# C++プログラミングIII

第5回 キュー

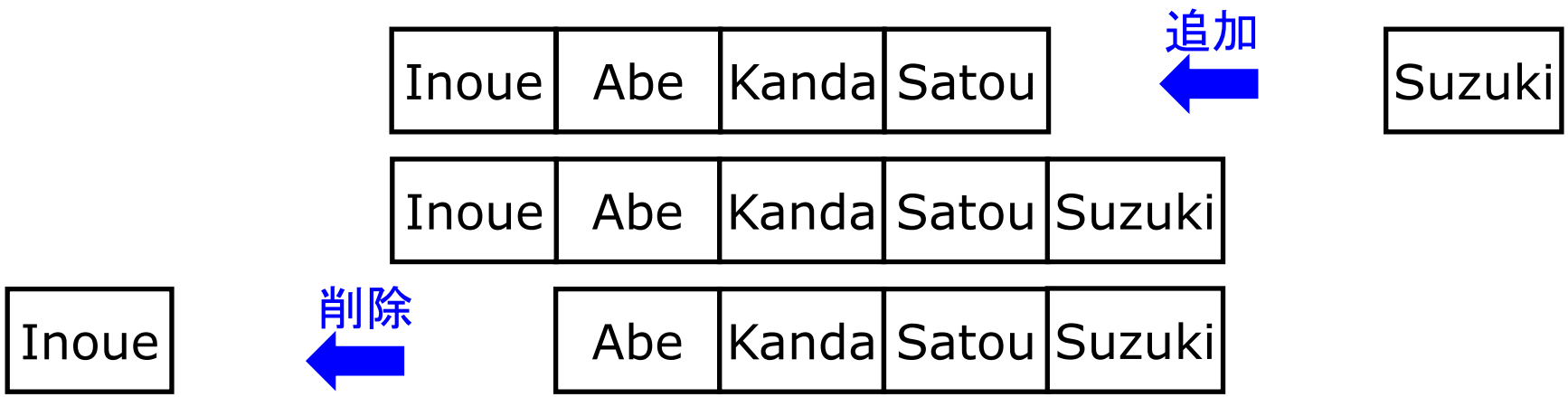
# 本日のお話

1. キューの概要
2. キューを使ったプログラムの作成
3. 連結リストを用いたキューの実装

# キュー（queue）とは

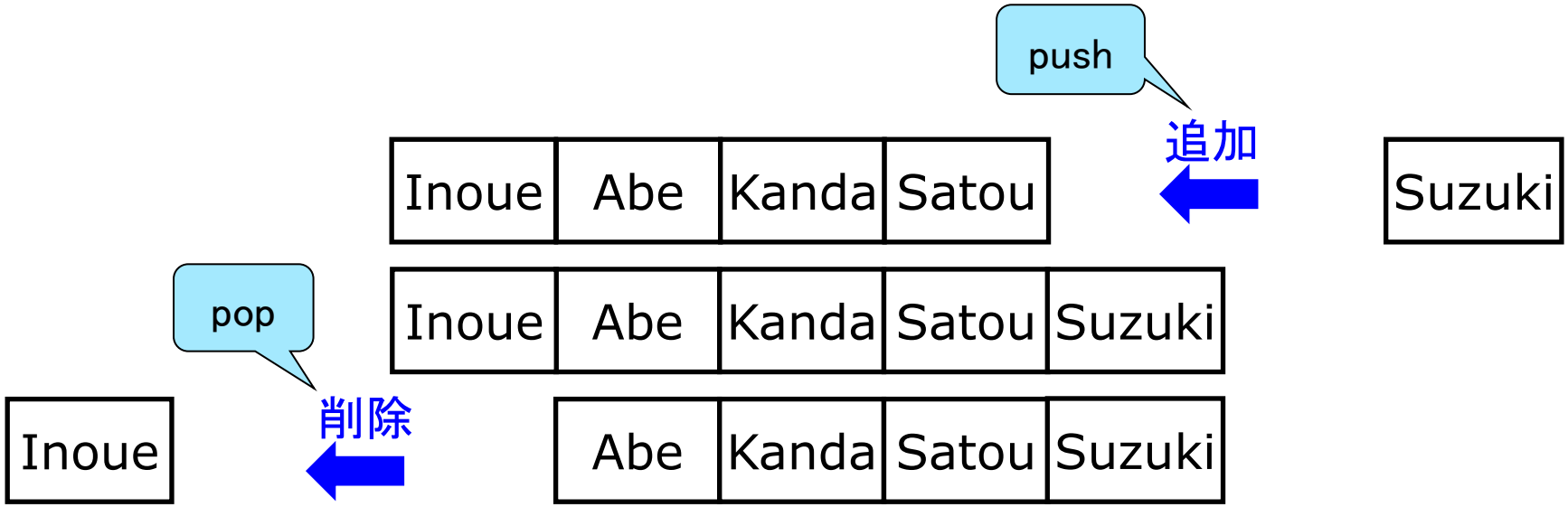
- queue：「待ち行列」
  - 要素を一列に並べたデータ構造（リストの一種）
- 代表的なキュー：
  - 「先入れ，先出し」
    “First In First Out”： FIFO
  - 見ることができるのは先頭（front）のデータのみ

- 参考：スタック（stack）
  「後入れ，先出し」
  “Last In First Out”：
  LIFO



# キューに対する操作

- push（enqueue）
  - データを末尾に追加する
- pop（dequeue）
  - 先頭のデータを取り出す



# 本日のお話

1. キューの概要
2. キューを使ったプログラムの作成
3. 連結リストを用いたキューの実装

# キューを操作するプログラム

```
#include <iostream>
#include <queue>
using namespace std;

void print_queue(queue<string>& q){
    size_t n{ q.size() };
    cout << "num: " << n << ", data:";
    for (int i = 0; i < n; i++) {
        string s = q.front();
        cout << " " << s;
        q.pop();
        q.push(s);
    }
    cout << endl;
}
```

# キューを操作するプログラム

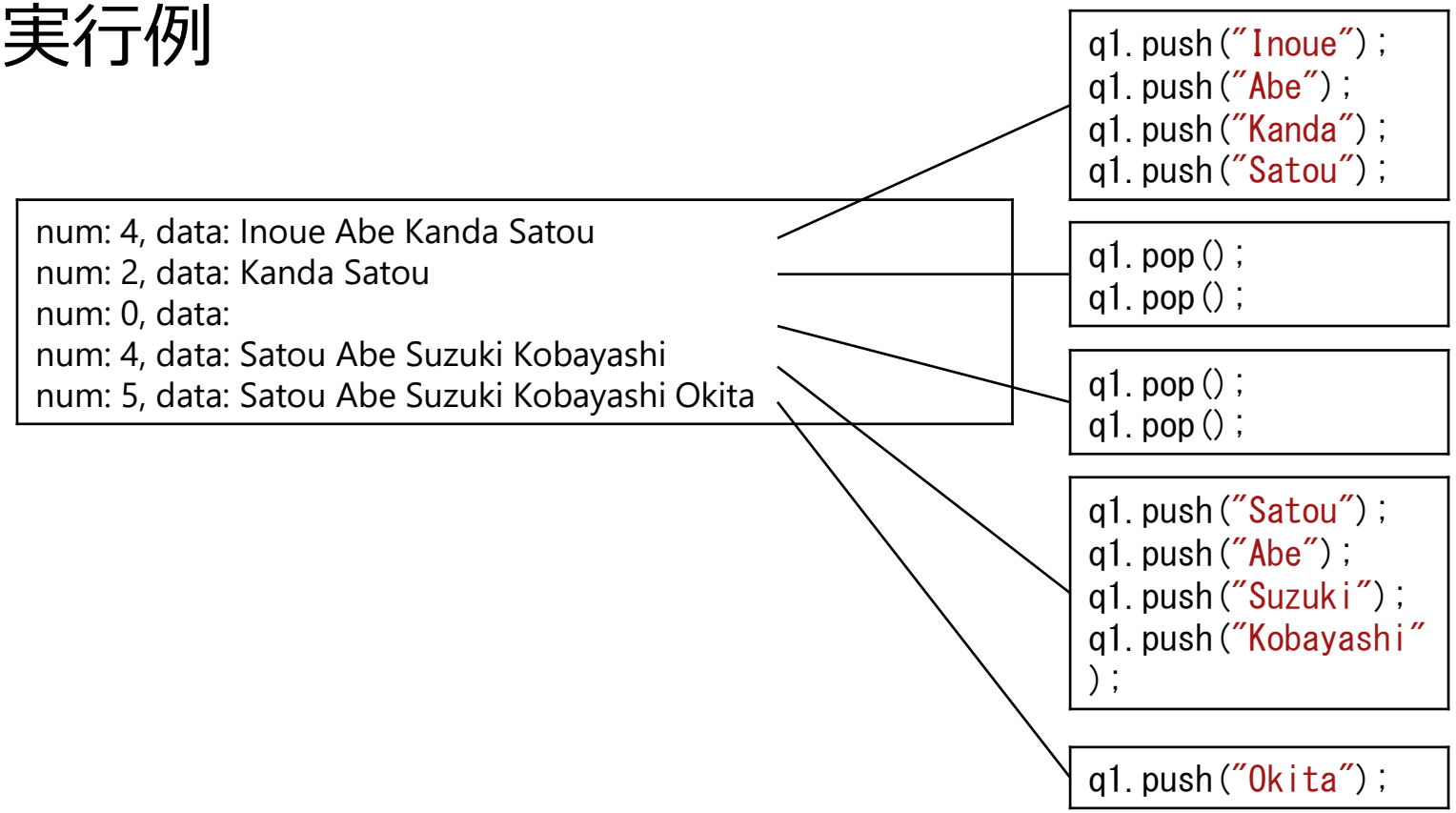
```
int main(){
    queue<string> q1;

    q1.push("Inoue"); q1.push("Abe");
    q1.push("Kanda"); q1.push("Satou");
    print_queue(q1);

    q1.pop(); q1.pop();
    print_queue(q1);

    q1.pop(); q1.pop();
    print_queue(q1);

    q1.push("Satou");  q1.push("Abe");
    q1.push("Suzuki"); q1.push("Kobayashi");
    print_queue(q1);

    q1.push("Okita");
    print_queue(q1);

    return 0;
}
```

標準ライブラリのqueueを利用したプログラム例
queue<string> ： string型のデータを扱うキュー
push(x)： 値がxのデータをpushする
size()： キューの要素数を返す
empty()： キューの要素数が0のときtrueを返す
front()： キューの先頭の要素の値を返す
pop()： キューをpopする

print_queue(q)： キューの全データを表示する

# キューを操作するプログラム

実行例

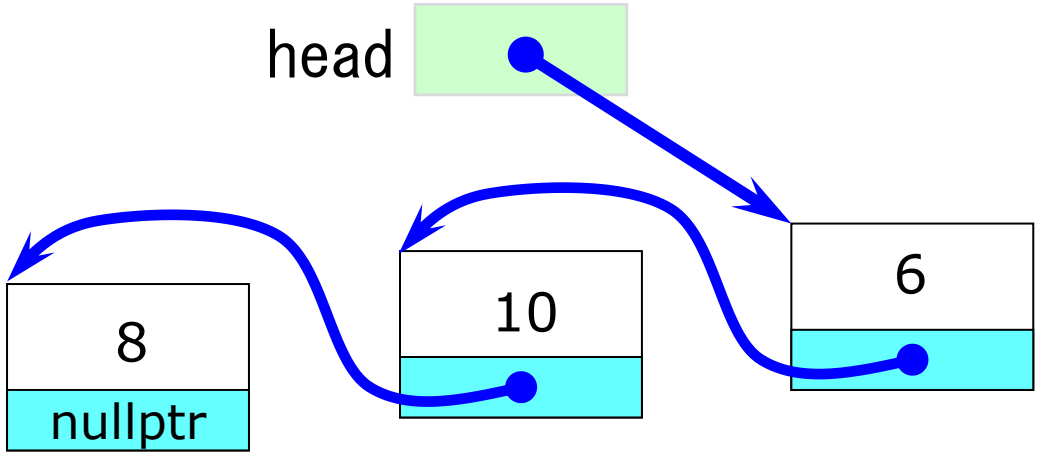


# 本日のお話

1. キューの概要
2. キューを使ったプログラムの作成
3. 連結リストを用いたキューの実装

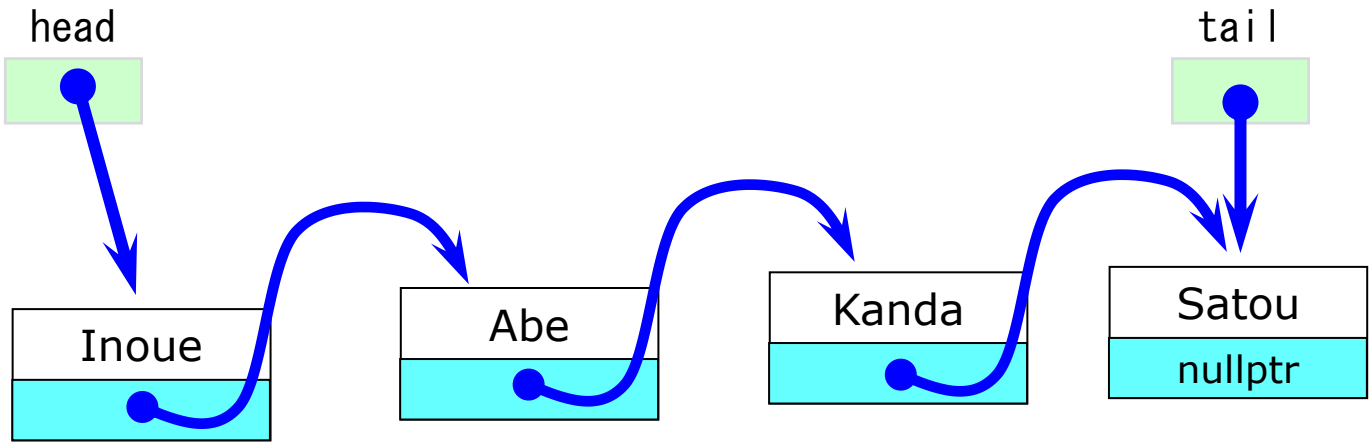
# 連結リスト

- 一連のノードが，データの値と次のデータへのリンクを持つデータ構造
- 各データにはノードのリンクを順に辿ることでアクセスできる
- 単方向リストは連結リストの一種



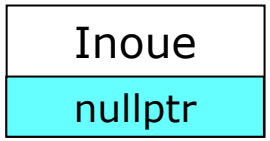
# キューの実装に用いる連結リスト

- head： 先頭のノードへのリンク
- tail： 末尾のノードへのリンク
- ノード間は単方向のリンク



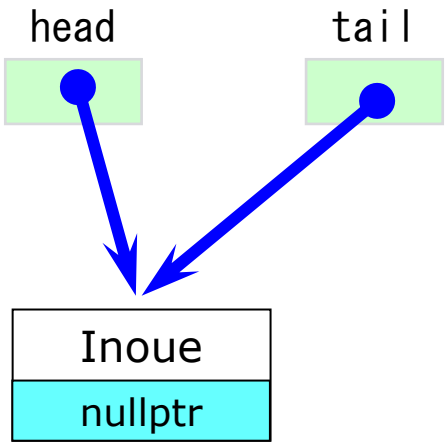
# 連結リストに対する操作①

- 末尾にデータを挿入する
  1. 挿入するオブジェクトを作成する

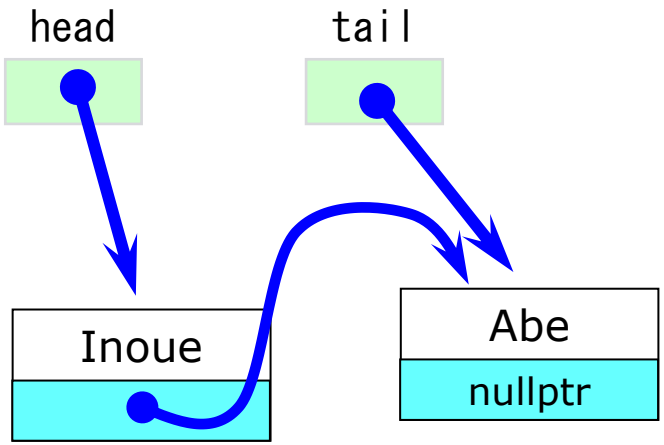
# 連結リストに対する操作②

- 末尾にデータを挿入する
  1. 挿入するオブジェクトを作成する
  2. キューに要素がない場合はheadとtailが同じノードを指すようにする



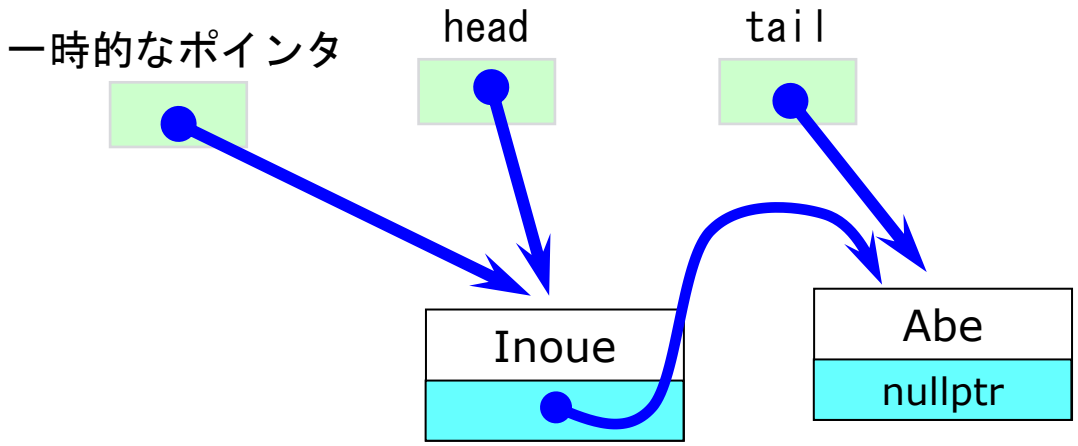
# 連結リストに対する操作③

- 末尾にデータを挿入する
  1. 挿入するオブジェクトを作成する
  2. キューに要素がない場合はheadとtailが挿入するオブジェクトを指すようにする
  3. そうでない場合は，tailとそれまでのtailのnextが挿入するオブジェクトを指すようにする



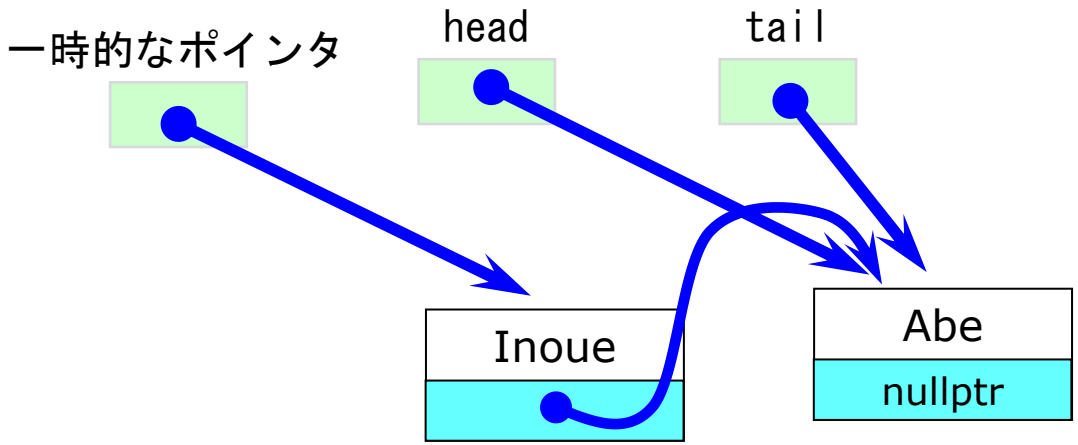
# 連結リストに対する操作④

- 先頭からデータを削除する
  1. headを一時的なポインタに代入する



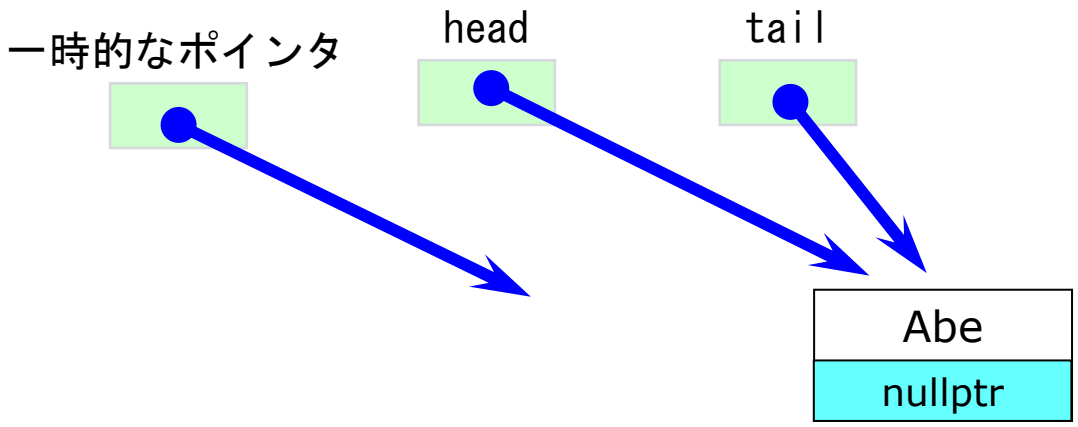
# 連結リストに対する操作⑤

- 先頭からデータを削除する
  1. headを一時的なポインタに代入する
  2. headが先頭のオブジェクトのnextを指すようにする



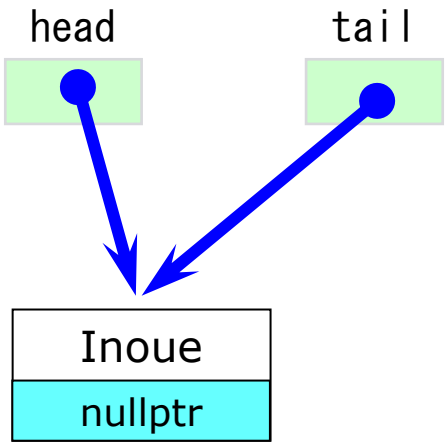
# 連結リストに対する操作⑥

- 先頭からデータを削除する
  1. headを一時的なポインタに代入する
  2. headが先頭のオブジェクトのnextを指すようにする
  3. 一時的なポインタを使ってオブジェクトを削除する



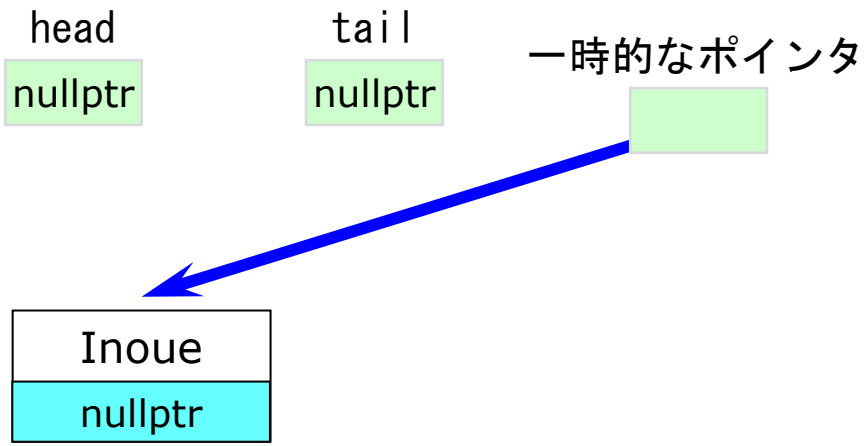
# 連結リストに対する操作⑦

- 先頭からデータを削除する
  1. headを一時的なポインタに代入する
  2. headが先頭のオブジェクトのnextを指すようにする
  3. 一時的なポインタを使ってオブジェクトを削除する
  4. 削除する要素しかない場合
     （headのnextがnullptrの場合），tailもnullptrにする



# 連結リストに対する操作⑧

- 先頭からデータを削除する
  1. headを一時的なポインタに代入する
  2. headが先頭のオブジェクトのnextを指すようにする
  3. 一時的なポインタを使ってオブジェクトを削除する
  4. 削除する要素しかない場合
     （headのnextがnullptrの場合），tailもnullptrにする



# 連結リストを用いたキューの実装

```
class Queue {
    Node* head;                                  // キューの先頭
    Node* tail;                                  // キューの末尾
public:
    Queue() { head = nullptr; tail = nullptr; }     // コンストラクタ
    ~Queue() { while (!empty()) pop(); }        // デストラクタ

    void push(string s);   // 値sを持つノードをキューの末尾に追加
    void pop();                          // キューの先頭のノードを削除する
    string front() const;  // キューの先頭のノードの値を返す

    bool empty() const { return head == nullptr; }    // キューが空かの判定
    int size() const;
};
```

上記のpush (), front(), pop ()を
Nodeクラスを利用してどのように作るか？

# 連結リストを用いたキューの実装

```
int Queue::size() const {        // キューの要素数を返す

    int count = 0;       // 要素数をカウント
    Node* node = head;   // 最初はキュー先頭のノードを指すようにする

    while (node != nullptr) {    // nodeがnullptrでないかぎり
        count++;                 // ノードをカウントして
        node = node->getNext(); // 次のノードをnodeが指すようにする
    }

    return count;
}
```

# 連結リストを用いたキューの実装

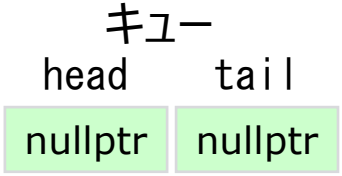
## 演習問題

```
// 値xを持つノードをキューの末尾に追加する
void Queue::push(string s)
{

}

// キューの先頭のノードの値を返す
string Queue::front()
{

}

// キューの先頭のノードを削除し，nextを新しい先頭にする
void Queue::pop()
{

}
```

# 連結リストを用いたキューの実装

実行例

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```




```
q1.push("Inoue");
q1.push("Abe");
q1.push("Kanda");
q1.push("Satou");
```

```
q1.pop();
q1.pop();
```

```
q1.pop();
q1.pop();
```

```
q1.push("Satou");
q1.push("Abe");
q1.push("Suzuki");
q1.push("Kobayashi");
```

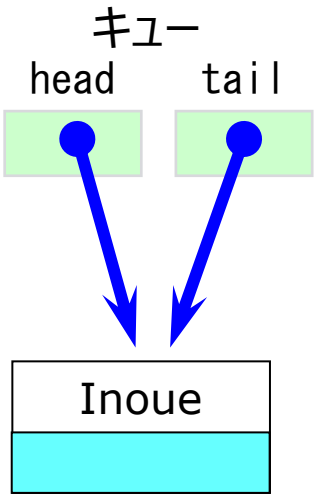
```
q1.push("Okita");
```

# 連結リストを用いたキューの実装

実行例

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

```
q1.push("Inoue");
q1.push("Abe");
q1.push("Kanda");
q1.push("Satou");
```

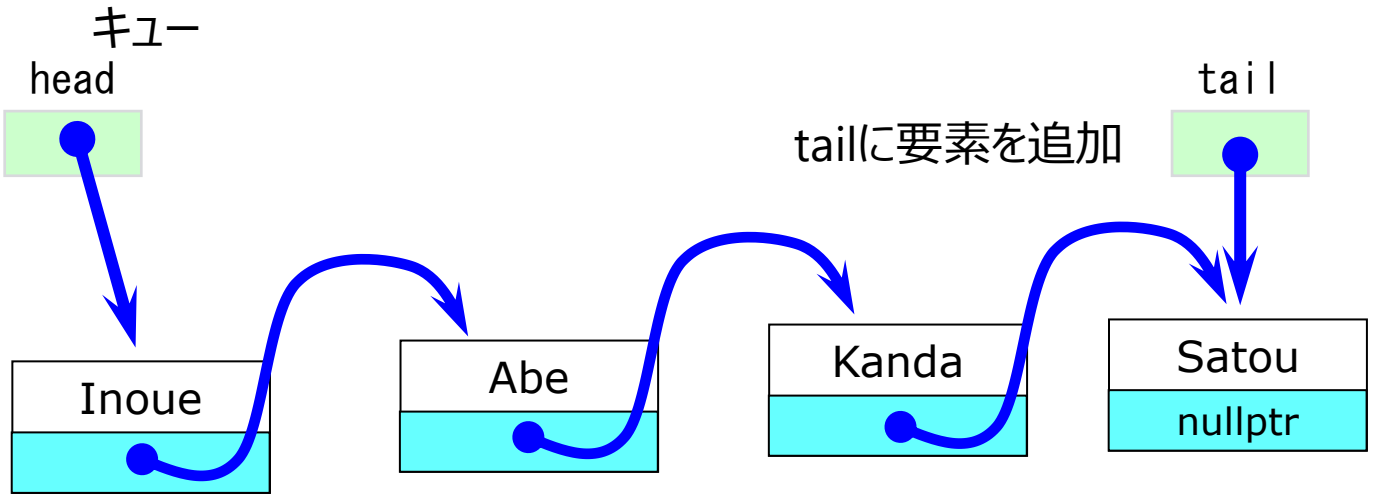


# 連結リストを用いたキューの実装

実行例

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

```
q1.push("Inoue");
q1.push("Abe");
q1.push("Kanda");
q1.push("Satou");
```

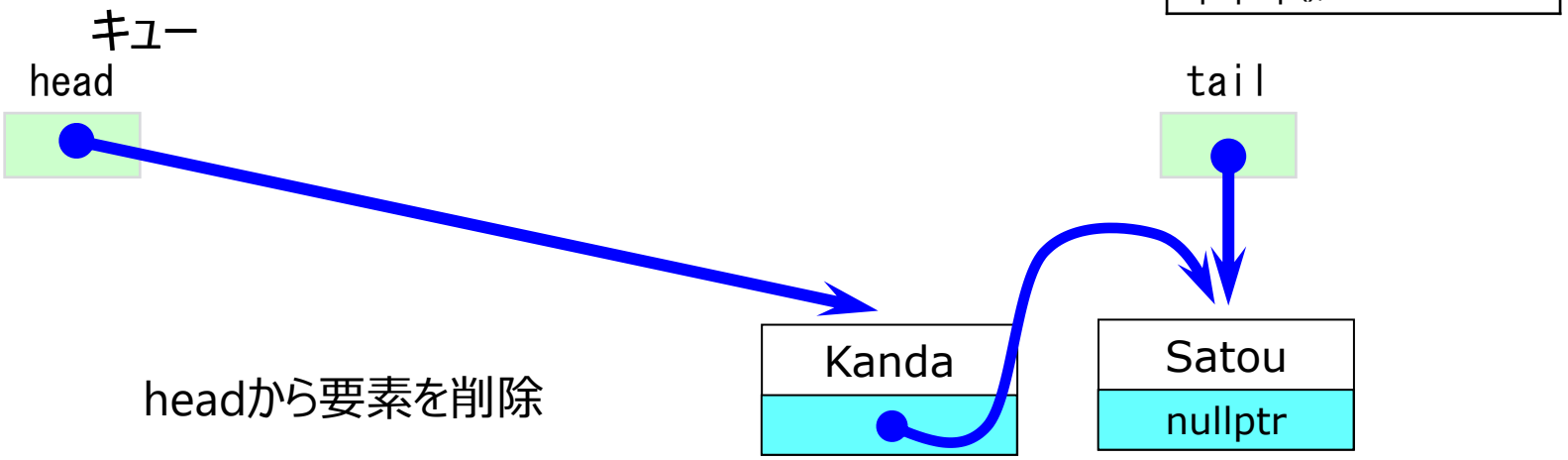


# 連結リストを用いたキューの実装

実行例

num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita

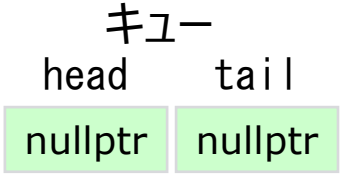
q1.pop();
q1.pop();



# 連結リストを用いたキューの実装

実行例

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```




```
q1.pop();
q1.pop();
```

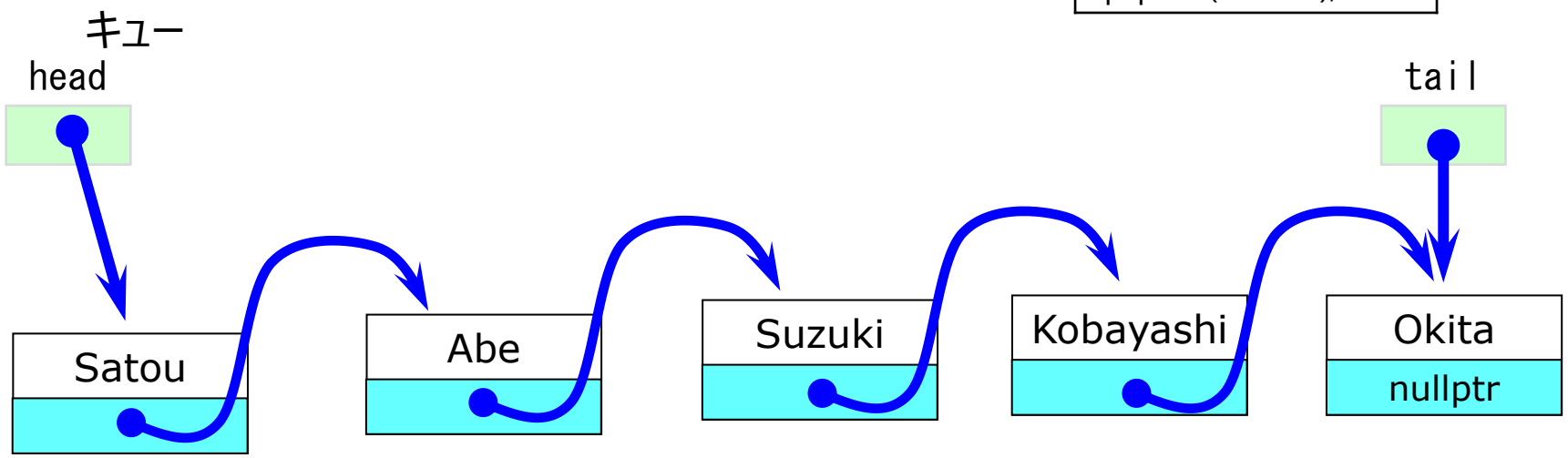
# 連結リストを用いたキューの実装

実行例

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

```
q1.push("Satou");
q1.push("Abe");
q1.push("Suzuki");
q1.push("Kobayashi");
```

```
q1.push("Okita");
```



# 連結リストを用いたキュー設計のポイント

- キューに出し入れするデータのデータ構造のクラスを定義する
  （今回の例ではNodeクラス）
  - 格納したいデータのメンバに加えて，連結リスト用のポインタを準備
  - 各データメンバへのアクセス用関数を作成する

- キュー自体のクラスを定義する
  （今回の例ではQueueクラス）
  - キュー用の基本的な関数push()，front()，pop()を，
    キューに出し入れするデータのデータ構造を用いて作成する
  - キューの情報を取得する関数empty()，size()を作成する

# 本日はここまで

- お疲れさまでした

# 動的割り付けのお話

1. キューの概要
2. キューを使ったプログラムの作成
3. 連結リストを用いたキューの実装

4. 環状配列を用いたキューの実装

# Queue

- データ構造
  - キューの特性をどのように実現するか？
    - キューがキューたるに必要な何か
    - **先頭**
    - 最後尾
    - **データ長（キューにあるデータの個数：可変）**
  - 2通りの実装方法が存在

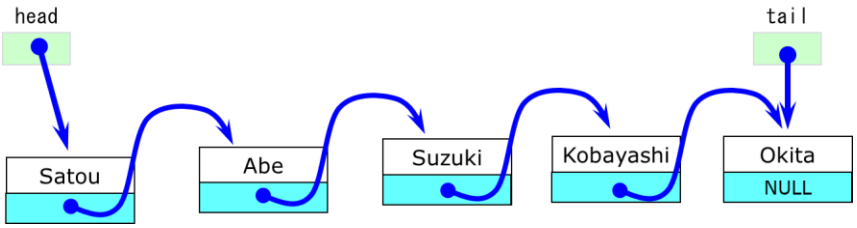



図１．連結リストを用いた実装

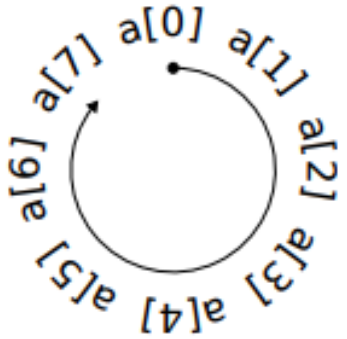
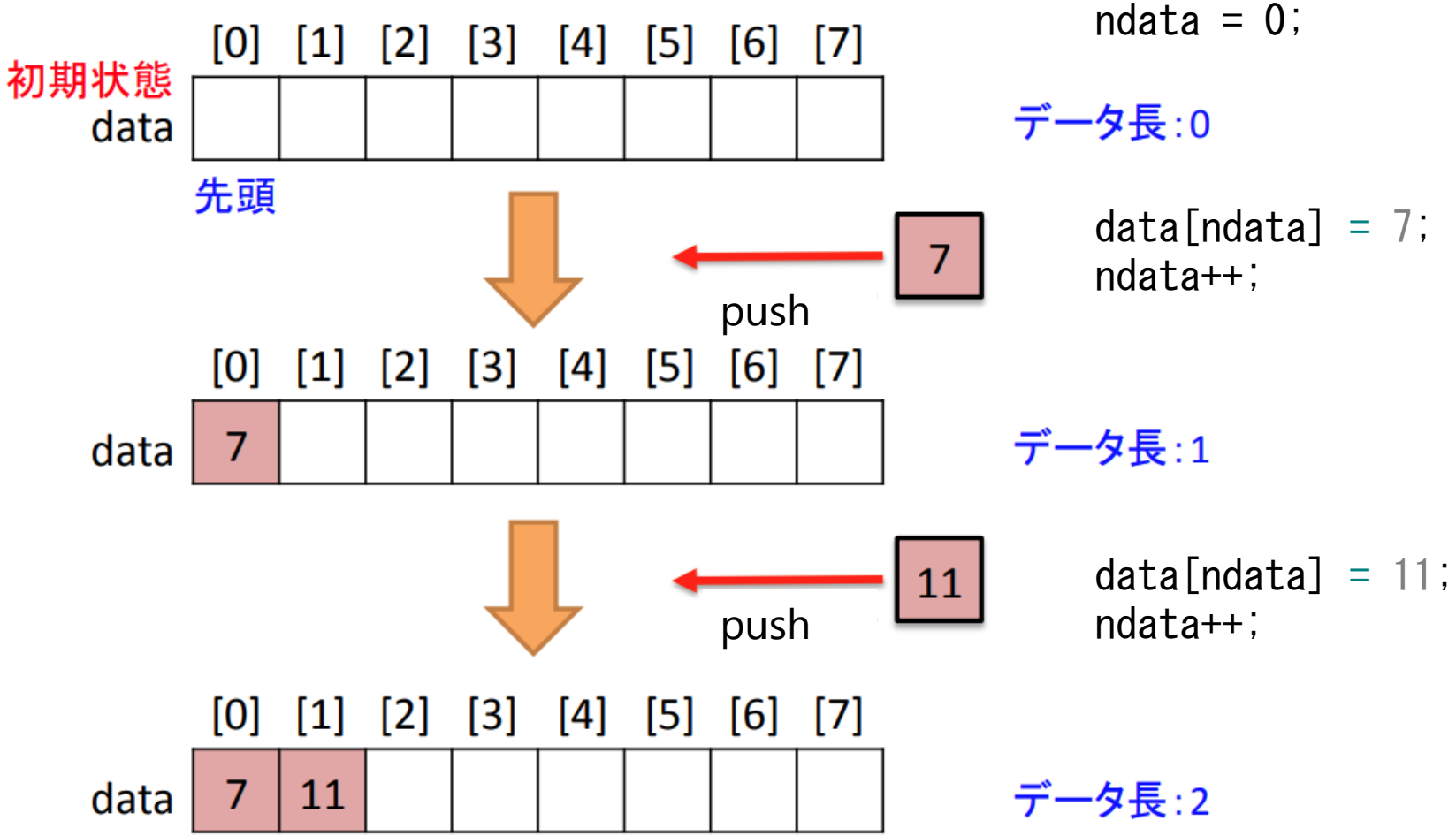
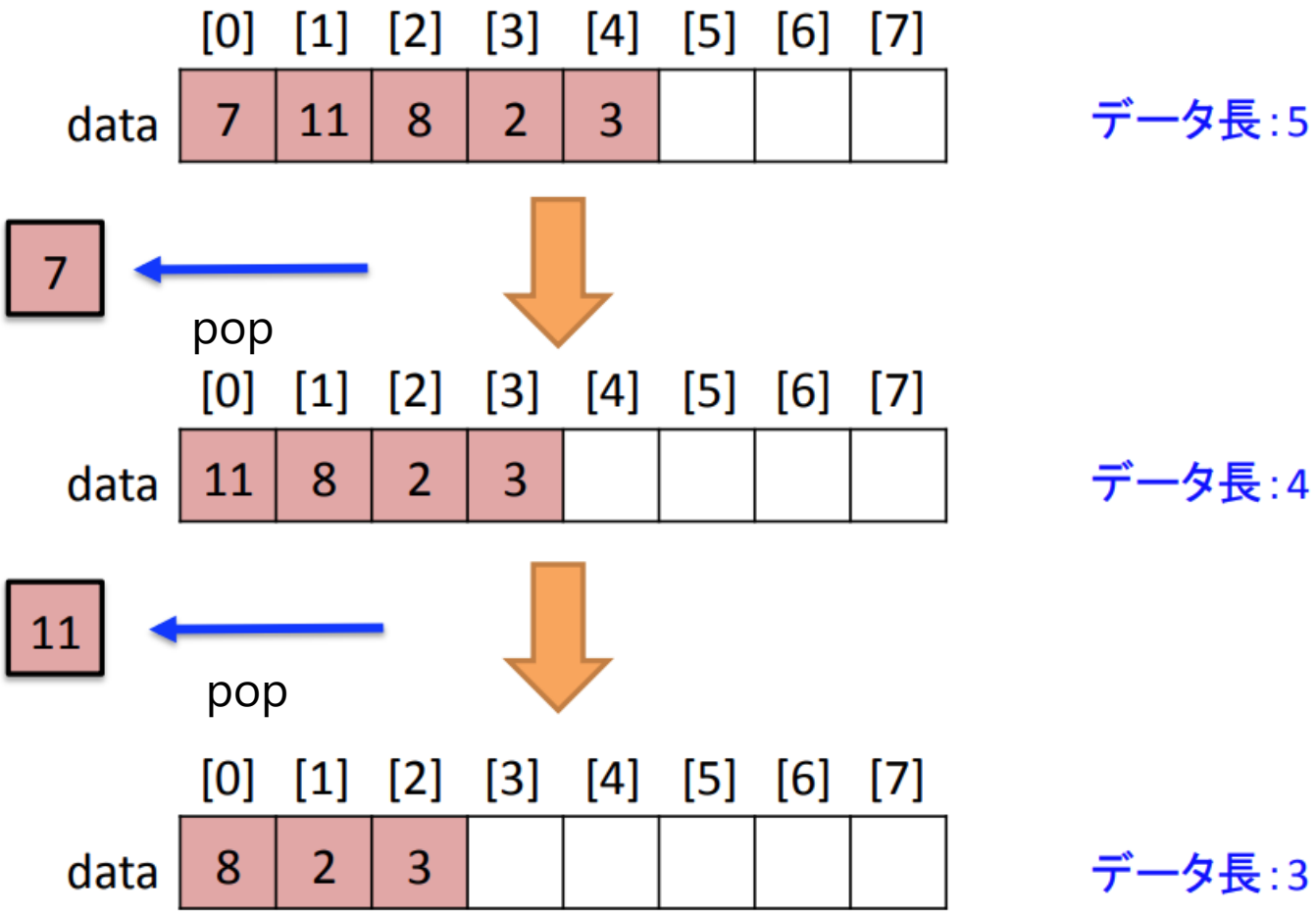



図２．環状配列を用いた実装

# 配列による実現（入力）


初期状態
data
[0] [1] [2] [3] [4] [5] [6] [7]
ndata = 0;
データ長:0
先頭
7
push
data[ndata] = 7;
ndata++;
data
7
データ長:1
11
push
data[ndata] = 11;
ndata++;
data
7
11
データ長:2

# 配列による実現（出力）


[0] [1] [2] [3] [4] [5] [6] [7]
data 7 11 8 2 3
データ長：5
7
pop
[0] [1] [2] [3] [4] [5] [6] [7]
data 11 8 2 3
データ長：4
11
pop
[0] [1] [2] [3] [4] [5] [6] [7]
data 8 2 3
データ長：3

# 固定長配列を用いたキューの実装

- 最も初歩的な実装の例

```
#include <iostream>
using namespace std;

const size_t CAPACITY{ 5 };

class Queue {
    string ar[CAPACITY];// 大きさCAPACITYの配列
    size_t cursize{ 0 };// 有効な要素数
public:
    ~Queue() { while (!empty()) pop(); }// デストラクタ
    void push(string);// 値sを持つ要素をキューの末尾に追加
    void pop();// キューの先頭の要素を削除する
    string front();// キューの先頭の要素の値を返す
    bool empty() { return cursize == 0; }// キューが空かの判定
    size_t size() { return cursize; }// キューの要素数を返す
};
```

# 固定長配列を用いたキューの実装

```
// 配列の先頭の要素の値を返す
string Queue::front() {
    return ar[0];
}
// 値xの要素を配列の末尾に追加する
void Queue::push(string s) {
    ar[cursize] = s;
    ++cursize;
}
// 配列の要素を先頭の方向へシフトする
void Queue::pop() {
    if (cursize > 0) {
        cursize--;
        for (size_t i{ 0 }; i < cursize; i++)
            ar[i] = ar[i + 1];
    }
}
```

# 固定長配列を用いたキューの実装

実行例

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

キュー

```
q1.push("Inoue");
q1.push("Abe");
q1.push("Kanda");
q1.push("Satou");
```

```
q1.pop();
q1.pop();
```

```
q1.pop();
q1.pop();
```

```
q1.push("Satou");
q1.push("Abe");
q1.push("Suzuki");
q1.push("Kobayashi");
```

```
q1.push("Okita");
```

# 固定長配列を用いたキューの実装

実行例（push）

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

```
q1.push("Inoue");
q1.push("Abe");
q1.push("Kanda");
q1.push("Satou");
```

**キュー**

| | | | | |
|---|---|---|---|---|
| **Inoue** | **Abe** | **Kanda** | **Satou** | |

```
// 値xの要素を配列の末尾に追加する
void Queue::push(string s) {
    ar[cursize] = s;
    ++cursize;
}
```

# 固定長配列を用いたキューの実装

## 実行例（pop）

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

```
q1.pop();
q1.pop();
```

キュー

| | | | | |
|---|---|---|---|---|
| Inoue | Abe | Kanda | Satou | |

要素をシフト


| | | | | |
|---|---|---|---|---|
| **Kanda** | **Satou** | | | |

```
// 配列の要素を先頭の方向へシフトする
void Queue::pop() {
    if (cursize > 0) {
        cursize--;
        for (size_t i{ 0 }; i < cursize; i++)
            ar[i] = ar[i + 1];
    }
}
```

# 固定長配列を用いたキューの実装

実行例（pop）

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

キュー

```
q1.pop();
q1.pop();
```

# 固定長配列を用いたキューの実装

実行例（push&終了）

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

キュー

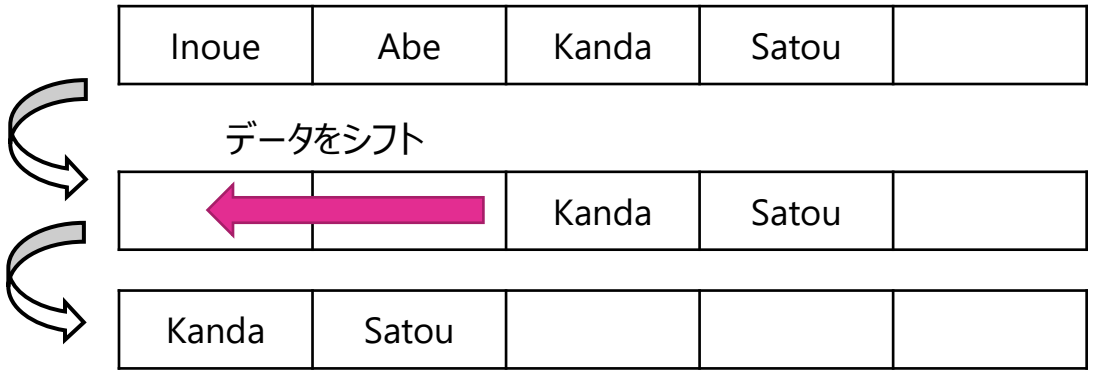
| Satou | Abe | Suzuki | Kobayashi | Okita |
|---|---|---|---|---|

```
q1.push("Satou");
q1.push("Abe");
q1.push("Suzuki");
q1.push("Kobayashi");
```

```
q1.push("Okita");
```

# 固定長配列を用いたキューの問題点

- データを取り出すたびに配列の要素をシフトするのは効率が悪い

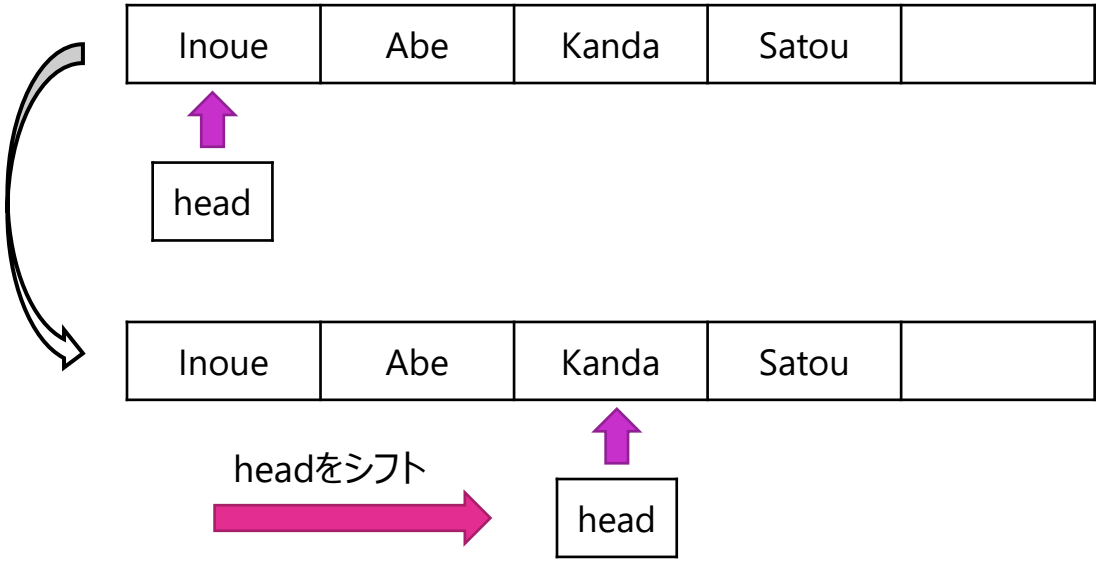



```
q1.pop();
q1.pop();
```

```
// 配列の要素を先頭の方向へシフトする
void Queue::pop() {
    if (cursize > 0) {
        cursize--;
        for (size_t i{ 0 }; i < cursize; i++)
            ar[i] = ar[i + 1];
    }
}
```

# 環状配列を用いたキューの実装

- 配列の要素をシフトするのは効率が悪いので，先頭（head）をずらす




```
q1.pop();
q1.pop();
```

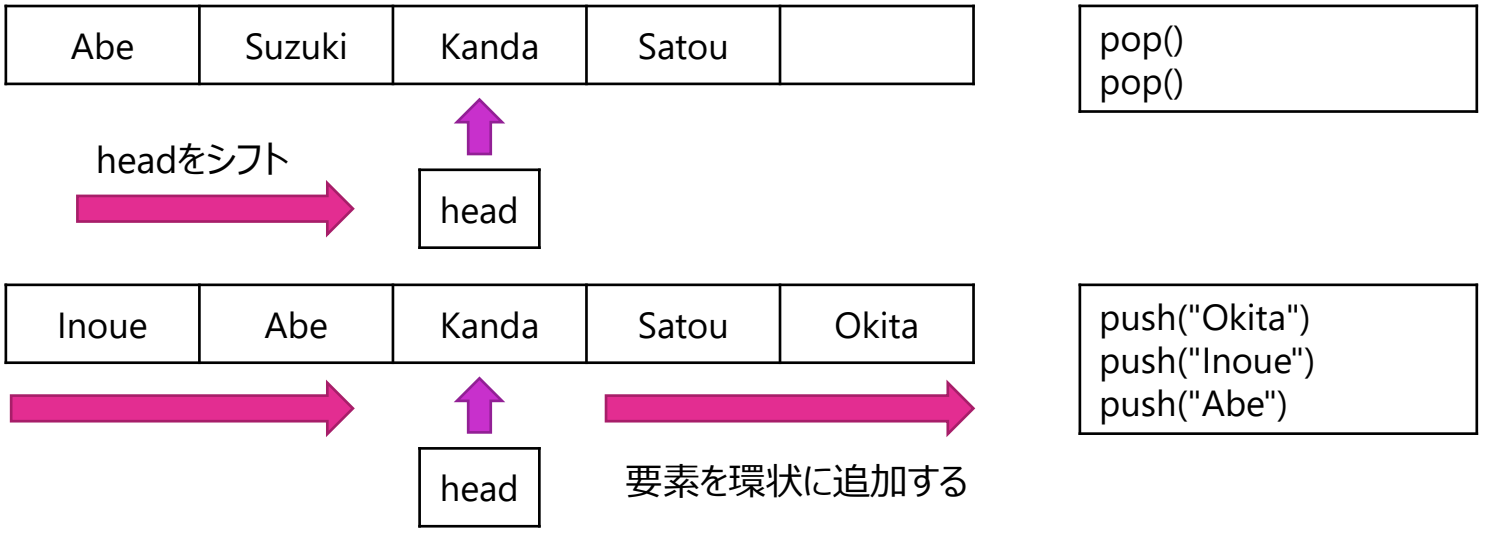
```
class Queue {
    string ar[CAPACITY];
    size_t cursize
    size_t head; // ←
public:
...
```

```
// 配列の要素を先頭の方向へシフトする
void Queue::pop() {
    if (cursize > 0) {
        cursize--;
        head = head + 1;
        //for (size_t i{ 0 }; i < cursize; i++)
        //    ar[i] = ar[i + 1];
    }
}
```

とはいえこのままでは
データの最大長が減ってしまう

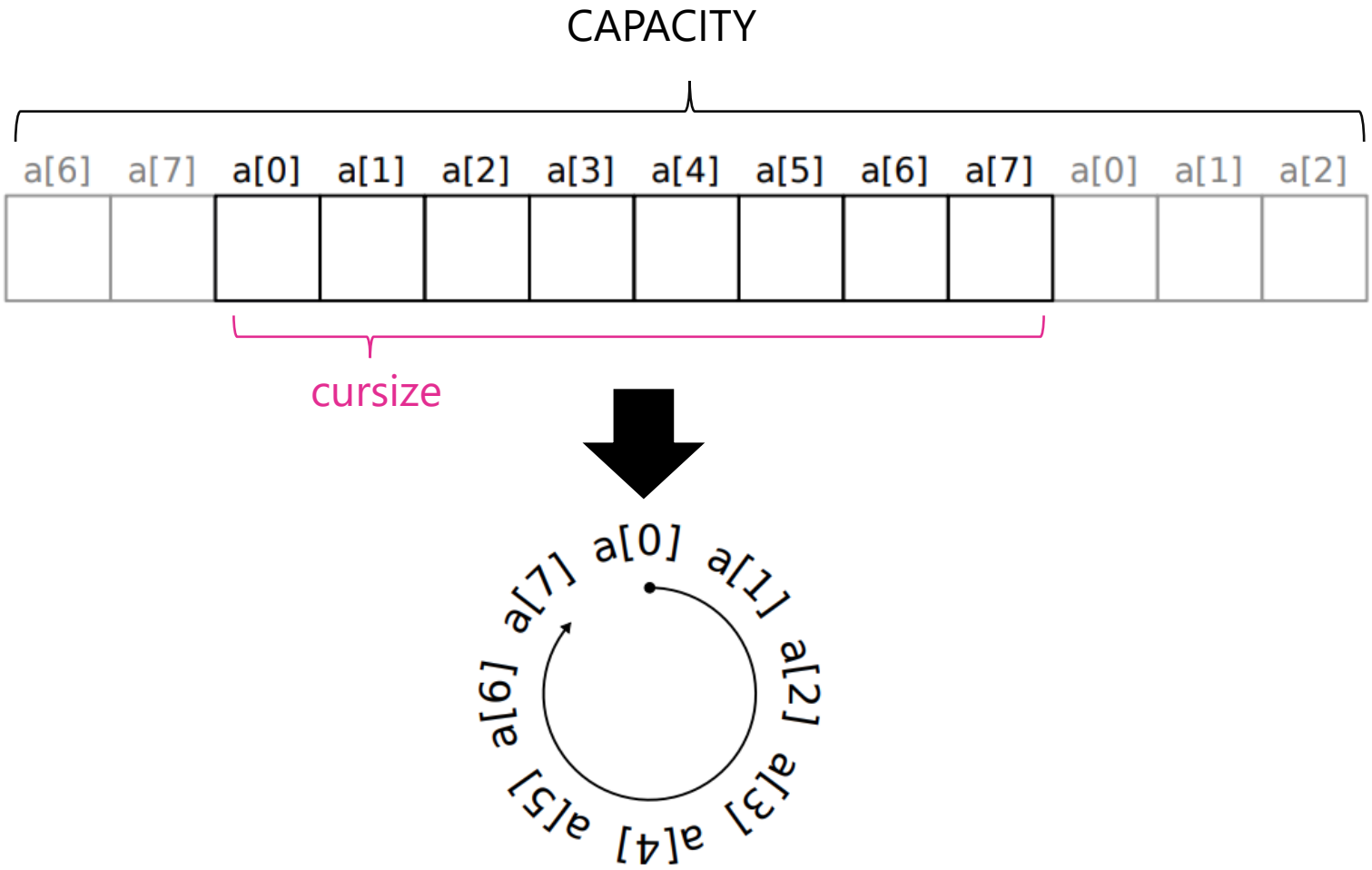
# 環状配列を用いたキューの実装

- 末尾に到達したら配列の先頭に詰めるようにする（環状配列）



# 環状配列を用いたキューの実装

- 環状配列のイメージ



# 固定長配列を用いたキューの実装

```
#include <iostream>
using namespace std;

const size_t CAPACITY{ 5 };

class Queue {
    string ar[CAPACITY];// 大きさCAPACITYの配列
    size_t cursize{ 0 };// 有効な要素数
    size_t head{ 0 };
public:
    ~Queue() { while (!empty()) pop(); }// デストラクタ
    void push(string);// 値sを持つ要素をキューの末尾に追加
    void pop();// キューの先頭の要素を削除する
    string front();// キューの先頭の要素の値を返す
    bool empty() { return cursize == 0; }// キューが空かの判定
    size_t size() { return cursize; }// キューの要素数を返す
};
```

# 固定長配列を用いたキューの実装

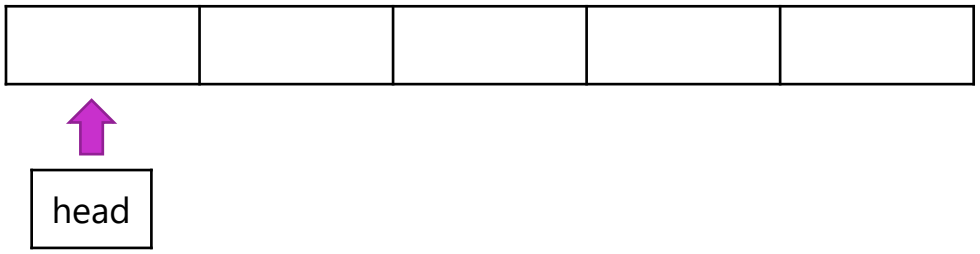
```
// 値xの要素を配列の末尾に追加する
void Queue::push(string s) {
    // ar[cursize] = s;
    ar[(head + cursize) % CAPACITY ] = s;
    ++cursize;
}
// 配列の要素を先頭の方向へシフトする
void Queue::pop() {
    if (cursize > 0) {
        cursize--;
        //head = head + 1;
        head = (head + 1) % CAPACITY;
    }
}
```

# 環状配列を用いたキューの実装

実行例

num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita

キュー




```
q1.push("Inoue");
q1.push("Abe");
q1.push("Kanda");
q1.push("Satou");
```

```
q1.pop();
q1.pop();
```

```
q1.pop();
q1.pop();
```

```
q1.push("Satou");
q1.push("Abe");
q1.push("Suzuki");
q1.push("Kobayashi");
```

```
q1.push("Okita");
```

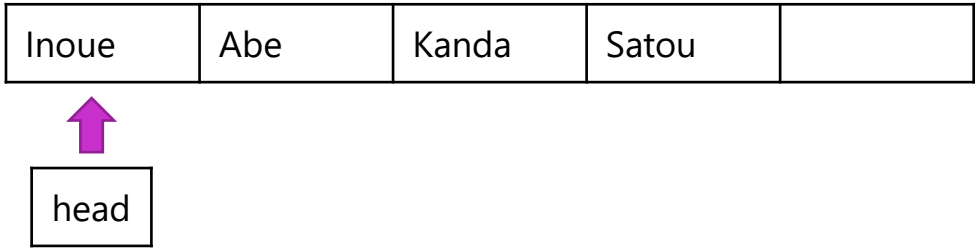
# 環状配列を用いたキューの実装

実行例

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

```
q1.push("Inoue");
q1.push("Abe");
q1.push("Kanda");
q1.push("Satou");
```

キュー



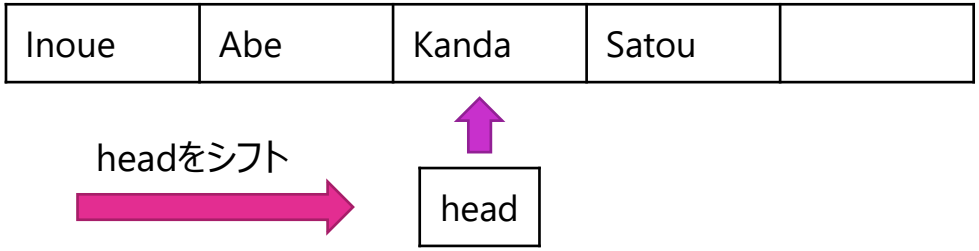
# 環状配列を用いたキューの実装

実行例

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

```
q1.pop();
q1.pop();
```

キュー

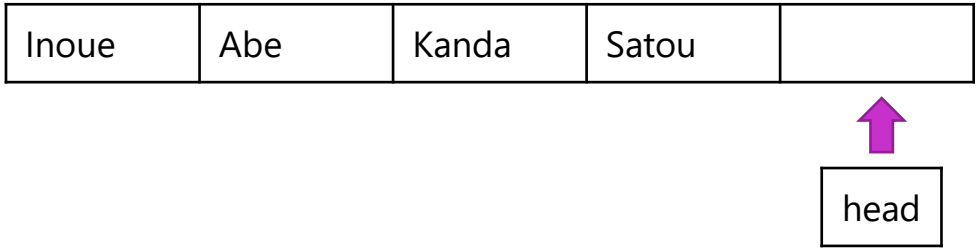


# 環状配列を用いたキューの実装

実行例

num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita

キュー

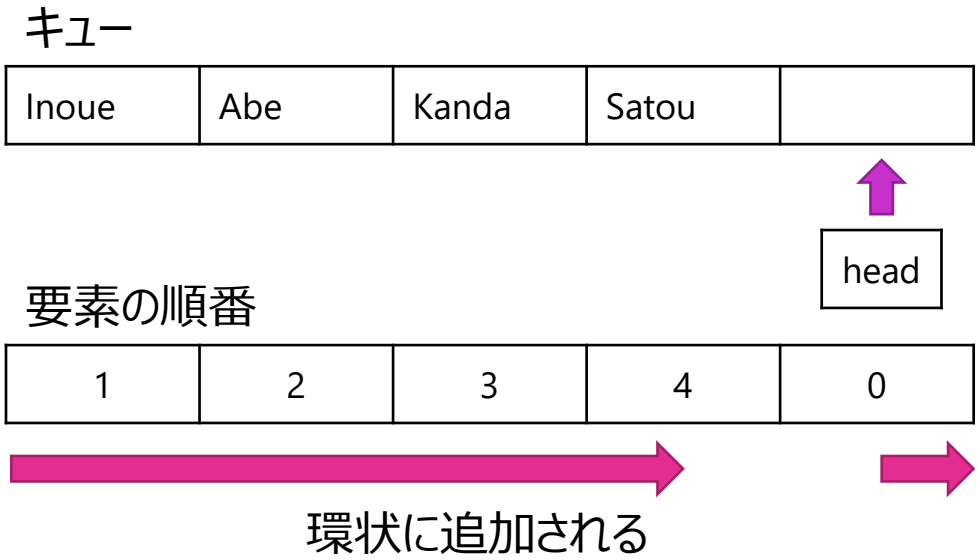



```
q1.pop();
q1.pop();
```

# 環状配列を用いたキューの実装

実行例

num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita

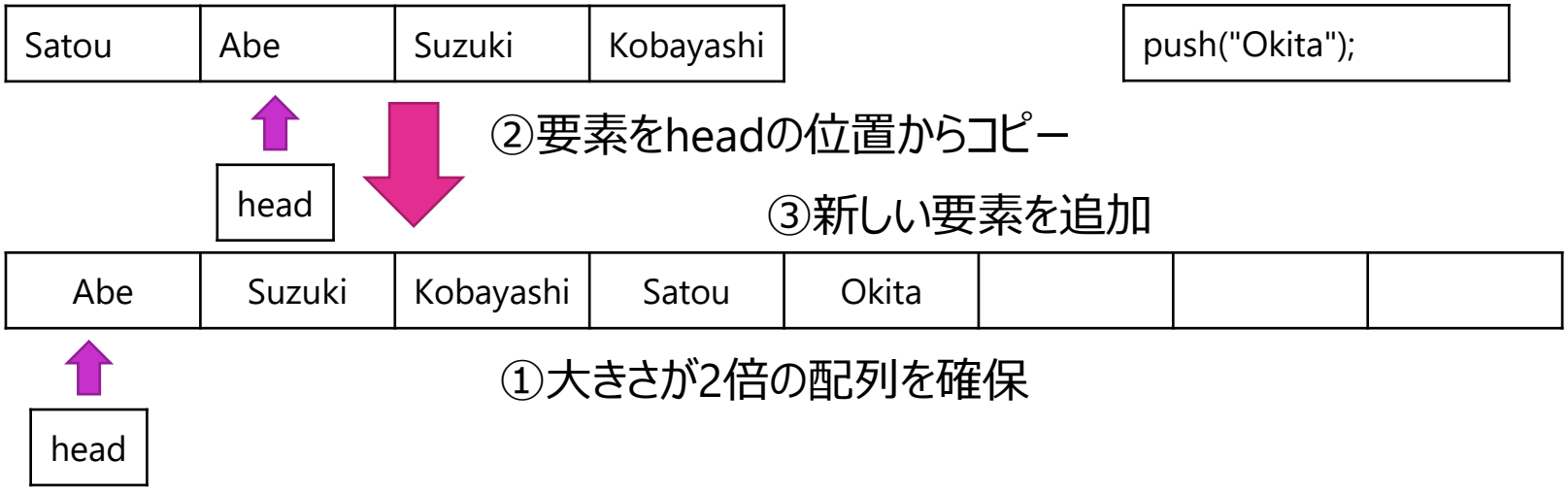



```
q1.push("Satou");
q1.push("Abe");
q1.push("Suzuki");
q1.push("Kobayashi");
```

```
q1.push("Okita");
```

# 動的配列を用いたキューの実装

- 固定長配列ではあらかじめ決められた要素数までしか対応できない
- **動的配列**を用いる
  - 要素を追加する際，現在の要素数の上限を超えている場合は2倍の大きさの配列を確保
  - 現在の配列の要素を先頭からコピーしてから，新しい要素を追加する



# 固定長配列を用いたキューの実装

```
class Queue {
    string* ar;// 配列
    size_t head;
    size_t cursize;// 有効な要素数
    size_t capsize;
public:
    Queue(size_t n = 4) { // コンストラクタ
        ar = new string[n];
        capsize = n;
        cursize = 0;
        head = 0;
    }
    ~Queue() { while (!empty()) pop(); }// デストラクタ
    void push(string);// 値sを持つ要素をキューの末尾に追加
    void pop();// キューの先頭の要素を削除する
    string front();// キューの先頭の要素の値を返す
    bool empty() { return cursize == 0; }// キューが空かの判定
    size_t size() { return cursize; }// キューの要素数を返す
};
```

# 動的配列を用いたキューの実装

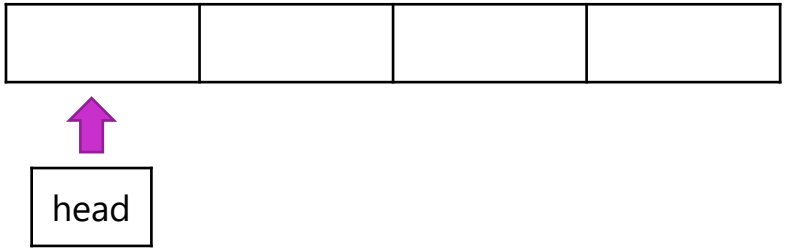
```
// 値xの要素を配列の末尾に追加する
void Queue::push(string x) {
    if (cursize < capsize) {
        ar[(head + cursize) % capsize] = x;
        ++cursize;
    }
    else { // 2倍の大きさの配列を確保
        string* n = new string[capsize * 2];
        for (size_t i = 0, j = head; i < cursize; i++, j = (j + 1) % capsize)
            n[i] = ar[j];
        n[cursize] = x;
        delete[] ar;
        ar = n;
        ++cursize;
        capsize = capsize * 2;
        head = 0;
    }
}
```

# 動的配列を用いたキューの実装

実行例

num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita

キュー




```
q1.push("Inoue");
q1.push("Abe");
q1.push("Kanda");
q1.push("Satou");
```

```
q1.pop();
q1.pop();
```

```
q1.pop();
q1.pop();
```

```
q1.push("Satou");
q1.push("Abe");
q1.push("Suzuki");
q1.push("Kobayashi");
```

```
q1.push("Okita");
```

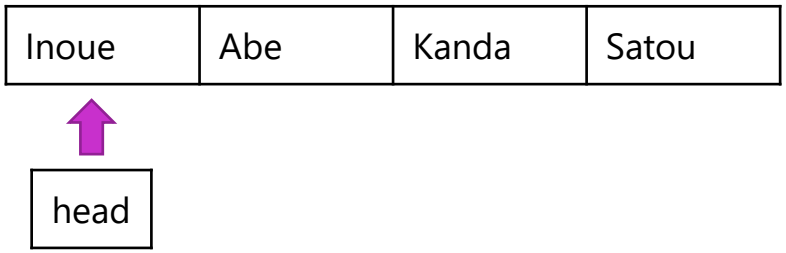
# 動的配列を用いたキューの実装

実行例

```
num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita
```

```
q1.push("Inoue");
q1.push("Abe");
q1.push("Kanda");
q1.push("Satou");
```

キュー



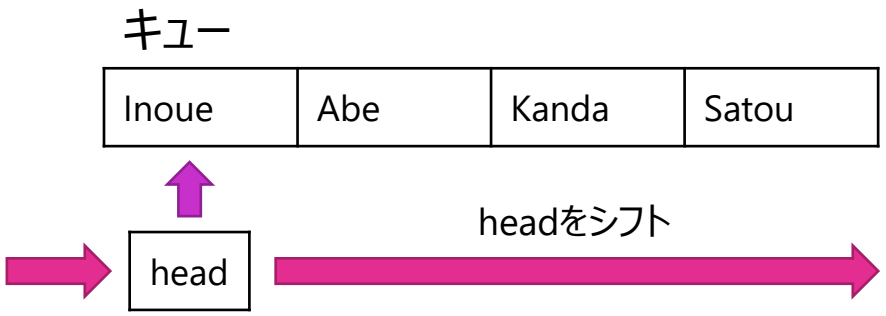
# 動的配列を用いたキューの実装

実行例

num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita

q1.pop();
q1.pop();

q1.pop();
q1.pop();



# 動的配列を用いたキューの実装

実行例

num: 4, data: Inoue Abe Kanda Satou
num: 2, data: Kanda Satou
num: 0, data:
num: 4, data: Satou Abe Suzuki Kobayashi
num: 5, data: Satou Abe Suzuki Kobayashi Okita

```
q1.push("Satou");
q1.push("Abe");
q1.push("Suzuki");
q1.push("Kobayashi");
```

```
q1.push("Okita");
```

キュー