Stos i kolejka

#adapter #interfejs
#stos #stack<>
#! #negacja
#kolejka_LIFO
#kolejka_FIFO #queue<>
#kolejka_priorytetowa
#priority_queue<>

Czas na nieco ambitniejsze tematy, nie uważacie? Stos i kolejki rozmaitych rodzajów to bardzo ważne struktury danych, które wykorzystywane są w zaawansowanych algorytmach i rozwiązaniach wypasionych zadań algorytmicznych. W rozdziale o kontenerach pojawiło się co prawda pojęcie kolejki o dwóch końcach (`deque<>`), ale co by to nie było, bardzo przypominało wektor (`vector<>`), a ten z kolei przypominał zwykłą tablicę, więc głowy nie urywało.

Do takich zwykłych, Bożych kontenerów dodaje się czasem specjalne *adaptery* (*interfejsy*), które zapewniają bardzo szczególną funkcjonalność tej struktury danych. Zaczniemy od *stosu* (ang. **stack**, wymawiaj: *stak*), struktury danych bardzo zasłużonej w historii algorytmiki.*

Adapter to nic innego, jak coś w rodzaju *nakładki*, czyli zestawu metod (funkcji) umożliwiających specyficzne operacje na danym kontenerze. W przypadku stosu jest to znany nam już `deque<>`, ale programista może wybrać dowolny kontener, jeśli ma w tym szczególny cel.

Stos

Stos charakteryzuje się tym, że w danym momencie dostępny jest tylko jego wierzchni element (tak jak w przypadku stosu papierów na biurku widzimy tylko wierzchnią kartkę). Taki element możemy obejrzeć (bez zdejmowania, metoda `top()`), albo zdjąć (metoda `pop()`). Jeśli dodajemy nowy element do stosu (metoda `push()`), wówczas wędruje on na szczyt stosu.

Możemy zapytać się o rozmiar stosu (metoda `size()`), ale na ogół wystarczy nam prostsze pytanie: czy stos jest pusty (metoda `empty()`). To właściwie tyle, i aż tyle.

Przećwiczmy dodawanie kilku elementów do stosu stringów i ich zdejmowanie w typowym trybie:

```
#include <iostream>
#include <stack>
using namespace std;

int main()
{
```

*Trochę o stosie wspominaliśmy przy omawianiu rekurencji, teraz czas na pełnowymiarowy adapter `stack<>`.

```
  stack<string> S;
  S.push("alpha");
  S.push("beta");
  S.push("gamma");
  while(!S.empty())
  {
    cout << S.top() << endl;
    S.pop();
  }
}
```

O, nowy operator: `!`. Oznacza on „nieprawda, że", czyli jest operatorem logicznej *negacji*. Tak więc ze stosu będą zdejmowane kolejne elementy, dopóki on się nie opróżni.

Na ekranie pojawi się:

```
gamma
beta
alpha
```

czyli dane wstawione na stos w porządku odwrotnym. Ale stos nie służy tylko do odwracania kolejności wprowadzonych danych, przecież do tego wystarczyłaby zwykła tablica odczytana od tylca. Rzecz w tym, że operacje wstawiania i zdejmowania ze stosu mogą się przeplatać, co w wyniku daje specyficzną kolejność przetwarzania danych. Pokażemy to na przykładach w dalszych podrozdziałach.

Stos określa się też mianem *kolejki LIFO* (ang. **Last In, First Out**, wymawiaj: *last in, ferst ałt*), czyli tak jak w Biblii: „ostatni będą pierwszymi".

Kolejka FIFO

Jeśli ktoś mówi o *kolejce* w aspekcie algorytmicznym, z reguły ma na myśli właśnie *kolejkę FIFO*.[†] Skrót ten oznacza **First In, First Out**, czyli „kto pierwszy, ten lepszy". Dane przetwarzane są w takiej kolejności, jak zostały umieszczone w kolejce. Ta struktura ma początek, na który może działać metoda `front()` (bezinwazyjnie podgląda czołowy element z kolejki) oraz `pop()` (usuwa czołowy element). Kolejka ma też koniec, na który możemy wstawić nowy element przy pomocy funkcji `push()`.

Napiszmy analogiczny program, jak w przypadku stosu:

```
#include <iostream>
#include <queue>
using namespace std;

int main()
{
  queue<string> Q;
```

[†]Był taki genialny utwór supergrupy Sky, ale kto o tym pamięta. Qrczę, jaki ja jestem stary...

```
  Q.push("Athos");
  Q.push("Porthos");
  Q.push("Aramis");
  Q.push("d'Artagnan");
  while(!Q.empty())
  {
    cout << Q.front() << endl;
    Q.pop();
  }
}
```

Angielskie słowo oznaczające kolejkę wymawiaj: *kiu*. Po uruchomieniu programu na ekranie pojawią się imiona muszkieterów w kolejności wstawienia ich do kolejki:

```
Athos
Porthos
Aramis
d'Artagnan
```

Kolejka w tej formie będzie nam niezbędna w algorytmie przeszukiwania grafu wszerz, co opiszemy w oddzielnych *Miaukotach*.

Kolejka `queue<>` (jeśli nie zażyczymy sobie inaczej), tak samo jak stos, opiera się na kontenerze `deque<>`.

Kolejka priorytetowa

Nie sądzę, aby wielu z Czytelników pamiętało czasy komuny – może niektórzy nauczyciele. Wtedy to były kolejki! I do tego miały one pewną szczególną cechę: nie liczyło się, kiedy kto stanął w kolejce, tylko jaką miał legitymację. Byli posłowie na sejm PRL, byli kombatanci wszelakiego sortu, członkowie ZBoWiD-u, honorowi dawcy krwi i rozmaici partyjni dostojnicy, że o niewiastach przy nadziei nie wspomnimy. Tak więc miejsce w kolejce było ustalane w oparciu o pewne kryterium i najpierw obsługiwany był ten, kto był najrówniejszy z równych. Taką kolejkę nazwiemy właśnie *kolejką priorytetową* (ang. **priority queue**, wymawiaj: *prajority kiu*, z akcentem na *o*).

U nas reguły będą proste: na czele kolejki będzie wstawiany element największy, cokolwiek by to oznaczało. Proponujemy tym razem wstawić do kolejki bohaterów „Kubusia Puchatka”:

```
#include <iostream>
#include <queue>
using namespace std;

int main()
{
  priority_queue<string> P;
  P.push("Winnie-the-Pooh");
  P.push("Christopher Robin");
```

```
  P.push("Piglet");
  P.push("Eeyore");
  P.push("Kanga");
  P.push("Roo");
  P.push("Rabbit");
  P.push("Tigger");
  while(!P.empty())
  {
    cout << P.front() << endl;
    P.pop();
  }
}
```

No i jaką kolejność zobaczymy? Odwrotną, niż alfabetyczną (a raczej leksykograficzną):

```
Winnie-the-Pooh
Tigger
Roo
Rabbit
Piglet
Kanga
Eeyore
Christopher Robin
```

Jasno stąd widać, że konstrukcja tej kolejki ma coś wspólnego z sortowaniem. Kontenerem, który „siedzi pod spodem" jest tym razem `vector<>` i naprawdę ciekawe będzie zbadanie, jak to wszystko działa (bo działa superszybko!) – wrócimy jeszcze do tego!