National and Kapodistrian
University of Athens

Inverted Search Engine

Trigaks Efstathios
Tsiamouras Dimitrios

Abstract

A typical search engine has a large and mostly static pool of text at its disposal. At regular intervals, different queries are made to the machine with the aim of finding and retrieving the texts related to them. A classic example of such an application is the Google search engine, in which users write queries about the information they are looking for, and the engine returns the texts it considers relevant.

An inverted search engine works in the same way, but assumes that the queries given to it are mostly static, and the available texts are those that come dynamically. A relevant example of such an application is Twitter, where texts here correspond to tweets. New tweets, of different content, are constantly coming to the application and must be retrieved when a relevant search is made.

Distinctions

This project has been crated for the purposes of our final project as students in the Department of Informatics, UOA. It has been graded with 10/10 “Excellent”.

GitHub Repository

You can inspect the code for this project in the following link:
https://github.com/stathis99/Inverted-Search-Engine

Introduction

In the context of this work, we have developed an application in C/C++ that accepts a multitude of texts that arrive with a continuous flow (stream of documents) and queries that must be answered, as long as they are related to some text.
A text is represented as a sequence of words separated by spaces, while a query is represented as a set of words. Both queries and texts are represented by a set of keywords. Whenever a new text arrives, the application should quickly find all the queries related to it. For a query to be considered to answer a document, all the words in the query must match to a subset of the words in the incoming document. The process by which we examine whether one word matches another is called “keyword matching” and can have multiple variations:

1) Exact Matching: Both of the words are absolutely identical to each other.
2) Approximate Matching: There is a tolerance in difference in some points with deviation in distance restrictions:
    a) Edit Distance
    b) Hamming Distance

Our goal was to reduce the response time of the system as much as possible.

Input

The system accepts an input of texts that arrive with a continuous flow. The text is divided in 2 interchangeable segments. The first segment is always a sum of queries. The queries in the file have the following format:

```
s 1 0 0 3 edit gaither anderson
s 2 1 2 2 diocese pgdma
s 3 1 2 2 diocese pgdma
```

- Every row is a different query
- column1: s = query
- column2: id
- column3: type of query
- column 4: threshold
- column 5: sum of words for this query

Type of query

There are 3 different types of queries. Every type corresponds to different "keyward matching" that is examined in the Introduction.

- 0: Exact Match
- 1: Edit Distance
- 2: Hamming Distance

For example, if a query has a type of "Exact Matching" it means that all the words for this query must match identically to a subset of words from the document we examine.
For each type of query, there is also a different data structure where the words of the queries will be stored.

When the system has read all the queries of the first segment and has stored all the words in the structures it can accept Documnts. Every Document has the following format:

```
m 1 1428 http dbpedia resource list women national
current players bold also list national basketball
aguilar matee ajavon marcie alberts markita aldridg
anderson keisha anderson mery andrade yvette angel
```

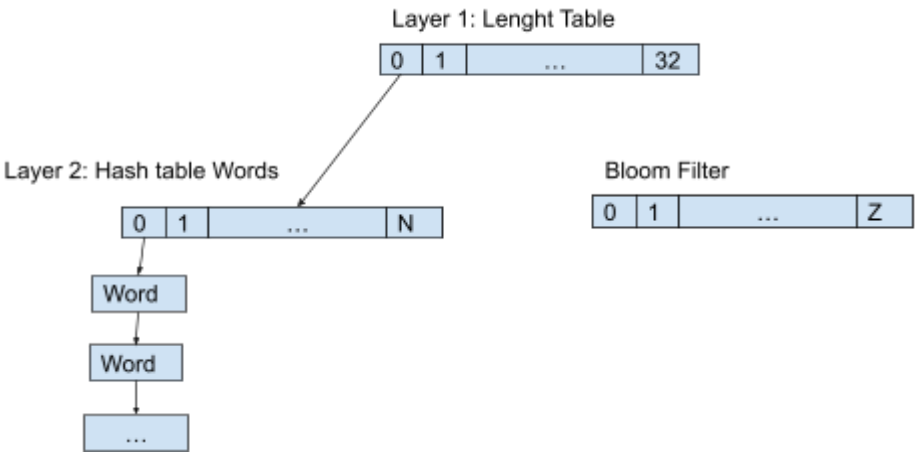
- m: Document
- column1: id
- column2: sum of words
- words of documnt

Data Structures

All the words from the queries have to be stored in our structures. We have created 3 different Data structures, one for every type of query.

1) Exact Match Data Structure

In this structure we have 2 layers. In the first layer, there is a Lenght Table of 32 pointers (32 is the max word length). Every pointer points to a Layer 2 Hash Table. This Hash Table has N pointers which point to lists of words.



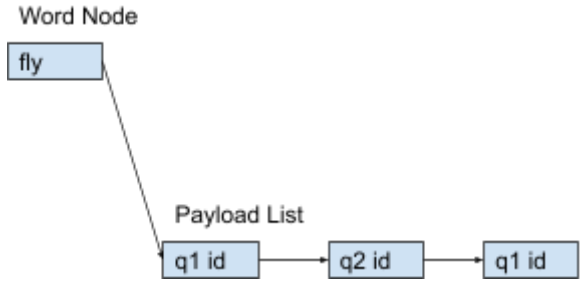
Example of word insertion:

The system reads a query
If (the query is a type of Exact Match) then:
For every word in the query:
Insert word in the Data Structure:
Hash word twice and refresh the bloom filter
Depending on the word lenght find Layer 2 Hash table

Depending on the hash value, find the list of words
Insert word in the list

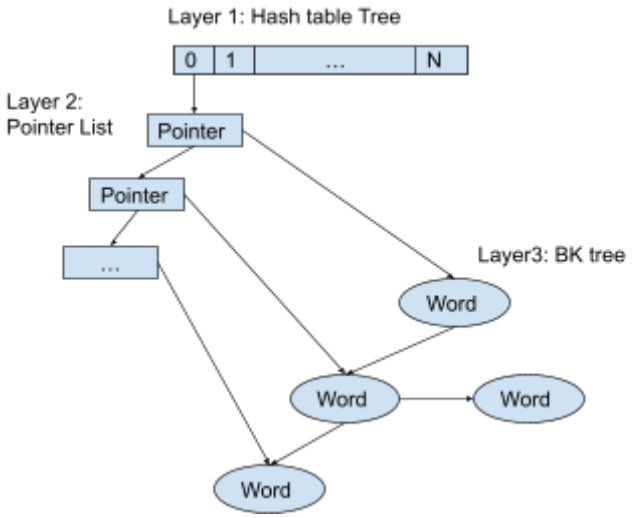
Insert word in the list

Their is an issue regarding duplicate words. If a word f.e “fly” is in q1,q3 and q7 then we dont save the word 3 times. We save it once as a text and then we use a list called payload to save the query id of the other queries



2) Edit Distance Data Structure

In this structure we have 3 layers. The first layer is a Hash Table with N backets. (N is defined during compiling). Every bucket points to a second Layer Pointer List. This list contains pointers that point to Nodes of a BK tree. The Layer 3 BK tree has multiple nodes, and each one contains a different word.



Example of word insertion:

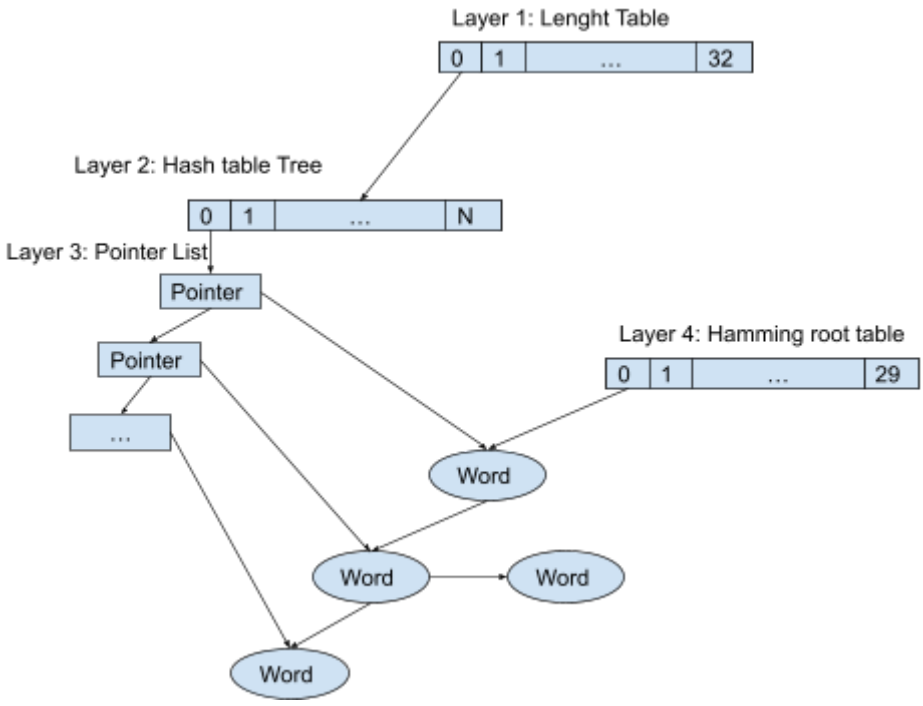
The system reads a query
If (the query is a type of Edit Distance) then:
For every word in the query:
Insert word in the Data Structure:
Hash word
Depending on the Hash value, find Hash table backet
Follow the bucket pointer and iterate through the list to find if word exists.
If word exists then update payload
Else add word to BK tree and add a new pointer to the list

3) Hamming Distance Data Structure

Δομή για hamming distance

In this structure we have 4 layers. In the first layer their is a Lenght Table of 32 pointers (32 is the max word lenght). Every pointer points to a Layer 2 Hash Table. This Hash Table has N pointers which point to lists of pointers. Every pointer points to a BK tree node. The difference with Edit distande structure is that we dont have one BK tree for all words but different trees for every lenghts. So in

some cases we might have 33 Bk trees. Layer 4 Hamming Root table is a Table that stores every BK tree Root.



Example of word insertion:

The system reads a query
If (the query is a type of Hamming Distance) then:
For every word in the query:
Insert word in the Data Structure:
Depending on the lenght of the word find hash table.
Hash word
Depending on the Hash value, find Hash table backet
Follow the backet pointer and iterate through the list to find if word exists.
If word exists then update payload
Else add word to BK tree and add a new pointer to the list

Bk Tree

A BK-tree is a tree data structure specialized to index data in a metric space. A metric space is essentially a set of objects which we equip with a distance function d(a, b) for every pair of elements (a, b). This distance function must satisfy a set of axioms in order to ensure it's well-behaved. The exact reason why this is required will be explained in the "Search" paragraph below.

Construction of the tree

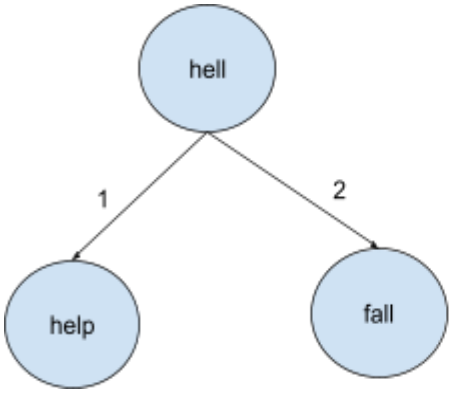
BK-tree is defined in the following way. An arbitrary element a is selected as root. Root may have zero or more sub-trees. The k-th sub-tree is recursively built of all elements b such that $d(a,b) = k$

To see how to construct a BK-tree, let's use a real scenario. We have a dictionary of words and we want to find those that are most similar to a given query word. To gauge how similar two words are, we are going to use the Levenstein Distance.

The Levenshtein distance between two words is derived by minimum number of edits of a single character (edit can be addition,deletion or substitution), required to transform one word into another. For example the distance between the words 'hell' and 'small' is 3, because to go from 'hell' to 'small', we can add s to the beginning and then replace h with m and e with a.

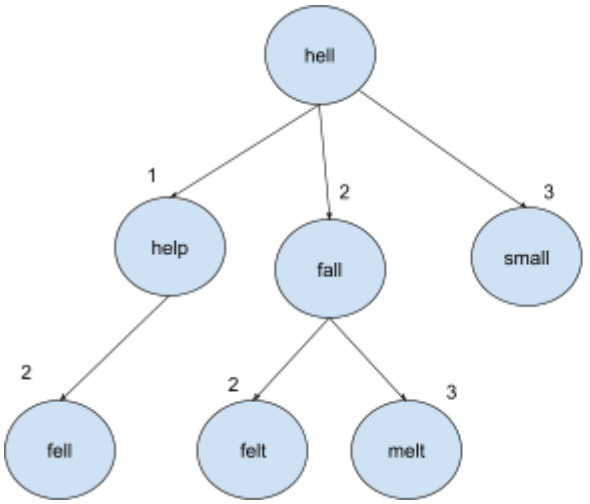
For example Let the set of words ['hell', 'help', 'fall', 'felt', 'fell', 'small', 'melt']

To build the tree, we choose arbitrarily as root a word, even 'hell' and then we add the rest by calculating their distance from the root. Then for the first 3 words, the tree will have the following form:



.

The word 'help' is distance 1 away from 'hell' (replacing p with l), while the word 'fall' is 2 away (replacing h and e with f and a) . Then we add the word 'felt'. This word is 2 away from 'hell', so it will be added to the right subtree as a child of 'fall', using an edge containing it distance between 'fall' and 'felt', i.e. 2. After adding all the words, our tree will have the following form:



Search in BK tree

The original problem was to find all words that are "close" to a given query word. Let n be the maximum allowed distance (also called the radius). The algorithm has the following steps:

1. Create a list of candidate words and add the root to that list.
2. Take a candidate word from the list, calculate its distance d from its word question and compare the distance to the radius. If it is smaller, add it to the list of found words.
3. Selection criterion: add all children of the specified word to the list of candidate words nodes having a distance from the parent in the interval $[d - n, d + n]$

Suppose we want to find all words that are no more than 2 from the keyword 'henn'. The only candidate word in the list is the root word, ie 'hell'. We start by calculating the distance between 'hell' and 'henn', i.e. $d = 2$. Since the distance is less than the radius, we add the word 'hell' to the list of found words

We then expand the list of candidate words with the words of nodes/children whose distance from the root is between $[d - n, d + n] = [0, 4]$. All the subtrees satisfy this constraint, so all words in the first sublevel are added to the list of candidate words ['help', 'fall', 'small']. The distances of these words are 2, 4 and 5 respectively from the keyword. Therefore only the word 'help' is added to the index list of words. The other 2 words are not added to the results. Their subtrees are examined normally.

Then we add in the list of candidate words the only subtree child of ‘help’, that is the word 'fell', and the 2 children of 'fall' are examined, namely the words 'felt' and 'melt'. These 2 kids have a distance of 3 from the keyword and should be in the interval [4-2, 4+2]=[2,6]. Therefore they are added to the list of candidate words. The word 'small' has no children

Finally, all candidate words in the list ['fell', 'felt', 'melt'] are examined in turn, none of which does not satisfy the minimum distance and none of which has children.

At this point the search is complete and the list of found words consists of words ['hell', 'help'], both words distance 2 from keyword 'henn'.

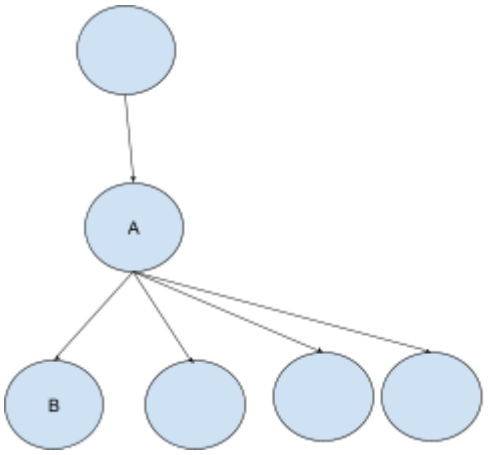
Analysis

It is interesting to see why we can exclude all subtrees that do not cover the selection criterion 3. We have seen that the distance metric function d must satisfy a set axioms in order to be able to exploit the structure of the metric space. For all elements a, b, c of the metric space the following must hold:

1. non-negativity: $d(a, b) \geq 0$.
2. $d(a, b) = 0$ means that $a = b$ (and vice versa)
3. symmetry: $d(a, b) = d(b, a)$
4. triangular inequality: $d(a, b) \leq d(a, c) + d(c, b)$

As it turns out, the Levenshtein distance satisfies all 4 postulates, so it can used as a metric function.

Let x be the query key, and that we evaluate the child B of any node A, for which we have determined to be a distance $D = d(x, A)$ from the query key. The general structure of the tree can be depicted in the figure below:



Since the Levenshtein distance we use is a metric function, it holds that:

$d(A, B) \leq d(x, A) + d(x, B)$

Therefore: $d(x, B) \geq d(A, B) - d(A, x) = x - \mathrm{D}$

Since we are only interested in nodes that are at most n from its key question x, we should $d(x, B) \leq n$. This means that:

$x - D \leq n$ και $D - x \leq \mathrm{n}$

which is equivalent to: $D - n \leq x \leq D + \mathrm{n}$

Therefore, if the distance d is the metric function, we can safely discard them nodes that do not satisfy the above criteria. Finally, each child of B will be at a distance x from A, and thus the total subtree can be ignored if B alone does not satisfies the criterion.

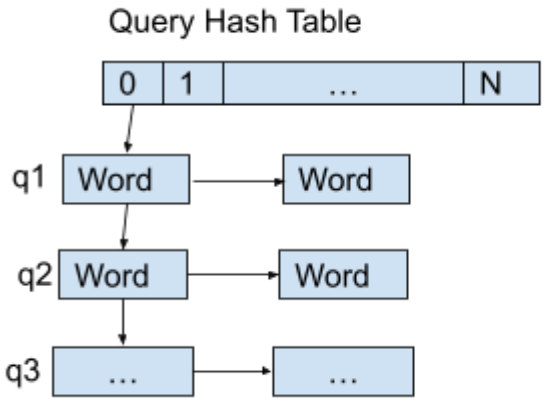
Match Document

When a document is read from the file, the system has to search in the structures which words match every word in the word. Once it finds the matching words it must find the queries that all of its words were found in the search. Then the system returns the queries that “match” the incoming document.

In order to know for each query which words it contains, we use an a structure(Query Hash Table) which stores all the queries with their words.
Finally Deduplication is done for the words of the Documents so as to reduce searches.

Query Hashtable Structure

A hash table that points to a list of queries. Each element of the list points to a list where the words are stored.



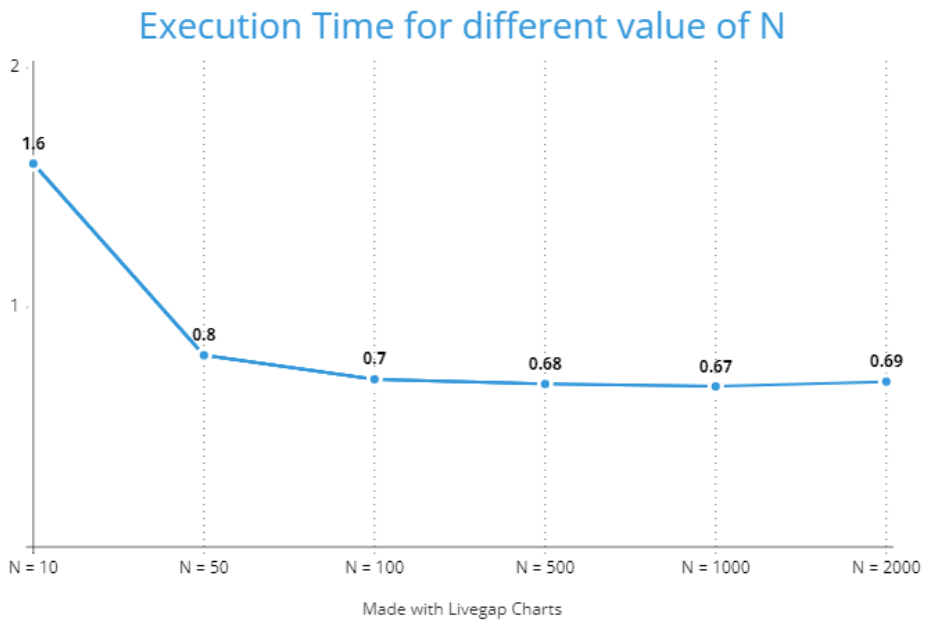
Matching Document Example:

For each word we first check if we already have cheched it
    If we have checked it, we skip it
    Else we check the 3 structures with which words it "matches"
        The matching words for each structure are put into a list.
        Using these lists and the Q hash Table structure, we find which queries match the document
return the queries list
.

In order to perform the search in the “Hamming” and “Edit” structures, an iterative search is performed in the bk trees from 1 to 3. This number refers to the threshold that we give as a tolerance for the difference between the word we are looking for and the words in the structures.

Below you can see the execution time for diffenrent N values. N reffers to the

Hash table size.

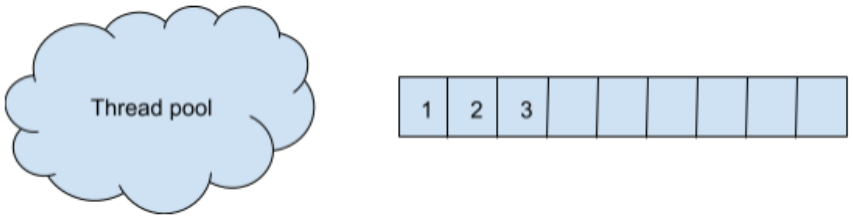


We can conclude that the optimal value of N is 1000 with an overall execution time of 0,67 sec.

Multithreading and Job Scheduler

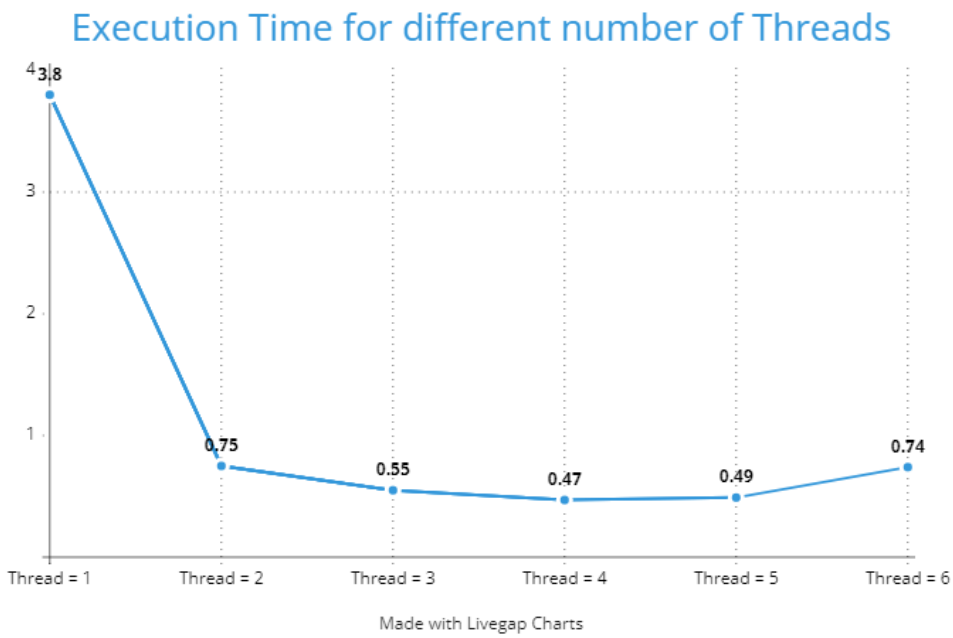
In order to reduce the overall execution time even more we have introduced parallelism. Parallelism has been implemented through multithreading.

More specifically, we have created a Job Scheduler. The Job Scheduler consists of a job queue, and a set of threads called workers. Workers wait for a job to be entered into the queue, and one by one they take over to execute the jobs.



In our system we use the Job scheduler to parallelize the Match Document. Every time a Documet is read by the program instead of searching for its words in the structures from the main thread, a job is created that goes into the queue, and thread workers take over to do the searching.

As a result, searches on the structures are done in parallel for many documents and the time is significantly reduced. As you can see in the following graph depending on the number of worker-threads we have defined, we get different execution time.



We can see from this graph that the optimal thread number is 4 with an execution time of 0,47 sec.

The files we tested in our program have alternately query insertions, document reading and query deletions in various combinations. A problem we encountered at this point was that after a series of insertion and deletion queries while reading a batch of documents, the main thread must stop the insertion and deletion of queries until the documents have completed their searches in the structures.

Score Results

The optimal Score we have achieved using small_test.txt is 0,47 sec.

Evaluation Machine

Programming Language	C/C++
Envirnment	Mac OS X
Compiler	GCC > 5.5
Test file	small_text.txt
Processor	Apple M1

References

https://signal-to-noise.xyz/post/bk-tree/

https://dl.acm.org/doi/10.1145/362003.362025

https://sigmod.kaust.edu.sa/index.html