

```
// algorytm odwracania stosu (wizualizacja)
void reverse() {
    Node* prev = nullptr;
    Node* curr = topNode;
    while (curr) {
        Node* next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    topNode = prev;
}
```

*curr != nullptr*

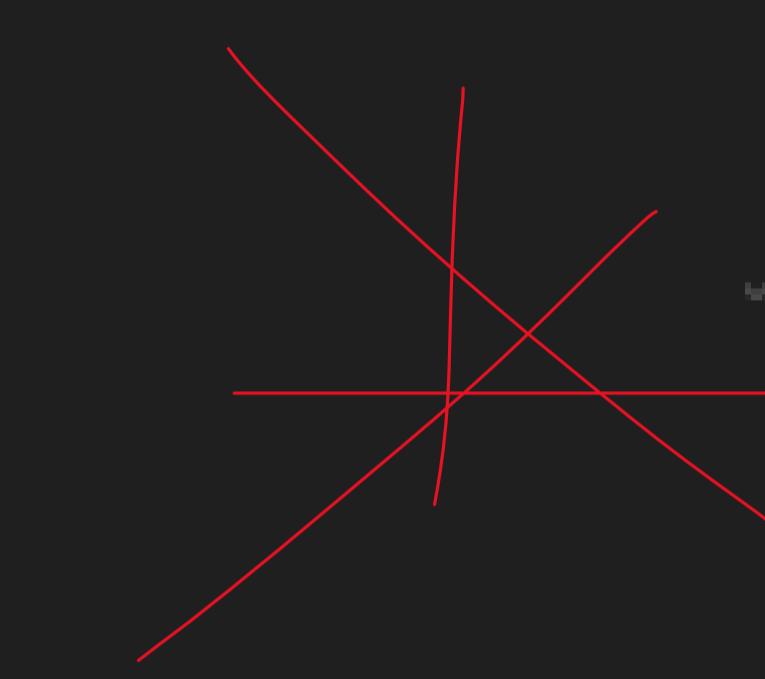
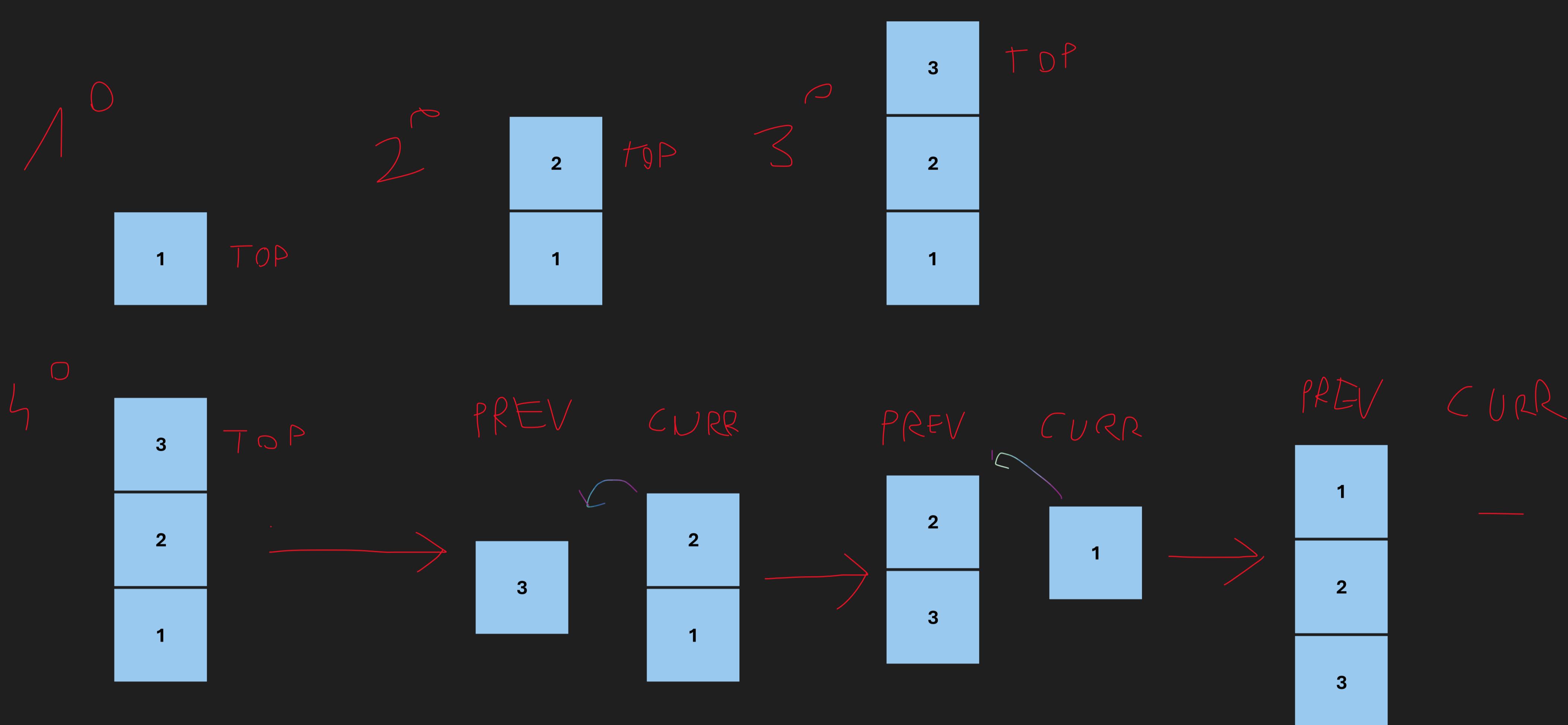
Petla skonczy sie gdy curr = nullptr  
prev wskazuje ostatni node na którym pracowaliśmy  
więc staje się 'wierzchołkiem' stosu

- 1° Zapamiętujemy kolejny node przed zmianą kierunku pointera
- 2° Odwracamy kierunek pointerów:  
-> curr wskazywał na następny node (z 3 do 2)  
-> teraz wskazuje odwrotnie (z 3 do nullptr)
- 3° prev przesuwany na aktualny curr
- 4° Przechodzimy do kolejnego node'a (zapamiętanego wcześniej w next)

```
ZADANIE 1:
Stos przed odwróceniem: Stos: 3 2 1
Rozpoczynam odwracanie stosu...
Krok 1:
prev: 3
curr: 2 1
Krok 2:
prev: 2 3
curr: 1
Krok 3:
prev: 1 2 3
curr: 1
Stos po odwróceniu:
Stos: 1 2 3
Stos po odwróceniu: Stos: 1 2 3
```

```
// Zadanie 1: Stos (listowy) - odwracanie stosu
struct Node {
    int data;
    Node* next; — Wskaźnik na kolejny element stosu
};

1 s.push(1);
2 s.push(2);
3 s.push(3);
cout << "Stos przed odwróceniem: " s.print();
4 s.reverse();
cout << "Stos po odwróceniu: " s.print();
```



```
// Zadanie 2: Stos (tablicowy) - scalanie dwóch stosów rosnących
class ArrayStack {
    int data[100]; — Tablica na elementy stosu
    int size;

    // funkcja sortująca stos - klasyczny Bubble sort
    void sortArray(int arr[], int n) {
        for (int i = 0; i < n - 1; ++i) {
            for (int j = 0; j < n - i - 1; ++j) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

Bubble sort -> porównujemy sąsiadujące elementy zamieniając je miejscami dopóki nie będzie posortowane :)

Fajna interaktywna wizja  
<https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/visualize/>



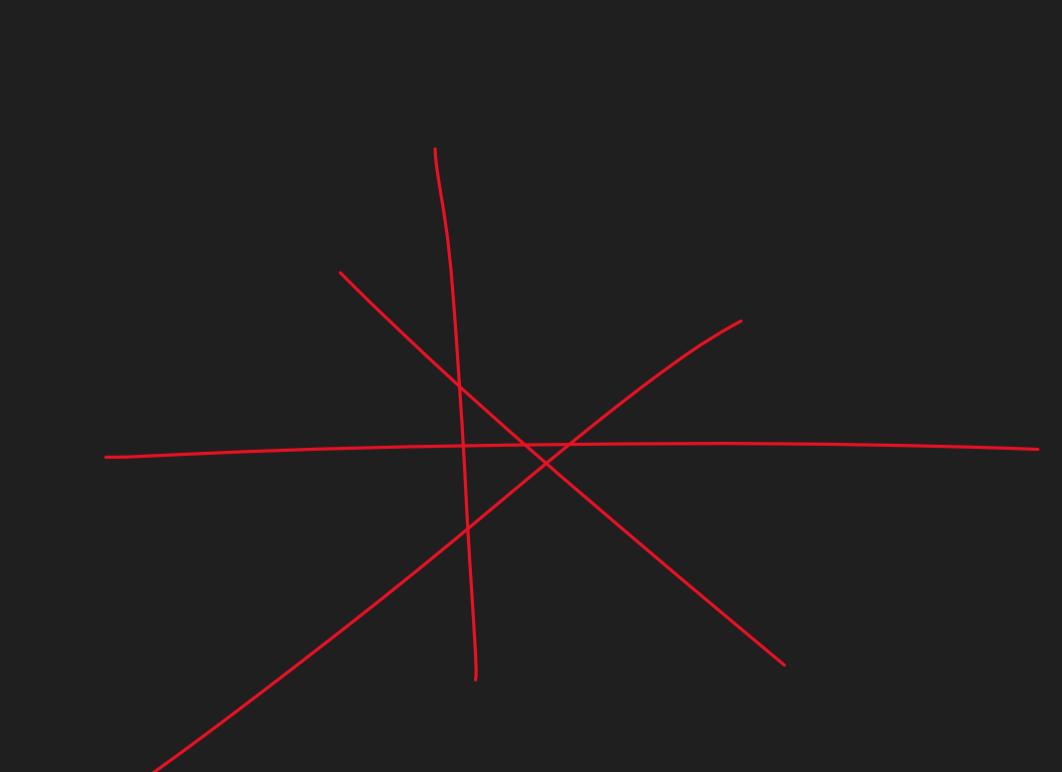
```
// Zadanie 3: Kolejka (tablicowa) liczb rzeczywistych
class FloatQueue {
    float data[100]; — tablica na floaty (max 100)
    int frontIndex, rearIndex;
}
```

Index pierwszego elem w kolejce

Index pierwszego wolnego miejsca z tyłu kolejki

```
// usuń pierwszy element z kolejki (na frontIndex) -> przesuń frontIndex do przodu
void dequeue() {
    if (frontIndex < rearIndex) ++frontIndex;
}
```

Ważne jest to że dane 'odkolejkowane' dane nie są usuwane, tylko logiczny początek kolejki zostaje przesunięty tak że slot z danymi 'logicznie' staje się wolny;



```
// Zadanie 4: Kolejka napisów (listowa) + wyszukiwanie
// FIFO - First In First Out
struct StringNode {
    char str[100]; — Tablica charów na tekst
    StringNode* next; — Pointer na następny znak w kolejce
};
```

```
rearNode->next = newNode;
```

PRZED:  
[ hello ] -> [ world ] -> nullptr  
^ rearNode

PÓŁ:

[ hello ] -> [ world ] -> [ nowy ] -> nullptr  
^ rearNode->next

Wizualizacja :)