第11回
知識(履歴)の蓄積

情報処理演習II

Literacy

例題

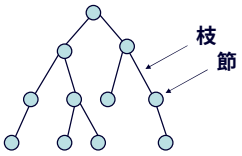
Literacy

例題1．二分木

- 二分木の1つの節を，構造体 node 型として定義する
  - node 型は，left，right の2つの属性から構成

node 型

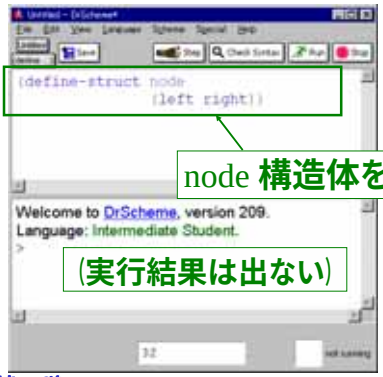
left
right



二分木の例

Literacy

二分木の定義



node 構造体を定義している

(実行結果は出ない)

Literacy

node 構造体

構造体の名前

```
(define-struct node
                (left right))
```

属性の並び
(それぞれの属性にも名前がある)

Literacy

node 構造体

名前　　属性の並び

```
(define-struct node (left right))
```

```
make-node : (tree tree -> node)   コンストラクタ
node-left : (node -> tree)
node-right : (node -> tree)       } セレクタ
node? : (anything -> boolean)     node 構造体かを調べる
```

(注) コンストラクタ，セレクタで「tree」とあるのは，二分木のこと
(次ページ参照)

Literacy

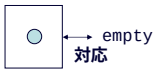
二分木(binary tree)

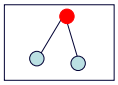
二分木 (binary tree) の再帰的定義

1. empty (再帰的定義の基底)
2. 部分木を持つ節を表す node 構造体

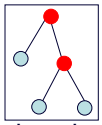
```
(make-node tleft tright)
          tleft ･･･ 左の部分木
          tright ･･･ 右の部分木
```

empty の場合





```
(make-node
  empty
  empty)
```



```
(make-node
  empty
  (make-node
    empty
    empty))
```

左の部分木も右の部分木も無いような二分木
(中身が無い二分木)

Literacy

実習

Literacy

実習の進め方

- 資料を見ながら、「実習」を行ってみる
- その後、各自「課題」に挑戦する
  - 各自で自習 ＋ 巡回指導
  - 遠慮なく質問してください
- 自分のペースで先に進んで構いません

Literacy

DrScheme の使用

- DrScheme の起動
  - プログラム → PLT Scheme → DrScheme
- 今日の実習では「Intermediate Student」に設定
  - Language
  - → Choose Language
  - → Intermediate Student
  - → OK ボタン

Literacy

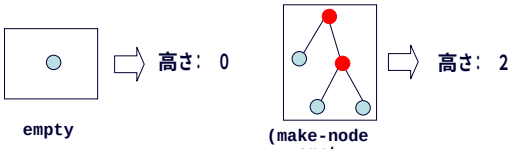
DrScheme の使用 / ステップ実行

- プログラム実行の振る舞いを観察するツール
- 「定義用ウインドウ」のみを使用
  (通常の実行と異なる)
- 「Intermediate Student」に設定する必要あり
  - Language → Choose Language
  - → Intermediate Student
  - → Run ボタン

Literacy

実習1. 二分木の高さ

二分木の高さを求める関数 height を定義し，実行する




```
empty
```

```
(make-node
  empty
  (make-node
    empty
    empty))
```

Literacy

「実習1．二分木の高さ」の手順

1. 次を「定義用ウインドウ」で，実行しなさい
   - 入力した後に，Run ボタンを押す

```
(define-struct node
          (left right))
;; height : tree  ->  number
;; measure the height of a tree abt
(define (height abt)
 (cond
  [(empty? abt) 0]
  [else (+ 1
           (max (height (node-left abt))
                (height (node-right abt))))]))
```

2. その後，次を「実行用ウインドウ」で実行しなさい

```
(height empty)
(height (make-node
     empty
     (make-node
        empty
        empty)))
```

Literacy

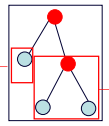
二分木の高さの実行例

Literacy

二分木の高さ

1. 入力 abt が empty ならば： → 終了条件
   0 → 自明な解
2. そうで無ければ：
   – 右の部分木と、左の部分木の高さを求め、大きいほうに1を加える

左の部分木の高さは0 (empty なので)



右の部分木の高さは1

Literacy

height 関数

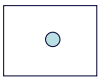
```
;; height : tree  ->  number
;; measure the height of a tree abt
(define (height abt)
 (cond   終了条件   自明な解
  [(empty? abt) 0]
  [else (+ 1
           (max (height (node-left abt))
                (height (node-right abt))))]))
```

height の内部に height (再帰による繰り返し)

Literacy

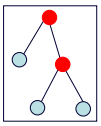
実習2．アキュムレータスタイル

二分木の高さを求める関数 height を，今度は，アキュムレータスタイル(accumulator style)で定義し，実行する



高さ： 0

empty



高さ： 2

```
(make-node
   empty
   (make-node
      empty
      empty))
```

Literacy

実習2の手順

1. 次を「定義用ウインドウ」で，実行しなさい
   - 入力した後に，Run ボタンを押す

```
(define-struct node
          (left right))
;; height : tree  ->  number
;; measure the height of a tree abt0
(define (height abt0)
 (local (;;accum represents how many nodes height-a
         ;;has encountered on its way to abt from abt0
         (define (height-a abt accum)
          (cond
            [(empty? abt) accum]
            [else (max (height-a (node-left abt)
                                 (+ accum 1))
                       (height-a (node-right abt)
                                 (+ accum 1)))]))))
   (height-a abt0 0)))
```

Literacy

実習2の手順

2. その後，次を「実行用ウインドウ」で実行しなさい

```
(height empty)
(height (make-node
      empty
      (make-node
         empty
         empty)))
```

Literacy

実習2の実行例

Literacy

二分木の高さ

1. 入力 abt が empty ならば： → 終了条件
   現在のアキュムレータの値 → 自明な解
2. そうで無ければ：
   – 「アキュムレータの値」に1を加えて，右の部分木、左の部分木の高さを求める．
   * アキュムレータには，現在計算中の木の高さが入る

Literacy

height 関数

```
;; height : tree  ->  number
;; measure the height of a tree abt
(define (height abt0)
 (local (;;accum represents how many nodes height-a
         ;;has encountered on its way to abt from abt0
        (define (height-a abt accum)
         (cond   終了条件     自明な解
            [(empty? abt) accum]
            [else (max (height-a (node-left abt)
                                 (+ accum 1))
                       (height-a (node-right abt)
                                 (+ accum 1)))]))))
   (height-a abt0 0)))
```

左右の部分木の高さの大きい方を選択(max演算を使う)

Literacy

課題

Literacy

課題1．二分探索木

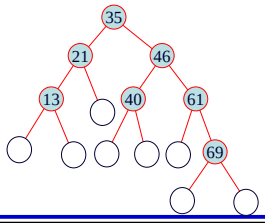
- 次ページ以降で説明する二分探索木 (binary search tree) についての問題
  – ここで説明している二分探索木は，数値属性をキーとする二分探索木である
  – 数値による探索を行う関数 search を説明している
  – この search 関数を，文字列データについて動くように書き換えなさい

Literacy

二分探索木(binary search tree)

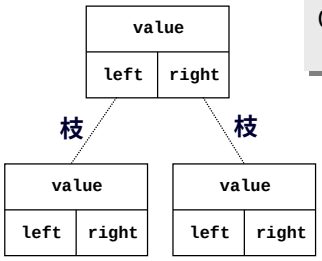
- 二分木(binary tree)の一種
- データの配置に規則あり
  - 左の部分木の節に含まれるデータは，自分より小さい
  - 右の部分木の節に含まれるデータは，自分より大きい
- データの探索のためのデータ構造

二分探索木の例



Literacy

二分探索木のための valnode 構造体




```
(define-struct valnode
   (value left right))
```

Literacy

二分探索木の節

- 二分探索木の節 (node) を，valnode 構造体として定義する

名前

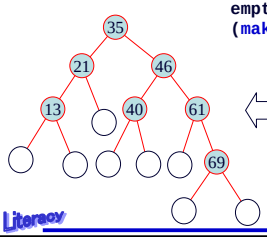
```
(define-struct valnode
        (value left right))
```

属性の並び
(それぞれの属性にも名前がある)

Literacy

二分探索木の例

```
(make-valnode 35
      (make-valnode 21
            (make-valnode 13 empty empty)
            empty)
      (make-valnode 46
            (make-valnode 40 empty empty)
            (make-valnode 61
                  empty
                  (make-valnode 69 empty empty))))
```



⇦ 上の Scheme 式が表現する二分探索木

Literacy

二分探索木による探索

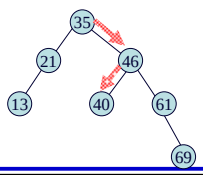
- 根(root)から始める
- 探索キーの値と，各節点のデータを比較し，目標となるデータを探す
  - 探索キーよりも節点のデータが小さいときは，右側の子をたどる
  - 探索キーよりも節点のデータが大きいときは，左側の子をたどる

Literacy

二分探索木による探索の例

(例) 40である節点を探す場合

1. 根の値(35)と，探索キー(40)を比較
2. 探索キーの方が大きいので，右側の子節点へ移る
3. 次に移った節点の値(46)と探索キー(40)を比較し
4. 探索キーの方が小さいので，左の子節点へ移る
5. 次に移った節点(40)が，目標の節点である



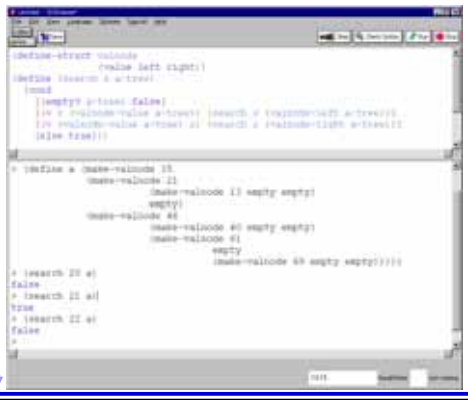
Literacy

二分探索木による探索

```
(define-struct valnode
                (value left right))
(define (search x a-tree)
  (cond
    [(empty? a-tree) false]
    [(< x (valnode-value a-tree))
     (search x (valnode-left a-tree))]
    [(< (valnode-value a-tree) x)
     (search x (valnode-right a-tree))]
    [else true]))
```

Literacy

二分探索木による探索実行例



Literacy

課題2．二分探索木

- 次ページ以降で説明するように，二分探索木の節 (node) を，今度は，次のように定義する

```
(define-struct node (sn pn left right))
```

node 型

sn	数値 (シリアル番号)
pn	シンボル
left	左の部分木
reight	右の部分木

sn をキーとする
- 左の部分木の節に含まれる sn は，自分より小さい
- 右の部分木の節に含まれる sn は，自分より大きい

Literacy

課題2．二分探索木

- 次のデータについての二分探索木を作り，関数 search-bt の実行結果を報告しなさい

63	'A
29	'B
89	'C
15	'D
77	'E
95	'F

Literacy

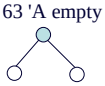
課題2での二本探索木の節の構造体定義

```
(define-struct node (sn pn left right))
```

課題2での，二分探索木の再帰的定義

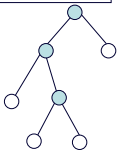
1. empty (再帰的定義の基底)
2. 部分木を持つ節を表す node 構造体
   `(make-node sn pn tleft tright)`
   tleft ･･･ 左の部分木，tright ･･･ 右の部分木
   sn ･･･ 数値，pn ･･･ シンボル

empty



(make-node 63 'A empty empty)

(make-node 63 'A (make-node 29 'B empty (make-node 89 'C empty empty)) empty)



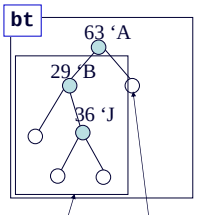
Literacy

二分探索木への節の挿入

二分探索木に節を挿入する関数add-node

```
;; add-node : node (listof number symbol) -> node
(define (add-node bt alosp)
  (cond
    [(empty? bt)
     (make-node (first alosp) (first (rest alosp))
                empty empty)]
    [else
     (cond
       [(= (node-sn bt) (first sp)) bt]
       [(> (node-sn bt) (first sp))
        (make-node  (node-sn bt) (node-pn bt)
                    (add-node (node-left bt) alosp)
                    (node-right bt))]
       [else
        (make-node  (node-sn bt) (node-pn bt)
                    (node-left  bt)
                    (add-node (node-right bt) alosp))])]))
```

```
(add-node bt (list 89 'J))
```



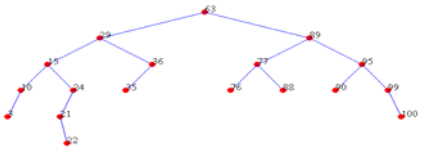

```
(add-node (node-right bt) (list 89 'J))
```

Literacy

二分探索木への節の一括挿入

```
;;create-bt : node  (lof (lof number symbol))->node
(define (create-btree bt lof-alosp)
    (cond
        [(empty? lof-alosp) bt]
        [else
           (create-btree
               (add-node bt (first lof-alosp))
               (rest lof-alosp))]))
```

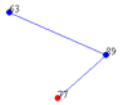
```
(define lof-alosp
 (list (list 63 'A)
       (list 29 'B)
       (list 89 'C)
       (list 15 'D)
       (list 77 'E)
       (list 95 'F)
       (list 10 'G)
       (list 24 'H)
       (list 99 'I)
       (list 36 'J)
       (list 21 'K)
       (list 22 'L)
       (list 88 'M)
       (list 90 'N)
       (list 100 'O)
       (list 76  'P)
       (list 35 'Q)
       (list 3  'R)
 ))
```



二分探索木の探索

```
;;search-bt : node number->symbol
(define (search-bt bt sn)
  (cond
    [(empty? bt) 'none]
    [else
      (cond
       [(= sn (node-sn bt))(node-pn bt)]
       [(< sn (node-sn bt))
         (search-bt (node-left bt) sn)]
       [else (search-bt (node-right bt) sn)])]))
```

```
;; test
>(define bt (create-bt empty
                       lof-alosp))
>(search-bt bt 77)
'E
```



(define lof-alosp
 (list (list 63 'A)
       (list 29 'B)
       (list 89 'C)
       (list 15 'D)
       (list 77 'E)
       (list 95 'F)
       (list 10 'G)
       (list 24 'H)
       (list 99 'I)
       (list 36 'J)
       (list 21 'K)
       (list 22 'L)
       (list 88 'M)
       (list 90 'N)
       (list 100 'O)
       (list 76 'P)
       (list 35 'Q)
       (list 3 'R)
 ))

課題3．二分探索木のグラフィックス表示

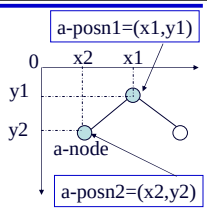
- 次ページ以降で説明する `draw-tree` 関数を使って，課題2の二分探索木をグラフィックス表示しなさい

木を描く

初期設定

```
;; set a canvas size
(define WIDTH   700) (define HEIGHT  730)
;; set disk radius and color
(define d-radius 5)  ;; a disk radius
(define d-color ‘red) ;; a disk color
(define l-color ‘blue) ;; a line color
;; set root position
(define TopMargine  30) ;; top margin
(define Root-Pos (make-posn (/ WIDTH 2)
                            (+ d-radious TopMargine)))
;;
(define bt  (create-btree empty lof-alosp)) ;; create a binary tree
(start  WIDTH   HEIGHT)                     ;; create a canvas
(draw-tree bt Root-Pos)                     ;; draw a binary tree
(stop)                                      ;; end
```

draw-tree



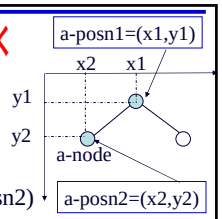

```
;; 節を描く
draw-node : node posn -> boolean
;;(define (draw-node  a-node  a-posn) …)
;;枝(branch)を描く
draw-line : node  posn  posn -> true
;;(define (draw-line  a-node  a-posn1  a-posn2) … )
節と枝を描く
;; draw-node-and-branches : node posn number -> true
;; (define (draw-node-and-branches node a-posn width) .. )
左右の節の位置を求める
;; get-next-left-posn : posn    number -> posn
;;(define (get-next-left-posn a-posn width) .. )
;; get-next-right-posn : posn    number -> posn
;;(define (get-next-right-posn a-posn width) .. )
```

節(node)を描く

```
;; draw-node : node posn -> boolean
;; draw a disk and a string at a-posn position
(define (draw-node  a-node  a-posn)
  (and
   (draw-solid-disk   a-posn d-radius d-color)
   (draw-solid-string a-posn
                      (number->string (node-sn a-node)))
  ))
;; test
(start   WIDTH   HEIGHT)
(draw-node (make-node 63 ‘A empty empty)
           (make-posn 350 20))
```

Nodeの番号を表示．その際，文字列に変換

親子間の枝(branch)を描く



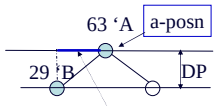

```
;; draw-line : node posn posn -> true
(define l-color ‘blue) ;; a line color
(define (draw-line a-node a-posn1 a-posn2)
 (cond
   ;; if a-node  is empty, the branch will not be drawn.
   [(empty? a-node)  true]
   [else
     (draw-solid-line a-posn1 a-posn2 l-color)]))
;;test
(draw-line (make-node 29 ‘B emtpy empty)
  (make-posn  350 20) (make-posn 120 30))
```

Literacy

左右の子節の場所計算

```
(define TopMargin 10)   ;;キャンバスの上端のマージン
(define N  (length lof-alosp)) ;;節の数
;; DP:節間の高さ
(define DP  (/ (- HEIGHT TopMargin) (- N (log N))))

;; get-next-left-posn : posn number -> posn
(define (get-next-left-posn a-posn width)
  (make-posn (- (posn-x a-posn)  width)
             (+ (posn-y a-posn) DP)))
```




```
;; get-next-right-posn : posn number number -> posn
;; (define (get-next-right-posn a-posn width)
    (make-posn (+ (posn-x a-posn) width)
               (+ (posn-y a-posn) DP)))
```

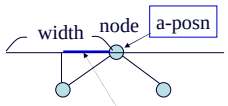
width

Literacy

節と枝を描く

```
(define Interval-Time  0.1)  ;; 0.1秒次の実行を待つ

;; draw-node-and-branches : node posn number -> true
(define (draw-node-and-branches node a-posn width)
  (and
    ;; draw node
    (draw-node node a-posn)
    (sleep-for-a-while Interval-Time)
     ;; draw left and right branches
    (draw-line (node-left node) a-posn
               (get-next-left-posn a-posn (/ width 2)))
    (sleep-for-a-while Interval-Time)
    (draw-line (node-right node) a-posn
               (get-next-right-posn a-posn (/ width 2)))      ))
```



width/2

Literacy

木を描く

```
;; draw-tree : node posn number -> true
;; draw a node and  branches and then traverse left and rigth nodes
(define (draw-tree  node a-posn width)
 (cond
   [(empty? node)   true]
   [else
    (and
      ;; draw node and edges
     (draw-node-and-branches node  a-posn  width)
      ;; traverse left and right nodes
     (draw-tree (node-left  node)
                (get-next-left-posn a-posn  (/ width 2)) (/ width 2))
     (draw-tree (node-right node)
                (get-next-right-posn a-posn (/ width 2)) (/ width 2))
    )]))
;;
(start   WIDTH   HEIGHT)
(draw-tree (create-btree empty lof-alosp) Root-Pos)
```

根(root)

キャンバスを描く

rootの位置

Root-Pos=(make-posn (/ WIDTH 2) (+ d-radious TopMargine))

Literacy