大阪電気通信大学 情報通信工学部 光システム工学科 2年次配当科目

コンピュータアルゴリズム

探索アルゴリズム (1)

第7講: 平成20年11月21日 (金) 4限 E252教室

中村 嘉隆(なかむら よしたか)
奈良先端科学技術大学院大学 助教
y-nakamr@is.naist.jp
http://narayama.naist.jp/~y-nakamr/

第 4 講の復習

- 整列アルゴリズム
  - ソーティング，並べ替え
  - $O(n^2)$ のアルゴリズム
    - 選択ソート
      - 最小値を探して前から並べる
    - バブルソート
      - 隣の要素の大小関係で交換していく
    - 挿入法
      - 前から順番に入るべき位置に入れていく

第 5，6 講の復習

- 整列アルゴリズム
  - $O(n \log n)$ のアルゴリズム
    - マージソート
      - 2 つ，4 つ，8 つと整列する列を併合(マージ)していく
    - クイックソート
      - 基準値(ピボット)を選んで，それより小さい数値の列と大きい数値の列に分けていく
      - 分割統治法

本日の講義内容

- 探索アルゴリズム
  - 探索するデータ構造
    - レコードの列 → 表
  - 線形探索(linear search)
  - 2 分探索(binary search)
  - 2 分探索木(binary search tree)

探索(サーチング)問題とは

- サーチング: Searching，探索
  - n 個のレコード列から，キーの値を指定して，それと等しいキーを持つレコードを選ぶ処理
- レコード(record)とキー(key)
  - レコードとは，ひとかたまりのデータ
  - キーとは，レコードの中にある 1 つのフィールド(要素)
  - 例:成績{学籍番号，名前，出席点，試験点}
    - レコードは 1 人分のデータ(例:{5433，中村，30，55})
    - キーは，要素のどれか(例えば，学籍番号)
  - ここでは簡単のため同じキーを持つレコードは複数存在しないとする

探索するレコードの表とサイズ

- 探索はある列 (表) に対して行う
  - その表を作るのに必要な計算量も考慮が必要
  - 問題のサイズ ＝レコード数

番号	名前	点数
1	たろう	76
2	はな	82
3	こん	74

レコード

問題の サイズ n

キー

- 表の分類
  - 静的な表
    - 一度表を作ると二度と作り替えない
    - ➡ 探索さえ早くすればよい
  - 動的な表
    - 表を作ったあとでも，レコードの追加，削除がある
    - ➡ レコードの追加，削除の手間も考慮

表の例

- 静的な表の例
  - 学食のメニュー
    - 新学期に作成すると 1 年(数年？）はほとんど変わらない
    - レコードの例：{メニュー名，カロリー，値段}
- 動的な表の例
  - 電話帳
    - 新しい友達ができると追加
    - 音信不通になると削除
    - レコードの例：{名前，電話番号，メールアドレス}

線形探索

- 線形探索：linear search，sequential search，逐次探索，順探索
  - アルゴリズム
    - 配列，またはリストに並べられたデータを一つ一つ順に端から調べる

5 回優勝した横綱は？(キー：優勝回数)

143kg の横綱は？(キー：体重)

朝青龍	武蔵丸	若乃花	貴乃花	曙	旭富士	大乃国
139kg	235kg	134kg	159kg	232kg	143kg	203kg
15回	12回	5回	22回	11回	4回	2回
[1]	[2]	[3]	[4]	[5]	[6]	[7]

線形探索の計算量

- 探索のみの計算量を考える

```
linear_search (keytype target)
{
        pos ← 1;
        while (pos ≦ n) and (target ≠ table[pos].key) {
                pos ← pos + 1;
        }
        if (pos ≦ n) {
                return pos;
        } else {
                return -1; /* 見つからなかった */
        }
}
```

探索するキーの値

列の最後になるまで

pos 番目のレコードの要素が target と違うなら pos を 1 進める

見つかった位置を返す

線形探索の計算量

- 探索のみの計算量を考える

```
linear_search (keytype target)
{
        pos ← 1;
        while (pos ≦ n) and (target ≠ table[pos].key) {
                pos ← pos + 1;
        }
        if (pos ≦ n) {
                return pos;
        } else {
                return -1; /* 見つからなかった */
        }
}
```

平均で $n/2$ 回，最大で n 回まわる

繰り返し

基本操作

$O(n)$

線形探索のデータ構造

- 前から辿るだけ
  - 配列なら，添え字 1 の要素からキーを調べる
  - リストなら，先頭からキーを調べる

➡ どちらでも良いように思える

- 表の作りやすさを考える
  - レコードの追加があった場合にどうするか
    - 追加しやすい場所に追加すればよい(順番はどうでも構わない)
    - 配列もリストも $O(1)$ で追加可能
  - レコードの削除があった場合にどうするか
    - 配列はその要素以降を前に 1 つずつ詰める必要がある：$O(n)$
    - リストは $O(1)$ で削除可能
    - でも結局，どちらも削除する要素を探索するのに $O(n)$ かかる

➡ 配列 $O(n)+O(n) = O(n)$，リスト $O(n)+O(1)=O(n)$ ➡ 同じ

線形探索の計算量のまとめ

- 探索の計算量
  - $O(n)$
- 表へのレコードの追加，削除の計算量
  - 追加 $O(1)$
  - 削除 $O(n)$
- データ構造は配列を使っても，リストを使ってもあまり変わらない
  - しかし，リストの方が望ましい(後述の理由でもそれは言える)

線形探索の高速化：番兵の利用

- **while** ループを回るたびに **pos** がサイズ n を超えていないかチェックしている

  平均で $n/2$ 回，最大で n 回チェック

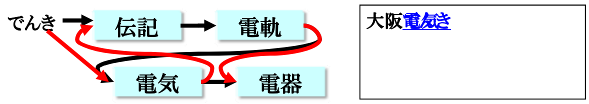
- 解決法：
  - 最後の次（n+1 番目の要素）に，探索するキーと同じ値を持つレコードを入れておく
  - 列の最後まで来ると必ずキーに一致する
  - キーに一致するレコードが見つかったとき，その位置が n 番目以下か n+1 番目かチェック

    最後に 1 回だけチェック

  - n+1 番目ならキーは見つからなかったとする
- **while** ループの度にチェックする必要はなくなる
- こういうものを番兵と呼ぶ

自己再構成リスト

- 線形探索は，列（表）の最初の方に目的のレコードがあれば性能はよい
- 自己再構成リスト
  - 自分で順番を再構成するリスト
  - 探索される頻度の高いレコードは前につなぎ変える
  - 例：漢字変換プログラム
    - 最近使われた変換候補は前につなぎ直す



線形探索のまとめ

- 入力
  - レコードの列（並び方は自由）
- アルゴリズム
  - 前から順番にキーを調べていく
- 計算量
  - 探索 $O(n)$，表への追加 $O(1)$，削除 $O(n)$
- その他
  - 番兵による高速化
  - 応用例：自己再構成リスト

2 分探索

- 2 分探索：binary search
- もっと賢く探索したい
  - 線形探索はキーに合うか否かの判断だけ
  - 普通はキーには意味があって，それらには大小関係があることが多い（ほとんど）
  - 値の大小比較もすればもっと効率良くできるかも
- 入力をキーであらかじめ整列された列（表）とする
  - 整列は前に勉強した
  - キーの大小判定することで，目的のキーが列（表）の前にあるか後ろにあるか判断できる

身近な 2 分探索

- 辞書を引く（キーは見出し語）
  - 辞書は見出し語が五十音順に並んでいる
    - このような文字列の並ぶ順を辞書式順という
  - とりあえず辞書の半分ぐらいの場所（ページ）を開く
  - その見出し語が目的の語より前（後）なら，辞書の前（後）の部分のまた半分ぐらいのページを開く
  - 繰り返す
  - 辞書が 1000 ページなら，範囲が 500 ページ，250 ページ，125 ページ，63 ページ，32 ページ，16 ページ，8 ページ，4 ページ，2 ページ，目的のページと半々に絞られていく
  - 最悪で 10 ページ見るだけで目的の語に到達できる
  - ちなみに線形探索なら最悪で前から 1000 ページ分調べないといけない

2 分探索のアルゴリズム

1. 入力は長さ n（添え字は 1～n）のキーであらかじめ整列された配列 A とする
2. 目的のキーを **target**，調べる範囲は最初 **lo** ← **1** から **hi** ← **n** までである
3. **mid** ← **(lo+hi)/2** とする
4. **A[mid]** のキーと **target** を比較
5. **A[mid].key = target** なら **mid** が目的のレコードの位置
6. **A[mid].key < target** なら **lo** ← **mid + 1** として 3. に戻る
7. **A[mid].key > target** なら **hi** ← **mid - 1** として 3. に戻る
8. **lo > hi** になると目的のレコードが見つからなかった

2 分探索の概念図

- キー 21 を持つ動物を探したい
  - lo = 1, hi = 16, mid = 8

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
キー	5	8	13	19	21	26	33	34	36	40	45	55	58	69	74	81
	虎	牛	馬	猫	鶏	犬	鷹	鼠	狸	兎	羊	豚	猿	狐	人	魚

  - lo = 1, hi = 7, mid = 4

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
5 虎	8 牛	13 馬	19 猫	21 鶏	26 犬	33 鷹	34 鼠	36 狸	40 兎	45 羊	55 豚	58 猿	69 狐	74 人	81 魚

  - lo = 5, hi = 7, mid = 6

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
5 虎	8 牛	13 馬	19 猫	21 鶏	26 犬	33 鷹	34 鼠	36 狸	40 兎	45 羊	55 豚	58 猿	69 狐	74 人	81 魚

  - lo = 5, hi = 5, mid = 5　見つかった！！

2 分探索の計算量

```
binary_search (keytype target)
{
        lo ← 1; hi ← n;
        while (lo ≦ hi) {
                mid ← (lo + hi) / 2;
                if( A[mid].key = target) {
                        return mid;
                } else if( A[mid].key < target) {
                        hi ← mid – 1;
                } else {
                        lo ← mid + 1;
                }
        }
        return -1; /* 見つからなかった */
}
```

探索するキーの値

列の範囲を表す lo と hi の位置が矛盾しない間

A[mid].key と target の大小関係で表の範囲を絞っていく

2 分探索の計算量

```
binary_search (keytype target)
{
        lo ← 1; hi ← n;
        while (lo ≦ hi) {
                mid ← (lo + hi) / 2;
                if( A[mid].key = target) {
                        return mid;
                } else if( A[mid].key < target) {
                        hi ← mid – 1;
                } else {
                        lo ← mid + 1;
                }
        }
        return -1; /* 見つからなかった */
}
```

範囲が必ず半分になっていく $\log_2 n$ 回まわる

$O(\log n)$

繰り返し

基本操作

2 分探索のデータ構造

- 配列型でないといけない
  - 配列型は添え字でちょうど真ん中の位置のレコードにアクセスできる
  - リストはランダムアクセスできない（前から辿るのみ）
- レコードの追加，削除はどうなる？
  - 表の中のレコードはキーの順に並んでないといけないので，線形探索のときと違い，どこに追加しても良いわけではない
  - 追加のときもどこに入るか調べる必要がある（探索を使えばよい）

2 分探索のデータ構造：追加と削除

- レコードの追加
  - 追加する位置の探索
    - これは 2 分探索すれば $O(\log n)$ で求まる
    - プログラムで見つからなかった場合に −1 を返すのではなく，直前の位置を返すようにすればよい
  - 配列への要素の挿入
    - 追加位置から後ろのレコードは 1 つずつ後ろにずらす必要がある $O(n)$
  - $O(\log n) + O(n) = O(n)$
- レコードの削除
  - 削除する位置の探索 $O(\log n)$
  - 配列の要素の削除 $O(n)$
  - $O(\log n) + O(n) = O(n)$

2 分探索の計算量のまとめ

- 探索の計算量
  - $O(\log n)$　線形探索より小さい
- 表へのレコードの追加，削除の計算量
  - 追加 $O(n)$　線形探索より大きい
  - 削除 $O(n)$
- データ構造は配列を使う
  - ランダムアクセス（列の真ん中の要素へのアクセス）が必要
  - そのためリストは使えない

2 分探索のまとめ

- 入力
  - 探索するキーで整列されたレコードの列
- アイデア
  - 探索するキーと，列の中央の要素のキーの大小関係で探索範囲を半減させる
- 計算量
  - 探索 $O(\log n)$，表への追加 $O(n)$，削除 $O(n)$
- その他
  - 線形探索に比べて，探索の計算量は小さいが，追加の計算量が多い
  - 表への追加が多い(動的な)場合はおすすめできない
  - 静的な表への探索に向いている

2 分探索木

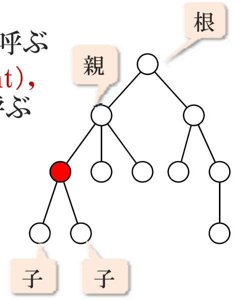
- 2 分探索木：binary search tree
- いままでの 2つの探索法のまとめ

計算量	探索	追加	削除
線形探索	$O(n)$	$O(1)$	$O(n)$
2 分探索	$O(\log n)$	$O(n)$	$O(n)$

  - 入力データ構造が単純な一直線の列であるこれらの探索法では，探索・追加・削除の全てにおいて $O(\log n)$ を実現することは不可能
- レコードのデータ列(表)を木構造にすることによって，探索・追加・削除の全てにおいて平均で $O(\log n)$ を実現するのが 2 分探索木

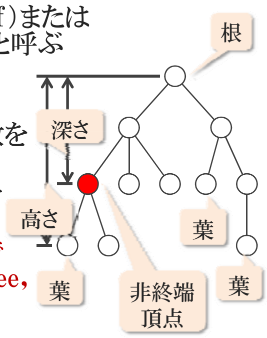
木構造(Tree)の復習

- 頂点(Vertex，Node(節点))と 枝(Branch Edge，Arc(辺))から構成される
- 一番上の頂点を根(Root)と呼ぶ
- 枝の上側の頂点を親(Parent)，下側の頂点を子(Child)と呼ぶ
  - ある頂点から見て親，親の親などをまとめて祖先(Ancestor)と呼ぶ
  - ある頂点から見て子，子の子などをまとめて子孫(Descendant)と呼ぶ



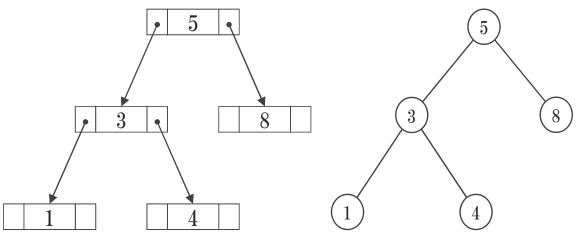
木構造(Tree)の復習

- 子を持たない頂点を葉(Leaf)または終端頂点(Terminal Node)と呼ぶ
- 子を持つ頂点を非終端頂点(Nonterminal Node)と呼ぶ
- 根からある頂点までの枝の数を深さ(Depth)と呼ぶ
- 根から最も遠い頂点の深さを木の高さ(Height)と呼ぶ
- 各頂点の子の数が高々 2 である木を 2 分木(Binary Tree，2 進木)と呼ぶ



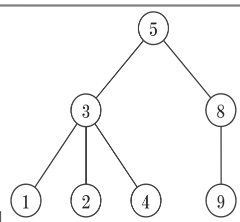
木(Tree)の実現

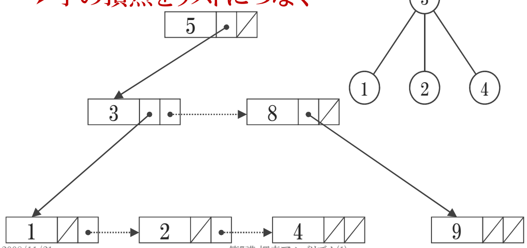
- 2 分木の場合
  - 2 つの子を指すポインタとデータをいれる箱で実現



木(Tree)の実現

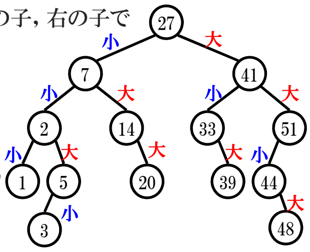
- 一般の木
  - 子の数に制限がない
  - 子の頂点をリストにつなぐ





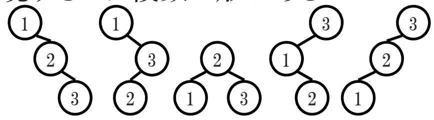
2 分探索木とは

- 以下の特徴を持つ木構造
  - 各節点は最大で 2 個の子を持つ
    - その 2 個の子は，左の子，右の子である
  - 左の子（子孫）は，親より小さな値を持つ
  - 右の子（子孫）は，親より大きな値を持つ

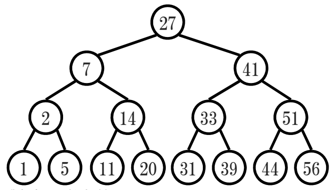


2 分探索木の形

- 同じ列を表現するのに複数の形がある
  - 例：{1,2,3}



- 完全 2 分木
  - 葉以外の全ての節点が 2 つずつ子を持つ

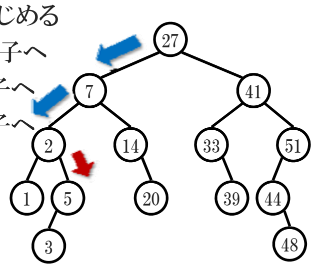


2 分探索木の探索アルゴリズム

1. 目的のキーを `target`，現在のノードを `root`（根）とする
2. 現在のノード `c` のキーと `target` を比較
3. `c.key = target` なら `c` が目的のレコード，探索終了
4. `target < c.key` のとき，
   - 左の子（`c.left`）があるなら，`c ← c.left`（左のノードを辿る）として 2. に戻る
   - 左の子がないなら，見つからなかったとして探索終了
5. `c.key < target` のとき，
   - 右の子（`c.right`）があるなら，`c ← c.right`（右のノードを辿る）として 2. に戻る
   - 右の子がないなら，見つからなかったとして探索終了

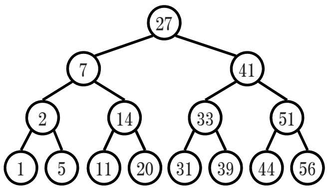
2 分探索木の概念図

- キー 5 を持つノードを探したい
  - 根（キー：27）からはじめる
  - 5 < 27 なので，左の子へ
  - 5 < 7 なので，左の子へ
  - 2 < 5 なので，右の子へ
  - 5 = 5 なので，終了



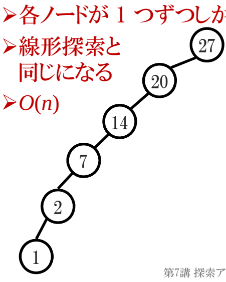
2 分探索木の計算量

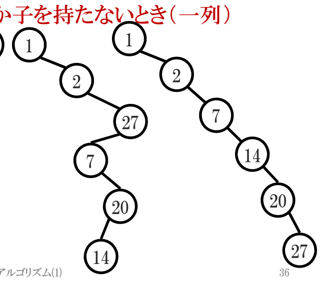
- 探索の計算量
  - 最良の場合
    - 完全 2 分木のとき
      - ノード数 n (=2m) に対して木の高さは $\log n$ (=m)
      - 最大でも $\log n$ 回木を辿れば，目的のノードに辿り着く
    - $O(\log n)$
  - 平均的な場合
    - このときも最良の場合の 1.39 倍しか悪化しない（証明略）
    - $O(1.39 \log n)$ $= O(\log n)$



2 分探索木の計算量

- 探索の計算量
  - 最悪の場合
    - 各ノードが 1 つずつしか子を持たないとき（一列）
    - 線形探索と同じになる
    - $O(n)$

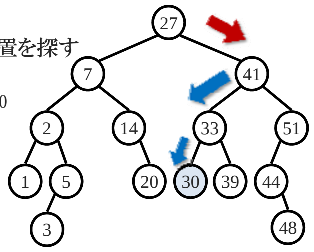




2 分探索木のデータ構造

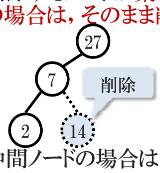
- リスト型で木構造を作る
- レコードの追加，削除はどうなる？
  - 追加
    - 探索して入るべき位置を探す

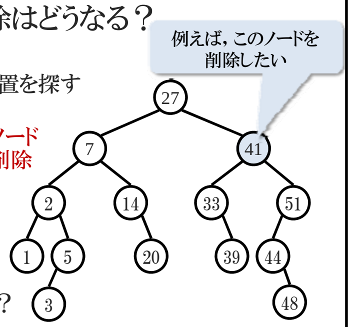
      例：キー 30 のデータ
      - 27 → 41 → 33 → 30
    - 探索 $O(\log n)$
    - 挿入は $O(1)$
    - 全体で $O(\log n) + O(n)$ $= O(\log n)$



2 分探索木のデータ構造

- レコードの追加，削除はどうなる？
  - 削除
    - 探索して入るべき位置を探す
      - 探索 $O(\log n)$
    - 削除するノードが葉ノードの場合は，そのまま削除

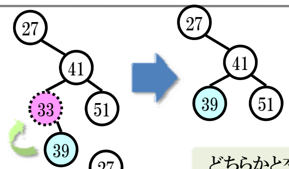




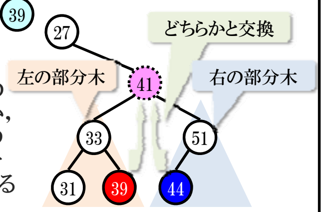
    - 中間ノードの場合は？

2 分探索木からのノードの削除

- 中間ノードの削除
  - 子が 1 つの場合
    - 子を親とつなげる



  - 子が 2 つの場合
    - 左の部分木の最大値のノード(最も右奥の子)か，右の部分木の最小値のノード(最も左奥の子)を持ってきて代わりをさせる



2 分探索木の削除の計算量

- 削除ノードの探索
  - $O(\log n)$
- 削除するノードが葉ノードの場合
  - $O(1)$ で削除可能
- 中間ノードの場合
  - 交換候補を左右どちらかの部分木を辿って見つける → $O(\log n)$
  - 見つかったら交換は $O(1)$ で可能
- 削除全体では，

  $O(\log n)+\{O(\log n)+O(1)\} = O(\log n)$

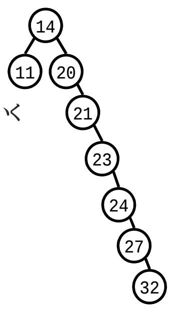
2 分探索木の計算量のまとめ

- 探索の計算量
  - 平均 $O(\log n)$，最悪 $O(n)$
  - 最悪 $O(n)$ なので保証が必要なら使わない方がよい
- 表へのレコードの追加，削除の計算量
  - 追加 $O(\log n)$
  - 削除 $O(\log n)$

  追加削除も小さい計算量で可能
- データ構造はリストを使って木構造にする

2 分探索木の落とし穴

- 木の形が最悪になりやすいことがある
  - 途中でどんどんレコードが追加されるとする(動的)
  - このとき，ある程度整列された順で追加されると，木の形が一直線になっていく
  - 例：{14,11,20} の木に，21，23，24，27，32 のキーの要素が入ってくるとする
  - このような入力は与えやすいので注意
  - そのような入力が予想されるときには 2 分探索木は使わない方がよい



2 分探索木のまとめ

- 入力
  - 左の子孫は小さなキー，右の子孫は大きなキーを持つ 2 分木
- アイデア
  - 各ノードのキーと探索したいキーを大小比較することで，探索範囲を片方の部分木に限定していく
- 計算量
  - 探索 平均 $O(\log n)$，最悪 $O(n)$
  - 表への追加 平均 $O(\log n)$，削除 平均 $O(\log n)$
- その他
  - 最悪で $O(n)$ になるため注意が必要（平均は $O(\log n)$）
  - 整列されたデータを追加していくと木の形が直線的になり，計算量が最悪に近づく

第 7 講のまとめ

- 探索アルゴリズム
  - 線形探索
  - 2 分探索
  - 2 分探索木