C++17 i STL

Kolekcje

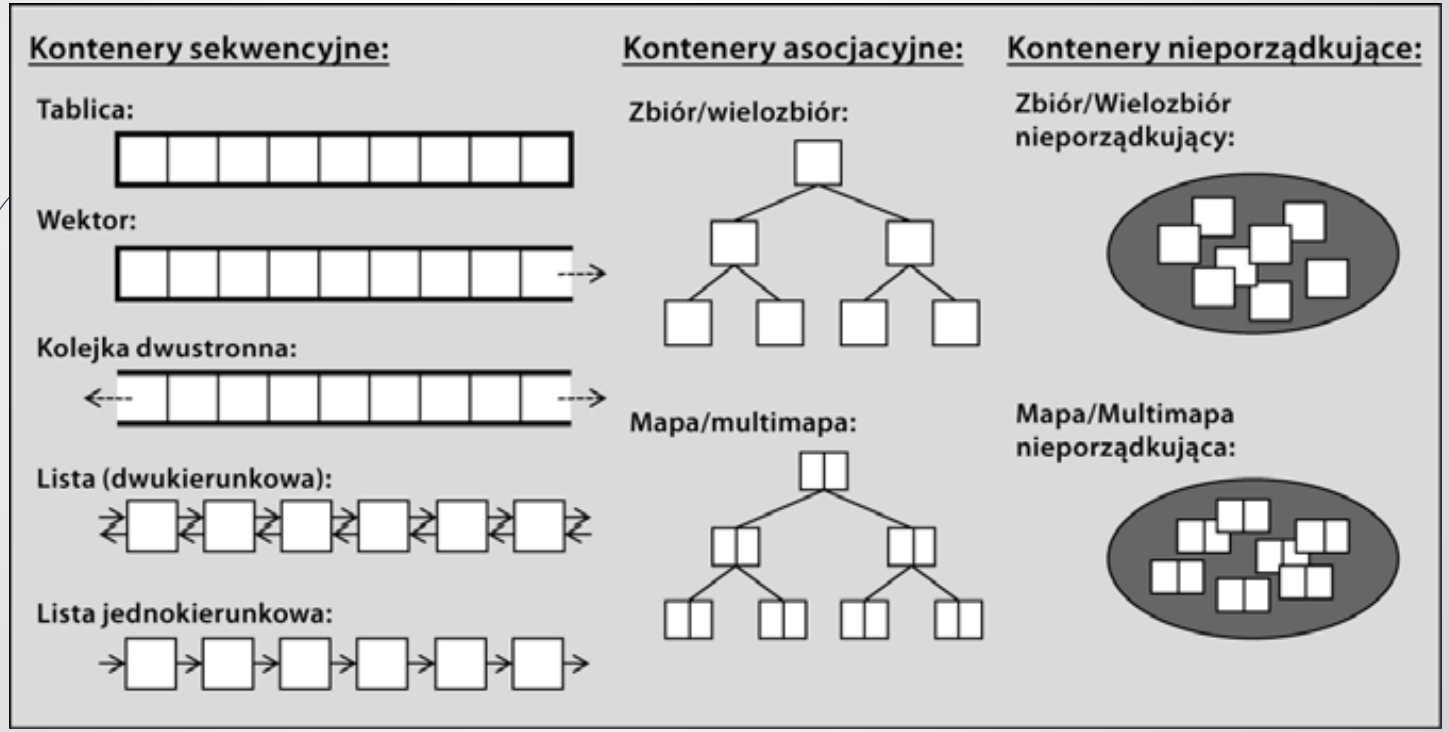
Kontenery

- **Kontenery** to obiekty, które zarządzają kolekcjami elementów:
  - **Kontenery sekwencyjne** będące kolekcjami uporządkowanymi, w których każdy element posiada określoną pozycję. Pozycja ta zależy od momentu oraz miejsca wstawienia, jest jednak niezależna od wartości elementu.
  - **Kontenery asocjacyjne** będące kolekcjami sortowanymi, w których aktualna pozycja elementu zależy od jego wartości (albo klucza w przypadku kontenerów operujących na parach klucz-wartość), zgodnie z określonym kryterium sortowania.
  - **Kontenery nieporządkujące** to kolekcje nieporządkujące i niezachowujące pozycji elementów, bo ich zadaniem głównym jest ustalanie, czy (a nie gdzie) element znajduje się w kolekcji. Elementy nie zachowują więc uporządkowania ani względem kolejności wstawiania, ani względem wartości – jedno i drugie może w czasie życia kontenera ulegać zmianie.

Kontenery

- Kontenery posiadają różne implementacje:
  - kontenery sekwencyjne są zazwyczaj implementowane jako tablice dynamiczne albo listy;
  - kontenery asocjacyjne są zazwyczaj implementowane jako zrównoważone drzewa binarnych poszukiwań (drzewa czerwono-czarne);
  - kontenery nieporządkujące są zazwyczaj implementowane jako tablice z haszowaniem.

Kontenery


Kontenery sekwencyjne:
Tablica:
Wektor:
Kolejka dwustronna:
Lista (dwukierunkowa):
Lista jednokierunkowa:
Kontenery asocjacyjne:
Zbiór/wielozbiór:
Mapa/multimapa:
Kontenery nieporządkujące:
Zbiór/Wielozbiór nieporządkujący:
Mapa/Multimapa nieporządkująca:

Cechy kontenerów

- Wszystkie **kontenery zapewniają semantykę wartości**, a nie semantykę referencji.
  - Podczas operacji wstawiania kontenery, zamiast operować na referencjach do elementów, wykonują wewnętrznie ich kopie.
  - Obiekty, które chcemy umieścić w kontenerze, powinny posiadać publiczny konstruktor kopiujący i publiczny operator przypisania kopiującego.
- Wszystkie **elementy kontenera posiadają określoną kolejność**. Każdy typ kontenerowy udostępnia operacje zwracające iteratory, służące do iteracji po elementach kolekcji.
  - Możemy wykonywać wielokrotnie iteracje po elementach kolekcji i będą one udostępniane w tej samej kolejności.
- **Operacje na kontenerach nie są bezpieczne**, czyli nie sprawdzają możliwości wystąpienia każdego rodzaju błędów.
  - Funkcja wywołująca operację na kontenerze musi zapewnić spełnienie określonych wymagań przez parametry tej operacji – naruszenie tych wymagań (na przykład użycie niepoprawnego indeksu) prowadzi do niezdefiniowanego zachowania.

Cechy elementów kontenerów

- Element musi być kopiowalny albo przenaszalny. Typ elementu musi więc niejawnie albo jawnie udostępniać **konstruktor kopiujący** albo **konstruktor przenoszący**.
  - Utworzona kopia powinna być równoważna elementowi źródłowemu. Oznacza to, że dowolny test równości powinien wykazać, że obydwa elementy są równe oraz że zarówno źródło, jak i kopia zachowują się identycznie.
- Element musi być przypisywalny przez **przypisanie kopiujące** albo **przypisanie przenoszące**.
  - Kontenery i algorytmy wykorzystują operatory przypisania do nadpisywania starych elementów nowymi.
- Element musi być zniszczalny przez **destruktor**.
  - Kontenery niszczą swoje wewnętrzne kopie elementów, gdy elementy te zostają usunięte z kontenera. Destruktor nie może więc być prywatny. Destruktor nie może również zgłaszać wyjątków.

Wspólne operacje na kontenerach

- Standard definiuje zestaw operacji wspólnych dla wszystkich kontenerów (z powodu różnorodności kontenerów dostępnych w C++11 zdarzają się wyjątki i niektóre kontenery nie spełniają wszystkich ogólnych wymogów):
  - konstruktor domyślny,
  - konstruktor kopiujący i przypisanie kopiujące,
  - konstruktor przenoszący i przypisanie przenoszące,
  - (opcjonalnie) konstruktor inicjalizowany kopiami elementów z podanego zakresu w innej kolekcji i przypisanie,
  - (opcjonalnie) konstruktor inicjalizowany kopiami elementów z listy wartości przekazanych za pomocą initializer_list<>,
  - destruktor publiczny, który usuwa wszystkie elementy kolekcji i zwalnia pamięć (o ile to jest możliwe),

Wspólne operacje na kontenerach

- Standard definiuje zestaw operacji wspólnych dla wszystkich kontenerów (z powodu różnorodności kontenerów dostępnych w C++11 zdarzają się wyjątki i niektóre kontenery nie spełniają wszystkich ogólnych wymogów):
  - funkcja składowa empty(), która sprawdza czy kontener jest pusty,
  - (opcjonalnie) funkcja składowa clear(), która usuwa wszystkie elementy z kontenera,
  - funkcje składowe size() i max_size(), które zwracają odpowiednio bieżącą i maksymalną liczbę elementów w kontenerze,
  - operatory == i != ustalające odpowiednio równość i nierówność kontenerów,
  - (opcjonalnie) operatory <, <=, > i >=, które ustalają relację między kontenerami,
  - funkcja składowa i statyczna swap(), która zamienia zawartość kontenerów,
  - funkcje składowe begin() i end() oraz cbegin() i cend(), które dostarczają iteratorów pracujących na kolekcji.

Inicjalizacja

- const std::vector<int> v1 = { 1, 2, 3, 5, 7, 11, 13, 17, 21 };
- std::list<int> l;
  …
  std::vector<float> c(l.begin(),l.end());
- std::list<std::string> l;
  …
  std::vector<std::string> c(
  std::make_move_iterator(l.begin()),
  std::make_move_iterator(l.end())
  );
- int carray[] = { 2, 3, 17, 33, 45, 77 };
  …
  std::set<int> c(std::begin(carray),std::end(carray));
- std::deque<int> c{
  std::istream_iterator<int>(std::cin),
  std::istream_iterator<int>()
  };

Przypisania i zamiany

- Przypisanie kontenera oznacza skopiowanie wszystkich elementów kontenera źródłowego i jednocześnie usunięcie wszystkich starych elementów kontenera docelowego.
- Po przypisaniu przenoszącym kontener po lewej stronie przypisania zawiera elementy, które wcześniej posiadał kontener znajdujący się po prawej stronie przypisania. Zawartość kontenera po prawej stronie po wykonaniu operacji jest niezdefiniowana.
- Funkcja składowa swap() zamienia zawartość dwóch kontenerów. W rzeczywistości zamianie ulegają jedynie wewnętrzne wskaźniki na dane.

Operacje dotyczące rozmiaru

- Funkcja **empty()** informuje o zerowej liczbie elementów (begin()==end()). Warto ją stosować zamiast konstrukcji size()==0. Funkcja empty() może być zaimplementowana bardziej wydajnie niż size().
- Funkcja **size()** zwraca aktualną liczbę elementów kontenera. Operacji tej nie zapewnia się dla list typu forward_list<>, gdyż nie miałaby stałej złożoności.
- Funkcja **max_size()** zwraca maksymalną liczbę elementów, które może zawierać kontener. Wartość ta jest różnie zdefiniowana w zależności od implementacji. Funkcja max_size() zwraca zwykle maksymalną wartość reprezentowaną przez typ indeksu.

Porównania

- Jeśli nie liczyć kontenerów nieuporządkowanych, zwykle operatory porównania ==, !=, <, <=, > oraz >= zdefiniowane są według następujących trzech reguł:
  - Obydwa kontenery muszą być tego samego typu.
  - Dwa kontenery są równe, jeśli ich elementy są równe i posiadają tę samą kolejność.
  - W celu sprawdzenia, czy jeden kontener jest mniejszy od innego kontenera, przeprowadzane jest porównanie leksykograficzne.

Dostęp do elementów

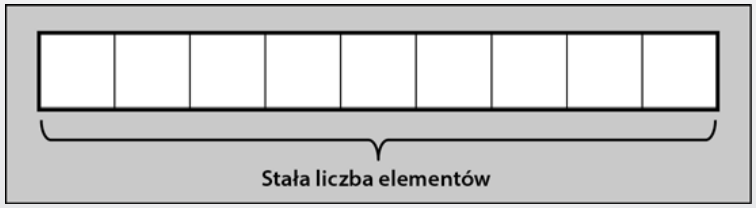
- Wszystkie kontenery udostępniają interfejs iteratora, co umożliwia stosowanie pętli for bazujących na zakresach wartości:
  - for (const auto& elem : coll) {
    std::cout << elem << std::endl;
    }
  - for (auto& elem : coll) {
    elem = ...;
    }
- By korzystać z danych o pozycjach (na przykład by móc wstawiać, usuwać lub przenosić elementy), można po prostu użyć iteratorów udostępnianych przez cbegin() i cend():
  - for (auto pos=coll.cbegin(); pos!=coll.cend(); ++pos) {
    std::cout << *pos << std::endl;
    }
  - for (auto pos=coll.begin(); pos!=coll.end(); ++pos) {
    *pos = ...;
    }

Definicje typów w kontenerach

- Typ size_type jest typem całkowitym bez znaku dla wartości rozmiaru.
- Typ value_type to typ elementów w kontenerze.
- Typ reference to typ referencji do elementu.
- Typ const_ reference to typ referencji do niemodyfikowalnego elementu.
- Typ iterator to typ iteratora.
- Typ const_iterator to typ iteratora, który nie modyfikuje kolekcji.
- Typ pointer to typ wskaźnika do elementu.
- Typ const_pointer to typ wskaźnika niemodyfikującego do elementu.

Tablice

- Kontener **array<>** modeluje tablicę statyczną – jest to otoczka dla statycznej tablicy z języka C, zapewniająca interfejs kontenera STL.
- Tablica to ciąg elementów o ustalonej długości – nie można więc dodawać ani usuwać elementów i w ten sposób zmieniać jej rozmiaru; możliwe jest tylko i wyłącznie zastępowanie elementów.



Tablice

- Tablice kopiują elementy do własnych wewnętrznych, statycznych tablic.
- Tablice są kolekcją uporządkowaną, ponieważ elementy tablic są zawsze ułożone w ścisłej kolejności.
- Tablice zapewniają swobodny dostęp do elementów – dostęp do każdego elementu tablicy jest bezpośredni i ma stały czas.
- Kontener array<> to jedyny, którego elementy są inicjalizowane domyślnie, jeśli nic nie zostanie przekazane jawnie – oznacza to, że dla typów podstawowych wartość początkowa może być niezdefiniowana, zamiast otrzymać wartość 0.

```
std::array<int,4> x;  // elementy x są niezdefiniowane
std::array<int,4> x = {};  // wszystkie elementy x mają wartość 0
std::array<int,5> coll = {42, 377, 611, 21, 44};
     // wskazania wartości początkowych tworzonej tablicy
std::array<int,10> c2 = {42};
     // pierwszy element ma wartość 42 pozostałe 0
```

Tablice

- Kontener array<> udostępnia operacje swap() – można zamienić elementy kontenera tego samego typu (taki sam typ elementów i taka sama liczba elementów), trzeba jednak pamiętać, że array<> nie może tak naprawdę podmienić wewnętrznych wskaźników (z tego powodu swap() ma złożoność liniową, a iteratory i referencje nie zamieniają kontenerów razem z wartościami).
- Poza operatorem przypisania możliwe jest tylko użycie operacji fill() do przypisania nowej wartości wszystkim elementom lub swap() do zamiany wartości z inną tablicą; w przypadku operatora = lub funkcji swap() obie tablice muszą być tego samego typu.
- Kontrolę zakresu robi to tylko funkcja at() – jeśli indeks nie mieści się w zakresie, zgłasza ona wyjątek out_of_range. Pozostałe funkcje nie wykonują tego sprawdzenia. Wywołanie operatora [] oraz funkcji front() i back() wobec pustego kontenera array<> powoduje zawsze niezdefiniowane zachowanie.

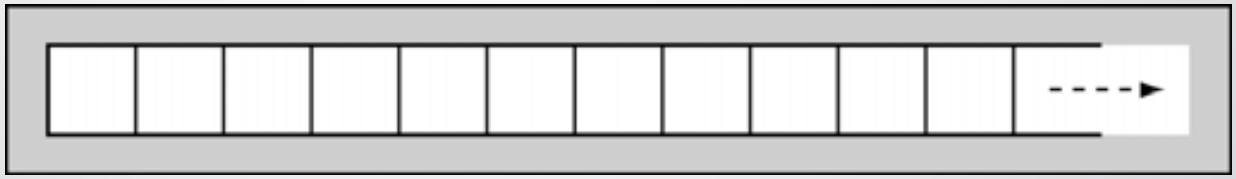
Tablice

- Tablice zapewniają interfejs krotki – oznacza to, że można użyć wyrażeń w stylu tuple_size<>::value do uzyskania liczby elementów, tuple_element<>::type do uzyskania typu konkretnego elementu i get() do uzyskania wartości konkretnego elementu.
- Oto przykład:

```
typedef std::array<std::string,5> FiveStrings;
FiveStrings a = { "witaj", "janie", "jak", "się", "masz" };
   std::tuple_size<FiveStrings>::value // zwraca 5
   std::tuple_element<1,FiveStrings>::type // zwraca std::string
   std::get<1>(a) // zwraca std::string("janie")
```

Wektory

- Kontener **vector<>** jest modelem tablicy dynamicznej – jest on abstrakcją, której elementy zarządzane są przy użyciu dynamicznej tablicy w stylu języka C.
- Pojemność wektorów nigdy nie się zmniejsza, istnieje więc gwarancja, że referencje, wskaźniki oraz iteratory pozostaną ważne nawet w przypadku usuwania elementów, pod warunkiem że odnoszą się one do pozycji występującej przed modyfikowanymi elementami.



Wektory

- Wektory kopiują swoje elementy do wewnętrznej tablicy dynamicznej. Elementy te zawsze posiadają określoną kolejność. Wektory są więc swego rodzaju kolekcją uporządkowaną.
- Wektory umożliwiają dostęp bezpośredni. Możemy więc bezpośrednio odwołać się do każdego elementu w stałym czasie, pod warunkiem że znamy jego pozycję.
- Iteratory wektora są iteratorami dostępu swobodnego, możemy więc użyć każdego algorytmu z biblioteki STL.
- Wektory zapewniają dobrą wydajność, jeśli dołączamy lub usuwamy elementy na ich końcu. W przypadku wstawiania lub usuwania elementów w środku lub na początku wektora wydajność spada.

Wektor bitów

- Dla elementów boolowskich wektora biblioteka STL udostępnia specjalizację kontenera vector<bool>, która przeznacza wewnętrznie zwykle tylko 1 bit na każdy element.
- Klasa vector<bool> jest czymś więcej niż tylko specjalizacją – klasa vector<> dla typu bool udostępnia kilka specjalnych operacji bitowych, które umożliwiają operowanie na bitach lub znacznikach w wygodniejszy i szybszy sposób.

Kolejki o dwóch końcach

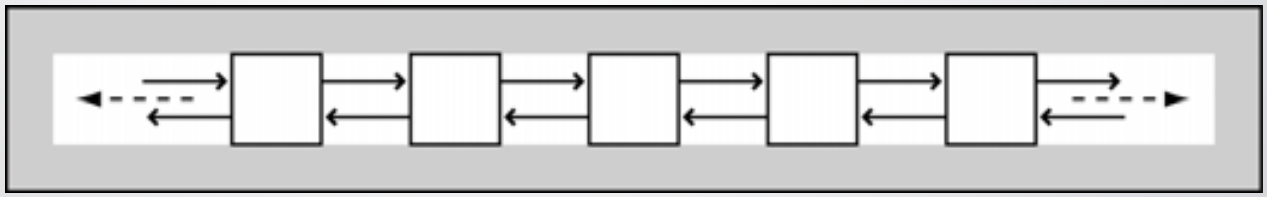
- Kontener **deque<>**, czyli kolejka o dwóch końcach, jest bardzo podobna do wektora – zarządza swoimi elementami, wykorzystując tablicę dynamiczną, zapewnia dostęp swobodny oraz posiada prawie taki sam interfejs co wektor; różnica polega na tym, że w przypadku kolejki deque<> tablica dynamiczna jest otwarta z obydwu końców.

Kolejki o dwóch końcach

- Wstawanie i usuwanie elementów jest szybkie zarówno na początku, jak i na końcu kontenera.
- Kolejki deque<> nie oferują możliwości sterowania pojemnością ani momentem realokacji.
- Bloki pamięci mogą zostać zwolnione w przypadku, gdy nie są już wykorzystywane, tak więc rozmiar obszaru pamięci zajmowanego przez kolejkę deque może się zmniejszyć.

Listy

- W liście list<> elementy zorganizowane są w postaci listy dwukierunkowej.
- Obiekt listy zawiera dwa wskaźniki nazywane też zakotwiczeniami, które wskazują pierwszy i ostatni element listy. Każdy element zawiera wskaźniki do następnego lub poprzedniego elementu (lub do zakotwiczenia). Wstawienie czy usunięcie elementu wymaga jedynie zmian odpowiednich wskaźników.

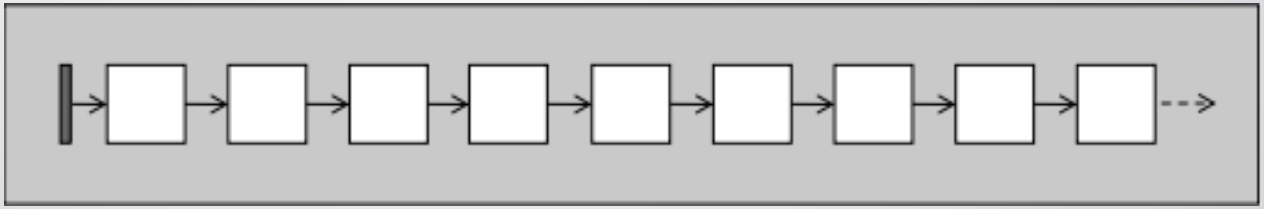


Listy

- Lista nie zapewnia dostępu swobodnego.
- Wstawianie i usuwanie elementów jest szybkie na każdej pozycji (o ile ma się już do niej dostęp).
- Operacje wstawiania i usuwania elementów nie powodują unieważnienia wskaźników, referencji ani iteratorów do innych elementów.
- Listy zapewniają funkcje front(), push_front() i pop_front(), a także back(), push_back() i pop_back().

Listy jednokierunkowe

- Lista jednokierunkowa forward_list<> została wprowadzona w C++11 do zarządza elementami jako lista związana z pojedynczymi wskaźnikami.
- Koncepcyjnie lista jednokierunkowa przypomina standardową listę, ale zubożoną w ten sposób, iż nie jest w stanie iterować w tył.
- Dla wszystkich funkcji składowych modyfikujących listę w taki sposób, że elementy są wstawiane lub usuwane na konkretnych pozycjach, lista jednokierunkowa oferuje ich specjalne wersje – powód jest bardzo prosty, bo trzeba przekazać położenie elementu przed pierwszym modyfikowanym elementem, ponieważ właśnie temu elementowi trzeba przypisać następnika.

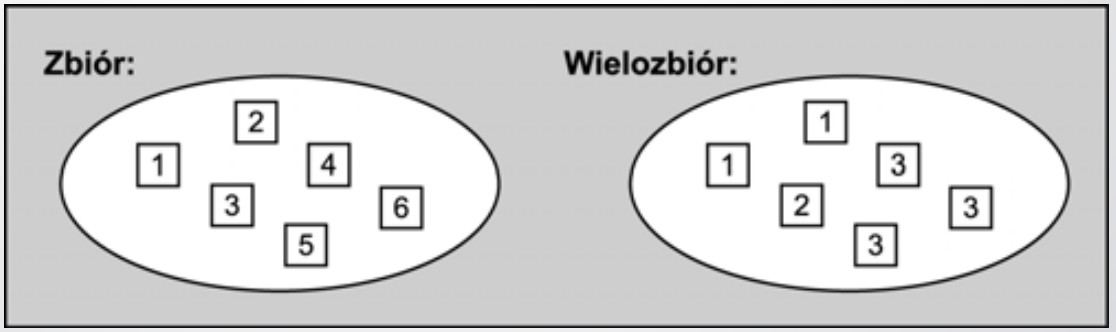


Inne kontenery STL

- Jako kontenerów *STL* możemy używać łańcuchów lub zwykłych tablic albo też sami zdefiniować specjalne kontenery zaspokajające specyficzne potrzeby. Takie postępowanie posiada tę zaletę, że możemy wykorzystać algorytmy, takie jak sortowanie czy scalanie, wobec własnego typu.
- Łańcuchy traktować można jako kontenery znaków. Znaki występujące wewnątrz łańcucha tworzą ciąg, po którym możemy iterować w celu przetwarzania poszczególnych znaków. Standardowe klasy łańcuchowe zapewniają zatem interfejs kontenerowy STL. Udostępniają one funkcje składowe begin() oraz end(), które zwracają iteratory dostępu swobodnego umożliwiające iterację po elementach łańcucha.
- Zwykłe tablice nie są jednak klasami, a więc nie udostępniają one funkcji składowych, takich jak begin() czy end(). Nie możemy również definiować dla nich funkcji składowych. W takiej sytuacji musi zostać zastosowane podejście nieinwazyjne lub osłonowe.

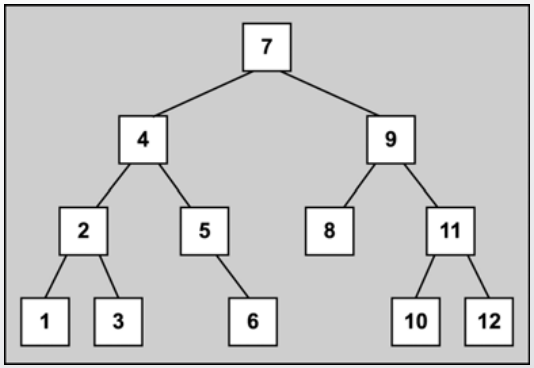
Zbiory i wielozbiory

- Kontenery set<> oraz multiset<> wykonują automatyczne sortowanie swoich elementów zgodnie z określonym kryterium sortowania.
- Różnica pomiędzy nimi polega na tym, że wielozbiory dopuszczają powtórzenia elementów, podczas gdy zbiory tego nie dopuszczają.



Zbiory i wielozbiory

- Elementy zbioru lub wielozbioru mogą być dowolnego typu, który umożliwia operacje przypisania, kopiowania i porównania zgodnie z kryterium sortowania (opcjonalny drugi parametr szablonu definiuje kryterium sortowania).
- Wszystkie asocjacyjne klasy kontenerowe, zbiory i wielozbiory są implementowane jako zrównoważone drzewa BST (drzewa czerwono-czarne).



Zbiory i wielozbiory

- Największą zaletą automatycznego sortowaniaw zbiorach i wielozbiorach jest wysoka wydajność drzew binarnych przy wyszukiwaniu elementów o określonej wartości (złożoność logarytmiczna).
- Automatyczne sortowanie nakłada jednak na zbiory i wielozbiory także istotne ograniczenie – nie można bezpośrednio zmienić wartości elementu, ponieważ mogłoby to zaburzyć prawidłową kolejność.
- Zbiory i wielozbiory nie udostępniają operacji realizujących bezpośredni dostęp do elementów.