Лекция 9

Структури от данни в програмирането

Основни дефиниции

Най-общо, под ***структура от данни*** се разбира **организирана информация, която може да бъде описана, създадена и обработена с помощта на компютърна програма** (Майер и Бодуен).

Често (макар и не съвсем точно) като синоним на понятието структура от данни се използва терминът ***информационна структура***.

За да се определи една структура от данни, е необходимо да се зададат:

- **логическото описание на структурата**, което описва тази структура на базата на декомпозицията ѝ на по-прости структури (компоненти), основните операции над структурата и декомпозицията на основните операции на по-прости операции;
- **физическото представяне на структурата**, което дава методи за представяне на тази структура в паметта на компютъра.

Важна операция над коя да е структура от данни е операцията **достъп до компонентите на структурата**. Тази операция е тясно свързана с физическото представяне на структурата от данни. На базата на тази операция структурите от данни се класифицират като ***прости*** и ***съставни***.

Прости са тези структури от данни, за които операцията **достъп** се осъществява до структурата като цяло. Такива структури са числата, символните (знаковите) и булевите данни. Обикновено на всяка от тези структури в езиците за програмиране съответстват вградени типове данни.

Съставни са тези структури от данни, за които операцията **достъп** се осъществява до компонентите (елементите) на структурата, а не до структурата като цяло. Такива структури са масивът, записът, списъкът (т. нар. свързан списък), опашката, стекът и др.

Съставните структури от данни се делят на ***статични*** и ***динамични***.

Съставна структура от данни, която се състои от фиксиран брой елементи и за която не са допустими операциите включване и изключване на елемент, се нарича ***статична***. За статичните структури от данни в паметта на компютъра се отделя фиксирано количество памет.

Такива структури са ***масивът*** и ***записът***.

Съставна структура от данни, която се състои от променлив брой елементи и за която са допустими операциите включване и изключване на елемент, се нарича ***динамична***.

Такива структури са ***стекът***, ***опашката***, ***дървото***, ***графът*** и др.

Не е целесъобразно от практическа гледна точка в език за програмиране с общо предназначение да се поддържат вградени типове данни за всяка от динамичните структури.

Структура от данни масив

Логическо описание на масив

Масивът е **крайна редица** от фиксиран брой елементи от един и същ тип. Към всеки елемент от редицата е възможен пряк достъп, който се осъществява чрез ***индекс***. Операциите включване и изключване на елемент от масив са недопустими, т.е. масивът е **статична структура**.

Физическо представяне

Елементите на масива се записват последователно в оперативната памет. За всеки елемент от редицата се отделя фиксирано количество памет.

Структура от данни запис

Логическо описание

Записът е **крайна редица** от фиксиран брой елементи, които могат да са от различни типове. Възможен е пряк достъп до всеки елемент от редицата, който се осъществява чрез ***име***.

Елементите на редица, представляваща запис, се наричат ***полета на записа***. Операциите включване и изключване на елемент са недопустими, т.е. структурата е **статична**.

Физическо представяне

Полетата на записа се записват последователно в паметта на компютъра.

Структура от данни множество

Логическо описание

Множеството е **съвкупност** от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими. Възможен е достъп както до отделните компоненти на множеството, така и до структурата като цяло. Достъпът е пряк.

Физическо представяне

Използва се последователно представяне на структурата в паметта. За целта за всеки допустим елемент се посочва дали принадлежи или не принадлежи на множеството.

Структура от данни свързан списък

Логическо описание

Свързаният списък е **крайна редица** от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими в произволно място на редицата. Възможен е достъп до всеки елемент на списъка, като достъпът до първия елемент е пряк, а до останалите елементи – последователен.

Физическо представяне

Тъй като операциите включване и изключване на елемент са възможни на произволно място в списъка, най-естествено е свързаното физическо представяне на списък. Съществуват различни форми на свързано представяне. Най-често се използват:

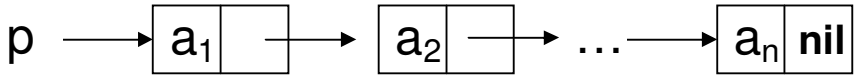
- свързано представяне с една връзка;
- свързано представяне с две връзки.

За свързания списък

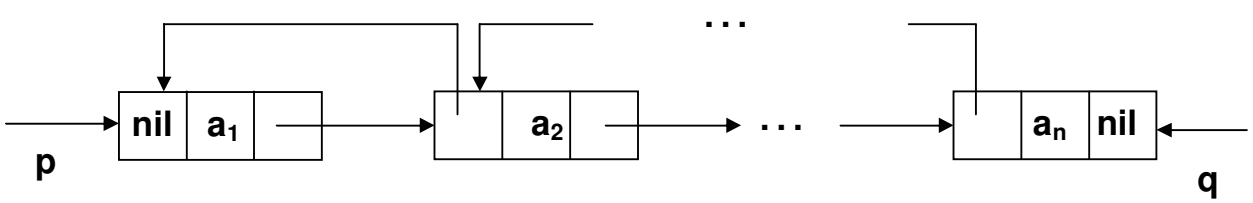
$a_1, a_2, \ldots, a_n$

тези представяния имат следния вид:

Свързано представяне с една връзка



Свързано представяне с две връзки



Структура от данни стек

Логическо описание

Стекът е **крайна редица** от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими само за единия край на редицата, който се нарича ***връх на стека***. Възможен е достъп само до елемента, намиращ се на върха на стека, при това достъпът е ***пряк***.

Примери

Ако редицата $a_1, a_2, \ldots, a_n$ е стек с връх a_1, включването на елемент a от тип, съвпадащ с типа на елементите на стека, води до получаването на нов стек, на върха на който е елементът a, т.е. $a, a_1, a_2, \ldots, a_n$.

Ако редицата $a_1, a_2, \ldots, a_n$ е стек с връх a_1, изключването на елемент (елемента от върха на стека) води до получаването на стека $a_2, \ldots, a_n$.

При тази организация на логическите операции, последният включен в стека елемент се изключва първи. Затова стекът се определя още като структура “последен влязъл – пръв излязъл” (last in – first out, LIFO).

Физическо представяне

Широко се използват два основни начина за физическо представяне на стек: последователно и свързано.

При последователното представяне се запазва блок от паметта, вътре в който стекът да расте и да се съкращава.

Свързаното представяне на стека е аналогично на свързаното представяне на списък с една връзка.

Структура от данни опашка

Логическо описание

Опашката е **крайна редица** от елементи от един и същ тип. Операцията *включване на елемент* е допустима само за единия (например десния) край на редицата, който се нарича ***край на опашката***. Операцията *изключване на елемент* е допустима само за другия (левия) край на редицата, който се нарича ***начало на опашката***. Възможен е пряк достъп само до елемента, намиращ се в началото на опашката.

При описаната организация на логическите операции, последният включен в опашката елемент се изключва последен, а първият – първи. Затова опашката се определя още като структура от данни “пръв влязъл – пръв излязъл” (first in – first out, FIFO).

Физическо представяне

Аналогично на стека, при опашката се използват два основни начина за физическо представяне: последователно и свързано.

При последователното представяне се запазва блок от паметта, вътре в който опашката да расте и да се съкращава.

Свързаното представяне на една опашка е аналогично на това на стека, но се добавя още и указател към последния елемент на опашката.

Структура от данни дърво

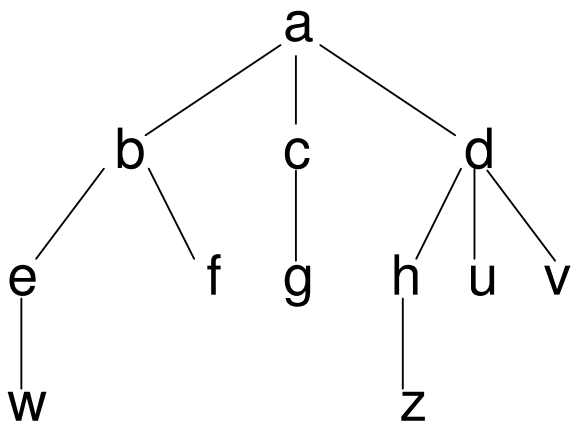
Логическо описание

Нека **Те** е даден тип данни. **Дърво от тип Те** е структура, която е образувана от:

- данна от тип **Те**, наречена **корен на дървото от тип Те**;
- крайно, възможно празно множество с променлив брой елементи – дървета от тип **Те**, наречени **поддървета на дървото от тип Те**.

Пример

Нека **a**, **b**, **c**, … , **x**, **y**, **z** са елементи от тип **Te**. Следното дърво от тип **Te**



има корен **a** и три поддървета, които са дървета от тип **Te**.

Някои определения

- ***Лист*** (листо) на дадено дърво – това е корен на поддърво на разглежданото дърво, което няма поддървета.
- ***Връх*** (***възел***) – това е корен на поддърво. Върховете, които не съвпадат с корена и листата, се наричат ***вътрешни върхове***.
- ***Родител (баща)*** на даден връх **v** – това е върхът **p**, който е такъв, че **v** е корен на поддърво на дървото с корен **p**. Коренът на дървото няма родител, а останалите му върхове имат точно по един родител (баща).
- ***Предшественици*** на даден връх са неговият родител (неговият баща) и предшествениците на неговия баща. Коренът на дървото няма предшественици.

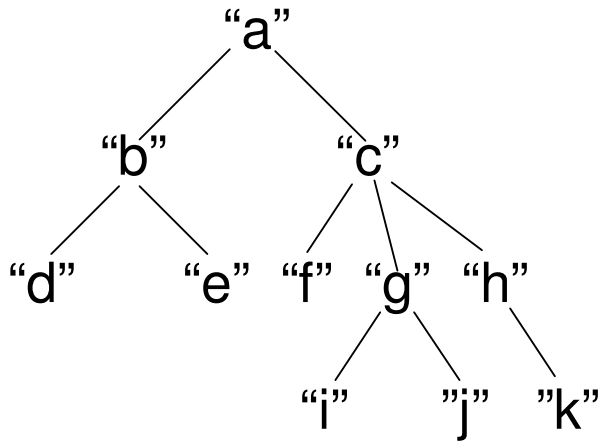
- ***Пряк наследник (син)*** на даден връх **v** е всеки връх, за който **v** е родител (баща). Листата на дървото нямат синове.
- ***Наследници*** на даден връх са неговите преки наследници (т.е. синовете му) и наследниците на неговите синове. Листата на дървото нямат наследници.
- ***Път*** в дървото се нарича всяка редица от вида $n_1, n_2, \ldots, n_k$ от върхове на дървото, която е такава, че n_i е родител (баща) на n_{i+1} ($i = 1,2, \ldots, k-1$).
- ***Дължина*** на пътя $n_1, n_2, \ldots, n_k, n_{k+1}$ е числото k ($k \geq 0$).
- ***Височина*** на едно дърво се нарича дължината на най-дългия път в дървото, свързващ корена и лист от това дърво.

Физическо представяне на дърво

Обикновено се използва свързаното представяне на дърво от тип **Te**, което се осъществява с помощта на свързан списък. При това физическо представяне в един свързан списък се реализират коренът и указателите към поддърветата на дървото от тип **Te**.

Често се използват свързани представяния, основани на описание на дървото чрез задаване на неговите върхове и техните синове (върховете на дървото и техните родители). Тези представяния са най-характерни за езиците за програмиране от типа на Haskell.

Пример
Нека е дадено следното дърво от тип **String**:



Представяне на даденото дърво чрез списък от върховете му и техните синове:

```
[("a",["b","c"]),("b",["d","e"]),("c",["f","g","h"]),
 ("g",["i","j"]),("h",["k"])]
```

Представяне на даденото дърво чрез списък от върховете му и техните бащи:

```
[("b","a"),("c","a"),("d","b"),("e","b"),("f","c"),
 ("g","c"),("h","c"),("i","g"),("j","g"),("k","h")]
```

Дефиниции на някои функции за работа с дървета

```
type Node = String

type Treetd = [(Node,[Node])]

type Treebu = [(Node,Node)]

type Path = [Node]

tree1 :: Treetd
tree1 = [("a",["b","c"]),("b",["d","e"]),
         ("c",["f","g","h"]),("g",["i","j"]),("h",["k"])]

tree2 :: Treebu
tree2 = [("b","a"),("c","a"),("d","b"),
         ("e","b"),("f","c"),("g","c"),
         ("h","c"),("i","g"),
         ("j","g"),("k","h")]
```

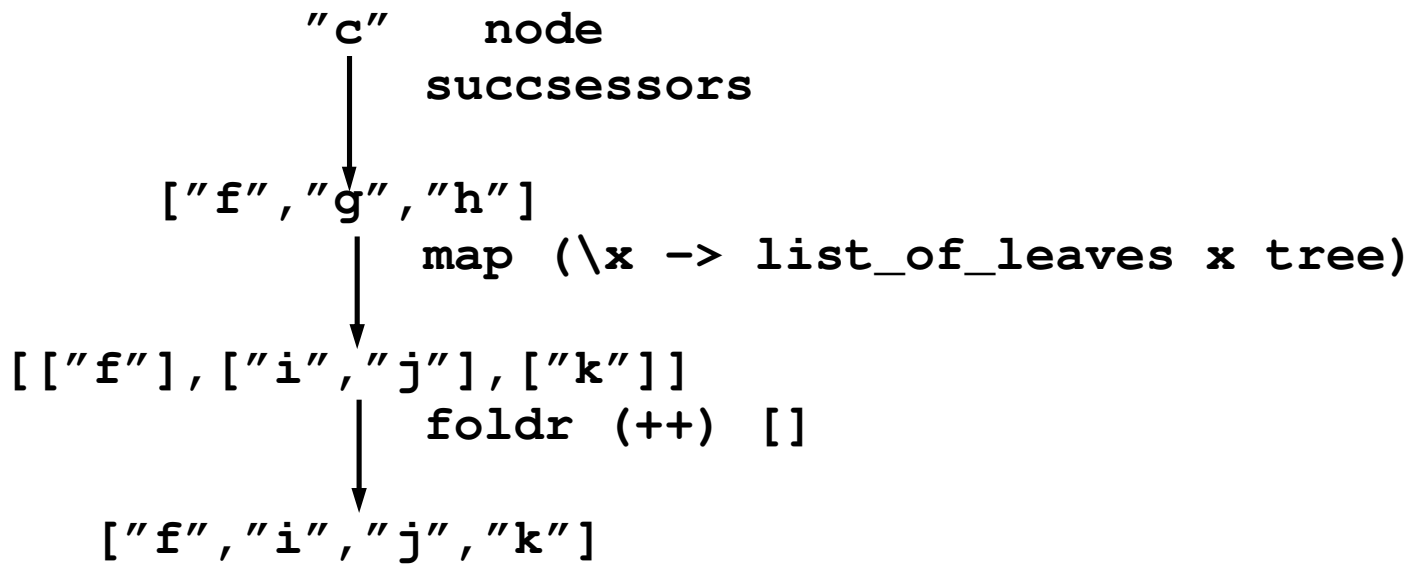
```
assoc :: Eq a => a -> [(a,[b])] -> (a,[b])
-- assoc :: Node -> [(Node,[Node])] -> (Node,[Node])
-- Осъществява търсене в асоциативен списък
-- по даден ключ.
assoc key [] = (key,[])
assoc key (x:xs)
  | fst x==key = x
  | otherwise  = assoc key xs
```

```
successors :: Node -> Treetd -> [Node]
-- Намира преките наследници на даден връх
-- (възел) node в дървото tree.
successors node tree = snd (assoc node tree)

is_a_node :: Node -> Treetd -> Bool
-- Проверява дали node е връх в дървото tree.
is_a_node node tree = rt node || lf node
  where rt :: Node -> Bool
        rt x = elem x (map fst tree)
        lf :: Node -> Bool
        lf x = elem x (concat (map snd tree))

is_a_leaf :: Node -> Treetd -> Bool
-- Проверява дали node е лист в дървото tree.
is_a_leaf node tree = is_a_node node tree
                      && null (successors node tree)
```

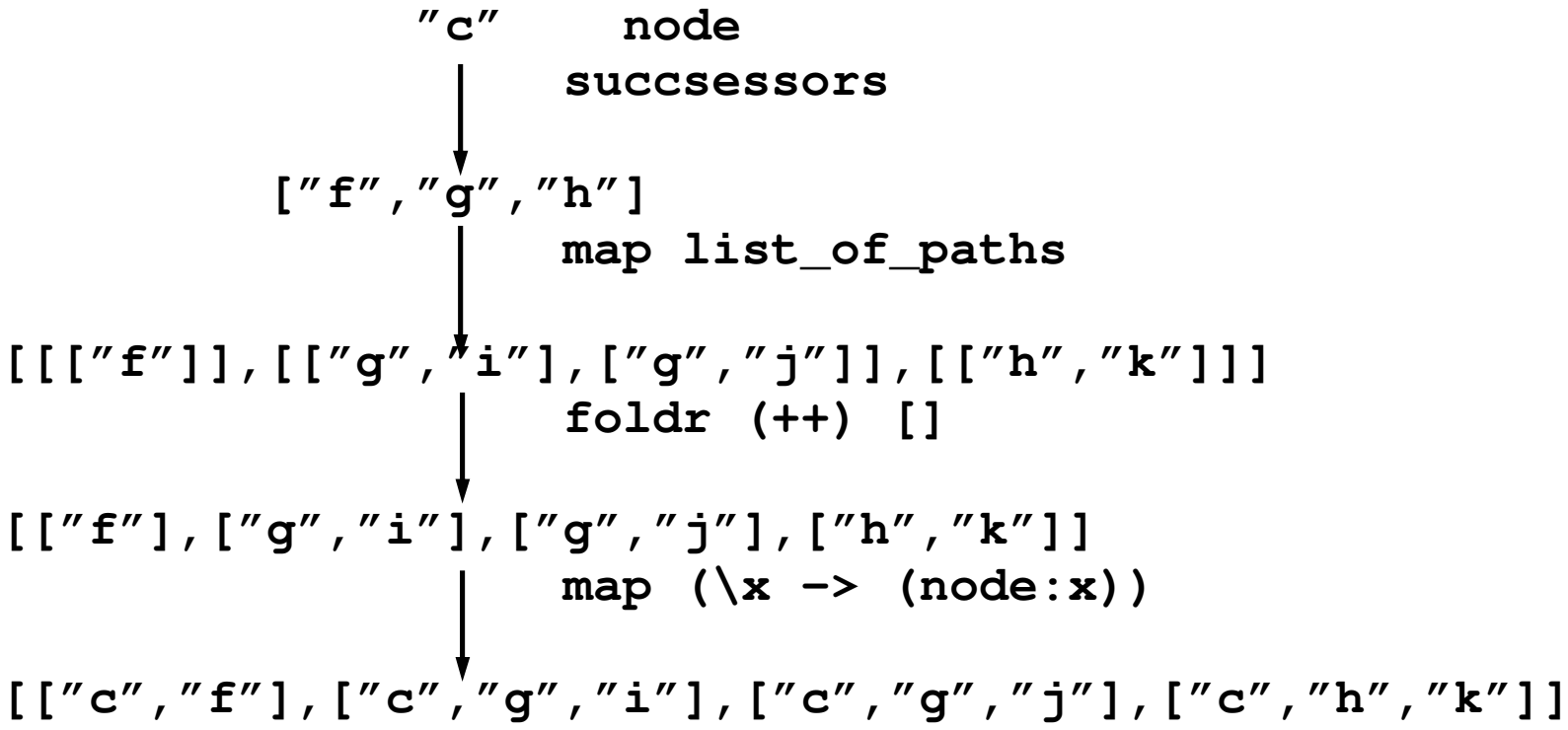
```
parent :: Node -> Treetd -> Node
-- Намира родителя (бащата) на върха node в дървото tree.
parent node [] = ""
parent node tree
  | elem node (successors first_node tree) = first_node
  | otherwise                  = parent node (tail tree)
    where first_node = fst (head tree)

root :: Treetd -> Node
-- Намира корена на дървото tree.
root tree
  | parent first_node tree=="" = first_node
  | otherwise                  = root (tail tree)
    where first_node = fst (head tree)
```

```
list_of_preds :: Node -> Treetd -> [Node]
-- Намира предшествениците на върха node в дървото tree.
list_of_preds node tree
  | node==(root tree) = []
  | otherwise         = father:(list_of_preds father tree)
    where father = parent node tree
```

```
list_of_leaves :: Node -> Treetd -> [Node]
-- Намира листата на поддървото на дървото tree
-- с корен върха node.
list_of_leaves node tree
  | is_a_leaf node tree = [node]
  | otherwise           = foldr (++) []
      (map (\x -> list_of_leaves x tree)
        (successors node tree))
```



```
list_of_paths :: Node -> Treetd -> [Path]
-- Намира пътищата в дървото tree от даден връх node
-- до листата на поддървото с корен node.
list_of_paths node tree
  | is_a_leaf node tree = [[node]]
  | otherwise           = map (\x -> (node:x))
     (foldr (++) []
       (map (\x -> list_of_paths x tree)
         (successors node tree)))
```


"c"
node
succsessors
["f","g","h"]
map list_of_paths
[[["f"]],[["g","i"],["g","j"]],[["h","k"]]]
foldr (++) []
[["f"],["g","i"],["g","j"],["h","k"]]
map (\x -> (node:x))
[["c","f"],["c","g","i"],["c","g","j"],["c","h","k"]]

```
height :: Treetd -> Int
-- Намира височината на дървото tree.
height tree = (foldr1 max
  (map length (list_of_paths (root tree) tree)))-1
```