Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Програмування

Лабораторна робота №6

*«Зв’язаний список»*

Виконав:

студент групи ІО-64

Андрійчук Д. А.

Залікова книжка №ІО 6401

Київ 2016

**ЛАБОРАТОРНА РОБОТА №6**

*Транспонування та множення матриць*

**Короткі теоретичні відомості**

***В інформатиці, зв'язний список - структура даних, що складається з вузлів, кожен з яких містить як власні дані, так і одну або дві посилання («зв'язки») на наступний і / або попередній вузол списку.***

******

Однобічно зв'язаний список з трьох елементів

В однобічно зв'язаному списку, який є найпростішим різновидом зв'язаних списків, кожний елемент складається з двох полів: data або даних, та вказівника next на наступний елемент. Якщо вказівник не вказує на інший елемент (інакше: next = NULL), то вважається, що даний елемент — останній в списку.

[Doubly-linked-list.svg](https://uk.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Doubly-linked-list.svg)  
Двобічно зв'язаний список

В двобічно зв'язаному списку елемент складається з трьох полів — вказівника на попередній елемент prev, поля даних data та вказівника next на наступний елемент. Якщо prev=NULL, то в елемента немає попередника (тобто він є «головою» списку), якщо next=NULL, то в нього немає наступника («хвіст» списка).

**Код програми**

**class** Node:  
 **def** \_\_init\_\_(self, value=**None**, next=**None**):  
 self.value = value  
 self.next = next  
  
**class** LinkedList:  
 **def** \_\_init\_\_(self):  
 self.first = **None** self.last = **None** self.length = 0  
  
 **def** \_\_str\_\_(self):  
 **if** self.first != **None**:  
 current = self.first  
 out = **'[\n'** + str(current.value) + **'\n'  
 while** current.next != **None**:  
 current = current.next  
 out += str(current.value) + **'\n'  
 return** out + **']'  
 else**:  
 **return '[]'  
  
 def** clear(self):  
 self.\_\_init\_\_()  
  
 **def** add(self, x):  
 self.length += 1  
 **if** self.first == **None**:  
 self.last = self.first = Node(x, **None**)  
 **else**:  
 self.last.next = self.last = Node(x, **None**)  
 **def** Read\_by\_iter(self,i):  
 **if** (self.first == **None**):  
 **return** curr = self.first  
 count = 0  
 **if** i == 0:  
 **return** self.first.value  
 **while** curr != **None**:  
 **if** count == i:  
 **return** curr.value  
 curr = curr.next  
 count += 1  
 **return '[]'  
  
 def** Del\_by\_iter(self,i):  
 **if** (self.first == **None**):  
 **return** curr = self.first  
 count = 0  
 **if** i == 0:  
 self.first = self.first.next  
 **return  
 while** curr != **None**:  
 **if** count == i:  
 **if** curr.next == **None**:  
 self.last = curr  
 old.next = curr.next  
 **break** old = curr  
 curr = curr.next  
 count += 1  
 **def** add\_in\_start(self, x):  
 self.length+=1  
 **if** self.first == **None**:  
 self.last = self.first = Node(x,**None**)  
 **else**:  
 self.first = Node(x,self.first)  
x = -1  
**while** x != **"stop"**:  
 x = input(**"0 - create list\n1 - add item in the end\n2 - add item in the start\n3 - del item by iter\n4 - read list\n5 - read by iter\n6 - clear list\nstop - stop\n:"**)  
 **if** x == **'0'**:  
 L = LinkedList()  
 **elif** x == **'1'**:  
 val = int(input(**'Value'**))  
 L.add(val)  
 **elif** x == **'2'**:  
 val = int(input(**'Value'**))  
 L.add\_in\_start(val)  
 **elif** x == **'3'**:  
 val = int(input(**'Iter'**))  
 L.Del\_by\_iter(val)  
 **elif** x == **'4'**:  
 print(L)  
 **elif** x == **'5'**:  
 val = int(input(**'Iter'**))  
 print(L.Read\_by\_iter(val))  
 **elif** x == **'6'**:  
 L.clear()  
 **elif** x == **'stop'**:  
 **pass  
 else**:  
 print(**'error'**)

**Висновок**

Переваги зв’язаних списків.

* ефективне (за константний час) додавання і видалення елементів
* розмір обмежений тільки об'ємом пам'яті комп'ютера і розрядністю покажчиків
* динамічне додавання і видалення елементів

Недоліки

Недоліки зв'язкових списків випливають з їх головного властивості - послідовного доступу до даних:

* складність прямого доступу до елементу, а саме визначення фізичної адреси за його індексом (порядковому номеру) в списку
* на поля-покажчики витрачають пам'ять (в масивах, наприклад, покажчики не потрібні)
* деякі операції зі списками повільніше, ніж з масивами, так як до довільного елементу списку можна звернутися, тільки пройшовши всі попередні елементи
* сусідні елементи списку можуть бути розподілені в пам'яті нелокального, що знизить ефективність кешування даних в процесорі