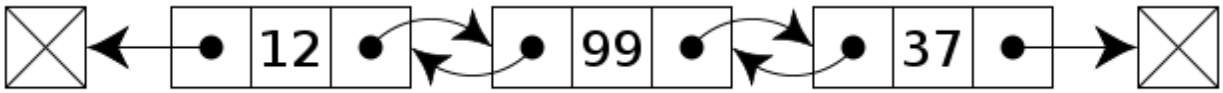
Języki programowania, Zadanie 6

Tworzymy listę podwójnie powiązaną (ang. *double-linked list*).
Listy są jednymi z najważniejszych struktur danych. Wyobraźmy sobie, że tworzymy bazę danych jakiś produktów, które sprzedajemy w naszym sklepie, np. gier komputerowych. Nie wiemy, ile tych gier będzie w naszym sklepie. Co więcej, ich liczba będzie się zmieniać w zależności od tego, ile będziemy ich sprzedawać oraz ile z nich będziemy otrzymywać z dostaw. Oczywiście moglibyśmy taką bazę danych stworzyć poprzez deklarowanie dużej tablicy gier komputerowych w naszym programie. Jeśli jednak mówimy tu o wielkościach rzędu setek tysięcy, to deklarowanie „na wyrost" np. 100 tysięcy więcej miejsc w naszej bazie niż ilość elementów, powodowałoby znaczną stratę ilości pamięci. Oczywiście jest to tylko ilustracja problemu. Jednym ze sposobów, by poradzić sobie z takim problemem są tzw. dynamiczne struktury danych.
Listy są dynamicznymi strukturami danych, które mogą zmieniać swój rozmiar w trakcie działania programu. Lista będzie miała zatem taką długość i rozmiar, jaka jest w niej aktualnie liczba elementów i długość ta oraz rozmiar będą się zmieniać w zależności od dodawania bądź usuwania z niej elementów.
Budowa listy dwukierunkowej (podwójnie powiązanej) jest następująca: każdy jej element (rekord) posiada swoje **dane** oraz **2 wskaźniki** – do elementu poprzedniego oraz następnego. Powstaje zatem łańcuch powiązań elementów pomiędzy sobą, ilustruje to schemat poniżej:



A doubly linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.

Zauważamy, że lista posiada ponadto 2 specjalne, wyróżnione elementy – pierwszy i ostatni (nazywane różnie, np. head i tail, first i last, itp.), które wskazują na koniec naszego łańcucha (czyli na `NULL`). Od razu widzimy, że bezpośredni dostęp do dowolnego elementu takiej struktury jest niemożliwy (mamy bezpośredni dostęp jedynie do pola początkowego lub końcowego). Żeby więc wybrać interesujący nas element ze środka listy, musimy ustawić się na początku lub końcu listy i po niej przeiterować do wybranej pozycji.
Podczas jednego z laboratoriów używaliśmy już listy zawartej w Standardowej Bibliotece Szablonów języka C++ (tzw. STL). Dzisiaj chcemy niejako sami zaimplementować podobną listę.

Dla uproszczenia napiszemy listę zmiennych typu `double`. Implementacja listy w języku C++ polega w ogólności na stworzeniu klasy `List` oraz prostej klasy `Element`. Struktura typu `Element` powinna zawierać wartość oraz wskaźniki na element następny oraz poprzedni. Klasa `List` powinna zawierać wskaźnik na element pierwszy w liście oraz na element ostatni w liście. Plik **List.h** powinien wyglądać następująco:

```
class List
{
  struct Element
  {
    double value; //nasze dane – wartosc typu double
    Element *prev; //wskaznik na element poprzedni
    Element *next; //wskaznik na element nastepny
  };

  int fCapacity; //rozmiar listy
  Element *firstElement; //wskaznik na pierwszy element w liscie
  Element *lastElement; //wskaznik na ostatni element w liscie

 public:
  List(); //konstruktor domyslny
  List(List& list); //konstruktor kopiujacy
  ~List(); //destruktor
```

```
  void AddLast(double val); //dodaje element na koncu listy
  void AddFirst(double val); //dodaje element na poczatku listy
  void AddAt(int id, double val); //ustawia element na pozycji id
  void RemoveLast(); //usuwa element z konca listy
  void RemoveFirst(); //usuwa element z poczatku listy
  void RemoveAt(int id); //usuwa element z pozycji id
  double GetLast(); //pobiera ostatni element z listy
  double GetFirst(); //pobiera pierwszy element z listy
  double GetAt(int id); //pobiera element z pozycji id
  void Clear(); //usuwa wszystkie elementy z listy
  int Capacity(); //zwraca rozmiar
  void PrintList(); //wypisyje liste na ekran
};
```

Opis poszczególnych metod:

- Konstruktor domyślny:
  Tworzy obiekt typu `List` i ustawia pole `fCapacity` na 0 oraz wskaźniki `firstElement` i `lastElement` na `NULL` (oznacza to, że tworzymy pustą listę bez żadnego elementu).
- Konstruktor kopiujący:
  Kopiuje pola `fCapacity`, `firstElement` i `lastElement` tworząc nowy obiekt `List` z podanego obiektu `List`.
- Destruktor:
  Jeśli lista nie jest pusta, to iteruje po wszystkich elementach listy i usuwa je operatorem `delete`. Jeśli lista jest pusta, to zwraca na ekran tekst „List is empty!”.
- `void AddLast(double val)`:
  Dodaje element na końcu listy i zwiększa `fCapacity` o 1.
  Wskazówka: należy stworzyć, używając operatora `new`, obiekt typu `Element`, przyporządkować jego polu `value` podaną wartość `val`, zaś wskaźnik na element nstępny `next` ustawiamy na `NULL`. Następnie sprawdzamy, czy lista jest pusta, czy nie (czy dostawiamy pierwszy element, czy już są jakieś elementy w liście). Jeśli lista **nie jest pusta**, wskaźnik `prev` naszego nowego obiektu typu `Element` ustawiamy na `lastElement`. Na końcu musimy ustawić (przesunąć) wskaźnik `lastElement` na nasz nowy obiekt. Jeśli lista **jest pusta**, wskaźnik `prev` ustawiamy na `NULL`, zaś `lastElement` i `firstElement` na stworzony obiekt typu `Element` (mamy tylko jeden element, więc wskaźniki na pierwszy i ostatni muszą na niego wskazywać). Analogicznie realizujemy wszystkie inne metody!
- `void AddFirst(double val)`:
  Dodaje element na początku listy i zwiększa `fCapacity` o 1.
- `void AddAt(int id, double val)`:
  Dodaje element na konkretnym miejscu w liście, podanym jako parametr `id`, i zwiększa `fCapacity` o 1.
- `void RemoveLast()`:
  Usuwa ostatni element z listy i zmniejsza `fCapacity` o 1.
- `void RemoveFirst()`:
  Usuwa pierwszy element z listy i zmniejsza `fCapacity` o 1.
- `void RemoveAt(int id)`:
  Usuwa element z listy na pozycji `id` i zmniejsza `fCapacity` o 1.
- `double GetLast()`:
  Zwraca wartość `value` ostatniego elementu z listy.
- `double GetFirst()`:
  Zwraca wartość `value` pierwszego elementu z listy.

- `double GetAt(int id)`:
  Zwraca wartość `value` elementu z listy na pozycji `id`.
- `void Clear()`:

1. Czyści całą listę – działa prawie identycznie jak destruktor (używamy tego na obiekcie typu `List`, gdy nie chcemy go usunąć ostatecznie, tak jak to robimy operatorem `delete`, chcemy tylko pozbyć się wszystkich elementów wpisanych do listy, czyli usunąć je operatorem `delete` i ustawić `firstElement` oraz `lastElement` na `NULL` i `fCapacity` na 0).

- `int Capacity()`:
  Zwraca rozmiar `fCapacity` – obecny rozmiar tablicy.
- `void PrintList()`:
  Iteruje po elementach listy i wypisuje wszystkie na ekran w formacie typu:
  `id: 0, value: 5.67`
  `id: 1, value: 7.89`

Do wykonania:

1. W pierwszej kolejności w pliku **List.cpp** tworzymy konstruktor, konstruktor kopiujący, metodę `void AddLast(double val)`, `int Capacity()` oraz `void PrintList()`. W pliku **main.cpp** tworzymy obiekt typu `List` używając operatora `new`, dodajemy dwa dowolne elementy i wypisujemy listę i jej rozmiar na ekran. **1.5 p.**
2. Jeśli poprzedni punkt działa, dodajemy metody `void AddFirst(double val)`, void `RemoveLast()`, `void RemoveFirst()` i sprawdzamy, czy działają. **1.5 p.**
3. Dodajemy pozostałe metody oraz destruktor i sprawdzamy, czy działają (na końcu usuwamy całą listę operatorem `delete`). **2 p.**
4. **Dodatkowo!** Porównujemy działanie naszej listy i listy z biblioteki STL (`#include <list>`), tworząc lużywając operatora `new` obiekt typu `list<double>` (przykład: `list<double> *listStd = new list<double>();`). Dodajemy obu listom 2 takie same elementy na końcu oraz po 1 na początku (**Przypomnienie!** metody `push_back` i `push_front`). Potem dodajemy po jednym elemencie na pozycji nr 2 (**Uwaga!** Aby to zrobić w liście z STL musimy użyć `list<double>::iterator`, przykład:

```
list<double>::iterator iter = listaStd->begin();
iter++; //przesuwa wskaznik iter na pole o id=1
iter++; //przesuwa wskaznik iter na pole o id=2
listaStd->insert(iter,89.45); //wstawia wartosc na pole id=2
```

   Wypisujemy elementy naszej listy za pomocą metody `PrintList()`, zaś listy z STL używając iteratora. **Przypomnienie:**

```
int i = 0;
for(iter=listaStd->begin();iter!=listaStd->end();iter++)
{
   cout<<"id: "<<i<<", value: "<<*iter<<endl;
   i++;
}
```

   Na końcu zaś usuwamy wszystkie elementy metodą `Clear()` naszej listy i `clear()` listy z STL.