3.3. 線形リストの実装

3.3.1. 線形リストとは

線形リストは、「データをポインタにより鎖状に結びつけたデータ構造」です。配列の場合、要素の順番を添字によって示していましたが、線形リストでは各要素(ノード)に次の要素へのアドレス(リンク)を保持し、そのアドレスによって順番を示しています。このため、線形リストの領域は配列のように連続した領域ではなく、ノードごとに不連続な領域でもかまいません。ただし、鎖状に結びつけられたノード群だけでは線形リストの先頭ノードと終端ノードが分からないため、先頭ノードのアドレスを持つ**ヘッダ**(ポインタ)を設け、終端ノードのアドレス部にはどのノードも示さないというアドレス(C 言語の場合は、NULL)を設定するようにしています。また、ノード領域は必要に応じて malloc 関数で確保し、不要になった時点で free 関数で解放すれば、領域を有効に使用できます。

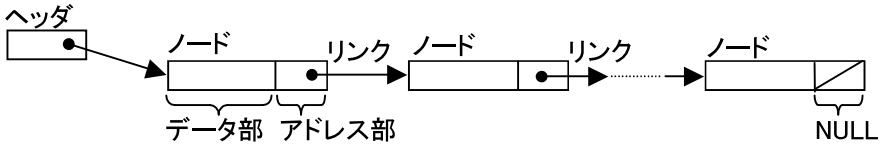

図 3.3-1 線形リスト

3.3.2. 配列と線形リストの違い

配列と線形リストの違いを表 3.3-1 に示します。

表 3.3-1 配列と線形リストの特徴比較

比較項目	配列	線形リスト
要素数の変更	領域の再確保(realloc)が必要になる。	領域の再確保をせずに変更できる。
要素の挿入・削除	挿入・削除したい要素より後ろの要素をすべてずらす必要がある。	リンク張り替えだけで挿入・削除ができるため、複雑なメモリ操作は不要。
要素の参照	添字によりランダムに参照できる。	先頭(または終端)から順に参照したい要素まで辿る必要がある。

C 言語の文法としてサポートされている配列に対し、線形リストを使用する場合は外部のライブラリを使用するか、本節でするように自作する必要があります。配列も線形リストもそれぞれ長所と短所がありますので、表 3.3-1 を目安に使い分けると良いでしょう。

3.3.3. 線形リストの種類

(1) 一方向リスト(単方向リスト)

一方向リストは、ノード内に次のノードのアドレスを持ったデータ構造です。従って、次のノードは簡単に参照できますが、前のノードを参照するためには、前のノードのアドレスを格納しておくポインタが必要です。線形リストの中では、操作が簡単で、分かりやすい線形リストです。

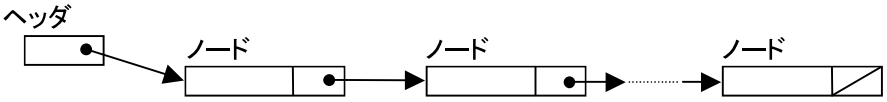

図 3.3-2 一方向リスト

(2) 両方向リスト(双方向リスト)

両方向リストは、ノード内に次のノードのアドレスと前のノードのアドレスを持ったデータ構造です。従って、一方向リストとは違い、前後のノードを簡単に参照できます。しかし、ノードを操作するとき、2つのアドレスを扱うことになるので、一方向リストに比べ操作が面倒で、分かりにくい線形リストです。

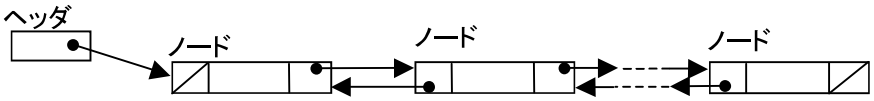

図 3.3-3 両方向リスト

(3) 環状リスト(循環リスト)

環状リストは、一方向リストと同じくノード内に次のノードのアドレスを持ったデータ構造です。しかし、一方向リストとは違い、終端ノードは、先頭ノードのアドレスを持っています。従って、ノード全体で 1 つの輪を形成しています。環状リストは、発生するデータに対し、巡回的に処理を施したい場合に用いられます。

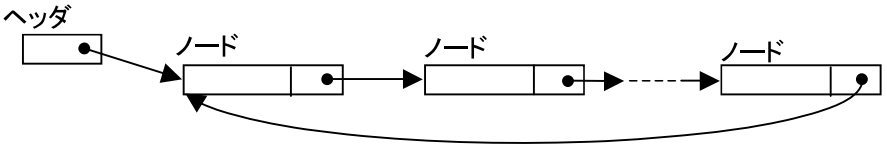

図 3.3-4 環状リスト

本コースでは線形リストとして一方向リストを扱います。以降、一方向の線形リストを単に線形リストと記すことにします。

3.3.4. ノードの実装

線形リストのノードは、次のノードへのポインタを要素に持つ構造体で実現できます。int 型データ(整数データ)を入れるノードの定義を次に示します。

```
struct Node {
    int data;
    struct Node *link;
};
```

ノードにはデータ部とアドレス部があります。

データ部を表すメンバが int data です。データ部の型は、線形リストに入れたいデータの型にします。もし、double 型データ(実数データ)を入れたい場合は double data とします。

アドレス部を表すメンバが struct Node *link です。

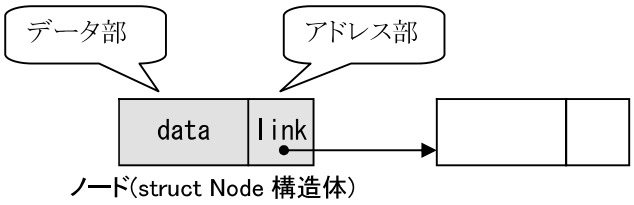

図 3.3-5 線形リストのノードの実装

線形リストのノードをひとつ作成して、データを参照するサンプルプログラムを示します。

例 3.3-1 線形リストのノードの作成

```
#include <stdio.h>
#include <stdlib.h>          /* malloc 関数、free 関数を使用するために include する */

/* 線形リストのノードを表す構造体 */
struct Node {
    int data;                /* データ部 */
    struct Node *link;       /* アドレス部 */
};

int main(void)
{
    struct Node *nodep;                                          /* ノードを指すポインタ */
    nodep = (struct Node *)malloc(sizeof(struct Node)); /* ノードの領域を確保する */
                                                                 /* 領域確保失敗時の処理は省略 */
    nodep->data = 50;                                            /* データ部にデータを代入する */
    nodep->link = NULL;                                          /* アドレス部に NULL を代入する */

    printf("%d  ¥n", nodep->data);                               /* データ部を出力する */

    free(nodep);                                                 /* ノードの領域を解放する */
    return 0;
}
```

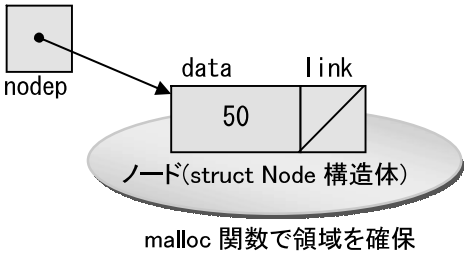

図 3.3-6 サンプルプログラムのイメージ

ソースコードの見やすさのため、以降では領域確保失敗時の処理は省略します。

続いて、線形リストのノードを 3 個作成して、ノードを繋ぐサンプルプログラムを示します。

例 3.3-2 線形リストのノードの作成とノードのリンク

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *link;
};

int main(void)
{
    struct Node *node1p;
    struct Node *node2p;
    struct Node *node3p;

    /* 1 個目のノード */
    node1p = (struct Node *)malloc(sizeof(struct Node));
    node1p->data = 50;
    node1p->link = NULL;
    /* 2 個目のノード */
    node2p = (struct Node *)malloc(sizeof(struct Node));
    node2p->data = 100;
    node2p->link = NULL;
    /* 3 個目のノード */
    node3p = (struct Node *)malloc(sizeof(struct Node));
    node3p->data = 150;
    node3p->link = NULL;

    /* 1 個目のノード → 2 個目のノードとなるように繋ぐ */
    node1p->link = node2p;
    /* 2 個目のノード → 3 個目のノードとなるように繋ぐ */
    node2p->link = node3p;

    printf("%d %d %d¥n",
        node1p->data, node1p->link->data, node1p->link->link->data);

    free(node1p);
    free(node2p);
    free(node3p);

    return 0;
}
```

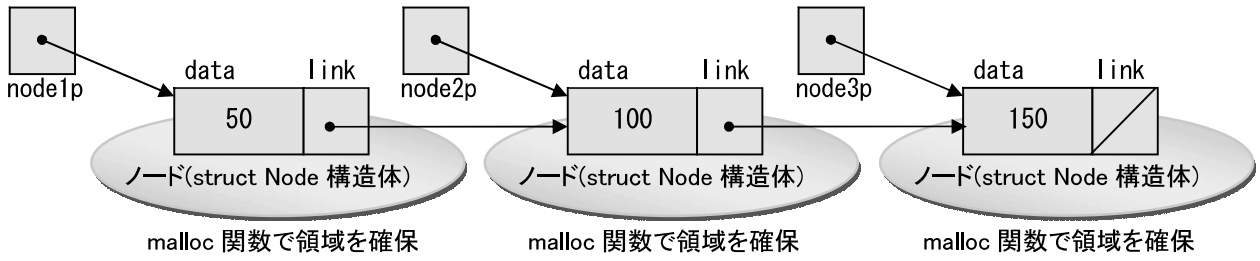

図 3.3-7 サンプルプログラムのイメージ

このサンプルプログラムには大きく 2 点の問題があります。

(1) 3 番目のノードの値を出力するために node1p->link->link->data としているが、この方式ではノードが増えるほどコードが長くなってしまう。

(2) 3 個のノードを扱うために node1p、node2p、node3p という 3 個のポインタを使用しているが、上記イメージを見ると、線形リストとしては最終的に node2p、node3p が不要。

次項以降で、この 2 点の問題を解決します。

3.3.5. 線形リストの巡回

まずは前項の問題(1)「3番目のノードの値を出力するためにnode1p->link->link->dataとしているが、この方式ではノードが増えるほどコードが長くなってしまう。」を解決するために、線形リストを先頭から最後まで巡回する方法を紹介します。

次に巡回のサンプルプログラムを示します。

例 3.3-3 線形リストの巡回

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *link;
};

int main(void)
{
    struct Node *node1p;
    struct Node *node2p;
    struct Node *node3p;
    struct Node *curp;                  /* 作業用のポインタ */
    /* 1個目のノード */
    node1p = (struct Node *)malloc(sizeof(struct Node));
    node1p->data = 50;
    node1p->link = NULL;
    /* 2個目のノード */
    node2p = (struct Node *)malloc(sizeof(struct Node));
    node2p->data = 100;
    node2p->link = NULL;
    /* 3個目のノード */
    node3p = (struct Node *)malloc(sizeof(struct Node));
    node3p->data = 150;
    node3p->link = NULL;
    /* 1個目のノード → 2個目のノードとなるように繋ぐ */
    node1p->link = node2p;
    /* 2個目のノード → 3個目のノードとなるように繋ぐ */
    node2p->link = node3p;

    curp = node1p;                      /* ① */
    while(curp != NULL) {               /* ②⑤⑧⑪ */
        printf("%d ", curp->data);      /* ③⑥⑨ */
        curp = curp->link;              /* ④⑦⑩ */
    }

    free(node1p);
    free(node2p);
    free(node3p);

    return 0;
}
```

線形リストの巡回は上記網掛け部分です。①→⑪の順に番号順に処理されます。この処理はノードを指し示す作業用のポインタをひとつ宣言することで実現できます。処理を順に追ってみましょう。

● 作業用ポインタの初期化(①の処理)

先頭ノードから巡回するために、curp に先頭ノードのアドレスを代入しています。

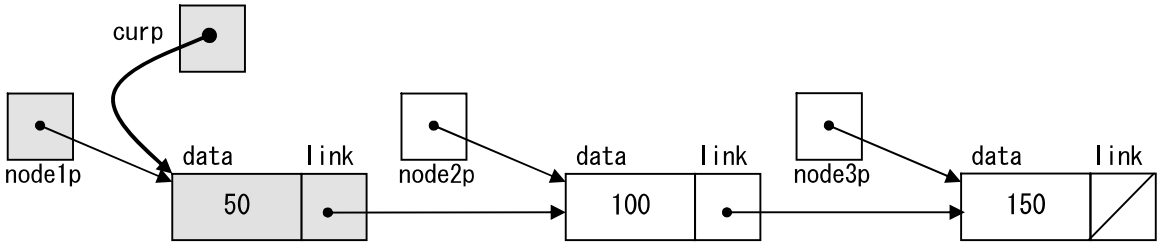

図 3.3-8 線形リスト巡回のイメージ(1)

● 線形リストの終わりまで達したかの判定(②⑤⑧⑪の処理)

後述の次のノードを指し示す処理(④⑦⑩の処理)により、curp が指すノードが後方へとずれていきますが、終了条件が無いと永遠と後方を辿り続けようとしてしまいます。しかし線形リストの長さは有限ですので、curp がノードをすべて巡回し、次のノードを指し示さなくなった時点でループを終了させるようにします。

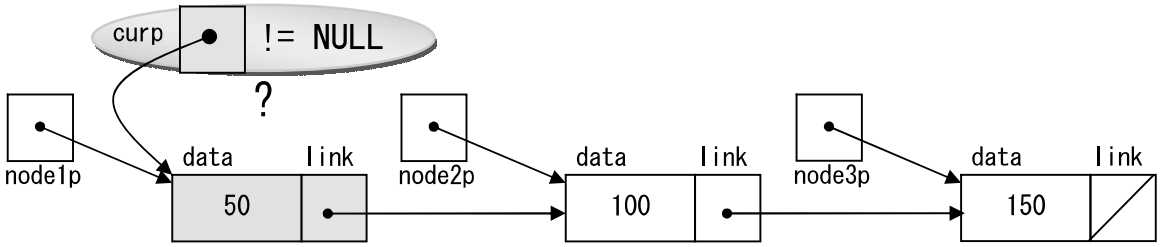

図 3.3-9 線形リスト巡回のイメージ(2)

● 線形リストのノードの値を出力(③⑥⑨の処理)

curp が指しているノードのデータ部を出力します。図は③の場合を示しています。

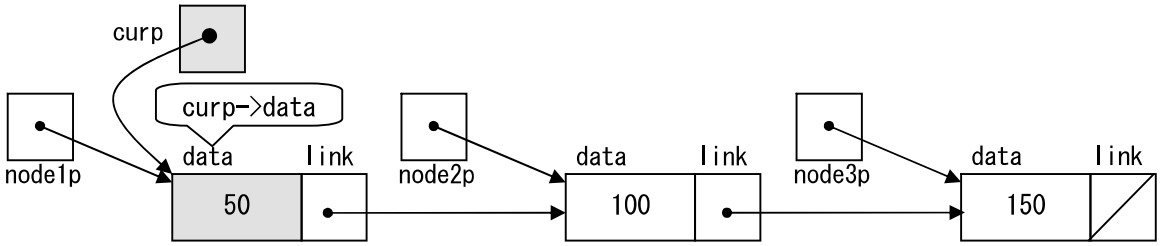

図 3.3-10 線形リスト巡回のイメージ(3)

●次のノードを指し示す(④⑦⑩の処理)

現在の curp に、curp->link の指している次のノードのアドレスを代入しなおすことで、curp が次のノードを指し示すようにします。次の図は④の場合を示しています。

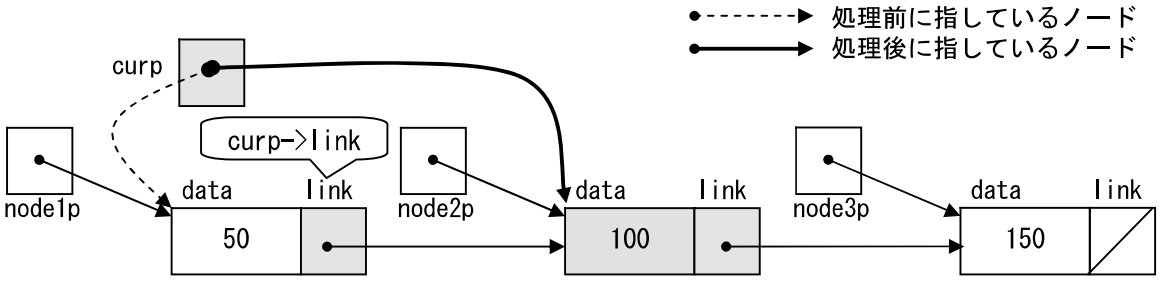

図 3.3-11 線形リスト巡回のイメージ(4)

curp が最後のノードを指しているとき(⑩の場合)は、次の図のように curp->link が NULL ですので、curp には NULL が代入されます。

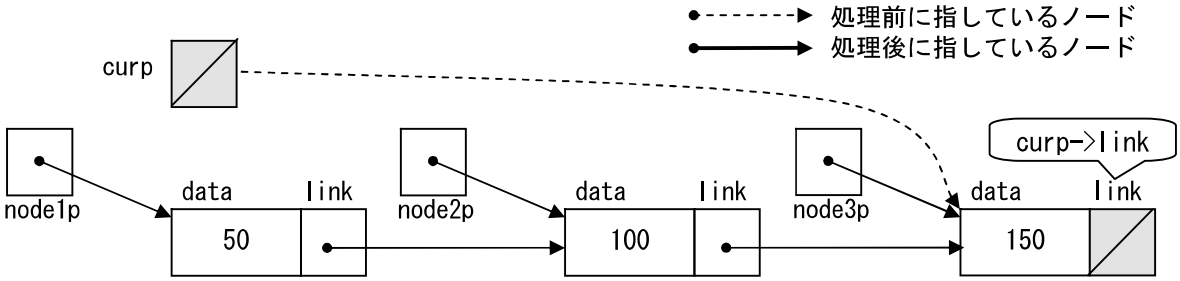

図 3.3-12 線形リスト巡回のイメージ(5)

3.3.6. ノードの追加

本項ではノードを追加する方法を示します。ここで示す方法により、前項までの問題(2)「3 個のノードを扱うために node1p、node2p、node3p という 3 個のポインタを使用しているが、上記イメージを見ると、線形リストとしては最終的に node2p、node3p が不要。」を解決できます。

線形リストを作成(ノードを追加)する方法としては、次の 3 通りが考えられます。

(1) 線形リストの先頭にノードを追加
(2) 線形リストの終端にノードを追加
(3) 線形リストの特定の位置にノードを挿入

本項では(2)の終端にノードを追加するサンプルプログラムを示します。

例 3.3-4 線形リストの終端にノードを追加

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *link;
};

int main(void)
{
    struct Node *head;      /* 線形リストの先頭を指すポインタ */
    struct Node *tail;      /* 線形リストの終端を指すポインタ */
    struct Node *newp;      /* 新しく作成するノードを指すポインタ */
    struct Node *curp;      /* 作業用のポインタ */
    /* 1個目のノード */
    newp = (struct Node*)malloc(sizeof(struct Node));
    newp->data = 50;
    newp->link = NULL;
    /* 線形リストの先頭を指すポインタ → 1個目のノードとなるように繋ぐ */
    head = newp;                        /* ① */
    tail = newp;                        /* ② */
    /* 2個目のノード */
    newp = (struct Node*)malloc(sizeof(struct Node));
    newp->data = 100;
    newp->link = NULL;
    /* 1個目のノード → 2個目のノードとなるように繋ぐ */
    tail->link = newp;                  /* ③ */
    tail = newp;                        /* ④ */
    /* 3個目のノード */
    newp = (struct Node*)malloc(sizeof(struct Node));
    newp->data = 150;
    newp->link = NULL;
    /* 2個目のノード → 3個目のノードとなるように繋ぐ */
    tail->link = newp;                  /* ⑤ */
    tail = newp;                        /* ⑥ */
    curp = head;
    while(curp != NULL) {
        printf("%d ", curp->data);
        curp = curp->link;
    }
    while(head != NULL) {
        curp = head;
        head = head->link;
        free(curp);
    }
    return 0;
}
```

ノードの追加は上記網掛け部分のうち①から⑥の部分です。この処理のために、線形リストの先頭を表すポインタ head と、作業用の 2 個のポインタ(tail、newp)を宣言しています。処理を順に追ってみましょう。

●線形リストの先頭を指すポインタに1個目のノードを繋ぐ(①②の処理)

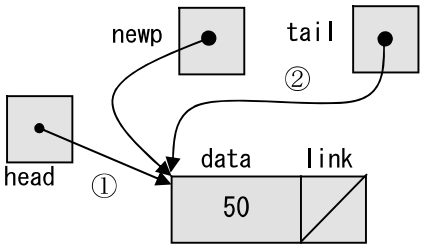

図 3.3-13 ノードの追加のイメージ(1)

●1個目のノードの後ろに2個目のノードを繋ぐ(③④の処理)

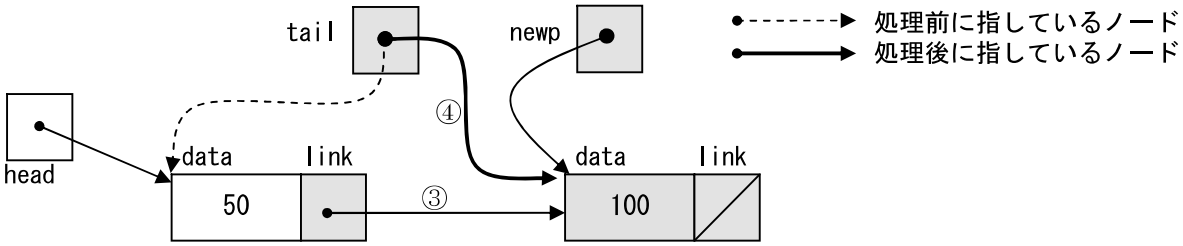

図 3.3-14 ノードの追加のイメージ(2)

●2個目のノードの後ろに3個目のノードを繋ぐ(⑤⑥の処理)

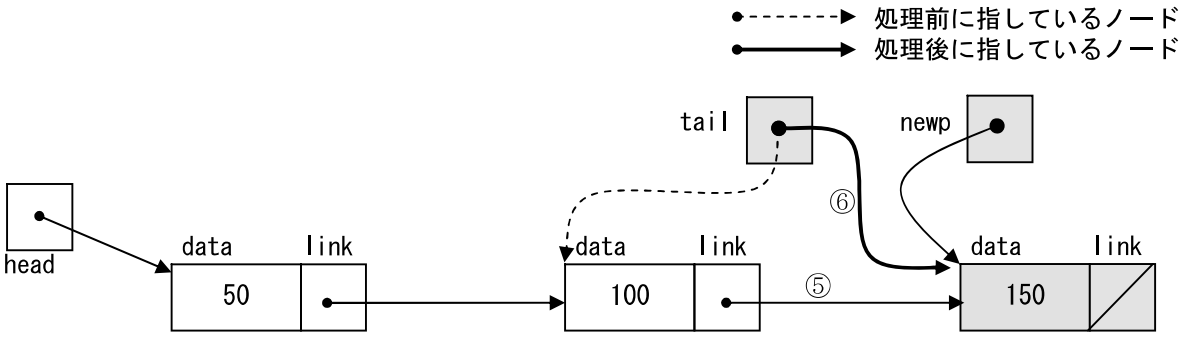

図 3.3-15 ノードの追加のイメージ(3)

なおこのサンプルプログラムでは、線形リストを巡回する方法を応用してノードの領域を解放しています。
この方法であればノードを4個5個・・・と追加していっても、作業用にnewpとtailという2個のポインタを用意するだけで済みます。