情報処理応用 (前期末試験に向けて)

山本昌志*

2007 年 7 月 27 日

概 要

前期末試験に向けて，前期中間試験以降に学習した内容をまとめる．このプリントは試験対策用である．

1 前期中間試験の傾向と対策

試験の範囲は，以下の通り．

- 第 10 回の講義から第 14 回の講義に配布したプリント
- 教科書「C 言語によるプログラミング　応用編」の第 6 章「アルゴリズムとデータ構造」pp.170–220 を試験範囲とする．
- すべてではないが，重要な課題の解答を WEB に載せている．解答を載せている問題は，必ず理解して解けるようになること．

少なくとも，ここにかかれた内容は，すべて理解して試験に臨むこと．丸暗記はダメである．丸暗記は，自分自身の学力の向上にはなっていない．努力して，理解せよ!

2 データ構造

リストとスタック，キュー，ツリーの 4 つのデータ構造を学習した．

2.1 リスト

2.1.1 配列とリストの違い

順序づけられたデータを格納するという点で，配列とリストはよく似ている．しかし，使い方やメモリーへの格納の仕方は，大きく異なるのでその違いを理解しておく必要がある．

例えば，次のような数列があるとする．

2, 63, 43, 12, 24, 77, 5, 23, 18, 37, 81, 57, 29, 62, 49, 30

*独立行政法人　秋田工業高等専門学校　電気情報工学科

この数列には，個々の値とともに，その並び方にも意味がある．ようするに値とその順序の情報を持つ．この順序づけられたデータをコンピュータープログラムで取り扱う場合，配列あるいはリストと呼ばれるデータ構造を使う[1]．先ほどの例では，要素の数が多く，図で表すのが不適切なので，次の数列

$$63, 27, 82, 79, 12$$

を使い，配列とリストの違いを説明する．

配列のモデルとメモリーへの格納の様子は，図 1 のようになる．その特徴は，次の通りである．

- 配列の添字を使って，順序を表している．
- メモリーの連続した領域に，データの順序通りに格納される．

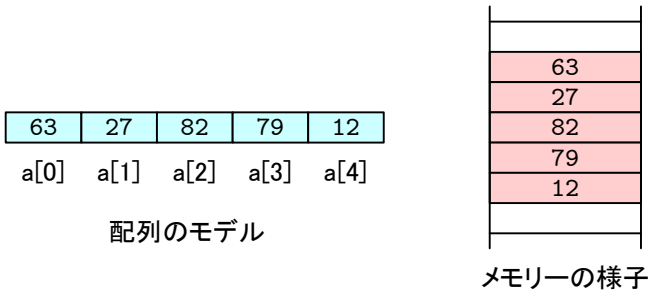

図 1: 配列のモデルとメモリーの様子．

それに対して，リストは図 2 のように表現することができる．その特徴は，次の通りである．

- ポインターを使って，順序を表している．
- メモリーの連続した領域にデータを格納する必要がない．また，データの順序通りに値を格納する必要もない．

[1] これらの他のデータ構造でも取り扱うこともできる

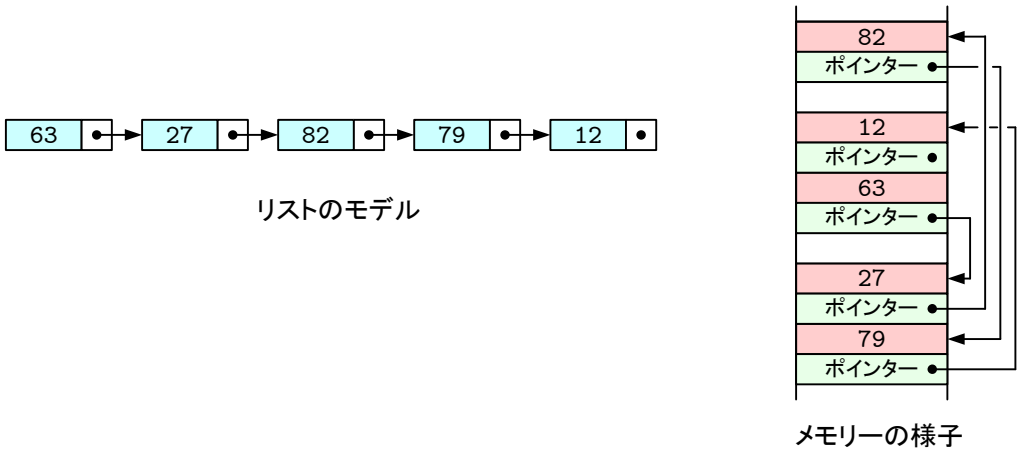

図 2: リストのモデルとメモリーの様子.

2.1.2 リストへのデータの挿入

配列とリストにデータを挿入する場合を考える．先ほど示した数列の 63 の次に 99 を挿入することを考える．

配列の場合，図 3 のようにしてデータを挿入する．後ろの方からひとつずつ，右側に値をコピーする．そして，データを挿入する場所にあった値のコピーが終われば，そこに挿入する値をコピーする．

データ数を N とすると，配列にデータを挿入する場合の処理 (コピー) の回数は，平均して $N/2$ 程度である．もちろん，コピー回数は挿入する場所にも依存するが，ここではあくまで平均を問題とする．したがって，平均の処理の時間はデータ数 N に比例し，それを $O(N)$ と表す[2].

[2]「オーダー N」と読む.

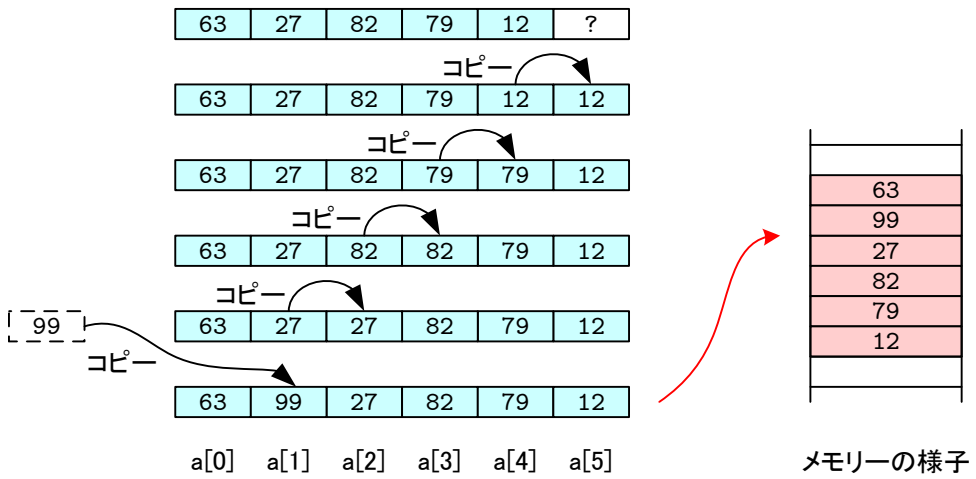

図 3: 配列のデータの挿入の様子.

それに対して，リストの場合は図 4 のようにしてデータを挿入する．データ 63 のポインターを 99 に接続し，データ 99 のポインターを 27 に接続する．この方法だと，データがいくらあっても処理の回数は同じである．$O(1)$ である．

配列とリストではデータを挿入する場合，処理の回数が決定的に異なる．リストの方が処理の回数が少ない．処理の回数が異なるのは，データのメモリーへの格納の仕方による．配列は頑固に，データと同じ並びで連続した領域に，値を格納する．一方，リストはメモリーに適当に格納して，順序を表すポインターを使う．

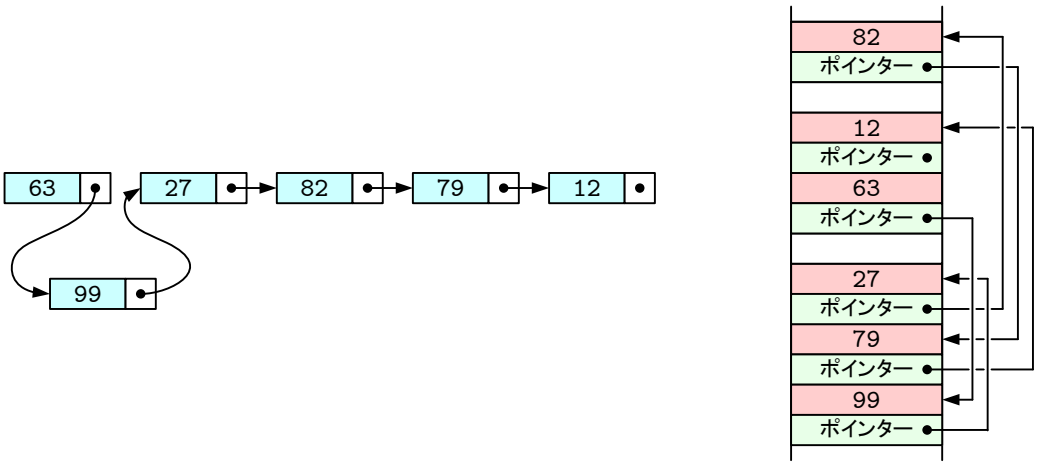

図 4: リストのデータの挿入の様子.

2.1.3　リストのデータの削除

次に，先ほど 99 を挿入した数列

$$63,\ 99,\ 27,\ 82,\ 79,\ 12$$

から，99 を削除することを考える．データの削除も挿入とほとんど同じ考え方ができる．

配列の場合，図 5 に示した方法で，データを削除する．挿入とは逆に，前の方から順番にコピーを繰り返す．処理の回数は N に比例するので，計算量のオーダーは $O(N)$ となる．

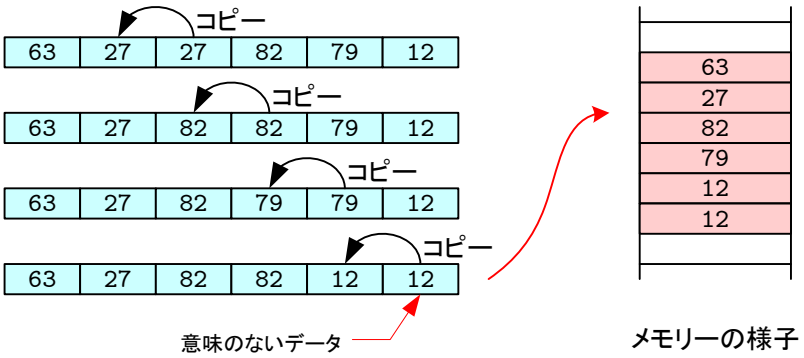

図 5: 配列のデータの削除の様子．

リストの場合，図 6 に示した方法で，データを削除する．データ 63 のポインターをデータ 27 に接続して，データ 99 を削除する．データ数とは関係なく処理の回数は決まっており，オーダーは $O(1)$ となる．

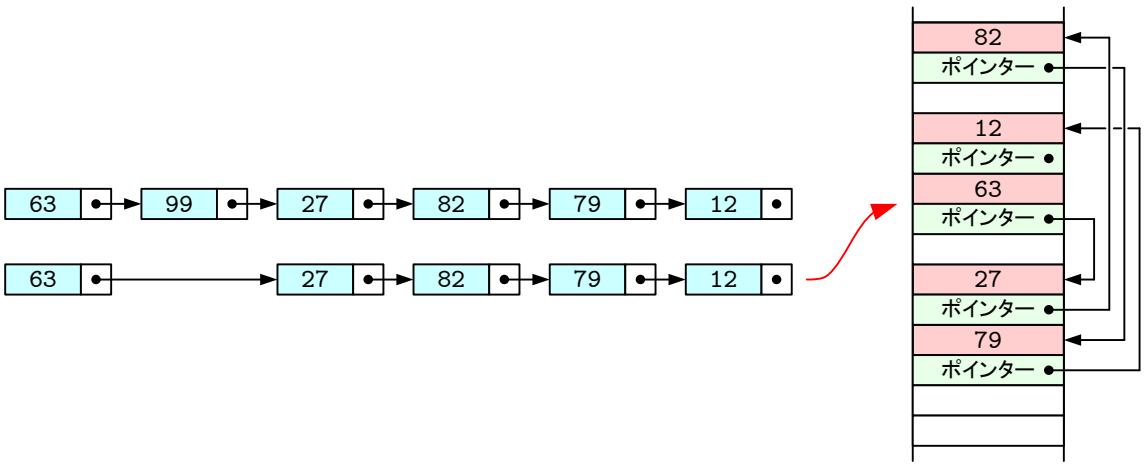

図 6: リストのデータの削除の様子．

2.1.4 データへのアクセス

次に，データのアクセスを考える．図 1 や 2 のように，配列やリストを用いて，数列

$$63,\ 27,\ 82,\ 79,\ 12$$

がメモリーに格納されているとする．これの 79 というデータにアクセスするにはどうするか?

配列の場合，a[3] とすれば 79 のデータにアクセスすることができる．メモリーのアドレスの順にデータが並んでいるので,「先頭のアドレス+3×(ひとつのデータのバイト数)」の計算により，目的のデータのアドレスは即座に計算できる．なんの迷いも無く目的のメモリーアドレスが分かるので，計算量は $O(1)$ となる．データがいくら増えてもデータのアクセスの時間は変化しない．

リストの場合，目的のデータへのアクセスは手間がかかる．データのある場所が分かるのは先頭のデータのみである．先頭のデータ 63 の次のデータ →27 の次のデータ →82 の次のデータ →79—というように先頭からたぐりよせなくてはならない．したがって，N 個のデータがある場合，処理の平均的な回数は $N/2$ となる．計算量は $O(N)$ である．データの数に比例して，処理の時間がかかることを表している．

2.1.5 リストと配列の違いのまとめ

配列は目的のデータにランダムアクセスが可能で，目的のデータの値を得たり，データの値の変更が高速にできる．添え字を指定するだけで，それらが可能である．しかし，データの削除と挿入には計算回数が多くなる．

一方，リストは目的のデータにアクセスするためには，シーケンシャルアクセス[3]を行う．そのため，データへのアクセス回数が多くなり配列より低速になる．しかし，データの削除と挿入は簡単は高速である．

また，データの個数の柔軟性はリストの方が高い．配列の場合，要素の数はコンパイル時に予め指定する必要がある．連続したメモリー領域を確保するためである．配列の場合，データ数が予め分からない場合は，十分大きなメモリー領域を確保することになる．そのため，メモリーを無駄遣いすることがある．それに対して，リストはデータの個数は後で追加/削除できる．

これらの違いをまとめると，表 1 のようになる．一般的には，データの追加/削除が多い場合にはリスト，データのアクセスが多い場合には配列を使う．

表 1: 配列とリストとの違い

	配列	リスト
データへのアクセス	添え字によるランダムアクセス可能	リストを順にたどる
アクセスのための計算量	$O(1)$	$O(N)$
データの挿入/削除	計算コスト大 ($O(N)$)	計算コスト小 ($O(1)$)
メモリーのコスト	小	配列より大
データ数	コンパイル時に決定	追加/削除可能

[3]データを先頭から順番に読み込み、あるいは書き込みを行なう方法。

2.2 スタック

最後に入れたデータをはじめに取り出すスタックと呼ばれるデータ構造について説明する．

2.2.1 スタックの概要

スタック (stack) とは，最後に入れたデータを最初に取り出せるようにしたデータ構造である．図 7 のようなデータ構造である．

データ構造であるから，データを蓄えることと，それを取り出すことができる．スタックの特徴は，最後に入れたデータが一番最初に取り出すことにある．取り出されるデータは，格納されている最新のデータで，最後に入れられたものが最初に取り出されることから，LIFO(last in first out, 後入れ先出し) と呼ばれる．スタックの途中のデータを取り出すことは許されない．

スタックにデータを積むこと—データの格納—をプッシュ(push) と，スタックからデータを取り出すことをポップ (pup) と呼ぶ．

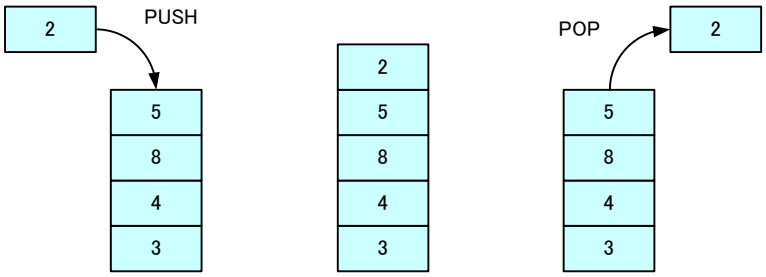

図 7: スタック

スタックは単純なデータ構造であるが，有用でいろいろな場面で使われる．例えば，次のような場合である．

- 関数を呼び出した場合，呼び出し元のデータをいったん保存するときのデータ構造として使われる．
- 算術式の評価 (教科書 [1] の pp.189–192)．これは逆ポーランド記法でかかれた数式を計算するプログラムである．

2.2.2 逆ポーランド記法

リストを使った応用として，逆ポーランド記法がある．コンパイラーの設計の時にこれが役に立つ—らしい．これは，逆ポーランド記法の計算式の文字列

```
8 4 + 9 6 - /
```

があるとすると，次のような順序で計算を行う．

[**手順 1**] 式を書いた文字列を取り出す．取り出す順序は式の左側から．

- 取り出した文字列が数字ならば，スタックにその数値を push する．
- 演算子 ($+ - */$) ならば，push を 2 回繰り返して，スタックから 2 つ数値を取り出す．そして，演算を行う．演算結果は，スタックに push する．

[**手順 2**] 式の次の文字列について，[手順 1] を行う．式の文字列がなくなるまで，これを繰り返す．

[**手順 3**] 式の文字列がなくなったとき，スタックに残った値が計算結果となる．

具体的な計算例を，図 8 に示す．この例をしっかり理解する必要がある．

二項演算子のうち，計算の順序の交換ができない場合— - / のとき—は演算の順序に注意する必要がある．たとえば， 5 3 - は，2 なのか? -2 なのか?と言うことである．これはどちらでもよく，試験で出題する場合は，定義を示す．例えば,「x y - は x から y を引く」と記述する．

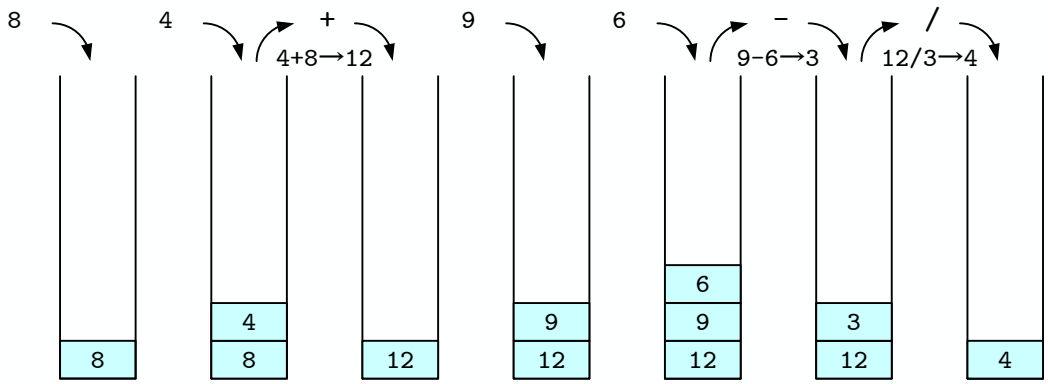

図 8: 逆ポーランド記法で記述された数式 8 4 + 9 6 - / の計算の様子

2.3 キュー

最初に入れたデータをはじめに取り出すキューと呼ばれるデータ構造について説明する．

2.3.1 イメージ

キュー (queue) は，待ち行列とも呼ばれるでーた構造である．これは，窓口に並ぶ順番待ちの行列の意味で，図 9 のようなデータ構造となっている．スタックではデータの挿入と取り出しが列の一方からのみであったの対して，キューは列の両端から行う．一方がデータの追加で一方がデータの取り出しとして使われる．キューでは，最初に入れたデータが一番最初に取り出されることにある．取り出されるデータは格納されている最古のデータで，最初に入れられたものが最初に取り出されることから，FIFO(first in first out, 先入れ先出し) と呼ばれる．スタック同様，スタックの途中のデータを取り出すことは許されないのである．

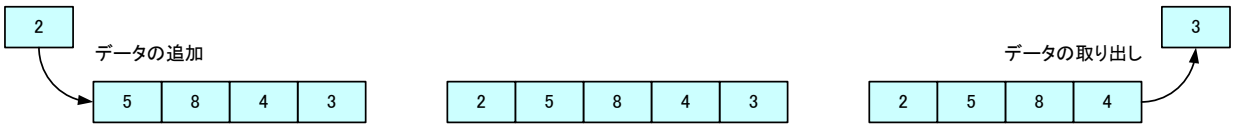

図 9: キュー

キューはスタックに比べて少しばかり，構造が複雑になっている．実際，それを直線的なイメージのメモリーにデータを追加しようとすると，以下のような問題が生じる．

- FIFO を続けると，いずれはメモリーの端に到達して，データの追加が出来なくなる．
- データを追加したり取り出したりする毎に，データの列を移動させることも考えられる．こうすると計算量が増加して不経済である．

これを防ぐために，図 10 のようなリングバッファと言うものが考えられた．

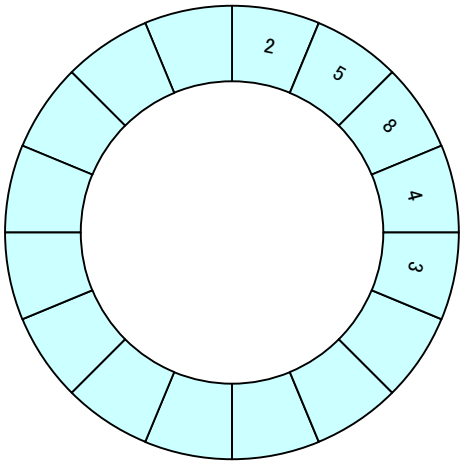

図 10: リングバッファー

これは，入れられた順序通りに処理すべきデータに使われる．たとえば，次のような応用がある．

- データをバッファーにためて，処理を行う場合．プリンターの処理などである．プリント待ちのデータをプリントキューと言うことがある．
- オンライントランザクション処理[4]の管理に用いれれる．この処理では，原則的には到着順に処理しなくてはならない．

[4]ネットワークに接続された複数のパソコンがホストコンピュータに処理要求を行い、ホストコンピュータがその要求にもとづいてデータを処理し、処理結果を即座にパソコンに送り返す処理方式。データベースの処理などに多く使われる．

2.4 ツリー

2.4.1 ツリーの例

ツリー (three:木) と呼ばれるデータ構造は，階層構造を持ったデータの集まりを表すのに都合が良い．たとえば，組織図，コンピューターのファイルシステムなどである．情報科学では，データベースの中のデータの表現やコンパイラーでの原始プログラムの文法構造などでも使われる．

2.4.2 ツリーの基本

基本用語　ツリー構造にはいろいろ名称があり，それを表 2 と図 11 に示す．この図を反対にして見ると，根が一番下になり枝や葉が上にあることが分かるであろう．まさに，木 (tree) である．

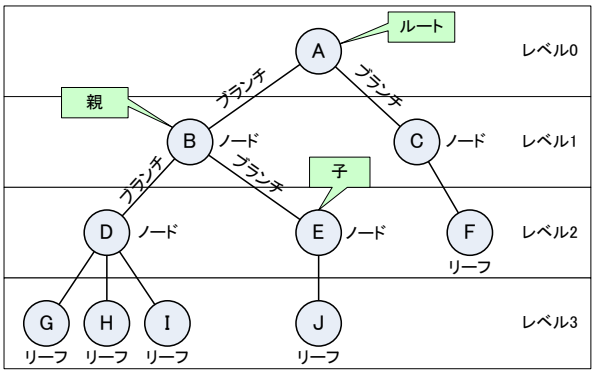

図 11: ツリー構造と名称．B と E は親子関係のの例である．

表 2: ツリー構造の名称

構成要素	内容
ルート (root:根)	最上位のノード
ノード (node:節)	枝が分かれるところ
リーフ (leaf:葉)	子がないノード
ブランチ (branch:枝)	親と子を結ぶ線
親 (parent)	上のノード
子 (child)	下のノード
兄弟 (brother)	同じ親を持つノード

特徴　子は一つの親を持つことがツリーの特徴である．二つの親を持ってはならないのである．そのため，任意の二つのノードあるいはリーフのつながりを表すパスは，一意に決まる．

2.4.3 2 分探索木

2 分木 (binary tree) は，各ノードの子の数がたかだか 2 であるようなツリーである．これは単純ではあるが，

- 左側の子は，その親より必ず小さい．
- 右側に子は，その親より必ず大きい．

のようにすると，順序木を作ることができる．特に 2 分木の順序木を 2 分探索木という．図 12 にその例を示す．2 分木を，中央順 (postorder)[5]に並べると，

$$11 \rightarrow 15 \rightarrow 20 \rightarrow 23 \rightarrow 31 \rightarrow 37 \rightarrow 48 \rightarrow 55 \rightarrow 65 \rightarrow 73 \rightarrow 81 \rightarrow 88 \rightarrow 94$$

となる．昇順にデータが並んでいることが分かる．

この 2 分探索木は応用範囲がとても広い．ツリー構造の中でもっとも単純ではあるが，アルゴリズムで使用される頻度はもっとも高い．単純なものほど応用範囲が広い—という一つの例である．

この構造のあるノードの左側と右側はそれぞれの子を頂点とする同じ性質を持った木構造になる．この子孫に対しても，同じ規準が適用されれるので，再帰による処理が可能となる．

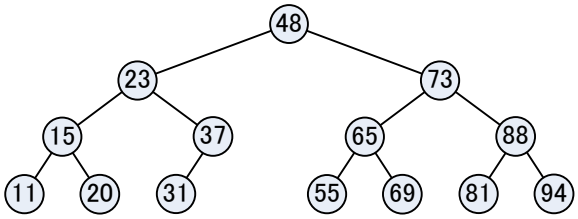

図 12: ツリー構造 (2 分探索木)

2.4.4 データの作成

以下の順序で 2 分探索木のデータの作成の手順は以下の通りである．

- 最初のデータ—データが無いとき—は，ルートにデータをおく．
- データがある場合には，ルートから大小関係をたどり，データがまだない子孫の場所へデータを置く．

以下の順序でデータでデータが送られてきた場合，2 分探索木の作成例を示す．

```
6,10,2,8,12,4
```

図 13 の順序で 2 分探索木ができる．

[5]左の子 → 親 → 右の子の順

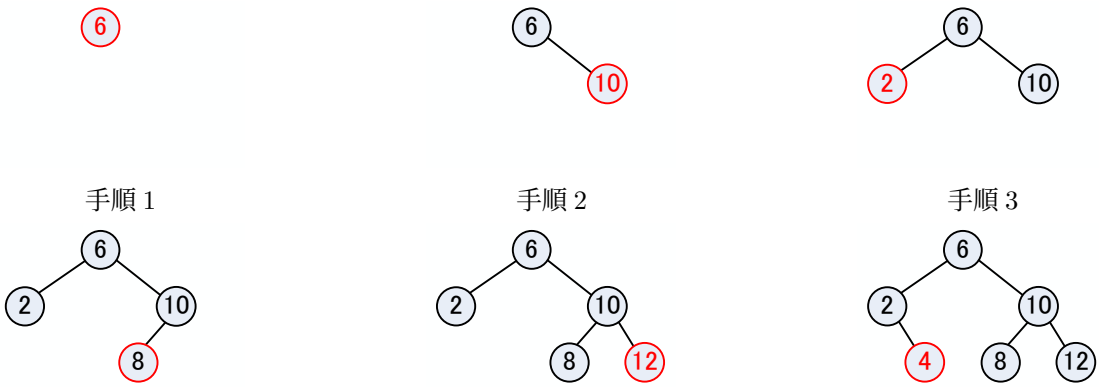

図 13: ツリーのノードの作成順序

2.4.5 データの追加

先ほど作成したツリーにデータ 9 を追加する．これも，ルートから大小関係をたどっていき，子が無いところへデータを挿入する．すると，図 14 のようになる．

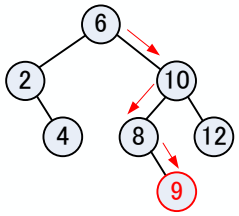

図 14: ツリーへのデータの追加

2.4.6 データの削除

元のツリー図 15 からデータを削除することを考える．

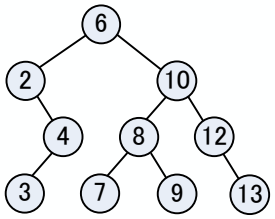

図 15: データを削除する前の 2 分木

データを削除する場合，子が無い場合，子がひとつの場合，子がふたつの場合にに分けて考えなくてはならない．

- 削除するデータに子が無い場合は，そのまま削除する (図 16).
- 削除するデータがひとつの子をもつ場合は，削除した場所にその子と子孫を入れる (図 17).
- 削除するデータがふたつの子を持つ場合．削除した場所に，左側の子孫の最大値 (あるいは右側の子孫の最少値) を移動させる (図 18).

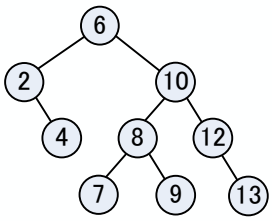

図 16: 元のツリー図 15 から 3 を削除．子が無い場合は，そのまま削除する．

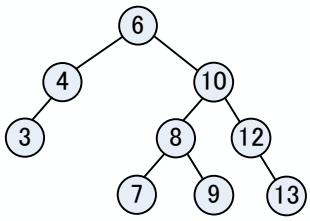

図 17: 元のツリー図 15 から 2 を削除．子がひとつの場合は，削除した部分にその子を移す.

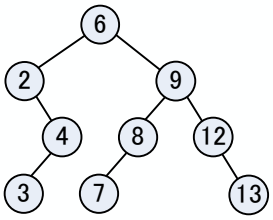

図 18: 元のツリー図 15 から 10 を削除．子が二つの場合は，削除した部分に，左側の子孫の最大の値をもつノードを移動させる．

2.4.7 2 分木の実装

2 分木は，ノードとその接続状態を表すブランチからできている．C 言語でこれを表現するためには，ノードの中でデータと接続状態を表す変数を用意すればよい．リストを思い出せ，それとほとんど同じではないか! リストと同じように，ノードをあわす構造体を使えばよい．

```
typedef struct node_tag{
  int value;
  struct node_tag *left;
  struct node_tag *right;
```

```
}node;
```

このようなノードを使うと，図 19 のようにツリー構造を表すことができる．ノードに子が無い場合には，子を表すポインターに NULL を入れる．これはヌルポインターと呼ばれ,「なにも指し示さないポインター」と言うことを明示している．

ツリー全体を表すために，ルートを表す構造体

```
node *root_tree=NULL;
```

を宣言しておく．初期値は意味のないポインター (NULL) を入れておく．最初はルートがないため，それを表すポインターは意味が無いためである．

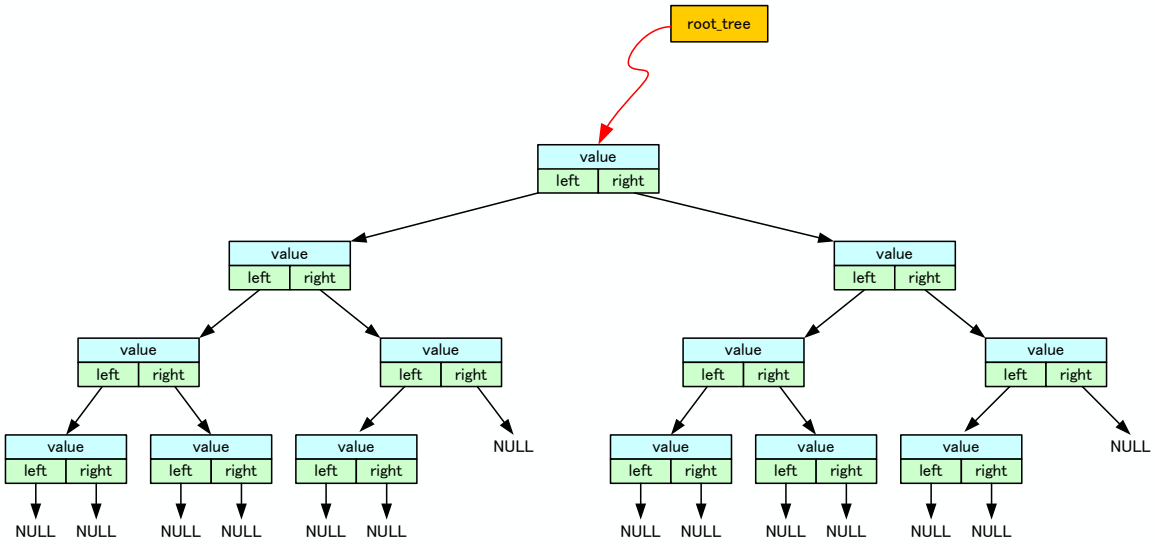

図 19: 構造体を使ったツリー構造

3　アルゴリズム

3.1　単純挿入ソート

3.1.1　単純挿入ソートの考え方

単純挿入ソートは，a[0] から a[i-1] まで並び替えが終わっているとき，a[i] を正しい位置に入れる—ことを繰り返す方法である．その様子を図 20 に示す．処理の手順は次の通りである．

1. 処理する a[i] の値 35 よりも，48 は大きいので 35 の場所へ移動させる．
2. 次の 44 も 35 より大きいので，48 の場所へ移動させる．
3. 次の 41 も 35 より大きいので，44 の場所へ移動させる．

4. 次の 39 も 35 より大きいので，41 の場所へ移動させる．
5. 次の 32 は 35 より小さいので，それは移動させない．そして，元の 35 を 39 の場所へ移動させる．

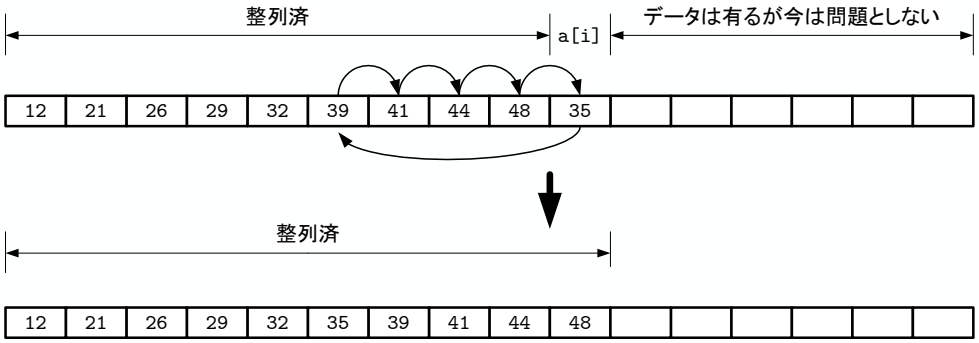

図 20: 単純挿入ソートの基本操作．a[i] を正しい位置に入れる．

3.1.2 プログラム

単純挿入ソートの実際のプログラム (関数) は，教科書 [1]p.219 のリスト 6.20 のように書く．整列させる整数のデータは，配列は array[] に格納されている．データの個数は num に格納されている．したがって，整列すべき整数のデータは，配列の array[0]〜array[num-1] に格納されている．

この関数がソートする様子を図 21 に示す．先に示したアルゴリズムの通りであることが分かるだろう．

この関数を呼び出すときには，次のようにする．

```
insertion_sort(a,10);
```

整列すべき整数は，a[0]〜a[9] に入っている．データの個数は 10 個である．

著作権の関係で，教科書のこのプログラムはここには書かないが，よく理解しておけ!

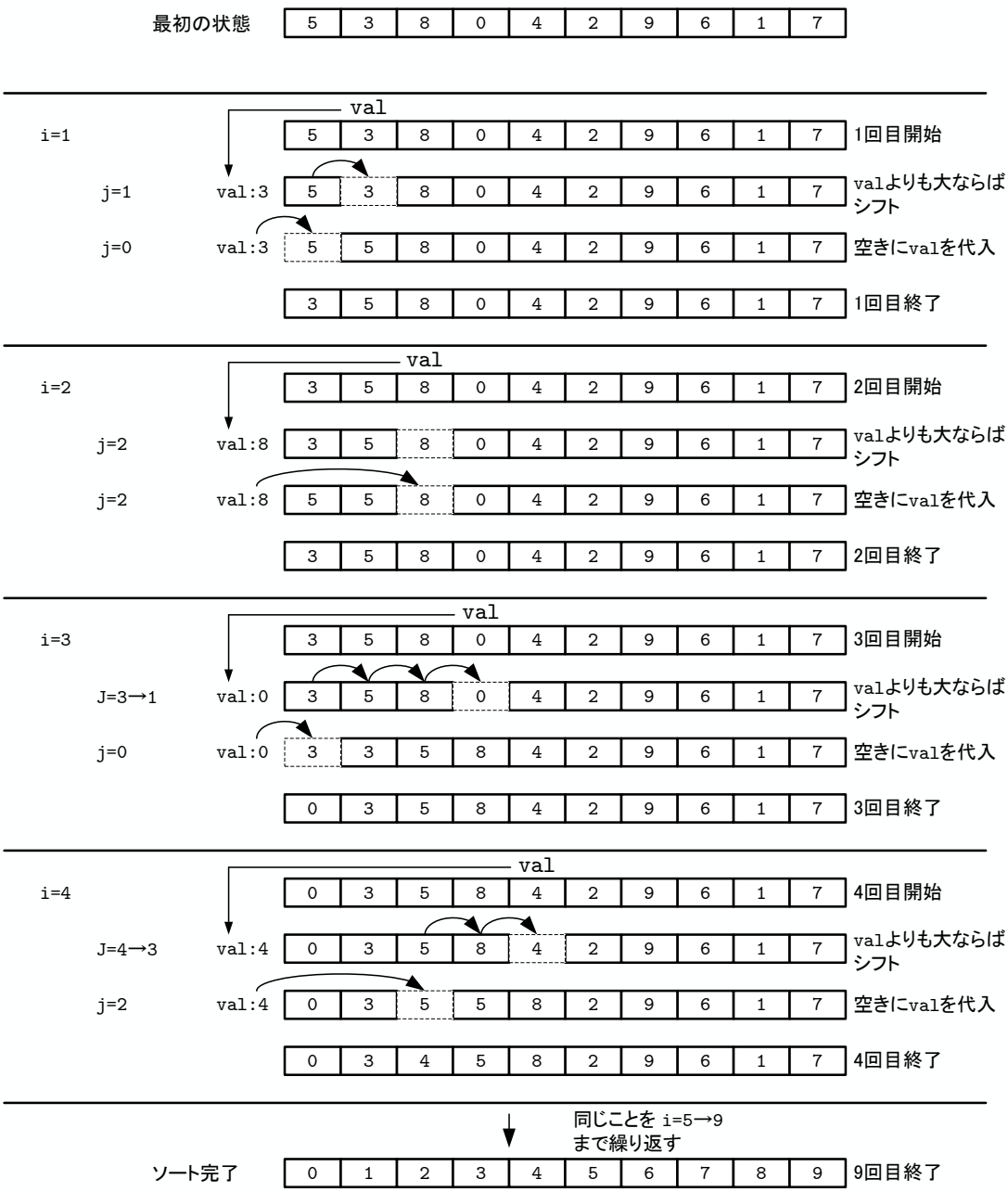

図 21: 単純挿入ソートを使った整数の整列．教科書 [1]p.219 のリスト 6.20 `insertion_sort()` 関数の動作.

3.1.3 計算量

単純挿入ソートの計算量は，内側の繰り返し分のループ回数で決まる．教科書 [1]p.219 のリスト 6.20 では，j の部分のループ回数である．ここでは，その計算量のオーダーを見積もる．

ソートするデータの個数を N とする．外側のループは $i = 1$〜$N - 1$ までである．データがランダムの場合，内側のループは平均して $i/2$ 回繰り返される．したがって，内側のループの総繰り返し回数は，

$$\text{内側のループの繰り返し回数} = \sum_{i=1}^{N-1} \frac{i}{2} = \frac{1}{2} \sum_{i=1}^{N-1} i = \frac{1}{2} \times \frac{(N-1)N}{2} = \frac{N^2 - N}{4} \tag{1}$$

となる．データの個数が大きくなると，N^2 が支配的になる．したがって，計算量のオーダーは $O(N^2)$ となる．コンピューターでの処理の時間は，データ数の二乗 (N^2) で増加するということである．

参考文献

[1] 内田智史監修, (株) システム計画研究所編. C 言語によるプログラミング　応用編　第 2 版. (株) オーム社, 2006.

[2] Willam H. Press et al. NUMERICAL RECIPES in C [日本語版]. 技術評論社, 1996.