UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

# COS212 - Data structures and algorithms

## Assignment 2 Specifications

Release date: 15-04-2024 at 06:00
Due date: 10-05-2024 at 23:59
Total marks: 165

# Contents

# 1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually; no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of the built-in Java data structures. Doing so will result in a zero mark. You may only use native 1-dimensional arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experiences a runtime error, you will be awarded a zero mark. Runtime errors are considered unsafe programming.
- Read the entire specification before you start coding.
- **Ensure your code compiles with Java 8**
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.

# 2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent), and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers to avoid any misunderstanding.** Also note that the OOP principle of code reuse does not mean that you should copy and adapt code to suit your solution.

# 3 Outcomes

On completion of this assignment, you will have gained experience with the following:

- Hashing, including:
  - Insertion
  - Deletion
  - Retrieval
  - Internal Maintenance
- B-Trees, including:
  - Insertion
  - Traversal

# 4 Introduction

## 4.1 Hashing

Hashing is a fundamental concept in computer science, used to efficiently store, retrieve, and manage data in a way that minimizes the need to search through all the data to find a specific item. It involves mapping data of arbitrary size to fixed-size values, known as hash codes, which are then used to index a hash table containing the keys and values. The efficiency of hashing comes from the direct access it provides to the memory location where the desired data is stored, making data retrieval operations extremely fast, ideally in constant time O(1).

However, one of the challenges with hashing is handling collisions—situations where different keys map to the same hash code. Various collision resolution strategies have been developed, among which double hashing stands out for its probing technique.

Double hashing is a form of open addressing where the interval between probes or the "step size" is computed using a second hash function. Unlike linear or quadratic probing, where the step size is fixed or determined by a fixed formula, double hashing uses a second hash function to calculate the step size. This method offers several benefits including reduced clustering and increased adaptability to different data distributions.

To implement double hashing effectively, both hash functions must be chosen with care. The first hash function usually determines the initial position, while the second hash function determines the step size for probing. It's crucial that the second hash function is relatively prime to the table size to ensure that all slots can be probed, thereby avoiding infinite loops and ensuring that the hash table remains effective even at high capacities.

## 4.2 BTrees

B-Trees are a type of self-balancing search tree that are especially useful for storage systems where reads and writes are expensive operations, such as databases and file systems. The structure of B-Trees allows them to maintain sorted data in a way that ensures efficient access, insertion, and deletion operations, all of which are crucial for the performance of disk-based storage systems.

A B-Tree of order $m$ is a tree which satisfies the following properties:

- Every node has at most $m$ children.
- Every node (except root and leaves) always has at least $\lceil m/2 \rceil$ children.
- The root has at least two children if it is not a leaf node.
- A non-leaf node with $k$ children contains $k - 1$ keys.
- Keys within a node are kept in a sorted order, and the sequence of nodes follows a pattern where:
  - Subtrees to the left of a key contain elements less than the key.
  - Subtrees to the right of a key contain elements greater than the key.

B-Trees are extensively utilized in databases and file systems, where their ability to efficiently handle large datasets and optimize disk accesses makes them invaluable for indexing and managing file storage. Their high fan-out and balanced nature facilitate quick data access, insertion, and deletion operations, ensuring optimal performance even with vast amounts of data.

# 5 Scenario

## 5.1 Introduction

In this assignment, you will first create a simple game using hash maps and b-trees. You will then create a bot to play the game. Your bot should outperform a mostly random implementation.

## 5.2 The game

In this game, the player is an explorer searching through multiple areas for a single treasure. The player has a map which they can use to fast-travel (or jump) to any area at any time as long as they know its name. When in an area the player can perform predefined actions such as moving to the next space, moving to the previous space, ascending a level or descending a level. Each space in an area contains either:

- Nothing
- A clue which is one of the following:
  - An area clue, which narrows down the area of the treasure
  - A range clue which narrows down the location of the treasure within an area.
- The treasure itself.

The player must use these clues to find the treasure in the lowest number of moves.

## 5.3 Implementation

Each area is represented by a B-Tree and each space in an area is a key in the B-Tree. The following commands can be used to traverse the tree:

- "N": this command moves the player to the next key in the same node.
- "P": this command moves the player to the previous key in the same node.
- "A": this command moves the player to the first key in the parent of the current node.
- "D": this command moves the player to the first key in the child at the same index as the current key. Note that in order to descend to the $m$th child the player must be able to stand at the last index even though there is no key there. I recommend adding an extra space to the key array to allow this.[1]

Below is an example of the result of each type of movement. See Figure 1

Each area will be stored in a hash map (See Figure 2). The key will be the name of the area and the value will be the BTree which represents the area. At any time a player can choose to jump ("J") to another area. After choosing to jump the player will have to select an area, by name, to jump to. This should set the current node to the root of the new area and the current index to 0. In other words, players will always start at the first key in the root of a new area.

In order to investigate the current key the player will have to call `visit` on the key. See the given `Player` hierarchy for how to interact with the `PlayerMap`.

[1] This extra space should not hold a key (although it could be useful when overflows occur)

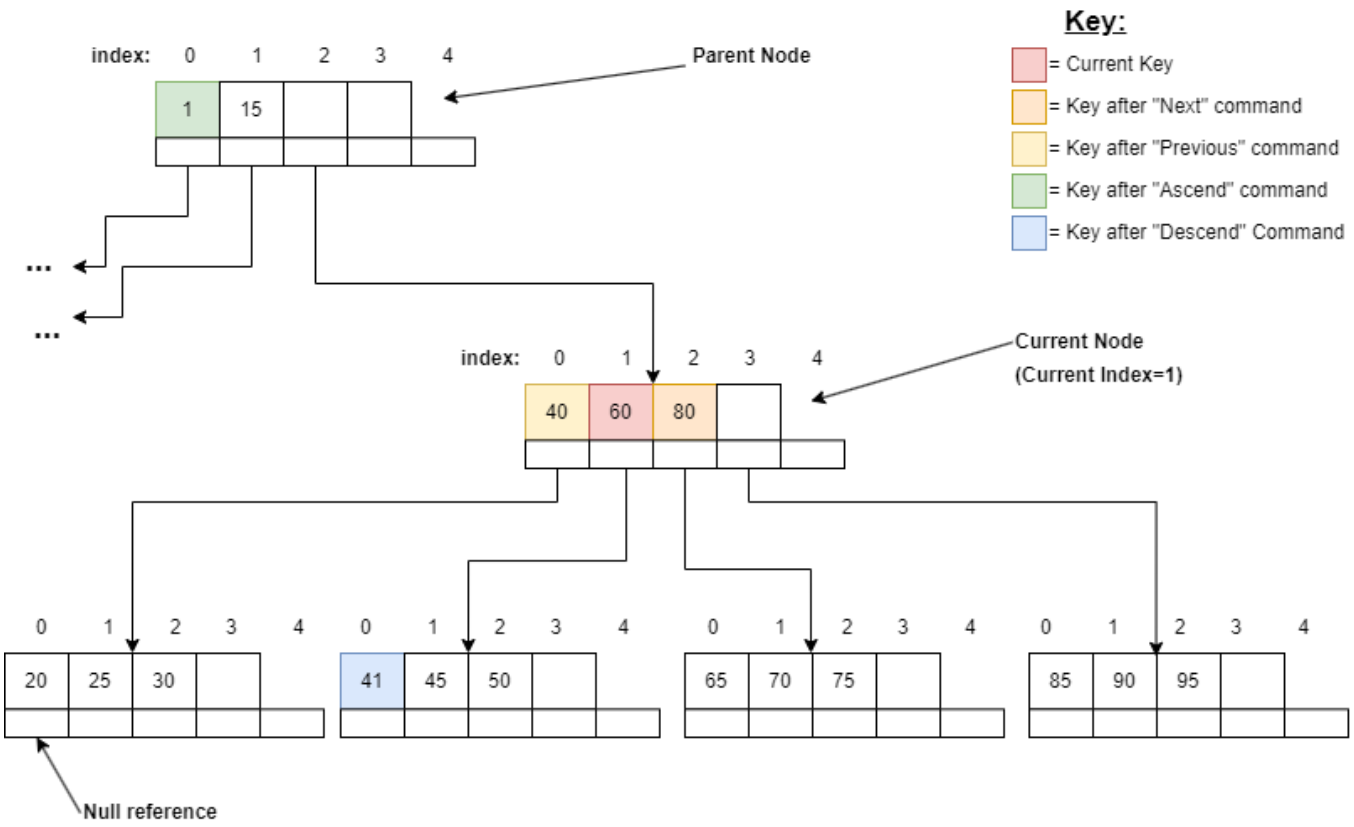


Figure 1: Example showing movement in the BTree using commands

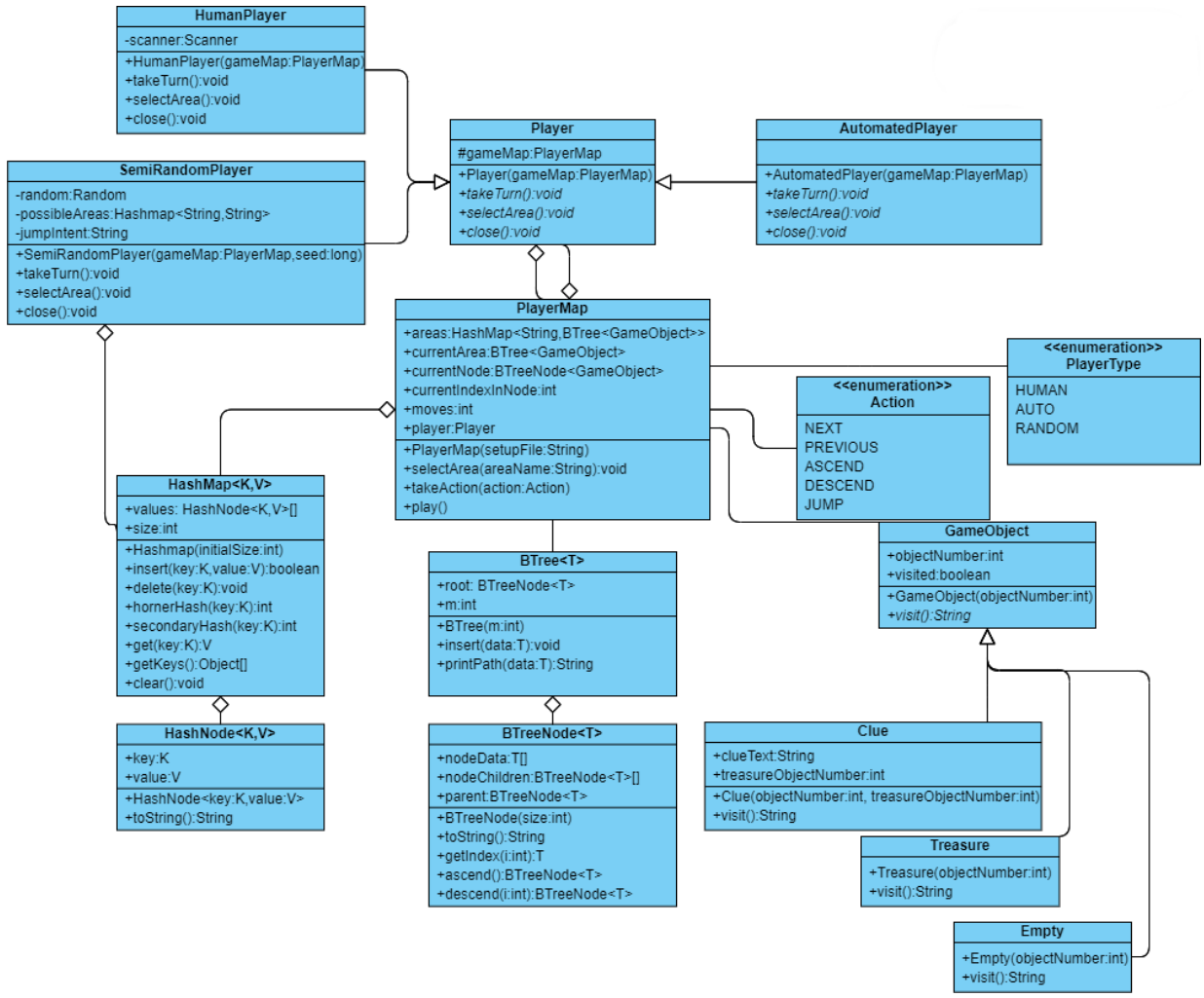


Figure 2: UML class diagram of the full game system

# 6 Tasks

You are tasked with implementing a hash map, a B-tree and a bot that plays the game described in Section 5. The `GameObject` hierarchy as well as `Player` class, `HumanPlayer` class, `SemiRandomPlayer` class and `PlayerMap` class have been provided for you. Also, `HashNode` has been completed for you. Please do not change any of these classes. You are, therefore, responsible for the `Hashmap`, `BTree`, `BTreeNode` and `AutomatedPlayer` classes. Your implementation must adhere to the given class diagram (Figure 2).

## 6.1 Hashmap<K,V>

- Members:
  - data: HashNode<K,V>[]
    - This is the array that stores all the data within the hash map.
    - Its initial size should be 2 and its size should double when it runs out of space.
  - capacity: int
    - This is the total capacity of the data array within the hash map.
    - The initial capacity, and therefore the initial size of the data array, should be 2.
  - numValues: int
    - This reflects the number of values in the hash map.
    - It should always be the same as the number of non-null values in the data array of the hash map.
- Methods:
  - Hashmap()
    - Constructor to initialize the hash map.
    - This should initialise the array, set the capacity to 2 and set the number of values to 0.
  - insert(key:K,value:V):boolean
    - Inserts a new HashNode into the data array at the correct index based on the hashed key.
    - Collisions should be handled using double hashing.
    - Use the given `hornerHash` as the primary hashing function and use the given `secondaryHash` as the secondary hashing function i.e. step size.
    - This function should return `true` if the item was inserted successfully. If the key was already present in the hash map nothing should change (the value should not be updated) and `false` should be returned.
    - If the hash map is full after an insertion then the capacity should be doubled and all the key-value pairs currently in the hash map should be reinserted into their new, correct index in the data array. When reinserting you should reinsert in the order the key-value pairs appear in the data array.
  - delete(key:K):void
    - If the key is present in the hash map then the corresponding index in the data array should be set to null. The number of values in the map should also be decremented
    - If the key is not present in the hash map nothing should happen.
  - get(key: K): V

    - Returns the value associated with a particular key in the hash map.
    - If the key is not present in the hash map null should be returned.
  - getKeys(): Object[]
    - Returns an array of type Object with all the keys currently in the hash map in the same order as they appear in the data array.
    - Null entries should, of course, not be added to the array of keys.
  - clear():void
    - Clears the data array and resets the size to 2
    - Resets the capacity of the hash map to a size of 2
    - Sets the number of values in the hash map to 0.
  - hornerHash(key:K):int
    - Returns the hashed value of the key using Horner's hashing algorithm. The ASCII values of the key should be used.
    - This has been implemented for you.
  - secondaryHash(key:K):int
    - Returns the hashed value of the key using a hashing algorithm which is guaranteed to return a result which is coprime with the capacity of the hash map.
    - This has been implemented for you.
  - toString():String
    - Returns a string representation of the hash map.
    - This has been implemented for you.

## 6.2 BTreeNode<T extends Comparable<T> >

- Members:
  - nodeData: Comparable<T>[]
    - This array holds the keys that are stored in the node.
    - Its logical size should be $m - 1$. However, making it larger in your implementation may be advantageous when dealing with overflows.
  - nodeChildren: BTreeNode<T>[]
    - This array holds the references to the node's children.
    - Its logical size should be $m$. However, making it larger in your implementation may be advantageous when dealing with overflows.
  - parent: BTreeNode<T>
    - This is the parent of the current node
    - Make sure that the parent is set correctly after splitting and merging.
  - size: int
    - This represents the size, $m$, of the node.
    - This should match the order of the tree that contains this node.
- Methods:
  - BTreeNode(size:int)
    - The constructor for a BTreeNode.

    - This sets the size member variable and initialises the nodeData and nodeChildren arrays to the correct sizes.
  - getIndex(i:int):T
    - Returns the key at the specified index in the nodeData array.
    - If the index is out of bound then null should be returned.
  - ascend():BTreeNode<T>
    - Returns the parent of the current node.
  - descend(i:int):BTreeNode<T>
    - Returns the child node at the specified index.

## 6.3 BTree<T extends Comparable<T> >

**NB: To disambiguate splitting, we only consider BTrees with an odd degree. BTrees with an even degree will not be tested.**

- Members:
  - root:BTreeNode<T>
    - This is the root of the BTree. It should be null when the tree is empty.
  - m:int
    - This represents the order of the BTree. Each node should have a maximum of $m$ children and $m-1$ keys.
- Methods:
  - insert(data:T)
    - This method should insert the data into the correct position in the BTree.
    - The insert should be done as discussed in the lectures.
    - The BTree should be able to handle duplicate keys. If a duplicate key is inserted then it should be inserted after the key(s) that is/are already there.
      For example, if we have:
      ```
      1,2,3,4
      ```
      and we insert another 2, we get:
      ```
      1,2,2,3,4
          ^ (new 2)
      ```
  - printPath(key: T):String
    - Returns a string representation of all the keys traversed to get to a particular key
    - The output should be the shortest path to the given data
    - The output should be in the following format:
      ```
      <root_data><space>-><space><step1_data><space>-><space><step2_data><space>
          -><space>...<space>-><space><stepN_data><space>-><space><goal_data>
      ```
    - For example with the given tree (Figure 1) the path to 70 is:
      ```
      1 -> 15 -> 40 -> 60 -> 65 -> 70
      ```
    - If the given data is not in the tree then go as far as possible and use `Null` as the `goal_data`. For example with the given tree (Figure 1) the path to 71 is:
      ```
      1 -> 15 -> 40 -> 60 -> 65 -> 70 -> Null
      ```

## 6.4 AutomatedPlayer

You have been given the skeleton for the `AutomatedPlayer` class. You should use this skeleton to create an `AutomatedPlayer` that should play the game in a similar way to the given `SemiRandomPlayer`. You may use the given `SemiRandomPlayer` to guide your implementation of the `AutomatedPlayer` class. However, your `AutomatedPlayer` should outperform the `SemiRandomPlayer` on any given seed. You may not change the `Player`, `SemiRandomPlayer` or the `PlayerMap` classes

Hints:

1. The `AutomatedPlayer` can access the `PlayerMap` through its superclass.
2. The `SemiRandomPlayer` does not take all clues into account
3. You may add attributes to the `AutomatedPlayer` class to assist with decision-making.
4. You may use the data structures that have been implemented within this assignment to assist with decision-making.

- Members:
  - Any members you need.
- Methods:
  - AutomatedPlayer(gameMap:PlayerMap)
    - The constructor for the AutomatedPlayer.
    - This must call the base class constructor to set the `PlayerMap`.
    - Any initialisation you need should be done here.
  - takeTurn():void
    - This is how your `AutomatedPlayer` chooses which action to take next.
    - This method is called from the `PlayerMap` from within the game loop.
    - This method should use the current state of `gameMap` to decide which action to take.
    - The legal actions are `N`, `P`, `A`, `D`, and `J`. The enum containing the legal actions can be accessed from the `PlayerMap` class.
    - If the action chosen is `J` then the `selectArea()` method will be called.
  - selectArea():void
    - This is how your `AutomatedPlayer` chooses which area to jump to.
    - This method will be called when the `J` action is chosen.
    - It should call at the end of its processing (if any) call `gameObject.selectArea(areaToJumpTo)`
  - close():void
    - This method is called when the game ends. It should be used to clean up any resources that are left open a the time.

# 7 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, approximately 10% of the assignment marks will be assigned to your testing skills. To do this, you will need to submit a testing main (inside the Main.java file) that will be used to test an Instructor-provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the jacoco tool. The following set of commands will be used to run jacoco:

```
javac *.java
rm -Rf cov
mkdir ./cov
java -javaagent:jacocoagent.jar=excludes=org.jacoco.*,destfile=./cov/output.exec
    -cp ./ Main
mv *.class ./cov
java -jar ./jacococli.jar report ./cov/output.exec --classfiles ./cov --html
    ./cov/report
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

and we will scale this ratio according to the size of the class.
The mark you will receive for the testing coverage task is determined using table 1:

| Coverage ratio range | % of testing mark |
|---|---|
| 0%-5% | 0% |
| 5%-20% | 20% |
| 20%-40% | 40% |
| 40%-60% | 60% |
| 60%-80% | 80% |
| 80%-100% | 100% |

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the instructor-provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

# 8 Upload checklist

The following files should be in the root of your archive

- AutomatedPlayer.java
- BTree.java
- BTreeNode.java
- Clue.java (Will be overridden)
- Empty.java (Will be overridden)
- GameObject.java (Will be overridden)

- Hashmap.java
- HashNode.java
- HumanPlayer.java (Will be overridden)
- Main.java
- Player.java (Will be overridden)
- PlayerMap.java (Will be overridden)
- SemiRandomPlayer.java (Will be overridden)
- Treasure.java (Will be overridden)

# 9 Allowed libraries

- java.util.Scanner
- java.util.Random

# 10 Submission

You need to submit your source files on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. Your code must be able to be compiled with the Java 8 standard.

For this practical, you will have 5 upload opportunities and your best mark will be your final mark. Upload your archive to the appropriate slot on the FitchFork website well before the deadline. **No late submissions will be accepted!**