Zestaw 6

Binarne drzewo poszukiwań

Algorytmy i strukury danych I

> ***Binarne drzewo poszukiwań*** *(ang.* binary search tree*, BST) to kolejna dynamiczna struktura danych. W drzewie binarnym każdy węzeł posiada co najwyżej dwoje* dzieci *(lewe i prawe). Z wyjątkiem* korzenia*, każdy węzeł posiada również* rodzica*. BST cechuje logarytmiczna złożoność podstawowych operacji.*

Własność drzewa BST:

- Dla każdego węzła y znajdującego się w lewym poddrzewie węzła x zachodzi $y_{key} < x_{key}$.
- Dla każdego węzła y znajdującego się w prawym poddrzewie węzła x zachodzi $y_{key} \geq x_{key}$.

Właściwości drzew BST:

- Typowa złożoność podstawowych operacji wynosi $O(\log\ n)$
- Wyznaczenie następnika i poprzednika nie wymaga porównywania kluczy

Szczegółowe informacje o drzewach binarnych można znaleźć w [Cormen, 2013] oraz na stronie.

Zadanie 1. Drzewo binarne (`BinaryTree.hpp`, `BinaryTree.cpp`)

Zaimplementować binarne drzewo poszukiwań przy pomocy wskaźników oraz operacje charakterystyczne dla tej struktury:

- `insert(x)` - wstawia element `x` do drzewa
- `search(x)`, `searchRecursive(x)` - sprawdza czy element należy do drzewa, zwraca wskaźnik do węzła lub `nullptr`. Napisać wersję iteracyjną i rekurencyjną
- `size()` - zwraca liczbę węzłów
- `minimum()` - zwraca wartość najmniejszego elementu
- `maximum()` - zwraca wartość największego elementu
- `depth()` - zwraca wysokość drzewa
- `inorder()`, `preorder()`, `postorder()` - wypisuje zawartość wszystkich węzłów odpowiednio w porządku *inorder*, *preorder* i *postorder*. Wykorzystać rekurencje.

Drzewo ma przechowywać elementy typu `int`.

Dane wejściowe

Dane wejściowe należy wczytać ze standardowego wejścia (`std::cin`).

- W pierwszym wierszu znajduje się liczba całkowita n
- W kolejnych n wierszach znajdują się liczby, które należy wstawić do drzewa BST - zgodnie z jego własnością.

Dane wyjściowe

- W pierwszej linii wypisać liczbę elementów (`size`), wysokość drzewa (`depth`), element minimalny (`minimum`) i element maksymalny (`maximum`).
- W kolejnych n liniach wypisać elementy w porządku preorder (`preorder`).
- W ostatnich 9-ciu liniach dla liczb od 1 do 9 wypisać "Yes", gdy należy ona do drzewa lub "No" w przeciwnym wypadku (`search`).
- Wzorcowe dane wejściowe i wyjściowe można znaleźć na stronie ćwiczeń w pliku `BinaryTreeData.tar.gz`.

Pytania

- Jakie operacje są relatywnie szybko wykonywane na drzewie BST, porównaj złożoności czasowe z innymi strukturami danych.
- Na czym polega iteracyjne przechodzenie drzewa bez użycia rekurencji?
- W jaki sposób można użyć drzewo BST do sortowania elementów? Wskaż podobieństwo do wcześniej poznanego algorytmu sortowania. Omów wady takiego rozwiązania.
- Na czym polega przechodzenie drzewa *preorder*?
- Na czym polega przechodzenie drzewa *inorder*? Jaka będzie kolejność wypisywanych elementów?
- Na czym polega przechodzenie drzewa *postorder*,

Zadania dodatkowe

Zadanie A1. Iteracyjne przechodzenie drzewa (1 pkt)

Napisać program, który **iteracyjnie** (nie **rekurencyjnie**) przechodzi przez elementy drzewa od najmniejszego do największego i je wypisuje.

> ***Wskazówka:*** *Iteracyjne przechodzenie drzewa BST można przyspieszyć używając* ***stosu****. Za każdym razem gdy idziemy na lewo wstawiamy węzeł na stos, a gdy nie ma prawego dziecka zdejmujemy.*
> ***Uwaga:*** *Wyznaczenie następnika i poprzednika nie wymaga porównywania kluczy.*

Zadanie A2. Klasy szablonowe (1pkt)

Napisać klasy szablonowe

```
template<typename T> class BinaryNode;
template<typename T> class BinaryTree;
```

implementujące odpowiednio drzewo BST i jego węzły. Użyć perfekcyjnego przenoszenia (`std::forward`). Aby skrócić zapis można użyć wewnątrz definicji klasy *alias*

```
using node_ptr = binaryNode<T> *;
```

Zadanie A3. Iterator (1 pkt)

Zaimplementować iterator dla drzewa binarnego

```
template<typename T> class BinaryIterator;
```

oraz następujące operatory:

- `++` - następnik (successor), zwraca `*this` czyli obiekt typu `binaryIterator<T>&`
- `--` - poprzednik (predecessor), zwraca `*this` czyli obiekt typu `binaryIterator<T>&`
- `*` - dereferencji, zwraca obiekt typu `T&`
- `==`, `!=` - porównanie

Uwaga: Iteracyjne przechodzenie drzewa BST można przyspieszyć używając stosu. Za każdym razem gdy idziemy na lewo wstawiamy węzeł na stos, gdy nie ma prawego dziecka zdejmiemy.

Implementacja powinna działać poprawnie dla pętli *for-each*:

```
for(const auto& a : lista)
    std::cout << a << std::endl;
```

Zadanie Zamiast `inorder()` wykorzystać pętlę *for-each* do wypisania elementów drzewa. Ponadto napisać testy dla dodatkowych operacji.

Zadanie A4. Operacje dodatkowe (1 pkt)

Oprócz operacji z wersji podstawowej dodatkowo zaimplementować:

- `erase(it)` - usuwa węzeł `it` (iterator) z drzewa
- `hasPathSum(x)` - sprawdza czy w drzewie istnieje droga korzeń-do-liścia dla której suma wartości w węzłach jest równa x
- `sameTree(tree1, tree2)` - sprawdza czy dwa drzewa są identyczne
- `begin(), end(), rbegin(), rend()` - zwraca odpowiedni iterator
- Operacje `insert` i `search` powinny zwracać iterator.

Zadanie Napisać testy dla dodatkowych operacji.

Andrzej Görlich
a.goerlich@outlook.com