アルゴリズムとデータ構造

記号表と2分探索木（その１）

目次

- 記号表
- 記号表抽象データ型
- 逐次探索
- 2分探索法
- 2分探索木
- 性能

探索

- 探索:
  - 前もって格納されている多くの情報の中から望みの情報を引き出す操作
  - データは，いくつかのレコード(=項目)から構成される
  - 各項目は，キーを持っている
  - 探索の目的は，与えられた探索キーと一致するキーを持つすべての項目を見つけだし，処理に必要な項目内の情報を得ること

記号表

- 記号表(symbol table), 辞書(dictionary):
  - キーを持つ項目からなるデータ構造であり, 2つの基本操作を提供する
    - 新しい項目を挿入する
    - 与えられたキーを持つ項目を返す

記号表抽象データ型

- 記号表抽象データ型に対する操作
  - insert: 新しい項目を挿入する
  - search: 与えられたキーを持つ項目を探索する
  - delete: 指定された項目を削除する
  - select: k番目の項目を選択する
  - sort: 記号表を整列する
  - join: 2つの記号表を合併して，1つの大きな記号表を作る
  - その他，初期設定(initialize)，空であるかの検査(test_if_empty)など

記号表抽象データ型

- 記号表抽象データ型に対する操作
  - insert: 新しい項目を挿入する
  - search: 与えられたキーを持つ項目を探索する
  - delete: 指定された項目を削除する
  - select: k番目の項目を選択する
  - sort: 記号表を整列する
  - join: 2つの記号表を合併して，1つの大きな記号表を作る
  - その他，初期設定(initialize)，空であるかの検査(test_if_empty)など

記号表抽象データ型

- 項目とキーの実装
- 操作のインタフェース等を実装
- 操作の実体を実装
  - 配列による実装
  - 二分探索木による実装

記号表抽象データ型

- 項目とキーの実装
- 操作のインタフェース等を実装
- 操作の実体を実装
  - 配列による実装
  - 二分探索木による実装

記号表抽象データ型

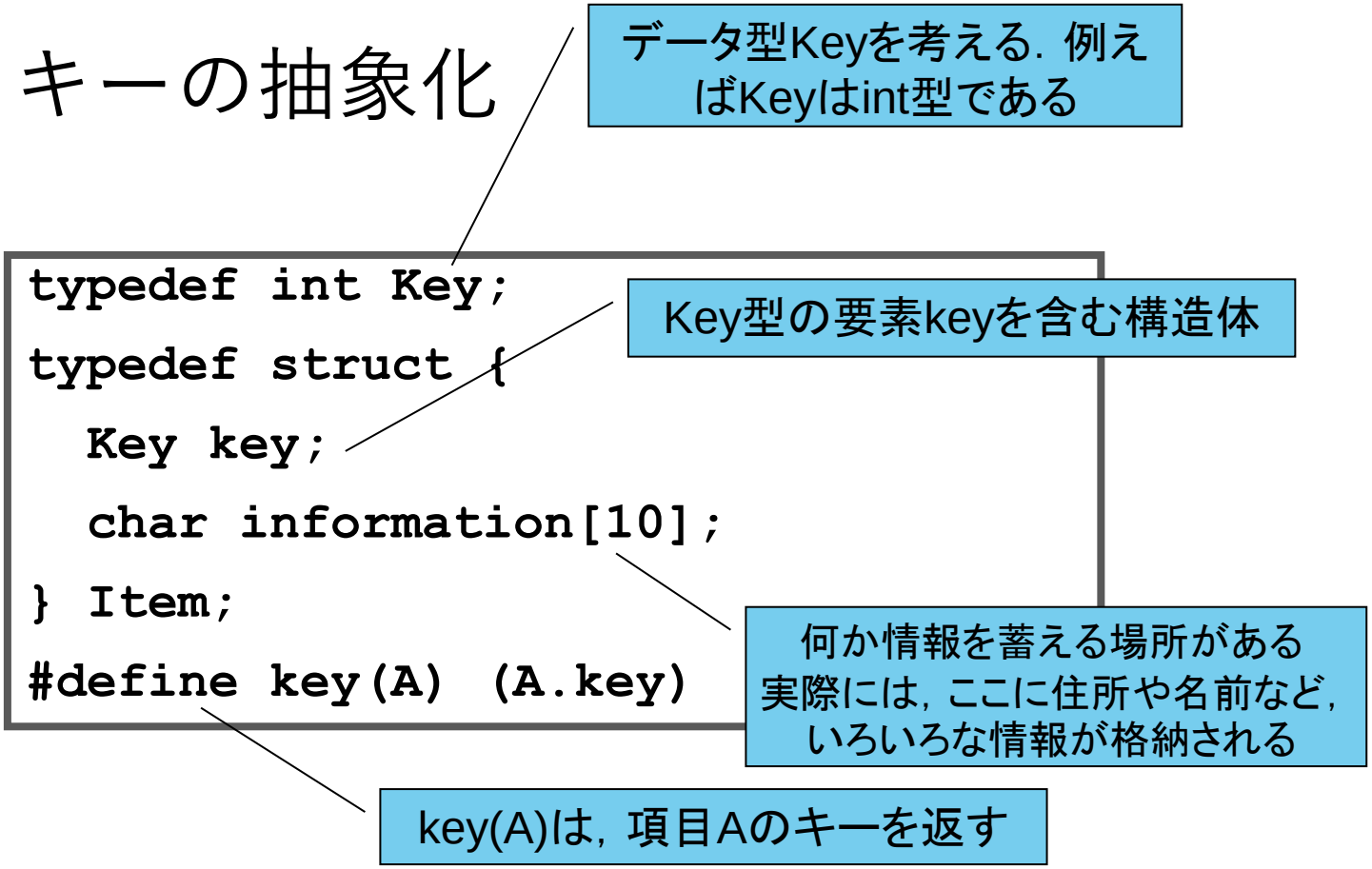
STsample.c

```
#include <stdio.h>
typedef int Key;
typedef struct { Key key; char information[10]; } Item;
#define key(A) (A.key)
int main(void)
{
    Item item = {5, "ABCDEFG"};
    printf("Key of the item is %d¥n", key(item));
    return 0;
}
```

```
$ ./STsample
Key of the item is 5
$
```

記号表抽象データ型

- 項目とキーの抽象化



記号表抽象データ型

STsample.c

```
#include <stdio.h>
typedef int Key;
typedef struct { Key key; char information[10]; } Item;
#define key(A) (A.key)
int main(void)
{
    Item item = {5, "ABCDEFG"};
    printf("Key of the item is %d¥n", key(item));
    return 0;
}
```

```
$ ./STsample
Key of the item is 5
$
```

記号表抽象データ型

- 項目とキーの実装
- 操作のインタフェース等を実装
- 操作の実体を実装
  - 配列による実装
  - 二分探索木による実装

記号表抽象データ型

- 次のように，NULLitemを定義し，searchが失敗したときには，NULLitemを返すようにする

```
Item NULLitem = {-1, "NULL"};
```

NULLitemのキーの値は，-1と定義する

(-1という数字に意味はなく，他の数字でもいい)

記号表抽象データ型

Item.h

```
typedef int Key;
typedef struct { Key key; char information[10]; } Item;
#define key(A) (A.key)

#define less(A, B) (A < B)
#define exch(A, B) { Item t = A; A = B; B = t; }
#define compexch(A, B) if (less(B, A)) exch(A, B)
#define eq(A, B) (A == B)

Item NULLitem;

Key ITEMrand(void);
int ITEMscan(Key *);
void ITEMshow(Item);
```

Item.hではNULLitemをグローバル変数として宣言だけしておく

記号表抽象データ型

Item.c

Item.cでNULLitemをグローバル変数として定義し，適当な値に初期化する

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"

Item NULLitem = {-1, "NULL"};

Key ITEMrand(void)
          { return rand()%100000; }
int ITEMscan(Key *x)
          { return scanf("%d", x); }
void ITEMshow(Item x)
          { printf("%3d ", key(x)); }
```

記号表抽象データ型

記号表抽象データ型のインタフェースを与える

ST.h

```
void STinit(int);
 int STcount(void);
void STinsert(Item);
Item STsearch(Key);
void STdelete(Item);
Item STselect(int);
void STsort(void (*visit)(Item));
```

記号表の中の
レコード数を数える

関数STsortは，引数として「**関数へのポインタ**」をとる

記号表抽象データ型

ST.c

ST.cの中身は，これから設計していく

設計1: 配列に基づく記号表(ST_ARRAY.c)

設計2: 2分探索木に基づく記号表(ST_BST1.cとST_BST2.c)

設計3: 赤黒木に基づく記号表(ST_RB.c)

実際に使用するときは，ST_ARRAY.c，ST_BST1.c，ST_BST2.cあるいは
ST_RB.cは，ST.cとファイル名を変更して使用することにする

クライアントプログラムの例

STtest.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "ST.h"

int main(int argc, char *argv[])
 {  int N, M, maxN = atoi(argv[1]), sw = atoi(argv[2]);
    Key v; Item item;
    STinit(maxN); srand(1);
    for (M = 0, N = 0; N < maxN; N++)
      {
        if (sw == 1) v = ITEMrand();
        else if (sw == 2) v = N+1;
        else if (ITEMscan(&v) == EOF) break;
        item = STsearch(v); if (item.key != NULLitem.key) continue;
        key(item) = v;
        STinsert(item); M++;
      }
    STsort(ITEMshow); printf("¥n");
    printf("%d keys ", N);
    printf("%d distinct keys¥n", STcount());
    return 0;
 }
```

乱数，小さい順，あるいはキーボードからキーを入力する

ダウンロードしたSTtest.cには，さらに多くの情報を表示する機能が付加されている

コンパイル/実行

STtest.cを，Item.cとST.cと一緒にコンパイルする

```
$ gcc -o STtest STtest.c Item.c ST.c
$ ./STtest 5 1
378 505 554 591 690
...
$
```

ランダムな5個の要素を「記号表」に挿入し，
整列して，表示せよ

コンパイル/実行

小さい順に5個の要素を挿入し，整列して表示せよ

```
$ ./STtest 5 2
 1 2 3 4 5
...
$ ./STtest 5 0
15 235 35 44 16
 15  16  35  44 235
...
$
```

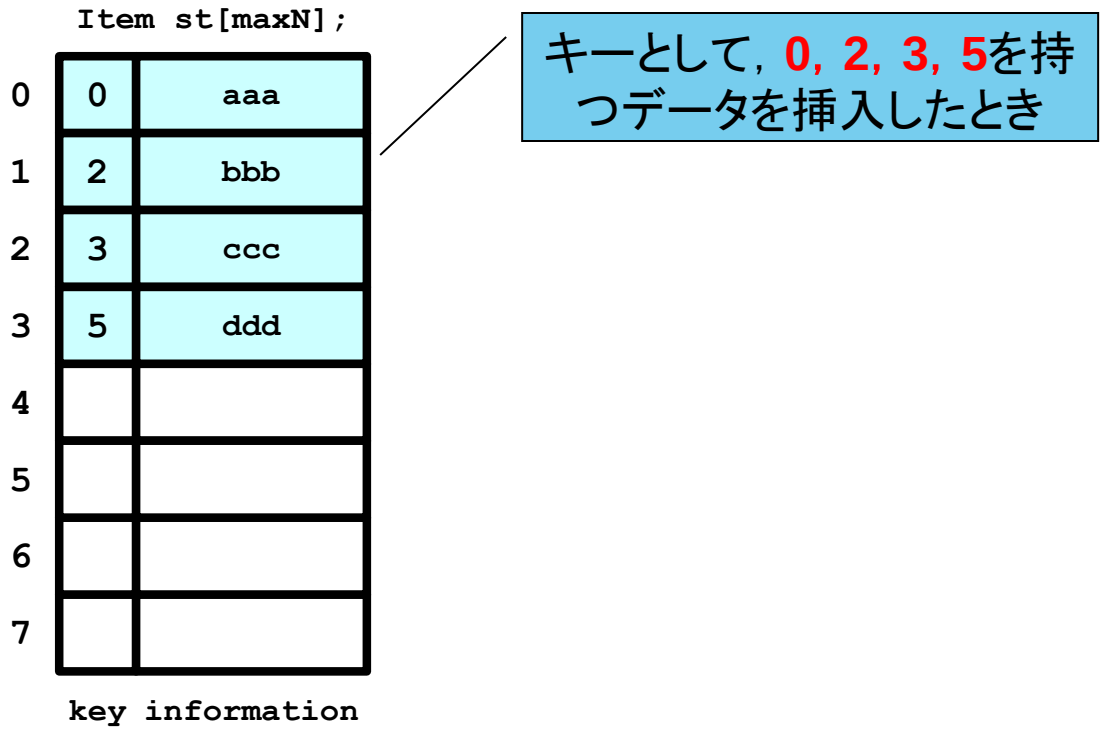
キーボードから5個の要素を入力した
全ての数字を入力した後，「Enter」，次に「Ctrl-d」

記号表抽象データ型

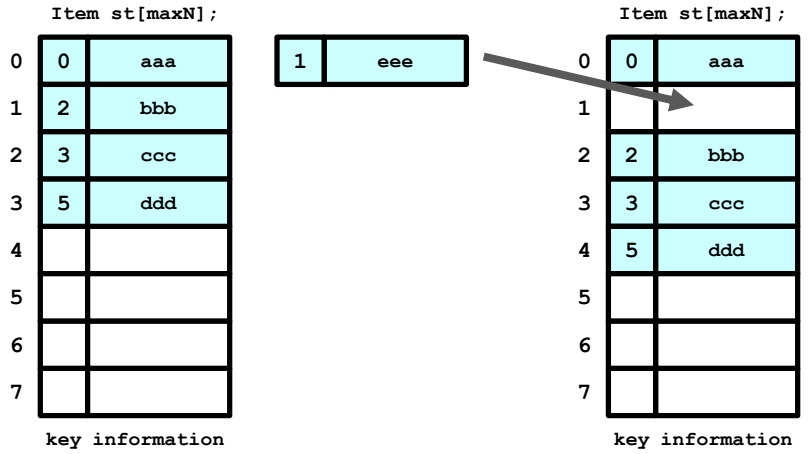
- 項目とキーの実装
- 操作のインタフェース等を実装
- 操作の実体を実装
  - 配列による実装
  - 二分探索木による実装

逐次探索

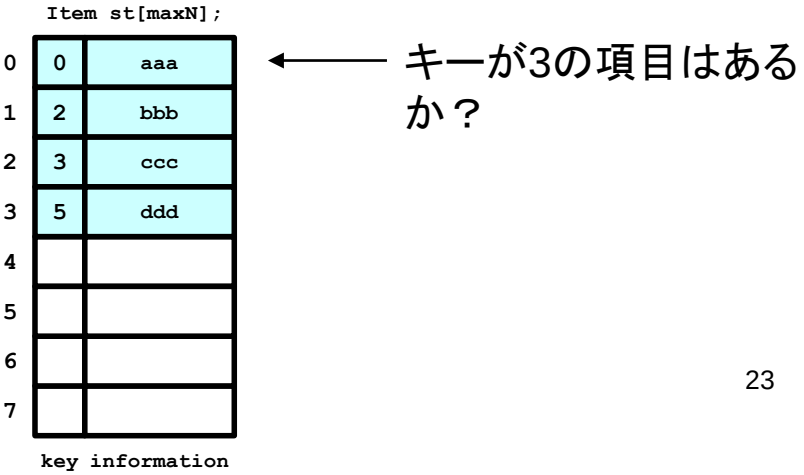
- 項目を配列の中に順番に連続して配置する
  - 配列に基づく記号表を作成することができる
  - 配列はキーによって整列しておく



逐次探索



- 新たな項目の挿入
  - 挿入整列と同様に，挿入する項目のキーより大きいキーを持つ項目を1つずつ移動させ，空いたところにその項目を挿入
- 探索
  - 配列を先頭から順に調べ，キーが一致/不一致をチェックする



配列に基づく記号表(整列)

ST_ARRAY.c

```
#include <stdlib.h>
#include "Item.h"
static Item *st;
static int N;
void STinit(int maxN)
  { st = malloc((maxN)*sizeof(Item)); N = 0; }
int STcount(void)
  { return N; }
void STinsert(Item item)
  { int j = N++; Key v = key(item);
    while (j>0 && less(v, key(st[j-1])))
      {  st[j] = st[j-1]; j--; }
    st[j] = item;
  }
```

挿入整列と同様にして，項目の挿入を行っている

配列に基づく記号表(整列)

ST_ARRAY.c (つづき)

```
Item STsearch(Key v)
  { int j;
    for (j = 0; j < N; j++)
      {
        if (eq(v, key(st[j]))) return st[j];
        if (less(v, key(st[j]))) break;
      }
    return NULLitem;
  }
```

一致したらその項目を返す

vより大きいキーが現れたら，ループから出て，
NULLitemを返す
(キーがvのものは見つからなかった)

配列に基づく記号表(整列)

ST_ARRAY.c (つづき)

```
Item STselect(int k)
  { return st[k]; }
void STsort(void (*visit)(Item))
  { int i;
    for (i = 0; i < N; i++) visit(st[i]);
  }
```

整列しているため，k番目の項目は，単にst[k]でよい

整列しているため，単に，先頭から訪問すればよい

逐次探索

- 整列した配列による実現
- 整列していない配列による実現
- 整列したリンクリストによる実現
- 整列しないリンクリストによる実現

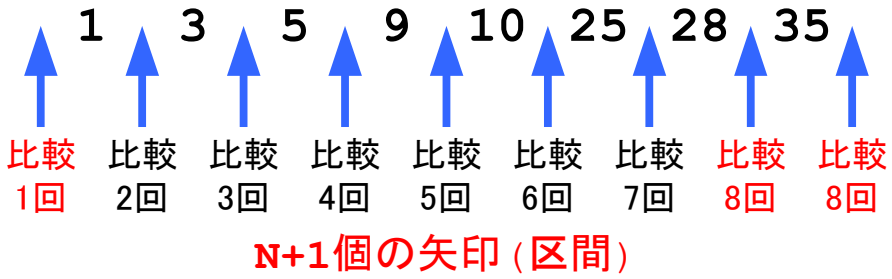
逐次探索

- 性質12.2
  - 項目がN個ある記号表上(整列されていても，整列されていなくても)における逐次探索は，成功探索では平均約N/2回の比較を行う
    - (1+2+・・・+N)/N = (N+1)/2
    - 配列でもリンクリストでも成立する
- 性質12.3
  - N個の項目が整列されていない記号表上では，逐次探索は不成功探索では(常に)N回の比較を行い，挿入は定数回の操作を行う
    - 配列でもリンクリストでも成立する

逐次探索

- 性質12.4
  - N個の項目が整列されている記号表上では，挿入，（成功および不成功）逐次探索は，どれも平均約N/2回の比較を行う
    - 挿入，成功探索では，性質12.2と同様にして明らか
    - 不成功探索では，探索が不成功になる箇所がN+1箇所(N個の項目の間と，先頭のさらに前，最後尾のさらに後ろ)ある．これらが等しい確率であり，その場合の比較回数は，1，2，・・・，N, Nだから
      - (1+2+・・・+N+N)/(N+1) = N/2+1-1/(N+1)
    - 配列でもリンクリストでも成立する

例えば，N=8のとき，



不成功探索は，左の矢印のどこかで終了する
最も左の矢印では，1回の比較が必要
右から2番目の矢印では，N回の比較が必要
最も右の矢印でも，N回の比較が必要

これらが等確率であり得る

記号表での挿入と探索のコスト

	最悪の場合			平均の場合		
	挿入	探索	選択	挿入	成功探索	不成功探索
キー添字配列	1	1	M	1	1	1
整列した配列	N	N	1	N/2	N/2	N/2
整列したリンクリスト	N	N	N	N/2	N/2	N/2
整列していない配列	1	N	NlgN	1	N/2	N
整列していないリスト	1	N	NlgN	1	N/2	N
2分探索	N	lgN	1	N/2	lgN	lgN
2分探索木	N	N	N	lgN	lgN	lgN
赤黒木	lgN	lgN	lgN	lgN	lgN	lgN
ランダム化木	N	N	N	lgN	lgN	lgN
ハッシュ法	1	N	NlgN	1	1	1

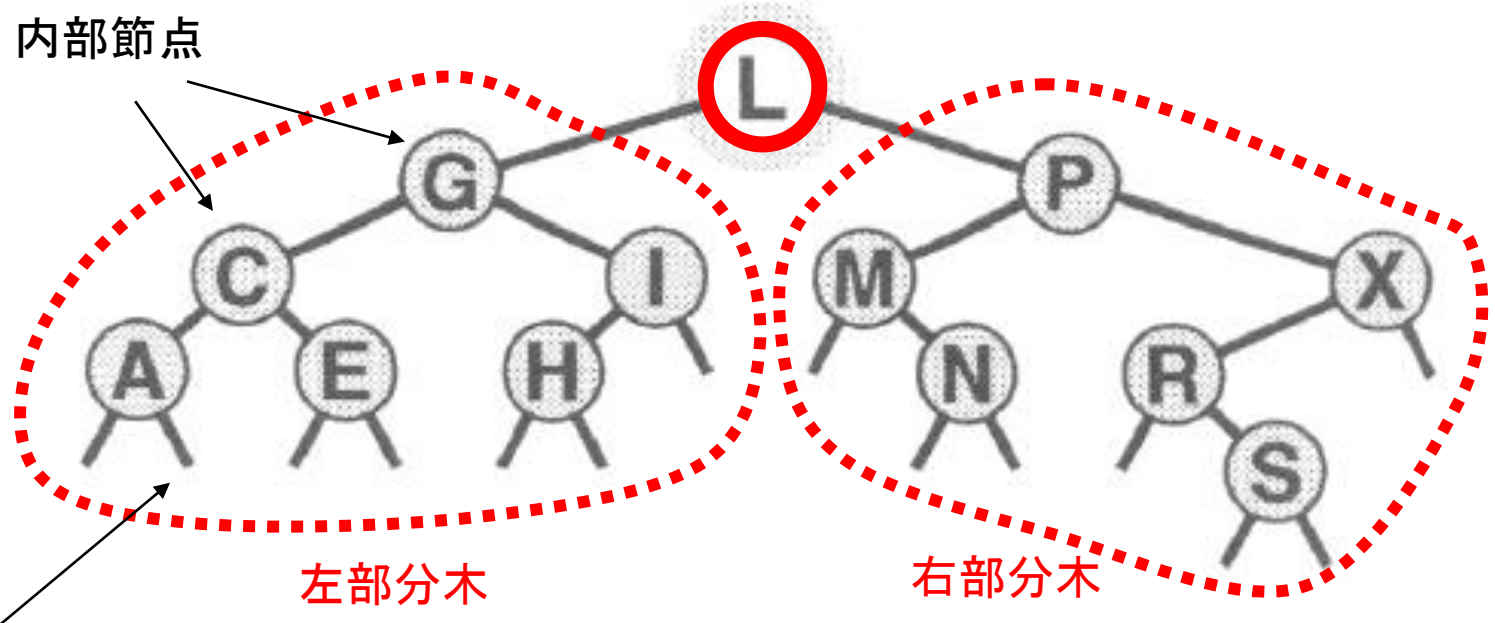
N:項目の数　　M:表のサイズ

整列した配列に項目を蓄える必要があるため，挿入は整列した配列と同じになる

2分探索木

- 挿入操作が高くつくという問題を解決するために，記号表実現の基礎として木構造を用いる
- 2分探索木(binary search tree, BST):
  - 内部節点にキーが置かれた2分木で，次の性質を満たすものである
    - 節点が置かれたキーより小さい(あるいは等しい)キーを持つ項目はすべてその節点の左部分木の中にある
    - 節点が置かれたキーより大きい(あるいは等しい)キーを持つ項目はすべてその節点の右部分木の中にある

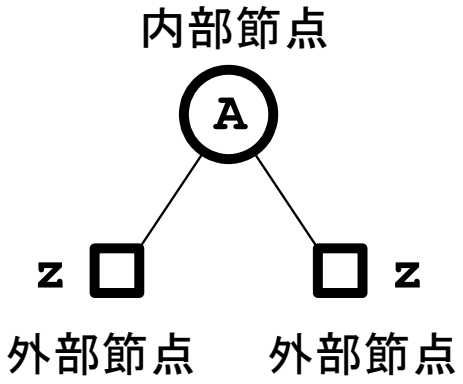
2分探索木

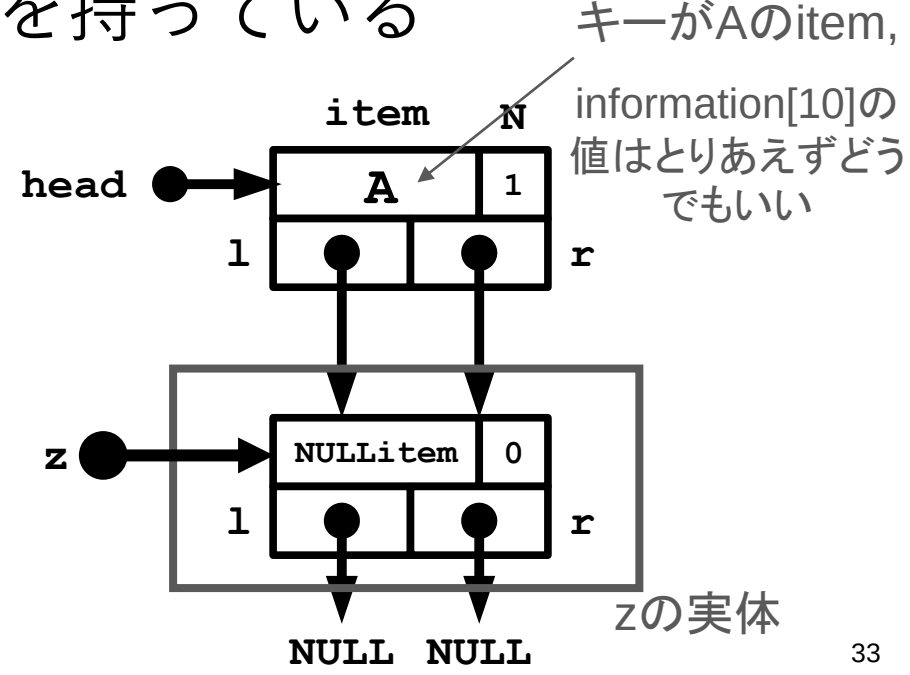


内部節点のみ表示している．外部節点は省略してある

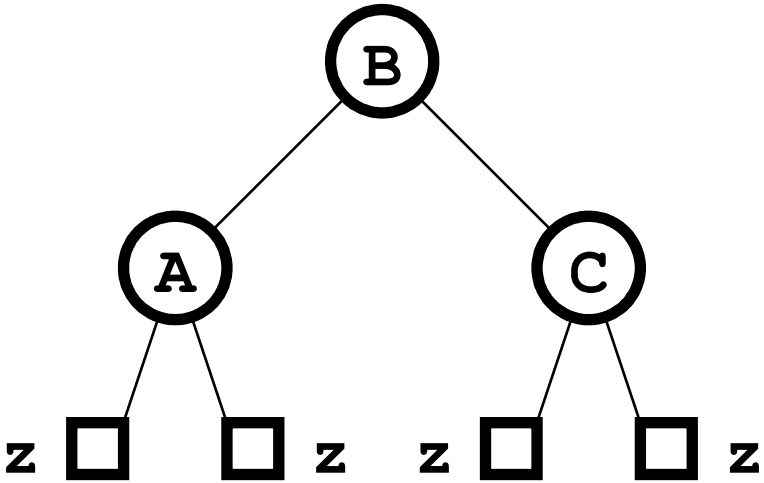
BSTによる記号表

- ダミー節点z
  - 外部節点は，ダミー節点zとする
  - 左あるいは右に子を持たない内部節点は，ダミー節点へのリンクを持っている

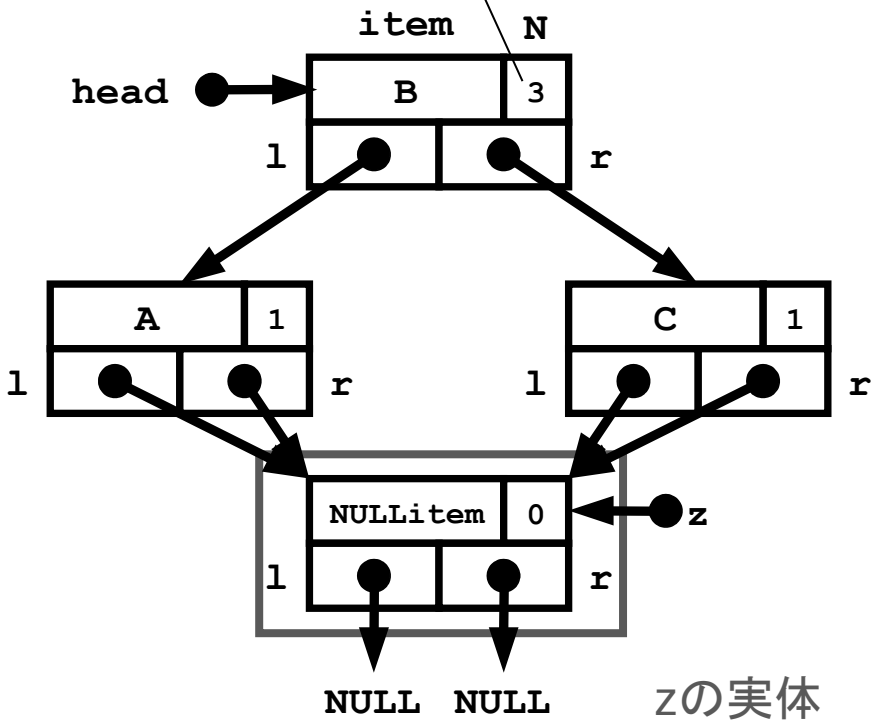




BSTによる記号表



その節点を根とする部分木の
内部節点数の総数

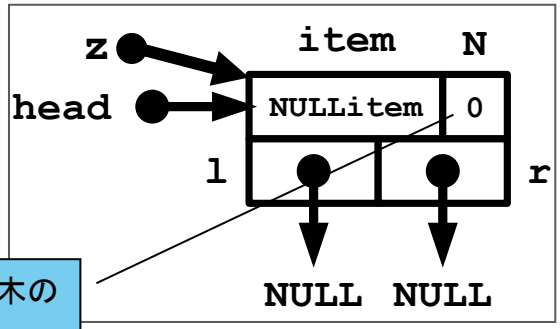


zの実体

BSTによる記号表

外部節点

z



その節点を根とする部分木の内部節点数の総数

ST_BST1.c

ST.cに名前を変更して使用する

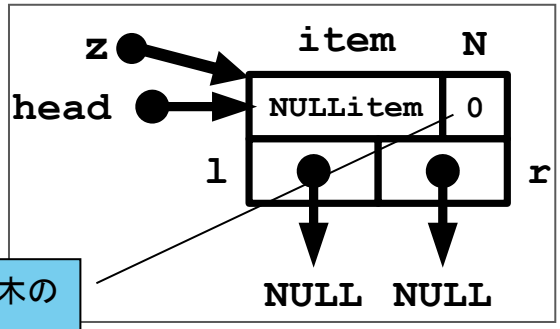
まず，STinitによって「外部節点1つの空な木」に初期化される

```
#include <stdlib.h>
#include "Item.h"
typedef struct STnode* link;
struct STnode { Item item; link l, r; int N; };
static link head, z;
link NEW(Item item, link l, link r, int N)
  { link x = malloc(sizeof *x);
    x->item = item; x->l = l; x->r = r; x->N = N;
    return x;
  }
void STinit()
  { head = (z = NEW(NULLitem, 0, 0, 0)); }
int STcount(void) { return head->N; }
```

BSTによる記号表

外部節点

z



その節点を根とする部分木の内部節点数の総数

ST_BST1.c

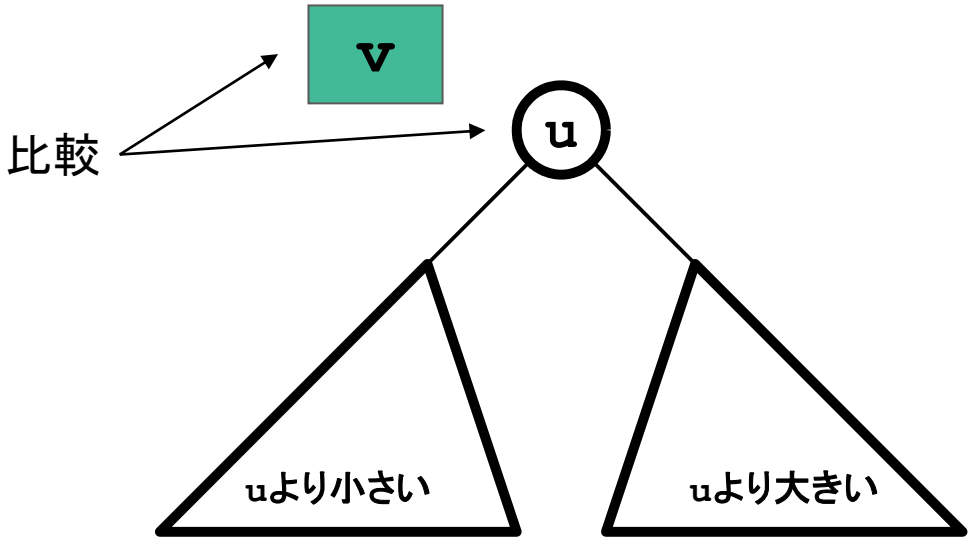
ST.cに名前を変更して使用する

まず，STinitによって「外部節点1つの空な木」に初期化される

```
#include <stdlib.h>
#include "Item.h"
typedef struct STnode* link;
struct STnode { Item item; link l, r; int N; };
static link head, z;
link NEW(Item item, link l, link r, int N)
  { link x = malloc(sizeof *x);
    x->item = item; x->l = l; x->r = r; x->N = N;
    return x;
  }
void STinit()
  { head = (z = NEW(NULLitem, 0, 0, 0)); }
int STcount(void) { return head->N; }
```

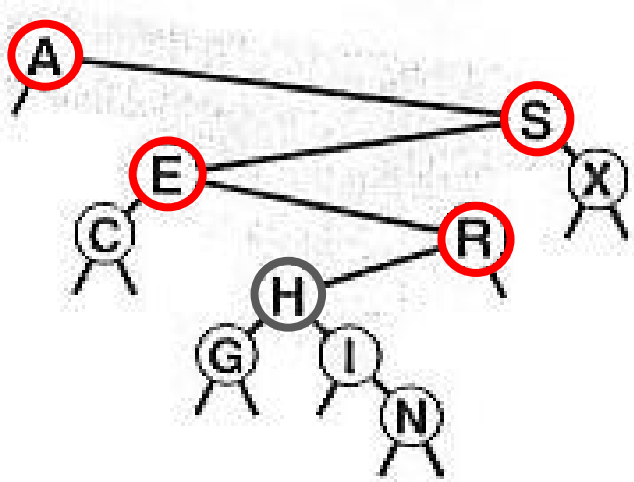
探索

- BST中のキーを探す手続き
  - 木が空ならば探索は不成功
  - 根にあるキーuが探索キーvならば，探索は成功
  - そうでなければ，キーの大小によって，左あるいは右の部分木を探索する

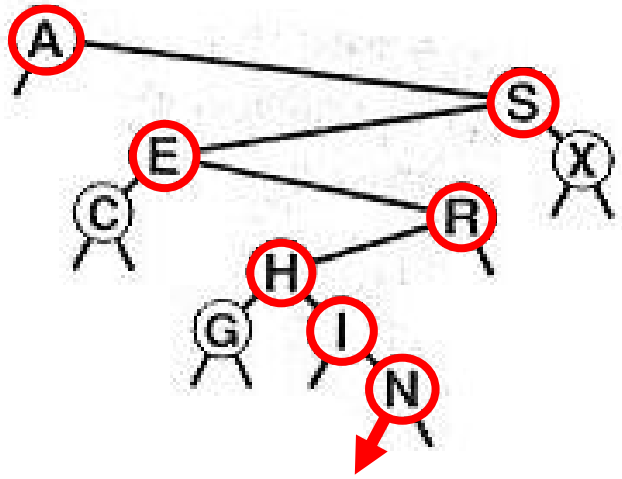


探索

Hに対する成功探索



Mに対する不成功探索



探索

ST_BST1.c (つづき)

```
Item searchR(link h, Key v)
  { Key t = key(h->item);
    if (h == z) return NULLitem;
    if eq(v, t) return h->item;
    if less(v, t) return searchR(h->l, v);
                  else return searchR(h->r, v);
  }
Item STsearch(Key v)
  { return searchR(head, v); }
```

外部節点まで来たら不成功

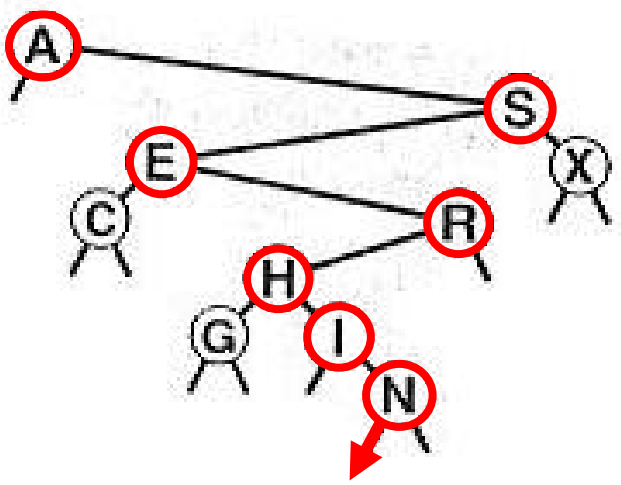
一致すれば成功

右の部分木あるいは
左の部分木を探索する

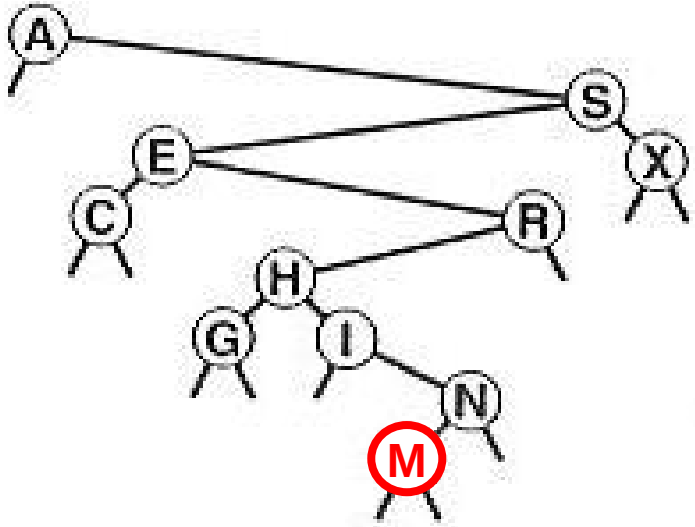
挿入

- BSTに新しい項目を挿入する手続き
  - 探索し，不成功した場所を見つける
  - その場所に節点を挿入する

Mに対する不成功探索



Mの挿入



挿入

hを根とする部分木にうまく項目itemを挿入する関数

ST_BST1.c (つづき)

```
link insertR(link h, Item item)
  { Key v = key(item), t = key(h->item);
    if (h == z) return NEW(item, z, z, 1);
    if less(v, t)
        h->l = insertR(h->l, item);
    else h->r = insertR(h->r, item);
    (h->N)++; return h;
  }

void STinsert(Item item)
  { head = insertR(head, item); }
```

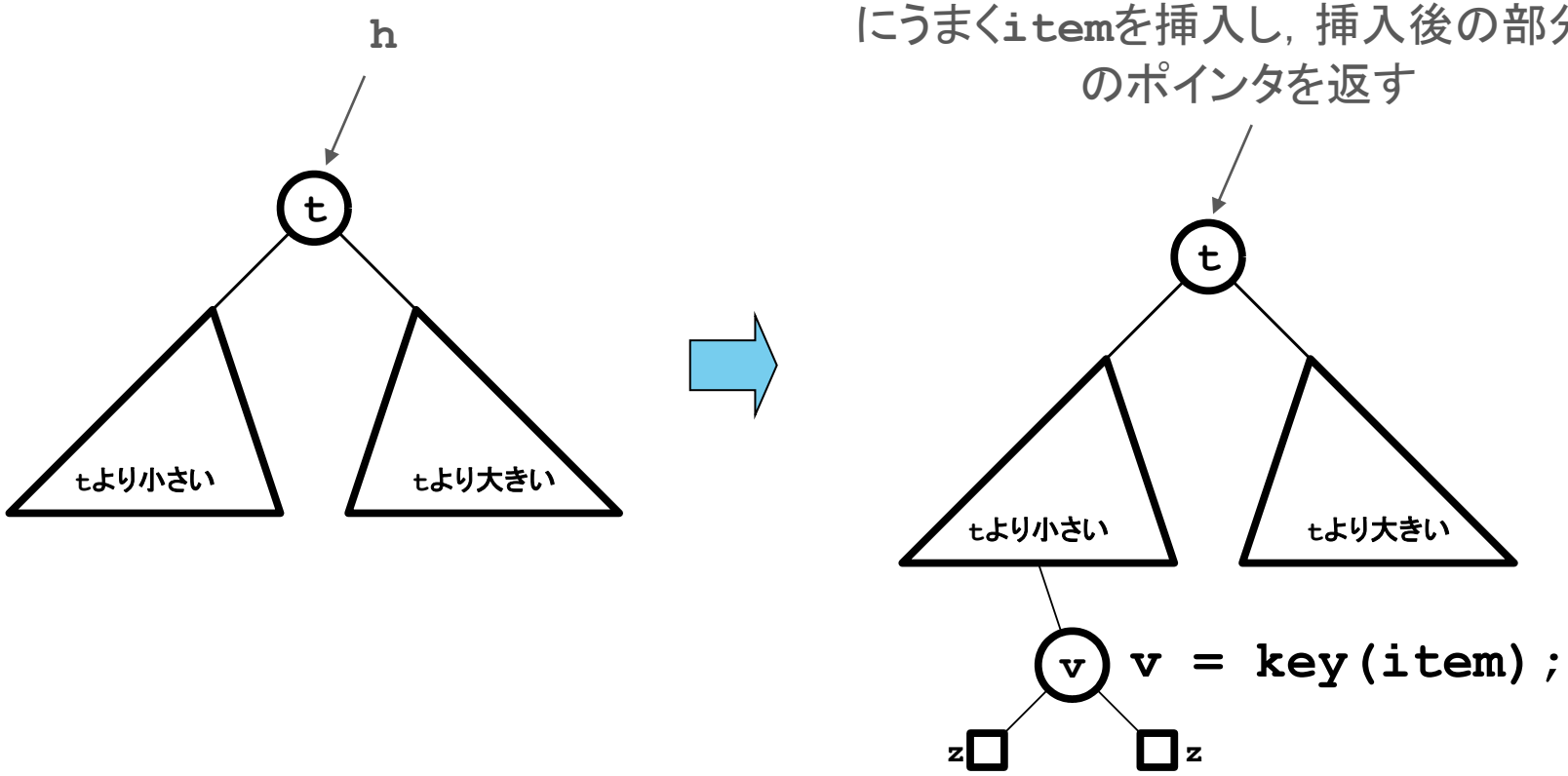
headを根とする部分木，つまり木全体にうまく項目itemを挿入しなさい．そして，その挿入された部分木の根を，改めてheadに代入しなさい，ということ

挿入

ここで，項目itemのキーをvと仮定する(v=key(item);)
ここで，hを根とする部分木の根のキーをtと仮定する(t=key(h->item);)

```
link insertR(link h, Item item)
```

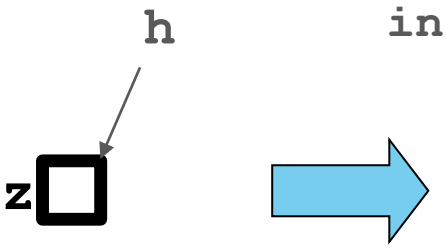
insertRは，hによって指される部分木にうまくitemを挿入し，挿入後の部分木のポインタを返す



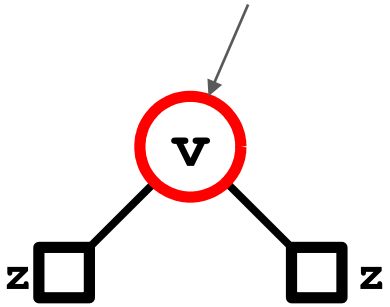
挿入

もし，hが，外部節点であったら

```
if (h == z) return NEW(item, z, z, 1);
```



insertRは，新たな節点として，項目itemを持つ節点を生成し，その節点へのポインタを返す



うまく新たな節点を挿入できた！！

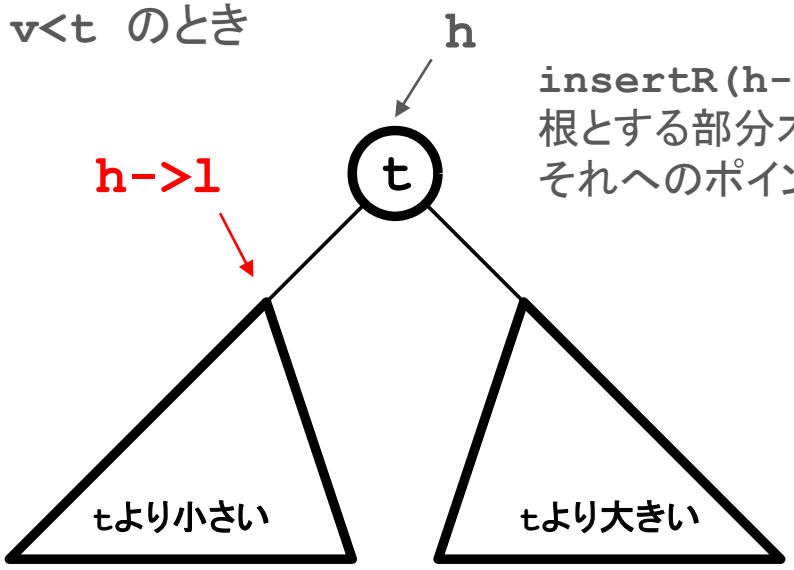
挿入

ここで，項目itemのキーをvと仮定する(v=key(item);)
ここで，hを根とする部分木の根のキーをtと仮定する(t=key(h->item);)

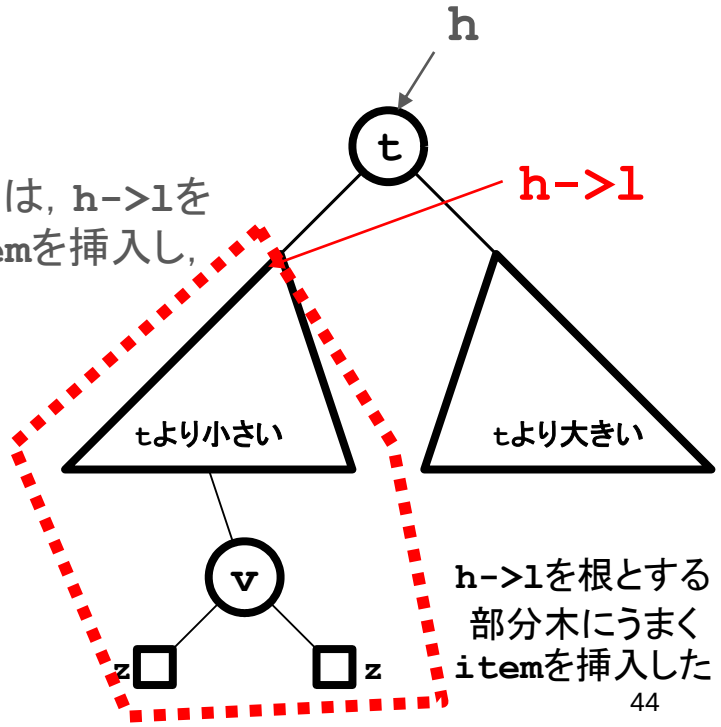
もし，hが，外部節点でなかったら

```
if less(v, t)
      h->l = insertR(h->l, item);
else h->r = insertR(h->r, item);
(h->N)++; return h;
```

v<t のとき



insertR(h->l, item)は，h->lを根とする部分木にうまくitemを挿入し，それへのポインタを返す



h->l =insertR(h->l, item);

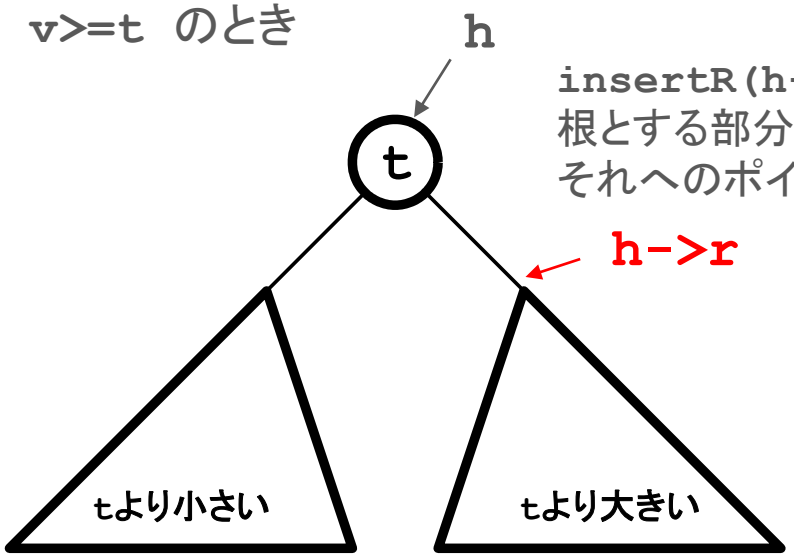
挿入

ここで，項目itemのキーをvと仮定する(v=key(item);)
ここで，hを根とする部分木の根のキーをtと仮定する(t=key(h->item);)

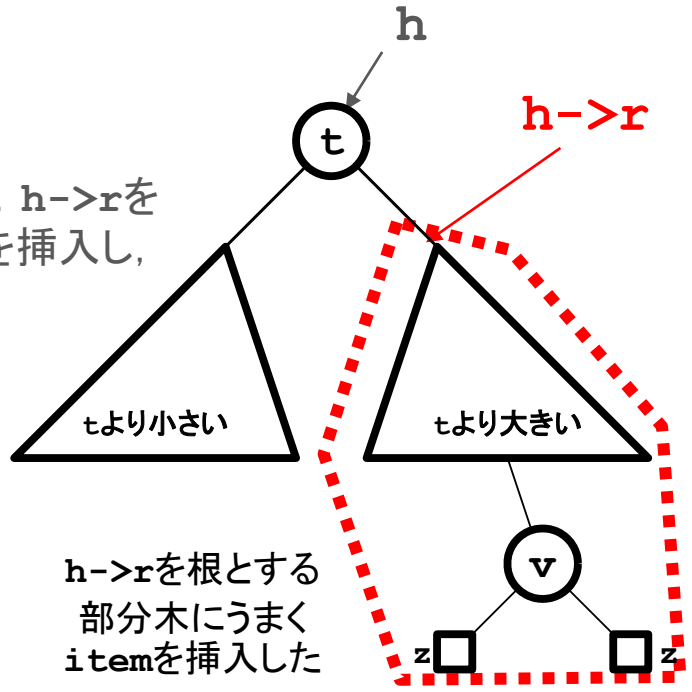
もし，hが，外部節点でなかったら

```
if less(v, t)
      h->l = insertR(h->l, item);
else h->r = insertR(h->r, item);
(h->N)++; return h;
```

v>=t のとき

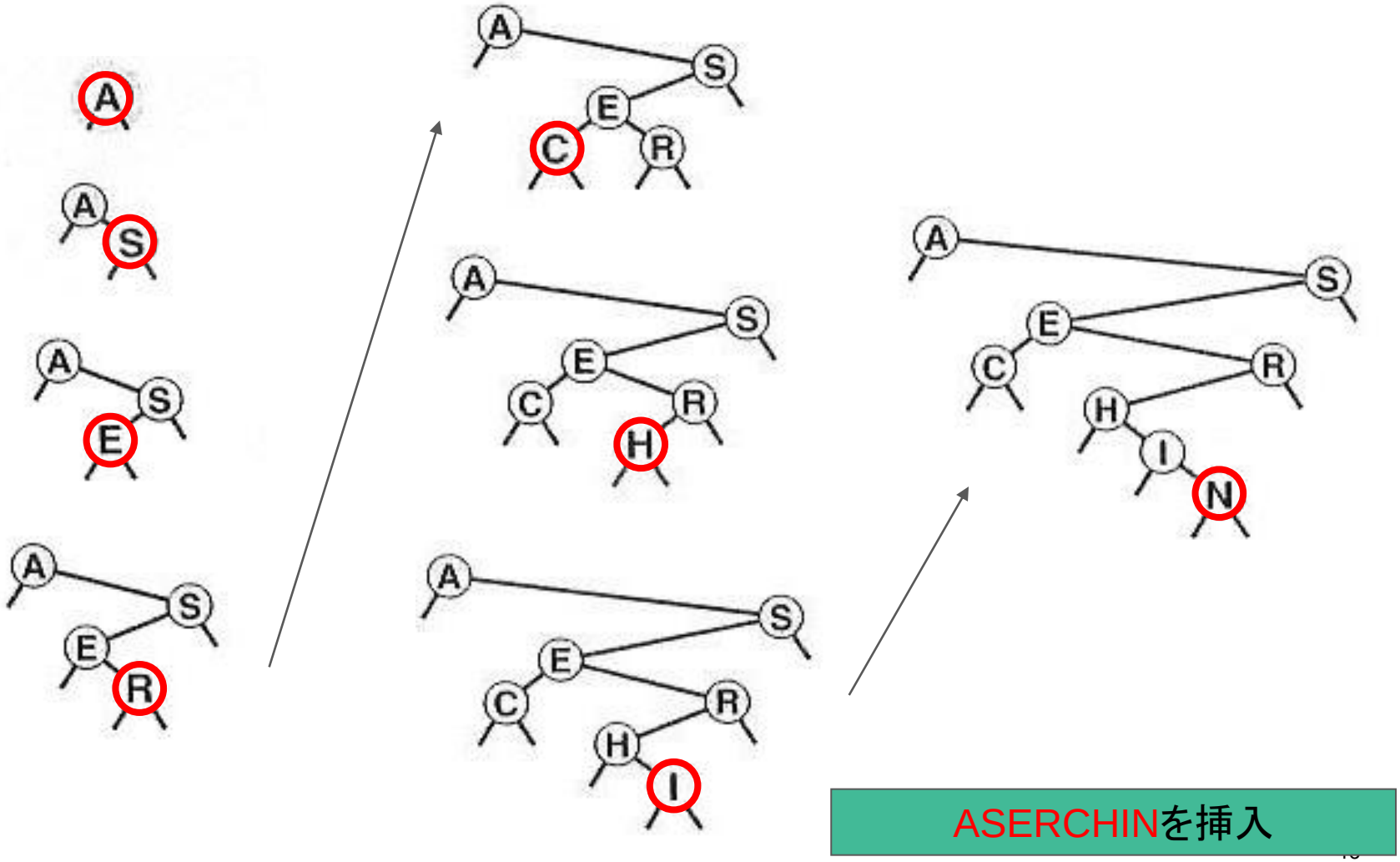


insertR(h->r, item)は，h->rを根とする部分木にうまくitemを挿入し，それへのポインタを返す



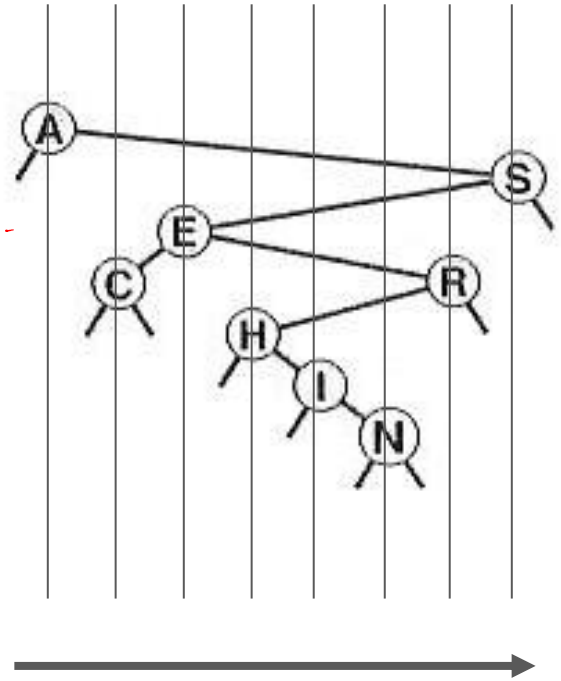
h->r =insertR(h->l, item);

挿入



整列

- 2分探索木の節点を左から順に見ると，すでに整列ずみなっている



根から見て，左の部分木，根の節点，右の部分木と見ればよい
→ 中央順走査

整列

ST_BST1.c（つづき）

```
void sortR(link h, void (*visit)(Item))
  {
    if (h == z) return;
    sortR(h->l, visit);
    visit(h->item);
    sortR(h->r, visit);
  }
void STsort(void (*visit)(Item))
  { sortR(head, visit); }
```

中央順走査

性能

- 性質12.6
  - N個のランダムなキーから生成された2分探索木では，1回の成功探索につき，平均2lnN=1.39lgN回の比較を必要とする
- 性質12.7
  - N個のランダムなキーから生成された2分探索木では，1回の不成功探索あるいは挿入につき，平均2lnN=1.39lgN回の比較を必要とする

キーの挿入順がランダムであることに注意

性能

距離=その節点のレベル

全内部節点のレベルの合計=内部道長

成功探索の平均比較回数
= (内部節点のレベル+1)の平均 = (内部道長/節点数+1)

内部節点の総数で平均をとる

- 性質12.6
  - N個のランダムなキーから生成された2分探索木では，1回の成功探索につき，平均2lnN=1.39lgN回の比較を必要とする
    - ある節点に至るまでのキーの比較回数は，(根からその節点までの距離+1)である
      - すなわち(平均の比較回数)=(内部道長/節点数+1)
    - 木の内部道長は，以下のように表され，クイックソートと同じ議論により，C_N=約2NlnN．N個の節点で平均して2lnN．約2lnN回(1回を無視).

$$C_N = N - 1 + \frac{1}{N}\sum_{1 \le k \le N}(C_{k-1} + C_{N-k}), C_1 = 1$$

性能

距離=その節点のレベル

全内部節点のレベルの合計=内部道長

成功探索の平均比較回数
= (内部節点のレベル+1)の平均 = (内部道長/節点数+1)

内部節点の総数で平均をとる

- 性質12.6
  - N個のランダムなキーから生成された2分探索木では，1回の成功探索につき，平均2lnN=1.39lgN回の比較を必要とする
    - ある節点に至るまでのキーの比較回数は，(根からその節点までの距離+1)である
      - すなわち(平均の比較回数)=(内部道長/節点数+1)
    - 木の内部道長は，以下のように表され，クイックソートと同じ議論により，C_N=約2NlnN．N個の節点で平均して2lnN．約2lnN回(1回を無視).

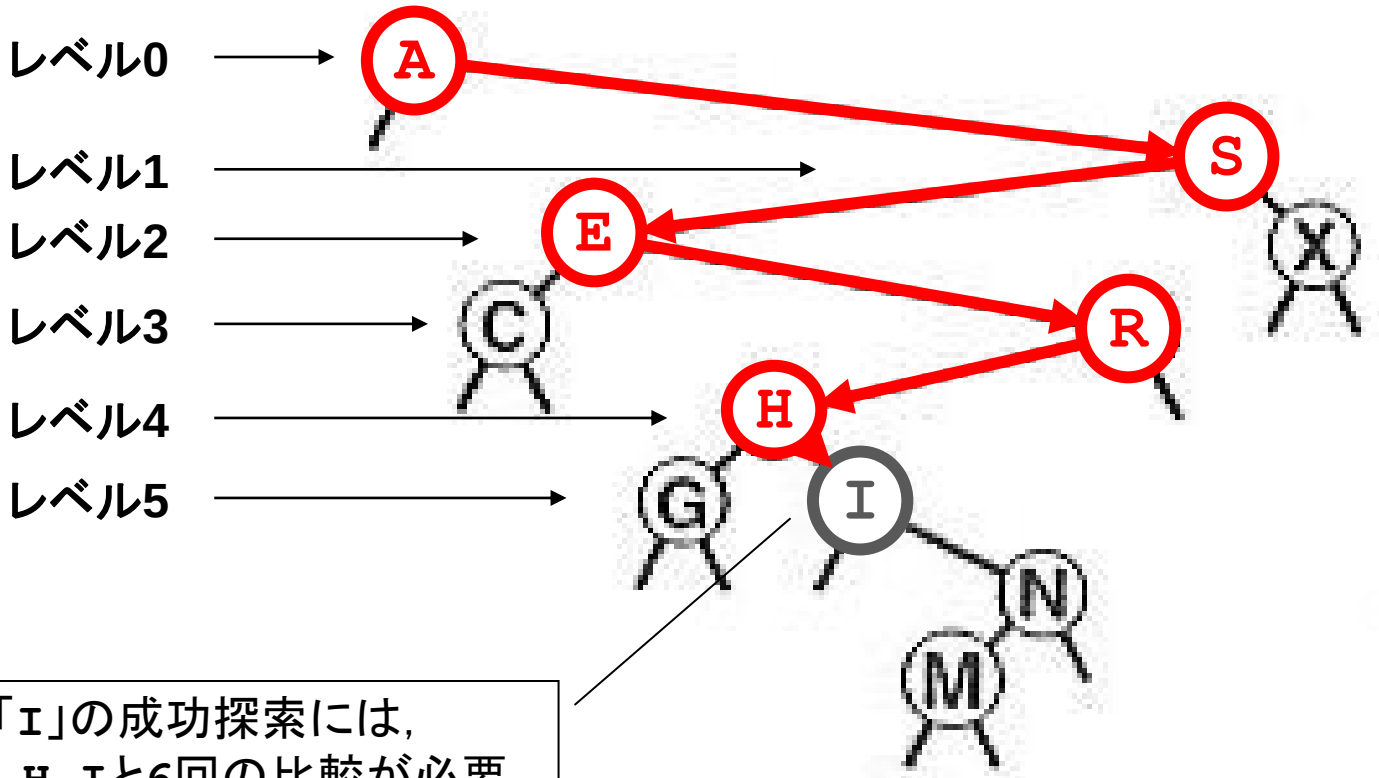
$$C_N = N - 1 + \frac{1}{N}\sum_{1 \le k \le N}(C_{k-1} + C_{N-k}), C_0 = 0, C_1 = 0$$

成功探索

全内部節点のレベルの合計=内部道長

成功探索の平均比較回数
= (内部節点のレベル+1)の平均 = (内部道長/節点数+1)

内部節点の総数で平均をとる



節点「I」の成功探索には，
A,S,E,R,H,Iと6回の比較が必要．

これは，Iのレベル5に(+1)したもの

性能

全外部節点のレベルの合計=外部道長

不成功探索の平均比較回数
= (外部節点のレベル)の平均 = (外部道長/節点数)

外部節点の総数で平均をとる

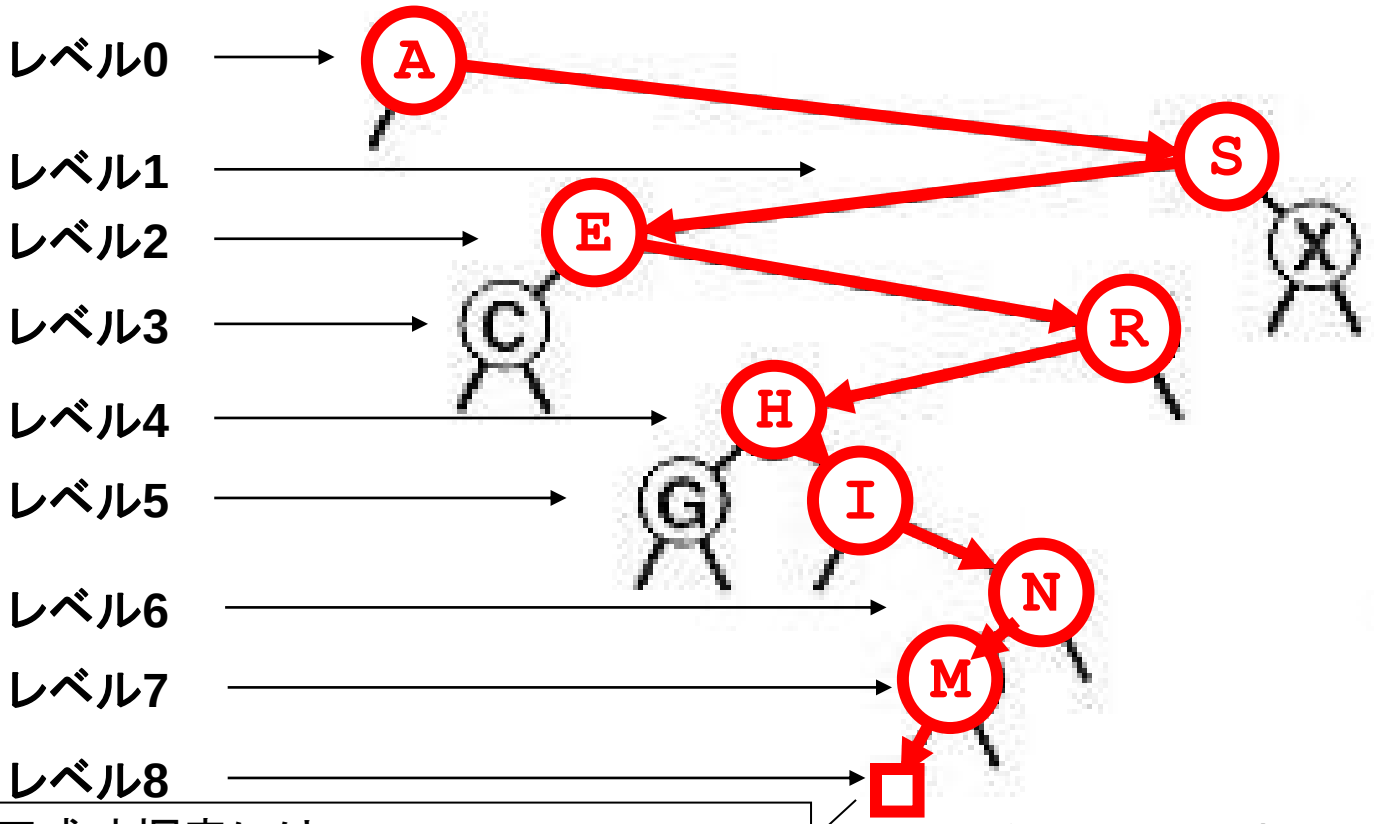
- 性質12.7
  - N個のランダムなキーから生成された2分探索木では，1回の不成功探索あるいは挿入につき，平均2lnN=1.39lgN回の比較を必要とする
    - N個の内部節点を持つ2分探索木は，N+1個の外部節点を持つ
      - **外部節点に至るまでのキーの探索回数**は，ちょうど**(根からその外部節点までの距離)**に等しい

不成功探索

全外部節点のレベルの合計=外部道長

不成功探索の平均比較回数
= (外部節点のレベル)の平均 = (外部道長/節点数)

外部節点の総数で平均をとる



節点「J」の不成功探索には，A,S,E,R,H,I,N,Mと8回の比較が必要．

これは，この位置の外部節点のレベル8そのもの

外部節点(通常は省略されていた)

木の性質
(木の外部道長)=(木の内部道長)+2N.
(13_tree1.pdf, 性質5.7)

内部道長の平均は，内部節点の数Nで平均をとったから，N倍する

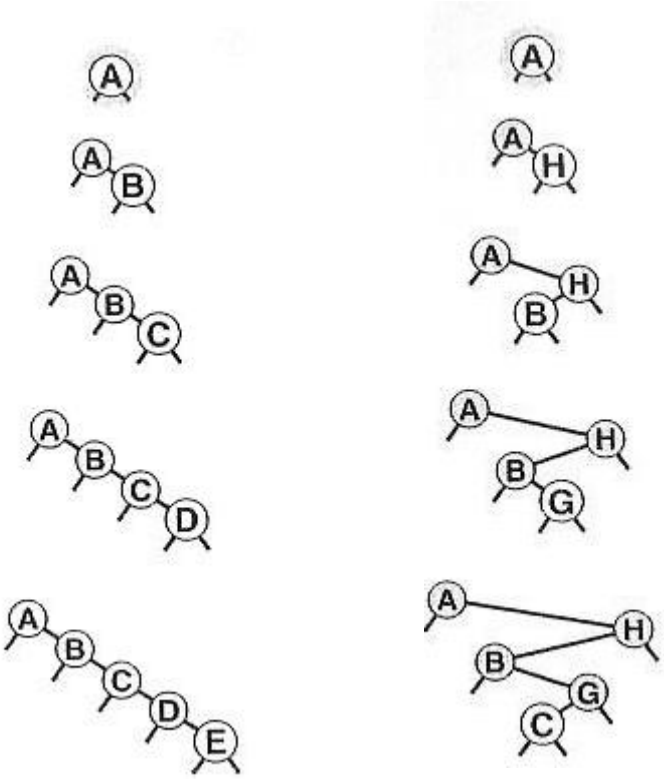
外部道長の平均は，外部節点の数(N+1)で平均をとったから，(N+1)倍する

- 性質12.7
  - (続き)
    - 今，成功探索の平均比較回数をN_s回とおくと，N_s =(木の内部道長の平均+1)．両辺をN倍すると，**$N \times N_s$=(木の内部道長)+N.**
    - また，不成功探索の平均比較回数をN_f回とおくと，N_f =木の外部道長の平均．両辺を(N+1)倍すると，**$(N+1) \times N_f$=(木の外部道長).**
    - 以上より，**$(N+1) \times N_f = N \times N_s + N$** が成立する．(N+1)で両辺を割ると，
      $N_f = N_s + 1$．ただし，Nが十分に大きく，N/(N+1)=1とした．
    - この結果は，成功探索の際の比較回数より1回だけ，不成功探索の方が比較回数が大きいことを表している．

キーの挿入順がランダムであることに注意

性能

- 性質12.8
  - N個のキーからなる2分探索木での探索は，最悪の場合にはN回の比較が必要となる



最悪の場合の2分探索木

キーの挿入がランダムでない

記号表での挿入と探索のコスト

	最悪の場合			平均の場合		
	挿入	探索	選択	挿入	成功探索	不成功探索
キー添字配列	1	1	M	1	1	1
整列した配列	N	N	1	N/2	N/2	N/2
整列したリンクリスト	N	N	N	N/2	N/2	N/2
整列していない配列	1	N	NlgN	1	N/2	N
整列していないリスト	1	N	NlgN	1	N/2	N
2分探索	N	lgN	1	N/2	lgN	lgN
2分探索木	N	N	N	lgN	lgN	lgN
赤黒木	lgN	lgN	lgN	lgN	lgN	lgN
ランダム化木	N	N	N	lgN	lgN	lgN
ハッシュ法	1	N	NlgN	1	1	1

N:項目の数　　M:表のサイズ

整列した配列に項目を蓄える必要があるため，挿入は整列した配列と同じになる

記号表での挿入と探索のコスト

	最悪の場合			平均の場合		
	挿入	探索	選択	挿入	成功探索	不成功探索
キー添字配列	1	1	M	1	1	1
整列した配列	N	N	1	N/2	N/2	N/2
整列したリンクリスト	N	N	N	N/2	N/2	N/2
整列していない配列	1	N	NlgN	1	N/2	N
整列していないリスト	1	N	NlgN	1	N/2	N
2分探索	N	lgN	1	N/2	lgN	lgN
2分探索木	N	N	N	lgN	lgN	lgN
赤黒木	lgN	lgN	lgN	lgN	lgN	lgN
ランダム化木	N	N	N	lgN	lgN	lgN
ハッシュ法	1	N	NlgN	1	1	1

N:項目の数　　M:表のサイズ

整列した配列に項目を蓄える必要があるため，挿入は整列した配列と同じになる

まとめ

- 記号表
- 記号表抽象データ型
- 逐次探索
- 2分探索法
- 2分探索木
- 性能