Cache-Oblivious: VanEmde-Boas-Trees

Jakob Eberhardt
jakob.eberhardt@estudiantat.upc.edu

June 9, 2025

Contents

Listings

List of Figures

List of Tables

1 Introduction & Problem Statement

In this assignment, we will employ VanEmde-Boas-Trees to study the benefits and importance of cache-optimized programming. Many workloads, e.g. database or graphics applications are so-called *memory bound*, meaning that their overall performance is dictated by how fast the CPU can read and write data to compute the problem. Although nowadays DRAM memory is typically available at a sufficiently low price, the latency for bringing data from DRAM or other external memories into the CPU is too long to saturate the CPU's capabilities in terms of computation, because the pipeline of a core in the CPU has to stall while waiting for data from a slow memory. To address this, microarchitects added multiple levels of increasingly fast but hence also more expensive memories in between the CPU and the external memories. These so-called caches are organized in lines of memory and enable the programmer to exploit temporal and spatial locality, meaning that the faster cache memory will hold the values of recently used variables and those that are close to it and likely accessed in the future. However, faster cache memories are expensive and hence many orders of magnitude smaller than the main memory, as can be seen in table 1. Therefore, if the CPU requests data while the respective cache level is already filled up, a cache line has to be evicted according to a certain policy, e.g. the least recently used line.

Table 1: Typical 2025 desktop memory hierarchy [1] [2] [3]. Caches are further decided in lines that can hold more than one variable of a program. The typical size is 64 bytes.

Memory level	**Typical capacity**	**Typical latency**
Registers	~2 KB	~0.3 ns
L1 Cache	~192 KB	~0.6 ns
L2 Cache	~3 MB	~2 ns
L3 Cache	~36 MB	~12 ns
Main Memory (DDR5)	~32 GB	~90 ns
Hard-Disk Drive	~8 TB	~5 ms

Since the cache resources are scarce, it requires the programmer to minimize the amount of expected cache misses in order to achieve higher levels of performance. In this case, we are considering binary search trees which we will construct and then query. To keep the experiment concise, we will only consider searches and will not mutate the tree after the initialization. We will compare the implementation of a plain search tree and a recursively built VanEmde-Boas tree in terms of their memory footprint and impact on caching.

2 Interface

In listing 1, we see the abstract interface for our BST implementations. We support three operations which are `insert` which allows us to insert an arbitrary `Key` into the tree, e.g. a `std::int64_t` integer value which we will use throughout the benchmarks. We will look up these keys using the `contains` function which will return true or false. Lastly, we define a `size_bytes()` function which is a helper function to return the used memory for a given implementation.

```
template<class Key>
class IBST {
public:
    virtual void insert(const Key& k)           = 0;
    virtual bool contains(const Key& k)    const = 0;
    virtual std::size_t size_bytes()       const = 0;
    virtual ~IBST() = default;
};
```

Listing 1: Abstract interface class for our Binary Search Tree implementation in `IBST.h`
The interface class will help us to avoid code duplication and also facilitate the creation of benchmarking instances. In the following sections 3 & 4 we will break down the actual implementations. The full code can be seen in the appendix section 8

3 Plain Binary Search Tree Implementation

In this section, we will break down the plain and canonical pointer binary search trees and their respective effects on memory layout. As can be seen in listing 2, the `BSTPtr` class implements our interface. To this end, we have to keep three data fields wrapped into the `Node` struct:

1. `Key`: E.g. a `std::int64_t` integer with eight bytes
2. `l` & `r`: Two `std::unique_ptr` pointers pointing to the left and right child node, each eight bytes.

Although the total memory used for the struct is 24 bytes, we probably have to take machine-specific padding and allocation memory overheads into account.

```cpp
template<class Key>
class BSTPtr : public IBST<Key> {
    struct Node {
        Key key;
        std::unique_ptr<Node> l, r;
        explicit Node(const Key& k) : key(k) {}
    };

    std::unique_ptr<Node> root_;
    std::size_t node_cnt_ = 0;
};
```

Listing 2: Plain Binary Search Tree implementation in `BSTPtr.h`

Additionally, we keep the root of our tree in `root_`. Hence, it will always be referenced when we later start to traverse the tree. Lastly, we keep the `node_cnt_` variable to later implement the `size_bytes()` function. This can be seen in listing 3 where we increment it every time we insert a node. The used memory can simply be computed by $node\_cnt\_ \times sizeof(Node)$.

3.1 Why Plain Insertion may be a Problem

The insertion of nodes into the tree is the critical point when looking at cache optimizations. In listing 3, we see how new nodes are inserted.

```cpp
void insertNode(std::unique_ptr<Node>& p, const Key& k) {
        if (!p) {
            p = std::make_unique<Node>(k);
            ++node_cnt_;
        } else if (k < p->key) {
            insertNode(p->l, k);
        } else if (k > p->key) {
            insertNode(p->r, k);
        }
    }
```

Listing 3: The internal `insertNode` implementation for standard BST in `BSTPtr.h`

We use `std::make_unique<Node>(k)` to allocate a new `Node` when the call to `insertNode` reaches a null pointer in order to obtain a unique pointer. If we check the C++-specific implementation in `unique_pointer.h` seen in listing 4, we see that this allocation happens on the heap, since the `new` keyword is used. Since the allocator will try to minimize the fragmentation on the heap, it likely will fill gaps in the heap with the small `Node` objects. This may cause performance issues since the memory addresses of two nodes which were created one after another may not map to the same cache line. During an insertion the algorithm walks from the root to the future parent of the new node, performing one key comparison per level. At each step, it dereferences either `l` or `r`, e.g. a pointer that almost certainly points to a line that is not yet in the cache. Hence, a random-order build of an n-node tree incurs $\Theta(n \log n)$ cache misses in addition to the usual $\Theta(n \log n)$ comparisons which is often the defining runtime factor.

```
template<typename _Tp, typename... _Args>
    _GLIBCXX23_CONSTEXPR
    inline __detail::__unique_ptr_t<_Tp>
    make_unique(_Args&&... __args)
    { return unique_ptr<_Tp>(new _Tp(std::forward<_Args>(__args)...)); }
```

Listing 4: The implementation of `std::make_unique<Node>(k)` in `unique_pointer.h` calling `new` for each node.

We will experimentally investigate if and how this allocation pattern which happens at construction time will affect the performance and cache-friendliness of the `contains` function of our plain BST. As can be seen in listing 5, we will start at the root node and iteratively decent the tree until we either find the key or encounter a null pointer. Hence, for every level we perform at most one comparison and one pointer dereference which may cause a cache miss.

```
static bool contains(const std::unique_ptr<Node>& p, const Key& k) {
        const Node* cur = p.get();
        while (cur) {
            if (k == cur->key) return true;
            cur = (k < cur->key) ? cur->l.get() : cur->r.get();
        }
        return false;
    }
```

Listing 5: The `contains` implementation of Plain BSTs in `BSTPtr.h`

3.2 Example

Even without fragmentation, there are reasons why a more sophisticated layout may be useful. In this example, we represent the integer keys 1-7 in memory according to the previously described plain binary search tree implementation, ensuring it remains perfectly balanced. During the experimentation, we used a random input order. In this case, we use a defined sequence in order to make the example comparable to the VEB implementation. The example tree can be seen in figure 1. If we apply the node insertion according to the code in listing 3, we see in figure 2 that although the subtree 4-2-6 will likely be located closely in memory, the subtree of 6, however, is separated from its children nodes which hence may map to different cache lines.

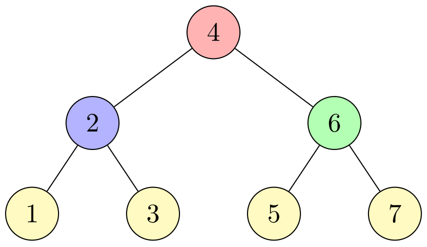

Figure 1: Example tree given the insertion sequence: $4 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 5 \rightarrow 7$.

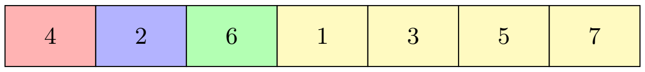

Figure 2: Plain BST insertion (level-order) memory blocks.

4 Van Emde-Boas Tree Implementation

Next, we will present the implementation of the Van Emde-Boas strategy for trees. The full implementation can be seen in the appendix section 8.3. First, if we consider the data fields of this implementation seen in listing 6, we see that we keep two vectors of our selected keys. We have to acknowledge that we do not rely on pointers to organize the data. We will see later how simple arithmetic will still allow us to query the tree efficiently with a smaller memory footprint. The array `inserts_` is an intermediate buffer for sorting out the insertion order of the keys. Once it is in the correct order, we will set the sentinel boolean variable `frozen_` to true which marks the actual and now immutable VEB array `a_` as valid.

```
template<class Key>
class BSTVEB : public IBST<Key> {
    std::vector<Key> a_;
    bool             frozen_ = false;
    std::vector<Key> inserts_;
```

Listing 6: Data Fields of the Van Emde-Boas Tree Implementation in `BSTVEB.h`
For a VEB tree, we aim at placing every complete subtree that is small enough to fit in one cache line stored contiguously in memory. To this end, we need a more sophisticated algorithm. Once we start to query the tree, we lock the insertion queue using the `freeze` function seen in listing 7. We sort the `inserts_` array and remove duplicates. Once the data is prepared, we paste the array to our `a_` array and start to recursively build the layout, as can be seen in listing 8.

```
void freeze() {
        if (frozen_) return;

        std::sort(inserts_.begin(), inserts_.end());
        inserts_.erase(std::unique(inserts_.begin(), inserts_.end()),
                       inserts_.end());

        a_.reserve(inserts_.size());
        help::build_veb(a_, inserts_, 0, inserts_.size());
        frozen_ = true;
    }
```

Listing 7: Freezing & Processing for Van Emde-Boas Tree Insertion Implementation in `BSTVEB.h`

The helper function `build_veb` will recursively generate the cache-oblivious layout. To this end, we find the median element of our input array and save it as the next index of our output array. Since the input is already ordered, in other words, we select the root of the current subtree and put it into the next memory address. Afterwards, we recursively build the layout for the left, then right subtree.

```
void build_veb(std::vector<Key>& out,
               const std::vector<Key>& sorted, std::size_t lo,
                   std::size_t hi)
{
    if (lo >= hi) return;
    std::size_t mid = (lo + hi) / 2;
    out.push_back(sorted[mid]);
    build_veb(out, sorted, lo, mid);
    build_veb(out, sorted, mid + 1, hi);
}
}
```

Listing 8: Insertion for Van Emde-Boas Tree Implementation in `BSTVEB.h`

The `containsRec` function seen in listing 9 performs a binary search over the VEB-ordered array `a_`. The arguments `lo` and `hi` limit the sorted key range. The `idx` variable points to the root of that range. If the range is empty, we return false, meaning we did not find the key and there are no segments left to explore. Otherwise, we compare the search key with `a_[idx]` and succeed immediately on a match. We then calculate the midpoint `mid` in the same way as during the construction phase of the layout. We observe that the left subtree occupies `left_size` = `mid - lo` elements immediately after the root, so the left and right children live at `idx` + 1 and `idx` + 1 + `left_size` respectively. Because those child indices are obtained by simple arithmetic instead of stored pointers, the search remains pointer-free while correctly descending the tree until we find the key or not.

```
bool containsRec(const Key& k,
                 std::size_t lo, std::size_t hi, std::size_t idx) const
    {
    if (lo >= hi) return false;

    const Key& key = a_[idx];
    if (k == key) return true;

    std::size_t mid      = (lo + hi) / 2;
    std::size_t left_size = mid - lo;
    std::size_t left_idx = idx + 1;
    std::size_t right_idx = idx + 1 + left_size;

    return (k < key)
        ? containsRec(k, lo, mid,        left_idx)
        : containsRec(k, mid + 1, hi,    right_idx);
    }
```

Listing 9: Recursive `contains` Function for Van Emde-Boas Tree Implementation in `BSTVEB.h`

The live memory that will be required during query time for the VEB implementation is given by the `size_bytes` function seen in listing 10. Since this implementation works pointer-free, we only need eight bytes to store an integer, unlike the plain implementation which requires 24. Hence, we can simply multiply the unique amount of keys we have by their size, e.g. eight for a `std::int64_t` integer.

```
std::size_t size_bytes() const override { return a_.size() *
     sizeof(Key); }
```

Listing 10: Cpmputing the required memory for the pointer-free Van Emde-Boas Tree implementation in `BSTVEB.h`

4.1 Example

In this example, we will lay out the same keys seen in section 3.2, but according to the Van Emde-Boas layout. The sequence can be seen in table 4.1. After we sorted the keys 1-7, we picked the median element 4 to be the root of the tree since it evenly divides the segment. Hence, it will get the VEB-index 0 or respectively the first address in memory as can be seen in figure 4. Afterwards, we repeat the same procedure for the left subtree 0-3 where 2 will become the root and VEB-index 1. Equally, we repeat the recursion on the left tree which now only contains 1, hence it becomes VEB-index 2 and we continue with the right subtree of its sibling which will also return and become VEB-index 3.
After we resolve the whole recursive building phase, we can observe how the children of the subtree roots 2 and 6 are located sequentially after them in

Table 2: VEB layout for the 7-key example.

VEB idx	**Key**	**Segment $[\mathbf{lo}, \mathbf{hi})$**	**Reason**
0	4	[0,7)	root of entire tree
1	2	[0,3)	root of left half
2	1	[0,1)	leaf
3	3	[2,3)	leaf
4	6	[4,7)	root of right half
5	5	[4,5)	leaf
6	7	[6,7)	leaf

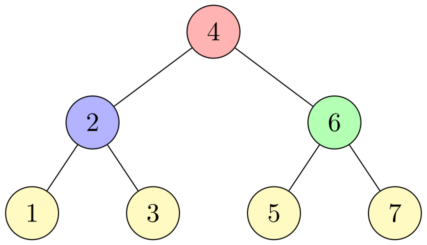

Figure 3: Binary-search tree arranged in VEB array order.

memory which increases the chances that they will be co-located in the same cache line. We will set up an experiment to check this theoretical guarantee in section 5.

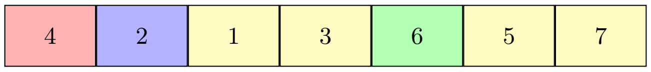

Figure 4: Memory View for the VEB-ordered layout

5 Experiment

In this section, we will introduce the experimental setup which will be used to benchmark the previously described implementations. An example of a configuration can be seen in listing 11 below. The parameter `n` is the total amount of keys we will insert upon a benchmark. The number of lookups is defined in the `q` parameter and `T` is the number of experiment repetitions. The `csv` flag is for plotting output and `seed` specifies the random number generator seed.

```json
{
  "n"   : 100000,
  "q"   : 100000,
  "T"   : 5,
  "csv" : false,
  "seed": 123
}
```

Listing 11: A small example instance configuration in `data/small.json`

We will compare the implementations and instance results in terms of the following metrics. We compute the averages among the `T` experiment repetitions:

- **Total Execution Time (`total_ns`)**: Average time (ns) taken to run a benchmark with a specific configuration (total time/`T`)
- **Nanoseconds per Search (`ns/search`)**: Average time (ns) taken for one lookup (`contains` function) (total query Time/`T`)
- **Total L3 Cache References (`cache_refs`)**: Average total L3 cache references (total cache references/`T`)
- **Total L3 Cache Misses (`cache_misses`)**: Average total L3 cache misses (total cache misses/`T`)
- **Bytes in Use (`bytes(MB)`)**: Bytes required to hold the `n` nodes

5.1 Experiment Driver

Listing 12 shows how we generate and process an experiment run. We can specify the variants we want to compile, e.g. the pointer-based BST implementation. By using a `Factory` pattern, we facilitate the integration of future implementations. For the insertion input, we generate `n` keys. Likewise, we populate a vector of length `q` with keys we will query during the benchmark. We accumulate the metrics over the `T` runs and compute their averages.

```cpp
const std::vector<Variant> variants = {
    {"BST_PTR", [] { return std::make_unique<BSTPtr<int>>(); }},
    {"BST_VEB", [] { return std::make_unique<BSTVEB<int>>(); }},
};

void runExperiment(int n, int q, int T, bool csv,
                   const Factory& make, unsigned seed,
                   const std::string& impl)
{
    std::mt19937 rng(seed);
    std::uniform_int_distribution<int> dist(1, n * 10);

    std::vector<int> inserts(n);
    for (int& x : inserts) x = dist(rng);

    std::vector<int> lookups(q);
    for (int& x : lookups) x = dist(rng);

    long long acc_ns  = 0;
    long long acc_refs = 0, acc_miss = 0;
    std::size_t bytes_used = 0;

    for (int t = 0; t < T; ++t) {
        auto tree = make();
        Metrics m = benchOnce(*tree, lookups, inserts);
        acc_ns  += m.ns;
        acc_refs += m.cache_refs;
        acc_miss += m.cache_miss;
        if (t == 0) bytes_used = tree->size_bytes();
    }

    double avg_ns    = double(acc_ns) / T;
    double ns_per_op = avg_ns / q;

    double avg_refs  = double(acc_refs) / T;
    double avg_miss  = double(acc_miss) / T;
    double miss_per_op = avg_miss / q;
    double miss_rate = (avg_refs > 0) ? avg_miss / avg_refs : 0.0;
    double avg_s     = avg_ns / 1e9;
    double bytes_mb  = bytes_used / 1024.0 / 1024.0;
```

Listing 12: Random input generation and bechmarkdriver in `benchmark.cpp`

5.2 Cache Monitoring

For our study, we want to be able to monitor and count the cache references and misses that occur during the execution of the program. The Linux tool perf [4] is a framework for accessing performance metrics and hardware counters, e.g. the L3 references and misses caused by a program. In listing 13 we can see the configuration which is used for our experiment. It defines a C++ wrapper around Linux's perf_event_open interface for measuring last-level-cache activity. The private helper openCounter() fills a perf_event_attr structure, invokes the __NR_perf_event_open system call, and returns a file-descriptor for either the PERF_COUNT_HW_CACHE_REFERENCES or PERF_COUNT_HW_CACHE_MISSES hardware event. start() resets both counters and enables them via the PERF_EVENT_IOC_RESET and PERF_EVENT_IOC_ENABLE ioctls; stop() disables the counters, reads the current values with read(), and caches the results in the data members refs_ and misses_, which our experiment driver code can access through the accessor functions refs() and misses(). In the configuration, we aim at only measuring the cache activity caused by our program, hence, e.g. we do not take references and misses into account which were caused by the kernel by setting exclude_kernel to true.

```
static int openCounter(uint32_t type, uint64_t config)
    {
        perf_event_attr pea{};
        pea.type           = type;
        pea.size           = sizeof(perf_event_attr);
        pea.config         = config;
        pea.inherit        = 0;
        pea.disabled       = 1;
        pea.exclude_kernel = 1;
        pea.exclude_hv     = 1;

        int fd = syscall(__NR_perf_event_open, &pea, 0, -1, -1, 0);
        if (fd == -1)
            throw std::runtime_error{"perf_event_open: " +
                std::string(strerror(errno))};
        return fd;
    }
```

Listing 13: Cache Monitoring & perf Setup in PerfCounters.h

The cache instrumentation is per-process, meaning every future benchmark will start on a cold cache. This way, we can ensure that no implementation benefits from remaining live lines in the cache. In fact, we run every implementation in a new process, as can be seen in the outer-level script in the appendix listing 20.

5.3 Testbed

We will run the experiment on a Intel(R) Core(TM) i7-8565U CPU at 1.80GHz. The memory hierarchy of this CPU can be seen in figure 5. Most importantly, we see that we have a total of four cores that have private L1 and L2 caches, yet they have to share the L3 cache. Since our application runs single-threaded, we therefore have the core-specific L1 data cache (32 KB), L2 cache (256 KB), and the shared L3 cache (8 MB).

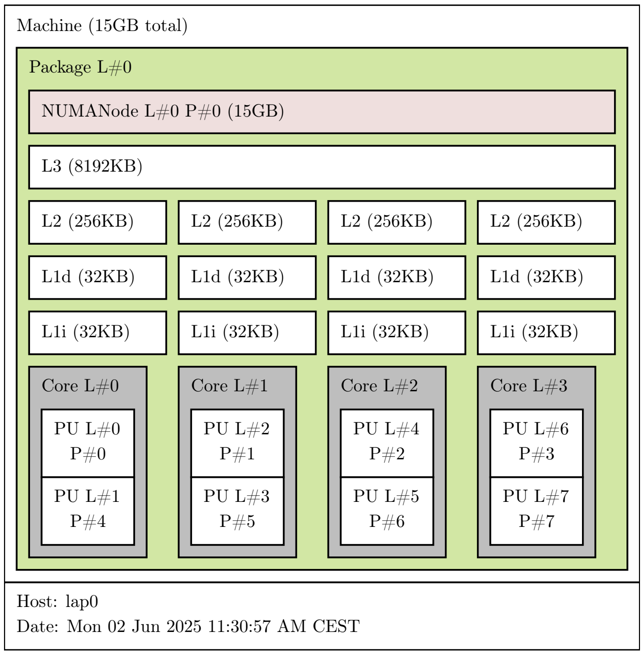

Figure 5: The available testbed memory hierarchy. The four cores have to share the 8MB of available L3 cache.

6 Results

In this section, we will present and analyze the results. In figure 6, we can see the evolution of the cache miss rate, meaning the percentage of referenced memory lines that have to be loaded from main memory while running q lookups on a tree with a total of n nodes where Miss Rate (%) is computed as $\left(\frac{\text{Number of Cache Misses}}{\text{Total Memory Accesses}}.\right) \times 100$. Most notably, there are almost no cache misses for the VEB implementation, especially if we take the mandatory cache misses into account which are necessary in any case to initially bring the required lines into the cache. The pointer-based implementation consistently causes more cache misses. Specifically, after we increased the size of the tree to $n \geq 300000$, we can see an order of magnitude of more cache misses.

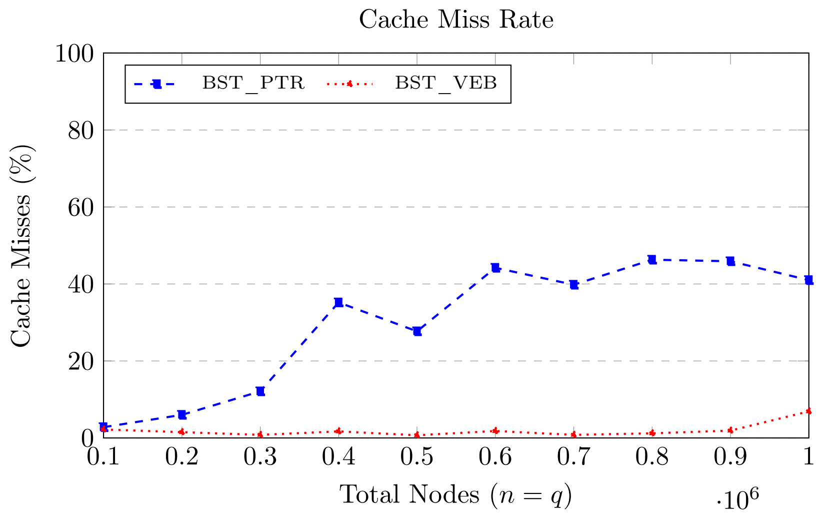

Figure 6: Cache miss rate (%) for nodes and lookups ($n = q$) for the pointer-based (`BST_PTR`) and VanEmde-Boas (`BST_VEB`) implementations.

If we recall the available L3 cache memory in the testbed shown in figure 5 which is 8 MB, we can follow the plot seen in figure 7 and see that at $n = 200000$, the memory required for the pointer-based implementation will amount to 4.4 MB which will likely fit into the cache, even though we have to consider other processes utilizing the cache. However, at $n = 300000$ we already require 6.5 MB of the total 8 MB which will likely cause contention on the cache which is expressed in the sudden increase in cache misses. Because of its much smaller memory footprint, the VEB implementation can basically fit the whole tree in L3 cache for the full series of input sizes up to 1.000.000 (only 3.6 MB in VEB-layout) which results basically zero cache misses besides the mandatory ones. This means that up to this instance size, basically, no node has to be evicted upon a traversal because all nodes fit into the cache.

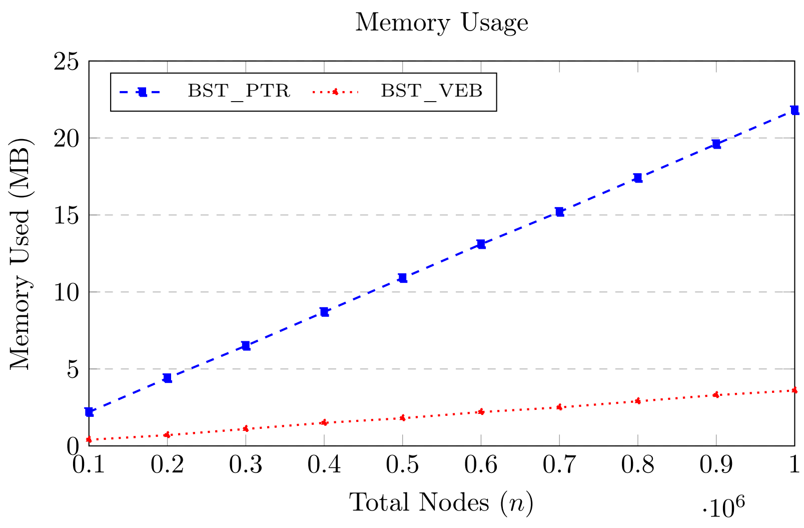

Figure 7: Memory usage in MB as a function of total nodes (n) for the pointer-based (`BST_PTR`) and VanEmde-Boas (`BST_VEB`) implementations.

However, because of the slight increase in cache misses caused by the VEB towards the end of the plot, we will now discuss the results for bigger instances, as seen in figure 8 & figure 9. The VEB implementation consistently causes fewer cache misses, however, the percentage of misses of both implementations converges once we allocate more than 5.5×10^6 of nodes where the pointer implementation requires about 120 MB and the VEB-layout 20 MB.

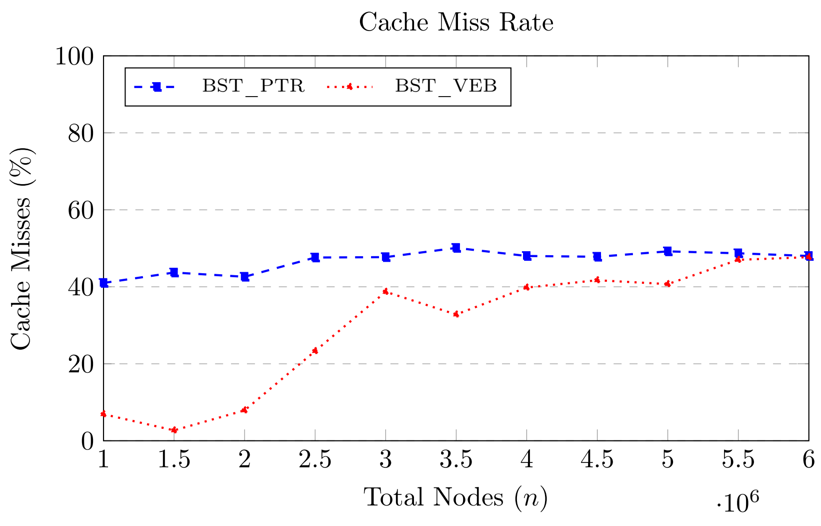

Figure 8: Cache miss rate (%) as a function of total nodes and lookups ($q = 1000000$) for the pointer-based (`BST_PTR`) and VanEmde-Boas (`BST_VEB`) implementations.

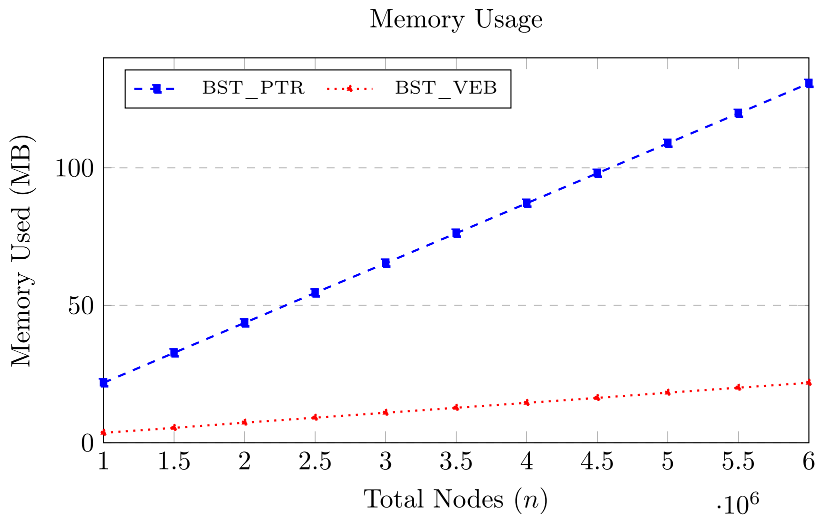

Figure 9: Memory usage in MB as a function of total nodes for the pointer-based (`BST_PTR`) and VanEmde-Boas (`BST_VEB`) implementations.

Although the `miss` rate seems to be similar once we use bigger instances, a miss implies a reference to the L3 cache, meaning it was not found in L1 or L2. This becomes clear when we plot the needed references to L3 to complete $q = 1000000$

lookups seen in figure 10. This means that even though the miss rate for the VEB layout may increase, more requested data fits into L1 and L2, resulting in fewer overall references to L3. When using the VEB layout, more nodes will fit into a cache line and additionally the nodes are mapped to memory such that a subtree root is followed up by its children which likely be in the same cache line or in a close one which will likely be prefetched.

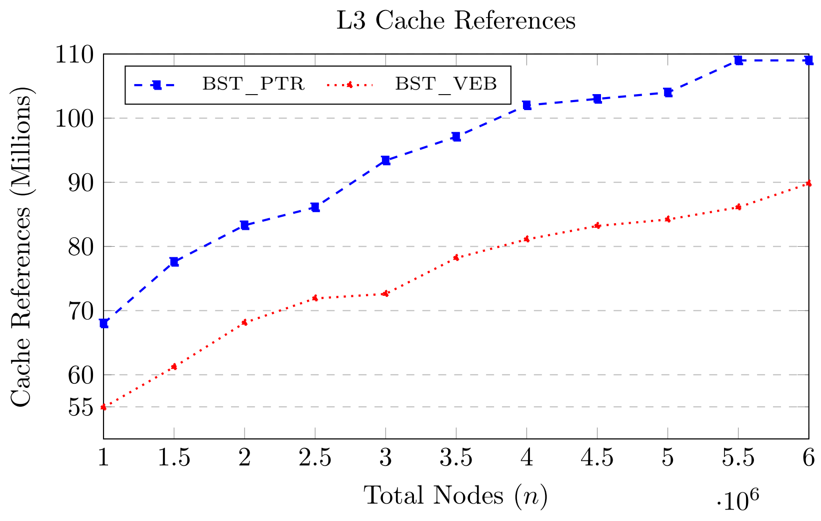

Figure 10: L3 Cache references in millions as a function of total nodes for $q = 1000000$ lookups for the pointer-based (`BST_PTR`) and Van Emde Boas (`BST_VEB`) implementations.

7 Conclusion

In this assignment, we have introduced the theoretical concepts regarding the importance of cache-friendly data structure implementations. To this end, we studied how Van Emde-Boas (VEB) trees are implemented and laid out in physical memory and compared them to plain binary search trees (BST). We instrumented the codes such that we can measure their impact on the L3 cache using the `perf` framework. We collected metrics regarding the performance of the implementation, e.g. the total memory used, and found that the pointerless VEB trees have a smaller memory footprint compared to standard BST. In terms of cache misses, we found that we have to put the results into the context of the testbed we are running the benchmarks in: We were able to observe how the plain BST implementation uses up the scarce cache memory much faster and hence triggers an order of magnitude more cache misses once the L3 cache is full. While we already saw contention and eviction when allocating 300000 nodes in the pointer-base implementation, we were still able to keep the whole VEB tree in the cache for up to one million nodes where we observed the first signs of contention. We found that both implementations converge towards a 40% miss rate if we increase the instance size to six million nodes. However, the VEB implementation required fewer references to the L3 cache overall, likely because it also improves locality on L1 and L2 caches. We were able to show how VEB trees improve cache friendliness and why they should be considered for relatively static cases where we do not have to insert and hence rebuild the layout frequently.

8 Appendix

8.1 IBST.h

```
#pragma once
#include <cstddef>

template<class Key>
class IBST {
public:
    virtual void insert(const Key& k)           = 0;
    virtual bool contains(const Key& k)    const = 0;
    virtual std::size_t size_bytes()       const = 0;
    virtual ~IBST() = default;
};
```

Listing 14: Abstract interface class for our Binary Search Tree implementationin in IBST.h

8.2 BSTPtr.h

```
#pragma once
#include "IBST.h"
#include <memory>
#include <utility>

template<class Key>
class BSTPtr : public IBST<Key> {
    struct Node {
        Key key;
        std::unique_ptr<Node> l, r;
        explicit Node(const Key& k) : key(k) {}
    };

    std::unique_ptr<Node> root_;
    std::size_t node_cnt_ = 0;

    void insertNode(std::unique_ptr<Node>& p, const Key& k) {
        if (!p) {
            p = std::make_unique<Node>(k);
            ++node_cnt_;
        } else if (k < p->key) {
            insertNode(p->l, k);
        } else if (k > p->key) {
            insertNode(p->r, k);
        }
    }

    static bool contains(const std::unique_ptr<Node>& p, const Key& k) {
        const Node* cur = p.get();
        while (cur) {
            if (k == cur->key) return true;
            cur = (k < cur->key) ? cur->l.get() : cur->r.get();
        }
        return false;
    }

public:
    void insert(const Key& k) override { insertNode(root_, k); }
    bool contains(const Key& k) const override { return contains(root_,
        k); }

    std::size_t size_bytes() const override { return node_cnt_ *
        sizeof(Node); }
};
```

Listing 15: Plain Binary Search Tree implementation in `BSTPtr.h`

8.3 `BSTVEB.h`

```
#pragma once
#include "IBST.h"
#include <vector>
#include <algorithm>
#include <stdexcept>


namespace help {
template<class Key>
void build_veb(std::vector<Key>& out,
               const std::vector<Key>& sorted, std::size_t lo,
                    std::size_t hi)
{
    if (lo >= hi) return;
    std::size_t mid = (lo + hi) / 2;
    out.push_back(sorted[mid]);
    build_veb(out, sorted, lo, mid);
    build_veb(out, sorted, mid + 1, hi);
}
}

template<class Key>
class BSTVEB : public IBST<Key> {
    std::vector<Key> a_;
    bool             frozen_ = false;
    std::vector<Key> inserts_;
```

```cpp


    void freeze() {
        if (frozen_) return;

        std::sort(inserts_.begin(), inserts_.end());
        inserts_.erase(std::unique(inserts_.begin(), inserts_.end()),
                       inserts_.end());

        a_.reserve(inserts_.size());
        help::build_veb(a_, inserts_, 0, inserts_.size());
        frozen_ = true;
    }

    bool containsRec(const Key& k,
                     std::size_t lo, std::size_t hi, std::size_t idx) const
    {
        if (lo >= hi) return false;

        const Key& key = a_[idx];
        if (k == key) return true;

        std::size_t mid      = (lo + hi) / 2;
        std::size_t left_size = mid - lo;
        std::size_t left_idx = idx + 1;
        std::size_t right_idx = idx + 1 + left_size;

        return (k < key)
            ? containsRec(k, lo, mid,        left_idx)
            : containsRec(k, mid + 1, hi,    right_idx);
    }

public:
    void insert(const Key& k) override {
        if (frozen_)
            throw std::logic_error("I am already frozen!");
        inserts_.push_back(k);
    }

    bool contains(const Key& k) const override {
        const_cast<BSTVEB*>(this)->freeze();
        return containsRec(k, 0, a_.size(), 0);
    }

     std::size_t size_bytes() const override { return a_.size() *
         sizeof(Key); }
};
```

Listing 16: Van Emde-Boas Tree Implementation in BSTVEB.h

8.4 benchmark.cpp

```cpp
#include "IBST.h"
#include "BSTPtr.h"
#include "BSTVEB.h"
#include "PerfCounters.h"

#include <vector>
#include <random>
#include <iostream>
#include <iomanip>
#include <fstream>
#include "util/json.hpp"
#include <functional>
#include <chrono>

using json = nlohmann::json;
using Clock = std::chrono::steady_clock;

struct Metrics {
    long long ns         = 0;
    long long ops        = 0;
    long long cache_refs = 0;
    long long cache_miss = 0;
};

template<class Key>
Metrics benchOnce(IBST<Key>& tree,
                  const std::vector<Key>& lookups,
                  const std::vector<Key>& inserts)
{
    for (const auto& k : inserts) tree.insert(k);

    PerfCounters pc;
    pc.start();

    auto start = Clock::now();
    for (const auto& k : lookups)
        (void)tree.contains(k);
    auto end  = Clock::now();

    pc.stop();

    Metrics m;
    m.ns       =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end -
        start).count();
    m.ops      = lookups.size();
    m.cache_refs = pc.refs();
    m.cache_miss = pc.misses();
    return m;
}

using Factory = std::function<std::unique_ptr<IBST<int>>(void)>;
struct Variant { std::string name; Factory make; };

const std::vector<Variant> variants = {
    {"BST_PTR", [] { return std::make_unique<BSTPtr<int>>(); }},
    {"BST_VEB", [] { return std::make_unique<BSTVEB<int>>(); }},
};

void runExperiment(int n, int q, int T, bool csv,
                   const Factory& make, unsigned seed,
                   const std::string& impl)
{
    std::mt19937 rng(seed);
    std::uniform_int_distribution<int> dist(1, n * 10);

    std::vector<int> inserts(n);
    for (int& x : inserts) x = dist(rng);

    std::vector<int> lookups(q);
    for (int& x : lookups) x = dist(rng);

    long long acc_ns  = 0;
    long long acc_refs = 0, acc_miss = 0;
    std::size_t bytes_used = 0;

    for (int t = 0; t < T; ++t) {
        auto tree = make();
        Metrics m = benchOnce(*tree, lookups, inserts);
        acc_ns  += m.ns;
        acc_refs += m.cache_refs;
        acc_miss += m.cache_miss;
        if (t == 0) bytes_used = tree->size_bytes();
    }

    double avg_ns    = double(acc_ns) / T;
    double ns_per_op = avg_ns / q;

    double avg_refs  = double(acc_refs) / T;
    double avg_miss  = double(acc_miss) / T;
    double miss_per_op = avg_miss / q;
    double miss_rate = (avg_refs > 0) ? avg_miss / avg_refs : 0.0;
    double avg_s     = avg_ns / 1e9;
    double bytes_mb  = bytes_used / 1024.0 / 1024.0;

    if (csv) {
        std::cout << impl << ','
                  << n << ',' << q << ','
```

```cpp
                  << avg_ns << ','
                  << std::fixed << std::setprecision(2)
                  << avg_s << ','
                  << std::defaultfloat
                  << ns_per_op << ','
                  << avg_refs << ',' << avg_miss << ','
                  << miss_per_op << ','
                  << miss_rate << ','
                  << bytes_used << '\n';
    } else {
        std::cout << std::fixed << std::setprecision(2)
                  << std::left
                  << std::setw(10) << impl
                  << std::setw(10) << n
                  << std::setw(10) << q
                  << std::setw(15) << avg_ns
                  << std::setw(10) << avg_s
                  << std::defaultfloat << std::setprecision(6)
                  << std::setw(15) << ns_per_op
                  << std::setw(15) << avg_refs
                  << std::setw(15) << avg_miss
                  << std::setw(12) << miss_per_op
                  << std::setw(10) << std::fixed << std::setprecision(3)
                      << miss_rate
                  << std::setw(12) << std::fixed << std::setprecision(1)
                      << bytes_mb
                  << '\n';
    }
}

int main(int argc, char* argv[])
{
    int        n    = 10000;
    int        q    = 10000;
    int        T    = 1;
    bool       csv  = false;
    unsigned   seed = 42;
    std::string impl = "ALL";

    if (argc >= 2) {
        std::ifstream in(argv[1]);
        if (!in) { std::cerr << "Cannot open " << argv[1] << '\n';
            return 1; }
        json cfg; in >> cfg;
        if (cfg.contains("n"))  n    = cfg["n"];
        if (cfg.contains("q"))  q    = cfg["q"];
        if (cfg.contains("T"))  T    = cfg["T"];
        if (cfg.contains("csv")) csv = cfg["csv"];
        if (cfg.contains("seed")) seed = cfg["seed"];
        if (cfg.contains("impl")) impl = cfg["impl"];
    }

    if (argc == 3) impl = argv[2];

    if (!csv) {
        std::cout << std::left
                  << std::setw(10) << "impl"
                  << std::setw(10) << "n"
                  << std::setw(10) << "q"
                  << std::setw(15) << "total_ns"
                  << std::setw(10) << "total_s"
                  << std::setw(15) << "ns/search"
                  << std::setw(15) << "cache_refs"
                  << std::setw(15) << "cache_miss"
                  << std::setw(12) << "miss/search"
                  << std::setw(10) << "missRate"
                  << std::setw(12) << "bytes(MB)" << '\n'
                  << std::string(134, '-') << '\n';
    } else {
        std::cout << "impl,n,q,total_ns,total_s,ns_per_search,"
            "cache_refs,cache_misses,misses_per_search,miss_rate,bytes\n";
    }

    for (const auto& v : variants) {
        if (impl != "ALL" && impl != v.name) continue;
        runExperiment(n, q, T, csv, v.make, seed, v.name);
    }
    return 0;
}
```

Listing 17: Benchmarking Driver in benchmark.cpp

8.5 PerfCounters.h

```cpp
#pragma once
#include <linux/perf_event.h>
#include <cstdint>
#include <sys/syscall.h>
#ifndef __NR_perf_event_open
#endif
#include <sys/ioctl.h>
#include <unistd.h>
#include <cstring>
#include <cerrno>
#include <stdexcept>
#include <array>

class PerfCounters {
   int fd_refs_{-1}, fd_misses_{-1};
   long long refs_{0}, misses_{0};

   static int openCounter(uint32_t type, uint64_t config)
   {
       perf_event_attr pea{};
       pea.type           = type;
       pea.size           = sizeof(perf_event_attr);
       pea.config         = config;
       pea.inherit        = 0;
       pea.disabled       = 1;
       pea.exclude_kernel = 0;
       pea.exclude_hv     = 1;

       int fd = syscall(__NR_perf_event_open, &pea, 0, -1, -1, 0);
       if (fd == -1)
           throw std::runtime_error{"perf_event_open: " +
               std::string(strerror(errno))};
       return fd;
   }
public:
   PerfCounters()
   {
       fd_refs_  = openCounter(PERF_TYPE_HARDWARE,
           PERF_COUNT_HW_CACHE_REFERENCES);
       fd_misses_ = openCounter(PERF_TYPE_HARDWARE,
           PERF_COUNT_HW_CACHE_MISSES);
   }
   ~PerfCounters() { close(fd_refs_); close(fd_misses_); }

   inline void start() const { ioctl(fd_refs_, PERF_EVENT_IOC_RESET,
       0); ioctl(fd_misses_, PERF_EVENT_IOC_RESET, 0);
                             ioctl(fd_refs_, PERF_EVENT_IOC_ENABLE, 0);
                                 ioctl(fd_misses_,
                                 PERF_EVENT_IOC_ENABLE, 0); }
   inline void stop()
{
   ioctl(fd_refs_, PERF_EVENT_IOC_DISABLE, 0);
   ioctl(fd_misses_, PERF_EVENT_IOC_DISABLE, 0);

   if (read(fd_refs_, &refs_, sizeof(refs_)) != sizeof(refs_))
       throw std::runtime_error("failed to read cache-refs counter");
   if (read(fd_misses_, &misses_, sizeof(misses_)) != sizeof(misses_))
       throw std::runtime_error("failed to read cache-miss counter");
}
   long long refs() const { return refs_; }
   long long misses() const { return misses_; }
};
```

Listing 18: Instrumentation Setup for Perf in `PerfCounters.h`

8.6 `small.json`

```json
{
  "n"   : 100000,
  "q"   : 100000,
  "T"   : 5,
  "csv" : false,
  "seed": 123
}
```

Listing 19: Example Instance in `small.json`

8.7 `run_bench.sh`

```bash
#!/bin/bash

make

IMPLS=("BST_PTR" "BST_VEB")

for impl in "${IMPLS[@]}"; do
  ./bst-bench "$1" "$impl"
  echo

  echo
done
```

Listing 20: Benchmarking Script in `run_bench.sh`. Every implementation is started in a new process.

References

[1] Chester Lam. *Analyzing Lion Cove's Memory Subsystem in Arrow Lake.* Chips and Cheese technical article. Jan. 6, 2025. URL: `https://chipsandcheese.com/p/analyzing-lion-coves-memory-subsystem` (visited on 06/06/2025).

[2] Micron Technology, Inc. *The Difference Between RAM Speed and CAS Latency.* Crucial Knowledge-Base article. 2025. URL: `https://www.crucial.com/articles/about-memory/difference-between-speed-and-latency` (visited on 06/06/2025).

[3] Serkay Olmez et al. "Revisiting HDD Rules of Thumb: 1/3 Is Not (Quite) the Average Seek Distance". In: *20th IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies (MSST '24).* 2024. URL: `https://www.msstconference.org/MSST-history/2024/Papers/msst24-1.1.pdf` (visited on 06/06/2025).

[4] perf: Linux profiling with performance counters. *perf: Linux profiling with performance counters, mainpage.* `https://perfwiki.github.io/main/`. Accessed: 5 June 2025. 2025.