**XOR linked list (AiSD 2021)**

Zadanie polega na implementacji listy dwukierunkowej. Klasyczna lista dwukierunkowa przechowuje w węzłach adresy następnika i poprzednika, w naszym zadaniu lista przechowuje w węzłach jedynie xor tych adresów. Pozwala to na ok. 50 zaoszczędzenie pamięci wykorzystywanej na przez struktur listy.

Określenie węzeł i element listy używane są zamiennie i oznaczają to samo.

Dodając nowy węzeł do listy zapisujemy w nim wartość xor poprzednika i następnika. Jeśli jest to pierwszy albo ostatni węzeł jako xor adresów wpisujemy odpowiednio adres następnika albo poprzednika ponieważ xor adresu z 0 (zero przyjmujemy jako wartość NULL) nie zmienia adresu. Z tego wynika, że musimy przechowywać wskaźnik na pierwszy oraz ostatni element listy, co pozwoli nam przeglądać listę w obydwu kierunkach. Chcąc przejść do kolejnego węzła w ustalonym kierunku jego adres odczytujemy jako xor wartości przechowywanych w aktualnym i wcześniej odwiedzonym węźle. Jak opisano wcześniej, w skrajnym węźle będzie to bezpośredni adres (ponieważ wartość hipotetycznego, nieistniejącego wcześniejszego węzła jest równa 0).

W przypadku usuwania węzła należy zaktualizować wartości xor poprzednika i następnika usuwanego węzła. W przypadku usuwania skrajnego węzła istnieje tylko jeden sąsiad wymagający aktualizacji.

Zadanie wymaga abyśmy dysponowali dodatkowym wskaźnikiem wskazującym aktualną wartość wykorzystywanym przez komendę ACTUAL opisaną później. W tym przypadku musimy dodatkowo przechowywać jednego z sąsiadów (następnika albo poprzednika, a najlepiej obydwu) wskaźnika wskazującego aktualną wartość. W przeciwnym przypadku nie będziemy w stanie odtworzyć ich adresów (następnika albo poprzednika elementu aktualnego). Wydawało by się, że nie dotyczy to pierwszego i ostatniego węzła ale nie jesteśmy w stanie (bez dodatkowych informacji) wywnioskować, że wskaźnik na element aktualny akurat wskazuje na jeden ze skrajnych elementów.

Lista udostępnia następujące funkcjonalności powiązane z następującymi komendami:

* ACTUAL - wartość aktualnie wskazywanego elementu. Dla pustej listy jest to NULL a po dodaniu pierwszego elementu do listy ACTUAL zwraca wartość tego elementu tak długo dopóki nie zostanie przesunięty przez inne operacje.
* NEXT - drukuje wartość następnika ACTUAL jednocześnie ustawiając na niego ACTUAL. Jeśli ACTUAL wskazuje na ostatni element kolejki jego następnikiem będzie pierwszy element kolejki.
* PREV - drukuje wartość poprzednika ACTUAL jednocześnie ustawiając na niego ACTUAL. Jeśli ACTUAL wskazuje na pierwszy element kolejki jego poprzednikiem będzie ostatni element kolejki.
* ADD\_BEG **N** - dodanie elementu z wartością **N** na początek listy.
* ADD\_END **N** - dodanie elementu z wartością **N** na koniec listy.
* ADD\_ACT **N** - dodanie elementu z wartością **N** jako **poprzednika** aktualnie wskazywanego elementu (ACTUAL).
* DEL\_BEG - usuwa z listy pierwszy węzeł.
* DEL\_END - usuwa z listy ostatni węzeł.
* DEL\_VAL **N** – usuwa z listy **wszystkie** węzły których wartość równa jest **N**.
* DEL\_ACT - usuwa z listy węzeł na który wskazuje ACTUAL, jednocześnie ustawiając ACTUAL na PREV. W przypadku kiedy PREV nie istnieje (ACTUAL był pierwszym elementem listy) ACTUAL pokazuje na ostatni element listy.
* PRINT\_FORWARD - drukuje zawartość listy od pierwszego do ostatniego elementu.
* PRINT\_BACKWARD - drukuje zawartość listy od ostatniego do pierwszego elementu.

Zachowanie komend NEXT, PREV i DEL\_ACT sygeruje działanie na kolejce cyklicznej w której (z wyjątkiem pustej kolejki) dla każdego węzła zawsze istnieje następnik i poprzednik. Kiedy taka kolejka posada tylko jeden element wartość xor tego węzła równa jest 0. Jednakże implementacja cyklicznej kolejki nie jest nakazana i nie jest zakazana. Możana w zamian zaimplementować dodatkową funkcjonalność komend NEXT i PREV. Będzie ona uruchamiana w momencie zaistnienia sytuacji wyjątkowej czyli wywołania NEXT gdy ACTUAL wskazuje na ostatni element kolejki lub PREV bądź DEL\_ACT gdy ACTUAL wskazuje na pierwszy element kolejki nie cyklicznej.   
  
Komendy DEL\_BEG, DEL\_END, DEL\_VAL i DEL\_ACT w przypadku pustej listy nic nie usuwają. W każdym z tych przypadków usunięcie aktualnie wskazywanego elementu (komenda ACTUAL) powinno owocować przesunięciem wskaźnika aktualnie wskazywanego elementu na poprzedzający go element a jeśli taki nie istnieje na ostatni element listy.   
  
Nie można używać standardowej biblioteki szablonów C++ (STL). Proszę pamiętać o odzyskiwaniu pamięci podczas usuwania elementów listy. Proszę również usunąć całą listę przed zakończeniem programu zwalniając pamięć zajmowaną przez wszystkie jej elementy.   
------------------------------------------------------------------------------------------------------------------------

The task is based on implement a bidirectional list. A classic bidirectional list stores successor and predecessor addresses in nodes, in our task the list stores only xor of these addresses in nodes. This saves approximately 50% of memory used by the list structure.

The terms list node and list element are used interchangeably and means the same.

During adding a new node to the list, we write the xor value of the predecessor and the successor to this node. If it is the first or the last node, as the xor of addresses, we write the address of the successor or the predecessor, respectively, because the address xor with 0 (we take zero as NULL) does not change the address value. From this it follows that we need to store a pointer to the first and last element of the list, which will allow us to browse the list in both directions. If we want to go to the next node in the assumed direction, we read its address as xor values stored in the current and previously visited node. As described earlier, at the boundary node this will be the direct address (since the value of a hypothetical nonexistent earlier node is 0).

In the case of deleting a node, updating the xor values of the deleted node's predecessor and successor is required. When deleting an boundary node, there is only one neighbor that needs to be updated.

The task requires that we have an additional pointer showing the current value used by the ACTUAL command described later. In this case, we also need to keep one of the neighbors (successor or predecessor, preferably both) of pointer showing the current value. Otherwise, we will not be able to restore their addresses (successor or predecessor of the current element). In this case, we must additionally store one of the neighbors (successor or predecessor, preferably both), otherwise we will not be able to restore their addresses. It would seem that this does not apply to the first and last node, but we are not able to conclude (without additional information) that the index to the current element just points to one of the boundary elements.

The list provides the following functionalities related to the following commands:

* ACTUAL - value of the currently pointed element. For an empty list it is NULL and after adding the first element to the list ACTUAL returns the value of that element as long as it is not moved by other operations.
* NEXT - prints the value of the ACTUAL successor setting it ACTUAL at the same time. If ACTUAL points to the last element of the queue, its successor will be the first element of the queue.
* PREV - prints the value of the predecessor ACTUAL setting it ACTUAL at the same time. If ACTUAL points to the first element of a queue, its predecessor will be the last element of the queue.
* ADD\_BEG **N** - adding an element with value **N** to the beginning of the list.
* ADD\_END **N** - adding an element with value **N** to the end of the list.
* ADD\_ACT **N** - adding an element with the value of **N** as a **predecessor** of the currently selected element (ACTUAL).
* DEL\_BEG - removes the first node from the list.
* DEL\_END - removes the last node from the list.
* DEL\_VAL **N** - removes from the list **all** nodes whose value is equal to **N**.
* DEL\_ACT - removes from the list the node that ACTUAL points to, simultaneously setting ACTUAL to PREV. In the case of the PREV does not exist (ACTUAL was the first element of the list) ACTUAL shows the last element of the list.
* PRINT\_FORWARD - prints the contents of the list from the first to the last element.
* PRINT\_BACKWARD - prints the contents of the list from the last to the first element.

The behavior of the NEXT, PREV and DEL\_ACT commands suggests action on a circular queue in which (except for an empty queue) there is always a successor and a predecessor for each node. When such a queue has only one element, the xor value of this node is 0. However, the implementation of the circular queue is not required and is not forbidden. Instead, it is possible to implement additional functionality of the NEXT and PREV commands. It will be run when an exceptional situation occurs, i.e. NEXT call when ACTUAL points to the last element of the queue, or PREV or DEL\_ACT when ACTUAL points to the first element of a non-cyclic queue.   
  
DEL\_BEG, DEL\_END, DEL\_VAL, and DEL\_ACT commands for an empty list do not remove anything. In each of these cases, removing the currently pointed element (ACTUAL command) should result in moving the currently pointed element pointer to the preceding element, and if it does not exist, to the last element of the list.   
  
Standard C ++ Template Library (STL) cannot be used. Please remember to free memory when deleting list elements. Please also delete the entire list before exiting the program, freeing up all memory occupied by all its elements.   
------------------------------------------------------------------------------------------------------------------------

**Wejście**

Pewna liczba komend uruchamiających określone funkcjonalności na liście.

**Wyjście**

Wyniki działania odpowiednich komend na liście charakteryzującej się pewnym stanem. Początkowo lista jest pusta a późniejszy jej stan zależy od kolejno wywoływanych komend. Niektóre komendy nie generują żadnego wyjścia np. (ADD\_BEG, DEL\_ACT) ale mają wpływ na stan listy i na kolejne komendy które wyświetlają pewne informacje np. (ACTUAL, PRINT\_FORWARD).   
------------------------------------------------------------------------------------------------------------------------

**Input**

A number of commands that run specific functionalities in the list.

**Output**

The results of the operation of the appropriate commands on the list with a certain state. Initially, the list is empty and its later status depends on the commands that are previously invoked. Some commands do not generate any output, e.g. (ADD\_BEG, DEL\_ACT) but affect the state of the list and other commands that display certain information, e.g. (ACTUAL, PRINT\_FORWARD).   
------------------------------------------------------------------------------------------------------------------------

**Przykład/Example**

**Wejście1/Input1**

ADD\_BEG 1

ADD\_BEG 2

ADD\_BEG 3

ADD\_END 4

PRINT\_FORWARD

PRINT\_BACKWARD

**Wyjście1/Output1**

3 2 1 4

4 1 2 3

**Wejście2/Input2**

ACTUAL

ADD\_BEG 1

ACTUAL

ADD\_BEG 2

ADD\_BEG 3

ADD\_END 4

PRINT\_FORWARD

PRINT\_BACKWARD

ACTUAL

NEXT

ACTUAL

NEXT

NEXT

PREV

ACTUAL

PREV

PREV

**Wyjście2/Output2**

NULL

1

3 2 1 4

4 1 2 3

1

4

4

3

2

3

3

4

1

**Wejście3/Input3**

ACTUAL

NEXT

PREV

ADD\_BEG 1

ACTUAL

NEXT

PREV

DEL\_BEG

ACTUAL

NEXT

PREV

ADD\_BEG 2

ACTUAL

NEXT

PREV

DEL\_END

ACTUAL

NEXT

PREV

ADD\_END 3

ACTUAL

NEXT

PREV

DEL\_BEG

ACTUAL

NEXT

PREV

**Wyjście3/Output3**

NULL

NULL

NULL

1

1

1

NULL

NULL

NULL

2

2

2

NULL

NULL

NULL

3

3

3

NULL

NULL

NULL

**Wejście4/Input4**

ADD\_END 1

ADD\_BEG 2

ADD\_END 3

ADD\_BEG 4

PRINT\_FORWARD

DEL\_BEG

PRINT\_FORWARD

DEL\_END

PRINT\_FORWARD

ACTUAL

NEXT

ADD\_ACT 5

PRINT\_FORWARD

ACTUAL

NEXT

ADD\_ACT 6

PRINT\_FORWARD

ACTUAL

NEXT

DEL\_ACT

PRINT\_FORWARD

**Wyjście4/Output4**

4 2 1 3

2 1 3

2 1

1

2

5 2 1

2

1

5 2 6 1

1

5

2 6 1

**Wejście5/Input5**

ADD\_END 1

ADD\_BEG 2

ADD\_END 3

ADD\_BEG 4

PRINT\_FORWARD

DEL\_VAL 3

PRINT\_FORWARD

DEL\_VAL 2

PRINT\_FORWARD

ADD\_END 1

ADD\_BEG 2

ADD\_END 3

ADD\_BEG 4

PRINT\_FORWARD

DEL\_VAL 4

PRINT\_FORWARD

**Wyjście5/Output5**

4 2 1 3

4 2 1

4 1

4 2 4 1 1 3

2 1 1 3

**Wejście6/Input6**

DEL\_BEG

DEL\_END

DEL\_ACT

ACTUAL

NEXT

PREV

PRINT\_FORWARD

ADD\_BEG 42

ACTUAL

NEXT

PREV

PRINT\_FORWARD

DEL\_ACT

ACTUAL

NEXT

PREV

PRINT\_FORWARD

ADD\_BEG 4

ADD\_END 3

ADD\_END 5

ADD\_BEG 2

PRINT\_FORWARD

ACTUAL

DEL\_ACT

ACTUAL

DEL\_ACT

ACTUAL

DEL\_ACT

ACTUAL

DEL\_ACT

ACTUAL

**Wyjście6/Output6**

NULL

NULL

NULL

NULL

42

42

42

42

NULL

NULL

NULL

NULL

2 4 3 5

4

2

5

3

NULL

**Wejście7/Input7**

ADD\_END 1

ADD\_END 2

ADD\_BEG 3

ADD\_ACT 4

NEXT

PREV

ACTUAL

PREV

DEL\_BEG

DEL\_BEG

DEL\_BEG

DEL\_BEG

ADD\_END 1

ADD\_END 2

ADD\_BEG 3

ADD\_ACT 4

PREV

ACTUAL

NEXT

NEXT

DEL\_END

DEL\_END

DEL\_END

DEL\_END

ADD\_END 1

ADD\_END 2

ADD\_BEG 3

ADD\_ACT 4

ACTUAL

PREV

NEXT

NEXT

DEL\_ACT

DEL\_ACT

DEL\_ACT

DEL\_ACT

ADD\_END 1

ADD\_END 2

ADD\_BEG 3

ADD\_ACT 4

ACTUAL

NEXT

NEXT

NEXT

DEL\_VAL 1

DEL\_VAL 3

DEL\_VAL 2

DEL\_VAL 4

ADD\_END 1

ADD\_END 2

ADD\_BEG 3

ADD\_ACT 4

ACTUAL

PREV

PREV

PREV

**Wyjście7/Output7**

2

1

1

4

4

4

1

2

1

4

1

2

1

2

3

4

1

4

3

2