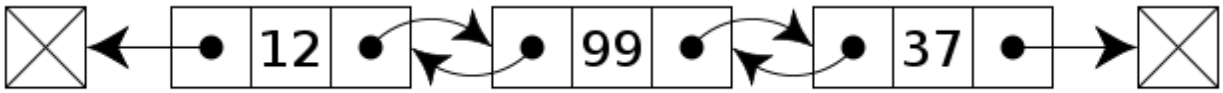
Podstawy programowania, Praca Domowa III

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.

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

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


typedef struct List
{
  int fCapacity; //rozmiar listy
  Element *firstElement; //wskaznik na pierwszy element w liscie
  Element *lastElement; //wskaznik na ostatni element w liscie
} List;

  void SetList(List *list); //ustawia poczatkowe wartosci listy
  void AddLast(List *list, double val); //dodaje element na koncu listy
  void AddFirst(List *list, double val); //dodaje element na poczatku listy
  void PrintList(List *list); //wypisyje liste na ekran
```

Opis poszczególnych funkcji:

- `void SetList(List *list)`
  Ustawia poczatkowe wartości parametrów listy, tzn:
  ```
  list->fCapacity = 0;
  list->firstElement = NULL;
  list->lastElement = NULL;
  ```
- `void AddLast(List *list, double val)`:
  Dodaje element na końcu listy i zwiększa `fCapacity` o 1.
  Wskazówka: należy stworzyć, używając operatora `malloc`, 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 funcje!
- `void AddFirst(List *list, double val)`:
  Dodaje element na początku listy i zwiększa `fCapacity` o 1.
- `void PrintList(List *list)`:
  Iteruje po elementach listy i wypisuje wszystkie na ekran w formacie typu:
  ```
  pozycja: 0, wartosc: 5.67
  pozycja: 1, wartosc: 7.89
  ```

Do wykonania:

1. W pierwszej kolejności w pliku **List.c** tworzymy fumkcję `void SetList(List *list)`, funkcję `void AddLast(List *list, double val)` oraz funkcję `void PrintList(List *list)`. W pliku **main.c** tworzymy zmienną typu `List` używając, dodajemy dwa dowolne elementy i wypisujemy listę na ekran.
2. Jeśli poprzedni punkt działa, dodajemy funkcję `void AddFirst(List *list, double val)` sprawdzamy, czy wszystko działa działają.
3. Jeśli działa nam dodawanie elementów do listy, możemy stworzyć kolejne funkcje (w celu uzyskania pełnej liczby punktów należy napisać 5 dowolnych funkcji z poniższej listy):
   ```
   void AddLast(List *list, double val); //dodaje element na koncu listy
   void AddFirst(List *list, double val); //dodaje element na poczatku listy
   void AddAt(List *list, int id, double val); //ustawia element na pozycji
   id
   void RemoveLast(List *list); //usuwa element z konca listy
   void RemoveFirst(List *list); //usuwa element z poczatku listy
   void RemoveAt(List *list, int id); //usuwa element z pozycji id
   double GetLast(List *list); //pobiera ostatni element z listy
   double GetFirst(List *list); //pobiera pierwszy element z listy
   double GetAt(List *list, int id); //pobiera element z pozycji id
   void Clear(List *list); //usuwa wszystkie elementy z listy
   int Capacity(List *list); //zwraca rozmiar
   ```