CH-231-A

# Algorithms and Data Structures

ADS

## Lecture 21

Dr. Kinga Lipskoch

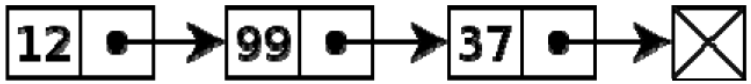
Spring 2022

# Linked List (1)

- Another elementary dynamic data structure.
- Flexible implementation of idea of dynamic set.
- Implies a linear ordering of the elements.
- However, in contrast to an array, the order is not determined by indices but by links or pointers.
- The pointer supports the operations finding the succeeding (next) entry in the list.
- In contrast to arrays, lists do typically not support random access to entries.

# Linked List (2)

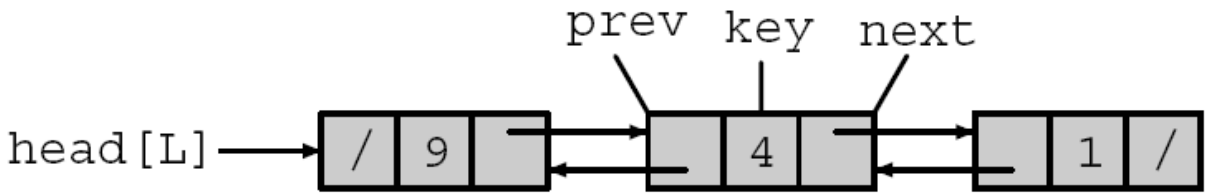
- Example of a linked list:




- Linked lists are dynamic data structures that allocate the requested memory when required.
- Start of linked list $L$ is referred to as $head[L]$.
- $next[x]$ calls the pointer of element $x$ and reports back the element to which the pointer of $x$ is linking.

# Doubly-Linked List

- A doubly-linked list enhances the linked list data structure by also storing pointers to the preceding (previous) element in the list.
- Hence, one can iterate in forward and backward direction.
- Example:



# Linked List Operations

Queries:

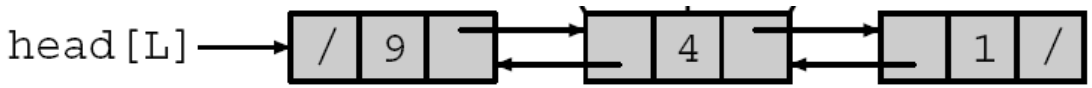
- Searching:

```
List-Search(L,k)
1   x ← head[L]
2   while x ≠ nil and key[x] ≠ k
3       do x ← next[x]
4   return x
```

- Time complexity: $O(n)$

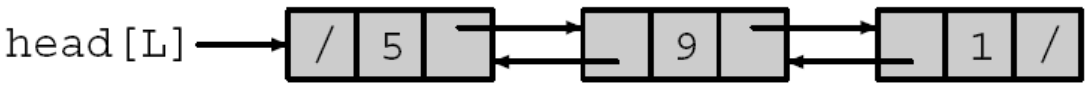
# Modify Operations: Examples

- Example:

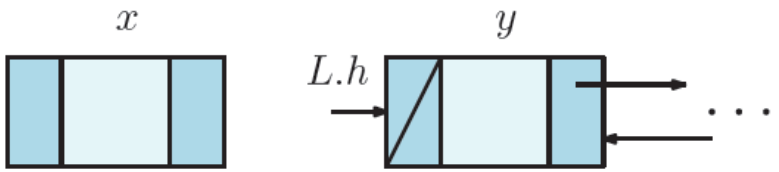



- Insert element $x$ with $key[x] = 5$ (at beginning):

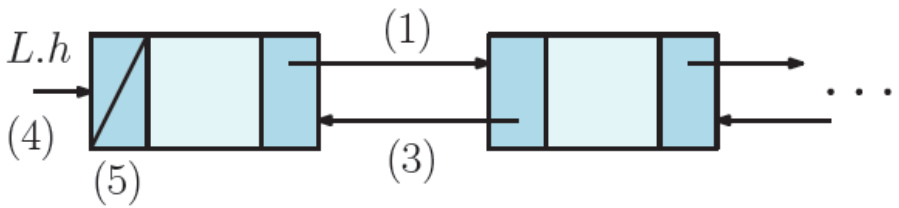

- Delete element $x$ with $key[x] = 4$:



# Insertion (at Beginning)




```
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
```

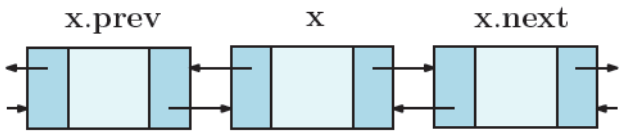



Time complexity: $\Theta(1)$

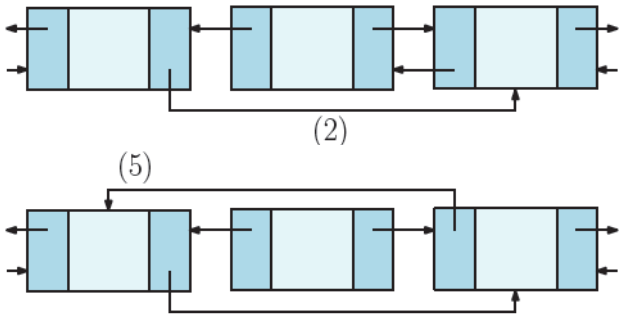
# Insertion (Middle or End)

- We can also insert after a given element $x$.
- Time complexity:
  - $O(1)$, if element $x$ is given by its pointer.
  - $O(n)$, if element $x$ is given by its key (because of searching).

# Deletion




```
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]
```

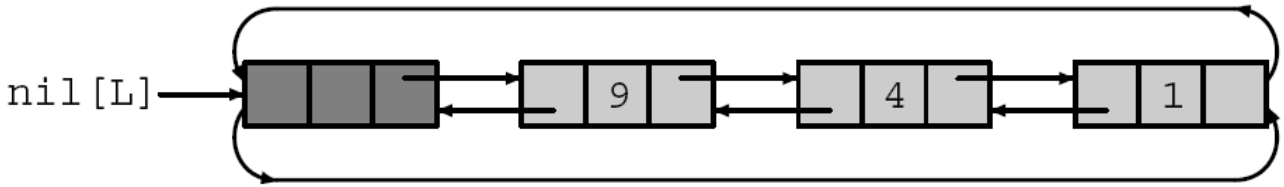



Time complexity:

O(1) if we use pointer and O($n$) if we use key (because of searching).

# Sentinels (1)

- In order to ease the handling of boundary cases, one can use dummy elements, so-called sentinels.
- Sentinels are handled like normal elements.
- One sentinel suffices when using circular lists.




```
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
```

# Sentinels (2)

```
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-Delete'(L,x)
1   next[prev[x]] ← next[x]
2   prev[next[x]] ← prev[x]
```