Improved Dynamic Rank-Select Entropy-Bound Structures

Rodrigo González[1] and Gonzalo Navarro[1]

[1]Department of Computer Science
University of Chile

Outline

1. Introduction
2. Collection of Searchable Partial Sums with Indels
3. Uncompressed Dynamic Rank-Select Structure for a Small Alphabet
4. Compressed Dynamic Rank-Select Structure
5. Conclusions

Outline

1. Introduction
2. Collection of Searchable Partial Sums with Indels
3. Uncompressed Dynamic Rank-Select Structure for a Small Alphabet
4. Compressed Dynamic Rank-Select Structure
5. Conclusions

Outline

1. Introduction
2. Collection of Searchable Partial Sums with Indels
3. Uncompressed Dynamic Rank-Select Structure for a Small Alphabet
4. Compressed Dynamic Rank-Select Structure
5. Conclusions

Outline

1. Introduction
2. Collection of Searchable Partial Sums with Indels
3. Uncompressed Dynamic Rank-Select Structure for a Small Alphabet
4. Compressed Dynamic Rank-Select Structure
5. Conclusions

Outline

1. Introduction
2. Collection of Searchable Partial Sums with Indels
3. Uncompressed Dynamic Rank-Select Structure for a Small Alphabet
4. Compressed Dynamic Rank-Select Structure
5. Conclusions

Motivation

- Operations *rank* and *select* over a sequence of symbols have many applications to the design of succinct and compressed data structures to manage text collections, structured text, binary relations, trees, graphs, and so on.
- We are interested in the case where the sequences can be updated via insertions and deletions of symbols.

Previous Work

- A solution by Mäkinen and Navarro, achieves compressed space, i.e., $nH_0 + o(n \log \sigma)$ bits and $O(\log n \log \sigma)$ worst-case time for all the operations.
- A solution, by Lee and Park, achieves uncompressed space, i.e. $n \log \sigma + O(n) + o(n \log \sigma)$ bits and $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ amortized time.
- We combine these two solutions to obtain $nH_0 + o(n \log \sigma)$ bits of space and $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ worst-case time for all the operations.

Definitions

For a sequence $T[1, n]$ over an alphabet Σ of size σ, we want to represent it efficiently and to support the following operations:

- $access(T, i)$ returns the symbol $T[i]$.
- $rank_c(T, i)$ returns the number of times symbol c appears in the prefix $T[1, i]$.
- $select_c(T, i)$ returns the position of the i-th c in T.

We also want to add dynamic capabilities to this sequence:

- $insert_c(T, i)$ inserts symbol c between $T[i]$ and $T[i + 1]$.
- $delete(T, i)$ deletes $T[i]$ from T.

Searchable Partial Sums with Indels

This problem consists in maintaining a sequence S of nonnegative integers $s_1, \ldots, s_n$, each one of $k = O(\log n)$ bits, supporting the following queries and operations:

- $sum(S, i)$ is $\sum_{l=1}^{i} s_l$.
- $search(S, y)$ is the smallest i' such that $sum(S, i') \geq y$.
- $update(S, i, x)$ updates s_i to $s_i + x$ (x can be negative as long as the result is not).
- $insert(S, i, x)$ inserts a new integer x between s_{i-1} and s_i.
- $delete(S, i)$ deletes s_i from the sequence.

Collection of Searchable Partial Sums with Indels

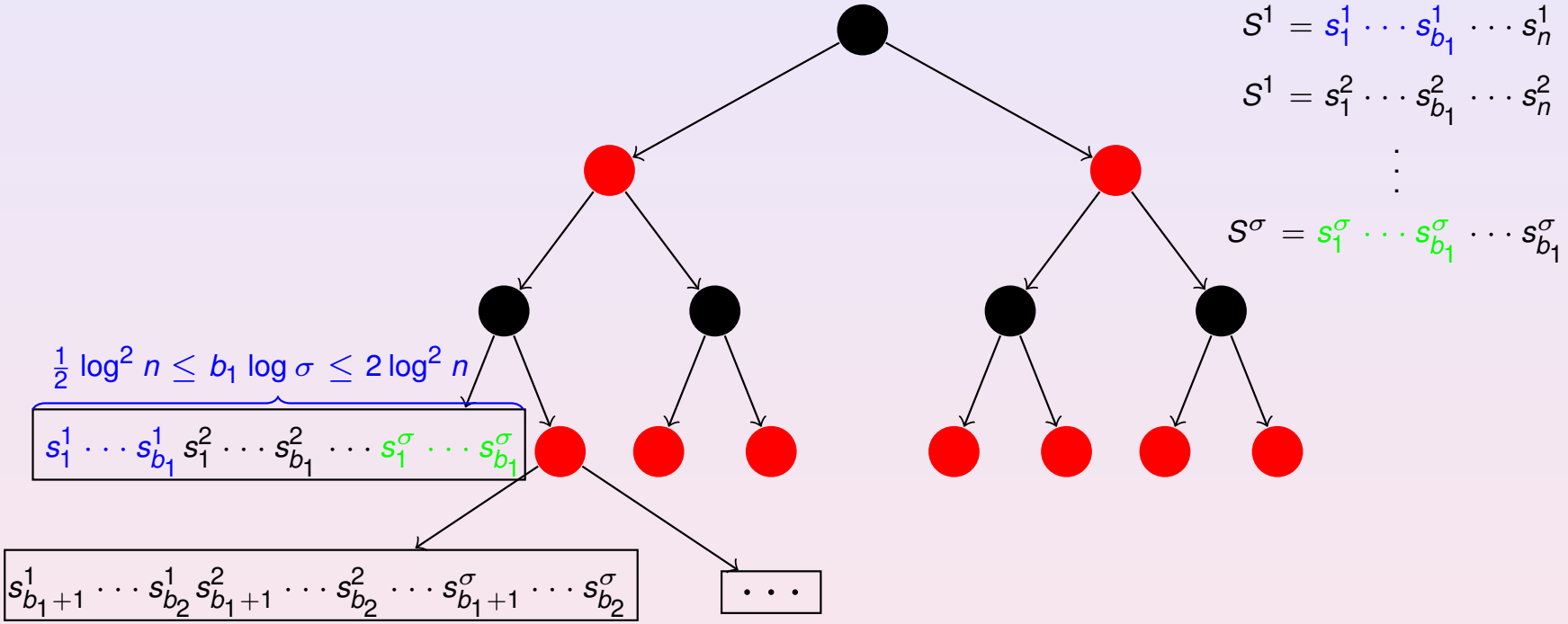
This problem consists in maintaining a collection of σ sequences $C = \{S^1, \ldots, S^\sigma\}$ of nonnegative integers $\{s_i^j\}_{1 \le j \le \sigma, 1 \le i \le n}$, each one of $k = O(\log n)$ bits. We support the following queries and operations:

- *sum*(C, j, i) is $\sum_{l=1}^{i} s_l^j$.
- *search*(C, j, y) is the smallest i' such that *sum*$(C, j, i') \ge y$.
- *update*(C, j, i, x) updates s_i^j to $s_i^j + x$.
- *insert*(C, i) inserts 0 between s_{i-1}^j and s_i^j for all $1 \le j \le \sigma$.
- *delete*(C, i) deletes s_i^j from the sequence S^j for all $1 \le j \le \sigma$. To perform this operation it must hold $s_i^j = 0$ for all $1 \le j \le \sigma$.

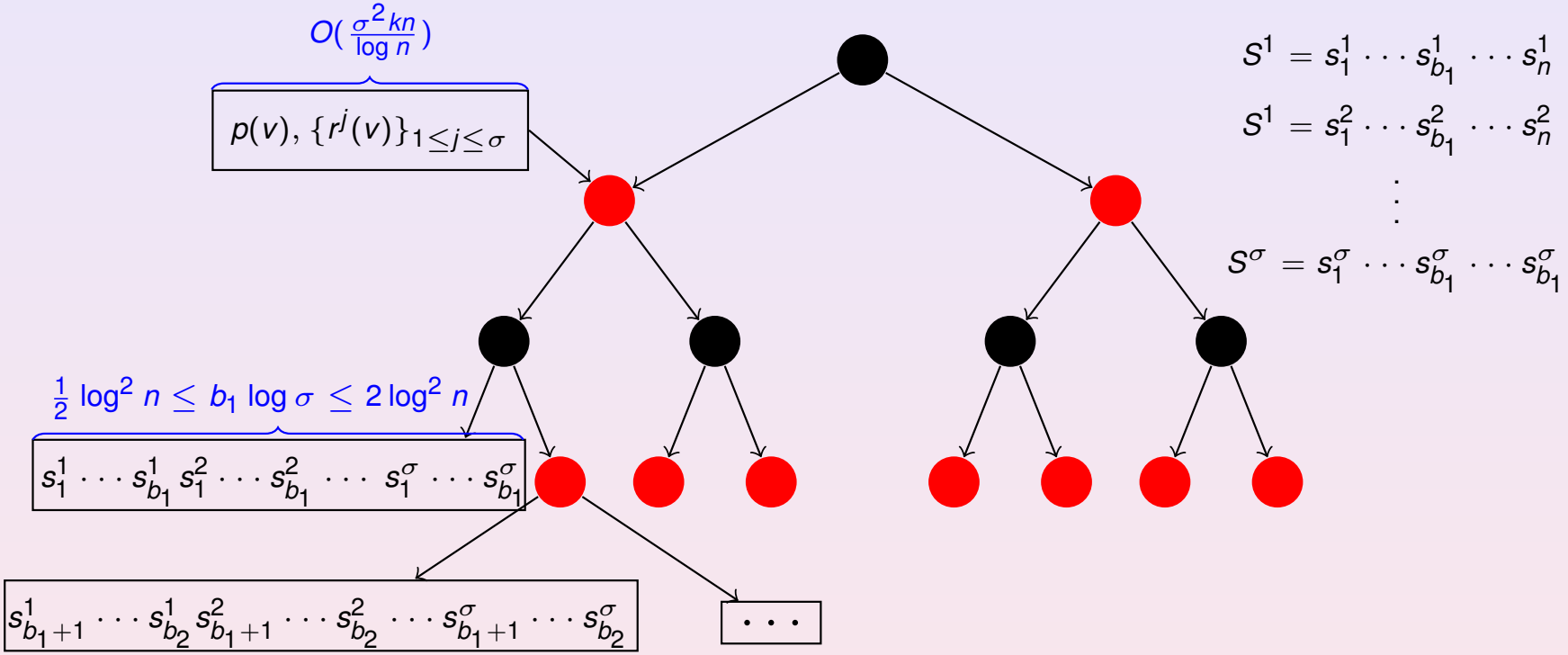
Collection of Searchable Partial Sums with Indels

$$
\begin{aligned}
S^1 &= s_1^1 \cdots s_{b_1}^1 \cdots s_n^1 \\
S^1 &= s_1^2 \cdots s_{b_1}^2 \cdots s_n^2 \\
&\vdots \\
S^\sigma &= s_1^\sigma \cdots s_{b_1}^\sigma \cdots s_{b_1}^\sigma
\end{aligned}
$$

Collection of Searchable Partial Sums with Indels



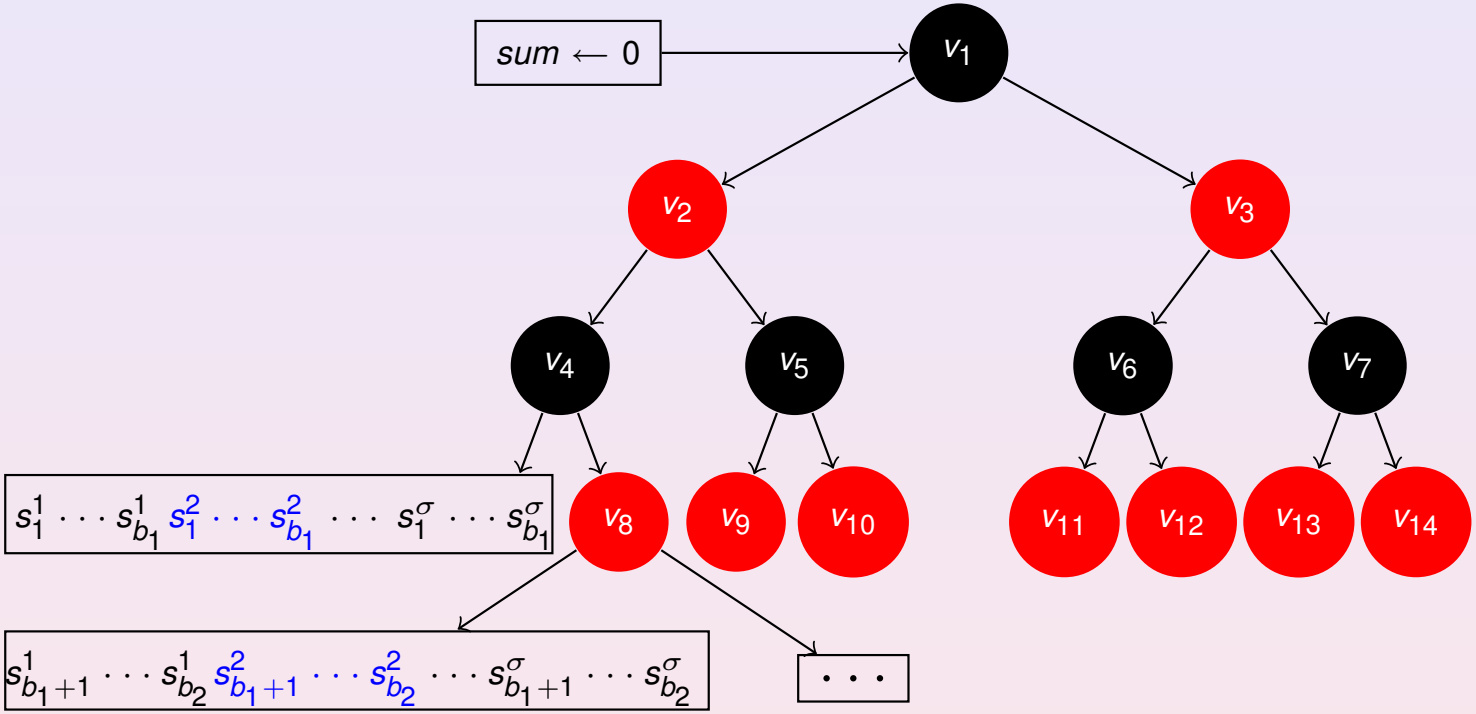
Collection of Searchable Partial Sums with Indels



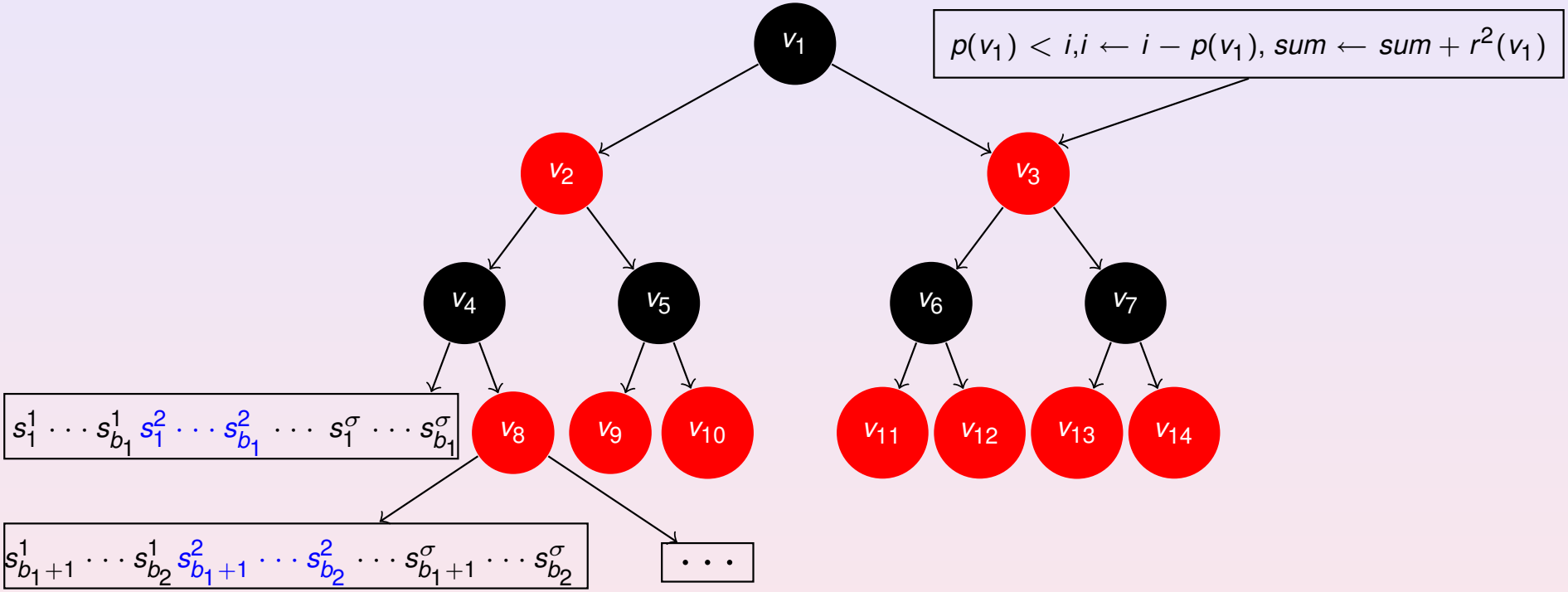
$p(v)$ is the number of positions stored in the left subtree.

$r^j(v)$ is the sum of the integers in the left subtree for sequence S^j.

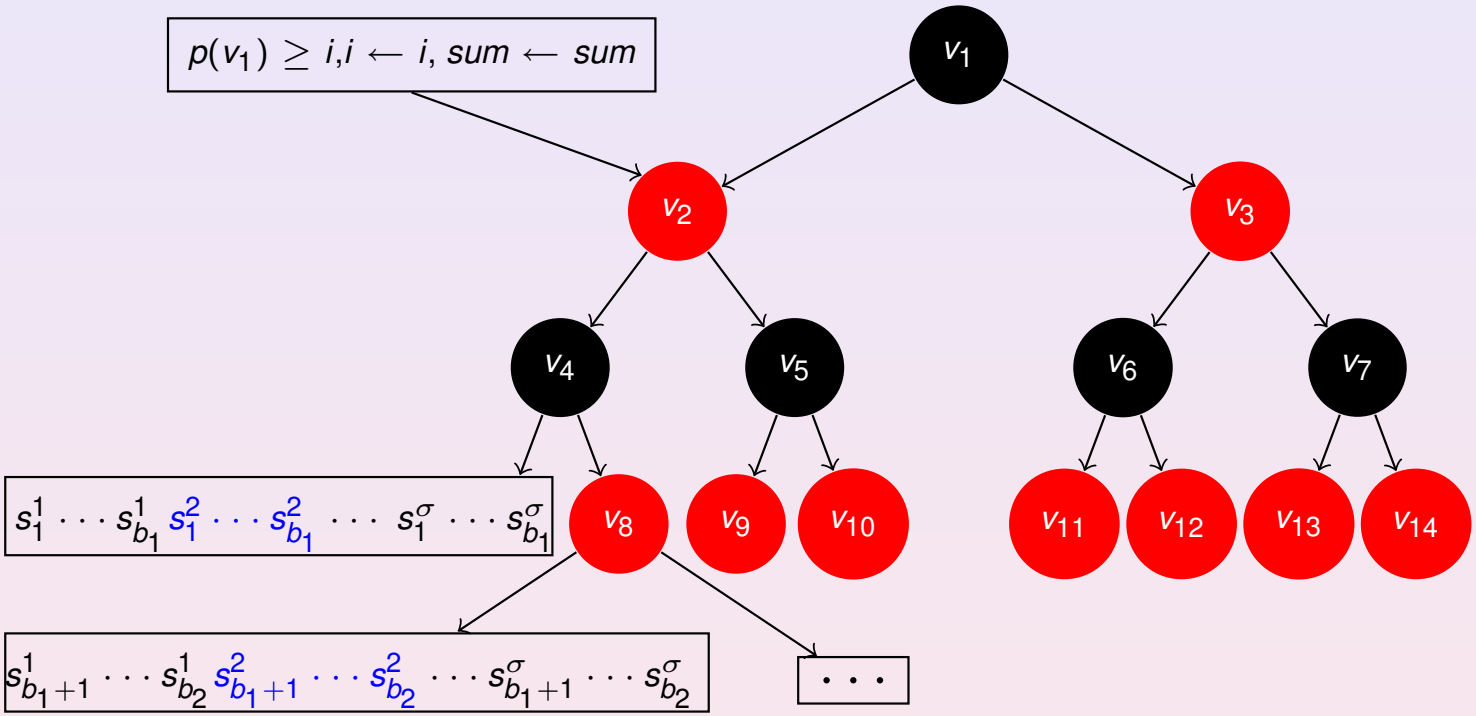
Computing $sum(C, 2, i)$



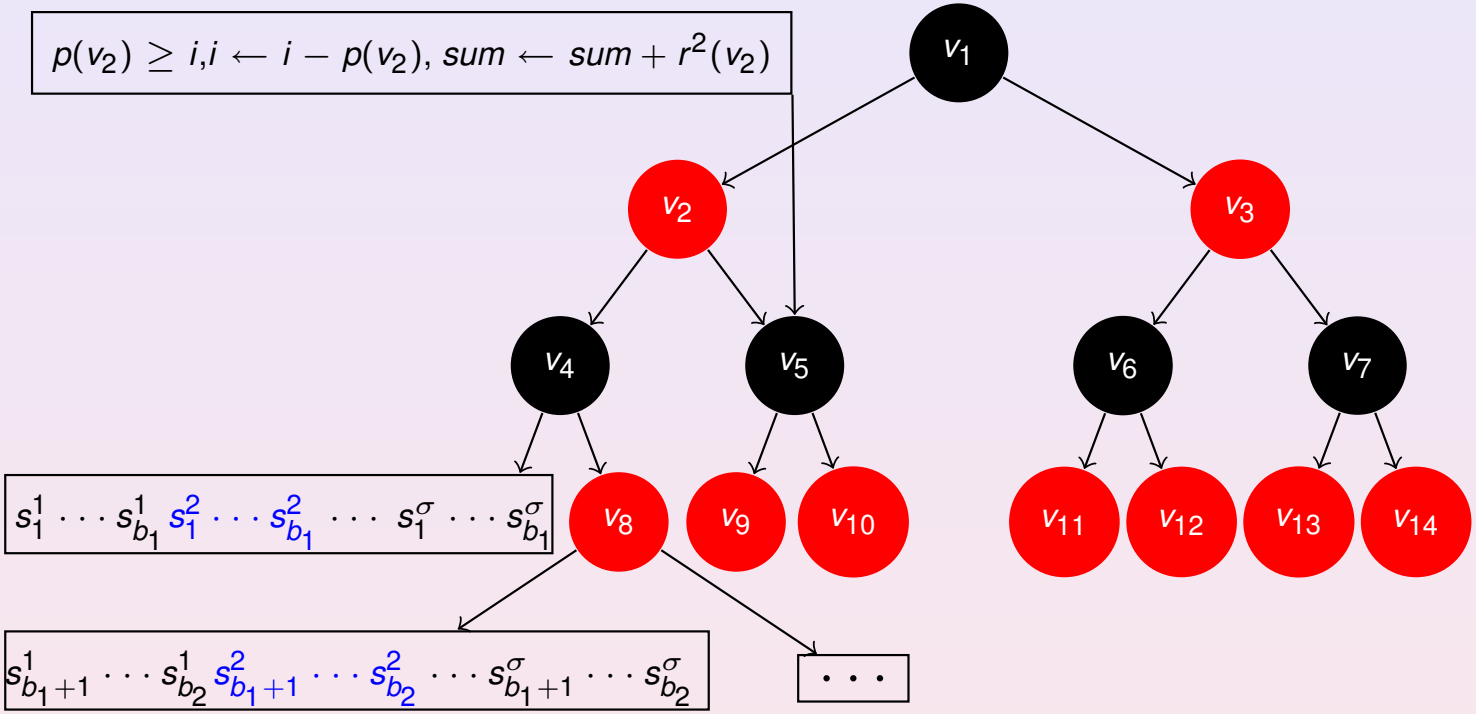
Computing $sum(C, 2, i)$



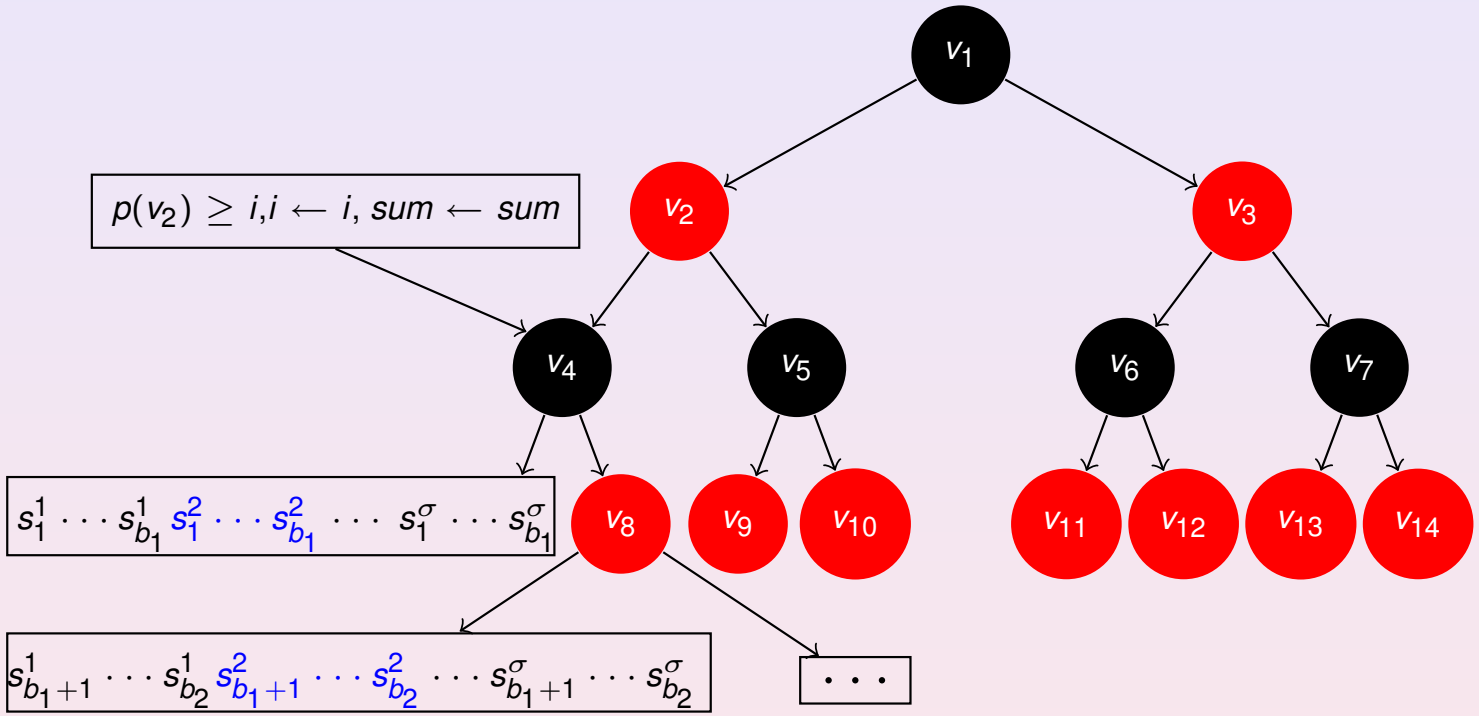
Computing $sum(C, 2, i)$



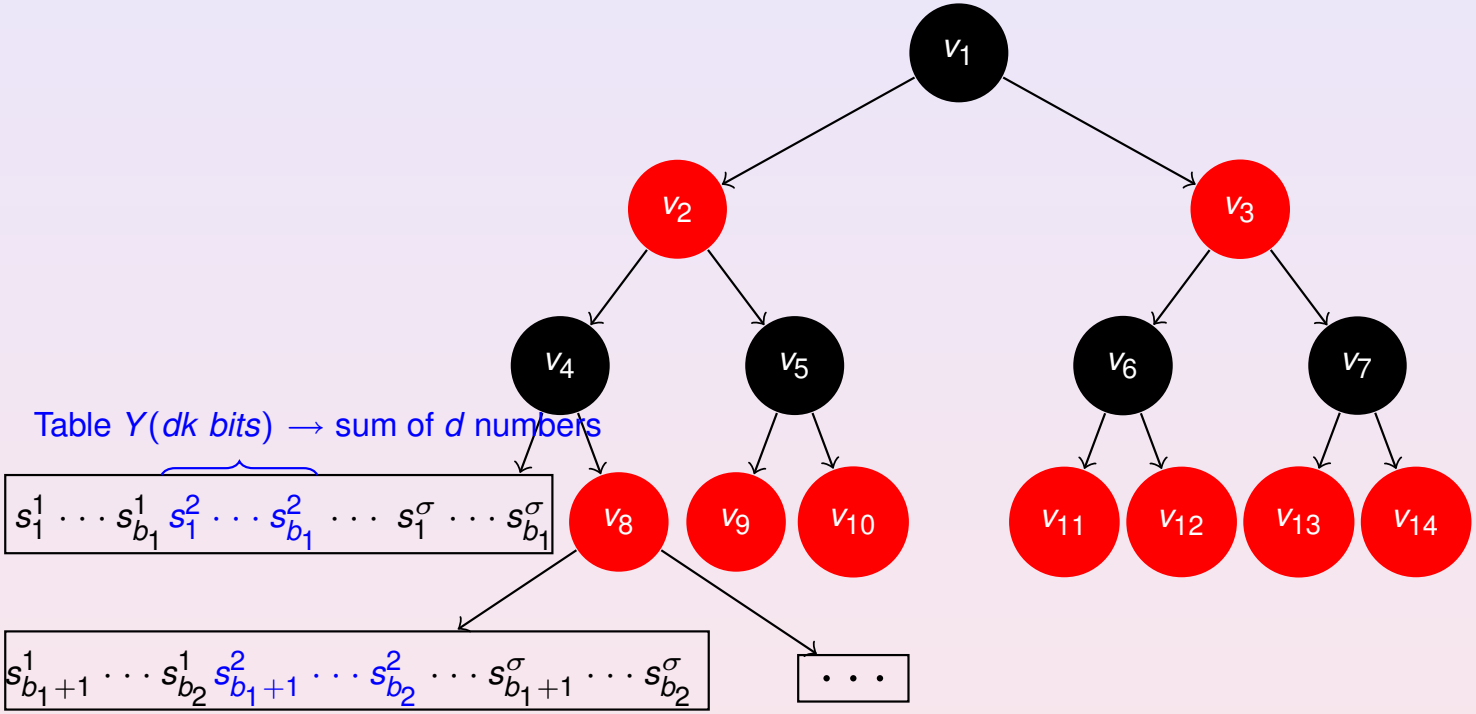
Computing $sum(C, 2, i)$



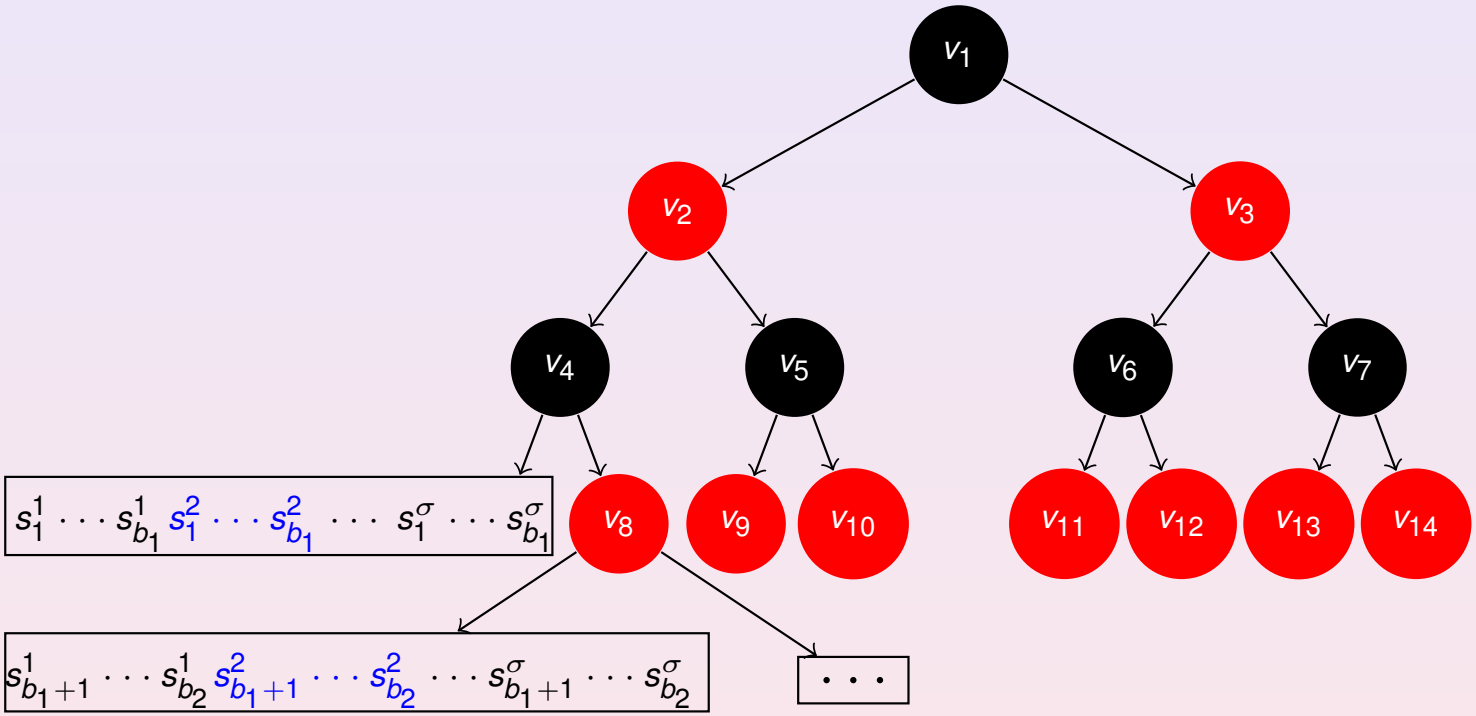
Computing $sum(C, 2, i)$



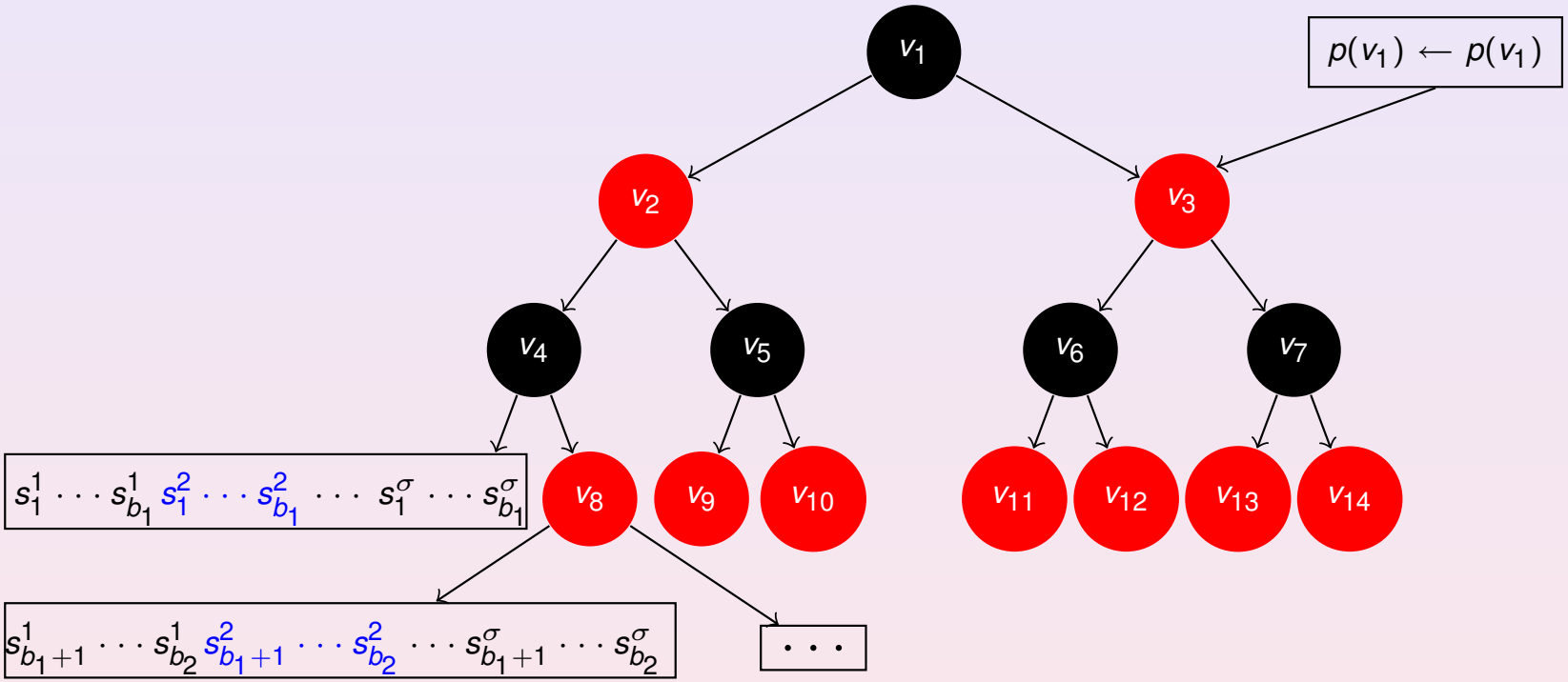
Computing $sum(C, 2, i)$



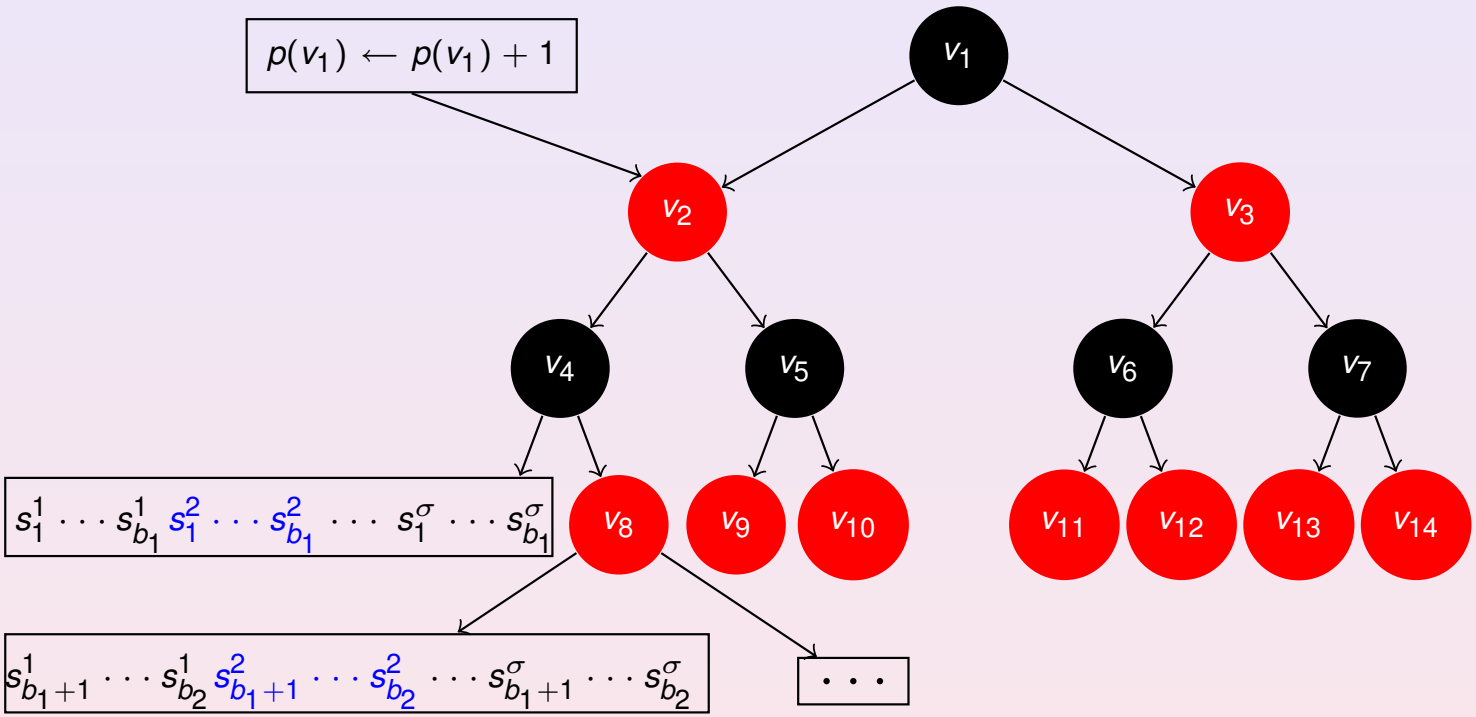
Computing *insert*(C, i)



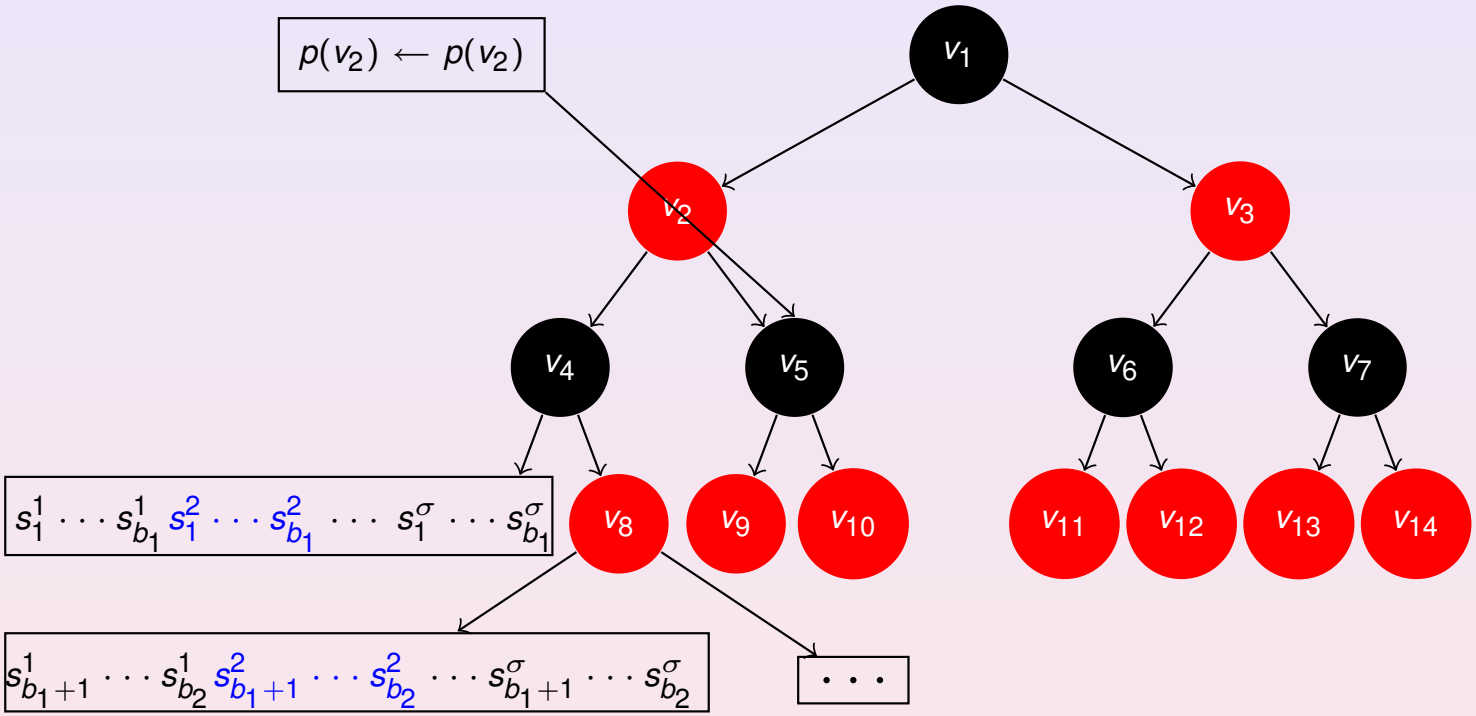
Computing *insert*(C, i)



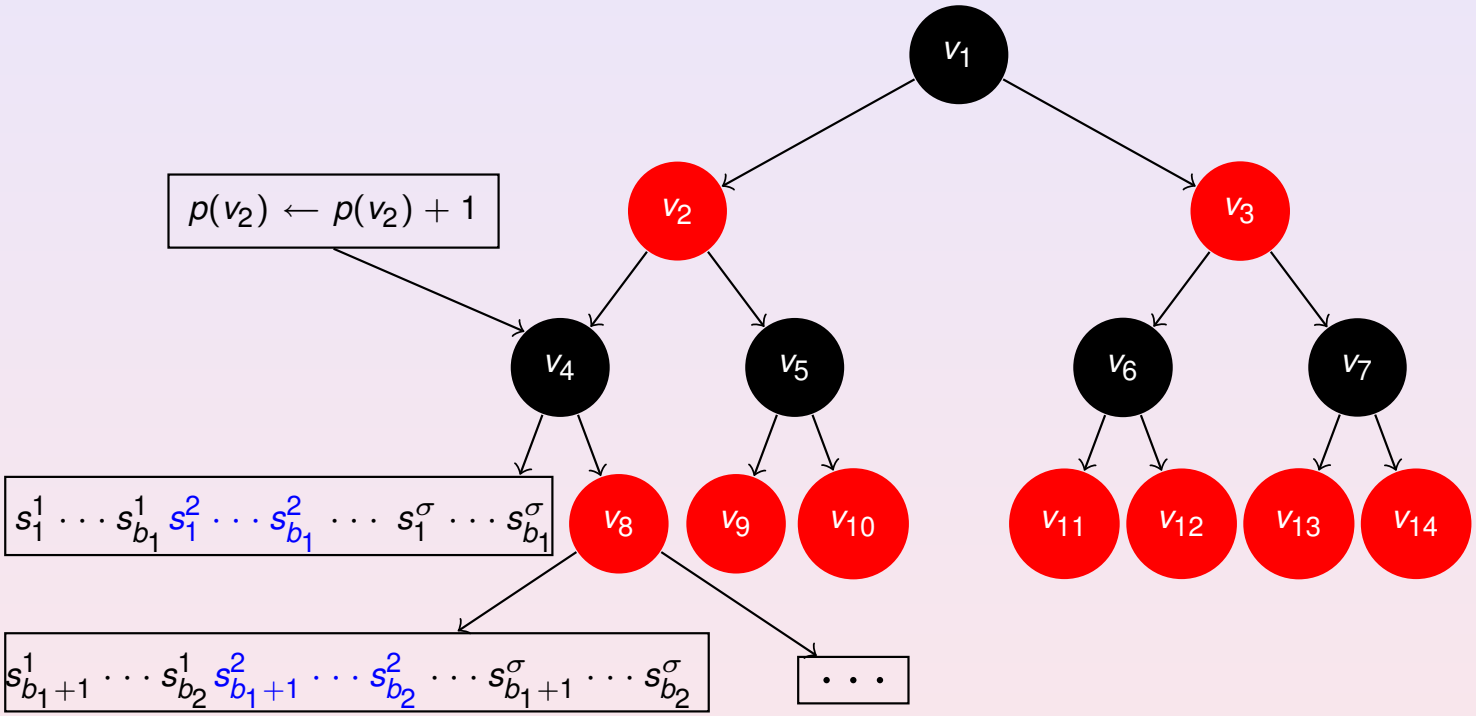
Computing *insert*(C, i)



Computing $insert(C, i)$



Computing *insert*(C, i)



Leaf processing

We can copy a leaf, adding a 0 in all the subsequences, in $O(\sigma + \log n)$ time.

$$s^1_{b_1+1} \cdots s^1_{i-1} s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} s^\sigma_i \cdots s^\sigma_{b_2}$$



Leaf processing

We can copy a leaf, adding a 0 in all the subsequences, in $O(\sigma + \log n)$ time.

$$s^1_{b_1+1} \cdots s^1_{i-1} s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} s^\sigma_i \cdots s^\sigma_{b_2}$$

$$s^1_{b_1+1} \cdots s^1_{i-1}$$

Leaf processing

We can copy a leaf, adding a 0 in all the subsequences, in $O(\sigma + \log n)$ time.

$$s^1_{b_1+1} \cdots s^1_{i-1} s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} s^\sigma_i \cdots s^\sigma_{b_2}$$

$$s^1_{b_1+1} \cdots s^1_{i-1} 0$$

Leaf processing

We can copy a leaf, adding a 0 in all the subsequences, in $O(\sigma + \log n)$ time.

$$s^1_{b_1+1} \cdots s^1_{i-1} s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} s^\sigma_i \cdots s^\sigma_{b_2}$$

$$s^1_{b_1+1} \cdots s^1_{i-1} 0 s^1_i \cdots s^1_{b_2}$$

Leaf processing

We can copy a leaf, adding a 0 in all the subsequences, in $O(\sigma + \log n)$ time.

$$s^1_{b_1+1} \cdots s^1_{i-1} s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} s^\sigma_i \cdots s^\sigma_{b_2}$$

$$s^1_{b_1+1} \cdots s^1_{i-1} 0 s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1}$$

Leaf processing

We can copy a leaf, adding a 0 in all the subsequences, in $O(\sigma + \log n)$ time.

$$s^1_{b_1+1} \cdots s^1_{i-1} s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} s^\sigma_i \cdots s^\sigma_{b_2}$$

$$s^1_{b_1+1} \cdots s^1_{i-1} 0 s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} 0$$

Leaf processing

We can copy a leaf, adding a 0 in all the subsequences, in $O(\sigma + \log n)$ time.

$$s^1_{b_1+1} \cdots s^1_{i-1} s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} s^\sigma_i \cdots s^\sigma_{b_2}$$

$$s^1_{b_1+1} \cdots s^1_{i-1} 0 s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} 0 s^2_i \cdots s^2_{b_2} \cdots$$

Leaf processing

We can copy a leaf, adding a 0 in all the subsequences, in $O(\sigma + \log n)$ time.

$$s^1_{b_1+1} \cdots s^1_{i-1} s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} s^\sigma_i \cdots s^\sigma_{b_2}$$

$$s^1_{b_1+1} \cdots s^1_{i-1} 0 s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} 0 s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1}$$

Leaf processing

We can copy a leaf, adding a 0 in all the subsequences, in $O(\sigma + \log n)$ time.

$$s^1_{b_1+1} \cdots s^1_{i-1} s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} s^\sigma_i \cdots s^\sigma_{b_2}$$

$$s^1_{b_1+1} \cdots s^1_{i-1} 0 s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} 0 s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} 0$$

Leaf processing

We can copy a leaf, adding a 0 in all the subsequences, in $O(\sigma + \log n)$ time.

$$s^1_{b_1+1} \cdots s^1_{i-1} s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} s^\sigma_i \cdots s^\sigma_{b_2}$$

$$s^1_{b_1+1} \cdots s^1_{i-1} 0 s^1_i \cdots s^1_{b_2} s^2_{b_1+1} \cdots s^2_{i-1} 0 s^2_i \cdots s^2_{b_2} \cdots s^\sigma_{b_1+1} \cdots s^\sigma_{i-1} 0 s^\sigma_i \cdots s^\sigma_{b_2}$$

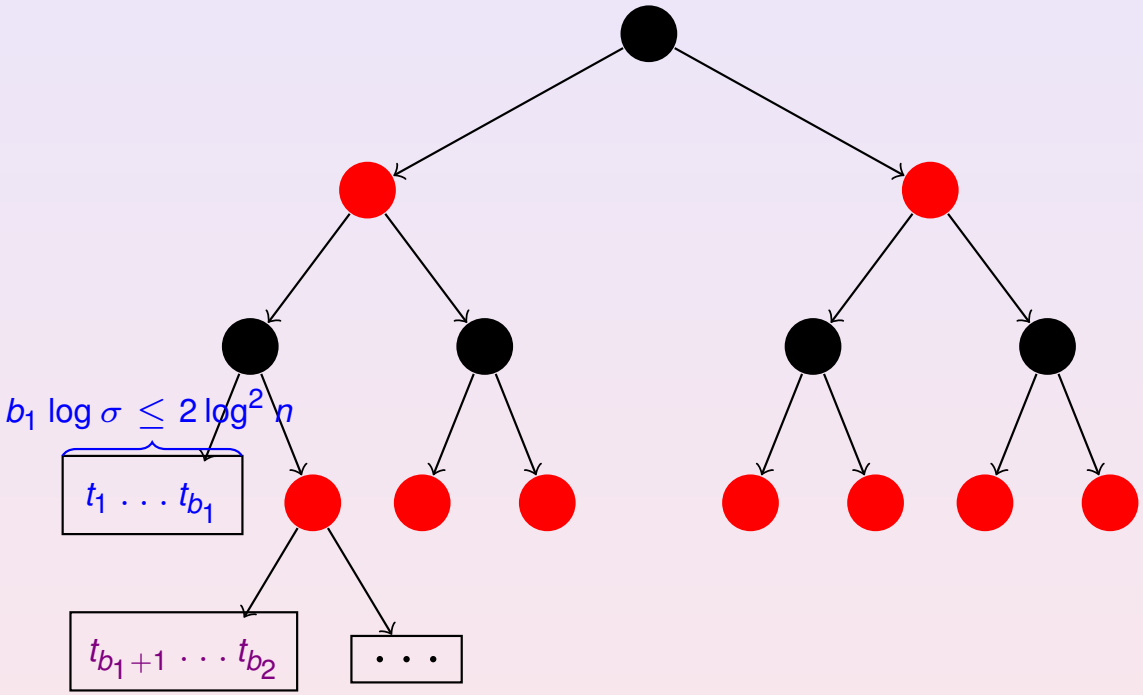
Computing $insert(C, i)$

Overflow

- If the leaf is overflowed, we split the leaf in two. These two copies can be done also in $O(\sigma + \log n)$ time.
- If the red-black tree is unbalance we rebalance it with one rotation and $O(\log n)$ red-black tag updates. After a rotation we need to update $r^j(\cdot)$ and $p(\cdot)$ only for three nodes, which is easily done in $O(\sigma)$ time.
- The *insert* operation takes in total $O(\sigma + \log n)$ time.

Uncompressed Dynamic Rank-Select Structure for a Small Alphabet

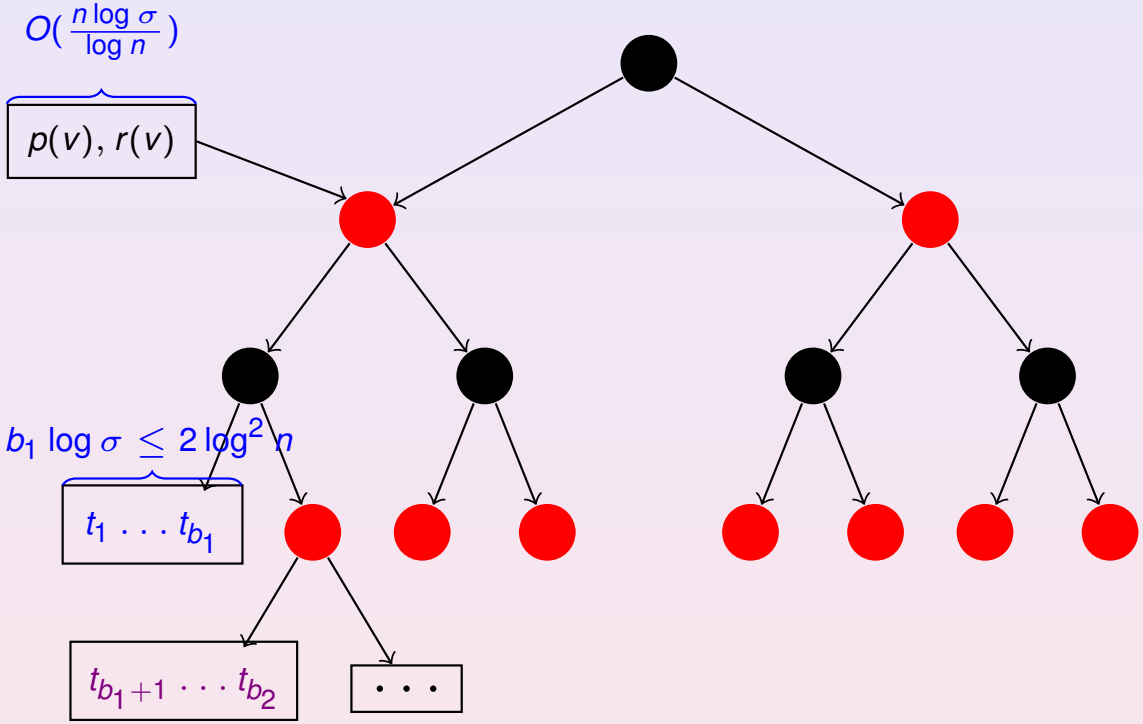
We construct a red-black tree over $T_{1,n} = t_1 \ldots t_{b_1} t_{b_1+1} \ldots t_{b_2} t_{b_2+1} \ldots t_n$.



A superblock storing less than $\log^2 n$ bits will be call sparse. Invariant that no two consecutive sparse superblock may exist.

Uncompressed Dynamic Rank-Select Structure for a Small Alphabet

We construct a red-black tree over $T_{1,n} = t_1 \dots t_{b_1} t_{b_1+1} \dots t_{b_2} t_{b_2+1} \dots t_n$.



A superblock storing less than $\log^2 n$ bits will be call sparse. Invariant that no two consecutive sparse superblock may exist.

$p(v)$ is the number of symbols stored in the left subtree.

$r(v)$ is the number of superblocks in the left subtree.

Uncompressed Dynamic . . .

Partial Sums

- For each superblock i, we maintain s_i^j, the number of occurrences of symbol j in superblock i.
- We store all these sequences of numbers using a *Collection of Searchable Partial Sums with Indels*, C.
- The length of each sequence will be at most $\frac{n \log \sigma}{\frac{1}{2} \log^2 n}$ integers, $\sigma = O(\log n)$ and $k = O(\log \log n)$.
- The partial sums operate in $O(\log n)$ worst-case time.

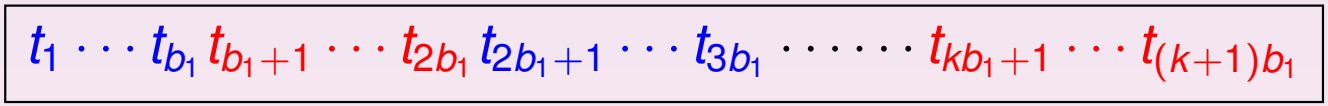
Superblock

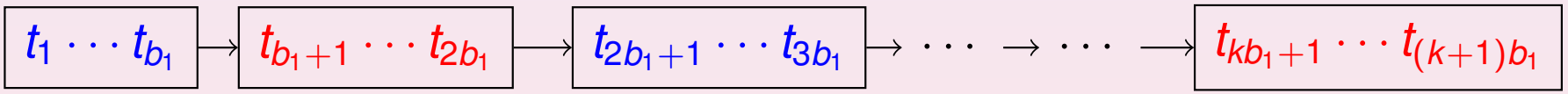
- Each superblock is further divided into *blocks* of $\sqrt{\log n} \log n$ bits
- Each superblock has up to $2\sqrt{\log n}$ blocks.
- We maintain these blocks using a linked list.
- Only the last block could be not fully used.

$$t_1 \cdots t_{b_1}\, t_{b_1+1} \cdots t_{2b_1}\, t_{2b_1+1} \cdots t_{3b_1} \cdots\cdots t_{kb_1+1} \cdots t_{(k+1)b_1}$$

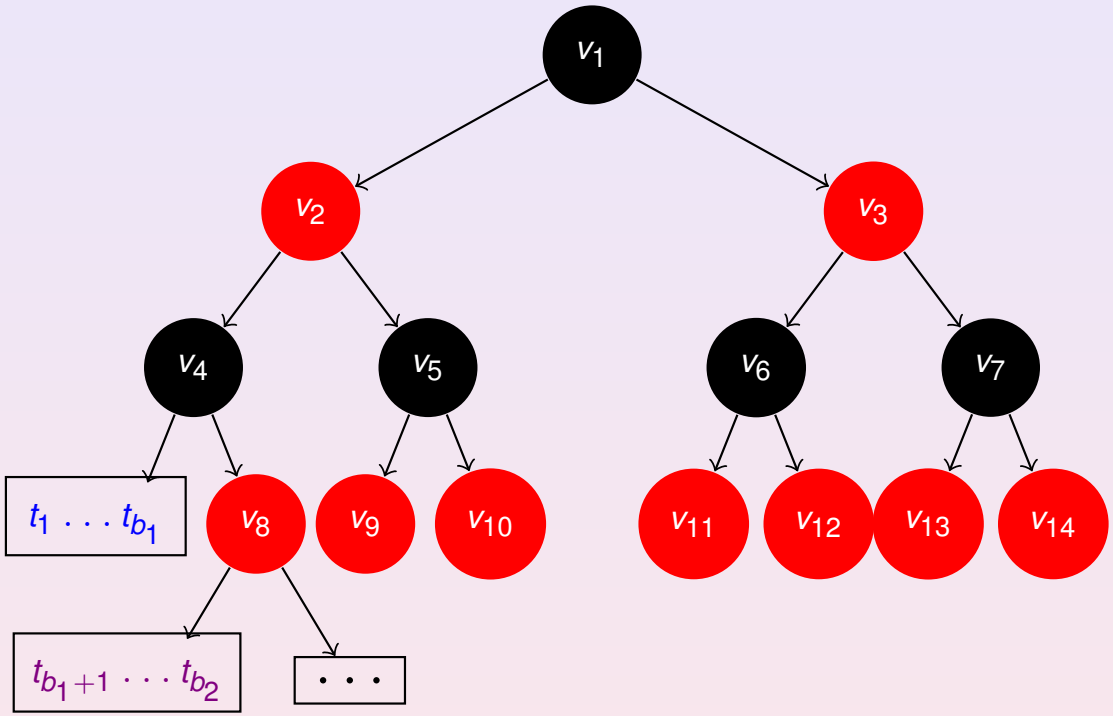
Superblock

- Each superblock is further divided into *blocks* of $\sqrt{\log n} \log n$ bits
- Each superblock has up to $2\sqrt{\log n}$ blocks.
- We maintain these blocks using a linked list.
- Only the last block could be not fully used.

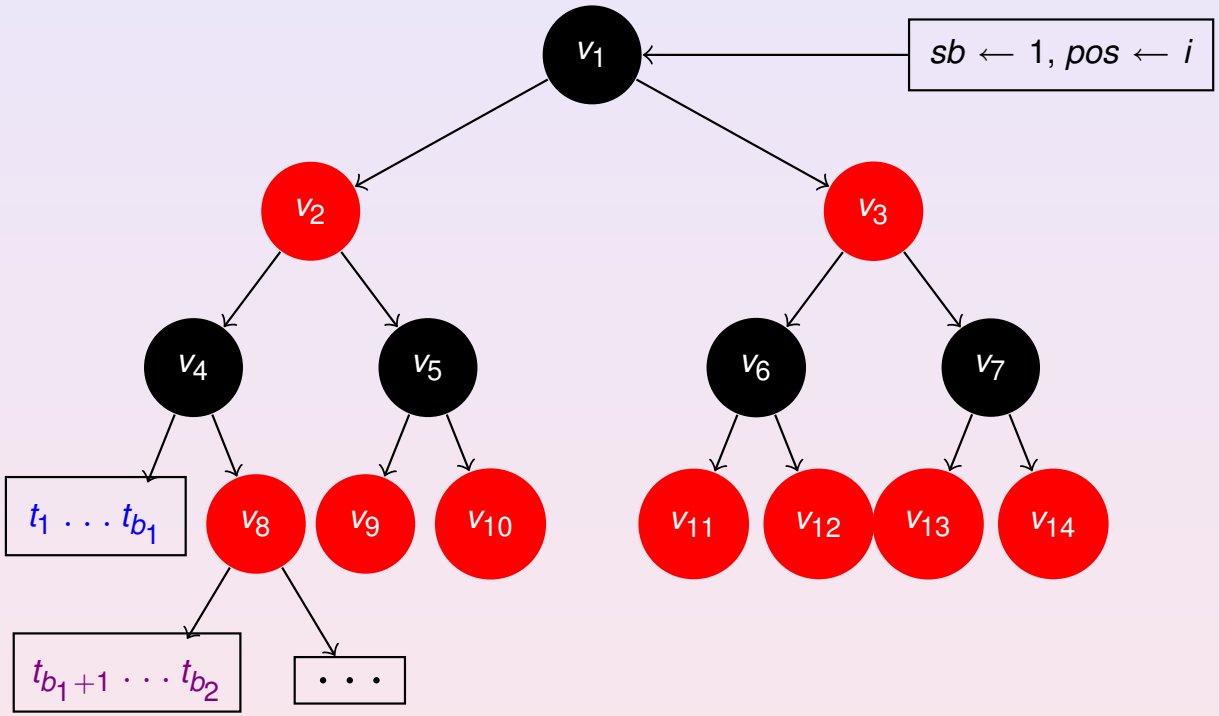




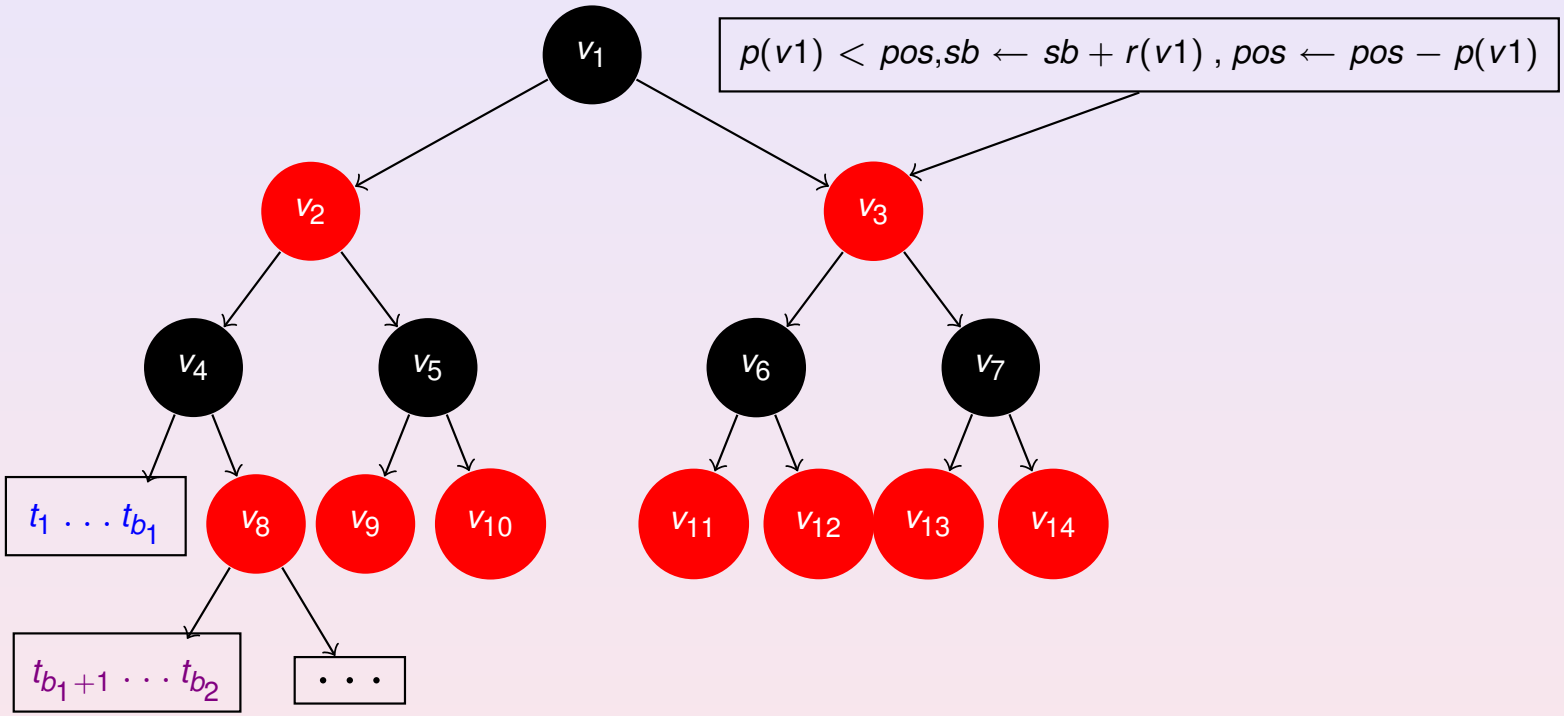
Computing $rank_c(T, i)$



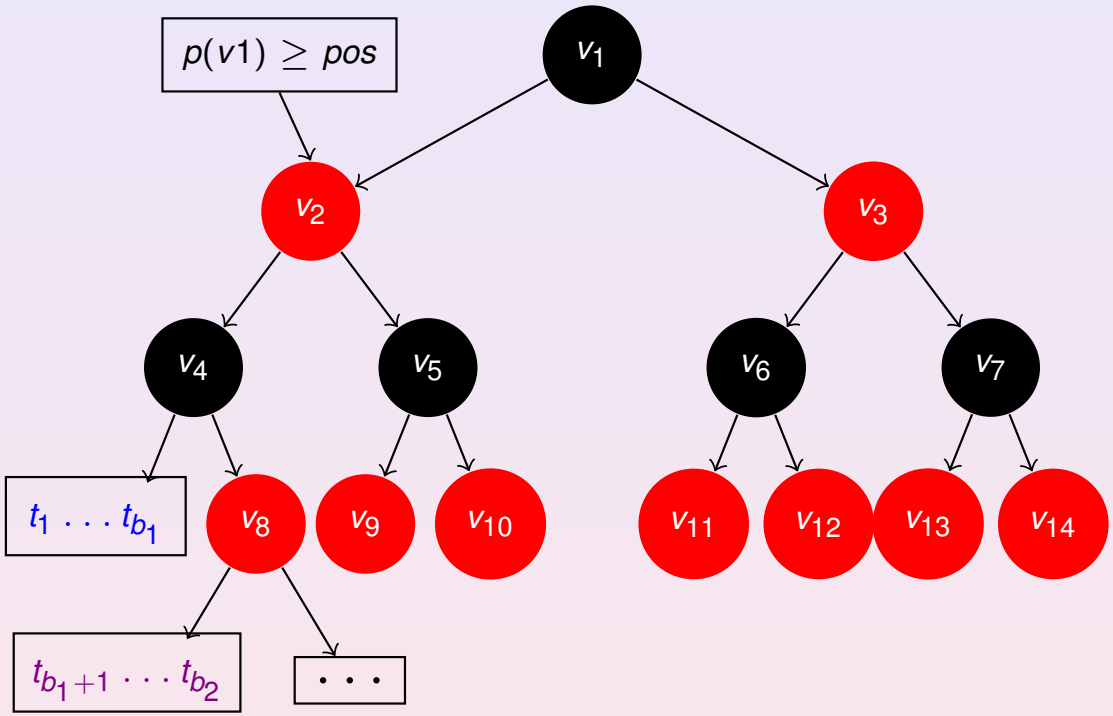
Computing $rank_c(T, i)$



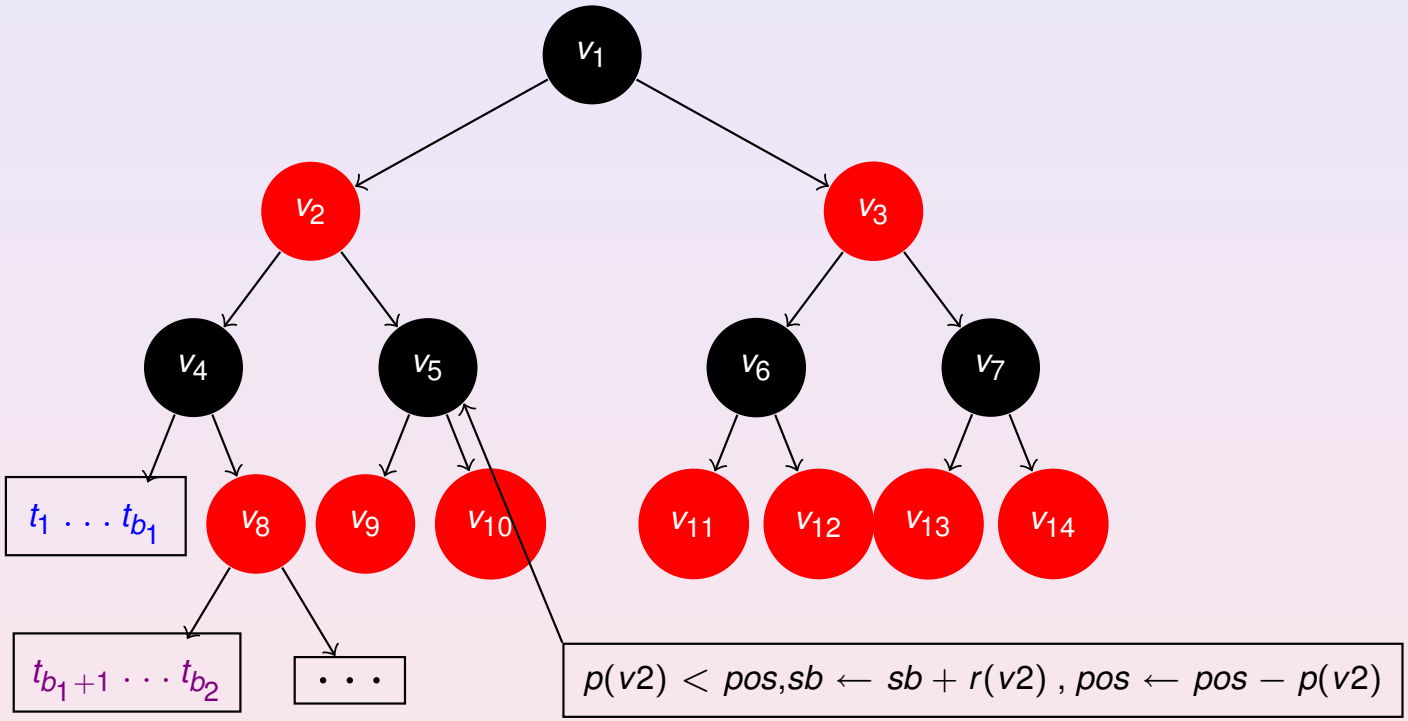
Computing $rank_c(T, i)$



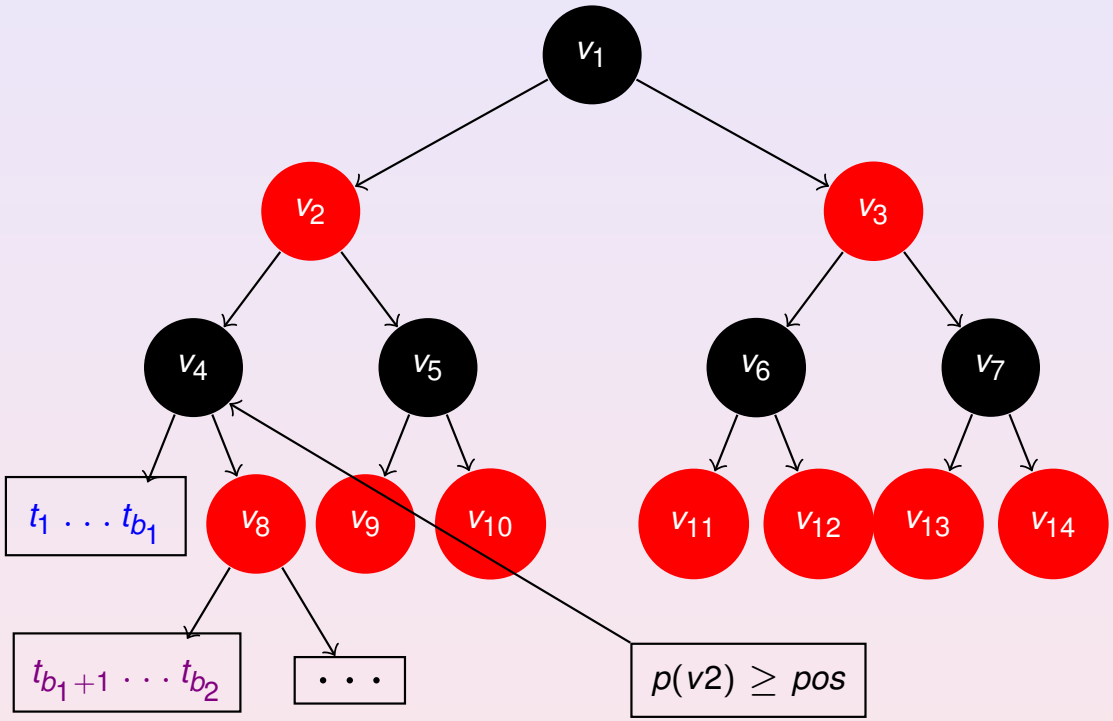
Computing $rank_c(T, i)$



Computing $rank_c(T, i)$



Computing $rank_c(T, i)$



Computing $rank_c(T, i)$

- We obtain that i is the symbol with offset pos within superblock sb.
- We scan superblock sb from the first block summing up the occurrences of c up to the position pos, using a table Z to sum the c's.
- We add to this quantity $sum(C, c, sb - 1)$.

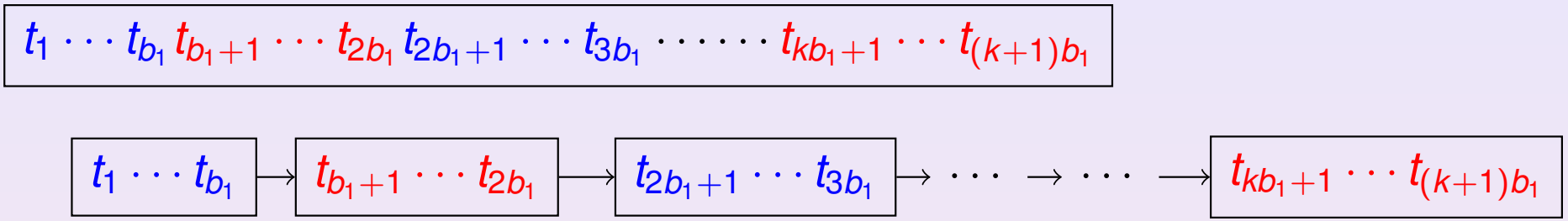
Computing $insert_c(T, i)$

- We obtain *sb* and *pos* similar to *rank* query.
- If superblock *sb* contains room for one more symbol, we insert c right after the *pos*-th position of *sb*, $update(C, c, sb, 1)$ and retraverse the path from the root to *sb* adding 1 to $p(v)$ each time we go left from v.
- If the superblock is full, we try to move one symbol to the previous superblock (creating a new one if this is not possible).
- If superblock $sb - 1$ is also full or does not exist, then we create a sparse superblock between $sb - 1$ and *sb* We create an empty superblock and insert symbol first f symbol of *sb*
- We need to execute $insert(C, sb)$ and $update(C, sb, f, 1)$.

Compressed Dynamic Rank-Select Structure

- Now superblock is be divided into *subblocks* representing $\frac{1}{2}\log n$ bits.
- We represent each subblock using the (c, o)-pair encoding. The c part is of fixed width and tells how many occurrences of each alphabet symbol are there in the subblock; whereas the o part is of variable width and gives the identifier of the subblock among those sharing the same c component.
- Now in a block of $\sqrt{\log n}\log n$ bits, we store as many subblocks as they fit, wasting at most $\sigma \log\log n + \frac{1}{2}\log n$ unused bits at the end.
- We proceed just in the uncompression version but using other tables that process subblocks.
- To keep the structure within the space complexities we need now need that $\sigma = o(\log n/\log\log n)$. In particular $\sigma = \sqrt{\log n}$.

Superblock


$t_1 \cdots t_{b_1}\ t_{b_1+1} \cdots t_{2b_1}\ t_{2b_1+1} \cdots t_{3b_1}\ \cdots\cdots\ t_{kb_1+1} \cdots t_{(k+1)b_1}$
$t_1 \cdots t_{b_1}$
$t_{b_1+1} \cdots t_{2b_1}$
$t_{2b_1+1} \cdots t_{3b_1}$
$\cdots \rightarrow \cdots \rightarrow$
$t_{kb_1+1} \cdots t_{(k+1)b_1}$

Superblock

$$t_1 \cdots t_{b_1}\, t_{b_1+1} \cdots t_{2b_1}\, t_{2b_1+1} \cdots t_{3b_1} \cdots\cdots t_{kb_1+1} \cdots t_{(k+1)b_1}$$

$$t_1 \cdots t_{b_1} \longrightarrow t_{b_1+1} \cdots t_{2b_1} \longrightarrow t_{2b_1+1} \cdots t_{3b_1} \longrightarrow \cdots \longrightarrow \cdots \longrightarrow t_{kb_1+1} \cdots t_{(k+1)b_1}$$

$b_2 = \frac{1}{2} \log n$ and $(k+1)b_1 = (h+1)b_2$

$$t_1 \cdots t_{b_2}\, t_{b_2+1} \cdots t_{2b_2}\, t_{2b_2+1} \cdots t_{3b_2} \cdots\cdots t_{hb_2+1} \cdots t_{(h+1)b_2}$$

Superblock

$$\boxed{t_1 \cdots t_{b_1}\, t_{b_1+1} \cdots t_{2b_1}\, t_{2b_1+1} \cdots t_{3b_1} \cdots\cdots t_{kb_1+1} \cdots t_{(k+1)b_1}}$$

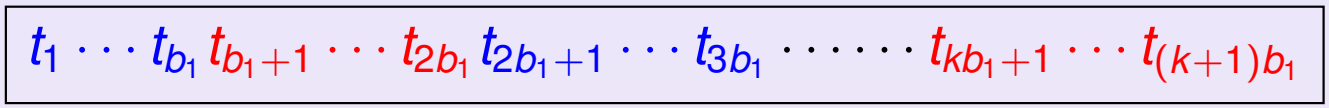
$$\boxed{t_1 \cdots t_{b_1}} \rightarrow \boxed{t_{b_1+1} \cdots t_{2b_1}} \rightarrow \boxed{t_{2b_1+1} \cdots t_{3b_1}} \rightarrow \cdots \rightarrow \cdots \rightarrow \boxed{t_{kb_1+1} \cdots t_{(k+1)b_1}}$$

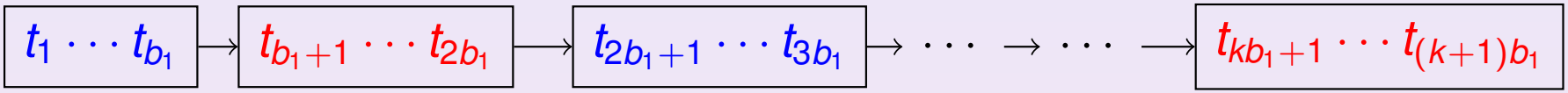
$$b_2 = \tfrac{1}{2}\log n \text{ and } (k+1)b_1 = (h+1)b_2$$

$$\boxed{t_1 \cdots t_{b_2}\, t_{b_2+1} \cdots t_{2b_2}\, t_{2b_2+1} \cdots t_{3b_2} \cdots\cdots t_{kh_2+1} \cdots t_{(h+1)b_2}}$$

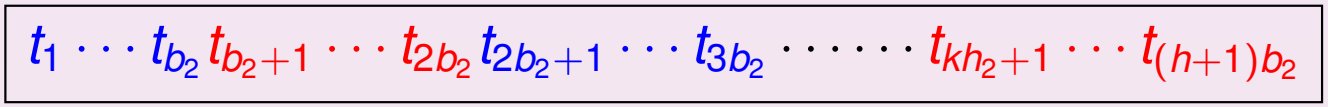
$$\boxed{(c_1, o_1)(c_2, o_2)(c_3, o_3) \cdots\cdots (c_h, o_h)}$$

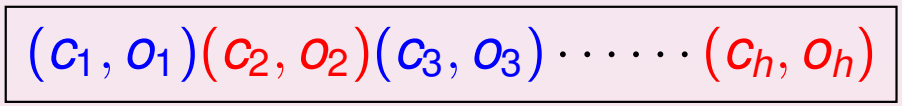
Superblock

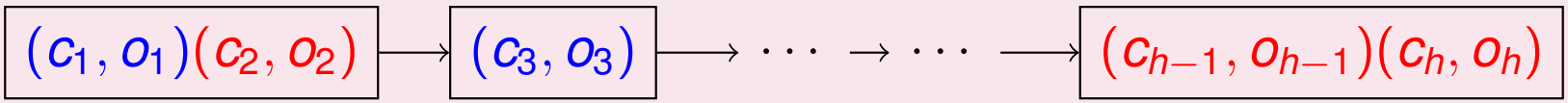




$b_2 = \frac{1}{2} \log n$ and $(k+1)b_1 = (h+1)b_2$







Extend our result to large alphabet

- We use a generalized ρ-ary wavelet tree [FMMN-TALG, 2007] over T, where $\rho = \Theta(\sqrt{\log n})$.
- Essentially, this generalized wavelet tree has $O(\log_\rho \sigma) = O(\frac{\log \sigma}{\log \log n})$ levels.
- We store on each level a sequence over an alphabet of size ρ, which can be handled using the dynamic solution
- Each query and operation takes $O(\log n)$ time per level, adding up $O(\log n \frac{\log \sigma}{\log \log n})$ worst-case time overall.

Main result

Theorem

Given a text T of length n over an alphabet of size σ and zero-order entropy $H_0(T)$, the Dynamic Sequence with Indels problem under RAM model with word size $w = \Omega(\log n)$ can be solved using $nH_0(T) + O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits of space, supporting the queries access, rank, select, insert and delete in $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ worst-case time.

Conclusions

- We have shown that the best two existing solutions to the *Dynamic Sequence with Indels* problem can be merged so as to obtain the best from both.
- This structure can be used in dynamic structures that use rank and select, like dynamic text index.
- Sub-logarithmic query times are not possible with $O(\mathrm{polylog}(n))$ update times, and hence $O(\log n)$ is the best we can hope for if we want to minimize the maximum complexity over all the operations.