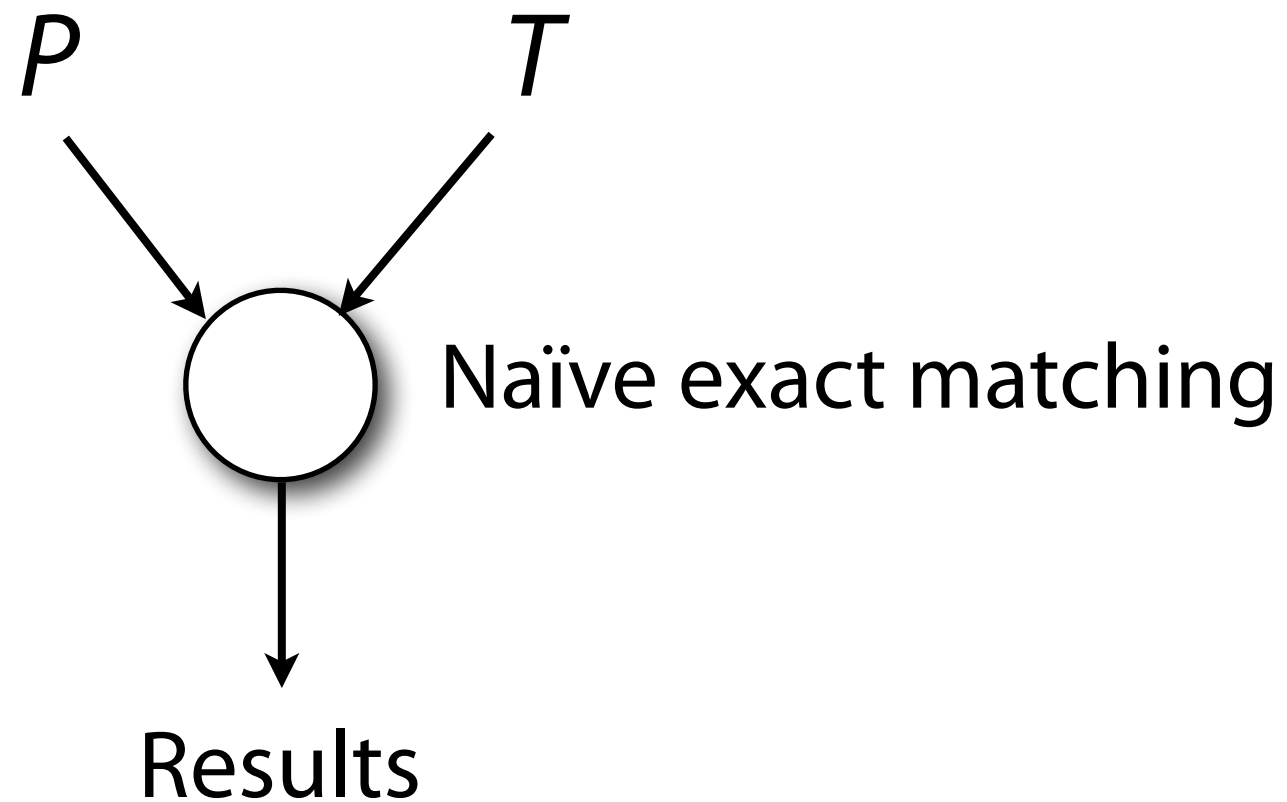
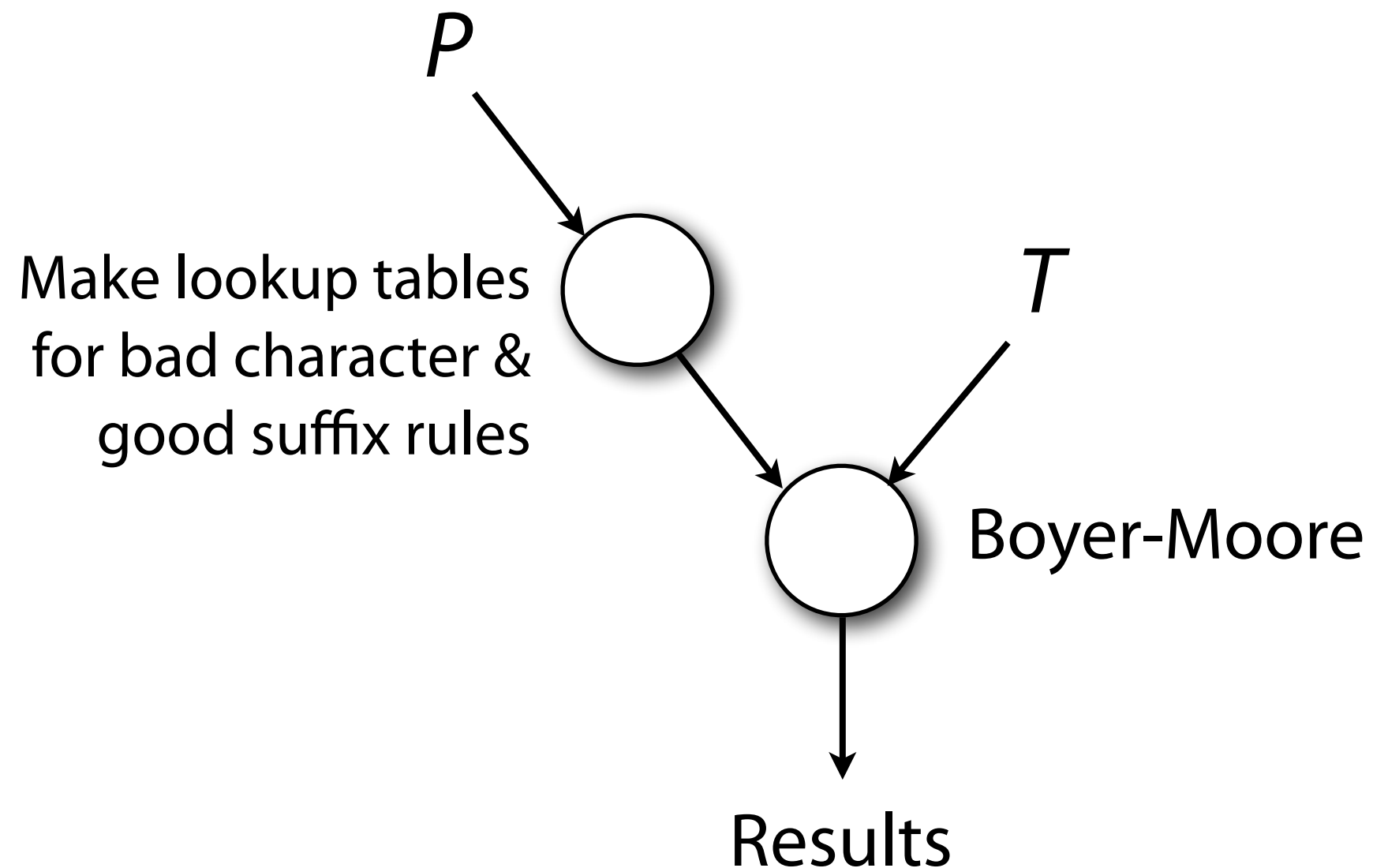


Indexing

Preprocessing: Naïve algorithm



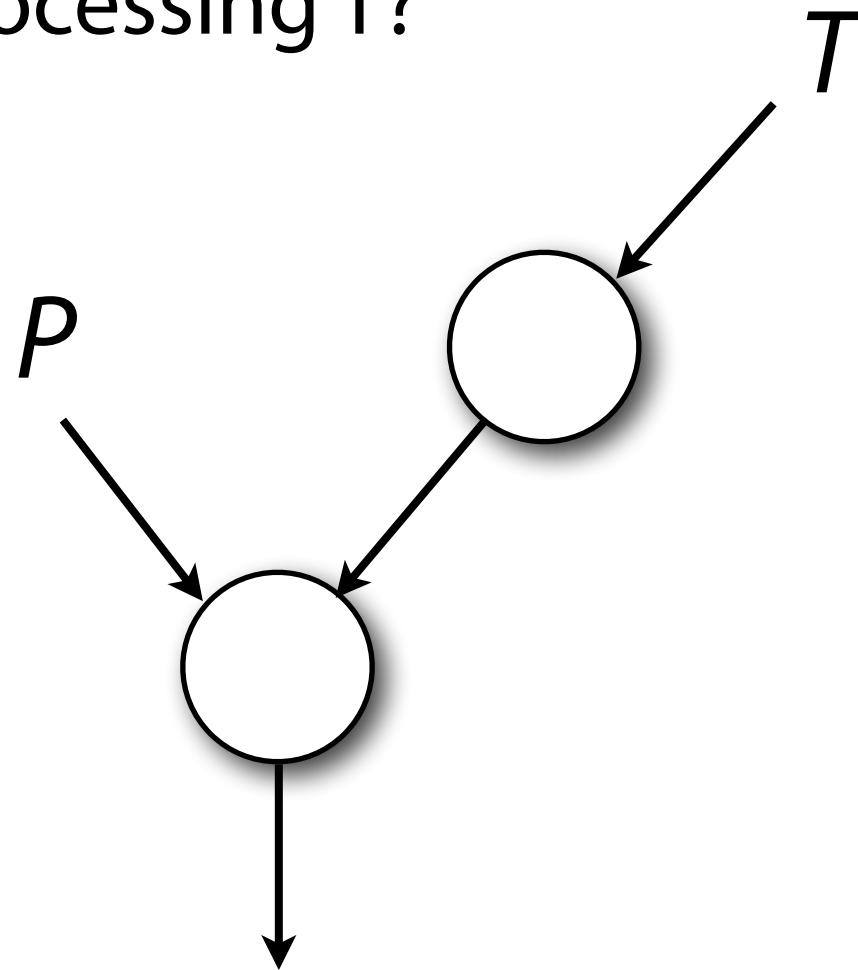
Preprocessing: Boyer-Moore



Preprocessing

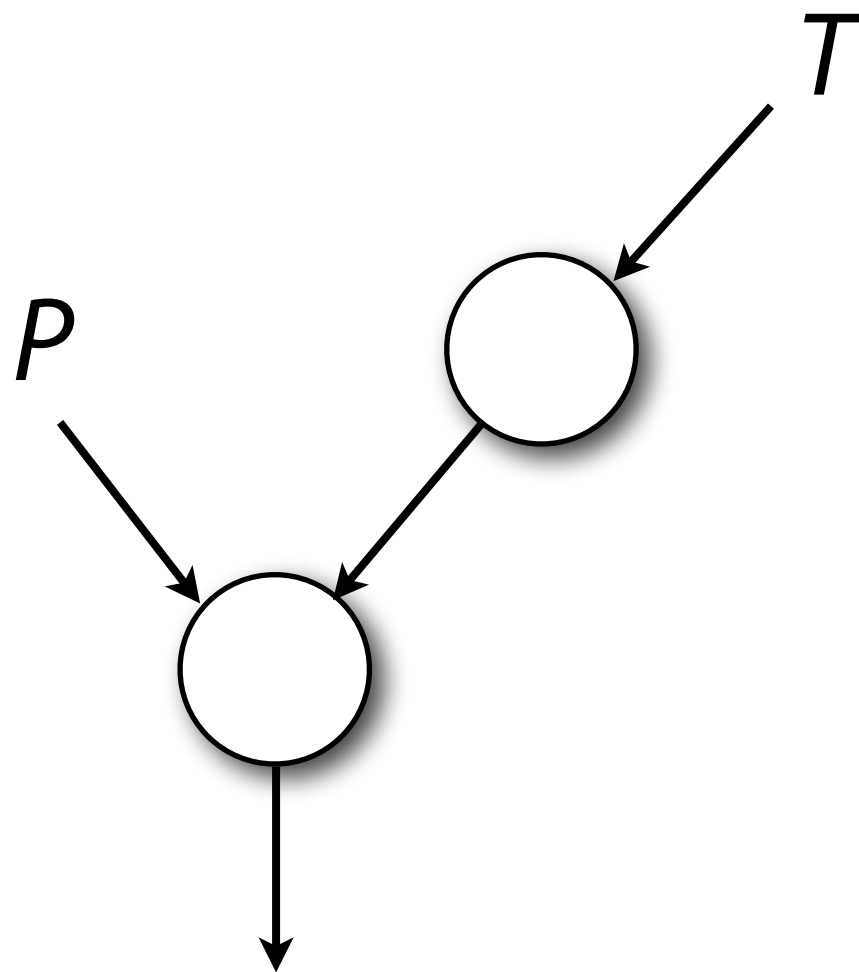
Boyer-Moore preprocessed P

What about preprocessing T ?



Preprocessing

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

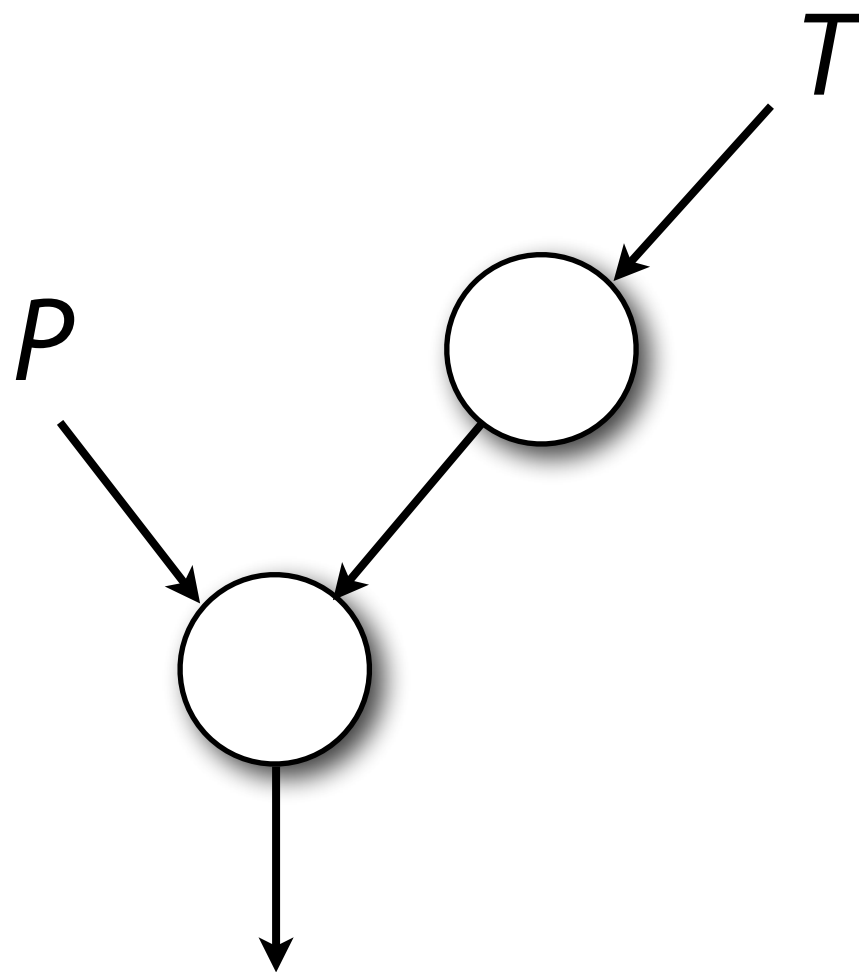


Online or offline?

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

Preprocessing

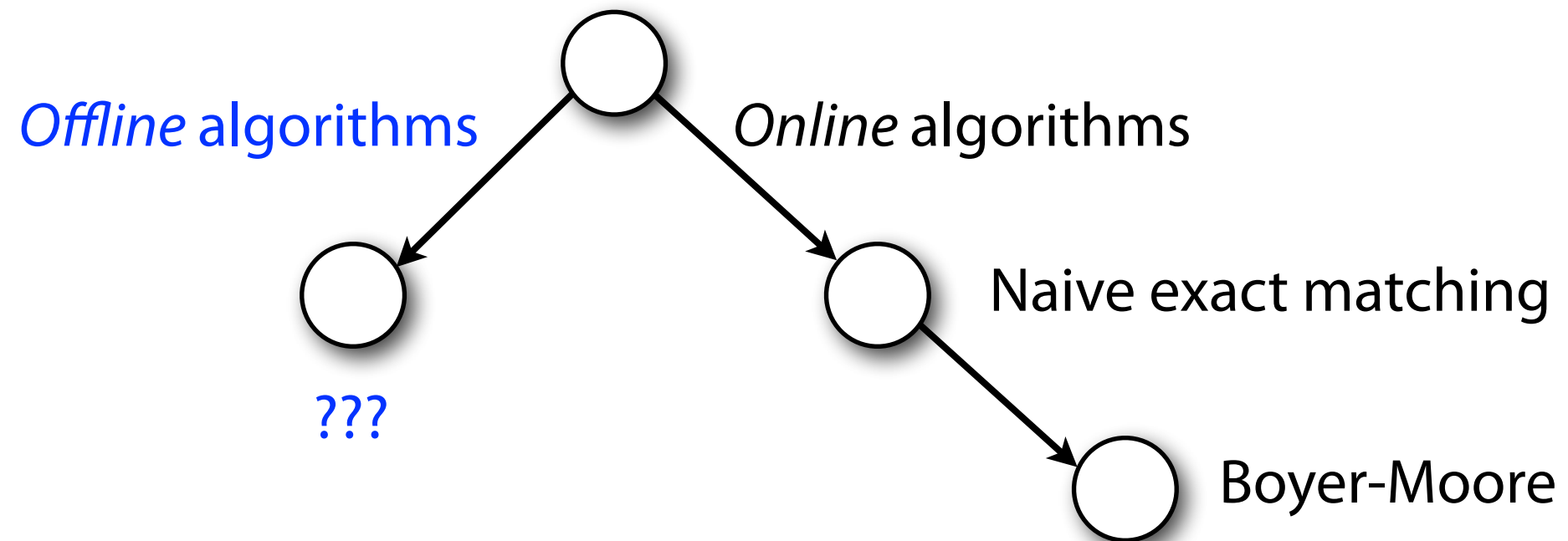
Algorithm that preprocesses T is *offline*.
Otherwise, algorithm is *online*.



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

nest site hunting, 482–87	<i>Macrotermes</i> (termites), 59–60
honeypot ants, <i>see Myrmecocystus</i>	male recognition, 298
hormones, 106–9	mass communication, 62–63, 214–18
<i>see also</i> exocrine glands	mating, multiple, 155
house (nest site) hunting, 482–92	maze following, 119
Hymenoptera (general), xvi	<i>Megalomyrmex</i> (ants), 457
haplodiploid sex determination, 20–22	<i>Megaponera</i> (ants), <i>see Pachycondyla</i>
<i>Hypoponera</i> (ants), 194, 262, 324, 388	<i>Melipona</i> (stingless bees), 129
	<i>Melophorus</i> (ants), repletes, 257
inclusive fitness, 20–23, 29–42	memory, 117–19, 213
information measurement, 251–52	<i>Messor</i> (harvester ants), 212, 232
intercastes, 388–89	mind, 117–19
<i>see also</i> ergatogynes; ergatoid queens;	<i>Monomorium</i> , 127, 212, 214, 216–17,
gamergates	292
<i>Iridomyrmex</i> (ants), 266, 280, 288, 321	motor displays, 235–47
Isoptera, <i>see</i> termites	mound-building ants, 2
	multilevel selection, 7, 7–13, 24–29
juvenile hormone, caste, 106–9, 372	mutilation, ritual, 366–73
kin recognition, 293–98	mutualism, <i>see</i> symbioses, ants
kin selection, 18–19, 23–24, 28–42, 299,	<i>Myanmyrma</i> (fossil ants), 318
386	<i>Myopias</i> (ants), 326

Key terms ordered alphabetically, with associated page #s

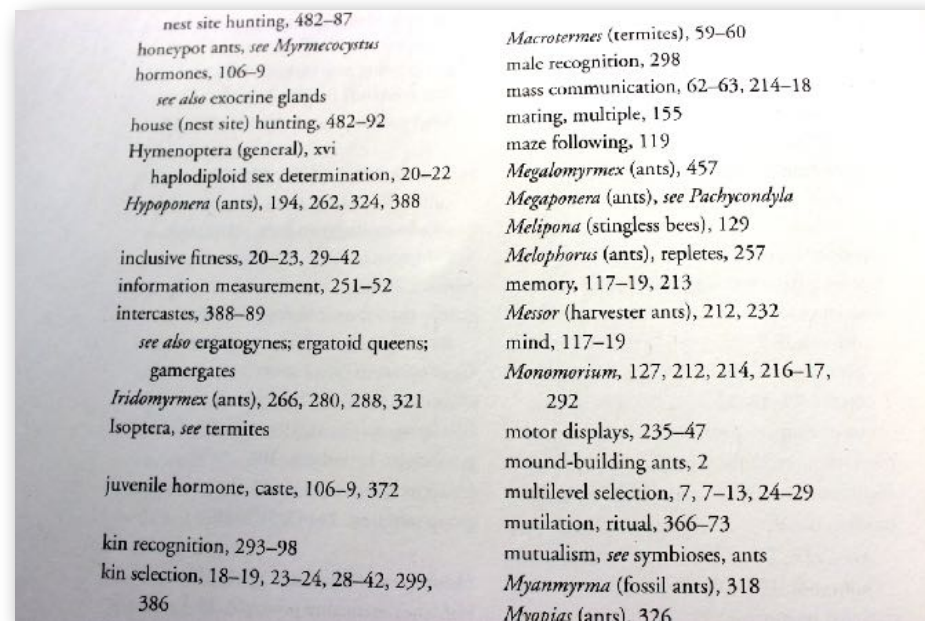
Index



Grocery store items grouped into aisles

Index

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



Indexing DNA

Index of T

A large orange square is positioned in the center of the slide. It is empty, representing a data structure or a table used for indexing.

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

Indexing DNA

Index of T
C G T G C : 0

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

Indexing DNA

Index of T

C G T G C : 0

G T G C G : 1

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

Indexing DNA

Index of T

C G T G C : 0

G T G C G : 1

T G C G T : 2

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

Indexing DNA

<i>Index of T</i>	
C G T G C :	0
→ G C G T G :	3
G T G C C :	1
T G C C T :	2

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

Indexing DNA

Index of T 

C G T G C :	0 , 4
G C G T G :	3
G T G C C :	1
T G C C T :	2

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

Indexing DNA

Index of T

C G T G C : 0 , 4

G C G T G : 3

G T G C C : 1

G T G C T : 5

T G C C T : 2

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

Indexing DNA

Index of T

C G T G C : 0 , 4

G C G T G : 3

G T G C C : 1

G T G C T : 5

T G C C T : 2

T G C T T : 6

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

Indexing DNA

k-mer: substring
of length *k*

<i>Index of T</i>	
C G T G C :	0 , 4
G C G T G :	3
G T G C C :	1
G T G C T :	5
T G C C T :	2
T G C T T :	6

5-mer index

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

Querying the index

Index of T

C G T G C : 0 , 4

G C G T G : 3

G T G C C : 1

G T G C T : 5

T G C C T : 2

T G C T T : 6

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

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

Querying the index

Index of T

C G T G C :	0 , 4
G C G T G :	3
G T G C C :	1
G T G C T :	5
T G C C T :	2
T G C T T :	6

?

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

P: G C G T G C

Querying the index

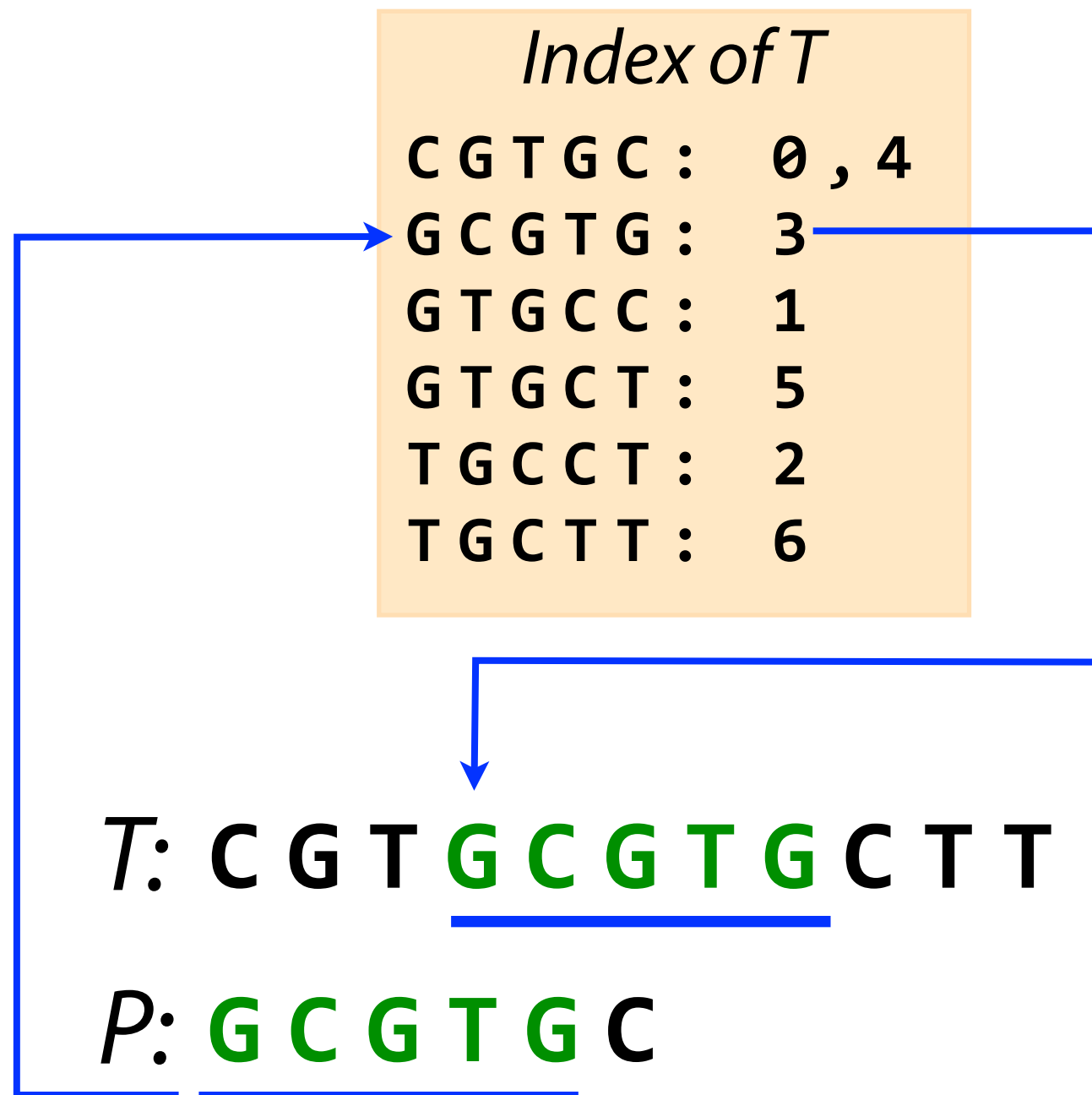
Index of T

C G T G C :	0 , 4
G C G T G :	3
G T G C C :	1
G T G C T :	5
T G C C T :	2
T G C T T :	6

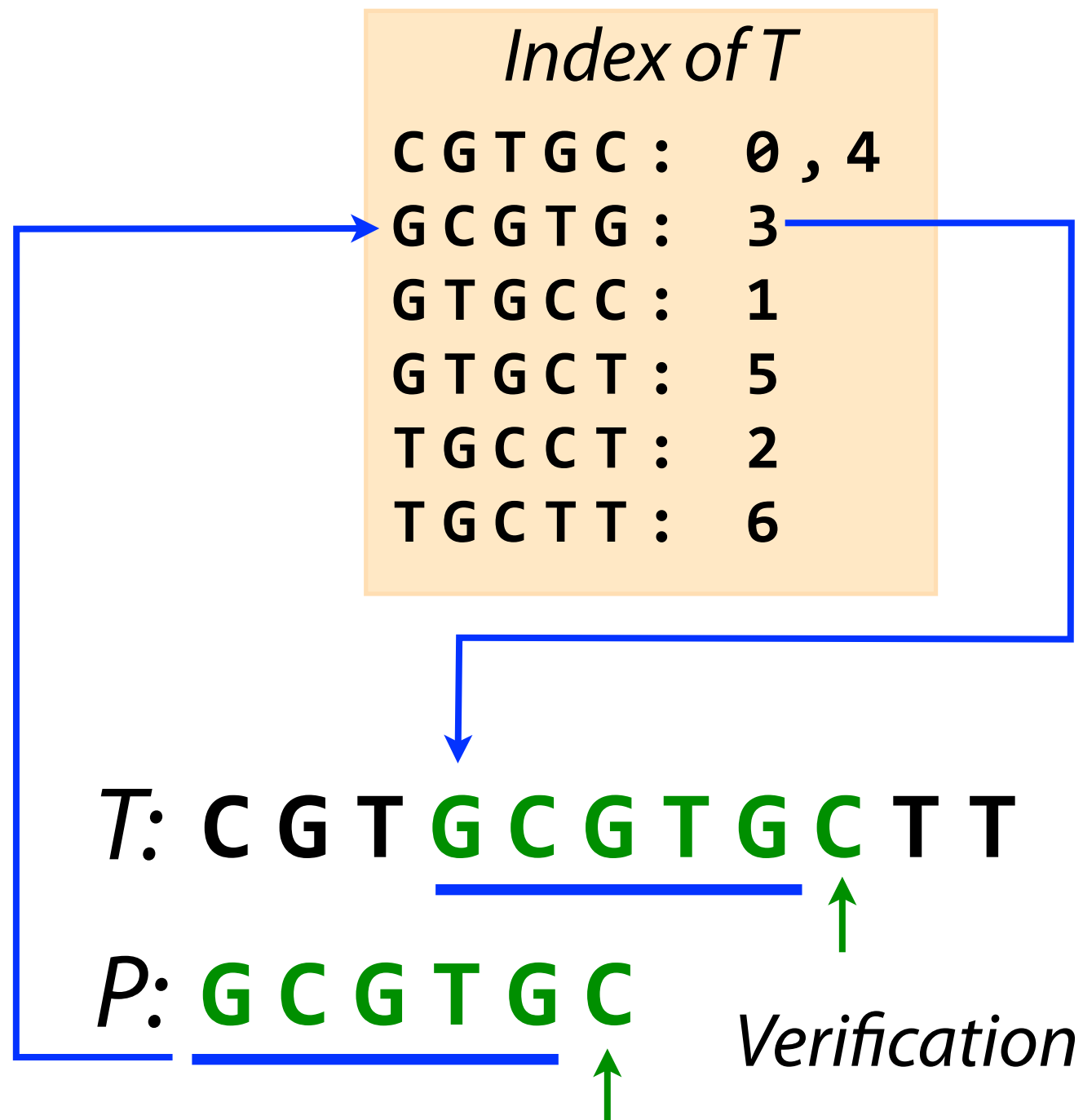
T: C G T G C G T G C T T

P: G C G T G C

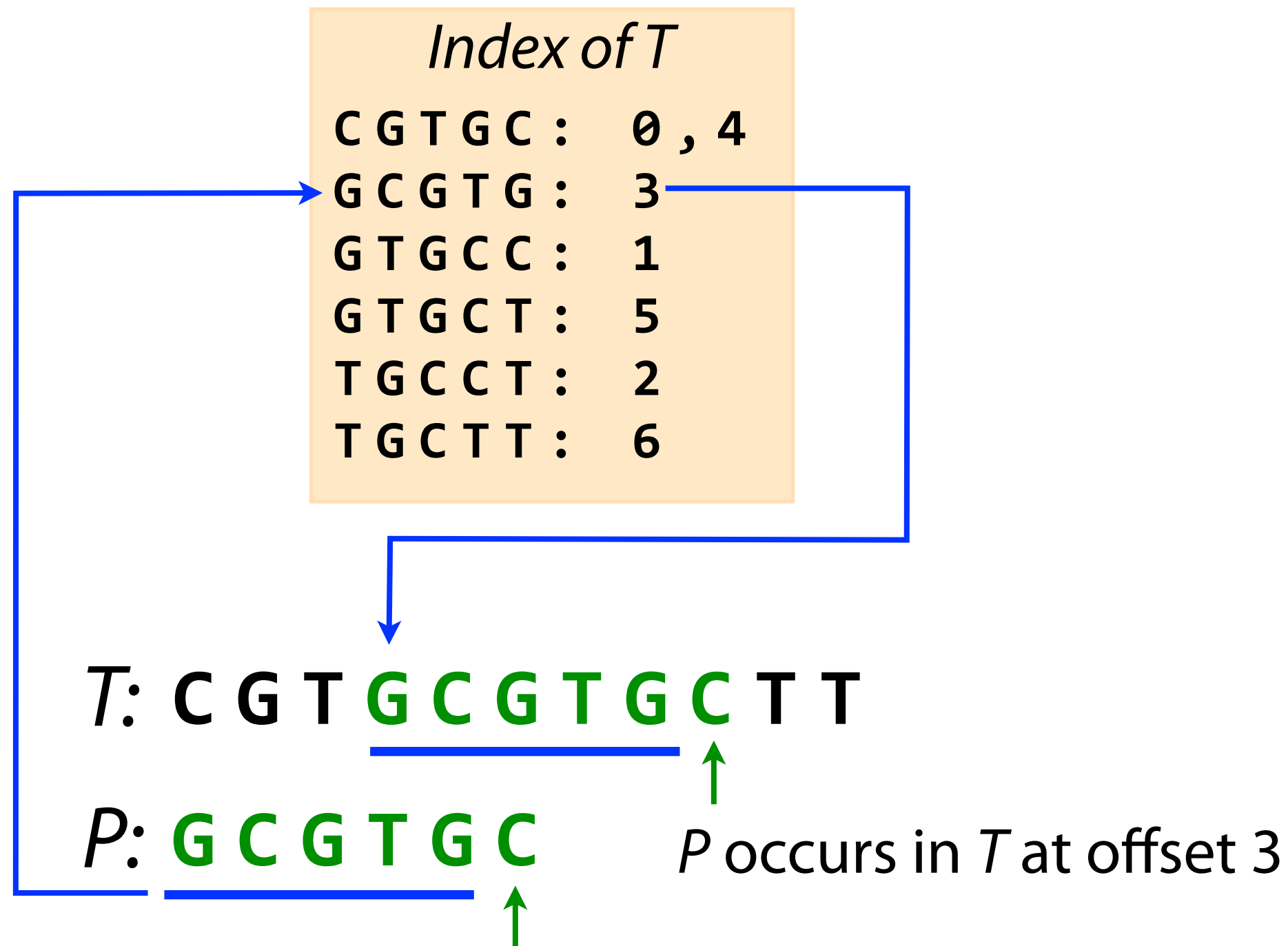
Querying the index



Querying the index



Querying the index



Querying the index

Index of T

C G T G C	:	0 , 4
G C G T G	:	3
G T G C C	:	1
G T G C T	:	5
T G C C T	:	2
T G C T T	:	6

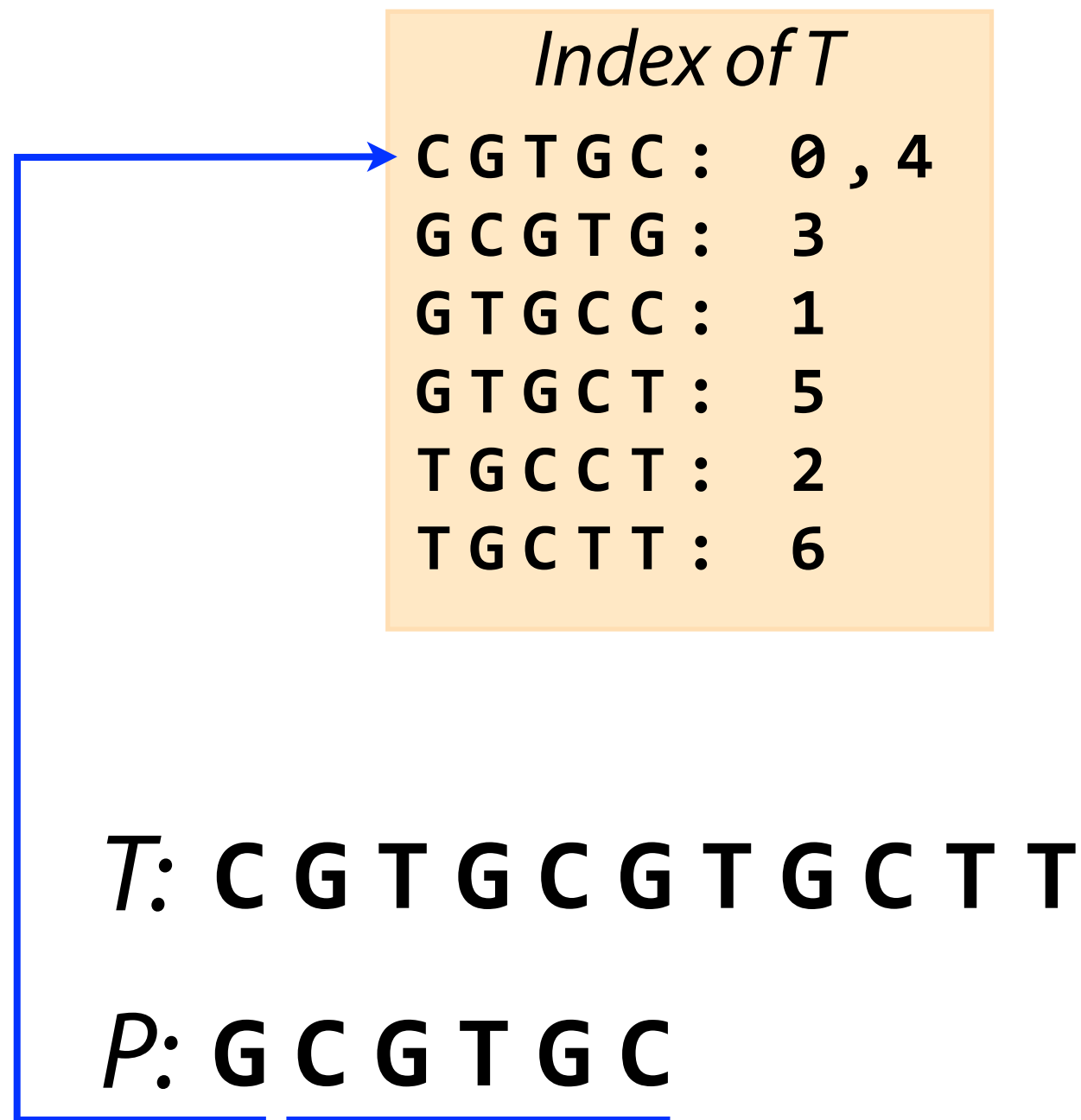
?

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

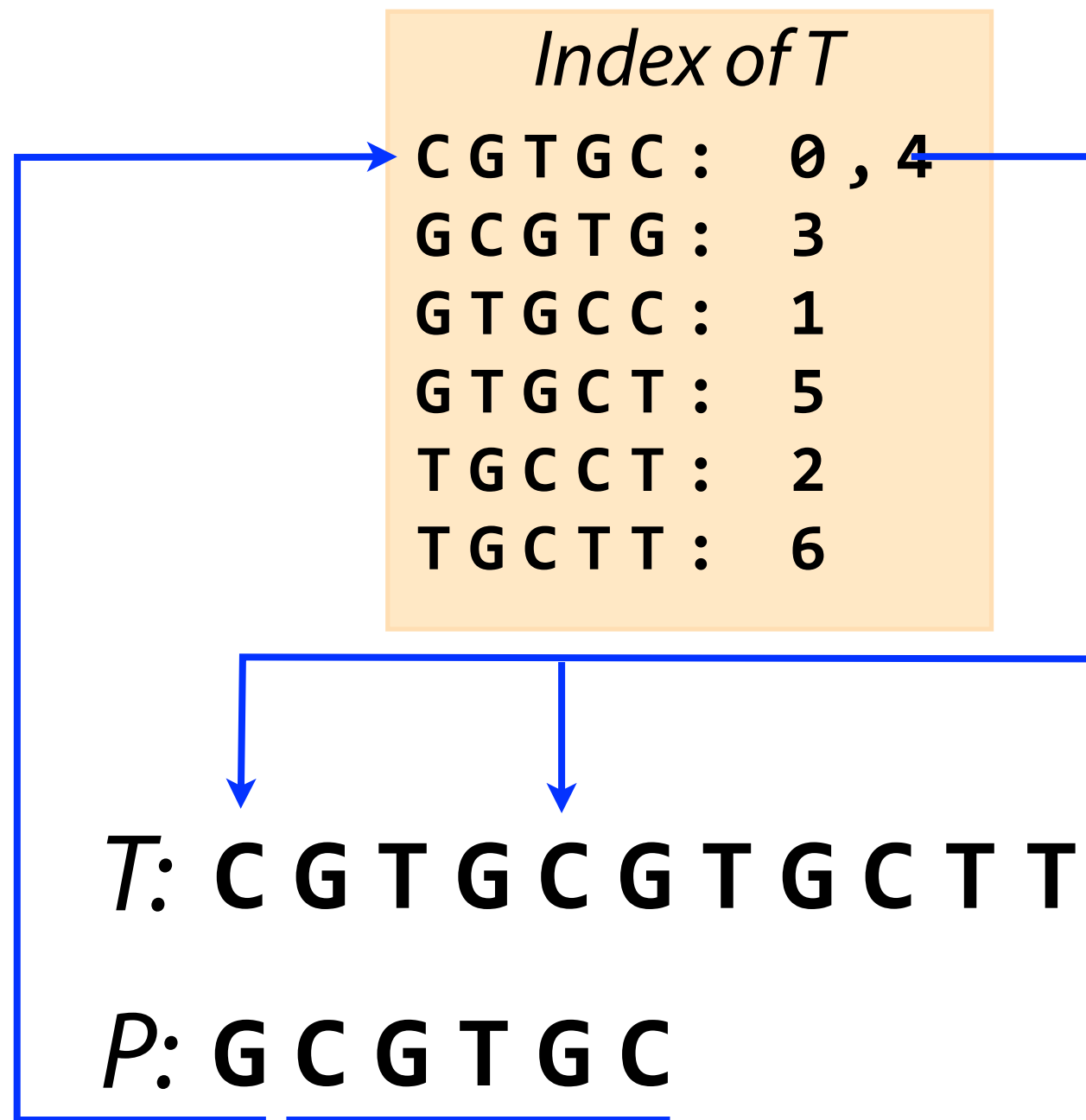
P: G C G T G C



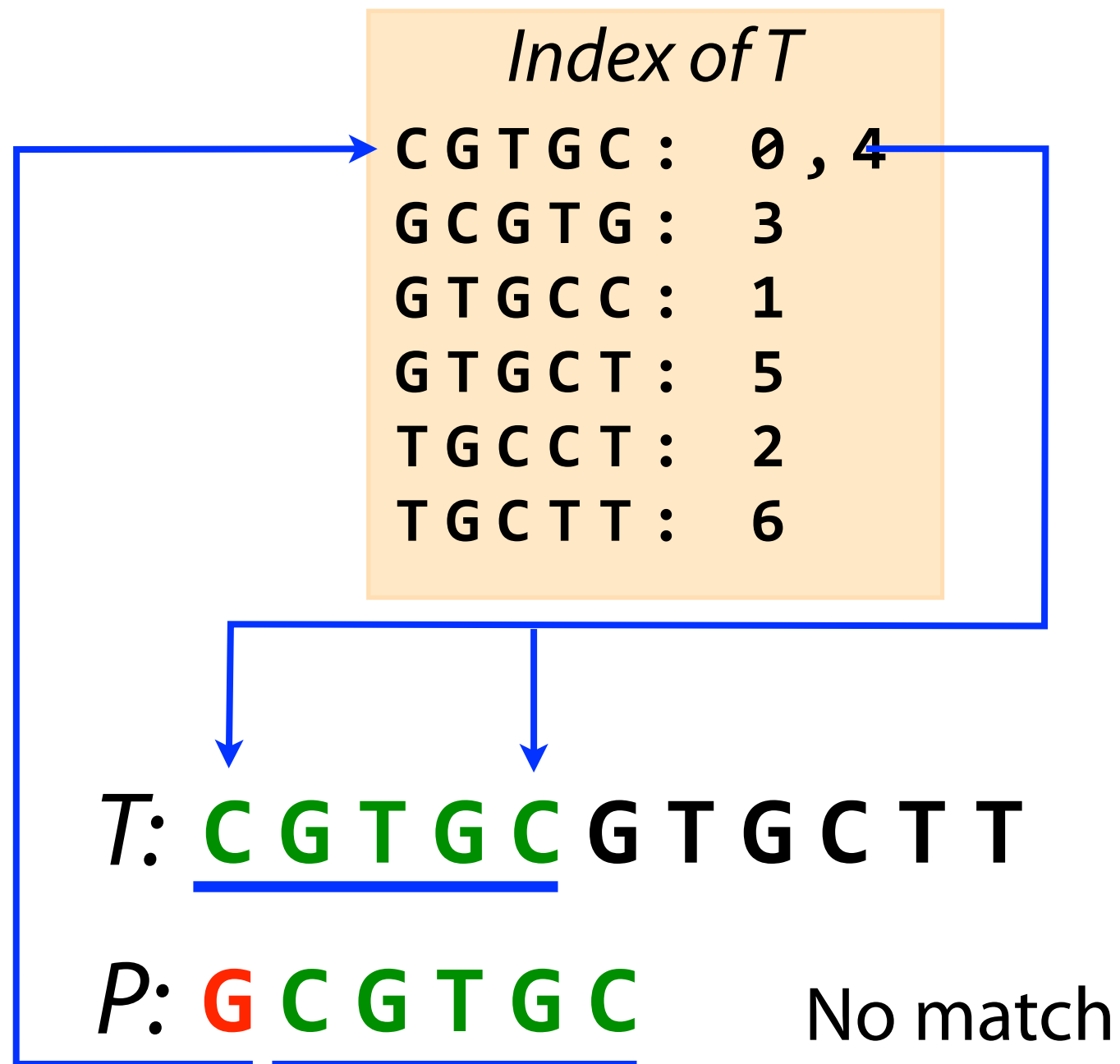
Querying the index



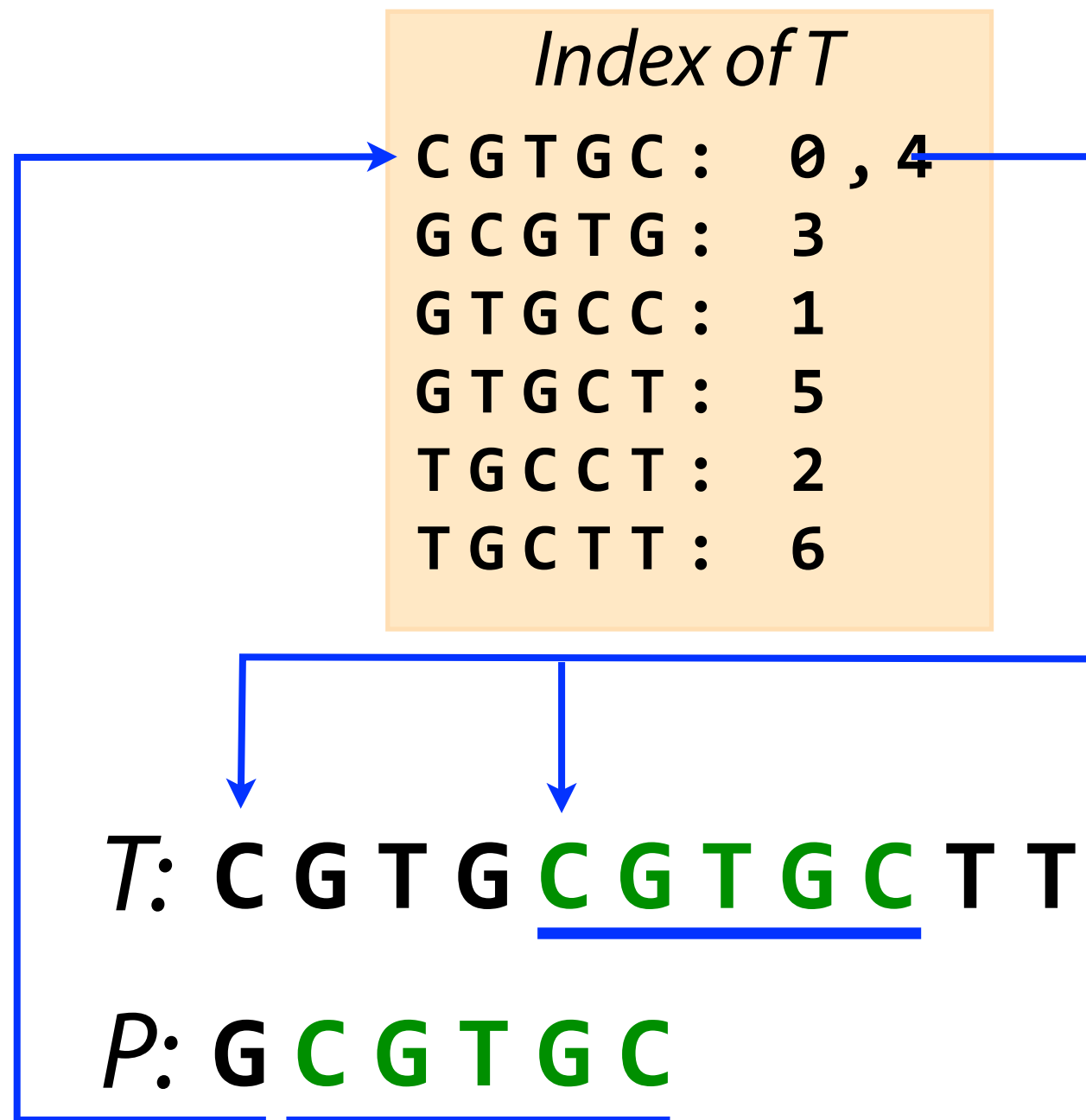
Querying the index



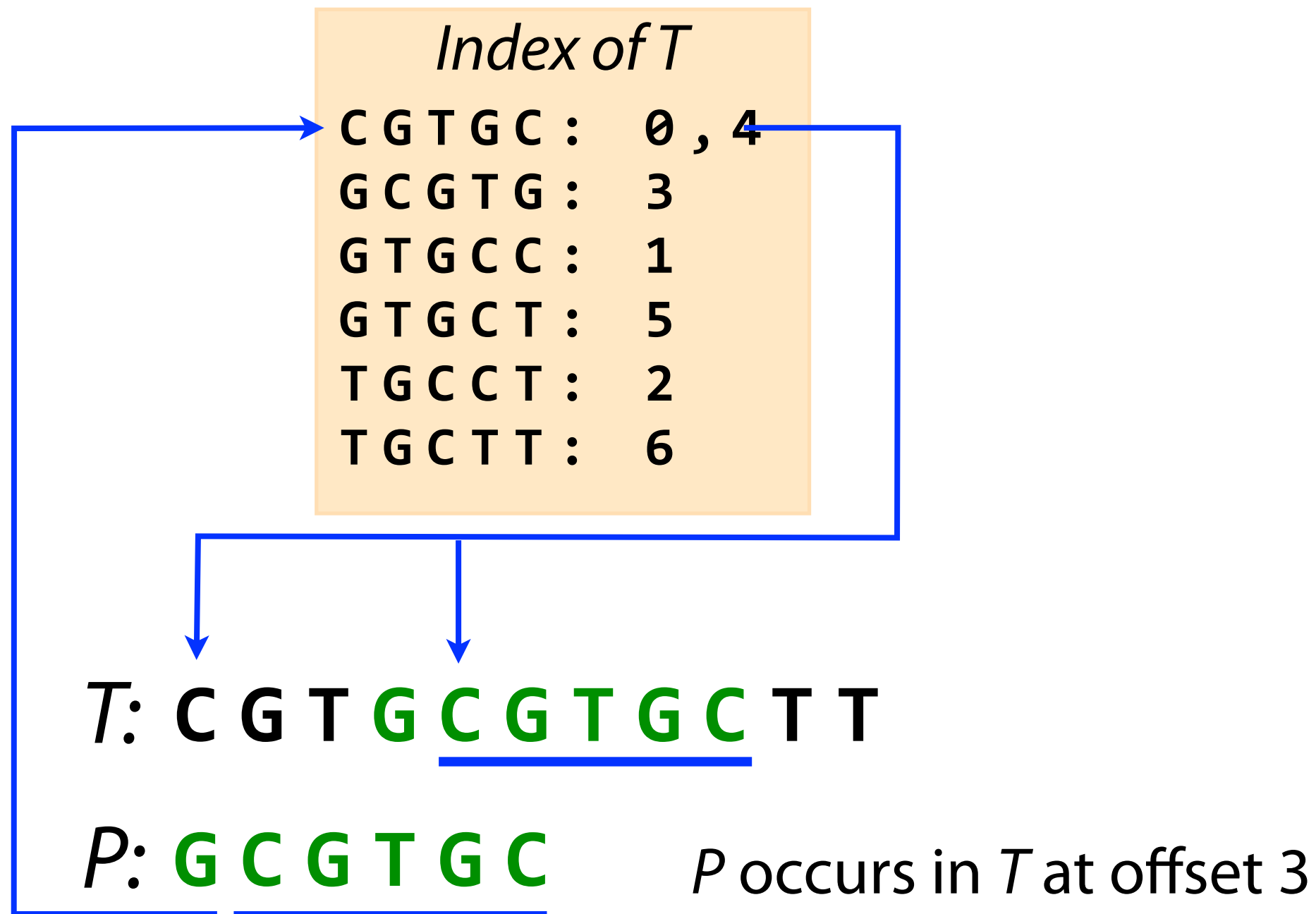
Querying the index



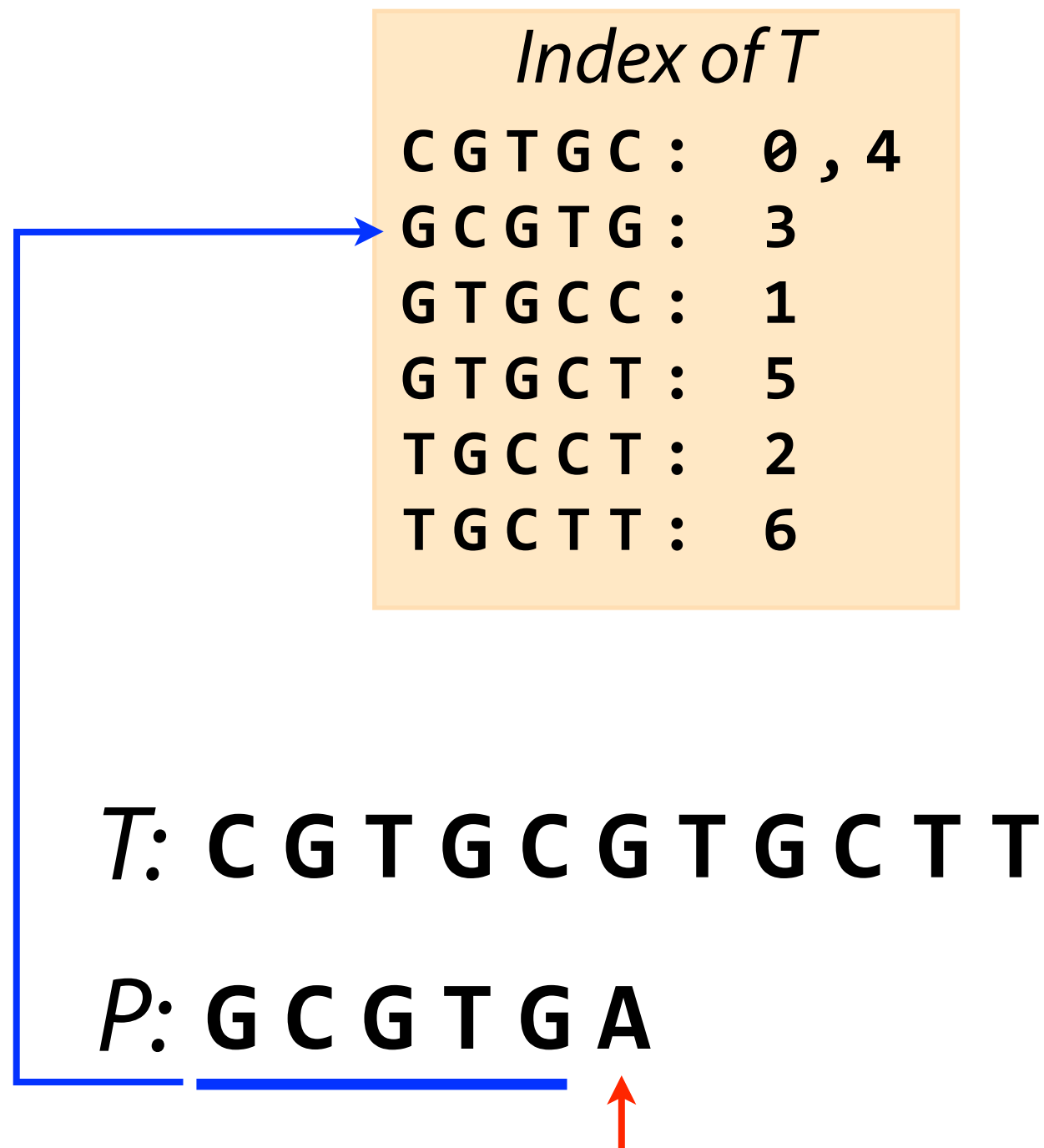
Querying the index



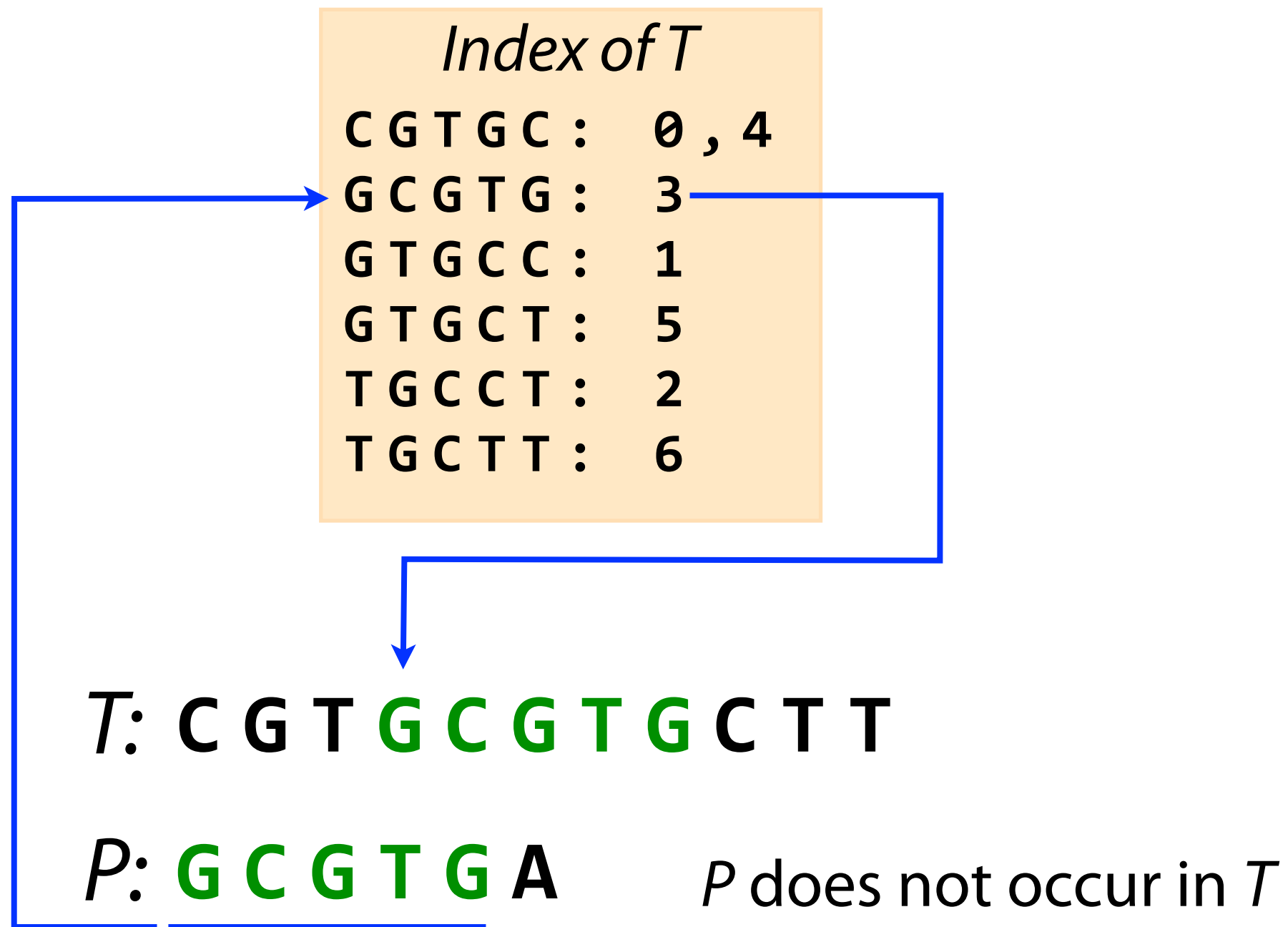
Querying the index



Querying the index



Querying the index



Querying the index

Index of T

C G T G C	:	0 , 4
G C G T G	:	3
G T G C C	:	1
G T G C T	:	5
T G C C T	:	2
T G C T T	:	6

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

***P*: G C G T A C**



Querying the index

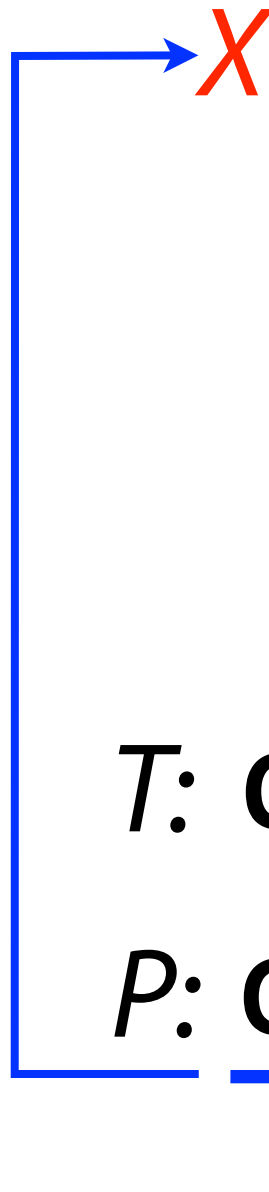
Index of T

C G T G C	:	0 , 4
G C G T G	:	3
G T G C C	:	1
G T G C T	:	5
T G C C T	:	2
T G C T T	:	6

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

P: G C G T A C

P does not occur in *T*



Querying the index

Index of T

C G T G C : 0 , 4

G C G T G : 3

G T G C C : 1

G T G C T : 5

T G C C T : 2

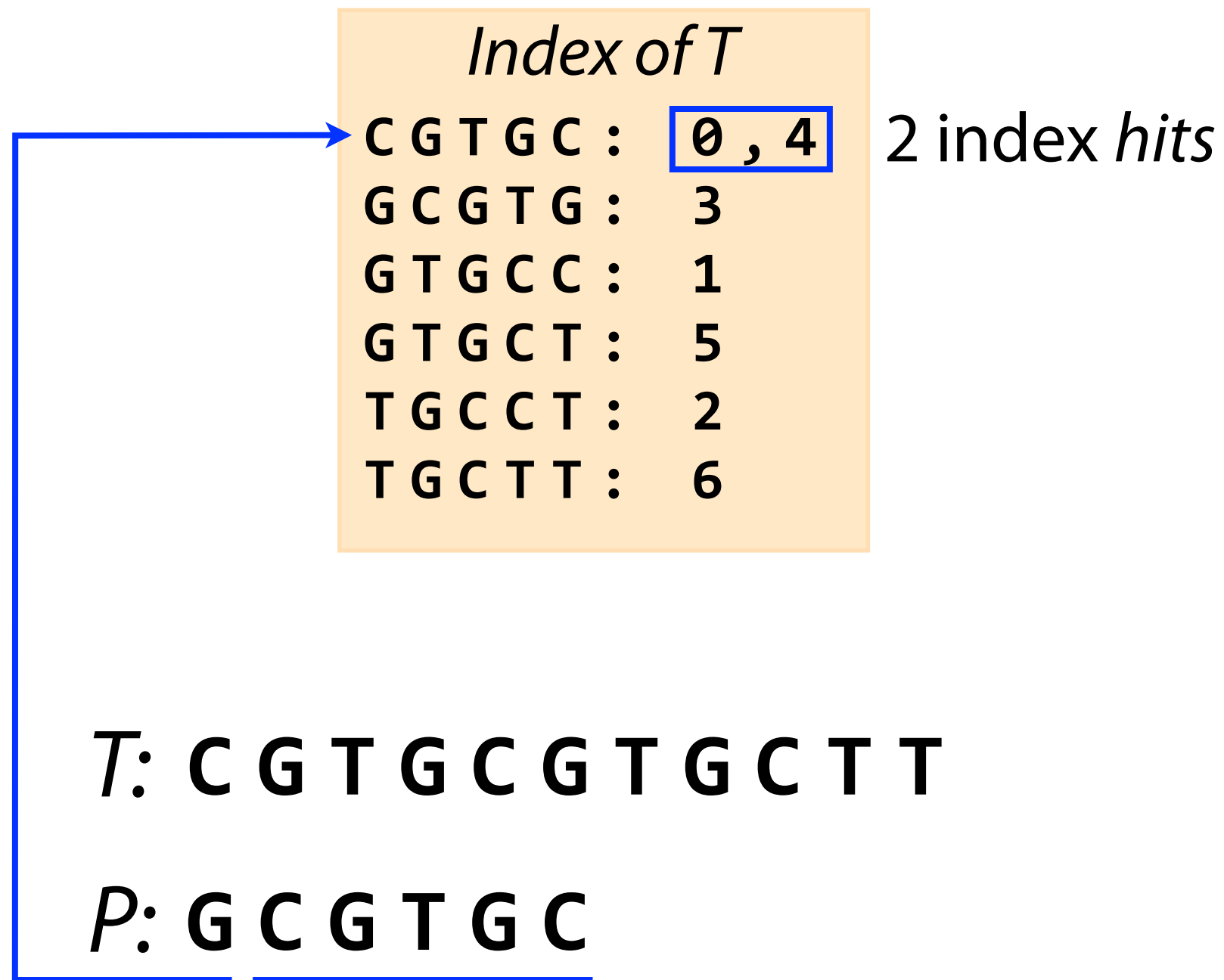
T G C T T : 6

1 index *hit*

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

P: G C G T G C

Querying the index



Data structures

<i>Index of T</i>	
C G T G C :	0 , 4
G C G T G :	3
G T G C C :	1
G T G C T :	5
T G C C T :	2
T G C T T :	6

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

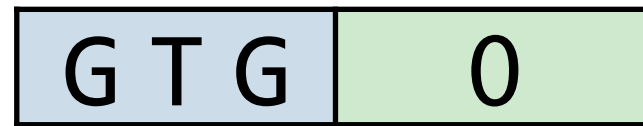
What data structures allow us to represent and query a multimap?

Data structures

First idea: add key-value pairs to an array & sort the array

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

Data structures



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

Data structures

G T G	0
T G C	1

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

Data structures

G T G	0
T G C	1
G C G	2

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


Data structures

G T G	0
T G C	1
G C G	2
C G T	3
G T G	4
T G T	5
G T G	6
T G G	7
G G G	8
G G G	9
G G G	10

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

Data structures

Alphabetical
by k-mer



C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

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

Binary search


C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

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

P: G C G T G G

Binary search

G C G < **G T G**



C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

**1st
bisection**

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

P: G C G T G G

Binary search

G C G < **G G G**



C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

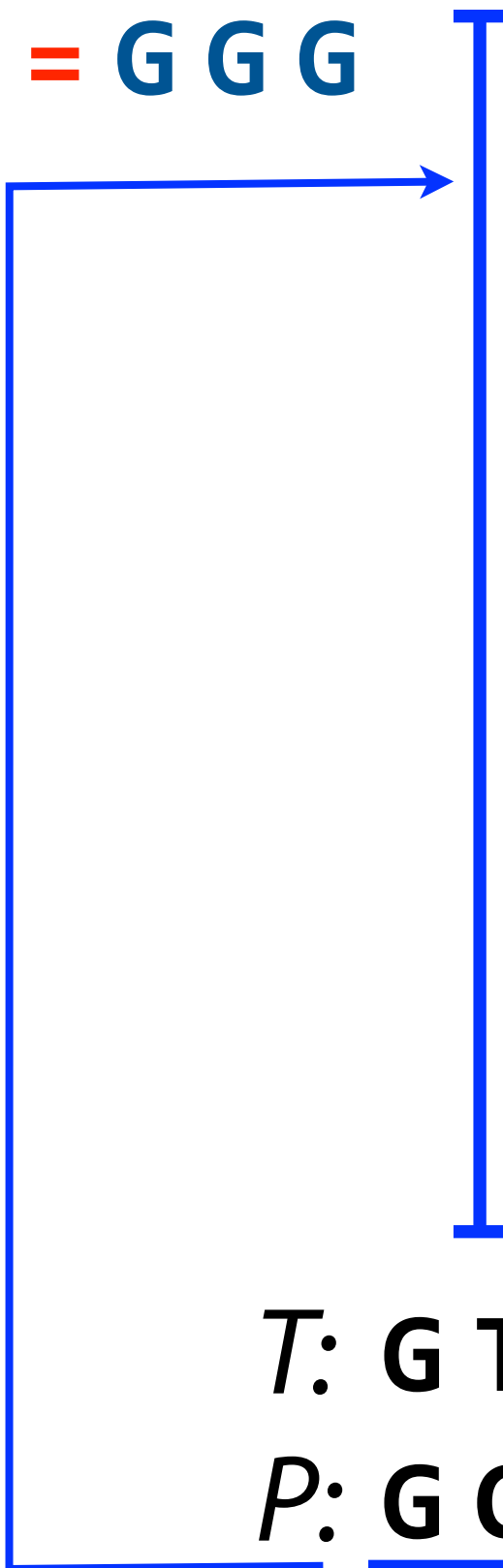
2nd bisection

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

P: G C G T G G

Binary search

G C G = G G G



C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5



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

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

Binary search


C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

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

P: G C G T G G

Binary search

T G G > **G T G**



C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

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

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

Binary search

After 1st
bisection

T G G > **T G C**

C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

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

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

Binary search

After 2nd
bisection

C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

T G G = T G G

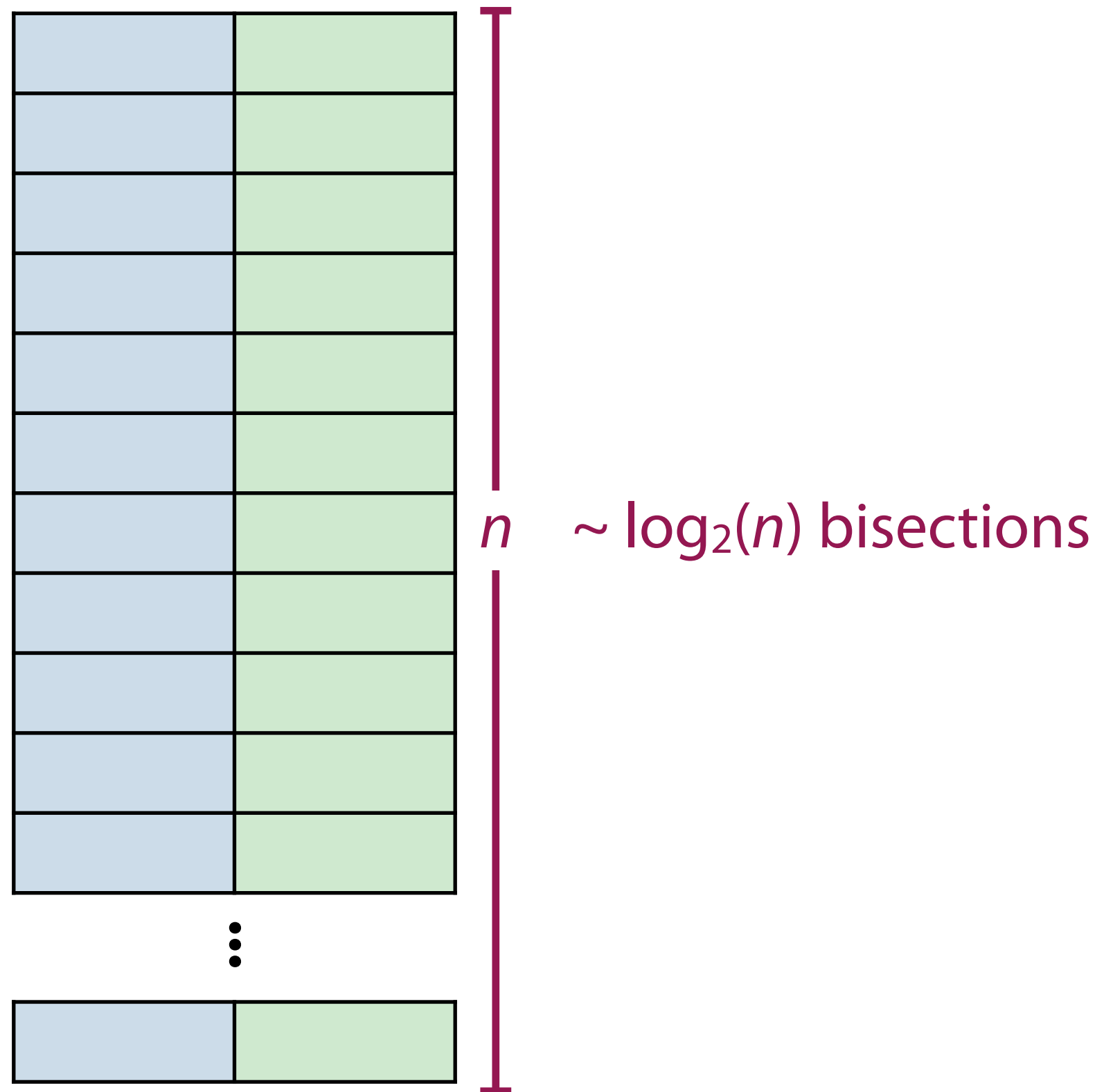
I

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

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

Binary search


About how many bisections in the worst case, as a function of n ?



bisect.bisect_left(a, x): Leftmost offset
where **x** can be inserted into **a** to maintain order

```
>>> a = [1, 3, 3, 6, 8, 8, 9, 10]  
>>> import bisect  
>>> bisect.bisect_left(a, 2)
```


bisect.bisect_left(a, x): Leftmost offset
where **x** can be inserted into **a** to maintain order



```
>>> a = [1, 3, 3, 6, 8, 8, 9, 10]
>>> import bisect
>>> bisect.bisect_left(a, 2)
```


1

`bisect.bisect_left(a, x)`: Leftmost offset
where **x** can be inserted into **a** to maintain order




```
>>> a = [1, 3, 3, 6, 8, 8, 9, 10]
>>> import bisect
>>> bisect.bisect_left(a, 2)
1
>>> bisect.bisect_left(a, 4)
```


bisect.bisect_left(a, x): Leftmost offset
where **x** can be inserted into **a** to maintain order




```
>>> a = [1, 3, 3, 6, 8, 8, 9, 10]
>>> import bisect
>>> bisect.bisect_left(a, 2)
1
>>> bisect.bisect_left(a, 4)
3
```

bisect.bisect_left(a, x): Leftmost offset
where **x** can be inserted into **a** to maintain order



```
>>> a = [1, 3, 3, 6, 8, 8, 9, 10]
>>> import bisect
>>> bisect.bisect_left(a, 2)
1
>>> bisect.bisect_left(a, 4)
3
>>> bisect.bisect_left(a, 8)
```

bisect.bisect_left(a, x): Leftmost offset where **x** can be inserted into **a** to maintain order



```
>>> a = [1, 3, 3, 6, 8, 8, 9, 10]
>>> import bisect
>>> bisect.bisect_left(a, 2)
1
>>> bisect.bisect_left(a, 4)
3
>>> bisect.bisect_left(a, 8)
4
```

a must already be sorted, or answer will be nonsense

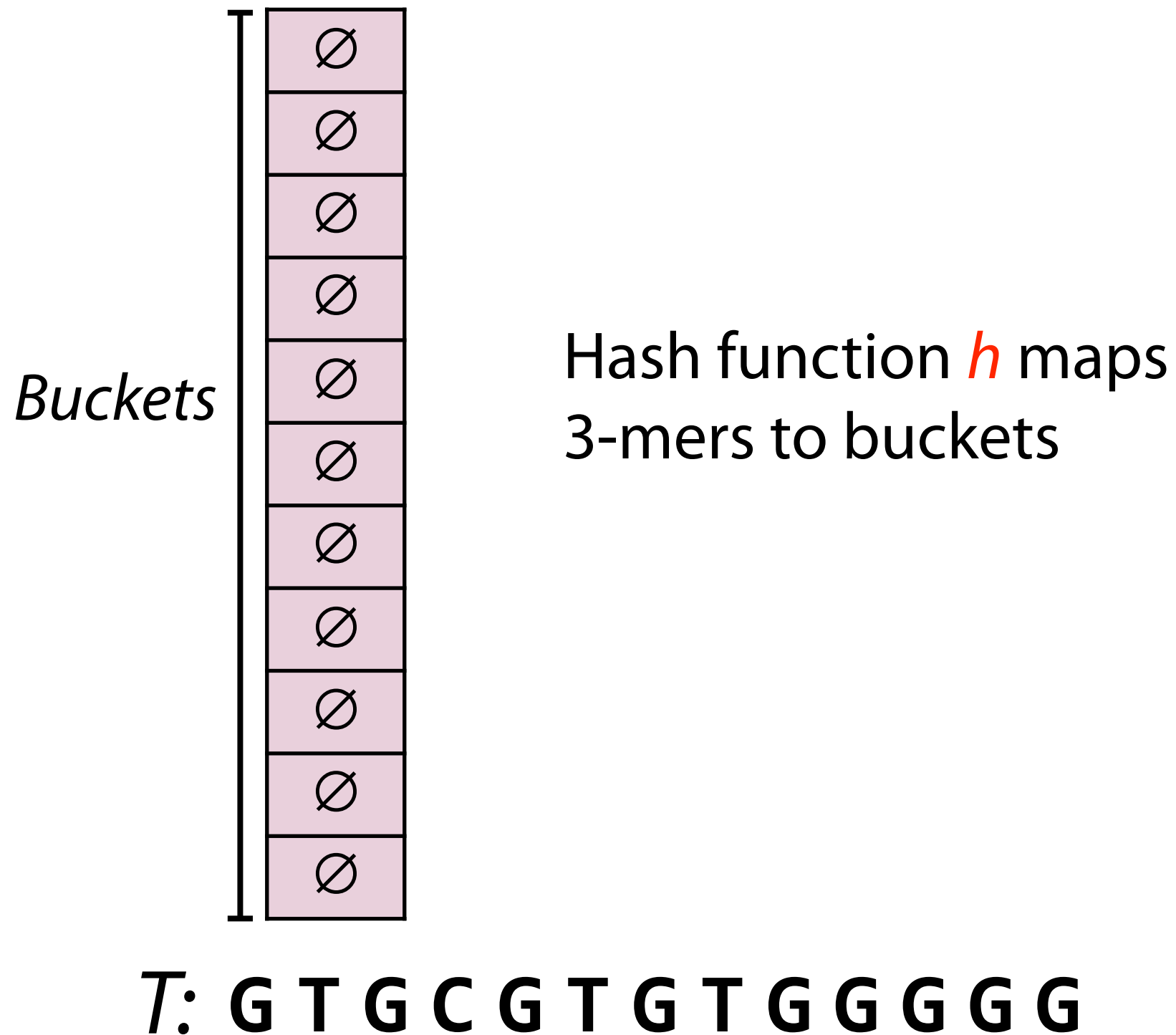
bisect_left(index, 'GTG')

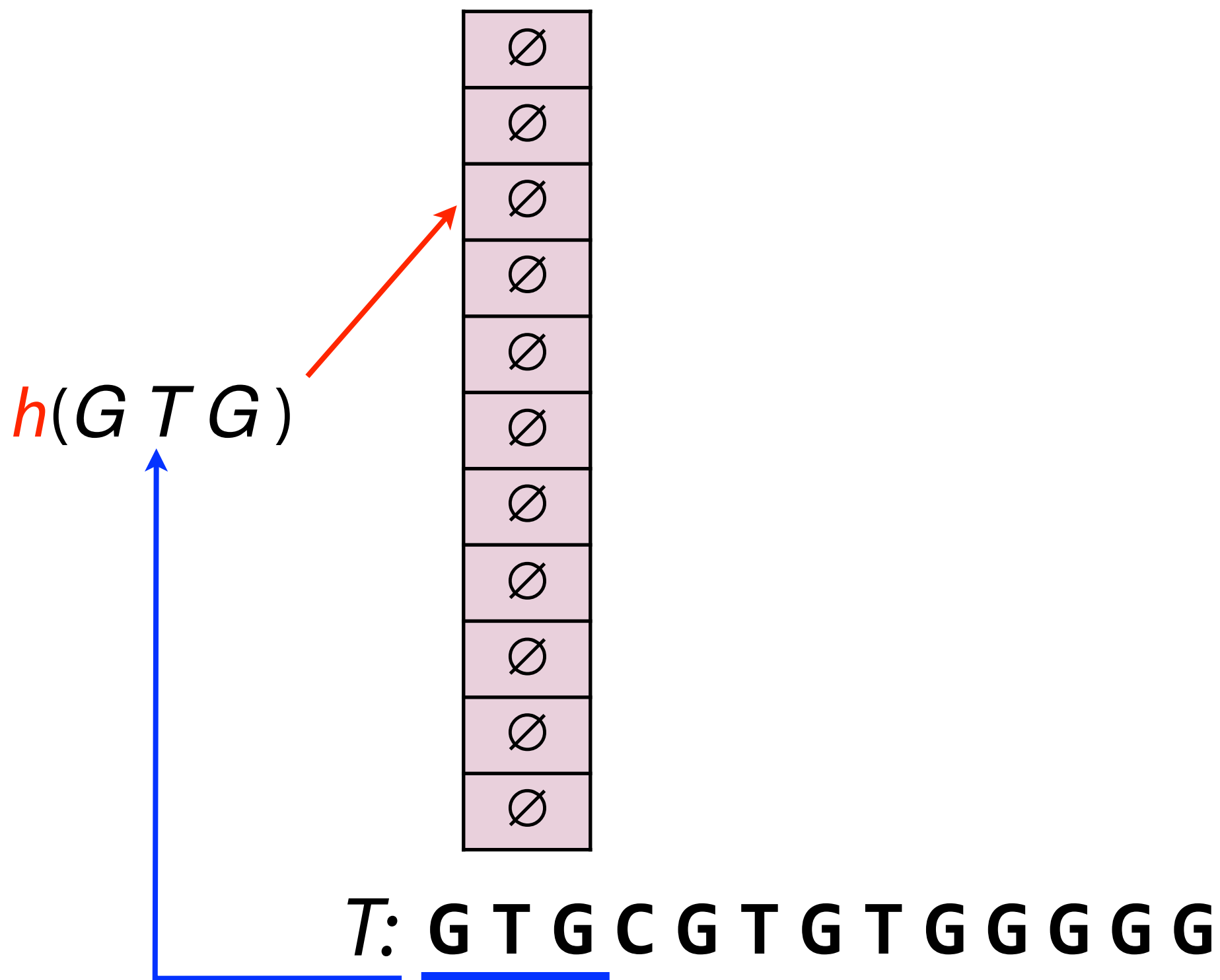
C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

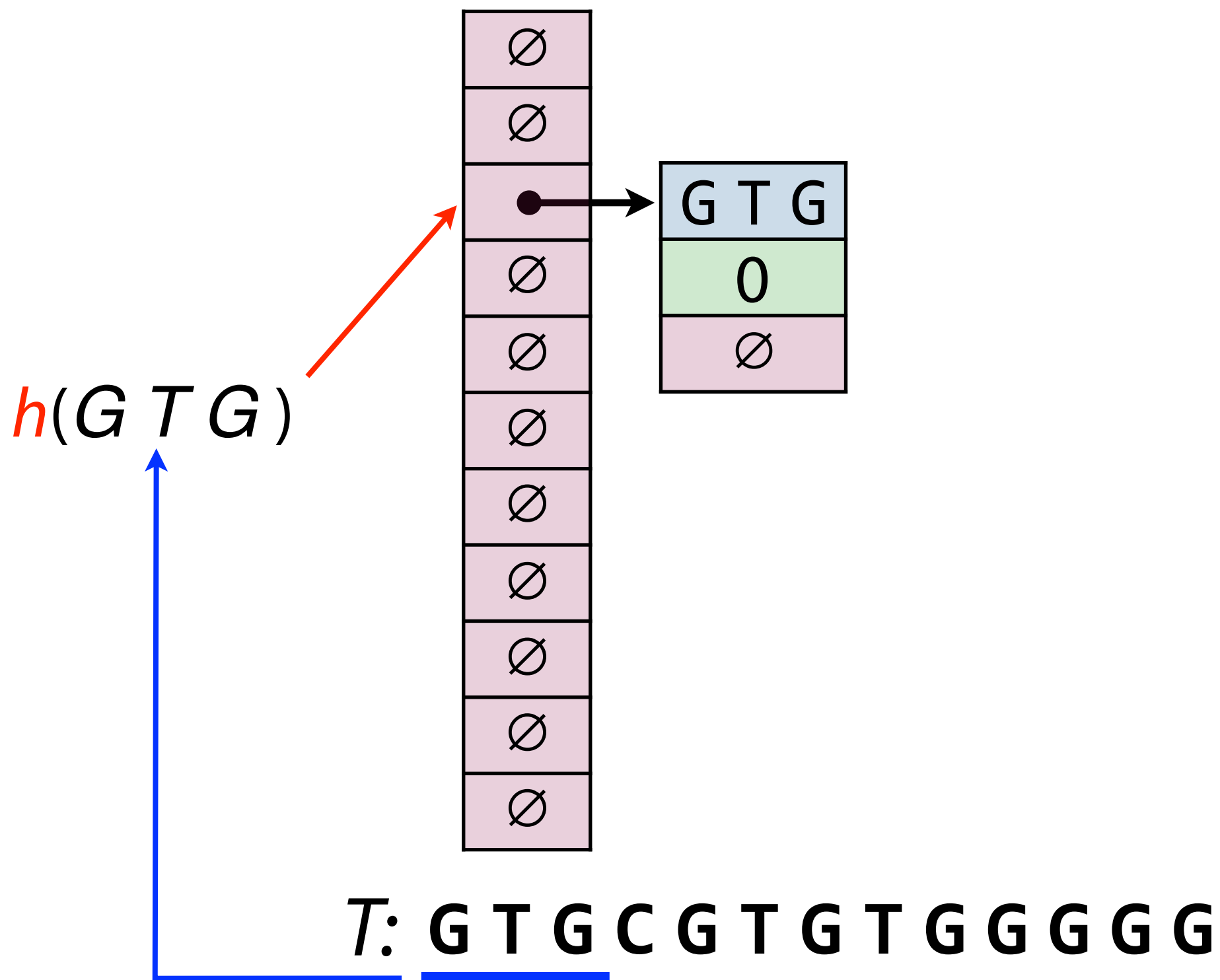
T: G T G C G T G G G G G

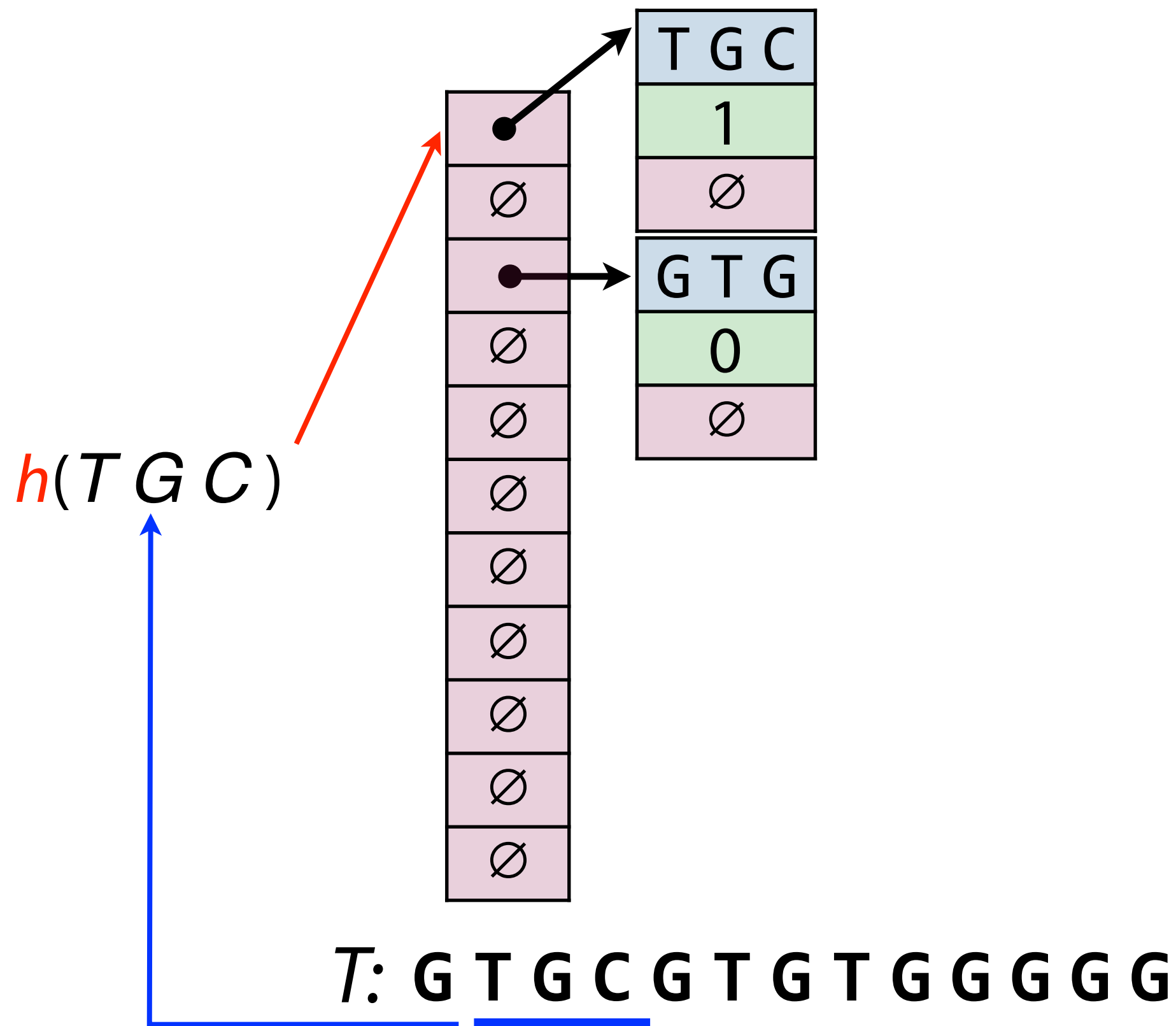
P: G C **G T G** G

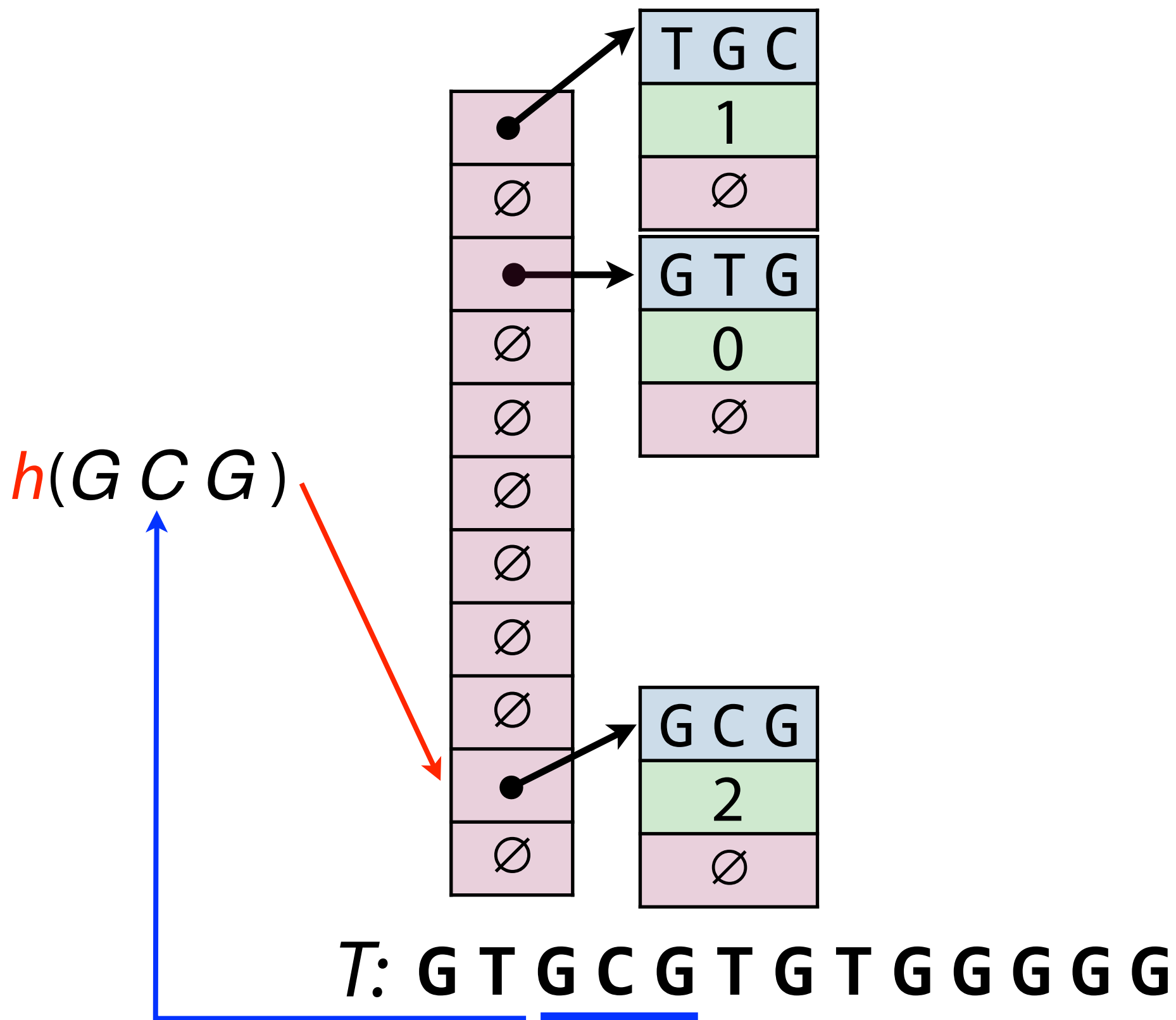
Hash table as multimap

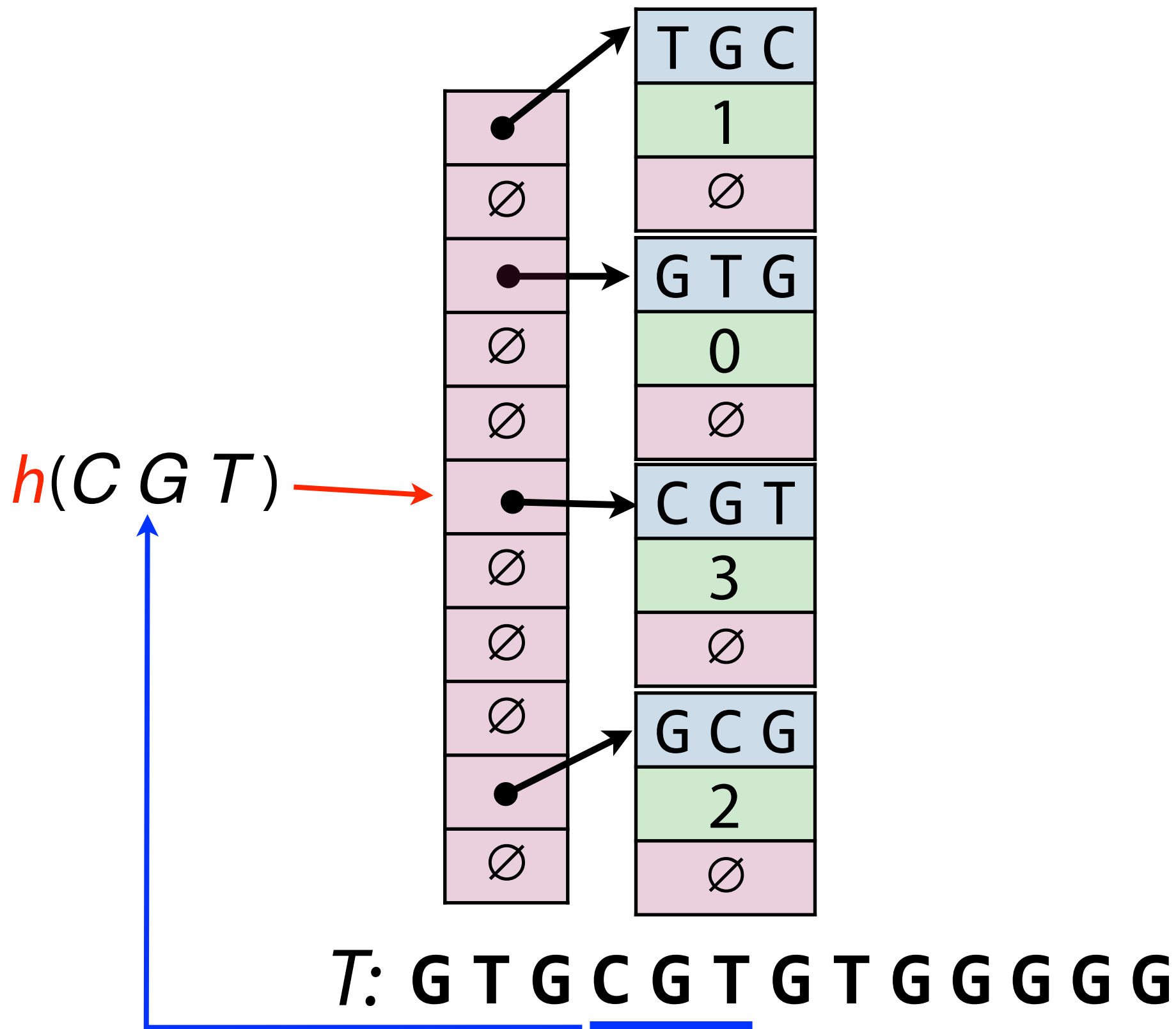


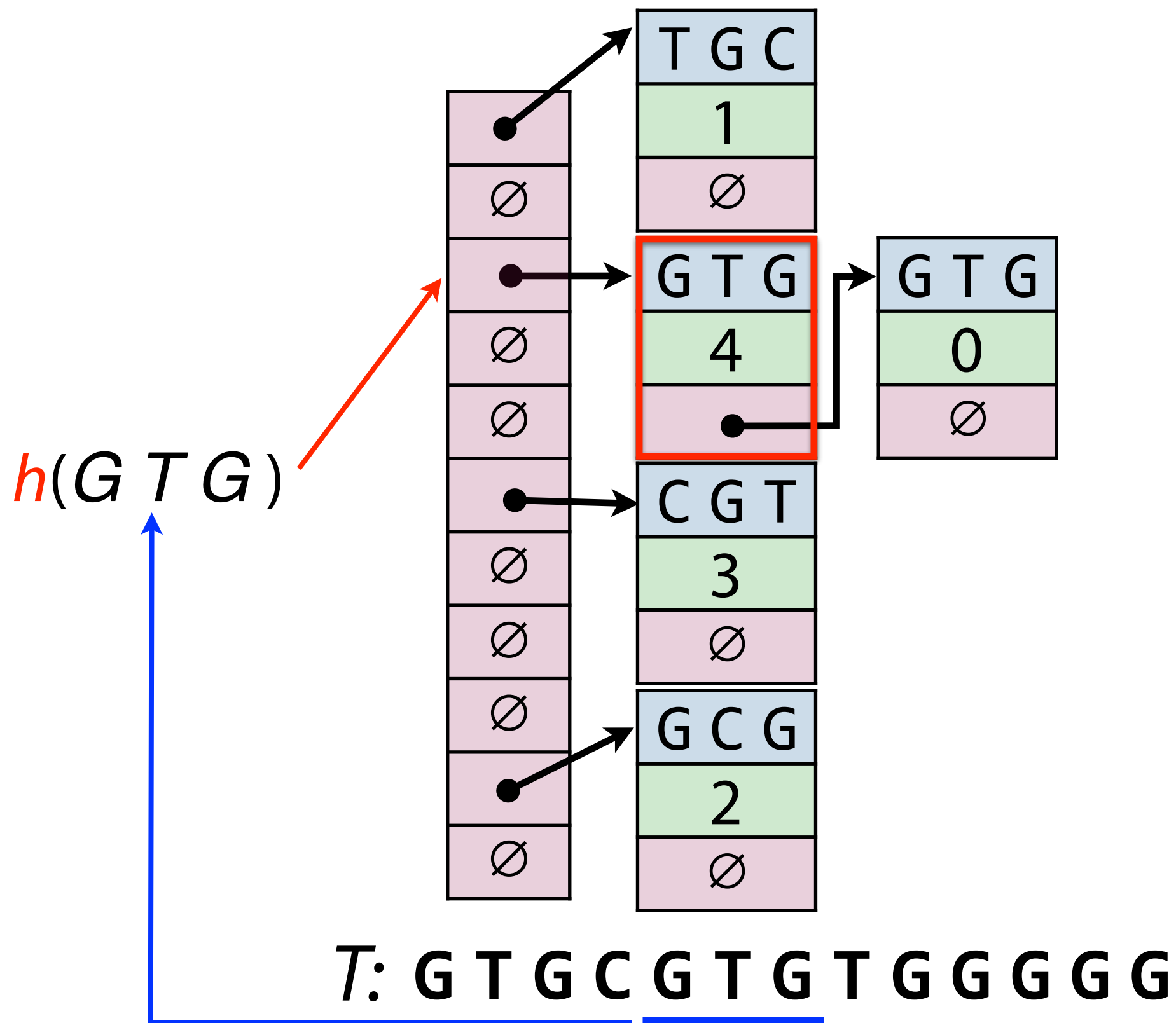


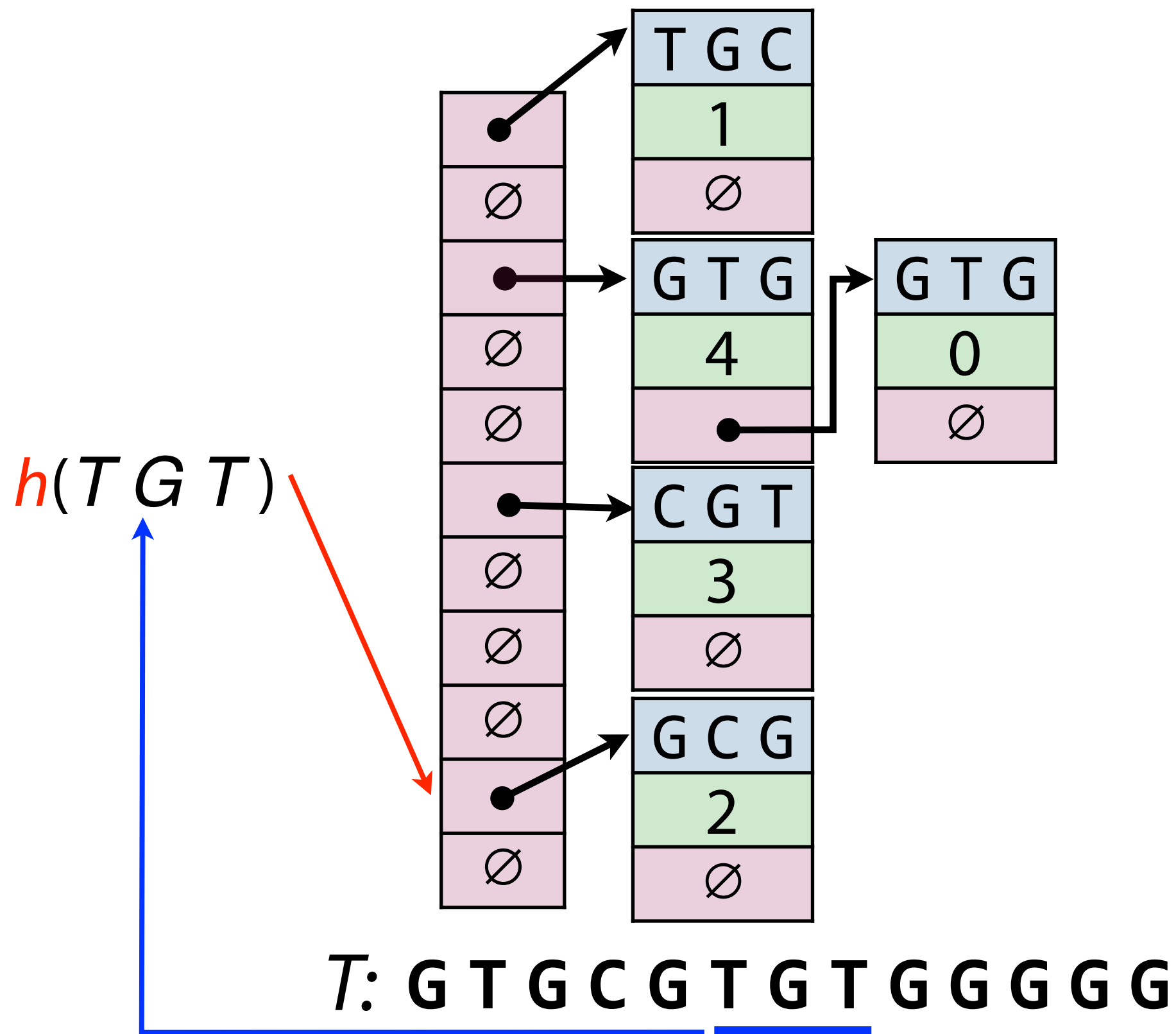


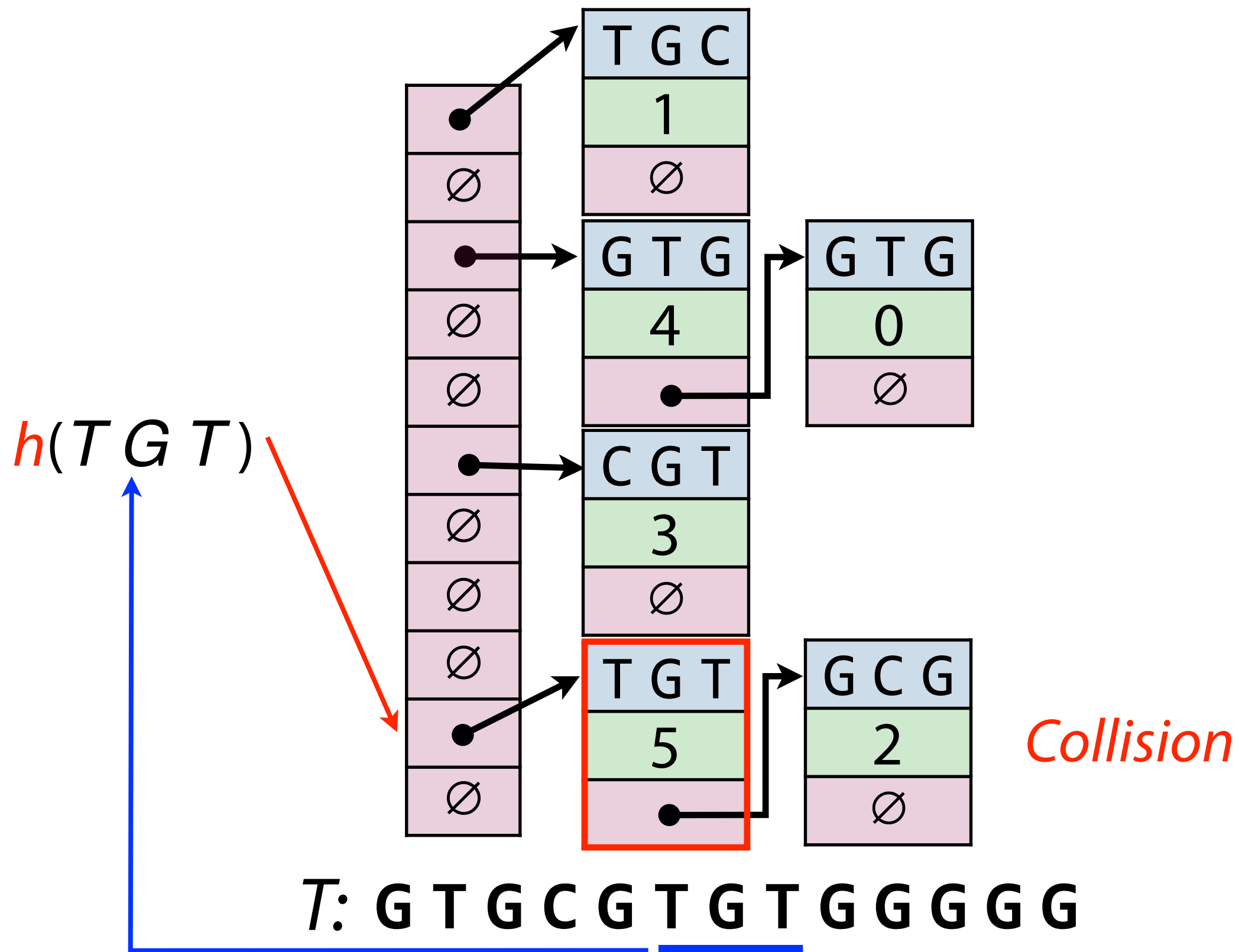


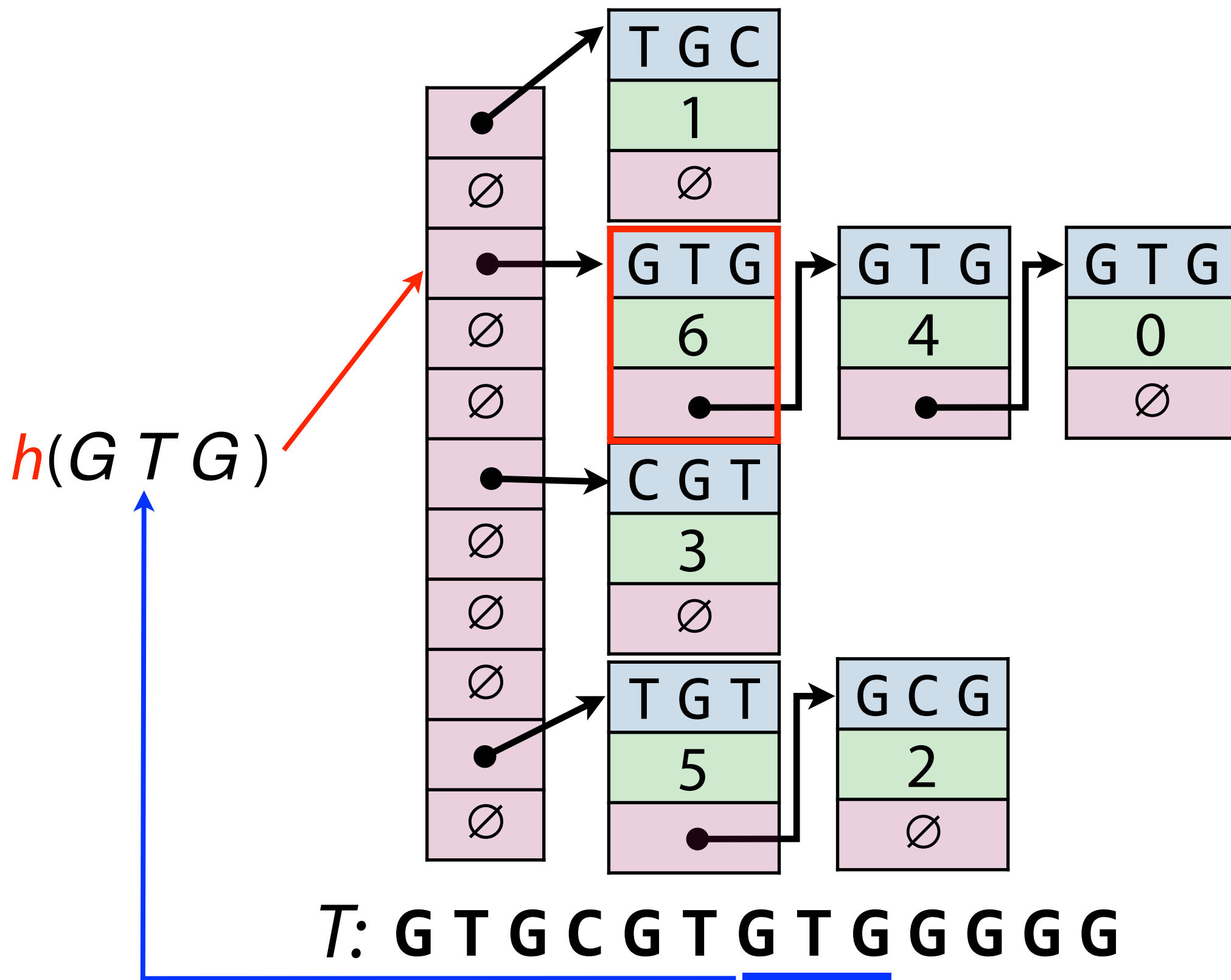


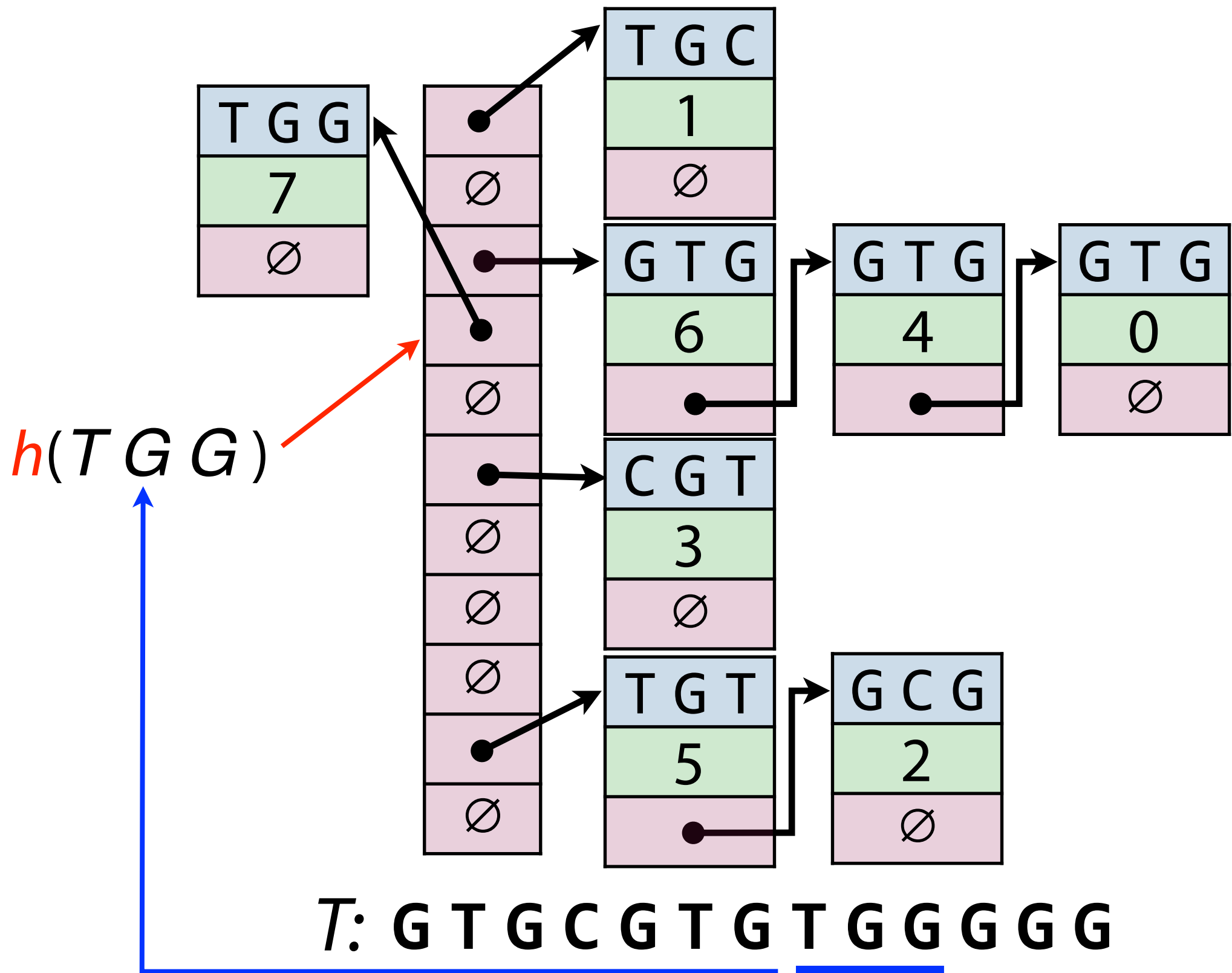


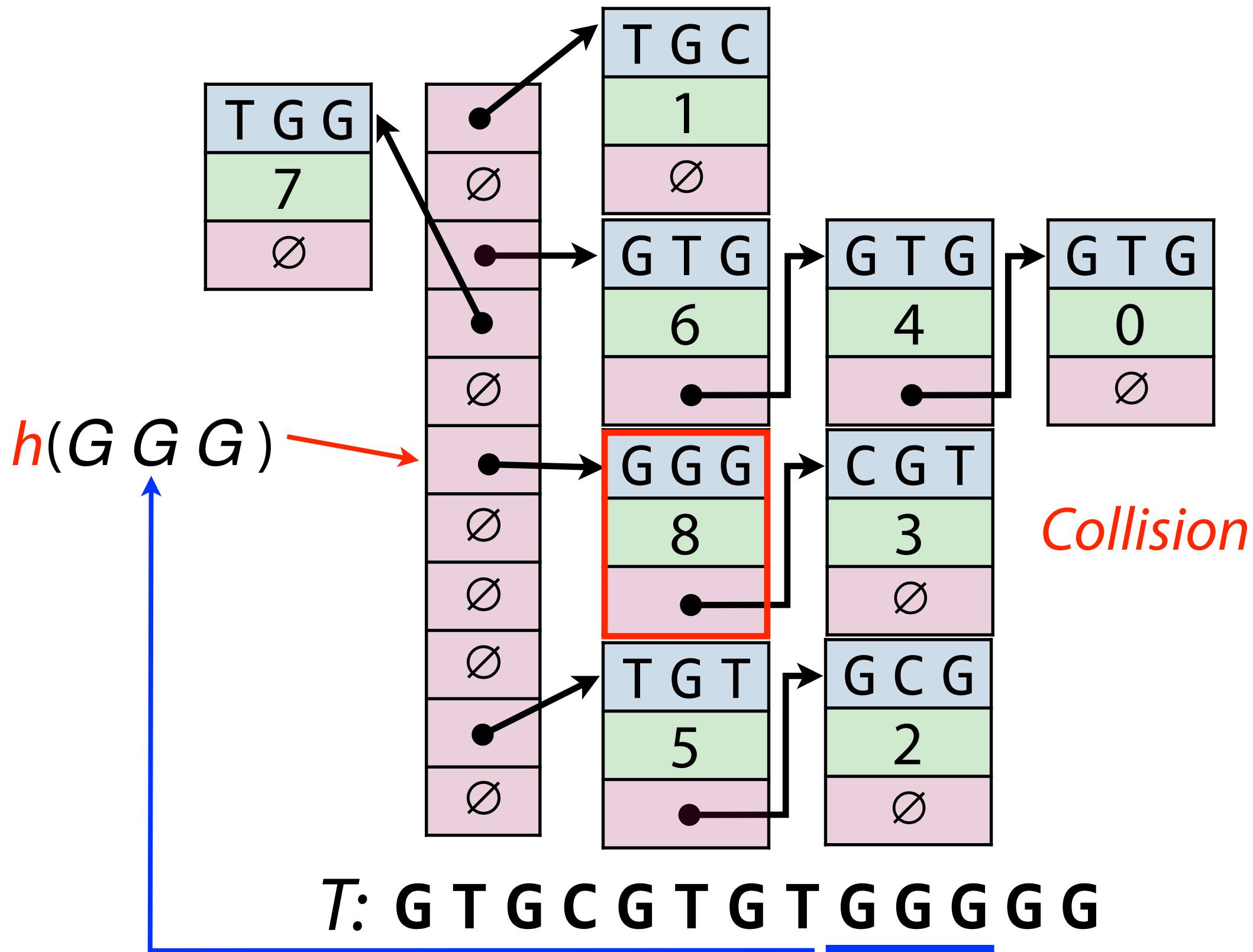


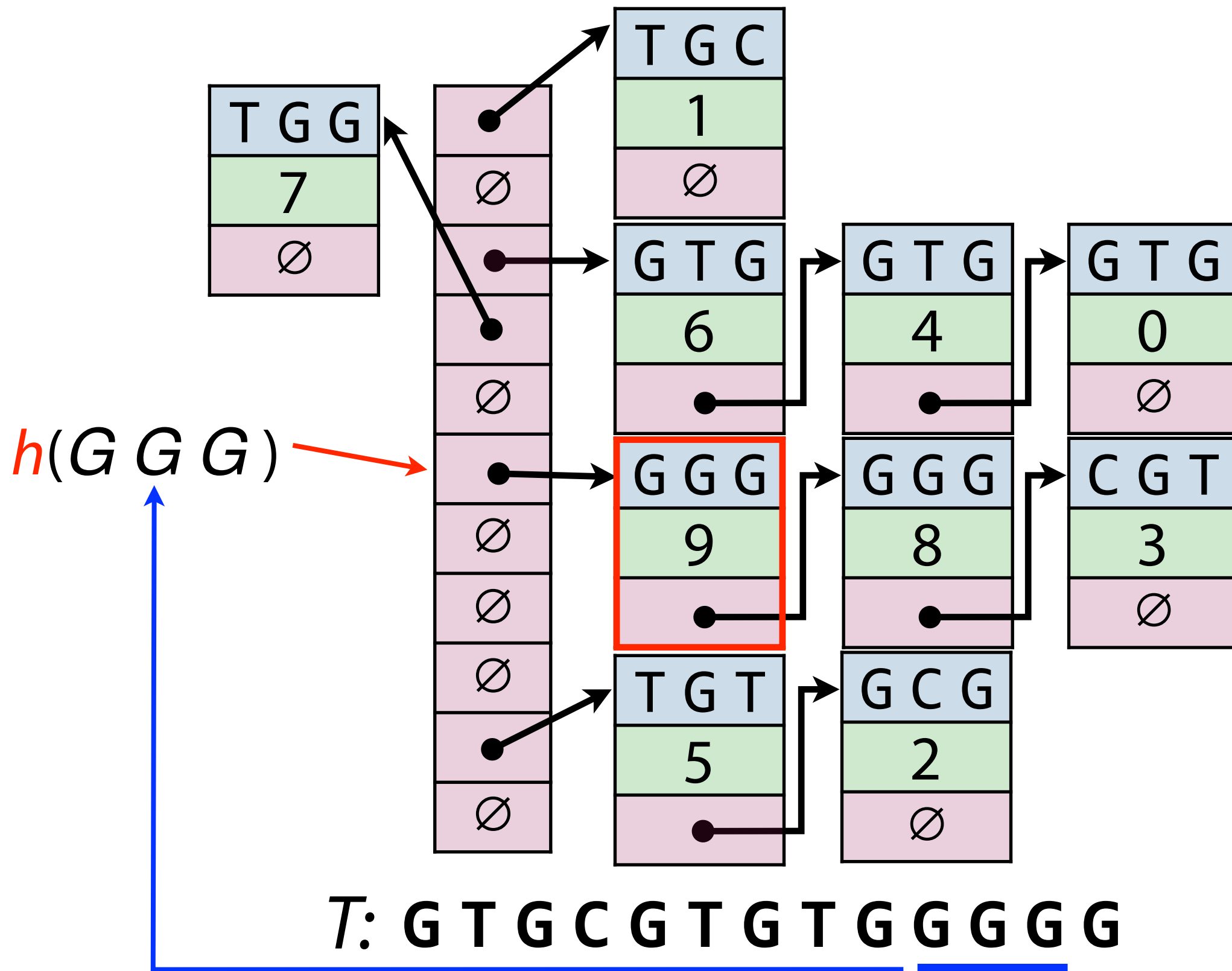


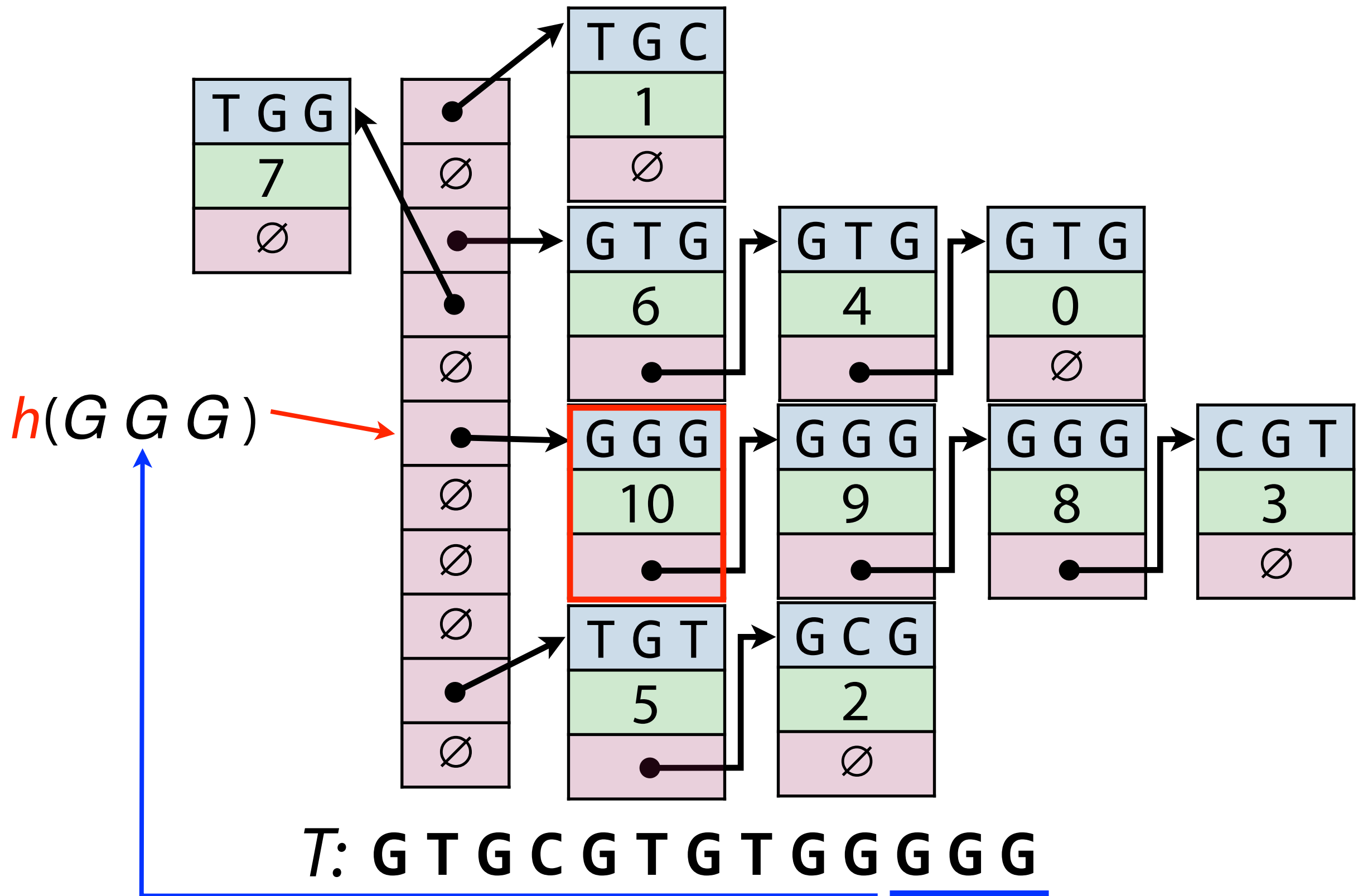


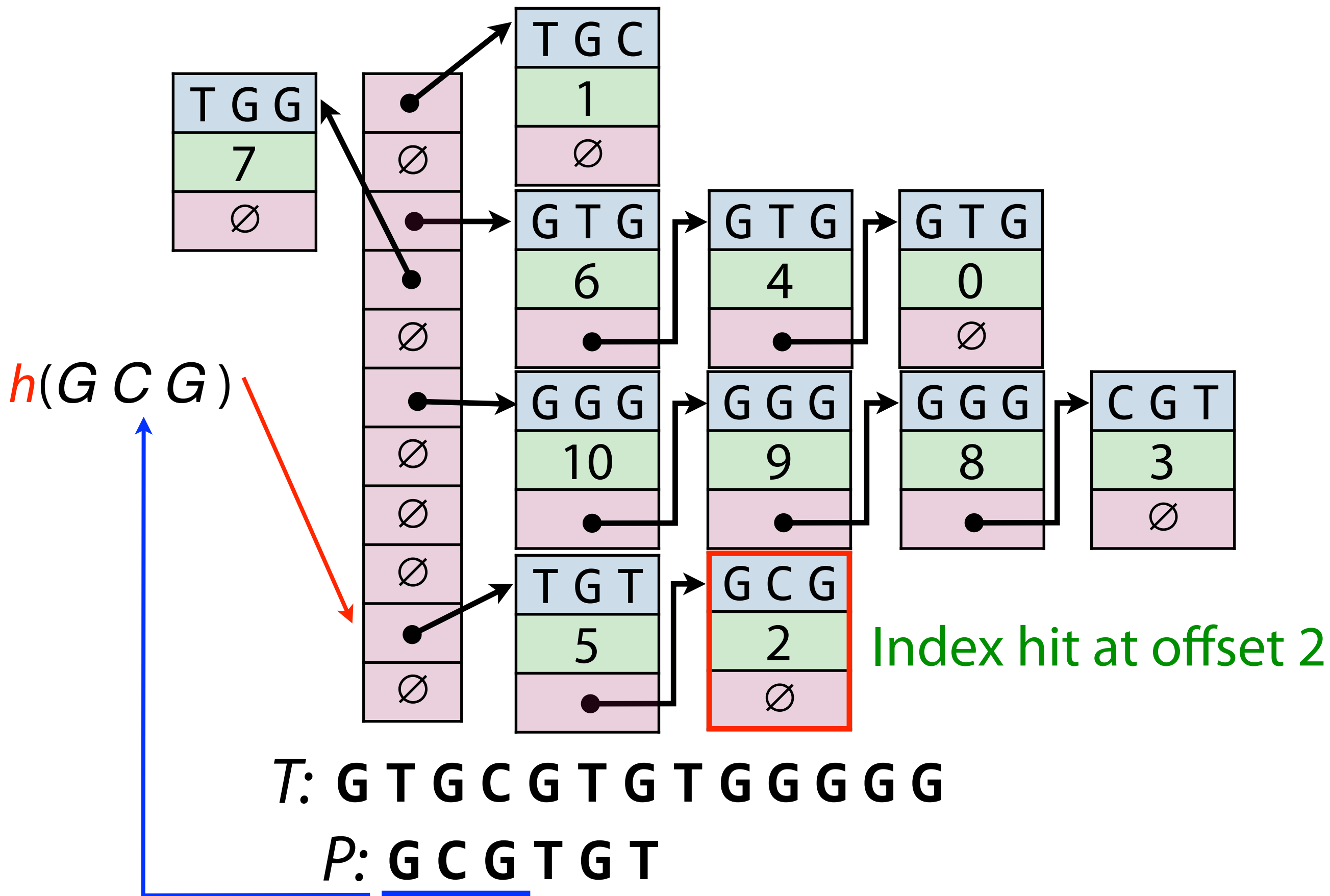












Python dictionary

```
>>> 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]
```

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

Index of T

C G T G C : 0 , 4

G C G T G : 3

G T G C C : 1

G T G C T : 5

T G C C T : 2

T G C T T : 6

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

Index of T

C G T G C : 0 , 4

G C G T G : ~~3~~

G T G C C : ~~1~~

G T G C T : ~~5~~

T G C C T : 2

T G C T T : 6

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

Index of T

C G T G C : 0 , 4

T G C C T : 2

T G C T T : 6

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

Index of T

C G T G C : 0 , 4

T G C C T : 2

T G C T T : 6

X

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

P: G C G T G C T T

Index of T

CGTGC: 0, 4
TGCCT: 2
TGCTT: 6

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

P: G C G T G C T T

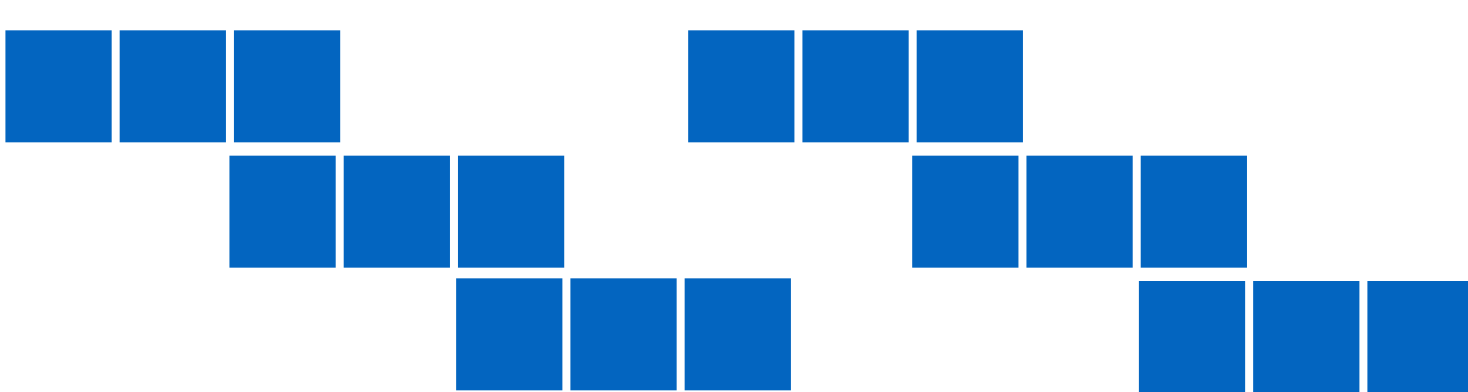
Index of T

C G T G C : 0 , 4
T G C C T : 2
T G C T T : 6


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

P: G C G T G C T T

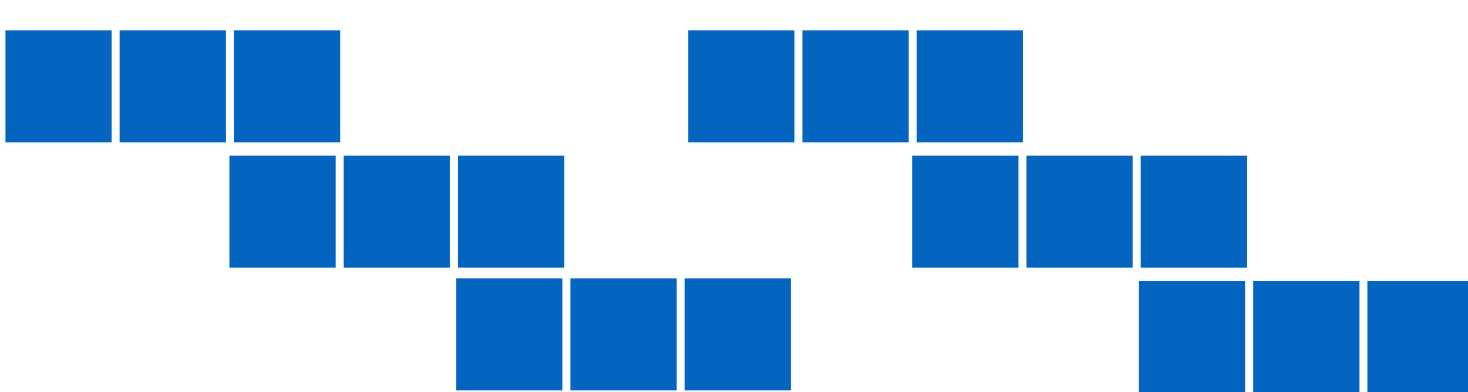
T : 

Index:
all even 

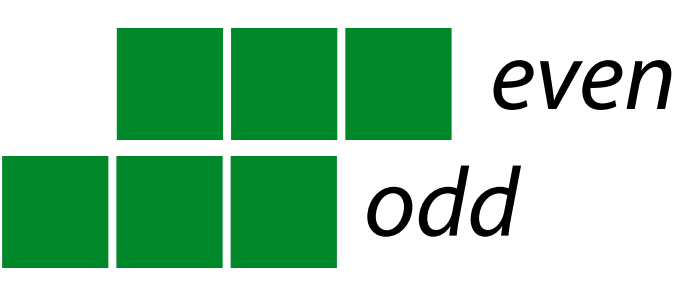
P : 

Query: 

T : 

Index:
all even 

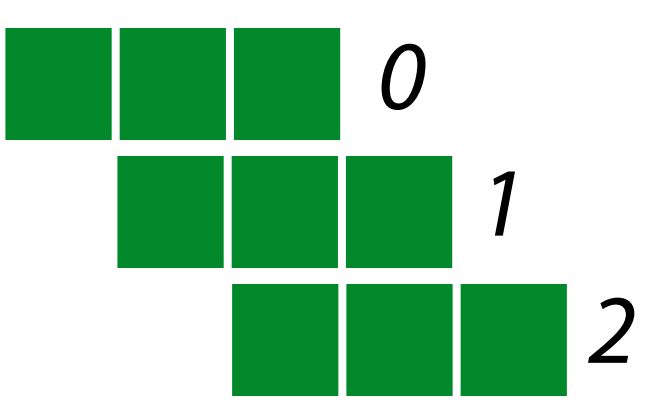
P : 

Query: 

T : 

Index:
every 3rd 

P : 

Query: 

T : 

Index:
every 3rd 

P : 

Query: 
 $0 \bmod 3$
 $1 \bmod 3$
 $2 \bmod 3$

Subsequence of *S*: string of characters also occurring in *S* in the same order

```
>>> seq = 'AACCGGTT'
>>> seq[0] + seq[1] + seq[5] + seq[7]
'AAGT' # subsequence
>>> seq.find('AAGT')
-1 # not a substring
```

Substrings are also subsequences, subsequences are not necessarily substrings

Index of T

C G G G T : 0

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

Index of T

C G G G T : 0

G T C T G : 1

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

Index of T

C G G G T : 0

G T C T G : 1

T G G G C : 2

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

Index of T

C G G G T : 0

C G G T T : 4

G C T C T : 3

G T C T G : 1

T G G G C : 2

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

Index of T

C G G G T : 0

C G G T T : 4

G C T C T : 3

G T C T G : 1

T G G G C : 2

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

P: G C G T A C T