

5 連結リスト

5.1 連結リスト

5.1.1 連結リストとは?

第4章では、リストを実現するデータ構造として配列を取り上げました。この章では、リストのもう1つの実現法として連結リストを取り上げます。

連結リスト(linked list)とは、リストに含まれる各要素をポインタによってつなぎ合わせたものです。ポインタによって連結されている(linkされている)ことから、連結リストと呼ばれます。各要素には次の要素へのポインタを付加します。たとえば、MYDATA型のデータを並べたリストを表現するには、次のように2つのメンバをもつ構造体を定義します。

```
struct CELL {  
    struct CELL *next;  
    MYDATA     data;  
};
```

メンバ next は、次の要素を指すポインタです。また最後の要素については、次にくる要素はありませんから、メンバ next に NULL ポインタをセットしておきます。

連結リストのように、ポインタを多用するデータ構造を理解するには、図を描くのがいちばんでしょう。Fig.5.1(a)は整数のリストを配列で表現したものでです。配列 a にリストの要素を入れ、要素の数を変数 n で管理しています。リストには4つの要素 2, 5, 12, -4 がこの順番で並んでいます。

このリストを、連結リストで表現すると Fig.5.1(b) のようになります。ここでは、箱の1つ1つがリストの要素を表し、箱と箱を結ぶ矢印がポインタを表しています。また、NULL ポインタは、矢印の代わりに斜めに線を引いて表現します。箱のことをセル(cell)とも呼びます。ここでは、セルの型が次のように定義されています。

Fig. 5.1 リストの表現法

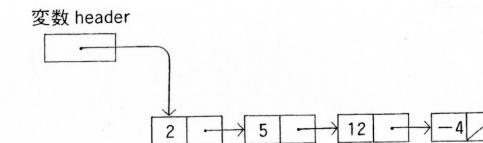
配列 a

2	a [0]
5	a [1]
12	a [2]
-4	a [3]
	a [4]
	a [5]
	a [6]
	a [7]
	a [8]
	a [9]

n=4(要素の数)

```
int a[10]; /*リストを表現する配列*/  
int n;      /*リストの要素数*/
```

(a) 配列による表現



```
struct CELL {  
    struct CELL *next; /*次のセルへのポインタ*/  
    int         value; /*値*/  
} *header;
```

- ・リストはポインタによって連結されたセルとして表現される。
- ・斜めに引いた線は、NULLポインタを表している。

(b) 連結リストによる表現

```
struct CELL {  
    struct CELL *next;  
    int         value;
```

};

メンバ next は次のセルへのポインタです。メンバ value は int 型で要素の値がセットされます。変数 header はセルへのポインタ(つまり, struct CELL *型)として定義されており、このリストの先頭を指しています。変数 header から、ポインタをたどって各要素を順番にアクセスすることができます。また、変数 header が NULL のとき、リストは空である(要素が1つもないリスト)とすることは自然な表現でしょう。

配列では、次のように for 文を使って添え字を1ずつ増加させながら、リストに含まれるすべての要素を順に処理することができます。

```
int i;  
for (i=0; i<n; i++)
```

```

printf("%d\n", a[i]);
連結リストでこれと同じことをするには、for文を使ってメンバnextに入っているポインタを順にたどっていきます。
struct CELL *p;
for (p=header; p!=NULL; p=p->next)
printf("%d\n", p->value);

```

5.1.2 ■ 連結リストの基本的性質

連結リストの性質について、配列との比較を交えながら考察しましょう。配列では何番目と指定することで、任意の要素を一定時間で参照することができます。つまり、要素の参照に必要な計算量は $O(1)$ です。配列では、要素をランダムアクセスすることが可能ですが。

たとえば、Fig.5.1(a) では、3番目の要素は $a[2]$ に格納されています。ここで、C 言語の仕様では配列の要素は 0 から始まるので、 k 番目の要素は $a[k-1]$ に対応していることに注意しましょう。

これに対して、連結リストでは、ポインタを順番にたどるシーケンシャルアクセスしかできません。3番目の要素を参照するには、変数 header から 3 回ポインタをたどる必要があります。リストの要素数が n 個であるとき、任意の要素を参照するには平均して $n/2$ 回ポインタをたぐらなければならぬので、計算量は $O(n)$ になります。このように、連結リストでは配列と違って、要素を参照するにはかなりコストがかかることになります。

ただし、先頭の要素はつねに変数 header で指されているので、 $O(1)$ で参照できます。また、先頭以外の要素についても、つねに特定の要素を指すポインタ変数を用意するという手があります。たとえば、待ち行列を実現するのなら、つねに最後の要素を指しているポインタ変数を用意すれば、最後の要素をつねに $O(1)$ で参照することができます。

セルは、次の要素へのポインタのみをもっていて、前の要素へのポインタをもっていません。したがって、リストの先頭から後ろに向かって進むことはできても、反対に前に向かって戻ることはできません。このように、要素のアクセスについては、連結リストはかなりのハンデを背負っています。しかし、実際には、前から後ろに向かって要素を順番にたどるだけで十分というケースが

多いので、この点はあまり大きな欠点ではありません(後ろ向きにたどれる連結リストについては、「5.3 双方向リスト」の項で説明します)。

連結リストでは、セルの 1 つ 1 つに、次のセルへのポインタをもたせなければなりません。つまり、セル 1 個につき、4 バイトの領域がポインタのために必要となります。Fig.5.1 の例のように、扱うデータ型の大きさが小さければこのオーバーヘッドも無視できません。反対に、ある程度以上大きなデータ型を扱うのなら、ポインタのために必要となるメモリは問題にならないと考えることができます。

のけから悪い点ばかりを取り上げてしまいました。「何だ、配列に比べてちっともいいところがないじゃないか。連結リストなんか勉強しないでいいや」と思った方もいらっしゃるでしょう。でも、これから連結リストのよい点を説明するので、もうちょっとだけ辛抱してください。つき合ってみれば、なかなかいい奴だということがわかるでしょうから。

連結リストの利点は、要素の挿入や削除が簡単に実行できます。第 4 章で説明したように、配列を使ってリストを表現する場合、要素の挿入・削除はかなり大仕事になります。挿入のときは、挿入する地点から後ろにあるすべての要素を 1 つずつ後ろにずらさなければなりません。リストの要素の数を n 個とすると、平均すると $n/2$ 個の要素をずらす必要があるので、挿入の計算量は $O(n)$ になります。また、削除についても同様に平均 $n/2$ 個の要素を移動する必要があります。 $O(n)$ の計算量になります。

これに対して、連結リストで表現した場合、挿入・削除を行うのにセル 자체を移動する必要がなく、数個のポインタを書き換えるだけですみます。これはつねに一定時間で実行することができる所以、計算量は $O(1)$ になります。

連結リストへの挿入の手順を簡単に説明しましょう。Fig.5.2(a) で、ポインタ x で指されているセルの次に、ポインタ p で指されている新しいセルをつなぎ込むと、Fig.5.2(b) のようになります。このとき、図中の灰色部分のポインタの値が書き換えられます。具体的にコーディングすると、

$p->next = x->next;$ ①

$x->next = p;$ ②

となります(丸数字は、Fig.5.2(b)の矢印に対応)。

Fig. 5.2 連結リストへの挿入

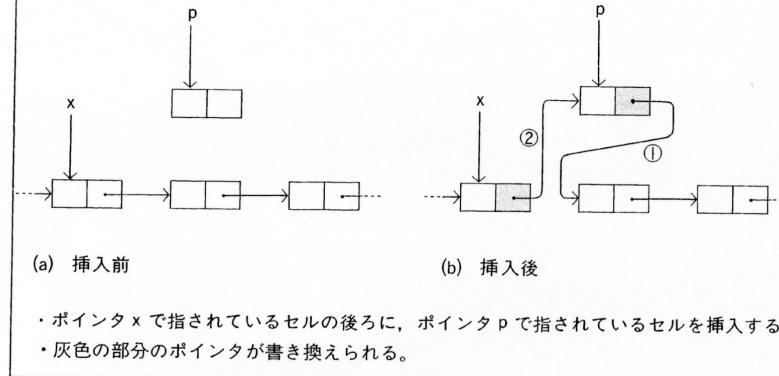
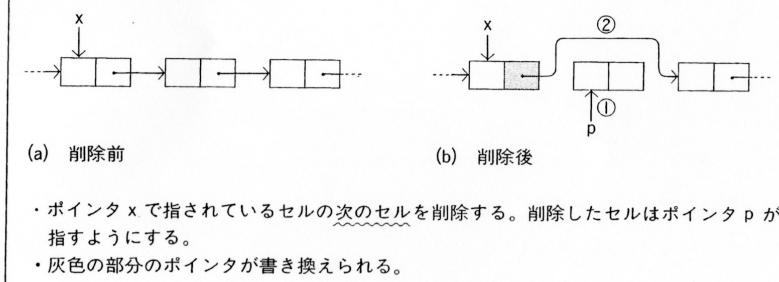


Fig. 5.3 連結リストからの削除



削除の手順は次のようにになります。Fig.5.3(a)で、ポインタ x で指されているセルの次のセルを削除してみます。この際に、削除処理が完了した時点で、ポインタ p が、削除されたセルを指しているようにします。削除が終わった状態がFig.5.3(b)です。ここで、変化するポインタは、灰色の部分とポインタpのみです。コーディングは次のようにになります(丸数字は、Fig.5.3(b)の矢印に対応)。

```
p      = x->next; .....①
x->next = p->next; .....②
```

ここで、挿入・削除を行う場所がどのように指定されるかに注目してみましょう。削除の場合は、削除するセルそのものではなく、1つ手前のセルへのポインタが必要となります。言い換えれば、連結リストでは、ある要素を指定してそれを削除することはできません。これは、ある意味ではかなりきつい制限です。また、同様に挿入では、指定した要素の後ろに挿入することは簡単ですが、前に挿入することはできません。これらの制限は、連結リストを扱ううえでの要注意事項です。

連結リストでは、あらかじめリストの長さを決めておく必要はありません。要素が増えれば増えた分だけ、いくらでもリストを長くすることができます。これに対して、配列ではあらかじめ長さの最大長を見積もって領域を確保しておかなければなりません。あらかじめ決めた個数以上の要素を登録することはできません。また利用しない部分は、ムダになってしまいます。

先ほど、連結リストではポインタの分だけ余分にメモリが必要であると説明しました。けれども、配列のようにあらかじめ領域を確保するために生じるムダがないので、連結リストのほうがメモリを有効に利用できることもあります。どちらが一方的に優れているということではなく、ケースバイケースでどちらを使うべきかを判断する必要があります。連結リストでのメモリ管理については、後ほど説明しましょう。

連結リストでは、複数のリストを連結して新しいリストを作る(Fig.5.4), 1つのリストを2つに分断する(Fig.5.5), という操作を容易に実現できます。あらかじめ、最後のセルへのポインタがわかっているれば、連結も分断も $O(1)$ で実行できます。これらの操作は、配列でもできないことはありませんが、要素がある配列から別の配列へコピーしなければならないので、要素の数に比例した計算量がかかってしまいます。

最後になりましたが、複雑なデータ構造を構築することができるというのも連結リストの大きな利点です。不定長の連結リストが不定個存在する、連結リストの要素が別の連結リストへのポインタをもっている(つまりリストのリストになっている), セルに複数のポインタをもたせることによって同時に複数の連結リストに所属できるようにする、などの工夫によって、非常に込み入った関係を表現するようなデータ構造を作ることが可能です。

Fig. 5.4 連結リストの連結

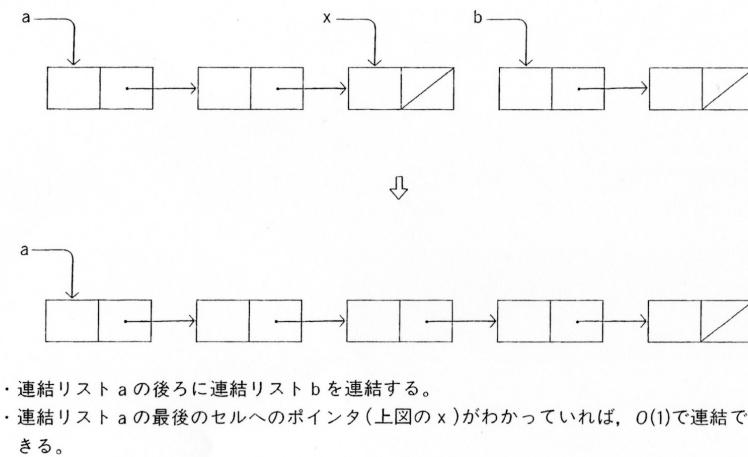
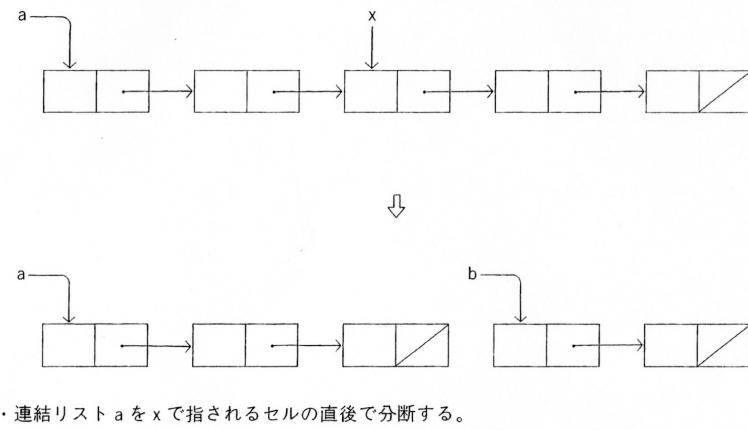


Fig. 5.5 連結リストの分断



5.1.3 ■ 連結リストとメモリ管理

連結リストを使う場合には、要素を入れるセルは、必要になるたびに割り当てることになります。つまり、プログラムを書くうえで、明示的にメモリ管理を行わなければなりません。

配列の場合には、必要な領域を配列としてあらかじめ割り当てておけばよいので、メモリ管理は特に必要ありません(厳密にいえば、配列内のデータを管理するのもメモリ管理には違いありませんが)。

C 言語では、変数は、その寿命を基準とすると、次の 2 種類に分類されます。

- ①プログラムの実行開始から実行終了まで存在するもの
- ②関数の実行が始まるときから、その関数の実行が終わるまで存在するもの

記憶クラスでいえば、`extern` と `static` が前者、`auto` と `register` が後者になります。関数の内部で定義した `static` 変数は、定義されている関数の外側では参照することはできないので、一見②に分類されるように思えますが、変数の値はプログラムの実行が終わるまで保持されるので、①に属しています。

しかし実際にプログラムを書いていると、関数の実行とは関係なく寿命をコントロールできるような変数——任意の時点に作成して、任意の時点に捨てられる——が欲しくなります。特に、連結リストのようなデータ構造を実現するためには、このように管理できる変数が必要です。

このようなニーズを満たすものとして、C 言語では標準ライブラリにメモリ管理用の関数が用意されています。メモリを割り当てるのは `malloc`、解放するのは `free` という関数で、これらを使えばプログラム実行中の任意の時点でメモリブロックを割り当てたり、解放したりすることができます。

のちほど第22章で取り上げますが、メモリ管理を行う関数の仕組みは興味深いアルゴリズムになっています。また、`malloc` 関数で割り当てたメモリは、不要になった時点で `free` 関数によってシステムに返すことになっていますが、使わなくなったメモリを自動的にシステムが回収する、ガベージコレクション(garbage collection: ゴミ集め)という技法もあります。

5.1.4 ■ 連結リストの操作

● 5.1.4.1 —— 要素の挿入

連結リストへ要素を挿入する操作をCのプログラムとして実現してみましょう。実際にプログラムを書くときには、特に境界条件に注意しなければなりません。

連結リストにつなぐセルを割り当てるには、先ほど説明したように、標準ライブラリ関数のmallocを利用するのがいちばん簡単です。セルの割り当ても含んだかたちの挿入はList 5.1(1)のようになります。先ほど説明した例と同様に、ポインタxで指されるセルの直後に新しいセルを挿入しています。「セルに値をセットする」という部分では、割り当てたばかりのセル(ポインタpが指している)に、必要な値をセットします。たとえば、

```
p->value = 125;
```

などとします。

関数mallocは割り当てるべきメモリがなくなってしまったら、NULLを返し

List 5.1 ● 連結リストへの挿入

(1) ポインタxで指されたセルの直後に新しいセルを挿入する

```
1: struct CELL *p;
2:
3: if ((p = malloc(sizeof(struct CELL))) == NULL)
4:     fatal_error("メモリが足りない");
5:     セルに値をセットする
6:     p->next = x->next;
7:     x->next = p;
```

(2) リストの先頭に新しいセルを挿入する

・変数headerがリストの先頭を指しているものとする。

```
1: struct CELL *header, *p;
2:
3: if ((p = malloc(sizeof(struct CELL))) == NULL)
4:     fatal_error("メモリが足りない");
5:     セルに値をセットする
6:     p->next = header;
7:     header = p;
```

ます。そこで、3~4行目で、関数mallocの返す値がNULLであるかをチェックしています。mallocのように、エラーが発生したときに、それを通知する特別な値を返すような関数については、必ずチェックを行うのが鉄則です。

このようなチェックを忘れても、テスト中は扱うデータが少なく、メモリを使い切ることはまずないので、なんら不都合は生じないでしょう。しかし、デバッグがすんで本格的に使用を開始して大量のデータを入力するとメモリを使い切ってしまい、このような潜在的なバグが顔を出すというのはよくあることです。

このような関数はエラーを通知するのにNULLという値を使うことが多い(たとえばfopenなど)ので、メモリ保護が不十分なOSでは、システム領域を書き換えてしまい、その結果暴走する可能性が大です(特にMS-DOS上でのlarge modelのプログラムではかなりの確率で暴走します)。

反対に、きちんとメモリ保護が行われているOS(たとえばUNIX)では、システム領域への書き込みはハード的に禁止され、NULLポインタが指しているアドレスに書き込もうとした時点で、プログラムは強制的に終了させられる(coredumpする、といいます)ので、MS-DOSなどに比べると“安全”です。とはいものの、プログラムが突然停止してしまうということは、ユーザからみれば、それまでに入力したデータが失われてしまうということになるので災難にはちがいありません。このようなチェックはめんどうがらずに行うことを行慣にしてください。

メモリが不足するといった事態を除けば、連結リスト自体には長さの制限がまったくありません。これが配列の場合との大きな違いです。

さてList 5.1(1)は、連結リストの途中に新たな要素を挿入するものです。連結リストの先頭に挿入するにはList 5.1(2)のようになります。境界条件として、変数headerがNULLの場合、つまりもともとリストが空であったときも、List 5.1(2)はうまく動きます。

連結リストに要素を挿入するのに、先頭に挿入する場合と途中に挿入する場合とでは、異なった処理をしなければなりません。これは「先頭に挿入する」という境界条件を特別扱いする必要があるからです。ところで、List 5.1(1)とList 5.1(2)を見比べると、その違いは6~7行目でx->nextとなっている部分が、headerに変わっている点だけです。これは、連結リストの先頭がstruct

CELL型そのものではなく、struct CELL型へのポインタで与えられているからです。そこで、変数 header を、

```
struct CELL *header;
```

ではなく、

```
struct CELL header;
```

と定義することになると、先頭への挿入も途中への挿入もまったく同等に扱うことが可能です。つまり、先頭へ挿入するには、

```
x = &header;
```

として、List 5.1(1)を適用すればうまくいきます。

また、セルを1つずつ順番にたどって処理を行うには、

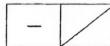
```
struct CELL *p;  
for (p = header.next; p != NULL; p = p->next)  
    printf("%d\n", p->value);
```

とします。

このような方式をとると、連結リストには最低でも1つのセルが存在することになります(Fig.5.6)。実際には、このただ1つのセルは連結リスト自身を表すためのダミーで、次のセルへのポインタのメンバのみに意味があり、その他

Fig. 5.6 セルを使ってリストの先頭を表す方法

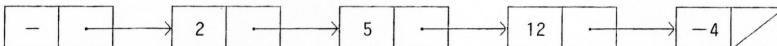
変数 header



※headerセルの左半分(セルの値が入る部分)にはゴミの値が入っている。

(a) 要素が1つもないリスト

変数 header



(b) 4つの要素(2, 5, 12, -4)をもつリスト

のメンバにはゴミの値が入っています。この方式によれば、見かけ上、最低でも1つのセルが存在することになり、境界条件をうまく扱うことができます。

● 5.1.4.2 —— 要素の削除

次に、連結リストから特定のセルを削除することを考えましょう。先ほど説明したように、連結リストでは指定した要素の次の要素が削除の対象になります。指定した要素そのものを削除することはできません。

ポインタ x で指されたセルの直後のセルを削除するには、List 5.2(1)のようにします。ここで、境界条件として、削除すべきセルが存在しないというケースを考慮する必要があります。ポインタ x が連結リストの最後のセルを指している場合が、このケースに該当します。ポインタ x 自身が最後のセルを指しているとき、当然のことながら次のセルは存在しません。ですから、この場合は削除しようにも無理な相談です。これをチェックしているのが、List 5.2(1)の3~4行目です。

List 5.2 • 連結リストからの削除

(1) ポインタ x で指されたセルの直後のセルを削除する

```
1: struct CELL *p;  
2:  
3: if (x->next == NULL)  
4:     fatal_error("最後の次にはセルはないので削除できない");  
5: p = x->next;  
6: x->next = p->next;  
7: pで指されたセルの内容を取り出す  
8: free(p);
```

(2) リストの先頭のセルを削除する

・変数 header がリストの先頭を指しているものとする。

```
1: struct CELL *header, *p;  
2:  
3: if (header == NULL)  
4:     fatal_error("リストが空なので削除できない");  
5: p = header;  
6: header = p->next;  
7: pで指されたセルの内容を取り出す  
8: free(p);
```

また、7行目の「pで指されたセルの内容を取り出す」という部分では、pで指されたセルの内容を使って必要な処理を行います(たとえば、セルの値をローカル変数にコピーしておき、後で関数の値としてリターンするなど)。最後に、8行目では、関数 free を呼び出して、削除したセルを解放します。

リストの先頭のセルを削除する手順は、List 5.2(2)のようになります。ここでは、連結リストは、先頭へのポインタ header によって管理されていることを想定しています。「リストが空である」というのが境界条件になります。3~4行目で、リストが空であればエラーにします。

挿入の場合と同様に、List 5.2(1)と List 5.2(2)の違いは、x->next が header に置き換わっただけです。削除の処理も、リストの先頭を

```
struct CELL *header;  
ではなく,
```

```
struct CELL header;  
とすることによって、先頭のセルの削除と途中のセルの削除を統一的に扱うこ  
とが可能です。つまり,
```

```
x = &header;  
として List 5.2(1)を実行すれば、先頭のセルを削除することができます。また、  
リストが空の場合も正しく扱われる(エラーになる)ことがわかります。
```

● 5.1.4.3 —— 境界条件の扱い

実際にプログラムを書くうえでは、境界条件に注意することがたいせつです。連結リストの処理では、先頭と末尾の要素の扱い、連結リストが空のときの処理が境界条件になります。

先ほどリストの先頭を示すのに、(セルへのポインタではなく)セルそのものを使えば、先頭への挿入と削除を特別扱いしないで、簡単に扱えることを示しました。また、リストの先頭をセルによって表す方法をとると、見かけ上連結リストが空になることがなくなり、プログラムを書くときに境界条件を意識しないで済みます。

ここで、数値が昇順(小→大の順)に並んでいる連結リストがあり、そこに新しい要素を追加したいものとしましょう。当然のことながら、要素の昇順の並びを保つ位置に挿入しなければなりません。セルは、

```
struct CELL {  
    struct CELL *next;  
    int         value;  
};
```

と定義されているものとします。また、次のように変数 header によって連結リストの先頭が与えられています。

```
struct CELL header;
```

この連結リストに新しい要素を追加する関数 insert を List 5.3 に示します。連結リストをたどるときに 2 つのポインタ変数 p と q を使うというのが、このプログラムのミソです。ポインタ p は、現在着目しているセルを指しています。もう一方のポインタ q は、p が指すセルの直前のセルを指すようになっています。

連結リストには、指定したセルの直後の挿入はできるが、直前の挿入はできないという性質がありました。そこで、関数 insert では着目するセルの直前のセルへのポインタをもつことによって、この問題を解決しています。

List 5.3 ● 関数 insert

```
struct CELL {  
    struct CELL *next;  
    int         value;  
} header;  
  
insert(int a)  
{  
    struct CELL *p, *q, *new;  
  
    /* 挿入すべき場所を探す */  
    p = header.next;  
    q = &header;  
    while (p != NULL && a > p->value) {  
        q = p;  
        p = p->next;  
    }  
  
    /* セルを挿入する */  
    if ((new = malloc(sizeof(struct CELL))) == NULL)  
        fatal_error("メモリが足りません");  
    new->next = p;  
    new->value = a;  
    q->next = new;  
}
```

まずは、ポインタ q がないものとして考えると、

```
p = header.next;
while (p != NULL && a > p->value)
    p = p->next;
```

となります。この while 文を実行し終えた時点では、

- ① p には NULL が入っている
- ② セルへのポインタが入っている

のいずれかになります。①のケースは、変数 a の値が既存のどのすべての要素よりも大きかった場合です。また、リストが空であった場合も p は NULL になります。②は、変数 a よりも大きい(より正確には大きいか、あるいは等しい)値をもつ要素が見つかった場合です。このときは新しい要素を p の直前に挿入する必要があります。

次に、ポインタ q の役目を考えてみましょう。連結リストにはあらかじめ 2, 5, 7, 12 がこの順に登録されていることにします。まず、while ループに入る前に、

```
p = header.next;
q = &header;
```

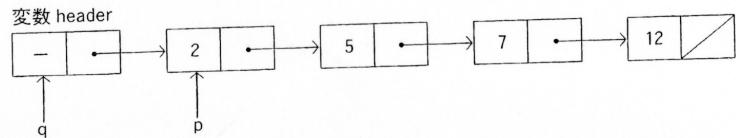
と初期化します。この時点では Fig.5.7(1) のようになっています。また、while ループを 1 回実行するたびに、

```
q = p;
p = p->next;
```

としているので、ポインタ q がポインタ p を追いかけるかたちになります。たとえば、a が 4 であった場合、p が value=5 のセル、q が value=2 のセルを指した状態で while ループを抜け出します。そして関数 insert の後半が実行されて、q が指すセルと p が指すセルの間に新しいセルが挿入されます(Fig. 5.7(2))。

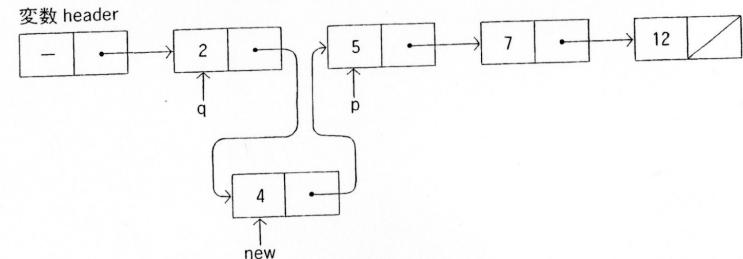
同様にして、連結リストが空のとき(Fig.5.7(3)), 先頭に挿入されるとき(Fig.5.7(4)), 末尾に挿入されるとき(Fig.5.7(5))の各場合にも、処理が正しく行われます。連結リストの先頭を(ポインタではなく)セルそのものによって表現すると、境界条件に対して特別な処理をしないでも正しく処理することができます。

Fig. 5.7 関数 insert の動作と境界条件(その1)

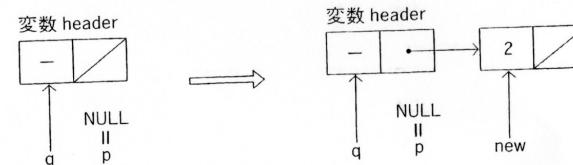


・ p は 1 番目のセルを指し、q は header を指す。

(1) p と q を初期化した状態



(2) value=5 のセルの直前に value=4 のセルを挿入



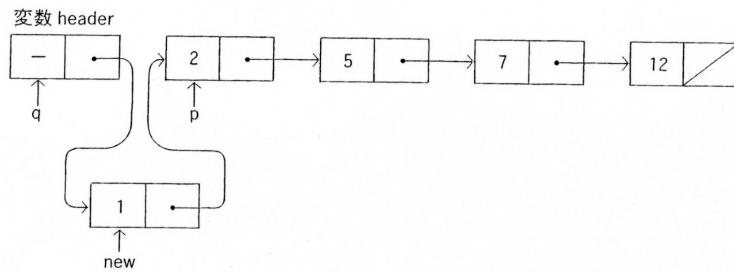
・ 空リストへ value=2 のセルを挿入する。

・ 左が挿入前、右が挿入後の状態である。

・ 変数 header がセルと同じ形式になっているので、特別扱いしなくても正しく処理される。

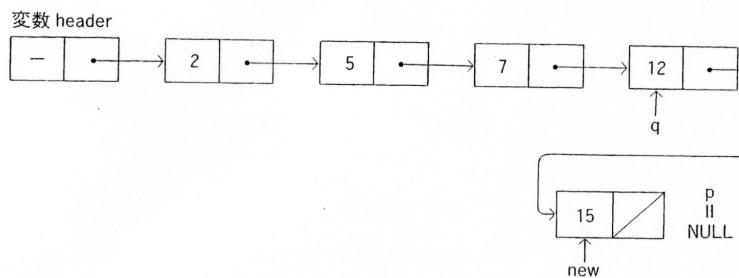
(3) 境界条件その1——空リストへの挿入

Fig. 5.7 関数 insert の動作と境界条件(その2)



- ・value=1のセルを追加する。このセルはリストの先頭に挿入される。
- ・変数 header がセルと同じ形式になっているので、特別扱いしなくても正しく処理される。

(4) 境界条件その2——リストの先頭への挿入



- ・value=15のセルを追加する。このセルはリストの末尾に追加される。

(5) 境界条件その3——リストの末尾への挿入

きます。もし、ポインタで表現したとすると、リストの先頭への挿入と空リストへの挿入を特別扱いにしなければなりません。

この例のように、一方が他方を追いかけるという関係にある2つのポインターを使ってリストをスキャンするというのは、連結リストを処理するための定石ということができます。そして、セルを使って連結リストの先頭を表す手法を用いることによって、境界条件を特に意識することなくコーディングを行うことができます。

5.2 | 循環リスト |

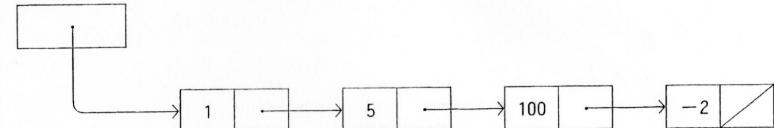
5.2.1 ■ 循環リストとは?

連結リストでは、それぞれの要素(セル)は、次の要素へのポインタをもっています。最後の要素の場合には、次にくる要素がありませんから、NULLがセットされます(Fig.5.8(a))。ここで、NULLの代わりに、最初の要素へのポインターをセットしてみましょう(Fig.5.8(b))。Fig.5.8(a)のような普通の連結リストでは各要素は1列に並んでいますが、Fig.5.8(b)のリストでは各要素が環状に連結されることになります。これを、**循環リスト**(circular list)と呼びます。

循環リストの本来の目的は、環状に並んだデータ構造を表現することです。要素を結び合わせるポインターをたどれば、循環リストに含まれる全部の要素を順番に処理することができます。循環リストでは、各要素はリング状に結合されているので、厳密にいえば「最初」や「最後」の要素というものは存在しないこ

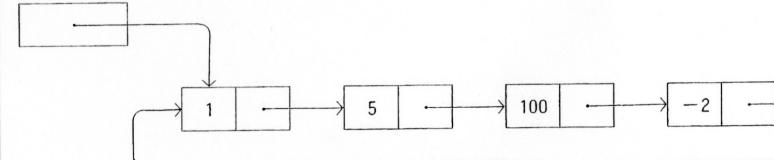
Fig. 5.8 連結リストと循環リスト

変数 ptr



(a) 普通の連結リスト

変数 ptr



(b) 循環リスト