Research Article

Michal Mrena*, Michal Varga, Miroslav Kvaššay, and Marek Klimo

Comparison of various in-order iterator implementations in C++

https://doi.org/10.1515/comp-2025-0043
received March 1, 2025; accepted June 14, 2025

Abstract: Data structures typically use sequential or hierarchical arrangements of elements. Hierarchical data structures are commonly referred to as trees. Trees can be implemented and applied in various ways from simple to relatively sophisticated structures. Undoubtedly, the most well-known tree structure is the binary tree. Such a tree is essential for efficient implementations of (binary) search trees, which typically provide operations like find, insert, and remove. In addition to these, the operation of tree traversal – that sequentially accesses all the elements – is also very important. Data structure libraries mostly implement traversal using the iterator design pattern. The article examines different approaches to implementing an iterator for a binary tree. The article's main contribution lies in an experimental comparison of various iterator implementations. The comparison also includes an AI-generated iterator. The results show that the simple, straightforward approach commonly used in standard libraries is the fastest and that the AI-generated iterator performs reasonably well.

Keywords: AI-assisted programming, binary tree, binary search tree, C++ compilers, in-order iterator, tree traversal

1 Introduction

An important characteristic of any data structure is how the elements and the relationships between them are organized. Based on the multiplicity of these relationships, we can divide the structures into sequential, hierarchical, and network. In the simplest sequential organization, each element has at most one predecessor and at most one successor – the multiplicity of the relationships is 1:1. The hierarchical organization extends this by allowing each element to have zero or more successors. Thus, the multiplicity is 1:N. Such an organization is also known as a tree, which is the term we will use in the rest of the article. The most general organization is the network, which allows for multiple predecessors and multiple successors for each element – such structures are known as graphs. The multiplicity of the relationships is denoted as M:N.

In this article, we focus on the tree structures, which play a pivotal role in various areas of computer science, such as operating systems [1], data structures [2], knowledge representation [3], or cybersecurity [4]. Out of those areas, we focus on the data structures, where one of the key applications lies in the implementation of search trees. Among the search trees, binary trees have a special status. They are used to implement the binary search tree (BST) and its extensions. Such a tree appears in some form in the standard libraries of almost every programming language, regardless of the programming paradigm (from functional [5] to imperative [6]).

BST typically provides operations like find, insert, and remove. These operations can be implemented efficiently, thanks to an internal ordering of the tree. Another important operation is tree traversal. This operation sequentially accesses all the elements of the tree. The ordering of BST allows for a special type of traversal known as in-order traversal. This traversal accesses the elements as an ordered sequence. In object-oriented languages, we usually use the iterator design pattern to implement the traversal. The iterator that implements the in-order traversal is, therefore, known as the in-order iterator. An iterator can be implemented in multiple ways, which differ in the way the current position is maintained and the algorithm for moving to the subsequent position.

Since tree traversal is a frequently used operation, the chosen implementation must be fast and efficient. We identified three principal approaches to the implementation of an iterator. In this article, we focus on an experimental comparison of the selected implementations. We perform the comparison between our implementation of BST and iterators in C++. We compare our implementations with

*** Corresponding author: Michal Mrena,** Department of Informatics, University of Zilina, Zilina, Slovakia, e-mail: michal.mrena@fri.uniza.sk
Michal Varga: Department of Informatics, University of Zilina, Zilina, Slovakia, e-mail: michal.varga@fri.uniza.sk
Miroslav Kvaššay: Department of Informatics, University of Zilina, Zilina, Slovakia, e-mail: miroslav.kvassay@fri.uniza.sk
Marek Klimo: Department of Informatics, University of Zilina, Zilina, Slovakia, e-mail: marek.klimo@fri.uniza.sk

one from the standard library. Since there are three notable implementations of the library, namely, libstd++, libc++, and Microsoft STL, we compared our implementation with each library utilizing the relevant compiler – gcc, clang, and MSVC.

Currently, the use of so-called artificial intelligence (AI) is omnipresent. This elusive term is usually just a placeholder for querying a large generative language model such as ChatGPT. In applied computer science, it is commonly used to generate source code. Students see this tool as an easy shortcut to achieving desired goals, such as solving an assignment or a term project without deeper prior knowledge of the matter. Therefore, we included iterators generated by different generative AI chatbots (chatbot in short) in our iterator comparison to see how they perform in comparison with our implementations.

Our preliminary research, published in a conference article [7], also examined the in-order iterators implementation. However, in the article, we used a rather special implementation of the BST, which uses two levels of abstraction. That implementation is part of a robust data structure framework designed primarily for teaching [8,9]. In this article, we proceed with a simpler, straightforward implementation that resembles those used in practice.

This article is structured as follows. Section 2 contains necessary definitions and descriptions of the properties of tree structures used in the rest of the article. In its second part, it describes BSTs as a specific application of the tree structure. Section 3 focuses on different approaches to the implementation of the traversal of a binary tree. Section 4 describes how we used different chatbot models to generate the iterators and discusses the quality of the generated code. Finally, Section 5 presents an experimental comparison of the approaches described in Section 3. Section 6 is the conclusion.

2 Binary tree

2.1 Tree

A tree is a special type of graph. Thus, the set of all trees is a subset of the set of all graphs. The literature provides multiple equivalent definitions of the tree. In this article, we use the definition that says that a graph is a tree if it is *acyclic* and *connected* – each edge is a *bridge* (removing the edge would cause the graph to split into two *disconnected* subgraphs) [10]. We use the term *node* as a synonym of the vertex since it is a more common name used in the context of trees.

A tree is a hierarchical structure. The edges denote the *parent–son* relationship. A node can have multiple sons but at most one parent. There is at most one node that has no parent, which we call the *root* of the tree. A node that has no sons is called *leaf*. All other nodes are *internal* nodes. The number of sons of a node is also known as its *degree*. Another important property of a node is its *level*, defined as the distance from the root node, i.e., how many times we need to access the parent to reach the root. The root node has a level of zero. A tree is considered *balanced* if the level of all leaves is roughly the same. In Figure 1, we can see an example of a balanced tree with nodes labeled using letters A–F. Node A is the root node. Nodes C, D, E, and F are leaves, and B is an internal node. Node A has degree three, node B has degree two, and the leaves have degree zero. Node A has a level of zero; nodes B, C, and D have a level of one; and nodes E and F have a level of two. *Depth* of a tree is defined as the maximum of levels of its nodes, i.e., a tree consisting of a single node has depth zero.

The tree is a recursive structure – each son of a node and, consequently, each node of a tree – can be considered as the root node of a standalone tree, which we sometimes refer to as *subtree*. This property is important in the design of recursive algorithms that operate on the tree.

The literature classifies trees in several ways using different criteria. One such criterion is the limit in the degree of nodes. Using this criterion, we can classify trees as:

- **multi-way** – there is no limit to the degree of nodes,
- **K-way** – each node has degree at most $K \in \mathbb{N}$,
- **binary** – a special case of K-way tree for $K = 2$.

Though the binary tree is just a special case of the more general K-way tree, it is listed as a separate category. The reason is that it has numerous applications and, therefore, has been studied extensively by computer scientists. In addition to the separate category, we also use specific names when we work with binary trees. Namely, we refer

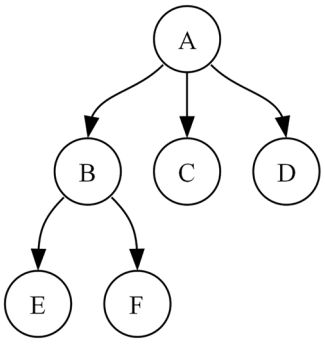

Figure 1: Example of a tree with seven nodes labeled using letters A–F.

to the two sons of a node as *left* son and *right* son. Furthermore, for a balanced binary tree of size n, we can estimate the *depth* as $\log_2(n + 1) - 1$.

2.2 BST

One of the numerous applications of binary trees is the implementation of BSTs. BST [11,12] is an implementation of the Abstract Data Type (ADT) *table*, which is also known as *map* [13], *dictionary* [14,15], or as an *associative array*[1] [16]. The ADT table restricts the domain of the elements by the condition that each element needs to be associated with a unique key. Consequently, its implementations typically store pairs of the form (*key*, *value*). The three main operations defined by the ADT table are summarized in Table 1.

Table 1: Operations defined by the ADT table

Operation	Description
Find (*key*): *value*	Returns *value* associated with *key*
Insert (*key*, *value*)	Inserts the pair (*key*, *value*)
Remove (*key*)	Removes *value* associated with *key*

The table implementations aim to provide an efficient implementation of the above operations. To do so, BST imposes another restriction on the domain – a strict weak ordering on the keys must exist, i.e., the keys must be comparable using the operator <. BST stores the (*key*, *value*) pairs in the nodes and forces a notoriously known restriction on the structure. Let us denote the key stored in node A as KEY(A), the left son of node A as LEFTSON(A), and the right son as RIGHTSON(A). Then, we denote the restriction as follows:

- KEY(LEFTSON(A)) < KEY(A),
- KEY(A) < KEY(RIGHTSON(A)).

This restriction allows for a simple implementation of the FIND operation. In each node A – starting in the root node – we perform one of the following actions:

- if KEY(A) = *key*, return VALUE(A);
- if *key* < KEY(A), repeat the procedure in LEFTSON(A);
- if KEY(A) < *key*, repeat the procedure in RIGHTSON(A);
- if the node does not exist, return failure.

Let us note the recursive nature of the operation, which follows from the recursive nature of the underlying tree structure. Implementation of other operations follows a similar procedure. For details on their implementation, we refer to the existing literature [17].

Figure 2 shows a simple BST with integer keys. The shape of the tree in the figure is just one of numerous possible shapes. Unfortunately, the shape of the tree influences the complexity of the operations. In the best case – when the tree is balanced – the time complexity of the operations is $O(\log n)$, where n is the number of nodes. However, the time complexity can be as bad as $O(n)$ in the worst case. This happens when the tree degenerates to a linked list (which also is a valid binary tree), for example, as a result of inserting keys in ascending or descending order.

This disadvantage is addressed by various extensions of the BST that implement a mechanism that automatically balances the tree after each operation. Examples of such extensions are the Treap [18], AVL tree [2], Splay tree [19], or Red-black tree [20], which are typically implemented in standard libraries of programming languages. Thus, the BSTs that programmers encounter in practice are practically guaranteed to be balanced, thanks to the mentioned extensions.

3 Binary tree traversal

Some operations, such as querying the size of the structure, are common for different ADTs. One such operation is *traverse*. The goal of the operation is to sequentially access each element of the underlying data structure and, typically, perform some operation on the element. Traversal of a tree can be implemented in multiple ways [17]. Since a tree is a graph, the traversals are based on the depth-first search (DFS) and breadth-first search (BFS) traversals from the domain of graph theory [2]. In general (for any tree), the following standard tree traversals exist [17]:

- **pre-order** – process the current node and then proceed with the traversal of the sons,
- **post-order** – traverse all sons and then process the current node,
- **level-order** – process nodes with level 0, then nodes with level 1, all the way to the nodes with the highest level.

In the special case of the binary tree, an additional traversal exists

- **in-order** – traverse left subtree, process current node, and then traverse right subtree.

In Table 2, we can see the results of the mentioned traversal applied to the BST in Figure 2, in which the process

1 On the one hand, the name *array* is a bit misleading; on the other hand, the adjective *associative* describes the table well.

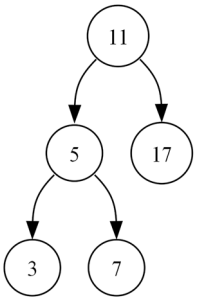

Figure 2: Example of a simple BST with integer keys depicted in the nodes.

operation reads and prints the key stored in the node. Let us note the last row of the table (in-order), which contains keys in ascending order. This is not a coincidence but rather a property of the traversal applied to a tree with BST order – the keys will always be accessed in ascending order (hence the name of the traversal).

3.1 Iterator

The implementation of the traversals in a specific programming language may differ according to the conventions of the language and the programming paradigm used. For example, in functional programming, libraries typically define a function that takes two parameters, namely, a data structure and a function to apply to each element of the data structure [21]. Each type of traversal is implemented as a standalone function.

In object-oriented programming, libraries usually use the iterator design pattern [22]. An iterator is an object that allows efficient access to elements of a data structure without exposing internal details of the data structure. An iterator usually possesses knowledge of the internal organization of a data structure, which allows it to access its elements efficiently. The interface of the iterator differs between different programming languages. However, conceptually, it can be summarized by the operations (methods) presented in Table 3.

Table 2: Results of different traversals applied to BST from Figure 2

Traversal	Order of node processing
Pre-order	11, 5, 3, 7, 17
Post-order	3, 7, 5, 17, 11
Level-order	11, 5, 17, 3, 7
In-order	3, 5, 7, 11, 17

Table 3: Basic operations provided by an iterator

Operation	Description
Init(*root*)	Initializes the iterator for a tree with *root*
CanAdvance(): **bool**	Checks if the iterator can be advanced
Advance()	Moves the iterator to a next element
Access(): *element*	Returns reference to the current element

Many programming languages contain a so-called for-each loop, which conveniently iterates over all elements of a data structure. Such loops are usually just syntactic sugar for an explicit iteration of the structure via iterator [23–25]. Such languages usually define an interface that prescribes a method that returns an iterator of the structure (which is used by the for-each loop). The iterator often also implements an interface prescribing methods similar to those presented in Table 3. Just like with the functional approach, each type of traversal is implemented as a different iterator type. However, since the interface prescribes only a single method, each (iterable) data structure has only one primary type of iterator[2].

As stated at the beginning of this section, BSTs and their extensions work conveniently with the in-order traversal. Consequently, implementations of such trees use an in-order iterator as the primary iterator type. In practice, this means that when we use a for-each loop over, e.g., `std::map` (C++), `TreeMap` (Java), or `SortedDictionary` (.NET), we access the keys in ascending order[3]. The mentioned data structures are part of the standard libraries of the respective languages. Thus, the in-order traversal is arguably the most frequently used traversal. Therefore, we focus solely on this type of traversal in the article. We start by describing different approaches to the implementation of in-order iterators.

3.2 Queued iterator

The first approach that we describe aims at simplicity at the cost of higher memory complexity. It builds on the fact that the functional approach can also be used in imperative languages. In Algorithm 1, we can see a simple recursive implementation of the in-order traversal. The algorithm has two parameters, *node* – the root of the tree to

2 Naturally, it can provide methods that return other iterator types.
3 More precisely, in the ascending order concerning the comparator used to compare the keys. In general, this is true for all data structures based on a BST, such as sets, multisets, and multimaps.

iterate over, and f – a function to apply to each element of the tree.

Algorithm 1. Functional approach to the implementation of in-order traversal.

```
procedure ProcessInOrder (node, f)
    if node ≠ NULL then
        ProcessInOrder (LeftSon (node), f)
        f(node)
        ProcessInOrder (RightSon (node), f)
    end if
end procedure
```

It might seem that the presented algorithm is sufficient and that there is no need for an actual iterator. Indeed, such a simple approach is sufficient in many situations. Especially since most imperative programming languages support lambda functions, which can be conveniently used as the parameter *f*. Containers often provide a method that accepts a function to be applied to all elements, for example, the `forEach` method in Java [26]. However, an iterator has multiple advantages. For example, it allows multiple simultaneous traversals, it can traverse only part of the structure, and, most importantly, its advancement can be programmatically controlled.

The implementation of the iterator is based on the usage of a First-In-First-Out (FIFO) queue. The queue is filled during the creation of the iterator using Algorithm 1 with a simple lambda function that pushes the element into the queue. The other operations of the iterator are then defined as simple operations on the queue. Consequently, the implementation is really simple – it fits into a single table (Table 4). In the table, we can see that the Init operation is the most expensive. On the contrary, Advance and Access operations do not require any navigation in the tree, so the traversal itself should be among the fastest. The actual performance also depends on the data structure used to implement the *queue*. Standard implementations usually use a linked list, a dequeue, or a circular buffer (array). In our implementation, we used `std::vector` with pre-allocated capacity. Finally, the last implementation remark concerns C++ and, possibly, other languages that support value semantics. C++ uses iterators extensively in its standard library and passes them around by value. Therefore, using an iterator with a state of considerable size (the *queue* member variable[4]) would result in many expensive copies. In the worst case, the size of the *queue* is the same as the size of the entire tree. Therefore, such an iterator is practically useless outside of a for-each loop.

Table 4: Implementation of the queued iterator operations – assuming that the iterator has a member variable *queue* – an FIFO queue

Operation	Implementation
Init	ProcessInOrder(*node*, $\lambda x \rightarrow$ Push(*queue*, *x*))
CanAdvance	¬IsEmpty(*queue*)
Advance	Pop(*queue*)
Access	Peek(*queue*)

3.3 Explicit iterator

As we mentioned, one of the disadvantages of the functional approach is that the execution of the ProcessInOrder function cannot be programmatically controlled – the function cannot be "paused" and "resumed" later –, thus, all the elements are visited within a single call of the function. Therefore, the second approach we present is based on the idea of the ProcessInOrder and allows the user to control the next recursive call by calling the Advance operation. Algorithm 1 is short and simple because, in the background, it implicitly uses the process call stack to store current and past positions in the iteration.

The explicit implementation is based on an explicitly maintained *stack* of positions (member variable of the iterator). The position is represented by a structure that matches the call stack frame of the ProcessInOrder function. Table 5 contains fields of the structure, their types, and get and set functions used in pseudocode. Algorithm 2 contains pseudocode describing the implementation of the operation Advance. The code uses two auxiliary operations tryGoLeft and tryGoRight, which are also presented in the pseudocode. Furthermore, the code also uses operations pushPosition and popPosition to maintain the *stack* of positions.

In our implementation, we use an intrusive linked list to implement the stack. Our structure described in Table 5 has one additional field – a pointer to an instance of the previous position. This is represented using the *stack* member variable in the pseudocode. Another option is to use a stack implementation from the standard library, which typically uses a linked list, an array list, or a deque. Just like with the queued iterator, in C++, we also need to consider the cost of copying the *stack* when we pass the

4 Member variables are also known as fields, attributes, or properties in some programming languages.

Table 5: List of fields of the structure representing a position in the iteration of a binary tree; the ↑ symbol represents a pointer

Field	Type	Get	Set
isProcessed	Boolean	IsProcessed	SetIsProcessed
node	↑ node	Node	SetNode
son	↑ node	Son	SetSon
sonOrder[a]	integer	SonOrder	SetSonOrder
next[a]	↑ position	—	—

[a]Value 0 represents left son, 1 right son, and −1 an invalid value and [b] Used in our intrusive list implementation. Used by the PUSHPOSITION and POPPOSITION helper functions.

iterator by value. Fortunately, the size of the stack is considerably limited – it is constrained by the depth of the tree, which, assuming that the tree is balanced, can be calculated as $\lceil \log_2(n+1) \rceil$.

Algorithm 2 Implementation of the ADVANCE operation of the explicit iterator.

```
Used member variables: stack, position
procedure ADVANCE()
    if ¬ ISPROCESSED (position) then
        if SONORDER (position) ≠ 0 ∧
            TRYGOLEFT (position) then
            PUSHPOSITION (stack, SON (position))
            ADVANCE ()
        end if                          ▷ Process left subtree.
    else
        if SONORDER (position) ≠ 1 ∧
            TRYGORIGHT (position) then
            PUSHPOSITION (stack, SON (position))
            ADVANCE ()
        else                            ▷ Process right subtree.
            POPPOSITION(stack)
            if positon ≠ NULL then
                ADVANCE ()
            end if
    end if          ▷ Current subtree processed. Backtrack
    end if
end procedure
procedure TRYGOLEFT (position)
    SETSON (position, LEFTSON (NODE(position)))
    if SON (position) ≠ NULL then
        SETSONORDER (position, 0)
        return True
    else
        SETSONORDER (position, −1)
        return False
    end if
end procedure
procedure TRYGORIGHT (position)
    SETSON (position, RIGHTSON (NODE(position)))
    if SON(position) ≠ NULL then
        SETSONORDER (position, 1)
        return True
    else
        SETSONORDER(position, −1)
        return False
    end if
end procedure
```

3.4 Stateless iterator

The last approach that we present does not require any auxiliary data structure. It maintains the current position in the form of a single pointer to the *current* tree node. Hence, we refer to it as *stateless*. The main logic of the iteration is implemented in the ADVANCE operation, and it requires a simple initialization procedure in the INIT operation. The pseudocode of these two operations can be found in Algorithms 4 and 3, respectively. Let us note that this exact algorithm is used in the implementation of the Red-black tree in libc++ standard library [27], MS STL standard library [28], and a similar algorithm is used in the libstdc++ standard library [29] of the C++ language. However, there is one a notable difference between our implementation of the INIT operations and analogous implementations from standard libraries – our INIT starts at the root node and searches the leftmost node (with $O(\log n)$ complexity) while the standard implementations track the leftmost node explicitly and, thus, create the iterator with $O(1)$ complexity.

Algorithm 3 Implementation of the INIT operation of the stateless iterator.

```
Used member variables: current
procedure INIT (root)
    current ← root
    next← LEFTSON (current)
    while next≠ NULL do
      current ← next
      next← LEFTSON (next)
    end while
end procedure
```

Algorithm 4 Implementation of the ADVANCE operation of the stateless iterator.

```
Used member variables: current
procedure ADVANCE ()
    right← RIGHTSON(current)
    if right≠ NULL then
        current ← right
        next← LEFTSON (right)
        while next≠ NULL do
            current ← next
            next← LEFTSON (next)
        end while
    else
        while ISRIGHTSON (current)
            current← PARENT (current)
        end while
        current← PARENT (current)
    end if
end procedure
```

4 AI-generated iterator implementation

The use of AI in programming has experienced a major boom in recent years. Development environments now directly integrate extensions that provide sophisticated code completion – not just based on identifiers but on the assumed intent of the programmer. Moreover, it is now possible to use various tools to generate code purely based on a verbal description. These tools are, naturally, widely used by students. The original motivation to explore different approaches to iterator implementation arose from a course in which we addressed this topic. Therefore, we decided to include iterators generated by chatbots in the comparison to see how they perform compared to human-written implementations.

4.1 Query

When writing the query for the chatbots, we avoided specifying iterator properties beyond the fact that we wanted an in-order iterator. This is because we did not want to affect the form of the generated iterator. The goal was to write the query similarly to how a student would write it. Our query for chatbots, therefore, looked like the following:

> Provide an implementation of an in-order iterator in C++ for a BST, which has the following node structure:

```
struct Node {
    Node(const K& key, const T& data);
    std::pair<K, T> m_data;
    Node* m_parent;
    Node* m_left;
    Node* m_right;
};
```

Figure 3: C++ structure representing node of BST.

and was followed by the structure of our node, which can be seen in Figure 3. We chose to query four popular chatbots, namely, ChatGPT, Gemini, Copilot, and DeepSeek [30–33].

4.2 Iterators in C++

Before we proceed with the description of the iterators generated by chatbost, let us briefly describe how iterators are implemented in C++ according to its standards to make the following section more clear. C++ does not have an interface (in the sense of virtual methods) that must be implemented by an iterator. Instead, any class that overloads a prescribed set of operators with certain behavior and provides prescribed member types can be used as an iterator. We can see a list of selected operators and the corresponding iterator operation in Table 6.

4.3 AI iterator

All chatbots responded to us with a fundamentally identical implementation of the in-order iterator. They differed only in details, which we describe later. Henceforth, in the article we will describe only one implementation of the iterator, which we will refer to as the *AI iterator*[5]. Concerning the implementation, the AI iterator comes closest to the implementation of the *explicit iterator* we described in Section 3.3. The

5 We want to to emphasize that the name does not imply that the iterator uses AI to traverse the tree, the AI part only indicates that the iterator was generated by AI.

implementation of the key operations Init and Advance of the *AI iterator* can be seen in Algorithms 5 and 6, respectively.

The implementation is built on a *stack* of nodes that is used to store the current position and previous positions. All generated implementations used `std::stack` from the standard library, which internally uses a deque to store the elements. Similar to the *explicit iterator*, when working with the *AI iterator*, we need to consider the impact of *stack* copying when we pass the iterator by value. Here, too, however, the stack *size* is considerably limited.

Algorithm 5 Implementation of the Init operation of the AI iterator.

```
Used member variables: stack
procedure Init (root)
  node ← root
  while node≠ NULL do
    Push (stack, node)
    node← LeftSon (node)
  end while
end procedure
```

Algorithm 6 Implementation of the Advance operation of the AI iterator.

```
Used member variables: stack
procedure Advance ()
   node← Pop (stack)
   right← RightSon (node)
   while right≠ NULL do
     Push (stack, right)
     right← LeftSon(right)
   end while
end procedure
```

Table 6: Operators in C++ and corresponding language-agnostic iterator operations

Operation	Operator
CanAdvance()	`operator!=`
	`operator==`
Advance()	`operator++`
AdvanceInOppositeDirection()	`operator-`
Access()	`operator*`

4.4 Summary

Each chatbot generated code that was functional and able to compile. However, each had minor C++ language-specific flaws and, in some cases, minor logic errors in some operations.

ChatGPT generated a valid C++ iterator. However, it introduced a bug in the `operator!=`. The issue was that it only checked if at most one of the *stacks* (of two iterators compared) was empty. Such implementation works correctly when we just use the iterators to traverse the entire tree (for example, by using the for-each loop), but would fail in a general comparison of iterators. After pinpointing the issue, ChatGPT provided the correct implementation.

Gemini provided a more extensive implementation of an in-order iterator. In addition to the `operator++`, it also provided (without explicitly asking for it to be generated) an implementation of `operator-`. Furthermore, it also generated the correct member types required by the C++ standards by an iterator. Also, in contrast with ChatGPT, it provided the correct implementation of the `operator!=`.

Copilot provided an interesting answer. The code it generated was almost identical to the code generated by ChatGPT. However, surprisingly, it used different names for the operations. Instead of overloding `operator!=` and `operator++`, it used operations named `hasNext()` and `next()`, which are used in the Java programming language. However, at least it generated the `hasNext()` (equivalent of the `operator!=`) correctly. After pointing out this issue, the Copilot provided the correct answer.

Deepseek was the last chatbot that we tried. The code provided was almost identical to the one provided by ChatGPT. Like ChatGPT, it also generated `operator!=` that would not work correctly in general. Once again, after pointing out the issue, the correct implementation was provided.

In summary, each chatbot provided a working and (almost) correct in-order iterator implementation. However, only Gemini generated a class that fulfilled all the C++ iterator requirements. Naturally, we assume that the other chatbots would also be capable of generating a complete iterator implementation (from the C++ language point of view) after explicitly asking for it. However, this requires a deeper knowledge of the chatbot from the user. Unfortunately, students or programmers coming from another programming language may not possess the required knowledge, and consequently, they may not know what to ask for. Also, the bug introduced to `operator-` by two chatbots is a bit concerning because it does not show up in normal use but may show up later in specific use of the iterator. Finally, an interesting question

Table 7: Names of iterator implementations used in the presentation of the results

Iterator name	Section
Stateless	Section 3.4
Explicit	Section 3.3
AI	Section 4.3
Queued	Section 3.2

is how the AI-generated implementation will perform compared to the ones written by humans. We provide the results of our comparison in the following section.

5 Experimental comparison

So far, in this article, we have described various approaches to the implementation of an in-order iterator, including an iterator generated by a chatbot. We proceed with an experimental comparison of all the described implementations. Table 7 contains the names of the iterators we use in the presentation of the results, along with references to the sections in which their implementation is described. The source code, along with the results of the experiment, is available online[6].

5.1 Experiment setup

In practice, programmers use one of the mentioned extensions of the BST, which means that the trees they work with are well-balanced. Therefore, in the experiment, we focused only on the traversal of balanced trees. For simplicity, we implemented a plain BST with the node type shown in Figure 3. To construct a balanced tree with n nodes, we first generate a sequence of the form $[0, 1, \ldots, n-1]$. Then, we inserted the median of the sequence, split the sequence in half, and repeated the procedure on each half until the halves consisted of a single number. Using this procedure, we constructed a tree for different values of n, specifically, for $n = 2^k - 1$ for $k = 1, 2, \ldots, 25$. The given values gave us a perfectly balanced tree (also known as a perfect binary tree) with depth $k - 1$. We believe that a BST generated in this way closely resembles, for example, a Red-black tree (which guarantees logarithmic complexities) and that it is a good-enough approximation of a tree structures that emerge in practice.

6 https://gitlab.kicon.fri.uniza.sk/varga02/itercomp.

Algorithm 7 Replication evaluating the speed of a particular iterator type.

```
procedure RunReplication (n)
    bst← GenerateBST (n)
    StartStopwatch ()
    iterator← CreateIterator(Root(bst)
    while CanAdvance(iterator) do
        Access (iterator)
        Advance (iterator)
    end while
    StopStopwatch ()
end procedure
```

The C++ language is implemented in multiple compilers. The three most notable ones are GCC, clang, and MSVC. In addition, each of those compilers comes with its own implementation of the standard library. Therefore, we decided to run the experiments with each compiler and its standard library. The summary of the compilers and standard libraries used is given in Table 8. The experiments were performed on a PC with an Intel i9-10900KF processor with 128 GB of DDR4 RAM running the Void Linux operating system (for gcc and clang) and Windows Education (for MSVC). For each value of n, we measured the average time required to iterate over all tree elements (including time to initialize the iterator with the Init operation) using different iterators. The averages were obtained by 100 replications for better accuracy. Algorithm 7 presents a summary of the code for single replication. Furthermore, we wanted to assess the quality of our implementation. Thus, in addition to our BST, we conducted the same experiment using `std::map` (which is implemented as a Red-black tree) of the same size, containing the same elements. Naturally, the exact shape of the Red-black tree may differ from our balanced BST. However, since the Red-black tree also guarantees the balance of the tree, we may consider them isomorphic (approximately).

5.2 Results and discussion

We present the results of the comparison in the form of line charts. In the charts, the x-axis displays the depth of

Table 8: Versions of the compilers and standard libraries used in the experiment

Compiler	Version	Standard library
gcc	13.2.0	libstdc++
clang	19.1.4	libc++
MSVC	19.43.34808	MS STL

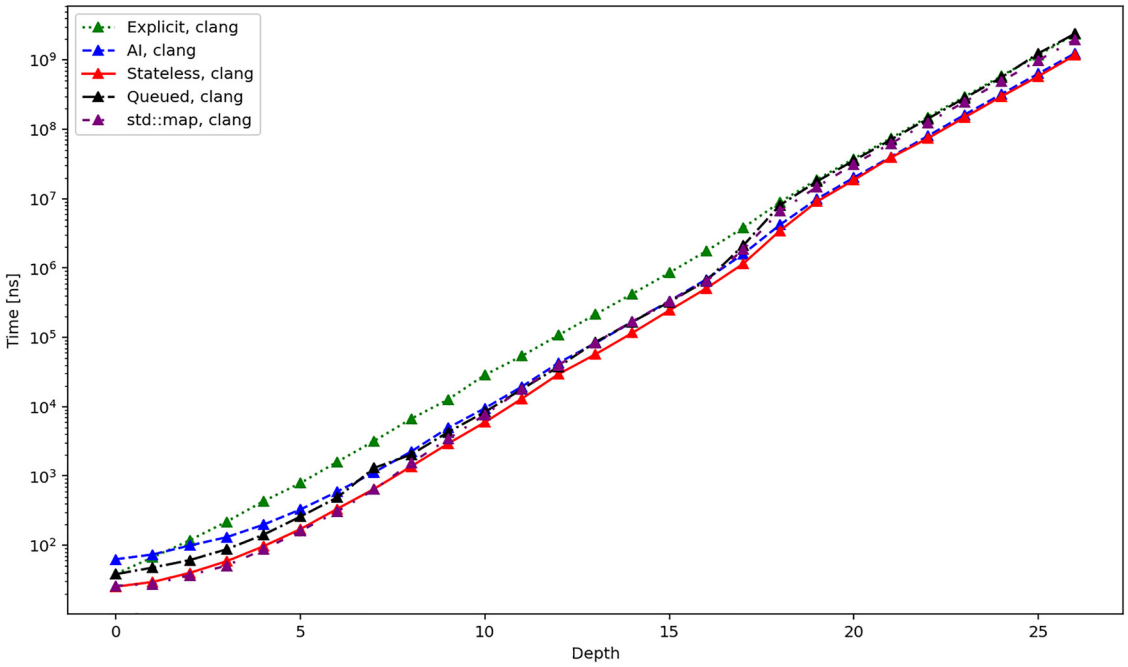

Figure 4: Average total duration in nanoseconds required to create and iterate over all elements of a tree using different iterators and the clang compiler.

the perfectly balanced tree. Therefore, it is scaled linearly in the depth and *logarithmically* in the number of nodes n. The y-axis displays nanosecond duration and is scaled *logarithmically*. The duration presented in each chart is an average value obtained from 100 replications.

Figures 4–6 show the average total time required to create an iterator and iterate over all elements of the tree using different compilers. The results are quite similar for the GCC and clang compilers. Up until depth 16, all the iterators perform roughly the same, except the *explicit*

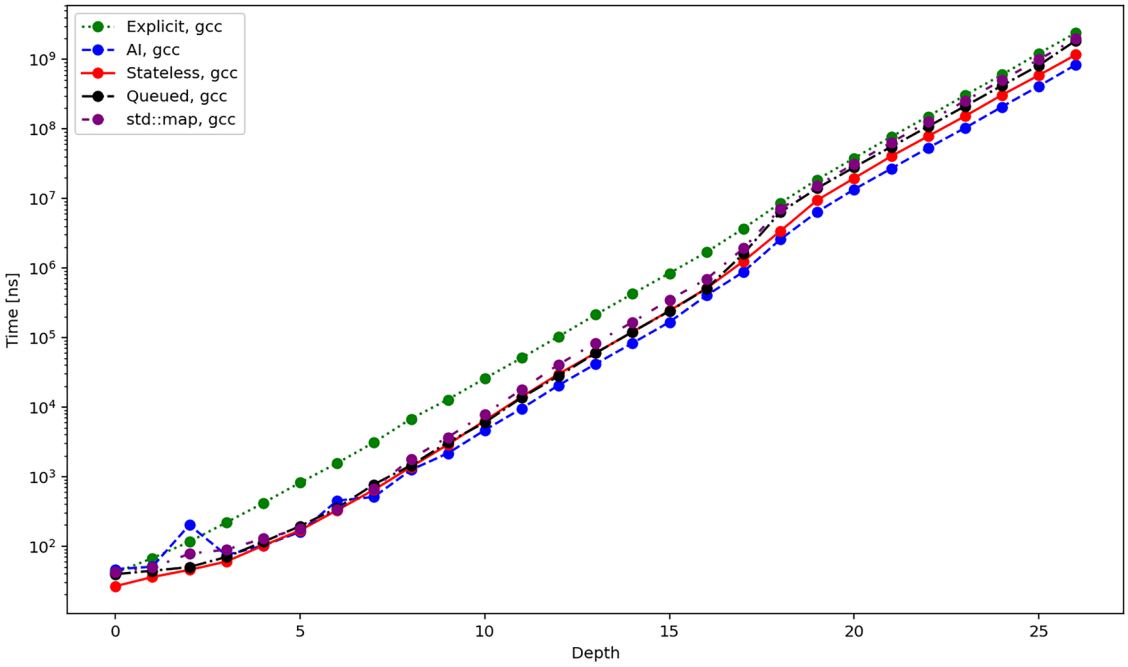

Figure 5: Average total duration in nanoseconds required to create and iterate over all elements of a tree using different iterators and gcc compiler.

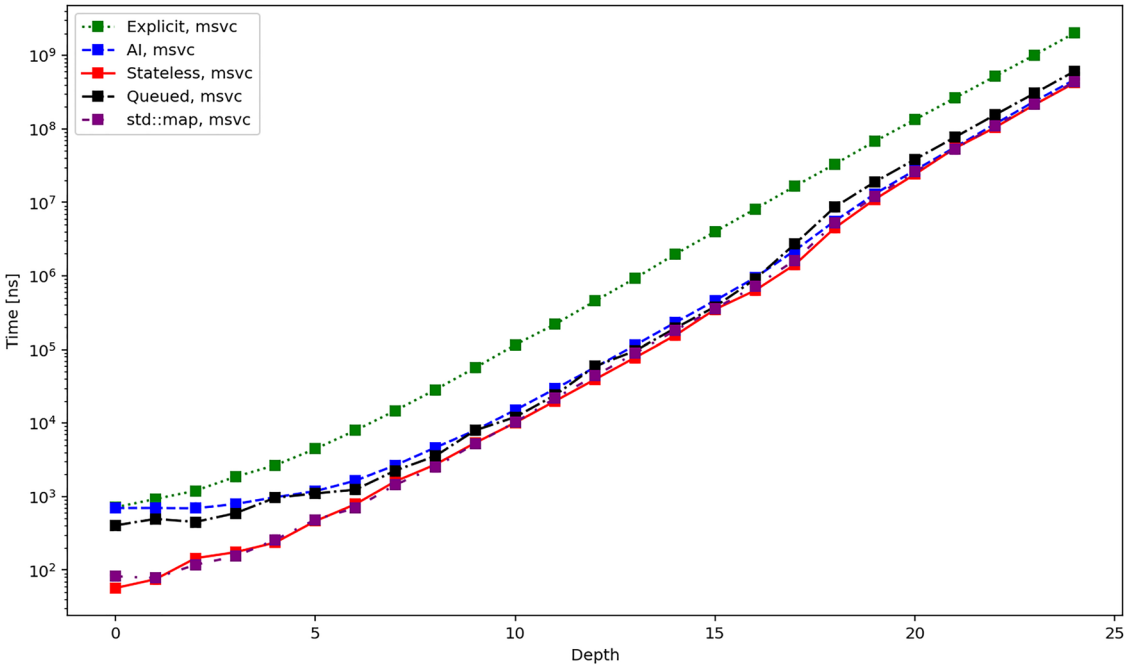

Figure 6: Average total duration in nanoseconds required to create and iterate over all elements of a tree using different iterators and MSVC compiler.

iterator, which is consistently slower. From this point on, the performance becomes more similar, while the *AI* and *stateless* iterators are slightly faster than the others. However, let us remind the logarithmic scale of the y-axis, which means that the real difference in the performance is more significant. The MSVC compiler shows slightly different results, mainly in the performance of the explicit *iterator*, which is consistently the slowest for all depths of the tree. However, the performance of the other iterators is almost identical, especially for higher depths of the tree.

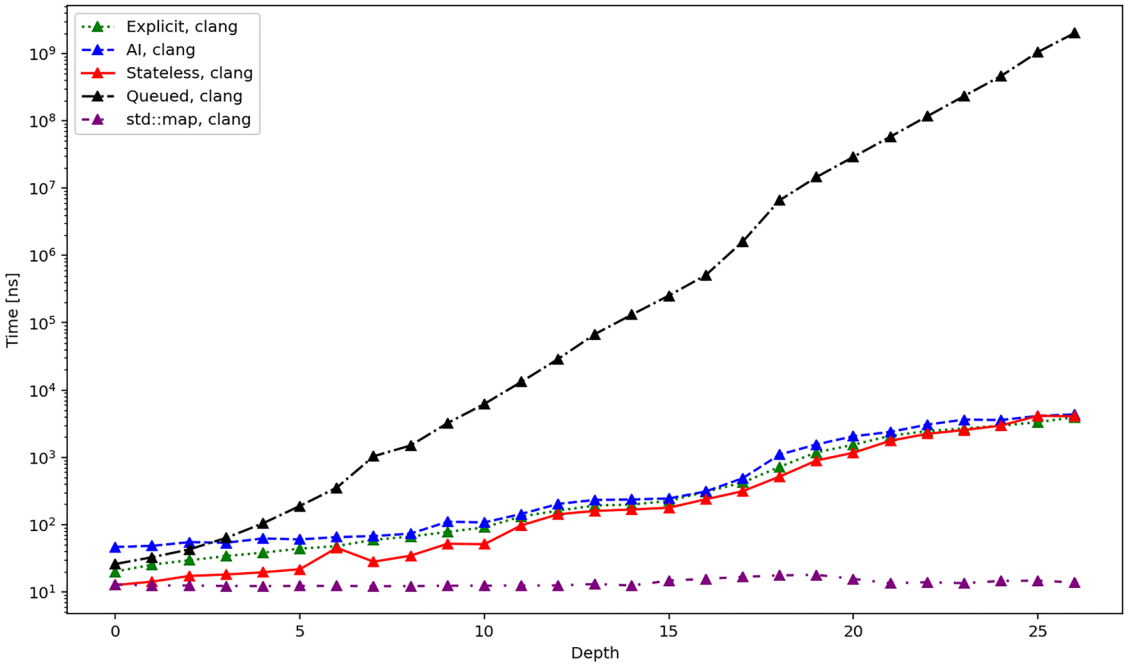

Figure 7: Average total duration in nanoseconds required to create different iterators when using clang compiler.

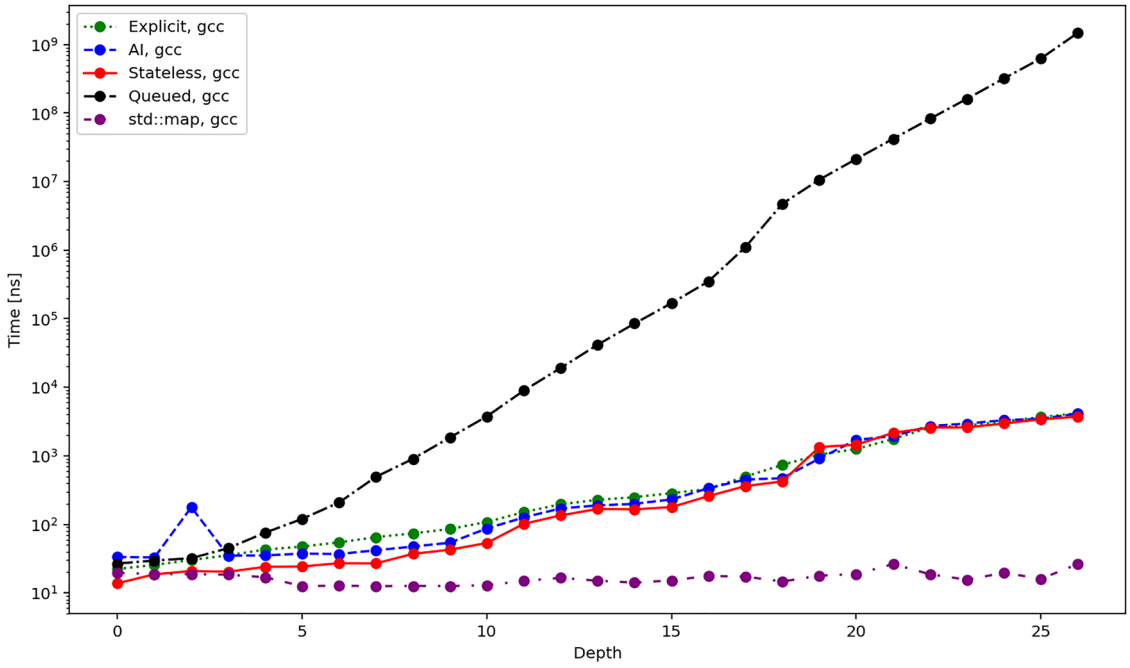

Figure 8: Average total duration in nanoseconds required to create different iterators when using gcc compiler.

Figures 7–9 present the average time required to Init an iterator. These results are more influenced by the specifics of the particular implementation. First, we can see that the time grows linearly (considering the logarithmic scale of the *y*-axis, the real dependency on the depth is exponential) with the depth for the *queued* iterator. This agrees with the expectation since this iterator needs to traverse the entire tree in the Init operation. It is interesting to observe that the complexity of the Init operation of the *std::map* iterator is constant. However, this also agrees with the expectation since all three tested standard libraries (Table 8) use an optimization, which remembers

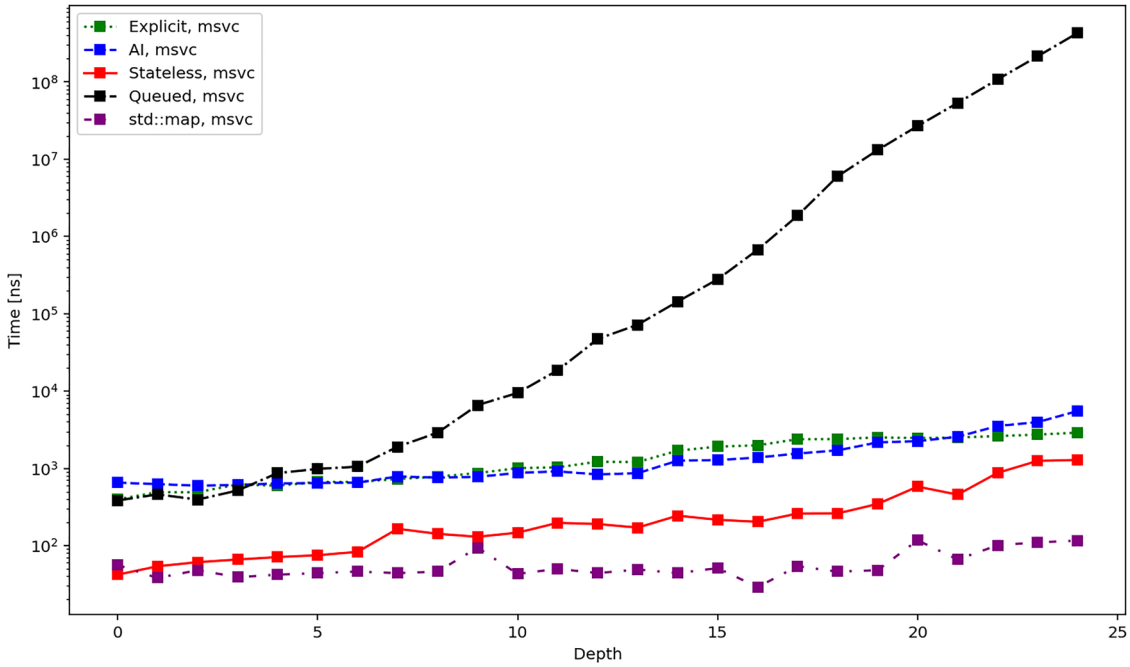

Figure 9: Average total duration in nanoseconds required to create different iterators when using MSVC compiler.

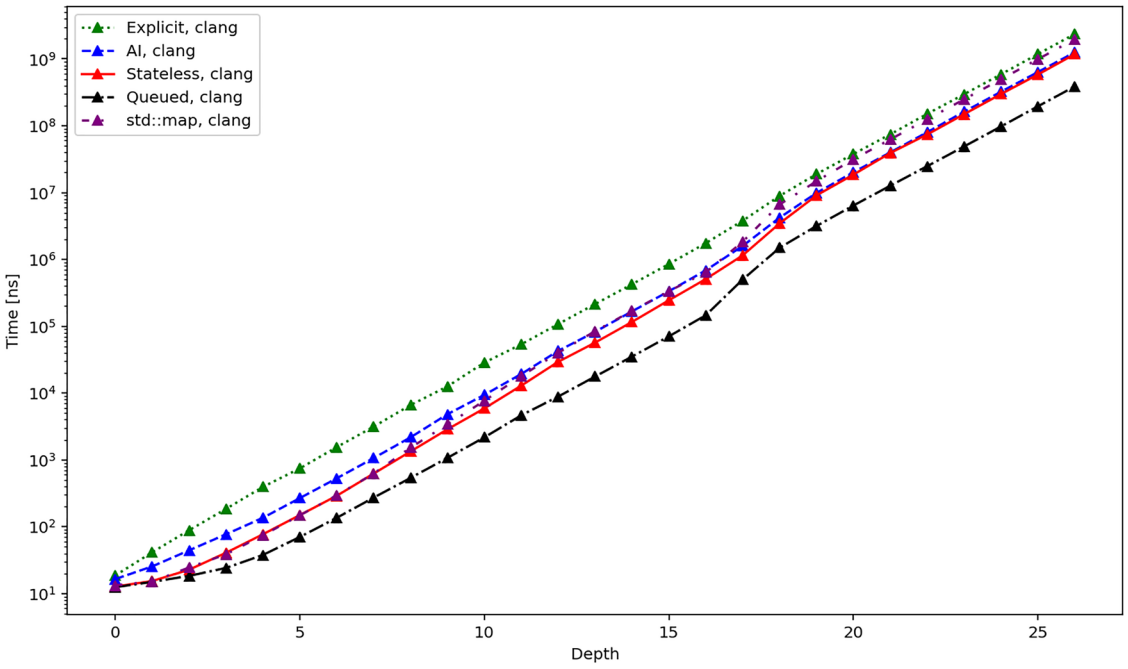

Figure 10: Average total duration in nanoseconds required to iterate over all elements of a tree using different iterators and clang compiler.

the leftmost node and, thus, all three implementations are able to initialize the iterator in constant time. The observed growth of the duration for the other iterators should be logarithmic (the real dependency is linear in the depth) since they need to visit all levels of the tree in their construction. This agrees with the results, especially for GCC and clang.

Figures 10–12 show the average time required to traverse the tree. These figures are similar to the figures in Figures 4–6. This once again agrees with the expectation

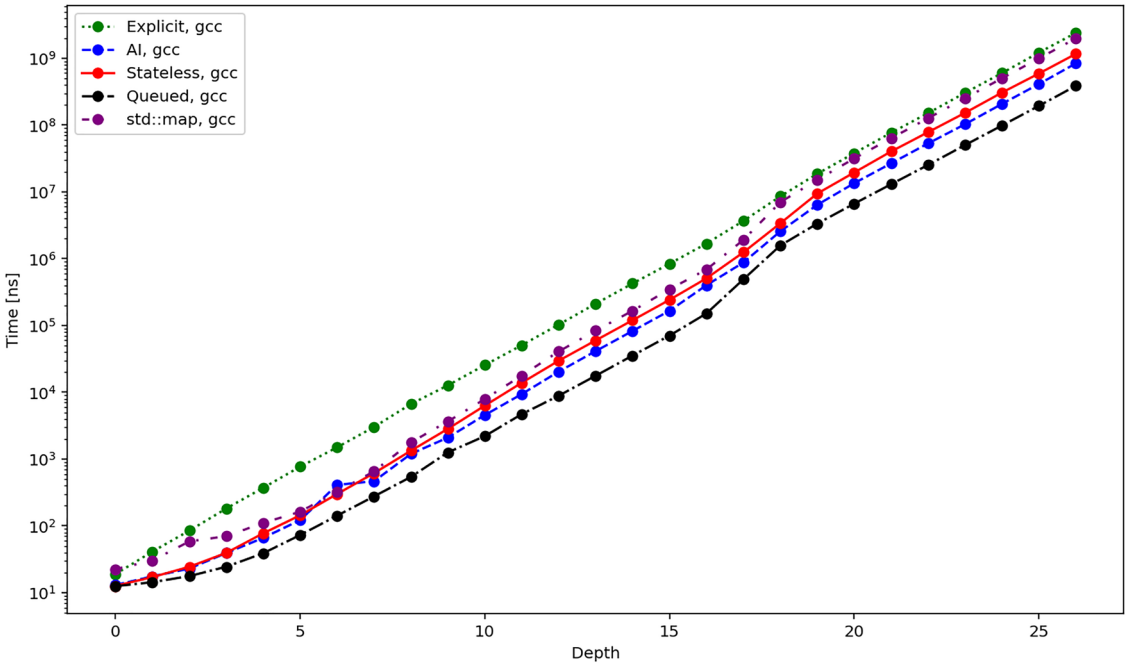

Figure 11: Average total duration in nanoseconds required to iterate over all elements of a tree using different iterators and gcc compiler.

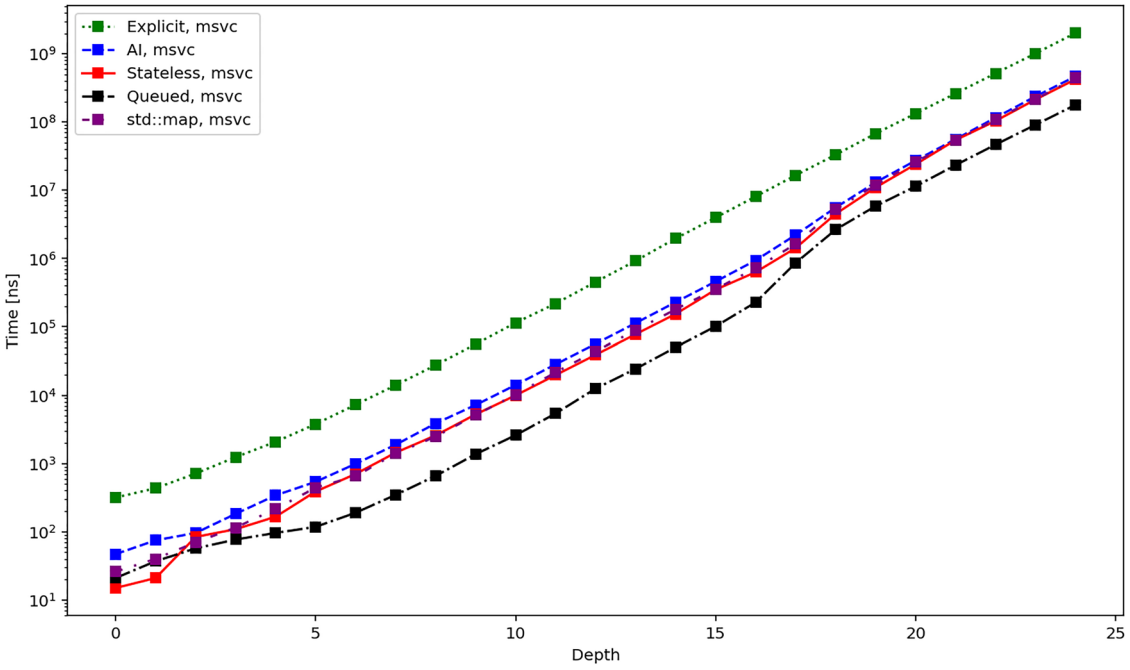

Figure 12: Average total duration in nanoseconds required to iterate over all elements of a tree using different iterators and MSVC compiler.

since the time for the initialization of the iterators becomes negligible for bigger trees. There is one exception, though, and it is the *queued* iterator, which does the majority of the work in the INIT operation. In the subsequent iteration, it should be the fastest one, and this is exactly confirmed by the results.

Finally, Figure 13 shows the average total time required to iterate over all elements of the tree. It presents the same data that was already in Figures 4–6, but this time, it compares the performance of *std::map* iterators from different standard libraries in a single chart. The results clearly show that the performance is practically identical for bigger trees.

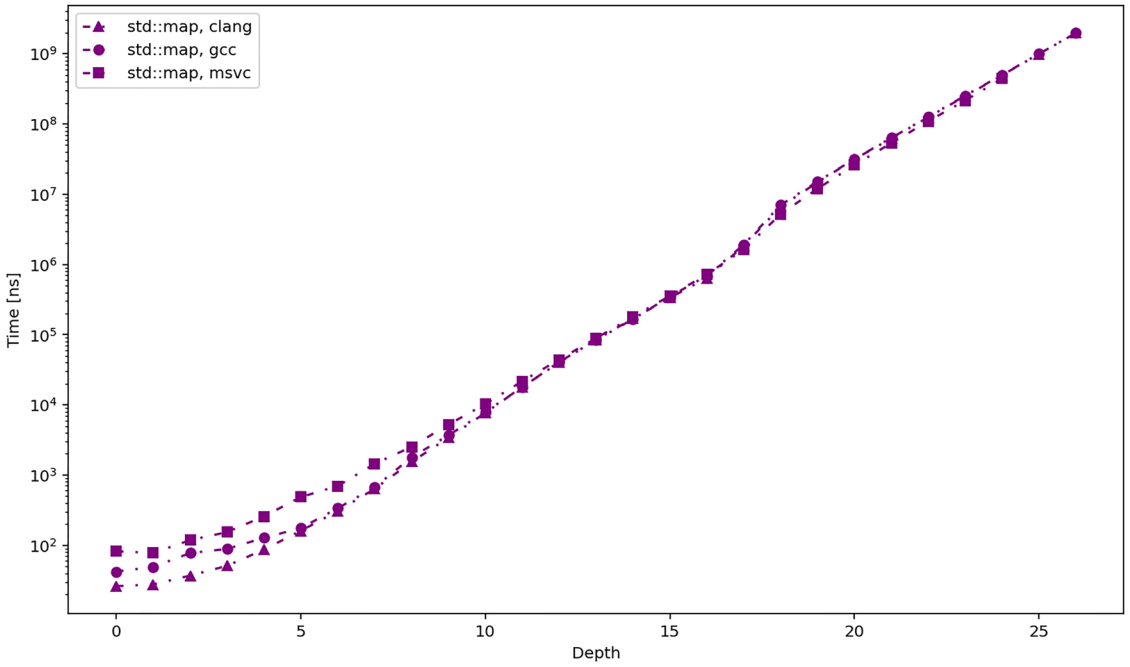

Figure 13: Average total duration in nanoseconds required to iterate over all elements of a tree using different iterators and MSVC compiler.

6 Conclusion

A binary tree is a special type of tree that has numerous applications in computer science. One such application is that it is the underlying memory structure of a BST. In addition to the standard table operations, the BST (and its extensions) also supports the operation of traversal. Traversal does a sequential access of all elements stored in the tree. The traversal is usually implemented using the iterator design pattern. In this article, we focused on the in-order traversal, which is specific to binary trees. The in-order traversal is used in a for-each loop that iterates an ordered table (a table that uses some form of a binary tree in the background) implementation, which is part of the standard library of all commonly used programming languages. Hence, we focused only on this traversal in this article, since it is arguably the one that is most frequently used. We aimed to compare different approaches to the implementation of in-order iterators. Students are often required to implement such an iterator in their algorithms and data structures courses. Since many of them nowadays use generative AI to write code, we also included an AI-generated implementation of an in-order iterator in the comparison.

The main conclusion of our comparison is that the simplest in-order implementations with relatively low memory complexity have the best performance. This was observed consistently for different sizes of the tree as well as different compilers. The implementation of one such iterator we used in the comparison was generated by an AI chatbot. This iterator was easy to incorporate into our project. This shows that we can use a simple query to an AI chatbot, and we get code that is easy to integrate and has decent performance. The results also provided topics for future research. For example, we will try to explain why the performance of the iterators changes for trees of depth 16 or more.

Funding information: This work was funded by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia under the project No. 09I03_03_V02_00030.

Author contributions: All authors have accepted responsibility for the entire content of this manuscript and approved its submission.

Conflict of interest: The authors state no conflict of interest.

Data availability statement: The data that supports the findings ara available in the following repository: https://gitlab.kicon.fri.uniza.sk/varga02/itercomp or from the corresponding author upon request.

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*, 10 edn., Wiley, Hoboken, NJ, USA, 2018.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd edn., The MIT Press, 2009.

[3] E. N. Zaitseva and V. G. Levashenko, “Construction of a reliability structure function based on uncertain data,” *IEEE Trans. Reliability*, vol. 65, pp. 1710–1723, 2016.

[4] J. Piątkowski and S. Szymoniak, “Methodology of testing the security of cryptographic protocols using the cmmtree framework,” *Appl. Sci.*, vol. 13, no. 23, 12668, 2023.

[5] C. Okasaki, *Purely functional data structures*, Cambridge University Press, New York, NY, 1998.

[6] F. M. Carrano and T. M. Henry, *Data abstraction and problem solving with C++: Walls and mirrors*, 7th edn., Pearson Addison Wesley, 2017.

[7] M. Mrena, M. Varga, and M. Kvassay, “Comparison of the approaches to the traversal of a binary tree structure using iterators,” In *2024 IEEE 17th International Scientific Conference on Informatics (Informatics)*, pp. 248–253, 2024.

[8] M. Varga, M. Kvassay, M. Mrena, V. Klima, A. Kaviçï£¡ka, and N. Adamko, *Algoritmy a údajové štruktúry, 2. diel: Abstraktné pamät’ové typy a štruktúry*, in slovak. Žilinská Univerzita v Žiline, EDIS-vydavatel’o UNIZA, 2024, ISBN: 978-80-554-2135-3.

[9] M. Varga, M. Kvašay, M. Mrena, V. Klima, A. Kaviçï£¡ka, and N. Adamko, *Algoritmy a údajové štruktúry, 3. diel: Abstraktné údajové typy a štruktúry*, in slovak. Žilinská Univerzita v Žiline, EDIS-vydavatel’o UNIZA, 2024, ISBN: 978-80-554-2136-0.

[10] J. A. Bondy and U. S. R. Murty, *Graph theory with applications*. Elsevier Science Publishing, New York, 1976.

[11] P. F. Windley, “Trees, forests and rearranging,” *Comput. J.*, vol. 3, no. 2, pp. 84–88, 1960.

[12] D. E. Knuth, *The art of computer programming, Volume 3: (2nd ed.) sorting and searching*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[13] java.util.map. 2025. https://docs.oracle.com/javase/8/docs/api/java/util/Map.html, Accessed: 2025-02-24.

[14] .net dictionary. 2025. https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-8.0. Accessed: 2025-02-24.

[15] Python dictionary. 2025. https://docs.python.org/3/tutorial/datastructures.html#dictionaries, *Accessed: 2025-02-24*.

[16] Php associative array. https://www.php.net/manual/en/language.types.array.php, 2025, Accessed: 2025-02-24.

[17] N. Wirth, *Algorithms and data structures*, Prentice-Hall International editions, Prentice-Hall International, 1986.

[18] R. Seidel and C. R. Aragon, “Randomized search trees,” *Algorithmica*, vol. 16, pp. 464–497, 1996.

[19] D. D. Sleator and R. E. Tarjan, “Self-adjusting binary search trees,” *J. Acm*, vol. 32, no. 3, pp. 652–686, 1985.

[20] L. J. Guibas and R. Sedgewick, “A dichromatic framework for balanced trees,” In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pp. 8–21, 1978.

[21] G. Hutton, *Programming in Haskell*, 2nd edn., Cambridge University Press, 2016.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[23] Range-based for loop. 2025. https://en.cppreference.com/w/cpp/language/range-for, Accessed: 2025-02-24.
[24] The for-each loop. 2025, https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html, Accessed: 2025-02-24.
[25] The foreach statement. 2025, https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements#the-foreach-statement, Accessed: 2025-02-24.
[26] Iterable.foreach. 2025. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Iterable.html#forEach(java.util.function.Consumer), *Accessed: 2025-02-24*.
[27] Llvm project - libc++. 2025. https://github.com/llvm/llvm-project/blob/f11c0a1a0d9306456a99e609833d7b188fa904fb/libcxx/include/__tree#L194, Accessed: 2025-02-24.
[28] Microsoft stl. 2025, https://github.com/microsoft/STL/blob/ef1d621d51263285aff8e560a214f5477d63d687/stl/inc/xtree#L49, Accessed: 2025-02-24.
[29] Gcc – libstdc++. https://github.com/gcc-mirror/gcc/blob/6fce4664d4a2e44843bd1464930696c819906d0f/libstdc%2B%2B-v3/src/c%2B%2B98/tree.cc#L83, 2025. Accessed: 2025-02-24.
[30] Openai's chatgpt 3.5. 2025. https://chatgpt.com/, Accessed: 2025-02-24.
[31] Gemini 2.0. https://gemini.google.com/app?hl=sk/, 2025. Accessed: 2025-02-24.
[32] Microsoft copilot. 2025, https://copilot.microsoft.com/, Accessed: 2025-02-24.
[33] Deepseek v2.5. 2025, https://www.deepseek.com/, Accessed: 2025-02-24.