Succinct Data Structures for Baxter Permutation and Related Families

Sankardeep Chakraborty
The University of Tokyo, Japan

Seungbum Jo[1]
Chungnam National University, Daejeon, Republic of Korea

Geunho Kim
Pohang University of Science and Technology, Republic of Korea

Kunihiko Sadakane
The University of Tokyo, Japan

Abstract

A permutation $\pi : [n] \to [n]$ is a Baxter permutation if and only if it does not contain either of the patterns 2–41–3 and 3–14–2. Baxter permutations are one of the most widely studied subclasses of general permutation due to their connections with various combinatorial objects such as plane bipolar orientations and mosaic floorplans, etc. In this paper, we introduce a novel succinct representation (i.e., using $o(n)$ additional bits from their information-theoretical lower bounds) for Baxter permutations of size n that supports $\pi(i)$ and $\pi^{-1}(j)$ queries for any $i \in [n]$ in $O(f_1(n))$ and $O(f_2(n))$ time, respectively. Here, $f_1(n)$ and $f_2(n)$ are arbitrary increasing functions that satisfy the conditions $\omega(\log n)$ and $\omega(\log^2 n)$, respectively. This stands out as the first succinct representation with sub-linear worst-case query times for Baxter permutations. The main idea is to traverse the Cartesian tree on the permutation using a simple yet elegant *two-stack algorithm* which traverses the nodes in ascending order of their corresponding labels and stores the necessary information throughout the algorithm.

Additionally, we consider a subclass of Baxter permutations called *separable permutations*, which do not contain either of the patterns 2–4–1–3 and 3–1–4–2. In this paper, we provide the first succinct representation of the separable permutation $\rho : [n] \to [n]$ of size n that supports both $\rho(i)$ and $\rho^{-1}(j)$ queries in $O(1)$ time. In particular, this result circumvents Golynski's [SODA 2009] lower bound result for trade-offs between redundancy and $\rho(i)$ and $\rho^{-1}(j)$ queries.

Moreover, as applications of these permutations with the queries, we also introduce the first succinct representations for mosaic/slicing floorplans, and plane bipolar orientations, which can further support specific navigational queries on them efficiently.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Succinct data structure, Baxter permutation, Mosaic floorplan, Plane bipolar orientation

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2024.17

Related Version *Full Version*: `https://arxiv.org/abs/2409.16650` [8]

Funding *Seungbum Jo*: This work was supported by research fund of Chungnam National University.
Geunho Kim: This research was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00905, Software Star Lab (Optimal Data Structure and Algorithmic Applications in Dynamic Geometric Environment)).

[1] Corresponding author

1 Introduction

A permutation $\pi : [n] \to [n]$ is a Baxter permutation if and only if there are no three indices $i < j < k$ that satisfy $\pi(j+1) < \pi(i) < \pi(k) < \pi(j)$ or $\pi(j) < \pi(k) < \pi(i) < \pi(j+1)$ (that is, π does not have pattern 2–41–3 or 3–14–2) [2]. For example, 3 5 2 1 4 is not a Baxter permutation because the pattern 2–41–3 appears ($\pi(2+1) = 2 < \pi(1) = 3 < \pi(5) = 4 < \pi(2) = 5$ holds). A Baxter permutation π is *alternating* if the elements in π rise and descend alternately. One can also consider *separable permutations*, which are defined as the permutations without two patterns 2–4–1–3 and 3–1–4–2 [7]. From the definitions, any separable permutation is also a Baxter permutation, but the converse does not hold. For example, 2 5 6 3 1 4 8 7 is a Baxter permutation but not a separable permutation because of the appearance of the pattern 2–4–1–3 (2 5 1 4).

In this paper, we focus on the design of a succinct data structure for a Baxter permutation π of size n, i.e., the data structure that uses up to $o(n)$ extra bits in addition to the information-theoretical lower bound along with supporting relevant queries efficiently. Mainly, we consider the following two fundamental queries on π: (1) $\pi(i)$ returns the i-th value of π, and (2) $\pi^{-1}(j)$ returns the index i of $\pi(i) = j$. We also consider the design of a succinct data structure for a separable permutation ρ of size n that supports $\rho(i)$ and $\rho^{-1}(j)$ queries. In the rest of this paper, log denotes the logarithm to the base 2, and we assume a word-RAM model with $\Theta(\log n)$-bit word size, where n is the size of the input. Also, we ignore all ceiling and floor operations that do not impact the final results.

1.1 Previous Results

For general permutations, there exist upper and lower bound results for succinct data structures supporting both $\pi(i)$ and $\pi^{-1}(j)$ queries in sub-linear time [20, 27]. However, to the best of our knowledge, there does not exist any data structures for efficiently supporting these queries on any subclass of general permutations. One can consider suffix arrays [21] as a subclass of general permutations, but their space consumption majorly depends on the entropy of input strings. This implies that for certain input strings, $\Omega(n \log n)$ bits (asymptotically the same space needed for storing general permutations) are necessary for storing the suffix arrays on them.

Baxter permutation is one of the most widely studied classes of permutations [5] because diverse combinatorial objects, for example, plane bipolar orientations, mosaic floorplans, twin pairs of binary trees, etc. have a bijection with Baxter permutations [1, 15]. Note that some of these objects are used in many applied areas. For example, mosaic floorplans are used in large-scale chip design [25], plane bipolar orientations are used to draw graphs in various flavors (visibility [34], straight-line drawing [17]), and floorplan partitioning is used to design a model for stochastic processes [29]. The number of distinct Baxter permutations of size n is $\Theta(8^n/n^4)$ [32], which implies that at least $3n - o(n)$ bits are necessary to store a Baxter permutation of size n. Furthermore, the number of distinct alternating Baxter permutations of size $2n$ (resp. $2n+1$) is $(c_n)^2$ (resp. $c_n c_{n+1}$) where $c_n = \frac{(2n)!}{(n+1)!n!}$ is the n-th Catalan number [11]. Therefore, at least $2n - o(n)$ bits are necessary to store an alternating Baxter permutation of size n. Dulucq and Guibert [12] established a bijection between Baxter permutations π of size n and a pair of unlabeled binary trees, called *twin binary trees*, which are essentially equivalent to the pair of unlabeled minimum and maximum Cartesian trees [35] for π. They provided methods for constructing π from the structure of twin binary trees and vice versa, both of which require $O(n)$ time. Furthermore, they presented a representation scheme that requires at most $8n$ bits for Baxter permutations of size n and $4n$

bits for alternating Baxter permutations of size n. Gawrychowski and Nicholson proposed a $3n$-bit representation that stores the tree structures of alternating representations of both minimum and maximum Cartesian trees [18]. Based on the bijection established in [12], the representation in [18] gives a succinct representation of a Baxter permutation of size n. Moreover, this representation can efficiently support a wide range of tree navigational queries on these trees in $O(1)$ time using only $o(n)$ additional bits. However, surprisingly, all of these previous representations of π crucially fail to address both, perhaps the most natural, $\pi(i)$ and $\pi^{-1}(j)$ queries efficiently as these queries have a worst-case time complexity of $\Theta(n)$.

Separable permutation was introduced by Bose et al.[7] as a specific case of patterns for the permutation matching problem. It is known that the number of separable permutations of size n equals the *large Schröder number* A_n, which is $\Theta\left(\frac{(3+2\sqrt{2})^n}{n^{1.5}}\right)$[36]. Consequently, to store a separable permutation ρ of size n, at least $n\log(3+2\sqrt{2})-O(\log n) \simeq 2.54n-O(\log n)$ bits are necessary. Bose et al. [7] also showed that ρ can be encoded as a *separable tree*, which is a labeled tree with at most $2n-1$ nodes. Thus, by storing the separable tree using $O(n\log n)$ bits, one can support both $\rho(i)$ and $\rho^{-1}(j)$ queries in $O(1)$ time using standard tree navigation queries. Yao et al. [36] showed a bijection between all canonical forms of separable trees with n leaves and the separable permutations of size n. To the best of our knowledge, there exists no $o(n\log n)$-bit representation for storing either separable permutations or their corresponding separable trees that can be constructed in polynomial time while supporting ρ queries in sub-linear time.

A mosaic floorplan is a collection of rectangular objects that partition a single rectangular region. Due to its broad range of applications, there is a long history of results (see [22, 36] and the references therein) concerning the representation of mosaic floorplans of size n in small space [1, 22, 23]. Ackerman et al. [1] presented a linear-time algorithm to construct a mosaic floorplan of size n from its corresponding Baxter permutation of size n and vice versa. Building on this construction algorithm, He [22] proposed the current state-of-the-art, a succinct representation of a mosaic floorplan of size n using $3n-3$ bits. Again, all of these previous representations primarily focus on constructing a complete mosaic floorplan structure and do not consider supporting navigational queries, e.g., return a rectangular object immediately adjacent to the query object in terms of being left, right, above, or below it, without constructing it completely. Note that these queries have strong applications like the placement of blocks on the chip [1, 37]. There also exists a subclass of mosaic floorplans known as *slicing floorplans*, which are mosaic floorplans whose rectangular objects are generated by recursively dividing a single rectangle region either horizontally or vertically. The simplicity of a slicing floorplan makes it an efficient solution for optimization problems, as stated in [38]. Yao et al. [36] showed there exists a bijection between separable permutations of size n and slicing floorplans with n rectangular objects. They also showed that separable trees can be used to represent the positions of rectangular objects in the corresponding slicing floorplans. However, to the best of our knowledge, there exists no representation of a slicing floorplan using $o(n\log n)$ bits that supports the above queries without reconstructing it.

1.2 Our Results and Main Idea

In this paper, we first introduce a $(3n+o(n))$-bit representation of a Baxter permutation π of size n that can support $\pi(i)$ and $\pi^{-1}(j)$ queries in $O(f_1(n))$ and $O(f_2(n))$ time respectively. Here, $f_1(n)$ and $f_2(n)$ are any increasing functions that satisfy $\omega(\log n)$ and $\omega(\log^2 n)$, respectively. We also show that the same representation provides a $(2n+o(n))$-bit representation of an alternating Baxter permutation of size n with the same query times. These are the first succinct representations of Baxter and alternating Baxter permutations that can support the queries in sub-linear time in the worst case.

Our main idea of the representation is as follows. To represent π, it suffices to store the minimum or maximum Cartesian tree defined on π along with their labels. Here the main challenging part is to decode the label of any node in either of the trees in sub-linear time, using $o(n)$-bit auxiliary structures. Note that all the previous representations either require linear time for the decoding or explicitly store the labels using $O(n \log n)$ bits. To address this issue, we first introduce an algorithm that labels the nodes in the minimum Cartesian tree in ascending order of their labels. This algorithm employs two stacks and only requires information on whether each node with label i is a left or right child of its parent, as well as whether it has left and/or right children. Note that unlike the algorithm of [12], our algorithm does not use the structure of the maximum Cartesian tree. We then proceed to construct a representation using at most $3n + o(n)$ bits, which stores the information used throughout our labeling algorithm. We show that this representation can decode the minimum Cartesian tree, including the labels on its nodes. This approach was not considered in previous succinct representations that focused on storing the tree structures of both minimum and maximum Cartesian trees, or their variants. To support the queries efficiently, we show that given any label of a node in the minimum or maximum Cartesian tree, our representation can decode the labels of its parent, left child, and right child in $O(1)$ time with $o(n)$-bit auxiliary structures. Consequently, we can decode any $O(\log n)$-size substring of the balanced parentheses of both minimum and maximum Cartesian trees with dummy nodes to locate nodes according to their inorder traversal (see Section 4.1 for a detailed definition of the inorder traversal) on π in $O(f_1(n))$ time. This decoding step plays a key role in our query algorithms, which can be achieved from non-trivial properties of our representation, and minimum and maximum Cartesian trees on Baxter permutations. As a result, our representation not only supports $\pi(i)$ and $\pi^{-1}(j)$ queries, but also supports range minimum/maximum and previous/next larger/smaller value queries efficiently.

Next, we give a succinct representation of separable permutation ρ of size n, which supports all the operations above in $O(1)$ time. Our result implies the Golynski's lower bound result [20] for trade-offs between redundancy and $\rho(i)$ and $\rho^{-1}(j)$ queries does not hold in separable permutations. The main idea of the representation is to store the separable tree of ρ using the *tree covering algorithm* [14], where each micro-tree is stored as its corresponding separable permutation to achieve succinct space. Note that a similar approach has been employed for succinct representations on some graph classes [4, 9]. However, due to the different structure of the separable tree compared to the Cartesian tree, the utilization of non-trivial auxiliary structures is crucial for achieving $O(1)$ query time on the representation.

Finally, as applications of our succinct representations of Baxter and separable permutations, we present succinct data structures of mosaic and slicing floorplans and plane bipolar orientations that support various navigational queries on them efficiently. While construction algorithms for these structures already exist from their corresponding Baxter or separable permutations [1, 6], we show that the navigational queries can be answered using a constant number of $\pi(i)$ (or $\rho(i)$), range minimum/maximum, and previous/next smaller/larger value queries on their respective permutations, which also require some nontrivial observations from the construction algorithms. This implies that our succinct representations allow for the first time succinct representations of these structures that support various navigation queries on them in sub-linear time. For example, we consider two queries on mosaic and slicing floorplans as (1) checking whether two rectangular objects are adjacent, and (2) reporting all rectangular objects adjacent to the given rectangular object. Note that the query of (2) was previously addressed in [1], as the *direct relation set* (DRS) query, which was computed in $O(n)$ time, and important for the actual placement of the blocks on the chip.

The paper is organized as follows. We introduce the representation of a Baxter permutation π of size n in Section 3. In Section 4, we explain how to support $\pi(i)$ and $\pi^{-1}(j)$ queries on π, in addition to tree navigational queries on both the minimum and maximum Cartesian trees. In Section 5.1, we present a succinct representation of separable permutation ρ that can support $\rho(i)$ and $\rho^{-1}(j)$ in $O(1)$ time. Finally, some preliminaries are outlined in the next section. Due to the space limit, the remaining results of our work (succinct representations of mosaic/slicing floorplans and plane bipolar orientations) are included in the full version of the paper [8].

2 Preliminaries

In this section, we introduce some preliminaries that will be used in the rest of the paper.

Cartesian trees. Given a sequence $S = (s_1, s_2, \dots, s_n)$ of size n from a total order, a *minimum Cartesian tree* of S, denoted as $\mathsf{MinC}(S)$ is a binary tree constructed as follows [35]: (a) the root of the $\mathsf{MinC}(S)$ is labeled as the minimum element in S (b) if the label of the root is s_i, the left and right subtree of S are $\mathsf{MinC}(S_1)$ and $\mathsf{MinC}(S_2)$, respectively where $S_1 = (s_1, s_2, \dots, s_{i-1})$ and $S_2 = (s_{i+1}, s_{i+2}, \dots, s_n)$. One can also define a maximum Cartesian tree of S (denoted as $\mathsf{MaxC}(S)$) analogously. From the definition, in both $\mathsf{MinC}(S)$ and $\mathsf{MaxC}(S)$, any node with inorder i is labeled with s_i.

Balanced parentheses. Given an ordered tree T of n nodes, the BP of T (denoted as $BP(T)$) is defined as a sequence of open and closed parentheses constructed as follows [28]. One traverses T from the root node in depth-first search (DFS) order. During the traversal, for each node $p \in T$, we append "(" when we visit the node p for the first time, and append ")" when all the nodes on the subtree rooted at p are visited, and we leave the node p. From the construction, it is clear that the size of $BP(T)$ is $2n$ bits, and always balanced. Munro and Raman [28] showed that both (a) $\mathsf{findopen}(i)$: returns the position of matching open parenthesis of the close parenthesis at i, and (b) $\mathsf{findclose}(i)$: returns the position of matching close parenthesis of the open parenthesis at i, queries can be supported on $BP(T)$ in $O(t(n))$ time with $o(n)$-bit auxiliary structures, when any $O(\log n)$-bit substring of the $BP(T)$ can be decoded in $t(n)$ time. Furthermore, it is known that the wide range of tree navigational queries on T also can be answered in $O(t(n))$ time using $BP(T)$ with $o(n)$-bit auxiliary structures [30]: Here, each node is given and returned as the position of the open parenthesis that appended when the node is first visited during the construction of $BP(T)$ (for the full list of the queries, please refer to Table I in [30]).

Rank and Select queries. Given a sequence $S = (s_1, s_2, \dots, s_n) \in \{0, \dots, \sigma - 1\}^n$ of size n over an alphabet of size σ, (a) $\mathsf{rank}_S(a, i)$ returns the number of occurrence of $a \in \{0, \dots, \sigma-1\}$ in $(s_1, s_2, \dots, s_i)$, and (b) $\mathsf{select}_S(a, j)$ returns the first position of the j-th occurrence of $a \in \{0, \dots, \sigma - 1\}$ in S (in the rest of this paper, we omit S if it is clear from the context). The following data structures are known, which can support both rank and select queries efficiently using succinct space [3, 31]: (1) suppose $\sigma = 2$, and S has m 1s. Then there exists a $(\log \binom{n}{m} + o(n))$-bit data structure that supports both rank and select queries in $O(1)$ time. The data structure can also decode any $O(\log n)$ consecutive bits of S in $O(1)$ time, (2) there exists an $(n \log \sigma + o(n))$-bit data structure that can support both rank and select queries in $O(1)$ time, and (3) if $\sigma = O(1)$ and one can access any $O(\log n)$-length sequence of S in $t(n)$ time, one can support both rank and select queries in $O(t(n))$ time using $o(n)$-bit auxiliary structures.

Range minimum and previous/next smaller value queries. Given a sequence $S = (s_1, s_2, \ldots, s_n)$ of size n from a total order with two positions i and j with $i \leq j$, the *range minimum query* $\mathsf{RMin}(i, j)$ on S returns the position of the smallest element within the range $s_i, \ldots, s_j$. Similarly, a *range maximum query* $\mathsf{RMax}(i, j)$ on S is defined to find the position of the largest element within the same range.

In addition, one can define *previous (resp. next) smaller value queries* at the position i on S, denoted as $\mathsf{PSV}(i)$ (resp. $\mathsf{NSV}(i)$), which returns the nearest position from i to the left (resp. right) whose value is smaller than s_i. If there is no such elements, the query returns 0 (resp. $n + 1$). One can also define *previous (resp. next) larger value queries*, denoted as $\mathsf{PLV}(i)$ (resp. $\mathsf{NLV}(i)$) analogously.

It is known that if S is a permutation, RMin, PSV, and NSV queries on S can be answered in $O(1)$ time, given a BP of $\mathsf{MinC}(S)$ with $o(n)$ bit auxiliary structures [16, 30].

Tree Covering. Here, we briefly outline Farzan and Munro's [14] tree covering representation and its application in constructing a succinct tree data structure. The core idea involves decomposing the input tree into *mini-trees* and further breaking them down into smaller units called *micro-trees*. These micro-trees can be efficiently stored in a compact precomputed table. The shared roots among mini-trees enable the representation of the entire tree by focusing only on connections and links between these subtrees. We summarize the main result of Farzan and Munro's algorithm in the following theorem.

▶ **Theorem 1** ([14]). *For a rooted ordered tree with n nodes and a positive integer $1 \leq \ell_1 \leq n$, one can decompose the trees into subtrees satisfying the following conditions: (1) each subtree contains at most $2\ell_1$ nodes, (2) the number of subtrees is $O(n/\ell_1)$, (3) each subtree has at most one outgoing edge, apart from those from the root of the subtree.*

See Figure 3 for an example. After decomposing the subtree as above, any node with an outgoing edge to a child outside the subtree is termed a *boundary node*. The corresponding edge is referred to as the *non-root boundary edge*. Each subtree has at most one boundary node and a non-root boundary edge. Additionally, the subtree may have outgoing edges from its root node, designated as *root boundary edges*. For example, to achieve a tree covering representation for an arbitrary tree with n nodes, Theorem 1 is initially applied with $\ell_1 = \log^2 n$, yielding $O(n/\log^2 n)$ mini-trees. The resulting tree, formed by contracting each mini-tree into a vertex, is denoted as the *tree over mini-trees*. This tree, with $O(n/\log^2 n)$ nodes, can be represented in $O(n/\log n) = o(n)$ bits through a pointer-based representation. Subsequently, Theorem 1 is applied again to each mini-tree with $\ell_2 = \frac{1}{6} \log n$, resulting in a total of $O(n/\log n)$ micro-trees. The *mini-tree over micro-trees*, formed by contracting each micro-tree into a node and adding dummy nodes for micro-trees sharing a common root, has $O(\log n)$ vertices and is represented with $O(\log \log n)$-bit pointers. Encoding the non-root/root boundary edge involves specifying the originating vertex and its rank among all children. The succinct tree representation, such as balanced parentheses (BP) [26], is utilized to encode the position of the boundary edge within the micro-tree, requiring $O(\log \ell_2)$ bits. The overall space for all mini-trees over micro-trees is $O(n \log \log n/ \log n) = o(n)$ bits. Finally, the micro-trees are stored with two-level pointers in a precomputed table containing representations of all possible micro-trees, demonstrating a total space of $2n + o(n)$ bits. By utilizing this representation, along with supplementary auxiliary structures that require only $o(n)$ bits of space, it is possible to perform fundamental tree navigation operations, such as accessing the parent, the i-th child, the lowest common ancestor, among many others, in $O(1)$ time [14].

3 Succinct Representation of Baxter Permutation

In this section, we present a $(3n + o(n))$-bit representation for a Baxter permutation $\pi = (\pi(1), \ldots, \pi(n))$ of size n. We begin by providing a brief overview of our representation. It is clear that the tree structure of $\mathsf{MinC}(\pi)$, along with the associated node labels can decode π completely. However, the straightforward storage of node labels uses $\Theta(n \log n)$ bits, posing an efficiency challenge. To address this issue, we first show that when π is a Baxter permutation, a two-stack based algorithm can be devised to traverse the nodes of $\mathsf{MinC}(\pi)$ according to the increasing order of their labels. After that, we present a $(3n + o(n))$-bit representation that stores the information used throughout the algorithm, and show that the representation can decode $\mathsf{MinC}(\pi)$ with the labels of the nodes.

Algorithm 1 Two-stack based algorithm.

```
Initialize two empty stacks L and R.
Visit φ(1) (i.e., the root of MinC(π)).
while i = 2...n do
    // The last visited node is φ(i − 1) by Lemma 2.
    if φ(i) is a left child of its parent then
        if φ(i − 1) has a left child then
            Visit the left child of φ(i − 1).
        else
            Pop a node from stack L, and visit the left child of the node.
        end
        if φ(i − 1) has a right child that has not yet been visited then
            Push φ(i − 1) to the stack R.
        end
    else // φ(i) is a right child of its parent
        if φ(i − 1) has a right child then
            Visit the right child of φ(i − 1).
        else
            Pop a node from stack R, and visit the right child of the node.
        end
        if φ(i − 1) has a left child that has not yet been visited then
            Push φ(i − 1) to the stack L.
        end
    end
end
```

Note that our encoding employs a distinct approach compared to prior representations, as seen in references [12, 13, 18, 24]. These earlier representations store the tree structures of $\mathsf{MinC}(\pi)$ and $\mathsf{MaxC}(\pi)$ (or their variants) together, based on the observation that there always exists a bijection between π and the pair of $\mathsf{MinC}(\pi)$ and $\mathsf{MaxC}(\pi)$ if π is a Baxter permutation [12]. We show that for any node in $\mathsf{MinC}(\pi)$, our representation allows to decode the labels of its parent, left child, and right child in $O(1)$ time using $o(n)$-bit auxiliary data structures. Using the previous representations that only store tree structures of $\mathsf{MinC}(\pi)$ and $\mathsf{MaxC}(\pi)$, these operations can take up to $\Theta(n)$ time in the worst-case scenario, even though tree navigation queries can be supported in constant time.

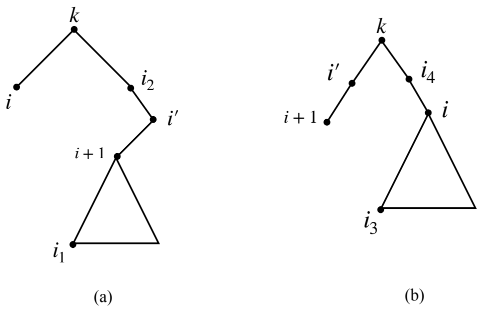


Figure 1 (a) the case when $\phi(i)$ is in the left subtree of $\phi(k)$, and (b) the case when $\phi(i)$ is in the right subtree of $\phi(k)$.

Now we introduce a two-stack based algorithm to traverse the nodes in $\mathsf{MinC}(\pi)$ according to the increasing order of their labels. Let $\phi(i)$ denote the node of $\mathsf{MinC}(\pi)$ with the label i. The algorithm assumes that we know whether $\phi(i)$ is left or right child of its parent for all $i \in \{2, \ldots, n\}$.

The following lemma shows that if π is a Baxter permutation, the two-stack based algorithm works correctly.

▶ **Lemma 2.** *If π is a Baxter permutation, the two-stack based algorithm on $\mathsf{MinC}(\pi)$ traverses the nodes according to the increasing order of their labels.*

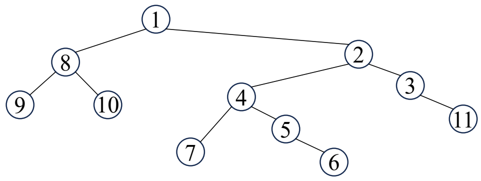
Proof. From Algorithm 1, it is clear that we first visit the root node, which is $\phi(1)$. Then we claim that for any i, the two-stack based algorithm traverses the node $\phi(i+1)$ immediately after traversing $\phi(i)$, thereby proving the theorem.

Suppose not. Then we can consider the cases as (a) the left child of $\phi(i)$ exists, but $\phi(i+1)$ is not a left child of $\phi(i)$, or (b) the left child of $\phi(i)$ does not exist, but $\phi(i+1)$ is not a left child of the node at the top of L. For the case (a) (the case (b) can be handled similarly), suppose $\phi(i+1)$ is a left child of the node $\phi(i')$. Then $i' < i$ by the definition of $\mathsf{MinC}(\pi)$ and the case (a). Now, let $\phi(k)$ be the lowest common ancestor of $\phi(i)$ and $\phi(i')$. If $\phi(i)$ is in the left subtree of $\phi(k)$ (see Figure 1(a) for an example), k cannot be i' from the definition of $\mathsf{MinC}(\pi)$. Then consider two nodes, $\phi(i_1)$ and $\phi(i_2)$, which are the leftmost node of the subtree rooted at node $\phi(i')$ and the node whose inorder is immediately before $\phi(i_1)$, respectively. Since $\phi(i_2)$ lies on the path from $\phi(k)$ to $\phi(i')$, we have $i+1 \le i_1$ and $k \le i_2 < i'$. Therefore, there exists a pattern 3–14–2 induced by $i - i_2, i_1 - i'$, which contradicts the fact that π is a Baxter permutation.

If $\phi(i)$ is in the right subtree of $\phi(k)$ (see 1(b) for an example), k cannot be i from the definition of $\mathsf{MinC}(\pi)$. Consider two nodes, $\phi(i_3)$ and $\phi(i_4)$, which are the leftmost node of the subtree rooted at node $\phi(i)$ and the node whose inorder is immediately before $\phi(i_3)$, respectively. Since $\phi(i_4)$ lies on the path from $\phi(k)$ to $\phi(i)$, we have $i+1 < i_3$ (i_3 is greater than i and cannot be $i+1$) and $k \le i_4 < i$. Therefore, there exists a pattern 3–14–2 induced by $(i+1) - i_4, i_3 - i$, which contradicts the fact that π is a Baxter permutation.

The case when $\phi(i+1)$ is a right child of its parent can be proven using the same argument by showing that if the algorithm fails to navigate $\phi(i+1)$ correctly, the pattern 2–41–3 exists in π. ◀

The representation of π encodes the two-stack based algorithm as follows. First, to indicate whether each non-root node is whether a left or right child of its parent, we store a binary string $\mathsf{lr}[1, \ldots n-1] \in \{l, r\}^{n-1}$ of size $n-1$ where $\mathsf{lr}[i] = l$ (resp. $\mathsf{lr}[i] = r$) if the node $\phi(i+1)$ is a left (resp. right) child of its parent. Next, to decode the information on



	1	2	3	4	5	6	7	8	9	10	11
lr	*r*	*r*	*l*	*r*	*r*	*l*	*l*	*l*	*r*	*r*	
lp	(	(	)	(		)	)				
rp			{					{	}	}	
E	3	3	2	3	2	0	0	3	0	0	
lrp	(	(	{)	(	[]	)	)	{	}	}	
U	1	1	2	1	2	1	1	1	1	1	

	1	2	3	4	5	6	7	8	9	10	11
permutation	9	8	10	1	7	4	5	6	2	3	11

Figure 2 An example of the representation of the Baxter permutation $\pi = (9, 8, 10, 1, 7, 4, 5, 6, 2, 3, 11)$. Note that the data structure maintains only E and lr along with $o(n)$-bit auxiliary structures.

the stack L during the algorithm, we define an imaginary string of balanced parentheses $\mathsf{lp}[1 \dots n-1]$ as follows: After the algorithm traverses $\phi(i)$, $\mathsf{lp}[i]$ is (1) "(" if the algorithm pushes $\phi(i)$ to the stack L, (2) ")" if the algorithm pops a node from the stack L, and (3) undefined otherwise. We also define an imaginary string of balanced parentheses $\mathsf{rp}[1, \dots n]$ in the same way to decode the information on the stack R during the algorithm. We use "{" and "}" to denote the parentheses in rp. Then from the correctness of the two-stack algorithm (Lemma 2), and the definitions of lr, lp, and rp, we can directly derive the following lemma:

▶ **Lemma 3.** *For any $i \in \{1, \dots, n-1\}$, the following holds:*

- *Suppose the node $\phi(i)$ is a leaf node. Then either $\mathsf{lp}[i]$ or $\mathsf{rp}[i]$ is defined. Also, $\mathsf{lr}[i]$ is l (resp. r) if and only if $\mathsf{lp}[i]$ (resp. $\mathsf{rp}[i]$) is a closed parenthesis.*
- *Suppose the node $\phi(i)$ only has a left child. In this case, $\mathsf{lr}[i]$ is l if and only if both $\mathsf{lp}[i]$ and $\mathsf{rp}[i]$ are undefined. Also, $\mathsf{lr}[i]$ is r if and only if $\mathsf{lp}[i]$ = "(" and $\mathsf{rp}[i]$ = "}".*
- *Suppose the node $\phi(i)$ only has a right child. In this case, $\mathsf{lr}[i]$ is l if and only if $\mathsf{lp}[i]$ = ")" and $\mathsf{rp}[i]$ = "{". Also, $\mathsf{lr}[i]$ is r if and only if both $\mathsf{lp}[i]$ and $\mathsf{rp}[i]$ are undefined.*
- *Suppose the node $\phi(i)$ has both left and right child. In this case, $\mathsf{lr}[i]$ is l if and only if $\mathsf{lp}[i]$ is undefined and $\mathsf{rp}[i]$ = "{". Also, $\mathsf{lr}[i]$ is r if and only if $\mathsf{lp}[i]$ = "(" and $\mathsf{rp}[i]$ is undefined.*

To indicate whether each node $\phi(i)$ has a left and/or right child we store a string $E \in \{0, 1, 2, 3\}^{n-1}$ of size $n-1$ where (a) $E[i] = 0$ if $\phi(i)$ is a leaf node (b) $E[i] = 1$ if $\phi(i)$ has only a left child, (c) $E[i] = 2$ if $\phi(i)$ has only a right child, and (d) $E[i] = 3$ if $\phi(i)$ has both left and right children. We store E using $2n + o(n)$ bits, which allows support both rank and select operations in $O(1)$ time [3]. Thus, the overall space required for our representation is at most $3n + o(n)$ bits ($2n$ bits for E, n bits for lr along with $o(n)$-bit auxiliary structures). From Lemma 3, our representation can access $\mathsf{lp}[i]$ and $\mathsf{rp}[i]$ in $O(1)$ time by referring $\mathsf{lr}[i]$ and $E[i]$. Next, we show both $\mathsf{findopen}$ and $\mathsf{findclose}$ on lp and rp in $O(1)$ time using the representation. We define an imaginary string lrp of length at most $2(n-1)$ over an alphabet of size 6 that consists of three different types of parentheses (), {}, [] constructed as follows. We first initialize lrp as an empty string and scan lr and E from the leftmost position. Then based on Lemma 3, whenever we scan $\mathsf{lr}[i]$ and $E[i]$, we append the parentheses to lrp as follows:

$$\begin{cases} (& \text{if } \mathsf{lr}[i] = r \text{ and } E[i] = 3 \\ ) & \text{if } \mathsf{lr}[i] = l \text{ and } E[i] = 0 \\ \{ & \text{if } \mathsf{lr}[i] = l \text{ and } E[i] = 3 \\ \} & \text{if } \mathsf{lr}[i] = r \text{ and } E[i] = 0 \\ (\} & \text{if } \mathsf{lr}[i] = r \text{ and } E[i] = 1 \\ \{) & \text{if } \mathsf{lr}[i] = l \text{ and } E[i] = 2 \\ [] & \text{if (1) } \mathsf{lr}[i] = l \text{ and } E[i] = 1\text{, or (2) } \mathsf{lr}[i] = r \text{ and } E[i] = 2 \end{cases}$$

We store a precomputed table that has all possible pairs of lr and E of size $(\log n)/4$ as indices. For each index of the table, it returns lrp constructed from the corresponding pair of lr and E. Thus, the size of the precomputed table is $O(2^{\frac{3}{4}\log n} \log n) = o(n)$ bits

Additionally, we define an imaginary binary sequence $U \in \{1,2\}^{n-1}$ of size $n-1$, where $U[i]$ denotes the number of symbols appended to lrp during its construction by scanning $\mathsf{lr}[i]$ and $E[i]$. Then by Lemma 3, we can decode any $O(\log n)$-sized substring of U starting from position $U[i]$ by storing another precomputed table of size $o(n)$ bits, indexed by all possible pairs of lr and E of size $(\log n)/4$. Consequently, we can support both rank and select queries on U by storing $o(n)$-bit auxiliary structures, without storing U explicitly [3].

To decode any $O(\log n)$-sized substring of lrp starting from position $\mathsf{lrp}[i]$, we first decode a $O(\log n)$-sized substring of E and lr from the position $i' = i - \mathsf{rank}_U(2, i)$ and decode the substring of lrp by accessing the precomputed table a constant number of times (bounded conditions can be easily verified using $\mathsf{rank}_U(2, i-1)$). Thus, without maintaining lrp, we can support rank, select, findopen, and findclose queries on lrp in $O(1)$ time by storing $o(n)$-bit auxiliary structures [3, 10]. With the information provided by lrp and U, we can compute $\mathsf{findopen}(i)$ and $\mathsf{findclose}(i)$ operations on lp in $O(1)$ time as follows: To compute $\mathsf{findopen}(i)$, we compute $i_1 - \mathsf{rank}_U(2, i_1 - 1)$, where i_1 is the position of the matching "(" corresponding to $\mathsf{lrp}[i + \mathsf{rank}_U(2, i)]$. For computing $\mathsf{findclose}(i)$, we similarly compute $i_2 - \mathsf{rank}_U(2, i_2)$, where i_2 corresponds to the position of the ")" corresponding to $\mathsf{lrp}[i + \mathsf{rank}_U(2, i-1)]$. Likewise, we can compute $\mathsf{findopen}(i)$ and $\mathsf{findclose}(i)$ operations on rp by locating the matching "{" or "}" in lrp. In summary, our representation enables findopen and findclose operations on both lp and rp to be supported in $O(1)$ time without storing them explicitly.

Now we show that our representation is valid, i.e., we can decode π from the representation.

▶ **Theorem 4.** *The strings* lr *and* E *give a* $(3n + o(n))$*-bit representation for the Baxter permutation* $\pi = (\pi(1), \ldots, \pi(n))$ *of size* n.

Proof. It is enough to show that the representation can decode $\mathsf{MinC}(\pi)$ along with the associated labels. For each non-root node $\phi(i)$, we can check $\phi(i)$ is either a left or right child of its parent by referring $\mathsf{lr}[i-1]$. Thus, it is enough to show that the representation can decode the label of the parent of $\phi(i)$. Without loss of generality, suppose $\phi(i)$ is a left child of its parent (the case that $\phi(i)$ is a right child of its parent is analogous). Utilizing the two-stack based algorithm and referring to Lemma 3, we can proceed as follows: If no element is removed from the L stack after traversing $\phi(i-1)$ (this can be checked by referring $\mathsf{lr}[i]$ and $E[i]$), we can conclude that the parent node of $\phi(i)$ is indeed $\phi(i-1)$. Otherwise, the parent of $\phi(i)$ is the node labeled with $\mathsf{findopen}(i-1)$ on lp from the two-stack based algorithm. ◀

▶ **Example 5.** Figure 2 shows the representation of the Baxter permutation $\pi = (9, 8, 10, 1, 7, 4, 5, 6, 2, 3, 11)$. Using the representation, we can access $\mathsf{lp}[3] =$"$)$" by referring $\mathsf{lr}[3] = l$ and $E[3] = 2$ by Lemma 3. Also, $\mathsf{findopen}(6)$ on lp computed by (1) computing the position of the matching "$($" of the parenthesis of lrp at the position $i' = 6 + \mathsf{rank}_U(2, 6) = 8$, which is 5, and (2) returning $5 - \mathsf{rank}_U(2, 5-1) = 4$. Note that lrp is not explicitly stored. Finally, we can decode the label of the parent of $\phi(4)$ using $\mathsf{findopen}(3)$ on lp ($\phi(4)$ is the left child of its parent since $\mathsf{lr}[3] = l$), resulting in the value 2. Thus, $\phi(2)$ is the parent of $\phi(4)$.

Representation of alternating Baxter Permutation. Assuming π is an alternating permutation of size n, one can ensure that $\mathsf{MinC}(\pi)$ always forms a full binary tree by introducing, at most, two dummy elements $n+1$ and $n+2$, and adding them to the leftmost and rightmost positions of π, respectively [11, 12]. Specifically, we add the node $\phi(i+1)$ as the leftmost leaf of $\mathsf{MinC}(\pi)$ if $\pi(1) < \pi(2)$, Similarly, we add the node $\phi(i+2)$ as the rightmost leaf of $\mathsf{MinC}(\pi)$ if $\pi(n-1) > \pi(n)$.

Since no node in $\mathsf{MinC}(\pi)$ has exactly one child in this case, we can optimize the string E in the representation of Theorem 4 into a binary sequence of size at most $n-1$, where $E[i]$ indicates whether the node $\phi(i)$ is a leaf node or not. Thus, we can store π using at most $2n + o(n)$ bits. We summarize the result in the following corollary.

▶ **Corollary 6.** *The strings* lr *and* E *give a* $(2n + o(n))$*-bit representation for the alternating Baxter permutation* $\pi = (\pi(1), \ldots, \pi(n))$ *of size* n*.*

4 Computing the BP sequence of Cartesian trees

Let π be a Baxter permutation of size n. In this section, we describe how to to compute $\pi(i)$ and $\pi^{-1}(j)$ for $i, j \in \{1, 2, \ldots, n\}$ using the representation of Theorem 4. First in Section 4.1 we modify Cartesian trees so that inorders are assigned to all the nodes. Then we show in Section 4.2 we can obtain the BP sequence of $\mathsf{MinC}(\pi)$ from our representation. By storing the auxiliary data structure of [30], we can support tree navigational operations in Section 2. Finally, in Section 4.4, we show that our data structure can also support the tree navigational queries on $\mathsf{MaxC}(\pi)$ efficiently, which used in the results in the succinct representations of mosaic floorplans and plane bipolar orientations.

To begin discussing how to support $\pi(i)$ and $\pi^{-1}(j)$ queries, we will first show that the representation of Theorem 4 can efficiently perform a depth-first traversal on $\mathsf{MinC}(\pi)$ using its labels. We will establish this by proving the following lemma, which shows that three key operations, namely (1) $\mathsf{left\_child\_label}(i)$: returns the label of the left child of $\phi(i)$, (2) $\mathsf{right\_child\_label}(i)$: returns the label of the right child of $\phi(i)$, and (3) $\mathsf{parent\_label}(i)$: returns the label of the parent of $\phi(i)$ on $\mathsf{MinC}(\pi)$, can be supported in $O(1)$ time.

▶ **Lemma 7.** *The representation of Theorem 4 can support* $\mathsf{left\_child\_label}(i)$*,* $\mathsf{right\_child\_label}(i)$*, and* $\mathsf{parent\_label}(i)$ *in* $O(1)$ *on* $\mathsf{MinC}(\pi)$ *in* $O(1)$ *time.*

Proof. The proof of Theorem 4 shows how to support $\mathsf{parent\_label}(i)$ in $O(1)$ time. Next, to compute $\mathsf{left\_child\_label}(i)$, it is enough to consider the following two cases according to Lemma 3: (1) If $\mathsf{lr}[i] = l$ and $\mathsf{lp}[i]$ is undefined, $\mathsf{left\_child\_label}(i)$ is $i+1$, and (2) if $\mathsf{lr}[i] = r$ and $\mathsf{lp}[i] =$ '$($', we can compute $\mathsf{left\_child\_label}(i)$ in $O(1)$ time by returning $\mathsf{findclose}(i)$ on lp. Similarly, $\mathsf{right\_child\_label}(i)$ can be computed in $O(1)$ time using lr and rp analogously. ◀

Now we can compute $\phi(i+1)$ from $\phi(i)$ without using the two stacks in $O(1)$ time. We denote this operation by $\mathsf{next}(i)$.

1. If $\mathsf{lr}[i] = l$ and the left child of $\phi(i)$ exists, $\mathsf{next}(i)$ is the left child of $\phi(i)$.
2. If $\mathsf{lr}[i] = r$ and the right child of $\phi(i)$ exists, $\mathsf{next}(i)$ is the right child of v.
3. If $\mathsf{lr}[i] = l$ and the left child of $\phi(i)$ does not exist, $\mathsf{next}(i)$ is the left child of $\phi(j)$ where $j = \mathsf{findclose}(i)$ on lp.
4. If $\mathsf{lr}[i] = r$ and the right child of $\phi(i)$ does not exist, $\mathsf{next}(i)$ is the left child of $\phi(j)$ where $j = \mathsf{findclose}(i)$ on rp.

4.1 Computing inorders

First, we define the inorder of a node in a binary tree. Inorders of nodes are defined recursively as follows. We first traverse the left subtree of the root node and give inorders to the nodes in it, then give the inorder to the root, and finally traverse the right subtree of the root node and give inorders. In [30], inorders are defined for only nodes with two or more children. To apply their data structures to our problem, we modify a binary tree as follows. For each leaf, we add two dummy children. If a node has only right child, we add a dummy left child. If a node has only left child, we add a dummy right child. Then in the BP sequence B of the modified tree, i-th occurrence of ")(" corresponds to the node with inorder i. Therefore we can compute rank and select on ")(" in constant time using the data structure of [30] if we store the BP sequence B of the modified tree explicitly. However, if we do so, we cannot achieve a succinct representation of a Baxter permutation. We implicitly store B. The details are explained next.

4.2 Implicitly storing BP sequences

We first construct B for $\mathsf{MinC}(\pi)$ and auxiliary data structures of [30] for tree navigational operations. In their data structures, B is partitioned into blocks of length ℓ for some parameter ℓ, and search trees called *range min-max trees* are constructed on them. In the original data structure, blocks are stored explicitly, whereas in our data structure, they are not explicitly stored and temporarily computed from our representation. If we change the original search algorithm so that an access to an explicitly stored block is replaced with decoding the block from our representation, we can use the range min-max trees as a black box, and any tree navigational operation works using their data structure. Because the original algorithms have constant query time, they do a constant number of accesses to blocks. If we can decode a block in t time, A tree navigational operation is done in $O(t)$ time. Therefore what remains is, given a position of B, to extract a block of ℓ bits.

Given the inorder of a node, we can compute its label as follows. For each block, we store the following. For the first bit of the block, there are four cases: (1) it belongs to a node in the Cartesian tree. (2) it belongs to two dummy children for a leaf in the Cartesian tree. (3) it belongs to the dummy left child of a node. (4) it belongs to the dummy right child of a node. We store two bits to distinguish these cases. For case (1), we store the label and the inorder of the node using $\log n$ bits, and the information that the parenthesis is either open or close using 1 bit. For case (2), we store the label and the inorder of the parent of the two dummy children, and the offset in the pattern "(()())" of the first bit in the block. For cases (3) and (4), we store the label and the inorder of the parent of the dummy node and the offset in the pattern "()".

To extract a block, we first obtain the label of the first non-dummy node in the block. Then from that node, we do a depth-first traversal using $\mathsf{left\_child\_label}(i)$, $\mathsf{right\_child\_label}(i)$, and $\mathsf{parent\_label}(i)$, and compute a sub-sequence of B for the block. During the traversal, we also recover other dummy nodes. Because the sub-sequence is of length ℓ, there are $O(\ell)$

nodes and it takes $O(\ell)$ time to recover the block. To compute an inorder rank and select, we use a constant number of blocks. Therefore it takes $O(\ell)$ time. The space complexity for additional data structure is $O(n \log n/\ell)$ bits. If we choose $\ell = \omega(\log n)$, the space is $o(n)$.

To support other tree operations including RMin, NSV, and PSV queries on π, we use the original auxiliary data structures of [30]. The space complexity is also $O(n \log n/\ell)$ bits.

4.3 Converting labels and inorders

For the minimum Cartesian tree $\mathsf{MinC}(\pi)$ of Baxter permutation π, the label of the node with inorder i is $\pi(i)$. The inorder of the node with label j is denoted by $\pi^{-1}(j)$.

We showed how to compute the label of the node with given inorder i above. This corresponds to computing $\pi(i)$. Next we consider given label j, to compute the inorder $i = \pi^{-1}(j)$ of the node with label j. Note that $\pi(i) = j$ and $\pi^{-1}(j) = i$ hold.

We use $\mathsf{next}(\cdot)$ to compute the inorder of the node with label j. Assume $i\ell+1 \le j < (i+1)\ell$. We start from the node $\phi(i\ell+1)$ with label $i\ell+1$ and iteratively compute $\mathsf{next}(\cdot)$ until we reach the node with label j. Therefore for $i = 0, 1, \ldots, n/\ell$, we store the positions in the modified BP sequence for nodes $\phi(i\ell+1)$ using $O(n \log n/\ell)$ bits. If $\mathsf{next}(i\ell+k)$ is a child of $\phi(i\ell+k)$, we can compute its position in the modified BP sequence using the data structure of [30]. If $\mathsf{next}(i\ell+k)$ is not a child of $\phi(i\ell+k)$, we first compute $p = \mathsf{findclose}(i\ell+k)$ on lp or rp. A problem is how to compute the node $\phi(p)$ and its inorder. To compute the inorder of $\phi(p)$, we use *pioneers* of the BP sequence [19]. A pioneer is an open or close parenthesis whose matching parenthesis belongs to a different block. If there are multiple pioneers between two blocks, only the outermost one is a pioneer. The number of pioneers is $O(n/\ell)$ where ℓ is the block size. For each pioneer, we store its position in the BP sequence. Therefore the additional space is $O(n \log n/\ell)$ bits. Consider the case we obtained $p = \mathsf{findclose}(v)$. If v is a pioneer, the inorder of $\phi(p)$ is stored. If v is not a pioneer, we go to the pioneer that tightly encloses v and $\phi(p)$, obtain its position in the BP sequence, and climb the tree to $\phi(p)$. Because $\phi(p)$ and the pioneer belong to the same block, this takes $O(\ell)$ time. Computing a child also takes $O(\ell)$ time. We repeat this $O(\ell)$ times until we reach $\phi(j)$. Therefore the time complexity for converting the label of a node to its inorder takes $O(\ell^2)$ time. The results are summarized as follows.

▶ **Theorem 8.** *For a Baxter permutation π of size n, $\pi(i)$ and $\pi^{-1}(j)$ can be computed in $O(\ell)$ time and $O(\ell^2)$ time, respectively, using a $3n + O(n \log n/\ell)$ bit data structure. This is a succinct representation of a Baxter permutation if $\ell = \omega(\log n)$. The data structure also can support the tree navigational queries in Section 2 on $\mathsf{MinC}(\pi)$, RMin, PSV, and NSV queries in $O(\ell)$ time.*

Note that Theorem 8 also implies that we can obtain the $(2n + o(n))$-bit succinct data structure of an alternating Baxter permutation of size n that support $\pi(i)$ and $\pi^{-1}(j)$ can be computed in $O(\ell)$ time and $O(\ell^2)$ time, respectively, for any $\ell = \omega(\log n)$.

4.4 Navigation queries on Maximum Cartesian trees

In this section, we show the representation of Theorem 4 can also support the tree navigational queries on $\mathsf{MaxC}(\pi)$ in the same time as queries on $\mathsf{MinC}(\pi)$, which will be used in the succinct representations of mosaic floorplans and plane bipolar orientations.

Note that we can traverse the nodes in $\mathsf{MaxC}(\pi)$ according to the decreasing order of their labels, using the same two-stack based algorithm as described in Section 3. Now, let $\phi'(i)$ represent the node in $\mathsf{MaxC}(\pi)$ labeled with i. We then define sequences lr and E

on $\mathsf{MaxC}(\pi)$ in a manner analogous to the previous definition (we denote them as lr', and E', respectively). The only difference is that the value of i-th position of these sequences corresponds to the node $\phi'(n-i+1)$ instead of $\phi(i)$, since we are traversing from the node with the largest label while traversing $\mathsf{MaxC}(\pi)$. Then by Theorem 4 and 8, it is enough to show how to decode any $O(\log n)$-size substring of lr' and E' from lr and E, respectively.

We begin by demonstrating that for any $i \in [1, \ldots, n-1]$, the value of $\mathsf{lr}'[i]$ is l if and only if $\mathsf{lr}[n-i]$ is r. As a result, our representation can decode any $O(\log n)$-sized substring of lr' in constant $O(1)$ time. Consider the case where $\mathsf{lr}[i]$ is l (the case when $\mathsf{lr}[i] = r$ is handled similarly). In this case, according to the two-stack based algorithm, $\phi(i+1)$ is the left child of $\phi(i_1)$, where $i_1 \leq i$. Now, we claim that $\phi'(i)$ is the right child of its parent. Suppose, for the sake of contradiction, that $\phi'(i)$ is a left child of $\phi'(i_2)$. Then $\phi(i+1)$ cannot be an ancestor of $\phi(i)$, as there are no labels between $i+1$ and i. Thus, $i_2 > i+1$, and there must exist a lowest common ancestor of $\phi'(i)$ and $\phi'(i+1)$ (denoted as $\phi'(k)$). At this point, $\phi'(i+1)$ and $\phi'(i_2)$ reside in the left and right subtrees rooted at $\phi'(k)$, respectively. Now $i_3 \leq i$ be a leftmost leaf of the subtree rooted at $\phi'(i)$. Then there exists a pattern 2–41—3 induced by $(i+1) - k$ and $i_3 - i_2$, which contradicts the fact that π is a Baxter permutation.

Next, we show that the following lemma implies that the representation can also decode any $O(\log n)$-size substring of E' in $O(1)$ time from E along with $\pi(1)$ and $\pi(n)$.

▶ **Lemma 9.** *Given a permutation π, $\phi(i)$ has a left child if and only if $\pi^{-1}(i) > 1$ and $\pi(\pi^{-1}(i) - 1) > i$. Similarly, $\phi(i)$ has a right child if and only if $\pi^{-1}(i) < n$ and $\pi(\pi^{-1}(i) + 1) > i$.*

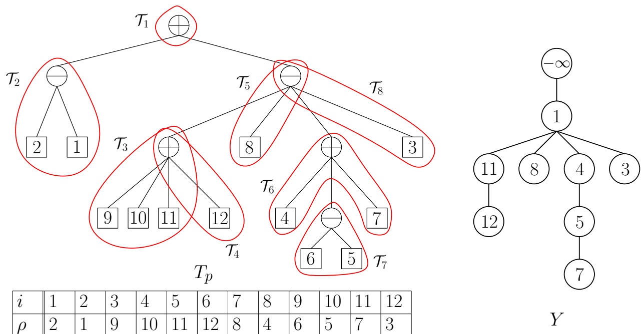
Proof. We only prove that $\phi(i)$ has a left child if and only if $\pi^{-1}(i) > 1$ and $\pi(\pi^{-1}(i)-1) < i$ (the other statement can be proved using the same argument). Let i_1 be $\pi(\pi^{-1}(i)-1)$. From the definition of the minimum Cartesian tree, if $\phi(i_1)$ is at the left subtree of $\phi(i)$, it is clear that $i_1 > i$. Now, suppose $i_1 > i$, but $\phi(i)$ does not have a left child. In this case, $\phi(i)$ cannot be an ancestor of $\phi(i_1)$. Thus, there must exist an element in π positioned between i_1 and i, which contradicts the fact that they are consecutive elements. ◀

As a conclusion, the data structure of Theorem 8 can support the tree navigational queries in Section 2 on $\mathsf{MaxC}(\pi)$, and RMax, PSV, and NSV queries in $\omega(\log n)$ time using $o(n)$-bit auxiliary structures from the results in Section 4.2. We summarize the results in the following theorem.

▶ **Theorem 10.** *For a Baxter permutation π of size n, The succinct data structure of Theorem 8 on π can support the tree navigational queries in Section 2 on $\mathsf{MaxC}(\pi)$, RMax, PLV, and NLV queries in $O(f_1(n))$ time for any $f_1(n) = \omega(\log n)$.*

5 Succinct Data Structure of Separable Permutation

In this section, we present a succinct data structure for a separable permutation $\rho = (\rho(1), \ldots, \rho(n))$ of size n that supports $\rho(i)$ and $\rho^{-1}(j)$ in $O(1)$ time. The main idea of the data structure is as follows. It is known that for the separable permutation ρ, there exists a unique *separable tree* ($v-h$ *tree*) T_ρ of n leaves [7,33], which will be defined later. Since T_ρ is a labeled tree with at most $2n-1$ nodes, $O(n \log n)$ bits are necessary to store T_ρ explicitly. Instead, we store it using a tree covering where each micro-tree of T_ρ is stored as an index of the precomputed table that maintains all separable permutations whose separable trees have at most ℓ_2 nodes, where ℓ_2 is a parameter of the size of the micro-tree of T_ρ, which will be decided later. After that, we show how to support the queries in Theorem 8 and 10 in $O(1)$ time using the representation, along with $o(n)$-bit auxiliary structures.



i	1	2	3	4	5	6	7	8	9	10	11	12
ρ	2	1	9	10	11	12	8	4	6	5	7	3
B	1	0	1	1	1	0	1	1	1	0	0	1

S_ρ = 1 9 12 8 4 5 7 3
ρ' = 1 11 12 8 4 5 7 3

Figure 3 An example of the representation of the separable permutation $\rho = (2, 1, 9, 10, 11, 12, 8, 4, 6, 5, 7, 3)$. Each tree within the red area represents a mini-tree of T_ρ with $\ell_1 = 3$.

5.1 Succinct Representation

Given a separable permutation ρ of size n, the separable tree T_ρ of ρ is an ordered tree with n leaves defined as follows [33]:

- Each non-leaf node of T_ρ is labeled either $\oplus$ or $\ominus$. We call a $\oplus$ node as an internal node labeled with $\oplus$, and similarly, a $\ominus$ node as an internal node labeled with $\ominus$.
- The leaf node of T_ρ whose leaf rank (i.e., the number of leaves to the left) i has a label $\rho(i)$. In the rest of this section, we refer to it as the leaf $\rho(i)$.
- Any non-leaf child of $\oplus$ node is a $\ominus$ node. Similarly, any non-leaf child of $\ominus$ node is a $\oplus$ node.
- For any internal node $p \in T_\rho$, let ρ_p be a sequence of the labels of p's children from left to right, as replacing the label of non-leaf child of p to the label of the leftmost leaf node in the rooted subtree at the node. Then if p is a $\oplus$ (resp. $\ominus$ node), ρ_p is an increasing (resp. decreasing) subsequence of ρ.

See Figure 3 for an example. Szepienic and Otten [33] showed that for any separable permutation of size n, there exists a unique separable tree of it with n leaves.

We maintain ρ through the tree covering algorithm applied to T_ρ, with the parameters for the sizes of mini-trees and micro-trees as $\ell_1 = \log^2 n$ and $\ell_2 = \frac{\log n}{6}$, respectively. Here, the precomputed table maintains all possible separable permutations whose corresponding separable trees have at most ℓ_2 nodes. Additionally, two special cases are considered: when the micro-tree is a singleton $\oplus$ or $\ominus$ node. Since any separable permutation stored in the precomputed table has a size at most ℓ_2, there exist $o(n)$ indices in the precomputed table.

The micro-trees of T_ρ are stored as their corresponding indices in the precomputed table, using $n \log(3 + 2\sqrt{2}) + o(n) \simeq 2.54n + o(n)$ bits in total. In the full version of the paper [8], we consider how to support $\rho(i)$ and $\rho^{-1}(j)$ in $O(1)$ time. The data structure also supports RMin, RMax, PSV, PLV, NSV, and NLV on ρ in $O(\log \log n)$ time.

6 Future Work

We conclude with the following concrete problems for possible further work in the future: (1) Can we improve the query times of π and π^{-1} for Baxter permutations? (2) can we show any time/space trade-off lower bound for Baxter permutation similar to that of general permutation [20]? and (3) are there any succinct data structures for other pattern-avoiding permutations?

References

1 Eyal Ackerman, Gill Barequet, and Ron Y. Pinter. A bijection between permutations and floorplans, and its applications. *Discret. Appl. Math.*, 154(12):1674–1684, 2006. doi:10.1016/J.DAM.2006.03.018.

2 Glen Baxter. On fixed points of the composite of commuting functions. *Proceedings of the American Mathematical Society*, 15(6):851–855, 1964.

3 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms (TALG)*, 11(4):1–21, 2015. doi:10.1145/2629339.

4 Guy E. Blelloch and Arash Farzan. Succinct representations of separable graphs. In *CPM*, volume 6129 of *Lecture Notes in Computer Science*, pages 138–150. Springer, 2010. doi:10.1007/978-3-642-13509-5_13.

5 Miklós Bóna. *Combinatorics of Permutations, Second Edition*. Discrete mathematics and its applications. CRC Press, 2012.

6 Nicolas Bonichon, Mireille Bousquet-Mélou, and Éric Fusy. Baxter permutations and plane bipolar orientations. *Electron. Notes Discret. Math.*, 31:69–74, 2008. doi:10.1016/j.endm.2008.06.011.

7 Prosenjit Bose, Jonathan F. Buss, and Anna Lubiw. Pattern matching for permutations. *Inf. Process. Lett.*, 65(5):277–283, 1998. doi:10.1016/S0020-0190(97)00209-3.

8 Sankardeep Chakraborty, Seungbum Jo, Geunho Kim, and Kunihiko Sadakane. Succinct data structures for baxter permutation and related families, 2024. arXiv:2409.16650.

9 Sankardeep Chakraborty, Seungbum Jo, Kunihiko Sadakane, and Srinivasa Rao Satti. Succinct data structures for bounded clique-width graphs. *Discret. Appl. Math.*, 352:55–68, 2024. doi:10.1016/J.DAM.2024.03.016.

10 Richie Chih-Nan Chuang, Ashim Garg, Xin He, Ming-Yang Kao, and Hsueh-I Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. In *Automata, Languages and Programming: 25th International Colloquium, ICALP'98 Aalborg, Denmark, July 13–17, 1998 Proceedings 25*, pages 118–129. Springer, 1998. doi:10.1007/BFB0055046.

11 Robert Cori, Serge Dulucq, and Gérard Viennot. Shuffle of parenthesis systems and baxter permutations. *Journal of Combinatorial Theory, Series A*, 43(1):1–22, 1986. doi:10.1016/0097-3165(86)90018-X.

12 Serge Dulucq and Olivier Guibert. Stack words, standard tableaux and baxter permutations. *Discrete Mathematics*, 157(1-3):91–106, 1996. doi:10.1016/S0012-365X(96)83009-3.

13 Serge Dulucq and Olivier Guibert. Baxter permutations. *Discrete Mathematics*, 180(1-3):143–156, 1998. doi:10.1016/S0012-365X(97)00112-X.

14 A. Farzan and J. I. Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, January 2014. doi:10.1007/S00453-012-9664-0.

15 Stefan Felsner, Éric Fusy, Marc Noy, and David Orden. Bijections for baxter families and related objects. *J. Comb. Theory, Ser. A*, 118(3):993–1020, 2011. doi:10.1016/J.JCTA.2010.03.017.

16 Johannes Fischer. Combined data structure for previous- and next-smaller-values. *Theor. Comput. Sci.*, 412(22):2451–2456, 2011. doi:10.1016/J.TCS.2011.01.036.

17 Éric Fusy. Straight-line drawing of quadrangulations. In Michael Kaufmann and Dorothea Wagner, editors, *Graph Drawing, 14th International Symposium, GD 2006, Karlsruhe, Germany, September 18-20, 2006. Revised Papers*, volume 4372 of *Lecture Notes in Computer Science*, pages 234–239. Springer, 2006. doi:10.1007/978-3-540-70904-6_23.

18 Paweł Gawrychowski and Patrick K Nicholson. Optimal encodings for range top-k k, selection, and min-max. In *Automata, Languages, and Programming: 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I 42*, pages 593–604. Springer, 2015. doi:10.1007/978-3-662-47672-7_48.

19 Richard F. Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006. Combinatorial Pattern Matching. doi:10.1016/J.TCS.2006.09.014.

20 Alexander Golynski. Cell probe lower bounds for succinct data structures. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 625–634. SIAM, 2009. doi:10.1137/1.9781611973068.69.

21 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

22 Bryan Dawei He. A simple optimal binary representation of mosaic floorplans and baxter permutations. *Theoretical Computer Science*, 532:40–50, 2014. doi:10.1016/J.TCS.2013.05.007.

23 Xianlong Hong, Gang Huang, Yici Cai, Jiangchun Gu, Sheqin Dong, Chung-Kuan Cheng, and Jun Gu. Corner block list: An effective and efficient topological representation of non-slicing floorplan. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD-2000. IEEE/ACM Digest of Technical Papers (Cat. No. 00CH37140)*, pages 8–12. IEEE, 2000. doi:10.1109/ICCAD.2000.896442.

24 Seungbum Jo and Geunho Kim. Space-efficient data structure for next/previous larger/smaller value queries. In *LATIN*, volume 13568 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 2022. doi:10.1007/978-3-031-20624-5_5.

25 Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout.* Springer Science & Business Media, 2012.

26 J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

27 J Ian Munro, Rajeev Raman, Venkatesh Raman, et al. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012. doi:10.1016/J.TCS.2012.03.005.

28 J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

29 Masahiro Nakano, Akisato Kimura, Takeshi Yamada, and Naonori Ueda. Baxter permutation process. *Advances in Neural Information Processing Systems*, 33:8648–8659, 2020.

30 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):1–39, 2014. doi:10.1145/2601073.

31 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4):43–es, 2007.

32 Zion Cien Shen and Chris CN Chu. Bounds on the number of slicing, mosaic, and general floorplans. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(10):1354–1361, 2003. doi:10.1109/TCAD.2003.818136.

33 Antoni A. Szepieniec and Ralph H. J. M. Otten. The genealogical approach to the layout problem. In *DAC*, pages 535–542. ACM/IEEE, 1980. doi:10.1145/800139.804582.

34 Roberto Tamassia and Ioannis G. Tollis. A unified approach a visibility representation of planar graphs. *Discret. Comput. Geom.*, 1:321–341, 1986. doi:10.1007/BF02187705.

35 Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980. doi:10.1145/358841.358852.

36 Bo Yao, Hongyu Chen, Chung-Kuan Cheng, and Ronald L. Graham. Floorplan representations: Complexity and connections. *ACM Trans. Design Autom. Electr. Syst.*, 8(1):55–80, 2003. doi:10.1145/606603.606607.

37 Xiaoke Zhu, Changwen Zhuang, and Y. Kajitani. A general packing algorithm based on single-sequence. In *2004 International Conference on Communications, Circuits and Systems*, volume 2, pages 1257–1261 Vol.2, July 2004. doi:10.1109/ICCCAS.2004.1346402.

38 Changwen Zhuang, Xiaoke Zhu, Y. Takashima, S. Nakatake, and Y. Kajitani. An algorithm for checking slicing floorplan based on hpg and its application. In *2004 International Conference on Communications, Circuits and Systems (IEEE Cat. No.04EX914)*, volume 2, pages 1223–1227 Vol.2, 2004. doi:10.1109/ICCCAS.2004.1346395.