3　リスト (List) 構造の設計と実装 (補足)

リストで表せるデータ構造の中で、それに対して限られた操作しか行なわない特殊な構造があり、良く使われる重要なものなので、ここで補足しておく。

3–7　スタック (stack)

リストの先頭でしか挿入・削除を行なわないものを、**プッシュダウンスタック (pushdown stack)**、又は単に**スタック (stack)** と呼ぶ。即ち、次の二つの操作のみが許される。

- 新しい要素を先頭に挿入する (**プッシュ (push)**)
- 先頭の要素を取り出して削除する (**ポップ (pop)**)

すると、後から挿入された要素が先に取り出されることになる。これを **last-in first-out (LIFO)** と呼ぶ。実装に際しては、この二つの操作を行なう関数 `push()`, `pop()`、及び、スタックが空であるかどうかを判定する関数 `isstackempty()` を用意することとなろう。プロトタイプ及びスタックの初期化は次のようにするのが一例である。(ダミー節点を用いる場合。要素の型は `int` 型とした。)

```
typedef struct node *Link;
struct node {
    int val;
    Link next;
};

void push( Link, int );
int pop( Link );
int isstackempty( Link );
```

```
Link head=NULL;

head = (Link)malloc(sizeof(struct node)); /* dummy node */
head->next = NULL;
```

前回の課題の締切は次回の授業の前日 (11/3)。解答例は次回掲載の予定。既に提出した人も改良できたら再提出歓迎。
ヒント: 先頭から順に辿っていって新節点を挿入する場所を決める訳だが、この問題の意外に厄介な所は、その辿っていく loop の終了条件が

- 入力値より大きな値に辿り着く
- 末端 (`NULL`) まで到達する

の 2 通りあること。これをどう扱うか。

---

stack: 積み重ね (る)
新着書類がどんどん積まれていって、上から片付けていく様子を思い浮かべよ。

型定義は従来通りの書き方でも可。

`isstackempty()` は引数のリストが空 (要素を持たない) なら 1 (真) を、そうでないなら 0 (偽) を返し、`if` 文などの条件式として、次のように用いることを想定している。

```
if ( !isstackempty(head) ){
    a = pop(head);
      ...
}
```

スタックは計算機内部でも頻繁に用いられている重要なデータ構造である。例えば、関数呼出に於いては、現在のレジスタの値や実行後に戻るべき場所など、覚えておくべきデータを所定のスタックに所定の順番で積んでおき、関数の実行終了後、スタックからポップして呼び出す前の状況を復元している。関数を入れ子で呼出すと、そのスタックに沢山のデータが積まれることになる。

演習 3. スタックの基本的操作の関数 `push()`, `pop()`, `isstackempty()` を作成せよ。動作確認の為のサンプルプログラムとしては、極く原始的に「入力値を次々にスタックにプッシュし、入力が終了したら、次々とポップして値を表示する」で良かろう (正に LIFO を実感できる筈)。

演習 4. スタックでは、要素を途中に挿入したり途中から削除したりということがないので、リストの形でなく、配列によって実装することも大いに考えられる (演習 5 参照)。これを試みよ。(ヒント: 先頭の要素の添字を保持する変数を用意する。配列は大きめにとっておくことにしよう。)

3–8　キュー (queue)

リストの末尾への追加と先頭からの削除としか行なわないものを、**FIFO キュー (FIFO queue)**、又は単に**キュー (queue)** と呼ぶ。すると、最初に挿入された要素が先に取り出されることになる。これを **first-in first-out (FIFO)** と呼ぶ訳である。実装等の詳細は省略。同様の理由で配列での実装も有力である。

queue: (順番待ちの) 列
リストのように要素が一列にならんでいる構造を、一般にキューと呼ぶことも多い。
順番待ちの列がスタック (LIFO) だったら、最初に来て待ってる人は怒ってしまいますね。

演習 5. 配列とリストとについて、それぞれの長所・短所をまとめ、どのような場合にどちらを使うのがどんな意味で効率的か、例を挙げて考察せよ。

4　木 (tree) 構造

4–1　木構造とは

リストでは各節点の「次」は一つだけであったが、これを幾つもに枝分かれすることを許したものが「**木 (き, tree)**」と呼ばれる構造である。一般には「次」は幾つあっても良いが、以下では簡単の為、「次」が 2 つである場合 (**二分木 (binary tree)**) を専ら考える。

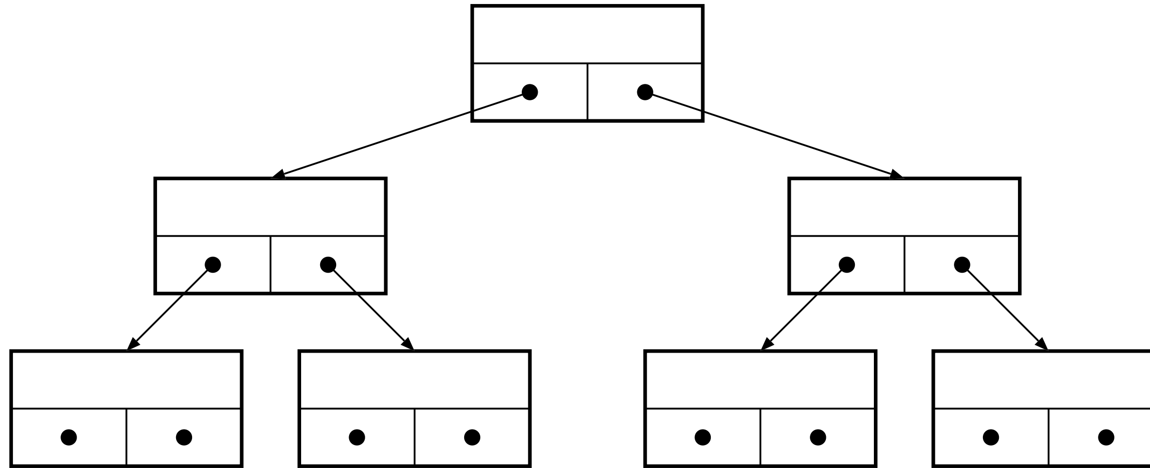

図 5: 木構造 (二分木) の概念図

リストの先頭に相当する出発点の節点を**根 (ね, root)**、節点を繋ぐ矢印を**枝 (えだ)・辺 (edge)**、末端の節点 (つまり「次」が全て NULL なもの) を**葉 (は, leaf)** などど、縁語で呼ぶことが多い。

木の各節点のどの枝に対しても、その先にあるのは、それ自身が再び木である。これを元の木の**部分木 (subtree)** と呼ぶべきなのは、数学科の学生であれば当然であろう。

グラフ理論の言葉で言えば、木とは閉路のない連結グラフ (1 次元複体) であって、特にここで考えているのは、全ての辺に向きが指定されている**有向木 (oriented tree)** であり、中でも根から出発して辺の向きに順ってどの節点にも行ける**根付き木 (rooted tree)** である。

4–2　節点の設計

例として、int 型の値を要素として持つ二分木の実装を考える。基本的な考えはリストの節点と同じであるが、「次」が 2 つある (ここでは「左」「右」としよう) ので、構造体のメンバを増やせば良い。ここでは節点の構造体型 struct node へのポインタ型を、Link 型として型名定義し、これを通常使う型としよう。

実際、今までに扱ってきたリスト構造の場合も、要素へのアクセスや関数呼出は「節点へのポインタ」によって行なうことが殆どであった。リストの場合も、実際には「節点へのポインタ」を通常使う型とする方針で実装する方が多い (んじゃないかな)。

```
typedef struct node {
    int val;
    struct node *left;        /* "struct" is necessary */
    struct node *right;
} *Link;
```

木の基本操作を説明する為の例題として、先に課題を提示しておこう。

課題 4 (〆切 11/10(月)). 二分木を用いて、次々と入力した整数値を大きさの順に並べ替えて表示するプログラム tree.c を作成せよ。

説明は多分次回に続くと思うので、〆切は次々回授業の前日としておく。

- 木のどの節点についても次の条件が成り立つように、入力した整数を値とする新節点を、木の適切な先端に追加する。
  - ⋆ その左にある部分木のどの要素の値も、その節点の要素の値より小さい。
  - ⋆ その右にある部分木のどの要素の値も、その節点の要素の値より大きい。
- 0 が入力されたら終了。(0 を値とする節点は作らない。)
- 先頭から木を辿って全ての値を順に表示する。

入力データ数を n 個 (充分大) とする時、比較の回数は如何程と見積もれるか。(都合の良い場合・最悪の場合・平均の場合)

実習 4.1. コーディングする前に、適当なデータを例に、「箱と矢印の図式」を自分で書いてみて、木の「成長」の仕組みを理解せよ。

4–3　節点の追加

木の場合には、新要素の追加は (途中に挿入するのでなく) その先端で行なうのが普通である。リストと異なりその先端が多数あるので、その中のどれかを選ぶ訳である。左右のどちらを辿るかの選択がある他は、先端まで辿る要領はリストと同じ。次のような関数 `new_Node()` を作っておくのが良かろう。

(1) 動的に領域確保を行ない、
(2) 要素の値を設定し、
(3) 左右のリンクを `NULL` に初期化して、
(4) 節点へのポインタを返す。

4–4　木の走査・表示

リストと異なり枝分かれがあるので、全ての節点を走査して表示するのは自明な問題ではない。今の問題では、「左が小さく右が大きい」という作り方なので、これを大きさの順に表示するには、次の方針となるだろう。

(1) 左の部分木を表示
(2) 自分自身の要素を表示
(3) 右の部分木を表示

これを**そのまま**プログラムにすると、次のようになるだろう。

```
void print_tree( Link p )        /* not complete!! */
{
    print_tree(p->left);
    printf("-> %d ", p->val);
    print_tree(p->right);

    return;
}
```

この例では、関数 print_tree() の中で関数 print_tree()(つまり自分自身) を呼び出している。こんなことをして良いのだろうか。

実はこんな呼出をしても構わない (というか、それが出来ないと容易には書けない)。このように自分自身を呼び出すことを**再帰呼出 (recursive function call)** という。構わないのではあるが、心配なのは自分自身を呼び出し続けて止まらなくなること (いわゆる無限ループ) であり、然るべき終了条件を満たしたら、順次引き揚げて来なくてはならない。実際、先の例ではポインタを辿りながら自分自身を呼び出し続け、遂には末端に達して NULL を辿ろうとして実行時エラーになるだろう。

要は **(数学的) 帰納法**である。

今の例では、木の末端 NULL まで来たらもう何もしなくて良いのだから、そこで止まって引返すことになる。修正版の例は次の通り。

```
void print_tree( Link p )
{
    if ( p == NULL )
        return;

    print_tree(p->left);
    printf("-> %d ", p->val);
    print_tree(p->right);

    return;
}
```

このように途中に return 文を置いて、帰りたい時にとっとと帰ってしまって構わない。また、中身の 3 行全体を

```
if ( p != NULL )
{
    ...
}
```

で囲む手もあるだろう。

もう一つの考え方として、自分自身を呼ぶ前にチェックして、NULL なら呼ばない、という手もあるが、どっちもどっち。帰納法で、$n = 0$ を出発点にするか、$n = 1$ を出発点にするか、という程度の違いと言えよう。

考察 4.1.1. (二分木と限らず) 木構造を持つと見るべきデータの具体例を、身の回りで探せ。

考察 4.1.2. 例えば、3 + 4 * 2 とか 3 * 4 + 2 とかのように、二項演算子で書かれた式は、内在的に二分木の構造を持っている。それを図に表せ。また、その式の値を計算するにはどうしたらよいか。

(上記 2 つの考察は次回までの宿題とする。)

演習 6. 今回は木の走査は再帰を利用して行なったが、スタック・キューを利用して、後で走査すべき所を覚えておくことにより、再帰呼出を用いずに実装することも可能である。これを試みよ。スタックの場合とキューの場合とで走査順はどう変わるか。