FM-Index and the Reverse Prefix Trie

Jouni Sirén
Wellcome Trust Sanger Institute

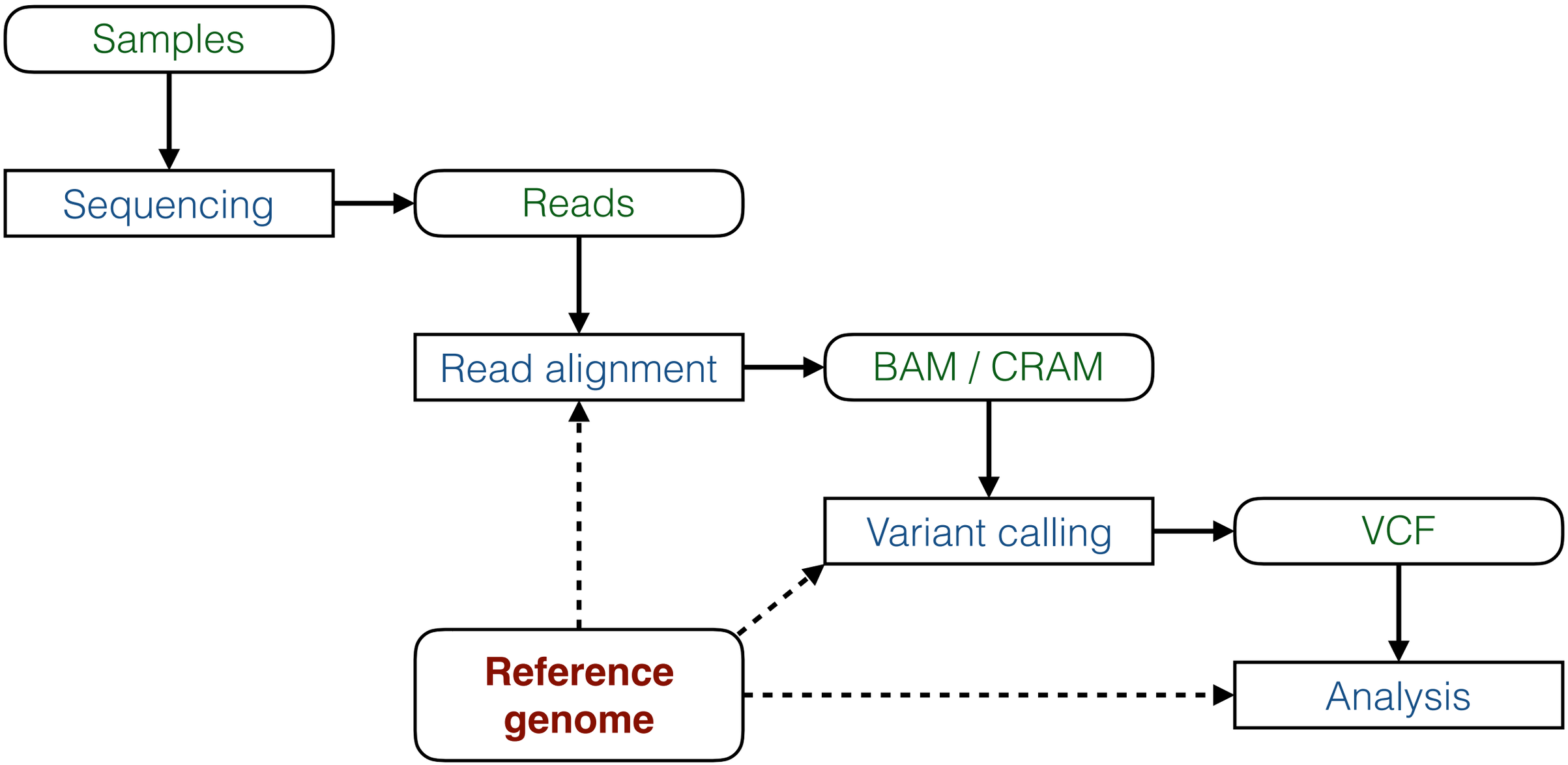
This lecture was part of the 1st Summer School on Bioinformatics Data Structures, funded by BIRDS project (www.birdsproject.eu)
This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 690941

Contents

1. Reference-free analysis
2. FM-index and the reverse prefix trie
3. Algorithms using the trie
4. Kmer counting
5. All-against-all comparison of sequence collections
6. BWT merging

Reference-Free Analysis

Typical pipeline



Reference bias

- Read alignment, variant calling, and the subsequent analysis all depend on the reference genome.

- Most reference genomes are based on the genomes of a small number of individuals or populations.

- The analysis may be biased towards those individuals and populations.

Avoiding reference bias

1. De novo assembly of individual genomes (not really possible with Illumina reads)
2. Reference-free analysis based on the reads (today)
3. Using a graph reference (tomorrow)
4. Analysis based on assembly graphs (a combination of the above)

Reference-free analysis

- A single high-coverage human sample is around 100 Gbp in 1 billion sequences.

- Large projects sequence thousands of samples.

- Data structure construction is a major bottleneck, often taking many CPU years and weeks of real time.

- We must commit to our data structure choices before we know precisely what we are going to do.

- Versatility and the ease of construction must be key properties of our data structures.

Population BWT

Dirk D. Dolle et al: Using reference-free compressed data structures to analyse sequencing reads from thousands of human genomes. bioRxiv, 2016.

→ 1000 GP phase 3: 922G reads, 87.1 Tbp.
→ Error correction & trimming: 819G reads, 75.5 Tbp.
→ 16 FM-indexes: 53.0G distinct sequences, 4.88 Tbp, 561.5 GB.
→ Metadata: 4.75 TB.

This talk is largely based on ideas how to develop the population BWT further and to use it more effectively.

FM-Index and the Reverse Prefix Trie

Definitions

- Better to use the same definitions both in papers and in implementations.

- Array indices start from 0: S[0…n–1].

- S.rank(i, c) is the number of occurrences of character c in the prefix S[0…i–1].

- S.select(i, c) is the position of the ith occurrence of character c (the last position j where S.rank(j, c) < i).

- Time complexities indicate the number of rank / select operations.

Burrows-Wheeler transform

- Add a unique terminator ($) to the end of the text, sort the suffixes in lexicographic order, and output the preceding character for each suffix.

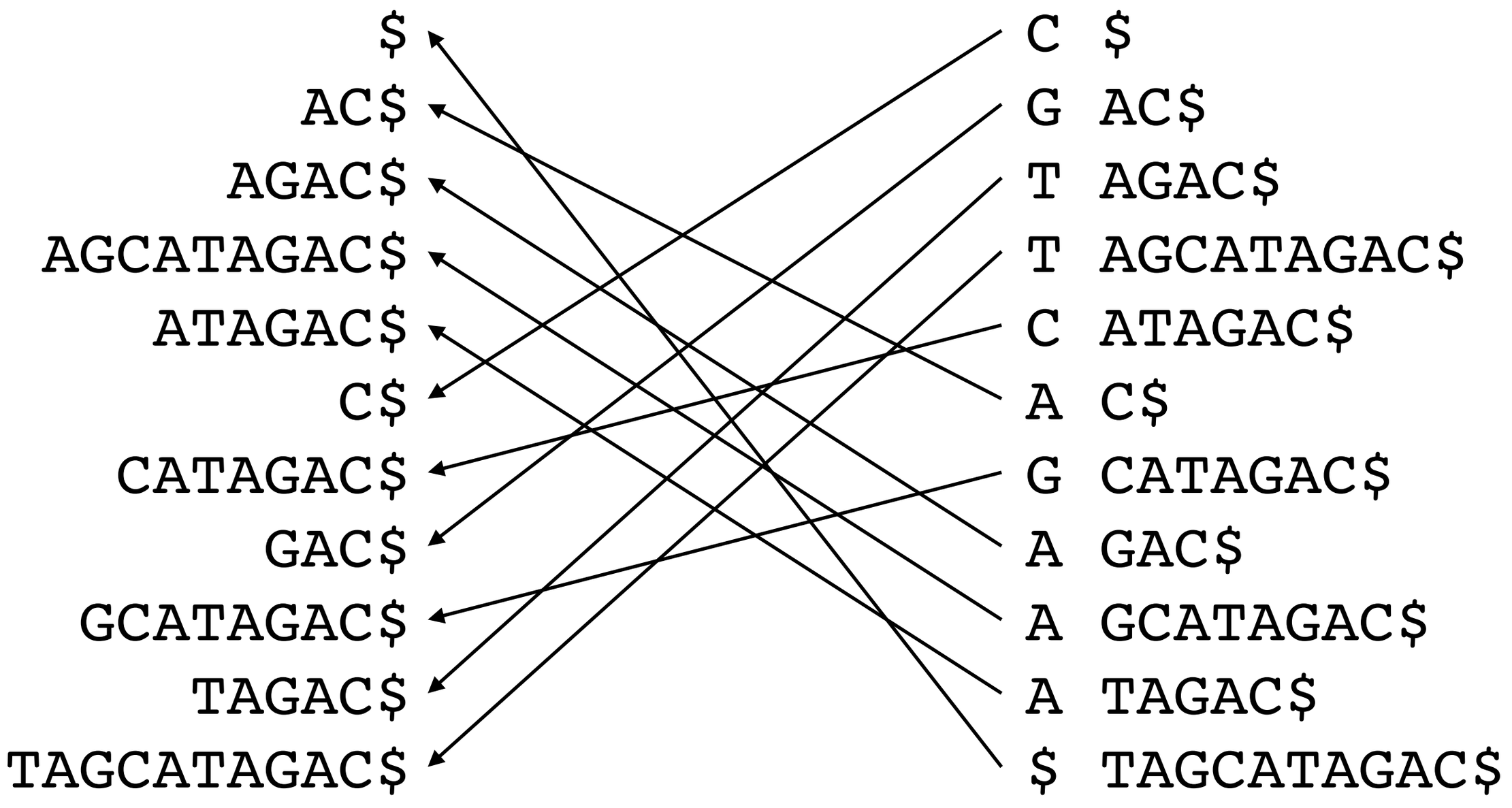
- Use distinct terminators for multiple texts.

- The permutation is easily reversible and makes the text easier to compress (Burrows & Wheeler, 1994).

- The combinatorial structure is similar to the suffix array, which makes the BWT useful as a space-efficient text index (Ferragina & Manzini, 2000, 2005).

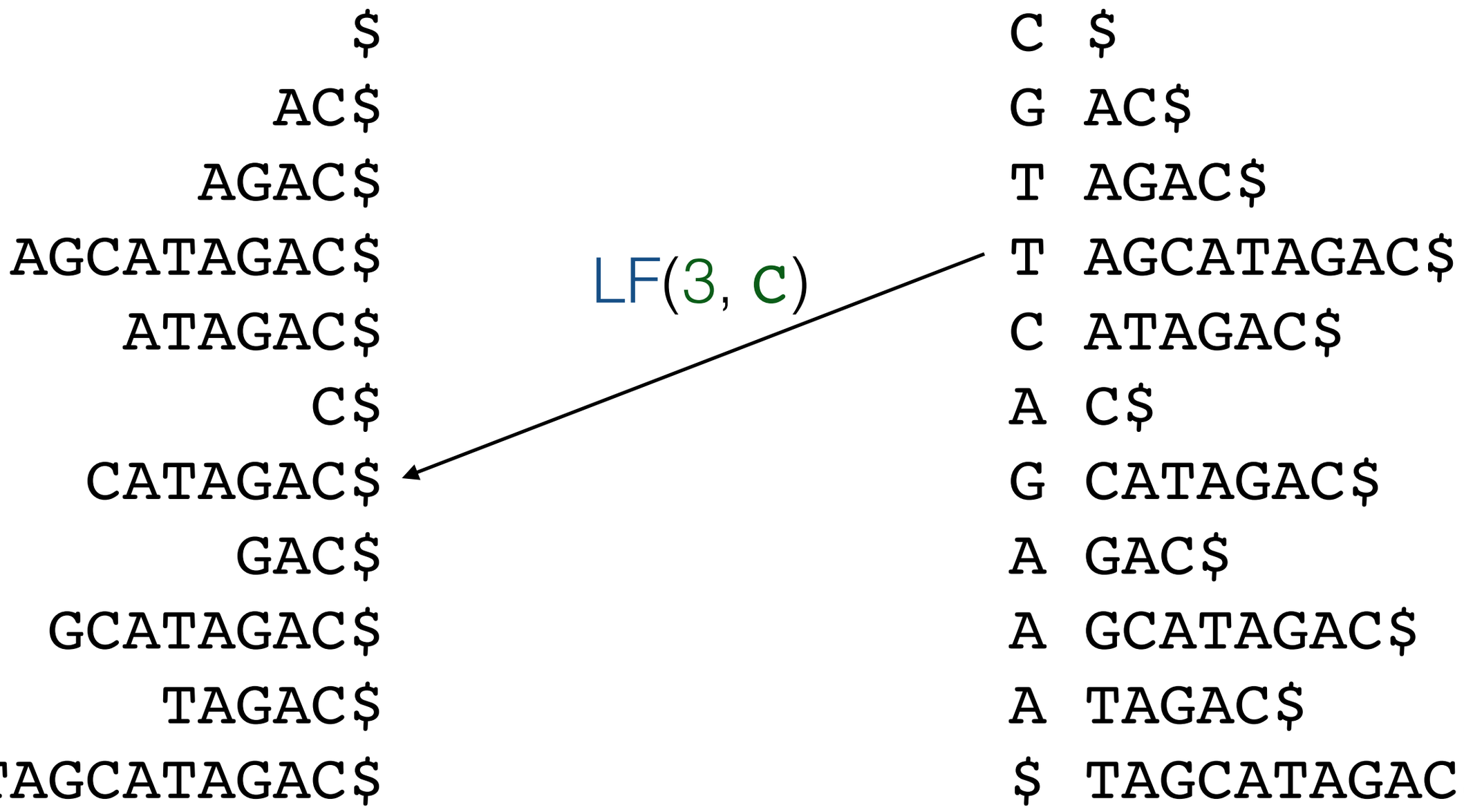
```
TAGCATAGAC$

C $
G AC$
T AGAC$
T AGCATAGAC$
C ATAGAC$
A C$
G CATAGAC$
A GAC$
A GCATAGAC$
A TAGAC$
$ TAGCATAGAC$
```

LF-mapping



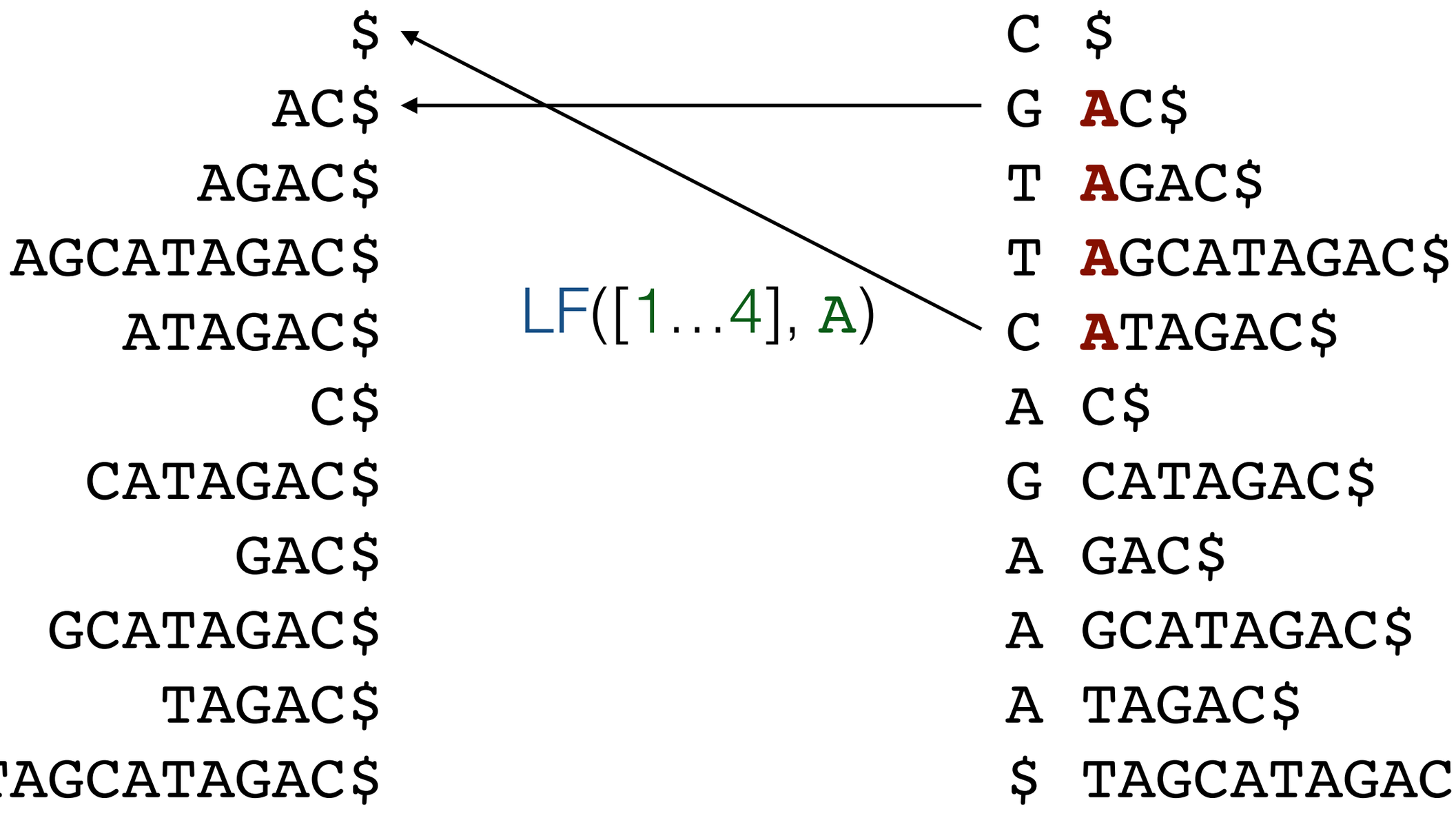
LF(i) = C[BWT[i]] + BWT.rank(i, BWT[i])

Hypothetical suffixes



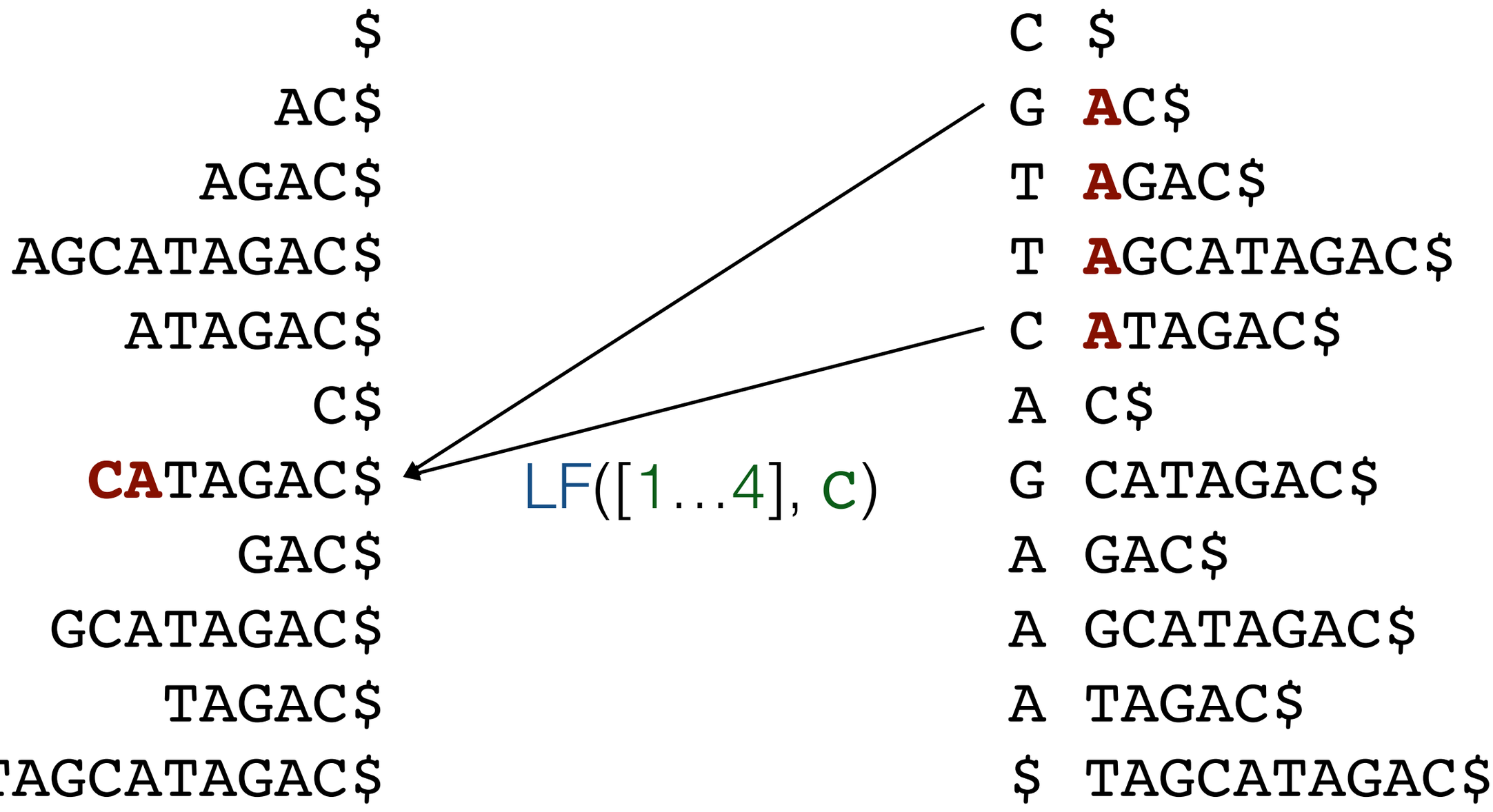
Interpretation: LF(i, c) = C[c] + BWT.rank(i, c) suffixes are **strictly before** the hypothetical suffix.

Backward searching



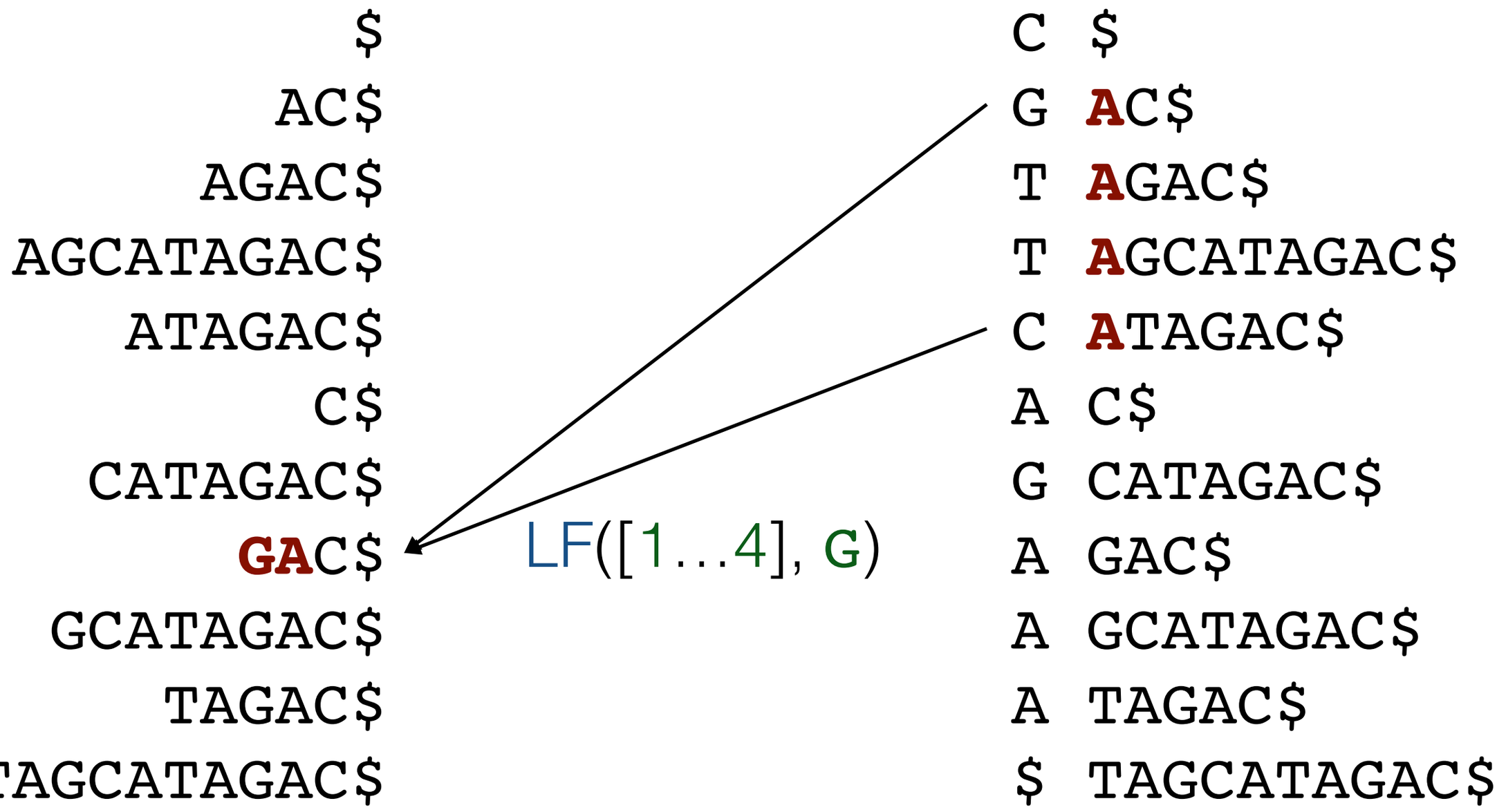
LF([sp…ep], c) = [LF(sp, c)…LF(ep+1, c) −1]

Backward searching



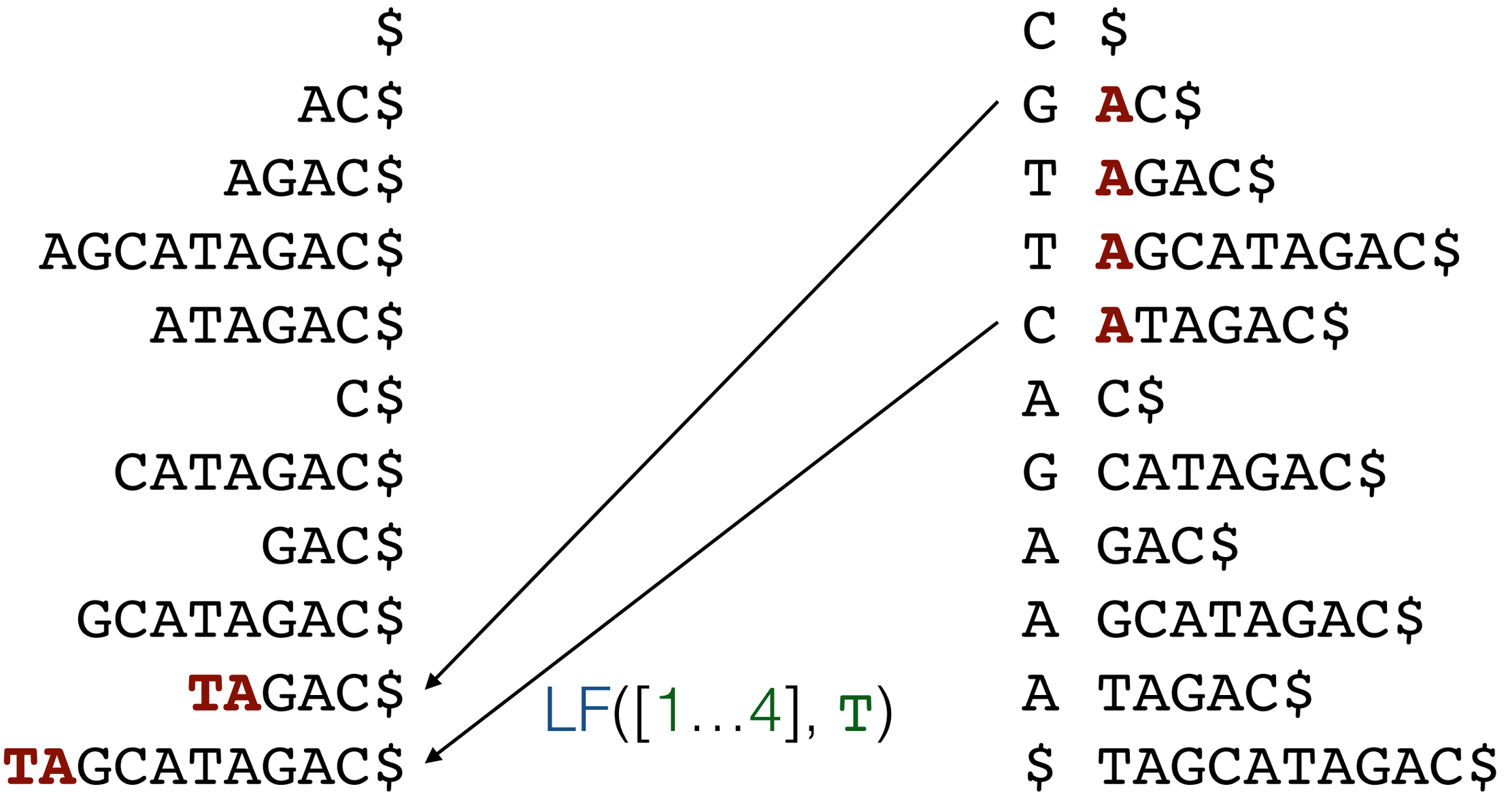
LF([sp…ep], c) = [LF(sp, c)…LF(ep+1, c) −1]

Backward searching

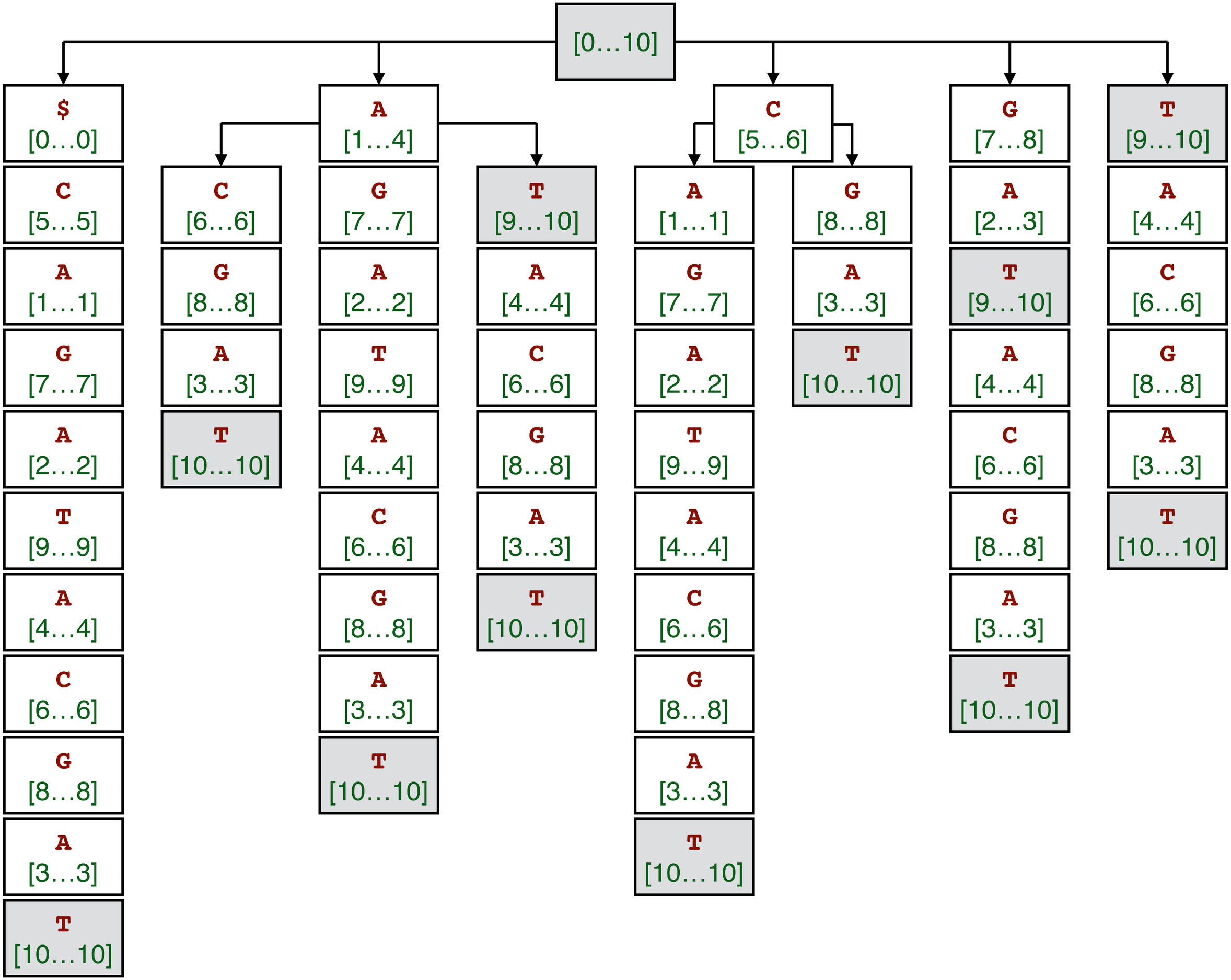


LF([sp…ep], c) = [LF(sp, c)…LF(ep+1, c) −1]

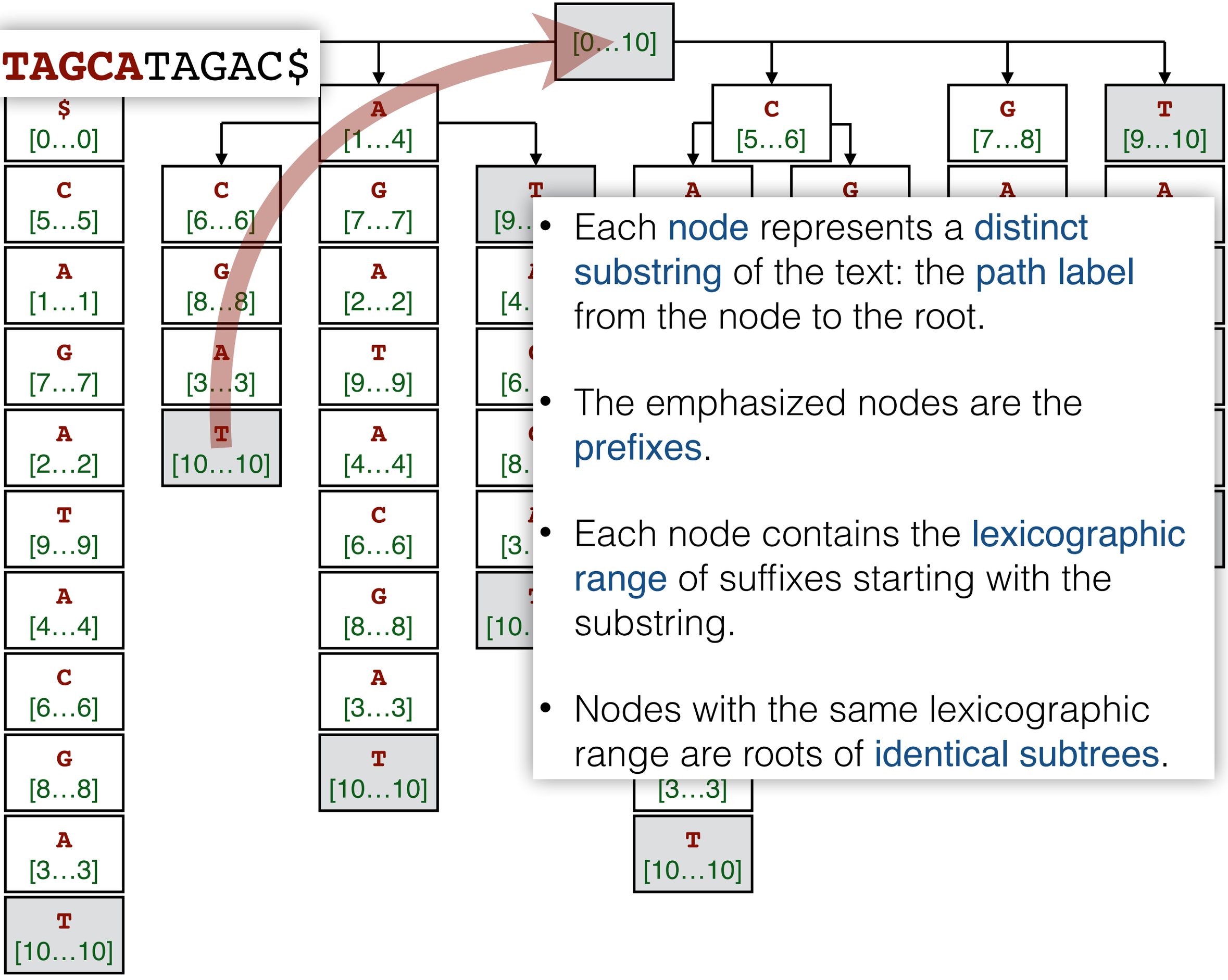
Backward searching



LF([sp…ep], c) = [LF(sp, c)…LF(ep+1, c) –1]


[0…10]
$ [0…0]
C [5…5]
A [1…1]
G [7…7]
A [2…2]
T [9…9]
A [4…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
A [1…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
G [7…7]
A [2…2]
T [9…9]
A [4…4]
T [9…10]
A [4…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
C [5…6]
A [1…1]
G [7…7]
A [2…2]
T [9…9]
G [8…8]
A [3…3]
T [10…10]
G [7…8]
A [2…3]
T [9…10]
A [4…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
T [9…10]
A [4…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]

TAGCATAGAC$



- Each node represents a distinct substring of the text: the path label from the node to the root.
- The emphasized nodes are the prefixes.
- Each node contains the lexicographic range of suffixes starting with the substring.
- Nodes with the same lexicographic range are roots of identical subtrees.

Algorithms Using the Trie

Reverse prefix trie algorithms

- Many algorithms using the FM-index can be understood as traversals of the reverse prefix trie.

- It is often easier to forget the FM-index and think about the trie instead.

- As an introduction, we will take a look at approximate searching (as in the old BWA) and bidirectional BWT.

Approximate searching

Mismatch

```
TAGCATAGAC$
TAGCATCGAC$
```

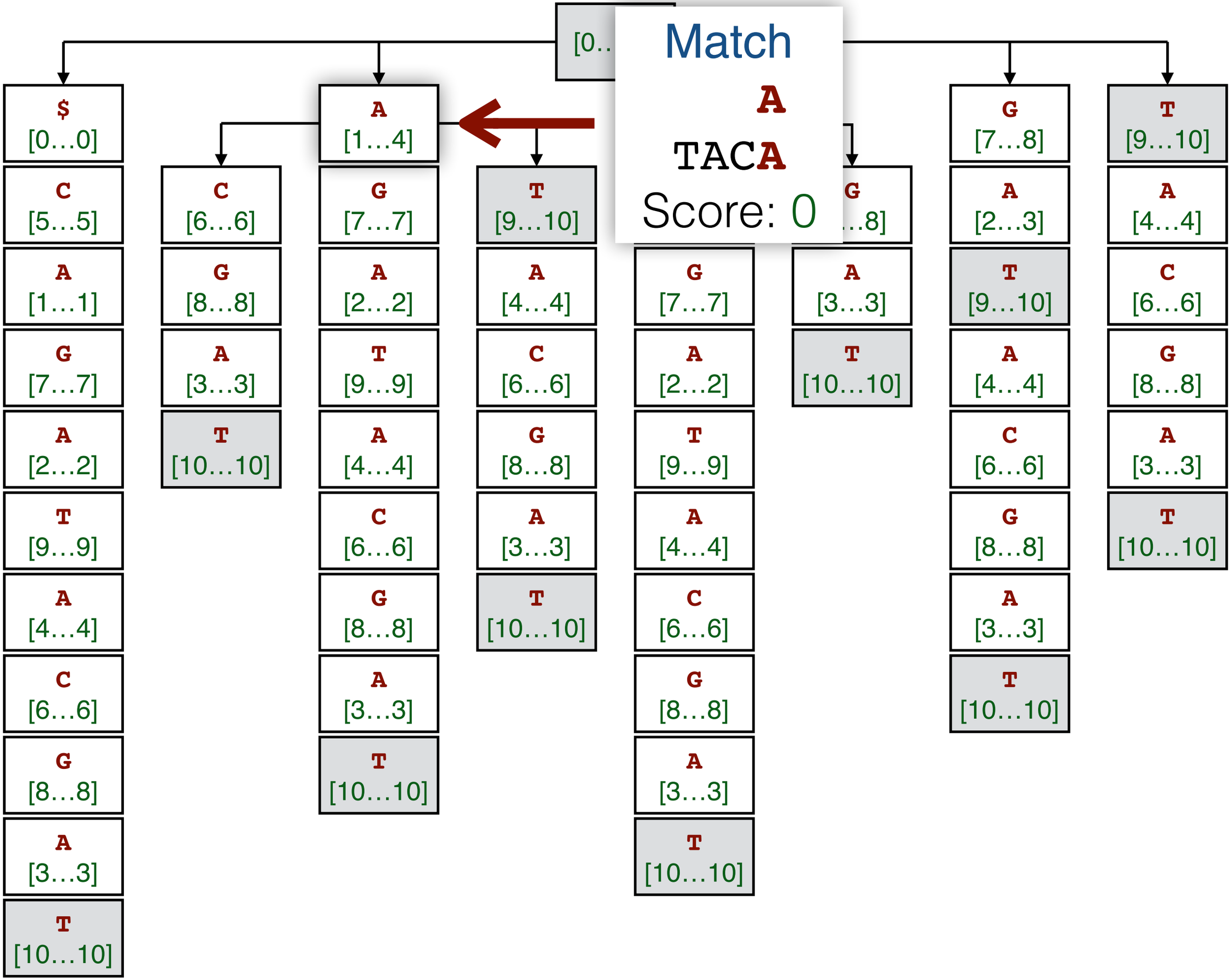
Insertion

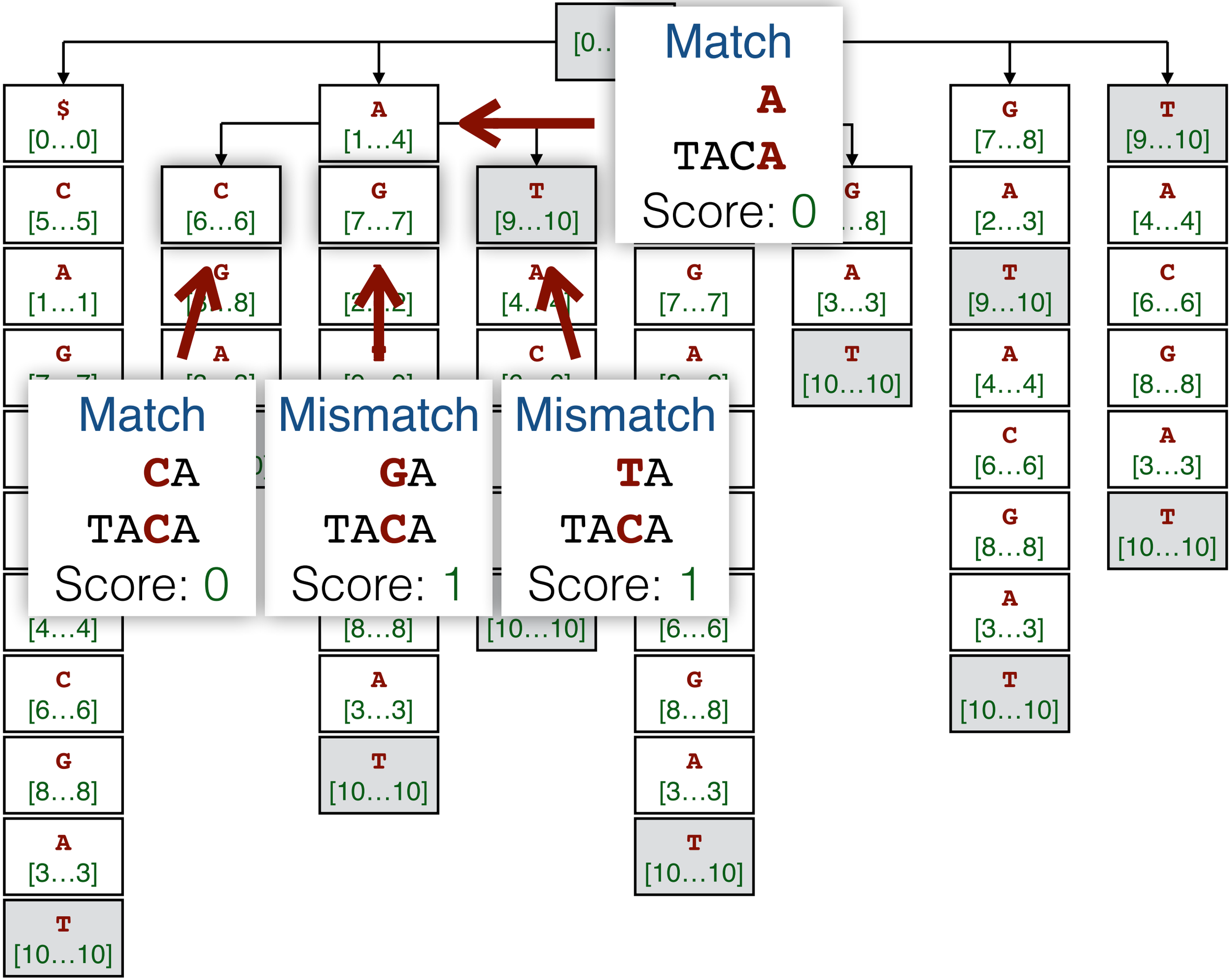
```
TAGCATA-GAC$
TAGCATAGGAC$
```

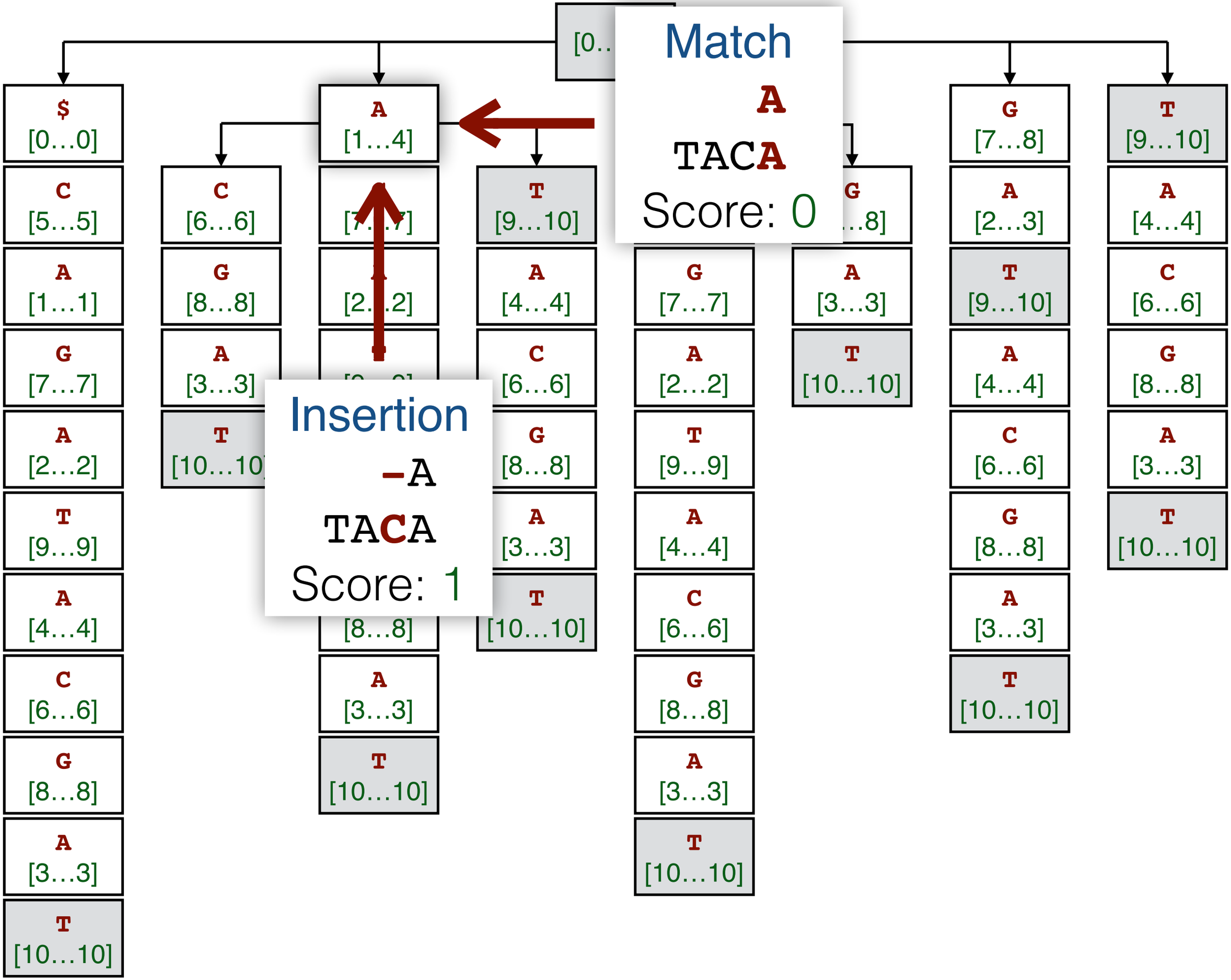
Deletion

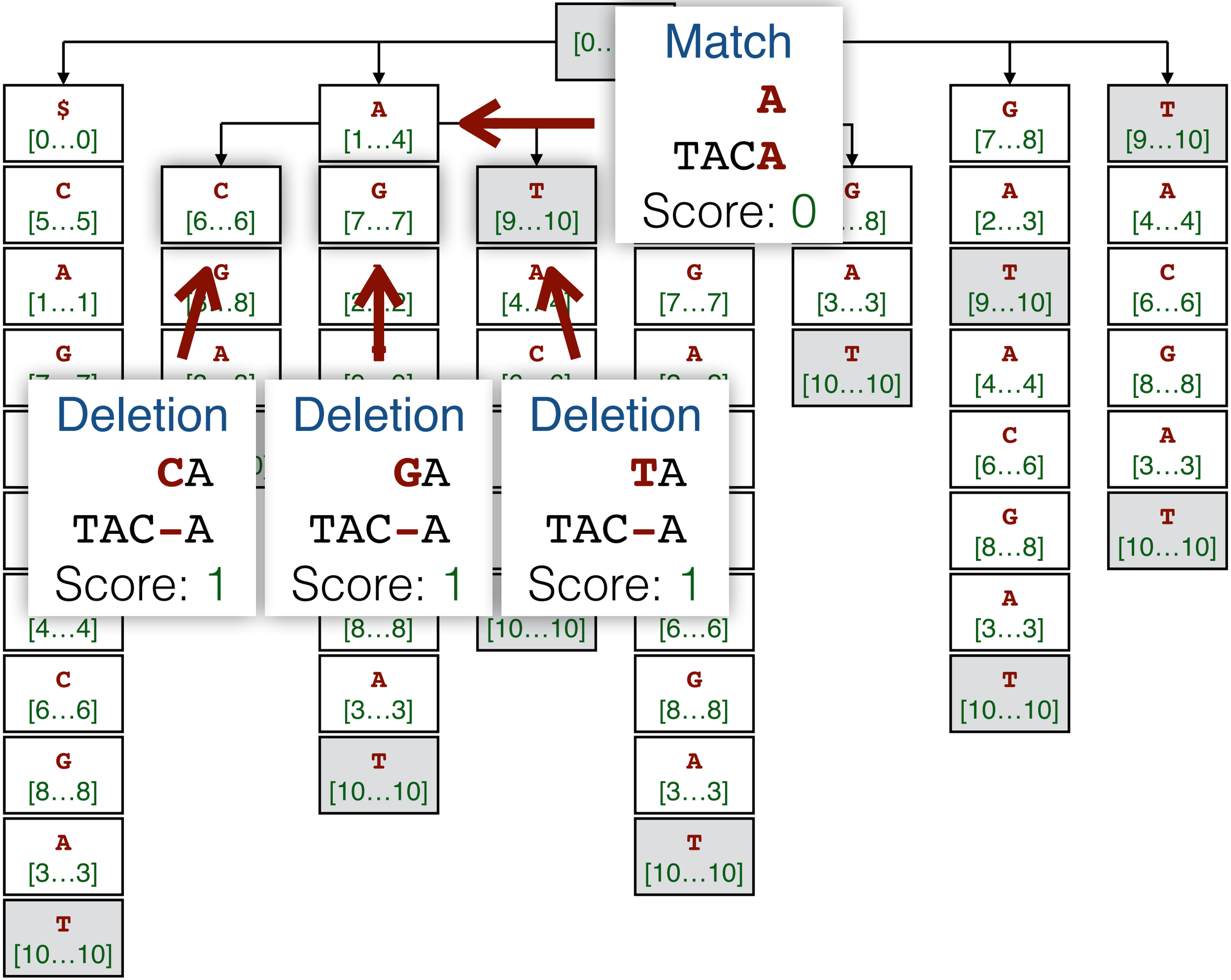
```
TAGCATAGAC$
TAGCA-AGAC$
```

- The edit distance between strings A and B is the number of edit operations required to transform A into B.
- In approximate searching, we want to find the substring of the text with the smallest edit distance to the pattern.
- In practice, we want to minimize (or maximize) the score function between the substring and the pattern.


Match
A
TACA
Score: 0
$ [0…0]
C [5…5]
A [1…1]
G [7…7]
A [2…2]
T [9…9]
A [4…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
A [1…4]
G [7…7]
A [2…2]
T [9…9]
A [4…4]
T [10…10]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
T [9…10]
G [7…7]
A [2…2]
T [9…9]
A [4…4]
G [8…8]
A [3…3]
T [10…10]
G [7…8]
A [2…3]
T [9…10]
A [4…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
T [9…10]
A [4…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]


Match
A
TACA
Score: 0
Match
CA
TACA
Score: 0
Mismatch
GA
TACA
Score: 1
Mismatch
TA
TACA
Score: 1


Match
A
TACA
Score: 0
Insertion
-A
TACA
Score: 1


Match
A
TACA
Score: 0
Deletion
CA
TAC-A
Score: 1
Deletion
GA
TAC-A
Score: 1
Deletion
TA
TAC-A
Score: 1

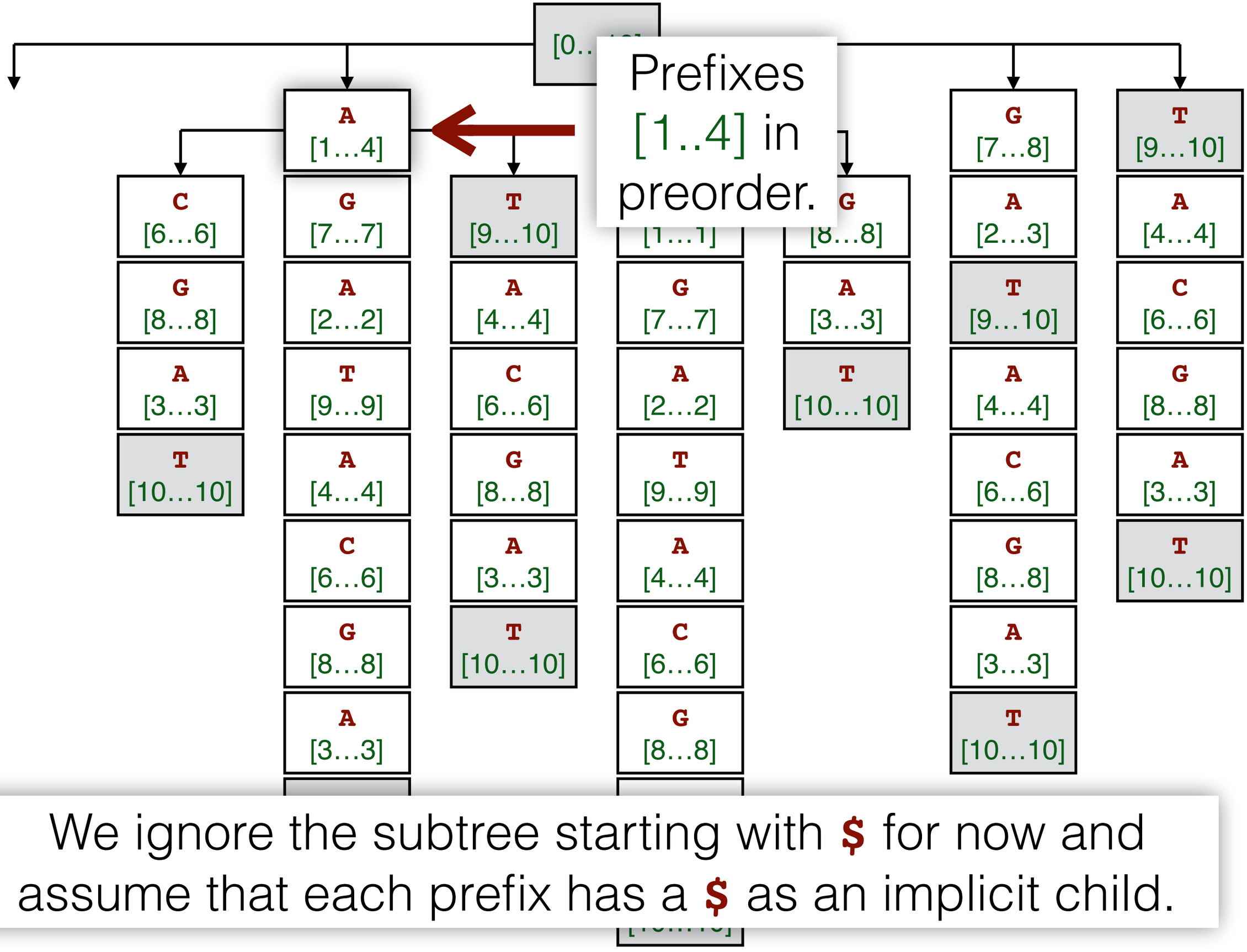
Approximate searching

- We traverse a tree of search states: lexicographic range, matched suffix, score, edit operations.

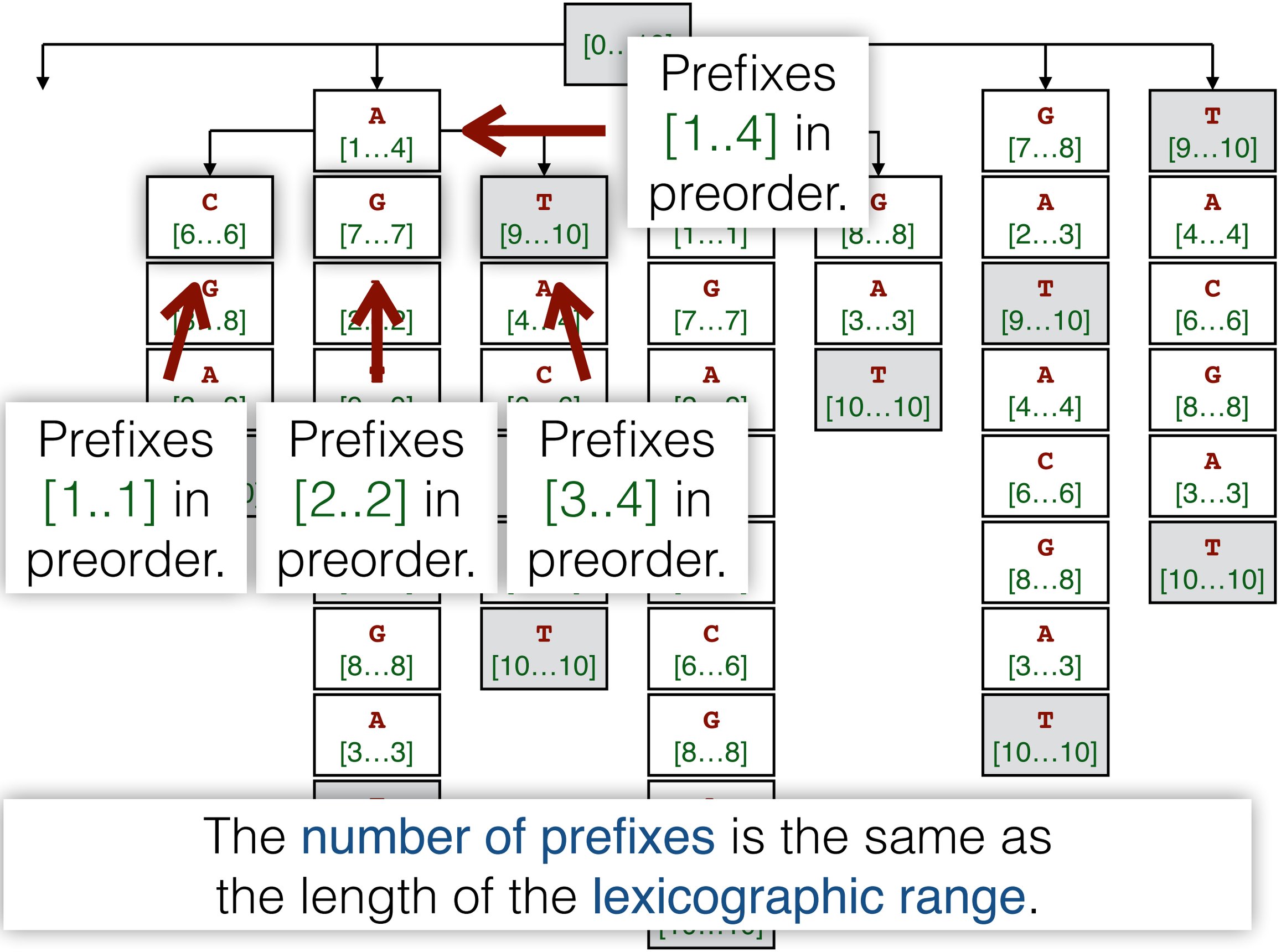
- Use an oracle to give a lower bound for the score of a full match expanded from the current state.

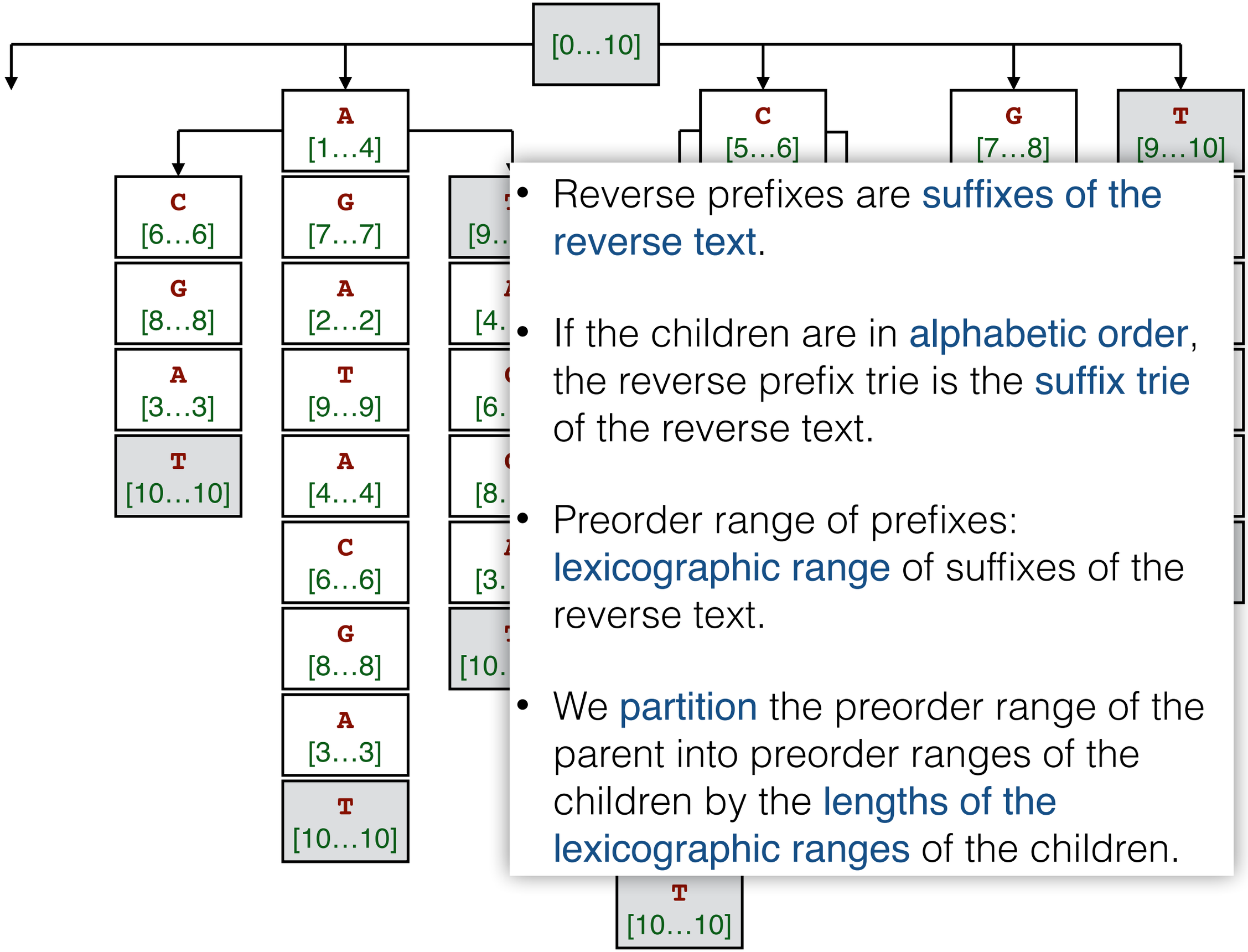
- Place the states into a priority queue by the lower bounds and use A* search (most promising first) to find the best match.

- This is essentially the backtracking algorithm used in bwa aln (Li & Durbin, 2009).

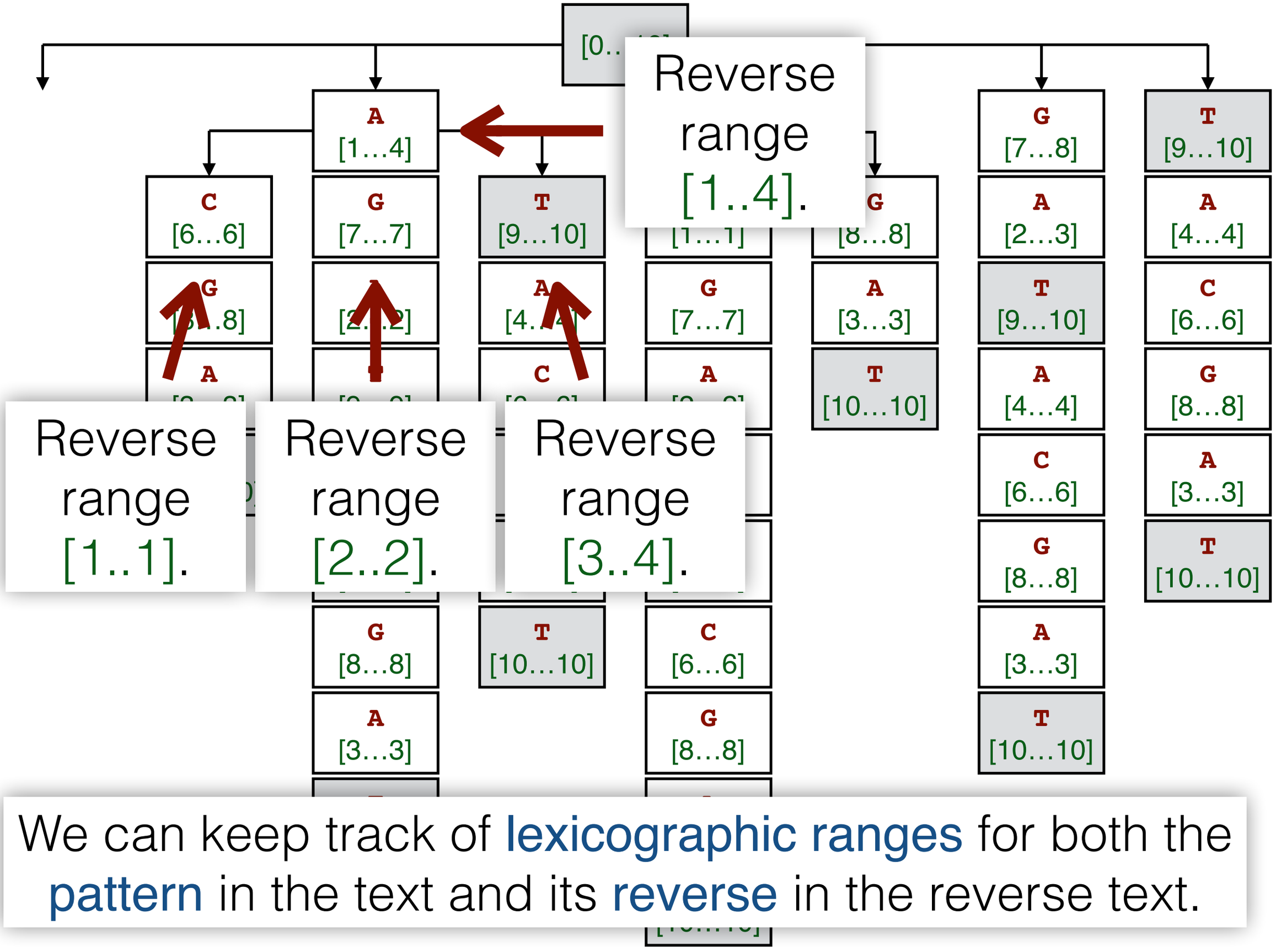


We ignore the subtree starting with **$** for now and assume that each prefix has a **$** as an implicit child.


Prefixes [1..4] in preorder.
Prefixes [1..1] in preorder.
Prefixes [2..2] in preorder.
Prefixes [3..4] in preorder.
The number of prefixes is the same as the length of the lexicographic range.

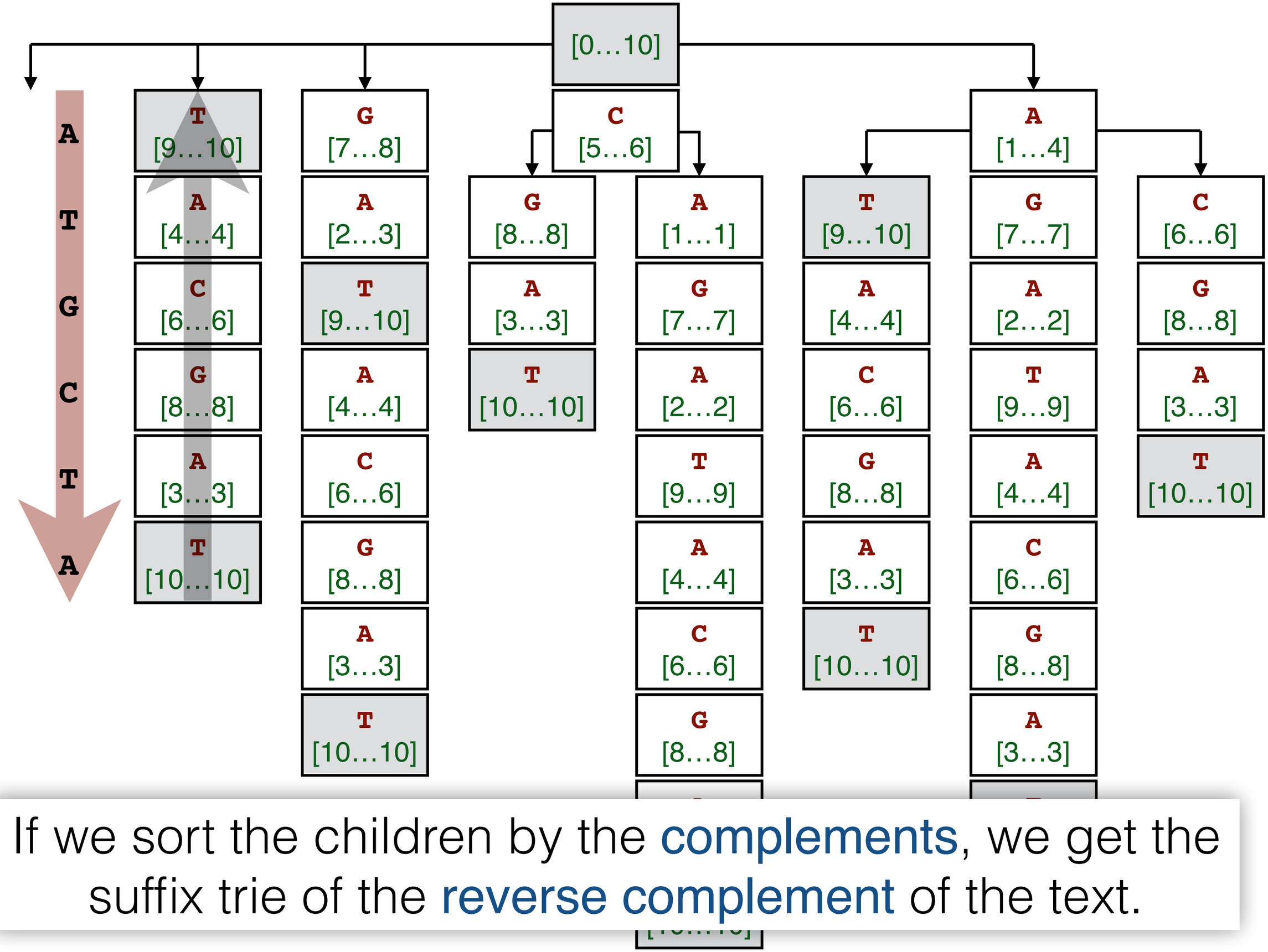


- Reverse prefixes are suffixes of the reverse text.
- If the children are in alphabetic order, the reverse prefix trie is the suffix trie of the reverse text.
- Preorder range of prefixes: lexicographic range of suffixes of the reverse text.
- We partition the preorder range of the parent into preorder ranges of the children by the lengths of the lexicographic ranges of the children.


Reverse range [1..4].
Reverse range [1..1].
Reverse range [2..2].
Reverse range [3..4].
We can keep track of lexicographic ranges for both the pattern in the text and its reverse in the reverse text.

Bidirectional searching

- We have one trie / FM-index for the **text** and another for the reverse.
- The lexicographic range in one trie is the reverse range in another.
- Backward searching in one FM-index extends the match forward in another.
- This combination of indexes is frequently called the bidirectional BWT (Lam et al, 2009).



If we sort the children by the complements, we get the suffix trie of the reverse complement of the text.

FMD-index

- The FMD-index (Li 2012) has both the text and its reverse complement in the same FM-index.

- This saves time, as we often search for both pattern and its reverse complement.

- Bidirectional search: the reverse range for the pattern is the lexicographic range for the reverse complement of the pattern.

- Used in e.g. BWA-MEM (Li 2013) to find maximal exact matches.

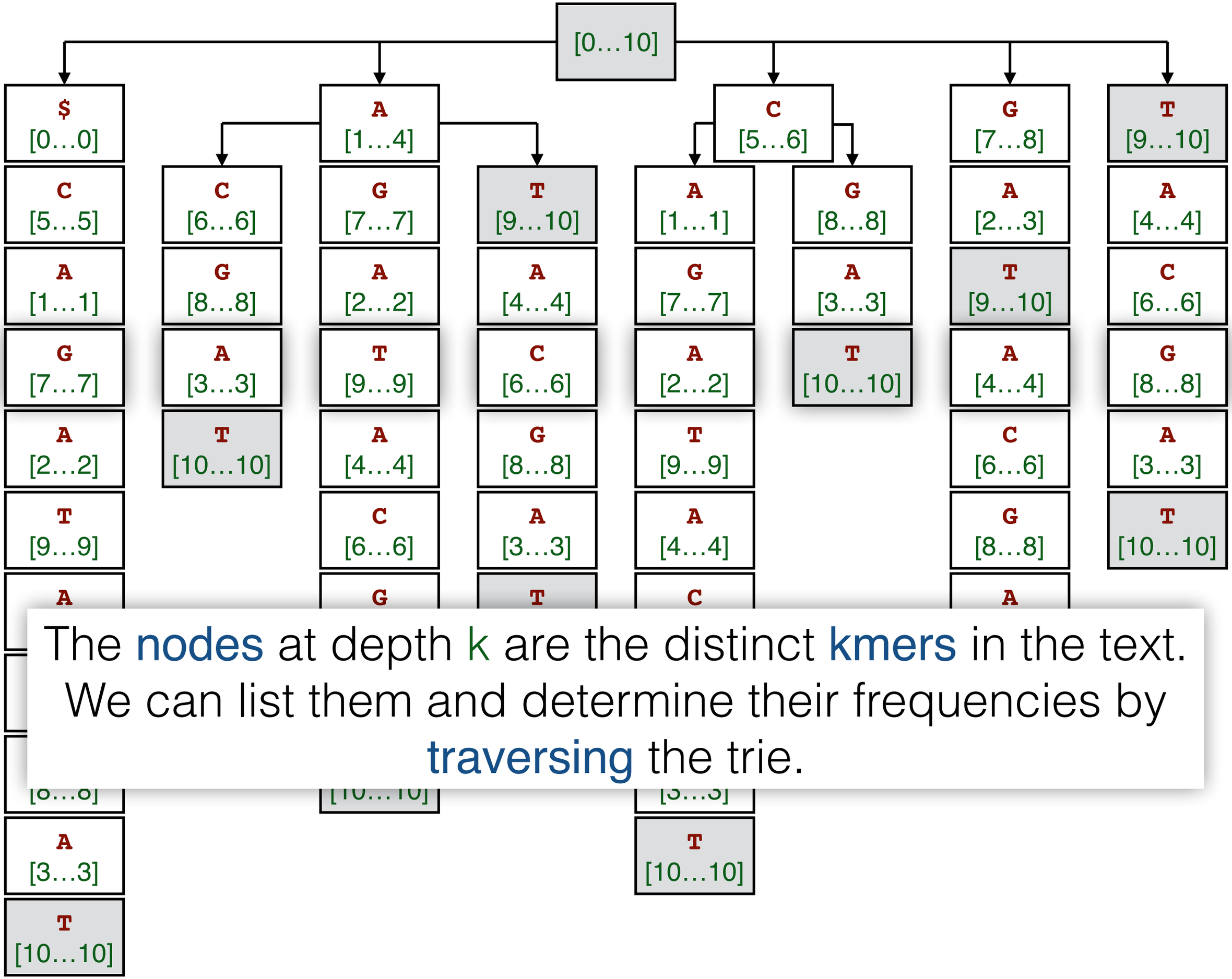
Kmer Counting

Why kmer counting?

- Kmers are easy to understand.
- Determining the kmers and their frequencies in a sequence collection is a common task in bioinformatics.
- Kmers are used for e.g. error correction, indexing, de Bruijn graph construction, genome size / read coverage estimation…

Kmer counting with FM-index

- With an FM-index, the hard part is already done.
- The counting algorithm is reasonably fast and easy to parallelize.
- Uses existing data structures and requires very little additional code.
- Particularly fast with repetitive sequence collections.



The nodes at depth k are the distinct kmers in the text. We can list them and determine their frequencies by traversing the trie.

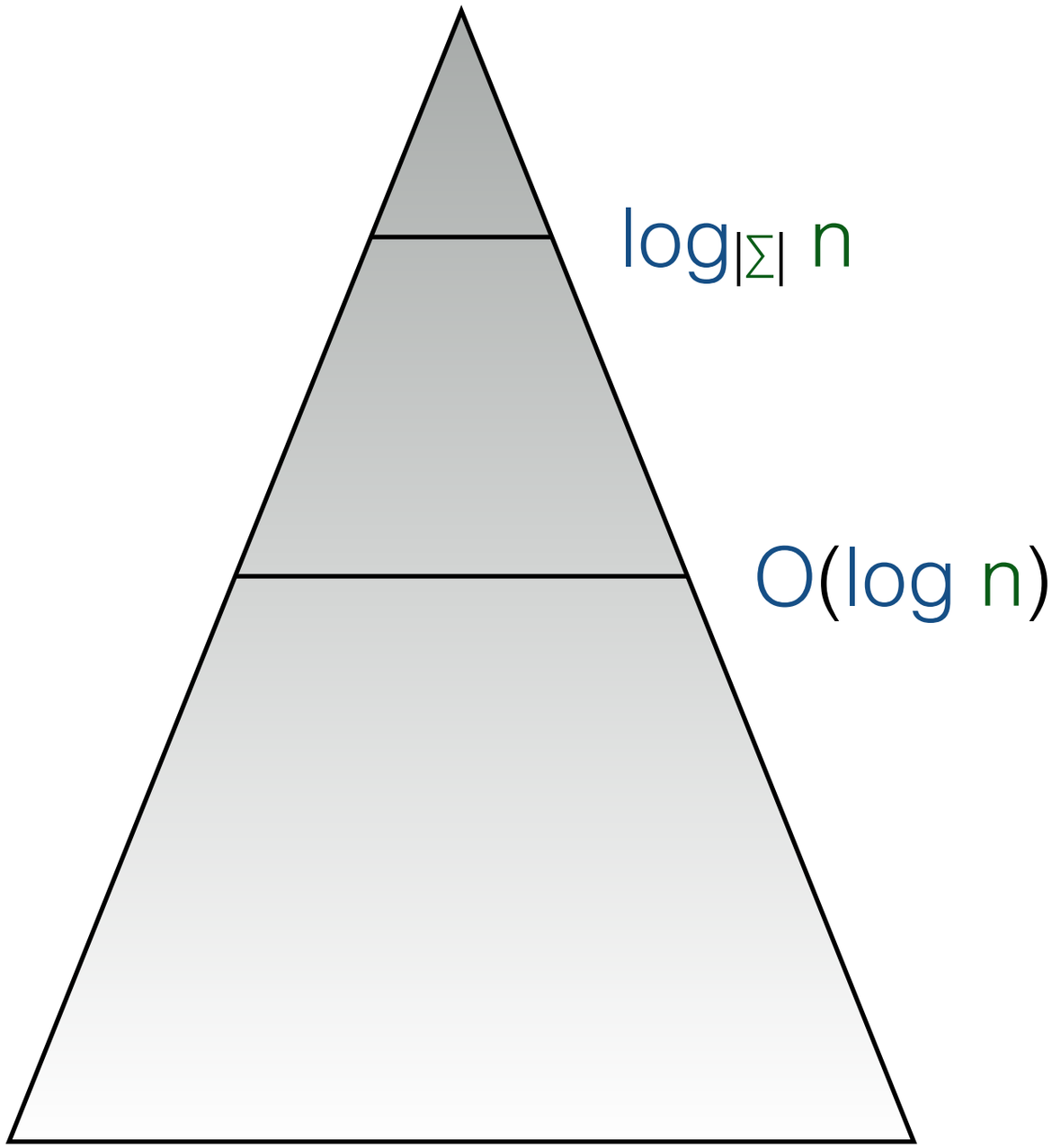
Basic algorithm

```
function traverse(k):
    S.push(ε, [0…|BWT| –1])
    while S ≠ ∅:
        (X, [sp…ep]) ← S.pop()
    if [sp…ep] = ∅:
        continue
    if |X| = k:
        report(X, [sp…ep])
    if |X| < k:
        for c ∈ ∑:
            S.push(cX, LF([sp…ep], c))
```

Multithreading:

- Use traverse(k') for k' < k to generate seed sequences.
- Traverse the resulting subtrees in separate threads.

Time complexity



Dense part: Most kmers exist, O(1) / kmer.

Interesting part: Much branching, most kmers missing.

Sparse part: Unary paths, O(k) / kmer. Better to use other algorithms.

All-Against-All Comparison of Sequence Collections

All-against-all comparison

- We have two FM-indexes containing e.g. assembled genomes, unitigs, or reads.

- Traverse both trees at the same time.

- List the kmers that are specific to / frequent in one of the collections and missing from / rare in the other?

- Cox, Jakobi, Rosone, Schulz-Trieglaff: Comparing DNA sequence collections by direct comparison of compressed text indexes. WABI 2012.

Basic algorithm

```
function compare(A, B):
    S.push(ε, [0…|A| –1], [0…|B| –1])
    while S ≠ ∅:
        (X, [spA…epA], [spB…epB]) ← S.pop()
        if [spA…epA] = ∅ and [spB…epB] = ∅:
            continue
    if report_condition(X, [spA…epA], [spB…epB]):
        report(X, [spA…epA], [spB…epB])
    if expand_condition(X, [spA…epA], [spB…epB]):
        for c ∈ ∑:
            S.push(cX,
                A.LF([spA…epA], c),
                B.LF([spB…epB], c))
```

Population BWT

- We have a massive collection of reads in multiple FM-indexes distributed over several servers.
- If we want to query the population BWT with another sequence collection, we extract kmers from the query sequences and query the servers with them.
- The intermediate results can take terabytes.
- Ideally we would want to filter the results on the servers.

Another approach

- Build an FM-index for the **query sequences** and submit it to the servers.

- Write and submit **functions** report_condition(), report(), and expand_condition().

- Because the collection is in **multiple FM-indexes**, we also need function reduce() (as in MapReduce) to **merge** the results for the same substring.

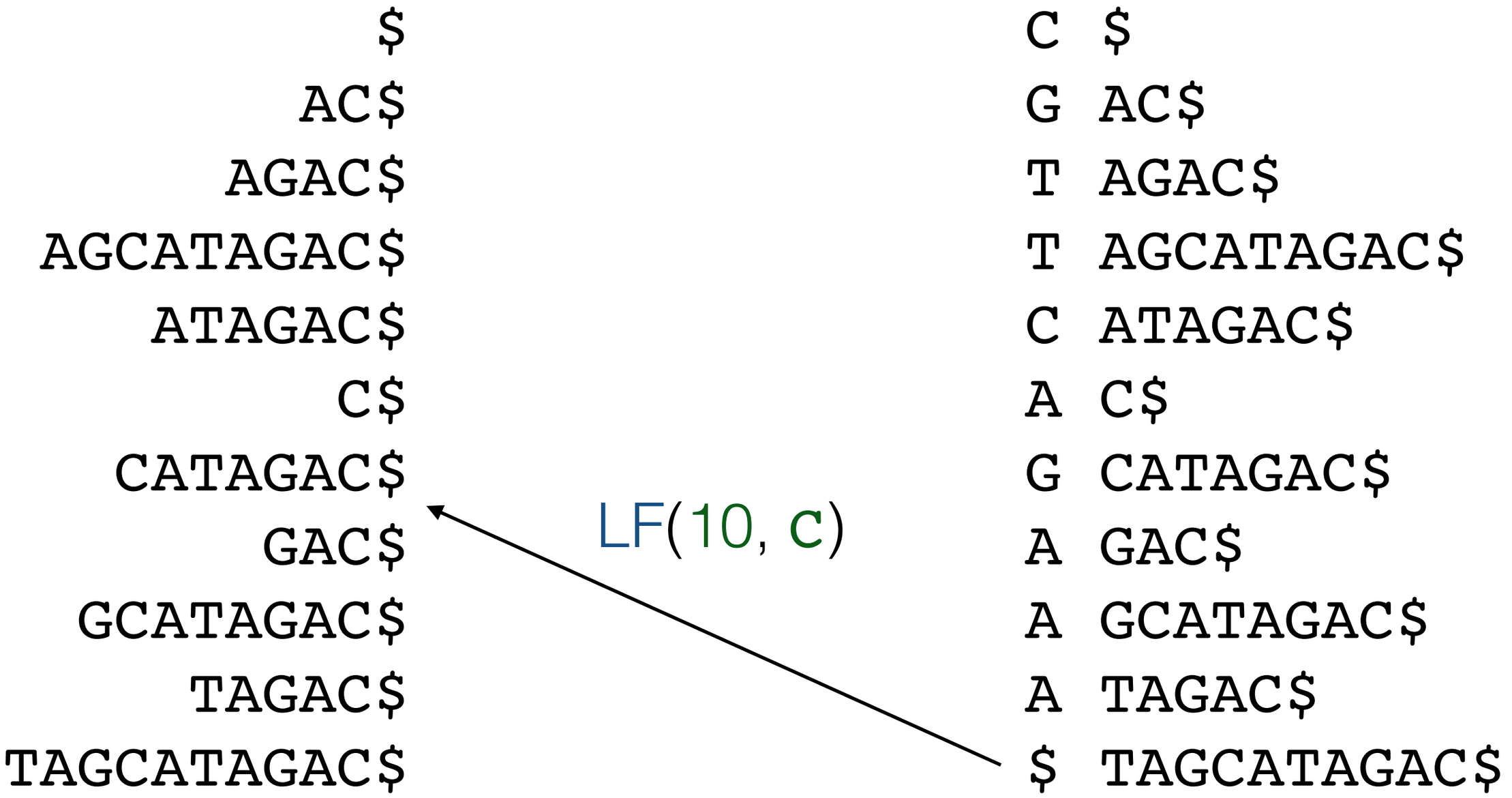
- What would be possible with this approach?

BWT Merging

Large-scale BWT construction

- Data structure construction is a major bottleneck.
- We must sort n suffixes quickly using less than n bits of memory.
- There is no such thing as large amounts of fast disk space in high-performance computing.
- Distributing the work over multiple nodes is possible, but everyone else wants to use the nodes as well.
- The construction algorithms should be incremental to avoid redundant work over time.

Hypothetical suffixes



Interpretation: LF(i, c) = C[c] + BWT.rank(i, c) suffixes are strictly before the hypothetical suffix.

TAGCATAGAC$ ⟶ **C**TAGCATAGAC$

Insert **C** to the beginning

$	C
AC$	G
AGAC$	T
AGCATAGAC$	T
ATAGAC$	C
C$	A
CATAGAC$	G
GAC$	A
GCATAGAC$	A
TAGAC$	A
TAGCATAGAC$	$

$	C
AC$	G
AGAC$	T
AGCATAGAC$	T
ATAGAC$	C
C$	A
CATAGAC$	G
CTAGCATAGAC$	**$**
GAC$	A
GCATAGAC$	A
TAGAC$	A
TAGCATAGAC$	**C**

2. Insert **$** after LF(i, **C**) suffixes.

1. Replace the **$** at position i with the inserted **C**.

$	**C**
AC$	**G**
AGAC$	**T**
AGCATAGAC$	**T**
ATAGAC$	**C**
C$	**A**
CATAGAC$	**G**
CTAGCATAGAC$	**$**
GAC$	**A**
GCATAGAC$	**A**
TAGAC$	**A**
TAGCATAGAC$	**C**

$	C
AC$	G
AGCATCGAC$	T
ATCGAC$	C
C$	A
CATCGAC$	G
CGAC$	T
CTAGCATCGAC$	$
GAC$	C
GCATCGAC$	A
TAGCATCGAC$	C
TCGAC$	A

Merge the BWT of **TAGCATAGAC$** with the BWT of CTAGCATCGAC$, assuming that $ < **$**.

Rank array RA tells how many black suffixes are before each red suffix in lexicographic order.

We start with RA[0] = 1. Once we know RA[i], we can set RA[LF_{red}(i)] to LF_{black}(RA[i], BWT_{red}[i])

Because the rank array is sorted, we can output it in any order.

RA		
	$	C
1	**$**	**C**
	AC$	G
2	**AC$**	**G**
2	**AGAC$**	**T**
2	**AGCATAGAC$**	**T**
	AGCATCGAC$	T
3	**ATAGAC$**	**C**
	ATCGAC$	C
	C$	A
5	**C$**	**A**
5	**CATAGAC$**	**G**
	CATCGAC$	G
	CGAC$	T
7	**CTAGCATAGAC$**	**$**
	CTAGCATCGAC$	$
	GAC$	C
9	**GAC$**	**A**
9	**GCATAGAC$**	**A**
	GCATCGAC$	A
10	**TAGAC$**	**A**
10	**TAGCATAGAC$**	**C**
	TAGCATCGAC$	C
	TCGAC$	A

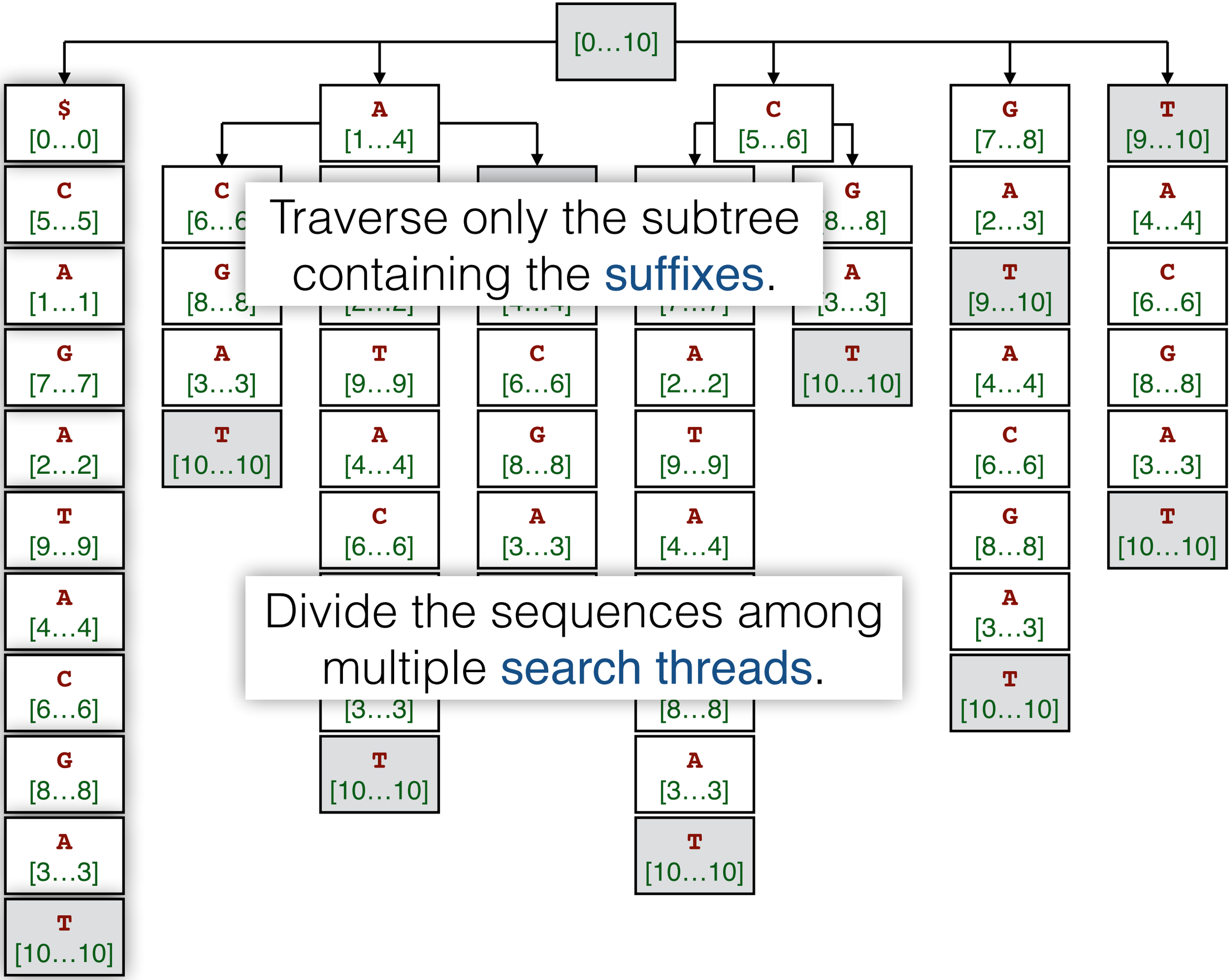
BWT merging

Jouni Sirén: Burrows-Wheeler transform for terabases. DCC 2016. https://github.com/jltsiren/bwt-merge (Also: Hon et al, 2007; Sirén, 2009)

Search: Generate the ranks in any order by e.g. traversing the subtrees corresponding to the suffixes.

Sort: Sort the ranks to build the rank array. This can be done in parallel with the other phases.

Merge: Interleave the source BWTs according to the rank array. This can be done almost in-place with two-level arrays.


[0…10]
$ [0…0]
A [1…4]
C [5…6]
G [7…8]
T [9…10]
C [5…5]
A [1…1]
G [7…7]
A [2…2]
T [9…9]
A [4…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
A [3…3]
T [10…10]
C [6…6]
A [4…4]
G [8…8]
A [3…3]
T [10…10]
A [2…2]
T [9…9]
A [4…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
A [4…4]
C [6…6]
G [8…8]
A [3…3]
T [10…10]
Traverse only the subtree containing the suffixes.
Divide the sequences among multiple search threads.

Search algorithm

```
function search(A, B, n_A, n_B):
    S.push(n_A, [0…n_B –1])
    while S ≠ ∅:
        (r, [sp…ep]) ← S.pop()
        if [sp…ep] = ∅:
            continue
        report(r, ep + 1 – sp)
        for c ∈ ∑:
            S.push(A.LF(r, c), B.LF([sp…ep], c))
```

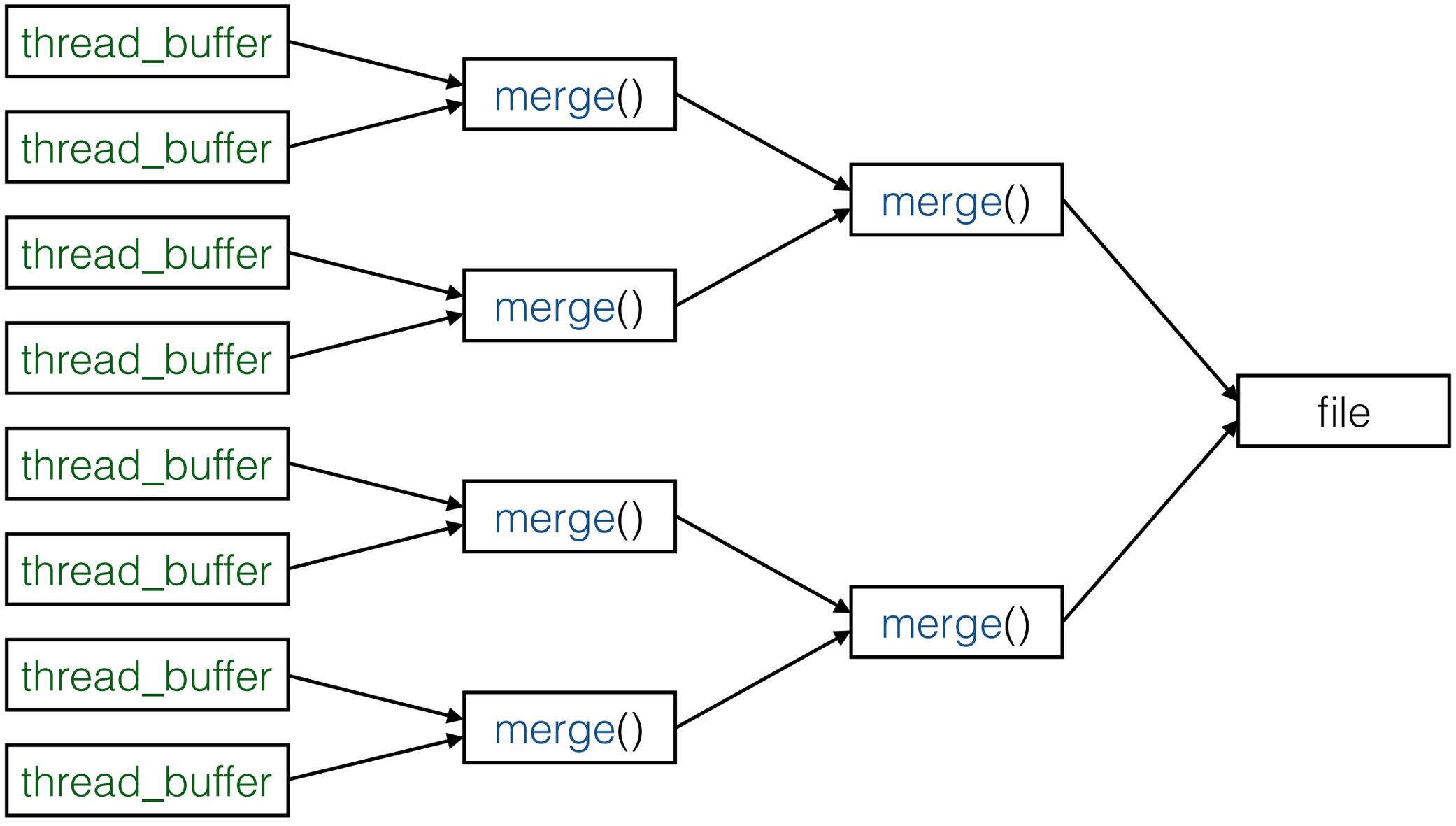
Insert the n_B sequences from FM-index B into index A containing n_A sequences.

Sorting 1/3

```
function report(rank, count):
    run_buffer.insert(rank, count)
    if run_buffer.full():
        sort(run_buffer)
        compress(run_buffer)
        thread_buffer ← merge(run_buffer, thread_buffer)
        if thread_buffer.full():
            merge(thread_buffer, merge_buffers)
```

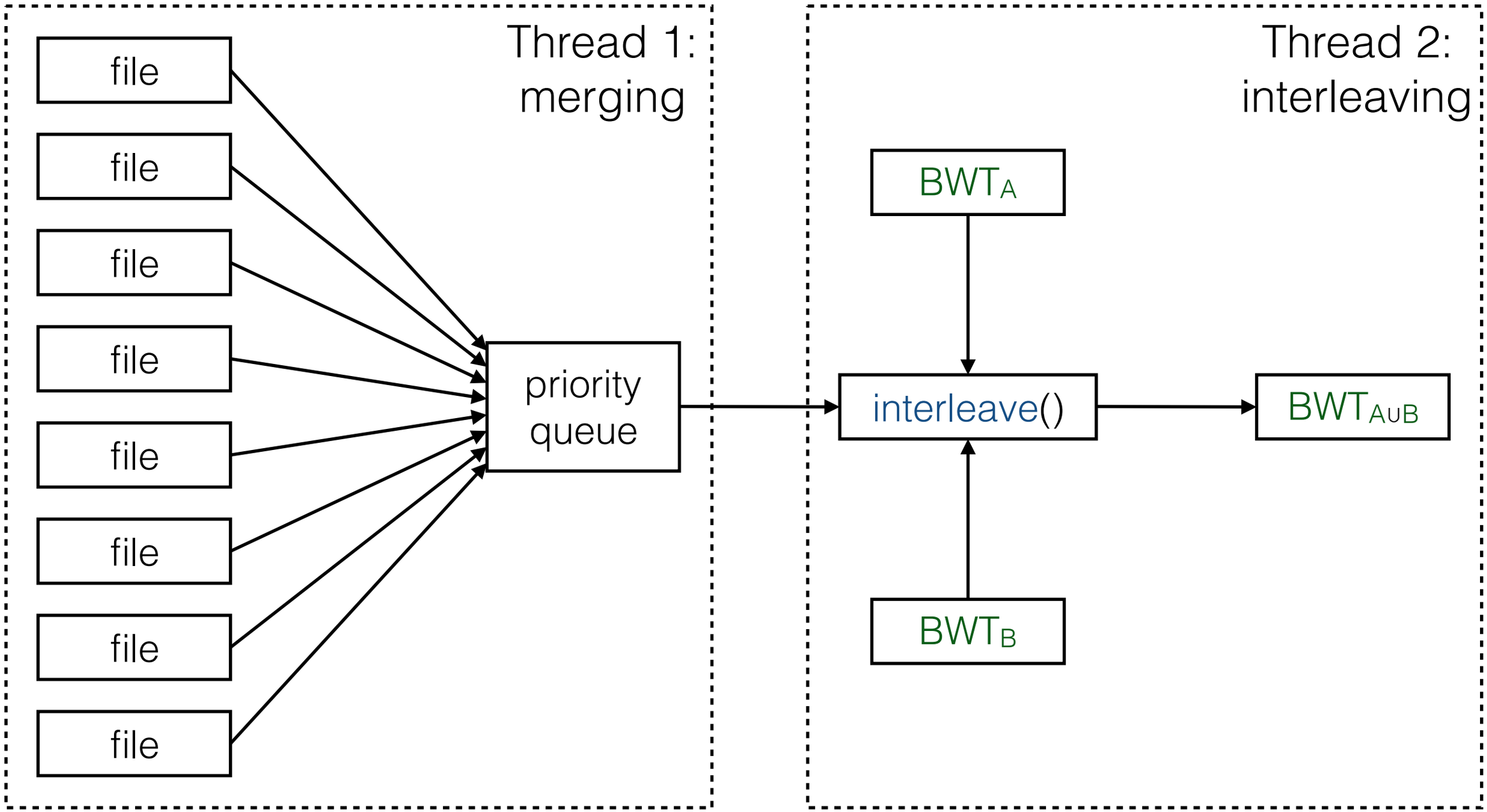
The buffers are **thread-specific**.

Sorting 2/3



Each level below the root has one **merge buffer**.

Sorting 3/3 + merging



Multithreaded merging would help with a faster disk.

Benchmark: Population BWT

- We use 32 threads, 128 MB run buffers, 256 MB thread buffers, and 6 levels of merge buffers.
- Input: 16 FM-indexes partitioned by last two bases; 53.0G distinct sequences, 4.88 Tbp, 561.5 GB.
- Output: 2 FM-indexes partitioned by the last base.
- Switch from 3+5-bit run-length encoding to another byte-level code that can handle long runs.

Merging: *A, *C / *G, *T

	Input		Merged
SGA format	281 GB 281 GB	81.3 hours 221 GB memory 297 GB disk	239 GB 239 GB
BWT-merge format	225 GB 226 GB	83.0 hours 219 GB memory 300 GB disk	181 GB 180 GB

7 Mbp/s, <n bits of memory, 2n bits of working space.

BWT construction

- Use RopeBWT to build indexes for subcollections. (In-memory implementation of Bauer et al: Lightweight algorithms for constructing and inverting the BWT of string collections. TCS, 2013)

- Slightly faster than merging, and we often have enough memory to run 2 or 3 processes in parallel.

- We can build BWT for terabytes of short reads at ~5 MB/s in <n bits of memory and 2n bits of working space.

Conclusions

- FM-index works with lexicographic ranges of suffixes but encodes the reverse prefix trie of the text.

- Many algorithms can be understood as traversals of the trie.

- We can compare sequence collections and merge FM-indexes by traversing the reverse prefix tries.

- These algorithms are fast and space-efficient and require very little additional code or data structures.