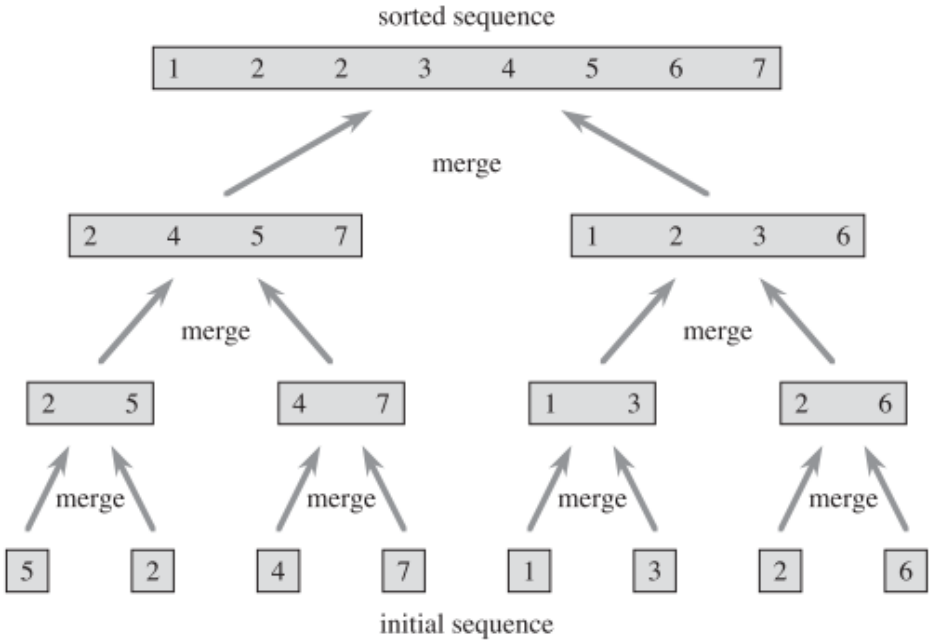
アルゴリズムの設計と解析

教授: 黄 潤和 （W4022）
rhuang@hosei.ac.jp
SA: 広野 史明 （A4/A8）
fumiaki.hirono.5k@stu.hosei.ac.jp

Contents (L4 - Trees)

- Review of Data structures for trees
- Tree traversal
  - Preorder 行きがけ順なぞり
  - Postorder 帰りがけ順なぞり
  - Inorder traversal 通りがけ順なぞり
- Binary Tree 二分木
- Binary Tree Applications

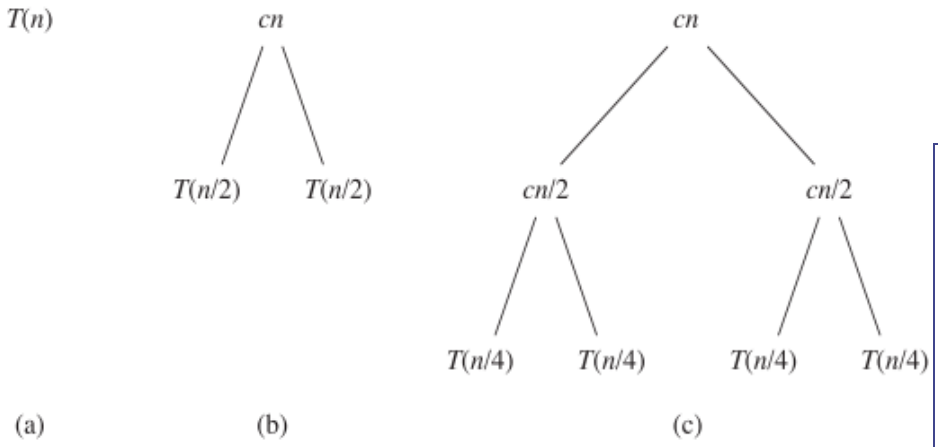
An example



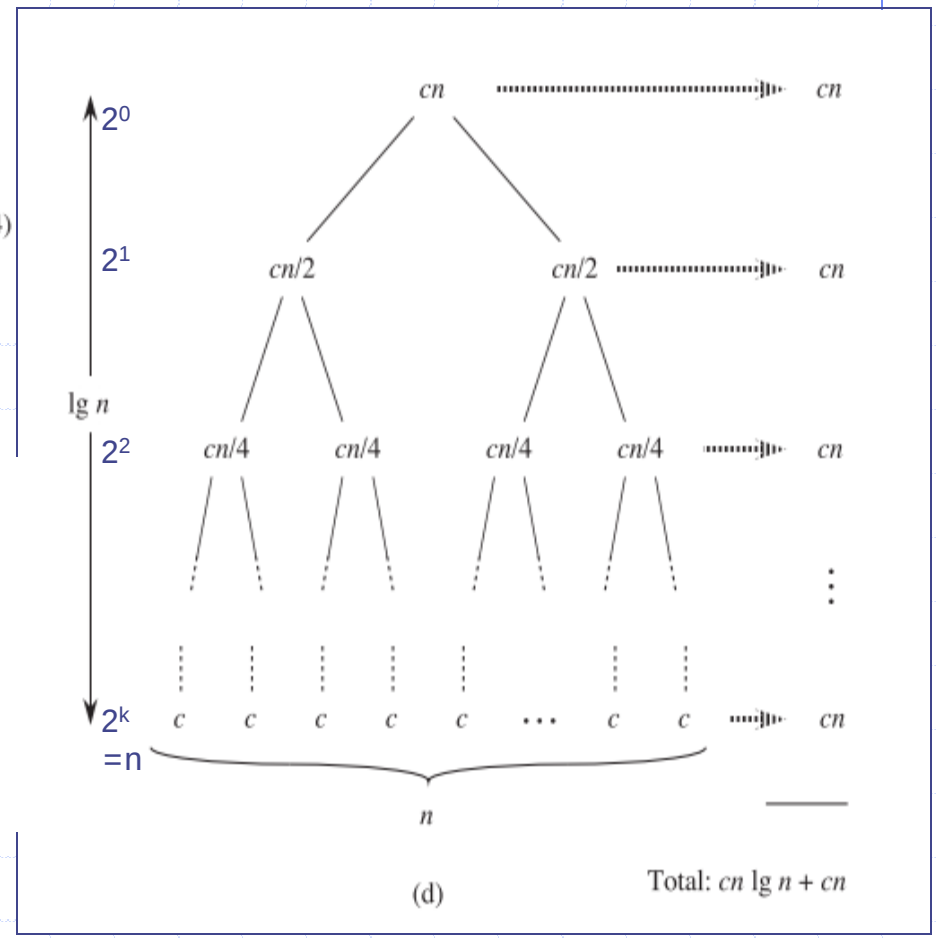
The operation of merge sort on the array A [5; 2; 4; 7; 1; 3; 2; 6]. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

MERGE-SORT $A[1 \ldots n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \ldots \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \ldots n\,]$.
3. "***Merge***" the 2 sorted lists.



$2^k = n$
$k \lg(2) = \lg(n)$
$k = \lg(n)$



$T(n)$ **MERGE-SORT** $A[1 \ldots n]$

$\Theta(1)$ 1. If $n = 1$, done.

$2T(n/2)$ 2. Recursively sort $A[1 \ldots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$.

$\Theta(n)$ 3. "***Merge***" the 2 sorted lists.

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

$\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.

$T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

The worse case: reverse ordered;
The best case: complete sorted

Review of some analysis notations

Θ-notation

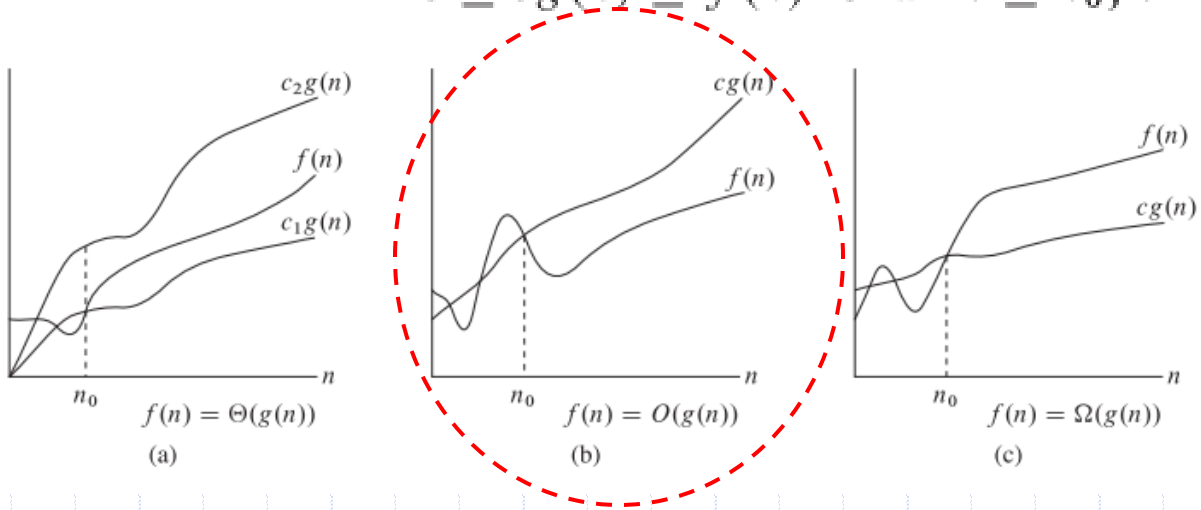
$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.^1$$

O-notation asymptotic upper bound,

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

Ω-notation asymptotic lower bound

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$



漸近的な解析で用いる４つの漸近記号

実行時間の漸近的な解析においては，次の5つの漸近記号を用いる

- Θ-記法（シータ記法）
  - 実行時間の漸近的な振る舞いを厳密に（正確に）評価
- O-記法（ビッグオー記法）
  - 実行時間の漸近的な振る舞いの上界を評価
- Ω-記法（ビッグオメガ記法）
  - 実行時間の漸近的な振る舞いの下界を評価
- o-記法（リトルオー記法）
  - 実行時間の漸近的な振る舞いの上界を（ゆるく）評価
- ω-記法（リトルオメガ記法）
  - 実行時間の漸近的な振る舞いの下界を（ゆるく）評価

漸近記号（まとめ）asymptotic summary

記法	定義	直観的解釈
$f(n) = \Theta(g(n))$	$\exists c_1, c_2, n_0 \forall n \geq n_0 :$ $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$	$f(n) = cg(n) + g'(n)$ $\lim_{n\to\infty} g'(n)/g(n) = 0$
$f(n) = O(g(n))$	$\exists c, n_0 \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ (c ≥ 0)
$f(n) = \Omega(g(n))$	$\exists c, n_0 \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)$	$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ (c ≥ 0)
$f(n) = o(g(n))$	$\forall c, \exists n_0 \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) = \omega(g(n))$	$\forall c, \exists n_0 \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)$	$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Working in class (復習)

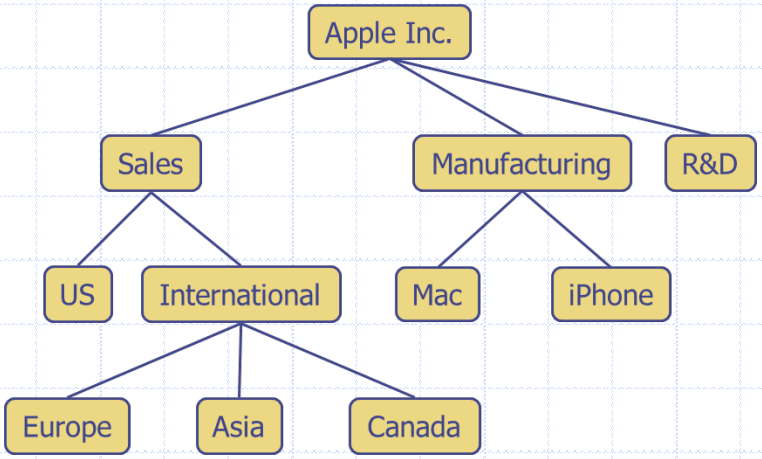
以下の表の各関数f(n)と時間tに対してもアルゴリズムが問題を解くのにf(n)マイクロ秒(10^{-6})かかるとき，t時間(1 second)で解くことができる最大のサイズnを求めよ.

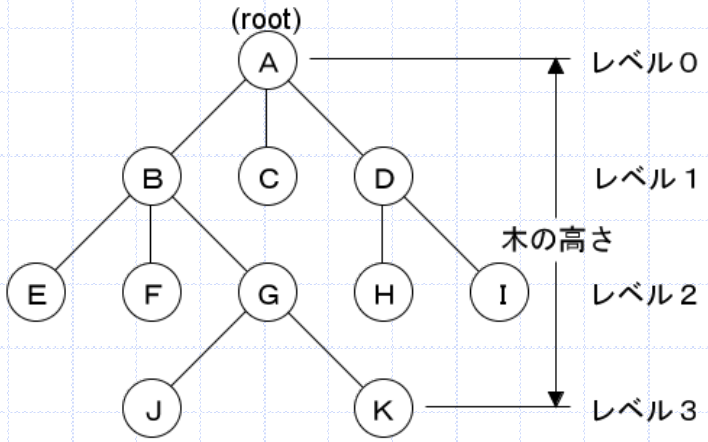
$f(n)$ \ t	1秒
$\lg n$	
$\sqrt{n}$	
n	
$n \lg n$	
n^2	
n^3	
2^n	
$n!$	

$f(n)$ \ t	1 秒
$\lg n$	$\lg n=10^6$ $n=2^{(10^6)}$
$\sqrt{n}$	10^{12}
n	10^6
$n \lg n$	$n\lg n=10^6$ $n=6\times10^4$
n^2	10^3
n^3	10^2
2^n	$2^n=10^6$ $n=\lg(10^6)=19$
$n!$	$n!=10^6$ $n=10$

What is a Tree
木とは？

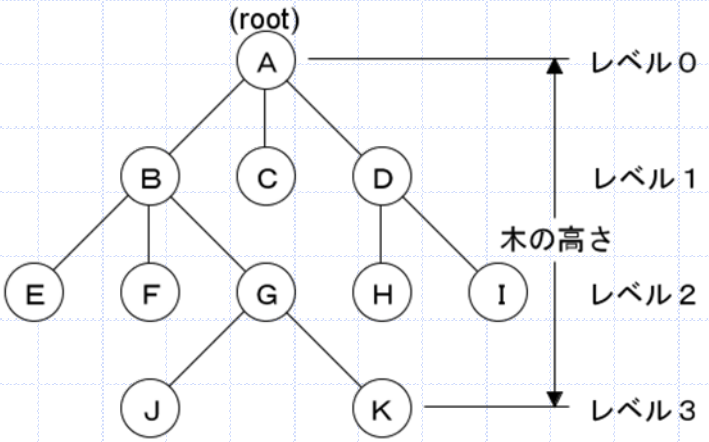
- A tree is an abstract model of a hierarchical structure
  木構造は階層構造の抽象的なモデル
- A tree consists of nodes with a parent-child relation
  木は、親子関係のあるノードで構成
- Applications:
  - Organization charts
    組織図
  - File systems
    ファイルシステム
  - Programming environments
    プログラミング環境





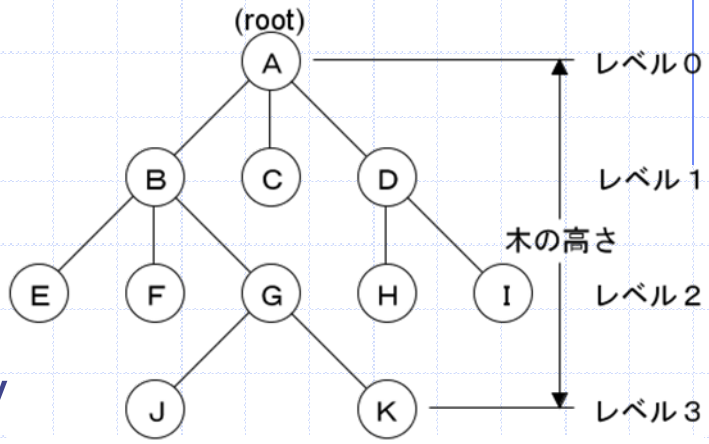
Basic elements 基本要素

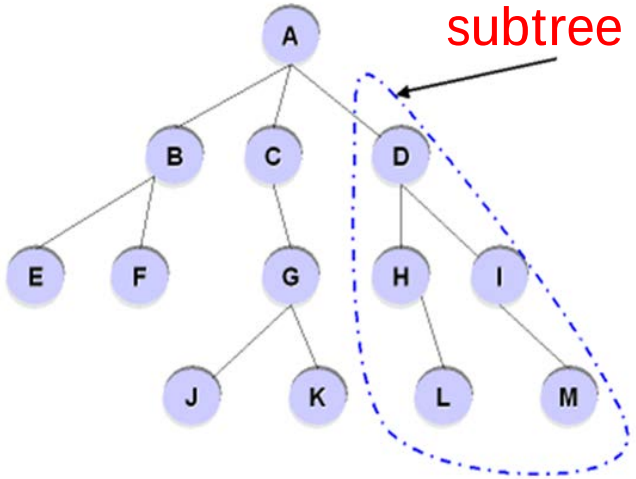
- Root: node without parent (A)
  根：親のいないノード
- Internal node: node with at least one child (A, B, C, D)
  内部節：少なくともひとつの子をもつノード
- External node (a.k.a. leaf): node without children (E, F, J, K, L, M)
  外部節(通称. 葉)：子のないノード



Terms 用語

- Ancestors of a node: parent, grandparent, grand-grandmother, etc.
  ノードの先祖：親、祖父母、曾祖母など
- Depth of a node: number of ancestors
  ノードの深さ：先祖の数
- Height of a tree: maximum depth of any node
  木の高さ：ノードの最大の深さ
- Descendant of a node: child, grandchild, grand-grandchild, etc.
  ノードの子孫：子供、孫、曾孫など
- Subtree: tree consisting of a node and its descendants
  部分木：ノードとその子孫から成る木





Tree ADT (abstract methods)

ADT: abstract data type

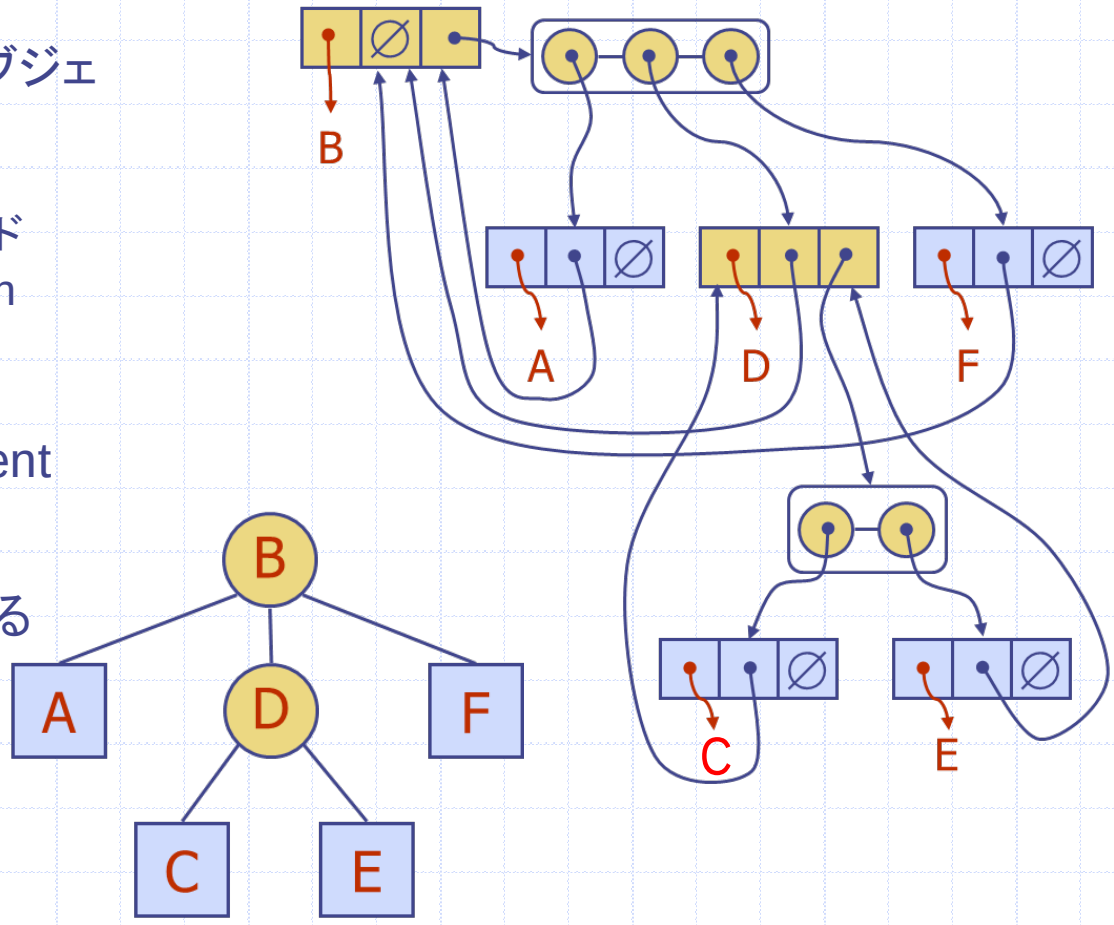
An example

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
  - nodeIterator nodes()
- Accessor methods:
  - node root()
  - node parent(p)
  - nodeIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update methods:
  - void swapElements(p, q)
  - object replaceElement(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Data Structure for Trees
木のためのデータ構造

- A node is represented by an object storing
  ノードは以下を持つオブジェクトで表現される
  - Element　要素
  - Parent node　親ノード
  - Sequence of children nodes　子ノードの列

- Node objects implement the Position ADT
  ノードオブジェクトは Position ADTを実装する



An Node implementation example in Python

http://www.quesucede.com/page/show/id/python-3-tree-implementation

node.py

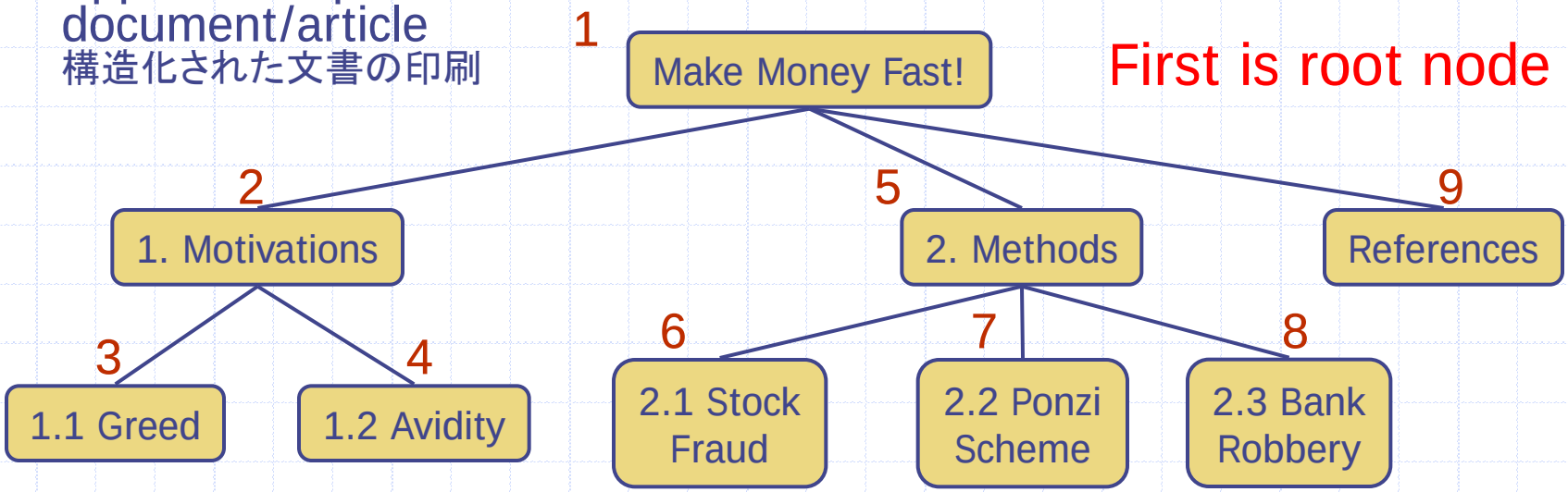
```
# Copyright (C) by Brett Kromkamp 2011-2014 (brett@perfectlearn.com)
# You Programming (http://www.youprogramming.com)
# May 03, 2014

class Node:
    def __init__(self, identifier):
        self.__identifier = identifier
        self.__children = []

    @property
    def identifier(self):
        return self.__identifier

    @property
    def children(self):
        return self.__children

    def add_child(self, identifier):
        self.__children.append(identifier)
```

Preorder Traversal
行きがけ順なぞり

- A traversal visits the nodes of a tree in a systematic manner
  なぞりは、木のノードに規則正しい方法でなぞる
- In a preorder traversal, a node is visited before its descendants ‘
  行きがけ順なぞりでは、最初に節に立ち寄る
- Application: print a structured document/article
  構造化された文書の印刷

Image:
Top (root) down to leave

```
Algorithm preOrder(T,v)
  visit(v)
  for each w in T.children(v)
  do
      preorder (T,w)
```



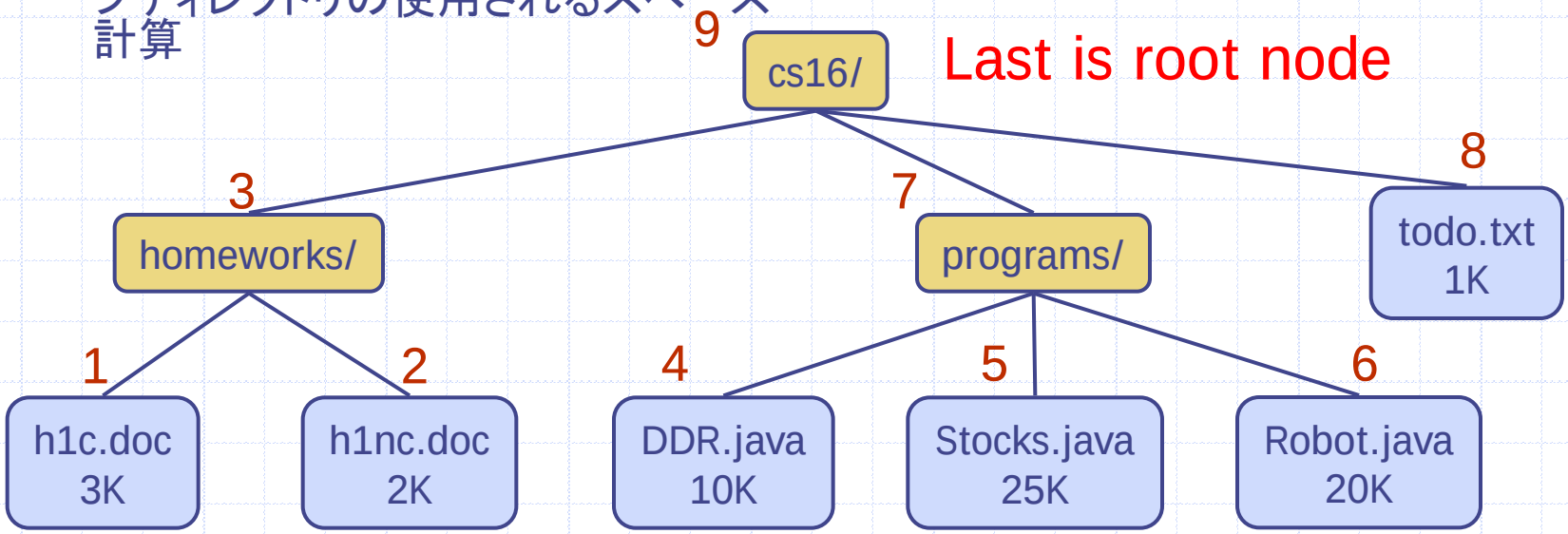
First is root node

Postorder Traversal
帰りがけ順なぞり

- In a postorder traversal, a node is visited after its descendants
  帰りがけ順なぞりでは、子孫の後にノード(節)に立ち寄る。
- Application: compute space used by files in a directory and its subdirectories
  ディレクトリの中のファイルやそのサブディレクトリの使用されるスペース計算

Image:
Bottom (leave) up to root

Algorithm **postOrder**(T,v)
for each w in T.children(v)
do
    postOrder (T,w)
    visit(v)



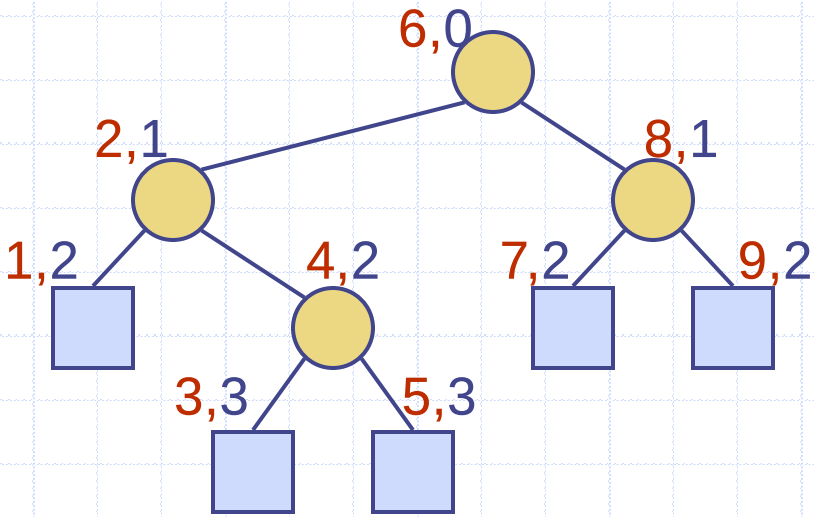
Inorder Traversal
通りがけ順なぞり

- In an inorder traversal a node is visited after its left subtree and before its right subtree
  通りがけ順なぞりでは、ノードの前に左の部分木、ノードの後に右の部分木をなぞる。
- Application: draw a binary tree
  - $x(v)$ = inorder rank of v
  - $y(v)$ = depth of v

Note: node insertion follows Inorder

Image:
left branch (**middle**) **right** branch

```
Algorithm inOrder(T,v)
    if T.hasLeft (v)
        inOrder (T.left(v))
    visit(v)
    if T.hasright (v)
        inOrder (T.right (v))
```



BinaryTree ADT
二分木ADT

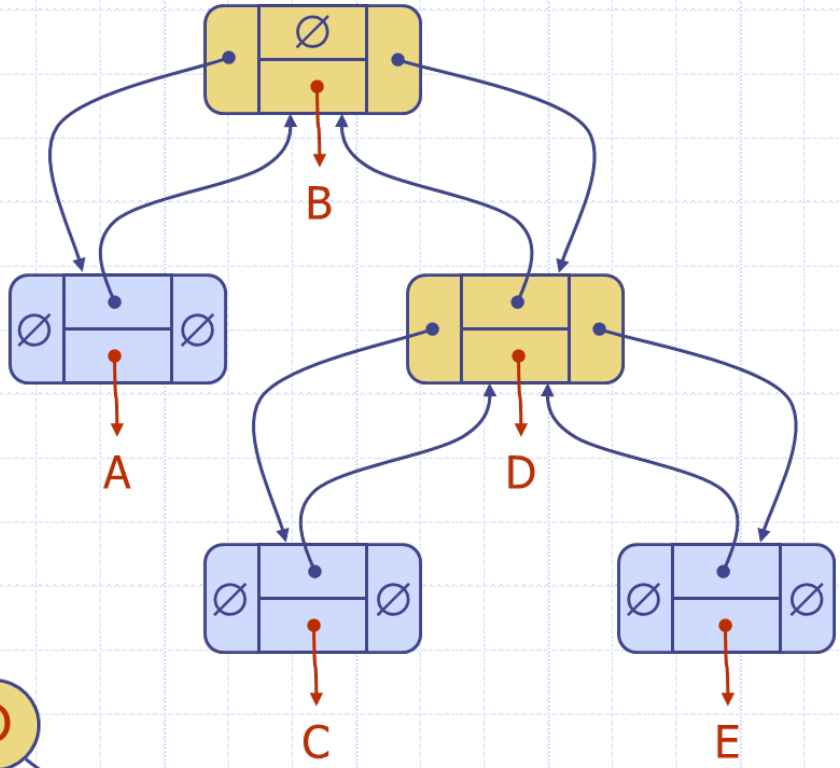
- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT

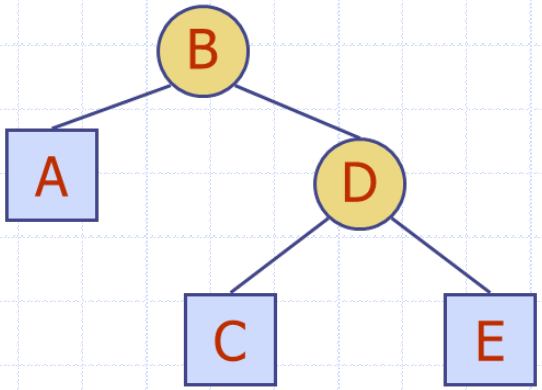
  二分木ADTは木ADTを拡張したものである。
- Additional methods:
  - node leftChild(*p*)
  - node rightChild(*p*)
  - node sibling(*p*)
- Update methods may be defined by data structures implementing the BinaryTree ADT

  二分木ADTを実装するデータ構造によりメソッドが定義される

Data Structure for Binary Trees
二分木のデータ構造

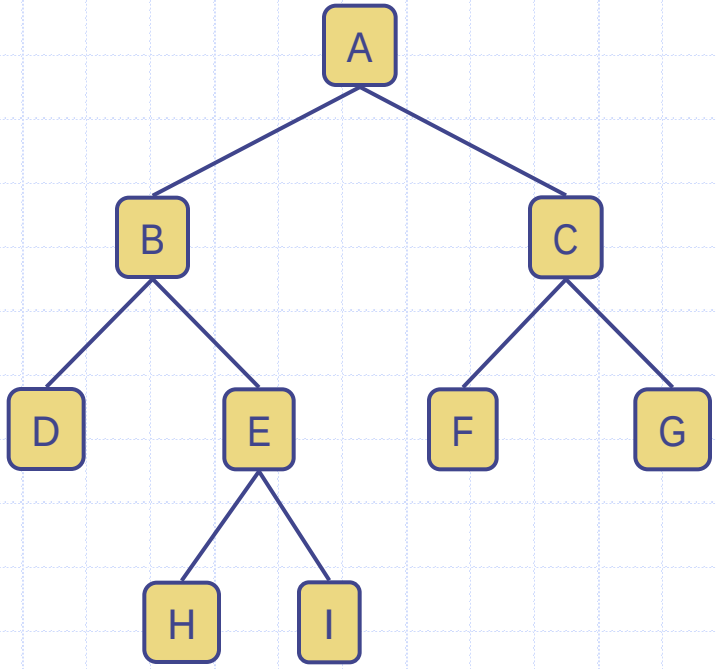
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT





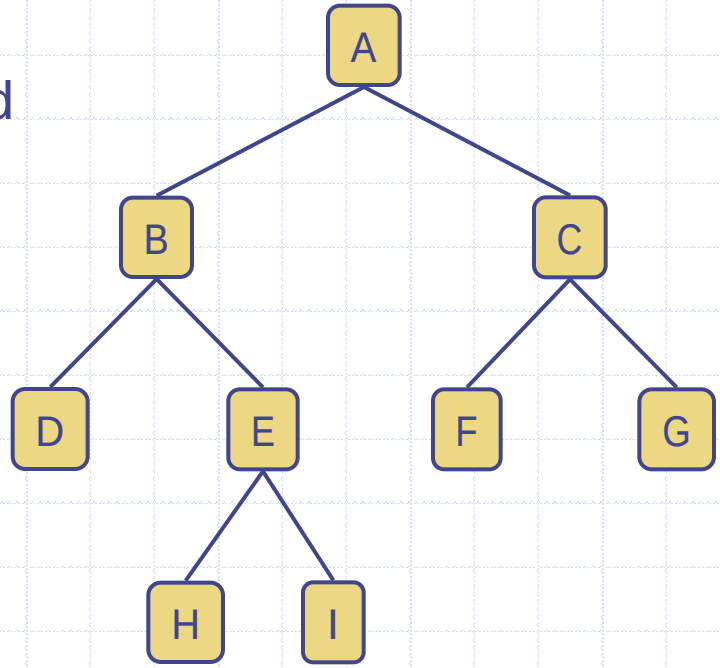
Binary Tree
二分木

- A binary tree is a tree with the following properties:
  二分木は以下の特徴がある:
  - Each internal node has two children
    それぞれの内部節には2つの子がある
  - The children of a node are an ordered pair
    ノードの子は順序対である
- We call the children of an internal node left child and right child
- それぞれの子ノードは「左」「右」と呼ばれる



Binary Tree
二分木

- Alternative recursive definition: a binary tree is either
  別の再帰的定義：二分木は
  - a tree consisting of a single node, or
    ただひとつのノードから成るか、
  - a tree whose root has an ordered pair of children, each of which is a binary tree
    根が二分木である子供の1順序対から成る木
- Applications:
  - arithmetic expressions
    算術表現
  - decision processes
    決定プロセス
  - searching
    探索

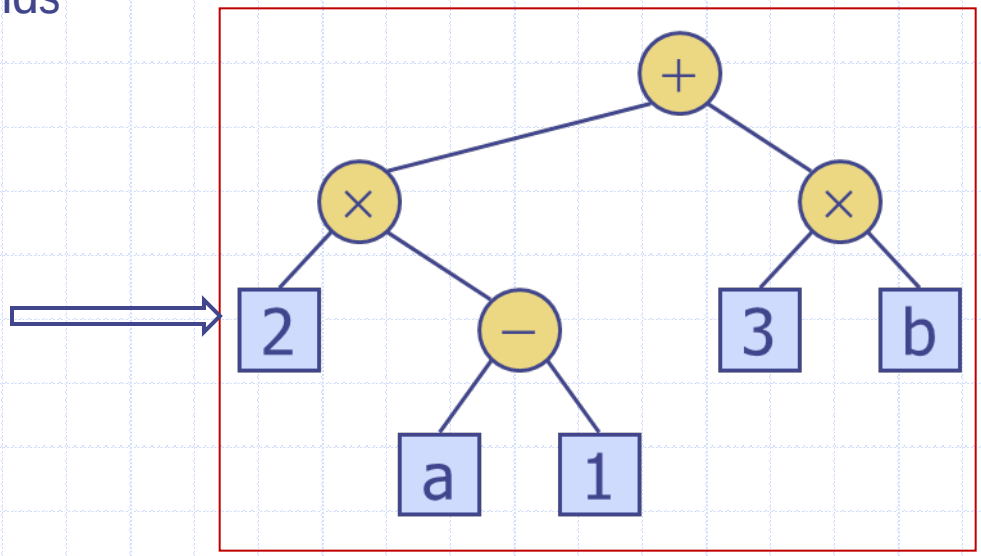


Arithmetic Expression Tree
算術式の木

- Binary tree associated with an arithmetic expression
  二分木の内部節に算術式を関連づける
  - internal nodes: operators
    内部節：算術演算子
  - external nodes: operands
    外部節：算術演算子

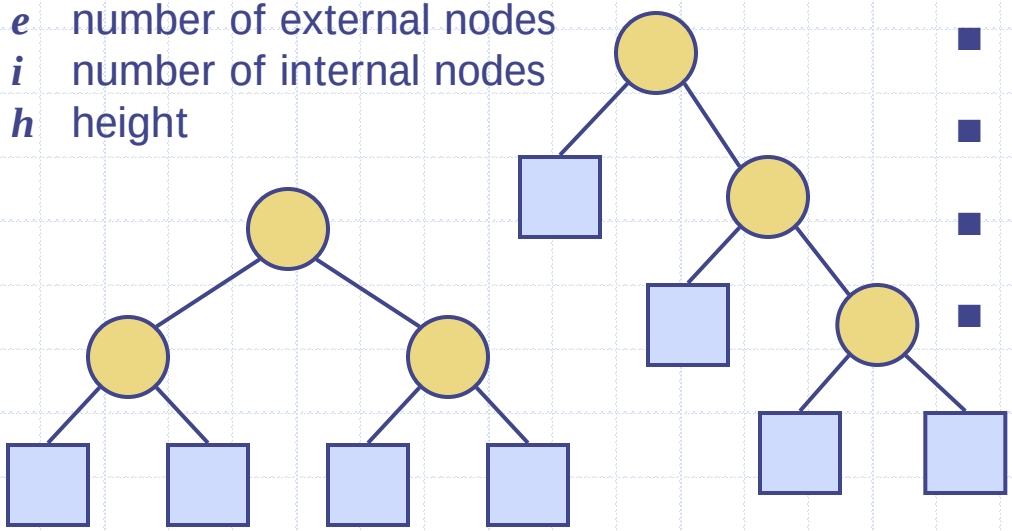
  Example: arithmetic expression tree for the expression

$(2 \times (a - 1) + (3 \times b))$



Properties of Proper Binary Trees
適切な二分木の特性

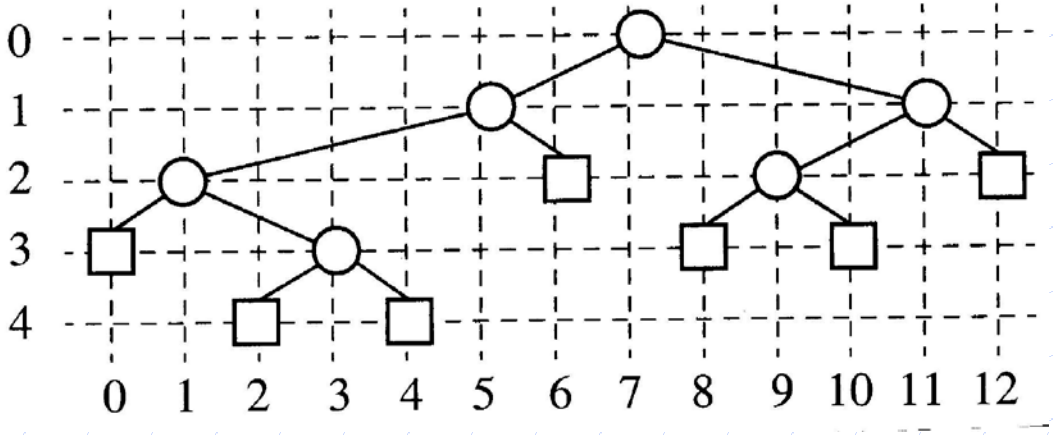
- A binary tree is proper if each internal node has exactly two children.
  それぞれの内部節に2つの子があるなら、その二分木は適切である
- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height



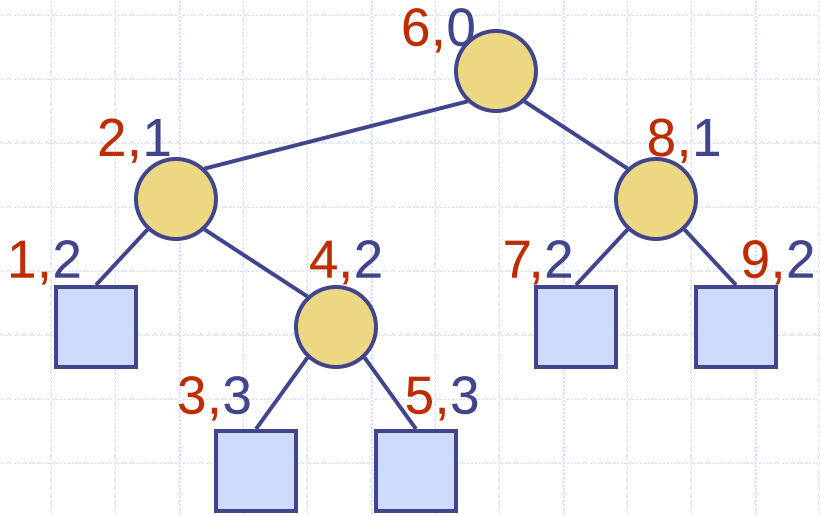
- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

Tree Drawing
木の書き方

https://cis.k.hosei.ac.jp/~rhuang/Miccl/Algorithm3/treeDemo/index.html



⇒ ?

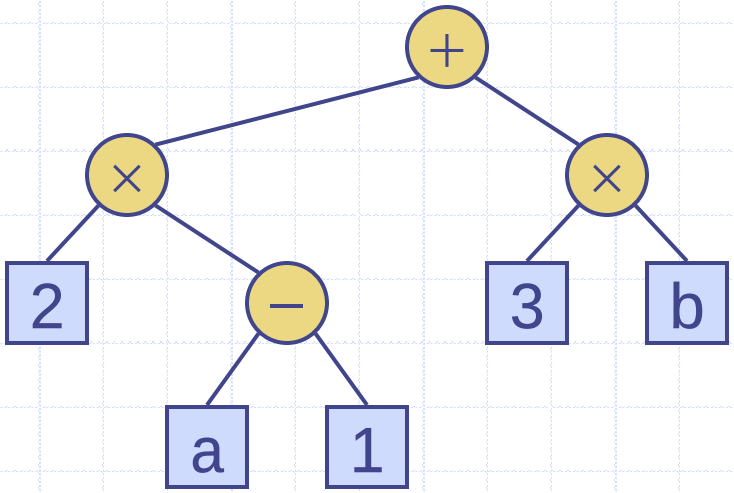


height
in blue: 0, 1 ,2 , 3

Visiting order (inorder traversal)
in red: 1, 2, 3, 4, 5, 6, 7, 8, 9

Print Arithmetic Expressions
算術式の表示 (using inorder)

- Specialization of an inorder traversal
  通りがけ順なぞりの特殊な点
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

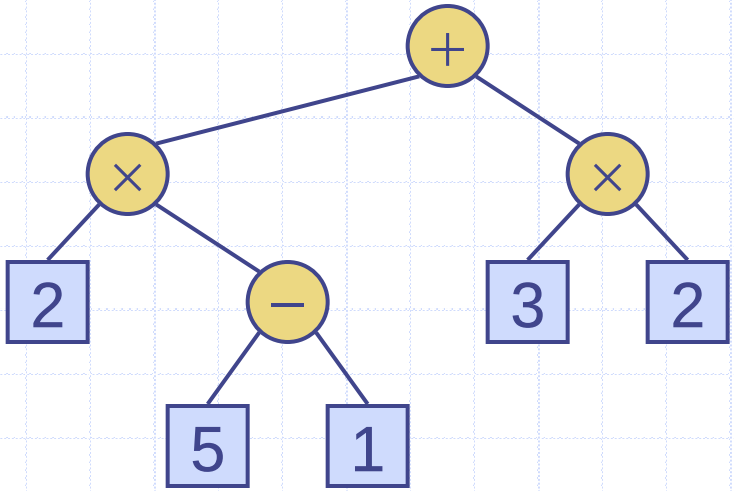


Algorithm printExpression(T,v)
  if T.isExternal (v)
    print the value at v
  else
    print("(")
    printExpression(T,T.left(v))
    print the operator at v
    printExpession(T,T.right(v)
    print (")")

$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expressions
算術式の評価

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
    部分木の値を返す再帰的メソッド
  - when visiting an internal node, combine the values of the subtrees
    内部節をなぞったとき、部分木の値を結合



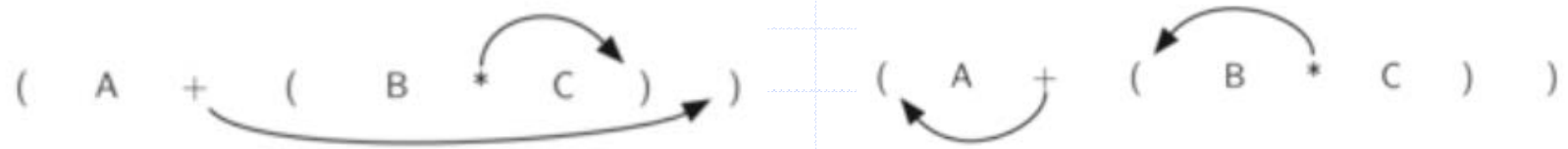
Algorithm evalExpr(*T*,*v*)

```
if T.isExternal (v)
    return v.element ()
else
    x ← evalExpr(T.left(v))
    y ← evalExpr(T.right (v))
    ◊ ← operator stored at v
    return x ◊ y
```

Infix to Prefix/Infix to Postfix Expressions

Infix Expression	Prefix Expression	Postfix Expression
(A + B) * C	* + A B C	A B + C *
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

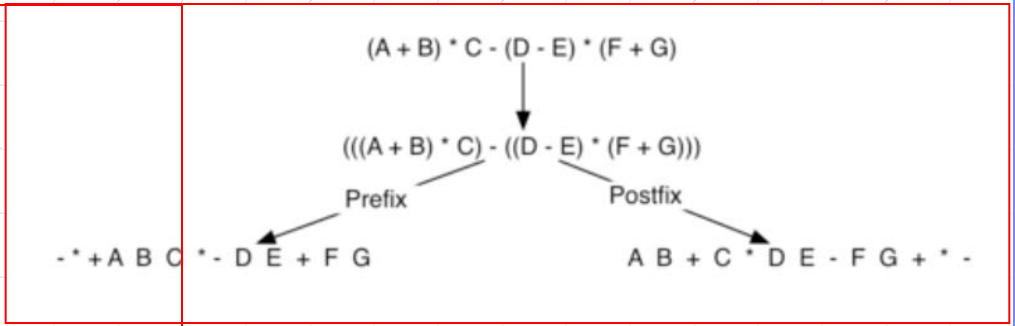
Conversion of Infix Expressions to Prefix and Postfix



http://interactivepython.org/courselib/static/pythonds/BasicDS/InfixPrefixandPostfixExpressions.html

Infix to Postfix Expression in Python

```
def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    rec["-"] = 2
    prec["("] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.isEmpty()) and ¥
               (prec[opStack.peek()] >= prec[token]):
                  postfixList.append(opStack.pop())
            opStack.push(token)

    while not opStack.isEmpty():
        postfixList.append(opStack.pop())
    return " ".join(postfixList)

print(infixToPostfix("A * B + C * D"))
print(infixToPostfix("( A + B ) * C - ( D - E ) * ( F + G )"))
```




```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

```
A B * C D * +

A B + C * D E - F G + *
```

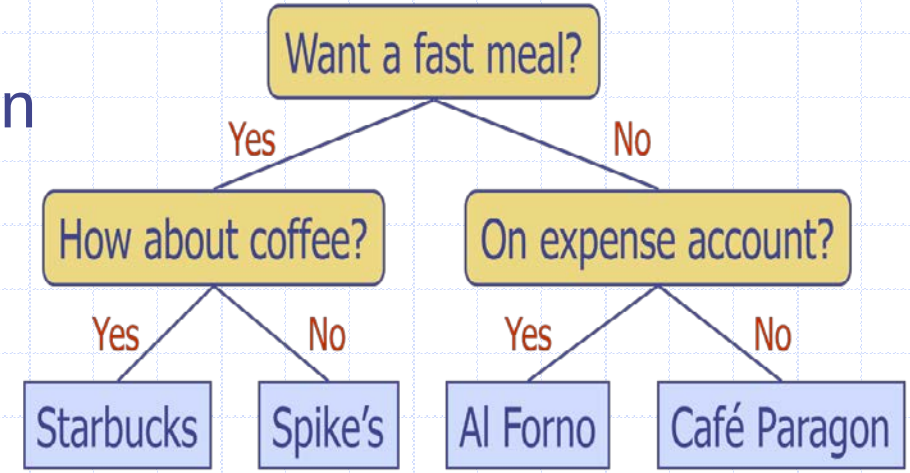
Decision Tree
決定木

- Binary tree associated with a decision process

  決定プロセスに関連付けられる二分木

  - internal nodes: questions with yes/no answer
    内部節：yes/no 質問事項
  - external nodes: decisions
    外部節：決定

- Example: dining decision



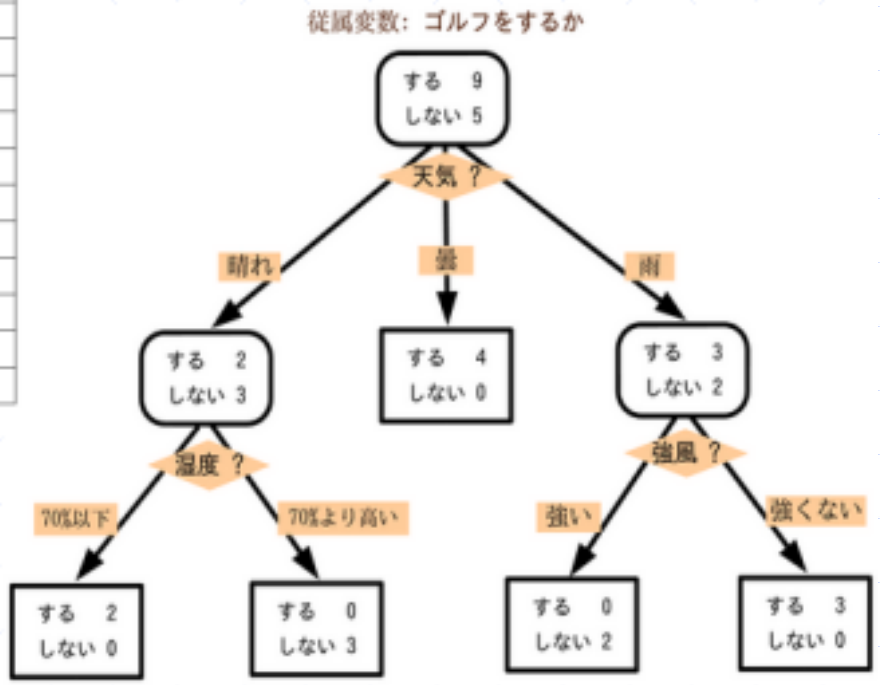
Decision Tree for the Golf Problem
決定木の応用

Decision tree can make complicated data support easy decision.
元の複雑なデータの表現を簡単な構造に変換にするのに役に立つ。

Play golf dataset

Independent variables				Dep. var
OUTLOOK	TEMPERATURE	HUMIDITY	WINDY	PLAY
sunny	85	85	FALSE	Don't Play
sunny	80	90	TRUE	Don't Play
overcast	83	78	FALSE	Play
rain	70	96	FALSE	Play
rain	68	80	FALSE	Play
rain	65	70	TRUE	Don't Play
overcast	64	65	TRUE	Play
sunny	72	95	FALSE	Don't Play
sunny	69	70	FALSE	Play
rain	75	80	FALSE	Play
sunny	75	70	TRUE	Play
overcast	72	90	TRUE	Play
overcast	81	75	FALSE	Play
rain	71	80	TRUE	Don't Play

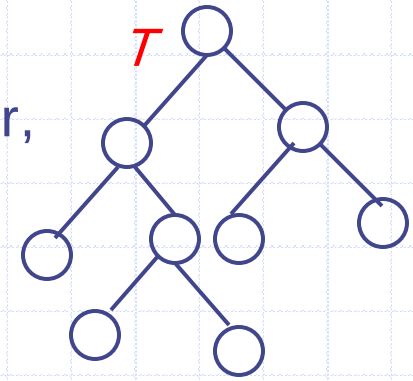
Guests data for a golf club
ゴルフクラブの来客データセット



Exercise 4-1（work in class）

For each node of the binary tree *T*, show its ranks for the three traversals, (pre-order, in-order, post-order).

Tのノードが行きがけ順と通りがけ順と帰りがけ順の順位を書いてください。



Exercise 4-2

Draw a single binary tree T such that

- each internal node of T is a single character
- a pre-order traversal of T yields EXAMFUN; and
- an in-order traversal of T yields MAFXUEN

Tのそれぞれの内部のノードが単独の文字であるように
行きがけ順でEXAMFUN　通りがけ順でMAFXUEN
となるような単一の二分木を書いてください。

Exercise 4-3

Draw a single binary tree T such that

- each internal node of T is a single character
- a post-order traversal of T yields AGLRTHSMIO; and
- an in-order traversal of T yields ALGORITHMS

Tのそれぞれの内部のノードが単独の文字であるように
帰りがけ順でAGLRTHSMIO　通りがけ順でALGORITHMS
となるような単一の二分木を書いてください。

Exercise 4-4

Implement Infix to Prefix expression in Python. You may refer to the source code of Infix to Postfix Expression presented in this slide.

References:

- http://interactivepython.org/courselib/static/pythonds/BasicDS/toctree.html
- http://knuth.luther.edu/~leekent/CS2Plus/genindex.html