CH08-320201
Algorithms and Data Structures

Lecture 13/14 — 03 Apr 2018

Prof. Dr. Michael Sedlmair

Jacobs University
Spring 2018

This & that

- 2nd half: main focus on data structures
- Medical excuse policy: strict now
- coding assignments will be tested with unit tests
  —> exactly follow the descriptions!
- Missed the midterm
  —> final exam will be scaled up
- Final exam: in the last slot—>
  2018-05-15 — 9:00-11:00

3. Fundamental Data Structures

Data Structure

Definition (recall):

A data structure is a way to store and organize data in order to facilitate access and modification.

Examples we have seen so far:

- Array
- Heap
- Max-priority queue
- Linked list

Array

Definition:

- An array is a random-access data structure consisting of a collection of elements, each identified by an index or key.
- The simplest type of data structure is a linear array, where the indices are one-dimensional.
- A dynamic array refers to an array which can change its size.

Array

Examples of operations:

- Getting or setting the value at a particular index:
  - constant time.
- Iterating over the elements in order:
  - linear time.
- Inserting or deleting an element:
  - beginning (linear time),
  - middle (linear time),
  - end (constant time).

Dynamic set

- In the following, we assume that we are interested in storing and handling dynamic sets.
- Dynamic sets are sets of elements that can change its size.
- Elements are identified by a key from a totally ordered set.

Operations

Two categories of operations:

- Queries return the information of a stored object.
- Modify operations alter the set.

Examples for queries

- Search (S, k):
  - returns element x ϵ S with key[x] = k (nil if not existent).
- Minimum (S):
  - returns element x ϵ S with smallest key[x].
- Maximum (S):
  - returns element x ϵ S with largest key[x].
- Successor (S, x):
  - returns for element x ϵ S the next-larger element in S (nil if x is element with largest key).
- Predecessor (S, x):
  - returns for element x ϵ S the next-smaller element in S (nil if x is element with smallest key).

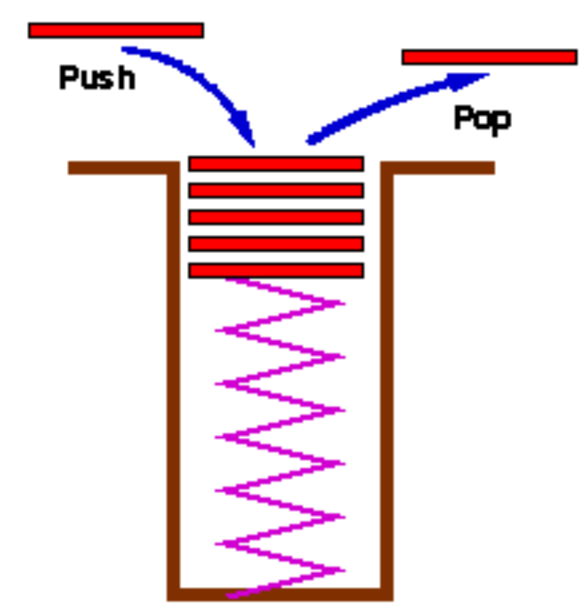
Examples for modify operations

- Insert (S, x):
  - adds element x to dynamic set S (S grows).
- Delete (S, x):
  - deletes element x from dynamic set S (S shrinks).

3.1 Stacks and Queues

Stack

- Elementary dynamic data structure.
- Implements idea of dynamic set.
- Idea follows that of a coin stacker.
- Delete operation is called pop.
- Insert operation is called push.
- LIFO principle (Last In First Out): The element that is returned by the pop operation is the last one that has been added (via push).



Stack operations

Queries:

- Stack-Empty (S):
  - True iff. stack S is empty.

Modify operations:

- Push (S,x):
  - Add element x on top of stack S and push other elements down.
- Pop (S):
  - If stack is non-empty, remove top-most element and return it.

Implementation as an array

S.top is the index of the top of the stack

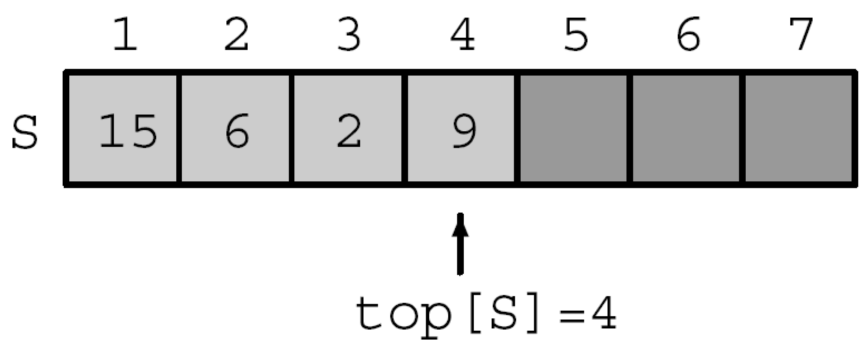
```
STACK-EMPTY(S)
if S.top == 0
    return TRUE
else return FALSE
```

```
PUSH(S, x)
S.top = S.top + 1
S[S.top] = x
```

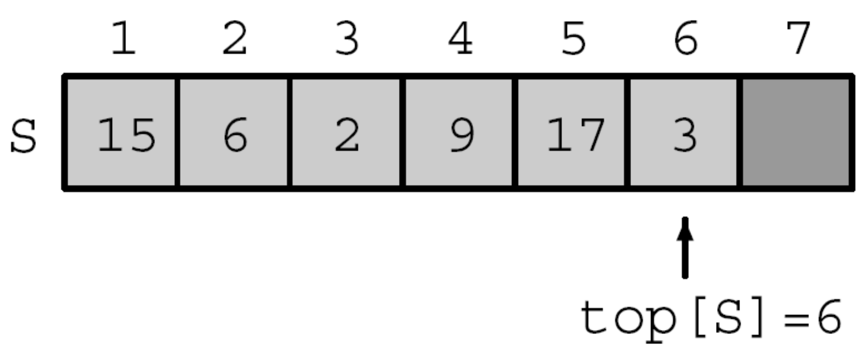
```
POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```

Stack example (array implementation)

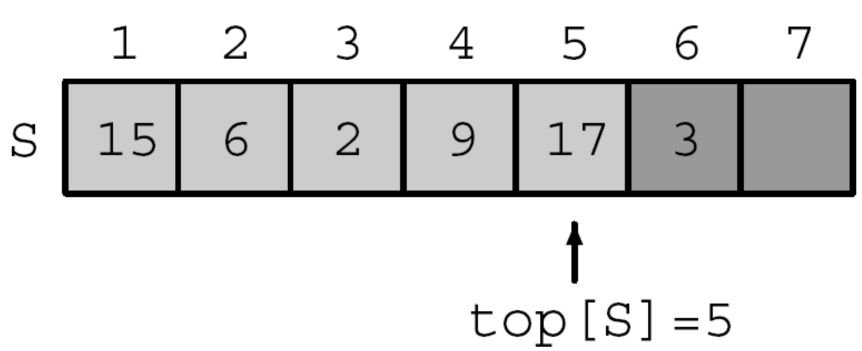
- Stack with four elements:



- Performing operations Push (S,17) and Push (S,3



- Performing operation Pop (S) returning entry 3:



Stack operations: Complexity

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

```
PUSH(S, x)
S.top = S.top + 1
S[S.top] = x
```

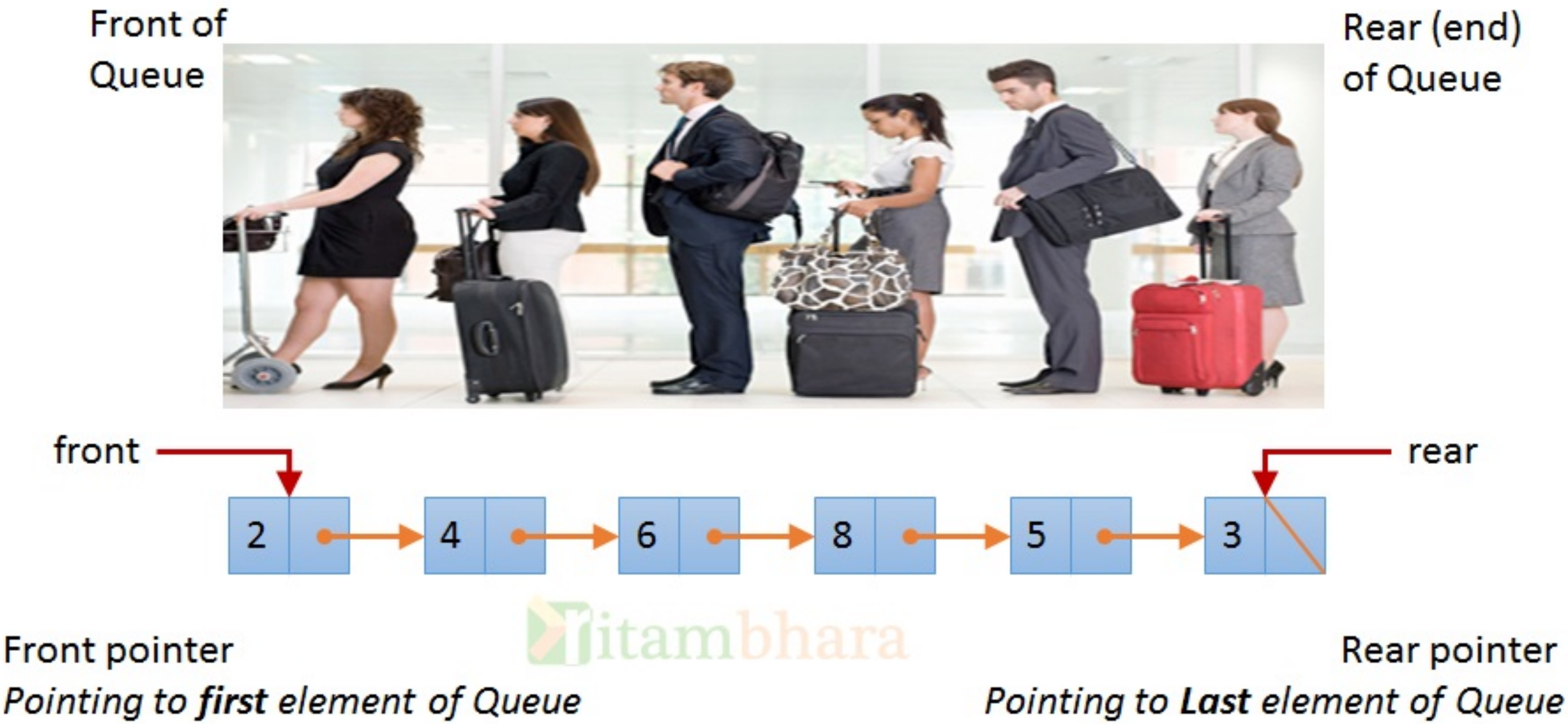
```
POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```

Complexity:
when implemented as an array all operations are O(1).

Stack operations: underflow and overflow

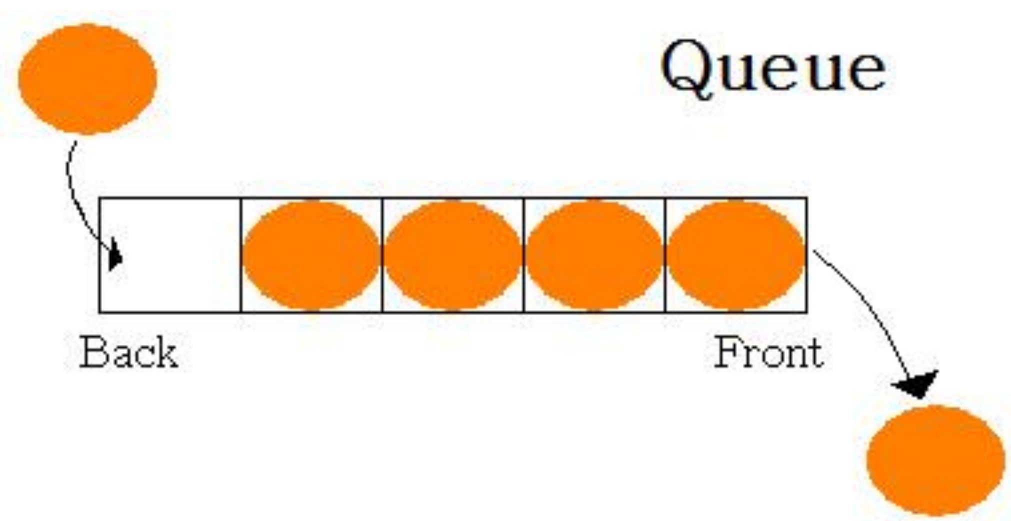
- If we want to perform a Pop-operation on the empty stack, we have a stack-underflow situation.
- We may also have a stack-overflow situation, if we assume that the stack has a maximum amount of entries (not considered in the array implementation).

Queue


Front of Queue
Rear (end) of Queue
front
rear
2
4
6
8
5
3
ritambhara
Front pointer
*Pointing to **first** element of Queue*
Rear pointer
*Pointing to **Last** element of Queue*

Queue

- Elementary dynamic data structure.
- Implements idea of dynamic set.
- Delete operation is called dequeue.
- Insert operation is called enqueue.
- FIFO principle (First In First Out):
  The element that is removed from the queue is the oldest one in the queue.



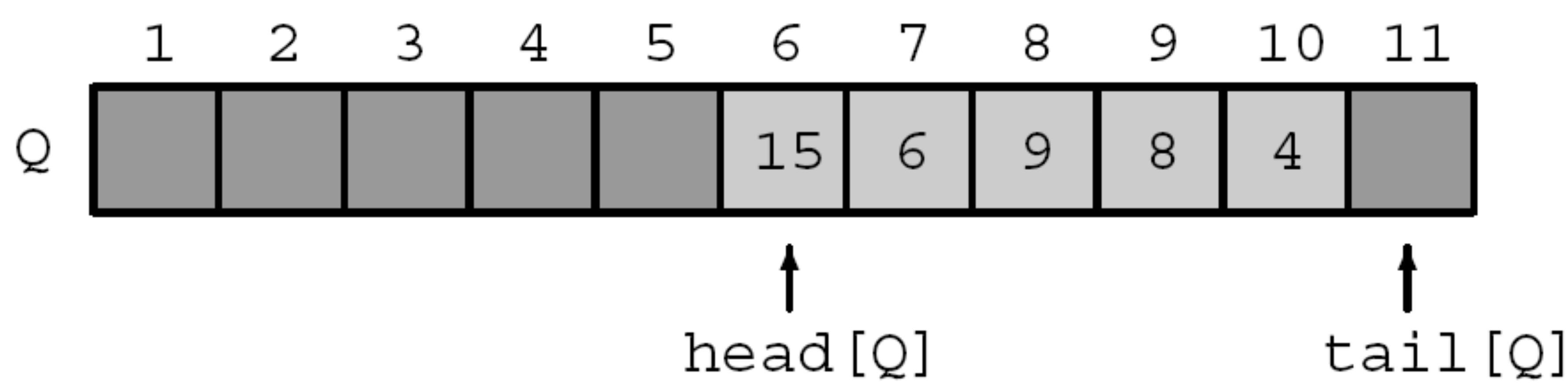
Queue operations

Modify operations:

- Enqueue (Q,x):
  - Add element x on at the tail of queue Q.
- Dequeue (Q):
  - If queue is non-empty, remove head element and return it.

Queue example (array implementation)

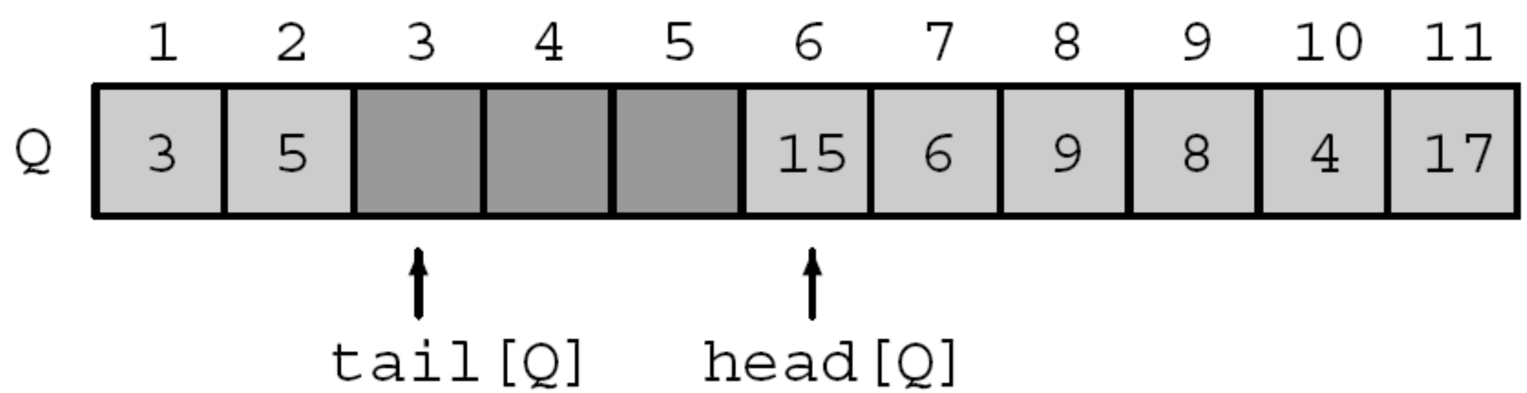
- head[Q] and tail[Q] mark the index of the first entry and the one following the last entry of the queue.
- Example:
  Queue with 5 elements between indices 6 (head) and 10 (tail).



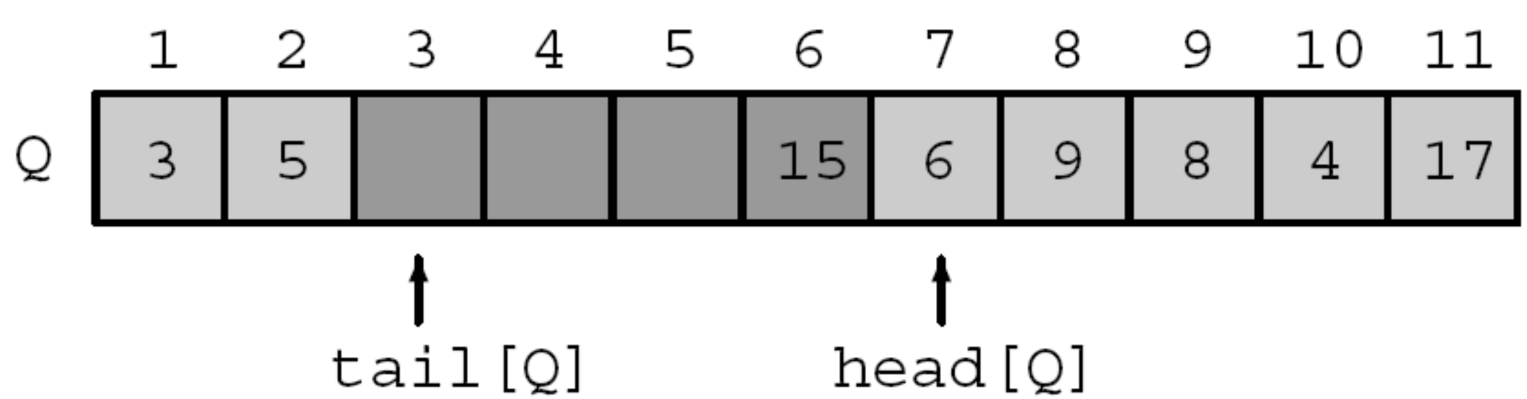
- We can also have under- and overflow.

Queue example (array implementation)

- Apply operations Enqueue (Q, 17), Enqueue (Q,3), and Enqueue (Q, 5):

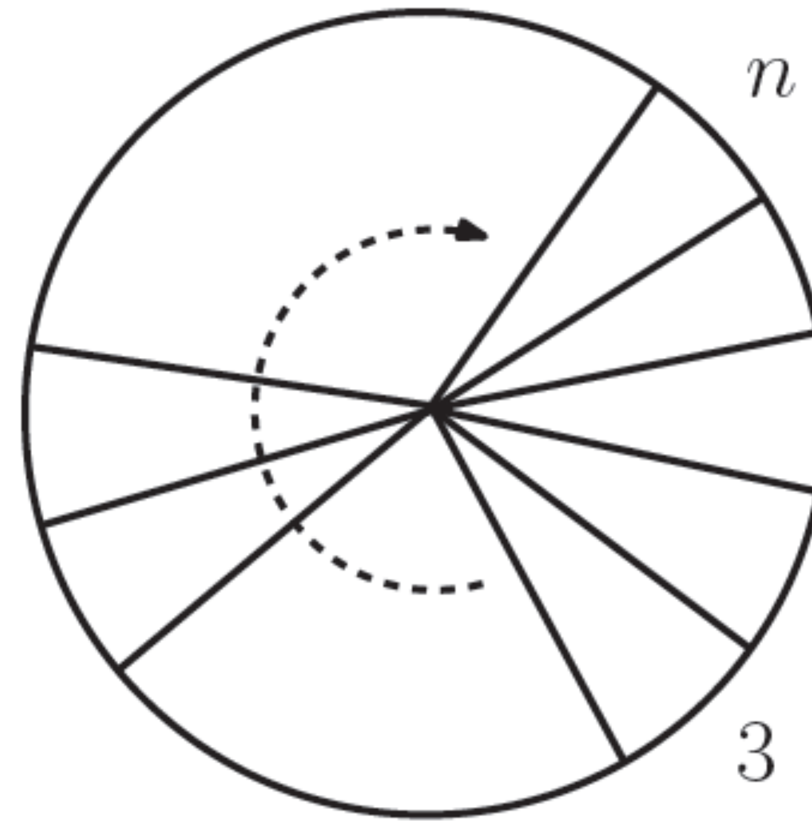


- Apply operation Dequeue (Q) returning entry 15:



Modulo operations

- Circular structure of filling the array with queue entries:



- Head[Q] = 1 and Tail[Q] = 5:
  - 4 entries
- Head[Q] = n-1 and Tail[Q] = 1:
  - 2 entries
- Head[Q] = n and Tail[Q] = n-1:
  - n-1 entries (full queue)

Queue operations (array implementation)

```
Enqueue(Q,x)
if tail[Q] = head[Q]-1 then
  error ‘overflow’
Q[tail[Q]] ← x
if tail[Q] = length[Q]
  then tail[Q] ← 1
  else tail[Q] ← tail[Q]+1
```

```
Dequeue(Q,x)
if tail[Q] = head[Q] then
  error ‘underflow’
x ← Q[head[Q]]
if head[Q] = length[Q]
  then head[Q] ← 1
  else head[Q] ← head[Q]+1
return x
```

Queue operations: Complexity

```
Enqueue(Q,x)
if tail[Q] = head[Q]-1 then
  error 'overflow'
Q[tail[Q]] ← x
if tail[Q] = length[Q]
  then tail[Q] ← 1
  else tail[Q] ← tail[Q]+1

Dequeue(Q,x)
if tail[Q] = head[Q] then
  error 'underflow'
x ← Q[head[Q]]
if head[Q] = length[Q]
  then head[Q] ← 1
  else head[Q] ← head[Q]+1
return x
```

Complexity: when implemented as an array all operations are O(1).

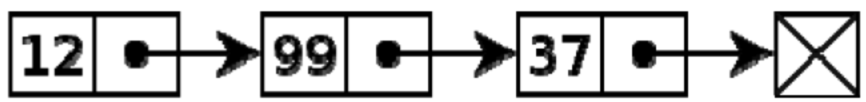
3.2 Linked Lists

Linked List

- 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

- 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.

Linked list operations

Queries:

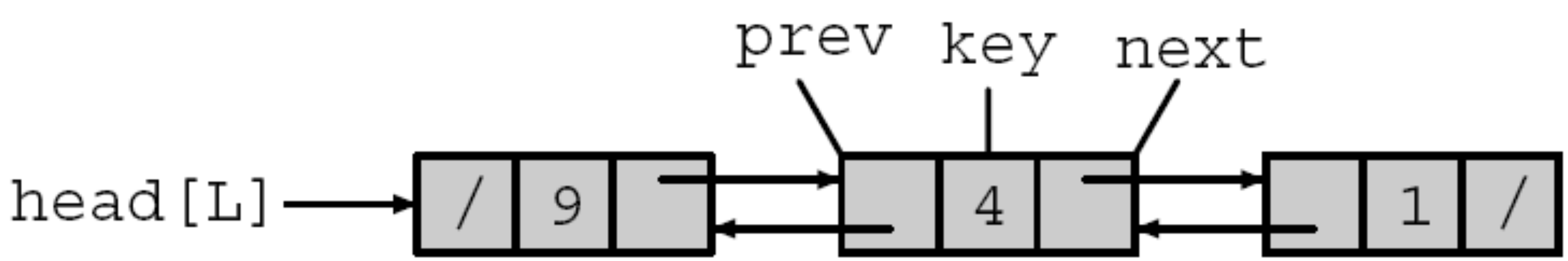
- Searching:

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

Running time:
O(n).

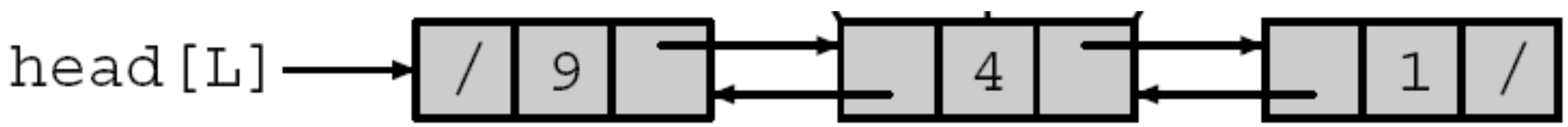
Doubly-linked list

- 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:



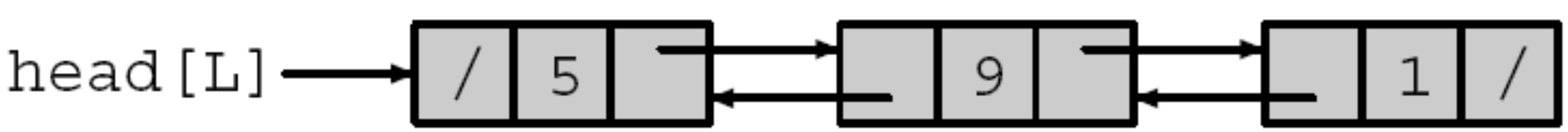
Modify operations: Examples

Example:

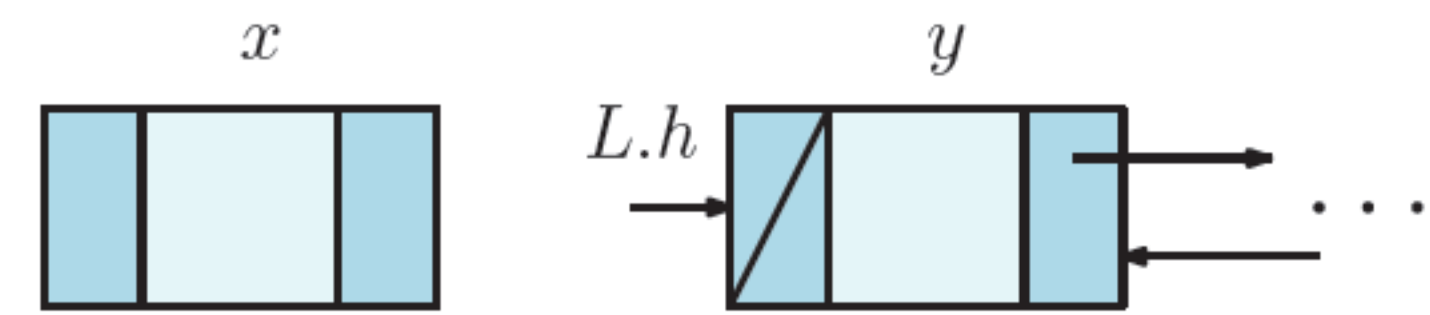


- Insert element x with key [x] = 5 (at beginning):

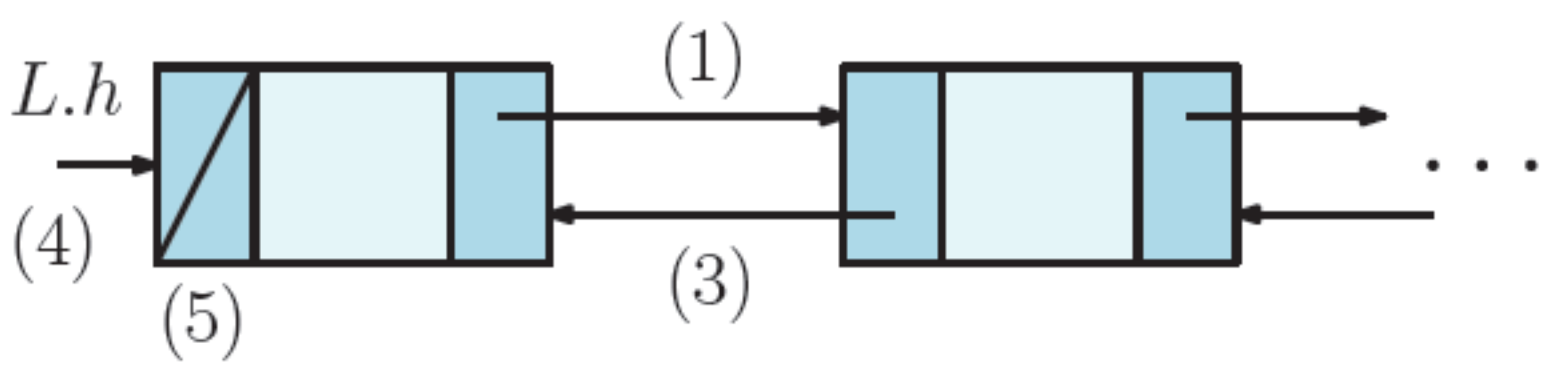
- Delete element x with key [x] = 4:



Insertion (at beginning)




```
List-Insert(L,x)
    next[x] ← head[L]
    if head[L] ≠ nil
        then prev[head[L]] ← x
        head[L] ← x
        prev[x] ← nil
```

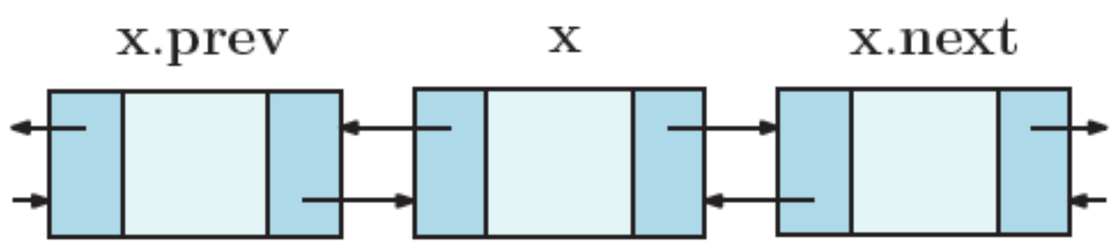


Computation time:
$\Theta(1)$.

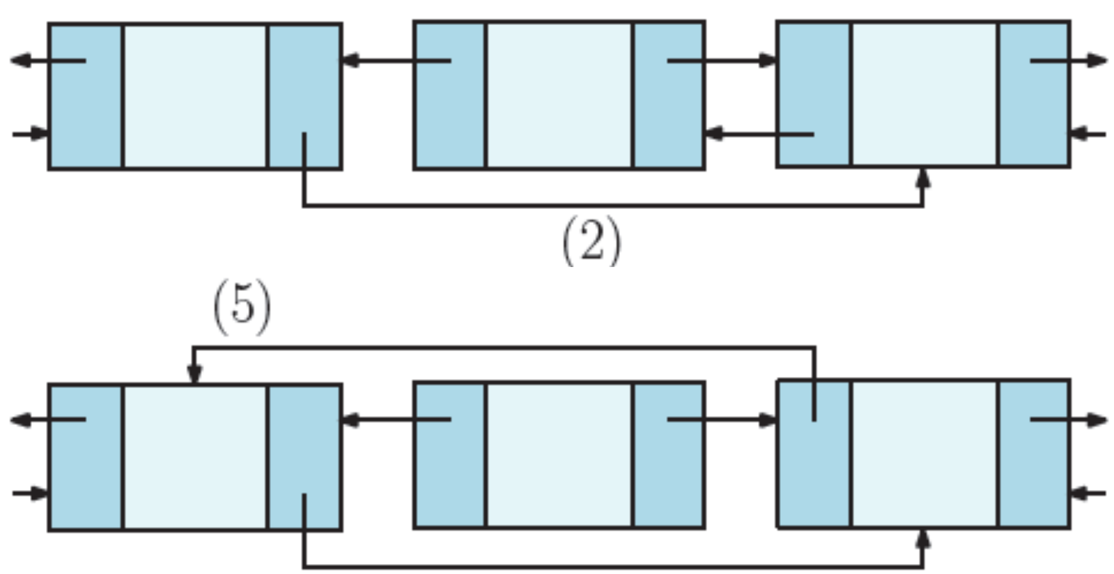
Insertion (middle or end)

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

Deletion



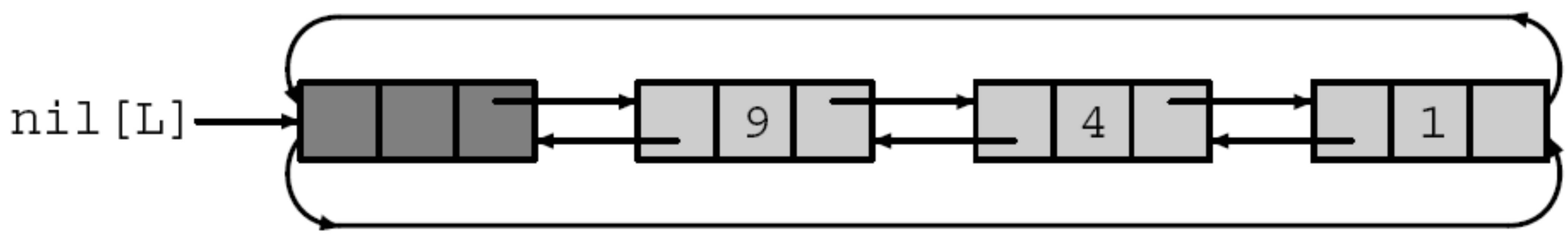

```
List-Delete(L,x)
    if prev[L] ≠ nil
        then next[prev[x]] ← next[x]
        else head[L] ← next[x]
    if next[x] ≠ nil
        then prev[next[x]] ← prev[x]
```



Computation time:
O(1) if we use pointer.
O(n) if we use key (because of searching).

Sentinels

- 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.

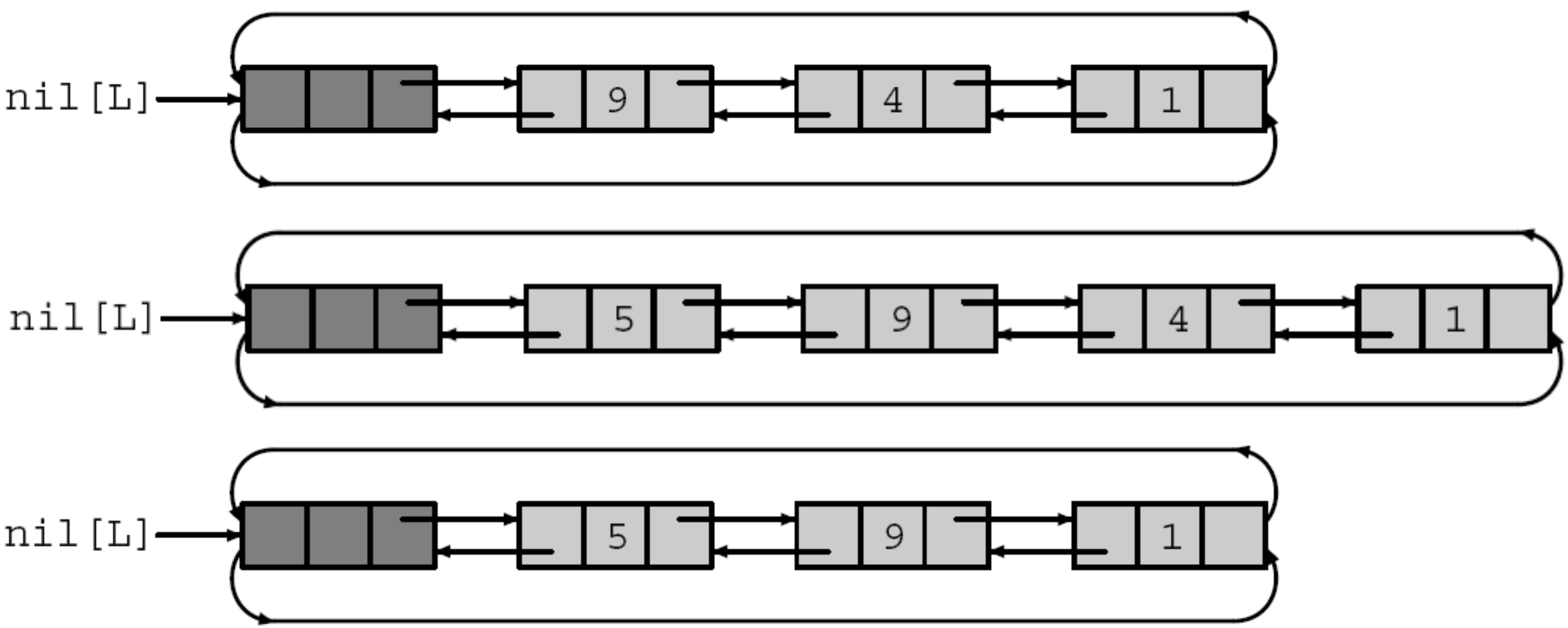



```
List-Search'(L,k)
x ← next[nil[L]]
while x ≠ nil[L] and key[x] ≠ k
    do x ← next[x]
return x
```

Sentinels

```
List-Insert'(L,x)
    next[x] ← next[nil[L]]
    prev[next[nil[L]]] ← x
    next[nil[L]] ← x
    prev[x] ← nil[L]
```

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



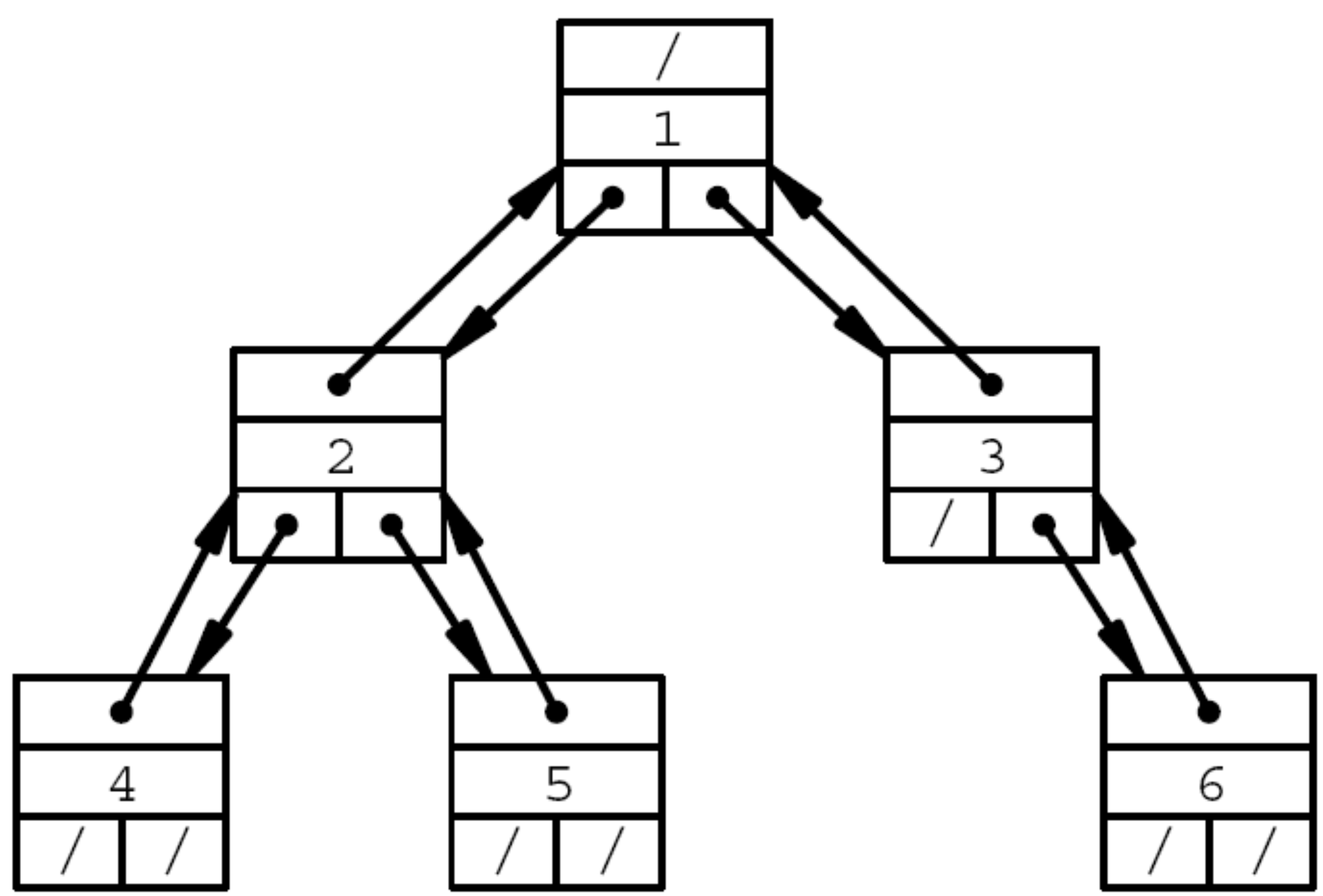
3.3 Rooted Trees

Representing rooted trees

- Traversing a rooted tree requires us to know about the hierarchical relationships of their nodes.
- Similar to linked list implementations, such relationships can be stored by using pointers.

Binary tree

- Binary trees T have an attribute T.root.
- They consist of nodes x with attributes x.parent (short x.p), x.left, and x.right (in addition to x.key).



d-ary trees

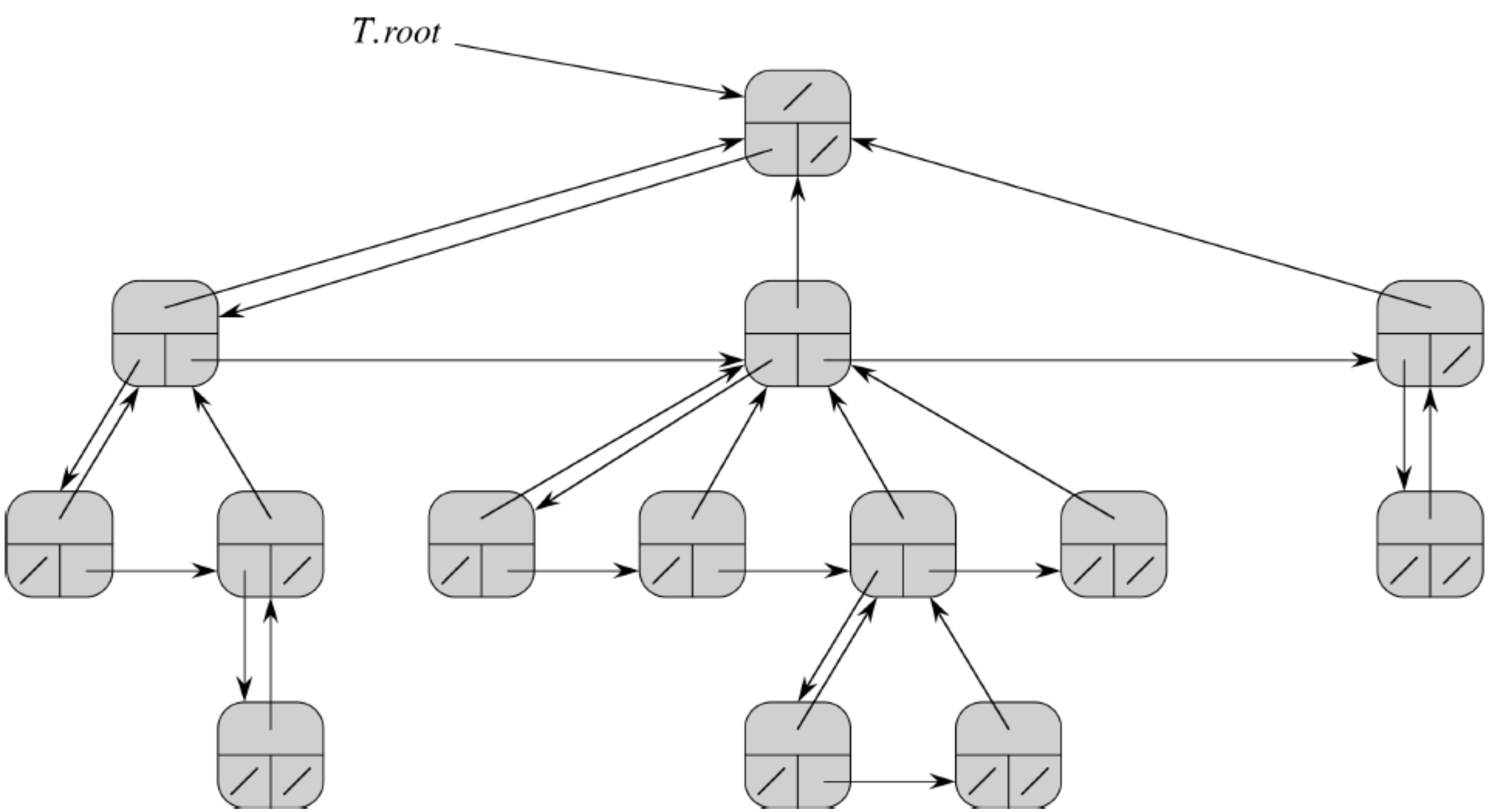
- d-ary trees are rooted trees with at most d children per node.
- They can be handled analogously to binary trees.

```
struct node {
    int val;
    node* parent;
    node* child[d];
};

typedef node* tree;
```

Rooted trees with arbitrary branching

- Rooted trees T with arbitrary branching consist of nodes x with attributes x.p, x.leftmost-child, and x.right-sibling (in addition to x.key).



Discussion

- Representing trees with pointers allows for a simple and intuitive representation.
- It also allows for a dynamic data management.
- Modifying operations can be implemented efficiently.
- However, extra memory requirements exist for storing the pointers.

3.4 Binary Search Trees

Definition

- A binary search tree (BST) is a binary tree with the following property:

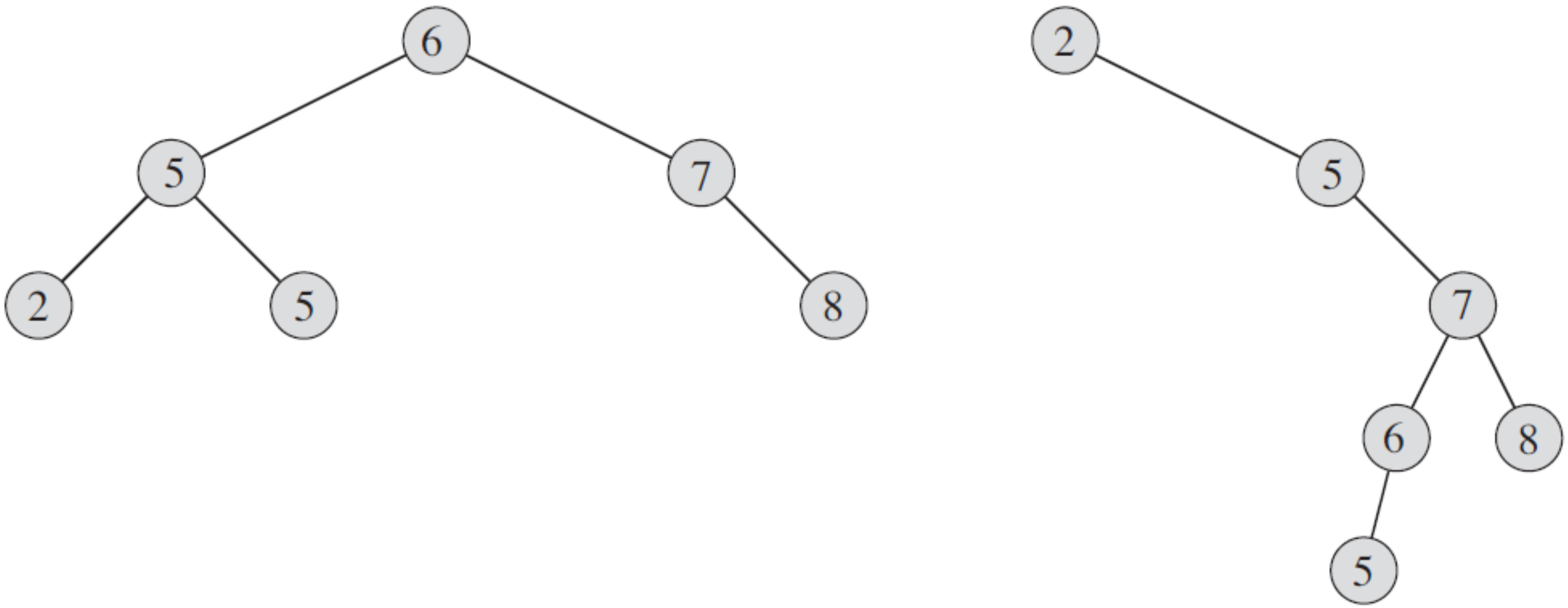
  Let x be a node of the BST.

  If y is a node in the left subtree of x,
  then y.key $\leq$ x.key.

  If y is a node in the right subtree of x,
  then x.key $\leq$ y.key.

- The idea of a BST data structure is to support efficient dynamic set operations (many in O(h), where h is the tree's height).

Examples



Query: In order visit

- Visit all nodes in order and execute an operation:

Function DFS-Inorder-Visit(Node n)

```
if n = NIL then return;
DFS-Inorder-Visit(n.left) ;
n.Operation() ;
DFS-Inorder-Visit(n.right)
```

- The operation could, e.g., be printing the key.
- This tree traversal is also referred to as in-order tree walk.
- Running time (n = number of nodes):
  O(nk) when assuming that the operation is in O(k).

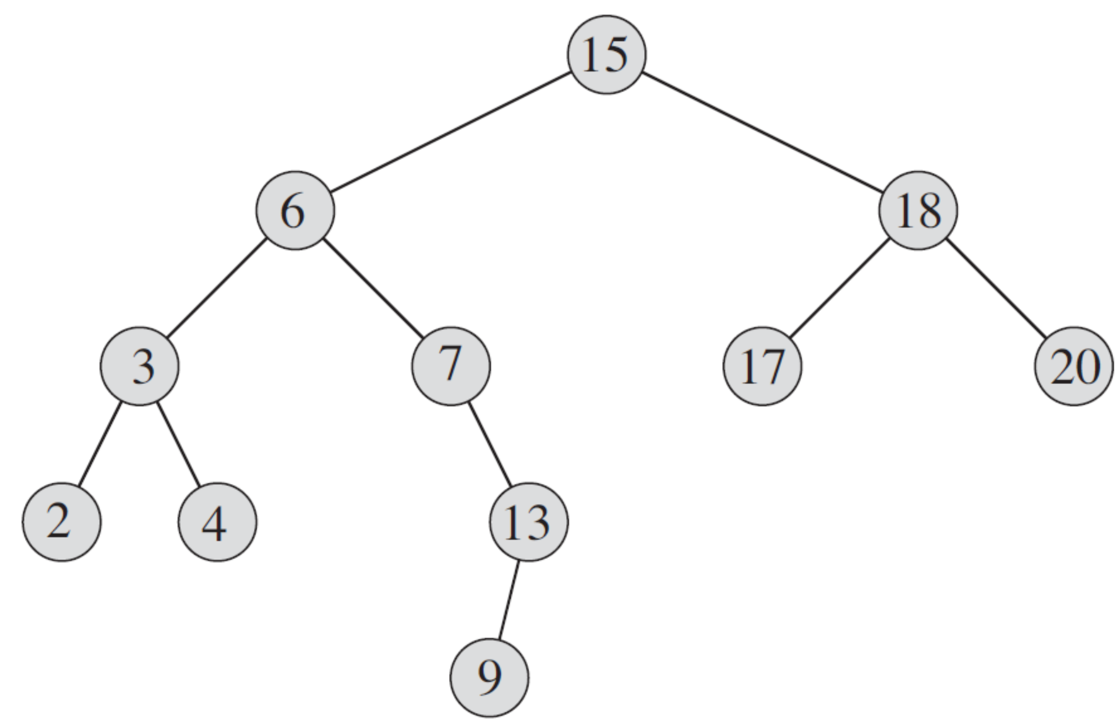
Query: Searching

- Recursive tree search:

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

- Iterative tree search:

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

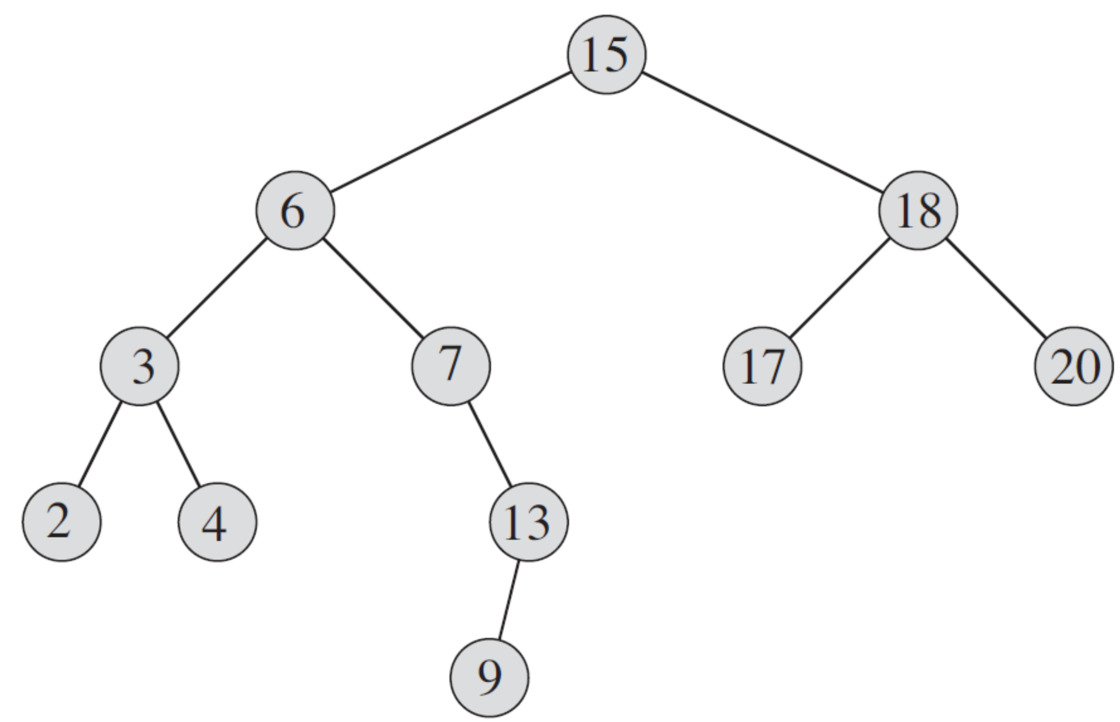


Running time: O(h).

Query: Finding minimum / maximum

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

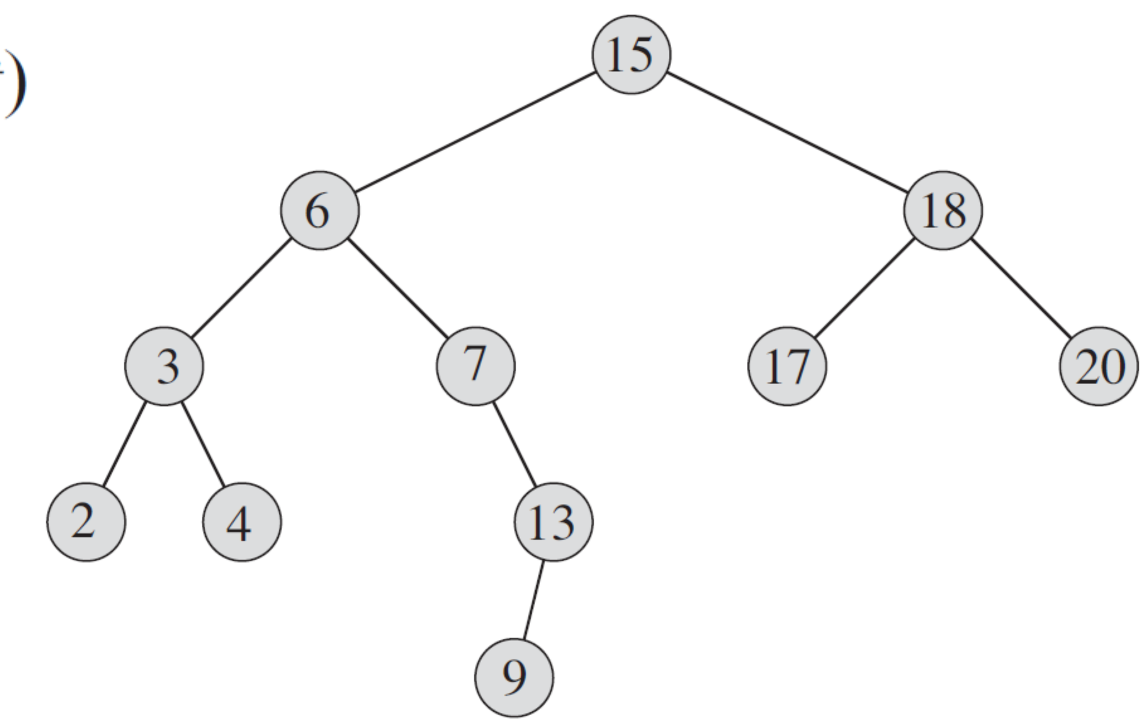
```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```



Running time: O(h).

Query: Finding successor (in order)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

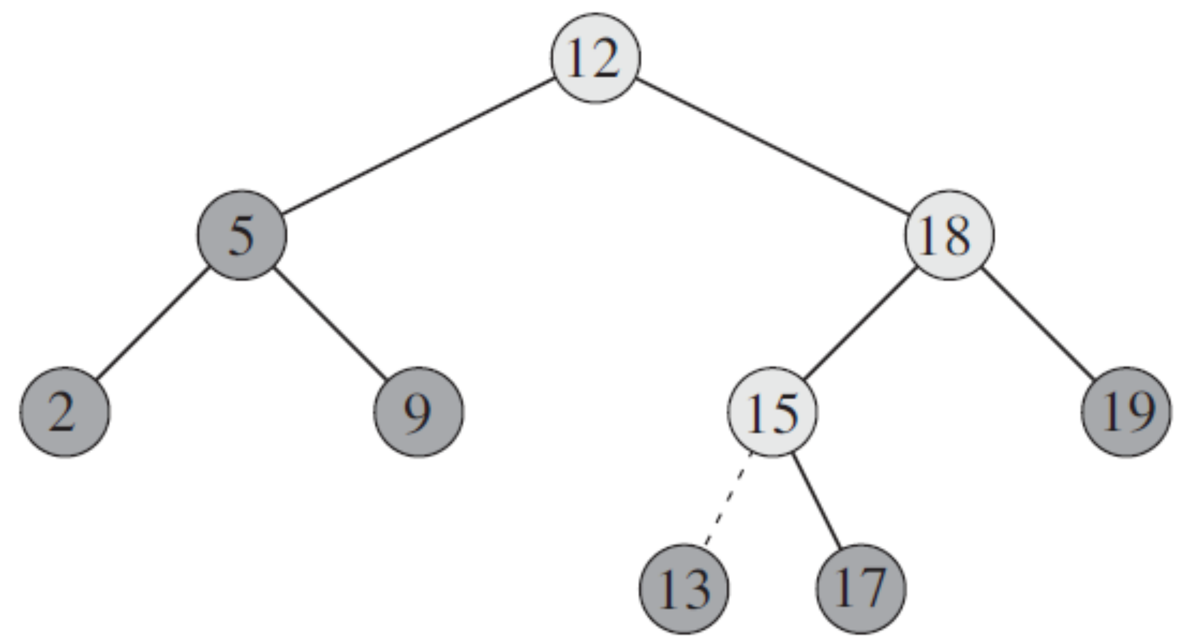


Running time: O(h).

Modify operation: Insertion (in order)

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```



Running time: O(h).

Modify operation: Transplant

- Replaces a subtree rooted at node u with a subtree rooted at node v.

$\text{TRANSPLANT}(T, u, v)$

```
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

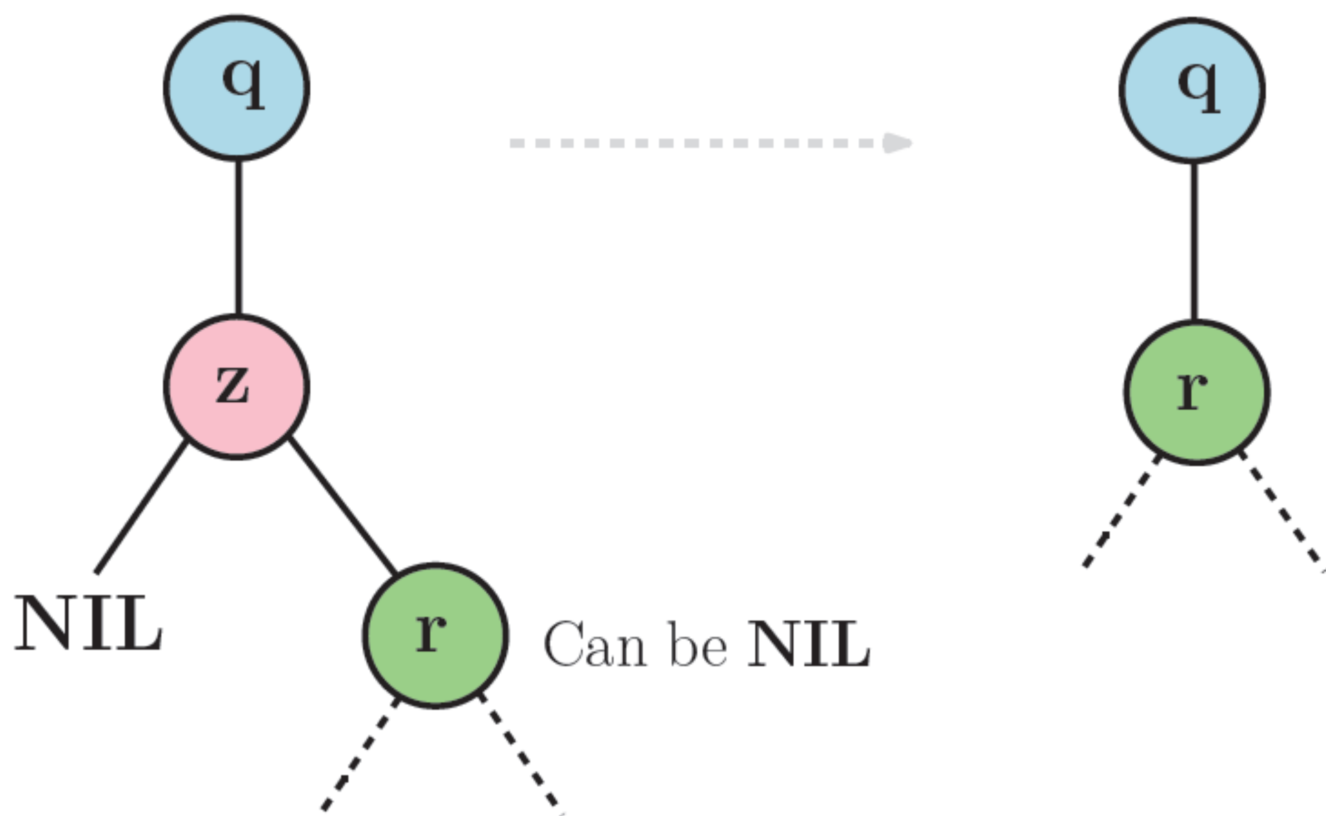
Remarks:

- u.p can be nil.
- v can be nil.

Running time: O(1).

Modify operation: Deletion

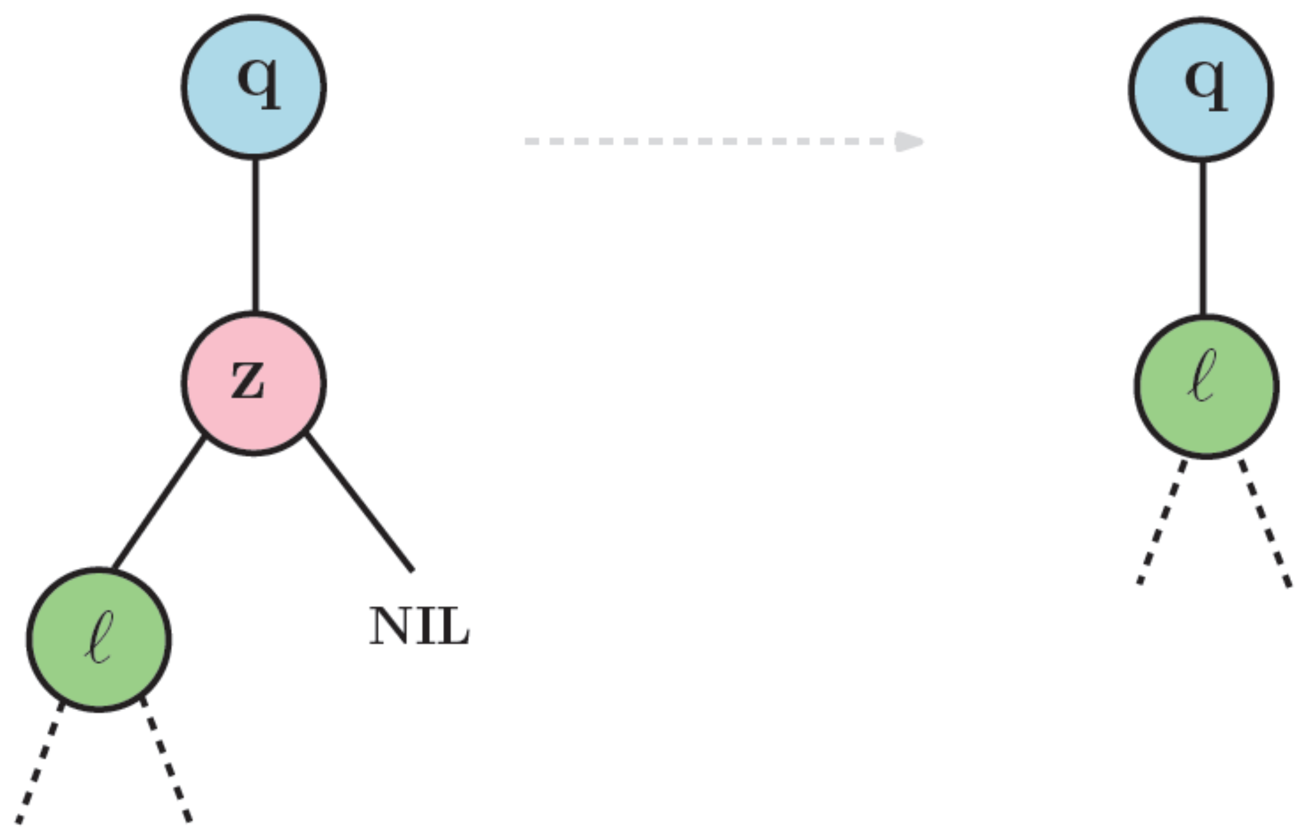
- Case 1:
  Deleted node z has no or only right child.




```
if z.left == NIL
    TRANSPLANT(T, z, z.right)
```

Modify operation: Deletion

- Case 2:
  Deleted node z has only left child.

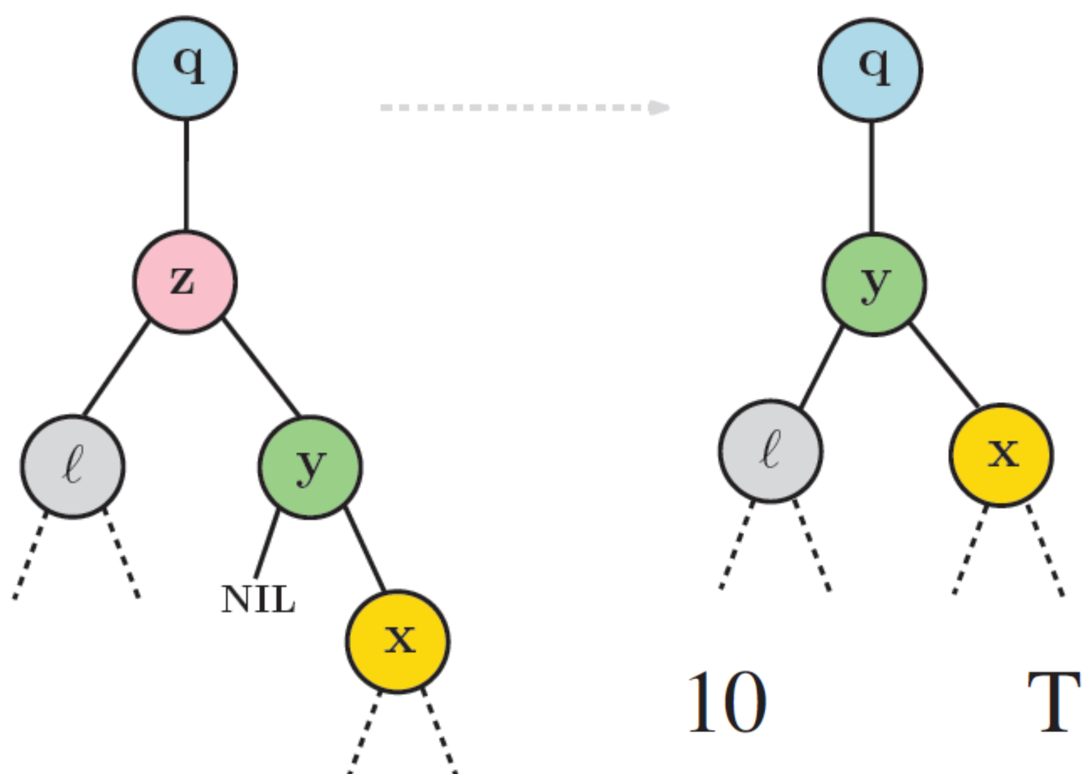


Remark: For both cases, it does not matter whether z is q.left or q.right.

$$
\begin{array}{l}
\textbf{elseif } z.\mathit{right} == \text{NIL} \\
\qquad \text{TRANSPLANT}(T, z, z.\mathit{left})
\end{array}
$$

Modify operation: Deletion

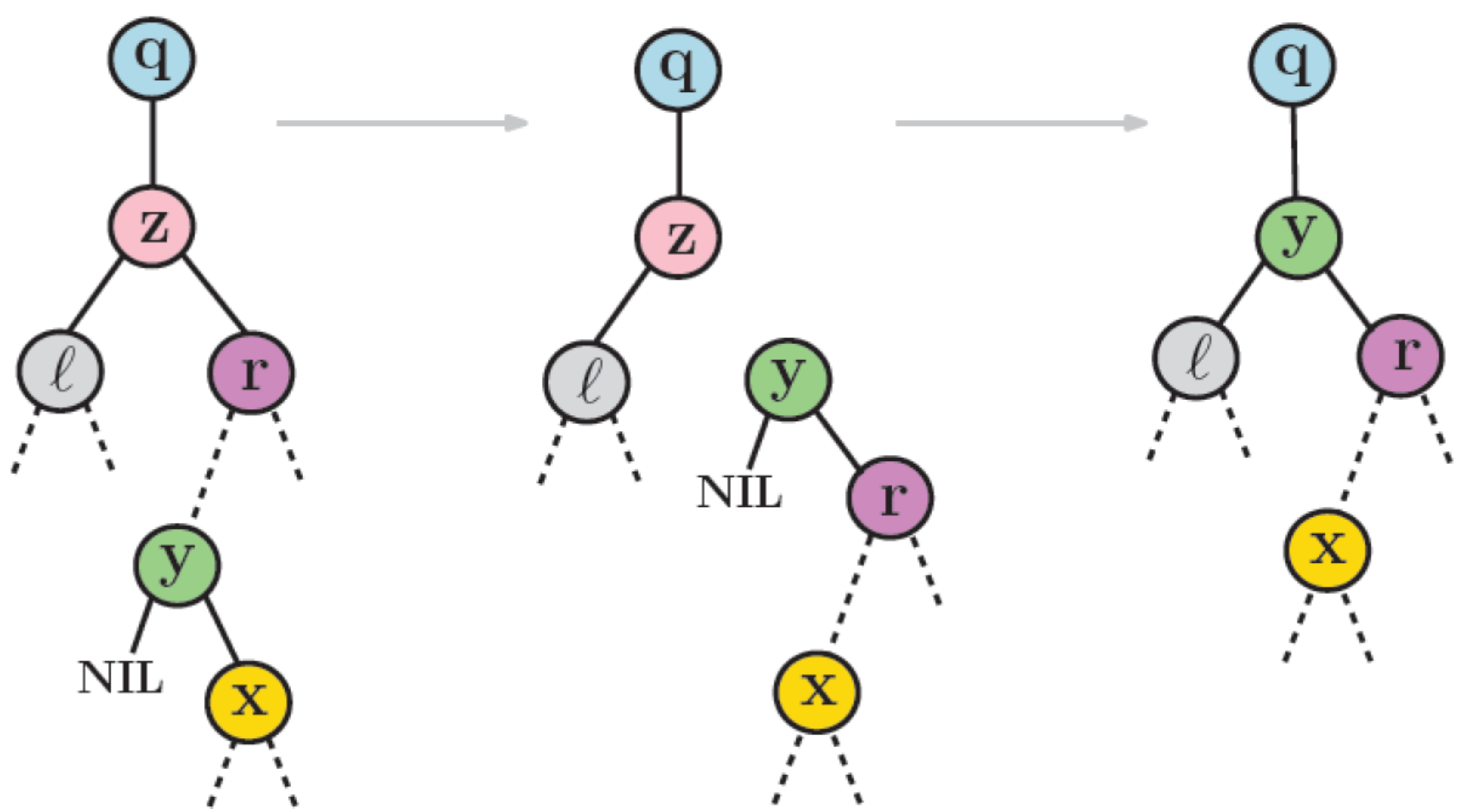
- Case 3a:
  Deleted node z has both children and Successor (z) = z.right.




```
TRANSPLANT(T, z, y)
y.left = z.left
y.left.p = y
```

Modify operation: Deletion

- Case 3b:
  Deleted node z has both children and
  Successor (z) = y ≠ z.right.



Modify operation: Deletion

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

Running time: O(h).

Summary

- BST provides all basic dynamic set operations in O(h) running time, including:
  - Search
  - Minimum
  - Maximum
  - Predecessor
  - Successor
  - Insert
  - Delete
- Hence, BST operations are fast if h is small, i.e., if the tree is balanced. Then, O(h) = O(lg n).