



# Struktury danych

## ArrayList



# Wstęp

Struktury danych w Javie zostały zaprojektowane po to, aby ułatwić korzystanie z kolekcji danych. W Javie pojawia się bardzo dużo typów kolekcji, które różnią się od siebie zachowaniem, czasem wykonania poszczególnych operacji, a także możliwościami jakie oferują (zakresem operacji które można na nich wykonać).

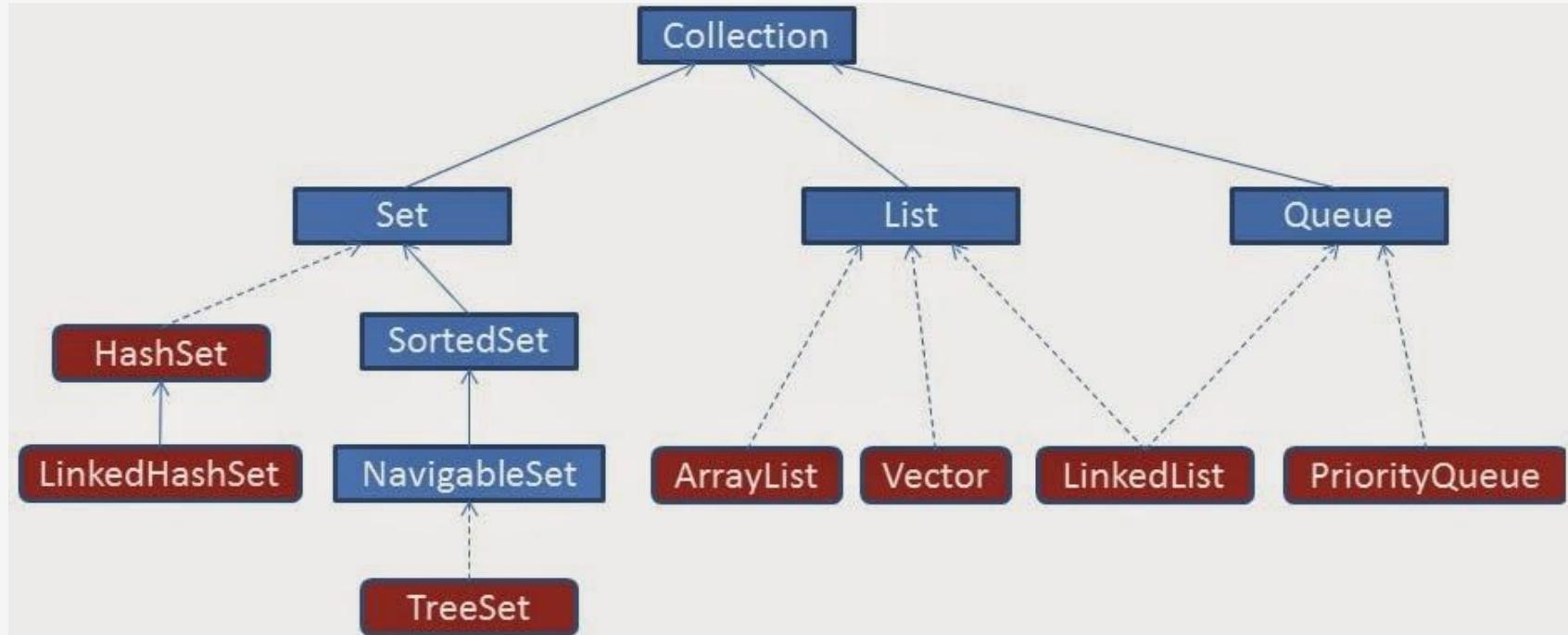
Znana jest nam podstawowa kolekcja, na której możemy operować, z której obecnie korzysta się dosyć rzadko. Mowa tutaj o strukturze tablicowej.

W prezentacji przedstawimy sobie kilka wad i zalet struktury tablicowej, oraz sukcesywnie przejdziemy do prezentacji kolejnych typów danych. Na samym początku jednak, wymienimy sobie wszystkie struktury.



# Struktury danych w Javie

Kiedy mówimy o kolekcjach w Javie, mamy na myśli dynamiczne struktury które są zdolne do przechowywania instancji naszych obiektów. Mówimy na nie kolekcje, ponieważ wszystkie implementują interfejs **Collection<T>**.



Na załączonym powyżej obrazku znajduje się opis postawowych **kolekcji** oraz **interfejsów** które implementują.



# Struktury danych w Javie - generyczność

Zaczniemy od omawiania kolekcji które implementują interfejs **List<T>**. Do tych kolekcji należą między innymi klasy **ArrayList<T>** oraz **LinkedList<T>**. Wszystkie kolekcje są klasami generycznymi. Oznacza to, że podczas deklaracji instancji tego typu powinniśmy w nawiasie diamentowym (**<T>**) podać **T**, czyli typ obiektu który tworzoną instancja będzie przechowywać. Po zadeklarowaniu typu w obiekcie zmienia się zachowanie tego obiektu, a mianowicie typ danych, który jest przyjmowany przez te instancje.

```
public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();
    list.add(
}  m  add(String e)          boolean
  m  add(int index, String element)      void
  m  addAll(Collection<? extends String>... boolean
  m  addAll(int index, Collection<? exte... boolean
Ctrl+Down and Ctrl+Up will move caret down and up in the editor >> π
```

```
public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<>();
    list.add(
}  m  add(Integer e)          boolean
  m  add(int index, Integer element)      void
  m  addAll(Collection<? extends Integer>... boolean
  m  addAll(int index, Collection<? exte... boolean
Did you know that Quick Definition View (Ctrl+Shift+I) works in completion lookups as well? >> π
```

Powyżej zaprezentowana została operacja dodania obiektu do listy. Jeśli zadeklarowanym typem jest **String**, to metoda przyjmuje obiekty typu **String**. Kiedy zadeklarujemy inny typ (w przykładzie po prawej stronie **Integer**) to klasa przyjmuje obiekty tylko tego typu.



# Struktury danych w Javie

Do omawiania list, zaczniemy od omówienia struktury tablicowej. Tablica:

```
Integer[] tablica = new Integer[30];  
int[] tablica_2 = new int[] {3, 5, 6};
```

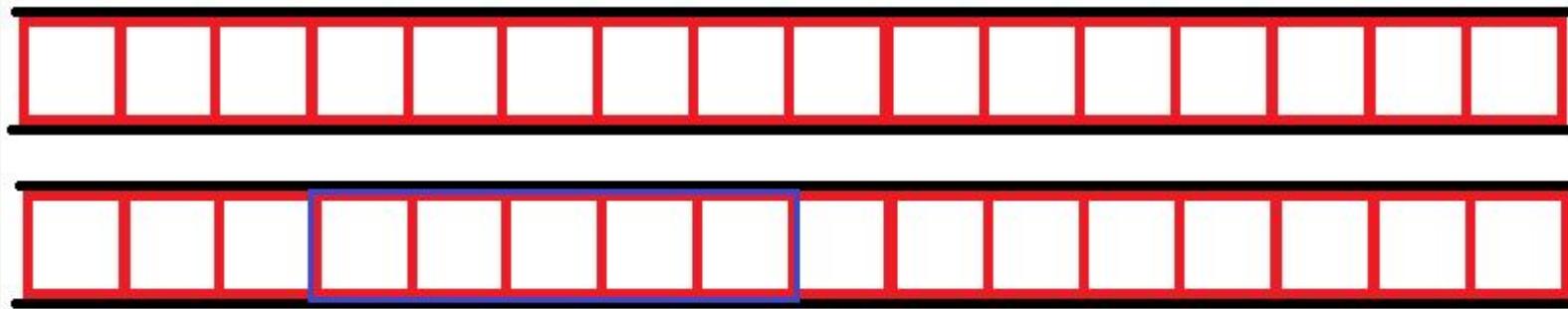
Posiada kilka wad o których warto wspomnieć. Tablica jest reprezentowana w pamięci RAM w postaci ciągłego bloku pamięci. Aby być w stanie przechować podaną ilość pamięci konieczna jest wiedza na temat finalnego rozmiaru bloku. Z tego właśnie względu, za każdym razem kiedy chcemy zadeklarować tablicę to możemy to zrobić tylko, jeśli podamy rozmiar tablicy (linia **1** kodu powyżej), lub jeśli wiemy jakie obiekty będzie przechowywać i podamy te obiekty (linia **2** kodu wyżej). Jest to odosyć uciążliwe, z uwagi na fakt iż czasami ciężko jest przewidzieć finalny rozmiar kolekcji z której będziemy korzystać.

Niestety to nie jedyna wada tej struktury.



# Struktury danych w Javie

Wyobraźmy sobie sytuacje, kiedy to chcemy wykorzystać tablicę w naszej aplikacji. W takiej sytuacji zasady *Quality Assurance* którymi powinniśmy się kierować (zapewnianie jakości kodu) mówią, że powinniśmy zabezpieczyć się przed ewentualnym wykroczeniem poza tablicę. Tak więc jeśli zadeklarowaliśmy sobie tablicę, powiedzmy w następujący sposób:



```
int[] tab = new int[5];
```

Po zadeklarowaniu w pamięci tablicy, jest nam rezerwowany blok pamięci tak, jak na powyższym rysunku. U góry widoczne są komórki pamięci RAM, na dole zaś zarezerwowany dla nas blok pamięci.

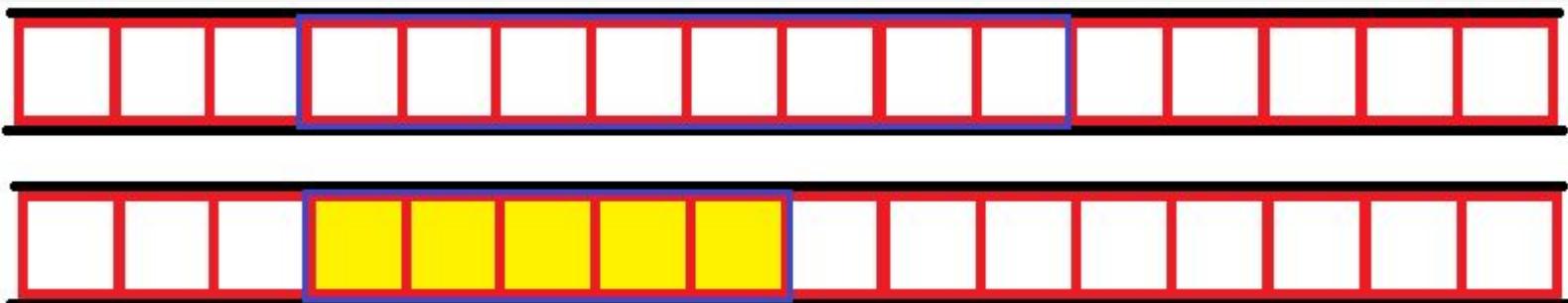


# Struktury danych w Javie

Kiedy zachodzi potrzeba rozszerzania naszej tablicy, musimy liczyć się z tym, że nie ma prostego sposobu na dodanie do niej miejsca. Wyobraźmy sobie naszą tablicę:



Na dole znajduje się pełna tablica. Jeśli chcę zmieścić w niej więcej, to moja jedyna opcja, to stworzyć sobie drugą tablicę:

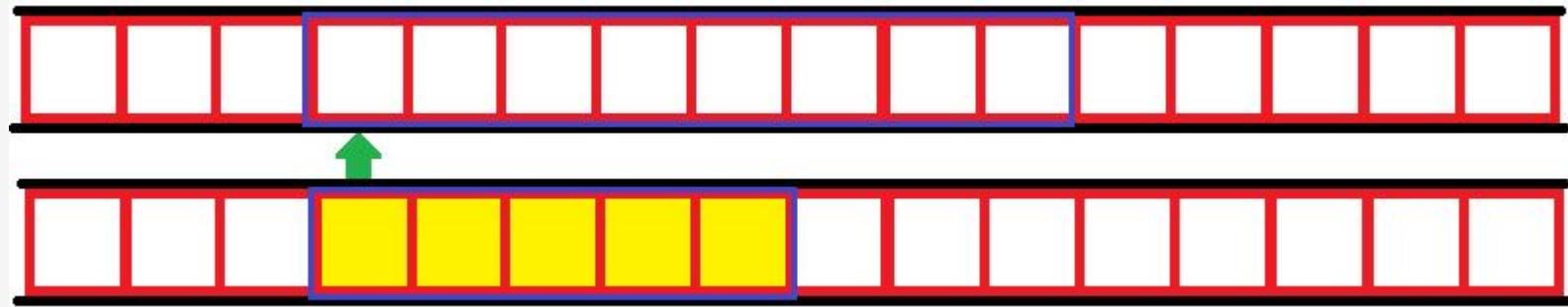


Następnie muszę przepisać wszystkie elementy ze starej tablicy do nowej.



# Struktury danych w Javie

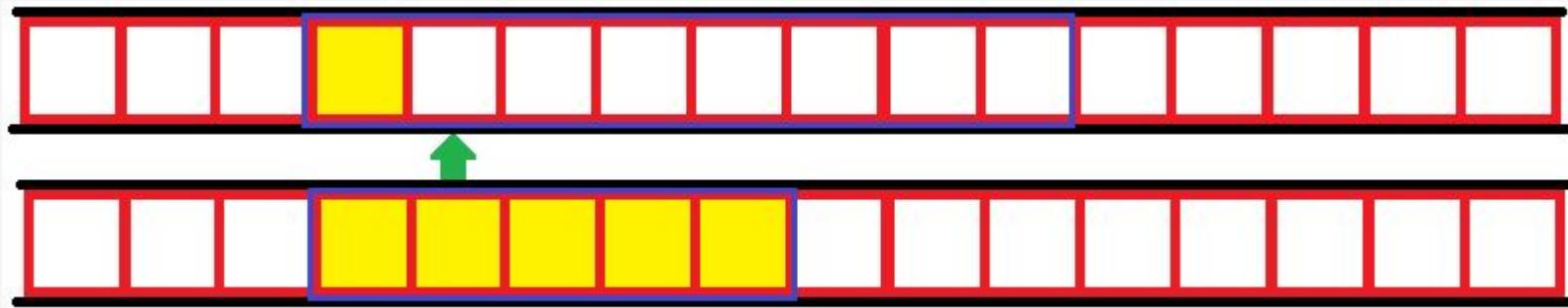
Tak więc przepisując elementy, muszę wykonać  $N$  operacji, gdzie  $N$  oznacza pojemność pierwotnej tablicy:





# Struktury danych w Javie

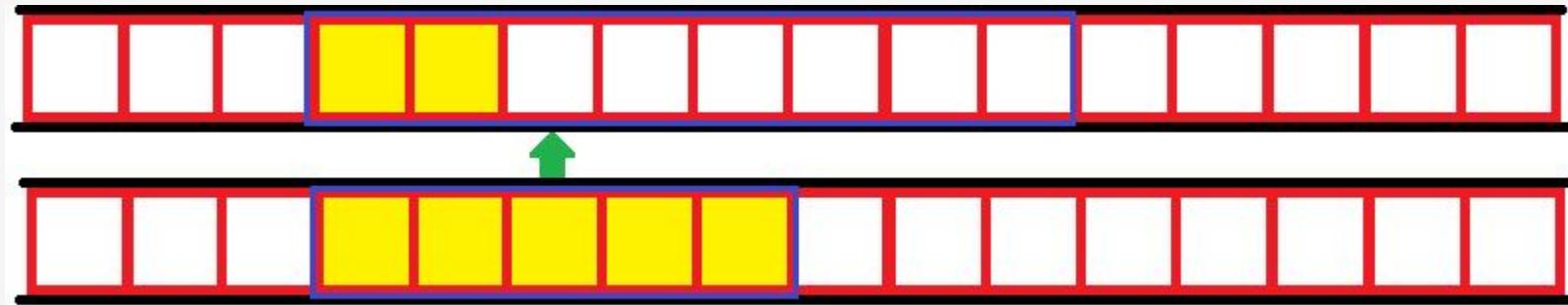
Tak więc przepisując elementy, muszę wykonać  $N$  operacji, gdzie  $N$  oznacza pojemność pierwotnej tablicy:





# Struktury danych w Javie

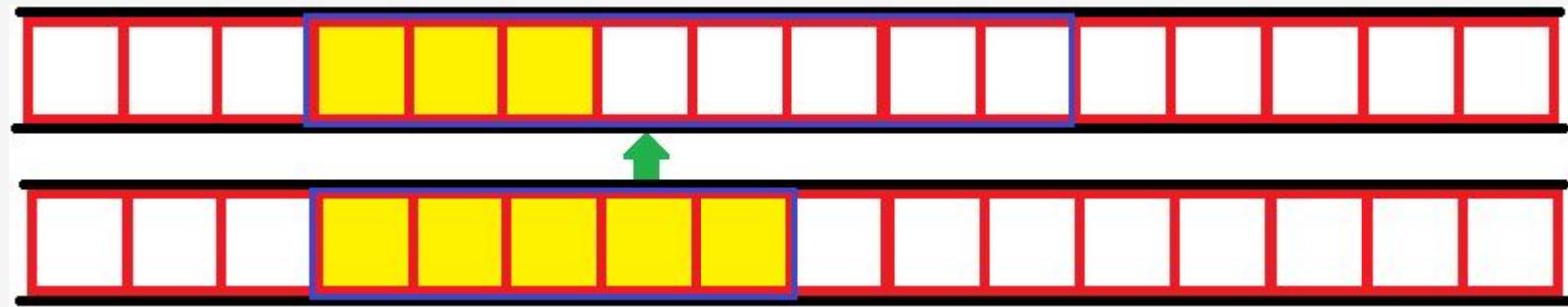
Tak więc przepisując elementy, muszę wykonać  $N$  operacji, gdzie  $N$  oznacza pojemność pierwotnej tablicy:





# Struktury danych w Javie

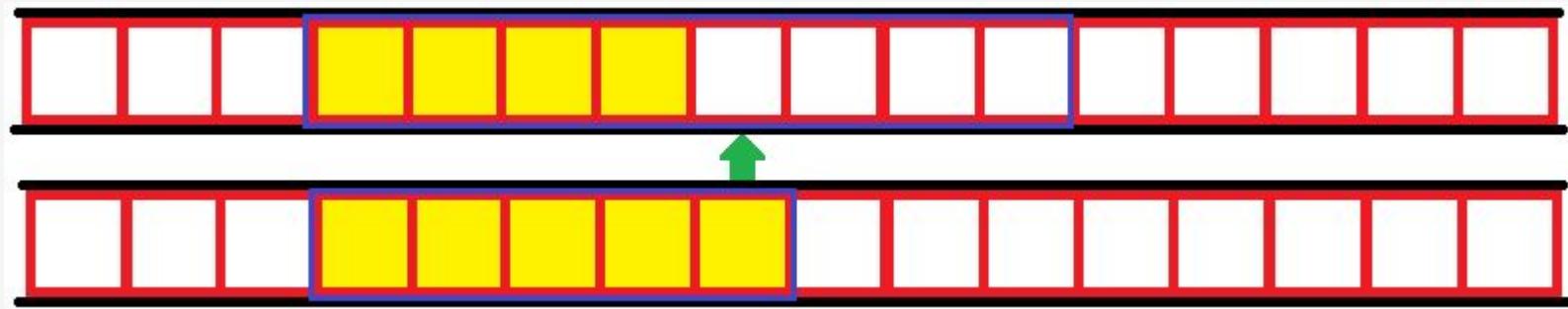
Tak więc przepisując elementy, muszę wykonać  $N$  operacji, gdzie  $N$  oznacza pojemność pierwotnej tablicy:





# Struktury danych w Javie

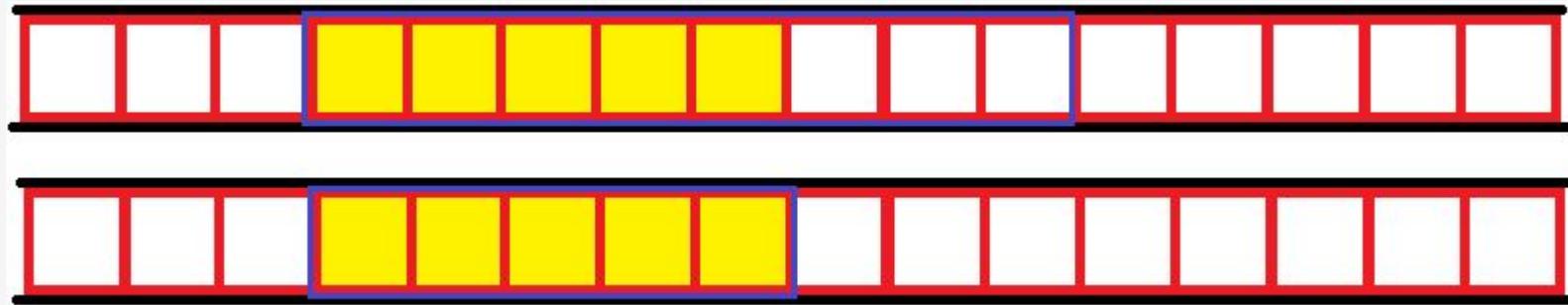
Tak więc przepisując elementy, muszę wykonać  $N$  operacji, gdzie  $N$  oznacza pojemność pierwotnej tablicy:





# Struktury danych w Javie

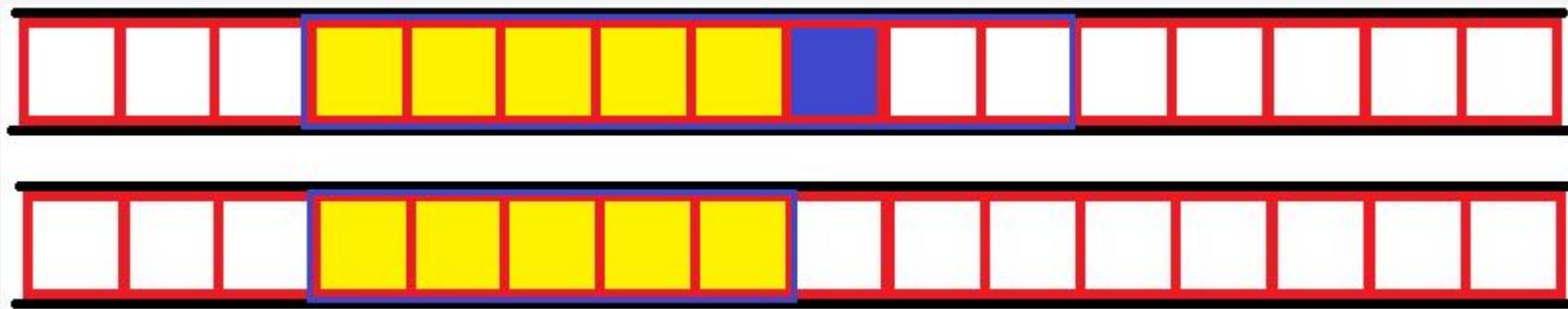
Tak więc przepisując elementy, muszę wykonać  $N$  operacji, gdzie  $N$  oznacza pojemność pierwotnej tablicy:





# Struktury danych w Javie

Po przepisaniu elementów do nowej tablicy możemy w wolne miejsce dodać nasz nowy element:

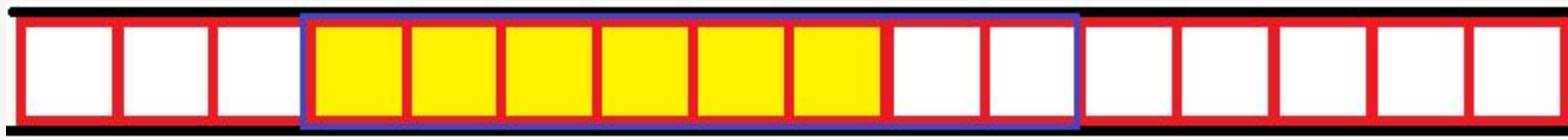


A następnie zastąpić starą tablicę nową tablicą, co spowoduje zwolnienie starej tablicy.



# Struktury danych w Javie

Po tej operacji mamy nową tablicę z wolnymi dwoma miejscami.



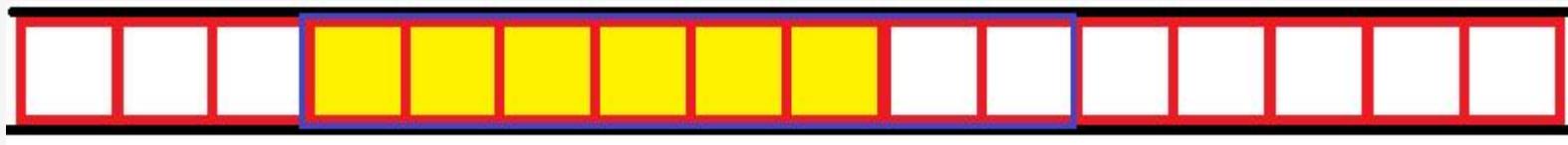
Oczywiście, gdybyśmy chcieli pracować na takiej tablicy, to żeby uniknąć częstego przepisywania elementów dokonywalibyśmy rozszerzenia o więcej niż 3 komórki.

Przejdźmy teraz do usunięcia jednego elementu z naszej nowej tablicy.



# Struktury danych w Javie

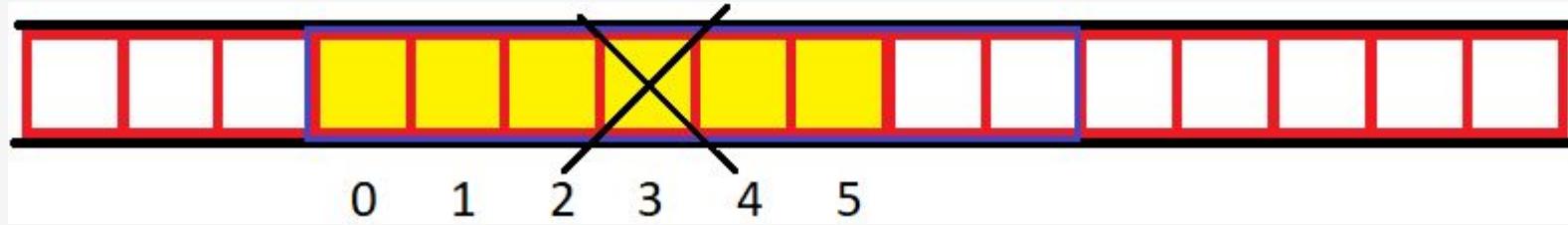
Zakładamy usunięcie elementu na pozycji 3:





# Struktury danych w Javie

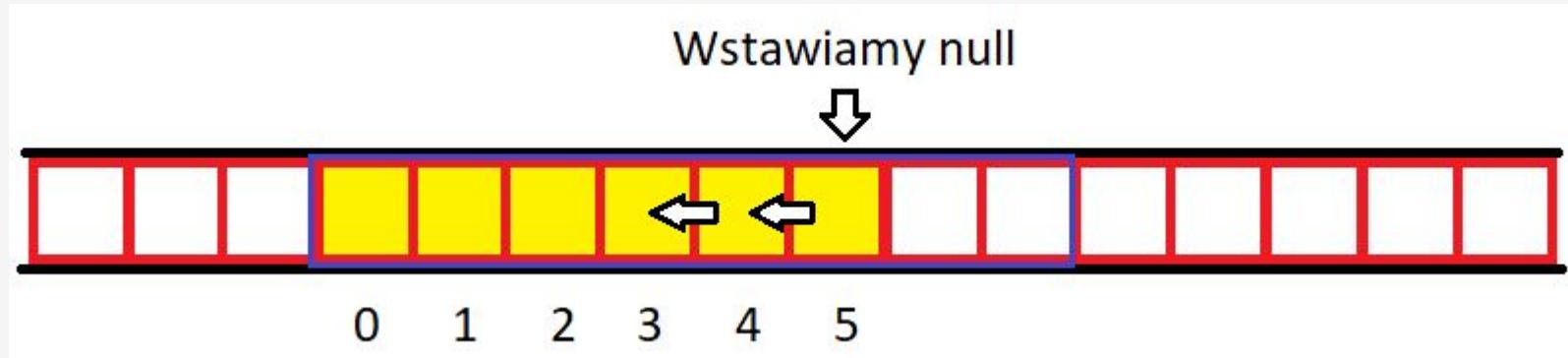
Zakładamy usunięcie elementu na pozycji 3:





# Struktury danych w Javie

Zakładamy usunięcie elementu na pozycji 3:



Aby dokonać tej operacji musimy najpierw przesunąć dwa obiekty. Jeśli usuwam obiekt na indeksie 3, to pierwszym krokiem będzie przesunięcie obiektów z pozycji 4 na pozycję 3, a następnie z pozycji 5 na pozycję 4.

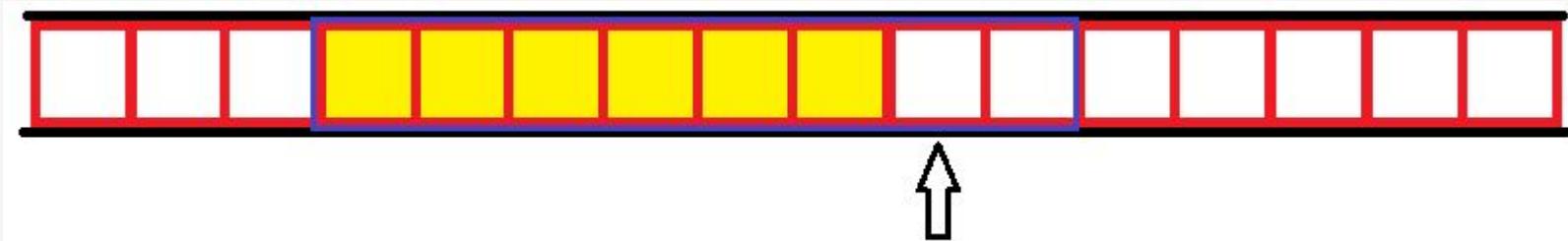
Po dokonaniu przesunięć musimy na ostatniej pozycji w tablicy umieścić null, aby oznaczyć miejsce jako wolne.

Złożoność tej operacji to pesymistycznie  **$N$**  operacji, czyli w najgorszym przypadku (jeśli usuwamy 1 element) musimy przemieścić  **$N-1$**  obiektów, a następnie dopisać null na końcu.



# Struktury danych w Javie

Rozpatrując dodawanie obiektu do tablicy okazuje się, że operacja również nie jest trywialna. Jeśli chcemy dodać element na koniec tablicy, to musimy wykonać iterację przez tablicę, aby odnaleźć pierwszą komórkę w której znajduje się null.



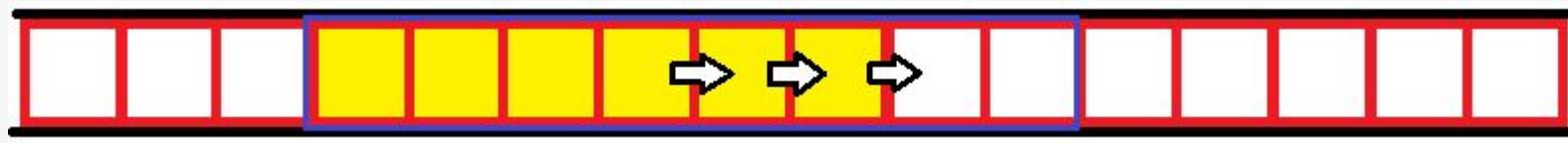
Pesymistycznie musimy dokonać **N** operacji, czyli w zależności od zajętości tablicy istnieje szansa że pójdziemy aż na sam koniec tablicy zanim odnajdziemy **null**.

Tego typu informacje nie znajdują się w samej implementacji tablicy, dlatego musimy sami zadbać o to, aby obiekty umieszczać w odpowiednim miejscu i zadbać o ich porządek.

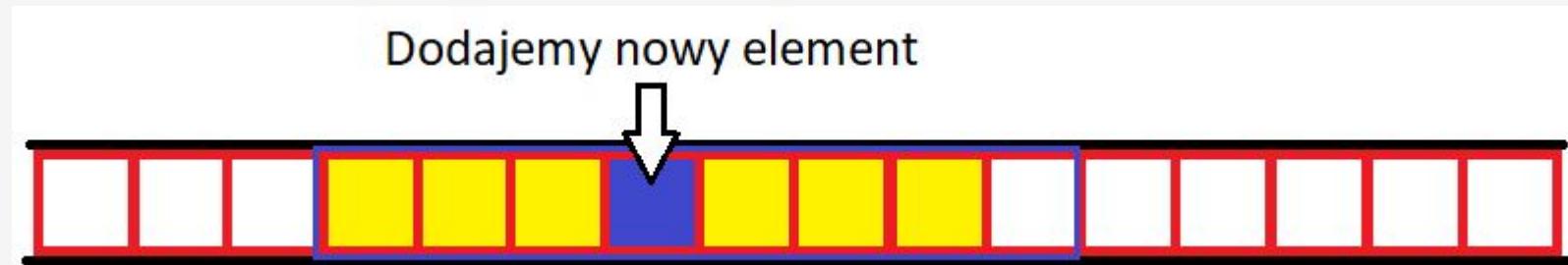


# Struktury danych w Javie

Ostatnią rzeczą którą należy zauważyc jest również wstawianie elementu gdzieś w środku. Aby tego dokonać, musimy po kolei przestawiać elementy od końca tablicy do miejsca gdzie chcemy wstawić obiekt, aby zwolnić sobie miejsce. Na poniższej grafice chcę wstawić obiekt na pozycji 3 w naszej tablicy.



Po przesunięciu zwalniamy miejsce, po czym możemy w wolne pole wstawić nasz nowy element:



Złożoność tej operacji to znowu – pesymistycznie  $N$  operacji – w przypadku gdy chcemy umieścić nowy element na pozycji  $0$  musimy przemieścić  $N$  elementów w prawo.



# Struktury danych w Javie

Wszystkie te operacje, które właśnie omówiliśmy to operacje:

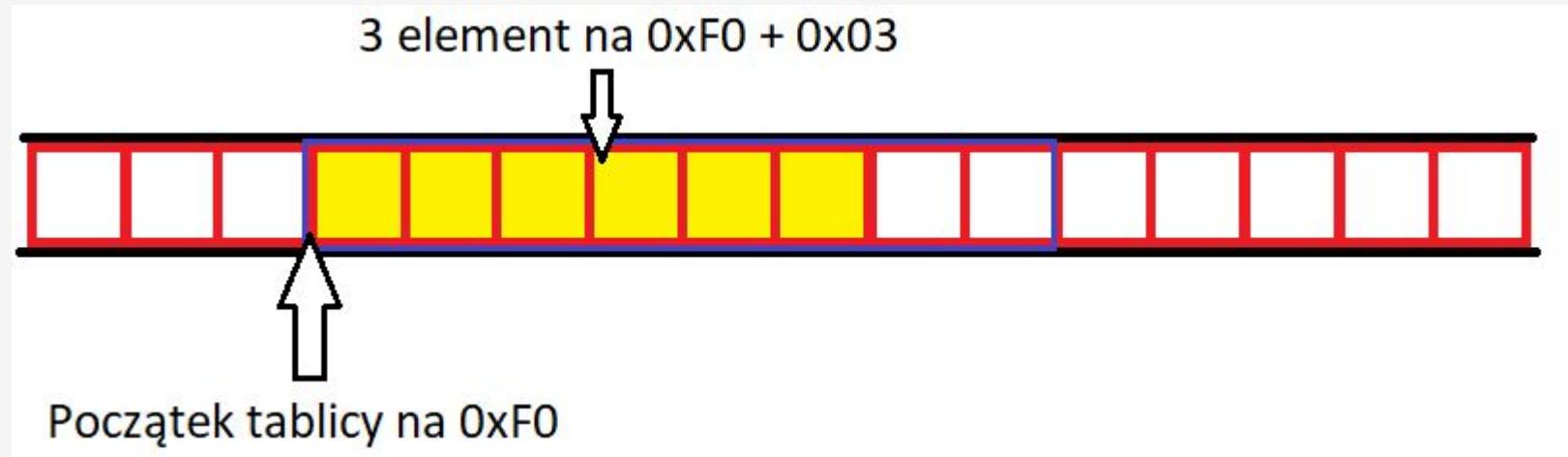
- dodawania
- usuwania
- dodawania na pozycji x
- usuwania z pozycji x
- określania rozmiaru (znajdowanie wolnego miejsca – null)

Do zbioru tych operacji należy również dodać ostatnią operację, którą jest wybieranie n-tego elementu. Ta operacja jest szczególnie ważna w przypadku tablicy, ponieważ zachowanie tego typu struktury jest proste i szybkie.



# Struktury danych w Javie

Kiedy chcemy uzyskać dostęp do  $i$ -tego elementu, oraz kiedy znamy rozmiar obiektu, wystarczy, że policzymy tzw. offset, czyli przesunięcie względem początku tablicy.



Dokonanie prostego obliczenia dodania do adresu początkowego tablicy przesunięcia (offsetu) jest szybkie, a czas wykonania niezależnie od rozmiaru tablicy będzie zawsze stały. Mówimy wtedy że czas wykonania to **1**.



# Struktury danych w Javie

Wszystkie czynności które sobie omówiliśmy to rzeczy które zostały zaimplementowane na strukturze ArrayList. Korzystając z kolekcji ArrayList mamy do dyspozycji operacje:

- |                                   |  |
|-----------------------------------|--|
| -add(Object what);                | -dodanie elementu  |
| -add(Integer index, Object what); | -dodanie elementu na i-tej pozycji                       |
| -remove(Object what);             | -usunięcie pierwszego wystąpienia obiektu <b>what</b>    |
| -remove(Integer index);           | - usunięcie obiektu na indeksie <b>index</b>             |
| -set(Integer index, Object what); | - ustawienie obiektu <b>what</b> na pozycji <b>index</b> |
| -get(Integer index);              | - pobranie obiektu o indeksie <b>index</b>               |



# Struktury danych w Javie

Metoda ***add(Object o)*** – dodaje element na końcu listy. Złożoność operacji jest uzależniona od dostępności miejsca w tablicy. Jeśli brakuje miejsca, konieczne jest rozszerzenie tablicy, dlatego pesymistycznie złożoność operacji wynosi:

***O(n)*** – operacji nie można wykonać szybciej niż dokonując ***n*** operacji, gdzie ***n*** to ilość elementów w tablicy. Złożoność ta jest tak wysoka głównie z powodu ograniczeń rozmiarowych na tablicy. Jeśli brakuje miejsca w tablicy, powstaje konieczność przepisywania wszystkich elementów. (prezentowane na wcześniejszych slajdach)

Optymistycznie złożoność jest znacznie szybsza i dokonując prostej optymalizacji w postaci przechowywania indeksu ostatniego elementu w liście czas dodania spada i może się równać ze stałym czasem dodania elementu.



# Struktury danych w Javie

Metoda ***add(int i, Object o)*** – dodaje element na i-tej pozycji w liście. Złożoność operacji jest również uzależniona od dostępności miejsca w tablicy. Jeśli brakuje miejsca, konieczne jest rozszerzenie tablicy, dlatego pesymistycznie złożoność operacji wynosi:

***O(n)*** – operacji nie można wykonać szybciej niż dokonując  $n$  operacji, gdzie  $n$  to ilość elementów w tablicy. Taka sama złożoność wynika głównie z faktu, iż pesymistycznie będziemy zmuszeni do przepisywania tablicy, ale również jeśli wstawiamy obiekt na pozycji 0, to powstanie konieczność przesunięcia wszystkich obiektów. W takiej sytuacji mówimy o złożoności ***O(2\*n)*** jednak dwójka z wielomianu może zostać pominięta, ponieważ symbol dużego O mówi, że operacja **nie wykona się szybciej niż**, więc oba oznaczenia są poprawne.

Nie jest możliwe większe zoptymalizowanie tej operacji.



# Struktury danych w Javie

Metoda ***remove(Object o)*** – usuwa pierwsze wystąpienie obiektu *o* na liście. Złożoność operacji:

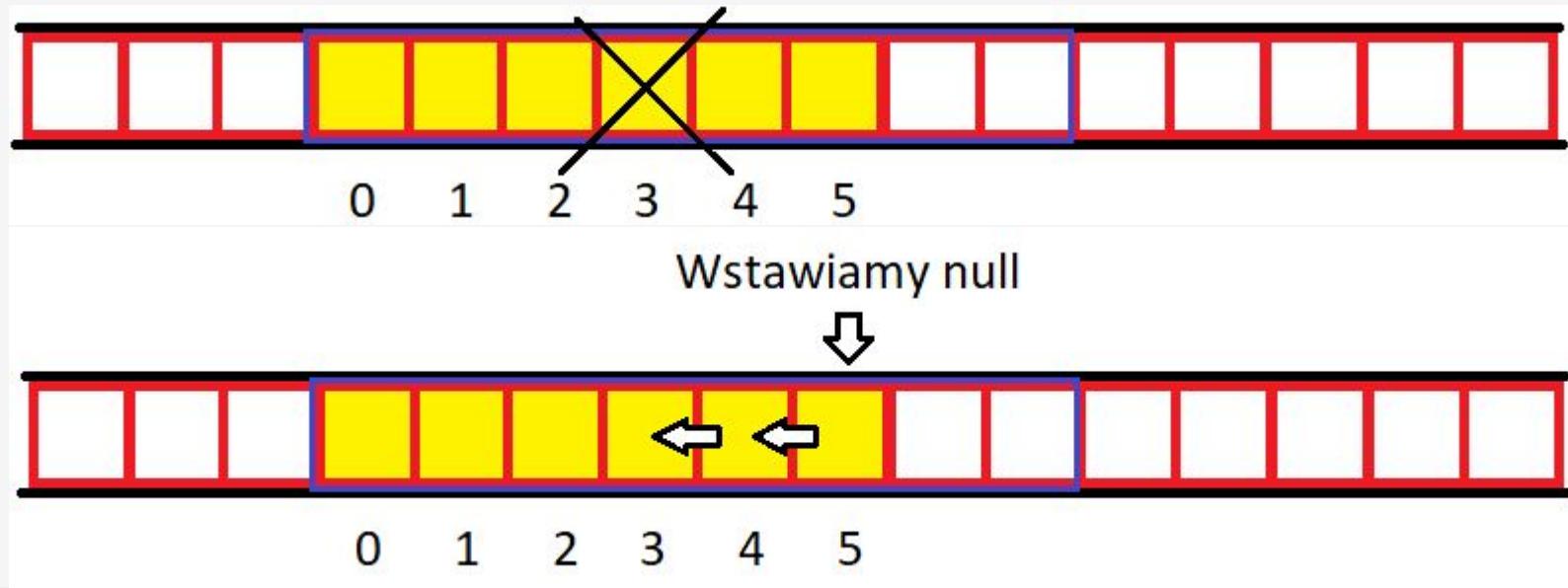
***O(n)*** – złożoność jest bardzo wysoka, głównie z uwagi na fakt, że konieczne jest przejście po tablicy elementów oraz sprawdzanie każdego elementu z kolejna w celu sprawdzenia czy jest równy obiekowi *o*.



# Struktury danych w Javie

Metoda ***remove(int i)*** – usuwa obiekt na indeksie *i* z tablicy. Złożoność operacji:

**O(n)** – mogłoby się wydawać że tutaj złożoność będzie niższa, jednak tak jak pokazane na wcześniejszych obrazkach: możliwe, że pesymistycznie będziemy zmuszeni do przesuwania wszystkich obiektów od końca do początku. Jeśli usuniemy obiekt o indeksie 0, to konieczne będzie przepisanie **N-1** elementów.





# Struktury danych w Javie

Metoda ***get(int i)*** – pobiera obiekt na indeksie *i* z tablicy. Złożoność operacji:

**O(1)** – złożoność tej operacji jest stała. Niezależnie jak daleki jest indeks obiektu do którego musimy uzyskać dostęp, czas wykonania obliczeń będzie zawsze stały – a raczej nie będzie uzależniony od rozmiaru tablicy.



# Struktury danych w Javie

Wszystkie operacje dostępne na liście ***ArrayList*** wynikają z implementacji interfejsu ***List***. W następnej prezentacji przejdziemy do wykonywania operacji na liście ***LinkedList***, oraz dokonamy porównania czasów działań poszczególnych kolekcji.



# Struktury danych w Javie

Dziękuję za uczestnictwo w zajęciach, jeśli posiadasz jakieś sugestie na temat informacji które powinny zostać zamieszczone w tej prezentacji, lub uważasz, że w prezentacji poruszana jest tematyka która wymaga głębszego wyjaśnienia/analizy, proszę o kontakt:

[pawel.reclaw@gmail.com](mailto:pawel.reclaw@gmail.com)