Snake in Optimal Space and Time

Philip Bille
Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark

Martín Farach-Colton
New York University, NY, USA

Inge Li Gørtz
Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark

Ivor van der Hoog
Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark

Abstract

We revisit the classic game of Snake and ask the basic data structural question: how many bits does it take to represent the state of a snake game so that it can be updated in constant time? Our main result is a data structure that uses optimal space (within constant factors). To achieve our results, we introduce several interesting data structural techniques, including a decomposition technique for the problem, a tabulation scheme for encoding small subproblems, and a dynamic memory allocation scheme.

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

Keywords and phrases Data structure, Snake, Nokia, String Algorithms

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.3

Funding *Philip Bille*: Supported by the Independent Research Fund Denmark (DFF-9131-00069B and 10.46540/3105-00302B).
Inge Li Gørtz: Supported by the Independent Research Fund Denmark (DFF-9131-00069B and 10.46540/3105-00302B).
Ivor van der Hoog: This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 899987.

1 Introduction

The classic game of *Snake* involves players navigating a grid with a growing line (themed as a snake) while avoiding collisions with itself and the boundary. As the line increases in length, the game gets progressively harder. The game originated in the 1976 arcade video game Blockade and later evolved into a single-player version (see the history overview of the game [11]). In 1988, the game largely became responsible for introducing mobile phone gaming to the world after being included with the Nokia 6110 cellular phone [12]. Subsequently, the Snake game has appeared in several different versions on Nokia phones and elsewhere. From a CS perspective, the Snake game has been studied in the context of motion-planning problems [9] and deep learning [1, 8, 10]. Surprisingly, the game has not been studied from a data structural perspective. In this paper, we ask the basic question: how many bits does it take to represent the state of a snake game so that it can be updated in constant time? Our main result uses asymptotically optimal bits of space and constant time per operation. To achieve our results, we introduce several interesting data structural techniques, including a decomposition technique for the problem, a tabulation scheme for encoding small subproblems, and a dynamic memory allocation scheme.


© Philip Bille, Martín Farach-Colton, Inge Li Gørtz, and Ivor van der Hoog;
licensed under Creative Commons License CC-BY 4.0

1.1 Setup and Results

Consider a $u \times u$ grid G. A *snake* S is a sequence of n distinct points $s_1, \ldots, s_n$ in G such that any two consecutive points are adjacent in either the vertical or horizontal direction. We call s_n and s_1 the *head* and the *tail* of S, respectively. Our goal is to maintain S dynamically and compactly while supporting the following operations:

- extend(d): where d is a direction (up, down, left, or right). Add a new point to S adjacent to the head in the direction d. If the new point is already a point on the S or outside of G, we terminate and report a collision.
- reduce(): remove the tail from S.

There are two immediate solutions to this problem. The first solution is to explicitly store the grid as a bit string of length u^2 with 1's at all positions containing a point of S and 0's elsewhere. We additionally maintain the positions of the head and tail of S. To implement the operations, we use the bit string to check for collisions and update the bit string and head and tail pointers accordingly. This solution uses $O(u^2 + \log u) = O(u^2)$ bits of space and implements both operations in constant time. Alternatively, we can store the head and tail of S separately plus all points in S in a balanced search tree, where the points in S are sorted in lexicographical order. To implement the operations, we search the tree for the collision check and update the snake by insertions and deletions in the tree.

This solution uses $O(n \log u)$ bits and supports operations in $O(\log n)$ time. Using exponential search trees [2], we can improve the time of this solution to $O(\sqrt{\log n / \log\log n})$, or if we allow randomization, to constant time with high probability [3–7]. All of the above solutions work on a standard word RAM model of computation. Each word can store the location of a coordinate in the grid and hence the word length $w \geq \log u$. In the same model, we show the following result.

▶ **Theorem 1.** *We can represent a snake of length n in $O(n + \log u)$ bits and support* extend *and* reduce *operations in constant time.*

Note that this improves all of the above combinations of time and space bounds. Any solution must use at least $\Omega(n + \log u)$ bits.

Techniques. To obtain the results of Theorem 1, we introduce and combine several interesting data structural techniques. We first present a simple solution using $O(n \log u)$ bits of space and constant time for operation. This solution is based on a partitioning of the grid into square subgrids of size $\log u \times \log u$, called *tiles*. The main component of this structure is a standard balanced binary search tree that stores all of the non-empty tiles (i.e., the tiles intersected by the snake). For each such tile, we then store a bit string of length $\log^2 u$ that encodes the positions of the snake within the tile. This uses $O(n \log u)$ bits of space.

To support the operations in constant time, we show how to efficiently schedule and buffer operations. Intuitively, since the tiles have size $\log u \times \log u$ and the snake only moves a single position in each operation, we can schedule the needed traversals and updates of the balanced binary search tree using constant additional time at each operation.

Next, we extend our solution to add another level of tiles of doubly logarithmic size "nested" within the first level. The new level also maintains balanced binary search trees of non-empty tiles and an encoding of the snake within each such tile. We show how to maintain the structure as above in constant time per operation. Here, the key challenge is achieving and implementing the structure in optimal space. First, we cannot afford to explicitly encode a tile at level 2 as above. Instead, we show how efficiently encode the

snake within these tiles and tabulate the operations. Secondly, we cannot afford to explicitly store $\log u$ pointers between different components of the data structure (such as the balanced binary search trees at the lowest level). Instead, we present a new memory allocation scheme that efficiently packs together components of the data structure such that the entire structure fits on $O(n + \log u)$ bits of space.

2 Fast Snake with $O(n \log u)$ Bits of Space

As a warm-up, we first demonstrate an approach that uses $O(n \log u)$ bits of space and constant time for each operation. Our solution relies upon *tilings* of the grid G.

▶ **Definition 2.** *Given a parameter $\tau > 1$, we partition G into $\lfloor u^2/\tau \rfloor$ squares called* tiles*, of size $\tau \times \tau$ (the bottom row and rightmost column may be smaller). We call this a* τ-tiling *of G. A tile v is* empty *if it does not contain a point of S. Otherwise, it is* non-empty*. The* τ-tile set *for S, denoted by T_S^τ, is the set of non-empty tiles in a τ-tiling of G.*

Since S is a collection of consecutively adjacent points in G, we may immediately conclude:

▶ **Lemma 3.** *For all snakes of length n, for all τ, $|T_S^\tau| \leq 4\lceil \frac{n}{\tau} \rceil$.*

2.1 Data Structure

We construct a tiling for G with parameter $\tau = \log u$. Let S be a snake of length n. Our data structure consists of the following components (Figure 1):

1. A *direction string* D of length $n-1$ that stores for each position in S, except the head h, the direction of its successor. It also stores the *relative coordinates* of h and t.
2. A *head and tail scope*. The head scope stores a pointer to the τ-tile $v \in T$ that contains h, the relative coordinates of the head h in v, and the up to eight non-empty τ-tiles incident to v. The tail scope stores a pointer to each of the τ-tiles intersected by the last $\tau/2$ points of S, the coordinates of the tail t, and the coordinates of the point $t_\tau = s_\tau$. If $n < \tau$ then $t_\tau = h$.
3. A balanced binary search tree B containing the tile set T_S^τ. The tiles in B are ordered by their top-left coordinates in lexicographical order.
4. An *insert and delete* buffer. The insert buffer stores up to the last four τ-tiles that were empty before the head entered them (these tiles are not yet in B). The delete buffer stores up to the first four τ-tiles that became empty after the tail left them (these tiles are not yet deleted from B).
5. For each τ-tile $v \in T_S^\tau$,
   a. A counter recording $|S \cap v|$.
   b. A bit string $E(v)$ that stores a single bit for each position in v, indicating if the position is empty or non-empty.

The direction string D uses $O(n)$ bits, as we encode each direction with two bits. The head and tail scope use $O(\log u)$ bits as they store $O(1)$ pointers. The counters use $O(n)$ bits in total, as the total number of counters is at most $O(n/\tau)$ and each counter is at most $2 \log \tau$ bits. The binary search tree uses $O(n)$ bits as by Lemma 3, $|T_S^\tau| \in O(n/\tau)$ and each pointer uses at most $\tau = \log u$ bits. A single bitstring $E(v)$ uses τ^2 bits as each tile v contains τ^2 points. Thus the total space for the bitstrings is $O(\tau^2(n/\tau)) = O(n \log u)$.

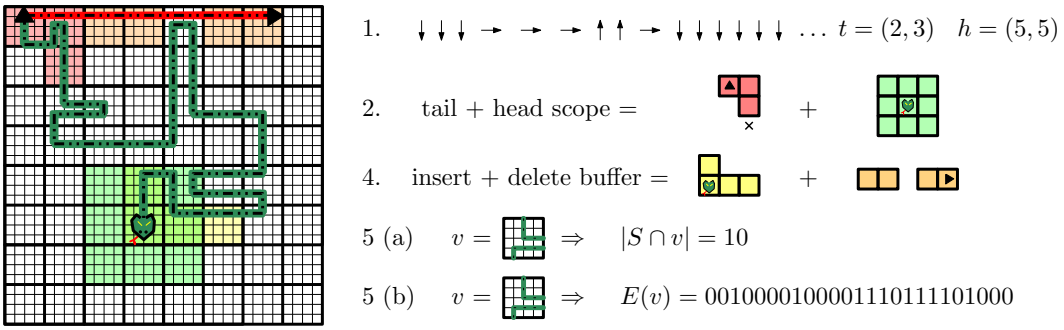

Figure 1 On the left we show a grid G, tiled with τ-tiles for $\tau = 5$. There is a snake in green. The red indicates the previous positions of the snake, which are no longer occupied. On the left, we show the components of our data structure.

2.2 The extend Operation

We now explain how to implement the extend operation. Let h be the head of S before the operation, and let h' be the new position of the head. Also, let v and v' be the τ-tiles in the grid that contain h and h', respectively. We proceed as follows:

Step 1: Check for Collisions. We check if there is a collision of h' with a point on S. Given h, D, and the head scope, we first identify v' and compute the relative coordinates of h' in constant time. We can test whether v' is empty using its pointer to the head scope in $O(1)$ time. There are two cases depending on whether v' is empty or not:

- **(i)** If v' is empty, there is no collision. We then construct $E(v')$ in $O(1)$ time. We set the corresponding counter to one. We then check if v' is in the delete buffer (in this case, we may be in the process of deleting v' from our data structure). If it is and it is not the first tile in there, we remove it from the delete buffer. Otherwise, we add v' to the back of the insert buffer.
- **(ii)** If v' is non-empty, we use $E(v')$ and the relative coordinates of h' in v' to check if there is a collision. If so, we stop and output this. Otherwise, we update $E(v')$ and the corresponding counter in $O(1)$ time.

Finally, we update D in constant time.

Step 2: Update Search Tree and Buffers. We move τ-tiles from the insert buffer into B. To do so, we do a constant amount of work towards inserting the first τ-tile x in the insert buffer into B. We do enough work to ensure that after $\tau/8$ extend and reduce operations, x is fully inserted into B. Since an insertion or a deletion in B takes $O(\tau)$ time, this can be obtained by doing $O(1)$ work for each operation. When the tile is fully inserted into B, we remove it from the insert buffer. Similarly, we do a constant amount of work towards deleting the first τ-tile x in the deletion buffer from B. By Lemma 3, it follows that there can be no more than 4 tiles in each of the buffers at any point in time.

Step 3: Update Scopes. The head scope is a set of 3×3 tiles centered at the τ-tile that contains h. To maintain the invariant that the scope is in memory at all times, we store slightly more than the scope. Consider the set of 7×7 tiles centered at the τ-tile that contains h and denote it by N_h. Each non-empty $x \in N_h$ is either in B (which supports lookups in $O(\tau)$ time) or in the insert/delete buffer (which have constant size). Each time

we perform extend, we do constant work towards a lookup for each tile in N_h. If we finish a lookup and find a τ-tile z, we add z to our head scope list. If a tile $z' \in N_h$ is in the head scope list and $z' \notin N_{h'}$, we delete z' from the list. This way, we maintain that the head scope is always in the head scope list whilst having $O(1)$ τ-tiles in the head scope list at all times. Note that this does not change the asymptotic space of the data structure. By Lemma 3, if $|S| > \tau$, the tail scope remains unchanged. Else, we add v' to the tail scope if $v \neq v'$ and update t_τ.

2.3 The reduce Operation

We show how to implement the reduce operation. Let t be the tail of S before the operation and let v be the τ-tile in the grid that contain t. We proceed as follows:

Step 1: Update Search Trees and Buffers. Given t and the tail scope, we identify v in $O(1)$ time. Given the relative coordinates of t in v, we update $E(v)$ and the corresponding counter in $O(1)$ time. If v is now empty, we first check if it is in the insert buffer. If v is in the insert buffer and it is not the first tile in there, then we remove it from the insert buffer. Otherwise, we add it to the delete buffer. Using t, t_τ and D we update the tail t and t_τ to their new positions. As in the extend operation, we then do a constant amount of work towards inserting the first tile from the insert buffer into B and deleting the first τ-tile x in the delete buffer from B.

Step 2: Update Tail Scope. Let the extended tail scope N_t be the set of tiles intersected by the last τ points of the snake. As with the head scope, we will maintain a set of tiles, including all the tiles in the tail scope and possibly some of the tiles in N_t that are not in the tail scope. Let v' be the τ-tile containing t_τ. If v' is already in our tail scope list, we do nothing. Otherwise, we do a constant amount of work towards the search for each of the tiles in the extended tail scope N_t that are not yet in the tail scope list. We do this fast enough to ensure that we have finished the search for v' when it belongs to the tail scope. When we have finished the search, we add v' to the tail scope list. By Lemma 3, there can be at most 4 tiles in the extended tail scope, and thus, we never search for more than 4 tiles at a time. Note that this does not change the asymptotic space of the data structure. Since a search in B takes $O(\tau)$ time, this can be done with a constant amount of work for each reduce operation.

2.4 Summary

In summary, the space of the data structure is $O(n \log u)$ bits, and each of the operations uses constant time. Hence, we have the following result.

▶ **Lemma 4.** *We can represent a snake S of length n in $O(n \log u)$ bits and support* extend *and* reduce *operations in constant time.*

In the following sections, we improve our solution to obtain the result of Theorem 1. We first show how to efficiently tabulate small subproblems for tiles of size $\log \log n$ in Section 3. In Section 4, we then extend the above solution to a two-level data structure with nested tilings of sizes $\log n$ and $\log \log n$ and apply the tabulation at the lowest level. Unfortunately, naively storing our data structure would use $\log u$ pointers between different components, which we can not afford within our space bound of $O(n + \log u)$ bits. In Section 5, we show how to dynamically allocate all components of the data structure compactly leading to Theorem 1.

3 Tabulation

We now show how to efficiently tabulate τ-tiles for $\tau = \log\log n$. This will be a key component in our multi-level data structure. Intuitively, we consider all possible ways a snake S can intersect a tile v of dimension $\tau \times \tau$, together with all placements of the (future) head of S, and store for each combination the outcome after executing the update. Formally, fix any snake S and let v be any $\tau \times \tau$ square. We define two concepts:

▶ **Definition 5.** *We denote for any $\tau \times \tau$ square v by δv all vertices of the grid that are in v and incident to the boundary of v, in the clockwise direction.*

▶ **Definition 6.** *We define a* marked grid *as any $\tau \times \tau$ square X that marks each point in the tile as either $\{$empty, occupied, head, tail$\}$. For each snake S and τ-tile v we immediately have a corresponding marked grid $S \cap v$. Finally, for any pair of marked grids X, Y, we denote $X \simeq Y$ whenever the two marked grids are identical.*

3.1 Tabulation Code

We define what we call our *tabulation code* $\text{ENC}_S(v)$ which is the concatenation of four strings $\alpha(v), \beta(v), \gamma(v), \epsilon(v)$. To this end, we note that S uniquely corresponds to a rectilinear curve that is obtained by connecting consecutive grid points in S by an edge. For any rectilinear curve S and any square v, we intuitively define the set of maximally connected subcurves in $S \cap v$. We define:

- The string $\alpha(v)$ is defined by traversing the vertices $x \in \delta v$ in order and denoting a 1 whenever $x \in S$ and a zero otherwise.
- The string $\beta(v)$ considers all maximal subcurves $S' \subset S \cap v$ that intersect δv sorted by their first point of intersection with δv. The string $\beta(v)$ concatenates for each maximal subcurve S' a string $\beta(S')$. The string $\beta(S')$ denotes for all $x \in S'$ the direction of its successor by using two bits, followed by a symbol that denotes the end of S'.
- There are at most two subcurves in $S' \subset S \cap v$ that do not intersect δv; these must contain the head and/or tail of S and are considered at the end. If the head and tail of S are not in v then $\gamma(v)$ denotes a *null* symbol. Otherwise, $\gamma(v)$ uses $2\log\tau$ bits to specify the relative position of h (and/or t) in v.
- If the string $\gamma(v)$ is not null, the string $\epsilon(v)$ considers the maximal subcurve S' of $S \cap v$ that contains the head (or tail) and denotes for all $x \in S'$ the direction of its successor by using two bits, followed by a symbol that denotes the end of S'.

▶ **Lemma 7.** *For any snake S and any τ-tile v, $|\text{ENC}_S(v)| \leq 4\tau + 2|S \cap v| \leq 2\tau^2 + 4\tau$ and $\sum_{v \in T_S^\tau} |\text{ENC}_S(v)| \in O(n)$.*

Proof. The strings $\alpha(v)$ and $\gamma(v)$ contain together fewer than 4τ bits. The strings $\beta(v)$ and $\epsilon(v)$ contain fewer than four times the number of points in $S \cap v$. It immediately follows that $\text{ENC}_S(v)$ has fewer than $\tau + 2|S \cap v| \leq 2\tau^2 + 4\tau$ bits. By Lemma 3, there are at most $O(n/\tau)$ non-empty τ-tiles and thus $\sum_{v \in T_S^\tau} |\text{ENC}_S(v)| \in O(n)$. ◀

▶ **Lemma 8.** *For any pair of snakes S, S' and any pair of τ-tiles v, v' if $\text{ENC}_S(v) = \text{ENC}_{S'}(v')$ then $S \cap v \simeq S' \cap v'$.*

Lemma 8 allows us to define the *inverse* of an encoding:

▶ **Definition 9.** *For any string s of at most $4(\tau^2 + \tau)$ bits, we denote by $\text{ENC}^{-1}(s)$ the unique marked grid X such that for all snakes S and τ-tiles v with $\text{ENC}_S(v) = s$, $X \simeq S \cap v$. $\text{ENC}^{-1}(s)$ is NULL whenever there exist no snakes S and τ-tiles v with $\text{ENC}_S(v) = s$.*

3.2 Constructing Tables

Given our encoding scheme, we define tables M_{coll}, M_{head}, and M_{tail} to check for collisions, update the head, and update the tail in constant time:

- Each entry in M_{coll} corresponds to an encoding string s and a position p and stores a single bit that indicates whether or there is a collision with p and $\text{ENC}^{-1}(s)$.
- Each entry in M_{head} corresponds to an encoding string s and a position p adjacent to the head of s and stores the encoding string s' resulting from moving the head to position p.
- Each entry in M_{tail} corresponds to an encoding string s and stores the encoding string s' resulting from reduce the tail.

If an entry to a table is not a valid input, i.e., the encoding string e does encode a valid snake or the position p in M_{head} is not adjacent to the head of the snake, the entry stores an arbitrary value. We use the tables to simulate the extend and reduce operations in constant time. Initially, when we first enter a tile v, we fetch the corresponding initial encoding from the table and store it in the search tree for v. All subsequent new encoding strings for v are obtained from the above tables, and hence, it inductively follows that we only access entries corresponding to valid input in the tables.

The space for the tables is dominated by the space M_{head} that has $2^{4(\tau^2+\tau)+2\log\tau}$ entries each storing $O(\tau^2)$ bits. Thus, in total we use $O(\tau^2 \cdot 2^{(4\tau^2+\tau)}) = O((\log\log n)^2 \cdot 2^{4(\log\log n)^2+\log\log n}) = O(n)$ bits of space. We can compute an entry in $O(\tau^2) = O((\log\log n)^2)$ time, and hence construct the tables in $O((\log\log n)^2 2^{(4(\log\log n)^2+\log\log n)}) = O(n)$ time. We use a standard global rebuilding with deamortization to construct the tables for exponentially increasing values of n as the length of the snake changes. I.e., we assume that $n \in [\frac{1}{2}N, 2N]$ and set $\tau = \log\log N$. Since we can construct the tables in linear time, the overhead of rebuilding our tabulation before n leaves $[\frac{1}{2}N, 2N]$ is constant (see also Section 4.3 for a detailed description of the deamortization).

4 Fast Snake Using Two Levels

We now describe our two-level data structure.

4.1 Data Structure

Let S be a snake of length n. For now, we assume that $n \in [\frac{1}{2}N, 2N]$ for some given N (we show how to lift this assumption later). Let $\tau_1 = \log N$ and $\tau_2 = \log\log N$. We construct tilings $T_1 = T_S^{\tau_1}$ and $T_2 = T_S^{\tau_2}$ for G. Our data structure stores the following components for each T_i where $i \in \{1, 2\}$.

(a) A *direction string* D of length $n-1$ that stores for each position in S, except the head h, the direction of its successor. It also stores the *relative coordinates* of h and t in their τ_i-tiles.

(b) A *head and tail scope*. The head scope stores a pointer to the τ_i-tile $v \in T$ that contains h, the relative coordinates of h in v, and the up to eight non-empty τ_i-tiles incident to v. The tail scope stores a pointer to each of the τ_i-tiles intersected by the last $\tau_i/2$ points of S, the coordinates of the tail t, and the coordinates of the point $t_{\tau_i} = s_{\tau_i}$. If $n < \tau_i$ then $t_{\tau_i} = h$.

(c) An *insert and delete* buffer. The insert buffer stores up to the last four τ_i-tiles that were empty before the head entered them. The delete buffer stores up to the last four τ_i-tiles that became empty after the tail left them.

(d) If $i = 1$, we store for all $v_j \in T_1$ a counter recording $k_j = |S \cap N(v_j)|$ where $N(v_j)$ is the tile v_j plus the at most four tiles immediately above, left, right or below v_j.
If $i = 2$, we store for all $w_l \in T_2$ a counter recording $k'_l = |S \cap w_l|$.

(e) If $i = 1$ we store one balanced binary tree B on all τ_1 tiles in T_1. If $i = 2$ we maintain for each $v \in T_1$ a balanced binary search tree B_v storing $v \cap T_2$ in lexicographical order.

(f) If $i = 1$, we maintain for each $v \in T_1$ a pointer to the set $N(v)$: the tile v plus the at most four non-empty τ_1-tiles immediately above, left, right or below v.

(g) If $i = 2$, we store for each tile $v \in T_2$ the tabulation code $\textsc{enc}_S(v)$.

4.2 Operations

We implement the extend operation as in Section 2.2. Instead of querying and updating an explicit bit string for a tile v on the lowest level we use the corresponding tables and $\textsc{enc}_S(v)$. Note that we can query, update, and replace $\textsc{enc}_S(v)$ in constant time using our tabulation. Since the scope at level 2 is fully included in the scope of level 1, we can maintain the scope at level 2 in constant time using the same technique as in Section 2.2. Similarly, we can implement the reduce operation in constant time.

4.3 Lifting the Assumption

Since our data structure has $O(1)$ update time, we may trivially lift the assumption that $n \in [\frac{1}{2}N, 2N]$ using standard deamortization techniques. Suppose that n starts out as N. If $n = \frac{3}{2}N$, we make a copy S' of the current S and start building a second copy of our data structure on S' with $N' = \frac{3}{2}N$. Whilst n remains greater than $\frac{3}{2}N$ but smaller than $2N$, we perform our updates on S as regular. In the meantime, we do eight extend operations per operation on S per update on S. Each update instruction on S gets additionally recorded in a queue. If we have our data structure on S', we perform eight updates in the queue on S' per operation on S. This way, when $|S| = 2N$, it must be that $S' = S$. We make the copy of our data structure our primary data structure (setting $N \leftarrow N'$). By doing a symmetric procedure for when $|S| < \frac{2}{3}N$, we may always assume that $|S| \in [\frac{1}{2}N, 2N]$.

5 Achieving $O(n + \log u)$ Bits

Naively, pointers take up a word and thus use $O(\log u)$ bits. Even with more clever pointer management dependent on the input size, a collection of pointers that point to n arbitrary objects in memory require $\Omega(\log n)$ bits per pointer for $O(n \log n)$ bits in total. We show, through clever pointer management, that our data structure can be implemented using $\Theta(n + \log u)$ bits instead. This is asymptotically tight, since the input size is n, and all data requires at least one word.

Storing components (a),(b), and (c). For both T_1 and T_2, we store components (a), (b), and (c) in an arbitrary contiguous set of memory. The direction string requires $O(n)$ bits. We require $O(1)$ pointers to this string (specifying its start, end, and the location required by the tail buffer) which take $O(\log u)$ bits. Components (b) and (c) each have constant size, storing these objects plus a pointer to their location thus requires $O(\log u)$ bits and so these components can trivially be stored using $O(n + \log u)$ bits.

Storing components (d),(e), (f) and (g). We store components (d),(e), (f) and (g) through a two-level definition: where we first store the data structure of T_1 and then the data structure on T_2 within it. Our approach is dependent on the following geometric lemma:

▶ **Lemma 10.** *For any snake S, for any $v_j \in T_1$, the number of τ_2-tiles in v_j is at most $4k_j/\tau_2$.*

Proof. We illustrate the proof by Figure 2. Recall that S uniquely corresponds to a rectilinear curve and consider the set Z of maximal subcurves of S in $S \cap v_j$. We can construct a snake S' by connecting all curves Z with a path along the boundary, so that S' intersects a minimal number of grid points. Note that the number of points $|S \cap v_j|$ is at least the number of points in S' since S is a connected curve. We may now apply Lemma 3 to note that:

$$|T_2 \cap v_j| \leq 4|S'|/\tau_2 \leq 4|S \cap N(v_j)|/\tau_2 = 4k_j/\tau_2$$ ◀

▶ **Corollary 11.** *We may store for each v_j, for all $w_l \in T_2 \cap v_j$ a constant number of values of $O(\tau_2) = O(\log \log N)$ bits each, using at most $O(k_j) = O(|S \cap v_j|)$ bits.*

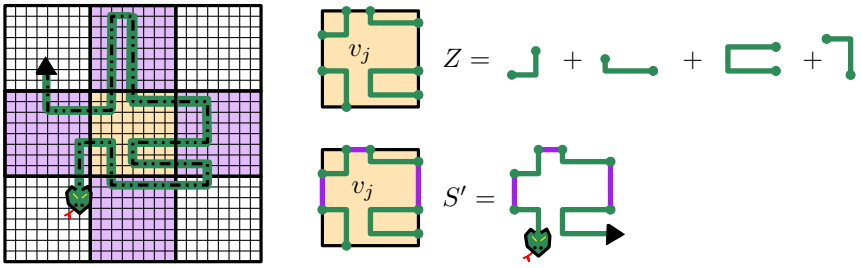

Figure 2 Left we show a snake S intersecting a τ_1-tile v_j in orange. We show $N(v)$ as the orange and purple τ_1-tiles. The set Z are all maximally connected curves in $S \cap v_j$. We construct a snake S' by connecting Z using a minimal number of points.

5.1 Technical overview

Before we state our argument, we first provide a technical overview.

First, we show a static algorithm to store a snake S of size N in $O(N)$ space. We view memory as a contiguous interval of $1000N$ bits denoted by $[1, 1000N]$. We assume that $N > 1000$ (else, we have constant input size and may deploy a trivial polynomial dynamic solution). A consequence of this assumption is that any pointer to a location in $[1, 1000N]$ uses $2 \log N = 2\tau_1$ bits.

Each $v \in T_1$ requires two types of memory as it wants to store:

1. Constantly many pointers and counters of $O(\log N)$ bits each,
2. The set $T_2 \cap v$, which we store using $O(|S \cap N(v)|)$ bits.

By Lemma 3, we have at most N/τ tiles in T_1 and so storing these pointers takes $O(N)$ bits of space. However, storing $T_2 \cap v$ using $O(|S \cap N(v)|)$ bits is considerably more difficult:

Each $w \in T_2 \cap v$ requires two types of memory as it wants to store:

1. Constantly many counters, pointers, and the strings $\alpha_S(w), \gamma_S(w)$ of component (g).
2. The strings $\beta_S(w), \epsilon_S(w)$ of component (g) which have $O(|S \cap w|)$ bits.

If we manage to store $T_2 \cap v$ in at most $O(\log^2 N)$ bits of contiguous memory, pointers to locations within that memory require at most $O(\log \log N)$ bits. We can restrict our counters to have size $O(\log \log N)$ and the strings $\alpha_S(w), \gamma_S(w)$ have $O(\log \log N)$ bits each. It follows

by Corollary 11 that, if we can store $T_2 \cap v$ in a contiguous interval in memory, we can store $w \in T_2 \cap v$ using $O(|S \cap N(v)|)$ bits. Statically, guaranteeing that we store data in contiguous intervals is trivial and this leads to a static algorithm to store our data structure using $O(N)$ bits and time.

From static to dynamic. To dynamically maintain our data structure we use our deamortization rebuilding technique. We execute $\frac{N}{200}$ updates without ever deallocating memory. After $\frac{N}{100}$ updates, we apply the deamortization technique. We set $N' \leftarrow |S|$ and run our static algorithm at 200 operations per update to store our data structure using $O(N')$ bits, at which point we release our previous memory. This way, our analysis only has to show that during $\frac{N}{100}$ extend operations, our data structure still fits within $1000N$ bits. The reduce operation is thus for free. To illustrate this fact, consider for example $\frac{N}{200}$ consecutive reduce updates. Afterwards, $N' = N - \frac{N}{200}$ and so our rebuilt data structure will be considerably smaller.

To make sure that during the first $\frac{N}{100}$ extend operations all data structure components stay within their allotted memory, we recursively apply the deamortization technique. However, to apply this technique we must be very careful: Each tile has two types of objects which require a different order of space. Indeed consider a tile $v \in T_1$ where $O(|S \cap N(v)|)$ is constant. The tile v still requires pointers of size $O(\log N)$ each. If we then double $|S \cap N(v)|$ using $O(1)$ extend operations, we have neither the time nor the space to make a copy of these pointers. So, we split our data into these two types and store (and double) them separately. Similarly, the data corresponding to each $w \in T_2$ must be stored and treated by two separate categories, as they grow at different rates.

Our solution is to partition our memory into three types, indicated by the color red, green and yellow (see Figure 3). Red memory contains all pointers and counters. We only allocate red memory whenever tiles get added to either T_1 or T_2 which requires $\Omega(N/\log N)$ and $\Omega(N/\log\log N)$ extend operations, respectively. Green memory contains data (either the set $T_2 \cap v$ or the strings $\beta_S(w) + \epsilon_S(w)$). We allocate more green memory each extend operation, and must thus carefully to only allocate $O(1)$ data per update. Yellow intervals are search trees that for any tile $v \in T_1$ or $w \in T_2$ can return two pointers: indicating the location of its data in the red and green interval. These pointers again take up non-constant space; and are only allocated during tile inserts.

5.2 A static algorithm to allocate space

Given are $N = |S|$, $\tau_1 = \log N$, $\tau_2 = \log\log N$ and the set $T_1 = (v_1, v_2, \ldots)$ (sorted lexicographical order by their top-left coordinate) where each $v_j \in T_1$ stores:

- A pointer to each τ_1-tile in the set $N(v_j)$.
- An integer counter k_j recording $|S \cap N(v_j)|$,
- The set of τ_2-tiles $v_j \cap T_2$ in lexicographical order where each $w_l \in T_2$ stores:
  - the integer $k_l = |S \cap w_l|$ and the string $\textsc{enc}_S(w_l)$.

Space allocation. We allocate a contiguous set of $1000N$ bits in memory. For brevity, we consider memory as an interval in $\mathbb{R}^1$ and thus our memory is $[1, 1000N]$. For any integer a and interval $[b, c]$ we denote by $a + [b, c]$ the interval $[a + b, a + c]$. We assume that $N > 1000$. Therefore, pointers that point within $[1, 1000N]$ have a size of at most $2\log N$ bits.

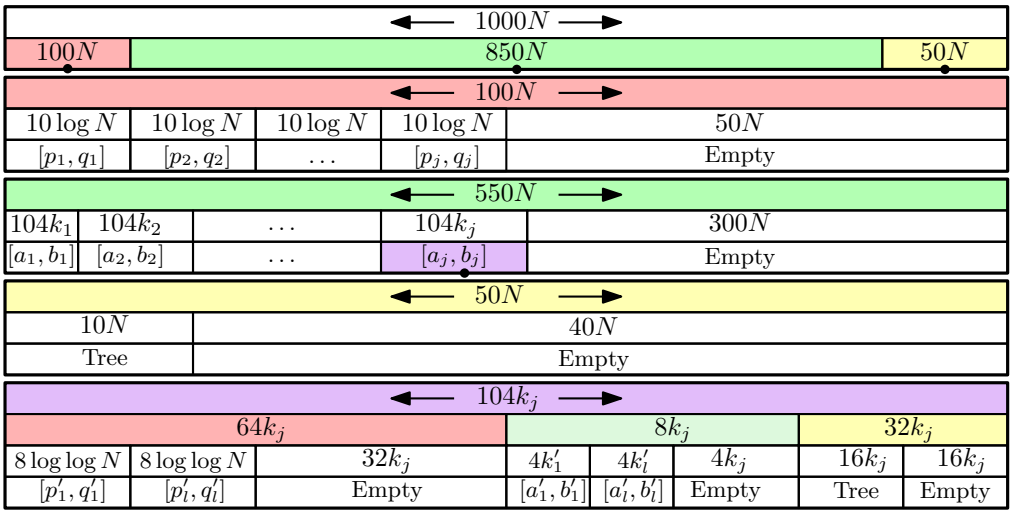

Figure 3 An illustration for how we allocate memory.

We define our memory allocation and then prove that our data structure fits. We store:

1. In $[1, 100N]$ for all $v_j \in T_1$ an interval $[p_j, q_j]$. Specifically, $[p_1, q_1] = [1, 10\log N]$ and $[p_j, q_j] = q_{j-1} + [1, 10\log N]$. $[50N, 100N]$ remains empty. Each $[p_j, q_j]$ stores:
   - Components (d) and (f) of $v_j \in T_1$.
2. In $[100N, 950N]$ for all $v_j \in T_1$ an interval $[a_j, b_j]$. Specifically, $[a_1, b_1] = 100N + [1, 104k_1]$ and $[a_j, b_j] = b_{j-1} + [1, 104k_j]$. The interval $[650N, 950N]$ remains empty. $[a_j, b_j]$ stores:
   - **i.** In $a_j + [1k_j, 32k_j]$ for all $w_l \in T_2 \cap v_j$ an interval $[p'_l, q'_l]$. $[p'_1, q'_1] = a_j + [1, 8\log\log N]$ and $[p'_l, q'_l] = q'_{l-1} + [1, 8\log\log N]$. Each interval $[p'_l, q'_l]$ stores:
     - Component (d) of $w_l \in T_2 \cap v_j$ and from Component (g) the strings $\alpha_S(w_l) + \gamma_S(w_l)$.

     In $a_j + [32k_j, 64k_j]$ we keep empty space (for now).
   - **ii.** In $a_j + [64k_j, 68k_j]$ for all $w_l \in T_2 \cap v_j$ an interval $[a'_l, b'_l]$ of size $4k'_l$. Specifically, $[a'_1, b'_1] = a_j + 64k_j + [1, 4k_j]$ and $[a'_l, b'_l] = b'_{l-1} + [1, 4k_j]$. Each interval $[a'_l, b'_l]$ stores:
     - From component (g) the strings $\beta_S(w_l) + \epsilon_S(w_l)$ plus $2k'_l$ bits of empty space.

     In $a_j + [68k_j, 72k_j]$ we store empty space (for now).
   - **iii.** In $a_j + [72k_j, 88k_j]$ component (e): a balanced binary tree B_{v_j} over $T_1 \cap v_j$.
     - Each node storing w_l also stores a pointer to the location of p'_l and a'_l.

     In $a_j + [88k_j, 104k_j]$ we store empty space (for now).
3. In $[950, 100N]$ component (e): a balanced binary tree B over T_1. Each node storing v_j also stores pointers to the locations of p_j and a_j. $[450N, 500N]$ remains empty.

▶ **Lemma 12.** *Components (d), (e), (f) and (g) fit within their allocated memory. Moreover, given our input as specified we can allocate our data structure in $O(N)$ time.*

Proof. We prove the statement in order:

We allocate intervals $[p_j, q_j]$ for $v_j \in T_1$ of $10\log N$ bits. By Lemma 3, there are at most $4\lceil N/\tau_1 \rceil = 4\lceil N/\log N \rceil$ τ_1-tiles v_j in $T_S^{\tau_1}$. So, $\sum_{v_j \in T_1} 4\log N \leq 40N$ and $\bigcup_j [p_j, q_j] \subset [1, 40N]$.

- Each interval $[p_j, q_j]$ stores components (d) and (f) of v_j. The counter k_j is at most $\log((\tau_1)^2) \leq 2\log N$ bits and four pointers in $[0, 1000N]$ use fewer than $8\log N$ bits.

We allocate intervals $[a_j, b_j]$ of width $104k_j$. $\forall s_i \in S$, there are at most 5 τ_1-tiles with $s_i \in N(v_j)$. Thus, $\sum_j 104k_j = \sum_j 104|S \cap N(v_j)| \leq 5 \cdot \sum_{s_i \in S} 104 \leq 520N$. For each $[a_j, b_j]$:

i. We allocate intervals $[p'_l, q'_l]$ for $w_l \in T_2 \cap v_j$ of $8 \log\log N$ bits. By Lemma 10, there are at most $4k_j/\tau_2 = 4k_j/\log\log N$ τ_2-tiles in $T_2 \cap v_j$. So, $\sum_{w_l \in T_2 \cap v_j} 8\log\log N \leq 32k_j$. Thus $\bigcup[p'_l, q'_l]$ has a width of at most $32k_j$ and an interval of width $32k_j$ remains empty.
- Each interval $[p'_l, q'_l]$ stores component (d) of w'_l and of component (g) $\alpha(w_l) + \gamma(w_l)$. The counter k'_l is at most $\log((\tau_2)^2) \leq 2\log\log N$ bits. By Lemma 7, these strings use $4\log\log N$ bits. So the total takes $6\log\log N < 8\log\log N$ bits.

ii. We allocate $[a'_l, b'_l]$ for $w_l \in T_2 \cap v_j$ of $4k'_l$ bits. $\sum_{w_l \in T_2 \cap v_j} 4k'_l = 4 \sum_{w_l \in T_2 \cap v_j} |S \cap w| \leq 4k_j$. Thus, we may leave an interval of $4k_j$ bits empty.
- By the proof of Lemma 7, the strings $\beta_S(w_l)$ and $\epsilon_S(w_l)$ of $\textsc{enc}_S(w_l)$ use $2k'_l$ bits and so $[a'_l, b'_l]$ has $2k'_l$ empty bits remaining.

iii. By Lemma 10, $|T_2 \cap v_j| \leq 4k_j/\log\log N$. We store component (e), a balanced binary tree B_{v_j} where each node representing w_l stores a pointer to the location of p'_l and a'_l of at most $2\log\log N$ bits. Moreover, $|T_2 \cap v_j| \leq (\log N)^2$. Thus, we may construct a balanced binary tree on $T_2 \cap v_j$ using $16k_j$ bits in total, leaving $16k_j$ bits empty.

By Lemma 3, there are at most $4\lceil N/\tau_1 \rceil = 4\lceil N/\log N \rceil$ τ_1-tiles in T_1. We store component (e) as a balanced binary tree where nodes store two pointers of at most $2\log N$ bits. Thus, the balanced binary tree fits within an interval of width $10N$.

Since the input is already sorted, the whole allocation can trivially be done in $O(N)$ time. ◀

5.3 Dynamically maintaining our space allocation

We dynamically maintain our space allocation as follows. For $\frac{N}{100}$ updates, we never deallocate any space. Thus, the reduce operation has no effect on our storage. For any extend operation during this time, we allocate additional space: filling up the pre-allocated empty space in our memory. Whenever a pre-allocated empty interval is half full, we allocate a new interval of twice the size and start copying the data into the new interval. We prove that this way, we never store data in an interval that is empty. We first introduce some additional concepts:

- Denote for each τ_1-tile $v_j \in T_1$ a counter $K^*_j = [a_j, b_j]$.
  - There are two ways this counter can be stored. Firstly this counter never exceeds $2\log N$ bits and store it alongside k_j in $[p_i, q_j]$ by increasing our space by a small constant factor. If the reader is hesitant towards increasing space any further, this counter may always be computed from $[a_j, b_j]$ in $O(1)$ time.

We describe the compounded effect of extend and reduce operations through five *events:*

1. In the MINI-SPAWN event, we add a τ_2-tile w_l to T_2 with $w_l \subset v_j \in T_1$.
- Denote by $[p, q]$ and $[a, b]$ the empty set of bits in $a_j + [16k_j, 32k_j]$ and $a_j + [64k_j, 72k_j]$. We allocate $[p'_l, q'_l] = q + [1, 8\log\log N]$ and $[a'_l, b'_l] = b + [1, 4k_j]$ in $[a, b]$ (shrinking $[p, q]$ and $[a, b]$ accordingly). Then we insert w_l into B_{v_j}.

2. In the SPAWN event, we add a τ_1-tile v_j to T_1.
- Denote by $[p, q]$ the remaining space in $[1, 100N]$. We allocate $[p_j, q_j] = q + [1, 10\log N]$ and shrink $[p, q]$ accordingly. Similarly let $[a, b]$ be the remaining space in $[650N, 950N]$. We allocate $[a_j, b_j] = b + [1, 104k_j]$ and shrink $[a, b]$ accordingly. Finally, we insert v_j into the balanced binary tree B, allocating $4\log\log N$ bits in $[950N, 1000N]$.

3. In the MINI-DOUBLE event for $w_l \in T_2$, $[a'_l, b'_l]$ has three quarters of its bits allocated.
- Conceptually, we set $[x'_l, y'_l] \leftarrow [a'_l, b'_l]$.
- Let $[a, b]$ be the empty space in a space in $a_j + [68k_j, 72k_j]$. We allocate a new interval $[a'_l, b'_l] = a + [1, 2 \cdot |[x'_l, y'_l]|]$ to w_l and reduce $[a, b]$ accordingly.
- We refer to $[x'_j, y'_j]$ as the *shadow* of $[a'_j, b'_j]$.

4. In the DOUBLE event for $v_j \in T_1$, the empty half of the red, green or yellow interval in $[a_j, b_j]$ has at least half of its bits allocated. I.e., either $a_j + [32K_j^*, 64K_j^*]$, or, $a_j + [68K_j^*, 72K_j^*]$, or, $a_j + [88K_j^*, 104K_j^*]$ has at least half of its bits allocated.
   - Conceptually, we set $[x_j, y_j] \leftarrow [a_j, b_j]$.
   - Let $[a, b]$ be the empty space in $[550N, 950N]$. We allocate a new interval $[a_j, b_j] = b + [1, 104k_j]$ to v_j and reduce $[a, b]$ accordingly.
   - We refer to $[x_j, y_j]$ as the *shadow* of $[a_j, b_j]$ and set $K_j^* \leftarrow 2K_j^*$.
5. In the MEGA-DOUBLE-event, the empty half of the red, green or yellow interval in $[1, 1000N]$ has at least half of its bits allocation. I.e., one of the tree intervals: $[50N, 100N]$, $[650N, 950N]$ or $[984N, 1000N]$ has at least half of its bits allocated.

▶ **Lemma 13.** *For any $w_l \in T_2$, it takes at least $|[a'_l, b'_l]|/2$ extend operations (where the head is in w_l) before we trigger a MINI-DOUBLE event for w_l.*

Proof. By Lemma 7, the length of the substrings $\beta_S(w_l) + \epsilon_S(w_l)$ of $\text{ENC}_S(w_l)$ have at most $2k'_l$ bits. When $[a'_l, b'_l]$ was created let $|S \cap w_l| = X$. Per definition, the width of $[a'_l, b'_l]$ was $4X$ with $2X$ bits remaining empty and so the lemma follows. ◀

▶ **Lemma 14.** *For any $v_j \in T_2$, it takes at least $K_j^*/4$ extend operations (with the head in $N(v_j)$) before we trigger a DOUBLE event for v_j.*

Proof. We do a case distinction on what triggered the event:

- Suppose that $a_j + [32K_j^*, 64K_j^*]$ has at least half of its bits allocated (this is the empty space of the red interval). We only allocate memory in these intervals during a MINI-SPAWN event in v_j. At a MINI-SPAWN event, we allocate $8 \log\log N$ bits in the red interval. Thus, by the time that we have filled $16K_j^*$ bits, we must have triggered $2K_j^*/\log\log N$ MINI-SPAWN events. By Lemma 3, it takes at least $\log\log N$ extend operations in $N(v_j)$ to trigger four MINI-SPAWN events in v_j and so the lemma follows.
- Suppose that $a_j + [88K_j^*, 104K_j^*]$ has at least half its bits allocated (this is the empty space of the yellow interval). We only allocate memory in these intervals during a MINI-SPAWN event. At a MINI-SPAWN event, we allocate $4 \log\log N$ bits. Thus, by the time that we have filled $8K_j^*$ bits we must have triggered $K_j^*/\log logN$ MINI-SPAWN events. By Lemma 3, it takes at least $\log\log N$ extend operations in $N(v_j)$ to trigger four MINI-SPAWN events in v_j and so the lemma follows.
- Suppose that $a_j + [68K_j^*, 72K_j^*]$ has at least half of its bits allocated (this is the empty space of the green interval). If this memory was (largely) allocated through MINI-SPAWN events, then since each such event allocates $O(1)$ bits in this range and requires at least one extend operation. Thus, the lemma trivially follows. So suppose that this memory was allocated through MINI-DOUBLE events instead.
  For any $w_l \in T_2 \cap v_j$ denote $[a'_l(1), b'_l(1)] = [a'_l, b'_l]$ whenever $[a'_l, b'_l] \subset a_j + [68K_j^*, 72K_j^*]$. I.e., consider the current interval associated to w_l if it is stored in the empty half of the green interval. We recursively define the intervals $[a'_l(t), b'_l(t)]$ as the *shadow* of $[a'_l(t-1), b'_l(t-1)]$. We note two facts: Firstly, our assumption implies that the intervals $[a'_l(t), b'_l(t)]$ use at least $2K_j^*$ bits in $a_j + [68K_j^*, 72K_j^*]$. Secondly, for all l, for all t, $|[a'_l(t), b'_l(t)]| = \frac{1}{2}|[a'_l(t-1), b'_l(t-1)]|$. It follows from the geometric series that:

$$\sum_{w_l \in T_2 \cap v_j} \sum_t |[a'_l(t), b'_l(t)]| \geq 2K_j^* \Rightarrow \sum_{w_l \in T_2 \cap v_j} |[a'_l(1), b'_l(1)]| \geq K_j^*.$$

  The interval $[a'_l(1), b'_l(1)]$ can only have been created through a MINI-DOUBLE event on $[a'_l(2), b'_l(2)]$. By Lemma 13, it takes $\frac{1}{2}|[a'_l(2), b'_l(2)]| \geq \frac{1}{4}|[a'_l(1), b'_l(1)|$ extend operations in w_l before this event is triggered and so the lemma follows. ◀

▶ **Lemma 15.** *It takes at least $N/100$* extend *operations to trigger a* MEGA-DOUBLE *event.*

Proof. The proof is analogous to Lemma 14 as a case distinction on the event trigger:

- Suppose that $[50N, 100N]$ has at least half of its bits allocated (this is the empty space of the red interval). We only allocate memory in these intervals during a SPAWN event. At a SPAWN event, we allocate $10 \log \log N$ bits in the red interval. Before we filled $25N$ bits, we must have triggered $2N/\log \log N$ SPAWN events. By Lemma 3, it takes at least $\log \log N$ extend operations to trigger four SPAWN events and so the lemma follows.
- Suppose that $[984N, 1000N]$ has at least half its bits allocated (this is the empty space of the yellow interval). We only allocate memory in these intervals during a SPAWN event. At a SPAWN event, we allocate $4 \log \log N$ bits. Thus, by the time that we have filled $40N$ bits we must have triggered $10N/\log logN$ SPAWN events. By Lemma 3, it takes at least $\log \log N$ extend operations to trigger four SPAWN events and so the lemma follows.
- Suppose that $[650N, 950N]$ has at least half of its bits allocated (this is the empty space of the green interval). The space required by $\frac{1}{10}N$ SPAWN events may be charged to all the other space in memory (since each SPAWN event allocates memory proportional to how much memory is already in space). Thus, we may suppose that this memory was allocated through DOUBLE events.
  For any $v_j \in T_1$ denote $[a_j(1), b_j(1)] = [a_j, b_j]$ whenever $[a_j, b_j] \subset [650N, 950N]$. I.e., consider the current interval associated to v_j if it is stored in the empty half of the green interval. We recursively define $[a_j(t), b_j(t)]$ as the *shadow* of $[a_j(t-1), b_j(t-1)]$.
  We note two facts: Firstly, our assumption implies that the intervals $[a_j(t), b_j(t)]$ use at least $150N$ bits in $[650N, 950N]$. Secondly, for all l, for all t, $|[a_j(t), b_j(t)]| = \frac{1}{2}|[a_j(t-1), b_j(t-1)]|$. It follows from the geometric series that:

$$\sum_{v_j \in T_1} \sum_{t} |[a_j(t), b_j(t)]| \geq 150N \Rightarrow \sum_{v_j \in T_1} |[a_j(1), b_j(1)]| \geq 75N.$$

  $[a_j(1), b_j(1)]$ can only have been created through a DOUBLE event on $[a_j(2), b_j(2)]$. By Lemma 14, it takes $\frac{1}{8}K_j^* \geq \frac{1}{8 \cdot 104}|[a_j(1), b_j(1)]|$ extend operations in $N(v_j)$ before this event is triggered.[1] Each extend operation occurs in $N(v_j)$ for at most five tiles in T_1. This implies that at least $\frac{75}{8 \cdot 104 \cdot 5} \geq \frac{N}{100}$ extend operations must have occurred. ◀

5.3.1 Proving our main theorem

We are now ready to prove:

▶ **Theorem 1.** *We can represent a snake of length n in $O(n + \log u)$ bits and support* extend *and* reduce *operations in constant time.*

Proof. We assume that $n \in [0.5N, 2N]$ and that we have built our data structure with $\tau_1 = \log N$ and $\tau_2 = \log \log N$. On a macro-level, we use the standard rebuilding deamortization technique. For the first $\frac{N}{200}$ updates, we update our data structure in $O(1)$ worst-case time using our update strategy of Section 4.2; allocating space until we trigger a MINI-DOUBLE or DOUBLE event.

Whenever we trigger a MINI-DOUBLE event for a tile $w_l \in T_2$, we reallocate the new memory $[a'_l, b'_l]$ in $O(1)$ time. Over the next $|[a'_l, b'_l]|/10$ updates that change $S \cap W_l$, we perform ten times the work to both execute the work in the shadow of $[a'_l, b'_l]$ and copy all

[1] Indeed, it took $\frac{1}{4}K_j^*$ extend operations to trigger a DOUBLE event, after which K_j^* was doubled. Moreover, per definition, $[a_j, b_j]$ is $104K_j^*$ bits wide.

info from its shadow into $[a'_l, b'_l]$ (whilst queueing updates not executed in $[a'_l, b'_l]$). When we have successfully copied the shadow of $[a'_l, b'_l]$ we dequeue updates at four times the pace we are queueing them so that our interval $[a'_l, b'_l]$ is up to date before its shadow is full.

Whenever we trigger a DOUBLE event for a tile $v_j \in T_1$, we reallocate the new memory $[a_l, b_l]$ in $O(1)$ time. Over the next $K^*_j/10$ updates that change $S \cap v$, we perform ten times the work to both execute the update in the shadow of $[a_j, b_j]$ and copy all info from its shadow into $[a_l, b_l]$ (whilst queueing updates not executed in $[a_j, b_j]$). When we have successfully copied the shadow of $[a_j, b_j]$ we dequeue updates at four times the pace we are queueing them so that our interval $[a_j, b_j]$ is up to date before its shadow is full.

This way, we maintain our data structure in $O(1)$ time for the first $\frac{N}{200}$ updates. By Lemma 15, we cannot trigger a MEGA-DOUBLE event in this time and thus everything fits well within memory. At this point, we set $N = n$ and we reallocate a new interval of $1000N$ bits. Over the next $\frac{N}{200}$ updates, we do four times the work: executing updates on our original interval whilst copying the info from our original interval into the new interval (queuing updates in the process). When we have successfully copied the original interval into the new interval, we dequeue the updates at four times the pace that we are queuing them so that the new copy is up to date before we reach $\frac{N}{200}$ updates in the first interval. Thus, we never trigger a MEGA-DOUBLE event and we always fit within $O(N)$ bits. ◀

References

1 Ali Jaber Almalki and Pawel Wocjan. Exploration of reinforcement learning to play Snake game. In *Proc. CSCI*, pages 377–381, 2019.
2 Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3), 2007.
3 Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proc. 51st FOCS*, pages 787–796, 2010.
4 Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. All-purpose hashing. *arXiv preprint arXiv:2109.04548*, 2021.
5 Michael A Bender, Martín Farach-Colton, John Kuszmaul, and William Kuszmaul. Modern hashing made simple. In *Proc. 7th SOSA*, pages 363–373. SIAM, 2024.
6 Michael A Bender, Martín Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. On the optimal time/space tradeoff for hash tables. In *STOC*, pages 1284–1297, 2022.
7 Ioana O Bercea and Guy Even. Dynamic dictionaries for multisets and counting filters with constant time operations. *Algorithmica*, 85(6):1786–1804, 2023.
8 Russell Sammut Bonnici, Chantelle Saliba, Giulia Elena Caligari, and Mark Bugeja. Exploring reinforcement learning: A case study applied to the popular Snake game. In *DTMAD*, 2022.
9 Marzio De Biasi and Tim Ophelders. The complexity of snake. In *FUN*, 2016.
10 Shubham Sharma, Saurabh Mishra, Nachiket Deodhar, Akshay Katageri, and Parth Sagar. Solving the classic Snake game using AI. In *Proc. PuneCon*, pages 1–4, 2019.
11 Wikipedia. URL: `https://en.wikipedia.org/wiki/Snake_(video_game_genre)`.
12 Wikipedia. URL: `https://en.wikipedia.org/wiki/Snake_(video_game)`.