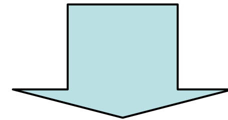
第5章 データ構造

概要

- 1～4章でならったことより類推されること
  - 情報を保持したり移動させたりするにもコストはかかる
  - 効果的に情報を保持したり移動させたりするには？



- 適切なデータ構造を使う
  - 過去の様々な試みの集大成
  - 次章で説明するアルゴリズムの適用には、適切なデータ構造が必要なことも

節概要

- 基本的なデータ構造
  - 配列
  - リスト
  - キュー
  - スタック
- ハッシュ
- 2分木

5.1 基本的なデータ構造
(1) 配列

- 数値のみを並べたデータ構造
- 要素は添え字でアクセス
  - プログラミング言語が主記憶アドレスに変換
- 1次元配列
  - 添え字の数は1つ
  - 注: 添え字の最小値は0とすることがほとんど
    - nエントリの配列の添え字は0からn-1になる
    - 符号無し2進数で効率よく表現するため
    - プログラミングに慣れると0から数える習慣がつきます(嘘)

1次元配列

a[0]
a[1]
a[2]
...
a[n-1]

(1) 配列

- 2次元配列
  - 添え字が2つある配列
  - 主記憶上の実体は1次元の並び
    - プログラミング言語によっては、1次元配列としてもアクセス可能
- 添え字の数を増やすことで、n次元配列も可能

2次元配列の実体

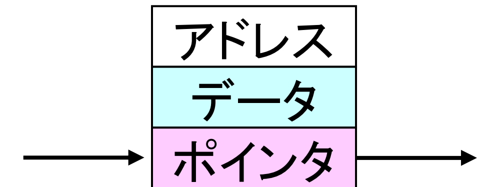
a[0,1]
a[1,1]
a[2,1]
...
a[m-1,0]
a[0,1]
a[1,1]
a[2,1]
...
a[m-1,1]
a[2,0]
a[2,1]
...

2次元配列

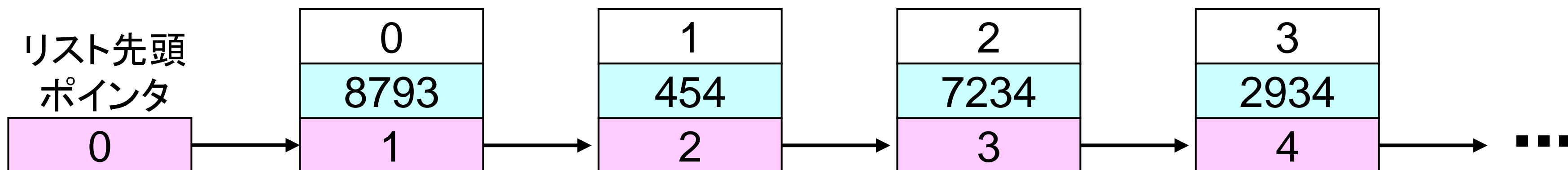
a[0,0]	a[1,0]	...	a[m-1,0]
a[0,1]	a[1,1]	...	a[m-1,1]
a[0,2]	a[0,2]	...	a[m-1,2]
...	...	...	...
a[0,n-1]	a[0,n-1]	...	a[m-1,n-1]

(2) リスト

- データのつながりに意味を持たせたデータ構造
- 2つの情報を追加
  - ポインタ: 次のデータの位置(アドレス)を示す
  - アドレス: データの位置
    - 通常は主記憶アドレスや添え字を使う



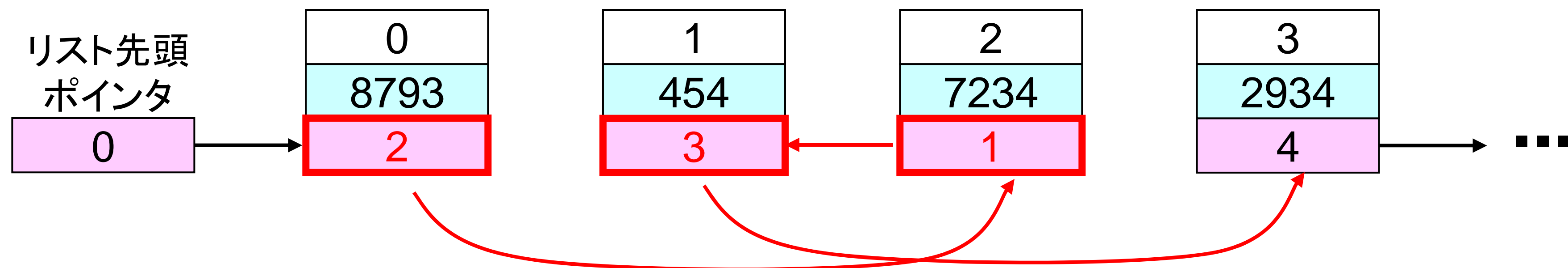
- ポインタでデータを順次たどることができる
  - 下の図では8793, 454, 7234, 2934の順にデータは並んでいる
  - 先頭についてはポインタのみとかデータ空のエントリとかを利用



ポインタ操作によるリストの並べ替え

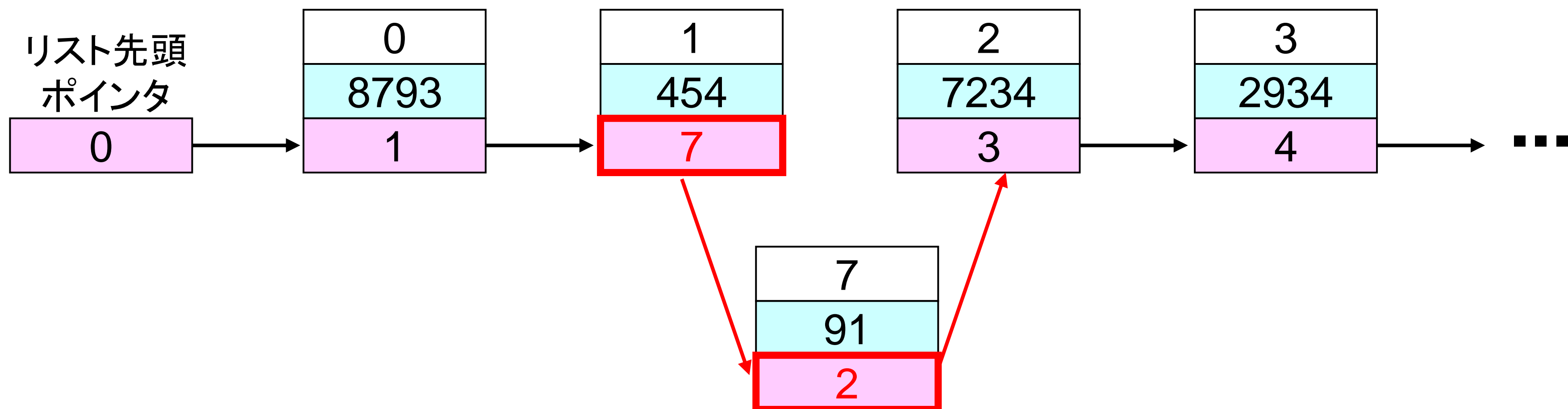
- ポインタの数値を変更することで並び替えが可能
  - 下の図では、データは8793, 7234, 454, 2934の順に並んでいる
  - リスト先頭ポインタを変更することにより、先頭のデータも変更可能

->リストにおけるデータの順序は、配列における物理的位置ではなく、論理的に決まる



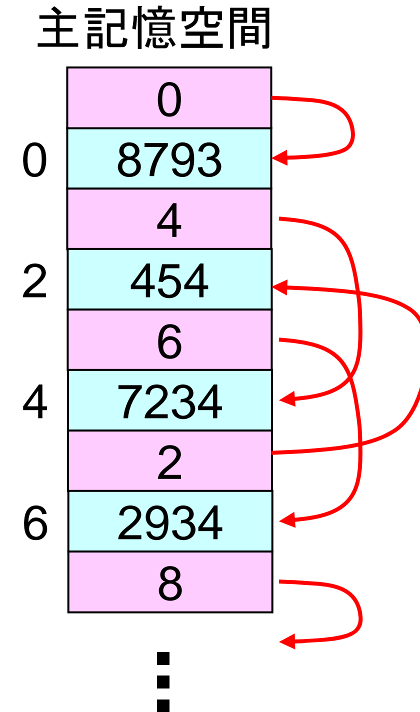
ポインタ操作による
データの追加／削除

- ポインタの数値を変更することで並び替えが可能
  - 下の図では、データは8793, 454, 91, 7234, 2934の順に並んでいる
  - リスト先頭ポインタの変更により、先頭にデータを追記することも可能
- ポインタを2つ後のアドレスに設定することで、データの削除が可能
  - アクセスできないデータは削除されたも同然
  - 削除されたエントリは再利用のため、再利用リストに追加したりする



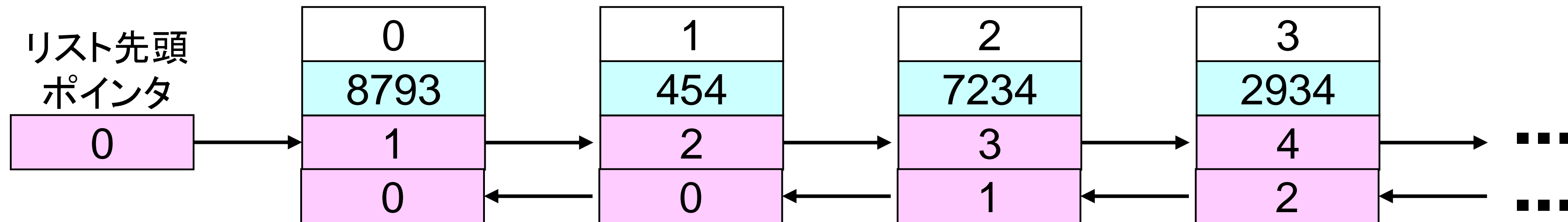
主記憶空間でのリストの実装

- データのポインタの2つのエントリをペアで作成する
- アドレスは主記憶アドレスを利用
- 構造体を使えるプログラミング言語は構造体で実現

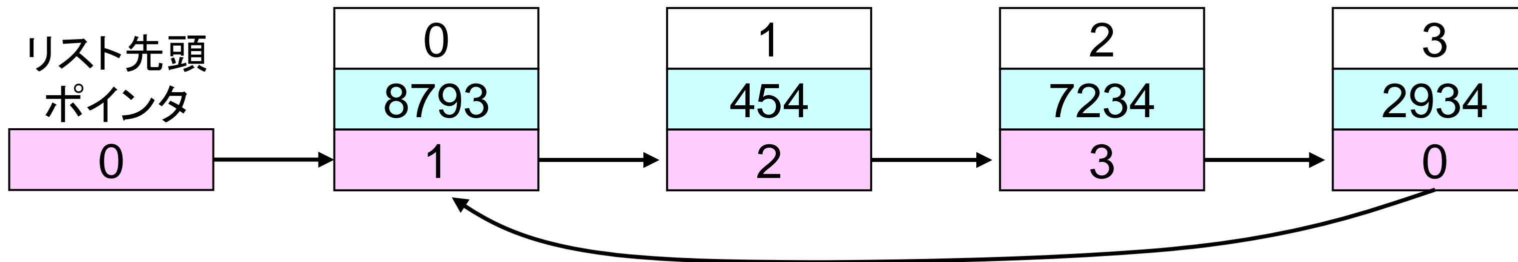


リストの派生

- 双方向リスト
  - 前のデータのポインタと次のデータのポインタを持つ

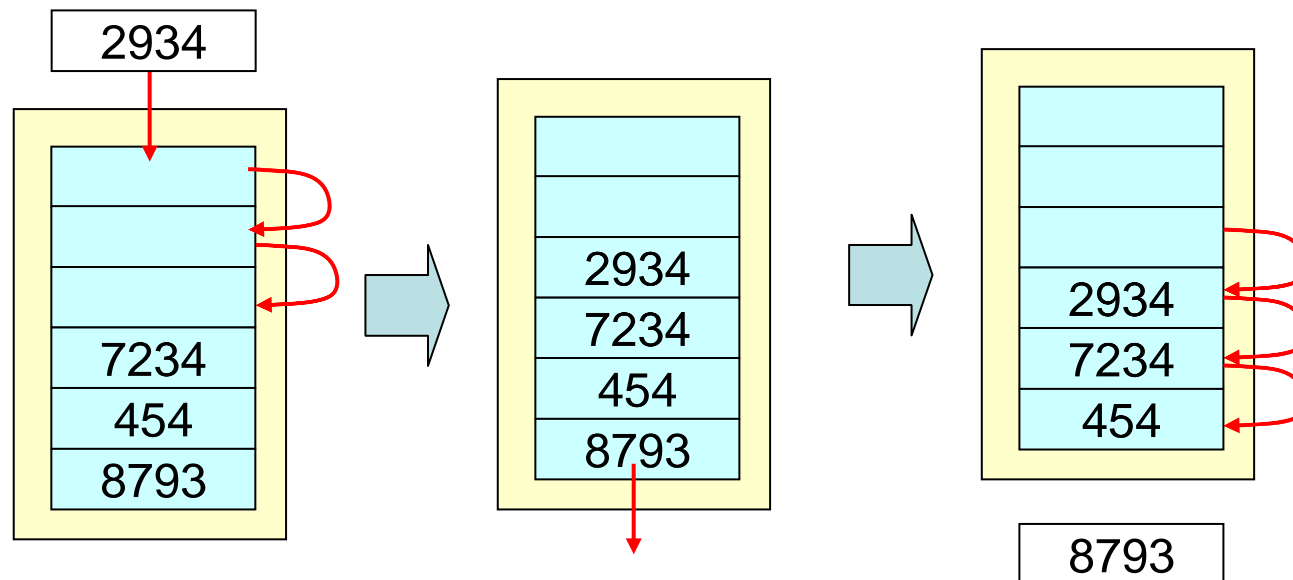


- 環状リスト
  - 最後のデータの次のポインタを先頭データとする



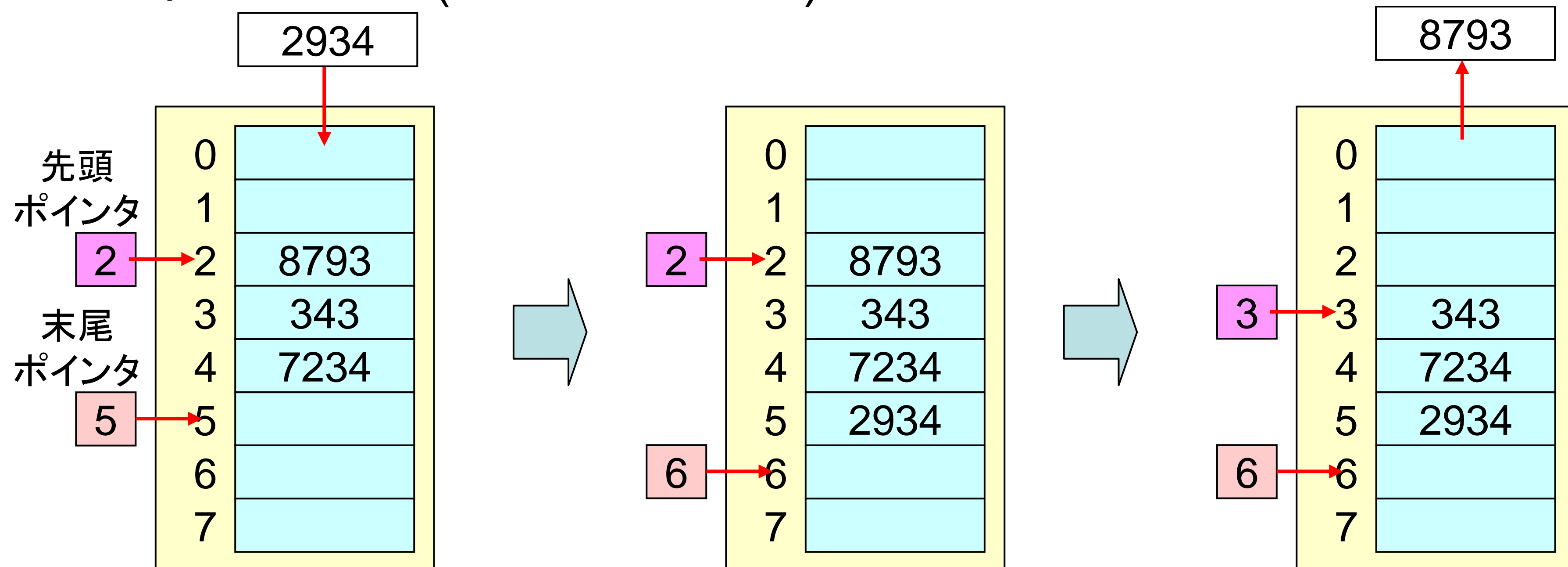
(3) キュー

- 配列のようなデータの並びがあり、データの格納と取り出しの順番に着目したデータ構造
- 配列の一方の端でデータが挿入され、他方の端でデータが取り出される
  - 先入れ先出し(FIFO: First In First Out)
  - 人の行列を考えれば分かりやすい



主記憶上でのキューの実現

- 配列にデータを格納
- データを読み出す先頭ポインタとデータを書き込む末尾ポインタを付加
  - ポインタは読み書き後に(ポインタ+1)%エントリ数
- 環状バッファ(circuilar buffer)とも呼ばれる

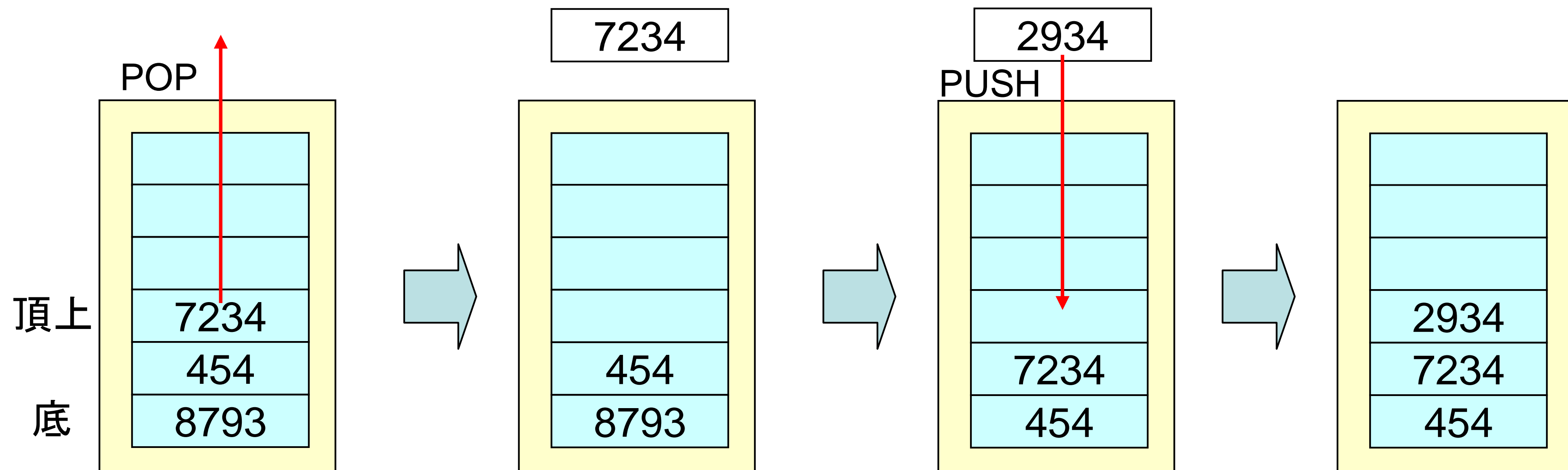


(4) スタック

- 配列のようなデータの並びがあり、データの格納と取り出しの順番に着目したデータ構造
- 配列の一方の端でデータが挿入され、他方の端でデータが取り出される
  - 先入れ後出し(FILO: First In Last Out)
  - 積み上げた未読本を上から順番に読む感じ

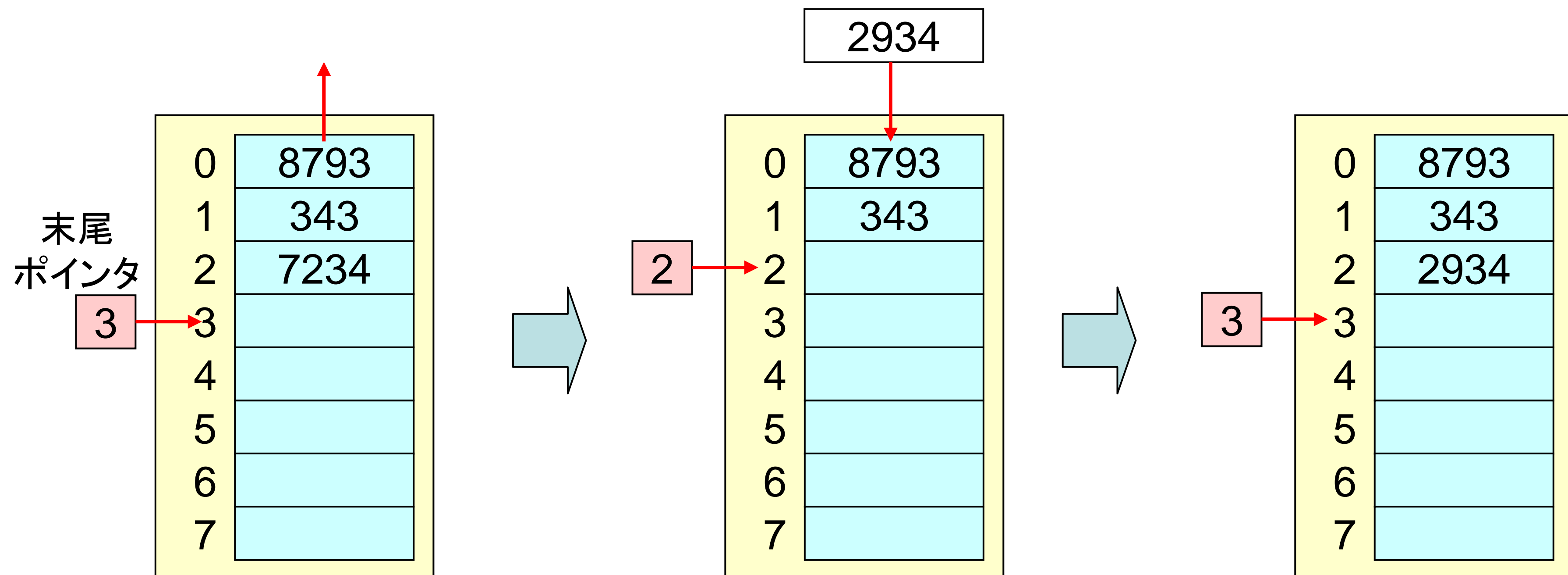
スタックの構造と用語

- 読み出しはPOPと呼ぶ
- 書き込みはPUSHと呼ぶ
- 配列の先頭を底と呼ぶ
- 配列の末尾を頂上と呼ぶ



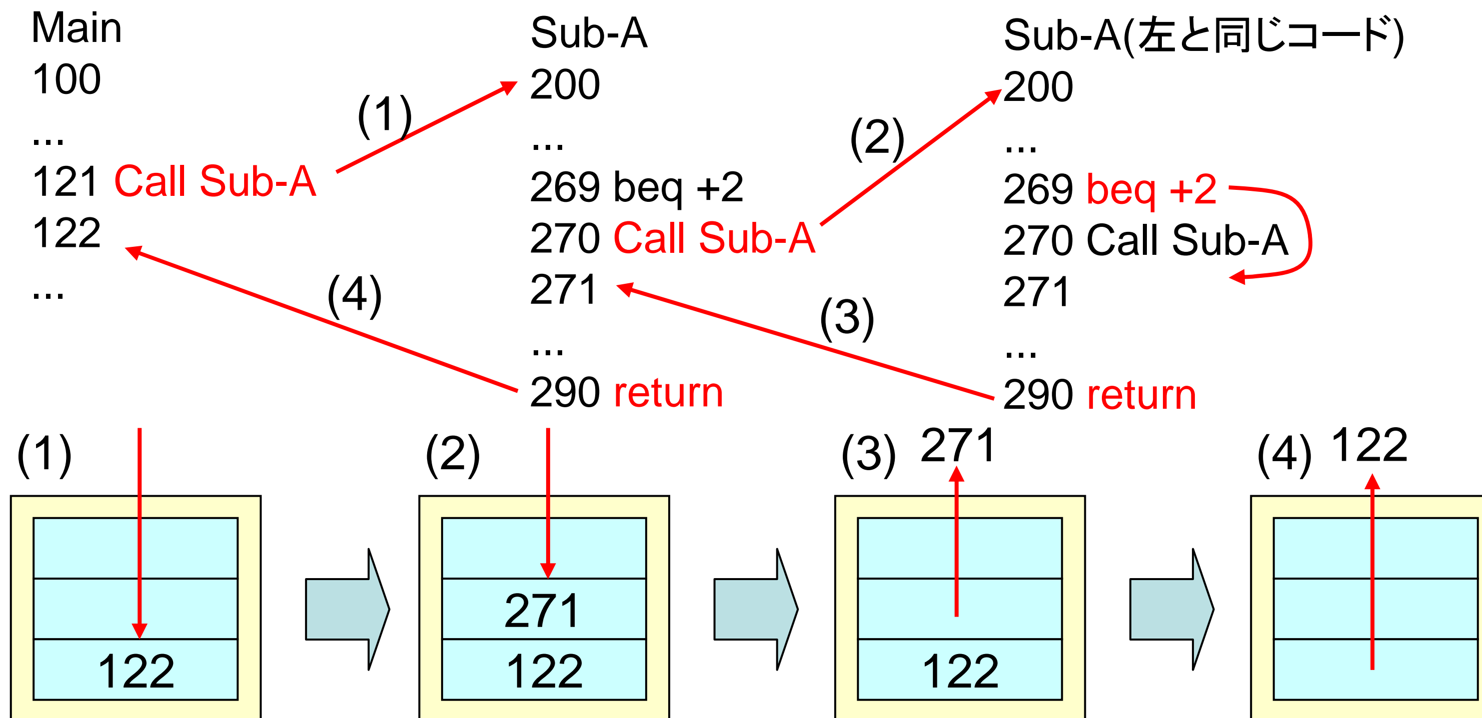
配列によるスタックの実現

- キューとは異なり、ポインタ1つで実現
  - ポインタが値域を越えていないかチェックが必要
  - ポインタの範囲をチェックするのはキューでも必要



関数の再帰呼び出し時のスタックの利用

- 関数はどこから呼ばれるか分からない
  ->戻り先アドレスはどこかに保存する必要がある
- 関数の再帰呼び出しなどに備え、多数の戻り先も保存
- 関数呼び出し時や割り込み時のレジスタ退避にも利用



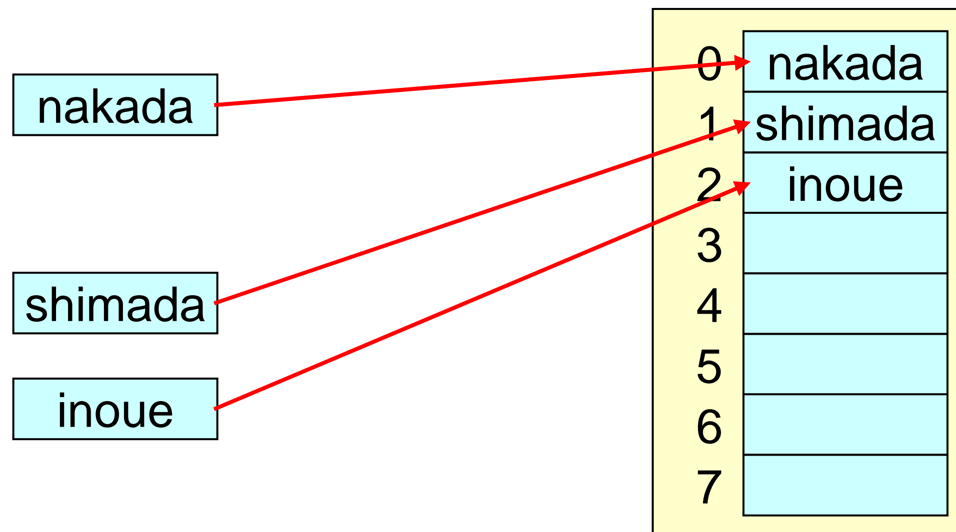
演習

- 以下の作業後のキューの状態は？
- 以下の作業後のスタックの状態は?
  - “246”を書く
  - “135”を書く
  - 読み出す
  - “680”を書く
  - “579”を書く
  - 読み出す
  - “150”を書く

5.2 ハッシュ

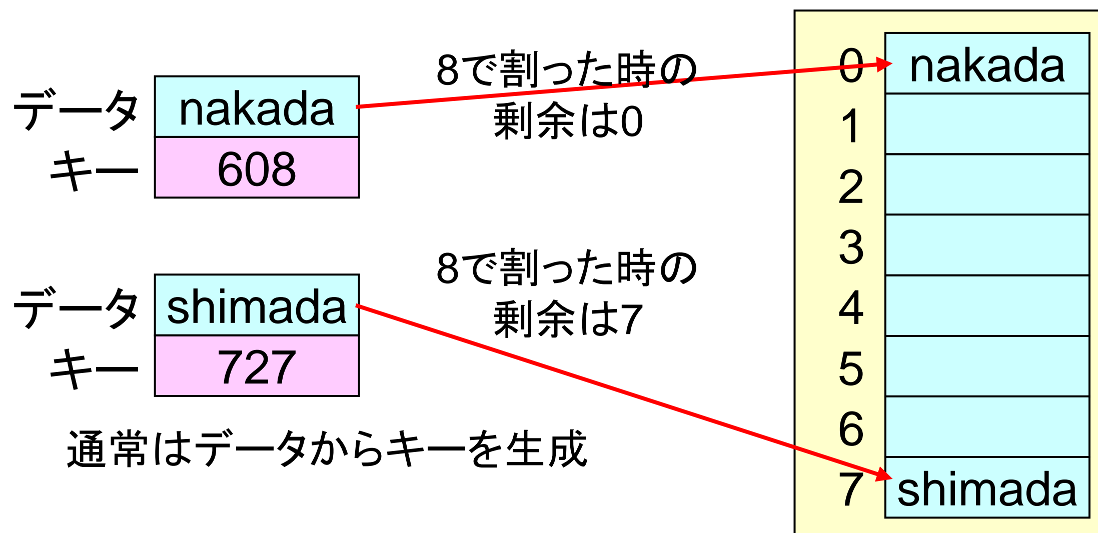
- 配列等に適当にデータを格納することを考える
- 読み出す時のことを考え、どのように格納するか？
  - 順番に読み出して探すのは効率が悪い
  - 容量効率も考えたい(できるだけ詰め込みたい)

->ハッシュというデータ構造



5.2 ハッシュ

- データに対してキーの値を作成
  – 例: shimada = 115+104+105+109+97+100+97 = 727
- キーの値と配列(ハッシュ表)のエントリ数からハッシュ値を作成
  – ハッシュ値の作成方法をハッシュ関数と呼ぶ
    - 簡単な物では剰余(下の例)、複雑な物ではMD5
    - 後述するように、値ができるだけかぶらない方が嬉しい

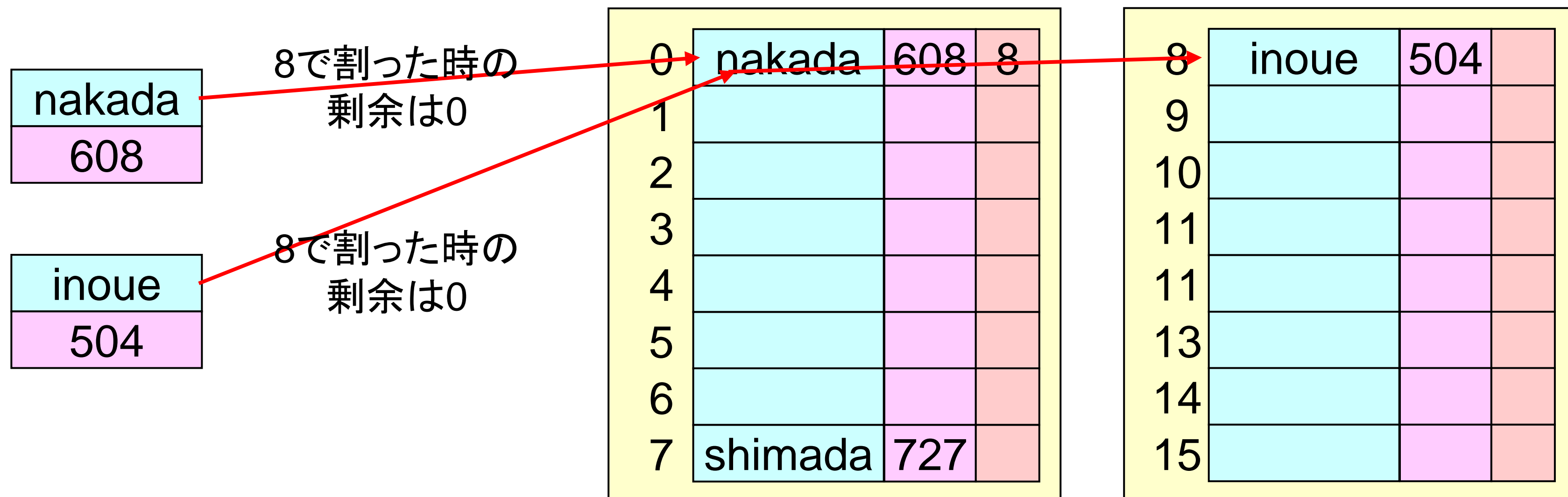


オープンアドレス法

- ハッシュ値がかぶってしまったら？
  ->別のエントリに書き込む
  - キーの値もエントリに格納して、区別できるようにする必要がある
- オープンアドレス法ではハッシュ値を+1していく
  - 空いているエントリがある所まで+1を続ける

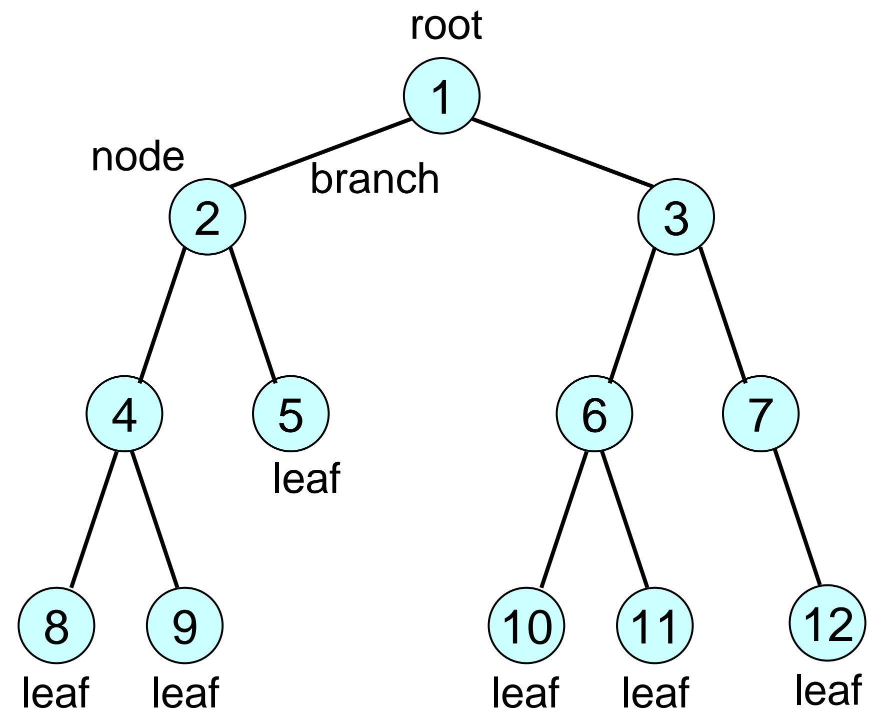
チェイン法

- チェイン法では、ポインタで別エントリを示す
  - かぶった場合は、ポインタの先に格納
  - ポインタの先は同一の配列でも別の配列でもかまわない
- ポインタを利用して後に登録したものを素早く参照可能
  - ハッシュ値はまずポインタ表を引いて、対応するエントリを見に行く
  - 後に登録した物をポインタ表に登録することで、最初に参照可能



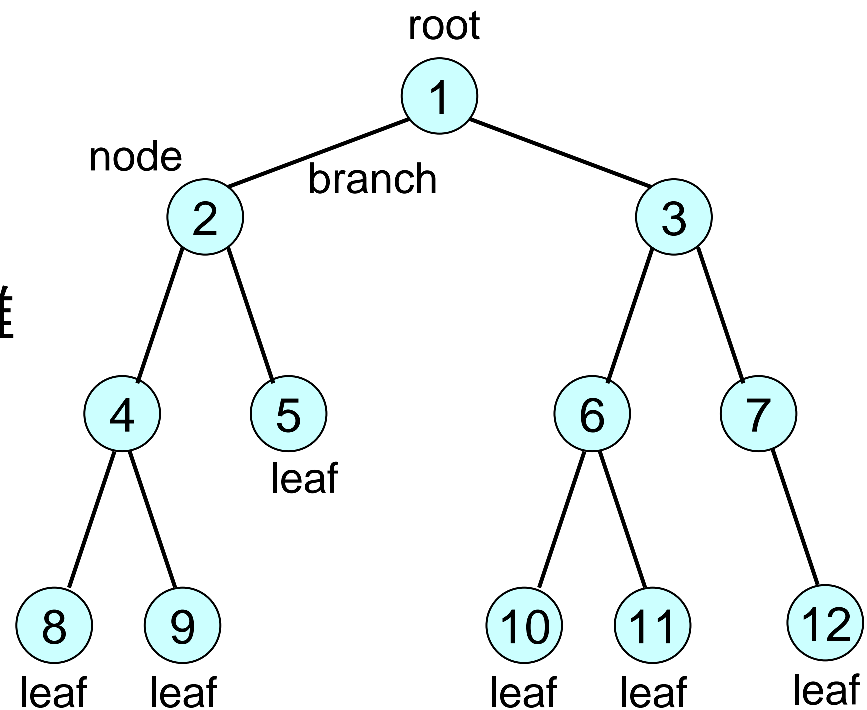
5.3 木構造

- データの階層的な関係を表す
- 用語
  - 節(node): 木構造においてデータが格納される部分
  - 枝(branch): 節を結ぶ線
  - 根(root): 最上位の節
  - 葉(leaf): 自分の下に節を持たない節



5.3 木構造

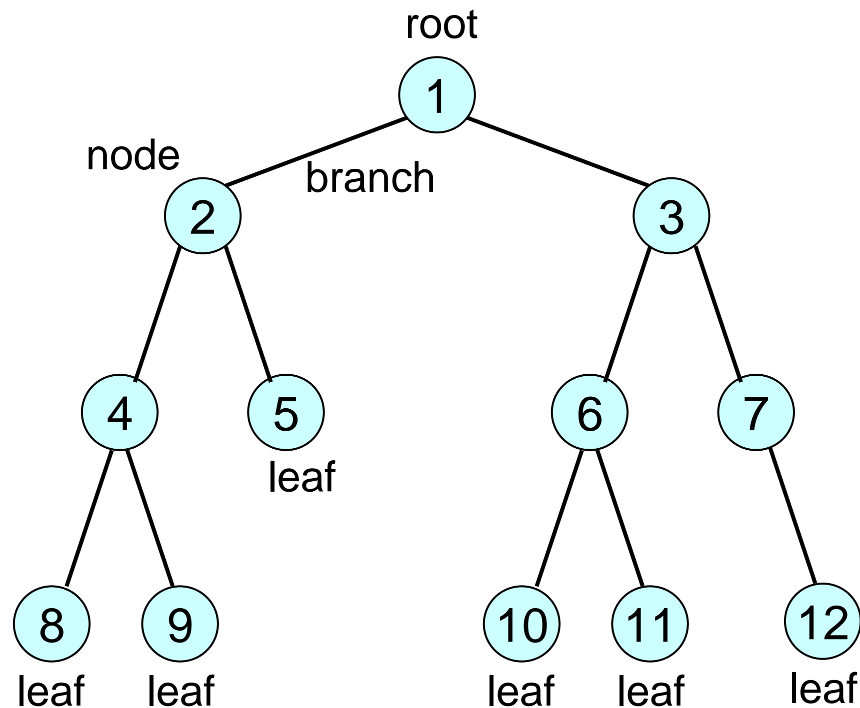
- 用語(続き)
  - 部分木: 木構造のある節以下を、新たな木構造として考えたもの
  - 深さ: 根から葉までの距離
- 2分木がよく用いられる
  - 1つの節に0-2個の子の節を持つ木構造



木構造の主記憶上の構造

- 双方向リストの拡張で実現可能
  - 必要に応じて、親接点へのポインタを減らしたり

アドレス
データ
親へのポインタ
子1へのポインタ
子2へのポインタ



1
データ
なし
2
3

2
データ
1
4
5

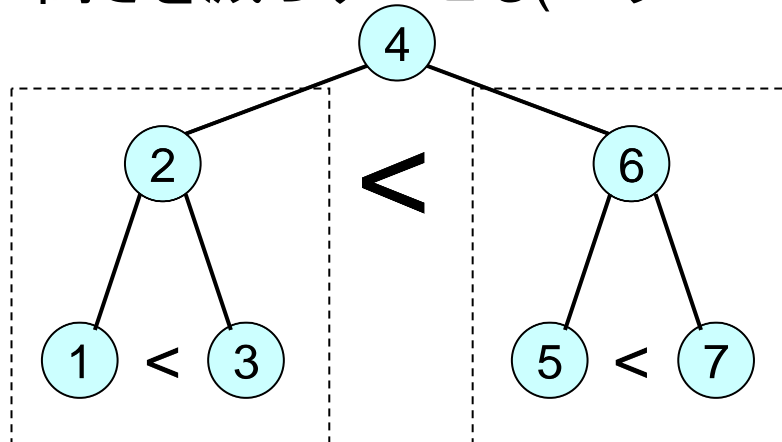
3
データ
1
6
7

(1) 2分探索木

- 以下のルールに従ってデータを配置することにより探索効率を上げる方法

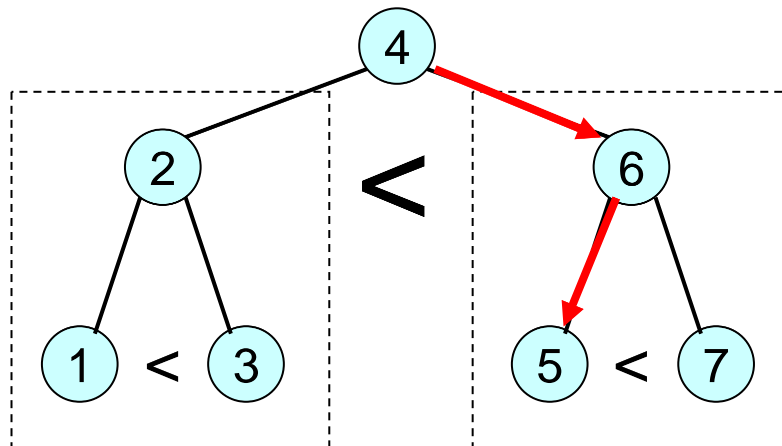
任意の根とその部分木について ルール1: 左部分木に含まれる値は根の値より小さい ルール2: 右部分木に含まれる要素は根の値よりも大きい

  - さらに効率を上げるため、左右の木をバランスさせて木の高さを減らすことも(バランス木)



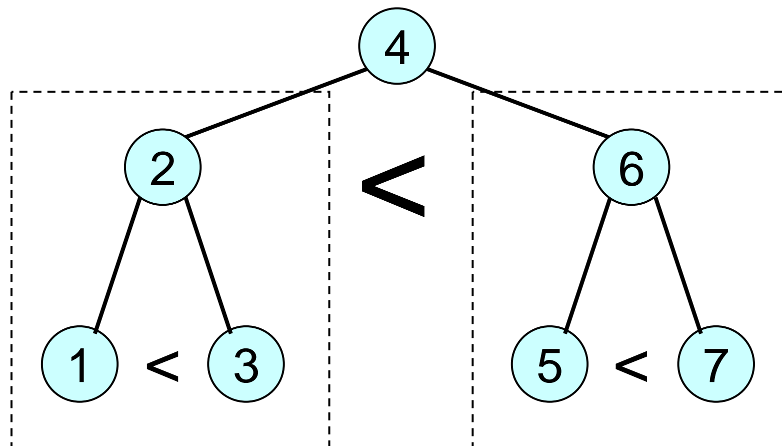
2分探索木上での探索

- 探索したい値が節の値より大きいとき右、小さいとき左の枝をたどれば目的の値に到達する
- 例: 探したい値が5
  - 4 < 5なので右に行く
  - 5 < 6なので左へ行く
  - 5に到達



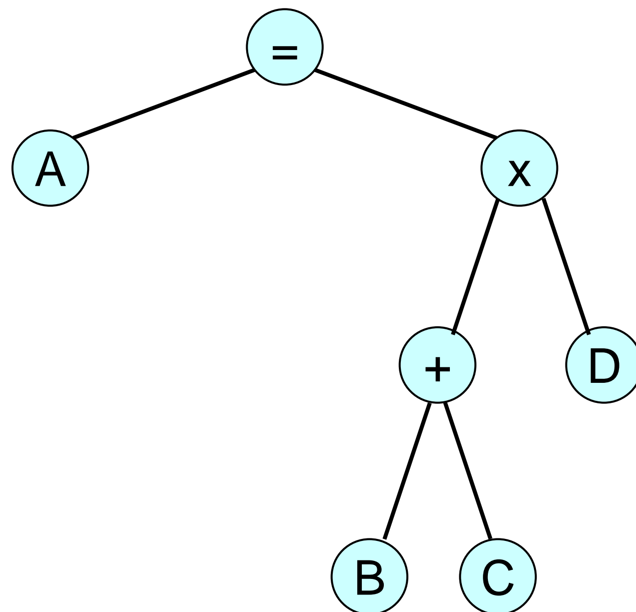
(2) 2分木の走査

- 2分木の走査法には3種類がある
  - 前順：節点、左部分木、右部分木の順に走査
  - 間順：左部分木、節点、右部分木の順に走査
  - 後順：左部分木、右部分木、節点の順に走査
- コンパイラ作成時の構文解析の最適化などで関係してくる



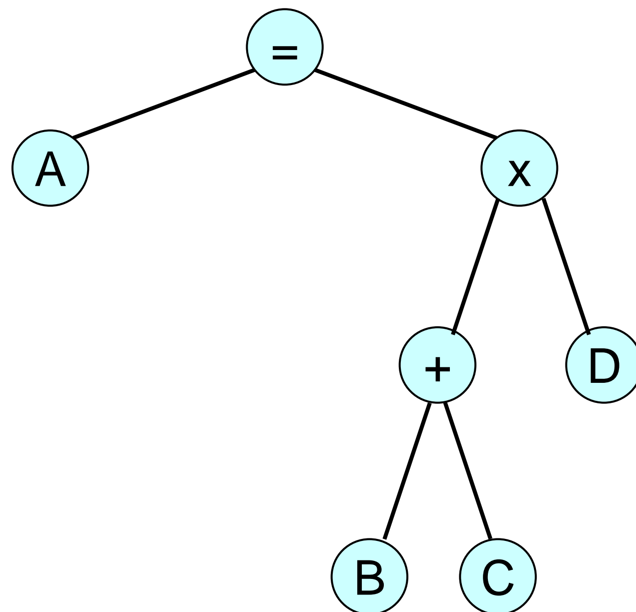
(3) 数式と木構造

- 節に四則演算を記入し、葉に変数を記入することにより、四則演算を木で表現できる



(3) 数式と木構造

- 前順走査結果:　=A×＋BCD　(ポーランド記法)
- 間順走査結果:　A=(B+C)×D
- 後順走査結果:　ABC+D×＝　(逆ポーランド記法)



その他のデータ構造に関する よもやま話

- 他によく使われるデータ構造にグラフ構造がある
  - 上下関係の無い木構造と考えてよい
  - 有向グラフなどの派生がある
- 無理やり規定されたデータ構造に合わせることもある
  - 少し前のGPUコンピューティングは、いかに3Dグラフィックのデータ構造に処理したいデータを合わせるか...
  - SIMD(Single Instruction Multiple Data)命令を利用するために、SIMDに適したデータ構造を考える...

5章のまとめ

- データを効率良く操作／格納するための基本的なデータ構造がある
- うまくデータ構造を作らないと、アルゴリズムの適用が難しくなる
- どのデータ構造も典型的な主記憶上での実現方法がある
  - ポインタをうまく使う