Algorithms and Data Structures Conditional Course

Lecture 3

Abstract Data Types, Simple Data Structures, Binary Search

Fabian Kuhn
Algorithms and Complexity

UNI FREIBURG

Data Structures

Algorithms

- How to solve a given problem efficiently?
- Goal: smallest possible complexity
  - small runtime / small memory usage
  - asymptotically, dependent on the problem size

Data Structures

- How to save data such that the access becomes as efficient as possible?
- Depends on the types of operations we have to support!
- Good data structures necessary to obtain fast algorithms
- One needs fast algorithms to carry out data structure operations optimally

Abstract Data Types

Abstract Data Type:

- Specification which kind of data one can support
- Specification of the operations to access the data
  - including the semantics of the operations

Data Structure:

- A concrete way of implementing an abstract data type
- Depending on the implementation, the same operations might have different runtimes (complexities)

We will now first briefly discuss the most important abstract data types…

Abstract Data Types: Examples

Array:

- holds a collection of elements (of the same type)

Operations:

- *create(n)* : creates an array of length n
- *A.get(i)* : returns the element at position i
- *A.set(x, i)* : writes element x to position i
- *A.size()* : returns the length of the array (not always available)

For dynamic arrays (can change size):

- *A.append(x)* : appends element x at the end
- *A.deleteLast()* : deletes last element

Abstract Data Types: Examples

Dictionary: (also: maps, associative arrays)

- holds a collection of elements where each element is represented by a unique key

Operations:

- *create* : creates an empty dictionary
- *D.insert(key, value)* : inserts a new *(key,value)*-pair
  - If there already is an entry with the same *key*, the old entry is replaced
- *D.find(key)* : returns entry with key *key*
  - If there is such an entry (returns some default value otherwise)
- *D.delete(key)* : deletes entry with key *key*

Abstract Data Types: Examples

Dictionary:

Additional possible operations:

- *D.minimum()* : returns smallest *key* in the data structure
- *D.maximum()* : returns largest *key* in the data structure
- *D.successor(key)* : returns next larger *key*
- *D.predecessor(key)* : returns next smaller *key*
- *D.getRange(k1, k2)* : returns all entries with keys in the interval *[k1,k2]*

Abstract Data Types: Examples

Queue:

- Holds a collection (sequence) of elements

Operations:

- *create* : creates an empty queue
- *Q.enqueue(x)* : appends element x at the end
- *Q.dequeue()* : returns element at front element and removes it
- *Q.isEmpty()* : Is the queue empty?

Is also called FIFO queue (FIFO = first in first out)

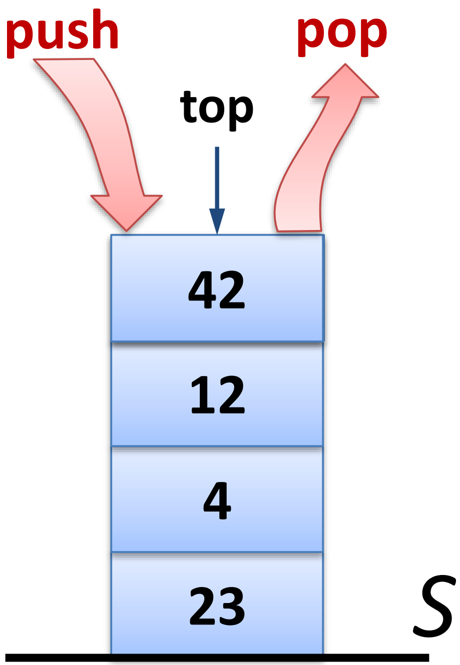
Abstract Data Types: Examples

Stack:

- Holds a collection (sequence) of elements

Operations:

- *create* : creates an empty stack
- *S.push(x)* : puts an element *x* on the stack
- *S.pop()* : returns and deletes top element of stack
- *S.isEmpty()* : Is the stack empty?



Is also called LIFO queue (LIFO = last in first out)

Abstract Data Types: Examples

Heap / Priority Queue :

- Holds a collection of *(key,value)* pairs

Operations:

- *create()* : creates an empty heap
- *H.insert(x, key)* : inserts element *x* with key *key*
- *H.getMin()* : returns element with smallest key
- *H.deleteMin()* : deletes element with smallest key
  - *H.getMin()* and *H.deleteMin()* have to be consistent
- *H.decreaseKey(x, newkey)* : If *newkey* is smaller than the current key of *x*, the key of x is set to *newkey*

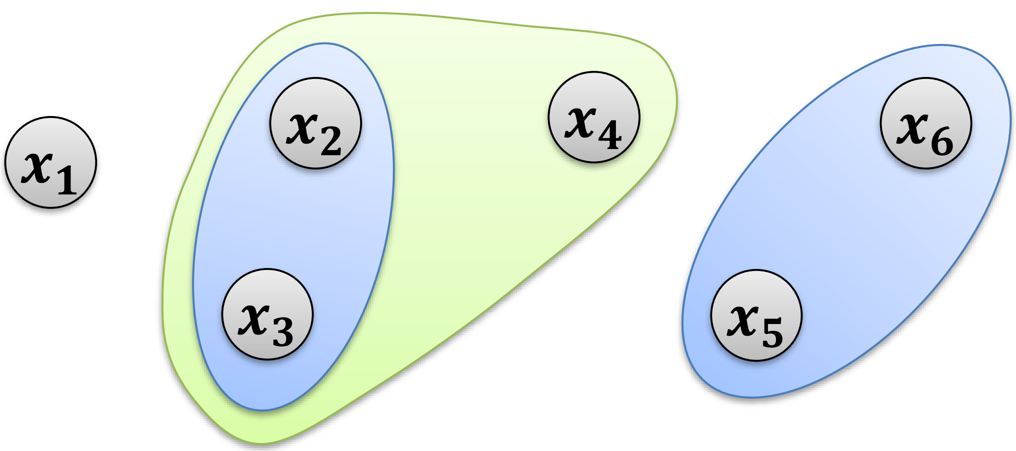
Abstract Data Types: Examples

Union-Find / Disjoint Sets:

- Manages a partition of elements

Operationen:

- *create()* : creates an empty union-find data structure
- *U.makeSet(x)* : adds a set $\{x\}$ to the partition
- *U.find(x)* : returns the set containing element x
- *U.union(S1, S2)* : unites sets $S1$ and $S2$ to set $S1 \cup S2$



Array Implementation of Stack

Let us try to implement the stack data type

- **Operations:** *create, push, pop, isEmpty*
- **Assumption:** Stack only needs to be able to hold *NMAX* elements

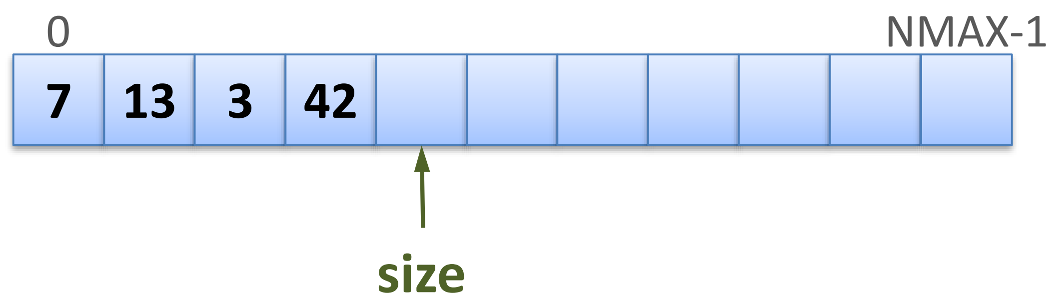
Variables to store the state of the stack:

- $stack$: array of length *NMAX*
- $size$: current number of elements in stack

```
create():
  stack = new array of length NMAX
  size = 0
```

Array Implementation of Stack

```
isEmpty():
   return (size == 0)

S.push(x):
   if (size < NMAX):
      stack[size] = x
      size += 1

S.pop():
   if (size == 0):
      report error (or return default value)
   else:
      size -= 1
      return stack[size]
```



Analysis: Array Implementation of Stack

Runtime (complexity) of the operations:

- create: $O(1)$
  - If we assume that memory can be allocated in $O(1)$ time
- push: $O(1)$
- pop: $O(1)$
- isEmpty: $O(1)$

Disadvantages of the implementation:

- Memory usage (space complexity) : $\Theta(NMAX)$
  - always needs the same amount of memory, no matter how many elements there are on the stack!
- The stack can only hold *NMAX* elements…
- We will see that both these things can be fixed…

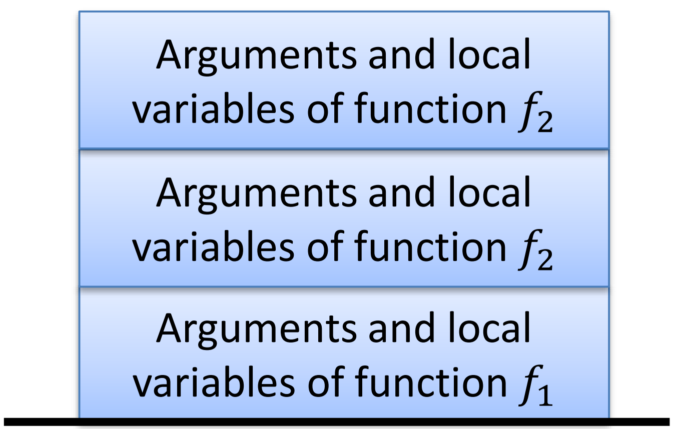
Stack : Applications

- Reversing a sequence: A, B, C

  push(A), push(B), push(C), pop() → C, pop() → B, pop() → A

- Undo operation for editors:
  - Put description of (reversible) edit operations on the stack

- Program stack for function / method calls
  - Remark: With a stack, it is possible to write down recursion explicitly



def $f_1(x, y)$:
  …
  $f_2(z)$
  …

def $f_2(a)$:
  …
  $f_2(b)$
  …

Array Implementation of Queue

Let us try to implement the queue data type

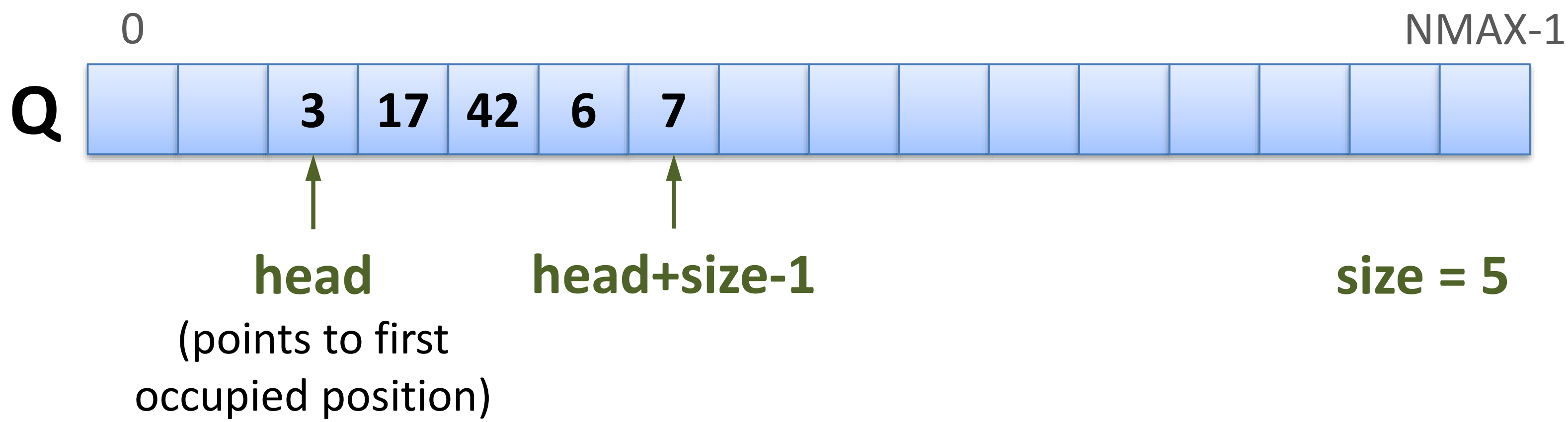
- **Operations:** *create, enqueue, dequeue, isEmpty*
- **Assumption:** Queue only needs to be able to hold *NMAX* elements

Variables to store the state of the queue:

- $queue$: array of length *NMAX*
- $head$: position of the first element (cyclic)
  - if the queue is not empty.
- $size$: number of elements in the queue

```
create:
  queue = new array of length NMAX
  head  = 0
  size  = 0
```

Array Implementation of Queue



- Q.dequeue() returns element at pos. head, if Q is not empty
- Q.enqueue(x) inserts element at pos. head + size

- Array is used cyclically:

Array Implementation of Queue

```
S.isEmpty():
  return (size == 0)

S.enqueue(x):
  if (size < NMAX)
    pos = (head + size) mod NMAX
    queue[pos] = x
    size += 1

S.dequeue():
  if (size == 0)
    report error (or return default value)
  else
    x = queue[head]
    head = (head + 1) mod NMAX
    size = size - 1
    return x
```

Analysis: Array Implementation of Queue

Runtime (complexity) of the operations:

- create: $O(1)$
  - If we assume that memory can be allocated in $O(1)$ time
- enqueue : $O(1)$
- dequeue : $O(1)$
- isEmpty : $O(1)$

Disadvantages of the implementation:

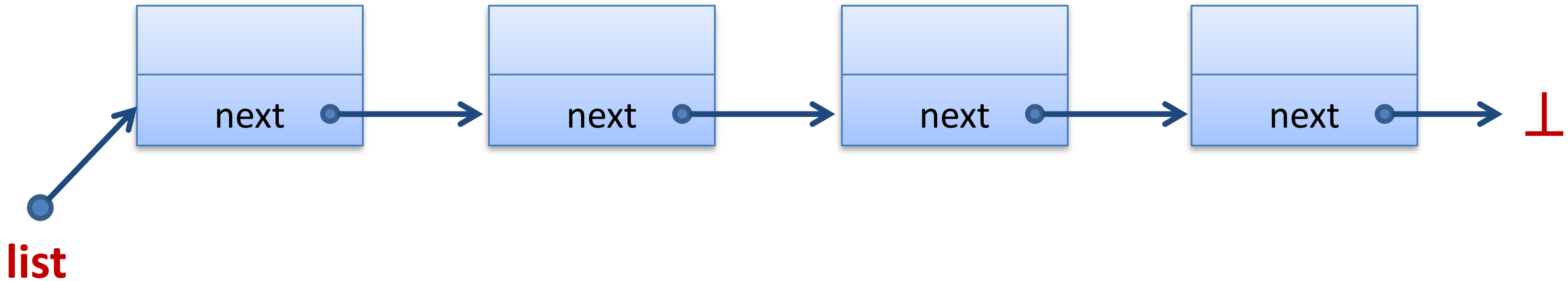
- Memory usage (space complexity) : $\Theta(NMAX)$
  - always needs the same amount of memory, no matter how many elements there are in the queue!
- The queue can only hold up to *NMAX* elements…
- We will see that both these things can be fixed …

Linked Lists

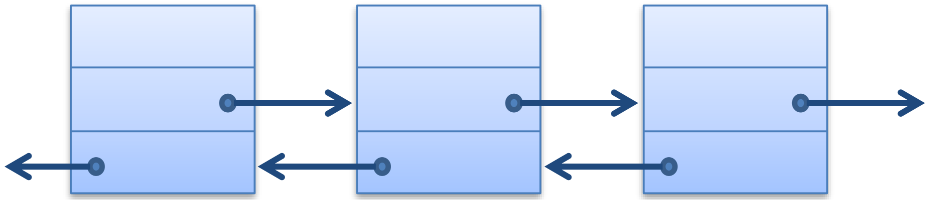
- Data structure to hold a list (sequence) of elements

Array:

Linked list:

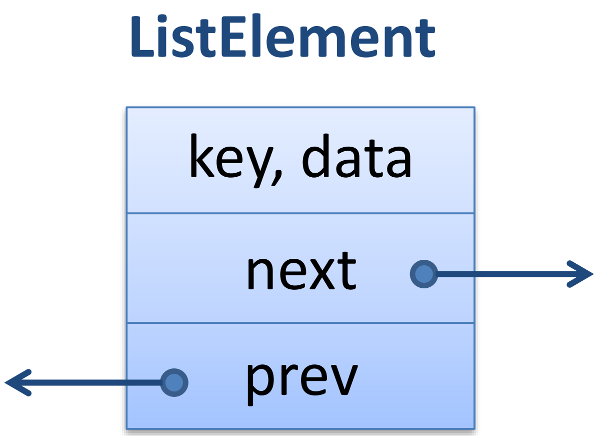


Doubly linked list:



List Elements

- Class to describe list elements

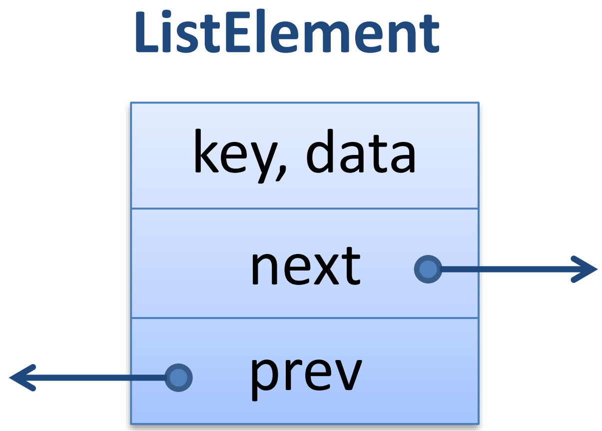


Python:

```
class ListElement:

    def __init__(self, key=0, data=None, next=None, prev=None):
        self.key = key
        self.data = data
        self.next = next
        self.prev = prev
```

List Elements

- Class to describe list elements



Java:

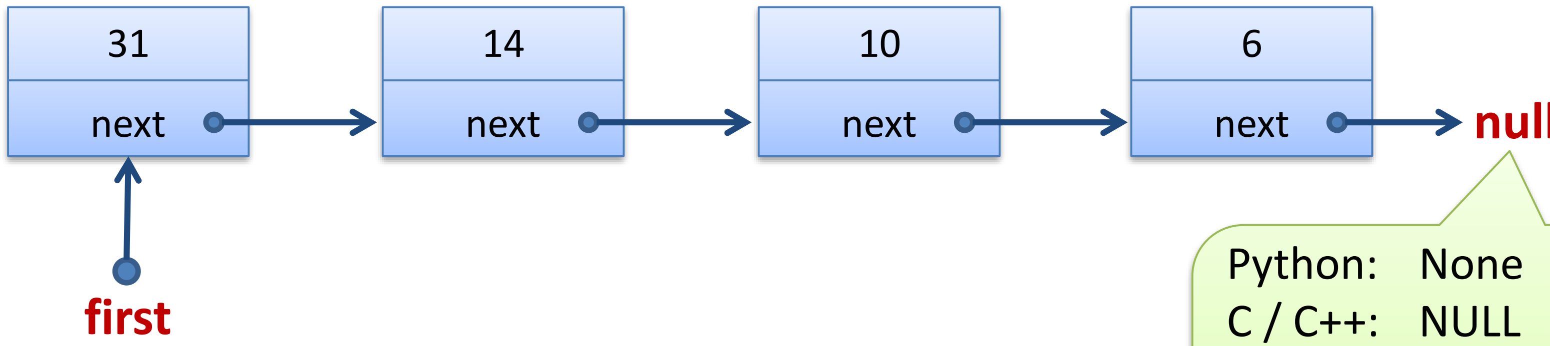
```
public class ListElement {

   int/String/… key;
   Object/… data;

   ListElement next;
   ListElement prev;

}
```

C++:

```
class ListElement {
public/private:

   int/… key;
   void*/… data;

   ListElement* next;
   ListElement* prev;

}
```

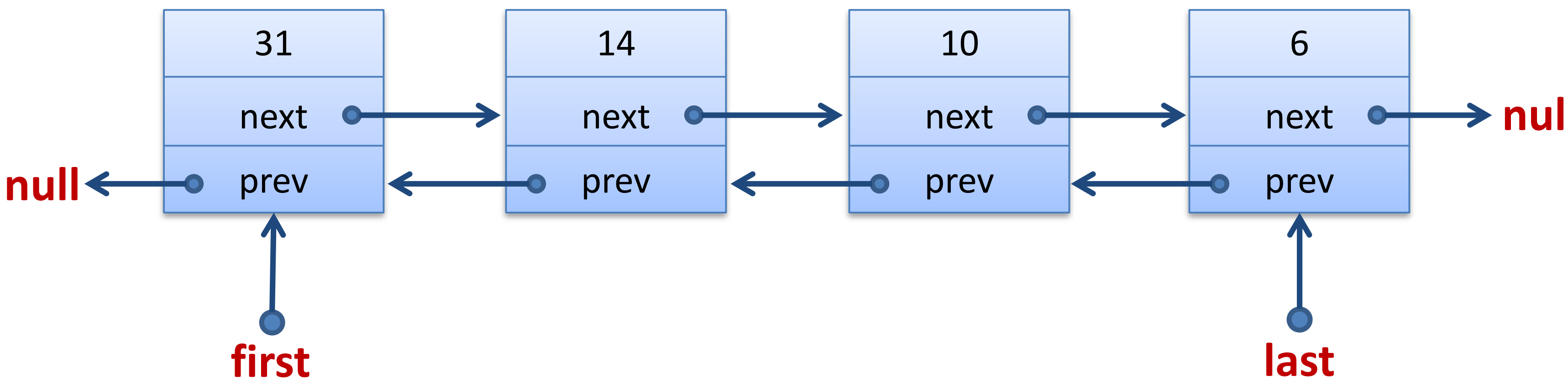
Linked Lists: Structure

Singly Linked List:



Python:	None
C / C++:	NULL
Java:	null
others:	nil
symbol:	⊥

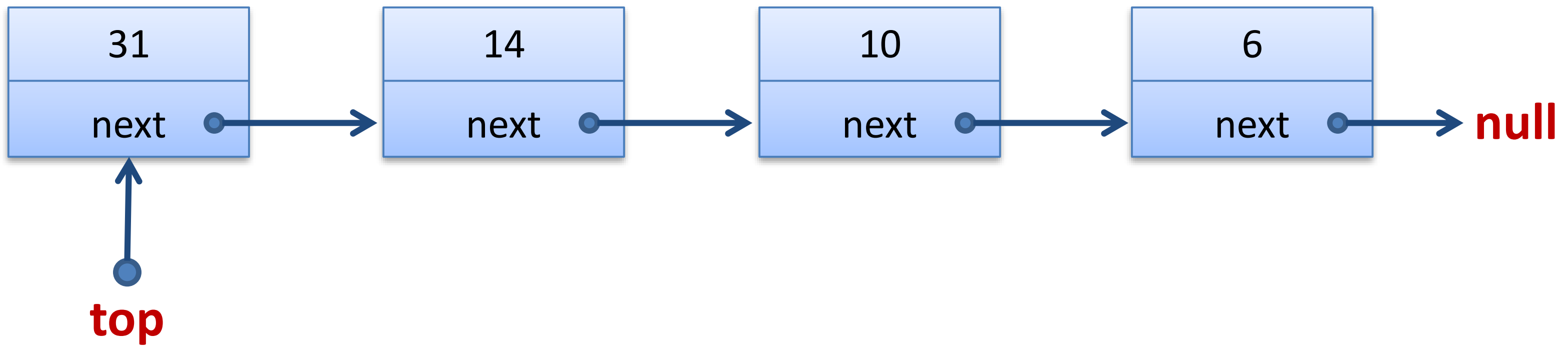
Doubly Linked List:



Stack and FIFO Queue with Lists

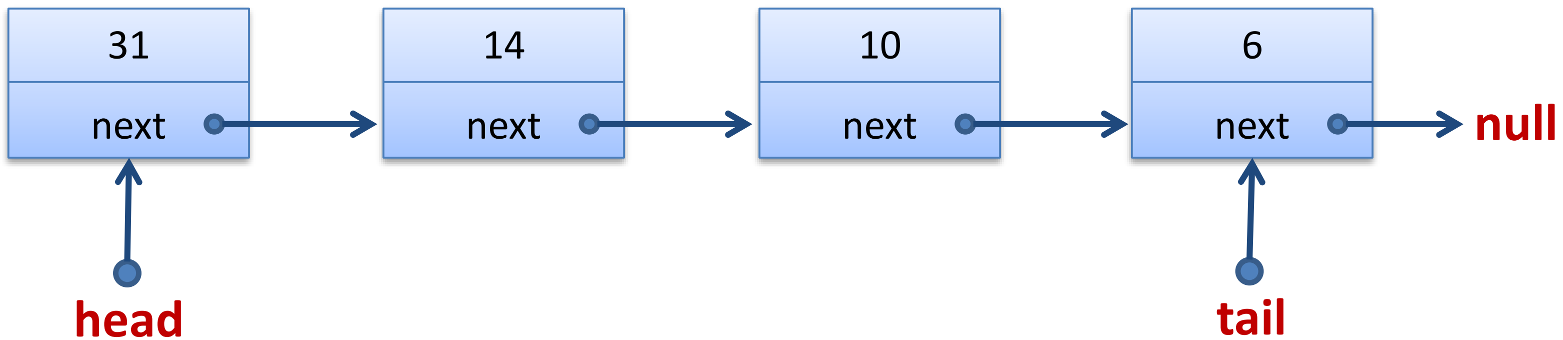
With singly linked lists, all operations in time $O(1)$

Stack:



- Elements can be added (push) und deleted (pop) at front

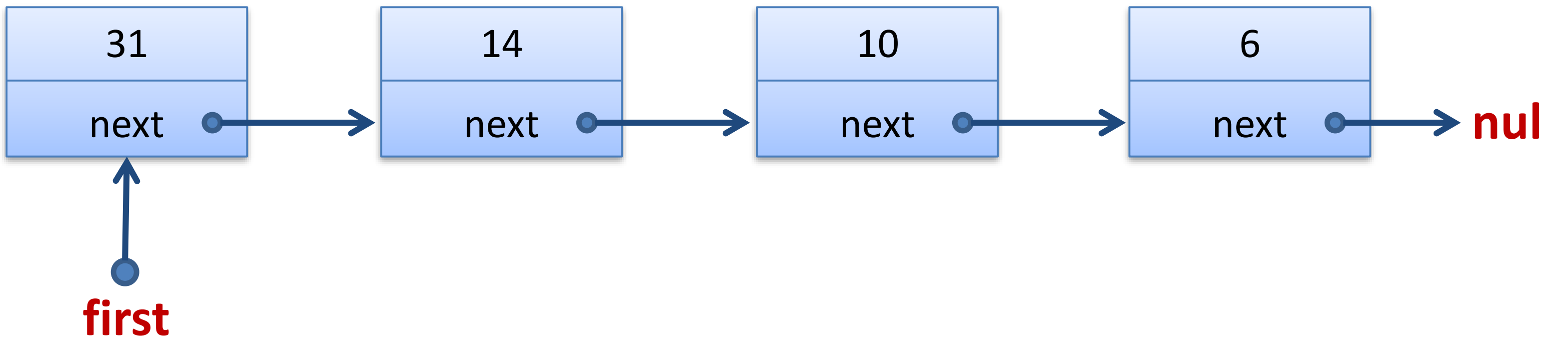
Queue:



- enqueue: add element at end (tail) of list
- dequeue: delete element at front (head) of list

Search in Linked Lists

Singly Linked List:



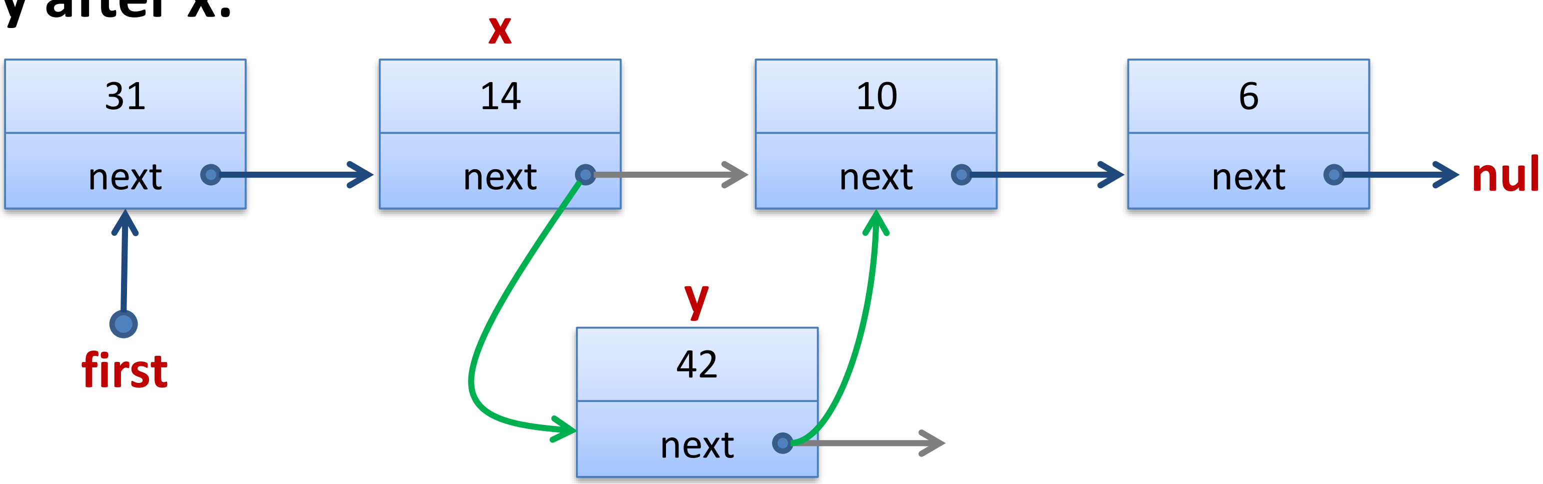
Goal: Find element with key x

```
current = first
while current != None and current.key != x:
    current = current.next
```

Runtime: List of length n : $O(n)$

Insertion in Singly Linked Lists

Insert y after x:

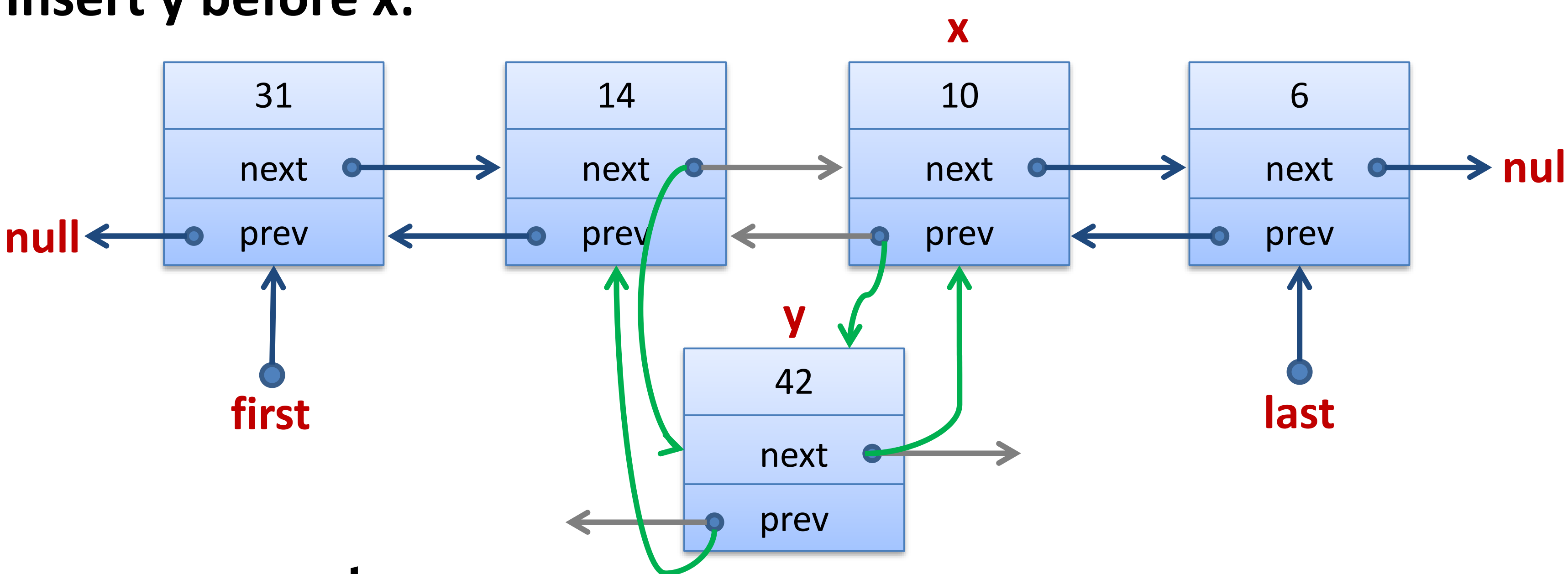



```
y.next = x.next
x.next = y
```

Attention: Take care of special cases at beginning and end of list!

Insertion in Doubly Linked Lists

Insert y before x:

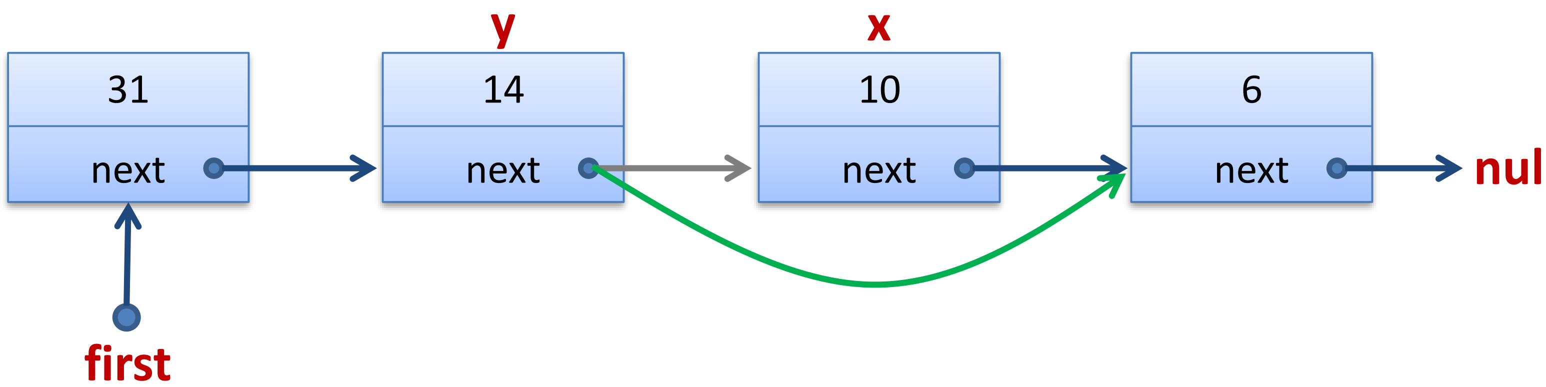



```
y.next = x
y.prev = x.prev
x.prev.next = y
x.prev = y
```

Attention: Take care of special cases at beginning and end of list!

Deletion in Singly Linked Lists

Delete element x:



Assumption: Predecessor element y is given

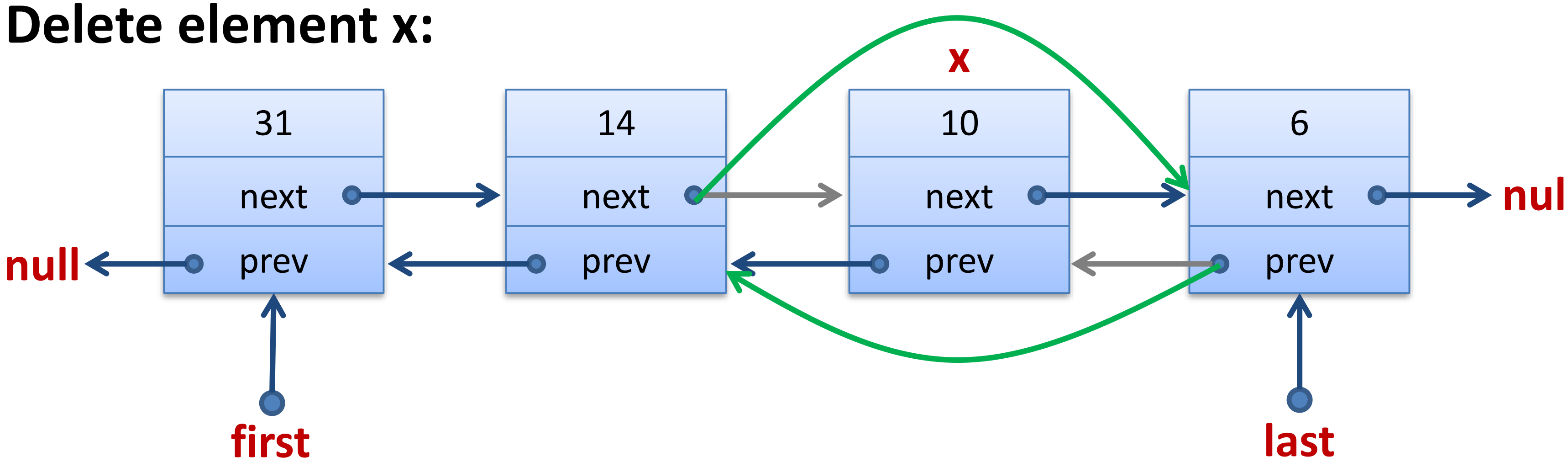
```
y.next = x.next
```

- In C++ one would also need to free the memory used by element x, in Python / Java, this is done by the garbage collector

Attention: Take care of special cases at beginning and end of list!

Deletion in Doubly Linked Lists

Delete element x:




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

Attention: Take care of special cases at beginning and end of list!

Runtime List Operations

Assumption: List of length n

Search for element with key x: $O(n)$

Insertion of an element: $O(1)$

- if reference to predecessor is given, otherwise $O(n)$

Deletion of an element: $O(1)$

- if ref. to predecessor (singly linked lists) or to element itself (doubly linked lists) is given, otherwise $O(n)$

Concatenation of two lists: $O(1)$

- if last pointer to first list is given

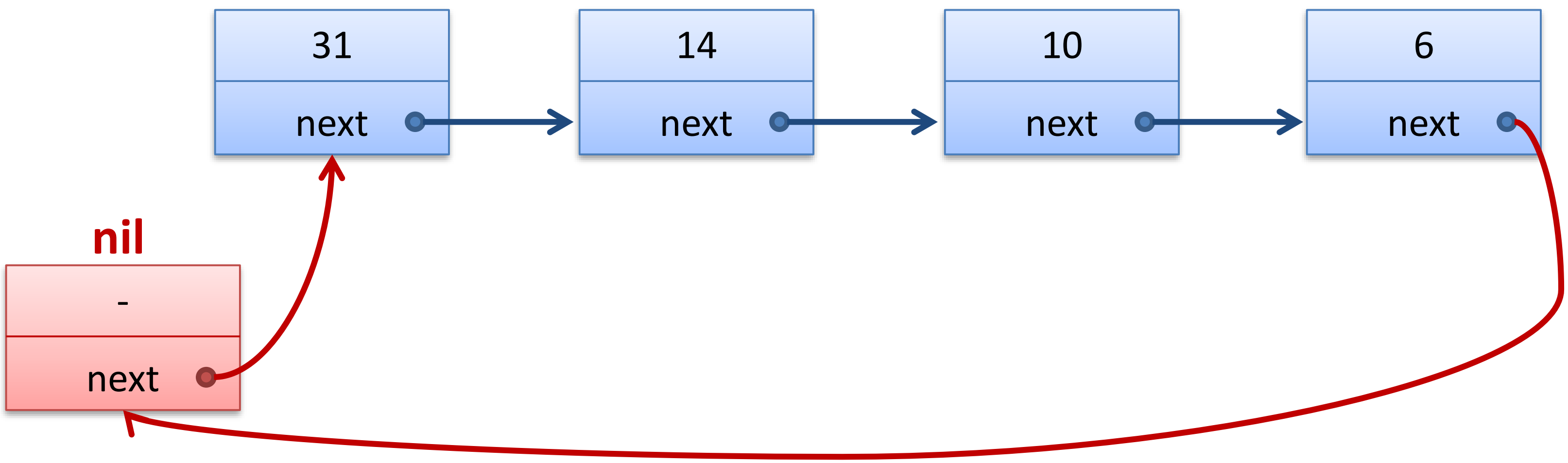
Stack and Queue with linked lists:

- all operations in time $O(1)$
- Size not restricted, memory usage $O(n)$

Lists with a Sentinel

Sentinel:

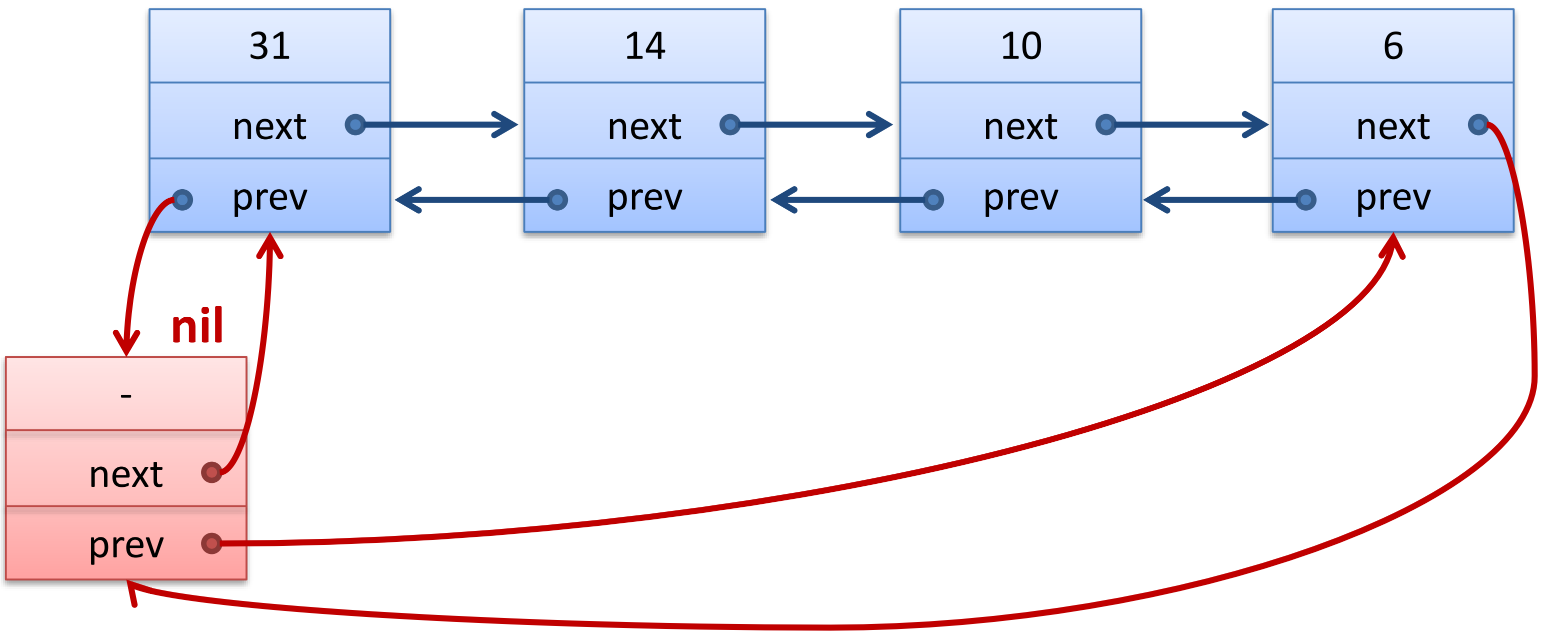
- A dummy element that form the start and end of the list



- list is accessed through *nil.next* instead of *first*
- replaces null pointer at the end of list
- empty list: sentinel points to itself (*nil.next = nil*)
- sentinel is just a part of the implementation and should **not** be visible from outside

Lists with a Sentinel

Sentinel for doubly linked lists:



- list is accessed through *nil.next, nil.prev* instead of *first, last*
- replaces null pointers at start and end of list
- results in a cyclic doubly linked list
- empty list: *nil.next = nil* , *nil.prev = nil*

Sentinel : Remarks

Advantages:

- Avoids special cases at start / end of list when inserting / deleting
- Code becomes simpler and possibly also faster
- Null pointer exceptions are avoided ...
  - Not clear to what extent this improves robustness ...

Disadvantages:

- In case of many small lists, the additional memory useage for the sentinels might become relevant (never asymptotically)
- Sentinels make most sense if they really simplify the code

Abstract Data Types: Dictionary

Dictionary: (also: maps, associative arrays, symbol tables)

- holds a collection of elements where each element is represented by a unique key

Operations:

- *create* : creates an empty dictionary
- *D.insert(key, value)* : inserts a new *(key,value)*-pair
  - If there already is an entry with the same *key*, the old entry is replaced
- *D.find(key)* : returns entry with key *key*
  - If there is such an entry (returns some default value otherwise)
- *D.delete(key)* : deletes entry with key *key*

Dictionary

- In a first phase, we deal with implementing the basic operations *insert, find, delete* (und *create*)

Dictionary Examples:

- Dictionary (key: wort, value: definition / translation)
- Phone Book (key: name, value: phone number)
- DNS Server (key: URL, value: IP address)
- Python interpreter (key: variable name, value: value of variable)
  Java/C++ compiler (key: variable name, value: type information)

In all these cases, it is particularly important to have a fast *find* operation!

Dictionary with Linked Lists

Operations:

- *create*:
  - creates an empty list
- *D.insert(key, value)*:
  - inserts element at front
  - Assumption: There is no previous entry with key *key*
- *D.find(key)*:
  - traverse list sequentially
- *D.delete(key)*:
  - first search the list element with key *key* (as in *find*)
  - delete element from the list
  - For singly linked list, one has to stop as soon as *current.next.key == key* !

Dictionary with Linked Lists

Runtimes:

create: $O(1)$

insert: $O(1)$

- If one does not need to check if key is already present

find: $O(n)$

- We potentially have to iterate over the whole list

delete: $O(n)$

- We potentially have to iterate over the whole list

Is this good?

- In particular *find* is very expensive!

Dictionary with an Array

Operations:

- *create*:
  - allocates a new array of size *NMAX*
- *D.insert(key, value)*:
  - inserts new element at end (if there still is space)
  - Assumption: There is no previous entry with key *key*
- *D.find(key)*:
  - Iterate over all the elements starting at front (or end)
- *D.delete(key)*:
  - first, search for *key*
  - delete element from array, then:

  Move all elements after the deleted element one position to the left!

Dictionary with an Array

Runtimes:

create: $O(1)$

insert: $O(1)$

find: $O(n)$

- We potentially need to iterate over the whole array

delete: $O(n)$

- We potentially have to iterate over the whole array and might need to copy $\Omega(n)$ elements

Better ideas?

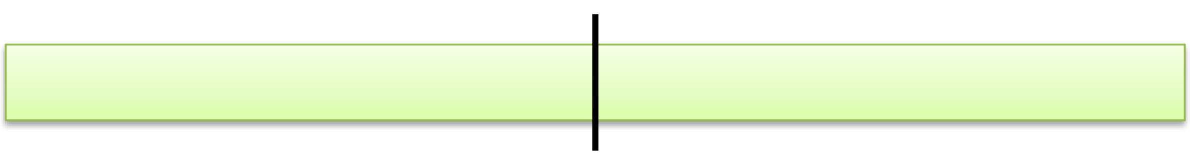
- In particular *find* is still very expensive!

Use a Sorted Array?

- Expensive operation for list / array, in particular ***find***
- If (as soon as) the entries do not change too much, *find* becomes the most important operation!
- Can we search for a given key faster if the entries in an array are sorted by their keys?
  - Example: Search phone number of a person in a phone book…

Ideas for searching x:

- We open the phone book approximately in the middle and check if the name we look for in before or after that position.



y

Is $y < x$ or is $y > x$ or is $y = x$?

Binary Search

Use the divide-and-conquer idea!

Search for the number (the key) 19:

2	**3**	**4**	**6**	**9**	**12**	**15**	**16**	**17**	**18**	**19**	**20**	**24**	**27**	**29**

Binary Search

2	**3**	**4**	**6**	**9**	**12**	**15**	**16**	**17**	**18**	**19**	**20**	**24**	**27**	**29**

Algorithm (array A of length n, search for key x):

- Manage left and right boundary l und r, s. t. (if x is contained in A)

$$A[l] \leq x \leq A[r]$$

- At the beginning, we set $l = 0$ and $r = n - 1$
- Go to the midele $m = (l + r)/2$
  - If $A[m] = x \implies x$ found!
  - If $A[m] < x \implies x$ is in right part $\implies l = m + 1$
  - If $A[m] > x \implies x$ is in left part $\implies l = m - 1$

Binary Search

2	3	4	6	9	12	15	16	17	18	19	20	24	27	29

Algorithm (array A of length n, search for key x):

```
l = 0; r = n - 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then
        l = m + 1
    else if A[m] > x then
        r = m - 1
    else
        l = m; r = m
```

If key x is in array, we have $A[l] = x$ at end

Is the algorithm correct?

How can we verify this?

- Empirically: unit tests or more systematic tests…
- Formally?
  - Correctness is (usually) even more important than performance!

Hoare Logic

- We only look at the basic ideas
- **Precondition**
  - Condition that holds at the beginning (of a method / loop / …)
- **Postcondition**
  - Condition that holds at the end (of a method / loop / …)
- **Loop invariant**
  - Condition that holds at beginning and/or end of each loop iteration

Is the algorithm correct?

```
l = 0; r = n – 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then l = m + 1
    else if A[m] > x then r = m – 1
    else l = m; r = m
```

Precondition

- *array is sorted, array is of length* n

Postcondition

- *If* x *is contained in array, then* $A[l] = x$

Loop invariant

- *If* x *is contained in array, then* $A[l] \leq x \leq A[r]$

Is the algorithm correct?

Precondition

- *array is sorted, array is of length n*

```
l = 0; r = n - 1;
```

loop invariant

Loop invariant

- *If x is contained in array, then $A[l] \leq x \leq A[r]$*
- Precondition and assignment for l and r → loop invariant
  - Loop invariant holds at beginning of first loop iteration

Postcondition

- *If x is contained in array, hen $A[l] = x$*
- Termination condition of while loop $\boxed{\rightarrow l \geq r}$ and thus $\boxed{A[l] \geq A[r]}$
- If x is contained in array, then the loop invariant and the fact that A is sorted imply that $\boxed{A[l] = A[r] \text{ and thus } A[l] = x}$

Is the algorithm correct?

```
l = 0; r = n - 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then l = m + 1
    else if A[m] > x then r = m - 1
    else l = m; r = m
```

Schleifeninvariante

- *If x is contained in array, then $A[l] \leq x \leq A[r]$*
  - The loop invariant holds at the beginning of the loop, it can only be invalidated if we change the variables l and r
  - If we set $l = m + 1$, we know that $A[m] < x$; therefore, we afterwards have $A[m+1] \leq x$ if x is contained in A.
  - Analogously, if we set $r = m - 1$, we know that $A[m] > x$; therefore, we afterwards have $x \leq \ A[m-1]$ if x is contained in A.

Does the algorithm terminate?

```
l = 0; r = n - 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then l = m + 1
    else if A[m] > x then r = m - 1
    else l = m; r = m
```

- Change of number of elements $(r - l + 1)$ per iteration?
  - $l = m + 1$:
  $$r - (m + 1) + 1 \leq r - \left(\frac{l + r}{2} + \frac{1}{2}\right) + 1 = \frac{r - l + 1}{2}$$
  - $r = m - 1$:
  $$(m - 1) - l + 1 \leq \frac{l + r}{2} - 1 - l + 1 = \frac{r - l}{2} < \frac{r - l + 1}{2}$$
  - Otherwise x is found and $r - l + 1$ becomes 1

Runtime

Does the algorithm terminate?

- The number of active elements is at least halved in each iteration
- The algorithm terminates!

Runtime?

$$T(n) \leq T\left(\left\lfloor {}^{n}/_{2} \right\rfloor\right) + c, \qquad T(1) \leq c$$

$$
\begin{aligned}
T(n) &\leq T(n/2) + c \\
&\leq T(n/4) + \underbrace{c + c}_{2c} \\
&\leq T(n/8) + 3c \\
&\vdots \\
&\leq T(n/2^i) + i \cdot c \\
&\vdots \\
&\leq T(1) + c \cdot \log_2 n \leq \underline{c(\log_2 n + 1)}
\end{aligned}
$$

<u>guess</u>

Runtime Binary Search

The algorithm terminates in time $O(\log n)$.

$T(n) \leq T(n/2) + c, \quad T(1) \leq c$

guess: $T(n) \leq c(\log_2 n + 1)$

base: $n = 1 \quad T(1) \leq c(0 + 1) = c$ ✓

step: $n > 1 \quad T(n) \leq T(n/2) + c$

$$\leq c(\underbrace{\log_2 \tfrac{n}{2} + 1}_{\log_2 n}) + c$$

$$= c(\log_2 n + 1).$$ ✓

Dictionary with Sorted Array

Operations:

- *create*:
  - allocates new array of size *NMAX*
- *D.find(key)*:
  - **search for *key* by using binary search**
- *D.insert(key, value)*:
  - Search for *key* and insert element at the right position
  - Insertion: All elements after the insertion have to move one to the right!
- *D.delete(key)*:
  - First search for *key* and remove the respective element
  - Deletion: All elements after the deletion have to move one to the left!

Dictionary with Sorted Array

Runtimes:

create: $O(1)$

insert: $O(n)$

find: $O(\log n)$

delete: $O(n)$

Can we make all operations fast?

- and *find* even faster?