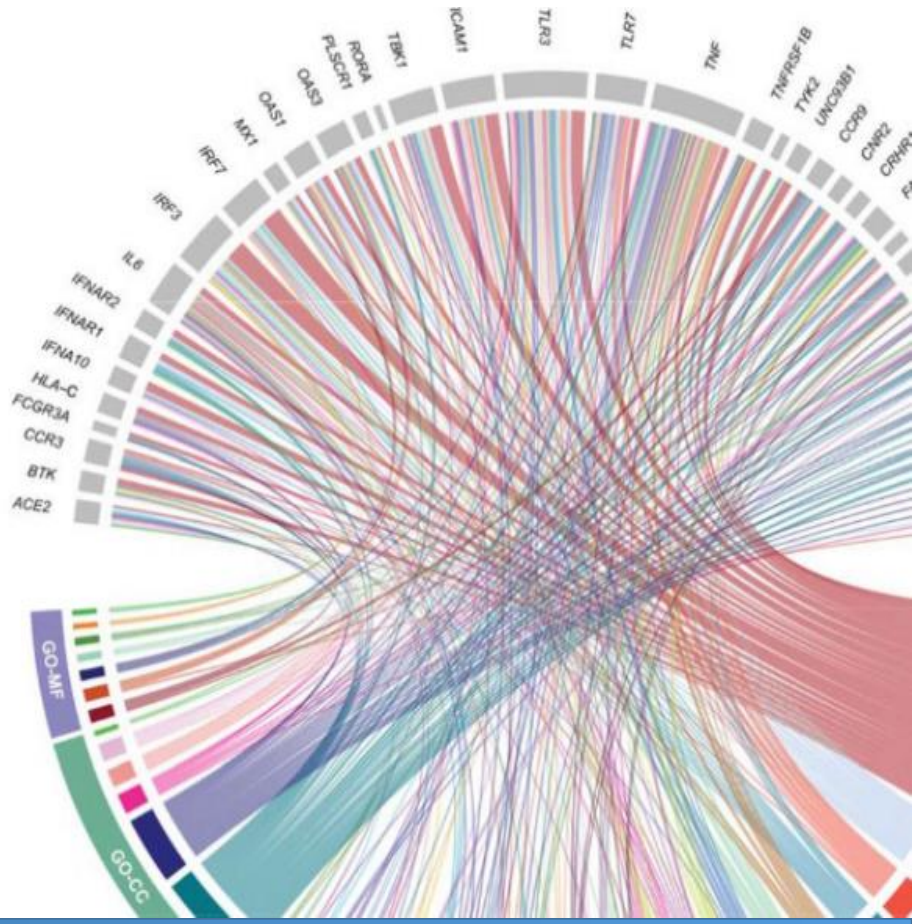
# Tècniques i Eines Bioinformàtiques

## Kmer Indexes




***Santiago Marco-Sola (santiago.marco@upc.edu)***

*Màster en Enginyeria Informàtica, UPC*
*Departament of Computer Science*
*Facultat d'Informàtica de Barcelona (FIB), UPC*

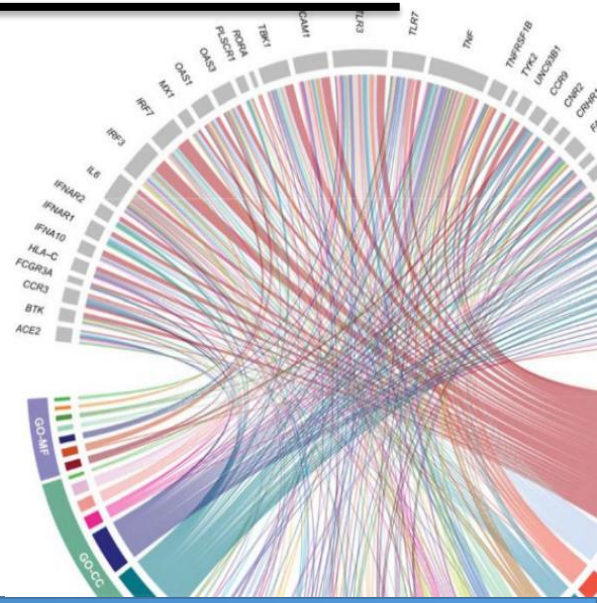
# Acknowledgements

Many pictures and materials are taken from **Ben Langmead's course**.

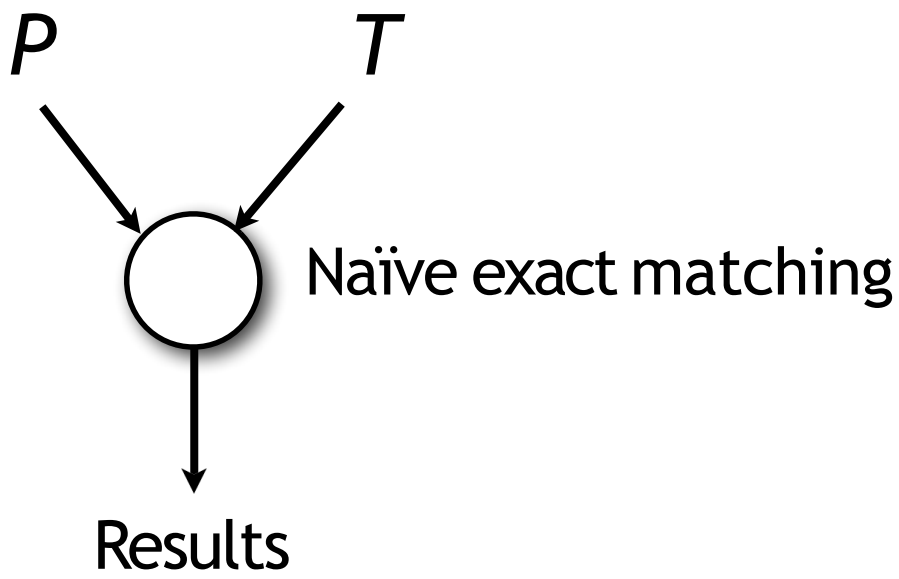
Course heavily inspired in:

- **Genome-Scale Algorithm Design**. Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, Alexandru I. Tomescu. Cambridge University Press.
- **Algorithms on Strings, Trees, and Sequences**. Dan Gusfield. Cambridge University Press.
- **An Introduction to Bioinformatics Algorithms.** Neil C. Jones, Pavel A. Pevzner. MIT Press.

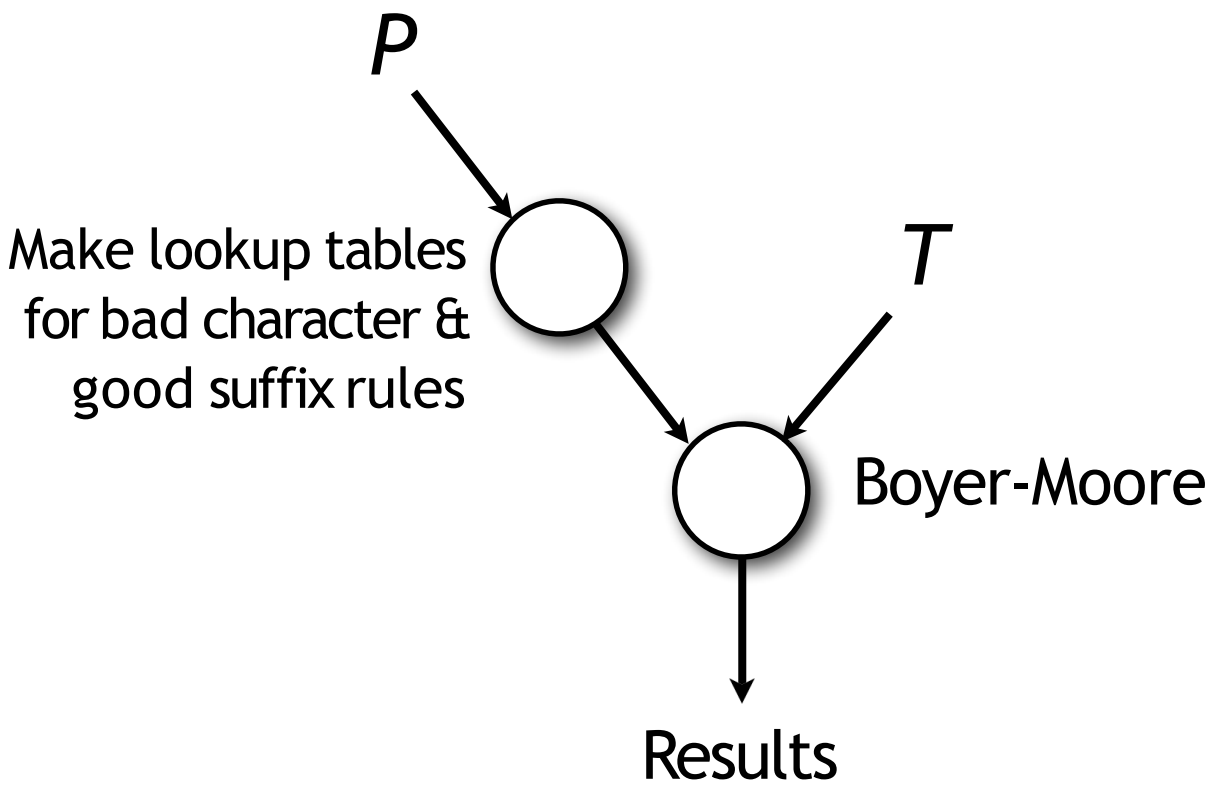
# 1 Indexes for Genomics



# Preprocessing: Naïve algorithm



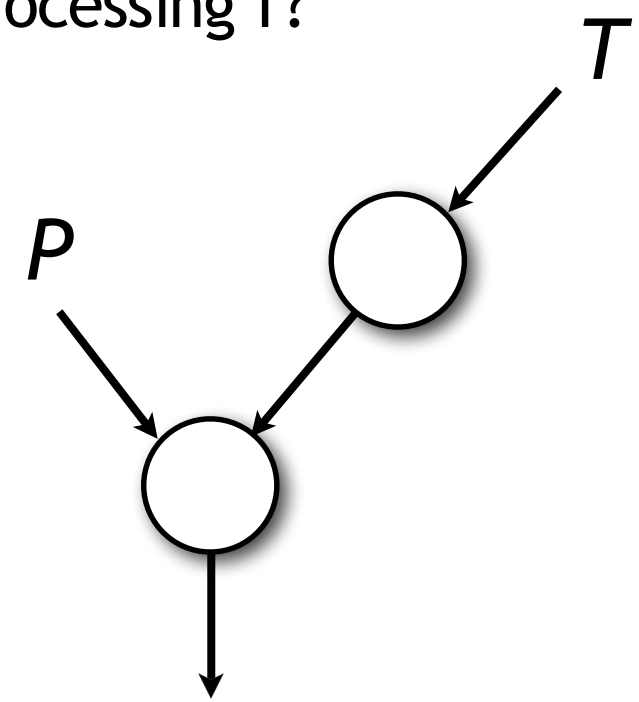
# Preprocessing: Boyer-Moore



# Preprocessing

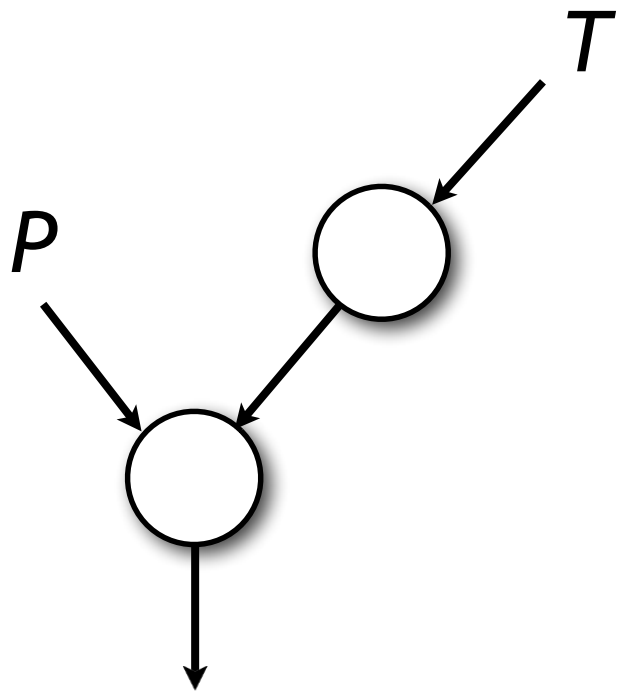
Boyer-Moore preprocessed P

What about preprocessing T?



# Preprocessing

Algorithm that preprocesses *T* is *offline*.
Otherwise, algorithm is *online*.

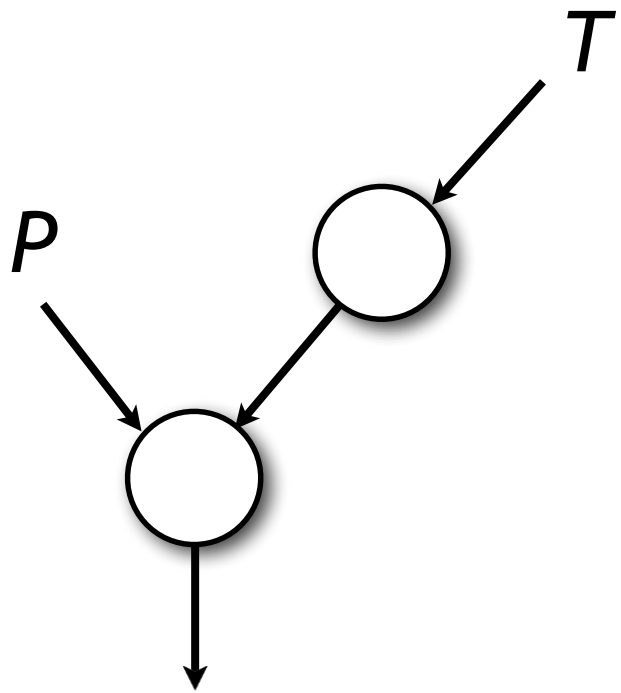



Online or offline?

- Naïve algorithm
- Boyer-Moore
- Web search engine
- Read alignment

# Preprocessing

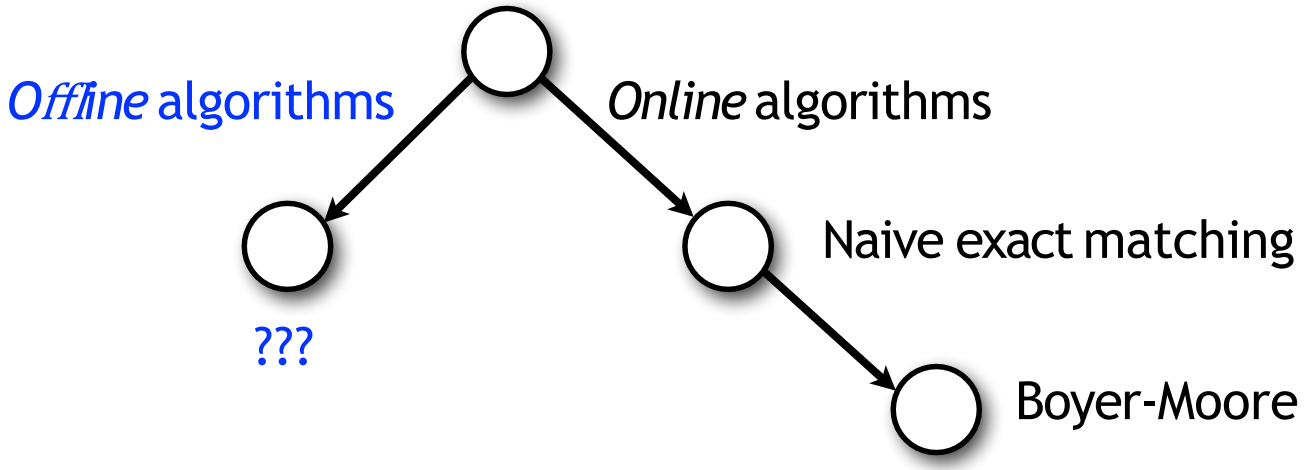
Algorithm that preprocesses *T* is *offline*.
Otherwise, algorithm is *online*.




Online or offline?

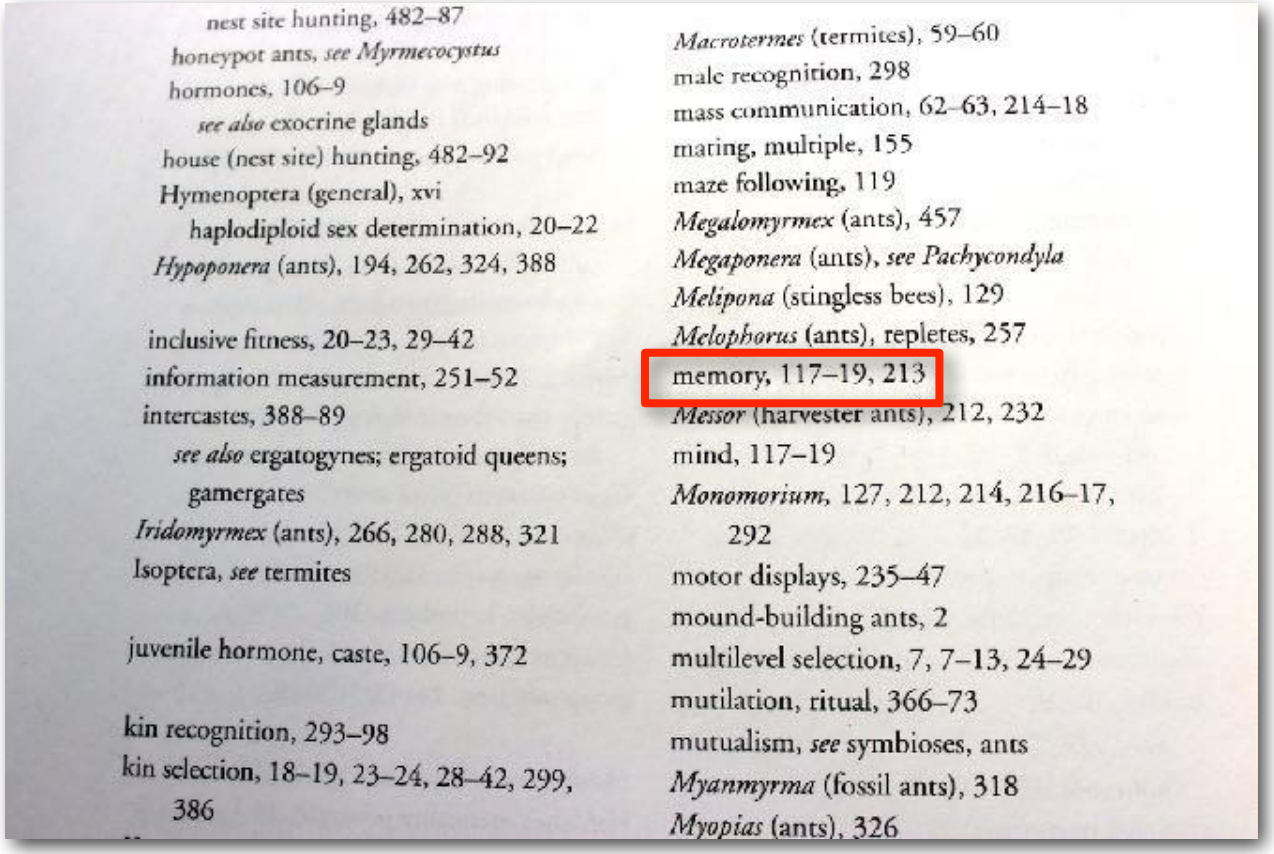
- Naïve algorithm
- Boyer-Moore
- Web search engine
- Read alignment

# Offline algorithms




Still focusing on exact matching problem: find all places where pattern *P* exactly matches a substring of text *T*

# Index





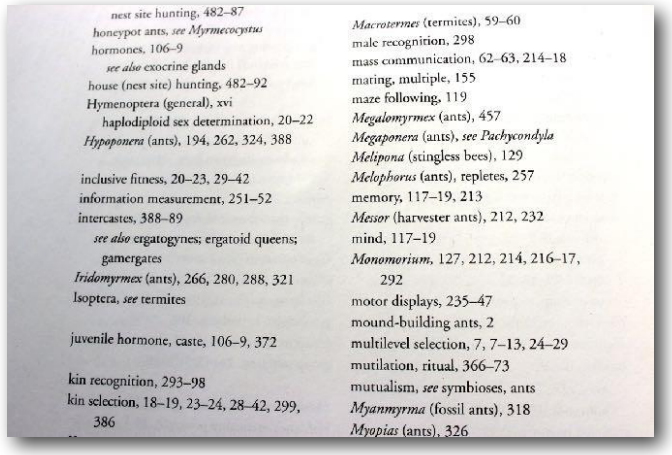
Key terms ordered alphabetically, with associated page #s

# Index

Grocery store items grouped into aisles

# Index

Indexes use *ordering* and *grouping* to make it easy to jump to relevant portions of the data



nest site hunting, 482–87
honeypot ants, *see Myrmecocystus*
hormones, 106–9
*see also* exocrine glands
house (nest site) hunting, 482–92
Hymenoptera (general), xvi
haplodiploid sex determination, 20–22
*Hypoponera* (ants), 194, 262, 324, 388

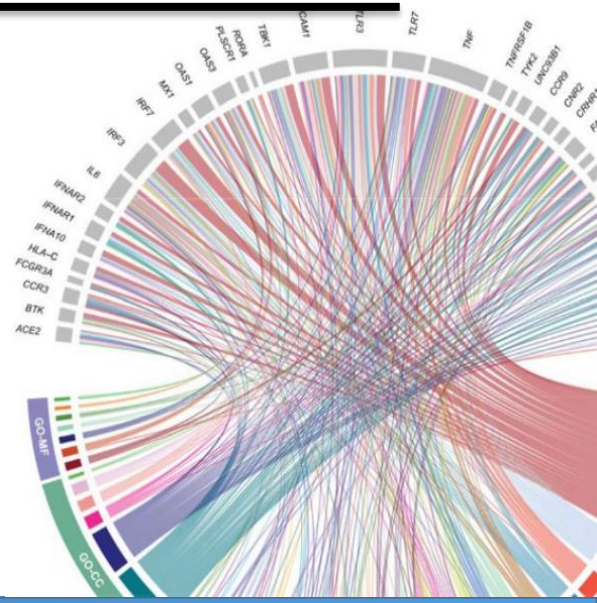
inclusive fitness, 20–23, 29–42
information measurement, 251–52
intercastes, 388–89
*see also* ergatogynes; ergatoid queens; gamergates
*Iridomyrmex* (ants), 266, 280, 288, 321
Isoptera, *see* termites

juvenile hormone, caste, 106–9, 372

kin recognition, 293–98
kin selection, 18–19, 23–24, 28–42, 299, 386

*Macrotermes* (termites), 59–60
male recognition, 298
mass communication, 62–63, 214–18
mating, multiple, 155
maze following, 119
*Megalomyrmex* (ants), 457
*Megaponera* (ants), *see Pachycondyla*
*Melipona* (stingless bees), 129
*Melophorus* (ants), repletes, 257
memory, 117–19, 213
*Messor* (harvester ants), 212, 232
mind, 117–19
*Monomorium*, 127, 212, 214, 216–17, 292
motor displays, 235–47
mound-building ants, 2
multilevel selection, 7, 7–13, 24–29
mutilation, ritual, 366–73
mutualism, *see* symbioses, ants
*Myanmyrma* (fossil ants), 318
*Myopias* (ants), 326

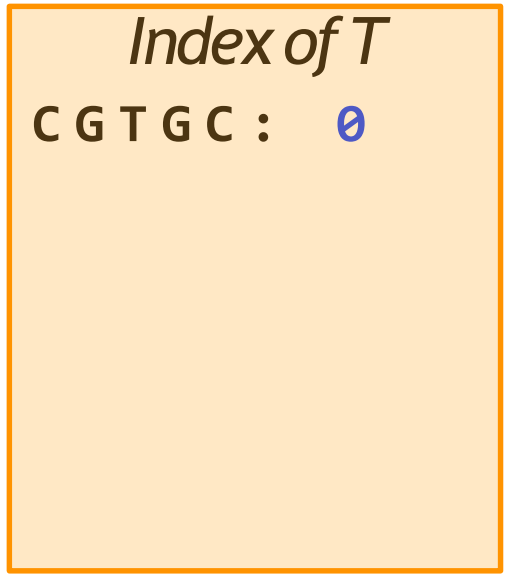
# 2 Multimap Index



# Indexing DNA

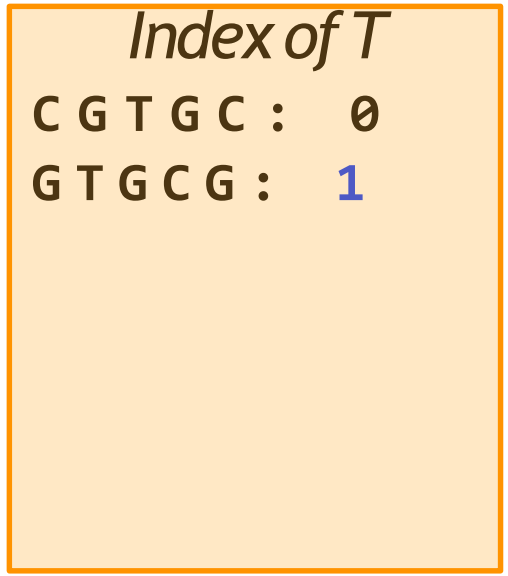

*T*: C G T G C G T G C T T

# Indexing DNA




*T:* C G T G C G T G C T T

# Indexing DNA




*T*: C G T G C G T G C T T

# Indexing DNA

*Index of T*

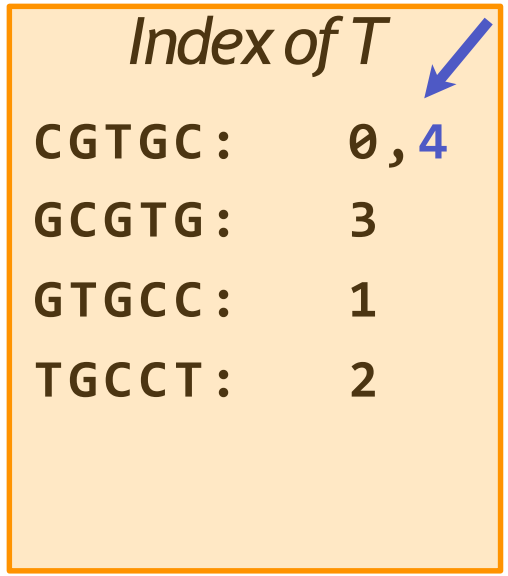
```
CGTGC:  0
GTGCG:  1
TGCGT:  2
```

*T*: C G T G C G T G C T T

# Indexing DNA


*T*: C G T G C G T G C T T

# Indexing DNA




*T*: C G T G C G T G C T T

# Indexing DNA

*Index of T*

| | |
|---|---|
| CGTGC: | 0,4 |
| GCGTG: | 3 |
| GTGCC: | 1 |
| GTGCT: | 5 |
| TGCCT: | 2 |

*T*: C G T G C G T G C T T

# Indexing DNA

*Index of T*

| | |
|---|---|
| CGTGC: | 0,4 |
| GCGTG: | 3 |
| GTGCC: | 1 |
| GTGCT: | 5 |
| TGCCT: | 2 |
| TGCTT: | 6 |

*T*: C G T G C G T G C T T

# Indexing DNA

*k-mer*: substring of length k

*Index of T*

| | |
|---|---|
| CGTGC: | 0,4 |
| GCGTG: | 3 |
| GTGCC: | 1 |
| GTGCT: | 5 |
| TGCCT: | 2 |
| TGCTT: | 6 |

5-mer index

*T*: C G T G C G T G C T T

# Querying the index

*Index of T*

| | |
|---|---|
| CGTGC: | 0,4 |
| GCGTG: | 3 |
| GTGCC: | 1 |
| GTGCT: | 5 |
| TGCCT: | 2 |
| TGCTT: | 6 |

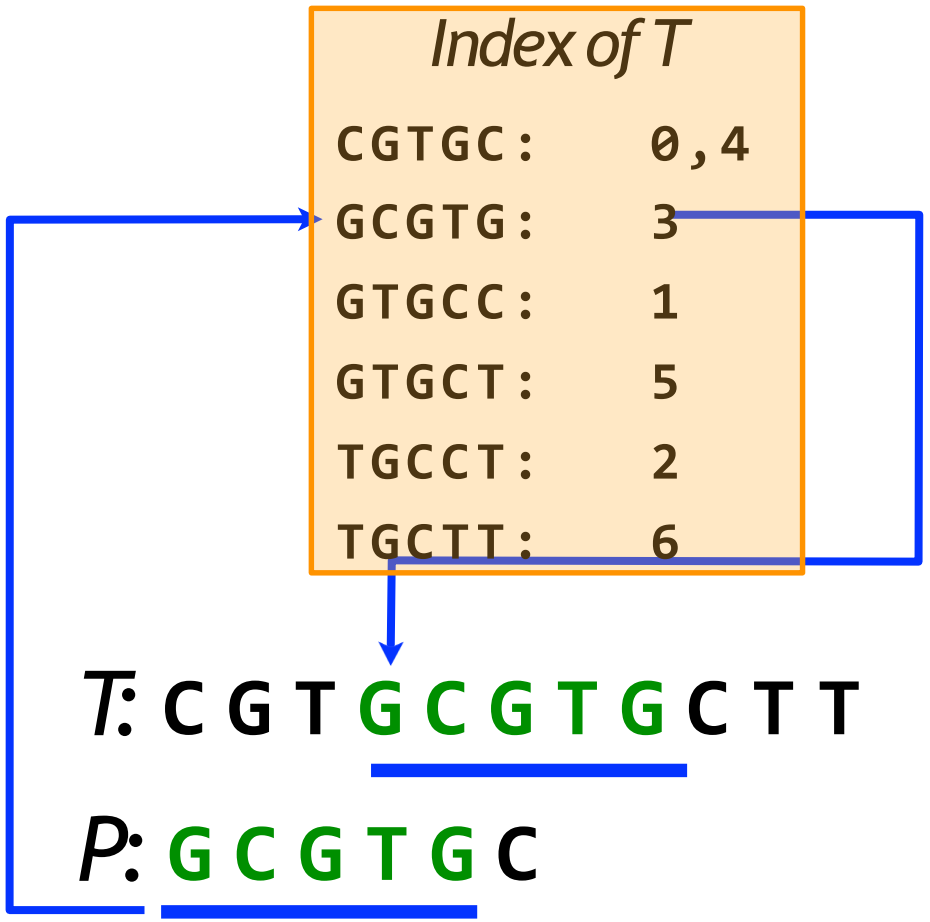
*T*: C G T G C G T G C T T

*P*: G C G T G C

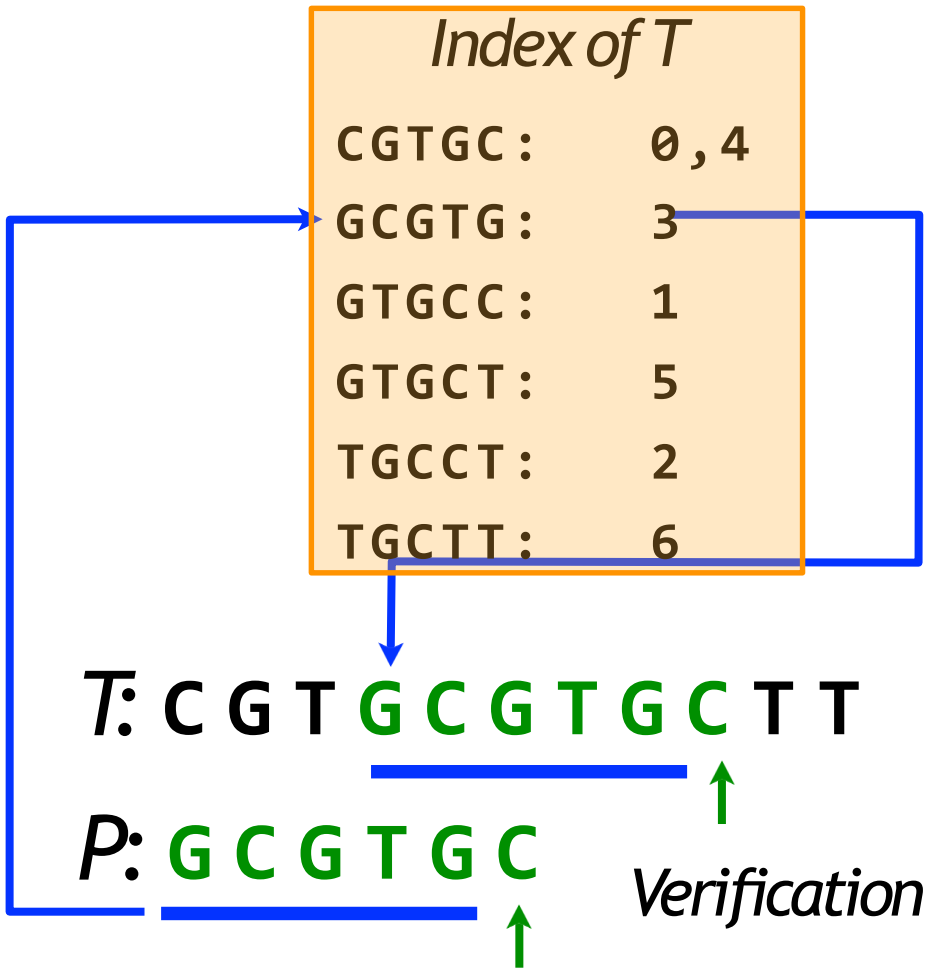
# Querying the index

# Querying the index

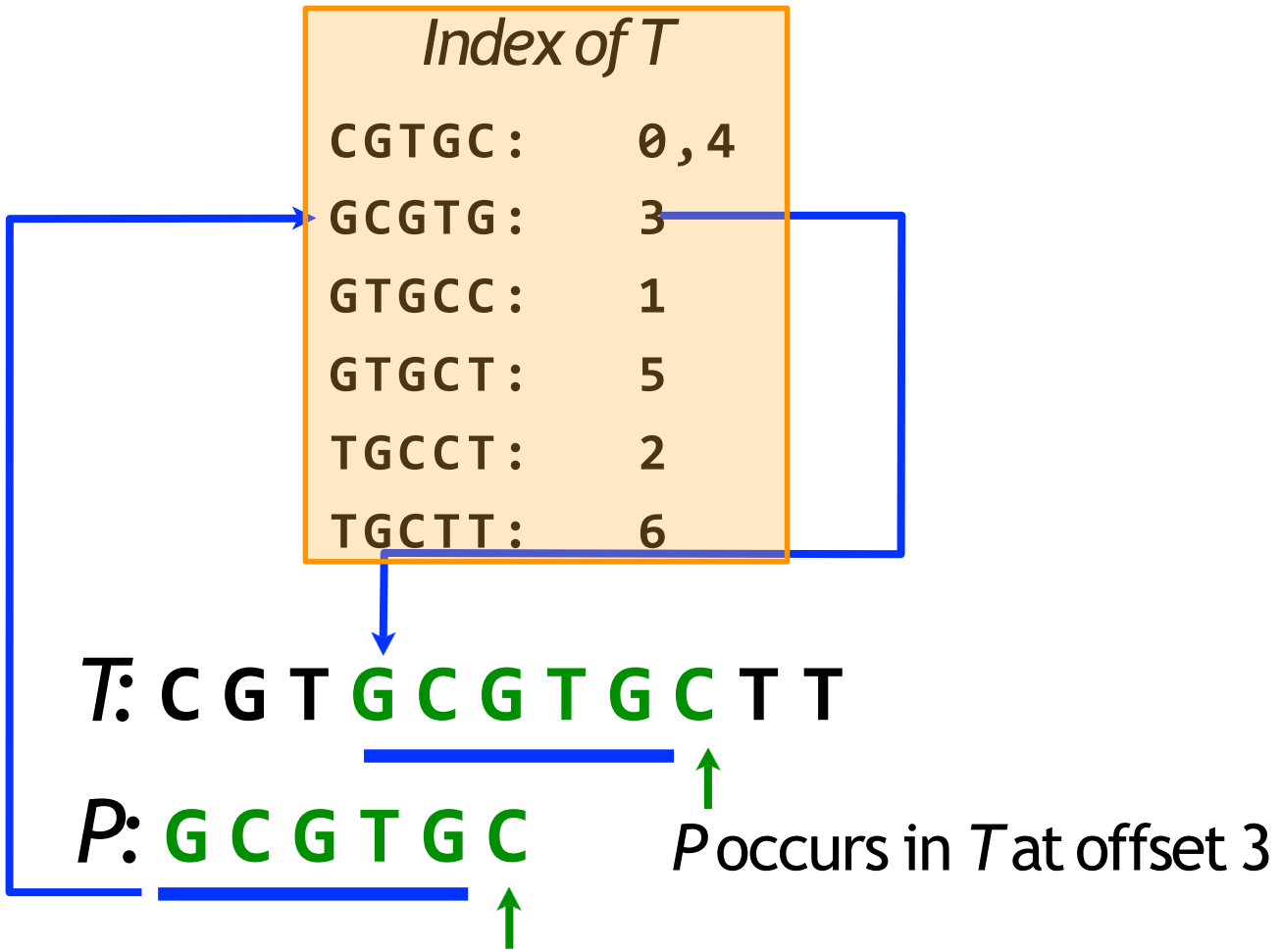
# Querying the index



# Querying the index

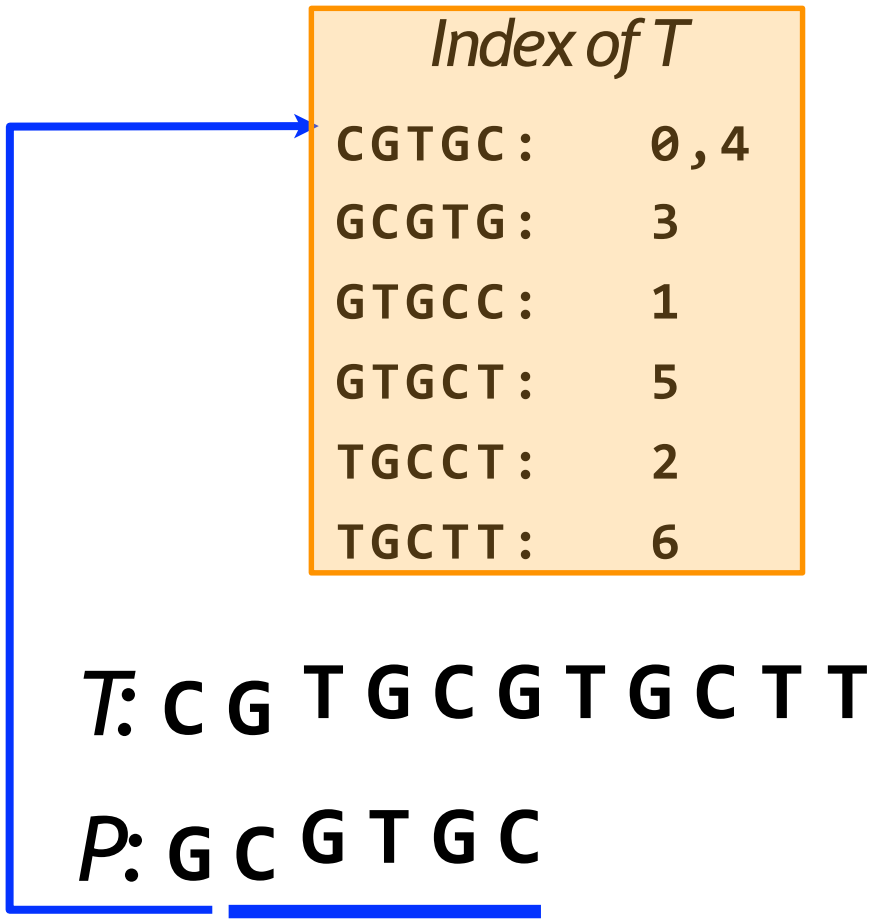


# Querying the index

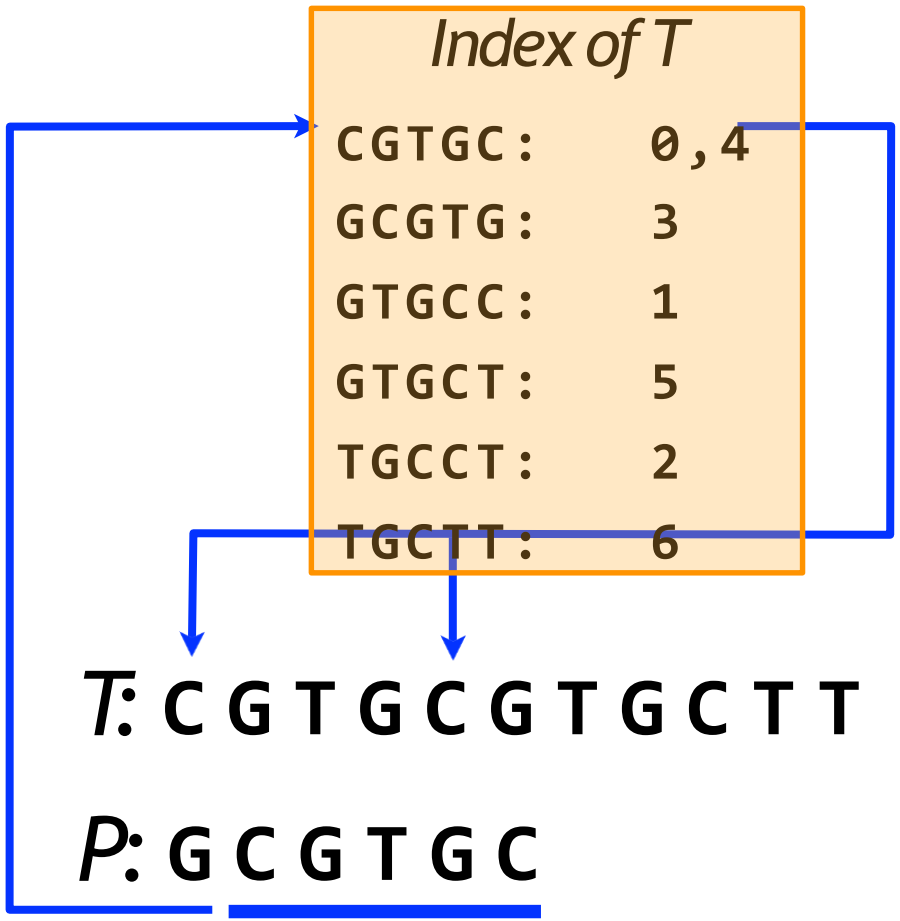


# Querying the index

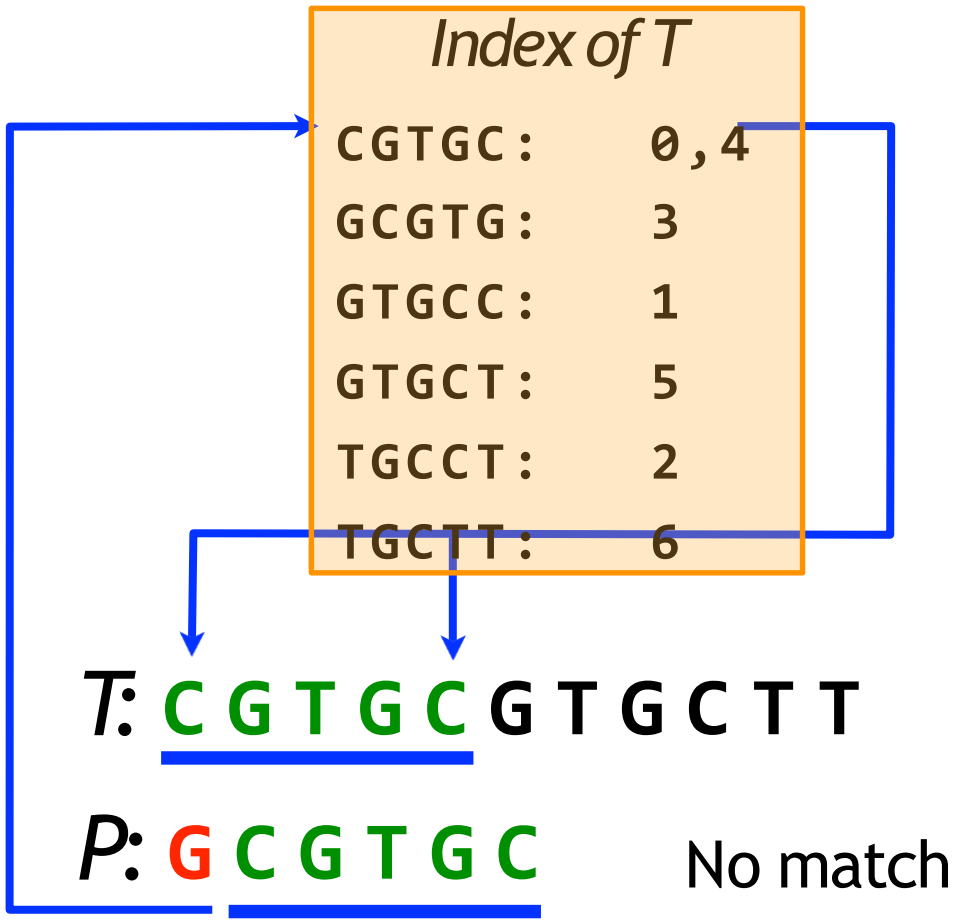
# Querying the index



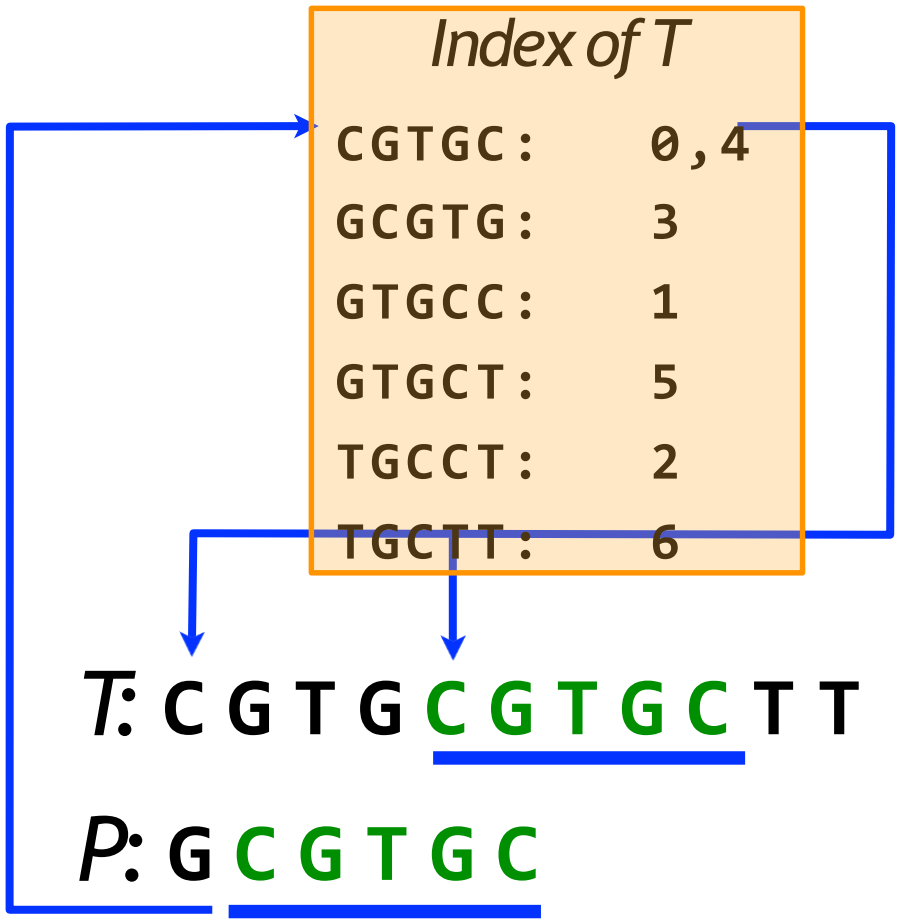
# Querying the index



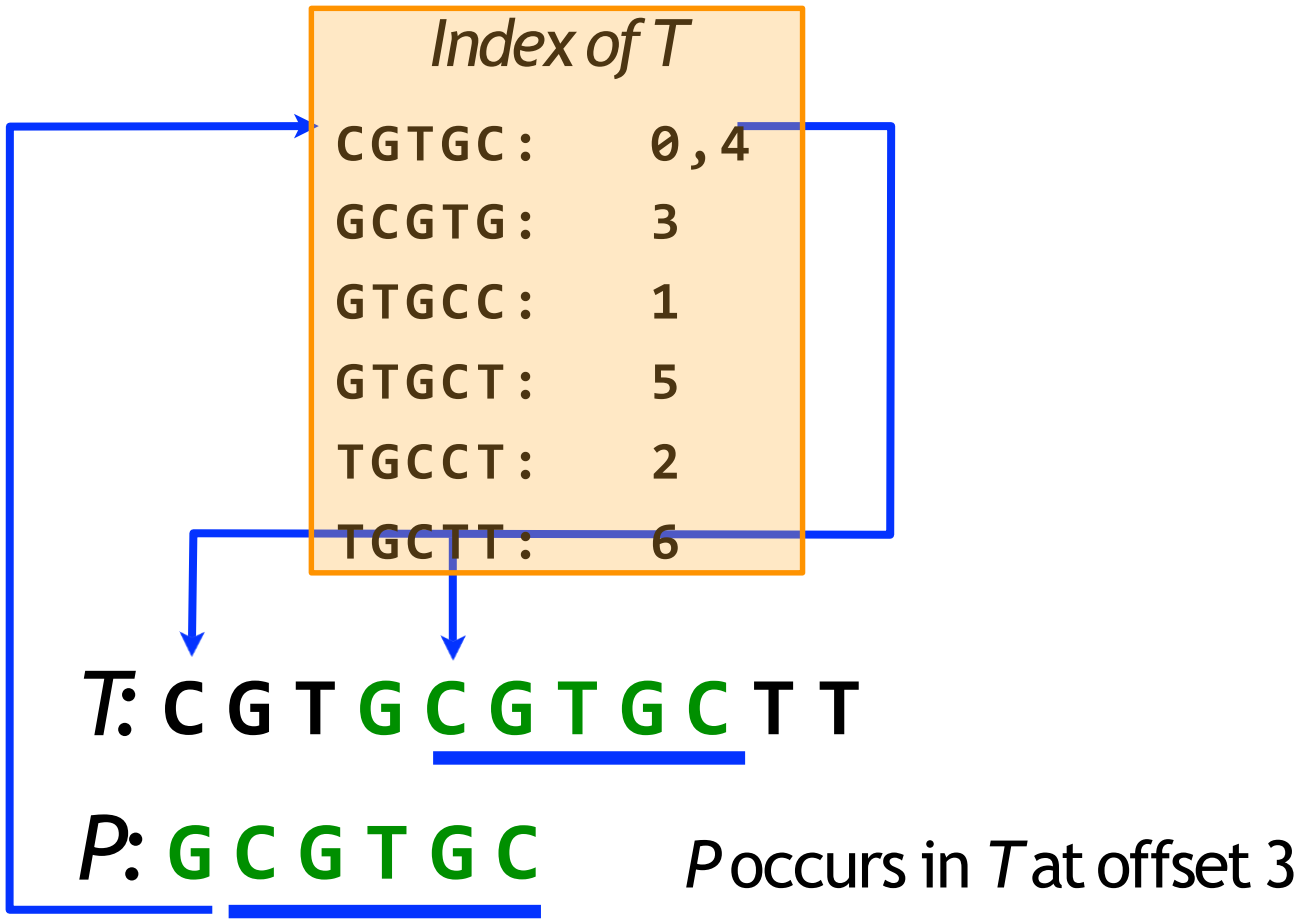
# Querying the index



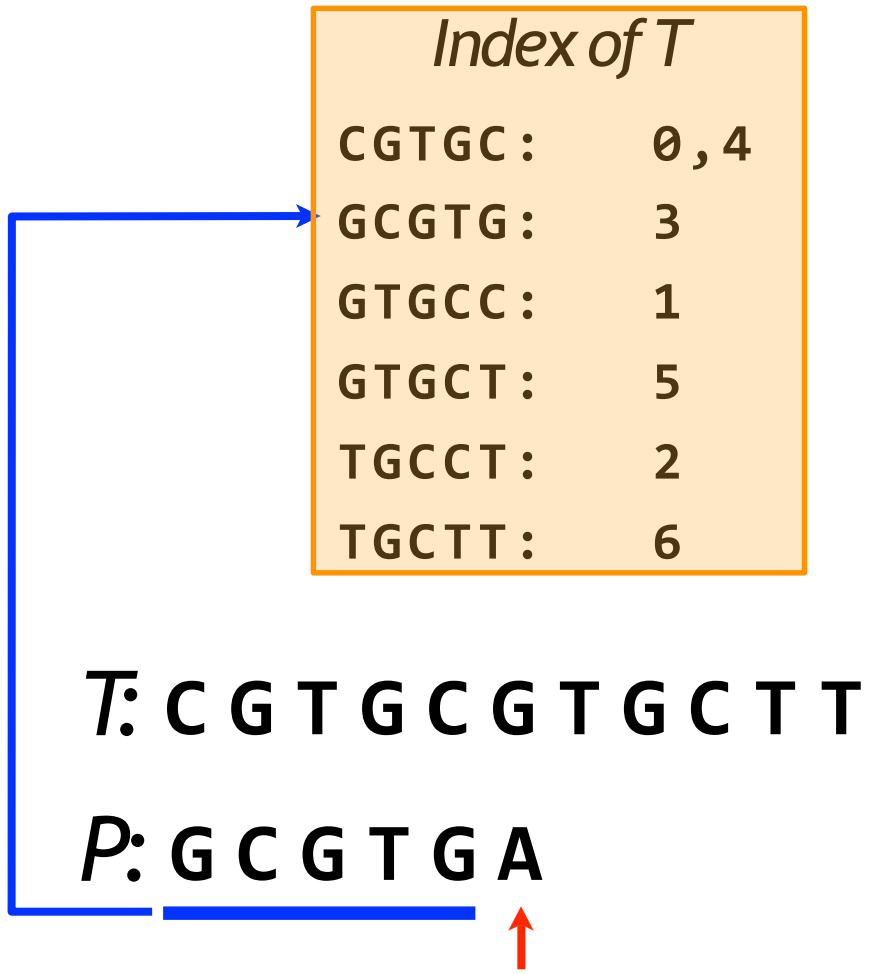
# Querying the index



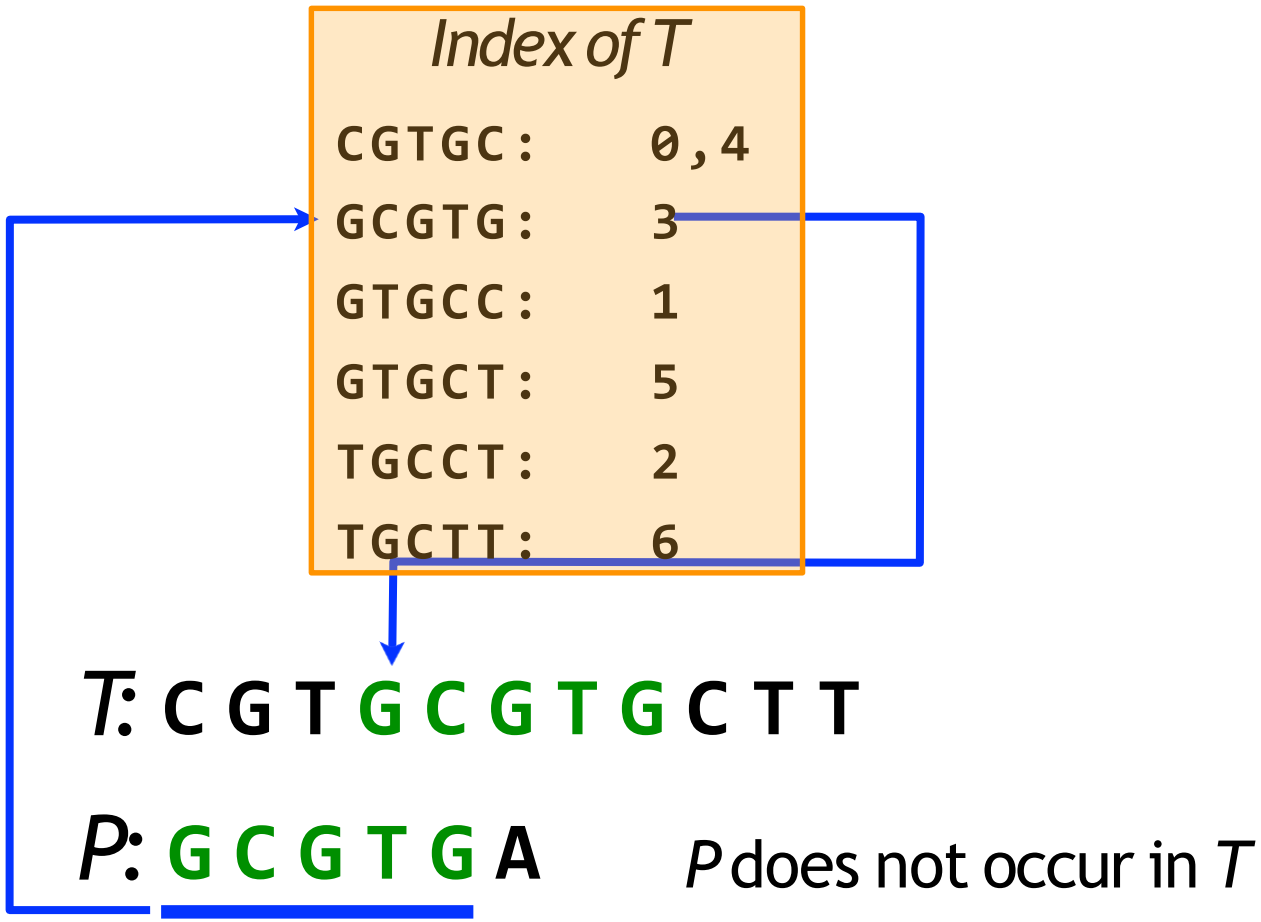
# Querying the index



# Querying the index



# Querying the index



# Querying the index

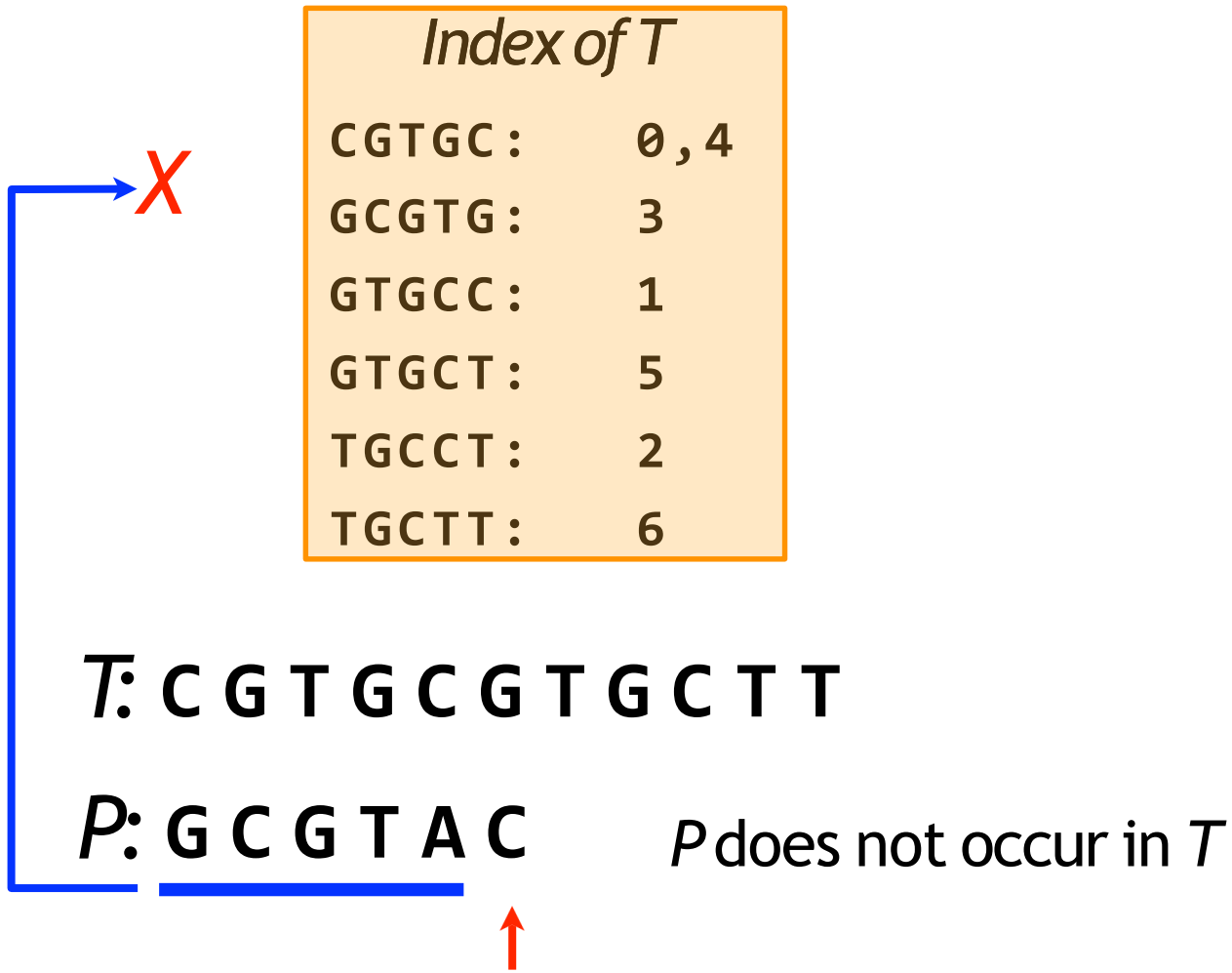
*Index of T*

| | |
|---|---|
| CGTGC: | 0,4 |
| GCGTG: | 3 |
| GTGCC: | 1 |
| GTGCT: | 5 |
| TGCCT: | 2 |
| TGCTT: | 6 |

*T*: C G T G C G T G C T T

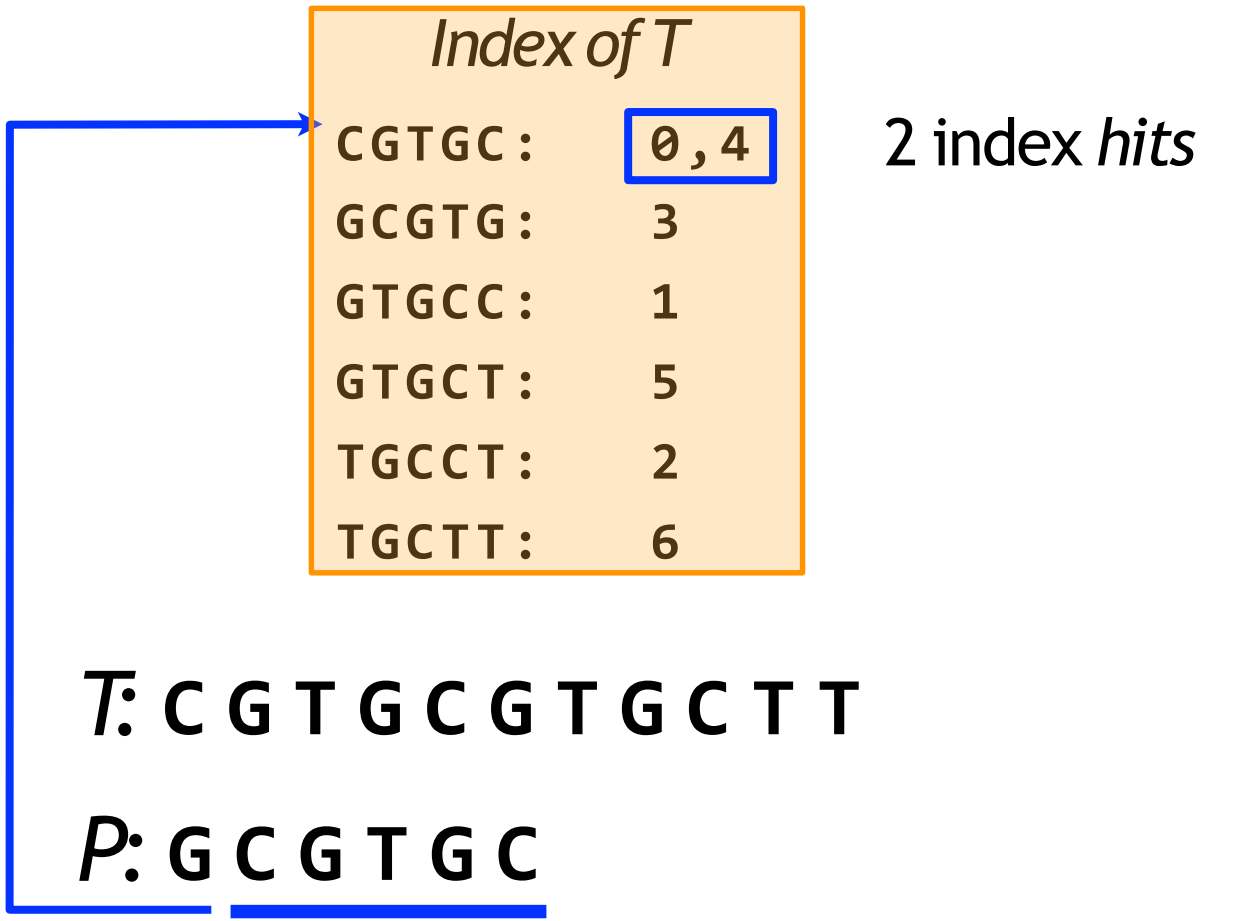
*P*: G C G T A C

# Querying the index



# Querying the index

# Querying the index



# Data structures

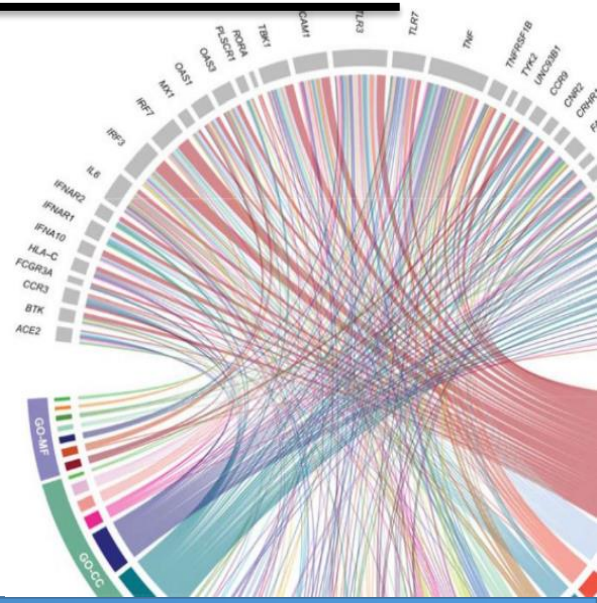
*Index of T*

```
CGTGC:   0,4
GCGTG:   3
GTGCC:   1
GTGCT:   5
TGCCT:   2
TGCTT:   6
```

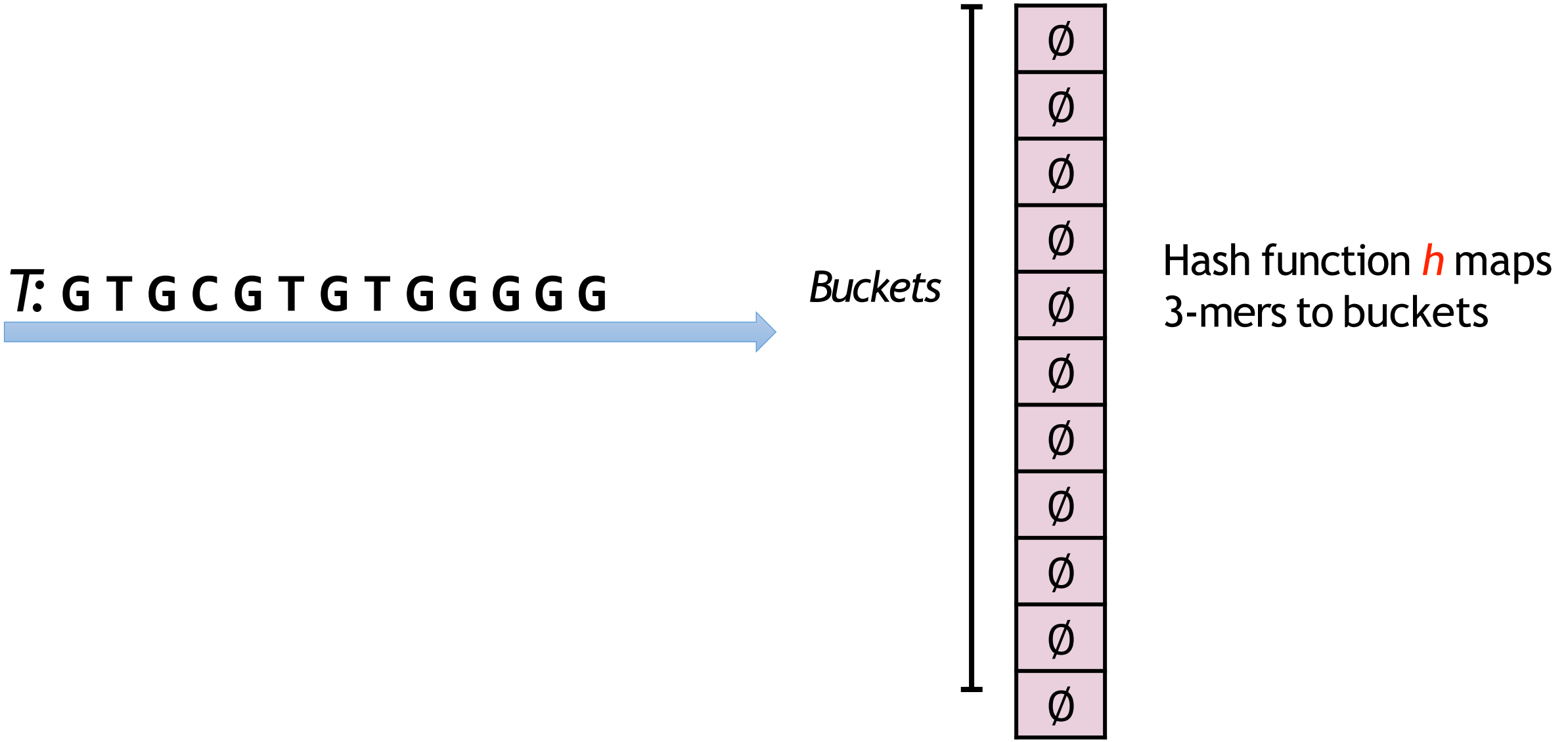
Abstractly, index is a *multimap* associating keys (k-mers) with one or more values (offsets)

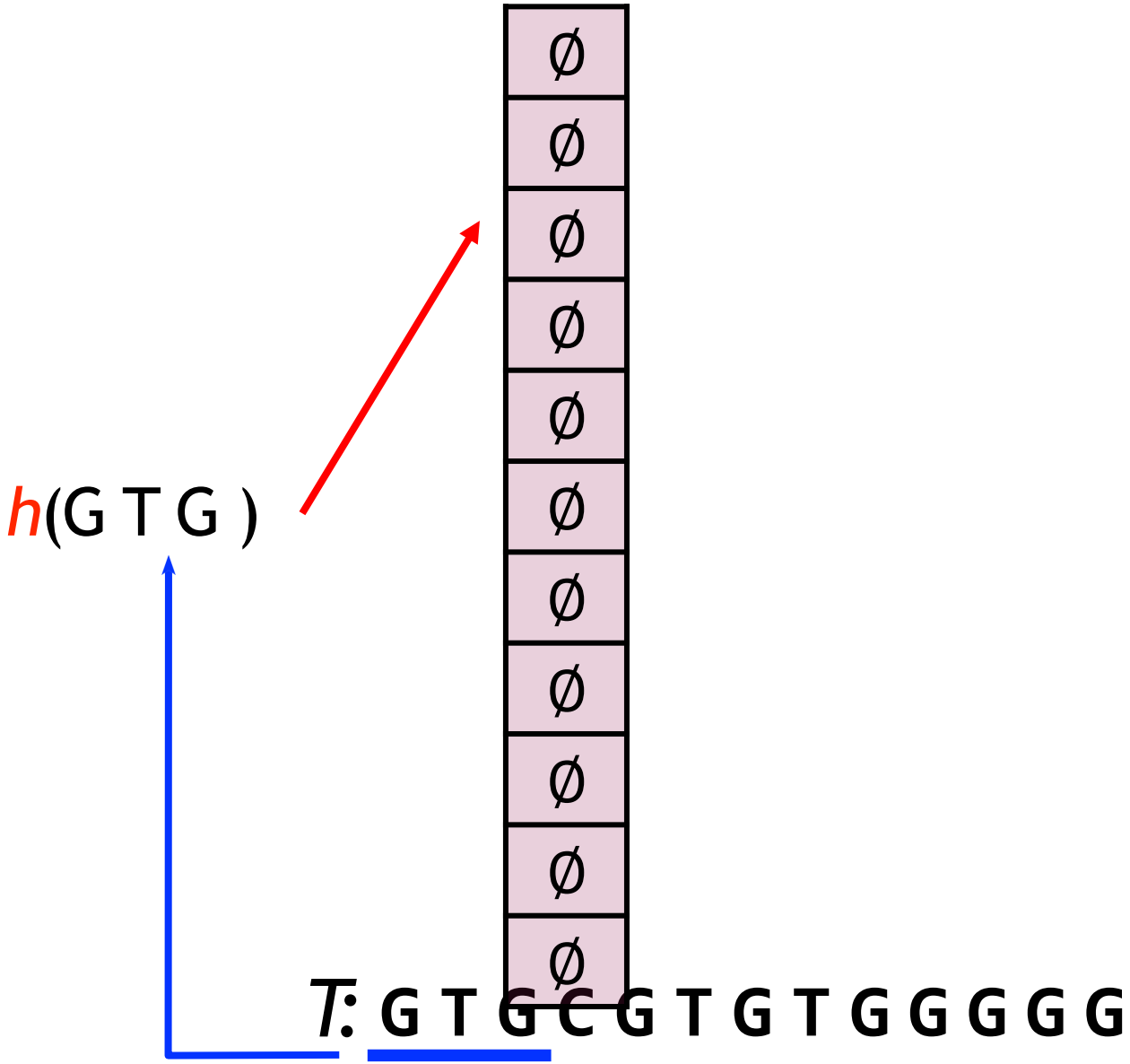
What data structures allow us to represent and query a multimap?

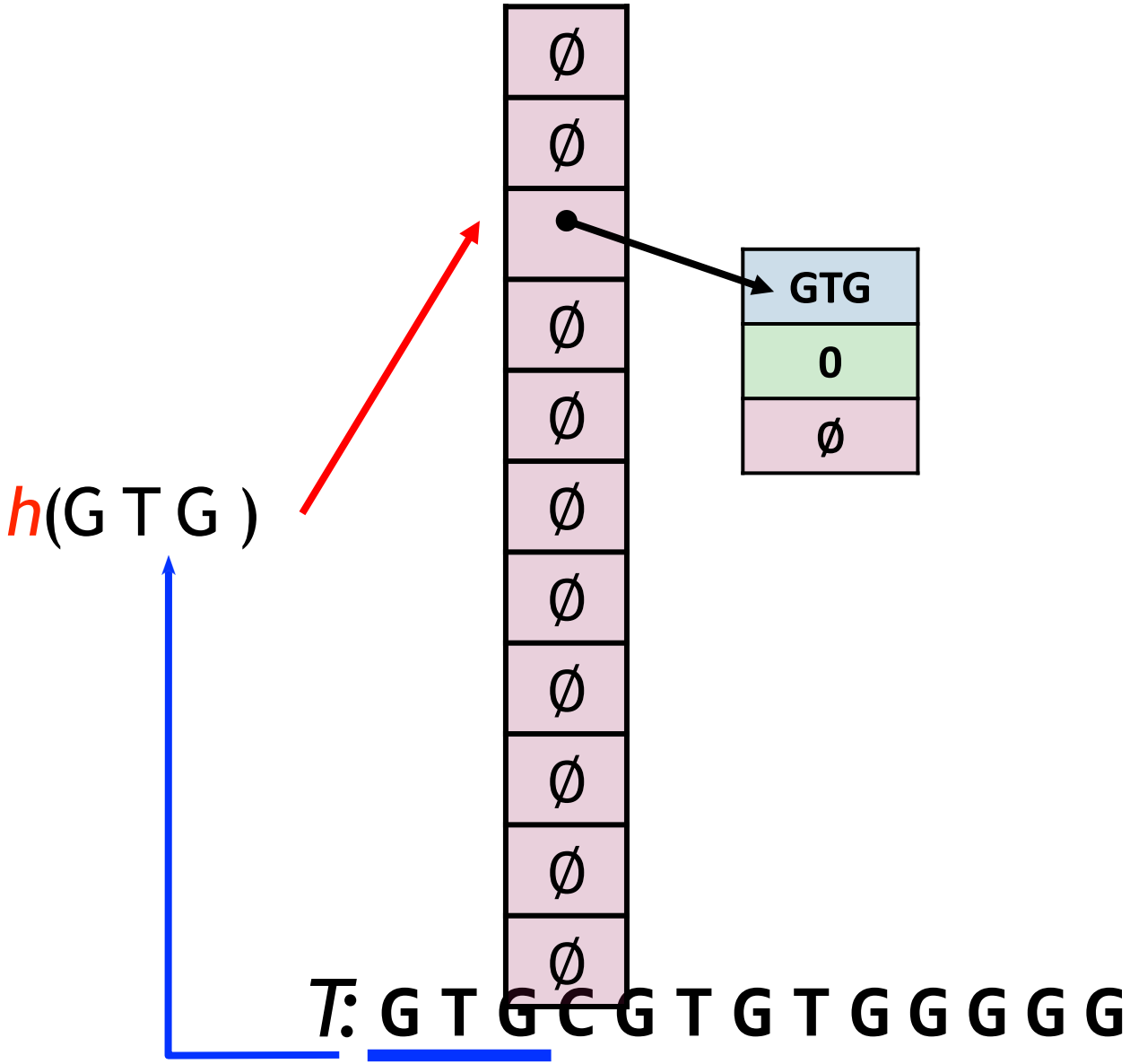
# 3 Hash Tables

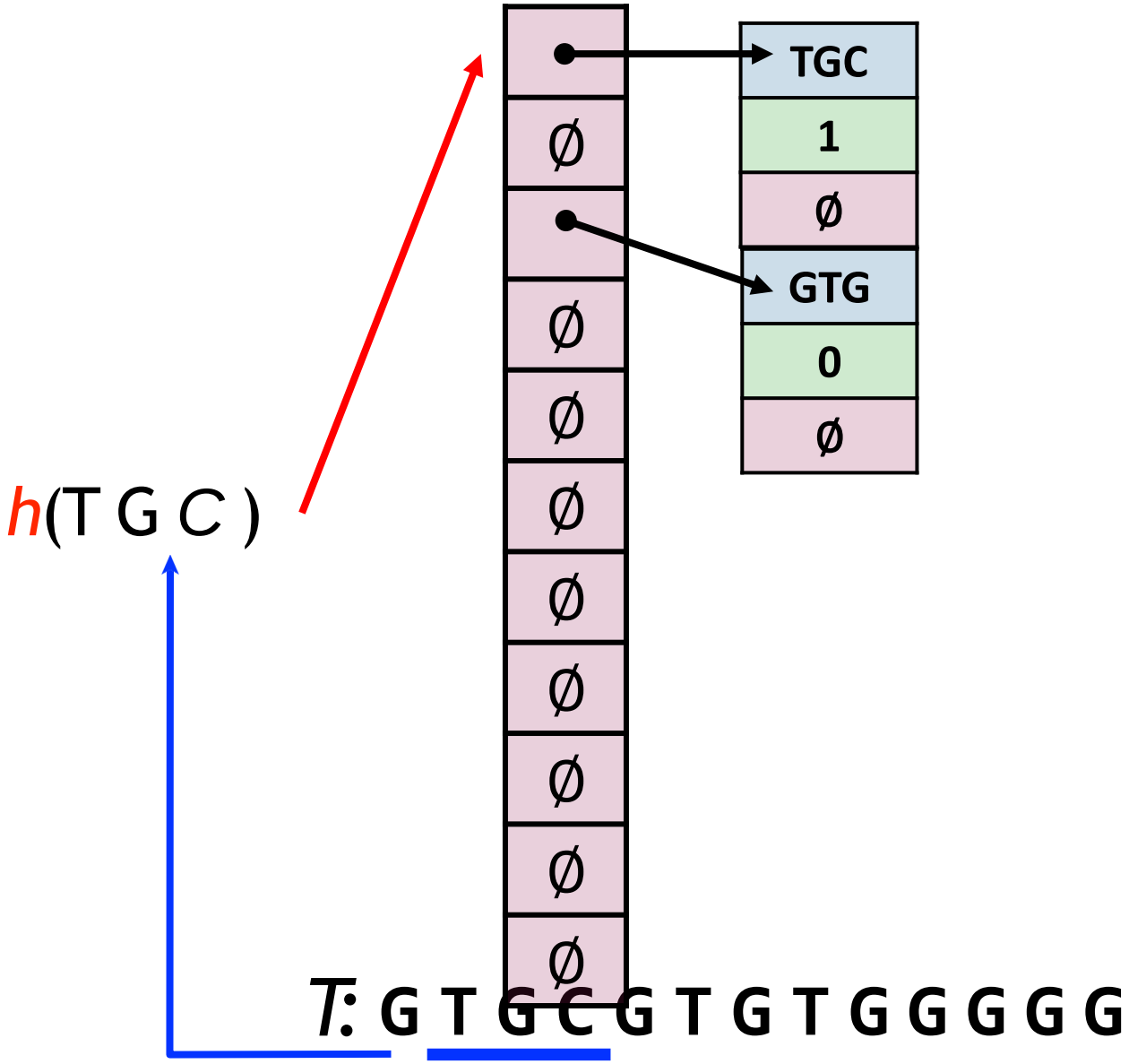


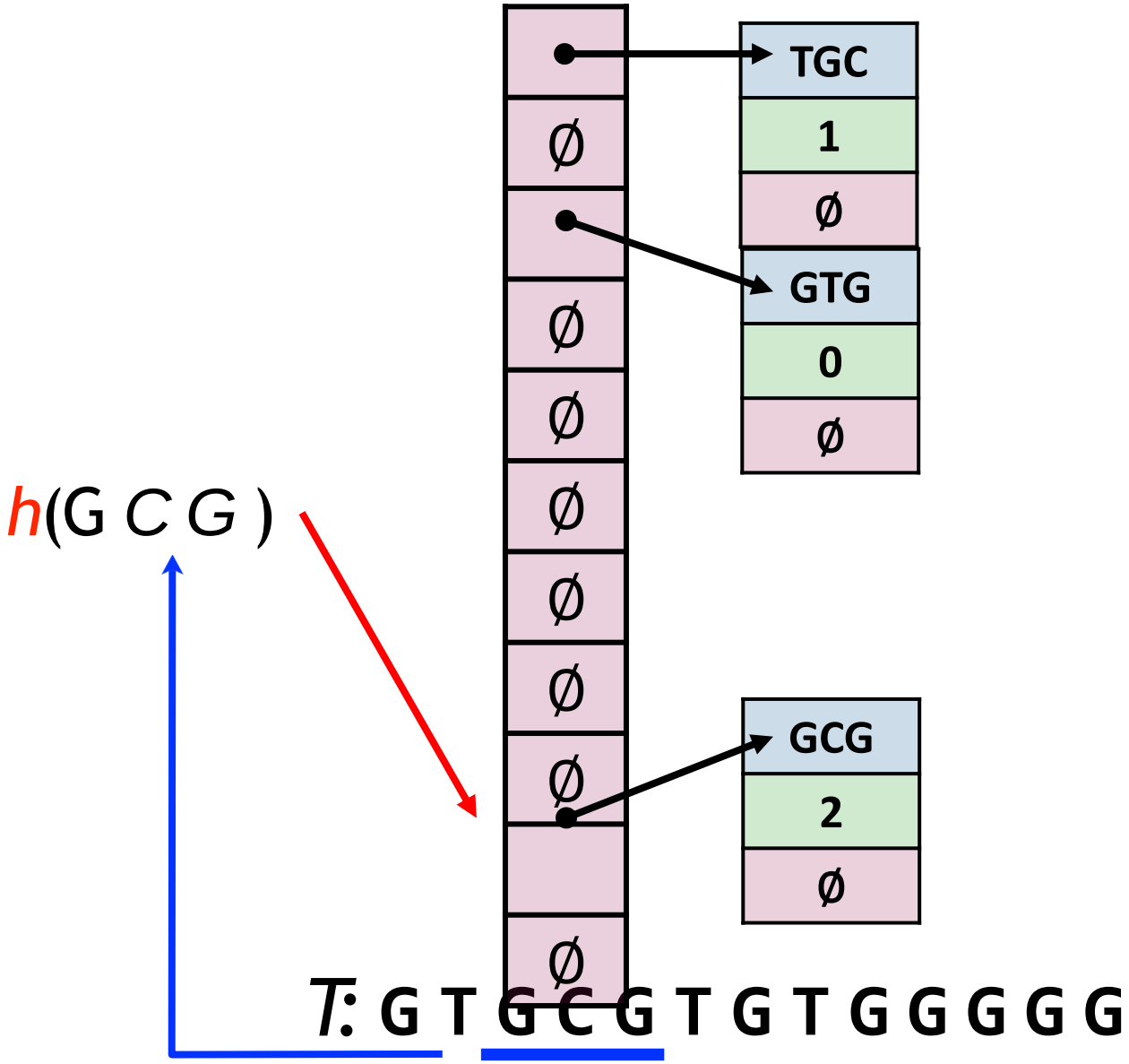
# Hash table as multimap

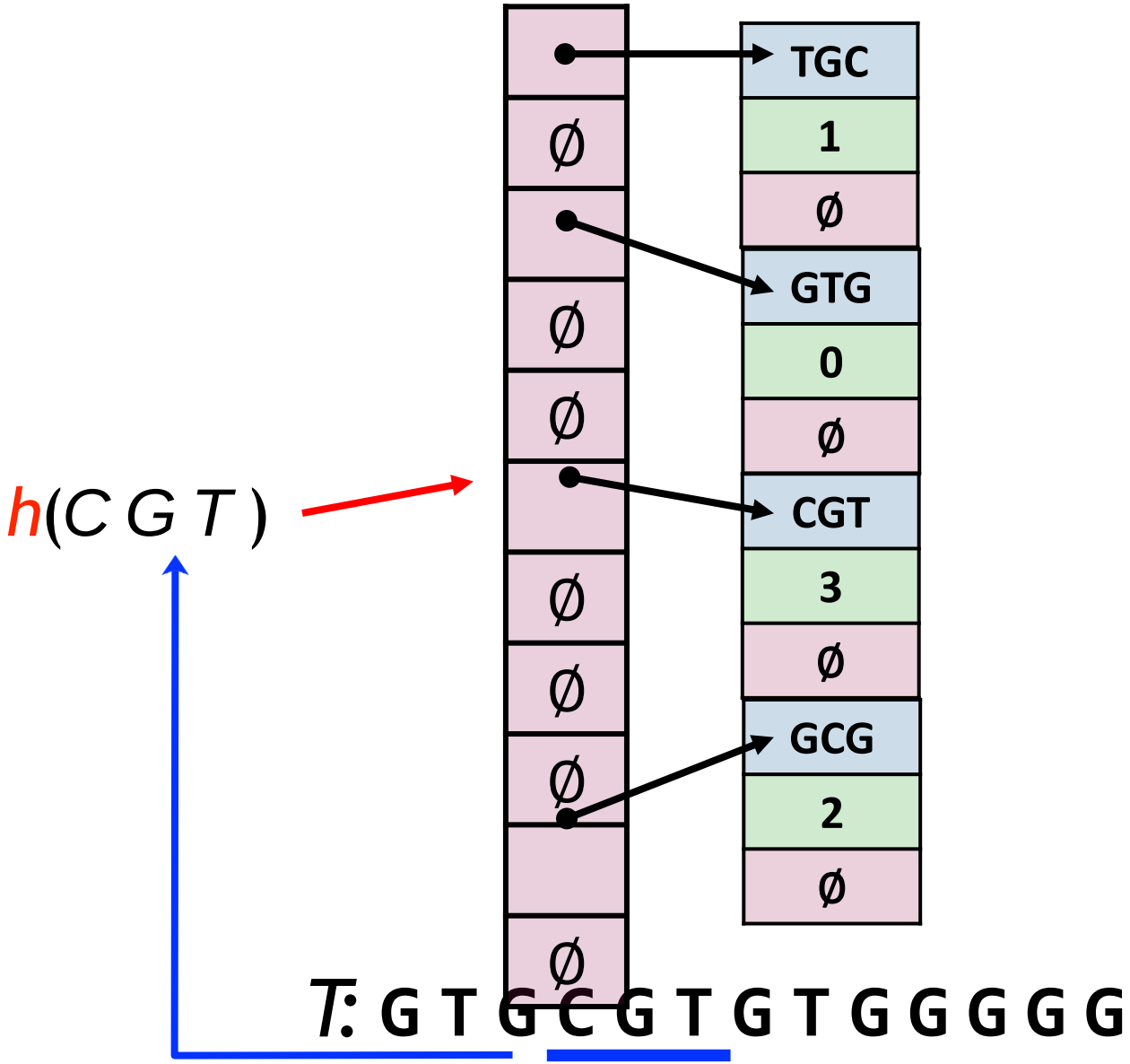


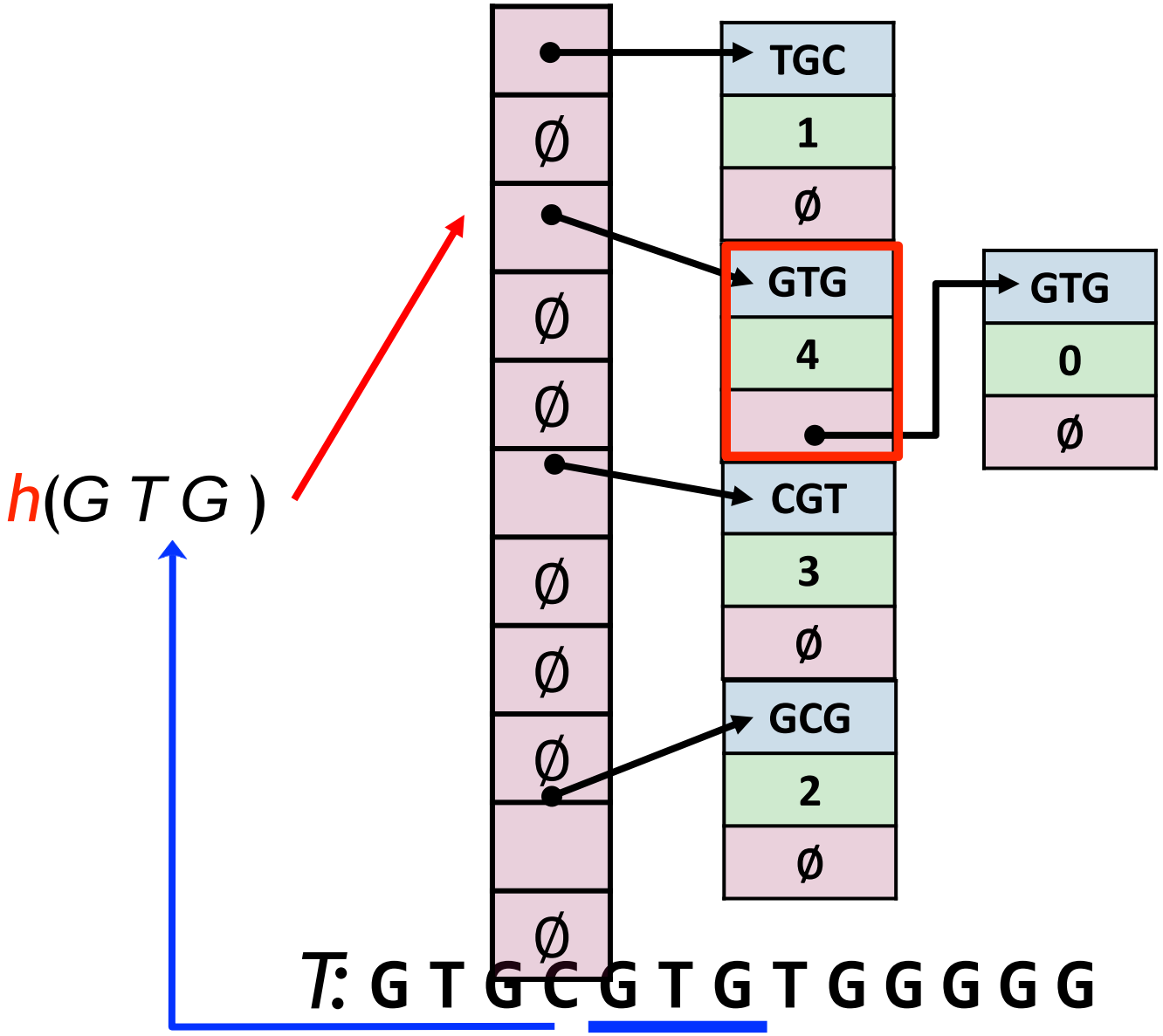

h(G T G )
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
T: G T G C G T G T G G G G

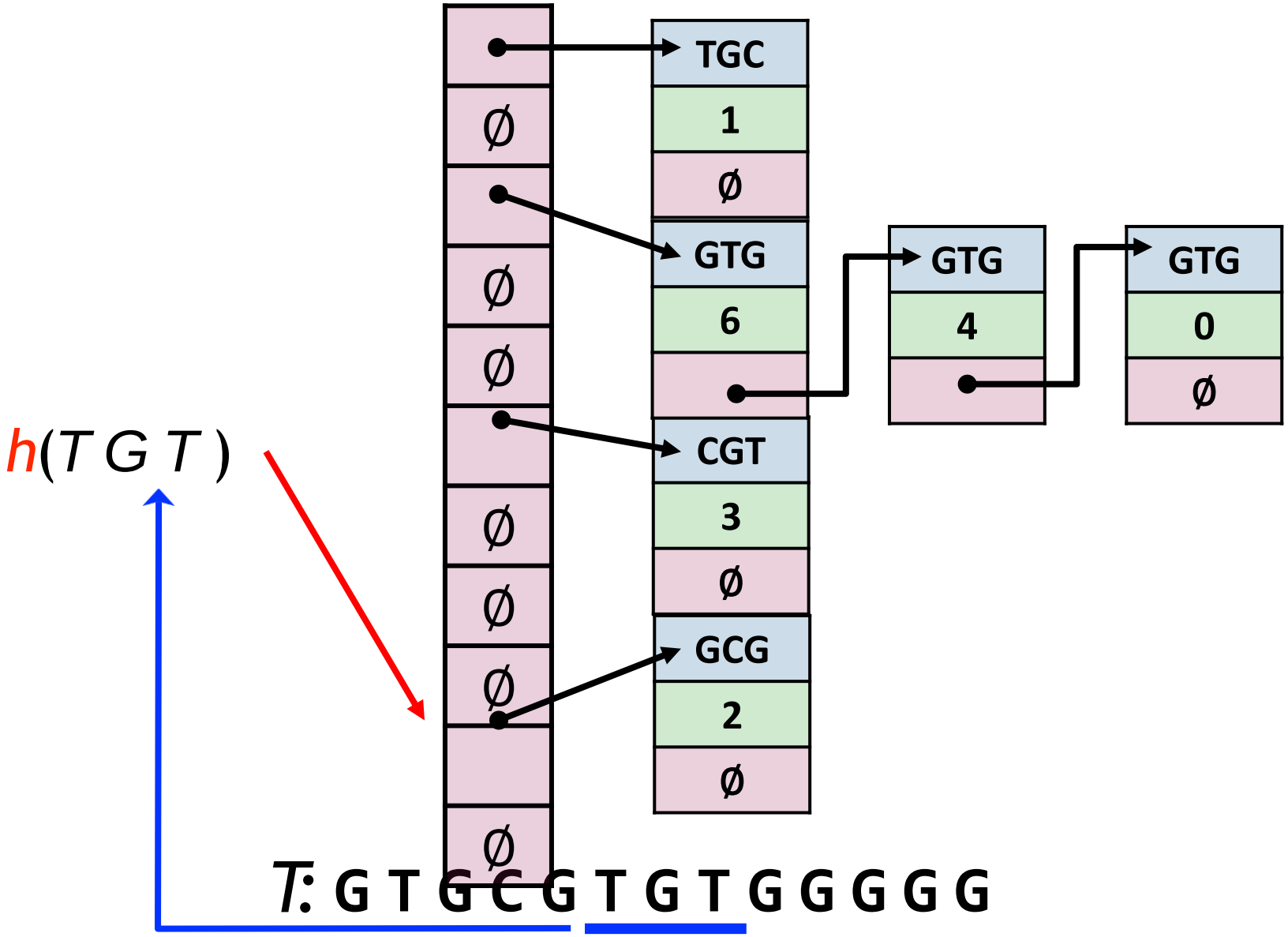

Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
GTG
0
Ø
h(G T G )
T: G T G C G T G T G G G G

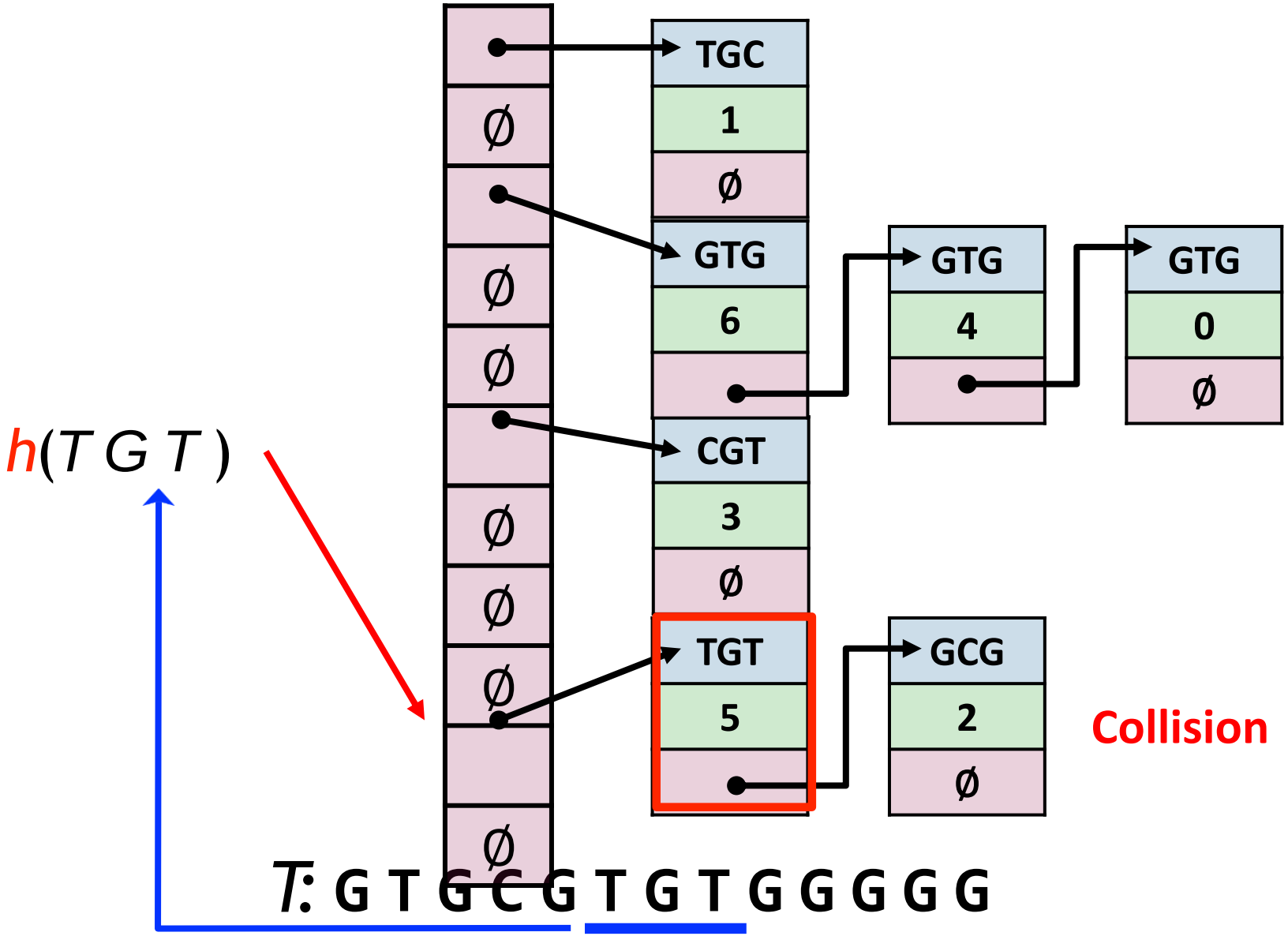

TGC
1
Ø
GTG
0
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
h(T G C )
T: G T G C G T G T G G G G

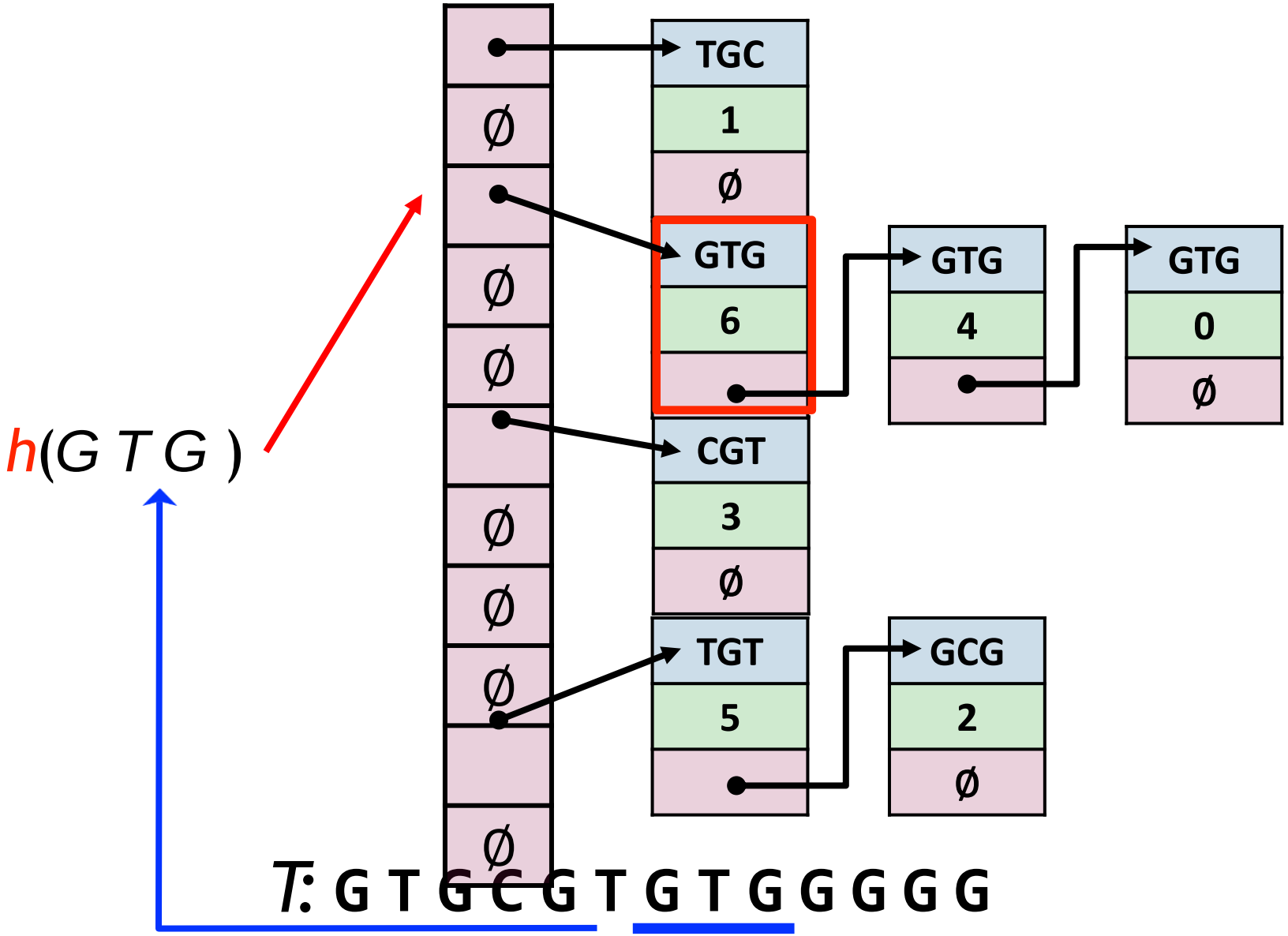

TGC
1
Ø
GTG
0
Ø
GCG
2
Ø
h(G C G )
T: G T G C G T G T G G G G G

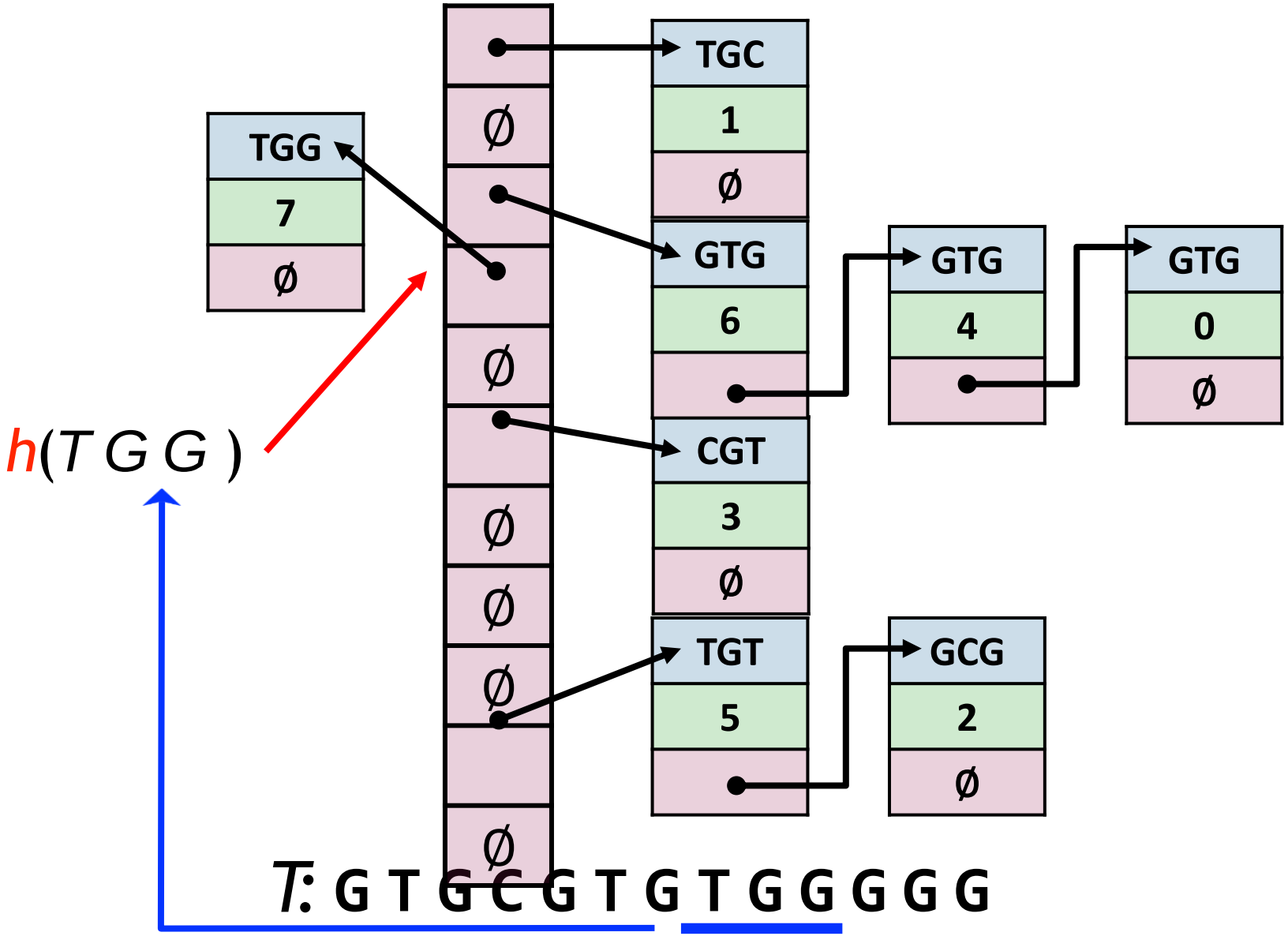

h(C G T )
TGC
1
Ø
GTG
0
Ø
CGT
3
Ø
GCG
2
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
T: G T G C G T G T G G G G G

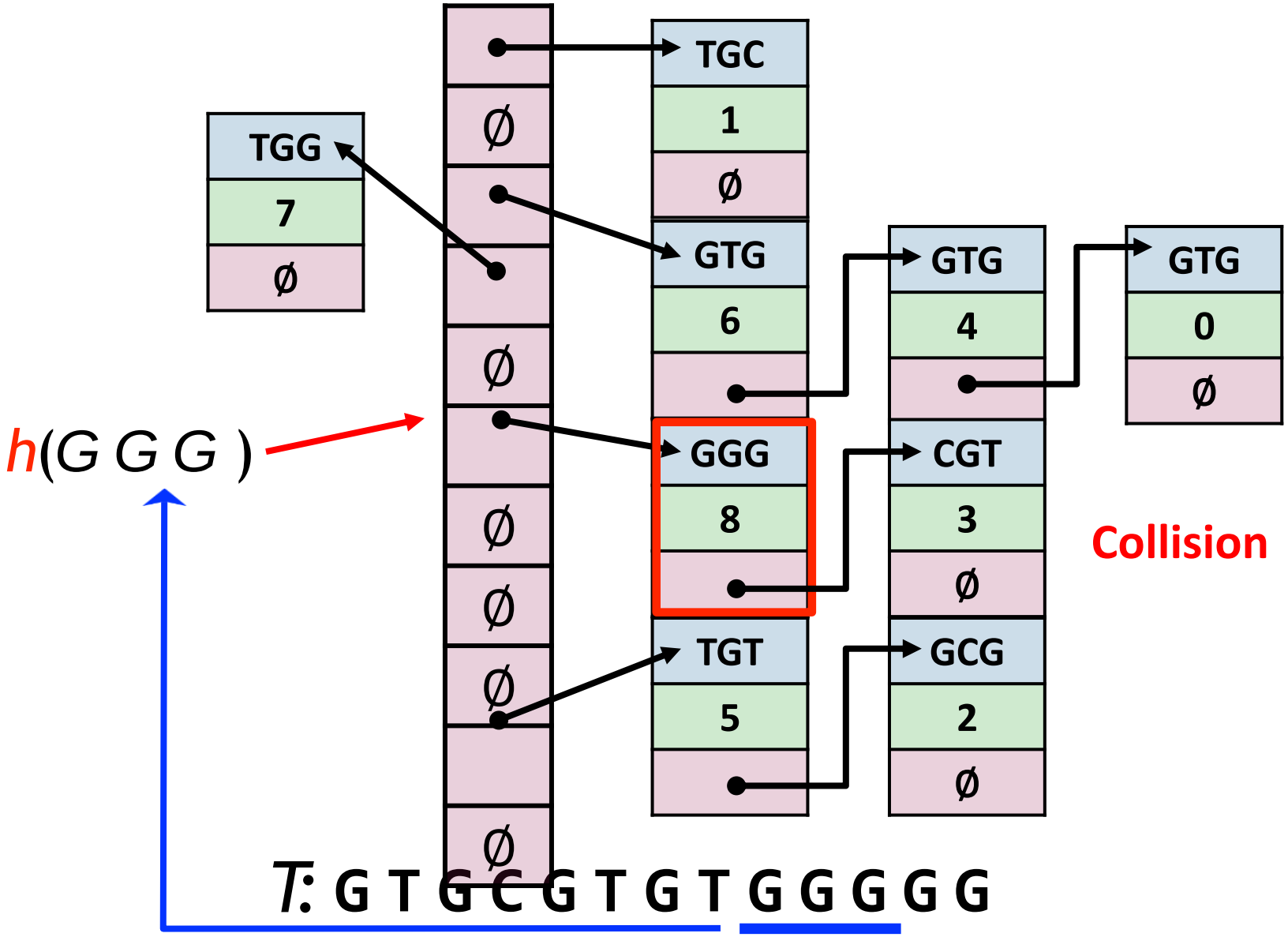

h(G T G )
TGC
1
Ø
GTG
4
GTG
0
Ø
CGT
3
Ø
GCG
2
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
T: G T G C G T G T G G G G G

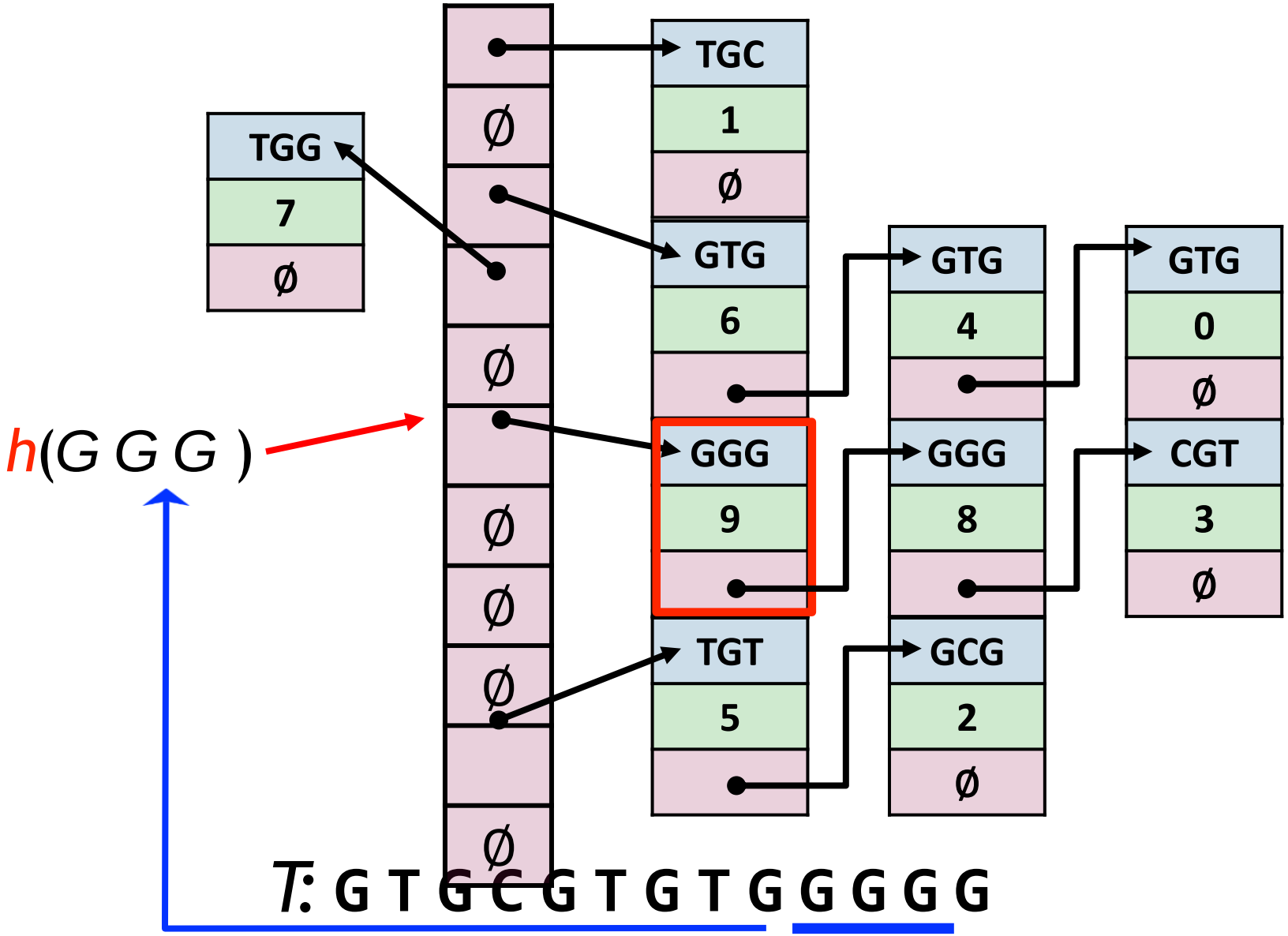

h(T G T )
TGC
1
Ø
GTG
6
CGT
3
Ø
GCG
2
Ø
GTG
4
GTG
0
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
T: G T G C G T G T G G G G G

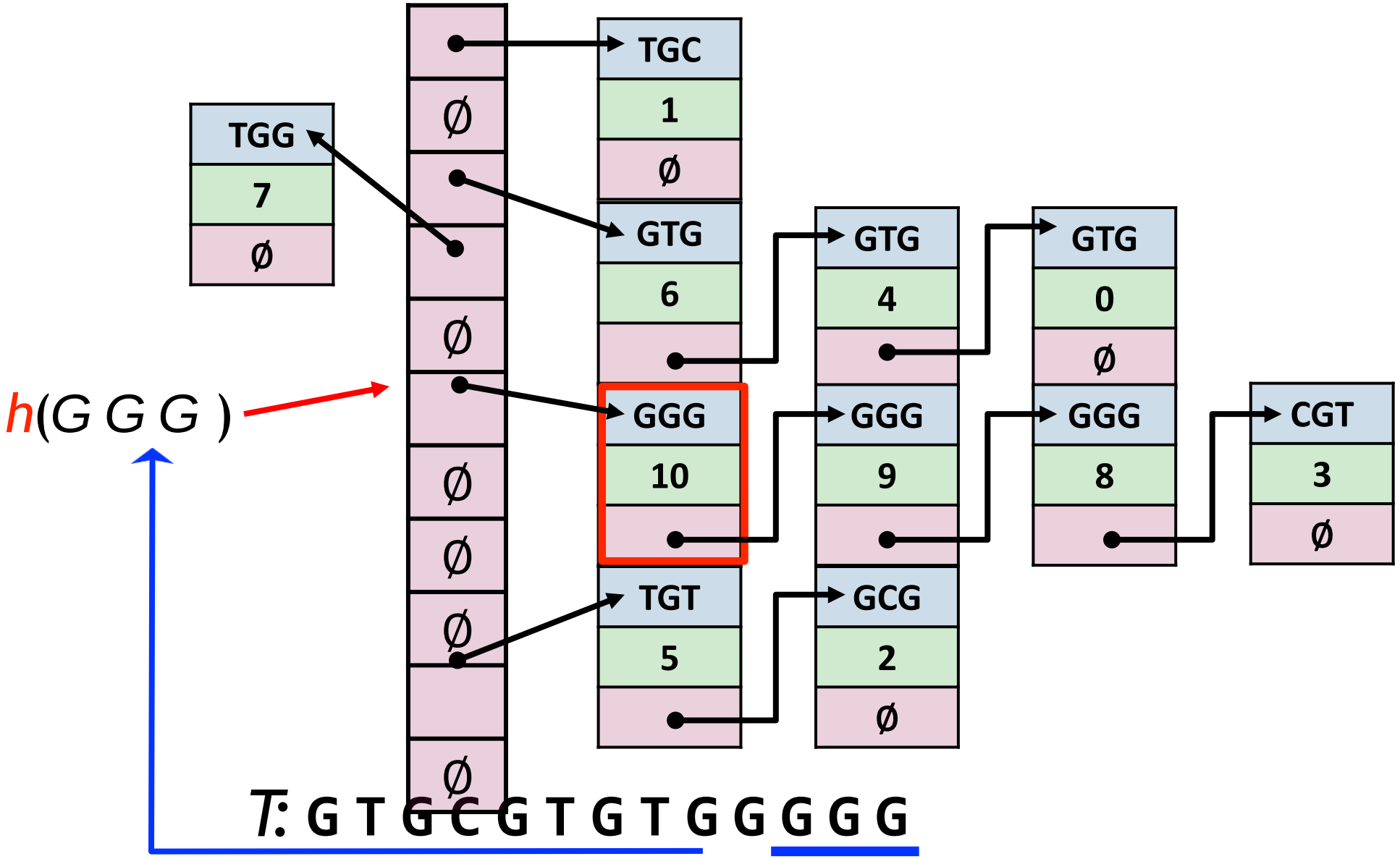

h(T G T )
TGC
1
Ø
GTG
6
GTG
4
GTG
0
Ø
CGT
3
Ø
TGT
5
GCG
2
Ø
Collision
T: G T G C G T G T G G G G G

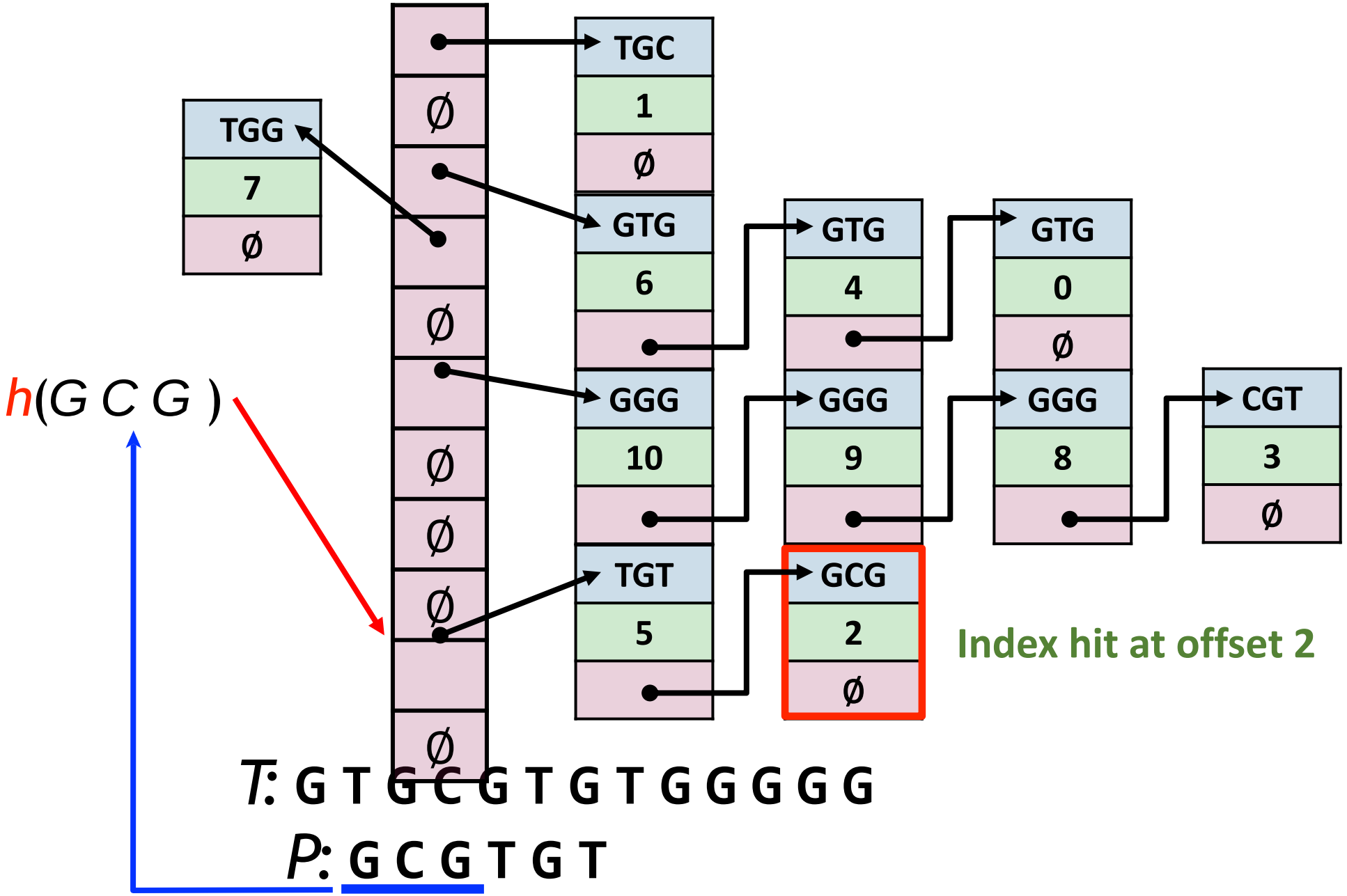

h(G T G )
TGC
1
Ø
GTG
6
GTG
4
GTG
0
Ø
CGT
3
Ø
TGT
5
GCG
2
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
T: G T G C G T G T G G G G


TGC
1
Ø
TGG
7
Ø
GTG
6
GTG
4
GTG
0
Ø
CGT
3
Ø
TGT
5
GCG
2
Ø
h(T G G )
T: G T G C G T G T G G G G

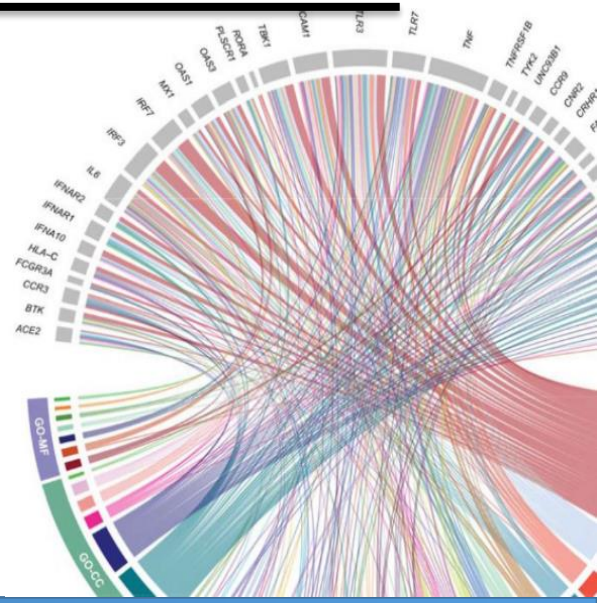

TGG
7
Ø
TGC
1
Ø
GTG
6
GGG
8
TGT
5
GTG
4
CGT
3
Ø
GCG
2
Ø
GTG
0
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
h(G G G )
T: G T G C G T G T G G G G
Collision


TGG
7
Ø
TGC
1
Ø
GTG
6
GGG
9
TGT
5
GTG
4
GGG
8
GCG
2
Ø
GTG
0
Ø
CGT
3
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
h(G G G )
T: G T G C G T G T G G G G


TGG
7
Ø
TGC
1
Ø
GTG
6
GGG
10
TGT
5
GTG
4
GGG
9
GCG
2
Ø
GTG
0
Ø
GGG
8
CGT
3
Ø
Ø
Ø
Ø
Ø
Ø
Ø
Ø
h(G G G )
T: G T G C G T G T G G G G G


TGG
7
Ø
TGC
1
Ø
GTG
6
GGG
10
TGT
5
GTG
4
GGG
9
GCG
2
Ø
GTG
0
Ø
GGG
8
CGT
3
Ø
h(G C G )
T: G T G C G T G T G G G G G
P: G C G T G T
Index hit at offset 2

# 4 Exact Matching Using a Dictionary



# Python Dictionary

- The built-in *dictionary* type in Python is a building block for a map or (in this case) a multimap.

**Python Dictionary**

```
>>> t=  'GTGCGTGTGGGGG'
>>> table = {'GTG':[0, 4, 6], 'TGC':[1],
             'GCG':[2],       'CGT':[3],
             'TGT':[5],       'TGG':[7],
             'GGG':[8, 9, 10]}
>>>  table['GGG']
[8,  9,  10]
>>>  table['CGT']
[3]
```

http://j.mp/CG_KmerIndexHash

# Exact Matching using Python Dictionaries

## (Step 1) Read a FASTA FILE

**Parse FASTA**

```
import argparse

def read_fasta(file_path):
  """Read a FASTA file and return a string with all the content"""
  sequence = []
  with open(file_path, 'r') as file:
    for line in file:
      if not line.startswith('>'):
        sequence.append(line.strip())  # Remove newline and concatenate
  return ''.join(sequence)
```

# Exact Matching using Python Dictionaries

## (Step 2) Create a Kmer Hash Table (Using a Python Dictionary)

**Create Hash Table**

```
def build_kmer_index(sequence, k):
  """Create a hash table of k-mers and their positions."""
  kmer_index = {}
  for i in range(len(sequence) - k + 1):
    kmer = sequence[i:i+k]
    if kmer in kmer_index:
      kmer_index[kmer].append(i)
    else:
      kmer_index[kmer] = [i]
  return kmer_index
```

# Exact Matching using Python Dictionaries

## (Step 3) Search for all the kmers contained in the sequence and verify (exact match)

**Search for k-mers**

```
def search_sequence(reference, kmer_hash, k, sequence):
  """Search all k-mers from the input sequence and verify exact matches."""
  if len(sequence) < k:
    print("Input sequence is shorter than k-mer size.")
    return []

  occurrences = []
  for i in range(len(sequence) - k + 1):
    # Find where this k-mer appears
    kmer = sequence[i:i+k]
    candidate_positions = kmer_hash.get(kmer,[])
    print("For k-mer '%s' found %d candidate positions" % (kmer,len(candidate_positions)))
    # Verify exact-match of the sequence at each found position in the reference
    for pos in candidate_positions:
      if reference[pos-i:pos-i+len(sequence)] == sequence:
        occurrences.append(pos - i)

  return list(set(occurrences)) # Remove duplicates
```

# Exact Matching using Python Dictionaries

## (Step 4) Test it!

**Testing…**

```
# Arguments Parser
parser = argparse.ArgumentParser(description="…")
parser.add_argument("-r","--reference",help="Path to the FASTA file",required=True)
parser.add_argument("-i","--input-seq",help="Input Sequence",required=True)
parser.add_argument("-k","--kmer-len",help="K-mer length",default=6)
args = parser.parse_args()
sequence = args.input_seq
# Read FASTA file
print("Reading input FASTA ...",end="")
reference = read_fasta(args.reference)
print(" read %d bases from '%s'." % (len(reference),args.reference))
# Build hash table
print("Building hash table ...")
kmer_hash = build_kmer_index(reference,int(args.kmer_len))
# Search sequence
print("Searching '%s' sequence ..." % sequence)
occ = search_sequence(reference,kmer_hash,int(args.kmer_len),sequence)
# Print Results
print("Found %d exact matches:" % len(occ))
for pos in occ:
  print("%d " % pos,end="")
```

# Exact Matching using Python Dictionaries

## (Step 4) Test it!

**Download chr1.fa**

```
> wget http://hgdownload.soe.ucsc.edu/goldenPath/hg38/chromosomes/chr1.fa.gz
> gunzip chr1.fa.gz
```

**Execute test program**

```
python3 hash_index.py -r ../chr1.fa -i GAGCAATAAATT
Reading input FASTA ... read 248956422 bases from '../chr1.fa'.
Building hash table ...
Searching 'GAGCAATAAATT' sequence ...
For k-mer 'GAGCAA' found 30694 candidate positions
For k-mer 'AGCAAT' found 35723 candidate positions
For k-mer 'GCAATA' found 23696 candidate positions
For k-mer 'CAATAA' found 42709 candidate positions
For k-mer 'AATAAA' found 117816 candidate positions
For k-mer 'ATAAAT' found 85561 candidate positions
For k-mer 'TAAATT' found 76296 candidate positions
Found 24 exact matches:
76905859 197681927 55911317 99978007 186064024 74531228 35650472 18284467
108760244 202986424 63232569 223912637 273214 231021635 170791884 95476817
71594585 62609371 56068705 200519651 188493415 207659497 243015401 96630
```

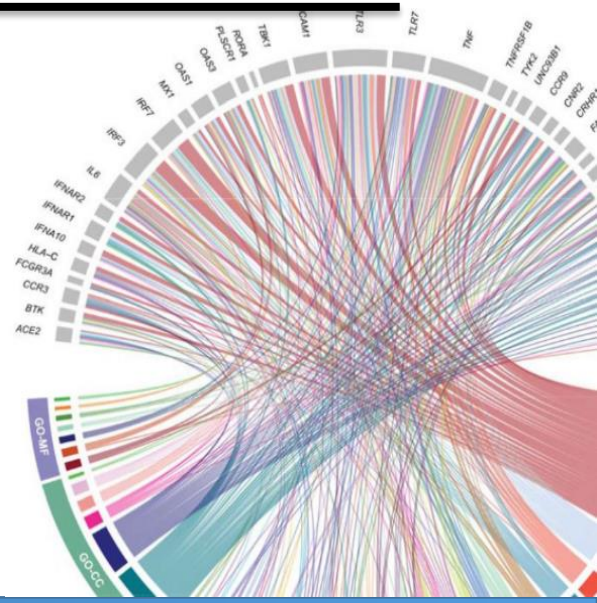
# Exercises

1. **Check only the least repetitive k-mer (Easy)**
   - Implement the necessary modifications to only verify (i.e., exact matching) that kmer that occurs in the reference (hash table) the least amount of times.
2. **Influence of the value k in the performance (Mid)**
   - Investigate the trade-off between time and memory depending on the value of k

**Testing values of k**

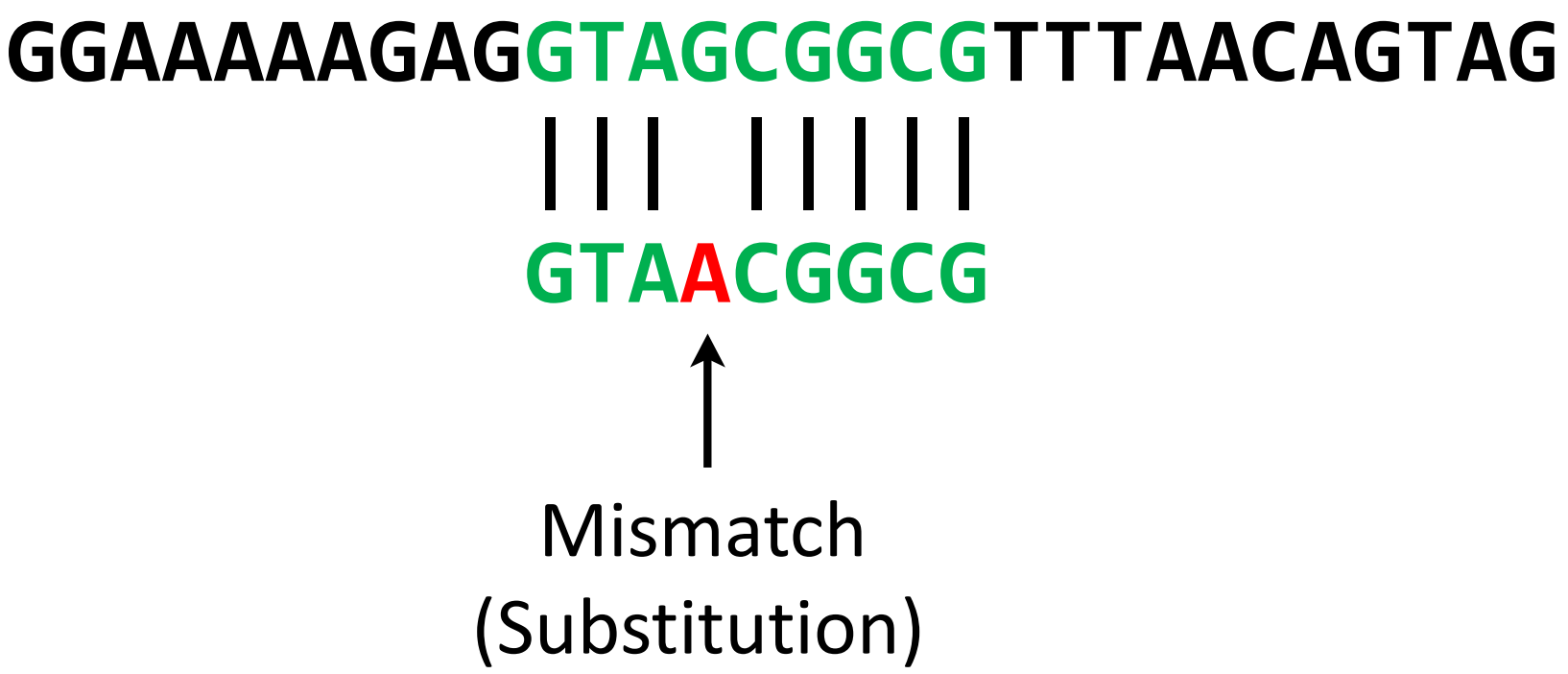
```
> \time -v python3 hash_index.py -r ../chr1.fa -i GAGCAATAAATT -k 6
Found 24 exact matches.
Time: 74.39 s
Memory: 10.01 GB

> \time -v python3 hash_index.py -r ../chr1.fa -i GAGCAATAAATT -k 8
Found 24 exact matches.
Time: 150.53 s
Memory: 10.31 GB

> \time -v python3 hash_index.py -r ../chr1.fa -i GAGCAATAAATT -k 10
Found 24 exact matches.
Time: 216.48 s
Memory: 11.5 GB
```

# 5 The Problem of Approximate Matches



# Exact Matching using Python Dictionaries

- Sequences not always match the reference genome exactly (exact match).
- Many sources for differences: Single Nucleotide Polymorphism (SNP), errors during sequencing or base calling.
- We need to allow errors or differences -> Approximate String Matching



# Hamming Distance

For $X$ & $Y$ where $|X| = |Y|$, *hamming distance* = minimum # substitutions needed to turn one into the other


*Hamming distance* = 3

# Hamming Distance (between 2 Sequences)

**Compute Hamming Distance**

```
seq1 = "ACGTGCA"
seq2 = "ATGAGGA"

distance = hamming_distance(seq1, seq2)
print(distance)  # Output: 3

alignment = hamming_alignment(seq1, seq2)
print(alignment)  # Output: ['M', 'X', 'M', 'X', 'M', 'X', 'M']

# Pretty print
# ACGTGCA
# | | | |
# ATGAGGA
pretty_print_alignment(seq1, seq2, alignment)
```

# Hamming Distance (between 2 Sequences)

**Compute Hamming Distance**

```
def hamming_distance(seq1, seq2):
  """Compute the Hamming distance between two sequences of equal length."""
  if len(seq1) != len(seq2):
    raise ValueError("Sequences must be of the same length")

  distance = 0
  for i in range(len(seq1)):
    if seq1[i] != seq2[i]:
      distance += 1
  return distance
```

# Hamming Alignment (between 2 Sequences)

**Compute Hamming Alignment**

```
def hamming_alignment(seq1, seq2):
  """Generate a list of 'M' (match) and 'X' (mismatch) based on Hamming distance."""
  if len(seq1) != len(seq2):
    raise ValueError("Sequences must be of the same length")

  alignment = []
  for i in range(len(seq1)):
    if seq1[i] == seq2[i]:
      alignment.append('M')
    else:
      alignment.append('X')
  return alignment
```

# Hamming Alignment (between 2 Sequences)

**Pretty Print Hamming Alignment**

```
def pretty_print_alignment(seq1, seq2, alignment):
  """Pretty prints the alignment of two sequences with match markers."""
  if len(seq1) != len(seq2) or len(seq1) != len(alignment):
    raise ValueError("Sequences and alignment must have the same length")

  match_line = ''.join('|' if symbol == 'M' else ' ' for symbol in alignment)

  print(seq1)
  print(match_line)
  print(seq2)
```

# Exercises

**What if? …**

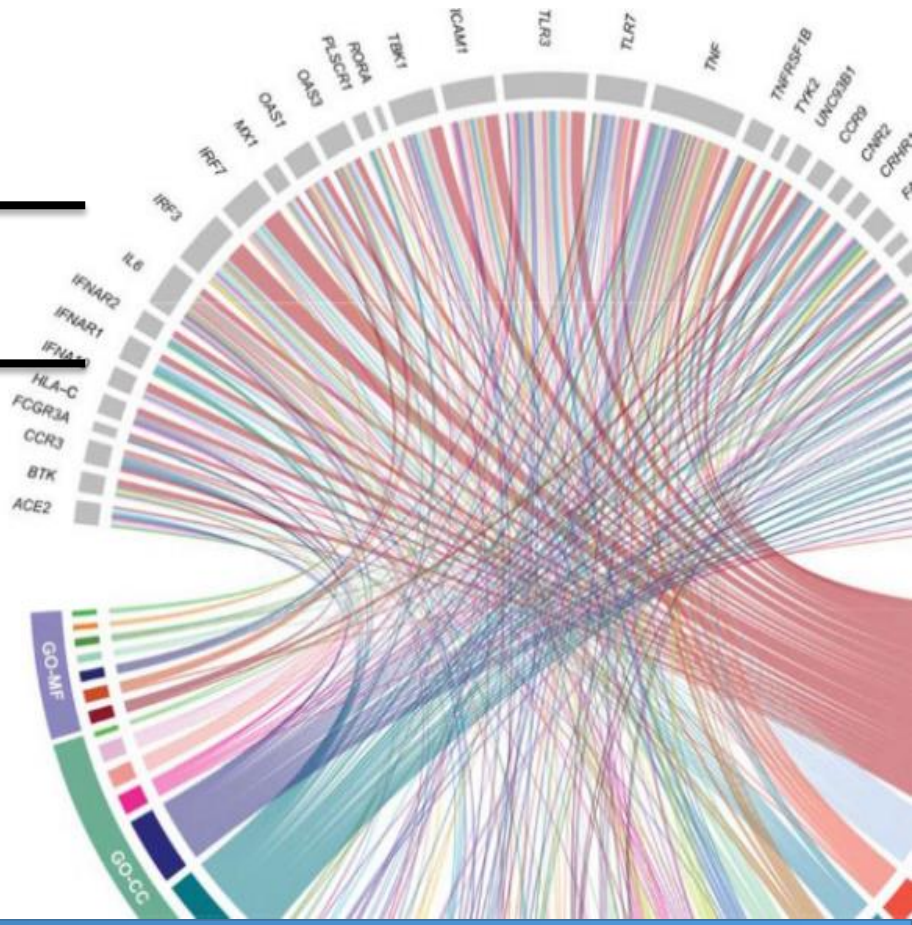
**… Enhance our kmer-index based mapper to use the hamming distance.**

**… k values?**

**… error values?**

**… which mappings are better?**

# Questions


ICAM1
TLR3
TLR7
TNF
TNFRSF1B
TYK2
UNC93B1
CCR9
CNR2
TBK1
RORA
PLSCR1
OAS3
OAS1
MX1
IRF7
IRF3
IL6
IFNAR2
IFNAR1
HLA-C
FCGR3A
CCR3
BTK
ACE2
GO-MF
GO-CC