# 卒 業 研 究 報 告

題　　目

データ構造とアルゴリズム

「リスト構造」と「木構造」

---

指 導 教 員

山本　哲也教授

報 告 者

西村　岳史

平成　　14 年　　2 月　　5 日

高知工科大学 電子・光システム工学科

目次

**第 1 章　はじめに……………………………………………………………………3**
1.1　　はじめに……………………………………………………………………3
1.2　　アルゴリズムの例……………………………………………………………4
1.3　　効率性の評価方法……………………………………………………………4

**第 2 章　リスト……………………………………………………………………6**
2.1　　配列……………………………………………………………………6
2.2　　リスト構造……………………………………………………………………7
2.3　　リストの表現……………………………………………………………………7
2.4　　リストの操作……………………………………………………………………10
2.5　　リストの探索アルゴリズム……………………………………………………13

**第 3 章　木構造……………………………………………………………………15**
3.1　　木構造……………………………………………………………………15
3.2　　ヒープ……………………………………………………………………16
3.2.1　ヒープの操作……………………………………………………………………18
3.3　　2 分探索木……………………………………………………………………21
3.3.1　2 分探索木の操作……………………………………………………………22
3.4　　2 分探索木の探索アルゴリズム………………………………………………24

**第 4 章　データ効率の比較……………………………………………………………26**

**第 5 章　結論……………………………………………………………………29**

**第 6 章　まとめ……………………………………………………………………30**

**第 7 章　謝辞……………………………………………………………………32**

**第 8 章　参考文献……………………………………………………………………33**

**<付録>　プログラムリスト……………………………………………………………34**

# 第1章　はじめに

本研究は調査研究である。

## 1.1 はじめに

コンピュータは“計算をする機械”として生まれ発達してきたが、コンピュータの性能が格段に進化し、様々な分野で利用されるようになった最近では、単なる計算をする機械というよりは、むしろもっと広く“情報を処理する機械”と捕らえる方が自然に思えるようになってきている。いずれにしても、コンピュータに計算や情報を処理させるには、その処理の手順(どのような処理をどのように実行してどのような結果を出力すればいいのか等)をコンピュータに指示してやる必要がある。その処理手順を、決められた言語(プログラミング言語)で記述したものがプログラムである。プログラムを自分で開発するにしても、市販のプログラムを購入して利用するにしても、プログラムの良し悪しが処理時間や処理結果を左右することはいうまでもない。

では、良いプログラムとはどのようなプログラムなのだろうか。プログラムの良し悪しを判断する基準となり得るものを挙げる。

1) 処理時間　　（処理時間の速いプログラムが良い）
2) 記憶領域　　（記憶領域あまり必要としないプログラムが良い）
3) プログラム　（わかりやすいプログラムは良い）

等々、プログラムの良し悪しを判断する評価基準として、いろいろ考えられる。しかし、一般にプログラムを評価、比較するときには、その処理時間や記憶領域を評価基準とすることがほとんどであり、その他のものはまだ評価基準としては定着していない。スーパーコンピュータやワークステーション等が非常に高速になり、又、メモリも安くなったので、力づくで問題を解決し、プログラムの処理時間や記憶領域よりもプログラムのわかりやすさに注目するべきであるという意見も一方で存在する。しかし我々はコンピュータの性能が向上したことによって、これまで現実的に処理できなかった大規模な問題を処理できるような環境が与えられたが、以前にも増してプログラムの処理時間や記憶領域などが重要であると考えられる。

では良いプログラムを開発するにはどのようにすればいいのだろうか。それには、プログラムを書く前の段階で、処理の基本的な考え方、解き方、一連の手順を良く考えなければならない。この一連の手順はその通りに実行していけば、必ず意図した通りの結果が得られるものである。このような一連の処理手順をアルゴリズム(algorithm)という。プログラムはアルゴリズムをプログラミング言語で記述したものに他ならない。したがって、良いプログラムを開発するには良いアルゴリズムが必要である。

本稿では、良いアルゴリズムに焦点をあて、様々なアルゴリズムを紹介するとともに、それらの基本的データ構造(与えられた問題をコンピュータで扱えるように形式的に表現すること)を用いた探索法(リスト構造と木構造)について述べ、それぞれのデータ構造の違い

による時間計算量の違いについて評価する。又、リスト、木のどちらがよりデータ処理に適した探索アルゴリズムかを調べるのを目的とする。

本稿で使用したプログラムは参考資料として末稿の<付録>で記す。プログラムはリンク構造を実現するデータ型が豊富であり、データ構造の実現に有効とされている C 言語を用いた。

### 1.2 アルゴリズムの例

アルゴリズムの代表例としては最大公約数を求めるユークリッドの互除法、最小木を求めるクラスカルのアルゴリズム、指定したものが存在すれば yes、なければ no という答えを導きだす SUBSET-SUM 等がある。本稿ではアルゴリズムの中でも最も古いユークリッドの互除法について述べる。

ある目的のために開発されたアルゴリズムが無限ループに陥ることなく、正しい解あるいは意図した解を導くかどうかを検証することは重要である。ユークリッドの互除法とは 2 つの自然数$a,b(a \geq b \geq 0)$の最大公約数(GCD：Greatest Common Divisor)を求めるアルゴリズムである。

2 つの整数$S_1$、$S_2$があり、$S_1 = a, S_2 = b$とおき　$S_i = S_i - 2 \quad \text{mod} \quad S_i - 1$　($S_i - 2$を$S_i - 1$で割る)をする。この時、第$k$項において$S = 0$となったとする。その時の一つ前の項$S$が 2 数$a,b$の GCD となっているという。

### 1.3 効率性の評価方法

アルゴリズムに従って処理を実行したときに使用される記憶領域をそのアルゴリズムの領域計算量(メモリサイズ)といい、その時にかかる計算時間を時間計算量という。アルゴリズムの効率性を比較するのには領域計算量、時間計算量のどちらかが主に使用される。本稿では、時間計算量を中心にアルゴリズムの効率性を調べていく。

求めた計算量が少し複雑な関数となり、アルゴリズムの効率が比較しにくくなる状況が起こる。そこで、実際には要素$n$の値が大きくなったところで計算量がどれだけあるかということが重要になるので$n$が十分に大きいところ($n \to \infty$)で計算量を漸近的に評価する手段としてオーダ記法を考える。

オーダ記法とは、ある関数$f(n)$に対して、計算量$T(n)$が$O(f(n))$(オーダ$f(n)$という)であるという。これは 2 つの正の定数$n_0$と$C$が存在して、$n_0$以上のすべての$n$に対して、

$T(n) \leq Cf(n)$

が成り立つことをいう。

このようなオーダ記法によって計算量$T(n)$を評価する場合には、できるだけ$T(n)$の増加の様子をよく表す単純な関数（$n, n\log n, n^2, n^3, 2^n, n!, n^n$のような関数)を用いる。例えば時間計算量が　$T_a(n) = 1000n^3$，$T_b(n) = 1000n^2 + 10\log n$，$T_c(n) = n^4$　のアルゴリズムがあるとする。これらはオーダ記法によりそれぞれ$O(n^3), O(n^2), O(n^4)$となり$T_b(n)$のア

ルゴリズムが一番効率が良い事がわかる。

オーダ記法を用いる事により、漸近的ではあるが、計算量の評価をすることができるようになりアルゴリズムの効率を簡単に比較することができるようになる。ただ注意しなければいけないのは、この評価は漸近的な評価であって、入力のサイズが十分に大きくなったときに成り立つ評価であるということである。

例えば先程の計算量$T_a(n)=O(n^3), T_c(n)=O(n^4)$の場合であるが、これは$n$の値が大きい時では$T_a$の方が効率が良いが、$n$の値が 10000 より小さい場合には$T_a(n)\geq T_c(n)$となりアルゴリズム$C$の方がかえって効率が良いということになる。このような場合や、同じオーダの計算量のアルゴリズムを比較するような場合には、係数や無視した低次の項等も考慮に入れたもっと細かい計算をする必要がある。図 1.1, 1.2 は計算量のオーダ評価によく用いられる関数$n, n\log n, n^2, n^3, 2^n, n!, n^n$の値の変化に従ってどのように変化するかをグラフにしたものである。横軸は入力のサイズ$n$、縦軸は計算時間(ただし$10^6$を 1 秒とする)を示す。

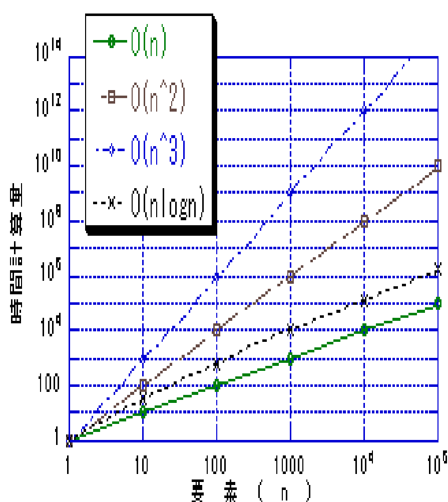



図 1.1 多項式時間アルゴリズム

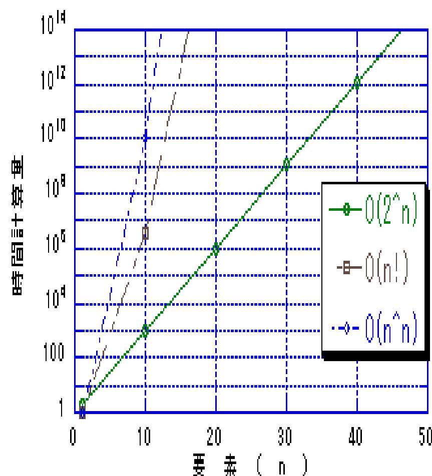



図 1.2 指数時間アルゴリズム

図 1.1 のようなアルゴリズムの時間計算量が$n, n\log n, n^2, n^3$等のような$n$の多項式のオーダであるようなアルゴリズムを多項式時間アルゴリズム(polynomial time algorighm)と呼び、図 1.2 のような$n$の指数関数($2^n, n!, n^n$等)のオーダであるようなアルゴリズムを指数時間アルゴリズム(exponential time algorithm)と呼ぶ。図からわかるように、多項式アルゴリズムは現実的に計算機に実行させ得るアルゴリズムであるのに対して、指数時間アルゴリズムは入力のサイズがちょっと大きくなっただけで爆発的に計算時間が大きくなり現実的なアルゴリズムとはいえない。本稿でデータ探索に用いるアルゴリズムは多項式アルゴリズムを用いる。以上のようなことを含めつつ、第 4 章で探索アルゴリズムの要素数と時間計算量を評価し、データ量に対して有効なデータ構造を検討する。

# 第2章　リスト

## 2.1 配列

配列とは、メモリ上に同じ大きさで区切って並べてとった記憶場所である。したがって、配列に蓄えられるデータの性質は、次のようになる。

(1) 配列に蓄えられるデータは同じ種類のものである。異なった種類のものが混在することはない。

(2) 全体として、一つの配列名がつけられる。

(3) 特定のデータは、頭から何番目のものかにより指定する。それは最初を 0 とするものと 1 とするものがある。本稿では $C$ 言語を用いるので最初を 0 とする。

配列の表し方を図 2.1.1 に表す。

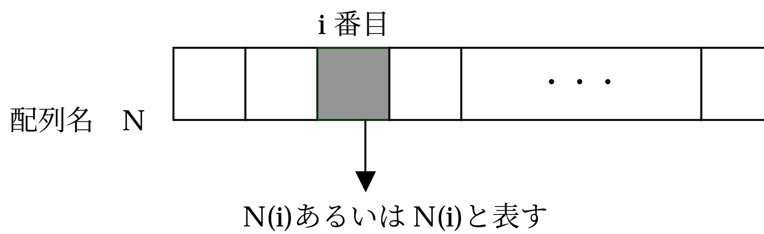



図 2.1.1　配列の表し方

配列にデータがすでにあるとして、そこに新しいデータをつけ加えたり削除することを考える。最後につけ加えたり、あるいは最後のデータを削除する時には何の問題もないが、配列の途中で追加削除をする時の操作には注意が必要であり、配列の途中に新しいデータを追加したり削除したりするには多くのデータ移動をともなわなければならない。
(図 2.1.2)

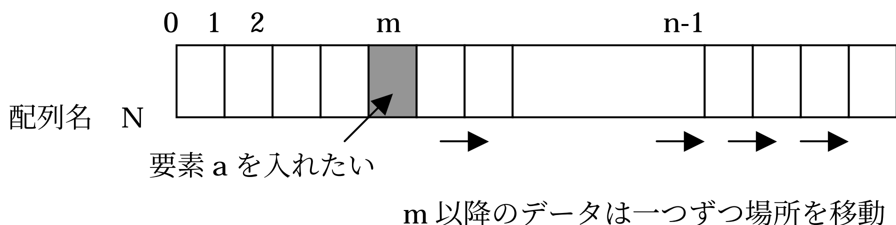



図 2.1.2　配列の途中 m に a を入れるとき

## 2.2 リスト(list)構造について

配列は便利なものであるが、途中に新しいデータを追加したり、あるいは削除したりするには、多くのデータ移動を必要とする。決まったデータを取り扱い、データ数の変化がない場合(静的)は、配列は扱いやすいデータ構造といえる。しかし、処理の途中に新しいデータが発生したり、あるいは削除されたりする場合、データがダイナミック(動的)に変化する場合は、データの蓄え方の新しい工夫が必要となる。リスト構造は対象となるデータが動的に発生消滅する場合に、それに対処するためのデータ構造である。

リストという言葉は情報処理の分野では、”順序付けられたデータ要素の並び”とされていて、並びと言われる事もある。リストは、それを構成する要素が順番に並んだ構造をしている。

$a_1, a_2, a_{3,}........., a_n$

とリストの表現をしたとき、$n$ をリストの長さ、$a_1$ が先頭要素、$a_n$ が末尾要素である。(図 2.2.1)

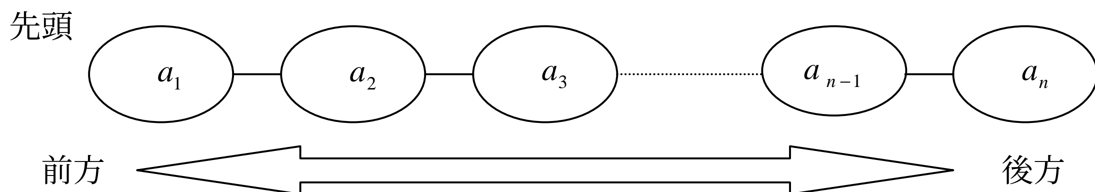



図 2.2.1　リストの概念

リストの要素は値に 1～$n$ までの順序が付けられており、$a_i$ の要素の直前要素は $a_{i-1}$、直後の要素は $a_{i+1}$ である。

## 2.3 リストの表現

リストの実現方法は 2 通りある。

一つは要素を 1 次元配列に連続的に格納する方法(2 章 2.1　配列参照)(図 2.3.1)

もう一つは

要素とそれに隣接するセル(cell)の位置を指すポインタ(pointer)の組によるリンク構造(図 2.3.2)

により実現される。

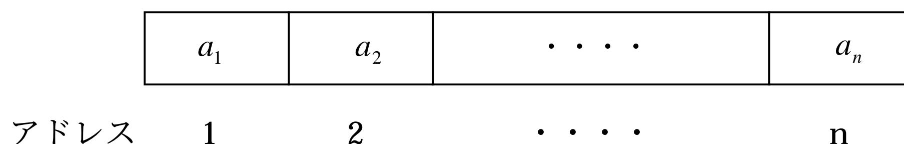



図 2.3.1　1 次元配列によるリスト

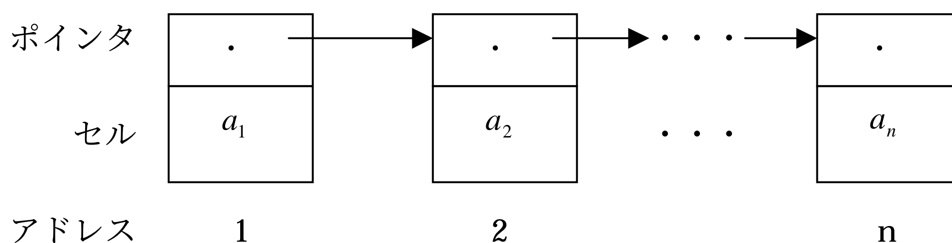
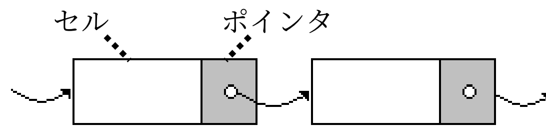


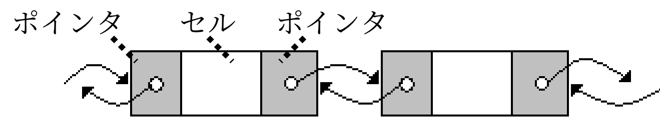

図 2.3.2　リンク構造による実現

ポインタとは、次のセルの位置を示すデータであり、矢印で示される。つまり次のセルのアドレスとなる。要素の事をセルとも呼ぶ。1 次元配列により実現されたリストはメモリ上に本当に並んでいるので。物理的に並んでいるといい、リンク構造により実現されたリストはデータが次々にたどれるという意味でデータが並んでいるのと同じことなので、論理的に並んでいるという。前者での要素の削除・挿入は、通常、その後方の全要素を移動させる必要が生じるため効率が悪い。一般には、リストというと後者のリンク構造で実現されたリストを指す事が多い。本稿でも”リスト”という用語はリンク構造で実現されたリストを指す事にする。

リストには図 2.3.3 に示すように、1 つのポインタが直後の要素へのリンク情報しか持たないようなリスト構造をした片方向リスト(リンク情報が 1 つしかないためリストの前方から後方への検索のみ行う事ができる)や、ポインタが前と後ろの両方あるため、直前と直後への要素の両方向へのリンク情報を持つようなリスト構造をした双方向リスト(前後 2 つのリンク情報がある為、リストの前方と後方の両方向から検索することができる)がある。



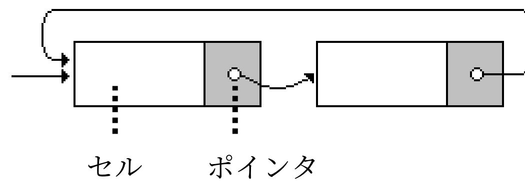

片方向リスト



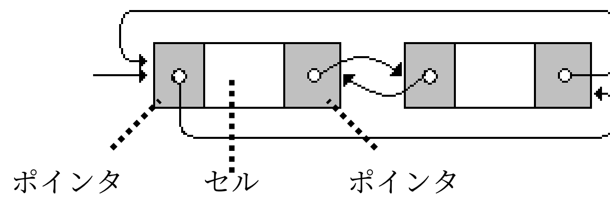

双方向リスト

図 2.3.3 リスト

又、リストの末尾からリストの先頭へのポインタをもつようなリストを環状リストという。環状リストを図 2.3.4 に示す。




片方向環状リスト




双方向環状リスト

図 2.3.4　環状リスト

### 2.4 リスト構造の操作

#### 2.4.1　要素の探索

リストを使い要素の探索を行う。これはリストの中に、指定された特定の値をもつ要素が存在するかどうかを調べ、存在していたらその要素の配列アドレスを表示し、存在していなければ 0(NULL)を表示する。又、入力された要素の数が用意したセル領域(今回はセル領域を 100 とおく)を超えるとエラーメッセージが表示される。このプログラムを末稿の" プログラム 1　リストの要素の探索 " に示す。

#### 2.4.2 要素の挿入

リストを使い指定した位置に、与えられた要素を挿入する。これはリストの中に、新たな要素を指定した場所に挿入するというものである。今回は、プログラムの中で、リストを作成しておき、5 番目の配列に 12 という要素を挿入するというものである。このプログラムを末稿の " プログラム 2　リストの要素の挿入 " に示す。

#### 2.4.3 要素の削除

リストを使い指定した要素をリストの中から削除する。これは指定した要素をリストの中から削除するというものである。挿入の時と同じく、リストはプログラムの中で作成しておく。今回は配列 7 の要素 12 を削除するようにプログラムを作成した。このプログラムを末稿の " プログラム 3　リストの要素の削除 " に示す。

その他の操作として

**スタック(stack)**

**キュー(queue)**

**デク(dequeue)**

がある。

- **スタック**　スタック構造はデータを積み重ねて保存するデータ構造である。積み重ねてデータを保存するため、積み重ねた途中のデータを取り出すことはできない。データを積むことをプッシュ(push)、取り出すことをポップ(pop)という。スタック構造でのデータの出し入れは、最後のデータを最初に取り出すことになる。このような方式を LIFO（last in fast out：後入れ先出し）と呼び、C 言語における関数のデータの受け渡しなどに使われている。スタック構造は配列を使って図 2.4.1 のように実現する。
- **キュー**　**キュー構造はデータを待ち行列を作って保存するデータ構造である。キュー構造で使われる待ち行列とは、ちょうどスーパーのレジの清算を待つ行列のように、あるサービスを受けるために順番を待って並ぶ行列のことである。データを入れることを enqueue、取り出すことを dequeue という。キュー構造では、**

**新しく追加されるデータは列の最後に追加され、取り出されるデータは列の最初から取り出される。したがって、このような構造を待ち行列、又は FIFO（first in last out：先入れ先出し）と呼び、UNIX でのプロセス間通信を行うパイプ処理などに使われている。キュー構造は配列を使って図 2.4.1 のように実現する。**

**・デク**　**デク構造はキュー構造と同じで待ち行列を用いるデータ構造であるが、新しく追加するデータは列の最後からでも最初からでも挿入することができ、取り出すデータも同じように列の最初、最後から取り出すことができる。デク構造は配列を用いて図 2.4.1 のように実現できる。しかし、デク構造はプログラム構造が複雑であり、その操作はスタックとキューで行う事ができるということから使用される事はあまりない。**

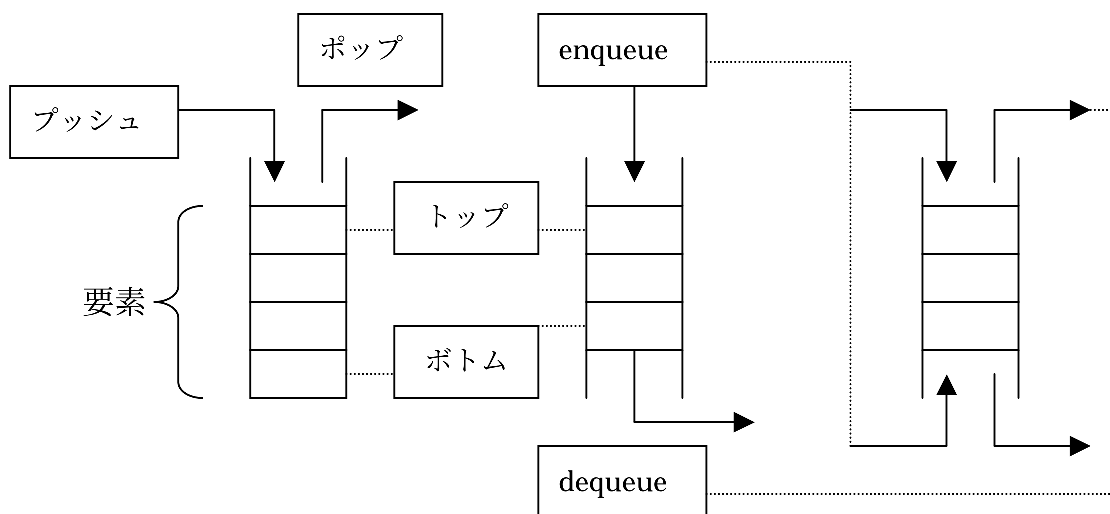



図 2.4.1　左から順にスタック、キュー、デク

### 2.4.4 スタックの操作

実際にプログラムを作成し、スタックを操作する。スタックの操作はリストの先頭に限定されているため、1 件の要素の挿入、削除にかかる時間計算量は

$T(N) = O(1)$

となる。

スタックのプログラムを末稿の“プログラム 4　スタックの操作”に示す。

このプログラムの流れを説明する。

1　スタックに要素を挿入する。・・・キーボードを使い、I“要素”により要素をプッシュする。

2　要素を削除する。　　　　　・・・キーボードより、D“要素”により要素をポップする。

3　実行結果の表示　　　　　　・・・実行結果をプログラムに示す。要素は挿入した順に表示されていくが、削除する時は最後に挿入した要素からでないと削除できない。

### 2.4.5 キューの操作

実際にプログラムを作成し、キューを操作する。キューの操作はリストの先頭と末尾の要素が最初に選ばれる、1 件の要素の挿入、削除にかかる時間計算量は

$T(N) = O(1)$

となる。

キューのプログラムを末稿の“プログラム 5　キューの操作”に示す。

プログラムの流れを説明する。

1　キューに要素を挿入する。・・・キーボードを使い、I“要素”により要素を追加する。

2　キューの要素を削除する。・・・キーボードより、D“要素”により要素を取り出す。

3　実行結果の表示　　　　　　・・・実行結果をプログラムに示す。要素は前の方から順に詰められていき、取り出される時は最初に追加した要素から取り出される。

## 2.5 リストの探索アルゴリズム

これまで述べてきたように、リスト構造は要素が線形に並んでできたデータ構造である。そのため指定した要素を探す時には先頭要素から順に探さなければならない。要素数を $N$ としたとき、探索する要素が存在した場合は、 $N$ に比例した手間が必要であり、存在する場合には最も速くて 1 に、最も遅くて $N$ に比例した手間を必要とする。平均では $N/2$ に比例する。よって指定した要素がリストの中の要素から探索される確立は $1/N$ となるので探索の手間は、

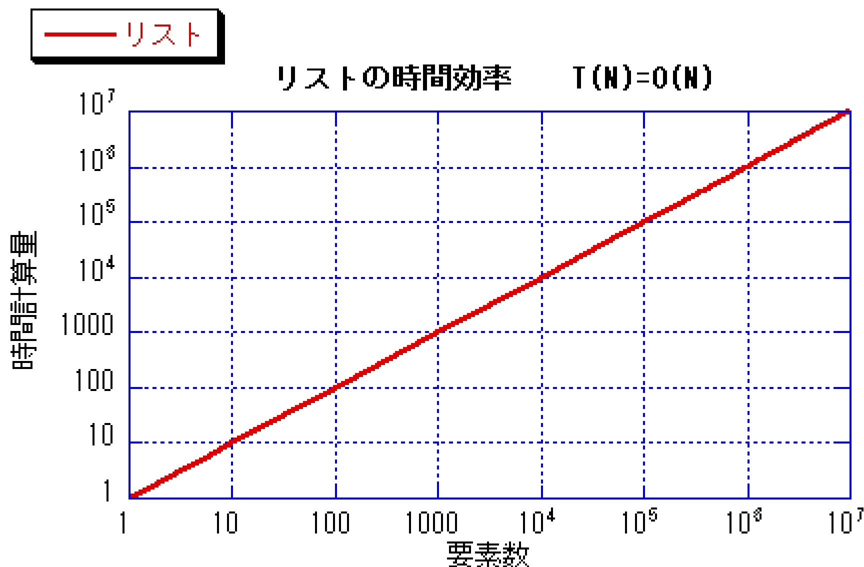
$$\sum_{i=1}^{N} i*1/N = 1/N\sum_{i=1}^{N} i = 1/N*N(N+1)/2 = (N+1)/2$$

となる。これをオーダーを使って表すと

$$T(N) = O(N)$$

となり、要素 $N$ に比例して時間計算量が増加することがわかる。

下にリストの時間計算量の増加をグラフに表すとグラフ 1 のようになる。両軸とも対数で表し、要素数の範囲は 1～$10^7$ で表示する。




グラフ 1　リストの時間増加傾向

片方向リスト、片方向リストの環状リスト、双方向リスト、双方向リストの環状リストの要素の挿入、削除に要する効率をオーダを用いて表 1 に示す。要素の先頭へは全リストとも 1 つ目の要素なので $O(1)$ となり、直後への操作では今ある要素の 1 つ後なので $O(1)$ となる。直前への操作は両方向リストなら手前に戻る事が可能であり $O(1)$ となる。片方向リストの場合は今ある要素の場所から要素数分の移動が必要なので $O(n)$ となる。また、末

尾からの挿入は片方向リストは直前への挿入と同じで$O(n)$となり、双方向リストと環状リストは戻る事が可能なので$O(1)$と示す事ができる。したがってリストの要素の探索、挿入、削除は$O(n)$, $O(1)$で表す事ができる。

| | 片方向リスト | (環状) | 双方向リスト | (環状) |
|---|---|---|---|---|
| 先頭への挿入・削除 | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| 直後への挿入・削除 | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| 直前への挿入・削除 | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ |
| 末尾からの挿入・削除 | $O(n)$ | $O(1)$ | $O(1)$ | $O(1)$ |

表 1 リストの効率

# 第3章　木構造

## 3.1　木構造について

扱うデータ間に階層性や分岐の関係があるときは木(tree)を用いる。データを図に表現した時、それが木の構造に見られる事から木構造と呼ばれる。木構造は多くの分野で見られる自然なデータ構造であり、ファイルのディレクトリの階層的な管理構造や、トーナメント表等、多くの階層構造は木により表現される。

木はグラフ(graph)の特別な場合であり、定義すると以下のようである。

グラフは図 3.1.1 に示すように、

点(vertex,node)の集合とその集合に属する 2 点を結ぶ辺(edge)集合からなる

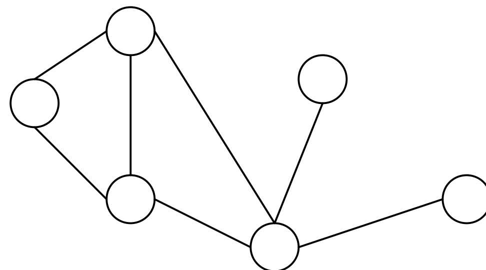


図 3.1.1 グラフ

又、木構造の木というのは、各要素が二つ以上のポインタをもち、枝分かれできるようなデータ構造を示す。木の各要素は点（接点ともいう）と点の間を結ぶポインタ（枝ともいう）からなり、出発点となるただ一つの点のことを根(root)という。

つまり、木構造というのは根となる要素から始まって、枝分かれしながら要素を配置していくものがあり、そのような木構造を根付き木(tooted tree)という。根が最上部にあり、そこから下へ枝分かれしている。各ノードからでる枝の本数が２本以下であるような木を２分木(binary tree)という。これに対し、各ノードからでる枝の本数が３以上であるような木の事を多分木という。木の種類や特性、用途のよって木のデータ表現は何通りか考えられる。

**木の定義**

- 根を頂点に、階層関係にある点を線で結んだもの。
- 各点はラベル（＝値）を持つ。
- 最も上位の点を根、又は親、同じ親を持つ同レベルの点を兄弟、下位の節を葉、親を持つ点を子と呼ぶ。
- 根は必ず一つである。
- 根以外の点は必ず一つの親を持つ。

図 3.1.2 に根付き木の例を示す。

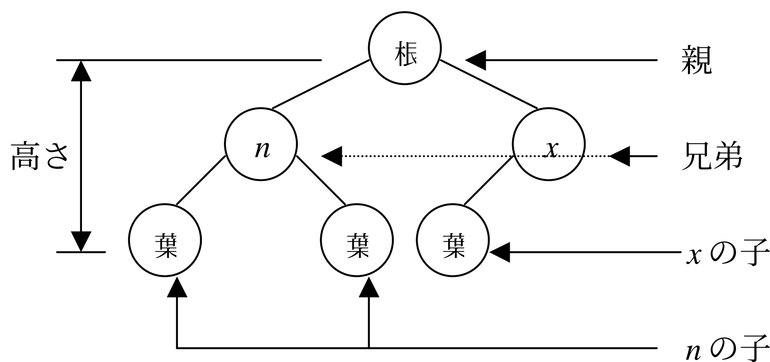



図 3.1.2 根付き木の概念

次に木の順序付けについて説明する。順序付けとは要素の操作を効率よく実行する為、要素に順序をつけるというものである。順序付けには幅優先順(depth first order)と、深さ優線順(breadth first order)がある。

幅優線順　…高さの低い所から、かつ左から順に数えていく(つまり根は必ず 1)。幅優線順はヒープの実現に使われる。

深さ優先順…根から順に左部分木から順番をつけていく。

深さ優先順による順序付けには三つのパターンがあり先行順(preorder)、中間順(inorder)、後行順(postorder)がある。

先行順…根から始まり左部分木、右部分木の順に進んで行く。

中間順…左部分木から始まり根、右部分木の順に進んで行く。

後行順…左部分木から始まり右部分木、根の順に進んで行く。

このように順序付けをされた木には順序木と無順序木があり、

順序木　…兄弟間に順序がある木

無順序木…兄弟間に順序がない木

という。2 分木の順序付けで一定の順序をたどることを、2 分木の走査という。

## 3.2 ヒープ

ヒープ(heap:整列 2 分木)とは、根に近い方から、かつ左の方から点が詰められてできた２分木(このように左の方から点が詰められた２分木のことを完全２分木という)であり、その各点に値をもつ要素が割り当てられているもので、すべての親子の要素間に

“親の要素の値は子の要素の値を超えてはならない”

という関係がみたされている。つまり根の方に向かって要素が小さくなっていく。ただし、兄弟の要素同士の大小関係には制約がない。このことから根の値が最小要素ということになる。

又、根は node 1(ノード 1)であり、配列の 1 番地に対応し、i の親は node[i/2]であり、

“ node i の左の子は node 2i、右の子は node 2i+1 となる。”
という関係が成立する。

　このようなデータ構造のことを半順序木という。ヒープは点の間の要素の複写が操作の中心であり、点の追加・削除も木の最後で行われる(このような操作を幅優線順という)ことにより木の構造が大きく変更されることがない。これにより 1 次元配列で実現できることができる。図にヒープの例(図 3.2.1)、ヒープにおける添字(図 3.2.2)、ヒープの配列による実現例(図 3.2.3)を示す。

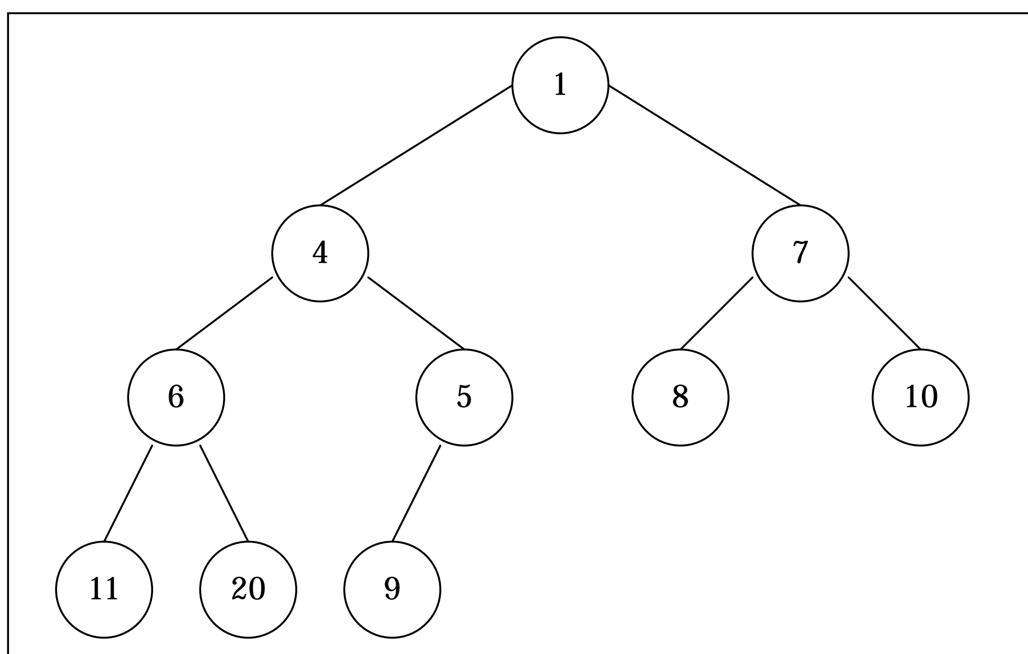



図　3.2.1　ヒープの例

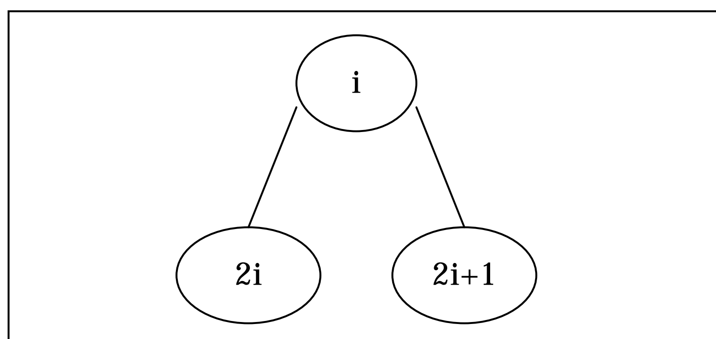



図　3.2.2　ヒープにおける添字

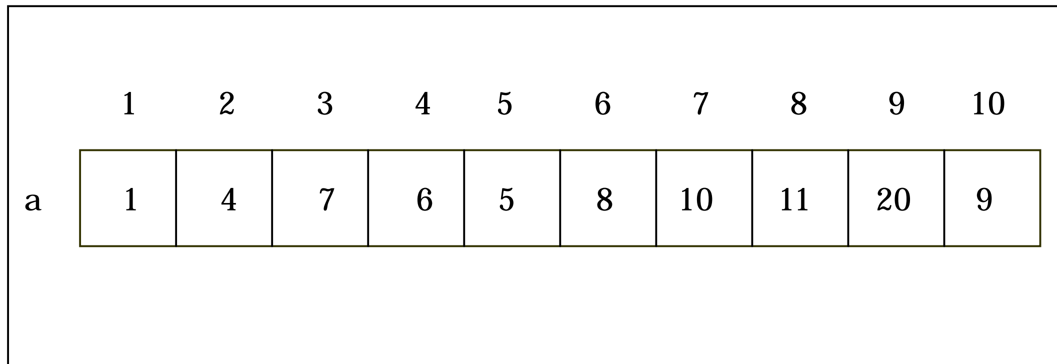



図 3.2.3 ヒープの配列による実現

### 3.2.1 ヒープの操作

ヒープを使う事により、最小要素の参照、要素の挿入と最小要素の削除ができる。又、集合のヒープ化がある。これらの操作の説明をしていく。ヒープに含まれる要素数を示すには SIZE という変数を使う。

#### 3.2.1.1 最小要素の探索

最小要素はヒープの性質
“ 親の要素の値は子の要素の値を超えない ”
ということからヒープの根が必ず最小要素となる。よって最小要素の参照は
$T(N) = O(1)$
でできる。

#### 3.2.1.2 要素の挿入

要素の挿入は、まずヒープの最後(幅優線順で)に点を追加し要素を挿入する。そして、上方移動の操作(shift up)により、再度 N+1 点からなる完全 2 分木全体をヒープの性質が満足するように点を適切な位置まで移動させる。要素の挿入のプログラムとその実行結果を“ プログラム 6　ヒープへの要素の挿入 ” に示す。

**・上方移動**

node n の親 p(n)=[n/2]との間でヒープ条件(HEAP[p(n)]<=HEAP[n])が成立しないうちは(HEAP[n]<HEAP[p(n)]であるので)、node n の値 HEAP[n]と node p(n)の値 HEAP[p(n)]と交換し、n:=p(n)として以下これを繰り返す。
したがって、最終的には n は根に到達するかヒープ条件を満たすようになる。

#### 3.2.1.3 要素の削除

要素の削除は、ヒープ条件により根は常に最小要素を持っていることがわかる。その値を読み取り、ヒープの最後の点を除き、その要素を親の要素に移動する。そして、下方移動の操作(shift down)により、再度 N-1 点からなる完全 2 分木全体をヒープの性質が満足するように点を適切な位置まで移動させる。要素の削除のプログラムとその実行結果を“ プログラム 7　ヒープの最小要素の削除 ”に示す。

- **下方移動**

  node u を node v に付随する 2 つの子のうちの値の小さい方とする。v=p(n)と node u との間でヒープ条件(HEAP[v]<=HEAP[u])が成立しないうちは (HEAP[u]<HEAP[p(u)]であるので)、node v の値 HEAP[v]と node u の値 HEAP[u] とを交換し、v=u としてこれを繰り返す。

これらのヒープの要素の挿入、削除はヒープの高さに比例する時間でできる。ヒープの深さは、ヒープに含まれる要素数 SIZE を n とおけば、

$$\log_2(n+1)-1$$

となりこれをオーダで表すと

$$O(\log_2 n)$$

となることがわかる。

実際、ヒープの深さを $h$ とすれば、任意の $h' < h$ に対して深さ $h'$ のノード数は $2^{h'}$ であり、深さ $h$ のノード数は 1 以上 $2^h$ 以下であるので、

$$2^h - 1 = \sum_{i=0}^{h-1} 2^i < n \le \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

となり、$h = \log_2(n+1) - 1$　が得られる。これをオーダで表すと

$$h = O(\log_2 n)$$

となり、オーダを使うとヒープに限られず、2 分木の深さは $O(\log_2 n)$ となる。

#### 3.2.1.4 集合のヒープ化

以上の事から、与えられたデータ集合をヒープ化するには 2 通りあることがわかる。ヒープ化というのは与えられたデータ集合をヒープの条件を満足するような根付き木にすることである。

1・・・空なヒープから始め集合の要素を一つずつ挿入しヒープ条件を満たすようヒープ化を行う上方移動。

2・・・集合の全ての要素を完全 2 分木の各点に割り当て、最後の要素から各点にヒープ化する下方移動。

の 2 通りである。

上方移動も下方移動もヒープの高さに比例した手間が必要である。多くの上方移動が大きなヒープに対して適用され、逆に多くの下方移動は小さなヒープに対して適用されている。

では次に、両方の計算量を求めてみてどちらが早くヒープ化できるかを示す。ヒープの高さを$h$、全体の手間を$T(N)$とする。
上方移動では：

$$\begin{aligned} T(N) &\propto \sum_{i=1}^{N} (\log i + 1) \\ &= \log(N!) + N \\ &= O(N \log_2 N) \end{aligned}$$

下方移動では：

$$\begin{aligned} T(N) &\propto \sum_{i=0}^{h} 2^i (h+1-i) \\ &= (h+1)(2^{h+1}-1) - h2^{h+1} - 2 \\ &\leq 2^{h+2} \end{aligned}$$

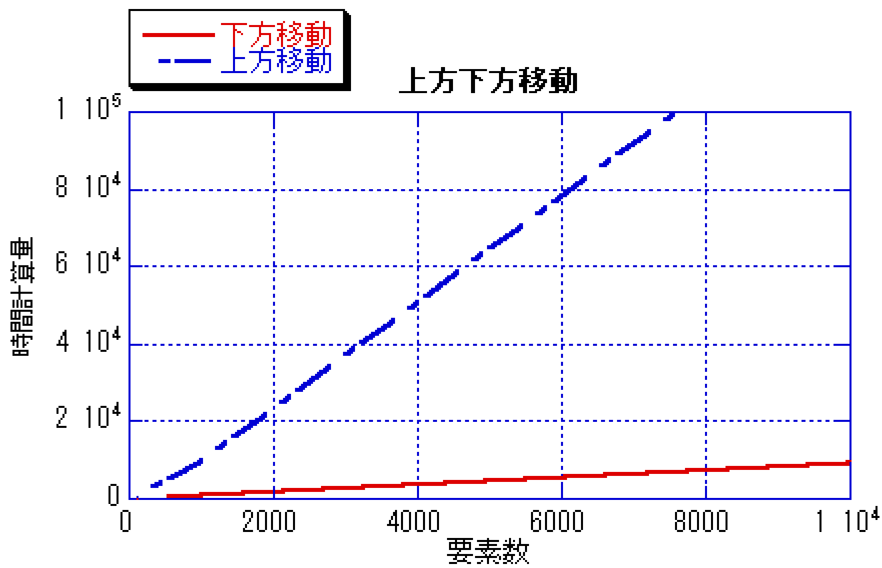
ここで、$h \leq \log N$であるので、

$$= O(N)$$

ここで$N$に値を入れ、どちらが効率が良いのかを表 3.1 とグラフ 3.1 に示し比較してみる。

| アルゴリズム / 要素数 | 上方移動 $O(N \log_2 N)$ | 下方移動 $O(N)$ |
|---|---|---|
| 10 | 33 | 10 |
| 100 | 664 | 100 |
| 1000 | 9966 | 1000 |
| 10000 | 130000 | 10000 |
| 100000 | 1660000 | 100000 |
| 1000000 | 20000000 | 1000000 |

表 3.1　上方移動、下方移動の計算量




グラフ 3.1　上方移動と下方移動の計算量

グラフを見てわかる通り、明らかに計算時間が下方移動の方が少なくてすむ。
よって集合のヒープ化を行うのは下方移動の方が処理が早い。

### 3.3　2分探索木

探索木(search tree)とは木の点に集中の要素を割り当てたものであり、最小要素の判定(find min)、挿入、削除を効率よく実行するためのものである。
基本的な探索木としては、２分探索木、２－３木、AVL 木などがあるが、本稿では最も基本的とされている２分探索木について取り上げる。
２分探索木は２分木の点に集合の要素を以下のような性質をもつように割り当てたものである。
“ある節点の要素を x とするとき、その左部分木の点はすべて x より小さく、右部分木の点はすべて x より大きい (図 3.3.1 参照) ”

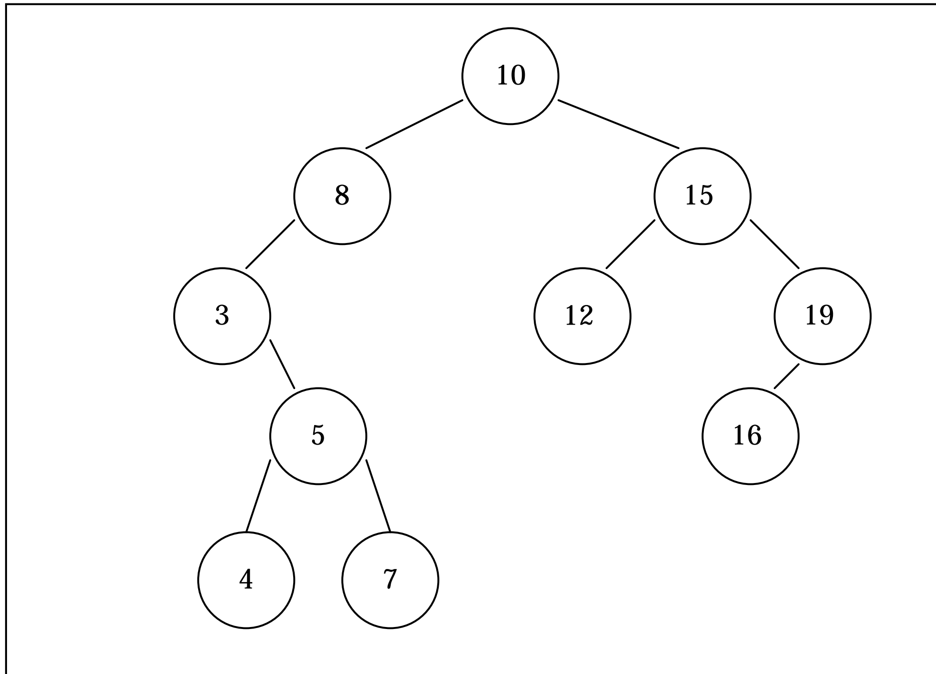



図 3.3.1　2 分探索木の例

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| FATHER | 0 | 1 | 1 | 2 | 3 | 3 | 4 | 6 | 7 | 7 |
| LEFT | 2 | 4 | 5 | 0 | 0 | 8 | 9 | 0 | 0 | 0 |
| RIGHT | 3 | 0 | 6 | 7 | 0 | 0 | 10 | 0 | 0 | 0 |
| VALUE | 10 | 8 | 15 | 3 | 12 | 19 | 5 | 16 | 4 | 7 |

図 3.3.2　2 分探索木

以上のような性質をもつように割り当てられたものである。この方法の要点は、検索データを x とするとき、ファイル・データの中央のものと x を比べれば、探すもの x はファイルの前半にあるか、後半にあるかがわかることになる。すなわち探索範囲の中央のデータとの 1 回の比較により、次の探索範囲を半分に縮めることができる。

### 3.3.1　2 探索木の操作

2 分探索木のプログラムを作成し最小要素の探索、要素の挿入、削除を行う。今回は 2 分探索木はすでに完成しているものとしてプログラムを操作する。要素の操作には、図 3.3.1 の 2 分探索木を用い、図 3.3.2 のようにリスト構造によって実現した。根を 10 とおく。

#### 3.3.1.1　最小要素の探索

2 分探索木の中の最小要素を見つけるものである。最小要素は 2 分探索木の条件によって、最小要素は最も左の要素である。言い換えると、

“ 最初に見つかる左の子をもたない点 ”

が最小要素となる。つまり左部分木の左へのポインタが格納されている配列 LEFT が最初に 0 になった要素を探せばよい。逆に最大要素の探索は、根から順に配列 RIGHT が最初に 0 になった要素を探せばよい。

“ プログラム 8　2 分探索木の最小要素の探索 ” に最小要素の探索プログラムを示す。

最大要素の探索は findmin の LEFT を RIGHT に置き換えればよい。

#### 3.3.1.2 要素の挿入

2 分探索木の中に要素の挿入を実行するものである。挿入した後も 2 分探索木の性質を満たさなければならない。要素の挿入は挿入する位置を探索しなければならないが、挿入位置さえわかればポインタをつなぎかえるだけで挿入完了となるので手間は

$T(N) = O(\log_2 N)$

で示される。“ プログラム 9　2 分探索木の要素の挿入 ” にこのプログラムを示す。

#### 3.3.1.3 要素の削除

要素の削除は 2 分探索木から指定した要素を削除し、削除した後も 2 分探索木の性質を満足していなければならない。削除の場合では削除する要素の位置により 3 つのパターンを考え要素を削除しなければならない。

削除する要素を $n$ とする。

・$n$ が葉である場合

・$n$ が 1 つの子を持つ場合

・$n$ が 2 つの子を持つ場合

2 分探索木で要素を削除も 2 分探索でたどられる経路に比例した手間で実行できる。よって手間は

$T(N) = (\log_2 N)$

となる。“ プログラム 10　2 分探索木の要素の削除 ” に 2 分探索木の要素の削除のプログラムを示す。

### 3.4　2 分探索木の探索アルゴリズム

2 分探索木の時間計算量の増加傾向を求める。

探索に要する時間計算量はループの反復回数を求めることによりわかる。以下に探索に要する最大時間計算量と平均時間計算量を求める。

最大時間計算量

ループの反復回数を $h$ とし要素数を $N$ とおく。

ループの反復回数 $h$ に対し、$h$ 回目の反復で探索される要素数は $2^{h-1}$ であるから $h$ 回の反復で探索できる要素数 $N$ は

$$N = \sum_{i=1}^{h} 2^{i-1} = 2^h - 1$$

である。つまり、探索の対象である要素数全体が $N$ であれば $h$ 回の反復で必ず探索ができることになる。それゆえ

$$h = \log_2(N+1)$$

となる。又

$$2^{h-1} - 1 < N \leq 2^h - 1$$

の時も $h$ 回の反復で探索ができる。したがって探索に必要な最大反復回数 $h$ は要素数 $N$ に対して

$$h = \log_2(N+1)$$

となる。ゆえに最大時間計算量をオーダで表すと $O(\log_2 N)$ となる。

平均時間計算量

平均反復回数を $E(N)$ とおき探索対象となる確立を等しいとして $N = 2^h - 1$ とする。この時、探索に必要な平均平均反復回数 $E(N)$ は

$$E(N) = \sum_{j=1}^{k} jP(j) = \sum j * 2^{j-1} / 2$$

となる。$P(j)$ は $j$ 回反復するときの確立である。

$$R = \sum_{j=1}^{k} j * 2^{j-1}$$

とおき、

$$R - 2R = 1 + 2^1 + 2^2 + \cdots + 2^{k-1} - k * 2^k$$

に注意すれば、

$$R = 1 - 2^k + k * 2^k$$

を得ることができる。ゆえに、

$$E(N) = R / N = 1 - 2^k + k * 2^k / 2^k - 1 = k - 1 + k / 2^k - 1$$

また、$2^{k-1}-1<N\le 2^k-1$のとき、次のようになる。

・$k$が十分大きい時、

$E(2^{k-1}-1)<E(N)\le E(2^k-1)$

よって

$k-2+k-1/2^{k-1}-1<E(N)\le k-1+k/2^k-1$

より

$k-2<E(N)<k$

といえる。さらにこのときの必要な最大反復回数は$k$であるから、

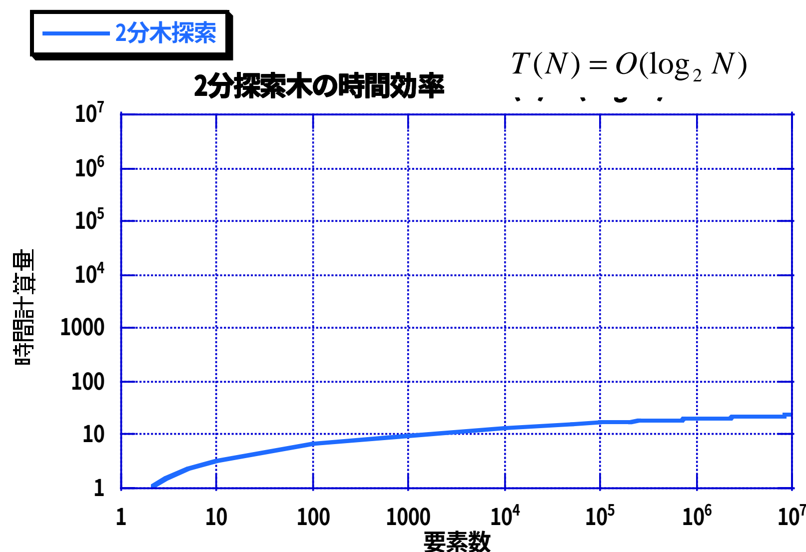
$k=\log_2(N+1)$

となる。よって、

$\log_2(N+1)-2<E(N)<\log_2(N+1)$

したがって、平均時間計算量は$O(\log_2 N)$となる。

　下にリストの時間計算量の増加を表すグラフ3を載せる。両軸とも対数で表し、要素数の範囲は 1～$10^7$ で表示する。




グラフ3　2分探索木の時間効率

# 第4章　データ効率の比較

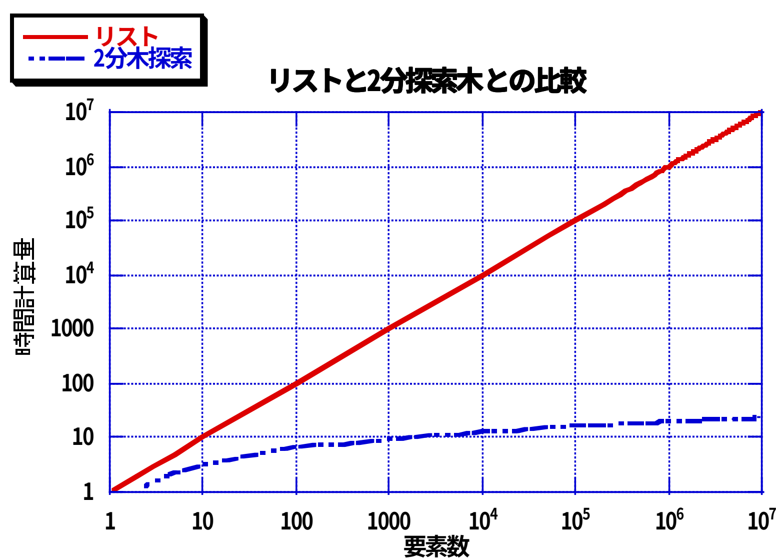
では、リストと 2 分探索木の時間計算量を、グラフで比べてみる。

リスト、2 分探索木の時間計算量はこれまでと同じで

リスト　　　　　　$T(N) = O(N)$

2 分探索木　　　　$T(N) = O(\log_2 N)$

で、それぞれ要素に値を入れて増加傾向を調べる。グラフ 4.1 に比較したグラフを示す。




グラフ 4.1　リストと 2 分探索木の比較

リストはデータ容量と処理の手間が比例しているので、データ容量が大きくなるにしたがって比例して処理の手間も大きくなっている。それに対して 2 分探索木の方はデータ容量が大きくなっても、それほど時間計算量は増加していない。要素数が少ない時にはあまり差がないが、要素数が増えるにつれ差ははっきりとわかるようになる。リストは要素数が 2 乗になると時間計算量も 2 乗するのに対し 2 分探索木の方は要素数が 2 乗になっても時間計算量は 2 倍にしかなっていない。

そのため、オーダを使用した時、探索にかかる手間は要素数が多い時は 2 分探索木を使うほうが効率が良いといえる。次に実際にプログラムを作成し、リスト、2 分探索木のデータの処理時間を調べてみる。これはそれぞれの探索アルゴリズムを UNIX の time コマンドを使い、実際の処理時間を調べる。

time コマンドとは、指定したコマンドの実行、および実行中に経過した時間、システム

例で処理に要した時間、コマンド側で実行に要した時間を計測し、これを出力するものである。

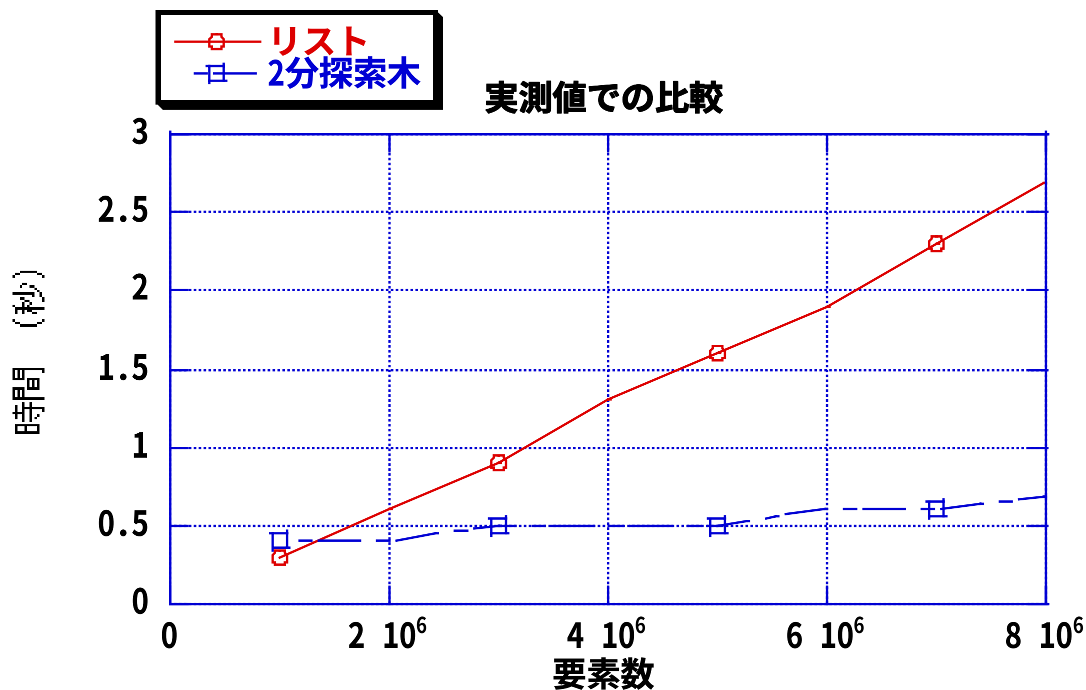
% time　コマンド名

1.6u 0.9s 0:07 34% 0+0k 0+0io 0pf+0w

- 0:07 ─ 経過時間
- 0.9s ─ システム使用の CPU 時間
- 1.6u ─ ユーザ使用の CPU 時間

今回調べたのは左端にあるユーザ使用の CPU 時間である。

time コマンドを用いると、表示する時間は$10^{-1}$秒まで測定できない。そのため要素数が少ない場合では表示が 0.0 と表される。よって、今回は要素数の測定がわかる$10^6$から測定を開始する。又、要素数が$9*10^6$以上になると" core dumped"と表示されてエラーが起こる。そのため今回の測定は要素の測定のできる範囲$1*10^6$〜$8*10^6$の測定を行った
実行結果をグラフ 4.2 に示す。




グラフ 4.2　探索アルゴリズムの実測値

リストの探索アルゴリズム

要素数が$1*10^6$から測定時間を求めることができるようになるが、それと同時に要素数

の増加に伴い、処理に時間がかかってくるのがわかる。要素数が$6*10^6$くらいになると処理に2秒近くかかってくる。$1*10^6$の時の処理時間は0.3秒であり、$6*10^6$の場合では1.9秒かかっている。これにより処理計算の傾きがわかる事ができる。

$T(N) = 1.9 - 0.3 / 6*10^6 - 1*10^6$

より傾きは

$T(N) = (3.2*10^{-7})N$

となる。この傾きで進んでいくと要素数が$1*10^{10}$等の時は処理時間が$3.2*10^3$秒かかることになり、処理時間の遅さを感じてしまう。したがって要素数が多い場合、リスト構造では時間がかかるので適したデータ構造とはいえない。

2分探索木の探索アルゴリズム

一方、2分探索木のほうは要素数が$1*10^6$の時は処理時間が0.4秒とリスト構造とあまり変わらないが、要素数が$6*10^6$になっても処理時間は0.7秒とあまり変わっていない。

両者の要素数が$1.3*10^6$の時、探索にかかる時間は同じで0.4秒となっている。要素数が$1.3*10^6$以下ではリストの方が効率が良くなっているが探索にかかる時間は0.4秒以下となっており、処理にかかる時間を感じないくらいである。しかし、リストの要素数が$6*10^6$の場合と2分探索木の要素数が$6*10^6$の場合の処理の差は1.3秒となっており、さらに$8*10^{10}$の場合では両者にかかる処理の差が2秒となっており我々にも処理の効率の差を感じられる程である。

よって要素数が多い場合では、2分探索木等のデータ構造を用いた方が効率が良い。

# 第5章　結論

本稿の目的は最初に述べたように基本的データ構造を用いた探索法(リスト構造と木構造)について述べ、それぞれのデータ構造の違いによるアルゴリズムの違い、時間計算量の違いについて評価をし、データ処理に最も適しているのはどれか、というものであった。

第 5 章までの以上の考察から、データ処理に適した探索アルゴリズムは何か、ということを今回の結論として述べる。

今回データ処理を行った探索アルゴリズムはリストと木構造であった。それぞれの処理の結果を第 4 章で述べている。結果、データ量が多い時は木構造、少ない時はリストを用いる方が効率が良い、が、今後先、要素数が増えるということを考えると要素数が多くても処理の効率がよい木構造を使用するほうがよい。

第 2、3 章にそれぞれオーダで評価した場合の増加傾向を比較しているが、グラフを見る限りでは要素数が少ない時も 2 分探索木の方が計算時間が早かったが、実戦値では要素数が少ない場合ではリストの方が処理がわずかではあるが早かった。オーダによる計算は定数係数を考えないというものであり、要素数が少ない場合では定数係数も考えなければならないというのがわかった。

# 第6章　まとめ

## 1.　配列

　配列で変数やデータを準備すれば、コンピュータの最も得意な処理である繰り返し処理にもちこむことができ、全体の処理がわかりやすく、かつ拡張性のある表現となる。配列の特徴は蓄えられたデータがメモリ上に並んでいるということであり、その為、新しいデータを配列の途中に挿入する時や途中のデータが不要になり削除したい時には、多くのデータの移動が生じてしまう。すなわち配列に蓄えるデータは、途中にデータの追加削除が生じないものが望ましい。

## 2.　リスト

　リストは対象となるデータが処理の途中に新しいデータが発生したり、あるいは削除されたりする場合、すなわちデータが動的に発生消滅する場合に、それに対処する為のデータ構造である。リストは複数の項目を持つデータリストを作るのに便利であり、リストのアルゴリズムは以降の木構造でも用いられる。リストの表現は

・要素を 1 次元配列に連続的に格納する方法
・要素とそれに隣接する　セル(cell)　の位置を指す　ポインタ(pointer)　の組によるリンク構造

の 2 通りがあり後者のリンク構造で実現されたリストを用いる事が多い。時間計算量は要素数に比例して増加する。

　$T(N) = O(N)$

## 3.　木構造

　木構造は家系図や、データのディレクトリの階層的な管理構造等に用いられる。木は点の集合とその集合に属する 2 点間を結んだ辺の集合からなる。又、根と呼ばれる特別な点を持つ木を根付き木という。根は木の頂上にあり根以外の点で次数が 1 の点を葉という。根から最も遠い葉までの距離を高さという。各点が 2 個の子を持つような根付き木の事を 2 分木といい、高さの浅い方から、かつ左の方から点がつめられている 2 分木の事を完全 2 分木という。完全 2 分木の分析方法にヒープ、2 分探索法がある。

　完全 2 分木では、高さ h までに登録できる要素数 n は、

$n = 2^{h+1} - 1$

と表現でき、

$h = \log_2(n+1) - 1$

となり、要素数 n の完全 2 分木の高さは $O(\log_2 n)$ となる。

## 4. ヒープ

ヒープとは

・親の要素の値は子の要素の値よりも小さくなければならない。

・子同士の値、つまり兄弟間の要素の大小は意味を持たない。

・要素は根に近い方から、かつ左の方から詰められてできていなければならない。

というヒープ条件を満たす完全 2 分木である。ヒープの要素の挿入、削除、ヒープ化という操作ができる。ヒープの操作は親から子へと、又は子から親へと要素の移動を繰り返しヒープ化を行う上方移動と下方移動があり、

上方移動にかかる手間　$O(N \log N)$

下方移動にかかる手間　$O(N)$

ということにより下方移動の方が効率の良いデータ処理である。

## 5. 2 分探索木

2 分探索木は 2 分木の探索法の中でも基礎的なものであり、ヒープと同じ完全 2 分木である。同じような 2 分木の構造をしているが 2 分探索木は

・すべての点において、その左部分木はすべて親より小さく、右部分木はすべて親より大きい。

という条件を満たさなければならない。

リストよりもデータ処理の時間効率が良く、時間計算量は

$T(N) = O(\log_2 N)$

となる。

# 第7章　謝辞

今回の論文を書くにあたり、指導教員である山本哲也教授には色々とお騒がせいたしましたが最後まで丁寧な御指導、御鞭撻をいただいた事に誠に感謝致します。

又、本研究を進めるにあたり、M1 の浜小路欣大さんには日々励ましの言葉を頂きました。ありがとうございました。

そして私と同じ研究室の研究員が卒研を制作している姿がなければ私はここにいなかったでしょう。ありがとうございます。

最後に、改めてここに記すことで謝辞とさせて頂きます。

# 第8章　参考文献


- C アルゴリズム全科

著者　千葉　則茂、村岡　一信、小沢　一文、海野　啓明
1995 年　近代科学社

- アルゴリズムの基礎

著者　大森　克文、木村　晴彦、広瀬　貞樹
1997 年　共立出版

- C で学ぶデータ構造とアルゴリズム

著者　杉山　行浩
1995 年　東京電機大学出版局

- 情報の構造　上

著者　浅野　考夫
1994 年　日本評論社

- データ構造とアルゴリズムシリーズ 3

著者　浪平　博人
1995 年　CQ 出版株式会社

## <付録>プログラムリスト

**<プログラム 1 リストの要素探索>　　“ C アルゴリズム全科　P 70 引用 ”**

```
#define SIZE 100                                  /*セルの個数*/

void main()
{
   int VALUE[SIZE+1];                             /*セル領域の宣言*/
   int NEXT[SIZE+1];
   int head,                                      /*リストの先頭セルを指すポインタ*/
         tail;                                    /*リストの末尾を指すポインタ*/
   int newcell,i,p,q,x;

newcell=1;
for(i=1;i<SIZE;++i)NEXT[i]=i+1;
NEXT[SIZE]=0;

head=0;
tail=newsell;
printf("入力・・・");scanf("%d",&x);              /*要素の入力*/
while(x!=999&&newcell!=0){                        /*要素の終了は x=999 で示す*/
   p=newcell;newcell=NEXT[p];                     /*セルの用意*/
   VALUE[p]=x;                                    /*値の書き込み*/
   NEXT[p]=head;head=p;                           /*先頭セルとして挿入*/
   printf("入力・・・");scanf("%d",&x);           /*次の要素の入力*/
}
   if(x!=999&&newcell==0){                        /*領域を越える場合エラー表示*/
   printf("Error:memory resort¥n");exit(1);
}
printf("要素・・・");scanf("%d",&x);

q=0;                                              /*要素が存在していないとする*/
p=head;
while(p!=0){
   if(VALUE[p]==x){q=p;p=0;}                      /*要素が存在、ループの停止*/
```

```
   else                    p=NEXT[p];
}
printf("結果・・・%d¥n",q);
}
```

<実行結果>

・　要素が存在する場合

入力・・・1
入力・・・2
入力・・・3
入力・・・4
入力・・・999
要素・・・3
結果・・・3

・　要素が存在しない場合

入力・・・1
入力・・・2
入力・・・3
入力・・・4
入力・・・999
要素・・・6
結果・・・0

・　要素が 101 以上ある場合

入力・・・1
入力・・・2
　　・
　　・
入力・・・100
入力・・・101
Error:memory resort

<プログラム２　リストの要素挿入>　　“ C アルゴリズム全科　P 74 引用 ”

```
#define SIZE 10

void main(void);
void insert(int q,int x);
void writelist(void);
     /*セル領域              1,2,3,4,5, 6, 7,8, 9*/
int VALUE[SIZE+1]={0,0,3,1,8,23,0,6,63,0};
int NEXT[SIZE+1]={0,9,4,8,5,3,1,2,0,17};
int head,tail,newcell;

void main(void)
{
int q,x;
head=7;newcell=6;
writelist();
  x=11;                                  /*要素の入力*/
  q=5;                                   /*直前セルの指定*/
insert(q,x);                             /*挿入*/
writelist();                             /*リストの表示*/
}

void insert(int q,int x)                 /*要素を q の指すセルの直後に挿入*/
{
int p;

p=newcell;newcell=NEXT[p];
VALUE[p]=x;
  if(q==0){                              /*先頭セルとして挿入するときは q=0*/
  NEXT[p]=head;head=p;}
  else{                                  /*先頭以外のセルとして挿入*/
  NEXT[p]=NEXT[q];NEXT[q]=p;
  }
}
```

```
void writelist(void)                    /*リストの表示*/
{
int p;

   p=head;
   while(p!=0){
   printf("%6d",VALUE[p]);
   p=NEXT[p];
}
printf("¥n");
}
```

<実行結果>

```
     6     3     8    23     1    63
     6     3     8    23    11     1    63
```

<プログラム 3 リスト要素削除> “ C アルゴリズム全科 P 77 引用 ”

```
#define SIZE 10

void main(void);
void delete(int q);
void writelist(void);
/*[実行例]用リスト 1,2,3,4, 5, 6, 7,8,9*/
int VALUE[SIZE+1]={0,0,3,1,8,23,12,6,63,0};
int NEXT[SIZE+1]={0,9,4,8,5,6,3,2,0,17};
int head,
tail;
int newcell;

void main(void)
{
   int q;

   head=7;
   newcell=1;
   writelist();                          /*リストの表示*/
   q=6;                                  /*セル番号 6 を削除*/
   delete(q);
   writelist();
}
void delete(int q)                       /*要素の削除*/
{
int p,s;

   if(head==q)head=NEXT[q];              /*直前セルの探索*/
else{
   p=head;                               /*削除するセルが先頭以外*/
   while(p!=0){
   if(NEXT[p]==q){s=p;p=0;}
else        p=NEXT[p];
}
```

```
    NEXT[s]=NEXT[q];                                  /*要素の削除*/
}
    NEXT[q]=newcell;newcell=q;                        /*未使用セルへの変換*/
}
void writelist(void)                                  /*リストの表示*/
{
int p;

    p=head;
    while(p!=0){
    printf("%6d",VALUE[p]);
    p=NEXT[p];
}
printf("¥n");
}
```

<実行結果>

```
     6     3     8    23    12     1    63
     6     3     8    23     1    63
```

<プログラム 4　スタックの操作> " **C で学ぶデータ構造とアルゴリズム　P 27 引用** "

```
#include <stdio.h>
#include <stdlib.h>
#define STACK_SIZE 8

typedef int KEY;
typedef int DATA;
typedef struct{                          /*リスト構造をセル型として宣言*/
   KEY key;
   DATA data;
} CELL;
static CELL stack[STACK_SIZE];
static CELL *sp;
/*初期化関数*/
void init(void)
{
   sp = stack + STACK_SIZE;
}
/*プッシュ関数*/
void push(CELL cell)
{
if(sp<=stack){
   fprintf(stderr,"スタックオーバーフローです。¥n");
   exit(EXIT_FAILURE);
}
*--sp = cell;
}
CELL pop(void)
{
if(sp>=stack+STACK_SIZE){
   fprintf(stderr,"スタックが空です。¥n");
   exit(EXIT_FAILURE);
}
return*sp++;
}
```

```
/*表示関数*/
void disp(void)
{
int i;
   for (i=0;i<STACK_SIZE;i++)
   if (i>=sp-stack)
      printf("%d",stack[i].key);
      else
   printf("--");
putchar('¥n');
}
/*対話的に動作確認をする main 関数*/
main()
{
   char buff[8];
   KEY key;
   CELL cell,result;

init();
while(1){
   printf("In:プッシュ,D:ポップ?");
   gets(buff);
   key=atoi(&buff[1]);
   switch(buff[0]){
      case'I':case'i':
      cell.key=key;
      push(cell);
      printf("%d をプッシュしました。¥n",key);
      break;
   case'D':case'd':
   result=pop();
   printf("%d がポップされました。¥n",result.key);
   break;
default:
return 0;
```

```
}
disp();
   }
}
```

<実行結果>

```
In:プッシュ,D:ポップ?I 10
10 をプッシュしました。
--------------10
In:プッシュ,D:ポップ?I 20
20 をプッシュしました。
------------2010
In:プッシュ,D:ポップ?I 30
30 をプッシュしました。
----------302010
In:プッシュ,D:ポップ?I 40
40 をプッシュしました。
--------40302010
In:プッシュ,D:ポップ?D 20
40 がポップされました。
----------302010
In:プッシュ,D:ポップ?D 30
30 がポップされました。
------------2010
In:プッシュ,D:ポップ?q
```

<プログラム 5　キューの操作> “ **C で学ぶデータ構造とアルゴリズム　P 30 引用** ”

```
include <stdio.h>
#include <stdlib.h>
#define QUEUE_SIZE 8

typedef int KEY;
typedef int DATA;
typedef struct{                    /*リスト構造をセル型として宣言*/
   KEY key;
   DATA data;
}CELL;
static CELL queue[QUEUE_SIZE];
static CELL *front,*rear;
/*初期化関数*/
void init(void)
{
   front=queue;
   rear=queue;
}
/*追加関数*/
void enqueue(CELL cell)
{
CELL *next=rear+1;

   if(next==queue+QUEUE_SIZE)
   next=queue;
   if(next==front){
   fprintf(stderr,"キューがいっぱいです。¥n");
   exit(EXIT_FAILURE);
}
*rear=cell;                          /*リストの末尾に追加*/
rear=next;
}
/*取り出し関数*/
CELL dequeue(void)
```

```
{
CELL retv;

   if(front==rear){
   fprintf(stderr,"キューが空です。¥n");
   exit(EXIT_FAILURE);
}
retv=*front++;                         /*リストの先頭から取りだす*/
   if(front==queue+QUEUE_SIZE)
      front=queue;
   return retv;
}
/*表示関数*/
void disp(void)
{
CELL *p;

for(p=queue;p<queue+QUEUE_SIZE;p++)
   if(front<=rear)
   if(p>=front && p<rear)
   printf("%d",p->key);
else
   printf("-- ");
else
   if(p>=front || p<rear)
   printf("%d",p->key);
else
   printf("-- ");
   putchar('¥n');
}
/*対話的に動作確認をする main 関数*/
main()
{
char buff[8];
   KEY key;
   CELL cell,result;
```

```
init();
while( 1 ){
   printf("In:追加,D:取り出し?");
   gets(buff);
   key=atoi(&buff[1]);
   switch( buff[0] ){
      case'I':   case'i':
      cell.key=key;
      enqueue(cell);
      printf("%d を追加しました。¥n",key);
      break;
   case'D':case'd':
   result=dequeue();
   printf("%d が取り出されました。¥n",result.key);
   break;
default:
return 0;
}
disp();
}
}
```

<実行結果>

```
In:追加,D:取り出し?I 10
10 を追加しました。
10-- -- -- -- -- -- --
In:追加,D:取り出し?I 20
20 を追加しました。
1020-- -- -- -- -- --
In:追加,D:取り出し?I 30
30 を追加しました。
102030-- -- -- -- --
In:追加,D:取り出し?I 40
40 を追加しました。
10203040-- -- -- --
```

In:追加,D:取り出し?D 10
10 が取り出されました。
-- 203040-- -- -- --
In:追加,D:取り出し?D40
20 が取り出されました。
-- -- 3040-- -- -- --
In:追加,D:取り出し?q

**<プログラム 6　ヒープへの要素の挿入>" C アルゴリズム全科　P 93 引用 "**

```
#include<math.h>

#define HSIZE 100

void main(void);
void insert(int e);
void writeheap();
int pow2(int n);

int HEAP[HSIZE+1]={0,2,9,6,12,10,35,20,30,15,11};
int n={10};

void main(void)
{
   int e;

   e=5;
   insert(e);
}

void insert(int e)                              /*要素の挿入*/
{
int i,i2,w;

   writeheap();
   ++n;HEAP[n]=e;                               /*挿入すべき要素 e を 2 分木の*/
                                                /*最後の要素として追加       */
writeheap();
i=n;i2=i/2;                                     /*点 i2 は点 I の親*/
while(i>1 && HEAP[i2]>HEAP[i]){
w=HEAP[i2];HEAP[i2]=HEAP[i];HEAP[i]=w;
printf("要素%d と要素%d を交換¥n",HEAP[i],HEAP[i2]);
writeheap();
i=i2;i2=i/2;
```

```
    }
}
/*ヒープの表示*/
void writeheap()
{
int depth,i,j,endofprint;

if(n!=0){
    depth=(n);
    for(i=0;i<=depth;++i){
if(n>pow2(i+1)-1)endofprint=pow2(i+1)-1;
else                    endofprint=n;
for(j=pow2(i);j<=endofprint;++j)
    printf("%6d",HEAP[j]);                          /*深さ一定の点の表示*/
    printf("¥n");
    }
}
printf("¥n");
}

int pow2(int n)                                       /*2 の n 乗を求める関数*/
{
    if(n>0)return(2<<(n-1));
    else    return(1);
}
```

<実行結果>

```
     2
     9     6
    12    10    35    20
    30    15    11

     2
     9     6
    12    10    35    20
    30    15    11     5
```

要素 10 と要素 5 を交換

|  |  |  |  |
|---|---|---|---|
| 2 |  |  |  |
| 9 | 6 |  |  |
| 12 | 5 | 35 | 20 |
| 30 | 15 | 11 | 10 |

要素 9 と要素 5 を交換

|  |  |  |  |
|---|---|---|---|
| 2 |  |  |  |
| 5 | 6 |  |  |
| 12 | 9 | 35 | 20 |
| 30 | 15 | 11 | 10 |

**<プログラム 7　ヒープの最小要素の削除>** “ C アルゴリズム全科　**P 96** 引用 ”

```
#include<math.h>

#define HSIZE 100

void main(void);
void deletemin(void);
void shiftdown(int i);
void writeheap(void);
int pow2(int n);

int HEAP[HSIZE+1]={0,2,9,6,12,10,35,20,30,15,11};
int n={10};
int min;

void main(void)
{
   deletemin();
}

void deletemin(void)                    /*最小要素の削除*/
{
   writeheap();
   min=HEAP[1];                          /*最小要素の記憶*/
   HEAP[1]=HEAP[n];                     /*最小要素の削除、最後の要素の根へのコピー*/
   --n;
   writeheap();
   shiftdown(1);
}

void shiftdown(int i)
{
int l,r;                                  /*l は左の子、r は右の子*/
int w;
```

```
l=2*i;r=2*i+1;
while( (l<=n && HEAP[l]<HEAP[i])
        ||(r<=n && HEAP[i]>HEAP[r])){
if(n<r){
   w=HEAP[l];HEAP[l]=HEAP[i];HEAP[i]=w;
   printf("要素%d と要素%d を交換¥n",HEAP[i],HEAP[l]);
   i=l;goto SKIP;
}
if(HEAP[l]<=HEAP[r]){
   w=HEAP[l];HEAP[l]=HEAP[i];HEAP[i]=w;
   printf("要素%d と要素%d を交換¥n",HEAP[i],HEAP[l]);
   i=l;
}
else{
   w=HEAP[i];HEAP[i]=HEAP[r];HEAP[r]=w;
   printf("要素%d と要素%d を交換¥n",HEAP[i],HEAP[r]);
   i=r;
}
SKIP:
   l=2*i;r=2*i+1;
   writeheap();
   }
}

void writeheap()
{
int depth,i,j,endofprint;

/*ヒープの表示*/
if(n!=0){
   depth=(n);
   for(i=0;i<=depth;++i){
if(n>pow2(i+1)-1)endofprint=pow2(i+1)-1;
else                endofprint=n;
for(j=pow2(i);j<=endofprint;++j)
   printf("%6d",HEAP[j]);                          /*深さ一定の点の表示*/
```

```
    printf("¥n");
    }
}
    printf("¥n");
}

int pow2(int n)                                              /*2 の n 乗を求める関数*/
{
if(n>0)return(2<<(n-1));
    else     return(1);
}
```

<実行結果>

```
     2
     9     6
    12    10    35    20
    30    15    11

    11
     9     6
    12    10    35    20
    30    15
```

要素 6 と要素 11 を交換

```
     6
     9    11
    12    10    35    20
    30    15
```

**<プログラム 8　2 分探索木の最小要素の探索>** “ C アルゴリズム全科　P 107 引用 ”

```
#define SIZE 100

void main(void);
void buildtree(void);
int findmin(int v);

int VALUE[SIZE+1],FATHER[SIZE+1],
      LEFT[SIZE+1],RIGHT[SIZE+1];
int root;

void main(void)
{
int v;

   buildtree();                                        /*木の入力*/
   v=findmin(root);                                    /*最小要素の探索*/
   printf("最小要素の値・・・%d¥n",VALUE[v]);
}
void buildtree(void)                                   /*2 分木の入力*/
{
int v,e,l,r;

   printf("velr=");scanf("%d%d%d%d",&v,&e,&l,&r);
   while(v!=9999){
   VALUE[v]=e;
   LEFT[v]=l;RIGHT[v]=r;
   FATHER[l]=v;FATHER[r]=v;
   printf("velr=");scanf("%d%d%d%d",&v,&e,&l,&r);
};

   printf("root=");scanf("%d",&root);
}
int findmin(int v)                                      /*最小要素の探索*/
```

```
{
    while(LEFT[v]!=0)v=LEFT[v];                                /*左の子を探索*/
    return(v);
}
```

<実行結果>

```
velr=1   10   2   3
velr=2    8   4   0
velr=3   15   5   6
velr=4    3   0   7
velr=5   12   0   0
velr=6   19   8   0
velr=7    5   9   10
velr=8   16   0   0
velr=9    4   0   0
velr=10   7   0   0
velr=9999   0   0   0
root=1
最小要素の値・・・3
```

**<プログラム 9　2 分探索木の要素の挿入>“ C アルゴリズム全科　P 114 引用 ”**

```
#define SIZE 100

void main(void);
void buildtree(void);
void dfsearch(void);
void insert(int x);
int search(int x);
int newcell(void);

int VALUE[SIZE+1],FATHER[SIZE+1],
      LEFT[SIZE+1],RIGHT[SIZE+1];                    /*セル領域*/
int CELLSTACK[SIZE+1];
int root,top;

void main(void)
{
int i,x;

for(i=1;i<=SIZE;++i)CELLSTACK[i]=i;
   top=SIZE;                                         /*スタックの先頭要素のインデックス*/
   buildtree();                                      /*木の構築*/
   dfsearch();                                       /*木の表示*/
   printf("挿入する要素　=");scanf("%d",&x);         /*挿入する要素の値*/
   insert(x);
dfsearch();
}

void buildtree(void)                                 /*2 分木の入力  */
{
int v,e,l,r;

printf("velr=");scanf("%d%d%d%d",&v,&e,&l,&r);      /*要素の値の入力*/
   while(v!=9999){
   VALUE[v]=e;
   LEFT[v]=l;RIGHT[v]=r;
```

```
   FATHER[l]=v;FATHER[r]=v;
printf("velr=");scanf("%d%d%d%d",&v,&e,&l,&r);
};
printf("root=");scanf("%d",&root);
}

void dfsearch(void)                                    /*深さ優先順*/
{
int i,v;
int VISIT[SIZE+1];                                     /*巡回用フラグ*/

for(i=1;i<=SIZE;++i)VISIT[i]=0;
   printf("¥n 点番号　要素の値¥n");
   v=root;
   while(VISIT[root]==0){                              /*根が巡回済みになれば終了*/
if(LEFT[v]!=0&&VISIT[LEFT[v]]==0){                     /*未巡回の左の子の検索*/
   printf("%6d %6d¥n",v,VALUE[v]);v=LEFT[v];goto SKIP3;    /*左の子を巡回*/
}
if(RIGHT[v]==0 || VISIT[RIGHT[v]]!=0)goto SKIP1;
   else if(LEFT[v]==0)printf("%6d %6d¥n",v,VALUE[v]);
   v=RIGHT[v];goto SKIP3;                                  /*右の子を巡回*/
SKIP1:
if(RIGHT[v]!=0)goto SKIP2;
   else if(LEFT[v]==0)printf("%6d %6d¥n",v,VALUE[v]);
SKIP2:
   VISIT[v]=1;
   v=FATHER[v];
SKIP3:;
}
   printf("¥n");
}

void insert(int x)
{
int e,v;
   e=newcell()                                         ;/*挿入セルの用意*/
```

```
    VALUE[e]=x;
    LEFT[e]=0;RIGHT[e]=0;
    v=search(x);                                    /*挿入する場所の探索*/
    FATHER[e]=v;                                    /*セルの接続*/
if(x<VALUE[v])LEFT[v]=e;
if(VALUE[v]<x)RIGHT[v]=e;
}

int search(int x)
{
int flag;                                    /*探索の終了判定用フラグ(flag=1:終了)*/
int v;

    flag=0;
    v=root;
    while(flag==0){
if(x==VALUE[v]){printf("点番号　%d¥n",v);flag=1;}
    else if(x<VALUE[v]&&LEFT[v]!=0)v=LEFT[v];
    else if(VALUE[v]<x&&RIGHT[v]!=0)v=RIGHT[v];
    else{printf("存在せず¥n");flag=1;}
    }
return(v);
}

int newcell(void)                             /*未使用セルの確保*/
{
int e;

    e=CELLSTACK[top];--top;
    return(e);
}
```

<実行結果>

```
velr=1   10   2   3
velr=2    8   4   0
velr=3   15   5   6
velr=4    3   0   7
```

velr=5　12　0　0

velr=6　19　8　0

velr=7　 5　9　10

velr=8　16　0　0

velr=9　 4　0　0

velr=10　7　0　0

velr=9999　0　0　0

root=1

| 点番号 | 要素の値 |
|---|---|
| 1 | 10 |
| 2 | 8 |
| 4 | 3 |
| 7 | 5 |
| 9 | 4 |
| 10 | 7 |
| 3 | 15 |
| 5 | 12 |
| 6 | 19 |
| 8 | 16 |

挿入する要素　6

存在せず

| 点番号 | 要素の値 |
|---|---|
| 1 | 10 |
| 2 | 8 |
| 4 | 3 |
| 7 | 5 |
| 9 | 4 |
| 10 | 7 |
| 11 | 6 |
| 3 | 15 |
| 5 | 12 |
| 6 | 19 |
| ８ | 16 |

**<プログラム 10　2 分探索木の要素の削除>** “ C アルゴリズム全科　P 117 引用 ”

```
#define SIZE 100                                /*セルの個数*/
void emptycell(int e);

void main(void);
void buildtree(void);
void dfsearch(void);
void delete(int x);
int search(int x);
int findmin(int p);
void emptycell(int e);

int VALUE[SIZE+1],RIGHT[SIZE+1],
   FATHER[SIZE+1],LEFT[SIZE+1];                 /*セル領域*/
int CELLSTACK[SIZE+1];
int root,top;

void main(void)
{
int i,x,v;
   buildtree();
   dfsearch();
   printf("削除する要素　");scanf("%d",&x);
   delete(x);
   dfsearch();
}

void buildtree(void)                            /*2 分木の入力*/
{
int v,e,l,r;

   printf("velr=");scanf("%d%d%d%d",&v,&e,&l,&r);   /*要素の値の入力*/
   while(v!=9999){
   VALUE[v]=e;
   LEFT[v]=l;RIGHT[v]=r;
```

```
    FATHER[l]=v;FATHER[r]=v;
    printf("velr=");scanf("%d%d%d%d",&v,&e,&l,&r);
}
    printf("root=");scanf("%d",&root);
}

void dfsearch(void)                                    /*深さ優先順*/
{
int i,v;
int VISIT[SIZE+1];                                     /*巡回用フラグ*/
    for(i=1;i<=SIZE;++i)VISIT[i]=0;
    printf("¥n 点番号　要素の値¥n");

v=root;
    while(VISIT[root]==0){                             /*根を巡回したら終了*/
if(LEFT[v]!=0&&VISIT[LEFT[v]]==0){                     /*左の子の検索 */
    printf("%6d %6d¥n",v,VALUE[v]);v=LEFT[v];goto SKIP3;
}
if(RIGHT[v]==0||VISIT[RIGHT[v]]!=0)goto SKIP1;
    else if(LEFT[v]==0)printf("%6d %6d¥n",v,VALUE[v]);
    v=RIGHT[v];goto SKIP3;                             /*右の子を巡回*/

SKIP1:
    if(RIGHT[v]!=0)goto SKIP2;
    else if(LEFT[v]==0)printf("%6d %6d¥n",v,VALUE[v]);

SKIP2:
    VISIT[v]=1;
    v=FATHER[v];

SKIP3:;
}
printf("¥n");
}

void delete(int x)
```

```
{
int e,u,v;

  v=search(x);                                              /*削除すべき要素の探索*/
if(LEFT[v]==0&&RIGHT[v]==0){
if(LEFT[FATHER[v]]==v)LEFT[FATHER[v]]=0;
  else          RIGHT[FATHER[v]]=0;
  e=v;
}
else if(LEFT[v]!=0&&RIGHT[v]!=0){
  u=findmin(RIGHT[v]);                                      /*u は v の右部分木の最小要素*/
if(RIGHT[u]==0){                                            /*u は葉*/
if(FATHER[u]==v)RIGHT[v]=0;                                 /*u は v の右の子*/
else      LEFT[FATHER[u]]=0;                                /*u は親の左の子*/
}
else{                                                       /*u の右の子が存在*/
  if(FATHER[u]==v){                                         /*u は v の右の子*/
  RIGHT[v]=RIGHT[u];FATHER[RIGHT[u]]=v;
}
else{
  LEFT[FATHER[u]]=RIGHT[u];
  FATHER[RIGHT[u]]=FATHER[u];                               /*u は親の左の子*/
  }
}
  VALUE[v]=VALUE[u];                                        /*u の要素を親に割り当てる*/
  e=u;
}
  else{
if(LEFT[v]!=0)u=LEFT[v]                                     ; /*子が一つだけ存在*/
    else        u=RIGHT[v];
if(LEFT[FATHER[v]]==v)LEFT[FATHER[v]]=u;
                  else         RIGHT[FATHER[v]]=u;
  FATHER[u]=FATHER[v];
  e=v;
}
  emptycell(e);
```

```
}

int search(int x)
{
int flag;                                    /*探索の終了判定用フラグ(flag=1:終了)*/
int v;

   flag=0;
   v=root;
   while(flag==0){
if(x==VALUE[v]){printf("点番号%d¥n",v);flag=1;}
   else if(x<VALUE[v]&&LEFT[v]!=0)v=LEFT[v];
   else if(VALUE[v]<x&&RIGHT[v]!=0)v=RIGHT[v];
   else{printf("存在せず¥n");flag=1;}
}
   printf("入れ替えられた点　%d",v);
   return(v);
}

int findmin(int v)
{
   while(RIGHT[v]!=0)v=RIGHT[v];
   return(v);
}

void emptycell(int e)
{
   ++top;CELLSTACK[top]=e;
}
```

<実行結果>

```
velr=1   10   2   3
velr=2    8   4   0
velr=3   15   5   6
velr=4    3   0   7
velr=5    12    0    0
velr=6    19    8    0
```

velr=7 5 9 10

velr=8 16 0 0

velr=9 4 0 0

velr=10 7 0 0

velr=9999 0 0 0

root=1

点番号　要素の値

| 点番号 | 要素の値 |
|---|---|
| 1 | 10 |
| 2 | 8 |
| 4 | 3 |
| 7 | 5 |
| 9 | 4 |
| 10 | 7 |
| 3 | 15 |
| 5 | 12 |
| 6 | 19 |
| 8 | 16 |

削除する要素　5

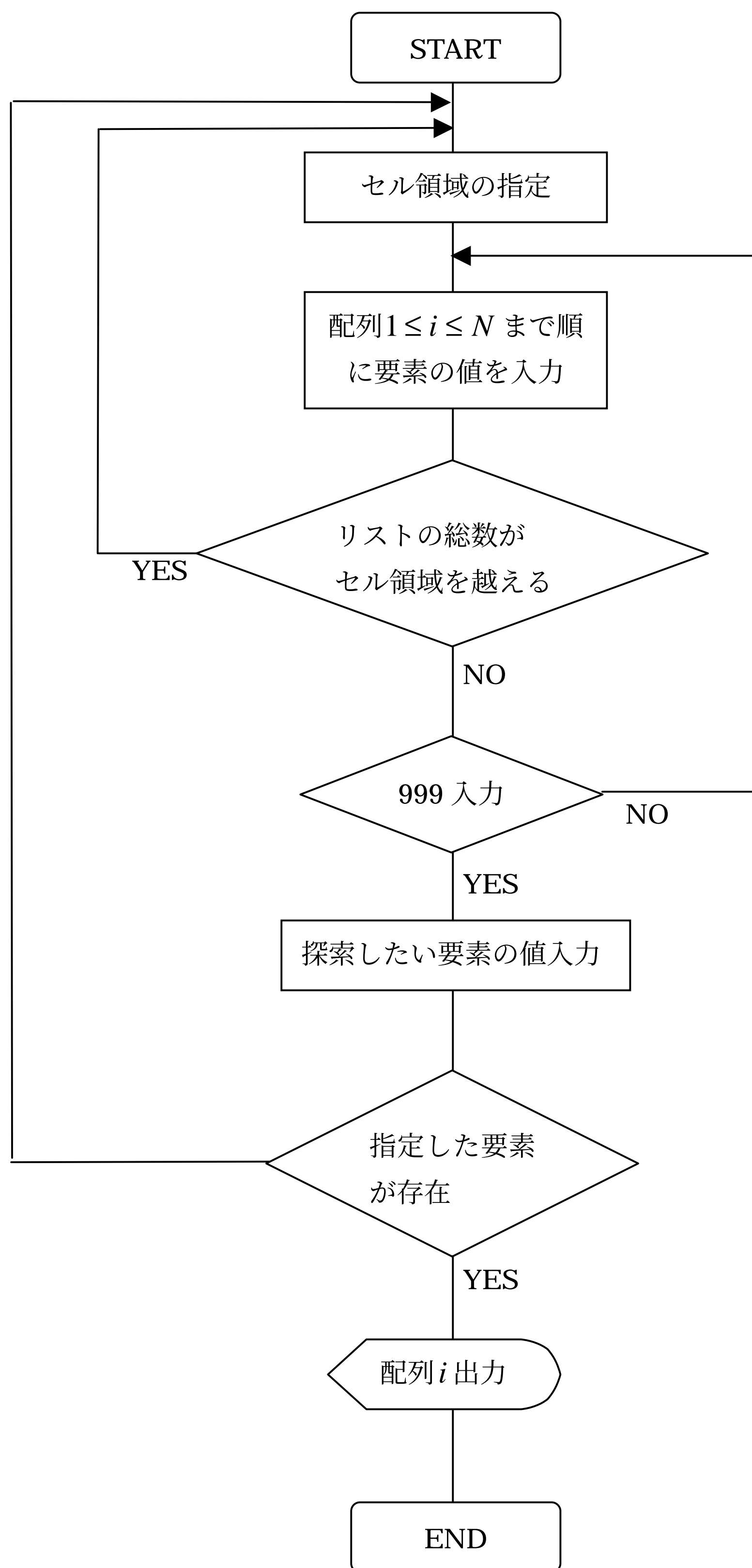
点番号　7

入れ替えられた点　7

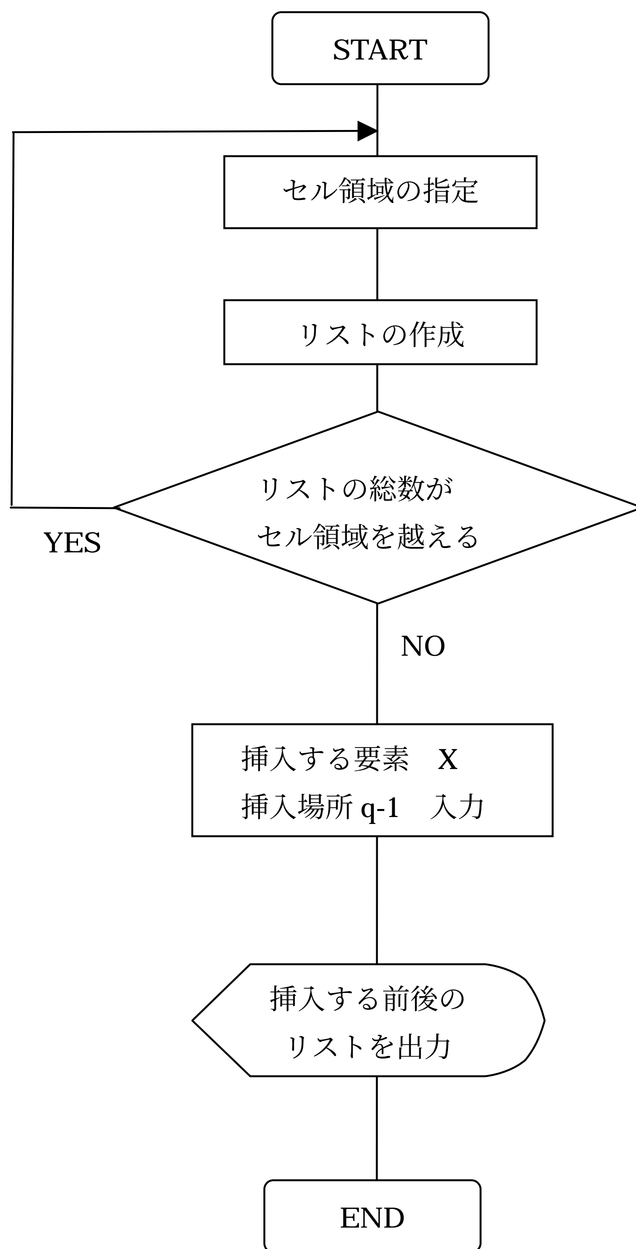
点番号　要素の値

| 点番号 | 要素の値 |
|---|---|
| 1 | 10 |
| 2 | 8 |
| 4 | 3 |
| 7 | 7 |
| 9 | 4 |
| 3 | 15 |
| 5 | 12 |
| 6 | 19 |
| 8 | 16 |

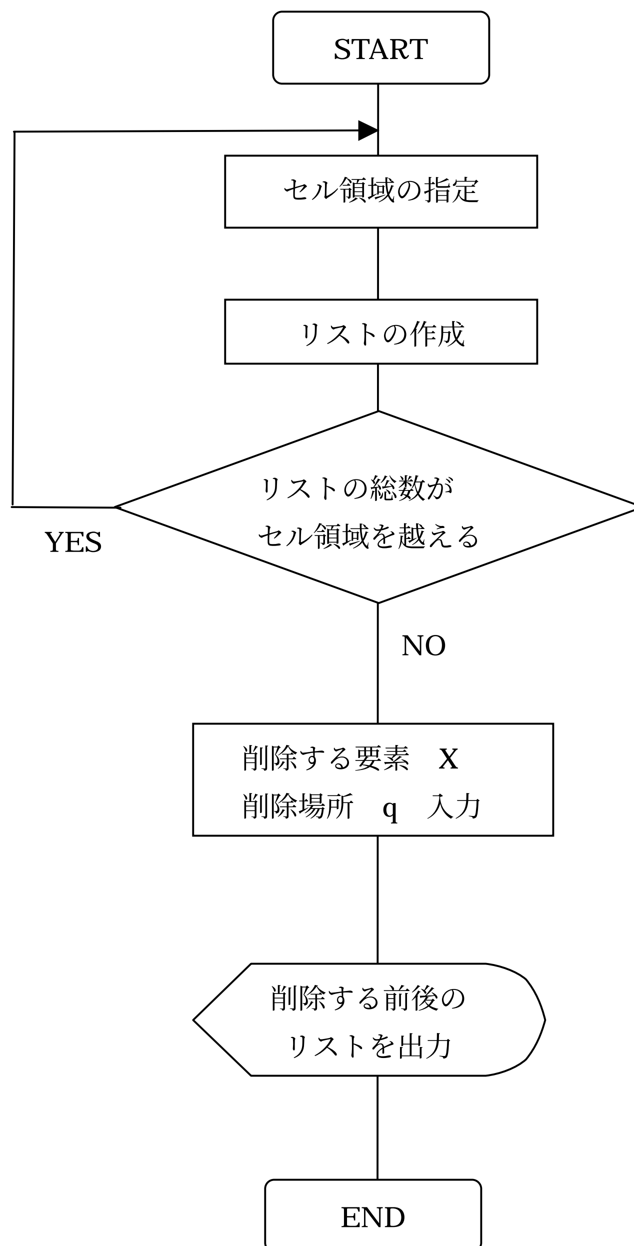
<リスト　要素の探索>



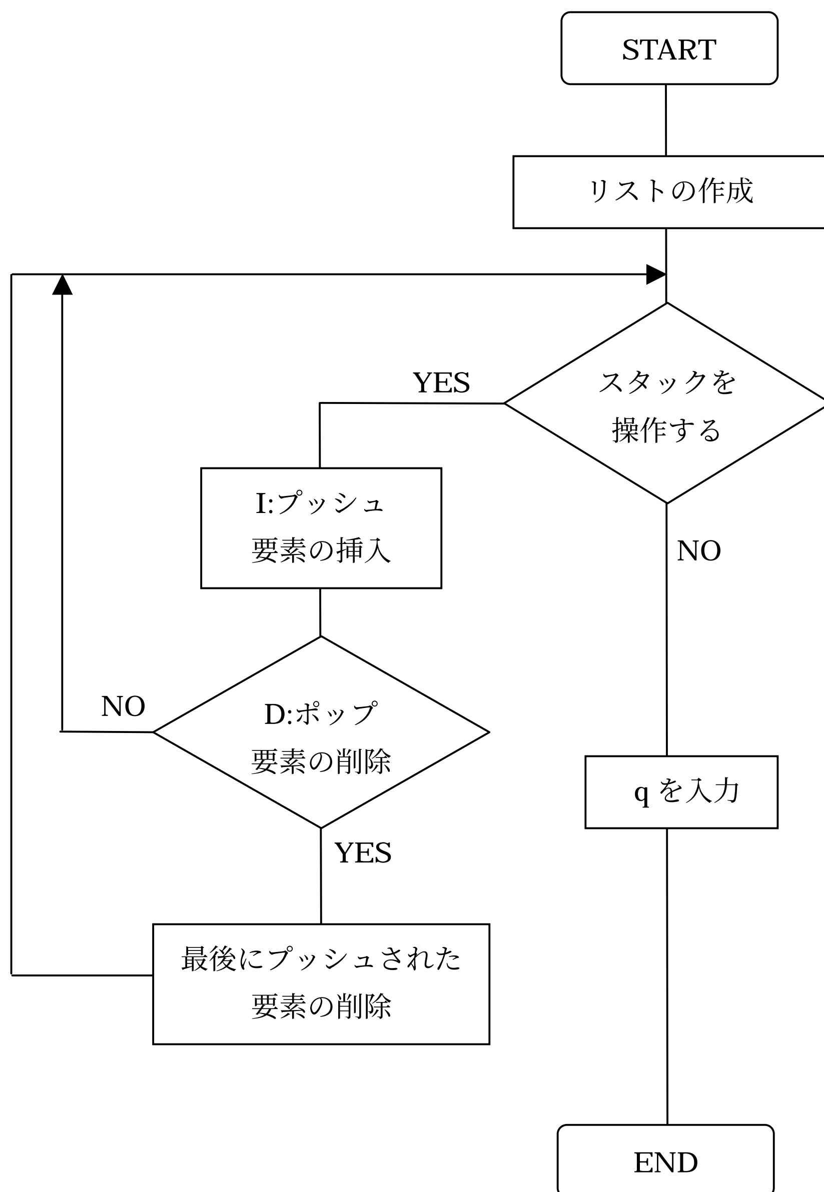
<リスト　要素の挿入>



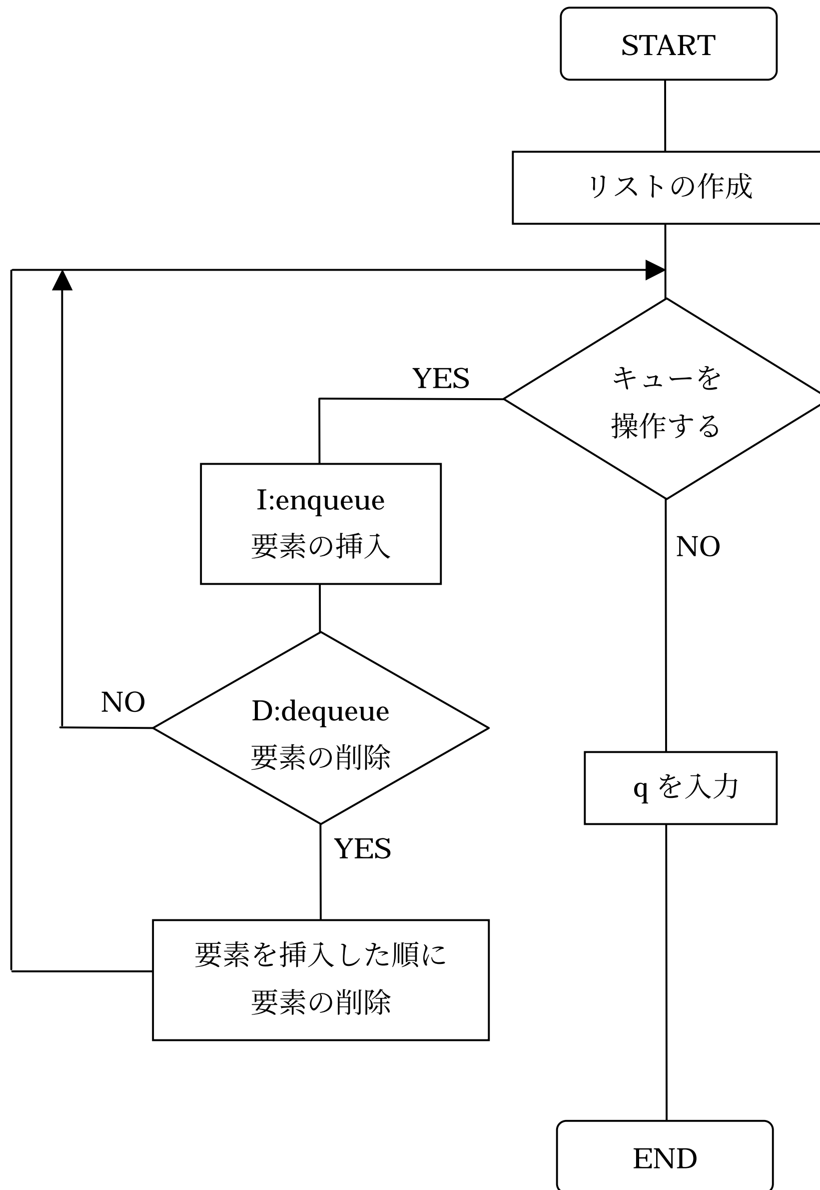
<リスト　要素の削除>



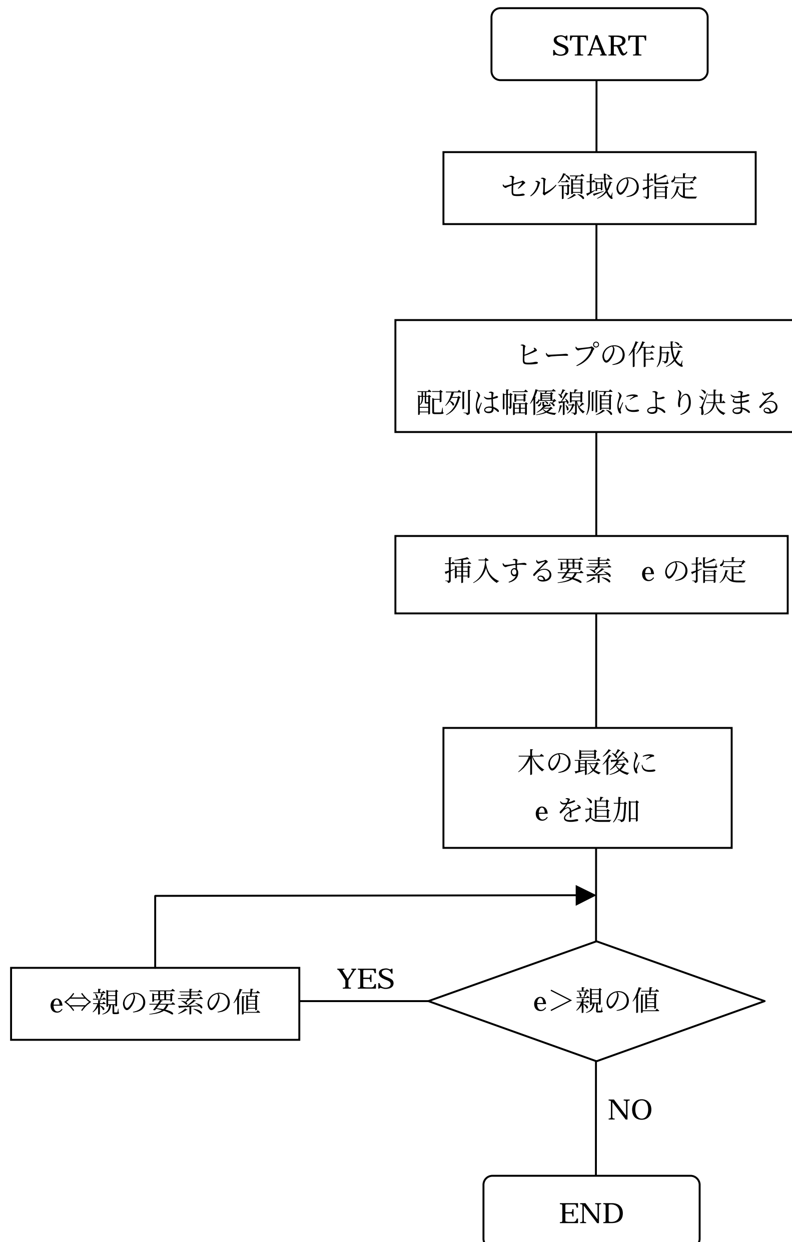
<スタックの操作>



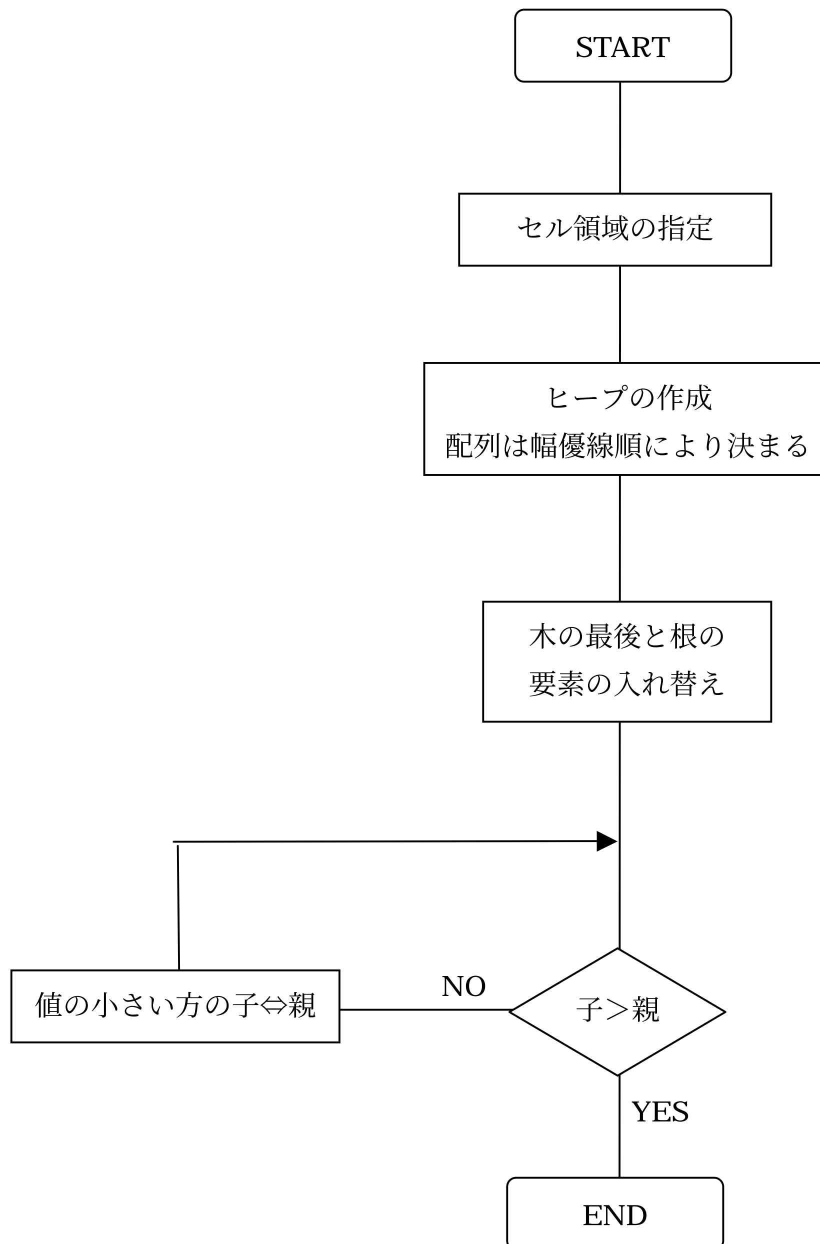
<キューの操作>



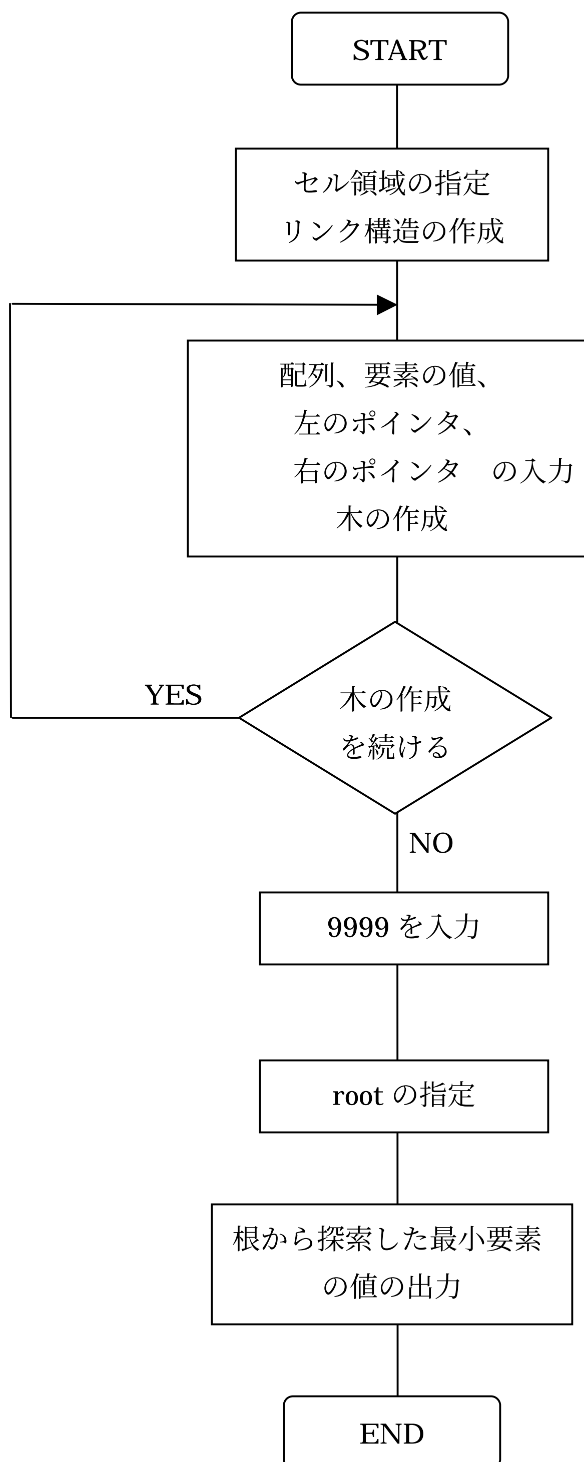
<ヒープ　要素の挿入>



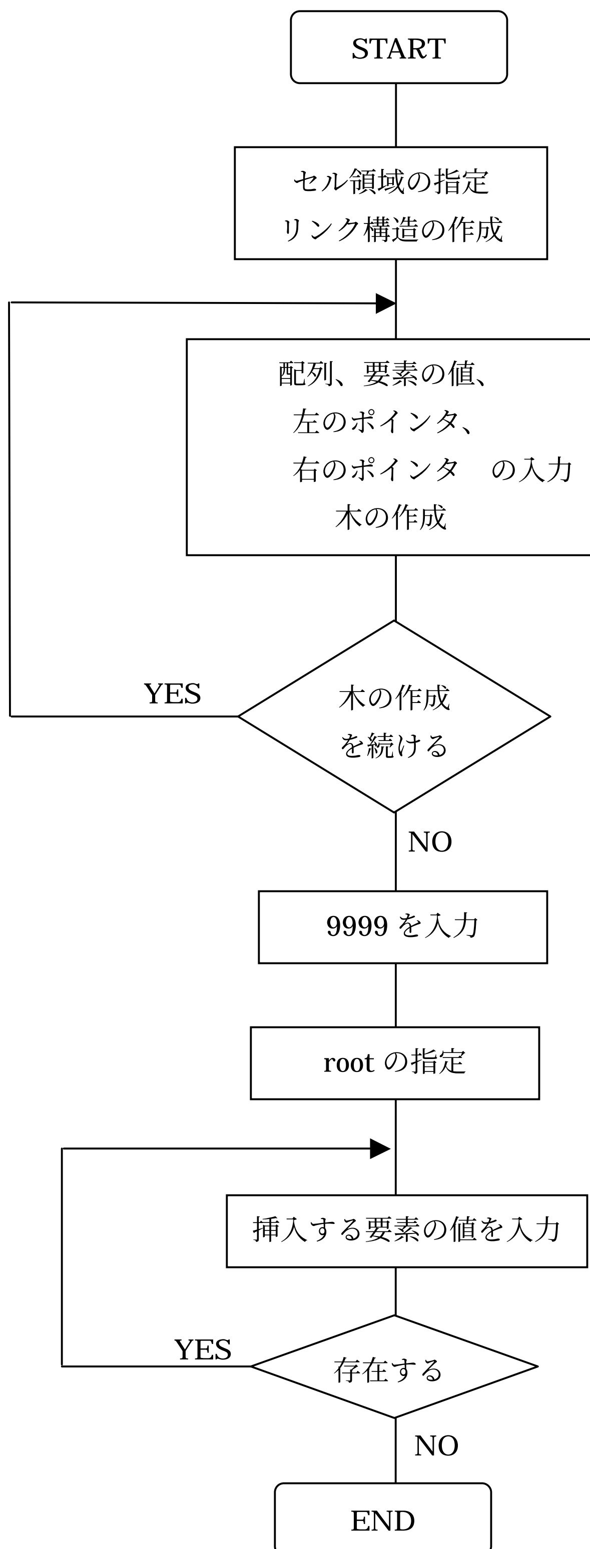
<ヒープ　最小要素の削除>



<2 分探索木　最小要素の探索>



<2 分探索木　要素の挿入>



<2 分探索木　要素の削除>