Kontenery specjalne

Wykład 14

Kontenery specjalne

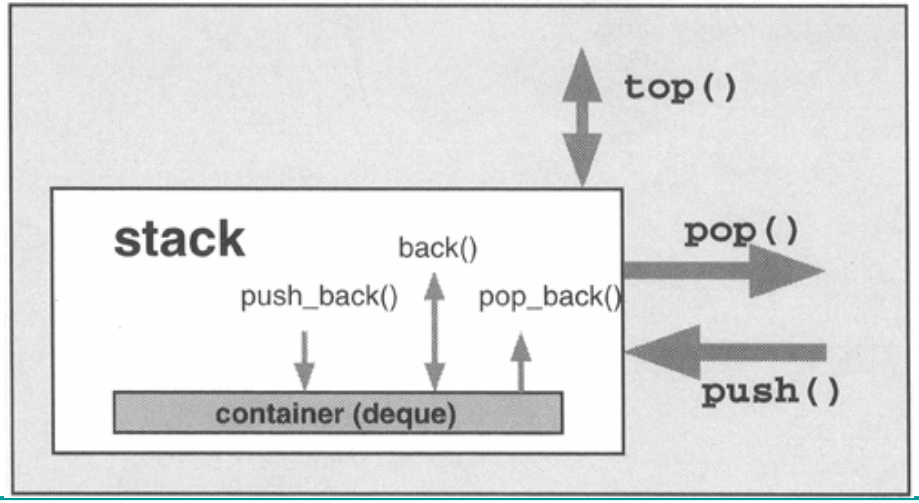
- Adaptatory kontenerów – przystosowują standardowe kontenery do specjalnych celów
  - Stosy
  - Kolejki
  - Kolejki priorytetowe
- Kontener specjalny `bitset`
  - Jest polem bitowym o ściśle określonym, ale dowolnym rozmiarze
  - Dla zmiennego rozmiaru przystosowana jest specjalizowana wersja wektora `vector<bool>`

Stosy

- Stos nazywany jest inaczej kolejką LIFO
- Pobieranie elementów następuje w kolejności odwrotnej do ich umieszczania
- Stos jest w szablonie klasy **`stack`** i umieszczonym w nagłówku **`<stack>`**
- Domyślnie zaimplementowany jako kolejka dwustronna **`deque`**
  - Użyta dlatego, że automatycznie zwalnia pamięć oraz nie wymaga przekopiowywania elementów przy realokacji pamięci

Interfejs stosu

- Funkcje składowe
  - `push()` – umieszcza element na stosie
  - `top()` – zwraca kolejny element stosu
  - `pop()` – usuwa element ze stosu
- Stos może być oparty na kontenerze, który udostępnia następujące metody
  - `back()`, `push_back()`, `pop_back()`
- Przykład cpp_14.1

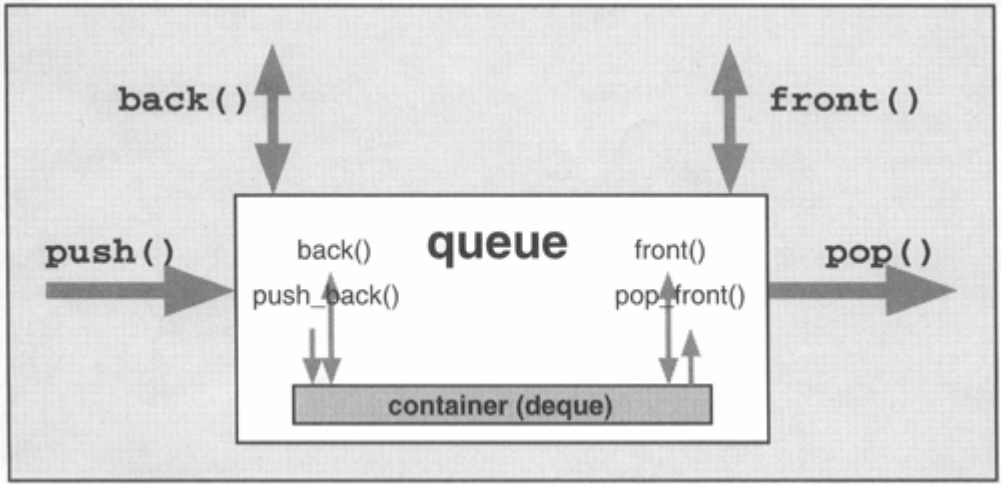


Kolejki

- Kolejka występuje również pod nazwą kolejki FIFO
- Pobieranie elementów z kolejki następuję zgodnie z kolejnością ich wstawiania
- Szablon klasy implementujący kolejkę to **queue** znajdujący się w nagłówku o tej samej nazwie
- Domyślnie kolejka implementowana jest za pomocą kolejki dwustronnej **deque**

Interfejs kolejki

- Funkcje składowe
  - `push()` – umieszcza element w kolejce
  - `pop()` – usuwa element z kolejki
  - `front()` – zwraca kolejny element
  - `back()` – zwraca ostatni element
- Kolejka może być oparta na kontenerze, który udostępnia następujące metody
  - `back()`, `push_back()`, `front()`, `pop_front()`
- Przykład cpp_14.2

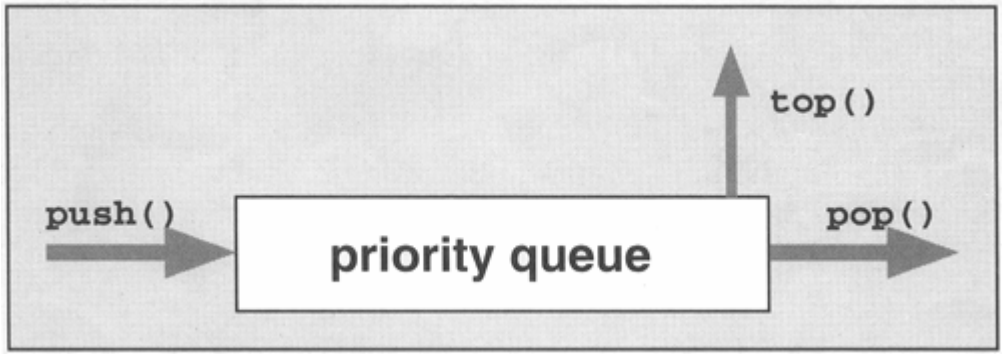


Kolejki priorytetowe

- Szablon klasy **`priority_queue`** implementuje kolejkę priorytetową
- Elementy kolejki priorytetowej sortowane są zgodnie z kryterium sortowania (domyślenie **`less`**)
  - Dlatego kolejny element w kolejce ma największą wartość
- Domyślnym kontenerem do przechowywania kolejek priorytetowych jest wektor

Interfejs kolejki priorytetowej

- Funkcje składowe
  - `push()` – umieszcza element w kolejce
  - `pop()` – usuwa element z kolejki
  - `top()` – zwraca kolejny element
- Kolejka priorytetowa może być oparta na kontenerze, który udostępnia iteratory dostępu swobodnego oraz następujące metody
  - `push_back()`, `front()`, `pop_front()`
- Przykład cpp_14.3



Kontener `bitset`

- Kontenery **`bitset`** są tablicami o ustalonym rozmiarze, zawierającymi bity lub wartości logiczne
- Są przeznaczone do zarządzania znacznikami
- Posiada zalety w stosunku do przechowywanie bitów w liczbie **`long`**
  - Może przechowywać dowolna liczbę bitów, zawiera również dodatkowe operatory
- Szablon klasy **`bitset`** zdefiniowany jest w nagłówku **`<bitset>`**
  - Parametrem szablony jest nie typ, a liczba bitów

Funkcje składowe niemodyfikujące

- `bitset(unsigned long val)` - tworzy kontener bitów i inicjalizuje zgodnie z bitami zmiennej `val`
- `bitset(const string& s)`, `bitset(const string& s, idx)`, `bitset(const string& s, idx, num)` - tworzy i incjalizuje kontener bitów łańcuchem, lub częścią łańcucha
- `c.size()` - zwraca liczbę bitów
- `c.count()` - zwraca liczbę ustawionych bitów
- `c.any()` - zwraca wartość logiczną czy ustawiony został jakikolwiek bit
- `c.none()` - zwraca wartość logiczną czy nie został ustawiony żaden bit
- `c.test(idx)` - zwraca wartość logiczną czy ustawiony został bit na pozycji `idx`
- `c.operator==` - porównuje czy są równe
- `c.operator!=` - porównuje czy są różne

Funkcje składowe modyfikujące

- `c.set()` – ustawia wszystkie bity na 1, zwraca zmodyfikowany zestaw bitów
- `c.set(idx)` – ustawia bit na pozycji `idx`, zwraca zmodyfikowany zestaw bitów
- `c.set(idx, val)` – ustawia bit na pozycji `idx` na `val`, zwraca zmodyfikowany zestaw bitów
- `c.reset()` – ustawia wszystkie bity na 0, zwraca zmodyfikowany zestaw bitów
- `c.reset(idx)` – zeruje bit na pozycji `idx`, zwraca zmodyfikowany zestaw bitów
- `c.flip()` – zamienia wartości wszystkich bitów, zwraca zmodyfikowany zestaw bitów
- `c.operator^=` - wykonuje `XOR`, zwraca zmodyfikowany zestaw bitów
- `c.operator|=` - wykonuje `OR`, zwraca zmodyfikowany zestaw bitów
- `c.operator&=` - wykonuje `AND`, zwraca zmodyfikowany zestaw bitów
- `c.operator<<=` - przesuwa w lewo, zwraca zmodyfikowany zestaw bitów
- `c.operator>>=` - przesuwa w prawo, zwraca zmodyfikowany zestaw bitów
- Zdefiniowane są również powyższe operatory bez `=`
- `c.operator[]` – zwracający referencję do bitu w niestałego kontenerze lub wartość logiczną

Konwersja typów

- `to_ulong()` - zwraca wartość `unsigned long` reprezentowana przez bity
- `to_string()` - zwraca `string` reprezentujący poszczególne bity
- Operatory we/wy
  - `operator>>` - wczytuje ze strumienia do zmiennej reprezentację bitów
  - `operator<<` - zapisuje do strumienia reprezentację bitów w postaci `string`
- Przykład cpp_14.4

Łańcuchy (string-i)

- Klasy określające łańcuch znakowe umożliwiają wykorzystanie ich tak jak normalnych typów danych
  - Można kopiować, porównywać przypisywać itp.
  - Nie trzeba pamiętać o przydzielaniu pamięci
- Wszystkie typy oraz funkcje dotyczące łańcuchów znakowych zdefiniowano w pliku nagłówkowym **<string>**
- Łańcuchy zdefiniowane są w szablonie **basic_string**
  - STL definiuje dodatkowo specjalizowane wersje **string** oraz **wstring**
  - **typedef basic_string<char> string;**
  - **typedef basic_string<wchar_t> wstring;**
- Łańcuchy mają duże możliwości, ale niestety nie obsługują pewnych operacji
  - Wyrażeń regularnych
  - Przetwarzanie tekstu – stosowanie wersalików, porównań łańcuchów bez znaczenia na wielkość liter

Konstruktory

- `string s` - tworzy pusty łańcuch znakowy
- `string s(str)` - tworzy łańcuch znakowy będący kopią `str`
- `string s(str, idx)` - tworzy łańcuch znakowy będący kopią `str` od indeksu `idx`
- `string s(str, idx, len)` - tworzy łańcuch znakowy będący kopią `str` od indeksu `idx` o długości co najwyżej `len`
- `string s(cstr)` - tworzy łańcuch znakowy będący kopią łańcucha języka C `cstr`
- `string s(chars, len)` - tworzy łańcuch znakowy inicjalizowany `len` elementami tablicy znakowej `chars`
- `string s(num, c)` - tworzy łańcuch znakowy inicjalizowany `n` znakami `c`
- `string s(beg, end)` - tworzy łańcuch znakowy inicjalizowany elementami z zakresu `[beg, end)`
- Przykład cpp_14.5

Iteratory i indeksy

- Metody dostarczone przez łańcuch współpracują z indeksami i iteratorami
- Iteratory używamy tak jak dla wektorów
- Zamiast iteratorów możemy korzystać z indeksów
  - Indeksy powinny być typu **`size_type`**
  - Przy przeszukiwaniu zwracana jest specjalna wartość **`string::npos`**
  - Problem wynika z tego, że wartość **`string::npos`** wynosi -1, a indeksy są typu całkowitego bez znaku

Konwersja łańcucha do postaci tablicy znakowej lub łańcucha znakowego języka C

- `s.data()` – zwraca zawartość stringu w postaci tablicy znaków, nie dodaje na końcu znaku '\0'
- `s.c_str()` – zwraca zawartość stringu w postaci łańcucha znakowego języka C, znak '\0' jest dołączany
- `s.copy(buff, n)` – kopiuje co najwyżej `n` znaków do tablicy znakowej `buff`
- `s.copy(buff, n, pos)` – kopiuje co najwyżej `n` znaków od pozycji `pos`, do tablicy znakowej `buff`
- Przykład cpp_14.6

Metody

- Metody związane z pojemnością stringu są takie same jak dla wektora
  - Np. `s.capacity()`, `s.reserve()`
- Dostęp do elementów, porównanie oraz przypisania są znowu podobne do tych znanych z wektora
  - Np. `operator[]`, `s.assign(aString, 1, 3);`
- Wycinanie fragmentów
  - `s.substr(idx)` - zwraca łańcuch od pozycji `idx` do końca
  - `s.substr(idx, len)` - zwraca łańcuch od pozycji `idx` o długości `len`
- Konkatenacja
  - `operator+`, `operator+=`
- Operatory `<<` i `>>`
  - Zdefiniowane tak jak dla łańcuchów języka C
- Przykład cpp_14.7

Wstawianie i usuwanie znaków

- `s.clear()`, `s.erase()` - usuwa wszystkie znaki z łańcucha (`s = ""`)
- `s.append(str)`, `s.push_back(c)` - dołączają na koniec łańcucha drugi łańcuch lub znak
  - `append(str)`, `append(str, 1, 3)`, `append("lancuchC")`, `append(5, 'x')`,
- `s.insert(idx, ...)` - działa podobnie jak `append` tylko wstawia na pozycję `idx` łańcuchy (ale nie znak)
- `s.erase(idx)` - usuwa od pozycji `idx` do końca
  `s.erase(idx, len)` - usuwa od pozycji `idx`, `len` znaków
- `s.replace(idx, len, str)` - zastępuje znaki od pozycji `idx` o długości `len` znakami z `str`
  `s.replace(idx, len, str, stridx, strlen)` - zamienia co najwyżej `len` znaków począwszy od `idx` znakami z `str` od pozycji `stridx` o długości `strlen`
- Przykład cpp_14.8

Przeszukiwanie

- `s.find(str)` – poszukuje od początku znaku łańcucha `str`, ewentualnie można podać pozycję indeksu i jego długość od którego wyszukiwać
- `s.rfind(str)` – jw. tylko od końca
- `s.find_first_of(str)` – poszukuje pierwszego wystąpienia znaku będącego elementem `str`, można podać również `index`
- `s.find_first_not_of(str)` – jw. tylko nie występującego w `s`
- `s.find_last_of(str)` – jw. tylko od końca
- `s.find_last_not_of(str)` – jw. tylko od końca
- Wszystkie zwracają indeks lub wartość `string::npos` w przypadku niepowodzenia
- Przykład cpp_14.9 i cpp_14.10