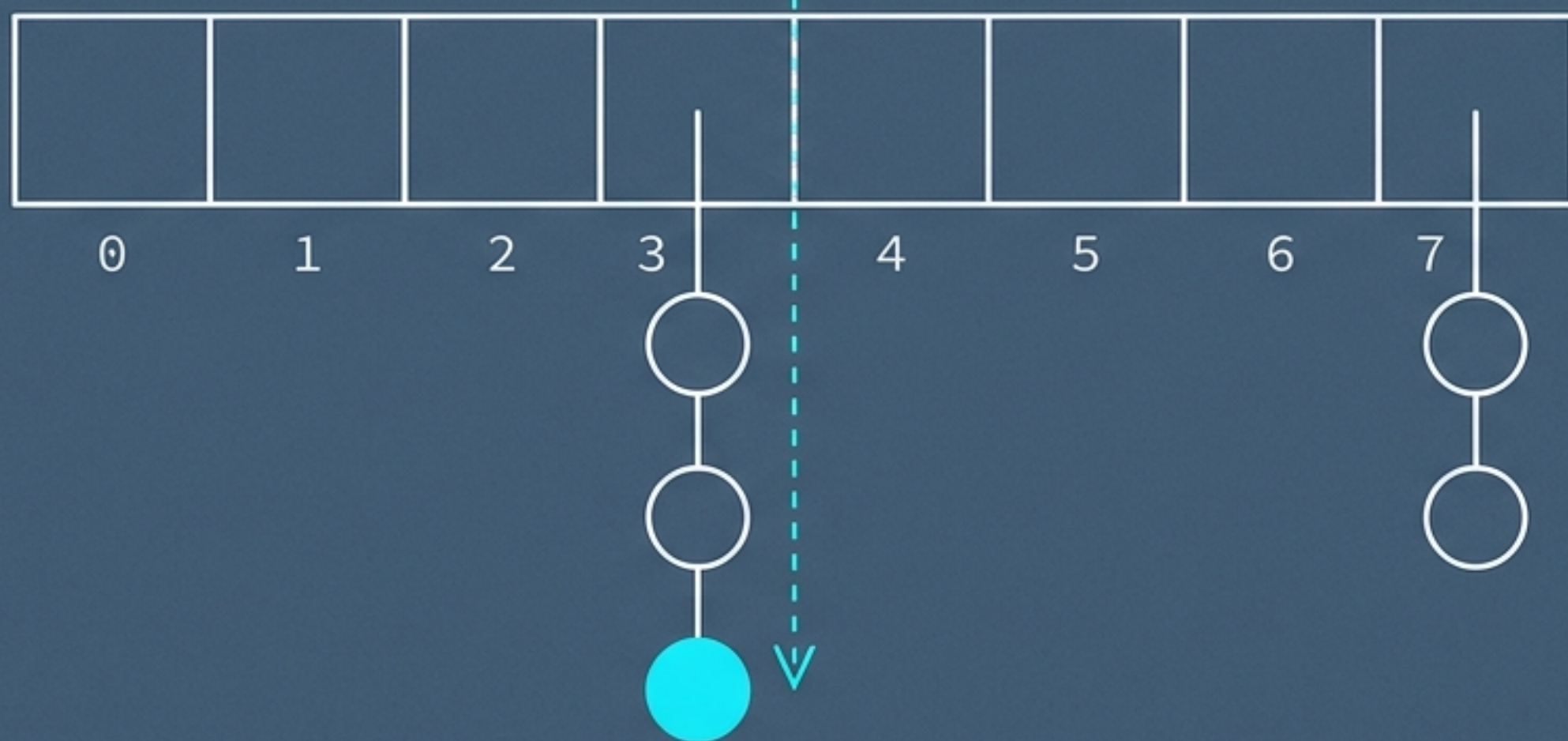


データの「住所」を整理する：チェイン法入門

ハッシュテーブルの衝突を美しく解決するアルゴリズム

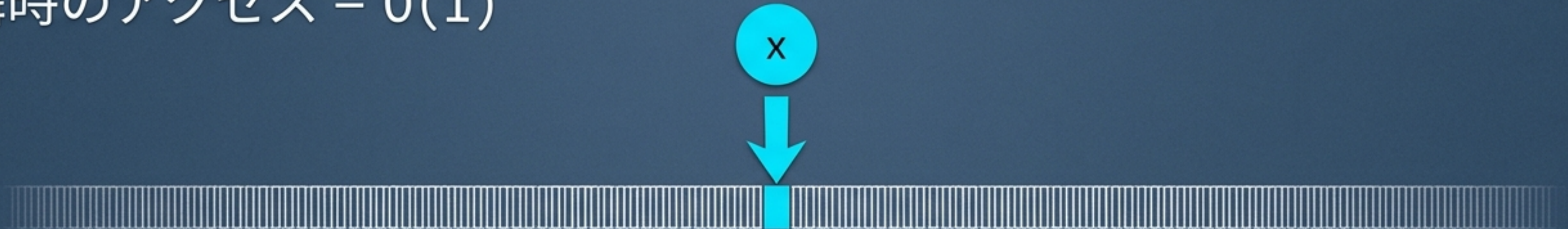


線形探索 - $O(n)$



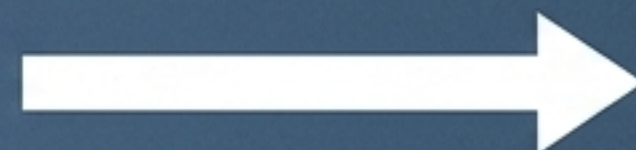
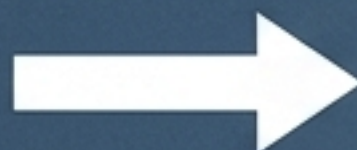
膨大なデータからの探索は遅い: データが100万個ある場合、端から探すと最悪100万回の確認が必要。

瞬時のアクセス - $O(1)$



目標: データ量に依存せず、常に「一瞬」でデータを見つけ出すこと (計算量 $O(1)$)。

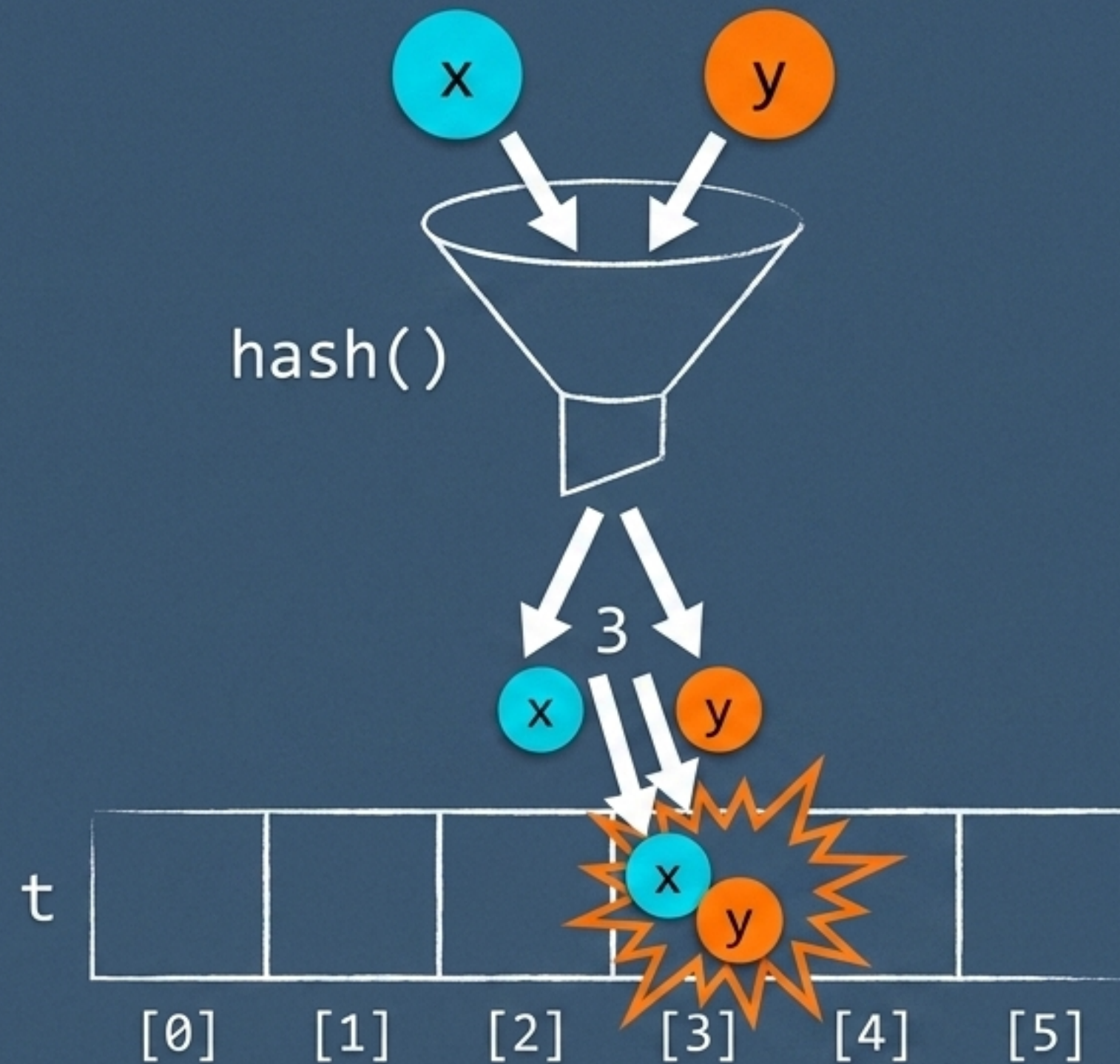
データ
(x)



3



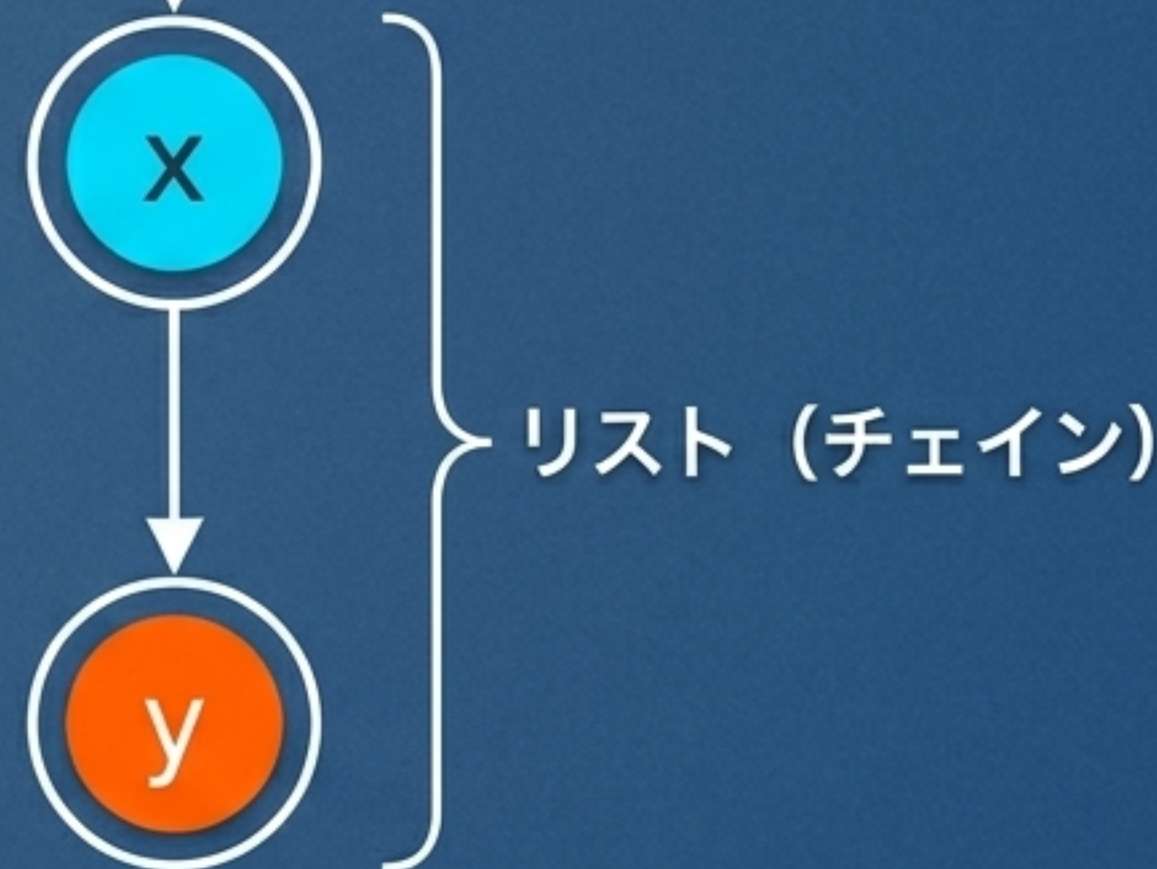
- ・魔法の箱: ハッシュ関数: データを入力すると、配列の「インデックス (住所)」を計算して返す関数。
- ・データを直接その「住所」に保管することで、探す時も同じ計算をするだけで瞬時に場所を特定できる。

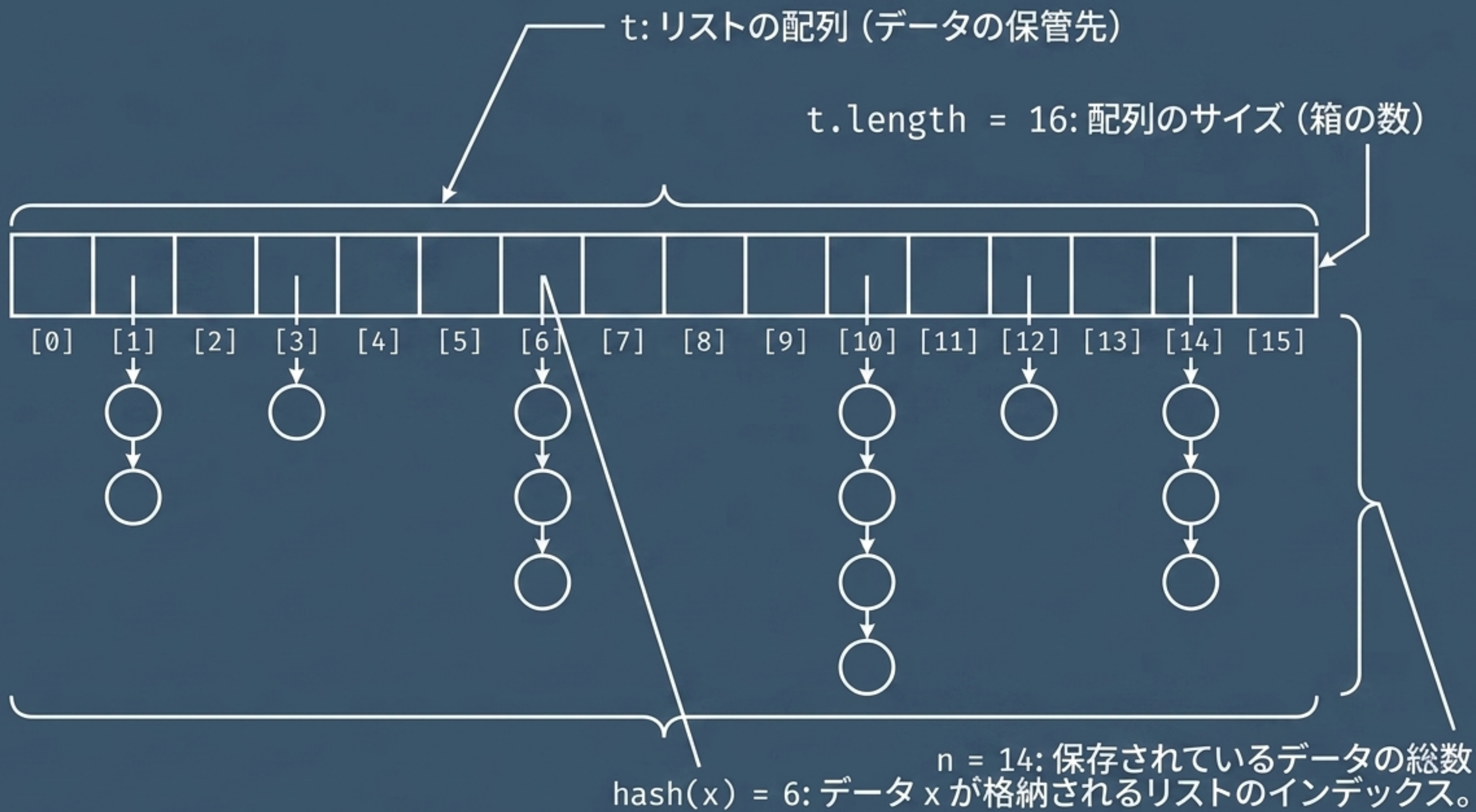


致命的な弱点：ハッシュ値の衝突 (Collision): 異なるデータでも、計算結果（住所）が偶然同じになってしまうことがある。
ひとつの箱にはひとつのデータしか入らない。後から来たデータはどこへ行けばいいのか？



- 解決策：チェイン法 (Chaining) : 配列の箱に直接データを入れるのではなく、箱には「リスト (鎖)」をぶら下げる。
- 衝突が起きても、リストの末尾にデータを繋ぐだけで美しく解決できる。

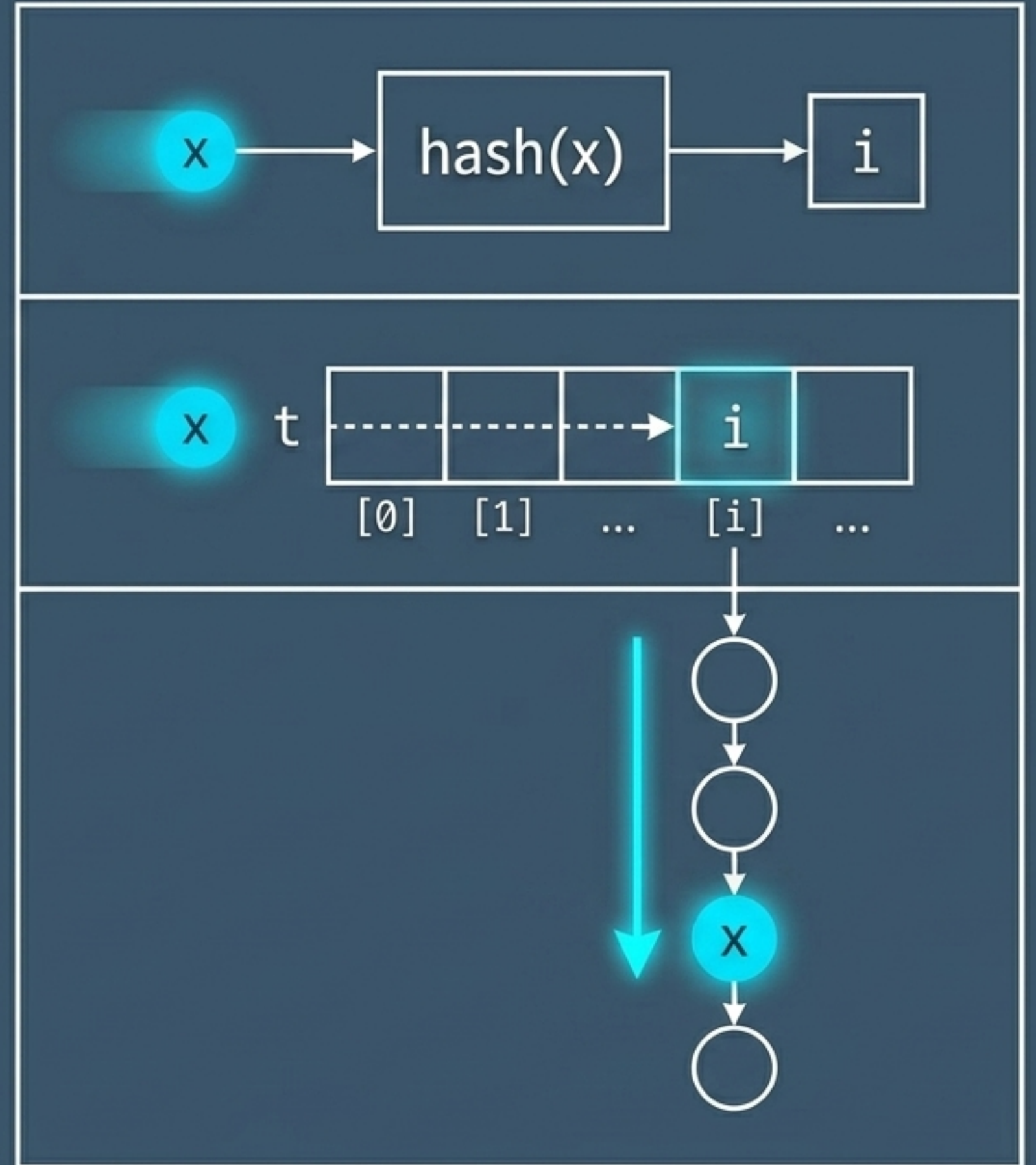




データの検索 find(x)

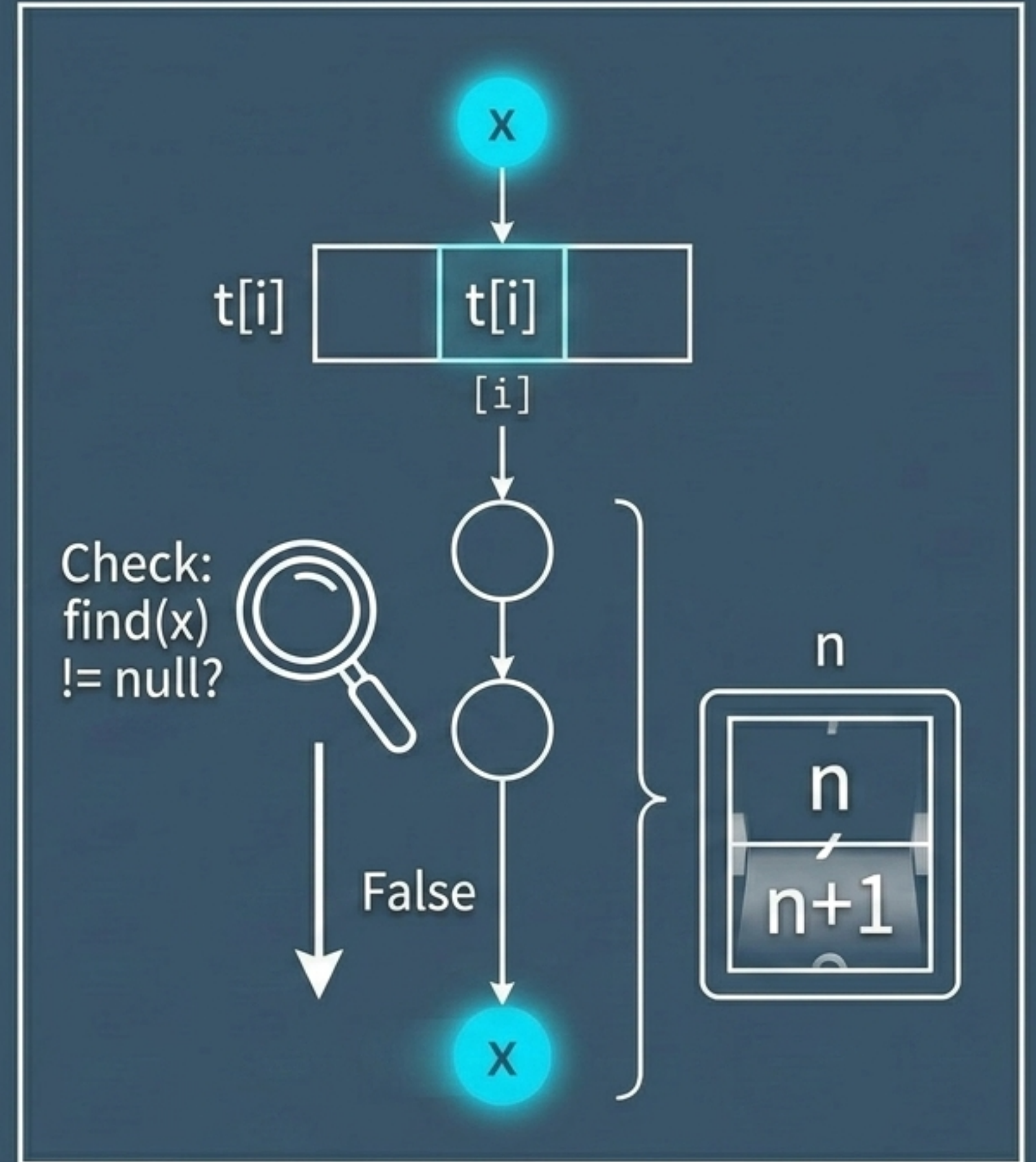
1. $\text{hash}(x)$ を計算し、インデックスを特定。
2. 配列 $t[\text{hash}(x)]$ へ瞬時にジャンプ。
3. そのリストだけを上から順に探索。

ポイント：100万個のデータがあっても、探すのはその「住所」にある短いリストだけ。



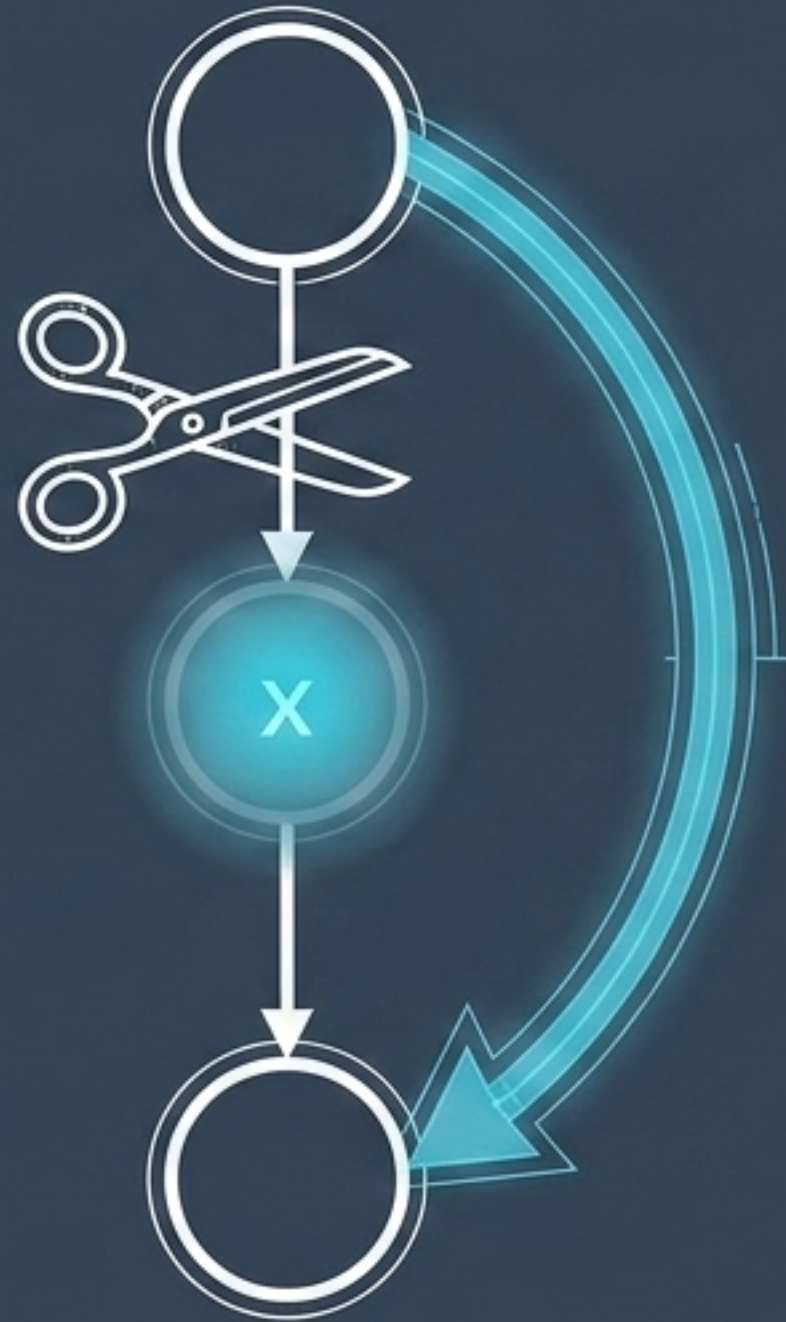
データの追加 add(x)

- 同じデータが既に存在しないか確認する（重複防止）。
- 存在しなければ、リスト $t[\text{hash}(x)]$ の末尾にデータを追加。
- データ総数 n を1増やす。



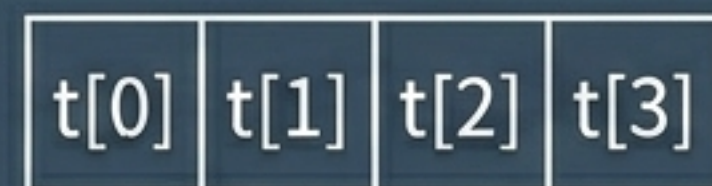
データの削除 remove(x)

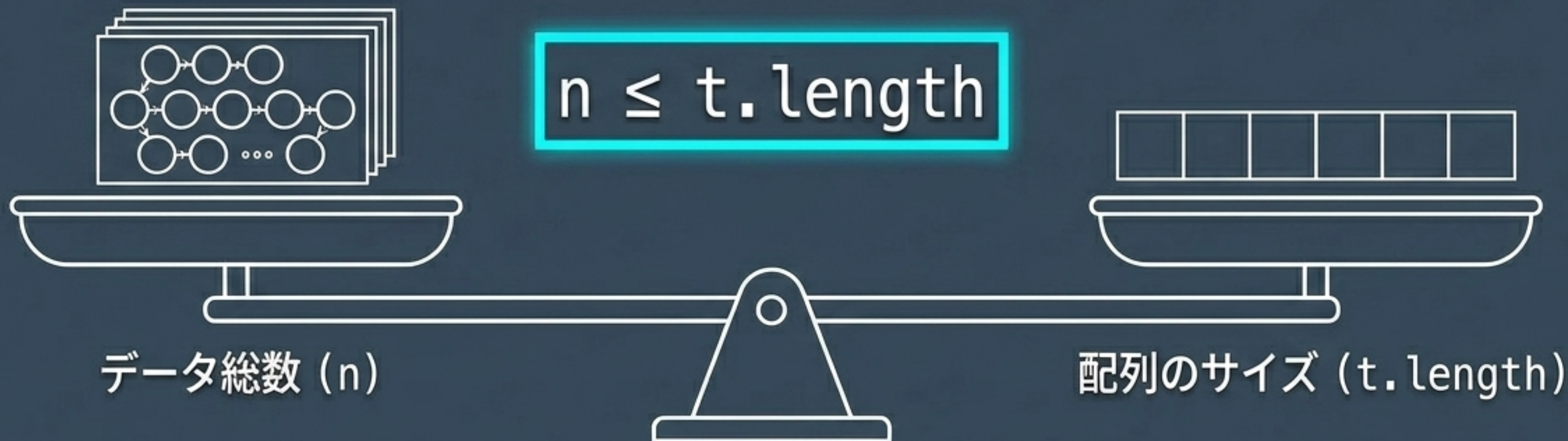
- リスト $t[\text{hash}(x)]$ を辿り、対象のデータを発見する。
- 前後のノードの「ポインタ（繋がり）」を繋ぎ変える。
- データ総数 n を1減らす。



隠れた危険：長すぎるチェーン

- データが増えすぎると、1つの箱にぶら下がるリストが長大化する。
- リストの探索は1つずつ確認するため、長すぎると「瞬時のアクセス ($O(1)$)」が「遅い線形探索 ($O(n)$)」に劣化してしまう。



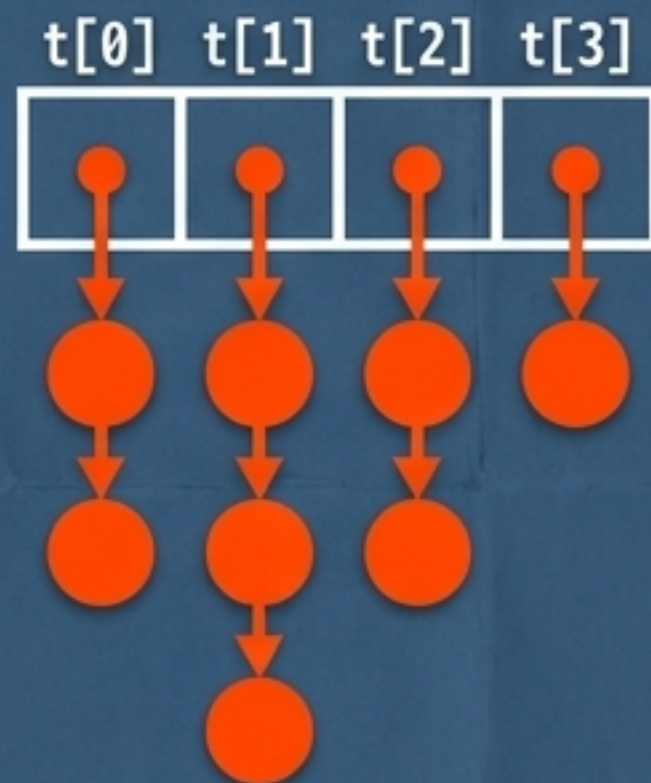


絶対のルール（不変条件）：常に「データ総数」 $\leq$ 「配列のサイズ」を保つこと。

この条件が守られている限り、リストの平均的な長さは常に「1以下」になり、
検索にかかる時間は期待値で $O(1)$ に保証される。

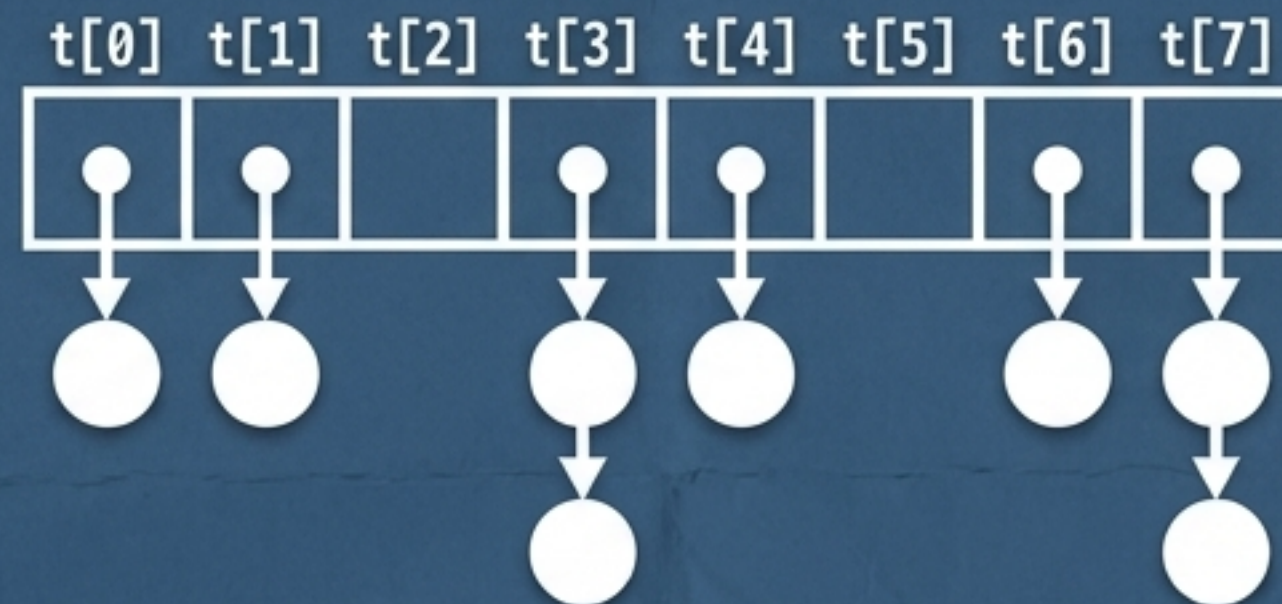
配列の拡張 `resize()`

限界超過



`n=8`, `t.length=4`

リサイズ完了



`n=8`, `t.length=8`

- 限界を超えたら、箱の数を2倍にして再分配する！
- 新しいサイズの配列を用意し、全てのデータを新しいハッシュ関数で計算し直して（再ハッシュ）、新しい配列に入れ直す。

償却実行時間 (Amortized Analysis)

- リサイズは重い、でも稀:
全データの入れ直しには
時間がかかる ($O(n)$)。
- しかし、配列を「2倍」ず
つ増やすことで、リサイズ
が発生する頻度はどんどん
減っていく。
- 償却実行時間 (Amortized
Time): 何度も操作を行った
平均で見ると、1回あたりの
コストは $O(1)$ に収まる。



比較表 (Comparison Matrix): Array vs. Linked List vs. Hash Table

	検索 (Find)	追加 (Add)	削除 (Remove)
配列 (Array)	$O(n)$ 遅い	$O(n)$ 遅い	$O(n)$ 遅い
連結リスト (Linked List)	$O(n)$ 遅い	$O(1)$ 速い	$O(n)$ 遅い
チェイン法ハッシュテーブル	期待値 $O(1)$ 速い	期待値 $O(1)$ 速い	期待値 $O(1)$ 速い

究極のハイブリッド構造:

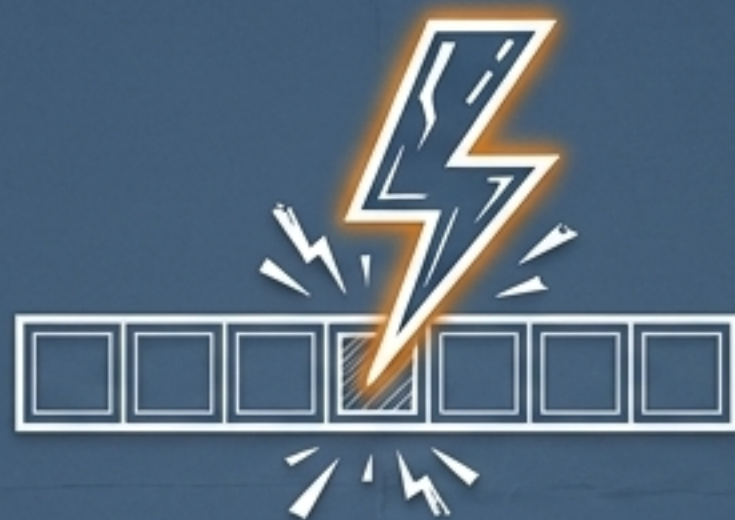
- 配列の「計算による瞬時アクセス」と、連結リストの「柔軟な追加・削除」の長所を完璧に融合。
- 平均して全ての基本操作を $O(1)$ で実行できる。
- ハッシュテーブルは現代のソフトウェア工学における不可欠な基盤技術。

チェイン法を用いたハッシュテーブルの3つの要点



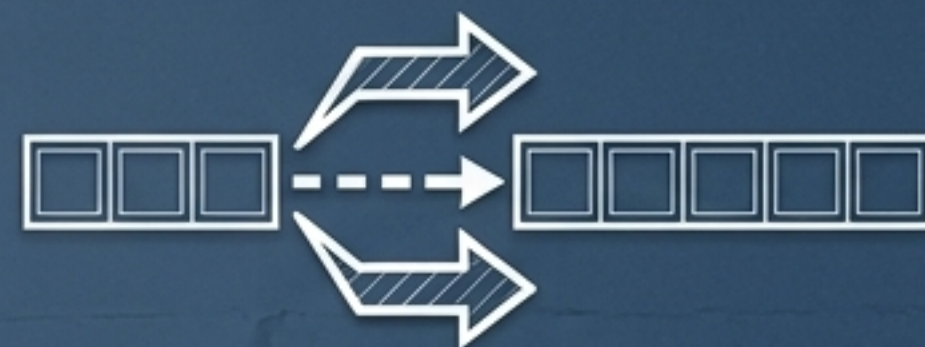
1. 配列 × リスト

衝突をリストをぶら下げる
ことで柔軟に解決。



2. 圧倒的な速度

ハッシュ関数により、平均
 $O(1)$ でデータにアクセス。



3. 動的リサイズ

限界が来たら配列を2倍に拡
張し、リストを常に短く保つ。

複雑なデータも、美しい構造によって瞬時に整理・検索できる。