Block Binary Search Tree

Shengdun Wang
Stevens Institute of Technology, Hoboken, NJ
swang115@stevens.edu

Keywords: Block Binary Search Tree, B+ tree, Red-Black tree, Binary Search.

Abstract. This paper illustrates a novel data structure created by me called Block Binary Search Tree (BBST). BBST is a data structure which extends Binary Search Tree by replacing the key-value pair domain with an array of key-value pairs in each node of Binary Search Tree. It greatly reduces the space usage cost compared to B+ tree, Binary Search Trees and all kinds of different Self-Balancing Binary Search Trees such as AVL, RBT, Splay.

1. Introduction

std::vector (dynamic sized array‡ in C++§) is the "default" container in C++ language. C++ Core Guideline [9] claims: Even when other containers seem more suited, such a map for O(log N) lookup performance or a list for efficient insertion in the middle, a vector will usually still perform better for containers up to a few KB in size.

If vector would perform better for in a few KB in size than map like the Guideline says since it utilizes the cache far better than a Binary Search Tree, the question is that whether it is possible to create a new tree structure which can actually be cache-friendly.

The paper Columnstore and B+ tree – Are Hybrid Physical Designs Important? [3] proves B+-tree is over-complicated. It distinguishes internal and leaf nodes in order to distinguish key and value.

1.1 Skip list

The array and the linked list are contradict with each other. Their properties are always on the contradictory.

1.1.1 The array is contiguous while linked list is randomly-located in the memory;
1.1.2 The array can be random-accessed while linked list can only be linear-accessed in Θ(1) time complexity;
1.1.3 The array can only insert element in the back (which is vector) in Θ(1), while list can insert element in any place in Θ(1);
1.1.4 The array utilizes the cache greatly, while linked-list causes a lot of cache misses; The array has zero space-overhead on pointers, while uses a lot of pointers. The idea behind skip list is trying to improve linked list by adding some pointers to skip data. When I first found this skip list data structure, the question to me is that whether it is possible to do the same thing to the array. If that was successful, then it solves this cache-unfriendly issue, thanks to the cache-friendly nature of The array and the Θ(ln(N)) insert, delete and look-up nature of tree structures.

However, I failed to create this "skipped-array" data structure so I decided to try to improve other tree structures. It is yet a problem whether such data structure can be constructed.

‡ The array in this paper represents C-style or std: vector-like contiguous container.
§ This paper will use C++ as demo language to describe the data structure since C++ is the most powerful language to deal with abstractions and low level bits.[7] Other languages like Java cannot describe this easily.
1.2 Binary Search Tree

The Binary Search Tree (BST) is a basic data structures being widely used in nearly all modern basic computer systems, including operating systems, database management systems, gaming, cloud computing etc. Though it is a basic data structure which every computer scientist would learn and understand, it was created in 1960 before CPU providing Caches, so naturally it is not designed to be cache-friendly. Nowadays, all modern computers, even cellphones, have multiple levels of cache in the CPU chips which mean Binary Search Tree in the real world would be extremely cache-unfriendly in modern computers.

1.3 Self-Balancing Binary Search Trees

Different kinds of self-balancing Binary Search Trees[12], such as Red-Black, AVL and Splay are still Binary Search Trees. Consequently, they are cache-unfriendly as well.[2] Though they share the same cache-unfriendly issue as normal BSTs, they provide potential solutions for creating self-balancing Block Binary Search Tree. This paper will discuss whether Block Binary Search Tree can be self-balancing just like Binary Search Tree. If it can be self-balancing, due to the time restriction of this paper, it would only be a future topic to discuss which self-balancing binary search tree could be used or is best for creating a self-balancing block search tree.

1.4 B+ Tree

B+ Tree[2] is widely used in the query systems involving a lot of disk I/O operations. The advantage of B+ tree over Binary Search Trees is that it is much more cache-friendly.[8] However, B+-tree wastes a lot memories on pointers. Block Binary Search Tree which this paper introduces would do greatly than B+-tree and provides nearly all functionality B+-tree could provide, including in non-volatile memory.[1] [4]

1.5 Hash Table

In this paper, hash table would be baseline in order to compare the benchmark of Block Binary Search Tree and other tree structures.

Mostly, it is preferred to use hash table rather than any tree structure for fast queries. Hash table is fast for its amortized Ω(1) time-complexity insertion, deletions and queries. It is cache-friendly as well. However, hash table has its restrictions:

1.5.1 The elements are not ordered which can not be used for queries of range of data.
1.5.2 Elements must be hashable.
1.5.3 Hash function can potentially be exploited which would create security issues.

1.6 B+ Tree

Using a sorted vector and binary search could help us determine the lower bound of Θ (ln(n)) families of trees. Query and iteration operations of any tree structures can never be faster than directly applying binary search algorithm to a sorted-vector.

2. Design and implementation of BBST

BBST is a binary tree with each tree node with a fixed-size array. It has following properties:

• The array in all nodes is sorted.
• The array in all non-leaf nodes is full.
• The first element of the array in a node is not smaller than the maximum element in the node’s left subtree.
• The last element of the array in a node is not larger than the minimum element in the node’s right subtree.
For example, a 3-node-array tree node is defined as:

```cpp
template<typename T>
struct bbst_node {
    Bbst_node *left, *right;
    size_t n;
    array<T,3> values;
};
```

A sample 3-nodes BBST is look like in Figure 1

![Sample 3-nodes BBST](image)

**2.1 Search**

Searching in a M-node BBST is a mix of BST searching with binary search. The time complexity is $\Theta(\ln(n))$, same as BST. M is a constant. The calculation of time complexity:

$$\Theta((\ln(M))*(\ln(n/M)))=\Theta(\ln(n)-\ln(M))=\Theta(\ln(n))$$

**2.2 Emplace (Insert)**

Emplace is basically insert but it contains the semantics of construct.

It is a mix of insert element into the middle of array and insert node into the binary search tree. Although insert an element into middle of an array has a $\Theta(n)$ time complexity, the actual cost of inserting into the middle array is still $\Theta(1)$ in BBST since our array has a fixed size. $\Theta(100000) = \Theta(1)$.

An example of insert 4 elements (1,20,24,22) into the Figure 1 BBST

![Insert 1 into the BBST](image)

![Insert 20 into the BBST](image)

In the Figure 3, inserting 20 pushes 17 to the most-right element of the left subtree of the root and moving 19 to the begin of the array.
2.3 Insert

Fig 4. Insert 24 into the BBST.

Inserting 22 into the BBST creates imbalance of the binary search tree, so BBST actually needs be implement as Red-Black tree or AVL tree or Splay tree just like ordinary BST to rotate the tree in order to solve the self-balancing issue of BBST.

2.3 Erase

Erasure is very similar to insert. It would be easy to remove elements in the leaf nodes.

Fig 5. Insert 22 into the BBST.

For non-leaf n it would be a bit harder, it will always remove element in the right-most node in the left sub-tree. Here are two cases which required to be discussed here.

2.3.1 Right-most node in the left sub-tree has no left sub-tree

This case is simple. Just erase the element from the node and append the first node from the array of Right-most node in the left sub-tree to the front of the array of the node.

For example, remove 23 from Figure 5 would be

Fig 6. Erase 23 from the BBST in Figure 5.

2.3.2 Right-most node in the left sub-tree has left sub-tree

For BBST like in Figure 7, suppose we try to remove the key 22

Fig 7. Aa sample BBST.
It cannot just move the last element of the right-most node in the left sub-tree to the front of the array in the root like in case 1 does since the definition of BBST requires all non-leaf nodes to be full.

In this case, the correct way is to do rotation in BBST. Rotate the right-most node in the left sub-tree to the right node of its own right-most node in its left sub-tree.

Then fill the space in the non-leaf node to satisfy the definition of BBST.

Now it becomes the case 1 where Right-most node in the left sub-tree has no left sub-tree.

The final result after erasure 22 is the BBST in Figure 11.
2.4 BBST Traverse

The traversal of BBST is also similar to the traversal in BST. The paper will only show the procedure of in-order traversal.

2.5 Destroying BBST

After finishing using a BBST, we need destroy it and get our space back. Destroying a BBST is to do a tree-traversal, destroy all elements in the array first and then deallocate the node space.

However, for destroying, it would be better not to access the same memory block again since this would reduce the potential cache-missing probability, although the compiler might help us optimize it.

3. Benchmark

In my PC, the BBST is fastest for setting 1500-2000 bytes per node in general. The block size should be set differently for different computers, and the block size could even be different for different layers of BBST nodes. (for example, set 1500-2000 bytes per node in memory and 10000 bytes per node for disk.)

<table>
<thead>
<tr>
<th>Platform</th>
<th>AMD Ryzen 2600</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>16GB DDR4 3000MHZ</td>
</tr>
<tr>
<td>Operating System</td>
<td>Windows 10 17763.55 x64 Pro</td>
</tr>
<tr>
<td>Compiler</td>
<td>MCF-MinGW-g++-8.2.1 20181011 x64</td>
</tr>
<tr>
<td>C++ Standard</td>
<td>C++17</td>
</tr>
<tr>
<td>Optimization Level</td>
<td>-O2</td>
</tr>
</tbody>
</table>

template<typename T>
struct bbst_node
{
    bbst_node * left=nullptr ,* right=nullptr ;
    std::size_t n=0;
    std::array<T, sizeof (T)<1500?1500/sizeof (T):1> values ;
};

<table>
<thead>
<tr>
<th>Data Structures</th>
<th>Type std::size_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500bytes BBST</td>
<td>187 elements per node BBST</td>
</tr>
<tr>
<td>BST</td>
<td>No self-balancing</td>
</tr>
<tr>
<td>std::multiset</td>
<td>C++ Standard Red Black Tree</td>
</tr>
<tr>
<td>std::unordered_multiset</td>
<td>C++ Standard Hash Table</td>
</tr>
<tr>
<td>std::vector+Binary Search</td>
<td>Dynamic sized array</td>
</tr>
<tr>
<td>stx::btree_multiset</td>
<td>B+ Tree Implementation GitHub</td>
</tr>
</tbody>
</table>
4. Restrictions of BBST

BBST cannot always replace BST. Here are comparisons of among BBST, BST and other data structures.

4.1 Iterator Invalidation

Emplacing or erasing elements in BBST would invalidate all pointers, references and iterators of BBST while emplacing or erasing elements in BST only invalidates the pointers, references and iterators of elements got erased themselves. [5]

4.2 Impossible to have strong exception safety guarantee unless moving the elements does not throw.

It is not possible to revert the operations when moving the elements from the array in node A to the array in node B fails for BBST. [6]

However, it is possible to ensure basic guarantee[11] for all types.

4.3 Implementation is intricate

Self-balancing Binary Search Tree like AVL or RB-tree is already too complex. Block AVL or Block RB-tree would be even more complicated. Probably not friendly for novices to write it.

5. Literature references

Like BST, inserting or erasing elements would still cause unbalance of BBST. It is possible to use self-balancing BST in order to implement self-balanced BBST. The details and comparisons between different self-balancing BBST would be in another paper in the future.

5.1.1 Rotation of BBST The rotation of BBST is more complicated than BST. When we do rotation, we need move all elements of non-leaf nodes to fill the space of leaf nodes.

5.1.2 Block AVL Block AVL would be the best self-balancing BBST from theory since the each node is a block and rotation cost would be diminished to nearly zero. Also it provides the best performance of caching. However, AVL itself is not as widely used as RB tree and a lot of implementations could not be used for doing this Block AVL. It would require further discussions to determine whether it is actually good.

5.1.3 Block Red-Black Tree RB tree code are robust and widely implemented in a lot of languages. Block RB tree would reduce their work on maintaining different data structures compared to block AVL tree.
6. Acknowledgement

Thanks Bjarne Stroustrup and Herb Sutter [10] for providing vector vs linked list data graphs in order to prove the usefulness of making data structures to be contiguous.

Thanks Jon Kalb for his presentations about exception safe which helps me determines the exception safety of BBST.

Thanks Professor Dominic Duggan. He provided the concept from the Google File System/Hadoop File System to make the data large enough in order to avoid caching. That concept greatly inspired me for making this data structure.

Thanks Professor Sam Kim. He taught me about B+ tree indices in database management system.

Thanks my advisor David Klappholz as well.

References


[9] Bjarne Stroustrup and Herb Sutter. Prefer using stl vector by default unless you have a reason to use a different container, 2018.

