

アルゴリズムとデータ構造2

2019.5.30

課題 STEP3.2A (③④を意識して読もう)

「構造体で(全てを)実装しなさい」「(全部)作りなさい」は、重たすぎるので、

- 教科書 p.32 演習問題 2.3 の解答にあたる STEP3.2Aを読もう
- リストの基本操作を追加を考えてみよう (操作名は、言語によって、若干異なる)
 - リストの先頭に、要素を一つ追加する操作 `cons` (コンス) `push`とも
 - リストの最後に、要素を一つ追加する操作 `append` (アペンド)

課題:

1. STEP3.2 A のプログラムを使用して、`insert`, `delete`操作により、オリジナルなリストを作成する実行例を示し、自分なりの説明してください。
(実行例は、`main`関数を変えて、自分の例を作ってください) ここは、③と④を
2. `delete`関数の定義に自分の言葉でコメントをつけなさい。
(`create`関数や`insert`関数を参考にしてください) ポインタ操作を読み取ること
3. `cons`関数と、`append`関数を作成せよ。(STEP 3.2 B)

3は、オプションとします。

ウォーミングアップ

1. リスト

- リストとは (pp.25 – 27 上)
 - データ構造 (ポインターでセルを連結)
 - 順序関係のあるデータ
 - 追加、挿入、削除など、変化のあるデータ
 - 基本操作 (access, insert, delete) (再帰的に定義)

2. リストの実装法 — 構造体の動的確保

3. リストのまとめ

リスト 構造体で実装 4つの表現を確認しよう

①リスト 略記 $[a_1, a_2, \dots, a_n]$

②リスト構造 (セル)

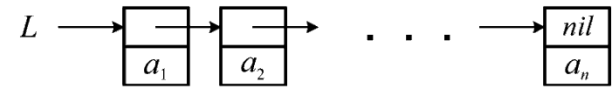


図 2.14: 連結リスト

②までは、配列の場合と同じ

②のセルを、そのままCの構造体へ対応づけて

③ は、メモリ上のセル構造体 (メモリーのイメージ図)

④ Cによる実装コード (ヘッダー ダミーセルの使用に注意)

STEP3.2 (教科書 p.32 演習 2.3 解答 pp.164-166)

構造体定義、確保 (アドレス、メモリ操作)

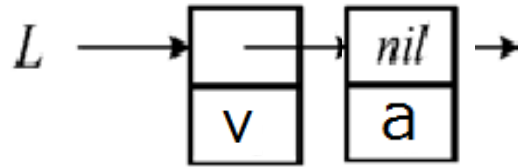
insert, deleteは、再帰関数

③は、メモリー図

①

[v, a]

②

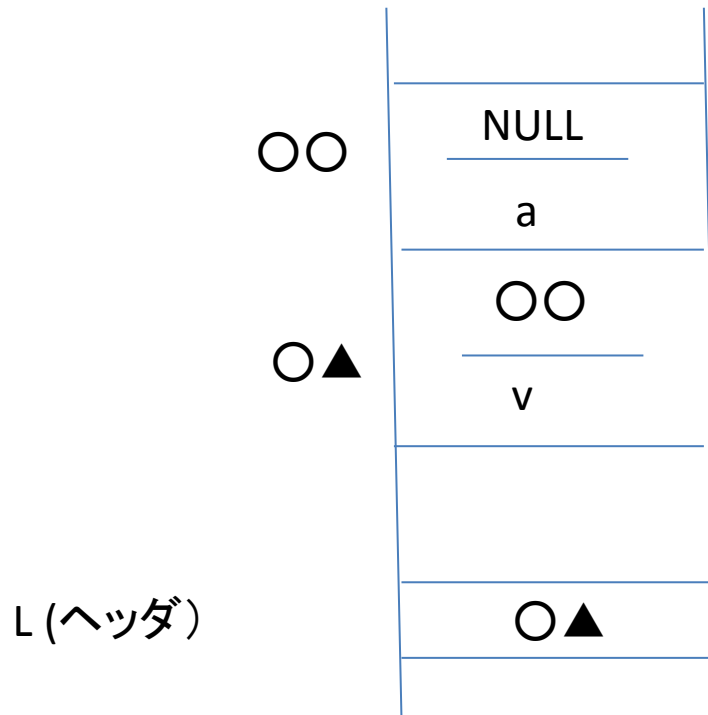


③ はどうなるだろう？ メモリ上のイメージ図を書こう

- (A) どんなデータを
- (B) どのように

- Lを変数、リストの各要素をセル構造体としてメモリー図を（書いてみよう、イメージしよう）

①[v,a]の ③メモリー図(例) (④の実装とは異なるので、注意)



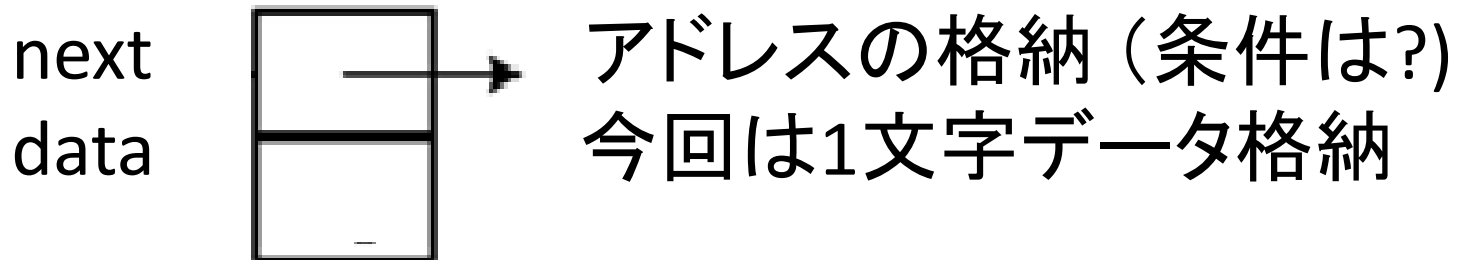
- ・工学実験のリストは、この図のまま実装
- ・教科書の実装では、ヘッダもセルで表現
Lは、ヘッダのアドレスを格納
この後の説明で確認

接続関係が正しければ、どんな配置でもよい
OO、O▲は、アドレスのつもり

③を作っていると④を読む まずは、List 構造体

```
typedef struct list {  
    struct list *next;    /* nextは(メモリの)アドレス */  
    char data;            /* dataは、char */  
} List;
```

- nextはアドレス、実際にアクセスすると、struct listが格納されているようなアドレス
- 自分と同じ構造体、つまり、次のセルを指すポインタ



このような構造体のことを、自己参照構造体
(データ構造が再帰的。いよいよ再帰の上級編)

② insert, delete 操作確認

④ではC言語のメモリ、ポインタ操作

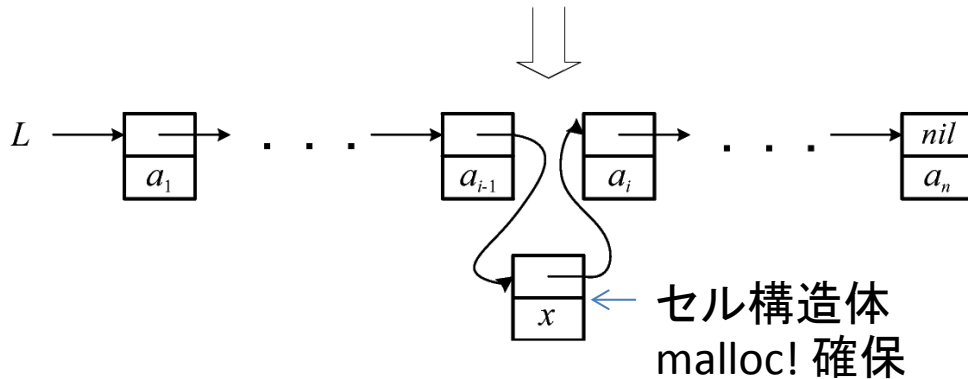
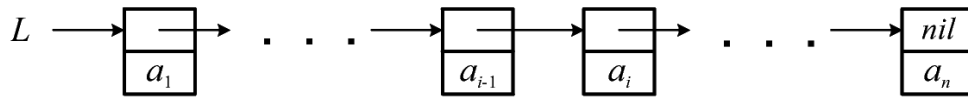


図 2.15: リストへの要素の挿入

- アクセス (access)
 - i 番目のデータの値取り出し
 - 先頭から辿る

- 挿入 (insert) `insert(p, i, data)`
 - i 番目に、insert
 - 新規セルを準備して、ポインタの付け替え

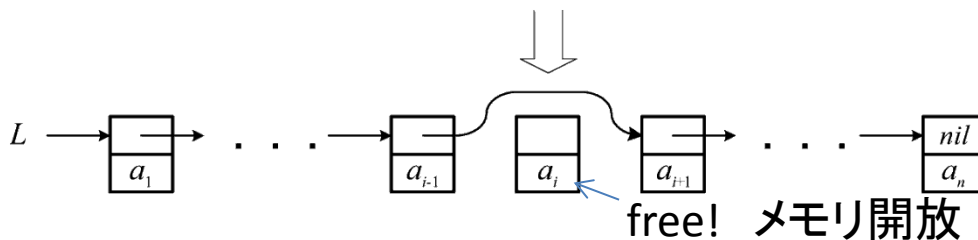
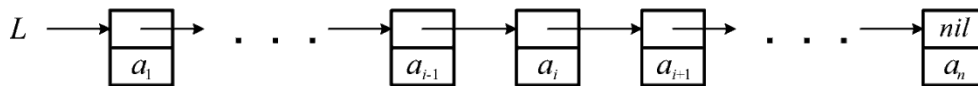


図 2.16: リストからの要素の削除

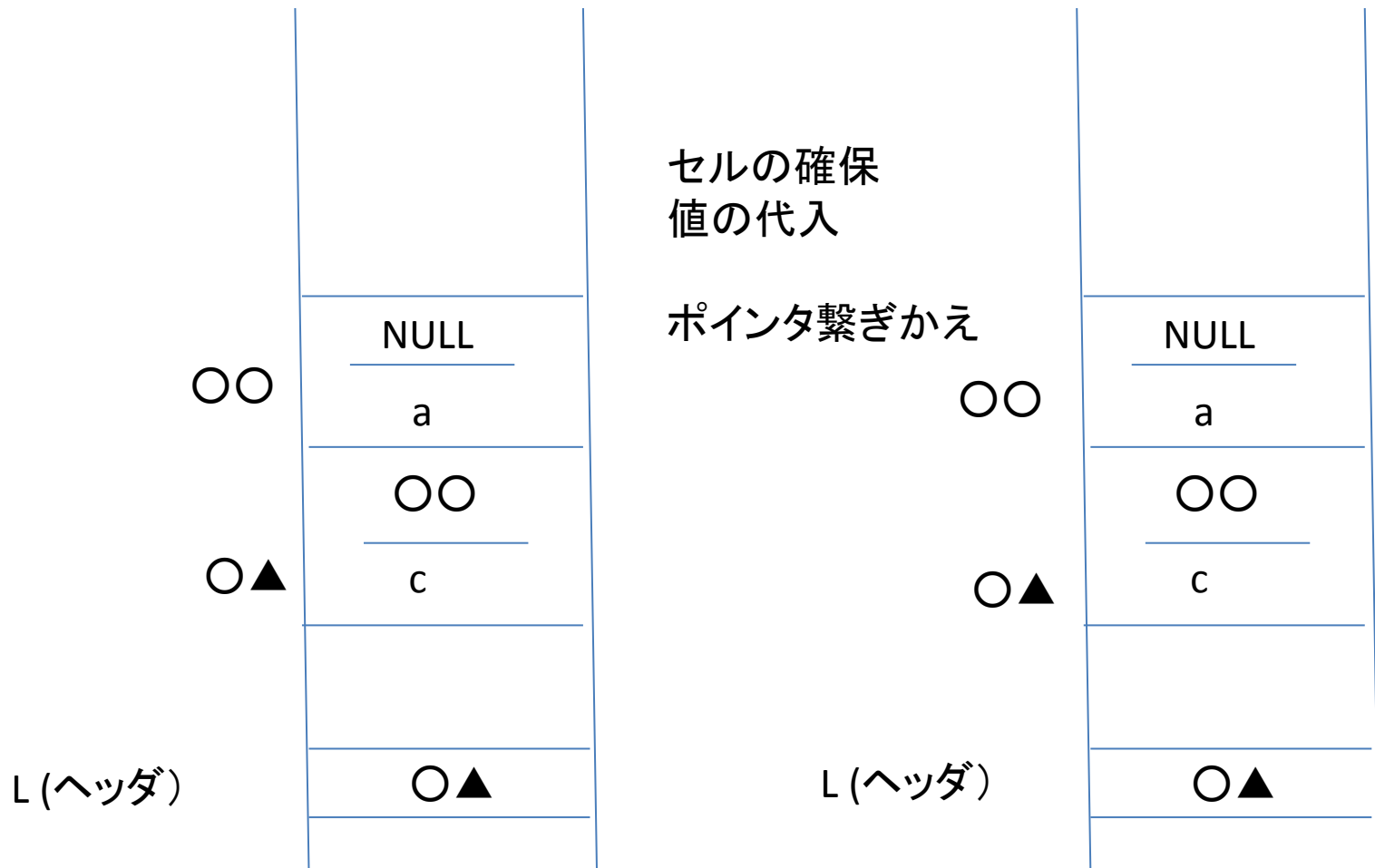
- 削除 (delete) `delete (p, i)`
 - i 番目をdelete
 - 削除 (Free)
 - ポインタの付け替え

③のメモリー図をイメージしよう

[c,a]のあとに、insert(L, 2, v)

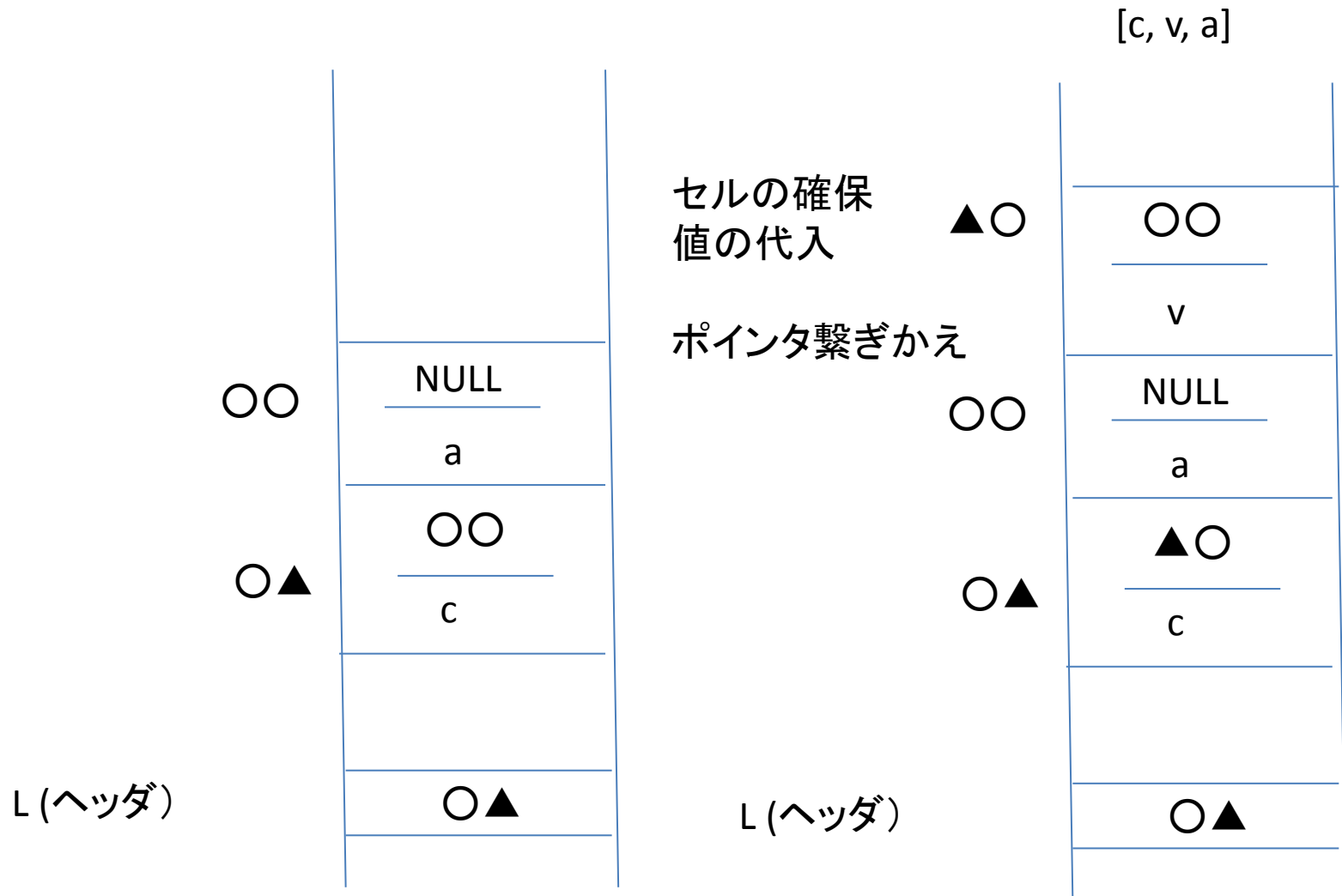
③メモリ表現は、どうなるだろう？

[c, v, a] になるはず



接続関係が正しければ、どんな配置でもよい
○○、○▲は、アドレスのつもり

[c,a] insert(L, 2, v) の③(解答例)



3.2 のメイン関数から④を

```
int main(void){           ①      ②(セル)   ③ (メモリ図)   ④ C言語コード (関数、変数の値)
    List *L;
    L = create();         []
    insert (L,1,'a');
    insert (L,1,'c');
    insert (L,2,'v');
    printlist(L);         [c,v,a]

    delete (L,1);
    insert(L,2,'a');
    delete(L,3);

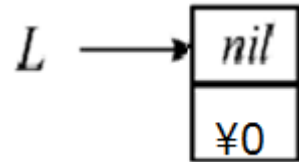
    printlist(L);         [v,a]

    FreeData(L);
    return(0);
}
```

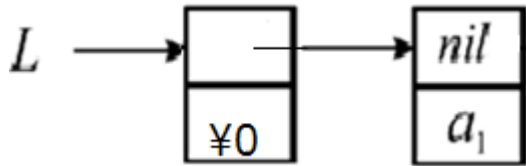
Main 関数 最初はこんなはず

② データリストの実現

1. まず、最初にヘッダを準備
(ヘッダーのnext nil 空リスト)



2. `insert(next[0],1,'a');` (ヘッダは先頭セルの
アドレス格納。
リストの最後は、nil)



- ③ 1, 2のメモリの図を描こう。
- ④ Cのプログラムとの対応をつけよう

④ main関数から、おいかけよう

main関数の始まりから

1. ダミーセル(ヘッダ)を用意するまで。ここでは空リスト。
2. 要素を1つinsertするまで。リスト[a]となるはず。

```
int main(void) {
```

```
List *L;          /* Lはアドレス 条件は? */
```

```
L = create();     /* セルに接続 ヘッダの作成 [] */
```

(1. ③ 何をしているのか。どうなるのか。メモリーの図)

④ create関数は、どのようにプログラムされているか)

```
insert (L,1,'a'); /* 1番目にinsert [a]ということ*/
```

(2. ③何をしているのか。どうなるのか。

④insert関数は、どのようにプログラムされているか)

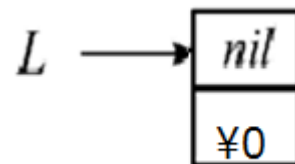
1. ヘッダの用意 []

```
int main(void) {
```

```
    List *L;          /* Lはどんなアドレス? */
```

```
    L = create();      /* セルに接続 空 [] */  
                      /* ヘッダの作成 */
```

(セルの絵を)



(メモリの絵を)

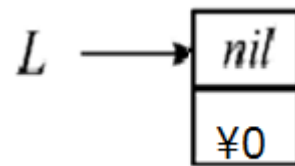
梯子型のイメージ図

1. main 関数 ヘッダの用意 まで ③のメモリ図を描いてみよう

```
int main(void) {
```

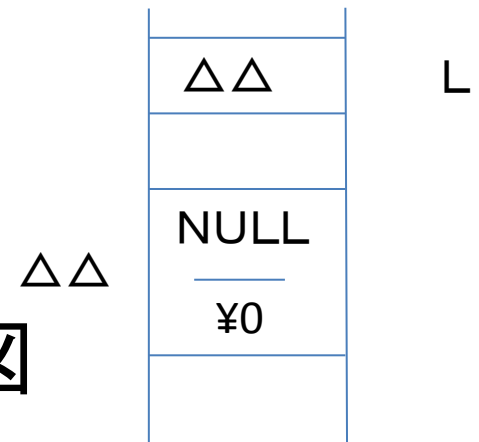
```
List *L;          /* Lはどんなアドレス? */  
L = create();      /* セルに接続 */  
/* ヘッダの作成 空リスト */
```

(② セルの絵)



(③ メモリの絵を)

梯子型のイメージ図



④ の create 関数 を読む

- create関数は何をするはずか。③ メモリ図にしよう (トップダウン)
- どうやっているのか、④ プログラムを読もう (ボトムアップ)
- List構造体を確保のはず。malloc 関数の呼び出しに注目！

```
List *create (void) { /* アドレスを返す */  
    List *p;          /* Listを指すポインタ変数 */  
    p = (List *) malloc(sizeof(List));  
    /* メモリにセルを1つ確保して、pにつなぐ。図にしよう*/  
    p->next = NULL;  
    /* pから構造体のメンバーにアクセスし  
       初期化    p->nextという表現に注意 */  
    p->data = '¥0';    /* data はchar。ヌル文字で初期化 */  
    return (p);  
}
```

アドレス、ポインタ、構造体、メモリ確保が勢揃い
下線部を、説明しきること！

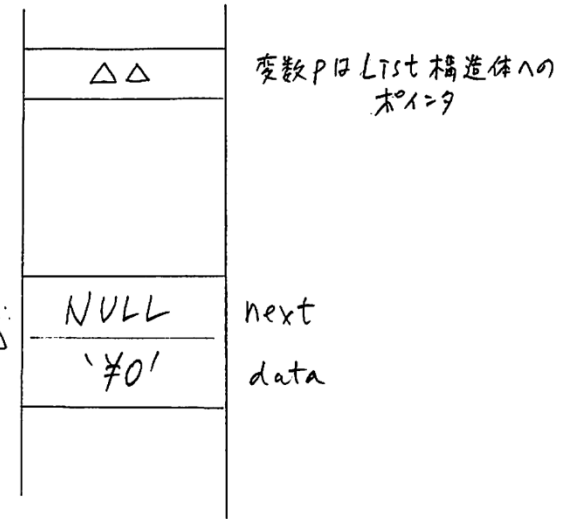
Create 関数

① `List *p` (ここで、`p`の値は不定)

② `p = (List *) malloc (sizeof(List));`

- `List` サイズのメモリーを確保(`alloc`)
- `malloc`の戻り値は、アドレス
- どんなアドレスなのか 型を宣言
(`List *`) (キャストと呼ぶ) $\Delta\Delta$
- `p`に代入

`p`は、`List` 構造体へのポインタ となる



③ `p->next = NULL`

- `(*p).next = NULL` と同じ

④ `p->data = '¥0'`

- `(*p).data = '¥0'` と同じ

`data`は `char` `'¥0'`は 文字

⑤ `return p;` アドレスを戻り値に
($\Delta\Delta$)

どんな アドレスか

`List *create(void)`

つまり `List` 構造体へのポインタ

ドライブにInsert関数の例
delete関数も、読もう。
メモリ図にしてみよう

次は、insert 関数 [a]となる操作まで表現してみよう

```
int main(void) {  
    List *L;  
    L = create();    /* [] */  
    insert(L, 1, 'a') /* 1番目にinsert [a] ということ*/  
}
```

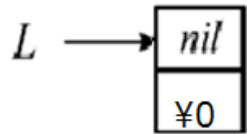
- ② セル（④の実装にあわせよう。ヘッダあり）
- ③ メモリ変化のイメージ図（④の実装に）
- ④ insert関数は、どのようにプログラムされているか

Insert(L, 1, 'a') ②③ (ヘッダ付)

①

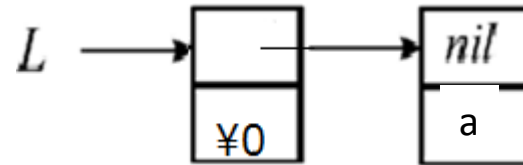
[]

②



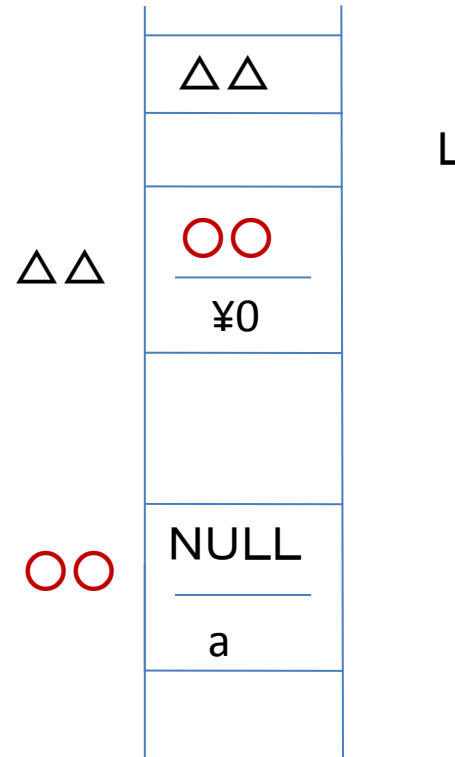
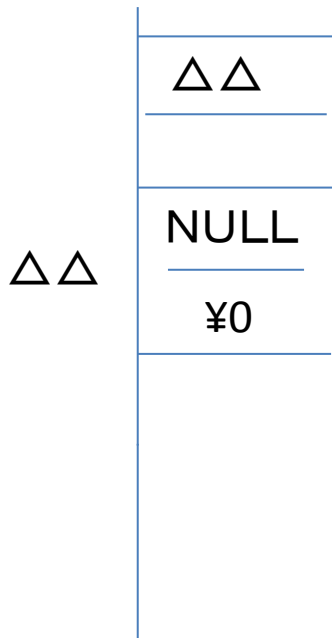
ヘッダ
nextがnilなの
で、空リスト

[a]



最初のセル
はヘッダ

③



Insert(L, 1, 'a') 読もう

- 構造体の操作処理が、配列処理と、異なるだけ。大まかには同じもの

```
void insert(List *L, int i, char x) { /* ヘッダLから辿り、i番目に 文字 x */
    List *p; /* 新セルのアドレス p */
    if (L != NULL) {
        if (i > 1) { /* 再帰条件 */
            insert(L->next,i-1,x); /* 1つ先へ進め */
        } else { /* 停止条件 i== 1 のとき 次のセルの前に、挿入 */
            p = create();          /* 新しいセルを1つ。確保したアドレスをp */
            p->data = x;
            p->next = L->next; /* 後ろのポインターをnextに */
            L->next = p;       /* 前のポインターに、自分を */
        }
    }
    else {
        printf("List L ends before arriving ");
        printf("at the position.¥n");
    }
}
```

こちらのポイントは、「再帰」（関数もデータも）

Lはアドレス List構造体のポインタ
i番目に x を insert

void insert(List *L, int i, char x) {

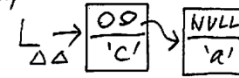
List *p

if (L != NULL)

if (i > 1) { /* 再帰条件・次の要素を先頭として
i-1番目に x を insert */

insert(L->next, i-1, x)

insert(ΔΔ, 1, 'v')



} else { /* i==1 停止条件 */

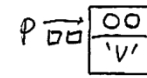
p = create();

p->data = x;

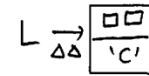
p->next = L->next;

L->next = p;

p->next = L->next
dataに v を代入



p->next = L->next
00



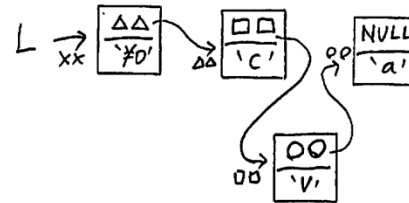
L->next = p
00

} else {

1 になる前に NULL. リストの長さが足りなり

}

}



メモリ関係 2つの関数を確認 (まずは、何をするか、読めるように)

- 確保する malloc 関数 (memory allocation)
 - 「サイズ」のデータをメモリに確保
 - 確保したアドレスを、戻り値に
 - 戻り値のアドレスに条件づけを(キャスト)

```
List *p;
```

```
p = (List *) malloc (sizeof(List)); /* p に確保したアドレス */
```

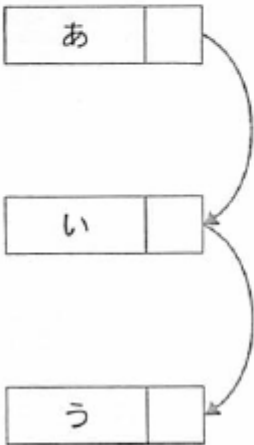
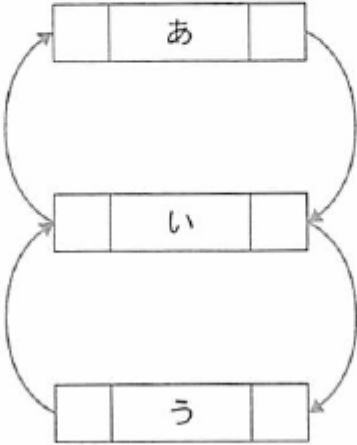
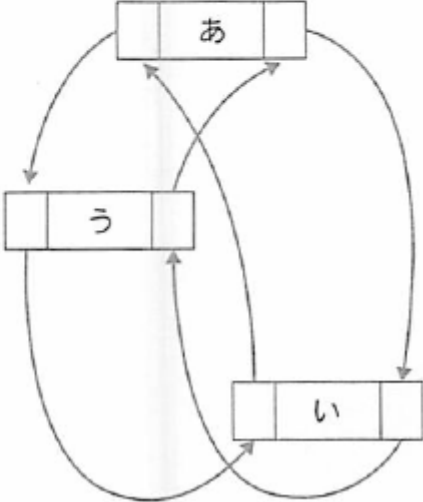
- 削除する、不要になったとき free
 - データを(OSに)開放。再利用される。

オプション consとappend

- 言葉で表現しよう
- cons : 先頭にセルを追加して、つなぐ
 - Insert関数に読み替えできる
- append : リストに要素をappend :
 - 停止条件: 次が空なら、セルを追加
 - 再帰条件: 次が空ではないなら
 - (リストが続くなら)
 - 次の要素を先頭とするリストにappend

関数もデータも再帰的 (授業のページを確認してください)

リストのいろいろ(工夫)

単方向リスト	双方向リスト (対称リスト)	環状リスト (循環リスト)
次のデータへのポインタだけをもっている	前のデータと次のデータへのポインタをもっている	前のデータと次のデータへのポインタをもち、最後のデータと先頭のデータが連結され、環状になっている
		

- リストの最後に操作が多いなら、リストの先頭だけでなく、リストの最後をポインターで管理するほうがよい。
- 末尾まで辿る計算量は、 $O(n)$

ここで、一息

図は単方向リストを表している。“東京”がリストの先頭であり、そのポインタには次のデータのアドレスが入っている。また、“名古屋”はリストの最後であり、そのポインタには0が入っている。

アドレス 150 に置かれた“静岡”を、“熱海”と“浜松”の間に挿入する処理として正しいものはどれか。

先頭へのポインタ	アドレス	データ	ポインタ
10	10	東京	50
	30	名古屋	0
	50	新横浜	90
	70	浜松	30
	90	熱海	70
	150	静岡	

- ア 静岡のポインタを 50 とし、浜松のポインタを 150 とする。
- イ 静岡のポインタを 70 とし、熱海のポインタを 150 とする。
- ウ 静岡のポインタを 90 とし、浜松のポインタを 150 とする。
- エ 静岡のポインタを 150 とし、熱海のポインタを 90 とする。

- リストの図を描いてみよう。STEP3.3は構造体による実装

配列と比較した場合の連結リストの特徴に関する記述として、適切なものはどれか。(FE-H21 春-前 6)

- ア 要素を更新する場合、ポインタを順番にたどるだけなので、処理時間は短い。
- イ 要素を削除する場合、削除した要素から後ろにあるすべての要素を前に移動するので、処理時間は長い。
- ウ 要素を参照する場合、ランダムにアクセスできるので、処理時間は短い。
- エ 要素を挿入する場合、数個のポインタを書き換えるだけなので、処理時間は短い。

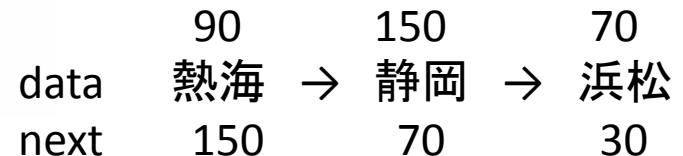
リストの問題

図は単方向リストを表している。“東京”がリストの先頭であり、そのポインタには次のデータのアドレスが入っている。また、“名古屋”はリストの最後であり、そのポインタには0が入っている。

アドレス 150 に置かれた“静岡”を、“熱海”と“浜松”の間に挿入する処理として正しいものはどれか。

先頭へのポインタ	アドレス	データ	ポインタ
10	10	東京	50
	30	名古屋	0
	50	新横浜	90
	70	浜松	30
	90	熱海	70
	150	静岡	

- ア 静岡のポインタを 50 とし、浜松のポインタを 150 とする。
- イ 静岡のポインタを 70 とし、熱海のポインタを 150 とする。
- ウ 静岡のポインタを 90 とし、浜松のポインタを 150 とする。
- エ 静岡のポインタを 150 とし、熱海のポインタを 90 とする。



配列と比較した場合の連結リストの特徴に関する記述として、適切なものはどれか。(FE-H21 春-前 6)

- ア 要素を更新する場合、ポインタを順番にたどるだけなので、処理時間は短い。
- イ 要素を削除する場合、削除した要素から後ろにあるすべての要素を前に移動するので、処理時間は長い。
- ウ 要素を参照する場合、ランダムにアクセスできるので、処理時間は短い。
- エ 要素を挿入する場合、数個のポインタを書き換えるだけなので、処理時間は短い。

今日の話題

① データ構造表現 基本操作 ② 実装

1. スタック (stack) (pp.17 – 20)
2. キュー (queue) (pp.21 – 24)
3. 逆ポーランド記法
(コンパイラ論 構文解析の基本練習問題)

資格試験 演習問題

4. ② 配列による実装
(配列をスタック、キューとして見る)
5. ② リストによる実装を用いて (演習問題 p.32)
(リストをスタック、キューとして見る)

1. スタック (stack) (pp.17-18)

データ構造 と 基本操作

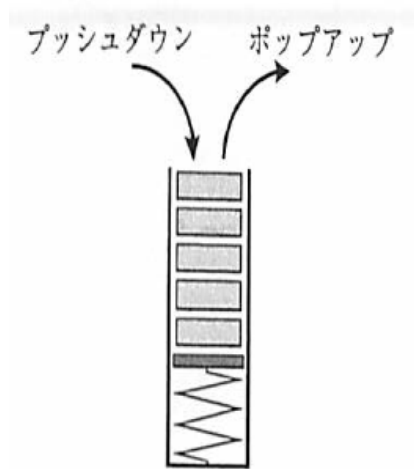


図 2.1: スタック

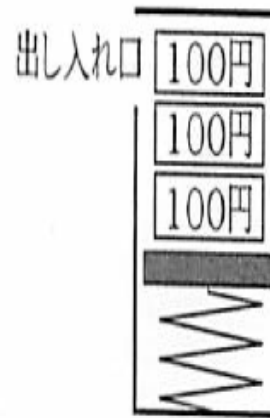


図 2.2: スタックの身近な例 (コイン入れ)

- pushdown (push とだけいうことが多い) ; データを積む
- popup (pop とだけいうことが多い) ; データを取り出す
- 後に記憶されたデータが先に取り出される
後入れ先出し (LIFO: Last-In First-Out)

2. キュー (queue) 待ち行列 (p.21) データ構造 と 基本操作

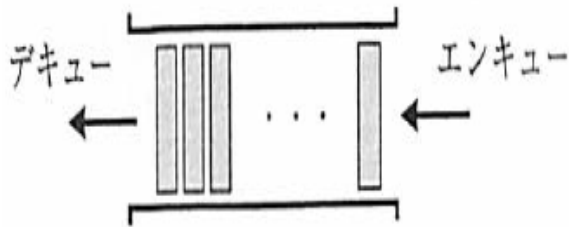


図 2.6: キュー

- enqueue データを入れる (エンキュー)
- dequeue データを取り出す (デキュー)
- 先に記憶されたデータが先に取り出される
先入れ先出し (FIFO: First-In First-Out)

① スタック、キュー (代表的な図)

名称	スタック (後入先出リスト)	キュー (先入先出リスト)
特性	LIFO (Last In, First Out)	FIFO (First In, First Out)
データの格納	PUSH	enqueue
データの取出し	POP	dequeue
配列による実現		

福嶋 “真図解 基本情報技術者”, pp.52-55, 日本経済新聞出版社, 2011

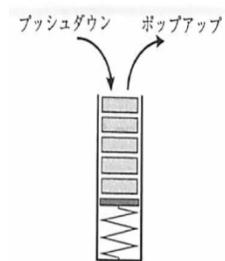


図 2.1: スタック

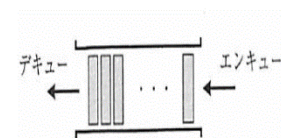


図 2.6: キュー

① 問2.1 (p.20) と 問2.6(p.24) (図にしてみよう)

【問 2.1】 スタック S に次のような一連の操作を実行したとき、スタック S の内容がどのようなになるか示しなさい.

pushdown(S, a); pushdown(S, b); pushdown(S, c); popup(S);
popup(S); pushdown(S, d); popup(S); pushdown(S, e);

【問 2.6】 次のような一連の操作を実行した後のキュー Q の内容を示しなさい.

enqueue(Q, a); enqueue(Q, b); enqueue(Q, c); dequeue(Q);
dequeue(Q); enqueue(Q, d); dequeue(Q); enqueue(Q, e);

スタックの代表的な利用例

逆ポーランド記法による演算

演算子を後置する記述法

四則演算 (+, -, *, /) が例題になることが多い

3 4 + これは、3+4

3 4 - これは、3-4

1 5 + 2 3 + * これは、6*5

- 値ならpush
- 演算子なら、2つの値をpop、計算結果をpush

② スタックの実装 (配列版)

配列をスタックとして見る

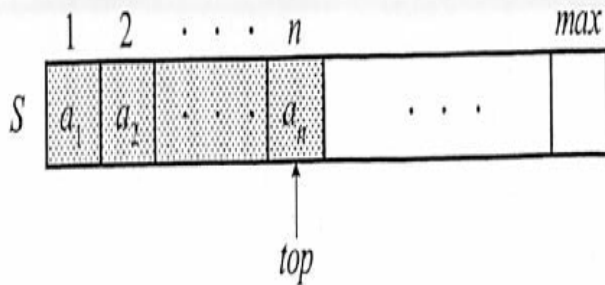


図 2.3: 配列によるスタックの実現

変数Topは、次の要素の添え字と格納個数の情報を示している

```
pushdown(S, x){
    if (top < max){
        top ← top + 1;
        S[top] ← x;
    }
    else{
        print ("Stack S overflows.");
    }
}
```

図 2.4: pushdown(S, x)

```
popup(S){
    if (top > 0){
        top ← top - 1;
        return(S[top + 1]);
    }
    else{
        print ("Stack S is empty.");
    }
}
```

図 2.5: popup(S)

② キューの実装（配列版）

配列をキューとして見る

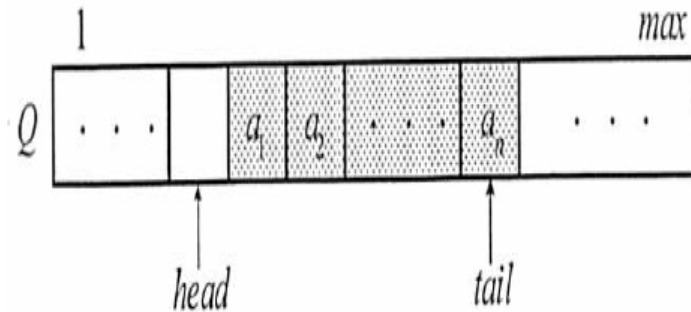


図 2.7: 配列によるキューの実現

全体が、右へ移動する

デキューすると、配列の
先頭部分がもったいない

```
enqueue(Q, x){
    tail ← tail + 1;
    Q[tail] ← x;
}
```

図 2.8: enqueue(Q, x)

```
dequeue(Q){
    head ← head + 1;
    return(Q[head]);
}
```

図 2.9: dequeue(Q)

② キュー 改良版 (mod 剰余関数)

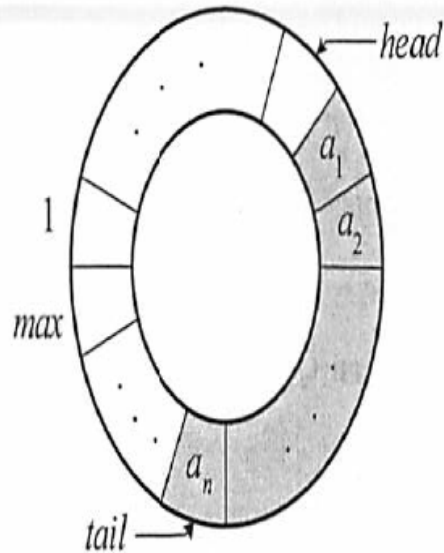


図 2.10: 環状につなげた配列によるキューの実現

number は、格納個数

```
enqueue(Q, x){  
    if (number < max){  
        number ← number + 1;  
        tail ← (tail mod max) + 1;  
        Q[tail] ← x;  
    }  
    else{  
        print ("Queue Q overflows.");  
    }  
}
```

図 2.11: 改良した enqueue(Q, x)

```
dequeue(Q){  
    if (number > 0){  
        number ← number - 1;  
        head ← (head mod max) + 1;  
        return(Q[head]);  
    }  
    else{  
        print ("Queue Q is empty.");  
    }  
}
```

図 2.12: 改良した dequeue(Q)

2章 終了 まとめ

データ構造、基本操作（絵をかけるように）

- スタック、キュー
- リスト

実装

- 配列の図、メモリの図（絵をかけるように）
- C言語（コードとの対応。飛ばさずに読めるように）
 - （メモリ） アドレス、ポインター
 - 構造体 定義、操作
 - メモリにデータ確保（malloc） sizeofやキャスト
 - メモリの開放（free）

今日の課題 (p.32 演習問題)

- 配列を用いて、スタックとキューを実装する課題は、自習
- STEP4.1 : スタック STEP4.2 : キュー に解答例を置くので、読む
- 構造体による実装解答例は、「富山大に質問したい」もの
 - オブジェクトとして実装しているとも読める。
そういうものもありか、と見ておくとい

課題 リストを「スタックやキュー」と見る

1. STEP 4.1 C : 演習問題 2.6 を解こう。オリジナルな実行例を示しなさい。
 - STEP 3.2のリスト関数を参考に、pushとpopを作成せよ
 - 「リスト関数の読み替え」。見方を変えてみよう。「あ、そうか」となるはず。
2. STEP4. 2C : 演習問題 2.7を解こう。オリジナルな実行例を示しなさい。
 - STEP 3.2のリスト関数を参考に、dequeueとenqueueを作成せよ。
 - 考えてみるとdequeueは。。面倒なのは、enqueueだが。。
 - キューは必ず末尾を操作。末尾のセルをポイントしておく方法もよい。

ついにC 頂上 : そこで一言

- 今週の課題が、「まあ何とかなるな」と見えるなら、Cの実力はOK！
- 余裕があるうちに、必ずリスト関係のプログラムの確認を
- 各関数の定義と、mainからの実行を、どこも飛ばさずに、確実に読んでおこう
- リストでは、③ メモリの(操作)図がとても重要。
④のプログラムがしていることをイメージできないと、プログラムを読めない
- ただし、
 - 皆さんに書いてもらったメモリの図(イメージ)は、実は基礎編
 - 「関数の格納」、「関数の実行とメモリ」、「変数が確保されるメモリ領域の区別」など、「Cを徹底的に使う人」は、さらに学習が必要
 - ここから先は、基礎編を足場にして、必要に応じて学習しよう。
 - 応用編に進むと、そこに「スタック」という言葉が登場する。
 - 関数の呼び出しが、「スタックで管理」されるのは、想像つくでしょう
 - 各関数が確保されている領域も、アドレスでポイントされる
 - 「共用体」の理解にも、メモリが関係している

次週

- 6章 グラフと探索に入ります
- 実装は、Cの頂上を維持
 - リスト、スタック、キューの使用
 - グラフデータの絵、メモリの絵、C言語
 - 基本操作、C言語
- (情報工学科) 工学実験と連動します。
- UNIXらしい操作の準備