離散数学

グラフ探索アルゴリズム

落合 秀也

今日の内容

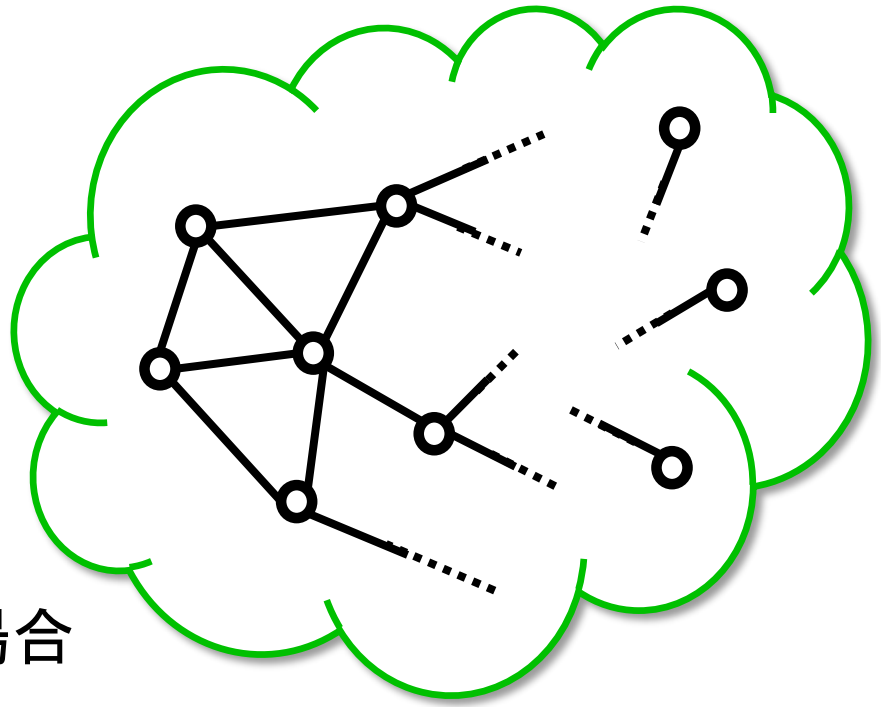
- 「グラフの連結性」の判定
- 幅優先探索
- 幅優先探索の実現方法
- 深さ優先探索
- 深さ優先探索の実現方法
- 木の構造
- 探索木
- パトリシア・トライ

連結性の判定問題を考える

- グラフG(V,E)が与えられたとき、
  Gが連結かどうか、を判定したい。

- 小さいグラフなら、
  紙に書いてみればよい

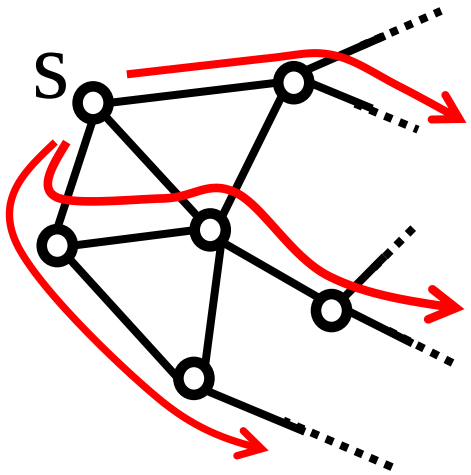
- 一般には簡単ではない
  - 大きいグラフの場合
  - コンピュータに判断させる場合



グラフG

コンピュータに判定させるには

- 次の方法で判定できそう:
  - ある頂点sを開始点とし、辺をたどって、動けるだけ動き回る（Traverseする）。
    - すべての頂点に到達(reach)できれば、連結である。
    - 到達可能な範囲をすべて動き回っても、まだ到達できていない頂点があれば、連結でない。



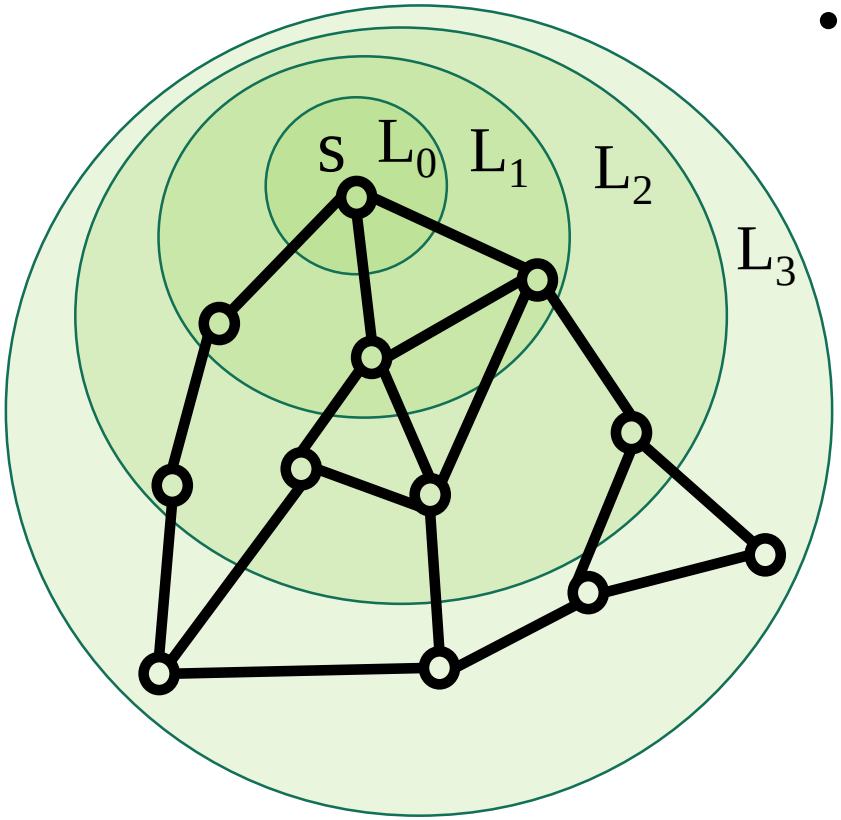
すべて到達可能 → 連結である

到達できない頂点が残る → 連結でない

今日の内容

- 「グラフの連結性」の判定
- 幅優先探索
- 幅優先探索の実現方法
- 深さ優先探索
- 深さ優先探索の実現方法
- 木の構造
- 探索木
- パトリシア・トライ

幅優先探索(Breadth-First Search)



- 基本的な考え方
  - 頂点sから開始・・・L_0とする
  - L_0から、外向きに辺をつたい、接続する頂点を取り込む ・・・L_1とする
  - L_1から、外向きに辺をつたい、接続する頂点を取り込む・・・L_2とする
  - L_2から、外向きに辺をつたい、接続する頂点を取り込む・・・L_3とする

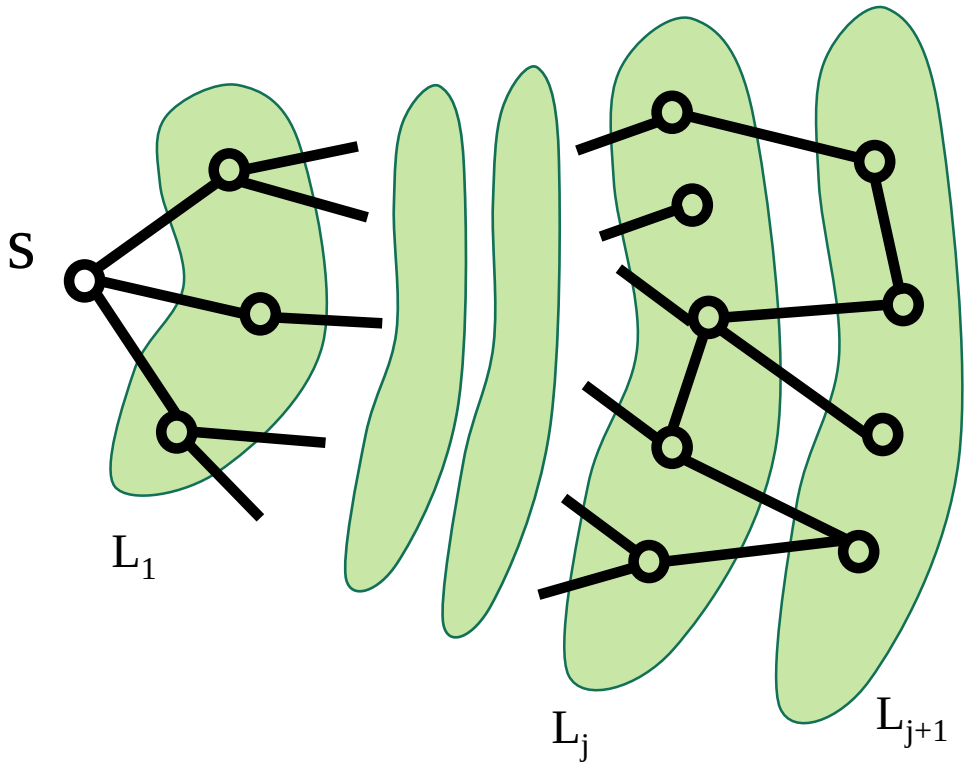
    ・・・

頂点sから湧き出す水によって引き起こされる洪水（Flood）が到達可能な頂点すべてに伝搬するイメージ

幅優先探索 (もう少し正確な定義)

層(Layer) $L_1, \ldots, L_n$ を次のように作成する

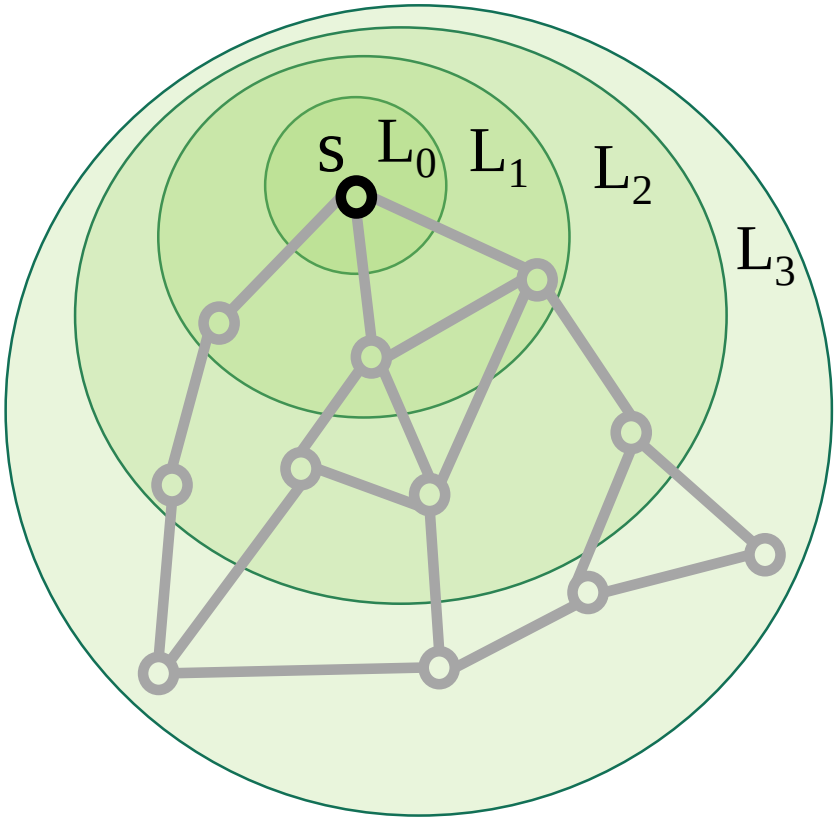
- 開始点sの近隣の頂点（の集合）をL_1とする
- $L_1, \ldots, L_j$ がすでに作られているとき、L_j に辺でつながっていて、$L_1, \ldots, L_j$ に含まれない頂点（の集合）をL_{j+1}とする



- sからL_jの頂点までの道(path)は存在する。
- L_jの頂点は、sからjの距離にある。
- sから頂点tまでの道が存在すれば、頂点tはいずれかの層に属する。

幅優先探索によって作られる木

- L_{j+1}を作る実際的な方法：
  - L_jの各頂点に辺で接続する頂点が、L_{j+1}に新たに含まれるか、否かを判定する



ここで、
「新たにL_{j+1}に含まれる場合」の辺で、グラフを作っていくと、木になる。

※ 右図の場合で、検証してみよう
（コンピュータになった気分で1つずつ、試してみよう）

幅優先探索木と呼ばれる

幅優先探索の終了

- 幅優先探索の終了条件
  - 目的1 -- 特定の頂点(例: 頂点 t) を発見する場合
    - 頂点 t が見つかった時点で終了
    - 探索可能なすべての頂点を巡回した時点で終了
  - 目的2 -- グラフが連結かどうかを判定する場合
    - 探索可能なすべての頂点を巡回した時点で終了
- 幅優先探索が終了しないケース
  - 頂点 t が見つからない(or 特定の頂点を探していない) かつ グラフが無限大に大きい

今日の内容

- 「グラフの連結性」の判定
- 幅優先探索
- 幅優先探索の実現方法
- 深さ優先探索
- 深さ優先探索の実現方法
- 木の構造
- 探索木
- パトリシア・トライ

幅優先探索を実現する

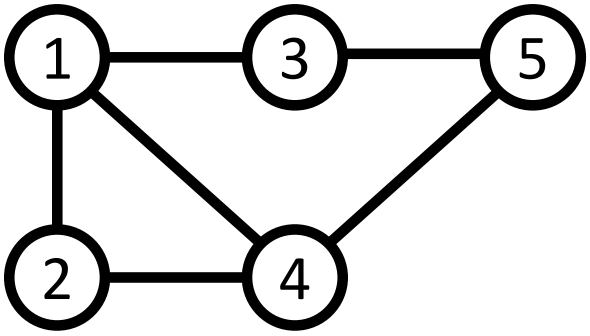
- 幅優先探索を実現するアルゴリズムを設計する

- 予備知識１
  - グラフの隣接リストによる表現

- 予備知識２
  - キュー(Queue)の働きと使い方

幅優先探索を実現する
グラフを隣接リストで表現する

- 隣接リスト(adjacency list)
  1. ある頂点vの、近隣の頂点をリストで表現したもの: Adj[v]
  2. 全頂点に対して、同様のリストを作る: Adj – 配列

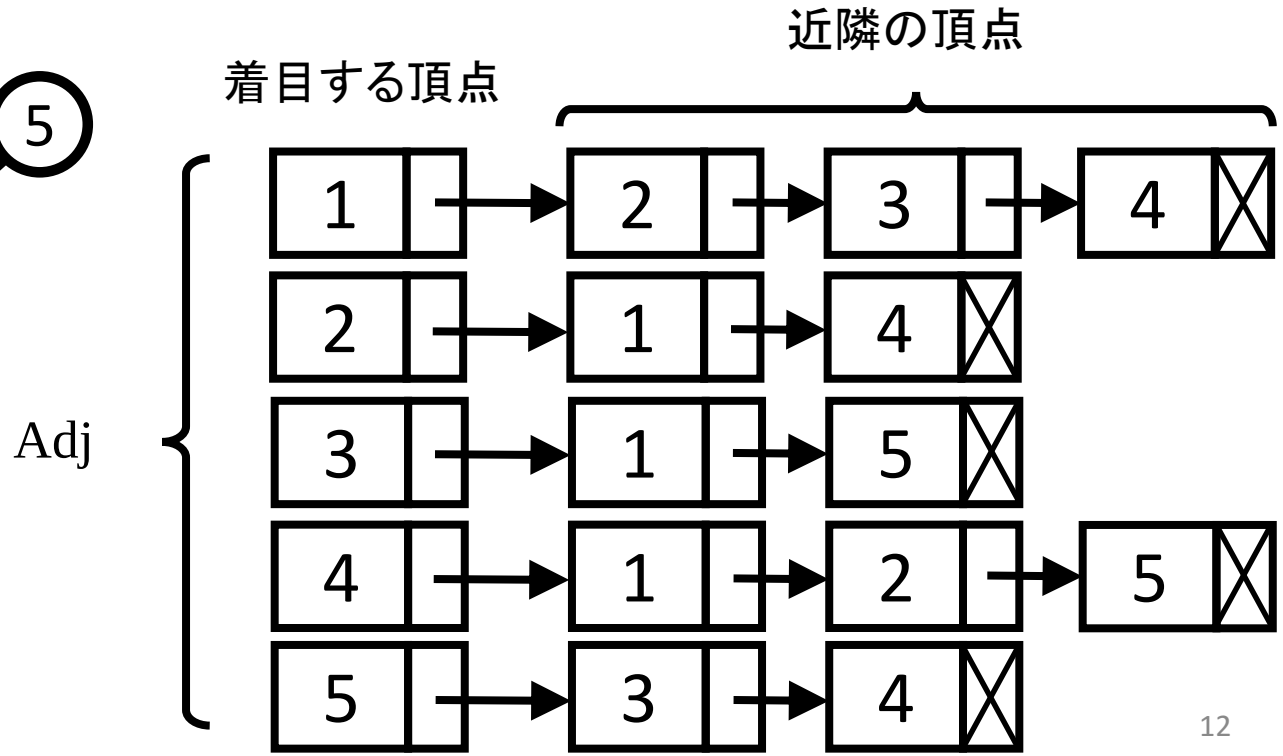


Adjの大きさは

$O(n+m)$

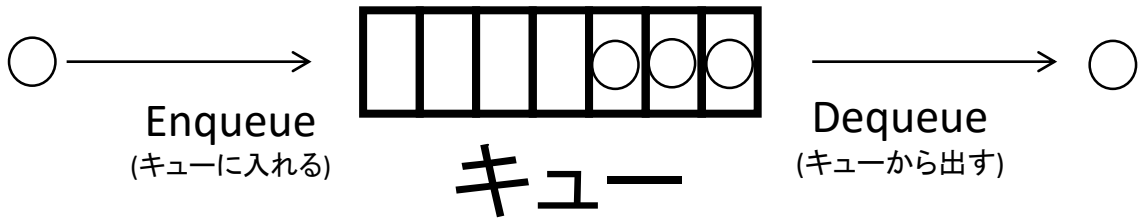
n: 頂点の数

m: 辺の数



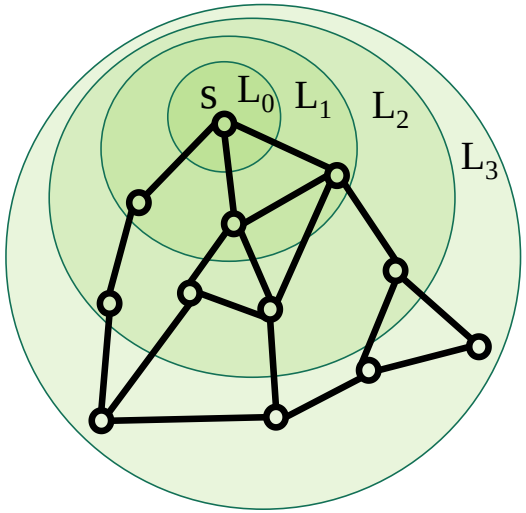
幅優先探索を実現する キュー(Queue)の働きと使い方

- 先入れ先出し装置: Fast In Fast Out (FIFO)



- 入れた順番に取り出せるメモリ:
  - 3, 1, 2, 8, 6, 5, 4 の順に投入すれば、
    3, 1, 2, 8, 6, 5, 4 の順に出てくる

幅優先探索アルゴリズムの設計



- 準備するもの
  - グラフGの隣接リスト配列: Adj
    - 頂点vの隣接ノードは Adj[v]
  - 訪問判別フラグ: F[v]
    - vに訪問済みなら、F[v] = true の値を取る
    - vに訪問していなければ、F[v] = false の値を取る (初期値)
  - キュー: Q
    - キューにvを入れる操作

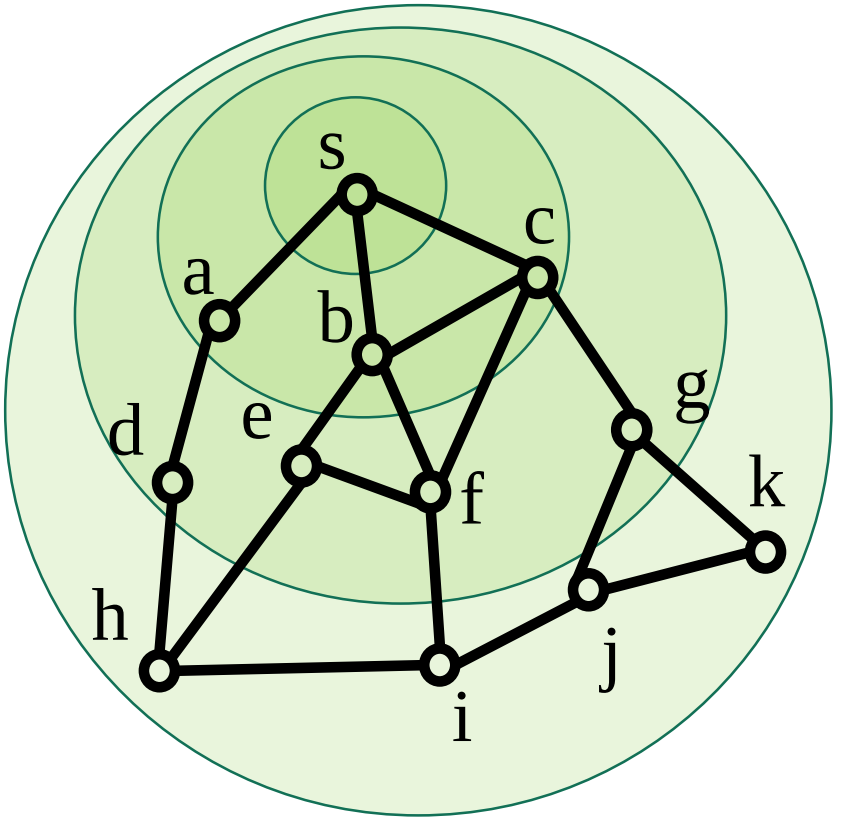
      Q.enqueue(v)
    - キューから取り出したものを、vとする操作

      v := Q.dequeue()

幅優先探索アルゴリズム

開始点sから 幅優先探索を行うアルゴリズム

```
F[s] := true
Q.enqueue(s)
While Q is not empty
  v := Q.dequeue()
  Foreach node u in Adj[v]
    If F[u] = false Then,
      F[u] := true
      Q.enqueue(u)
    EndIf
  EndFor
EndWhile
```



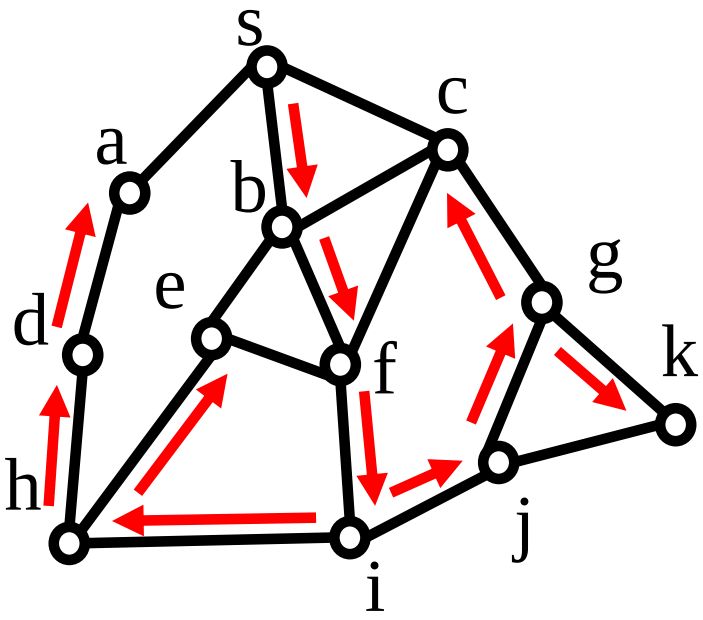
時間計算量：O(n+m)

(*) 厳密には初期化処理が必要だが省略している

今日の内容

- 「グラフの連結性」の判定
- 幅優先探索
- 幅優先探索の実現方法
- 深さ優先探索
- 深さ優先探索の実現方法
- 木の構造
- 探索木
- パトリシア・トライ

深さ優先探索 (Depth-First Search)

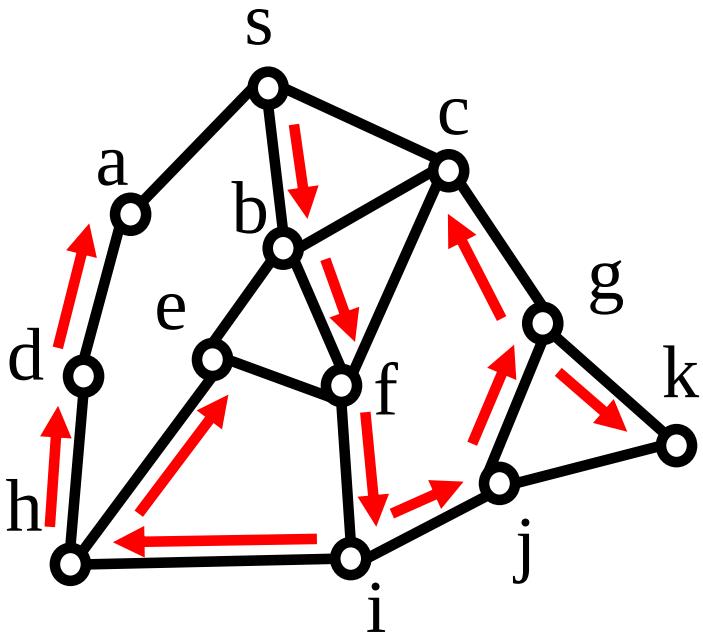


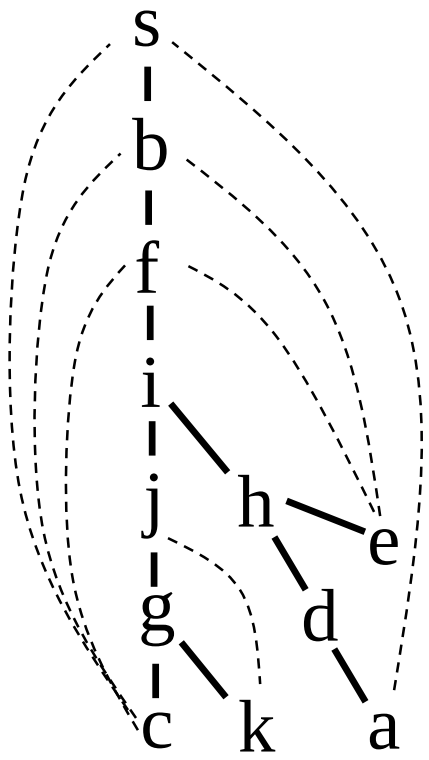
- 基本的な考え方
  - 開始点sから、辺をたどって行きつくところ（=dead end）まで行ってみる（ただし同じ頂点は取らない）
  - 行き止まり(=dead end)に当たれば、1つ戻って、別の辺をたどってみる
  - 1つ戻ってもダメなら、2つ戻ってみる
  - 2つ戻ってもダメなら、3つ戻ってみる
  - …

頂点sから歩き始めた人が、すべての行き止まりにあたるまで、隈なく歩いていくイメージ

深さ優先探索によって作られる木

- 出来るだけ深いところまで一気に作る
- 戻りながら、別に行ける頂点があれば、そちらの枝を作る





深さ優先探索木と呼ばれる

深さ優先探索の終了

- 深さ優先探索の終了条件
  - 目的1 -- 特定の頂点(例: 頂点 t) を発見する場合
    - 頂点 t が見つかった時点で終了
    - 探索可能なすべての頂点を巡回した時点で終了
  - 目的2 -- グラフが連結かどうかを判定する場合
    - 探索可能なすべての頂点を巡回した時点で終了
- 深さ優先探索が終了しないケース
  - 頂点 t が見つからない(or 特定の頂点を探していない) かつ グラフが無限大に大きい

今日の内容

- 「グラフの連結性」の判定
- 幅優先探索
- 幅優先探索の実現方法
- 深さ優先探索
- 深さ優先探索の実現方法
- 木の構造
- 探索木
- パトリシア・トライ

深さ優先探索を実現する

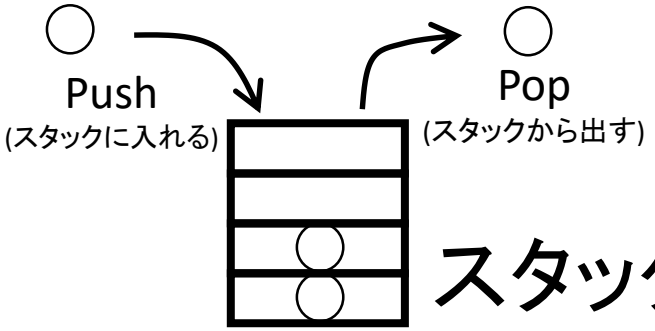
- 深さ優先探索を実現するアルゴリズムを設計する

- 予備知識１
  - グラフの隣接リストでの表現

- 予備知識２
  - スタック(Stack)の働きと使い方

深さ優先探索を実現する
スタック(Stack)を利用する

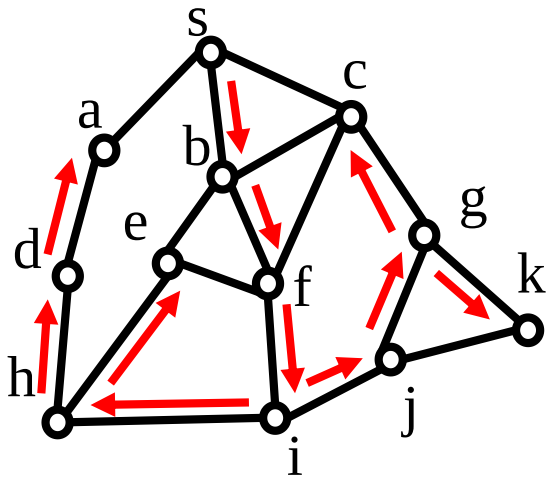
- 後入れ先出し装置： Last In Fast Out (LIFO)



- 入れた順番の逆に取り出せるメモリ：
  (1) 3, 1, 2を投入
  (2) 2個取り出す
  (3) 8, 6 を投入
  (4) 3個取り出す
  (5) 5, 4 を投入
  (6) 2個取り出す

取り出される列は
2, 1, 6, 8, 3, 4, 5
となる

深さ優先探索アルゴリズムの設計



- 準備するもの
  - グラフGの隣接リスト配列: Adj
    - 頂点vの隣接ノードは Adj[v]
  - 訪問判別フラグ: F[v]
    - vに訪問済みなら、F[v] = true の値を取る
    - vに訪問してなければ、F[v] = false の値を取る (初期値)
  - スタック: S
    - スタックにvを入れる操作

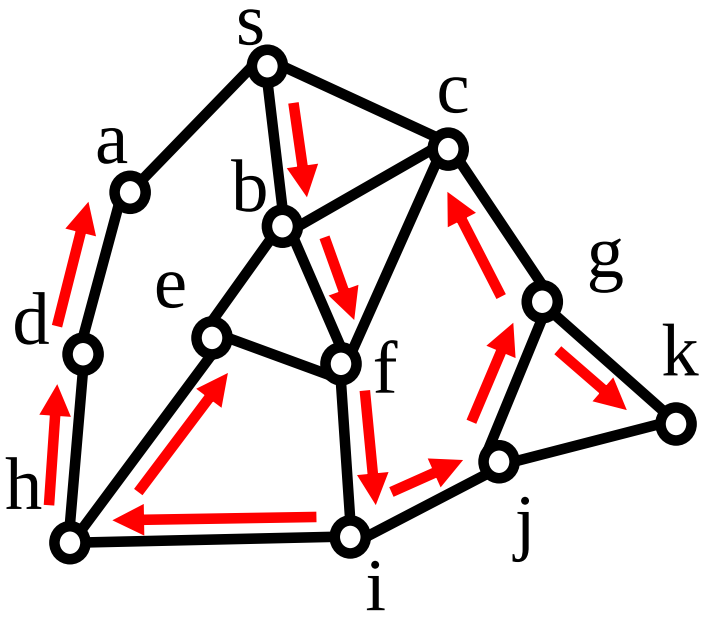
      S.push(v)
    - スタックから取り出したものを、vとする操作

      v := S.pop()

深さ優先探索アルゴリズム

開始点sから 深さ優先探索を行うアルゴリズム

```
F[s] := true
S.push(s)
While S is not empty
  v := S.pop()
  If F[v] = false Then,
    F[v] := true
    Foreach node u in Adj[v]
      If F[u] = false Then,
        S.push(u)
      EndIf
    EndFor
  EndIf
EndWhile
```



時間計算量：O(n+m)

(*) 厳密には初期化処理が必要だが省略している

深さ優先探索アルゴリズム2
再帰呼び出しによる方法

深さ優先探索を行うアルゴリズム（再帰呼び出し版）

```
Function DFS(v)
  If F[v] = false Then,
    F[v] := true
    Foreach node u in Adj[v]
      If F[u] = false Then,
        DFS(u)
      EndIf
    EndFor
  EndIf
EndFunction
```

頂点sから探索を行う場合：

DFS(s)

を実行すれば良い

再帰呼び出しは、実際には、裏ではスタックを使って実行される

幅優先探索と深さ優先探索の比較

幅優先探索

```
F[s] := true
Q.enqueue(s)
While Q is not empty
  v := Q.dequeue()
  Foreach node u in Adj[v]
    If F[u] = false Then,
      F[u] := true
      Q.enqueue(u)
    EndIf
  EndFor
EndWhile
```

深さ優先探索

```
F[s] := true
S.push(s)
While S is not empty
  v := S.pop()
  If F[v] = false Then,
    F[v] := true
    Foreach node u in Adj[v]
      If F[u] = false Then,
        S.push(u)
      EndIf
    EndFor
  EndIf
EndWhile
```

今日の内容

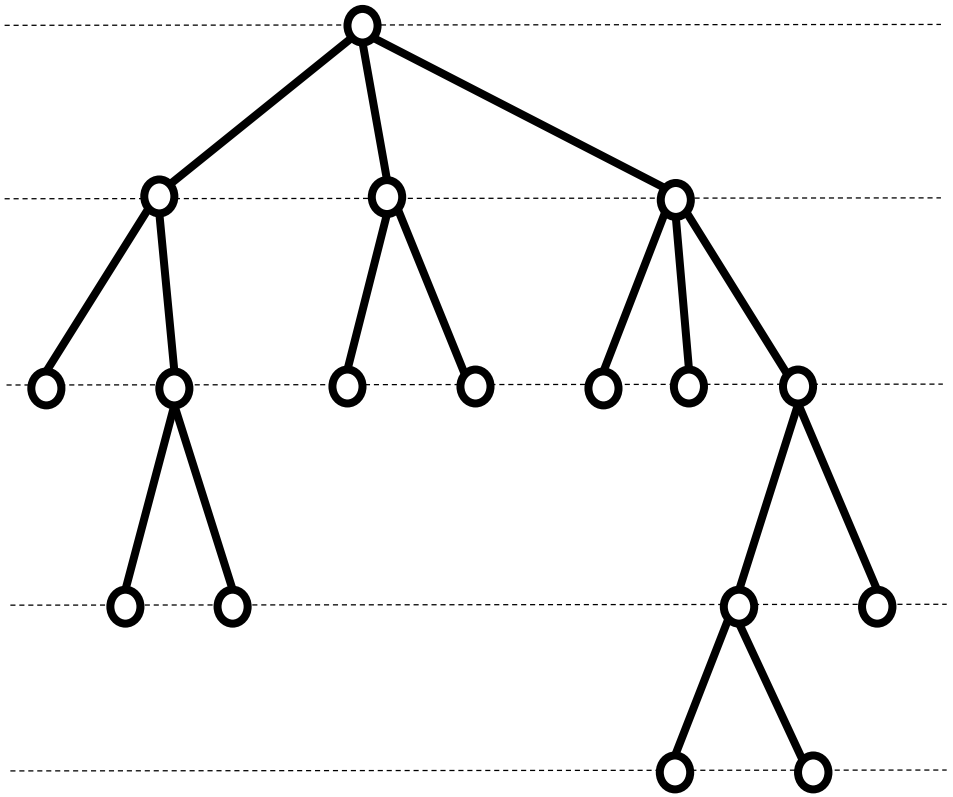
- 「グラフの連結性」の判定
- 幅優先探索
- 幅優先探索の実現方法
- 深さ優先探索
- 深さ優先探索の実現方法
- 木の構造
- 探索木
- パトリシア・トライ

木の構造

- 根(root)
- 枝(branch)
- 葉(leaf)

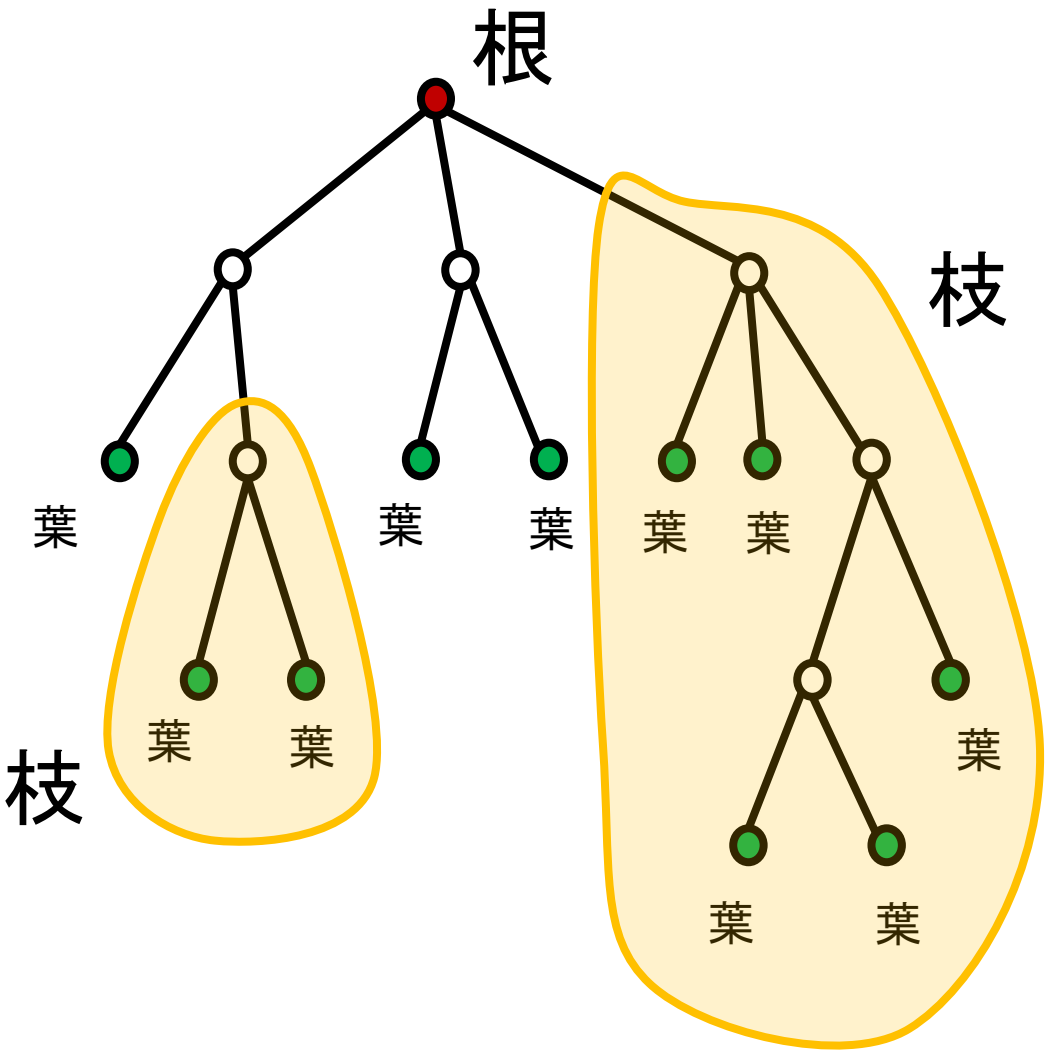
- 親(parent)
- 子(child)
- 先祖(ancestor)
- 子孫(descendant)

- 深さ(depth, level)



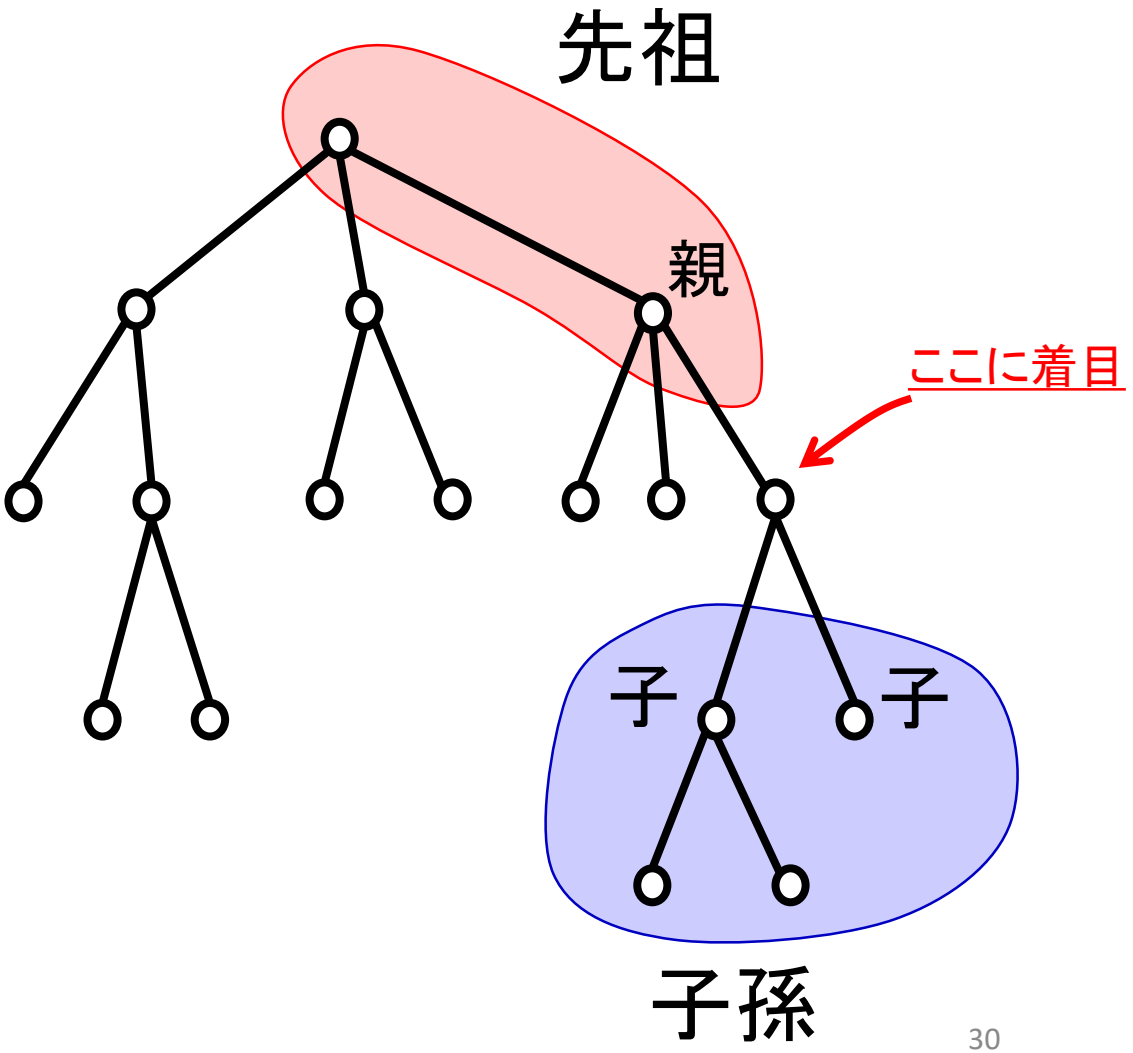
木(根付き木)の構造

- 根(root)
- 枝(branch)
- 葉(leaf)

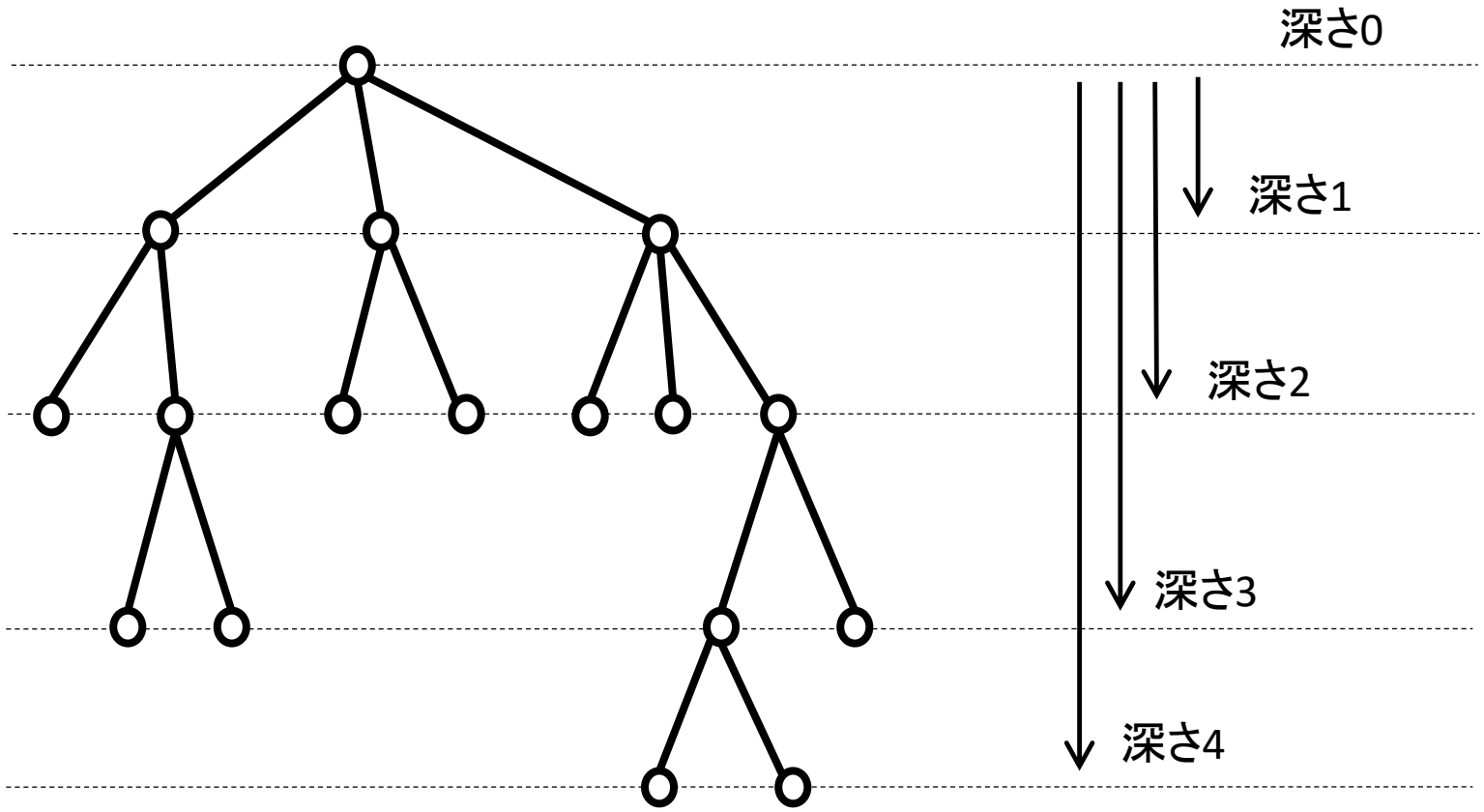


木(根付き木)の構造

- 親(parent)
- 子(child)
- 先祖(ancestor)
- 子孫(descendant)



木の深さ (depth, level)

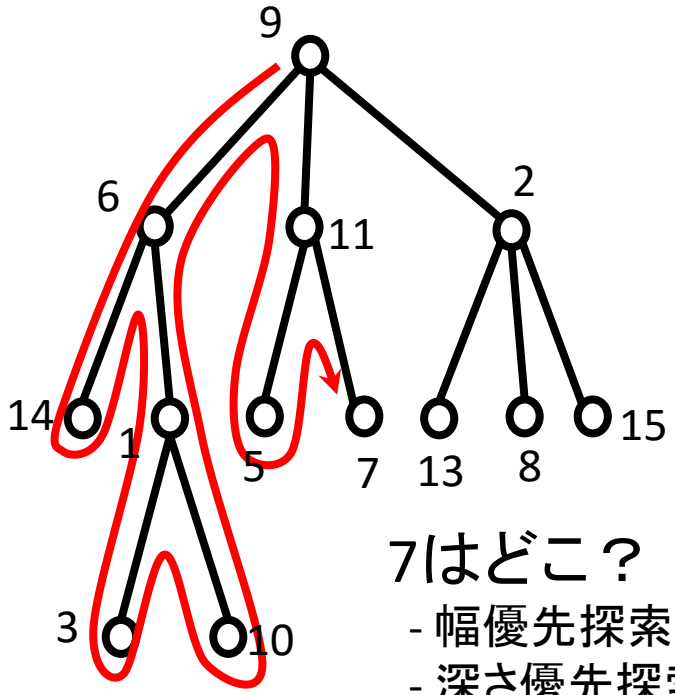


今日の内容

- 「グラフの連結性」の判定
- 幅優先探索
- 幅優先探索の実現方法
- 深さ優先探索
- 深さ優先探索の実現方法
- 木の構造
- 探索木
- パトリシア・トライ

探索木

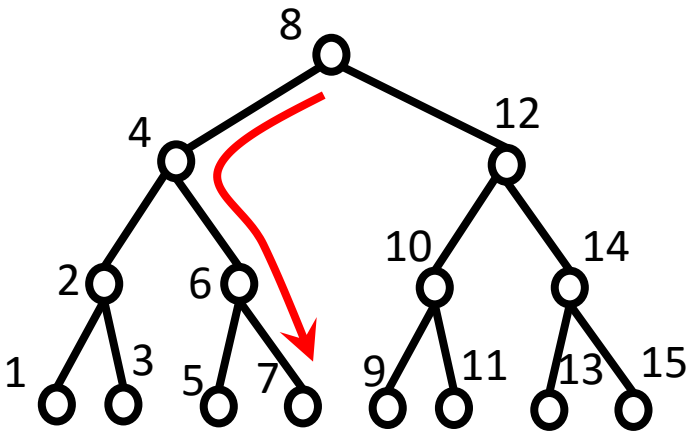
- 木の組換えを許すとする
- このとき、何らかの規則に沿って木を組み換えれば、探索効率を向上できることがある



適当な木

7はどこ？
- 幅優先探索
- 深さ優先探索

などで探す



整理整頓された木

7はどこ？
- 8よりは小さい
- 4よりは大きい
- 6よりは大きい

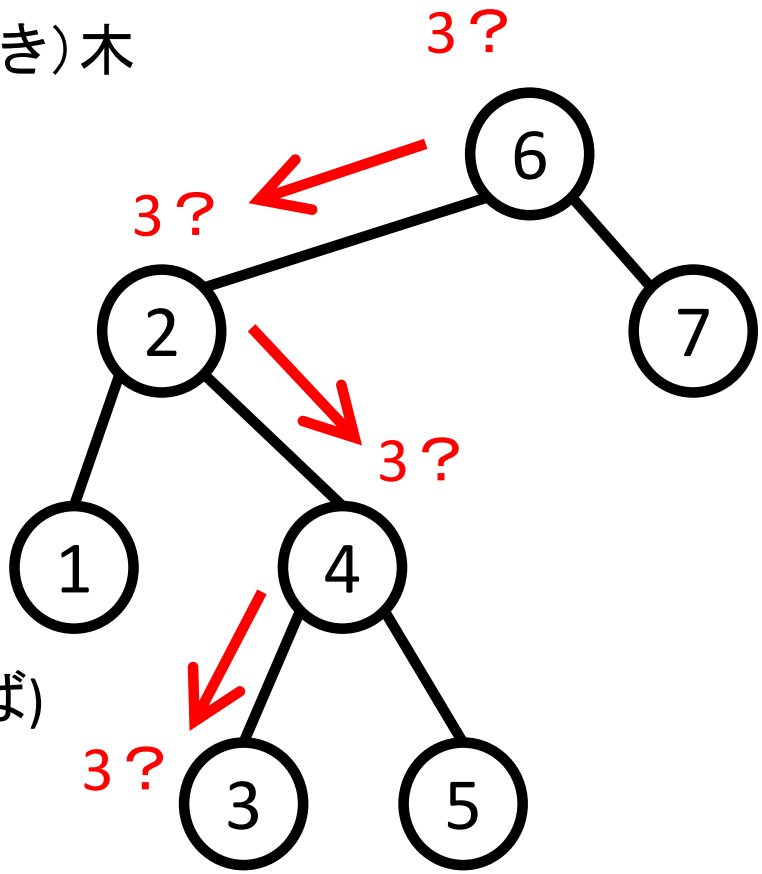
二分探索木

- 二分木
  - 各ノードが高々2個の子を持つ（根付き）木
  - 片方を左、もう片方を右 と呼ぶ

- 二分探索木
  - ノードの値を n とするとき
    左枝の各ノードの値 < n
    右枝の各ノードの値 > n
    となるように構成された木

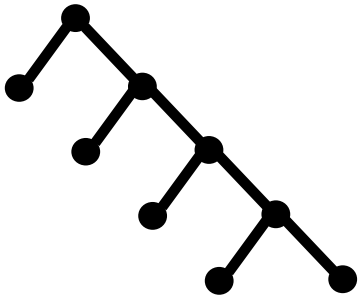
  - 探索したい値を s とするとき
    以下の方法で発見できる(存在すれば)

    Function 二分探索(s,n)
    s=n → 探索終了
    s<n → 左の子lに対して 二分探索(s,l) を実行
    s>n → 右の子rに対して二分探索(s,r) を実行



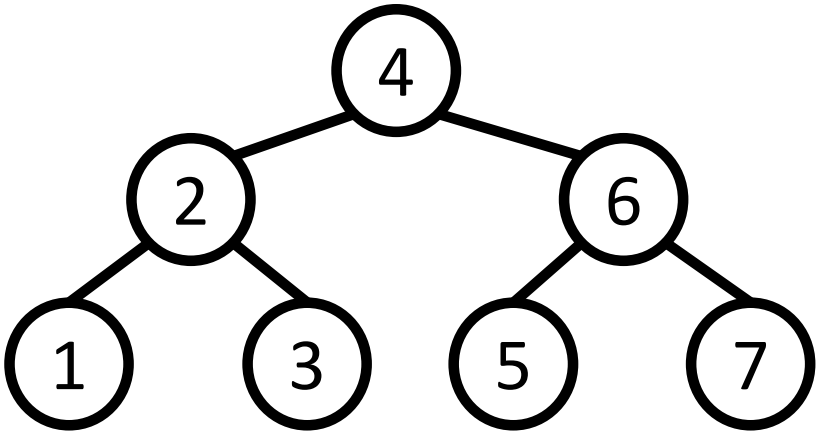
平衡二分探索木

- 二分探索木は、深さがアンバランスになることがある



浅いノードを探索した場合は、効率がよいが、
最悪の場合、最も深いところまで探索する必要がある
つまり最悪の場合、O(n) となる (nは頂点の数)

- 平衡二分探索木は、深さができるだけ等しくなるように構成された木



深さは log n なので、
O(log n)
で処理できる

今日の内容

- 「グラフの連結性」の判定
- 幅優先探索
- 幅優先探索の実現方法
- 深さ優先探索
- 深さ優先探索の実現方法
- 木の構造
- 探索木
- パトリシア・トライ

パトリシア・トライ (Patricia Trie)

Practical Algorithm to Retrieve Information Coded in Alphanumeric

- 辞書式順序を定義可能なノード探索のための探索木

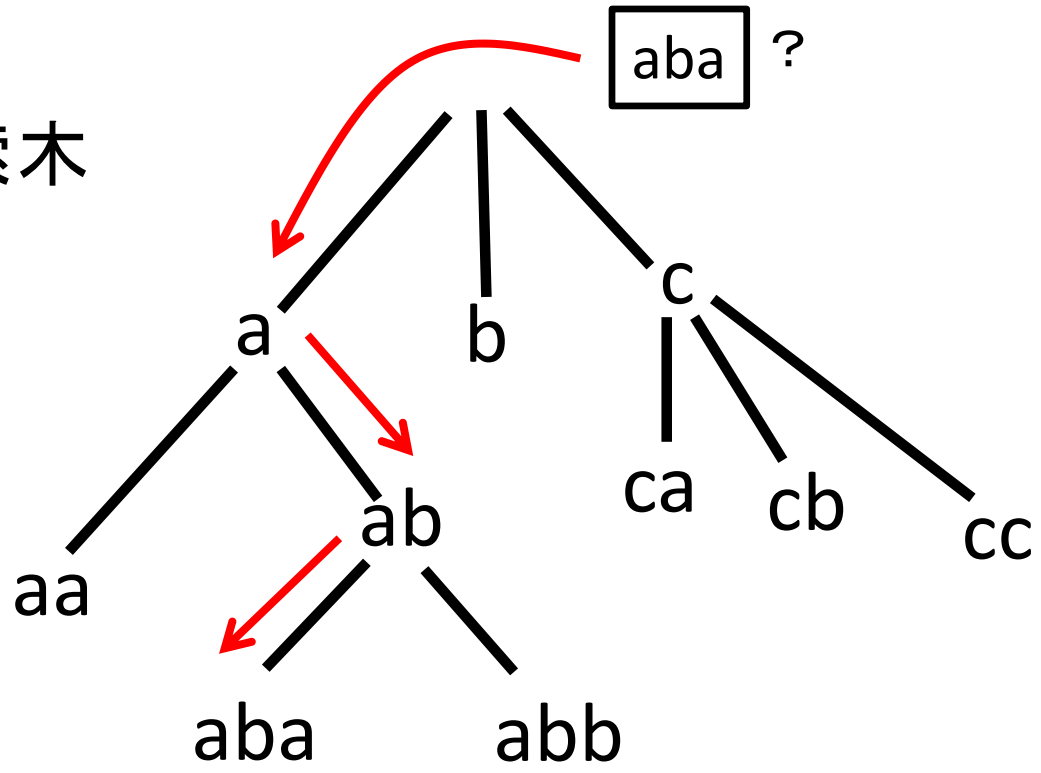
- 例）以下のノードの探索木

  a, aa, ab, aba, abb,

  b, c, ca, cb, cc

- 例) abaを検索する場合
  - 1文字目(a)で振り分け
  - 2文字目(b)で振り分け
  - 3文字目(a)で振り分け

    abaに到達する



DNSの名前体系と探索木

- DNSの名前体系(例: www.u-tokyo.ac.jpなど)は、インターネット上にパトリシアを構築して管理されている
- 根(root)は . のみで表現される
- 深さ1のノードに、

  .com.　　.jp.　　.net.　　.org.

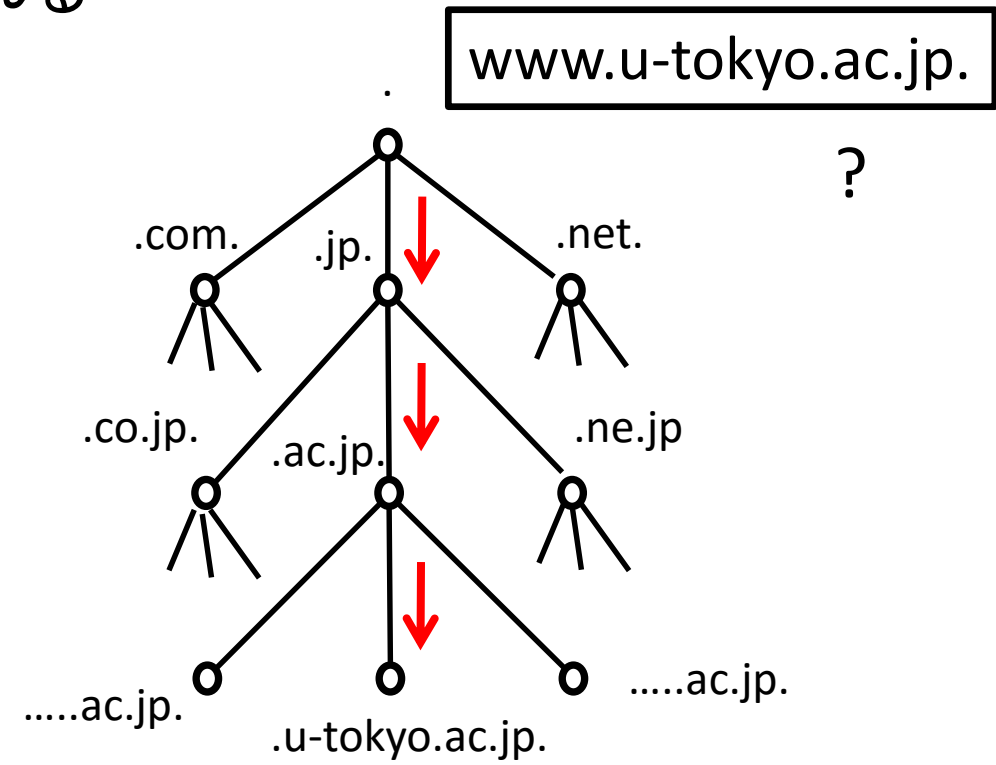
  などがある
- .jp.の子に、

  .co.jp.　.ac.jp.　.ne.jp.

  などがある
- .ac.jp. の子に、

  .u-tokyo.ac.jp.

  がある



今日の内容

- 「グラフの連結性」の判定
- 幅優先探索
- 幅優先探索の実現方法
- 深さ優先探索
- 深さ優先探索の実現方法
- 木の構造
- 探索木
- パトリシア・トライ