アルゴリズムとデータ構造 プログラミング演習

森 立平 mori@c.titech.ac.jp

2017 年 7 月 18 日

今週の目標

- ポインタ, malloc を使ったプログラムが書けるようになる.

今週の課題

1. 二分探索木のプログラムを書く．7/25 の授業時間に提出.

今日のワークフロー

1. 資料を読む.
2. 課題をやる.

1 ポインタ, malloc, free について

動的計画法のプログラムではあらかじめ配列の長さを固定していた．C 言語では配列の長さはコンパイル時に決まっていなければならないため, どのような入力にも対応できるように, 十分な長さの配列を確保することにしていた．もしも, 入力によって配列の長さを変えることができれば, メモリの無駄を防ぐことができる．そのために関数 malloc を使うことができる.

```
// malloc を使うために必要
#include <stdlib.h>

int main(){
// int へのポインタを宣言
  int *p;
/*
  関数 malloc(size) はメモリ領域を size バイト確保して，
  その先頭へのアドレスを返す
*/
  p = malloc(sizeof(int));
}
```

ここで p は「ポインタ」と呼ばれる, メモリアドレスを代入するための変数である．最後の行で p に malloc によって確保されたアドレスの先頭を代入した．ここで sizeof(int) とは int 型の変数が必要とする領域のバイト数である．以降 *p で値を代入したり, 読み出したり, 通常の変数と同じように扱うことができる.

```
// 2 を *p に代入
*p = 2;

// 2 が表示される
printf("%d\n", *p);

/*
  p が指しているアドレスが表示される
  通常はプログラムの中でアドレスを表示する必要はない
*/
printf("%x\n", p);
```

また, 配列のように多くの領域をいっぺんに確保することもできる.

```
int *p;

// int 3個分のメモリ領域を確保
p = malloc(sizeof(int)*3);

// 確保したメモリの先頭に 0 を書き込む
*p = 0;

// 確保したメモリの先頭から int 一つ分進めたアドレスに 1 を書き込む
*(p+1) = 1;

// 確保したメモリの先頭から int 二つ分進めたアドレスに 2 を書き込む
*(p+2) = 2;

/*
 エラー！ 確保したメモリの先頭から int 三つ分進めたアドレスに
 3 を書き込もうとしているが, ここは malloc によって確保されていないので,
 書き込めない
*/
*(p+3) = 3;
```

このように *(p+i) で p から i だけ進んだアドレスを参照して書き込んだり, 読み出したりできる. また *(p+i) の代わりに p[i] でも同じ意味になる. 逆に配列をポインタのように扱うこともできる.

```
int *p;
int A[100];

// A[4] = 2 と同じ意味
*(A+4) = 2;

// p を A[10] を指すようにする
p = A+10;

// A[11] に 5 を代入
p[1] = 5;

// エラー！Aに代入することはできない
A = p;
```

メモリは有限なのでいくらでも大きな領域が確保できるわけではない. もしも, malloc がメモリの確保に失敗すると NULL が返される.

```
int *p;
// すごく大きなメモリ領域を確保しようとするが
// 確保できず NULL が p に代入される
p = malloc(sizeof(int)*1000000000);

// エラー！ p が NULL なのに参照しようとしている
*p = 3;
```

ここで NULL とは具体的には 0 のことである. 0 は無効なアドレスなので参照できない. C 言語では無効なアドレスを表現する方法として 0 を用いるのだが, 無効なアドレスとして 0 を使っていることが分かるように NULL と書くことにする.

```
int *p;

p = malloc(sizeof(int)*1000000000);
```

```
// メモリが確保できたかどうかチェック
if(p == NULL){
  printf("メモリ確保できなかった\n");
  return 1;
}
```

本当はこのように malloc を使ったら毎回, ポインタの値が NULL かどうかチェックする必要がある.

malloc で確保したメモリ領域はプログラムの中のどこでも有効である.

```
struct list {
  int x;
  struct list *next;
};

struct list *create_singleton(int n){
  struct list *p = malloc(sizeof(struct list));
  p->x = n;
  p->next = NULL;
  return p;
}

int main(){
  struct list *p = create_singleton(10);
  struct list *q = create_singleton(100);
  q->next = p;
}
```

このように関数 create_singleton の中で確保したメモリ領域のアドレスを main 関数の中で使うことができる. ここで p->x は (*p).x と同じ意味である. このように必要になったら malloc を使ってメモリを確保すれば, メモリの許す限りいくらでもメモリ領域を使うことができる. 一方で必要なくなったメモリを解法することもできる.

```
int *p;
p = malloc(sizeof(int)*10000);

// p を使って計算する
f(p);

// p はもう必要ないので解放する
free(p);

// エラー！もう p は使えない
*p = 2;
```

このように関数 free を使うことで必要なくなったメモリを解放することができる. free に渡すアドレスは必ず以前 malloc で受けとったものでなくてはならない. 上記で free(p+10) などとしてはいけない. malloc を使い続けるとメモリを食いつぶしていって, いつか利用可能なメモリが無くなってしまうが, 必要なくなったメモリを free で解放してあげることで, 使い終ったメモリを再利用できる. このように C 言語ではプログラマがメモリを管理しなくてはならない. そのような言語は C と C++ くらいで, その他のプログラミング言語では, 使用されていないアドレスを自動的に検出して解放するガベージコレクション (GC) と呼ばれる高度な機能が備わっている. しかしそのような場合でも, 本当はプログラミング言語がメモリ管理を請け負っているだけで, 内部では malloc と free でメモリ管理をしていることを知っている必要がある.

2 課題: 二分探索木の実装

二分探索木のノードを構造体で表すことにする.

```
struct tree {
  struct tree *l;
  int v;
  struct tree *r;
  int h;
};

// 以降 struct tree の代わりに tree_t と書ける
typedef struct tree tree_t;
```

毎回 struct tree と書くのは面倒なので typedef を使って tree_t という型名を与えることにする．ここで l は左のノード, r は右のノード, v はそのノードが持つ値, h はそのノードの高さとする（高さは発展的課題で用いる）．「ノードの高さ」は左のノードの高さ（無い場合は 0）と右のノードの高さ（無い場合は 0）の最大値 $+1$ と定義する．この構造体では左及び右のノードが無い場合は NULL が代入されているとする．以下の説明ではノードとそのノードを根とする二分探索木を同一視することにする．また, ポインタ tree_t *t と二分探索木を同一視することにする．

二分探索木に関して以下の関数を実装したい．

- int tree_height(tree_t *t): 二分探索木 t の高さを返す関数．t が NULL のときは 0 を返す．
- tree_t *tree_create(tree_t *l, int v, tree_t *r): 左のノードが l で右のノードが r で値が v の二分探索木を作って二分探索木へのポインタを返す関数 (malloc が必要)．二分探索木 l が持つ値はすべて v 未満, 二分探索木 r が持つ値はすべて v より大きいとする．
- void tree_print_elements(tree_t *t): 二分探索木 t に含まれている値を小さい順に printf で表示する関数．
- int tree_mem(int x, tree_t *t): 二分探索木 t 全体の中に値 x があれば 1 なければ 0 を返す関数．
- tree_t *tree_add(int x, tree_t *t): 二分探索木 t に値 x を加えた二分探索木を返す関数．x が既に存在する場合は t と等価な二分探索木を返す．
- int tree_min(tree_t *t): 二分探索木 t に含まれる一番小さい値を返す関数．t が NULL の場合の挙動は未定義．
- int tree_max(tree_t *t): 二分探索木 t に含まれる一番大きい値を返す関数．t が NULL の場合の挙動は未定義．
- tree_t *tree_remove_min(tree_t *t): 二分探索木 t に含まれる一番小さい値以外の値を持つ二分探索木を返す関数．t が NULL の場合の挙動は未定義．
- tree_t *tree_merge(tree_t *t1, tree_t *t2): 二分探索木 t1, t2 に含まれる値をすべてを持つ二分探索木を返す関数．ただし二分探索木 t1 に含まれる値は二分探索木 t2 に含まれる値より小さいとする．
- tree_t *tree_remove(int x, tree_t *t): 二分探索木 t から値 x を除いた二分探索木を返す関数．x が存在しない場合は t と「等しい」二分探索木を返す．

以下は注意点とルールである．

- tree_t * 型の引数は常に NULL の可能性があることに注意すること．
- 一度 tree_create で作ったノードの持つ変数 (l, v, r, h) の値は**二度と書き換えてはならない**．必要があれば tree_create で新しくノードを作ることにする．
- malloc は tree_create の中だけで使うこと．

以下課題である.

- 上記の関数の関数定義のプログラムを書け.

3 発展的課題: 回転による平衡性の維持

二分探索木はできるだけ平衡しているとよい. 値を追加するときに単純にノードを二分探索木に追加すると最悪の場合とても深い木になってしまう (値が昇順に追加される場合を考えるといい). もしも, 二分探索木に含まれるすべてのノードが「左の子供の高さと右の子供の高さの差が 1 以内」という性質を満たしているとき, その二分探索木は「平衡」していると呼ぶことにする. 平衡二分探索木に値を追加, 削除したときに平衡でなくなってしまうことがあるため, 修正して平衡性を保つ必要がある.

- `tree_t *tree_bal(tree_t *l, int v, tree_t *r)`: 「平衡二分探索木 `l` に含まれるすべての値, `v`, 平衡二分探索木 `r` に含まれるすべての値」をすべて含む平衡二分探索木を返す関数. ただし, `l`, `v`, `r` は `tree_create` の引数の条件を満たしているとする. さらに, `l` の高さと `r` の高さの差は 2 以内であるとする.

この関数を実装するためには回転という操作が必要である. 以下 `l` の高さが `r` の高さ +1 より大きい場合のアルゴリズムを説明する. 簡単のために二分探索木のノードを `(l v r)` で表現することにする. `tree_create` のように単純に二分探索木を作ると `(l v r)` が得られるが, これは平衡していない. `l` をさらに展開することでこの二分探索木は `((ll lv lr) v r)` と書ける (`l` は必ず `NULL` でないことに注意する). ここで `ll` の高さが `lr` の高さ以上の場合は `(ll lv (lr v r))` が平衡二分探索木となる. `ll` の高さが `lr` の高さ未満の場合は `lr` をさらに展開して `((ll lv (lrl lrv lrr)) v r)` という表現が得られる. そして `((ll lv lrl) lrv (lrr v r))` が平衡二分探索木となる.

今まで作った関数の中で `tree_create` を `tree_bal` に置き換えることで, 常に二分探索木が平衡するようにできる. 以下課題である.

- 関数 `tree_bal` のプログラムの定義を書け.
- (超発展的課題) 上記の回転操作用いて `tree_bal` が実装できることを説明せよ. また、`tree_create` を `tree_bal` に置き換えた箇所で必ず左右の木の高さが 2 以内であるという条件を満たしていることを説明せよ。

4 発展的課題: free することでのメモリリークの防止

課題では不変な (値が書き換えられることのない) 二分探索木を実装した. 仮に**すべての二分探索木は一箇所からしか参照されていない (ポインタで指されていない)** とすると, もう必要のない構造体のメモリを解放することができる.

- `void tree_free(tree_t *t)`: ノード `t` のメモリを解放する関数.

を実装し, 今まで作った関数の中で適切なところで使用せよ. そうすることで `main` 関数の中で一回だけ `tree_create` を呼んで, そこに値を追加, 削除したときに, `main` 関数から辿れないノードは解放されるようにせよ. 動作を確認するために

- `int tree_size(tree_t *t)`: 二分探索木 `t` に含まれる値の個数を返す関数.

を作るとよい. グローバル変数 `count` を用意して, `tree_create` の中でインクリメント, `tree_free` の中でデクリメントすることで現在生きているノードの個数が分かる. それが `main` 関数が持っている二分探索木のサイズと一致することを確認すればよい.

以下課題である.

- 関数 `tree_free` を使用する場所及びその説明を書け.