平成１２年度

卒業論文

C 言語を使ったデータ構造と探索アルゴリズムの効率性の評価

指導教員

山本　哲也　助教授

電子・光システム工学科

学籍番号：１０１０３３８

松岡　茂行

テーマ：C 言語を使った、データ構造と探索アルゴリズムの効率性

目次

付録

プログラムリスト　1，2，3，4，5

グラフ

卒業研究概要（プレゼンテーションセミナー資料）

C 言語を使ったデータ構造と探索アルゴリズムの効率性

１．テーマと目的

１．１　テーマ

本稿のテーマは、C 言語を使ったデータ構造と探索アルゴリズムの効率性についてである。

１．２　背景

コンピュータで扱うデータとは、コンピュータ内で扱うことのできるひとつの要素のことである。データは組織化されておらず、ばらばらなデータの集まりでは、扱いにくいので、扱いやすいように、このデータの集まりを組織化し構造を持たせたものがデータ構造である。データ構造の概念は“ データ処理の効率化を図るためにデータを組織化する。”というものである。よく使われるデータ構造は、リスト構造、木構造、表構造がある。

そこで、扱いたいデータ群に対して、どういうデータ構造で組織化し、扱うと効率が良いかということを考える。以下では、探索アルゴリズムの効率という部分から説明する。

プログラムを作成する時には、データの表現方法と処理手順を明確に定義し、それらをプログラム言語で書くことが必要になってくる。その基礎となるのがデータ構造とアルゴリズムなのである。

１．３　目的

本論文の目的と概要は、次の２つである。

- データ構造のそれぞれの機構や性質、またプログラムでの実現方法などを理解する。
- 数値データの容量に対して、どのデータ構造が有効かを考える。

データ構造の中でも、よく使われるリスト構造、木構造、表構造の３つについて説明する。この３つのデータ構造の機構、性質などを理解したうえで、データ構造はなぜいくつもあるのか、どう使い分けるのかを明確にするためにそこでデータ容量（今回は数値データ）と、探索アルゴリズムの効率について議論し、有効なデータ構造について検討する。

２．データ構造とアルゴリズムの効率

２．１　データ構造の概念

プログラミングを行うとき、様々な形式、形態をもつ情報を扱わなければならない。しかし、一般にそのままの形では処理することができない。そこでコンピュータで扱うことのできるように情報を形式化したものをデータと呼び、データ構造の概念は“データ処理の効率化を図るためにデータを組織化する。”ことである。

プログラムミングの時、データ構造はプログラム言語のデータ型を組み合わせて実現することができる。

データ構造の特徴は、大きくわけて、静的なデータ構造と動的なデータ構造の２つがある。

静的なデータ構造とは、プログラムの実行中にデータの大きさや形態が変化しないというものである。そのため、配列を使う時などは、あらかじめデータ件数がわかってない場合は、要素数を多めに宣言しなければならない。要素数を少なく取ってメモリが足りなくなった時は、その時点で、処理がとまってしまう。そのため、メモリを余計に使わなければならないという欠点がある。

これに対し、動的なデータ構造とは、プログラムの実行中にデータの大きさや形態が変化するというものである。これは、静的なデータ構造に、メモリを動的に割り当てることで実現される。これは、高等なデータ処理に使われる。

今回は、静的なデータ構造を使って、プログラムを実現していく。

２．２　アルゴリズムとデータ構造の関係

プログラムを考える時は、データ構造とアルゴリズムを同時に考えていかなければならない。どちらかを先に考えるということはできない。このことから、データ構造とアルゴリズムは、不可分の関係にあるといえるだろう。さて、そのアルゴリズムとは、問題を解決するための計算手順や解法手順など、手順の集まりのことである。アルゴリズムの表現方法は、文章や図などを使う。コンピュータで扱う場合には、プログラム言語を使いプログラムとして扱わなければならない。しかし、アルゴリズムは、データの処理手順を示したものであり、データの表現についての概念がない。そこで、データ構造という概念を取り入れる。言ってみれば、プログラムとはデータをデータ構造によって形式化し、アルゴリズムという処理手順にしたがい処理するものである。

ある問題を解くとき、そのデータをどのようなデータ構造で表現し、それをどのようなアルゴリズムで処理するのかを、検討しなければならない。これが、プログラミングをする際に重要なことである。そこで、本論でも、アルゴリズムの効率という観点からそのことを考えていく。

２．３　アルゴリズムの効率の評価方法

アルゴリズムの効率をどの様に評価するかを考える。アルゴリズムは、たとえ同じ問題を解く時にでも、一意的ではなく、いくつかの可能性がある。その中で、その問題に最も有効なアルゴリズムは何かということを計算量という概念を使って考える。

あらゆる処理系で、アルゴリズムの効率を評価することのできる唯一の方法は“このアルゴリズムはどの程度の手間がかかるか”というものである。これは、プログラムの記憶数や実行回数を評価するものである。なぜなら、コンピュータの処理系の性能が違えば実際の実行時間だけで効率を評価することはできないからである。

計算量で評価する時には、O 記法（ビッグオー記法）を使う。これは、プログラム文の実行回数を、表現する式の各項の係数や定数の値は考えない桁数評価方法である。これは、n に関する２つの関数 f(n)、g(n)に対し、

$$f(n) <= cg(n), \quad n >= n_0$$

を満たす定数 c、n_0 が存在するとき、

「f(n)は g(n)のオーダーである」

と表現され、

$$f(n) = O(g(n))$$

と記述する。

これによって関数の増加傾向を見積もることができる。この増加傾向からアルゴリズムの効率を評価する。オーダーを使った増加傾向の評価なら、複雑なアルゴリズムも評価することができる。なぜなら、オーダーの性質として、“計算量の関数の最も大きい項に対するアルゴリズム部分だけを、かつ定数係数を無視して考慮するだけでよい”からである。これは、アルゴリズムの計算量の評価を容易に可能とさせる。

アルゴリズムの計算量には、領域計算量と時間計算量とがある。領域計算量とは、プログラムで使った記憶領域のサイズ（変数の数や配列）の、程度を示したものである。時間計算量とは、処理時間の程度を示したものである。これらを、データ件数に対する増加傾向で評価する。またその際、実行回数は最悪（最大）の場合を考える。

注意点として、計算量のオーダーの小さいアルゴリズムは一般に複雑であり、定数係数が大きくなる傾向がある。そのため、データ件数が少ない場合は、各項の定数係数まで考えた計算量の評価が重要になる時もある。そのためデータ件数に応じたアルゴリズムを採用することが大切である。

また、領域計算量を増加させることで、時間計算量を減少させることができる場合もある。その逆もある。

以上の注意点も含め、第 6 章で探索アルゴリズムの領域計算量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。

２．４　C 言語を使う利点

本論に進む前に、今回プログラム言語としてC言語を用いた理由について説明する。

C 言語はリンク構造などを実現するデータ型が豊富で、データ構造の実現に有効な言語である。

データ構造をプログラムで実現する際、プログラム言語に備わっているデータ型を組合すことで実現できる。そのため、データ型が豊富で表現力が豊かな言語が、データ構造の実現には向いているだろう。そこで、データ型が豊富な C 言語を使えば、いくつもの種類のデータ構造を実現することが可能である。一方、BASIC や FORTRAN などは、リンク構造のあるデータ型がないので、データ構造を実現する場合には向いていない。

今回はデータ構造の機構などを理解するという目的があるので、リンク構造のあるデータ型は使わず、データ構造の機構を明らかにするため、配列型をいくつか組み合わせ、リンク構造を実現する。

３．リスト構造

３．１　リスト構造とは

リスト構造の概念は、情報処理の分野では“ 順序付けられた要素の並び ”とされる。また並びとも表現される。リスト構造は、データ構造の中でも単純であり、最も自然に使われる。リストを図で示す。（図 1）

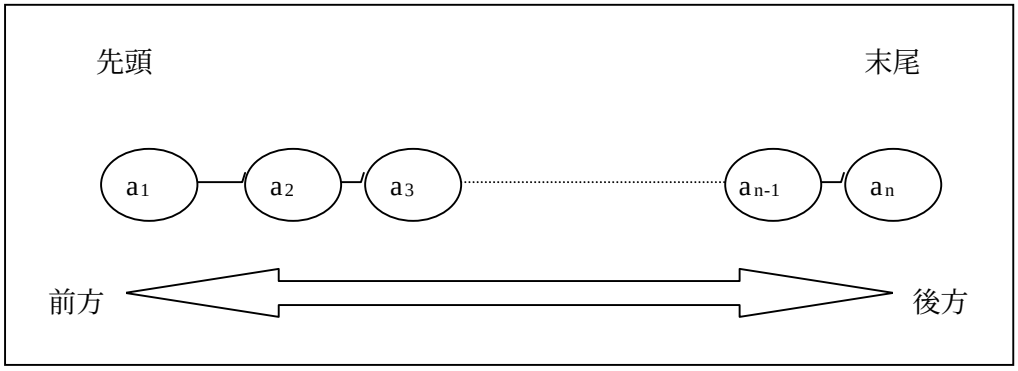


図３．１．１　リストの概念図

リストの要素は、値に１～ｎまで順番がついて並んでいる。そのため、a_i の要素の直前要素は a_{i-1} となっている。このように順序のついた要素が並んでいるものをリストと呼ぶ。

リストは、要素の探索、挿入、削除の操作をすることができる。またリストの種類によって操作方法が変わる。それはまず、リストの種類として、スタック、キュー、デク、連結リストがある。以下それぞれの操作方法を説明する。

スタックは、リストの先頭位置で要素の挿入、削除がされる。キューは、リストの末尾位置に要素を挿入し、先頭位置から要素が削除される。デクは、先頭位置、末尾位置で、要素の挿入、削除がされる。連結リストは、リストの任意の位置に、要素の挿入、削除ができる。今回は、連結リストを使って、任意の位置への要素の操作を行う。

３．２　リストの実現

リストの実現方法は、２つある。1 つは 1 次元配列へ、要素を順番に格納していく方法（図 3.2.1)、もう 1 つは、配列を 2 つ使い、1 つの要素（セル）をつくり、リンク構造によってつながっていく（図 3.2.2)。その時、2 つの配列には、要素の値と隣接要素の位置を示したポインタが格納される。連結リストを実現するために、2 つ目の方法を使う。1 つめの方法には、要素の操作がしにくいという欠点がある。

図３．２．１　１次元配列での実現

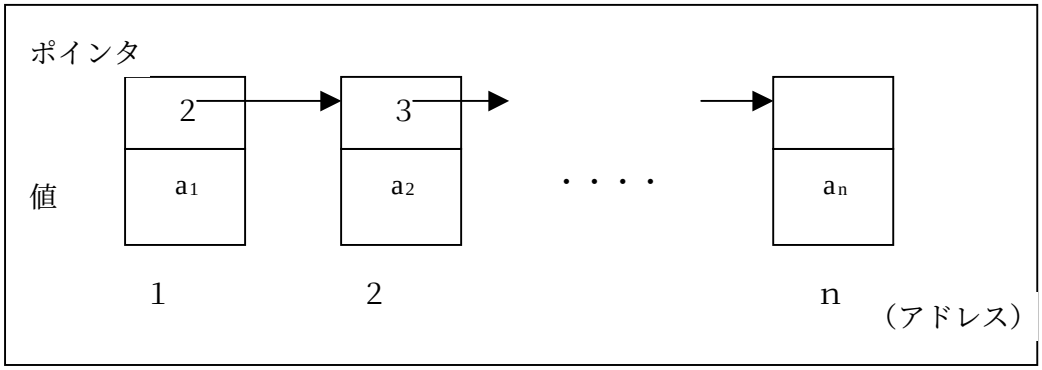


図３．２．２　リンク構造での実現

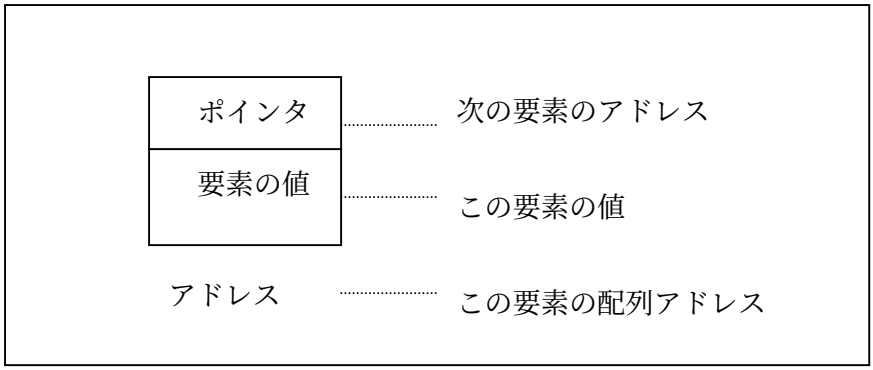


図３．２．３　１つの要素

リンク構造で実現する場合、今回は要素間のリンクを、片方向リンクを使って行った。要素の１つは図 3.2.3 に示したように、２つの構造からできている。それぞれ、次の要素へのポインタを入れる配列ＮＥＸＴ、要素の値を入れる配列ＶＡＬＵＥを使ってリンク構造を実現した。それでは、実際にリストを使って、要素の操作を試みる。

３．３　リスト構造の操作

【要素の探索】

リストを使って要素の探索を行う。これは、リスト中から探索したい要素が存在するかを調べ、存在していればその要素の配列アドレスを結果として表示、存在していなければＮＵＬＬ（０）を表示するというプログラムである。
それでは、このプログラムの流れを示す。（プログラムリスト１“要素の探索”参照）

①リストの作成　　・・・キーボードより数値を入力していき、９９９を入力すると作成完了で次へ行く。値は配列ＶＡＬＵＥに入り、その要素の値となる。

②探索したい要素の入力・・・キーボードより探索したい要素の値を入力。今回は１つの値とする。

③リストの中から探索　・・・①で作ったリストの中から、②で入力した値が存在するかどうかを調べる。

④結果の表示　　・・・探索結果を表示する。存在するなら、その要素のアドレスを表示。存在しなければ、０を表示とする。

以上がプログラムの流れである。では、次に探索部分のアルゴリズムを説明する。
リストの探索アルゴリズムは、先頭要素から順に、その要素の値と探索したい値とが等しいかどうかを判定する。等しい値が見つかったその時点で、結果を表示。見つからなければ、この作業をリストの最後まで行う。

【要素の挿入】

次に要素の挿入について説明する。これは、すでにできているリスト中に、新たな要素を 1 つ、任意の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く。リストは図 3.3.1 に示す。このリストの 5 番目に１２と

いう要素の値を挿入する。（プログラムリスト１“ 要素の挿入 ” 参照）

配列	1	2	3	4	5	6	7	8	9
VALUE		3	1	8	２３		6	６３	
NEXT	9	4	8	5	3	1	2	0	１７

図３．３．１　配列で作成したリスト

では、このプログラムの流れを説明する。

① 挿入前のリストの表示　・・・　図 3.3.1 の配列を、先頭要素を配列[７]とし
て要素の値（ＶＡＬＵＥ）を順に表示していく。結果を下に示す。
６　３　８　２３　１　６３

② 挿入したい値と位置とを決める・・・挿入したい値を１２、位置は直前要素
の値を指定する。今回は５とする。

③　挿入する要素の作成　・・・配列[６]に要素の値１２の新しい要素を作る。

④　挿入後のリストの表示　・・・新しい要素の直前、直後の要素のポインタをつなぎ合わせ、リストを表示する。
下に結果を示す。
６　３　８　２３　１２　１　６３

以上が、プログラムの流れである。次に挿入部分のアルゴリズムについて説明する。要素を挿入するアルゴリズムは、まず挿入する値のための要素ｘを作成する。つぎに、要素ｘのポインタ部（NEXT[x]）を、直前要素の本来のポインタ部（NEXT[y]）の値とかえる。そして、直前要素のポインタ部（NEXT[y]）が、要素ｘを指すように、NEXT[y]=x にする。

【要素の削除】

リストの最後に要素の削除について説明する。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内で作っておく。また、挿入後の時のものと同じリストを使う。（図 3.3.2）

配列	1	2	3	4	5	6	7	8	9
VALUE		3	1	8	２３	１２	6	６３	
NEXT	9	4	8	5	6	3	2	0	１７

図３．３．２　配列で作成したリスト挿入後

それでは、プログラムの流れを説明する。（プログラムリスト１“要素の削除”参照）

① リストの表示　・・・　図 3.1.6 の配列を、先頭要素を配列[７]として要素
の値（ＶＡＬＵＥ）を順に表示していく。結果を下に示す。
６　３　８　２３　１２　１　６３

② 削除したい要素を決める・・・　削除したい要素のアドレスを入力しておく。
配列[６]の要素を削除する。

③ ②の要素の直前要素を見つける・・・　ポインタ部（配列 NEXT）が６の要素を見つける。

④ 要素を削除しリストを表示　・・・　直前要素がわかれば、ポインタをつなぎ変え要素を削除する。そして、その結果を表示する。結果を下に示す。
６　３　８　２３　１　６３

以上が要素を削除するときの、プログラムの流れである。それでは、削除部分のアルゴリズムについて説明する。削除したい要素ｘを決め、直前要素のポインタ

（NEXT[y]）を探す。その要素ｙのポインタ（NEXT[y]）が、要素ｘの次の要素を、指すようにポインタにする。具体的に、要素ｘのポインタ（NEXT[x]）の値を、直前要素のポインタ（NEXT[y]）とすればよい。

以上がリストの操作である。リストの要素は、ポインタと要素の値でできているので、挿入や削除は、直前要素が見つかれば、ポインタをつなぎかえるだけでできる。今回は片方向リンクを使って実現したが、両方向リンクを使えば、直前要素を探す手間が省ける。双方向リンクは、１つの要素に、直後の要素へのポインタと値のほかに、直前要素へのポインタの３つの部分で構成されているものである。

そして、要素を探索する時は、最悪の場合要素の数だけ、比較していかないといけない。そのため、比較する実行回数は、要素の数に比例する。このことについては、第６章で詳しく説明する。

リスト構造は、値を入力していくだけで、容易に実現することができるので、よく使われるデータ構造である。

４．木構造

４．１　木構造とは

木構造の概念は、根となる要素からはじまって、枝分かれしながら要素を配置していくものである。根が最上部になり、そこから下へ枝分かれしている。また、各点が２つ（最高で２つであり、１つでも０でもよい）に分かれているものを２分木という。この２分木をベースにした、２分探索木とヒープとをつかって、要素を操作していく。

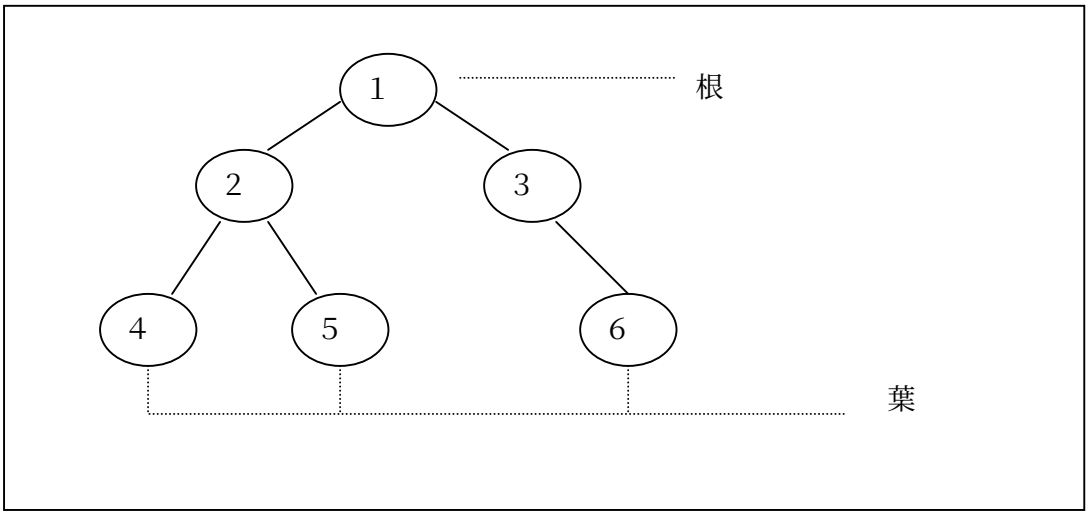


図４．１．１　木構造の概念図

また、この図以外にも木には、次のような関係がある。まず木の直接つながっている上下間の関係を親子といい、同じ列にある関係を兄弟という。ある要素を根として、そこから広がってく木を部分木という。

つぎに、木の順序付けは、幅優先順と、深さ優先順がある。幅優先順は、深さの浅い所から、左から順に数えていく。上の図１は、幅優先順に番号を付けている。これは、ヒープの実現の時に使われる。深さ優先順は、根から順に左部分木からから順番をつけていく。この方法では、順序付けに３パターンのタイミングがある。今回は中間順を使って順番をつけていく。中間順とは、１度左部分木の葉まで行って帰ってくる時に順番をつけ、次に右部分木の葉まで行って帰ってくる時に順番をつける方法である。

木構造の実現方法は、１つの配列を使って実現する方法と、配列を４つ使って１つの要素を表現する方法がある。最初の方法は、ヒープを実現する時に使い、次の方法は、２分探索木を実現する時に使う。それでは、まず２分探索木について説明する。

４．２　２分探索木とは

２分探索木は、２分木の各点に、次のような性質に従い、登録したものである。その性質というのは、“すべての点ｖについて、その左部分木の点はすべてｖより小さく、右部分木の点はすべてｖより大きい”というものである。これは、簡単に言うと、親子間で、親より小さい値は左へ、大きい値は右へ登録されていくというものである。そのため、最小要素は左端にあり、最大要素は右端にある。

この２分探索木を使って要素の探索、挿入、削除ができる。

下に２分探索木の例を示す。（図 4.2.1）

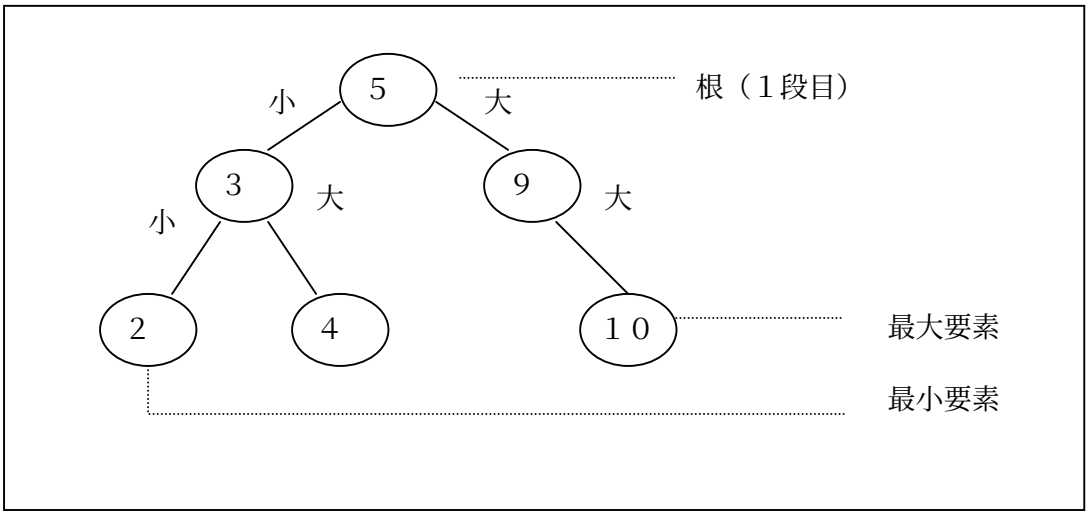


図４．２．１　2 分探索木の概念図

図4.2.1の２分探索木は、根の値が５なので、１段目から２つに枝分かれしているが、根の値を２にした場合は、２分探索木の性質どおり並べると１列に並んでしまう。このような木では、木構造の特徴が生かせず、リストと同じになってしまうので、要素を登録する順番にも注意しなければならない。もちろんこの状態においても２分探索木という。

４．３　２分探索木の実現

それでは、２分探索木の実現について説明する。２分探索木は、４つの配列を使って１つの要素を実現する。その配列に入るものは、まず要素の値用配列ＶＡＬＵＥ、その要素の親へのポインタ用配列ＦＡＴＨＥＲ、左要素へのポインタ用配列ＬＥＦＴ、右要素へのポインタ用配列ＲＩＧＨＴである。この要素を配列のリンク構造によってつなげていく。

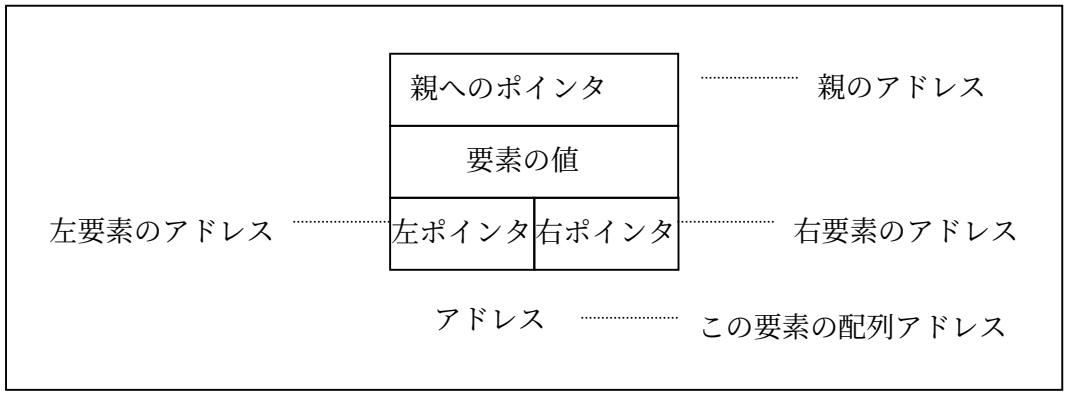


図４．３．１　2 分探索木の 1 つの要素

左右ポインタには、それぞれの子のアドレス（点番号）を入力する。ない場合は、０を入力する。点番号は、根から順に１、次の段の左から２，３というように、上の段の左から順に番号を付けていく。

それでは、実際に 2 分探索木を実現して、操作を行う。

４．４　２分探索木の操作

２分探索木を使って、要素の探索を行う。最小要素の探索は、根から順に、左部分木の左の子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いい。それに対して最大要素の探索は、根から順に右部分木の配列ＲＩＧＨＴが最初に０になった要素を探せばいい。（プログラムリスト２“ 最小要素の探索 ”“ 最大要素の探索 ” 参照）詳しい説明は省略する。

【要素の探索－2 分探索木】

２分探索木の中から、指定した要素を探索する。これは要素を探索し、存在しているかを判定するプログラムである。今回要素の操作には、図 4.4.1 の２分探索木を用いる。それでは、このプログラムの流れを示す。(プログラムリスト２“ ２分探索木の要素の探索 ” 参照)

① ２分木の入力　・・・キーボードから点番号 v、要素の値 e、左右リンク先 l,r を入力する。V に９９９９を入力で入力終了。結果図 4.4.1 の２分探索木になる。

② 探索したい値の入力・・・キーボードより探索したい要素ｘを入力。

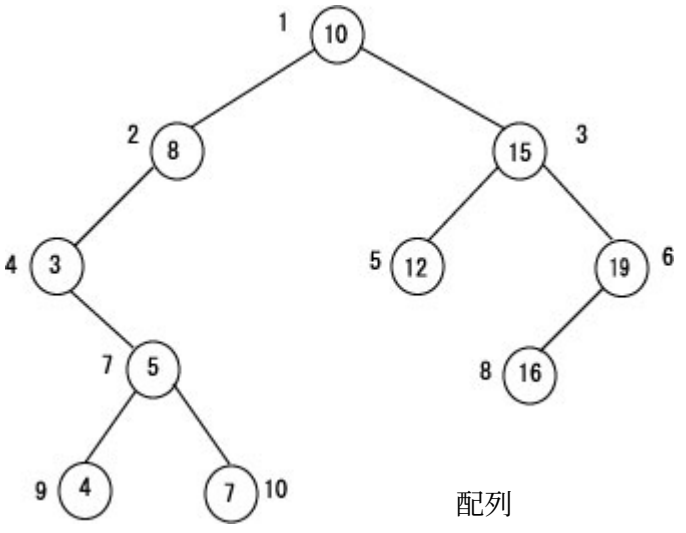
③ 要素の探索　・・・要素ｘを、２分探索木の中から探索する。

④　結果の表示　　・・・２分探索木の中に要素ｘと同じ値があれば、その要素の点番号を表示する。なければ“存在せず”を表示する。

以上がプログラムの流れである。つぎに探索部分③のアルゴリズムについて説明する。２分探索木の探索のアルゴリズムは、２分探索木の根の値より小さければ左へ、大きければ右へという性質を使って行われる。

(1) 木に要素がある間、根の値と探索したい要素ｘの値を比較する。
　(a)　値が同じなら、探索成功で④へ。
　(b)　ｘ＜根の値なら、左部分木をたどる。
　(c)　ｘ＞根の値なら、右部分木をたどる。
(2) 次の部分木に行き①の作業を繰り返す。
(3) 木に要素がなくなると探索失敗で④へ

これが、２分探索木の探索のアルゴリズムである。リストと違い２分木の高さだけ、比較を行えばいい。



配列

	1	2	3	4	5	6	7	8	9	10
F		1	1	2	3	3	4	6	7	7
L	2	4	5	0	0	8	9	0	0	0
R	3	0	6	7	0	0	10	0	0	0
V	10	8	15	3	12	19	5	16	4	7

図４．４．１　２分探索木

【要素の挿入－2 分探索木】

次に要素の挿入について説明する。これは２分探索木に、要素を挿入するものである。また、挿入後も２分探索木の性質を満たさなければならない。使用する２分探索木は図 4.4.1 のものを使う。それでは、プログラムの流れを示す。（プログラムリスト２“ ２分探索木の要素の挿入 ”参照）

① ２分木の入力 ・・・要素の探索と同じ。（図 4.4.1）

② 木の表示 ・・・深さ優先順で、①で入力した図 4.4.1 の木（点番号と要素の値）を表示していく。中間順で順番をつける。

③ 挿入したい要素の入力・・・キーボードより挿入したい要素ｘの値を入力。

④ 要素の挿入 ・・・新しい要素ｘをつくり。関数 newcell より点番号 100 の要素を用意される。それを２分探索木に挿入する。その位置は関数 search で探す。

⑤ 木の表示 ・・・要素挿入後の木を深さ優先順で表示する。

以上が要素の挿入の、プログラムの流れである。次に挿入部分の④の部分のアルゴリズムを説明する。2 分探索木は、まず挿入位置を探し、そこにポインタをつなぎ挿入するアルゴリズムである。

(1) 木に要素がある間、根の値と探索したい要素ｘの値を比較する。
(ア) 値が同じなら、既に挿入されている。
(イ) ｘ＜根の値なら、左部分木をたどる。
(ウ) ｘ＞根の値なら、右部分木をたどる。
(2) 部分木に対して①を繰り返す。
(3) 部分木に要素がなくなった根の位置に新しく用意した要素ｘをつなぐ。

以上が挿入のアルゴリズムである。要素の挿入は、挿入する位置を探索しなければならない。しかし、挿入位置さえ決まれば、ポインタをつなぎかえるだけなので手間はかからない。よって、要素の探索と同じくらいの手間がかかる。

【要素の削除－2 分探索木】

最後に、要素の削除について説明する。要素の削除は、図４の 2 分探索木から指定した要素を削除するというものである。また削除後も 2 分探索木の条件を満たしていないといけない。それでは、プログラムの流れを示す。（プログラムリスト２“ 要素の削除 ” 参照）

① ２分木の入力 ・・・要素の探索と同じ。（図 4.4.1）

② 木の表示 ・・・深さ優先順で、①で入力した図 4.4.1 の木（点番号と要素の値）を表示していく。中間順で順番をつける。

③ 削除したい要素の入力・・・キーボードより削除したい要素ｘの値を入力。

④ 要素の削除 ・・・2 分探索木から要素ｘと同じ値を関数 search で探し、その要素を削除する。

⑤ 木の表示 ・・・図 4.4.1 から要素ｘを削除したものを深さ優先順で表示する。

以上が要素の削除の、プログラムの流れである。それでは、削除部分のアルゴリズムを説明する。要素を削除する時、削除する要素が木のどこにあるかによってアルゴリズムが変わる。３つの場合がある。

(A) 要素ｘが葉の場合。
(B) 要素ｘが左右どちらに子を 1 つだけもつ場合。
(C) 要素ｘが左右両方に子を持つ場合。

要素がどこにあるか（上のどの条件になるか）は、探索の時と同じアルゴリズムで見つける。それぞれの場合のアルゴリズムを説明する。

まず A の場合は、要素ｘは葉なので、そのまま削除しても 2 分探索木は成り立つので、その要素をそのまま削除する。要素ｘに対する親の要素のポインタ（左右どちらか）を０にすればよい。

次に B の場合は、要素ｘを削除して、要素ｘのただ 1 つの子の要素を、要素ｘの位置にもってくればよい。なぜなら、要素ｘが根からみて左部分にある場合、要素ｘを根とした部分木を見た時に、すべての要素は、要素ｘの親の値より小さいから、2 分探索木は成り立つ。要素ｘが根より右にある場合も同様である。そのとき、すべての要

素は親の値より大きい。

次に C の場合は、要素ｘの右の子が根となる右部分木を見る。その右部分木の最小要素ｙを見つける。最小要素は、関数 findmin で見つける。その最小要素ｙが葉であれば、要素ｘを削除して、その要素ｙを要素ｘの位置に置き換える。また、要素 y に右の子（要素ｚ）があれば要素ｙの位置を、要素ｚに置き換え、要素ｘの位置を要素ｙに置き換える。（図 4.4.2 参照）

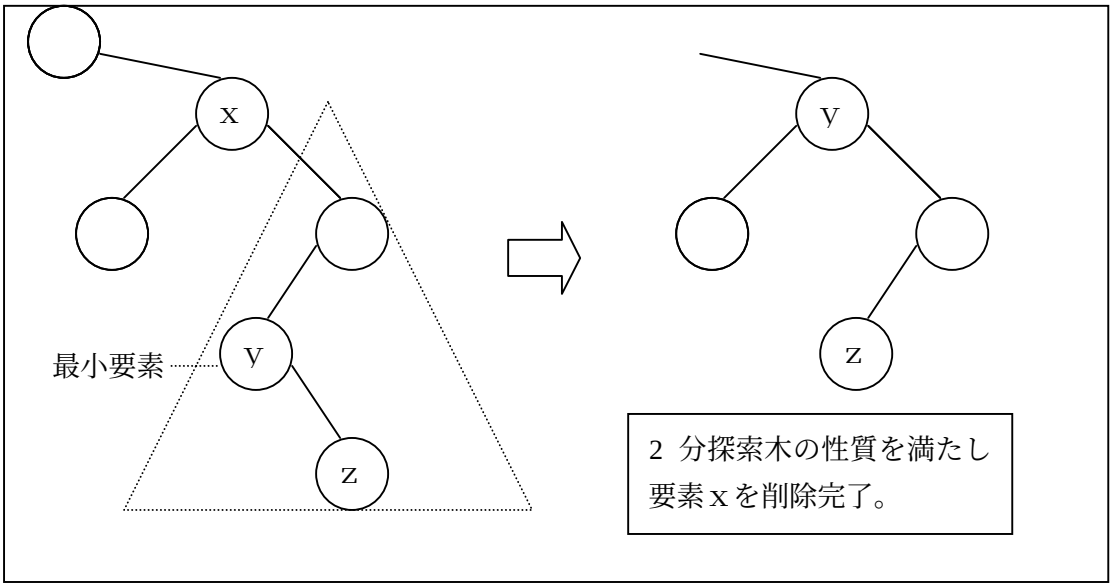


図４．４．２　要素の削除 C の場合

以上が要素の削除のアルゴリズムである。削除の場合を 2 分探索木の性質を満たすために削除したい要素を探索しなければならない。そして、その要素の位置（条件）によって、削除のアルゴリズムは変わるが、探索の手間とあまり変わらない。

以上が 2 分探索木の要素の操作である。2 分探索木で要素を操作する時、どの操作も木の高さだけ、繰り返しを行えばよい。よって、2 分探索木の時間計算量は、木の高さに比例していることがわかる。

2分探索木は､配列を4つ使って実現しているので､構築するのに少し手間がかかる。

４．５　ヒープとは

ヒープは完全２分木に要素をヒープ条件に従い登録したものである。ヒープ条件とは､“ 親の要素の値は子要素の値を超えない ” というものである。兄弟同士は、大小関係をもっていない。このことから、根の値が最小要素ということになる。そのため最小要素を取り出すのに適したデータ構造である。このようなデータ構造を半順序木といい、ヒープとは半順序木を配列で実現したデータ構造である。

ヒープを使って、要素の挿入と削除、と最小要素の参照ができる。また、ヒープになってない数列を、ヒープ化し、昇順に並び替えることができる。これをヒープソートという。これらの操作を行う。

完全２分木について説明しておく。完全２分木は、根から下のほうへ、また左の方から右へ順に要素を並べたものである。最後の段には、要素が全部詰まってなくてもいい。しかし、それまでの段には、要素が全部詰まってないといけない。

下の図はヒープを満たしている木である。

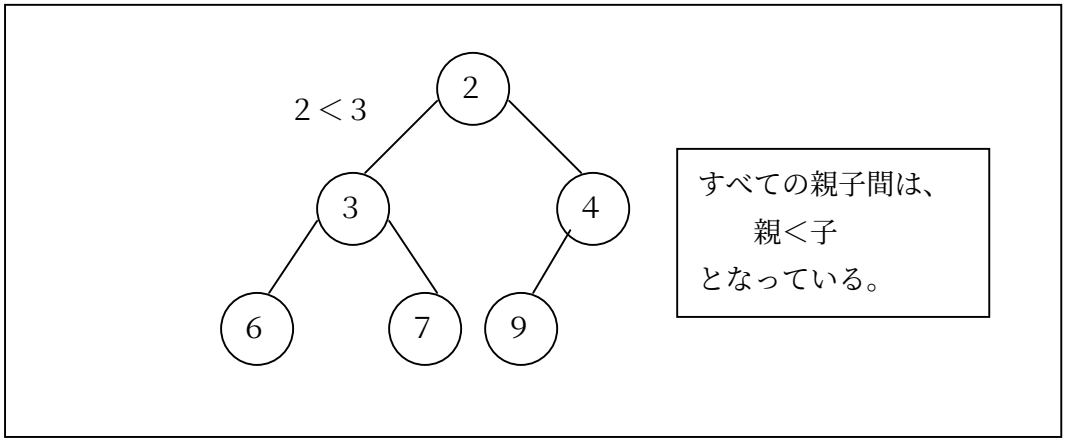


図４．５．１　木構造の概念図

４．６　ヒープの実現

ヒープは完全２分木なので、幅優先順を点番号として、１次元配列１つで実現することができる。なぜなら、ヒープでは要素間の位置関係が明らかであり、要素の削除や挿入は、木の最後の部分で行われ、木の構造が大きく変わることはないからである。よって図 4.5.1 の木は図 4.6.1 のようにあらわす。

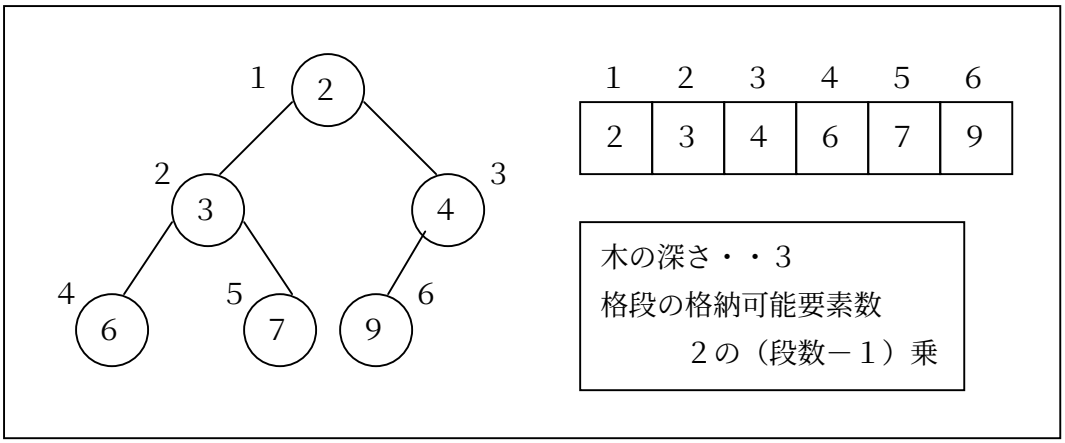


図４．６．１　木構造の概念図

１次元配列をヒープの木で表すとき、まず木の深さを求める。完全 2 分木なのでｋ段目までに登録できる要素数 N は「$N=2^k-1$」個である。これをｋについて解くと、「$k=\log_2(N+1)$」となる。要素数 N が十分大きいものとするとし、N で近似する。よって要素数がＮの木の深さは、「$\log_2 N$」となる。次に格段数の格納可能要素数は、「２の（段数－１）乗」で求めることができる。後はこの 2 つの数を使って、1 段目から順に表示していく。

例）図 4.6.1 の 1 段目

1 段目に入る要素数　・・・　２＾（１－１）＝１個

配列[1]を 1 段目に表示 ⟶ 2 段目へ

2 段目

2 段目に入る要素数　・・・　2 個

配列[2]から 2 個 2 段目に表示 ⟶ 3 段目へ

というアルゴリズムで、表示していく。これをプログラムで表すと、プログラムリスト２“ 要素の挿入ヒープ ”の関数 writeheap　となる。

また、配列[i]の子は、左が配列[i×2]で右が配列[i×2+1]（i=1~N/2）となる。また配列[i]の親は、配列[i/2]（i=2~N）となる。これは、完全 2 分木なので、このことが成り立つ。図 4.6.1 でも確認できる。

以上のように 1 次元配列をヒープで表すことができる。

４．７　ヒープの操作

【要素の挿入－ヒープ】

それでは、実際にヒープを操作してみる。まずヒープに要素を挿入する。ヒープに要素を挿入した後もヒープ条件を満たしていないといけない。よってまずヒープの最後に要素を挿入し、そこからその要素を、ヒープ条件を満たす位置まで移動させる。（プログラムリスト２“ 要素の挿入ヒープ ”参照）

このプログラムでは、ヒープの木となる配列は、あらかじめプログラム内で作っておく。下にその配列を示す。

(1, 2 , 3 , 4 , 5 ,6 , 7 , 8 , 9 ,10) ―― 配列 HEAP[ ]

{2, 9, 6,12,10,35,20,30,15,11} ―― 要素の値

この配列に新しい要素５を挿入するものである。

それでは、挿入部分のアルゴリズムを説明する。

①　新しい要素をヒープの最後（HEAP[11]）に挿入する。

② その要素の値と親の値を比較して、親より小さければ親の位置に移動する。
③ 親の方が小さくなるか、または根に達するまで、②を繰り返す。

これが、挿入のアルゴリズムである。ヒープで 1 件の要素の挿入は、最悪でも根まで比較すれば要素を挿入し、ヒープ状態になることができる。そのため時間計算量は、木の深さである、「$\log_2 N$」（要素数 N）に比例しているといえる。

【要素の削除－ヒープ】

次に最小要素の削除について説明する。（プログラムリスト２“最小要素の削除ヒープ”参照）要素の挿入の時に使ったものと同じヒープの木を使う。（プログラム内に記述）このプログラムは、ヒープの中から最小要素である、根の値を削除し、ヒープ条件を満たすように並び替えるものである。

それでは、削除部分のアルゴリズムについて説明する。

① 最小要素である根の要素を取り出す。
② ヒープの木（配列）の最後の要素を根に移動させる。
③ 左右の子の値の小さい方と比較し、子の値より大きければ子の位置に移動する。
④ 子の値の方が大きくなるか、または葉に達するまで、③を繰り返す。

以上が最小要素の削除のアルゴリズムである。この場合も最悪、根から葉まで、作業を繰り返せばよいので、時間計算量は木の深さに比例している。また、この作業を繰り返して最小要素を取り出していけば、昇順にデータを取り出すことができる。

【要素の整列－ヒープ】

ヒープを使って数列のソーティングを行う。これは、ある数列をヒープ化して、昇順に並び替えるプログラムである。全体のアルゴリズムとしては、先ほどの最小要素の削除を要素の数だけ繰り返していくというものである。それでは、プログラムの流れを説明する。

① 集合の入力 ・・・ キーボードより数値をいくつか入力する。そして配列 HEAP に順に格納される。９９９９で入力終了。

② ①の集合をヒープ化する・・・ ヒープ条件に従い、配列 HEAP を並び替える。それを木で表示する。

③　ソートを行う　　・・・　最小要素を取り出していき、取り出したあとの木を表示する。それを要素がなくなるまで繰り返す。

以上がこのプログラムの流れである。②のヒープ化の部分のアルゴリズムと③のソートの部分のアルゴリズムについて説明する。

②のアルゴリズム

(1) 配列の最後から、要素ｘとその要素の親の値と比較する。要素ｘの方が親より小さければ､親の要素と交換する｡親の位置は要素ｘのアドレスが[n]の時、[n/2]である。
(2) 次は最後の要素の、ひとつ手前の要素に対して、親と比較していき、ヒープ条件を満たすように根まで順番に繰り返す。
(3) 根まで行ったら(1)を最後の要素から順に根まで比較していく。
(4) すべての位置でヒープ条件が満たされたらヒープ化が完了となる。

③のアルゴリズム

(1) 根の値が最小要素なので、根の値を取り出す。削除された後も、ヒープ条件をみたすように並び替える。(要素の削除参照)
(2) (1)の作業を要素がなくなるまで繰り返す。
(3) (1)と(2)で取り出した要素を順番に並べると昇順に並ぶ。

以上がヒープソートのアルゴリズムである。ヒープを使えば、数列を昇順に並べ替えることができる。このときの時間計算量は、ソートの部分だけ考えると、要素の削除を要素数 N 回繰り返すというアルゴリズムなので､要素の削除 1 回の時間計算量は、木の高さが「$\log_2 N$」なので、これに N をかけた、「$N \log_2 N$」となる。時間計算量をオーダーで評価すると、

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

となる。ヒープソートは、整列アルゴリズムの中でも効率のよいものである。

ヒープ構造は、ヒープ条件を満たした完全 2 分木なので、1 次元配列で簡単に実現することができる。また要素の挿入と削除も時間計算量は、木の深さである「$\log_2 N$」ですむ。そして、ヒープを使った整列アルゴリズムは、効率もよく、集合をヒープ化するのもたいして手間はかからないので、ヒープはデータの整列によく使われるデータ構造のひとつである。

５．表構造

５．１　ハッシュ法とは

ハッシュ表は表構造のひとつのデータ構造である。表構造とは、私達が日常使っているものと同じ概念のものである。表構造の概念は、1 つ以上のフィールドを持ったレコードがあり、それを配列で並べたものである。表の列のことをレコード、行のことをフィールドという。図に示す。（図 5.1.1）

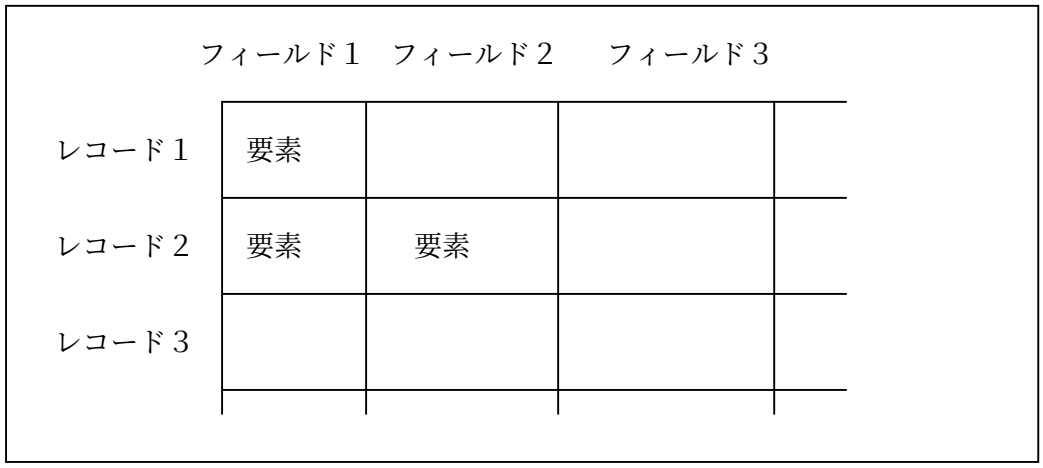


図５．１．１　表の概念

表構造の探索には、ハッシュ法以外にも、線形探索法、２分探索法などがあるが、今回は表構造の中でも使いやすく、効率の良いので、ハッシュ表について説明する。

ハッシュ表は、ハッシュによってデータを表に登録していくデータ構造であり、要素の操作が時間計算量 O(1)で行うことができるという優れた特徴をもつ。ハッシュとは、要素の値をハッシュ関数でレコードの格納位置を示す値に変換することである。そのため、ハッシュ表から要素を探索する時は、探索したい要素をハッシュ関数で変換するだけで、どのレコードに入っているかがわかる。（図 5.1.2 参照）しかし、1 つのレコードには、1 つ以上のフィールドを持っているので、1 つのレコードにいくつものフィールド（要素）がある場合がある。それを衝突といい、それを解決するためにハッシュ表には、２つの種類がある。それは、オープンハッシュ法とクローズドハッシュ法である。

オープンハッシュ法とは、ハッシュ表には、レコードへのポインタを格納する方法である。また、衝突がおこったときは、レコードを連結リストで、つなげていく。それに対して、クローズドハッシュ法とは、ハッシュ表にレコードを直接登録する方法である。衝突が起こったときは、再ハッシュの手順を定めておき、それに従い表の空いている場所を探す。再ハッシュの最も簡単なものは、衝突が起こったときその位置から後方に隣接要素を順に探索し、空いている表があればそこに要素を登録する。1 次

ハッシュ法がある。このクローズドハッシュ法は複雑なので、今回はオープンハッシュ法を使って、ハッシュ表を実現していく。

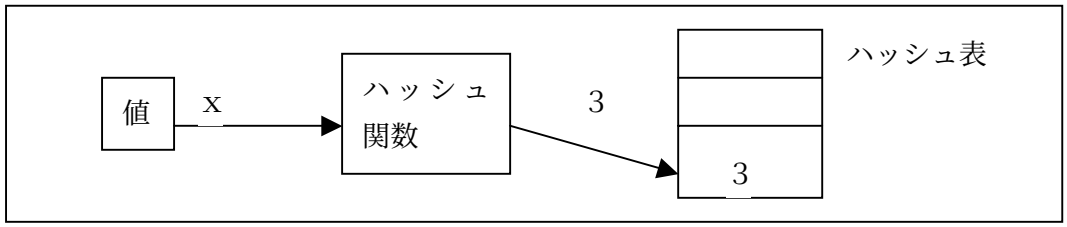


図５．１．２　ハッシュの概念

５．２　ハッシュ表の実現

オープンハッシュ法とは、同じハッシュ値(値をハッシュ関数で変換したもの)のレコードを、リストにするというものである。そして、ハッシュ表の要素には、リストへのポインタが格納される。またハッシュ表の要素のことをバケットという。

この方法を使うためのハッシュ関数は、衝突の回数が少ないほうが良い。そのためハッシュ関数は次のものを使う。要素の値をｘ、ハッシュ表のサイズを M とする。

$$H(x) = x \bmod M$$

Mod とは、剰余演算のことである。要素の値ｘをハッシュ表のサイズ M でわったものの余りが、ハッシュ表のレコードの格納位置を示す値となる。この関数は、計算の手間もかからず、ハッシュ値も偏らないので、よく使われる。

下にオープンハッシュ法の図を示す。(図 5.2.1)

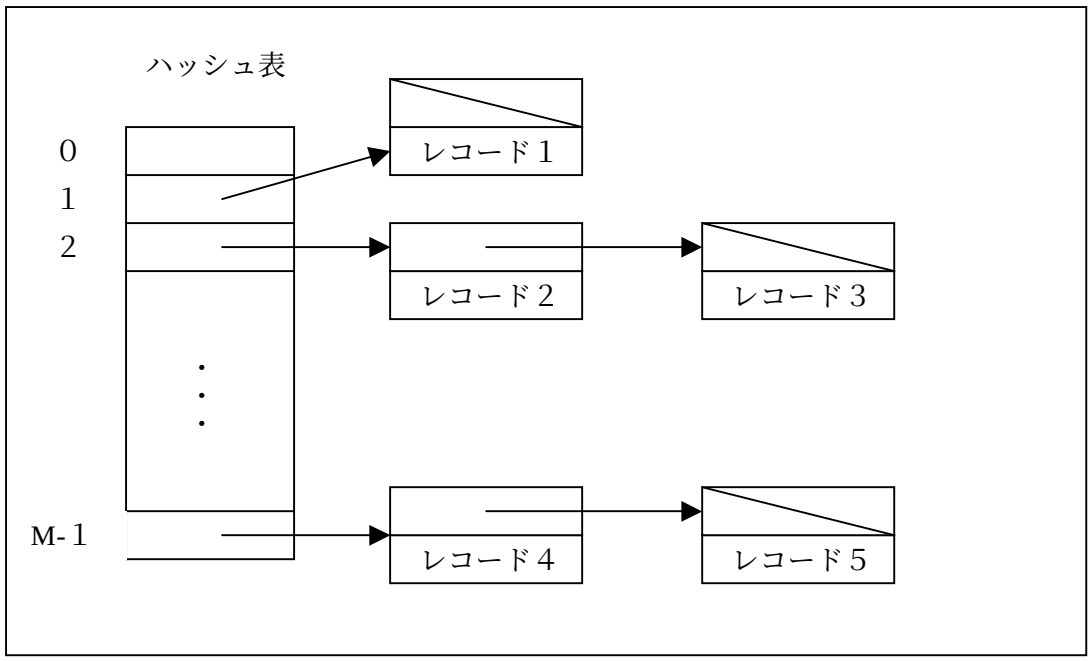


図５．２．１　オープンハッシュ法の概念図

まずハッシュ表に入れたい要素の値からハッシュ値を求める。(図 5.1.2) そのハッシュ値がハッシュ表の位置となる。ハッシュ値と同じ数字の列に、その要素のアドレス（配列）を記憶させる。すなわち、リストの先頭要素へのポインタをハッシュ表に登録するのである。ハッシュ値が同じ値の要素がきた時は、リスト構造を使ってつなげていく。そのため、ハッシュ表の１つの要素は、リスト構造の時と同じく、要素の値と、次の要素へのポインタによって構成されている。これを、配列を２つ使って実現する。要素の値用配列 VALUE、ポインタ用配列 NEXT で構成する。また、ハッシュ表は、配列 HASH を使って実現する。

それでは、実際にハッシュ表へ要素を挿入していく。(プログラムリスト３“ハッシュ表への要素の挿入”参照)

このプログラムは、オープンハッシュ法で、要素をハッシュ表に挿入していくものである。また、ハッシュ表のサイズは６とする。それでは、このプログラムの流れを示す。

① ハッシュ表の初期化 ・・・ ハッシュ表の中身を全部０にする。ハッシュ表は、配列 HASH[i]で表し i の範囲は 0~5 となる。

② 要素の入力 ・・・ キーボードより要素の値を入力する。

③ ハッシュ表への挿入 ・・・ ②の要素をハッシュ表へ挿入する。挿入後②へ戻り、②で９９９９が入力されるまで、②, ③を繰り返す。

④ ハッシュ表の表示 ・・・ ③でハッシュ作成されたハッシュ表を表示する。

以上がこのプログラムの流れである。③の要素を挿入する部分のアルゴリズムについて説明する。

まず、ハッシュ表である０～５のアドレスをもつ配列 HASH には、初期化によってすべて０が入っている。②で入力した値が３の場合を説明する。②で入力した値は、配列 VALUE に入る。このとき、要素１の値は３となる。

(1) ３をハッシュ関数でハッシュ値に変換する。ハッシュ表のサイズ６でわる。

３ ÷ ６ ＝０ ・・３

余りの３がハッシュ値になる。

(2) ハッシュ値が３なので、HASH[３]に要素１（要素の値３のある要素のアドレス）へのポインタを格納する。
(3) 次の要素はないので、要素１のポインタを０とする。結果図 5.2.2 のようになる。（図４）
(4) ハッシュ表へ挿入完了となり、要素の入力に戻る。

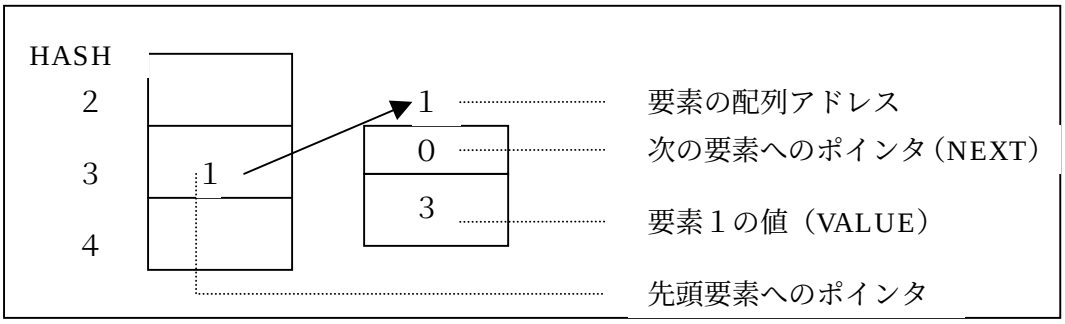


図５．２．２　ハッシュ表への挿入

これは、要素の衝突が起こってない状態である。つぎに衝突が起きるように、要素２に、ハッシュ値が３となる、２１を入力する。このときハッシュ表には、すでに、要素１へのポインタがはいっているので、衝突が起こる。その場合は、先にハッシュ表に入っている要素１へのポインタを、要素２のポインタとして NEXT[２]に１を格納する。ハッシュ表には要素２へのポインタとして２が入り、要素１と要素２がリスト構造でつながる。図 5.2.3 が衝突後のハッシュ表である。

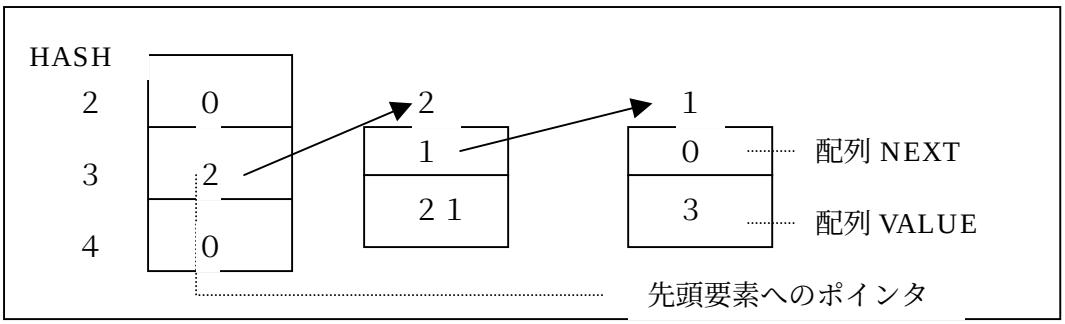


図５．２．３　ハッシュ表への挿入の衝突あり

以上のようにして、オープンハッシュ法では、ハッシュ表に要素を挿入していく。そのためハッシュ表の中はリスト構造でつながっていることがわかる。そのため要素を挿入する時は、ハッシュ関数でハッシュ化して、後はポインタのつなぎ変えだけでできるので、１つの要素を挿入する時の時間計算量は O(1)ですむ。O(1)というのは、要素の値に関係なく、定数回の作業で終わることである。

これがハッシュ表に要素を挿入する時のアルゴリズムである。このようにして、ハ

ッシュ表に要素を挿入していく。

５．３　ハッシュ表の操作

【要素の判定－ハッシュ表】

それでは、オープンハッシュ法を使って要素の探索を行う。ここでいう要素の探索は、ハッシュ表の中から指定した要素が存在するかを判定するものである。よって以降、要素の判定という。なぜなら、リストや２分探索木のように、存在する場所を見つけることは、ハッシュ表では困難である。

それでは、プログラムの流れを示す。(プログラムリスト３“ 要素の判定 ”参照)

① ハッシュ表の作成　・・・ “ ハッシュ表への要素の挿入 ”と同じ作業で、ハッシュ表に要素を挿入しハッシュ表を作成して、表示する。

② 判定したい要素の入力　・・・ 存在判定したい要素ｘをキーボードより入力する。

③ 要素を判定する　・・・ ハッシュ表の中に、要素ｘが存在するかを判定する。

④ 結果を表示　・・・ ハッシュ表内に要素ｘがあれば、“ 存在 ”を表示、なければ“ 存在しない ”を表示する。

以上がプログラムの流れである。つぎに③の判定部分のアルゴリズムを説明する。

(1) 要素ｘをハッシュ関数で、ハッシュ値に変換する。
(2) ハッシュ値で示された配列ＨＡＳＨに行き、ＨＡＳＨ内に格納されているポインタの指すリストを先頭から順にたどる。
(3) 要素ｘと同じ値が見つかれば探索成功、なければ探索失敗となる。

以上が要素判定のアルゴリズムである。このアルゴリズムは、集合が多くても、判定したい要素ｘをハッシュ化するだけで、ハッシュ表の位置がわかる。ハッシュ表には、ハッシュ値の偏りが余りなく、各バケットに一様に登録されていると考える。その場合各バケットには、要素数Nで、ハッシュのサイズMの時、平均N/M個のレコードがリストでつながっていることになる。よって要素の判定にかかる時間計算量は、O(1+N/M)となる。なぜなら、要素ｘの変換は１回だけでよく、あとは平均N/M個の

リストを先頭からたどればよいからである。N/Mが定数で押さえられるなら、時間計算量は、

$$T(N) = O(1)$$

となる。N/Mが定数で押さえられる場合とは、NとMが同程度の大きさ、もしくはMの方が大きい時である。定数とならない場合は、NがMより十分に大きくNの値が支配的になる場合である。その場合は、衝突がいくつも起こり、1つのバケットにいくつもリストがつながり、結果リストと同じになってしまう。そうなると、ハッシュ表の特徴が生かせなくなる。その対策として、Mの値を十分に大きくとるという方法がある。

【要素の削除－ハッシュ表】

つぎに要素の削除について、説明する。これは、ハッシュ表から指定した要素を削除するというものである。今回もオープンハッシュ法を使ってハッシュ表を実現する。

下が削除のプログラムのアルゴリズムである。

① 削除したい要素ｘの値をハッシュ関数で、ハッシュ値に変換する。
② ハッシュ値の示す配列ＨＡＳＨのリストから、要素ｘを探し、存在すればその要素のアドレスを記憶する。
③ そのアドレスが、ＨＡＳＨ内の値と同じなら先頭リストで、違ったら先頭リスト以外ということになる。
   (a) 先頭リストの場合は、要素ｘのポインタ（NEXT[x]の値）を配列 HASH 内に格納する。
   (b) それ以外は、直前要素ｙを探し、要素ｙのポインタ（NEXT[y]の値）と要素ｘのポインタ（NEXT[x]の値）を入れ替える。

以上が要素を削除するアルゴリズムである。要素の削除は、ハッシュ表から、削除したい要素を探索すれば、あとはポインタをつなぎ変えるだけなので、この時の時間計算量は、要素の判定(探索)と同じである。よって、

$$T(N) = O(1)$$

となる。

以上が要素の操作である。このことからもわかるように、ハッシュ表での要素の操作は、先に述べたある条件下では、時間計算量 O(1)で行うことができる。要素数 N がある程度予測できるような場合は、ハッシュ表のサイズ M も決めることができるので、その場合は、データ数に関係なくハッシュ表は有効なデータ構造であると

いえる。しかし、用途によっては、ハッシュ表は有効なデータ構造とならない場合もある。その場合とは、最小要素や最大要素の探索の場合や、要素を探索し、その要素の位置を知りたい場合などである。なぜなら、まず前者の場合は、全レコードをひと通り調べなければならない。また、後者の場合は、ハッシュ表では、要素の位置というもの自体明確になってないので、困難である。このように、いくつか弱点はあるが問題によっては、ハッシュ表を使うと時間計算量が少なく最も効率の良いデータ構造となる場合もある。

６．探索アルゴリズムの効率の評価

６．１　探索アルゴリズム

これまでの章で、リスト、2 分探索木、ヒープ、ハッシュ表と、様々なデータ構造があることがわかった。また、それぞれの機構や特徴なども理解した。そこで、これらのデータ構造をどう使い分けるのかということを考える。考えられるのは、プログラムで実現しやすいか、データを処理するときの効率がよいかということである。そこで、データ処理するときの効率に注目した。しかし、データを処理するといっても、いくつかの処理方法がある。例えば、要素の挿入や削除、探索や整列などがある。本稿では、データ構造で組織化したデータ群から、指定した要素のある場所を探すという、要素を探索する時の探索アルゴリズムの効率について検討した。ここでいう効率というのは、探索にかかる時間のこととする。

つぎにデータの探索に適しているデータ構造として、リストと 2 分探索木とを考えた。ハッシュ表は、要素の場所が明確ではないので今回の探索には適していないと考えた。そしてつぎに、この２つの探索アルゴリズムの、効率の比較の仕方であるが、データの種類や量の何について比較するというのも問題である。そこで今回は、数値データの量という面で効率の比較を行う。

よって今回は、数値データのデータ容量について、探索アルゴリズムの効率の比較を行う。それでは、まずリストと 2 分探索木それぞれの探索アルゴリズムについて考えてみる。

６．２　リストと 2 分探索木の探索アルゴリズム

(1) リストの探索アルゴリズム

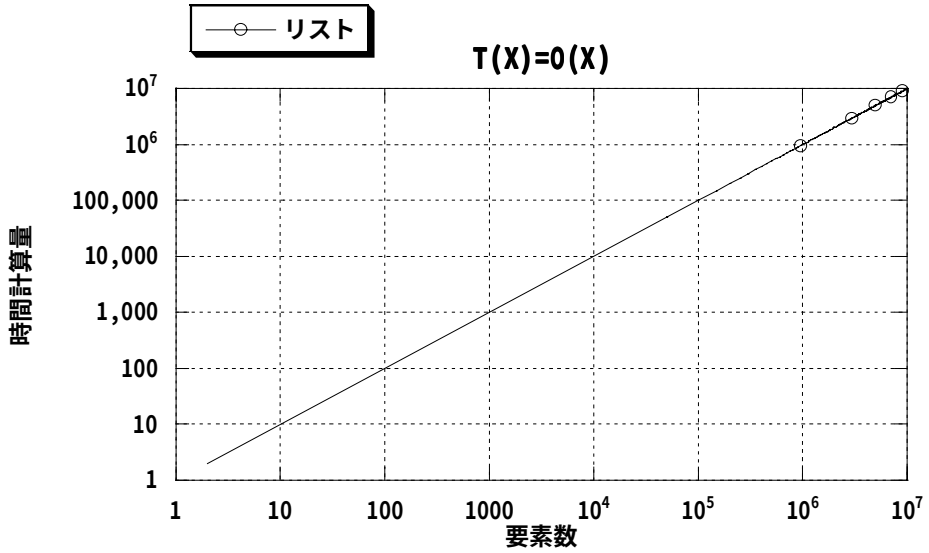
リスト構造は、第 3 章で述べたように、要素が線形に並んだデータ構造である。そのため指定した要素を探すときは、先頭要素から順に探索を行うというアルゴリズムである。要素数が N のときは、1 回目で見つかるときもあれば、最悪 N 回目で見つかるか、N 回目まで行って見つからないときもある。その時の時間計算量は、要素数 N に比例していることがわかる。また、指定した要素が、見つかる（リスト内の要素と指定した要素が一致する）確率は、1/N なので、探索の手間は、

$$\sum_{i=1}^{N} i*1/N = 1/N\sum_{i=1}^{N} i = 1/N*N(N+1)/2 = O(N) \quad \text{式６．２．１}$$

となる。これは、N に関する 1 次関数なので、オーダーを使って表すと、時間計算量は

$$T(N)=O(N) \quad \text{式６．２．２}$$

となる。要素数 N に比例していることがわかる。下がリストの増加傾向を表すグラフである。両軸とも対数で表し、要素数の範囲は 1～10^7 とする。（グラフ 1）



グラフ６．２．１（1～10^7）

(2) 2 分探索木の探索アルゴリズム

2 分探索木は、2 分木に 2 分探索木の規則にしたがって要素を登録した探索木である。2 分探索木の規則とは、親より小さい子は左に、大きい子は右にというものである。そのため要素の探索アルゴリズムは、根から木の高さ＋１だけ比較を行うアルゴリズムである。バランスを崩してない 2 分木であることを想定すると、要素数 N の 2 分木の高さは、「$\log_2$（N＋１）」となる。なぜなら、まず完全 2 分木での高さは $\log_2 N$ だからである。（4.6 参照）そこで、退化してない完全 2 分木に近い状態の 2 分探索木を考えると、2 分探索木の高さは、「$\log_2 N$」のオーダーとなる。また、実際には、要素の挿入のみによって作られた 2 分探索木は、完全 2 分木の高さよりも、4 割ほど増えるだけである。そのときの時間計算量は下の式 6.2.3 となる。詳しくは、参考文献３を参照。

$$T(n) = 2\sum_{i=2}^{n+1}\frac{1}{i} \le 2\int_{1}^{n+1}(\frac{1}{x})dx = 2\log_e(n+1) \le 1.4\log_2(n+1) \qquad \text{式６．２．３}$$

よって「$\log_2 N$」のオーダーといってもよい。しかし、退化して直線になった 2 分探索木は、リストと区別がつかず、時間計算量もリストのものと同じになる。このような場合は、特殊な場合であり実際ではこのようなことは、ほとんどないと考えてよい。2 分探索木は、バランスを保てるかが問題になる。

よって今回の 2 分探索木はバランスのとれているという条件下で、時間計算量を求める。要素をｎ個 2 分探索木に挿入する時の時間計算量から求める。最初に入力する要素ｘが i 番目の大きさならば、構成される 2 分探索木の、根の左部分木は i-1 個の節点を、右部分木は n-i の節点をもつ。よって

$$T(n)=\frac{1}{n}\sum_{i=1}^{n}(n-1+T(i-1)+T(n-i)) \qquad \text{式６．２．４}$$

となる。これを解くと、

$$\frac{T(n)}{n+1}=4\sum_{i=1}^{n}(\frac{1}{i})-2\sum_{i=1}^{n}(\frac{1}{i})-4+\frac{4}{n+1} \qquad \text{式６．２．５}$$

$$\approx 2\log_e n-4+\frac{4}{n+1} \qquad \text{式６．２．６}$$

となり、これを近似式$\sum_{i=1}^{n}\frac{1}{n}\approx 2\log_e n$　を用いた。結果、要素を挿入するときの 1 回の比較回数は、

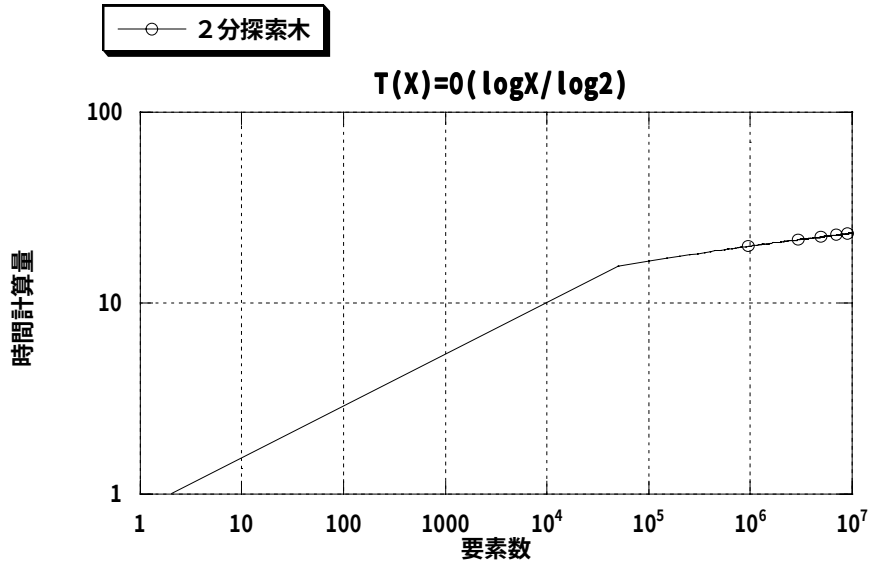
$$T(n)\approx 2\log_2 n \qquad \text{式６．２．７}$$

となる。この比較回数は、要素の探索の手間と等しいので、探索にかかる時間計算量は、式 6.2.7 となる。

結果、要素数が N のとき 2 分探索木の時間計算量は、

$$T(N)=O(\log_2 N) \qquad \text{式６．２．８}$$

となる。下が 2 分探索木の増加傾向である。両軸とも対数で表し、要素数の範囲は１～10^7とする。（グラフ 6.2.2）



グラフ６．２．２（１～10^7）

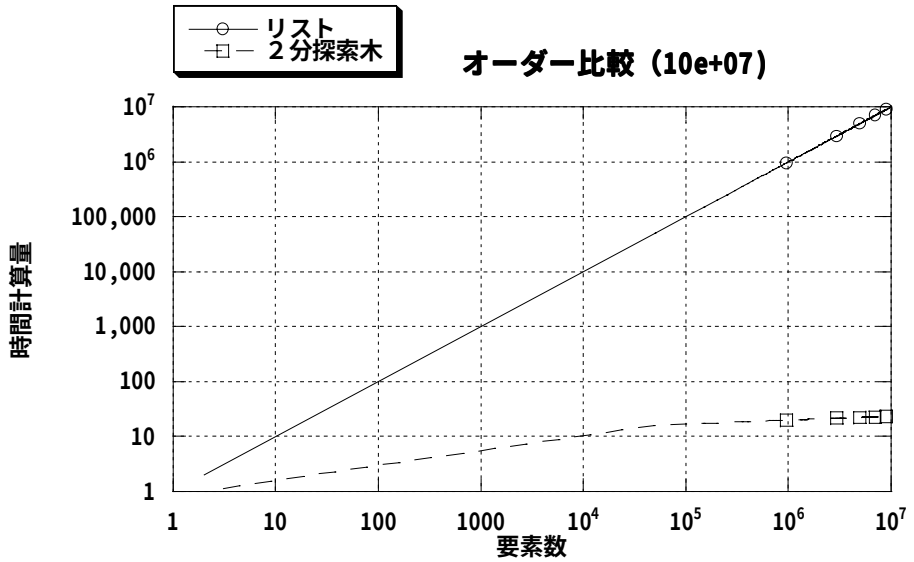
６．３　データ容量と効率（時間計算量）の関係

それでは、リストと 2 分探索木の時間計算量を使って、実際に要素数の変化との関係を考えてみる。それぞれの時間計算量を下に示す。

リスト	T(N)＝O(N)	（P29 式 6.2.2 参照）
2 分探索木	T(N)= $O(\log_2 N)$	（P31 式 6.2.8 参照）

それぞれ、要素数に値をいれて増加傾向をしらべる。今回はリストとの比較なので、実際の時間に近いようにするため log の底を 2 とした。ちなみに、増加傾向だけを見る場合だけなら底の値は関係ない。

下が要素数の変化による、時間計算量の増加傾向のグラフである。（グラフ 6.3.1）



グラフ６．３．１　（1～10^7）

それでは、このグラフを使って評価していく。

リストはデータ容量に対して単純比例しているので、データ容量が大きくなるにつれ同じぐらい大きくなっている。それに対して、2 分探索木は、log で比例しているので、データ容量が大きくなっても、リストほど増加していかない。そのためデータ件数が 10 ぐらいと少ない時は、あまり差がないが、データ件数が増えていくに伴って、差が大きくなっていく。付録に要素数が 1～100 の時のグラフを、リスト(付録グラフ１), 2 分探索木(付録グラフ 2)、2 つ比較のグラフ(付録グラフ 3)のそれぞれを記載する。付録参照。

結果２分探索木はデータ件数 N が２乗になっても、時間計算量が２倍にしかならないのに対して、リストはデータ件数が２乗になると時間計算量も２乗になる。そのためデータ容量が多い場合は、2 分探索木の方が時間計算量は少ないので、効率がよいといえるだろう。しかし、2 分探索木を構築するためのプログラムは、リストを構築するプログラムより複雑なのでデータ容量が少ない場合は、時間計算量の差も

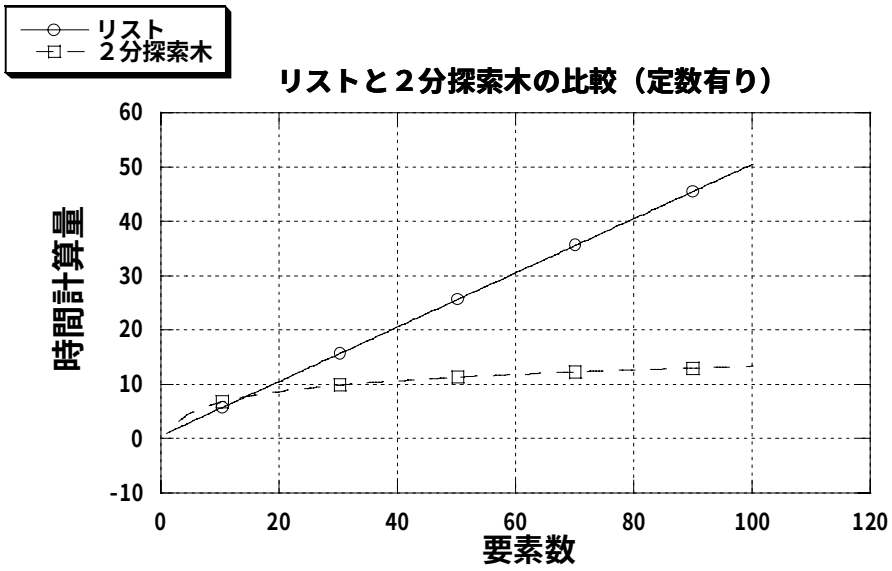
小さいのでリストを使ったほうが、総合的な効率を考えた場合有効である。２分探索木を構築するプログラムは複雑なうえに、木のバランスを保つような要素の入力の順番が必要になる。それは、入力した順に、根から要素が挿入されていくからである。そのため、扱うデータにある程度の順位がついていないと２分探索木の性質を使えない場合がある。よってバランスのとれた 2 分探索木なら、リスト構造よりも効率がよいと言えるだろう。

次に要素数が少ない時は、オーダーで評価するより、各項の定数係数まで考慮した値の方が重要である。リストと 2 分探索木それぞれの、定数係数まで書いた式を下に示す。（要素数 N）

リスト　　T(N)=(N+1)/2　　（P29　式 6.2.1 参照）

2 分探索木　　$T(N)=2\log_2 N$　　（P31　式 6.2.7 参照）

この式で要素数 1～100 までの時の時間計算量を比較する。（グラフ 6.3.2）



グラフ６．３．２（1~100）

定数を考慮した時、要素数が 16 以下まではリスト構造の方が、時間計算量が小さく効率が良い。このように、要素数が少ない時は、定数係数まで考えた方が重要な場合もある。しかし、これは計算だけで評価する場合だけであり、実際計算機で処理する場合は、要素数が 16 程度ならどちらのデータ構造で探索してもほとんど差はない。なぜなら、実際の時間は 10^{-5} や 10^{-6} と一瞬なので、われわれが差を感じることはない。

６．４　実際のデータを使っての評価

これまで、オーダーによる時間計算量で探索アルゴリズムの効率を評価してきた。では実際の計算機ではデータ数によって、処理時間はどの様に変化していくかを説明する。また、オーダーで予測した増加傾向は正しいかということも考察する。

今回はリスト構造のデータの探索アルゴリズムを UNIX の time コマンドを使って、実際の処理時間を求める。

【測定方法】

使用するプログラムは、プログラムリスト４のプログラムを使用する。このプログラムはリスト構造を構築し、要素を探索するものである。3 章の“リストの探索のプログラム”を一部変更したものである。要素の値は、配列のアドレスをそのまま要素の値として用いる。よって、下のようなリストがプログラム内で構築される。（図 6.4.1）

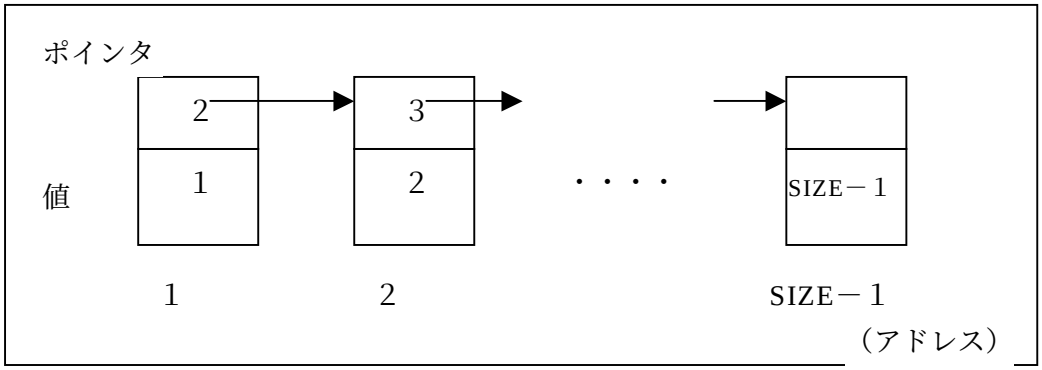


図６．４．１　リンク構造での実現

そのため、SIZE の値を変えることで、要素数（リストの数）を増加（減少）させることができる。これでリストを構築することができる。このリストからキーボードで指定した要素を探索する。指定する要素は、リストの最後まで、比較を行うために、リスト内にない値を入力する。今回は、SIZE の値を入力した。また探索アルゴリズムは 3 章のリストの探索と同じである。

上記のプログラムを time コマンドで処理時間を測定していく。time コマンドとは、指定したコマンドの実行時間を計測し、また実行に関係する情報を表示するコマンドである。

(使用例)

%time コマンド名

0.3u 0.1s 0:05 10% 0+0k 0+0io 0pf+0w

0.3u：ユーザ使用の CPU 時間

今回はユーザ使用時間を測定する。

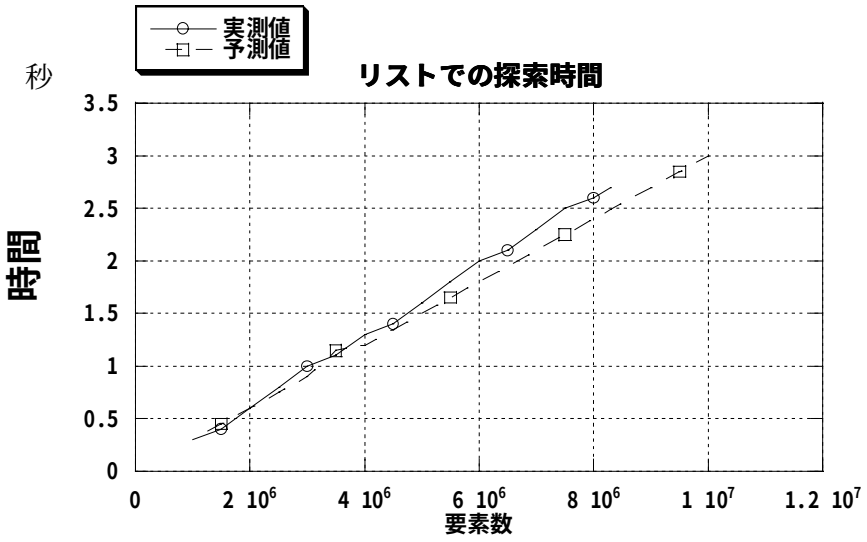
【測定】

それでは測定をはじめる。しかしここで問題なのは、time コマンドで表示する時間は、10^{-1} 秒までしか測定できないということである。そのため、要素数が少ない場合は、表示が“ 0.0”になってしまう。そのため今回は要素数 10^6から測定する。

また、要素数が 10^7近くになると、実行すると要素が多すぎて“ core ”になってしまい、実行できなくなる。そのため今回の測定の範囲は、1.0×10^6~8.3×10^6（”core”になる限界の値）とする。この範囲で実行時間と増加傾向を評価していく。

【測定結果】

測定結果をグラフに示す。（グラフ 6.4.2）



（グラフ６．４．２）

【グラフの評価】

まずグラフの予測値であるが、リスト構造の探索アルゴリズムの時間計算量は、

$$T(N)=O(N) \quad \cdots\text{（式６．４．１）}$$

なので要素数 N に 1 次関数で比例していることになる。またこの時間計算量は要素を探索するのに何回（何個の要素）と比較するかというものなので、計算機が 1 回の比較にかかる時間をグラフの傾きとして、O(N)にかけなければならない。そこで、測定範囲の最初の測定値 1.0×10^6のとき 0.3 秒なので、少し強引だがこの数字を使って予測値のグラフの傾きを求める。

オーダー評価の場合 N が 1.0×10^6 の時、時間計算量も 1.0×10^6 となる。そこで、その時の時間計算量は、測定値の時間を使う。よって 1.0×10^6 の時 0.3 秒となる。そして、傾きを a として次の式を立てる。(実際では a の値は計算機の 1 回の処理時間と実数係数のことである。)

$$T(1.0\times10^6)=a\times1.0\times10^6 \quad =0.3 \quad \cdots\text{(式６．４．２)}$$

これを a について解くと、

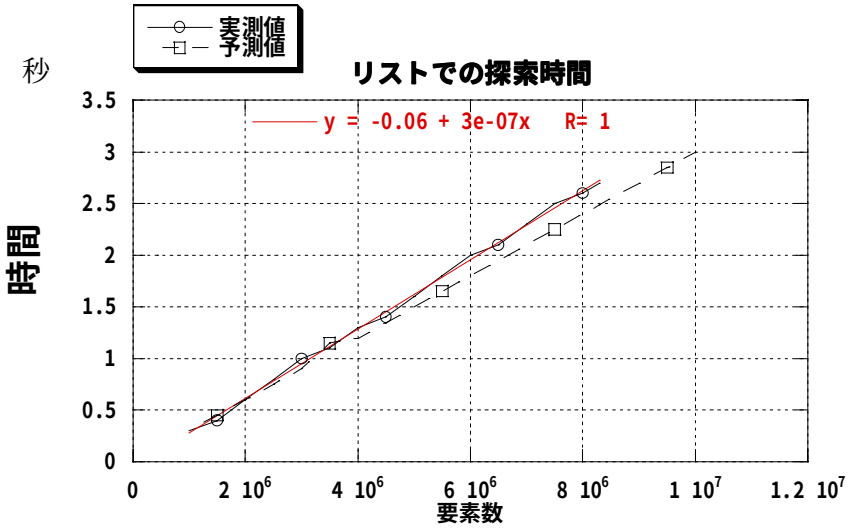
$$a=3.0\times10^{-7}$$

となる。よって予測値の関数は

$$T(N)= 3.0\times10^{-7}\ N \qquad \cdots\text{(式６．４．３)}$$

となる。これが、予測値の式である。

それでは、実数値を評価していく。実数値はほとんど 1 次関数に近い形で増加していっているのがわかる。それを証明するために、この実数値のグラフに回帰直線を引く。(グラフ 6.4.3)



(グラフ６．４．３)

その結果、回帰直線の式は、

$$y=3.0\times10^{-7}x-0.06 \qquad R=1 \quad \cdots\text{(式６．４．４)}$$

となる。R=1 なので、ほとんど式 4 に近いということになる。よって実測値の式は、

$$T(N)= 1.0\times10^7N-0.06 \quad \cdots\text{(式６．４．5)}$$

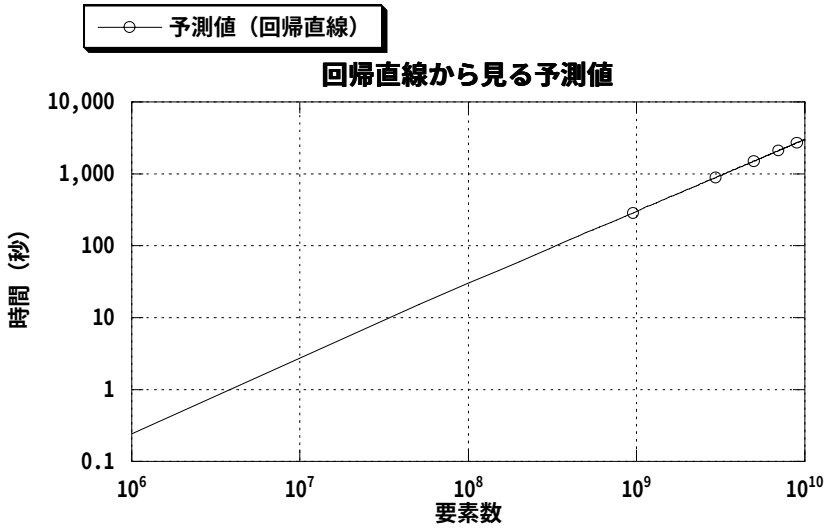
となる。

よって、リストの探索アルゴリズムの処理時間は、1 次関数で増加していっていることになる。

次にこの実数値(式 6.4.5)と予測値(式 6.4.3)の増加傾向を比較してみる。式 6.4.5 と式 6.4.3 は、ともに 1 次関数のグラフである。傾きもともに 3.0×10^{-7}

なので、実数値と予測値のグラフは、切片が－0．06 違うだけで近似しているといえる。そのため、オーダーで評価した増加傾向は、実数値との差はほとんどないので、正しい評価方法だということがいえる。

また、10^7を超えると時間は、実数となりはじめ、10^8では 30 秒、10^9では 300 秒と大きく時間がかかる。そのため、リストでは10^7を超えるようなデータを扱うのには、時間がかかりすぎてしまう。（グラフ 6．4．4 参照）



（グラフ６．４．４）

【リストの探索アルゴリズム】

リストの探索アルゴリズムは、要素数が 1.0×10^6 あたりから 10^{-2}秒以内で押さえられなくなり、処理に時間がかかってくる。そのため要素数が 1.0×10^7 ぐらいになると、3 秒近く時間がかかってくるので、処理時間の遅さを感じることができるようになる。そのため、1.0×10^7を超えるような要素数であれば、リスト構造では、時間がかかってしまうので適したデータ構造とはいえないだろう。

それに対しバランスのとれた 2 分探索木なら、時間計算量は「$\log_2N$」で押さえられる。要素数が 1.0×10^7でも、比較回数（時間計算量）は約 23（グラフ 6.2.2 参照）である。1 回の比較にかかる時間を、リストと同じ 3.0×10^{-7}秒と仮定すると、処理時間は、6.9×10^{-6} 秒なので、要素数が 1.0×10^7でも一瞬で探索することができる。（実際は 1 回の比較にかかる時間は大きく、リストよりも実数係数も大きい上に、複雑なので 3.0×10^{-7}秒よりかかる。しかし、それほど差はない。）

よって 1.0×10^7を超えるような要素は、2 分探索木などのデータ構造を用いたほうが、時間がかからないので効率がいい。もちろん 2 分探索木がバランスよく構築できる場合である。

７．結論とまとめ

７．１　データ容量に対してどのデータ構造が有効か

以上の考察から、数値データの容量に対してどのデータ構造が有効かということを今回の結論とする。それは、

“ データ容量に応じたデータ構造を用いる。”

ということである。それでは、データ容量に応じたデータ構造を具体的に説明する。

まず、データ容量が 1.0×10^{6} や 1.0×10^{7} 以上と膨大なときは、リスト構造よりも 2 分探索木を使ったほうが有効である。それは、2 分探索木は「$\log_2 N$」に比例して増加していくので、例えば 1.0×10^{7} の時、時間計算量は約 23、またデータ容量が２乗になっても、時間計算量は２倍にしかならない。それに対してリスト構造は要素数 N に 1 次関数で比例して増加するので、1.0×10^{7} の時、時間計算量 1.0×10^{7} となり、データ容量が２乗になると、時間計算量も２乗になってしまうからである。またデータ容量が小さい時は、さほど差は表れず、1.0×10^{6} 以内なら 0.3 秒以内で押さえられるので、プログラムもしやすいリスト構造を使ったほうが効率はいい。

オーダーで評価した場合は、増加傾向での比較になり、データ容量が小さい時もグラフからは 2 分探索木の方が有効に思える。しかし容量が小さい場合は、定数係数まで考慮した方が重要になる時もある。それは、時間計算量の小さいアルゴリズムは一般的に、複雑であり定数係数が大きくなる傾向があるからである。このようにデータ容量によって、データ構造を使い分けるということがわかった。

しかし、これは 2 分探索木のオーダーが「$\log_2 N$」で押さえられた場合である。ある集合を組織化して探索する時、データ容量が大きいので 2 分探索木で組織化するとする。しかし 2 分探索木を構築する時に問題になることがある。それは、根の値と挿入する順番である。2 分探索木は挿入した順に根から要素が挿入されるので、最小値を最初に入力すると根の値が最小値になる。根の値がその集合の最小値になった場合、その 2 分探索木は、右に伸びていくだけしかなくなり、時間計算量が「$\log_2 N$」のオーダーで押さえられなくなる。そのために、根の値にはその集合の中間値だと、ある程度のバランスは保てるので、「$\log_2 N$」のオーダーで押さえることができる。よって 2 分探索木を使って集合を組織化する時は、集合のある程度の順位がわかっていないと 2 分探索木のバランスが保てなくなり性質を生かしきれなくなる。

そのため、次のことがいえる。

・2 分探索木は、データ容量が 1.0×10^{7} と膨大で、集合にある程度の順位付けがされている場合 2 分探索木は有効である。またデータ容量が大きくても集合の順位がわからないものは、2 分探索木のバランスが崩れる可能性があるので、リスト構造を使ったほうが有効となる場合もある。

・リスト構造は、時間計算量による効率はあまりよくないが、リストを構築するのは２分探索木より容易なので、データ容量が 1.0×10^{7} 以内のような膨大でないとき

は、実際の時間は３秒以内とさほどかからないので有効である。

今回は、データ容量に対する時間計算量の効率について、採用するデータ構造を考えた。しかし、実際にプログラミングを行う際、採用するデータ構造とアルゴリズムの決定には、時間計算量以外にもプログラムの使用頻度や、実現のしやすさなど総合的に考慮しなければならない。例えばデータ容量が大きかったので、2 分探索木を使って探索を行った。しかし、2 分探索木を構築するプログラムは容易ではなく、またそのプログラム自体 1 回しか使わないような場合は、効率が多少悪くても、プログラミングのしやすいデータ構造やアルゴリズムを使ったほうが、全体的な効率はよくなる。そのためデータ構造とアルゴリズムの決定は、いくつかの要素から決定することが大切である。

このように、与えられた問題や状況に対して適したデータ構造とアルゴリズムを採用することが問題解決の本質であり、アルゴリズムの効率を上げるための最善の策であるといえる。

以上を今回の結論とする。

７．２　まとめ

今回の最初に立てた下の 2 つの目的についてまとめる。

- データ構造のそれぞれの機構や性質などを理解すること。
- データ構造の探索アルゴリズムの効率性を数値データの容量について評価し、データ量に対してどのデータ構造が有効かを考える。

最初の目的について説明する。最初の目的はデータ構造の機構や性質を理解するということである。データ構造とは、ばらばらなデータ群に対して、探索などのデータ操作をしやすくするために組織化するための概念である。データ構造で、データを組織化することで、効率よくデータを扱うことができる。それぞれのデータ構造について説明していく。

【リスト構造】

リスト構造は、要素の値とポインタによって構成された１つの要素が、いくつも１列に並んだデータ構造である。また、探索アルゴリズムは、先頭要素から順に探索していくという力ずくの方法と呼ばれるアルゴリズムである。そのため時間計算量は、要素数に比例する。

【木構造】

木構造である２分探索木の１つの要素は値と親へのポインタ、左右リンク先へのポインタの４つで、構成されている。その要素が、根より小さいものは左部分木へ、大きいものは右部分木へというルールで作られた２分木のデータ構造であ

る。この探索アルゴリズムは、根の値から比較し、左右どちらかの子をたどるというものなので、時間計算量は、ある条件下では木の高さ比例している。ある条件というのは、木のバランスが取れているときであり、バランスをくずして退化したものはリストと同じになってしまう。バランスの保たれた２分探索木は、要素の探索・挿入・削除が効率よく行うことができる。

もう１つの木構造であるヒープは、数値データを昇順に整列するときによく使われるデータ構造である。その構造は完全２分木に、ヒープ条件に従い要素を登録したものである。ヒープ条件とは、“ 親の値は子の値を超えない、また兄弟間の大小関係はない。” というものである。そのため根の値がそのデータの最小要素となる。またデータの整列アルゴリズムは、根の要素（最小要素）を順に取り出し残った要素をヒープ化することを、要素がなくなるまで繰り返していくと、データを昇順に整列することができる。そのときの時間計算量は、要素数 N の時、ヒープ化を N 回繰り返したものである。１回のヒープ化にかかる時間は木の高さに比例しているので、木の高さと要素数をかけた「$N\log_2 N$」となる。
プログラムで実現する時は、完全２分木なので１次配列１つで実現できる。

【表構造】

表構造であるハッシュ表は、要素の操作が O(1)で行えるデータ構造である。今回は、オープンハッシュ法を使った。オープンハッシュ法はハッシュ表の中には、その列に入る要素へのポインタが格納される。同じ列に要素がいくつか入り衝突が起こった時は、リストでつながっていく。ハッシュ表の構築は、要素の値をハッシュ関数でハッシュ化した値が、ハッシュ表に格納される位置を示す。そして、その値と対応したハッシュ表に、その要素へのポインタを格納する。

要素の操作が O(1)で行える理由は、例えばハッシュ表からある要素の存在を判定する時、探索したい要素をハッシュ関数で計算するだけで、その要素のあるハッシュ表の列がわかるからである。後は、ハッシュ表内のポインタに従い、リストをたどればよい。そのときの条件として、ハッシュ表に要素が偏ってないことを仮定している。偏っている場合は、衝突が多くなり、リストでつながる部分が長くなるので、O(1)で行うことはできず、リスト構造と同じ O(N)になってしまう。

次に２つ目の目的について説明する。今回は、リスト構造と２分探索木の探索アルゴリズムをオーダー評価で比較した。その結果を示す。

- データ量が 1.0×10^7以上と膨大な場合は、バランスが保てていたら 2 分探索木のほうが有効である。
- データ容量が 1.0×10^7以下の小さい場合や 2 分探索木のバランスが保てない場合は、リスト構造のほうが有効である。2 分探索木がバランスを保つために

は、扱う集合にある程度の順位がついていると、バランスを大きく崩すことはない。

また今回は、リスト構造の探索アルゴリズムの、実際の時間を計測した。その結果、オーダーで予測した増加傾向の正確さを証明することができた。また、要素数が 1.0×10^7 になると 3 秒近く時間がかかるので、それ以上はリストで扱うには、実際の時間がかかってしまうので、要素数が 1.0×10^7 以上になるとリスト構造は有効でないと考えた。それが先に述べた、“ データ量が 1.0×10^7 以上と膨大な時は、バランスが保てていたら 2 分探索木のほうが有効である。” ということの理由である。

今回はリストと 2 分探索木の効率を比較し、数値データの容量に対して有効なデータ構造を考えた。しかし、データ構造には他にもいくつもある。例えば、2 分探索木に更に工夫を加え、バランスを保てるようにした AVL 木がある。これは、最悪でも木の高さが $O(\log_2 N)$ になる。他には 2 分探索木より木の高さを低く押さえることのできる、多分木をベースにした B 木がある。これを使えば、データベースなどの膨大なデータを扱う時、2 分探索木よりも効率よく扱うことのできる。このように、リストや 2 分探索木以外にもいくつもデータ構造があり、それぞれに特性をもっている。様々なデータ構造の特性や機構を理解し、与えられた問題やデータ容量に対して、適したデータ構造とアルゴリズムを採用することが、問題解決の本質であり、プログラミングする上で大切なことだと思う。そのためにこれからも様々なデータ構造を研究していきたいと思う。

最後に今回使用した C 言語のプログラムは、データ構造とアルゴリズムを理解するために、リンク構造のあるデータ型をつかわず、配列の組み合わせでリンク構造を実現していった。そこで、リンク構造を持つデータ型である構造体”NODE”型を、使ってリストの探索のプログラムを作った。（プログラムリスト６）これまで、配列を 2 つ使って実現していた 1 つの要素も構造体”NODE”型を使えば簡単に実現することができる。また、配列を 4 つ使って実現していた、木構造の 1 つの要素も構造体”NODE”型を使えば簡単に実現することができる。

このように、C 言語には豊富なデータ型が備わっているので、かなりの種類のデータ構造を記述することができる。なぜならデータ構造は、プログラム言語に備わっているデータ型を組み合すことで実現できるからである。そのため、データ型が豊富で表現力の豊かな C 言語は、データ構造の実現には適している。これからもデータ構造の研究をするために適している言語として私は C 言語を使い、また C 言語を更に勉強していく。

８．参考文献

１．C アルゴリズム全科：　千葉　則茂、村岡　一信、小沢　一文、海野　啓明
近代科学社、2000 年第 6 版

２．C で学ぶデータ構造とアルゴリズム：　杉山　行浩
東京電機大学出版局、1997 第 2 版

３．アルゴリズムとデータ構造：　平田　富夫
森北出版株式会社、1994 第 6 版

４．C によるアルゴリズムとデータ構造：　茨木　俊秀
株式会社　昭昇堂、1999 初版

謝辞

指導教員である、高知工科大学助教授山本哲也先生には、研究全般にわたってお世話になり、さらには本稿を書くにいたっても助言をいただいた。ここに深く感謝いたします。また、卒業研究発表を評価していただいた、河津教授、河東田教授、さらに参考文献に示した著書を参考にさせていただいた著者の方々、ならびに山本研究室の室員にも、ここに感謝の意をあらわしたい。

プログラムリスト１

１．１　要素の検索

```
#define SIZE 100

void main(void)
{
  int VALUE[SIZE+1],NEXT[SIZE+1];
  int head,
      tail;
  int newcell,i,q,p,x;

  newcell=1;
  for(i=1; i<SIZE; ++i) NEXT[i]=i+1;
  NEXT[SIZE]=0;
  /*input      */
  head=0;
  printf("?");   scanf("%d",&x);
  while(x!=999){                                    ┐
   p=newcell; newcell=NEXT[p];                      │
   VALUE[p]=x;                                      │
   NEXT[p]=head; head=p;                            │  ①
   printf("? ");   scanf("%d",&x);                  │
  }                                                 ┘
  printf("? "); scanf("%d",&x);                          ・・・②
  /*serch       */
  q=0;                                              ┐
  p=head;                                           │
  while(p!=0){                                      │  ③
    if(VALUE[p]==x) {q=p; p=0;}                     │
    else                  p=NEXT[p];                │
  }                                                 ┘

  printf("%d¥n",q);                                     ・・・④
 }
```

１．２　要素の挿入

```
/*   insert2.2(3)      */
/*                        */
#define SIZE 20

void main(void);
void insert(int q, int x);
void writelist(void);

/*   area of cell         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;
int newcell;

void main(void)
{
   int q,x;
   head=7;   newcell=6;
   writelist();                                          ・・・①
   /*   insert       */
   x=12;
   q=3;
   insert(q,x);                                          ・・・②, ③
   writelist();                                            ・・・④
}

void insert(int q, int x)
{
      int p;

      p=newcell; newcell=NEXT[p];
      VALUE[p]=x;
      if(q==0){
```

```
            NEXT[p]=head;   head=p;   }
       else{
            NEXT[p]=NEXT[q]; NEXT[q]=p;
          }
}
void writelist(void)
/*      print of list      */
{
    int p;

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

１．３　要素の削除

```
#define SIZE 9

void main(void);
void delete(int q);
void writelist(void);

/*  list                    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();                                    ・・・①

    q=6;
    delete(q);                                      ・・・②, ③
    writelist();                                    ・・・④
}

void delete(int q)
{
   int p,s;

   if(head==q) head=NEXT[q];
   else {
     p=head;
     while(p!=0){
```

```
            if(NEXT[p]==q) {s=p; p=0;}

            else          p=NEXT[p];
          }
      NEXT[s]=NEXT[q];
      }
      NEXT[q]=newcell; newcell=q;
   }

void writelist(void)
{
     int p;

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

プログラムリスト２

２．１　最小要素の探索

```
#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();
    v=findmin(root);
    printf("最小要素の値は %d ¥n",VALUE[v]);
}

void buildtree(void)
{
    int v,e,l,r;

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

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

２．２　２分探索木の要素の探索

```
#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];
#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();                                    ・・・①
    v=findmin(root);
    printf("最小要素の値は %d ¥n",VALUE[v]);
}

void buildtree(void)
{
    int v,e,l,r;

    printf("v e l r = "); 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("v e l r = "); scanf("%d%d%d%d",&v,&e,&l,&r);
          };
       printf("root      = "); scanf("%d",&root);
}

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

２．３　2 分探索木の要素の挿入

```
#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],
     RIGHT[SIZE+1],LEFT[SIZE+1];
int CELLSTACK[SIZE+1];
int root,top;

void main(void)
{
    int i,x;

    for(i=1; i<=SIZE; ++i) CELLSTACK[i]=i;
    top=SIZE;
    buildtree();                                    ・・・①
    dfsearch();                                     ・・・②
    printf("x = "); scanf("%d",&x);                 ・・・③
    insert(x);                                      ・・・④
    dfsearch();                                     ・・・⑤
}

void buildtree(void)
{
  int v,e,l,r;

  printf("v e l r = "); 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("v e l r = ");scanf("%d%d%d%d",&v,&e,&l,&r);
      }
    printf("root     = "); scanf("%d",&root);
}

void dfsearch(void)
{
    int i,v;
    int VISIT[SIZE+1];

    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){
        if (LEFT[v]!=0 && VISIT[LEFT[v]]==0){
                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;

        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();
        VALUE[e]=x;
        LEFT[e]=0;    RIGHT[e]=0;
        v=search(x);
        FATHER[e]=v;
        if(x<VALUE[v])    LEFT[v]=e;
        if(VALUE[v]<x)    RIGHT[v]=e;
}
int search(int x)
{
        int flag;
        int v;

        flag=0;
        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)
{
        int e;

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

```
}
```

２．４　2 分探索木の要素の削除

```
#define SIZE 100

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];
int CELLSTACK[SIZE+1];
int root,top;

void main(void)
{
    int i,x,v;

    buildtree();                                    ・・・①
    dfsearch();                                     ・・・②
    printf("x = "); scanf("%d",&x);                 ・・・③
    delete(x);                                      ・・・④
    dfsearch();                                     ・・・⑤
}

void buildtree(void)
{
  int v,e,l,r;

  printf("v e l r = "); 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("v e l r = ");scanf("%d%d%d%d",&v,&e,&l,&r);
      }
    printf("root     = "); scanf("%d",&root);
}

void dfsearch(void)
{
    int i,v;
    int VISIT[SIZE+1];

    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){
          if (LEFT[v]!=0 && VISIT[LEFT[v]]==0){
                    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;

          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);
    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]);
        printf("u = %d",u);
        if(RIGHT[u]==0){
            if(FATHER[u]==v) RIGHT[v]=0;
            else             LEFT[FATHER[u]]=0;
           }
        else{
            if(FATHER[u]==v){
                RIGHT[v]=RIGHT[u]; FATHER[RIGHT[u]]=v;
            }
          else{
            LEFT[FATHER[u]]=RIGHT[u]; FATHER[RIGHT[u]]=FATHER[u];
            }
         }

        VALUE[v]=VALUE[u];
        e=u;
      }
    else{
       if(LEFT[v]!=0) u=LEFT[v];
       else           u=RIGHT[v];
       if(LEFT[FATHER[v]]==v) LEFT[FATHER[v]]=u;
```

A

C

B

```
        else                    RIGHT[FATHER[v]]=u;
        FATHER[u]=FATHER[v];
        e=v;
      }
     emptycell(e);
}

int search(int x)
{
    int flag;
    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("v = %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;
}
```

B

プログラムリスト３

３．１　ヒープへの要素の挿入

```
#include <studio.h>                                    */
#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) /*youso sounyu*/
 {
    int i,i2,w;

    writeheap();
    ++n; HEAP[n]=e; /*sounyusubekiyousowo2bunnkino*/
                        /*saigonoyousotoshitetuika       */
    writeheap();
    i=n;   i2=i/2; /*i2: ten1nooya*/
    /*jyohouidou*/
```

```
    while(i>1 && HEAP[i2]>HEAP[i]) {
    w=HEAP[i2]; HEAP[i2]=HEAP[i]; HEAP[i]=w;
    printf("element%d to elements%d¥n",HEAP[i],HEAP[i2]);
    writeheap();
    i=i2;i2=i/2; /*i2:ten1nooya*/
   }
}


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

    /*heep hyouji*/
     if(n!=0) {
         depth=(int)(log((double)n)/log(2.0));
         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]);
         printf("¥n");
       }
    }
    printf("¥n");
   }

 int pow2(int n)
  {
    if(n>0) return(2<<(n-1));
    else      return(1);
 }
```

３．２　ヒープの要素の削除

```
#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)
 {
    writeheap();
    min=HEAP[1];
    HEAP[1]=HEAP[n];

  --n;
  writeheap();
  shiftdown(1);
 }

void shiftdown(int i)
 {
```

```
  int l,r;
  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("elemet%d to element%d ¥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("element%d to element%d¥n",HEAP[i],HEAP[l]);
       i=l;
   }
   else {
            w=HEAP[i]; HEAP[i]=HEAP[r]; HEAP[r]=w;
            printf("element%d to element%d¥n",HEAP[i],HEAP[r]);
            i=r;
   }
  SKIP:
       l=2*i; r=2*i+l;
       writeheap();
   }
  }

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

   /*heep hyouji*/
    if(n!=0) {
        depth=(int)(log((double)n)/log(2.0));
        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]);
        printf("¥n");
      }
   }
   printf("¥n");
 }

int pow2(int n)
 {
   if(n>0) return(2<<(n-1));
   else      return(1);
}
```

３．３　ヒープソート

```
#include <math.h>
#define HSIZE 100

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

int HEAP[HSIZE+1];
int n,min;

void main(void)
{
   buildheap();                    ・・・①

  while(n>0) {
        writeheap();              ・・・②
        deletemin();              ・・・③
        printf("MIN=%8d¥n",min);
   }
}

void buildheap()
{
   int k;

   n=1;
   printf("?"); scanf("%d",&HEAP[n]);
   while(HEAP[n]<9999) {
```

```
   ++n;
    printf("?"); scanf("%d",&HEAP[n]);
   }
   --n;
   writeheap();
   for(k=n; k>=1; --k) {
    shiftdown(k); writeheap();
    }
   }

void deletemin(void)
 {
    writeheap();
    min=HEAP[1];
    HEAP[1]=HEAP[n];
    HEAP[n]=min;

   --n;
   writeheap();
   shiftdown(1);
 }

void shiftdown(int i)
 {
   int l,r;
   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("elemet%8d to element%8d ¥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("element%8d to element%8d¥n",HEAP[i],HEAP[l]);
        i=l;
    }
    else {
            w=HEAP[i]; HEAP[i]=HEAP[r]; HEAP[r]=w;
            printf("element%8d to element%8d¥n",HEAP[i],HEAP[r]);
            i=r;
    }
  SKIP:
        l=2*i; r=2*i+l;
        writeheap();
    }
  }

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

    /*heep hyouji*/
     if(n!=0) {
         depth=(int)(log((double)n)/log(2.0));
         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("%8d",HEAP[j]);
         printf("¥n");
       }
    }
    printf("¥n");
  }

 int pow2(int n)
  {
   if(n>0) return(2<<(n-1));
```

```
    else      return(1);
 }
```

プログラムリスト４

４．１　ハッシュ法要素の挿入

```
#define SIZE 100
#define M        6

void main(void);
void emptycell(void);
void insert(int x);
int   h(int x);
void writehash(void);

int VALUE[SIZE+1],NEXT[SIZE+1];
int HASH[M],TAIL[M];
int newcell;

void main(void)
{
       int i,x;
       for(i=0;i<M;++i)                                  ┐
           HASH[i]=TAIL[i]=0;                            ┘ ①

       emptycell();

       printf("? "); scanf("%d",&x);                     ・・・②
       while(x!=9999){                                   ┐
              insert(x);                                 ┘ ③
              printf("? "); scanf("%d",&x);
            }
       writehash();                                      ・・・④
}

void emptycell(void)
{
```

```
    int i;

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

void insert(int x)
{
    int i,p;

    i=h(x);
    p=newcell; newcell=NEXT[p];
    VALUE[p]=x;
    if(TAIL[i]==0) TAIL[i]=p;
    NEXT[p]=HASH[i]; HASH[i]=p;
}

int h(int x)
{
    return(x % M);
}

void writehash(void)
{
    int i,p;

    printf("ハッシュ表¥n");
    for(i=0;i<M;++i){
        printf("%3d          :",i);
        p=HASH[i];
        while(p!=0){
            printf("%4d",VALUE[p]);
            p=NEXT[p];
        }
        printf("¥n");
```

```
        }
}
```

４．２　ハッシュ法要素の探索

```
#define SIZE 100
#define M       6
#define SIZE 100
#define M       6

void main(void);
void emptycell(void);
void insert(int x);
int   h(int x);
void search(int x);
void writehash(void);

int VALUE[SIZE+1],NEXT[SIZE+1];
int HASH[M],TAIL[M];
int newcell;

void main(void)
{
      int x;

      emptycell();
      printf("? "); scanf("%d",&x);
      while(x!=9999){
            insert(x);
            printf("? "); scanf("%d",&x);
          }
      writehash();

      printf("要素判定¥n");
      printf(" ? "); scanf("%d",&x);
      while(x!=9999){
                search(x);
```

①

・・・②

③

```
                printf(" ? "); scanf("%d",&x);
              }
}

void emptycell(void)
{
    int i;

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

void insert(int x)
{
    int i,p;

    i=h(x);
    p=newcell; newcell=NEXT[p];
    VALUE[p]=x;
    if(TAIL[i]==0) TAIL[i]=p;
    NEXT[p]=HASH[i]; HASH[i]=p;
}

int h(int x)
{
    return(x % M);
}
void search(int x)
{
    int i,p,q;

    i=h(x);

    q=0;
    p=HASH[i];
```

```
    while(p!=0){
        if(VALUE[p]==x) {q=p; p=0;}
        else                    p=NEXT[p];
     }
    if(q==0) printf("存在せず　¥n");
    else       printf("存在 ¥n");
}

void writehash(void)
{
    int i,p;

    printf("ハッシュ表¥n");
    for(i=0;i<M;++i){
            printf("%3d             :",i);
            p=HASH[i];
            while(p!=0){
                    printf("%4d",VALUE[p]);
                    p=NEXT[p];
                }
           printf("¥n");
          }
}
```

４．３　ハッシュ法の要素の削除

```
#define SIZE 100
#define M        6

void main(void);
void emptycell(void);
void insert(int x);
int   h(int x);
int search(int x);
void delete(int x);
void writehash(void);

int VALUE[SIZE+1],NEXT[SIZE+1];int HASH[M],TAIL[M];
int newcell;

void main(void)
{
      int x;

      emptycell();
      printf("? "); scanf("%d",&x);
      while(x!=9999){
            insert(x);
            printf("? "); scanf("%d",&x);
          }
      writehash();
      printf("要素削除¥n");
      printf("x= ");   scanf("%d",&x);
      while(x!=9999){
                delete(x);
                writehash();
                printf("x=   "); scanf("%d",&x);
```

```
                }
}

void emptycell(void)
{
    int i;

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

void insert(int x)
{
    int i,p;

    i=h(x);
    p=newcell; newcell=NEXT[p];
    VALUE[p]=x;
    if(TAIL[i]==0) TAIL[i]=p;
    NEXT[p]=HASH[i]; HASH[i]=p;
}

int h(int x)
{
    return(x % M);
}
int search(int x)
{
    int i,p,q;

    i=h(x);

    q=0;
    p=HASH[i];
    while(p!=0){
```

```
        if(VALUE[p]==x) {q=p; p=0;}
        else                    p=NEXT[p];
      }
    if(q==0) printf("存在せず　¥n");
    else       printf("存在 ¥n");
}

void writehash(void)
{
    int i,p;

    printf("ハッシュ表¥n");
    for(i=0;i<M;++i){
          printf("%3d             :",i);
          p=HASH[i];
          while(p!=0){
                printf("%4d",VALUE[p]);
                p=NEXT[p];
              }
         printf("¥n");
        }
}

void delete(int x)
{
    int p,q,s,i;
    q=search(x);

    if(q!=0) {
        i=h(x);
        if(HASH[i]==q) HASH[i]=NEXT[q];
        else{
          p=HASH[i];
          while(p!=0){
                if(NEXT[p]==q) {s=p; p=0;}
                else                p=NEXT[p];
```

```
                }
                NEXT[s]=NEXT[q];
            }
        NEXT[q]=newcell; newcell=q;
    }
}
```

プログラムリスト５

５．１　要素の探索（実測値用）

```
#define SIZE 4500000

void main(void)
{
   unsigned long VALUE[SIZE+1],NEXT[SIZE+1];
   double head,
          tail;
   unsigned long newcell,i,q,p,x;

   newcell=1.0;
   for(i=1.0; i<SIZE; ++i) {
   NEXT[i]=i+1;
    VALUE[i]=i;
   }
   printf("? "); scanf("%d",&x);
   /*serch        */
   q=0.0;
   p=1.0;
   while(p!=0){
      if(VALUE[p]==x) {printf("%d¥n",p); p=0;}
      else                    p=NEXT[p];
   }

   printf("%d¥n",q);
 }
```

プログラムリスト　６

６．１　構造体でのリストの構築と探索

```
typedef int KEY;
typedef int DATE;
typedef struct node{
            struct node *next;
            KEY key;
            DATE date;
       } NODE;
  static NODE list_head;

void init(void)
 {
     list_head.next=NULL;
 }

NODE *search(KEY key)
{
    NODE *p;

    p=list_head.next;
    while (p!=NULL){
            if (p->key ==key)
                  return p;
            p=p->next;
           }
     return NULL;
}

NODE *insert(NODE node)
{
        NODE *p, *q, *pos;
```

```
        if ((pos=malloc(sizeof(NODE))) == NULL){
              fprintf(stderr, " メモリ不足です ．¥n");
               exit(EXIT_FAILURE);
             }
        q=&list_head;
        p=q->next;
        while(p!=NULL){
               if(p->key == node.key){
                        free(pos);
                        return NULL;
                       }
               else if(p->key > node.key)
                    break;
               q=p;
               p=p->next;
             }
         *pos = node;
          pos->next = q->next;
          q->next =pos;
          return pos;
}

void disp(void)
{
     NODE *p;

     p=list_head.next;
     while(p!=NULL){
          printf("%d -> ", p->key);
          p=p->next;
       }
     printf("¥b¥b¥b      ¥n");
}

main(){
```

```
    char buff[8];
    int succeeded;
    KEY key;
    NODE node, *result;

    init();
    while(1){
        printf("Sn: search, In: insert ? ");
        gets(buff);
        key = atoi(&buff[1]);
        switch(buff[0]){
            case 'S':     case 's':
                result=search(key);
                if (result!=NULL)
                        printf(" キー %d 見つけました。¥n",key);
                else
                        printf(" キー %d は見つかりません。 ¥n",key);
                break;
            case 'I':       case'i':
                node.key=key;
                result=insert(node);
                if(result !=NULL)
                        printf(" キー %d を挿入しました。 ¥n",key);
                else
                        printf(" キー %d は挿入済みです。 ¥n",key);
                break;
            default:
                return 0;
            }
           disp();
        }
}
```

卒業研究の概要

私たちは、就職活動のために前半プレゼンテーションセミナーを行った。ここでは、新聞記事などから、自分の興味のあるものをテーマに選び、それについて深く調査し、まとめそして、プレゼンテーションを行うというものである。そこで私は、「携帯電話のネット接続の普及」と、「ハッカー対策完備のビル」という 2 つのテーマでプレゼンテーションを行った。

また、携帯電話のネット接続の方は、３人でグループを作り、さらに調査していき、学祭でもプレゼンテーションを行った。ここでいかに自分の調査結果を，相手に伝えるということの難しさ、そして重要かということを学んだ。

そこで、本稿にも学祭で、発表した「携帯電話のネット接続はなぜ普及したか」を、記載する。このテーマを選んだ理由は、現在急速に普及している携帯電話のネット接続が、これからの IT 革命の主役になると考えたからである。また、そのことから２１世紀の IT 革命についても考えた。

発表グループ：　発表－松岡　茂行
資料作成、質疑応答－木村　麻由子、窪添絵里子

後半は今回の論文内容である、データ構造とアルゴリズムについて、調査、研究を行った。

説明

テーマ：「携帯電話におけるネット接続なぜ普及したのか」

背景

今日までの IT 革命の主役となっていたのは、まぎれもなくパソコンである。しかし、最近では、i モードに代表されるモバイルインターネットによってパソコンは主役の座から降りてしまうかもしれない。その理由としてパソコンに対する不満である。それは、パソコンは以前より操作は簡単になったが、それでもいくらか知識が必要である。また、価格も高く１人が１台もつのは無理である。サイズにしても小さくはなったが、持ち運びするには重くスペースも必要である。それを解消するものとして現れたのがモバイルインターネットである。いってみればパソコンは２０世紀の IT 革命であり、I モードなどのモバイルインターネットは２１世紀の IT 革命といえるだろう。いまはその移行のときである。
そこで、携帯電話のネット接続という部分なかでも、NTT ドコモの I モードに注目して説明する。

内容

記事の内容、今年の２月２３日の日本経済新聞に記載されていた記事である。この記事を読んで、ｉモードのことを調べるきっかけになった。
目的、本論、結論、「ｉモード」を調べていってヒット理由と今後の課題を説明する。
まとめ
以上の項目で今回説明していく。

記事の内容

これは２０００年２月２３日の日経新聞の記事をまとめたものである。
内容はＮＴＴドコモが今年の夏以降の携帯電話ほぼすべてに「ｉモード」を標準搭載するといった記事である。今までは「ｉモード」が搭載されているのは、１部の携帯電話だけだった。これがすべての携帯電話に「ｉモード」が搭載されるのである。現在では５０１や５０２以外に２０９も i モードに対応している。

これは「ｉモード」加入者を増やすためにおこなわれる。現在「ｉモード」加入者は２０００年２月２１日現在４２４万人に達していいる。これは、ドコモの予想をはるかに上回る数字である。そこで当初目標としていた数字を修正することが決まった。当初の予定

はサービス開始の１９９９年から３年間で１千万を目標としていたが、この勢いを考えて、目標は変更された。その目標が２０００年中に１千万人である。目標達成のために、標準搭載という形をとったのである。その結果今年の８月には目標の１０００万人を超えることができた。

目的
今回の目的は、まずｉモードの仕組みを知ることと、そして、それを知りどうしてこんなにヒットしたかということをユーザーの目から見て考える。

ドコモ側の考え
最初にドコモ側が考えるヒットした理由である。
ドコモはこのＩモードもヒットの理由を「ユーザー、インフォメーションプロバイダ(IP)そしてドコモのＩモードの関わるみんなが得をする仕組みをつくったから」といっている。その仕組みというのは有料コンテンツの料金徴収代行システムを導入したということである。図を参照。

ネット接続の利用者
携帯電話でのネット接続は2000年2月から3月までの約1ヶ月間で、180万人増えている。それに対し、パソコンで、プロバイダ経由でのネット接続は、46万人の増加にとどまった。このことからも、携帯電話主導で、ネット接続が普及しているといえる。

携帯電話でネット接続できるサービス
携帯電話のネット接続は、i モード以外にもあるが、約７５％が NTT ドコモの i モードである。

i モードとは
　ｉモードは 1999 年 2 月にＮＴＴ移動通信（ＮＴＴドコモ）が開始したサービス。サービス内容は、携帯電話で直接インターネットに接続するというものである。そして電子メールの送受信や、i モード対応のホームページをみることができる。それまでは、インターネットをするには、パソコンと電話回線を使ってはじめてすることができた。しかし、ここでいう携帯電話を使ってネット接続した時、パソコンほど膨大なデータを扱うことはできない。
　そのため NTT ドコモと契約した「公式サイト」があります。これ以外にも企業や個人が作った i モード対応のホームページもあります。しかし「公式サイト」ではないので、安全性という部分ではリスクを負わないといけないが、役にたつサイトも多数ある。現在公式サイト、非公式サイト合わせて３０００以上のホームページが i モードで活用することがで

きる。
　提供されるサービスは、モバイルバンキングやチケット予約など、生活に身近で便利なものを中心にラインナップしている。また、通信方法はパケット通信を採用している。
　パケット通信とは、従来パソコンからネットに接続した時は、接続時間で課金していく。これに対してパケット通信は送受信したデータ量で課金される方法である。パケット通信だから常につないだ状態でも、課金されないという特徴があり、料金が安くてすむ。

i モードのネットワーク
　ｉモードのネットワーク構成である。この図を参照。携帯電話からサイトに接続するとき直接ＩＰという情報提供者に接続するのではない。サイトに接続するときはドコモｉモードセンターを経由して行われる。
　インターネットと専用線にわかれている。インターネットというのは、おもに非公式サイトの URL をみるときに使う。専用線は、公式サイトのモバイルバンキングやチケット予約などのドコモと契約している IP が行っているオンラインサービスを行うときに使われる。

IP とは
　IP とは i モードへの情報提供者のことである。サービス開始当初から、現在まで約 9 倍近く増えている。

IP のグラフ
　これは i モードの IP 数のグラフである。縦軸が IP 数で横軸は I モードサービス開始時から現在までの月である。グラフ参照。
　まず１９９９年の８月から９月の間に大きくふえている。これはショートメールなど機能が充実し、また機種も充実してきた事から、i モード契約数も８月には１００万人を越えたことが原因である。１００万人を越えることで、各企業が IP として I モードに参入しはじめた。
　また今年の２月から３月にかけても大きくふえている。これは I モード契約者数が増えてきたこともあるが、新しい機種５０２シリーズがでてきたことで、着信音や画像のダウンロードが出来るようになり、それに対応した IP がふえてきたのである。
　このように各企業も i モードに注目し、IP として参入する企業がふえている。各企業は i モードでのビジネスに注目しているのである。
このグラフのように、右上がりを続けていくには、ここにもあるように新しい機能の追加や新技術の導入が必要である。そのため、ドコモでも年内に５０３、21 世紀には W-CDMA 方式の携帯電話の発売を予定している。よってこれからも、このグラフは右上がりを続けていくと考えられる。
また、周りの人が i モードを使っている人が多いというのも、i モードの良さだと思う。こ

のように、連鎖反応的な市場もヒットした理由のひとつである。

結論

それでは、結論としてヒットした理由と今後の課題について説明する。

最初に携帯電話はパソコンの問題点を解決した。

そしてヒットした理由は、手軽に情報を入手できる点だろう。携帯電話だけでインターネットに接続できるというのは魅力的だろう。これによってパソコンに対する不満も解決することができたと考える。操作もリモコン感覚で使えるうえに、価格も安く、携帯電話は、ほとんどの人が持っている端末である。また、近い将来人口普及率６０％になると言われている端末なのでこれからも普及していくと思う。

また、多くのサービスがあり、用途が多彩ということで、利用者の幅が広いというのもヒットした理由のひとつだろう。多くのの企業が IP としてｉモードに参加しているので、サービスも多くある。しかし、これからの課題はＩＰがどれだけ良いコンテンツを作っていくかもかぎだと思う。

マーケティングについていえば、テレビＣＭに人気タレントの出演によりファンからも支持をえました。また、新機種をだすことで、新しいサービスや機能を追加し、そのたびに多くのユーザーをつかんできた。ｉモードという言葉を知らない人がいないぐらいこの社会に浸透してきた。これはマーケティングが成功したということだと思う。

今後の課題として、今回その話にはふれなかったが、セキュリティ面だろう。モバイルバンキングや、チケット予約、購入などお金に絡むサービスがあるのにセキュリティ面は気になるところである。

現在は、携帯電話の無線プロトコルとして暗号化はされているだけである。

対策として、ドコモのｉモードサーバと IP のサーバ間では SSL による通信を検討はしている。しかし、携帯電話とｉモードサーバ間の暗号化は検討されていない。

まとめ

最後に今回のまとめです。

最初に立てた 2 つの目的について

１つ目の目的は「ｉモード」の仕組みを知ること。「ｉモード」はパケット通信を利用する事により、ユーザに快適なネット環境を提供可能にした。それはｉモード携帯電話機から、ｉモードセンターを経由してＩＰ（情報提供者）のサイトに接続して、情報のやり取りをするネットワークである。

２つ目は今回の最大の目的であるヒットした理由を考えること。私たちは手軽に情報を入手でることと、サービスが多く用途が多彩ということを理由と考えた。この二つは私たちユーザーにとって最大の魅力である。年齢・用途に関係なく幅広く使える。また海外でも携帯電話のネット接続は注目されている。これからのビジョンとして「いつでも、どこ

でも、だれとでも」コミニュケーションできる通信技術が実現していくのである。ネット接続できることで携帯電話は多機能化し、２０１０年には人口普及率６０％を見込まれている。世界中で携帯電話のネット接続が使えるようになる日も遠くないだろう。

最後に携帯電話のネット接続は急速に普及していっている。このことからも最初にもいったように２１世紀の IT 革命は携帯電話等のモバイルが主役になるだろう。

そして２１世紀の IT 革命の主役は高度な知識をもった一握りの人間ではなく、モバイルの出現と発展により、これまで IT とは無縁だった一般の人々が主役になる。

そして、一般の人にこれまで考えられないような可能性がひろがり、ビジネスでも IT 関連の市場はこれまでとは比べようのないくらい大きく広がっていくであろう。

これこそが２１世紀の IT 革命の真価だと思う。

2000. 10. 21

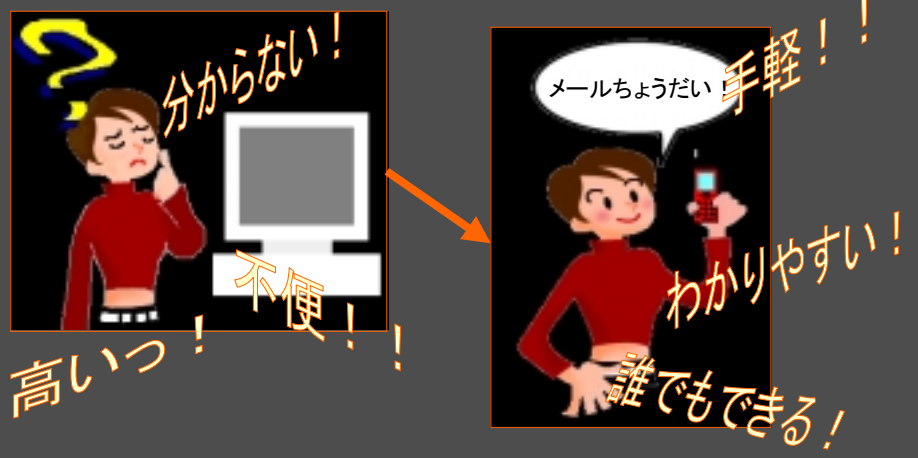
携帯電話におけるネット接続は なぜ普及したのか

（日本経済新聞　2000年2月23日
「iモード、携帯に標準装備」 参考）

電子･光システム工学科　4年

＊　松岡　茂行･　木村　麻友子　･窪添　絵里子

パソコンから携帯へ



内容

- 記事の内容
- 目的
- 本論
- 結論
- まとめ

記事の内容

NTTドコモが今夏発売以降から
携帯電話に「iモード」を標準搭載

理由：「iモード」加入者を増やすため

→当時、加入者は424万人（2000年2月21日現在）

目標：当初の目標　サービス開始から3年で1千万人

予想以上のペースで普及

目標の修正：2000年中に1千万人
→　結果：　8月現在で1109．5万人（532日目で突破）

目的

- 「iモード」の仕組みを知る
- ヒットした理由を考える

iモードの料金徴収システム

ネット接続の利用者

携帯電話でのインターネット接続の利用者
570万人（2000年2月）→750万人（2000年3月末）
約1カ月で180万人増加　（全体の30%以上）

大手ネット接続業者15社の利用者
1147万人（2000年2月）→1193万人（2000年3月末）
約1カ月で46万人の増加

「ケータイ」主導でネット利用が急拡大している

携帯電話でネット接続できるサービス

- NTTドコモ：「iモード」
  加入者：560．3万人（2000年3月末）
- 第二電電グループと日本移動体通信（現au）
  ：「EZウェブ・EZアクセス」
  加入者：135万人（2000年3月末）
- J-フォングループ：「J-SKY」
  加入者：56．5万人（2000年3月末）

※携帯電話のネット接続の約75%がNTTドコモの「iモード」

iモードとは

NTTドコモが1999年2月にサービス開始

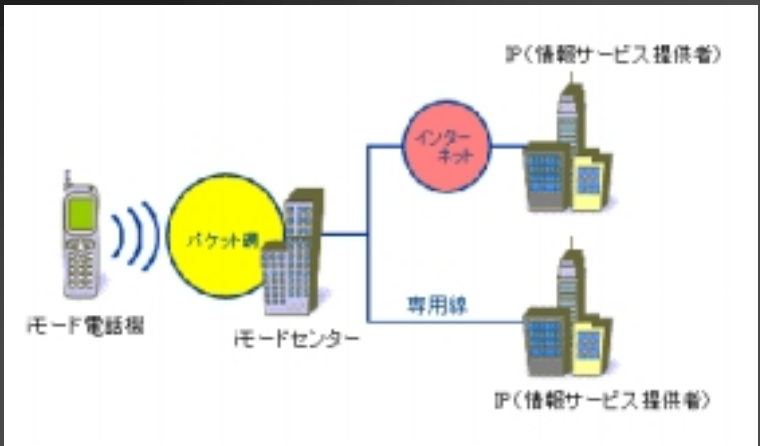
携帯電話で直接インターネットに接続し、
オンラインサービスが利用可能

特徴

iモードキーを押すだけで多彩なサイトにアクセス
パケット通信を採用

→ 接続時間だけでなく、送受信したデータ量で課金

ネットワーク構成



http://www.nttdocomo.co.jp/i/ より引用

IP　(Information Provider)

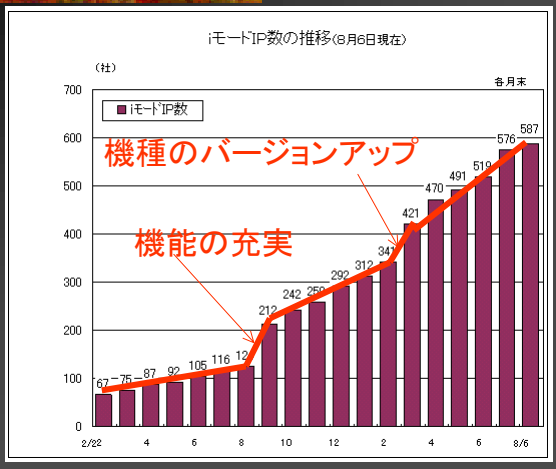
プラットホームを利用して各IPがビジネスを展開

現在のIPは587社 （2000年8月6日現在）

⇒ サービス開始当初（67社）の約9倍

※一般ホームページは約18700サイト

iモードIP数の推移



ヒットした理由と今後の課題

携帯電話はパソコンの問題点を解決した

- ネットから手軽に情報を入手できる
- 多くのサービスがあり、用途が多彩
- マーケティングが他社より優れていた

＜問題点＞

お金に絡むサービスを行うのにセキュリティーが弱い

＜対策＞

iモードサーバとIPのサーバ間では
SSLによる通信を検討
(Secure Sockets Layer Protocol)

まとめ

「iモード」の仕組み
パケット通信を使用
携帯電話からiモードセンターを経由してネットに接続

ヒットした理由
手軽に情報を入手できる
パソコンを使用するよりも気軽
マーケティングが優れている

携帯電話のネット接続は急速に普及している

コンテンツ（情報の内容）の作成

- ドコモ自信がコンテンツを作るのではない
- 事業者の利用できる共通の基盤（プラットホーム）を作る
- プラットホームを利用してIPがコンテンツを作る

IPの主な種類

- エンターテイメント系
  （ゲーム、映画、占い、FM局情報）
- 情報系
  （ニュース、株価、天気予報）
- 取引系
  （モバイルバンキング、トラベル、チケット予約）
- データベース系
  （レストランガイド、料理レシピ）

SSLとは

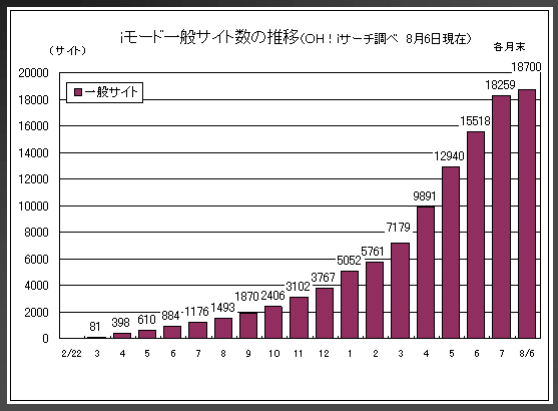
WEB（ネットワーク上のホームページの場所）ブラウザとWEBサーバの間を安全にやりとりするためのプロトコル.

機能

認証：ブラウザとサーバで互いに相手の身分を確認する

暗証：インターネット上でやり取りされる情報を覗き見される可能性を小さくしている

iモードー般サイト数の推移



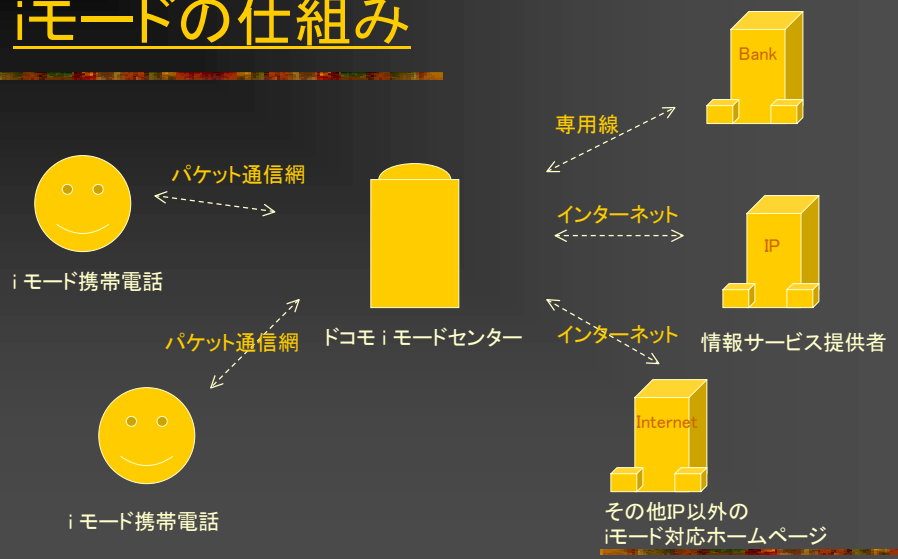
プロトコルとは

通信規約

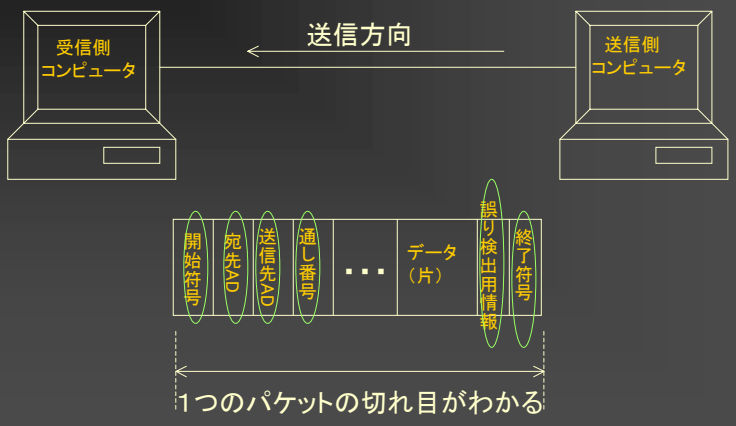
・・・コンピュータ相互間あるいはコンピュータとネットワーク間等でデータを交換するために決められた約束事.

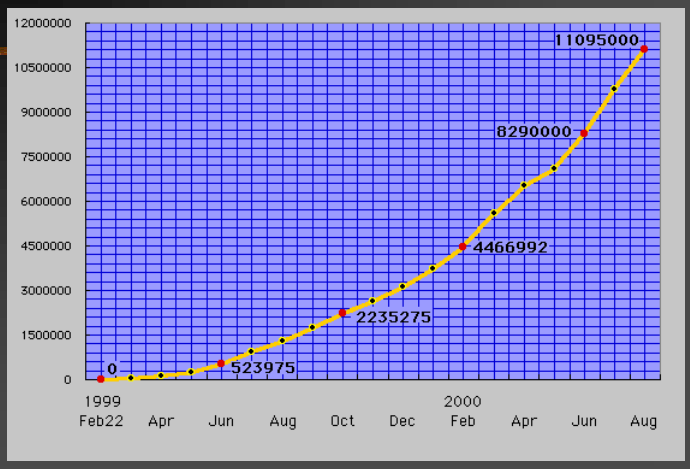
取り扱う情報のformatと、その情報の送受信の手順を記述したもの.

iモードの仕組み



パケットの構造例





月別

iモードのホームページ

iモード向けホームページスペック

- HTML 2.0、 3.2、 4.0 のサブセット
- 文字コードはS-JIS、画像はgifのみ表示可能

特徴

- iモードは5キロバイトのバッファを持っている
- グラフィックをとばして表示する機能を持っている
- グラフィックが5キロバイトを越えると表示できない
- Phone To 、Web To、Mail To 等のオリジナル機能がある

Phone To、Mail To、Web To

- **Phone To**
  ホームページ上に表示されている電話番号に
  ワンタッチでかけることができる
- **Mail To**
  iモードの各サイトやメール内のアドレスに
  ワンタッチでアドレス入力したメール作成画面を表示できる
- **Web To**
  メール内に“http”で始まるホームページのURLが
  書いてあれば、ワンタッチでインターネット接続可能

マーケティング

- コマーシャル
  人気タレントの出演により、ファンからの支持も得た.

  i モードのメインに田村正和、加藤あい、広末涼子

  （その他にも地域で多数のタレントがCM出演）

iモードの情報料金徴収システム（2）

有料コンテンツの料金代行徴収システム

「ユーザ、IP、ドコモのiモードに関わる
みんなが得する仕組み」

- 有料コンテンツに支払う利用料金（月100～300円）を
ドコモが通話料金とともに徴収
- ドコモからIPに利用料が支払われ、9%が手数料として
ドコモに入ってくる

利用者が増えればIPの数が増え、
それによりIPの質が高まりユーザが増える。

iモードの当初の目標

- ドコモ全体で携帯電話を1000万台製造し、
そのうちの10%をiモードとして売り出す計画

1年目　100～200万台

2年目　200～300万台　　約2割り増し

3年目　400万～　　約2割り増し