Zestaw 3

Lista I

Algorytmy i strukury danych I

Lista

> ***Lista*** *(ang.* list*) to* abstrakcyjny typ danych*, w którym skończona liczba mogących się powtarzać elementów jest ułożona w porządku liniowym. Lista jest jedną z podstawowych reprezentacji zbiorów dynamicznych. Typowe implementacje listy to* implementacja wskaźnikowa *(lista z dowiązaniami) i* implementacja tablicowa.

Proszę zapoznać się z wywiadem *Taste of Linus Torvalds* z twórcą jądra *Linux* oraz systemu kontroli wersji *Git* na temat implementacji listy (14:20-16:20).

Interfejs

```
class List {
    void push_front(int x);     // Dołącza element na początek listy
    int pop_front();            // Usuwa i zwraca element z początku listy
    void push_back(int x);      // Dołącza element na koniec listy
    int pop_back();             // Usuwa i zwraca element z końca listy
    int size();                 // Zwraca liczbę elementów w liście
    bool empty();               // Zwraca true gdy lista jest pusta
    void clear();               // Czyści listę
    int find(int x);            // Zwraca pozycję pierwszego elementu o wartości x
    int erase(int i);           // Usuwa i zwraca element na pozycji i
    void insert(int i, int x);  // Wstawia element x przed pozycję i
    int remove(int x);          // Usuwa wystąpienia x i zwraca ich liczbę
};
```

Uwagi

- Zdefiniować konstruktor tworzący pustą listę.
- Złożoność obliczeniowa operacji powinna być *optymalna* dla danej implementacji.
- Funkcje usuwające elementy, w przypadku gdy nie jest to możliwe, powinny wyrzucać wyjątek.
- Funkcja `find()` zwraca `-1` gdy element nie występuje.

Zadanie 1. Implementacja wskaźnikowa dwukierunkowa listy

Napisać podwójnie wiązaną implementację wskaźnikową listy (`LinkedList.hpp`).

Program `LinkedList.cpp` ma wczytywać dane wejściowe ze standardowego wejścia, wykonać odpowiednie operacje zgodnie z poniższym formatem wykorzystując implementację wskaźnikową listy i wypisać rezultat na standardowe wyjście.

Format danych wejściowych

W pierwszej linii podana jest liczba $n \leq 10^3$ wskazującą na liczbę operacji do wykonania. W kolejnych n liniach znajdują się operacje następującego typu:

- `F x` - wstaw liczbę $0 \leq x \leq 10^3$ na początek listy (ang. *front*)
- `B x` - wstaw liczbę $0 \leq x \leq 10^3$ na koniec listy (ang. *back*)
- `f` - usuń z listy pierwszy element i go wypisz, jeśli lista jest pusta wypisz "EMPTY"
- `b` - usuń z listy ostatni element i go wypisz, jeśli lista jest pusta wypisz "EMPTY"
- `R x y` - jeżeli x nie jest obecny w liście wypisz `FALSE`, w przeciwnym razie zastąp pierwsze wystąpienie wartości x przez y i wypisz `TRUE` (ang. *replace*)
- `S` - wypisz rozmiar listy

Lista powinna przechowywać elementy typu `int`. Należy obsłużyć ewentualne błędy (wyjątki). Maksymalnie naraz w liście może znajdować się do 10^3 elementów.

Zadanie 2. Generator

Proszę napisać program `Generator.x`, który generuje *losowe* dane wejściowe dla programów z tego zestawu zgodne z podanym formatem. Liczbę operacji podać jako argument linii komend.

Uwagi

- Na platformę Pegaz należy wysłać spakowany katalog w formacie `.tar.gz` lub `zip`.
- Katalog musi się nazywać `Zestaw03` i zawierać tylko pliki źródłowe i `Makefile`.
- Pliki źródłowe muszą mieć podaną nazwę, a programy wykonywalne muszą mieć *rozszerzenie* `.x`.
- Wywołanie komendy `make` w tym katalogu powinno kompilować wszystkie programy i tylko kompilować.
- Kompilacja musi przebiegać bez błędów ani ostrzeżeń.
- Należy używać własnych implementacji typów danych w programach.
- Programy nie powinny wypisywać niczego ponad to co opisano w instrukcji. Proszę dokładnie czytać opis formatu danych wejściowych i wyjściowych.
- Implementacje klas mogą znajdować się w pliku nagłówkowym. Taka konstrukcja jest konieczna w przypadku szablonów klas.

Dodatkowe punkty

Dodatkowe punkty (po 1 pkt) można zdobyć za:

- Implementacja iteratora (patrz poniżej)
- Napisanie szablonów klas, konstruktorów (domyślny, kopiujący i przenoszący), destruktorów, operatory przypisania (kopiujący i przenoszący)
- Wykorzystanie referencji do r-wartości, semantyki przenoszenia, uniwersalnych referencji, doskonałego przekazywanie
- Napisanie testera

Pytania

1. Jakie są zalety implementacji wskaźnikowej, a jakie implementacji tablicowej?
2. Zastanowić się jak zaimplementować listę dwukierunkową zapamiętując tylko **jeden** wskaźnik?
3. Czym się różni *odwołanie uniwersalne* od *odwołania do r-wartości* (dla chętnych)?

Wskazówki

Implementacja wskaźnikowa

Klasa zagnieżdżona węzła `Node` przechowuje element typu `T` oraz wskaźniki do poprzedniego i następnego węzła (lub `nullptr` gdy jest to skrajny element listy). W klasie `LinkedList` wskaźniki na początek i koniec listy (odpowiednio `head` i `tail`) warto zastąpić obiektem typu `Node`. Wówczas jego pole `next` pełni rolę wskaźnika `head`, a pole `prev` pełni rolę wskaźnika `tail`, rozmiar listy można przechować w polu `x`. Taka konstrukcja upraszcza kod ponieważ pozwala na automatyczne uwzględnienie przypadków dodania elementu na początek, środek bądź koniec listy. O tym mówi Linus Torvalds w wywiadzie.

```
template<class T>
class LinkedList {  // Klasa listy
    struct Node {       // Zagnieżdżona klasa węzła
        T x;            // Element przechowywany przez węzeł listy
        Node* prev;     // Wskaźnik do poprzedniego węzła
        Node* next;     // Wskaźnik do kolejnego węzła
    };
    struct Iterator {
        Node *ptr;
    }

    Node guard;         // Wskaźniki do pierwszego i ostatniego węzła
    int size;           // Ew. rozmiar listy
    ...                 // Reszta zgodna z AbstractList
};
```

Szablony klas

W ramach tego zestawu należy napisać szablony klas będące implementacją tablicową, wskaźnikową oraz kursorową *abstrakcyjnego typu danych* jakim jest **lista** według następującego schematu:

```
template<class T>
class AbstractList {
    struct Iterator;            // Zagnieżdżona klasa iteratora

    template<class U>           // Uniwersalna referencja U&&
    void push_front(U&& x);     // Wstawia element na początek listy
    T pop_front();              // Usuwa element z początku listy i zwraca jego
                                // wartość lub wyrzuca wyjątek gdy lista jest pusta
    template<class U>
    void push_back(U&& x);      // Wstawia element na koniec listy
    T pop_back();               // Usuwa element z końca listy i zwraca jego
                                // wartość lub wyrzuca wyjątek gdy lista jest pusta
    Iterator find(const T& x);  // Wyszukuje element o wartości `x`
                                // i zwraca jego pozycję
    Iterator erase(Iterator);   // Usuwa element wskazywany przez iterator
                                // i zwraca iterator do kolejnego elementu
    template<class U>
    Iterator insert(Iterator it, U&& x);    // Wstawia element x przed pozycję
                                            // it i zwraca pozycję x
    int remove(const T& x);     // Usuwa wystąpienia x i zwraca ich liczbę
```

```
    int size();                 // Zwraca liczbę elementów w liście
    bool empty();               // Zwraca `true` gdy lista jest pusta
    Iterator begin();           // Zwraca iterator na pierwszy element
    Iterator end();             // Zwraca iterator na koniec listy,
                                // czyli za ostatnim elementem
};
```

> ***Uwaga:*** *Poszczególne klasy* ***nie*** *mają dziedziczyć po* `AbstractList`*, tylko się na niej wzorować. W* języku C++ *nazywamy to konceptem. W języku Java stosowany jest mechanizm interfejsów. Podany interfejs różni się od klasy szablonowej* `std::list`*.*

Klasy implementujące struktury danych powinny również zawierać konstruktory (domyślny, kopiujący i przenoszący), destruktor, operatory przypisania (kopiujący i przenoszący), iterator, operacje `push_back` i `push_front` oraz `insert`, których argumentem jest uniwersalna referencja. Domyślny konstruktor tworzy pustą listę. W rozwiązaniach należy wykorzystać następujące elementy: szablony, referencje do r-wartości, semantyka przenoszenia, uniwersalne referencje, doskonałe przekazywanie. Więcej informacji można znaleźć w *Wskazówkach i elementach języka C++* (zakładka Materiały na MS Teams).

> *Złożoność obliczeniowa programów powinna być* optymalna *dla danej implementacji. Dla klas szablonowych* deklaracje *i* definicje *muszą znajdować się w jednym pliku nagłówkowym. Nie należy używać kontenera* `std::vector`*.*

Iteratory

Napisać zagnieżdżony szablon klas `struct Iterator`, który będzie **iteratorem** dwukierunkowym (jednokierunkowym) dla listy. W przypadku implementacji wskaźnikowej, *iterator* powinien przechowywać jedynie wskaźnik do węzła listy (koniec listy to `nullptr`). Dla implementacji kursorowej, struktura *iteratora* jest nieco bardziej skomplikowana. Klasa `iterator` powinna spełniać warunki iteratora dwukierunkowego. Zaimplementować operatory:

- `++` - inkrementacji, zwraca `*this` czyli obiekt typu `iterator&`
- `--` - dekrementacji, zwraca `*this` czyli obiekt typu `Iterator&` (nie dotyczy implementacji kursorowej)
- `*` - dereferencji, zwraca obiekt typu `T&`
- `==`, `!=` - porównania zwraca typ `bool` (operator `!=` należy wyrazić przez `==`)

Dodatkowo klasy implementujące listy muszą posiadać następujące metody:

Prawidłowa implementacja powinna zapewniać poprawne działanie pętli *for-each*:

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

Uwaga: Standard *C++17* wycofał kilka elementów, które były w C++ od początku. Jednym z nich jest `std::iterator`.

Andrzej Görlich
a.goerlich@outlook.com