Politechnika Śląska w Gliwicach

Zaawansowane biblioteki programistyczne
15 stycznia 2017

Trwałe drzewo poszukiwań binarnych

Autorzy:
Barbuletis Jakub
Forczmański Mateusz

Informatyka SSM, semestr II
Rok akademicki 2016/2017
Grupa OS1
Rocznik 2016

"Lepszy wyjątek w garści niż kod, który nie działa."

— Mikołaj Kopernik

1 Wstęp

Naszym zadaniem projektowym z przedmiotu *Zaawansowane biblioteki programistyczne* było napisanie klasy bibliotecznej PersistentTree reprezentującej trwałe drzewo poszukiwań binarnych. Ta klasa miała udostępniać interfejs dla programistów podobny do tego, jaki posiada klasa std::set ze standardowej biblioteki szablonów C++. Miał być on jak najbardziej praktyczny i posiadać wszystkie niezbędne funkcjonalności związane z tworzeniem drzewa i nie udostępniać jego pośrednich, niekompletnych wersji. Program został napisany w język C++. Poza bibliotekami zawartymi w standardzie C++11, nie zostały wykorzystane żadne biblioteki zewnętrzne. Do zapewnienia trwałości struktury został wykorzystany algorytm Sleatora, Tarjana i innych.

2 Specyfikacja zewnętrzna

2.1 Deklaracja

Główną klasą przechowującą trwałe drzewo poszukiwań binarnych jest PersistentTree. Nagłówek klasy wygląda następująco:

```
1 template<class Type, class OrderFunctor = std::less<Type>>
2 class PersistentTree;
```

Parametry szablonowe:

- Type - typ przechowywanych danych w drzewie,
- OrderFunctor - funkcja porządku definiująca sposób wstawiania elementów do drzewa. Domyślną wartością jest typ std::less<Type>.

Z powyższych dwóch punktów wynika, że typ szablonowy powinien umożliwiać operacje na operatorach relacyjnych, jak < lub >.

2.2 Konstruktory

Listing 1: Konstruktor bezparametrowy

```
1 PersistentTree();
```

Tworzy puste drzewo bez żadnych węzłów, z pierwszym numerem wersji.

Listing 2: Konstruktor z iteratorem

```
1 template <class Iter>
2 PersistentTree(Iter begin, Iter end);
```

Konstruktor przyjmujący początek i koniec iteratora kolekcji. Tworzy trwałe drzewo wstawiając jako korzeń pierwszy element kolekcji, dodając kolejne, aż wstawi ostatni z nich. Numer wersji tak utworzonego drzewa jest równy liczbie elementów kolekcji. Jeżeli kolekcja jest pusta, efekt jest taki sam jak przy użyciu konstruktora bezparametrowego.

- Iter - klasa iteratora,
- begin - początek kolekcji wskazany przez iterator,
- end - koniec kolekcji wskazany przez iterator.

2.3 Destruktor

Listing 3: Destruktor

```
1 ~PersistentTree();
```

Zwalnia z pamięci wszystkie zaalokowane przez drzewo węzły.

2.4 Metody

Klasa PersistentTree udostępnia następujący zestaw zewnętrznych instrukcji.

Listing 4: begin

```
1 iterator begin(int version = CURRENT_VERSION);
```

Funkcja zwracająca iterator na pierwszy (skrajnie lewy) węzeł drzewa poszukiwań. Przyjmuje pojedynczy parametr *version*, który decyduje o wersji drzewa na które wskazuje iterator. Domyślną wartością jest wersja aktualna - w przypadku nie podania parametru, iterator wskaże na najnowszą wersję drzewa.

Listing 5: clear

```
1 void clear();
```

Usuwa z drzewa wszystkie węzły i zapisuje ten stan jako nową wersję drzewa - utrwalona historia pozostaje nienaruszona.

Listing 6: end

```
1 iterator end() const;
```

Zwraca iterator, który wskazuje na miejsce za kolekcją wszystkich węzłów. Nie przyjmuje parametru wskazującego na wersję drzewa - dla wszystkich wartość funkcji end pozostaje taka sama.

Listing 7: erase

```
1 bool erase(Type value);
```

Usuwa z drzewa węzeł o wskazanej wartości i zapisuje ten stan jako nową wersję drzewa. Jeżeli element istnieje i operacja zakończyła się sukcesem - funkcja zwraca wartość true. Jeżeli element w drzewie nie istnieje, wówczas zwrócona zostaje wartość false.

Listing 8: find

```
1 iterator find(Type value, int version = CURRENT_VERSION) const;
```

Funkcja wyszukująca węzeł o wskazanej wartości (parametr *value*) i zwracająca iterator do niego. Jeżeli wartość nie istnieje w drzewie, wówczas iterator przyjmie wartość poza kolekcją węzłów (patrz funkcja end 6). Parametr *version* wskazuje w której wersji drzewa należy wyszukiwać wartości. Domyślną wartością tego parametru jest najnowsza wersja drzewa.

Listing 9: getCopy

```
1 PersistentTree<Type> * getCopy(int version);
```

Zwraca kopię drzewa o wskazanej wersji w parametrze *version*. Kopia nie posiada historii - zawiera dokładnie taką samą strukturę drzewa jak oryginał w podanej wersji, ale w kopii otrzymuje ona wersję pierwszą, nie znajduje się nic poza nią.

Listing 10: getCurrentVersion

```
1 int getCurrentVersion() const;
```

Zwraca aktualną wersję drzewa w formie liczbowej.

Listing 11: insert

```
1 bool insert(Type value);
```

Wstawia do drzewa nowy węzeł o wartości wskazanej w parametrze *value* i zapisuje ten stan jako nową wersję. Funkcja zwraca true jeżeli wartość została prawidłowo wstawiona i operacja zakończyła się sukcesem. Wartość false jest zwracana gdy element nie został wstawiony do drzewa, np. dlatego, że owa wartość już istnieje w drzewie (w najnowszej wersji).

Listing 12: print

```
1 void print(int version = CURRENT_VERSION) const;
```

Wypisuje całe drzewo w sformatowanej formie na standardowym wyjściu. Parametr *version* określą, którą wersję drzewa należy wydrukować. Domyślnie przyjmuje najnowszą wersję drzewa.

Listing 13: purge

```
1 void purge();
```

Całkowicie likwiduje drzewo wraz z całą jego historią. Wszystkie węzły zostają usunięte, a puste drzewo zostaje oznaczone jako pierwsza wersja.

Listing 14: size

```
1 int size(int version = CURRENT_VERSION) const;
```

Zwraca rozmiar drzewa reprezentowany jako liczba wszystkich jego węzłów. Parametr *version* określa wersję drzewa, którego rozmiar należy zwrócić, domyślnie przyjmuje, że ma to być wersja najnowsza.

3 Specyfikacja wewnętrzna

3.1 Reprezentacja drzewa

Reprezentacja została rozłożona na dwie klasy: PersistentTree oraz Node. Pierwsza reprezentuje trwałe drzewo poszukiwań binarnych, a druga pojedynczy węzeł ów drzewa.

3.1.1 Klasa węzła

Node przechowuje informacje jakie są potrzebne do działania drzewa poszukiwań: wartość oraz wskaźniki na lewego i prawego potomka, a także pola potrzebne do działania zgodnie z algorytmem Sleatora, Tarjana i innych: rodzaj, czas i wartość zmiany. Ponieważ zmianą może być albo zmiana wartości w węźle, albo zmiana wskaźnika na jednego z potomków, został wykorzystany typ union do reprezentacji wartości zmiany: przyjmuje on albo wartość prostego wskaźnika, albo całą nową wartość węzła. Do reprezentacji typu zmiany został wykorzystany typ enum. Klasa Node udostępnia na zewnątrz gettery do wartości drzewa: potomków oraz wartości, które jako parametr przyjmują numer wersji. W każdym z nich sprawdzany jest rodzaj i czas zmiany, i jeżeli są one odpowiednie, zwracana jest zmieniony element, a nie ten przechowywany w polach węzła drzewa.

3.1.2 Klasa drzewa

Głównym zadaniem klasy PersistentTree od wewnętrznej strony jest takie zarządzanie węzłami i relacjami między nimi, aby po operacjach *insert* i *erase* wszystkie zmiany zostały właściwie spropagowane. Klasa posiada wektor wskaźników na korzenie drzewa wraz z czasem (wersją) ich utworzenia, które są punktami wejściowymi wszystkich operacji. Nowy wskaźnik jest dodawany do wektora tylko gdy po propagacji zmian zostanie utworzony nowy obiekt korzenia. Ponieważ reprezentacja węzła posiada tylko wskaźniki na potomków, drzewo ma szereg pomocniczych funkcji, m. in. do wyszukiwania rodzica wskazanego węzła we wskazanej wersji.

3.2 Propagacja zmian

Za każdym razem, gdy zostaje wykonana operacja wstawiania lub usuwania w drzewie, może ruszyć cała fala zmian, jakie muszą być wykonane w strukturze drzewa. Jeżeli węzeł, w którym dokonywana jest zmiana, posiada już jakąś, zostaje utworzona jego kopia i jest uruchomiany algorytm propagacji zmian.

Algorithm 1 Algorytm propagacji zmian po utworzeniu kopii węzła

```
procedure PROPAGUJ ZMIANY(Pierwszy rodzic, Pierwsze dziecko)
    Aktualny rodzic = Pierwszy rodzic
    Aktualne dziecko = Pierwsze dziecko
    while Nie koniec propagacji do
        if Aktualny rodzic jest nullem then
            Aktualne dziecko jest korzeniem dla najnowszej wersji
            Koniec obliczeń
        else
            if Aktualny rodzic nie ma zmiany then
                Ustaw zmianę w rodzicu
                Koniec obliczeń
            else
                Utwórz kopię rodzica z uwzględnioną zmianą
                Aktualne dziecko = Aktualny rodzic
                Aktualny rodzic = Rodzic aktualnego rodzica
            end if
        end if
    end while
end procedure
```

3.3 Alokacja pamięci dla węzłów

W celu rezerwowania pamięci dla węzłów drzewa została utworzona klasa NodeAllocator, która odpowiada za tworzenie i usuwanie węzłów. Za każdym razem gdy drzewo potrzebuje nowego węzła, np. podczas wstawiania nowej wartości lub kopiowania w czasie propagacji zmian, używa w tym celu obiektu alokatora. Klasa NodeAllocator udostępnia na zewnątrz metody, które rezerwują pamięć funkcją malloc i wywołują konstruktor węzła oraz takie, które wywołują destruktor i zwalniają pamięć funkcją free. Klasa ta przechowuje również informację o łącznej liczbie zaalokowanych węzłów oraz rozmiarze przydzielonej pamięci w bajtach.

3.4 Iterator

Klasa PersistentTree<Type> udostępnia jako swój iterator klasę PersistentTreeIterator<Type> (pod nazwą "iterator" odwołując się z przestrzeni nazw). Implementuje ona iterator typu *forward* dla trwałego drzewa poszukiwań. Jej deklaracja wygląda następująco.

Listing 15: PersistentTreeIterator

```
1 template<class Type, class UnqualifiedType = std::remove_cv_t<Type>>
2 class PersistentTreeIterator : public std::iterator<std::
      forward_iterator_tag, UnqualifiedType, std::ptrdiff_t, Type*, Type&>;
```

Tworząc obiekt iteratora należy przekazać mu w parametrze konstruktora wskaźnik na węzeł, od którego będzie się zaczynać iteracja oraz wersję drzewa po której będzie iterował. Obiekt potrafi przejść przez wszystkie kolejne elementy zgodnie z porządkiem wskazanym przez OrderFunctor, aż do końca. Aby umożliwić poprawne przechodzenie przez wszystkie elementy, została wykorzystana struktura stosu. Iterator udostępnia na zewnątrz operatory porównywania oraz inkrementacji, w celu wykorzystania go w podstawowych pętlach, gdzie ważny jest warunek stopu oraz pojedynczy krok do przodu.

3.5 Funkcja porządku

W parametrze szablonowym można przekazać klasę funktora, która będzie definiować porządek węzłów w drzewie. Aby to było możliwe, funktor musi posiadać przeciążony operator wywołania, który przyjmuje jako parametry dwie stałe referencje do obiektów typu. Ten operator jest wykorzystywany w wielu metodach drzewa, m. in. wstawiania, wyszukiwania, usuwania i propagacji. Dobrze nadają się do tego obiekty klas z biblioteki *functional*, jak std::less, std::greater itp.

4 Scenariusze testowe

Naszym zadaniem było wykonanie testów czasowych i pamięciowych dla typów int oraz std :: string, a także porównanie ich z wynikami takich samych testów dla kontenera std :: set.

4.1 Testy czasowe

Czasy wstawiania i usuwania dla kontenerów std :: set oraz PersistentTree:

- Miliona losowych wartości typu int.
- Ponad miliona wartości typu std :: string pobranych ze słownika języka polskiego.

Ponadto zostały przeprowadzone testy wyszukiwania 100 tysięcy losowo wybranych elementów ze struktur opartych o oba rodzaje danych.
Przed wstawianiem i usuwaniem porządek elementów został ustalony losowo metodą std :: shuffle. Testy zostały przeprowadzone w trybie *release*, w programie skompilowanym przez MCVS14 na architekturę 64-bitową oraz uruchomionym na procesorze Intel Celeron G530.

Tabela 1: Testy czasowe

Czasy obliczeń [s]	**std::set**			**PersistentTree**		
Typ danych	*insert*	*find*	*erase*	*insert*	*find*	*erase*
int	1.109	0.114	1.467	2.276	0.287	4.398
std::string	2.679	0.165	3.549	5.4837	0.439	11.287

Tabela 2: Testy pamięciowe

Pamięć [MB]	**std::set**	**PersistentTree**
int	3.81	6.06
std::string	41.67	66.20

5 Podsumowanie