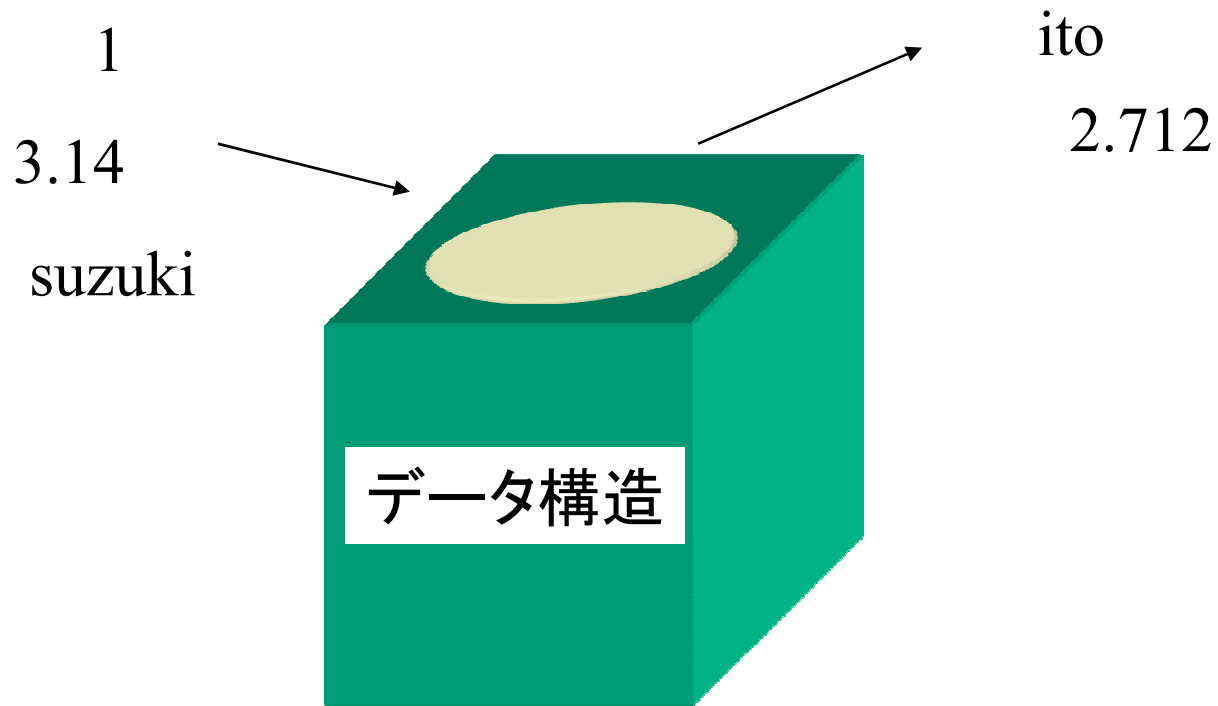


## 6. データ構造入門

- 6－1. 連結リスト(Linked List)
- 6－2. スタック(Stack)
- 6－3. キュー(Queue)
- 6－4. デク(Double-Ended-Queue)
- 6－5. 抽象データ型 (Abstract Data Type)

# データ構造とは、

データの保存を効率的に行うもの。

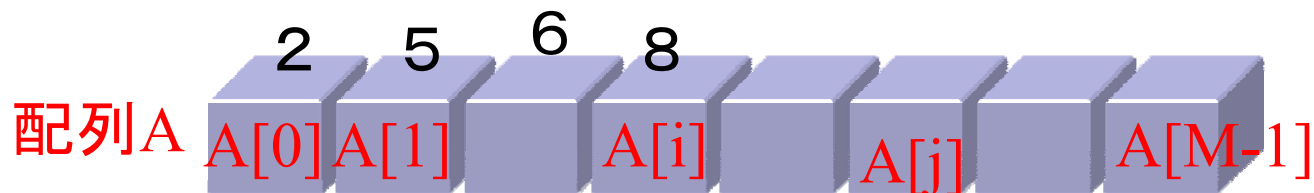


## 6-1. 連結リスト(Linked List)

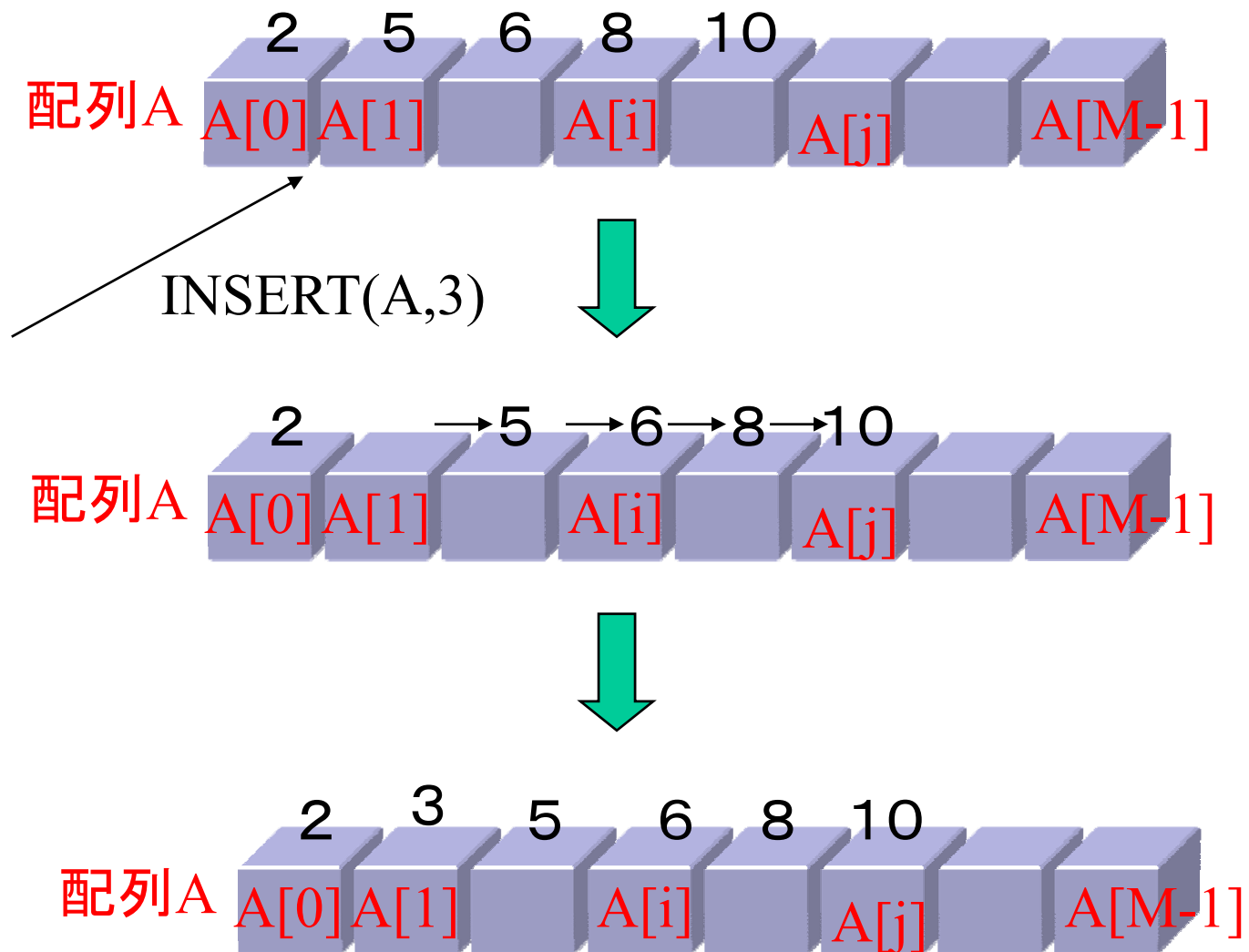
- 配列が不得意とする問題

多量のデータをあつかうためのデータ構造として、配列がある。しかし、配列では扱いにくい問題もある。

例えば、整列してあるデータに、新しいデータを挿入して、また整列の状態にする問題を考えよう。



整列されている配列へのデータ挿入。



このような問題では、配列操作に余分な手間が必要。 4

## 柔軟なデータ構造の構築にむけて

- アイディア
  - 構造体とポインタを組み合わせる。
  - 必要な分だけのメモリを確保する。(配列では、プログラムの開始時点で余分なメモリが確保されていた。)
  - 自己再帰的な定義を用いる。

# データ構造の基本単位(セル)

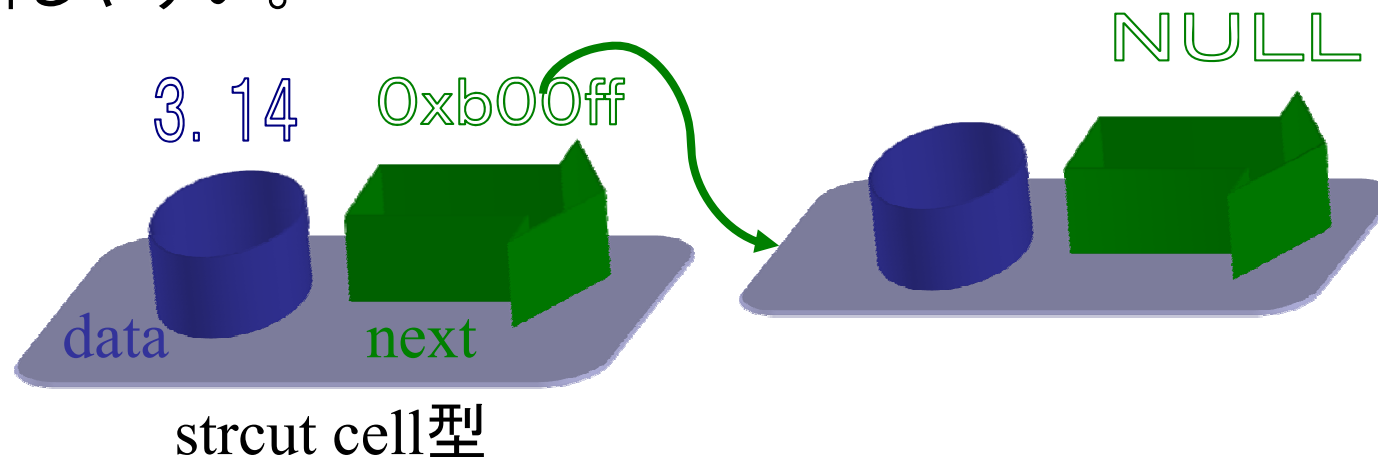
- 自己参照構造体を用いる。

```
struct cell
{
    double data;
    struct cell * next;
};
```

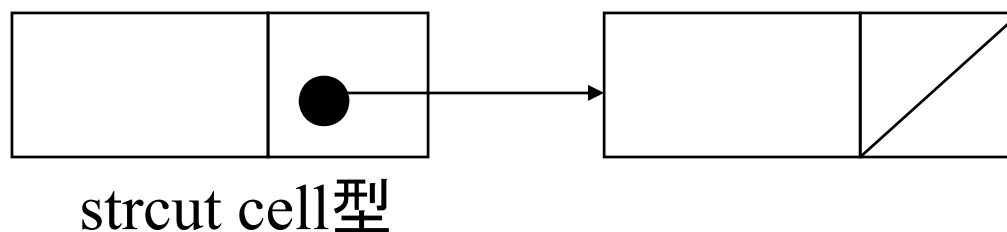
struct cellという構造体を定義するのに、  
struct cellを指すポインタをもちいている。

# イメージ

一見すると、奇異な感じをうける定義であるが、ポインタがアドレスを保持する変数ということに注意すると理解しやすい。



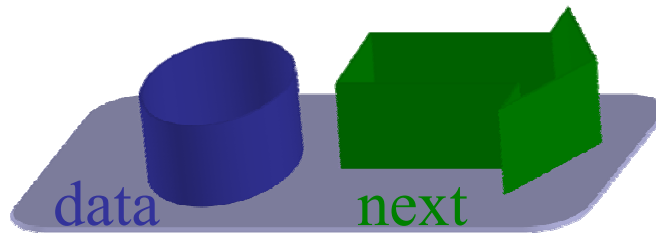
このことを、次のような図を用いて表すこともある。



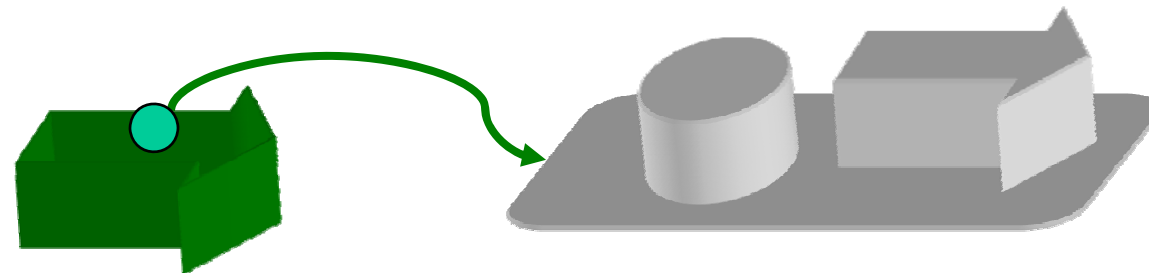
終わりを  
意味する。

# セル型の定義

```
typedef struct cell  Cell;
```



Cell型



Cell \* 型

ポインタでは、保持しているアドレスの先がデータ型であることに注意する。



# セルの動的なメモリ確保

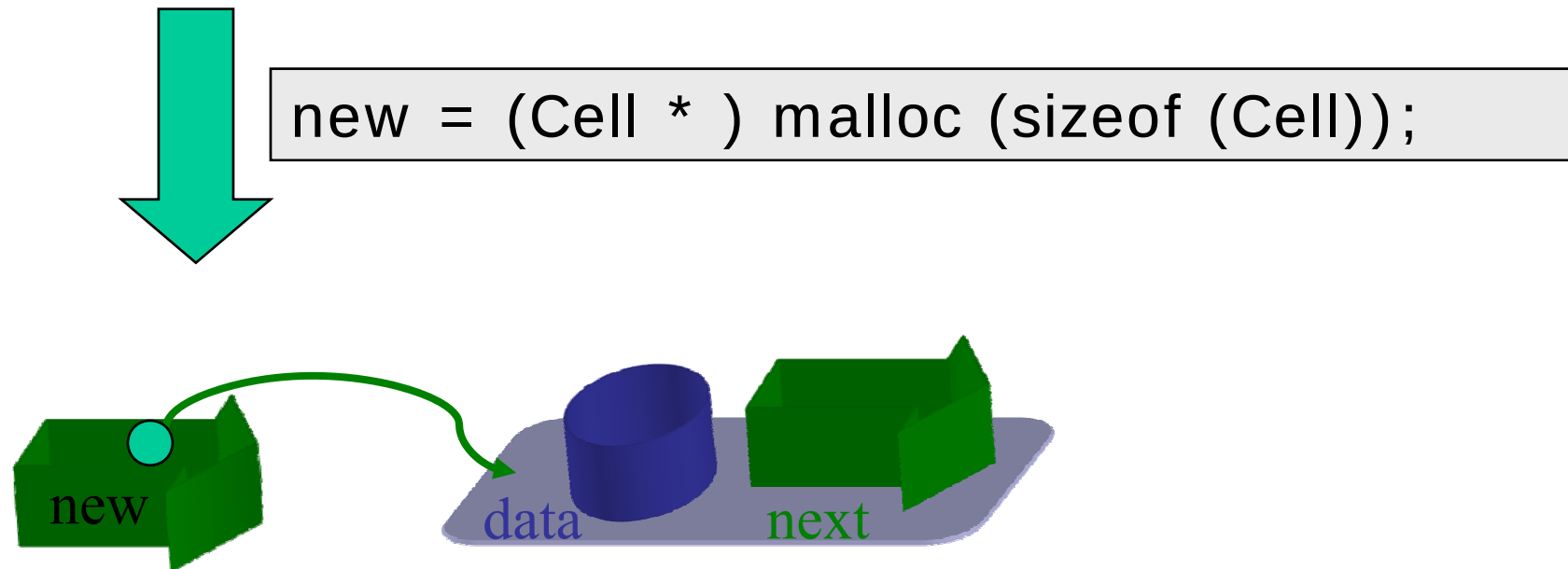
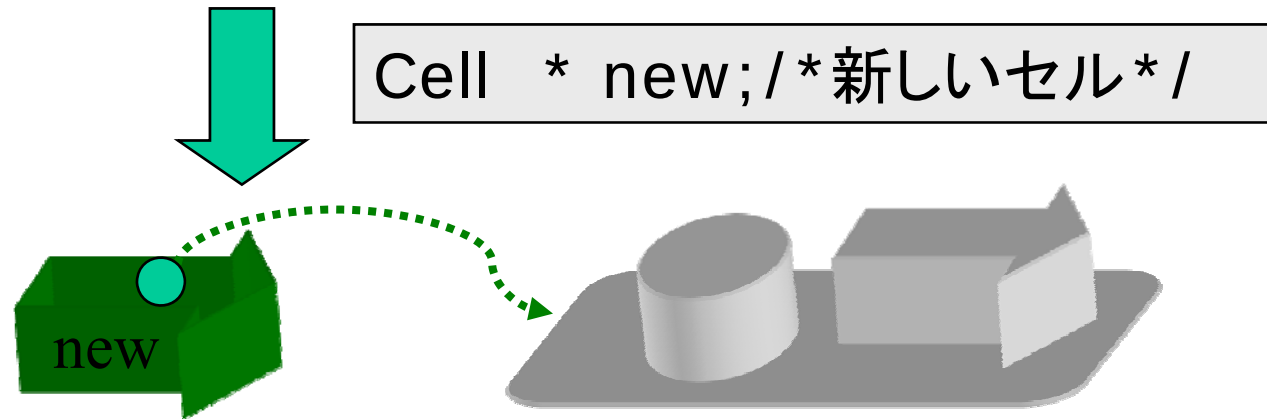
```
Cell * new; /*新しいセル*/  
new = (Cell *) malloc (sizeof (Cell));
```

キャストする。

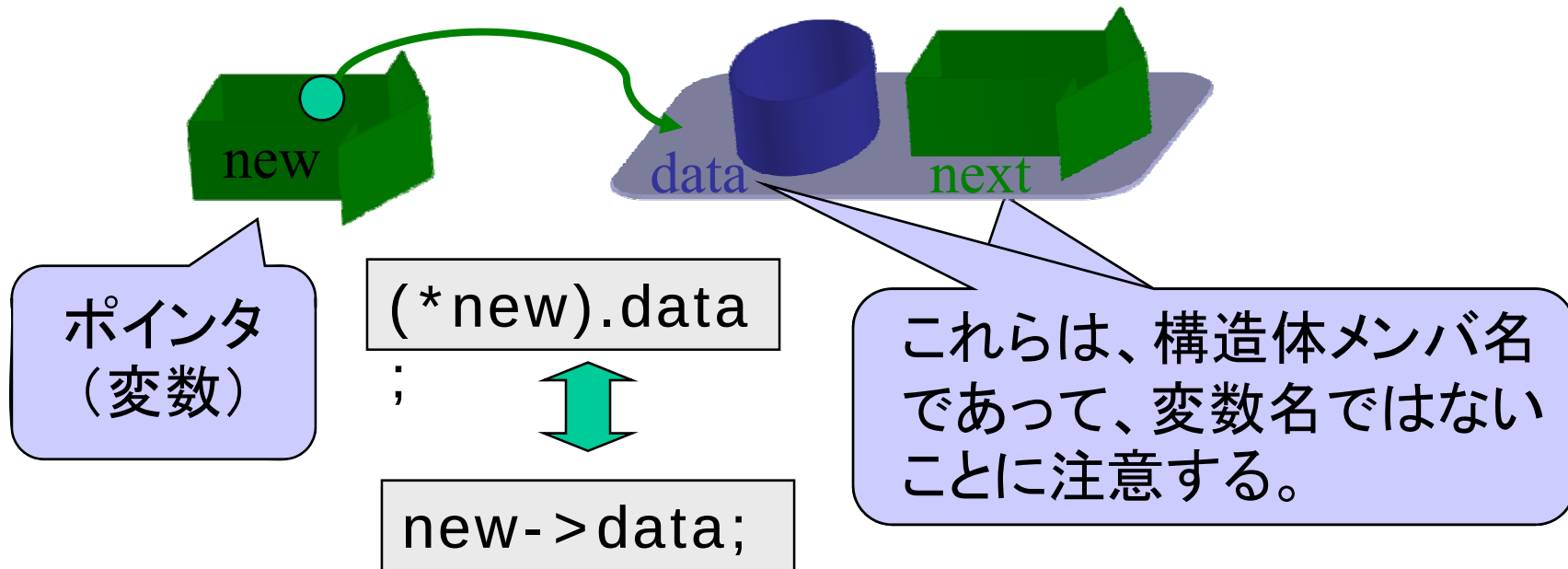
メモリを確保する。  
(void\*を返す。)

引数の型のサイズを求  
める。

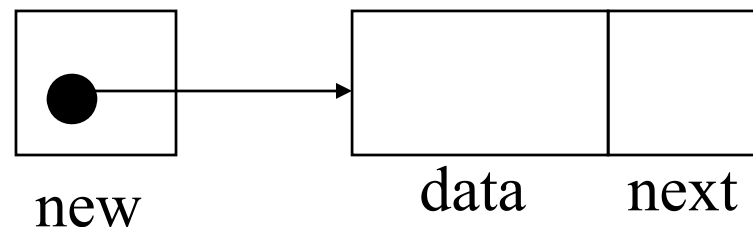
一般のポインタ  
を表す型。



# C言語における略記法

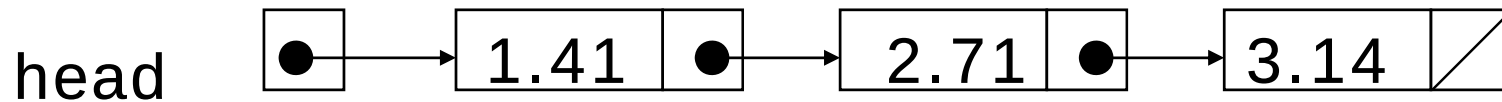
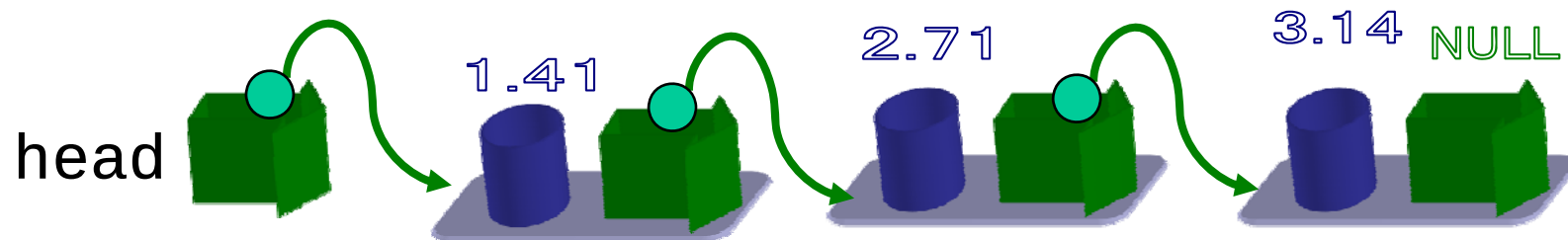


C言語では、上の2つは同じ意味で用いられる。  
下の表記法は、ポインタの図式と類似点がある。



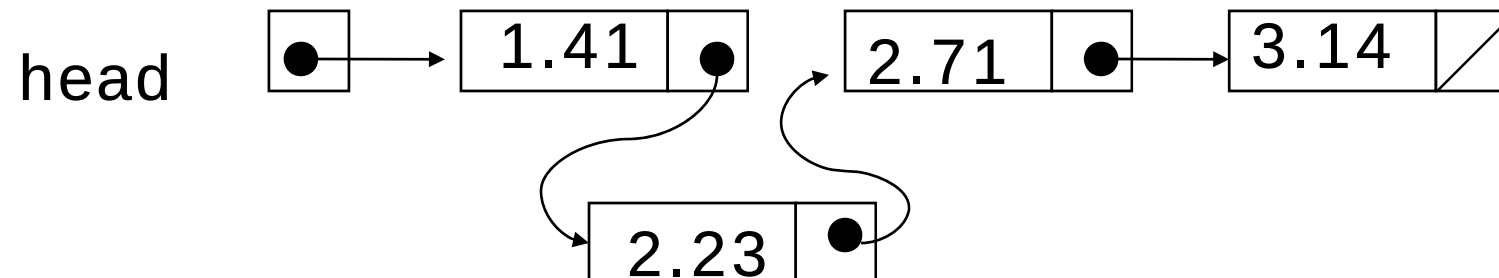
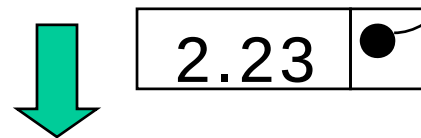
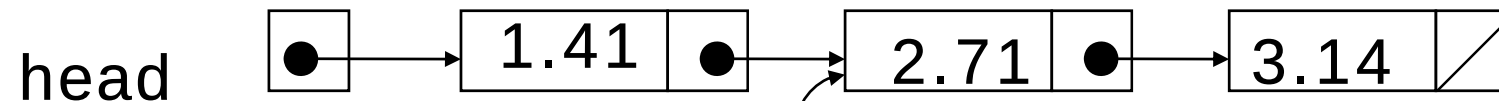
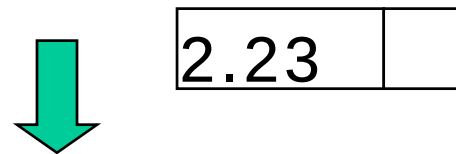
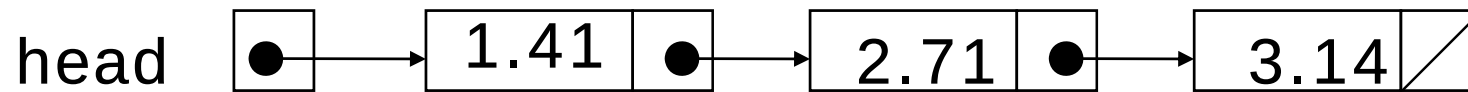
# 連結リスト

セルをポインタで一直線上にならべたもの。



連結リストの途中は、直接参照(アクセス)できない。  
(変数による“名前”がついていない。)

# 連結リストへの挿入



# 実現例

```
/*位置posの後に新しいセルを挿入*/  
void insert(Cell * pos,double x)  
{  
    Cell *new=(Cell *)malloc(sizeof(Cell));  
    new->data=x;  
    new->next=pos->next;  
    pos->next=new;  
    return;  
};
```

ポインタにはアドレスが保持してあることに注意して更新の順序を考えること。

# 連結リストへの挿入の計算量

- 定数回の代入演算を行っているだけであるので、1回あたりの時間計算量は、

$O(1)$  時間

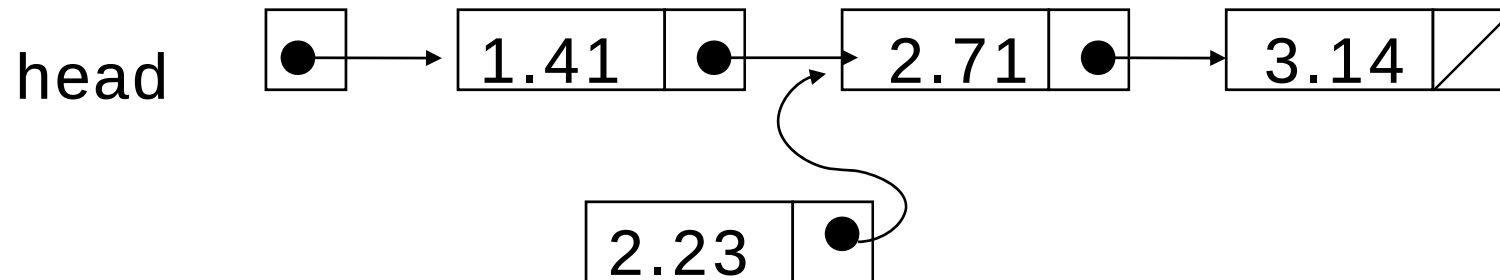
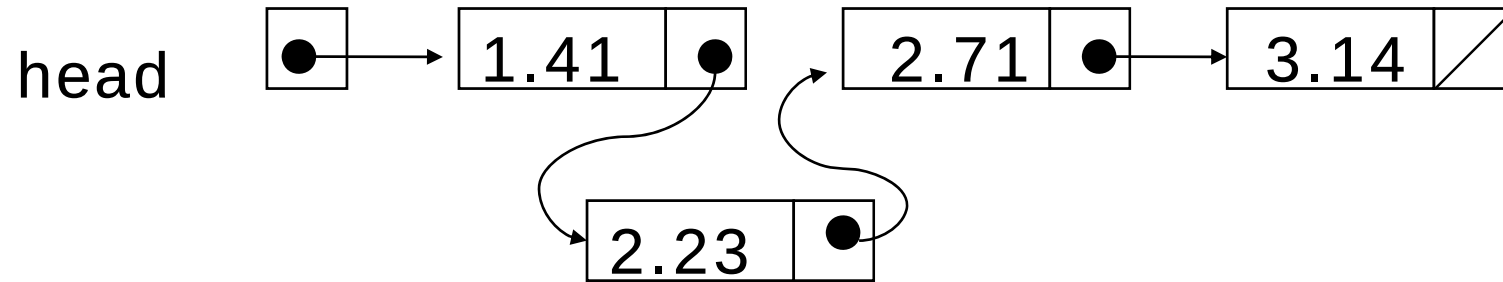
である。

- (nデータが整列してある配列に、整列を保ったまま挿入する時間計算量は、1回あたり、

$O(n)$  時間

であることに注意する。)

# 連結リストからのデータ削除





# 実現例

```
/*位置posの後のセルを削除(概略)*/  
void delete(Cell * pos)  
{  
    Cell *old;  
    old=pos->next;  
    pos->next=old->next;  
    free(old);  
    return;  
};
```

メモリの解放

# 連結リストへの削除の計算量

- 定数回の代入演算を行っているだけであるので、1回あたりの時間計算量は、  
 $O(1)$  時間  
である。
- (nデータが整列してある配列に、整列を保ったまま削除する時間計算量は、1回あたり、  
 $O(n)$  時間  
であることに注意する。)

# 連結リストと配列1

## (データ構造の準備)

head 

NULL

連結リストは、  
宣言時には、先頭の  
ポインタだけが必要。

配列は、宣言時にメモリが  
確保される。

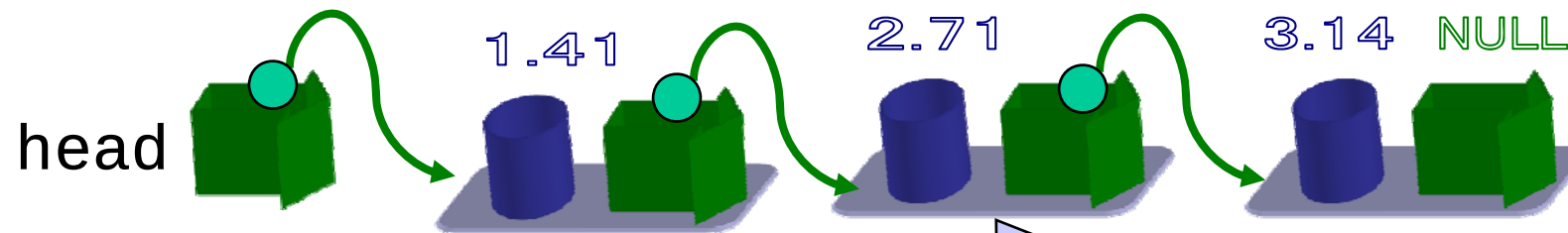
配列A   
A[0] A[1] A[i] A[j] A[M-1]

# 実現例

```
/*リストの用意*/  
Cell * make_list(void)  
{  
    Cell * head=NULL;  
    return head;  
}
```

```
/*配列の用意*/  
double * make_array(void)  
{  
    double A[100];  
    return A;  
}
```

# 連結リストと配列2 (要素へのアクセス)



途中のセルには、  
名前がついていない。

$A[i]$ として途中のセルにも  
アクセスできる。



# 実現例

```
/*リスト2番目のデータを返す。(概略)*/  
double retrun_second_list(Cell * head)  
{  
    doulbe tmp;  
    tmp=head->next->next->data;  
    return tmp;  
}
```

```
/*配列の2番目のデータを返す。*/  
double return_second_array(double *A)  
{  
    return A[1];  
}
```

# 練習

連結リストおよび、配列において、  
3番目の要素を返すC言語の関数の概略を示せ。

さらに、連結リストおよび、配列において、  
k番目の要素を返すC言語の関数の概略を示せ。

## 連結リストのk番目の要素参照に必要な計算量

- 定数回の代入演算を行っているだけであるので、1回あたりの時間計算量は、  
 $O(k)$  時間  
である。
- (nデータがの配列のk番目を参照するには、1回あたり、  
 $O(1)$  時間  
であることに注意する。)



# 連結リストと配列(まとめ)

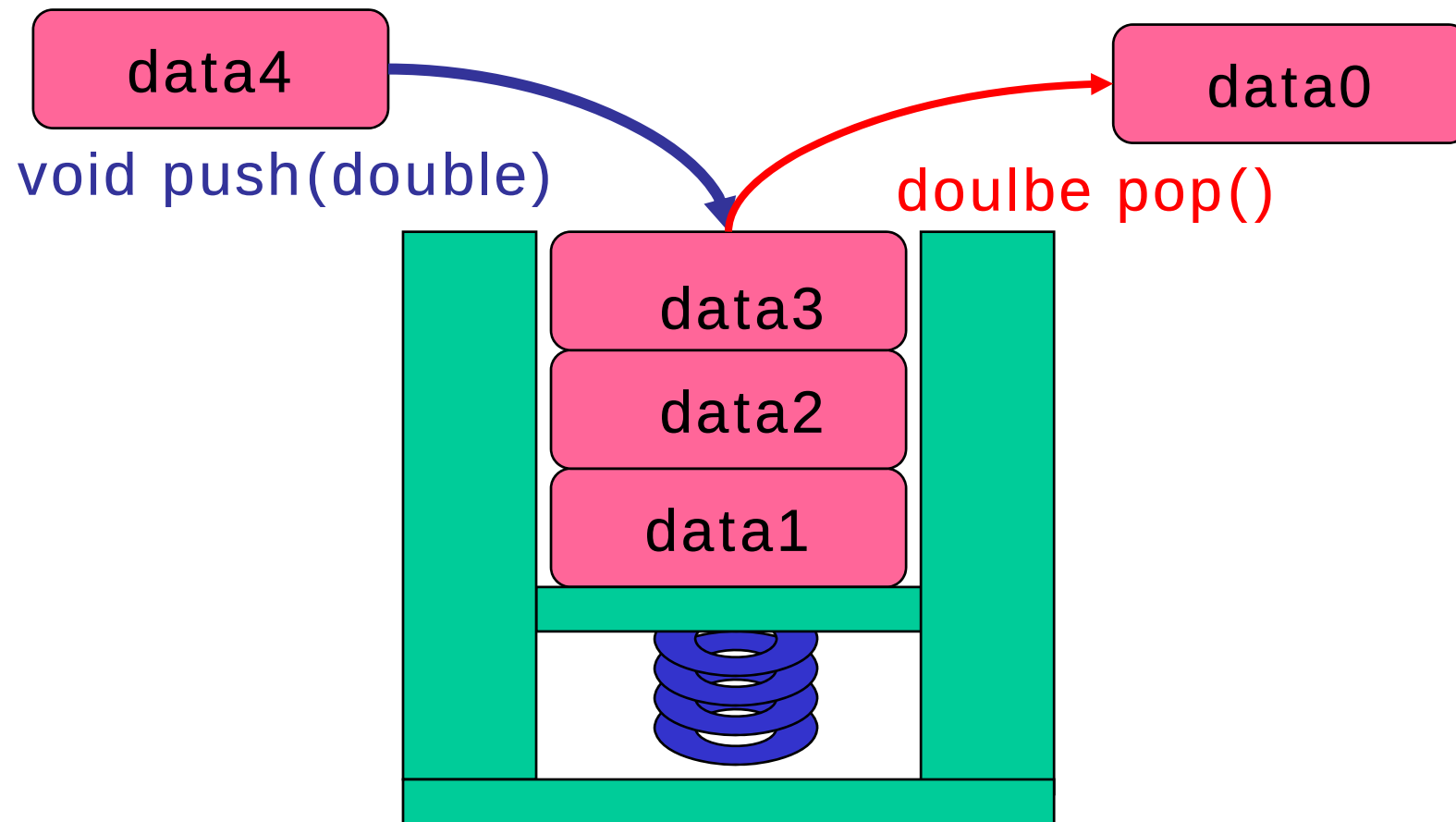
|           | 連結リスト  | 配列     |
|-----------|--------|--------|
| 挿入        | $O(1)$ | $O(n)$ |
| 削除        | $O(1)$ | $O(n)$ |
| $k$ 番目の参照 | $O(k)$ | $O(1)$ |

$n$  : データ数

## 6ー2. スタック(Stack)

- 後入れ先出し (Last In First Out, LIFO) の効果を持つデータ構造。
- 先頭においてしかデータの挿入、削除を行わない連結リスト

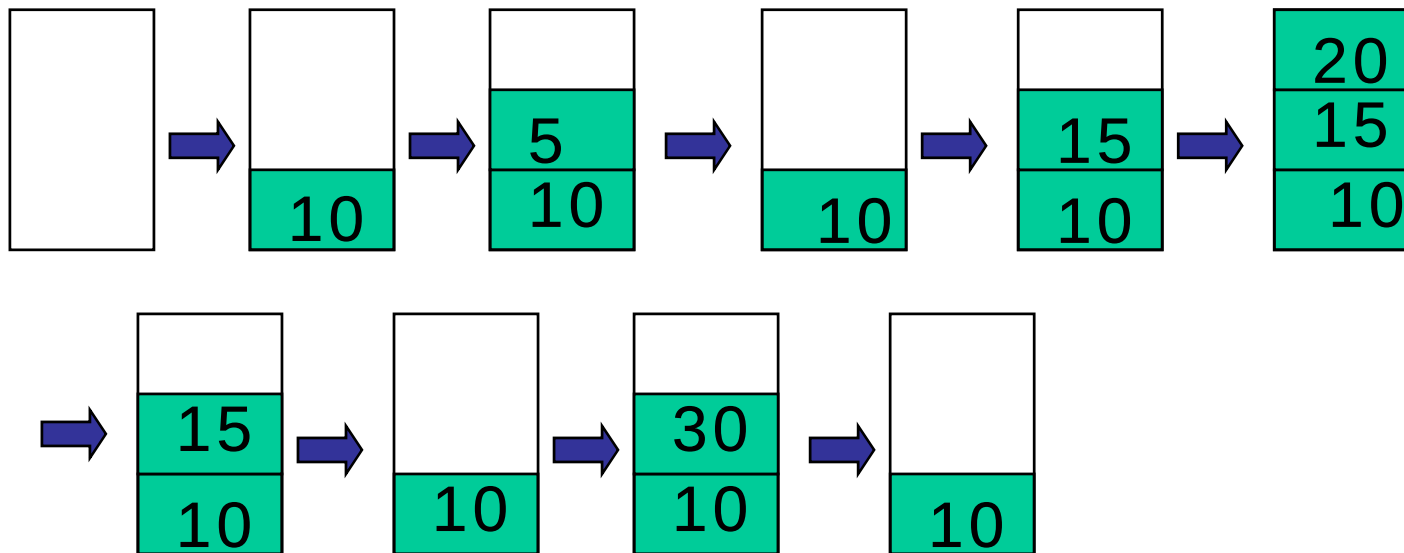
# スタックのイメージ



# 例題

空のスタックに対して、次の系列で演算を行った場合、  
取り出されるデータの順序および、  
最後のスタックの状態を示せ。

push(10), push(5), pop(), push(15), push(20),  
pop(), pop(), push(30), pop()



5 → 20 → 15 → 30

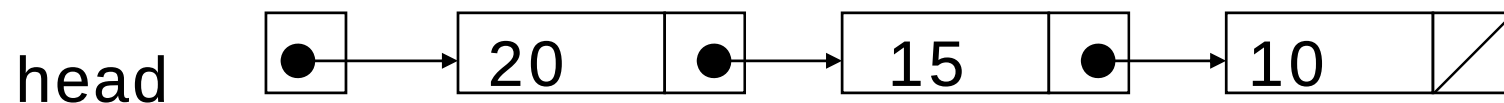
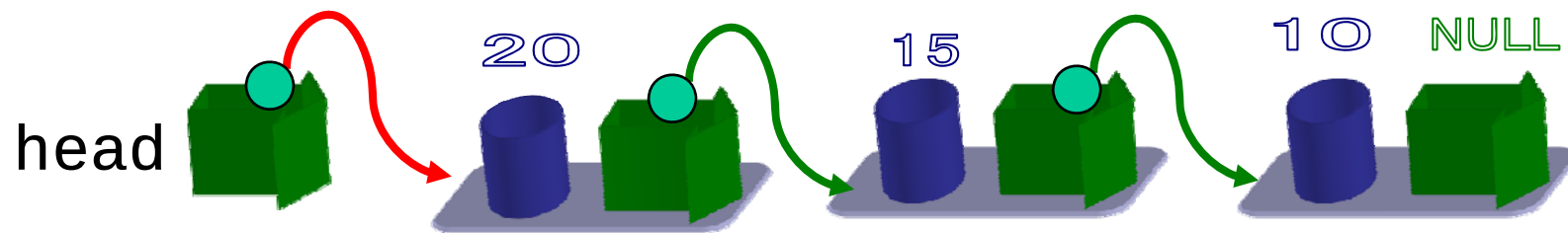
# 練習

空のスタックに対して、次の系列で演算を行った場合、  
取り出されるデータの順序および、  
最後のスタックの状態を示せ。

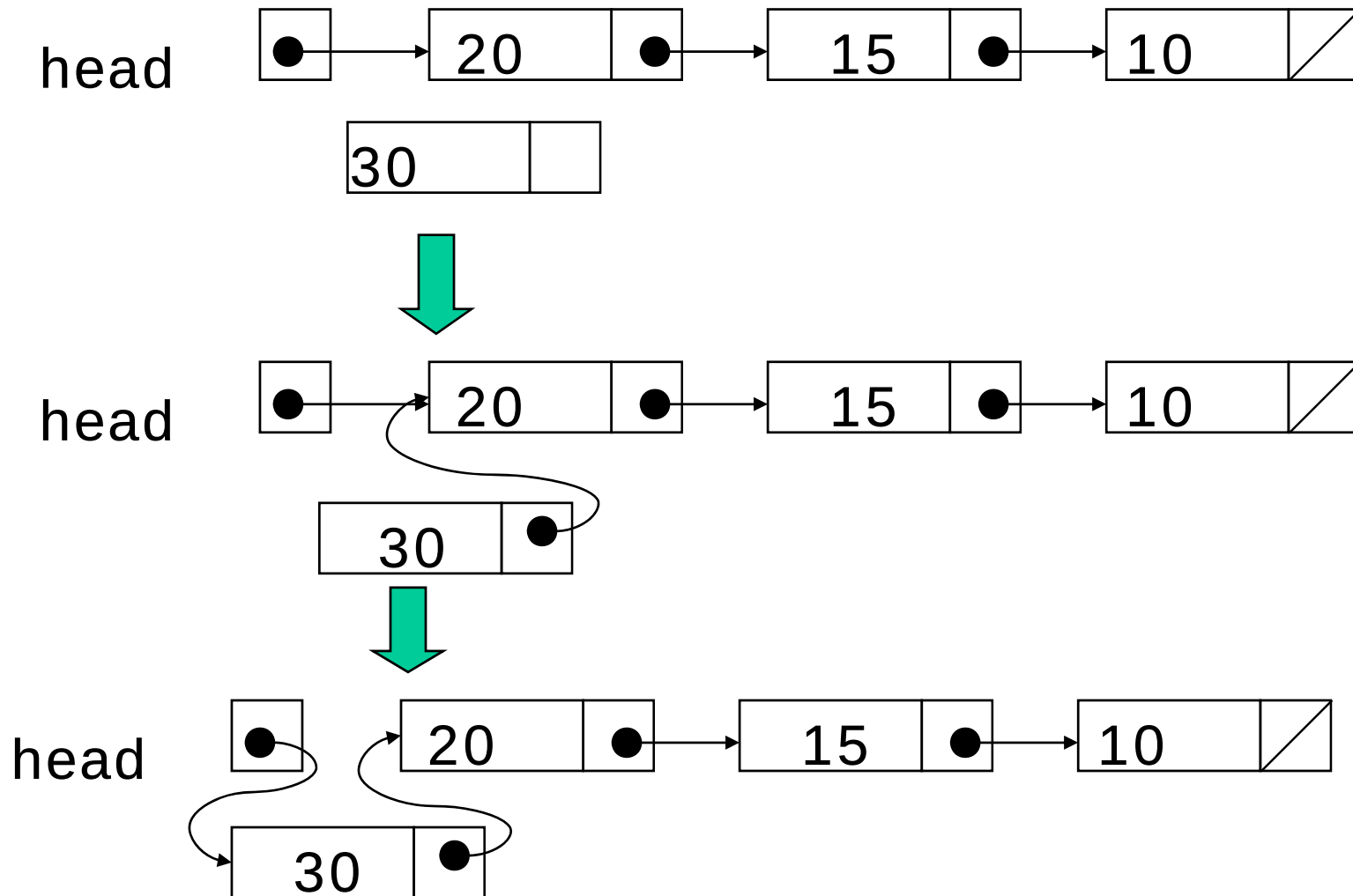
`push(5),pop(),push(7),push(3),pop(),push(8),  
,push(1),pop(),pop(),push(9),pop(),pop()`

# 連結リストによるスタック

先頭だけで、データの挿入削除を行う。  
途中の接続関係は維持したままにする。



# push(x)



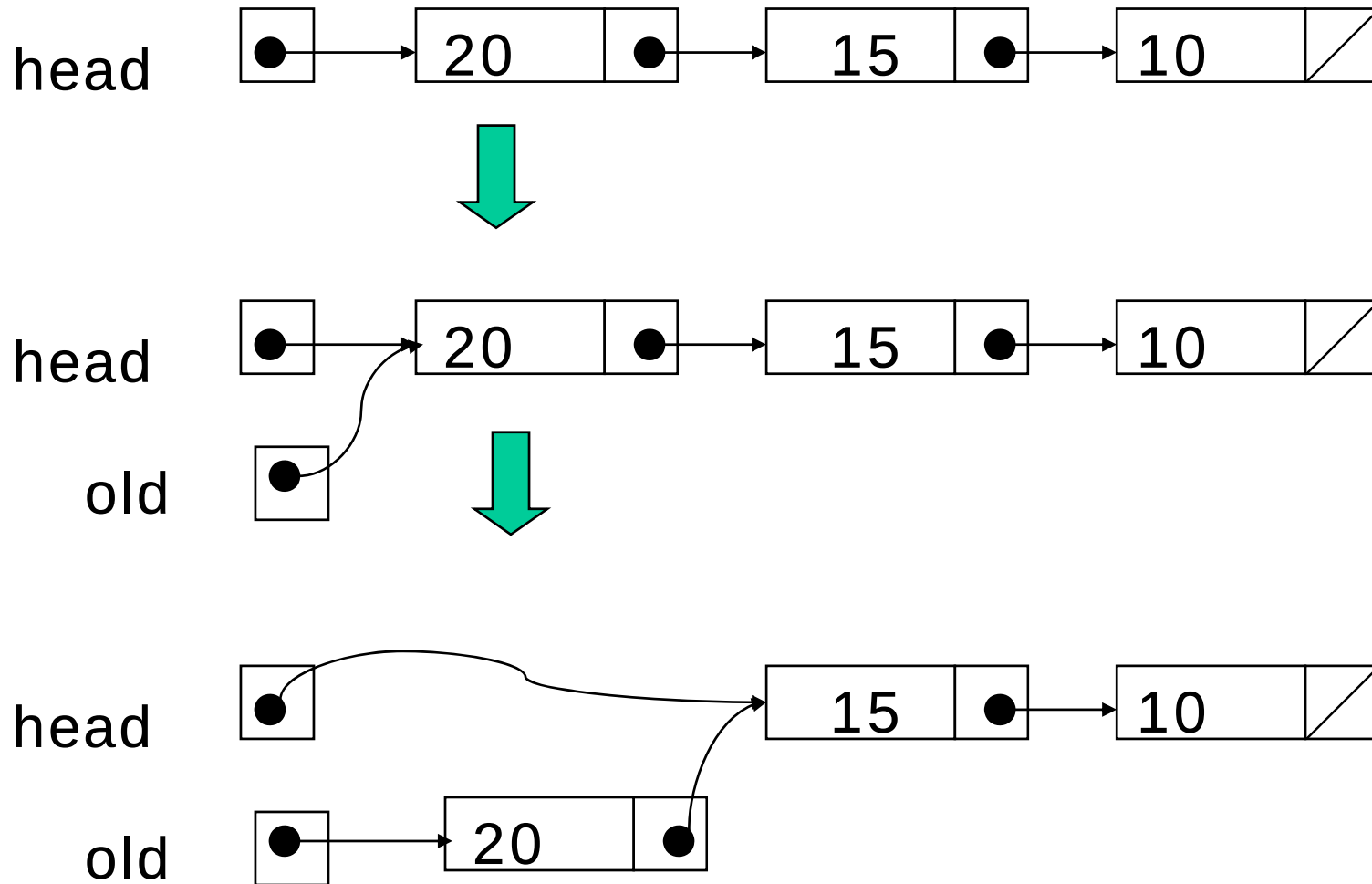
# 実現例

```
/*スタックへのpush(x)*/  
void push(Cell * head,double x)  
{  
    Cell *new=(Cell *)malloc(sizeof(Cell));  
    new->data=x;  
    new->next=head;  
    head=new;  
    return;  
};
```

ポインタにはアドレスが保持してあることに注意して更新の順序を考えること。



# pop()

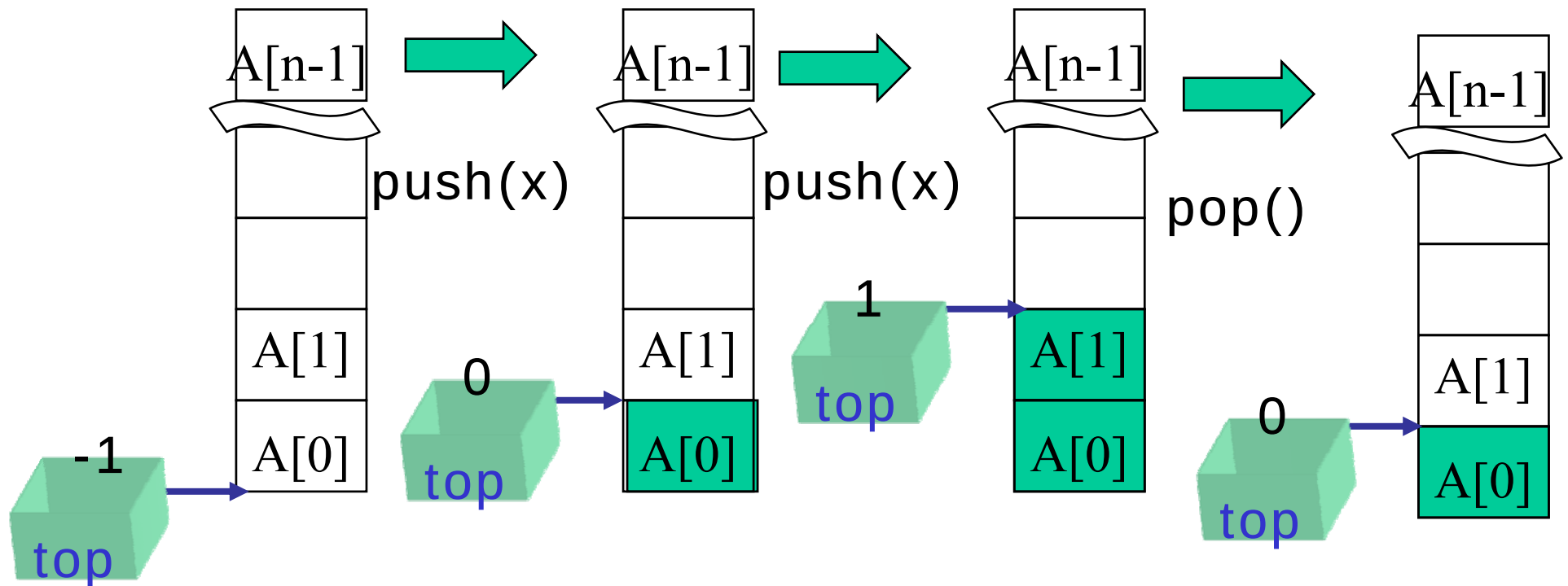


# 実現例

```
/*スタックからのpop()概略*/  
double pop(Cell * head)  
{  
    Cell *old;  
    double x;  
    if(head == NULL) return -1; /*アンダーフロー*/  
    old = head;  
    head = head->next;  
    x = old->data;  
    free(old);  
    return x;  
};
```

# 配列によるスタック

配列でもスタックを実現することができる。



# 実現例

```
/* 配列でのスタック(概略) */  
int Top;  
double Stack[100];  
void push(double x){  
    Top++;  
    if(Top >= 100) return; /* オーバーフロー */  
    Stack[Top] = x;  
    return;  
};  
  
double pop(void){  
    double x;  
    if(Top < 0) return -1; /* アンダーフロー */  
    x = Stack[Top];  
    Top--;  
    return x;  
}
```

# スタック操作の計算量

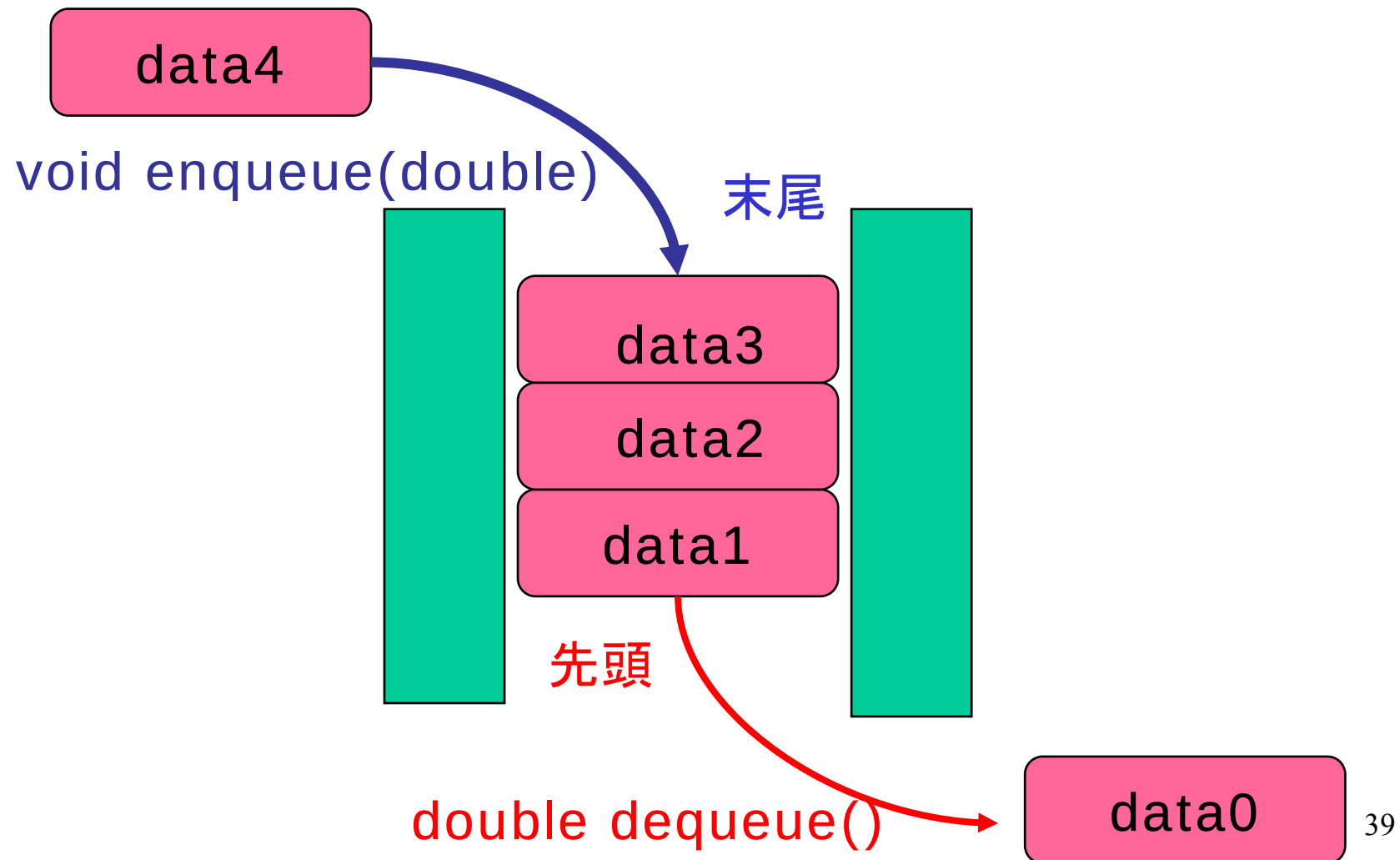
|                         | 連結リスト  | 配列     |
|-------------------------|--------|--------|
| push<br>(先頭への要素<br>挿入)  | $O(1)$ | $O(1)$ |
| Pop<br>(先頭要素の<br>削除と取得) | $O(1)$ | $O(1)$ |

$n$  : データ数

## 6－3. キュー(Queue)

- 先入れ先出し(First In First Out,FIFO)の効果を持つデータ構造。
- 先頭と末尾を管理する。
- データの挿入を末尾から行い、データの削除を先頭から行う連結リスト

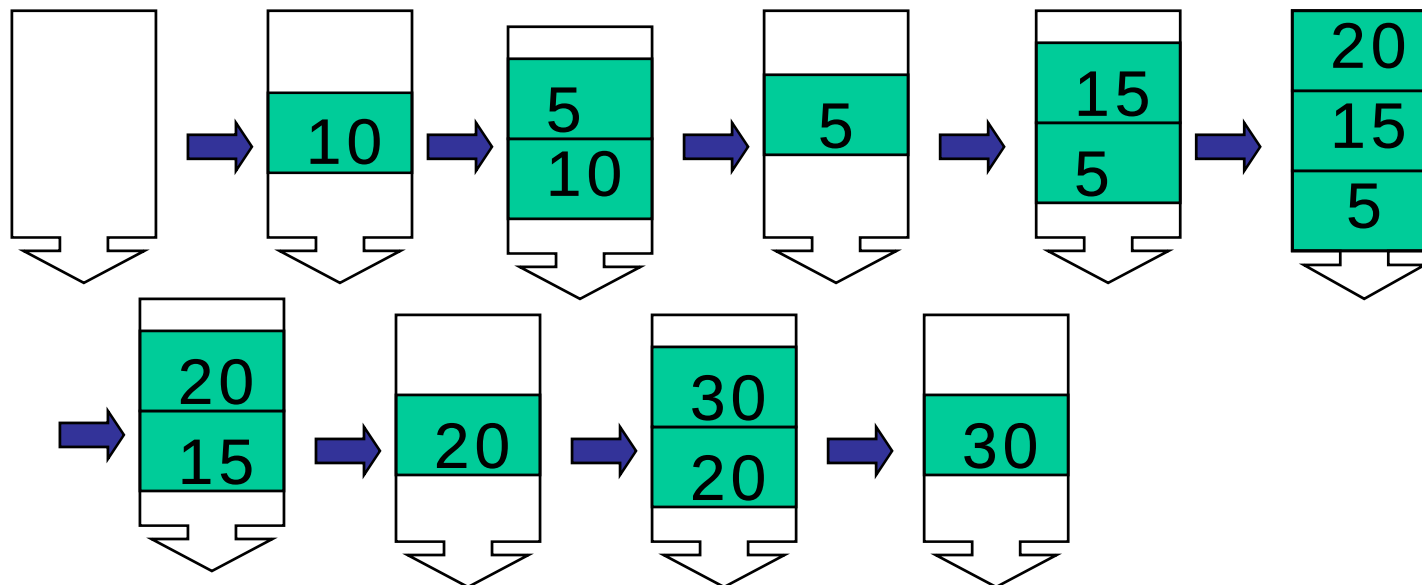
# キューのイメージ



# 例題

空のキューに対して、次の系列で演算を行った場合、  
取り出されるデータの順序および、  
最後のスタックの状態を示せ。

enqueue(10), enqueue(5), dequeue(), enqueue(15),  
enqueue(20), dequeue(), dequeue(), enqueue(30),  
dequeue()



10 → 5 → 15 → 20



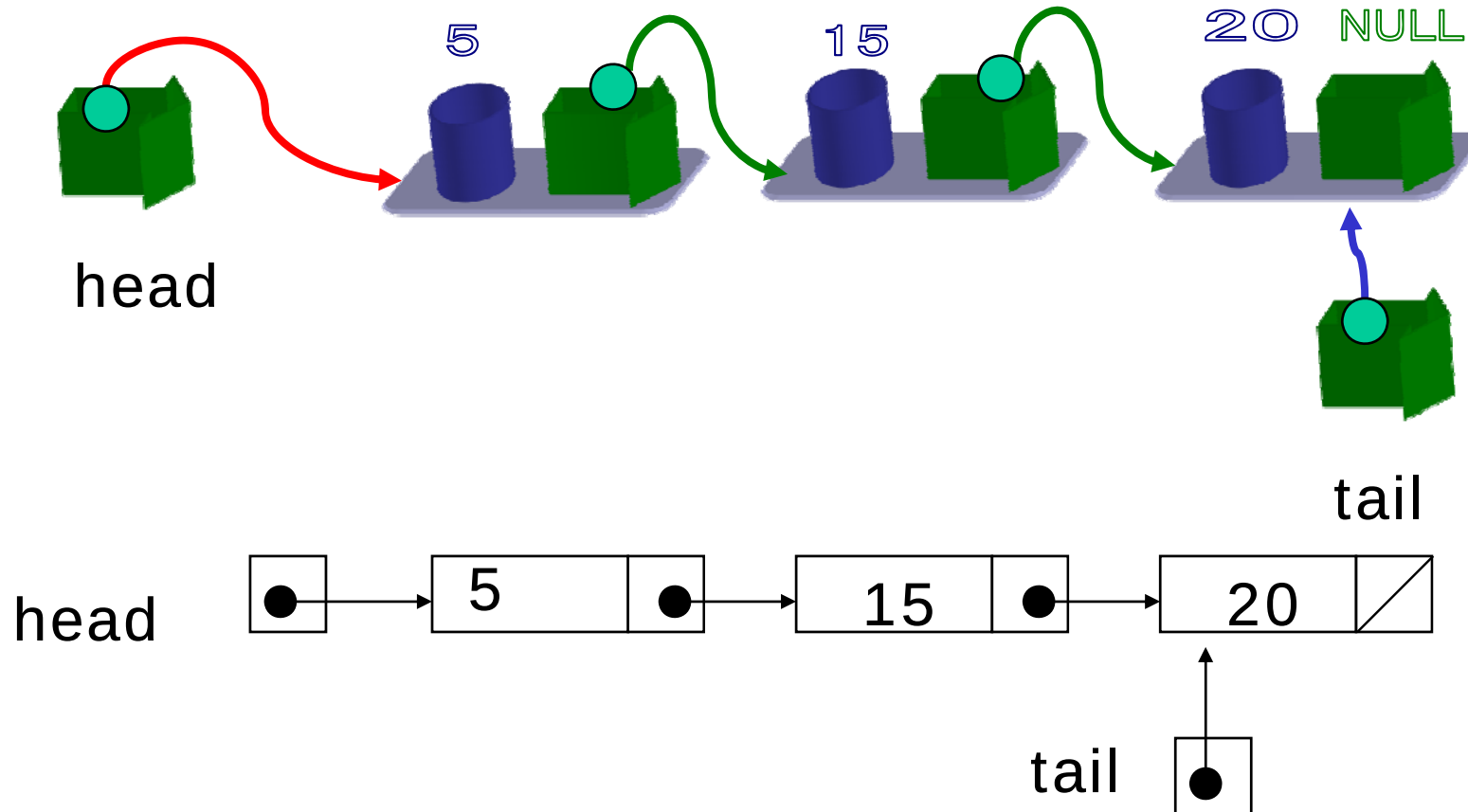
# 練習

空のキューに対して、次の系列で演算を行った場合、  
取り出されるデータの順序および、  
最後のキューの状態を示せ。

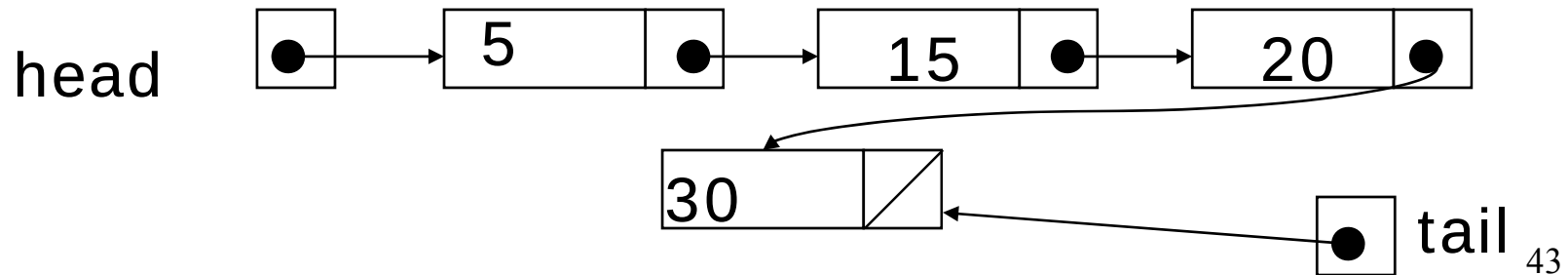
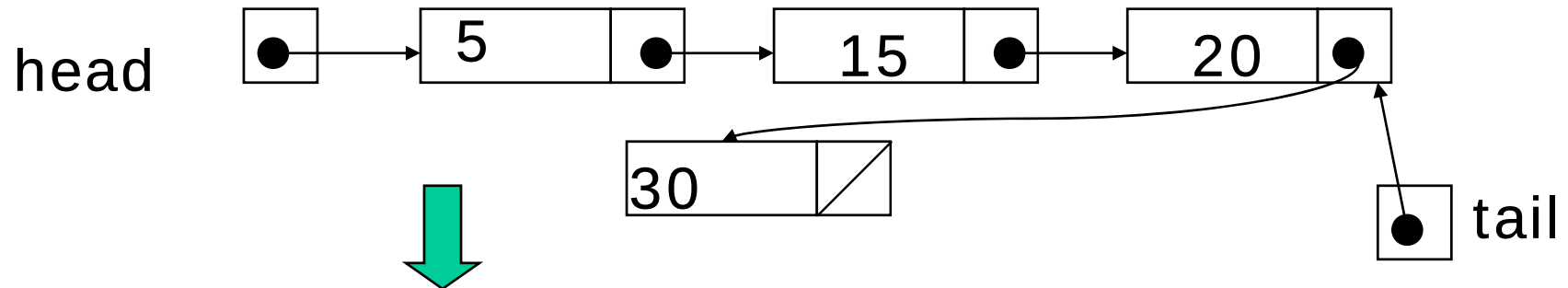
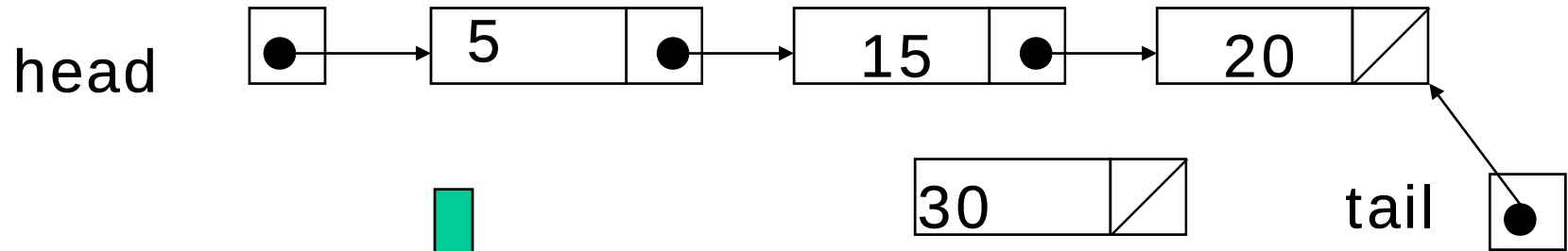
```
enqueue(5),dequeue(),enqueue(7),enqueue(3),  
dequeue(),enqueue(8),enqueue(1),dequeue(),  
dequeue(),enqueue(9),dequeue(),dequeue()
```

# 連結リストによるキュー

末尾からデータの挿入し、  
先頭からデータを削除を行う。  
途中の接続関係は維持したままにする。



# enqueue(x)

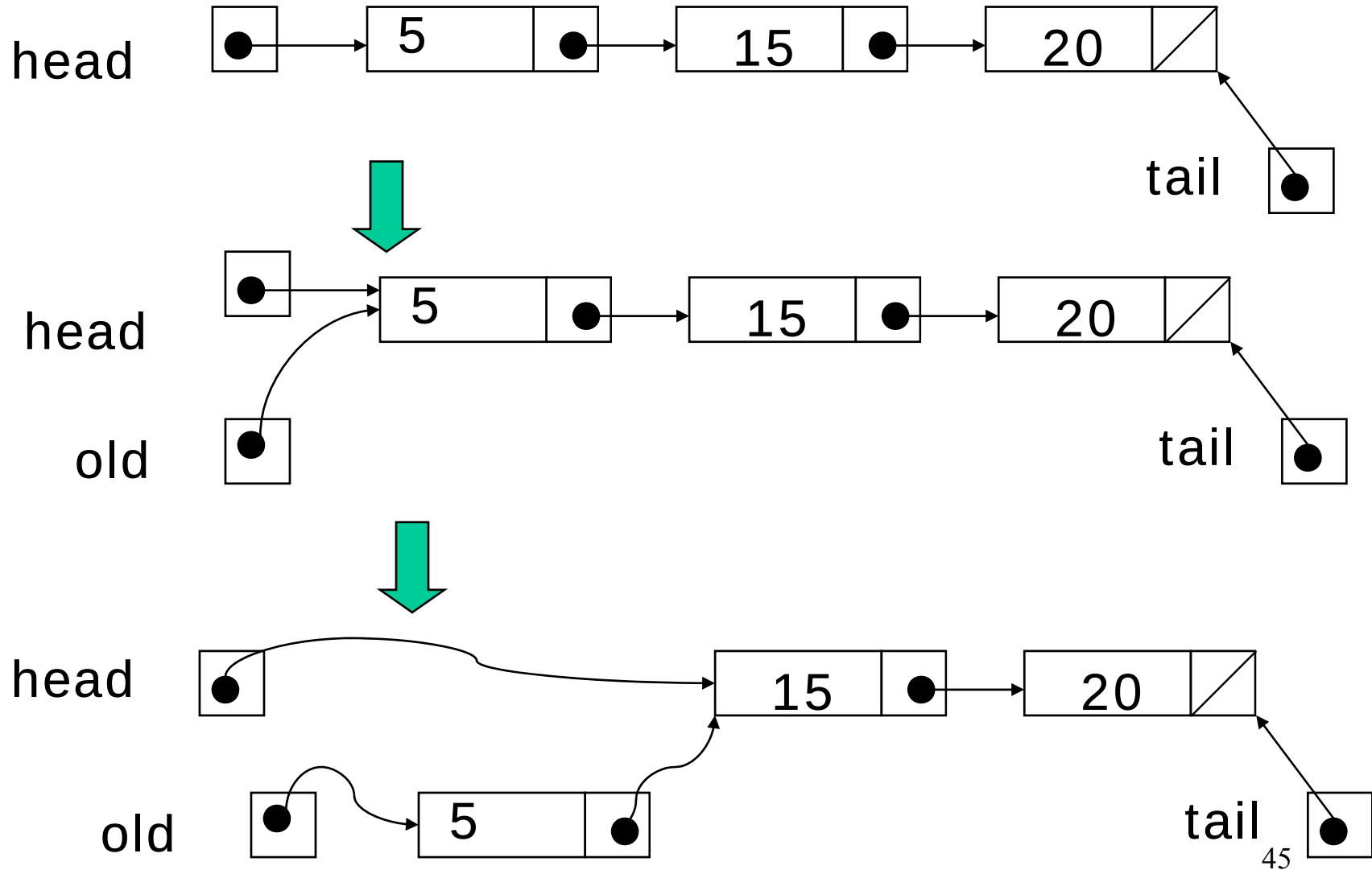


# 実現例

```
/* キューへのenqueue(x) */  
void enqueue(Cell * head, Cell * tail, double x)  
{  
    Cell *new = (Cell *) malloc(sizeof(Cell));  
    new->data = x;  
    new->next = NULL;  
    tail->next = new;  
    tail = new;  
    return;  
};
```

ポインタにはアドレスが保持してあることに注意して更新の順序を考えること。

# dequeue()

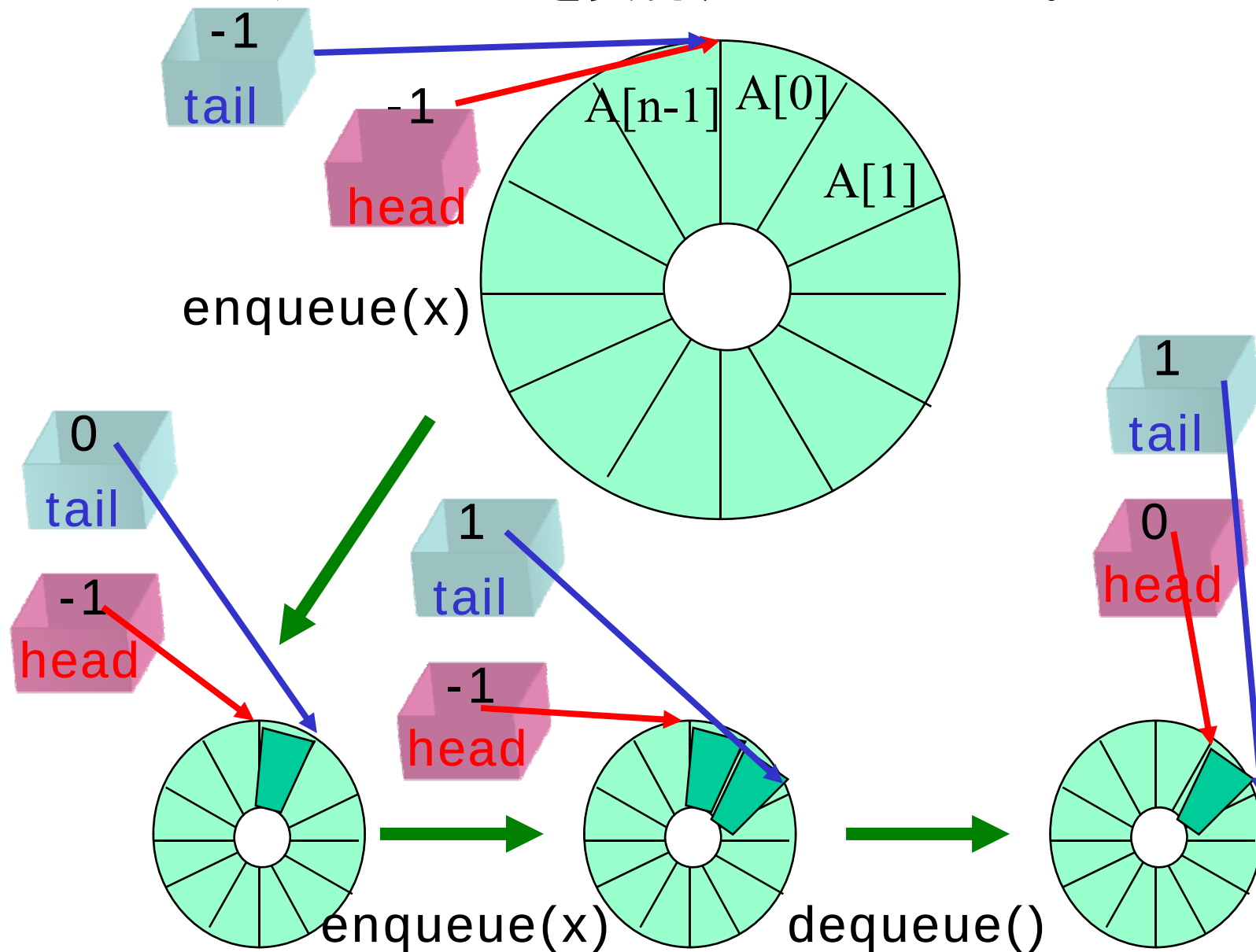


# 実現例

```
/* キューからのdequeue()概略 */  
double dequeue(Cell * head, Cell * tail)  
{  
    Cell *old;  
    double x;  
    if(head == NULL) return -1; /* アンダーフロー */  
    old = head;  
    head = head->next;  
    x = old->data;  
    free(old);  
    return x;  
};
```

# 配列によるキュー(リングバッファ)

配列でもキューを実現することができる。



# 実現例

```
/* 配列でのキュー(概略) */
int Head;
int Tail;
double Queue[100];
void enqueue(double x){
    Tail = (Tail + 1) % 100; /* 添え字の循環 */
    if(Tail == Head) return; /* オーバーフロー */
    Queue[Tail] = x;
    return;
};

double dequeue(void){
    double x;
    if(Head == Tail) return -1; /* アンダーフロー */
    Head = (Head + 1) % 100; /* 添え字の循環 */
    x = Queue[Head];
    return x;
}
```



# キュー操作の計算量

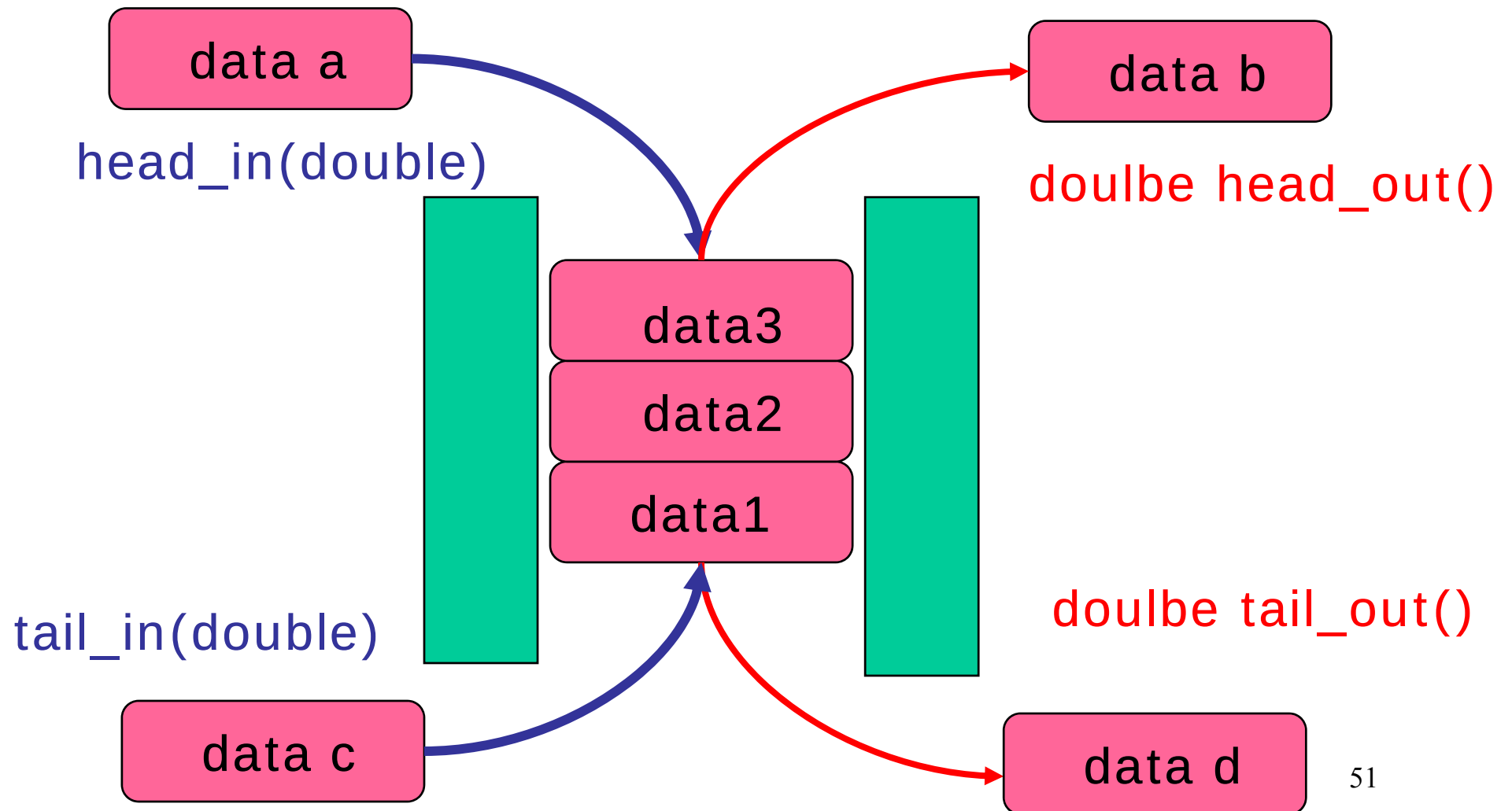
|                             | 連結リスト  | 配列     |
|-----------------------------|--------|--------|
| enqueue<br>(末尾への要素<br>挿入)   | $O(1)$ | $O(1)$ |
| dequeue<br>(先頭要素の<br>削除と取得) | $O(1)$ | $O(1)$ |

$n$  : データ数

## 6-4. デク (Double Ended Queue)

- 先頭と末尾の両方からデータを出し入れできるデータ構造
- 先頭と末尾を管理。
- スタックとキューの性能を合わせ持つ。
- 双方向リストを持ちいて実現される。

# デクのイメージ



# デクの実現のためには

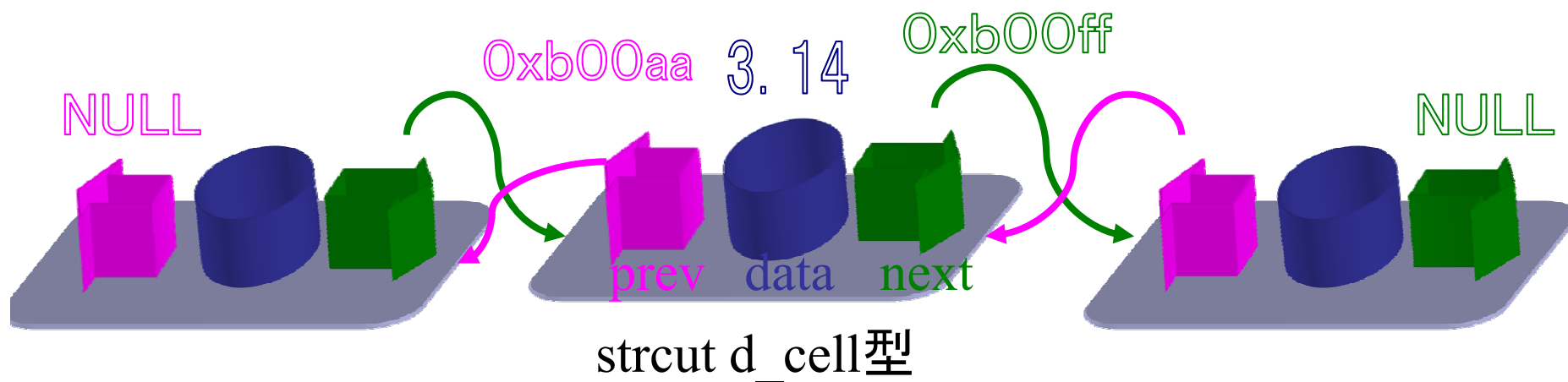
- 連結リストを拡張し、双方向リストにする必要がある。
- セルを拡張する。

# 双方向リストのセル

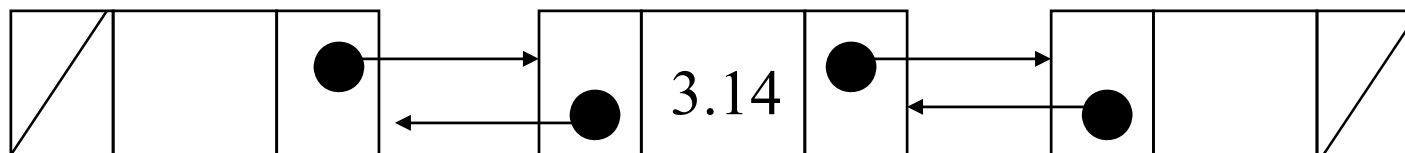
```
struct d_cell  
{  
    double data;  
    struct d_cell * prev;  
    struct d_cell * next;  
};
```

前方を指すポインタと、後方を指すポインタとして実現。

# イメージ



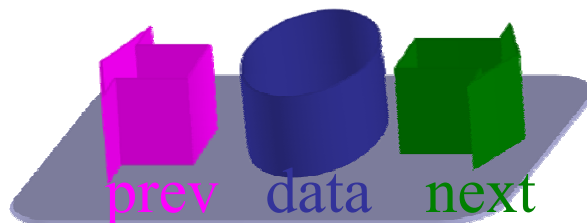
このことを、次のような図を用いて表すこともある。



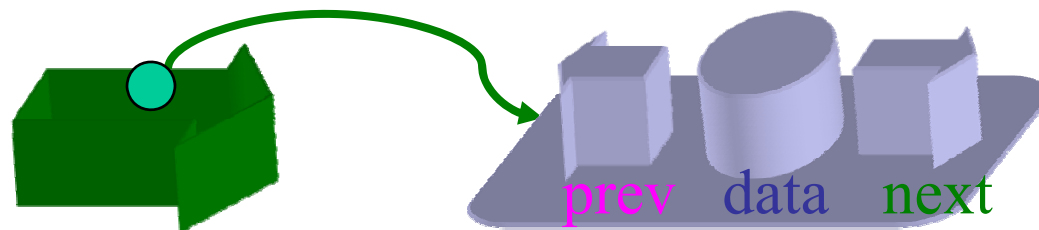
strcut d\_cell型

# 双方向リストのセル型の定義

```
typedef struct d_cell  D_cell;
```



D\_cell型



D\_cell \* 型

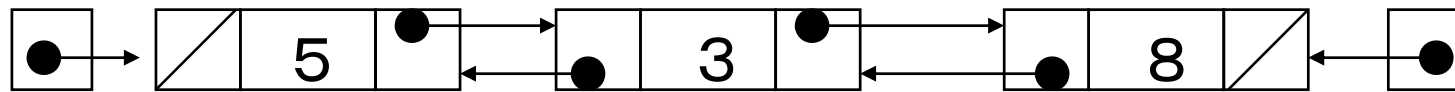
# 双方向リストによるデク





# 練習

デクにおける挿入および削除の様子を図示せよ。  
また、挿入を表す関数を作成せよ。



head

tail

操作例: head\_in(6) → tail\_in(2) → head\_out() → tail\_out()

# デク操作の計算量

|                          | 連結リスト  | 配列     |
|--------------------------|--------|--------|
| head_in<br>(先頭へ要素挿入)     | $O(1)$ | $O(1)$ |
| head_out<br>(先頭要素の削除と取得) | $O(1)$ | $O(1)$ |
| tail_in<br>(末尾へ要素挿入)     | $O(1)$ | $O(1)$ |
| tail_out<br>(末尾要素の削除と取得) | $O(1)$ | $O(1)$ |

## 6-5. 抽象データ型 (Abstract Data Type)

- 同じ操作法と、同じ効果を持つデータ構造を抽象データ型という。
- 例えば、スタックやキューは抽象データ型。  
(個々の実装は、連結リストや配列で行われる。しかし、重要なのは、その操作法と効果である。)

# 抽象データ型としてのスタック

## スタック

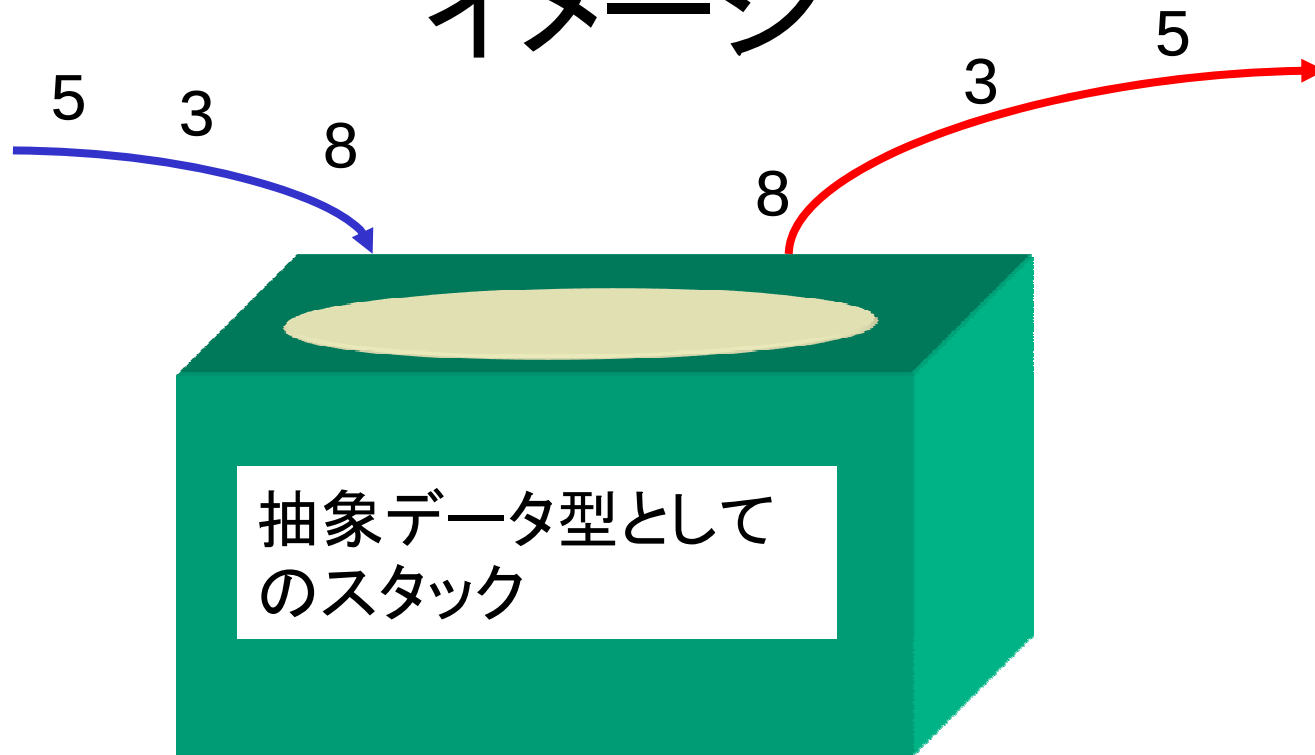
### 操作

```
/*データの挿入*/  
void push(x);/  
  
/*データの取り出し*/  
double pop(void);
```

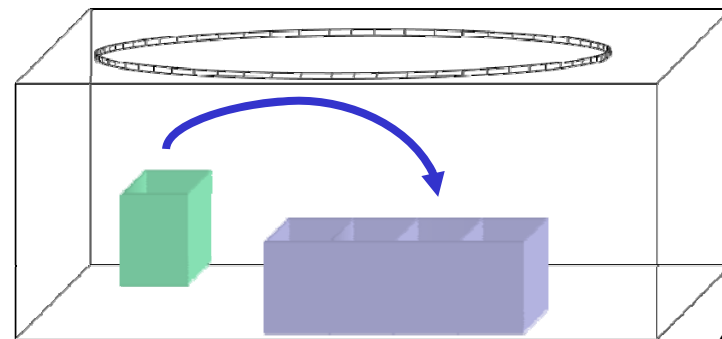
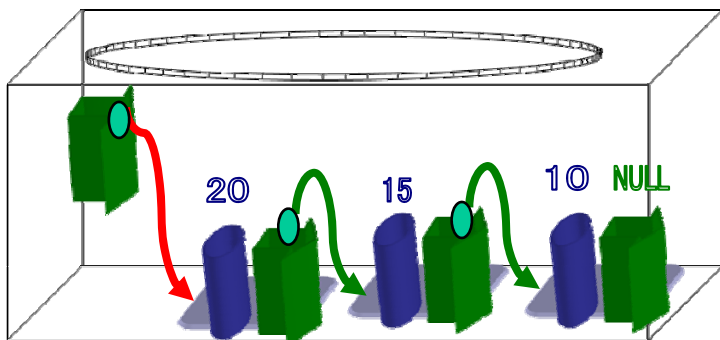
### 効果

LIFO  
(後入れ先だし)  
つまり、  
x=pop()を行って  
push(x)を行えば  
もとの状態にも  
どる。

# イメージ



データ構造では、操作法とその効果だけが重要



# 抽象データ型としてのキュー

## キュー

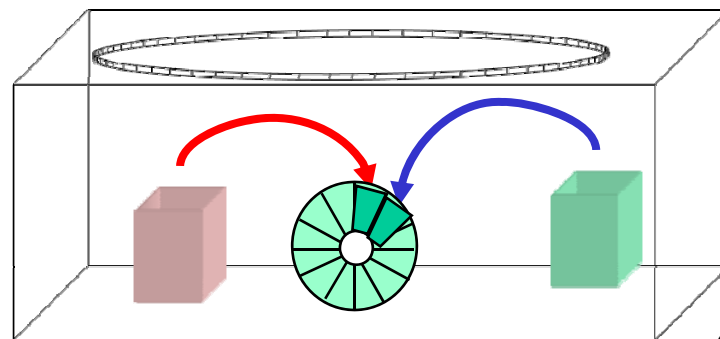
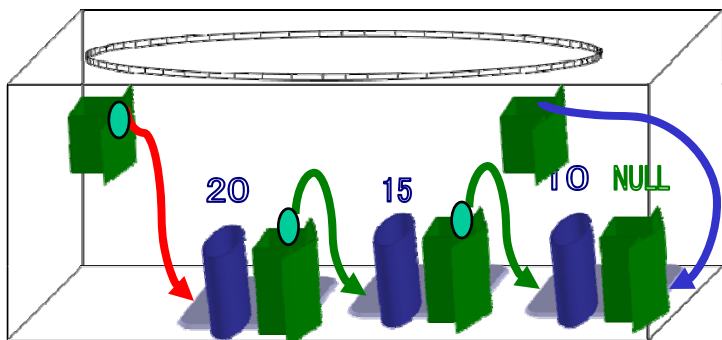
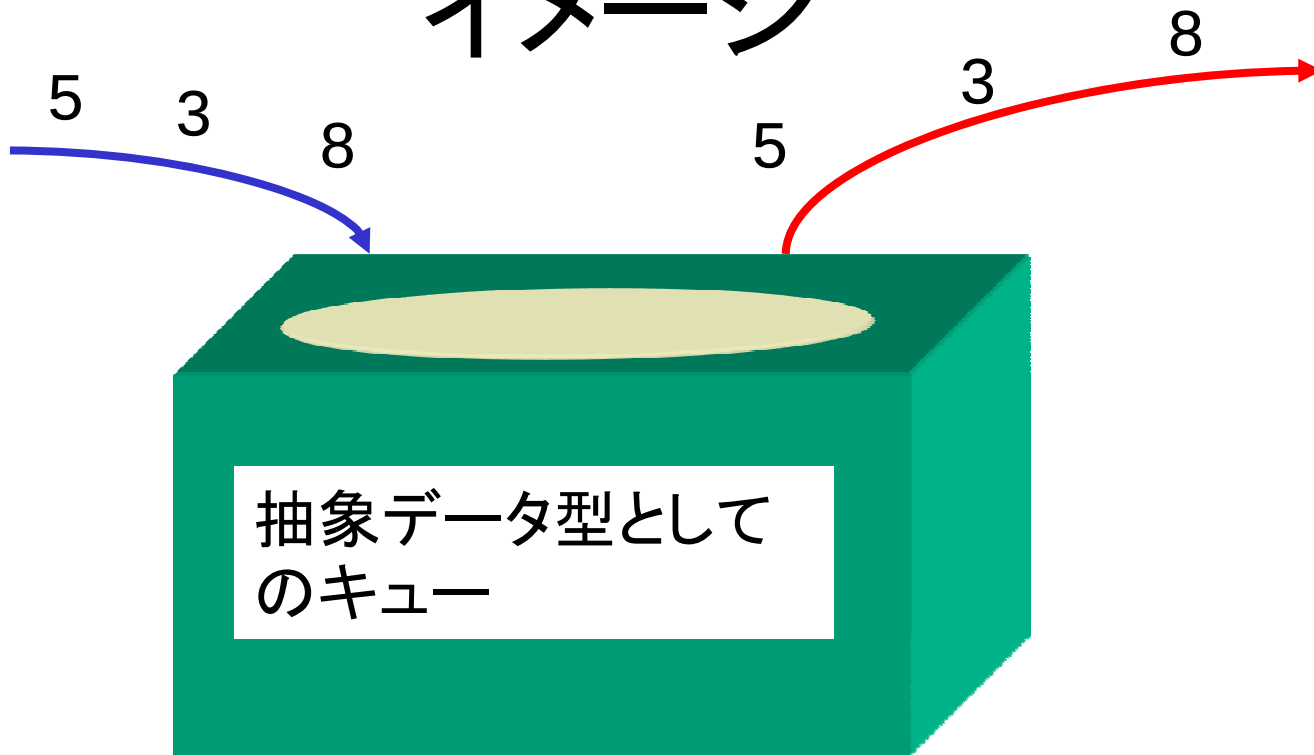
### 操作

```
/*データの挿入*/  
void enqueue(x);  
  
/*データの取り出し*/  
double dequeue(void);
```

### 効果

FIFO  
(先入れ先だし)  
つまり、  
dequeueで取り  
出される順番は、  
各要素が  
enqueueされた  
順番とひとしい。

# イメージ



# 抽象データ型の利用

```
/*プロトタイプ宣言*/  
/*連結リストによるスタック*/  
void push( double);  
double pop();  
main()  
{  
    push(4);  
    push(5);  
    x=pop();  
    y=pop();  
}
```

```
/*プロトタイプ宣言*/  
/*配列によるスタック*/  
void push( double);  
double pop();  
main()  
{  
    push(4);  
    push(5);  
    x=pop();  
    y=pop();  
}
```

全く同一。