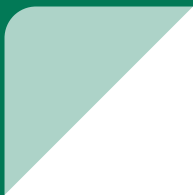
データ構造

技術研修



データ構造とは

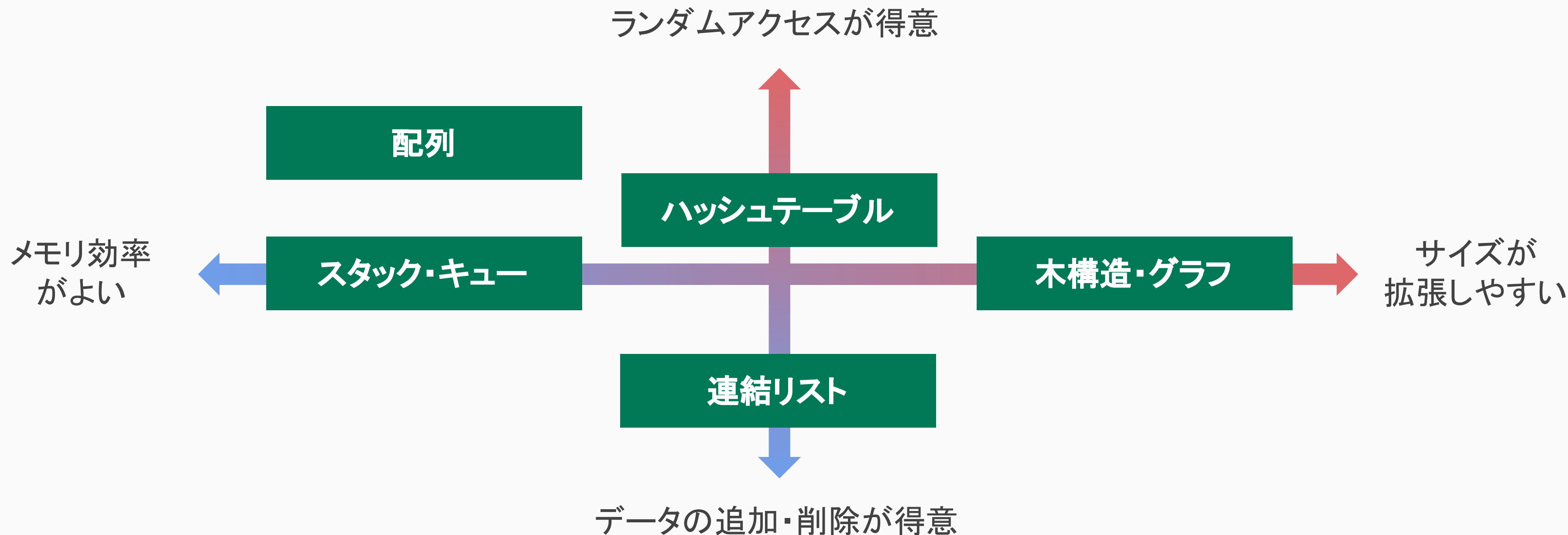
データ構造とは、**形式化されたデータの集まり**である。

データ構造の例：

- 配列
- ハッシュテーブル(辞書構造)

データ構造の選択

データ構造は種類によって得意なことと苦手なことがあるので、データ構造の選択は**プログラムの動作効率**を決める重要な要素である。



もくじ

1.単純なデータ構造

- 配列
- 連結リスト
- スタック
- キュー

2.複雑なデータ構造

- ハッシュテーブル
- グラフ構造
- 木構造
- 二分探索木
- 赤黒木
- B+木とRDBのインデックス

単純なデータ構造

配列 (array)

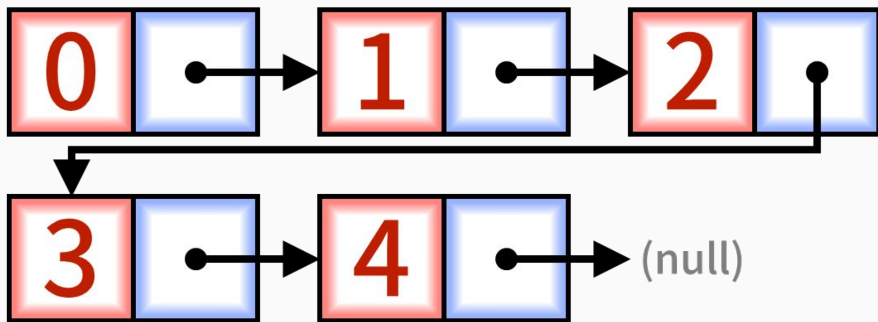
- データを連続したメモリに並べたもの
- 「先頭のアドレス」と「長さ」で管理することが多い
- 長さが決まっており、ランダムアクセスが早い
- 一般的に広く使われる

Note

「配列型の変数」に管理領域が入るプログラミング言語(C#, Javaなど)と、連続したメモリ領域が直接入る言語(C++, Goなど)がある

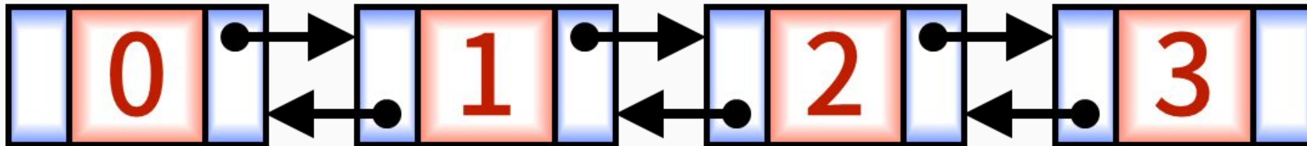
連結リスト (linked list)

- データと「次のデータへのポインタ」をセットにして管理
- 長さが無制限で、(直前のデータが事前に分かっていれば)途中へのデータ追加・削除が早い
- 先頭から順にデータを辿るのは早いが、ランダムアクセスが遅い
- 途中へのデータの追加・削除を多く行う場面で使われる



連結リスト (linked list)

- 各データが前後両方のデータへのポインタを持つような連結リストを**双方向連結リスト**という（対して、後ろへのポインタのみを持つものを片方向連結リストという）
- 末尾からもデータを列挙することができる



スタック (stack)

- 後入れ先出し(LIFO: Last In First Out)で保存するデータ構造
- プログラムの基本構造など、様々な場面で使われる (例: **スタック**トレース)
- ふつう配列を使って実装され、容量が足りなくなったらより大きな配列に乗り換える

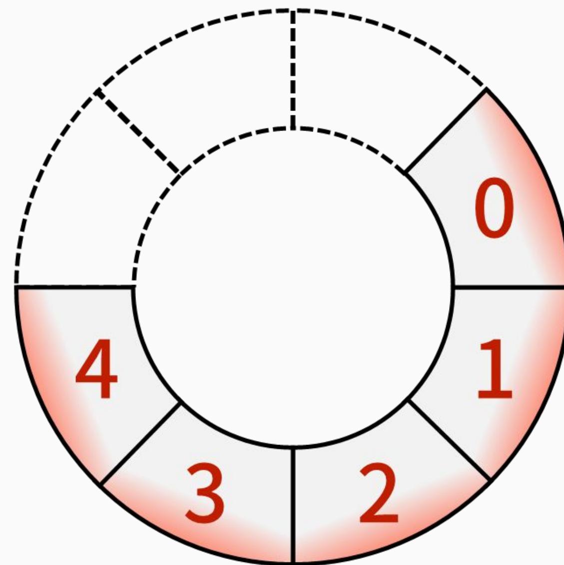
キュー (queue)

- 先入れ先出し(FIFO: First In First Out)で保存するデータ構造
- 順序が重要なものによく使われる (例: メッセージ**キュー**)
- ふつう配列を使って実装される

循環バッファ (ring buffer)

キューを配列で素直に実装しようとすると、データが先頭から消え、末尾に追加されていくので少しずつ後ろにずれていってしまう。

これを解消するため、配列の先頭と末尾が繋がっていると"見なす"**循環バッファ**を使う。

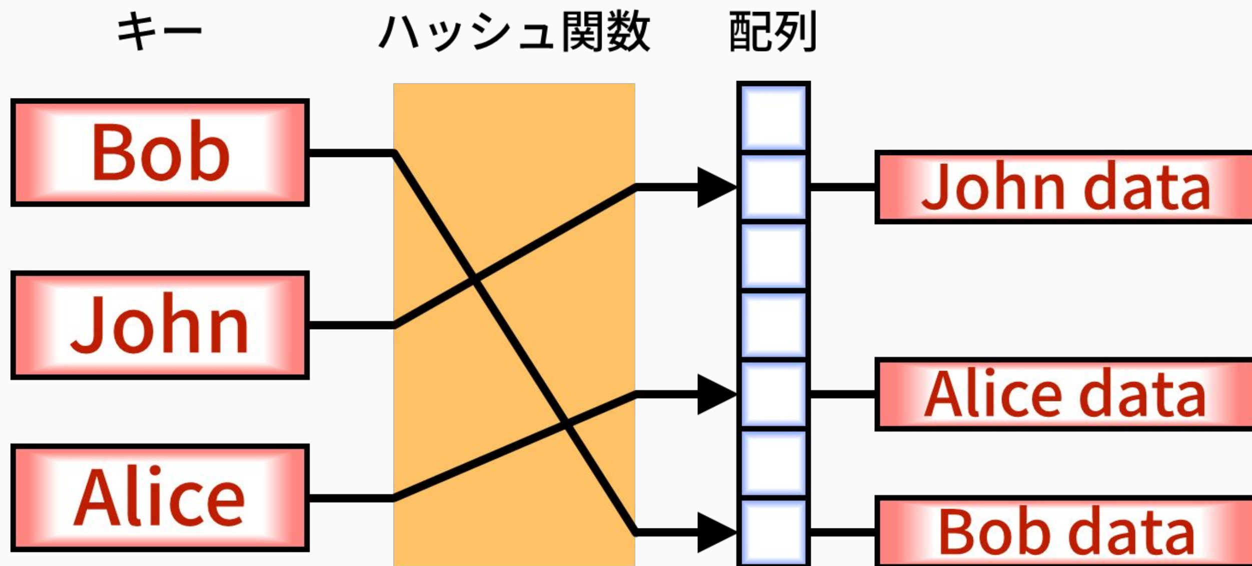


複雑なデータ構造

ハッシュテーブル (hash table)

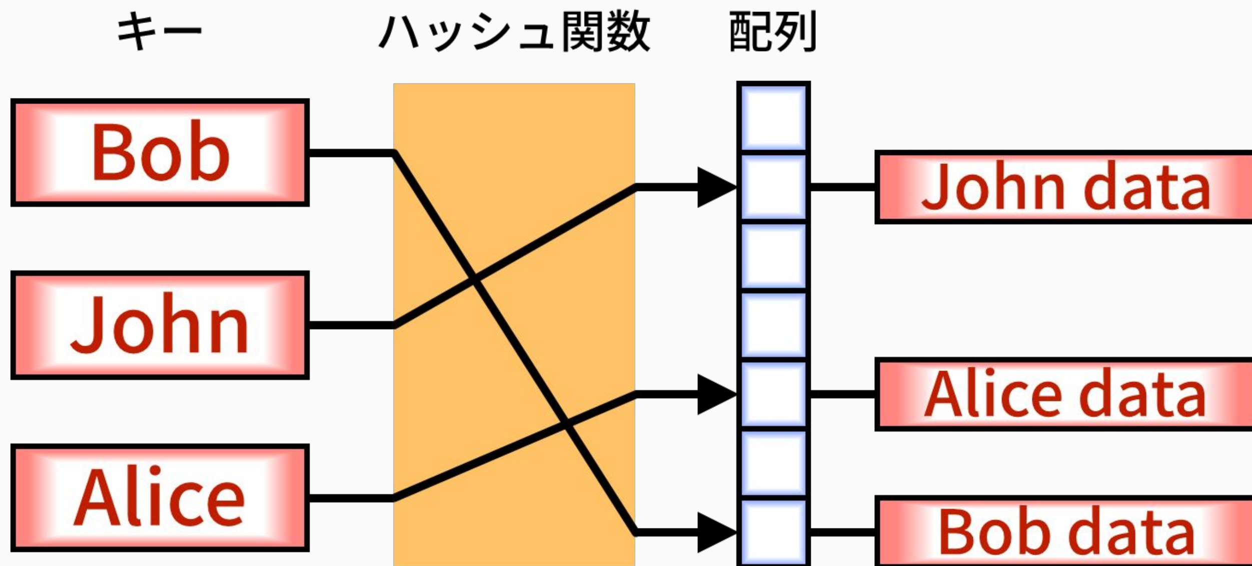
ハッシュテーブル (hash table)

- Go の map、C# の Dictionary で使われる構造 (Javaは知らん)
- 整数以外のインデックスを使ってデータを管理するための構造



ハッシュテーブル (hash table)

内部的に配列を持っていて、キーから配列のインデックスに変換する仕組み (ハッシュ関数) を使ってアクセスする



ハッシュ関数 / ハッシュ値

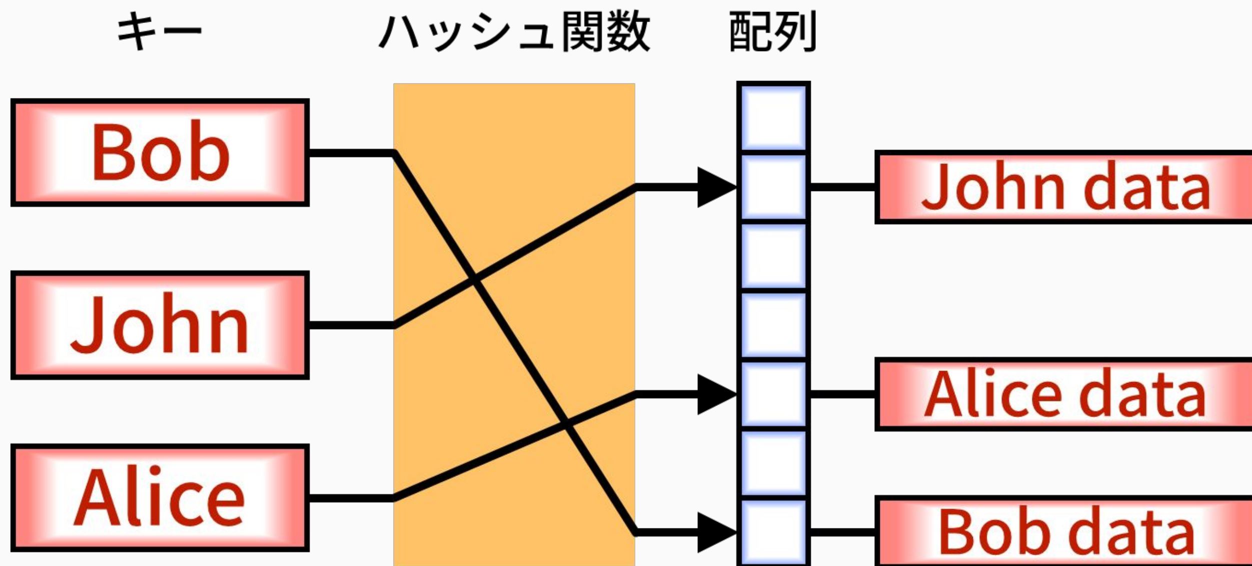
様々な型の値を、別の共通した型(多くの場合は整数)に変換する関数を**ハッシュ関数**、変換後の値を**ハッシュ値**という。

C# の GetHashCode() メソッドはハッシュ関数。

ハッシュ値は**偏りがない**ようにする必要がある。標準機能として用意されている場合はこれを満たすように実装されている。

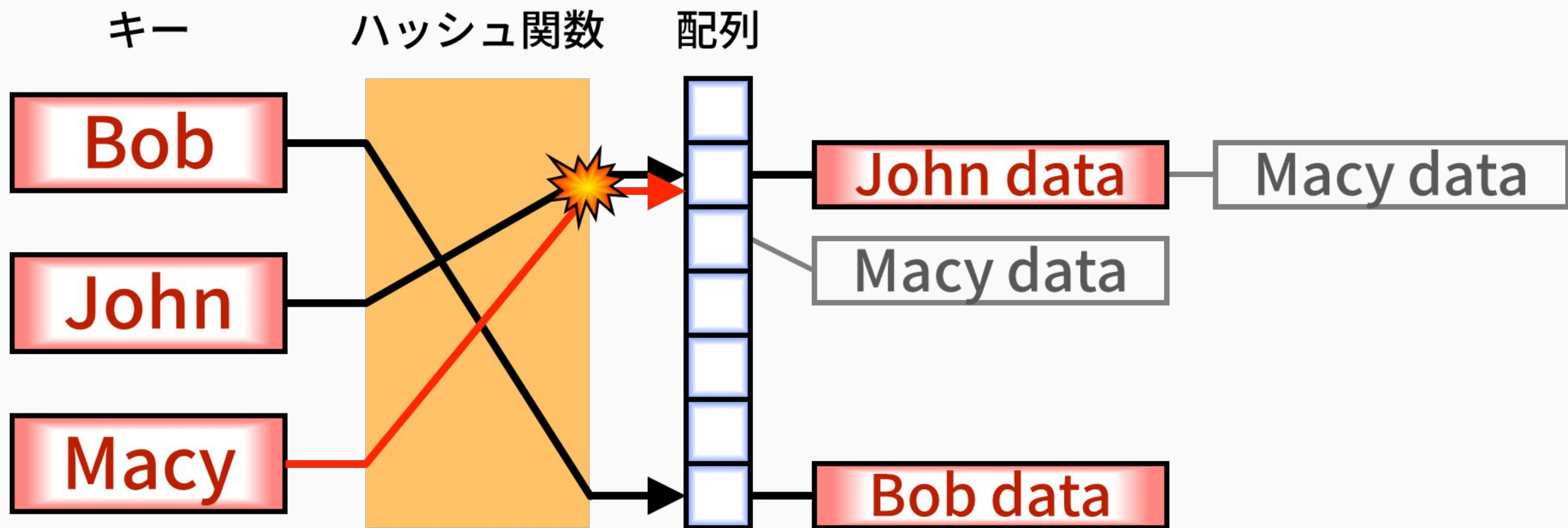
キーから配列インデックスへの変換

キーのハッシュ値をとり、**配列の長さで割った余り**をインデックスとする



キーから配列インデックスへの変換

ハッシュ値を配列の長さで割った余りが重複した場合、次のインデックスにデータを保存するか、連結リストにして後ろにデータを繋げる。



ハッシュテーブルの操作

- データの追加: $O(1)$
- データの検索: $O(1)$
- データの削除: $O(1)$
- 全データ列挙: $O(n)$ であることが多い、実装による

キーによるアクセスを、配列のインデックスと同じ感覚で使っていける

ハッシュテーブルの豆知識

配列長の工夫

ハッシュ値を配列の長さで割った余りはできるだけ重複してほしくないので、**配列の長さを素数にする**ことで余りが重複しにくくなるようにすることが多い。

容量の拡充

配列にデータが収まりきらなくなったら、新しい配列を用意して全データを入れ直す。新しい配列の長さは、元の配列の長さの2倍より少し大きい素数を使うことが多い。この操作は$O(n)$である。

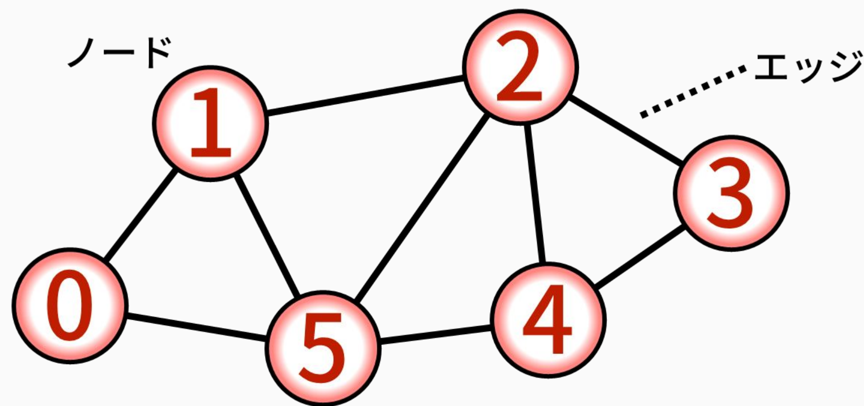
グラフ構造 (graph)

グラフ構造 (graph)

双方向連結リストと似ているが、各ノードが自由な数のノードと連結している構造

扱いが難しいので、ぶっちゃけあまり使わない

- ノード : データを格納する場所
- エッジ : ノード同士の繋がり



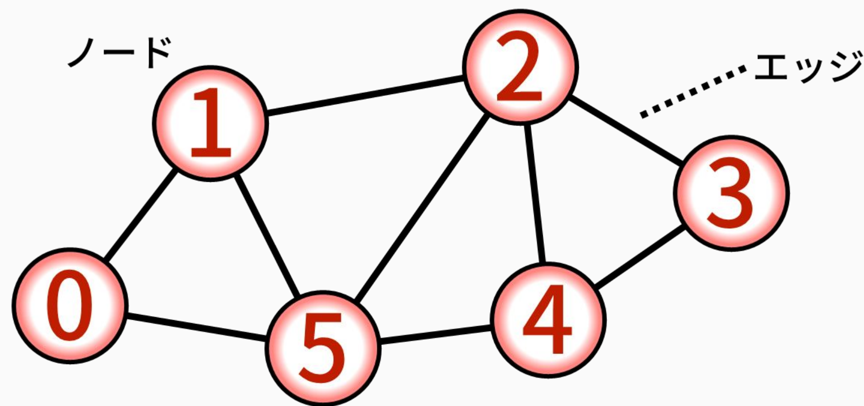
グラフ構造の実装と表現

隣接リスト

各ノードが「自身とエッジで繋がっているノード」へのポインタを持つ

隣接行列

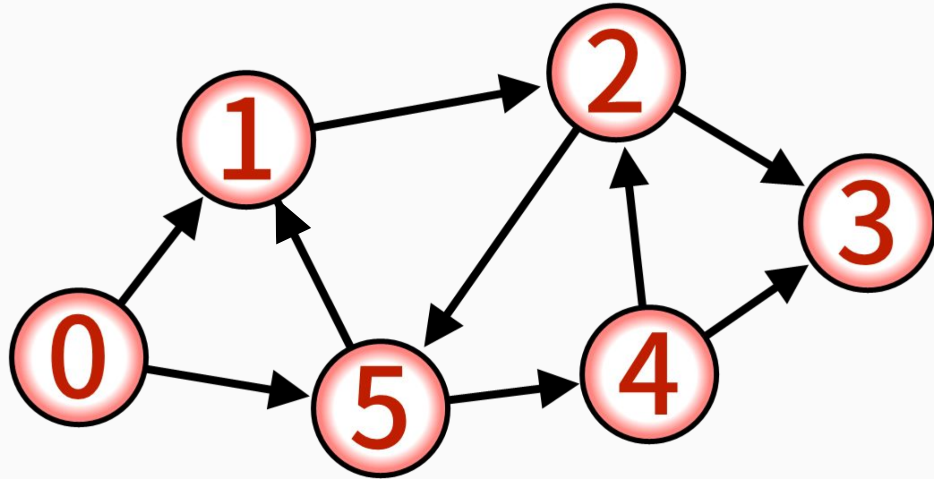
ノード数 × ノード数サイズの2次元配列を用意し、各要素に「行と列で表されるノード同士が繋がっているか」を示す値を格納する



有向グラフ (directed graph) / 無向グラフ (undirected graph)

有向グラフ : エッジに向きがある。

無向グラフ : エッジに向きがない。



グラフ構造の用途

正直「グラフ構造を使えば問題が解決できる！」
って場面はあまりない…

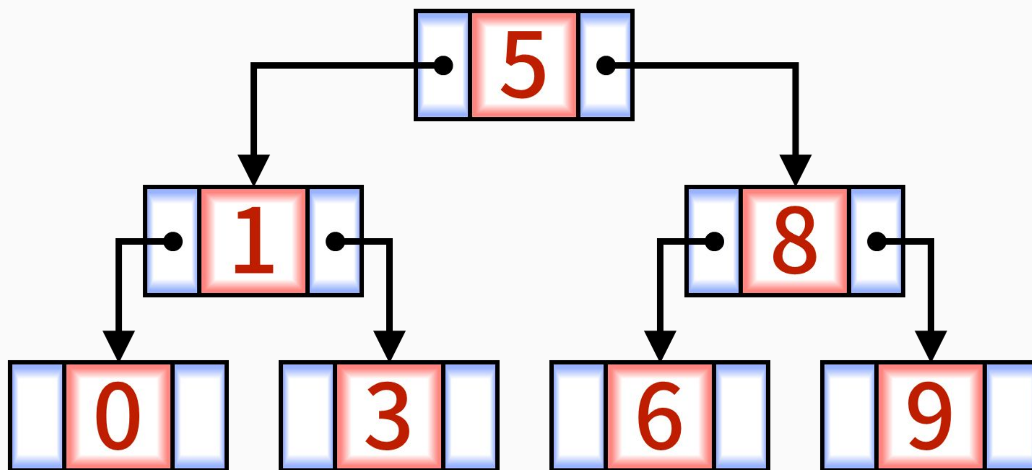
実装も複雑になりやすいので、可能なら木構造を使う

最近流行りの機械学習には、どうやらグラフ構造を使って
いるものがあるらしい…

木構造 (tree)

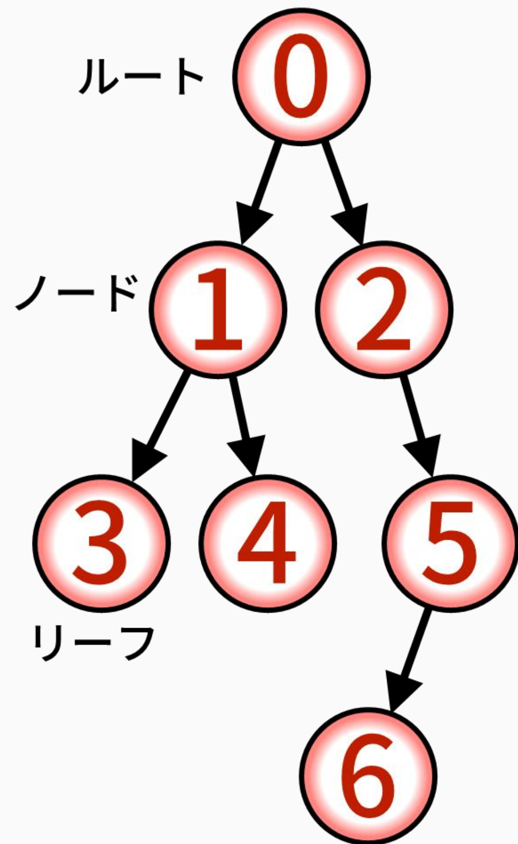
木構造 (tree)

- 次々に枝分かれしていくデータ構造
- 各ノードが関連するノードへのポインタを持つ(グラフ構造と違ってループは許されない)
- 単方向連結リストの派生系で、「次のノードへのポインタ」を複数持てるようにしたもの



木構造 (tree)

- 親ノード: あるノード**を**参照しているノード
- 子ノード: あるノード**が**参照しているノード
- 先祖ノード: 再帰的な親ノード
- 子孫ノード: 再帰的な子ノード
- ルートノード: 親ノードを持たないノード
- リーフノード: 子ノードを持たないノード
- ルートノード以外のノードは必ず親ノードを1つだけ持つ
- あるノードの子ノードはいくつあってもよい



木構造の用途

- JavaScript で HTML を扱う際のドキュメント**ツリー**は木構造
- ディレクトリ構造など、コンピュータには木構造になっている概念が結構あったりする
- 事前に個数が分からないデータのソート (後述)

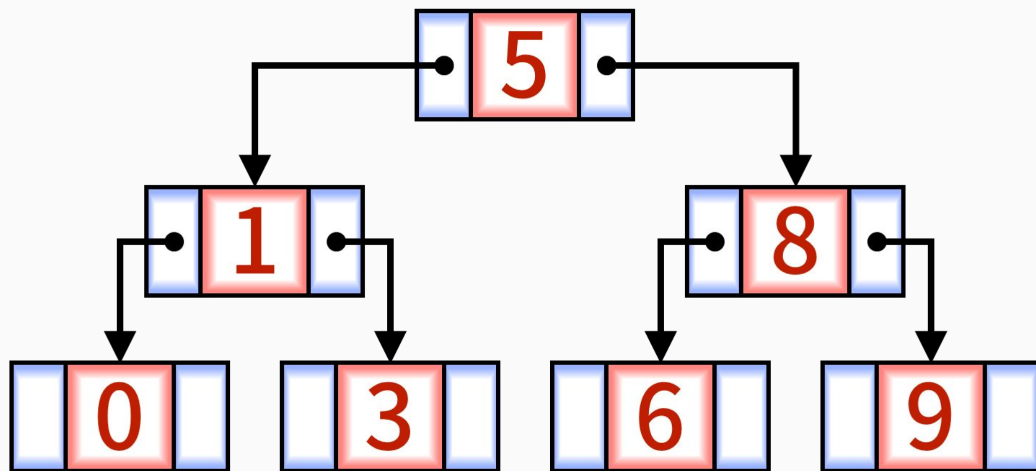
二分探索木 (binary search tree)

二分探索木 (binary search tree, BST)

各ノードが最大で2つの子ノードを持ち、それらのデータが

左子孫ノードの値 ≦ 親ノードの値 ≦ 右子孫ノードの値

のような順序になっている木構造を**二分探索木**という。



二分探索木の各種操作

格納しているデータ数を n とする。

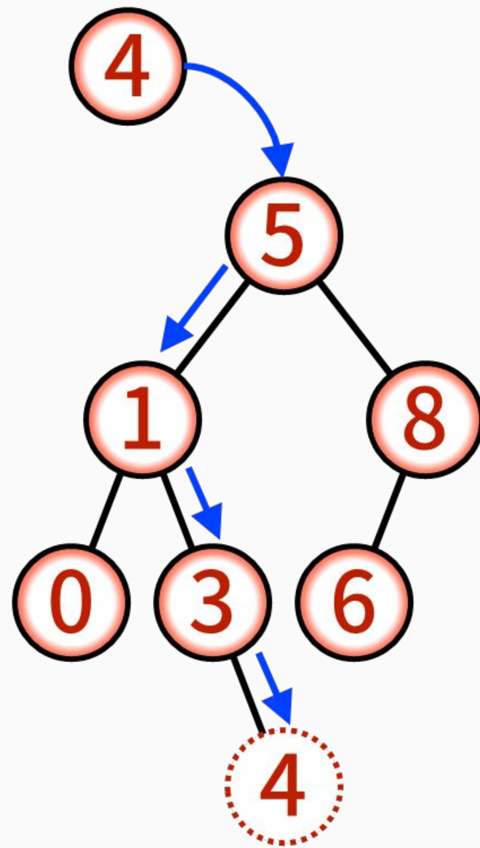
- データの追加: $O(\log n)$
- データの検索: $O(\log n)$
- 全データ列挙: $O(n)$

全体的に優秀

二分探索木へのデータ追加

右図の木構造に4を追加する場合

- ルートノードである5から探索開始
- 4 < 5 なので、5の左子ノードに追加したい→既存の子である1に注目する
- 4 > 1 なので、1の右子ノードに追加したい→既存の子である3に注目する
- 4 > 3 なので、3の右子ノードに追加したい→子がいないので、ここに追加する

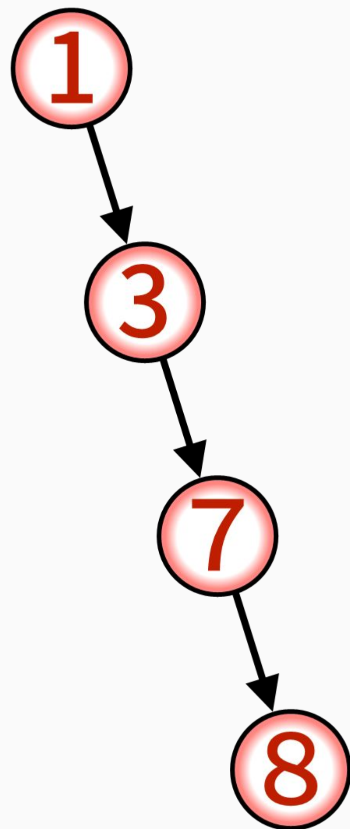


二分探索木へのデータ追加

追加アルゴリズムの問題点

データを小さいほうから順に追加していった場合、延々と木が右に伸びていく**非効率な二分探索木**ができる。

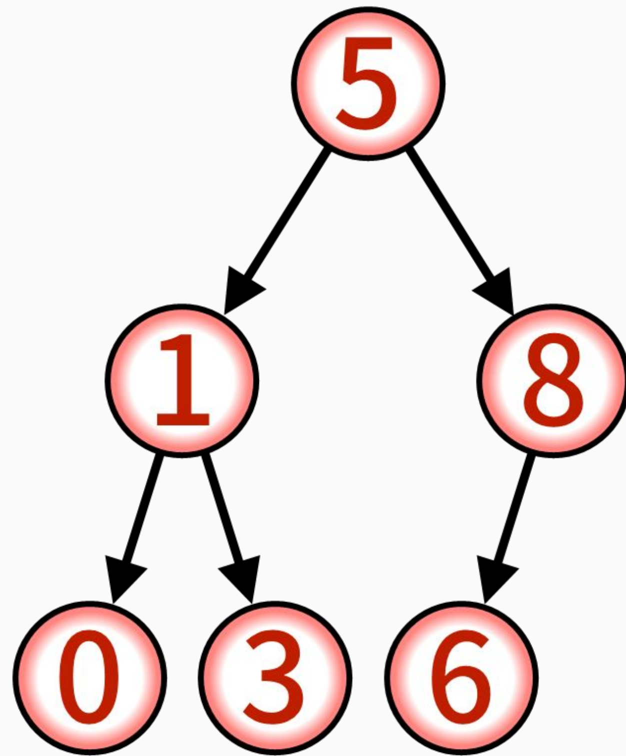
このような構造になるとデータの追加と検索の計算効率が悪くなり、最悪で$O(n)$になる。



二分探索木の全データ列挙

1. ルートノードである5から処理開始
2. 注目しているノードの**左**子ノードのデータを再帰的に(1〜4の手順で)列挙する
3. 注目しているノード**自身**のデータを出力する
4. 注目しているノードの**右**子ノードのデータを再帰的に(1〜4の手順で)列挙する

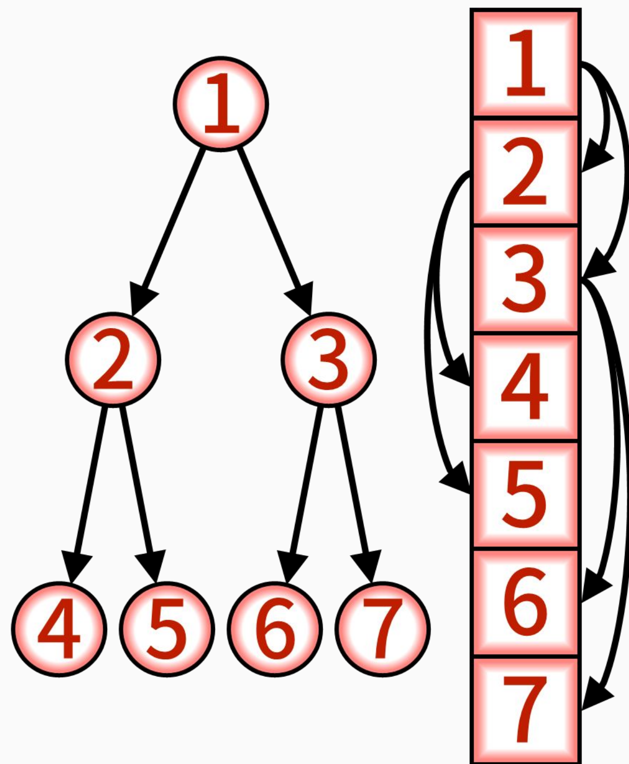
この手順で、データが**昇順に**列挙される。



二分探索木の実装・表現

二分探索木はノードとポインタで実装してもよいが、単なる配列に次のルールを設けることで実装・表現できる。

- 1番目の要素をルートとする
- n番目の要素の子ノードは$2n$番目と$2n+1$番目



二分探索木の派生

赤黒木

各ノードに赤・黒という色をつけてバランスを保つアルゴリズムを組み込み、二分探索木の「データ追加順によって効率が悪くなる」問題を解消したもの

B木, B+木, B*木

木構造を改良してまとまったメモリ領域を効率よく使えるようにしたもので、**RDBのインデックス保存に使われる**。

赤黒木

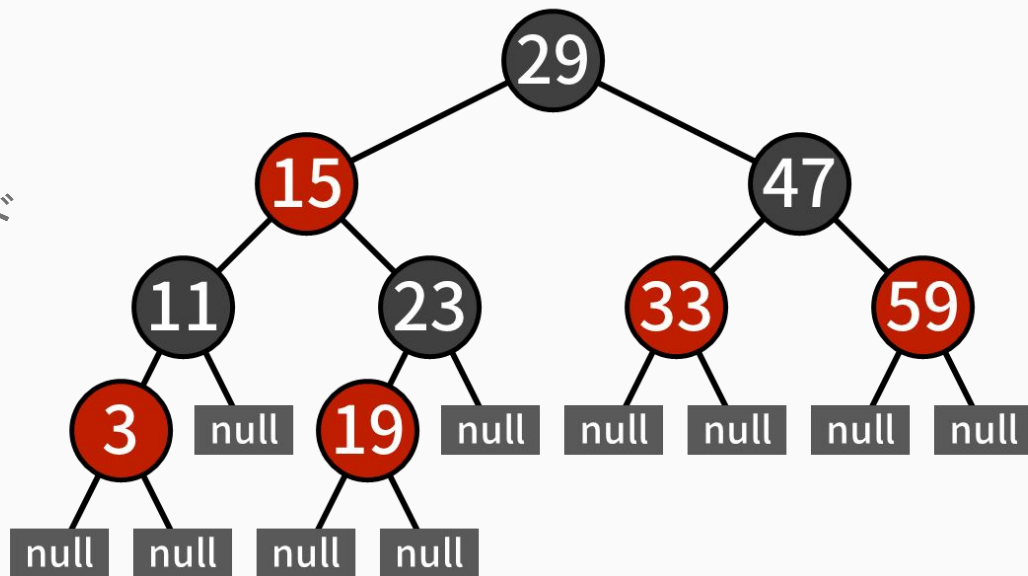
- 二分探索木を改良し、データの追加・検索の**最悪計算量**を$O(\log n)$にしたもの
- 通常の二分探索木の構造に加えて、各ノードが**赤**または**黒**の「色」を持つ
- ノードの色を使って偏った木になるのを防ぐ
- 要素の探索や列挙は二分探索木と同じ

C# の SortedSet<T> が赤黒木で実装されている。

赤黒木の色が満たすべき条件

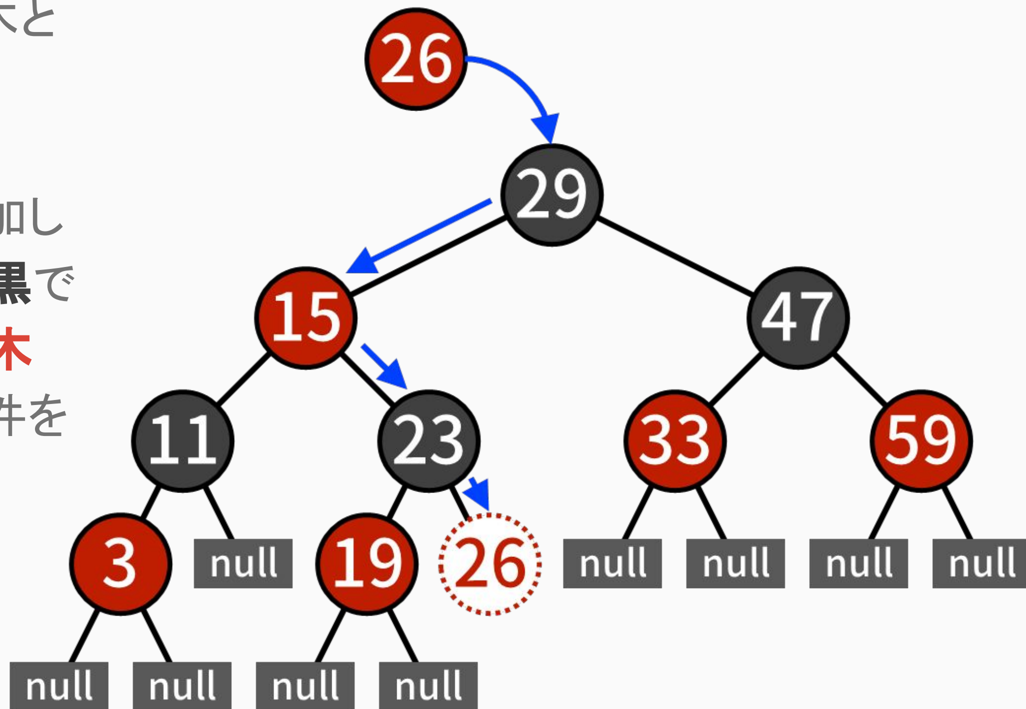
1. ルートノードは**黒**である
2. リーフノードは全て**黒**である
3. **赤**ノードの子はすべて**黒**である
(**赤**ノードの親は必ず**黒**である)
4. 全てのリーフノードからルートノードまでの道には、同じ数の**黒**ノードがある

(リーフノードはデータを持たない)



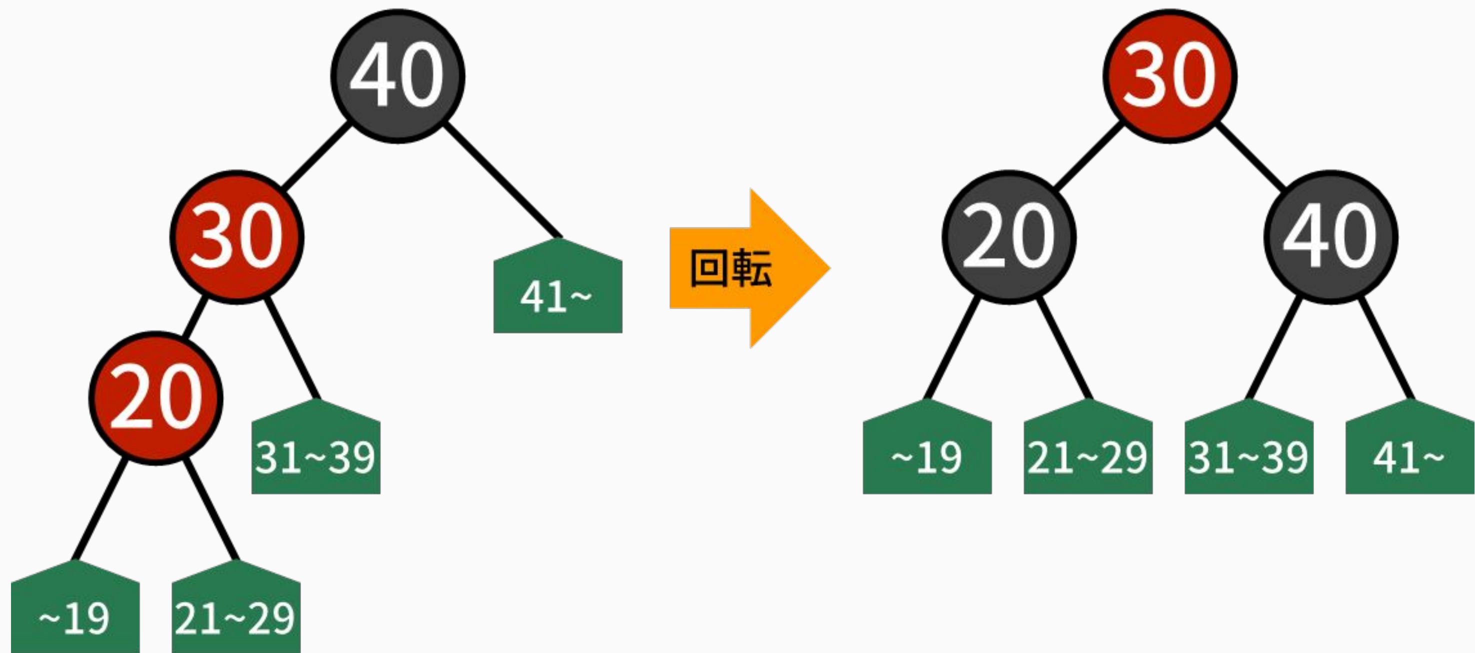
赤黒木へのデータ追加

- 新しいデータは通常の二分探索木と同じアルゴリズムで追加する
- 新しいデータは常に**赤**
- **赤**ノードの子に新しいデータを追加した場合は「**赤**ノードの子はすべて**黒**である」条件を満たさなくなるので、**木構造の回転**を行って赤黒木の条件を満たすように修正する



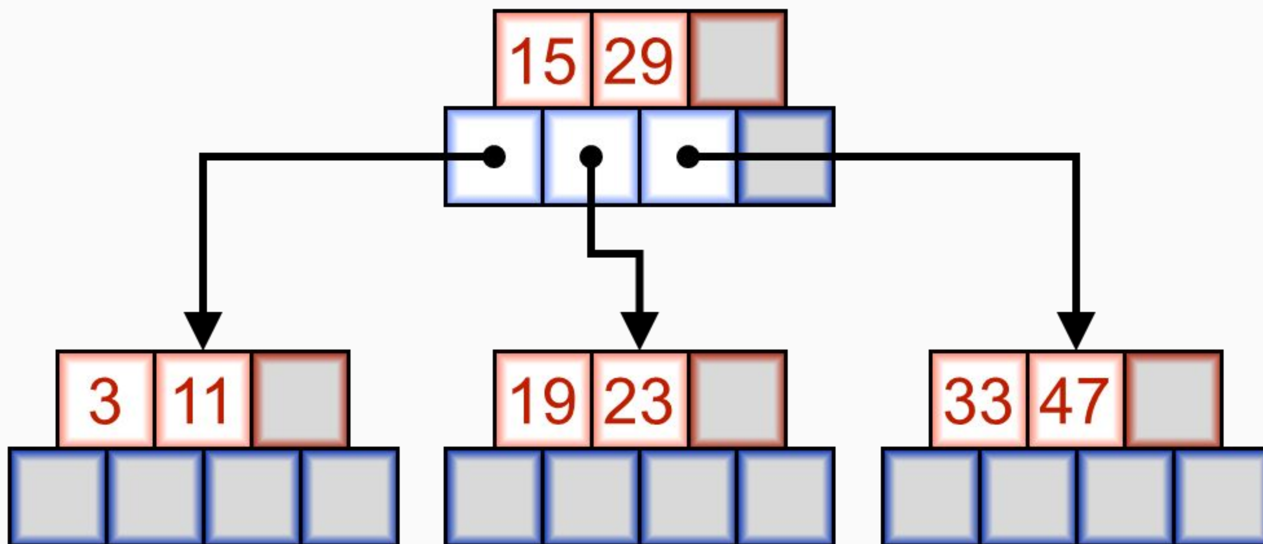
木構造の回転

二分探索木やそこから派生した木構造は、条件を崩さないように木構造を変形する**回転**という操作ができる。赤黒木は回転を使って常にバランスを保つ。



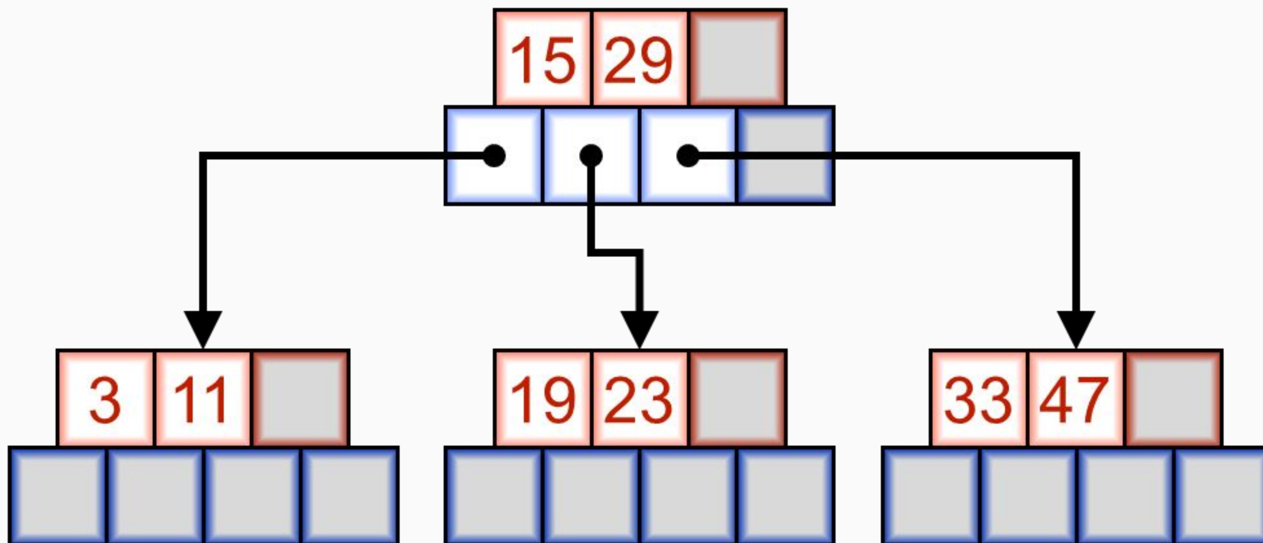
B木 (B tree)

基本的に二分探索木と同じ考え方で、1ノードがm個のデータと$m+1$個の子ノードを持つようにした探索木。赤黒木と類似したアルゴリズムを組み込んだ**B+木**が **RDB（リレーショナルデータベース）のインデックスに使われている。**



B木

- ひとつのノードが持つ値は右のほうが大きい
- 15 の左にある子孫ノードの値は全て $x < 15$
- 15と29の間にある子孫ノードの値は全て $15 < x < 29$



書籍紹介

『アルゴリズムとデータ構造』

競技プログラミング経験が豊富な著者が「アルゴリズムを自分の道具にしたい」という読者に向けて執筆。

前回・今回の講義に含まれるデータ構造やアルゴリズムもだいたい載っている。