# データ構造

2022/04/27

スタートアップゼミ #05

D1 黛風雅 / Fuga MAYUZUMI

# 目次

# データ構造とは



## データ構造

データをどのようにコンピュータのメモリに並べるかという方式・形式
適切なアルゴリズム・データ構造の選択→計算の高速化・汎用化

# オーダー記法



## オーダー記法(ランダウの記号)

…関数の極限における値の変化を大まかに評価するための記法

- 影響力の大きい(次数の大きい)項以外は無視する
- 定数倍の差は無視する

### 漸近的上界[ビッグ・オー記法:$O$]

十分大きな $n$ について，定数倍を無視すれば計算量 $T(n)$ は $f(n)$ が上限となる

$$T(n) = O\big(f(n)\big) \quad \leftrightarrow$$

ある実数 $c$ と自然数 $n_0$ が存在して，
全ての $n \geq n_0$ に対して $T(n) \leq c \cdot f(n)$ が成り立つ

例. $T(n) = 2n^2 + 3n + 1$ のとき，$f(n) = n^2$ $(2n^2 + 3n + 1 = O(n^2))$

※$f(n)$ は上限であるため，$f(n) = n^3$ でも成り立つ

→アルゴリズムの計算量の評価に用いられる

あるアルゴリズムについて $n$ 個の入力データに対して必要な計算量のオーダーを示す

### cf. 漸近的下界[ビッグ・オメガ記法:$\Omega$]

十分大きな $n$ について，定数倍を無視すれば計算量 $T(n)$ は $g(n)$ が下限となる

$$T(n) = O\big(g(n)\big) \quad \leftrightarrow$$

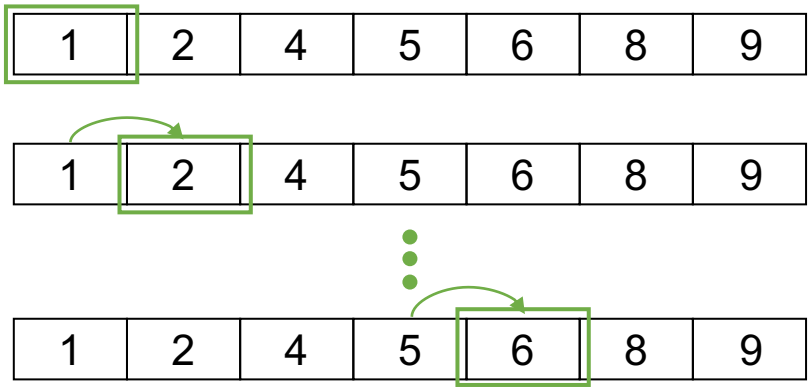
ある実数 $c$ と自然数 $n_0$ が存在して，
全ての $n \geq n_0$ に対して $T(n) \geq c \cdot f(n)$ が成り立つ

# 例. 線形探索と二分探索



## 線形探索

探索値: 6




- 端点から1つずつ順番に探索
- 6に到達するまでに5回の操作が必要

## 二分探索

※昇順/降順にソートされた配列に対して適用可能

探索値: 6


- 配列の中間の値をチェックし，探索対象とその値の大小を比較する
- 探索範囲を絞り込んで探索していく
- 6に到達するまで3回の操作が必要

## 計算量の比較

線形探索の場合・・・ 1番目から$n$番目まで最大$n$回全て探索する $\rightarrow O(n)$

二分探索の場合・・・ $n$個のデータのとき最大分割数 $\log_2 n$回 $\rightarrow O(\log_2 n)$

# 最短経路探索の計算量



## Dijkstra法のアルゴリズム

$|V|$ :ノード数，$|E|$ :リンク数 とする．

1. 始点$s$以外の全てのノードを終点とし，リンク $l_{ij}$の長さを$d_{ij}$とする．
2. 始点ノードに永久ラベル$u_0^* = 0$を与える．$i = 0$とする．
   始点ノード以外のノードは一時ラベルを $\infty$ に設定する．
   ➡ 一時ラベルを与えたノード数 … $|V|$
3. ノード$i$を始点に持つ全てのリンク$l_{ij}$に対して，
   終点ノード$j$が永久ラベルを持っていないとき，
   i. ノード$j$が一時ラベルを持っていない場合，
      一時ラベル$u_j = u_i^* + d_{ij}$を与える．
   ii. ノード$j$が一時ラベル$u_j$を持っており，その一時ラベル$u_j$が$u_i^* + d_{ij}$よりも大きい場合，この一時ラベルを$u_i^* + d_{ij}$の値に改訂する．
   ➡ 合計で $|E|$ 個のリンクが一回ずつ計算される
4. 一時ラベル全体の中で最小の値を持つもの1つ選び，$i$をそのノード番号とする．また，一時ラベル$u_i$を永久ラベル$u_i^*$に変える．
   ➡ 走査：最大 $|V|$ 個のノードから最小ラベル値のノードを選択
5. すべての集中ノードに永久ラベルがつくまで，3・4を繰り返す．
   ➡ 走査を $|V|$ 回繰り返し

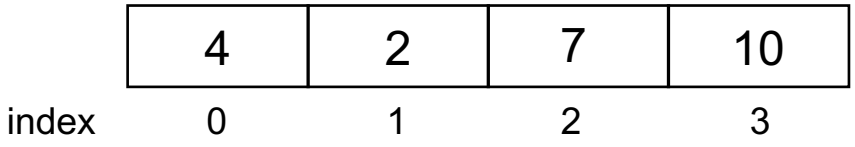
▷ **$O(|V|^2 + |E|)$ の計算量を要するアルゴリズム**

# 配列と連結リスト



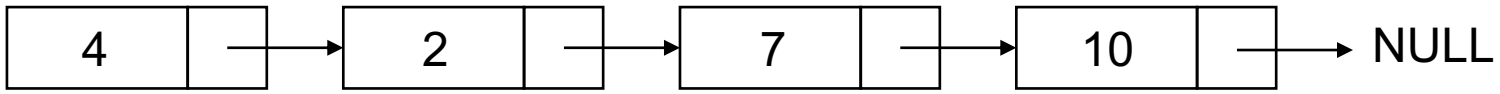
## 配列

配列全体に対して1つのメモリブロックが割り当てられる.
配列要素はindexを添字として用いることで一定時間でアクセス可能.
**利点**:個々の要素に$O(1)$で高速アクセス
**欠点**:配列サイズが固定である(使う前にサイズ指定)




## 単一連結リスト

ノードが次の要素を指す next ポインタ を持つ複数の節点からなる.
リスト終端のノードのリンクはNULL <- リストの末尾であることを表す.
**利点**:定数時間で拡張が可能. <-> 配列では新しい配列を作らなくてはならない
**欠点**:個々の要素へのアクセスに $O(n)$ がかかる.

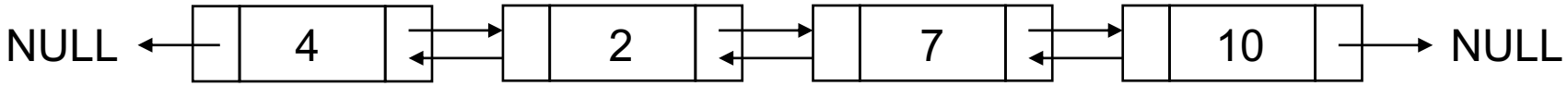


# 双方向連結リスト



## 双方向連結リスト(二重結合リスト)

直前ノードへの previousポインタ も持つ.
利点:前方向にも辿れる(Dijkstra法における最短経路探索への適用)
欠点:各ノードが余分なポインタを持つためにスペースがさらに必要になる




cf.
構造体型:
(C言語)

```
struct 構造体名 {
  型名 識別子;
  型名 識別子;
  ・・・
};
```

複数の変数をまとめるもの

```
/*ノードの情報を格納するNODE構造体型*/
struct NODE {
    char ID[40]; //ノードID
    double lat; //緯度
    double lon; //経度
    int state; //dijkstra法で使う状態変数
    double mincost; //dijkstra法で使う最小コスト
    struct LINK *beforelink; //dijkstra法で使う前リンク
    struct LINKLIST *nextlinklist; //このノードを起点とするリンク
    struct NODE *next;
};
```

# スタックとキュー



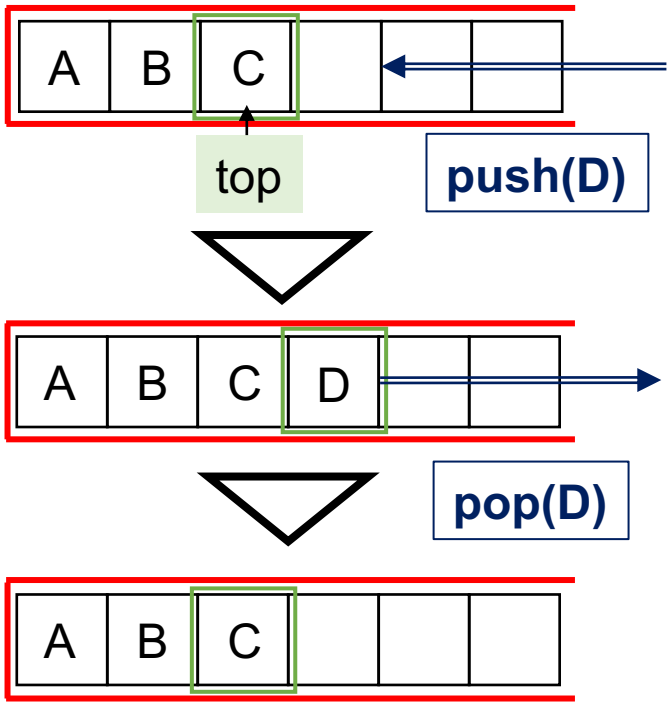
- 追加された処理待ちデータをどの順番で取り出すかが異なる
- スタックは **LIFO** (Last-In-First-Out), キューは **FIFO** (First-In-First-Out)

## スタック

データ構造に入っている要素のうち，**最後**に追加した要素を取り出す

topの端点で挿入/削除が行われる

- push(x): 要素 x をデータ構造に追加する
- pop(x): データ構造から要素 x を取り出す

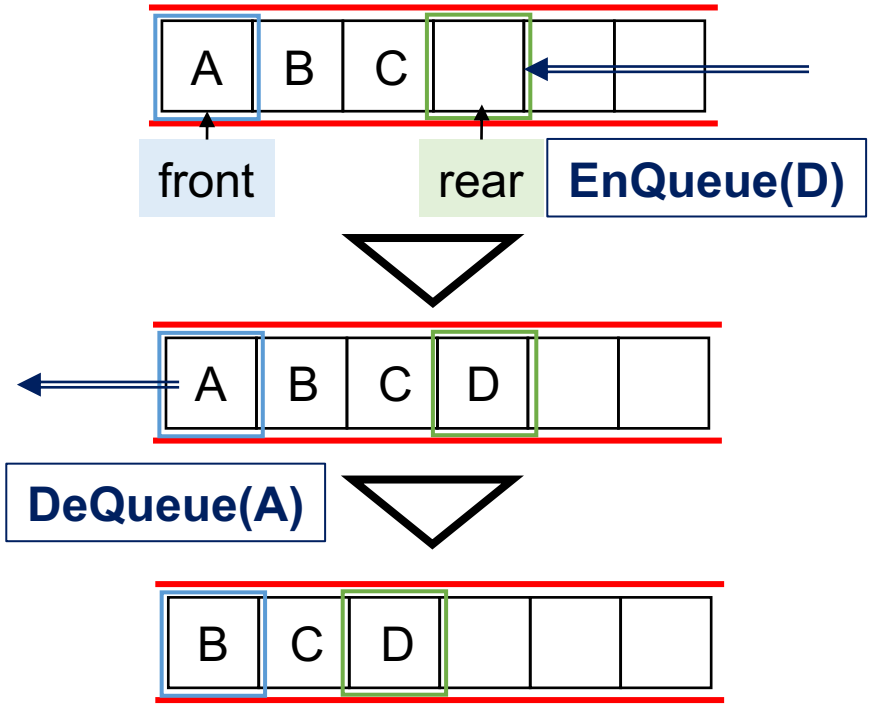



## キュー

データ構造に入っている要素のうち，**最初**に追加した要素を取り出す

挿入が後方:rearから行われ，
削除が前方:frontから行われる

- EnQueue(x): 要素 x を追加する
- DeQueue(x): 要素 x を取り出す



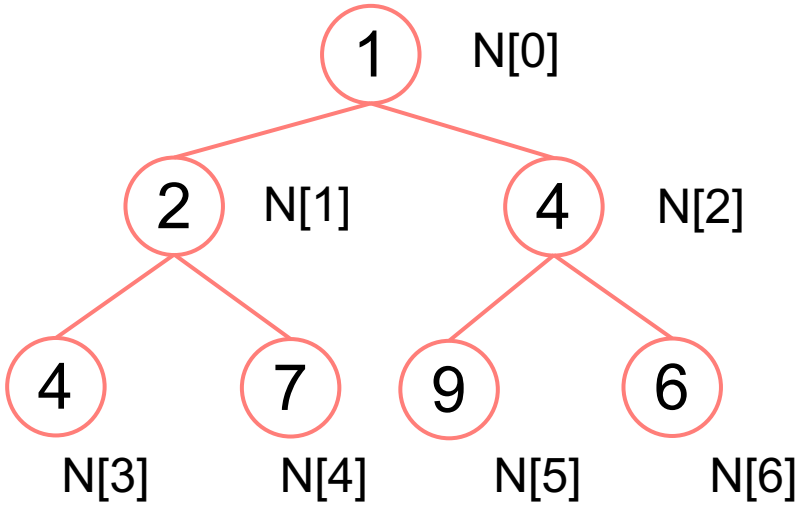
# 優先度付きキューとヒープ



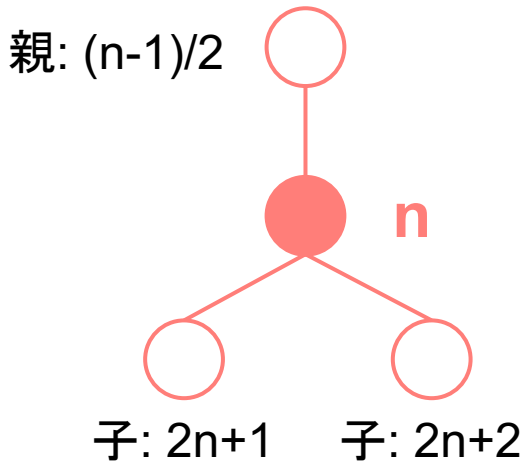
- 要素の集まりから最小/最大要素を見つける必要がある状況で活用
- 優先度付きキューでは，演算Insert，DeleteMin/DeleteMax(最小/最大要素を取り除いてそれを返す)が出来る
- 常に根がその木の中の最小値を格納している
  →最小値取り出しの計算量はO(1)
- **処理順序が要素の入力順とはならない点で，キューの演算と異なる**

## ヒープ(Heap)構造

- 親の要素は子の要素より小さい
- 木の根の要素は最小の要素



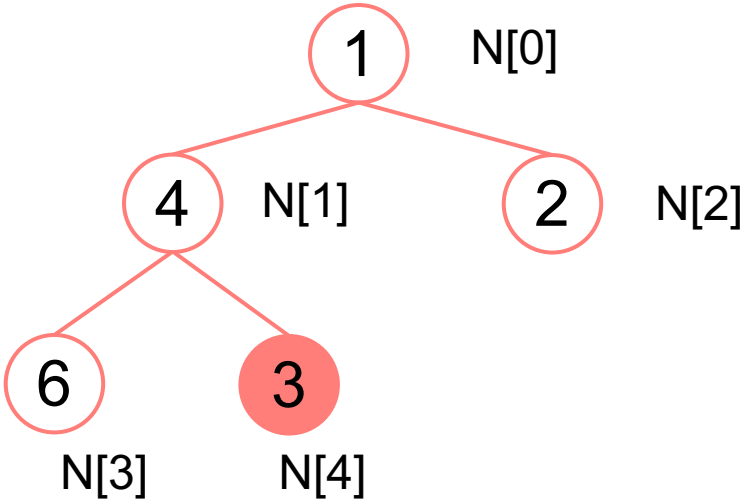

- 木の構造と配列の要素を対応させてデータを格納すると，indexの関係が以下のように与えられる



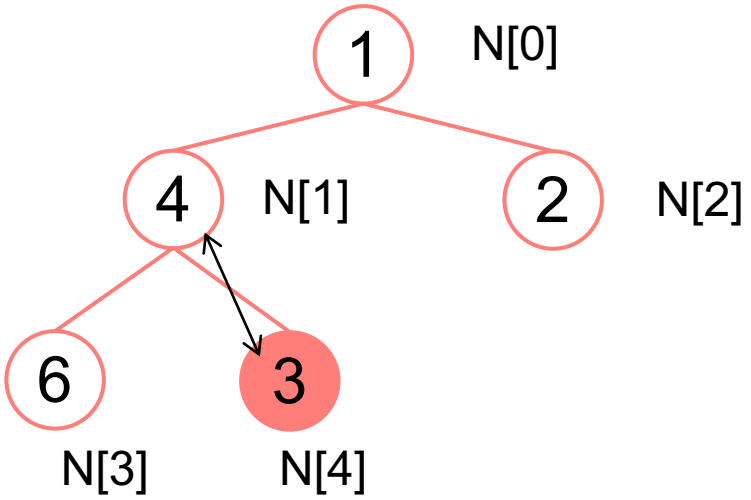
# ヒープの操作-要素の挿入



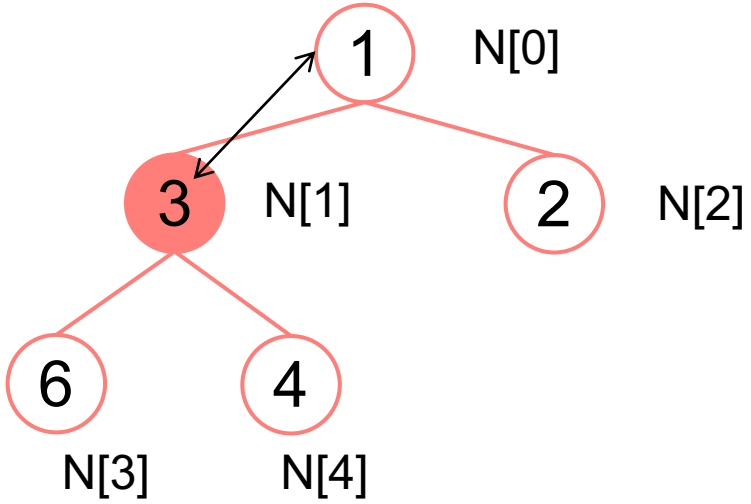
## 要素の挿入




① 木の最後(配列の最後)に追加




② 親と比較して親が大きければ入れ替え

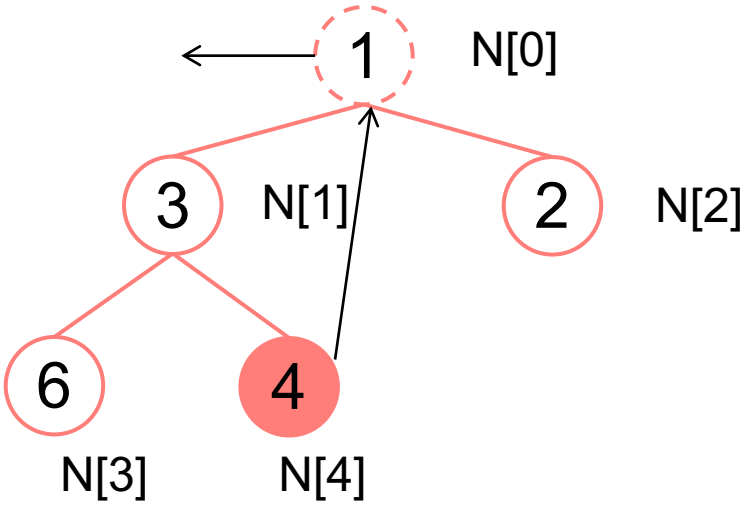



③ 親の方が小さくなったら操作終了

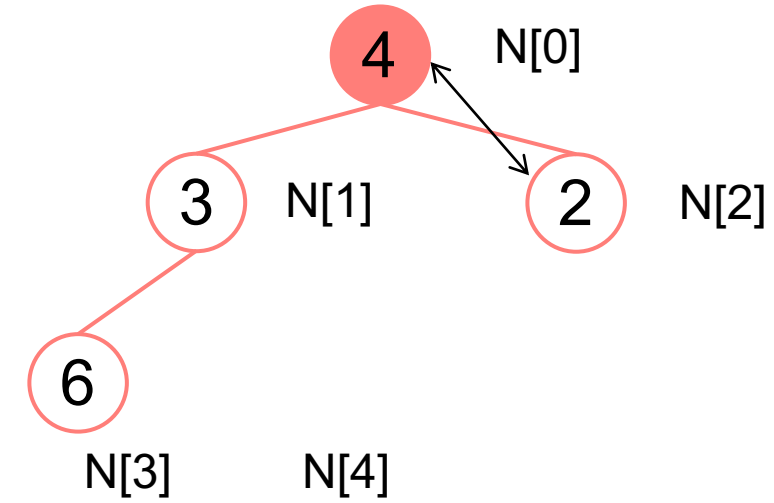
# ヒープの操作-要素の取り出し



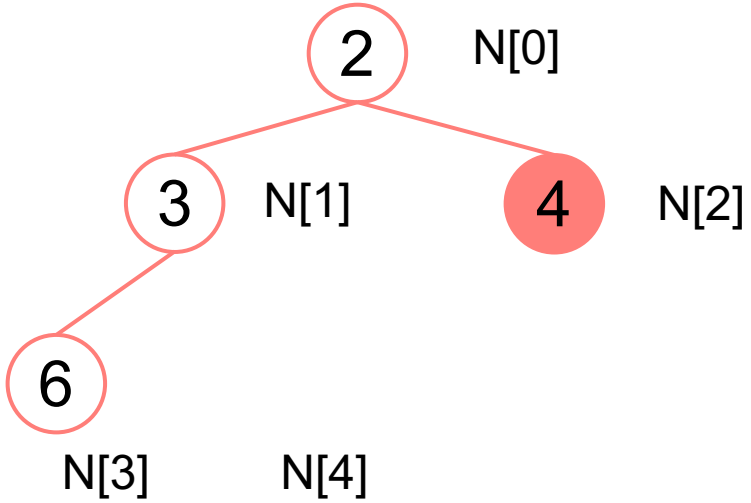
## 要素の取り出し




① 対象の要素を木から取り除き，
最後の要素を先頭に入れる




② 子の方が小さければ入れ替え




③ 親の方が小さくなったら操作終了

# ヒープによる最短経路探索

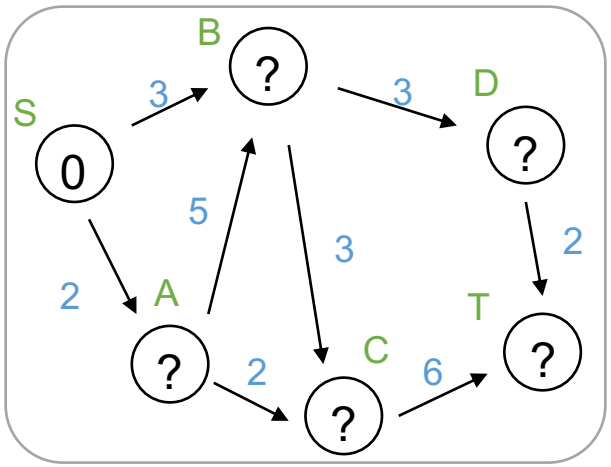


ヒープ：データ構造を採用すると，計算量が$O\big((|E|+|V|)\log|V|\big)$に抑えられる．

- ヒープへの要素の追加，位置の更新，最小値取得+再構成は $O(\log|V|)$ で可能
- 最小値を取り出す操作は$|V|$ 回，位置の更新は合計で最大$|E|$ 回実行

$\rightarrow O\big((|E|+|V|)\log|V|\big)$

Dijkstra法を適用するNW




- リンク$(i,j)$間の距離: $w(i,j)$
- ラベル値の確定したノード集合を $S$
- それ以外のノード集合を $Q$ とする

→ラベル値 $d(y)$ s.t. $y \in Q$ は，
ラベル値の確定したノード $x \in S$ のみ
を経由した場合の 始点 S からの経路長

**Heapを用いたアルゴリズム**

$Q=\phi$ となるまで，AとBを実行

**[A] DeQueue**

$Q$の中から $d(x)$ が最小の頂点を取り出し，集合 $S$ に追加する

**[B] changekey**

x の各隣接頂点について，

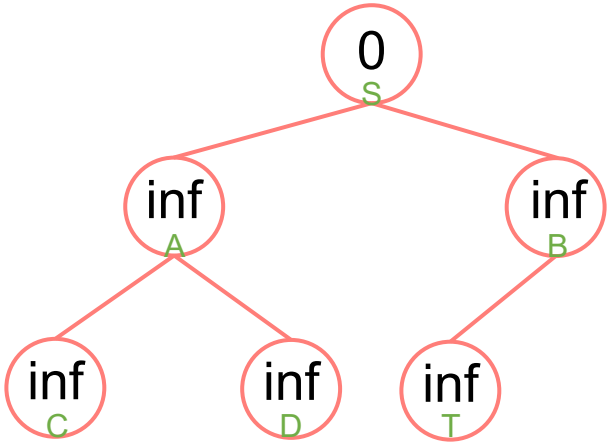
$$d(y) > d(x) + w(x,y)$$

であれば，$d(y) \leftarrow d(x) + w(x,y)$

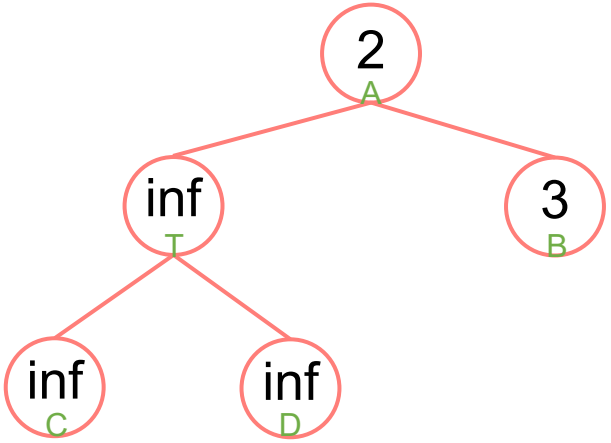
# ヒープによる最短経路探索



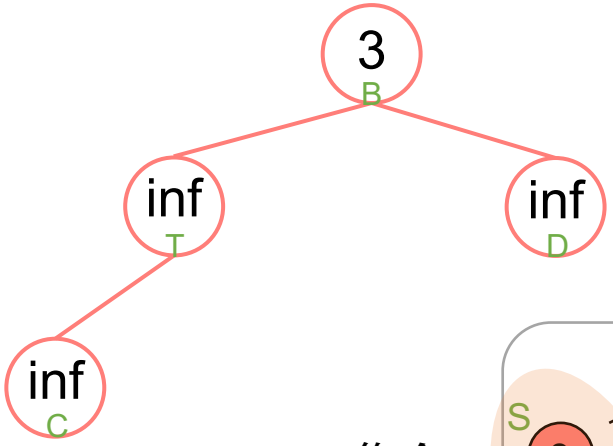
⓪ 一時ラベルでヒープ構築



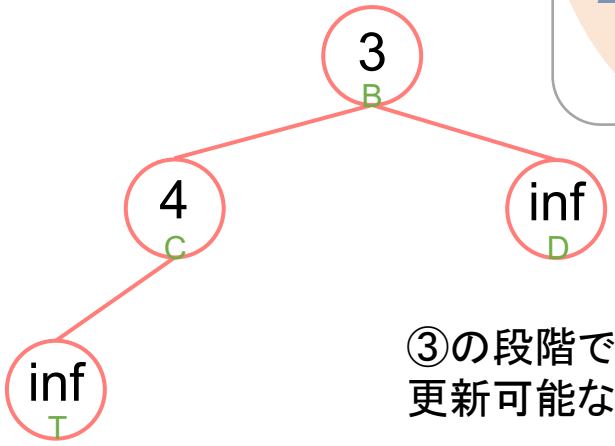

① DeQueue S, changekey



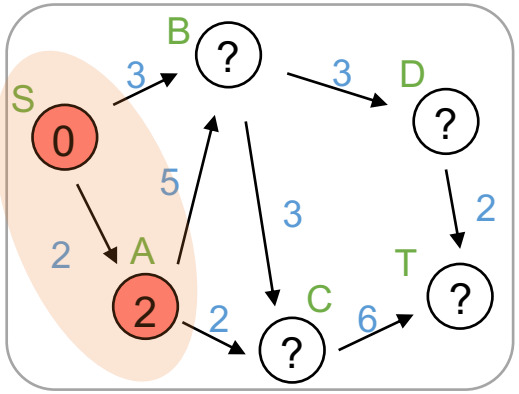

② DeQueue A, changekey




③ changekey




集合 $S$

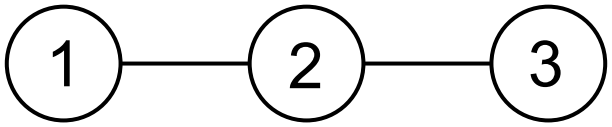



③の段階で集合 $S$ からラベル値を更新可能なのは，ノードBまたはC

# 二分木



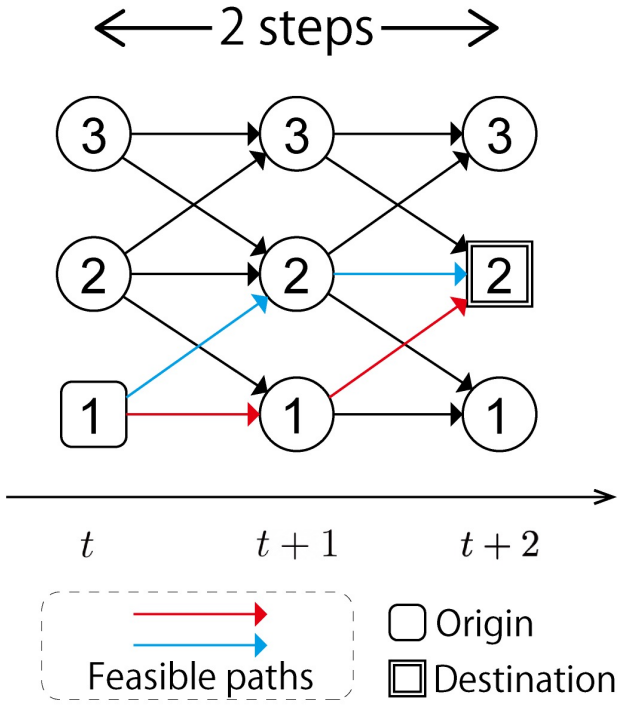
線形都市上の移動を考える



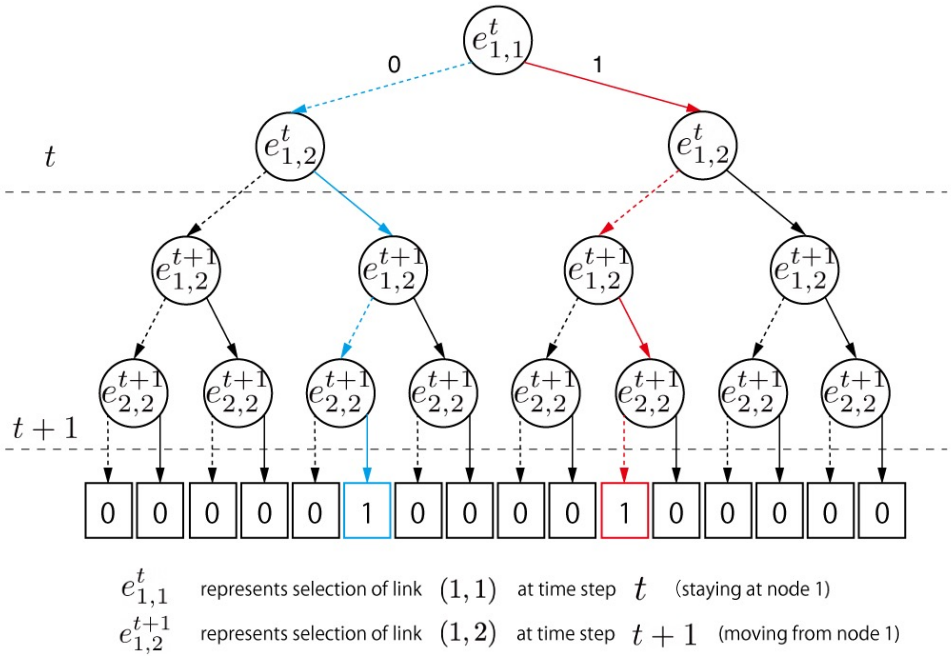

お題:
時刻 $t$ に 都市 1 を出発し，時刻 $t+2$ に
都市 2 に到着する経路を求めよ.

✓ ノード間の移動に加えて，
同一ノードへの滞在を考える

## 時間構造化NWへの拡張




## 場合分け二分木を用いた選択肢表現



# BDD/ZDD



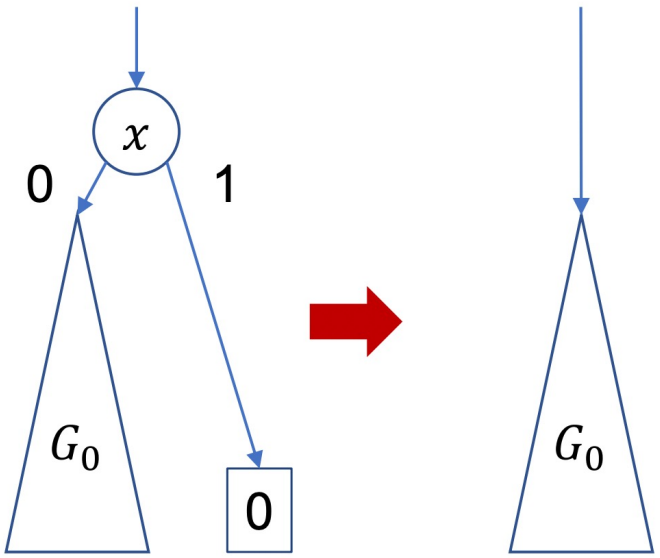
BDD/ZDD: ブール関数の有向グラフ表現であり，大規模な組合せ論理データを操作可能

- BDDでは，分岐節点における 0−枝と 1−枝の行き先が同じであれば削除して読み飛ばすことで，等価な接点の共有+冗長節点の削除を行う
- ZDDでは，1-枝が0-終端節点を直接指している場合にその節点を取り除き 0-枝の行き先に直結させている (BDDとは節点の削除規則が異なる)

## ZDDの場合分け二分木からの圧縮規則

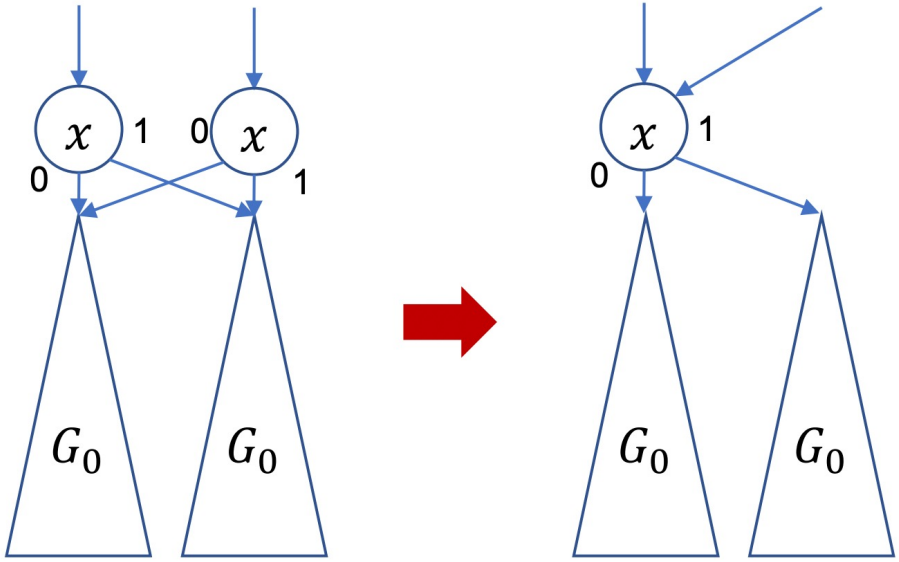
①冗長節点の削除

1-枝が0の値を持つ葉を指している場合に，この節点を取り除き，0-枝の行先に直結




②等価節点の共有

等価な節点（アイテム名が同じで，0-枝同士，1-枝同士の行き先が同じ）を共有

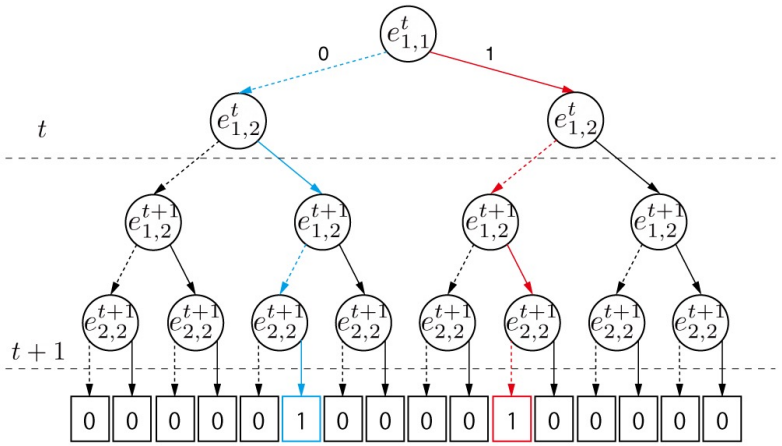


# BDD/ZDD

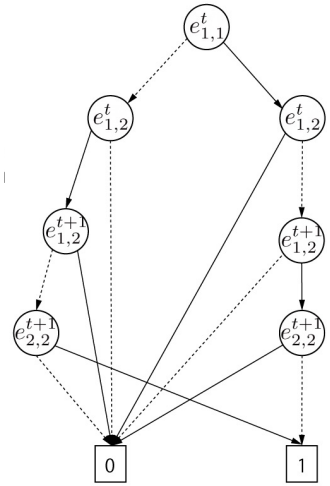


## 場合分け二分木

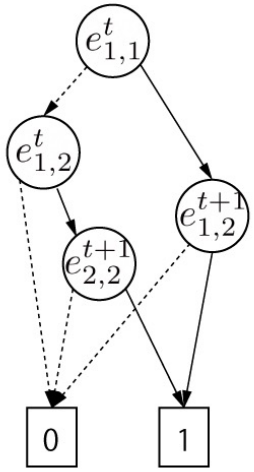
冗長節点の削除＋等価節点の共有でコンパクトに表現




## BDD: 二分決定図



## ZDD:ゼロサプレス型 二分決定図




## まとめ

- 組合せ集合に無関係なアイテム(0-終端節点を指す)に関する節点の削除規則は，**疎な組合せ集合に対して顕著な効果**
- 組合せ集合の各要素に含まれるアイテムの平均出現頻度が 1% であれば，ZDD は BDD よりも 100 倍コンパクトになる可能性があるとされている．
- BDDやZDDは，それ同士で**集合演算が可能**であるため，複数エージェントの問題にも発展させることが容易．

# 参考書籍

Narasimha Karumanchi 著, 黒川 利明, 木下 哲也 訳 (2013)
入門 データ構造とアルゴリズム, オライリー・ジャパン

湊真一 (2015), 超高速グラフ列挙アルゴリズム, 森北出版,

K.メールホルン, P.サンダース 著, 浅野哲夫 訳 (2009)
アルゴリズムとデータ構造, シュプリンガー・ジャパン