担当：長尾 智晴 (nagao@ynu.ac.jp)
濱上 知樹 (hamagami@ynu.ac.jp)

「アルゴリズムとデータ構造」講義日程

~~1. 基本的データ型~~
~~2. 基本的制御構造~~
~~3. 変数のスコープルール．関数~~
~~4. 配列を扱うアルゴリズムの基礎(1)．最大値，最小値~~
~~5. 配列を扱うアルゴリズムの基礎(2)．重複除去，集合演算，ポインタ~~
~~6. ファイルの扱い~~
~~7. 整列(1)．単純挿入整列・単純選択整列・単純交換整列~~
~~8. 整列(2)．マージ整列・クイック整列~~
~~9. 再帰的アルゴリズムの基礎．再帰におけるスコープ．ハノイの塔など．~~
~~10. バックトラックアルゴリズム．8 王妃問題など．~~
~~11. 線形リストを扱うアルゴリズム~~
~~12. 木構造を扱うアルゴリズム(1) 基礎~~
13. 木構造を扱うアルゴリズム(2) 挿入，削除，バランスなど． ← 本日の内容
14. ハッシング
15. その他のアルゴリズム

※ 試験期間中に期末試験が行われるので必ず参加しましょう．欠席の場合は「単位放棄」とみなされます．

第 13 回「動的データ構造 ～木構造２～ 」

第 12 回「動的データ構造～木構造１～」の要約・復習

☆ 線形リストより複雑な構造

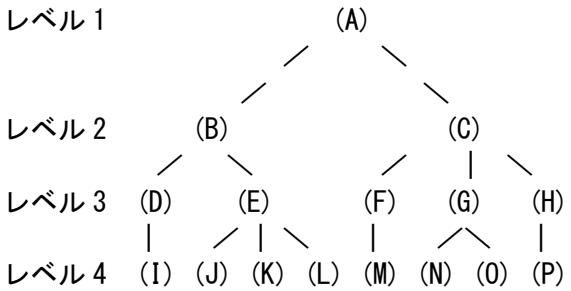
・算術式の構造，さまざまな項目の分類（の構造）（図書の分類，生物学上の分類，計算機のディレクトリ構造）

☆ 木構造（tree）

◎ 定義

・空構造（「空木」）は木構造である．一つの項目 = **節点**と有限個の木構造（部分木という）からなる構造も木構造である．

◎ グラフ表現



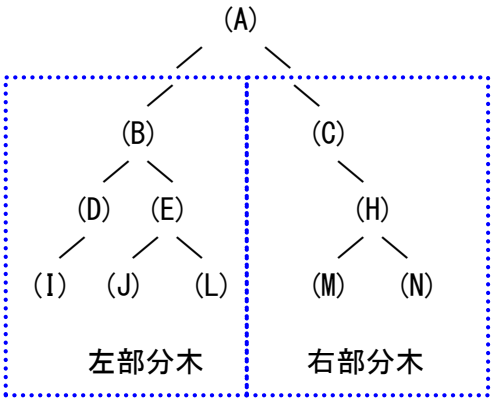
← 「根」(root)：最上位の節点
← 「枝」

(C)は(H)の(直接の)「祖先」(母親とも)
(H)は(C)の(直接の)「子孫」(娘とも)
(I)～(P)：「端点」，「葉」(=子孫なし)
(A)～(H)：「内点」(=端点でない節点)
←「高さ」あるいは「深さ」=4

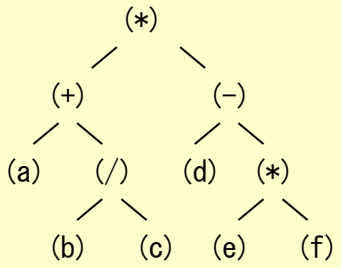
「位数」　：内点の（直接の）子孫の数
「木の位数」　：位数の最大値
「順序木」　：各節点から出る枝に順序がついている木

◎ 2 分木（binary tree）

・位数 2 の順序木



例）(a+b/c)*(d-e*f)の木表現



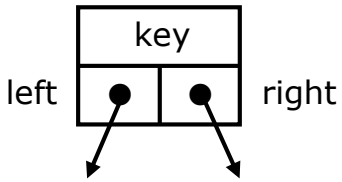
☆ 木のデータ構造

◎ データ構造

```
struct tree {
   int key;
        …
   struct tree *left;
   struct tree *right;
}
```

← 識別キー（なくてもよい）
← 一つの節点が保持するデータ
← 左部分木の根へのポインタ
← 右部分木の根へのポインタ

◎ 図による表記



☆ 木の生成(完全バランス木)

◎ 完全にバランス木：各々の節点において，左部分木中の節点数と右部分木中の節点数が高々1 つしか違わない

◎ 前回プログラム：struct tree *tree(int n)：n 個の節点を持つ完全バランス木を生成

☆ 木の走査(先順，中順，後順)

◎ 木の走査（tree traversal）

・与えられた操作を木の各節点に施す.

◎ 木構造から自然に生まれる 3 つの基本順序

・先順（preorder）　：根（操作）　→ 左部分木　→ 右部分木
・中順（inorder）　：左部分木　→ 根（操作）　→ 右部分木
・後順（postorder）　：左部分木　→ 右部分木　→ 根（操作）

◎ プログラム

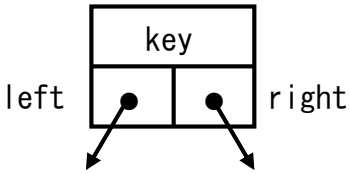
```
void preorder(struct tree *t)      ：ポインタ t で指し示される木を先順で走査
void inorder(struct tree *t)       ：ポインタ t で指し示される木を中順で走査
void postorder(struct tree *t)     ：ポインタ t で指し示される木を後順で走査
```

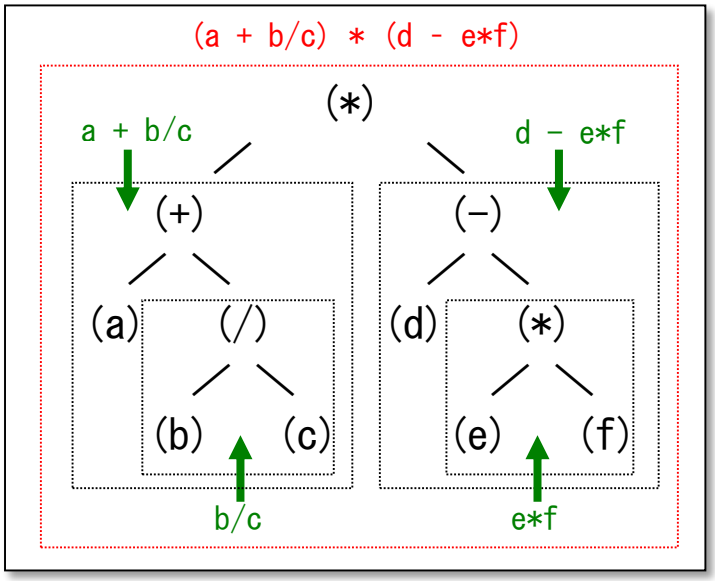
第 12 回の復習の終わり

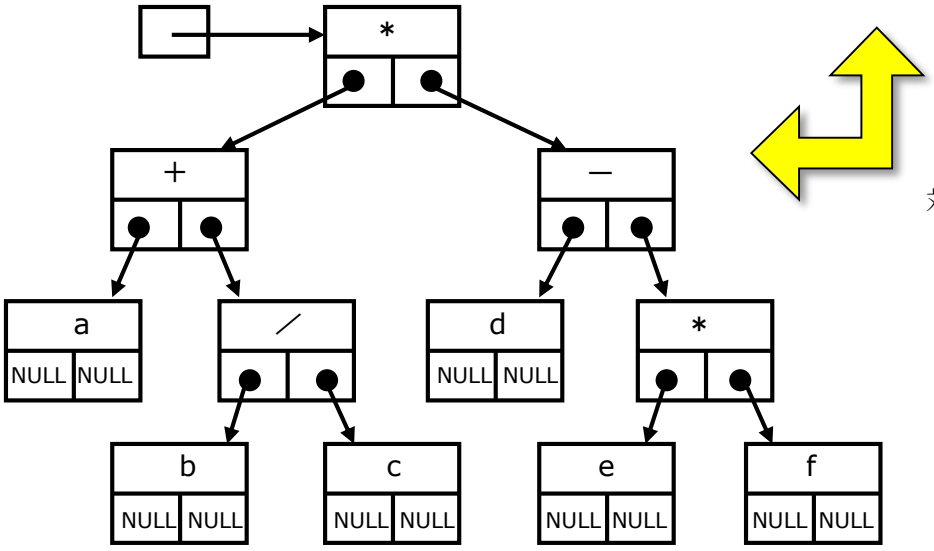
第 13 回「動的データ構造　～木構造２～」のはじまり

☆ 木のデータ構造(復習)

◎ 図による表記







対応している

※ NULL を 「空木」とする.

☆ 木の探索と節点挿入

◎ 探索木

・任意の節点 N について，以下の条件を満たしている木

1)（N の）左部分木中の全ての節点のキーが，N のキーよりも小さい

2)（N の）右部分木中の全ての節点のキーが，N のキーよりも大きい

・木が完全にバランスしていれば高さは log n → log n の比較で探索可能

（cf. 線形リストでは平均 n/2）

◎ 挿入を伴う木の探索

・あるキーを木の中で探索する時に，

あればそのキーをもつ節点の情報を得る．なければ新しい節点を木に付加する．

・空の木から出発

◎ プログラム

struct tree *search(int x,　struct tree *t)

・すでにある木（の節点）から x と同じキーを持つ節点を見つけ出す．

あればその節点の count を 1 増やす．

なければ木に（x をキーに持ち count が 1 である）新しい節点を挿入．

・返り値が tree 構造体へのポインタであることに注意．新しく節点が作成されるので呼出側（main など）で変数の値を変更する必要があるため．

☆ 節点の削除

◎ 削除

・端点，一つの子孫しか持たない節点の削除は簡単．

・難しいのは 2 つの子孫（部分木）を持つ節点の削除

◎ 2 つの子孫を持つ節点の削除

削除すべき節点を，

a) 左部分木の最右要素で置き換える，　もしくは，

b) 右部分木の最左要素で置き換える

◎ プログラム

struct tree *delete(int x, struct tree *t)

・すでにある木(の節点)から x と同じキーを持つ節点を見つけ出す．

存在すればその節点を削除する．

返り値が tree 構造体へのポインタであるのは，search( )の場合と同様で，呼出側（main など）で変数の値を変更する必要があるためである．

struct tree *del(struct tree *dstt, struct tree *t)

・第二引数に与えられた木の最右節点を探しだし，その内容を第一引数の指し示す節点にコピーする．コピーされた元の節点は必要なくなるので，これへのポインタを q に保持し，切り離す．切り離された部分木を返す．

```c
/*************************************************************
      「アルゴリズムとデータ構造」
      サンプルプログラム treeop2.c
      <<動的データ構造の例：木の探索および節点挿入，節点削除>>
      copyright (c)   T.Mori <mori@forest.dnj.ynu.ac.jp>
*************************************************************/
#include <stdio.h>
#include <stdlib.h>

/* 木構造のために再帰的データ型の定義.
   ある項目が何回出現したかを表す count も加える */
struct tree {
     int key;
     int count;
     struct tree *left, *right;
};

struct tree *search( int x,   struct tree *t );
void print_tree( struct tree *t,   int h );
struct tree *delete( int x,   struct tree *t );
struct tree *del( struct tree *dstt,   struct tree *t );

#define EOD -1
/* 初期データリスト */
int a[ ] = { 7, 2, 9, 1, 6, 9, 8, 10, 4, 1, 2, 7, 3, 5, EOD };

int get_data( void );

int main(void)
{
     struct tree *root;
     int y;

     root = NULL;
     while( ( y=get_data( ) ) != EOD )
          root = search( y, root );
             /* 値 y をキーにもつ節点を木 root の中において探索/挿入する.
              search( )の中で木が新しく生成され，木の根が変化するので，
              新しい木の根を関数の値として返し，その値を root に代入する．*/
     printf("最初の木構造:¥n");
     print_tree( root, 0 );
     printf("delete( 7, root )¥n");
          /* 値 7 をキーに持つ節点を見つけ，探索木の性質を保ったまま削除する.
           delete( )の中で木の根が変化するので，
           新しい木の根を関数の値として返し，その値を root に代入する．  */
     root = delete( 7, root );
     print_tree( root, 0 );
     printf("delete( 7, root )¥n");
     root = delete( 7, root );
```

関数のプロトタイプ宣言

関数のプロトタイプ宣言

main

```
                    /* もう一度削除．既に節点はないので削除できないはず．*/
    printf("delete( 4, root )¥n");
    root = delete( 4, root );    print_tree( root, 0 );
    return 0;
}

/* すでにある木(の節点)から x と同じキーを持つ節点を見つけ出す．
   存在すればその節点の count を 1 増やす．
   なければ木に(x をキーに持ち count が 1 である)新しい節点を挿入．
   新しく節点が作成されるので，関数の戻り値として，新しい木の根を返す．  */
struct tree * search( int x,  struct tree *t )
{
    if ( t == NULL ) {
        /* NULL ならば見つからなかったことになるので，新しく節点生成 */
        t = (struct tree *)malloc(sizeof(struct tree));
        t->key = x;    t->count = 1;
        t->left = NULL;    t->right = NULL;
    }
    else  if ( x  <  t->key )
        t->left = search( x, t->left );
        /* 現在の節点のキーよりも小さいので左部分木を探索 */
    else  if ( x  >  t->key )
        t->right = search( x, t->right );
        /* 現在の節点のキーよりも大きいので右部分木を探索 */
    else
        (t->count)++;      /* 見つかったのでカウントを増やす */
    return( t );
}

/* すでにある木(の節点)から x と同じキーを持つ節点を見つけ出す．
   存在すればその節点を削除する．
   search( )の場合と同様で木の根が削除されることもあるので，
  新しい木の根を返す．  */
struct tree * delete( int x, struct tree *t )
{
    struct tree *q;

    if ( t == NULL )    /* NULL なら見つからなかったことになる */
        printf("キー<%d>は見つかりませんでした．¥n", x);
    else if ( x < t->key )
        t->left = delete(x, t->left);
        /* 現在の節点のキーよりも小さいので左部分木を探索 */
    else if ( x > t->key )
        t->right = delete( x, t->right );
        /* 現在の節点のキーよりも大きいので右部分木を探索 */
    else {     /* 見つかった．この木の根を削除する */
        printf("キー<%d>が見つかりました．削除します．¥n", x);
        if ( t->right == NULL ) {
```

search

delete

```
98              q = t;
99              t = t->left;
100             /* 左部分木だけなら，この木自身を左部分木に置き換える */
101             free( q );        /* 領域解放 */
102         }   else   if ( t->left == NULL ) {
103             q = t;
104             t = t->right;
105             /* 右部分木だけなら，この木自身を右部分木に置き換える */
106             free( q );        /* 領域解放 */
107         }   else {            /* 左右の部分木があるならば */
108             t->left = del( t, t->left );
109             /* 左部分木の最右節点をこの木の根に移動する. */
110         }
111     }
112     return( t );
113 }
114
115 /* 第二引数に与えられた木の最右節点を探しだし，その内容を第一引数の
116    指し示す節点にコピーする.
117    コピーされた元の節点は必要なくなるので，これを切り離す.   */
118 struct tree * del( struct tree *dstt,   struct tree *t )
119 {
120     struct tree *q;
121
122     if ( t->right != NULL )
123         t->right = del( dstt, t->right );
124         /* 最右節点でなければ，右部分木を調べる */
125     else {      /* 最右節点なので，*/
126         dstt->key = t->key;   dstt->count = t->count;
127         /* 削除すべき節点にこの節点の内容をコピー */
128         q = t;
129         /* コピー後はこの節点が不要になるのでこれへのポインタを q に保持*/
130         t = t->left;    /* 左部分木を繰り上げる */
131         free( q );    /* 領域解放 */
132     }
133     return( t );
134 }
135
136 /* 木を印刷する.   */
137 void print_tree( struct tree *t,   int h )
138 {
139     int i;
140
141     if ( t != NULL ) {
142         print_tree( t->right, h+1 );   /* 右部分木を印刷 */
143         for( i=0; i<h; i++ )
144         /* 木の深さの分だけ TAB(8 文字字下げ)を印刷 */
145         printf("¥t");
```

del

print_tree

```
146             printf("<%d,%d>¥n",t->key,t->count);
147             /* その節点のキーと出現回数を印刷 */
148             print_tree( t->left, h+1 );   /* 左部分木を印刷 */
149         }
150     }
151
152     /* データ取得 */
153     int get_data( void )
154     {
155         static int i = 0;
156
157         return a[ i++ ];
158     }
159
```

get_data

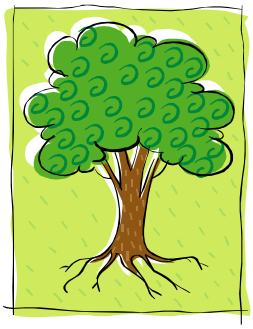
【実行結果】(木構造の印字は，時計方向に 90 度回転させて見る)

最初の木構造:

```
                    <10,1>
          <9,2>
                    <8,1>
<7,2>
                    <6,1>
                                        <5,1>
                              <4,1>
                                        <3,1>
          <2,2>
                    <1,2>
```

点線は分かり易くするために加えたものです．本当は表示されません．

```
delete( 7, root )
キー<7>が見つかりました．削除します．
                    <10,1>
          <9,2>
                    <8,1>
<6,1>
                              <5,1>
                    <4,1>
                              <3,1>
          <2,2>
                    <1,2>
delete( 7, root )
キー<7>は見つかりませんでした．
```



「アルゴリズムとデータ構造」はあと１回で終わりです．
では，次週「ハッシュ」（最終回）でお会いしましょう！