ツリー構造 (その 2)

山本昌志*

2006 年 1 月 23 日

1　本日の学習内容

1.1　前回の復習

前回はツリー構造を学習した．これは，樹形図のように階層構造を持つデータ構造であった．ノードの位置は親子関係により示される．特に，2 分木と呼ばれる図 1 のデータ構造は重要である．以下のことをしっかり理解する必要がある．

- データの追加方法
- データの削除方法
- データのサーチ (探索) 方法

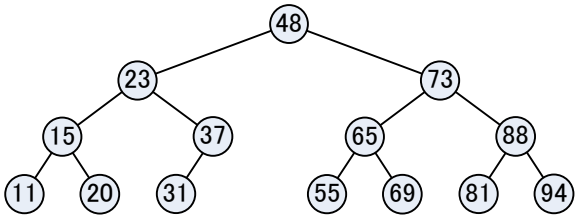

図 1: ツリー構造 (2 分木)

1.2　本日の学習内容

前回の講義でツリー構造の概要は分かったと思う．本日は，ツリー構造を C 言語で実装する方法を教科書 [1] のプログラムを例にして，説明する．

*独立行政法人　秋田工業高等専門学校　電気情報工学科

2　C 言語でのツリー構造

2.1　ツリー構造

C 言語でツリー構造を実装する方法を教科書 [1] の List 6-1(p.169)〜List 6-4(p.176-180) のプログラムをつかって説明する．このプログラムのツリー構造は，図 2 のようになっている．このデータ構造を C 言語で取り扱うことにする．そのためには，以下のようなことが必要である．

- ノードを表す構造体の定義
- ノードの追加 (ノードの作成を含む)
- ノードのサーチ (探索)
- ノードの削除 (メモリーの解放を含む)

これらのことを，説明する．

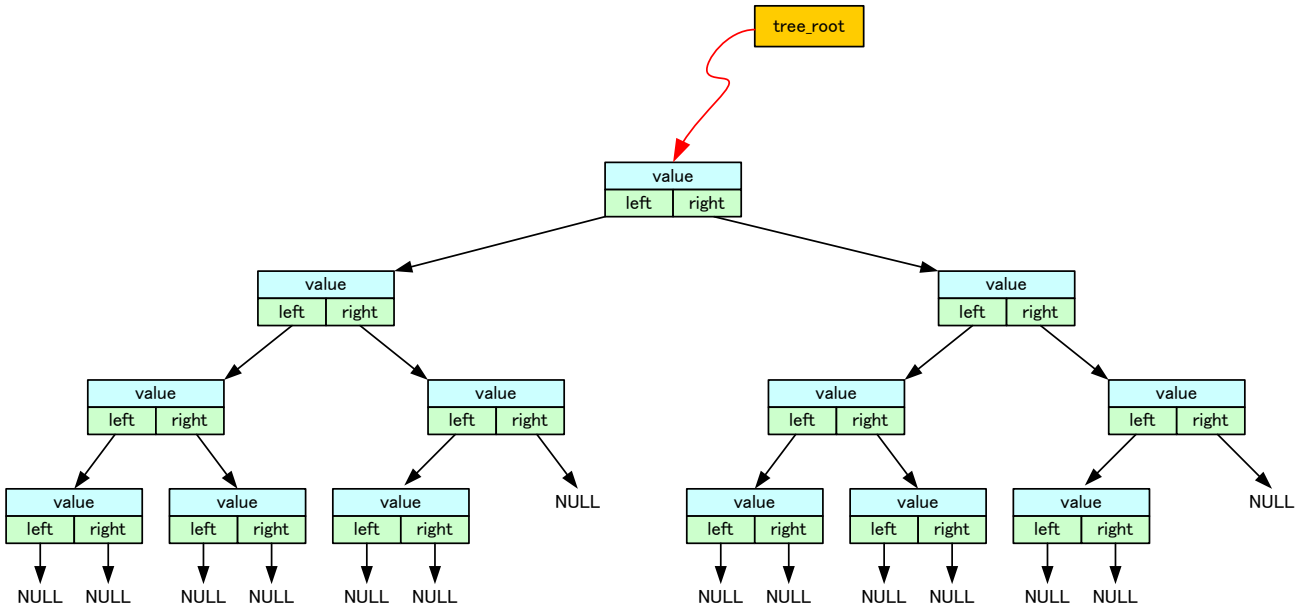

図 2: 教科書 List6-1〜List 6-4 のツリー構造

2.1.1　ノードを表す構造体

図 2 が 2 分木のツリー構造である．一つのノードは，データとその 2 つの子を表すポインターからなる．このように複数の異なる型のデータから成る情報を表すためには，構造体を用いる．その構造体をリスト 1 に示す．

このツリー構造を使うためには，型宣言として以下のものが必ず必要である．

ノードを表す構造体 データと子を示すポインターから構成され，ノードを表す．

ルートを表すポインター ツリー構造のデータをたどるために，ルートを示すポインターが必ず必要である．

リスト 1: 2 分木のノードを表す構造体 (教科書 List 6-1)

```
1 typedef struct _tag_tree_node
2 {
3     /* このノードが保持する値 */
4     int value;
5     /* 自分より小さい値の node：図では左側のノード */
6     struct _tag_tree_node *left;
7     /* 自分より大きい値の node：図では右側のノード */
8     struct _tag_tree_node *right;
9 } tree_node;
```

ツリーはノードのみならず，ルートを表すポインターが必要である．教科書の List 6-4(p.176) のように，

```
tree_node *tree_root=NULL;
```

とそのポインターを宣言しておく．もちろん，これはノードを表す構造体のよりも後で宣言する必要がある．先に宣言すると，tree_node という型がコンパイラーが分からないからである．また，初期値は意味のないポインター (NULL) を入れておく．最初はルートがないため，それを表すポインターは意味が無いためである．

2.1.2 ノードの作成

データである整数値を引数に取るノード生成の関数は，リスト 2 の通りである．この関数の動作は，以下の通りである．

1. malloc 関数でノードを表す構造体 (型は tree_node) ひとつ分のメモリーを確保する．そして，確保されたメモリーの先頭アドレスを tree_new というポインターに代入している．
2. メモリーの確保に失敗すると，tree_new は NULL ポインターとなる．確保に失敗した場合は，exit 関数によりプログラムを終了する．
3. メモリーの確保に成功したならば，ノードを表す構造体のメンバーに初期値を与える．
   - 左側の子を示すポインター left には，NULL ポインターを代入する．これは，まだ子が未定なのでこのポインターは意味がないと言うことを明記している．
   - 右側の子を示すポインター left も，NULL ポインターを代入する．
   - メンバー value には，ノードの値 num を代入する．

リスト 2: 2 分木のノードの作成 (教科書 List 6-2 の一部)

```
1 tree_node* create_new_node(int num)
2 {
3     tree_node *tree_new;
4
```

```
5       /* 新しいnodeを作成して，初期化する */
6       tree_new=(tree_node*)malloc(sizeof(tree_node));
7       if(tree_new==NULL)
8           exit(EXIT_FAILURE);
9       tree_new->left=NULL;
10      tree_new->right=NULL;
11      tree_new->value=num;
12
13      return tree_new;
14  }
```

2.1.3 ノードの追加

ノードを追加する関数は，リスト 3 の通りである．この関数の引数は 2 つで，

- 追加する整数データ (num)
- 追加するノードの親を表すポインター (*node)．最初に関数を呼び出すときには，ルートのポインターを与える (教科書 List 6-4 の p.180 参照)．

である．戻り値は無い (void)．

リスト 3 の動作は，以下の通りである．

1. ポインター node が何も示していない (NULL) ならば，新たにノードを作成 (creat_new_node(num)) して，メモリーの先頭アドレスをルートを示すポインター (tree_root) に代入する (4-8 行)．これが実行されるのは，node がルートを示すポインター (tree_root) の場合に限る．次に示す再帰呼び出しで呼び出された場合には node は NULL にならない．
2. node が NULL とならない場合，ノードの値 (node->value) と追加する値 (num) の大小関係により，2 つの場合に分けられる．
   - **ノードの値** (node->value) > **追加する値** (num)　　要するにノードの左側の子になる可能性がある場合である (12-18 行)．
     - もし左側の子があれば (node->left!=NULL)，左側の子を親として再帰呼び出しをする．
     - 子が無ければ (else)，新たにノードを作成して，それを左側の子にする．
   - **さもなければ (else)**　　要するにノードの右側の子になる可能性がある場合である (21-27 行)．
     - もし右側の子があれば (node->right!=NULL)，右側の子を親として再帰呼び出しをする．
     - 子が無ければ (else)，新たにノードを作成して，それを右側の子にする．

リスト 3: 2 分木のノードの追加 (教科書 List 6-2 の一部)

```
1   void insert_tree(int num,tree_node *node)
2   {
3       /* 1つも挿入されていない場合 */
4       if(node==NULL)
5       {
6           tree_root=create_new_node(num);
```

```
7           return;
8       }
9
10      /* numが現在のnodeの値よりも小さい場合 */
11      if(node->value>num)
12      {
13          if(node->left!=NULL)
14              insert_tree(num,node->left);
15          else
16              /* 左側に追加する */
17              node->left=create_new_node(num);
18      }
19      else
20      /* numが現在のnodeの値以上の場合 */
21      {
22          if(node->right!=NULL)
23              insert_tree(num,node->right);
24          else
25              /* 右側に追加する */
26              node->right=create_new_node(num);
27      }
28
29      return;
30  }
```

2.1.4 ノードのサーチ

ノードを探索する関数は，リスト 4 の通りである．この関数の引数は 2 つで，

- 探索するノードを表すポインター (`*node`)．最初に関数を呼び出すときには，ルートのポインターを与える (教科書 List 6-4 の p.180 参照)．
- 追加する整数データ (`val`)

である．戻り値は，探索で整数データ (`val`) と一致するノードがあった場合，そのノードのポインター (`node`) を返す．見つからなかった場合は，`NULL` ポインターを返す．

リスト 4 の動作は，以下の通りである．要するにルートから，大小関係を調べて，ツリーをたどって探索をしているのである．

1. **ノードの値** (`node->value`) > **探索する値** (`num`)　　要するにノードの左側の子と一致する可能性がある場合である (6-10 行)．
   - もし左側の子が無ければ (`node->left==NULL`)，一致するデータは無いので，`NULL` ポインターを返して，探索は終了．
   - 左側に子があれば，それを探索するノードとし，再帰呼び出しする..
2. **ノードの値** (`node->value`) < **探索する値** (`num`)　　要するにノードの右側の子と一致する可能性がある場合である (13-17 行)．
   - もし左側の子が無ければ (`node->right==NULL`)，一致するデータは無いので，`NULL` ポインターを返して，探索は終了．

- 右側に子があれば，それを探索するノードとし，再帰呼び出しをする．

3. 上の 2 つに当てはまらなければ，それは **ノードの値** (node->value) = **探索する値** (num) の場合である．探索のデータが見つかったので，ノードのポインターを返す．

リスト 4: 2 分木のサーチ (教科書 List 6-3)

```
1  tree_node* find_value(tree_node* node,int val)
2  /* 発見したtree_nodeのポインタを返す。ない場合はNULL */
3  {
4      /* 自分より小さい値ならば，左側 */
5      if(node->value>val)
6      {
7          if(node->left==NULL)  /* もし左側になければ，valはない */
8              return NULL;
9          return find_value(node->left,val);
10     }
11     /* 自分より大きい値ならば，右側 */
12     if(node->value<val)
13     {
14         if(node->right==NULL)
15             return NULL;       /* もし右側になければ，valはない */
16         return find_value(node->right,val);
17     }
18
19     /* 見つかれば，見つかった値を返す */
20     return node;
21 }
```

2.1.5 ノードの削除

ノードを削除する関数は，リスト 4 の通りである．この関数の引数は 1 つで，削除するデータ (val) である．戻り値は整数で，削除成功ならば 1 を，削除すべきデータが無いならば 0 を返す．

この関数の動作に先立って，使われている変数の役割を示しておく．

`node` 削除するべきか否か，調査しているノードを示すポインター．初期値は，ルート (tree_root)．

`parent_node` 調査しているノードの親ノードを示すポインター．初期値は NULL ポインター．

`left_biggest` 削除すべきノードの左側の子孫で最大の値をもつノードを示すポインター

`direction` 削除すべきノードが左側の子 (-1) か右側の子 (1) かを示す整数．初期値は 0 である．

リスト 4 の動作は，以下の通りである．

1. 削除すべきノードのサーチ (12-28 行)

- 削除すべきノードが見つかったならば，そのノードをポインター node で示す．削除すべきノードの親ノードはポインター parent_node で示す．node が左の子の場合 direction=-1 で，右側の子の場合 direction=1 である．
- 削除すべきノードがない場合，node は NULL ポインターとして，呼び出し元へ 0 を返す．

2. 削除すべきノードを削除

- 削除すべきノードの両方に子が無い，あるいは片方に子が無い場合．30 行で判断し，31-56 行で処理している．ここで，注意すべきは，direction=0 の時である．これは，削除すべきノードがルートであることを示している．
- 削除すべきノードの両方に子がある場合．58-80 行で処理している．
  - 左側の子孫の最大値を探索する (65-70 行)．
  - 削除すべきノードの値を，左側の子孫の最大値を代入している (74 行)．左側の子孫の最大値を持つノードの親ノードのポインターを付け替えている (75-78 行)．

リスト 5: 2 分木の削除 (教科書 List 6-4 の一部)

```
1  int delete_tree(int val)
2  /* valを削除する。成功すれば1，失敗すれば0を返す */
3  {
4      tree_node *node,*parent_node;
5      tree_node *left_biggest;
6      int direction;
7      node=tree_root;
8      parent_node=NULL;
9      direction=0;
10
11     /* while文で削除すべき対象を見つける(find_valueと同じ) */
12     while(node!=NULL&&node->value!=val)
13     {
14         if(node->value>val)
15         {
16             parent_node=node;
17             node=node->left;
18             direction=-1;        /* 親の左側 */
19         }
20         else
21         {
22             parent_node=node;
23             node=node->right;
24             direction=1;         /* 親の右側 */
25         }
26     }
27     if(node==NULL)  /* 見つからなかった */
28         return 0;
29
30     if(node->left==NULL||node->right==NULL)
31     {
32         /* 左か右，どちらかがNULLであった場合
33            (両方NULLの場合も含む) */
34         if(node->left==NULL)
35         {
36             /* 親のポインタを変更する */
37             if(direction==-1)
```

```
38                      parent_node->left=node->right;
39                  if(direction==1)
40                      parent_node->right=node->right;
41                  if(direction==0)
42                      tree_root=node->right;
43              }
44              else
45              {
46                  /* 親のポインタを変更する */
47                  if(direction==-1)
48                      parent_node->left=node->left;
49                  if(direction==1)
50                      parent_node->right=node->left;
51                  if(direction==0)
52                      tree_root=node->left;
53              }
54
55              free(node);
56          }
57          else
58          {
59              /* 両者ともNULLでなかった場合 */
60
61              /* nodeの左側の最も大きな値（最も右側の値）を取得する */
62              left_biggest=node->left;
63              parent_node=node;
64              direction=-1;
65              while(left_biggest->right!=NULL)
66              {
67                  parent_node=left_biggest;
68                  left_biggest=left_biggest->right;
69                  direction=1;
70              }
71
72              /* left_biggestの値をnodeに代入し，
73                 left_biggestは左側の枝を入れる */
74              node->value=left_biggest->value;
75              if(direction==-1)
76                  parent_node->left=left_biggest->left;
77              else
78                  parent_node->right=left_biggest->left;
79              free(left_biggest);
80          }
81
82          return 1;
83      }
```

2.1.6 メモリーの解放

メモリーを解放する関数は，リスト 6 の通りである．引数で渡されたポインターが示すノードとその子孫が使っているメモリーを解放している．

リスト 6: メモリーの解放 (教科書 List 6-4 の一部)

```
1   void free_tree(tree_node* node)
2   {
3       if(node==NULL)
4           return;
```

```
5       /* まず子 node のメモリを解放する */
6       free_tree(node->left);
7       free_tree(node->right);
8       /* 自分自身を解放 */
9       free(node);
10  }
```

3 多数の子を持つツリー

これには，いろいろな方法がある．あまり踏み込まないことにする．興味のある者は，自分で調べよ．

4 平衡木

4.1 計算量のオーダー

データの数が N 個の場合，サーチの計算回数を考えよう．もっともバランス良く，ツリー構造ができたとする．その深さを m とすると，

$$\sum_{n=1}^{m-1} 2^{n-1} \leq N = 1 + 2 + 4 + 8 + \cdots \leq \sum_{n=1}^{m} 2^{n-1} \tag{1}$$

$$\tag{2}$$

が成り立つ．これから，

$$2^{m-1} - 1 \leq N \leq 2^m - 1 \tag{3}$$

となる．これから，深さ m は

$$m - 1 \leq \log_2(N + 1) \leq m \tag{4}$$

を満たす整数となる．サーチの回数はツリーの深さと同程度であることが直感的に分かるだろう．従って，データ数 N が大きい場合，その計算回数 (比較回数) は $O(\log_2 N)$ となることが分かるだろう．

バランスが悪い木 (教科書 Fig 6-19 右図) の場合の，比較の回数は平均で $N/2$ 回である．従って，計算量は $O(N)$ となる．

データ量が非常に多い場合，すなわち N が非常に大きい場合，この計算量の差は顕著な者となる．したがって，バランスの良いツリー構造を作らなくてはならないことが分かるだろう．

4.2 平衡木の作成

4.2.1 AVL 木

教科書を使って，説明する．

4.2.2 B 木

教科書を使って説明する.

参考文献

[1] 紀平拓男, 春日伸弥. プログラミングの宝箱　アルゴリズムとデータ構造. ソフトバンクパブリッシング(株), 2004 年.