k-d tree

Gerard Martin Teixidor

May 11, 2021

Description

A k-d tree [1] is a data structure used to store points in a k-dimensional space. Its implementation relies on a binary tree in which every non-leaf node represents a space partition. Each level of the tree is associated with one of the k dimensions which, at each node, splits the space into two subspaces. These two subspaces are represented by the left and right subtrees.

Implementation

This k-d tree has been implemented using generics. This allows a user defined type to represent the elements (points) which are going to be stored.

It is important to note that this implementation of a k-d tree does not grantee that the tree is always balanced. A balancing algorithm could be implemented with a $O(kn\ log\ n)$ time complexity by using the median [2].

The data structure code has been programed using the *C++17* and the *GNU Compiler Collection*.

Operations

List of the allowed operations with a brief description and their time complexity:

empty Test whether the collection is empty.

Time complexity: $O(1)$

size Return the collection size.

Time complexity: $O(1)$

insert Insert an element.

Time complexity: $O(log\ n)$

erase Erase an element.

Time complexity: $O(log\ n)$

clear Erase all elements.

Time complexity: $O(1)$

contains Test whether the collection contains an element.

Time complexity: $O(log\ n)$

min Return the element with the minimum dimension.

Time complexity: $O(n)$

range Return the elements that are inside a given range.

Time complexity: $O(log\ n + m)$ where m is the number of reported points.

nearest_neighbor Return the nearest neighbor element of a given element.

Time complexity: $O(log\ n)$

operator<< Inserts the tree string representation into an output stream.

Experiments

To test the time complexity of the data structure two experiments have been performed. Note that, in both tests, the results have variability. This is due unbalanced nature of the data structure. Another explanation is the memory management by the operative system since this implementation does not preallocate memory when the collection is created. In both experiments the outliers have been removed by using a handpicked quartile.

The fist experiment (Figure: 1) shows that, in effect, the time complexity of the *insertion* operation is $O(log\ n)$. As in the previous experiment, this second experiment (Figure: 2) shows that the *contains* (search) operation also follows a logarithmic order.

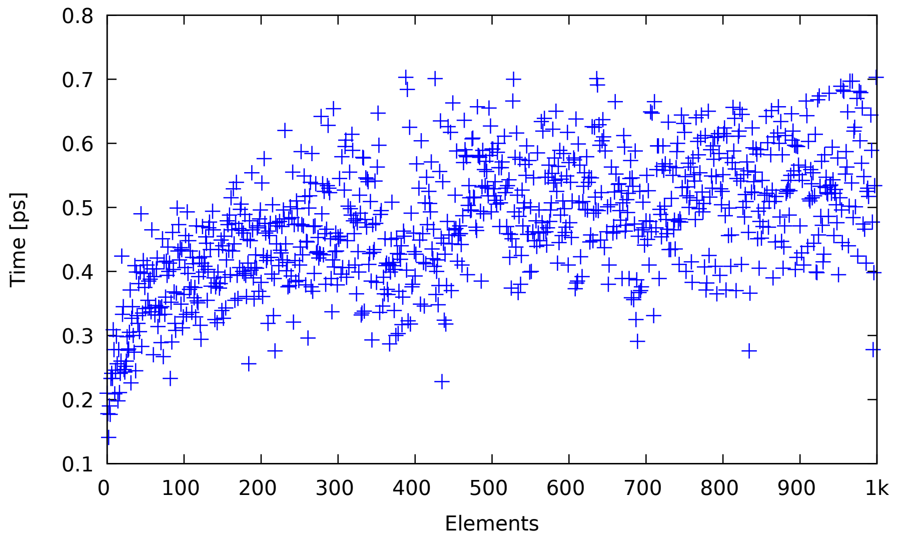

Figure 1: Plot showing the *insert* operation $O(log\ n)$ time complexity.

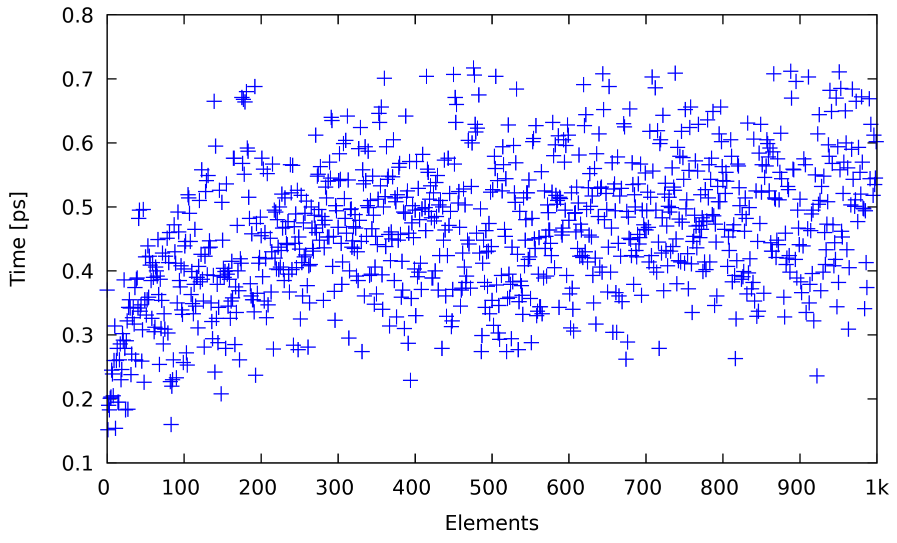

Figure 2: Plot showing the *contains* operation $O(log\ n)$ time complexity.

References

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[2] Russell A. Brown. Building a balanced k-d tree in $o(kn \log n)$ time. *Journal of Computer Graphics Techniques (JCGT)*, 4(1):50–68, March 2015.