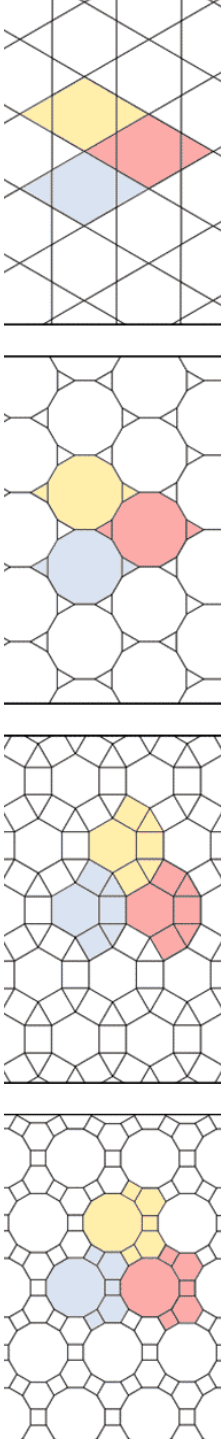
生産情報システム工学

#04 計算幾何学の基礎概念と基本的な手法

2015/05/13(水)
溝口 知広　准教授(居室：61-408室)
mizo@cs.ce.nihon-u.ac.jp



第1回レポート提出

- 締切：5/19(火)
- 提出場所：61-408室前のレポートボックス

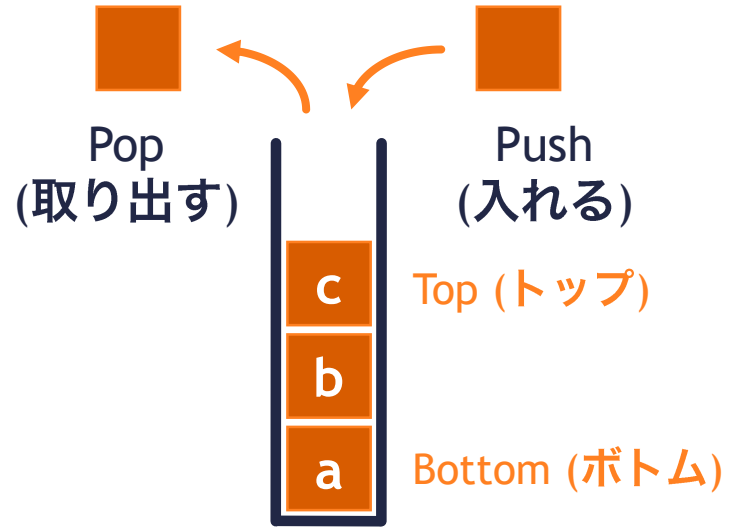
1.4 基本データ構造

1. リスト (D構造入門で学習済み)
2. スタックとキュー (D構造入門で学習済み)
3. ヒープ (今日の内容)
4. 2分探索木 (今日の内容)
5. 平衡2分探索木 (省略)

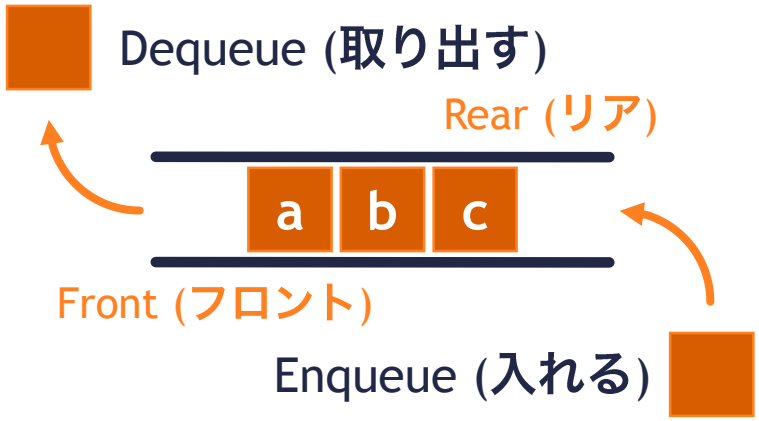
1.4.2 スタックとキュー(復習)

■ データを一時的に保存する際のデータ構造

スタック (Stack)	キュー (Queue)

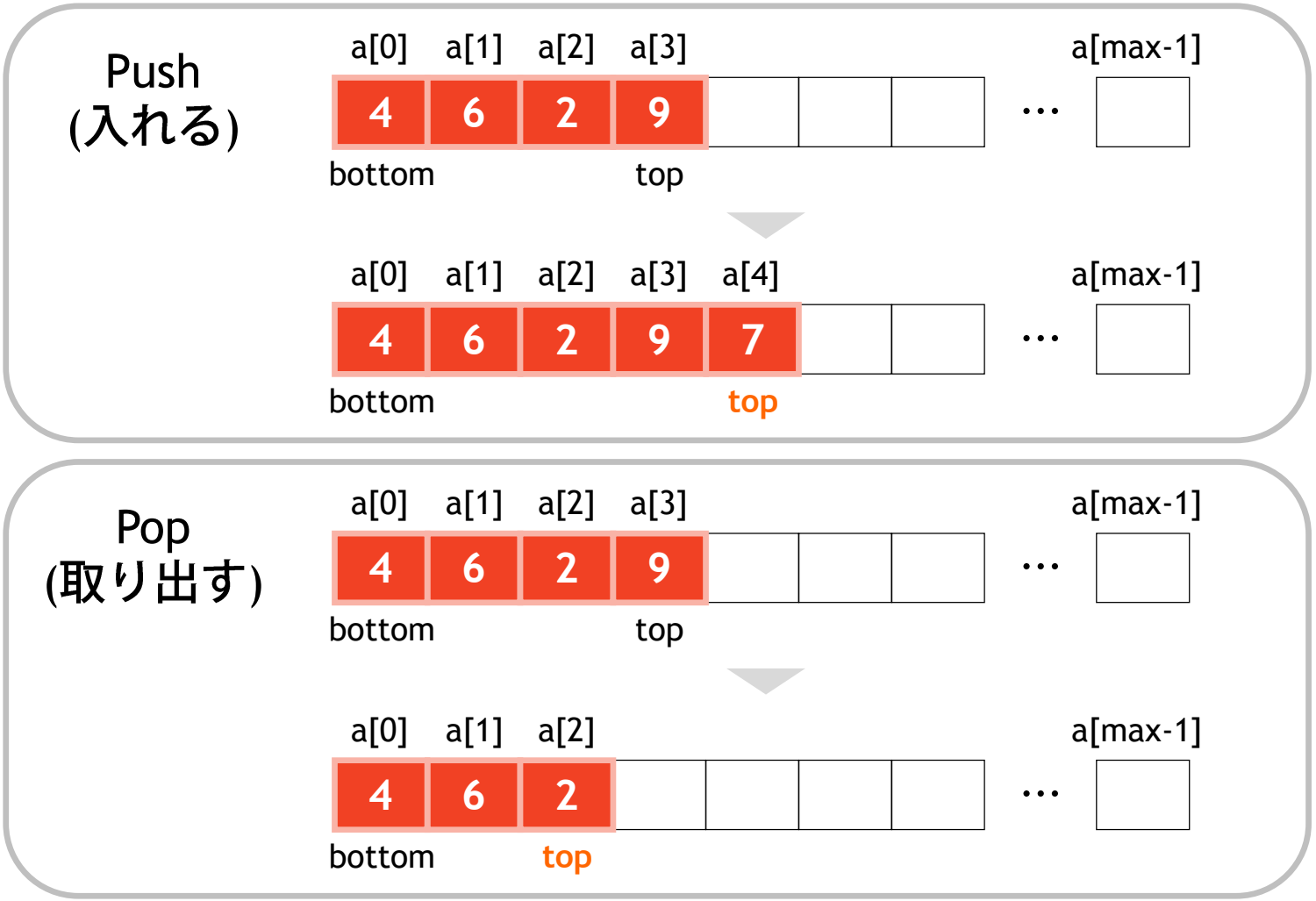


LIFO: Last In First Out
(最後に入ったものが最初に出る)



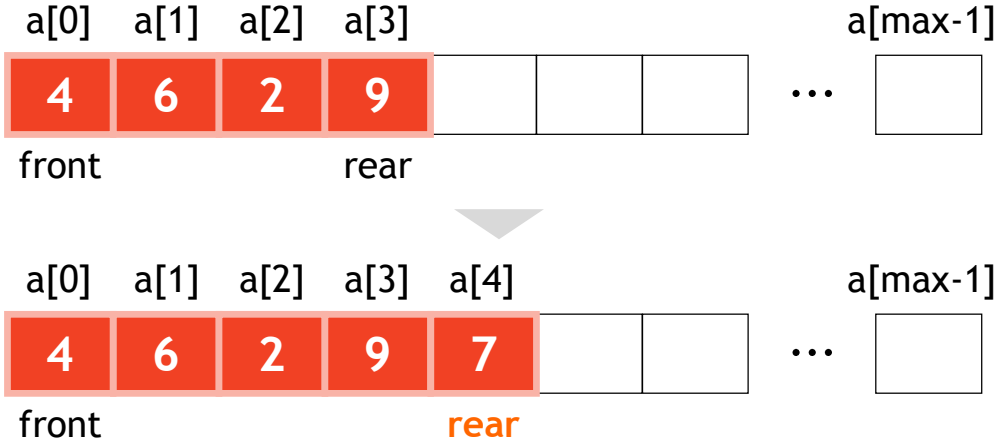
FIFO: First In First Out
(最初に入ったものが最初に出る)

1.4.2 スタック(復習)

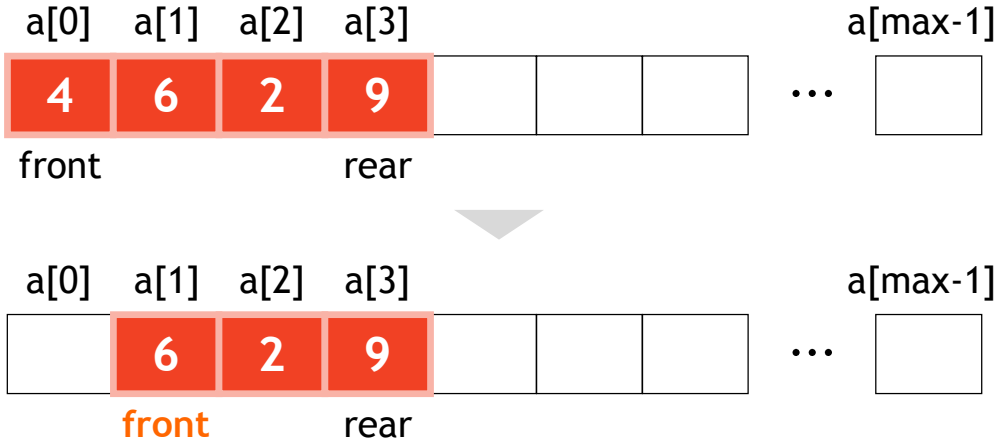

Push
(入れる)
a[0] a[1] a[2] a[3] a[max-1]
4 6 2 9
bottom top
a[0] a[1] a[2] a[3] a[4] a[max-1]
4 6 2 9 7
bottom top
Pop
(取り出す)
a[0] a[1] a[2] a[3] a[max-1]
4 6 2 9
bottom top
a[0] a[1] a[2] a[max-1]
4 6 2
bottom top

1.4.2 キュー(復習)

Enqueue
(入れる)



Dequeue
(取り出す)



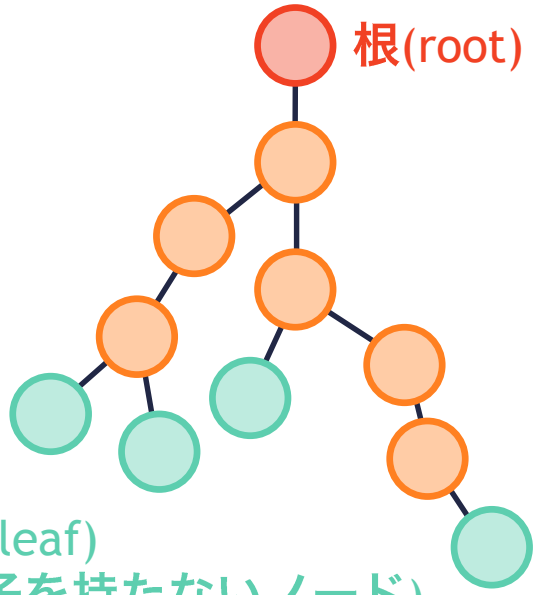
1.4.3 ヒープ

■ スタック・キュー
- 取り出す順番：挿入された順番で決まる
  - スタック：最後に入ったもの
  - キュー：最初に入ったもの

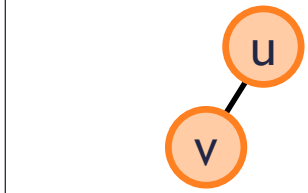
■ ヒープ(順位付きキュー，Priority Queue)
- 取り出す順番：挿入された順番と無関係
- 最大，または最小のものを取り出す

1.4.3 ヒープ

- 木(Tree)
  - いくつかのノード(頂点，節点)とそれらを結ぶエッジ(枝，辺)から構成される



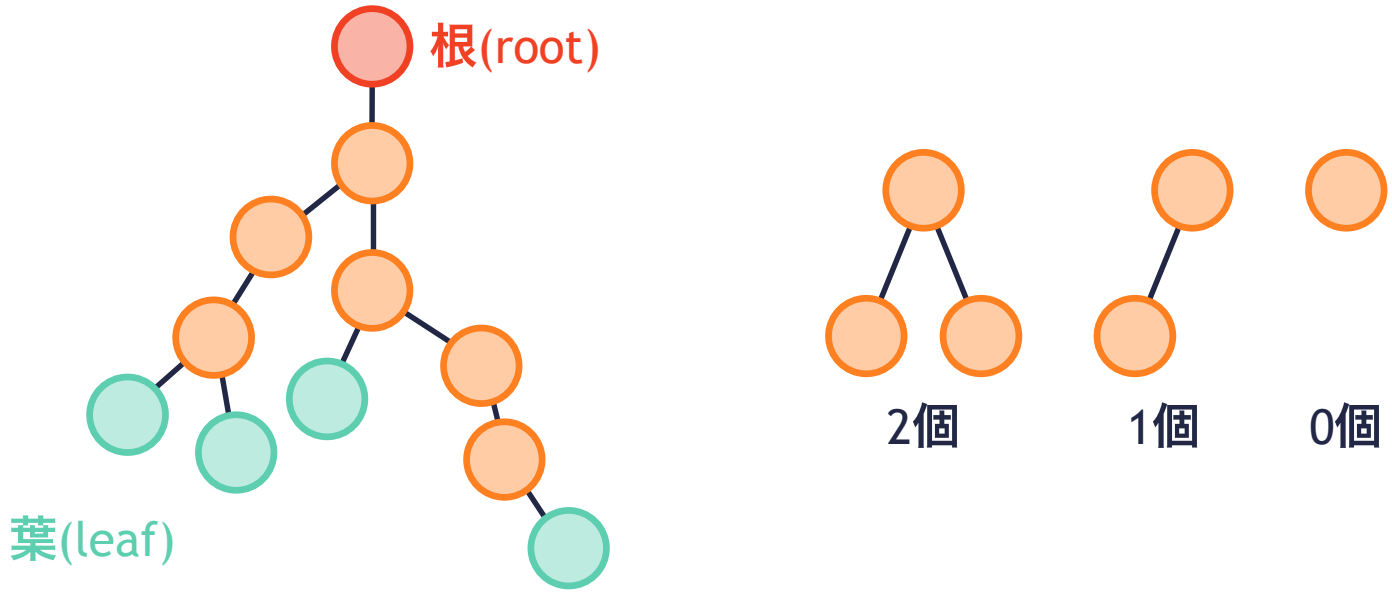
高さ(=枝の数)
根から葉への経路の中で，最も長いものの長さ
(この例の場合5)



uはvの親，vはuの子

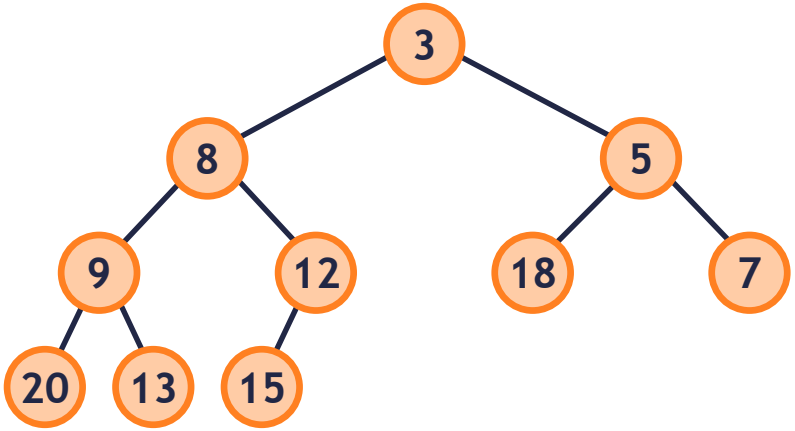
1.4.3 ヒープ

- 2分木(Binary Tree)
  - どのノードも2個以下の子を持つ木(3個以上はだめ)



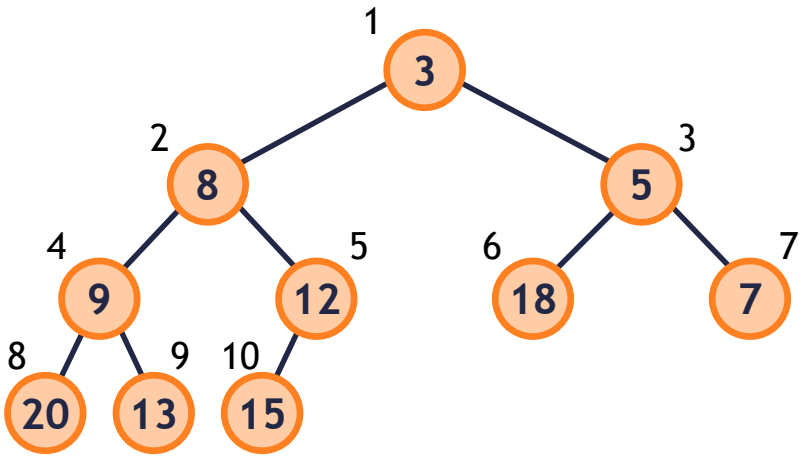
1.4.3 ヒープ

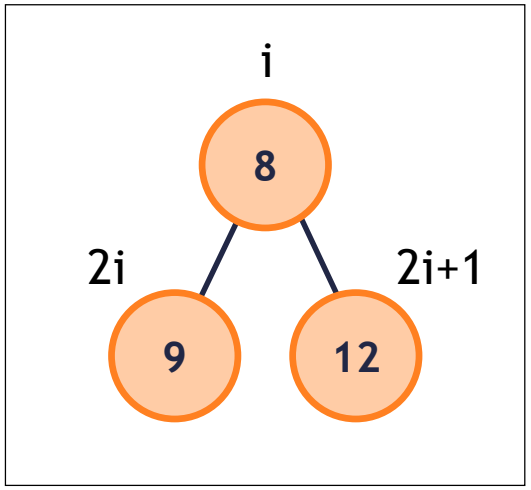
- ヒープは2分木の一種
  - 各ノードの要素がその全子孫より小さいか等しい
  - 最小の要素は常に木の根に蓄えられる



1.4.3 ヒープ

- ヒープは2分木の一種
  - 一般に，頂点vに割り当てられた要素が配列のi番目ならば，左の子は2i番目，右の子は2i+1番目に入る





1.4.3 ヒープ

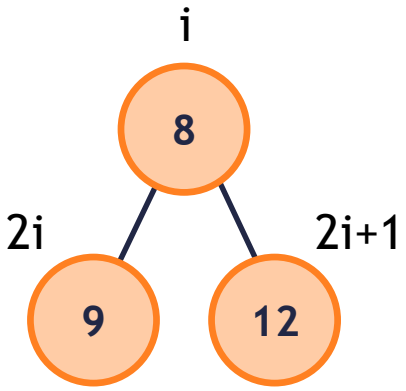
- 主な基本操作
  - 挿入(データの追加)
  - 削除(データの取り出し)

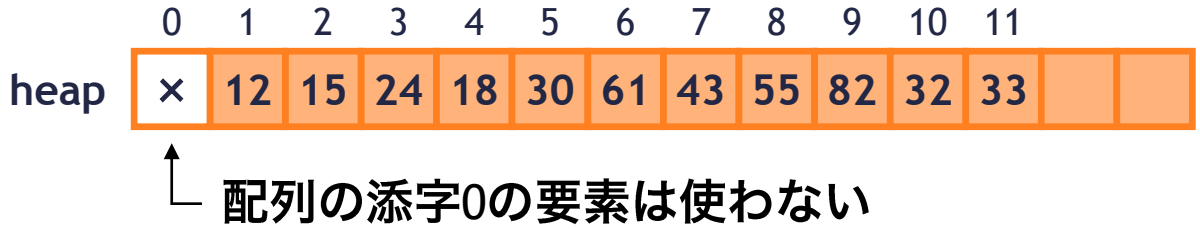
  - スタックの場合，プッシュとポップ
  - キューの場合，エンキューとデキュー

1.4.3 ヒープ

■ ヒープを実現する構造体

```
#define hmax 100

struct heap {
    int box[hmax+1];   //   データ
    int size;          //   データの個数
};
```





1.4.3 ヒープ

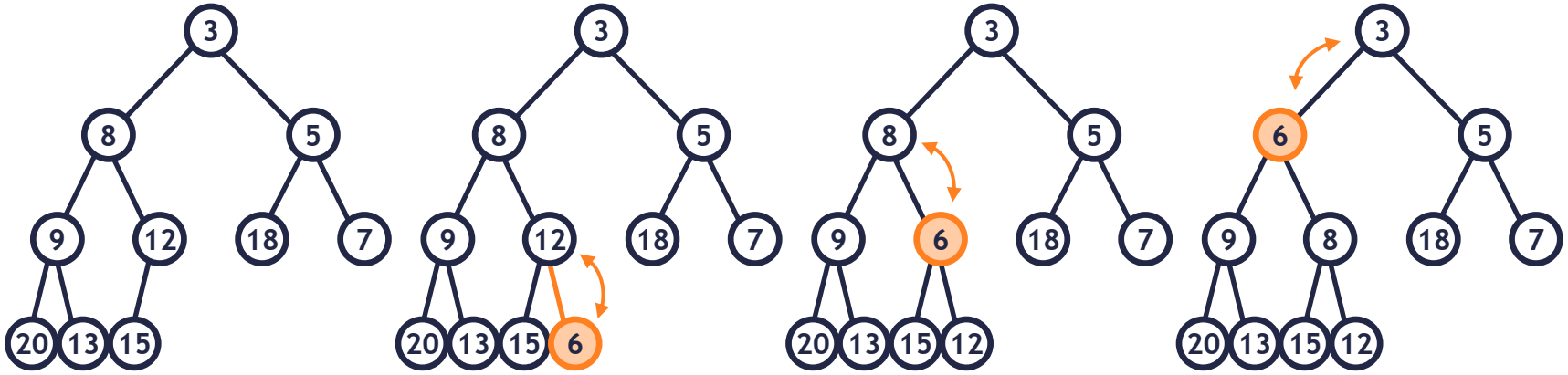
1	2	3	4	5	6	7	8	9	10			
3	8	5	9	12	18	7	20	13	15			

1	2	3	4	5	6	7	8	9	10	11		
3	8	5	9	12	18	7	20	13	15	6		

■ 挿入：insert(6)

1. 新たに挿入する要素を配列の末尾に入れる
2. 挿入要素とその親(12)で，ヒープ条件が成立するか調べる
3. もし成立しなければ，親と子を入れ替える
4. 終了条件：①ヒープ条件が成立，②挿入要素が根になる

繰返し（2.〜3.）



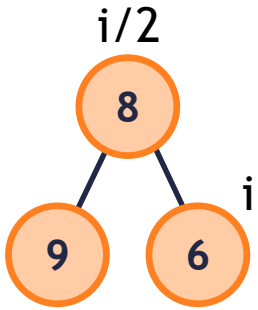
初期状態

6を入れる
12と6で条件を
調べる

12と6を入れ替える
8と6で条件を
調べる

8と6を入れ替える
3と6で条件が成立
終了

1.4.3 ヒープ



＊データを下から上へ移動させる
i ← i/2 (iは奇数でも偶数でもOK)
注意：int型では，5/2→2になる

■ 挿入プログラム

```
void insert( struct heap *h, int item )
{
  // 配列が満杯でなければ
  if( h->size < hmax ){
    h->size++;              // データ数+1
    int i = h->size;        // 末尾の添字を保存
    h->box[i] = item;       // 末尾にデータを代入
    // 根ではなく，かつ，子が親よりも小さければ
    while( 1<i && h->box[i] < h->box[i/2] ){
      swap( &h->box[i], &h->box[i/2] );  // 親子を入れ替える
      i /= 2;                            // 親の位置へ移動(上へ)
    }
    printf("%dを挿入しました．¥n",item);
  }
  else
    printf("満杯で挿入できません¥n");
}
```

1.4.3 ヒープ

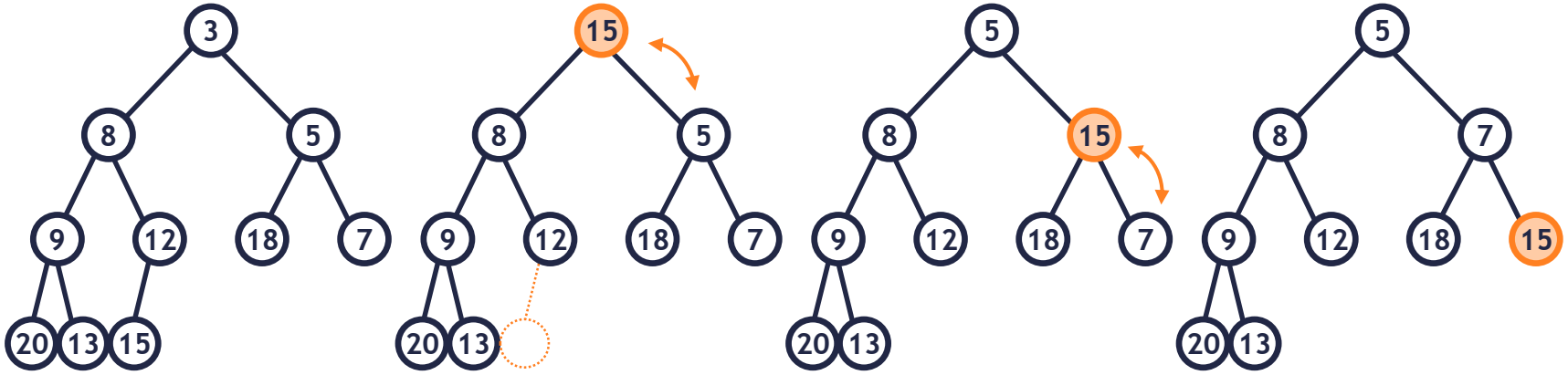
1	2	3	4	5	6	7	8	9	10
3	8	5	9	12	18	7	20	13	15

1	2	3	4	5	6	7	8	9
15	8	5	9	12	18	7	20	13

■ 削除：deletemin()

1. 末尾の要素を先頭に書き込む(最小要素の削除)
2. 書き込んだ要素とその子でヒープ条件が成立するか調べる
3. もし成立しなければ，左右の子の小さい方と入れ替える
4. 終了条件：①ヒープ条件成立，②書き込んだ要素が葉になる

繰返し



初期状態

15を根に入れる
15と5で条件を
調べる

15と5を入れ替える
15と7で条件を
調べる

15と7を入れ替える
15が葉なので条件が
成立，終了

1.4.3 ヒープ

＊データを上から下へ移動させる
i ← 2i(左の子)，i ← 2i+1(右の子)

■ 削除プログラム

```
void deletemin( struct heap *h )
{
  if( 0 < h->size ){

    int i,k;
    i = 1;                                  // 根に初期化する
    int tmp = h->box[1];                    // 削除する要素を保存

    h->box[1] = h->box[ h->size ];          // 末尾を先頭に上書き
    h->size--;                              // データ数-1
    while( 2*i <= h->size ){                // 子があれば
      // 左右の子の小さい方を選ぶ
      k = 2*i;                              // 左の子の添字を保存
      if( k < h->size && h->box[k+1] < h->box[k] )
        k++;                                // 右の子の方が小さければ添字を+1
      if( h->box[i] <= h->box[k] )          // 条件を満たせば終了
        break;
      swap( &h->box[i], &h->box[k] );       // 満たさなければ親子を入れ替え
      i = k;                                // 子へ移動
    }
    printf("最小値%dを削除しました．¥n", tmp);
  }
  else
    printf("データがありません¥n");
}
```

1.4.3 ヒープ

■ 探索プログラム

- 最小要素を取り出すのみ(添字1の要素)

```
int findmin( struct heap *h )
{
    if( 0 < h->size){
        printf("最小値%dを探索しました. ¥n", h->box[1]);
        return h->box[1];
    }
    else{
        printf("データがありません¥n");
        return -1;
    }
}
```

1.4.3 ヒープ

■ main関数

```
int main( void )
{
  struct heap h;
  h->size = 0;              //  ヒープの初期化
  int mode, item;
  while(1){
    printf("モードを入力：(1)挿入, (2)削除, (3)探索, (4)終了 --> ");
    scanf_s("%d", &mode);
    if( mode == 1 ){
      printf("挿入する値を入力：");
      scanf_s("%d", &item);
      insert( &h, item );
    }
```

1.4.3 ヒープ

■ main関数(続き)

```
    else if( mode == 2 )
      deletemin( &h );
    else if( mode == 3 )
      findmin( &h );
    else if( mode == 4 )
      break;
    else{
      printf("入力された値が不正です¥n");
    }
  }
  return 0;
}
```

1.4.4 2分探索木

■ 復習：2分探索(データ構造入門で学習済み)

1.4.4 2分探索木

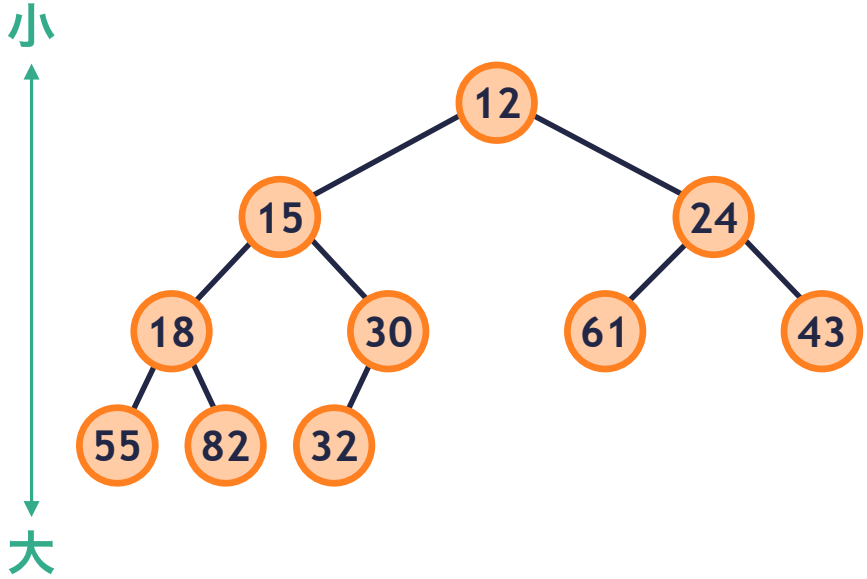
- 復習：2分探索(データ構造入門で学習済み)
  - あらかじめデータを整列させておき，配列の中央要素との比較と探索範囲を縮小を繰り返し行う
  - 探索は高速に行える($O(\log N)$)
  - データの挿入・削除に時間がかかる($O(N)$)

- 2分探索木
  - データの挿入・削除も高速に行える($O(\log N)$)
  - アルゴリズム理論における基本的なデータ構造

1.4.4 2分探索木

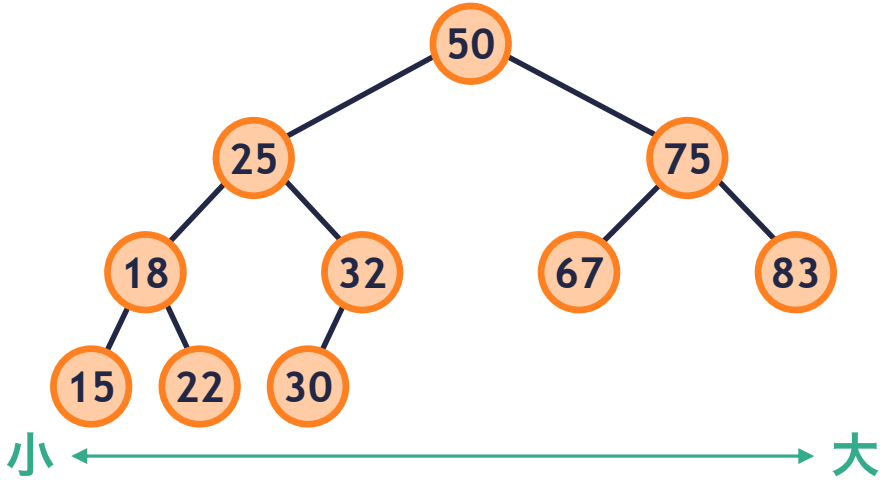
■ ヒープと2分探索木

ヒープ



各頂点の要素はすべての子孫の
要素よりも小さいか等しい

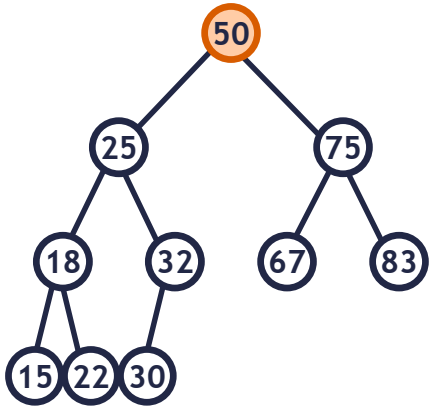
2分探索木



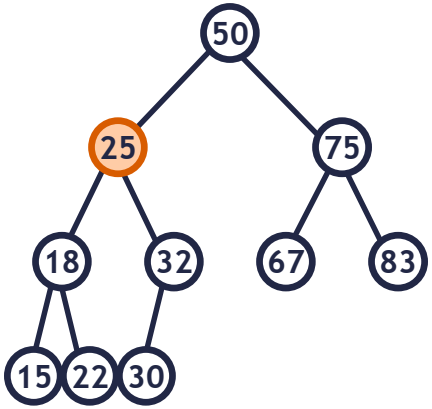
各頂点の要素は左部分木の
すべての要素より大きい

1.4.4 2分探索木

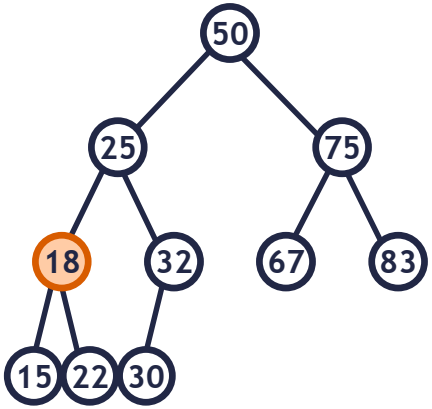
■ 探索の例1 (木に含まれるkey=22を探索する場合)



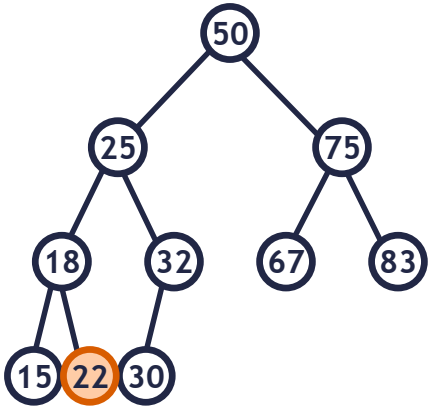
根からスタートする
key<50，左部分木へ



key<25，左部分木へ



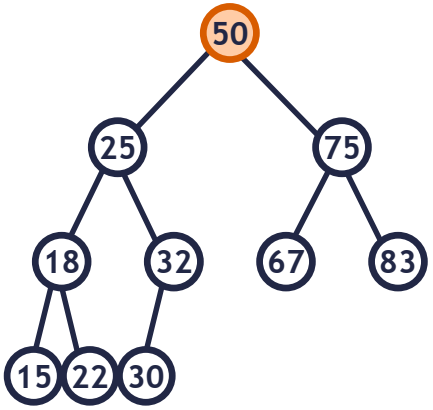
18<key，右部分木へ



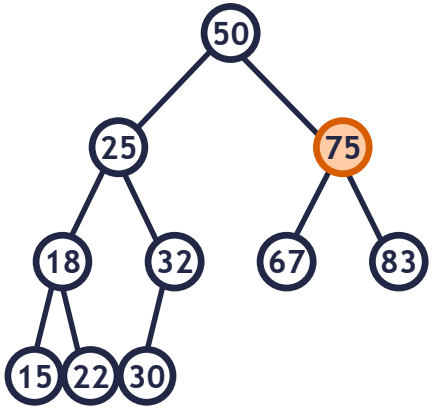
key==22
見つかったので終了

1.4.4 2分探索木

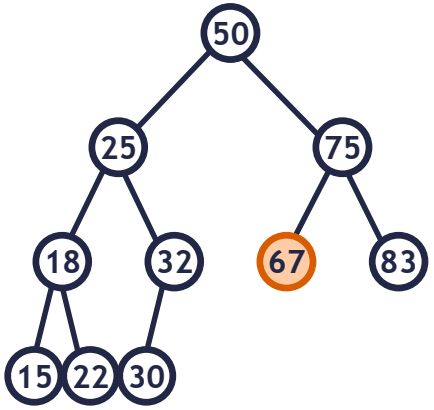
■ 探索の例2 (木に含まれないkey=68を探索する場合)



根からスタートする
50<key，右部分木へ



key<75，左部分木へ



67<key，葉に到達しても
見つからないので終了

1.4.4 2分探索木

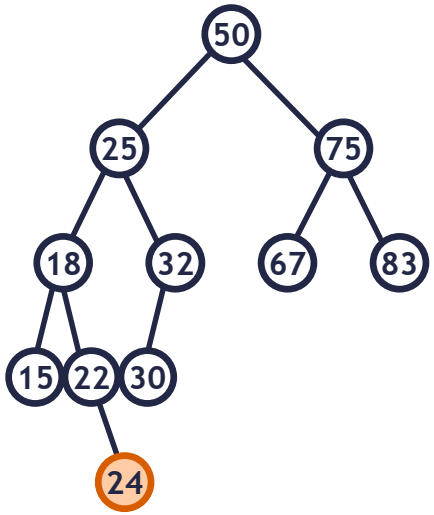
■ 探索の例

1. 初期化：根からスタートする
2. 比較：探索するデータkeyを現在訪れているノードの要素と比較する
3. 移動：keyの方が小さければ左部分木へ，keyの方が大きければ右部分木へ移動する
4. 終了条件(1)：keyと等しい要素が見つかれば終了する
5. 終了条件(2)：葉に到達しても見つからなければ，この木にkeyは含まれないので終了する

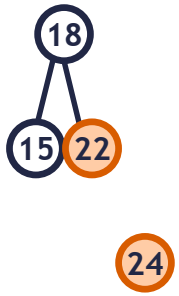
繰返し

1.4.4 2分探索木

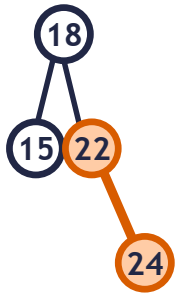
- 挿入の例 (24を新たに追加する場合)
  1. 探索の場合と同様に，根から比較と移動を繰り返す
  2. 最後に訪れたノードにxを新たな子として追加する
     1. 新たな子のためのメモリを割り当てる
     2. データを追加する
     3. 親子関係を更新する



1)葉ノードに到達



2-1)メモリ割り当て
2-2)データの追加



2-3)親子関係の更新

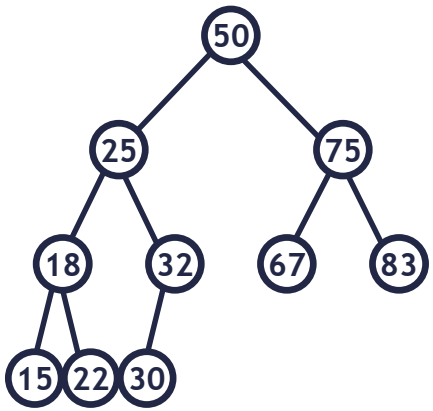
1.4.4 2分探索木

- 削除の例
  1. 探索の場合と同様に，削除するノードへ移動する
  2. ノードを削除する

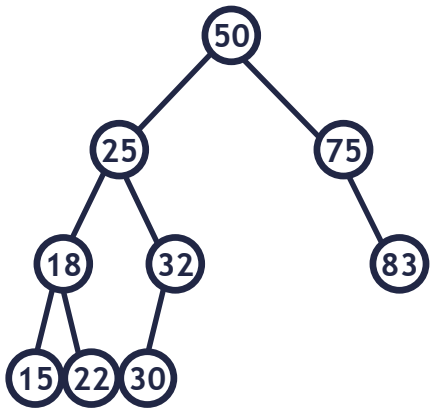
     Case1：削除するノードが葉の場合 → 葉を削除する

     Case2：葉ではなく，1つの子を持つ場合 → 子で置き換える

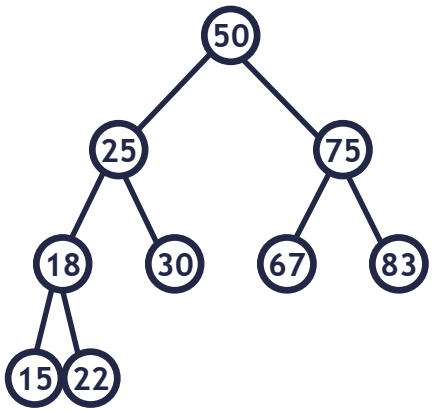
初期状態



Case1：delete(67)

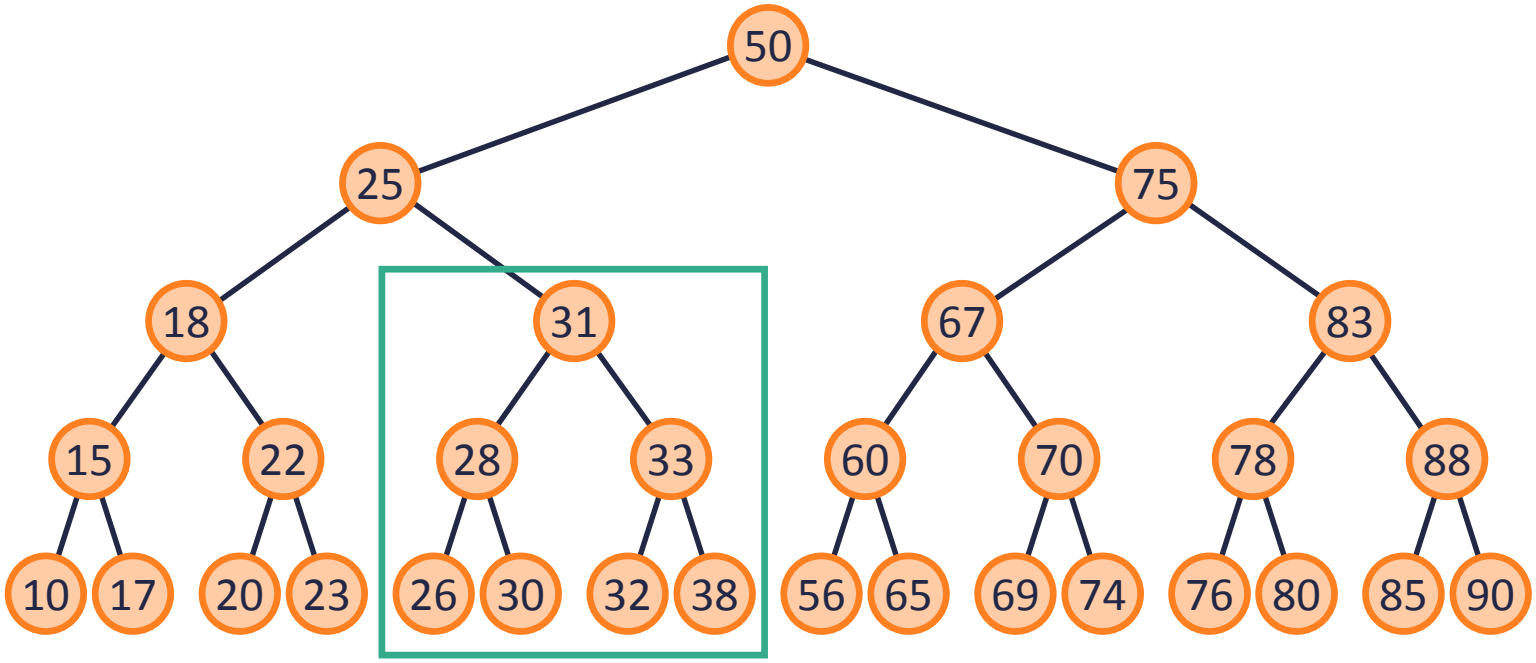


Case2：delete(32)



1.4.4 2分探索木

- 削除の例
  - あるノードの要素の次に大きな要素は，右部分木の左を繰り返し辿った先のノードの要素
  - 例：25の次に大きな値は，その右部分木の左端の26



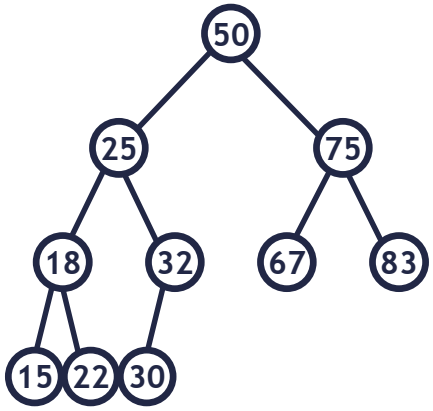
1.4.4 2分探索木

- 削除の例
  1. 探索の場合と同様に，削除するノードへ移動する
  2. ノードを削除する

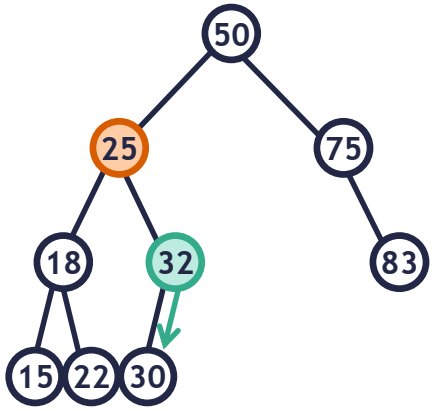
     Case3：葉ではなく，2つの子を持つ場合

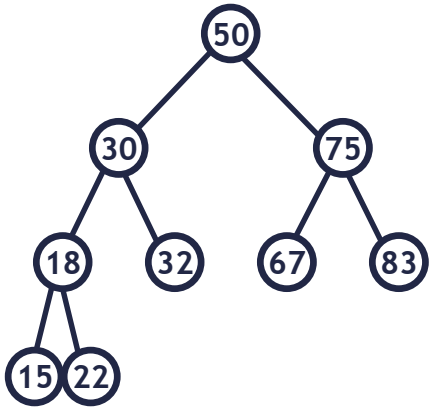
     1. 削除ノードの右の子(32)を出発点とし，左の子を繰り返し辿る (削除ノードの次に大きな要素を見つける)
     2. 到達したノードの要素(30)を削除ノードに上書き

初期状態



Case3：delete(25)





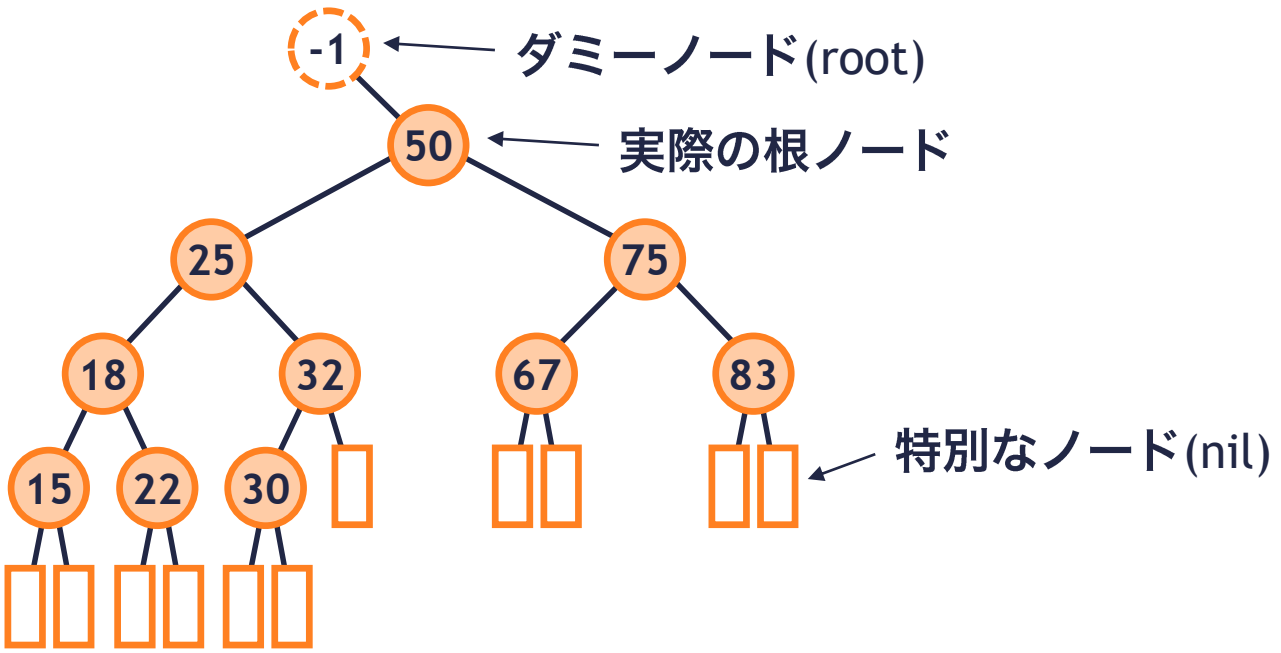
1.4.4 2分探索木

■ ノードを表す構造体

```
struct node{
  int data;             // 値
  struct node *lson;    // 左の子
  struct node *rson;    // 右の子
};
```

1.4.4 2分探索木

- 初期化
  - 基本操作プログラムを簡単にするための工夫
    1. ダミーノード(node root)のみの探索木を生成する
    2. 特別なノード(node nil)を宣言し全ての葉の子にする



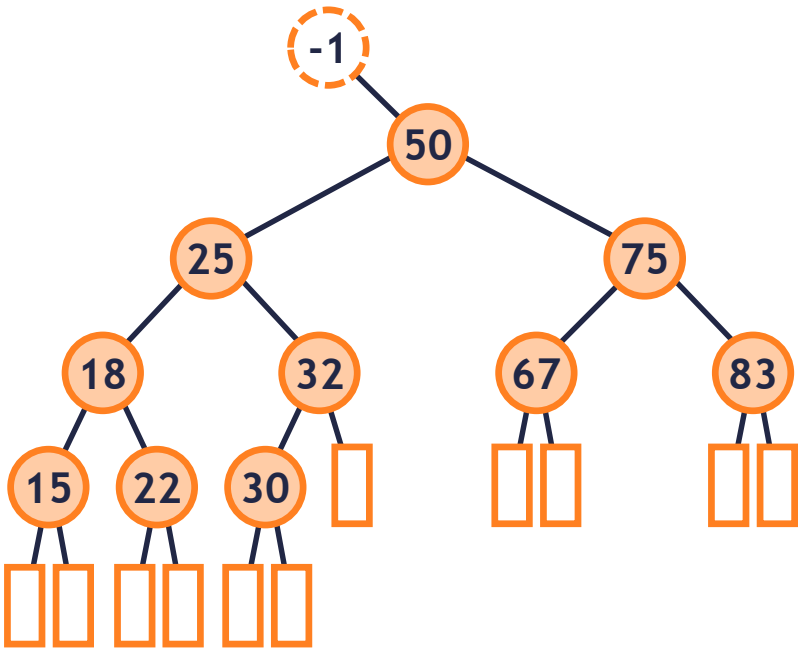
1.4.4 2分探索木

■ 初期化プログラム

```
struct node nil;        // 特別なノード
struct node root;       // ダミーノード

int initialize()
{
  root.data = -1;
  root.lson = &nil;
  root.rson = &nil;

  return 1;
}
```

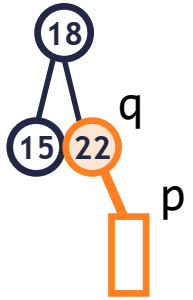
1.4.4 2分探索木

■ 探索プログラム




```
int find( int key )
{
  struct node* p;  // 現在位置を示す
  p = &root;       // 根で初期化する

  // 葉に到達するまで繰り返す
  while( p != &nil ){
    // 一致すれば終了
    if( p->data == key ){
      printf("見つかりました¥n");
      return 1;
    }
    // 現在のノードより小さい場合
    else if( key < p->data )
      p = p->lson;       // 左へ
    // 現在のノードより大きい場合
    else
      p = p->rson;       // 右へ
  }
  printf("見つかりませんでした¥n");
  return 0;
}
```

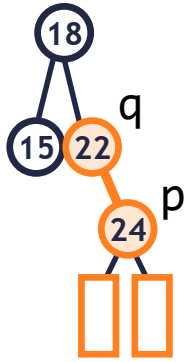
1.4.4 2分探索木



■ 挿入プログラム

```
int insert( int x )
{
  struct node* p = &root; // 初期位置を根に設定
  struct node* q;

  // 探索
  // 1)葉に到達するまで繰り返す
  while( p != &nil ){

    q = p;        // 現在位置を保存

    if( x == p->data ){
      printf("%dはすでに入力されています¥n",x);
      return 0;
    }

    if( x < p->data )     // 挿入する値が現在ノードより小さければ
      p = p->lson;        // 左部分木へ移動
    else                  // 挿入する値が現在ノードより大きければ
      p = p->rson;        // 右部分木へ
  }
```

1.4.4 2分探索木



■ 挿入プログラム

```
    // 2-1)メモリ割り当て
    p = (struct node*)malloc( sizeof(struct node) );

    // エラー処理
    if( p == NULL )  return 0;

    // 2-2)データの追加
    p->data = x;
    p->lson = &nil;
    p->rson = &nil;

    // 2-3)親子関係の更新
    if( p->data < q->data )
      q->lson = p;    // 左の子
    else
      q->rson = p;    // 右の子

    return 1;
}
```

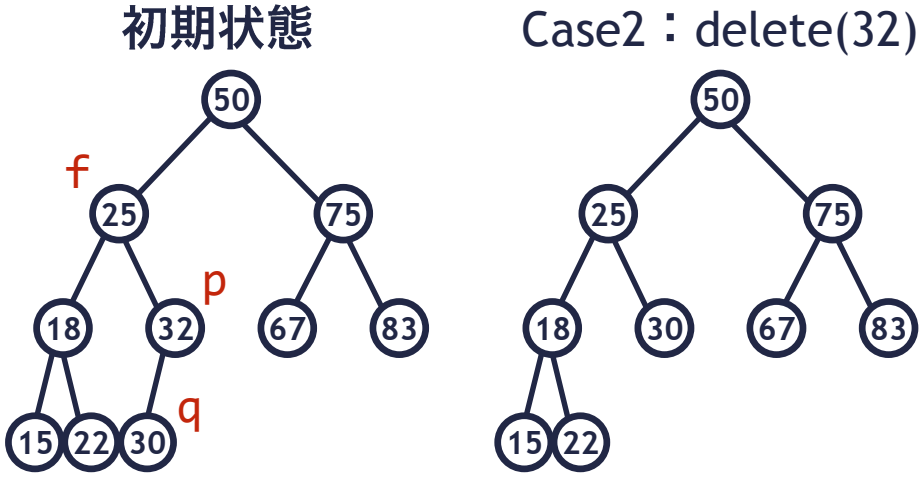
1.4.4 2分探索木

■ 削除プログラム

```
int remove( int x )
{
  struct node* f;
  struct node* p = &root;   // 削除ノード：根からスタート
  struct node* q;

  // 探索
  while( x != p->data && p != &nil ){

    f = p;

    if( x < p->data )
      p = p->lson;
    else
      p = p->rson;
  }

  if( p == &nil ){
    printf("その値は見つかりません¥n");
    return 0;
  }
```

1.4.4 2分探索木

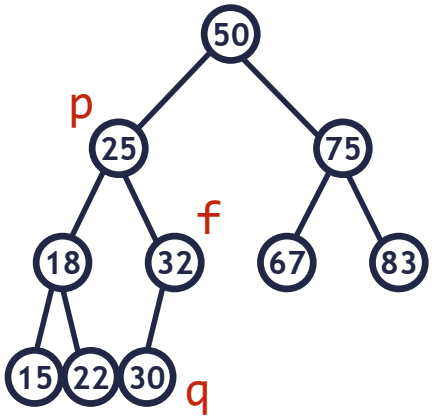
■ 削除プログラム



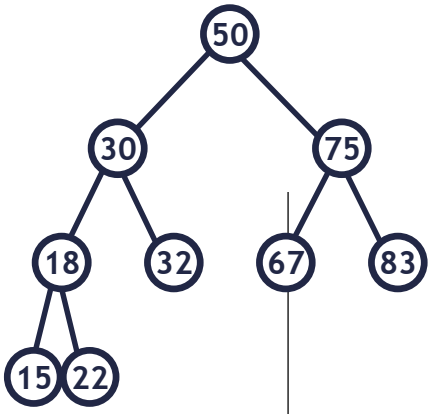

```
// 削除：case1, case2
if( p->lson == &nil || p->rson == &nil ){

  // case2では子ノードが，case1ではnilがqに格納される
  if( p->lson == &nil )
    q = p->rson;
  else
    q = p->lson;

  // 親子関係の更新
  if( f->lson == p )
    f->lson = q;
  else
    f->rson = q;
}
```

1.4.4 2分探索木

初期状態



Case3：delete(25)




```
  // 削除：case3
  else{

    // 削除ノードの右の子
    q = p->rson;
    f = q;
    // 左の子がいなくなるまで左の子を辿る
    while( q->lson != &nil ){

      f = q;              // 現在のノードを保存
      q = q->lson;        // 左の子へ移動
    }                     // fはqの親，qはfの左の子

    // 上書き
    p->data = q->data;

    if( q == f ) // 削除ノード(p)の右の子(f)に子がいない場合
      p->rson = q->rson;
    else
      f->lson = q->rson;
  }

  printf("%dを削除しました¥n", x);
  return 1;
}
```

1.4.4 2分探索木

■ main関数

```
int main( void )
{
  initialize(); //  初期化
  int mode, val;
  while(1){
    printf("モードを入力：(1)挿入, (2)削除, (3)探索, (4)終了 --> ");
    scanf_s("%d", &mode);
    if( mode == 1 ){
      printf("挿入する値を入力：");
      scanf_s("%d", &val);
      insert( val );
    }
    else if( mode == 2 ){
      printf("削除する値を入力：");
      scanf_s("%d", &val);
      remove( val );
    }
    else if( mode == 3 ){
      printf("探索する値を入力：");
      scanf_s("%d", &val);
      find( val );
    }
    else if( mode == 4 )
      break;
    else
      printf("入力された値が不正です¥n");
    return 0;
}
```

参考書

- 平田富夫著，アルゴリズムとデータ構造(第2章)，森北出版株式会社