

チェイン法の完全解剖

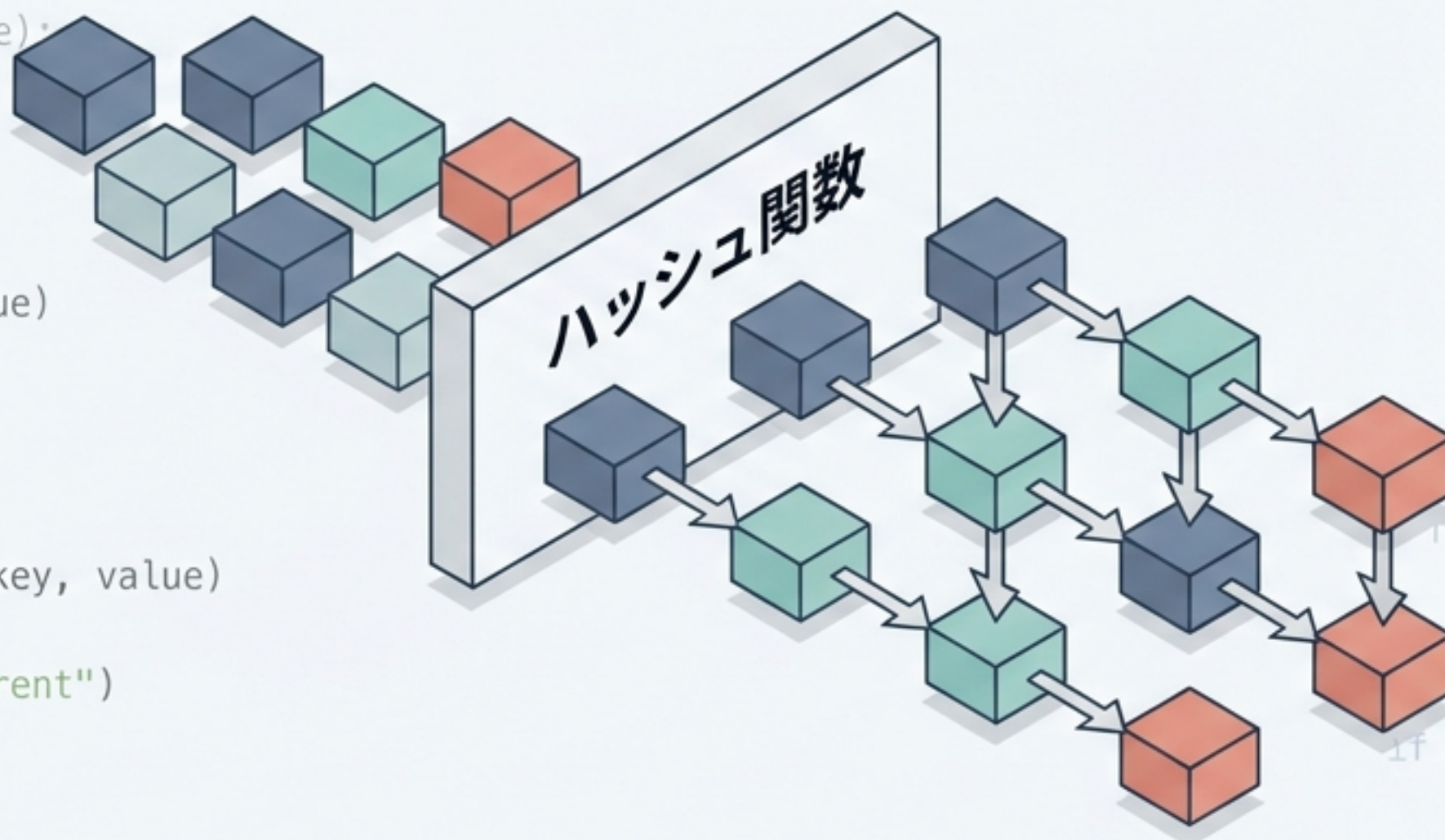
ハッシュ表アルゴリズムの論理からPython実装まで

```
def hash_function(key, table_size):  
    hash_function(key, table_size):  
    return key % table_size
```

```
def hash_function(key, table_size):  
    return key % table_size
```

```
def hash_function(key, table_size):  
    return key % table_size
```

```
if current_node is None:  
    new_node = Node(key, value)  
    new_notus = value  
    value = value  
  
if current_node:  
    current_ = Node(key(key, value)  
    current = n(key key)  
    print("current_+"current")  
else:
```



```
return hash(table_size):  
current(table_size):  
key = size * 10  
value = 2
```

```
if current_node is None:  
    new_node = Node(key, value)  
    new_convty = value  
    value = value
```

```
else:  
    return new_table_size
```

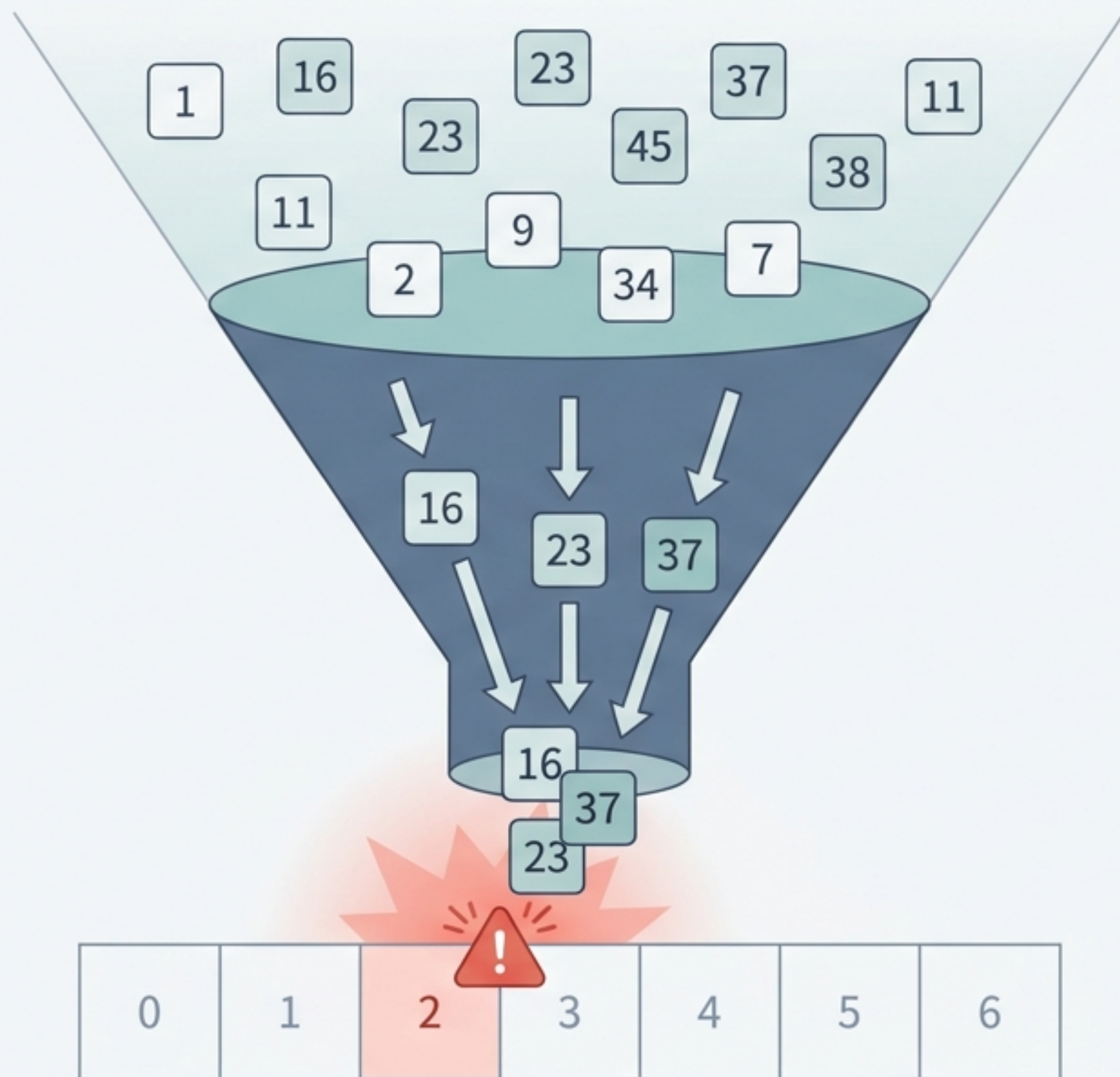
```
hash_function(key, table_size):  
    return key % table_size
```

```
if current_node is None:  
    new_node = Node(key, value)  
    if current_node == None:  
        + i= node  
        _node.log("rs + value")  
    return node
```

```
def hash_function(key,  
    return key % table_size  
if current_node is None:  
    new_node = Node(key, value)  
    return node
```

応用情報技術者試験（2009年春 問2）徹底解説

キー空間 (Key Space)



ハッシュテーブル配列 (Hash Table Array)

なぜ「チェーン法」が必要なのか？

広いキー空間に対して配列の枠は限られているため、マッピング時に重複が起こる。

Q1 解答

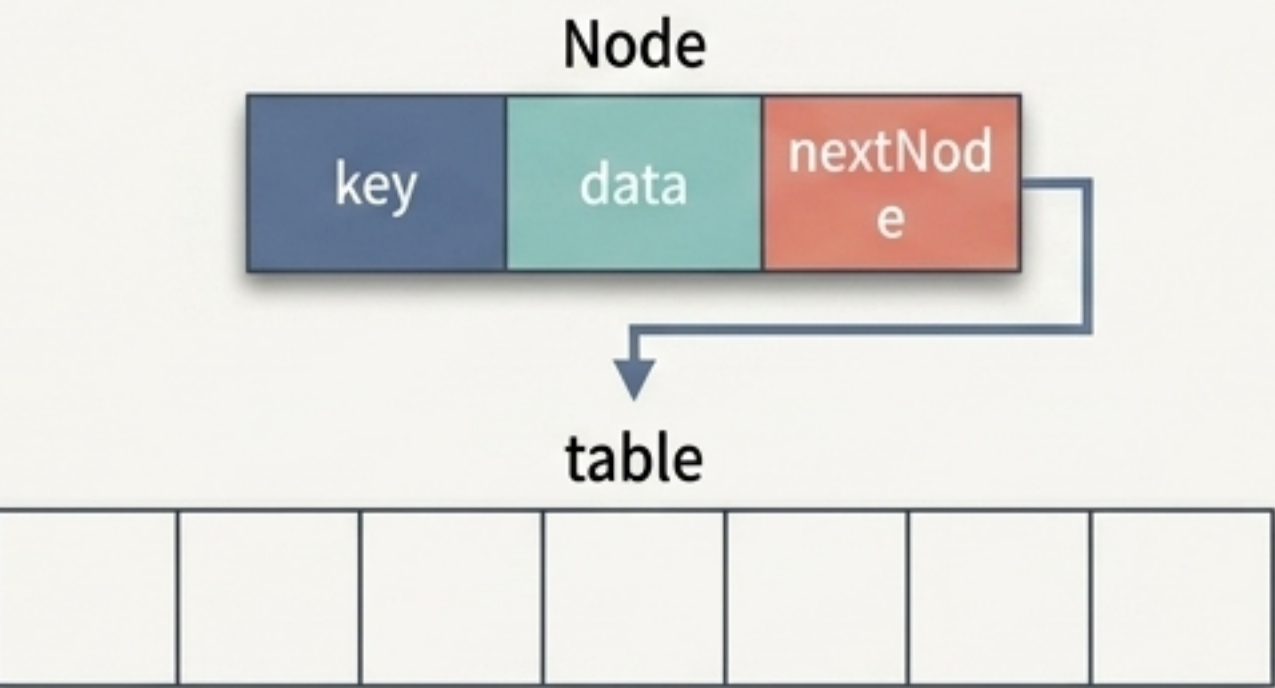
問：衝突 (collision) とはどのような現象か。

答：異なるキーに対して、ハッシュ関数を通した結果が同じ値になってしまう現象。

解決策：同一ハッシュ値のデータを「線形リスト (チェーン)」で繋いで回避する (=チェーン法)。

表 使用する構造体, 配列及び関数

名称	種類	内容
Node	構造体	線形リスト中の各ノードのデータ構造で、次の要素から構成される。 key...キー値 data...データ nextNode...後続ノードへの参照
table	配列	ノードへの参照を格納する。この配列をハッシュ表という。配列の各要素は、table[n]と表記する (n は配列の添字)。配列の添字は 0 から始めるものとする。各要素の初期値は null である。
hashValue(key)	関数	キー値 key を引数として、ハッシュ値を返す。



データ構造のPython実装

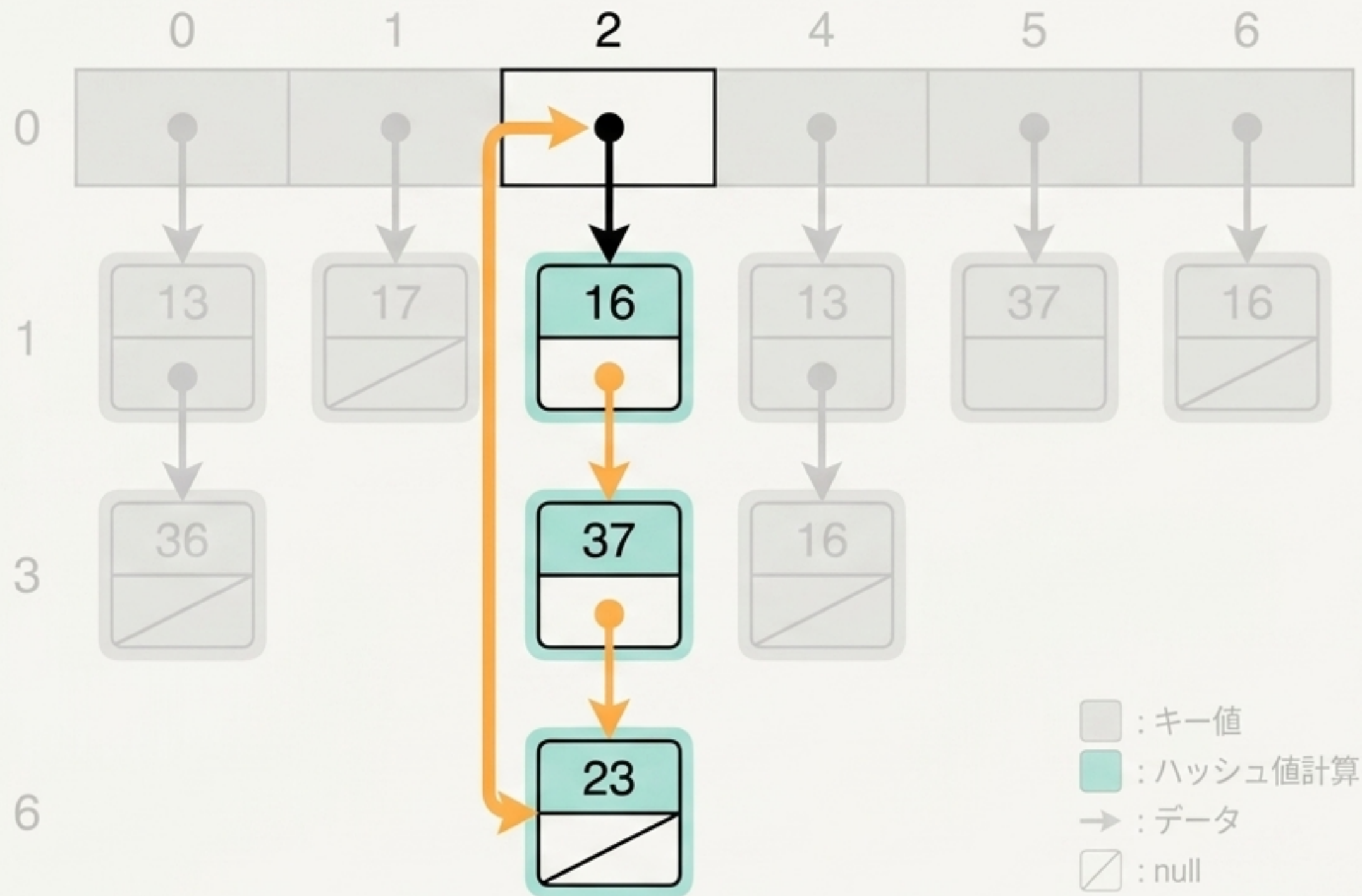
```
1 class Node:
2     def __init__(self,
3         key, data,
4         nextNode=None):
5         nextNode= key,
6             data =None)
7
8
9     table = [None] * 7
10
11
12     def hashValue(key):
13         return key % 7
```

データの器。
nextNodeの初期値
はNone。

ハッシュ表（配列）
の初期化。最初は
すべて空。

ハッシュ関数（7で
割った余りを返
す）。

探索のアルゴリズムとルート追跡



キー値「**23**」を探す場合

Step 1: ハッシュ値の計算: $23 \% 7 = 2$
(添字2のリストに着目)

Step 2: リストを先頭から順に辿る。

Q2-1 解答

問: キー値23のデータを探索するアクセス順序は?

答: $16 \rightarrow 37 \rightarrow 23$

💡 目的のデータがリストの奥にあるほど、アクセスするノード数は増える。

```
function search(key)
  hash ← hashValue(key);
  node ← table[hash];
while(node が null でない)
    if(  )
      return node.data;
    endif
     ;
  endwhile
  return  ;
endfunction
```

図2 探索関数 search のプログラム

探索関数 search() の骨格

```
// 探索するデータのハッシュ値
// 着目する線形リストへの参照
```

```
def search(key):
    hash_idx = hashValue(key)
    node = table[hash_idx]
    while node is not None:
```

- ①探す場所（配列のインデックス）を特定。
- ②該当する線形リストの「先頭ノード」を取得。
- ③ノードが存在する限りループを回す。

(ループ内の処理は次スライドへ)

コードの核心：探索の終了条件と移動

```
while node is not None:
```



[ア] 探索成功

擬似コード:

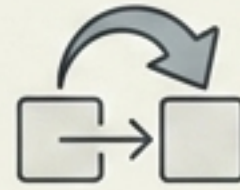
```
node.key == key
```

Python:

```
if node.key == key:  
    return node.data
```

解説:

現在のノードのキーが、探しているキーと一致。



[イ] 次へ移動

擬似コード:

```
node ← node.nextNode
```

Python:

```
node = node.nextNode
```

解説:

一致しなかった場合、注目ノードを「次」にずらす（ポインタの更新）。



[ウ] 探索失敗

擬似コード:

```
null
```

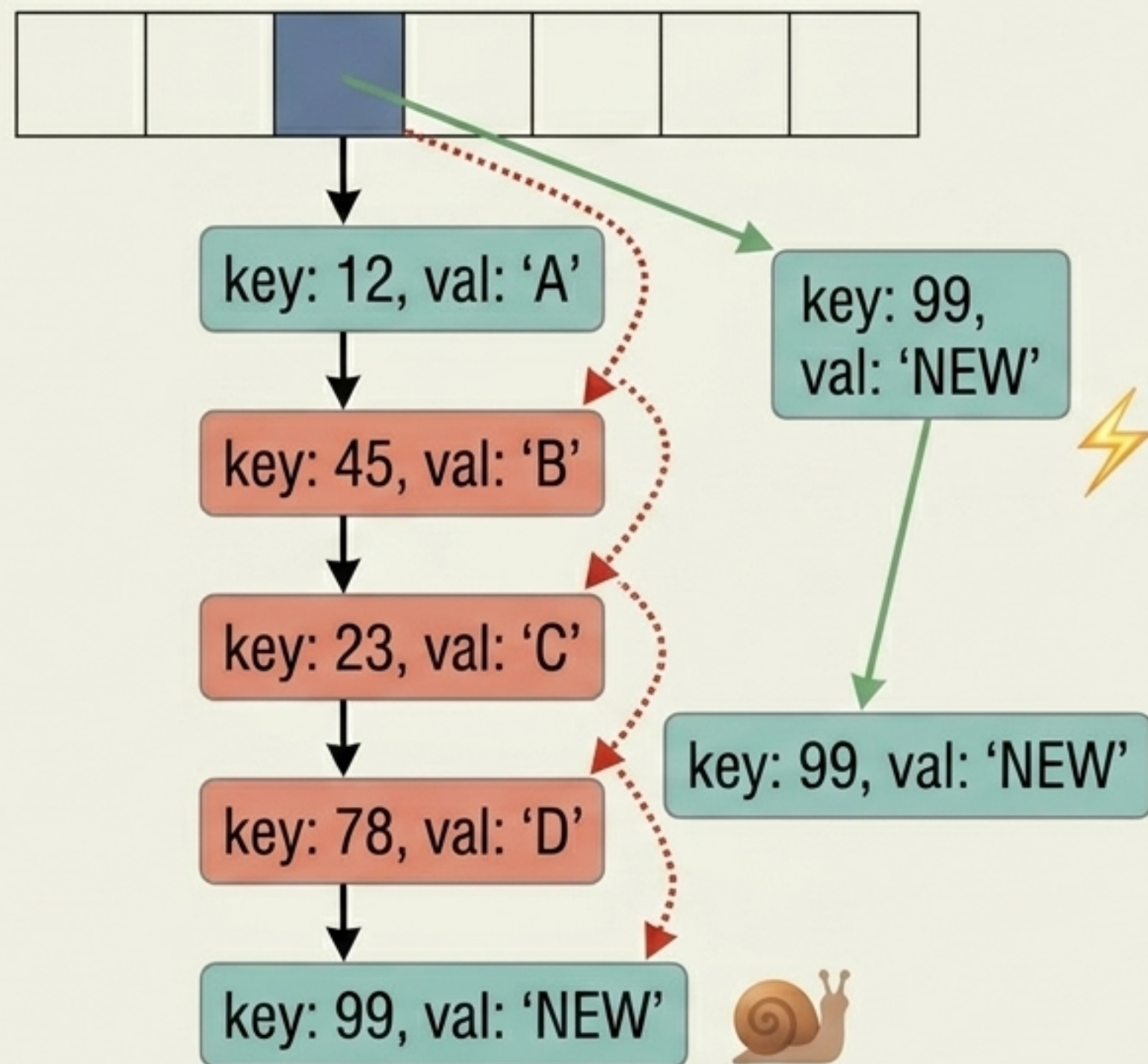
Python:

```
return None
```

解説:

ループを抜けた＝末尾まで探しても見つからなかった。

追加アルゴリズムの肝：なぜ「先頭」に入れるのか？



末尾追加 (Tail Insert)

リストの最後までポインタを辿る必要があり、要素数に比例して時間がかかる。

先頭追加 (Head Insert)

配列の参照先を繋ぎ変えるだけで済むため、即座に挿入可能 (**0(1)**の操作)。



チェーン法では、効率化のために新しいデータは常に「リストの先頭」に割り込ませるのが鉄則。

```

function addNode(key, data)
  hash ← hashValue(key);
  node ← table[hash];
  while (node が null でない)
    if( ア )
      return false;
    endif
    イ ;
  endwhile
  tempNode ← new Node( エ );
  オ ← tempNode;
  return true;
endfunction

```

図3 追加関数 addNode のプログラム

追加関数 addNode() の前半（重複排除）

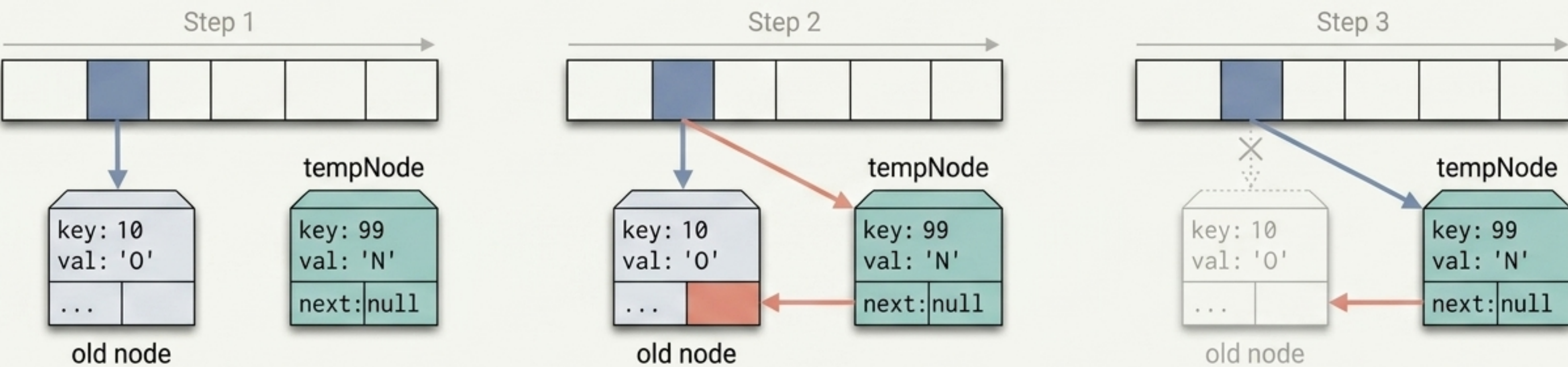
```

def addNode(key, data):
    hash_idx = hashValue(key)
    node = table[hash_idx]
    while node is not None:
        if node.key == key:
            return False
        node = node.nextNode

```

【重要】探索関数とほぼ同じループ。
すでに同じキーが存在する場合は追加せず「False」を返す（重複登録の防止）。

コードの核心：ノード生成とポインタの繋ぎ変え

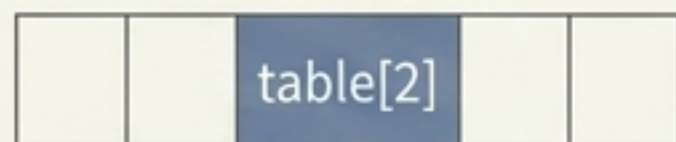


【エ】 ノード生成		【オ】 先頭の更新	
擬似コード	key, data, node	擬似コード	table[hash]
Python	tempNode = Node(key, data, node)	Python	table[hash_idx] = tempNode
解説	新しいノードの nextNode に「現在の先頭ノード (node)」を指定して箱を生成。	解説	配列の入りを、新しく作った tempNode に向け直して繋ぎ変え完了。

実践：Pythonコードの動作トレース

Step 1

```
addNode(16, "DataA")
```

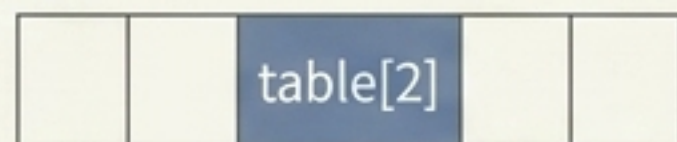


key: 16
val: 'DataA'
next: null

(新規作成)

Step 2

```
addNode(37, "DataB")
```



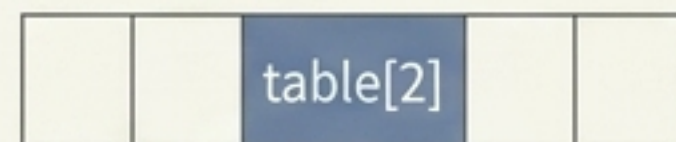
key: 37
val: 'DataB'
next: —

key: 16
val: 'DataA'
next: null

(衝突！先頭に37が割り込む)

Step 3

```
addNode(23, "DataC")
```



key: 23
val: 'DataC'
next: —

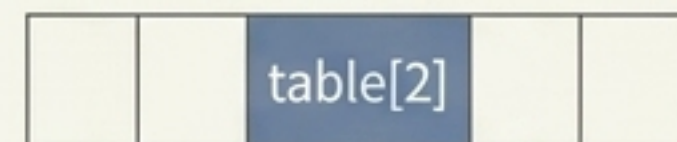
key: 37
val: 'DataB'
next: —

key: 16
val: 'DataA'
next: null

(衝突！さらに先頭に23が割り込む)

Step 4

```
search(23)
```



key: 23
val: 'DataC'
next: —

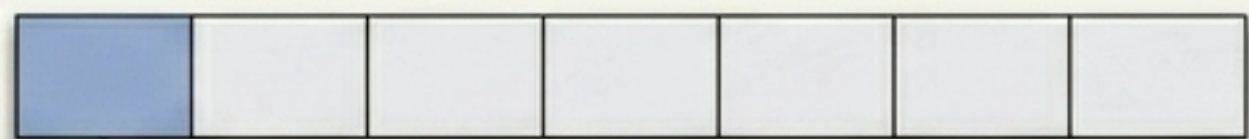
key: 37
val: 'DataB'
next: —

key: 16
val: 'DataA'
next: null

先頭が23なので、一発でヒット！
("DataC" が返る)

チェイン法の性能評価（計算量）

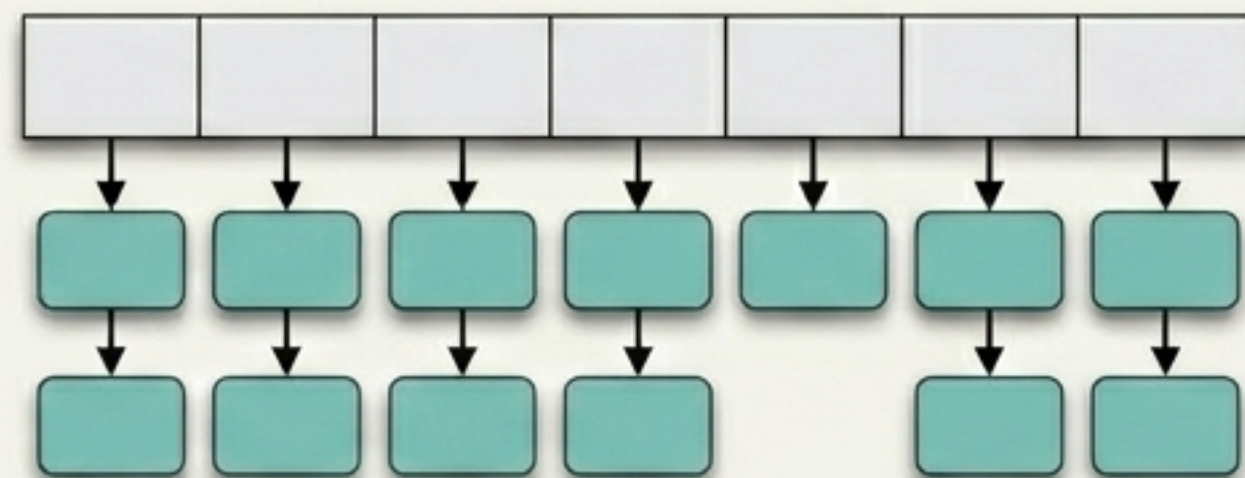
【力】 最悪のケース (Worst Case)



状態: すべてのデータのハッシュ値が同じ値になる

評価: 一つの長い線形リストとなり、ハッシュのメリットが消滅 (実質的な線形探索となり計算量 $O(n)$)。

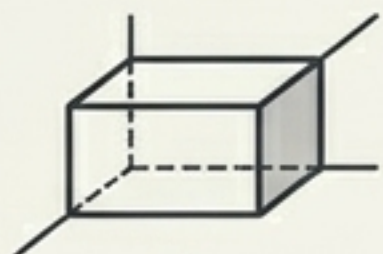
【キ】 理想のケース (Average/Ideal Case)



状態: ハッシュ表の大きさがデータ数に対して十分大きい場合

評価: リストの平均長は短く一定とみなせるため、計算量は $O(1)$ 。2分探索木 ($O(\log n)$) よりも高速。

Summary: 概念とコードの対応マップ



箱（ノード）を作る →

```
Node(key, data, nextNode)
```



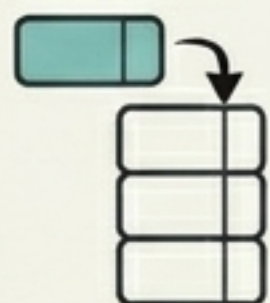
探す場所を決める →

```
hash_idx = key % 7
```



次の箱へ進む →

```
node = node.nextNode
```



先頭に割り込む →

```
temp = Node(..., table[h]);  
table[h] = temp
```

データ構造とポインタの動きを視覚的にイメージすることで、
コードの1行1行が持つ「物理的な意味」が明確になります。