14 線形リストを使ったスタックを作ってみる

後期で扱った内容の総まとめとして、線形リストを使ったスタックを作ってみましょう。

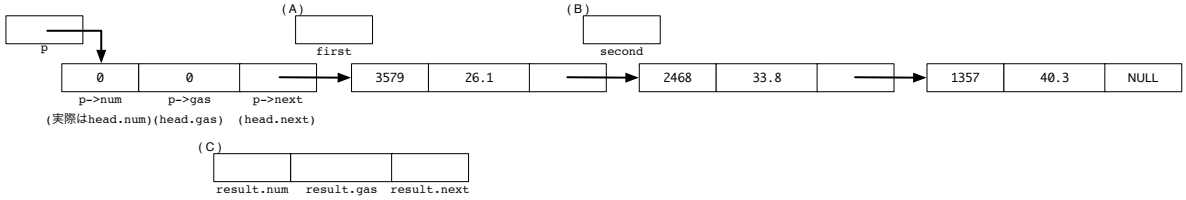
14.1 データを格納する操作「push」を作ってみる

第 13 週に作成した「リストに要素を格納する処理」がスタックの「push」に相当します。処理の様子は第 13 週の解説を参照して下さい。

14.2 データを取り出す操作「pop」を作ってみる

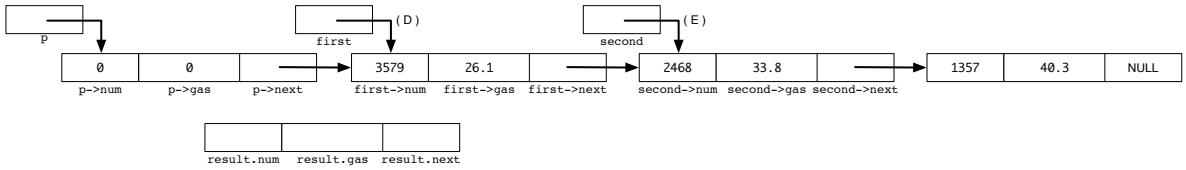
【手順 1】必要な変数を宣言する

処理するために必要な変数（下図 (A), (B), (C) の変数）を宣言します。



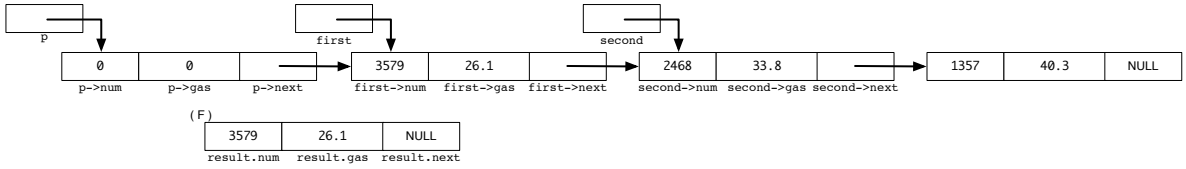
【手順 2】ポインタで参照する

スタックの 1 番目の要素を first で参照します（下図 (D) の参照）。2 番目の要素を second で参照します（下図 (E) の参照）。



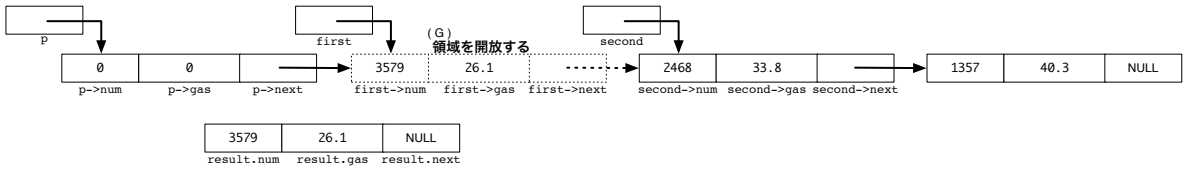
【手順 3】取り出す値をコピーする

first が参照している要素のメンバ num と gas を、result のそれぞれのメンバに値をコピーします（下図 (F) のように各メンバを代入）。



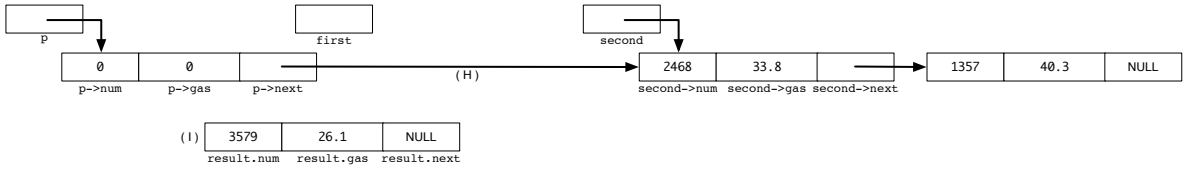
【手順 4】不要な要素をメモリから解放する

first が参照している要素（下図 (G) の部分）は不要となるため、関数 free() を使ってメモリから解放します。



【手順 5】ポインタの参照を変更し、取り出した値を返す

second が参照している要素が、スタックの先頭となるように、p->next が second の要素を参照するようにします（下図 (H) の参照）。最後に、この関数の戻り値として、result（下図 (I) の変数）を返します。



14.3 課題

【課題 14-1】 課題 13-1 で作成した関数 add_car2() の名前を変更して、「スタックにデータを push する」関数 push_car() を作成してください。この関数のプロトタイプ宣言は以下のようになります。また、「スタックを出力する」関数 show_carlist() は、第 13 週で作った関数をそのまま使います。

```
void push_car(Car *p, int n, double g);
  /* 課題 13-1 の add_car() の関数名を変える（内容は全て同じ） */
```

main() で動作を確認してください。

```
[main での処理]
 Car head1, car1;
 head1.num = 0; head1.gas = 0;        /* スタックの head を作成 */
 head1.next = NULL;
 show_carlist(&head1, "head1 (1)");  /* 空のスタックを出力 */
 push_car(&head1, 1357, 40.3);        /* スタックに 3 回 push する */
 push_car(&head1, 2468, 33.8);
 push_car(&head1, 3579, 26.1);
 show_carlist(&head1, "head1 (2)");  /* push 後のスタックを出力 */

[実行結果]
 --- head1 (1) ---
 num: 0, gas: 0.000000
 --- head1 (2) ---
 num: 0, gas: 0.000000
 num: 3579, gas: 26.100000
 num: 2468, gas: 33.800000
 num: 1357, gas: 40.300000
```

【課題 14-2】 14.2 節の解説を参考にして、「スタックからデータを pop する」関数 pop_car() を作成してください。この関数のプロトタイプ宣言は以下のようになります。

```
Car pop_car(Car *p);
  /* 14.2 節の手順 1〜5 の処理を作る */
```

main() で動作を確認してください。

```
[main での処理（課題 14-1 の続きに書いた場合）]
 car1 = pop_car(&head1);  /* pop して取り出したデータを car1 に代入 */
 printf("\n(pop) num: %d, gas: %lf\n", car1.num, car1.gas);  /* 取り出したデータを出力 */
 show_carlist(&head1, "head1 (3)");                          /* pop 後のスタックを出力 */
 car1 = pop_car(&head1);                                     /* もう一回 pop する */
 printf("\n(pop) num: %d, gas: %lf\n", car1.num, car1.gas);
 show_carlist(&head1, "head1 (4)");

[実行結果]
 (pop) num: 3579, gas: 26.100000
 --- head1 (3) ---
 num: 0, gas: 0.000000
 num: 2468, gas: 33.800000
 num: 1357, gas: 40.300000

 (pop) num: 2468, gas: 33.800000
 --- head1 (4) ---
 num: 0, gas: 0.000000
 num: 1357, gas: 40.300000
```

【課題 14-3】 （まだ余裕のある人は…）課題 14-2 で作成した pop の処理は、「スタックが空の状態で、さらに pop する」処理には対応していません。例えば、次のような実行結果となります。

```
[main での処理（課題 14-2 の続きに書いた場合）]
 car1 = pop_car(&head1);                                     /* pop する */
 printf("\n(pop) num: %d, gas: %lf\n", car1.num, car1.gas);  /* 取り出したデータを出力 */
 show_carlist(&head1, "head1 (5)");                          /* 空のスタックが出力される */
 car1 = pop_car(&head1);                                     /* 空のスタックから pop する */
 printf("\n(pop) num: %d, gas: %lf\n", car1.num, car1.gas);
 show_carlist(&head1, "head1 (6)");

[実行結果]
 (pop) num: 1357, gas: 40.300000
 --- head1 (5) ---
 num: 0, gas: 0.000000
 Segmentation fault: 11          （←空のスタックから pop したため、エラーとなり終了する）
```

このエラーを回避できるように、関数 pop_car() に「スタックが空だった場合の処理」を追加してください。次のような処理を追加します。

- スタックが空の状態かどうか調べる。（p->next が NULL の時が空の状態）
- 空の場合は、戻り値のメンバ num と gas を-1 として返す。

main() で動作を確認してください。

```
[main での処理]
 （上述の main と同じ）
[実行結果]
 (pop) num: 1357, gas: 40.300000
 --- head1 (5) ---
 num: 0, gas: 0.000000

 (pop) num: -1, gas: -1.000000          (←空のスタックから pop して戻ってきたデータ)
 --- head1 (6) ---
 num: 0, gas: 0.000000                  (←スタックは空のまま)
```

【課題 14-4】 （さらに余裕のある人は…）スタックに対して、「1. push」「2. pop」「3. スタックを出力」「4. プログラムを終了」の 4 つの操作を選択し実行する処理を繰り返す main() を作成して下さい。main() で作る処理は次のようになります。

1. 空のスタックを準備する（課題 14-1 の main 参照）
2. 以下の処理を繰り返す
   (a) 操作に対応した整数を入力する
   (b) 入力された値が 1 の場合、ナンバーとガソリンの量を入力し、そのデータをスタックに push する
   (c) 入力された値が 2 の場合、スタックから pop し、取り出したデータを出力する
   (d) 入力された値が 3 の場合、関数 show_carlist() を呼び出しスタックを出力する
   (e) 入力された値が 4 の場合、繰り返し処理を抜ける

```
[実行結果]
 $ ./a.out
 1: push  2: pop  3: show  4: quit
 cmd > 1                          (← 1 を入力して push する)
 number > 1111                    (←ナンバーを入力する)
 gas > 20                         (←ガソリンの量を入力する)
 cmd > 1                          (←もう一つ push する)
 number > 2222
 gas > 23.5
 cmd > 3                          (← 3 を入力して push 後のスタックを出力する)
 --- head ---
 num: 0, gas: 0.000000
 num: 2222, gas: 23.500000
 num: 1111, gas: 20.000000
 cmd > 2                          (← 2 を入力して pop する)
 (pop) num: 2222, gas: 23.500000
 cmd > 1                          (← 1 を入力して push する)
 number > 3333
 gas > 25.5
 cmd > 3                          (← 3 を入力して出力する)
 --- head ---
 num: 0, gas: 0.000000
 num: 3333, gas: 25.500000
 num: 1111, gas: 20.000000
 cmd > 4                          (← 4 を入力して終了する)
```

定期試験の実施について

試験中に使用できるもの

- 筆記用具（メモ用紙は必要な人に配布）
- 演習室のコンピュータ一台（一つの机に一人の配置で、座る場所はどこでもよい）

試験中に参照できるもの

- 自分のホームディレクトリ（ホームフォルダ）以下に保存されているファイル

* **上記以外の情報を参照することは不正行為とする**
（例：USB で接続された機器に保存されているファイルの参照など）

* 授業で配布した資料の PDF ファイルは、**事前にダウンロードしておくこと**

「端末」での作業に関する補足

gedit を開く場合

「端末」から gedit を開く場合、以下のように最後に「&」を付けると、gedit 起動後も端末で引き続きコマンドが入力できるようになります。コンパイル・実行をするために、毎回 gedit を終了する手間を省けます。

```
$ gedit test.c &
```

コマンドの履歴機能

「端末」で以前入力したコマンドは履歴に残っています。
この機能を活用して、端末にコマンドを入力する手間をできるだけ少なくしましょう。

- カーソルキーの「↑」で**前に入力したコマンドをさかのぼれる**
- カーソルキーの「↓」でその逆にたどれる