K-ary Search Tree Revisited:Improving Construction and Intersection Efficiency

Xingshen Song, Jinsheng Deng, Fengcai Qiao, Kun Jiang

Frontiers of Computer Science, DOI: 10.1007/s11704-022-0493-2

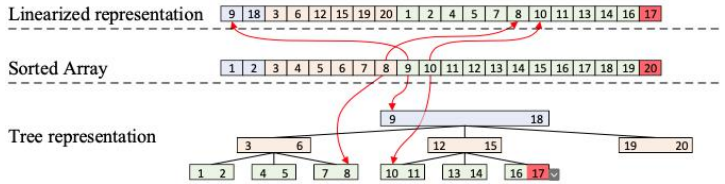
Drawbacks and Solutions

There are two drawbacks remains in *k*-ary search tree, which impair its efficiency and make it uncompetitive with traditional data structures.

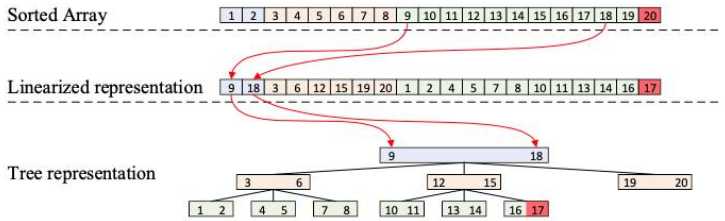
- **Construction efficiency**: the mapping function used for converting sorted array into tree structure involves too much arithmetic
- **Query processing**: its application in query processing is confined within membership test

Solution to the construction problem:

Instead of sequentially calculating the mapped position in the tree for the index in sorted array, we perform a reverse level-wise node gathering from the array. Our method is able to omit unnecessary arithmetic with the help of the spatial adjacency of each node.



(a) sequentially map contiguous entries 8, 9, 10 from sorted array to linearized representation

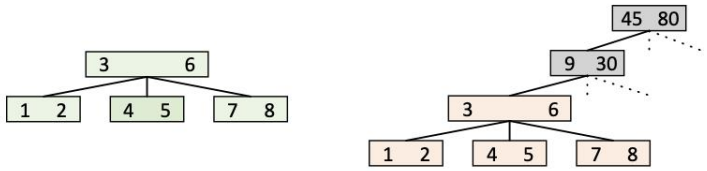


(b) Gather entries 9, 18 for root node

Solution to the processing problem:

we introduce two early termination techniques, ***skip*** and ***narrow***, collaborating with particular traversal methods to improve its intersection efficiency.

(a) The way *skip* works. After searching entry 3 and 13 of the left tree, we would know there are no entries smaller than 5 or larger than 12. The entries shaded can be droped without validation.



(b) The way *narrow* works. Visually, all the potential matches reside in the bottom-left subtree of the right tree, nodes in grey are trivial when intersecting with the left tree.

Experiments

- We evaluated our methods against their counterparts on both GOV2 and synthetic data.
- In our experiment, available values for *k* are 3, 5 and 17.
- Our implementation is compiled with G++ 5.4.0 with -O3 optimizations on an Intel i7-9700 processor running CentOS 7.

Construction Efficiency:

We refer to our method as **recursive** and the original method as **original**. It is conspicuous that **recursive** outperforms **original** by an order of magnitude in most of the cases.

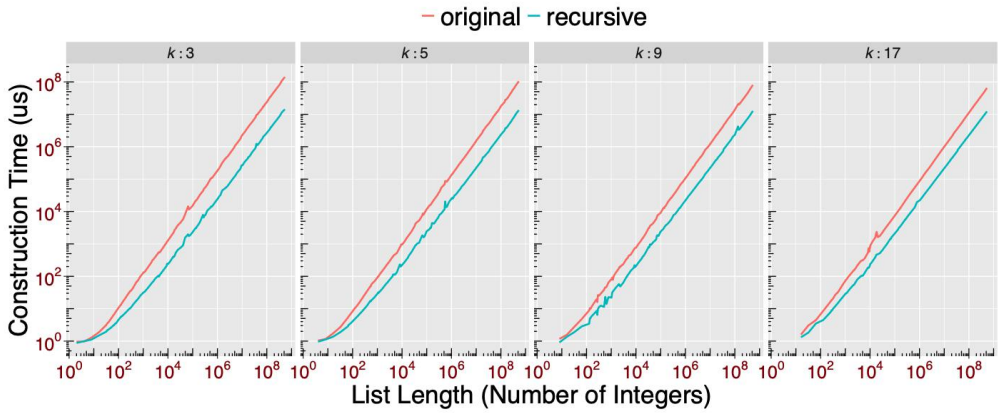


Fig. 2: Building time of a *k*-ary search tree for different sizes and *k* values via two construction algorithms. Note the axes are plotted in log scale for better visualization.

Intersection Efficiency:

We name our methods as **sequential_opt**, **sorted_opt** and **hierarchical_opt**, their performances are evaluated against another state-of-the-art SIMD-based method: **svs_opt**.

We set k=1 to denote no SIMD parallel involved. As shown in the result, our methods work better when $k \geq 9$ and the size ratios>2^{10}

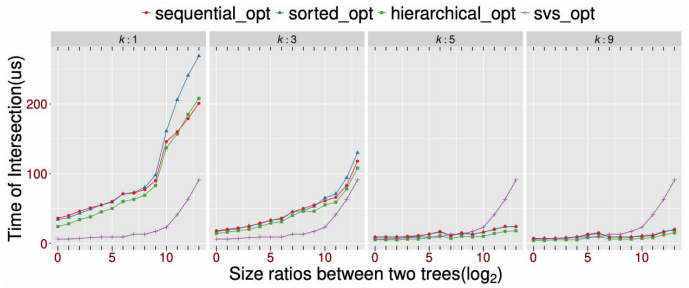


Fig. 3: Intersection time on synthetic data for different *k* and SIMD registers, the selectivity is fixed to 60%.