教科書輪講

プログラミングコンテスト攻略のための
アルゴリズムとデータ構造

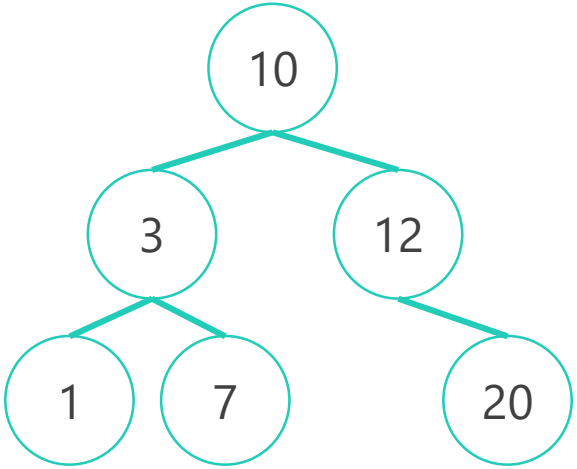
第9章　二分探索木

秋山研究室 M2 山本悠生

二分探索木

二分探索木とは

- 二分木の一種
- 基本的な探索木
- 各接点にキーを持つ
- ある節点と左右の子のキーをk, l, rとすると
  $l \leq k$ かつ $k \leq r$となるように
  木を構築する
- 左右の部分木の要素はすべてk以下（以上）

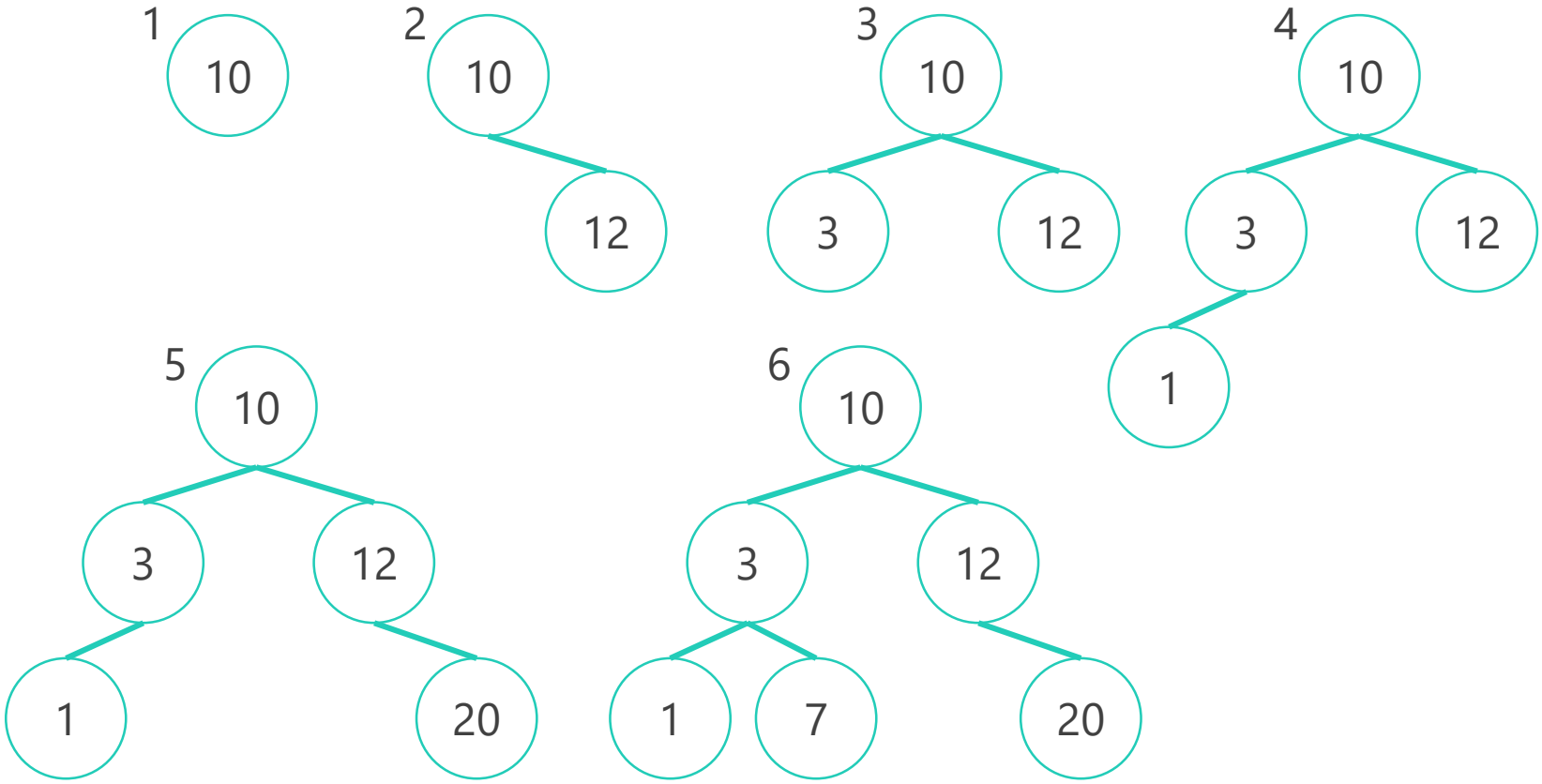


二分探索木の挿入

- 挿入アルゴリズム
  1. 根がNILであれば挿入したい要素で置き換えて終了する
  2. 根のキーと挿入したいキーを比較する
  3. 小さいなら左、大きいなら右を新たな根とみなす
  4. 1.に戻る
- 次のページでこのアルゴリズムでの挿入の例を示す
  - 葉についているNILは図では省略する

二分探索木の挿入

空の木に{10,12,3,1,20,7}をこの順で挿入すると



二分探索木の挿入

- 一見よさそうに見えるが…
  - 仮に挿入順序が$\{1,3,7,10,12,20\}$の時を考える
    - 全部右に行ってしまって実質リスト
    - これを解決するデータ構造に平衡二分探索木がある
      - 具体的な構造は赤黒木など複数の種類
      - ここでは扱わない
- 単純な探索二分木の挿入にかかる計算量は
  - 木の高さをhとすると$O(h)$
  - 入力に偏りがなければ$O(\log n)$
  - 上のような最悪の場合$O(n)$

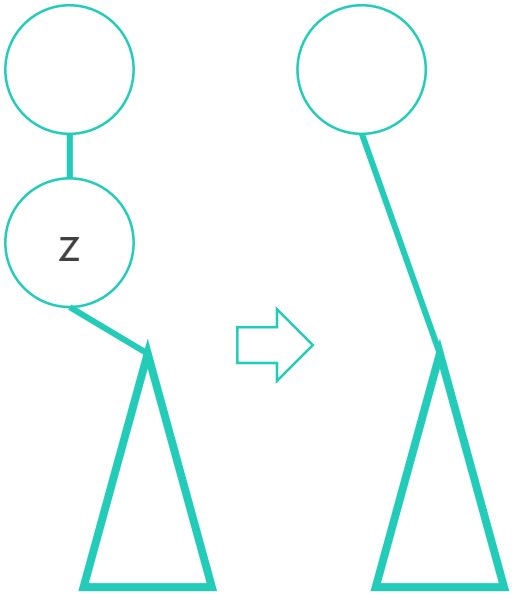
二分探索木の探索

- 探索アルゴリズム
  - 木の中からキーがkである節点を探すことを考える
  1. 根がNILなら見つからなかったとして終了する
  2. 根のキーrとkが等しいか調べて
     等しければ見つかったとして終了する
  3. $k < r$なら左、そうでなければ右を
     新しい根とみなす
  4. 1.に戻る
  - 計算時間のオーダーは挿入と同じ
    - 挿入する場所を探索して挿入しているので
      当たり前と言えば当たり前

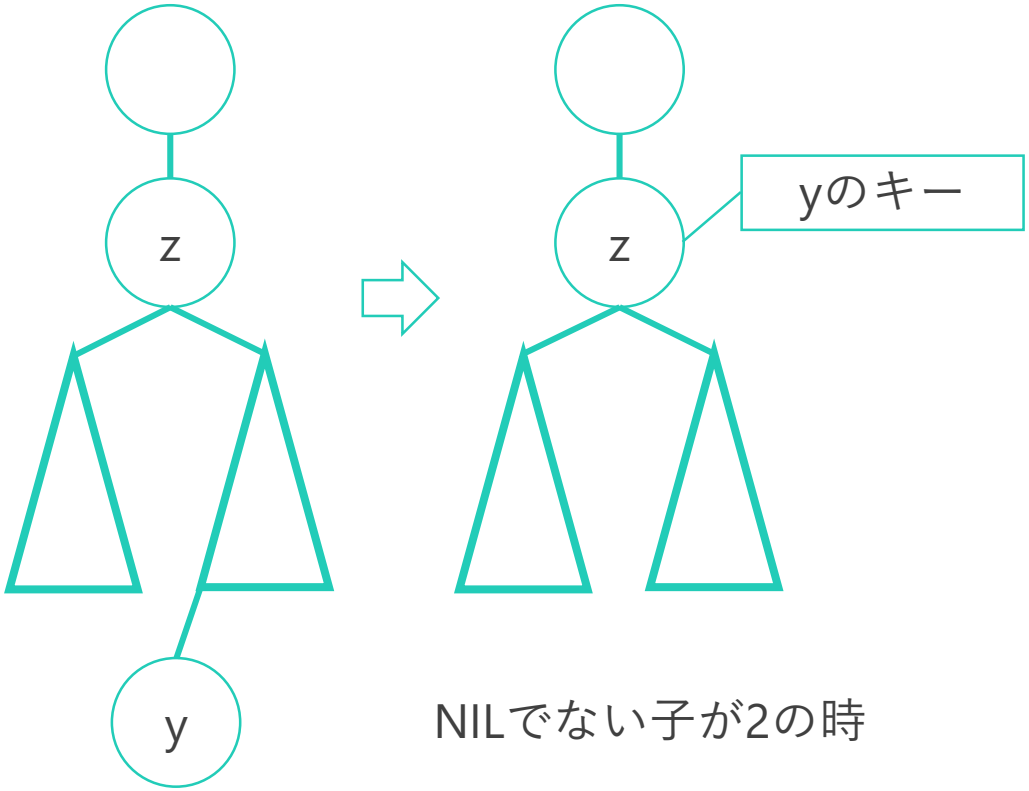
二分探索木の削除

- 削除アルゴリズム
  1. 探索のアルゴリズムを用いて削除したいキーの節点を見つけ、zとする
  2. zのNILでない子の数が
     1. 0ならzの代わりに親にNILをつなげる
     2. 1ならzの代わりに唯一の子をつなげる
     3. 2のときはzの次節点（中間順巡回で次に巡回される節点）yのデータをzにコピーし、yを削除する
        - このときzの右の子は必ず存在するので
          次節点は右部分木で一番左側にある葉となる

二分探索木の削除



NILでない子が0か1の時
（この部分木はNILでもよい）



NILでない子が2の時

キーの重複がなければキーの値は
左部分木 $< z < y <$ y以外の右部分木
となるのでyのキーにはz削除後の値となる資格がある
（左部分木の最右節点をyとしてもよい）

問題：二分探索木の操作

ALDS1_8_{A,B,C}: Binary Search Tree {I,II,III}

問題：二分探索木に関する操作をm個与えるとき

操作のリストに従って二分探索木を操作する

操作リスト：

A: insert k: キーがkである要素を挿入する

A: print: キーを中間順と先行順で出力する

B: find k: キーkが存在するかどうか報告する

C: delete k: キーkを持つ節点を削除する

A,B,CはAOJでの問題記号に対応

AではAの実装、BではAとBの実装、CではABC全ての実装が必要

問題：二分探索木の操作

入力：
1行目に命令数mが与えられる
続くm行に操作が与えられる

```
18
insert 8
insert 2
insert 3
insert 7
insert 22
insert 1
find 1
find 2
find 3
find 4
find 5
find 6
find 7
find 8
```

（続き）

```
print
delete 3
delete 7
print
```

出力：
find操作があったときは
yesかnoで答える
print操作があったときには
中間順、先行順でそれぞれ1行に出力
全てのキーの前に空白を入れる

```
yes
yes
yes
no
no
no
yes
yes
 1 2 3 7 8 22
 8 2 1 3 7 22
 1 2 8 22
 8 2 1 22
```

問題：二分探索木の操作

制約：

$m \leq 500{,}000$

print命令数 ≤ 10

$-2{,}000{,}000{,}000 \leq k \leq 2{,}000{,}000{,}000$（32bit整数で表現可能）

説明したアルゴリズムで実装する限り木の高さは100を超えない

キーの重複は起こらない

C言語を使用している場合はtemplate.cを使うと楽かもしれない

`./test.sh hoge.c` とすると自動的にコンパイルして4つの

テストケースで実行してくれる

（Pythonなどの場合でも少し変更すれば可能）

解説：二分探索木の操作

insert: 入れるべきところは今はNILのはずなので、たどり着いたら置き換える

```
void insert(node **tree, const int key, node *parent) {
  if(*tree == NIL) {
    *tree = createNode(parent, key);
  } else {
    node **next = (*tree)->key > key ? &(*tree)->left : &(*tree)->right;
    insert(next, key, *tree);
  }
}
```

print: 8章でやった通り

```
void print(const node *tree) {
  printInOrder(tree);
  printf("¥n");
  printPreOrder(tree);
  printf("¥n");
}
```

```
void printInOrder(const node *t) {
  if(t == NIL) return;
  printInOrder(t->left);
  printf(" %d", t->key);
  printInOrder(t->right);
}
```

解説：二分探索木の操作

find: 見つかるかNILまでkeyを比較して左右に潜る

```
node* findNode(const node *tree, const int key) {
  if(tree == NIL) {
    return NIL;
  } else if(tree->key == key) {
    return (node*)tree;
  } else if(tree->key > key) {
    return findNode(tree->left, key);
  } else {
    return findNode(tree->right, key);
  }
}

int find(const node *tree, const int key) {
  return findNode(tree, key) != NIL;
}
```

解説：二分探索木の操作

delete: updateParent(a, b)はaの親のaに向いているポインタをbに変える

```
void delete(node **tree, const int key) {
  node *hit = findNode(*tree, key);
  if(hit == NIL) return;
  if(hit->left == NIL || hit->right == NIL) {
    node *next = hit->left == NIL ? hit->right : hit->left;
    if(hit->parent == NIL) {
      *tree = next;                    //根かどうか
    } else {
      updateParent(hit, next);
    }
    next->parent = hit->parent;
    freeNode(hit);
  } else {
    node *minSub = minNode(hit->right);
    hit->key = minSub->key;
    delete(&minSub, minSub->key); //子は必ず1つか0
  }
}
```

NILでない子が0か1つ

NILでない子が2つ