

Programowanie obiektowe

Sieci powiązań

Paweł Daniluk

Wydział Fizyki

Jesień 2014



Sieci powiązań

Można (bardzo zgrubnie) wyróżnić dwa rodzaje powiązań pomiędzy obiektami:

- 1 obiekt nadrzędny do wielu podrzędnych (np. bycie elementem, zawieranie) – zazwyczaj pomiędzy obiektami różnych klas
- 2 pomiędzy obiektami równoważnymi (np. sąsiedztwo, występowanie oddziaływania) – często pomiędzy obiektami jednej klasy

Pierwszy rodzaj powiązań zazwyczaj skutkuje powstaniem jednej (lub wielu) hierarchii. Dobrą implementacją są konterery.

Sytuacja jest bardziej złożona, gdy sieć powiązań jest bardziej skomplikowana i chcemy obliczyć np.:

- najszybsze połączenie lotnicze z wylotem po południu
- najliczniejszą grupę studentów, z których każdych dwoje chodziło wspólnie na co najmniej jedno zajęcia

Różne topologie sieci

- 1 Lista jednokierunkowa
- 2 Lista dwukierunkowa
- 3 Lista cykliczna
- 4 Drzewo binarne
- 5 Drzewo o dowolnym stopniu wierzchołków
- 6 Graf

Uwaga

Może być wiele rodzajów powiązań.

Listy jednokierunkowe

Każdy obiekt zna swój następnik.

Reprezentacja

Przez pierwszy element

Przechodzenie

Od początku do końca w jednym kierunku. Nie ma możliwości cofania.

Wstawianie

Najłatwiej na początek

Usuwanie

Najłatwiej z początku

Lista jednokierunkowa – przykład

```
class ListElement:
    def __init__(self, val, next=None):
        self.val = val
        self.next = next

    def find(self, val):
        if self.val == val:
            return self
        else if self.next:
            return self.next.find(val)

        return None

    def insert_after(self, val):
        self.next = ListElement(val, self.next)

def push(l, val):
    return ListElement(val, l)

def pop(l):
    return l.val, l.next
```

Listy dwukierunkowe

Każdy obiekt zna swój następnik i poprzednik.

Reprezentacja

Przez pierwszy i ostatni element.

Przechodzenie

Sekwencyjne w obie strony.

Wstawianie

Gdziekolwiek, ale trzeba sekwencyjnie dojść we właściwe miejsce.

Usuwanie

J.w.

Listy cykliczne

Każdy element ma następnik – “pierwszy” jest następnikiem “ostatniego”

Zastosowania

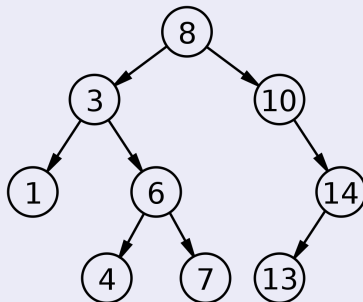
- bufor cykliczny
- równomierny przydział zasobów

Drzewa binarne

Drzewo binarne jest strukturą danych, w której każdy węzeł ma co najwyżej dwa węzły potomne, każdy węzeł z wyjątkiem korzenia ma rodzica oraz żaden węzeł nie może być swoim przodkiem.

Ważne pojęcia

- korzeń
- poddrzewo (lewe, prawe)
- potomek
- rodzic
- przodek



Drzewa binarne

Reprezentacja

Przez korzeń

Wstawianie

Najłatwiej liść

Przechodzenie

DFS i BFS

Usuwanie

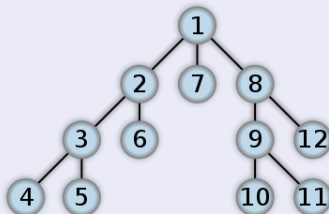
Liście. Usuwanie węzłów może wymagać przebudowy drzewa.

Obchodzenie drzewa

Większość algorytmów operujących na drzewach wymaga sprawdzenia zawartości wierzchołków. Można to robić na wiele sposobów. Wyróżnia się dwa podstawowe: Depth First Search (DFS) i Breadth First Search (BFS).

DFS

Polega na schodzeniu wgłąb. Po odwiedzeniu wierzchołka, odwiedza się wszystkie wierzchołki w lewym poddrzewie, następnie wszystkie wierzchołki w prawym poddrzewie.

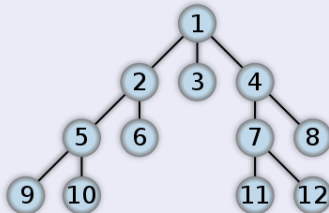


Obchodzenie drzewa

Większość algorytmów operujących na drzewach wymaga sprawdzenia zawartości wierzchołków. Można to robić na wiele sposobów. Wyróżnia się dwa podstawowe: Depth First Search (DFS) i Breadth First Search (BFS).

BFS

Polega na odwiedzaniu wierzchołków według odległości od korzenia.



Obchodzenie drzewa – DFS

Stos

0 1

1 8 7 2

2 8 7 6 3

3 8 7 6 5 4

4 8 7 6 5

5 8 7 6

6 8 7

7 8

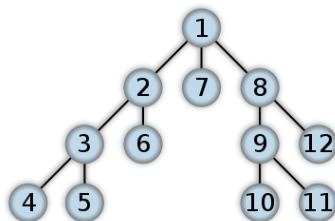
8 12 9

9 12 11 10

10 12 11

11 12

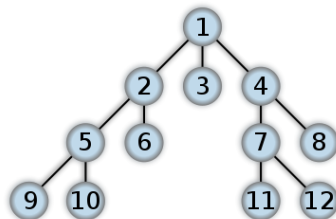
12



Obchodzenie drzewa – BFS

Kolejka

| | |
|----|--------------|
| 0 | 1 |
| 1 | 2 3 4 |
| 2 | 3 4 5 6 |
| 3 | 4 5 6 |
| 4 | 5 6 7 8 |
| 5 | 6 7 8 9 10 |
| 6 | 7 8 9 10 |
| 7 | 8 9 10 11 12 |
| 8 | 9 10 11 12 |
| 9 | 10 11 12 |
| 10 | 11 12 |
| 11 | 12 |
| 12 | |



Drzewo binarne – przykład

```
class Node:
    def __init__(self, val, parent=None, left=None, right=None):
        self.val = val
        self.parent = parent
        self.left = left
        self.right = right

    def DFS(self, pre_op = lambda x: x, in_op = lambda x: x,
            post_op = lambda x: x):
        pre_op(self)
        if self.left is not None:
            self.left.DFS(pre_op, in_op, post_op)
        in_op(self)
        if self.right is not None:
            self.right.DFS(pre_op, in_op, post_op)
        post_op(self)
```

Drzewa – rozszerzenia

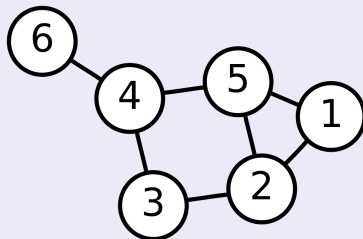
- 1 wierzchołki wyższego stopnia – lista poddrzew zamiast atrybutów `left` i `right`
- 2 wagi krawędzi
- 3 fastrygowanie – dodatkowe połączenia pomiędzy wierzchołkami

Grafy

Graf jest strukturą danych, w której dowolne węzły mogą być połączone krawędzią.

Ważne pojęcia

- węzeł (ang. node, vertex)
- krawędź (ang. edge, arc)
- sąsiedzi
- ścieżka
- cykl
- klika
- spójna składowa
- drzewo rozpinające



Reprezentacje grafów

- 1 Listy sąsiedztwa
- 2 Macierz sąsiedztwa
- 3 Lista krawędzi

Każda reprezentacja ma swoje silne i słabe strony.

Listy sąsiedztwa

Zalety

- relatywnie niewielkie zużycie pamięci (zwłaszcza dla rzadkich grafów)
- łatwo określić sąsiadów wężła
- wygodne obchodzenie grafu

Wady

- pracochłonne sprawdzanie, czy konkretna krawędź należy do grafu

Macierz sąsiedztwa

Zalety

- łatwość implementacji
- sprawdzanie, czy krawędź należy do grafu
- wiele operacji da się zdefiniować w języku macierzy

Wady

- pracochłonne znajdowanie sąsiadów wężła
- zużycie pamięci

Zastosowania grafów

- mapy
- sieci telekomunikacyjne
- cytowania, powiązania pomiędzy dokumentami
- interakcje pomiędzy białkami
- szlaki sygnałowe

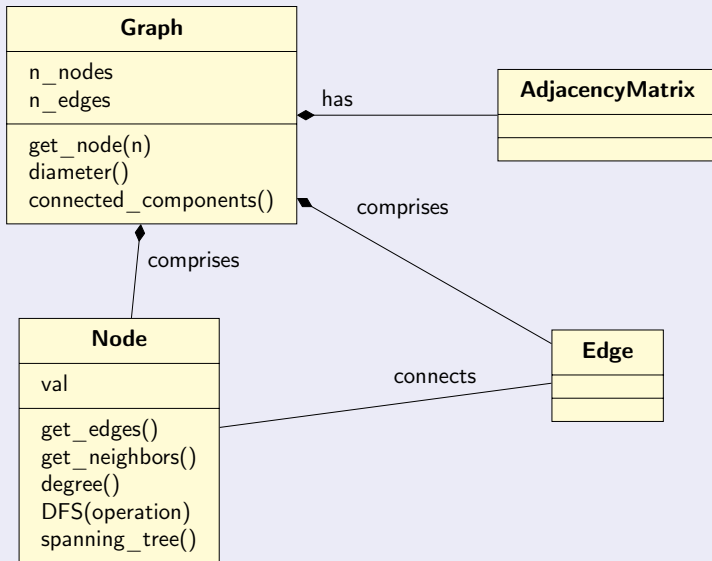
Operacje na elementach i na strukturze

Listę jednokierunkową albo drzewo łatwo można utożsamić z pierwszym elementem (korzeniem). Bardziej skomplikowane struktury na to nie pozwalają.

Niektórych operacji nie da się zdefiniować na elementach grafu. Na przykład:

- znajdowanie spójnych składowych
- znajdowanie najdłuższej ścieżki (średnicy grafu)
- znajdowanie minimalnego drzewa rozpinającego

Przykładowe rozwiązanie



Zadanie 1 – Stacje benzynowe

Problem

Na zamkniętym torze znajduje się N stacji benzynowych. W ich zbiornikach jest w sumie dokładnie tyle paliwa, ile potrzeba do przejechania całego toru. Należy wskazać stację (o ile istnieje), od której można zacząć jazdę z pustym bakiem tak, aby tankując całe paliwo na odwiedzanych stacjach można było przejechać dookoła toru. Zakładamy, że samochód ma nieograniczenie duży zbiornik.

Dane wejściowe

- liczba stacji
- odległości pomiędzy stacjami (położenie stacji na torze)
- ilość paliwa na stacjach

Zadanie 2 – Drzewo BST

Zadanie

Zaimplementuj drzewo BST, które wstawianie i usuwanie liczb, wyszukiwanie ich w drzewie, oraz szybkie znajdowanie liczby poprzedniej i kolejnej dla wskazanej.

Wskazówka

Fastrygowanie jest tutaj pomocne.

Zadanie 3 – Graf

Zadanie

Wykonaj implementację grafu umożliwiającą:

- 1 znajdowanie liczby wierzchołków i krawędzi
- 2 budowanie drzewa rozpinającego o minimalnej wysokości dla wskazanego wierzchołka
- 3 stwierdzanie, czy pomiędzy wskazanymi wierzchołkami istnieje ścieżka