Introduction to Data Structures and Algorithms

Chapter: **Elementary Data Structures(1)**

Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl Informatik 7 (Prof. Dr.-Ing. Reinhard German)
Martensstraße 3, 91058 Erlangen

Elementary Data Structures

Overview on simple data structures
for representing dynamic sets of data records

- Main operations on these data structures are
  - **Insertion** and **deletion** of an element
  - **searching** for an element
  - finding the **minimum** or **maximum** element
  - finding the **successor** or the **predecessor** of an element
  - And similar operations …

- These data structures are often implemented using **dynamically allocated objects** and **pointers**

Elementary Data Structures

Typical Examples of Elementary Data Structures

- Array

- Stack
- Queue
- Linked List
- Tree

Stack

- A **stack** implements the LIFO (last-in, first-out) policy
  - like a stack of plates, where you can either place an extra plate at the top or remove the topmost plate
- For a stack,
  - the **insert** operation is called **Push**
  - and the **delete** operation is called **Pop**

Where are Stacks used?

- A *call stack* that is used for the proper execution of a computer program with subroutine or function calls

- Analysis of *context free languages* (e.g. properly nested brackets)
  - Properly nested: (()(()())), Wrongly nested: (()((())

- Reversed Polish notation of terms
  - Compute 2 + 3*5 ⇨ 2 Push 3 Push 5 * +

Properties of a Stack

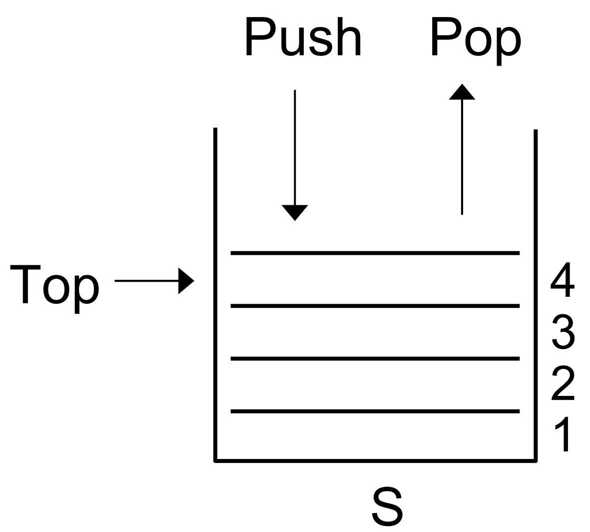
- Stacks can be defined by axioms based on the stack operations, i.e. a certain data structure is a stack if the respective axioms hold

- For illustration some examples for such axioms - the "typical" axioms are
(where S is a Stack which can hold elements x of some set X)

  - If not full(S): Pop(S) o (Push(S,x)) = x for all $x \in X$
  - If not empty(S): Push(S, Pop(S)) = S

Typical Implementation of a Stack

- A typical implementation of a stack of size n is based on an **array** S[1…n]
  ⇨ so it can hold at most n elements
- top(S) is the index of the most recently inserted element
- The stack consists of elements S[1 … top(S)], where
  - S[1] is the element at the bottom of the stack,
  - and S[top(S)] is the element at the top.
- The unused elements S[top(S)+1 … n] are not in the stack



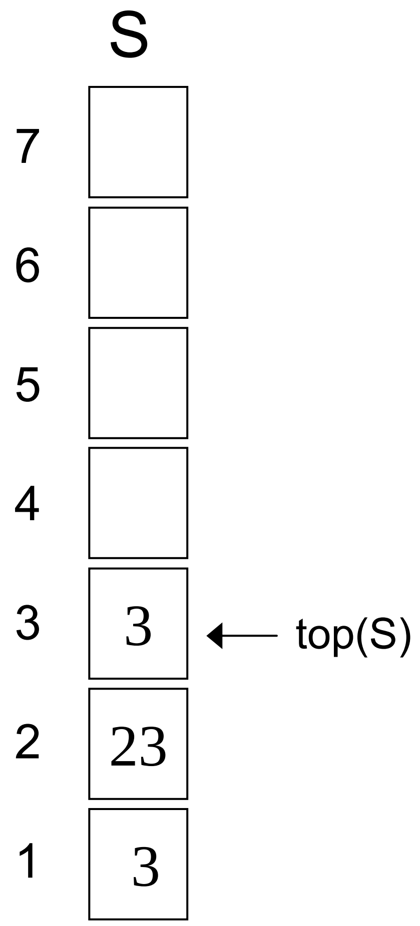
Stack

- If top(S) = 0 the stack is empty ⇨ no element can be popped
- If top(S) = n the stack is full ⇨ no further element can be pushed

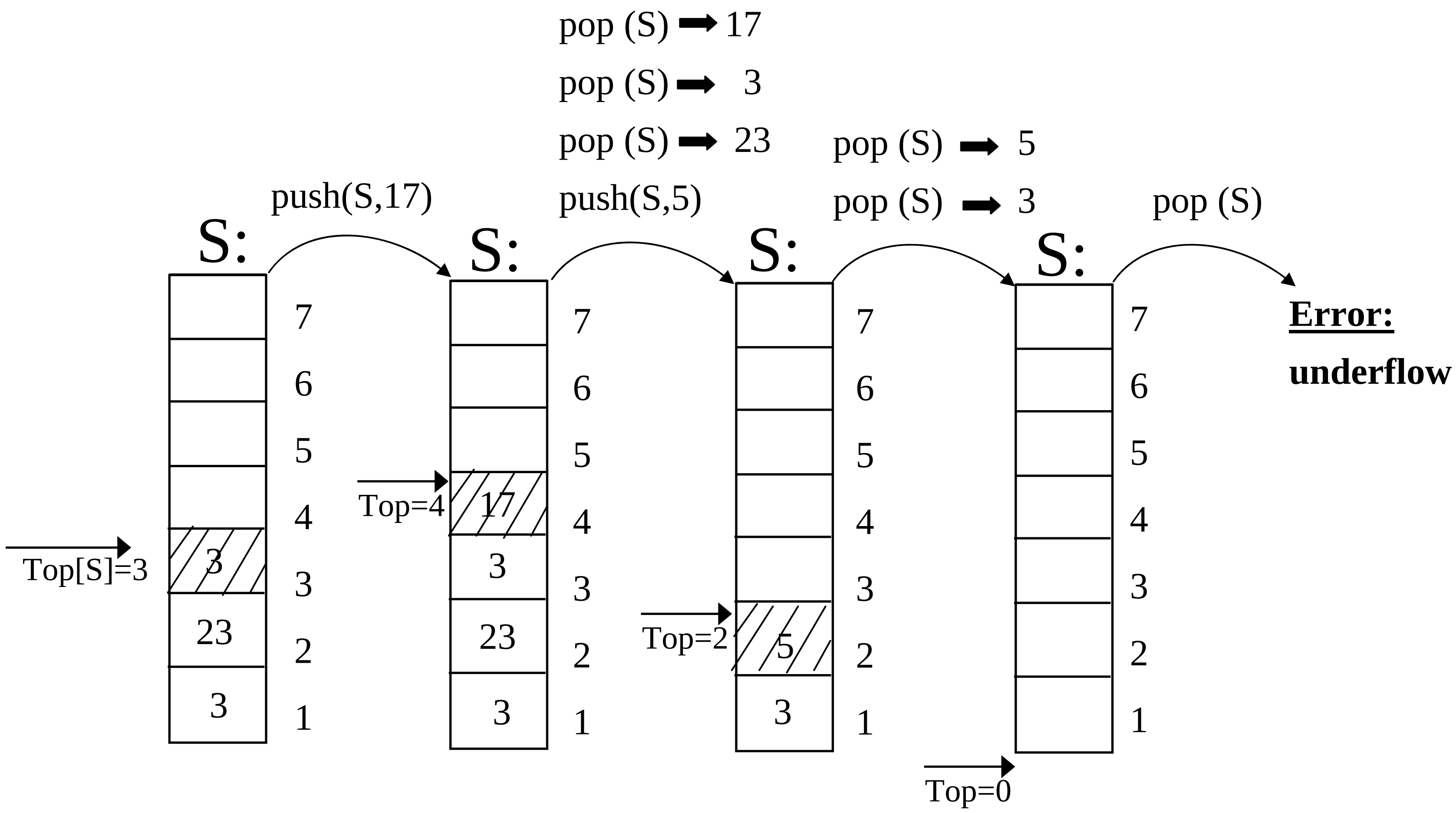
Example (Stack Manipulation)

Start with stack given,
denote changes of “stack state”

- Push(S, 17)
- Pop(S), Pop(S), Pop(S), Push(S, 5)
- Pop(S), Pop(S)
- Pop(S)



Elementary Data Structures



Pseudo Code for Stack Operations

- Number of elements

```
NumElements (S)
    return top[S]
```

Pseudo Code for Stack Operations

- Test for emptiness

```
Stack_Empty(S)
    if top[S]=0
       then return true
       else return false
```

- Test for “stack full”

```
Stack_Full (S)
    if top[S]=n
       then return true
       else return false
```

Pseudo Code for Stack Operations

- Pushing and Popping

This pseudo code contains error handling functionality

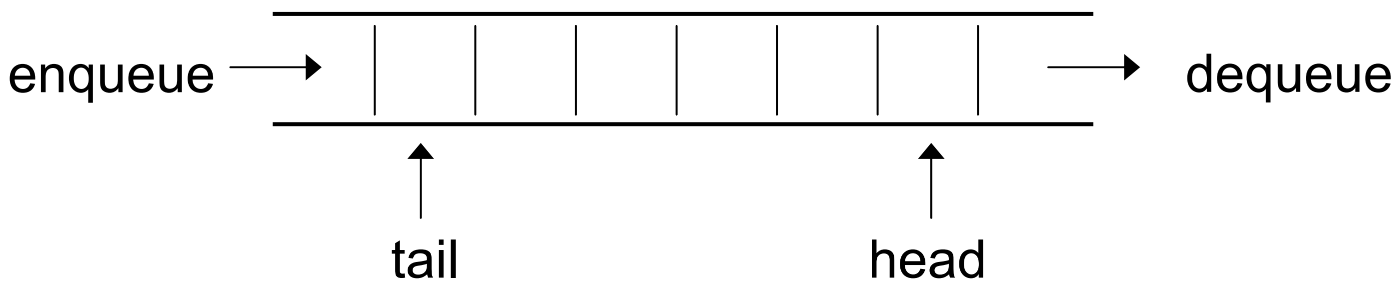
```
Push(S,x)
  if Stack_Full(S)
     then error "overflow"
     else top[S] := top[S]+1
          S[top[S]] := x

Pop(S)
  if Stack_Empty(S)
     then error "underflow"
     else top[S] := top[S]-1
          return S[top[S]+1]
```

Pseudo Code for Stack Operations

- **(Asymptotic) Runtime**
  - **NumElements**:
    number of operations independent of size n of stack
    ➪ constant ➪ O(1)
  - **Stack_Empty** and **Stack_Full**:
    number of operations independent of size n of stack
    ➪ constant ➪ O(1)
  - **Push** and **Pop**:
    number of operations independent of size n of stack
    ➪ constant ➪ O(1)

Queue

- A **queue** implements the FIFO (first-in, first-out) policy
  - Like a line of people at the post office or in a shop



- For a queue,
  - the insert operation is called **Enqueue**
    (=> place at the tail of the queue)
  - and the delete operation is called **Dequeue**
    (=> take from the head of the queue)

Where are Queues used?

- In multi-tasking systems (communication, synchronization)
- In communication systems (store-and-forward networks)
- In servicing systems (queue in front of the servicing unit)
- Queuing networks (performance evaluation of computer and communication networks)

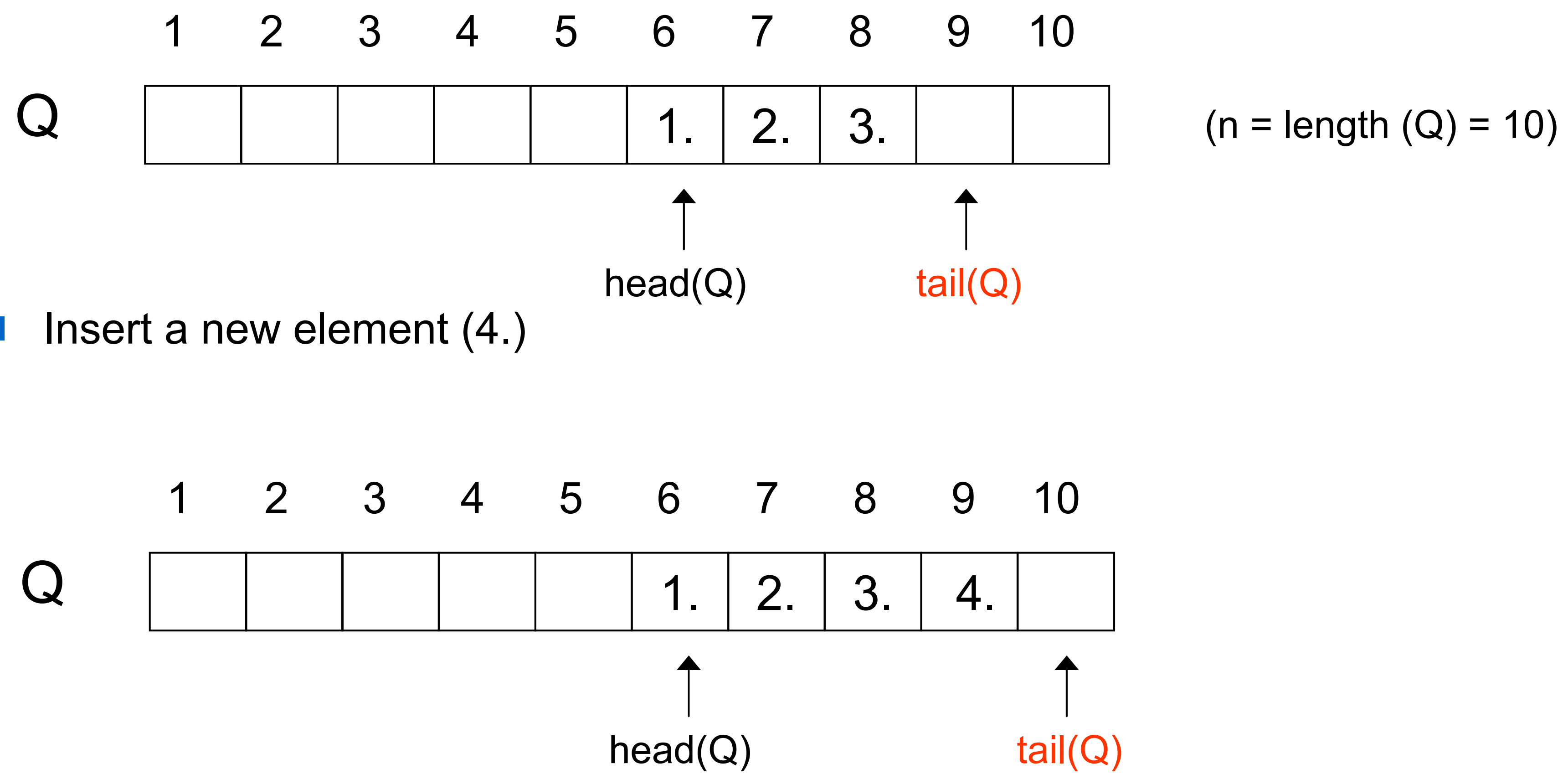
Typical Implementation of a Queue

- A typical implementation of a queue consisting of at most n-1 elements is based on an **array** Q[1 … n]

- Its attribute **head(Q)** points to the head of the queue.
- Its attribute **tail(Q)** points to the position where a new element will be inserted into the queue (i.e. one position behind the last element of the queue).

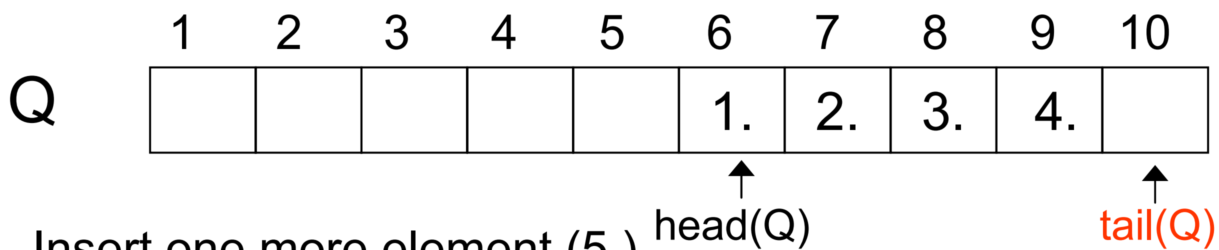
- The elements in the queue are in positions head(Q), head(Q)+1, …, tail(Q)-1, where we wrap around the array boundary in the sense that Q[1] immediately follows Q[n]

Example (1)

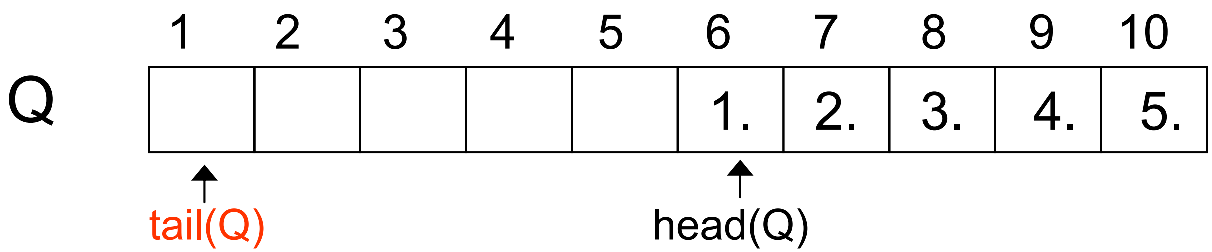


- Insert a new element (4.)

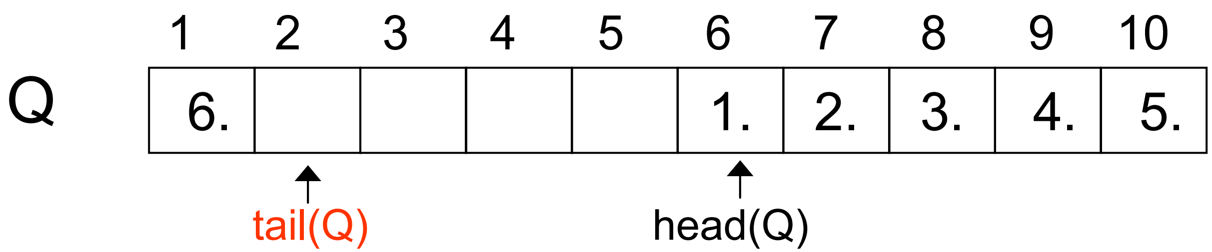
Example (2)



- Insert one more element (5.)



- And again: Insert one more element (6.)



Typical Implementation of a Queue

- Number of elements in queue
  - If tail > head:
    NumElements(Q) = tail - head
  - If tail < head:
    NumElements(Q) = tail – head + n
  - If tail = head:
    NumElements(Q) = 0
  - Initially: head[Q] = tail[Q] = 1

- Position of elements in queue
  - The x. element of a queue Q (1 ≤ x ≤ NumElements(Q)
    is mapped to array position

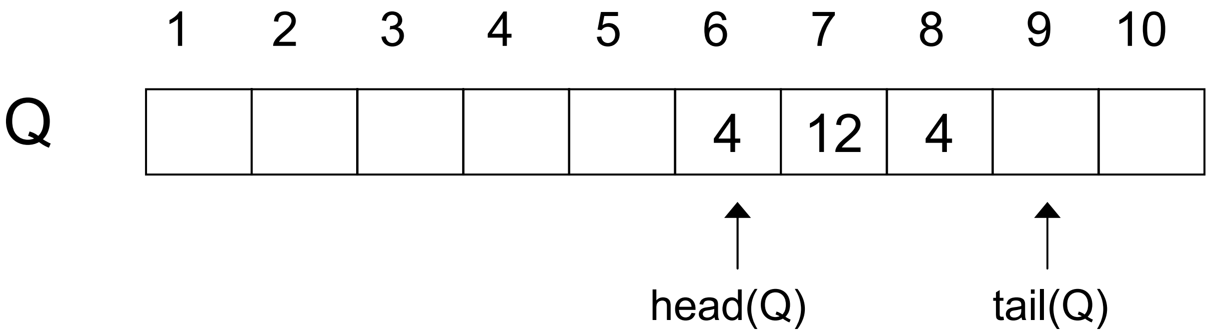
    head(Q) + (x - 1)   if x ≤ n – head +1 (no wrap around)
    head(Q) + (x - 1) - n   if x > n – head +1 (wrap around)

Typical Implementation of a Queue

- Remark:
  - A queue implemented by a n-element array can hold at most n-1 elements
  - otherwise we could not distinguish between an empty and a full queue

- A queue Q is empty: (⇔ NumElements(Q) = 0)
  - if head(Q) = tail(Q)
- A queue Q is full: (⇔ NumElements(Q) = n-1)
  - if head(Q) = (tail(Q) + 1) (head(Q) > tail(Q))
  - if head(Q) = (tail(Q) - n + 1) (head(Q) < tail(Q))

Example (Queue Manipulation)



Start with queue given, denote changes of “queue state”

- Enqueue(Q, 2), Enqueue(Q, 3), Enqueue(Q, 7)
- Dequeue(Q)

Queue Operations

- Enqueue and Dequeue

This pseudo code does not contain error handling functionality (see stack push and pop)

```
Enqueue(Q,x)
  Q[tail[Q]] := x
  if tail[Q]=length[Q]
     then tail[Q] := 1
     else tail[Q] := tail[Q]+1
```

Precondition: queue not full

```
Dequeue(Q)
  x := Q[head[Q]]
  if head[Q]=length[Q]
     then head[Q] := 1
     else head[Q] := head[Q]+1
  return x
```

Precondition: queue not empty

Pseudo Code for Queue Operations

- **(Asymptotic) Runtime**
  - **Enqueue** and **Dequeue**:
    number of operations independent of size n of queue
    - ⇨ constant
    - ⇨ O(1)

Introduction to
Data Structures and Algorithms

Chapter: **Elementary Data Structures(2)**

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl Informatik 7 (Prof. Dr.-Ing. Reinhard German)
Martensstraße 3, 91058 Erlangen

Elementary Data Structures

Typical Examples of Elementary Data Structures

- Array

- Stack
- Queue
- Linked List
- Tree

Linked List

- In a **linked list**, the elements are arranged in a linear order, i.e. each element (except the first one) has a **predecessor** and each element (except the last one) has a **successor**.
- Unlike an array, elements are not addressed by an index, but by a **pointer** (a reference).

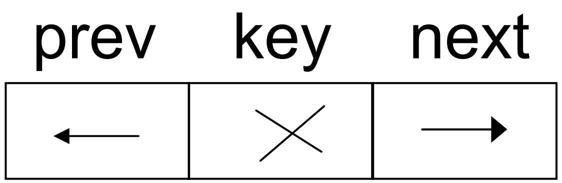
- There are *singly* linked lists and *doubly* linked lists.
- A list may be sorted or unsorted.
- A list may be circular (i.e. a ring of elements).

- Here we consider mainly unsorted, doubly linked lists

Linked List

- Each element x of a (doubly) linked list has three fields
  - A pointer **prev** to the previous element
  - A pointer **next** to the next element
  - A field that contains a **key** (value of a certain type)
  - Possibly a field that contains satellite data (ignored in the following)

One element x :
(consist of 3 fields)

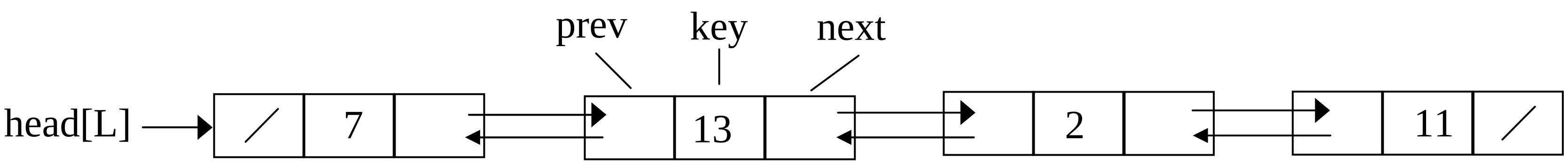


- Pointer fields that contain no pointer pointing to another element contain the special pointer **NIL** (\)
- The pointer **head**[L] points to the first element of the linked list
- If head[L] = NIL the list L is an **empty list**

Linked List

- In a linked list, the insert operation is called **List_Insert**, and the delete operation is called **List_Delete**.
- In a linked list we may search for an element with a certain key k by calling **List_Search**.

Linked List Example**:** dynamic set {11, 2 ,7 , 13}



Notice:

prev[head] = NIL and *next*[tail] = NIL

Some Examples for the Use of Linked Lists

- Lists of passengers of a plane or a hotel
- Card games (sorting cards corresponding to a certain order, inserting new cards into or removing cards out of the sequence)
- To-do lists (containing entries for actions to be done)
- Hash Lists (⇨ Hashing, dealt later in this lecture)

Searching a Linked List

- The procedure List_search (L, k) finds the first element with key k in list L and returns a pointer to that element.
- If no element with key k is found, the special pointer NIL is returned.

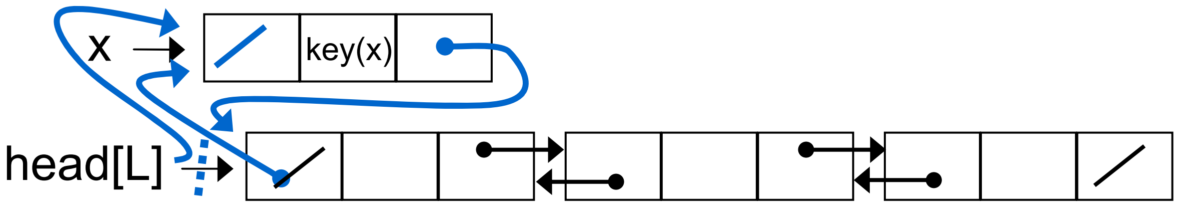
```
List_Search(L,k)
  x := head[L]
  while x!=NIL and key[x]!=k do
    x := next[x]
  return x
```

- It takes at most Θ(n) time to search a list of n objects (linear search)

Inserting into a Linked List

- The procedure List_insert(L,x) inserts a new element x as the new head of list L

```
List_Insert(L,x)
  next[x] := head[L]
  if head[L]!=NIL then
     prev[head[L]] := x
  head[L] := x
  prev[x] := NIL
```



- The runtime for List_Insert on a list of length n is constant (O(1))

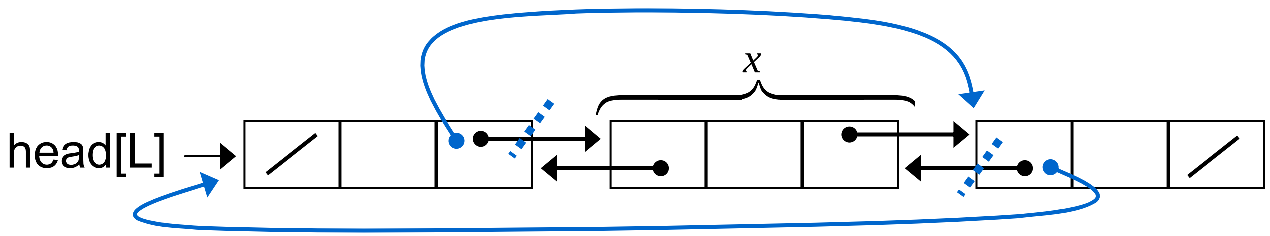
Deleting from a Linked List

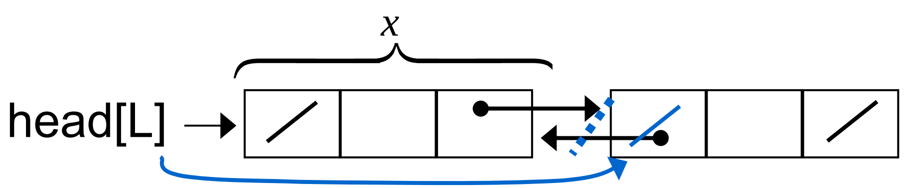
- The procedure List_Delete (L, x) removes an element x from the linked list L, where the element is given by a pointer to x.
- If you want to delete an element given by its key k, you have to compute a pointer to this element (e.g. by using List_search(L, k))

```
List_Delete(L,x)
  if prev[x]!=NIL                ⇨ x not the first element
     then next[prev[x]] := next[x]
     else head[L] := next[x]
  if next[x]!=NIL                ⇨ x not the last element
     then prev[next[x]] := prev[x]
```

Deleting from a Linked List

```
     List_Delete(L,x)
a)     if prev[x]!=NIL
          then next[prev[x]] := next[x]
b)        else head[L] := next[x]
       if next[x]!=NIL
          then prev[next[x]] := prev[x]
```

a) 

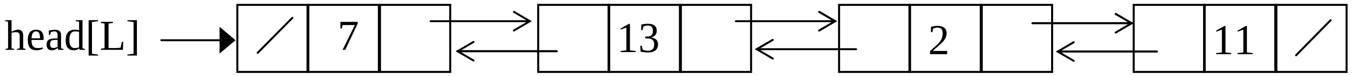
b) 

Deleting from a Linked List

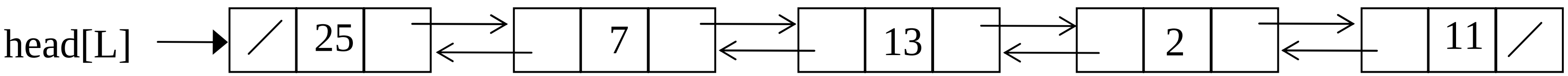
- The runtime for List_Delete on a list of length n is constant (O(1))

- If you want to delete an element with a certain key, you must first find that element by executing List_Search, which takes Θ(n) time in the worst case

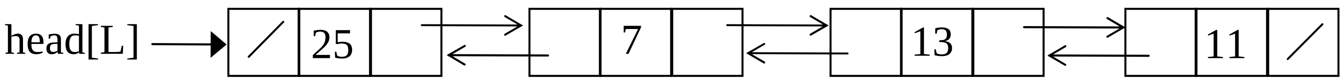
Inserting and deleting :



List_insert (L,x) with key[x] = 25



List_Delete (L,x) where x points to element with key[x] = 2



Tree

- Any data structure consisting of elements of the same type can be represented with the help of pointers (in a similar way as we implemented lists).
- Very important examples of such data structures are **trees**.
  - Trees are graphs that contain no cycle: every non-trivial path through a tree starting at a node and ending in the same node, does traverse at least one edge at least twice.
- There exist many kinds of trees. Examples are:
  - Binary trees
  - Trees with unbounded branching
  - Binary search trees
  - Red-black trees

Some Examples for the Use of Trees

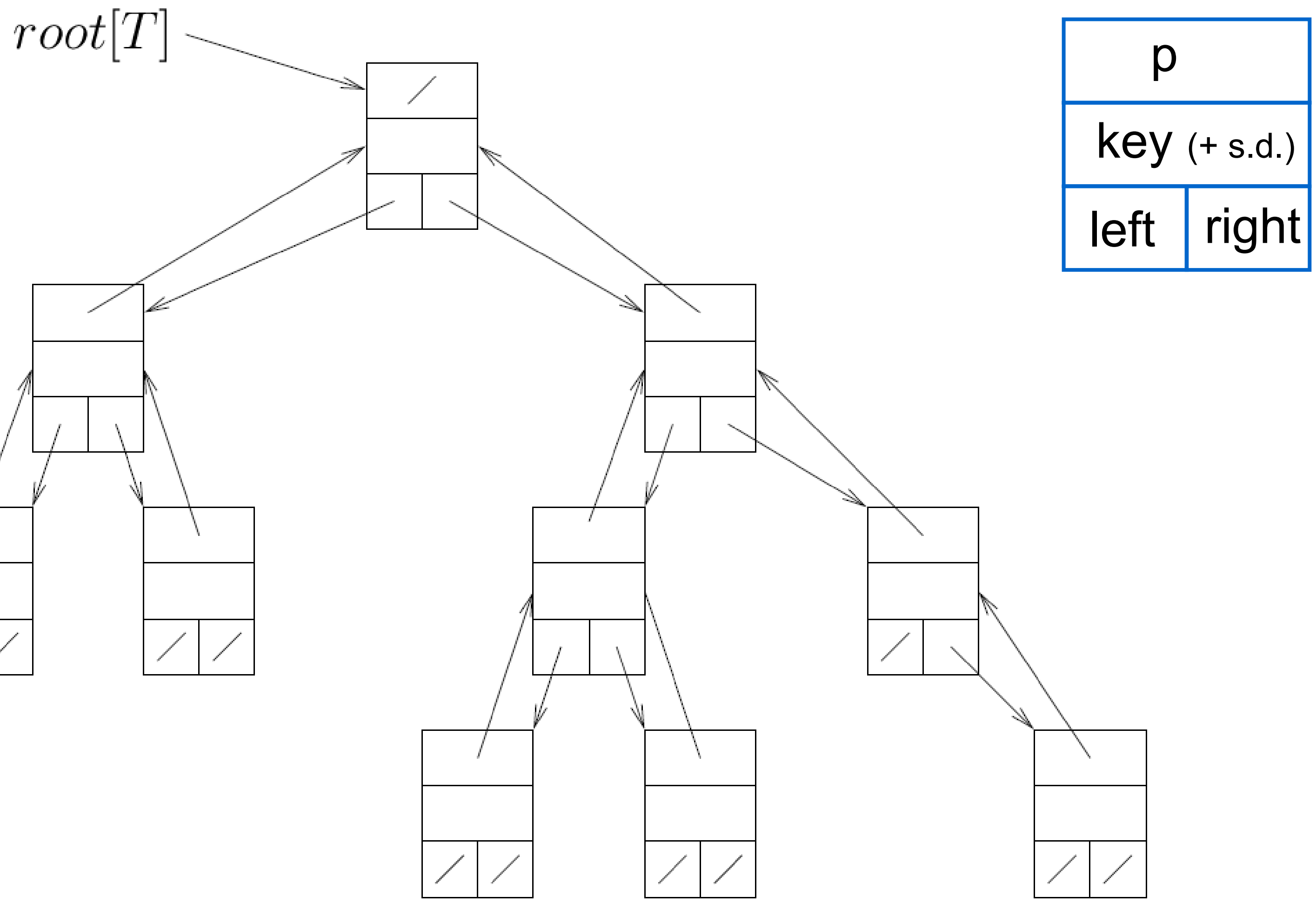
- Systematically exploring various ways of proceeding (e.g. in chess or planning games)
- Morse trees (coding trees)
- Heaps (⇨ heap sort)
- Search trees

Tree

- A **binary tree** consists of nodes with the following fields
  - A **key** field
  - Possibly some satellite data (ignored in the following)
  - Three pointers **p**, **left** and **right** pointing to the parent node, left child node and right child node
- Be x an element (or node) of a tree
  - If p[x] = NIL ⇨ x represents the **root** node
  - If both left[x] = NIL and right[x] = NIL

    ⇨ x represents a **leaf** node
- For each tree T there is a pointer root[T] that points to the root of T
- If root[T] = NIL, the tree T is empty

Binary Tree (Example)



P-nary Trees

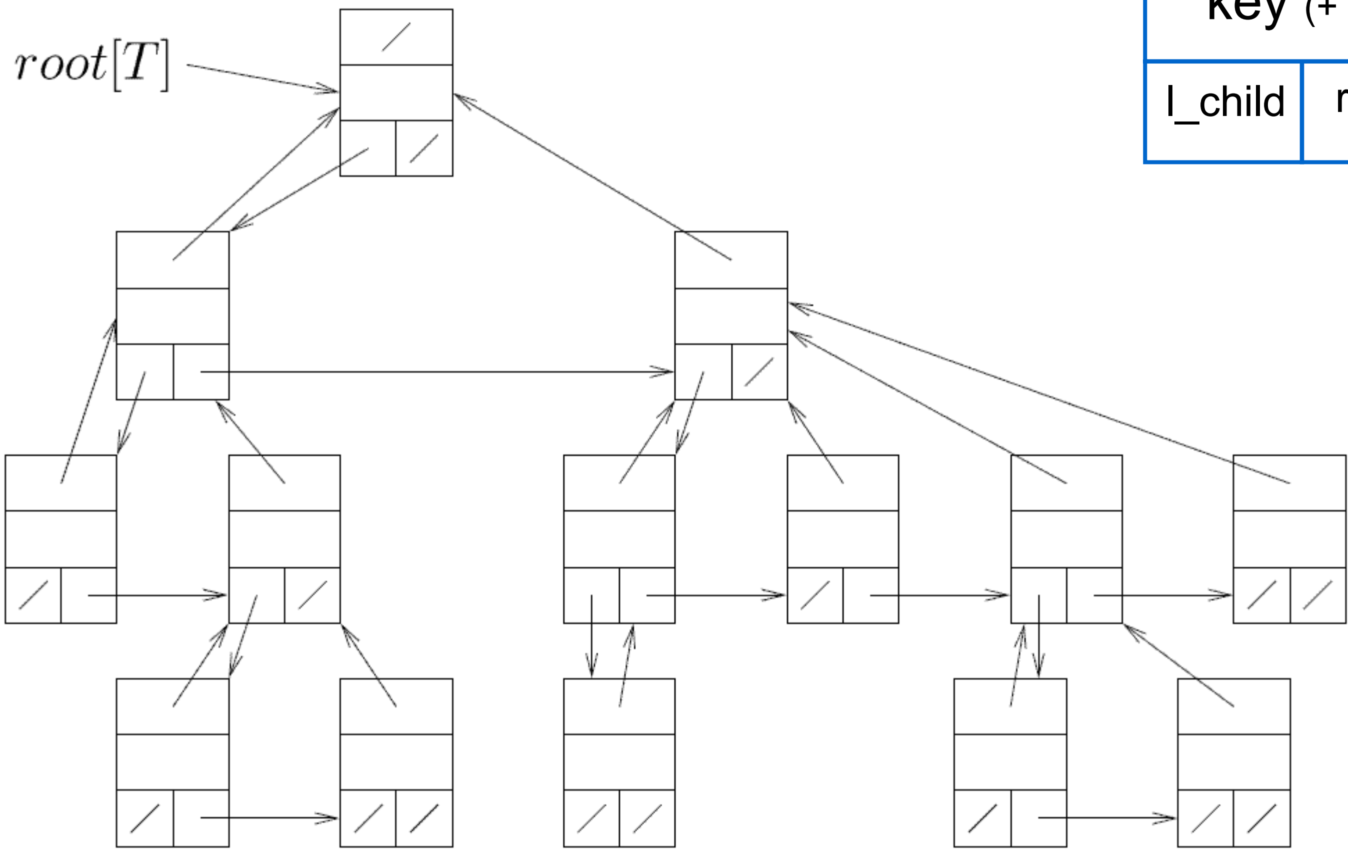
- The above scheme can be extended to any class of trees where the number of children is bounded by some constant $k: \ child_1, \cdots, child_k \quad k \in \mathbb{N}$
  - a bit of memory space may be wasted for pointers which are not actually used

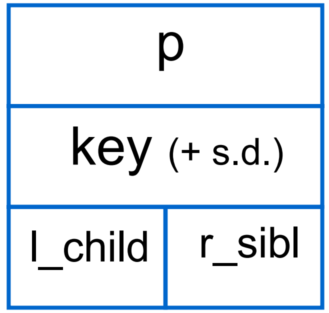
Trees with unbounded branching

- A **tree with unbounded branching** (if no upper bound on the number of a node's children is known a priori) can be implemented by the following scheme:
  - Each node has a key field (and possibly some satellite data),
  - and three pointers **p**, **left_child** and **right_sibling**
  - In a leaf node, left_child=NIL
  - If a node is the rightmost child of its parent, then right_sibling=NIL

Elementary Data Structures

Unbounded Tree (Example)





Unbounded Tree (Example)