# Implementing Data Structures

*1DV501/1DT901: Introduction to programming*

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

The slides are available in Moodle

October 6, 2020

# Today ...

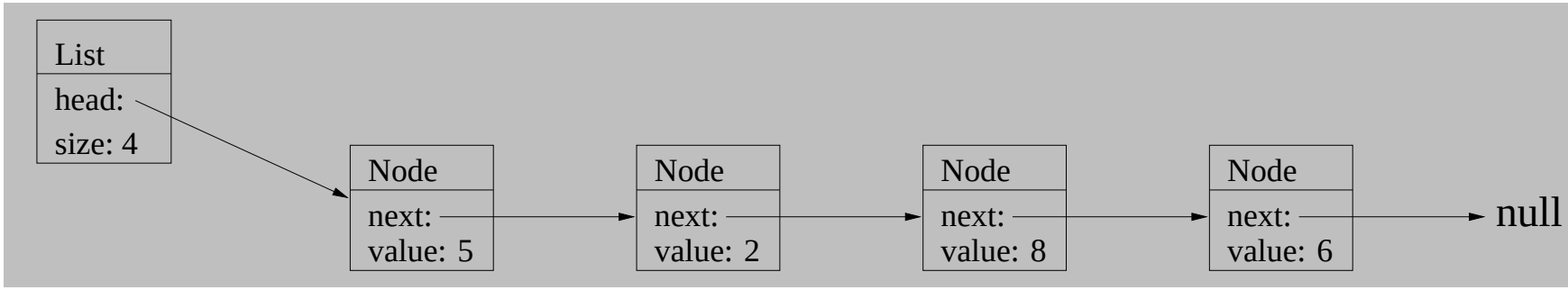
- Implementing linked lists
- Algorithms and Time Complexity
- Hashing
- Binary search trees
- Mini-project information

**Reading instructions:** Only these slides!

# Linked Lists

List can be implemented using a technique called **Linked Lists**

- Usually made up of two entities (list + node)
- The list holds a reference to the first node (the list field `head`)
- Each element is stored in a node (the node field `value`)
- Each node knows its predecessor node (the node field `next`)
- A user interacts with the list, nodes are encapsulated within the list class



# My linked list implementation

The following slides will show fragments of code from my linked list implementation `linked_list.py`. The implementation (`.py`-files) is also available in Moodle.

**Notice**

- An implementation using only Python features presented in the course
- Each node is a list of length 2: `[value,next]`
- The `head` node is our list reference ⇒
- ⇒ all list operations are applied on the head node
- User scenario:

```
import linked_list as ll

# Program starts
head = ll.get_new_head()          # Gives access to head node

# Add and print
for i in range(1,21):
    ll.add(head,i)                # Add elements to list
print( ll.to_string(head) )       # Print list content
```

Notice that we use the head node as argument in all calls.

# A Linked Implementation

Start of file `linked_list.py`

```
# A linked list using lists of size 2 as nodes. The head node
# looks like [None,next], the last node (tail) looks like
# [value,None], all other nodes look like [value,next]

# Returns head node to be used in subsequent calls
def get_new_head():
    return [None,None]

# Append value n to the end of the list
def append(head, n):
    node = head
    while node[1] != None:      # Find tail node
        node = node[1]          # Move to next node
    node[1] = [n,None]          # Attach new node to tail node
```

`while node[1] != None: node = node[1]` ⇒ move along the node chain until we find tail node identified as having `None` as next node (`node[1]`).

# The function `to_string(head)`

```
# Returns a string representation of the list content
def to_string(head):
    result = "{ "
    node = head[1]    # Node following head
    while node != None:
        result += str(node[0]) + " "  # Add to result string
        node = node[1]                # Move to next node
    return result + "}"
```

Usage:

```
print( ll.to_string(head) )  # { 1 2 3 ... 18 19 20 }
```

Hence, we move along the node chain and add `str(node[0])  " "`+ to the result string for each node.

# `count(head)` **and** `contains(head, n)`

```
# Returns the number of
# elements stored in list
def count(head):
    c = 0
    node = head[1]
    while node != None:
        c += 1
        node = node[1]
    return c
```

```
# Returns True if n is in
# list, otherwise False
def contains(head, n):
    node = head[1]
    while node != None:
        if node[0] == n:
            return True
        node = node[1]
    return False
```

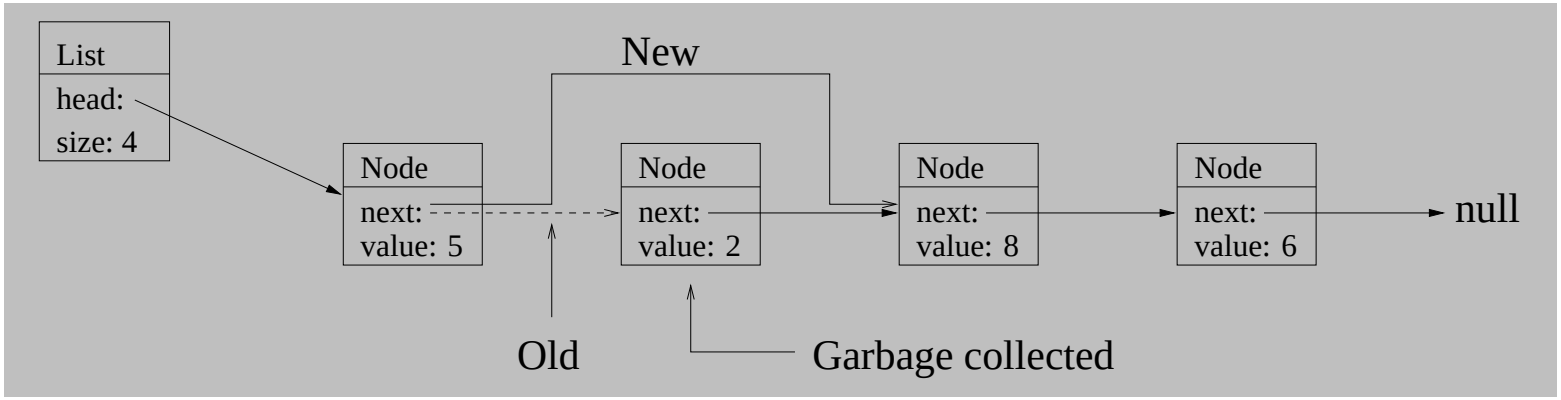
Same idea used twice, we start in node after head (`node = head[1]`) and move along node chain as long as we have more nodes (`while node != None:`).

# Function `get(head, pos)` raises exception

```
# Get element at position pos, raise IndexError
# if position pos is out of range
def get(head, pos):
    n = count(head)                 # Costly!
    if 0 <= pos < n:                # If valid pos
        node = head[1]
        for i in range(pos):        # Move pos steps forward
            node = node[1]
        return node[0]
    else:                           # Raise excepition
        msg = f"Index {pos} out valid range [0,{n-1}]"
        raise IndexError(msg)
```

Calling `count(head)` (which traverses the list) for each call to `get(head, pos)` is costly. Solution: Introduce global variable `size` keeping track of the current list size.

# The `remove(pos)` function



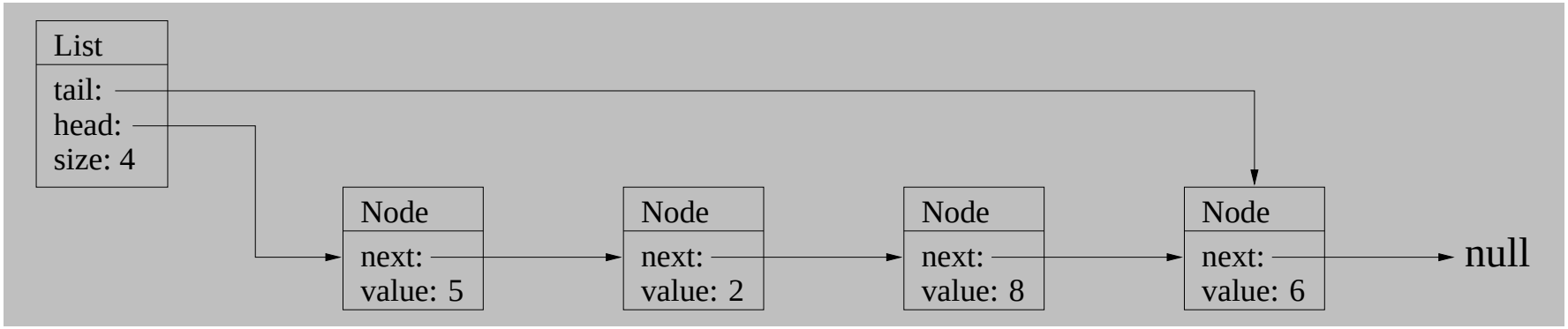

**Note:** We can't move backwards $\Rightarrow$ we must operate from the node *before* the node we want to remove.

```
# Remove and return element pos
def remove(head, pos):
    # Find node before pos
    before = head[1]
    for i in range(pos-1):      # Move pos-1 steps forward
        before = before[1]

    #By-pass pos node
    delete = before[1]          # Node to be removed
    before[1] = delete[1]       # By-pass "delete"
    return delete[0]
```

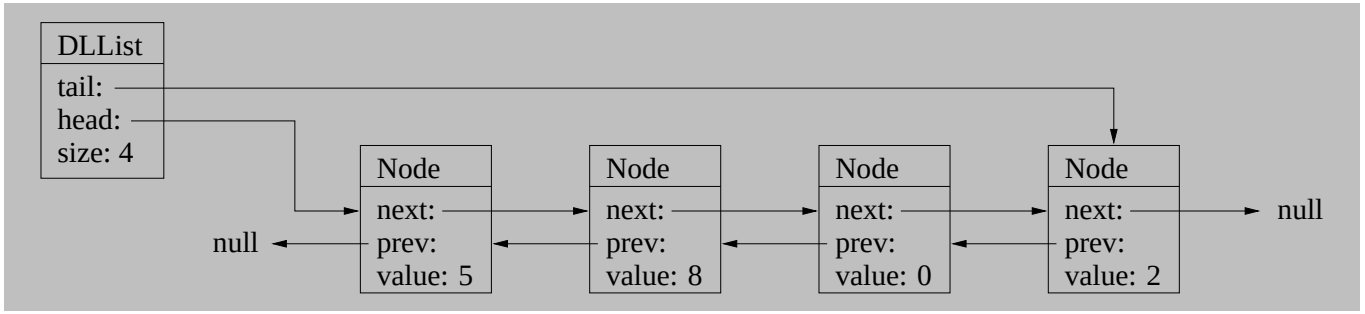
# Variant 1: Head and tail

**Problem:** `append()` ⇒ step through the whole list ⇒ very slow
(Serious since `append()` is a frequently used operation.)
**Solution:** Keep also track of the tail node ⇒ we can jump there directly



# Variant 2: Double-linked List

**Problem:** We can only traverse list in one direction
$\Rightarrow$ (for example) printing the list content backwards is very slow
**Solution:** Each node has a field `prev` that references the previous node.



# My linked list vs Pythons list

- Python lists are not linked lists
- Python lists are implemented as *dynamic arrays*
  (They start using a mutable C array of size (say) 1000 to store the data. They then double the size every time we need more size $\Rightarrow$ sizes are 1000, 2000, 4000, 8000, ...)
- Faster than linked lists in operations like `append` and `get`
  (since we don't need to move along the chain of nodes).
- Slower than linked list in operations like `remove(0)` since it must shuffle all elements one step left to cover for the hole at position 0.
- **Python lists are much (much!) faster since implemented in C**
- Many time critical parts in Python are implemented in highly optimized C code and called from the main program by the virtual machine.

However, linked data structures (one node referencing others) is an important concept that are used in trees and graphs.

# How do we recognize a good algorithm?

- We expect each algorithm to be *correct* . . .
- . . . but there might be more than one correct algorithm.
- Which one is the best?

Possible criteria:

- The algorithm is easy to understand and implement
  - simple
  - clearly written
  - well documented
  - . . .
- The algorithm is *efficient*
  - uses resources efficiently, for example memory or network capacity
  - time efficient $\Rightarrow$ fast!

We will concentrate on time efficiency but that does NOT mean that the other criteria are not important. (As a first try I will always go for the simple solution. It might be good enough.)

# Asymptotic Analysis

We would like to:

- Analyse algorithms without knowing on which computer they will execute
- Answer questions like "Which of these two algorithms are faster if the input size is big?".
- Answer questions like "How much will the computation time increase if the size of the input is multiplied by 2?"

We will achieve this by using *asymptotic analysis* and the big-oh notation $\Rightarrow$ a *Time Complexity* estimate.

Asymptotic Analysis $\Rightarrow$ Behaviour when input size is big.

# Time Complexity (Introduction)

Time Complexity: An estimate of required computation time.

- Number of required computations often depend on input data
  - Find integer in a list $\Rightarrow$ time depends on list size N
  - Check if N is a prime number $\Rightarrow$ time depends on N size
  - Sort list $\Rightarrow$ time depends on list size $N$
- We say that an algorithm have time complexity
  - $O(N)$ if computation time is proportional to $N$
  - $O(N^2)$ if computation time is proportional to $N^2$
  - $O(1)$ if computation time is constant
  - in general, $O(F(N))$ if computation time is proportional to $F(N)$
- $O(\dots)$ is pronounced *Big-Oh of* $\dots$. (Example Big-Oh of N-square.)
- or sometimes *Ordo of* $\dots$
- **Basic assumption: Each simple computation takes time 1**
- Simple operations: `+,-,\,*,%`, assignment, ...
- We are always interested in the **worst case scenario**
  $\Rightarrow$ the case requiring most computations

# Time Complexity: Examples

- Print multiplication table for N $\Rightarrow O(N^2)$

```
def print_table(N):
    for i in range(1,N+1):          # O(N)
        for j in range(1,N+1):      # O(N)
            print(i*j)
```

print statement is executed $N \times N$ times $\Rightarrow O(N^2)$

- Search for X in list of size N $\Rightarrow O(N)$

```
def search(X, lst):
   for n in lst:            # O(N)
      if n == X:            # O(1) executed N times
         return True
   return False
```

**Note:** A loop with N iterations over a body with time complexity $O(X)$
$\Rightarrow$ time complexity $O(N \cdot X)$

# Asymptotic Handling in Practise

Assume time $T(n) =$ in terms of input size $n$.

1. Constant factors do not matter.
2. In a sum, only the term that grows fastest is important.

- $T(n) = 3n \Rightarrow O(n)$,
- $T(n) = 4n^4 - 45n^3 + 102n + 5 \Rightarrow O(n^4)$,
- $T(n) = 16n - 3n \cdot \log_2(n) + 102 \Rightarrow O(n \cdot \log_2(n))$,
- $T(n) = 9168n^{88} - 3n \cdot \log_2(n) + 5 \cdot 2^n \Rightarrow O(2^n)$

- The $O(...)$ notation describes the behaviour when input size is big
- We are always interested in the *worst-case scenario*
  $\Rightarrow$ Not when we are finding an element at the first position in a list

# Frequent Big-Oh Expressions

| | |
|---|---|
| $O(1)$ | At most constant time, i.e. not dependent on the size of the input. |
| $O(\log n)$ | At most a constant times the logarithm of the input size. |
| $O(n)$ | At most proportional to $n$. |
| $O(n \log n)$ | At most a constant times $n$ times the logarithm of $n$. |
| $O(n^2)$ | At most a constant times the square of $n$. |
| $O(n^3)$ | At most a constant times the cube of $n$. |
| $O(2^n)$ | At most exponential to $n$. |

They are ordered from fastest ($O(1)$) to slowest ($O(2^n)$).

# Linear Search

- Problem: Find $x$ in list with $N$ elements
- Basic Idea: Sequential search

```
def search(X, lst):
    for n in lst:          # O(N)
        if n == X:         # O(1) executed N times
            return True
    return False
```

- We must check every element in the list
  $\Rightarrow O(N)$, where N is the list/array size.

Q: Do we have better algorithms?

A: No, not for an arbitrary list (in a single-core machine).

# Binary Search

- Problem: Find $n$ in list with $N$ elements
- **Assumption: The list is sorted**
- Basic idea: Look at the middle element $m = arr[M]$
  - If $n = m$, return *True*
  - If $n < m$, repeat search in [0,M-1]
  - If $n > m$, repeat search in [M+1,N]
- Each "search" halves the problem
  $\Rightarrow T(N) = T(N/2) + O(1)$
- $n$ not in list $\Rightarrow$ empty list in next search

```
Find 8 i [1,3,5,7,8,9,10] ==> middle element is 7 ==>
Find 8 i [8,9,10] ==> middle element is 9 ==>
Find 8 i [8] ==> OK!
```

- Much faster than linear search
  $\Rightarrow$ Might be worth sorting the list if searched many times.

# Binary Search

- Steps (time) required to search list of different sizes
  - Size: $1 \Rightarrow$ Time $= 1$
  - Size: $2 \Rightarrow$ Time $= 2$
  - Size: $4 \Rightarrow$ Time $= 3$
  - Size: $8 \Rightarrow$ Time $= 4$
  - Size: $16 \Rightarrow$ Time $= 5$
  - Size: $32 \Rightarrow$ Time $= 6$
  - ...
  - Size: $2^p \Rightarrow$ Time $= p + 1$
- Thus, $N \propto 2^t$ (Size as a function of time)
- $\Rightarrow t \propto \log_2(N)$ (Time as a function of size)
- $\Rightarrow T(N) = O(\log_2(N))$

In general, an algorithm that halves the problem in a fix number of computations has time-complexity $O(\log_2(N))$

# Recursive Binary Search

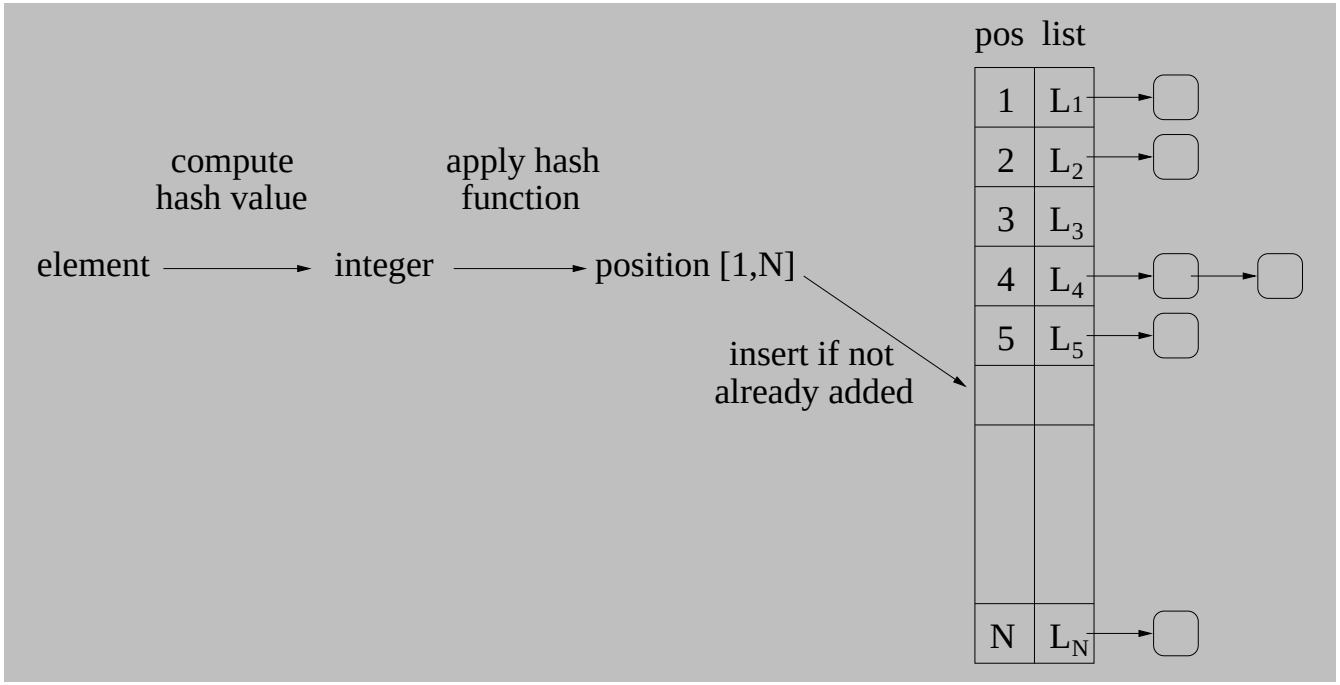
```
# Returns True if x in list lst, otherwise False
def binary_search(lst, low, high, x):
    if high < low:  # Element is not present in the list
        return False
    else:
        mid = (high + low) // 2   # Find mid position
        if lst[mid] == x:
            return True
        elif lst[mid] > x:        # Search lower half
            return binary_search(lst, low, mid - 1, x)
        else:                     # Search higher half
            return binary_search(lst, mid + 1, high, x)

# Program starts
arr = [ 2, 3, 4, 10, 40 ]
x = 10
if binary_search(arr, 0, len(arr)-1, x):
    print("Element is present at index", str(result))
else:
    print("Element is not present in array")
```

# A 10 Minute Break

zzzzzzzzzzzzz ...

# Hashing – A Brief Presentation




**A hash based set implementation**

Assume table with *N buckets* (A pair position/list)

- Associate each element with a hash value (an integer): `element --> int`
- Apply hash function (maps hash value to a bucket): `int --> bucket`
- Add to the bucket (the list part) if not already added

# Hashing – A Concrete Example

**A hash table for strings**

Assume that ...

- We have a table with 64 *buckets* (current bucket size)
- We compute the hash value for a string by summing up the ASCII codes for each character
- We use a simple modulus operator (`... % 64`) as our hash function

**Example**

- Adding `"Hello"` $\Rightarrow$ hash value 500 ($= 72 + 101 + 108 + 108 + 111$)
  $\Rightarrow$ bucket 52 (since `500 % 64 = 52`)
  $\Rightarrow$ insert `"Hello"` in bucket 52 (if not already added)
- Adding `"Jonas"` $\Rightarrow$ hash value 507 $\Rightarrow$ bucket 59 (`= 507 % 64`)
  $\Rightarrow$ insert `"Jonas"` in bucket 59 (if not already added)

# Hashing – Result

Assume that:

- all elements are evenly distributed across all buckets
  $\Rightarrow$ puts demands on the hash values/functions
- number of elements $\approx$ number of buckets
  $\Rightarrow$ average bucket size is $\approx 1$

Table access then involves:

1. Compute hash value
2. Decide which bucket to use
3. Search list (of average size 1)

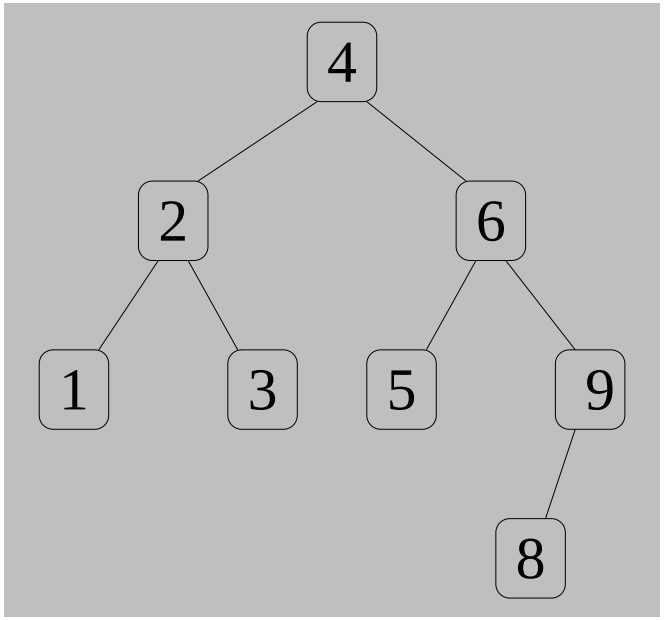
**Result:** `add`/`contains`/`remove` executes in fix number of steps independent of the number of stored elements $\Rightarrow O(1)$

# Rehashing

- Number of elements $\approx$ number of buckets in order to maintain $O(1)$ for `add/contains/remove`
- Hence, number of buckets must increase when we add more elements
- This process is called **rehashing**
- For example, each time number of elements equals number of buckets

  1. Make a copy of bucket list
  2. Clear bucket list and and make it twice as large
  3. For each element in the copy: add it to enlarged bucket list using the add function
  4. Continue with the enlarged bucket list
- Notice
  - Rehashing only occurs at certain points (when number of elements equals number of buckets)
  - We double the bucket list size each time $\Rightarrow$ 100, 200, 400, 800, 1600, 3200, ...
  - It is important that you add all elements using the add function to make sure that each of them is inserted in the correct bucket in the new enlarged bucket list.

# Binary Search Trees (BST)




**Note:**

- A *tree* consists of *nodes*
- The top-most node (4) is called the *root*
- *Binary trees* $\Rightarrow$ a maximum of two children for each node
- *Binary search trees* $\Rightarrow$ left child is always smaller than right child

**Question:** Where should 7 be placed?

# Implementing Binary Search Trees (BST)

The following slides will outline the basic ideas for how to implement a set using binary search tree.

- It is not Python code! (Starting point is Java)
- Each `node` has three attributes: `node.value`, `node.left`, `node.right` storing the node value, and it's left and right child
- `node.left` (or `node.right`) equals `null` $\Rightarrow$ no such child
- In a BST based dictionary (map or table) each node would have four attributes: `node.key`, `node.value`, `node.left`, `node.right`
- Implementing a BST based map is a part of the mini-project
- Implementing a hashing based set is a part of the mini-project

# The recursive function `add(node, n)`

Add value `n` to the tree. Initially called as `add(root, n)`

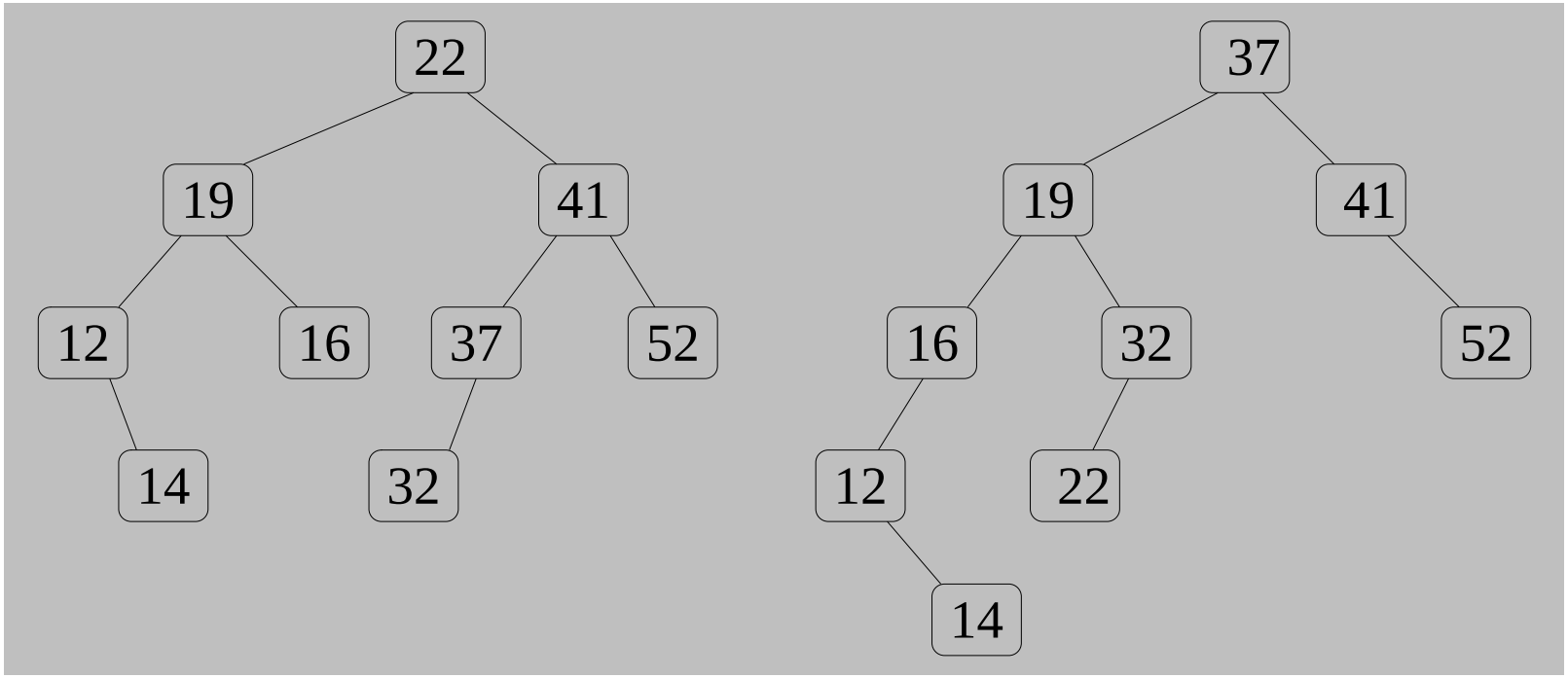
```
add(node, n) {  // recursive add
    if (n < node.value) {              // add to left branch
        if (node.left == null)
            node.left = new Node(null,null,n)
        else
            add(node.left, n)          // Recursive call
    }
    else if (n > node.value) {         // add to right branch
        if (node.right == null)
            node.right = new Node(null,null,n)
        else
            add(node.right, n)         // Recursive call
    }
}
```

- The recursive functions describes what we do in each node
- If value n less than current node value:
  - If node has no left child ⇒ attach new node as left child
  - If node has left child ⇒ call add with left child as input
- Note: `n == node.value` ⇒ duplicate element ⇒ we do nothing

# Binary Search Trees: Two Examples

Ex1: 22,19,41,52,37,16,12,14,32

Ex2: 37,19,16,12,14,32,41,52,22




**Notice:**

- **Error in first figure! 16 is at wrong position!**
- Same elements added in different order $\Rightarrow$ two different trees
- No duplicated entries

Recursive method for look-up?

# The recursive function `contains(node, n)`

Returns true if value `n` is in the tree, otherwise false.
Initially called as `contains(root, n)`

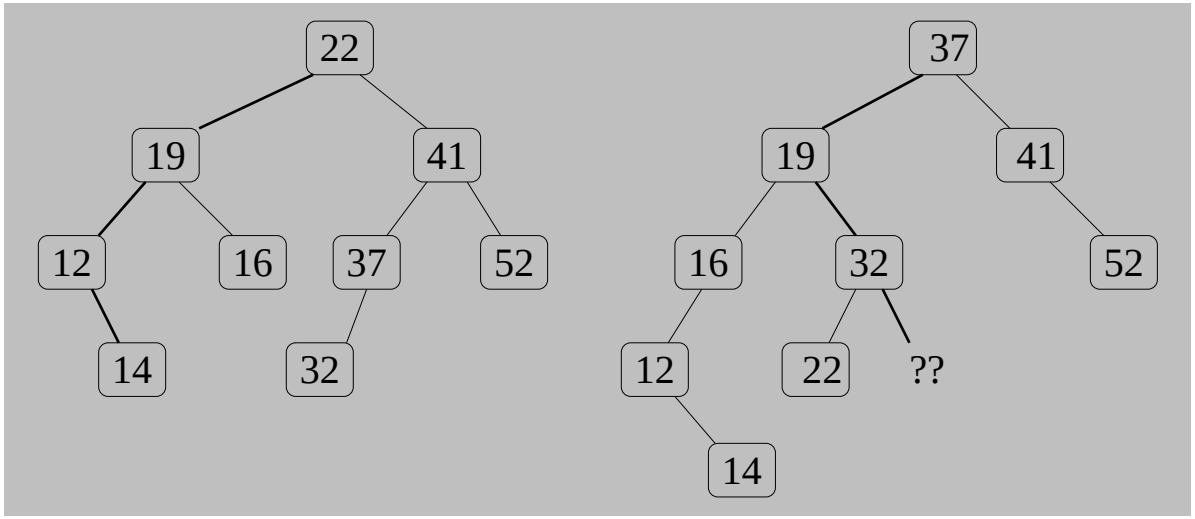
```
contains(node, n) { // recursive look-up
    if (n < node.value) {       // search left branch
        if (node.left == null)
            return false
        else
            return contains(node.left, n);
    }
    else if (n > node.value) { // search right branch
        if (right == null)
            return false
        else
            return contains(node.right, n);
    }
    return true;                        // Found!
}
```

- Similar to `add` but we return false when we find a missing child

# Binary Search Trees: Two Examples

Ex1: Search for 14

Ex2: Search for 34




**Notice:**

- Search 14: completed after 4 steps
- Search 34: completed after 3 steps
- Similar to Binary Search in sorted list
- In general: A search in a tree with $N$ elements requires $\log_2(N)$ steps
  $\Rightarrow$ Time-Complexity for `add`, `remove`, `contains` is $O(\log_2(N))$

**Exercise:** Find insertion order for `1,2,3,4,5,6,7` that (on average) gives:

- a) the fastest search? b) the slowest search?

# Balanced Trees and Speed

From previous slide: fastest search: 4,2,6,1,3,5,7, slowest search: 1,2,3,4,5,6,7

- Balanced tree $\Rightarrow$ uniform tree with minimum depth
- $\Rightarrow$ Every level of the tree is full
- A balanced tree with depth $n$ contains $2^{n+1} - 1$ elements
- depth $n \Rightarrow 2^{n+1} - 1$ elements can be searched in $n$ steps
- Examples
  - $n = 10 \Rightarrow$ tree size 2047
  - $n = 15 \Rightarrow$ tree size 65535
  - $n = 20 \Rightarrow$ tree size 2097151
  - $n = 30 \Rightarrow$ tree size 2147483647
  - $n = 40 \Rightarrow$ tree size 2199023255551
- This is very fast compared to sequential search for larger sets
- Microseconds rather than seconds
- More advanced BST algorithms (e.g. Red-Black Trees) always keep the tree balanced $\Rightarrow$ no need to worry about adding elements in a certain order.

# Time-complexity for Hashing and BSTs?

Time-complexity for lookup in hash tables and binary search trees?

## Hash tables

- Assume number of buckets $\geq$ number of elements and that elements are evenly distributed over all buckets. We can then look up an element in three steps
  1. compute hash value
  2. identify bucket
  3. traverse (very short) list

  $\Rightarrow$ A fix number of computations (independent of table size) $\Rightarrow O(1)$

## Binary Search Trees

- 1) Each visited node halves the number of remaining elements, and
  2) The number of operations performed in each node is fix
  $\Rightarrow$ Very similar to binary search $\Rightarrow O(\log_2(N))$

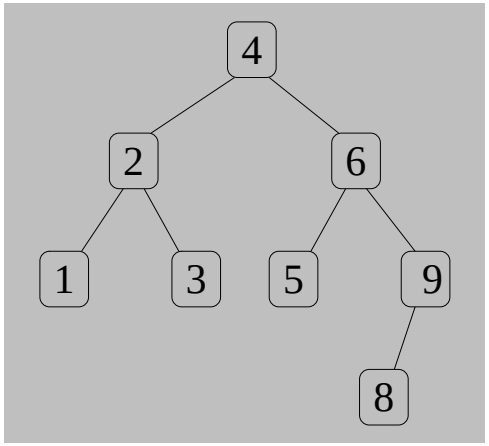
# remove(...) – A nightmare, dropped!

```
remove(int n) {
    if (n<value) {
        if (left != null)  left = left.remove(n);
    }
    else if (n>value) {
        if (right != null) right = right.remove(n);
    }
    else {  // remove this node value
        if (left==null) return right;
        else if (right==null) return left;
        else {                          // The tricky part!
            if (right.left == null) {
                value = right.value;
                right = right.right; }
            else
                value = right.delete_min();
        }
    }
    return this;
}
int delete_min() {  // more code here ...
    if (left.left==null) {
```

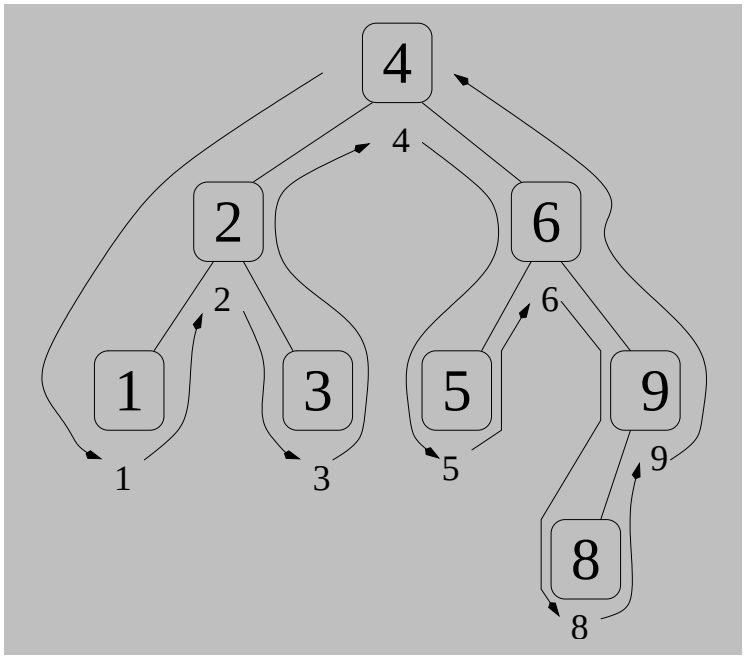
# The function `print()`

```
print_tree(node) {
    if (node.left != null)        // visit left child
        print_tree(node.left)
    print(" ", node.value)        // in-order print value
    if (node.right != null)       // visit right child
        print_tree(node.right)
}
```

Apply function on the following tree: What is printed?



# In-order visit




**Print-out:** `1,2,3,4,5,6,8,9,` $\Rightarrow$ BST are sorted in principle.
**Find min/max:**

- Always pick the left-most child $\Rightarrow$ the lowest added number
- Always pick the right-most child $\Rightarrow$ the highest added number

# Binary Tree Visiting Strategies

```
Left-to-Right, In-order
   visit left subtree (if exist)
   visit node                        ( Do something, e.g., print node value)
   visit right subtree (if exist)

Right-to-Left, Post-order
   visit right subtree (if exist)
   visit left subtree (if exist)
   visit node
```

- `Left-to-Right, Right-to-Left` ⇒ traversal strategies ⇒ decides in which order we visit the children ⇒ a left or right traversal around the tree
- `Pre-order, In-order, Post-order` ⇒ decides when we do something in the node ⇒ before (pre), in between (in), or after (post) we visit the children.

# Mini-project preview

- The mini-project starts on Wednesday $\Rightarrow$ problem task is published in Moodle
- Today (later on, Monday) we will present the project teams.
- Each team has three members
- The tutoring supervisors are putting together the teams
- Basic ideas
  - A team consists of students from the same program
  - Campus students with campus students
  - Distance students with distance students
  - Kalmar students with Kalmar students
  - ...
- Thus, we will try to put together students having a similar study situation
- Contact your tutoring supervisor if you end up in "wrong" team.