

# Algorithms and Datastructures

## Linked Lists, Binary Search Trees

Albert-Ludwigs-Universität Freiburg



**UNI  
FREIBURG**

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science  
Algorithms and Datastructures, January 2018

Sorted Sequences

Linked Lists

Binary Search Trees

### Structure:

### Structure:

- We have a set of **keys** mapped to **values**

### Structure:

- We have a set of **keys** mapped to **values**
- We have a ordering  $<$  applied to the keys

### Structure:

- We have a set of **keys** mapped to **values**
- We have a ordering  $<$  applied to the keys
- We need the following operations:

### Structure:

- We have a set of **keys** mapped to **values**
- We have a ordering  $<$  applied to the keys
- We need the following operations:
  - `insert(key, value)`: Insert the given pair

### Structure:

- We have a set of **keys** mapped to **values**
- We have a 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**



### Structure:

- We have a set of **keys** mapped to **values**
- We have a 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

### Structure:

- We have a set of **keys** mapped to **values**
- We have a 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:**

### **Application examples:**

- Example: Database for books, products or apartments

### **Application examples:**

- Example: Database for books, products or apartments
- Large number of records (data sets / tuples)

### 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€

### 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**

### 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()**



### 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 if an apartments exists with **exactly** 400€ monthly rent

### 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 if an apartments exists with **exactly** 400€ monthly rent
- We do not want to sort all elements every time on an **insert** operation

### 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 if an apartments 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

## Implementation 1 (not good) - Static Array



### Static array:

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

### Static array:

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

- **lookup** in time  $O(\log n)$

### Static array:

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

- **lookup** in time  $O(\log n)$ 
  - With **binary search**

### 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)`

### 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  $O(1)$



### 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

### 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  $\Theta(n)$

### 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:



### Hash map:

- `insert` and `remove` in  $O(1)$



### Hash map:

- `insert` and `remove` in  $O(1)$

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

### 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  $O(1)$

### 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  $\Theta(n)$



### 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:**

### **Linked list:**

- Runtimes for doubly linked lists:



### Linked list:

- Runtimes for doubly linked lists:
  - `next` / `previous` in time  $O(1)$



### Linked list:

- Runtimes for doubly linked lists:
  - `next` / `previous` in time  $O(1)$
  - `insert` and `remove` in  $O(1)$

### Linked list:

- Runtimes for doubly linked lists:
  - `next` / `previous` in time  $O(1)$
  - `insert` and `remove` in  $O(1)$
  - `lookup` in time  $\Theta(n)$

### 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

### 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



Sorted Sequences

Linked Lists

Binary Search Trees



## Linked list:



### **Linked list:**

- Dynamic datastructure



### **Linked list:**

- Dynamic datastructure
- Number of elements changeable

### Linked list:

- Dynamic datastructure
- Number of elements changeable
- Data elements can be simple types or composed datastructures

### Linked list:

- Dynamic datastructure
- Number of elements changeable
- Data elements can be simple types or composed datastructures
- **Elements are linked** through references / pointer to the predecessor / successor

### Linked list:

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

### Linked list:

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



Figure: Linked list





### **Properties in comparison to an array:**



### **Properties in comparison to an array:**

- Minimal extra space for storing pointer

### Properties in comparison to an array:

- Minimal extra space for storing pointer
- We do not need to copy elements on `insert` or `remove`

### 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

### 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
  - ⇒ We have to iterate over the list



### List with head / last element pointer:

### List with head / last element pointer:



Figure: Singly linked list

### List with head / last element pointer:



Figure: Singly linked list

- Head element has pointer to first list element



### List with head / last element pointer:



Figure: Singly linked list

- Head element has pointer to first list element
- May also hold additional information:

### 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



## Doubly linked list:

### Doubly linked list:



Figure: Doubly linked list

### Doubly linked list:



Figure: Doubly linked list

- Pointer to successor element

### Doubly linked list:



Figure: Doubly linked list

- Pointer to successor element
- Pointer to predecessor element

### Doubly linked list:

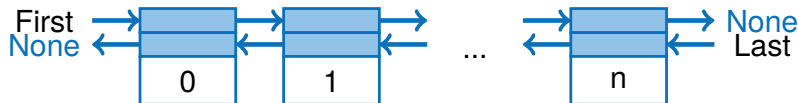


Figure: Doubly linked list

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

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

    def __init__(self, value, nextNode):
        self.value = value
        self.nextNode = nextNode

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





## Creating linked lists - Python:

### Creating linked lists - Python:

```
■ first = Node(7)
```



### Creating linked lists - Python:

■ `first = Node(7)`



■ `first.nextNode = Node(3)`



### Creating linked lists - Python:

■ `first = Node(7)`



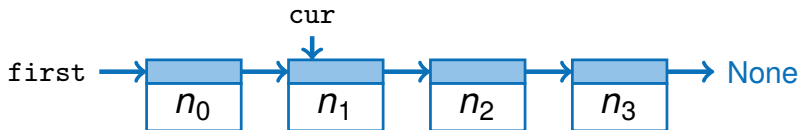
■ `first.nextNode = Node(3)`



■ `first.nextNode.value = 4`



Inserting a node after node `cur`:





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

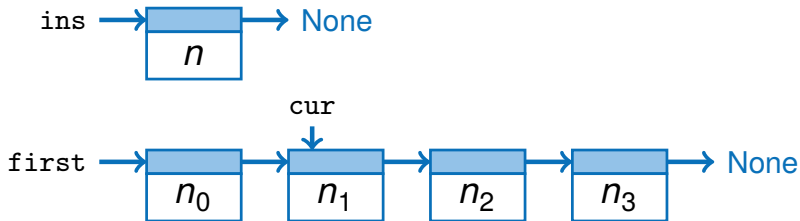


### Inserting a node after node `cur`:

- `ins = Node(n)`

### Inserting a node after node `cur`:

■ `ins = Node(n)`





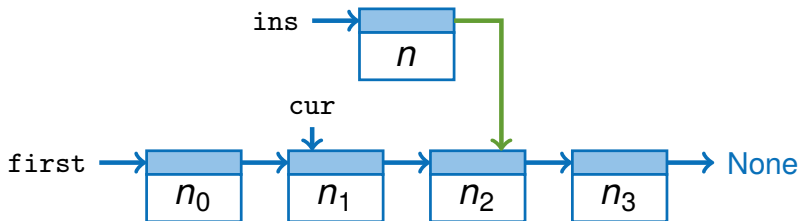


### Inserting a node after node `cur`:

- `ins.nextNode = cur.nextNode`

### Inserting a node after node `cur`:

■ `ins.nextNode = cur.nextNode`



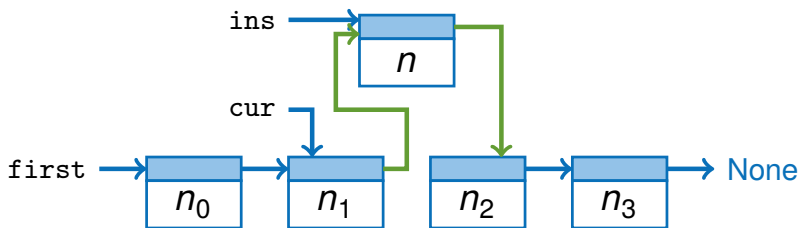


### Inserting a node after node `cur`:

- `cur.nextNode = ins`

### Inserting a node after node `cur`:

■ `cur.nextNode = ins`





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

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



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

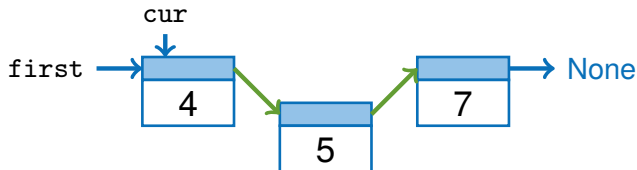


■ `cur.nextNode = Node(value, cur.nextNode)`

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



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





### Removing a node `cur`:





### Removing a node `cur:`



### Removing a node `cur`:

- Find the predecessor of `cur`:

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



### Removing a node `cur`:

- Find the predecessor of `cur`:

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

- Runtime of  $O(n)$



### 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!

### 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!





### Removing a node `cur`:



### Removing a node `cur`:

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

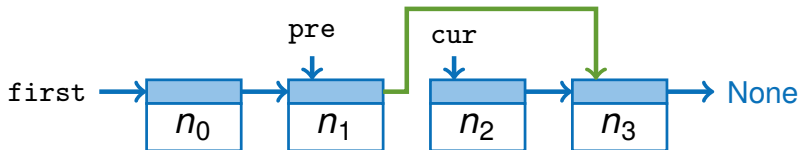


### Removing a node `cur`:

- Update the pointer to the next element:  
`pre.nextNode = cur.nextNode`
- `cur` will get automatically destroyed if no more references exist (`cur=None`)

### Removing a node `cur`:

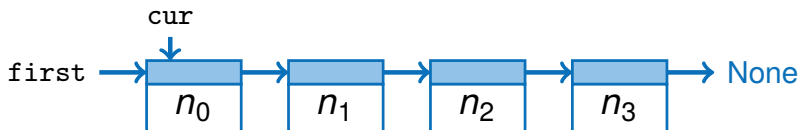
- Update the pointer to the next element:  
`pre.nextNode = cur.nextNode`
- `cur` will get automatically destroyed if no more references exist (`cur=None`)





## Removing the first node:

### Removing the first node:



### Removing the first node:



- Update the pointer to the next element:

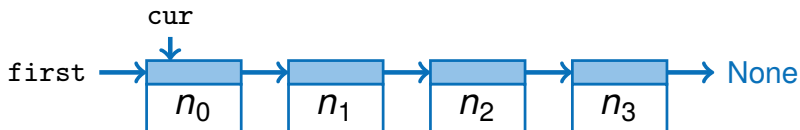
```
first = first.nextNode
```

### Removing the first node:

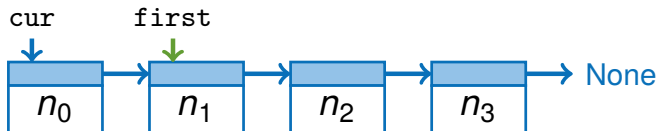


- Update the pointer to the next element:  
`first = first.nextNode`
- `cur` will get automatically destroyed if no more references exist (`cur=None`)

### Removing the first node:



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



### 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
```





### **Using a head node:**



### Using a head node:

- Advantage:



### Using a head node:

- Advantage:
  - Deleting the first node is no special case



### Using a head node:

- Advantage:
  - Deleting the first node is no special case
- Disadvantage
  - We have to consider the first node at other operations

### 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

### 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
  - ...

### 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
  - ...



```
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
```



```
def append(self, value):  
    ...  
  
def insertAfter(self, cur, value):  
    ...  
  
def remove(self, cur):  
    ...  
  
def get(self, position):  
    ...  
  
def contains(self, value):  
    ...
```

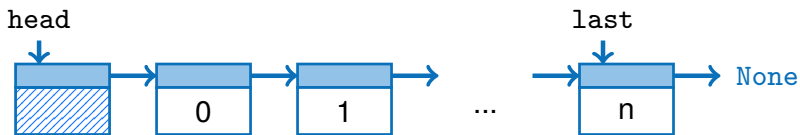


**Head, last:**

### Head, last:



### Head, last:



- Head points to the first node, last to the last node

### 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

### 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



### Appending an element:

### Appending an element:





### Appending an element:



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

### 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

### Inserting after node `cur`:





### Inserting after node `cur`:

- The pointer to `head` is not modified

### 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
```

### Remove node cur:





### **Remove node** `cur`:

- Searching the predecessor in  $O(n)$

### Remove node cur:

- Searching the predecessor in  $O(n)$

```
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 at position in  $O(n)$

### Getting a reference to node at pos:

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

```
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
```



### Searching a value:



### Searching a value:

- First element is head without an assigned value

### Searching a value:

- First element is head without an assigned value
- Iterate the entries of the list until value found in  $O(n)$

### Searching a value:

- First element is head without an assigned value
- Iterate the entries of the list until value found in  $O(n)$

```
def contains(self, value):  
    cur = head  
  
    for i in range(0, itemCount):  
        cur = cur.nextNode  
        if cur.value == value:  
            return True  
  
    return False
```



**Runtime:**



### Runtime:

- Singly linked list:



### Runtime:

- Singly linked list:
  - `next` in  $O(1)$



### Runtime:

- Singly linked list:
  - `next` in  $O(1)$
  - `previous` in  $\Theta(n)$



### Runtime:

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



### Runtime:

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

### Runtime:

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

### 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`



### **Doubly linked list:**

### **Doubly linked list:**

- Each node has a reference to its successor and its predecessor



### 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:

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





### **Doubly linked list:**



### Doubly linked list:

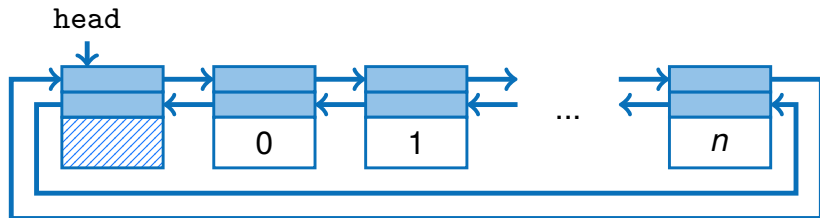
- It is helpful to have a **head** node

### Doubly linked list:

- It is helpful to have a **head** node
- We only need **one head** node if we connect the list cyclic

### Doubly linked list:

- It is helpful to have a **head** node
- We only need **one head** node if we connect the list cyclic





## Runtime of doubly linked list:



### Runtime of doubly linked list:

- `next` and `previous` in  $O(1)$



### Runtime of doubly linked list:

- `next` and `previous` in  $O(1)$

Each element has a pointer to pred-/sucessor

### Runtime of doubly linked list:

- `next` and `previous` in  $O(1)$

Each element has a pointer to pred-/sucessor

- `insert` and `remove` in  $O(1)$

### 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

### 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  $\Theta(n)$

### 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  $\Theta(n)$

### 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 list in book:



# Linked Lists

List in real program



## Linked list in memory:





Sorted Sequences

Linked Lists

Binary Search Trees

### Runtime of a search tree:

### Runtime of a search tree:

- `next` and `previous` in  $O(1)$

### Runtime of a search tree:

- `next` and `previous` in  $O(1)$

Pointers corresponding to linked list

### Runtime of a search tree:

- `next` and `previous` in  $O(1)$

Pointers corresponding to linked list

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

### 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  $O(\log n)$

### 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



**Idea:**



### Idea:

- We define a total order for the search tree

### Idea:

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

### 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

- Edge direction indicates ordering

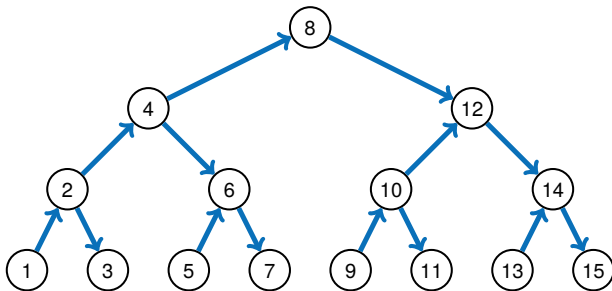


Figure: A binary search tree



Figure: Another binary search tree



Figure: **Not** a binary search tree



## Implementation:

### Implementation:

- For the heap we had all elements stored in an array
- Here we link all nodes through pointer / references, like linked lists



### Implementation:

- For the heap we had all elements stored in an array
- Here we link all nodes through pointer / references, like linked lists
- Each node has a pointer / reference to its children (`leftChild` / `rightChild`)

### Implementation:

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

### Implementation:

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





## Implementation:



### Implementation:

- We create a sorted doubly linked list of all elements

### Implementation:

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

### Implementation:

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

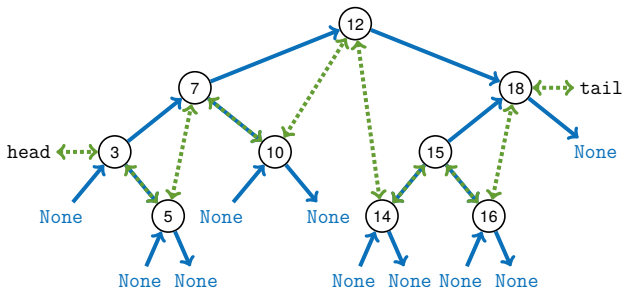


Figure: Binary search tree with links



### Lookup:



### Lookup:

- Definition:  
“ Search the element with the given key. If no element is found return the element with the next (bigger) key. ”

### 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:

### 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

### 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

### 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



**For each node applies the total order:**



**For each node applies the total order:**

keys of left subtree < `node.key` < keys of right subtree

**For each node applies the total order:**

keys of left subtree < `node.key` < keys of right subtree



**Examples:**

**Figure:** Binary search tree with total order “<”



**For each node applies the total order:**

keys of left subtree < `node.key` < keys of right subtree



**Examples:**

lookup(14)

Figure: Binary search tree with total order “<”

**For each node applies the total order:**

keys of left subtree < `node.key` < keys of right subtree



**Examples:**

lookup(14)

lookup(6)

Figure: Binary search tree with total order “<”

**For each node applies the total order:**

keys of left subtree < `node.key` < keys of right subtree



**Examples:**

lookup(14)

lookup(6)

lookup(19)

Figure: Binary search tree with total order “<”

# Binary Search Trees

## Implementation - Insert



### Insert:



### **Insert:**

- We search for the key in our search tree

### Insert:

- We search for the key in our search tree
- If a node is found we replace the value with the new one

### 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

### 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



### 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

### 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 “<”



**Remove:** Case 1: The node “5” has no children



**Remove:** Case 1: The node “5” has no children

- Find **parent** of node “5” (“6”)

**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”

**Remove:** Case 1: The node “5” has no children

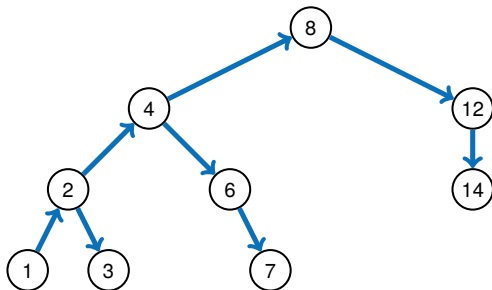
- Find **parent** of node “5” (“6”)
- Set left / right child of node “6” to **None** depending on position of node “5”



Figure: Binary search tree with total order “<”

**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”



**Remove:** Case 2: The node “12” has one child





**Remove:** Case 2: The node “12” has one child

- Find the **child** of node “12” (“14”)



**Remove:** Case 2: The node “12” has one child

- Find the **child** of node “12” (“14”)
- Find the **parent** of node “12” (“8”)



**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”)

**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 “<”

**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 deleting node “12”



**Remove:** Case 3: The node “4” has two children



**Remove:** Case 3: The node “4” has two children

- Find the **successor** of node “4” (“5”)



**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”



**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

**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**

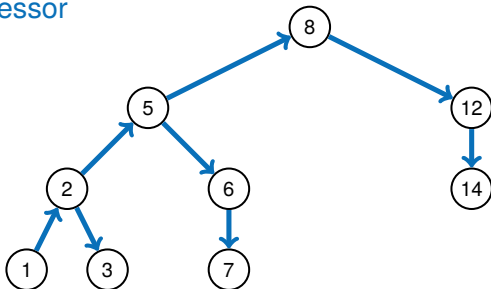
**Remove:** Case 3: The node “4” has two children

- 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**



**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**





How long takes `insert` and `lookup`?

### 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)

### 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  $\Theta(\log n)$

### 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  $\Theta(n)$

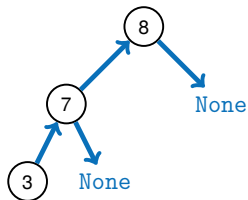


### How long takes **insert** and **lookup**?

- 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

### 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  $d = n$

### 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  $d = \log n$

## ■ General

[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`