INF-2700 DATABASE SYSTEMS

DBMS

ALEXANDER EINSHØJ

18.10.16

Introduction

This paper consists of two parts. The first part allows for multiple operators to be used during a search, while the second part describes the implementation of binary search as an alternative searching algorithm.

Technical Background

Binary search

The binary search algorithm is a simple but effective algorithm for finding a given element in a large dataset, given by its average case performance of O(log n). The algorithm presumes a sorted array of data in order to work. In other words, if the first element is larger than the last element, it terminates. The search begins in the middle of the dataset, extracts a value, and compares it to the value which is being searched for. If the value found is larger than the target value, the last half of the dataset is now considered irrelevant. This procedure continues until the algorithm is left with one value, which should be the target value.

Design

The binary search in this implementation consists of both binary and linear search. Given that the first page exists, and there is more than one page, the binary search will occur. At all times, the binary search needs to keep track of where it currently is in the dataset, compared to the relative beginning and end. The relative beginning and end is called that because they both have the possibility of changing throughout the search. As explained earlier, if the value that resides in the middle of the dataset is larger than the value being searched for, the new relative beginning will be at that value. This is because we now know the value can't be any lower than the value we found, and because we assume the dataset is sorted. Without the presumption that the set is sorted, binary search won't work at all. In that case you would have a tree with randomly placed values, and the algorithm would just be jumping back and forth in the dataset without getting anywhere.

The search begins at the middle page of the dataset, at the first element. While going back and forth through the pages, the value searched for will only be compared to the first element on any given page. This implies that we need a reference to another value at another page. If the value we are looking for resides somewhere after the first element on a page(which it most likely will), then a mechanism needs to be implemented in order to not go back and forth forever between those pages. If a value in the beginning of a page is smaller than the value we are looking for, but at the same time bigger than a value at another page, we know that this is the correct page to look for the value.

There are a couple of special cases that needs to be considered. The most basic being when there is either no or one page only. An other is when the current page is the last page. Because we only check for the first value in each page, we will in most cases receive a value that is lower than the one we are looking for. At the same time, there are no more pages to look through, so the algorithm has to know that this is the last page, and that if the element does exist at all, it resides on this page.

Once the search is narrowed significantly by the binary search in terms of pages left, it's relatively safe to begin iterating through the pages until the value is found. Once an element in a page is read, the entire page is read. Therefore, the time spent sifting through a page is close to nonexistent compared to reading from disk. It is also assumed from the definition of a BST that duplicates are to be ignored. Given this, the algorithm will also ignore duplicates and show the first value which fulfills the criteria of being equal to the value searched for.

Implementation

The program is implemented in C. The database creation program is implemented in Python.

Discussion

The current implementation has a terrible worst case scenario which I have not been able to handle properly. Originally, the thought was to do the entire search binary down to the correct page. This proved to be difficult, which is the reason the implementation is as it is. The worst case scenario will do a linear search throughout 25% of the entire dataset. The larger the dataset, the worse the performance.
That version of the dataset, while still being better than linear search in most cases, does not fulfill the purpose of a binary search algorithm. The solution in thought, which is not implemented due to the amount of time left, is to begin the linear search when the current page is close to the page where the value resides. By doing this, the amount of disk reads are limited by a lot. Instead of sifting through potentially 25% of the original dataset, one would perhaps not consider more than 10 pages. The bigger the dataset, the more effective this would be.

In a real life situation, one would not utilize linear search to find data in a database, as the values could not be presumed sorted, as they are in this case.

Conclusion

Both tasks are implemented within the requirements, and the only thing not included is performance testing. This is left out due to lack of time, but as stated above, I'm aware of the

worst case scenario. It’s self-explanatory that even with this implementation, depending of the size of the dataset, the binary search will be quicker than linear search in all cases where the right part of the dataset is searched for. If the dataset is small, this won’t be the case, but that is not the goal of binary search either.