アルゴリズムとデータ構造
第5回: データ構造 (1)
探索問題に対応するデータ構造

担当: 上原隆平 (uehara)

2015/04/17

アルゴリズムとデータ構造

- アルゴリズム: 問題を解く手順を記述
- データ構造:
  - データや計算の途中結果を蓄える形式
  - 計算の効率に大きく影響を与える
  - 例: 配列、連結リスト、スタック、キュー、優先順位付きキュー、木構造

今回と次回で探索問題を例に説明

- 配列
- 一方向連結リスト
  - 要素の追加、挿入、削除

配列と連結リスト

配列: アクセスが容易

- 任意の場所に蓄えられたデータに定数時間でアクセスできる(ランダムアクセス性)
  - cf. 先頭からしかアクセスできないデータ構造
    ➔ i番目の要素のアクセスにはci時間かかる
    e.g., 連結リスト
- インデックス順にアクセスするのも容易(シーケンシャルアクセス性)
  - cf. データがインデックス順ではない順序で蓄えられるデータ構造
    e.g., ランダム木

連結リスト (linked list)

- 前後の要素の場所を明示的に指定
- レコードの連なり

データ部

ポインタ部

  - データを蓄える部分: データ部
  - 前後を指す部分: ポインタ部
- バリエーション
  - 1方向連結リスト
  - 双方向連結リスト
  - 木構造も表現可能

一方向連結リスト

- レコードの連なり
  - データ部: データを蓄える
  - ポインタ部: 次のレコードを指すポインタを蓄える

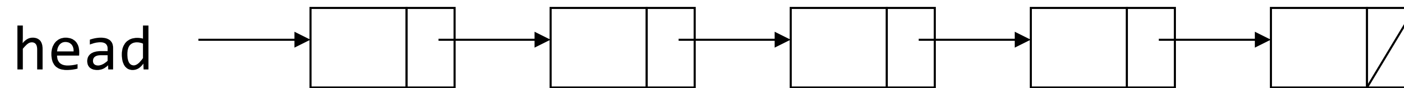
```
typedef struct{
  int data;
  struct list_t *next;
} list_t;
list_t *new_r;
new_r =
 (list_t *)
   malloc(sizeof(list_t));
```

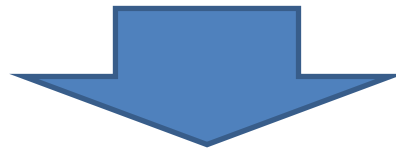
データ部

ポインタ部

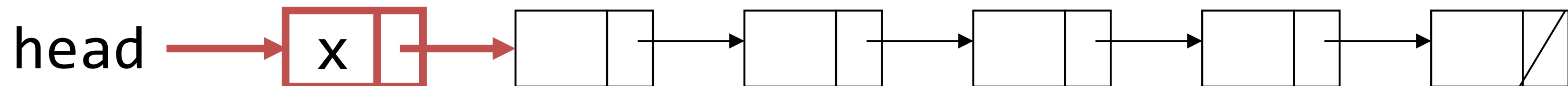
例: 多数のデータを一方向連結リストに蓄える

- 基本:
  - レコードrを作る
  - rのデータ部にデータxを入れる
  - rをリストに連結する





新たなレコードr

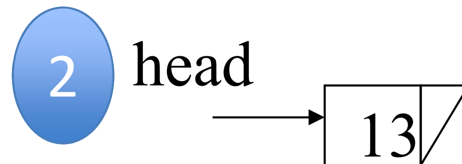


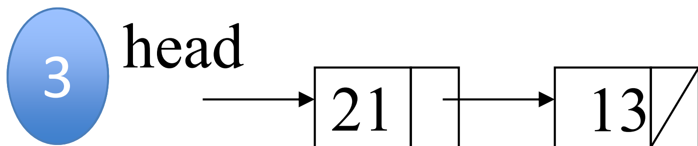
- 連結先: 先頭か末尾

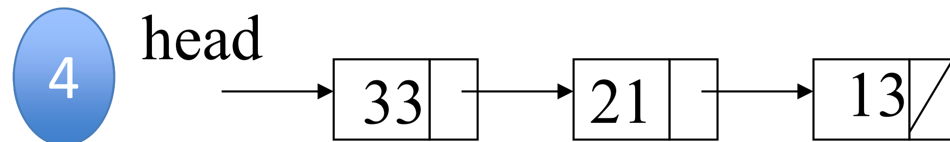
一方向連結リストの先頭に新しいレコードを連結するプログラム

```
list_t *head, *new_r;
int x;
head = NULL;
while(/*入力データがある*/){
    new_r = (list_t *)
        malloc(sizeof(list_t));
    new_r->data = x;
    new_r->next = head; head = new_r;
}
```

新しいレコードは先頭に→
入力順と逆順に蓄えられる







一方向連結リストの末尾に新しいレコードを連結するプログラム

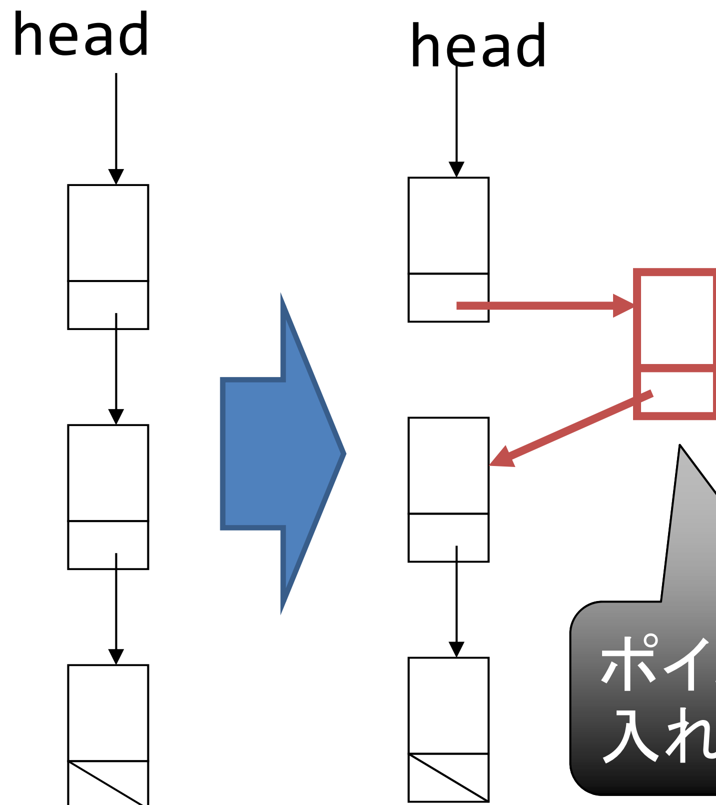
```
list_t *head, *new_r, *tail;
int x = /*some value*/;
new_r =(list_t *)
  malloc(sizeof(list_t));
new_r->data = x;
new_r->next = NULL; head = new_r; tail = new_r;
while(/*蓄えるデータがある*/){
    x=/* next data */;
    new_r =(list_t *)
       malloc(sizeof(list_t));
    new_r->data = x;
    tail->next = new_r;
    new_r->next = NULL; tail = new_r;
}
```

末尾のレコードを指すポインタ

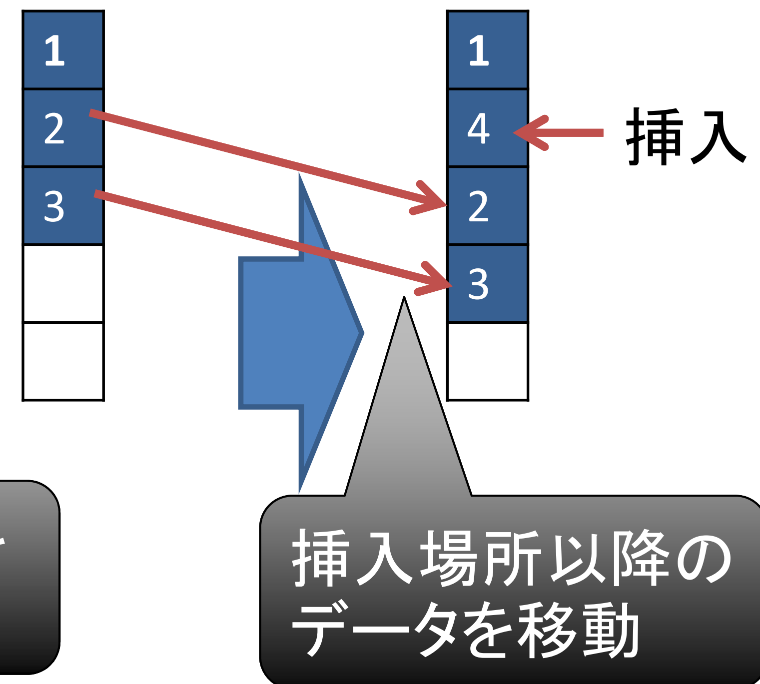
末尾を新しいレコードへのポインタに

連結リストの利点 (配列と比較): データの挿入が容易

連結リスト
- データの移動なし



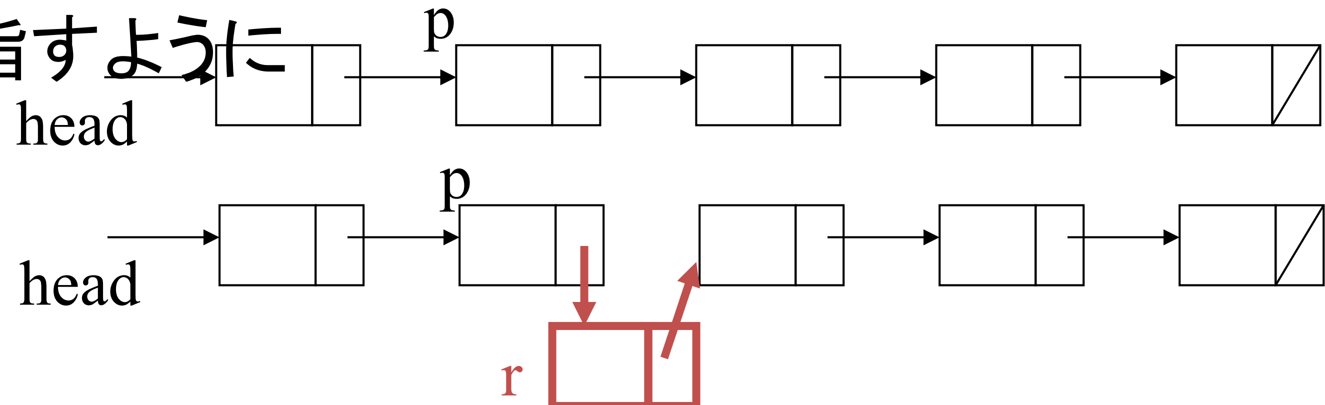
配列
- (多数の) データを移動する必要がある



一方向連結リスト: データの挿入

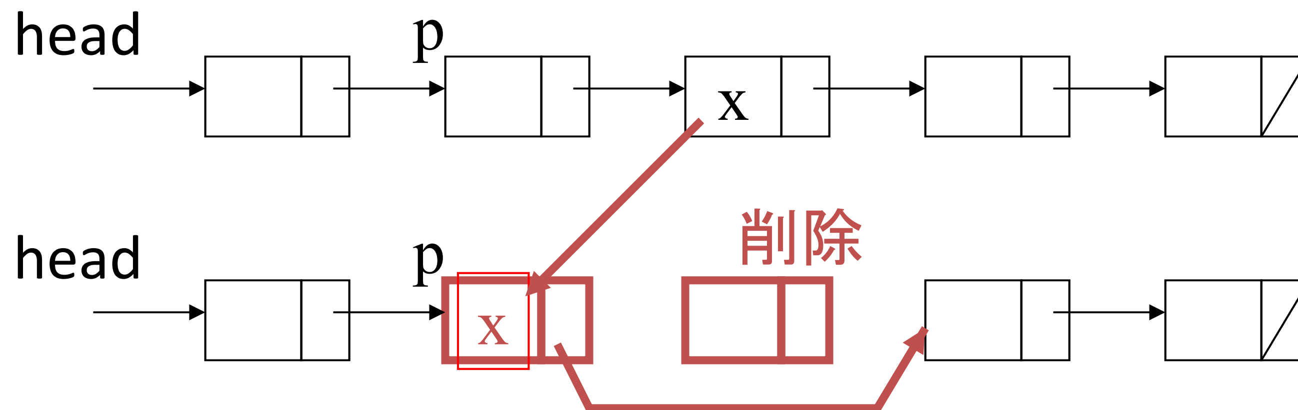
- xをデータ部に含むレコードrを作る
- rのポインタ部にポインタpで示されるレコードのnextポインタの値p->nextを代入
- pのポインタがxを含むレコードを指すようにセット

```
new_r = (list_t *)
  malloc(
   sizeof(list_t)
  );
new_r->data = x;
new_r->next = p->next;
p->next = new_r
```



一方向連結リスト:
データの削除

- ポインタpが指すレコードの削除
  - p->data = p->next->data
  - p->next = p->next->next



pが指すレコードを削除する代わりに，pの次のレコードを削除
（削除する前に，pのデータを移しておく）

- 逐次探索法
- mブロック法
- 2重mブロック法
- 2分探索法

連結リストの応用: それぞれの探索法に対応するデータ構造

逐次探索法に対応するデータ構造

- 先頭からデータxに等しいデータが含まれているかどうか探索
  - 含まれている ➔ 含んでいるレコードのアドレス
  - 含まれていない ➔ NULL

```
typedef struct{
  double data;
  struct list_t *next;
} list_t;
p = head;
while(p != NULL && p->data != x)
   p = p->next;
return p;
```

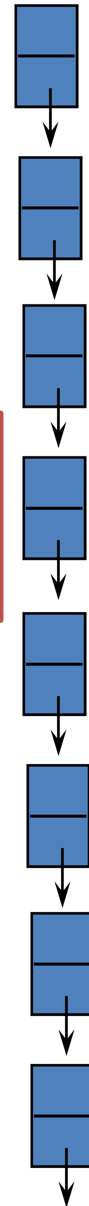
満たされている?

YES

脱出時はp==NULLまたは p->data == x

これでいいの？

YES



- 逐次探索法

- mブロック法
- 2重mブロック法
- 2分探索法

それぞれの探索法に対応するデータ構造

mブロック法

mブロック法の考え方

(0)ソートされた配列全体をm個のブロックB_0, B_1, ... , B_{m-1}に分割.

(1)各ブロックの最大値と比較することにより，質問xを含みうるブロックB_jを求める.

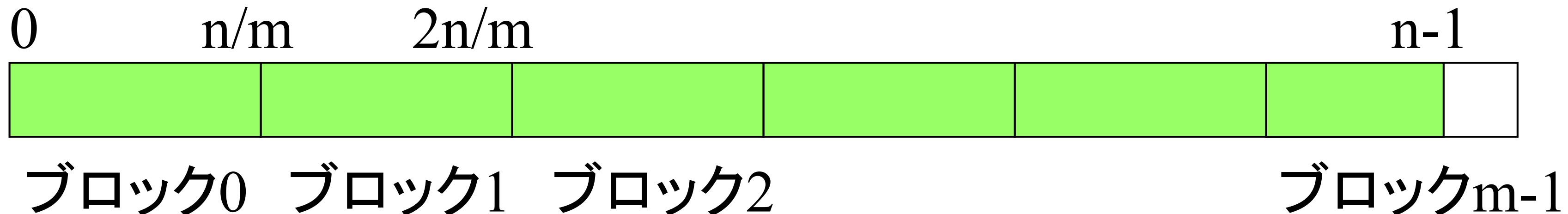
(2)ブロックB_jの中を逐次探索する.

mブロック法

mブロック法の考え方

(0)ソートされた配列全体をm個のブロックB_0, B_1, ... , B_{m-1}に分割.
(1)各ブロックの最大値と比較することにより，質問xを含みうるブロックB_jを求める.
(2)ブロックB_jの中を逐次探索する.

最も簡単な実現方法は，各ブロックの長さを同じにすること.
ただし，最後のブロックだけは長さが異なる.



・ブロック長を k とすると k = ceil(n/m)
・ブロック B_j は配列の jk 番目から (j+1)k-1番目: $B_j = [jk, (j+1)k-1]$

mブロック法

mブロック法の考え方

(0)ソートされた配列全体をm個のブロックB_0, B_1, … , B_{m-1}に分割.
(1)各ブロックの最大値と比較することにより，質問xを含みうる
ブロックB_jを求める.
(2)ブロックB_jの中を逐次探索する.

```
j=0;
while(j<=m-2)
  if x<=s[(j+1)*k-1] then ループから出る
  else j=j+1;
```

途中でループを出たときはそのときのjの値がブロックを指す。
そうでないときは最後のブロックが対象となる。

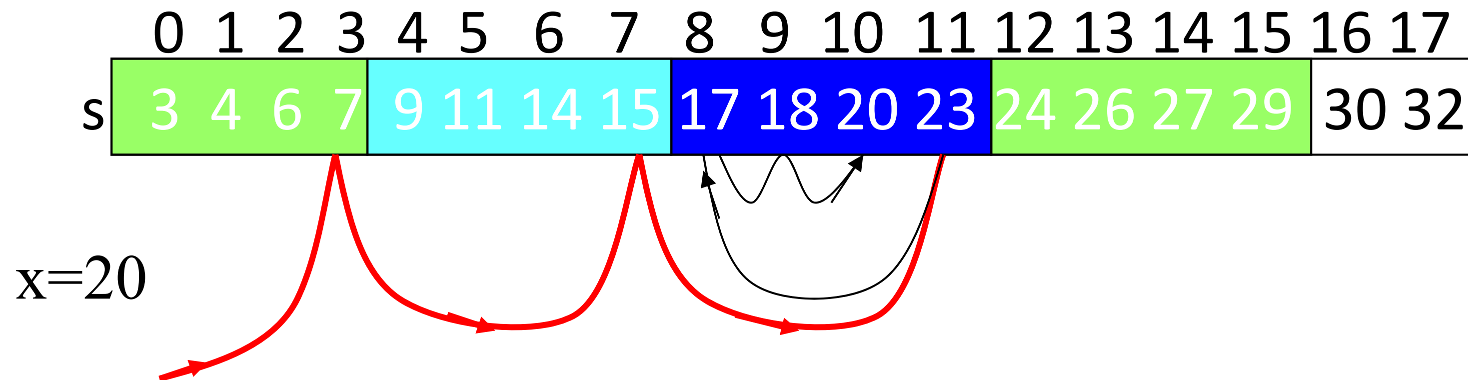
mブロック法

mブロック法の考え方

(0)ソートされた配列全体をm個のブロックB_0, B_1, … , B_{m-1}に分割.
(1)各ブロックの最大値と比較することにより，質問xを含みうるブロックB_jを求める.
(2)ブロックB_jの中を逐次探索する.

```
i=j*k; t = min{ (j+1)*k-1, n-1 };
while( i < t )
  if x<=s[i] then ループから出る;
  else i=i+1;  //ブロック内の次の要素へ
if x = s[i] then iを返して終了;
else  -1を返して終了;
```

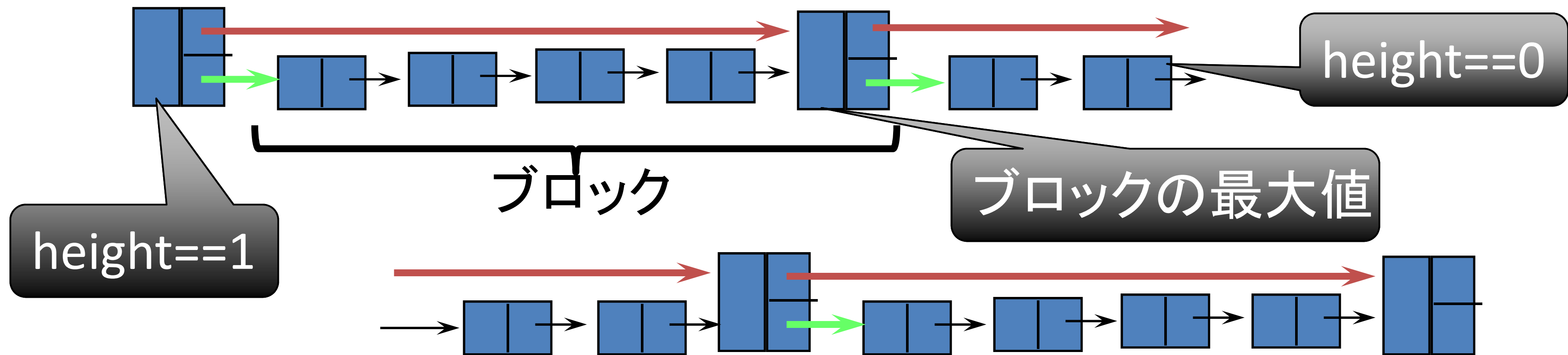
探索例と計算時間



- 比較回数 ≦ ブロック数＋ブロック
  ＝ $m + n/m$
- $m + n/m$ を最小にする m の値は？
  - $f(m) = m + n/m$ とおいて，m に関して偏微分する
  - $f'(m) = 1 - n/m^2 = 0 \rightarrow m = \sqrt{n}$
  - $m = \sqrt{n}$ のとき，比較回数 ≦ $\sqrt{n} + n/\sqrt{n} = 2\sqrt{n}$
  - よって，時間複雑度は $O(\sqrt{n})$

mブロック法に対応するデータ構造

- レコードに高さの概念を導入



最初は上のpointerをたどって，どのブロックが質問要素を含み得るかを調べる．次に下のpointerをたどってブロック内を逐次調べる．

```
typedef struct{
  int data; int height; struct mlist_t **next;
}mlist_t;
```

next[0]: block内の次へ, next[1]: 次のblockへ

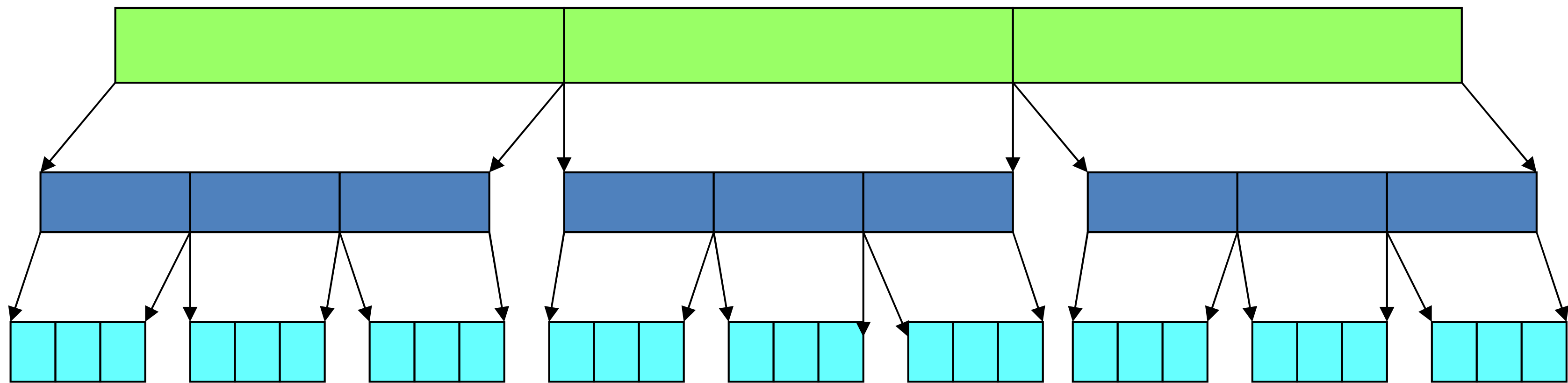
- 逐次探索法
- mブロック法
- 2重mブロック法
- 2分探索法

それぞれの探索法に対応するデータ構造

2重mブロック法

mブロック法では、ブロック内の探索に逐次探索を利用

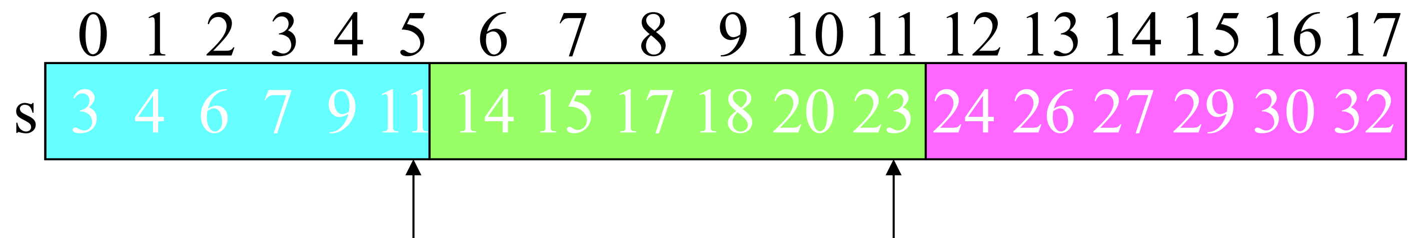
→ ブロック内も同じ方法で探索

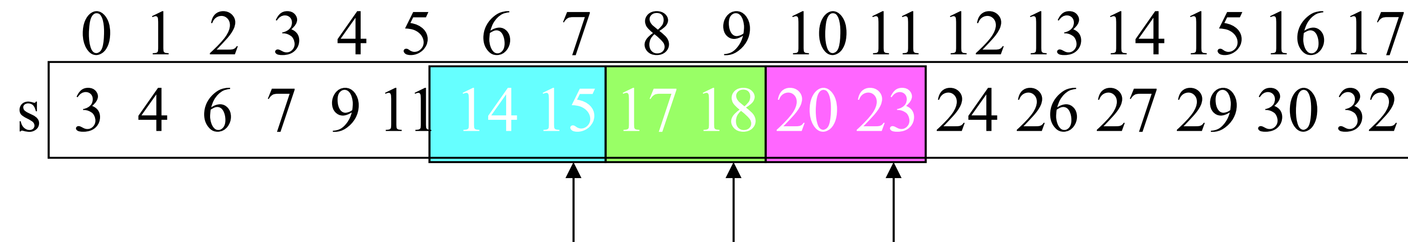


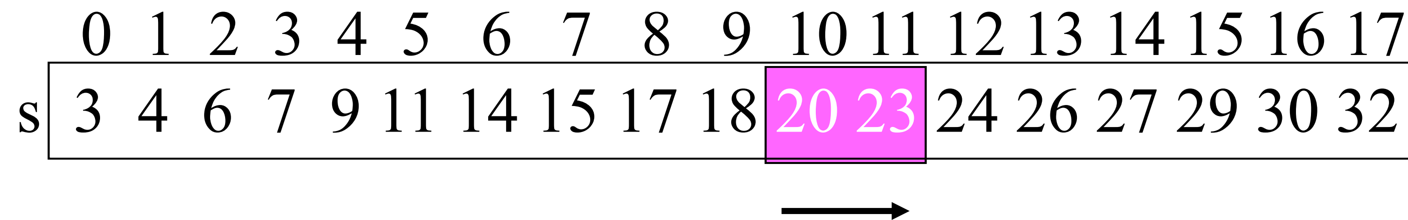
二重mブロック法の原理

探索区間をm個のブロックに分割し、xを含むブロックに対して、同じ探索を再帰的に適用。これを、ブロック長が定数N以下になるまで行なう。

探索例: 20を探す (x=20)
ブロック数を3とした場合



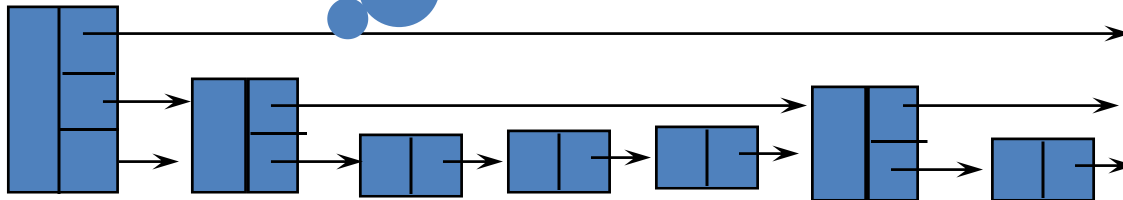




2重mブ

```
p = head;
for(i = MAX_H - 1; i >= 0; i--)
   while( (p->next[i])->data < x)
      p = p->next[i];
p = p->next[0];
return p->data==x?p:NULL;
```

- 各レ
  - cf. mブ
- 順次高さをT て探索



```
typedef struct{
  int data;
  int height;
  struct mlist_t **next;
} mlist_t;
```

next[i]は高さiのポインタ
(0≤i≤height<MAX_H)

- 逐次探索法
- mブロック法
- 2重mブロック法

- 2分探索法

それぞれの探索法に対応するデータ構造

2分探索法

- ソートされた探索空間を二つに分けて探索

5を見つける

33より小さい

33より大きい

78を見つける

2	5	6	19	33	54	67	72	78

2	5	6	19

54	67	72	78

6より小さい

72より大きい

2	5

78

発見!

発見!

  – 分けるポイントはだいたい真ん中にすると良い

2分探索法

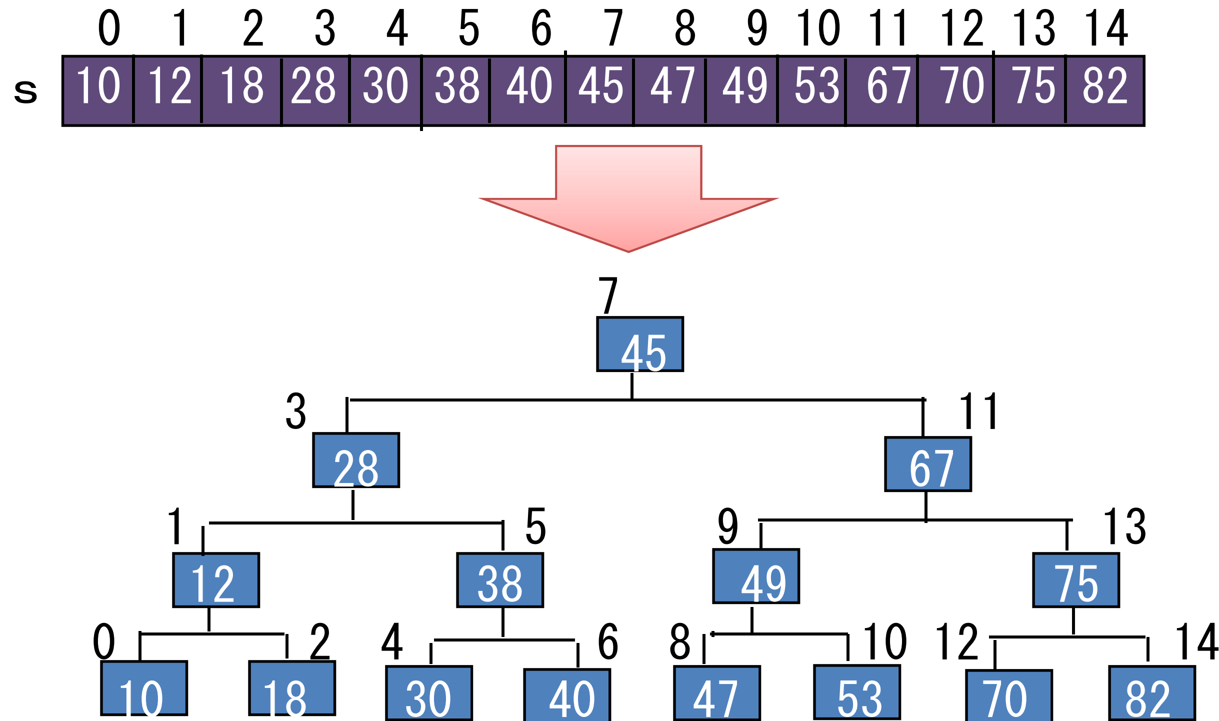
2	5	6	19	33	54	67	72	78

2	5	6	19

2	5

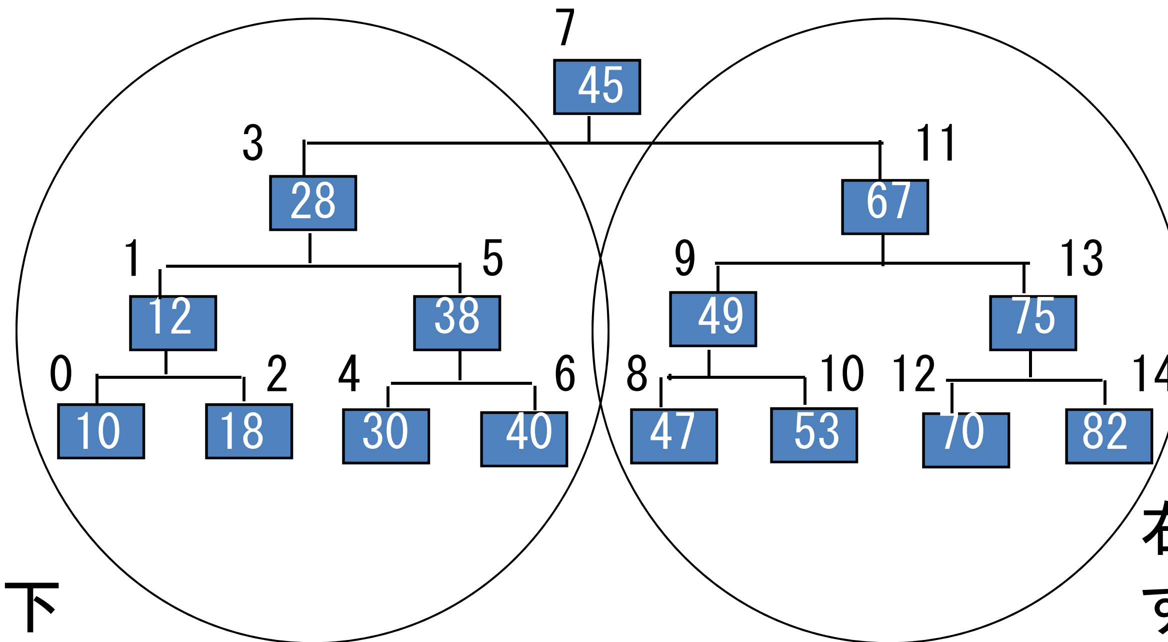
- 探索範囲[left, right]の中央の値 s[mid] と探したい値を比較
  - x > 中央の値 ➔ 探索を右半分に限定できる
    left = mid+1; rightは同じ
  - x < 中央の値 ➔ 探索を左半分に限定できる
    leftは同じ, right = mid-1
  - x = 中央の値 ➔ 見つかった
- 以上を探索範囲がなくなるまで繰り返す

2分探索法に対応する データ構造: 2分探索木



- データサイズnが固定されていれば，予め中央の位置が計算できる

2分探索木の性質



左部分木
すべて45以下

右部分木
すべて45以上

- あるノードnについて、
  - 右の子部分木に現れる値はnの値よりも大きい
  - 左の子部分木に現れる値はnの値よりも小さい

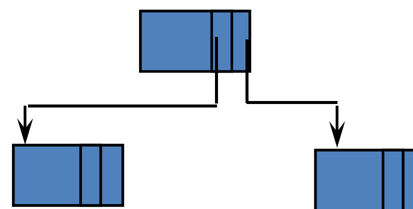
2分探索木における探索

```
BSTnode *root, *v;
x=/*some value*/;
v = root;
while( v ){
  if(v->data == x)
    break;
  if(v->data > x)
    v = v->lson;
  else
    v = v->rson;
}
return v;
```

小さければ左

大きければ右

```
typedef struct{
  int data;
  struct BSTnode
    *lson, *rson;
}BSTnode;
```



各レコードで
左の子へのポインタと
右の子へのポインタを
もつ

ミニ演習

- $pn^2 + qn + r \in O(n^2)$ を証明せよ
  - p = 21, q = 4, r = 2014 の場合
  - p, q, r が全て正の場合
  - そうとは限らない場合

- $O(\log_2 n) = O(\log_{10} n)$ を証明せよ
  (どちらも集合であるとみなして解くこと)