DVA245, Laboration 3 - Rekursiv dynamisk (länkad) lista

Mål
3- kunna implementera algoritmer utifrån beskrivningar i pseudokod
6 - vara tillräckligt bekant med några specifika abstrakta datatyper för att vid behov kunna lägga till operationer på dessa. Exempel på sådana abstrakta datatyper är binära träd, dynamiska listor, direktacesslistor, olika sökdatastrukturer, grafer

Ni ska ändra medlemsfunktioner i klassen `IterableLinkedList` som finns i filen `IterableLinkedList.py` så att de blir rekursiva. Funktionerna ni ska skriva om är:

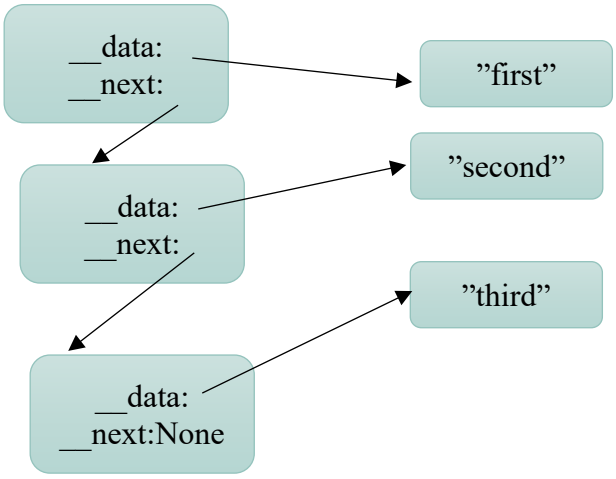
- `__iter__(self)`-som anropas vid iteration över värdena i listan, till exempel i funktionen `__str__` där man går igenom värdena med `for elem in self`
- `__len__(self)` som returnerar antalet länkar/ värden i listan och anropas i main-funktionen: `len(linked_list)`.
- `find(self, goal, key)` som returnerar det första `Link`-objekt där funktionen `key(link.get_value())` returnerar `goal`. Om inte `key` returnerar `goal` för något av listans värden så ska `find` returnera `None`. `find` anropas i funktionerna `search` och `insert_after` som anropas i main-funktionen.
- `delete(self, goal, key)` som tar bort det första `Link`-objekt där funktionen `key(link.get_value())` returnerar `goal`. Om inte `key` returnerar `goal` för något av listans värden så ska funktionen signalera undantag (raise exception). Funktionen returnerar värdet för länken som tagits bort.

Börja med att se till att funktionerna testas innan ni ändrar dem till rekursiva. Gå igenom var de anropas från main-funktionen i `IterableLinkedList.py`. Ändra eller lägg till tester om ni tycker att det behövs. När ni har tester kan ni skriva om funktionerna till rekursiva och veta att de fortfarande fungerar.

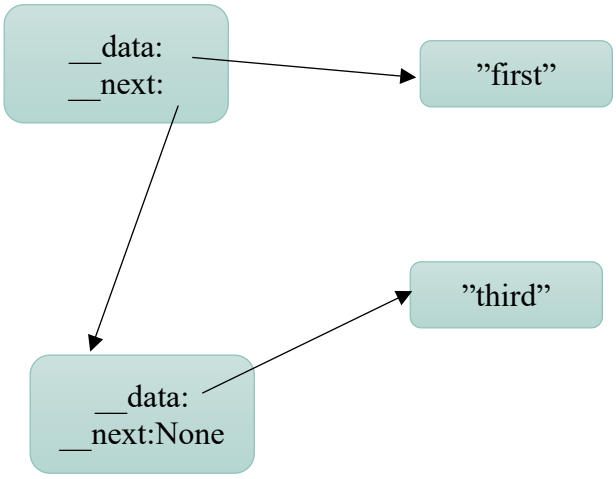
Dessa funktioner är medlemsfunktioner som tar argumentet `self`. Era rekursiva funktioner behöver en länk som argument. Koden blir tydligast om ni delar upp varje funktion ovan i två - en rekursiv funktion, till exempel `iter_recur(self, link)` och en yttre funktion `__iter__(self)` som anropar den rekursiva med den första länken i listan. De rekursiva funktionerna kan gärna vara medlemfunktioner i `IterableLinkedList`. (Ni kan också implementera dem som medlemsfunktioner i klassen `Link` eller som fristående funktioner utanför klassen.)

Fundera över vad som är basfallet i varje funktion, och hur ni kommer närmare basfallet i varje rekursivt anrop.

Funktionen `delete` är den mest komplicerade, eftersom ni måste ändra next-referensen för länken innan i kedjan när ni tar bort en länk. Om vi ska ta bort länken med värde ”second” nedan:



Så behöver vi ändra next-referensen för länken med värde ”first”.



För att underlätta finns nedan pseudokod för funktionen `delete_recur(self, prev_link, goal, key)` som ni får använda er av om ni vill. I pseudokoden är self-referensen borttagen.

```
function DELETE_RECUR(prev_link, goal, key)
         returns data of deleted link
input:   prev_link – link prior to the one to
check next
         goal – the value to look for
         key – delete first link where
         key(link.get_data()) returns goal
local variables:  current – link to check


current←prev_link.GET_NEXT()
if not current then
         raise exception
if KEY(current.GET_DATA()) = goal then
         # base case – goal found!
         prev_link.SET_NEXT(current.GET_NEXT())
         return current.GET_DATA()
# recursive call one link further in list
return DELETE_RECUR(current, goal, key)
```

I det första anropet till `delete_recur` kan ni använda den länkade listan som `prev_link` eftersom den har funktionrna `get_next` och `set_next` som returnerar/ sätter den första länken i listan (medlemsvariableln `__first`).

För redovisning:
Ni behöver kunna förklara hur era nya funktioner fungerar, och hur de testas i main-funktionen eller separat test.

English:

Learning outcomes
3- be able to implement algorithms given descriptions as pseudocode
6 - be sufficiently familiar with some specific abstract data types in order to be able to add new operations on theses. Some examples of such abstract data types are dynamic lists, direct access lists various seach data structures, graphs

You shall change member functions of the class `IterableLinkedList` in the file `IterableLinkedList.py` so that they are recursive. The functions you shall change are:

- `__iter__(self)`-that is called in iteration over the values in the list, for example in the function `__str__` where the values are stepped through with `for elem in self`
- `__len__(self)` that returns the number of links/ values in the list and is called in the main function: `len(linked_list)`
- `find(self, goal, key)` that returns the first `Link`-object where `key(link.get_value())` returns `goal`.If `key` does not return `goal` for any of the values in the list `find` shall return `None`. `find` is called in the functions `search` and `insert_after` that are called in the main function.
- `delete(self, goal, key)` that removes the first `Link`-object where `key(link.get_value())` returns `goal`. If `key` does not return `goal` for any of the values in the list, the function shall raise an exception. The function returns the value of the removed link.

Start with making sure that the functions are tested before you change them into recursive. Go through how they are called from the main function in `IterableLinkedList.py`. Change or add tests if you think it is needed. When you have the tests in place you can rewrite the functions to recursive and know they still work.

These functions are member functions that take the argument `self`. Your recursive functions need a link as argument. The code will be clearest if you split each function above in two – one recursive function, for example `iter_recur(self, link)` and an outer function `__iter__(self)` that calls the recursive with the first link in the list. The recursive functions can kan be member functions of `IterableLinkedList`. (You can also implement them as member functions in the `Link` class or as functions outside the class.)

Think about what is the base case for each function, and how you get closer to the base case in each recursive call.

The function `delete` is the most complicated, since you need to change the next reference of the previous link in the chain when you remove a link. (See illustration and pseudocode as help above)

In the first call to `delete_recur` you can use the linked list as `prev_link` because it has the functions `get_next` and `set_next` that return/ set the first link of the list (the member variable `__first`).

For presentation:
You need to be able to explain how the new functions work and how they are tested in the main function or in a separate test.