A picture containing diagram

Description automatically generated

Data Structures and Algorithms II

## Assignment Guidelines

Read the following instructions carefully before you start the assignment. If you do not understand any of them, ask your lecturer.

* The assignment coversheet should be the first sheet in your assignment. Moreover, the coversheet should be fully completed with all the necessary details.
* All text\code must be properly referenced. In the absence of proper referencing, the assignment will be regarded as plagiarised.
* Copying is strictly prohibited and will be penalized in line with the College’s disciplinary procedures.
* When the deadline specified by your lecturer is due, you shall hand all the required deliverables as explained in class.
* You are also required to submit your assignment to the relevant plagiarism detection service by the same deadline. If necessary, your lecturer will forward you details in order to submit your assignment to this service.
* The lecturer may hold a post-submission interview. Attendance to such interview is mandatory. Moreover, marks assigned to the criteria may be affected by the interview performance.
* **All work that has been carried out, must be written down and included within the assignment as evidence. No marks will be awarded for work that is not presented.**
* The deadline for this assignment is Monday 16th March 2025.

# Section 1 (17 marks)

Chaining Hash Table (KU3.1, 5 marks) (AA1.4, 7 marks)(KU4.1, 5 marks)

### Task 1 (KU3.1):

Describe how hashtables work (2 marks)

Hashtables are a type of data structure that store Key/Value pairs. Hashtables have an average speed for the Insert, Update, Remove and Get operations of O(1). Values in hashtables are accessed using the Key rather than an index number.

Compare the following types of collision resolution in hashtables:

* + Chaining (1 mark)
    - Uses a linked list (or other data structure) at each index to store multiple values that hash to the same position.
    - Allows for dynamic growth, but can degrade to O(n) if many elements collide.
  + Linear Probing (1 mark)
    - Uses open addressing.
    - Suffers from clustering, where consecutive occupied slots slow down lookups.
  + Robin Hood Hashing (1 mark)
    - A variation of open addressing that prioritizes fairness by ensuring elements with higher probe counts replace those with lower probe counts.
    - Balances the table and reduces variance in lookup times, improving worst-case performance.

### Task 2 (AA1.4):

Implement a hashtable that uses chaining to store key, value pairs. The hashtable must:

* Have an AddOrUpdate operation with the following method signature:

void AddOrUpdate(K key, V value)

The key and value pair are added to the hash table if the key is not present, however, if the key is present, the value corresponding to the key is updated instead. Use a maximum load factor of 80%. The maximum length of the linked list where new key, value pairs are added in the hashtable should be limited to 3. If the limit is exceeded, the hashtable needs to be rehashed with a larger array, however, make sure that the rehash does not cause a recursive rehash that keeps increasing the array size, even if the limit of the linked list is still exceeded. (3 marks)

* Have two operations, Get and GetAll with the following method signature:

V Get(K key)

List<K> GetAll()

The Get operation returns the value associated with the key, if the key is present in the hashtable, or throws an Exception otherwise.

The GetAll operation returns all the keys in the hashtable. (2 marks)

* Has a Remove operation with the following method signature:

V Remove(K key)

The remove operation, removes the key, value pair from the hashtable and returns the value associated with the key, if the key is present in the hashtable, or throws an Exception otherwise. (2 marks)

### Task 3 (KU4.1):

Illustrate the usage of your hashtable implementation by:

* Generating and storing 100 random key value pairs (no duplicate keys) (1 mark)
* Adding all the 100 key value pairs to the hash table (1 mark)
* Getting all the keys from the hashtable and confirming that exactly all of the original 100 keys are obtained back from the hashtable (1 mark)
* Getting the value associated with each of the keys obtained from the hashtable and confirming that the value associated with each key is the same as the original value that has been added (2 marks)

# Section 2 (25 marks)

Dijkstra’s Algorithm (SE3.4, 10 marks)(SE3.3, 10 marks) (KU1.3, 5 marks)

### Task 1 (SE3.4):

Implement a data structure for a Weighted Undirected Graph, using Adjacency Lists (4 marks)

The implementation must implement all the necessary operations that are required for the completion of this assignment (4 marks)

Use the Hash Table implemented in Section 1 to store and search the vertices and to look up the edges and/or adjacencies (2 marks)

### Task 2 (SE3.3):

Implement Dijkstra’s algorithm for an undirected graph using an ArrayBasedVector or List as the priority queue[[1]](#footnote-1). The code for this implementation must be kept in the final submission (5 marks).

Create another implementation of Dijkstra’s algorithm for an undirected graph using the Chaining Hash Table implemented in Section 1 as the priority queue (5 marks).

### Task 3 (KU1.3):

Create an instance of a Graph for the following weighted graph (presented in both dot notation, as well as the corresponding image) (2 marks):

graph G {

A -- B [label=" 8 "]

A -- C [label=" 12 "]

B -- C [label=" 13 "]

B -- D [label=" 25 "]

C -- D [label=" 14 "]

B -- E [label=" 9 "]

D -- E [label=" 20 "]

D -- F [label=" 8 "]

E -- F [label=" 19 "]

C -- G [label=" 21 "]

D -- G [label=" 12 "]

D -- H [label=" 12 "]

F -- H [label=" 11 "]

D -- I [label=" 16 "]

G -- I [label=" 11 "]

H -- I [label=" 9 "]

}

A diagram of a network

AI-generated content may be incorrect.

Use your implementation of Dijkstra’s algorithm to find the shortest path from A to H. Present the shortest path and the total weight of the shortest path in your document (3 marks).

A black background with white text

AI-generated content may be incorrect.

# Section 3 (14 marks)

Analysis of Dijkstra’s Algorithm (AA3.2, 7 marks)(AA2.4, 7 marks)

### Task 1 (AA3.2):

Investigate how different types of Priority Queues can affect the speed of Dijkstra’s algorithm (2 marks).

Include in your description how the following data structures affect the speed of Dijkstra’s algorithm:

* + ArrayBasedVector (2 marks)
  + Binary MinHeap (2 marks)
  + Brodal Heap (1 mark)

**Since priority queue is central to Dijkstra’s Algorithm, the insertion of a new vertex, extraction of the minimum vertex, and decreasing the key when a shorter path is found are all factors which will affect the speed of the Dijkstra’s algorithm.**

* **Array-Based Vector** 
  + **Inserting elements takes O(1) time, but finding the minimum element takes O(n) time (because the array isn’t sorted). Decreasing the key also takes O(n).**
  + **The complexity for Array-Based Vector is O(n²) which is slow for large graphs.**
* **Binary MinHeap**
  + **Insertion and decreasing key both take O(log n) time, and finding the minimum takes O(1) time.**
  + **The complexity for Binary MinHeap is O((n + m) log n) which is much faster than an array-based vector, especially for larger graphs.**
* **Brodal Heap**
  + **Insertion takes O(1), finding the minimum element is O(1), and decreasing key is O(log n).**
  + **The complexity for Brodal Heap is O(n log n + m) which is efficient but is drawn back by the complex implementation.**

### Taks 2 (AA2.4):

Analyse the asymptotic speed for your implementation of Dijkstra’s algorithm using an ArrayBasedVector for the Priority Queue (2.5 marks)

**In the case of ArrayBasedVector, both RemoveMin and DecreaseKey are O(n), making the overall complexity O(n²). This is because the RemoveMin operation must scan all vertices to find the minimum distance. The DecreaseKey operation also requires a linear scan of the array to locate the vertex to update.**

Analyse the asymptotic speed for your implementation of Dijkstra’s algorithm using the Hashtable for the PriorityQueue (2.5 marks)

**In ChainingHashTable, the RemoveMin operation is O(n) because it must scan all the entries to find the vertex with the minimum distance. However, DecreaseKey is O(1), making it faster than ArrayBasedVector. The overall complexity is O(n + m), which is significantly more efficient than O(n²), especially for scattered graphs.**

Compare the asymptotic speed of both your implementations against the theoretical speeds for different data structures for the priority queues (2 mark)

**Binary MinHeap provides O(log n) for both Extract-Min and Decrease-Key, which results in a time complexity of O((n + m) log n), making it the a very efficient priority queue for Dijkstra’s algorithm.**

**Fibonacci Heap improves on this with O(1) Decrease-Key, resulting in O(n log n + m), which is even faster than Binary MinHeap for graphs with many edges.**

**In conclusion, both these algorithms are theoretically much faster than ArrayBasedVector and ChainingHashTable, and are also better suited for large graphs.**

# Section 4: (12 marks)

Implement a sorting algorithm (AA2.3, 7 marks)(KU2.2, 5 marks)

### Task 1 (AA2.3):

Implement a Merge Sort with the following method signature (7 marks):

* + List<int> MergeSort(List<int> numbers)

The List in the parameter must *not* be modified by the MergeSort. The output of the MergeSort should be a sorted List.

### Taks 2 (KU2.2):

Create a List with 1000 random numbers

Use MergeSort to sort the list of numbers (2 marks)

Use a sorting algorithm from the programming language that you are using to sort the numbers (1 mark)

Compare the two sorted lists. Make sure that you sort the same original random numbers and that the output of the two algorithms are compared with each other. Make sure to not pass the output of one algorithm to the other algorithm for sorting. Use a test in your implementation to check that the results of both sorting algorithms match exactly (2 marks).

1. The priority queue is the data structure that will be used to store the visited vertices and find the vertex with the minimum distance to the source vertex. [↑](#footnote-ref-1)