

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

```
def rank(root, m, r):
    nodep = root
    d = 1
```

ウェーブレット木とPython実装详解: 2018年秋 応用情報 午後問3

長大な文字列データを極限まで圧縮し、
高速検索を実現するデータ構造とアルゴリズムの完全解剖

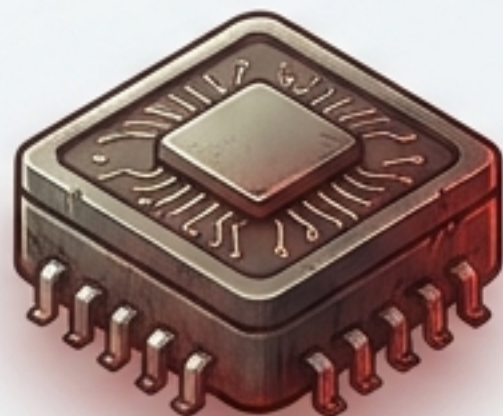


巨大なデータをコンパクトに圧縮し、高速に検索する

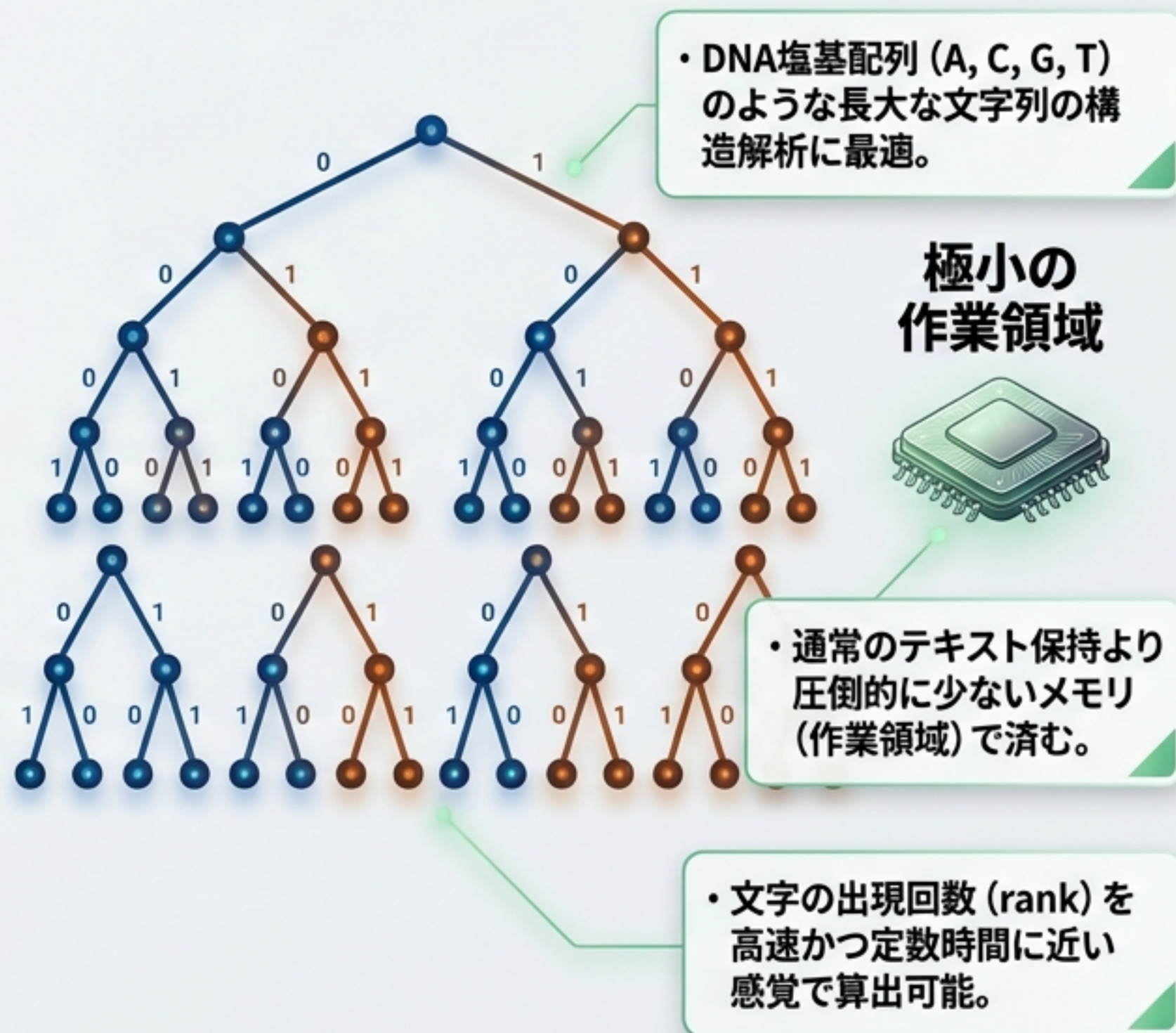
従来のテキスト保持



8～32ビット/文字

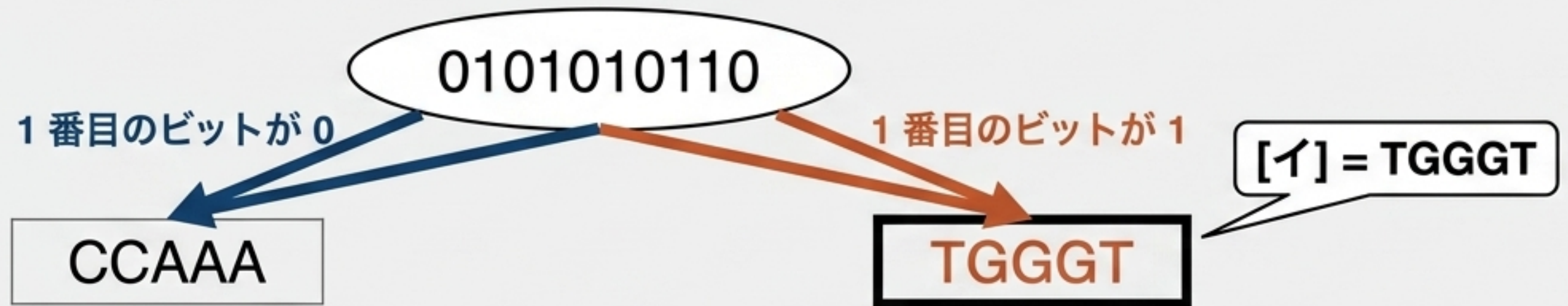


ウェーブレット木

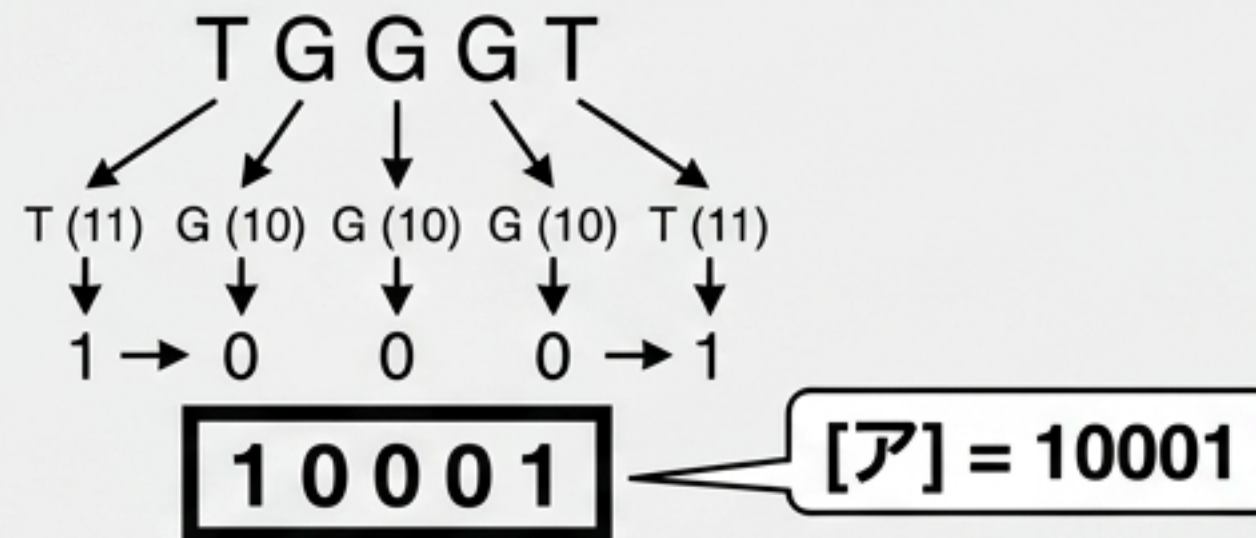


構築ルール：文字を2ビットに符号化し、深さごとに振り分ける

A = 00	C = 01	G = 10	T = 11
--------	--------	--------	--------



深さ 1 (右の子ノード) でのキー値作成 (空欄アの特定)



深さ d のノードでは、各文字の『左から d+1 番目』のビットをキー値とする。ビットが0なら左、1なら右へ。

アーキテクチャの全体像：データ構造と探索アルゴリズム

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

木の構築 / データ構造

指定文字の出現回数カウント / アルゴリズム

```
def rank(root, m, r):
    # 1. ビット抽出
    x = (1 << (DEPTH - d))
    # 2. カウント
    for i in range(n):
        # 3. ノード遷移
        if b == 0:
            # 4. 状態更新
            n = count
```

1. ビット抽出

2. カウント

3. ノード遷移

4. 状態更新

Block 1: 探索アルゴリズムの初期状態

```
def rank(root, m, r):  
    nodep = root  
    d = 1  
    n = m
```

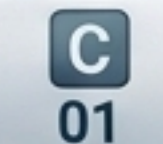
Memory State Visualization



root = 0101010110 (根ノード)



m = 10 → n = 10 (検索対象の長さ)



r = 1 (対象文字 'C' の整数値: 01)



d = 1 (スタートする深さ/ビット位置)

探索が始まる直前。どの文字 (r) を、先頭から何文字 (n) まで探すか、どの階層 (d) からスタートするかをセットする。

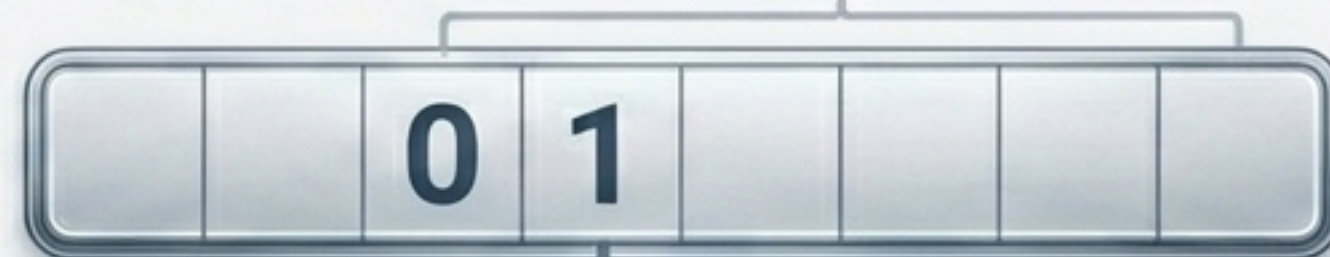
Block 2: 調べたいビット位置だけを狙い撃つ「マスク」の作成

$$x = 1 \ll (\text{DEPTH} - d)$$

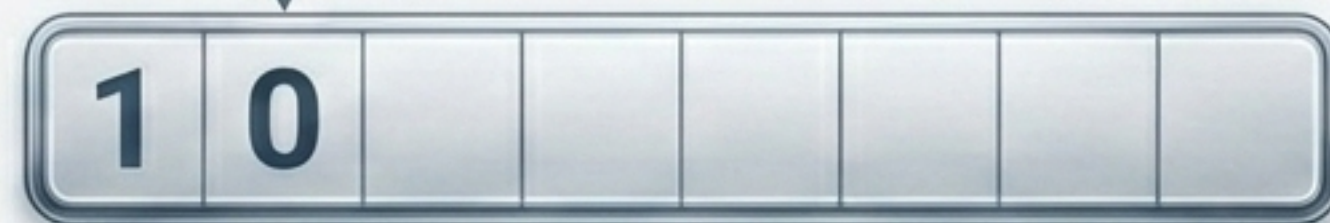
DEPTH = 2

d = 1

Value 1



2 - 1 = 1 shift



変数 d が増える（階層が深くなる）ごとに、マスク位置を右へずらしていく必要がある。よってシフト量は $[エ] = \text{DEPTH} - d$ となる。

Block 3: 論理積 (AND) による対象ビットの抽出と確定

```
x = x & r # [オ] = r  
b = x >> (DEPTH - d)
```

Vertical Math Sandbox

x (Mask) : 1 0
AND r : 0 1

Result : 0 0 (DEPTH - d)

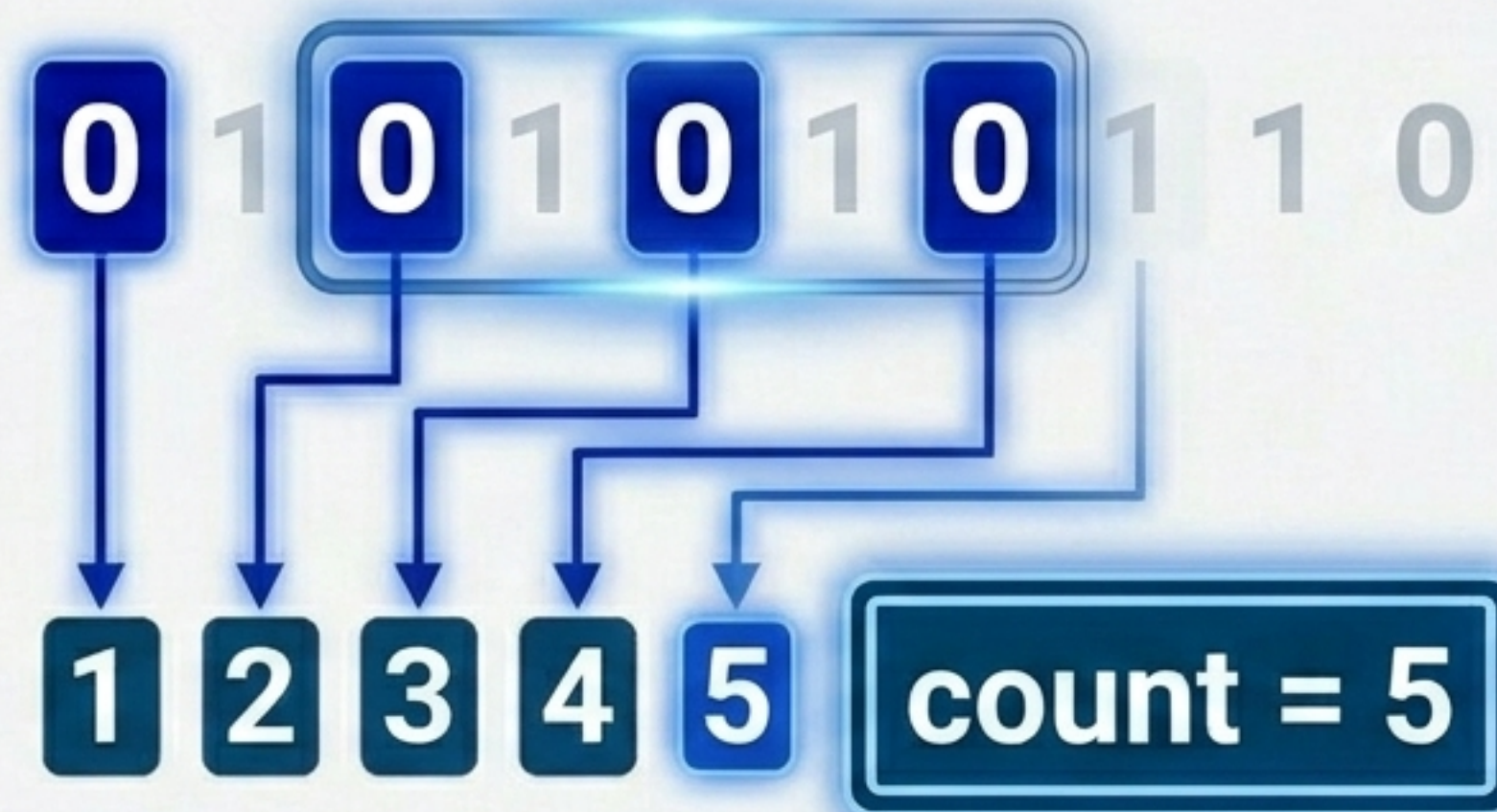
b = 0

マスク x と論理積をとるべき相手は、検索対象の文字コードが格納されている変数 r のみ。よって [オ] = r。

Block 4: 現在のノード内で一致するビットを数え上げる

```
for i in range(n):  
    if b == nodep.key[i]:  
        count += 1
```

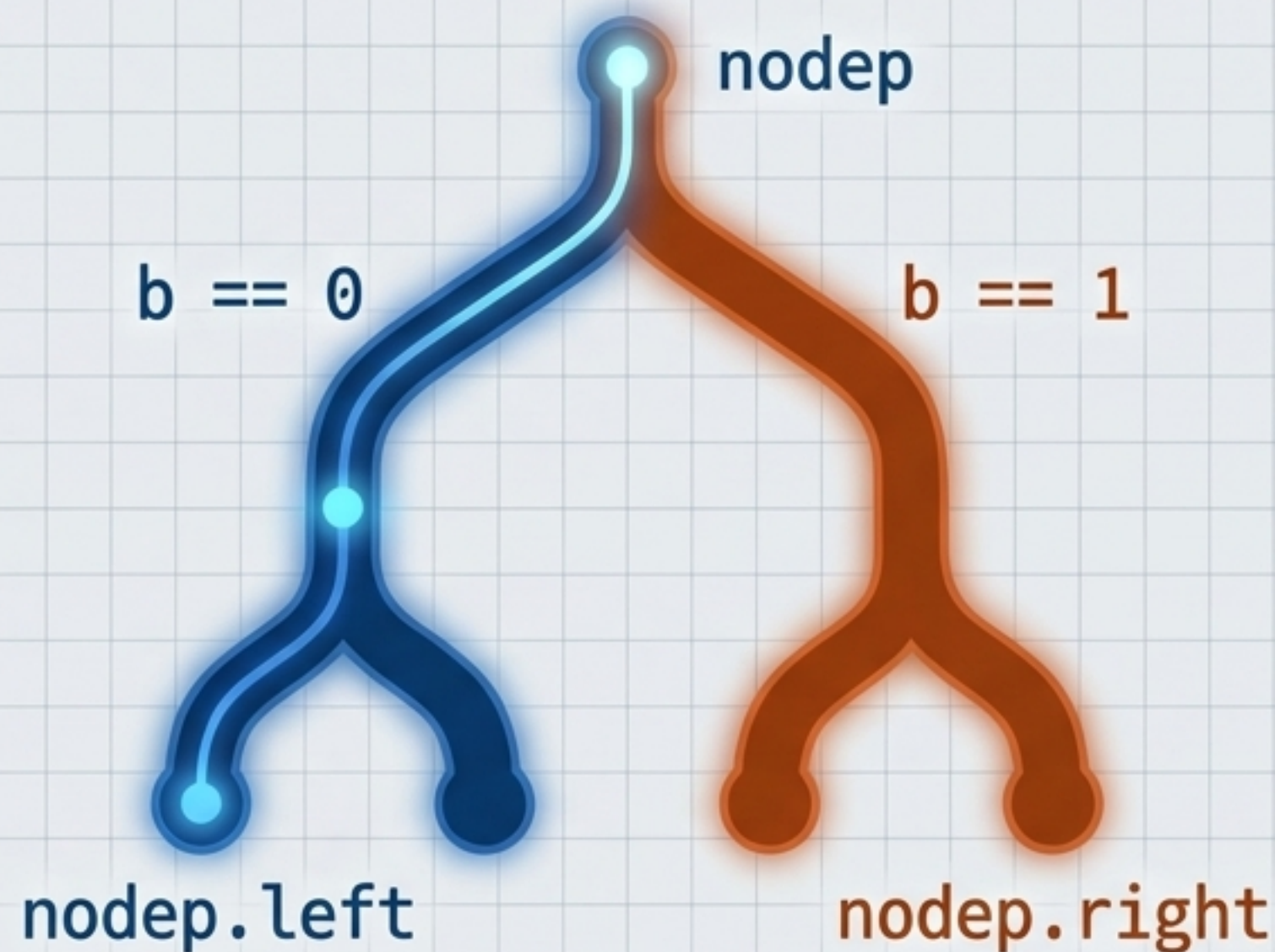
Scanning Animation Frame



抽出した対象ビット b (今回は0) と同じ値が、キー値の中にいくつあるかを走査する。根ノードには0が5個存在するため、 $[ウ] = 5$ 。

Block 5: 「0なら左、1なら右」 — 鉄則に基づくノード遷移

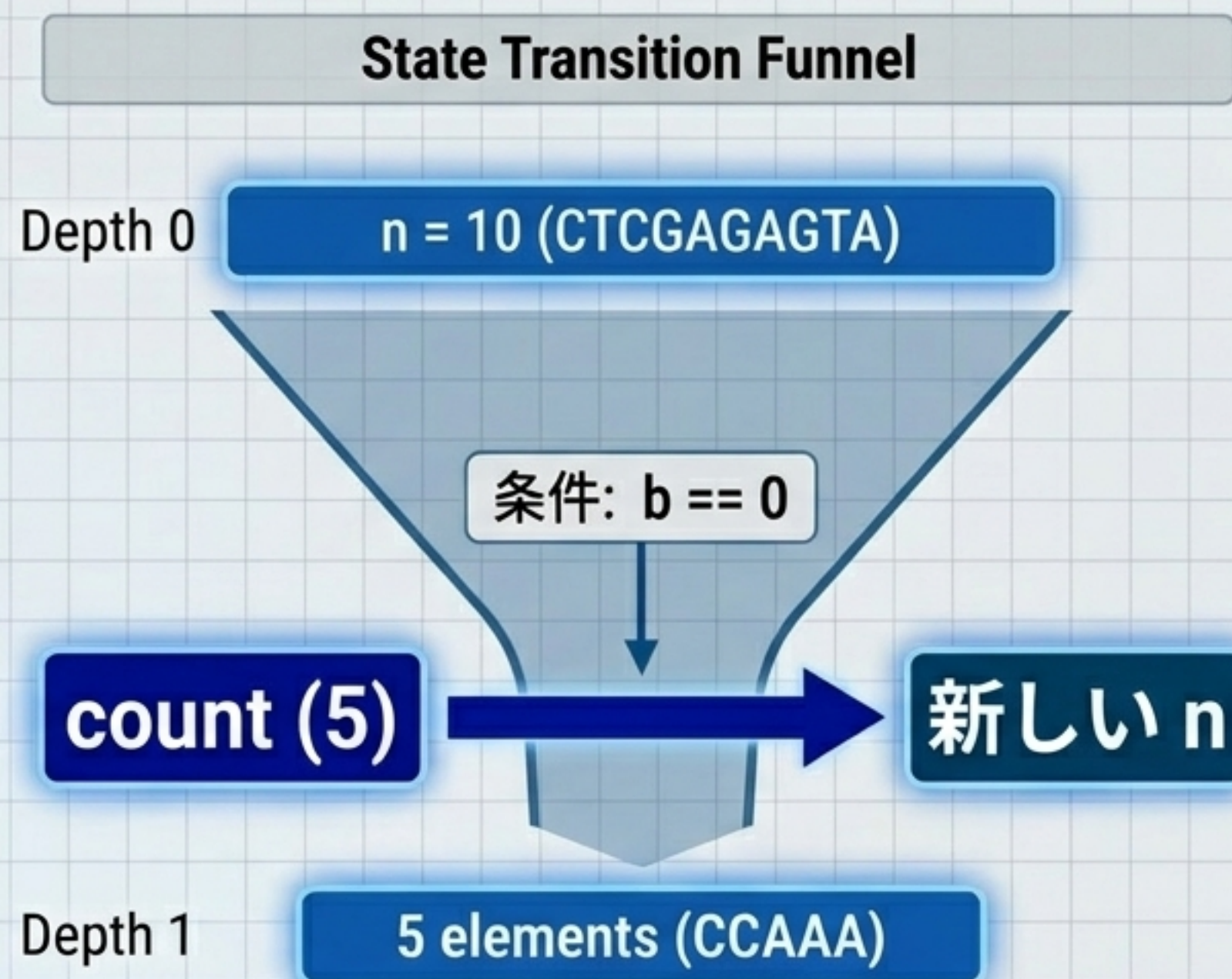
```
if b == 0:    # [力] = 0
    nodep = nodep.left
else:
    nodep = nodep.right
```



ウェーブレット木の構造的絶対ルール。左（left）へ進むための条件基準値は0以外にあり得ない。よって[力]=0。

Block 6: 次の階層へ引き継ぐ「文字列の長さ (n)」の更新

```
n = count    # [キ] = count  
d = d + 1
```



現在の階層で条件に合致し振り分けられた文字の総数 (count) が、そのまま次のノードで調査対象となる長さになる。よって [キ] = count。

計算時間の評価：巨大なデータでも定数時間のように振る舞う理由

- 文字の種類数 = σ
- 文字列の長さ = N

Complexity Matrix

木の深さ (Depth): $\log_2(\sigma)$ 段

常に N 文字分の処理

$$\text{縦} \times \text{横} = O(N \times \log_2(\sigma))$$

完全に文字を振り分けるための深さは $\log_2(\sigma)$ 段 ($[\text{ク}] = \sigma$)。
どの階層を横に見ても、操作する文字数は常に N ($[\text{ケ}] = N$)。
したがって、構築の総計算時間は縦×横の面積である $O(N \times \log_2(\sigma))$ となる。

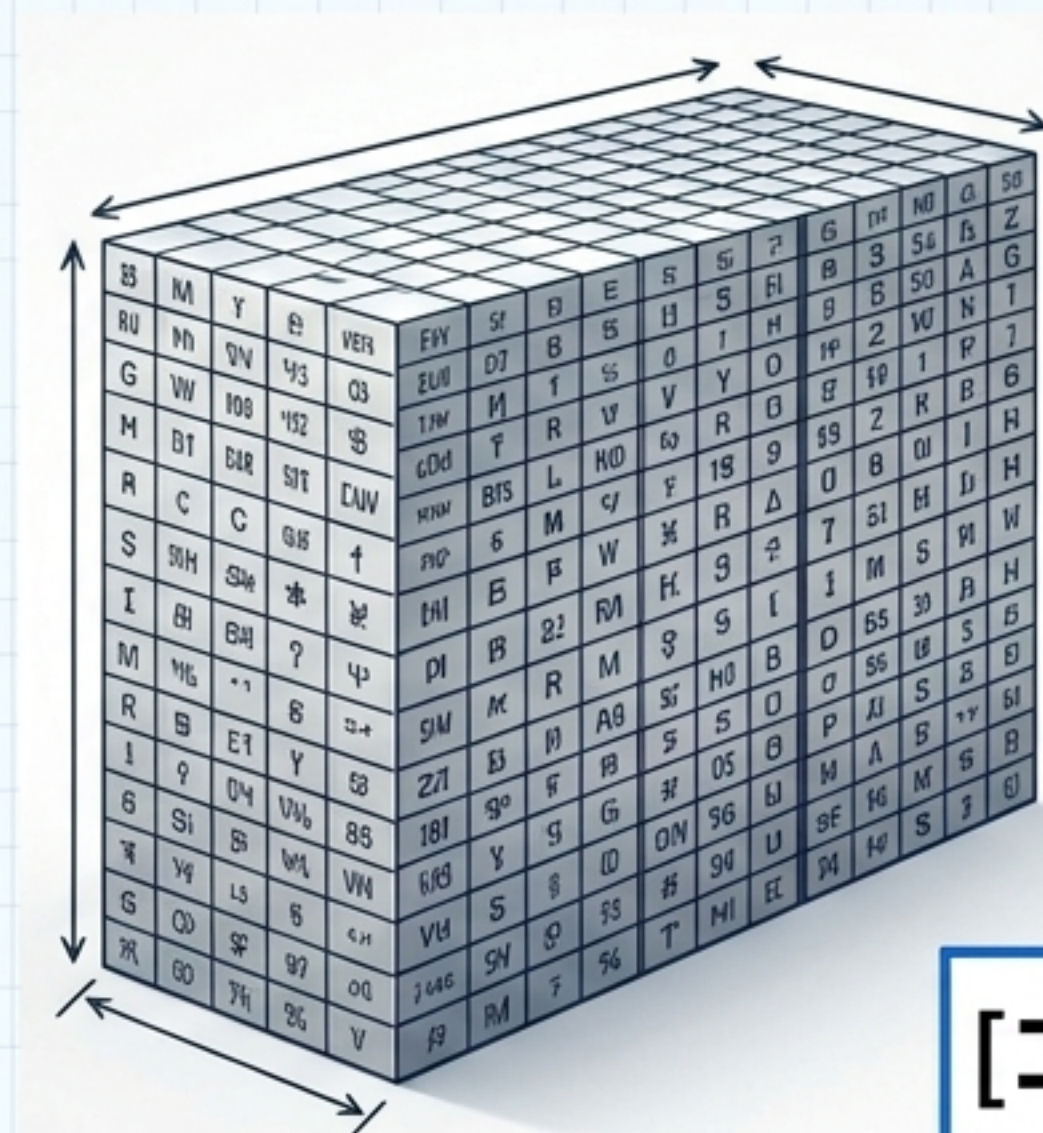
空間計算量の評価：究極の省メモリ構造の証明

1階層あたりの合計ビット数 (N)

×

階層の数 ($\log_2(\sigma)$)

$$\text{総和} = N \times \log_2(\sigma)$$



$$[\text{コ}] = N \times \log_2(\sigma)$$

Takeaway Box

文字の種類数 (σ) が少ないDNAなどのデータにおいて、データ全体をわずか $N \times \log_2(\sigma)$ ビットという極めてコンパクトなサイズに凝縮。これがウェーブレット木が構造解析で重宝される最大の理由である。