Algorithms and Data Structures
Linked Lists, Binary Search Trees

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science

Algorithms and Data Structures, January 2019

Structure

Sorted Sequences

Linked Lists

Binary Search Trees

Sorted Sequences

Introduction

Structure:

- ▶ We have a set of keys mapped to values
- ▶ We have an ordering ¡ applied to the keys
- ▶ We need the following operations:
  - ▶ `insert(key, value)`: insert the given pair
  - ▶ `remove(key)`: remove the pair with the given key
  - ▶ `lookup(key)`: find the element with the given key, if it is not available find the element with the next smallest key
  - ▶ `next()/previous()`: returns the element with the next bigger/smaller key. This enables iteration over all elements

Application examples:

- ▶ Example: database for books, products or apartments
- ▶ Large number of records (data sets / tuples)
- ▶ Typical query: return all apartments with a monthly rent between 400€ and 600€
    - ▶ This is called a range query
    - ▶ We can implement this with a combination of `lookup(key)` and `next()`
    - ▶ It's not essential that an apartment exists with **exactly** 400€ monthly rent
- ▶ We do not want to sort all elements every time on an insert operation
- ▶ How could we implement this?

Sorted Sequences

Static array:

3	5	9	14	18	21	26	40	41	42	43	46

- `lookup` in time $O(\log n)$
  - With **binary search**
  - Example: `lookup(41)`
- `next` / `previous` in time $O(1)$
  - They are next to each other
- `insert` and `remove` up to $\Theta(n)$
  - We have to copy up to n elements

Sorted Sequences

Implementation 2 (bad) - Hash Table

Hash map:

▶ `insert` and `remove` in $O(1)$

If the hash table is big enough and we use a good hash function

▶ `lookup` in time $O(1)$

If element with **exactly** this key exists, otherwise we get None as result

▶ `next` / `previous` in time up to $\Theta(n)$

Order of the elements is independent of the order of the keys

Sorted Sequences

Implementation 3 (good?) - Linked List

Linked list:

- ▶ Runtimes for doubly linked lists:
  - ▶ `next` / `previous` in time $O(1)$
  - ▶ `insert` and `remove` in $O(1)$
  - ▶ `lookup` in time $\Theta(n)$
- ▶ Not yet what we want, but structure is related to binary search trees
- ▶ Let's have a closer look

Linked Lists

Linked list:

- ▶ Dynamic datastructure
- ▶ Number of elements changeable
- ▶ Data elements can be simple types or composed data structures
- ▶ Elements are linked through references / pointer to the predecessor / successor
- ▶ Single / doubly linked lists possible

Pointer to next element

first

Data

...

None
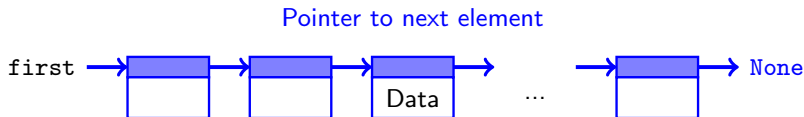
Figure: Linked list

Properties in comparison to an array:

- ▶ Minimal extra space for storing pointer
- ▶ We do not need to copy elements on `insert` or `remove`
- ▶ The number of elements can be simply modified
- ▶ No direct access of elements
  $\Rightarrow$ We have to iterate over the list

Linked Lists
Variants

List with head / last element pointer:
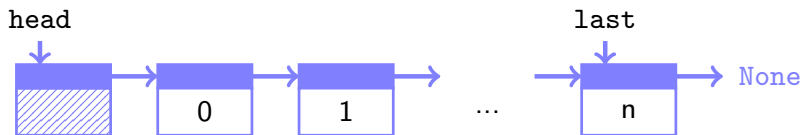
Figure: Singly linked list

- ▶ Head element has pointer to first list element
- ▶ May also hold additional information:
  - ▶ Number of elements

Linked Lists
Variants

Doubly linked list:
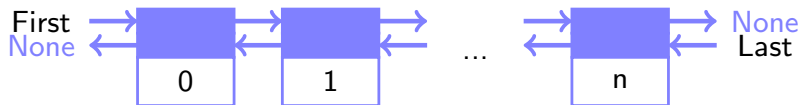
Figure: Doubly linked list

- ▶ Pointer to successor element
- ▶ Pointer to predecessor element
- ▶ Iterate forward and backward

Linked Lists

Implementation - Node/Element - Python

```python
class Node:
    """ Defines a node of a singly linked
        list.
    """

    def __init__(self, value, nextNode=None):
        self.value = value
        self.nextNode = nextNode
```

Linked Lists

Creating linked lists - Python:

- `first = Node(7)`

  first →　7　→ None

- `first.nextNode = Node(3)`

  first →　7　→　3　→ None

- `first.nextNode.value = 4`

  first →　7　→　4　→ None

Linked Lists
Implementation - Insert

Inserting a node after node `cur`:

Linked Lists
Implementation - Insert

Inserting a node after node `cur`:

▶ `ins = Node(n)`
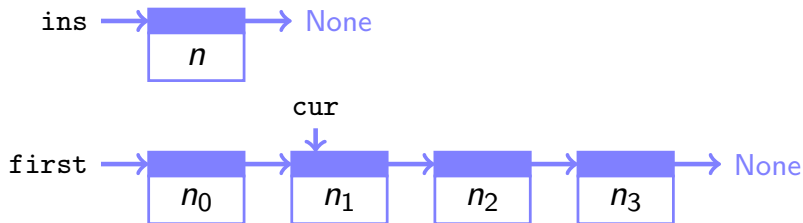
Linked Lists

Implementation - Insert

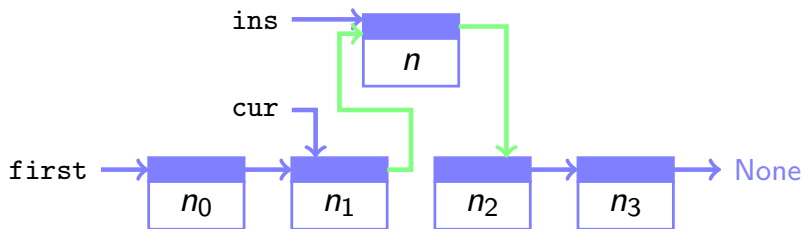**Inserting a node after node** `cur`**:**

▶ `ins.nextNode = cur.nextNode`
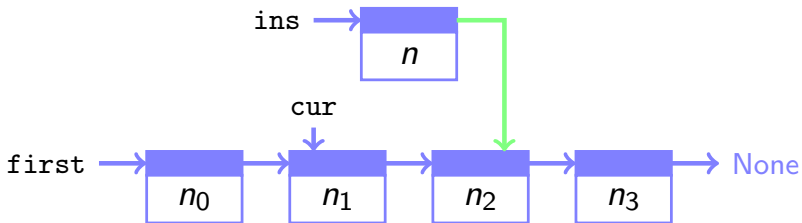
Linked Lists
Implementation - Insert

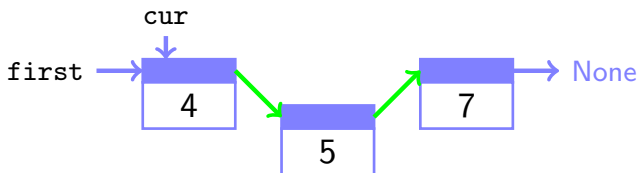**Inserting a node after node** `cur`**:**

▶ `cur.nextNode = ins`

Linked Lists
Implementation - Insert

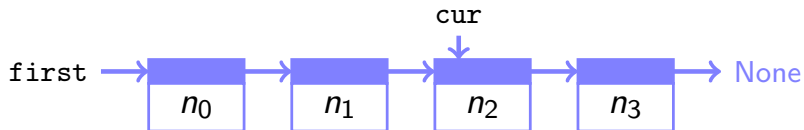**Inserting a node after node** `cur` - **single line of code:**

▶ `cur.nextNode = Node(value, cur.nextNode)`

Linked Lists
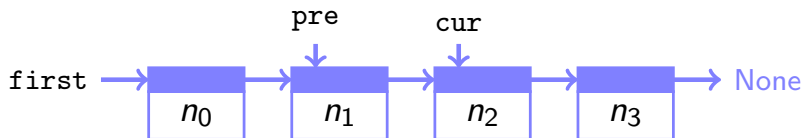Implementation - Remove

Removing a node `cur`:

Linked Lists

Removing a node `cur`**:**

- ▶ Find the predecessor of `cur`:

```
pre = first
while pre.nextNode != cur:
    pre = pre.nextNode
```

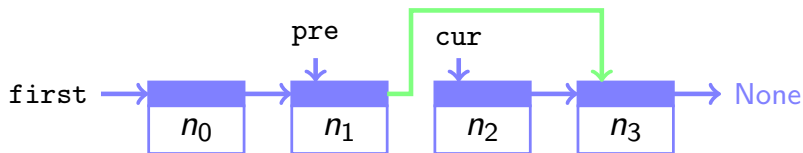- ▶ Runtime of $O(n)$
- ▶ Does not work for first node!

Linked Lists
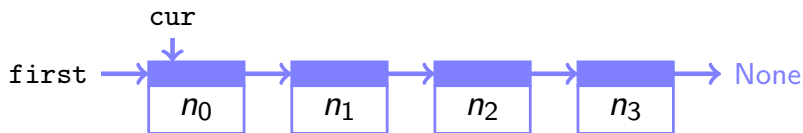
Implementation - Remove

Removing a node `cur`**:**

▶ Update the pointer to the next element:
   `pre.nextNode = cur.nextNode`

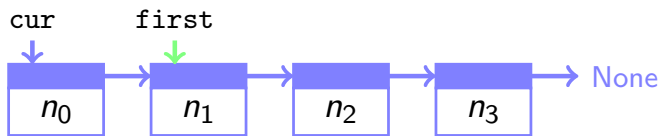▶ cur will get destroyed automatically if no more references
   exist (cur=None)

Removing the first node:

- ▶ Update the pointer to the next element:
  first = first.nextNode
- ▶ cur will get automaticly destroyed if no more references exist
  (cur=None)

Linked Lists
Implementation - Remove

Removing a node cur: (General case)

```
if cur == first:
    first = first.nextNode
else:
    pre = first
    while pre.nextNode != cur:
        pre = pre.nextNode

    pre.nextNode = cur.nextNode
```
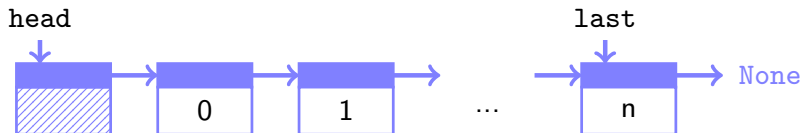
Linked Lists
Implementation - Head Node

Using a `head` node:

- ▶ Advantage:
  - ▶ Deleting the first node is no special case
- ▶ Disadvantage
  - ▶ We have to consider the first node at other operations
  - ▶ Iterating all nodes
  - ▶ Counting of all nodes
  - ▶ . . .

Linked Lists

Implementation - LinkedList - Python

```python
class LinkedList:
    def __init__(self):
        self.itemCount = 0
        self.head = Node()
        self.last = self.head

    def size(self):
        return self.itemCount

    def isEmpty(self):
        return self.itemCount == 0
```

Linked Lists

Implementation - LinkedList - Python

```python
def append(self, value):
    ...

def insertAfter(self, cur, value):
    ...

def remove(self, cur):
    ...

def get(self, position):
    ...

def contains(self, value):
    ...
```
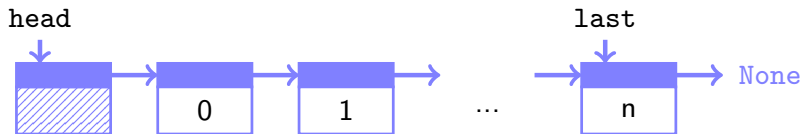
Linked Lists
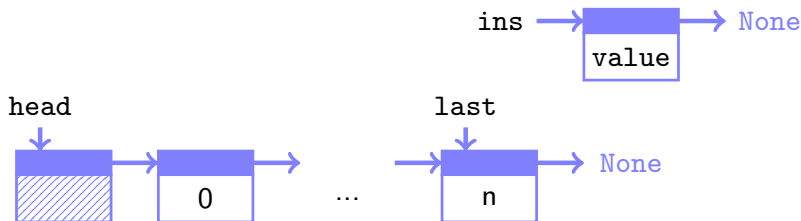Implementation

Head, last:

- ▶ Head points to the first node, `last` to the last node
- ▶ We can append elements to the end of the list in $O(1)$ through the `last` node
- ▶ We have to keep the pointer to `last` updated after all operations

Linked Lists
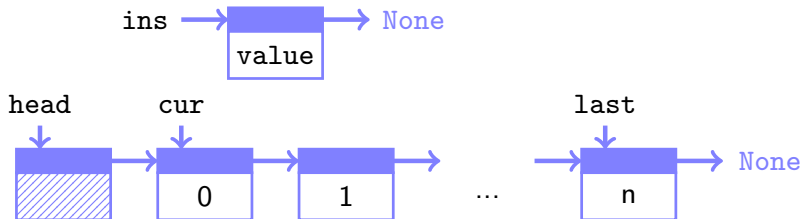Implementation - Append

Appending an element:


```
def append(self, value):
    last.nextNode = Node(value)
    last = last.NextNode
    itemCount += 1
```

▶ The pointer to last avoids the iteration of the whole list

Linked Lists

Implementation - Insert After

Inserting after node `cur`**:**

Linked Lists
Implementation - Insert After

Inserting after node `cur`:

▶ The pointer to head is not modified

```
def insertAfter(self, cur, value):
    if cur == last:
        # also update last node
        append(value)
    else:
        # last node is not modified
        cur.nextNode = Node(value, \
            cur.nextNode)
        itemCount += 1
```

Linked Lists
Implementation - Remove

Remove node `cur`:

Remove node `cur`**:**
- ► Searching the predecessor in $O(n)$

```python
def remove(self, cur):
    pre = first
    while pre.nextNode != cur:
        pre = pre.nextNode

    pre.nextNode = cur.nextNode
    itemCount -= 1

    if pre.nextNode == None:
        last = pre
```

Getting a reference to node at pos:

▶ Iterate the entries of the list until position in $O(n)$

```python
def get(self, pos):
    if pos < 0 or pos >= itemCount:
        return None

    cur = head
    for i in range(0, pos):
        cur = cur.nextNode

    return cur
```

Linked Lists
Implementation - Contains

Searching a `value`**:**
- ▶ First element is head without an assigned value
- ▶ Iterate the entries of the list until value found in $O(n)$

```python
def contains(self, value):
    cur = head

    for i in range(0, itemCount):
        cur = cur.nextNode
        if cur.value == value:
            return True

    return False
```
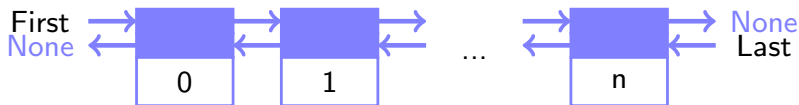
Runtime:

▶ Singly linked list:
  ▶ `next` in $O(1)$
  ▶ `previous` in $\Theta(n)$
  ▶ `insert` in $O(1)$
  ▶ `remove` in $\Theta(n)$
  ▶ `lookup` in $\Theta(n)$

▶ Better with doubly linked lists

Linked Lists
Doubly Linked List

Doubly linked list:

► Each node has a reference to its successor and its predecessor
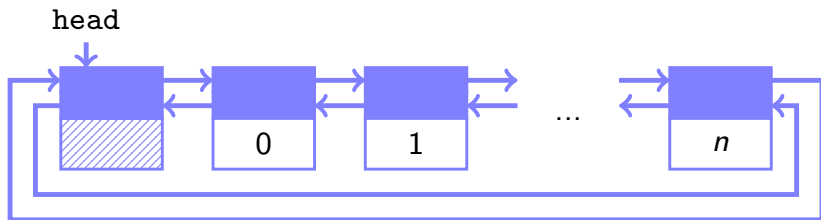
► We can iterate the list forward and backward

Doubly linked list:
- ▶ It is helpful to have a head node
- ▶ We only need one head node if we cyclically connect the list

Runtime of doubly linked list:

- ▶ `next` and `previous` in $O(1)$

  Each element has a pointer to pred-/sucessor

- ▶ `insert` and `remove` in $O(1)$

  A constant number of pointers needs to be modified

- ▶ `lookup` in $\Theta(n)$

  Even if the elements are sorted we can only retrieve them in $\Theta(n)$ Why?

Linked Lists
List in real program

Linked list in book:

Linked Lists

Linked list in memory:

Binary Search Trees
Introduction

Runtime of a search tree:

- next and previous in $O(1)$

  Pointers corresponding to linked list

- insert and remove in $O(\log n)$

- lookup in $O(\log n)$

  The structure helps searching efficiently

Binary Search Trees
Introduction

Idea:

- ▶ We define a total order for the search tree
- ▶ All nodes of the left subtree have smaller keys than the current node
- ▶ All nodes of the right subtree have bigger keys than the current node

Binary Search Trees
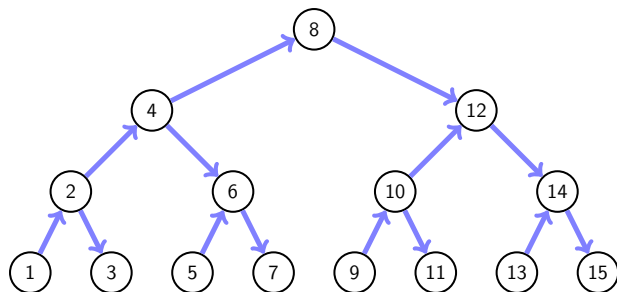Introduction

- ▶ Edge direction indicates ordering

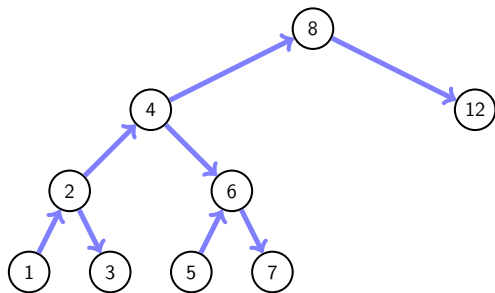Figure: a binary search tree

Binary Search Trees
Introduction

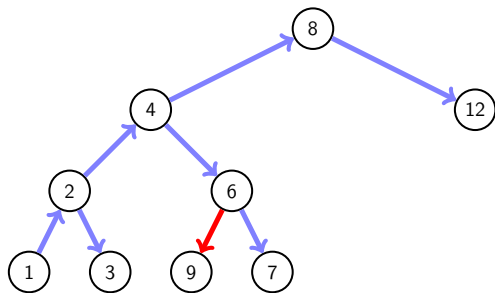Figure: another binary search tree

Binary Search Trees

Introduction

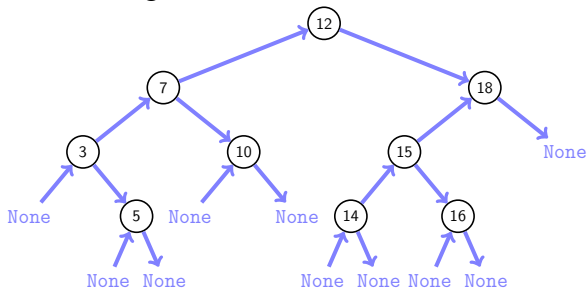Figure: **not** a binary search tree

Binary Search Trees

Implementation:

- ▶ For the heap we had all elements stored in an array
- ▶ Here we link all nodes through pointers / references, like linked lists
- ▶ Each node has a pointer / reference to its children (`leftChild` / `rightChild`)
- ▶ `None` for missing children

Binary Search Trees

Implementation:

- ▶ We create a sorted doubly linked list of all elements
- ▶ This enables an efficient implementation of (`next` / `previous`)

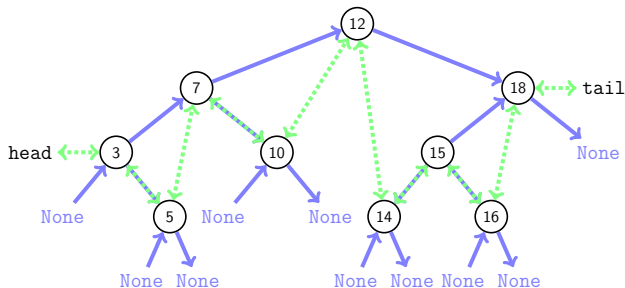Figure: binary search tree with links

Binary Search Trees

Lookup:

- ▶ Definition:
  " Search the element with the given key. If no element is found return the element with the next (bigger) key. "
- ▶ We search from the root downwards:
  - ▶ Compare the searched key with the key of the node
  - ▶ Go to the left / right until the child is `None` or the key is found
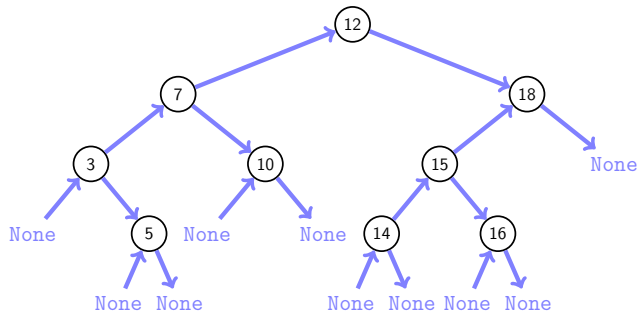 - ▶ If the key is not found return the next bigger one

Binary Search Trees

Implementation - Lookup

For each node applies the total order:
    keys of left subtree ¡ `node.key` ¡ keys of right subtree

Figure: binary search tree with total order "¡"

Examples:

`lookup(14)`

`lookup(6)`

`lookup(19)`

Binary Search Trees

Insert:

▶ We search for the key in our search tree

▶ If a node is found we replace the value with the new one

▶ Else we insert a new node

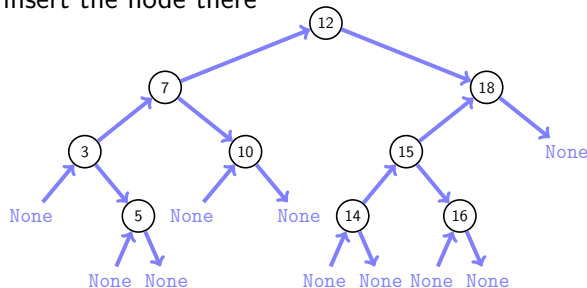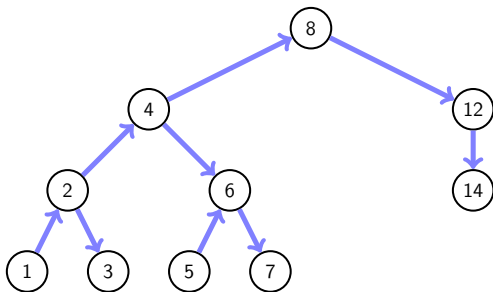▶ If the key was not present we get a None entry

▶ We insert the node there

Figure: Binary search tree with total order "¡"

Binary Search Trees

Implementation - Remove

Remove: case 1: the node "5" has no children

- ▶ Find parent of node "5" ("6")
- ▶ Set left / right child of node "6" to None depending on position of node "5"

Figure: Binary search tree with total order "¡"

Binary Search Trees

Implementation - Remove

Remove: Case 1: The node "5" has no children

- ▶ Find parent of node "5" ("6")
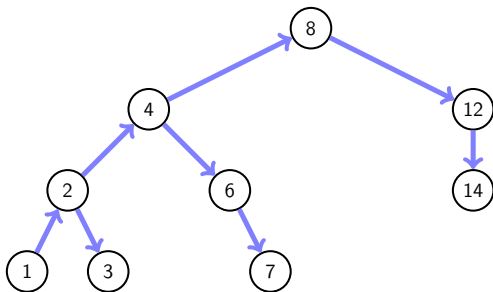- ▶ Set left / right child of node "6" to None depending on position of node "5"

Figure: binary search tree after deleting node "5"

Binary Search Trees

Implementation - Remove

Remove: Case 2: The node "12" has one child

- ▶ Find the child of node "12" ("14")
- ▶ Find the parent of node "12" ("8")
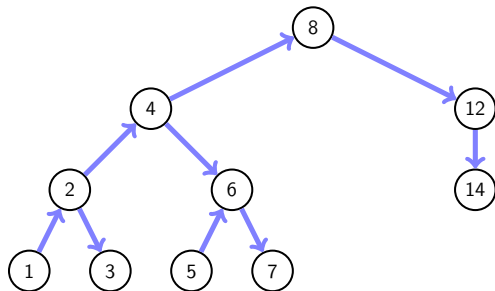- ▶ Set left / right child of node "8" to "14" depending on position of node "12" (skip node "14")

Figure: binary search tree with total order "¡"

Binary Search Trees

Implementation - Remove

Remove: Case 2: The node "12" has one child

- ▶ Find the child of node "12" ("14")
- ▶ Find the parent of node "12" ("8")
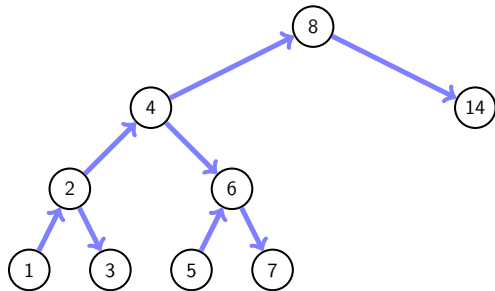- ▶ Set left / right child of node "8" to "14" depending on position of node "12" (skip node "14")

Figure: binary search tree after delting node "12"

Binary Search Trees
Implementation - Remove

Remove: Case 3: The node "4" has two children
- ▶ Find the successor of node "4" ("5")
- ▶ Replace the value of node "4" with the value of node "5"
- ▶ Delete node "5" (the successor of node "4") with remove-case 1 or 2
- ▶ There is no left node because we are deleting the predecessor

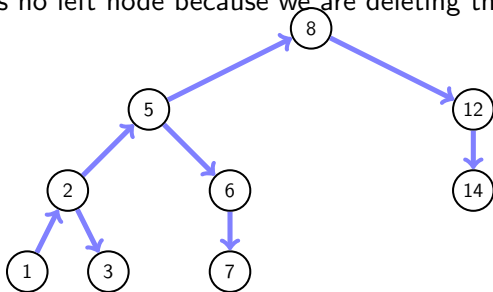# Binary Search Trees
Implementation - Remove

Remove: Case 3: The node "4" has two children

▶ Find the successor of node "4" ("5")

▶ Replace the value of node "4" with the value of node "5"

▶ Delete node "5" (the successor of node "4") with remove-case 1 or 2

▶ There is no left node because we are deleting the predecessor
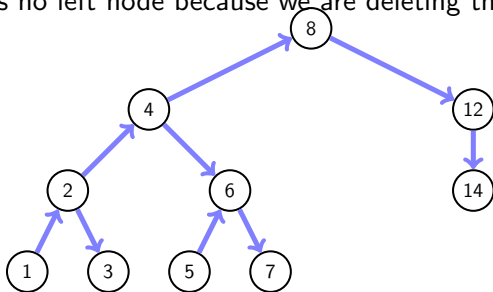
Binary Search Trees

How long takes `insert` and `lookup`?

- ▶ Up to $\Theta(d)$, with d being the depth of the tree
  (The longest path from the root to a leaf)
- ▶ Best case with $d = \log n$ the runtime is $\Theta(\log n)$
- ▶ Worst case with $d = n$ the runtime is $\Theta(n)$
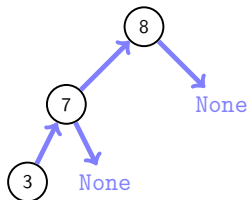- ▶ If we **always** want to have a runtime of $\Theta(\log n)$ then we have to rebalance the tree
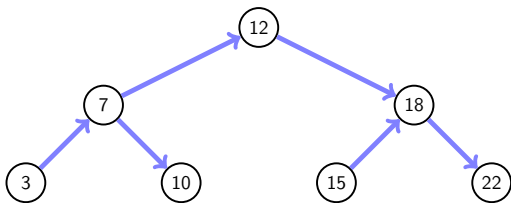

Figure: degenerated binary tree $d = n$

Figure: complete binary tree $d = \log n$

▶ **Course literature**

[CRL01] Thomas H. Cormen, Ronald L. Rivest, and
         Charles E. Leiserson.
         Introduction to Algorithms.
         MIT Press, Cambridge, Mass, 2001.

[MS08]   Kurt Mehlhorn and Peter Sanders.
         Algorithms and data structures, 2008.
         https://people.mpi-inf.mpg.de/~mehlhorn/
         ftp/Mehlhorn-Sanders-Toolbox.pdf.

► **Linked List**

[Wik] Linked list
https://en.wikipedia.org/wiki/Linked_list

► **Binary Search Tree**

[Wik] Binary search tree
https:
//en.wikipedia.org/wiki/Binary_search_tree