アルゴリズムとデータ構造

基数探索（その１）

目次

- 基数探索
- 離散探索木
- トライ

基数探索

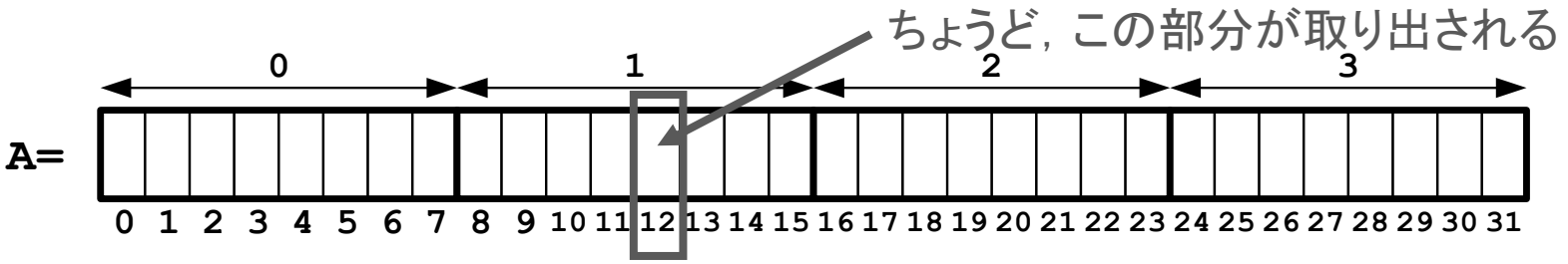
- これまでの探索法
  - 探索の各ステップでキー全体を比較
- 基数探索法
  - 一度に探索キーの「部分」(ビットやバイト)を比較

ビット，バイト，ワード

- 2進数AのBビット目を取り出したいとき，

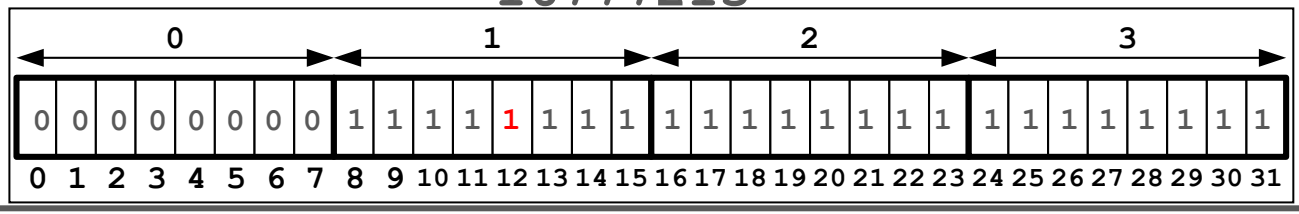
```
#define bitsword 32

#define digit(A, B) (((A) >> (bitsword-((B)+1))) & 1)
```

Aの12ビット目を取り出すとき，B=12として



ビット，バイト，ワード

16777215 =



radix_test.c

```
#include <stdio.h>
#define bitsword 32
#define digit(A, B) (((A) >> (bitsword-((B)+1))) & 1)

int main(void)
{
    int b, d;

    d = 16777215;
    b = digit(d, 12);
    printf("12th bit of %d = %d¥n", d, b);
    return 0;
}
```

```
$ ./radix_test
12th bit of 16777215 = 1
$
```

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 = {0, "NULL"};
```

ここでは，**NULLitem**のキーの値は，0と定義する

記号表抽象データ型

ItemD.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)

#define bitsword 17
#define digit(A, B) (((A) >> (bitsword-((B)+1))) & 1)

#ifndef _ITEM_D_
extern Item NULLitem;
#endif

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

ItemD.hで，これらを定義しておく

bitswordを**17**にしておく(これで，約100,000までの数を表現できる)

記号表抽象データ型

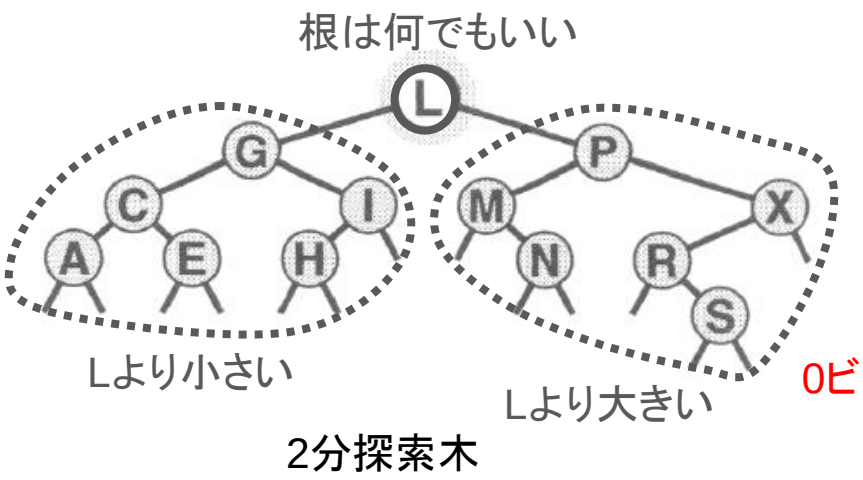
ItemD.c

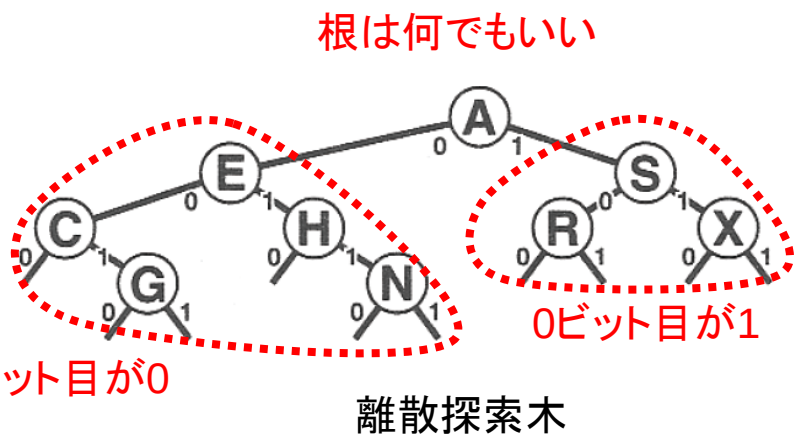
ItemD.cでNULLitemをグローバル変数として定義し，適当な値に初期化する

```
#define _ITEM_D_
#include <stdio.h>
#include <stdlib.h>
#include "ItemD.h"

Item NULLitem = {0, "NULL"};

Key ITEMrand(void)
          { return rand()%100000+1; }
int ITEMscan(Key *x)
          { return scanf("%d", x); }
void ITEMshow(Item x)
          { printf("%3d ", key(x)); }
```

離散探索木

- 2分探索木において，木の分岐をキーの各ビットの順で行う木

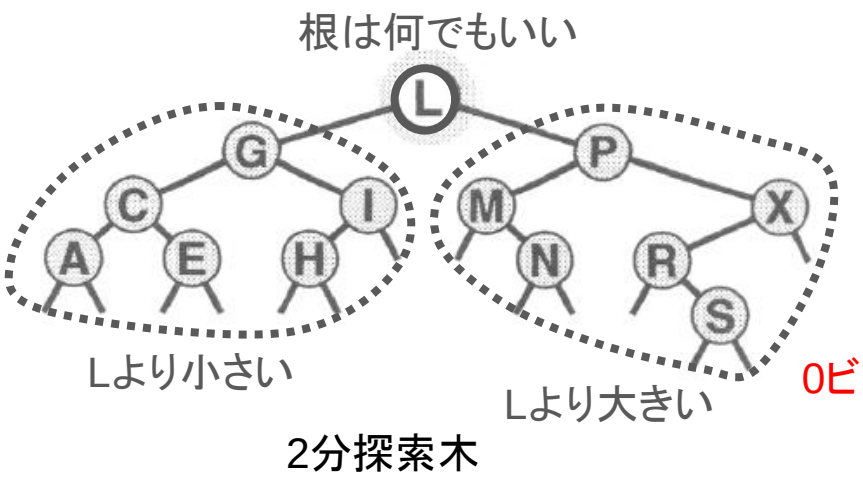


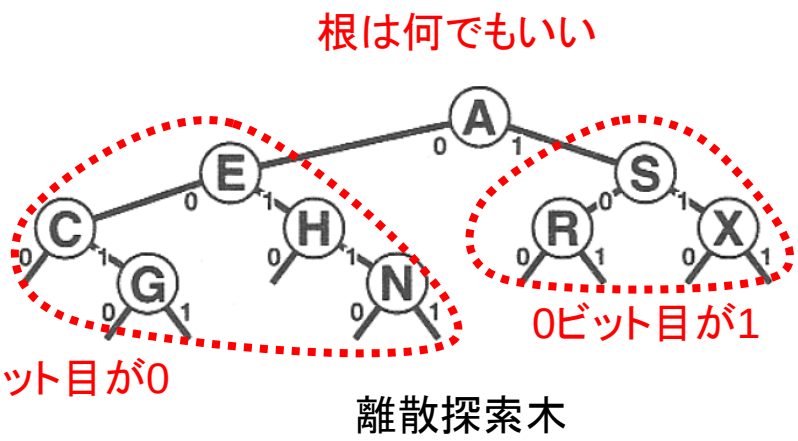



```
A 00001
S 10011
E 00101
R 10010
C 00011
H 01000
I 01001
N 01110
G 00111
X 11000
M 01101
P 10000
L 01100
```

離散探索木

- 2分探索木において，木の分岐をキーの各ビットの順で行う木

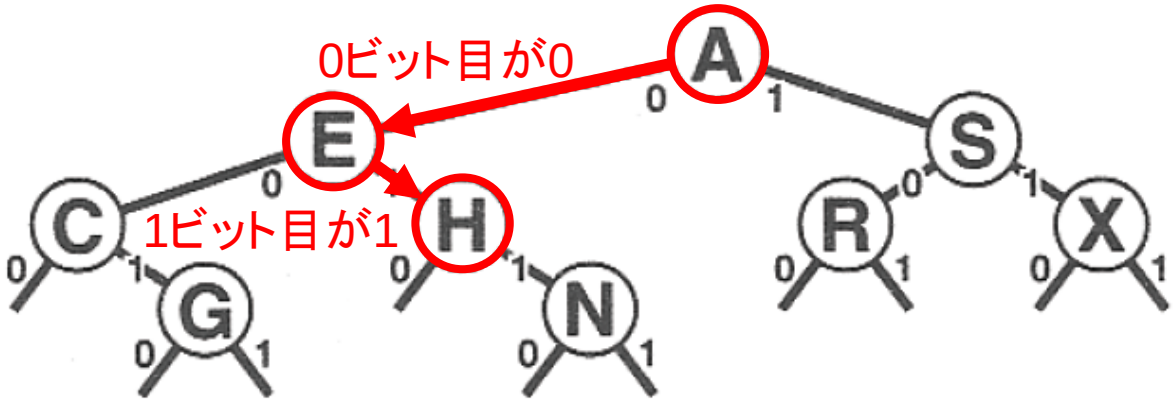





```
A 00001
S 10011
E 00101
R 10010
C 00011
H 01000
I 01001
N 01110
G 00111
X 11000
M 01101
P 10000
L 01100
```

探索

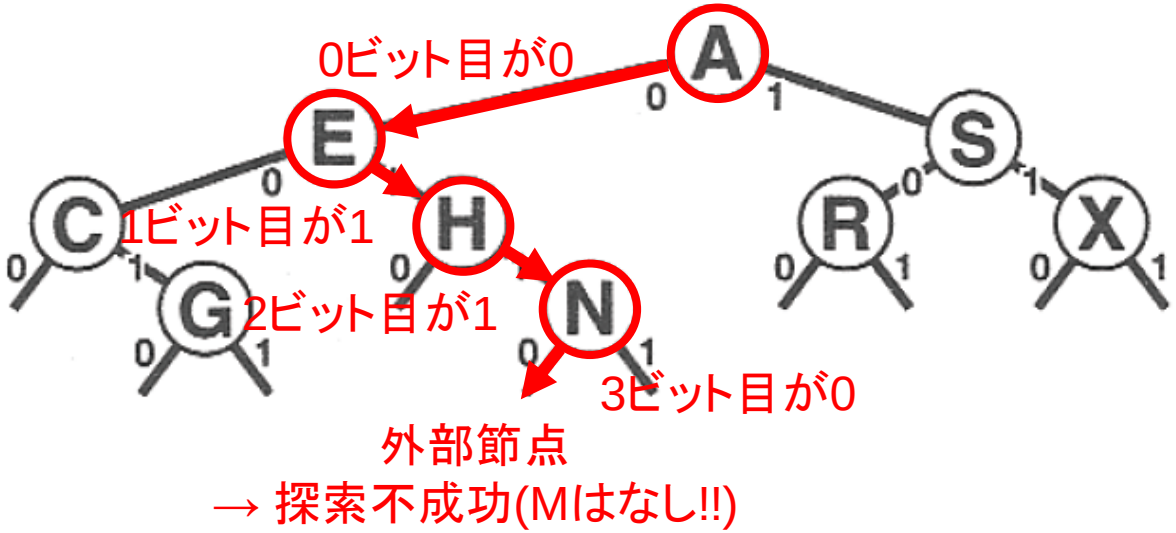
H=01000の探索
　0ビット目=0
　1ビット目=1



A	00001
S	10011
E	00101
R	10010
C	00011
H	**01000**
I	01001
N	01110
G	00111
X	11000
M	01101
P	10000
L	01100

探索

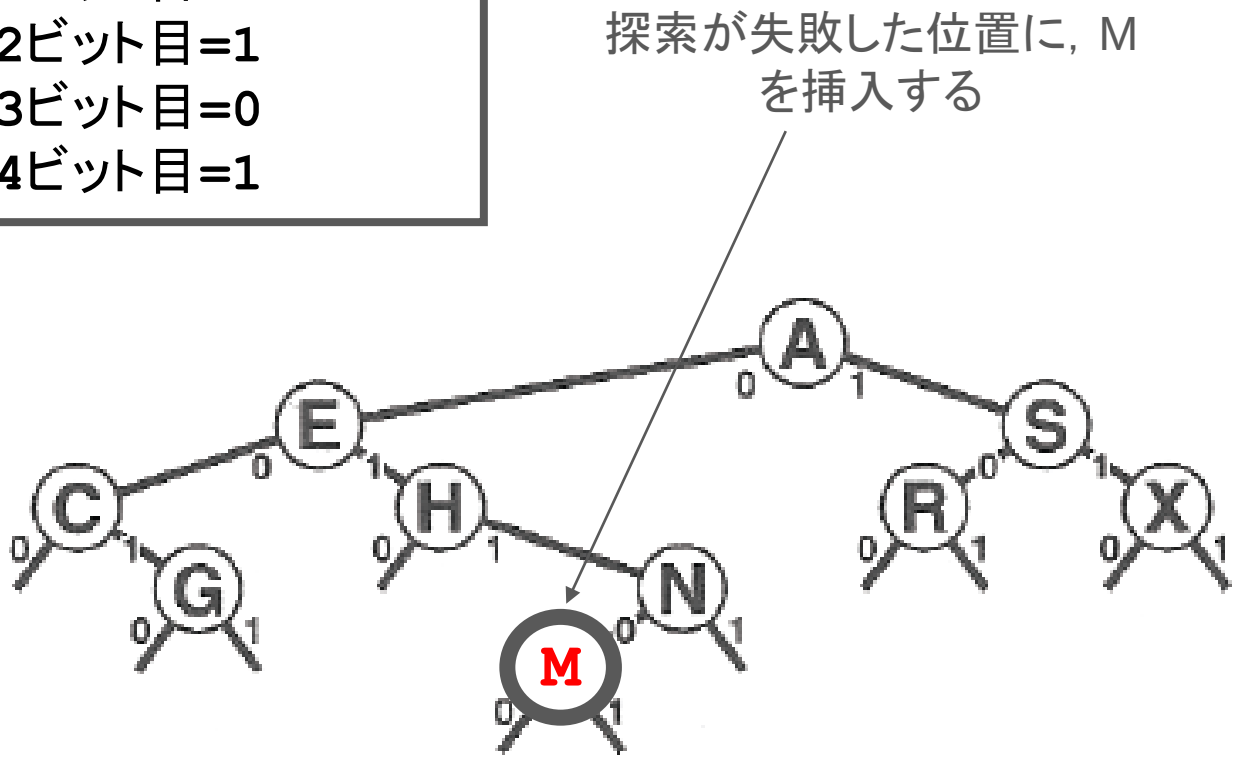
M=01101の探索
- 0ビット目=0
- 1ビット目=1
- 2ビット目=1
- 3ビット目=0
- 4ビット目=1



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110
G	00111
X	11000
M	01101
P	10000
L	01100

挿入

M=01101の探索
　0ビット目=0
　1ビット目=1
　2ビット目=1
　3ビット目=0
　4ビット目=1



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110
G	00111
X	11000
M	01101
P	10000
L	01100

離散探索木での探索操作

2分探索木のSTsearch()関数，searchR関数を次のものに変更

```
Item searchR(link h, Key v, int w)
  { Key t = key(h->item);
    if (h == z) return NULLitem;
    if eq(v, t) return h->item;
    if (digit(v, w) == 0)
          return searchR(h->l, v, w+1);
    else return searchR(h->r, v, w+1);
  }
Item STsearch(Key v)
  { return searchR(head, v, 0); }
```

外部節点ならば探索不成功

そのキーが見付かれば探索成功

w番目のビットに基づいて，分岐

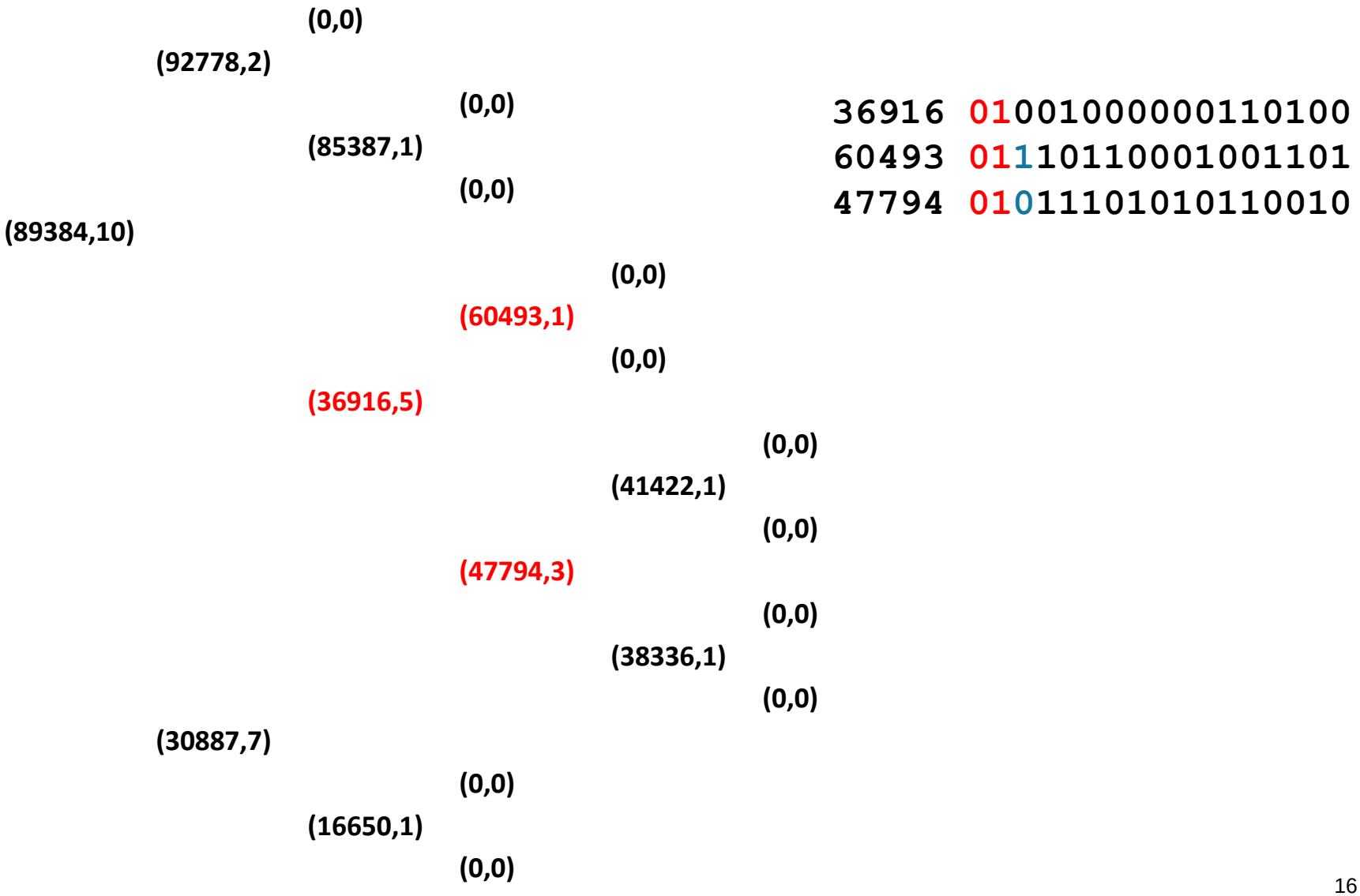
searchR関数に関して，2分探索木と離散探索木を比較してみる事

離散探索木

- 性質15.1
  - ランダムなキーN個から作られた離散探索木での探索と挿入は，平均lgN回の比較を行う．比較回数は探索キーのビット長を越えることはない

16650 30887 38336 47794 41422 36916 60493 89384 85387 92778

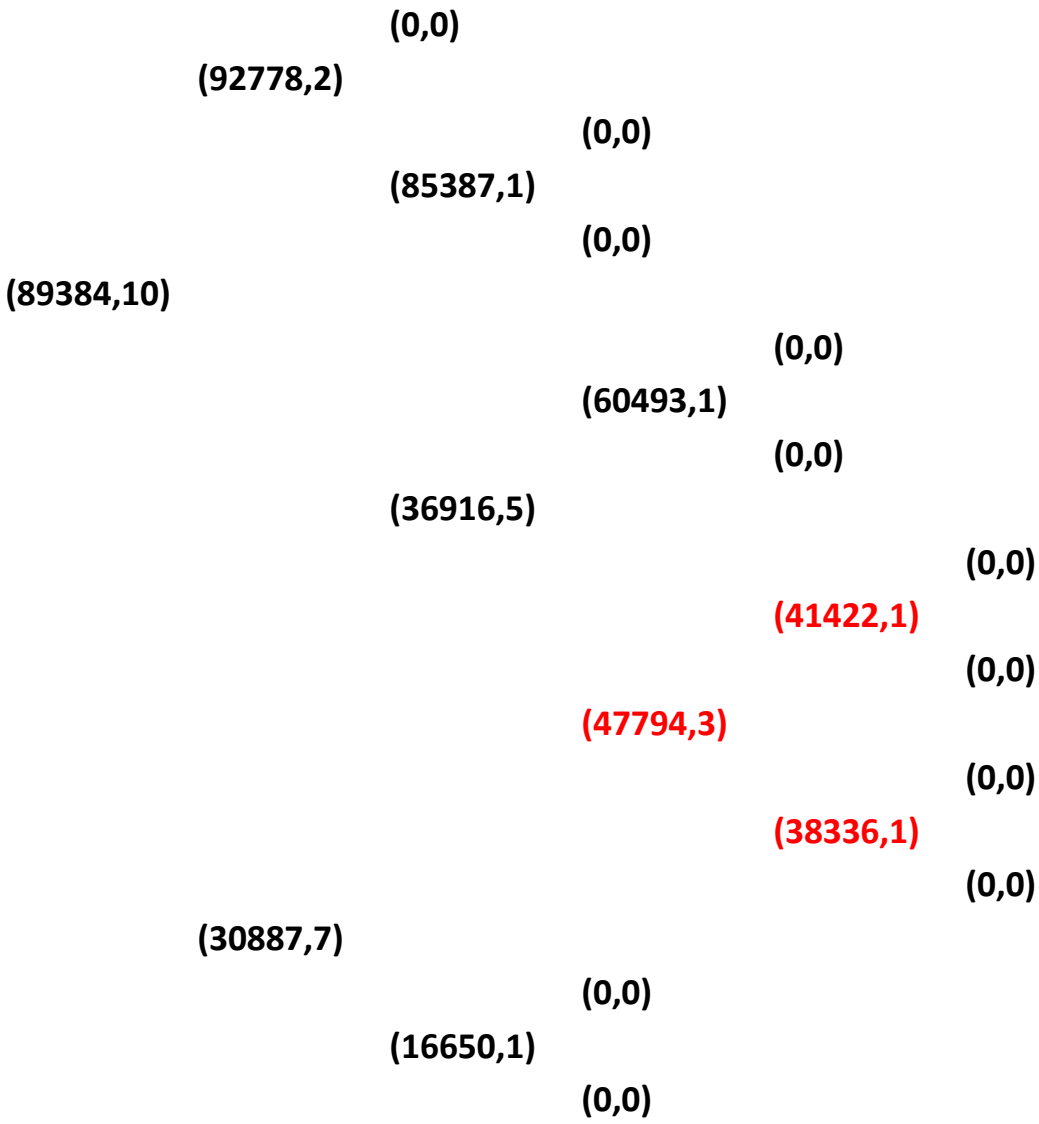
10 keys 10 distinct keys




```
36916 0100100000110100
60493 0111011000100110 1
47794 0101110101011001 0
```

16650 30887 38336 47794 41422 36916 60493 89384 85387 92778

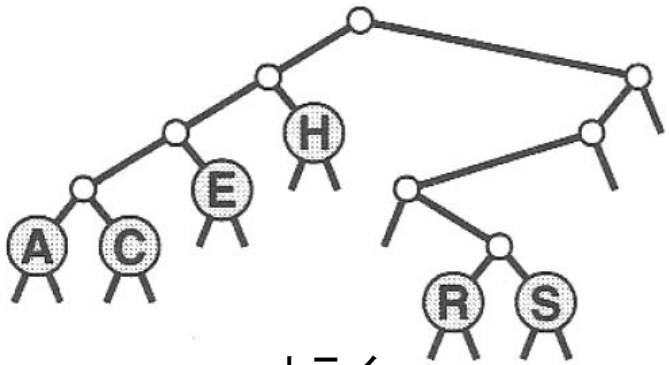
10 keys 10 distinct keys




```
36916 0100100000110100
60493 0111011000100l101
47794 0101111010110010
38336 0100101011100000
41422 0101000011100lll0
```

トライ（Trie）

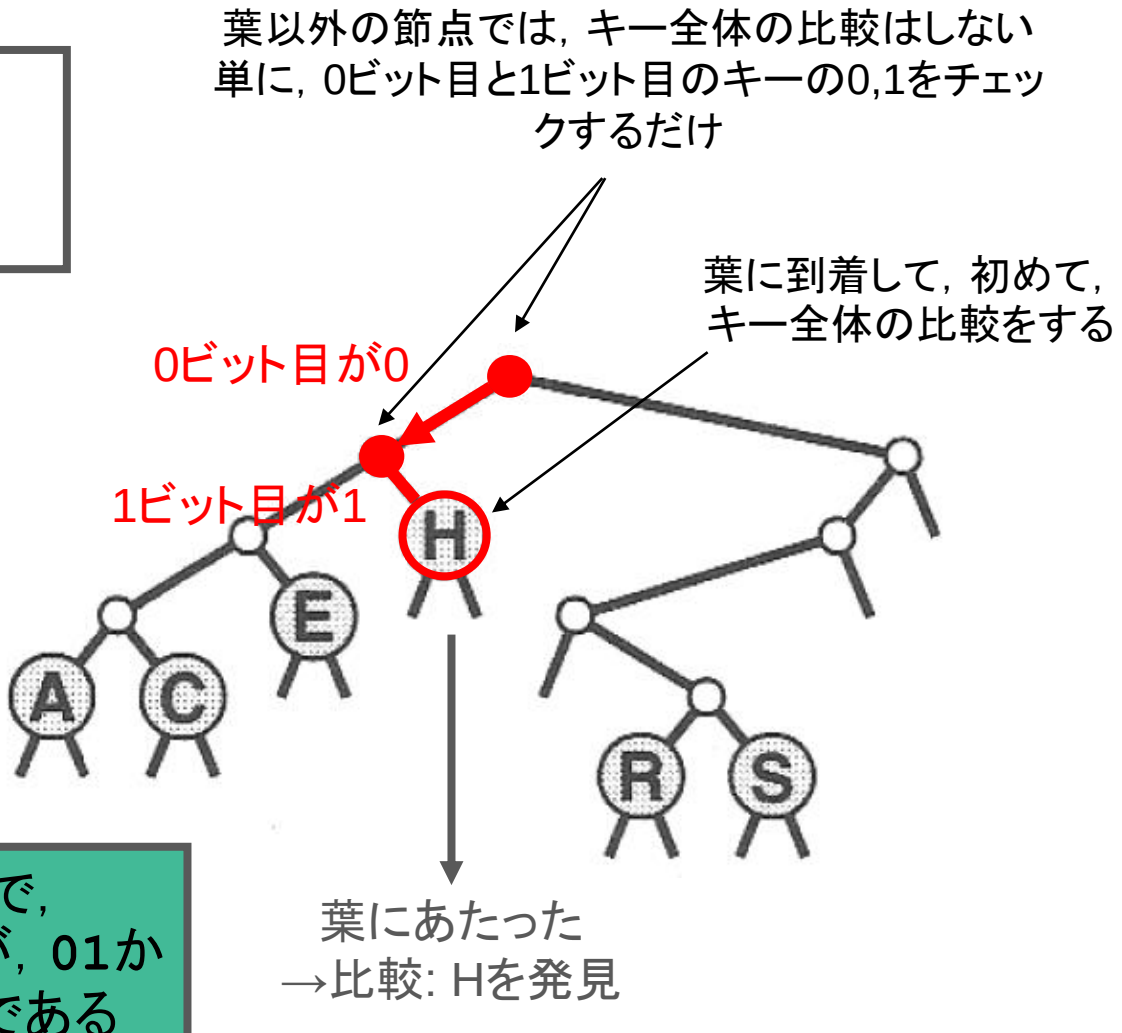
- キーを葉にだけ置くとどうなるか？
  - 各キーは，キーの先頭からのビットパタンで表される道の終端の葉に格納される
  - キー全体の比較は，最後の葉でだけ
  - こうすると，葉以外の内部節点では，キーの比較をしなくてよい!



トライ

探索

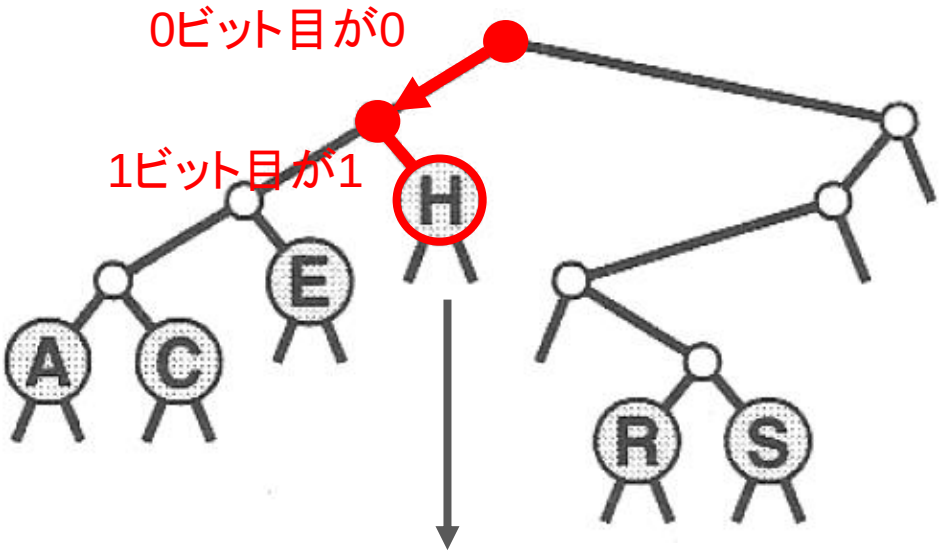
H=01000の探索
　0ビット目=0
　1ビット目=1



この木の中で，
H=01000だけが，01から始まるキーである

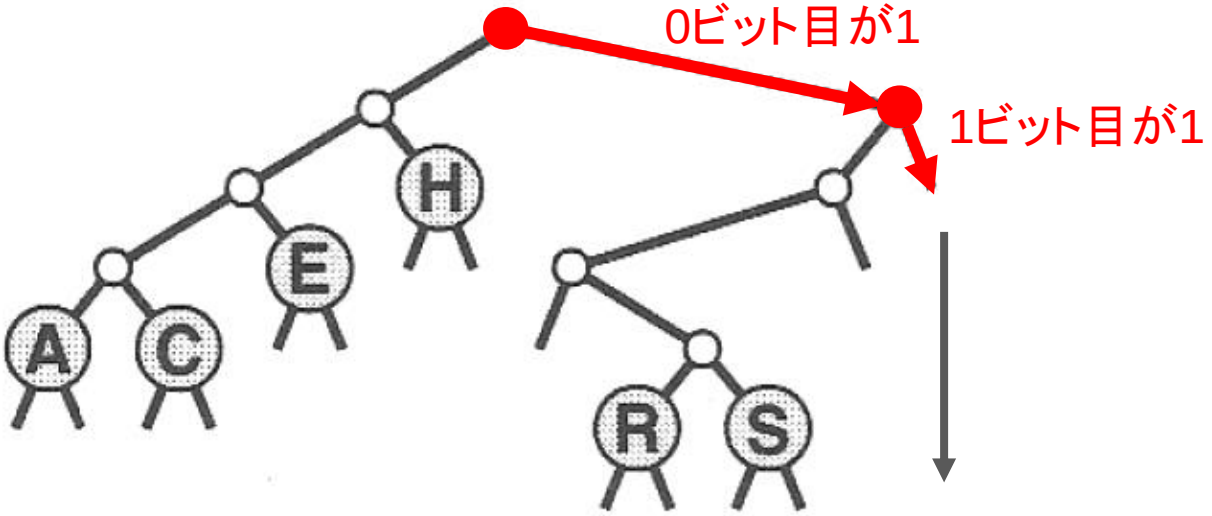
探索

I=01001の探索
　0ビット目=0
　1ビット目=1



探索

X=11000の探索
　0ビット目=1
　1ビット目=1



外部節点にあたった
→Xを発見できない

探索

```
#define leaf(A) ((h->l == z) && (h->r == z))
Item searchR(link h, Key v, int w)
  { Key t = key(h->item);
    if (h == z) return NULLitem;
    if (leaf(h))
      return eq(v, t) ? h->item : NULLitem;
    if (digit(v, w) == 0)
      return searchR(h->l, v, w+1);
    else return searchR(h->r, v, w+1);
  }
Item STsearch(Key v)
  { return searchR(head, v, 0); }
```

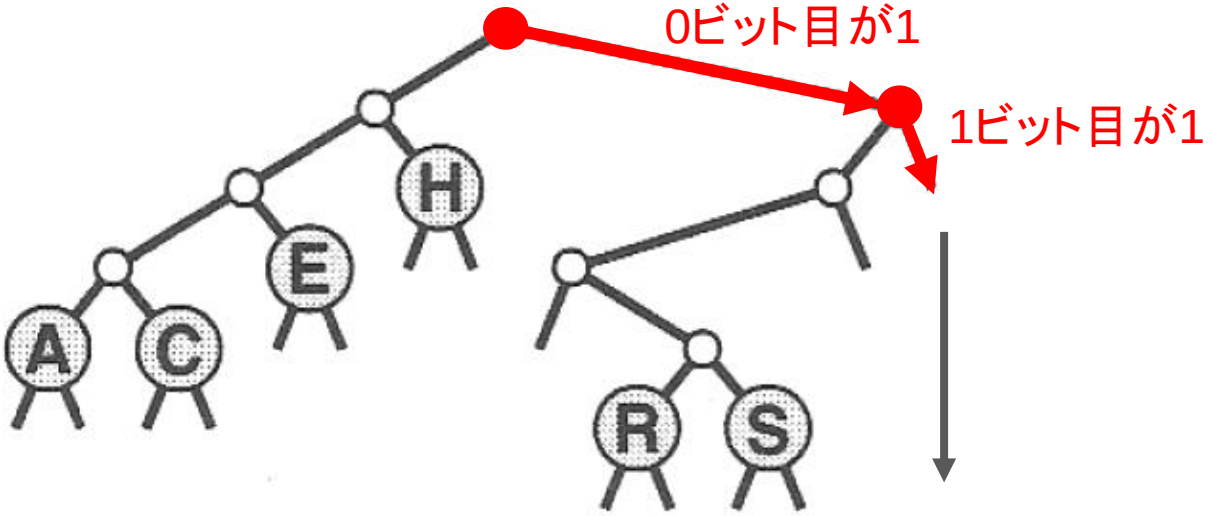
外部節点ならば探索不成功

葉にあたったら，キーと比較して，成功/不成功を判断

w番目のビットに基づいて，分岐

挿入1

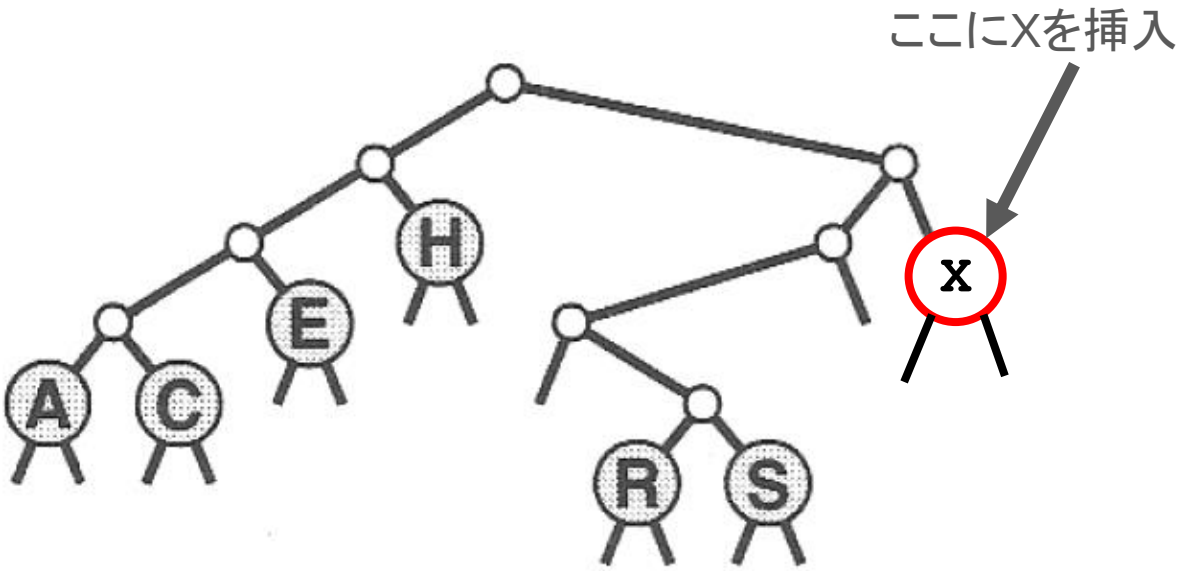
x=11000の挿入
　0ビット目=1
　1ビット目=1



外部節点にあたった
→Xを発見できない

挿入1

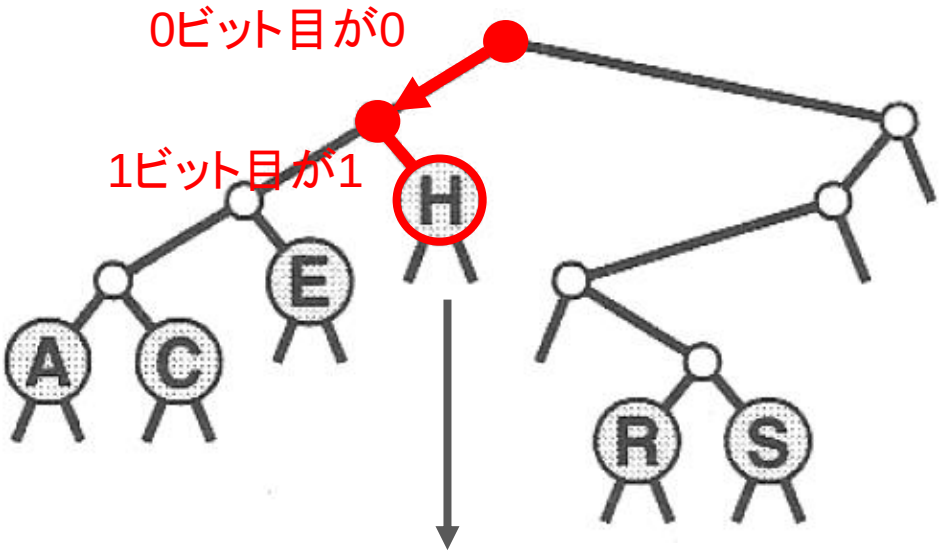
x=11000の挿入
　0ビット目=1
　1ビット目=1



挿入2

I=01001の挿入

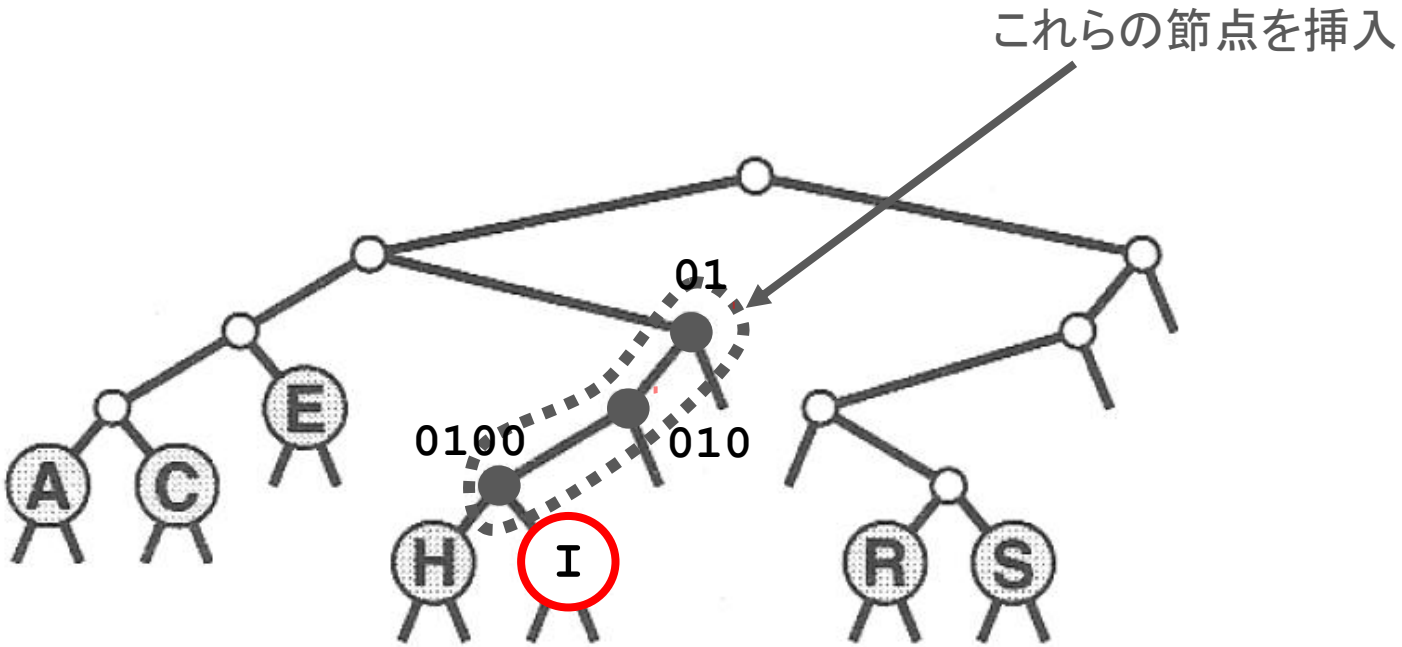
H=01000との差異は，4ビット目



挿入2

I=01001の挿入

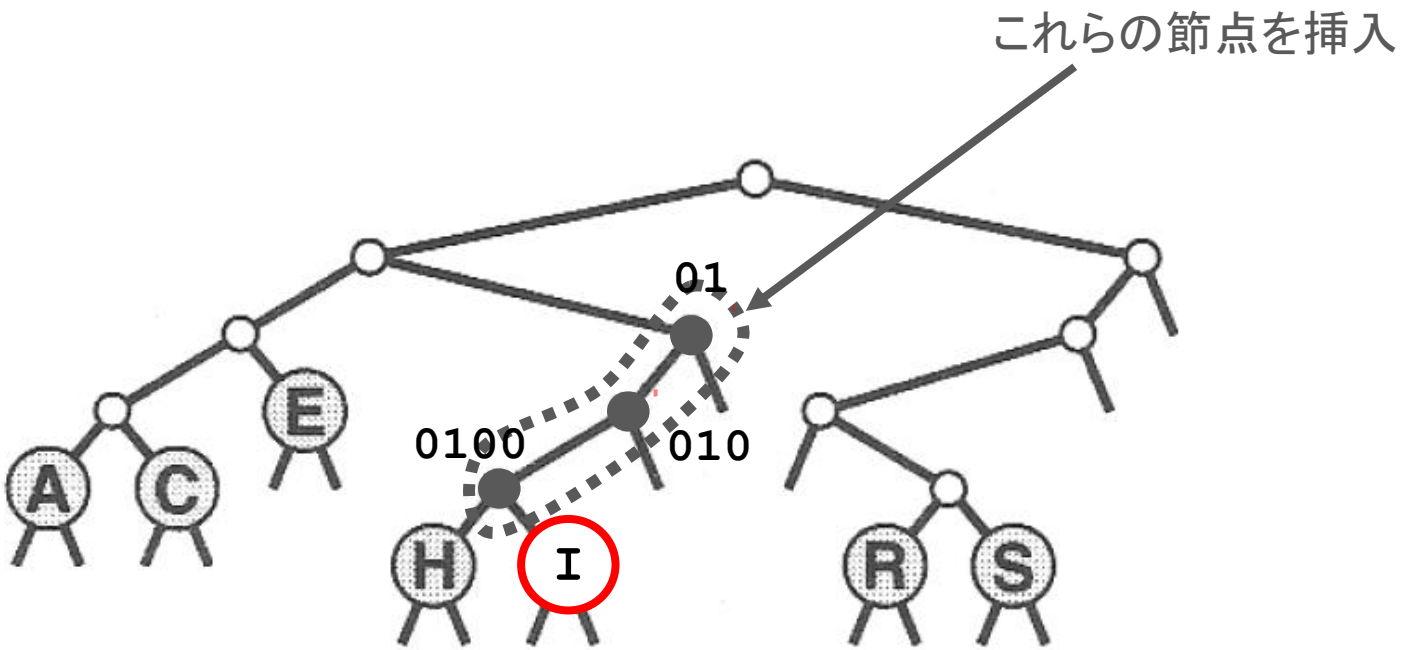
H=01000との差異は，4ビット目



挿入2

I=01001の挿入

H=01000との差異は，4ビット目



挿入

```
void STinit()
  { head = (z = NEW(NULLitem, 0, 0, 0)); }
link insertR(link h, Item item, int w)
  { Key v = key(item);
    if (h == z) return NEW(item, z, z, 1);
    if (leaf(h))
      { return split(NEW(item, z, z, 1), h, w); }
    if (digit(v, w) == 0)
        h->l = insertR(h->l, item, w+1);
    else h->r = insertR(h->r, item, w+1);
    return h;
  }
void STinsert(Item item)
  { head = insertR(head, item, 0); }
```

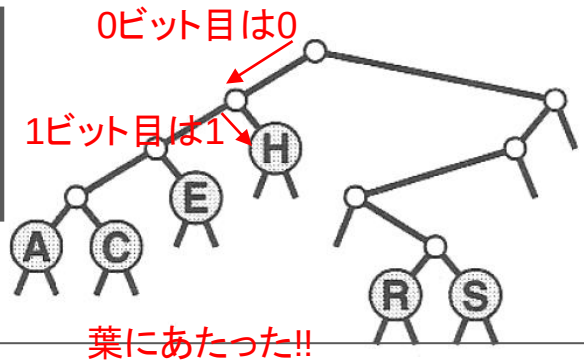
外部節点ならばそこに挿入(挿入1)

葉ならば，いくつかの節点を挿入した後に，挿入(挿入2)

挿入

I=01001の挿入

I=01001
H=01000




```
void STinit()
  { head = (z = NEW(NULLitem, 0, 0, 0)); }
link insertR(link h, Item item, int w)
  { Key v = key(item);
    if (h == z) return NEW(item, z, z, 1);
    if (leaf(h))
      { return split(NEW(item, z, z, 1), h, w); }
    if (digit(v, w) == 0)
        h->l = insertR(h->l, item, w+1);
    else h->r = insertR(h->r, item, w+1);
    return h;
  }
void STinsert(Item item)
  { head = insertR(head, item, 0); }
```

葉ならば，いくつかの節点を挿入した後に，挿入(挿入2)

挿入

新しく節点を作る(下図の赤い節点)

```
link split(link p, link q, int w)
  { link t = NEW(NULLitem, z, z, 2);
    switch(digit(key(p->item), w)*2 + digit(key(q->item), w))
      {
        case 0: t->l = split(p, q, w+1); break;
        case 1: t->l = p; t->r = q; break;
        case 2: t->r = p; t->l = q; break;
        case 3: t->r = split(p, q, w+1); break;
      }
    return t;
  }
```

挿入

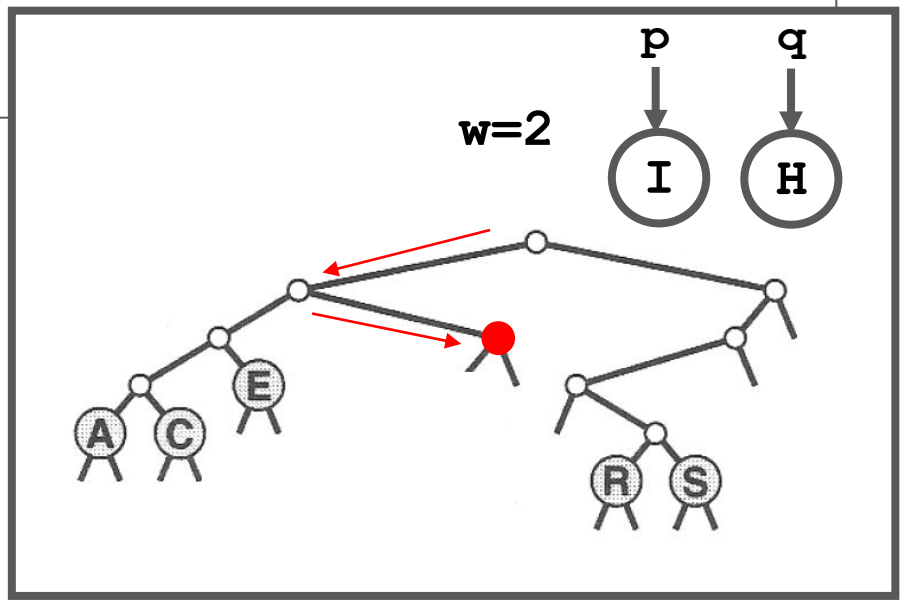
```
link split(link p, link q, int w)
  { link t = NEW(NULLitem, z, z, 2);
    switch(digit(key(p->item), w)*2 + digit(key(q->item), w))
      {
        case 0: t->l = split(p, q, w+1); break;
        case 1: t->l = p; t->r = q; break;
        case 2: t->r = p; t->l = q; break;
        case 3: t->r = split(p, q, w+1); break;
      }
    return t;
  }
```

新しく節点を作る(下図の赤い節点)

I=01001の挿入

I=01001
H=01000

2ビット目が00なので，case 0を実行



挿入

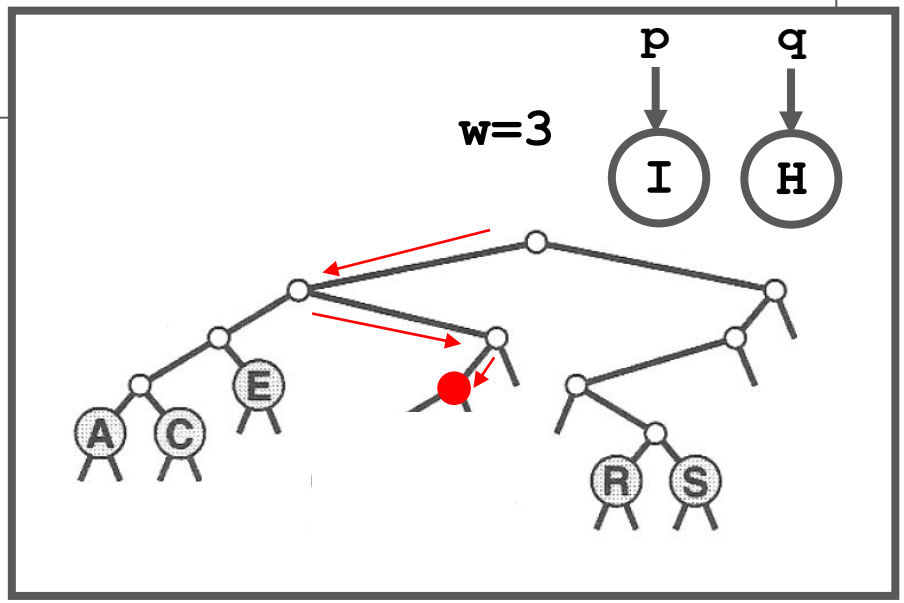
```
link split(link p, link q, int w)
  { link t = NEW(NULLitem, z, z, 2);
    switch(digit(key(p->item), w)*2 + digit(key(q->item), w))
      {
        case 0: t->l = split(p, q, w+1); break;
        case 1: t->l = p; t->r = q; break;
        case 2: t->r = p; t->l = q; break;
        case 3: t->r = split(p, q, w+1); break;
      }
    return t;
  }
```

新しく節点を作る(下図の赤い節点)

I=01001の挿入

I=01001
H=01000

3ビット目が00なので，case 0を実行



挿入

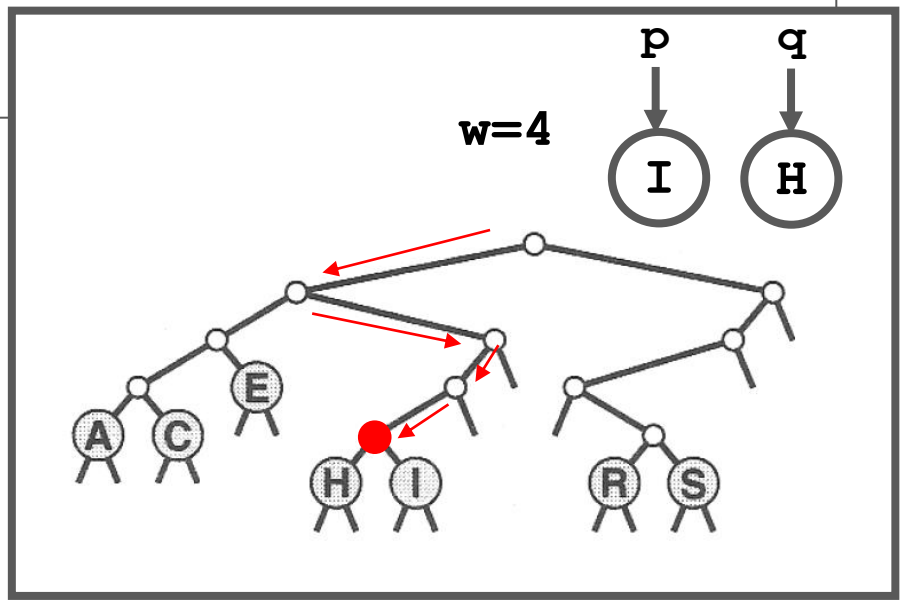
```
link split(link p, link q, int w)
  { link t = NEW(NULLitem, z, z, 2);
    switch(digit(key(p->item), w)*2 + digit(key(q->item), w))
      {
        case 0: t->l = split(p, q, w+1); break;
        case 1: t->l = p; t->r = q; break;
        case 2: t->r = p; t->l = q; break;
        case 3: t->r = split(p, q, w+1); break;
      }
    return t;
  }
```

新しく節点を作る(下図の赤い節点)

I=01001の挿入

I=01001
H=01000

4ビット目が10なので，case 2を実行



トライ

- 性質15.2
  - トライの構造はキーの挿入順序には依存せず，与えられた相異なるキーの場合に対して一意に定まる
- 性質15.3
  - N個のランダムな(相異なる)ビット列から作られたトライでは，1回の探索で平均約lgN回のビット比較を行う．最悪の場合のビット比較回数は，探索キーのビット長で抑えられる