# データ構造とアルゴリズム

## 第３回 基本的データ構造

小池 英樹 (*koike@c.titech.ac.jp*)

# 抽象データ型(ABSTRACT DATA TYPE)

- データ構造とそれを操作する手続きをまとめてデータ型の定義とすることでデータ抽象(data abstraction)を行う手法.

- 例：線形リストとそれを操作する手続き

  データ構造：線形リスト

  操作：create()

  insert()
  delete()
  …

- オブジェクト指向言語へと展開.

# リスト

- 要素を0個以上１列に並べたもの
  - $a_0, a_1, \ldots, a_{n-1}$
- リストの長さ: 要素数n
- n=0のとき、空リスト(null list)という

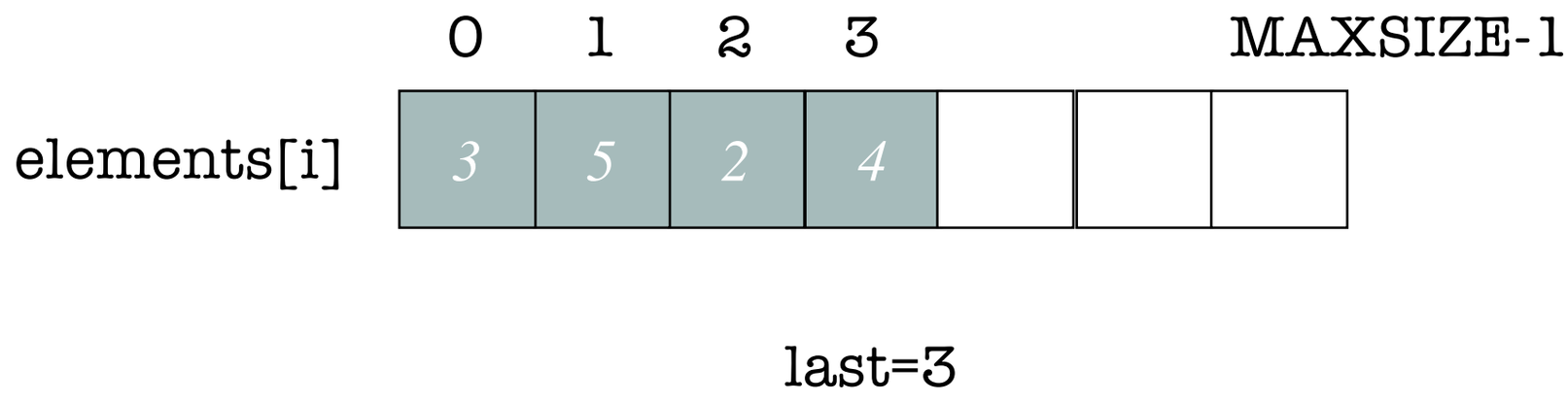
# リスト：抽象データ型

- どのような操作が必要か
  - 新しいリストを作成する　　()
  - 要素を挿入する
    - 例：前から3番目に'1'を挿入する　　(3, 5, 2, 4) -> (3, 5, 1, 2, 4)
  - 要素を削除する
    - 例：前から2番目の要素を削除する　　(3, 5, 1, 2, 4) -> (3, 1, 2, 4)
  - 要素を順に出力する
  - …

- API (Application Programmers' Interface) という考え方

# リスト：配列による実現

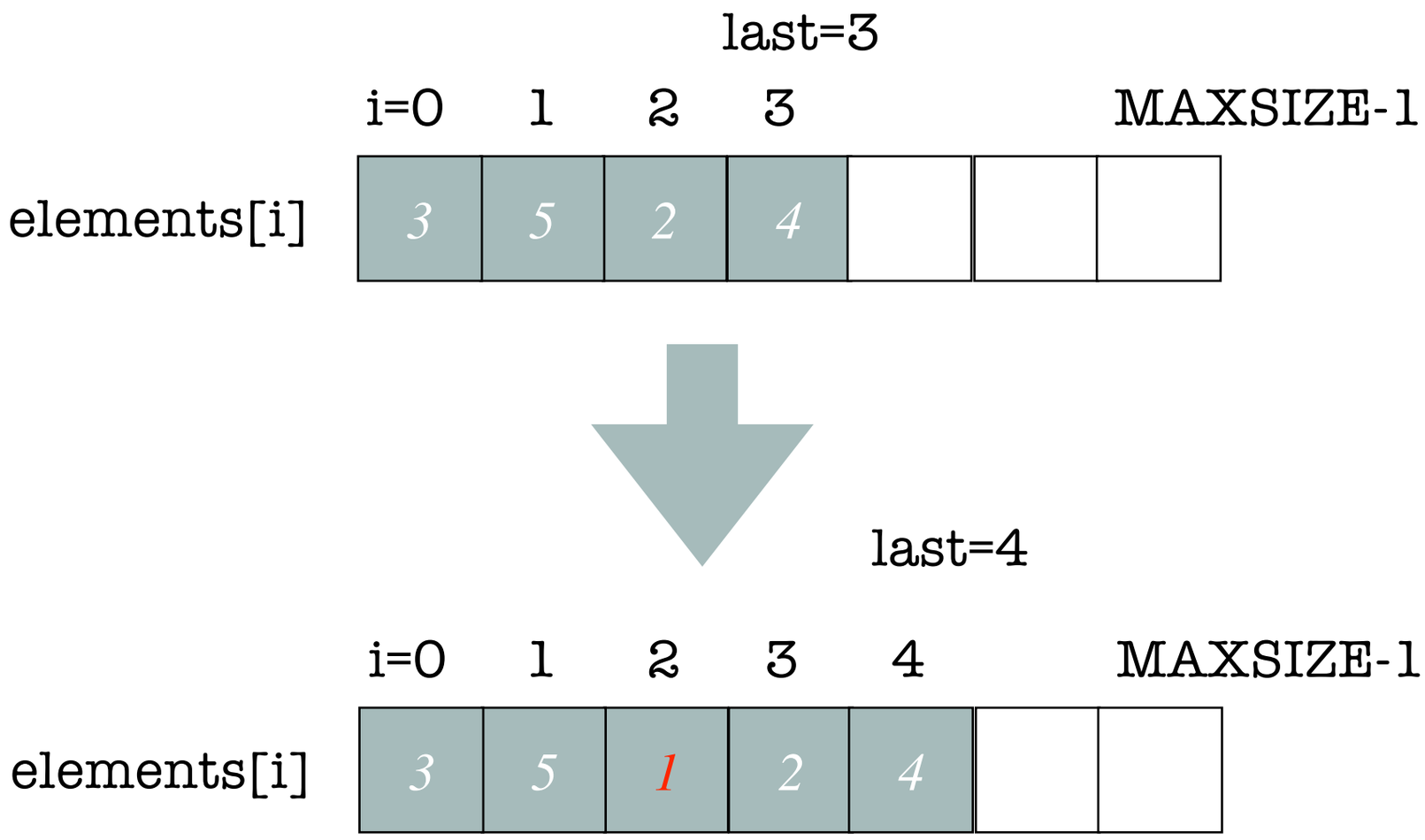
```
struct list {
    int elements[MAXSIZE];
    int last;
};
```



# リスト：配列による実現

・挿入

例：前から3番目に'1'を挿入する




4番目以降を1つずつずらし、lastを1増加させる。

その後、3番目に'1'を代入する

# リスト：配列による実現

擬似コードによる段階的詳細化（第 1 段）

```
void insert(int x, int p, struct list *l) {
    for lastからp番目まで以下を繰り返す
        値を1つずつ右にシフト
        lastの値を1増加
        p番目にxを代入
}
```

# リスト：配列による実現

擬似コードによる段階的詳細化（第２段）

```
void insert(int x, int p, struct list *l) {
    if (リストが一杯) {
        エラー処理
    } else if (pの値が範囲外) {
        エラー処理
    } else {
        for lastからp番目まで以下を繰り返す
            値を1つずつ右にシフト
        lastの値を1増加
        p番目にxを代入
    }
}
```

# リスト：配列による実現

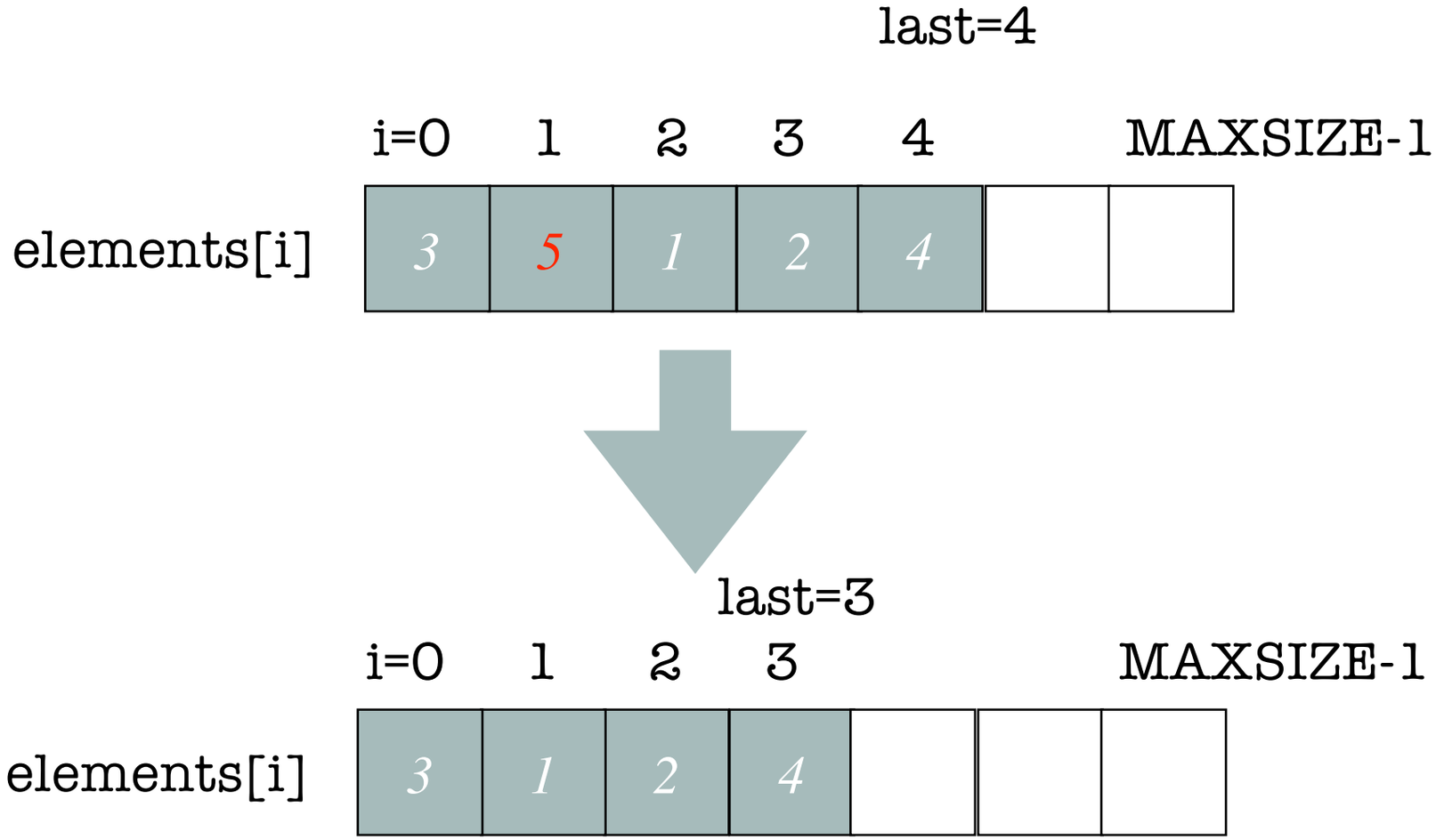
Cによる実装

```
void insert(int x, int p, struct list *l) {
    if (l->last >= MAXSIZE-1) {
        printf("error: list is full.\n”);
        exit(1);
    } else if (p-1 > l->last+1 || p < 0) {
        printf("error: no such position.\n”);
        exit(1);
    } else {
        for (int i=l->last; i >=p-1; i--) {
            l->elements[i+1] = l->elements[i];
        }
        l->last = l->last+1;
        l->elements[p-1] = x;
    }
}
```

# リスト：配列による実現

・削除

例：前から2番目の要素を削除する




3番目以降を左に1つずつずらして，lastを1減少．

# リスト：配列による実現

擬似コードによる段階的詳細化（第 1 段）

```
void delete(int p, struct list *l) {
      for p番目からlastまで繰り返す
            1つずつ左にシフト
            lastの値を1つ減少
}
```

# リスト：配列による実現

擬似コードによる段階的詳細化（第２段）

```
void delete(int p, struct list *l) {
    if pの範囲が適当でない
        エラー処理
    else
        for p番目からlastまで繰り返す
            1つずつ左にシフト
        lastの値を1つ減少
}
```

# リスト：配列による実現

```
void delete(int p, struct list *l) {
    if (p > l->last +1 || p < 0) {
        printf("error: no such position.\n”);
        exit(1);
    } else {
        for (int i=p-1; i<l->last; i++) {
            l->elements[i] = l->elements[i+1];
        }
        l->last = l->last-1;
    }
}
```

# リスト：配列による実現

・要素を出力する関数

```
void printlist(struct list *l) {
  for (int i=0; i<=l->last; i++) {
      printf(“%d ”, l->elements[i]);
  }
  printf(“\n”);
}
```

# リスト：配列による実現

・新しいリストを生成する関数

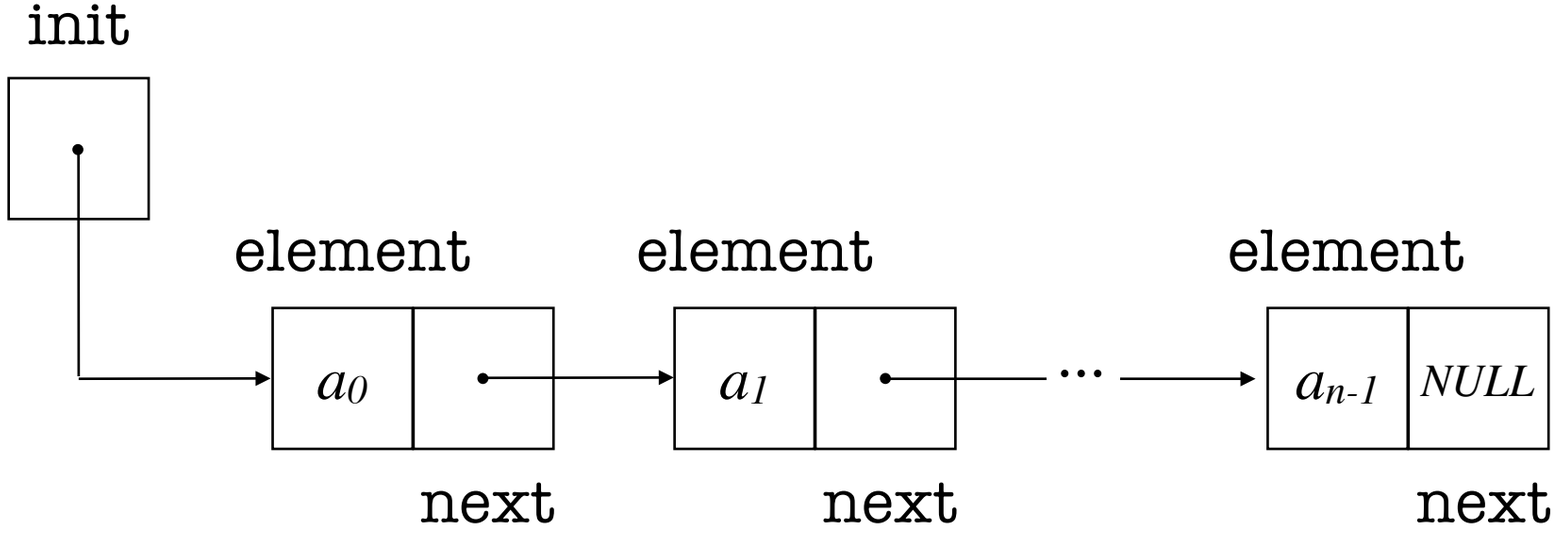
```
struct list *newlist() {
  struct list *l = (struct list *)malloc(sizeof(struct list));
  l->last = -1;

  return(l);
}
```

# リスト：配列による実現

```
int main() {
    struct list *l = newlist();

    insert(3, 1, l);
    insert(5, 1, l);
    insert(4, 2, l);

    delete(2, l);

    printlist(l);
}
```

# リスト：ポインタによる実現

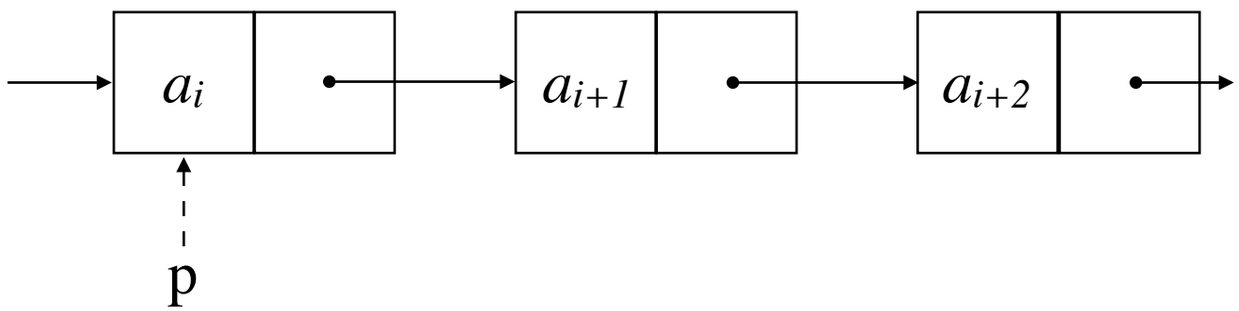
```
struct cell {
    int element;
    struct cell *next;
};
```



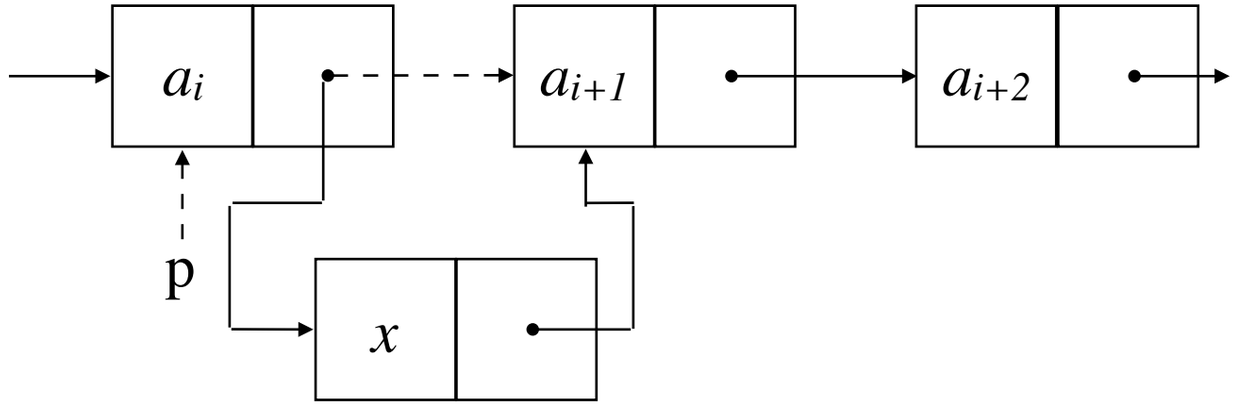

連結リスト（linked-list）線形リストとも言う

# リスト：ポインタによる実現

・挿入




insert(x, p, L)



# リスト：ポインタによる実現

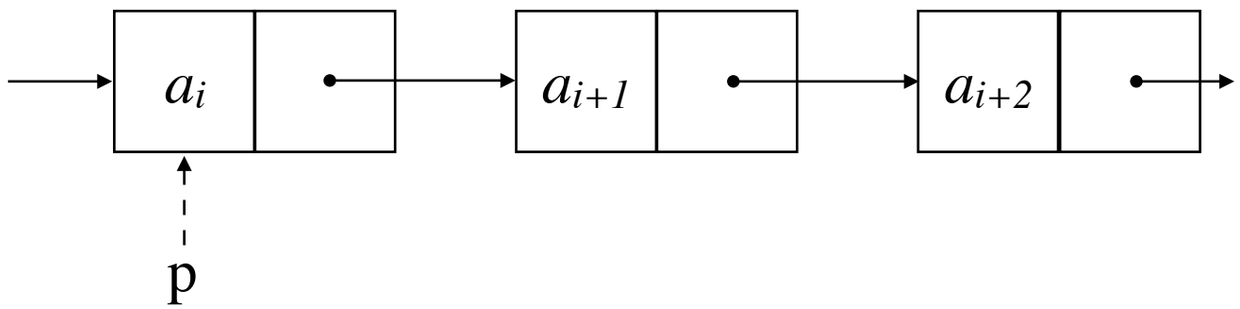
```
struct cell *insert(int x, struct cell *p, struct cell *init) {
    struct cell *q, *r;

    新しいセルを１つ作り，rとする.

    p->nextを一時的に保存
    p->nextがrを参照するようにする

    r->elementにxを代入
    r->nextがqを参照するようにする

    リストの先頭へのポインタinitを返す
}
```

# リスト：ポインタによる実現

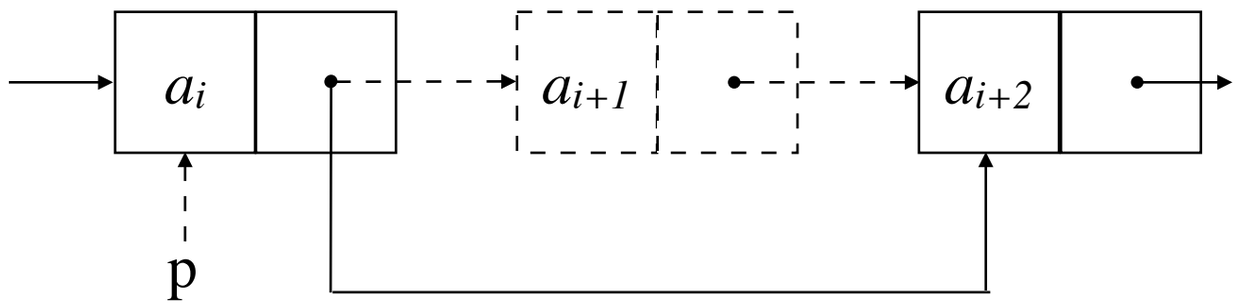
```
struct cell *insert(int x, struct cell *p, struct cell *init) {
    struct cell *q, *r;

    r = (struct cell *)malloc(sizeof(struct cell));
    if (p == NULL) {
        q = init;
        init = r;
    } else {
        q = p->next;
        p->next = r;
    }
    r->element = x;
    r->next = q;
    return(init);
}
```

# リスト：ポインタによる実現

・削除



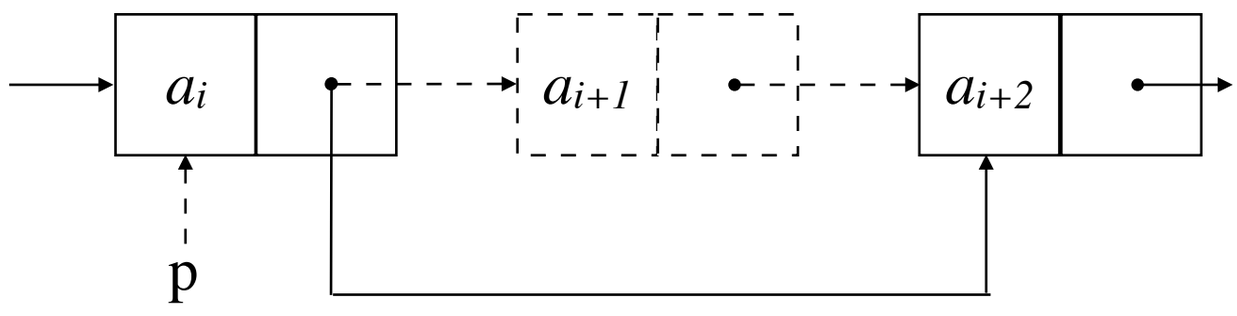

delete(p, L)



# リスト：ポインタによる実装

擬似コードによる段階的詳細化（第 1 段）

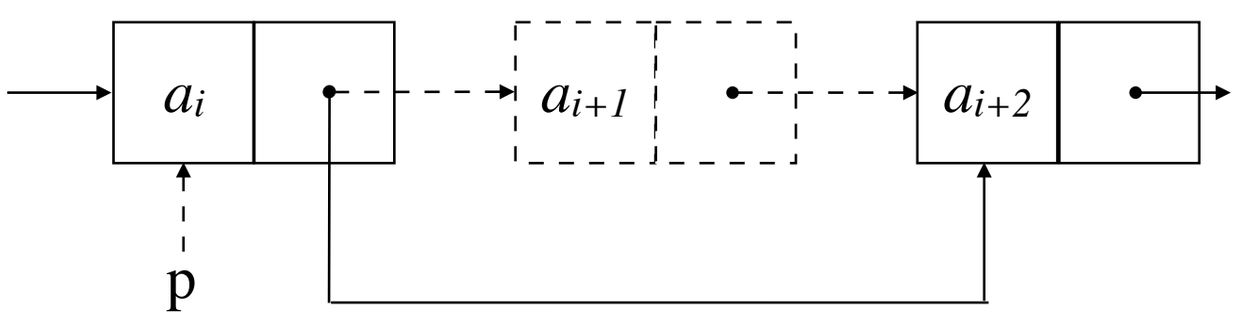
```
struct cell *delete(struct cell *p, struct cell *init) {
    一時的なポインタ変数qにp->nextを代入
    p->nextにq->nextを代入
    リストの先頭へのポインタを返す
}
```



# リスト：ポインタによる実装

擬似コードによる段階的詳細化（第２段）

```
struct cell *delete(struct cell *p, struct cell *init) {
    q = p->next;
    p->next = q->next;
    return(init);
}
```



# リスト：ポインタによる実装

```
struct cell *delete(struct cell *p, struct cell *init) {
    struct cell *q;
    if (init == NULL) {
        printf("error: list is empty.\n");
        exit(1);
    }
    if (p == NULL) {
        q = init;
        init = init->next;
        free(q);
    } else {
        if (p->next == NULL) {
            printf("error: no element to remove.\n");
            exit(1);
        } else {
            q = p->next;
            p->next = q->next;
            free(q);
        }
    }
    return(init);
}
```

# LIST API

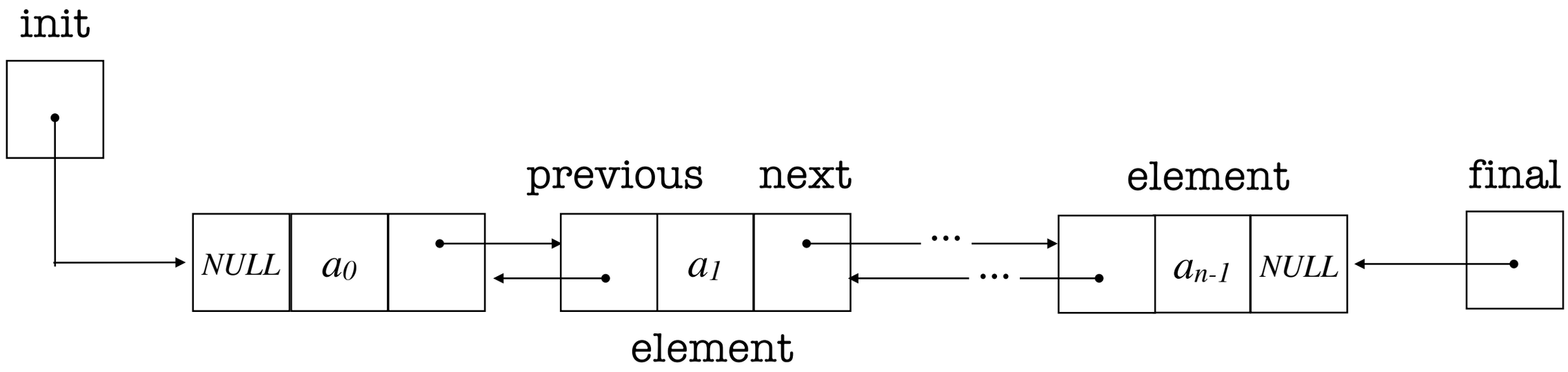
| | | |
|---|---|---|
| struct cell * | newlist() | 空リストを準備し、その先頭の位置を返す |
| struct cell * | insert(x, p, L) | リストLの位置pの次に要素xを挿入する |
| struct cell * | delete(p, L) | リストLの位置pの次の要素（もし存在すれば）を削除する |
| struct cell * | locate(x, L) | 要素xがL中に存在すればその位置（つまりそのセルを指すポインタ）を返す |
| int | retrieve(p, L) | 位置pのセルの内容(element部)を返す |
| int | find(i, L) | Lのi番目のセルの内容を返す |
| struct cell * | top(L) | Lの最初の位置を返す |
| struct cell * | last(L) | Lの最後の位置を返す |
| struct cell * | next(p, L) | 位置pの後のセルの位置を返す |
| struct cell * | previous(p, L) | 位置pの前のセルの位置を返す |

# リスト：実現法の比較

- 配列による実現
  - 挿入も削除もO(n)
  - ◯：一般的にはポインタより使用メモリが少ない
  - ×：リストがMAXSIZEを超えた時の処理が必要
- ポインタによる実現
  - 挿入も削除もO(1)
  - ◯：リストの長さが大きくなっても対応可能
  - ×：一般的には配列よりメモリ使用量が多い

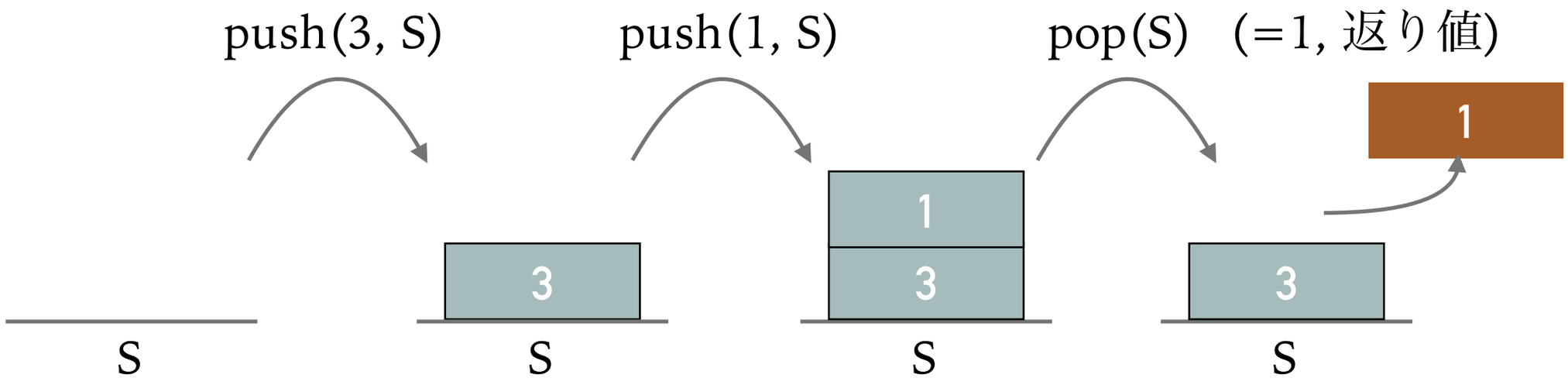
# 双方向リスト(DOUBLY-LINKED LIST)

```
struct cell {
    int element;
    struct cell *next;
    struct cell *previous;
};
```



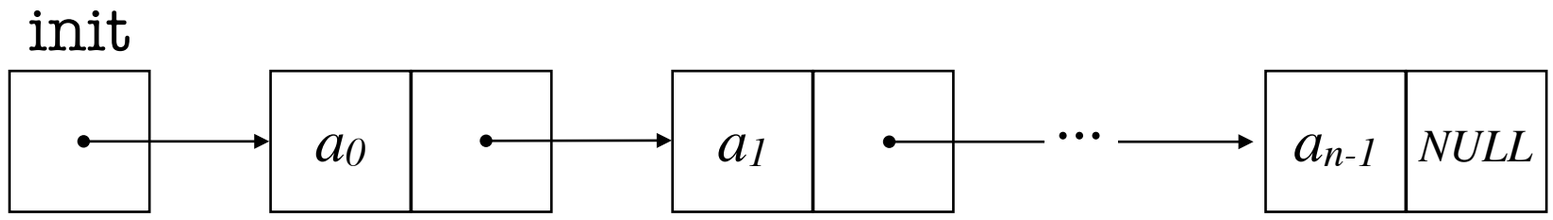
# スタック(STACK)

- 要素の挿入、削除がいつも先頭からなされるリスト
- LIFO(Last In Fast Out)
- 基本操作 (stack API)
  - 先頭に要素を挿入(push)、先頭から要素を削除(pop)、...



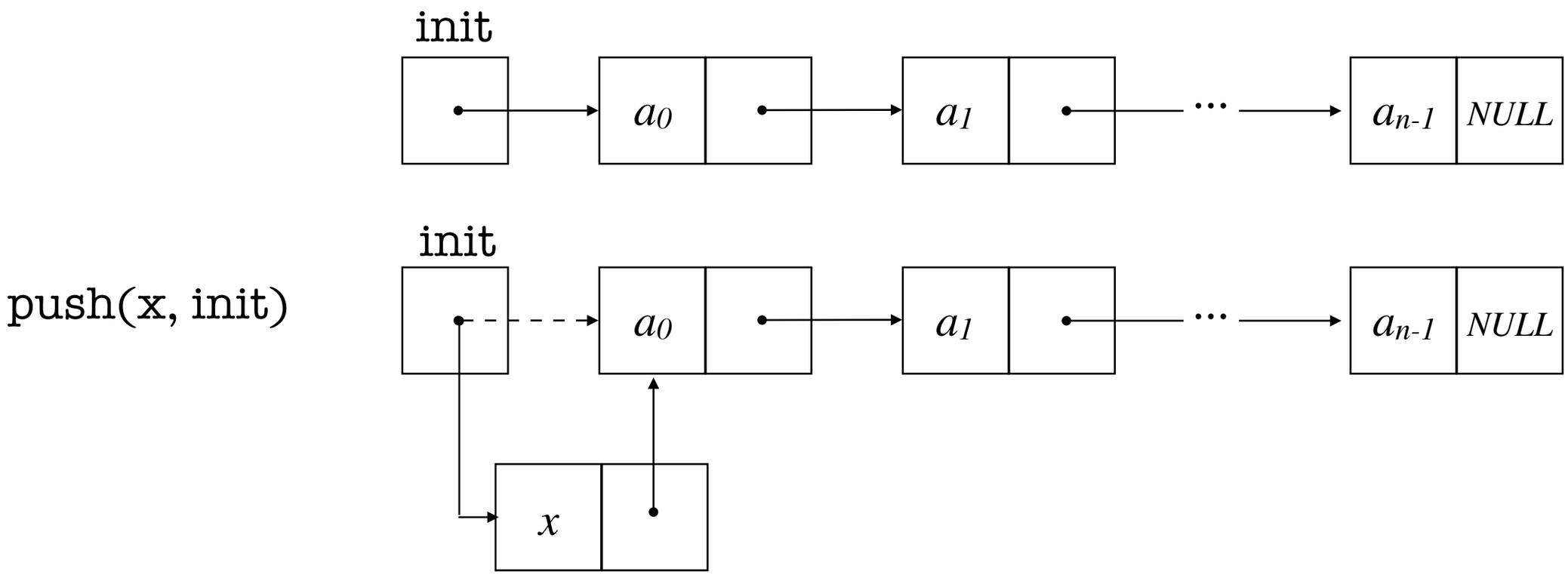
# スタック：リストによる実装

```
struct cell {
    int element;
    struct cell *next;
};
```



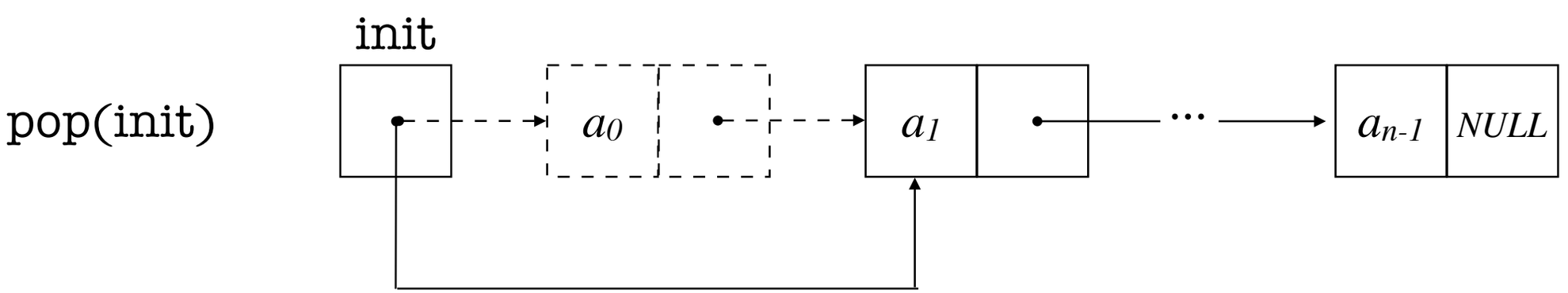
# スタック：リストによる実装

```
struct cell *push(int x, struct cell *init) {
    struct cell *q, *r;

    r = (struct cell *)malloc(sizeof(struct cell));
    q = init;
    init = r;
    r->element = x;
    r->next = q;
    return(init);
}
```



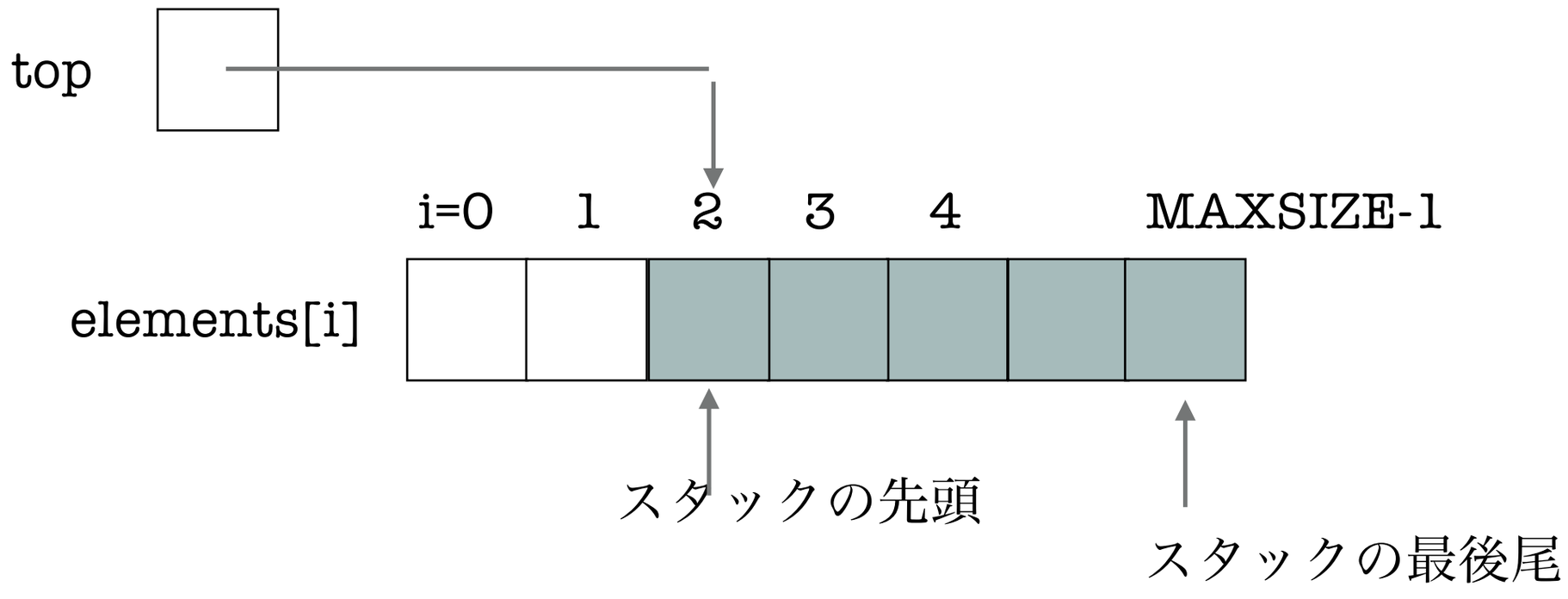
# スタック：リストによる実装

```
struct cell *pop(struct cell *init) {
    struct cell *q;

    if (init != NULL)  {
        q = init;
        init = init->next;
        free(q);
        return(init);
    } else {
        printf("error: stack is empty.\n");
        exit(1);
    }
    return;
}
```



# スタック：配列による実装

```
struct stack {
    int top;
    int element[MAXSIZE]
};
```



# スタック：配列による実装

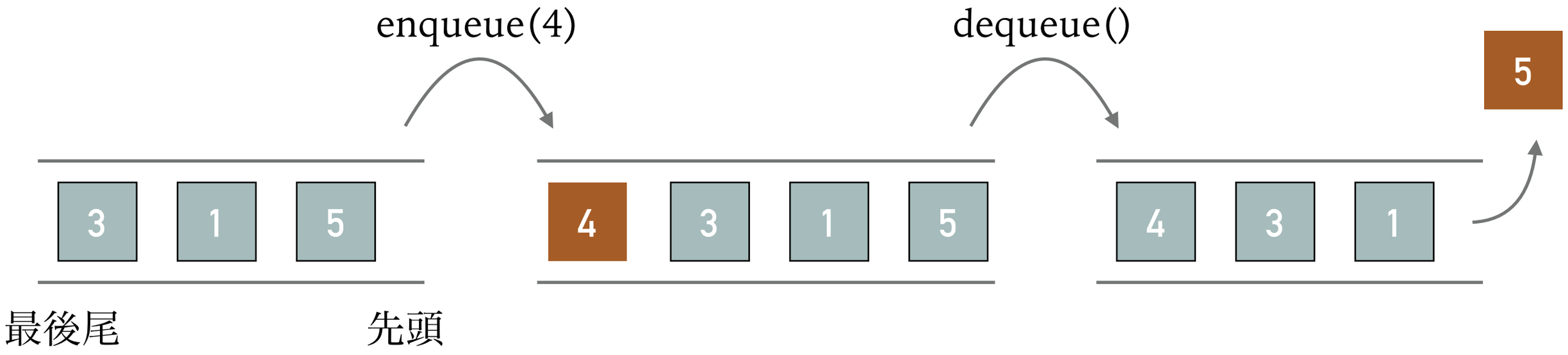
```
void push(int x, struct stack *s) {
    if (s->top >= MAXSIZE || s->top < 0)
        s->top = MAXSIZE;
    if (s->top == 0) {
        printf("error: stack is full.\n");
        exit(1);
    } else {
        s->top = s->top-1;
        s->element[s->top] = x;
    }
    return;
}
```

# スタック：配列による実装

```
void pop(struct stack *s) {
    if (s->top < MAXSIZE) {
        s->top = s->top+1;
    } else {
        printf("error: stack is empty.\n");
        exit(1);
    }
    return;
}
```

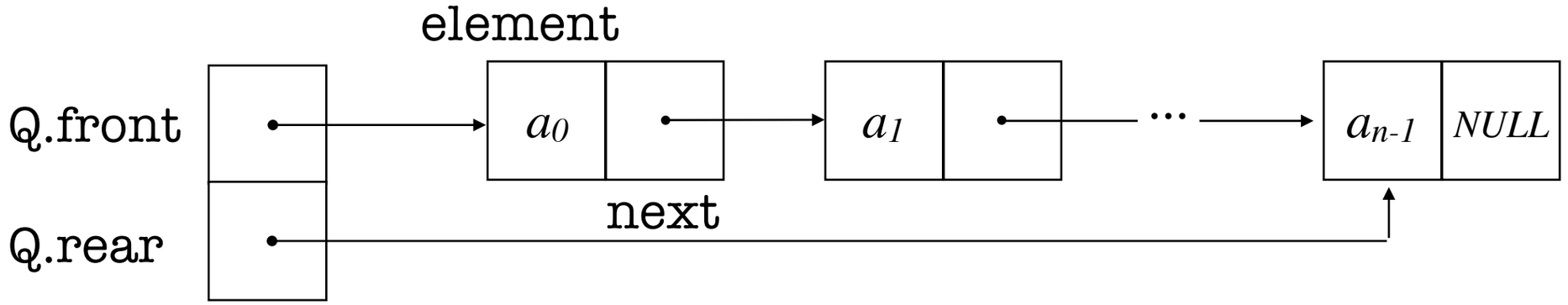
# 待ち行列（QUEUE）

- 要素の挿入は先頭から，取り出しは最後から
- FIFO: First In First Out
- 基本的操作 (queue API)
  - 最後尾に要素を追加(enqueue)，先頭から要素を取り出し(dequeue)



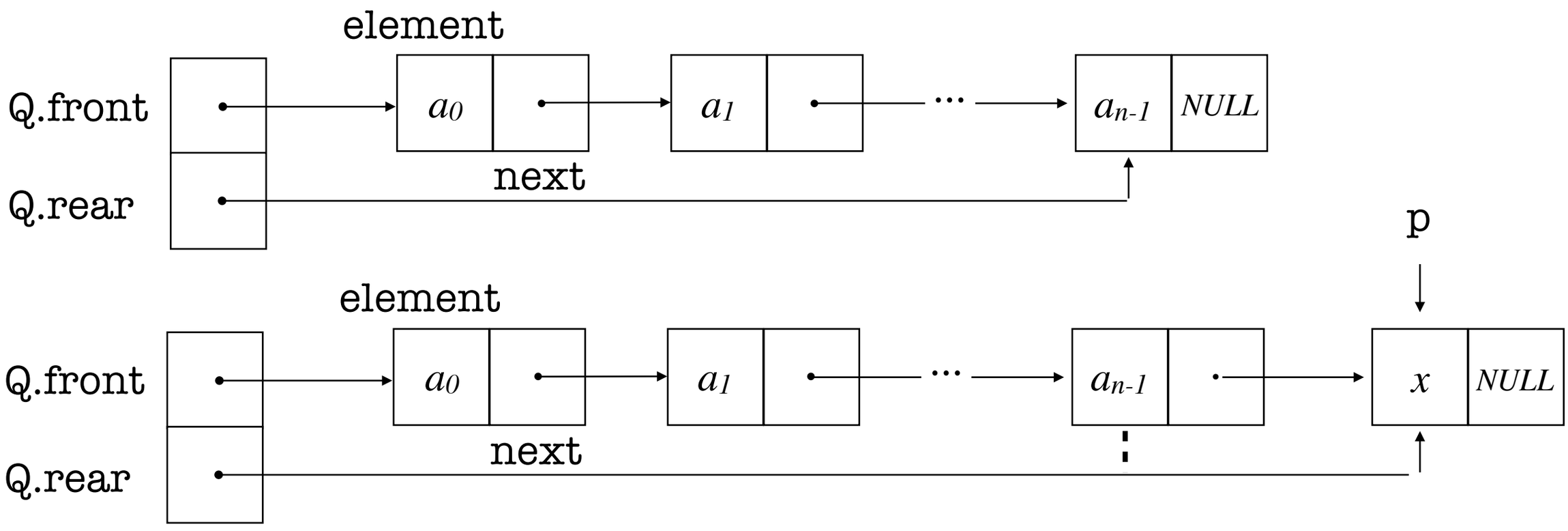
# キュー：リストによる実装

```
struct queue {
    struct cell *front;
    struct cell *rear;
};
```



# キュー：リストによる実装

```
void enqueue(int x, struct queue *q) {
    セルを１個作ってpで参照
    q->rear->nextがpを参照
    q->rearがpを参照
    q->rear->elementにxを代入
    q->rear->nextにNULLを代入
}
```

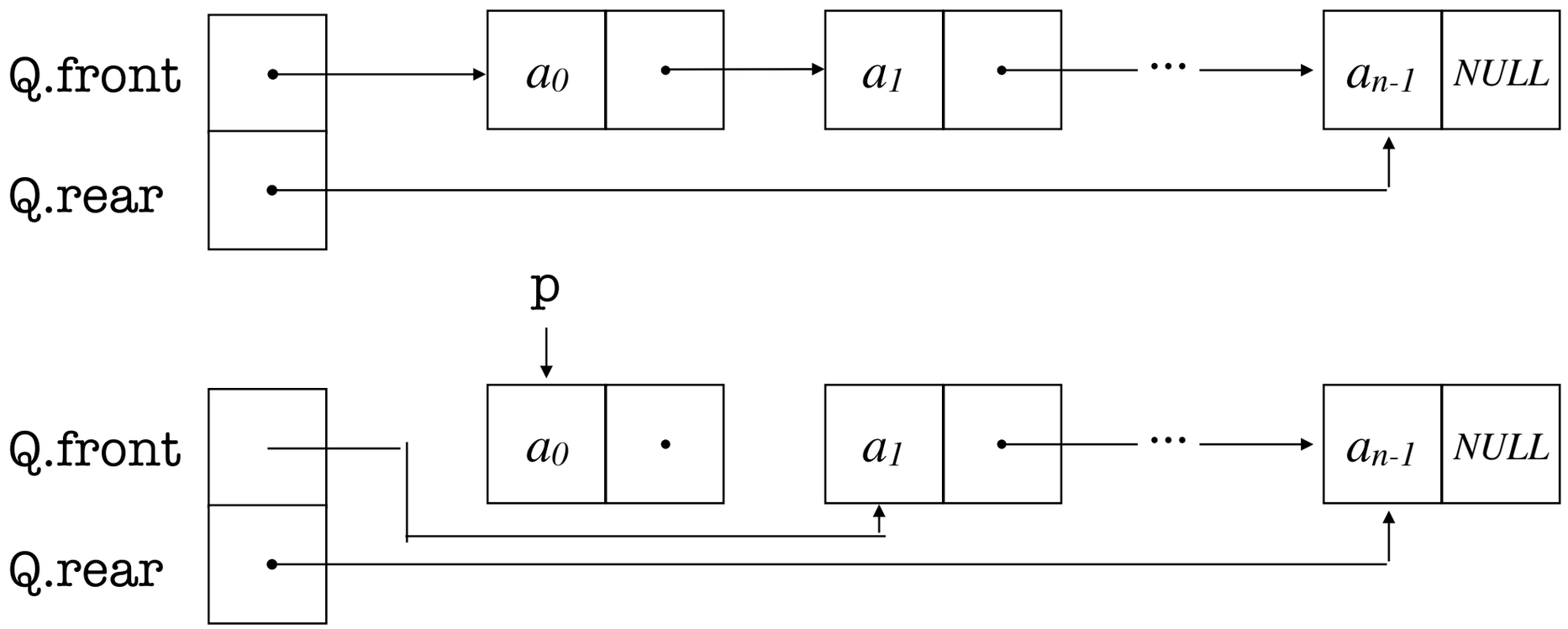


# キュー：リストによる実装

```
void enqueue(int x, struct queue *q) {
    struct cell *p;

    p = (struct cell *)malloc(sizeof(struct cell));
    if (q->rear != NULL)
        q->rear-next = p;
    q->rear = p;
    if (q->front == NULL)
        q->front = p;
    q->rear->element = x;
    q->rear->next = NULL;
    return;
}
```

# キュー：リストによる実装

```
void dequeue(struct queue *q) {
    q->frontが参照するセルをpに参照させる

    q->front->nextをq->frontに代入

}
```



# キュー：リストによる実装

```
void dequeue(struct queue *q) {
    struct cell *p;
    int x;

    if (q->front == NULL) {
        printf("error: queue is empty.\n");
        exit(1);
    } else {
        p = q->front;
        x = p->element;
        q->front = q->front->next;
        free(p);
    }
    if (q->front == NULL)
        q->rear = NULL;
    return;
}
```

# キュー：配列による実装

```
struct queue {
    int elements[MAXSIZE];
    int head, tail, count;
};
```

# キュー：配列による実装

```
void enqueue(int x, struct queue *q) {
    if (q->count > MAXSIZE) {
        printf("error: queue is full.\n");
        exit(1);
    }
    elements[q->tail] = x;
    q->tail = q->tail+1;
    if (q->tail > MAXSIZE)
        q->tail = 0;
    q->count = q->count+1;
}
```

# キュー：配列による実装

```
int dequeue(struct queue *q) {
    int x;

    if (q->count <= 0) {
        printf("Error: Queue is empty.\n");
        exit(1);
    }
    x = q->elements[q->head];
    q->head = q->head+1;
    if (q->head >= MAXSIZE)
        q->head = 0;
    q->count = q->count-1;
    return x;
}
```

# 課題

- ポインタを用いたリストのプログラムを書いて，関数 insert()，delete()が正しく動くことを示しなさい．もし正しく動かない場合は，どこが正しくないか述べなさい．
- 上記ポインタを用いたリストに以下の関数を追加しなさい．
  - printlist(l): 要素を先頭から順に出力する．
  - find(p, l): lのpの指すセルの内容を返す．
  - next(p, l), previous(p, l): それぞれpの指すセルの後および前のセルの位置を返す．