アルゴリズム論
（第３回）

岩手県立大学
Iwate Prefectural University

佐々木研（情報システム構築学講座）
講師　山田敬三
k-yamada@iwate-pu.ac.jp

リストと木構造

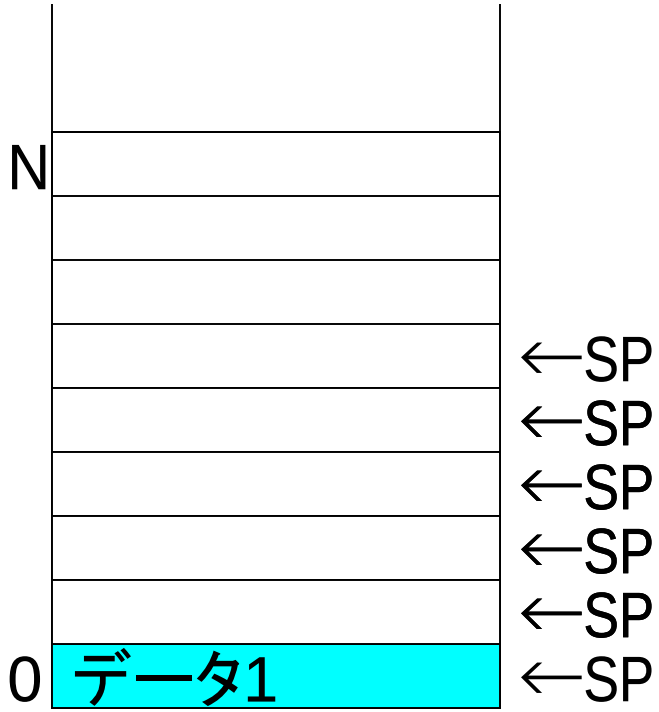
岩手県立大学
Iwate Prefectural University

リスト

- 順序付けされたデータの並び
  - 一次元配列もリストの一種
- データの追加，削除，参照，置き換えなどが行われる．
  - それぞれ関数化しておくと便利
- 操作方法に応じたデータ構造が必要

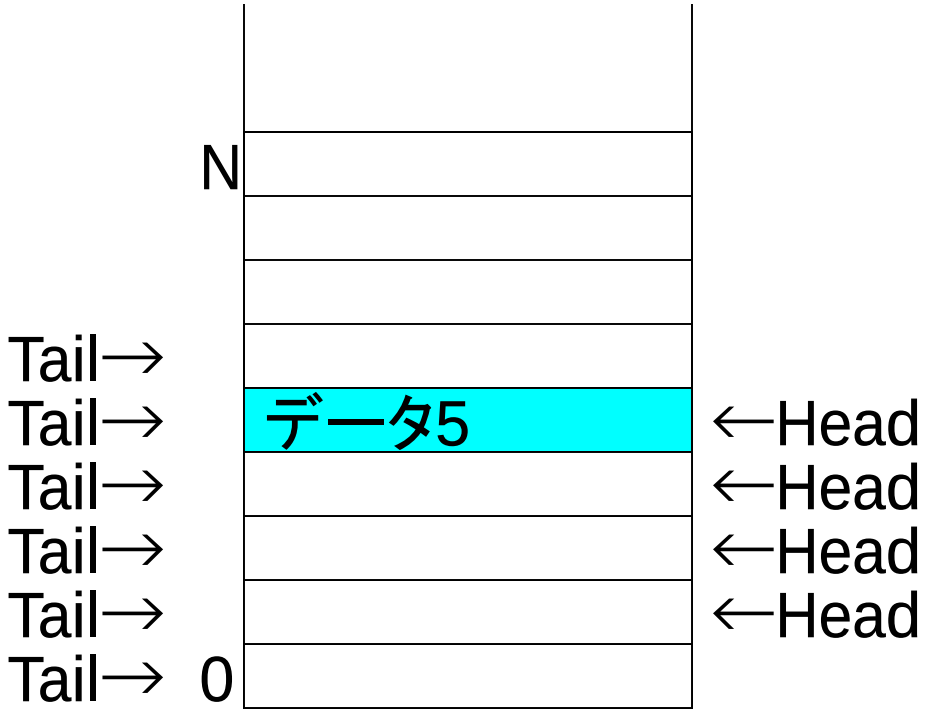
スタック

- データを順番に追加
- 最後に追加されたデータから順番に取り出す.
  - 例）食堂のトレー
- LIFO (Last In First Out)



キュー（待ち行列）

- 最初に追加したデータを最初に取り出す.
  - 例）映画館のチケット売り場
- FIFO (First In First Out)



スタックの実装

- 宣言
  - データを蓄えておくためのリスト
  - データの追加，取り出し位置を示すポインタ

```
/* データを蓄積するリスト */
float stack[STACKSIZE];
/* スタックポインタ */
int sp = 0;
```

push（データの追加）

```
int push(float data) {
  if(sp>=STACKSIZE)
    return 0;
  else {
    stack[sp++]=data;
    return 1;
  }
}  /* 返却値は、成功時1、失敗時0 */
```

pop (データの取り出し)

```
int pop(float *data) {
  if(sp<=0)
    return 0;
  else {
    *data=stack[--sp];
    return 1;
  }
}  /* 返却値は、成功時1、失敗時0 */
```

逆ポーランド記法

- 演算子を対象項の後ろに記述
  - 後置記法

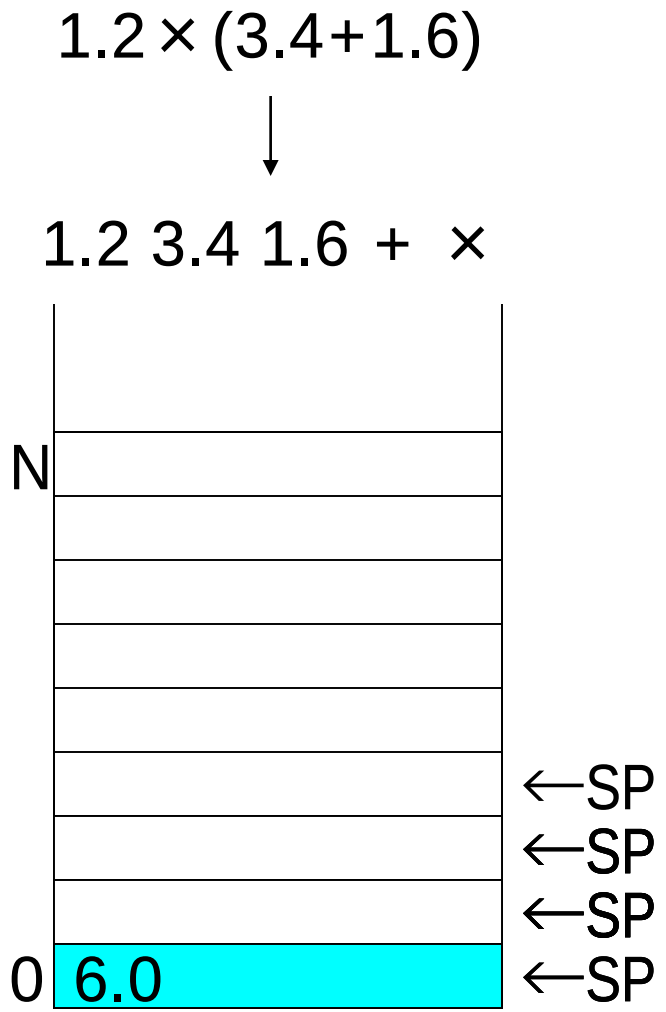
- スタック構造により実現可能
  - かっこ操作が必要ない.

$$1.2 \times (3.4+1.6)$$

$$1.2\ 3.4\ 1.6\ +\ \times$$

逆ポーランド記法の演算

- 数値なら
  - push
- 演算子なら
  - 2つ pop
  - 演算結果を push
- 最終的に残ったものが答え

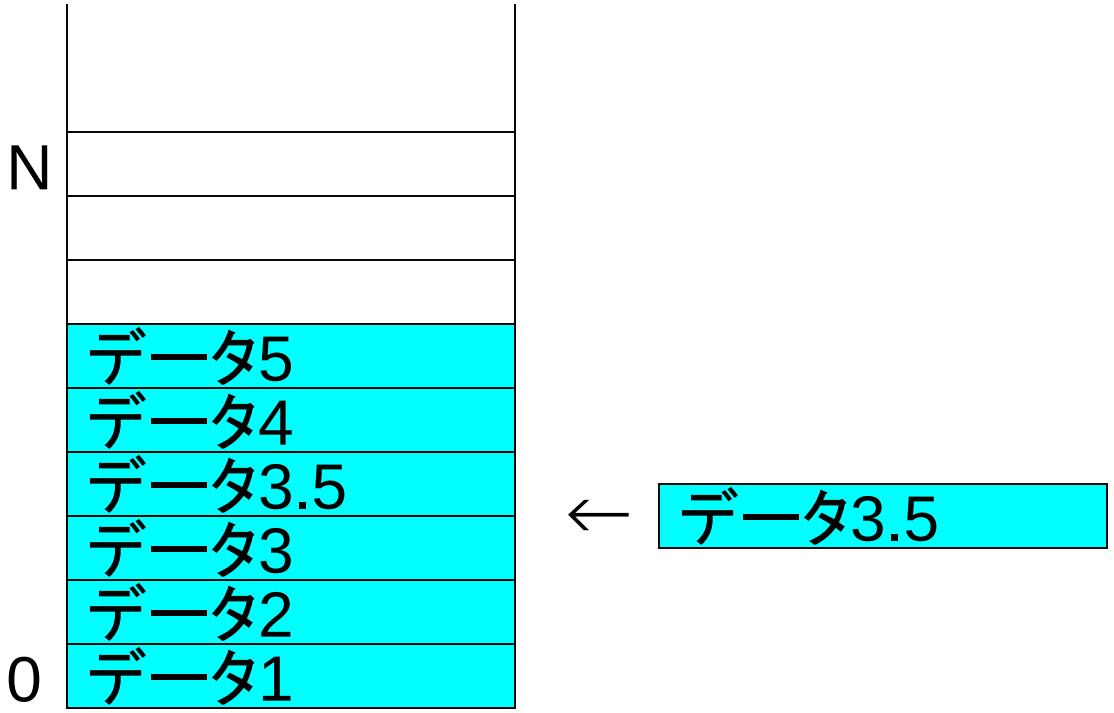


線形連結リスト

岩手県立大学
Iwate Prefectural University

配列

- リストの途中での追加，削除は困難

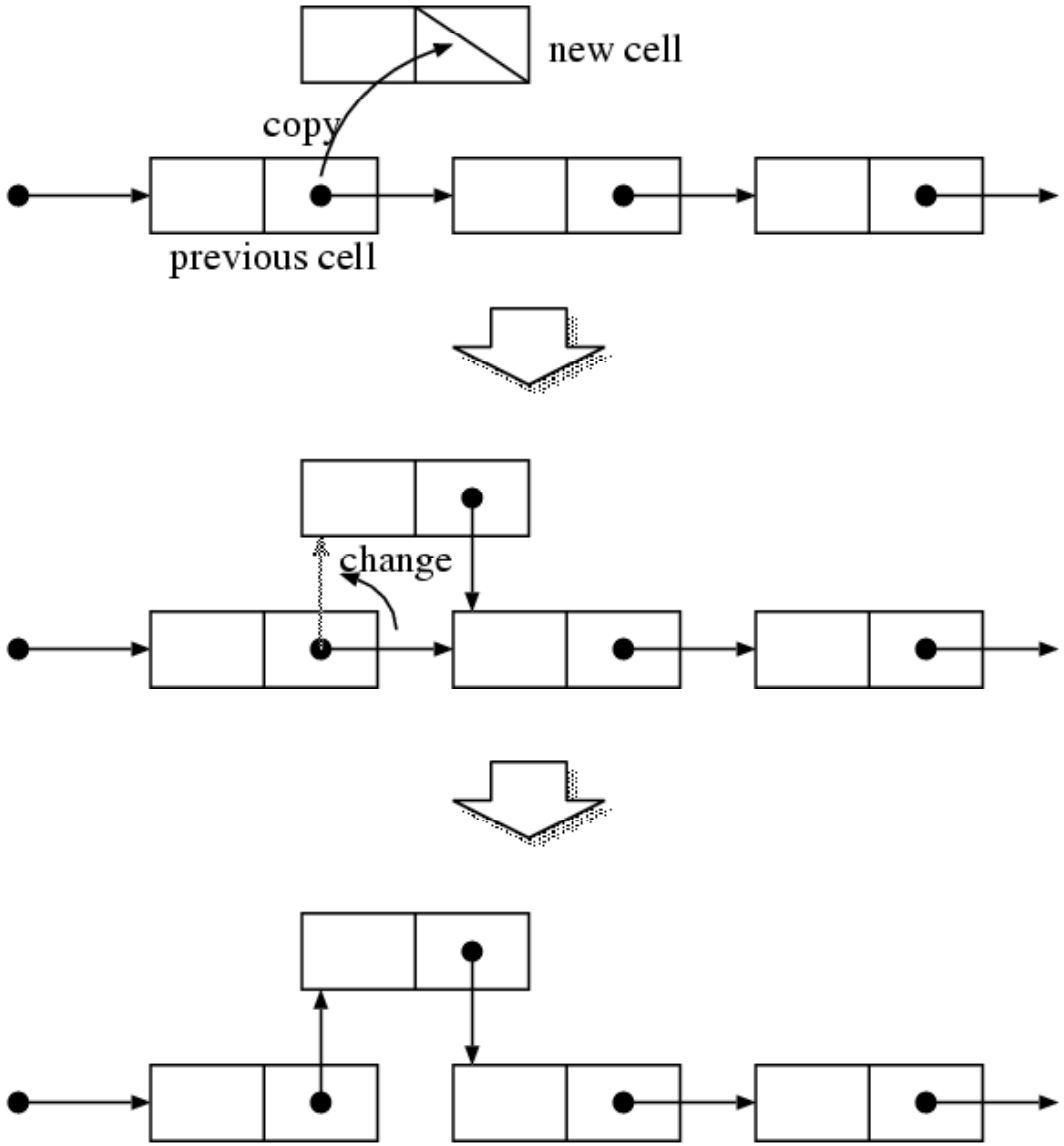


線形連結リスト (Linked List)

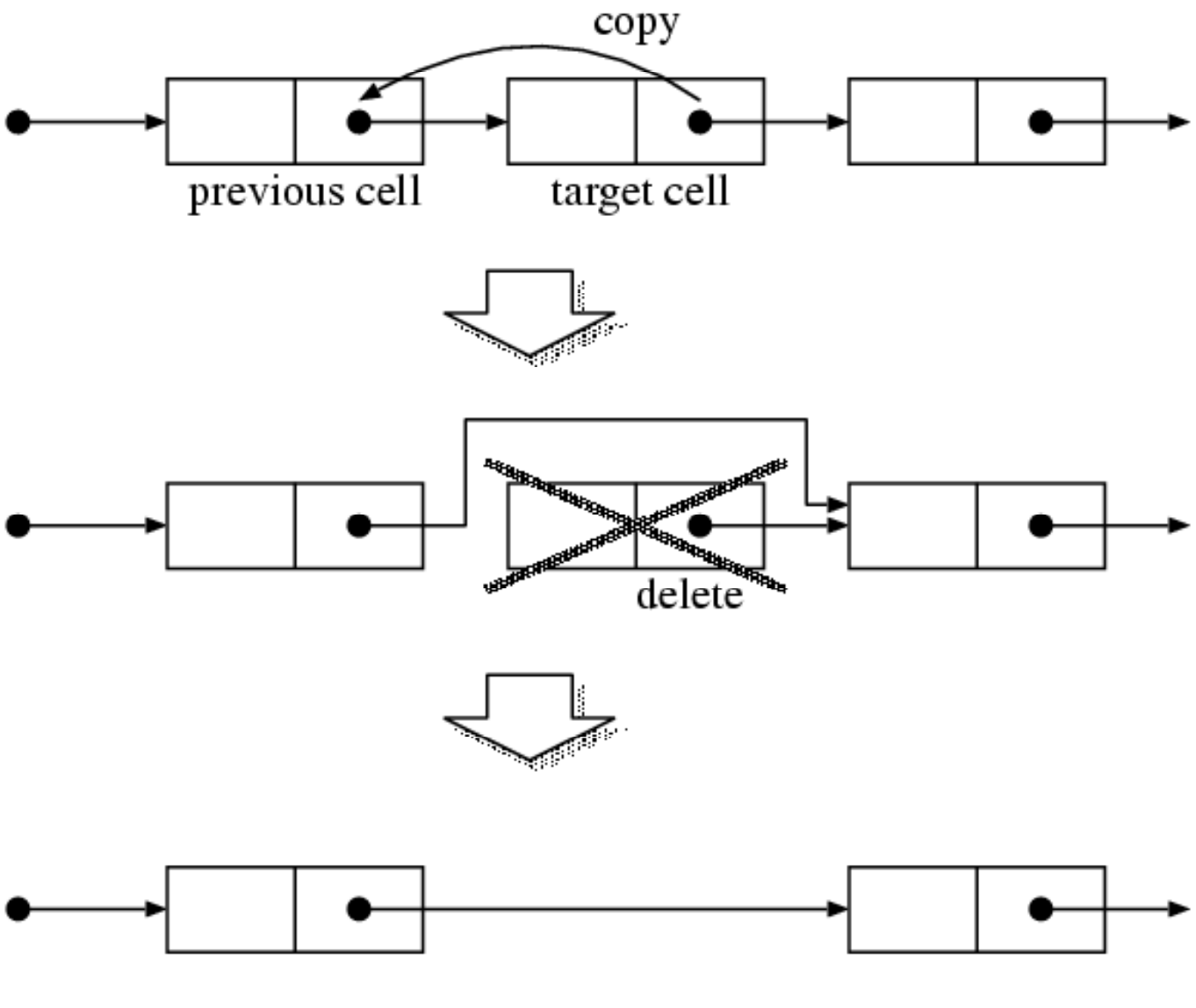
- 各要素をポインタで結合
  - 相対的に参照

- 配列の要素番号のようなインデックスはなし
  - 「何番目のデータ」という絶対的な参照は不可

セルの挿入

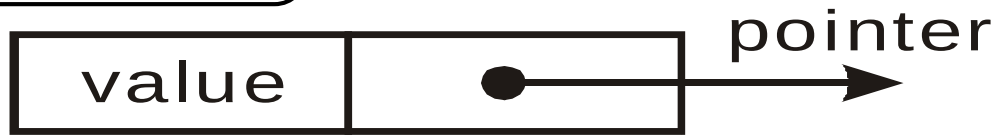


セルの削除



線形連結リスト (Linked List)

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

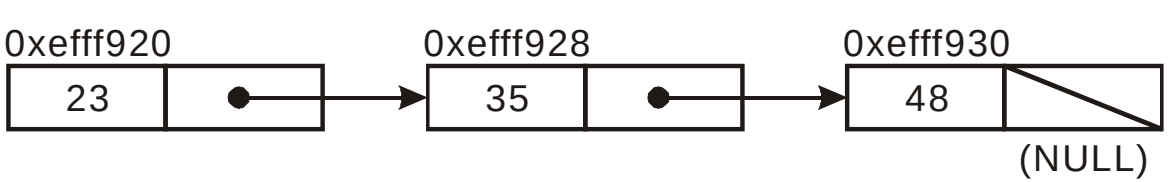


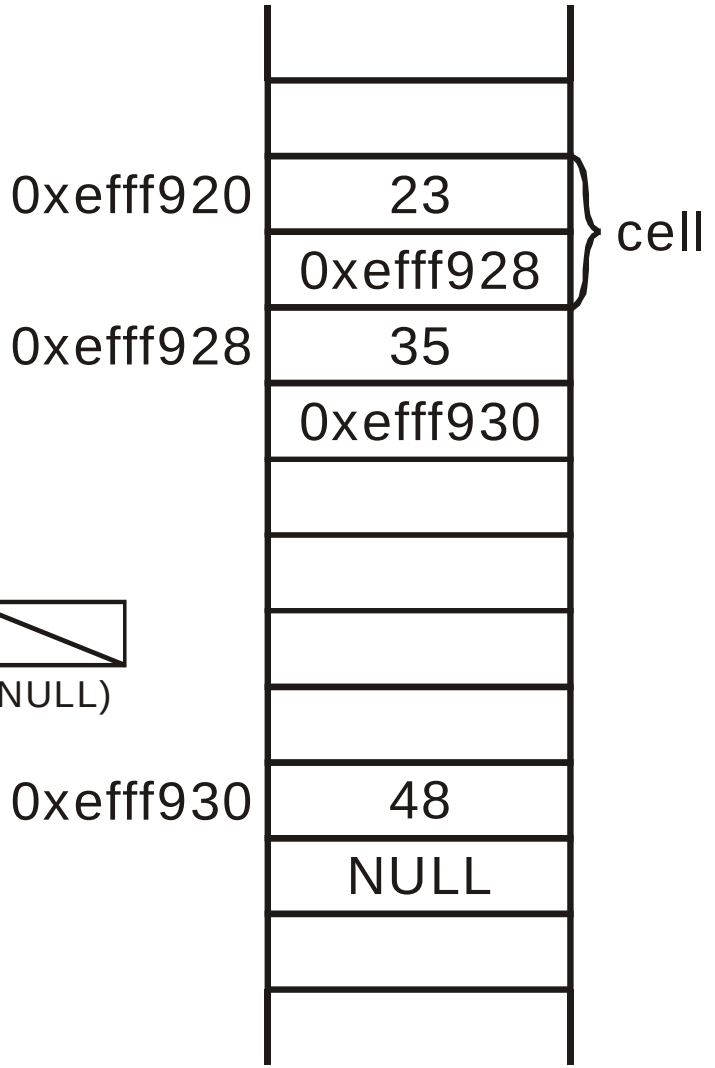
セル

- セル
  - データ，及び，次の要素の格納場所を示すポインタで構成
  - 最後のデータの，次を示すポインタは **NULL**

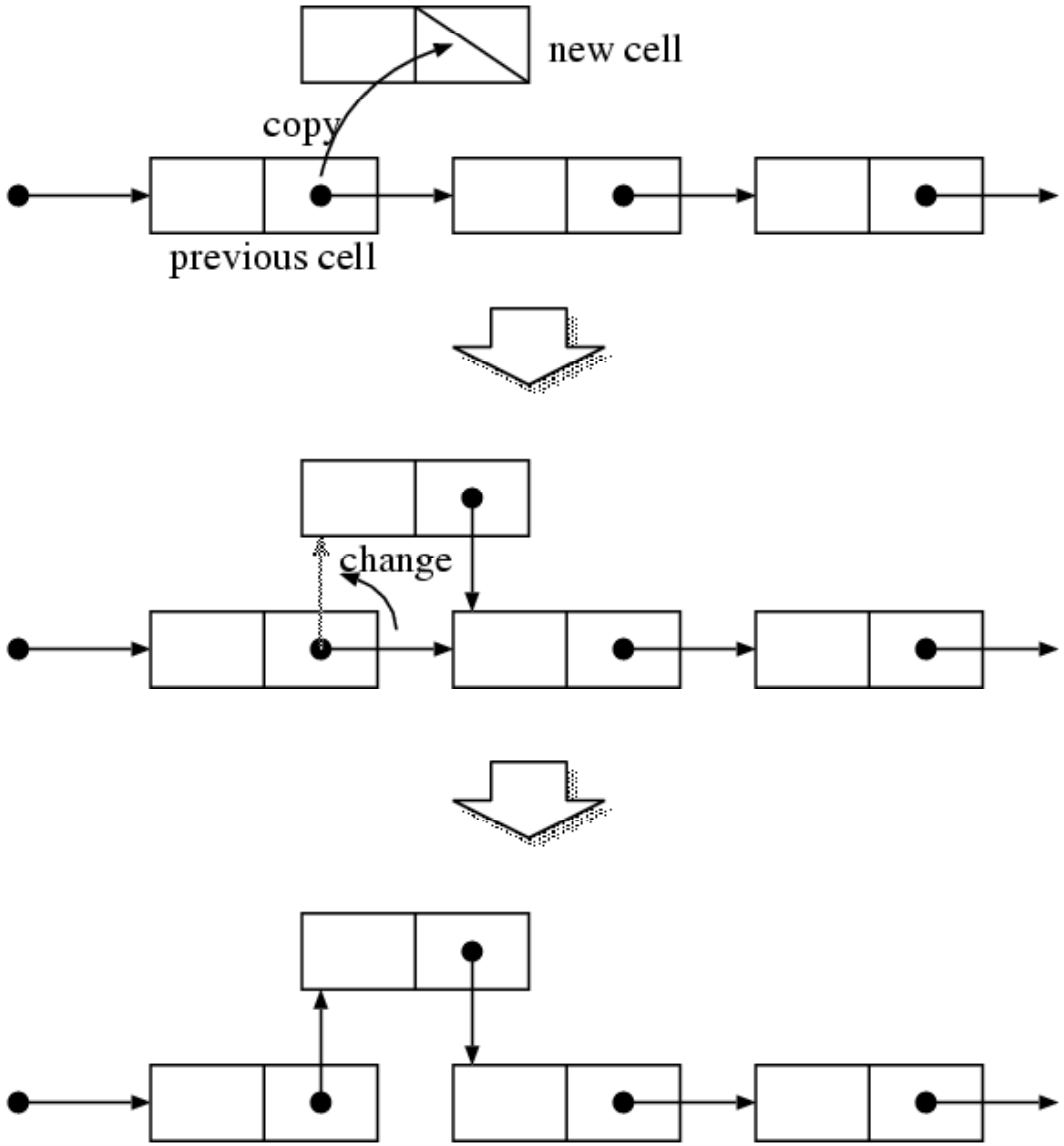
線形連結リスト (Linked List)

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





セルの挿入



セルを挿入する関数

```
void insert_cell(struct cell
  **pointer, int new_value)
```

- 挿入する位置を示すポインタへのポインタと，格納する値を引数とする.
  ※ポインタ自身を変更する必要があるため

セルを挿入する関数

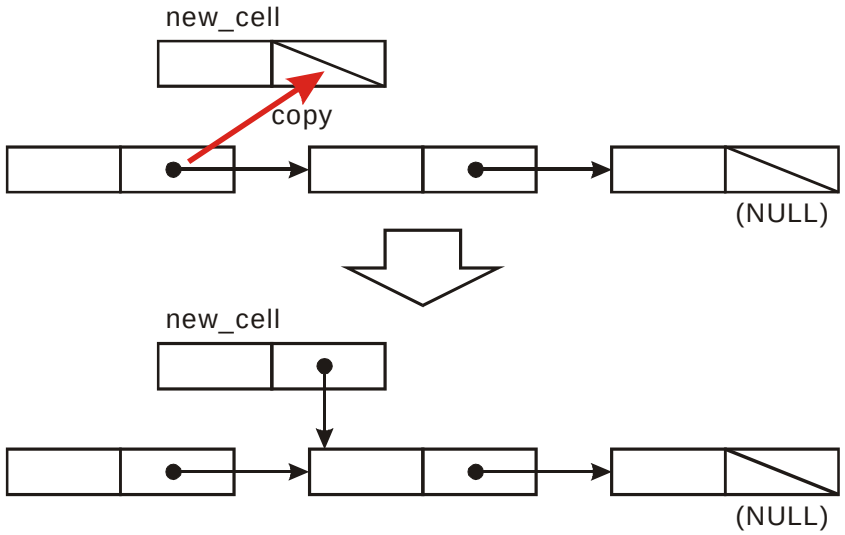
1．新たなセルを用意し，値を代入

```
new_cell=(struct cell *)malloc
  (sizeof(struct cell));
new_cell->value=new_value;
```

セルを挿入する関数

2. 新たなセルの次へのポインタに，元の場所へのポインタをコピー

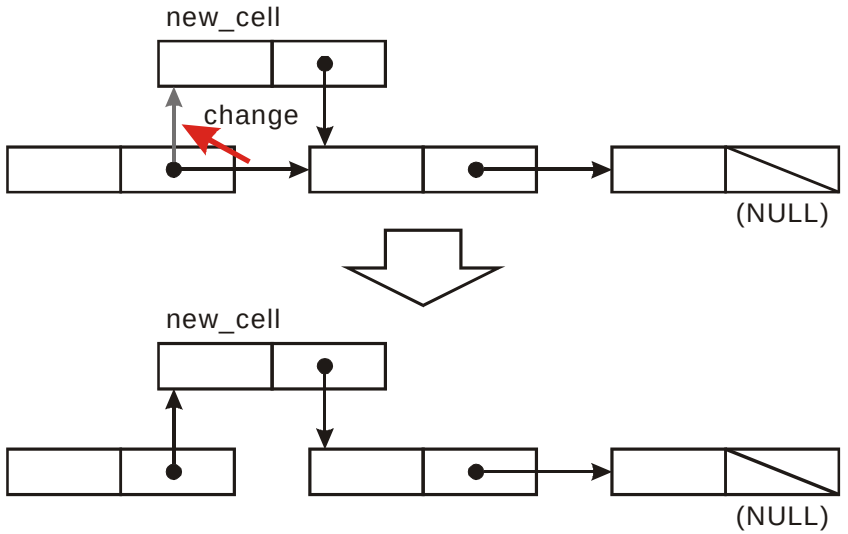
```
new_cell->next=*pointer;
```



セルを挿入する関数

3.　元のポインタの値を新たなセルに変更

```
*pointer=new_cell;
```

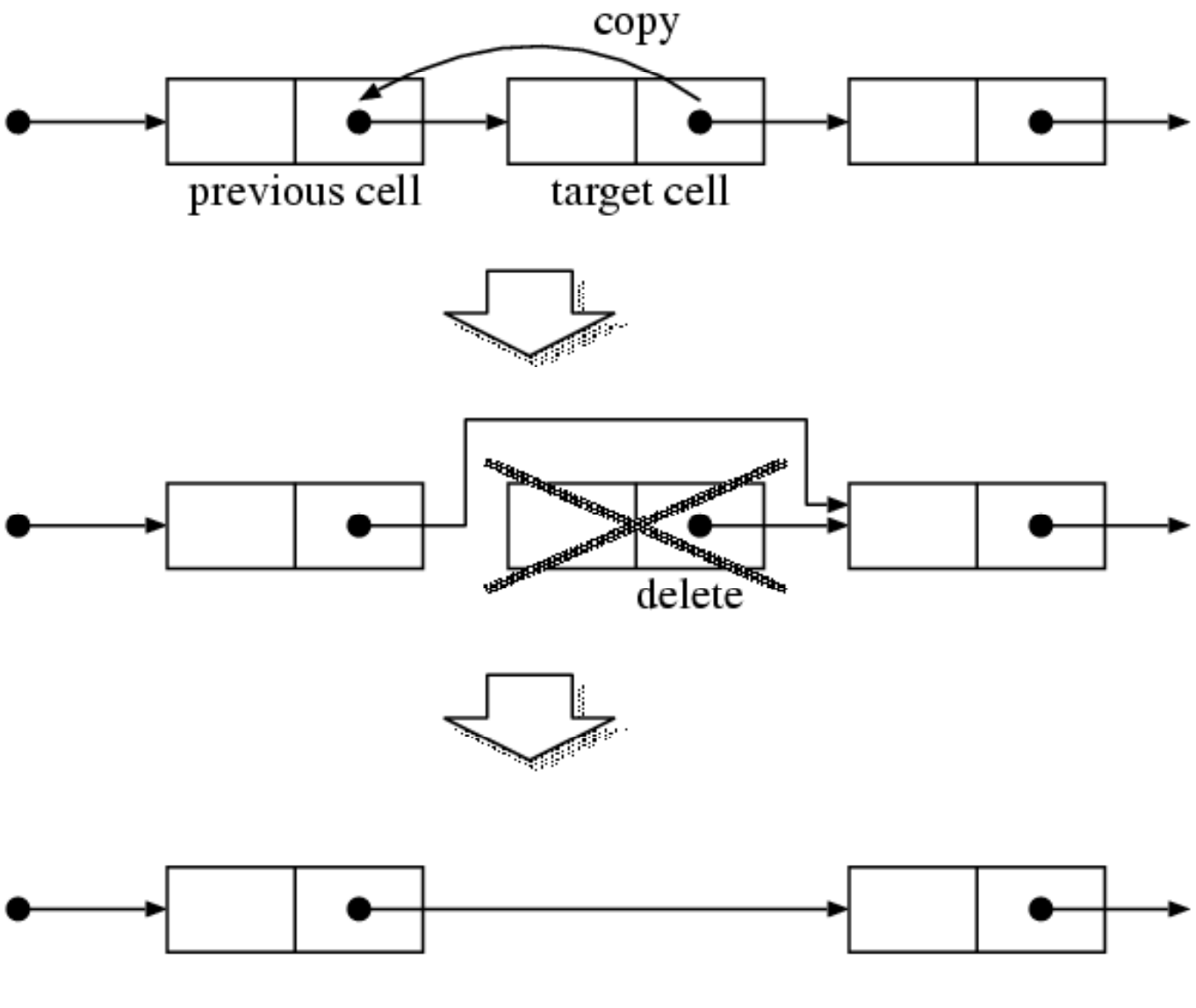


セルの挿入

```
struct cell *head=NULL, **p;
p=&head;
while(…) {
  insert_cell(p,data);
  p=&((*p)->next);
}
```

- 引数
  - 挿入場所のアドレスを格納している変数のアドレス
  - つまり，前のセルの次へのポインタが格納されている変数(**next**)のアドレス

セルの削除



セルを削除する関数

```
void delete_cell(struct cell **pointer)
```

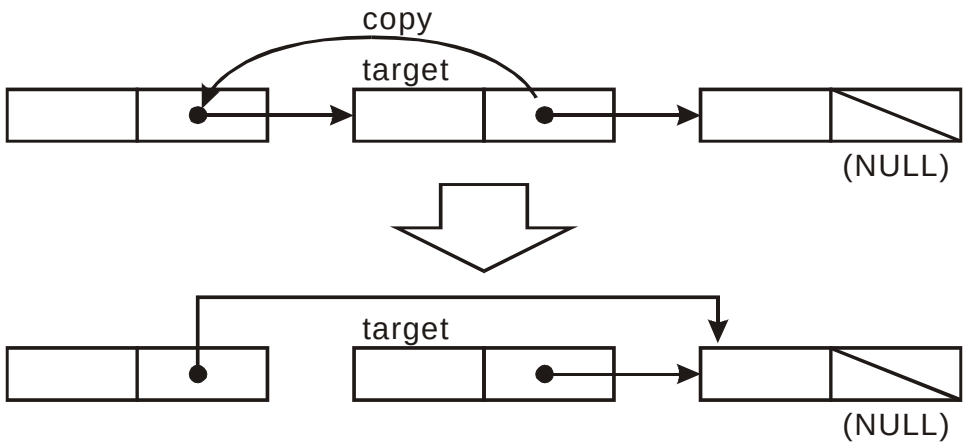
- 削除するセルへのポインタへのポインタを引数とする.
  - つまり，前のセルの，次のセルへのポインタのアドレス

※ポインタ自身を変更する必要があるため

セルを削除する関数

1. 削除したいセルへのポインタを求め，元のポインタの値を，削除するセルの後のセルに変更

```
target=*pointer;
*pointer=target->next;
```



セルを削除する関数

2.　セルを削除

```
free(target);
```