Eytzinger Binary Search Trees: Data Prefetching & Branch Prediction

Advanced Data Structures

Jakob Eberhardt
jakob.eberhardt@estudiantat.upc.edu

June 15, 2025

Contents

Listings

List of Figures

List of Tables

1 Introduction & Problem Statement

In this project, we will implement and compare different pointer-less implementations of binary search trees (BST) in terms of overall performance, cache-friendliness, and efficiency. We will study the impact of increasingly aggressive prefetching regarding the performance and cache pressure. Additionally, we will study how more complex and sophisticated implementations which tend to add more branching to our program will affect the CPUs ability to correctly predict the branch direction which can have a major impact on overall performance.

1.1 The Cost of Branch Misprediction

The reason why branch mispredictions represent a big obstacle for modern machines has to do with the design choices described so far. Another central component of modern designs is branch predictors. Branch predictors allow for one side of conditional branches to be speculatively executed before the branch is committed. When the speculated side and the evaluated side coincide, we have the benefit of keeping the pipeline full and executing instructions ahead of time. When these speculations fail, the speculatively executed instructions have to be discarded and the correct branch target needs to be executed. Due to out-of-order execution, between the time the branch is speculatively executed and the time it is evaluated, a high number of instructions might have been retired. In addition to this, as the pipelines are deep, several cycles have to pass before the pipeline is filled again. Based on the work of Kwan Lin et al. [1], branch misprediction accounts for 20% of the IPC in modern processors and represents the main limit to having deeper, more efficient pipelines. A good empirical average measure of the cost of a branch misprediction comes from the weight used in the heuristics of *LLVM* [2] reported in table 1.

Architecture	**Misprediction Penalty**	**Optimistic Load Cost**
Sapphire Rapids	14	5
Alder Lake-P	14	5
Ice Lake	14	5
Broadwell	16	5
Haswell	16	5
Cortex A57	14	4
Cortex R52	8	1
Cortex M4	2	2

Table 1: Branch Misprediction Penalty and Optimistic Load Cost used in *LLVM*'s heuristics for various Intel and ARM architectures.

2 Implementation

In the following, we will introduce the seven binary search tree implementations. Listing 1 shows the common interface. It consists of the `insert` function which will add an arbitrary comparable key, e.g. an integer value. The boolean function `contains` will return true or false depending on the presence of the key in the tree. The helper function `size_bytes()` returns the memory needed to hold all keys which is important if we want to interpret and analyze cache behavior later.

```
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`

2.1 VanEmde-Boas Tree

The VanEmde-Boas tree is a recursive method to lay out the tree in memory as a perfectly balanced binary search tree in a pointer-free array such that nodes likely to be visited together sit close in memory or ideally in the same cache line. To this end, we choose each segment's median as the root, then recursively write the left half immediately after it and the right half after the left subtree, producing a so-called *cache oblivious* layout, which aims at reducing cache misses independently of the underlying architecture of the cache memories. This process guarantees every subtree occupies one contiguous power-of-two block, so the algorithm can compute child-root positions with simple index arithmetic instead of pointers, as can be seen in listing 2. The query helper `containsRec` takes the candidate key plus the inclusive bounds `[lo, hi)` and the current root index `idx`, hence, it works without any need to chase a pointer across the heap. After comparing the search key to `a_[idx]`, the function either recurses into `left_idx`, which is located one slot past the root, or into `right_idx` which is offset by the left-subtree size. This shrinks the interval by half each step. Because the layout keeps each level's nodes on adjacent cache lines, a lookup still needs only $\mathcal{O}(\log n)$ probes but typically touches far fewer cache lines than a pointer-based tree, giving a higher performance in practice. The full code can be seen in the appendix listing 18.

```
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 2: Overview of the recursive Van Emde-Boas layout search-tree implementation in `BSTVEB.h` reported as `BST_VEB`.

2.2 Eytzinger Tree

The Eytzinger layout [3] writes a complete binary search tree into an array in breadth-first order, so a node at position `i` has children at `2i + 1` and `2i + 2`. The array is built recursively, as can be seen in listing 3. During construction, an in-order traversal over the sorted keys stores each value into `arr_` while recursing first to `2idx + 1`, then the current slot, then `2idx + 2`, guaranteeing the array still represents a valid BST. Unlike the van Emde Boas layout, which aims at packing every recursively balanced subtree into a contiguous cache block, Eytzinger interleaves nodes from different subtrees across successive cache lines. This way, the arithmetic progression `2i + 1`/`2i + 2` is so predictable that modern hardware prefetchers can pull the next cache line well before the branch outcome is known. Consequently, the `contains` loop simply updates `i` with a multiply-add and performs a comparison, yielding a tight, branch-friendly sequence with no pointer chasing. The full code of the implementation is shown in the appendix listing 19.

```
void buildEyt(std::size_t idx, std::size_t& pos,
                const std::vector<Key>& sorted)
    {
        if (idx >= sorted.size()) return;
        buildEyt(2*idx + 1, pos, sorted);
        arr_[idx] = sorted[pos++];
        buildEyt(2*idx + 2, pos, sorted);
    }

```

```
bool contains(const Key& k) const override {
       const_cast<BSTEyt*>(this)->freeze();

       std::size_t i = 0;
       while (i < arr_.size()) {
           if (k == arr_[i])        return true;
           i = (k < arr_[i]) ? 2*i + 1 : 2*i + 2;
       }
       return false;
   }
```

Listing 3: An overview of the Eytzinger-layout binary-search-tree implementation in `BSTEyt` reported as `BST_EYT`.

2.3 Eytzinger Tree with Prefetching

In this section, we will extend the standard Eytzinger implementation of section 2.2 with programmer-controlled prefetching to benchmark its performance. Prefetching means issuing a non-blocking hint to the memory subsystem so that the cache line holding data which possibly will be needed soon will be pulled into the cache before the CPU really needs it. The goal is to overlap main-memory round-trip with useful computation, turning what would have been a hard stall into ideally zero visible latency. To implement it, we can simply calculate the indices of the to-be-prefetched child nodes and issue a respective prefetch using the function `__builtin_prefetch` seen in listing 4 which will be discussed in detail in the following. The full code can be seen in the appendix listing 20

```
bool contains(const Key& k) const override {
       const_cast<BSTEytPref*>(this)->Base::freeze();

       const auto& a = Base::arr_;
       std::size_t i = 0;

       while (i < a.size()) {
           std::size_t l = 2*i + 1;
           std::size_t r = l + 1;
           if (l < a.size()) __builtin_prefetch(&a[l], 0, 1);
           if (r < a.size()) __builtin_prefetch(&a[r], 0, 1);

           if     (k == a[i]) return true;
           else if (k < a[i]) i = l;
           else               i = r;
       }
       return false;
   }
```

Listing 4: Single-level software prefetch variant of the Eytzinger implementation in `BSTEytPrefetch.h` reported as `BST_EYT_PREF`.

2.4 Prefetching in Practice

Listing 5 shows the prefetch function and its parameters. The address variable `addr` is a pointer to the start of the cache line which we want to fetch, in our case `&a[l]` or `&a[r]` which were formally computed. The second parameter `rw` hints to the memory controller if we want the line in read-only or read-write mode. Depending on the hardware, we have a better chance of getting the memory line faster if we hint read-only, for example, because the memory system can avoid certain consistency protocols in multi-core systems. Hence, we use `0` to request a read-only line. Practically, we could still write to it, however. Lastly, we can specify the likelihood that we will reuse the line from `0-3` using the `locality` parameter. Zero indicates that we do not want to keep it and that it can be evicted as soon as needed. With locality level three, we ask the memory subsystem to keep the line hot as long as possible. In principle, we could adapt the locality level of a prefetched line to its position in the tree: If it is close to the root or the root itself, we always want it in the cache because many other or all queries will need this cache line. A leaf, however, can be evinced as soon as we find it because it is very unlikely that we will need this node or one close to it again in the future. In our implementations, we always use locality level one.

```
__builtin_prefetch(const void* addr,
                   int         rw   = 0,
                   int         locality = 3);
```

Listing 5: Prefetch Function and Parameters used in `BSTEytPrefetch.h`

Listing 6 shows how the prefetching is compiled to machine instructions. In this case, `lea rax,[rdx+rdx1]` doubles the loop index in `rdx`, putting $2 \times i$ in `rax`. Then `lea rdi,[rax+0x1]` and `add rax,0x2` compute the child indices `rdi = 2i + 1` (left) and `rax = 2i + 2` (right). The comparison `cmp rdi,rcx` tests whether the left child is beyond the array size, and if not, `prefetcht2 BYTE PTR [r8+rdi4]` pulls the hopefully soon-to-be-touched element into the L2 cache to hide memory latency.

```
lea    rax,[rdx+rdx*1]
lea    rdi,[rax+0x1]
add    rax,0x2
cmp    rdi,rcx
jae    4e <BSTEytPref<int>::contains(int const&) const+0x4e>
prefetcht2 BYTE PTR [r8+rdi*4]
```

Listing 6: The resulting Assembly code of the Single-level software prefetch into the L2 variant of the Eytzinger implementation.

2.5 Eytzinger Tree with two-level Prefetching

This variant of the `contains` function for the Eytzinger layout works the same as the prefetching implementation seen before in section 2.3, but it issues prefetch instructions for two levels of child nodes per step. This way, we can benchmark how more aggressive but plain prefetching affects the performance and cache pressure. As can be seen in listing 7, we simply calculate the indices of the grand-child nodes and prefetch them if they are within the array bounds. The full code is listed in appendix listing 21.

```
std::size_t l = 2*i + 1;
std::size_t r = l + 1;
if (l < a.size()) __builtin_prefetch(&a[l], 0, 1);
if (r < a.size()) __builtin_prefetch(&a[r], 0, 1);

std::size_t ll = 4*i + 3;
std::size_t lr = ll + 2;
std::size_t rl = 4*i + 5;
std::size_t rr = rl + 2;

if (ll < a.size()) __builtin_prefetch(&a[ll], 0, 1);
if (lr < a.size()) __builtin_prefetch(&a[lr], 0, 1);
if (rl < a.size()) __builtin_prefetch(&a[rl], 0, 1);
if (rr < a.size()) __builtin_prefetch(&a[rr], 0, 1);
```

Listing 7: Two-level software prefetch variation in `BSTEytPrefetchTwo.h` reported as `BST_EYT_PREF_TWO`.

2.6 Eytzinger Tree with three-level Prefetching

The implementation seen in listing 9 works in analogy to the prefetching implementation presented in section 2.3, however, it triggers a special compiler optimization which is explained in more detail in subsection 2.6.1. It uses the lambda function `pf` seen in listing 8 to safely prefetch three generations of children whose indices have been calculated prior. The full implementation can be found in the appendix listing 22.

```
auto pf = [&](std::size_t idx) {
          if (idx < a.size()) __builtin_prefetch(&a[idx], 0, 1);
      };
```

Listing 8: Prefetching Lambda Expression with bound checks in `BSTEytPrefetchThree.h`

```
std::size_t l = 2*i + 1;
std::size_t r = l + 1;
pf(l); pf(r);

std::size_t ll = 2*l + 1;
std::size_t lr = ll + 1;
std::size_t rl = 2*r + 1;
std::size_t rr = rl + 1;
pf(ll); pf(lr); pf(rl); pf(rr);

pf(2*ll + 1); pf(2*ll + 2);
pf(2*lr + 1); pf(2*lr + 2);
pf(2*rl + 1); pf(2*rl + 2);
pf(2*rr + 1); pf(2*rr + 2);

if      (k == a[i]) return true;
else if (k < a[i]) i = l;
else               i = r;
```

Listing 9: Three-level software prefetch variant in `BSTEytPrefetchThree.h` reported as `BST_EYT_PREF_THREE`.

2.6.1 Predication & If-Convertion

In the results section 4 we can see that in fact, this implementation misses a lot fewer branching instructions compared to others, even though the concept is fundamentally the same and the executed branches are in the same order of magnitude. However, after a careful study of the actual binary, we were able to find the reason for this drastic result. In all of the implementations, the branch predictor of the CPU will pick up quickly on the branching behavior of code sections like `while` loops, bound checks, or the `if (k == a[i])` check which is usually false. Unlike this, the final `else if` branch which determines if we go left or right is very hard to predict and hence dictates many performance metrics of the program. Upon closer inspections of the machine instruction generated by `gcc`, it turned out that the compiler applied the so-called *if-conversion* [4] optimization to the critical `else if` statement only in the case of the three-level prefetch variant. This control flow optimization employs predication [5] which uses conditional instructions such as `cmov` to enable branchless control flow. In our case, it basically removes the risk of a branch misprediction, as can be seen in the result section in plot 9.

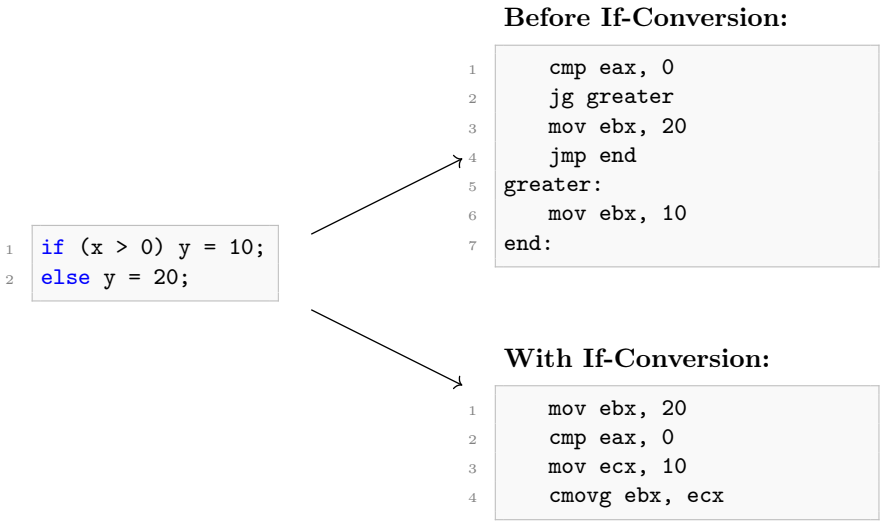

Figure 1: If-conversion applied to a conditional assignment in C++, showing both the original x86 assembly with branching and the optimized version using the conditional move (`cmov`) instruction.

In figure 1 we can see the concept of predication. With branching instructions such as `jg` and `jmp`, the CPU will have to guess the outcome of the branch and speculatively execute one branch. If the branch was predicted wrong, the computed values have to be squashed once the control-dependent instruction, in this case the `cmp` instruction, was committed. In the `if-converted` example, we can do multiple things. Once, we have no branching instructions and hence also no branching target labels. Additionally, we can see the `cmovg` instructions which move the value of register `ecx` into register `ebx` if the greater-then flag was set by the `cmp` instruction earlier. This can be useful if the performance of the program is not limited by functional resources, e.g. ALUs but rather by latencies in the pipeline, e.g. because we took a cache miss and have to wait for the value of `x` to compare it.

```
prefetcht2 BYTE PTR [rdx+r8*4]
add    rcx,0x4
cmp    rcx,rax
jae    1919c <BSTEytPrefThree<int>::contains(int const&) const+0x10c>
prefetcht2 BYTE PTR [rdx+rdi*8+0x18]
mov    edi,DWORD PTR [rdx+rdi*1-0x4]
cmp    DWORD PTR [rbx],edi
je     191c0 <BSTEytPrefThree<int>::contains(int const&) const+0x130>
cmovl  rsi,r9
```

Listing 10: Assembly of the `BSTEytPrefThree<int>::contains` function which uses the predicated `cmovl` instruction reported as `BST_EYT_PREF_THREE_IFC`.

In listing 10 we can see part of the actual `contains` function assembly code of the three-level prefetching implementation which is the only one that includes conditinal instructions. The register `edi` holds the value of `a[i]` so the value is stored in the current node. Register `rbx` holds the address of the search key `k`. The `je` instruction corresponds to the `if (k == a[i]) return true` path seen in listing 9. It has to be speculatively executed before the `cmp` instruction above can commit, however, it is easy to predict, because it will not be taken most of the time. The final `cmovl` instruction corresponds to the `k < a[i]` case. In fact, register `r9` will hold the index of the left child which will be put into the target register if we go left. If we go right, the target register `rsi` already holds the index of the right child.
In this case, the CPU can do useful work by executing both branches and then just shift and push the right value into the final register. In our case, however, we are certainly memory-bound. Therefore, this optimization reduces mispredictions, but the overall performance is dictated by the latency to get the child nodes from memory. This can be seen in the results section figure 3 which shows the nanoseconds per search. Although the three-level prefetch implementation has practically no branch mispredictions, the performance is still in the same area. To prevent this optimization in order to make the implementations comparable, we will use the compiler flags `-fno-if-conversion` and `-fno-if-conversion2` to disable if conversion.

2.7 Eytzinger Tree with four-level Prefetching

In this aggressive variant of the prefetching implementation, we use a simple queue to hold for generations of child node indices to prefetch them upon the execution of the `contains` function, as can be seen in listing 11. The full code can be found in the appendix listing 23.

```
std::size_t q[32];
int front = 0, back = 0;
q[back++] = i;

for (int depth = 0; depth < 4; ++depth) {
    int levelCount = back - front;
    for (int n = 0; n < levelCount; ++n) {
        std::size_t parent = q[front++];
        std::size_t l = 2*parent + 1;
        std::size_t r = l + 1;
        pf(a, l); pf(a, r);
        q[back++] = l;
        q[back++] = r;
    }
}
```

Listing 11: Four-level software prefetch variant in `BSTEytPrefetchFour.h` reported as `BST_EYT_PREF_FOUR`.

2.8 Eytzinger Tree with probability-guided Prefetching

Prefetching memory lines can help to hide latencies. However, in the formally described implementations, we are guaranteed to prefetch memory lines which we will not use for the execution of the `contains` function. Depending on how many generations we prefetch, we will put more and more pressure on the cache by loading memory which has no practical use but rather causes disturbance in the execution. To address this issue, we present a probability-based implementation that uses the key value `k` to decide in which direction the program should statistically invest more prefetches from a given budget. To this end, we first have to determine the minimum and maximum key of the tree using the `ensureMinMax` function seen in listing 12. It is used to guard the execution of the `contains` function seen in listing 13 to make sure we obtained the minimum and maximum upon the first query. Since this will only have to run once, we decorate the immediate return branch with `[[likely]]` and `__builtin_expect` to give a hint to the compiler to optimize this path. In any case, the runtime branch predictor would probably quickly pick up on the branching behavior.

```
[[gnu::always_inline]] inline void ensureMinMax() const
   {
       if (__builtin_expect(minmax_ready_, 1)) [[likely]]
           return;

       const auto& a = Base::arr_;
       if (__builtin_expect(a.empty(), 0)) [[unlikely]] return;
       auto [mn, mx] = std::minmax_element(a.begin(), a.end());
       min_key_ = *mn;
       max_key_ = *mx;
       minmax_ready_ = true;
   }
```

Listing 12: Programmer-guided Optimization for `ensureMinMax()` Function in `BSTEytPrefProb.h`.

We then use the `min_key_` and `max_key_` to allocate the total budget of prefetched lines, e.g. eight. For example, if key `k` is `max_key_` - 1, it makes much more sense to aggressively prefetch to the right rather than spend the budget on very unlikely cache lines. This way, we can avoid a certain amount of useless prefetches and hence useless traffic on the memory bandwidth and cache contention. The full code can be seen in the appendix listing 24.

```
bool contains(const Key& k) const override
    {
        const_cast<BSTEytPrefProb*>(this)->Base::freeze();
        const auto& a = Base::arr_;
        if (a.empty()) return false;

        ensureMinMax();

        double ratio = (max_key_ == min_key_)
                    ? 0.5
                    : double(k - min_key_) / double(max_key_ - min_key_);
        ratio = std::clamp(ratio, 0.0, 1.0);

        std::size_t spent = 0;
        std::size_t i = 0;
        std::size_t l = 1;
        std::size_t r = 2;

        if (l < a.size()) { pf(a, l); ++spent; }
        if (r < a.size()) { pf(a, r); ++spent; }

        std::size_t remaining = (Budget > spent) ? Budget - spent : 0;
        std::size_t left_budget = std::size_t(std::round(remaining *
            (1.0 - ratio)));
        if (left_budget > remaining) left_budget = remaining;
        std::size_t right_budget = remaining - left_budget;

        prefetchSubtree(a, l, left_budget);
        prefetchSubtree(a, r, right_budget);

        while (i < a.size()) {
            const Key& key = a[i];
            if (k == key) return true;
            i = (k < key) ? 2 * i + 1 : 2 * i + 2;
        }
        return false;
    }
```

Listing 13: Probabilistic, path-biased prefetcher in `BSTEytPrefProb.h` reported as `BST_EYT_PREF_PROB`.

3 Metrics & Measurement

In our experimental setup, we can specify the repetitions of the individual experiment by using the configuration parameter `T`. Hence, the metrics regarding total values, e.g. total time in nanoseconds report the average among the `T` repetitions. How we extract these metrics is explained in section 3.1.

- **ns_per_search**: Average time in nanoseconds spent for a `contains` query.
- **misses_per_search**: Average total cache misses for a `contains` query .
- **miss_rate**: The total cache miss rate, meaning the percentage of load operations where we have to go to the main memory.
- **l1_rate**: The L1 miss fraction in percent.
- **l3_rate**: The L3 or last-level miss fraction in percent.
- **branches**: Total static branch instructions retired during the benchmark.
- **branch_rate**: Quantifying predictor accuracy, meaning the fraction of branches that were mispredicted
- **branch_per_search**: The average amount branches we have to execute per `contains` function

3.1 Hardware Counters & `Perf`

In this project, we use `perf` [6] as a framework for accessing the hardware counters of our CPU. To this end, we implement a wrapper class for the Linux-specific interface which can be seen along with the full setup in the appendix listing 25. In listing 14, we can see the respective file descriptors through which we will be able to access the metric during the benchmarking runs. We instrument the level one and level three caches as well as the branch predictor.

```
long long refs() const { return refs_; }
long long misses()const { return miss_; }

long long l1_refs() const { return l1_refs_; }
long long l1_misses() const { return l1_miss_; }

long long l3_refs() const { return l3_refs_; }
long long l3_misses() const { return l3_miss_; }

long long branches()    const { return br_;    }
long long branch_misses() const { return br_miss_; }
```

Listing 14: The accessors for the wrapper around Linux for hardware counter sampling in `PerfCounters.h`. Even with `T` = 5 we have some headroom by using `long` datatypes.

3.2 Experiment

The experiment driver is configured using an instance file which can be seen in listing 16. We can specify the number of nodes (`n`) we want to allocate, the number of queries we want to run on the tree (`q`), the repetitions of the experiment (`T`) if we want CSV or a table output (`csv`), the seed for the random number generator (`seed`) and if we also want to measure the construction time of the tree for the given implementation by using the `measure_construction`. For all of the presented benchmarks, we have disabled this option, meaning we only measure the metric for the `q` queries. The whole driver can be seen in the appendix listing 28. We also take additional measures to reduce noise, e.g. by preemptions and increase core affinity, as can be seen in appendix listing 34.

```
    for (int t = 0; t < T; ++t) {
        auto tree = make();
        Metrics m = benchOnce(*tree, lookups, inserts,
            measure_construction);

        acc_ns     += m.ns;
        acc_c_refs += m.c_refs; acc_c_miss += m.c_miss;
        acc_l1_refs += m.l1_refs; acc_l1_miss += m.l1_miss;
        acc_l2_refs += m.l2_refs; acc_l2_miss += m.l2_miss;
        acc_l3_refs += m.l3_refs; acc_l3_miss += m.l3_miss;
        acc_br     += m.branches;acc_br_miss += m.br_miss;

        if (t == 0) bytes_used = tree->size_bytes();
    }
```

Listing 15: Main benchmarking in `benchmark.cpp`.

```
{
  "n"   : 1000000,
  "q"   : 1000000,
  "T"   : 1,
  "csv" : false,
  "seed": 123,
  "measure_construction": false
}
```

Listing 16: Quick-run configuration used in unit-test and CI pipelines in `test.json`.

3.3 Testbed

We will run the experiment on an Intel(R) Core(TM) i7-8565U CPU at 1.80GHz. The memory hierarchy of this CPU can be seen in figure 2. 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).

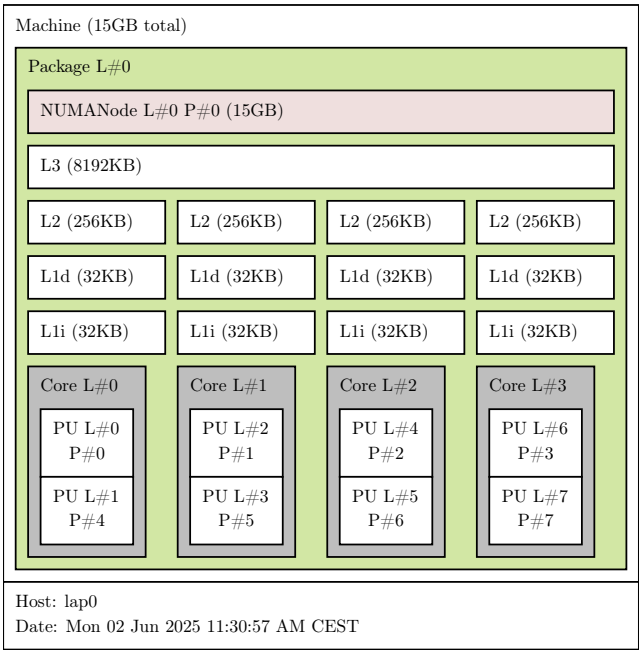

Figure 2: The available testbed memory hierarchy. The four cores have to share the 8MB of available L3 cache.

4 Results

In this section, we will compare the different implementations in terms of cache-friendliness, branch predictability, and overall performance. We only consider the query phase of the implementations and not the overhead to build the trees (`"measure_construction": false`). Since we encountered the if-conversion compiler optimization for the three-level prefetching implementation introduced in section 2.6 and further explained in section 2.6.1, we will use the compiler flags `-fno-if-conversion` and `-fno-if-conversion2` to generate a comparable binary for the standard three-level implementation reported as `BST_EYT_PREF_THREE`. However, we still include the if-converted variant as `BST_EYT_PREF_THREE_IFC`. The probability-biased prefetching implementation (`BST_EYT_PREF_PROB`) uses a total budget of eight prefetches. In figure 3 we can see the average nanoseconds taken for one execution of the `contains` function of the different implementations depending on the number of nodes. Most of the variants take between one and two hundred nanoseconds whereas the most aggressive four-level prefetching implementation (`BST_EYT_PREF_FOUR`) takes around four hundred nanoseconds more on average. This is likely due to the huge pressure on the cache generated by prefetching large amounts of memory lines. We can see that the plain Eytzinger implementation (`BST_EYT`) delivers the best performance for small instance sizes of one to three million nodes. After that, single-prefetch (`BST_EYT_PREF`) and if-converted three-level prefetch variants become faster. A possible explanation for this could be that the cost for the additional index calculations for prefetching may amortize for larger instances. In addition to that, the single-prefetch variant likely loads an adequate amount of lines to the cache when eviction becomes more likely as the total size of the tree increases and hence fills up the level three cache. We will study the evolution of cache misses and evictions in section 4.1. Unlike the standard three-level implementation which probably loads to many lines like other aggressive implementations, e.g. two, three, or four-level prefetch, the if-converted variant can still keep up with the single-level implementation, even though it probably suffers from memory bandwidth contention since it fetches the same amount as the standard three-level implementation. In section 4.2 we will see why branch prediction is a possible explanation for this outcome.

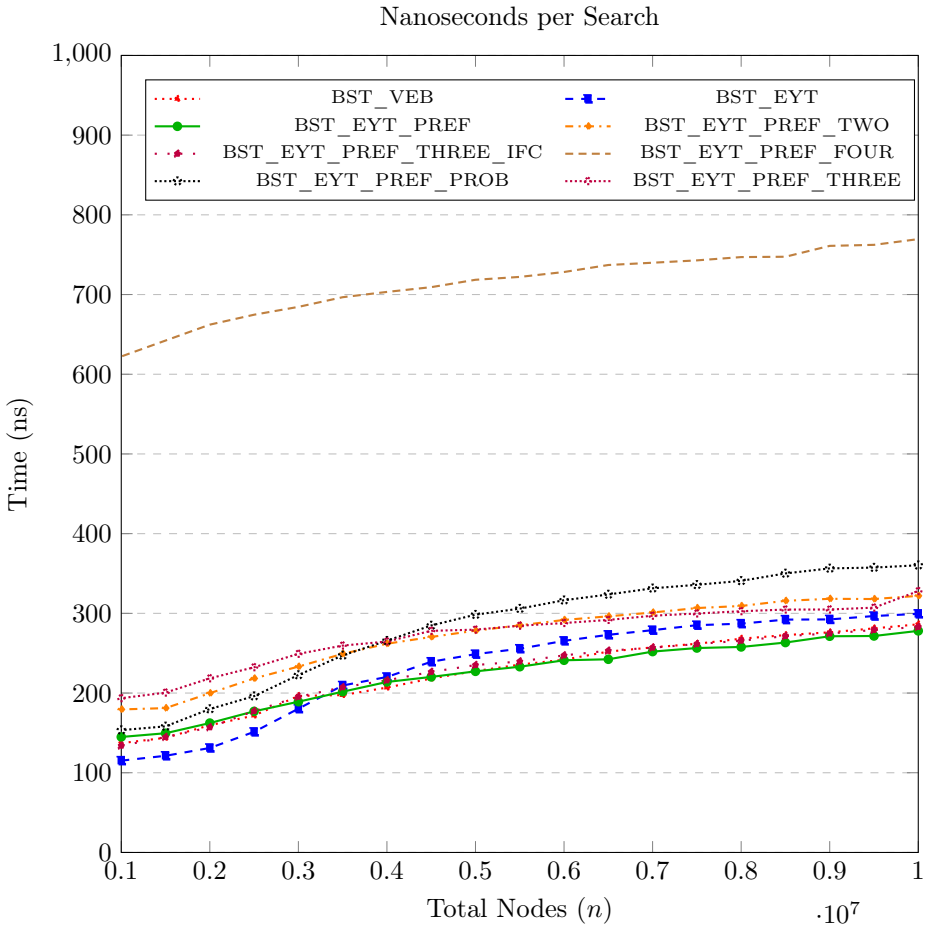

Figure 3: Average Nanoseconds per Search (`contains()`) for the different implementations. We do not take into account the time take to build the respective trees and we always run $q = 1000000$ lookups.

4.1 Cache

In this section, we will compare the implementations regarding their locality properties. In figure 4, we see the overall miss rates of the different versions as a function of n nodes in the tree. We can observe a clear separation between the VanEmde-Boas implementation (`BST_VEB`) and the Eytzinger-based versions. The miss rate of the VEB layout lifts off much earlier and causes a significantly worse miss rate for larger tree sizes. The Eytzinger implementations all maintain a moderate 20-30% miss rate and the aggressive four-level-prefetch variant can even maintain a rate below below twenty percent. The benefit of the Eytzinger layout becomes clear if we compare the plain implementation to the ones that

employ manual prefetching. The layout is so predictable for the CPU-internal prefetchers that all implementations up to three levels of preloading result in comparable miss rates. This is a strong hint that in the case of the Eytzinger layout the microarchitecture of the given CPU gives us most of the benefit of prefetching for free without the need for manual programming. However, the lowest miss rate was still achieved by the four-level implementation, yet, we saw that the overall pressure of loading whole areas of the tree into the cache greatly affects performance, as reported previously in figure 3.

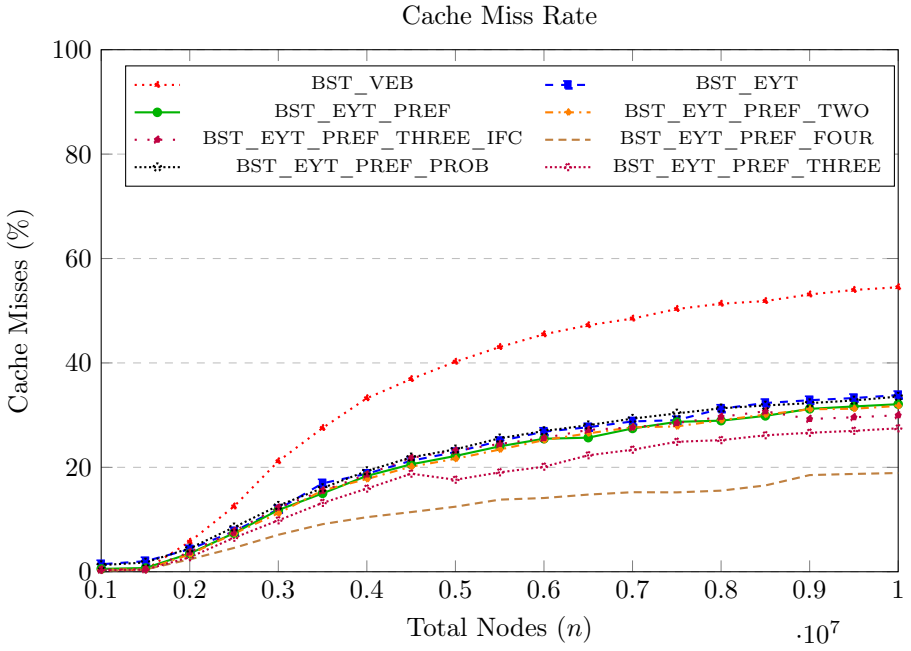

Figure 4: Total Cache miss rate (%) as a function of total nodes and lookups ($q = 1000000$) for the different implementations. A miss in this case means we have to load the value from main memory.

The level-one miss rate seen in figure 5 reveals a different outcome from the total miss results. If we only consider the level-one miss rate, the plain Eytzinger version has to go to higher memory levels significantly more often followed by the VEB implementation. Depending on how aggressively we prefetch an area of the tree, the fewer misses we will cause. Untimely, the four-level version is constantly overwriting large parts of the cache with the area that is currently queried which results in a low miss rate, but high contention and hence negative performance impact.

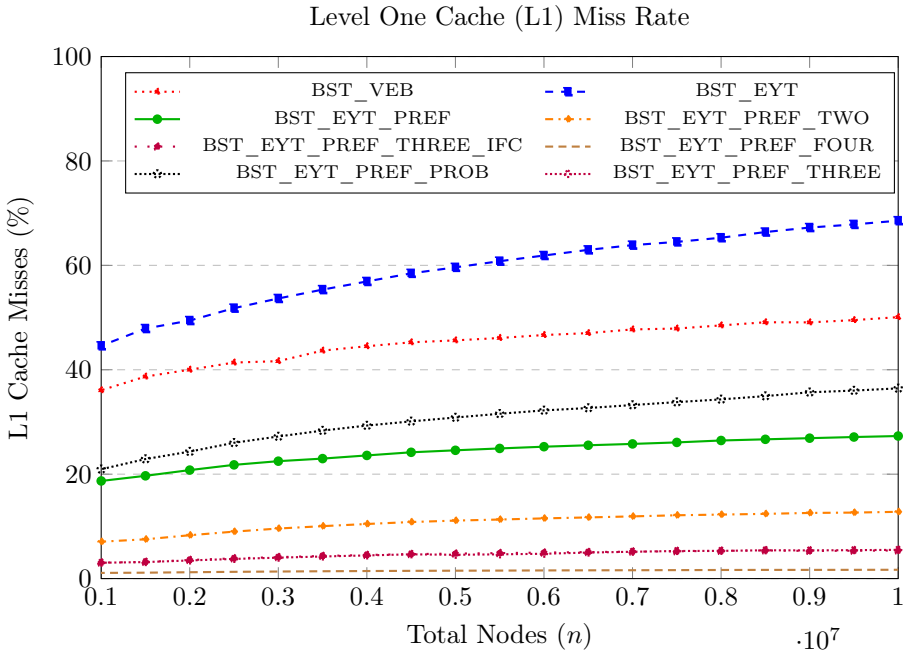

Figure 5: L1 Cache miss rate for the different implementations.

In figure 6, we can see how we can utilize the L3 cache more aggressively by employing manual prefetching. Because of its larger size, we can fit larger areas of the tree into fast cache memory and reduce the miss rate, even if we mostly fetch useless lines with respect to the current query. If we compare the nanoseconds taken from figure 3 with the L3 cache miss rate, we can conclude that a moderate two to three-level prefetch is a good balance between enhanced locality and contention on the memory system.

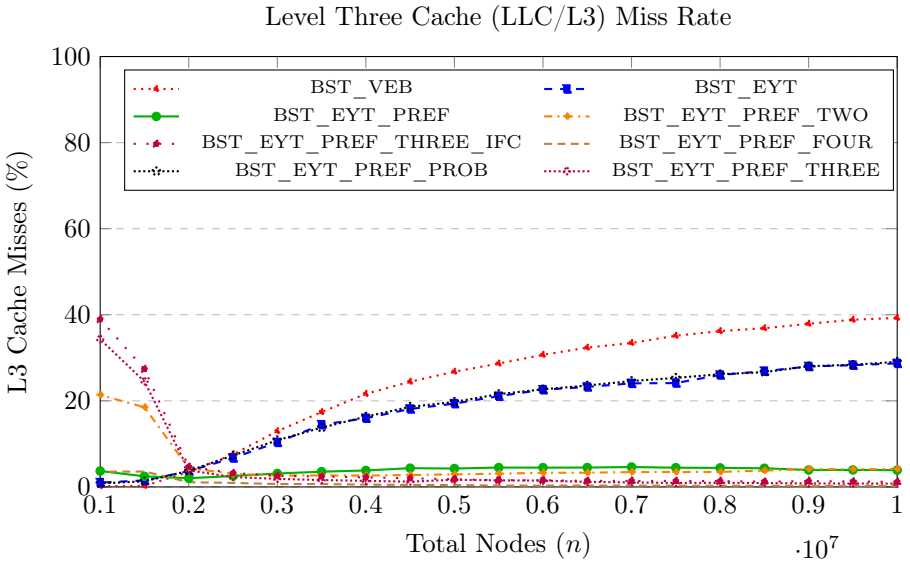

Figure 6: L3 Cache miss rate for the different implementations.

The plots seen in figure 7 compare absolute numbers of full cache misses of the different implementations. In particular, we see the average number of cache misses taken per search operation. It is important to note that these misses also include those triggered by prefetching operations. Hence, the plain Eytzinger version tries to touch the least amount of lines and hence takes the least misses per search on average. Interestingly, the probabilistic implementation closely follows this trend. Once we process about eight to nine million nodes, we see a sudden increase for the four-level prefetching while the other implementations can maintain a shallow development. At this point, the aggressive prefetching may constantly have to evict important sections of the tree, e.g. the root area, upon every traversal which causes this sudden increase. The VEB implementation consistently causes the most misses per search, but also only pulls in memory lines that are actually useful for the current query.

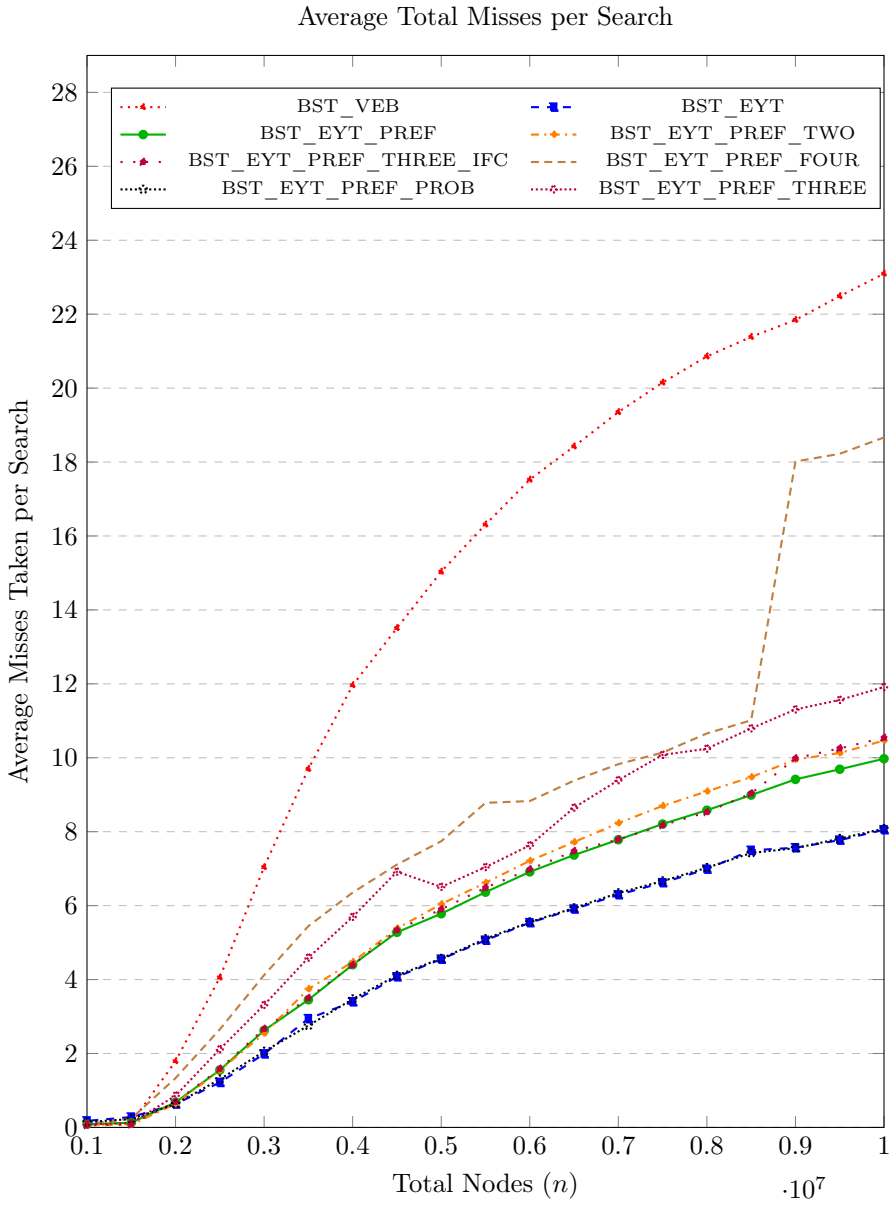

Figure 7: Average Cache Misses Taken Per Search for the different implementations

4.2 Branch Prediction

In this section, we will report the control-flow properties and branch prediction friendliness as well as the effects of predication. We will differentiate between those branches that are easy to predict and hence have no or very little performance impact on the program and those that are basically impossible to consistently predict correctly. In figure 8, we see the total amount of branches that have to be executed for the given implementation. The more lines we want to attempt to prefetch, the more bound checks we have to perform. This is why the four-level implementation executed by far the most branches. The other implementations require near-proportionally less bound checks and hence branches. The two three-level versions are offset by a small margin, likely the branch which was if-converted. Even though the difference is very small, we can observe a massive difference in terms of branch predictability, as can be seen in figure 9.

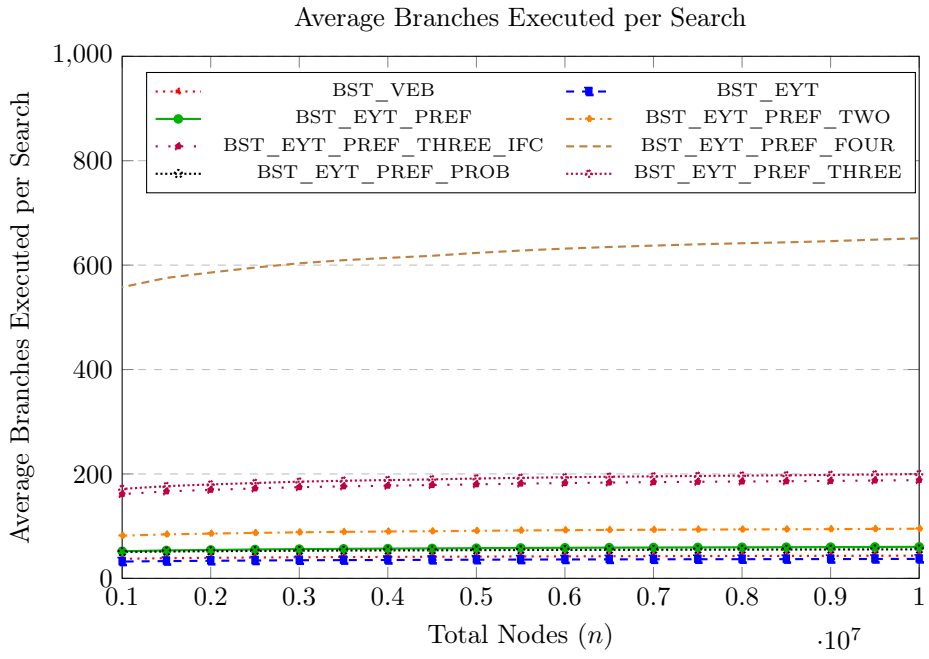

Figure 8: The figure includes the average branches which have to be executed for the given implementation per search operation.

In fact, the if-converted three-level prefetch implementation virtually causes no mispredictions because the critical if-statement was predicated, as we saw earlier in section 2.6.1. For the remaining implementations, we can see how the easy-to-predict bound checks water down the miss rate. Up to this point, the three-level prefetching implementation and its if-converted version showed very similar metrics, e.g. in terms of locality. However, we saw in figure 3 that the predicated version constantly delivers better performance. We can certainly relate this increase in performance to the fact that the predicated version does not suffer from hard-to-predict branches and the resulting performance penalty of branch misprediction.

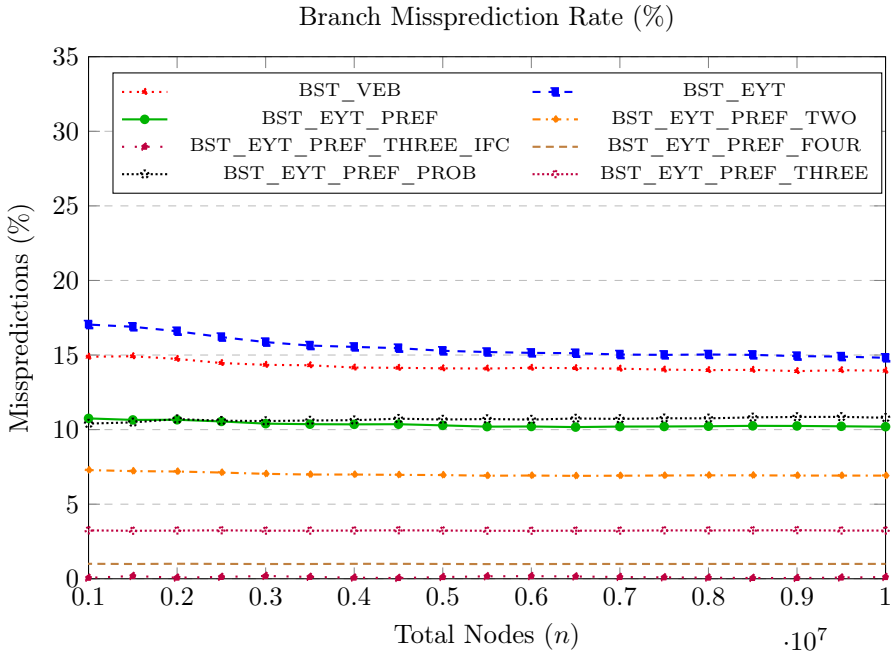

Figure 9: Branch Misprediction Rate in percent of the different implementations.

5 Conclusion

In this project, we implemented and benchmarked different variants of pointer-free binary search trees such as VanEmde-Boas and Eytzinger trees with increasingly aggressive manual data prefetching. We instrumented the program using the `perf` framework to access metrics such as cache misses and branch mispredictions to evaluate the behavior of the different implementations. We surprisingly encountered the *if-conversion* control-flow optimization of the compiler and included it in our analysis. We found that the Eytzinger-based implementations by default enable the CPU-internal prefetchers to pull in more useful data ahead of time, even without explicit prefetching for up to three levels of the tree. We identified that prefetching beyond three levels likely causes too much contention in the memory system and hence degenerates the performance of the program. We categorized the branches of the different programs into easy and hard-to-predict and found that only the final branch which determines whether to go left or right causes problems to the branch predictor. In fact, the if-converted program uses predicated instructions to virtually eliminate any critical branch misspredition during the execution which results in a significant performance increase compared to the branching variant.

6 Appendix

6.1 include/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 17: Tiny pure-virtual interface all BST variants must implement.

6.2 include/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_;


    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 18: Van Emde-Boas layout search-tree implementation.

6.3 include/BSTEyt.h

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

template<class Key>
class BSTEyt : public IBST<Key> {
protected:
    std::vector<Key> arr_;
    static void dedupSort(std::vector<Key>& v) {
        std::sort(v.begin(), v.end());
        v.erase(std::unique(v.begin(), v.end()), v.end());
    }

    void buildEyt(std::size_t idx, std::size_t& pos,
                  const std::vector<Key>& sorted)
    {
        if (idx >= sorted.size()) return;
        buildEyt(2*idx + 1, pos, sorted);
        arr_[idx] = sorted[pos++];
        buildEyt(2*idx + 2, pos, sorted);
    }

    void freeze()
    {
        if (frozen_) return;
        dedupSort(inserts_);
        arr_.resize(inserts_.size());
        std::size_t p = 0;
        buildEyt(0, p, inserts_);
        frozen_ = true;
        inserts_.clear();
        inserts_.shrink_to_fit();
    }

private:
    std::vector<Key> inserts_;
    bool             frozen_ = false;

public:
    void insert(const Key& k) override {
        if (frozen_)
            throw std::logic_error("BST_EYT: insert after first query");
        inserts_.push_back(k);
    }

    bool contains(const Key& k) const override {
```

```
        const_cast<BSTEyt*>(this)->freeze();

        std::size_t i = 0;
        while (i < arr_.size()) {
            if (k == arr_[i])        return true;
            i = (k < arr_[i]) ? 2*i + 1 : 2*i + 2;
        }
        return false;
    }

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

Listing 19: Eytzinger-layout binary-search-tree base class (`BSTEyt`).

6.4 `include/BSTEytPrefetch.h`

```
#pragma once
#include "BSTEyt.h"
#include <cstddef>

template<class Key>
class BSTEytPref : public BSTEyt<Key> {
    using Base = BSTEyt<Key>;

public:
    bool contains(const Key& k) const override {
        const_cast<BSTEytPref*>(this)->Base::freeze();
        const auto& a = Base::arr_;
        std::size_t i = 0;
        while (i < a.size()) {
            std::size_t l = 2*i + 1;
            std::size_t r = l + 1;
            if (l < a.size()) __builtin_prefetch(&a[l], 0, 1);
            if (r < a.size()) __builtin_prefetch(&a[r], 0, 1);

            if      (k == a[i]) return true;
            else if (k < a[i]) i = l;
            else               i = r;
        }
        return false;
    }
};
```

Listing 20: Single-level software prefetch variant.

6.5 include/BSTEytPrefetchTwo.h

```cpp
#pragma once
#include "BSTEyt.h"
#include <cstddef>

template<class Key>
class BSTEytPrefTwo : public BSTEyt<Key> {
    using Base = BSTEyt<Key>;

public:
    bool contains(const Key& k) const override {
        const_cast<BSTEytPrefTwo*>(this)->Base::freeze();
        const auto& a = Base::arr_;

        std::size_t i = 0;
        while (i < a.size()) {
            std::size_t l = 2*i + 1;
            std::size_t r = l + 1;
            if (l < a.size()) __builtin_prefetch(&a[l], 0, 1);
            if (r < a.size()) __builtin_prefetch(&a[r], 0, 1);

            std::size_t ll = 4*i + 3;
            std::size_t lr = ll + 2;
            std::size_t rl = 4*i + 5;
            std::size_t rr = rl + 2;

            if (ll < a.size()) __builtin_prefetch(&a[ll], 0, 1);
            if (lr < a.size()) __builtin_prefetch(&a[lr], 0, 1);
            if (rl < a.size()) __builtin_prefetch(&a[rl], 0, 1);
            if (rr < a.size()) __builtin_prefetch(&a[rr], 0, 1);

            if      (k == a[i]) return true;
            else if (k < a[i]) i = l;
            else               i = r;
        }
        return false;
    }
};
```

Listing 21: Two-level software prefetch variant.

6.6 include/BSTEytPrefetchThree.h

```cpp
#pragma once
#include "BSTEyt.h"
#include <cstddef>

template<class Key>
class BSTEytPrefThree : public BSTEyt<Key> {
    using Base = BSTEyt<Key>;

public:
    bool contains(const Key& k) const override
    {
        const_cast<BSTEytPrefThree*>(this)->Base::freeze();
        const auto& a = Base::arr_;

        auto pf = [&](std::size_t idx) {
            if (idx < a.size()) __builtin_prefetch(&a[idx], 0, 1);
        };

        std::size_t i = 0;
        while (i < a.size()) {

            std::size_t l = 2*i + 1;
            std::size_t r = l + 1;
            pf(l); pf(r);

            std::size_t ll = 2*l + 1;
            std::size_t lr = ll + 1;
            std::size_t rl = 2*r + 1;
            std::size_t rr = rl + 1;
            pf(ll); pf(lr); pf(rl); pf(rr);

            pf(2*ll + 1); pf(2*ll + 2);
            pf(2*lr + 1); pf(2*lr + 2);
            pf(2*rl + 1); pf(2*rl + 2);
            pf(2*rr + 1); pf(2*rr + 2);

            if      (k == a[i]) return true;
            else if (k < a[i]) i = l;
            else               i = r;
        }
        return false;
    }
};
```

Listing 22: Three-level software prefetch variant.

6.7 include/BSTEytPrefetchFour.h

```cpp
#pragma once
#include "BSTEyt.h"
#include <cstddef>

template<class Key>
class BSTEytPrefFour : public BSTEyt<Key> {
    using Base = BSTEyt<Key>;

    static inline void pf(const std::vector<Key>& a, std::size_t idx)
    {
        if (idx < a.size()) __builtin_prefetch(&a[idx], 0, 1);
    }

public:
    bool contains(const Key& k) const override
    {
        const_cast<BSTEytPrefFour*>(this)->Base::freeze();
        const auto& a = Base::arr_;

        std::size_t i = 0;
        while (i < a.size()) {

            std::size_t q[32];
            int front = 0, back = 0;
            q[back++] = i;

            for (int depth = 0; depth < 4; ++depth) {
                int levelCount = back - front;
                for (int n = 0; n < levelCount; ++n) {
                    std::size_t parent = q[front++];
                    std::size_t l = 2*parent + 1;
                    std::size_t r = l + 1;
                    pf(a, l); pf(a, r);
                    q[back++] = l;
                    q[back++] = r;
                }
            }

            if      (k == a[i]) return true;
            else if (k < a[i]) i = 2*i + 1;
            else               i = 2*i + 2;
        }
        return false;
    }
};
```

Listing 23: Four-level software prefetch variant (very aggressive look-ahead).

6.8 include/BSTEytPrefProb.h

```cpp
#pragma once
#include "BSTEyt.h"
#include <cstddef>
#include <array>
#include <algorithm>
#include <cmath>

template<class Key, std::size_t Budget = 8>
class BSTEytPrefProb : public BSTEyt<Key> {
    using Base = BSTEyt<Key>;
    static_assert(Budget >= 2, "Budget must be at least two");
    static inline void pf(const std::vector<Key>& a, std::size_t idx)
    {
        if (idx < a.size()) __builtin_prefetch(&a[idx], 0, 1);
    }

    static void prefetchSubtree(const std::vector<Key>& a,
                                std::size_t          start,
                                std::size_t&         quota)
    {
        if (quota == 0 || start >= a.size()) return;
        std::array<std::size_t, Budget> q{};
        std::size_t front = 0, back = 0;
        q[back++] = start;

        while (front < back && quota) {
            std::size_t parent = q[front++];
            std::size_t l = 2 * parent + 1;
            std::size_t r = l + 1;

            if (l < a.size() && quota) {
                pf(a, l); --quota;
                if (quota && back < q.size()) q[back++] = l;
            }
            if (r < a.size() && quota) {
                pf(a, r); --quota;
                if (quota && back < q.size()) q[back++] = r;
            }
        }
    }
    mutable bool minmax_ready_ = false;
    mutable Key min_key_{};
    mutable Key max_key_{};

    [[gnu::always_inline]] inline void ensureMinMax() const
    {
        if (__builtin_expect(minmax_ready_, 1)) [[likely]]
            return;
```

```
        const auto& a = Base::arr_;
        if (__builtin_expect(a.empty(), 0)) [[unlikely]] return;
        auto [mn, mx] = std::minmax_element(a.begin(), a.end());
        min_key_ = *mn;
        max_key_ = *mx;
        minmax_ready_ = true;
    }

public:
    bool contains(const Key& k) const override
    {
        const_cast<BSTEytPrefProb*>(this)->Base::freeze();
        const auto& a = Base::arr_;
        if (a.empty()) return false;

        ensureMinMax();

        double ratio = (max_key_ == min_key_)
                    ? 0.5
                    : double(k - min_key_) / double(max_key_ - min_key_);
        ratio = std::clamp(ratio, 0.0, 1.0);
        std::size_t spent = 0;
        std::size_t i = 0;
        std::size_t l = 1;
        std::size_t r = 2;

        if (l < a.size()) { pf(a, l); ++spent; }
        if (r < a.size()) { pf(a, r); ++spent; }

        std::size_t remaining = (Budget > spent) ? Budget - spent : 0;
        std::size_t left_budget = std::size_t(std::round(remaining *
            (1.0 - ratio)));
        if (left_budget > remaining) left_budget = remaining;
        std::size_t right_budget = remaining - left_budget;

        prefetchSubtree(a, l, left_budget);
        prefetchSubtree(a, r, right_budget);

        while (i < a.size()) {
            const Key& key = a[i];
            if (k == key) return true;
            i = (k < key) ? 2 * i + 1 : 2 * i + 2;
        }
        return false;
    }
};
```

Listing 24: Probabilistic, path-biased prefetcher (`Budget`-controlled).

6.9 include/PerfCounters.h

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

class PerfCounters {
    int fd_refs_{-1},   fd_miss_{-1};
    int fd_l1_refs_{-1}, fd_l1_miss_{-1};
    int fd_l2_refs_{-1}, fd_l2_miss_{-1};
    int fd_l3_refs_{-1}, fd_l3_miss_{-1};
    int fd_br_{-1},     fd_br_miss_{-1};

    long long refs_{0}, miss_{0};
    long long l1_refs_{0}, l1_miss_{0};
    long long l2_refs_{0}, l2_miss_{0};
    long long l3_refs_{0}, l3_miss_{0};
    long long br_{0},   br_miss_{0};

    static int open(uint32_t type, uint64_t cfg)
    {
        perf_event_attr pea{};
        pea.type         = type;
        pea.size         = sizeof(pea);
        pea.config       = cfg;
        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;
    }
    static int openCache(uint64_t id, uint64_t result)
    {
        uint64_t cfg = id
                    | (PERF_COUNT_HW_CACHE_OP_READ << 8)
                    | (result                     << 16);
        return open(PERF_TYPE_HW_CACHE, cfg);
    }
    static int tryOpenCache(uint64_t id, uint64_t result)
    {
```

```
        try { return openCache(id, result); }
        catch (...) { return -1; }
    }
public:
    PerfCounters()
    {
        fd_refs_ = open(PERF_TYPE_HARDWARE,
            PERF_COUNT_HW_CACHE_REFERENCES);
        fd_miss_ = open(PERF_TYPE_HARDWARE, PERF_COUNT_HW_CACHE_MISSES);

        fd_l1_refs_ = openCache(PERF_COUNT_HW_CACHE_L1D,
                                PERF_COUNT_HW_CACHE_RESULT_ACCESS);
        fd_l1_miss_ = openCache(PERF_COUNT_HW_CACHE_L1D,
                                PERF_COUNT_HW_CACHE_RESULT_MISS);

    #if defined(PERF_COUNT_HW_CACHE_L2)
        fd_l2_refs_ = tryOpenCache(PERF_COUNT_HW_CACHE_L2,
                                   PERF_COUNT_HW_CACHE_RESULT_ACCESS);
        fd_l2_miss_ = tryOpenCache(PERF_COUNT_HW_CACHE_L2,
                                   PERF_COUNT_HW_CACHE_RESULT_MISS);
    #elif defined(PERF_COUNT_HW_CACHE_L2D)
        fd_l2_refs_ = tryOpenCache(PERF_COUNT_HW_CACHE_L2D,
                                   PERF_COUNT_HW_CACHE_RESULT_ACCESS);
        fd_l2_miss_ = tryOpenCache(PERF_COUNT_HW_CACHE_L2D,
                                   PERF_COUNT_HW_CACHE_RESULT_MISS);
    #endif

        fd_l3_refs_ = openCache(PERF_COUNT_HW_CACHE_LL,
                                PERF_COUNT_HW_CACHE_RESULT_ACCESS);
        fd_l3_miss_ = openCache(PERF_COUNT_HW_CACHE_LL,
                                PERF_COUNT_HW_CACHE_RESULT_MISS);

        fd_br_     = open(PERF_TYPE_HARDWARE,
                          PERF_COUNT_HW_BRANCH_INSTRUCTIONS);
        fd_br_miss_ = open(PERF_TYPE_HARDWARE,
                           PERF_COUNT_HW_BRANCH_MISSES);
    }
    ~PerfCounters()
    {
        for (int fd : {fd_refs_, fd_miss_,
                       fd_l1_refs_, fd_l1_miss_,
                       fd_l2_refs_, fd_l2_miss_,
                       fd_l3_refs_, fd_l3_miss_,
                       fd_br_, fd_br_miss_})
            if (fd != -1) close(fd);
    }

    void start() const
    {
        for (int fd : {fd_refs_, fd_miss_,
```

```
                    fd_l1_refs_, fd_l1_miss_,
                    fd_l2_refs_, fd_l2_miss_,
                    fd_l3_refs_, fd_l3_miss_,
                    fd_br_, fd_br_miss_})
            if (fd != -1) {
                ioctl(fd, PERF_EVENT_IOC_RESET, 0);
                ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
            }
    }
    void stop()
    {
        for (int fd : {fd_refs_, fd_miss_,
                    fd_l1_refs_, fd_l1_miss_,
                    fd_l2_refs_, fd_l2_miss_,
                    fd_l3_refs_, fd_l3_miss_,
                    fd_br_, fd_br_miss_})
            if (fd != -1) ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);

        auto rd = [](int fd, long long& dst){
            if (fd != -1 && read(fd, &dst, sizeof dst) != sizeof dst)
                throw std::runtime_error("read perf counter failed");
        };
        rd(fd_refs_, refs_);   rd(fd_miss_, miss_);
        rd(fd_l1_refs_, l1_refs_); rd(fd_l1_miss_, l1_miss_);
        rd(fd_l2_refs_, l2_refs_); rd(fd_l2_miss_, l2_miss_);
        rd(fd_l3_refs_, l3_refs_); rd(fd_l3_miss_, l3_miss_);
        rd(fd_br_, br_);        rd(fd_br_miss_, br_miss_);
    }

    long long refs() const { return refs_; }
    long long misses()const { return miss_; }

    long long l1_refs() const { return l1_refs_; }
    long long l1_misses() const { return l1_miss_; }

    long long l2_refs() const { return l2_refs_; }
    long long l2_misses() const { return l2_miss_; }

    long long l3_refs() const { return l3_refs_; }
    long long l3_misses() const { return l3_miss_; }

    long long branches()    const { return br_;    }
    long long branch_misses() const { return br_miss_; }
};
```

Listing 25: Wrapper around Linux interface for hardware counter sampling.

6.10 Makefile

```
CXX     := g++
CXXFLAGS := -std=c++20 -O3 -march=native -DNDEBUG -Iinclude -Wall -Wextra

SRC  := $(wildcard src/*.cpp)
OBJ  := $(SRC:src/%.cpp=build/%.o)
BIN  := bst-bench

TEST_SRC  := $(wildcard test/*.cpp)
TEST_OBJ  := $(TEST_SRC:test/%.cpp=build/test/%.o)
TEST_BIN  := bst-tests

TEST_CXXFLAGS := $(filter-out -DNDEBUG,$(CXXFLAGS))

.PHONY: all
all: $(BIN)

.PHONY: test
test: $(TEST_BIN)
    ./$(TEST_BIN)

$(TEST_BIN): $(TEST_OBJ)
    $(CXX) $(TEST_CXXFLAGS) $^ -o $@

$(BIN): $(OBJ)
    $(CXX) $(CXXFLAGS) $^ -o $@

build/test/%.o: test/%.cpp
    @mkdir -p $(@D)
    $(CXX) $(TEST_CXXFLAGS) -c $< -o $@

build/%.o: src/%.cpp
    @mkdir -p $(@D)
    $(CXX) $(CXXFLAGS) -c $< -o $@

.PHONY: clean
clean:
    rm -rf build $(BIN) $(TEST_BIN)
```

Listing 26: Build rules: default target builds `bst-bench`; `make test` builds+executes unit tests.

6.11 run_bench.sh

```
set -euo pipefail

usage() {
```

```
  echo "Usage: $0 <base_json> <n_min> <n_max> <step> [--link-q] [--cpu ID] [--nice N]"
  exit 1
}

if [[ $# -lt 4 ]]; then usage; fi

BASE_CFG=$1; shift
N_MIN=$1;  shift
N_MAX=$1;  shift
STEP=$1;   shift

CPU_ID=0
NICE_VAL=-20
LINK_Q=false

while [[ $# -gt 0 ]]; do
  case $1 in
    --link-q) LINK_Q=true; shift ;;
    --cpu)   CPU_ID=$2; shift 2 ;;
    --nice)  NICE_VAL=$2; shift 2 ;;
    *)       usage ;;
  esac
done

echo "# Running on CPU $CPU_ID (nice $NICE_VAL)"
echo "# n from $N_MIN to $N_MAX in steps of $STEP"
$LINK_Q && echo "# q linked to n"

make

IMPLS=(
  "BST_VEB" "BST_EYT" "BST_EYT_PREF"
  "BST_EYT_PREF_TWO" "BST_EYT_PREF_THREE"
  "BST_EYT_PREF_FOUR" "BST_EYT_PREF_PROB"
)

TMP=$(mktemp)
cleanup() { rm -f "$TMP"; }
trap cleanup EXIT

for (( N=N_MIN; N<=N_MAX; N+=STEP )); do
  if $LINK_Q; then
    jq --argjson n "$N" '.n=$n | .q=$n' "$BASE_CFG" > "$TMP"
  else
    jq --argjson n "$N" '.n=$n' "$BASE_CFG"    > "$TMP"
  fi

  for impl in "${IMPLS[@]}"; do
```

```
  nice -n "$NICE_VAL" chrt --fifo 99 taskset -c "$CPU_ID" ./bst-bench
      "$TMP" "$impl"
done
done


# sudo bash scale_bench.sh data/base.json 500000 10000000 500000 --cpu 2
```

Listing 27: Helper script: compiles and launches `bst-bench` for all variants. Sets up the system to give high priority to the benchmark process.

6.12 src/benchmark.cpp

```
#include "IBST.h"
#include "BSTVEB.h"
#include "BSTEyt.h"
#include "BSTEytPrefetch.h"
#include "BSTEytPrefetchTwo.h"
#include "BSTEytPrefetchThree.h"
#include "BSTEytPrefetchFour.h"
#include "BSTEytPrefetchProb.h"
#include "PerfCounters.h"
#include <vector>
#include <random>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <functional>
#include <chrono>
#include "util/json.hpp"

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

struct Metrics {
    long long ns = 0, ops = 0;

    long long c_refs = 0, c_miss = 0;
    long long l1_refs = 0, l1_miss = 0;
    long long l2_refs = 0, l2_miss = 0;
    long long l3_refs = 0, l3_miss = 0;

    long long branches = 0, br_miss = 0;
};

template<class Key>
Metrics benchOnce(IBST<Key>& tree,
                  const std::vector<Key>& lookups,
```

```
                const std::vector<Key>& inserts,
                bool      measure_construction)
{
    for (const auto& k : inserts) tree.insert(k);

    if (!measure_construction && !lookups.empty())
        (void)tree.contains(lookups[0]);

    PerfCounters pc; pc.start();
    auto t0 = Clock::now();
    for (const auto& k : lookups) (void)tree.contains(k);
    auto t1 = Clock::now();
    pc.stop();

    Metrics m;
    m.ns      = std::chrono::duration_cast<std::chrono::nanoseconds>(t1
        - t0).count();
    m.ops     = lookups.size();

    m.c_refs  = pc.refs();     m.c_miss  = pc.misses();
    m.l1_refs = pc.l1_refs();  m.l1_miss = pc.l1_misses();
    m.l2_refs = pc.l2_refs();  m.l2_miss = pc.l2_misses();
    m.l3_refs = pc.l3_refs();  m.l3_miss = pc.l3_misses();
    m.branches = pc.branches(); m.br_miss = pc.branch_misses();
    return m;
}

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

const std::vector<Variant> variants = {
    {"BST_VEB",     [] { return std::make_unique<BSTVEB<int>>(); }},
    {"BST_EYT",     [] { return std::make_unique<BSTEyt<int>>(); }},
    {"BST_EYT_PREF", [] { return std::make_unique<BSTEytPref<int>>(); }},
    {"BST_EYT_PREF_TWO", [] { return
        std::make_unique<BSTEytPrefTwo<int>>(); }},
    {"BST_EYT_PREF_THREE", [] { return
        std::make_unique<BSTEytPrefThree<int>>(); }},
    {"BST_EYT_PREF_FOUR", [] { return
        std::make_unique<BSTEytPrefFour<int>>(); }},
    {"BST_EYT_PREF_PROB", [] { return
        std::make_unique<BSTEytPrefProb<int>>(); }},
};

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

```
80
81      std::vector<int> inserts(n);
82      for (int& x : inserts) x = dist(rng);
83      std::vector<int> lookups(q);
84      for (int& x : lookups) x = dist(rng);
85
86      long long acc_ns = 0,
87                acc_c_refs = 0, acc_c_miss = 0,
88                acc_l1_refs = 0, acc_l1_miss = 0,
89                acc_l2_refs = 0, acc_l2_miss = 0,
90                acc_l3_refs = 0, acc_l3_miss = 0,
91                acc_br = 0,     acc_br_miss = 0;
92
93      std::size_t bytes_used = 0;
94
95      for (int t = 0; t < T; ++t) {
96          auto tree = make();
97          Metrics m = benchOnce(*tree, lookups, inserts,
                measure_construction);
98
99          acc_ns     += m.ns;
100         acc_c_refs += m.c_refs; acc_c_miss += m.c_miss;
101         acc_l1_refs += m.l1_refs; acc_l1_miss += m.l1_miss;
102         acc_l2_refs += m.l2_refs; acc_l2_miss += m.l2_miss;
103         acc_l3_refs += m.l3_refs; acc_l3_miss += m.l3_miss;
104         acc_br     += m.branches;acc_br_miss += m.br_miss;
105
106         if (t == 0) bytes_used = tree->size_bytes();
107     }
108
109     auto avgLL = [T](long long v){ return double(v) / T; };
110
111     double avg_ns     = avgLL(acc_ns), ns_per_op = avg_ns / q, avg_s =
            avg_ns / 1e9;
112     double avg_c_refs = avgLL(acc_c_refs), avg_c_miss =
            avgLL(acc_c_miss);
113     double avg_l1_refs = avgLL(acc_l1_refs), avg_l1_miss =
            avgLL(acc_l1_miss);
114     double avg_l2_refs = avgLL(acc_l2_refs), avg_l2_miss =
            avgLL(acc_l2_miss);
115     double avg_l3_refs = avgLL(acc_l3_refs), avg_l3_miss =
            avgLL(acc_l3_miss);
116     double avg_br     = avgLL(acc_br),    avg_br_miss =
            avgLL(acc_br_miss);
117
118     auto rate = [](double miss, double ref){ return ref ? miss / ref :
            0.0; };
119
120     double miss_per_op = avg_c_miss / q;
121     double miss_rate  = rate(avg_c_miss, avg_c_refs);
```

```
    double l1_rate    = rate(avg_l1_miss, avg_l1_refs);
    double l2_rate    = rate(avg_l2_miss, avg_l2_refs);
    double l3_rate    = rate(avg_l3_miss, avg_l3_refs);
    double br_rate    = rate(avg_br_miss, avg_br);

    double bytes_mb   = bytes_used / 1024.0 / 1024.0;

    if (csv) {
        std::cout << impl << ','
                  << n << ',' << q << ','
                  << avg_ns << ',' << avg_s << ','
                  << ns_per_op << ','
                  << avg_c_refs << ',' << avg_c_miss << ','
                  << miss_per_op << ',' << miss_rate << ','
                  << bytes_used << ','
                  << avg_l1_refs << ',' << avg_l1_miss << ',' << l1_rate
                      << ','
                  << avg_l2_refs << ',' << avg_l2_miss << ',' << l2_rate
                      << ','
                  << avg_l3_refs << ',' << avg_l3_miss << ',' << l3_rate
                      << ','
                  << avg_br << ',' << avg_br_miss << ',' << br_rate
                  << '\n';
    } else {
    std::cout << std::left << std::fixed << std::setprecision(2)
              << std::setw(22) << impl
              << std::setw(8) << n
              << std::setw(8) << q
              << std::setw(14) << std::setprecision(0) << avg_ns
              << std::setw(8) << std::setprecision(2) << avg_s
              << std::setw(14) << std::setprecision(6) << ns_per_op
              << std::setw(12) << std::setprecision(0) << avg_c_refs
              << std::setw(12) << avg_c_miss
              << std::setw(12) << std::setprecision(2) << miss_per_op
              << std::setw(10) << std::setprecision(6) << miss_rate
              << std::setw(6) << std::setprecision(1) << bytes_mb
              << std::setw(12) << std::setprecision(0) << avg_l1_refs
              << std::setw(12) << avg_l1_miss
              << std::setw(10) << std::setprecision(6) << l1_rate
              << std::setw(12) << std::setprecision(0) << avg_l2_refs
              << std::setw(12) << avg_l2_miss
              << std::setw(10) << std::setprecision(6) << l2_rate
              << std::setw(12) << std::setprecision(0) << avg_l3_refs
              << std::setw(12) << avg_l3_miss
              << std::setw(10) << std::setprecision(6) << l3_rate
              << std::setw(12) << std::setprecision(0) << avg_br
              << std::setw(12) << avg_br_miss
              << std::setw(10) << std::setprecision(6) << br_rate
              << '\n';
}
```

```cpp
}

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

    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 (cfg.contains("measure_construction")) measure_construction =
            cfg["measure_construction"];
    }
    if (argc == 3) impl = argv[2];

    if (!csv) {
    std::cout << std::left
              << std::setw(22) << "impl"
              << std::setw(8) << "n"
              << std::setw(8) << "q"
              << std::setw(14) << "total_ns"
              << std::setw(8) << "total_s"
              << std::setw(14) << "ns/search"
              << std::setw(12) << "c_refs"
              << std::setw(12) << "c_miss"
              << std::setw(12) << "miss/sea"
              << std::setw(10) << "miss_rate"
              << std::setw(6) << "MB"
              << std::setw(12) << "L1_refs"
              << std::setw(12) << "L1_miss"
              << std::setw(10) << "L1_rate"
              << std::setw(12) << "L2_refs"
              << std::setw(12) << "L2_miss"
              << std::setw(10) << "L2_rate"
              << std::setw(12) << "L3_refs"
              << std::setw(12) << "L3_miss"
              << std::setw(10) << "L3_rate"
              << std::setw(12) << "branches"
              << std::setw(12) << "br_miss"
```

```
            << std::setw(10) << "br_rate"
            << '\n'
            << std::string(255, '-') << '\n';
}else {
       std::cout << "impl,n,q,total_ns,total_s,ns_per_search,"
                    "cache_refs,cache_misses,misses_per_search,"
                    "miss_rate,bytes,"
                    "l1_refs,l1_misses,l1_rate,"
                    "l2_refs,l2_misses,l2_rate,"
                    "l3_refs,l3_misses,l3_rate,"
                    "branches,branch_misses,branch_rate\n";
    }

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

Listing 28: Main benchmarking driver: parses JSON config, samples perf counters, and prints results.

6.13 test/ContainsTest.cpp

```
#include <cassert>
#include <vector>
#include <iostream>

#include "../include/BSTVEB.h"
#include "../include/BSTEyt.h"
#include "../include/BSTEytPrefetch.h"
#include "../include/BSTEytPrefetchTwo.h"
#include "../include/BSTEytPrefetchThree.h"
#include "../include/BSTEytPrefetchFour.h"
#include "../include/BSTEytPrefetchProb.h"

template<class Tree>
void sanity_check(std::size_t N = 1'024)
{
    Tree t;
    for (std::size_t i = 0; i < N; ++i) t.insert(static_cast<int>(i *
        2));

    for (std::size_t i = 0; i < N; ++i) {
        int present = static_cast<int>(i * 2);
        int absent = present + 1;
```

```
        assert(t.contains(present) && "key should be present");
        assert(!t.contains(absent) && "key should be absent");
    }
}

int main()
{
    sanity_check< BSTVEB<int>        >();
    sanity_check< BSTEyt<int>        >();
    sanity_check< BSTEytPref<int>    >();
    sanity_check< BSTEytPrefTwo<int> >();
    sanity_check< BSTEytPrefThree<int> >();
    sanity_check< BSTEytPrefFour<int> >();
    sanity_check< BSTEytPrefProb<int> >();

    std::cout << "all imps contains() tests passed\n";
    return 0;
}
```

Listing 29: Unit tests verifying that each BST variant correctly answers `contains()` queries.

6.14 `data/large.json`

```
{
  "n"   : 10000000,
  "q"   : 10000000,
  "T"   : 5,
  "csv" : false,
  "seed": 123,
  "measure_construction": false
}
```

Listing 30: Benchmark configuration for the largest test instance (`n`=10 000 000).

6.15 data/medium.json

```
{
  "n"   : 5000000,
  "q"   : 5000000,
  "T"   : 5,
  "csv" : false,
  "seed": 123,
  "measure_construction": false
}
```

Listing 31: Benchmark configuration for the medium-sized instance (n=5 000 000).

6.16 data/small.json

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

Listing 32: Benchmark configuration for the small instance (n=100 000).

6.17 data/test.json

```
{
  "n"   : 1000000,
  "q"   : 1000000,
  "T"   : 1,
  "csv" : false,
  "seed": 123,
  "measure_construction": false
}
```

Listing 33: Quick-run configuration used in unit-test and CI pipelines.

6.18 scale_bench.sh

```
set -euo pipefail

usage() {
  echo "Usage: $0 <base_json> <n_min> <n_max> <step> [--link-q] [--cpu ID] [--nice N]"
  exit 1
}

if [[ $# -lt 4 ]]; then usage; fi

BASE_CFG=$1; shift
N_MIN=$1;  shift
N_MAX=$1;  shift
STEP=$1;   shift

CPU_ID=0
NICE_VAL=-20
LINK_Q=false

while [[ $# -gt 0 ]]; do
  case $1 in
    --link-q) LINK_Q=true; shift ;;
    --cpu)   CPU_ID=$2; shift 2 ;;
    --nice)  NICE_VAL=$2; shift 2 ;;
    *)       usage ;;
  esac
done

echo "# Running on CPU $CPU_ID (nice $NICE_VAL)"
echo "# n from $N_MIN to $N_MAX in steps of $STEP"
$LINK_Q && echo "# q linked to n"

make

IMPLS=(
  "BST_EYT_PREF_THREE_NOIC"
)

TMP=$(mktemp)
cleanup() { rm -f "$TMP"; }
trap cleanup EXIT

for (( N=N_MIN; N<=N_MAX; N+=STEP )); do
  if $LINK_Q; then
    jq --argjson n "$N" '.n=$n | .q=$n' "$BASE_CFG" > "$TMP"
  else
    jq --argjson n "$N" '.n=$n' "$BASE_CFG"    > "$TMP"
  fi
```

```
48
49    for impl in "${IMPLS[@]}"; do
50    nice -n "$NICE_VAL" chrt --fifo 99 taskset -c "$CPU_ID" ./bst-bench
          "$TMP" "$impl"
51  done
52  done
53
54
55  # sudo bash scale_bench.sh data/base.json 500000 10000000 500000 --cpu 2
```

Listing 34: `scale_bench.sh` for running the large benchmark reported.

References

[1] Chit-Kwan Lin and Stephen J. Tarsa. "Branch Prediction Is Not a Solved Problem: Measurements, Opportunities, and Future Directions". In: *arXiv preprint arXiv:1906.08170* (2019).

[2] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665. URL: https://doi.org/10.1109/CGO.2004.1281665.

[3] Paul-Virak Khuong and Pat Morin. *Array Layouts for Comparison-Based Searching*. 2017. arXiv: 1509.05053 [cs.DS]. URL: https://arxiv.org/abs/1509.05053.

[4] GNU Project. *3.12 Options That Control Optimization*. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed: 2025-06-13. GNU Compiler Collection (GCC) Manual, 2025.

[5] David I. August et al. "Integrated predicated and speculative execution in the IMPACT EPIC architecture". In: *Proceedings of the 25th Annual International Symposium on Computer Architecture*. ISCA '98. Barcelona, Spain: IEEE Computer Society, 1998, pp. 227–237. ISBN: 0818684917. DOI: 10.1145/279358.279391. URL: https://doi.org/10.1145/279358.279391.

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