**Searching with binary search, hash tables and search trees**

**Search trees**

A search tree is a data structure. The key for each node must be greater than any keys in subtrees on the left, and less than any keys in subtrees on the right.

**B-Trees**

Each node can have x amount of children

Databases

**Binary search trees**

the average time complexity of BST operations is Θ(h)

All nodes in the tree contain two children nodes and so on

Huffman code – compressing data

**Binary search**

An divide an conquer algorithm used on sorted sequences to retrieve the needed index with a time complexity of O(log n).

A picture containing text, keyboard

Description automatically generated

A screen shot of a calculator

Description automatically generated with low confidence

A close-up of a calculator

Description automatically generated with medium confidence

**Hash table**

h(k) = k mod m(size of array)

Purpose of hashtable is to be able to store data in a way, where you don’t have to traverse all the different data to receive the data you are looking for.

We find the correct index for our hashtable to a key buy creating a hash function for it (unique value – email, phonenumber,cpr etc.…). In each index of the hashtable, we can store key value pairs, which can correspond to an object in OOP or a row in a DB.

Let’s say we have a phone number 01452 8345654. This is our key value or our unique identifier. We decide that our hashing function will be as follows:



We then modulus 218 against the size of the array. We take the remainder of the calculation – this is now our index for where to place the phonenumber(key), and whatever values corresponds to that key. When someone wants to retrieve the values from a unique identifier (Our phonenumber), they will input the phonenumber into the hashing function, and get the exact index of where the value to this key is located. This is immensely faster then traversing through an ordinary array until your input key matches a key from the array.

**Collision in hash table**

A collision happens, when the hashing function generates the same index for two different keys. There are many ways to solve this - open address and closed address

Open address:

The hashing function will treverse until it finds a free space in the hash table, and give the insertet key that index. This will increase the time it takes to find an excisting item in the hashtable, because sometimes you will have to traverse the table a bit (or a lot worst case), to find the correct key-value pair.

Close Address:

All entries to the hashtable will be added to linkedlists. This way, when to keys generate the same index from the hashfunction, they will simply be placed in a linked list, that we will have to traverse through when were trying to find some data in the hashtable.

Open:

Table

Description automatically generated

Closed:

Diagram

Description automatically generated

Open closed

Graphical user interface, text, application, chat or text message

Description automatically generated

You can make your hashtable dynamic with the help of a load factor.

Graphical user interface, text, application

Description automatically generated