := slow.next
  slow.next := NIL
  return (a,mid)
```



MERGE-SORT

- **Input: linked lists a**
- **Output: a sorted**

```
MERGE-SORT(a)
  if a = NIL or a.next = NIL
    return a
  (l,r) := SPLIT(x)
  x := MERGE-SORT(l)
  y := MERGE-SORT(r)
  return MERGE(x,y)
```

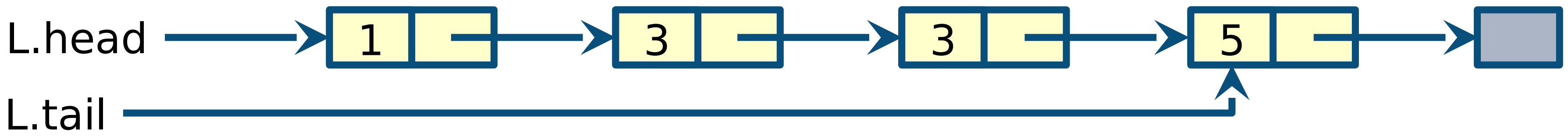
Insertion at the tail

- **We saw that insertion at the head is O(1)**
- **Insertion at the tail requires to scan the entire list**
  - Complexity O(n)
  - Iterator pattern
- **This is a problem when we use linked lists to implement queues which require fast insertions at the tail for the ENQUEUE operation**
  - We will study queues in the next lectures

```
INSERT-TAIL`(L,x)
  if L.head = NIL
    INSERT(L,x)
  else
    y := L.head
    while y.next != NIL
      y := y.next
    x.next := NIL
    y.next := x
```

Tail pointer

- **The definition of linked list is extended to include an attribute L.tail pointing to the last element of list L**



- **If L.head = L.tail = NIL, the list is empty**

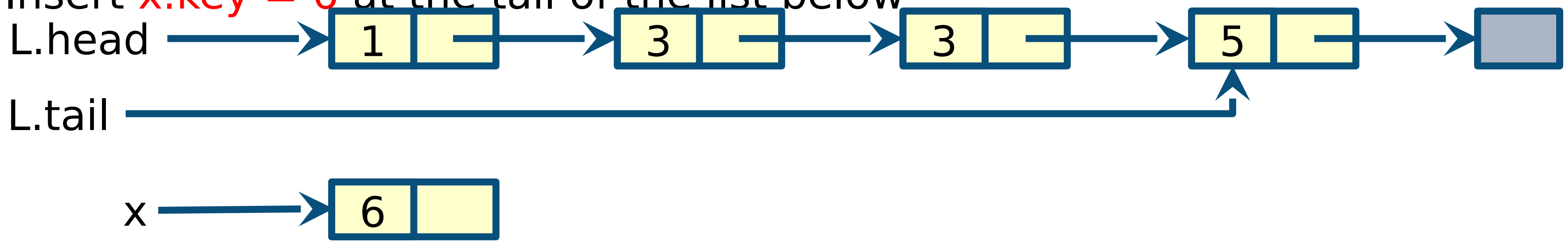
- **INSERT and DELETE have to be redefined to update L.tail when needed**

Insertion at the tail with tail pointer

- **Can be performed in constant time**

```
INSERT-TAIL(L,x)
  x.next := NIL
  if L.tail = NIL
    L.head := x
  else
    L.tail.next := x
  L.tail := x
```

- **Example**
  - Insert x.key = 6 at the tail of the list below

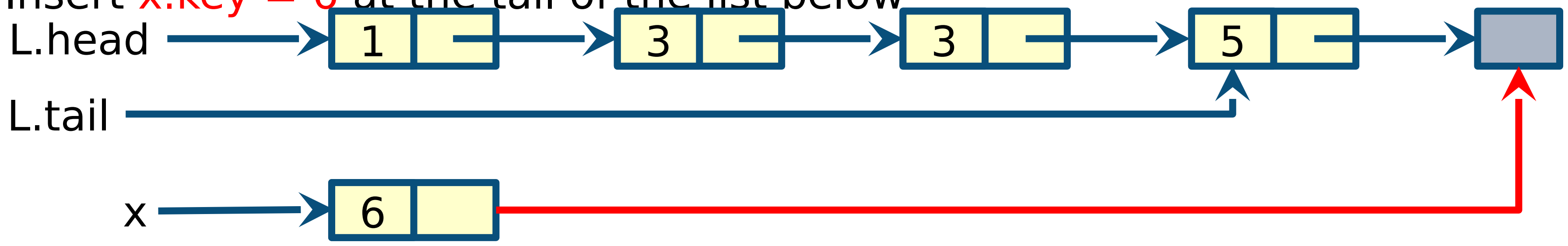


Insertion at the tail with tail pointer

- **Can be performed in constant time**

```
INSERT-TAIL(L,x)
  x.next := NIL
  if L.tail = NIL
    L.head := x
  else
    L.tail.next := x
  L.tail := x
```

- **Example**
  - Insert x.key = 6 at the tail of the list below



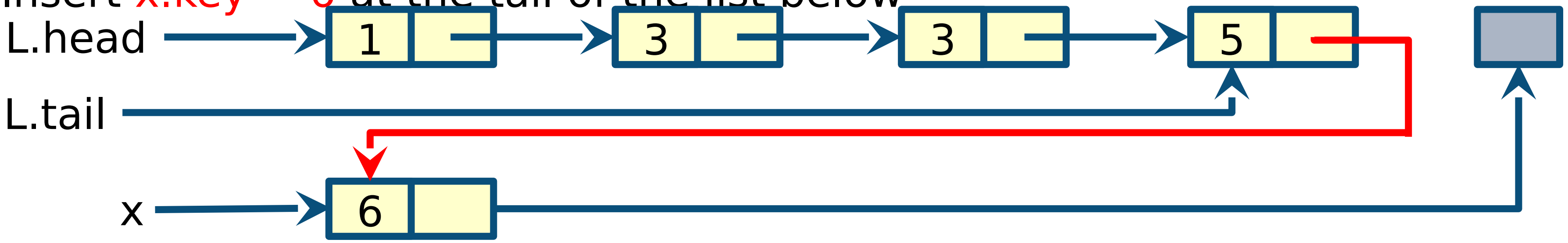
  - Update x.next

Insertion at the tail with tail pointer

- **Can be performed in constant time**

```
INSERT-TAIL(L,x)
  x.next := NIL
  if L.tail = NIL
    L.head := x
  else
    L.tail.next := x
  L.tail := x
```

- **Example**
  - Insert x.key = 6 at the tail of the list below



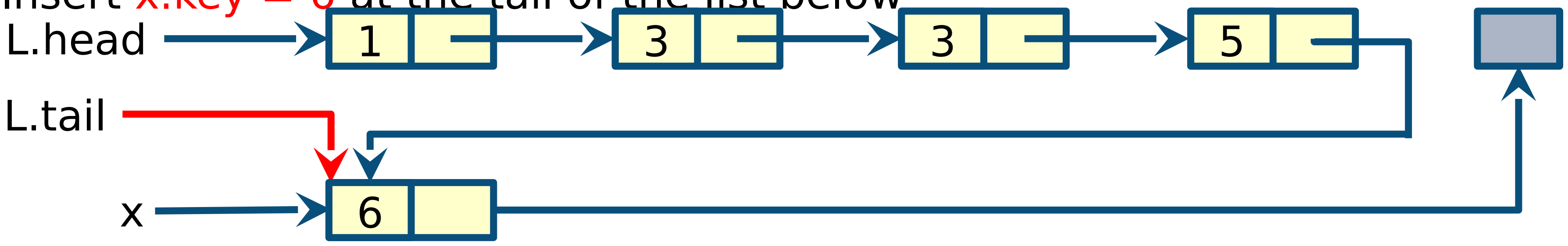
  - Update L.tail.next

Insertion at the tail with tail pointer

- Can be performed in constant time

```
INSERT-TAIL(L,x)
  x.next := NIL
  if L.tail = NIL
    L.head := x
  else
    L.tail.next := x
  L.tail := x
```

- Example
  - Insert x.key = 6 at the tail of the list below



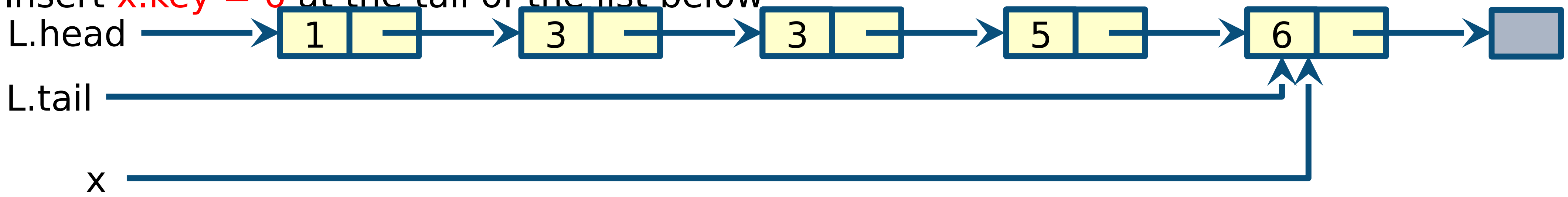
  - Update L.tail

Insertion at the tail with tail pointer

- **Can be performed in constant time**

```
INSERT-TAIL(L,x)
  x.next := NIL
  if L.tail = NIL
    L.head := x
  else
    L.tail.next := x
  L.tail := x
```

- **Example**
  - Insert x.key = 6 at the tail of the list below

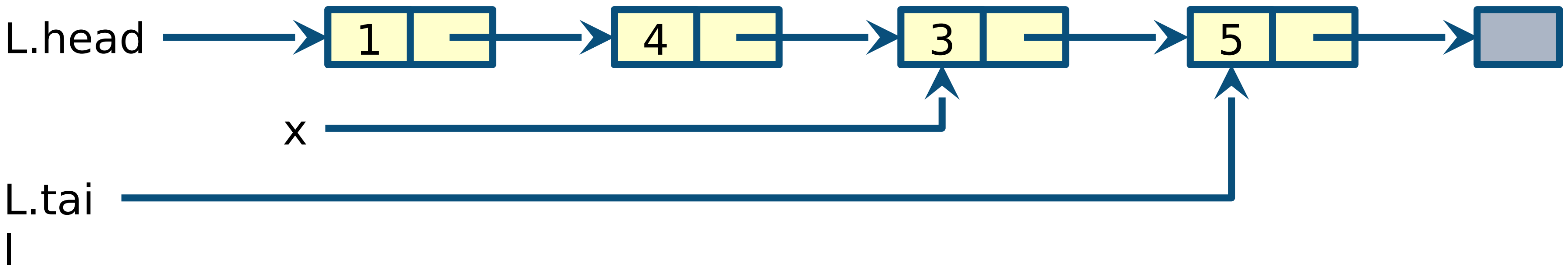


– Termination

Deletion of an element

- **Remove an element x from a linked list L**
  - Pointer to x must be retrieved first (for instance by calling SEARCH)

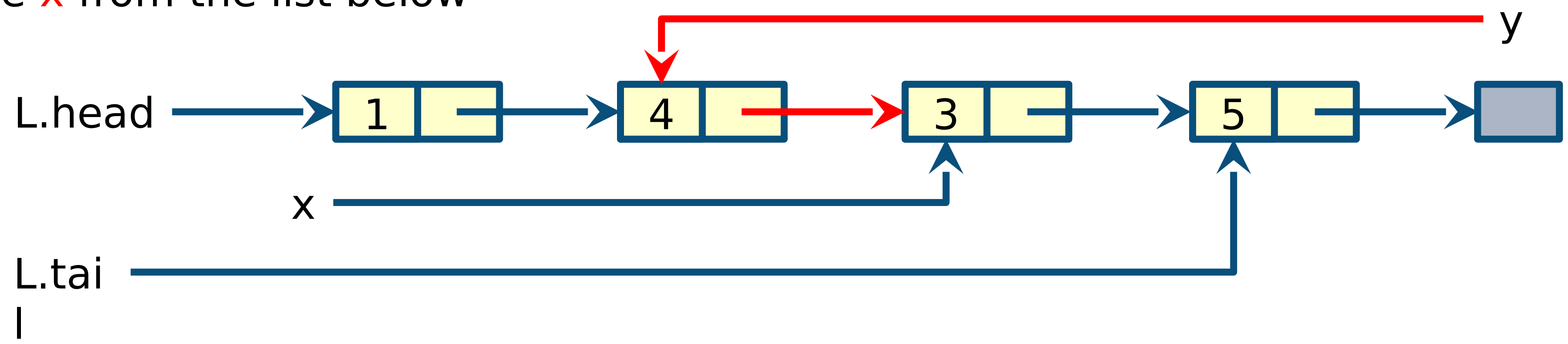
- **Example for you to try**
  - Delete x from the list below



Deletion of an element

- **Remove an element x from a linked list L**
  - Pointer to x must be retrieved first (for instance by calling SEARCH)

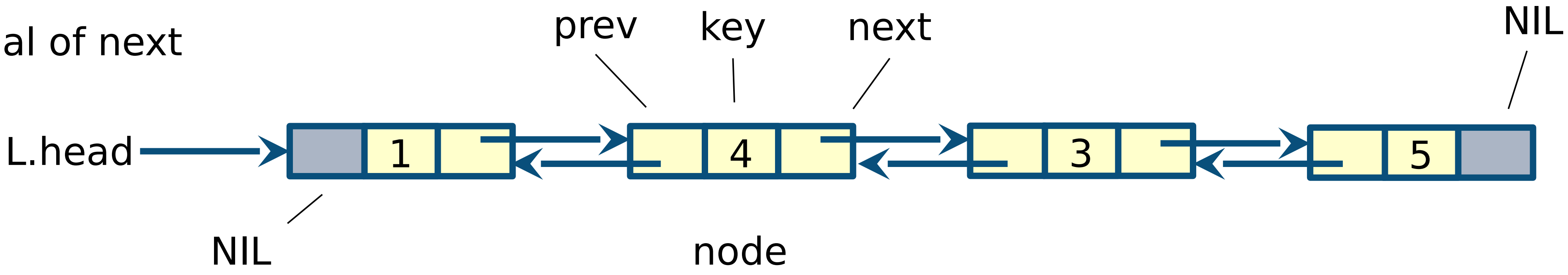
- **Example for you to try**
  - Delete x from the list below



- **We need a pointer y to 4 in order to update y.next := x.next**
  - Scan the list again and return y when y.next = x (or modify SEARCH to return y and x)

Doubly linked lists

- **Extension of singly linked list in which each node has a pointer attribute prev**
  - Given a node x in the list, x.prev points to the previous node in the linked list
  - If x.prev = NIL, x has no predecessor and is therefore the head of the list
  - Dual of next



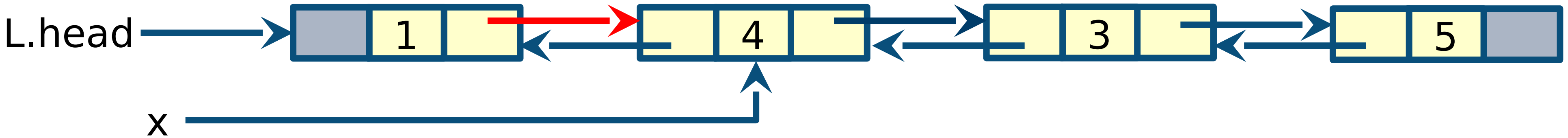
- **Pro: some operations are simplified and become more efficient**
- **Con: memory overhead**

Deletion of an element in doubly linked lists

- **Can be performed in constant time**

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

- **Example**
  - Delete x from the list below



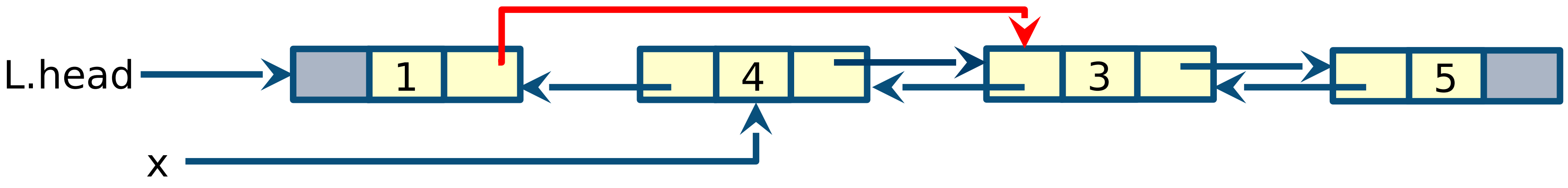
  - Update x.prev.next

Deletion of an element in doubly linked lists

- **Can be performed in constant time**

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

- **Example**
  - Delete x from the list below



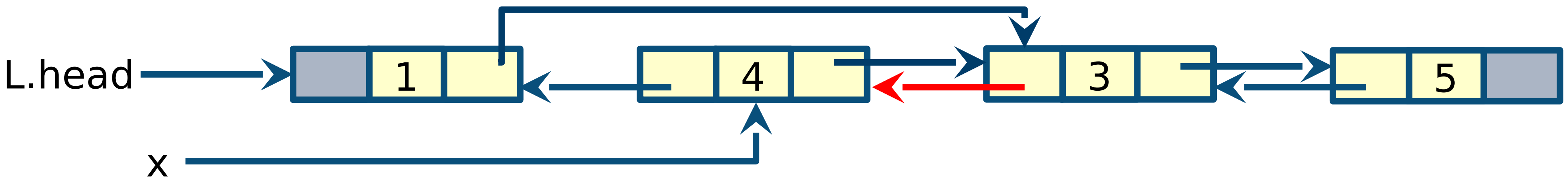
  - Update x.prev.next

Deletion of an element in doubly linked lists

- **Can be performed in constant time**

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

- **Example**
  - Delete x from the list below



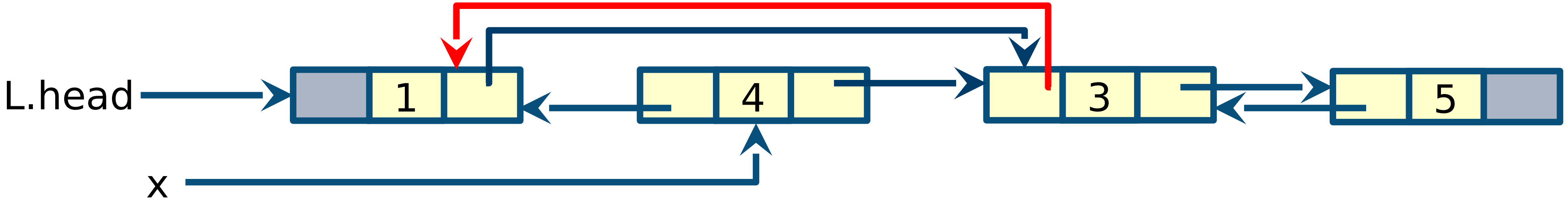
  - Update x.next.prev

Deletion of an element in doubly linked lists

- **Can be performed in constant time**

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

- **Example**
  - Delete x from the list below



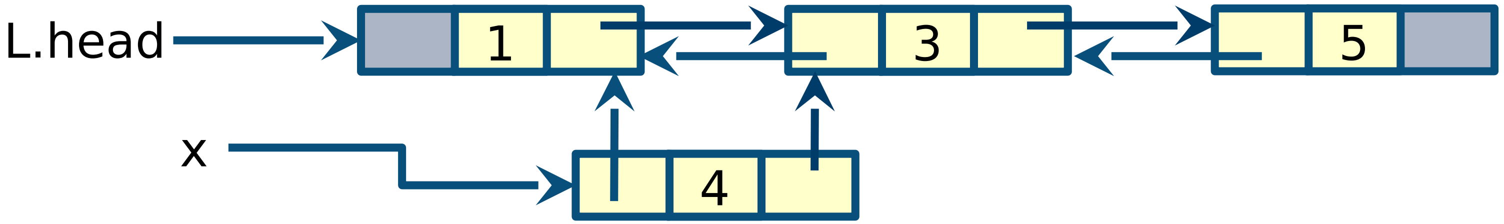
  - Update x.next.prev

Deletion of an element in doubly linked lists

- **Can be performed in constant time**

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

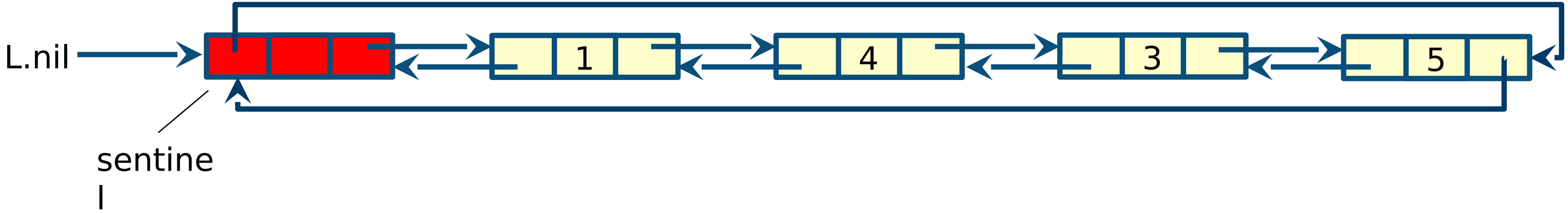
- **Example**
  - Delete x from the list below



  - Termination

Circular doubly linked list with a sentinel

- **Boundary conditions complicate the specification of the operations on doubly linked lists**
  - We can introduce sentinels to simplify the code
- **A sentinel is a dummy node L.nil between the head and tail**
  - L.nil.next points to the head
  - L.nil.prev points to the tail
  - The next attribute of the tail and the prev attribute of the head point to L.nil
  - Attribute L.head is no longer needed (use L.nil.next instead)



Operations

```
DELETE-CIRC(L,x)
  x.prev.next := x.next

  x.next.prev := x.prev
```

```
INSERT-CIRC(L,x)
  x.next := L.nil.next
  L.nil.next.prev := x
  L.nil.next := x
  x.prev := L.nil
```

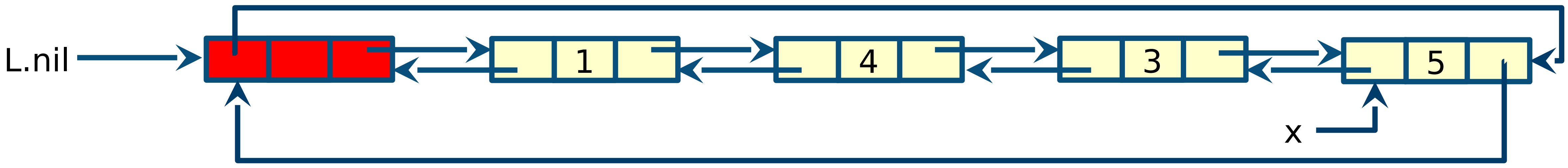
```
SEARCH-CIRC(L,k)
  x := L.nil.next
  while x != L.nil and x.key != k
    x := x.next
  return x
```

Example

- **Delete x from the list below**
  - No need to check if we are at the head or the tail of the list

```
DELETE-CIRC(L,x)
  x.prev.next := x.next

  x.next.prev := x.prev
```



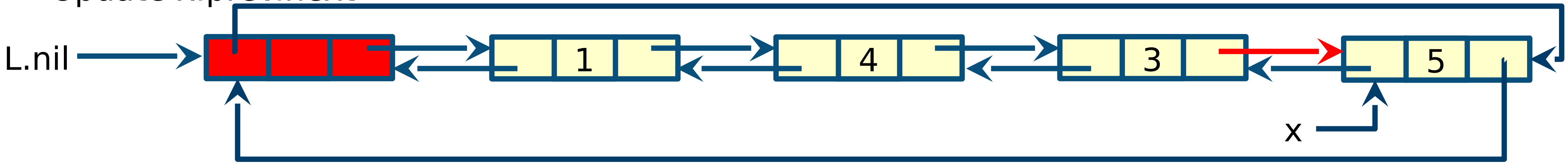
Example

- **Delete x from the list below**
  - No need to check if we are at the head or the tail of the list

```
DELETE-CIRC(L,x)
  x.prev.next := x.next

  x.next.prev := x.prev
```

  - Update x.prev.next



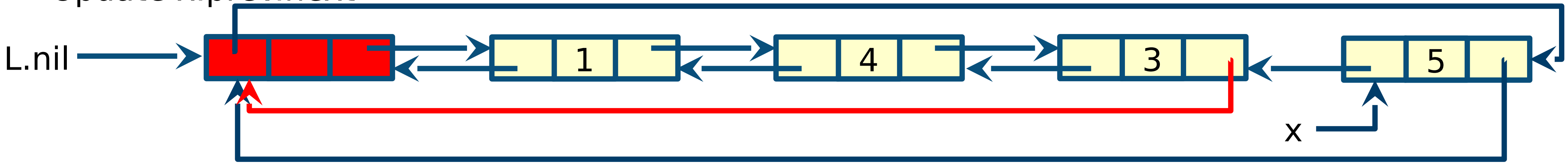
Example

- **Delete x from the list below**
  - No need to check if we are at the head or the tail of the list

```
DELETE-CIRC(L,x)
  x.prev.next := x.next

  x.next.prev := x.prev
```

  - Update x.prev.next



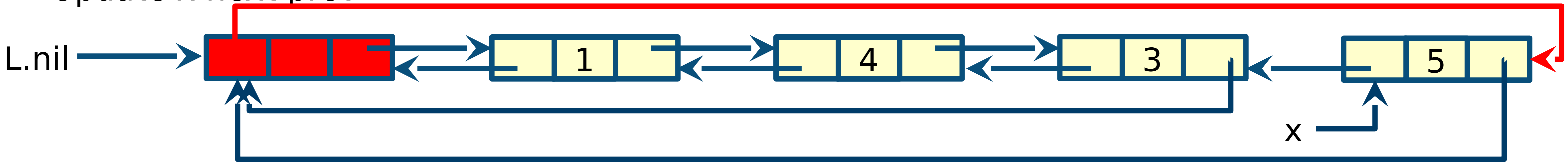
Example

- **Delete x from the list below**
  - No need to check if we are at the head or the tail of the list

```
DELETE-CIRC(L,x)
  x.prev.next := x.next

  x.next.prev := x.prev
```

  - Update x.next.prev



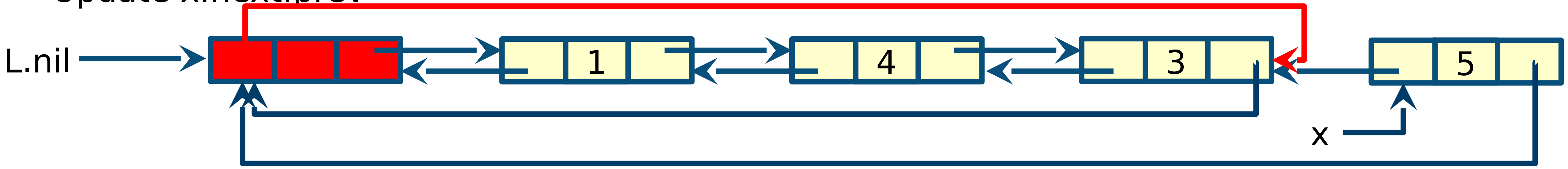
Example

- **Delete x from the list below**
  - No need to check if we are at the head or the tail of the list

```
DELETE-CIRC(L,x)
  x.prev.next := x.next

  x.next.prev := x.prev
```

  - Update x.next.prev



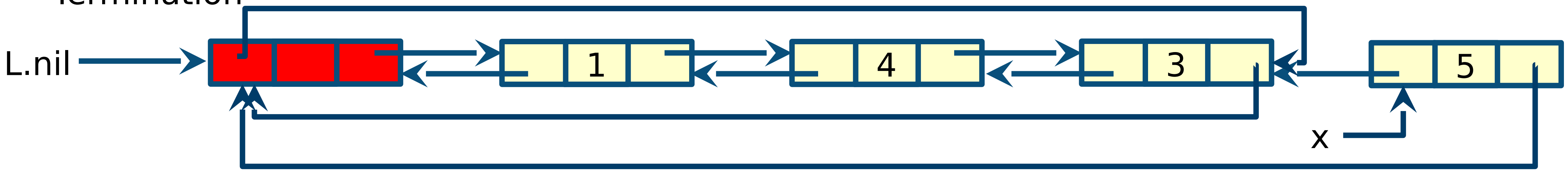
Example

- **Delete x from the list below**
  - No need to check if we are at the head or the tail of the list

DELETE-CIRC(L,x)
```
  x.prev.next := x.next

  x.next.prev := x.prev
```

  - Termination



Summary

- **Singly linked lists**
  - Insertion
  - Deletion
  - Search
  - Recursion
  - MERGE-SORT
- **Doubly linked lists**
  - Operations
- **Circular doubly linked list with a sentinel**
  - Operations