2020年度「プログラミング言語」配布資料 (6)

五十嵐 淳

2022年10月02日

1　2分探索木 in C (C/bst/)

いつものように永続的な2分探索木から実装しよう．

1.1　2分木の表現

2分木の構造の型定義は，Java や OCaml に比べると複雑である．まず，大枠としては，- 節点の種類 (葉か枝か) を表すデータと - 節点の情報を表すデータを構造体で組にする．

```
struct tree {
   T1 tag;  // for representing the kind of a node
   T2 dat;  // for representing the node itself
}
```

タグ名を tree，メンバを tag, dat とした．これにより struct tree 型の変数 t に対して，t.tag が節点の種類を，t.dat が節点 (葉か枝) のデータを表す表現となる．種類を表す部分の名前を tag にしているのは，データに，それが何を表すかのデータ (いわばメタデータ) を付加することを指して「タグ付け」ということから付けたが，構造体や共用体の名前を表す C 言語用語の「タグ」と紛らわしかったもしれない．

節点の種類を表すデータは (整数でもよいのだが) 列挙型を用いよう．また節点は，葉と枝を表すデータの共用体で定義する．

```
struct tree {
   enum nkind { LEAF, BRANCH } tag;
   union lf_or_br {
     T3 lf;  // for representing a leaf
     T4 br;  // for representing a branch
   } dat;
}
```

enum や union に nkind (node の kind) と lf_or_br (lf または br) という名前をつけたが，これらの型 (enum nkind 型や union lf_or_br 型) を持つ変数を使うつもりがないので，これらは実は省略可能である (実際，サンプルコードでは lf_or_br は省いた)．

ここまでの部分は OCaml で書くと，

```
type tree =
    Lf of T3
  | Br of T4
```

というような定義に対応する．OCaml の場合は `Lf` は何も情報を運ばないので，`of T3` 以下は削除，`T4` として，レコード型 `{left: tree; value: int; right: tree}` を使った．さて C コードに戻ろう．ここでは，`lf` の型 `T3` 部分は (どうせ使わないので) 何でもよいのだが，統一感から (？) 構造体を入れておくことにする．メンバも与える必要はないが，メンバを持たない構造体は許されていないので (試した限りでコンパイラは受理してくれるようだが)，ダミーのメンバを持つ構造体

```
struct leaf { int dummy; } lf;
```

とした．枝の方は，3 メンバを持つ構造体になるのだが，OCaml に倣って，

```
struct branch {
  struct tree left;
  int value;
  struct tree right;
} br;
```

とすると，コンパイルエラーになってしまう．(macOS の clang コマンドでコンパイルした場合は以下のようなメッセージが表示される．)

```
bst.c:16:19: error: field has incomplete type 'struct tree'
      struct tree left;
                  ^
```

エラーメッセージがわかりにくいが，incomplete type というのは (その型の値が占める領域の) サイズがわからない型のことである．コンパイラは，`struct tree` の定義を処理するにあたって，その構造体のサイズを計算する必要がある．しかし，定義の中に，今まさに定義しようとしている型が出てきてしまい，その型のサイズがわからない，といっているのである．

エラーメッセージとしては，型のサイズがわからない，といっているが，実は，このような定義は許しようがない．`struct tree`, `enum nkind`, `struct leaf`, `int` のサイズをそれぞれ s, e, l, i とすると，$s = e + \max(l, (s + i + s))$ を満たす必要がある (足し算が構造体，max が共用体のサイズ計算に対応している) が，これらのサイズは全て正整数なので，こんな s は存在しないわけである．要するに，ある領域の中に，すっぽりと自分自身をふたつもを含むようなレイアウトをしなければいけないのだが，そんなことは不可能なのである．

このような問題を回避して，C 言語で再帰的なデータ構造を扱うためには，ポインタを使うことになる．以下が最終的な `struct tree` の定義である．メンバ `left`, `right` はポインタになっている．

```
struct tree {
  enum nkind { LEAF, BRANCH } tag;
  union {
    struct leaf { int dummy; } lf;
    struct branch {
      struct tree *left;
      int value;
```

```
      struct tree *right;
    } br;
  } dat;  // standing for DATum
};
```

このように，枝は部分木へのポインタをメンバとするように定義する．ポインタであれば，その指す先の領域に関わらずポインタのサイズは一定であるため，先ほどのサイズを計算する式も，p をポインタのサイズとして，$s = e + max(l, (p + i + p))$ で与えることができる．

1.1.1 寄り道

先程も少し触れたように，葉を表すためのデータは何もないので，`struct leaf ... lf;` を共用体に含める必要は全くない．さらに，2 分木の場合，このメンバを削除すると，共用体のメンバ数が 1 になってしまい，共用体を使う意味すらなくなるため，

```
struct tree {
  enum nkind { LEAF, BRANCH } tag;
  struct branch {
    struct tree *left;
    int value;
    struct tree *right;
  } br;
};
```

という定義でプログラムを作ることも可能であるが，ここでは一般性を考慮して，少し煩雑な定義のまま話を進める．

1.1.2 補助関数

探索などの関数定義に進む前に，branch や leaf を作るための関数を用意しておく．これらは Java でいえばコンストラクタの定義に相当するものと考えられる．branch であれば，左右の部分木 (へのポインタ) と格納する整数を引数として，それらを格納した枝 (へのポインタ) を返す関数として定義する．

```
struct tree *newbranch(struct tree *left, int value, struct tree *right) {
  // Allocate a new object in the heap
  struct tree *n = (struct tree *)malloc(sizeof(struct tree));
  // And then initialize the members
  n->tag = BRANCH;            // could be written (*n).tag = BRANCH
  n->dat.br.left = left;
  n->dat.br.value = value;
  n->dat.br.right = right;
  return n;
}

struct tree *newleaf(void) {
```

```
  struct tree *n = (struct tree *)malloc(sizeof(struct tree));
  n->tag = LEAF;
  return n;
}
```

どちらの関数でも n を malloc で確保した領域へのポインタとして使っている．struct tree は内部に共用体さらには構造体を含んでいるので，どの部分にどのような表記でアクセスするかに注目して定義を読んでもらいたい．

- n->dat は共用体
- n->dat.br は共用体を構造体 struct branch として使うためのメンバアクセス
- さらにそれに .left などをつけることでようやく，部分木 (へのポインタ) などにアクセスできる．

1.2 探索

さて，ここまできちんと理解できれば 2 分探索木を操作する関数の定義はさほど難しくはない．以下は find の定義である．

```
bool find(struct tree *t, int n) {
  if (t->tag == LEAF) {
    return false;
  } else /* t->tag == BRANCH */ {
    struct branch b = t->dat.br;
    if (n == b.value) {
      return true;
    } else if (n < b.value) {
      return find(b.left, n);
    } else /* n > b.value */ {
      return find(b.right, n);
    }
  }
}
```

2 行目で節点の種類によって場合分けを行い，5 行目で branch の構造体を取り出し，その値とパラメータ n の大小によって場合分けを行う．

1.2.1 寄り道 (C 上級者への道)

5 行目で，

```
    struct branch b = t->dat.br;
```

としているが，これだと，変数 b の領域に構造体全体の内容がコピーされるので，メモリ効率を気にする場合はポインタ操作で済ませるのが C らしいプログラミングである．

```
    struct branch *b = &(t->dat.br);   // could be &t->dat.br
```

`&(t->dat.br)` (この括弧も不要だがわかりやすさのためにつけている) によって，`t` の領域の途中 (構造体 `branch` が始まる部分) を指すポインタが得られる．

もちろんこの場合，`b` の型が構造体ではなく構造体へのポインタになるので，以降の木の操作には．ではなく `->` を使うよう変更が必要である．

```
    struct branch *b = &(t->dat.br);
    if (n == b->value) {
      return true;
    } else if (n < b->value) {
      return find(b->left, n);
    } else /* n > b->value */ {
      return find(b->right, n);
    }
```

1.3 挿入

挿入については，あまりコメントすることがない．

```
struct tree *insert(struct tree *t, int n) {
  if (t->tag == LEAF) {
    return newbranch(newleaf(), n, newleaf());
  } else /* t->tag == BRANCH */ {
    struct branch b = t->dat.br;
    if (n == b.value) {
      return t;
    } else if (n < b.value) {
      return newbranch(insert(b.left, n), b.value, b.right);
    } else /* n > b.value */ {
      return newbranch(b.right, b.value, insert(b.right, n));
    }
  }
}
```

1.4 削除

削除についてもあまりコメントすることがない．`min` 関数中で，ライブラリの `assert` マクロ[1]が呼ばれているが，これは `min` 関数は `branch` のみに適用されるべきものなので，引数が本当に `branch` かどうかを確認している．(万が一 `branch` でなかったら，その時点で実行が終了する．)

```
int min(struct tree *t) {
  assert(t->tag == BRANCH);
```

[1] `assert` は関数に見えるが，実は関数ではなく，C 言語の **マクロ (macro)** と呼ばれる機能を使って定義されている．マクロはいわば略記であって，コンパイラ (正確にはプリプロセッサ) によって定義が展開されて，プログラムの字面上の置き換えが起こる．マクロについては，いずれ講義でも触れたい．

```
  struct branch b = t->dat.br;
  if (b.left->tag == LEAF) {
    return b.value;
  } else {
    return min(b.left);
  }
}

struct tree *delete(struct tree *t, int n) {
  if (t->tag == LEAF) {
    return t;
  } else /* t->tag == BRANCH */ {
    struct branch b = t->dat.br;
    if (n == b.value) {
      if (b.left->tag == LEAF) {
        if (b.right->tag == LEAF) {
          return newleaf();
        } else /* b.right->tag == BRANCH*/ {
          return b.right;
        }
      } else /* b.left->tag == BRANCH*/ {
        if (b.right->tag == LEAF) {
          return b.left;
        } else /* b.right->tag == BRANCH*/ {
          int m = min(b.right);
          struct tree *newRight = delete(b.right, m);
          return newbranch(b.left, m, newRight);
        }
      }
    } else if (n < b.value) {
      struct tree *newLeft = delete(b.left, n);
      return newbranch(newLeft, b.value, b.right);
    } else /* n > b.value */ {
      struct tree *newRight = delete(b.right, n);
      return newbranch(b.left, b.value, newRight);
    }
  }
}
```

1.5 葉を null ポインタで表現する (C/bstNull/)

Java の時と同様に，2 分木の表現のバリエーションとして，何の情報も運ばない `leaf` を null pointer で表現する方法を紹介する．null pointer は，何の領域も指さない特殊なポインタである．`malloc` で確保した領域も

含め，いかなる変数のアドレスとも異なる．既に紹介した通り C 言語では null pointer は NULL という定数で表す．

leaf を NULL で表すことにすると，そもそも共用体を使う必要がなくなる[2]ので，型の定義は非常に単純になる．

```
struct tree {
  struct tree *left;
  int value;
  struct tree *right;
};
```

1.5.1 leaf, branch を作る補助関数

```
struct tree *newbranch(struct tree *left,
                       int value,
                       struct tree *right) {
  // Allocate a new object in the heap
  struct tree *n = (struct tree *)malloc(sizeof(struct tree));
  // And then initialize the members
  n->left = left;
  n->value = value;
  n->right = right;
  return n;
}

struct tree *newleaf(void) {
  /* Real C programmers would avoid defining such a simple function.
   * It causes overhead of function calls.
   */
  return NULL;
}
```

newleaf は常に NULL を返す関数である．コメントにもあるように，効率重視のふつうの C プログラマであれば，こんな関数は定義しない (newleaf() を使うところを NULL で置き換える) と思うが，最初の実装方法との対応を取りやすくするために，あえて定義している．

残りの関数の定義を一気に示す．与えられた木が leaf か branch か判定するために，null pointer との比較 (t == NULL) を行っていること，struct tree の定義が簡単になった分，b を用意する必要がなくなったことなどに注意すれば，理解できるはずである．(ほとんど Java バージョンと同じであるし，むしろ，こちらの方がプログラムとしてはスッキリしているといってもよいだろう．)

```
bool find(struct tree *t, int n) {
  if (t == NULL) {
    return false;
```

[2]繰り返すが，これはこのデータ構造の特殊事情である．もしも，leaf, branch 以外にデータを構成する種類があったら共用体を使うことになる．

```
  } else /* t is a branch */ {
    if (n == t->value) {
      return true;
    } else if (n < t->value) {
      return find(t->left, n);
    } else /* n > t->value */ {
      return find(t->right, n);
    }
  }
}

struct tree *insert(struct tree *t, int n) {
  if (t == NULL) {
    return newbranch(newleaf(), n, newleaf());
  } else /* t is a branch */ {
    if (n == t->value) {
      return t;
    } else if (n < t->value) {
      return newbranch(insert(t->left, n), t->value, t->right);
    } else /* n > t->value */ {
      return newbranch(t->right, t->value, insert(t->right, n));
    }
  }
}

int min(struct tree *t) {
  assert(t != NULL);
  /* t is a branch */
  if (t->left == NULL) {
    return t->value;
  } else {
    return min(t->left);
  }
}

struct tree *delete(struct tree *t, int n) {
  if (t == NULL) {
    return t;
  } else /* t is a branch */ {
    if (n == t->value) {
      if (t->left == NULL) {
        if (t->right == NULL) {
          return newleaf();
```

```
        } else /* t->right is a branch */ {
          return t->right;
        }
      } else /* t->left is a branch */ {
        if (t->right == NULL) {
          return t->left;
        } else /* t->right is a branch */ {
          int m = min(t->right);
          struct tree *newRight = delete(t->right, m);
          return newbranch(t->left, m, newRight);
        }
      }
    } else if (n < t->value) {
      struct tree *newLeft = delete(t->left, n);
      return newbranch(newLeft, t->value, t->right);
    } else /* n > t->value */ {
      struct tree *newRight = delete(t->right, n);
      return newbranch(t->left, t->value, newRight);
    }
  }
}
```

1.6 短命な 2 分探索木 in C (C/bstMutable/)

次は, 短命な実装である. 木構造のデータ定義は (NULL を使わない) 最初と同じであるので, newbranch, newleaf なども含めて省略する. また find は木を書き換えないので, これも定義は同じであるので省略する.

insert で注意すべき点は, - leaf を branch に変化させる時には, leaf の領域が不要になるので free で解放している (3 行目) - branch を表す構造体は局所変数にコピーせずに& でポインタを取得している (6 行目, 上記「寄り道 (C 上級者への道)」も参照). コピーしてしまうと, 元の木構造を書き換えることにならないので, これは必須である.

といったあたりであろう.

```
struct tree *insert(struct tree *t, int n) {
  if (t->tag == LEAF) {
    free(t);
    return newbranch(newleaf(), n, newleaf());
  } else /* t->tag == BRANCH */ {
    struct branch *b = &(t->dat.br);
    if (n == b->value) {
      return t;
    } else if (n < b->value) {
      struct tree *newleft = insert(b->left, n);
```

```
      b->left = newleft;
      return t;
    } else /* n > b->value */ {
      struct tree *newright = insert(b->right, n);
      b->right = newright;
      return t;
    }
  }
}
```

削除についても，不要な領域の解放が注意点だろう (19-21 行目，25-26 行目，32-33 行目)．しかも，解放の順番も大事である．先に `free(t)` としてしまうと，`b` の指す領域 (これは `t` の領域の途中を指しているのであった) も解放されてしまうので，`free(b->left);` や `free(b->right);` が未定義動作を引き起こす不正な操作になってしまう．「解放する時には子供から」が大原則である．

```
int min(struct tree *t) {
  assert(t->tag == BRANCH);
  struct branch *b = &(t->dat.br);
  if (b->left->tag == LEAF) {
    return b->value;
  } else {
    return min(b->left);
  }
}

struct tree *delete(struct tree *t, int n) {
  if (t->tag == LEAF) {
    return t;
  } else /* t->tag == BRANCH */ {
    struct branch *b = &(t->dat.br);
    if (n == b->value) {
      if (b->left->tag == LEAF) {
        if (b->right->tag == LEAF) {
          free(b->left);
          free(b->right);
          free(t);
          return newleaf();
        } else /* b->right->tag == BRANCH*/ {
          struct tree *right = b->right;
          free(b->left);
          free(t);
          return right;
        }
      } else /* b->left->tag == BRANCH*/ {
```

```
        if (b->right->tag == LEAF) {
          struct tree *left = b->left;
          free(b->right);
          free(t);
          return left;
        } else /* b->right->tag == BRANCH*/ {
          int m = min(b->right);
          struct tree *newRight = delete(b->right, m);
          b->value = m;
          b->right = newRight;
          return t;
        }
      }
    } else if (n < b->value) {
      struct tree *newLeft = delete(b->left, n);
      b->left = newLeft;
      return t;
    } else /* n > b->value */ {
      struct tree *newRight = delete(b->right, n);
      b->left = newRight;
      return t;
    }
  }
}
```

メモリの解放を本当に気にするなら，main 関数を終了する前に，確保した領域を全て解放するのが行儀のよいプログラムである．以下は，与えられた木の領域を全て解放する関数 free_tree である．main (ここには掲載していない) の最後で呼んでいる．

```
void free_tree(struct tree *t) {
  if (t->tag == LEAF) {
    free(t);
  } else /* t->tag == BRANCH */ {
    struct branch *b = &(t->dat.br);
    free_tree(b->left);
    free_tree(b->right);
    free(t);
  }
  return;
}
```

1.6.1 葉を null ポインタで表現する (C/bstMutableNull/)

最後に，短命でかつ葉を NULL で表現したバージョンである．おそらく，これが C 言語で 2 分探索木を実装する場合の標準的な実装のひとつだろう．(おそらく一部の関数は再帰を使わずに繰り返しを使って定義すると，より C らしい．これについては 08 再帰と繰り返し を参照のこと．)

ほとんどの定義は，これまでの内容が理解できていればそれほど難しくはないが，挿入と削除関連のメモリ管理についてコメントする．

```
struct tree *insert(struct tree *t, int n) {
  if (t == NULL) {
    return newbranch(newleaf(), n, newleaf());
  } else /* t is a branch */ {
    if (n == t->value) {
      return t;
    } else if (n < t->value) {
      t->left = insert(t->left, n);
      return t;
    } else /* n > t->value */ {
      t->right = insert(t->right, n);
      return t;
    }
  }
}

// The definition of
//   int min(struct tree *);
// omitted

struct tree *delete(struct tree *t, int n) {
  if (t == NULL) {
    return t;
  } else /* t is a branch */ {
    if (n == t->value) {
      if (t->left == NULL) {
        if (t->right == NULL) {
          free(t);
          return newleaf();
        } else /* t->right is a branch */ {
          struct tree *right = t->right;
          free(t);
          return right;
        }
      } else /* t->left is a branch */ {
```

```
        if (t->right == NULL) {
          struct tree *left = t->left;
          free(t);
          return left;
        } else /* t->right is a branch */ {
          int m = min(t->right);
          t->value = m;
          t->right = delete(t->right, m);
          return t;
        }
      }
    } else if (n < t->value) {
      t->left = delete(t->left, n);
      return t;
    } else /* n > t->value */ {
      t->right = delete(t->right, n);
      return t;
    }
  }
}
```

まず，leaf については何の領域も確保されていないので，`free` をする必要はない (3 行目)．また，削除操作で，部分木を残したまま，それをぶら下げている branch を解放する場合には，まず，部分木へのポインタを保存しておいてから，領域の解放，保存しておいたポインタを返している (31-33 行目，37-39 行目)．これを

```
free(t);
return t->right;
```

などとすると，解放した領域へのアクセス (未定義動作) になってしまうのでまずい．