Containers

Wygenerowano przez Doxygen 1.9.1

Rozdział 1

Indeks hierarchiczny

1.1 Hierarchia klas

Ta lista dziedziczenia posortowana jest z grubsza, choć nie całkowicie, alfabetycznie:

Rozdział 2

Indeks klas

2.1 Lista klas

Tutaj znajdują się klasy, struktury, unie i interfejsy wraz z ich krótkimi opisami:

Rozdział 3

Indeks plików

3.1 Lista plików

Tutaj znajduje się lista wszystkich plików z ich krótkimi opisami:

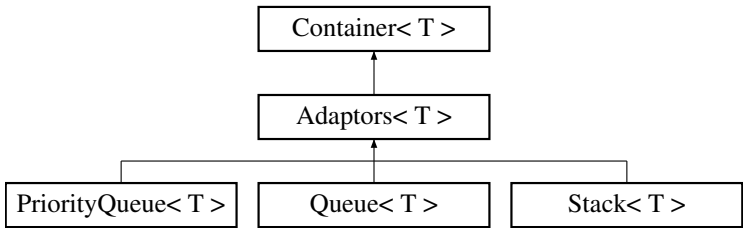
Rozdział 4

Dokumentacja klas

4.1 Dokumentacja szablonu klasy Adaptors< T >

`#include <Adaptors.h>`

Diagram dziedziczenia dla Adaptors< T >



Metody publiczne

- virtual void **Push** (const T &value)=0
- virtual void **Pop** ()=0
- virtual bool **allowedType** ()=0
- virtual T **returnEmpty** ()=0
- virtual bool **Empty** () const noexcept=0
- virtual int **Size** () const noexcept=0
- virtual void **Clear** ()=0
- virtual int **MaxSize** ()=0
- virtual void **Show** ()=0

4.1.1 Opis szczegółowy

template<class T>
class Adaptors< T >

Deklaracja klasy **Adaptors** (str. 7).

- podstawa wszystkich klas opisujących struktury danych będące typu adaptor.
- dziedzicząca po **Container** (str. 15).

4.1.2 Dokumentacja funkcji składowych

4.1.2.1 allowedType()

```
template<class T >
virtual bool  Adaptors< T >::allowedType ( )  [pure virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Container< T >** (str. 16).

Implementowany w **Stack< T >** (str. 167), **Queue< T >** (str. 137) i **PriorityQueue< T >** (str. 128).

4.1.2.2 Clear()

```
template<class T >
virtual void  Adaptors< T >::Clear ( )  [pure virtual]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0.

Implementuje **Container< T >** (str. 16).

Implementowany w **Stack< T >** (str. 167), **PriorityQueue< T >** (str. 129) i **Queue< T >** (str. 138).

4.1.2.3 Empty()

```
template<class T >
virtual bool  Adaptors< T >::Empty ( ) const  [pure virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Container< T >** (str. 16).

Implementowany w **Stack< T >** (str. 167), **Queue< T >** (str. 138) i **PriorityQueue< T >** (str. 129).

4.1.2.4 MaxSize()

```
template<class T >
virtual int  Adaptors< T >::MaxSize ( )  [pure virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener.

Zwraca

Maksymalna wielkość kontenera.

Implementuje **Container< T >** (str. 16).

Implementowany w **Stack< T >** (str. 167), **Queue< T >** (str. 139) i **PriorityQueue< T >** (str. 130).

4.1.2.5 Pop()

```
template<class T >
virtual void  Adaptors< T >::Pop ( )  [pure virtual]
```

Usunięcie elementu z TOPu i zmniejszenie jego wielkości.

Implementowany w **Stack< T >** (str. 169), **Queue< T >** (str. 140) i **PriorityQueue< T >** (str. 131).

4.1.2.6 Push()

```
template<class T >
virtual void  Adaptors< T >::Push (
            const T & value )  [pure virtual]
```

Dodanie nowego elementu na TOP kontenera i zwiększenie jego wielkości.

Parametry

value	Nowy element.

Implementowany w **Stack< T >** (str. 169), **Queue< T >** (str. 141) i **PriorityQueue< T >** (str. 132).

4.1.2.7 returnEmpty()

```
template<class T >
virtual T  Adaptors< T >::returnEmpty ( )  [pure virtual]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

Implementowany w **Stack< T >** (str. 169), **Queue< T >** (str. 141) i **PriorityQueue< T >** (str. 132).

4.1.2.8 Show()

```
template<class T >
virtual void  Adaptors< T >::Show ( )  [pure virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Container< T >** (str. 17).

Implementowany w **Stack< T >** (str. 170), **Queue< T >** (str. 141) i **PriorityQueue< T >** (str. 132).

4.1.2.9 Size()

```
template<class T >
virtual int  Adaptors< T >::Size ( ) const  [pure virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Container< T >** (str. 17).

Implementowany w **Stack< T >** (str. 170), **Queue< T >** (str. 142) i **PriorityQueue< T >** (str. 133).

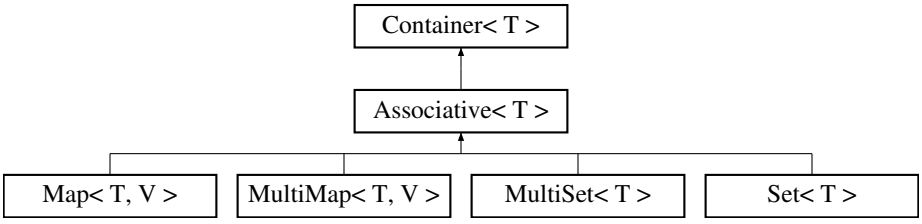
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Adaptors.h**

4.2 Dokumentacja szablonu klasy Associative< T >

```
#include <Associative.h>
```

Diagram dziedziczenia dla Associative< T >



Metody publiczne

- virtual void **PopBack** ()=0
- virtual void **PopFront** ()=0
- virtual bool **allowedType** ()=0
- virtual bool **Empty** () const noexcept=0
- virtual int **Size** () const noexcept=0
- virtual void **Clear** () noexcept=0
- virtual void **Erase** (int position)=0
- virtual void **Erase** (int first, int last)=0
- virtual int **MaxSize** ()=0
- virtual void **Reverse** () noexcept=0
- virtual void **Sort** ()=0
- virtual void **Show** ()=0

4.2.1 Opis szczegółowy

template<class T>
class Associative< T >

Deklaracja klasy **Associative** (str. 10).

- podstawa wszystkich klas opisujących struktury danych będące typu asocjacyjnego.
- dziedzicząca po **Container** (str. 15).

4.2.2 Dokumentacja funkcji składowych

4.2.2.1 allowedType()

```
template<class T >
virtual bool  Associative< T >::allowedType ( )  [pure virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Container< T >** (str. 16).

Implementowany w **Set< T >** (str. 155), **MultiSet< T >** (str. 115), **MultiMap< T, V >** (str. 103) i **Map< T, V >** (str. 91).

4.2.2.2 Clear()

```
template<class T >
virtual void  Associative< T >::Clear ( )  [pure virtual], [noexcept]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Container< T >** (str. 16).

Implementowany w **Set< T >** (str. 156), **MultiSet< T >** (str. 116), **MultiMap< T, V >** (str. 104) i **Map< T, V >** (str. 92).

4.2.2.3 Empty()

```
template<class T >
virtual bool  Associative< T >::Empty ( ) const  [pure virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Container< T >** (str. 16).

Implementowany w **Set< T >** (str. 156), **MultiSet< T >** (str. 117), **MultiMap< T, V >** (str. 104) i **Map< T, V >** (str. 92).

4.2.2.4 Erase() [1/2]

```
template<class T >
virtual void  Associative< T >::Erase (
            int first,
            int last )  [pure virtual]
```

Usunięcie wybranych elementów z przedziału.

Parametry

first	Pierwszy indeks przedziału.
last	Ostatni indeks przedziału.

Implementowany w **Set< T >** (str. 157), **MultiSet< T >** (str. 117), **MultiMap< T, V >** (str. 105) i **Map< T, V >** (str. 93).

4.2.2.5 Erase() [2/2]

```
template<class T >
virtual void  Associative< T >::Erase (
            int position )  [pure virtual]
```

Usunięcie wybranego elementu po pozycji.

Parametry

position	Pozycja elementu do usunięcia.

Implementowany w **Set**< **T** > (str. 157), **MultiSet**< **T** > (str. 118), **MultiMap**< **T, V** > (str. 105) i **Map**< **T, V** > (str. 93).

4.2.2.6 MaxSize()

```
template<class T >
virtual int  Associative< T >::MaxSize ( )  [pure virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Container**< **T** > (str. 16).

Implementowany w **Set**< **T** > (str. 159), **MultiSet**< **T** > (str. 120), **MultiMap**< **T, V** > (str. 107) i **Map**< **T, V** > (str. 95).

4.2.2.7 PopBack()

```
template<class T >
virtual void  Associative< T >::PopBack ( )  [pure virtual]
```

Usunięcie ostatniego elementu i zmniejszenie wielkości o jeden.

Implementowany w **Set**< **T** > (str. 160), **MultiSet**< **T** > (str. 122), **MultiMap**< **T, V** > (str. 108) i **Map**< **T, V** > (str. 96).

4.2.2.8 PopFront()

```
template<class T >
virtual void  Associative< T >::PopFront ( )  [pure virtual]
```

Usunięcie pierwszego elementu i zmniejszenie wielkości o jeden.

Implementowany w **Set**< **T** > (str. 161), **MultiSet**< **T** > (str. 122), **MultiMap**< **T, V** > (str. 108) i **Map**< **T, V** > (str. 96).

4.2.2.9 Reverse()

```
template<class T >
virtual void  Associative< T >::Reverse ( )  [pure virtual], [noexcept]
```

Odwrócenie elementów kontenera kolejnością.

Implementowany w **Set**< **T** > (str. 162), **MultiSet**< **T** > (str. 123), **MultiMap**< **T, V** > (str. 110) i **Map**< **T, V** > (str. 98).

4.2.2.10 Show()

```
template<class T >
virtual void  Associative< T >::Show ( )  [pure virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Container**< **T** > (str. 17).

Implementowany w **Set**< **T** > (str. 162), **MultiSet**< **T** > (str. 123), **MultiMap**< **T, V** > (str. 110) i **Map**< **T, V** > (str. 98).

4.2.2.11 Size()

```
template<class T >
virtual int  Associative< T >::Size ( ) const  [pure virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Container**< **T** > (str. 17).

Implementowany w **Set**< **T** > (str. 163), **MultiSet**< **T** > (str. 124), **MultiMap**< **T, V** > (str. 110) i **Map**< **T, V** > (str. 98).

4.2.2.12 Sort()

```
template<class T >
virtual void  Associative< T >::Sort ( )  [pure virtual]
```

Sortowanie elementów w kontenerze

Implementowany w **Set< T >** (str. 163), **MultiSet< T >** (str. 124), **MultiMap< T, V >** (str. 111) i **Map< T, V >** (str. 99).

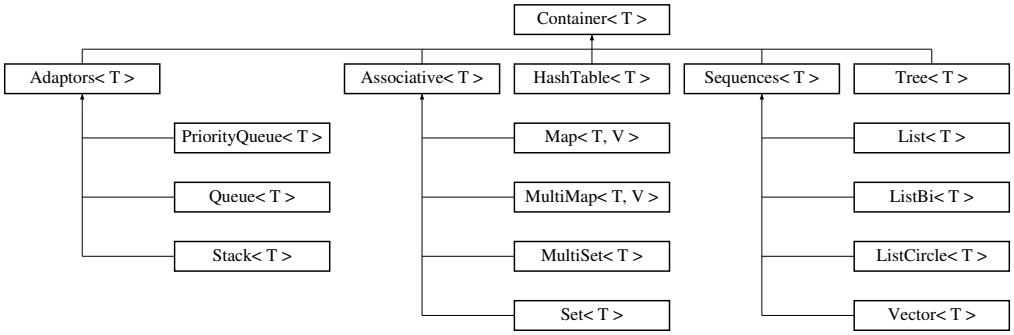
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Associative.h**

4.3 Dokumentacja szablonu klasy Container< T >

```
#include <Containers.h>
```

Diagram dziedziczenia dla Container< T >



Metody publiczne

- virtual bool **allowedType** ()=0
- virtual bool **Empty** () const noexcept=0
- virtual int **Size** () const noexcept=0
- virtual void **Clear** ()=0
- virtual int **MaxSize** ()=0
- virtual void **Show** ()=0

4.3.1 Opis szczegółowy

template<class T>
class Container< T >

Deklaracja klasy **Container** (str. 15).

- Podstawą wszystkich innych klas.

4.3.2 Dokumentacja funkcji składowych

4.3.2.1 allowedType()

```
template<class T >
virtual bool  Container< T >::allowedType ( )  [pure virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementowany w **Sequences**< **T** > (str. 144), **Associative**< **T** > (str. 11), **Adaptors**< **T** > (str. 8), **Vector**< **T** > (str. 194), **Tree**< **T** > (str. 174), **Stack**< **T** > (str. 167), **Set**< **T** > (str. 155), **Queue**< **T** > (str. 137), **PriorityQueue**< **T** > (str. 128), **MultiSet**< **T** > (str. 115), **MultiMap**< **T, V** > (str. 103), **Map**< **T, V** > (str. 91), **ListCircle**< **T** > (str. 73), **ListBi**< **T** > (str. 53), **List**< **T** > (str. 35) i **HashTable**< **T** > (str. 21).

4.3.2.2 Clear()

```
template<class T >
virtual void  Container< T >::Clear ( )  [pure virtual]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0.

Implementowany w **Sequences**< **T** > (str. 145), **Adaptors**< **T** > (str. 8), **Associative**< **T** > (str. 11), **Stack**< **T** > (str. 167), **Set**< **T** > (str. 156), **PriorityQueue**< **T** > (str. 129), **MultiSet**< **T** > (str. 116), **MultiMap**< **T, V** > (str. 104), **Map**< **T, V** > (str. 92), **ListCircle**< **T** > (str. 75), **ListBi**< **T** > (str. 55), **Vector**< **T** > (str. 196), **Tree**< **T** > (str. 174), **Queue**< **T** > (str. 138), **List**< **T** > (str. 37) i **HashTable**< **T** > (str. 21).

4.3.2.3 Empty()

```
template<class T >
virtual bool  Container< T >::Empty ( ) const  [pure virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementowany w **Sequences**< **T** > (str. 145), **Associative**< **T** > (str. 12), **Adaptors**< **T** > (str. 8), **Vector**< **T** > (str. 197), **Tree**< **T** > (str. 176), **Stack**< **T** > (str. 167), **Set**< **T** > (str. 156), **Queue**< **T** > (str. 138), **PriorityQueue**< **T** > (str. 129), **MultiSet**< **T** > (str. 117), **MultiMap**< **T, V** > (str. 104), **Map**< **T, V** > (str. 92), **ListCircle**< **T** > (str. 75), **ListBi**< **T** > (str. 56), **List**< **T** > (str. 38) i **HashTable**< **T** > (str. 22).

4.3.2.4 MaxSize()

```
template<class T >
virtual int  Container< T >::MaxSize ( )  [pure virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener.

Zwraca

Maksymalna wielkość kontenera.

Implementowany w **Sequences**< **T** > (str. 148), **Associative**< **T** > (str. 13), **Adaptors**< **T** > (str. 8), **Vector**< **T** > (str. 200), **Tree**< **T** > (str. 182), **Stack**< **T** > (str. 167), **Set**< **T** > (str. 159), **Queue**< **T** > (str. 139), **PriorityQueue**< **T** > (str. 130), **MultiSet**< **T** > (str. 120), **MultiMap**< **T, V** > (str. 107), **Map**< **T, V** > (str. 95), **ListCircle**< **T** > (str. 80), **ListBi**< **T** > (str. 60), **List**< **T** > (str. 42) i **HashTable**< **T** > (str. 26).

4.3.2.5 Show()

```
template<class T >
virtual void  Container< T >::Show ( )  [pure virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementowany w **Sequences**< **T** > (str. 151), **Associative**< **T** > (str. 14), **Adaptors**< **T** > (str. 10), **Vector**< **T** > (str. 205), **Tree**< **T** > (str. 187), **Stack**< **T** > (str. 170), **Set**< **T** > (str. 162), **Queue**< **T** > (str. 141), **PriorityQueue**< **T** > (str. 132), **MultiSet**< **T** > (str. 123), **MultiMap**< **T, V** > (str. 110), **Map**< **T, V** > (str. 98), **ListCircle**< **T** > (str. 84), **ListBi**< **T** > (str. 65), **List**< **T** > (str. 47) i **HashTable**< **T** > (str. 27).

4.3.2.6 Size()

```
template<class T >
virtual int  Container< T >::Size ( ) const  [pure virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementowany w **Sequences**< **T** > (str. 151), **Associative**< **T** > (str. 14), **Adaptors**< **T** > (str. 10), **Vector**< **T** > (str. 205), **Tree**< **T** > (str. 187), **Stack**< **T** > (str. 170), **Set**< **T** > (str. 163), **Queue**< **T** > (str. 142), **PriorityQueue**< **T** > (str. 133), **MultiSet**< **T** > (str. 124), **MultiMap**< **T, V** > (str. 110), **Map**< **T, V** > (str. 98), **ListCircle**< **T** > (str. 85), **ListBi**< **T** > (str. 65), **List**< **T** > (str. 47) i **HashTable**< **T** > (str. 28).

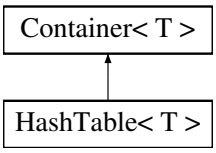
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Containers.h**

4.4 Dokumentacja szablonu klasy HashTable< T >

`#include <HashTable.h>`

Diagram dziedziczenia dla HashTable< T >



Metody publiczne

- **HashTable** ()
- **HashTable** (T value)
- **HashTable** (T value, int count)
- **HashTable** (**HashTable** &other)
- ~**HashTable** ()
- **HashTable**< T > & **operator=** (**HashTable** &rhs)
- bool **operator==** (**HashTable** &rhs)
- bool **operator!=** (**HashTable** &rhs)
- void **PopMax** ()
- void **PopMin** ()
- int **findValue** (T value)
- bool **findIfExist** (T value)
- **HashTableNode**< T > * **findMax** ()
- T **findMaxValue** ()
- **HashTableNode**< T > * **findMin** ()
- T **findMinValue** ()
- void **Insert** (T value)
- void **Insert** (T value, int amoundToAdd)
- void **Emplace** (T oldValue, T newValue)
- void **Swap** (**HashTable** &rhs)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** ()
- void **Erase** (T value)
- int **MaxSize** ()
- void **Show** ()

Metody prywatne

- int **HashFunction** (T value)
- void **addToBranch** (**HashTableNode**< T > *branch, int amoundToAdd)
- int **lengthOfBranch** (**HashTableNode**< T > *branch)
- void **deleteBranch** (**HashTableNode**< T > *branch)
- string **showType** ()

Atrybuty prywatne

- **HashTableNode::HashTableNode**< T > * **dataHash**
- int **setValues**

Statyczne atrybuty prywatne

- static const int **maxSize** = 200

4.4.1 Opis szczegółowy

template<**class T**>
class HashTable< **T** >

Deklaracja klasy **HashTable** (str. 18)

- Odpowiednik tablicy hashującej.

Parametry

key	Dynamicznie zaalokowana tablica **HashTableNode** (str. 29)
size	Liczba wpisanych elementów

4.4.2 Dokumentacja konstruktora i destruktora

4.4.2.1 HashTable() [1/4]

```
template<class T >
HashTable< T >::  HashTable ( )  [inline]
```

Konstruktor domyślny

4.4.2.2 HashTable() [2/4]

```
template<class T >
HashTable< T >::  HashTable (
            T value )  [inline]
```

Konstruktor jednoargumentowy.

- Dodaje podaną wartość na element o indeksie wykawanym przez funkcję hashującą

Parametry

value	Wartość do wpisania.

4.4.2.3 HashTable() [3/4]

```
template<class T >
HashTable< T >::  HashTable (
             T value,
             int count )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną wartość na element o indeksie wykawanym przez funkcję hashującą

Parametry

value	Wartość do wpisania.
count	Liczba elementów do stworenia

4.4.2.4 HashTable() [4/4]

```
template<class T >
HashTable< T >::  HashTable (
              HashTable< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.4.2.5 ~HashTable()

```
template<class T >
HashTable< T >::~ HashTable ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.4.3 Dokumentacja funkcji składowych

4.4.3.1 addToBranch()

```
template<class T >
void  HashTable< T >::addToBranch (
              HashTableNode< T > * branch,
             int amoundToAdd )  [inline], [private]
```

Dodaje nowe elementy do określonej gałęzi tablicy hashującej

Parametry

branch	Głąź do której dodajemy
amoundToAdd	Ilość elementów do dodania

4.4.3.2 allowedType()

```
template<class T >
bool  HashTable< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Container< T >** (str. 16).

4.4.3.3 Clear()

```
template<class T >
void  HashTable< T >::Clear ( )  [inline], [virtual]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Container< T >** (str. 16).

4.4.3.4 deleteBranch()

```
template<class T >
void  HashTable< T >::deleteBranch (
              HashTableNode< T > * branch )  [inline], [private]
```

Usuwanie elementów z gałęzi

Parametry

branch	Gałąź do usunięcia

4.4.3.5 Emplace()

```
template<class T >
void  HashTable< T >::Emplace (
            T oldValue,
            T newValue )  [inline]
```

Wstawienie nowego elementu na podaną pozycje.

Parametry

oldValue	Element do zastąpienia.
newValue	Element zastępujący.

4.4.3.6 Empty()

```
template<class T >
bool  HashTable< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Container< T >** (str. 16).

4.4.3.7 Erase()

```
template<class T >
void  HashTable< T >::Erase (
            T value )  [inline]
```

Usuwa wartość z tablicy hashującej o ile istnieje

Parametry

value	Wartość do usunięcia

4.4.3.8 findIfExist()

```
template<class T >
bool  HashTable< T >::findIfExist (
            T value )  [inline]
```

Sprawdzenie czy element o podanej wartości znajduje się w kontenerze.

Parametry

value	Wartość do sprawdzenia

Zwraca

Status operacji: true, jeżeli istnieje taki element; false, jeśli nie.

4.4.3.9 findMax()

```
template<class T >
HashTableNode<T>*  HashTable< T >::findMax ( )  [inline]
```

Zwrócenie pointera na największą wartość w konternerze.

Zwraca

Pointer na maxa.

4.4.3.10 findMaxValue()

```
template<class T >
T  HashTable< T >::findMaxValue ( )  [inline]
```

Zwrócenie największej wartości w konternerze.

Zwraca

Wartość max.

4.4.3.11 findMin()

```
template<class T >
HashTableNode<T>* HashTable< T >::findMin ( ) [inline]
```

Zwrócenie pointera na najmniejszą wartość w konternerze.

Zwraca

Pointer na mina.

4.4.3.12 findMinValue()

```
template<class T >
T HashTable< T >::findMinValue ( ) [inline]
```

Zwrócenie najmniejszej wartości w konternerze.

Zwraca

Wartość min.

4.4.3.13 findValue()

```
template<class T >
int HashTable< T >::findValue (
            T value ) [inline]
```

Zwraca pozycje podanej wartości

Parametry

value	Wartość do sprawdzenia

Zwraca

Indeks danej wartości lub -1 gdy nie występuje

4.4.3.14 HashFunction()

```
template<class T >
int HashTable< T >::HashFunction (
            T value ) [inline], [private]
```

Funkcja hashująca.

Parametry

value	Wartość do zashaszowania.

Zwraca

Hash

4.4.3.15 Insert() [1/2]

```
template<class T >
void  HashTable< T >::Insert (
            T value )  [inline]
```

Wprowadza wartość do tablicy haszującej z uwzględnieniem zabezpieczeń

Parametry

value	Wartość do wprowadzenia

4.4.3.16 Insert() [2/2]

```
template<class T >
void  HashTable< T >::Insert (
            T value,
            int amoundToAdd )  [inline]
```

Wprowadza wartość do tablicy haszującej z uwzględnieniem zabezpieczeń

Parametry

value	Wartość do wprowadzenia
amoundToAdd	Liczba danych do wprowadzenia

4.4.3.17 lengthOfBranch()

```
template<class T >
int  HashTable< T >::lengthOfBranch (
              HashTableNode< T > * branch )  [inline], [private]
```

Zlicza ilość elementów w danej gałęzi

Parametry

branch	gałąź do sprawdzenia

Zwraca

liczba elementów w gałęzi

4.4.3.18 MaxSize()

```
template<class T >
int  HashTable< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Container< T >** (str. 16).

4.4.3.19 operator"!=()

```
template<class T >
bool  HashTable< T >::operator!= (
              HashTable< T > & rhs )  [inline]
```

Operator przeciążony porównania "!="

Parametry

rhs	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.4.3.20 operator=()

```
template<class T >
HashTable<T>&  HashTable< T >::operator= (
              HashTable< T > & rhs )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.4.3.21 operator==()

```
template<class T >
bool  HashTable< T >::operator== (
              HashTable< T > & rhs )  [inline]
```

Operator przeciążony porównania "=="

Parametry

rhs	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.4.3.22 PopMax()

```
template<class T >
void  HashTable< T >::PopMax ( )  [inline]
```

Usunięcie największego elementu i zmniejszenie wielkości o jeden.

4.4.3.23 PopMin()

```
template<class T >
void  HashTable< T >::PopMin ( )  [inline]
```

Usunięcie najmniejszego elementu i zmniejszenie wielkości o jeden.

4.4.3.24 returnEmpty()

```
template<class T >
T  HashTable< T >::returnEmpty ( )  [inline]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

4.4.3.25 Show()

```
template<class T >
void  HashTable< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Container< T >** (str. 17).

4.4.3.26 showType()

```
template<class T >
string  HashTable< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.4.3.27 Size()

```
template<class T >
int  HashTable< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Container< T >** (str. 17).

4.4.3.28 Swap()

```
template<class T >
void  HashTable< T >::Swap (
              HashTable< T > & rhs )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.4.4 Dokumentacja atrybutów składowych

4.4.4.1 dataHash

```
template<class T >
HashTableNode::HashTableNode<T>*  HashTable< T >::dataHash  [private]
```

4.4.4.2 maxSize

```
template<class T >
const int  HashTable< T >::maxSize = 200  [static], [private]
```

4.4.4.3 setValues

```
template<class T >
int  HashTable< T >::setValues  [private]
```

Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **HashTable.h**

4.5 Dokumentacja szablonu klasy HashTableNode< T >

```
#include <Nudes.h>
```

Metody publiczne

- **HashTableNode** ()
- **HashTableNode** (int _key, T _data, **HashTableNode** *_pNext)
- T **returnEmpty** ()

Atrybuty publiczne

- int **key**
- T **data**
- **HashTableNode** * **pNext**

4.5.1 Opis szczegółowy

template<class T>
class HashTableNode< T >

-Klasa pomocnicza klasy **HashTable** (str. 18)

Parametry

key	klucz dostępu do danych
data	dane pod adresem zamieszczonym w kluczu
pNext	wskaźnik na następny element

4.5.2 Dokumentacja konstruktora i destruktora

4.5.2.1 HashTableNode() [1/2]

```
template<class T >
HashTableNode< T >::  HashTableNode ( )  [inline]
```

Konstruktor domyślny

4.5.2.2 HashTableNode() [2/2]

```
template<class T >
HashTableNode< T >::  HashTableNode (
            int _key,
            T _data,
             HashTableNode< T > * _pNext )  [inline]
```

Konstruktor trójargumentowy

Parametry

_key	klucz dostępu do danych
_data	dane pod adresem zamieszczonym w kluczu
_pNext	wskaźnik na następny element

4.5.3 Dokumentacja funkcji składowych

4.5.3.1 returnEmpty()

```
template<class T >
T  HashTableNode< T >::returnEmpty ( )  [inline]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

4.5.4 Dokumentacja atrybutów składowych

4.5.4.1 data

```
template<class T >
T  HashTableNode< T >::data
```

4.5.4.2 key

```
template<class T >
int  HashTableNode< T >::key
```

4.5.4.3 pNext

```
template<class T >
HashTableNode*  HashTableNode< T >::pNext
```

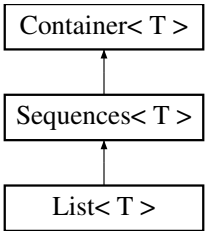
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Nudes.h**

4.6 Dokumentacja szablonu klasy List< T >

```
#include <List.h>
```

Diagram dziedziczenia dla List< T >



Metody publiczne

- **List** ()
- **List** (T value)
- **List** (T value, int count)
- **List** (int count, T ∗valuePtr)
- **List** (int count, **ListNode::ListNode**< T > ∗ptr)
- **List** (**List** &other)
- ∼**List** ()
- **List**< T > & **operator=** (**List** &value)
- bool **operator==** (**List** &rhs)
- bool **operator!=** (**List** &rhs)
- T **operator[ ]** (int index)
- void **PopBack** ()
- void **PopFront** ()
- void **PushBack** (const T &value)
- void **PushFront** (const T &value)
- **ListNode**< T > & **Back** ()
- **ListNode**< T > ∗ **End** () noexcept
- **ListNode**< T > & **Front** ()
- **ListNode**< T > ∗ **Begin** () noexcept
- **ListNode**< T > & **getElement** (int index)
- **ListNode**< T > ∗ **getElementPtr** (int index)
- T & **getElementValue** (int index)
- void **Insert** (int position, const T &value)
- void **Insert** (int position, int n, const T &value)
- void **Insert** (int position, int n, T ∗valuePtr)
- void **Emplace** (T value, int position)
- void **Assign** (int count, **ListNode**< T > ∗element)
- void **Assign** (int count, T ∗valuePtr)
- void **Assign** (int count, T value)
- void **Swap** (**List** &l2)
- void **Splice** (int position, **List** &x)
- **ListNode**< T > ∗ **findElement** (T value)
- bool **findIfExist** (const T value)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** ()
- void **Erase** (int position)
- void **Erase** (int first, int last)
- void **Remove** (T value)
- int **MaxSize** ()
- void **Resize** (int n)
- void **Resize** (int n, const T &value)
- void **Sort** ()
- void **Reverse** () noexcept
- void **Show** ()

Metody prywatne

- string **showType** ()

Atrybuty prywatne

- **ListNode::ListNode**< T > ∗ **pHead**

Przyjaciele

- ostream & **operator**<< (ostream &ostr, const **List** &list)

4.6.1 Opis szczegółowy

template<**class T**>
class List< **T** >

Deklaracja klasy **List** (str. 31)

- Odpowiednik STLowej listy jednokierunkowej.
- Dziedzicząca po **Sequences** (str. 143).

Parametry

pHead	Głowa listy jednokierunkowej.

4.6.2 Dokumentacja konstruktora i destruktora

4.6.2.1 List() [1/6]

```
template<class T >
List< T >::  List ( )  [inline]
```

Konstruktor domyślny

4.6.2.2 List() [2/6]

```
template<class T >
List< T >::  List (
             T value )  [inline]
```

Konstruktor jednoargumentowy.

- Dodaje podaną wartość jako pierwszy element.

Parametry

value	Wartość do wpisania.

4.6.2.3 List() [3/6]

```
template<class T >
List< T >::  List (
             T value,
             int count )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o podanej wartości.

Parametry

value	Wartość do wpisania.
count	Ilość elementów do stworzenia.

4.6.2.4 List() [4/6]

```
template<class T >
List< T >::  List (
             int count,
             T * valuePtr )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o wartościach ze wskaźnika.

Parametry

count	Ilość elementów do stworzenia.
valuePtr	Wskaźnik na wartości.

4.6.2.5 List() [5/6]

```
template<class T >
List< T >::  List (
```

```
            int count,
             ListNode::ListNode< T > * ptr )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o wartościach z elemntów innego kontenera.

Parametry

count	Ilość elementów do stworzenia.
ptr	Wskaźnik na wartości innego kontenera.

4.6.2.6 List() [6/6]

```
template<class T >
List< T >::  List (
              List< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.6.2.7 ∼List()

```
template<class T >
List< T >::∼ List ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.6.3 Dokumentacja funkcji składowych

4.6.3.1 allowedType()

```
template<class T >
bool  List< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Sequences< T >** (str. 144).

4.6.3.2 Assign() [1/3]

```
template<class T >
void  List< T >::Assign (
             int count,
              ListNode< T > * element )  [inline]
```

Zastąpienie starych elementów podanymi z innego kontenera.

Parametry

count	Ilość elementów do zastąpienia.
element	Nowe elementy.

4.6.3.3 Assign() [2/3]

```
template<class T >
void  List< T >::Assign (
             int count,
             T * valuePtr )  [inline], [virtual]
```

Zastąpienie starych elementów podanymi ze wskaźnika/tablicy.

Parametry

count	Ilość elementów do zastąpienia.
valuePtr	Pointer na nowe elementy.

Implementuje **Sequences< T >** (str. 144).

4.6.3.4 Assign() [3/3]

```
template<class T >
void  List< T >::Assign (
             int count,
             T value )  [inline], [virtual]
```

Zastąpienie starych elementów podaną wartością.

Parametry

count	Ilość elementów do zastąpienia.
value	Nowa wartość.

Implementuje **Sequences< T >** (str. 144).

4.6.3.5 Back()

```
template<class T >
ListNode<T>&  List< T >::Back ( )  [inline]
```

Zwrócenie referencji ostatniego elementu.

Zwraca

Referencja na ostatni element.

4.6.3.6 Begin()

```
template<class T >
ListNode<T>*  List< T >::Begin ( )  [inline], [noexcept]
```

Zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.6.3.7 Clear()

```
template<class T >
void  List< T >::Clear ( )  [inline], [virtual]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Sequences< T >** (str. 145).

4.6.3.8 Emplace()

```
template<class T >
void  List< T >::Emplace (
            T value,
            int position )  [inline], [virtual]
```

Wstawienie nowego elementu na podaną pozycje.

Parametry

value	Nowy element.
position	Pozycja, na którą metoda wstawia element.

Implementuje **Sequences< T >** (str. 145).

4.6.3.9 Empty()

```
template<class T >
bool  List< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Sequences< T >** (str. 145).

4.6.3.10 End()

```
template<class T >
ListNode<T>*  List< T >::End ( )  [inline], [noexcept]
```

Zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.6.3.11 Erase() [1/2]

```
template<class T >
void  List< T >::Erase (
            int first,
            int last )  [inline], [virtual]
```

Usunięcie wybranych elementów z przedziału.

Parametry

first	Pierwszy indeks przedziału.
last	Ostatni indeks przedziału.

Implementuje **Sequences< T >** (str. 146).

4.6.3.12 Erase() [2/2]

```
template<class T >
void  List< T >::Erase (
            int position )  [inline], [virtual]
```

Usunięcie wybranego elementu po pozycji.

Parametry

position	Pozycja elementu do usunięcia.

Implementuje **Sequences< T >** (str. 146).

4.6.3.13 findElement()

```
template<class T >
ListNode<T>*  List< T >::findElement (
            T value )  [inline]
```

Szukanie elementu po podanej wartości.

Parametry

value	Wartość elementu do odszukania.

Zwraca

Indeks elementu, jeżeli został odnaleziony.

4.6.3.14 findIfExist()

```
template<class T >
bool  List< T >::findIfExist (
            const T value )  [inline], [virtual]
```

Sprawdzenie czy element istnieje, szukany po wartości.

Parametry

value	Szukana wartość.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Sequences**< **T** > (str. 146).

4.6.3.15 Front()

```
template<class T >
ListNode<T>&  List< T >::Front ( )  [inline]
```

Zwrócenie referencji pierwszego elementu.

Zwraca

Referencja na pierwszy element.

4.6.3.16 getElement()

```
template<class T >
ListNode<T>&  List< T >::getElement (
             int index )  [inline]
```

Zwrócenie referencji elementu spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Referencja na dany element.

4.6.3.17 getElementPtr()

```
template<class T >
ListNode<T>*  List< T >::getElementPtr (
             int index )  [inline]
```

Zwrócenie wskaźnika na element spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Wskaźnik na dany element.

4.6.3.18 getElementValue()

```
template<class T >
T&  List< T >::getElementValue (
              int index )  [inline]
```

Zwrócenie wartości elementu spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Wartość danego elementu.

4.6.3.19 Insert() [1/3]

```
template<class T >
void  List< T >::Insert (
              int position,
              const T & value )  [inline], [virtual]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
value	Nowy element.

Implementuje **Sequences< T >** (str. 147).

4.6.3.20 Insert() [2/3]

```
template<class T >
void  List< T >::Insert (
            int position,
            int n,
            const T & value )  [inline], [virtual]
```

Wypełnienie kontenera elementem przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
n	Ilość kopii nowego elementu do wstawienia.
value	Nowy element.

Implementuje **Sequences< T >** (str. 147).

4.6.3.21 Insert() [3/3]

```
template<class T >
void  List< T >::Insert (
            int position,
            int n,
            T * valuePtr )  [inline], [virtual]
```

Wypełnienie kontenera przedziałem przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
n	Ilość elementów do wstawienia.
valuePtr	Pointer na nowe elementy.

Implementuje **Sequences< T >** (str. 148).

4.6.3.22 MaxSize()

```
template<class T >
int  List< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Sequences< T >** (str. 148).

4.6.3.23 operator"!=()

```
template<class T >
bool  List< T >::operator!= (
              List< T > & rhs )  [inline]
```

Operator przeciążony porównania "!="

Parametry

rhs	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.6.3.24 operator=()

```
template<class T >
List<T>&  List< T >::operator= (
              List< T > & value )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.6.3.25 operator==()

```
template<class T >
bool  List< T >::operator== (
              List< T > & rhs )  [inline]
```

Operator przeciążony porównania "=="

Parametry

rhs	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.6.3.26 operator[]()

```
template<class T >
T  List< T >::operator[] (
            int index )  [inline]
```

Operator przeciążony indeksowy "[ ]"

Parametry

index	Indeks elementu.

Zwraca

Wartość elementu spod podanego indeksu.

4.6.3.27 PopBack()

```
template<class T >
void  List< T >::PopBack ( )  [inline], [virtual]
```

Usunięcie ostatniego elementu i zmniejszenie wielkości o jeden.

Implementuje **Sequences< T >** (str. 149).

4.6.3.28 PopFront()

```
template<class T >
void  List< T >::PopFront ( )  [inline], [virtual]
```

Usunięcie pierwszego elementu i zmniejszenie wielkości o jeden.

Implementuje **Sequences< T >** (str. 149).

4.6.3.29 PushBack()

```
template<class T >
void  List< T >::PushBack (
            const T & value )  [inline], [virtual]
```

Dodanie nowego elementu na końcu kontenera, po ostatnimm oraz zwiększenie wielkości o jeden.

Parametry

value	Nowy element.

Implementuje **Sequences< T >** (str. 149).

4.6.3.30 PushFront()

```
template<class T >
void  List< T >::PushFront (
            const T & value )  [inline], [virtual]
```

Dodanie nowego elementu na początku kontenera, po ostatnimm oraz zwiększenie wielkości o jeden.

Parametry

value	Nowy element.

Implementuje **Sequences< T >** (str. 150).

4.6.3.31 Remove()

```
template<class T >
void  List< T >::Remove (
            T value )  [inline]
```

Usunięcie wybranego elementu ze zwolnieniem pamięci.

Parametry

value	Wartość elementu do usunięcia.

4.6.3.32 Resize() [1/2]

```
template<class T >
void  List< T >::Resize (
            int n )  [inline], [virtual]
```

Zmiana wielkości kontenera na podaną.

- Mniejsza niż aktualna: ucięcie kontenera.
- Większa niż aktualna: nowe elementy są puste.

Parametry

n	Nowa wielkość kontenera.

Implementuje **Sequences**< **T** > (str. 150).

4.6.3.33 Resize() [2/2]

```
template<class T >
void  List< T >::Resize (
            int n,
            const T & value )  [inline], [virtual]
```

Zmiana wielkości kontenera na podaną i dodanie nowych elementów.

- Mniejsza niż aktualna: ucięcie kontenera.
- Większa niż aktualna: nowe elementy mają podaną wartość.

Parametry

n	Nowa wielkość kontenera.
value	Wartość nowych elementów.

Implementuje **Sequences**< **T** > (str. 150).

4.6.3.34 returnEmpty()

```
template<class T >
T  List< T >::returnEmpty ( )  [inline], [virtual]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

Implementuje **Sequences**< **T** > (str. 151).

4.6.3.35 Reverse()

```
template<class T >
void  List< T >::Reverse ( )  [inline], [virtual], [noexcept]
```

Odwrócenie elementów kontenera kolejnością.

Implementuje **Sequences< T >** (str. 151).

4.6.3.36 Show()

```
template<class T >
void  List< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Sequences< T >** (str. 151).

4.6.3.37 showType()

```
template<class T >
string  List< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.6.3.38 Size()

```
template<class T >
int  List< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Sequences< T >** (str. 151).

4.6.3.39 Sort()

```
template<class T >
void  List< T >::Sort ( )  [inline], [virtual]
```

Sortowanie elementów w kontenerze

Implementuje **Sequences< T >** (str. 152).

4.6.3.40 Splice()

```
template<class T >
void  List< T >::Splice (
             int position,
              List< T > & x )  [inline]
```

Przepięcie elementów innego kontenera na odpowiednią pozycję.

Parametry

position	Pozycja na przepięcie.
x	Wskaźnik na inny kontener.

4.6.3.41 Swap()

```
template<class T >
void  List< T >::Swap (
              List< T > & l2 )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.6.4 Dokumentacja przyjaciół i funkcji związanych

4.6.4.1 operator<<

```
template<class T >
ostream& operator<< (
```

```
ostream & ostr,
const  List< T > & list )  [friend]
```

Operator przeciążony strumienia "<<"

Parametry

ostr	Lewostronny operand potrzebny funkcji do funkcjonowania.
list	Wskaźnik na inny kontener.

Zwraca

Zwrócenie Operandu.

4.6.5 Dokumentacja atrybutów składowych

4.6.5.1 pHead

```
template<class T >
ListNode::ListNode<T>*  List< T >::pHead  [private]
```

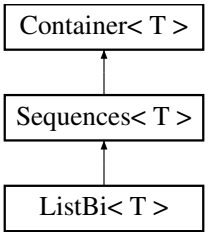
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **List.h**

4.7 Dokumentacja szablonu klasy ListBi< T >

```
#include <ListBi.h>
```

Diagram dziedziczenia dla ListBi< T >



Metody publiczne

- **ListBi** ()
- **ListBi** (T value)
- **ListBi** (T value, int count)
- **ListBi** (int count, T ∗valuePtr)
- **ListBi** (int count, **ListBiNode::ListBiNode**< T > ∗ptr)
- **ListBi** (const **ListBi** &other)
- ∼**ListBi** ()
- **ListBi**< T > & **operator=** (const **ListBi** &value)
- bool **operator==** (const **ListBi** &rhs) const
- bool **operator!=** (const **ListBi** &rhs) const
- T **operator[ ]** (int index)
- void **PopBack** ()
- void **PopFront** ()
- void **PushBack** (const T &value)
- void **PushFront** (const T &value)
- **ListBiNode**< T > & **Back** ()
- **ListBiNode**< T > ∗ **End** () const noexcept
- **ListBiNode**< T > & **Front** ()
- **ListBiNode**< T > ∗ **Begin** () const noexcept
- **ListBiNode**< T > & **getElement** (int index)
- **ListBiNode**< T > ∗ **getElementPtr** (int index)
- T & **getElementValue** (int index)
- void **Insert** (int position, const T &value)
- void **Insert** (int position, int n, const T &value)
- void **Insert** (int position, int n, T ∗valuePtr)
- void **Emplace** (T value, int position)
- void **Assign** (int count, **ListBiNode**< T > ∗element)
- void **Assign** (int count, T ∗valuePtr)
- void **Assign** (int count, const T value)
- void **Swap** (**ListBi** &l2)
- void **Splice** (int position, **ListBi** &x)
- **ListBiNode**< T > ∗ **findElement** (const T value)
- bool **findIfExist** (const T value)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** () noexcept
- void **Erase** (int position)
- void **Erase** (int first, int last)
- void **Remove** (const T &value)
- int **MaxSize** ()
- void **Resize** (int n)
- void **Resize** (int n, const T &value)
- void **Sort** ()
- void **Reverse** () noexcept
- void **Show** ()

Metody prywatne

- string **showType** ()

Atrybuty prywatne

- **ListBiNode**< T > ∗ **pHead**

Przyjaciele

- ostream & **operator**<< (ostream &ostr, const **ListBi**< T > &list)

4.7.1 Opis szczegółowy

template<**class T**>
class ListBi< **T** >

Deklaracja klasy **ListBi** (str. 49)

- Odpowiednik STLowej listy dwukierunkowej.
- Dziedzicząca po **Sequences** (str. 143).

Parametry

pHead	Głowa listy dwukierunkowej.

4.7.2 Dokumentacja konstruktora i destruktora

4.7.2.1 ListBi() [1/6]

```
template<class T >
ListBi< T >::  ListBi ( )  [inline]
```

Konstruktor domyślny

4.7.2.2 ListBi() [2/6]

```
template<class T >
ListBi< T >::  ListBi (
             T value )  [inline]
```

Konstruktor jednoargumentowy.

- Dodaje podaną wartość jako pierwszy element.

Parametry

value	Wartość do wpisania.

4.7.2.3 ListBi() [3/6]

```
template<class T >
ListBi< T >::  ListBi (
            T value,
            int count )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o podanej wartości.

Parametry

value	Wartość do wpisania.
count	Ilość elementów do stworzenia.

4.7.2.4 ListBi() [4/6]

```
template<class T >
ListBi< T >::  ListBi (
            int count,
            T * valuePtr )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o wartościach ze wskaźnika.

Parametry

count	Ilość elementów do stworzenia.
valuePtr	Wskaźnik na wartości.

4.7.2.5 ListBi() [5/6]

```
template<class T >
ListBi< T >::  ListBi (
```

```
            int count,
             ListBiNode::ListBiNode< T > * ptr )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o wartościach z elemntów innego kontenera.

Parametry

count	Ilość elementów do stworzenia.
ptr	Wskaźnik na wartości innego kontenera.

4.7.2.6 ListBi() [6/6]

```
template<class T >
ListBi< T >::  ListBi (
             const  ListBi< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.7.2.7 ~ListBi()

```
template<class T >
ListBi< T >::~ ListBi ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.7.3 Dokumentacja funkcji składowych

4.7.3.1 allowedType()

```
template<class T >
bool  ListBi< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Sequences< T >** (str. 144).

4.7.3.2 Assign() [1/3]

```
template<class T >
void  ListBi< T >::Assign (
             int count,
             const T value )  [inline], [virtual]
```

Zastąpienie starych elementów podaną wartością.

Parametry

count	Ilość elementów do zastąpienia.
value	Nowa wartość.

Implementuje **Sequences< T >** (str. 144).

4.7.3.3 Assign() [2/3]

```
template<class T >
void  ListBi< T >::Assign (
             int count,
              ListBiNode< T > * element )  [inline]
```

Zastąpienie starych elementów podanymi z innego kontenera.

Parametry

count	Ilość elementów do zastąpienia.
element	Nowe elementy.

4.7.3.4 Assign() [3/3]

```
template<class T >
void  ListBi< T >::Assign (
            int count,
            T * valuePtr )  [inline], [virtual]
```

Zastąpienie starych elementów podanymi ze wskaźnika/tablicy.

Parametry

count	Ilość elementów do zastąpienia.
valuePtr	Pointer na nowe elementy.

Implementuje **Sequences< T >** (str. 144).

4.7.3.5 Back()

```
template<class T >
ListBiNode<T>&  ListBi< T >::Back ( )  [inline]
```

Zwrócenie referencji ostatniego elementu.

Zwraca

Referencja na ostatni element.

4.7.3.6 Begin()

```
template<class T >
ListBiNode<T>*  ListBi< T >::Begin ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.7.3.7 Clear()

```
template<class T >
void  ListBi< T >::Clear ( )  [inline], [virtual], [noexcept]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Sequences< T >** (str. 145).

4.7.3.8 Emplace()

```
template<class T >
void  ListBi< T >::Emplace (
            T value,
            int position )  [inline], [virtual]
```

Wstawienie nowego elementu na podaną pozycje.

Parametry

value	Nowy element.
position	Pozycja, na którą metoda wstawia element.

Implementuje **Sequences< T >** (str. 145).

4.7.3.9 Empty()

```
template<class T >
bool  ListBi< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Sequences< T >** (str. 145).

4.7.3.10 End()

```
template<class T >
ListBiNode<T>*  ListBi< T >::End ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.7.3.11 Erase() [1/2]

```
template<class T >
void  ListBi< T >::Erase (
            int first,
            int last )  [inline], [virtual]
```

Usunięcie wybranych elementów z przedziału.

Parametry

first	Pierwszy indeks przedziału.
last	Ostatni indeks przedziału.

Implementuje **Sequences< T >** (str. 146).

4.7.3.12 Erase() [2/2]

```
template<class T >
void  ListBi< T >::Erase (
            int position )  [inline], [virtual]
```

Usunięcie wybranego elementu po pozycji.

Parametry

position	Pozycja elementu do usunięcia.

Implementuje **Sequences< T >** (str. 146).

4.7.3.13 findElement()

```
template<class T >
ListBiNode<T>*  ListBi< T >::findElement (
            const T value )  [inline]
```

Szukanie elementu po podanej wartości.

Parametry

value	Wartość elementu do odszukania.

Zwraca

Indeks elementu, jeżeli został odnaleziony.

4.7.3.14 findIfExist()

```
template<class T >
bool  ListBi< T >::findIfExist (
            const T value )  [inline], [virtual]
```

Sprawdzenie czy element istnieje, szukany po wartości.

Parametry

value	Szukana wartość.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Sequences**< **T** > (str. 146).

4.7.3.15 Front()

```
template<class T >
ListBiNode<T>&  ListBi< T >::Front ( )  [inline]
```

Zwrócenie referencji pierwszego elementu.

Zwraca

Referencja na pierwszy element.

4.7.3.16 getElement()

```
template<class T >
ListBiNode<T>&  ListBi< T >::getElement (
             int index )  [inline]
```

Zwrócenie referencji elementu spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Referencja na dany element.

4.7.3.17 getElementPtr()

```
template<class T >
ListBiNode<T>*  ListBi< T >::getElementPtr (
             int index )  [inline]
```

Zwrócenie wskaźnika na element spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Wskaźnik na dany element.

4.7.3.18 getElementValue()

```
template<class T >
T&  ListBi< T >::getElementValue (
              int index )  [inline]
```

Zwrócenie wartości elementu spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Wartość danego elementu.

4.7.3.19 Insert() [1/3]

```
template<class T >
void  ListBi< T >::Insert (
              int position,
              const T & value )  [inline], [virtual]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
value	Nowy element.

Implementuje **Sequences< T >** (str. 147).

4.7.3.20 Insert() [2/3]

```
template<class T >
void  ListBi< T >::Insert (
            int position,
            int n,
            const T & value )  [inline], [virtual]
```

Wypełnienie kontenera elementem przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
n	Ilość kopii nowego elementu do wstawienia.
value	Nowy element.

Implementuje **Sequences< T >** (str. 147).

4.7.3.21 Insert() [3/3]

```
template<class T >
void  ListBi< T >::Insert (
            int position,
            int n,
            T * valuePtr )  [inline], [virtual]
```

Wypełnienie kontenera przedziałem przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
n	Ilość elementów do wstawienia.
valuePtr	Pointer na nowe elementy.

Implementuje **Sequences< T >** (str. 148).

4.7.3.22 MaxSize()

```
template<class T >
int  ListBi< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Sequences< T >** (str. 148).

4.7.3.23 operator"!=()

```
template<class T >
bool  ListBi< T >::operator!= (
            const  ListBi< T > & rhs ) const  [inline]
```

Operator przeciążony porównania "!="

Parametry

rhs	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.7.3.24 operator=()

```
template<class T >
ListBi<T>&  ListBi< T >::operator= (
            const  ListBi< T > & value )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.7.3.25 operator==()

```
template<class T >
bool  ListBi< T >::operator== (
            const  ListBi< T > & rhs ) const  [inline]
```

Operator przeciążony porównania "=="

Parametry

rhs	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.7.3.26 operator[]()

```
template<class T >
T  ListBi< T >::operator[] (
            int index )  [inline]
```

Operator przeciążony indeksowy "[ ]"

Parametry

index	Indeks elementu.

Zwraca

Wartość elementu spod podanego indeksu.

4.7.3.27 PopBack()

```
template<class T >
void  ListBi< T >::PopBack ( )  [inline], [virtual]
```

Usunięcie ostatniego elementu i zmniejszenie wielkości o jeden.

Implementuje **Sequences< T >** (str. 149).

4.7.3.28 PopFront()

```
template<class T >
void  ListBi< T >::PopFront ( )  [inline], [virtual]
```

Usunięcie pierwszego elementu i zmniejszenie wielkości o jeden.

Implementuje **Sequences< T >** (str. 149).

4.7.3.29 PushBack()

```
template<class T >
void  ListBi< T >::PushBack (
            const T & value )  [inline], [virtual]
```

Dodanie nowego elementu na końcu kontenera, po ostatnimm oraz zwiększenie wielkości o jeden.

Parametry

value	Nowy element.

Implementuje **Sequences< T >** (str. 149).

4.7.3.30 PushFront()

```
template<class T >
void  ListBi< T >::PushFront (
            const T & value )  [inline], [virtual]
```

Dodanie nowego elementu na początku kontenera, po ostatnimm oraz zwiększenie wielkości o jeden.

Parametry

value	Nowy element.

Implementuje **Sequences< T >** (str. 150).

4.7.3.31 Remove()

```
template<class T >
void  ListBi< T >::Remove (
            const T & value )  [inline]
```

Usunięcie wybranego elementu ze zwolnieniem pamięci.

Parametry

value	Wartość elementu do usunięcia.

4.7.3.32 Resize() [1/2]

```
template<class T >
void  ListBi< T >::Resize (
            int n )  [inline], [virtual]
```

Zmiana wielkości kontenera na podaną.

- Mniejsza niż aktualna: ucięcie kontenera.
- Większa niż aktualna: nowe elementy są puste.

Parametry

n	Nowa wielkość kontenera.

Implementuje **Sequences**< **T** > (str. 150).

4.7.3.33 Resize() [2/2]

```
template<class T >
void  ListBi< T >::Resize (
            int n,
            const T & value )  [inline], [virtual]
```

Zmiana wielkości kontenera na podaną i dodanie nowych elementów.

- Mniejsza niż aktualna: ucięcie kontenera.
- Większa niż aktualna: nowe elementy mają podaną wartość.

Parametry

n	Nowa wielkość kontenera.
value	Wartość nowych elementów.

Implementuje **Sequences**< **T** > (str. 150).

4.7.3.34 returnEmpty()

```
template<class T >
T  ListBi< T >::returnEmpty ( )  [inline], [virtual]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

Implementuje **Sequences**< **T** > (str. 151).

4.7.3.35 Reverse()

```
template<class T >
void  ListBi< T >::Reverse ( )  [inline], [virtual], [noexcept]
```

Odwrócenie elementów kontenera kolejnością.

Implementuje **Sequences< T >** (str. 151).

4.7.3.36 Show()

```
template<class T >
void  ListBi< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Sequences< T >** (str. 151).

4.7.3.37 showType()

```
template<class T >
string  ListBi< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.7.3.38 Size()

```
template<class T >
int  ListBi< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Sequences< T >** (str. 151).

4.7.3.39 Sort()

```
template<class T >
void  ListBi< T >::Sort ( )  [inline], [virtual]
```

Sortowanie elementów w kontenerze

Implementuje **Sequences< T >** (str. 152).

4.7.3.40 Splice()

```
template<class T >
void  ListBi< T >::Splice (
            int position,
             ListBi< T > & x )  [inline]
```

Przepięcie elementów innego kontenera na odpowiednią pozycję.

Parametry

position	Pozycja na przepięcie.
x	Wskaźnik na inny kontener.

4.7.3.41 Swap()

```
template<class T >
void  ListBi< T >::Swap (
             ListBi< T > & l2 )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.7.4 Dokumentacja przyjaciół i funkcji związanych

4.7.4.1 operator<<

```
template<class T >
ostream& operator<< (
```

```
ostream & ostr,
const  ListBi< T > & list )  [friend]
```

Operator przeciążony strumienia "<<"

Parametry

ostr	Lewostronny operand potrzebny funkcji do funkcjonowania.
list	Wskaźnik na inny kontener.

Zwraca

Zwrócenie Operandu.

4.7.5 Dokumentacja atrybutów składowych

4.7.5.1 pHead

```
template<class T >
ListBiNode<T>*  ListBi< T >::pHead  [private]
```

Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **ListBi.h**

4.8 Dokumentacja szablonu klasy ListBiNode< T >

```
#include <Nudes.h>
```

Metody publiczne

- **ListBiNode** (T value, **ListBiNode** * **pNext**, **ListBiNode** * **pPrev**)

Atrybuty publiczne

- T **data**
- **ListBiNode** * **pNext**
- **ListBiNode** * **pPrev**

4.8.1 Opis szczegółowy

template<class T>
class ListBiNode< T >

Deklaracja klasy **ListBiNode** (str. 67)

- Klasa pomocnicza dla klasy **ListBi** (str. 49) i **ListCircle** (str. 69).

Parametry

data	Wartoœæ elementu.
pNext	WskaŸnik na nastêpny element.
pPrev	WskaŸnik na poprzedni element.

4.8.2 Dokumentacja konstruktora i destruktora

4.8.2.1 ListBiNode()

```
template<class T >
ListBiNode< T >::  ListBiNode (
             T value,
              ListBiNode< T > * pNext,
              ListBiNode< T > * pPrev )  [inline]
```

Konstruktor trójargumentowy

Parametry

value	Wartość do wpisania
pNext	Wskaźnik na następny element
pPrev	Wskaźnik na poprzedni element

4.8.3 Dokumentacja atrybutów składowych

4.8.3.1 data

```
template<class T >
T  ListBiNode< T >::data
```

4.8.3.2 pNext

```
template<class T >
ListBiNode*  ListBiNode< T >::pNext
```

4.8.3.3 pPrev

```
template<class T >
ListBiNode* ListBiNode< T >::pPrev
```

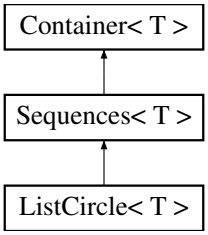
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Nudes.h**

4.9 Dokumentacja szablonu klasy ListCircle< T >

```
#include <ListCircle.h>
```

Diagram dziedziczenia dla ListCircle< T >



Metody publiczne

- **ListCircle** ()
- **ListCircle** (T value)
- **ListCircle** (T value, int count)
- **ListCircle** (int count, T *valuePtr)
- **ListCircle** (int count, **ListBiNode::ListBiNode**< T > *ptr)
- **ListCircle** (const **ListCircle** &other)
- ~**ListCircle** ()
- **ListCircle**< T > & **operator=** (const **ListCircle** &value)
- bool **operator==** (const **ListCircle** &rhs) const
- bool **operator!=** (const **ListCircle** &rhs) const
- T **operator[ ]** (const int index)
- void **PopBack** ()
- void **PopFront** ()
- void **PushBack** (const T &value)
- void **PushFront** (const T &value)
- **ListBiNode**< T > & **Back** ()
- **ListBiNode**< T > * **End** () const noexcept
- **ListBiNode**< T > & **Front** ()
- **ListBiNode**< T > * **Begin** () const noexcept
- **ListBiNode**< T > & **getElement** (int index)
- **ListBiNode**< T > * **getElementPtr** (int index)
- T & **getElementValue** (int index)
- void **Insert** (int position, const T &value)
- void **Insert** (int position, int n, const T &value)
- void **Insert** (int position, int n, T *valuePtr)

- void **Emplace** (T value, int position)
- void **Assign** (int count, **ListBiNode**< T > ∗element)
- void **Assign** (int count, T ∗valuePtr)
- void **Assign** (int count, const T value)
- void **Swap** (**ListCircle** &l2)
- void **Splice** (int position, **ListCircle** &x)
- **ListBiNode**< T > ∗ **findElement** (const T value)
- bool **findIfExist** (const T value)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** () noexcept
- void **Erase** (int position)
- void **Erase** (int first, int last)
- void **Remove** (const T &value)
- int **MaxSize** ()
- void **Resize** (int n)
- void **Resize** (int n, const T &value)
- void **Sort** ()
- void **Reverse** () noexcept
- void **Show** ()

Metody prywatne

- string **showType** ()

Atrybuty prywatne

- **ListBiNode**< T > ∗ **pHead**

Przyjaciele

- ostream & **operator**<< (ostream &ostr, const **ListCircle** &list)

4.9.1 Opis szczegółowy

template<**class T**>
class ListCircle< **T** >

Deklaracja klasy **ListCircle** (str. 69)

- Odpowiednik STLowej listy cyklicznej.
- Dziedzicząca po **Sequences** (str. 143).

Parametry

pHead	Głowa listy dwukierunkowej.

4.9.2 Dokumentacja konstruktora i destruktora

4.9.2.1 ListCircle() [1/6]

```
template<class T >
ListCircle< T >::  ListCircle ( )  [inline]
```

Konstruktor domyślny

4.9.2.2 ListCircle() [2/6]

```
template<class T >
ListCircle< T >::  ListCircle (
            T value )  [inline]
```

Konstruktor jednoargumentowy.

- Dodaje podaną wartość jako pierwszy element.

Parametry

value	Wartość do wpisania.

4.9.2.3 ListCircle() [3/6]

```
template<class T >
ListCircle< T >::  ListCircle (
            T value,
            int count )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o podanej wartości.

Parametry

value	Wartość do wpisania.
count	Ilość elementów do stworzenia.

4.9.2.4 ListCircle() [4/6]

```
template<class T >
ListCircle< T >::  ListCircle (
            int count,
            T * valuePtr )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o wartościach ze wskaźnika.

Parametry

count	Ilość elementów do stworzenia.
valuePtr	Wskaźnik na wartości.

4.9.2.5 ListCircle() [5/6]

```
template<class T >
ListCircle< T >::  ListCircle (
            int count,
             ListBiNode::ListBiNode< T > * ptr )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o wartościach z elemntów innego kontenera.

Parametry

count	Ilość elementów do stworzenia.
ptr	Wskaźnik na wartości innego kontenera.

4.9.2.6 ListCircle() [6/6]

```
template<class T >
ListCircle< T >::  ListCircle (
            const  ListCircle< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.9.2.7 ~ListCircle()

```
template<class T >
ListCircle< T >::~ ListCircle ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.9.3 Dokumentacja funkcji składowych

4.9.3.1 allowedType()

```
template<class T >
bool  ListCircle< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Sequences< T >** (str. 144).

4.9.3.2 Assign() [1/3]

```
template<class T >
void  ListCircle< T >::Assign (
            int count,
            const T value )  [inline], [virtual]
```

Zastąpienie starych elementów podaną wartością.

Parametry

count	Ilość elementów do zastąpienia.
value	Nowa wartość.

Implementuje **Sequences**< **T** > (str. 144).

4.9.3.3 Assign() [2/3]

```
template<class T >
void  ListCircle< T >::Assign (
            int count,
             ListBiNode< T > * element )  [inline]
```

Zastąpienie starych elementów podanymi z innego kontenera.

Parametry

count	Ilość elementów do zastąpienia.
element	Nowe elementy.

4.9.3.4 Assign() [3/3]

```
template<class T >
void  ListCircle< T >::Assign (
            int count,
            T * valuePtr )  [inline], [virtual]
```

Zastąpienie starych elementów podanymi ze wskaźnika/tablicy.

Parametry

count	Ilość elementów do zastąpienia.
valuePtr	Pointer na nowe elementy.

Implementuje **Sequences**< **T** > (str. 144).

4.9.3.5 Back()

```
template<class T >
ListBiNode<T>&  ListCircle< T >::Back ( )  [inline]
```

Zwrócenie referencji ostatniego elementu.

Zwraca

Referencja na ostatni element.

4.9.3.6 Begin()

```
template<class T >
ListBiNode<T>*  ListCircle< T >::Begin ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.9.3.7 Clear()

```
template<class T >
void  ListCircle< T >::Clear ( )  [inline], [virtual], [noexcept]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Sequences< T >** (str. 145).

4.9.3.8 Emplace()

```
template<class T >
void  ListCircle< T >::Emplace (
            T value,
            int position )  [inline], [virtual]
```

Wstawienie nowego elementu na podaną pozycje.

Parametry

value	Nowy element.
position	Pozycja, na którą metoda wstawia element.

Implementuje **Sequences< T >** (str. 145).

4.9.3.9 Empty()

```
template<class T >
bool  ListCircle< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Sequences**< **T** > (str. 145).

4.9.3.10 End()

```
template<class T >
ListBiNode<T>*  ListCircle< T >::End ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.9.3.11 Erase() [1/2]

```
template<class T >
void  ListCircle< T >::Erase (
            int first,
            int last )  [inline], [virtual]
```

Usunięcie wybranych elementów z przedziału.

Parametry

first	Pierwszy indeks przedziału.
last	Ostatni indeks przedziału.

Implementuje **Sequences**< **T** > (str. 146).

4.9.3.12 Erase() [2/2]

```
template<class T >
void  ListCircle< T >::Erase (
            int position )  [inline], [virtual]
```

Usunięcie wybranego elementu po pozycji.

Parametry

position	Pozycja elementu do usunięcia.

Implementuje **Sequences< T >** (str. 146).

4.9.3.13 findElement()

```
template<class T >
ListBiNode<T>*  ListCircle< T >::findElement (
             const T value )  [inline]
```

Szukanie elementu po podanej wartości.

Parametry

value	Wartość elementu do odszukania.

Zwraca

Indeks elementu, jeżeli został odnaleziony.

4.9.3.14 findIfExist()

```
template<class T >
bool  ListCircle< T >::findIfExist (
             const T value )  [inline], [virtual]
```

Sprawdzenie czy element istnieje, szukany po wartości.

Parametry

value	Szukana wartość.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Sequences< T >** (str. 146).

4.9.3.15 Front()

```
template<class T >
ListBiNode<T>&  ListCircle< T >::Front ( )  [inline]
```

Zwrócenie referencji pierwszego elementu.

Zwraca

Referencja na pierwszy element.

4.9.3.16 getElement()

```
template<class T >
ListBiNode<T>&  ListCircle< T >::getElement (
             int index )  [inline]
```

Zwrócenie referencji elementu spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Referencja na dany element.

4.9.3.17 getElementPtr()

```
template<class T >
ListBiNode<T>*  ListCircle< T >::getElementPtr (
             int index )  [inline]
```

Zwrócenie wskaźnika na element spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Wskaźnik na dany element.

4.9.3.18 getElementValue()

```
template<class T >
T&  ListCircle< T >::getElementValue (
             int index )  [inline]
```

Zwrócenie wartości elementu spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Wartość danego elementu.

4.9.3.19 Insert() [1/3]

```
template<class T >
void  ListCircle< T >::Insert (
            int position,
            const T & value )  [inline], [virtual]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
value	Nowy element.

Implementuje **Sequences< T >** (str. 147).

4.9.3.20 Insert() [2/3]

```
template<class T >
void  ListCircle< T >::Insert (
            int position,
            int n,
            const T & value )  [inline], [virtual]
```

Wypełnienie kontenera elementem przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
n	Ilość kopii nowego elementu do wstawienia.
value	Nowy element.

Implementuje **Sequences< T >** (str. 147).

4.9.3.21 Insert() [3/3]

```
template<class T >
void  ListCircle< T >::Insert (
            int position,
```

```
            int n,
            T * valuePtr )  [inline], [virtual]
```

Wypełnienie kontenera przedziałem przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
n	Ilość elementów do wstawienia.
valuePtr	Pointer na nowe elementy.

Implementuje **Sequences**< **T** > (str. 148).

4.9.3.22 MaxSize()

```
template<class T >
int  ListCircle< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Sequences**< **T** > (str. 148).

4.9.3.23 operator"!=()

```
template<class T >
bool  ListCircle< T >::operator!= (
            const  ListCircle< T > & rhs ) const  [inline]
```

Operator przeciążony porównania "!="

Parametry

rhs	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.9.3.24 operator=()

```
template<class T >
ListCircle<T>&  ListCircle< T >::operator= (
            const  ListCircle< T > & value )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.9.3.25 operator==()

```
template<class T >
bool  ListCircle< T >::operator== (
            const  ListCircle< T > & rhs ) const  [inline]
```

Operator przeciążony porównania "=="

Parametry

rhs	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.9.3.26 operator[]()

```
template<class T >
T  ListCircle< T >::operator[] (
            const int index )  [inline]
```

Operator przeciążony indeksowy "[ ]"

Parametry

index	Indeks elementu.

Zwraca

Wartość elementu spod podanego indeksu.

4.9.3.27 PopBack()

```
template<class T >
void ListCircle< T >::PopBack ( ) [inline], [virtual]
```

Usunięcie ostatniego elementu i zmniejszenie wielkości o jeden.

Implementuje **Sequences< T >** (str. 149).

4.9.3.28 PopFront()

```
template<class T >
void ListCircle< T >::PopFront ( ) [inline], [virtual]
```

Usunięcie pierwszego elementu i zmniejszenie wielkości o jeden.

Implementuje **Sequences< T >** (str. 149).

4.9.3.29 PushBack()

```
template<class T >
void ListCircle< T >::PushBack (
            const T & value ) [inline], [virtual]
```

Dodanie nowego elementu na końcu kontenera, po ostatnimm oraz zwiększenie wielkości o jeden.

Parametry

value	Nowy element.

Implementuje **Sequences< T >** (str. 149).

4.9.3.30 PushFront()

```
template<class T >
void ListCircle< T >::PushFront (
            const T & value ) [inline], [virtual]
```

Dodanie nowego elementu na początku kontenera, po ostatnimm oraz zwiększenie wielkości o jeden.

Parametry

value	Nowy element.

Implementuje **Sequences< T >** (str. 150).

4.9.3.31 Remove()

```
template<class T >
void  ListCircle< T >::Remove (
            const T & value )  [inline]
```

Usunięcie wybranego elementu ze zwolnieniem pamięci.

Parametry

value	Wartość elementu do usunięcia.

4.9.3.32 Resize() [1/2]

```
template<class T >
void  ListCircle< T >::Resize (
            int n )  [inline], [virtual]
```

Zmiana wielkości kontenera na podaną.

- Mniejsza niż aktualna: ucięcie kontenera.
- Większa niż aktualna: nowe elementy są puste.

Parametry

n	Nowa wielkość kontenera.

Implementuje **Sequences< T >** (str. 150).

4.9.3.33 Resize() [2/2]

```
template<class T >
void  ListCircle< T >::Resize (
```

```
int n,
const T & value )  [inline], [virtual]
```

Zmiana wielkości kontenera na podaną i dodanie nowych elementów.

- Mniejsza niż aktualna: ucięcie kontenera.
- Większa niż aktualna: nowe elementy mają podaną wartość.

Parametry

n	Nowa wielkość kontenera.
value	Wartość nowych elementów.

Implementuje **Sequences**< **T** > (str. 150).

4.9.3.34 returnEmpty()

```
template<class T >
T  ListCircle< T >::returnEmpty ( )  [inline], [virtual]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

Implementuje **Sequences**< **T** > (str. 151).

4.9.3.35 Reverse()

```
template<class T >
void  ListCircle< T >::Reverse ( )  [inline], [virtual], [noexcept]
```

Odwrócenie elementów kontenera kolejnością.

Implementuje **Sequences**< **T** > (str. 151).

4.9.3.36 Show()

```
template<class T >
void  ListCircle< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Sequences**< **T** > (str. 151).

4.9.3.37 showType()

```
template<class T >
string  ListCircle< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.9.3.38 Size()

```
template<class T >
int  ListCircle< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Sequences< T >** (str. 151).

4.9.3.39 Sort()

```
template<class T >
void  ListCircle< T >::Sort ( )  [inline], [virtual]
```

Sortowanie elementów w kontenerze

Implementuje **Sequences< T >** (str. 152).

4.9.3.40 Splice()

```
template<class T >
void  ListCircle< T >::Splice (
            int position,
             ListCircle< T > & x )  [inline]
```

Przepięcie elementów innego kontenera na odpowiednią pozycję.

Parametry

position	Pozycja na przepięcie.
x	Wskaźnik na inny kontener.

4.9.3.41 Swap()

```
template<class T >
void  ListCircle< T >::Swap (
              ListCircle< T > & l2 )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.9.4 Dokumentacja przyjaciół i funkcji związanych

4.9.4.1 operator<<

```
template<class T >
ostream& operator<< (
             ostream & ostr,
             const  ListCircle< T > & list )  [friend]
```

Operator przeciążony strumienia "<<"

Parametry

ostr	Lewostronny operand potrzebny funkcji do funkcjonowania.
list	Wskaźnik na inny kontener.

Zwraca

Zwrócenie Operandu.

4.9.5 Dokumentacja atrybutów składowych

4.9.5.1 pHead

```
template<class T >
ListBiNode<T>* ListCircle< T >::pHead [private]
```

Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **ListCircle.h**

4.10 Dokumentacja szablonu klasy ListNode< T >

```
#include <Nudes.h>
```

Metody publiczne

- **ListNode** (T value, **ListNode** * **pNext**)

Atrybuty publiczne

- T **data**
- **ListNode** * **pNext**

4.10.1 Opis szczegółowy

template<class T>
class ListNode< T >

Deklaracja klasy **ListNode** (str. 87)

- Klasa pomocnicza dla klasy **List** (str. 31).

Parametry

data	Wartość elementu.
pNext	Wskaźnik na nastêpny element.

4.10.2 Dokumentacja konstruktora i destruktora

4.10.2.1 ListNode()

```
template<class T >
ListNode< T >:: ListNode (
```

```
T value,
 ListNode< T > * pNext )  [inline]
```

Konstruktor dwuargumentowy

Parametry

value	Wartość do wpisania
pNext	Wskaźnik na następny element

4.10.3 Dokumentacja atrybutów składowych

4.10.3.1 data

```
template<class T >
T  ListNode< T >::data
```

4.10.3.2 pNext

```
template<class T >
ListNode*  ListNode< T >::pNext
```

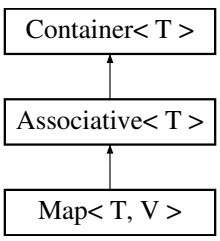
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Nudes.h**

4.11 Dokumentacja szablonu klasy Map< T, V >

```
#include <Map.h>
```

Diagram dziedziczenia dla Map< T, V >



Metody publiczne

- **Map** ()
- **Map** (T **key**, V value)
- **Map** (const **Map** &other)
- ~**Map** ()
- **Map**< T, V > & **operator=** (const **Map** &other)
- bool **operator==** (const **Map** &other) const
- bool **operator!=** (const **Map** &other) const
- void **PopBack** ()
- void **PopFront** ()
- V * **AtPtr** (T **key**)
- V * **End** () const noexcept
- V * **Rend** () noexcept
- V * **Begin** () const noexcept
- V * **Rbegin** () noexcept
- bool **usedKey** (const T & **key**)
- void **Insert** (const T & **key**, const V &value)
- void **Emplace** (T **key**, V value, int position)
- void **Swap** (**Map** &tmp)
- int **findElement** (const V value)
- bool **findIfExist** (const V value)
- bool **allowedType** ()
- T **returnEmptyKey** ()
- V **returnEmptyValue** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** () noexcept
- void **Erase** (int position)
- void **Erase** (int first, int last)
- int **MaxSize** ()
- void **Reverse** () noexcept
- void **Sort** ()
- void **Show** ()

Metody prywatne

- string **showTypeKey** ()
- string **showTypeValue** ()

Atrybuty prywatne

- T * **key**
- V * **data**
- int **size**

4.11.1 Opis szczegółowy

template<class T, class V>
class Map< T, V >

Deklaracja klasy **Map** (str. 88)

- Odpowiednik STLowego mapy/słownika.
- Dziedzicząca po **Associative** (str. 10).

Parametry

key	Dynamicznie zaalokowana tablica Tów.
data	Dynamicznie zaalokowana tablica Tów.
size	Ilość wpisanych elementów

4.11.2 Dokumentacja konstruktora i destruktora

4.11.2.1 Map() [1/3]

```
template<class T , class V >
Map< T, V >::  Map ( )  [inline]
```

Konstruktor domyślny

4.11.2.2 Map() [2/3]

```
template<class T , class V >
Map< T, V >::  Map (
             T key,
             V value )  [inline]
```

Konstruktor dwuoargumentowy.

- Dodaje podany klucz i wartość jako pierwszy element.

Parametry

key	Wartość do wpisania - klucz.
value	Wartość do wpisania.

4.11.2.3 Map() [3/3]

```
template<class T , class V >
Map< T, V >::  Map (
             const  Map< T, V > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.11.2.4 ~Map()

```
template<class T , class V >
Map< T, V >::~ Map ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.11.3 Dokumentacja funkcji składowych

4.11.3.1 allowedType()

```
template<class T , class V >
bool  Map< T, V >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Associative< T >** (str. 11).

4.11.3.2 AtPtr()

```
template<class T , class V >
V*  Map< T, V >::AtPtr (
            T key )  [inline]
```

Zwrócenie wskaźnika na wartość spod podanego klucza.

Parametry

key	Podany klucz.

Zwraca

Wskaźnik na dany element.

4.11.3.3 Begin()

```
template<class T , class V >
V*  Map< T, V >::Begin ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.11.3.4 Clear()

```
template<class T , class V >
void  Map< T, V >::Clear ( )  [inline], [virtual], [noexcept]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Associative< T >** (str. 11).

4.11.3.5 Emplace()

```
template<class T , class V >
void  Map< T, V >::Emplace (
            T key,
            V value,
            int position )  [inline]
```

Wstawienie nowego elementu na podaną pozycje.

Parametry

key	Nowy element - klucz.
value	Nowy element.
position	Pozycja, na którą metoda wstawia element.

4.11.3.6 Empty()

```
template<class T , class V >
bool  Map< T, V >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Associative< T >** (str. 12).

4.11.3.7 End()

```
template<class T , class V >
V*  Map< T, V >::End ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.11.3.8 Erase() [1/2]

```
template<class T , class V >
void  Map< T, V >::Erase (
            int first,
            int last )  [inline], [virtual]
```

Usunięcie wybranych elementów z przedziału.

Parametry

first	Pierwszy indeks przedziału.
last	Ostatni indeks przedziału.

Implementuje **Associative< T >** (str. 12).

4.11.3.9 Erase() [2/2]

```
template<class T , class V >
void  Map< T, V >::Erase (
            int position )  [inline], [virtual]
```

Usunięcie wybranego elementu po pozycji.

Parametry

position	Pozycja elementu do usunięcia.

Implementuje **Associative**< **T** > (str. 12).

4.11.3.10 findElement()

```
template<class T , class V >
int  Map< T, V >::findElement (
             const V value )  [inline]
```

Szukanie elementu po podanej wartości.

Parametry

value	Wartość elementu do odszukania.

Zwraca

Wartość odszukanego elementu.

4.11.3.11 findIfExist()

```
template<class T , class V >
bool  Map< T, V >::findIfExist (
             const V value )  [inline]
```

Sprawdzenie czy element istnieje, szukany po wartości.

Parametry

value	Szukana wartość.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

4.11.3.12 Insert()

```
template<class T , class V >
void  Map< T, V >::Insert (
            const T & key,
            const V & value )  [inline]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
key	Nowy element - klucz.
value	Nowy element.

4.11.3.13 MaxSize()

```
template<class T , class V >
int  Map< T, V >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Associative< T >** (str. 13).

4.11.3.14 operator"!=()

```
template<class T , class V >
bool  Map< T, V >::operator!= (
            const  Map< T, V > & other ) const  [inline]
```

Operator przeciążony porównania "!="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.11.3.15 operator=()

```
template<class T , class V >
Map<T, V>&  Map< T, V >::operator= (
             const  Map< T, V > & other )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.11.3.16 operator==()

```
template<class T , class V >
bool  Map< T, V >::operator== (
             const  Map< T, V > & other ) const  [inline]
```

Operator przeciążony porównania "=="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.11.3.17 PopBack()

```
template<class T , class V >
void  Map< T, V >::PopBack ( )  [inline], [virtual]
```

Usunięcie ostatniego elementu i zmniejszenie wielkości o jeden.

Implementuje **Associative< T >** (str. 13).

4.11.3.18 PopFront()

```
template<class T , class V >
void  Map< T, V >::PopFront ( )  [inline], [virtual]
```

Usunięcie pierwszego elementu i zmniejszenie wielkości o jeden.

Implementuje **Associative< T >** (str. 13).

4.11.3.19 Rbegin()

```
template<class T , class V >
V*  Map< T, V >::Rbegin ( )  [inline], [noexcept]
```

Zamienienie key i data oraz zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.11.3.20 Rend()

```
template<class T , class V >
V*  Map< T, V >::Rend ( )  [inline], [noexcept]
```

Zamienienie key i data oraz zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.11.3.21 returnEmptyKey()

```
template<class T , class V >
T  Map< T, V >::returnEmptyKey ( )  [inline]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

4.11.3.22 returnEmptyValue()

```
template<class T , class V >
V  Map< T, V >::returnEmptyValue ( )  [inline]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

4.11.3.23 Reverse()

```
template<class T , class V >
void  Map< T, V >::Reverse ( )  [inline], [virtual], [noexcept]
```

Odwrócenie elementów kontenera kolejnością.

Implementuje **Associative< T >** (str. 14).

4.11.3.24 Show()

```
template<class T , class V >
void  Map< T, V >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Associative< T >** (str. 14).

4.11.3.25 showTypeKey()

```
template<class T , class V >
string  Map< T, V >::showTypeKey ( )  [inline], [private]
```

4.11.3.26 showTypeValue()

```
template<class T , class V >
string  Map< T, V >::showTypeValue ( )  [inline], [private]
```

4.11.3.27 Size()

```
template<class T , class V >
int  Map< T, V >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Associative< T >** (str. 14).

4.11.3.28 Sort()

```
template<class T , class V >
void  Map< T, V >::Sort ( )  [inline], [virtual]
```

Sortowanie elementów w kontenerze

Implementuje **Associative< T >** (str. 14).

4.11.3.29 Swap()

```
template<class T , class V >
void  Map< T, V >::Swap (
              Map< T, V > & tmp )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.11.3.30 usedKey()

```
template<class T , class V >
bool  Map< T, V >::usedKey (
             const T & key )  [inline]
```

Sprawdzenie czy podany klucz jest unikatowy.

Parametry

key	Podany klucz do sprawdzenia.

Zwraca

Status operacji: true, jeżeli jest unikatowy; false jeśli nie.

4.11.4 Dokumentacja atrybutów składowych

4.11.4.1 data

```
template<class T , class V >
V* Map< T, V >::data [private]
```

4.11.4.2 key

```
template<class T , class V >
T* Map< T, V >::key [private]
```

4.11.4.3 size

```
template<class T , class V >
int Map< T, V >::size [private]
```

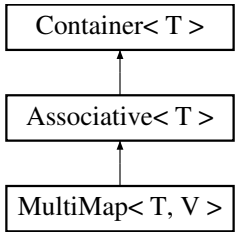
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Map.h**

4.12 Dokumentacja szablonu klasy MultiMap< T, V >

```
#include <MultiMap.h>
```

Diagram dziedziczenia dla MultiMap< T, V >



Metody publiczne

- **MultiMap** ()
- **MultiMap** (T **key**, V value)
- **MultiMap** (const **MultiMap** &other)
- ~**MultiMap** ()
- **MultiMap**< T, V > & **operator=** (const **MultiMap** &other)
- bool **operator==** (const **MultiMap** &other) const
- bool **operator!=** (const **MultiMap** &other) const
- void **PopBack** ()
- void **PopFront** ()
- V * **AtPtr** (T **key**)
- V * **End** () const noexcept
- V * **Rend** () noexcept
- V * **Begin** () const noexcept
- V * **Rbegin** () noexcept
- bool **usedPair** (const T & **key**, const V &value)
- void **Insert** (const T & **key**, const V &value)
- void **Emplace** (T **key**, V value, int position)
- void **Swap** (**MultiMap** &tmp)
- int **findElement** (const V value)
- bool **findIfExist** (const V value)
- bool **allowedType** ()
- T **returnEmptyKey** ()
- V **returnEmptyValue** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** () noexcept
- void **Erase** (int position)
- void **Erase** (int first, int last)
- int **MaxSize** ()
- void **Reverse** () noexcept
- void **Sort** ()
- void **Show** ()

Metody prywatne

- string **showTypeKey** ()
- string **showTypeValue** ()

Atrybuty prywatne

- T * **key**
- V * **data**
- int **size**

4.12.1 Opis szczegółowy

template<class T, class V>
class MultiMap< T, V >

Deklaracja klasy **MultiMap** (str. 100)

- Odpowiednik STLowego MultiMapy/słownika.
- Dziedzicząca po **Associative** (str. 10).

Parametry

key	Dynamicznie zaalokowana tablica Tów.
data	Dynamicznie zaalokowana tablica Tów.
size	Ilość wpisanych elementów

4.12.2 Dokumentacja konstruktora i destruktora

4.12.2.1 MultiMap() [1/3]

```
template<class T , class V >
MultiMap< T, V >::  MultiMap ( )  [inline]
```

Konstruktor domyślny

4.12.2.2 MultiMap() [2/3]

```
template<class T , class V >
MultiMap< T, V >::  MultiMap (
            T key,
            V value )  [inline]
```

Konstruktor dwuoargumentowy.

- Dodaje podany klucz i wartość jako pierwszy element.

Parametry

key	Wartość do wpisania - klucz.
value	Wartość do wpisania.

4.12.2.3 MultiMap() [3/3]

```
template<class T , class V >
MultiMap< T, V >::  MultiMap (
            const  MultiMap< T, V > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.12.2.4 ~MultiMap()

```
template<class T , class V >
MultiMap< T, V >::~ MultiMap ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.12.3 Dokumentacja funkcji składowych

4.12.3.1 allowedType()

```
template<class T , class V >
bool  MultiMap< T, V >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Associative< T >** (str. 11).

4.12.3.2 AtPtr()

```
template<class T , class V >
V*  MultiMap< T, V >::AtPtr (
            T key )  [inline]
```

Zwrócenie wskaźnika na wartość spod podanego klucza.

Parametry

key	Podany klucz.

Zwraca

Wskaźnik na dany element.

4.12.3.3 Begin()

```
template<class T , class V >
V*  MultiMap< T, V >::Begin ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.12.3.4 Clear()

```
template<class T , class V >
void  MultiMap< T, V >::Clear ( )  [inline], [virtual], [noexcept]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Associative< T >** (str. 11).

4.12.3.5 Emplace()

```
template<class T , class V >
void  MultiMap< T, V >::Emplace (
            T key,
            V value,
            int position )  [inline]
```

Wstawienie nowego elementu na podaną pozycje.

Parametry

key	Nowy element - klucz.
value	Nowy element.
position	Pozycja, na którą metoda wstawia element.

4.12.3.6 Empty()

```
template<class T , class V >
bool  MultiMap< T, V >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Associative< T >** (str. 12).

4.12.3.7 End()

```
template<class T , class V >
V*  MultiMap< T, V >::End ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.12.3.8 Erase() [1/2]

```
template<class T , class V >
void  MultiMap< T, V >::Erase (
            int first,
            int last )  [inline], [virtual]
```

Usunięcie wybranych elementów z przedziału.

Parametry

first	Pierwszy indeks przedziału.
last	Ostatni indeks przedziału.

Implementuje **Associative< T >** (str. 12).

4.12.3.9 Erase() [2/2]

```
template<class T , class V >
void  MultiMap< T, V >::Erase (
            int position )  [inline], [virtual]
```

Usunięcie wybranego elementu po pozycji.

Parametry

position	Pozycja elementu do usunięcia.

Implementuje **Associative**< **T** > (str. 12).

4.12.3.10 findElement()

```
template<class T , class V >
int  MultiMap< T, V >::findElement (
             const V value )  [inline]
```

Szukanie elementu po podanej wartości.

Parametry

value	Wartość elementu do odszukania.

Zwraca

Wartość odszukanego elementu.

4.12.3.11 findIfExist()

```
template<class T , class V >
bool  MultiMap< T, V >::findIfExist (
             const V value )  [inline]
```

Sprawdzenie czy element istnieje, szukany po wartości.

Parametry

value	Szukana wartość.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

4.12.3.12 Insert()

```
template<class T , class V >
void  MultiMap< T, V >::Insert (
            const T & key,
            const V & value )  [inline]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

key	Nowy element - klucz.
value	Nowy element.

4.12.3.13 MaxSize()

```
template<class T , class V >
int  MultiMap< T, V >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Associative< T >** (str. 13).

4.12.3.14 operator"!=()

```
template<class T , class V >
bool  MultiMap< T, V >::operator!= (
            const  MultiMap< T, V > & other ) const  [inline]
```

Operator przeciążony porównania "!="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.12.3.15 operator=()

```
template<class T , class V >
MultiMap<T, V>&  MultiMap< T, V >::operator= (
             const  MultiMap< T, V > & other )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.12.3.16 operator==()

```
template<class T , class V >
bool  MultiMap< T, V >::operator== (
             const  MultiMap< T, V > & other ) const  [inline]
```

Operator przeciążony porównania "=="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.12.3.17 PopBack()

```
template<class T , class V >
void  MultiMap< T, V >::PopBack ( )  [inline], [virtual]
```

Usunięcie ostatniego elementu i zmniejszenie wielkości o jeden.

Implementuje **Associative< T >** (str. 13).

4.12.3.18 PopFront()

```
template<class T , class V >
void  MultiMap< T, V >::PopFront ( )  [inline], [virtual]
```

Usunięcie pierwszego elementu i zmniejszenie wielkości o jeden.

Implementuje **Associative< T >** (str. 13).

4.12.3.19 Rbegin()

```
template<class T , class V >
V*  MultiMap< T, V >::Rbegin ( )  [inline], [noexcept]
```

Zamienienie key i data oraz zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.12.3.20 Rend()

```
template<class T , class V >
V*  MultiMap< T, V >::Rend ( )  [inline], [noexcept]
```

Zamienienie key i data oraz zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.12.3.21 returnEmptyKey()

```
template<class T , class V >
T  MultiMap< T, V >::returnEmptyKey ( )  [inline]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

4.12.3.22 returnEmptyValue()

```
template<class T , class V >
V MultiMap< T, V >::returnEmptyValue ( ) [inline]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

4.12.3.23 Reverse()

```
template<class T , class V >
void MultiMap< T, V >::Reverse ( ) [inline], [virtual], [noexcept]
```

Odwrócenie elementów kontenera kolejnością.

Implementuje **Associative< T >** (str. 14).

4.12.3.24 Show()

```
template<class T , class V >
void MultiMap< T, V >::Show ( ) [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Associative< T >** (str. 14).

4.12.3.25 showTypeKey()

```
template<class T , class V >
string MultiMap< T, V >::showTypeKey ( ) [inline], [private]
```

4.12.3.26 showTypeValue()

```
template<class T , class V >
string MultiMap< T, V >::showTypeValue ( ) [inline], [private]
```

4.12.3.27 Size()

```
template<class T , class V >
int  MultiMap< T, V >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Associative< T >** (str. 14).

4.12.3.28 Sort()

```
template<class T , class V >
void  MultiMap< T, V >::Sort ( )  [inline], [virtual]
```

Sortowanie elementów w kontenerze

Implementuje **Associative< T >** (str. 14).

4.12.3.29 Swap()

```
template<class T , class V >
void  MultiMap< T, V >::Swap (
              MultiMap< T, V > & tmp )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.12.3.30 usedPair()

```
template<class T , class V >
bool  MultiMap< T, V >::usedPair (
             const T & key,
             const V & value )  [inline]
```

Sprawdzenie czy podana para jest unikatowa

Parametry

key	Podany klucz pary.
value	Wartość spod podanego klucza.

Zwraca

Status operacji: true - para jest unikalna, false - ukorzystaną już taką parę.

4.12.4 Dokumentacja atrybutów składowych

4.12.4.1 data

```
template<class T , class V >
V*  MultiMap< T, V >::data  [private]
```

4.12.4.2 key

```
template<class T , class V >
T*  MultiMap< T, V >::key  [private]
```

4.12.4.3 size

```
template<class T , class V >
int  MultiMap< T, V >::size  [private]
```

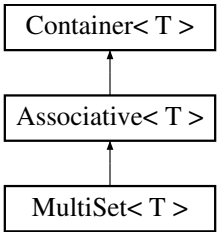
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **MultiMap.h**

4.13 Dokumentacja szablonu klasy MultiSet< T >

```
#include <MultiSet.h>
```

Diagram dziedziczenia dla MultiSet< T >



Metody publiczne

- **MultiSet** ()
- **MultiSet** (T **key**)
- **MultiSet** (T **key**, int count)
- **MultiSet** (const **MultiSet** &other)
- ~**MultiSet** ()
- **MultiSet**< T > & **operator=** (const **MultiSet** &other)
- bool **operator==** (const **MultiSet** &other) const
- bool **operator!=** (const **MultiSet** &other) const
- void **PopBack** ()
- void **PopFront** ()
- T * **AtPtr** (T **key**)
- T * **End** () const noexcept
- T * **Rend** () noexcept
- T * **Begin** () const noexcept
- T * **Rbegin** () noexcept
- int **Index** (T value)
- void **Insert** (const T &data)
- void **Insert** (int position, const T &data)
- void **Insert** (int position, const T &data, int count)
- void **Emplace** (T **key**, int position)
- void **Swap** (**MultiSet** &tmp)
- bool **findIfExist** (const T value)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** () noexcept
- void **Erase** (int position)
- void **Erase** (int first, int last)
- int **MaxSize** ()
- void **Remove** (T value)
- int **keyCount** (T value)
- void **Reverse** () noexcept
- void **Sort** ()
- void **Show** ()

Metody prywatne

- string **showType** ()
- void **insertionsort** (T *tab, int count)

Atrybuty prywatne

- T * **key**
- int **size**

4.13.1 Opis szczegółowy

template<**class T**>
class MultiSet< T >

Deklaracja klasy **MultiSet** (str. 112)

- Odpowiednik STLowego MutliSeta/multizbioru.
- Dziedzicząca po **Associative** (str. 10).

Parametry

key	Dynamicznie zaalokowana tablica Tów.
size	Ilość wpisanych elementów

4.13.2 Dokumentacja konstruktora i destruktora

4.13.2.1 MultiSet() [1/4]

```
template<class T >
MultiSet< T >::  MultiSet ( )  [inline]
```

Konstruktor domyślny

4.13.2.2 MultiSet() [2/4]

```
template<class T >
MultiSet< T >::  MultiSet (
            T key )  [inline]
```

Konstruktor jednoargumentowy.

- Dodaje podany klucz jako pierwszy element.

Parametry

key	Wartość do wpisania - klucz.

4.13.2.3 MultiSet() [3/4]

```
template<class T >
MultiSet< T >::  MultiSet (
            T key,
            int count )  [inline]
```

Konstruktor dwuoargumentowy.

- Dodaje podany klucz jako pierwszy element.

Parametry

key	Wartość do wpisania - klucz.

4.13.2.4 MultiSet() [4/4]

```
template<class T >
MultiSet< T >::  MultiSet (
             const  MultiSet< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.13.2.5 ~MultiSet()

```
template<class T >
MultiSet< T >::~ MultiSet ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.13.3 Dokumentacja funkcji składowych

4.13.3.1 allowedType()

```
template<class T >
bool  MultiSet< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Associative< T >** (str. 11).

4.13.3.2 AtPtr()

```
template<class T >
T*  MultiSet< T >::AtPtr (
            T key )  [inline]
```

Zwrócenie wskaźnika na wartość spod podanego klucza.

Parametry

key	Podany klucz.

Zwraca

Wskaźnik na dany element.

4.13.3.3 Begin()

```
template<class T >
T*  MultiSet< T >::Begin ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.13.3.4 Clear()

```
template<class T >
void  MultiSet< T >::Clear ( )  [inline], [virtual], [noexcept]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Associative< T >** (str. 11).

4.13.3.5 Emplace()

```
template<class T >
void  MultiSet< T >::Emplace (
            T key,
            int position )  [inline]
```

Wstawienie nowego elementu na podaną pozycje.

Parametry

key	Nowy element - klucz.
position	Pozycja, na którą metoda wstawia element.

4.13.3.6 Empty()

```
template<class T >
bool  MultiSet< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Associative< T >** (str. 12).

4.13.3.7 End()

```
template<class T >
T*  MultiSet< T >::End ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.13.3.8 Erase() [1/2]

```
template<class T >
void  MultiSet< T >::Erase (
            int first,
            int last )  [inline], [virtual]
```

Usunięcie wybranych elementów z przedziału.

Parametry

first	Pierwszy indeks przedziału.
last	Ostatni indeks przedziału.

Implementuje **Associative**< **T** > (str. 12).

4.13.3.9 Erase() [2/2]

```
template<class T >
void  MultiSet< T >::Erase (
            int position )  [inline], [virtual]
```

Usunięcie wybranego elementu po pozycji.

Parametry

position	Pozycja elementu do usunięcia.

Implementuje **Associative**< **T** > (str. 12).

4.13.3.10 findIfExist()

```
template<class T >
bool  MultiSet< T >::findIfExist (
            const T value )  [inline]
```

Sprawdzenie czy element istnieje, szukany po wartości.

Parametry

value	Szukana wartość.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

4.13.3.11 Index()

```
template<class T >
int  MultiSet< T >::Index (
            T value )  [inline]
```

Zwraca index podanę wartości o ile jest w kontenerze.

Parametry

value	Wartość do odnalezienia.

Zwraca

Indeks wartości o ile znajduje się w kontenerze, jeśli nie to zwracana jest wartość -1.

4.13.3.12 Insert() [1/3]

```
template<class T >
void  MultiSet< T >::Insert (
            const T & data )  [inline]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

data	Nowy element - klucz.

4.13.3.13 Insert() [2/3]

```
template<class T >
void  MultiSet< T >::Insert (
            int position,
            const T & data )  [inline]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
key	Nowy element - klucz.
value	Nowy element.

4.13.3.14 Insert() [3/3]

```
template<class T >
void  MultiSet< T >::Insert (
            int position,
            const T & data,
            int count )  [inline]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
key	Nowy element - klucz.
value	Nowy element.

4.13.3.15 insertionsort()

```
template<class T >
void  MultiSet< T >::insertionsort (
            T * tab,
            int count )  [inline], [private]
```

Funkcja wykorzystywana w **Sort()** (str. 124)

Parametry

tab	Wskaźnik na tablicę elementów kontenera.
count	Ilość elementów w tablicy.

4.13.3.16 keyCount()

```
template<class T >
int  MultiSet< T >::keyCount (
            T value )  [inline]
```

Zliczenie ilości wystąpienień klucza w kontenerze.

Parametry

value	Wartość klucza do zliczeia.

Zwraca

Status operacji.

4.13.3.17 MaxSize()

```
template<class T >
int  MultiSet< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Associative**< **T** > (str. 13).

4.13.3.18 operator"!=()

```
template<class T >
bool  MultiSet< T >::operator!= (
            const  MultiSet< T > & other ) const  [inline]
```

Operator przeciążony porównania "!="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.13.3.19 operator=()

```
template<class T >
MultiSet<T>&  MultiSet< T >::operator= (
            const  MultiSet< T > & other )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.13.3.20 operator==()

```
template<class T >
bool  MultiSet< T >::operator== (
            const  MultiSet< T > & other ) const  [inline]
```

Operator przeciążony porównania "=="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.13.3.21 PopBack()

```
template<class T >
void  MultiSet< T >::PopBack ( )  [inline], [virtual]
```

Usunięcie ostatniego elementu i zmniejszenie wielkości o jeden.

Implementuje **Associative< T >** (str. 13).

4.13.3.22 PopFront()

```
template<class T >
void  MultiSet< T >::PopFront ( )  [inline], [virtual]
```

Usunięcie pierwszego elementu i zmniejszenie wielkości o jeden.

Implementuje **Associative< T >** (str. 13).

4.13.3.23 Rbegin()

```
template<class T >
T*  MultiSet< T >::Rbegin ( )  [inline], [noexcept]
```

Zamienienie key i data oraz zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.13.3.24 Remove()

```
template<class T >
void  MultiSet< T >::Remove (
            T value )  [inline]
```

Usunięcie wybranego elementu po wartości.

Parametry

value	Wartość elementu do usunięcia.

4.13.3.25 Rend()

```
template<class T >
T*  MultiSet< T >::Rend ( )  [inline], [noexcept]
```

Zamienienie key oraz zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.13.3.26 returnEmpty()

```
template<class T >
T  MultiSet< T >::returnEmpty ( )  [inline]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

4.13.3.27 Reverse()

```
template<class T >
void  MultiSet< T >::Reverse ( )  [inline], [virtual], [noexcept]
```

Odwrócenie elementów kontenera kolejnością.

Implementuje **Associative< T >** (str. 14).

4.13.3.28 Show()

```
template<class T >
void  MultiSet< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Associative< T >** (str. 14).

4.13.3.29 showType()

```
template<class T >
string  MultiSet< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.13.3.30 Size()

```
template<class T >
int  MultiSet< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Associative< T >** (str. 14).

4.13.3.31 Sort()

```
template<class T >
void  MultiSet< T >::Sort ( )  [inline], [virtual]
```

Sortowanie elementów w kontenerze

Implementuje **Associative< T >** (str. 14).

4.13.3.32 Swap()

```
template<class T >
void  MultiSet< T >::Swap (
              MultiSet< T > & tmp )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.13.4 Dokumentacja atrybutów składowych

4.13.4.1 key

```
template<class T >
T*  MultiSet< T >::key  [private]
```

4.13.4.2 size

```
template<class T >
int  MultiSet< T >::size  [private]
```

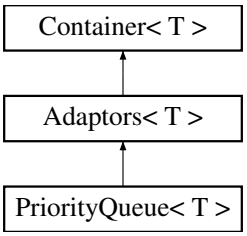
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **MultiSet.h**

4.14 Dokumentacja szablonu klasy PriorityQueue< T >

```
#include <PriorityQueue.h>
```

Diagram dziedziczenia dla PriorityQueue< T >



Metody publiczne

- **PriorityQueue** ()
- **PriorityQueue** (T value)
- **PriorityQueue** (T value, int **size**)
- **PriorityQueue** (const **PriorityQueue** &other)
- ~**PriorityQueue** ()
- **PriorityQueue**< T > & **operator=** (const **PriorityQueue** &value)
- bool **operator==** (const **PriorityQueue** &value) const
- bool **operator!=** (const **PriorityQueue** &value) const
- void **Push** (const T &value)
- void **Pop** ()
- T & **Back** ()
- T * **BackPtr** ()

- T & **Front** ()
- T * **FrontPtr** ()
- void **Swap** (**PriorityQueue** &tmp)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** () noexcept
- int **MaxSize** ()
- void **findMax** ()
- void **Show** ()

Metody prywatne

- string **showType** ()

Atrybuty prywatne

- T * **data**
- int **size**
- int **max**

Statyczne atrybuty prywatne

- static const int **front** = 0

4.14.1 Opis szczegółowy

template<class T>
class PriorityQueue< T >

Deklaracja klasy **PriorityQueue** (str. 125)

- Odpowiednik STLowego kolejki.
- Dziedzicząca po **Adaptors** (str. 7).

Parametry

data	Dynamicznie zaalokowana tablica Tów.
size	Ilość wpisanych elementów
front	Stała opisująca indeks TOPu.

4.14.2 Dokumentacja konstruktora i destruktora

4.14.2.1 PriorityQueue() [1/4]

```
template<class T >
PriorityQueue< T >::  PriorityQueue ( )  [inline]
```

Konstruktor domyślny

4.14.2.2 PriorityQueue() [2/4]

```
template<class T >
PriorityQueue< T >::  PriorityQueue (
            T value )  [inline]
```

Konstruktor jednoargumentowy.

- Dodaje podaną wartość jako pierwszy element.

Parametry

value	Wartość do wpisania.

4.14.2.3 PriorityQueue() [3/4]

```
template<class T >
PriorityQueue< T >::  PriorityQueue (
            T value,
            int size )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o podanej wartości.

Parametry

value	Wartość do wpisania.
size	Ilość elementów do stworzenia.

4.14.2.4 PriorityQueue() [4/4]

```
template<class T >
PriorityQueue< T >::  PriorityQueue (
            const  PriorityQueue< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.14.2.5 ∼PriorityQueue()

```
template<class T >
PriorityQueue< T >::~ PriorityQueue ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.14.3 Dokumentacja funkcji składowych

4.14.3.1 allowedType()

```
template<class T >
bool  PriorityQueue< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Adaptors< T >** (str. 8).

4.14.3.2 Back()

```
template<class T >
T&  PriorityQueue< T >::Back ( )  [inline]
```

Zwrócenie referencji na BACK.

Zwraca

Referencja na BACK.

4.14.3.3 BackPtr()

```
template<class T >
T*  PriorityQueue< T >::BackPtr ( )  [inline]
```

Zwrócenie wskaźnika na BACK.

Zwraca

Wskaźnik na BACK.

4.14.3.4 Clear()

```
template<class T >
void  PriorityQueue< T >::Clear ( )  [inline], [virtual], [noexcept]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0.

Implementuje **Adaptors< T >** (str. 8).

4.14.3.5 Empty()

```
template<class T >
bool  PriorityQueue< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Adaptors< T >** (str. 8).

4.14.3.6 findMax()

```
template<class T >
void  PriorityQueue< T >::findMax ( )  [inline]
```

Odnalezienie nowego Maxa i przypisanie go do zmiennej.

4.14.3.7 Front()

```
template<class T >
T&  PriorityQueue< T >::Front ( )  [inline]
```

Zwrócenie referencji na FRONT.

Zwraca

Referencja na FRONT.

4.14.3.8 FrontPtr()

```
template<class T >
T*  PriorityQueue< T >::FrontPtr ( )  [inline]
```

Zwrócenie wskaźnika na FRONT.

Zwraca

Wskaźnik na FRONT.

4.14.3.9 MaxSize()

```
template<class T >
int  PriorityQueue< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener.

Zwraca

Maksymalna wielkość kontenera.

Implementuje **Adaptors< T >** (str. 8).

4.14.3.10 operator"!=()

```
template<class T >
bool  PriorityQueue< T >::operator!= (
            const  PriorityQueue< T > & value ) const  [inline]
```

Operator przeciążony porównania "!="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.14.3.11 operator=()

```
template<class T >
PriorityQueue<T>&  PriorityQueue< T >::operator= (
             const  PriorityQueue< T > & value )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.14.3.12 operator==()

```
template<class T >
bool  PriorityQueue< T >::operator== (
             const  PriorityQueue< T > & value ) const  [inline]
```

Operator przeciążony porównania "=="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.14.3.13 Pop()

```
template<class T >
void  PriorityQueue< T >::Pop ( )  [inline], [virtual]
```

Usunięcie elementu Max i zmniejszenie jego wielkości.

Implementuje **Adaptors< T >** (str. 9).

4.14.3.14 Push()

```
template<class T >
void  PriorityQueue< T >::Push (
            const T & value )  [inline], [virtual]
```

Dodanie nowego elementu na TOP kontenera i zwiększenie jego wielkości.

Parametry

value	Nowy element.

Implementuje **Adaptors< T >** (str. 9).

4.14.3.15 returnEmpty()

```
template<class T >
T  PriorityQueue< T >::returnEmpty ( )  [inline], [virtual]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

Implementuje **Adaptors< T >** (str. 9).

4.14.3.16 Show()

```
template<class T >
void  PriorityQueue< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Adaptors< T >** (str. 10).

4.14.3.17 showType()

```
template<class T >
string  PriorityQueue< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.14.3.18 Size()

```
template<class T >
int  PriorityQueue< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Adaptors< T >** (str. 10).

4.14.3.19 Swap()

```
template<class T >
void  PriorityQueue< T >::Swap (
            PriorityQueue< T > & tmp )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.14.4 Dokumentacja atrybutów składowych

4.14.4.1 data

```
template<class T >
T*  PriorityQueue< T >::data  [private]
```

4.14.4.2 front

```
template<class T >
const int  PriorityQueue< T >::front = 0  [static], [private]
```

4.14.4.3 max

```
template<class T >
int  PriorityQueue< T >::max  [private]
```

4.14.4.4 size

```
template<class T >
int  PriorityQueue< T >::size  [private]
```

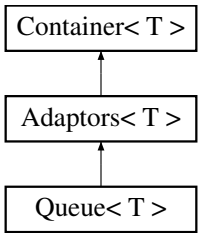
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **PriorityQueue.h**

4.15 Dokumentacja szablonu klasy Queue< T >

```
#include <Queue.h>
```

Diagram dziedziczenia dla Queue< T >



Metody publiczne

- **Queue** ()
- **Queue** (T value)
- **Queue** (T value, int **size**)
- **Queue** (const **Queue** &other)
- ~**Queue** ()
- **Queue**< T > & **operator=** (const **Queue** &value)
- bool **operator==** (const **Queue** &value) const
- bool **operator!=** (const **Queue** &value) const
- void **Push** (const T &value)
- void **Pop** ()
- T & **Back** ()
- T * **BackPtr** ()
- T & **Front** ()
- T * **FrontPtr** ()
- void **Swap** (**Queue** &tmp)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** ()
- int **MaxSize** ()
- void **Show** ()

Metody prywatne

- string **showType** ()

Atrybuty prywatne

- T * **data**
- int **size**

Statyczne atrybuty prywatne

- static const int **front** = 0

4.15.1 Opis szczegółowy

template<**class T**>
class Queue< **T** >

Deklaracja klasy **Queue** (str. 134)

- Odpowiednik STLowego kolejki.
- Dziedzicząca po **Adaptors** (str. 7).

Parametry

data	Dynamicznie zaalokowana tablica Tów.
size	Ilość wpisanych elementów
front	Stała opisująca indeks TOPu.

4.15.2 Dokumentacja konstruktora i destruktora

4.15.2.1 Queue() [1/4]

```
template<class T >
Queue< T >::  Queue ( )  [inline]
```

Konstruktor domyślny

4.15.2.2 Queue() [2/4]

```
template<class T >
Queue< T >::  Queue (
              T value )  [inline]
```

Konstruktor jednoargumentowy.

- Dodaje podaną wartość jako pierwszy element.

Parametry

value	Wartość do wpisania.

4.15.2.3 Queue() [3/4]

```
template<class T >
Queue< T >::  Queue (
              T value,
              int size )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o podanej wartości.

Parametry

value	Wartość do wpisania.
size	Ilość elementów do stworzenia.

4.15.2.4 Queue() [4/4]

```
template<class T >
Queue< T >::  Queue (
             const  Queue< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.15.2.5 ~Queue()

```
template<class T >
Queue< T >::~ Queue ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.15.3 Dokumentacja funkcji składowych

4.15.3.1 allowedType()

```
template<class T >
bool  Queue< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Adaptors< T >** (str. 8).

4.15.3.2 Back()

```
template<class T >
T&  Queue< T >::Back ( )  [inline]
```

Zwrócenie referencji na BACK.

Zwraca

Referencja na BACK.

4.15.3.3 BackPtr()

```
template<class T >
T*  Queue< T >::BackPtr ( )  [inline]
```

Zwrócenie wskaźnika na BACK.

Zwraca

Wskaźnik na BACK.

4.15.3.4 Clear()

```
template<class T >
void  Queue< T >::Clear ( )  [inline], [virtual]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0.

Implementuje **Adaptors**< **T** > (str. 8).

4.15.3.5 Empty()

```
template<class T >
bool  Queue< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Adaptors**< **T** > (str. 8).

4.15.3.6 Front()

```
template<class T >
T&  Queue< T >::Front ( )  [inline]
```

Zwrócenie referencji na FRONT.

Zwraca

Referencja na FRONT.

4.15.3.7 FrontPtr()

```
template<class T >
T*  Queue< T >::FrontPtr ( )  [inline]
```

Zwrócenie wskaźnika na FRONT.

Zwraca

Wskaźnik na FRONT.

4.15.3.8 MaxSize()

```
template<class T >
int  Queue< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener.

Zwraca

Maksymalna wielkość kontenera.

Implementuje **Adaptors< T >** (str. 8).

4.15.3.9 operator"!=()

```
template<class T >
bool  Queue< T >::operator!= (
            const  Queue< T > & value ) const  [inline]
```

Operator przeciążony porównania "!="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.15.3.10 operator=()

```
template<class T >
Queue<T>&  Queue< T >::operator= (
             const  Queue< T > & value )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.15.3.11 operator==()

```
template<class T >
bool  Queue< T >::operator== (
             const  Queue< T > & value ) const  [inline]
```

Operator przeciążony porównania "=="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.15.3.12 Pop()

```
template<class T >
void  Queue< T >::Pop ( )  [inline], [virtual]
```

Usunięcie elementu z FRONTu i zmniejszenie jego wielkości.

Implementuje **Adaptors< T >** (str. 9).

4.15.3.13 Push()

```
template<class T >
void  Queue< T >::Push (
            const T & value )  [inline], [virtual]
```

Dodanie nowego elementu na koniec kontenera i zwiększenie jego wielkości.

Parametry

value	Nowy element.

Implementuje **Adaptors< T >** (str. 9).

4.15.3.14 returnEmpty()

```
template<class T >
T  Queue< T >::returnEmpty ( )  [inline], [virtual]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

Implementuje **Adaptors< T >** (str. 9).

4.15.3.15 Show()

```
template<class T >
void  Queue< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Adaptors< T >** (str. 10).

4.15.3.16 showType()

```
template<class T >
string  Queue< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.15.3.17 Size()

```
template<class T >
int  Queue< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Adaptors< T >** (str. 10).

4.15.3.18 Swap()

```
template<class T >
void  Queue< T >::Swap (
              Queue< T > & tmp )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.15.4 Dokumentacja atrybutów składowych

4.15.4.1 data

```
template<class T >
T*  Queue< T >::data  [private]
```

4.15.4.2 front

```
template<class T >
const int  Queue< T >::front = 0  [static], [private]
```

4.15.4.3 size

```
template<class T >
int  Queue< T >::size  [private]
```

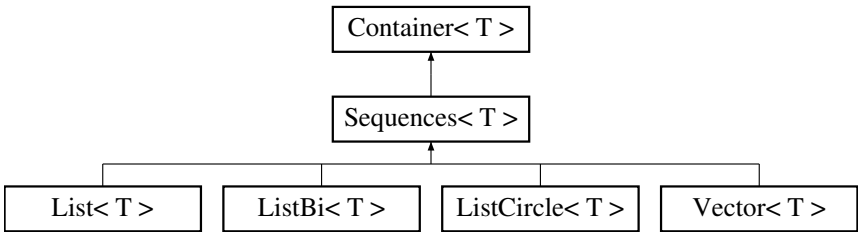
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Queue.h**

4.16 Dokumentacja szablonu klasy Sequences< T >

`#include <Sequences.h>`

Diagram dziedziczenia dla Sequences< T >



Metody publiczne

- virtual void **PopBack** ()=0
- virtual void **PopFront** ()=0
- virtual void **PushBack** (const T &value)=0
- virtual void **PushFront** (const T &value)=0
- virtual void **Insert** (int position, const T &value)=0
- virtual void **Insert** (int position, int n, const T &value)=0
- virtual void **Insert** (int position, int n, T *valuePtr)=0
- virtual void **Emplace** (T value, int position)=0
- virtual void **Assign** (int count, T *valuePtr)=0
- virtual void **Assign** (int count, const T value)=0
- virtual bool **findIfExist** (const T value)=0
- virtual bool **allowedType** ()=0
- virtual T **returnEmpty** ()=0
- virtual bool **Empty** () const noexcept=0
- virtual int **Size** () const noexcept=0
- virtual void **Clear** ()=0
- virtual void **Erase** (int position)=0
- virtual void **Erase** (int first, int last)=0
- virtual int **MaxSize** ()=0
- virtual void **Resize** (int n)=0
- virtual void **Resize** (int n, const T &value)=0
- virtual void **Sort** ()=0
- void **insertionsort** (T *tab, int count)
- virtual void **Reverse** () noexcept=0
- virtual void **Show** ()=0

4.16.1 Opis szczegółowy

template<class T>
class Sequences< T >

Deklaracja klasy **Sequences** (str. 143).

- Podstawa wszystkich klas opisujących sekwencyjne struktury danych.
- Dziedzicząca po **Container** (str. 15).

4.16.2 Dokumentacja funkcji składowych

4.16.2.1 allowedType()

```
template<class T >
virtual bool  Sequences< T >::allowedType ( )  [pure virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Container< T >** (str. 16).

Implementowany w **Vector< T >** (str. 194), **ListCircle< T >** (str. 73), **ListBi< T >** (str. 53) i **List< T >** (str. 35).

4.16.2.2 Assign() [1/2]

```
template<class T >
virtual void  Sequences< T >::Assign (
            int count,
            const T value )  [pure virtual]
```

Zastąpienie starych elementów podaną wartością.

Parametry

count	Ilość elementów do zastąpienia.
value	Nowa wartość.

Implementowany w **List< T >** (str. 36), **Vector< T >** (str. 194), **ListCircle< T >** (str. 73) i **ListBi< T >** (str. 54).

4.16.2.3 Assign() [2/2]

```
template<class T >
virtual void  Sequences< T >::Assign (
            int count,
            T * valuePtr )  [pure virtual]
```

Zastąpienie starych elementów podanymi ze wskaźnika/tablicy.

Parametry

count	Ilość elementów do zastąpienia.
valuePtr	Pointer na nowe elementy.

Implementowany w **Vector**< **T** > (str. 194), **ListCircle**< **T** > (str. 74), **ListBi**< **T** > (str. 54) i **List**< **T** > (str. 36).

4.16.2.4 Clear()

```
template<class T >
virtual void  Sequences< T >::Clear ( )  [pure virtual]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Container**< **T** > (str. 16).

Implementowany w **ListCircle**< **T** > (str. 75), **ListBi**< **T** > (str. 55), **Vector**< **T** > (str. 196) i **List**< **T** > (str. 37).

4.16.2.5 Emplace()

```
template<class T >
virtual void  Sequences< T >::Emplace (
            T value,
            int position )  [pure virtual]
```

Wstawienie nowego elementu na podaną pozycje.

Parametry

value	Nowy element.
position	Pozycja, na którą metoda wstawia element.

Implementowany w **Vector**< **T** > (str. 196), **ListCircle**< **T** > (str. 75), **ListBi**< **T** > (str. 55) i **List**< **T** > (str. 37).

4.16.2.6 Empty()

```
template<class T >
virtual bool  Sequences< T >::Empty ( ) const  [pure virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Container**< **T** > (str. 16).

Implementowany w **Vector**< **T** > (str. 197), **ListCircle**< **T** > (str. 75), **ListBi**< **T** > (str. 56) i **List**< **T** > (str. 38).

4.16.2.7 Erase() [1/2]

```
template<class T >
virtual void  Sequences< T >::Erase (
            int first,
            int last )  [pure virtual]
```

Usunięcie wybranych elementów z przedziału.

Parametry

first	Pierwszy indeks przedziału.
last	Ostatni indeks przedziału.

Implementowany w **Vector**< **T** > (str. 197), **ListCircle**< **T** > (str. 76), **ListBi**< **T** > (str. 56) i **List**< **T** > (str. 38).

4.16.2.8 Erase() [2/2]

```
template<class T >
virtual void  Sequences< T >::Erase (
            int position )  [pure virtual]
```

Usunięcie wybranego elementu po pozycji.

Parametry

position	Pozycja elementu do usunięcia.

Implementowany w **Vector**< **T** > (str. 198), **ListCircle**< **T** > (str. 76), **ListBi**< **T** > (str. 57) i **List**< **T** > (str. 39).

4.16.2.9 findIfExist()

```
template<class T >
virtual bool  Sequences< T >::findIfExist (
            const T value )  [pure virtual]
```

Sprawdzenie czy element istnieje, szukany po wartości.

Parametry

value	Szukana wartość.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementowany w **Vector**< **T** > (str. 198), **ListCircle**< **T** > (str. 77), **ListBi**< **T** > (str. 57) i **List**< **T** > (str. 39).

4.16.2.10 Insert() [1/3]

```
template<class T >
virtual void  Sequences< T >::Insert (
            int position,
            const T & value )  [pure virtual]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
value	Nowy element.

Implementowany w **Vector**< **T** > (str. 199), **ListCircle**< **T** > (str. 79), **ListBi**< **T** > (str. 59) i **List**< **T** > (str. 41).

4.16.2.11 Insert() [2/3]

```
template<class T >
virtual void  Sequences< T >::Insert (
            int position,
            int n,
            const T & value )  [pure virtual]
```

Wypełnienie kontenera elementem przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
n	Ilość kopii nowego elementu do wstawienia.
value	Nowy element.

Implementowany w **Vector**< **T** > (str. 199), **ListCircle**< **T** > (str. 79), **ListBi**< **T** > (str. 59) i **List**< **T** > (str. 41).

4.16.2.12 Insert() [3/3]

```
template<class T >
virtual void  Sequences< T >::Insert (
            int position,
            int n,
            T * valuePtr )  [pure virtual]
```

Wypełnienie kontenera przedziałem przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
n	Ilość elementów do wstawienia.
valuePtr	Pointer na nowe elementy.

Implementowany w **Vector**< **T** > (str. 200), **ListCircle**< **T** > (str. 79), **ListBi**< **T** > (str. 60) i **List**< **T** > (str. 42).

4.16.2.13 insertionsort()

```
template<class T >
void  Sequences< T >::insertionsort (
            T * tab,
            int count )  [inline]
```

Funkcja wykorzystywana w **Sort()** (str. 152)

Parametry

tab	Wskaźnik na tablicę elementów kontenera.
count	Ilość elementów w tablicy.

4.16.2.14 MaxSize()

```
template<class T >
```

```
virtual int  Sequences< T >::MaxSize ( )  [pure virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Container< T >** (str. 16).

Implementowany w **Vector< T >** (str. 200), **ListCircle< T >** (str. 80), **ListBi< T >** (str. 60) i **List< T >** (str. 42).

4.16.2.15 PopBack()

```
template<class T >
virtual void  Sequences< T >::PopBack ( )  [pure virtual]
```

Usunięcie ostatniego elementu i zmniejszenie wielkości o jeden.

Implementowany w **Vector< T >** (str. 202), **ListCircle< T >** (str. 82), **ListBi< T >** (str. 62) i **List< T >** (str. 44).

4.16.2.16 PopFront()

```
template<class T >
virtual void  Sequences< T >::PopFront ( )  [pure virtual]
```

Usunięcie pierwszego elementu i zmniejszenie wielkości o jeden.

Implementowany w **Vector< T >** (str. 202), **ListCircle< T >** (str. 82), **ListBi< T >** (str. 62) i **List< T >** (str. 44).

4.16.2.17 PushBack()

```
template<class T >
virtual void  Sequences< T >::PushBack (
            const T & value )  [pure virtual]
```

Dodanie nowego elementu na końcu kontenera, po ostatnimm oraz zwiększenie wielkości o jeden.

Parametry

value	Nowy element.

Implementowany w **Vector< T >** (str. 202), **ListCircle< T >** (str. 82), **ListBi< T >** (str. 62) i **List< T >** (str. 44).

4.16.2.18 PushFront()

```
template<class T >
virtual void  Sequences< T >::PushFront (
            const T & value )  [pure virtual]
```

Dodanie nowego elementu na początku kontenera, po ostatnimm oraz zwiększenie wielkości o jeden.

Parametry

value	Nowy element.

Implementowany w **Vector**< **T** > (str. 203), **ListCircle**< **T** > (str. 82), **ListBi**< **T** > (str. 63) i **List**< **T** > (str. 45).

4.16.2.19 Resize() [1/2]

```
template<class T >
virtual void  Sequences< T >::Resize (
            int n )  [pure virtual]
```

Zmiana wielkości kontenera na podaną.

- Mniejsza niż aktualna: ucięcie kontenera.
- Większa niż aktualna: nowe elementy są puste.

Parametry

n	Nowa wielkość kontenera.

Implementowany w **Vector**< **T** > (str. 203), **ListCircle**< **T** > (str. 83), **ListBi**< **T** > (str. 63) i **List**< **T** > (str. 45).

4.16.2.20 Resize() [2/2]

```
template<class T >
virtual void  Sequences< T >::Resize (
            int n,
            const T & value )  [pure virtual]
```

Zmiana wielkości kontenera na podaną i dodanie nowych elementów.

- Mniejsza niż aktualna: ucięcie kontenera.
- Większa niż aktualna: nowe elementy mają podaną wartość.

Parametry

n	Nowa wielkość kontenera.
value	Wartość nowych elementów.

Implementowany w **Vector**< **T** > (str. 204), **ListCircle**< **T** > (str. 83), **ListBi**< **T** > (str. 64) i **List**< **T** > (str. 46).

4.16.2.21 returnEmpty()

```
template<class T >
virtual T  Sequences< T >::returnEmpty ( )  [pure virtual]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

Implementowany w **Vector**< **T** > (str. 204), **ListCircle**< **T** > (str. 84), **ListBi**< **T** > (str. 64) i **List**< **T** > (str. 46).

4.16.2.22 Reverse()

```
template<class T >
virtual void  Sequences< T >::Reverse ( )  [pure virtual], [noexcept]
```

Odwrócenie elementów kontenera kolejnością.

Implementowany w **Vector**< **T** > (str. 204), **ListCircle**< **T** > (str. 84), **ListBi**< **T** > (str. 64) i **List**< **T** > (str. 46).

4.16.2.23 Show()

```
template<class T >
virtual void  Sequences< T >::Show ( )  [pure virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Container**< **T** > (str. 17).

Implementowany w **Vector**< **T** > (str. 205), **ListCircle**< **T** > (str. 84), **ListBi**< **T** > (str. 65) i **List**< **T** > (str. 47).

4.16.2.24 Size()

```
template<class T >
virtual int  Sequences< T >::Size ( ) const  [pure virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Container< T >** (str. 17).

Implementowany w **Vector< T >** (str. 205), **ListCircle< T >** (str. 85), **ListBi< T >** (str. 65) i **List< T >** (str. 47).

4.16.2.25 Sort()

```
template<class T >
virtual void  Sequences< T >::Sort ( )  [pure virtual]
```

Sortowanie elementów w kontenerze

Implementowany w **Vector< T >** (str. 205), **ListCircle< T >** (str. 85), **ListBi< T >** (str. 65) i **List< T >** (str. 47).

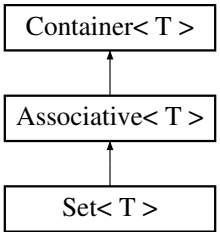
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Sequences.h**

4.17 Dokumentacja szablonu klasy Set< T >

```
#include <Set.h>
```

Diagram dziedziczenia dla Set< T >



Metody publiczne

- **Set** ()
- **Set** (T **key**)
- **Set** (const **Set** &other)
- ∼**Set** ()
- **Set**< T > & **operator=** (const **Set** &other)
- bool **operator==** (const **Set** &other) const
- bool **operator!=** (const **Set** &other) const
- void **PopBack** ()
- void **PopFront** ()
- T ∗ **AtPtr** (T **key**)
- T ∗ **End** () const noexcept
- T ∗ **Rend** () noexcept
- T ∗ **Begin** () const noexcept
- T ∗ **Rbegin** () noexcept
- void **Insert** (const T &data)
- void **Insert** (int position, const T &data)
- void **Emplace** (T **key**, int position)
- void **Swap** (**Set** &tmp)
- bool **findIfExist** (const T value)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** () noexcept
- void **Erase** (int position)
- void **Erase** (int first, int last)
- int **MaxSize** ()
- void **Remove** (T value)
- void **Reverse** () noexcept
- void **Sort** ()
- bool **ifUnique** (T **key**)
- void **Show** ()

Metody prywatne

- string **showType** ()
- void **insertionsort** (T ∗tab, int count)

Atrybuty prywatne

- T ∗ **key**
- int **size**

4.17.1 Opis szczegółowy

template<**class T**>
class Set< T >

Deklaracja klasy **Set** (str. 152)

- Odpowiednik STLowego Seta/zbioru.
- Dziedzicząca po **Associative** (str. 10).

Parametry

key	Dynamicznie zaalokowana tablica Tów.
data	Dynamicznie zaalokowana tablica Tów.
size	Ilość wpisanych elementów

4.17.2 Dokumentacja konstruktora i destruktora

4.17.2.1 Set() [1/3]

```
template<class T >
Set< T >::  Set ( )  [inline]
```

Konstruktor domyślny

4.17.2.2 Set() [2/3]

```
template<class T >
Set< T >::  Set (
            T key )  [inline]
```

Konstruktor jednooargumentowy.

- Dodaje podany klucz jako pierwszy element.

Parametry

key	Wartość do wpisania - klucz.

4.17.2.3 Set() [3/3]

```
template<class T >
Set< T >::  Set (
            const  Set< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.17.2.4 ~Set()

```
template<class T >
Set< T >::~ Set ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.17.3 Dokumentacja funkcji składowych

4.17.3.1 allowedType()

```
template<class T >
bool  Set< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Associative< T >** (str. 11).

4.17.3.2 AtPtr()

```
template<class T >
T*  Set< T >::AtPtr (
            T key )  [inline]
```

Zwrócenie wskaźnika na wartość spod podanego klucza.

Parametry

key	Podany klucz.

Zwraca

Wskaźnik na dany element.

4.17.3.3 Begin()

```
template<class T >
T*  Set< T >::Begin ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.17.3.4 Clear()

```
template<class T >
void  Set< T >::Clear ( )  [inline], [virtual], [noexcept]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Associative< T >** (str. 11).

4.17.3.5 Emplace()

```
template<class T >
void  Set< T >::Emplace (
            T key,
            int position )  [inline]
```

Wstawienie nowego elementu na podaną pozycje.

Parametry

key	Nowy element - klucz.
position	Pozycja, na którą metoda wstawia element.

4.17.3.6 Empty()

```
template<class T >
```

```
bool  Set< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Associative< T >** (str. 12).

4.17.3.7 End()

```
template<class T >
T*  Set< T >::End ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.17.3.8 Erase() [1/2]

```
template<class T >
void  Set< T >::Erase (
            int first,
            int last )  [inline], [virtual]
```

Usunięcie wybranych elementów z przedziału.

Parametry

first	Pierwszy indeks przedziału.
last	Ostatni indeks przedziału.

Implementuje **Associative< T >** (str. 12).

4.17.3.9 Erase() [2/2]

```
template<class T >
void  Set< T >::Erase (
            int position )  [inline], [virtual]
```

Usunięcie wybranego elementu po pozycji.

Parametry

position	Pozycja elementu do usunięcia.

Implementuje **Associative**< **T** > (str. 12).

4.17.3.10 findIfExist()

```
template<class T >
bool  Set< T >::findIfExist (
            const T value )  [inline]
```

Sprawdzenie czy element istnieje, szukany po wartości.

Parametry

value	Szukana wartość.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

4.17.3.11 ifUnique()

```
template<class T >
bool  Set< T >::ifUnique (
            T key )  [inline]
```

Sprawdzenie czy klucz jest wyjątkowy.

4.17.3.12 Insert() [1/2]

```
template<class T >
void  Set< T >::Insert (
            const T & data )  [inline]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

data	Nowy element - klucz.

4.17.3.13 Insert() [2/2]

```
template<class T >
void  Set< T >::Insert (
            int position,
            const T & data )  [inline]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
data	Nowy element.

4.17.3.14 insertionsort()

```
template<class T >
void  Set< T >::insertionsort (
            T * tab,
            int count )  [inline], [private]
```

Funkcja wykorzystywana w **Sort()** (str. 163)

Parametry

tab	Wskaźnik na tablicę elementów kontenera.
count	Ilość elementów w tablicy.

4.17.3.15 MaxSize()

```
template<class T >
int  Set< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Associative< T >** (str. 13).

4.17.3.16 operator"!=()

```
template<class T >
bool  Set< T >::operator!= (
            const  Set< T > & other ) const  [inline]
```

Operator przeciążony porównania "!="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.17.3.17 operator=()

```
template<class T >
Set<T>&  Set< T >::operator= (
             const  Set< T > & other )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.17.3.18 operator==()

```
template<class T >
bool  Set< T >::operator== (
             const  Set< T > & other ) const  [inline]
```

Operator przeciążony porównania "=="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.17.3.19 PopBack()

```
template<class T >
void  Set< T >::PopBack ( )  [inline], [virtual]
```

Usunięcie ostatniego elementu i zmniejszenie wielkości o jeden.

Implementuje **Associative< T >** (str. 13).

4.17.3.20 PopFront()

```
template<class T >
void  Set< T >::PopFront ( )  [inline], [virtual]
```

Usunięcie pierwszego elementu i zmniejszenie wielkości o jeden.

Implementuje **Associative< T >** (str. 13).

4.17.3.21 Rbegin()

```
template<class T >
T*  Set< T >::Rbegin ( )  [inline], [noexcept]
```

Zamienienie key i data oraz zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.17.3.22 Remove()

```
template<class T >
void  Set< T >::Remove (
            T value )  [inline]
```

Usunięcie wybranego elementu po wartości.

Parametry

value	Wartość elementu do usunięcia.

4.17.3.23 Rend()

```
template<class T >
T*  Set< T >::Rend ( )  [inline], [noexcept]
```

Zamienienie key oraz zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.17.3.24 returnEmpty()

```
template<class T >
T  Set< T >::returnEmpty ( )  [inline]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

4.17.3.25 Reverse()

```
template<class T >
void  Set< T >::Reverse ( )  [inline], [virtual], [noexcept]
```

Odwrócenie elementów kontenera kolejnością.

Implementuje **Associative**< **T** > (str. 14).

4.17.3.26 Show()

```
template<class T >
void  Set< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Associative**< **T** > (str. 14).

4.17.3.27 showType()

```
template<class T >
string  Set< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.17.3.28 Size()

```
template<class T >
int  Set< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Associative< T >** (str. 14).

4.17.3.29 Sort()

```
template<class T >
void  Set< T >::Sort ( )  [inline], [virtual]
```

Sortowanie elementów w kontenerze

Implementuje **Associative< T >** (str. 14).

4.17.3.30 Swap()

```
template<class T >
void  Set< T >::Swap (
              Set< T > & tmp )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.17.4 Dokumentacja atrybutów składowych

4.17.4.1 key

```
template<class T >
T* Set< T >::key [private]
```

4.17.4.2 size

```
template<class T >
int Set< T >::size [private]
```

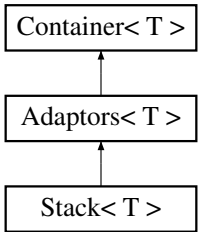
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Set.h**

4.18 Dokumentacja szablonu klasy Stack< T >

```
#include <Stack.h>
```

Diagram dziedziczenia dla Stack< T >



Metody publiczne

- **Stack** ()
- **Stack** (T value)
- **Stack** (T value, int **size**)
- **Stack** (const **Stack** &other)
- ~**Stack** ()
- **Stack**< T > & **operator=** (const **Stack** &value)
- bool **operator==** (const **Stack** &value) const
- bool **operator!=** (const **Stack** &value) const
- void **Push** (const T &value)
- void **Pop** ()
- T & **Top** ()
- T * **TopPtr** ()
- void **Swap** (**Stack** &tmp)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** () noexcept
- int **MaxSize** ()
- void **Show** ()

Metody prywatne

- string **showType** ()

Atrybuty prywatne

- T * **data**
- int **size**

4.18.1 Opis szczegółowy

template<class T>
class Stack< T >

Deklaracja klasy **Stack** (str. 164)

- Odpowiednik STLowego stosu.
- Dziedzicząca po **Adaptors** (str. 7).

Parametry

data	Dynamicznie zaalokowana tablica Tów.
size	Ilość wpisanych elementów

4.18.2 Dokumentacja konstruktora i destruktora

4.18.2.1 Stack() [1/4]

```
template<class T >
Stack< T >::  Stack ( )  [inline]
```

Konstruktor domyślny

4.18.2.2 Stack() [2/4]

```
template<class T >
Stack< T >::  Stack (
             T value )  [inline]
```

Konstruktor jednoargumentowy.

- Dodaje podaną wartość jako pierwszy element.

Parametry

value	Wartość do wpisania.

4.18.2.3 Stack() [3/4]

```
template<class T >
Stack< T >::  Stack (
            T value,
            int size )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o podanej wartości.

Parametry

value	Wartość do wpisania.
size	Ilość elementów do stworzenia.

4.18.2.4 Stack() [4/4]

```
template<class T >
Stack< T >::  Stack (
            const  Stack< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.18.2.5 ~Stack()

```
template<class T >
Stack< T >::~ Stack ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.18.3 Dokumentacja funkcji składowych

4.18.3.1 allowedType()

```
template<class T >
bool  Stack< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Adaptors< T >** (str. 8).

4.18.3.2 Clear()

```
template<class T >
void  Stack< T >::Clear ( )  [inline], [virtual], [noexcept]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0.

Implementuje **Adaptors< T >** (str. 8).

4.18.3.3 Empty()

```
template<class T >
bool  Stack< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Adaptors< T >** (str. 8).

4.18.3.4 MaxSize()

```
template<class T >
int  Stack< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener.

Zwraca

Maksymalna wielkość kontenera.

Implementuje **Adaptors< T >** (str. 8).

4.18.3.5 operator"!=()

```
template<class T >
bool  Stack< T >::operator!= (
            const  Stack< T > & value ) const  [inline]
```

Operator przeciążony porównania "!="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.18.3.6 operator=()

```
template<class T >
Stack<T>&  Stack< T >::operator= (
             const  Stack< T > & value )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.18.3.7 operator==()

```
template<class T >
bool  Stack< T >::operator== (
            const  Stack< T > & value ) const  [inline]
```

Operator przeciążony porównania "=="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.18.3.8 Pop()

```
template<class T >
void  Stack< T >::Pop ( )  [inline], [virtual]
```

Usunięcie elementu z TOPu i zmniejszenie jego wielkości.

Implementuje **Adaptors< T >** (str. 9).

4.18.3.9 Push()

```
template<class T >
void  Stack< T >::Push (
            const T & value )  [inline], [virtual]
```

Dodanie nowego elementu na TOP kontenera i zwiększenie jego wielkości.

Parametry

value	Nowy element.

Implementuje **Adaptors< T >** (str. 9).

4.18.3.10 returnEmpty()

```
template<class T >
T  Stack< T >::returnEmpty ( )  [inline], [virtual]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

Implementuje **Adaptors**< **T** > (str. 9).

4.18.3.11 Show()

```
template<class T >
void  Stack< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Adaptors**< **T** > (str. 10).

4.18.3.12 showType()

```
template<class T >
string  Stack< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.18.3.13 Size()

```
template<class T >
int  Stack< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Adaptors**< **T** > (str. 10).

4.18.3.14 Swap()

```
template<class T >
void  Stack< T >::Swap (
              Stack< T > & tmp )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.18.3.15 Top()

```
template<class T >
T&  Stack< T >::Top ( )  [inline]
```

Zwrócenie referencji na TOP.

Zwraca

Referencja na TOP.

4.18.3.16 TopPtr()

```
template<class T >
T*  Stack< T >::TopPtr ( )  [inline]
```

Zwrócenie wskaźnika na TOP.

Zwraca

Wskaźnik na TOP.

4.18.4 Dokumentacja atrybutów składowych

4.18.4.1 data

```
template<class T >
T*  Stack< T >::data  [private]
```

4.18.4.2 size

```
template<class T >
int  Stack< T >::size  [private]
```

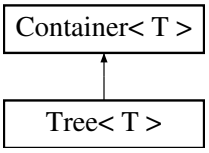
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Stack.h**

4.19 Dokumentacja szablonu klasy Tree< T >

`#include <Tree.h>`

Diagram dziedziczenia dla Tree< T >



Metody publiczne

- **Tree** ()
- **Tree** (T value)
- **Tree** (**Tree** &other)
- ∼**Tree** ()
- **Tree**< T > & **operator=** (**Tree** &rhs)
- bool **operator==** (**Tree** &rhs)
- bool **operator!=** (**Tree** &rhs)
- void **PopMin** ()
- void **PopMax** ()
- **TreeNode**< T > ∗ **findValue** (T value)
- **TreeNode**< T > ∗ **findParent** (T value)
- **TreeNode**< T > ∗ **findMinPtr** ()
- **TreeNode**< T > & **findMin** ()
- **TreeNode**< T > ∗ **findMaxPtr** ()
- **TreeNode**< T > & **findMax** ()
- void **Insert** (const T &value)
- void **Emplace** (T oldValue, T newValue)
- void **Swap** (**Tree** &rhs)
- bool **findIfExist** (T value)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- void **Clear** ()
- void **Remove** (T value)
- int **MaxSize** ()
- int **Height** ()
- int **leafCount** ()
- int **heightDiffrence** ()
- void **inOrderShow** ()
- void **preOrderShow** ()
- void **postOrderShow** ()
- void **levelOrderShow** ()
- **TreeNode**< T > ∗ **leftRotation** ()
- **TreeNode**< T > ∗ **rightRotation** ()
- void **Show** ()

Metody prywatne

- **TreeNode**< T > ∗ **CopyRec** (**TreeNode**< T > ∗rhs)
- bool **sameRec** (**TreeNode**< T > ∗ **root**, **TreeNode**< T > ∗rhs)
- **TreeNode**< T > ∗ **InsertRec** (**TreeNode**< T > ∗ **root**, T value)
- **TreeNode**< T > ∗ **removeRec** (**TreeNode**< T > ∗ **root**, T value)
- **TreeNode**< T > ∗ **findParentRec** (**TreeNode**< T > ∗ **root**, T value)
- **TreeNode**< T > ∗ **findMinRec** (**TreeNode**< T > ∗ **root**)
- **TreeNode**< T > ∗ **findMaxRec** (**TreeNode**< T > ∗ **root**)
- **TreeNode**< T > ∗ **findRec** (**TreeNode**< T > ∗ **root**, T value)
- void **clearRec** (**TreeNode**< T > ∗ **root**)
- int **heightRec** (**TreeNode**< T > ∗ **root**)
- int **leafRec** (**TreeNode**< T > ∗ **root**)
- void **InOrderRec** (**TreeNode**< T > ∗ **root**)
- void **preOrderRec** (**TreeNode**< T > ∗ **root**)
- void **postOrderRec** (**TreeNode**< T > ∗ **root**)
- void **levelOrderRec** (**TreeNode**< T > ∗ **root**, int level)
- void **treeRec** (**TreeNode**< T > ∗ **root**, int space, int count)
- string **showType** ()

Atrybuty prywatne

- **TreeNode**< T > ∗ **root**
- int **size**

4.19.1 Dokumentacja konstruktora i destruktora

4.19.1.1 Tree() [1/3]

```
template<class T >
Tree< T >::  Tree ( )  [inline]
```

Konstruktor domyślny

4.19.1.2 Tree() [2/3]

```
template<class T >
Tree< T >::  Tree (
            T value )  [inline]
```

Konstruktor jednoargumentowy.

- Dodaje podaną wartość jako pierwszy element.

Parametry

value	Wartość do wpisania.

4.19.1.3 Tree() [3/3]

```
template<class T >
Tree< T >::  Tree (
             Tree< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.19.1.4 ~Tree()

```
template<class T >
Tree< T >::~ Tree ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.19.2 Dokumentacja funkcji składowych

4.19.2.1 allowedType()

```
template<class T >
bool  Tree< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Container< T >** (str. 16).

4.19.2.2 Clear()

```
template<class T >
void  Tree< T >::Clear ( )  [inline], [virtual]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Container< T >** (str. 16).

4.19.2.3 clearRec()

```
template<class T >
void  Tree< T >::clearRec (
              TreeNode< T > * root )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla Clear. Wyczyszczenie drzewa.

Parametry

root	Wskaźnik na element drzewa.

4.19.2.4 CopyRec()

```
template<class T >
TreeNode<T>*  Tree< T >::CopyRec (
              TreeNode< T > * rhs )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla operatora =. Skopiowanie do kontenera wartości z drugiego.

Parametry

rhs	Wskaźnik na element z drugiego drzewa.

Zwraca

Wskaźnik na skopiowane elementy.

4.19.2.5 Emplace()

```
template<class T >
void  Tree< T >::Emplace (
             T oldValue,
             T newValue )  [inline]
```

Wstawienie nowego elementu na pozycje podanego starego.

Parametry

oldValue	Stary element.
newValue	Nowy element.

4.19.2.6 Empty()

```
template<class T >
bool  Tree< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Container< T >** (str. 16).

4.19.2.7 findIfExist()

```
template<class T >
bool  Tree< T >::findIfExist (
            T value )  [inline]
```

Stwierdzenie czy dany element istnieje w drzewie.

Parametry

value	Wartość, której poszukujemy.

Zwraca

Status operacji: true, jeżeli element z podaną wartością został odnaleziony; false, jeśli nie.

4.19.2.8 findMax()

```
template<class T >
TreeNode<T>&  Tree< T >::findMax ( )  [inline]
```

Znalezienie wartości największej w drzewie.

Zwraca

pointer na maxa.

4.19.2.9 findMaxPtr()

```
template<class T >
TreeNode<T>*  Tree< T >::findMaxPtr ( )  [inline]
```

Znalezienie wartości największej w drzewie.

Zwraca

pointer na maxa.

4.19.2.10 findMaxRec()

```
template<class T >
TreeNode<T>*  Tree< T >::findMaxRec (
              TreeNode< T > * root )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla findMax. Znalezienie elementu drzewa z największą wartością.

Parametry

root	Wskaźnik na element drzewa.

Zwraca

Wskaźnik na element z największą wartością.

4.19.2.11 findMin()

```
template<class T >
TreeNode<T>&  Tree< T >::findMin ( )  [inline]
```

Znalezienie wartości najmniejszej w drzewie.

Zwraca

pointer na mina.

4.19.2.12 findMinPtr()

```
template<class T >
TreeNode<T>*  Tree< T >::findMinPtr ( )  [inline]
```

Znalezienie wartości najmniejszej w drzewie.

Zwraca

pointer na mina.

4.19.2.13 findMinRec()

```
template<class T >
TreeNode<T>*  Tree< T >::findMinRec (
              TreeNode< T > * root )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla findMin. Znalezienie elementu drzewa z najmniejszą wartością.

Parametry

root	Wskaźnik na element drzewa.

Zwraca

Wskaźnik na element z najmniejszą wartością.

4.19.2.14 findParent()

```
template<class T >
TreeNode<T>*  Tree< T >::findParent (
             T value )  [inline]
```

Znalezienie rodzica dla podanej wartości.

Parametry

value	Wartość, dla której poszukujemy rodzica.

Zwraca

Pointer na rodzica.

4.19.2.15 findParentRec()

```
template<class T >
TreeNode<T>*  Tree< T >::findParentRec (
              TreeNode< T > * root,
             T value )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla findParent. Znalezienie rodzica elementu z podaną wartością.

Parametry

root	Wskaźnik na element drzewa.
value	Wartość do znalezienia.

Zwraca

Wskaźnik na rodzica elementu z podaną wartością.

4.19.2.16 findRec()

```
template<class T >
TreeNode<T>*  Tree< T >::findRec (
              TreeNode< T > * root,
             T value )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla findElement. Znalezienie elementu drzewa z podaną wartością.

Parametry

root	Wskaźnik na element drzewa.
value	Wartość do znalezienia.

Zwraca

Wskaźnik na element z podaną wartością.

4.19.2.17 findValue()

```
template<class T >
TreeNode<T>*  Tree< T >::findValue (
             T value )  [inline]
```

Znalezienie elementu w drzwie dla podanej wartości.

Parametry

value	Wartość, której poszukujemy.

Zwraca

Pointer na element.

4.19.2.18 Height()

```
template<class T >
int  Tree< T >::Height ( )  [inline]
```

Zliczenie wysokosci drzewa.

Zwraca

Wysokość drzewa.

4.19.2.19 heightDiffrence()

```
template<class T >
int  Tree< T >::heightDiffrence ( )  [inline]
```

Zliczenie różnicy miedzy lewą, a prawą wysokością drzewa.

Zwraca

Różnica między wysokościami drzewa.

4.19.2.20 heightRec()

```
template<class T >
int  Tree< T >::heightRec (
              TreeNode< T > * root )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla Height. Zliczanie wysokości drzewa.

Parametry

root	Wskaźnik na element drzewa.

4.19.2.21 InOrderRec()

```
template<class T >
void  Tree< T >::InOrderRec (
              TreeNode< T > * root )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla inOrderShow. Wyświetlanie drzewa według rosnącej kolejności.

Parametry

root	Wskaźnik na element drzewa.

4.19.2.22 inOrderShow()

```
template<class T >
void  Tree< T >::inOrderShow ( )  [inline]
```

Wyświetlenie wszystkich elementów kontenera rosnąco.

4.19.2.23 Insert()

```
template<class T >
void  Tree< T >::Insert (
             const T & value )  [inline]
```

Dodanie podanego elementu i zwiększenie wiekości kontenera.

Parametry

value	Nowy element.

4.19.2.24 InsertRec()

```
template<class T >
TreeNode<T>*  Tree< T >::InsertRec (
              TreeNode< T > * root,
             T value )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla Insert. Dodanie elementu z podaną wartością.

Parametry

root	Wskaźnik na element drzewa.
value	Wartość do dodania.

Zwraca

Wskaźnik na dodany element z podaną wartością.

4.19.2.25 leafCount()

```
template<class T >
int  Tree< T >::leafCount ( )  [inline]
```

Zliczenie ilości liści drzewa - elementów bez dzieci.

Zwraca

Ilość liści w drzewie.

4.19.2.26 leafRec()

```
template<class T >
int  Tree< T >::leafRec (
              TreeNode< T > * root )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla leafCount. Zliczanie ilości liści w drzewie.

Parametry

root	Wskaźnik na element drzewa.

4.19.2.27 leftRotation()

```
template<class T >
TreeNode<T>*  Tree< T >::leftRotation ( )  [inline]
```

Rotacja drzewa w lewą stronę.

Zwraca

Pointer na korzeń po rotacji.

4.19.2.28 levelOrderRec()

```
template<class T >
void  Tree< T >::levelOrderRec (
              TreeNode< T > * root,
             int level )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla levelOrderShow. Wyświetlanie drzewa według jego poziomów.

Parametry

root	Wskaźnik na element drzewa.
level	Ilość poziomów do wyświetlenia.

4.19.2.29 levelOrderShow()

```
template<class T >
void  Tree< T >::levelOrderShow ( )  [inline]
```

Wyświetlenie drzewa według poziomów drzewa.

4.19.2.30 MaxSize()

```
template<class T >
int  Tree< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Container< T >** (str. 16).

4.19.2.31 operator"!=()

```
template<class T >
bool  Tree< T >::operator!= (
              Tree< T > & rhs )  [inline]
```

Operator przeciążony porównania "!="

Parametry

rhs	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.19.2.32 operator=()

```
template<class T >
Tree<T>&  Tree< T >::operator= (
              Tree< T > & rhs )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.19.2.33 operator==()

```
template<class T >
bool  Tree< T >::operator== (
              Tree< T > & rhs )  [inline]
```

Operator przeciążony porównania "=="

Parametry

rhs	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.19.2.34 PopMax()

```
template<class T >
void  Tree< T >::PopMax ( )  [inline]
```

Usunięcie najmniejszego elementu i zmniejszenie wielkości o jeden.

4.19.2.35 PopMin()

```
template<class T >
void  Tree< T >::PopMin ( )  [inline]
```

Usunięcie najmniejszego elementu i zmniejszenie wielkości o jeden.

4.19.2.36 postOrderRec()

```
template<class T >
void  Tree< T >::postOrderRec (
              TreeNode< T > * root )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla postOrderShow. Wyświetlanie drzewa według kolejności Post.

Parametry

root	Wskaźnik na element drzewa.

4.19.2.37 postOrderShow()

```
template<class T >
void  Tree< T >::postOrderShow ( )  [inline]
```

Wyświetlenie wszystkich elementów kontenera in kolejności Post.

4.19.2.38 preOrderRec()

```
template<class T >
void  Tree< T >::preOrderRec (
              TreeNode< T > * root )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla preOrderShow. Wyświetlanie drzewa według kolejności Pre.

Parametry

root	Wskaźnik na element drzewa.

4.19.2.39 preOrderShow()

```
template<class T >
void  Tree< T >::preOrderShow ( )  [inline]
```

Wyświetlenie wszystkich elementów kontenera in kolejności Pre.

4.19.2.40 Remove()

```
template<class T >
void  Tree< T >::Remove (
             T value )  [inline]
```

Usunięcie wybranego elementu ze zwolnieniem pamięci.

Parametry

value	Wartość elementu do usunięcia.

Zwraca

Status operacji: true, jeżeli element istniał i został usunięty; false jeśli nie.

4.19.2.41 removeRec()

```
template<class T >
TreeNode<T>*  Tree< T >::removeRec (
              TreeNode< T > * root,
             T value )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla Remove. Usunięcie elementu z podaną wartością.

Parametry

root	Wskaźnik na element drzewa.
value	Wartość do usunięcia.

Zwraca

Wskaźnik na element z podaną wartością.

4.19.2.42 returnEmpty()

```
template<class T >
T  Tree< T >::returnEmpty ( )  [inline]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

4.19.2.43 rightRotation()

```
template<class T >
TreeNode<T>*  Tree< T >::rightRotation ( )  [inline]
```

Rotacja drzewa w prawą stronę.

Zwraca

Pointer na korzeń po rotacji.

4.19.2.44 sameRec()

```
template<class T >
bool  Tree< T >::sameRec (
              TreeNode< T > * root,
              TreeNode< T > * rhs )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza dla operatora ==. Sprawdzenie czy oba kontenery są identyczne.

Parametry

root	Wskaźnik na element pierwszego drzewa.
rhs	Wskaźnik na element drugiego drzewa.

Zwraca

Status operacji: true, jeżeli są równe; false, jeśli nie.

4.19.2.45 Show()

```
template<class T >
void  Tree< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie kontenera w formie drzewa.

Implementuje **Container< T >** (str. 17).

4.19.2.46 showType()

```
template<class T >
string  Tree< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.19.2.47 Size()

```
template<class T >
int  Tree< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Container< T >** (str. 17).

4.19.2.48 Swap()

```
template<class T >
void  Tree< T >::Swap (
              Tree< T > & rhs )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

rhs	Drugi kontener.

4.19.2.49 treeRec()

```
template<class T >
void  Tree< T >::treeRec (
              TreeNode< T > * root,
             int space,
             int count )  [inline], [private]
```

Rekurencyjna funkcja pomocnicza treeShow. Wyświetlanie kontenera w formie drzewa .

Parametry

root	Wskaźnik na element drzewa.
space	Odległość między poziomami.
count	Podana odległość.

4.19.3 Dokumentacja atrybutów składowych

4.19.3.1 root

```
template<class T >
TreeNode<T>*  Tree< T >::root  [private]
```

4.19.3.2 size

```
template<class T >
int  Tree< T >::size  [private]
```

Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Tree.h**

4.20 Dokumentacja szablonu klasy TreeNode< T >

```
#include <Nudes.h>
```

Metody publiczne

- **TreeNode** ()
- **TreeNode** (T **data**, **TreeNode** * **pLeft**, **TreeNode** * **pRight**)
- T **returnEmpty** ()

Atrybuty publiczne

- T **data**
- **TreeNode** * **pLeft**
- **TreeNode** * **pRight**

4.20.1 Opis szczegółowy

template<**class T**>
class TreeNode< T >

-Klasa pomocnicza klasy **Tree** (str. 172)

Parametry

data	Wartość elementu.
pLeft	Wskaźnik na lewe dziecko.
pRight	Wskażnik na prawe dziecko.

4.20.2 Dokumentacja konstruktora i destruktora

4.20.2.1 TreeNode() [1/2]

```
template<class T >
TreeNode< T >::  TreeNode ( )  [inline]
```

Konstruktor domyślny

4.20.2.2 TreeNode() [2/2]

```
template<class T >
TreeNode< T >::  TreeNode (
            T data,
             TreeNode< T > * pLeft,
             TreeNode< T > * pRight )  [inline]
```

Konstruktor trójargumentowy

Parametry

data	Wartość do wpisania.
pLeft	Wskaźnik na lewe dziecko.
pRight	Wskażnik na prawe dziecko.

4.20.3 Dokumentacja funkcji składowych

4.20.3.1 returnEmpty()

```
template<class T >
T  TreeNode< T >::returnEmpty ( )  [inline]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

4.20.4 Dokumentacja atrybutów składowych

4.20.4.1 data

```
template<class T >
T  TreeNode< T >::data
```

4.20.4.2 pLeft

```
template<class T >
TreeNode*  TreeNode< T >::pLeft
```

4.20.4.3 pRight

```
template<class T >
TreeNode*  TreeNode< T >::pRight
```

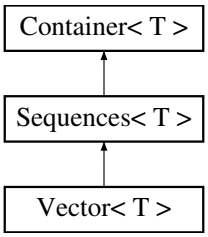
Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Nudes.h**

4.21 Dokumentacja szablonu klasy Vector< T >

`#include <Vector.h>`

Diagram dziedziczenia dla Vector< T >



Metody publiczne

- **Vector** ()
- **Vector** (T value)
- **Vector** (T value, int **size**)
- **Vector** (**Vector** &other)
- ~**Vector** ()
- **Vector**< T > & **operator=** (const **Vector** &value)
- bool **operator==** (const **Vector** &value) const
- bool **operator!=** (const **Vector** &value) const
- T **operator[ ]** (const int index)
- void **PopBack** ()
- void **PopFront** ()
- void **PushBack** (const T &value)
- void **PushFront** (const T &value)
- T & **At** (int index)
- T * **AtPtr** (int index)
- T & **Back** ()
- T * **End** () const noexcept
- T & **Front** ()
- T * **Begin** () const noexcept
- void **Insert** (int position, const T &value)
- void **Insert** (int position, int n, const T &value)
- void **Insert** (int position, int n, T *valuePtr)
- void **Emplace** (T value, int position)
- void **Assign** (int count, T *valuePtr)
- void **Assign** (int count, const T value)
- void **Swap** (**Vector** &tmp)
- int **findElement** (const T value)
- bool **findIfExist** (const T value)
- bool **allowedType** ()
- T **returnEmpty** ()
- bool **Empty** () const noexcept
- int **Size** () const noexcept
- const int **Capacity** () const noexcept
- void **Clear** ()
- void **Erase** (int position)
- void **Erase** (int first, int last)

- int **MaxSize** ()
- void **Reserve** (int n)
- void **Resize** (int n)
- void **Resize** (int n, const T &value)
- void **ShrinkToFit** ()
- void **Sort** ()
- void **Reverse** () noexcept
- void **Show** ()

Metody prywatne

- string **showType** ()

Atrybuty prywatne

- T * **data**
- int **size**
- int **capacity**

4.21.1 Opis szczegółowy

template<class T>
class Vector< T >

Deklaracja klasy **Vector** (str. 191)

- Odpowiednik STLowego wektora.
- Dziedzicząca po **Sequences** (str. 143).

Parametry

data	Dynamicznie zaalokowana tablica Tów.
size	Ilość wpisanych elementów
capacity	Ilość zaalokowanej pamięci.

4.21.2 Dokumentacja konstruktora i destruktora

4.21.2.1 Vector() [1/4]

```
template<class T >
Vector< T >::  Vector ( )  [inline]
```

Konstruktor domyślny

4.21.2.2 Vector() [2/4]

```
template<class T >
Vector< T >::  Vector (
             T value )  [inline]
```

Konstruktor jednoargumentowy.

- Dodaje podaną wartość jako pierwszy element.

Parametry

value	Wartość do wpisania.

4.21.2.3 Vector() [3/4]

```
template<class T >
Vector< T >::  Vector (
             T value,
             int size )  [inline]
```

Konstruktor dwuargumentowy.

- Dodaje podaną ilość elementów o podanej wartości.

Parametry

value	Wartość do wpisania.
size	Ilość elementów do stworzenia.

4.21.2.4 Vector() [4/4]

```
template<class T >
Vector< T >::  Vector (
              Vector< T > & other )  [inline]
```

Konstruktor kopiujący.

- Tworzy kontener będący kopią podanego.

Parametry

other	Wskaźnik na drugi kontener.

4.21.2.5 ∼Vector()

```
template<class T >
Vector< T >::∼ Vector ( )  [inline]
```

Destruktor

- główna metoda zwalniająca pamięć

4.21.3 Dokumentacja funkcji składowych

4.21.3.1 allowedType()

```
template<class T >
bool  Vector< T >::allowedType ( )  [inline], [virtual]
```

Sprawdzenie czy typ zmiennej jest właściwy

Zwraca

Status operacji: true, jeżeli jest; false jeśli nie.

Implementuje **Sequences< T >** (str. 144).

4.21.3.2 Assign() [1/2]

```
template<class T >
void  Vector< T >::Assign (
            int count,
            const T value )  [inline], [virtual]
```

Zastąpienie starych elementów podaną wartością.

Parametry

count	Ilość elementów do zastąpienia.
value	Nowa wartość.

Implementuje **Sequences< T >** (str. 144).

4.21.3.3 Assign() [2/2]

```
template<class T >
void  Vector< T >::Assign (
            int count,
            T * valuePtr )  [inline], [virtual]
```

Zastąpienie starych elementów podanymi ze wskaźnika/tablicy.

Parametry

count	Ilość elementów do zastąpienia.
valuePtr	Pointer na nowe elementy.

Implementuje **Sequences< T >** (str. 144).

4.21.3.4 At()

```
template<class T >
T&  Vector< T >::At (
            int index )  [inline]
```

Zwrócenie referencji elementu spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Referencja na dany element.

4.21.3.5 AtPtr()

```
template<class T >
T*  Vector< T >::AtPtr (
            int index )  [inline]
```

Zwrócenie wskaźnika na element spod podanego indeksu.

Parametry

index	Indeks podanego elementu.

Zwraca

Wskaźnik na dany element.

4.21.3.6 Back()

```
template<class T >
T&  Vector< T >::Back ( )  [inline]
```

Zwrócenie referencji ostatniego elementu.

Zwraca

Referencja na ostatni element.

4.21.3.7 Begin()

```
template<class T >
T*  Vector< T >::Begin ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na pierwszy elementu.

Zwraca

Wskaźnika na pierwszy element.

4.21.3.8 Capacity()

```
template<class T >
const int  Vector< T >::Capacity ( ) const  [inline], [noexcept]
```

Zwrócenie wielkości pamięci zaalokowanej dla kontenera.

Zwraca

Wielkość zaalokowanej pamięci.

4.21.3.9 Clear()

```
template<class T >
void  Vector< T >::Clear ( )  [inline], [virtual]
```

Zniszczenie zawartości kontenera i ustawienie jego wielkości na 0

Implementuje **Sequences< T >** (str. 145).

4.21.3.10 Emplace()

```
template<class T >
void  Vector< T >::Emplace (
            T value,
            int position )  [inline], [virtual]
```

Wstawienie nowego elementu na podaną pozycje.

Parametry

value	Nowy element.
position	Pozycja, na którą metoda wstawia element.

Implementuje **Sequences< T >** (str. 145).

4.21.3.11 Empty()

```
template<class T >
bool  Vector< T >::Empty ( ) const  [inline], [virtual], [noexcept]
```

Sprawdzenie czy kontener jest pusty.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Sequences< T >** (str. 145).

4.21.3.12 End()

```
template<class T >
T*  Vector< T >::End ( ) const  [inline], [noexcept]
```

Zwrócenie wskaźnika na ostatni elementu.

Zwraca

Wskaźnika na ostatni element.

4.21.3.13 Erase() [1/2]

```
template<class T >
void  Vector< T >::Erase (
            int first,
            int last )  [inline], [virtual]
```

Usunięcie wybranych elementów z przedziału.

Parametry

first	Pierwszy indeks przedziału.
last	Ostatni indeks przedziału.

Implementuje **Sequences**< **T** > (str. 146).

4.21.3.14 Erase() [2/2]

```
template<class T >
void  Vector< T >::Erase (
            int position )  [inline], [virtual]
```

Usunięcie wybranego elementu po pozycji.

Parametry

position	Pozycja elementu do usunięcia.

Implementuje **Sequences**< **T** > (str. 146).

4.21.3.15 findElement()

```
template<class T >
int  Vector< T >::findElement (
            const T value )  [inline]
```

Szukanie elementu po podanej wartości.

Parametry

value	Wartość elementu do odszukania.

Zwraca

Indeks elementu, jeżeli został odnaleziony.

4.21.3.16 findIfExist()

```
template<class T >
bool  Vector< T >::findIfExist (
            const T value )  [inline], [virtual]
```

Sprawdzenie czy element istnieje, szukany po wartości.

Parametry

value	Szukana wartość.

Zwraca

Status operacji: true jeżeli jest pusty, w innym przypadku false.

Implementuje **Sequences< T >** (str. 146).

4.21.3.17 Front()

```
template<class T >
T&  Vector< T >::Front ( )  [inline]
```

Zwrócenie referencji pierwszego elementu.

Zwraca

Referencja na pierwszy element.

4.21.3.18 Insert() [1/3]

```
template<class T >
void  Vector< T >::Insert (
            int position,
            const T & value )  [inline], [virtual]
```

Dodanie elementu przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
value	Nowy element.

Implementuje **Sequences< T >** (str. 147).

4.21.3.19 Insert() [2/3]

```
template<class T >
void  Vector< T >::Insert (
            int position,
            int n,
            const T & value )  [inline], [virtual]
```

Wypełnienie kontenera elementem przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
n	Ilość kopii nowego elementu do wstawienia.
value	Nowy element.

Implementuje **Sequences**< **T** > (str. 147).

4.21.3.20 Insert() [3/3]

```
template<class T >
void  Vector< T >::Insert (
            int position,
            int n,
            T * valuePtr )  [inline], [virtual]
```

Wypełnienie kontenera przedziałem przed podaną pozycją i zwiększenie jego wiekości.

Parametry

position	Pozycja, przed którą metoda wstawia element.
n	Ilość elementów do wstawienia.
valuePtr	Pointer na nowe elementy.

Implementuje **Sequences**< **T** > (str. 148).

4.21.3.21 MaxSize()

```
template<class T >
int  Vector< T >::MaxSize ( )  [inline], [virtual]
```

Zwrócenie maksymlanej ilości elementów jakie jest w w stanie zmieścić kontener

Zwraca

Wielkość maksymalna kontenera.

Implementuje **Sequences**< **T** > (str. 148).

4.21.3.22 operator"!=()

```
template<class T >
bool  Vector< T >::operator!= (
            const  Vector< T > & value ) const  [inline]
```

Operator przeciążony porównania "!="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli nie są równe; false jeśli są.

4.21.3.23 operator=()

```
template<class T >
Vector<T>&  Vector< T >::operator= (
            const  Vector< T > & value )  [inline]
```

Operator przeciążony przypisania "="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Przypisany kontener.

4.21.3.24 operator==()

```
template<class T >
bool  Vector< T >::operator== (
            const  Vector< T > & value ) const  [inline]
```

Operator przeciążony porównania "=="

Parametry

value	Wskaźnik na drugi kontener.

Zwraca

Status operacji: true, jeżeli są równe; false jeśli nie.

4.21.3.25 operator[]()

```
template<class T >
T  Vector< T >::operator[] (
            const int index )  [inline]
```

Operator przeciążony indeksowy "[ ]"

Parametry

index	Indeks elementu.

Zwraca

Wartość elementu spod podanego indeksu.

4.21.3.26 PopBack()

```
template<class T >
void  Vector< T >::PopBack ( )  [inline], [virtual]
```

Usunięcie ostatniego elementu i zmniejszenie wielkości o jeden.

Implementuje **Sequences< T >** (str. 149).

4.21.3.27 PopFront()

```
template<class T >
void  Vector< T >::PopFront ( )  [inline], [virtual]
```

Usunięcie pierwszego elementu i zmniejszenie wielkości o jeden.

Implementuje **Sequences< T >** (str. 149).

4.21.3.28 PushBack()

```
template<class T >
void  Vector< T >::PushBack (
            const T & value )  [inline], [virtual]
```

Dodanie nowego elementu na końcu kontenera, po ostatnimm oraz zwiększenie wielkości o jeden.

Parametry

value	Nowy element.

Implementuje **Sequences< T >** (str. 149).

4.21.3.29 PushFront()

```
template<class T >
void  Vector< T >::PushFront (
            const T & value )  [inline], [virtual]
```

Dodanie nowego elementu na początku kontenera, po ostatnimm oraz zwiększenie wielkości o jeden.

Parametry

value	Nowy element.

Implementuje **Sequences< T >** (str. 150).

4.21.3.30 Reserve()

```
template<class T >
void  Vector< T >::Reserve (
            int n )  [inline]
```

Zarezerwowanie pamięci dla podanej ilości elementów.

Parametry

n	Ilość pamięci do zarezerwowania.

4.21.3.31 Resize() [1/2]

```
template<class T >
void  Vector< T >::Resize (
            int n )  [inline], [virtual]
```

Zmiana wielkości kontenera na podaną.

- Mniejsza niż aktualna: ucięcie kontenera.
- Większa niż aktualna: nowe elementy są puste.

Parametry

n	Nowa wielkość kontenera.

Implementuje **Sequences**< **T** > (str. 150).

4.21.3.32 Resize() [2/2]

```
template<class T >
void  Vector< T >::Resize (
            int n,
            const T & value )  [inline], [virtual]
```

Zmiana wielkości kontenera na podaną i dodanie nowych elementów.

- Mniejsza niż aktualna: ucięcie kontenera.
- Większa niż aktualna: nowe elementy mają podaną wartość.

Parametry

n	Nowa wielkość kontenera.
value	Wartość nowych elementów.

Implementuje **Sequences**< **T** > (str. 150).

4.21.3.33 returnEmpty()

```
template<class T >
T  Vector< T >::returnEmpty ( )  [inline], [virtual]
```

Zwrócenie właściwego wyzerowania dla danego typu

Zwraca

"Zero" dla danego typu zmiennych.

Implementuje **Sequences**< **T** > (str. 151).

4.21.3.34 Reverse()

```
template<class T >
void  Vector< T >::Reverse ( )  [inline], [virtual], [noexcept]
```

Odwrócenie elementów kontenera kolejnością.

Implementuje **Sequences< T >** (str. 151).

4.21.3.35 Show()

```
template<class T >
void  Vector< T >::Show ( )  [inline], [virtual]
```

Wyświetlenie wszystkich elementów kontenera.

Implementuje **Sequences< T >** (str. 151).

4.21.3.36 showType()

```
template<class T >
string  Vector< T >::showType ( )  [inline], [private]
```

Informacja o typie danych w kontenerze.

Zwraca

String z nazwą typu danych.

4.21.3.37 ShrinkToFit()

```
template<class T >
void  Vector< T >::ShrinkToFit ( )  [inline]
```

Redukcja ilości zajętej pamięci, by móc przechowywać tylko niepuste elementy.

4.21.3.38 Size()

```
template<class T >
int  Vector< T >::Size ( ) const  [inline], [virtual], [noexcept]
```

Zwrócenie wielkości kontenera.

Zwraca

Wartość spod zmiennej size.

Implementuje **Sequences< T >** (str. 151).

4.21.3.39 Sort()

```
template<class T >
void  Vector< T >::Sort ( )  [inline], [virtual]
```

Sortowanie elementów w kontenerze

Implementuje **Sequences< T >** (str. 152).

4.21.3.40 Swap()

```
template<class T >
void  Vector< T >::Swap (
              Vector< T > & tmp )  [inline]
```

Zamiana zawartości między dwoma kontenerami.

Parametry

tmp	Drugi kontener.

4.21.4 Dokumentacja atrybutów składowych

4.21.4.1 capacity

```
template<class T >
int  Vector< T >::capacity  [private]
```

4.21.4.2 data

```
template<class T >
T*  Vector< T >::data  [private]
```

4.21.4.3 size

```
template<class T >
int  Vector< T >::size  [private]
```

Dokumentacja dla tej klasy została wygenerowana z pliku:

- C:/Users/olasz/source/repos/Containers/Containers/ **Vector.h**

Rozdział 5

Dokumentacja plików

5.1 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Adaptors.h

```
#include <iostream>
#include <limits>
#include <regex>
#include <iterator>
#include <exception>
#include <sstream>
#include "Containers.h"
```

Komponenty

- class **Adaptors**< **T** >

5.2 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Associative.h

```
#include <iostream>
#include <limits>
#include <regex>
#include <iterator>
#include <exception>
#include <sstream>
#include "Containers.h"
```

Komponenty

- class **Associative**< **T** >

5.3 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Containers.h

```
#include <iostream>
#include <limits>
#include <regex>
#include <iterator>
#include <exception>
#include <sstream>
```

Komponenty

- class **Container**< **T** >

5.4 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/HashTable.h

```
#include <iostream>
#include <limits>
#include <regex>
#include <iterator>
#include <exception>
#include <sstream>
#include <cstdint>
#include <functional>
#include "Containers.h"
#include "Nudes.h"
```

Komponenty

- class **HashTable**< **T** >

5.5 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/List.h

```
#include <iostream>
#include <exception>
#include <sstream>
#include "Sequences.h"
#include "Nudes.h"
```

Komponenty

- class **List**< **T** >

5.6 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/ListBi.h

```
#include <iostream>
#include "Sequences.h"
#include "Nudes.h"
```

Komponenty

- class **ListBi**< **T** >

5.7 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/ListCircle.h

```
#include <iostream>
#include "Sequences.h"
#include "Nudes.h"
```

Komponenty

- class **ListCircle**< **T** >

5.8 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/main.cpp

```
#include <iostream>
#include <cstdint>
#include <functional>
#include <string>
#include "HashTable.h"
#include "ListCircle.h"
#include "MultiMap.h"
#include "MultiSet.h"
#include "PriorityQueue.h"
#include "Stack.h"
#include "Vector.h"
#include "Tree.h"
```

Funkcje

- int **main** ()

5.8.1 Dokumentacja funkcji

5.8.1.1 main()

```
int main ( )
```

- TREE */

5.9 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Map.h

```
#include <iostream>
#include "Associative.h"
```

Komponenty

- class **Map**< **T, V** >

5.10 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/MultiMap.h

```
#include <iostream>
#include "Associative.h"
```

Komponenty

- class **MultiMap**< **T, V** >

5.11 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/MultiSet.h

```
#include <iostream>
#include "Associative.h"
```

Komponenty

- class **MultiSet**< **T** >

5.12 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Nudes.h

```
#include <iostream>
```

Komponenty

- class **ListNode**< **T** >
- class **ListBiNode**< **T** >
- class **HashTableNode**< **T** >
- class **TreeNode**< **T** >

5.13 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/↩ Containers/PriorityQueue.h

```
#include <iostream>
#include "Adaptors.h"
```

Komponenty

- class **PriorityQueue**< **T** >

5.14 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Queue.h

```
#include <iostream>
#include <limits>
#include <regex>
#include <iterator>
#include <exception>
#include <sstream>
#include "Adaptors.h"
```

Komponenty

- class **Queue**< **T** >

5.15 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Sequences.h

```
#include <iostream>
#include <limits>
#include <regex>
#include <iterator>
#include <exception>
#include <sstream>
#include "Containers.h"
```

Komponenty

- class **Sequences**< **T** >

5.16 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Set.h

```
#include <iostream>
#include "Associative.h"
```

Komponenty

- class **Set**< **T** >

5.17 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Stack.h

```
#include <iostream>
#include "Adaptors.h"
```

Komponenty

- class **Stack**< **T** >

5.18 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Tree.h

```
#include <iostream>
#include <limits>
#include <regex>
#include <iterator>
#include <exception>
#include <sstream>
#include "Containers.h"
#include "Nudes.h"
```

Komponenty

- class **Tree**< **T** >

5.19 Dokumentacja pliku C:/Users/olasz/source/repos/Containers/Containers/Vector.h

```
#include <iostream>
#include <limits>
#include <regex>
#include <iterator>
#include <exception>
#include <sstream>
#include "Sequences.h"
```

Komponenty

- class **Vector**< **T** >

Indeks