教養としての
アルゴリズムとデータ構造
「木構造」

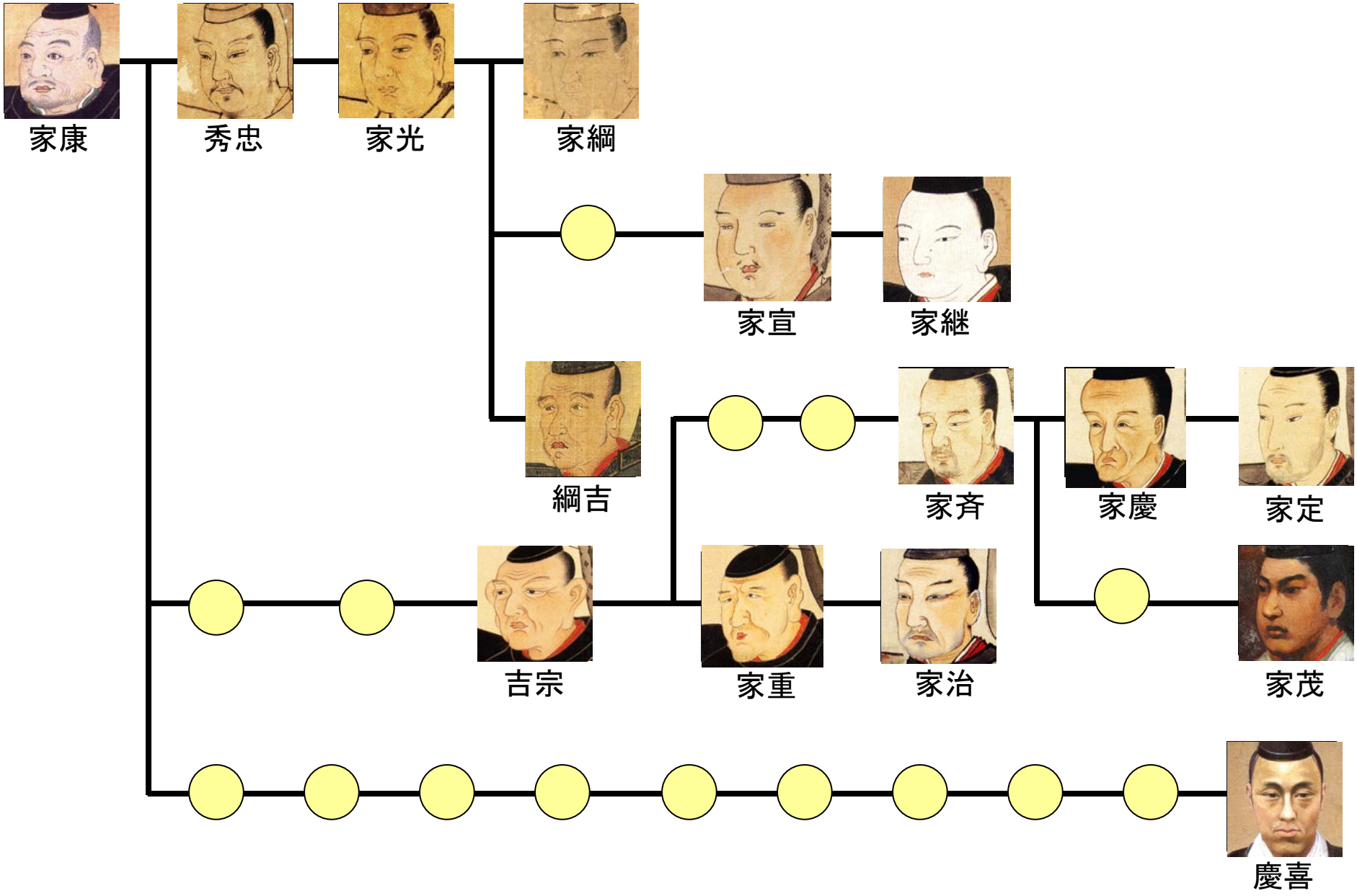
小林浩二
kojikoba@mi.u-tokyo.ac.jp

木構造とは

- 木構造とは、樹形図状にデータを保持するデータ構造のこと
  - 樹形図状のデータの例:
    - 家系図
    - 組織図
    - ファイル・フォルダの階層構造
    - 言語学の構文解析

    など
  - 木構造は単に「木」とも呼ばれます

家系図



組織図

- 研究科教授会
- 教育会議 — 専攻教育会議
- 学術運営・教育推進委員会 — 技術委員会 / 教務委員会 / 学生支援室
- 諮問会

技術部

- 共通系
- 機器分析・実習系
- 生命科学系

理学系研究科

- コンプライアンス室
- 理学系研究科長
  - 企画室
    - 情報システムチーム
    - 広報室
    - 男女共同参画室
    - キャリア支援室
    - 国際化推進室
    - キャンパス計画室
    - 研究支援総括室
    - 評価委員会
  - 環境安全管理室
- 副研究科長
- 研究科長補佐

専攻

- 物理学専攻
- 天文学専攻
- 地球惑星科学専攻
- 化学専攻
- 生物科学専攻

フォトンサイエンス・リーディング大学院

附属施設

- 植物園 — 日光分園
- 臨海実験所
- スペクトル化学研究センター
- 地殻化学実験施設
- 天文学教育研究センター — 木曽観測所 / アタカマ観測所
- 原子核科学研究センター
- ビッグバン宇宙国際研究センター
- 超高速強光子場科学研究センター
- 遺伝子実験施設
- フォトンサイエンス研究機構
- 生物普遍性研究機構
- 宇宙惑星科学機構
- 知の物理学研究センター

事務部

- 総務課
- 学務課 — 専攻チーム / 施設チーム
- 経理課

各種委員会等

国際交流委員会、広報委員会、男女共同参画委員会、情報倫理審査会、ネットワーク委員会、キャンパス計画委員会、建物委員会、防災委員会　ほか

理学部

- 学部教授会
- 理学部長

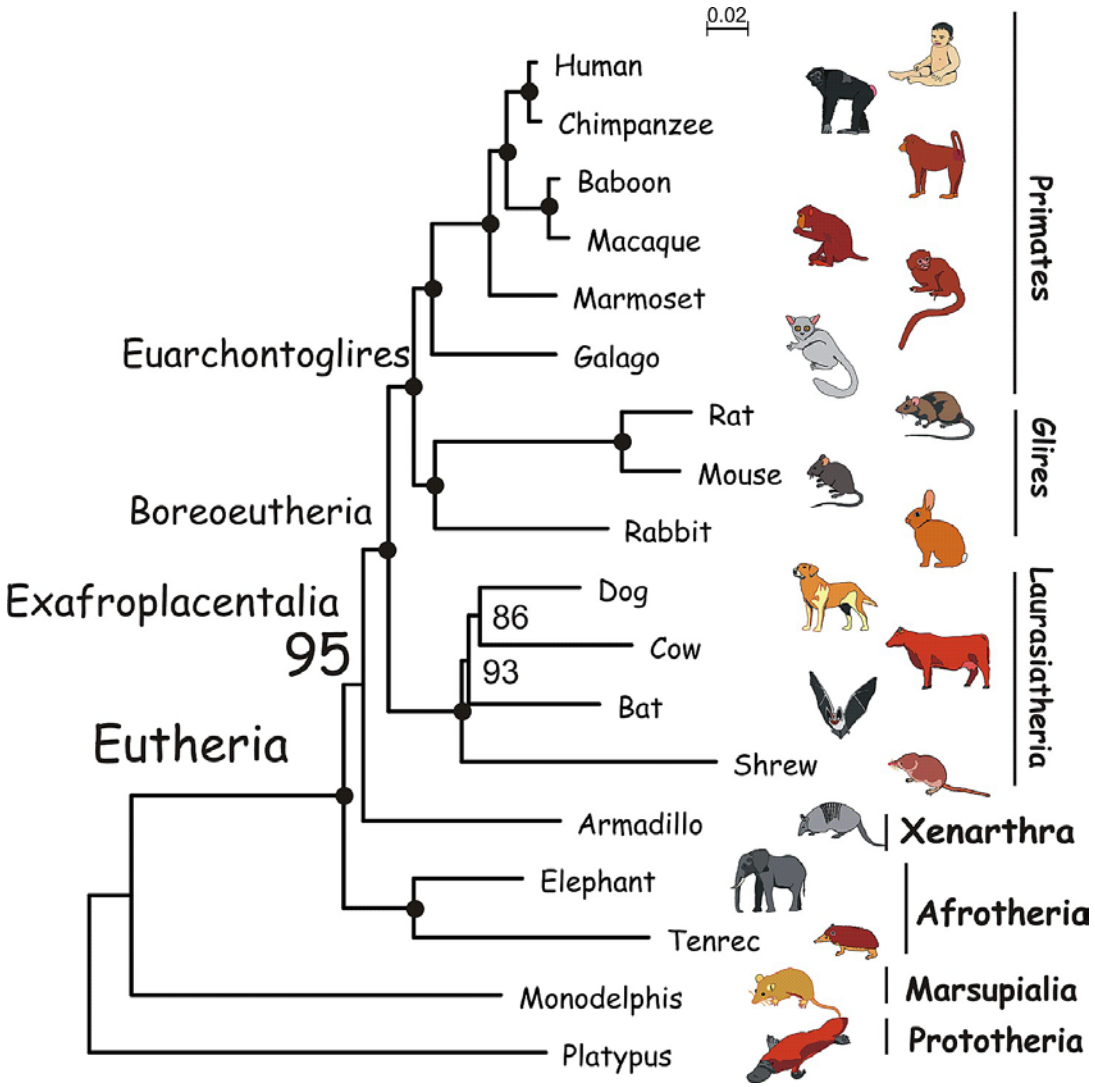
学科

- 数学科
- 情報科学科
- 物理学科
- 天文学科
- 地球惑星物理学科
- 地球惑星環境学科
- 化学科
- 生物化学科
- 生物学科
- 生物情報科学科

グローバル基礎科学教育プログラム
（グローバルサイエンスコース）

系統樹

- 系統樹：生物のどの種が近縁であるかを表す図



https://plos.figshare.com/articles/figure/_ML_Phylogenetic_Tree_of_Mammals_/624272/1

言語学における構文木

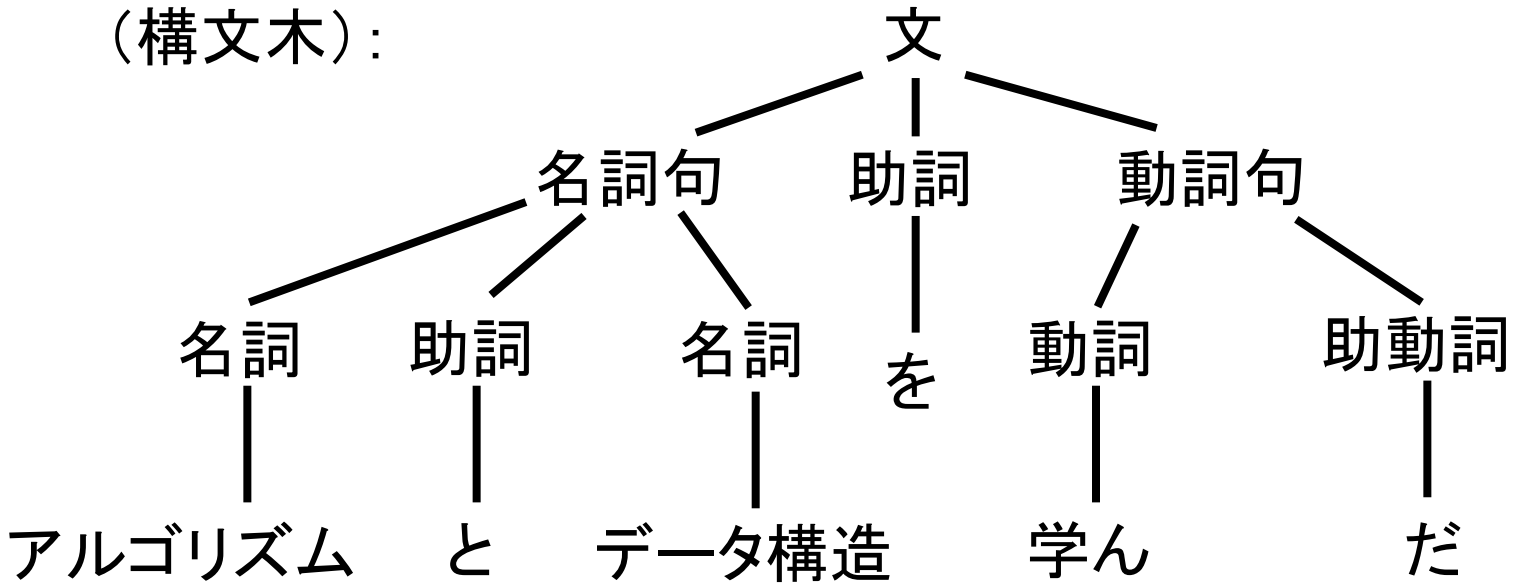
（元の文）: アルゴリズムとデータ構造を学んだ

形態素解析

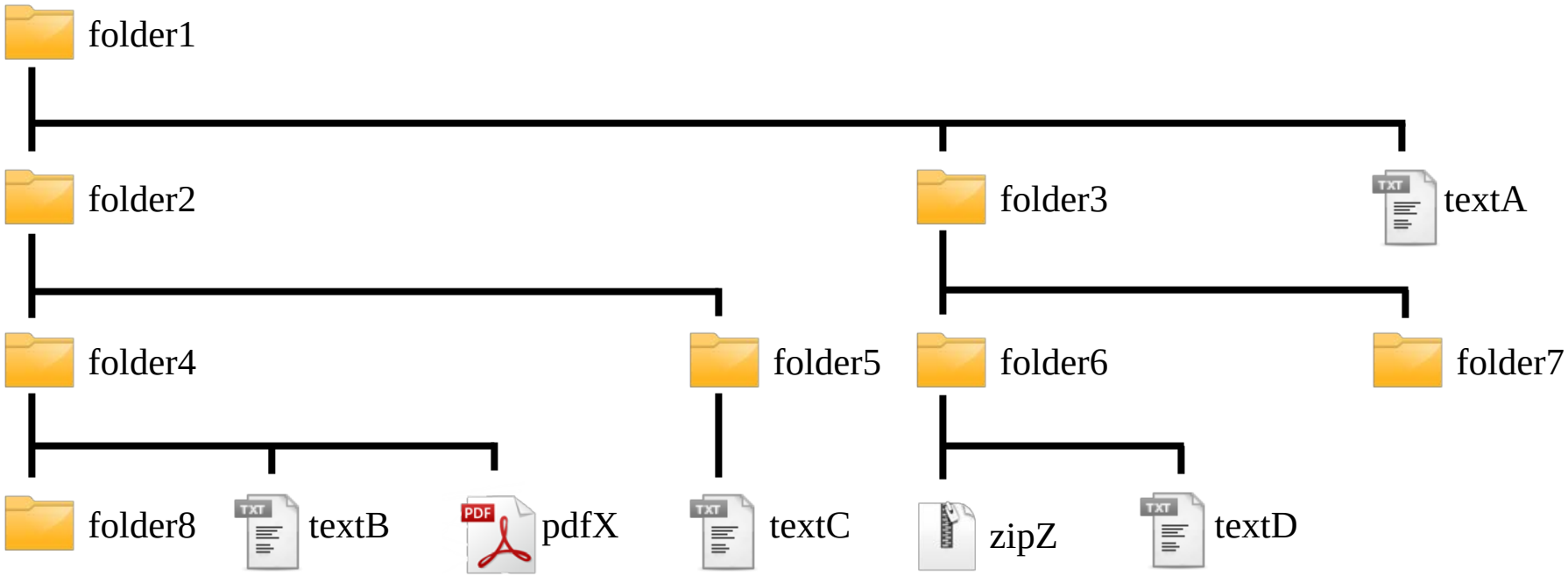
（形態素列）: アルゴリズム/と/データ構造/を/学ん/だ

構文解析

（構文木）:



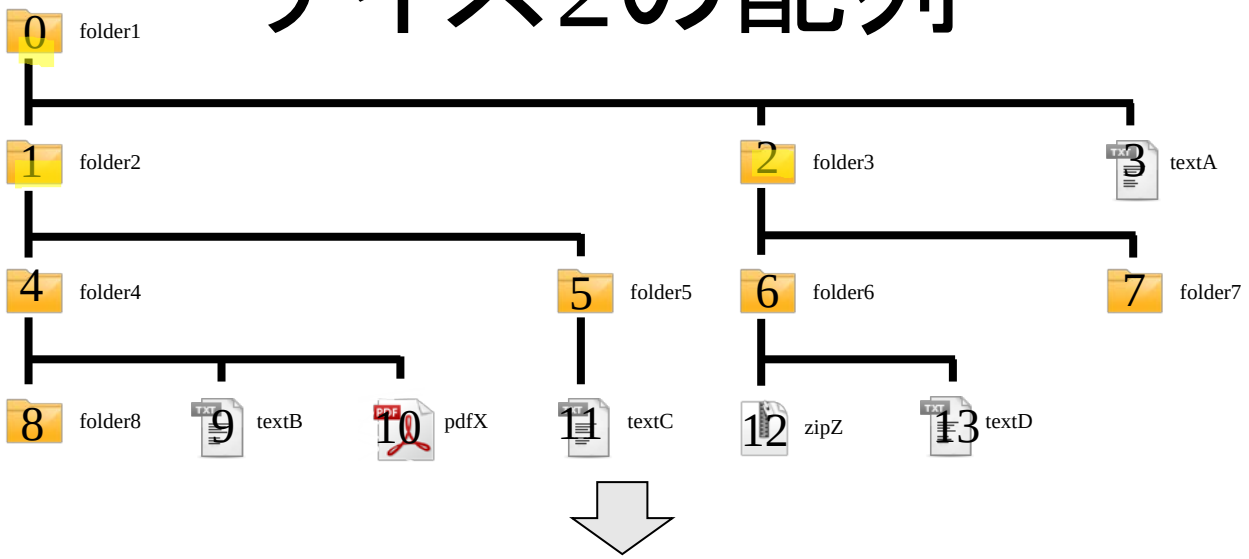
ファイルとフォルダの階層構造



木構造の表現方法

- 最も単純な方法:
  - 大きさ2の配列の集まりで表す
    - 非常に利便性が低い（計算量的によろしくない）

サイズ2の配列



- 例えば、Pythonでは「folder6 (6)の下にtextD (13)がある」場合、(6, 13) というタプル（もしくはリスト）で表す

```
[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6),
 (2, 7), (4, 8), (4, 9), (4, 10), (5, 11), (6, 12), (6, 13)]
```

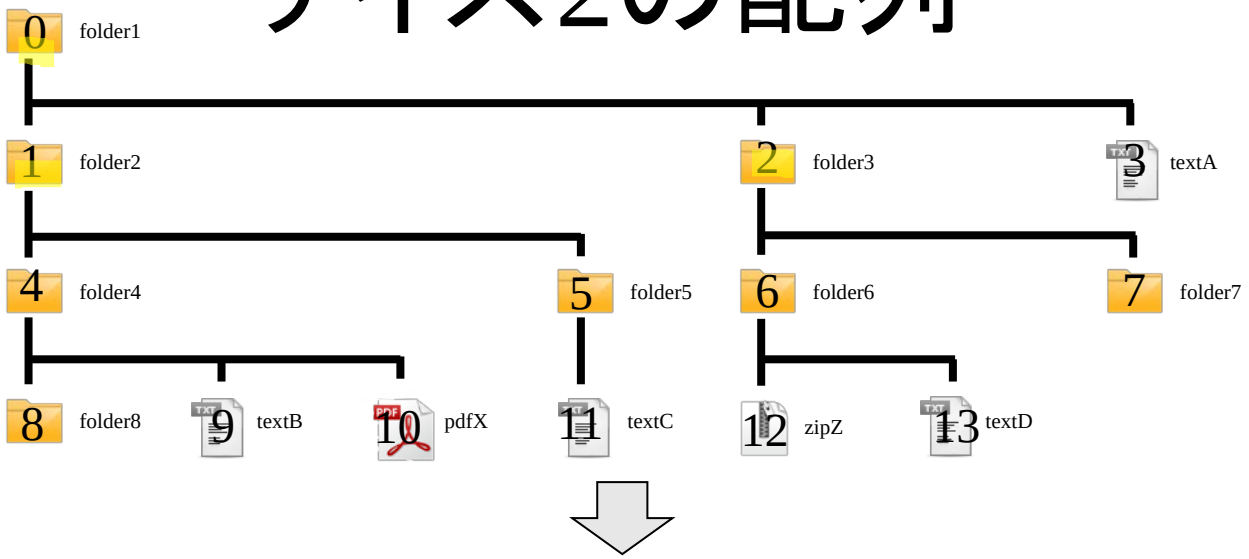
- 「folder6 (6)直下のファイル」を調べたい場合、リスト内を探索して(6, *)という値を探す
  - ファイル・フォルダの数がm個だと、線形探索で計算量O(m)
  - ソートO(m log m)→二分探索 O(log m+k) (ただし、kは(6, *)という値の数)
  - ハッシュでもO(m)
    - (6,*)となる値を(6,0),(6,1),…と順に探す

```
{(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6),
 (2, 7), (4, 8), (4, 9), (4, 10), (5, 11), (6, 12), (6, 13)}
```

用語解説

- 計算量（時間計算量）:
  - 計算機（コンピュータ）で何らかの処理（つまり、計算）を行うのに費やす操作の数
  - 演算（加減乗除など）や変数の使用、値の比較など単純な処理は1回の操作で行えると考える
    - Pythonのメソッド・関数などでは1回の操作で行っている様に見えてそうではないものが多数存在する
  - 計算量が少ないほど（計算）時間がかからないということ

サイズ2の配列



- 例えば、Pythonでは「folder6 (6)の下にtextD (13)がある」場合、(6, 13) というタプル（もしくはリスト）で表す

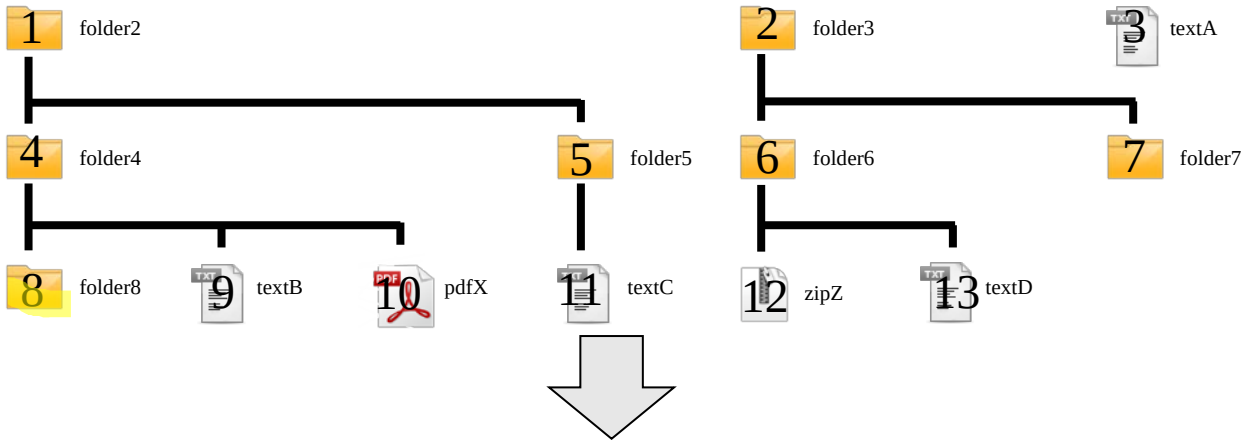
```
[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6),
 (2, 7), (4, 8), (4, 9), (4, 10), (5, 11), (6, 12), (6, 13)]
```

- 「folder6 (6)直下のファイル」を調べたい場合、リスト内を探索して(6, *)という値を探す
  - ファイル・フォルダの数がm個だと、線形探索で計算量O(m)
  - ソートO(m log m)→二分探索 O(log m+k) (ただし、kは(6, *)という値の数)
  - ハッシュでもO(m)
    - (6,*)となる値を(6,0),(6,1),…と順に探す

```
{(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6),
 (2, 7), (4, 8), (4, 9), (4, 10), (5, 11), (6, 12), (6, 13)}
```

サイブつの配列

```
[(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 6),
 (2, 7), (4, 8), (4, 9), (4, 10), (5, 11), (6, 12), (6, 13)]
```

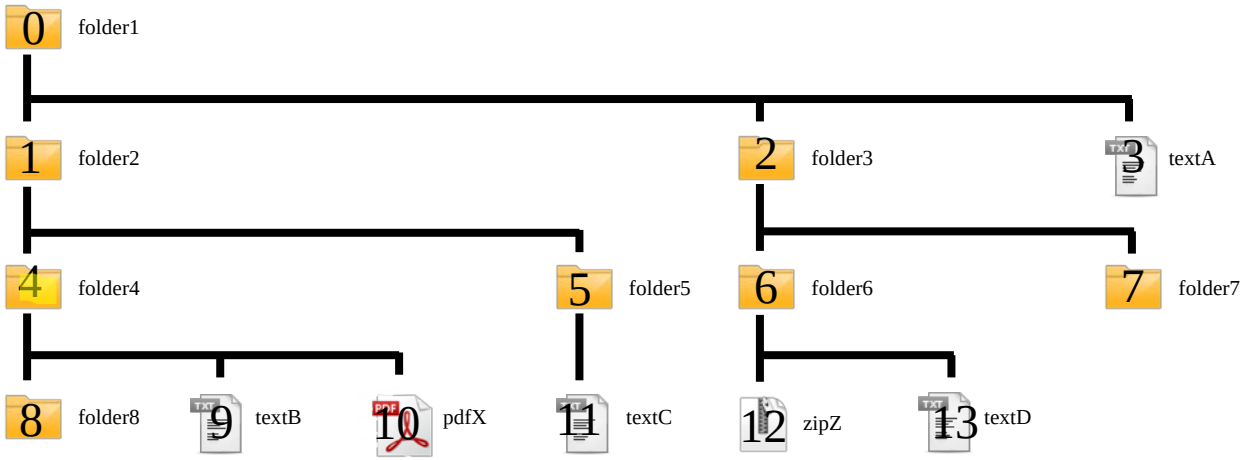


- 例えば、Pythonでは「folder6 (6)の下にtextD (13)がある」場合、(6, 13) というタプル（もしくはリスト）で表す
- 「folder1 (1)の下のファイル（直下とは限らない）」を調べたい場合は？
  - まずリストを探索して(1, *) という値を探す→(1,4)と(1,5)を発見
  - 次に(4,*)と(5,*)を探索…（以下繰返し）
  - どの手法でもO(m log m)やO(m^2)の計算量になってしまう

木構造の表現方法

- 最も単純な方法:
  - 大きさ2の配列の集まりで表す
    - 非常に利便性が低い(計算量的によろしくない)
- 木構造を活かした方法:
  - 隣接リスト
    - 多次元配列で実現
      - Pythonでは多重リスト(2次元)
    - Pythonの「リスト」とは(直接は)無関係
    - それ以外の方法:
      - 1次元の配列
        - » Pythonではリスト(1次元)→別の回にやります
      - 再帰構造

隣接リスト



外側の配列

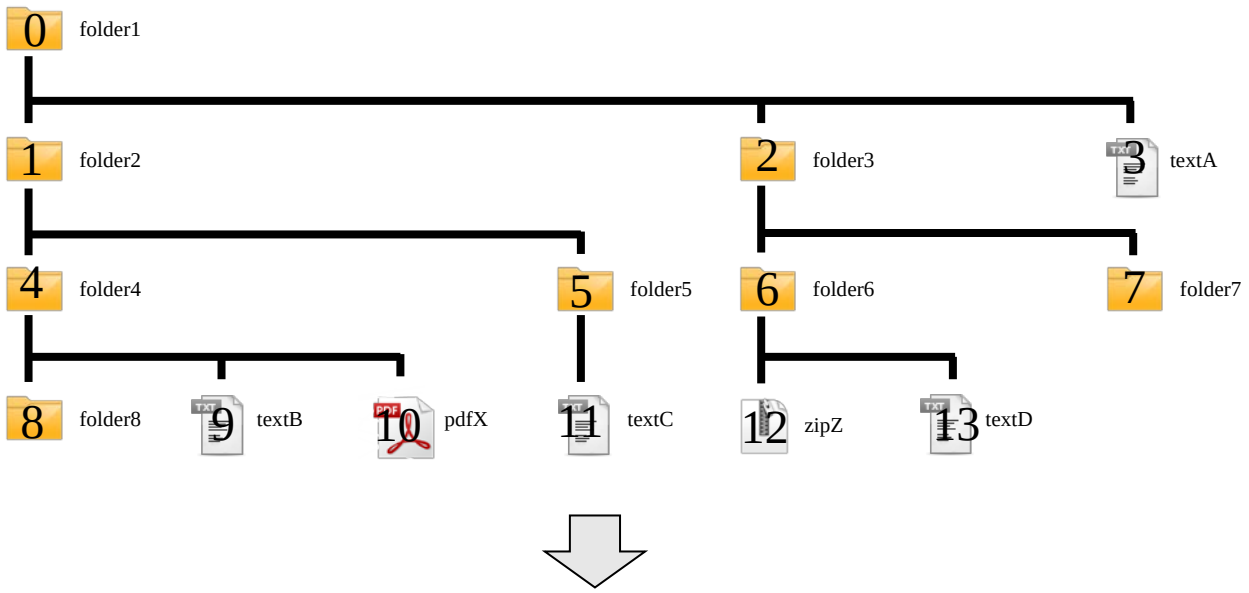
0	folder1 folder2 folder3 textA
1	folder2 folder4 folder5
2	folder3 folder6 folder7
3	textA
4	folder4 folder8 textB pdfX
5	folder5 textC
6	folder6

外側の配列

7	folder7
8	folder8
9	textB
10	pdfX
11	textC
12	zipZ
13	textD

```
In [1]: [[1,2,3], [4, 5], [6, 7], [], [8, 9, 10], [11], [12, 13], [], [], [], [], [], [], []]
```

隣接リスト

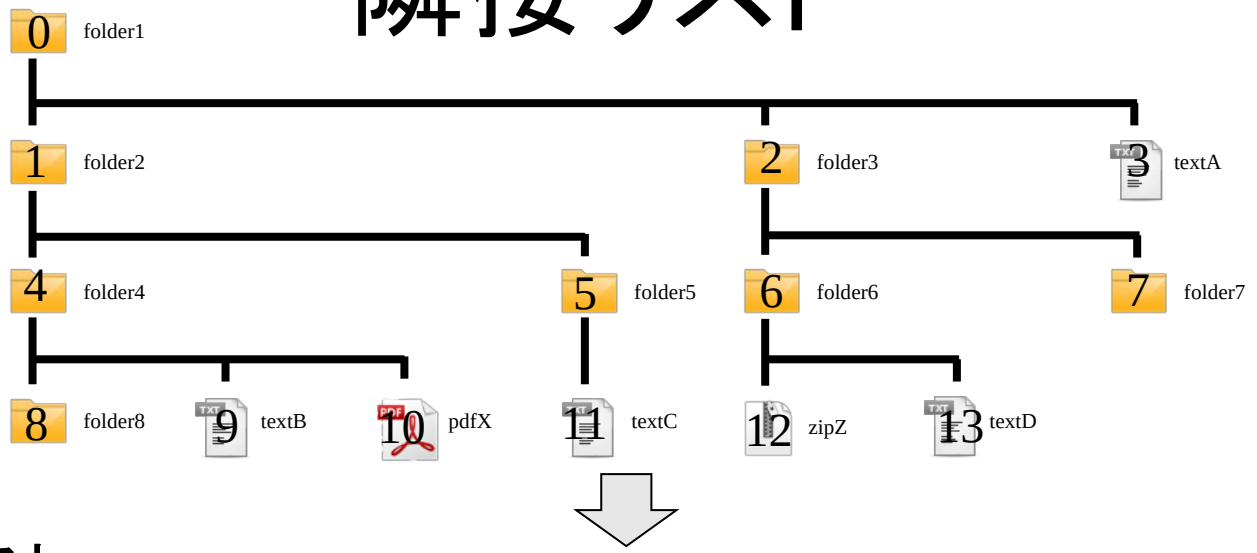



```
[1]: [[1,2,3], [4, 5], [6, 7], [], [8, 9, 10], [11], [12, 13], [], [], [], [], [], [], []]
```

- 以降は、簡単のため、各フォルダ/ファイル名を 0 から13の各整数で表します。

```
{"folder1":0, "folder2":1, "folder3":2, "textA":3, "folder4":4, "folder5":5,
 "folder6":6, "folder7":7, "folder8":8, "textB":9, "pdfX":10, "textC":11, "zipZ":12, "textD":13}
```

隣接リスト



- 隣接リスト:

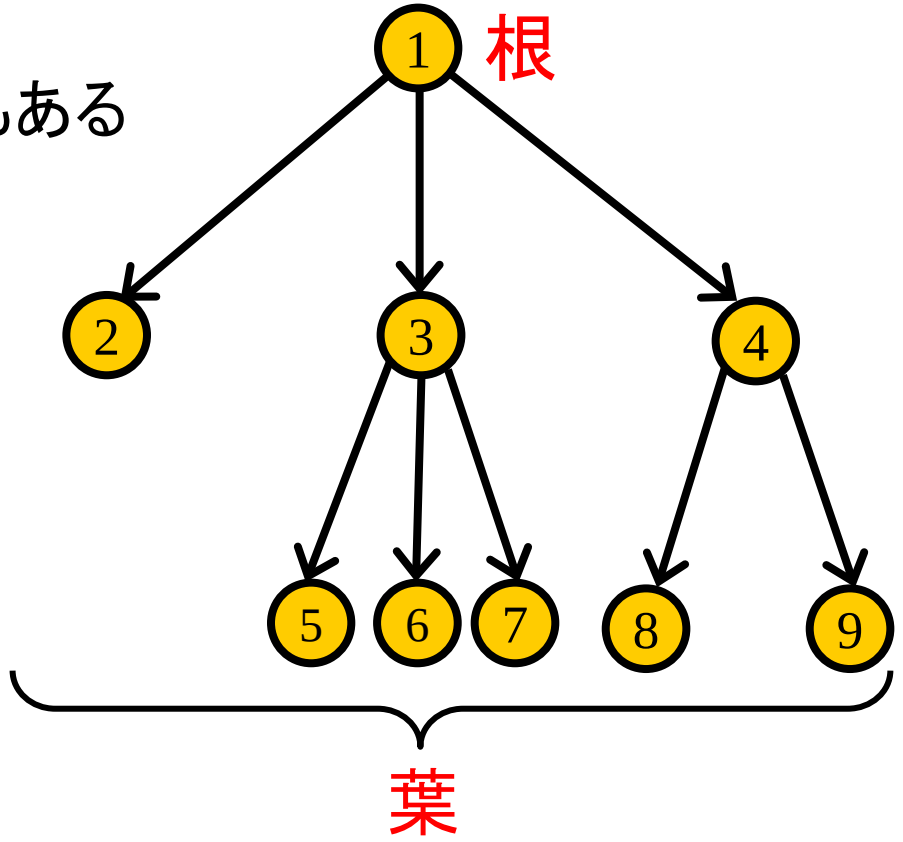
```
[1]: [[1,2,3], [4, 5], [6, 7], [], [8, 9, 10], [11], [12, 13], [], [], [], [], [], [], []]
```

- 「6の直下のファイル」を調べたい場合:
  - 隣接リストのインデックス6の値を調べれば良い:計算量O(1)
- 「1の下のファイル(直下とは限らない)」を調べたい場合:
  - 隣接リストのインデックス1の値を調べる→リスト[4,5]を発見
  - インデックス4と5の値を調べる→リスト[8,9,10]と[11]を発見(以下、繰返し)
  - ファイル・フォルダの数がmだと、最悪O(m)

木構造の用語1

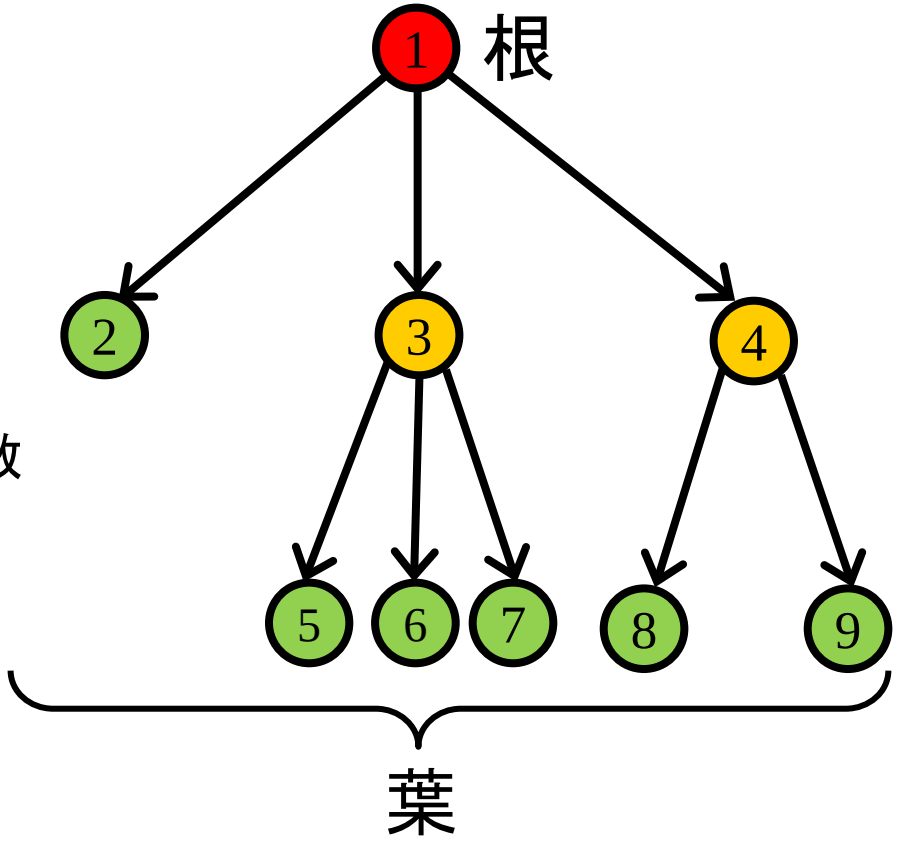
- 木構造は、点（もしくは頂点。右図では●）と枝（右図では⟶）で表す
- 点や枝には値・名称を書く場合もある

- 根：
  - 枝が1つも入って来ない点
- 葉：
  - 枝が1つも出ていない点



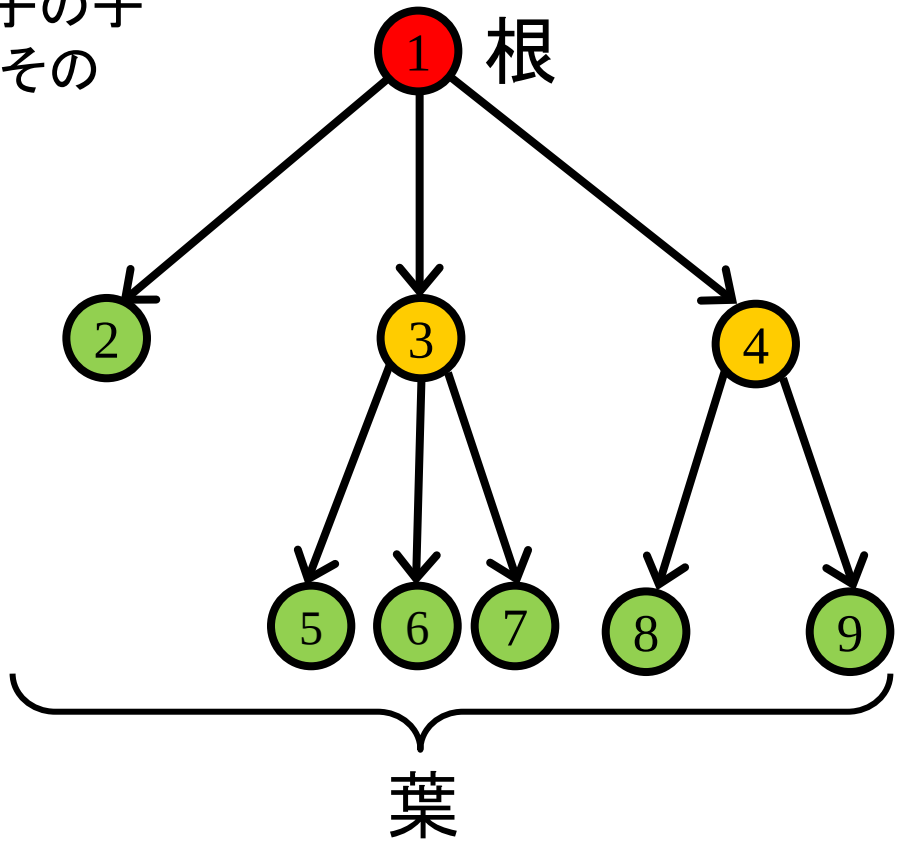
木構造の用語2

- 点 v の子（子供）:
  - 点 v からの枝が向かってくる点
  - 例：根（点1）の子＝点2, 3, 4
- 点 v の親:
  - 点 v が子である様な点
  - 例：点3の親＝根、点8の親＝点4
- 点 v の高さ:
  - 根からvまでの最短経路上の枝の数
    - 経路＝2点間をつなぐ点と枝の列
    - 距離＝経路上の枝数
  - 例：根=0, 点2 = 1, 点5 = 2
- 木の高さ:
  - 全ての点の高さの中で最大の高さ
  - 例の木の高さは2



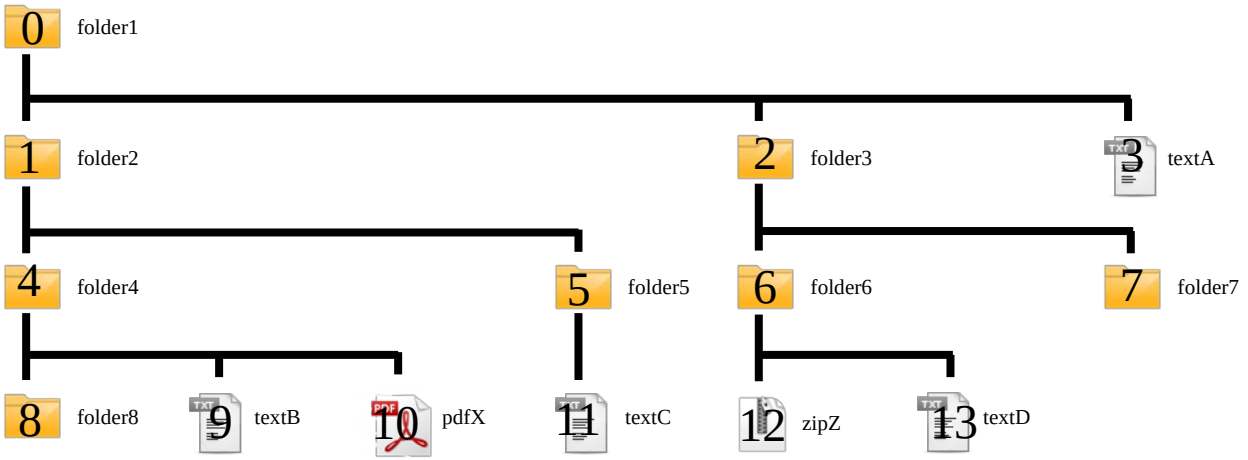
木構造の用語3

- 点 v の子孫:
  - 点 v の子、点 v の子の子、点 v の子の子の子、…を全て集めた点の集合（とその集合に属する点）
  - 例:
    - 根以外の全ての点は根の子孫
    - 点8,9は点4の子孫



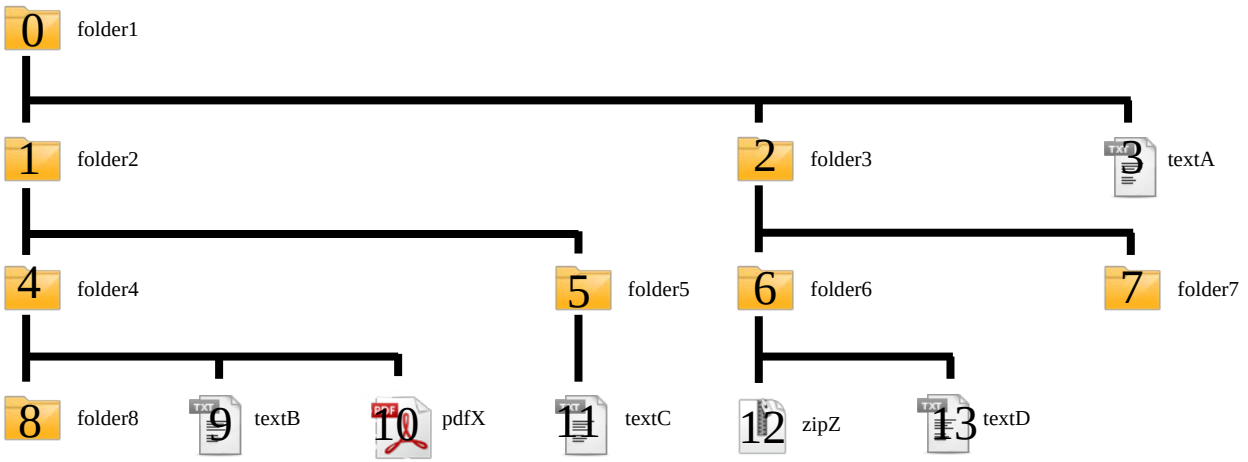
木構造に対する基本操作

- 木が指定した値（点）を含むか調べる
- 指定した2点の木構造上の関係を求める
  - 例：点Aは点Bの子孫かどうか
  - 例：2点間の木構造上の最短経路や最短距離
    - 経路＝2点間をつなぐ点と枝の列、距離＝経路上の枝数
- 条件を満たす点を全て求める
  - 例：点Aの全ての子孫



木構造上の探索

- 木構造に対する操作は、一般に木構造上の探索を利用して実現されます
  - 木構造が配列で実現されていれば、線形探索や二分探索が可能
  - 木構造の構成を活かした方が効率的(と思える)
- 木を探索する場合、どの様な順で調べる？
  - 例えば、0 の複数の子(1,2,3)とその子孫を調べる必要がある場合は？
  - 幅優先探索
  - 深さ優先探索

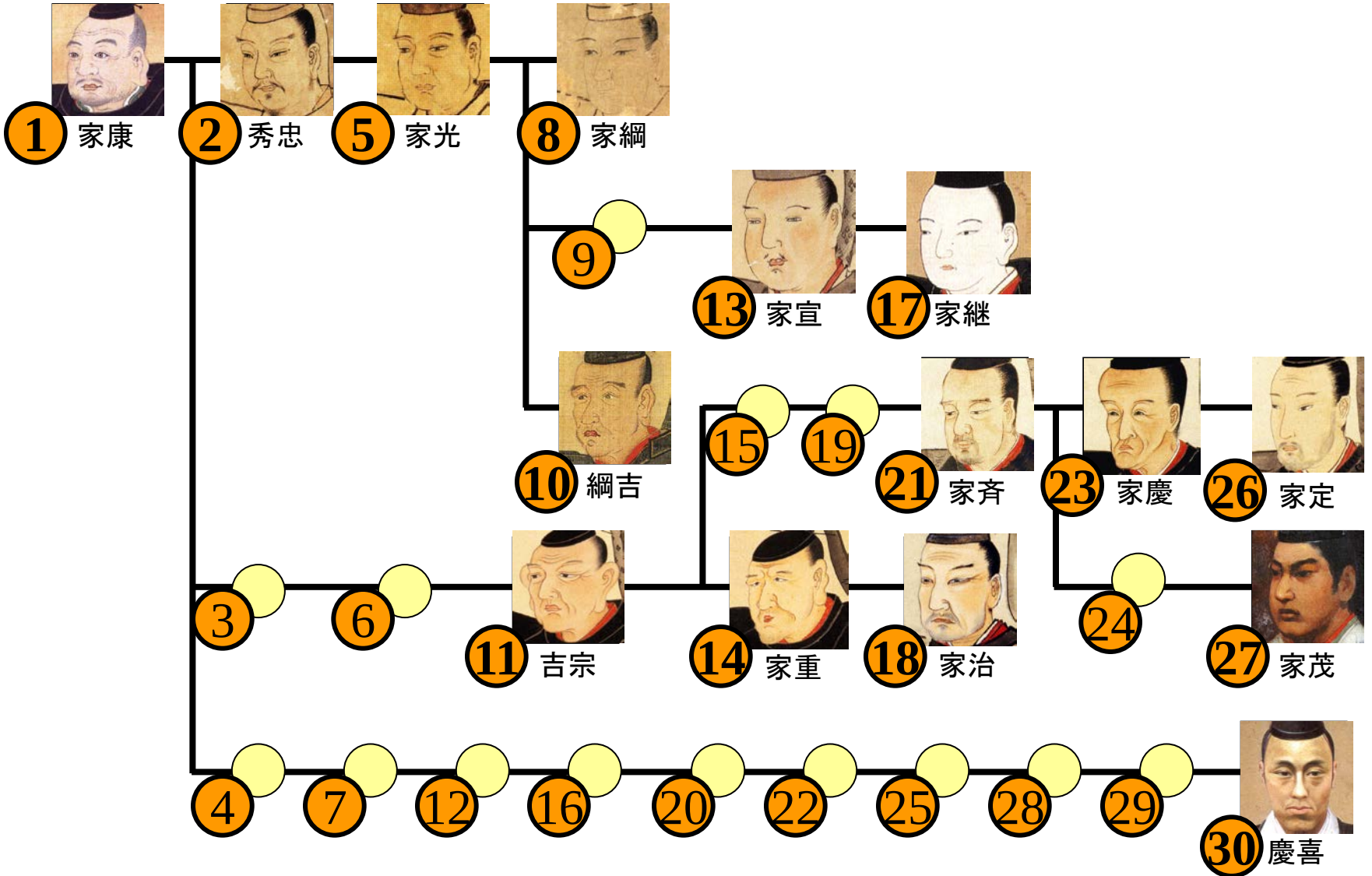


幅優先探索とは

- 幅優先探索 (Breadth first search)
  - 1つの点（値）vを起点とし、v, vの全ての子、vの全ての子の全ての子（いわゆる、vの全ての孫）、…という順序、すなわち、起点に近い順で v の全ての子孫の点を調べていく探索
    - 探索しながら、起点からの経路や距離を計算する手続きなども目的に合わせて行う

幅優先探索とは

- ある値（起点）から探索を開始し、起点に近い順に調べる



幅優先探索の定義

- 木構造Tを探索する
  - 1. Qを頂点（の名前）を格納するための集合とする
  - 2
  - 3. Qが空の場合、探索を終了
  - 4. Qから、Qの中で起点に最も近い頂点（の名前）vを取り出す
    - Qから頂点vを取り出すことを「vを訪問する」という
      - ここでvが探している点かどうか調べたり、起点からの距離を計算したりする
  - 5. Qにvの各子供を追加し、3 に戻る

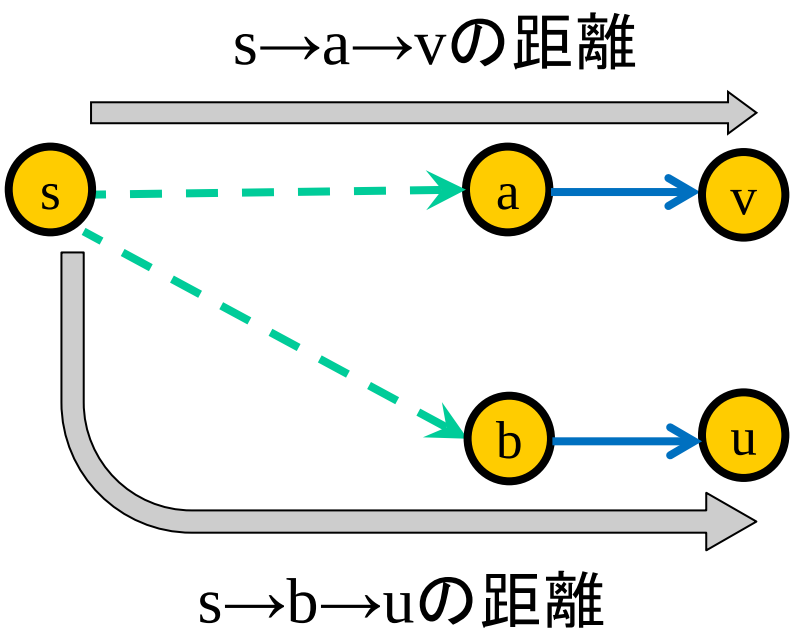
線形探索などでは探索中の点とそれ以外の点しかなかったが、
木構造の場合、探索中の点と後で探索する点とそれ以外が存在する
→「後で探索する点」をQに記録しておく

起点に最も近い点

- Qから、「Qの中で起点に最も近い頂点を取り出す」にはどうすれば良いか？
  - Qに頂点を入れるときに、起点からその頂点までの距離（枝の数）を記録する様にする
  - Qから点を取り出すとき、毎回Qの中の全ての頂点を調べて最小の距離を持つものを探す？
    - 木の頂点数がnの場合、取り出す計算量は最悪O(n)
  - 「Qの中で最も昔にQに加えた点＝Qの中で最も起点に近い頂点」が成立している
    - Qを配列にして、Qに加えた順に頂点を整列させておく
    - O(1)で頂点を追加・取り出しが実行可能

起点に最も近い点

- 「Qの中で最も昔にQに加えた点＝Qの中で最も起点に近い頂点」の証明（背理法）:
  - Qから頂点vを取り出すまでは、「Qの中で最も最初にQに加えた点＝Qの中で最も起点sに近い頂点」が成立していた
  - 更に v を取り出したときに、vは「起点sに最も近いがQの中で最も古い頂点ではなかった」と仮定する
  - u = Qの中で最も古い点（が、Qに一番近くはない）とする
  - a = vの親、b = uの親とする
  - vの定義より、(s→a→vの距離) < (s→b→uの距離)であり、(s→aの距離) < (s→bの距離) が成立(☆)
  - vとuの定義より、uはvより先にQに入っている。よって、uの親bはvの親aより先にQから取り出されており、すなわちbはaより先にQに加えられている
  - そのため、仮定と合わせて、(s→bの距離) ≦ (s→aの距離)。これは (☆)に矛盾する



起点に最も近い点

- Qから、「Qの中で起点に最も近い頂点を取り出す」にはどうすれば良いか？
  - Qに頂点を入れるときに、起点からその頂点までの距離（枝の数）を記録する様にする
  - Qから点を取り出すとき、毎回Qの中の全ての頂点を調べて最小の距離を持つものを探す？
    - 木の頂点数がnの場合、取り出す計算量は最悪O(n)
  - 「Qの中で最も昔にQに加えた点＝Qの中で最も起点に近い頂点」が成立している
    - Qを配列にして、Qに加えた順に頂点を整列させておく
    - O(1)で頂点を追加・取り出しが実行可能
    - この様なQをFIFOキューという

FIFOキューとは

- FIFOキュー（キュー）:
  - データ(要素)を時系列順に従って管理し、一時的に保持する為のデータ構造
    - (可変長の)配列によって実現
    - 配列に要素を追加した順に頂点を整列させておく
    - 要素を取り出す場合は、配列の中で最も昔に配列に加えた要素を取り出す
      - FIFO: First In First Out
  - FIFOキュー Q に対する操作:
    - 要素の追加:Qの最後尾に要素を追加
    - 要素の取り出し:Qの先頭から要素を取り出す

例:追加21→追加35→追加71 →取り出し→取り出し→追加19→追加92→追加65→取り出し→取り出し→追加48 →取り出し→取り出し→取り出し

21	35	71	19	92	65	48

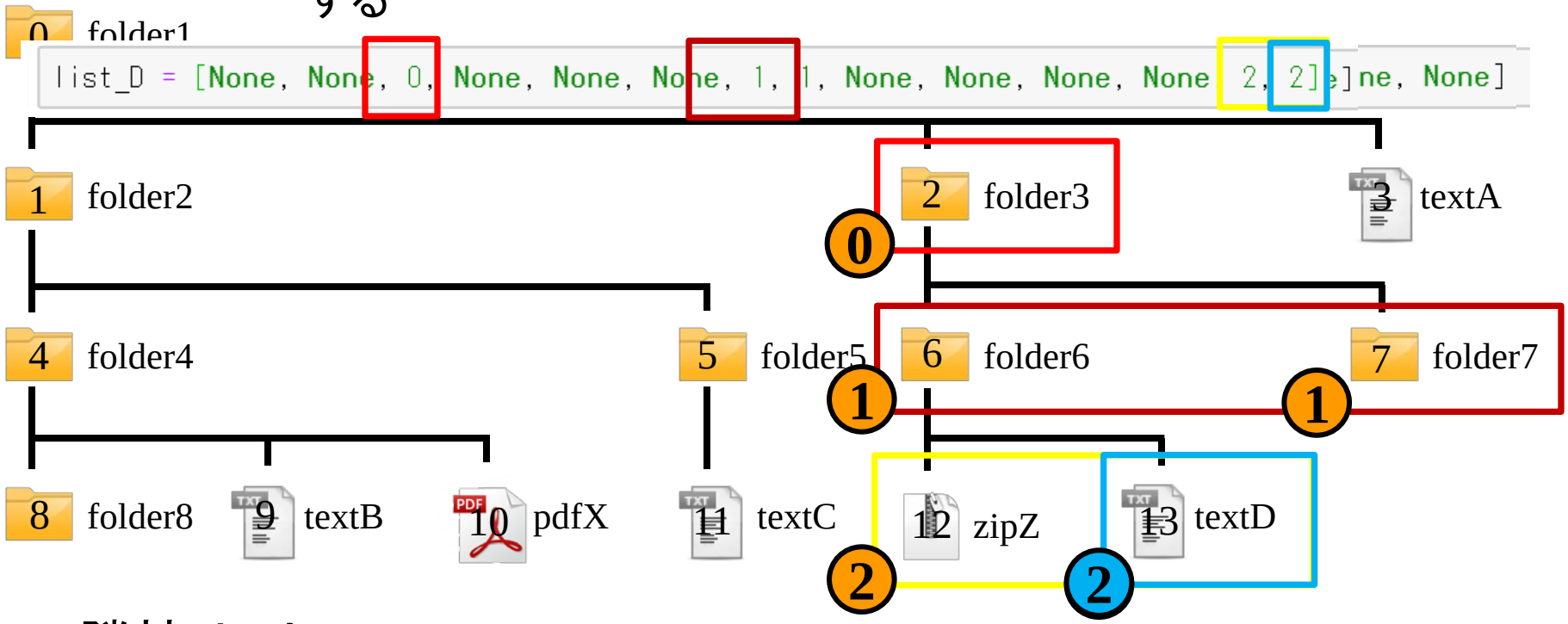
幅優先探索の実装

- Pythonで実装する場合:
  - (データ構造):
    - 木構造(隣接リスト):多重配列＝リスト
    - 幅優先探索:FIFOキュー＝リスト
  - (アルゴリズム)幅優先探索:
    - FIFOキューで点を管理
    - 目的の点を見つけるまで起点から順に点を調べる
      - 全ての点を調べる

```
#木構造の幅優先探索の典型的実装
def TreeSearch(list_adjlist):# list_adjlist=探索する木構造Tの隣接リスト
    FIFOキューQにTの根の名前を入れる
    # FIFOキューQが空になるまで探索（ループ）
    while(Qが空ではない):
        FIFOキューQから点node1を取り出す＝node1を訪問
        node1の子を全てFIFOキューQに入れる
```

指定した2点の木構造上の関係を求める

- 点folder3から点textDへの最短距離を求める
  - 2から幅優先探索を実行し、2から探索する点までの距離を記録する



- 隣接リスト:

```
[1]: [[1,2,3], [4, 5], [6, 7], [], [8, 9, 10], [11], [12, 13], [], [], [], [], [], [], []]
```

最短距離の算出

- 点vから点uまでの最短距離を知りたい場合
  - 点vを始点として、幅優先探索を行うと求まる
    - 各点に始点vからの「（最短）距離」を格納できる様にする
      - 一般的には、点の数の大きさの距離格納用の配列を用意
      - vからvまでの距離は0
      - vからvの子までの距離は1
      - vからvの子の子までの距離は2 …

幅優先探索の計算量

- データ（頂点）数が n 個の場合:
  - 全ての頂点を訪問する場合（最悪の場合） O(n)

課題

- 木構造とその探索を体験しましょう
  - basic3.ipynb : 基礎課題（25日締め切り）
  - ex3.ipynb : 本課題（30日締め切り）
    - 提出先を間違えない様にして下さい
    - 配布しているファイル（.ipynbのファイル）をそのまま提出して下さい
  - drill_03_Tree.ipynb
    - 課題の類題
  - note_networkx.ipynb
    - 木構造を扱うモジュール networkx について
      - 今回の課題を解くのには特に必要ありません
      - 木構造を可視化する為に使用されています
  - note_deque.ipynb
    - FIFOキューやスタックを効率的に実装出来るデータ型 deque について
      - 今回の課題を解くのには特に必要ありません

授業の計画（課題）

- 6/9: 第1回本課題（オプション）（-6/15）
- 6/16:第2回基礎課題(-6/18)、第2回本課題(-6/23)
- 6/23:第3回基礎課題(-6/25)、第3回本課題(-6/30)
- 6/30 :第4回基礎課題(-7/2)、第4回本課題(-7/7)
- 7/7 :第5回基礎課題(-7/9)、第5回本課題(-7/14)
- 7/14 :第6回基礎課題(-7/16)、第6回本課題(-7/21)
- 7/21 :第7回基礎課題(-7/23)、最終本課題(-8/4)