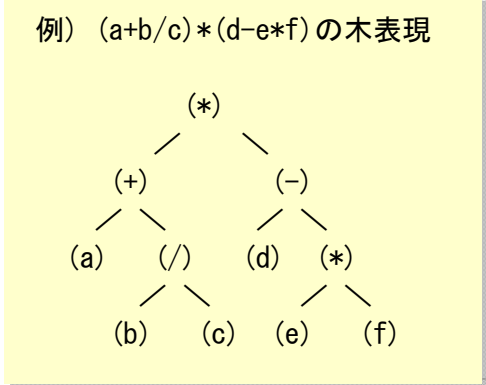
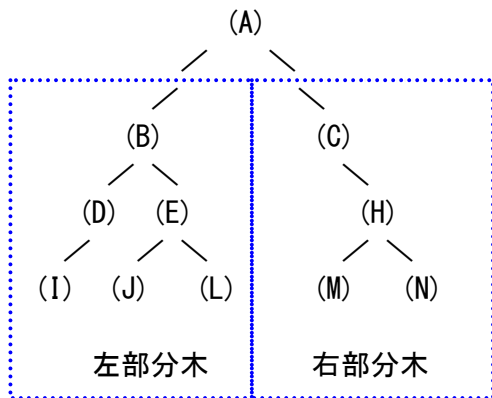


◎ 2 分木 (binary tree)

・位数 2 の順序木



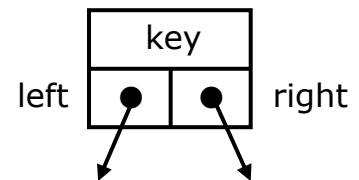
☆ 木のデータ構造

◎ 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つしか変わらない

◎ 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
```

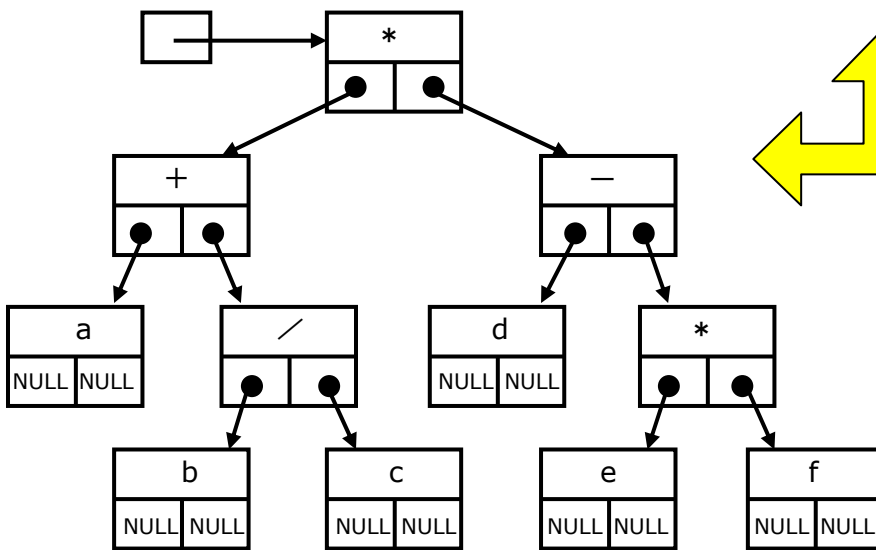
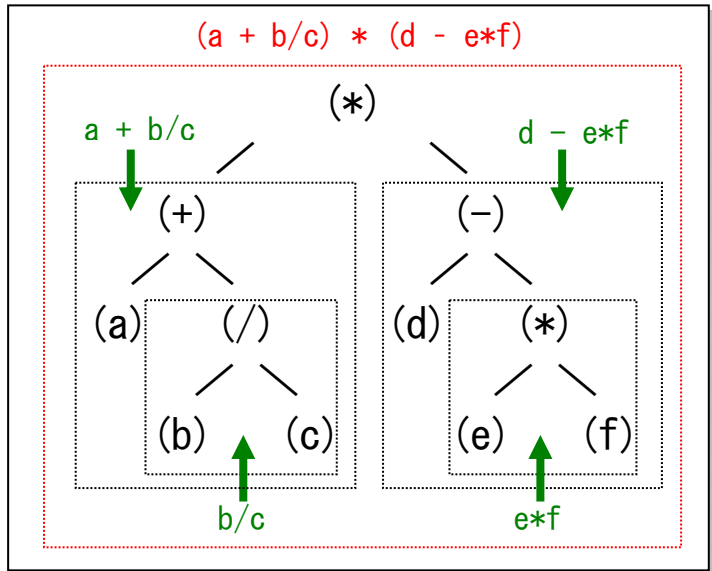
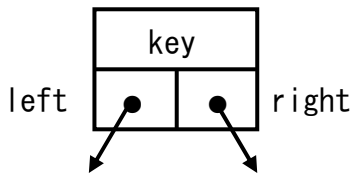


第 12 回の復習の終わり

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 $\log n \rightarrow \log n$ 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. 新しく節点を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。



```

1  /*****
2     「アルゴリズムとデータ構造」
3     サンプルプログラム treeop2.c
4     <<動的データ構造の例：木の探索および節点挿入，節点削除>>
5     copyright (c) T.Mori <mori@forest.dnj.ynu.ac.jp>
6  *****/
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 /* 木構造のために再帰的データ型の定義。
11     ある項目が何回出現したかを表す count も加える */
12 struct tree {
13     int key;
14     int count;
15     struct tree *left, *right;
16 };
17
18 struct tree *search( int x, struct tree *t );
19 void print_tree( struct tree *t, int h );
20 struct tree *delete( int x, struct tree *t );
21 struct tree *del( struct tree *dstt, struct tree *t );
22
23 #define EOD -1
24 /* 初期データリスト */
25 int a[ ] = { 7, 2, 9, 1, 6, 9, 8, 10, 4, 1, 2, 7, 3, 5, EOD };
26
27 int get_data( void );
28
29 int main(void)
30 {
31     struct tree *root;
32     int y;
33
34     root = NULL;
35     while( ( y=get_data( ) ) != EOD )
36         root = search( y, root );
37         /* 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。
38            search( )の中で木が新しく生成され，木の根が変化するので，
39            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。 */
40     printf("最初の木構造:¥n");
41     print_tree( root, 0 );
42     printf("delete( 7, root )¥n");
43     /* 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。
44        delete( )の中で木の根が変化するので，
45        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。 */
46     root = delete( 7, root );
47     print_tree( root, 0 );
48     printf("delete( 7, root )¥n");
49     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

```

50         /* もう一度削除. 既に節点はないので削除できないはず. */
51     printf("delete( 4, root )\n");
52     root = delete( 4, root );    print_tree( root, 0 );
53     return 0;
54 }

```

```

55
56 /* 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.
57 存在すればその節点の count を 1 増やす.
58 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.
59 新しく節点を作成されるので, 関数の戻り値として, 新しい木の根を返す. */
60 struct tree * search( int x,  struct tree *t )
61 {
62     if ( t == NULL ) {
63         /* 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 */
64         t = (struct tree *)malloc(sizeof(struct tree));
65         t->key = x;    t->count = 1;
66         t->left = NULL;    t->right = NULL;
67     }
68     else if ( x < t->key )
69         t->left = search( x, t->left );
70         /* 現在の節点のキーよりも小さいので左部分木を探索 */
71     else if ( x > t->key )
72         t->right = search( x, t->right );
73         /* 現在の節点のキーよりも大きいので右部分木を探索 */
74     else
75         (t->count)++;    /* 見つかったのでカウントを増やす */
76     return( t );
77 }

```

search

```

78
79 /* 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.
80 存在すればその節点を削除する.
81 search( )の場合と同様で木の根が削除されることもあるので,
82 新しい木の根を返す. */
83 struct tree * delete( int x, struct tree *t )
84 {
85     struct tree *q;
86
87     if ( t == NULL ) /* NULL なら見つからなかったことになる */
88         printf("キー<%d>は見つかりませんでした. \n", x);
89     else if ( x < t->key )
90         t->left = delete(x, t->left);
91         /* 現在の節点のキーよりも小さいので左部分木を探索 */
92     else if ( x > t->key )
93         t->right = delete( x, t->right );
94         /* 現在の節点のキーよりも大きいので右部分木を探索 */
95     else { /* 見つかった. この木の根を削除する */
96         printf("キー<%d>が見つかりました. 削除します. \n", x);
97         if ( t->right == NULL ) {

```

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 }

```

/* 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 }

```

del

/* 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");

```

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 }

```

```

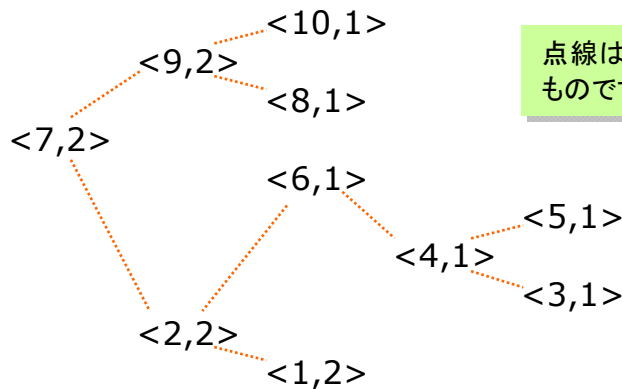
151
152 /* データ取得 */
153 int get_data( void )
154 {
155     static int i = 0;
156
157     return a[ i++ ];
158 }

```

get_data

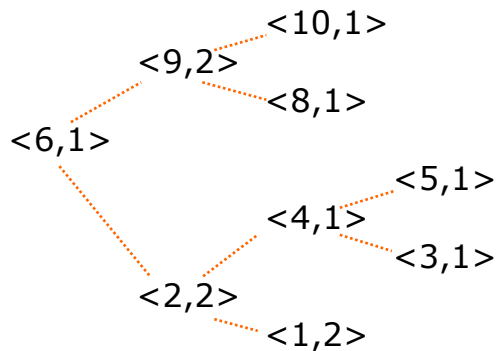
【実行結果】 (木構造の印字は、時計方向に 90 度回転させて見る)

最初の木構造:

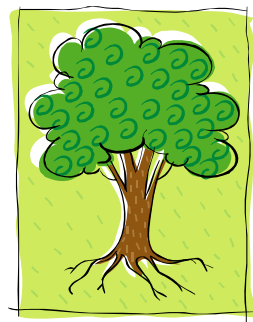


点線は分かり易くするために加えた
ものです。本当は表示されません。

delete(7,&root)
キー<7>が見つかりました。削除します。



delete(7,&root)
キー<7>は見つかりませんでした。



次週「ハッシュ」の前に宿題があります。
「ハッシュ関数」で席決めをしますので、事前に準備しておいてください!

