LABORATORIUM 8

TREŚCI KSZTAŁCENIA:

- F# -FUNKCYJNE STRUKTURY DANYCH

Spis treści

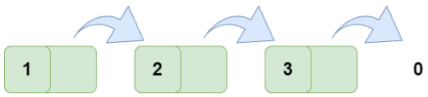
F# - funkcyjne struktury danych

Lista

Matematycznie listę można przedstawić za pomocą równania rekurencyjnego:

$$Lista = \begin{cases} () \\ Element :: Lista \end{cases}$$

Wynika z niego, że lista może być albo listą pustą lub może być parą $Element::Lista$, gdzie $Element$ jest pierwszym elementem listy, a $Lista$ pozostałymi elementami na tej liście. Pierwszy element określa się mianem **głowy listy**, a pozostałe elementy na liście jej **ogonem**. Zastosowany operator :: nazywany jest konstruktorem listy, a funkcję go realizujące nazywamy **cons**. Funkcja ta również służy do tworzenia listy. W praktyce najłatwiej takie listy zaimplementować jako listy łączone (patrz).

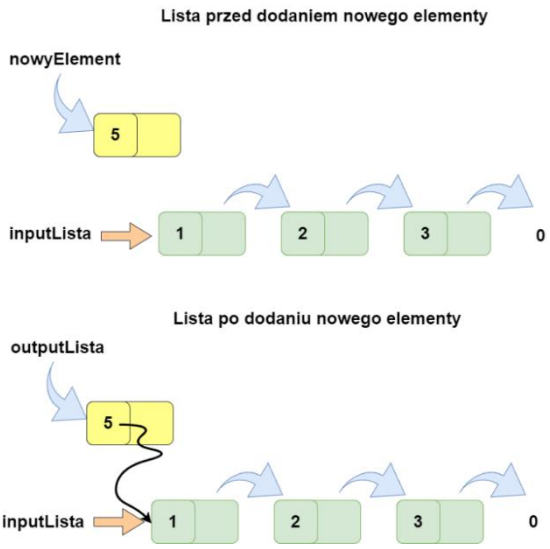


Rys. 1 Budowa listy łączonej

Każdy węzeł składa się z dwóch części: przechowywanych na niej danych oraz wskaźnika na następny element (w przypadku pierwszego węzła, części te to nic innego jak głowa listy i wskaźnik na jej ogon). Taka implementacja może pomóc w zapewnieniu, że struktura będzie zarówno niemutowalna, ale również wydajna.

Dodawanie elementu na początek listy łączonej

Jest to najprostszy przypadek operacji na liście. Wystarczy dodać nowy element na początek listy. Załóżmy, że istniejąca lista jest przesypiana do zmienne inputLista. Jeżeli dołączymy do niej nowy element, stanie się on jej nową głową. Jednak dopóki nie zmodyfikujemy inputLista, to będzie ona dalej wskazywać na głowę oryginalnej listy, natomiast nowy wskaźnik będzie pokazywał na głowę nowej.

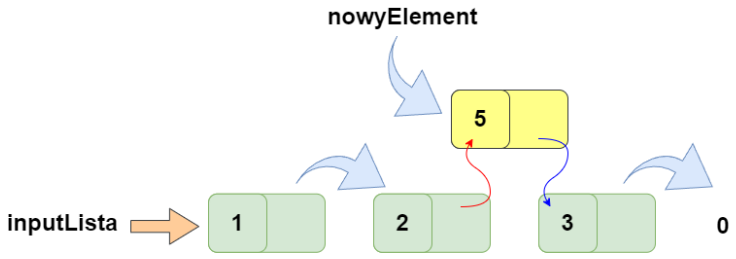


Rys. 2 Dodanie nowego elementu na początek listy łączonej

Dodawanie nowego elementy na początek listy łączonej nie powoduje dodatkowego obciążania dla pamięci, ani potrzeby kopiowania elementów. Istniejącą listę możemy wykorzystać w całości, bez wpływu na jej kształt.

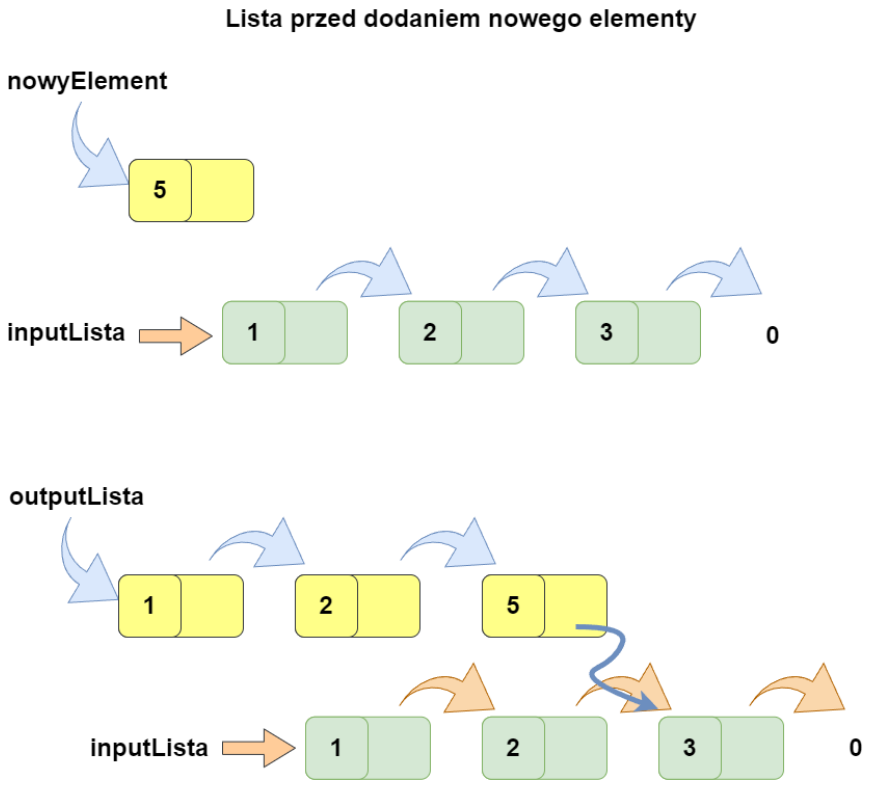
Dodawanie elementu do środka listy łączonej

Operacja dodawania elementu do środka jest bardziej skomplikowana. Załóżmy, że nowy element chcemy wstawić pomiędzy element „2” i „3”, co pokazano poniżej na rysunku:



Rys. 3 Dodanie nowego elementu do środka listy łączonej

Analizując powyższy rysunek, zauważamy że bez problemu możemy ustawić wskaźnik nowego elementu, aby wskazywał na trzeci element. jednak próba ustawienia wskaźnika drugiego elementu tak aby pokazywał na nowy spowoduje zmodyfikowanie listy wejściowych. Niestety nasza lista nie będzie dłużej niemodyfikowana. Zwróćmy uwagę na fakt, że zaburzamy wyłącznie ten fragment listy, który znajdzie się przed nowym elementem. Elementy te pozostaną nietknięte. Możemy ten fakt wykorzystać i ponownie użyć tych elementów. Dzięki temu, aby zachować niemodyfikalność listy, konieczje jest skopiowanie tylko elementów, które w wynikowej liście będą znajdowały się przed nowo dodanym elementem (patrz rysunek poniżej).



Rys. 4 Dodanie nowego elementu do środka listy łączonej

Wynika z tego, że im bliżej początku listy chcemy dodać nowy element, tym nasza funkcja będzie wydajniejsza (więcej elementów ponownie można wykorzystać). Podobnie można zdefiniować usuwanie elementów.

Podstawowe implementacje listy łączonej w F#.

- Empty reprezentuje koniec listy.
- Node zawiera wartość oraz referencję do następnego elementu listy.
- Funkcja addHead dodaje nowy element na początek listy.
- Funkcja printList rekurencyjnie wypisuje zawartość listy.

```
open System
open System.Collections.Generic

// Definicja listy łączonej
type LinkedList<'T> =
    | Empty                          // Pusta lista
    | Node of 'T * LinkedList<'T>  // Węzeł zawierający wartość i referencję do
następnego elementu

// Funckje przydatne podczas wykonywania operacji na głowie i ogonie
let Head =
    function
    | Empty -> failwith "Nie można pobrać głowy z listy pustej"
    | Node(Head,_) -> Head

let Tail =
    function
    | Empty -> failwith "Nie można pobrać ogona z listy pustej"
    | Node(Tail,_) -> Tail

// Funkcja dodająca element na początek listy
let addHead value list =
    Node(value, list)

// Funkcja wyświetlająca elementy listy
let rec printList list =
    match list with
    | Empty -> ()
    | Node(value, next) ->
        printf "%A " value
        printList next

// Funkcja rekurencyjna obliczająca ilość elementów na liście
let rec numberElements =
    function
    | Empty -> 0
    | Node (_,Tail) -> numberElements Tail + 1

// Funckja wyszukująca element na liście, i dodające nowy element za nią
// element - poszukiwany element listy
// newElement - nowy element, który chcemy wstawić
let rec addAfter element newElement =
    function
    | Empty -> failwith ("Nie znaleziono elementu: " + element.ToString())
    | Node (Head,Tail) ->
        if Head = element then
           Node(element, Node (newElement, Tail))
        else
            Node(Head, addAfter element newElement Tail)


// Przykład użycia
[<EntryPoint>]
let main argv =
```

```
    let list1 = Empty
    let list2 = addHead 1 list1
    let list3 = addHead 2 list2
    let list4 = addHead 3 list3
    let ilosc = numberElements list4

    printList list4  // Wynik: 3 2 1
    printf "%d" ilosc

    0
```

Powyższe przykłady są funkcjami rekurencyjnymi, poniżej przedstawiono implementację z wykorzystaniem rekurencji ogonowej.

```
open System
open System.Collections.Generic

type LinkedList<'T> =
    | Empty                          // Pusta lista
    | Node of 'T * LinkedList<'T>  // Węzeł zawierający wartość i referencję do
następnego elementu


let numberElements list =
    let rec count acc =
        function
        | Empty -> acc
        | Node (_, Tail) -> count (acc + 1) Tail
    count 0 list


let addAfter element newElement list =
    let rec addAfterTail acc =
        function
        | Empty ->
            failwith ("Nie znaleziono elementu: " + element.ToString())
        | Node (Head, Tail) ->
            if Head = element then
                // Znaleziono element - składamy nową listę z akumulatorem i
nowymi węzłami
                List.foldBack (fun x acc -> Node(x, acc)) acc (Node(Head,
Node(newElement, Tail)))
            else
                // Dodajemy bieżący element do akumulatora i kontynuujemy
przeszukiwanie
                addAfterTail (Head :: acc) Tail

    // Wywołujemy funkcję z pustym akumulatorem
    addAfterTail [] list
```

Lista łączona z rekordem i opcjonalnym wskaźnikiem

```
// Definicja węzła
type Node<'T> =
    { Value: 'T
      mutable Next: Node<'T> option }  // Referencja do następnego węzła (opcja)

// Funkcja tworząca nowy węzeł
let createNode value =
    { Value = value; Next = None }

// Dodanie elementu na początek listy
```

```
let addToHead newValue head =
    let newNode = createNode newValue
    newNode.Next <- Some head
    newNode

// Funkcja wyświetlająca elementy listy
let rec printNodes nodeOpt =
    match nodeOpt with
    | None -> ()
    | Some node ->
        printf "%A " node.Value
        printNodes node.Next

// Przykład użycia
let node1 = createNode 1
let node2 = addToHead 2 node1
let node3 = addToHead 3 node2

printNodes (Some node3)  // Wynik: 3 2 1
```

Lista łączona za pomocą klas (bardziej obiektowe podejście)

```
type Node<'T>(value: 'T) =
    member val Value = value with get, set
    member val Next: Node<'T> option = None with get, set

type LinkedList<'T>() =
    let mutable head: Node<'T> option = None

    member this.Add value =
        let newNode = Node(value)
        match head with
        | None -> head <- Some newNode
        | Some oldHead ->
            newNode.Next <- Some oldHead
            head <- Some newNode

    member this.Print() =
        let rec printNodes nodeOpt =
            match nodeOpt with
            | None -> ()
            | Some node ->
                printf "%A " node.Value
                printNodes node.Next
        printNodes head

// Przykład użycia
let myList = LinkedList<int>()
myList.Add(1)
myList.Add(2)
myList.Add(3)

myList.Print()  // Wynik: 3 2 1
```

Zadania do samodzielnego rozwiązania

Rozwiązania w postaci niezbędnych plików źródłowych należy przesłać do utworzonego zadania na platformie e-learningowej zgodnie ze zdefiniowanymi instrukcjami oraz w nieprzekraczalnym wyznaczonym terminie.

1. Napisz funkcję, która tworzy listę łączoną na podstawie zwykłej listy (List<'T>).
2. Napisz funkcję, która sumuje elementy listy zawierającej liczby całkowite.
3. Napisz funkcję, która znajduje maksymalny i minimalny element w liście liczbowej.
4. Napisz funkcję, która odwraca kolejność elementów listy.
5. Napisz funkcję, która sprawdza, czy dany element znajduje się w liście.
6. Napisz funkcję, która określi indeks podanego elementu, jeżeli element nie znajduje się na liście zwróć odpowiednią wartość (można wykorzystać unie z dyskryminatorem).
7. Napisz funkcję, która zlicza, ile razy dany element występuje w liście.
8. Napisz funkcję, która łączy dwie listy łączone w jedną.
9. Napisz funkcję, która będzie przyjmowała dwie listy liczb całkowitych i zwracała listę wartości logicznych, gdzie true określa, że liczba na pierwszej liście była większa, a false, że wartość na drugiej liście byłą większa. Jeżeli jednia lista jest dłużna od drugiej zwróć wyjątek informujący o tym fakcie.
10. Napisz funkcję, która zwraca listę zawierającą tylko elementy spełniające określony warunek.
11. Napisz funkcję, która usuwa duplikaty z listy.
12. Napisz funkcję, która dzieli listę na dwie części: jedną z elementami spełniającymi warunek, a drugą z pozostałymi elementami.

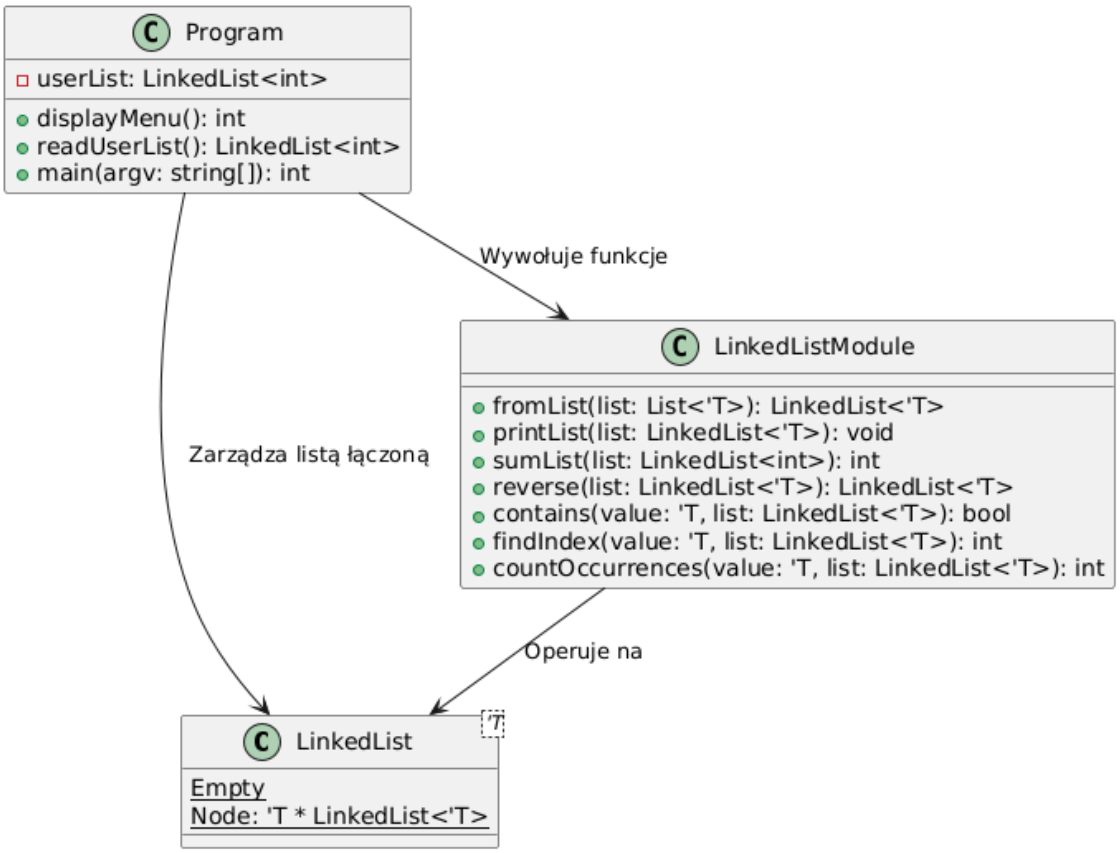
Rozwiązania dla powyższych zadań należy rozwiązać zgodnie z poniższym opisem:

- Struktura listy łączonej: definicja typ LinkedList<'T> z węzłami i pustą listą.
- Moduł LinkedList: Zawiera funkcje operujące na liście:
    - Tworzenie listy.
    - Wyświetlanie elementów.
    - Sumowanie, odwracanie, sprawdzanie obecności elementu.
    - Znajdowanie indeksu i liczenie wystąpień elementów.
- Menu: Program wyświetla menu i pozwala na wybór opcji.
- Interakcja z użytkownikiem:
    - Wczytywanie listy.
    - Wprowadzanie wartości do zadań, np. wyszukiwanie lub zliczanie.
- Główna pętla: Program działa w pętli, aż użytkownik wybierze opcję "0" (wyjście).

Poniżej przedstawiono przykładowy diagram klas dla powyższej struktury programu:

Przedstawiony kod można podzielić na dwie główne części:

- Struktura danych – Lista łączona zdefiniowana jako typ LinkedList<'T> oraz operacje na niej (funkcje modułu LinkedList).
- Logika aplikacji – Wywołania funkcji, pętla programu oraz obsługa menu dla użytkownika.



Rys. 5 Diagram klas

Wyjaśnienie diagramu

1. Klasa LinkedList<'T>:
   - To rekurencyjny typ danych, który reprezentuje listę łączoną.
   - Zawiera dwie możliwe wartości:
     - Empty – pustą listę.
     - Node – węzeł przechowujący element i referencję do kolejnego elementu listy.
2. Moduł LinkedList:
   - Zawiera statyczne metody (funkcje) do operacji na liście łączonej.
   - Przykładowe funkcje:
     - fromList: Konwertuje standardową listę (List<'T>) na listę łączoną (LinkedList<'T>).
     - printList: Wyświetla elementy listy.
     - sumList: Sumuje liczby całkowite w liście.
     - reverse: Odwraca listę.
     - contains: Sprawdza, czy dany element znajduje się w liście.
     - findIndex: Zwraca indeks podanego elementu.
     - countOccurrences: Zlicza liczbę wystąpień elementu.
3. Klasa Program (funkcja główna):
   - Zawiera zmienną userList przechowującą listę łączoną.
   - Zawiera funkcje:
     - displayMenu: Wyświetla menu i zwraca wybór użytkownika.
     - readUserList: Wczytuje listę od użytkownika i konwertuje ją na listę łączoną.
     - main: Główna funkcja programu, która obsługuje menu i wywołuje odpowiednie operacje z modułu LinkedList.