

## 3.1 データ構造とアルゴリズム

- 基本概念
- 代表的なデータ構造
- 代表的なアルゴリズム
- 補足



# 目次

- 1 基本概念
- 2 代表的なデータ構造
- 3 代表的なアルゴリズム
- 4 補足

# データ構造とは

- **データ構造**[Data Structure]とは、コンピュータ上のデータを保存、整理、管理するための形式です。
- 同じデータを異なる方法で保存できます。データ構造を選択する場合、以下の要素を考慮することが最も重要です：
  - データの**保存方法**は？ 例えば、データの順番は大事ですか？ データの間に何らかの相関性はありますか？
  - どんな**操作**（機能）が必要ですか？ 例えば、あるデータが保存されていますかどうかを確認することは可能ですか？ データが削除できますか？ データの並べ替えは可能ですか？
  - これらの操作の**効率**はどのぐらいですか？ すなわち、時間と空間のリソースをそれぞれどの程度消費していますか？

次へ 

 前へ

- 例えば、クラスの生徒のファイルを保存するために、次の二つの方法は考えられます：一つ目は、全てのファイルをランダムにテーブルに並べます。二つ目は、ファイルを学籍番号順にフォルダに入れます。
- それぞれの方法について、先程説明した 3 つの要素を考えましょう：
  - 保存方法：一つ目の方法は、個々のファイルの間に関係がありません。二つ目の方法は、ファイルを順番に保存します。
  - 操作：どっちの方法も、ファイルを預ける、ファイルを取り出す、ファイルを検索、などが可能です。
  - 効率：一つ目の方法では、ファイルの預け入れは速いですが、取り出しが遅いです。二つ目の方法では、ファイルの預け入れは遅いですが、取り出しが速いです。

# 抽象データ型

- 抽象データ型[Abstract Datatype]は、データの格納方法と操作の種類のみを考慮します。
- 抽象型が決まつたら、それを様々なデータ構造で実装できます。どのデータ構造を選択するかは、これらのデータ構造が実装した操作の効率によってで決めることがあります。
- データ構造を決める一般的なプロセスは：
  1. 必要な機能に基づいて、どの抽象データ型を使用するかを決定。
  2. どんなデータ構造が、その抽象型を実装できるかを複数特定。
  3. 実際の操作効率の必要性に基づいて、最終的にどのデータ構造を使用するかを決定します。

次へ 

 前へ

- 例えば、ブログサイトにある、ユーザーの全てのブログを保存したい際に、どのデータ構造を使うかを決めましょう：
  1. 抽象データ型： ブログを時間順に保存する必要があります。最新のブログを預け、任意のブログを閲覧する、ブログを削除するなどの操作が必要です。そこで調べてわかったのは、「リスト」タイプではこれらの機能を実現できます。したがって、リストの使用に決定。
  2. データ構造の候補： リストを実装できるのは、「配列」と「連結リスト」の二つのデータ構造があると、調べてわかりました。
  3. データ構造の決定： ユーザーがブログを閲覧する行動をよく使っていることがわかりました。この操作は、連結リストよりも配列の方が効率的なので、最終的に配列を使います。

# アルゴリズム

- アルゴリズム[Algorithm]は、問題を解決するために使用される一連の操作手順を記述したものです。
- 広義には、レシピや家電の設置説明書など、問題を解決するための具体的な手順を記述したものをアルゴリズムと呼ぶことができます。私たちを学ぶアルゴリズムとは、問題を解決するためのコンピュータ・プログラムの手順を意味します。
- アルゴリズムとデータ構造には密接な関係があります:
  - 正しいデータ構造を選択することで、アルゴリズムの効率を向上させることができます。
  - 一方、データ構造自体の運用には、それを実現するためのアルゴリズム設計が必要です。

# アルゴリズムとプログラム

- アルゴリズムとプログラムの違い: アルゴリズムは、問題解決のステップを抽象的に記述すればよいです。そして、プログラムはこれらの手順を対応する言語のコードで具体的に実装する必要があります。
- そのため、アルゴリズムは特定の言語に依存しないことが可能: 同じアルゴリズムを様々な言語で実装できます。

# アルゴリズムの評価基準

- 同じ問題を解決するために、様々な方法があります。異なるアルゴリズムのメリット・デメリットをどう比較しますか？
- 情報科学において、一般的にアルゴリズムの評価基準として使われるのは、そのアルゴリズムがどれだけのリソースを必要とするかということです。通常、リソースは大きく分けて「時間リソース」と「空間リソース」の2種類が考えられます。
- データ構造もアルゴリズムに従って実装されるため、上記の指標はデータ構造の各操作の評価にも利用できます。

次へ 

◀ 前へ

- 時間リソースの評価指標は、**時間計算量**[Time Complexity]といいます。簡単に言えば、時間計算量が高ければ高いほど、アルゴリズムの実行速度は遅くなります。
- 空間リソースの評価指標は、**空間計算量**[Space Complexity]です。空間計算量が高いほど、アルゴリズムが実行するために使用する記憶空間（メモリとか）が多くなります。
- これらの計算量は、**オーダー**[Order]という、数値のレベルによって評定されます。

次へ ➞

# 一般的な計算量のオーダー

- 次の表は、一般的な計算量のオーダー（低いものから高いものまで）を示しています。どれが高いのかどれが低いのかを知っていれば十分で、定量的に理解する必要はありません。

名称	記法	よく見る書き方
常数時間（空間）	$O(1)$	$O(1)$
対数時間	$O(\log n)$	$O(\log n)$ 、 $O(\log(n))$
線形時間	$O(n)$	$O(n)$
線形対数時間	$O(n \log n)$	$O(n \log n)$ 、 $O(n \log(n))$
二乗時間	$O(n^2)$	$O(n^2)$
多項式時間	$O(n^c)$	$O(n^c)$ 、 $O(n^k)$
指数時間	$O(2^n)$	$O(2^n)$ 、 $O(c^n)$



*Question and answer*

## Coffee ☕ Break

## 計算量と O-記法 (1)

なぜアルゴリズムの評価に計算量を利用するのか? アルゴリズムは特定の言語やマシンに依存しないため、実際の実行時間（空間）や実行したコード量などで直接評価することは困難です。また、ある簡単なテスト状況でアルゴリズムが優れっていても、状況がより複雑になった時に、その優位性が保たれることが保証できません。

計算量とは、単にリソースを消費する量ではなく、問題の規模が大きくなるにつれて消費量が変化する速度を評価するものです。言い換えれば、複雑性の低いアルゴリズムは、問題サイズが大きくなっても、リソースにそれほどコストをかけないです。

## Coffee ☕ Break

## 計算量と O-記法（2）

アルゴリズムの正確な計算量を計算するのは非常に難しいことが多い、結果が複雑すぎて比較できません。そのため、一般的には、統計数学の漸近的表記法を用いて、計算量の概算を計算して表します。

この記法は、**ランダウの漸近記法**[Landau Notation]といい、 $O(g(x))$  という形で結果の大雑把のレベルを表現します。**O-記法**[Big-O Notation]とも呼ばれます。



# 目 次

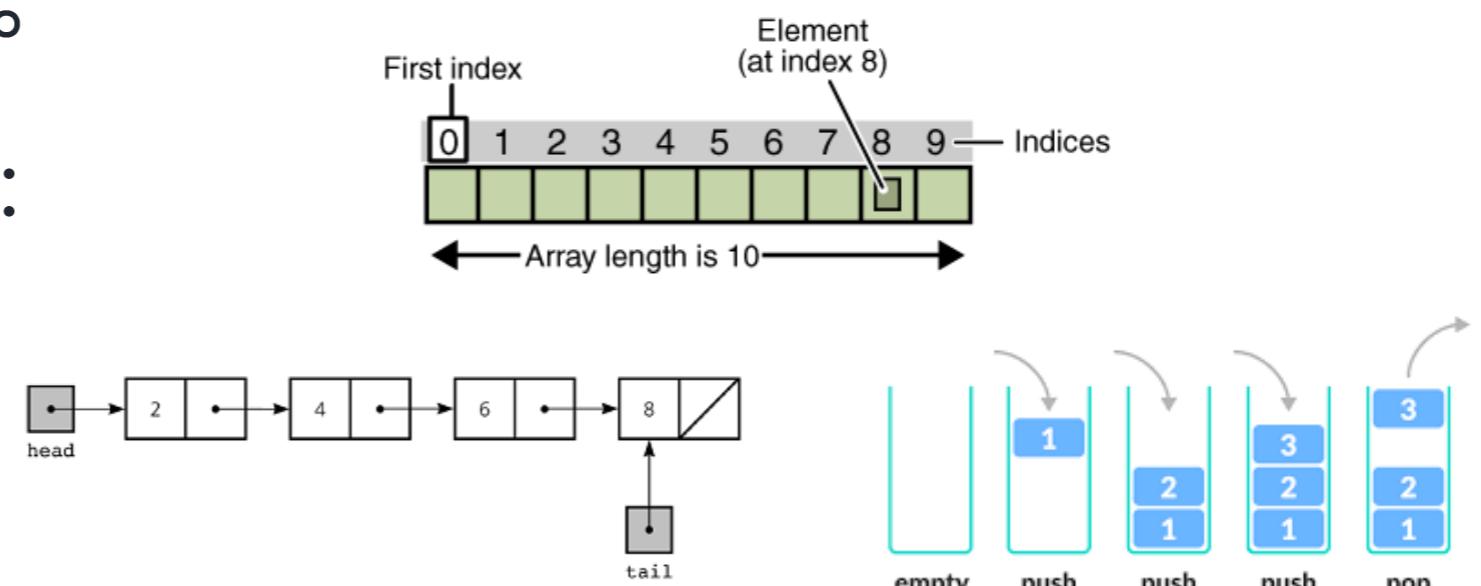
- 1 基本概念
- 2 代表的なデータ構造
- 3 代表的なアルゴリズム
- 4 補足

# 代表的なデータ構造と抽象型

- 実際には、私たちが使用するデータ構造のほとんどは、既にコードとして実装されます。したがって、私たちは、その機能と効率を理解し、システム設計時にどのように選択するかだけを知ればいいです。

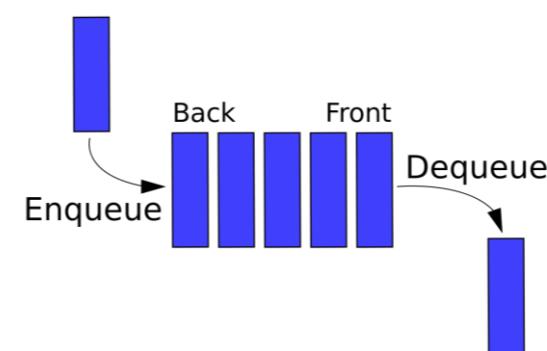
- 一般的なデータ構造は:

- 配列
- 連結リスト
- グラフ、木



- 一般的な抽象型は:

- リスト、スタック、キュー
- セット（集合）
- 連想配列（辞書）

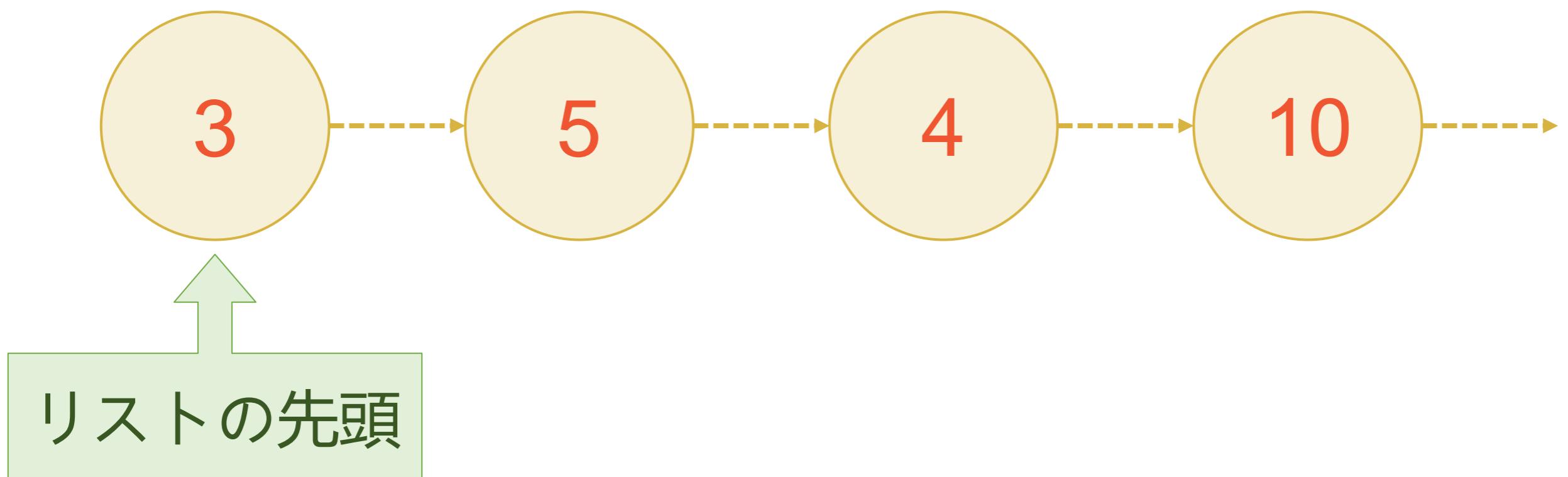


# 配列

- 配列[Array]は最も基本的なデータ構造の一つです。私たちは既に、Javaでその格納方法と機能について熟知したので、その特徴をおさらいしましょう:
  - 格納方法: 配列では、全てのデータが順番に格納され、各データは整数のインデックス(添え字)を持つ。
  - 取得操作: インデックス  $i$  によって  $i$  番目のデータの値を取得。
  - 設置操作: インデックス  $i$  によって  $i$  番目のデータの値を変更。
- コンピュータ言語では、配列の実装は比較的簡単です: 全てのデータをメモリに順次、連続的に格納すればよいです。そのため、ほとんどの言語では既に配列が用意されて、簡単な文法で利用できます。

# 連結リスト

- 連結リスト [Linked List] は比較的基本的なデータ構造です。配列と同様、データを順番に格納するために使用されますが、その方法は若干異なります： 連結リストのデータは、個々のノード [Node] に存在し、各ノードは、次のノードだけにアクセスできます：



# 連結リストの格納方法と操作

- 格納方法：連結リストでは、全てのデータが順番に格納されます。ただし、配列とは異なり、データにはインデックスが付かないが、各データは次のデータがどこに存在するかを知ります。
- 先頭の取得：連結リストの最初のノードを取得。
- 繰り返し：あるノードの次のノードを取得。
- データの取得、設置：ノードに格納されているデータを取得、またはデータの値を変更。
- データ挿入：ノードの次に新しいノードを追加し、データを挿入。
- データ削除：連結リストからあるノードを削除。

# 連結リストの実装

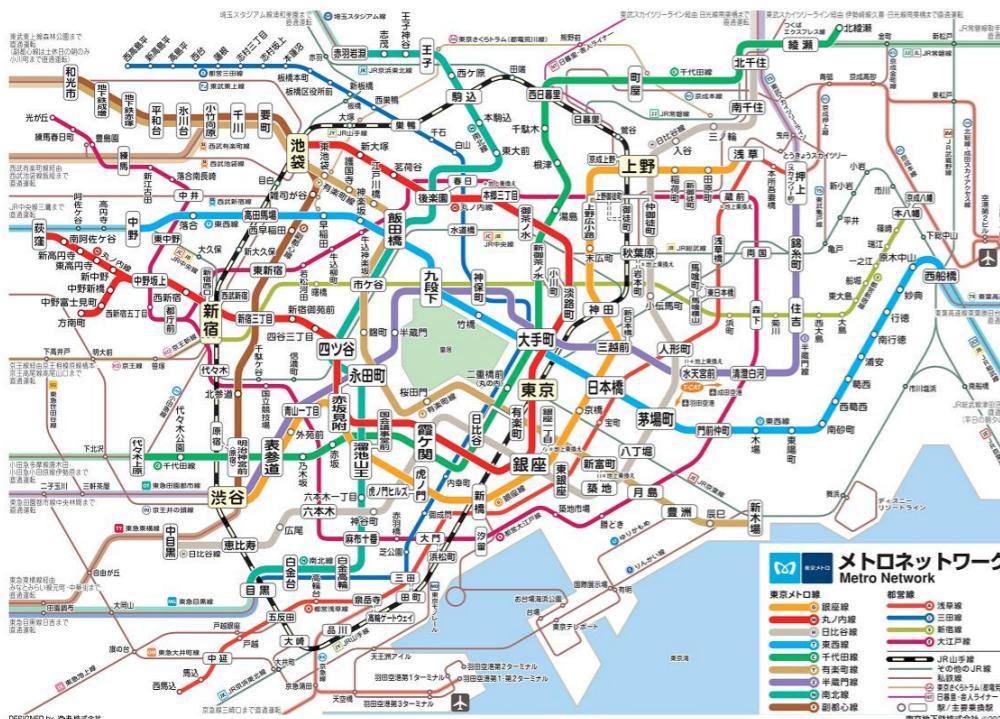
- 多くの言語は連結リストを直接サポートしていませんが、基本構文を使って非常に簡単に実装できます。例えば、C/C++では、「ポインター」を使用して次のノードのアドレスを記録し、連結リストの構造が実現できます。
- Java にはポインターはありませんが、「参照」の概念がそれに似ています。各ノードにデータと次のノードのオブジェクト（参照）を格納することによって連結リストが実現可能。



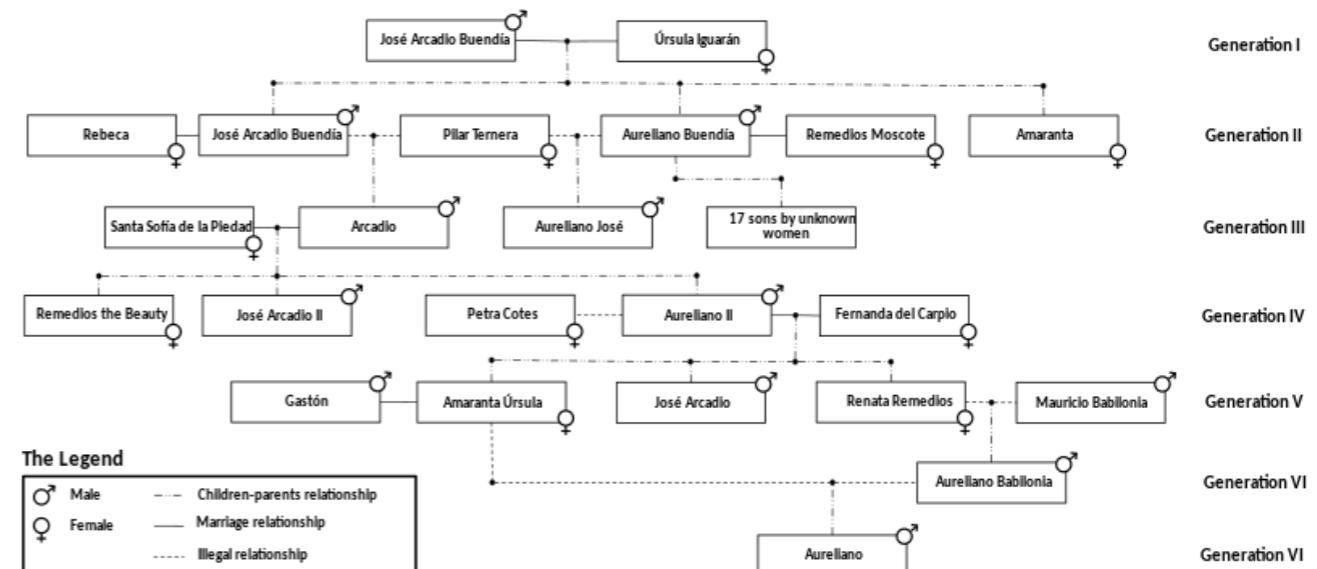
- 実際に使用される連結リストは、双向リスト、循環リストなどの変種が多いです。

# グラフ

- グラフ [Graph] はよく使われるデータ構造です。グラフ内のデータも個々のノードに格納されますが、各ノードはエッジ [Edge] を介して他の複数のノードと接続しています。日々使われるデータの多くは、グラフで表現できます：



地下鉄路線図



人物関係図

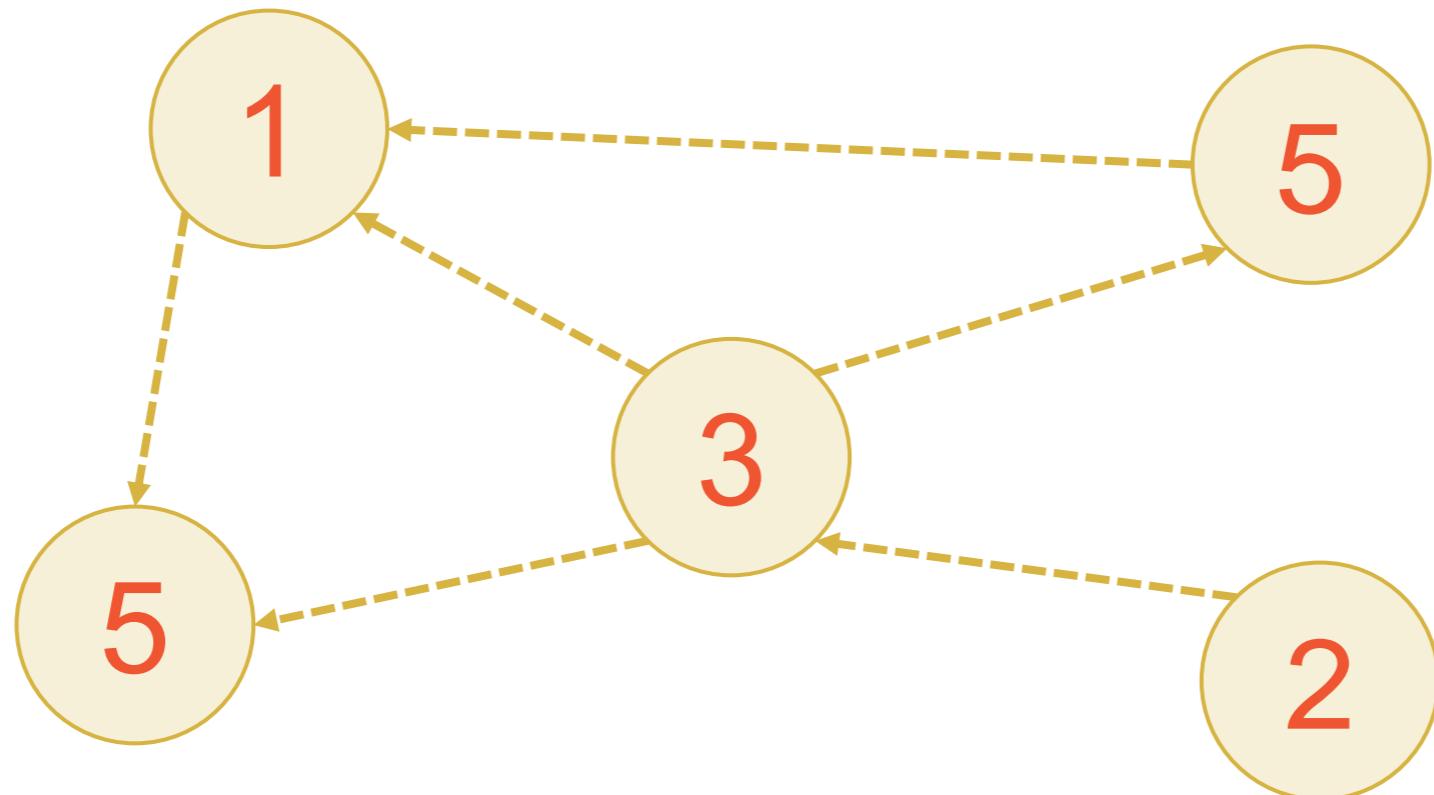
- グラフと図（写真、画像）の区別に注意してください。

# グラフの格納方法と操作

- 格納方法: グラフでは、データ間に順次的な関係はないです。データはノードごとに存在し、各ノードは自分に隣接するノード（隣接ノード [Neighbour]）にアクセスできます。
- 隣接ノードの取得: ノードの隣接ノードを取得します。
- データの取得、設置: ノードに格納されているデータを取得、またはそのデータの値を変更。
- ノードの追加、削除: ノードをグラフに追加、またはグラフから削除。
- エッジの追加、削除: エッジの追加または削除によって、ノード間の関係を変更。

# グラフの実装

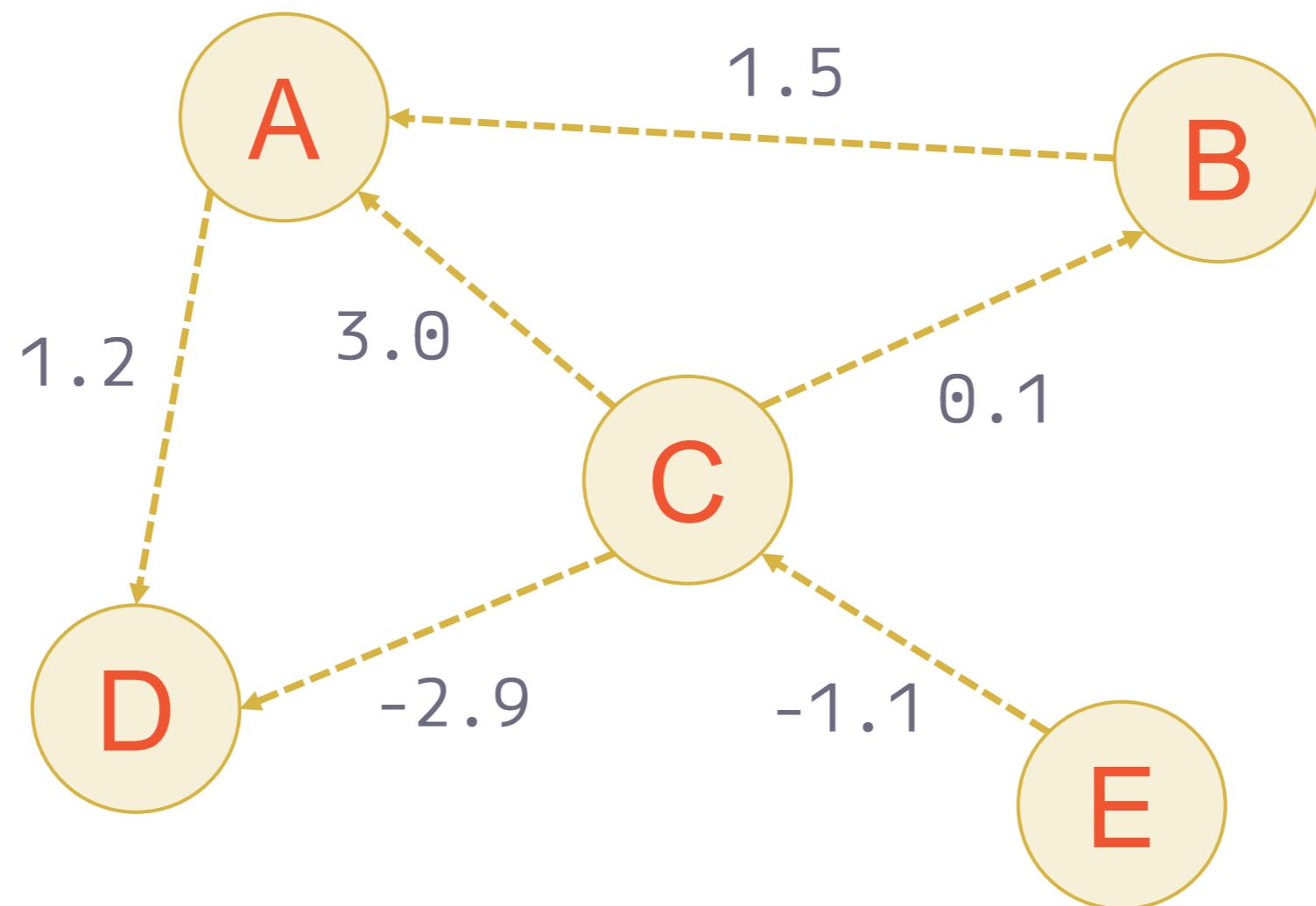
- Java では、グラフの構造もオブジェクトで実装できます：各ノードがデータと隣接ノードを保存。



- この形は、隣接リストという形式に近いです。この他、隣接行列や接続行列など、より多いエッジを持つグラフを格納するのに適した形式もあります。

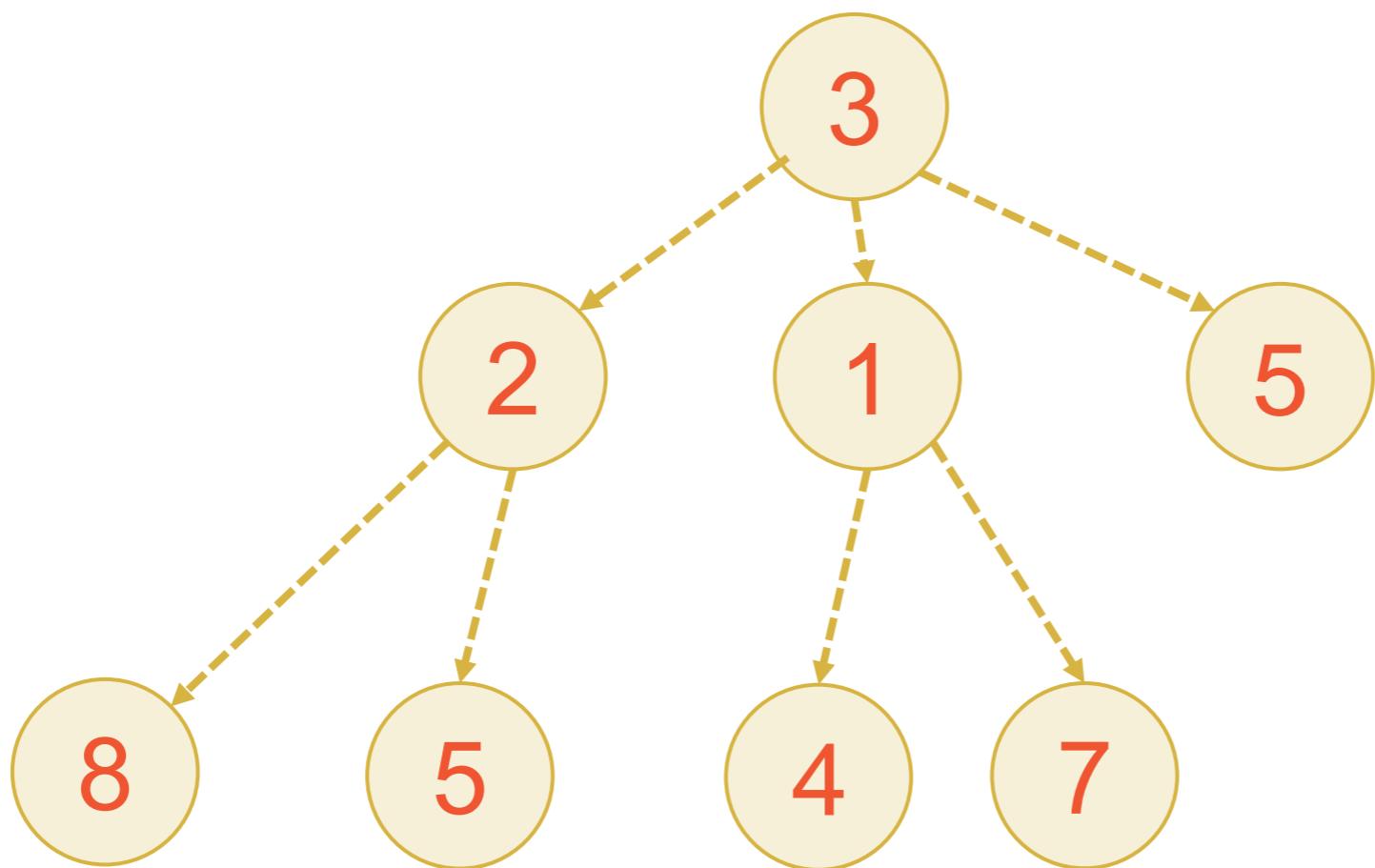
# ネットワーク

- また、グラフの各エッジに1つのデータを記録することも可能で、このようなグラフは**重み付きグラフ**または**ネットワーク**[Network]と呼ばれます：



# 木

- 木[Tree]とは、ループがない特別なグラフです。実際のデータ構造では、一般的に全てのノードがいくつかのレベルに分けられ、各ノードは次のレベルの何個のノードに接続されています：

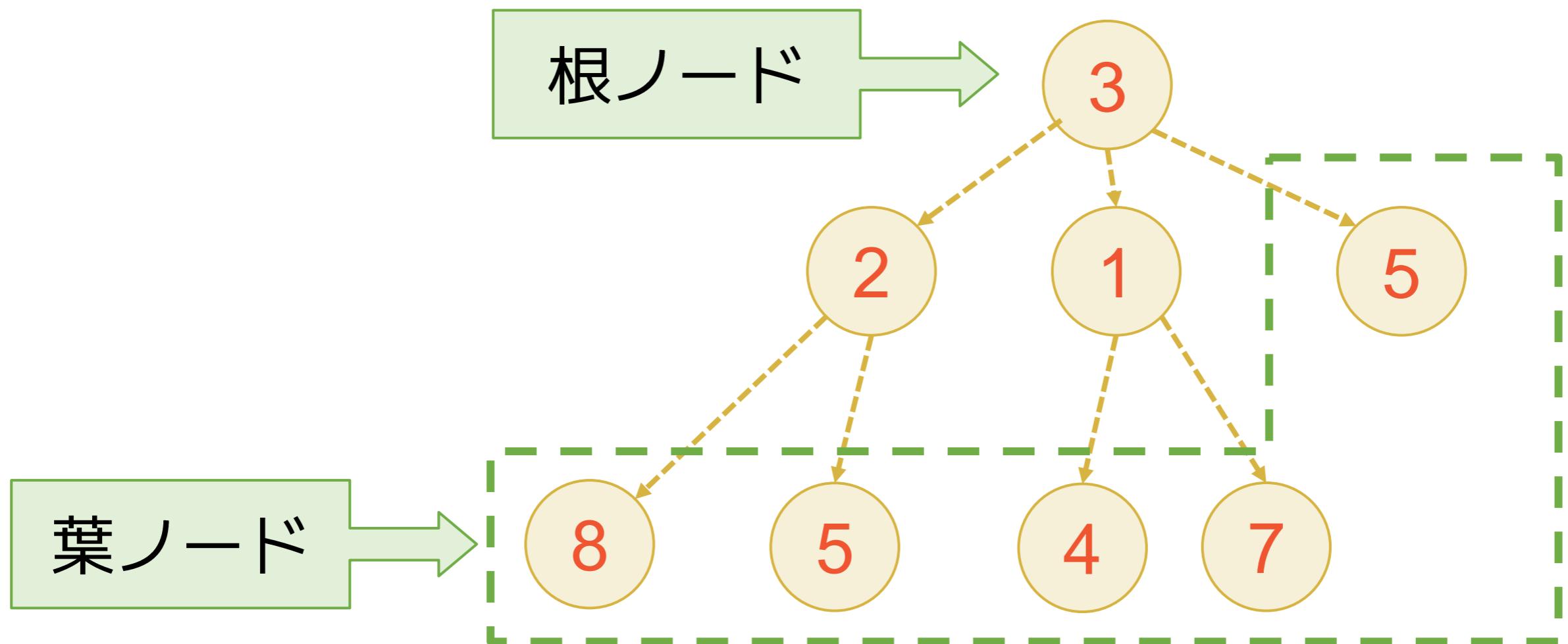


# 木に関する概念

- 一番高いレベルにあるは**根ノード**[Root]と呼ばれ、木の中に1つだけ存在します。
- 各ノードが接続する次のレベルのノードを**子ノード**[Child]と呼びます。各ノードは自分の子ノードの親ノード[Parent]です。各ノードには親ノードが1つだけ存在します。
- 自分と同じ親を持つノードを**兄弟ノード**[Sibling]といいます。親、または親の親、または親の親の親.....などのノードは自分の**祖先ノード**[Ancestor]といい、その逆は**子孫ノード**[Descendant]といいます。
- 子ノードを持たないノードを**葉ノード**[Leaf]と呼びます。

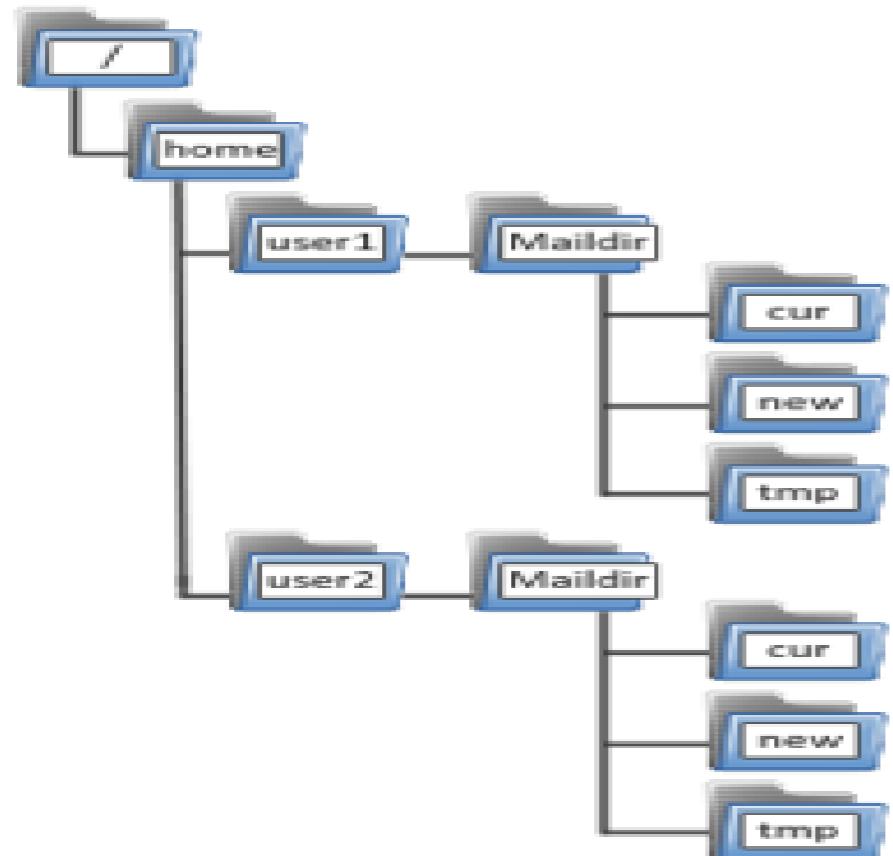
次へ 

◀ 前へ



# 木の応用

- 木をどう実装するかは当分考えなくてもよいですが、実は日常生活で、フォルダシステムなど、既に似たような構造を使っているところが多くあります：



Tips 

「ルートディレクトリ」  
「子フォルダ」等の言葉  
を聞いたことがあります  
か？ いまなら、これらの  
名前の意味は分かるで  
しょうか。

- 木構造の他の使い処を考えましょう。



*Question and answer*

# リスト

- **リスト**[List]は最もよく使われる抽象データ型の一つで、単純な順序データを表現します:
  - 格納方法: リスト内のデータは順次に格納。配列のように、各データには、そのデータの数に対応する整数のインデックスが必要。
  - 取得、設置操作: インデックスに基づく位置のデータの取得、またはデータの設定。
  - 任意な挿入、削除操作: リストの任意の位置（インデックスによって）にデータを追加または削除。
  - 特殊な挿入、削除操作: 先頭や末尾にデータを追加または削除。
  - 検索操作: リスト内のデータのインデックスを探し、またはリスト内にないことを確認。

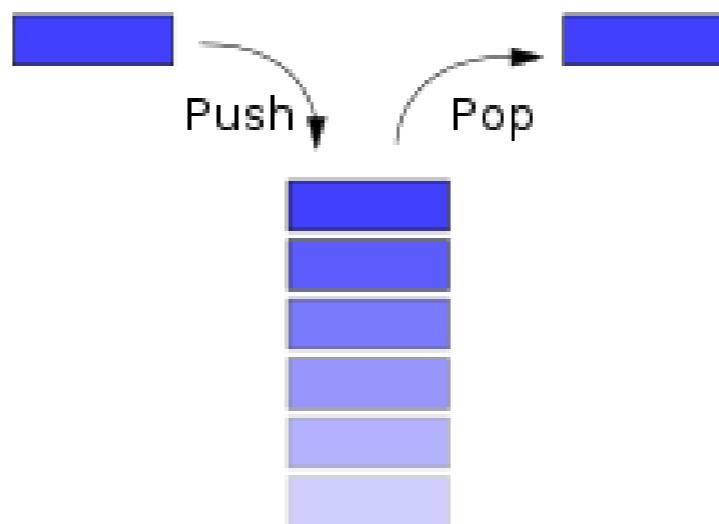
# リストの実装

- リストの実装方法が一般的に二つがあります：配列を使うか、連結リストを使います。選択するために、それぞれどのような操作が効率的なのかだけを覚えておくとよいです：

操作	配列で実装の時間 計算量	連結リストで実装の 時間計算量
取得、設置	$O(1)$	$O(n)$
任意の挿入、削除	$O(n)$	$O(n)$
特殊の挿入、削除	$O(n)$	$O(1)$
検索	$O(n)$	$O(n)$

# スタック

- **Stack** も、データを順番に格納する構造です。しかし、リストとは異なり、スタックは一方の端（先頭<sup>[Top]</sup>）からしかデータの取得、挿入、削除ができません：



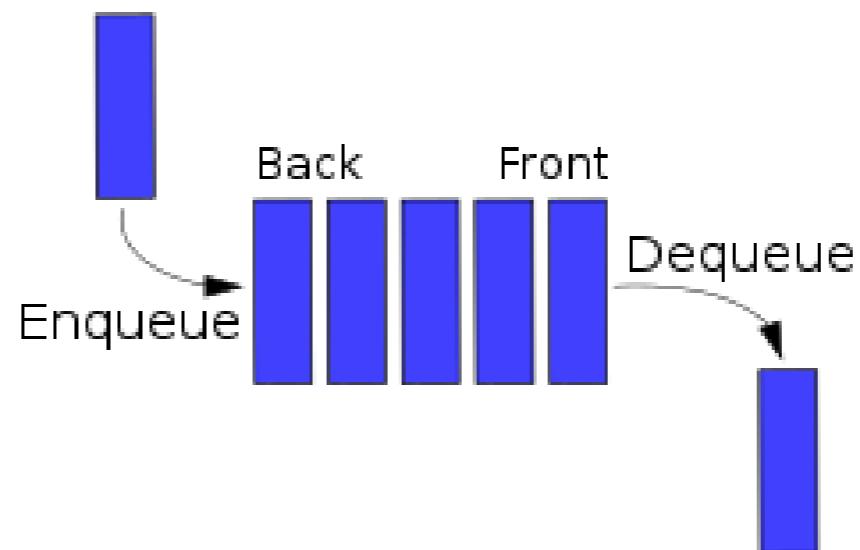
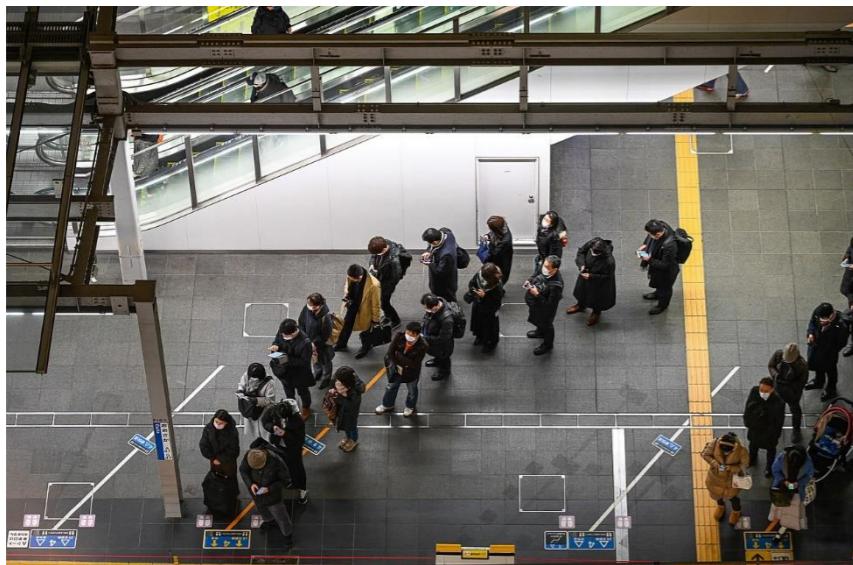
- つまり、スタックの中に保存されるデータは常に、**後入れ先出し**<sup>[Last In First Out, LIFO]</sup>です。

# スタックの操作と実装

- スタックの操作:
  - 格納方法: リストのデータは順次に格納。
  - プッシュ [Push]: スタックの先頭に新しいデータを挿入。
  - ポップ [Pop]: スタックの先頭のデータを取得して使用（このデータはスタックから削除）。
  - ピーク [Peek]: スタックの先頭のデータを取得（削除せず）。
- スタックは一般に配列や連結リンクを使って簡単に実装できます。どちらの実装でも上記 3 つの操作の計算量は  $O(1)$ 。

# キュー

- キュー [Queue] も、データを順番に格納する構造です。キューは、一方の端（末尾）またはからしかデータを挿入できず、もう一方の端（先頭）からしかデータの削除や検索ができません：



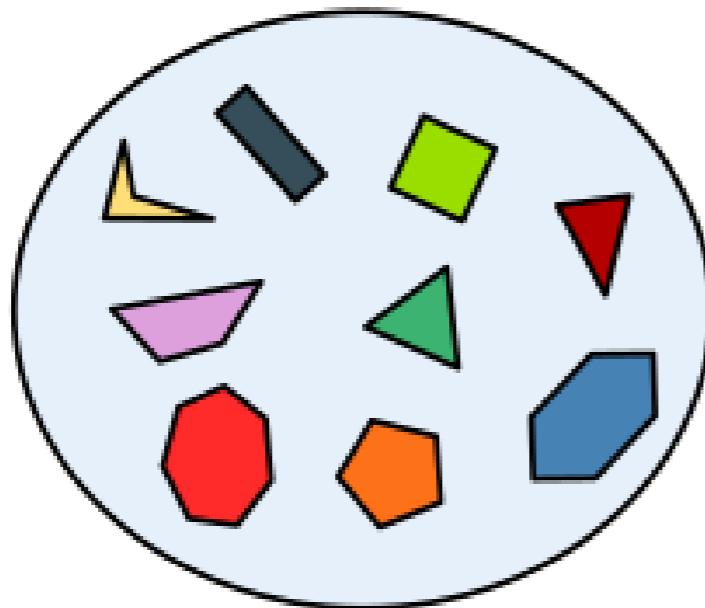
- スタックとは逆に、キュー内のデータは常に、**先入れ先出し** [First In First Out, FIFO] です。

# キューの操作と実装

- キューの操作:
  - 格納方法: キュー内のデータは順番に保存。
  - エンキュー[Enqueue]: キューの末尾に新しいデータを挿入。
  - デキュー[Dequeue]: キューの先頭のデータを取得して使用（このデータはキューから削除）。
  - ピーク: 先頭のデータを取得（削除せず）。
- キューは、配列や連結リストを使って簡単に実装できます。
- 一般的な実装では、上記 3 つの操作の計算量はともに  $O(1)$ 。

# 集合

- **集合**[Set]は、データを順番に並べずに保持する抽象型です。数学の集合と同様に、集合の中に重複する要素はないです：

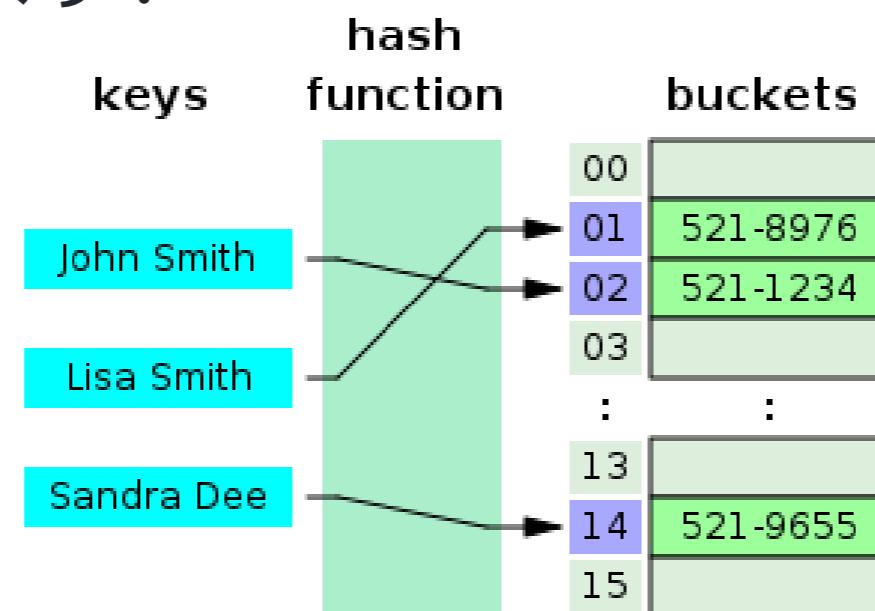

$$\{1, 3, 2, 7\}$$

# 集合の操作と実装

- 集合の操作:
  - 格納方法: 集合の中のデータは順序なしに保存。重複するデータはない。
  - 追加: 新しいデータを集合に追加。
  - 削除: 集合からデータを削除。
  - 検索: あるデータが、集合に含まれているかどうかを確認。
- 集合の一般的な実装として、ハッシュセット [Hash Table] があります。以上の全ての操作の時間計算量は  $O(1)$ 。

# 連想配列

- **連想配列**[Associative Array]、または**マップ**[Map]、**辞書**[Dictionary]は、添え字に従ってデータを格納する抽象型です。配列と異なり、連想配列は**数値以外の型**を添え字として扱えます。例えば、電話帳を保存するとき、連絡先の名前（文字列）を添え字として使用できます：



- ここで、添え字として使用されるデータを**キー**[Key]と呼ばれ、実際に保存されるデータを**値**[Value]と呼ばれます。1つのキーは1つの値に対応し、重複するキーは存在しません。

# 連想配列の操作と実装

- 連想配列の操作:
  - 格納方法: 連想配列のデータは、キーと値のペアで格納します。
  - 取得: キー（添え字）をもとに、対応する値（データ）を取得。
  - 設置: キーに対応する値を変更。
  - 検索: キーが連想配列にあるかどうかをチェック。
  - 削除: 連想配列からキーと値のペアを削除します。
- 一般的な実装は**ハッシュテーブル**[Hash Table]で、上記の操作の時間計算量は全て  $O(1)$ 。ハッシュテーブルの使用があんまりも頻繁のため、ハッシュテーブルと呼んだとき、実は連想配列という抽象型を指していることもあります。



*Question and answer*

## Coffee ☕ Break

## ハッシュテーブルとハッシュ関数（1）

ハッシュテーブルの基本的な実装は、実はまだ配列を使っています。任意の型のキーを整数に変換し、この整数を、連想配列の値を格納する配列のインデックスとして使用することが必要です。

ここでの中心的な問題は、潜在的に大きな範囲を持つ値（例えば、文字列）を、指定された範囲内の整数に変換する方法です。ハッシュテーブルの実装は、大きなデータから整数への適切なマッピング、すなわち **ハッシュ関数** [Hash Function] を選択することに重点が置かれます。

## Coffee ☕ Break

## ハッシュテーブルとハッシュ関数（2）

そして、ハッシュ関数によって得られた整数をハッシュ値[Hash]と呼びれます。ハッシュ値はデータの「指紋」のようなもの：異なるデータは（大きな確率で）異なるハッシュ値を持ちます。

さらに、ハッシュ関数にはもう一つ重要な特徴がある：ハッシュ値から元のデータを推測することができない、という不可逆性です。

## Coffee ☕ Break

## ハッシュテーブルとハッシュ関数（3）

この特徴から、ハッシュ関数は情報セキュリティの分野で非常に有用です。例えば、ウェブサイト利用者のパスワードが直接サーバに保存されている場合、ハッキングにより全ての利用者のパスワードが流出する可能性があります。しかし、ユーザのパスワードのハッシュを保存しておけば、ユーザーがログインする際にパスワードが正しく入力されたかどうかを判断することができる同时、サーバのデータが漏洩しても、ハッカーは元のパスワードが復元できません。

また、ハッシュ関数は、データの暗号化、データの検証、仮想通貨などの分野でも利用されています。



# 目 次

- 1 基本概念
- 2 代表的なデータ構造
- 3 代表的なアルゴリズム
- 4 補足

# 一般的なアルゴリズム

- 実際の開発では、ほとんどのアルゴリズムが既に実装されました。既に実装されている一般的なアルゴリズムを使って慣れ、次に他の既存のアルゴリズムの実装に挑戦し、最後に独自のアルゴリズムを設計という順に追って勉強しましょう。
- ここでは、よくある問題とそれらを解決できるアルゴリズムについて説明します：
  - 配列探索問題：線形探索、二分探索。
  - ソート問題：バブルソート、挿入ソート、クイックソート、マージソート。
  - グラフの検索問題：深さ優先検索、幅優先検索。

# 探索問題

- **探索**[Search]問題とは、データ構造の中から、ある特徴を持ったデータやデータを探し出す問題です。
- また、データは探索された構造にあるかどうかの判断も可能です。

## Example ✓

1. 全生徒のデータから「山田花子」のデータを探します。
2. 20世紀の全ての年から最初の閏年を探します。

- まず、単純なデータ構造から考えてみましょう。例えば、配列にあるデータを探索する問題。

# 線形探索

- 最も簡単な方法は、配列全体を走査[Traverse]し、各要素を順番に取って、条件に一致するかどうかを判断することです。
- 考えましょう：このアルゴリズムの最良の場合とは？ 最悪の場合とは？ それぞれどのくらいの時間がかかるのでしょうか？
- 最良の場合：最初の要素は、望むデータです。時間計算量は  $O(1)$ 。
- 最悪の場合：最後の要素は望むデータか、望むデータが配列にないかのどちらです。時間計算量は  $O(n)$ 。

# 順序付き配列の探索

- 配列のデータが順番に並んでいる場合、この性質を利用して探索を高速化できますか？

## Example ✓

1. 全生徒を背の低い順に並べました。身長が1.7m以上の生徒を探します。
2. 電話帳に登録されているすべての電話番号を、小さいものから順に並べ替えました。`012-3456-7890`が電話帳に登録されているかどうかを確認します。

# 二分探索

- **二分探索**[Binary Search]は順序配列の高速探索アルゴリズムです。
- 便宜上、以下のデータは小さいものから順に並べていると仮定します。
- 考え方：配列の**真ん中**にあるデータと、探索したいデータを比較します。その二つが同じであれば、結果が見つかりました。探索したいデータの方が小さければ、配列の左半分を検索すればよくて、大きければ、配列の右半分を検索すればよいのです。左半分と右半分の検索では、再び二分探索を使用して検索時間を短縮できます。
- このように、無駄な探索時間が大幅に削減できます。時間計算量は  $O(\log n)$  に最適化されました。

# 二分探索の例

## Example ✓

整列された配列 {1, 3, 6, 8, 15, 18, 20}。

6 が配列にあるかどうかを判定。

19 が配列にあるかどうかを判定。

Try   
BinarySearch.java



*Question and answer*

# 問題

- **整列**[Sort] 問題とは、リスト内のデータを一定の順序で並べ替える問題です。
- 数字の場合、単純に小さいものから大きいもの、大きいものから小さいものへと並べることができます。ただし、比較基準を決めておけば、他のデータ型も並べることができます。

## Example ✓

全生徒を背の低い順に整列。

全ての企業を平均給与の高いほうから整列。

全てのユーザー名を辞書順[Lexicographic Order]で整列。

- ここは、便宜上、数字の昇順配列のみを考慮します。

# バブルソート

- **バブルソート**[Bubble Sort]は、最も簡単な整列アルゴリズムの一つです。
- 考え方：配列が整列されたら、隣り合う 2 つの数値は、必ず左のほうが右より小さくなるはずです。ですから、ひたすら隣り合う数字のペアを見て、左が右より大きければそれらを入れ替えます。この操作を、配列が整列されるまで繰り返します。
- アルゴリズムの動画はここでご覧いただけます：  
 <https://visualgo.net/en/sorting>。

# 挿入ソート

- **挿入ソート** [Insertion Sort] も、簡単なソートアルゴリズムの一つです。
- 考え方：ポーカーでカードを管理するのと似ています。まず、全てのカードを右手に適当に並べます。右手から無作為にカード一枚取りだして、左手にそれを置きます。そして、右手から一枚ずつカードを取って、左手の正しい位置にカードを入れて、常に左手のカードを順番に並べていきます。これを、全てのカードは順番になるまで繰り返します。
- アルゴリズムの動画を見ましょう。

# クイックソート

- クイックソート[Quicksort]は、実装が複雑ですが、効率のいい整列アルゴリズムです。
- クイックソートの考え方自体は簡単です：
  1. 適当に数  $k$  を取り、全ての数を、「 $k$  より大きい数」と「 $k$  より小さい数」に分けます。
  2.  $k$  より小さい数を  $k$  の左側に置き、それらをクイックソートで並べ替えます。
  3.  $k$  より大きい数を  $k$  の右側に置き、それらをクイックソートで並べ替えます。
- 気づきましたか？ これは、前に (◀ § 1.3.3) 紹介した再帰の考え方です。再帰の終了条件は何でしょうか？
- アルゴリズムの動画を見ましょう。

# マージソート

- マージソート [Merge Sort] も、複雑な整列アルゴリズムの一つです。
- 考え方もまた再帰を利用：
  1. 配列を真ん中から 2 つに分けます。
  2. 左半分をマージソートで整列します。
  3. 右半分をマージソートで整列します。
  4. この 2 つの整列された配列を、1 つの整列された配列に結合（マージ [Merge]）します。
- マージソートが効率高いのは、ステップ 4 のマージ操作の時間計算量は  $O(n)$  だけからです。
- アルゴリズムの動画を見ましょう。

# 整列アルゴリズムの計算量の比較

- 各整列アルゴリズムの時間計算量は何でしょうか。
- これらのアルゴリズムの時間および空間計算量はこの通り：

アルゴリズム	最悪時間計算量	平均時間計算量	空間計算量
バブルソート	$O(n^2)$	$O(n^2)$	$O(1)$
挿入ソート	$O(n^2)$	$O(n^2)$	$O(1)$
クイックソート	$O(n^2)$	$O(n \log n)$	$O(\log n)$
マージソート	$O(n \log n)$	$O(n \log n)$	$O(n)$

- 実際、データの比較に基づいた整列アルゴリズムでは、 $O(n \log n)$  より早いもの存在しません。

# 整列アルゴリズムの使用

- 複雑すぎて覚えられないとお思いですか？ 実際には、効率的なクイックソートとマージソートのアルゴリズムだけを使用すればよいので、心配は要りません。
- そして、それらを実装する必要もありません： 現代のプログラミング言語では、基本的に**標準的な整列メソッド**が提供されているので、**それを直接利用**すればよいです（これについては次章で説明します）。
- 例えば、Java のリストでは標準な整列アルゴリズムが提供されて、マージソートの変種で実現されました。



*Question and answer*

# グラフの探索問題

- 偶に、複雑なデータ構造の中にあるデータを探索する必要があります。例えば、**グラフ**です。

## Example ✓

地下鉄で東京駅から渋谷駅を探します。

ユーザーのパソコンからインターネット上にあるサーバのパソコンを検索します。

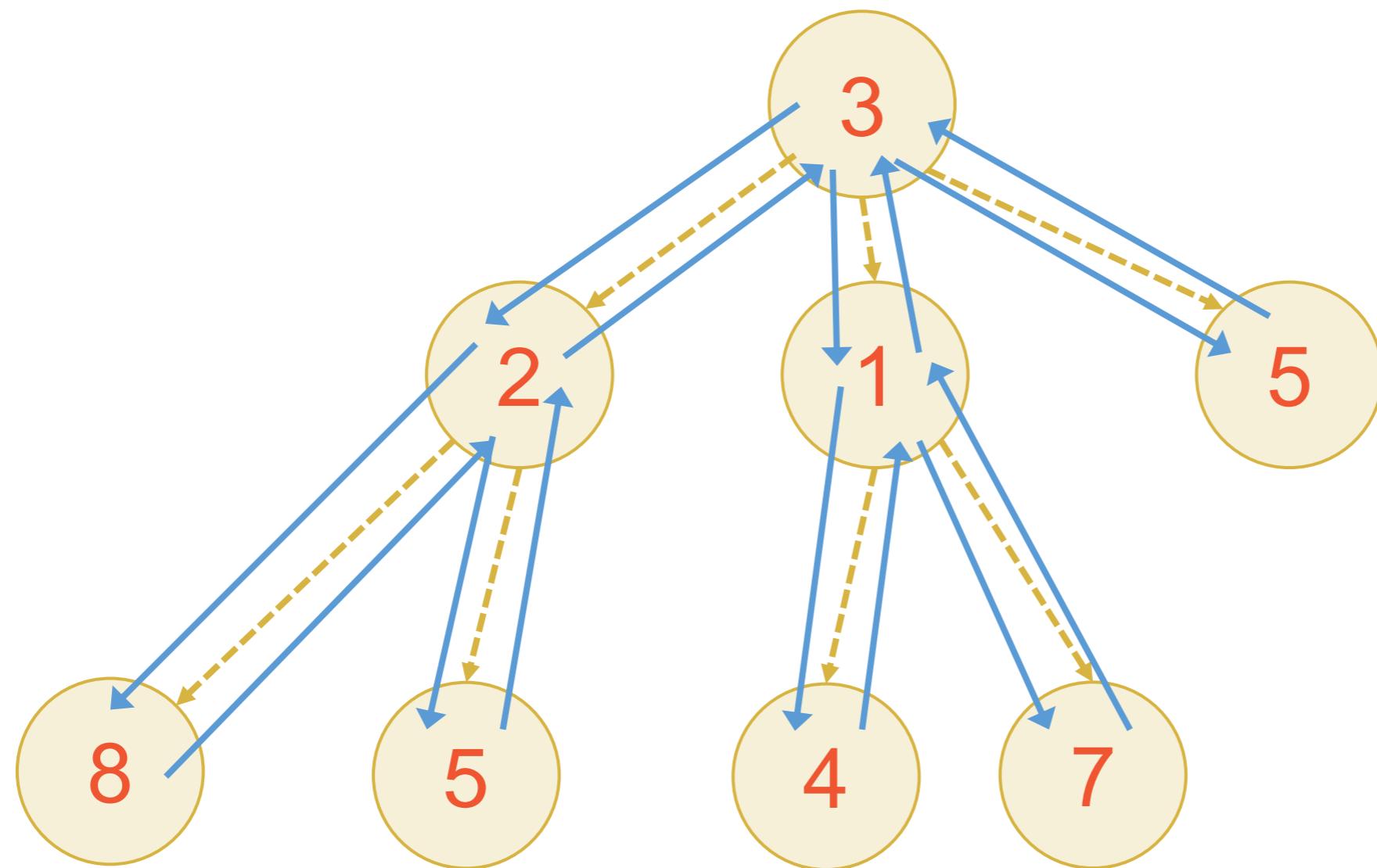
ルートディレクトリから、指定されたファイル名のファイルを見つけます。

# グラフの探索アルゴリズム

- 最も基本的で一般的に使用されているグラフの探索アルゴリズムは 2 つあります:
  - 深さ優先探索[Depth First Search, DFS]。
  - 幅優先探索[Breadth First Search, BFS]。
- どちらの探索方法も、グラフ内のすべてのノードを重複や省略なしに探索しようとします。
- ここで、簡単なグラフである木を例として、それらのアルゴリズムがどのように実装されるか見ていきましょう。より複雑なグラフでも考え方は基本同じです。

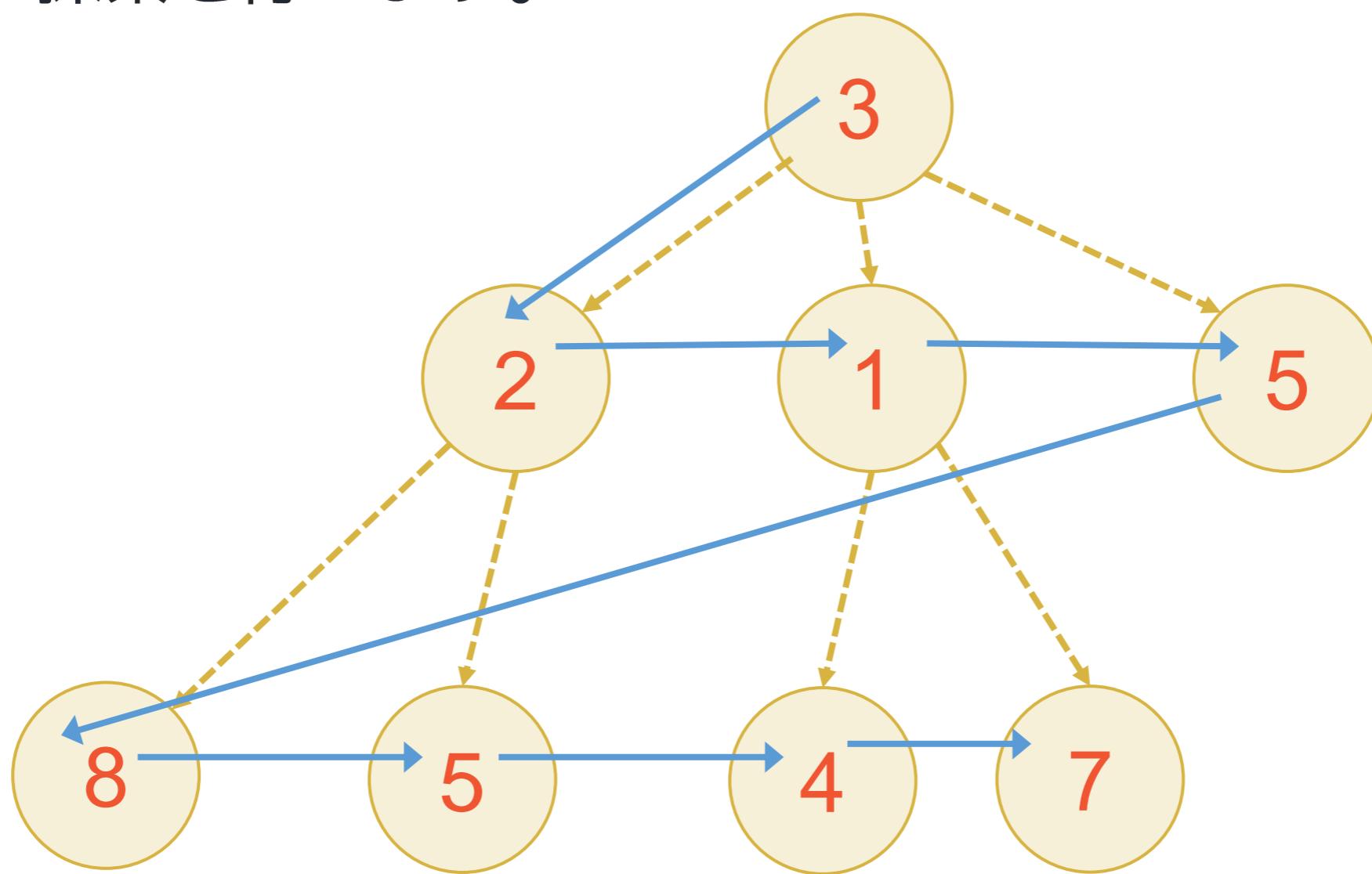
# 深さ優先探索

- 深さ優先探索は、ルートから始まり、できるだけ子を探索し、全ての子を探索し終えたら親ノードに戻ります：



# 幅優先探索

- 幅優先探索 モルートから始まり、その子をできるだけ探索してから、子の子をできるだけ探索し、子の子の子を探索……の順に探索を行います。





*Question and answer*



# 目 次

- 1 基本概念
- 2 代表的なデータ構造
- 3 代表的なアルゴリズム
- 4 補足

# その他のデータ構造とアルゴリズム

- この章では、より高度的に実用的データ構造とアルゴリズムを紹介します。
- 今は、これらのデータ構造とアルゴリズムが何の機能を持つかの印象を持つだけでよく、詳細な実装方法と計算量などは、実際に使いたいときに調べましょう。

# 二分探索木

- **二分木** [Binary Tree] は、特殊な木構造で、各ノードは2つ以上の子を持ちません。
- **二分探索木** [Binary Search Tree, BST] は、二分探索の考え方に基づく特殊な二分木です。データを順番に保存することができ、データの検索、データの削除、新しいデータの挿入の平均時間計算量は  $O(\log n)$  です。
- **平衡二分探索木** [Balanced BST] は二分探索木の最適化です。通常の BST では、最悪なケースの計算量が  $O(n)$  に対し、平衡化された二分木は最悪の場合も  $O(\log n)$  を保証します。AVL 木、赤黒木、スプレー木など様々な実装方法があります。
- 二分探索木は、主に頻繁に変更する必要がある順序付けられたデータを格納するために使用されます。

# ヒープと優先度付きキュー

- ヒープ [Heap] は特別な木です：各ノードは、そのどの子ノードよりも大きい（最大ヒープ）・小さい（最小ヒープ）。一般に使用されるヒープは二分ヒープです。
- 優先度付きキュー [Priority Queue] は抽象データ型で、通常はヒープで実装されます。優先度付きキューに格納されたデータは順番に保存され、以下の操作ができます：
  - データを挿入。ヒープで実装した場合の計算量は  $O(\log n)$ 。
  - 最大データを取得（そして削除）。ヒープで計算量は  $O(\log n)$ 。
  - 最大データを取得（削除せず）。ヒープで計算量は  $O(1)$ 。
- 優先度付きキューは、優先順位によって使用する必要があるデータを格納するために使います。

# 特別な集合

- **ビット集合** [Bitset] は、特別な集合で、整数しか格納できませんが、使われるメモリの容量を大幅に節約できます。
- **素集合** [Disjoint Set] は、特殊な集合で、共通のない複数の集合を扱います。集合の基本操作に加え、主に以下の集合間操作が可能:
  - ある要素が複数の集合のどれに属するかを調べます。
  - 二つの集合を一つにまとめます。
- 素集合の実装にはいくつか種類がありますが、最適化されたものは上記の操作の時間計算量が全て  $O(1)$  くらいになります。
- 素集合は、主にいくつかのグラフのアルゴリズムを補助するために使用されます。

# 他の整列アルゴリズム

- 先ほど説明した整列アルゴリズム以外にも、様々なアルゴリズムがあります:
  - **選択ソート** [Selection Sort]。もう一つの単純なソートアルゴリズム。時間計算量は  $O(n^2)$ 。
  - **ヒープソート** [Heap Sort]。ヒープに基づく整列アルゴリズムです。時間計算量は  $O(n \log n)$ ，マージやクイックソートの代わりに使われることもあります。
  - **シェルソート** [Shell Sort]、**カウントソート** [Counting Sort]、**バケットソート** [Bucket Sort]など。これらのアルゴリズムは、データの比較に基づいていないために、時間計算量は  $O(n)$  あります。ただし、限定的な状況のみ使えます。例えば、データがある範囲内の正の整数のみである場合。

# グラフのアルゴリズム

- **ダイクストラ法**[Dijkstra's Algorithm]: あるノードから他の全てのノードまでの最短距離を計算する。
- **ベルマン - フォード法**[Bellman-Ford Algorithm]: ノードからの最短距離を計算。ただし、エッジの値が負数でも可能。
- **ワーシャル - フロイド法**[Floyd-Warshall Algorithm]: 全ノード間の最短距離を計算。
- **A\* アルゴリズム**: 2つのノード間の最短なルートを探索でき、道案内やナビゲーションに広く使用されている。
- **フォード - ファルカーソン法**[Ford-Fulkerson Algorithm]: ある場所から別の場所への最大トラフィック量を計算。

# 文字列のアルゴリズム

- **KMP アルゴリズム** [Knuth-Morris-Pratt Algorithm]: 長い文字列の中の特定な単語の位置を求める場合で広く使用されます。
- **ボイヤー - ムーア法** [Boyer-Moore Algorithm]: 特定な単語の出現位置を求めるのにも使われるが、従来のアルゴリズムより効率的です。このアルゴリズムは、検索エンジンで広く使用されます。
- **トライ木** [Trie] は、多数の文字列の集合（辞書）を保存するためのデータ構造で、入力補完やスペルチェックなどで広く使用されます。

# アルゴリズム設計パターン

- これまでの紹介は、既に存在するアルゴリズムについてのものでしたが、自分たちでアルゴリズムを設計するとしたらどうでしょうか。ここでは、より一般的なアルゴリズム設計のアイデアを紹介します。自分で勉強したい方は、まずここから始めてみましょう：
  - しらみつぶし [Brute-force]
  - 貪欲法 [Greedy]
  - 分割統治法 [Divide-and-conquer]
  - バックトラック法 [Backtracking]
  - 動的計画法 [Dynamic Programming, DP]

# さらなる勉強

- 様々なデータ構造やアルゴリズムに慣れ、それらを実装し、さらには設計する能力を向上させるには、実際にプログラミングして、問題を解決するほどいい方法はありません。
- 実際に解くべき問題がない場合は、LeetCodeなどのアルゴリズム問題練習サイトを通じてオンラインで練習するのもいいのでしょうか：  
 <https://leetcode.com/>



*Question and answer*

# まとめ

## Sum Up

1. データ構造とアルゴリズムの基本概念:
  - ① データ構造の選択プロセス。
  - ② データ構造とアルゴリズムの評価基準。
2. 基本的なデータ構造:
  - ① リスト、集合、辞書などの主要な抽象構造を覚えましょう。
3. 基本的なアルゴリズム：探索、ソート問題:
  - ① 二分探索の考え方。
4. さらなる勉強方法：実践。

**THANK YOU!**