
**POLITECHNIKA ŚLĄSKA**

**WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI**

# ZAAWANSOWANE BIBLIOTEKI PROGRAMISTYCZNE

Sprawozdanie z projektu semestralnego „Drzepiec”

| | |
|---|---|
| Autor | Łukasz Proksa |
| Prowadzący | prof. dr hab. Sebastian Deorowicz |
| Rok akademicki | 2017 / 2018 |
| Kierunek | Informatyka |
| Rodzaj studiów | SSM |
| Semestr | 2 |
| Specjalność | OS |

Gliwice, luty 2018

# Spis treści

# 1. Treść zadania

Zaimplementować strukturę danych drzepiec (ang. *treap*) wzorując się na implementacji kontenera `std::set` dostępnej w standardowej bibliotece szablonów STL języka C++.

# 2. Analiza zadania

Drzepiec jest strukturą danych łączącą w sobie drzewo binarne i kopiec (stąd nazwa drze-piec). Posiada więc cechy drzewa przeszukiwań binarnych BST [1]:

- wartość lewego dziecka jest mniejsza od wartości rodzica
- wartość prawego dziecka jest większa od wartości rodzica

W którym każdy węzeł został wzbogacony o atrybut „priorytet", spełniający własność kopca typu max [2]:

- priorytet dziecka jest nie większy od priorytetu rodzica

Wzbogacenie BST o priorytet zapobiega np. degenerowaniu się drzewa do listy i zapewnia optymalną wysokość drzewa rzędu $O(log\ n)$, o ile priorytet nowego węzła jest wybierany losowo [3].

# 3. Algorytmy i struktury danych

Formalnie drzepiec jest ukorzenionym drzewem binarnym, w którym z każdym węzłem skojarzone są następujące informacje [3]:

- key[x] – klucz (inaczej wartość) węzła x
- left[x] – lewe dziecko węzła x
- right[x] – prawe dziecko węzła x
- parent[x] – rodzic węzła x
- priority[x] – priorytet węzła x

i każdy węzeł, nie będący liściem, spełnia następujące zależności:

- key[x] > key[left[x]]
- key[x] < key[right[x]]
- priority[x] >= priority[left[x]]
- priority[x] <= priority[right [x]]

Zastanówmy się jak będzie wyglądał algorytm wstawiania nowego węzła do drzewa. Czy może on być identyczny jak w drzewie BST? Nie, ponieważ po wstawieniu nowego węzła do drzewa, może zostać naruszona własność kopca typu max. Stanowi on jednak dobry punkt wyjścia. Zmodyfikujmy go więc tak, by po wstawieniu nowego węzła przywracał własność kopca typu max zachowując jednocześnie porządek inorder. Osiągniemy to wykonując rotacje na drzewie, które zmieniają jego strukturę, ale nie zmieniają kolejności inorder.

## 3.1 Pseudokod operacji Insert

Przyjrzyjmy się psudokodowi algorytmu wstawiania nowego węzła do drzewa poniżej [3].

```
insert(k, p, T)
1.    key[x] ← k
2.    priority[x] ← p
3.    insert-bst(x, T)
4.    while x != root[T] do
5.      if x == left[parent[x]] then
```

```
6.      if priority[x] < priority[parent[x]] then
7.        break
8.      else
9.          rotate-right(x, T)
10.   else
11.     if priority[x] < priority[parent[x]] then
12.       break
13.     else
14.       rotate-left(x, T)
```

W liniach 1-2 inicjujemy wartość i priorytet węzła. W linii 3 wstawiamy węzeł do drzewa BST. W liniach 4-14 kopcujemy względem priorytetu, by przywrócić własność kopca typu max.

Pozostałe algorytmy zostały skomentowane w kodzie źródłowym programu.

## 4. Specyfikacja zewnętrzna

Aby przetestować implementację drzepca, został napisany program mierzący czasy wykonania wybranych metod (kod źródłowy znajduje się w dodatku A). Program został skompilowany na systemie Linux poleceniem:

```
g++ -c -std=c++11 -O2 treap.cpp
```

i uruchamiany jest z wiersza poleceń poleceniem:

```
./treap < input.txt
```

Przykładowe dane wejściowe znajdują się w dodatku B.

## 5. Specyfikacja wewnętrzna

Autor starał się jak najdokładniej wzorować na implementacji kontenera `std::set` biblioteki STL ze standardu sprzed C++11. Kod został umieszczony w przestrzeni nazw `pols1`, najważniejsze fragmenty znajdują się na listingach poniżej.

### 5.1 Podstawa węzła

Dzięki rozdzieleniu węzła na 2 struktury, podstawa węzła nie zawiera pola „wartość” i nie wymaga użycia szablonu. Niektórym algorytmom wystarcza wskaźnik na podstawę węzła, jeśli nie wykorzystują wartości węzła.

```
struct _Treap_node_base {
        int _M_priority;
        _Base_ptr _M_left;
        _Base_ptr _M_right;
        _Base_ptr _M_parent;
}
```

### 5.2 Węzeł

Dodaje pole „wartość”. Metoda `_M_valptr()` jest wykorzystywana zamiast `operator&()` do pobrania wskaźnika na wartość, na wypadek gdyby `operator&()` był przeciążony przez użytkownika.

```
template <typename _Val>
struct _Treap_node : public _Treap_node_base {

        _Val*
        _M_valptr() {
                return std::__addressof(_M_value_field);
        }
```

```
		_Val _M_value_field;
}
```

### 5.3 Iterator

Definicje wymagane przez koncept *BidirectionalIterator*, niżej wskaźnik na aktualny węzeł. Stały iterator od modyfikowalnego iteratora, różni się tym, że przechowuje stały wskaźnik na aktualny węzeł i posiada jeden konstruktor więcej (pozwalający z modyfikowalnego iteratora utworzyć stały).

```
struct _Treap_iterator {
		typedef _Val value_type;
		typedef _Val& reference;
		typedef _Val* pointer;

		typedef std::bidirectional_iterator_tag iterator_category;
		typedef ptrdiff_t difference_type;

		typedef _Treap_iterator<_Val> _Self;
		typedef _Treap_node_base::_Base_ptr _Base_ptr;
		typedef _Treap_node<_Val>* _Link_type;

		_Base_ptr _M_node;
}
```

### 5.4 Generator priorytetów

Generator liczb pseudolosowych Mersenne Twister zaimplementowany jako singleton.

```
class _Treap_priority_generator {
		static std::mt19937 mt;
	public:

		static unsigned int get() {
			return mt();
		}
	};
```

### 5.5 Drzepiec

Dla tego konceptu STL wymaga modyfikowalnego iteratora, jednak to umożliwia modyfikowanie kluczy (co jest zabronione). Dlatego `iterator` i `reverse_terator` są stałym iteratorem, natomiast na wewnętrzny użytek implementacji zdefiniowano modyfikowalne `_iterator` i `_reverse_terator` (poprzedzone twardą spacją) nieudostępnione publicznie. STL rozwiązuje ten problem definiując dwie struktury `set` i `set_impl`. Pierwsza z nich deklaruje tylko interfejs dla użytkownika, prawdziwe implementacje metod znajdują się w drugiej.

```
template <typename _Key, typename _Compare = std::less<_Key>, typename _Alloc =
std::allocator<_Key> >
class treap {
public:
		// concept typedefs
		typedef typename _Alloc::template rebind<_Treap_node<_Key> >::other
_Node_allocator;

		typedef _Key _Val;
		typedef _Key key_type;
		typedef _Key value_type;
		typedef _Compare key_compare;
		typedef _Compare value_compare;
		typedef _Alloc allocator_type;

		typedef value_type* pointer;
```

```
		typedef const value_type* const_pointer;
		typedef value_type& reference;
		typedef const value_type& const_reference;

		typedef size_t size_type;
		typedef ptrdiff_t difference_type;

		// stl requires modifable iterator, but this allows modification on keys
		// define another real modifable _iterator and _reverse_iterator
		typedef _Treap_const_iterator<value_type> iterator;
		typedef _Treap_const_iterator<value_type> const_iterator;
		typedef std::reverse_iterator<iterator> reverse_iterator;
		typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

	protected:
		typedef _Treap_iterator<value_type> _iterator;
		typedef std::reverse_iterator<_iterator> _reverse_iterator;

		// functional object defining order relation between elements
		key_compare _M_key_cmp;
		// header node pointing to leftmost and rightmost element to allow
		constant time retrieving begin() and end() iterator
		_Treap_node_base _M_header;
		// keeps track on node count
		size_type _M_node_count;
		// client can pass custom allocator as template argument
		_Node_allocator _M_node_allocator;
}
```

### 5.6 Implementacja operacji Insert

Implementacja operacji Insert według wcześniejszego pseudokodu:

```
void
_Treap_insert_and_heapify(const bool __insert_left, _Treap_node_base* __x,
_Treap_node_base* __p, _Treap_node_base& __header) {

	_Treap_node_base*& __root = __header._M_parent;

	// initialize fields of new node
	__x->_M_parent = __p;
	__x->_M_left = 0;
	__x->_M_right = 0;
	__x->_M_priority = _Treap_priority_generator::get();

	// Insert
	// make new node child of parent and maintain root, leftmost and rightmost nodes
	// first node is always inserted left
	if (__insert_left) {
		__p->_M_left = __x; // also makes leftmost = __x when __p == &__header
		if (__p == &__header) {
			// x is the first node
			__header._M_parent = __x; // maintain root
			__header._M_right = __x; // maintain rightmost
		} else if (__p == __header._M_left) {
			// x is new minimum
			__header._M_left = __x; // maintain leftmost
		}
	} else {
		__p->_M_right = __x;
		if (__p == __header._M_right) {
			__header._M_right = __x; // maintain rightmost
		}
	}
	// Heapify
```

```
        while (__x != __root) {
                if (__x == __x->_M_parent->_M_left) {
                        if (__x->_M_priority < __x->_M_parent->_M_priority) {
                                break;
                        } else {
                                _Treap_rotate_right(__x->_M_parent, __root);
                        }
                } else {
                        if (__x->_M_priority < __x->_M_parent->_M_priority) {
                                break;
                        } else {
                                _Treap_rotate_left(__x->_M_parent, __root);
                        }
                }
        }
}
```

## 6. Testowanie i uruchamianie

Początkowo przeprowadzono proste testy poprawnościowe, następnie testy jakościowe. Jakość implementacji pod względem czasu wykonania operacji była mierzona programem z dodatku B. Poniżej czasy wykonania wybranych operacji dla słowników języka angielskiego i języka polskiego.

```
✓ ~/StudiaPS/zbp/treap [master]$ ./dist/Release/GNU-Linux/treap < en.txt
<treap>
czas wstawiania: 43.5199 ms
czas wyszukiwania: 36.3022 ms
czas zliczania: 33.9665 ms
✓ ~/StudiaPS/zbp/treap [master]$ ./dist/Release/GNU-Linux/treap < pl.txt
<treap>
czas wstawiania: 2246.55 ms
czas wyszukiwania: 103.692 ms
czas zliczania: 101.467 ms
```

*Rys. 1 Testy czasowe drzepca*

Dla porównania poniżej zamieszczono czasy wykonania tych samych operacji na tych samych danych wejściowych, gdy zamiast `polsl::treap` użyto `std::set` lub `std::unordered_set`.

```
✓ ~/StudiaPS/zbp/lab1$ ./zad1 < en.txt
<set>
czas wstawiania: 25.1847 ms
czas wyszukiwania: 17.9414 ms
czas zliczania: 17.0507 ms
<unordered_set>
czas wstawiania: 10.5161 ms
czas wyszukiwania: 4.87206 ms
czas zliczania: 6.7629 ms
✓ ~/StudiaPS/zbp/lab1$ ./zad1 < pl.txt
<set>
czas wstawiania: 905.199 ms
czas wyszukiwania: 35.9061 ms
czas zliczania: 35.4013 ms
<unordered_set>
czas wstawiania: 362.802 ms
czas wyszukiwania: 8.81497 ms
czas zliczania: 12.3937 ms
```

*Rys. 2 Testy czasowe set i unordered_set*

## 7. Wnioski

Na podstawie przeprowadzonych testów widzimy, że uzyskana implementacja drzepca jest około 2-3 razy wolniejsza od `std::set` i około 6-9 razy wolniejsza od `std::unordered_set`. Biorąc pod uwagę fakt, że kontenery STL są pisane przez profesjonalistów na przestrzeni wielu lat, otrzymane wyniki uważam za satysfakcjonujące. Ponadto sama specyfika drzepca powoduje pewien narzut czasowy, którego nie da się obejść, a który wynika z konieczność losowania priorytetu podczas wstawiania nowego węzła. W drzewach czerwono - czarnych, na których zaimplementowany jest `std::set`, takiego narzutu nie ma. Udało się również osiągnąć duże podobieństwo kodu źródłowego do istniejących kontenerów biblioteki STL.

## 8. Dodatek A

Kod programu testującego implementację drzepca znajduje się w pliku main.cpp

Kod programu testującego std::set i std::unordered_set znajduje się w pliku zad1.cpp

## 9. Dodatek B

Dane wejściowe użyte do testów ze względu na rozmiar (słowniki języka polskiego i angielskiego), można znaleźć w dołączonych plikach projektu pl.txt i en.txt

## 10. Bibliografia

[1] Drzewa wyszukiwań binarnych. (2007). W C. E. Thomas H. Cormen, *Wprowadzenie do algorytmów* (strony 253-264). WNT.

[2] Heapsort - sortowanie przez kopcowanie. (2007). W C. E. Thomas H. Cormen, *Wprowadzenie do algorytmów* (strony 124-134). WNT.

[3] Karpiński, M. (2010). *Drzewce*. Pobrano z lokalizacji Wrocławski Portal Informatyczny: http://informatyka.wroc.pl/node/787?page=0,0