Struktury danych

LinkedList

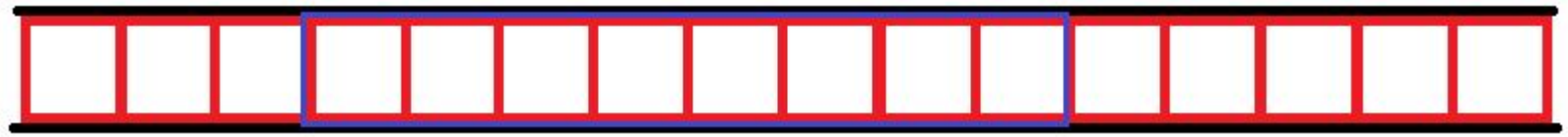
Wstęp

W tej prezentacji omówimy kolejną ze zbioru struktur danych w podstawowej bibliotece Javy. Omówimy strukturę LinkedList, wraz z szczegółowym opisem i analizą podstawowych instrukcji tej struktury.

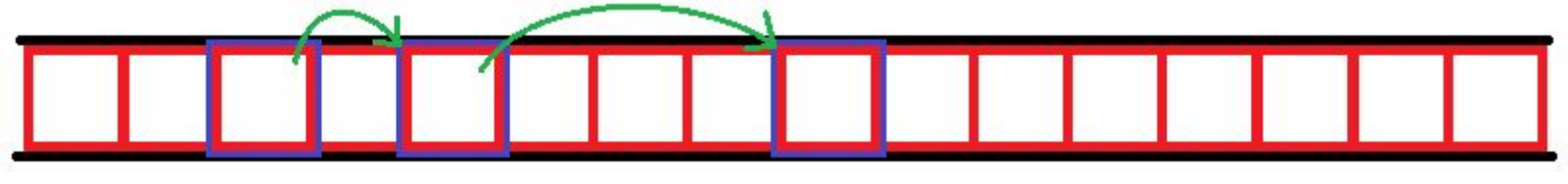
Struktury danych w Javie - LinkedList

Podczas omawiania podejmiemy się analizy i porównania między sobą wszystkich omawianych operacji między implementacją ***LinkedList***, oraz wcześniej omówioną implementacją ***ArrayList.***

Przypomnijmy sobie strukturę ArrayList, oraz to jak wyglądała w pamięci. Lista tablicowa (ArrayList) jest ciągłym blokiem danych w pamięci i wygląda następująco:

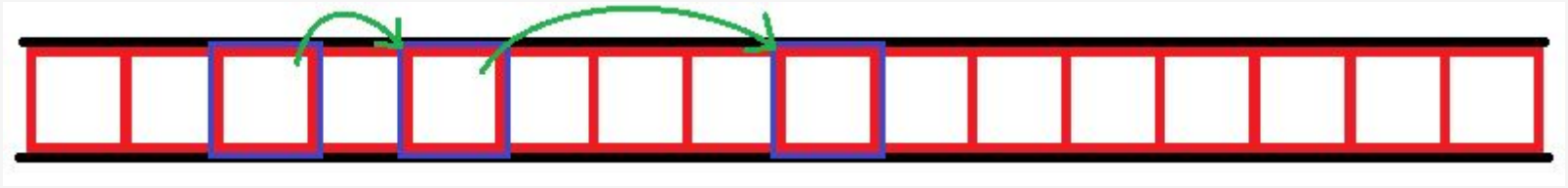


Pamięć w LinkedList jest zbudowana inaczej. Aby uniknąć problemu realokacji danych została ona zaprojektowana jako rozproszona struktura:

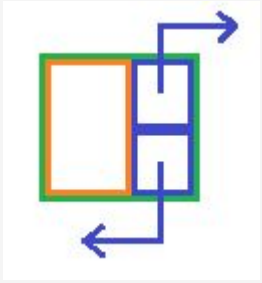


Struktury danych w Javie - LinkedList

Co oznacza rozproszoność LinkedList?

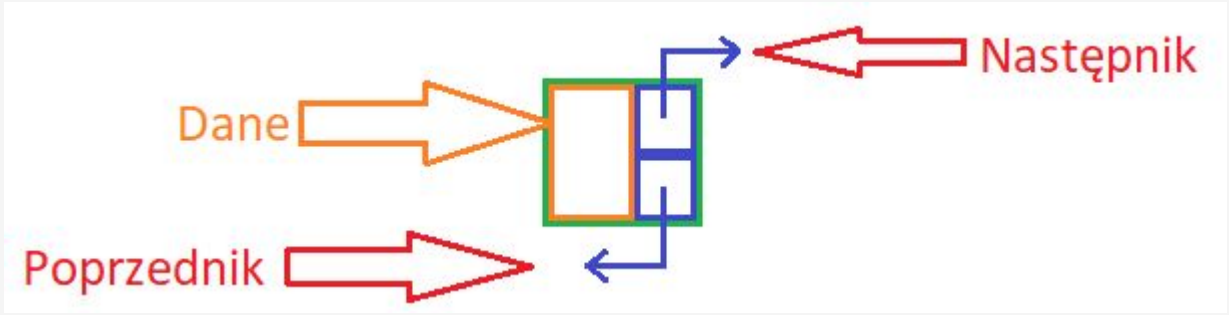


Oznacza, że dane nie znajdują się w ciągłym bloku. Oznacza to również, że nie jesteśmy w stanie przejść z elementu do elementu po kolei w taki sposób jak w ArrayList. Skupmy się najpierw na pojedyńczym węźle(węzłem będą nazywane komórki w liście) w LinkedList.



Powyżej znajduje się uproszczony model węzła listy.

Struktury danych w Javie - LinkedList



Każda komórka w LinkedList składa się z 3 pól. Posiada pole na dane, następnika i porzednika.

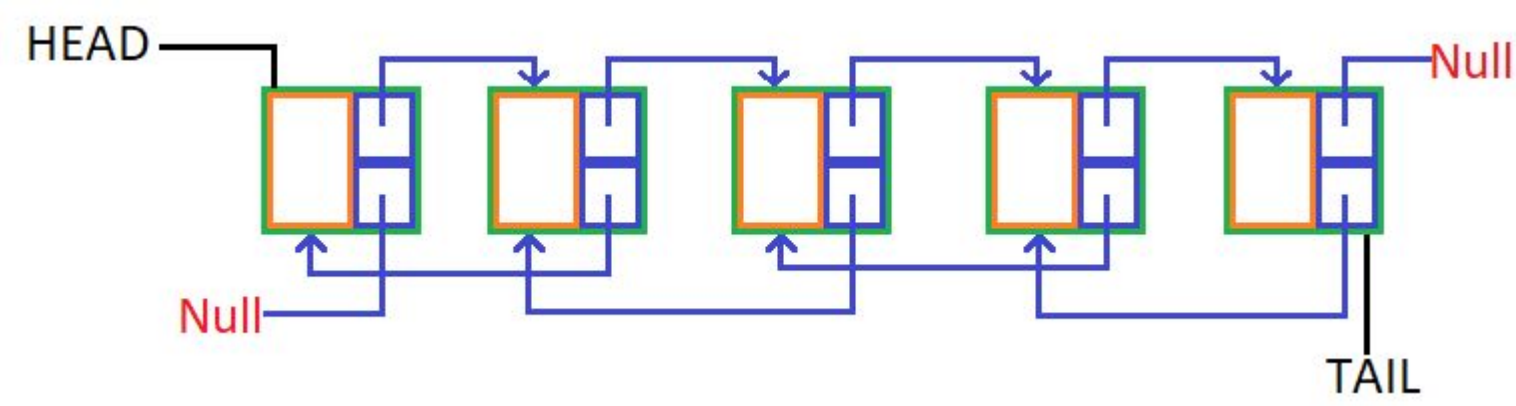
Poprzednik to element występujący przed obecnym węzłem. Następnik to element który występuje po obecnym węźle. Pole z danymi przechowuje wartość komórki. Dane będą takiego samego typu jak typ generyczny listy.

Podczas pracy z tego typu listą będziemy pracowali na listach jednokierunkowych, czyli takich, które mają informacje o następnikach, ale nie koniecznie o poprzednikach.

Zaprezentowany model jest modelem węzła w liście dwukierunkowej.

Struktury danych w Javie - LinkedList

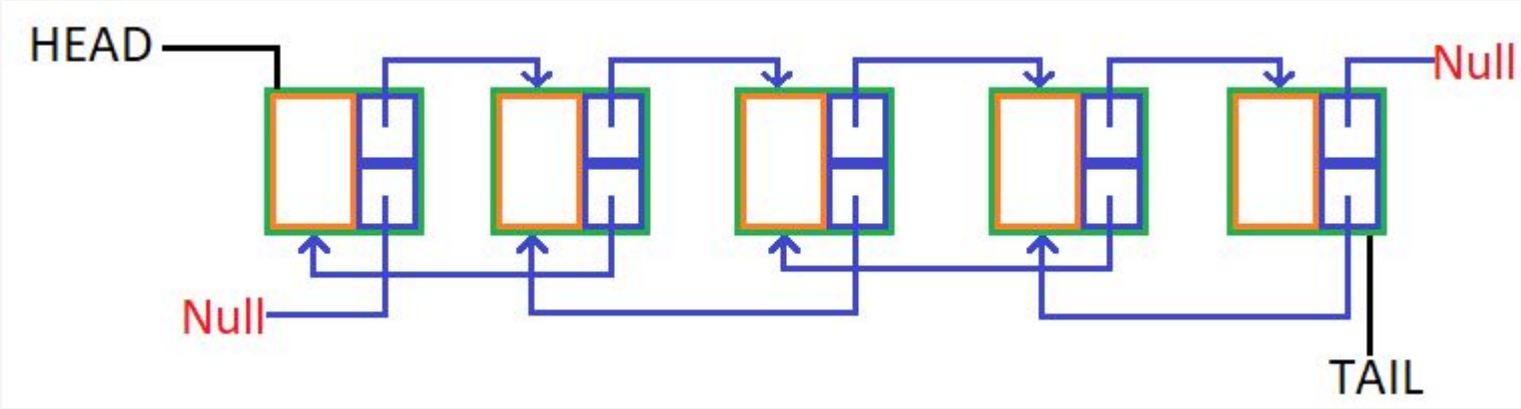
Tak więc w modelu uproszczonym elementy będą powiązane podobnie, jak na poniższym rysunku.



Elementy listy powiązane są ze sobą poprzez wskaźniki (referencje). Instancja klasy LinkedList posiada referencje na początek i koniec listy. Pierwszy element posiada poprzednik ustawiony na ***null.*** Ostatni element posiada następnik ustawiony na ***null***. Jedynym wyjątkiem od tej reguły jest lista cykliczna, w której ostatni element posiada wskaźnik na pierwszy, a pierwszy na ostatni (wstecz) – co czyni z listy zamknięte koło referencji.

Struktury danych w Javie - LinkedList

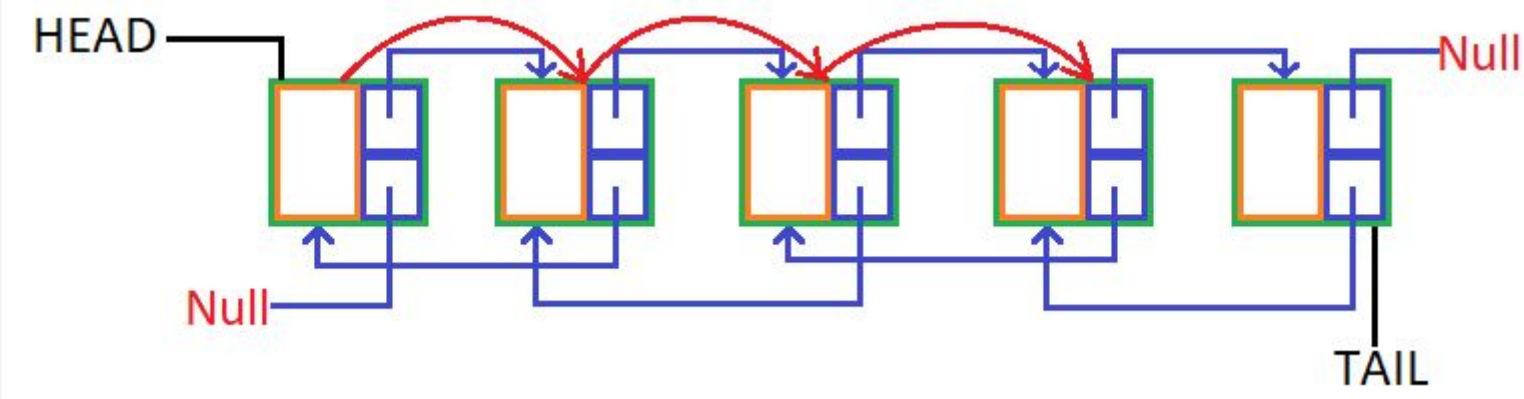
Domyślnie implementacje tego typu listy posiadają wskaźniki na HEAD i TAIL, czyli na pierwszy oraz ostatni element.



Jest to pewnego rodzaju optymalizacja. Przeanalizujmy przechodzenie między węzłami.

Struktury danych w Javie - LinkedList

Jeśli chcemy przejść do elementu o indeksie 3, to ile operacji musimy wykonać?

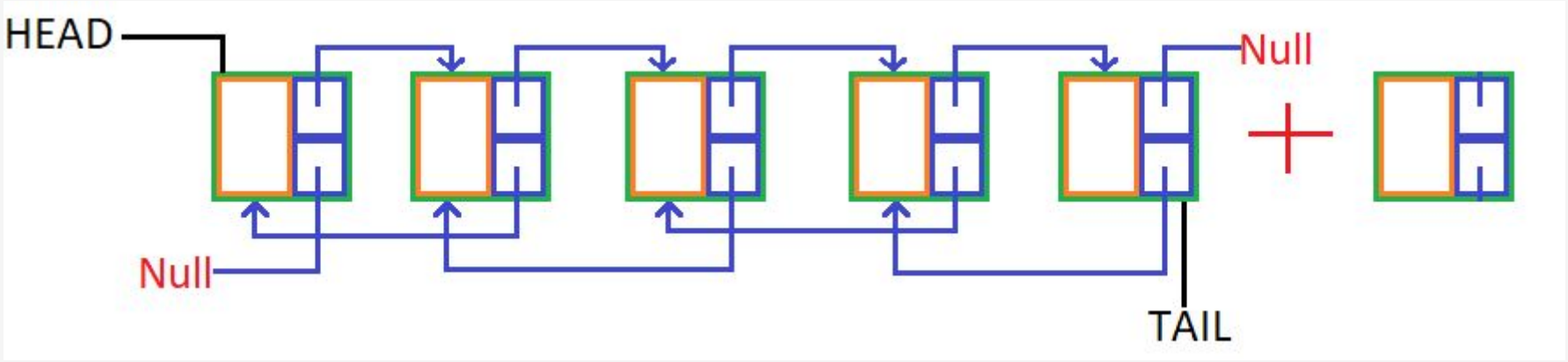


Na powyższym rysunku zaprezentowane zostało przejście z węzła 0 do węzła o numerze 3. Jaki jest czas dostępu do ***i-tego*** elementu – niestety jest to czas liniowy, zatem czas dostępu do tej komórki będzie uzależniony od numeru węzła, do którego próbujemy uzyskać dostęp.

Złożoność tej operacji będzie uzależnione od indeksu, dlatego pesymistycznie, jeśli będziemy próbowali uzyskać dostęp do i-tej komórki, będziemy mówili dostępie o złożoności: ***O(n)*** – liniowy czas dostępu.

Struktury danych w Javie - LinkedList

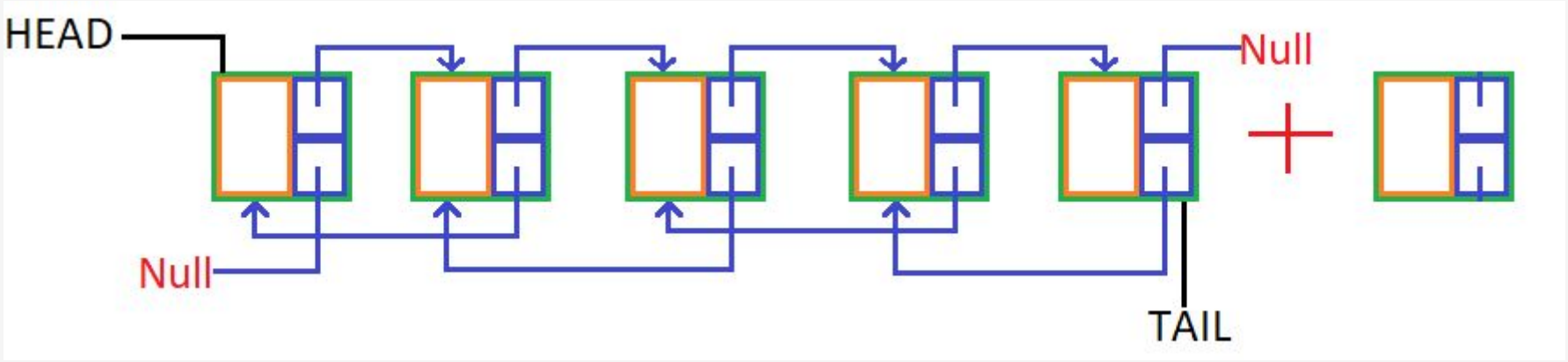
Przeanalizujmy dodawanie do tego typu kolekcji.



Tutaj powinno się pojawić pytanie – czy posiadamy wskaźnik na koniec? Jeśli nie, to musimy go najpierw odnaleźć, co sprawi że czas wykonania tej operacji znacznie się wydłuży. A co jeśli mamy ten wskaźnik? W tej sytuacji operacje które musimy wykonać, to podmiana referencji. Musimy ustawić referencje następnika elementu ***TAIL*** na nowy obiekt, oraz ustawienie poprzednika nowego elementu na ***TAIL,*** oraz przestawić referencję listy ***TAIL*** na nowy obiekt. Jaki jest czas tej operacji? Czy jest uzależniony od rozmiaru danych? Nie, stąd złożoność tej operacji dla LinkedList wynosi: **1**.

Struktury danych w Javie - LinkedList

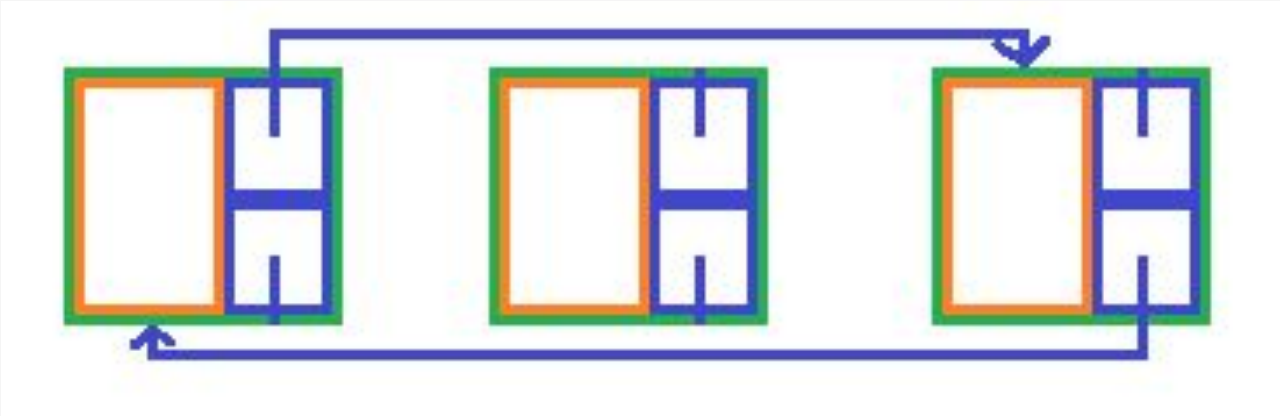
A co jeśli musimy ten sam element dodać w środku?



Niestety operacja wymaga odnalezienia miejsca, a raczej elementu na którego miejsce musimy wstawić nowy element. Niemniej jednak jeśli znamy miejsce w którym wstawiamy element, to czas operacji będzie również stały, ponieważ zawsze będzie to podmiana kilku referencji.

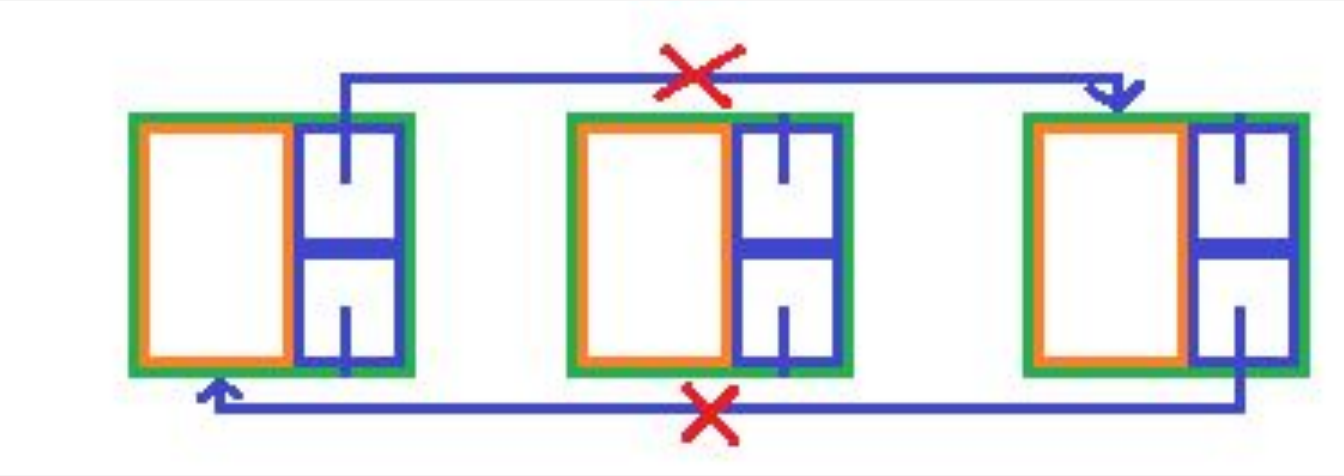
Struktury danych w Javie - LinkedList

Wstawienie elementu pomiędzy:



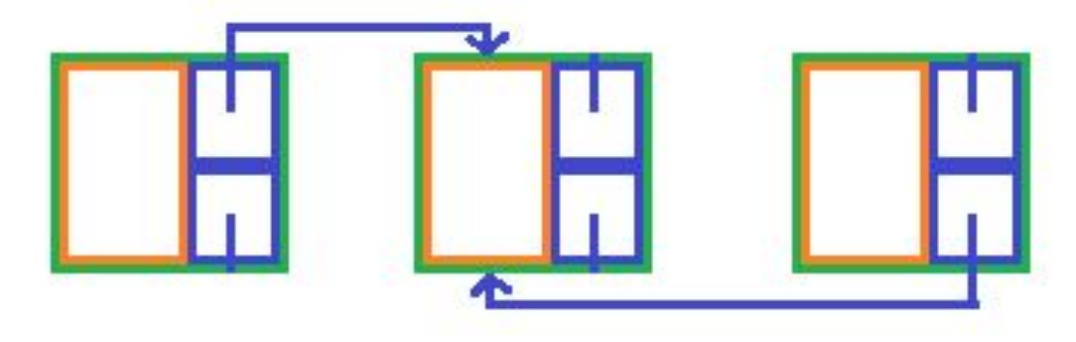
Struktury danych w Javie - LinkedList

Wstawienie elementu pomiędzy:

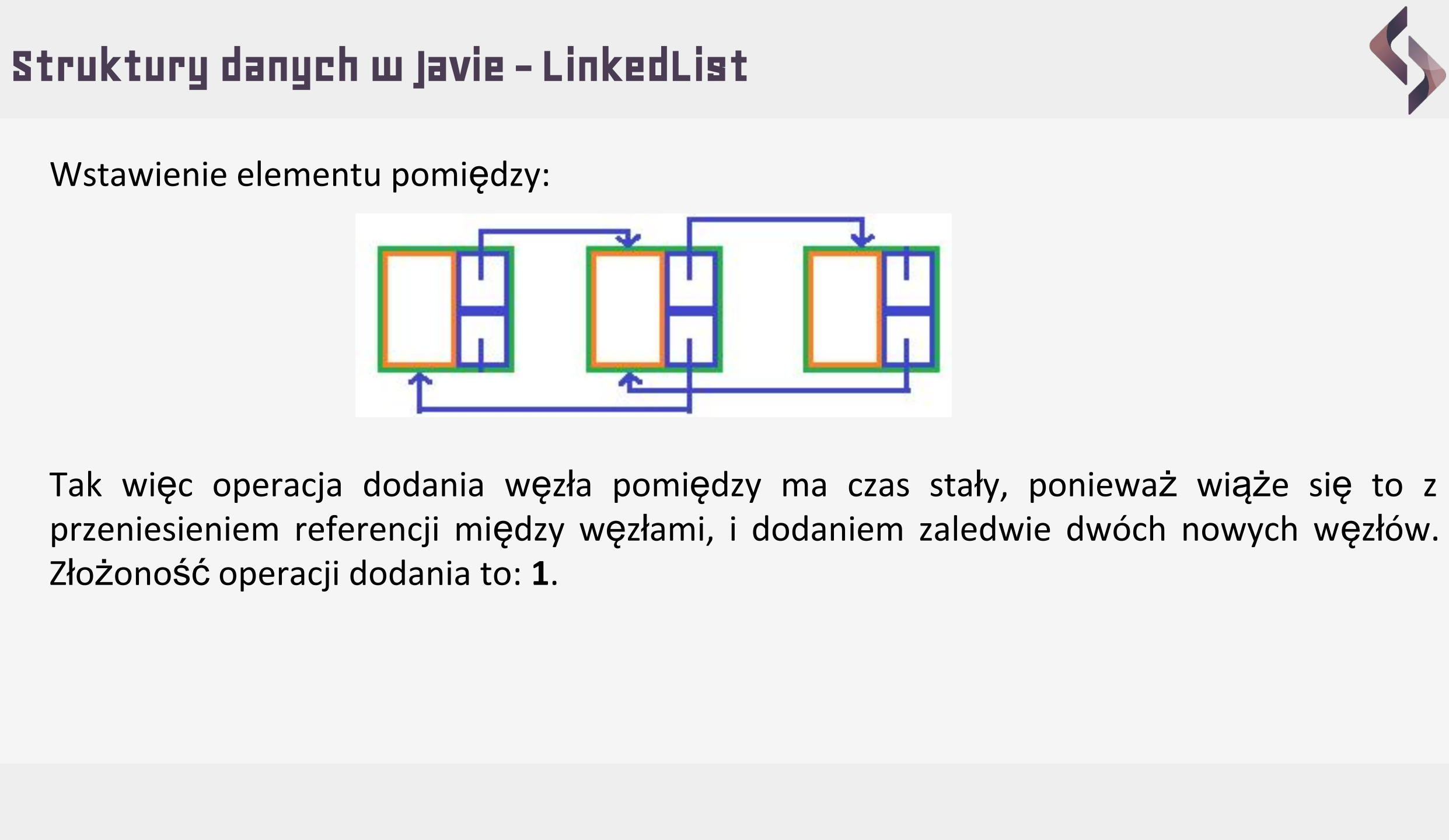


Struktury danych w Javie - LinkedList

Wstawienie elementu pomiędzy:



Struktury danych w Javie - LinkedList



Wstawienie elementu pomiędzy:

Tak więc operacja dodania węzła pomiędzy ma czas stały, ponieważ wiąże się to z przeniesieniem referencji między węzłami, i dodaniem zaledwie dwóch nowych węzłów. Złożoność operacji dodania to: **1**.

Struktury danych w Javie - LinkedList

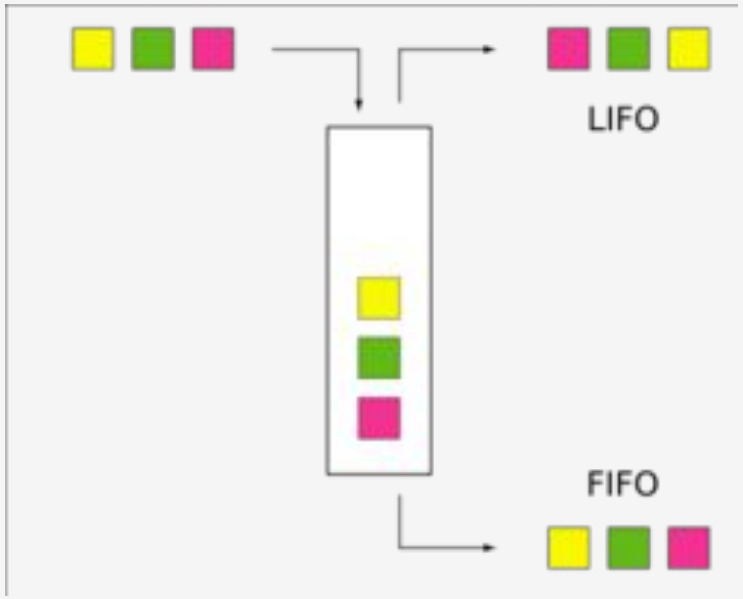
Co się dzieje z operacją usunięcia?
Podobnie jak w przypadku dodania czas złożoności uzależniony jest od pozycji obiektu. Jeśli mówimy o usunięciu elementu z środka, to pesymistycznie operacja ta może się wiązać z kosztem nawet ***N*** operacji.

Sytuacja zmienia się kiedy mówimy o operacji usunięcia i/lub dodania elementu na początku i/lub końcu listy. Zauważmy, że sama struktura listy posiada referencję na początek i koniec (czyli na pierwszy i ostatni element), co czyni ją idealną do dodawania i/lub usuwania elementów na obu końcach. Te właściwości listy bardzo często są wykorzystywane w strukturach które również wykorzystują te kolekcję.

LinkedList jest najlepszym rozwiązaniem implementującym bufory **LIFO** i **FIFO**, czyli bufor **First In First Out**, oraz **Last In Last Out**.

Struktury danych w Javie - LinkedList

LinkedList jest najlepszym rozwiązaniem implementującym bufory **LIFO** i **FIFO**, czyli bufor **First In First Out**, oraz **Last In Last Out**.



Na powyższym rysunku w środku znajduje się bufor (nasza lista). Operacja **LIFO**