二分探索木

今回の要点

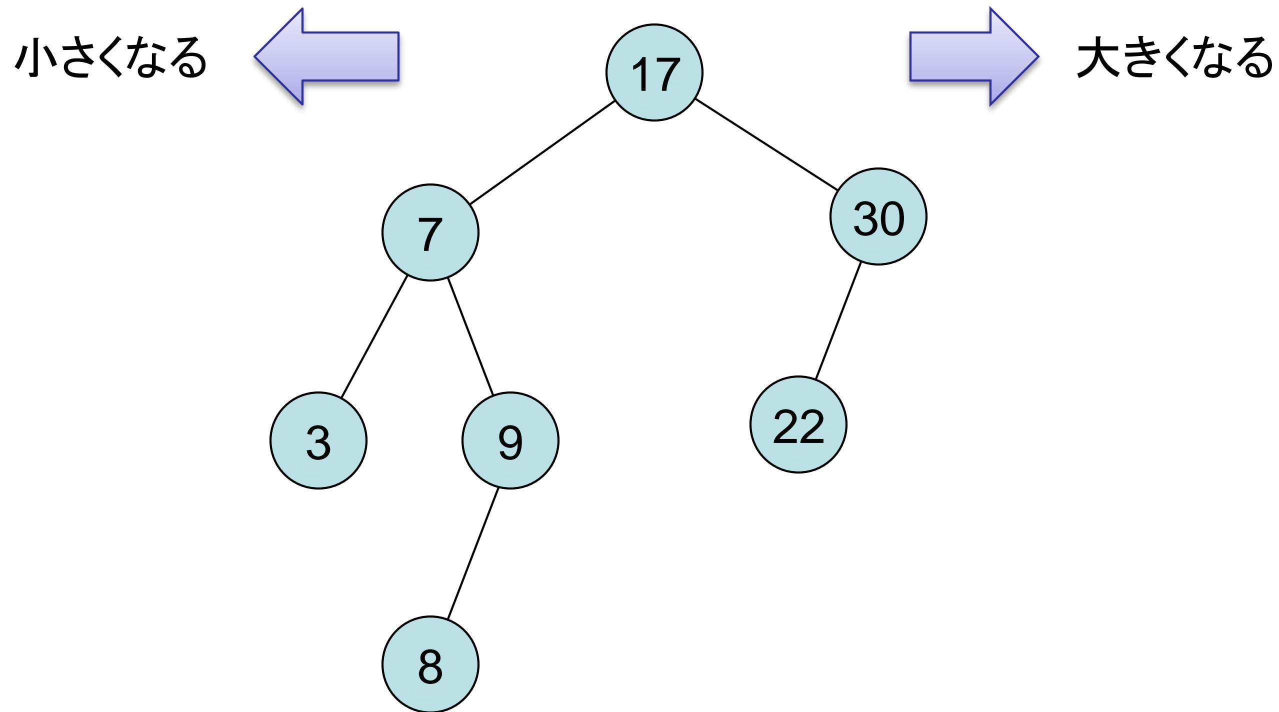
- 二分探索木の構造
  - どのような条件を満たさねばならないか？
    - 二分探索が効率よく出来るため
- 二分探索木の操作
  - 探索の方法
  - 挿入の方法
  - 削除の方法
  - 各操作の実装コード
- 二分探索木の性質
  - どのような形がもっと探索に適しているか？

二分探索木とは

- 木構造
  - 枝分かれした構造を表現するのに適する
  - 根から葉に向かってたどる＝探索
  - 何らかの特徴を持って構成されていると探索しやすい
- 二分探索木
  - 二分木を、データの探索に有効であるように構成した木
  - 任意のノード x について、
    - 左部分木の含まれる要素は x よりも小さく、
    - 右部分木に含まれる要素は x よりも大きい
  - x は「比較可能」でなければならない
- 二分探索法とは異なる
  - 配列に格納されている整列されたデータを探索
  - 配列では、「真ん中」がどこか、すぐにわかる
    - 長さnの場合、int(n/2) が真ん中
  - リスト構造や木構造ではこれが困難であり、使えない

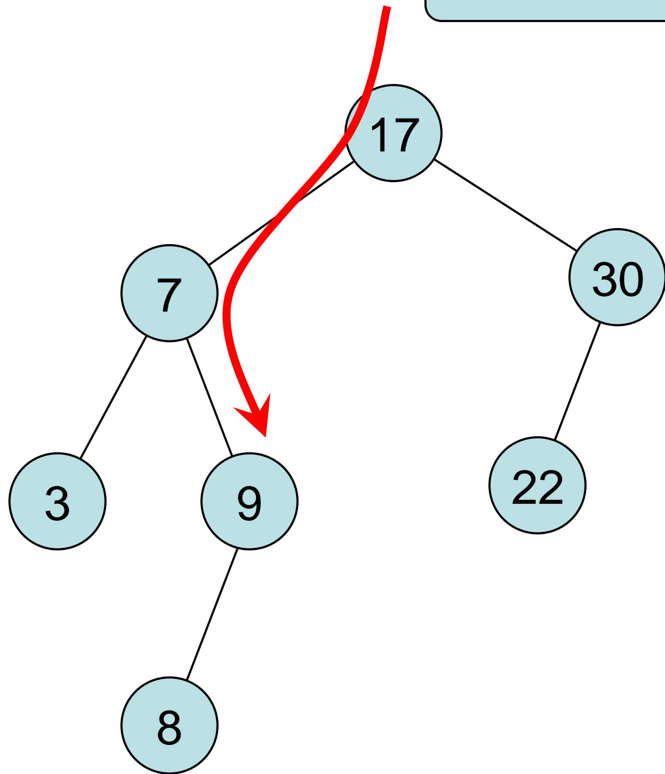
こうなるように、
注意深く作る
必要がある！

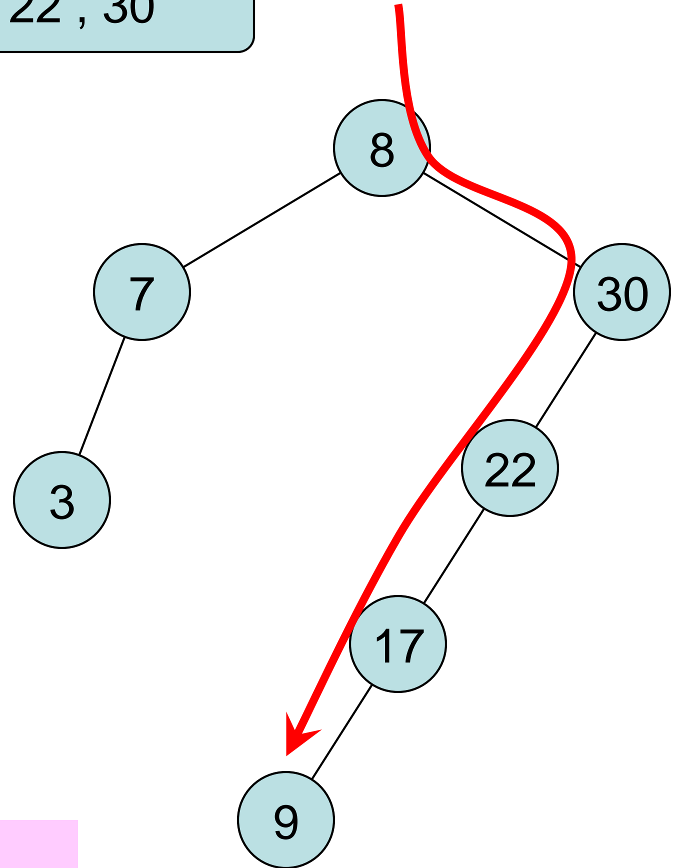
二分探索木の例



具体的な探索

3 , 7 , 8 , 9 , 17 , 22 , 30





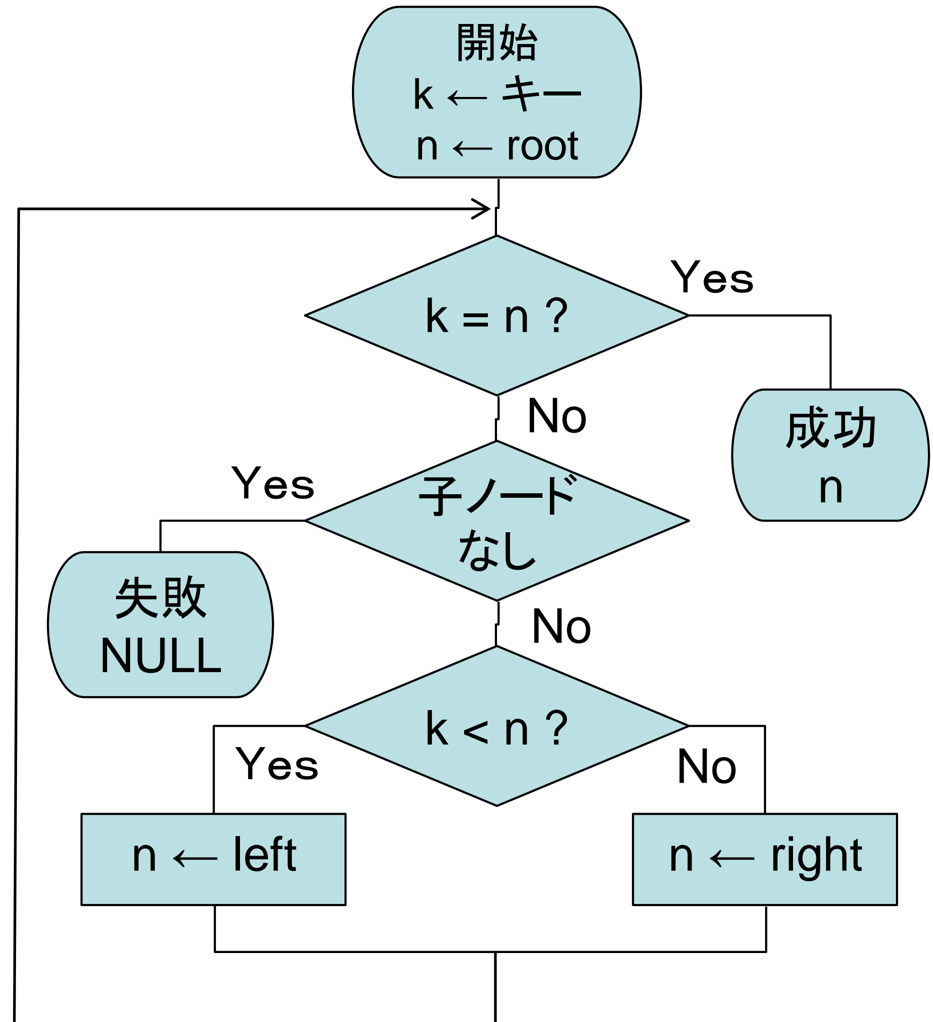
２回の比較

9 を探索

４回の比較

二分探索木の探索

- 探索手順
  1. 探索キーを k、根ノードの要素を n
  2. $n = k$ なら終了（探索成功）
  3. n に子ノードがなければ終了（探索失敗）
  4. $k < n$ なら 左の子ノードを n に
  5. $k > n$ なら 右の子ノードを n に
  6. 2. へ



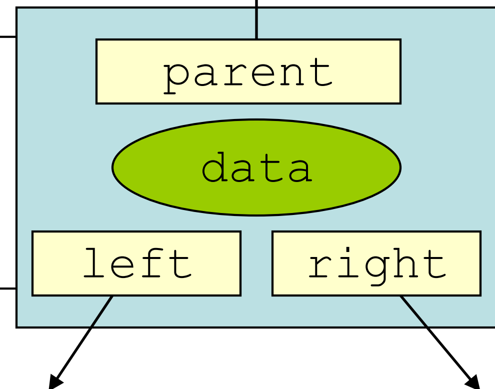
木構造の型

- BinTreeNode

```
typedef struct BinTreeNodeTag {
  struct BinTreeNodeTag *parent;
  struct BinTreeNodeTag *left,*right;
  void *data;
} BinTreeNode;
```

- BinTree

```
typedef struct {
  BinTreeNode *root
} BinTree;
```



二分探索を可能にする

- 従来の二分木に探索処理を実装する
  - 探索 (search)
    - 二分探索木を探索する
  - 挿入 (insert)
    - 二分探索木の構造を壊さないように挿入する
  - 削除 (remove)
    - 二分探索木の構造を壊さないように削除する

BinSearchTree.h

```
#ifndef __BinSearchTree__h
#define __BinSearchTree__h
/***
 *** 二分探索木
 ***/
#include "BinTree.h"

// プロトタイプ宣言
// comp(a,b):aとbを比較する関数
void *search(BinTree*,void*,int(*comp)(void*,void*));
void insert(BinTree*,void*,int(*comp)(void*,void*));
void remove(BinTree*,void*,int(*comp)(void*,void*));

#endif // __BinSearchTree__h
```

比較関数compを
引数として渡す

比較関数とは

- `int (*comp)(void *a,void *b);`
- 2つのデータaとbを比較する
  - a=b のときゼロ
  - a<b のとき負の数
  - a>b のとき正の数、を返す
- 例：

```
// 比較
int comparePDage(void *p1,void *p2){
  return ((PD*)p1)->age-((PD*)p2)->age;
}
```

関数名は何でもよいが、比較するデータ内容を表しているようにするとよい。

山田(18) ＜ 森(55) ＜ 今井(60)

BinSearchTree.cc

```
/***
 *** BinSearchTreeの実装
 ***/
#include "BinSearchTree.h"

// 探索
void *search(BinTree *tree,void *key,int(*comp)(void*,void*)){
}

// 挿入
void insert(BinTree *tree,void *key,int(*comp)(void*,void*)){
}

// 削除
void remove(BinTree *tree,void *key,int(*comp)(void*,void*)){
}
```

これらの関数を
実装する。

探索の実装

```
// 探索
void *search(BinTree *tree,void *key,int(*comp)(void*,void*)){
  BinTreeNode *n=tree->root;
  while(n){
    int c=comp(key,n->data);
    if(c==0) return n->data;
    n=(c<0)?n->left:n->right;
  }
  return NULL;
}
```

要素の比較

見つかった！

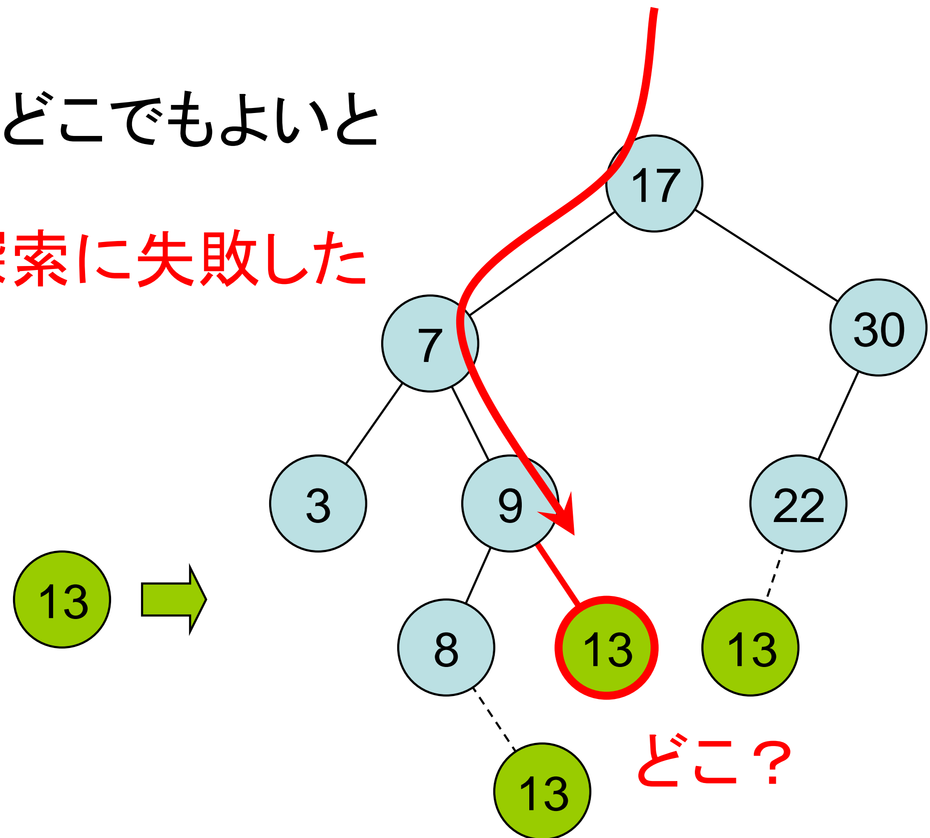
探索ループ

見つからない場合、keyの方が
小さければ左へ、そうでなければ右へ

見つからない

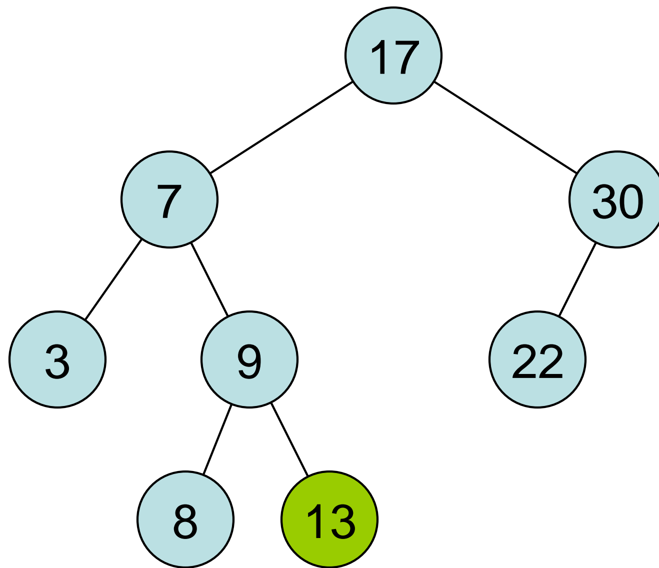
要素の挿入１

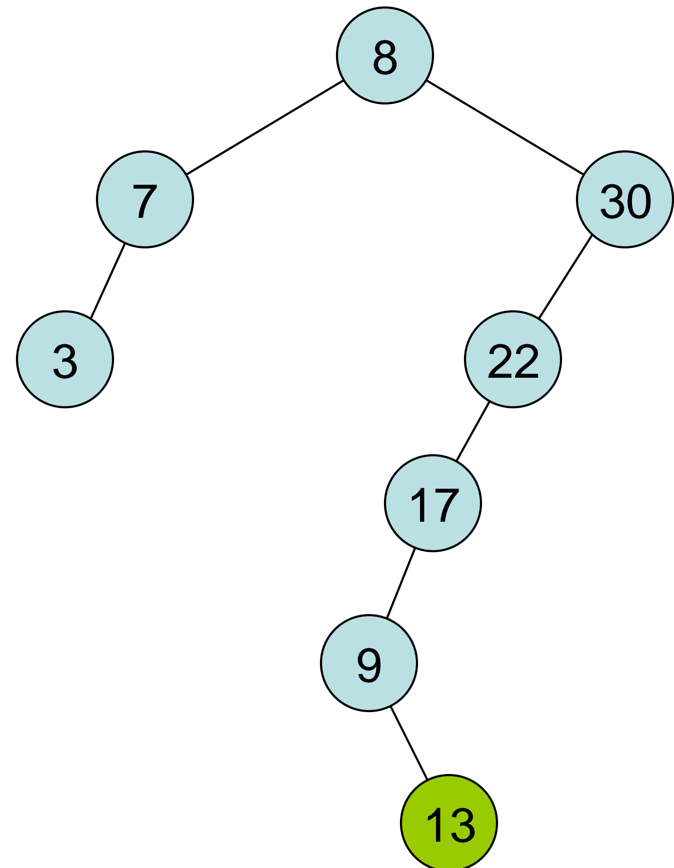
- 二分探索木の構造を保存
  - 要素を挿入した後も、左が小さく、右が大きい
- どこに入れるべきか？
  - 親ノードより大きければどこでもよいというわけではない
  - 根から探索していき、探索に失敗した地点が挿入ポイント



要素の挿入2

- 木の形によって、どこに挿入されるかが変化する





挿入の実装1

- 挿入を行うには、探索に失敗したポイントが必要
- 探索関数search()を下記のように変更する

```
BinTreeNode *prev;

// 探索
void *search(BinTree *tree,void *key,int(*comp)(void*,void*)){
  BinTreeNode *n=tree->root;
  prev=NULL;
  while(n){
    int c=comp(key,n->data);
    prev=n;
    if(c==0) return n->data;
    n=(c<0)?n->left:n->right;
  }
  return NULL;
}
```

探索失敗の直前のノードを保存しておくための変数

探索に成功したときはprevにそのノードが入っている

prevにnを代入してから、nを更新している

探索に失敗したときはprevに失敗の直前のノードが入っている

挿入の実装2

```
// 挿入
void insert(BinTree *tree,void *key,int(*comp)(void*,void*)){
  if(search(tree,key,comp)) return;
  BinTreeNode *n=makeBinTreeNode(key);
  if(!prev)
    tree->root=n;
  else if(comp(key,prev->data)<0)
    prev->left=n;
  else
    prev->right=n;
  n->parent=prev;
}
```

keyを探索し、見つかったときは入れられない

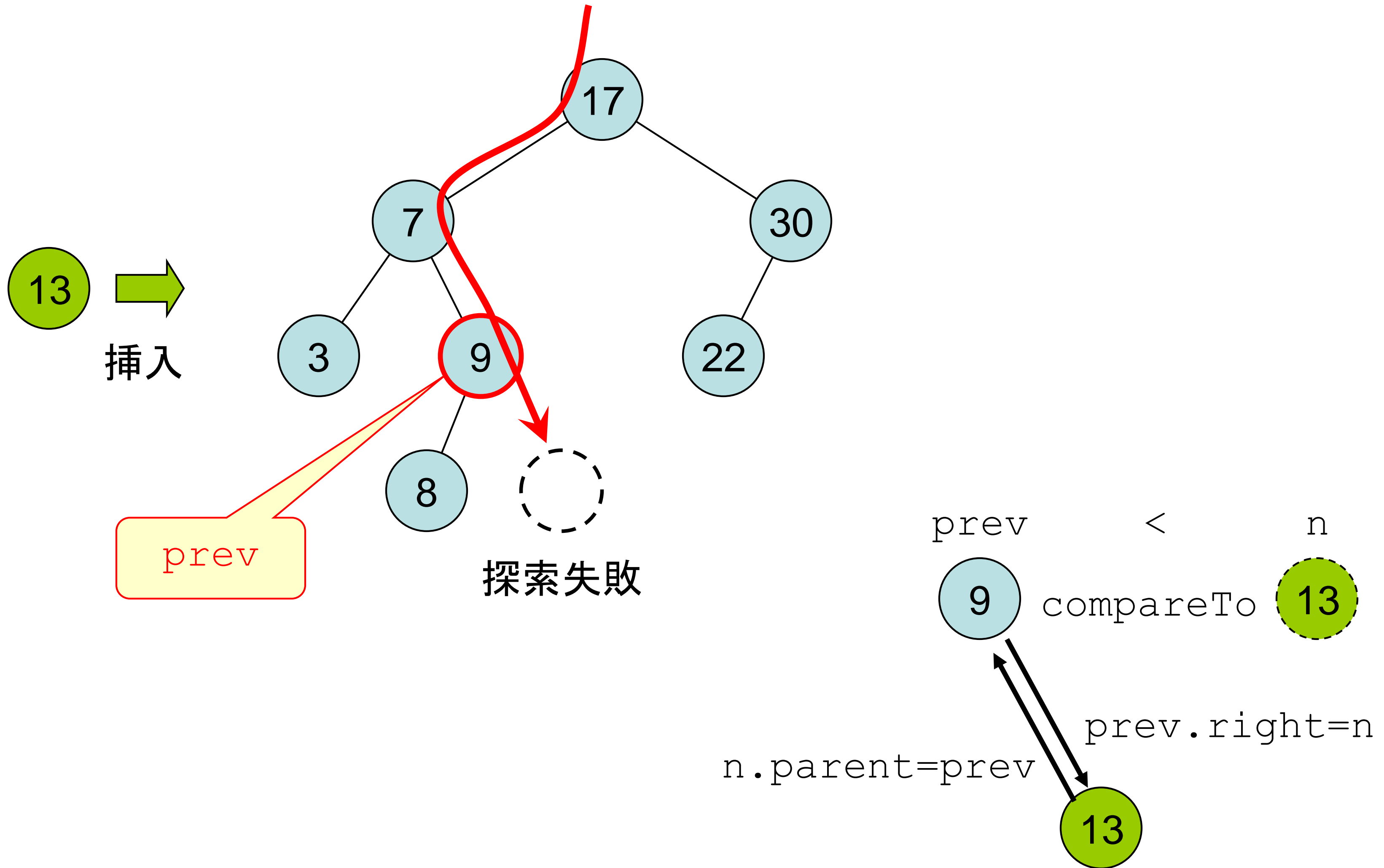
木が空だったとき

失敗したときはprevに直前のノードが入っている

失敗直前ノードとkeyを比較し、小さければ左につなぐ
そうでなければ右につなぐ

いずれの場合も、親はprevになる

挿入の実際

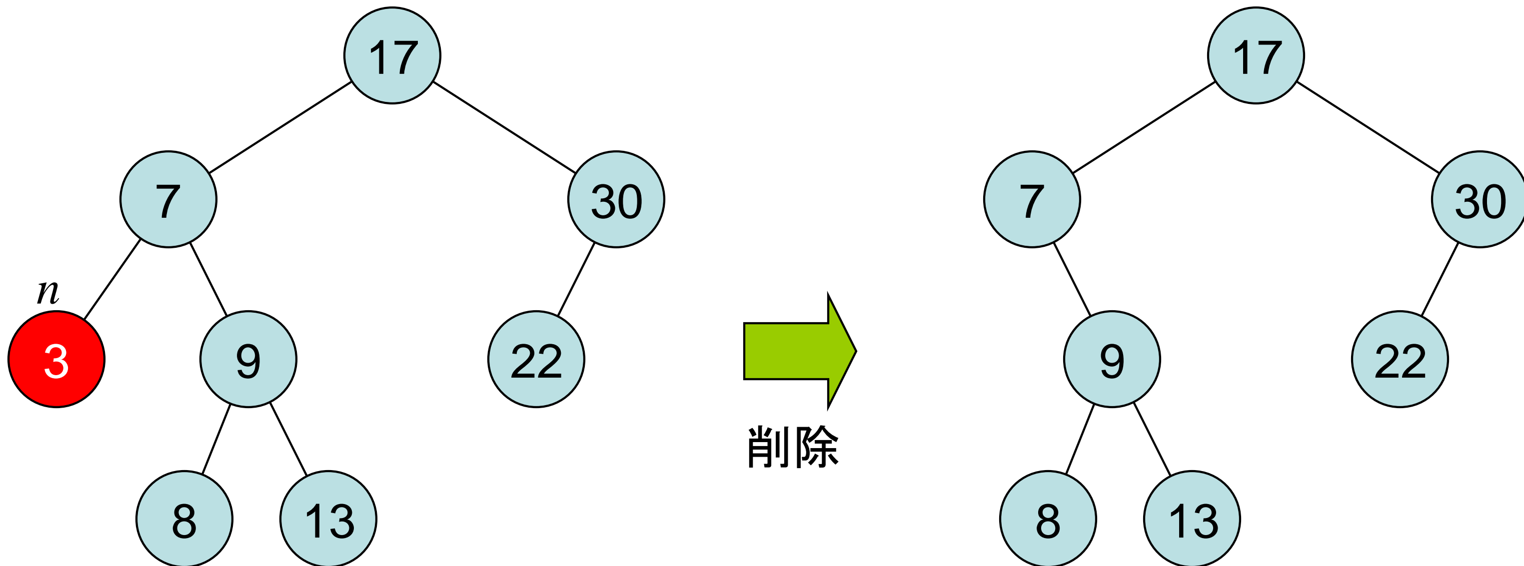


要素の削除1

- 要素を削除するとき
  - 二分探索木の構造を壊さないように
- 削除の手順
  1. 削除対象のノード n を探索
  2. 以下の場合によって処理が異なる
     - n が子ノードを持たない場合
     - n が片方の子ノードだけを持つ場合
     - n が両方の子ノードを持つ場合

要素の削除2

- 削除ノード n が子ノードを持たない場合
  - そのノードを取り去るだけ



葉ノードを取り去っても、性質は変化しない

削除の実装1(remove)

```
// 削除
void remove(BinTree *tree,void *key,int(*comp)(void*,void*)){
  if(!search(tree,key,comp)) return;
  // 子がない
  if(!prev->left && !prev->right){
    if(prev->parent==NULL)
      tree->root=NULL;
    else if(prev->parent->left==prev)
      prev->parent->left=NULL;
    else
      prev->parent->right=NULL;
  }
```

e を探索し、見つからなければ何もしない

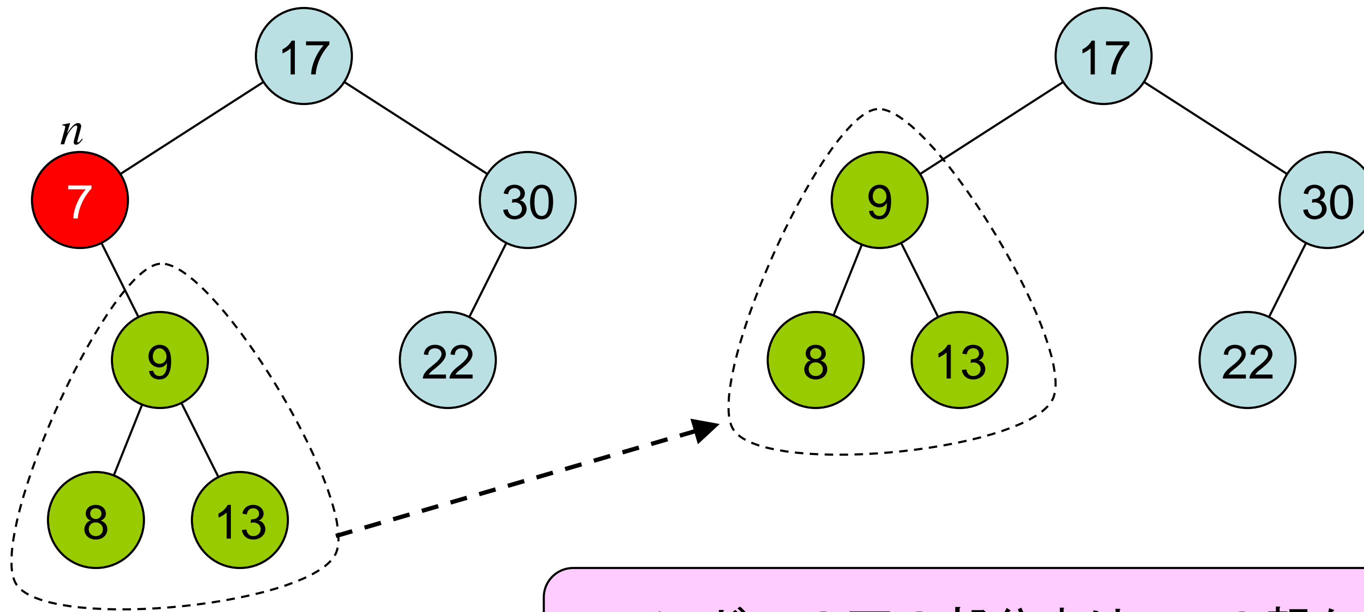
成功したときはprevにそのノードが入っている

ノードを木構造からはずす
親がなければ単一ノードなので木を空にする
親があれば、その親から見て自分の側を切る

続く

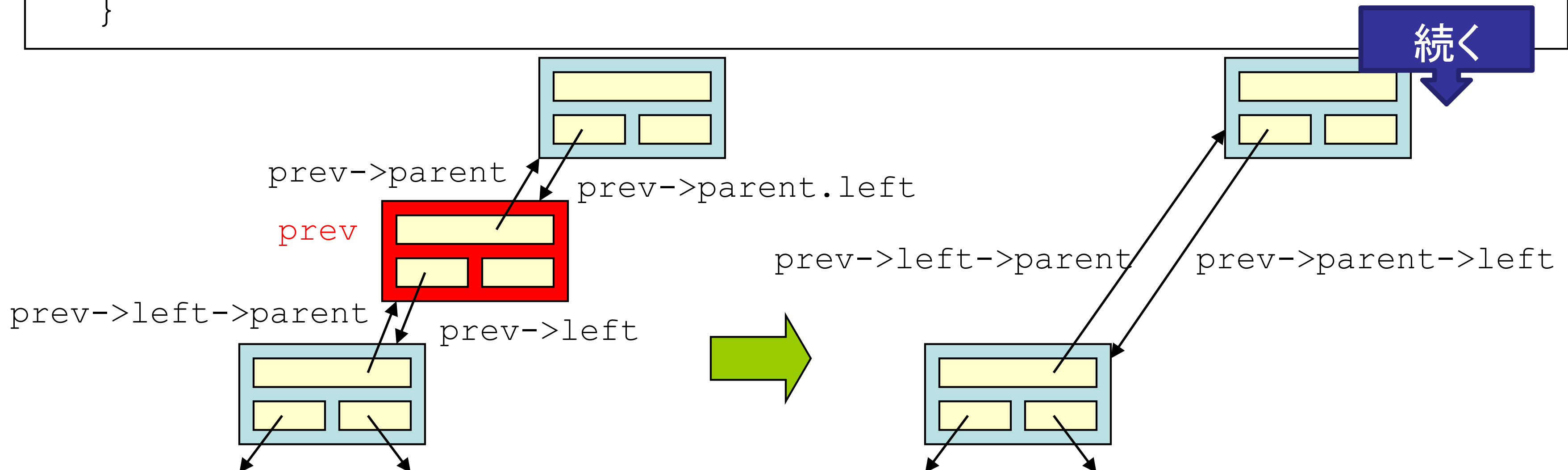
要素の削除3

- 削除ノード n が片方の子ノードだけを持つ場合
  - n の位置に、その子ノードを持ってくる



削除の実装2(remove)

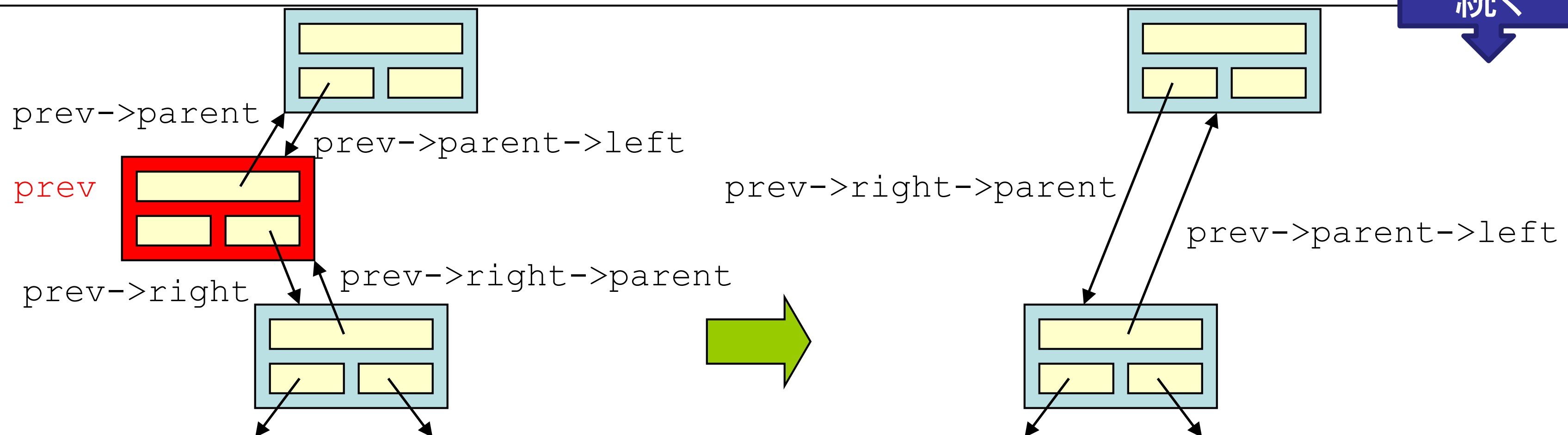
```
// 左だけ
else if(!prev->right){
  if(prev->parent==NULL)
    tree->root=prev->left;
  else if(prev->parent->left==prev)
    prev->parent->left=prev->left;
  else
    prev->parent->right=prev->left;
  prev->left->parent=prev->parent;
}
```



削除の実装3(remove)

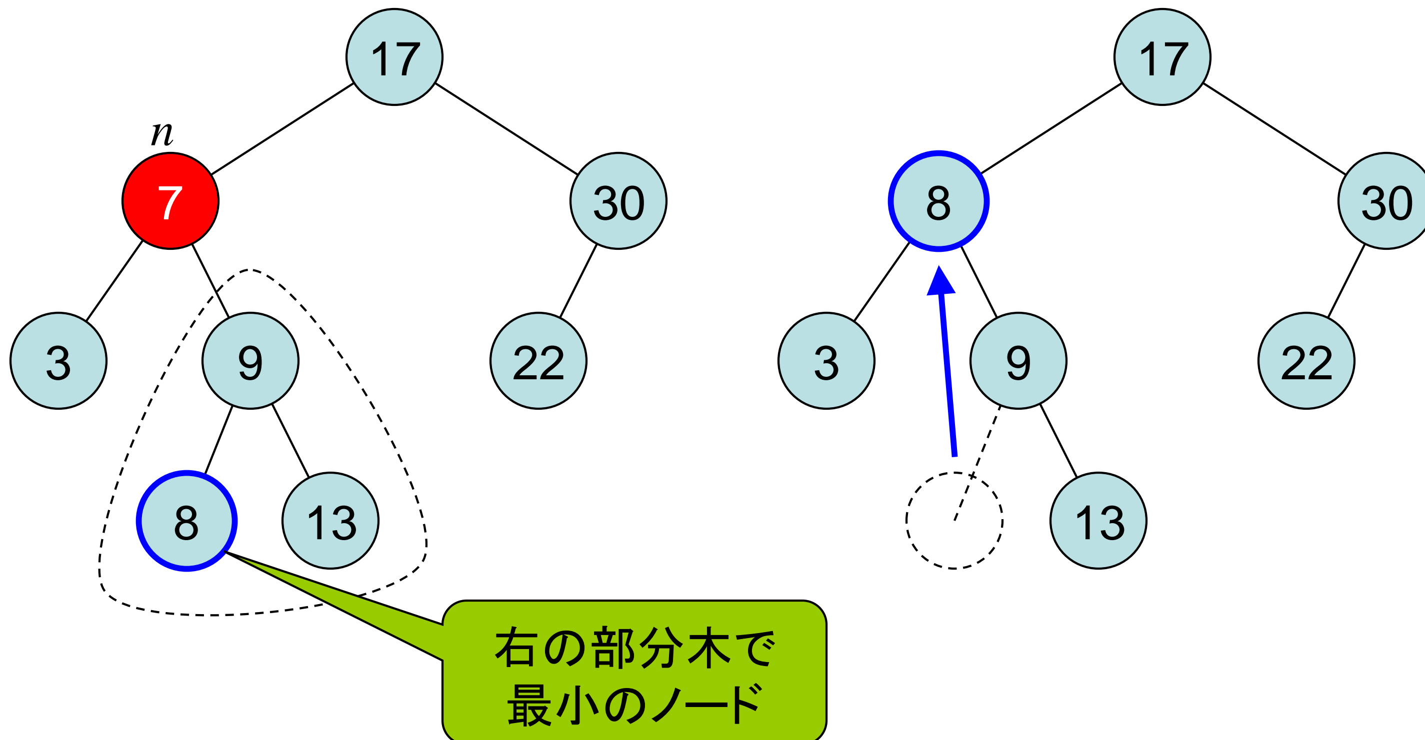
```
// 右だけ
else if(!prev->left){
  if(prev->parent==NULL)
    tree->root=prev->right;
  else if(prev->parent->left==prev)
    prev->parent->left=prev->right;
  else
    prev->parent->right=prev->right;
  prev->right->parent=prev->parent;
}
```

続く



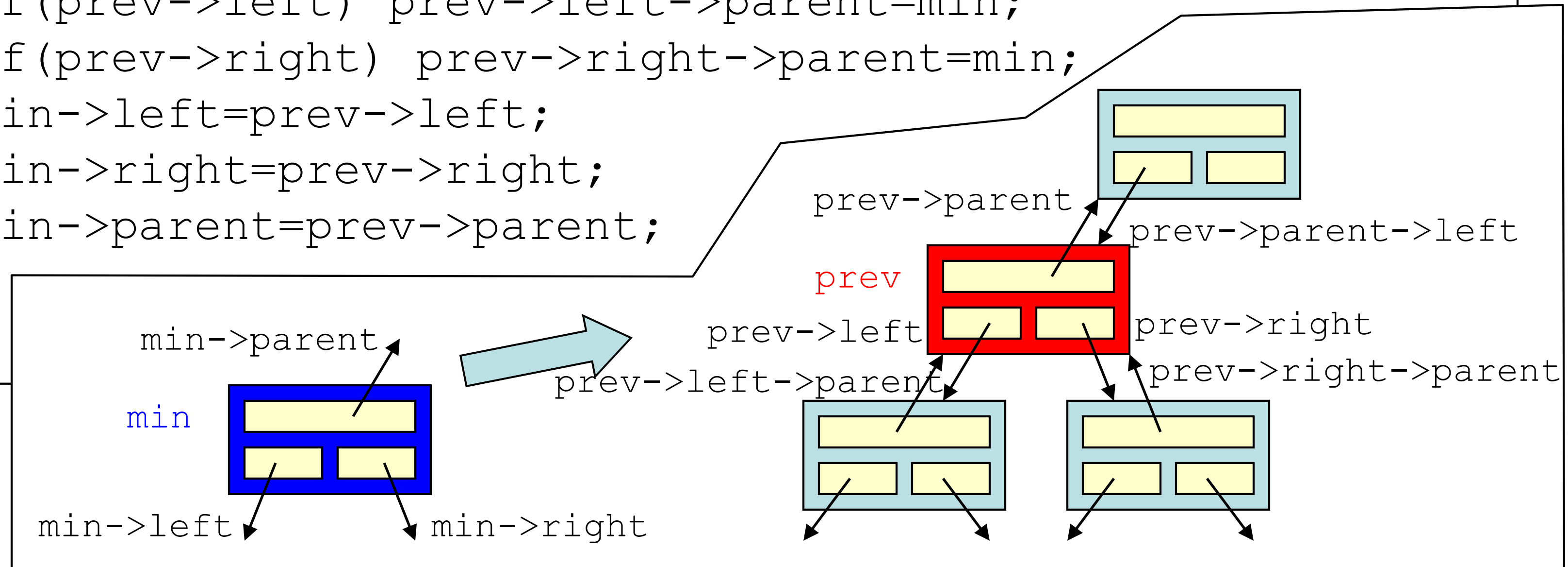
要素の削除4

- 削除ノード n が両方の子ノードを持つ場合
  - 右の部分木の最小ノードを削除して、それを n と置き換える
  - （左の部分木の最大ノードでも良い）



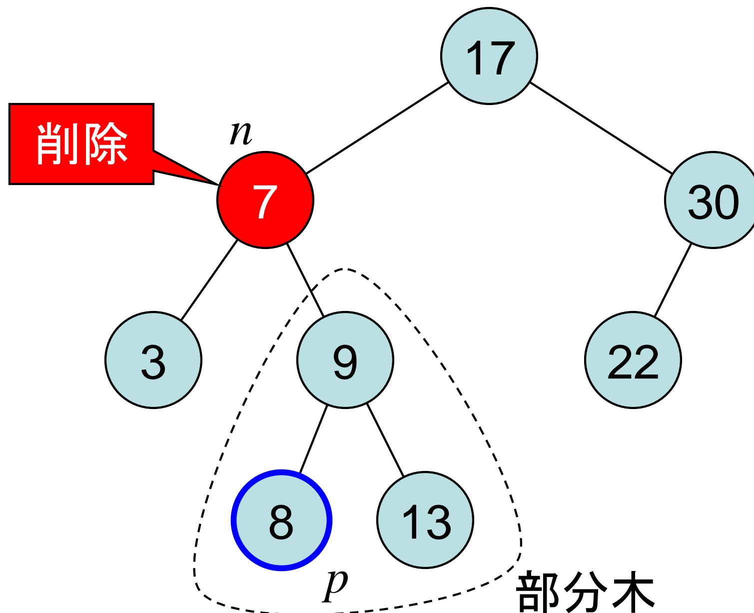
削除の実装4(remove)

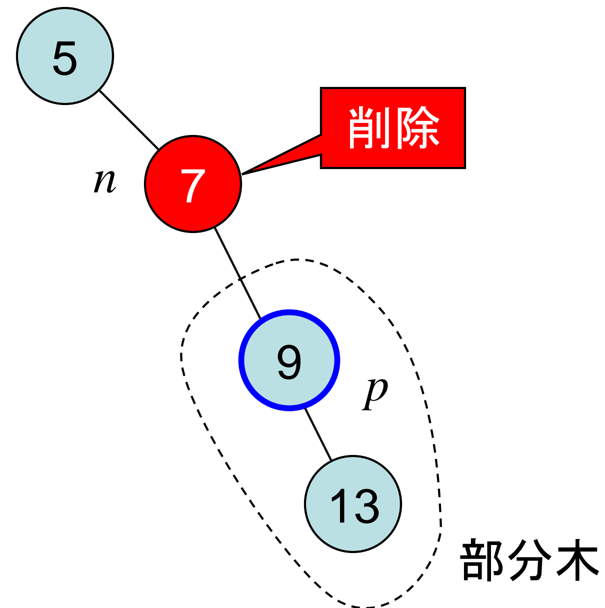
```
 // 両方
  else{
    BinTreeNode *min=removeMin(prev->right);
    if(prev->parent==NULL)
      tree->root=min;
    else if(prev->parent->left==prev)
      prev->parent->left=min;
    else
      prev->parent->right=min;
    if(prev->left) prev->left->parent=min;
    if(prev->right) prev->right->parent=min;
    min->left=prev->left;
    min->right=prev->right;
    min->parent=prev->parent;
  }
}
```



要素の削除5

- 部分木の中で最小のノードの探索
  - 根から左へたどる
  - 左へいけなくなったノードが最小
  - 最小ノードの削除は、「要素の削除1，2」と同様
    - 少なくとも右側のノードしかないため





削除の実装5(removeMin)

プロトタイプ宣言されていないので、
外部から使用できない関数

```
// 最小の要素を外して返す
BinTreeNode *removeMin(BinTreeNode *n){
  while(n->left)
    n=n->left;
  if(n->parent->left==n)
    n->parent->left=n->right;
  else
    n->parent->right=n->right;
  if(n->right)
    n->right->parent=n->parent;
  return n;
}

// 削除
void remove(...
```

左へ行けなくなるまで探索する(nに最小ノードが入る)

最小ノードを外して、その右側とつなぎかえる

外したノードを返す

TestBinSearchTree.cc

```
/***
 *** BinSearchTreeのテスト
 ***/
#include "BinSearchTree.h"
#include "PD.h"

// 比較
int compare(void *p1,void *p2){
  return ((PD*)p1)->age-((PD*)p2)->age;  // 年齢で比較
}

// 表示
void print(void *d){
  PD *pd=(PD*)d;
  printf("%s(%d)-",pd->name,pd->age);
}
```

TestBinSearchTree.cc

```
// メイン
int main(int argc,char **argv){
  PD *pd1;
  BinTree *tree=makeBinTree();
  insert(tree,makePD("山田",18),compare);
  insert(tree,makePD("森",55),compare);
  insert(tree,pd1=makePD("中村",33),compare);
  insert(tree,makePD("今井",60),compare);
  insert(tree,makePD("福元",44),compare);
  insert(tree,makePD("石田",27),compare);

  traverse(tree,1,print); printf("¥n");
  remove(tree,pd1,compare);
  traverse(tree,1,print); printf("¥n");
  insert(tree,pd1,compare);
  traverse(tree,1,print); printf("¥n");
}
```

実行結果

```
$ ./TestBinSearchTree
山田(18)-石田(27)-中村(33)-福元(44)-森(55)-今井(60)-
山田(18)-石田(27)-福元(44)-森(55)-今井(60)-
山田(18)-石田(27)-中村(33)-福元(44)-森(55)-今井(60)-
$
```

TestBinSearchTree2.cc

```
/***
 *** 二分探索木のテスト2
 ***/
#include "BinSearchTree.h"
#include "PD.h"
#include <string.h>
#include <time.h>

// 比較
int compare(void *d1,void *d2){
  PD *pd1=(PD*)d1,*pd2=(PD*)d2;
  int c=strcmp(pd1->name,pd2->name);
  if(c!=0) return c;            // 名前が違えば名前で比較
  return pd1->age-pd2->age;   // そうでなければ年齢で比較
}

// 表示
void print(void *d){
  PD *pd=(PD*)d;
  printf("%s(%d)-",pd->name,pd->age);
}
```

続く

TestBinSearchTree2.cc

```
// ランダムなデータで実験
void exp1(int max){
  int N=3;
  BinTree *tree=makeBinTree();
  for(int j=0;j<max;j++){
    char *name=(char*)malloc(N+1);
    for(int i=0;i<N;i++){
      if(i==0) name[i]='A'+rand()%26;
      else name[i]='a'+rand()%26;
    }
    name[N]=0;
    int age=rand()%90+5;
    insert(tree,makePD(name,age),compare);
  }
  freeBinTree(tree);
}
```

ランダムな名前
ランダムな年齢

続く

TestBinSearchTree2.cc

```
// 同じデータで実験
void exp2(int max){
  int N=3;
  BinTree *tree=makeBinTree();
  for(int j=0;j<max;j++){
    insert(tree,makePD("aaa",j),compare);
  }
  freeBinTree(tree);
}
```

同じ名前 “aaa”
1ずつ増える年齢

続く

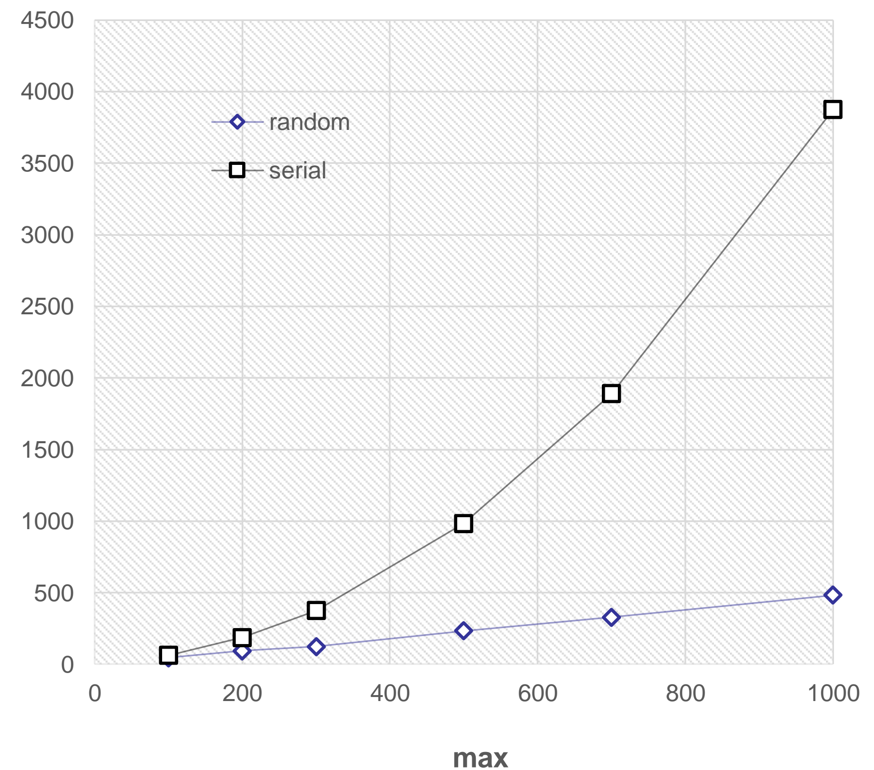
TestBinSearchTree2.cc

```
// メイン
int main(int argc,char **argv){
  int max=10,mode=0;
  if(argc>1) max=atoi(argv[1]);    // ノード数
  if(argc>2) mode=atoi(argv[2]);   // データの種類(0:random 1:serial)
  clock_t t1=clock();              // 開始時刻
  for(int i=0;i<1000;i++){
    switch(mode){
    case 0: exp1(max); break;
    case 1: exp2(max); break;
    }
  }
  clock_t t2=clock();              // 終了時刻
  printf("%d¥n",t2-t1);            // 経過時間を表示
  //traverse(tree,1,print); printf("¥n");
}
```

同じ二分木の構築を1000回繰り返す

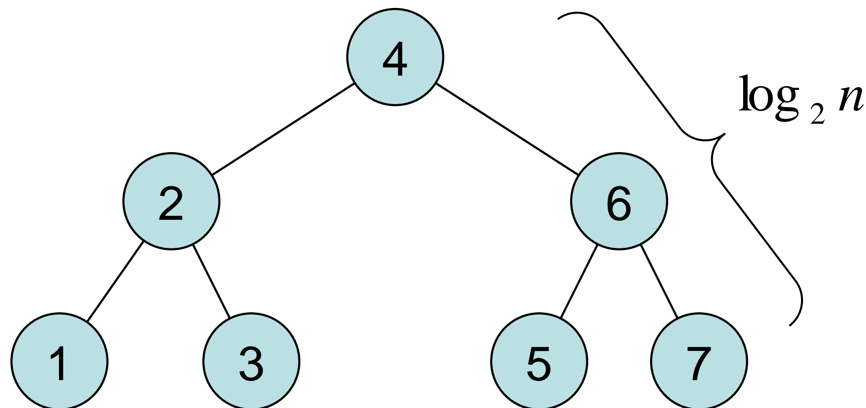
実行結果

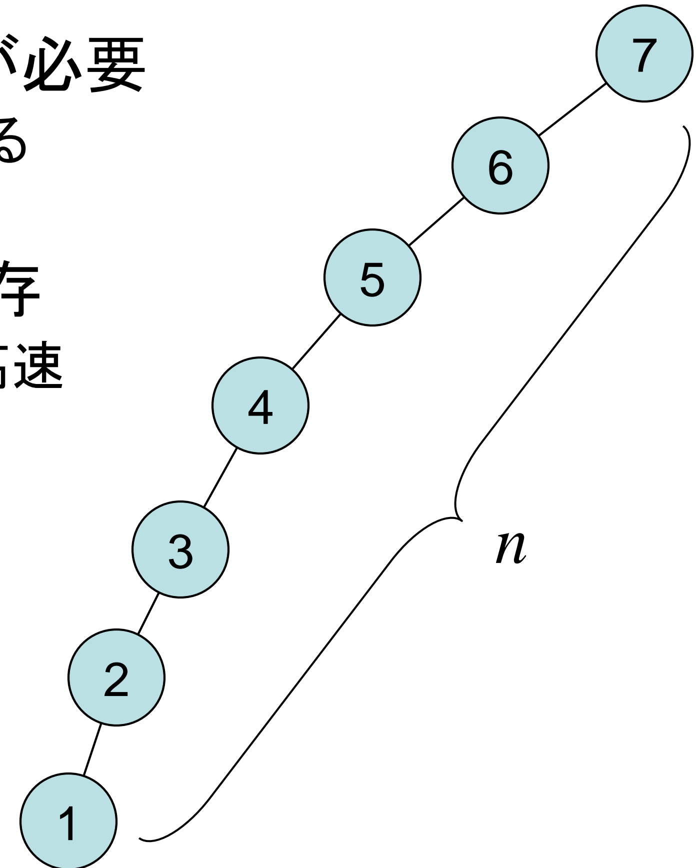
```
$ ./TestBinSearchTree2 100 0
46
$ ./TestBinSearchTree2 200 0
93
$ ./TestBinSearchTree2 300 0
124
$ ./TestBinSearchTree2 500 0
233
$ ./TestBinSearchTree2 700 0
328
$ ./TestBinSearchTree2 1000 0
483
$ ./TestBinSearchTree2 100 1
62
$ ./TestBinSearchTree2 200 1
187
$ ./TestBinSearchTree2 300 1
375
$ ./TestBinSearchTree2 500 1
984
$ ./TestBinSearchTree2 700 1
1890
$ ./TestBinSearchTree2 1000 1
3874
```



二分探索木の性質1

- 探索、挿入、削除のいずれも探索が必要
  - 探索の性能が全操作の性能を左右する
  - 探索が済めば、あとはつなぎ替えだけ
- 探索の要する手間は、木の形に依存
  - 木が低くて、枝分かれしているほうが高速





最良のパターン $O(\log n)$

完全二分木

最悪のパターン $O(n)$

二分探索木の性質2

- 木の形は何で決まるか？
  - データの挿入の順番
  - 最悪の場合
    - 小さい順、または大きい順に挿入された場合
  - しかし、これはあらかじめデータをランダムにシャッフルしておくことである程度回避できる
- 平均計算量は？
  - ランダムなデータに対して処理を行った場合、平均としては $O(\log n)$ になる

二分探索木の性質3

- ハッシュ法との探索性能の比較
  - ハッシュ法は $O(1)$ で可能
  - 二分探索木では、平均で $O(\log n)$、最悪で $O(n)$
  - 二分探索木に、いいところなし
- 最大、最小の要素を得るには効率が良い
  - ハッシュ法では $O(n)$
    - 順序情報が失われてしまっているため全部調べる必要がある
  - 二分探索では、平均で $O(\log n)$
  - 左端が最小、右端が最大のデータ
    - 二分探索木を通りがけ順でなぞると、データを昇順に整列できる

課題161219

- 二分探索木BinSearchTree.hと.ccを実装しなさい。
- 実験用プログラムTestBinSearchTree2.ccを実装し、以下のデータについて実験を行い、結果を示しなさい。
  - ランダムな名前と年齢のデータを、木に追加する (mode=0)
  - 同じ名前と順に並んだ年齢のデータを、木に追加する (mode=1)
  - データ数は max=100,200,300,500,700,1000

- 提出方法
  - 実装コードと実験結果を示したワード文書scXXXXXX-al161219.docxを作成し、メールに添付して送付すること。
  - ワード文書の先頭に、必ず学籍番号と氏名を記入すること。
  - 提出先：fuchida@ibe.kagoshima-u.ac.jp
  - メールタイトル：”アルゴリズム課題161219” ←厳守！
    - メール本文にも、学籍番号と氏名を必ず明記すること。
  - 期限：2016年12月25日（日） 24:00

二分探索木

終了