基本データ型

- プログラム＝データ＋手順（アルゴリズム）
  - データ：メモリ上のある領域（＝アドレス＋サイズ）
    - サイズは**変数の型**できまる
- 基本データ型
  - 文字型（char): 8bit (= 1byte)：日本語は2 byte
  - 文字列: char の配列：

\0: 終端記号

“hoge”（４文字）→{‘h’,’o’,’g’,’e’,’\0’}（5 byte)

文字種	文字コード名（通称）				
	JIS コード	EUC	MSK コード	UTF-8	Unicode
制御コード	00〜1F, 7F				00+{00〜1F,7F}
ASCII 文字	20〜7E (JIS コードでは漢字コードの後で，1B2842 を前置)				00+20〜7E
カナ・記号	A1〜DF	8E+A1〜DF	A1〜DF	EF+Bx+xx	FF+61〜9F
記号 かな 漢字	1B2442 以後 {21〜74} +{21〜7E}	{A1〜F4} +{A1〜FE}	{81〜9F, E0〜EA} +{40〜7E, 80〜FC}	Ex+xx+xx	FF+01〜5D , {30,4E〜9F} +{00〜FF}
補助漢字	1B242844 以後 {21〜74} +{21〜7E}	8F +{A1〜F4} +{A1〜FE}	(無し)	(1110aaaa +10bbbbcc +10ccdddd)	(aaaabbbb +ccccdddd)

＋ は連接，x は十六進数の任意の 1 桁，a,b,c,d は 0,1 (二進数 1 桁),
数字, 大文字は十六進数 1 桁

基本データ型（続き）

- 整数
  - char (8 bit), int (32 bit), long (64 bit)
  - 負の数は２の補数で表現
  - 正の数しか扱わない場合はunsigned intなどを使う
- 小数
  - 固定小数

$$[a_{m-1}a_{m-2}\cdots a_1 a_0 . b_1 b_2 \cdots b_n] = \sum_{i=0}^{m-1} a_i 2^i + \sum_{i=1}^{n} b_i \left(\frac{1}{2}\right)^i$$

例）$[11.001001]_2 = 2 + 1 + (1/2)^3 + (1/2)^6 = [3.140625]_D$

- 問題：0.2（１０）を2進数に直してみる
  - 有限桁の２進数小数→有限桁の１０進数小数
  - 逆は成り立たない

0.2を２進数に直そうとすると

$$\begin{array}{ccccccccc} 0.2 \rightarrow & 0.4 \rightarrow & 0.8 \rightarrow & 1.6 \rightarrow & 1.2 \rightarrow & 0.4 \rightarrow & 0.8 \rightarrow & 1.6 \rightarrow & \cdots \\ 0. & 0 & 0 & 1 & 1 & 0 & 0 & 1 & \cdots \\ a_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & \cdots \end{array}$$

無限に続く

[0.001100110011…]

基本データ型（続き）

- 丸め誤差

$[0.2]_D = [0.001100110011\ldots]_2$

- 無限に続く数字を計算機では扱えないので切り捨てる

→ **丸め誤差**が発生する

- 浮動小数点型（floating point number type)
  - 丸め誤差を少なくするために効率よく小数を表現したい
  - $[0.2]_D = [0.001100110011\ldots]_2$ の最初の０がもったいないので、最上位ビットが１になるまで左シフトする（３つシフト）と、表せる数の範囲が広がる

$[0.00110011|0011\ldots]$→左に３つシフト→$[1.10011001|1\ldots]$

小数部8 bit　　小数部8 bit

$2^{-3} + 2^{-4} + 2^{-7} + 2^{-8}$　　$1 + 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8}$

$\times 2^{-3}$ シフト分

このように小数点を固定せず浮動してその位置を指定する表現を**浮動小数点表現**と呼ぶ

基本データ型（続き）

- 浮動小数点の一般的表現
  - 先ほどの表現を一般化して（ $a_0 = 1$ は省略）

s	$\tilde{p}$	$a_1 a_2 \cdots a_{k_f}$
符号	指数部	仮数部

  - 符号は1bit
  - 指数と仮数は型（と処理系）による
    - float:　 1 + 　8 + 23 = 32 bit
    - double: 1 + 11 + 62 = 64 bit

      符号　指数部　仮数部
  - 仮数の表現：biased number, 指数の値に $2^{k-1} - 1$ を加える
    - float: +127 $(= 2^{8-1} - 1)$
    - double: +1023 $(= 2^{11-1} - 1)$
  - biased number を用いると、浮動小数点の大小比較が、指数部分の2進数表現の大小が一致する（整数の比較器が使える）

floatの値の範囲

- float型：指数8bit, 仮数23bit
  - 指数 p = 0 ~ 255→biasを考えて→ −127~128
  - p = 128は特別扱い（後述）なので、最大のp は 127
    - となると最大値は $[1.1111\cdots111] \times 2^{127}$

1が23+1個　　$(a_0 = 1$ の省略分$)$

$$= 2 + \left(\frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{23}}\right) \times 10^{127\log_{10}2}$$

$$= 3.402823466385 \times 10^{38}$$

float 型

土	指数部 8	1. 仮数部小数点以下 23 桁			十進数表示	p	
0	11111111	1111111	11111111	11111111	NaN	128	: 非数
0	11111111	0000000	00000000	00000001	NaN	128	: 非数
0	11111111	0000000	00000000	00000000	Infinity	128	=2^128 : 無限大
0	11111110	1111111	11111111	11111111	3.402823466385e+38	127	<2^128 : 最大数
0	11111110	0000000	00000000	00000001	1.701412037429e+38	127	
0	11111110	0000000	00000000	00000000	1.701411834605e+38	127	=2^127
0	11111101	1111111	11111111	11111100	1.701411428957e+38	126	
0	10000011	0000000	00000000	00000000	16	4	=2^4
0	10000010	0000000	00000000	00000000	8	3	=2^3
0	10000001	0000000	00000000	00000000	4	2	=2^2
0	10000000	1001001	00001111	11011011	3.141592741013	1	>π : 3.141592653589793
0	10000000	1001001	00001111	11011010	3.141592502594	1	<π
0	10000000	0101101	11111000	01010101	2.718281984329	1	>e
0	10000000	0101101	11111000	01010100	2.718281745911	1	<e : 2.718281828459045

浮動小数型の注意点

- 仮数部を有限桁で打ち切るので、常に誤差が付き回る
- 浮動小数型の計算は概算になる
  - 0.2 x 5.0 != 1.0　**（丸め誤差）**
    - $[0.2]_D = [0.001100110011\ldots]_2$
  - $2^{24} + 1 = 2^{24}$　**（情報落ち）**（floatを仮定、+2はOK）
    - $2^{24} + (1+1) = 2^{24} + 2$ は情報落ちしない

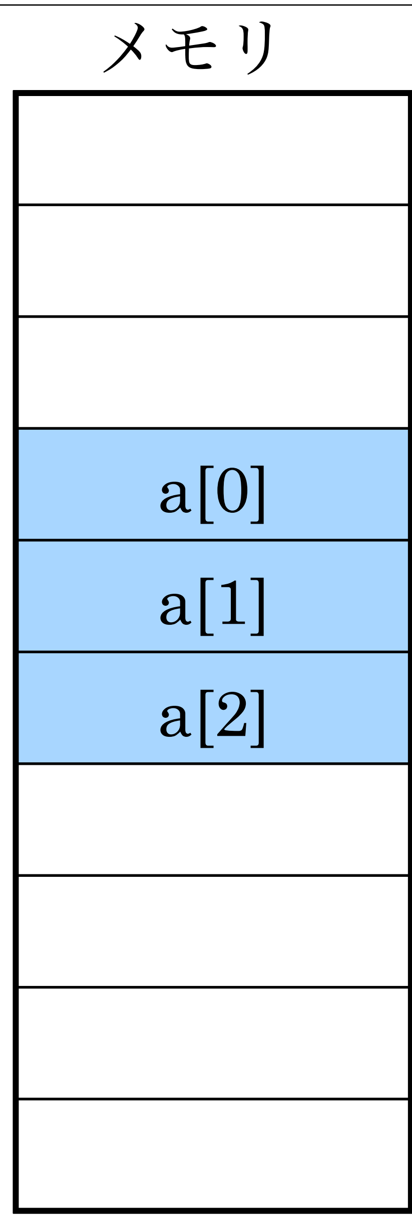
$$2^{24} + 1 = 1.0 \times 2^{24} + 1.0 \times 2^{0} = (1.0 + 1.0 \times 2^{-24}) \times 2^{24}$$

仮数部 = 1.0+0.000000000000000000000001 = 1.000000000000000000000001
桁番号:123456789012345678901234　123456789012345678901234

    - 計算の順序で結果が異なることがある
  - a - bで、a≒bの時は結果が不正確になる事がある**（桁落ち）**
    - 上位ビットが０になって有効数字が減る

基本データ型の前準備（多分復習）

- C言語の授業（プログラミング演習）でやっているはずの事
  - 配列型：同じ型のデータをメモリに連続して配置
  - 構造体：異なるデータをメモリに連続して配置
  - ポインタ：メモリの領域を格納する変数



int a[3];　利用する際にはサイズを決める必要がある

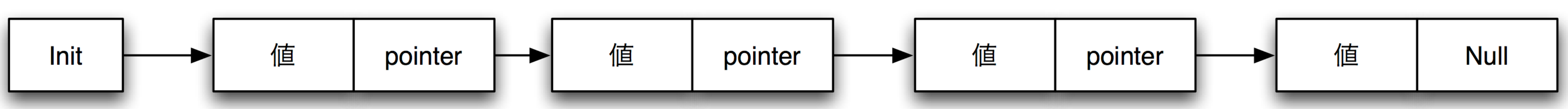
a[0] → *a　配列名は配列の最初の要素へのポインタ

a[1] → *(a+1)　メモリ上で連続している

基本データ型：リスト型

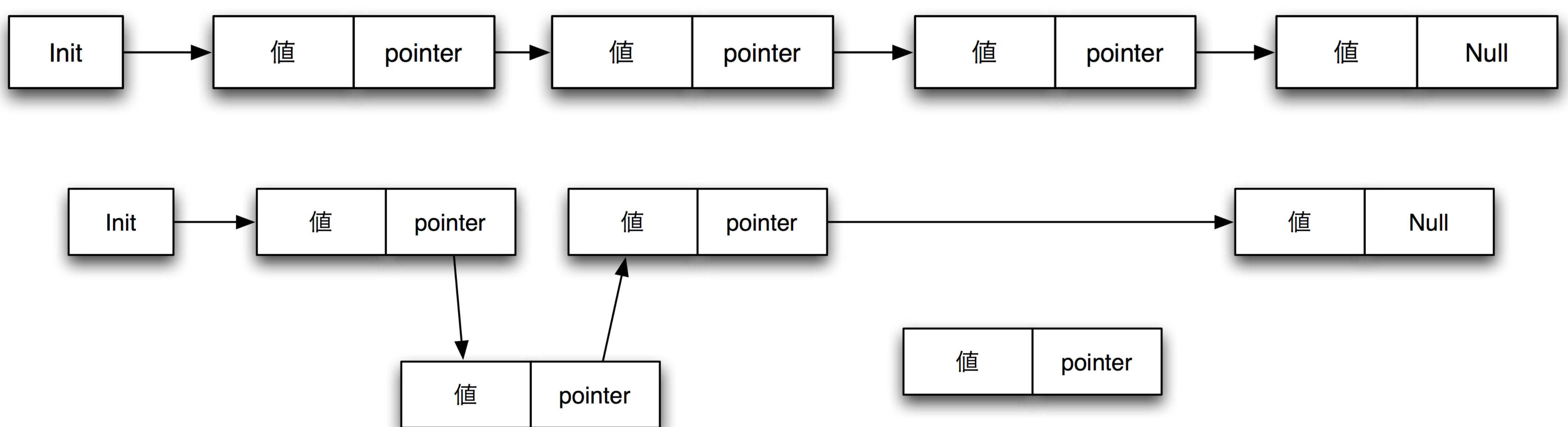
- リスト型
  - 要素を０個以上並べた型、配列型の拡張
  - 配列のように領域をあらかじめ確保する必要がない
    - 柔軟に要素の追加・削除が可能
  - メモリ上で連続していなくてもよい
    - 連続していないので、ポインタ演算で値を参照できない
    - 最初の要素から順に参照する

```
struct list {
    int element;
    struct list *next;
};
```



リストの基本操作

- 挿入
  - 値を入れる領域を確保（malloc)
  - このアドレスを挿入する箇所のpointerに代入
  - 確保した領域のpointerに次の要素のアドレスを代入
- 削除
  - 削除する前の要素のpointerへ次の要素のアドレスを代入
  - 削除する要素の領域を開放(free)



挿入（Insert)　　削除（Delete)

基本データ構造：スタック

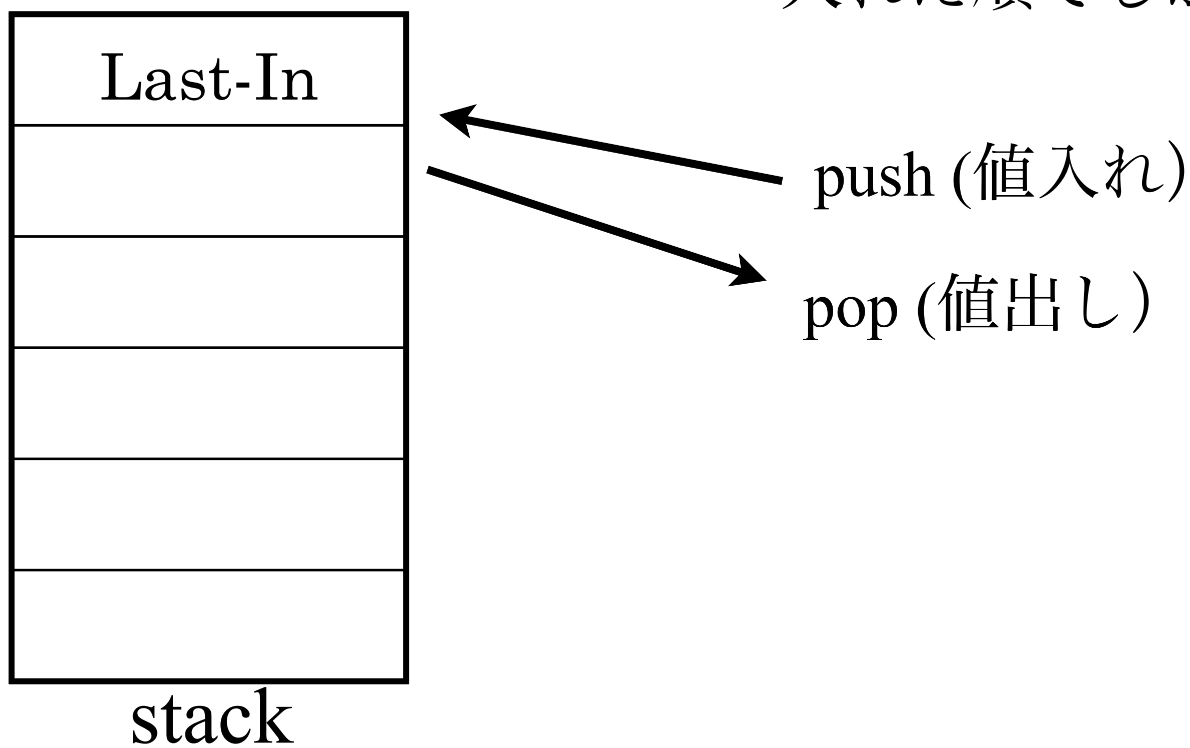
- スタック（stack):要素の取り出し・格納が先頭からなされるリスト

(Last-In-First-Out: LIFO)

後入れ先出し

入れた順でしか出せない

スタックに積む



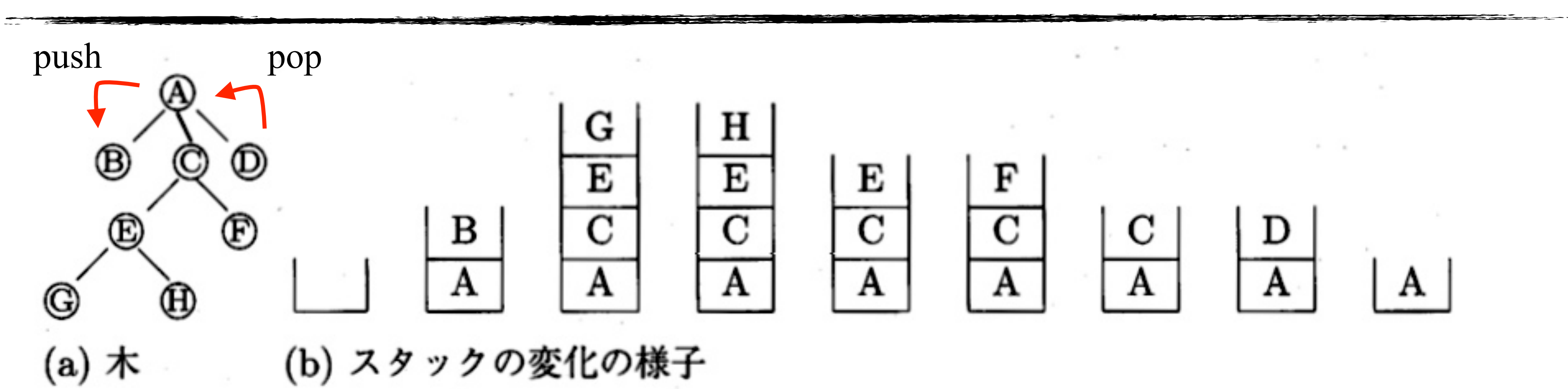


図 6.7 木構造の節点訪問

基本データ構造：キュー

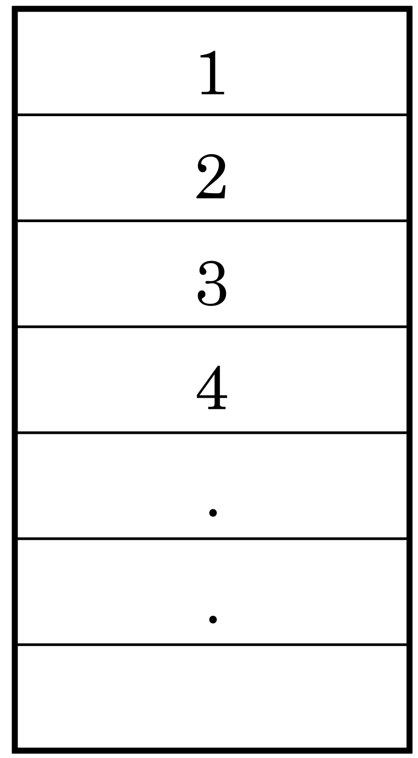
- キュー（queue）:要素の取り出しが入った順になされるリスト

(First-In-First-Out: LIFO)

先入れ先出し

先に入れたのから出す

deque (値出し)



enque (値入れ)

キューに並べる

1 2 3

待ち行列（ATM、レジ、エサ待ちなど）を表現

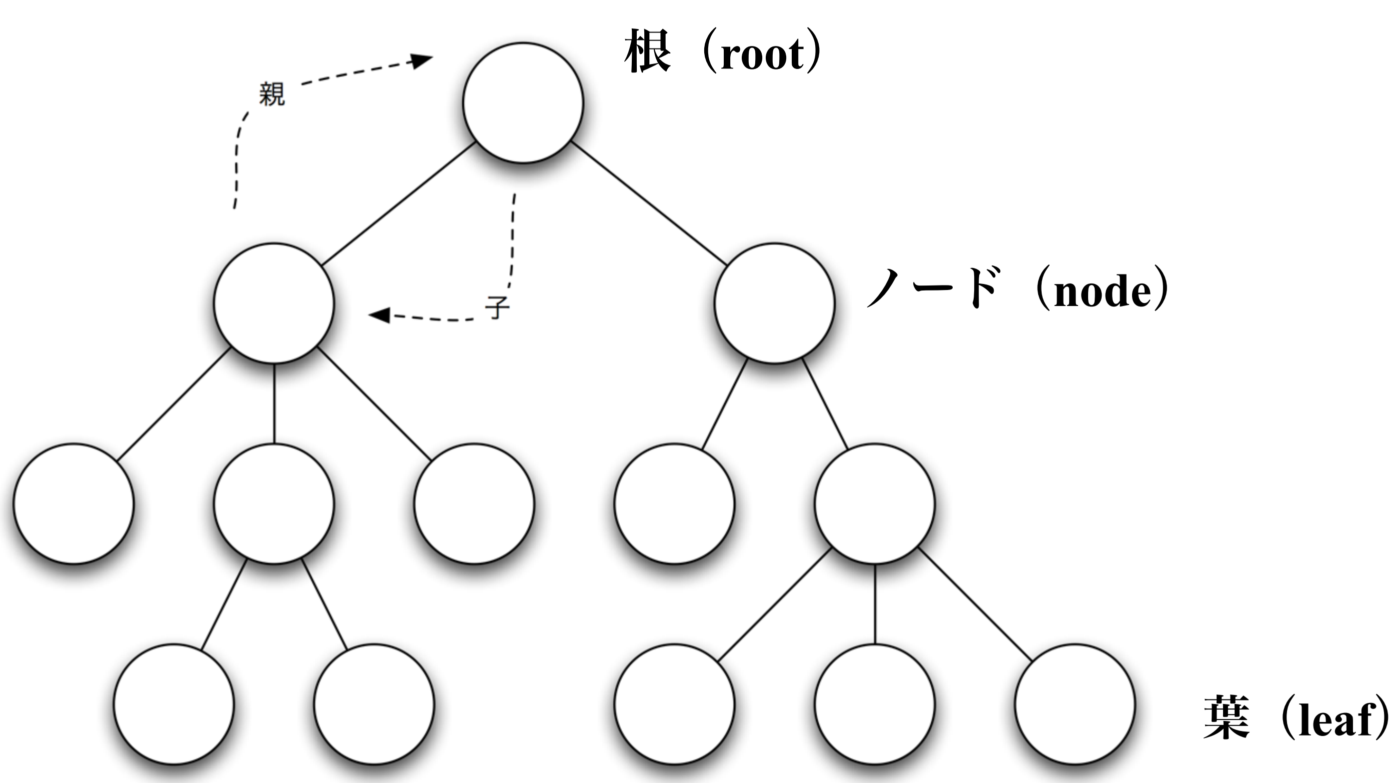
キュー・スタック共に配列・リストを利用して

実装可能

最近の言語には最初から型としてある事もある

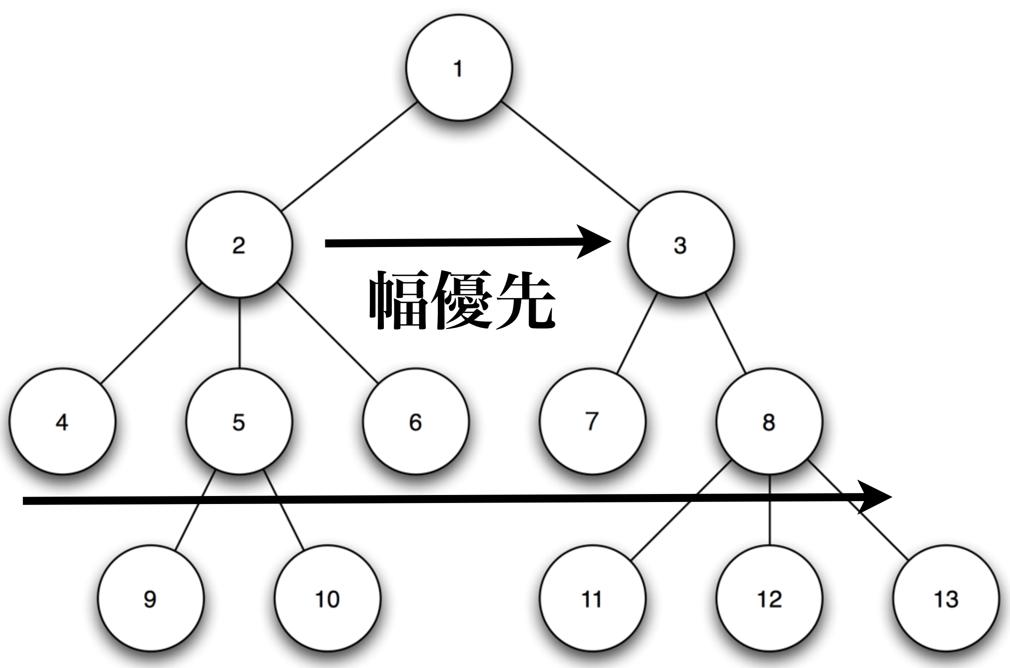
基本データ構造：木構造

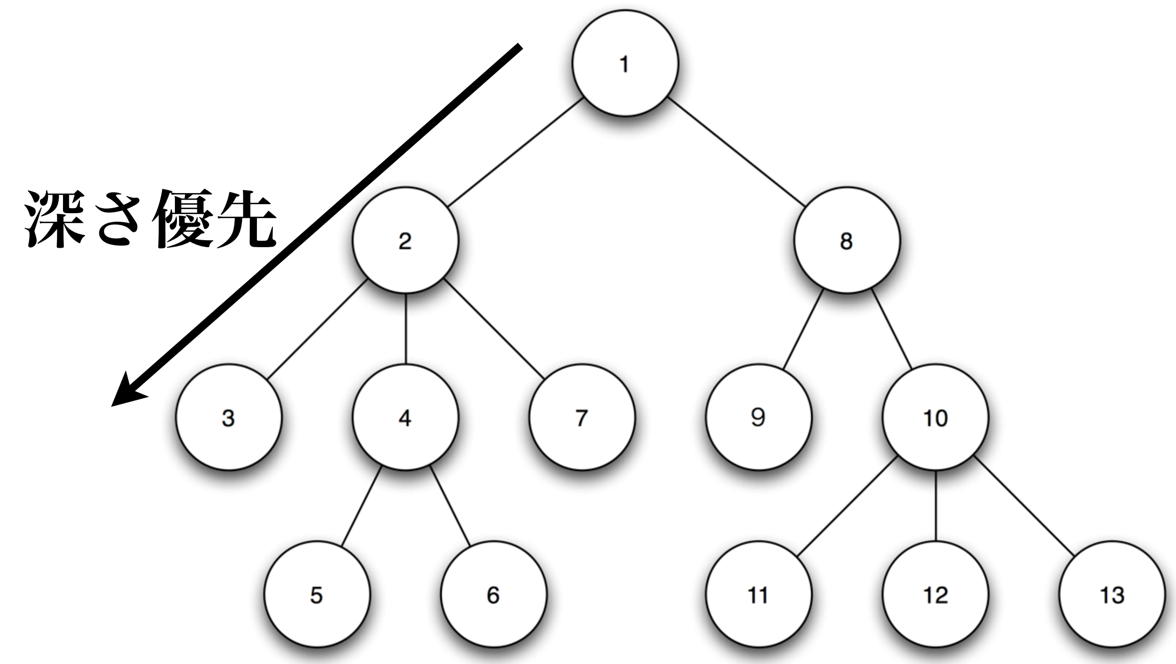
- 木（tree）：グラフの一種
  - グラフ：点（node)と線（edge)からなる集合
- 階層的なデータの表現に適している
- 子供が2つだけの木構造を2分木（binary tree)と呼ぶ



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

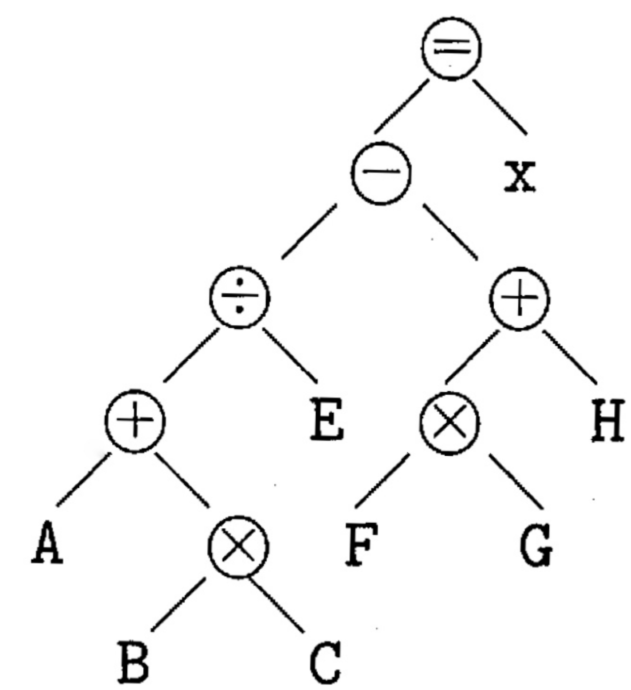


図 6.13 数式の木表現

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