Hashing, BSTs, and Project Information

1DV501/1DT901: Introduction to programming

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

The slides are available in Moodle

October 16, 2022

Today ...

- Algorithms and Time Complexity
- Hashing
- Binary Search Trees
- Mini-project Information

Reading instructions: Only these slides!

What is an Algorithm?

An **algorithm** is a step-by-step description of how to solve a problem.
In addition to being correct it should:

1. Give an unambiguous result
   $\Rightarrow$ only one result for each input
2. Be unambiguously presented
   $\Rightarrow$ a precise formulation that can't be misinterpreted
3. Terminate after a finite number of steps
   $\Rightarrow$ no infinite computations.

An infinite computation of π

$$\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots)$$

- **Why Algorithms?**
  - Document problem solutions
  - Communicate problem solutions
  - Compare problem solutions (time complexity!)
  - Also, sketchy algorithms are a good preparation before programming.

Problem: Wash your hair

- Q: How do you wash your hair?
- A: Start by wetting the hair. Then rub in shampoo and rinse it away. Repeat shampoo/rinse until you feel clean. If it is cold outside, use a hair-drier, otherwise let it dry by itself.
- Algorithm:
  1. Wet hair
  2. Repeat until hair is clean
     - 2.1 rub in shampoo
     - 2.2 rinse shampoo away
  3. If cold outside
     - 3.1 use hair-drier
  4. Otherwise
     - 4.1 let the hair dry by itself

An algorithm is a precise description of a problem solution. It is often structured as a program $\Rightarrow$ easy to convert into a running program.

Pseudo Code - Two Examples

Problem: Can N be divided by 3?
Method 1: Structured ordinary text

1. Ask user for an integer. Denote this integer N
2. If N modulus 3 equals 0
   2.1 Inform user that N is dividable by 3
3. Otherwise
   3.1 Inform user that N is **not** dividable by 3

Method 2: Almost like a program

```
N : integer to be tested
if  N  mod 3 = 0 then
    N is dividable by 3
else
    N is not dividable by 3
end if
```

Notice: There are no exact rules for pseudo code.
Everything that is structured, unambiguous, and easy-to-understand is OK.
Q: Which of the above methods do you prefer?

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 $F(N)$
- $O(\ldots)$ is pronounced *Big-Oh of* .... (Example Big-Oh of N-square.)
- or sometimes *Ordo of* ...
- **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.

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

- The $O(\ldots)$ 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 (on 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 $m = lst[M]$
  - 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!
```

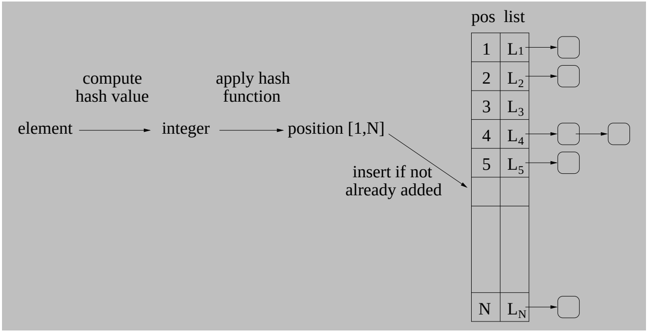
- Much faster than linear search
  $\Rightarrow$ Might be worth sorting the list if searched many times.

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 $O(\log_2(N))$

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)

Add n to hash table

Assume a bucket list containing N buckets where each bucket is a list.

add(n) $\Rightarrow$ add element n to table

1. Hashing: Associate n with a positive integer h (the hash value) (For example, sum of all ASCII for a string)
2. Find bucket: Associate h with a bucket $b \in [0, N-1]$ (For example $b = h \ \% \ N$)
3. Get list b from the bucket list
4. Add n to list b if not already added

contains(n) $\Rightarrow$ search table for element n

1. Steps 1 to 3 from "add" above
2. Return True if n is in list b, otherwise False

remove(n) $\Rightarrow$ remove n from table

1. Steps 1 to 3 from "add" above
2. Remove n from list b (if it exist)

Hashing – Result

Assume that:

- all elements are evenly distributed across all buckets
  $\Rightarrow$ puts demands on the hash values/functions
- number of elements $\leq$ number of buckets
  $\Rightarrow$ average bucket size is ≤ 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 $\Rightarrow O(1)$
Notice: Time-complexity $O(1)$ makes hashing much faster than lists (time-complexity $O(n)$) for large data sets.

Rehashing

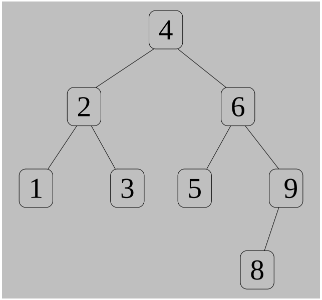
- We need the number of elements $\leq$ number of buckets in order to maintain $O(1)$ for `add/contains/remove`
- Hence, number of buckets must increase when we add more elements
- This process is called **rehashing**
- For example, each time number of elements equals number of buckets

  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.

A 10 Minute Break

zzzzzzzzzzzzz ...

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 hash 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)`

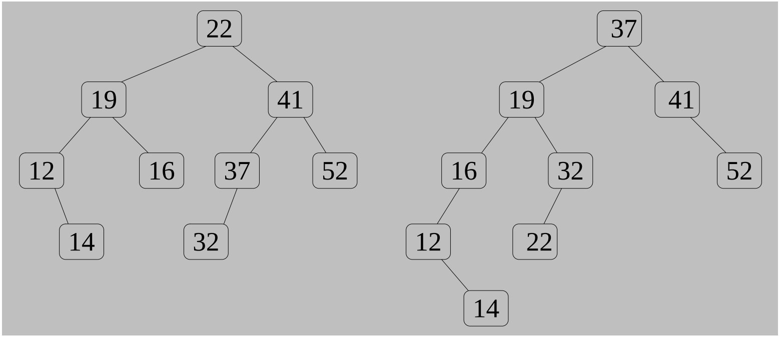
```
add(node, n) {  // recursive add
    if (n < node.value) {                 // add to left branch
        if (node.left == null)
            node.left = new Node(null,null,n)
        else
            add(node.left, n)             // Recursive call
    }
    else if (n > node.value) {            //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

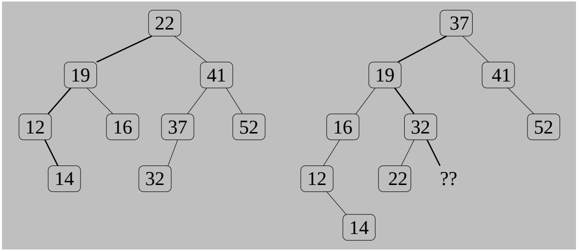
```
contains(node, n) { // recursive look-up
    if (n < node.value) {        // search left branch
        if (node.left == null)
            return false
        else
            return contains(node.left, n);
    }
    else if (n > node.value) { // search right branch
        if (right == null)
 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

Assume a rather balanced tree

- 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 $\Rightarrow O(\log_2(N))$

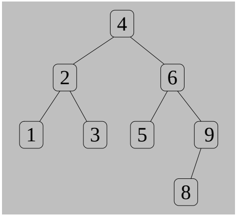
remove(...) – A nightmare, dropped!

```
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) {
```

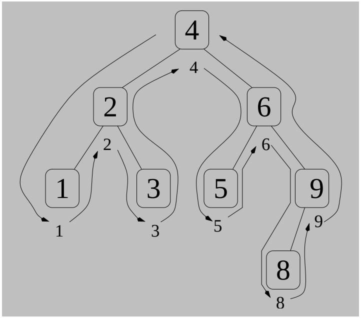
The function print()

```
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,` ⇒ BST are sorted in principle.
Find min/max:

- Always pick the left-most child ⇒ the lowest added number
- Always pick the right-most child ⇒ 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               (Do something, e.g., print node value)
```

- `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: Introduction

The Python Mini-Project is a small project exercise where you in a team (1-2 students) handle a given task which will be presented by the end of the course.

Deadline 1: Demonstrate project at a tutoring session before Deadline 2
Deadline 2: Submit code and report using GitLab, November 6 (at 23.59)

Mini-project information

- *The Mini-project* Moodle section is the main source of project information
- Here we will publish:
  - Rules
  - Project Instructions (the actual task to handle)
  - Templates for written report

Mini-project: Rules

- You work in teams of two. Register your team in sheet in Moodle.
- You choose your companion yourself – use, for example, Slack or the tutoring sessions to find someone to work with. If you are unable to find a companion, please contact your tutoring supervisor. You might be allowed to work alone.
- Help will be given at the tutoring sessions
- By the end of the course, before the given deadline, you will demonstrate your implementation at a tutoring session.
- Before the given deadline: Submit code and report using GitLab.
- The written report should follow a given template.
- All team members should be present at the demonstration and be prepared to answer questions regarding all parts of the project.
- Feel free to use information found on the Internet but:
  - Give a proper reference to the source
  - Be prepared to answer detailed questions, you must understand what you are doing
- Plagiarism is cheating! Any sign of plagiarism $\Rightarrow$ all involved students fail (also students giving away their code).

Tutoring Supervisors and Sessions

Each student group has a senior tutoring supervisor

- Ola Flygt: NGDNS-en, NGDNS-sv, TGI1E
- Tobias Andersson-Gidlund: NGDPV-sv, NGDPV-en, TGI1E
- Tobias Olsson: TGI1V, NGFYR
- Jonas Lundberg: TGI1D, NGMAT/Exchange, CTMAT, CIDMV

Contact your senior tutoring supervisor (or post a message in Slack) if you have any project related questions.

Project Tutoring

- Project help is given at the tutoring sessions. See time schedule for details.
- Project demo is also handled at the tutoring sessions
- Project tutoring is handled by the TAs and (often) your senior tutoring supervisor.

Project Deliverables

- The team effort is presented as:
  1. A project demonstration at a tutoring session.
  2. A written report available in Gitlab
  3. Entire team code available in Gitlab
- All team members should be present at the project demonstration and be prepared to answer questions regarding all parts of the project
- Code and report should be available in Gitlab at the given deadline (23.59)
- The project demonstration should take place before the given deadline
- Template for written report is available in Moodle

How to work as a team

For each team we suggest the following:

- Get started right away. Get in contact with your team members. Establish ways of communication.
- Daily meetings to give status updates. Progress made, problems faced.
- A team helps each other. You win (high grade) and lose (fail) as a team.
- Ask project supervisor if you don't understand a certain part of the task formulation. Better to ask for the way than to walk in the wrong direction!
- Supervisor will not act as project leader dividing work and telling you what to do. Each team leads themselves.
- Remember: Writing the report takes 1-2 days!

The Mini-project Programming Tasks

The project is about understanding hashing and binary search trees.

The problem can be divided in three parts:

1. Count unique words using Python's set and dictionary
2. Implement two data structures suitable for working with words as data:
    2.1 A hash based set, and
    2.2 binary search tree (BST) based map (dictionary).
3. Use your two data structures to repeat Part 1 (counting unique words)

Parts 1 and 3 use the word files you produced in Assignment 3.

Part 1: Counting unique words 1

In Exercises 8 and 9 in Assignment 3 you saved all words from the two text files `swe_news.txt` and `life_of_brian.txt` in two separate files. (Do it now if you haven't done this exercise already).

Your task here is to:

1. Use Python's set class to count the number of unique words in each file,
2. Use Python's dictionary class to produce a Top 10 list of the ten most frequently used words having a length larger than 4 in each file.

In Part 3 you will repeat the same computations using your own hash and BST based implementations.

Part 2: Implement data structures

Lecture 10 outlines the basic ideas of two implementation techniques:

1. Binary search trees
2. Hashing

Your task is to implement:

- a set `HashSet.py` (suitable for words) based on hashing
- a map `BstMap.py` (key-value pairs) based on binary search trees.

Both `HashSet` and `BstMap` are data classes. Follow the instructions in Lecture 10 about hashing and binary search trees. Look at the linked list example in Lecture 9 to see an example of how to use data classes.

Part 2: Additional limitations

- The BST based map is a linked implementation where each node has four fields (key, value, left-child, right-child).
- The hash-based set is built using a Python list to store the buckets where each bucket is another Python list. The initial bucket list size is 8 and rehashing (double the bucket list size) takes place when the number of elements equals the number of buckets.

Code skeletons outlining which methods we expect for each data structure are available. They also contains an example program showing how the various methods can be used.
Notice: You are not allowed to make any changes of the method signatures in

the given skeletons. Also, the demo programs should work as outlined in the provided example programs once your implementations are complete.

Part 3: Count unique words

In this exercise you should basically repeat Part 1 using your own data structures rather than Python's.

1. Count how many unique words that are used in the two given texts files using your `HashSet` implementation.
2. Also, print a) bucket list size, b) maximum bucket size, c) zero bucket ratio, for each file
3. Present a list of the top-10 most frequently used words having a length larger than 4 using your `BstMap` implementation.
4. Also, print a) number of tree nodes, b) max tree depth, c) leaf count, for each file

Notice: You are not allowed to use Python's set and dictionary classes to solve these problems. The results for the two parts should be the same as in Part 1.

This is it!

- This is the last lecture!
- You are now Python programmers!
- Regarding the project
  - Find a team member!
  - Read rules and instructions carefully
  - Start working right away
  - Problems? Visit tutoring sessions or post a question in Slack
  - Awkward team situation? 1) Try to sort it out, you are grown ups, 2) Contact your project supervisor.
- Good luck with the project!
- Good luck with your future programming!