プログラミング通論'19 ＃13 – 2分木と多分木

久野 靖 (電気通信大学)

2019.4.20

今回は次のことが目標となります。

- 2分木、2分探索木とバランスのためのアルゴリズムについて知る
- 多分木とB木について知る

2分探索木とバランス

2分探索木

前回、配列を用いた表のデータ構造を取り扱いましたが、配列は基本的に値を「詰めて並べて」しまうため、キーの昇順/降順に並べておこうとすると挿入/削除時にその時点のサイズをnとして$O(n)$の時間計算量が掛かります。今回は動的データ構造を用いて上記の問題を克服するという話題を扱います。なお、**2**分木については最大ヒープのところで完全**2**分木の配列表現を学びましたが、今回は配列表現ではなく普通の動的データ構造として木構造を扱います。

動的データ構造ではポインタを用いてノード(**node**, 節)を結びつけますが、今回は木なので「子」の節が複数あります。まず**2**分木、つまり子が最大で**2**つある木構造を扱います。図**1**左のように、個々の節には(整数を鍵・値とする表の例題なので) `key`、`val` というフィールドに加え、子へのポインタを格納する`left`、`right`があります。子の節もまた木構造になっているので、これらは左側と右側の部分木(**subtree**)、とも呼びます。ポインタ値が`NULL`であれば、その部分木は空ということになります。

2 分探索木 (2)

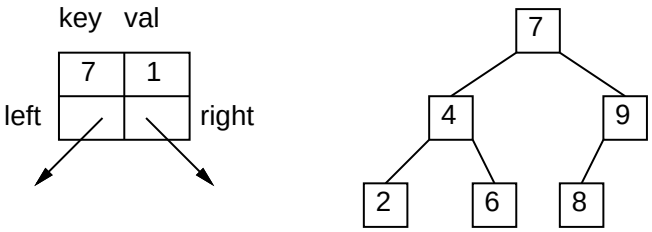

図 1: 2 分探索木の構造

ここで、2 分木の構造全体が 2 分探索木 (binary search tree) であるとは、次の条件が成り立っていることを言います。

> 任意の節 n について、n に格納された鍵 k に対し、n の左の部分木には k より小さい鍵のみが格納され、n の右の部分木には k より大きい鍵のみが格納されている。

図 1 右には、2 分探索木の例が示してあります。たとえば「7」の左の部分木には「2、4、6」が入っていて、これらは確かに「7」より小さいです。さらに、「4」の右の部分木には「6」だけがあり、これは確かに「4」より大きいです。

2分探索木を用いた表の実装

2分探索木をどのように用いて表の機能を実現するのかは、前節の説明でほぼ自明だと思います。探したい鍵の値を持って木の根からはじめ、今見ている節の鍵と一致すれば「見付かった」ことになります。そうでない場合は、今見ている節の鍵と探したい鍵の大小関係に応じて左または右の部分木へ行けばよいわけです。見付かる前にNULLに遭遇した場合は「無い」と分かります。

APIは同様なのでitbl.hを再掲しますが、末尾に項目を削除するitbl_delと現在の中身を打ち出すitbl_prという関数を追加しました(削除は当面コメントアウト)。

```
// itbl.h --- int table api.
struct itbl;
typedef struct itbl *itblp;
itblp itbl_new();                       // allocate new tbl
void itbl_put(itblp t, int k, int v); // store value
int itbl_get(itblp t, int k);           // obtain value
//void itbl_delete(itblp t, int k);     // delete key/value
void itbl_pr(itblp t);                  // print contents
```

2分探索木を用いた表の実装 (2)

実装を見ます。節の構造体struct entは先に示した通りです。して表を表す構造体struct itblは根の節へのポインタだけを持っています (無駄のようですが、空の表を表すにはこうする必要)。なので、空の表を作るのは簡単です。そのあと、今回はitbl_getから読みます。根をパラメタとして下請けの再帰関数getを呼びます。getでは、渡されたポインタがNULLなら「無い」の-1を返します。キーが一致するなら、対応する値を返します。それ以外は…探しているキーと現在の節のキーの大小に応じて、左または右の子のポインタを持って自分を再帰呼び出しします。

```
// bstree.c --- itbl impl with binary search tree.
#include <stdlib.h>
#include <stdio.h>
#include "itbl.h"
typedef struct ent *entp;
struct ent { int key, val; entp left, right; };
struct itbl { entp root; };
itblp itbl_new() {
```

```
  itblp p = (itblp)malloc(sizeof(struct itbl));
  p->root = NULL; return p;
}
static int get(entp p, int k) {
  if(p == NULL) { return -1; }
  if(k == p->key) { return p->val; }
  return get((k < p->key) ? p->left : p->right, k);
}
int itbl_get(itblp p, int k) { return get(p->root, k); }
```

2 分探索木を用いた表の実装 (3)

では itbl_put の方を見ましょう。これも下請けの get を呼び、再帰的にキーに対応する探して行きますが、今度は渡して行くパラメタの方が entp*つまりポインタのポインタになっています。なぜかというと、新しい節を追加する場合には「ポインタの根元を書き込む」必要があるからです。そして、その場所に入っているのが NULL であれば、新しい節を割り当ててそこへのポインタをその場所に書き込み、あとのデータを初期化します。それ以外の場合は探すので先の put と同様ですが、見つかったときはそこに値を設定します。

2分探索木を用いた表の実装 (4)

```
static void put(entp *p, int k, int v) {
  if(*p == NULL) {
    entp q = *p = (entp)malloc(sizeof(struct ent));
    q->left = q->right = NULL; q->key = k; q->val = v;
  } else if(k == (*p)->key) {
    (*p)->val = v;
  } else {
    put((k < (*p)->key) ? &((*p)->left) : &((*p)->right), k, v);
  }
}
void itbl_put(itblp p, int k, int v) { put(&(p->root), k, v); }
```

2分探索木を用いた表の実装 (5)

最後に内容表示を見ましょう。itbl_prはかっこで囲んで根に大して下請けの再帰手続きprを呼びます。prでは自分の内容を表示しますが、ただし左や右の子がNULLでないならそれらも(自分の表示の前/後に)かっこで囲んで出力します(出力そのものはprを再帰呼び出しして行います)。

```
static void pr(entp p) {
  if(p->left != NULL) { printf("("); pr(p->left); printf(") "); }
  printf("%d:%d", p->key, p->val);
  if(p->right != NULL) { printf(" ("); pr(p->right); printf(")"); }
}
void itbl_pr(itblp p) {
  if(p->root != NULL) { printf("("); pr(p->root); printf(")\n"); }
}
```

2 分探索木を用いた表の実装 (6)

では、これらを使って木構造のようすを見るためのプログラムを示します。コマンド引数として値を指定すると、それらを鍵として次々に表に登録します (値は何でもいいのですが「コマンド引数の何番目か」の値にしました。マイナスの値なら削除ということにしますが、まだ削除は実装していないのでコメントアウトしています。

2分探索木を用いた表の実装 (7)

```
// treedemo.c --- demonstration of bstree operation
#include <stdlib.h>
#include <stdio.h>
#include "itbl.h"

int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  itblp p = itbl_new();
  for(int i = 1; i < argc; ++i) {
    int v = atoi(argv[i]);
    if(v >= 0) { itbl_put(p, v, i); }
//  else       { itbl_delete(p, -v); }
    itbl_pr(p);
  }
  return 0;
}
```

2分探索木を用いた表の実装 (8)

では、実行のようすを見てみましょう。次々に節が追加されていくようすを図2に示します。なお、所要時間を計測してみたい場合は前回の計測プログラムを使えばよいです。

```
% gcc8 bstreedemo.c itbl3.c
% ./a.out 5 2 1 7 6 9
(5:1)
((2:2) 5:1)
(((1:3) 2:2) 5:1)
(((1:3) 2:2) 5:1 (7:4))
(((1:3) 2:2) 5:1 ((6:5) 7:4))
(((1:3) 2:2) 5:1 ((6:5) 7:4 (9:6)))
```

2 分探索木を用いた表の実装 (9)

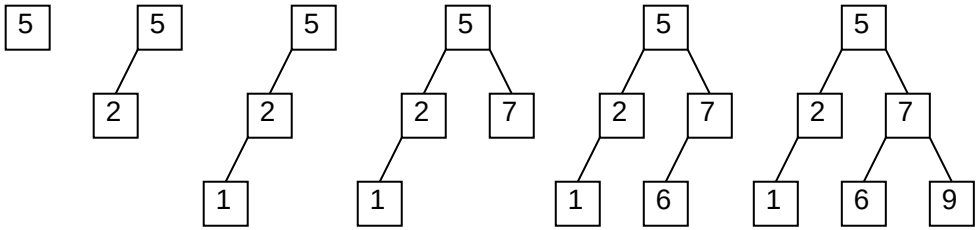

図 2: 2 分探索木ができていく様子

2 分探索木を用いた表の実装 (10)

演習 1 2 分探索木による表の実装を動かし動作を確認しなさい。OK なら以下をやってみなさい。

a. 書き込むところで「ポインタのポインタ」を使うと、コードはコンパクトですが分かりにくい点もあります。ポインタのポインタを使わないように書き換えてみなさい。
ヒント: 左や右に降りる際、再帰関数の返値を「`p->left = put(p->left, k, v)`」のように元のフィールドに入れ直すと書き換えが実現でき、ポインタのポインタを使わずに済みます。この場合、put は「NULL が渡されたら節を作ってデータを登録し、その節を返す」「NULL でなければ右ないし左の子に対して put を呼び出して返値を同じ場所に書き込み、自分の返値は元の節をそのまま返す」となります。

2分探索木を用いた表の実装 (11)

b. 探索 (get) のとき、再帰を使わない方が分かりやすいという人もいるかと思います。再帰を除去してループに書き直してみなさい。

c. 書き込み (put) のときも同様にループに書き直してみなさい。

d. 現在の表示方法は1行におさまるうちは見やすいですが、データが多くなると見づらいかもしれません。データが多くなっても見やすい表示方式を考えて実装してみなさい。

2分探索木からの削除

値の検索と追加は前述のように比較的簡単でしたが、削除には面倒なところがあります。削除したい値をまず探しますが、その値が木のどこに位置していたかで「面倒さ」が違います。

- 削除したい値が葉の位置にあったなら、その節を削除する (親の節からその節に至るポインタを NULL にする) だけで **OK** です。
- 削除したい値が途中の節にあったとしても、その左か右の子へのポインタが NULL であれば、線形リストと同じことで、親の節から直接子の節を指すように変更すればよい (図 **3** 左)。
- 一番面倒なのは、両方に子がある場合。削除した鍵の位置にどれかの鍵を持って来る必要があるが、**2**分探索木の条件を崩さないようにするため、持って来るのは「左の部分木で最大の鍵」または「右の部分木で最小の鍵」となる (どちらかは任意に決めてよい)。

2 分探索木からの削除 (2)

最後の場合についてもう少し説明しましょう。「左の部分木で最大の鍵」を用いるとして、最大の鍵ということは左部分木を行けるだけ右へたどった所にある鍵です。そこが葉ならその節を削除して持って来るだけですが、その節に左の子があるなら、左の子へのポインタを持って来る節へのポインタが入っていた箇所に格納することで木に残します。ですが、葉の場合は左ポインタが NULL ですから、左ポインタをコピーするという同じ動作で両方の場合が扱えます (図 3 中・右)。

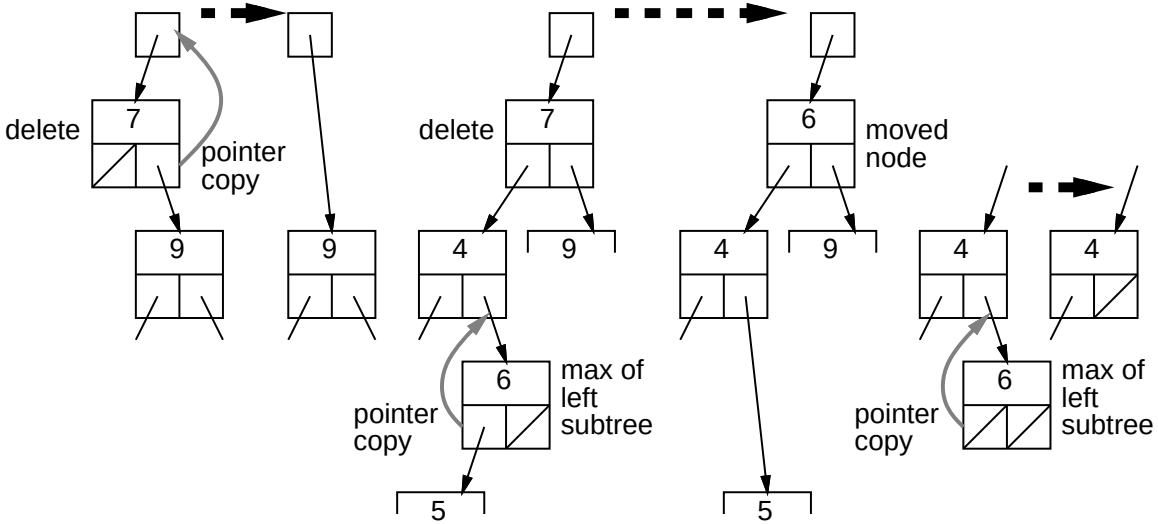

図 3: 2 分探索木からの鍵の削除

2分探索木からの削除 (3)

演習2 2分探索木による表の実装に、鍵と値の対を削除する機能を追加しなさい。
先のmain等のコメントアウトを外し、実際に木の挿入や削除のようすを表示して動作を確認すること。(「2」のように正の整数はその整数を登録するが、「-2」のように負にすると、対応する正の整数「2」を削除するようにできている。)

木のバランスと AVL 木

2 分探索木の (検索、追加、削除における) 時間計算量は木の高さで押えられ、木がバランスしていれば (偏りがなければ) 節数を n としたときに $O(\log n)$ となります。これは、鍵の登録時にランダムに鍵が追加されれば確かにそうなりますが、場合によっては「小さい順に」追加するプログラムもありそうですね。もしそうなると、片方向に伸びた (アンバランスな) 木ができてしまい、高さが n に比例するため、時間計算量も $O(n)$ になってしまいます。

この問題は古くから知られており、これを防ぐために「2 分探索木をバランスさせる」さまざまなアルゴリズムが考案されています。ここでは古くからある方法である AVL 木 (AVL tree) について説明します。[1]AVL 木は、2 分探索木の条件を満たした上で、さらに次の条件を満たすような木です。

> すべての節において、左部分木の高さと右部分木の高さの差はたかだか 1 である。

[1]ほかに代表的なものとして、AA 木、赤黒木、スプレー木などがあります。

木のバランスと AVL 木 (2)

確かにこの条件が満たされていれば、木はバランスしていると言えますが、それを具体的にどうやって実現するのでしょうか。それには、鍵の挿入や削除において常にこの条件を満たすように (なおかつ 2 分探索木の条件も満たすように) 木を変形するのです。

具体的にはまず、木の節数が 0 とか 1 であれば、部分木がないので (高さ 0)、確かに条件は満たされています。次に、鍵を 1 つ挿入・削除すると、挿入により高さが 1 高くなる「ことがあり」ますし、削除により高さが 1 低くなる「ことがあり」ます。1 つの挿入または削除なのですから、高さが 2 以上変化することはない、というのはよいでしょうか。

だとすれば、これまでは条件を満たしていて「挿入・削除により AVL 木の条件を満たさなくなった」時は、「高さの差が 2 になった」時です。その「2 になった」なるべく下の節 (それより下では AVL 木の条件は満たされている) に着目すると、状況は図 4(とその左右対称の状況) で網羅されます。

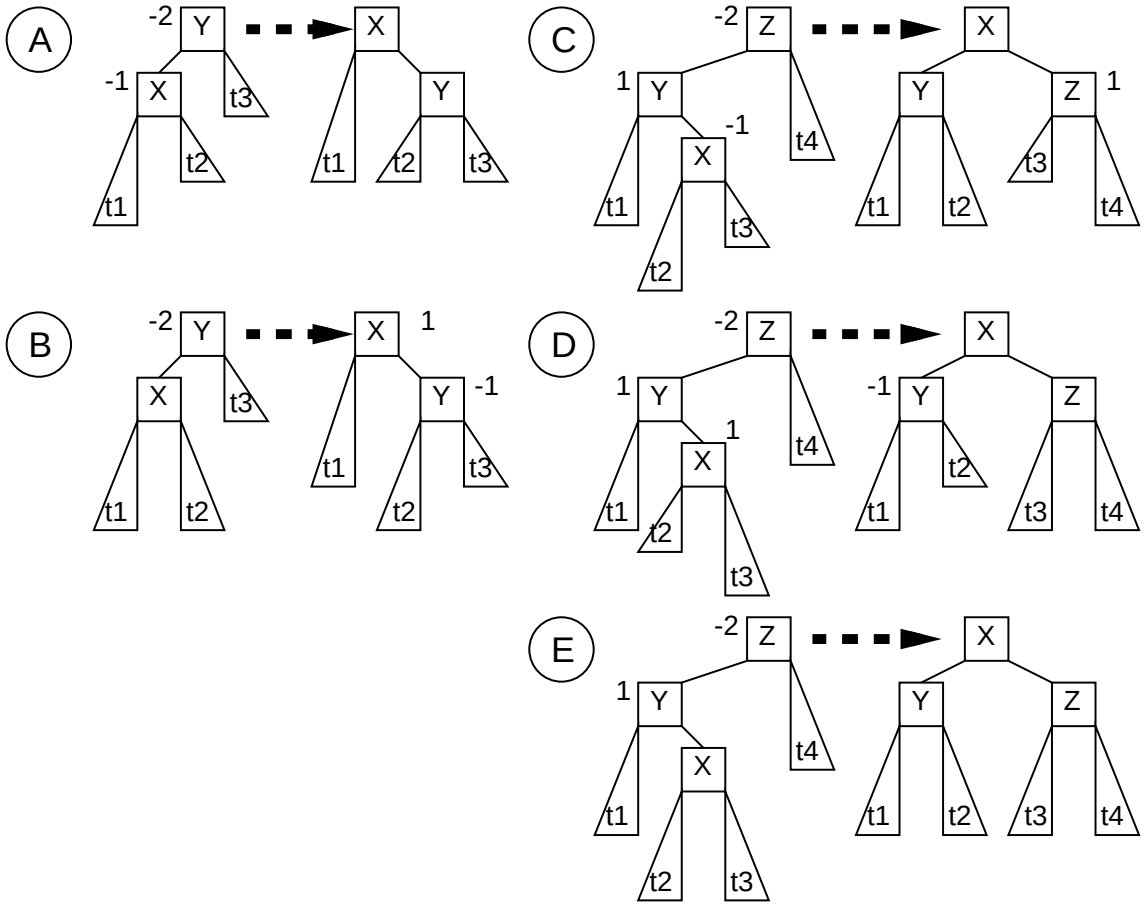

図 4: AVL 木のための木の回転

木のバランスと AVL 木 (3)

いずれのケースも書き換え矢線の左が「差が 2」の状況を表します。節の横に数字が書いてありますが、これは「右の部分木の高さから左の部分木の高さを引いた値」です (仮に「バランス値」と呼びます)。何も書いてない場合はバランス値が 0(両側の高さが同じ) です。そして左右対称のうち、図では「左の方が高い場合」を網羅しています (高さの差が 2 なのでバランス値はすべて-2 です)。

まず (A) を見ましょう。Y の左に X から始まる部分木があり、その X のバランス値は-1、0、1 のいずれかですが、そのうち-1 が (A) の場合です。先に言ってしまうと、0 は (B) であり、1 は (節の記号が着け変えてありますが)(C)、(D)、(E) のいずれかです。1 の時がなぜ 3 つかというと、左側の節のさらに右の子の節の-1、1、0 で分けてあるからです。これで-2 の場合は網羅され、2 の場合はこの左右対称のもので網羅されます。

木のバランスと AVL 木 (4)

話を戻して (A) の場合ですが、矢線で示すように、X が上でその右に Y がつながる形に変更すると (これを回転と呼びます)、X も Y もバランス値が 0 になり、木の高さは 1 減ります (よく見て確認のこと)。2 分探索木の条件については、部分木 t1 は「X より小さい値」、t2 は「X と Y の間の値」、t3 は「Y より大きい値」から成るので、回転した後でも維持されます。(B) の場合も同じ回転をなので 2 分探索木の条件についても同じですが、回転後のバランス値は (A) と異なります。

(C)、(D)、(E) を 3 つに分けた理由は、左の節の右部分木が 1 高いので、(A) や (B) と同じ回転ではバランス値が 1〜-1 にならないためです。そこで、右部分木の根の節をさらに細かく見て t1〜t4 の 4 つの部分木に分け、X〜Z の節を図のようにつなぎ替えます (2 重回転と呼びます)。これによってバランス値が AVL 木の条件を満たすよとともに、木の高さは 1 減ります。

実際に 2 分探索木の実装に組み込む場合は、まず各節のデータとしてバランス値も保持するようにし、挿入・削除の際に正しいバランス値が維持されるうようにします。そのうえで、バランス値が-2 や 2 になったときは、回転・2 重回転を行ってバランスを回復します。

木のバランスと AVL 木 (5)

なお、挿入時には「1 高くなることで」バランスが崩れるので、回転により木の高さが 1 減って「高くなる状況は無くなる」のでそこで **OK** ですが、削除時には「1 低くなることで」バランスが崩れ、回転によってその部分木のバランスは回復しても、「部分木の高さが 1 低くなる」状況は同じままですから、さらに上の (親の) 部分で回転が必要になることもあります。これに対処するのは、再帰手続きであれば「子の節の挿入削除が終わった後で自分の節のバランスをチェックして必要なら回転する」という形で自然に対応できます。

木のバランスと AVL 木 (6)

演習 3 次の順番で AVL 木を実装しなさい (すごく大変だと思うのでよほどやりたい人に限る)。

a. 各節にバランス値を保持するフィールドを追加し、まず挿入に限定してすべての節でバランス値を維持するようにしなさい。`itbl_pr` で表示するときにバランス値も表示するように変更した上で正しく維持されているか確認すること。

b. 上記に加えて、バランス値が 2 と -2 になったときに回転・2 重回転をおこなってバランスが回復されるようにしなさい。実際にさまざまな順番で挿入を行い、回転が正しく行えていることをチェックすること。

c. 引き続いて、削除時のバランス値の維持機能を追加しなさい。

d. 引き続いて、削除時にも回転を行い、AVL 木の実装を完成させなさい。

多分木とB木
平衡木とB木

前節までで見てきたように、木構造では木の高さにより処理時間が決まってくるので、木のバランスを維持することが重要になります。バランスを維持する機能を組み込んだ木構造を平衡木 (baranced tree) と呼びます。

2分木をもとにした平衡木では、AVL木のようにバランスが崩れたことを検出して回転によりバランスを回復したり、(本稿では述べていませんが) 乱数を用いてランダムに木を変形することで確率的にアンバランスが解消されるようにするなどの方法を取ります。

一方、多分木とは次数 (子の数) の最大が2より大きい木構造を言います。多文木では、その性質をうまく用いると、最初からすべての葉が同じ深さにあり、常にその性質が維持されるという形で平衡木を実現できます。その代表的なものであるB木 (B-tree) について見てみましょう。

B木も2分探索木同様、鍵を効率よく探索する機能を提供します。B木では、ある整数dが決められ、節に格納される鍵の個数はd〜$2d$の範囲に制約されます (根は例外で最小が1)。そして、節には鍵数より1多い子ポインタが格納され、それをたどって行くことで目的の鍵を効率よく探せます。

平衡木とB木 (2)

B木の次数は一般に子ポインタの数 $(d+1 \sim 2d+1)$ を用いて「**2-3 B 木 (2-3 木)**」、「**3-5 B木 (3-5木)**」のように表します (根は例外的に鍵が**1**個でもよいことに注意)。

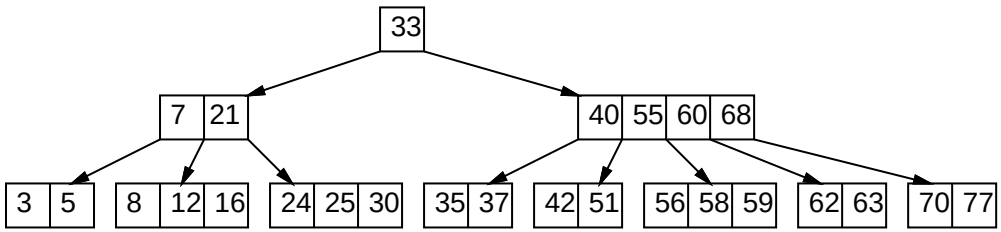

図 5: B 木の例 (3-5 木)

図**5**は**3-5**木の例です。そしてこの図のように、B木を描くときには鍵と子ポインタを交互に描くのが分かりやすいです。子ポインタが指す部分木は、両側の鍵ではさまれた範囲の値を格納します。

平衡木とB木 (3)

たとえば図5左では、「7」「21」の間にあるポインタで指されている部分木はその間の範囲の鍵だけを持ちますし、「21」の右の部分木については、根の「33」から左にたどってきてここへ来ているので、21〜33の範囲の鍵だけを持ちます。

この性質を用いれば、2分探索木と同様、根から始めて特定の鍵のある方へ間違いなく降りてゆけ、木の高さだけおりてゆけば鍵が見付かる(または無いことが分かる)わけです。

なお、実用的にはB木は2時記憶装置上のデータ構造として多く使われます。つまり、1つのディスクブロック(4096バイト等)に1つの節を格納し、次数を大きくすることで少ない段数(ブロック読み込み回数)で目的のデータに到達できるのです。ここでは概念の説明なので普通にメモリ上のデータ構造として扱っています。

B 木における鍵の追加

検索について分かったところで、鍵の追加がどのように働くかを見ます。図 6 も先と同様、3-5 木を表しているのとします。ここで下の節に「27」を追加しようとした場合、鍵の数が 5 になると $2d$ を超えてしまいます。そのような場合は、節全体を 2 つに分割し、中央の鍵 (この場合では「23」) を親に移します。このとき、親の節では「3」「23」の間の子ポインタが左の節、「23」「41」の間の子ポインタが右の節を指すようにすることで、B 木の性質が維持されます。

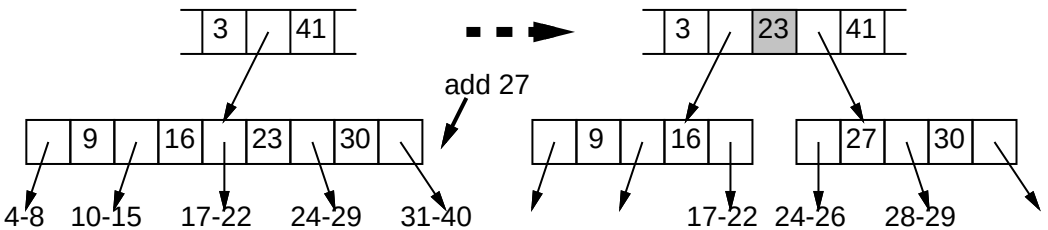

図 6: B 木への値の挿入

B木における鍵の追加 (2)

しかし最初の「27」はどこから来たのでしょうか。また親の節に「23」を追加しようとしたとき、親の節が満杯ならどうするのでしょうか。これらの答えは簡単で、まずB木では値の追加は葉の節でのみ行います。そして、葉の節で鍵の個数が$2d$を超えたため分割が起き、その結果図の節に「27」が挿入されようとしているわけです。

また、親の節も$2d$を超えるならやはり分割が起き、それが上に伝播していって$2d$で収まるところで止まります。根まで来ても$2d$を超えるときは、根の節を2つに分割し、1つの鍵と2つの子ポインタを持つ新しい根を作ってそこから指させます。このときは木の段数が1段増えます。削除については後の節で扱います。

例題: 2-3 木の実装

それでは例として、2-3 木の実装を見てみましょう。AVL 木と同様、削除は練習問題にするのでここでは省略しています。まず冒頭部分ですが、表は根だけを持つ構造体で、節が重要になります。

各節には葉かどうかを表すフィールド leaf、現在格納してる鍵の数 nkey と、鍵、対応する値、そして子ポインタの配列 key、val、child を持たせます。鍵 key[i] に対して、その左の子ポインタは child[i]、右の子ポインタは child[$i+1$] ということになります。配列のサイズは子ポインタだけ $2d+1$ であとは $2d$ でいいのですが、コードを少し簡単にするためすべて同じにしました。表の初期化は根を NULL に初期化するだけです。

例題: 2-3木の実装 (2)

```
// btree23.c --- itbl impl with 2-3 B-tree.
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include "itbl.h"
#define D 1
typedef struct ent *entp;
struct ent {
  bool leaf;
  int nkey, key[D*2+1], val[D*2+1];
  entp child[D*2+1];
};
struct itbl { entp root; };
itblp itbl_new() {
  itblp p = (itblp)malloc(sizeof(struct itbl));
  p->root = NULL; return p;
```

```
}
```

例題: 2-3 木の実装 (3)

検索ですが、まず根が NULL なら木が空っぽなので何も含まれていません。それ以外の場合は itbl_get は根をパラメタとして下請けの get を呼びます。

get は節に含まれている鍵との一致を見て行き、一致すれば対応する値を返します。一致せず、なおかつ探している鍵 k より大きい鍵が現れたら、その左の子ポインタをたどって探します。どの鍵も k より小さく右端に来たら、最後の子ポインタを使用します。いずれにせよ、子ポインタの指す先から探すのは再帰呼び出しによって行います。葉まで来ていたら見つからなかったので -1 を返します。

例題: 2-3 木の実装 (4)

```
static int get(entp p, int k) {
  int i;
  for(i = 0; k >= p->key[i] && i < p->nkey; ++i) {
    if(k == p->key[i]) { return p->val[i]; }
  }
  return (p->leaf) ? -1 : get(p->child[i], k);
}
int itbl_get(itblp p, int k) {
  return (p->root == NULL) ? -1 : get(p->root, k);
}
```

例題: 2-3 木の実装 (5)

さていよいよ、ややこしい挿入の処理です。下請け用として、鍵と対応する値を指定して鍵を 1 個持つ節を作る関数newent2(左と右の子ポインタを指定)、newent(葉の節なので子ポインタは NULL)、および鍵と値を別の節の i 番目から取って来る cpent2、cpent を用意しました。また、shift はある節の i 番に新しい鍵を挿入したい場合に使用し、i 番より後ろにある鍵、値、子ポインタをすべて 1 つずつ後ろにずらして場所をあけます。

例題: 2-3木の実装 (6)

```
static entp newent2(int k, int v, entp l, entp r) {
  entp p = (entp)malloc(sizeof(struct ent));
  p->leaf = false; p->nkey = 1; p->key[0] = k; p->val[0] = v;
  p->child[0] = l; p->child[1] = r; return p;
}
static entp newent(int k, int v) {
  entp q = newent2(k, v, NULL, NULL); q->leaf = true; return q;
}
static entp cpent2(entp p, int i, entp l, entp r) {
  entp q = newent2(p->key[i], p->val[i], l, r); return q;
}
static entp cpent(entp p, int i) {
  entp q = cpent2(p, i, NULL, NULL); q->leaf = true; return q;
}
static void shift(entp p, int i) {
  for(int j = ++p->nkey; j > i; --j) {
```

```
    p->key[j] = p->key[j-1]; p->val[j] = p->val[j-1];
    p->child[j] = p->child[j-1];
  }
}
```

例題: 2-3 木の実装 (7)

次のレベルの下請けとして、節が葉で位置 i に鍵を挿入すると決まった場合の処理を行う pleaf を示します。節を分割する場合があるので、分割した場合は上に移す 1 つの鍵と分割した 2 つの節を持つ節を作成して返すようにします (以下の pnode、put も同じ)。分割が不要なら NULL を返します。

節に入る鍵の上限までに余裕があるなら、i より後を shift でずらして空いた位置に鍵と値を挿入すれば完了で、NULL を返します。それ以外は分割が必要で、分割位置 i が 0、1、2 のそれぞれに応じて上に移す鍵、左右の節を適切に設定した節を作り返します。

例題: 2-3木の実装 (8)

```
static entp pleaf(entp p, int i, int k, int v) {
  if(p->nkey < D*2) {
    shift(p, i); p->key[i] = k; p->val[i] = v; return NULL;
  } else if(i == 0) {
    return newent2(p->key[0], p->val[0], newent(k, v), cpent(p, 1));
  } else if(i == 1) {
    return newent2(k, v, cpent(p, 0), cpent(p, 1));
  } else {
    return newent2(p->key[1], p->val[1], cpent(p, 0), newent(k, v));
  }
}
```

例題: 2-3 木の実装 (9)

中間ノードの場合は、葉の方から分割の情報が (上述のような形で) 節 q として渡されて来て、それを取り扱う下請け pnode を用意しました。まず q が NULL なら分割をしなかったのでこちらも作業はなく、NULL を返します。また格納する鍵の数に余裕があれば、shift でずらして空いた位置 i に格納すれば終わりで、これも NULL を返します。それ以外は葉の時と同じように 3 つに場合分けですが、ただしこちらは左右の子ポインタも適切に返す必要があるのでやや面倒です。

例題: 2-3木の実装 (10)

```
static entp pnode(entp p, entp q, int i) {
  if(q == NULL) { return NULL; }
  entp r = NULL;
  if(p->nkey < D*2) {
    shift(p, i); p->key[i] = q->key[0]; p->val[i] = q->val[0];
    p->child[i] = q->child[0]; p->child[i+1] = q->child[1];
  } else if(i == 0) {
    r = newent2(p->key[0], p->val[0],
                newent2(q->key[0], q->val[0], q->child[0], q->child[1]),
                cpent2(p, 1, p->child[1], p->child[2]));
  } else if(i == 1) {
    r = newent2(q->key[0], q->val[0],
                cpent2(p, 0, p->child[0], q->child[0]),
                cpent2(p, 1, q->child[1], p->child[2]));
  } else {
    r = newent2(p->key[1], p->val[1],
```

```
                cpent2(p, 0, p->child[0], p->child[1]),
                newent2(q->key[0], q->val[0], q->child[0], q->child[1]));
  }
  free(q); return r;
}
```

例題: 2-3 木の実装 (11)

put は部分木に鍵と値の組を書き込み、分割が起きた場合は分割の中央の鍵と左右の子ポインタのみを持つ節を返します。その内容ですが、まず鍵の並びを調べて一致する鍵があれば、対応する値を書き込むだけで完了です。そうでない場合、変数 i は k より大きい最初の鍵の位置になっていることに注意。そして、この節が葉かどうかに応じて pleaf、pnode のいずれかを呼び出せば仕事は完了です。

例題: 2-3木の実装 (12)

```
static entp put(entp p, int k, int v) {
  int i;
  for(i = 0; k >= p->key[i] && i < p->nkey; ++i) {
    if(k == p->key[i]) { p->val[i] = v; return NULL; }
  }
  return p->leaf ? pleaf(p, i, k, v) : pnode(p, put(p->child[i], k, v), i)
}
void itbl_put(itblp p, int k, int v) {
  int k1, v1;
  if(p->root == NULL) { p->root = newent(k, v); return; }
  entp r = put(p->root, k, v);
  if(r != NULL) { free(p->root); p->root = r; }
}
```

例題: 2-3 木の実装 (13)

以上の下請けが用意されているとして、itbl_put は根が NULL なら書き込む鍵と値の組を持った根ノード (葉でもある) を作成して指させます。それ以外の場合は根をパラメタとして下請けの put を呼び、返値がノードであればそれを新たな根とします (木の高さが 1 段高くなる場合)。NULL であれば根の節での分割は起きなかったのでとくにすることはありません。

最後に、B 木全体をプリントする関数も用意しました。2 分探索木と似ていますが、部分木と「鍵:値」とを交互に打ち出すように一般化されています。

例題: 2-3木の実装 (14)

```
static void pr(entp p) {
  int i;
  if(p == NULL) { printf("NULL"); return; }
  if(p->leaf) {
    if(p->nkey > 0) { printf("%d:%d", p->key[0], p->val[0]); }
    for(i = 1; i < p->nkey; ++i) {
      printf(" %d:%d", p->key[i], p->val[i]);
    }
  } else {
    for(i = 0; i < p->nkey; ++i) {
      printf("("); pr(p->child[i]);
      printf(") %d:%d ", p->key[i], p->val[i]);
    }
    printf("("); pr(p->child[i]); printf(")");
  }
}
```

```
void itbl_pr(itblp p) { printf("("); pr(p->root); printf(")\n"); }
```

例題: 2-3 木の実装 (15)

では、先のテストプログラムと組み合わせて動かします。確かに、番号順に挿入しているのにすべての葉の深さは同じになるように木が作られて行きます。

```
% gcc8 treedemo.c btree32.c
% ./a.out 1 2 3 4 5 6 7 8 9 10
(1:1)
(1:1 2:2)
((1:1) 2:2 (3:3))
((1:1) 2:2 (3:3 4:4))
((1:1) 2:2 (3:3) 4:4 (5:5))
((1:1) 2:2 (3:3) 4:4 (5:5 6:6))
(((1:1) 2:2 (3:3)) 4:4 ((5:5) 6:6 (7:7)))
(((1:1) 2:2 (3:3)) 4:4 ((5:5) 6:6 (7:7 8:8)))
(((1:1) 2:2 (3:3)) 4:4 ((5:5) 6:6 (7:7) 8:8 (9:9)))
(((1:1) 2:2 (3:3)) 4:4 ((5:5) 6:6 (7:7) 8:8 (9:9 10:10)))
```

例題: 2-3 木の実装 (16)

演習 4 2-3 木の例題を動かし、B 木の挿入アルゴリズムを確認しなさい。納得したら、3-5 木 (ないしもっと次数の大きい木) を実装してみなさい。(ヒント: 例題のコードは、分割処理けだけは 0/1/2 の場合に分けて扱っていますが、そこ以外は 2-3 木であることをとくに使っていないの、変更しないで大丈夫なはずです。)

B木における鍵の削除

ここまでは挿入だけを扱って来ましたが、B木における鍵の削除はどうでしょうか。削除ではどの位置の値でも削除できますが、中間の節から鍵を削除する場合は、2分探索木と同様に「その鍵の左の子ポインタが指している部分木で最大の鍵」または「その鍵の右の子ポインタが指している部分木で最小の鍵」を持って来てそこに移す必要があります。これらは葉にあるので、いずれにせよ葉から要素を削除するという動作になるわけです。

削除の結果、鍵の数がdより少なくなる場合は隣接する節から融通します。融通する場合は、隣接する節の間にある鍵を少ない方の節に移し、代わりにその位置に多い側の節から持って来た鍵を入れます (図7)。葉では子ポインタはないですが、中間の節の場合は隣接する子ポインタも対応して移すことになります。

B木における鍵の削除 (2)

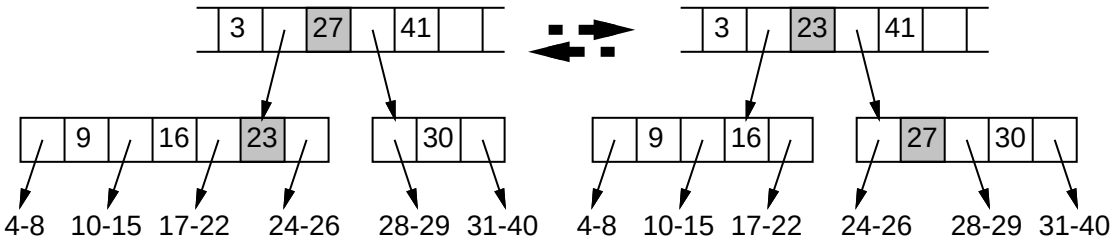

図 7: 隣接ノード間での鍵の融通

隣接する節の鍵の数もdで融通できないときは、2つの節を1つに統合し、親から間の鍵を取って来ることで$2d$の鍵を持つ節にします。このとき、親の節の鍵がdを下回る場合はさらに同じ処理を繰り返し、結果として根から鍵が1つも無くなる時は木の高さが1段減ります。

演習5 2-3木に削除操作を追加して動かしなさい。

演習6 より次数の大きいB木の実装について、削除操作を追加して動かしなさい。

本日の課題 13A

「演習 1」～「演習 6」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. sol または CED 環境で「/home3/staff/ka002689/prog19upload 13a ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. 2 分探索木がプログラムできるようになりましたか。

   Q2. AVL や B 木のアルゴリズムを理解しましたか。

   Q3. リフレクション (今回の課題で分かったこと) ・感想・要望をどうぞ。