データ構造とアルゴリズムI

馬　青

授業の実施形態

- 教室対面授業。PCは必携
- 講義資料の配布、授業関係の連絡はすべてTeamsにて行う
- 講義資料は約1週間前に配布（予習できるように）
- 授業中に毎回演習を実施する。演習は以下の二通りの形式で行う
  - manaba小テスト
    - 成績は記録され、平常点として使う
    - 小テストは自動採点を基本とするので、問題文に指示される答え方について細心の注意を
  - 授業中その都度の質問や演習
    - 成績は記録されないが、その場で指名して答えてもらう

科目teamのQRコード

成績評価

- 期末試験はない。代わりに確認テストを第8回（最終回）の授業で実施する
- 確認テスト30%，manaba小テスト70%で成績評価を行う
  - 授業・テストの受講・受験形態によって変更する可能性がある。その場合、別途授業やTeamsを通じて知らせる
- 重要なのは、毎回授業で実施する演習を重視すること。これらが最終の成績評価を大きく左右する

manaba小テストについての注意

- 自動採点を基本とするので、問題文に指示される答え方について細心の注意を！
- なお、問題文に指示がなくても以下のことは必ず守ること！！
  - セミコロン「;」は、manabaの仕組み上、予約語として使用されているので、答えに含んでしまうと、自動採点ができなくなることに注意
  - 教員は必ずセミコロン記入不要なように出題している
  - 答えに余分な空白を入れると、正解判定が正しくできなくなる
  - z=x+y○　z　＝x+　y×
  - しかし、必要最小限な空白は入れるべき！！
    - たとえば二人の名前を記入しなさいの場合、
      - 「東京太郎　京都次郎」はOK,「東京太郎京都次郎」はNG
  - 英数字はすべて半角：　aAはOK aAはNG
- 時間は多めに取るので、注意して解答すること。上記不注意による不正解は基本、救済しない

導入

データ構造とは

データ構造とは

- データをアルゴリズムで扱いやすい（効率的にアクセス・変更できる）ように、一定の形式で格納したもの
  - データ構造＝データの集まり＋データ間の関係＋データへの操作機能（探索・変更など）
- 様々なアルゴリズムの実現に必要なデータ構造に関する学問

例：

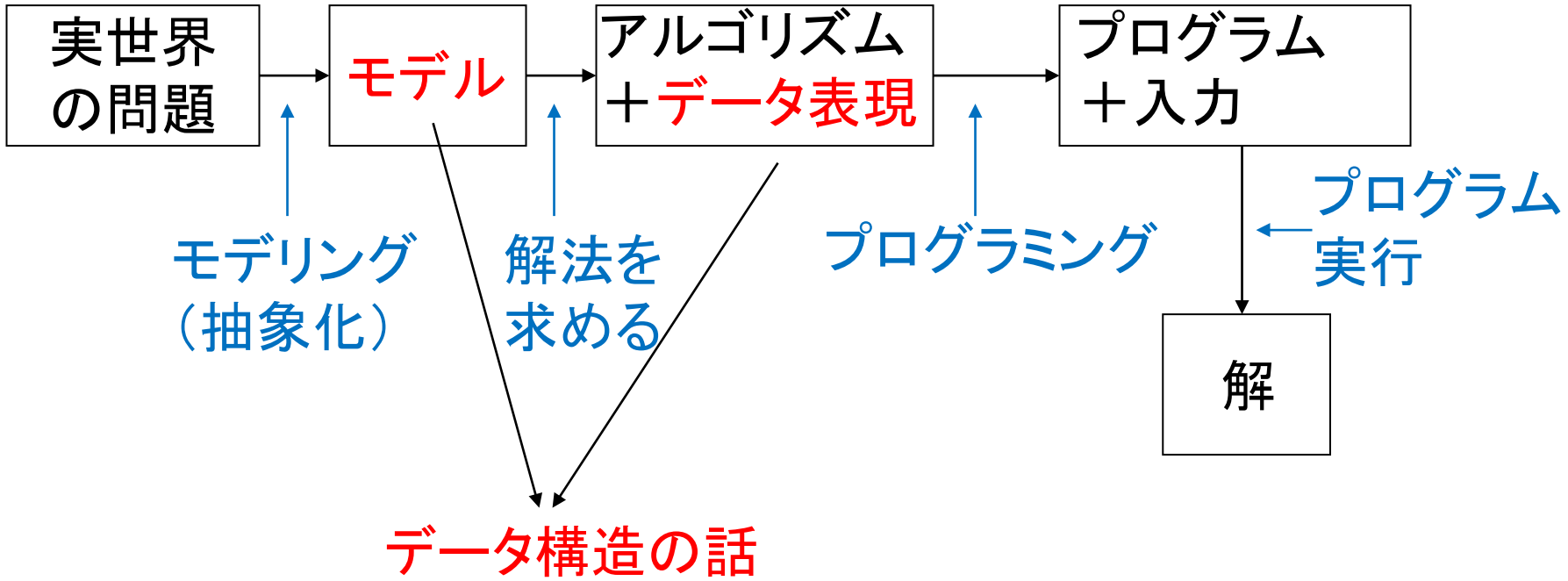
アルゴリズム	選択ソート	クイックソート（再帰なし）	ヒープソート	グラフ幅優先探索
データ構造	配列	スタック	ヒープ	キュー

データ構造とは

- データ構造は、実世界の問題を抽象化した結果に「モデル」としても用いられる。たとえば、
  - 実世界の問題：最短経路問題・日本語形態素解析
  - データ構造：グラフ

データ構造とは

- コンピュータで問題を解くのは、以下の段階を経て行われる



データ構造とは

- データ構造：配列、連結リスト、スタック、キュー、木、ヒープ、グラフ、ハッシュがある
- 本授業は上記について紹介していく

  （ハッシュは「アルゴリズム・演習II」で習得済み）

配列

データ構造の最も基本的な構成法

配列（Array）とは

- 各種のデータ構造の基本要素を「セル」と言い、１個のデータを格納するためのもの
- 配列は同じデータ型を格納する複数のセルを順次に並べたもの。つまり、セルを格納する計算機のメモリは物理的に並んでいる（メモリの番地が連番になっている）
- 配列中の個々のセルは配列名とセル番号（添え字）で指定する

配列 a	50	30	10	20	40
添え字（セル番号）	0	1	2	3	4
各セルの指定	a[0]	a[1]	a[2]	a[3]	a[4]

配列への基本操作

- データの参照
  - 添え字（セル番号）がわかっている場合
  - 添え字（セル番号）がわかっていない場合：データの探索
- データの削除
- データの挿入
- 注意：
  - 本授業ではわかりやすくするために、各セルに整数のみが格納されているようなデータ構造を例にしているが、実際は、各セルに複合データ（たとえば学籍番号、氏名、科目の成績など複数のフィールドからなるデータ）が格納される

  - 実用上は、たとえば、上記データの探索の目的は10という整数値を探索して10を取り出すようなものではなく、たとえば学籍番号Y210081を探索して、それ関連の情報（氏名、科目の成績など）を取り出す

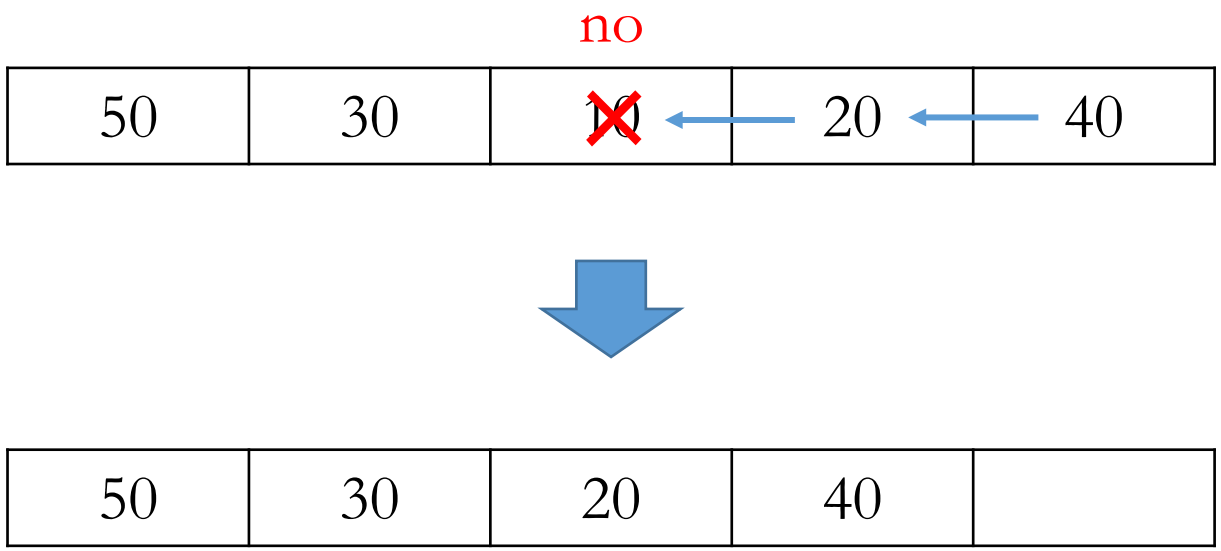
データの探索

- 探索データと配列の各セルとを順に比較し、一致したときのセル番号が探索の解になる（線形探索）

```
int ArraySearch(int n, int a[], int x)
{
  int i;
  for(i = 0; i < n; i++)
    if(x == a[i]) return i;
  return -1;
}
```

データの削除

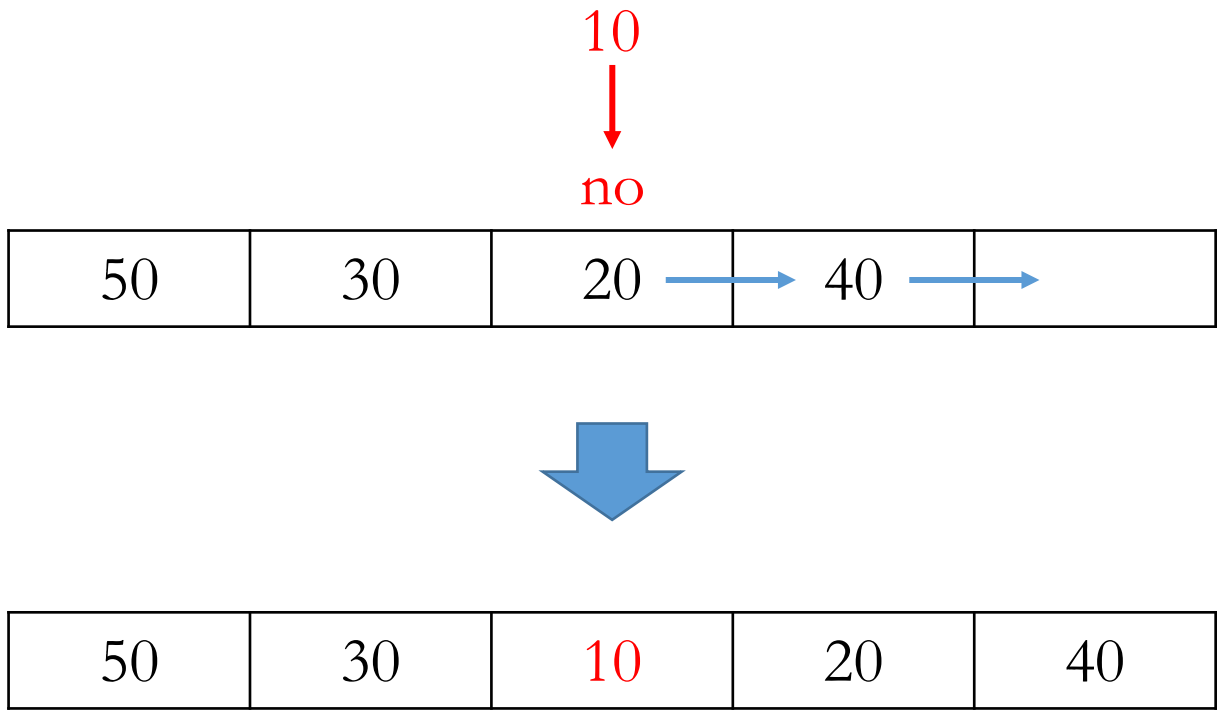
- 削除データのセルより後ろのセルのデータを1個ずつ前へシフトすることで実現
- 実質「削除」という操作はない




```
int n=5, a[100]={50,30,10,20,40}, no=2;
int ArrayDelete(int n, int a[], int no)
{
  int i;
  for(i = ?; i < ?; i++)
    a[i] = a[i+1];
  return n-1;
}
```

データの挿入

- データを挿入するセルより後ろのデータをまず1個ずつ後ろへシフトしてから挿入




```
int n=4, a[100]={50,30,20,40}, no=2;
int ArrayInsert(int n, int a[], int no, int x)
{
  int i;
  for(i = ?1; i > ?2; i--)
    a[i] = ?3;
  ?4 = x;
  return n+1;
}
```

manaba小テスト：01-1

- 10分
- 4点

配列の問題点

- データの削除と挿入処理がコスト大→大規模のデータの場合、処理に時間がかかる
- データの削除でメモリの無駄が生じる
- データの挿入では事前に大きめのメモリを確保しておかなければならない

- セルとセルが連番で結び付けられているのがこれら欠点の原因

ちょっとした工夫…連番問題を解消

配列

セル番号	（データ）
0	新大阪
1	京都
2	名古屋
3	新横浜
4	東京

ヘッドセル

2

新しいデータ構造

セル番号	（データ	次のセル番号）
0	東京	-1
1	京都	3
2	新大阪	1
3	名古屋	4
4	新横浜	0

実装

配列：

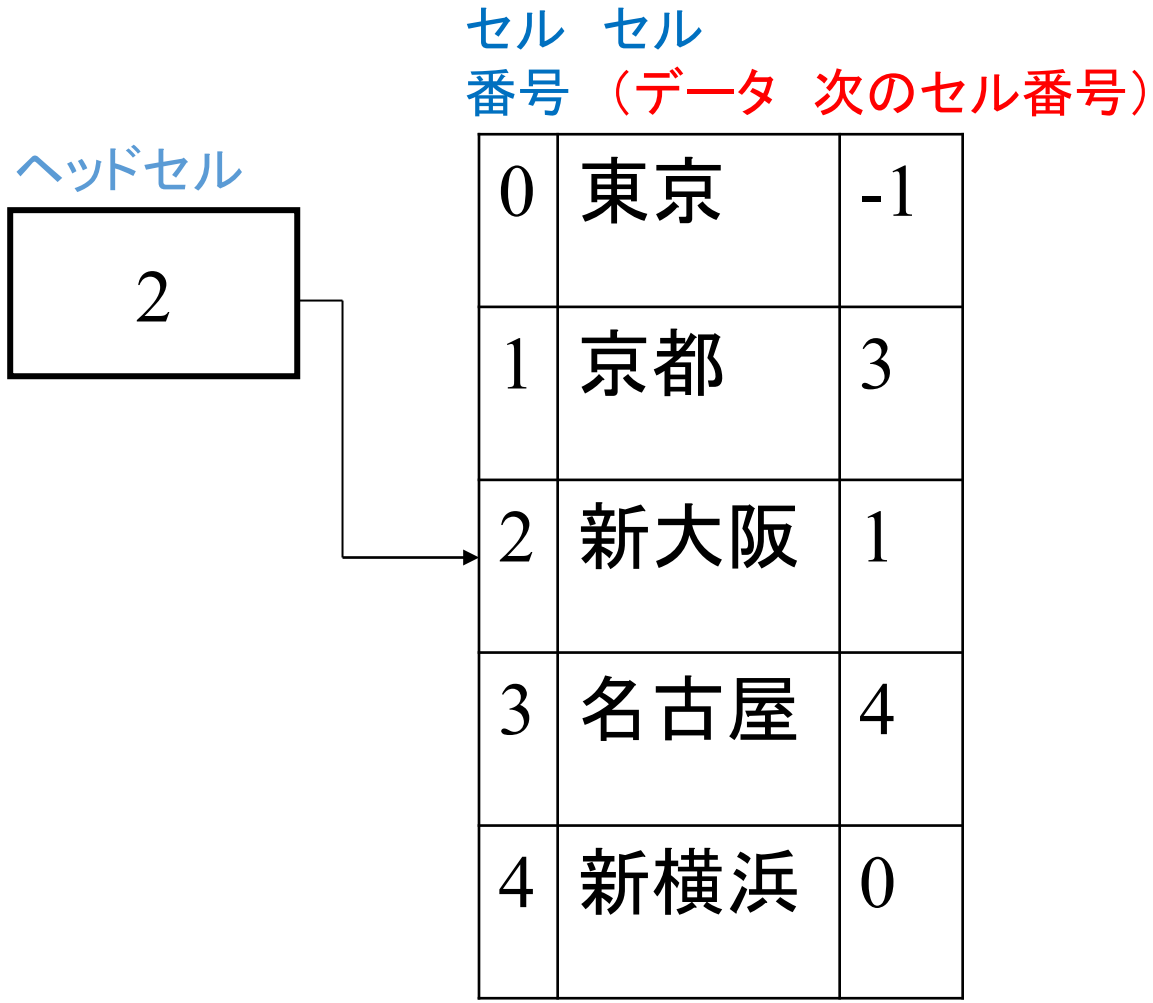
```
char a[100][64] = {“新大阪”, “京都”, “名古屋”,
“新横浜”, “東京”};
```

新しいデータ構造：

```
typedef  struct {
  char s[64]; int p;
}DATA;
int head = 2;
DATA a[100] = {
            {"東京", -1},
            {"京都", 3},
            {"新大阪", 1},
            {"名古屋", 4},
            {"新横浜", 0}
};
```

新しいデータ構造への探索

- たとえば、「名古屋」を格納しているセルの番号は？



- セル番号2から出発し、「次のセル番号」に従って順に各セルと探索データとの照合を行う
- 今の例では、新大阪→京都→名古屋の順に照合し番号3を答えとする

探索の手順

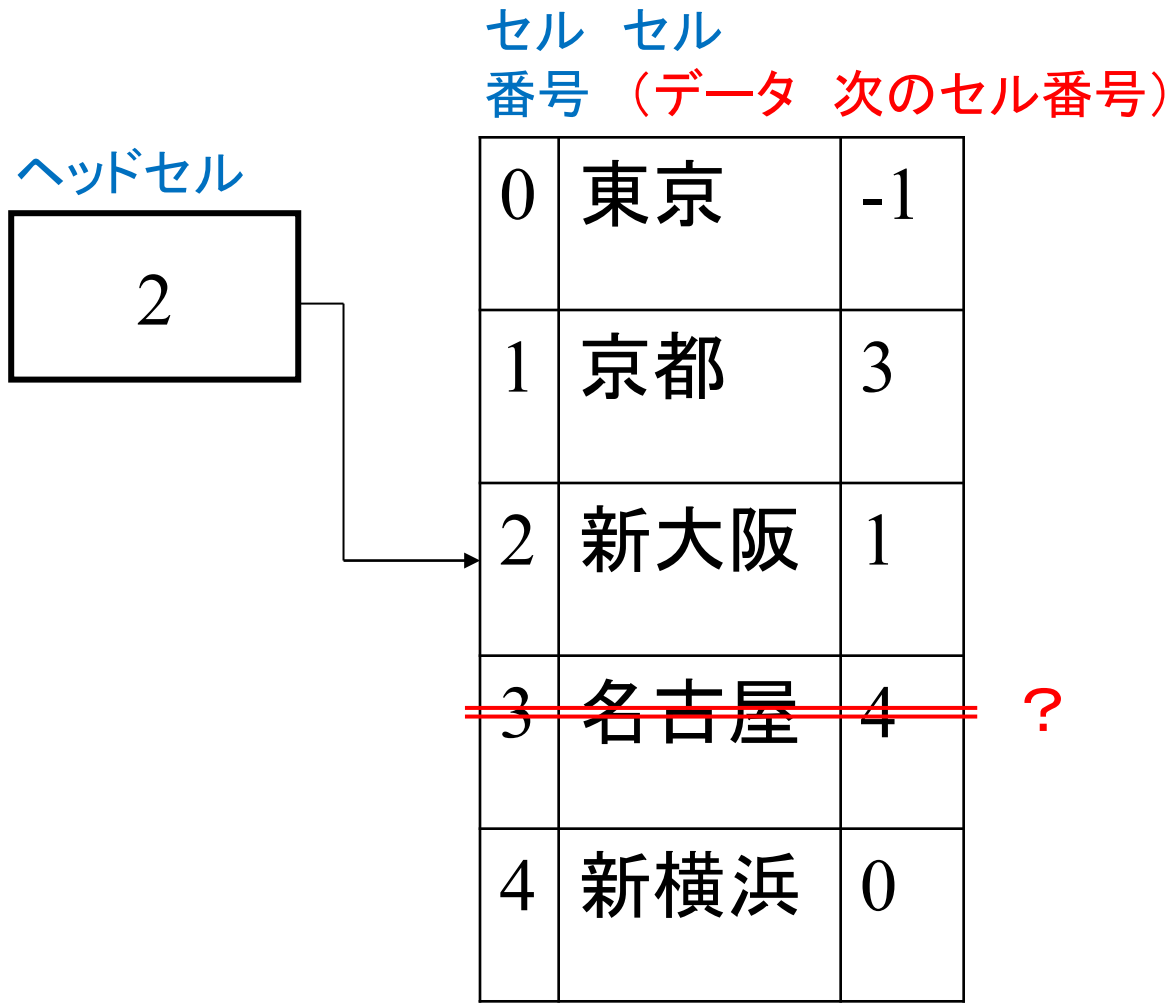
1. ヘッドセルの中身をセル番号noとする
2. セル番号noが-1になるまで以下を繰り返す
   2.1 セルnoのデータと探索データを比較する。一致していればセル番号noを返し、終了
   2.2 セルnoの「次のセル番号」をセルの番号noとする
3. -1を返して終了

- セルnoとは、番号noのセルの意味。以降同様
- ここの「探索順番」は「次のセル番号」の情報を利用しての論理上の順番
- 実装上、論理上の順番を用いなくても、つまり「次のセル番号」情報を用いなくても、物理的な順番、つまり、各セルを配列要素の順番で照合しても探索ができる

実装

```
typedef struct {
  char s[64]; int p;
}DATA;
int DataSearch(int head, DATA a[], char x[])
{
    int no = head;
    while(no != -1){
        if(strcmp(x, a[no].s) == 0) return no;
        ?
    }
    return -1;
}
```

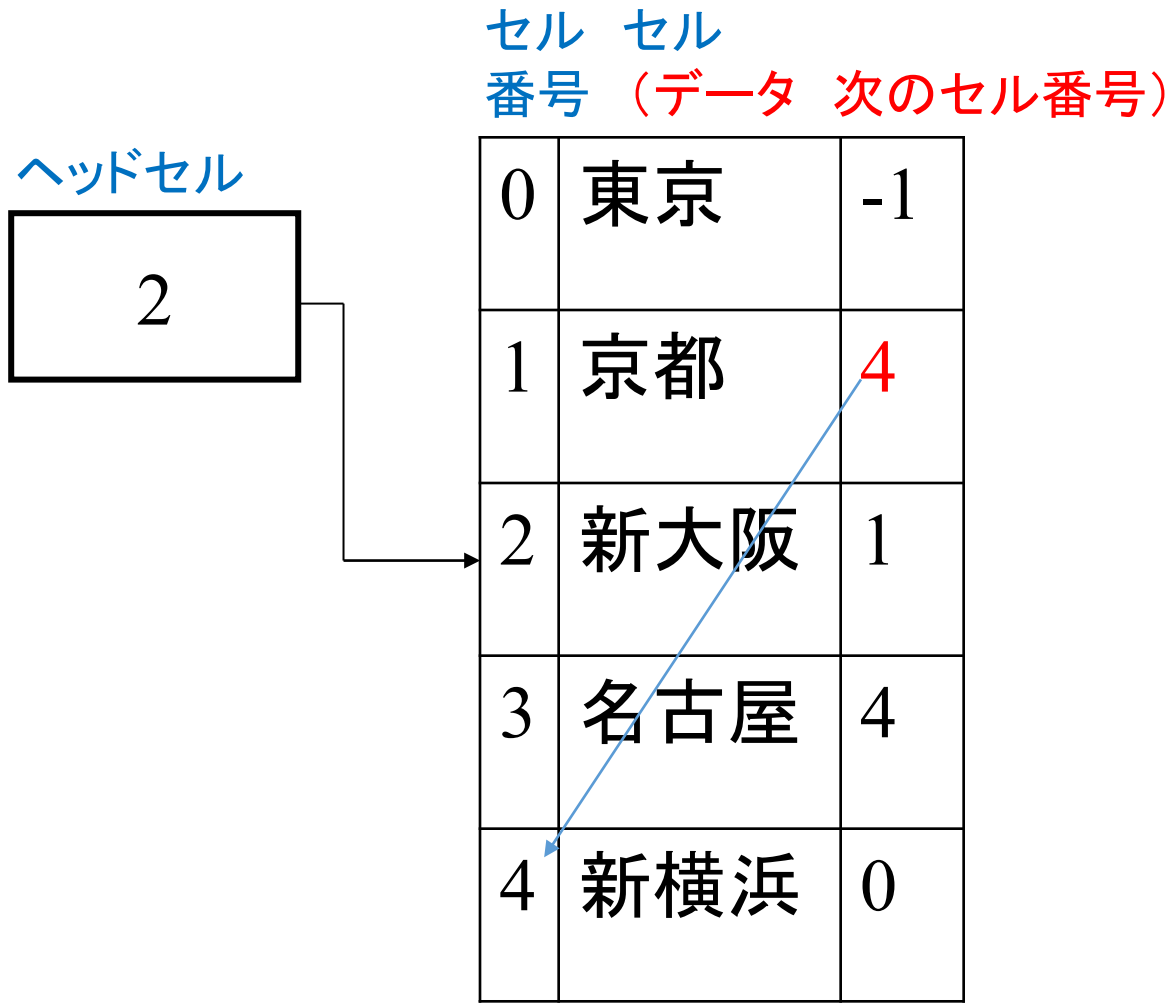
データ削除

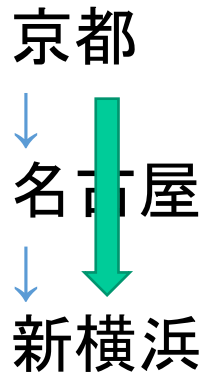


データ削除の前

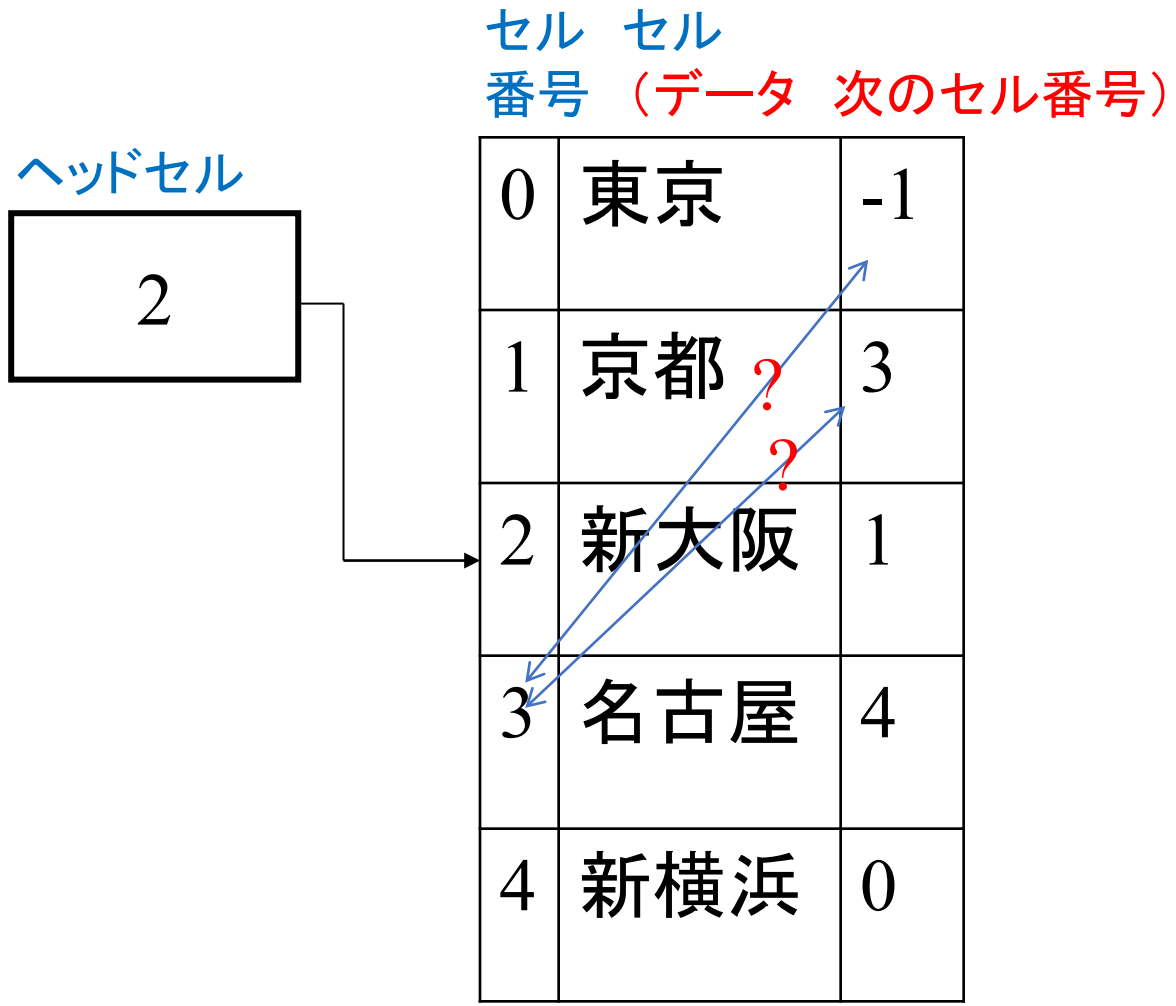

セル　セル
番号　（データ　次のセル番号）
ヘッドセル
2
0 東京 -1
1 京都 3
2 新大阪 1
3 名古屋 4
4 新横浜 0
京都
名古屋
新横浜

データ削除の後

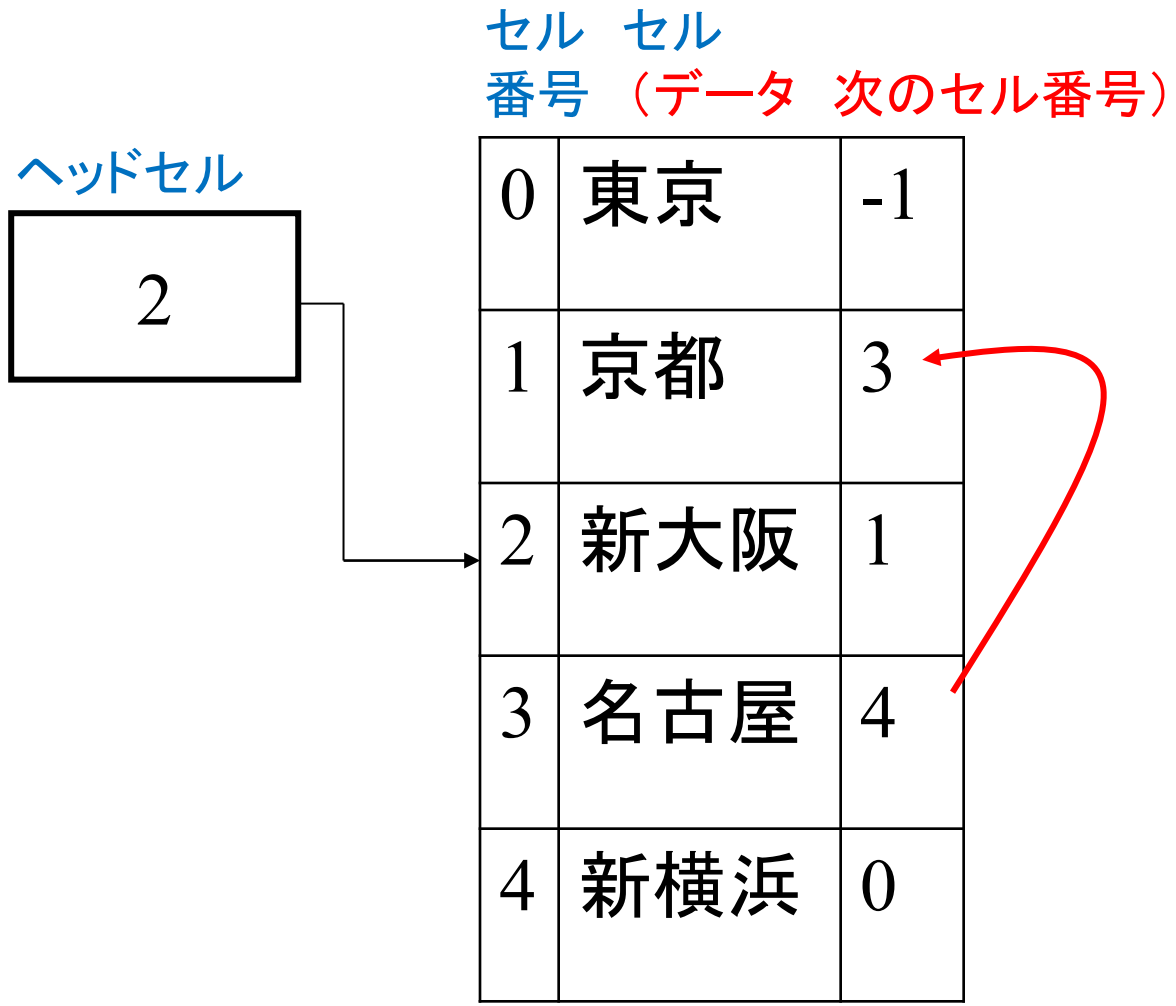

セル　セル
番号　（データ　次のセル番号）
ヘッドセル
2
0 東京 -1
1 京都 4
2 新大阪 1
3 名古屋 4
4 新横浜 0


京都
名古屋
新横浜

データ削除の方法



データ削除の方法



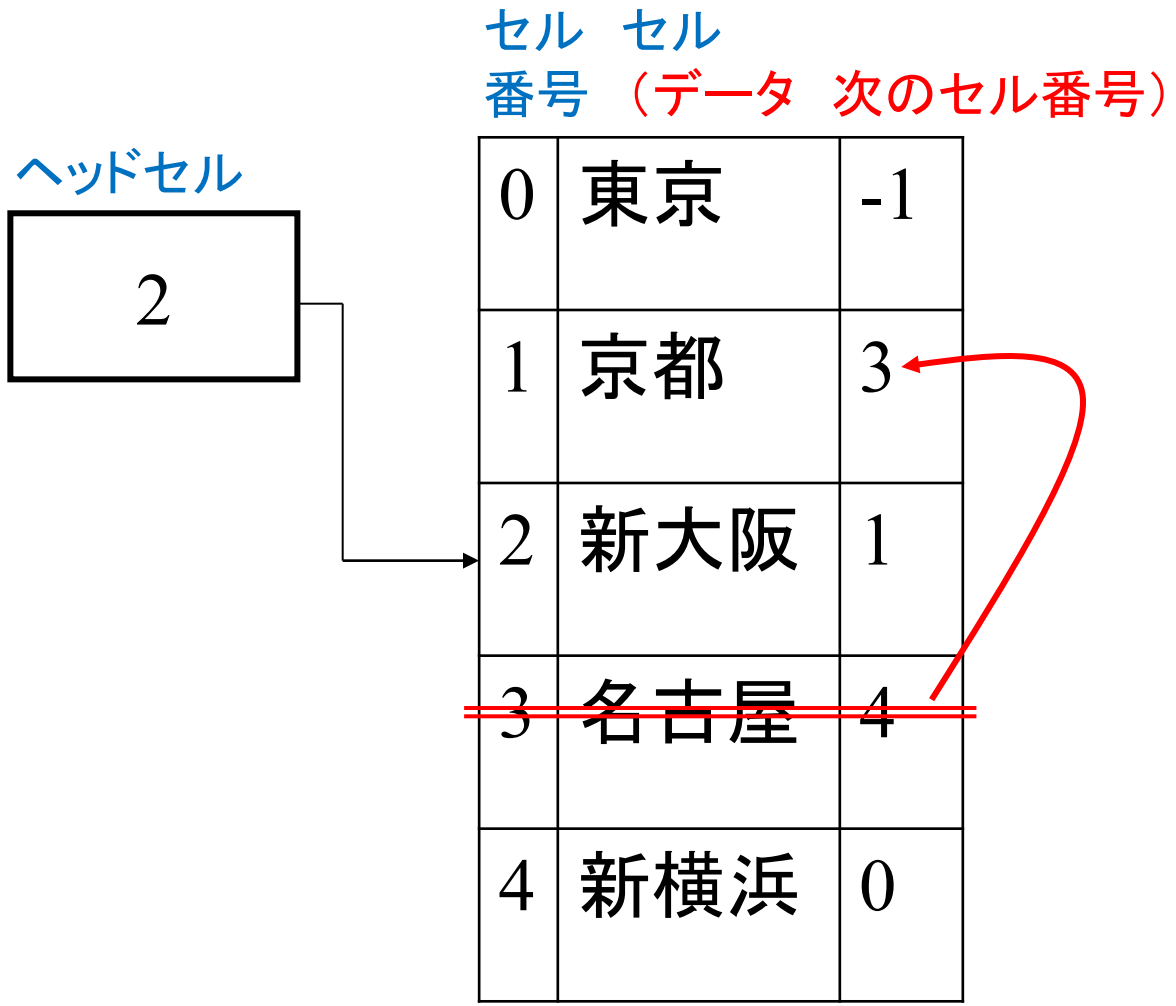
データ削除の結果

削除の手順

1. 削除すべきデータのセル番号をnoとし、セルを順番に調べ、「次のセル番号」がnoであるセルiを見つける（今の場合i=1）
2. セルiの「次のセル番号」にセルnoの「次のセル番号」を与え、終了

- ここの「順番」とは「次のセル番号」の情報を利用しての論理上の順番
- 実装上、論理上の順番を用いなくても、つまり「次のセル番号」情報を用いなくても、物理的な順番で、つまり、各セルを配列要素の順番で照合してもセルiを見つけることができる

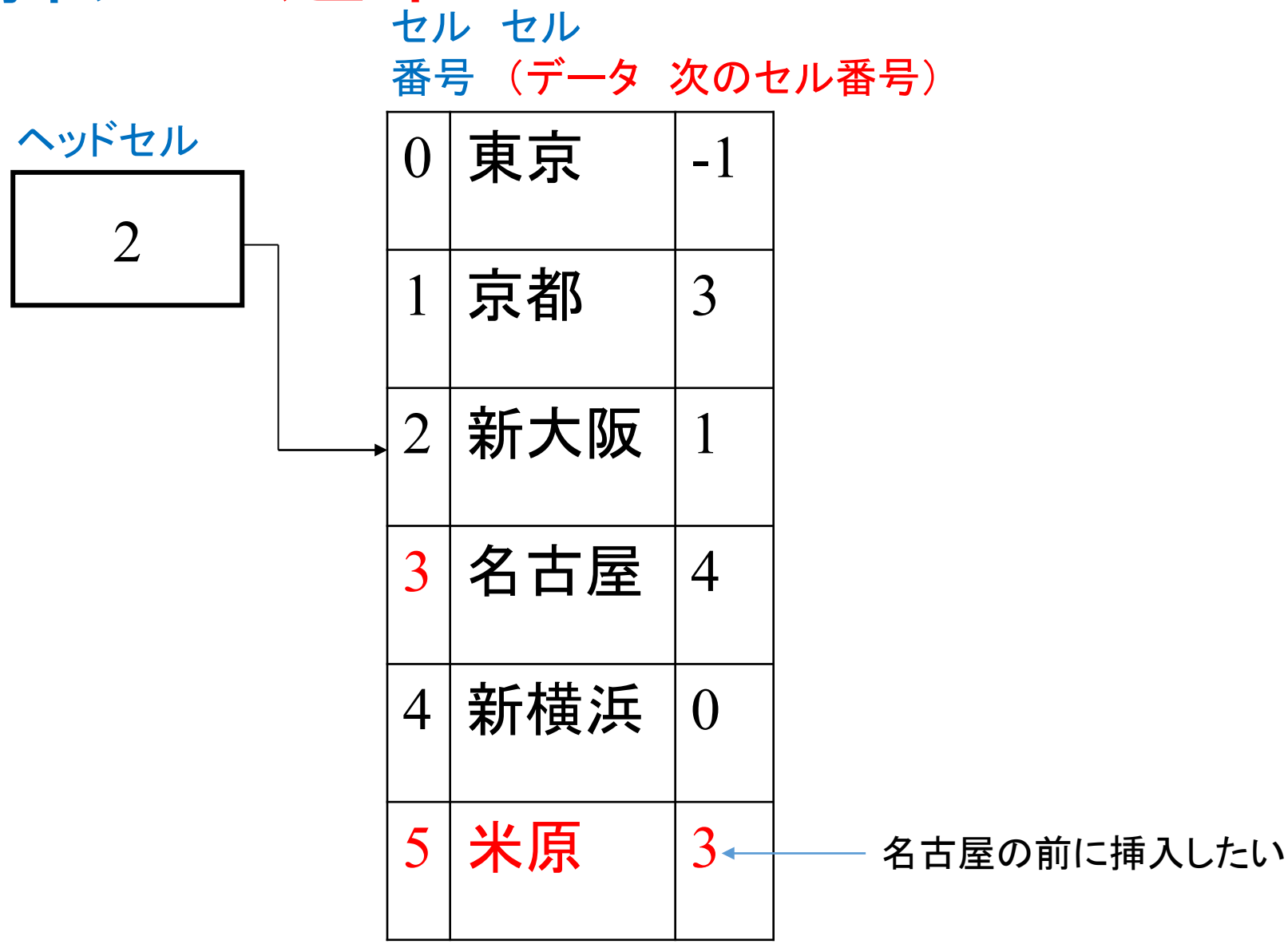
実装



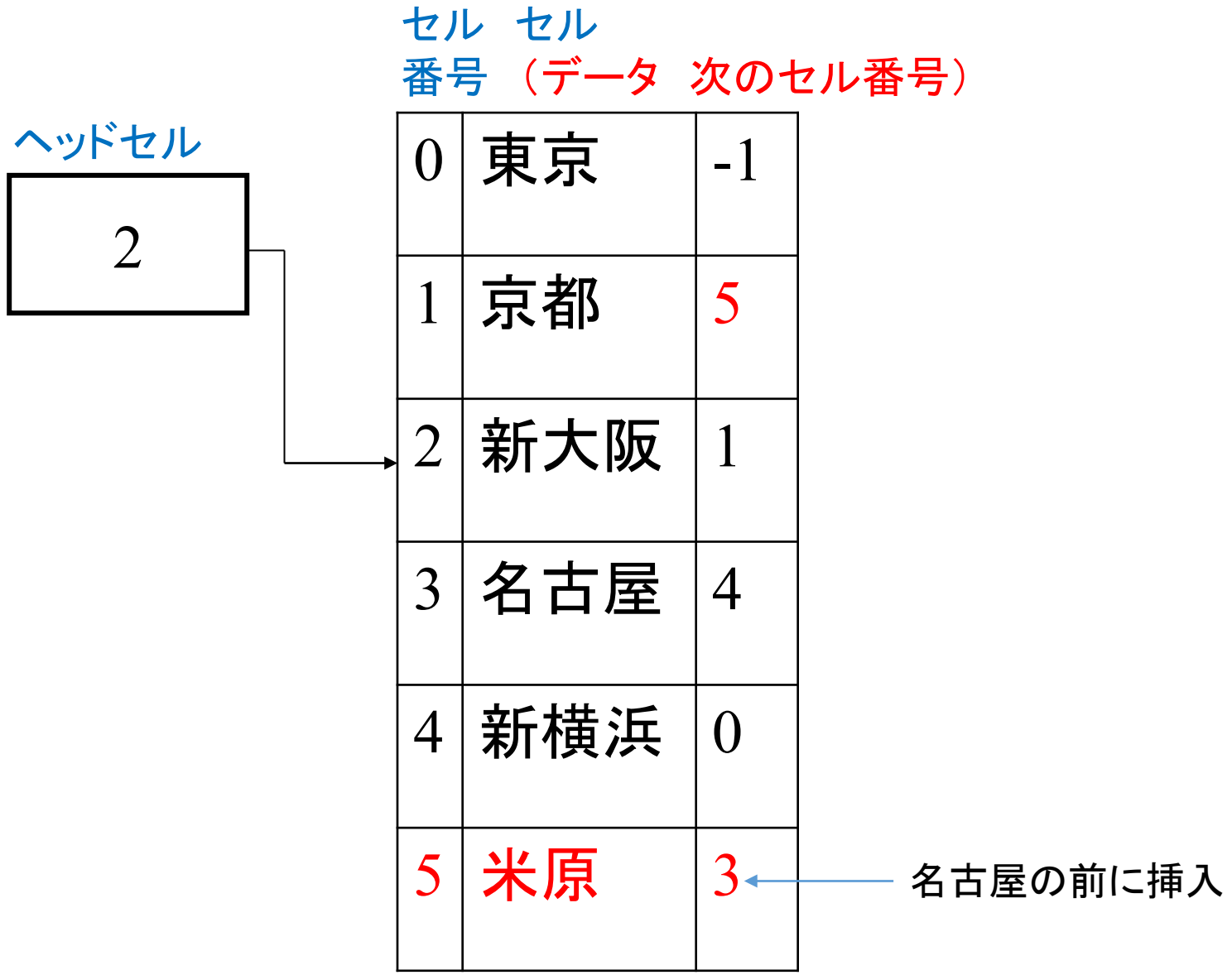

```
void DataDelete(int head, DATA a[], int no)
{
  int i = head;

  while(i != -1){
    if(a[i].p == ?){
      a[i].p = ?
      return;
    }
    i = ?
  }
}
```

データ挿入の前

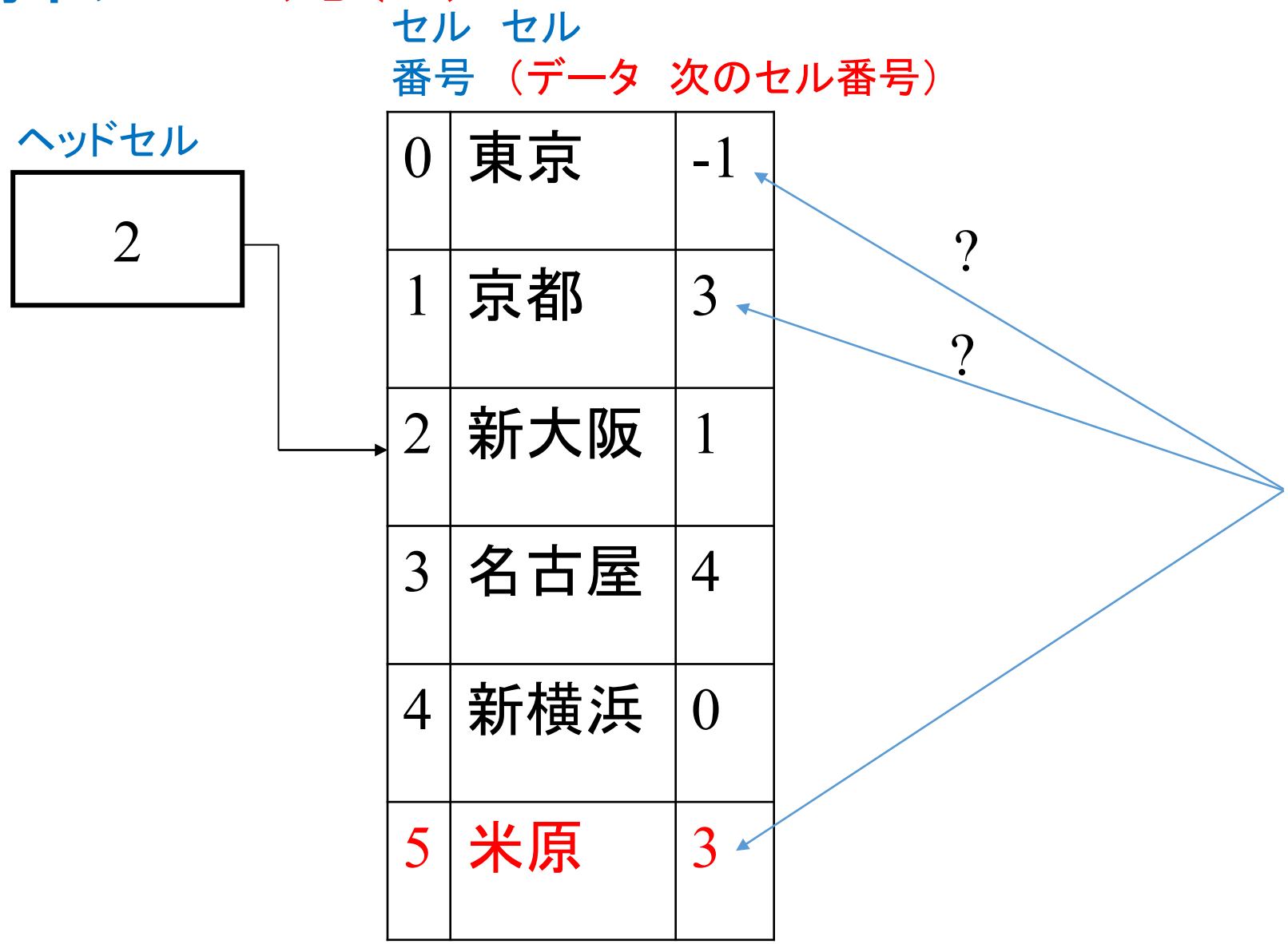
データ挿入の途中


セル　セル
番号　（データ　次のセル番号）
ヘッドセル
2
0 東京 -1
1 京都 3
2 新大阪 1
3 名古屋 4
4 新横浜 0
5 米原 3
名古屋の前に挿入したい

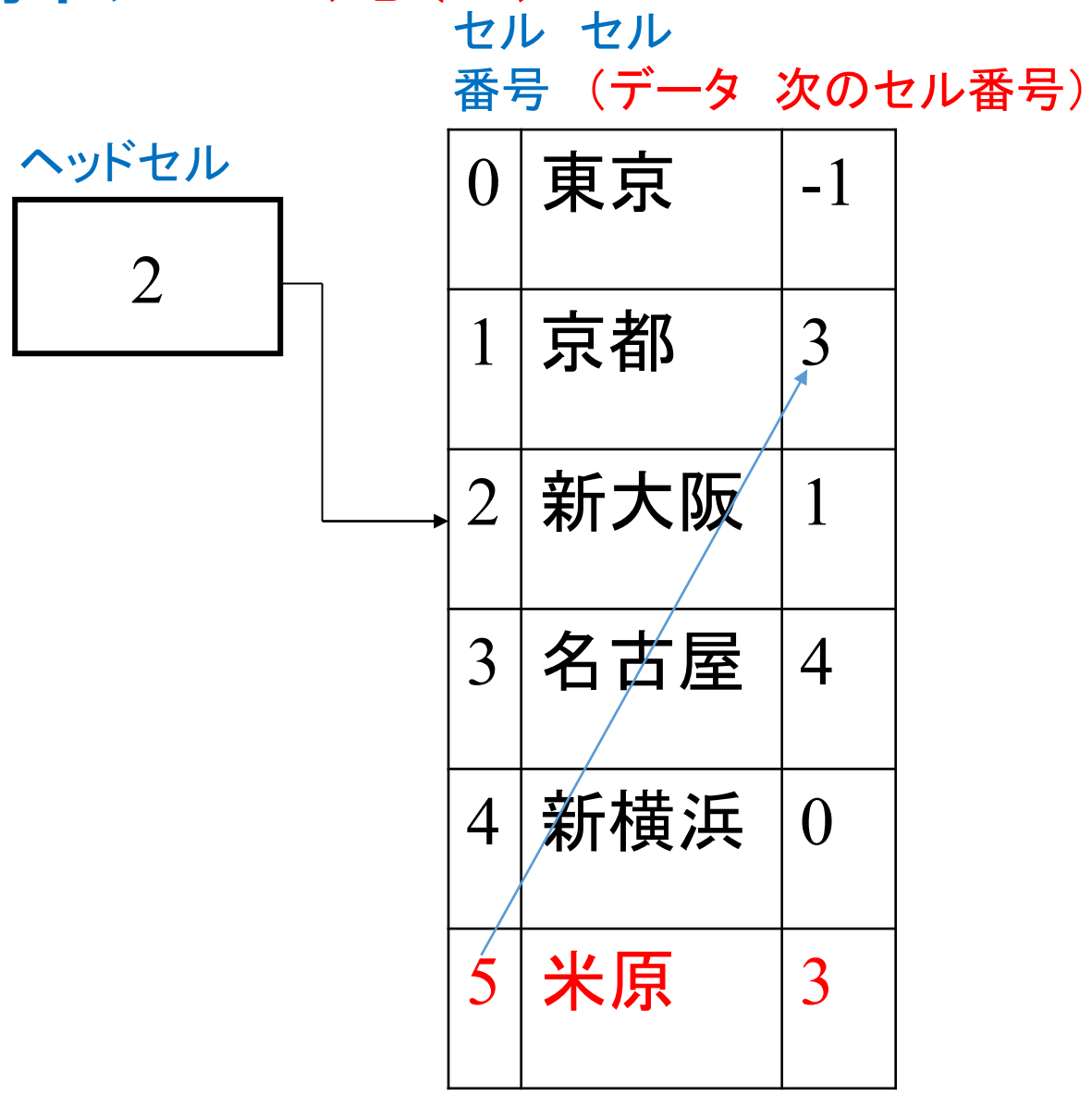
データ挿入の後


セル　セル
番号　（データ　次のセル番号）
ヘッドセル
2
0 東京 -1
1 京都 5
2 新大阪 1
3 名古屋 4
4 新横浜 0
5 米原 3
名古屋の前に挿入

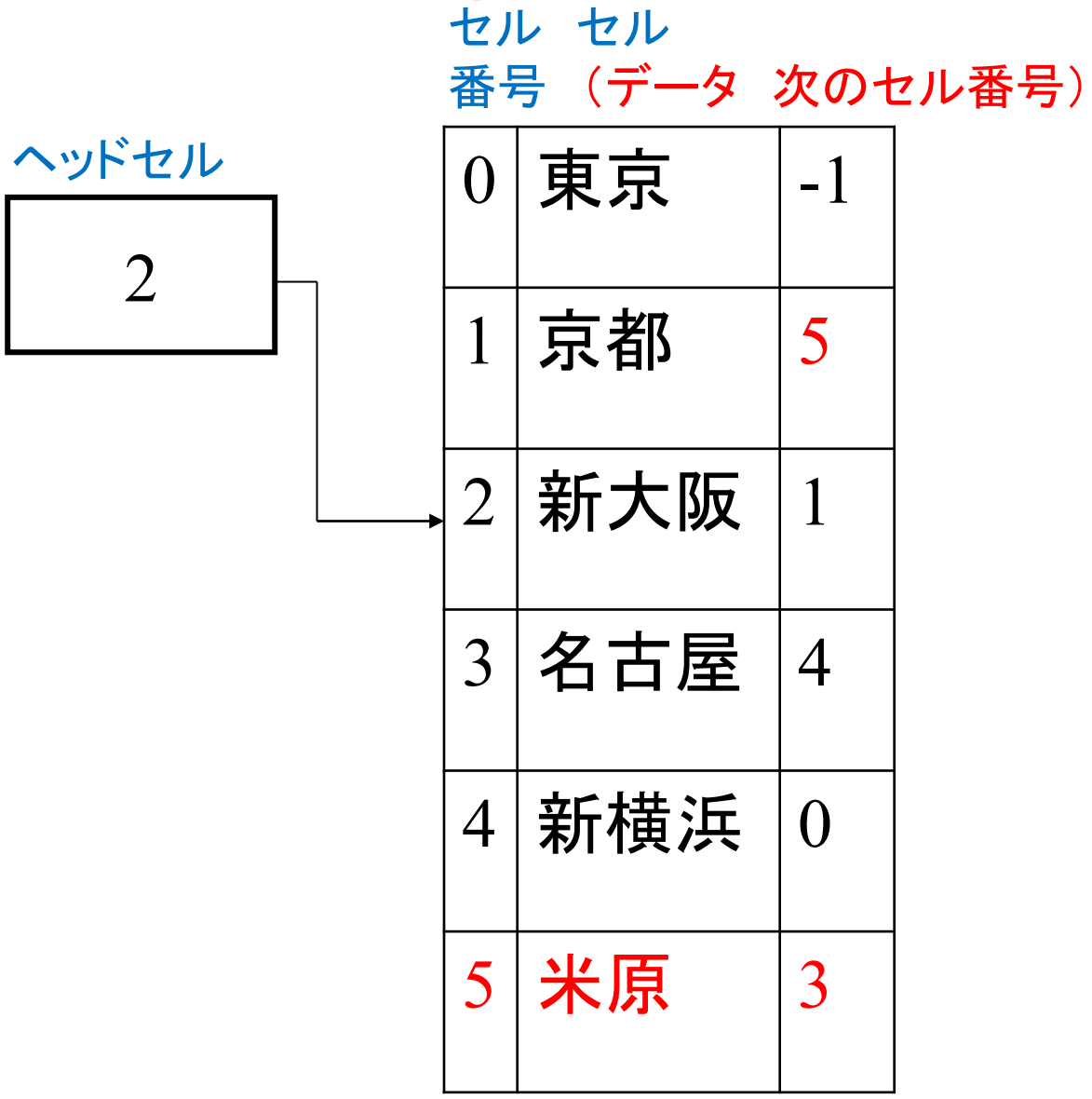
データ挿入の方法



データ挿入の方法



データ挿入の結果



挿入の手順

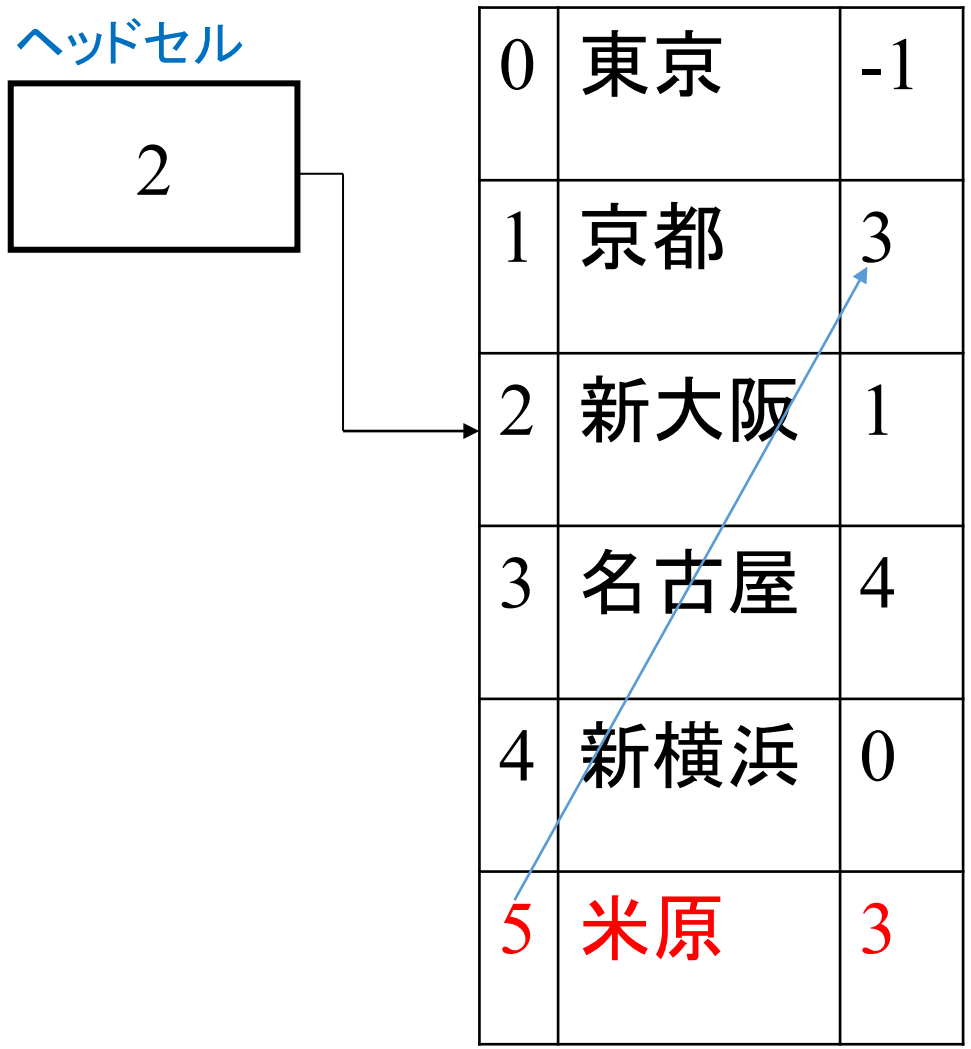
n個のデータを格納しているデータ構造に、構造体データxを挿入する

1. セル番号nのセルにデータxを格納（例：米原 3）
2. 「次のセル番号」がx.p（例：3）のセルiを見つける（今の場合i=1）
3. セルiの「次のセル番号」に1.で追加したセルの番号nを与え（今の場合は5）、処理終了

注意：

- 前述他の操作と同様、「次のセル番号」情報を用いなくても各セルを配列要素の順番で照合してもセルiを見つけることができる

実装




```
int DataInsert(int n, int head, DATA a[], DATA x)
{
  int i = head;
  a[n] = ?1;
  while(i != -1){
    if(x.p == ?2){
      a[i].p = ?3;
      return n+1;
    }
    i = ?4;
  }
}
```

manaba小テスト：01-2

- 15分
- 8点

Question

- セルの後ろへの挿入（例えば「米原」を「京都」の後ろに）は？

出力の実装

```
void DataPrint(int head, DATA a[])
{
  int no=head;
  puts("list");
  while(no != -1){
       ?
   }
}
```

プログラムを実行してみよう

```
int main(void)
{
  int head=2, n=5;
  DATA a[100]={{"東京", -1},{"京都", 3},{"新大阪", 1},{"名古屋", 4},{"新横浜", 0}};
  DATA x={"米原", 3};

  DataPrint(head, a);
  printf("search result: %s: %d¥n", "名古屋", DataSearch(head, a, "名古屋"));
  printf("search result: %s: %d¥n", "福岡", DataSearch(head, a, "福岡"));
```

プログラムを実行してみよう

```
  n=DataInsert(n, head, a, x);
  DataPrint(head, a);
  //DataDelete(head, a, "名古屋");
  DataDelete(head, a, 3);
  n--;
  DataPrint(head, a);

  return 0;
}
```

プログラムを実行してみよう

```
./a.out
list
新大阪
京都
名古屋
新横浜
東京
search result: 名古屋: 3
search result: 福岡: -1
```

```
list
新大阪
京都
米原
名古屋
新横浜
東京
list
新大阪
京都
米原
新横浜
東京
```

新しいデータ構造の特徴

- メリット
  - データの削除・追加がとても簡単
  - 既知の知識でプログラミングができる（高度なプログラミングが不要）
- デメリット
  - 配列というデータ構造にデータを格納したときと同様、削除があった場合、メモリの無駄が生じる
  - 追加を考えれば、逆に予め大きめのメモリを確保しなければならない