算法数理工学
第4回

定兼　邦彦

集合を扱うデータ構造

- 集合：数学，計算機科学において基本的
- 動的集合：要素が追加/削除/変更される
- 集合に対して行う操作によってデータ構造を変える
- 行いたい操作によって最適なデータ構造が決まる

行う操作	データ構造
挿入，削除， 存在判定	辞書
挿入 最小要素の取出し	プライオリティーキ ュー

動的集合の基本

- 各要素はオブジェクトとして表現される
- オブジェクトはキーと付属データからなる
- 集合の操作で扱うフィールドがあってもよい
  - アルゴリズム内部でのみ用いられる
  - 他のオブジェクトへのポインタなど
- キーは全順序を持つとする場合もある

動的集合に関する操作

1. 集合に関する情報を返す質問(query)
   - SEARCH(S,k): $key[x] = k$ である S の要素 x へのポインタを返す．存在しなければ NIL.
   - MINIMUM(T): 全順序集合 T において，最小のキーを持つ要素を返す
   - MAXIMUM(T): 全順序集合 T において，最大のキーを持つ要素を返す
   - SUCCESSOR(T,x): キーが x のキーの次に大きな要素を返す．x が最大なら NIL.
   - PREDECESSOR(T,x): キーが x のキーの次に小さな要素を返す．x が最小なら NIL.

動的集合に関する操作

2. 集合を変える修正操作 (modifying operation)
   - INSERT(S,x): 集合 S に要素 x を加える.
   - DELETE(S,x): x へのポインタが与えられたとき，S から x を取り除く.

- SUCCESSOR，PREDECESSORは同じキーが複数ある集合にも拡張される
- 集合操作を実行するのにかかる時間は集合のサイズで測る

配列による動的集合の実現

- 同じキーを持つ要素は複数ないとする
- 集合のサイズが n のとき，要素を配列 S の $S[0],\ldots,S[n-1]$ に格納する
- SEARCH(S,k), INSERT(S,x), DELETE(S,x) を実現する

各操作の実現

- 挿入 INSERT
  - 配列の最後に追加．O(1) 時間
  - 予め確保した配列がいっぱいになったらそれ以上追加できない．もしくは，別の大きな配列を確保し，全要素を移動する必要がある．
- 削除 DELETE
  - (削除した要素の右の要素を全てずらす．O(n) 時間)
  - 削除する場所へ最後の要素を移動．O(1) 時間
- キーの検索 SEARCH
  - 配列の先頭から (任意の順で) 1つずつ比較していく
  - O(n) 時間

二分探索

- アルゴリズムとデータ構造で重要な概念
- 全順序集合の探索を高速化する
- 集合 S の要素を L, E, G に分ける
  - $L = \{x \mid x \in S, x < p\}$ (p より小さい要素)
  - $E = \{x \mid x \in S, x = p\}$ (p と等しい要素)
  - $G = \{x \mid x \in S, x > p\}$ (p より大きい要素)
- k を探索するとき
  - $p = k$ ならば探索終了 (見つかった)
  - $p < k$ ならば G を二分探索
  - $p > k$ ならば L を二分探索

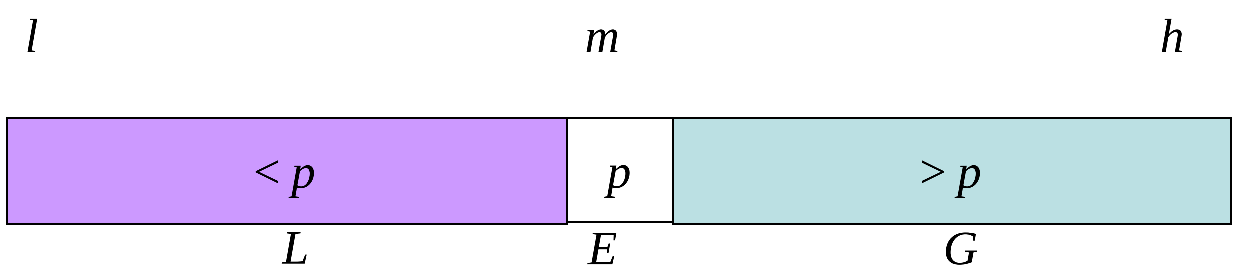
- サイズ n の集合の二分探索の時間を $T(n)$ とする
- L, G のサイズを n_1, n_2 とする $(n_1+n_2 \leq n)$
- $T(n) = \mathrm{O}(1) + \max\{T(n_1),T(n_2)\}$
- $n_1 \leq ½\, n$, $n_2 \leq ½\, n$ なら $T(n) = \mathrm{O}(1) + T(½\, n)$
- $T(n) = \mathrm{O}(\log n)$ となる

既ソート配列を用いた辞書

- 集合の要素を配列に格納するデータ構造
- 探索は二分探索を用いることができる
- 挿入はソート順を保つようにする必要がある
- 削除は (未ソートの場合と同じ)
  - 削除したところから右を1つずつ左にずらす
- 集合の要素は全て異なるとする

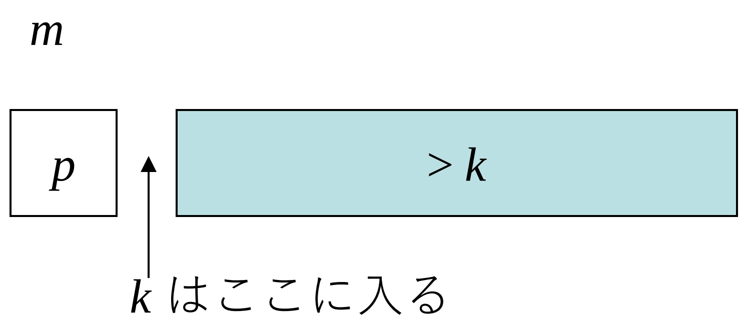
既ソート配列での二分探索

- E は配列の中央の要素 ($p = S[n/2]$)
- L は中央より左側の要素 ($S[0..n/2-1]$)
- G は中央より右側の要素 ($S[n/2+1..n-1]$)
- 集合が配列のl 番目からh 番目で表されているとき
- $m = (l+h) / 2$ とする
- $S[m]=k$ ならば探索終了 (k が存在した)
- $S[m]<k$ ならば k はL, E には存在しない⇒G を探索
- $S[m]>k$ ならば k はG, E には存在しない⇒L を探索

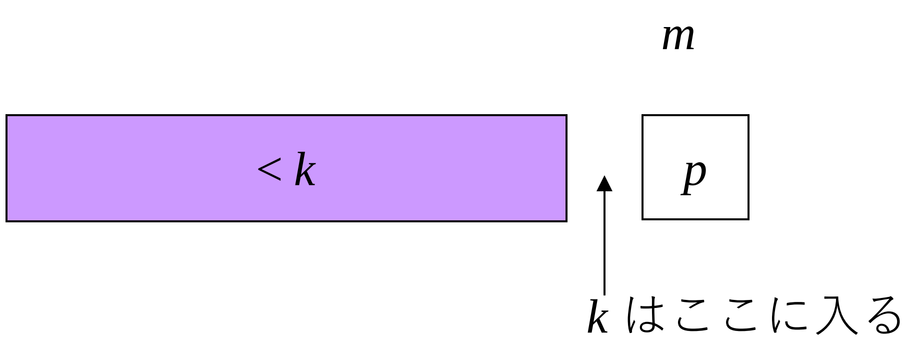


```
int search(dic_sortedarray *S, int k)
{
  int i;
  int high, low, mid;
  low = 0;                 // 探索の範囲は最初は配列全体 [0..n-1]
  high = S->n-1;
  i = 0;
  while (low <= high) {
    mid = (low + high) / 2;
    if (S->key[mid] == k) {
      return mid;
    } else if (S->key[mid] < k) {
      low = mid + 1;
      i = mid + 1;
    } else {
      high = mid - 1;
      i = mid;
    }
  }
  return -(i+1); // 見つからなかったときに挿入する場所を返す
}
```

- $l > h$ になったら探索終了 (見つからなかった)
- その直前では $l = h = m$
- $S[m] = p < k$ だったとき



- $S[m] = p > k$ だったとき



要素の挿入

- k が既に存在するなら挿入しない
- k を探索し，挿入場所 i を求める
- i から配列の最後までの要素を右に1つずらす
- 空いたところに k を入れる
- $O(n)$ 時間

```
void insert(dic_sortedarray *S,int k)
{int i,j;
  i = search(S, k);      if (i >= 0) return;
  if (S->n+1 > S->MAX) {
    printf("ERROR 配列のオーバーフロー\n");
    exit(1);}
  i = -i-1;
  for (j = S->n; j > i; j--) {S->key[j] = S->key[j-1];}
  S->key[i] = k;
  S->n = S->n + 1;
}
```

既ソート配列を用いた辞書のまとめ

- 探索 : O(log n) 時間
- 挿入: O(n) 時間
- 削除: O(n) 時間

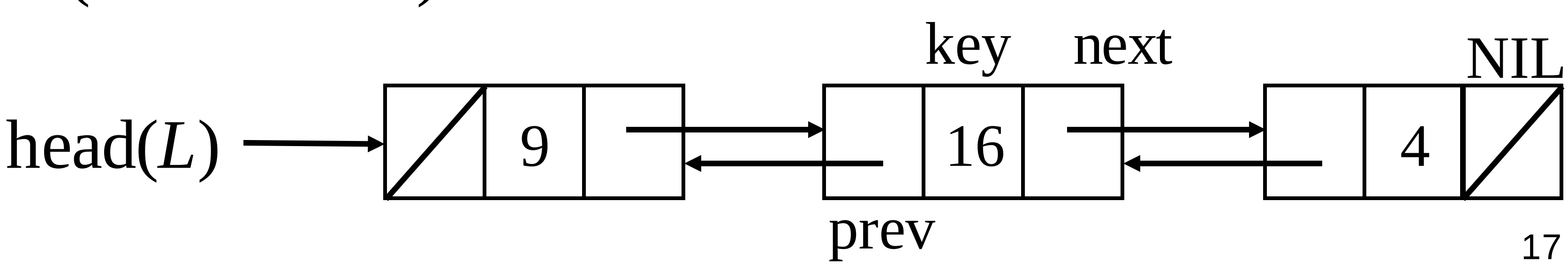
- 初めに指定したサイズのメモリを常に使用する
- それより多い数の要素は格納できない

配列による動的集合の問題点

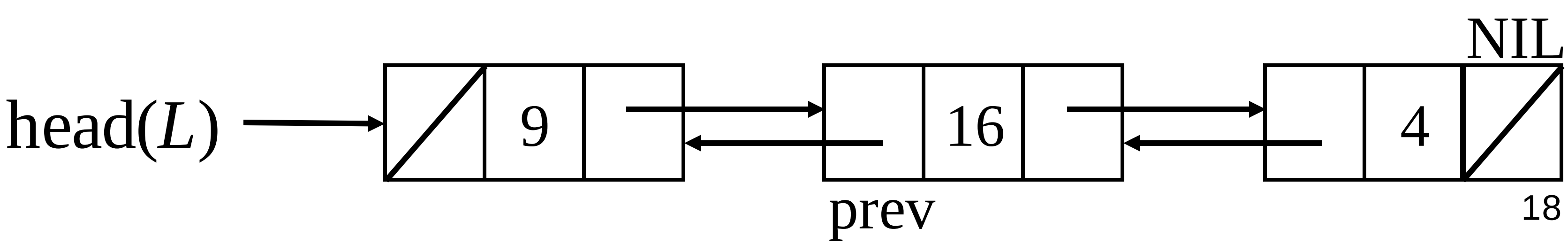
- 格納できる要素数に上限がある
- 常に最大要素数のメモリを使用する
- (削除が遅い)
- (検索が遅い)

連結リスト (Linked Lists)

- オブジェクトをある線形順序に並べて格納するデータ構造
- 連結リストでの線形順序は，オブジェクトが含むポインタで決定される
- 双方向連結リスト (doubly linked list) L の要素
  - キーフィールド key
  - ポインタフィールド prev, next
  - (付属データ)



- next(x): リスト中の x の直後の要素のポインタ
  - next(x) = NIL のとき, x は最後の要素
- prev(x): x の直前の要素のポインタ
  - prev(x) = NIL のとき, x はリストの最初の要素
- head(L): リストの先頭の要素のポインタ
  - head(L) = NIL のとき, リストは空



リストの種類

- 一方向 (singly linked) と双方向 (doubly linked)
  – 一方向のとき，各要素は prev を持たない
- 既ソート (sorted) と未ソート
  – 既ソート: リストの線形順序はキーの線形順序に対応
  – 未ソート: 任意の順序
- 循環 (circular list) と非循環
  – 循環: リストの先頭要素の prev はリストの末尾を指し，末尾の next はリストの先頭を指す
- 以下では未ソート双方向(連結)リストを扱う

双方向リストの構造体

- リストの要素

```
typedef struct dlobj {
  struct dlobj *prev;      // 前の要素へのポインタ
  struct dlobj *next;      // 後の要素へのポインタ
  data key;                // キー
} dlobj;
```

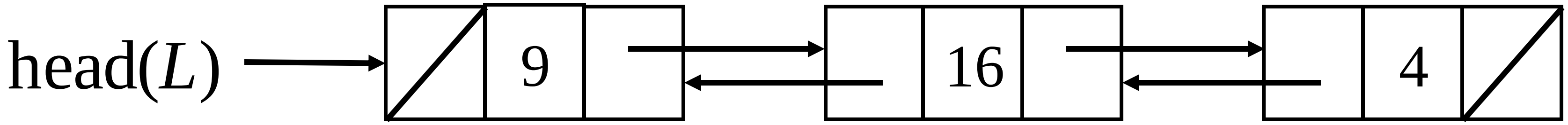
- 双方向リスト

```
typedef struct {
  dlobj *head;             // 先頭要素のポインタ
} dlist;
```

連結リストの探索

- list_search(L, k): リスト L に属する，キー k を持つ最初の要素のポインタを返す
- キー k を持つ要素が存在しなければ NIL を返す
- $\Theta(n)$ 時間

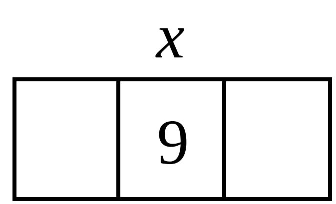
```
dlobj *list_search(dlist  *L, data k)
{
  dlobj *x;
  x = head(L);
  while (x != NIL && key(x)  != k) {
    x = next(x);
  }
  return x;
}
```

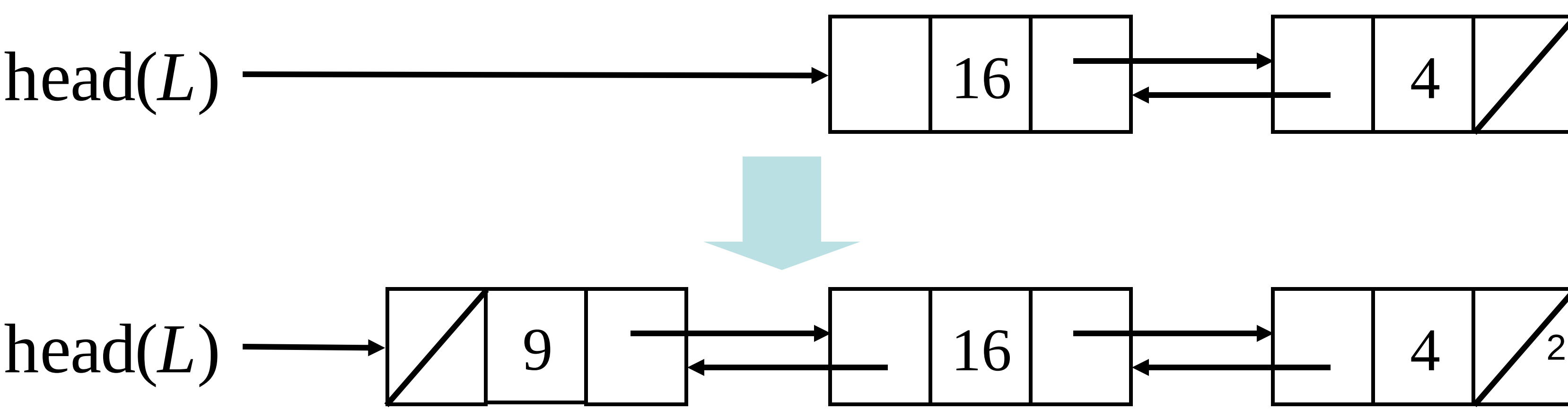


連結リストへの挿入

- list_insert(L, x): x を L の先頭に挿入
  - – x のキーは既にセットされているとする
- O(1) 時間

```
void list_insert(dlist *L, dlobj *x)
{
  next(x) = head(L);
  if (head(L) != NIL)
       prev(head(L)) = x;
  head(L) = x;
  prev(x) = NIL;
}
```

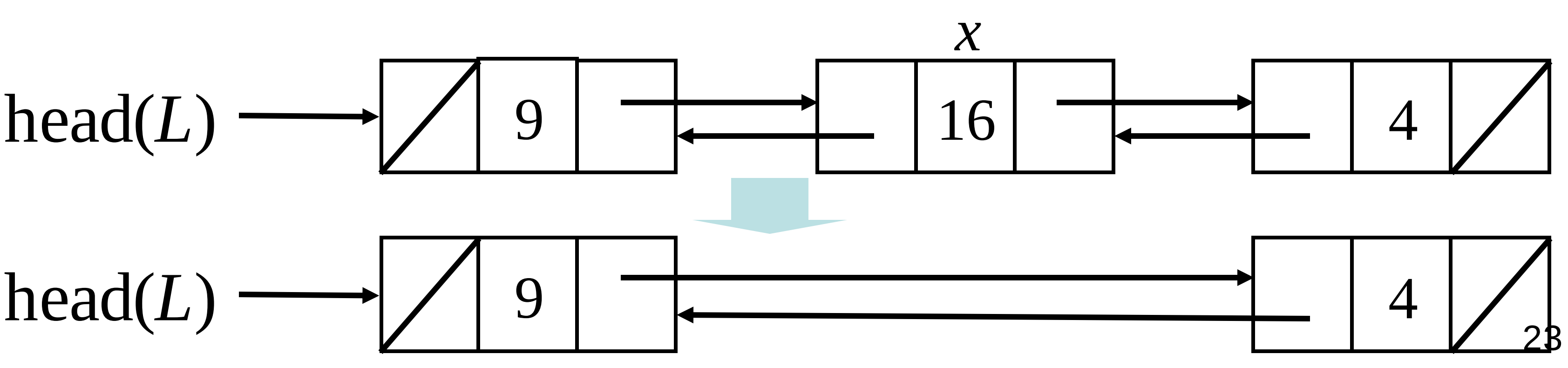




連結リストからの削除

- list_delete(L, x): L から x を削除
- O(1) 時間

```
void list_delete(dlist *L, dlobj *x)
{
  if (prev(x) != NIL)  next(prev(x)) = next(x);
  else                 head(L) = next(x);
  if (next(x) != NIL)  prev(next(x)) = prev(x);
}
```



双方向リストによる辞書の計算量

- 挿入
  - 常にリストの先頭に入れるので O(1) 時間
- 削除
  - 削除する要素のポインタが与えられれば O(1) 時間
- キーの検索
  - リストの要素を1つずつ見ていくので最悪 O(n) 時間
    n: リスト長 (要素数)
  - 既ソートリストでもリストの先頭から見ていくしかないので O(n) 時間

一方向リストによる辞書の計算量

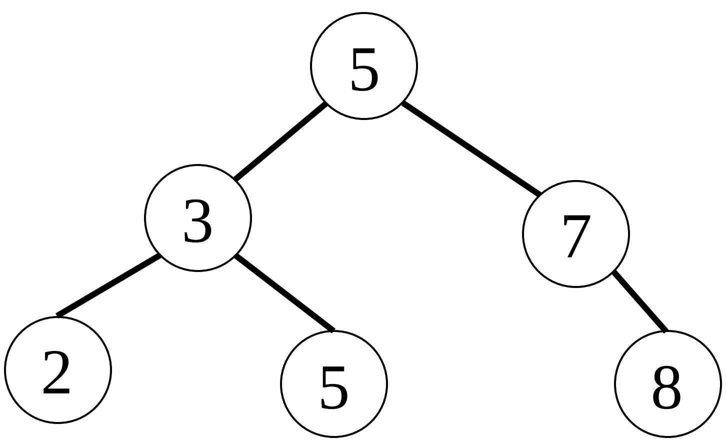
- 挿入
  - 常にリストの先頭に入れるので O(1) 時間
- 削除
  - 削除する要素のポインタが与えられても，リストの1つ前の要素が分からないので O(n) 時間
  - キーの検索の際に，目的の要素の1つ前の要素を求めるようにしておけば，削除は O(1) 時間
- キーの検索
  - O(n) 時間

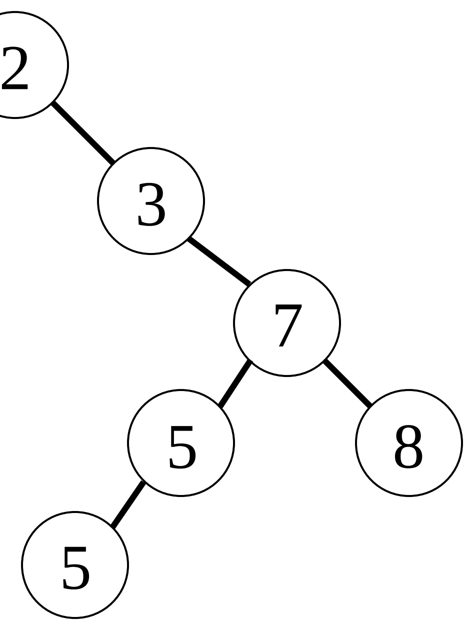
2分探索木

- 探索木: search, minimum, maximum, predecessor, successor, insert, delete等が利用できる動的集合用データ構造
- 辞書やプライオリティーキューとして利用できる
- 基本操作は木の高さに比例した時間がかかる
  - ランダムに構成された2分探索木の高さ: O(lg n)
  - 最悪時: O(n)
- 最悪時でもO(lg n) に改良できる (2色木)

2分探索木とは何か?

- 各節点は key, left, right, p フィールドを持つ
- 2分探索木条件 (binary-search-tree property)
  - 節点 y が x の左部分木に属する⇒ key(y)≤key(x)
  - 節点 y が x の右部分木に属する⇒ key(x)≤key(y)

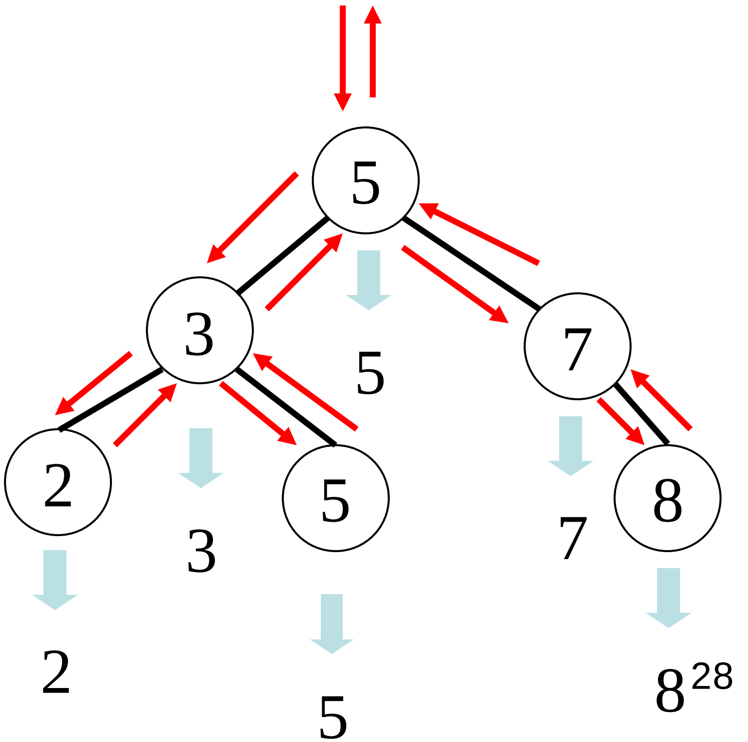




木の中間順巡回

- 木の中間順巡回 (通りがけ順, inorder tree walk)
  - 根の左部分木に出現するキー集合
  - 根のキー
  - 右部分木に出現するキー集合の順にキーを出力
- 木の根から辿ると，全てのキーをソートされた順序で出力できる
- $\Theta(n)$ 時間

```
inorder_tree_walk(node x)
{
  if (x != NIL) {
    inorder_tree_walk(left(x));
    print(key(x));
    inorder_tree_walk(right(x));
  }
}
```



その他の巡回法

- 先行順巡回 (行きがけ順, preorder tree walk):
  根節点を先に出力し，次に左右の部分木を出力
- 後行順巡回 (帰りがけ順, postorder tree walk):
  先に左右の部分木を出力し，最後に根節点を出力

```
preorder_tree_walk(node x)
{
    if (x != NIL) {
        print(key(x));
        preorder_tree_walk(left(x));
        preorder_tree_walk(right(x));
    }
}
```

```
postorder_tree_walk(node x)
{
    if (x != NIL) {
        postorder_tree_walk(left(x));
        postorder_tree_walk(right(x));
        print(key(x));
    }
}
```

2分探索木に対する質問操作

- 質問操作は高さに比例した時間で終了する
- 探索: 2分探索木の中から，ある与えられたキーを持つ節点のポインタを求める
  - 存在しなければNIL
  - 複数ある時はどれか一つ

```
tree_search(node x, data k)
{
    if (x == NIL || k == key(x)) return x;
    if (k < key(x)) return tree_search(left(x),k);
    else  return tree_search(right(x),k);
}
```

- 再帰はwhileループにすることができる

```
iterative_tree_search(node x, data k)
{
    while (x != NIL && k != key(x)) {
        if (k < key(x)) x = left(x);
        else  x = right(x);
    }
    return x;
}
```

探索の正当性

- キー k が見つかったら探索を終了する
- k が $\mathrm{key}(x)$ より小さい場合
  - 2分探索木条件より, k は x の右部分木にはない
  - 左部分木に対して探索を続行する
- k が $\mathrm{key}(x)$ より大きい場合
  - 右部分木に対して探索を続行する
- 探索する節点は根からのパスになる
  - 実行時間は $O(h)$ (h: 木の高さ)

最小値と最大値

- 最小/最大のキーを持つ要素のポインタを返す
- O(h) 時間

```
tree_minimum(node  x)
{
    while (left(x) != NIL) {
        x = left(x);
    }
    return x;
}
```

```
tree_maximum(node  x)
{
    while (right(x) != NIL) {
        x = right(x);
    }
    return x;
}
```

次節点と前節点

- 2分探索木のある節点が与えられたとき，木の中間順 (inorder) で次/前の節点を求める
- O(h) 時間

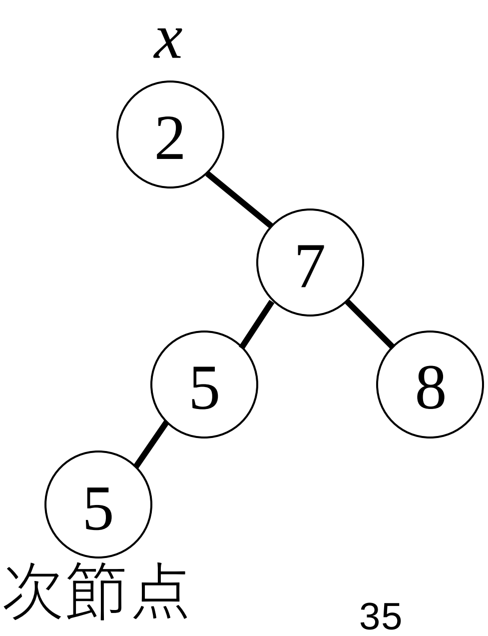
```
tree_successor(node x)
{
   node y;
   if (right(x) != NIL) return tree_minimum(right(x));
   y = p(x);
   while (y != NIL && x == right(y)) {
      x = y;
      y = p(y);
   }
   return y;
}
```

x が右部分木を持つ場合

- x の次節点は，x 以上の要素で最小

⇒ x の次節点は，x の右部分木の最小要素

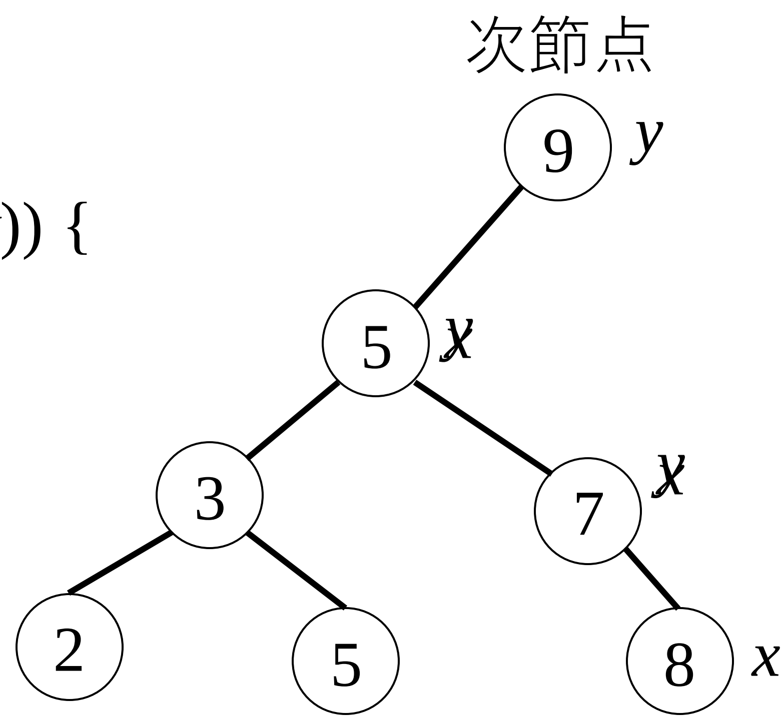
　= tree_minimum(right(x))



x が右部分木を持たない場合

- x の親を y とする
- x が，x の親の右の子ならば，親は x 以下
- y は，x を左部分木に持つ x の祖先で最も x に近いもの

```
y = p(x);
while (y != NIL && x == right(y)) {
    x = y;
    y = p(y);
}
return y;
```

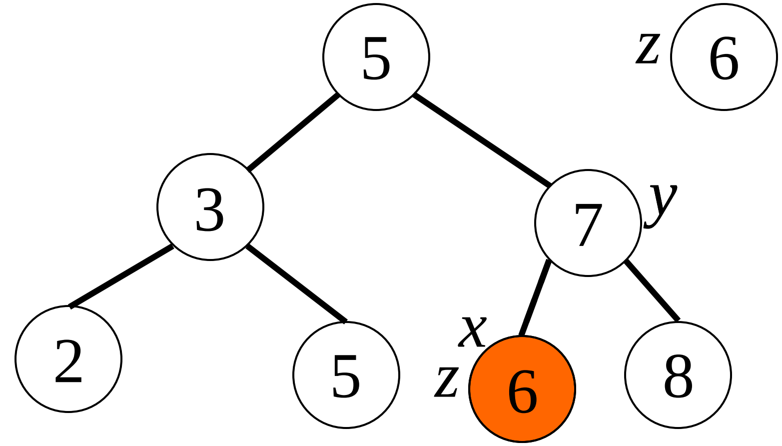


定理 高さ h の2分探索木上の動的集合演算
search, maximum, minimum, successor, predecessor は $O(h)$ 時間で実行できる.

挿入と削除

- 要素を挿入/削除したあとも2分探索木条件が満たされる必要がある
- 挿入は比較的簡単
- 削除は複雑
- どちらも O(h) 時間

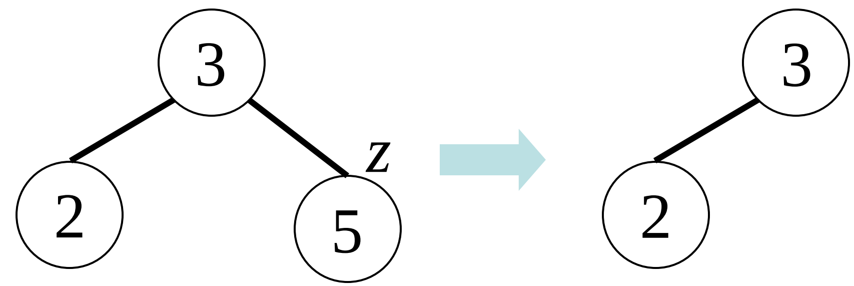
挿入



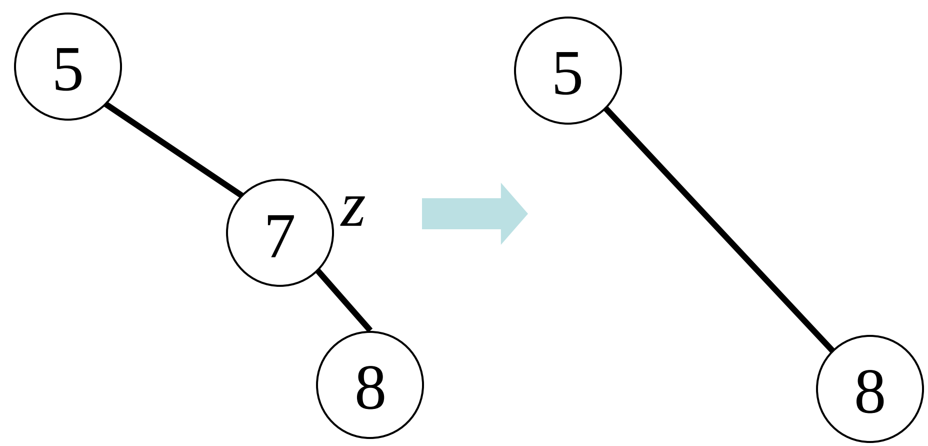

```
tree_insert(tree T, node z)
{
    node x,y;
    y = NIL;
    x = root(T);
    while (x != NIL) {                    // z を挿入する場所 x を決める
        y = x;
        if (key(z) < key(x)) x = left(x);        挿入場所は必ず葉
        else  x = right(x);
    }                                     // y は x の親
    p(z) = y;                             // z の親を y にする
    if (y == NIL) root(T) = z;            // T が空なら z が根節点
    else if (key(z) < key(y)) left(y) = z; // y の子を z にする
        else right(y) = z;
}
```

削除

- 探索木から節点 z を削除する
- z が子を持たない場合
  - z の親 $p(z)$ を変更する

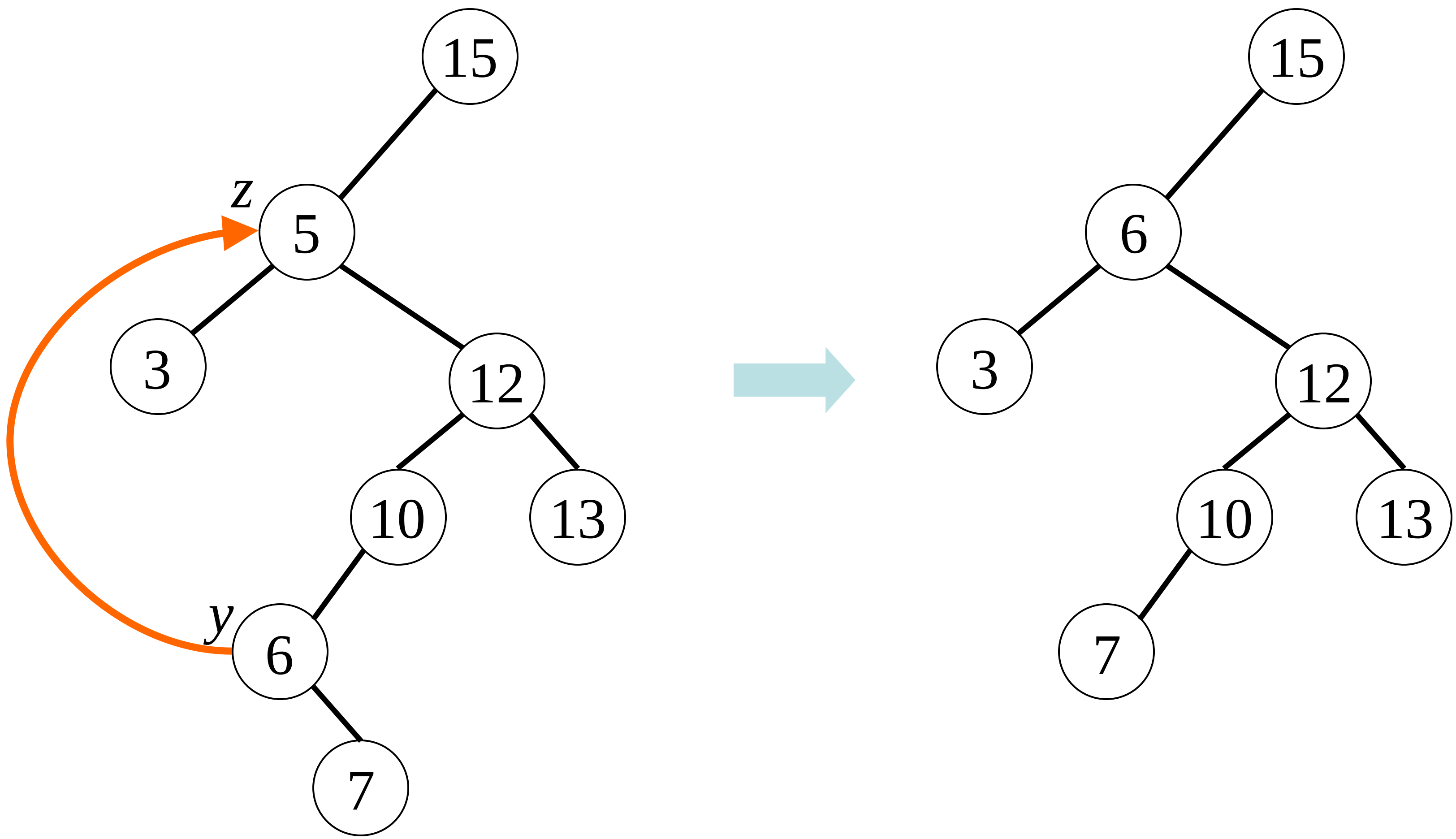


- z が子を1つ持つ場合
  - z の親と z の子を結ぶ



削除: z が子を2つ持つ場合

- z の次節点は左の子を持たない
- z の場所に y を入れ，元の y を削除する



```
tree_delete(tree T, node z)
{
  node x, y;
  if (left(z) == NIL || right(z) == NIL) y = z;  // z の子の数が1以下
  else y = tree_successor(z);                    // z は2つの子を持つ
  if (left(y) != NIL) x = left(y);  else x = right(y); //  x は y の子
  if (x != NIL) p(x) = p(y);                     // y を削除する
  if (p(y) == NIL) root(T) = x;                  // y が根なら x を根に
  else if (y == left(p(y))) left(p(y)) = x;      // y の親と子をつなぐ
       else right(p(y)) = x;
  if (y != z) {
    key(z) = key(y);                             // y の内容を z に移動
    // y の付属データを z にコピー
  }
  return y;                                      // 不必要な y を回収
}
```