Lists

- List ADT.
- Singly linked lists. Doubly linked lists
- Implementations

The List Abstract Data Type

- The list supports three fundamental operations:
  - insert(x): Insert ListElement *x* at the front of the list
    - *Input*: ListElement; *Output*: none
  - delete(x): Remove the ListElement from the front of the list; an error occurs if the list is empty
    - *Input*: pointer to element to delete; *Output*: none
  - search(k): search for ListElement of key *k* on list
    - *Input*: key ot search; *Output*: pointer to ListElement or nil if not found
- These support methods should also be defined:
  - size(): Return the number of ListElements in the lits
    - *Input*: none; *Output*: integer
  - isEmpty(): Return a boolean value that indicates whether the list is empty
    - *Input*: none; *Output*: boolean
  - first(): Return, but do not remove, the first ListElement in the list; an error occurs if the queue is empty
    - *Input*: none; *Output*: ListElement
  - last(): Return, but do not remove, the last ListElement in the list; an error occurs if the queue is empty
    - *Input*: none; *Output*: ListElement
  - prev(x), next(x): ListElement preceeding/succeeding ListElement *x* on list

The List ADT. Operations

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3        do x ← next[x]
4  return x
```

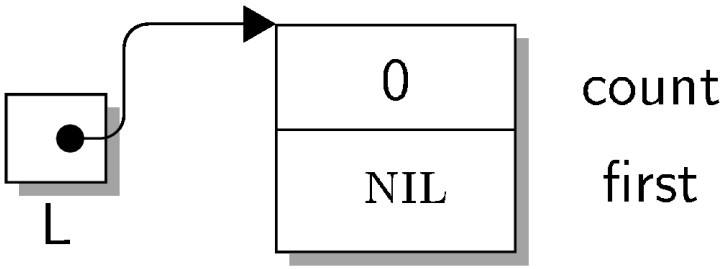
```
LIST-INSERT(L, x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL
3     then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

The List ADT. Operations

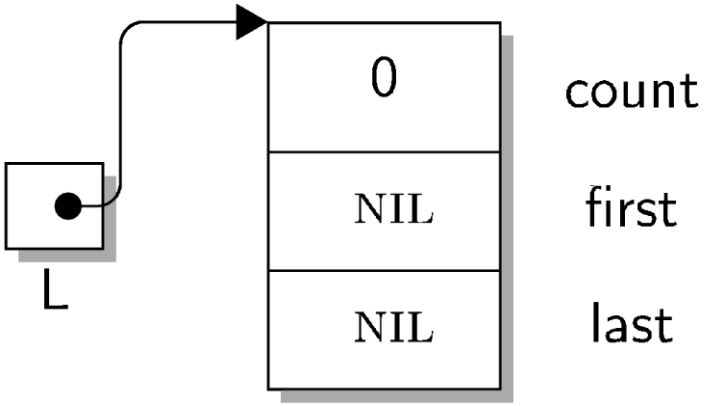
```
LIST-DELETE(L, x)
1   if prev[x] ≠ NIL
2       then next[prev[x]] ← next[x]
3       else head[L] ← next[x]
4   if next[x] ≠ NIL
5       then prev[next[x]] ← prev[x]
```

Singly Linked Lists

- One way chaining

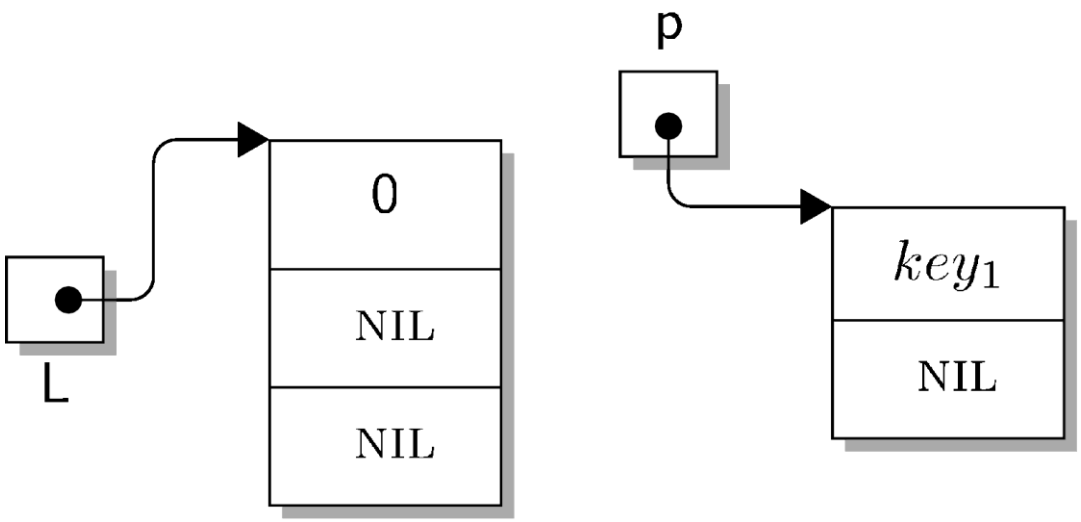


- Header cell:
  - Two (three) fields:
    - count
    - first
    - (last)



Singly Linked Lists

- Create an empty list:
  - Create header cell
  - Set *L* to point to header cell
- Create node to insert/append

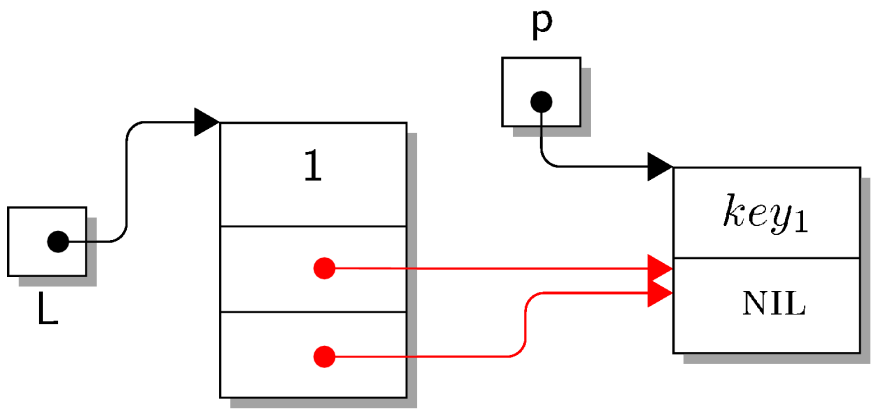


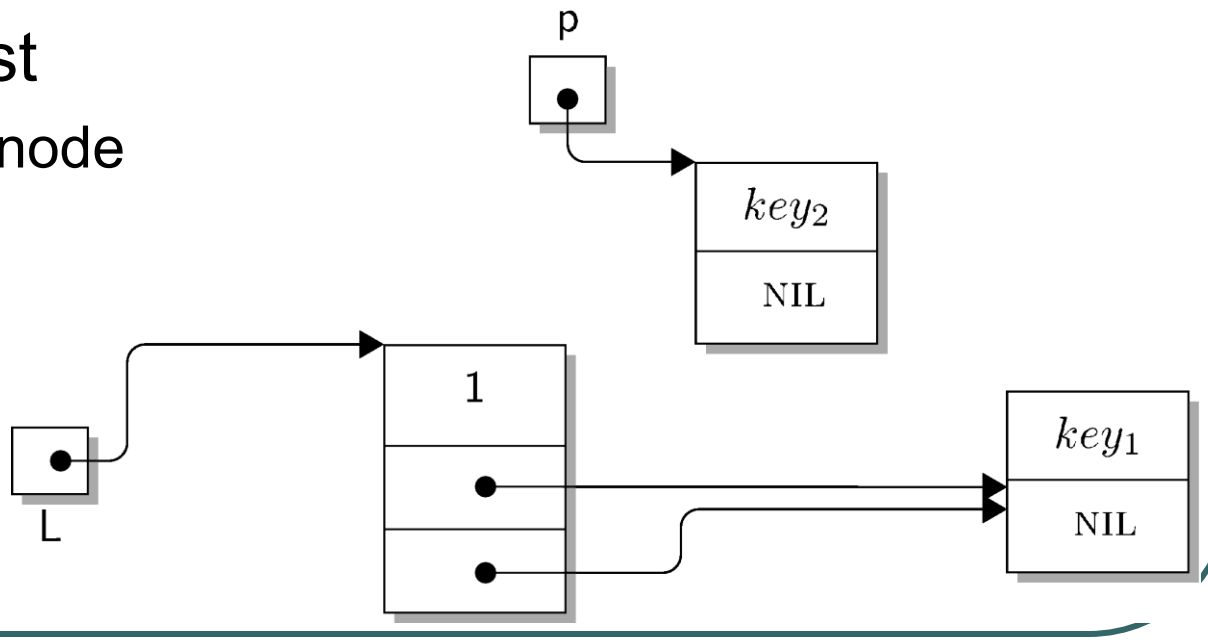
Singly Linked Lists

- Insert
  1. (Always) at front
  2. At both ends (front and rear)
  3. By Key value
- Create cell to insert
  - Case 1: insert
  - Case 2: append
  - Case 3:
    - find place
    - insert

Singly Linked Lists

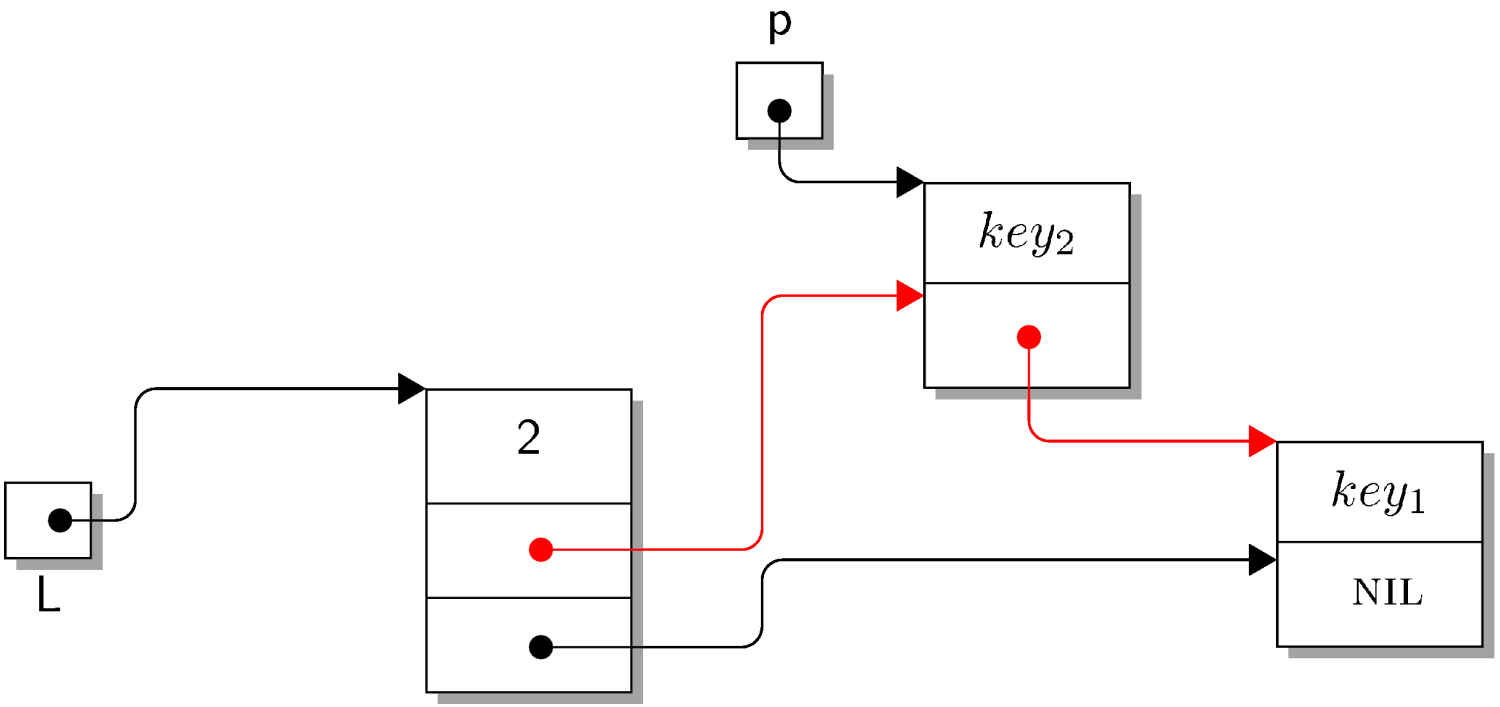
- Insert at front: cases
  - Empty list
    - Same for append
  - Non-empty list
    - Create new node





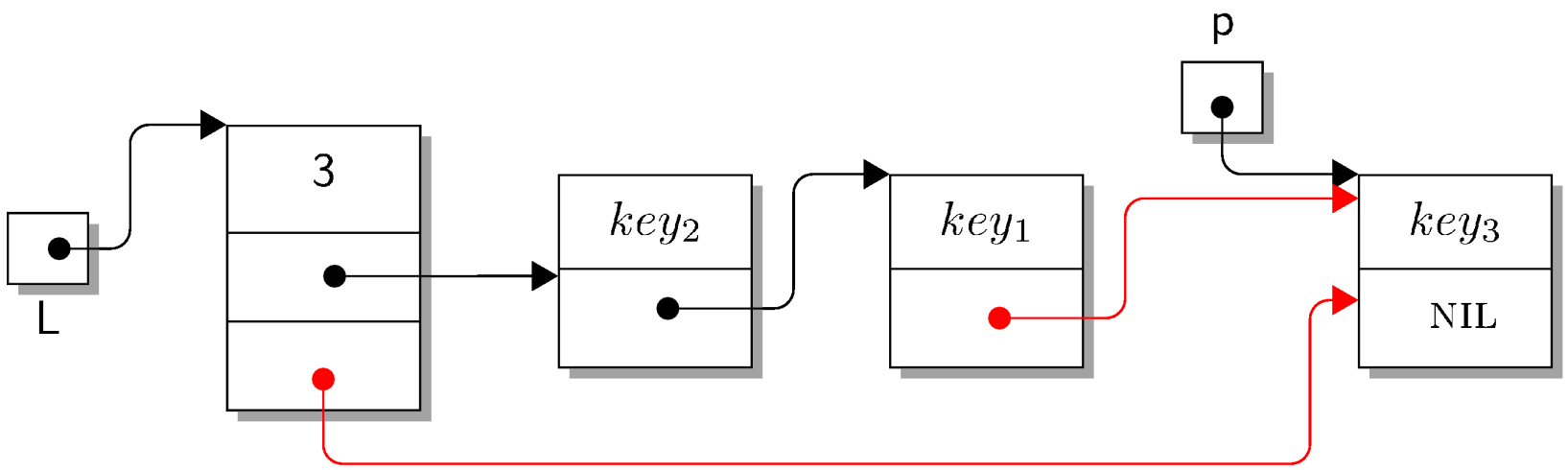
Singly Linked Lists

- Insert at front: cases
  - Non-empty list
    - Cont'd



Singly Linked Lists

- Append cases
  - Empty list: same as insert at front
  - Non empty list

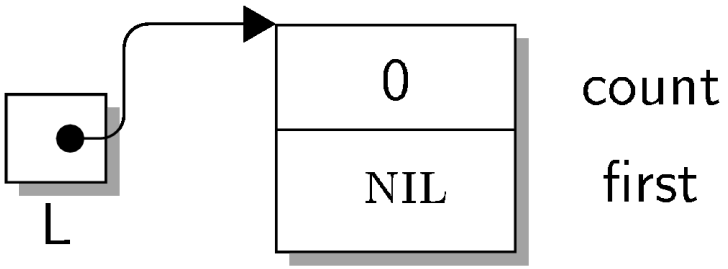


Singly Linked Lists

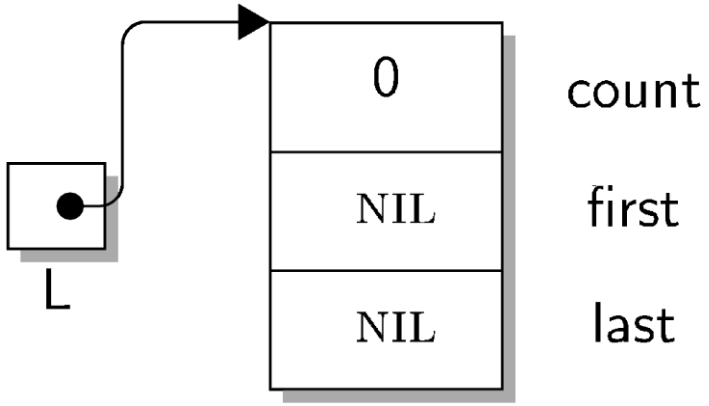
- Insert in order (place determined by key value)
  - Empty list: same as insert at front
  - Non empty list case:
    - Before first node (i.e. at front)
    - After last node (i.e. append)
    - Inside the list
  - List traversal needs two pointers:
    - current
    - previous: one node behind
  - Left as an exercise

Doubly Linked Lists

- Two way chaining

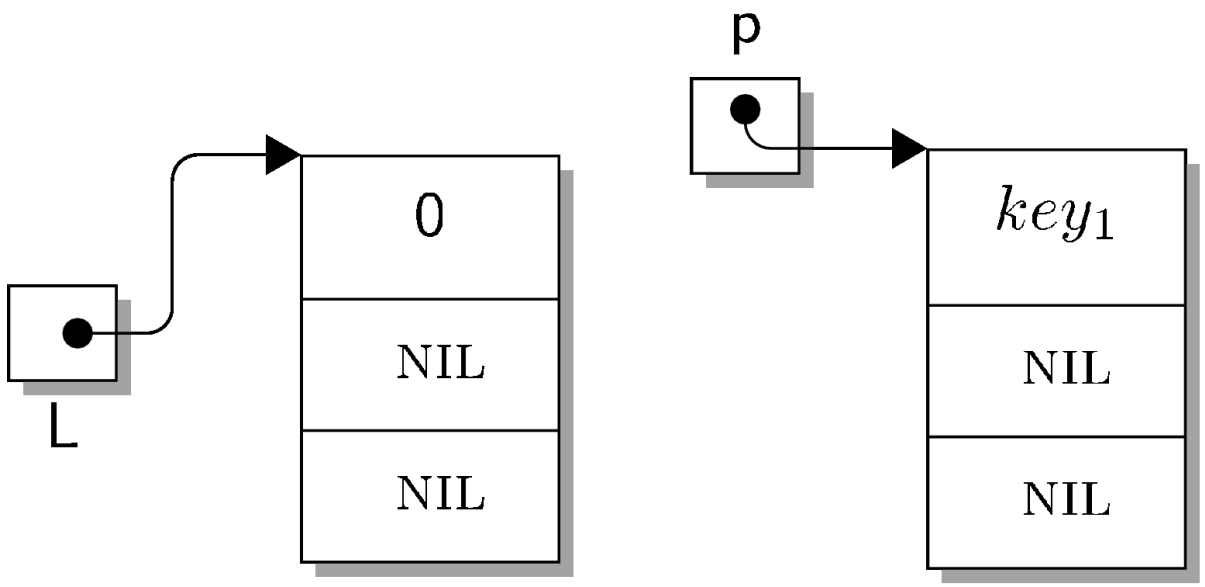


- Header cell
  - Same as for previous list
  - Two (three) fields:
    - count
    - first
    - (last)



Doubly Linked Lists

- Create an empty list:
  - Create header cell
  - Set *L* to point to header cell
- Create node to insert/append

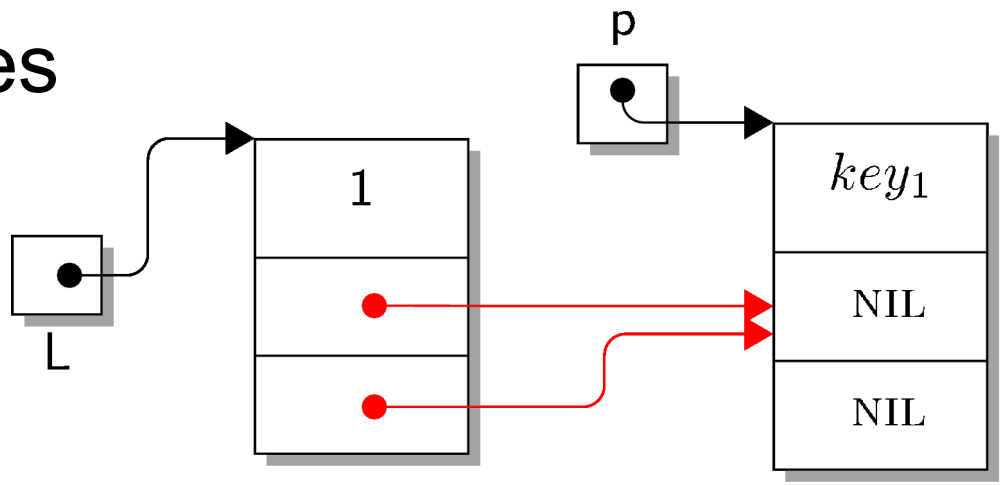


Doubly Linked Lists

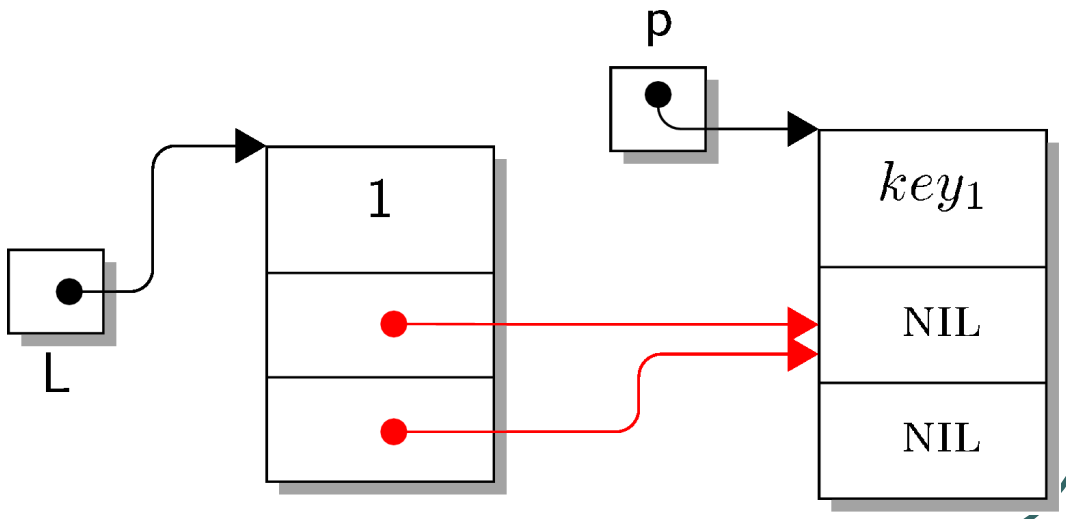
- Insert
  1. (Always) at front
  2. At both ends (front and rear)
  3. By Key value
- Create cell to insert
  - Case 1: insert
  - Case 2: append
  - Case 3:
    - find place
    - insert

Doubly Linked Lists

- Insert at front: cases
  - Empty list
    - Same for append

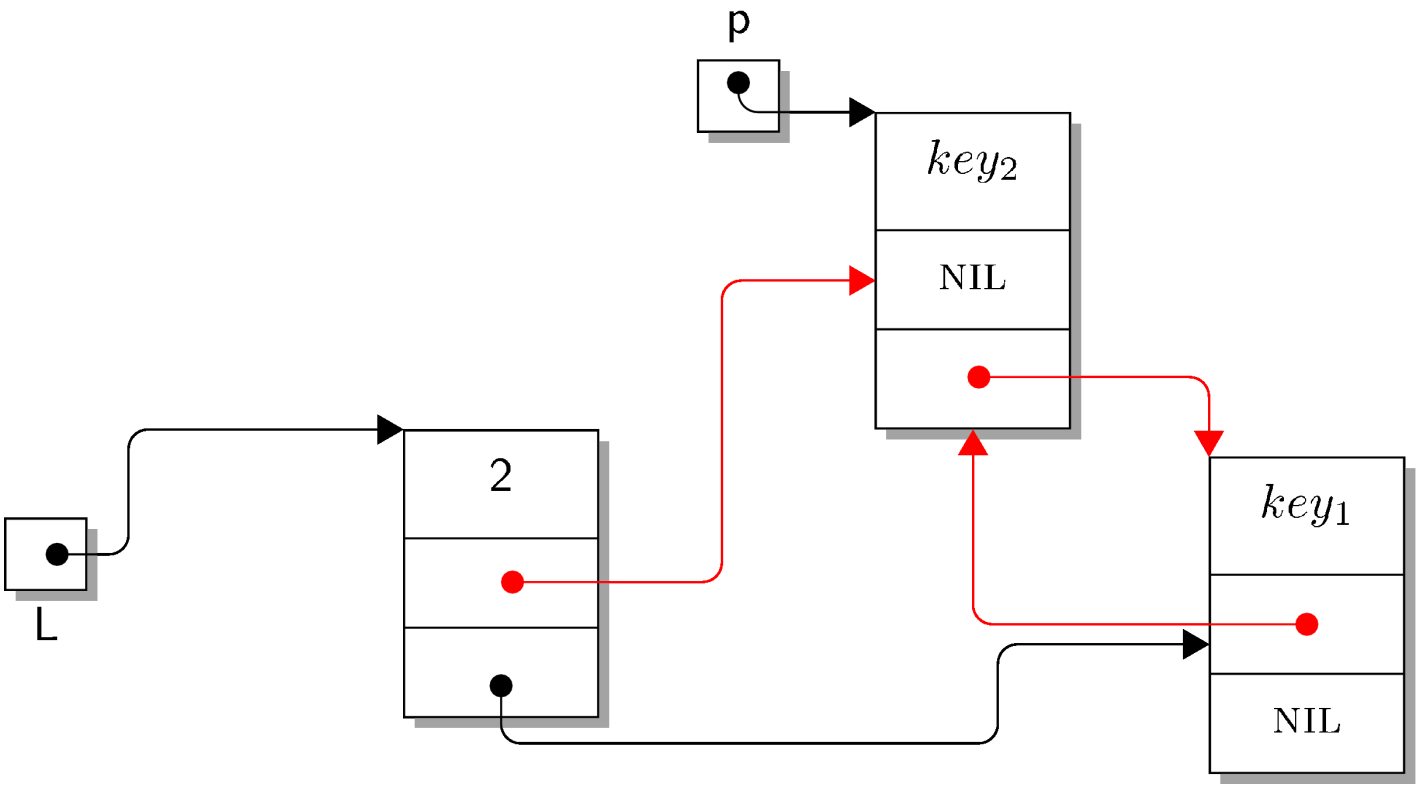


  - Non-empty list
    - Create new node



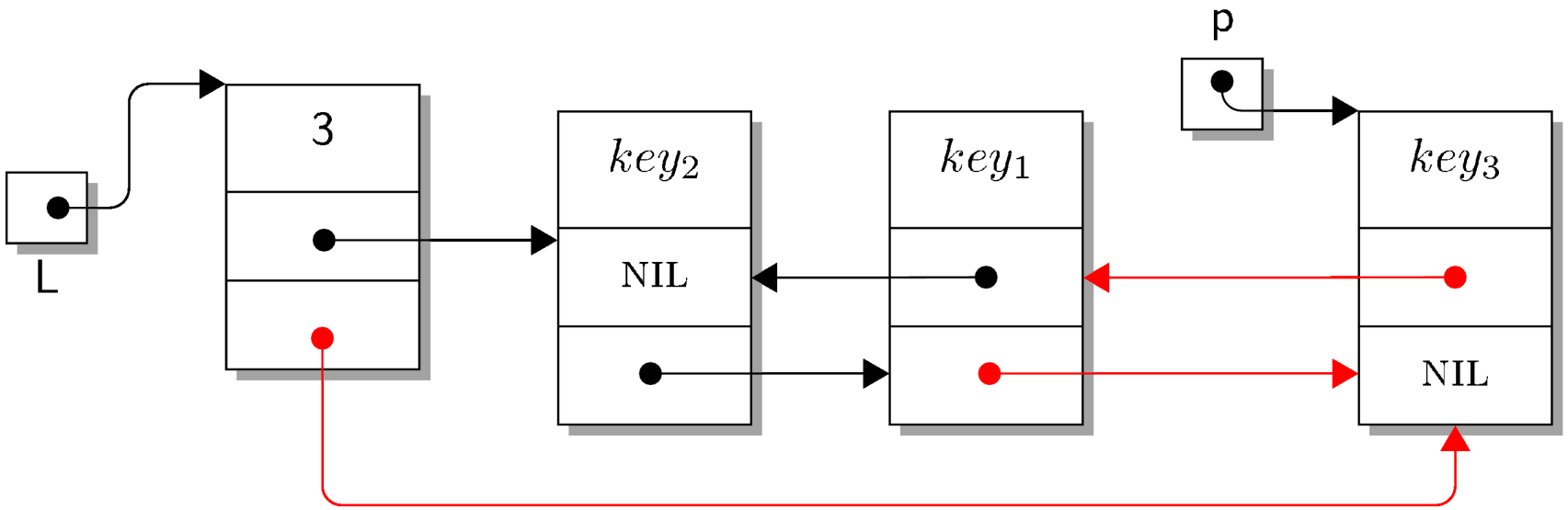
Doubly Linked Lists

- Insert at front: cases
  - Non-empty list
    - Cont'd



Doubly Linked Lists

- Append cases
  - Empty list: same as insert at front
  - Non empty list

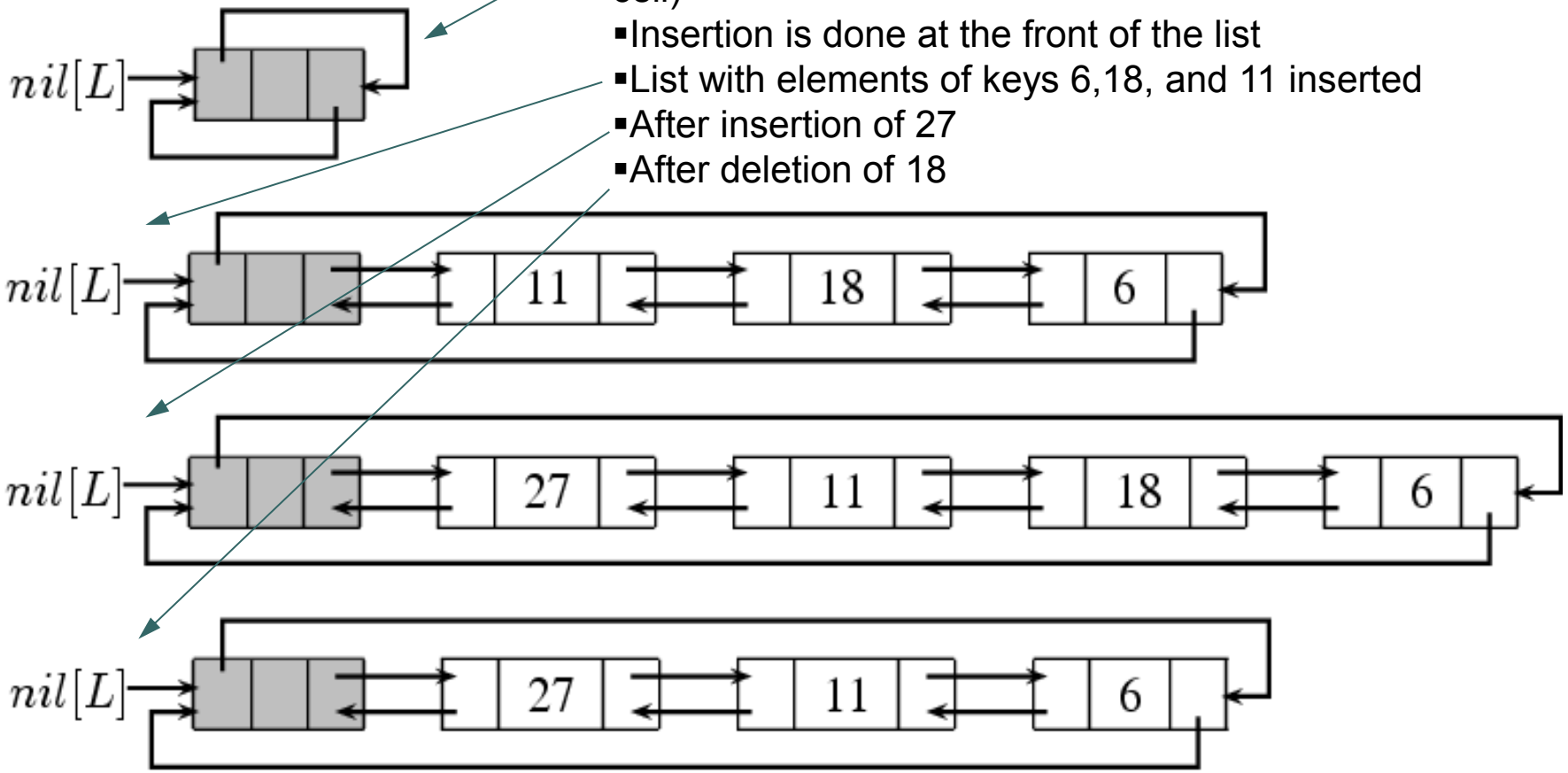


Doubly Linked Lists

- Insert in order (place determined by key value)
  - Empty list: same as insert at front
  - Non empty list case:
    - Before first node (i.e. at front)
    - After last node (i.e. append)
    - Inside the list
  - List traversal needs two pointers:
    - current
    - previous: one node behind
  - Left as an exercise

List examples with sentinels

- An empty list always contains the sentinel (grey cell)
- Insertion is done at the front of the list
- List with elements of keys 6,18, and 11 inserted
- After insertion of 27
- After deletion of 18



List ADT (w/ sentinels)

LIST-INSERT′ (L, x)

```
1  next[x] ← next[nil[L]]
2  prev[next[nil[L]]] ← x
3  next[nil[L]] ← x
4  prev[x] ← nil[L]
```

LIST-SEARCH′ (L, k)

```
1  x ← next[nil[L]]
2  while x ≠ nil[L] and key[x] ≠ k
3        do x ← next[x]
4  return x
```

LIST-DELETE′ (L, x)

```
1  next[prev[x]] ← next[x]
2  prev[next[x]] ← prev[x]
```

Reading

- AHU, chapters 1 & 2
- CLR, chapters 2, 11.1-11.3
- Preiss, chapters: Algorithm Analysis. Asymptotic Notation. Foundational Data Structures. Data Types and Abstraction. Stacks, Queues and Dequeues-4ups. Ordered Lists and Sorted Lists.
- Knuth, vol. 1, 2.1, 2.2
- Notes