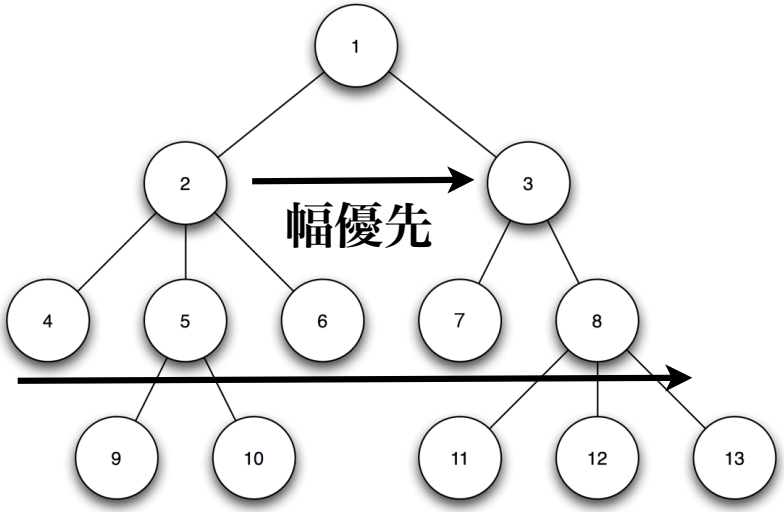
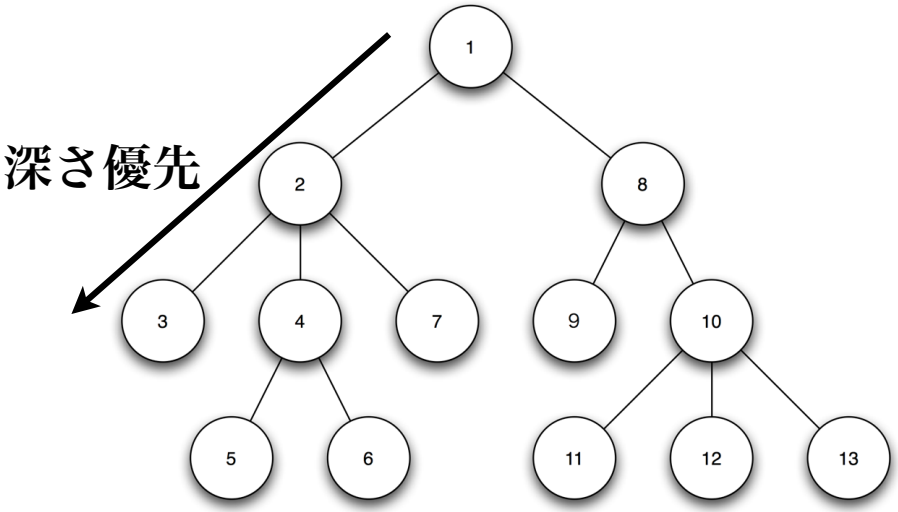
基本データ構造：木構造のたどり方

- 木構造データの各ノードをもれなく探索する方法
  - 深さ優先探索（depth first search)
  - 幅優先探索 (breadth first search)





- ラベルの出力順
  - pre-order（前順）　：最初の訪問
  - in-order　（中順）　：2度目の訪問
  - post-order　（後順）：全ての子供を回った後

数式の木表現

- (A + B × C) ÷ E - (F × G + H) = x

1. = を根とする
2. 優先順位の低い演算子で式を分割

   (A+B×C)÷E
   (F×G+H)
3. 各式を同様に分割

   A + B×C
   E
4. 変数単独になるまで繰り返す

   A
   B×C→B, C

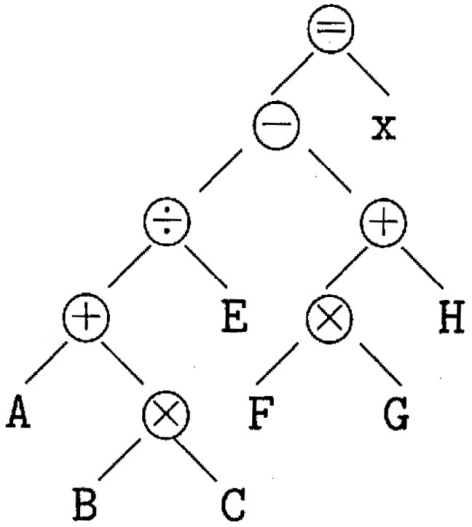


図 6.13 数式の木表現

問題：この数式の木表現を深さ優先＋後順でたどってみよう！

深さ優先＋後順：A B C × + E ÷ F G × H + − x = 逆ポーランド表記

ちなみに深さ優先＋中順：A + B × C ÷ E − F × G + H = x

逆ポーランド表記

- 前から逐次処理で読むことができる括弧の処理が不要にした表現

逆ポーランド記法：A B C × + E ÷ F G × H + − x =

- 演算子が要求する演算対象の個数が決まっているので、順番をうまく書く事で括弧を不要としている
- 日本語と同じように読むことが出来る記法でもある
  - AにBとCを掛けた結果を加えてEで割って、FとGを掛けてHを加えた結果を引いて、xに格納する（＝）
- 逆ポーランド記法の求め方は木表現を介さない方法もある（162ページに中程）

ちなみに深さ優先＋先順：= − ÷ + A × B C E + × F G H x : ポーランド記法

数式の機械語への変換

数式を機械語にする
人がやるのはたやすい

これをどうやって計算機にやらせるか？

- (A + B × C) ÷ E - (F × G + H) = x

1. B×C
2. Aを加える
3. Eで割る
4. 結果を覚える(tmp1)
5. F×G
6. ＋H
7. 結果を覚える(tmp2)
8. 前の結果を思い出す(tmp1)
9. tmp2を引く
10. 結果をxに格納

1. LOAD B, MULT C
2. ADD A
3. DIV E
4. STO tmp1
5. LOAD F, MULT G
6. ADD H
7. STO tmp2
8. LOAD tmp1
9. SUB tmp2
10. STO x

数式の機械語への変換

1. 逆ポーランド記法による表現を求める
2. スタックを用いて命令語を生成する
3. 無駄な命令を削除する

2. 命令語の生成（命令語としてはADD, SUB, MULT, DIV, STOを想定）

(1) 逆ポーランド記法の数式を順に読む

(2) 変数はスタックにpushして(1)へ、演算子なら(3)

(3) スタックから変数を２つ取り出す（pop → x1, pop→x2)

(4) LOAD x2, [演算命令] x1を出力

(5) [演算命令]が[=]なら終了、それ以外なら結果を変数に入れて(1)に戻る

黒板で順にやる

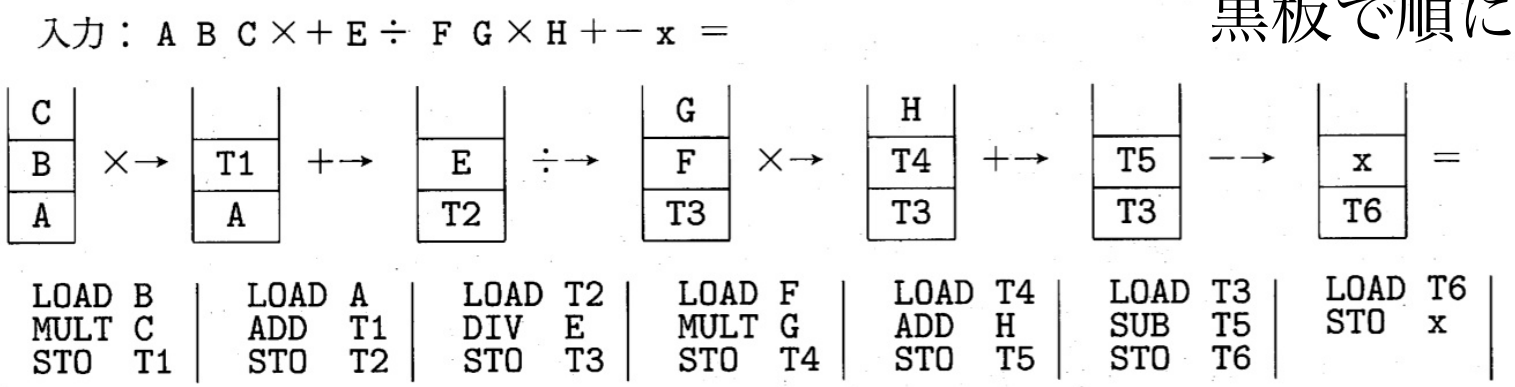


図 6.11　スタックの変化と出力の様子

数式の機械語への変換（続き）

- 先ほどの命令群を並べればＯＫ
- 無駄な計算の排除
  - STO tmp → LOAD tmp (無駄１：覚えてすぐに思い出す)
  - STO tmp → LOAD x2 → ADD tmp (無駄２：ADD x2と同じ)
  - STO tmp → LOAD x2 → MULT tmp (無駄２’：MULT x2と同じ)
    - 注意：ADD, MULTは可換なのでＯＫ、SUB, DIVはNG
- 無駄を排除すると

```
1. LOAD B, MULT C
2. ADD A
3.  DIV E
4. STO tmp1
5.  LOAD F, MULT G
6.  ADD H
7. STO tmp2
8. LOAD tmp1
9. SUB tmp2
10. STO x
```

コンパイラ

- 高級言語→[コンパイラ]→機械語命令列→[アセンブラ]→機械語
- 高級言語
  - C言語
  - Java
  - Standard MLなど多数
- 機械語はCPUによって異なる
- 機械語命令列も理解は簡単ではない（単にややこしいだけ）
- コンパイラの段階で間違い（バグ）を探せる言語が良い言語
  - 実際にはライブラリが充実しているとか、実行速度が速い言語が好まれる
  - プログラムの開発と保守も実行時間の一種なのだけど。。。

アルゴリズム

- プログラム＝データ＋手順（アルゴリズム）
- 良いアルゴリズムとは？
  - 分かりやすく、デバッグの楽なプログラム
    - プログラミングスタイルに関わる話。ここでは扱わない
  - 効率が良いプログラム
    - メモリの消費が少ない
    - 実行速度が速い
- プログラムの実行時間を決める要素
  - コンパイラの性能
  - 機械語命令の性質と速さ
  - **入力データ**
  - **アルゴリズムの計算量**

実行時間

- 入力データの大きさ
  - int, floatなどデータ型による差はここでは考えない
  - 要素の個数n（リストの長さ）を大きさの尺度として用いる
    - 例：2,1,3,1,5,8 （長さ：６）

      1.1, 2.3, 0.8, 0.2 （長さ：４）
- 実行時間は必ずしも入力データのサイズだけで決まらない
  - 例：[1,2,3, 4, 5]と[1,5,2,4,3]を並び替える場合など
- アルゴリズムでは最悪の場合の実行時間 $T(n)$ を考える
  - 平均の方が公平に思えるかもしれないが、すべての入力が等確率などの怪しい仮定が入るので良くない
- CPUやコンパイラの性能に左右されないアルゴリズムの善し悪しを評価するためにサイズに対するオーダーのみを議論する
  - 例：このアルゴリズムの（最悪の）実行時間は n^2 に比例する

ビッグオーとビッグオメガ

- オーダーの表記として

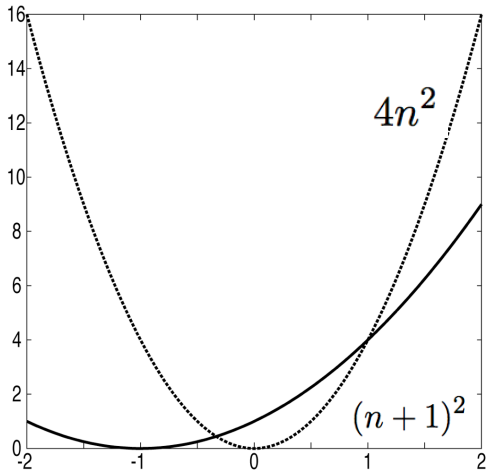
  「あるアルゴリズムの実行時間が $O(n^2)$ である」

  と書いて、「n^2 のオーダーである」と読む。意味は

  「正の定数 c, n_0 が存在して、n_0 以上の n に対しては $T(n) \leq cn^2$ となる」

  である（要は、ある程度大きなサイズの入力での上限に注目する

  例：$T(n) = (n+1)^2$ のアルゴリズムは、$n \geq 1$ の時に
  $T(n) \leq 4n^2$ なので、$O(n^2)$ である



- 同様に下限を考える際には $\Omega(f(n))$ を使い

  「実行時間は $f(n)$ のオメガである」と言う

  正確には

「$T(n) \geq cf(n)$ が無限回なり立つような定数 c が存在する」

  を意味する

増加率の影響

- オーダーの議論はアルゴリズムの善し悪しの目安に過ぎない
- それでも効率の良い（オーダーの低い）アルゴリズムは重要
  - 年々データは増えている！
  - 計算機が速くなったときのゲインが大きい

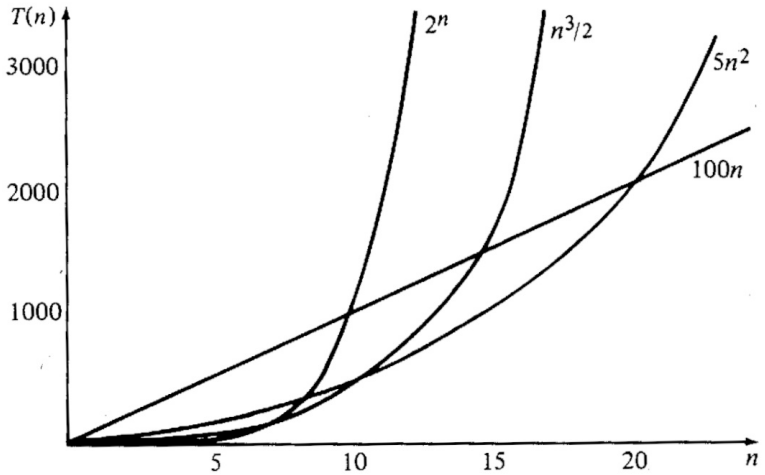


図 1.11　4 つのプログラムの実行時間

実行時間 $T(n)$	10^3 秒で解ける問題の大きさ	10^4 秒で解ける問題の大きさ	改善率
$100n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

図 1.12　コンピュータが 10 倍速くなったときの効果

実行時間の計算については「データ構造とアルゴリズム（培風館）、Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman著）」などを参照

エイホらの一言

- アルゴリズムの複雑さは重要な概念ではある。
- しかし
  - わずかな回数しか使わないプログラムでは、プログラム作りのコストの方が大きい
  - 小さな入力にしか使わないプログラムではオーダーよりも係数が重要
  - どんな効率の良いアルゴリズムでも、複雑すぎて多くの人が理解できないと意味がない
  - 効率は良くても、メモリを使うために遅い外部記憶に頼って遅くなるアルゴリズムもある
  - 数値計算では効率だけでなく、精度も重要である

試験について

- 持ち込み不可
- 不正行為は厳重に処分
- 再試の有無は皆さんの結果次第
  - ボーダーの人や不合格が多い場合は実施
  - 昨年はやりました