Lekcja 1: Przegląd algorytmów i elementarnych struktur danych

Wstęp

Zapraszamy do lekcji 1. Pełni ona rolę wprowadzającą do dalszej części podręcznika i składa się z kilku niezależnych rozdziałów.

Zaczniemy od pokazania pseudokodu, za pomocą którego będziemy Wam prezentować algorytmy w większości lekcji. W następnym rozdziale przedstawimy ogólną klasyfikację metod konstruowania algorytmów, połączoną z zapowiedzią tego, co w następnych lekcjach będzie opisane i narysowane w szczegółach.

A na zakończenie pokażemy, jak w nowy i nietypowy sposób można wykorzystać dobrze Wam znane tablice jednowymiarowe. Będziecie mogli "pobawić się" apletami i może od razu nabierzecie ochoty na pisanie własnych projektów na zaliczenie...

Zasady zapisu pseudokodu

Pseudokod, który opracowaliśmy dla naszych lekcji, ma formę wspólną dla obu wersji językowych podręcznika: Pascal/Delphi i C/C++. Jest efektem wielu prób, które wynikły w trakcie pisania lekcji, i w których staraliśmy się dobrać zapis jak najbardziej zwięzły, ale zarazem czytelny i zrozumiały dla osób korzystających z obu języków. Zawiera więc konstrukcje i takie bardzo podobne do C++, i takie "żywcem" wzięte z Pascala - ich wybór wydał się nam najbardziej korzystny. Reguły zapisu pokażemy i skomentujemy dla kolejnych grup konstrukcji językowych; mamy nadzieję, że bez trudu będziecie umieli je we właściwy sposób zapisać w swoim ulubionym języku.

Instrukcje proste, typy zmiennych

- Instrukcje podstawienia zapisywane są ze znakiem równości **=** pełniącym rolę operatora przypisania
- Znak równości wykorzystywany jest też do zapisu relacji równości - jego znaczenie wynika z kontekstu
- Instrukcje nie kończą się średnikami
- Instrukcje napisane w jednej linii rozdziela się przecinkami
- Instrukcje czytania i pisania zapisane są w postaci *wczytaj*(lista zmiennych), *drukuj*(ciąg wyrażeń)
- Instrukcja blokowa (złożona) tworzona jest poprzez ujęcie ciągu instrukcji w **nawiasy klamrowe** { i }, tak jak robi się to w C/C++. Są one odpowiednikiem słów kluczowych **begin end** w Pascalu.
- typy zmiennych są określane słownie, z wyjątkiem zmiennych o nazwach i, j które są zawsze zmiennymi całkowitymi
- Operator dzielenia jest zawsze w postaci *slasha* **/**. W przypadku dzielenia argumentów całkowitych daje wynik obcięty do całkowitego, więc na to należy uważać (zwykle jest o tym informacja)

Instrukcje warunkowe

Instrukcje warunkowe są w naszym pseudokodzie najbardziej oczywiste - używamy słów kluczowych **if** oraz **else**:

```
1. if (warunek)
2.    instrukcja1
3. else                 // tę część można pominąć
4.    instrukcja2
```

Używamy też nawiasów klamrowych, gdy trzeba z kilku instrukcji zrobić jedną:

```
1. if (warunek) {
2.    instrukcja1
3.    instrukcja2
4.    ....
5. }
6. else {
7.    instrukcja_1
8.    ...
9. }
```

Instrukcje pętli

W zależności od potrzeb używamy jednego z trzech typów pętli: **while**, **repeat** oraz **for**:

```
1.  while (warunek)   // dopóki warunek spełniony
2.     instrukcja    // wykonuj instrukcję
3.
4.  while (warunek) {
5.     instrukcja1
6.     instrukcja2
7.     ...
8.  }
```

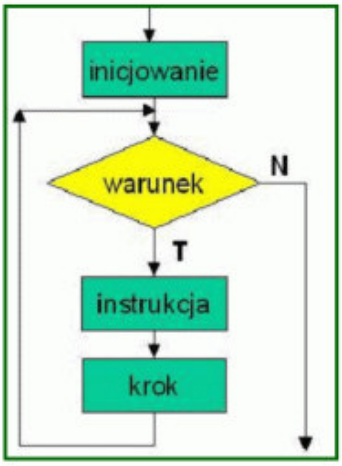
Pętla *repeat* jest "pascalopodobna", jako bardziej naturalna w zapisie; w pętli *do-while* języka C/C++ trzeba będzie warunek po *while* napisać odwrotnie:

```
1.  repeat            // powtarzaj instrukcję, zwykle złożoną
2.     instrukcja
3.  until (warunek) // aż do spełnienia warunku - UWAGA: to jest warunek wyjścia z pętli
```

Więcej jednak problemów mogą mieć "pascalowcy" z pętlą *for*:

```
1.  for (inicjowanie; warunek; krok)
2.     instrukcja
```

Jest to postać zgodna z postacią pętli *for* w C/C++, znacznie ogólniejsza niż pętla *for* w Pascalu. Zarówno część o nazwie *inicjowanie*, jak i *krok* mogą być ciągiem instrukcji, rozdzielonych przecinkami. Mogą też być instrukcjami pustymi. Schemat wykonania pętli *for* przedstawia poniższa sieć działań:



W większości jednak przypadków pętla *for* jest używana w standardowy sposób i szybko się do niej przyzwyczaicie. A za to zapis pseudokodu jest bardziej zwięzły i nie musicie się skupiać na nieistotnych szczegółach.

Funkcje i procedury

Rozróżniamy dwa typy podprogramów:

- funkcje, które zwracają jakąś wartość poprzez swoją nazwę i wywoływane są w wyrażeniach
- procedury, które nie muszą niczego zwracać (a jeśli zwracają, to poprzez parametry) i wywoływane są jak zwykłe instrukcje.

Odpowiednikiem procedur są w języku C/C++ funkcje bez typu (typu **void**)

Nagłówki funkcji i procedur umieszczamy w osobnej linii. Parametry podprogramów podajemy w nawiasach, poprzedzając je informacją o typie i rozdzielając przecinkami. Dodatkowe informacje o typie parametrów niekiedy dodajemy w treści podprogramu. Nawet jeśli podprogram nie ma parametrów, nawiasy przeznaczone na listę parametrów być powinny.

Procedura **NazwaProcedury** (tablica **T**, znak **x**)

```
1.  // T - tablica całkowita o n elementach
2.  ...
3.  // tu treść procedury
```

Nagłówek funkcji zawiera też informację o typie funkcji, umieszczoną za wykazem parametrów (jak w Pascalu) i podkreśloną. Wartość zwracaną podajemy po słowie **return** (jak w C/C++):

Funkcja **NazwaFunkcji** (l. całkowita **a**, znak **x**) typu znakowego

```
...
// tu treść funkcji
return wartosc_zwracana
```

Przegląd metod konstruowania algorytmów

Jak doskonale wiemy, tworzenie projektów informatycznych opiera się w dużej mierze na formułowaniu i implementacji algorytmów, które mają za zadanie właściwe przetworzenie danych i rozwiązanie postawionych przed nami problemów. Algorytmy można sklasyfikować na kilka różnych sposobów,ale wśród nich najważniejszy jest podział ze względu na techniki ich konstruowania. Są pewne techniki tworzenia algorytmów, których zastosowanie prowadzi do efektywniejszego rozwiązywania problemów niż za pomocą algorytmów konstruowanych (nieraz bardzo błyskotliwie) w sposób „spontaniczny" (choć może właściwszym określeniem byłoby użycie pojęcia *ad hoc*) i należy je poznać, zanim zabierzemy się do samodzielnego tworzenia algorytmów.

Do najważniejszych i zarazem najczęściej używanych technik konstruowania algorytmów należą:

- metody typu „dziel i zwyciężaj"
- algorytmy bazujące na programowaniu dynamicznym
- algorytmy zachłanne
- algorytmy z powrotami
- metody heurystyczne

Postaramy się w tym podrozdziale omówić wyżej wymienione grupy, a także wspomnimy, jakie różnice w budowie i podejściu do problemu występują miedzy nimi i powiemy, w którcy lekcjach będą one dokładniej omówione.

Metody typu „dziel i zwyciężaj"

Algorytmy z tej grupy (ang. *divide and conquer* –D&C) zaliczają się do najbardziej skutecznych metod rozwiązywania problemów. Algorytm jawnie oparty na zasadzie „dziel i zwyciężaj" został po raz pierwszy opracowany przez A. **Karatsubę** w 1960r. – mowa jest o metodzie mnożenia dwóch dużych liczb.

Zastosowana w nich strategia polega na prostej idei :

- dzielimy nasz początkowy problem na kilka podproblemów o podobnej (a najczęściej identycznej) strukturze co problem pierwotny
- tak uzyskane podproblemy dzielimy ponownie na mniejsze fragmenty (zazwyczaj rekurencyjnie) tak długo, aż uzyskamy problem o wielkości umożliwiającej niemalże natychmiastowe jego rozwiązanie
- otrzymane w ten sposób wyniki łączymy w rozwiązania dotyczące „rodziców" tych podproblemów, co ostatecznie prowadzi do otrzymania rozwiązania całościowego problemu.

Naturalnym sposobem implementowania algorytmów tej grupy jest użycie rekurencji (zwanej czasem rekursją). Zasady stosowania tej techniki programowania poznacie dokładnie w następnej lekcji; w skrócie polega ona na tym, że w kodzie funkcji (podprogramu) jest wywołanie tej samej funkcji (a więc wywołuje ona samą siebie). Jak widzimy, jest to bardzo podobne do zagadnienia podziału problemu na podproblemy, które z kolei są dzielone na swoje podproblemy itd... Dlatego też jest rzeczą naturalną, że metody „dziel i zwyciężaj" w bardzo przejrzysty sposób można zapisać z zastosowaniem rekurencji – niemniej jednak dosyć często skuteczniejszy jest iteracyjny zapis algorytmu.

Klasycznymi przykładami algorytmów D&C są metody sortowania szybkiego (*QuickSort*) oraz sortowania przez łączenie(*MergeSort*), a także przeszukiwania binarnego – wszystkie wymienione tu metody zostaną dokładnie omówione w dalszej części podręcznika.

Algorytm przeszukiwania binarnego często jest zaliczany do pewnego wariantu zasady D&C, a mianowicie określanego w wolnym tłumaczeniu mianem „zmniejszaj i zwyciężaj" - stosując tę metodę dochodzi się tylko do jednego podproblemu, którego rozwiązanie jest zarazem rozwiązaniem problemu właściwego.

Jakie są główne zalety algorytmów D&C? Spróbujmy je wymienić:

- są to wydajne metody do rozwiązywania złożonych problemów, które mogą być rozłożone na proste podzadania, dla których jesteśmy w stanie w nietrudny sposób znaleźć rozwiązanie
- efektywność algorytmów „dziel i zwyciężaj" pozwala na zmniejszenie złożoności obliczeniowej potrzebnej do rozwiązania problemu – niejednokrotnie redukują złożoność obliczeniową wielomianową (w przypadku zwykłego algorytmu) do czasu logarytmiczno-liniowego O(n lg n). Zagadnienia złożoności obliczeniowej przybliżymy w lekcji 2.
- stosowanie D&C jest niezmiernie użyteczne przy uruchamianiu programów na maszynach równoległych, gdy wiele procesorów może jednocześnie rozwiązywać pojedyncze podproblemy.

Natomiast główną wadą tego rodzaju algorytmów jest spowolnienie ich działania z powodu stosowania rekurencji, która wymaga użycia stosu odwołań do pamięci komputera – w niektórych przypadkach zysk czasowy osiągnięty poprzez zastosowanie

„dzielenia i zwyciężania" jest mniejszy, niż strata spowodowana działaniem rekurencji. To wszystko zrozumiecie dokładnie po przestudiowaniu lekcji 2, prawie w całości poświęconej rekurencji.

Programowanie dynamiczne

Programowanie dynamiczne (optymalizacja dynamiczna, PD) jest strategią tworzenia algorytmów przeznaczoną przede wszystkim do rozwiązywania zagadnień natury optymalizacyjnej (w skrócie – gdy mamy zadanie minimalizacji bądź maksymalizacji jakiejś wartości).

Zasada stosowana w optymalizacji dynamicznej jest pewnego rodzaju modyfikacją metody „dziel i rządź" – opiera się na jednokrotnym rozwiązywaniu tych samych podproblemów. Nierzadko zdarza się, że pewien podproblem niższego poziomu jest elementem składowym kilku podproblemów wyższych poziomów. Zatem redukcja czasu rozwiązania zadania polega na jednokrotnym tylko rozwiązywaniu podproblemów i zapamiętywaniu odpowiednich wartości rozwiązań, które będą mogły być wielokrotnie użyte przy rozwiązywaniu „większych" podproblemów. Jak łatwo zauważyć, mniejsza liczba wykonanych obliczeń prowadzi do obniżenia złożoności obliczeniowej algorytmu rozwiązującego dany problem – istnieją przypadki, gdy redukcja czasu wykonania metody jest znacząca – z wartości proporcjonalnej wykładniczo do rozmiaru zadania do złożoności algorytmu realizowanego w czasie wielomianowym.

Najczęściej rozwiązania podproblemów w programowaniu dynamicznym są zapisywane w pamięci komputera (na przykład w tablicy jedno- lub dwuwymiarowej), a następnie w momencie potrzeby rozwiązania danego podproblemu – jeśli już było wcześniej rozwiązane - zamiast konieczności rekurencyjnych wywołań funkcji odwołujemy się do uprzednio zapamiętanej w pamięci wartości rozwiązania. Powyższa koncepcja rozwiązywania zadań gwarantuje obniżenie „czasu pracy" algorytmu – należy rozwiązywać podproblemy od najmniejszych do największych (co oznacza, że optymalny koszt (rozwiązanie) jest obliczany metodą wstępującą), co zapewnia rozwiązywania konkretnego podproblemu tylko raz. Należy jednak pamiętać, że niezmiernie istotne jest określenie na początku algorytmu równania rekurencyjnego określającego wartość rozwiązania problemu na poziomie wyższym w postaci funkcji zależności od rozwiązań podproblemów na poziomie niższym.

Zatem kolejność działań w algorytmach stosujących programowanie dynamiczne można przedstawić następująco:

- określamy sposób podziału zadania na podproblemy
- poszukujemy równania rekurencyjnego określającego wartość rozwiązania problemu zależną od wartości rozwiązań jego podproblemów
- rozwiązujemy podproblemy metodą wstępującą (zaczynając od najmniejszych), aż do momentu uzyskania kosztu optymalnego zadania pierwotnego (całościowego).

Do tej pory nic nie mówiliśmy o typowych zastosowaniach programowania dynamicznego. Otóż wzorcowymi przykładami wykorzystania PD są zadania polegające na:

- rozwiązaniu problemu „plecakowego"
- obliczeniu wyrazów ciągu Fibonacciego lub kolejnych wartości symbolu Newtona
- znalezieniu najkrótszych ścieżek między parami węzłów grafu – algorytm **Floyda-Warshalla**

Problemem plecakowym zajmiemy się już w lekcji następnej, natomiast algorytmowi znajdowania najkrótszych ścieżek poświęcona będzie lekcja 7.

Algorytmy zachłanne

Rozwiązywanie zadań przy zastosowaniu algorytmów zachłannych (ang. *greedy algorithms*) polega na bardzo prostej zasadzie – w danym kroku (momencie) algorytm wybiera (w sposób zachłanny) to rozwiązanie, które wydaje się w danej chwili najkorzystniejsze - w sposób niezależny od wyborów dokonanych wcześniej oraz perspektywy znalezienia się przed przyszłymi wyborami (mniej lub bardziej korzystnymi). Oznacza to, że algorytm „nie patrzy się za siebie" ani nie „spogląda w przyszłość" – wybiera to, co wydaje mu się lokalnie optymalne (tzw. decyzja lokalnie optymalna). Tak, tak, ważne jest to słowo – „optymalne" – gdyż najczęściej algorytmy zachłanne są stosowane do rozwiązywania zagadnień optymalizacyjnych – szczególnie wtedy, gdy zależy na szybkości znalezienia rozwiązania. Ceną, jaką płacimy za ową szybkość, jest brak gwarancji znalezienia poprawnego (dopuszczalnego) wyniku - nie mówiąc już o optymalności tego rozwiązania.

Drugim częstym obszarem zastosowań algorytmów zachłannych jest dziedzina sztucznej inteligencji – zwyczajowo w tym przypadku tego typu algorytmy przyjmują nazwę metod „podchodzenia pod górę". Nazwa pochodzi od tego, że podobnie jak w przypadku prawdziwego szczytu, kroczymy pod górę - w momencie gdy wykonamy krok „schodzenia" w kierunku spadku terenu, to zatrzymujemy się i stwierdzamy, że dotarliśmy do szczytu. Może się jednak okazać, że jest to tylko lokalne wzniesienie, a prawdziwy szczyt znajduje się jeszcze szmat drogi przed nami. Ale przecież w końcu – tłumaczymy sobie - też stanęliśmy na jakimś mniejszym wzniesieniu, więc również są zalety bycia na nim – widoki może nie takie wspaniałe, jak z właściwego szczytu, ale mimo wszystko stąd panorama też jest ładna. Jak widzimy, algorytm zachłanny nie zapewnia nam wyszukania optimum globalnego, ale przynajmniej w prosty sposób jesteśmy w stanie zlokalizować optimum lokalne.

Przykładami zastosowania podejścia zachłannego są takie algorytmy, jak m.in.:

- algorytm Dijkstry – służący do znajdowania najkrótszych ścieżek w grafach
- metoda Kruskala – używana do wyznaczania minimalnego drzewa rozpinającego

- konstrukcja drzewa kodowego Huffmana służącego do kompresji danych
- algorytmy szeregowania zadań

Pierwsze dwie metody zostaną szerzej omówione w jednym z ostatnich rozdziałów podręcznika opisującym grafy i algorytmy działające na nich. Okazuje się, że zarówno algorytm Dijkstry, jak i Kruskala, znajdują zawsze rozwiązania optymalne – co świadczy o właściwym miejscu zastosowania reguły „zachłannej".

Algorytmy z powrotami

Czasem może nam się zdarzyć, że natrafimy na problem, którego rozwiązywanie będzie najefektywniejsze, jeśli zamiast stosowania specjalizowanych technik użyjemy reguły znanej nam wszystkim z życia codziennego pod nazwą „metody prób i błędów". Algorytmy odnoszące się do tej reguły określamy mianem algorytmów z powrotami (ang. *backtracking algorithms*).

Otóż wyobraźmy sobie nasz problem jako ciąg kroków, które należy wykonać, by dojść do celu. Każdy krok, który wykonamy, będziemy zapamiętywać – dzięki temu szybciej znajdziemy drogę do celu. Najpierw próbujemy wykonać pierwszy dopuszczalny krok. Jeśli znajdziemy się w miejscu, gdzie znów mamy możliwość wyboru „dalszej trasy", to po raz kolejny decydujemy się na postawienie następnego kroku. Wykonujemy tę czynność cyklicznie do momentu, aż wykonamy krok niedopuszczalny (np. znajdziemy się w „ślepym zaułku") – w tej sytuacji cofamy się o 1 krok do tyłu, zaznaczamy ten krok jako niewłaściwy (byśmy w przyszłości wiedzieli, żeby nie próbować iść tą drogą) i wybieramy inny „kierunek marszu" z aktualnego miejsca, w którym się znajdujemy. W momencie gdy wszystkie kierunki wychodzące z danego miejsca się wyczerpią, cofamy się ponownie o jeden krok i sprawdzamy nowe możliwości. Algorytm wykonuje więc ruchy naprzód i kroki do tyłu w poszukiwaniu poprawnego rozwiązania. Dlatego do naturalnego sposobu zapisu tego typu metod rozwiązywania problemów jest stosowana rekurencja.

Istnieje wiele różnych zagadnień, do rozwiązania których stosuje się algorytmy z powrotami, m.in.:

- problem „plecakowy" (optymalnego wyboru)
- grupa problemów dotyczących gier (w szachach, warcabach), np.:
  - problem rozmieszczenia n-królowych na szachownicy (znany najczęściej w wersji z 8 hetmanami (królowymi))
  - problem znalezienia drogi konika szachowego (odwiedzenie każdego pola szachownicy dokładnie raz) – *Knight's Tour*
- problemy „grafowe", np.:
  - problem kolorowania grafu (w taki sposób użyć m kolorów, by każde sąsiednie węzły były różnego koloru)

Ponieważ problem rozmieszczenia 8 hetmanów na szachownicy jest bardzo często poruszany w literaturze naukowej, więc dla odmiany w rozdziale 2 przedstawimy inny, równie interesujący przykład zadania z dziedziny szachowej.

Metody heurystyczne

Ostatnią grupą metod, którą chcielibyśmy omówić, są metody heurystyczne (heurystyki) - słowa te pochodzą od greckiego *Eureka*). Główna idea działania metod tego typu polega na szacowaniu rozwiązania podproblemów w „inteligentny" i zdroworozsądkowy sposób. Mówiąc "inteligentny", mamy na myśli oszacowanie na podstawie czasem niepełnych danych, jaka może być wartość wynikowa danego podproblemu (znając pewne fakty formułujemy hipotetyczne rozwiązanie) – na przykład poprzez uproszczenie pewnego modelu do minimum, zrelaksowanie ograniczeń podzadania czy przyjęcie typowych parametrów danych. Stosując te metody nie mamy pewności, ze znajdziemy optymalne rozwiązanie. Jednakże istnieją sytuacje, w których zastosowanie „normalnego" algorytmu powoduje bardzo duże koszty (czasowe i obliczeniowe) znalezienia właściwego rozwiązania, a w skrajnym przypadku nie znajduje go wcale bez użycia elementów heurystyki. Należy w tym miejscu zaznaczyć, że podejście algorytmiczne tym różni się od heurystycznego, że stosując algorytmy uruchamiamy pewien regularny, jednoznacznie określony proces gwarantujący nam znalezienie właściwego rozwiązania. Z kolei heurystyki są procesami twórczymi, zbiorami wskazówek i hipotez, które nie zapewniają wyszukania poprawnego rozwiązania.

Metodami heurystycznymi posługujemy się najczęściej w problemach, gdy brakuje nam pewnych danych, dzięki którym bylibyśmy w stanie znaleźć rozwiązanie przy użyciu klasycznych algorytmów.

Przykładami takich problemów są chociażby:

- prognozowanie pogody
- wykrywanie wirusów i robaków internetowych

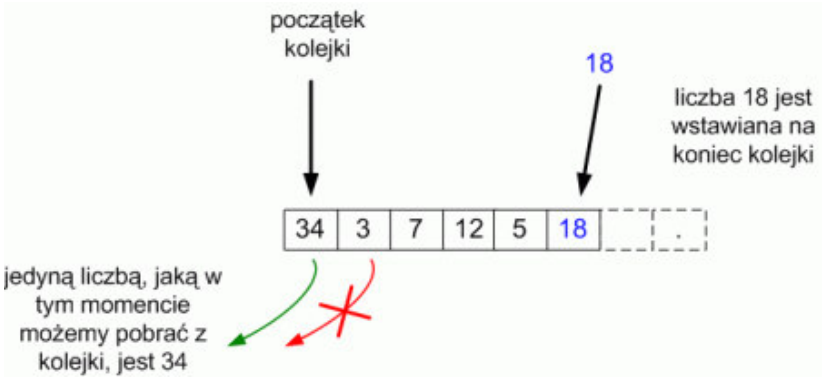
Oczywiście zastosowanie „czystej" metody heurystycznej nie doprowadzi nas na wyszukania prawidłowego rozwiązania problemu. Najczęściej stosuje się algorytmy łączone, gdzie pewne „oszacowane" rozwiązanie przybliżone jest wyznaczone metodami heurystycznymi w celu przyspieszenia działania programu – innymi słowy, heurystyki nakreślają kierunek, w którym należy prowadzić obliczenia za pomocą „klasycznej" części algorytmu. Przy takiej koncepcji tworzenia algorytmów bardzo często okazuje się, że jest ona zadowalająca - uzyskuje się redukcję czasu obliczeń przy braku utraty jakości rozwiązania. Jako przykład możemy przytoczyć bardzo ciekawy algorytm A* służący do znajdowania najkrótszej ścieżki w grafie – w którym wykorzystana jest metoda heurystyczna do szacowania odległości węzła od punktu (węzła) docelowego. Szczegóły tego algorytmu, stosowanego w grach komputerowych, i specjalnie napisany dla Was aplet, poznacie w lekcji 7 podręcznika.

Stosy i kolejki jako elementarne struktury danych

Kolejka, czyli struktura FIFO

Kolejka (ang. *queue*) jest jednym z podstawowych abstrakcyjnych typów danych. Oprócz kolejki równie ważną liniową strukturą danych jest stos, o którym także dowiecie się w tym rozdziale. W tej chwili spróbujemy przybliżyć najważniejsze właściwości kolejki, która jest często określana mianem **FIFO** (First In - First Out), co wolnym tłumaczeniu znaczy "pierwszy wchodzi - pierwszy wychodzi".

Otóż kolejka jest rodzajem struktury przechowującej zbiór danych. Do tego zbioru mogą być dodawane nowe elementy, jak również "wyjmowane" z niego. W przypadku składowania danych w strukturze FIFO, nowy element jest wstawiany na koniec kolejki, natomiast pobieranie danych (wyjmowanie) z kolejki jest możliwe jedynie poprzez wyciągnięcie elementu znajdującego się na jej początku.



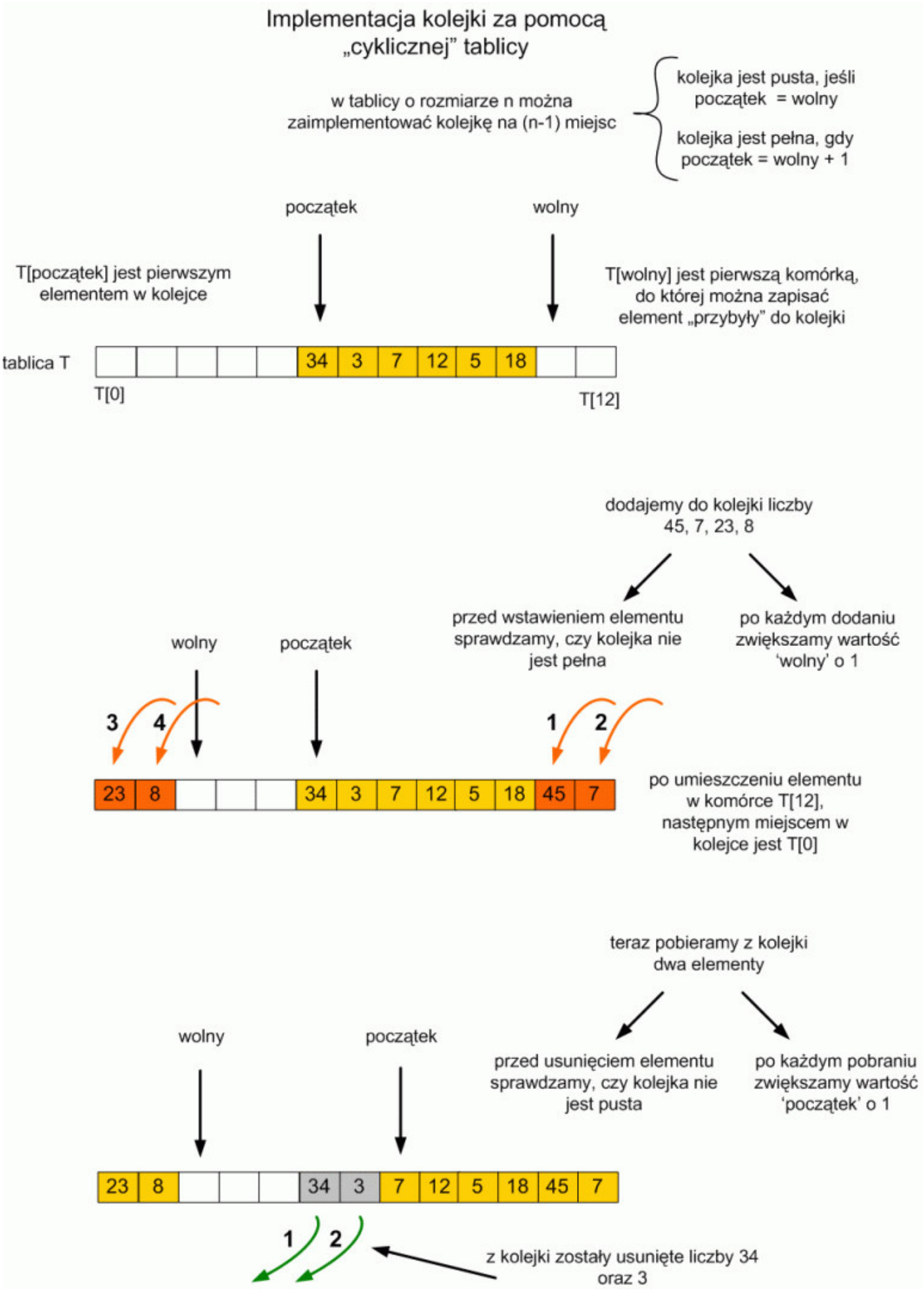
Tak więc jedyne możliwe operacje do wykonania na kolejce to:

- dodanie elementu na koniec kolejki
- usunięcie (pobranie) pierwszego elementu z kolejki

Struktura FIFO jest przybliżonym odzwierciedleniem kolejek występujących w życiu codziennym - często sami stajemy się "elementami" różnych kolejek, np. do kasy w kinie czy do "okienka" na poczcie. Zdajemy sobie sprawę, że wszystkie osoby stojące przed nami w kolejce zostaną "obsłużone" (np. kupią bilety), zanim my będziemy mieli taką szansę.

Na podobnej zasadzie działa kolejka FIFO. Należy pamiętać, że w momencie pobrania elementu z początku kolejki, element znajdujący się do tej pory za nim teraz staje na jej "czele". Podobnie każdy inny element w kolejce przesuwa się o jedno miejsce do przodu.

Najczęstszym sposobem implementacji struktur FIFO w programach jest użycie tablic (zazwyczaj w postaci tzw. bufora cyklicznego) bądź list. Zupełnie nowe dla Was struktury listowe będą tematem lekcji 4, natomiast ideę bufora cyklicznego wyjaśnia szczegółowo poniższy rysunek. Dodatkowy komentarz jest raczej zbyteczny - tablice są Wam dobrze znane z lekcji programowania. Warto tylko zwrócić uwagę na to, że w buforze cyklicznym elementy w tablicy się nie przesuwają (byłoby to bardzo nieekonomiczne) - przesuwają się tylko indeksy (wskaźniki) pokazujące początek i koniec kolejki. Nie należy więc bufora cyklicznego kojarzyć z cyklicznym przesuwaniem elementów w tablicy, znanym Wam z ćwiczeń na lekcjach programowania. Tu następuje cykliczne zapełnianie tablicy:



Zauważcie tylko przyglądając się rysunkowi, że w pełniącej rolę bufora cyklicznego tablicy, która ma miejsce na n elementów, zawsze jedno miejsce pozostaje nie wykorzystane, dzięki czemu możemy odróżnić stan, kiedy kolejka jest pusta (indeksy

poczatek i wolny sa sobie równe) od stanu jej zapełnienia (poczatek=wolny+1).

Struktura kolejki jest wykorzystywana w wielu algorytmach komputerowych, a jednym z najważniejszych zastosowań jest kolejkowanie procesów oczekujących na wykonanie na procesorze komputera.

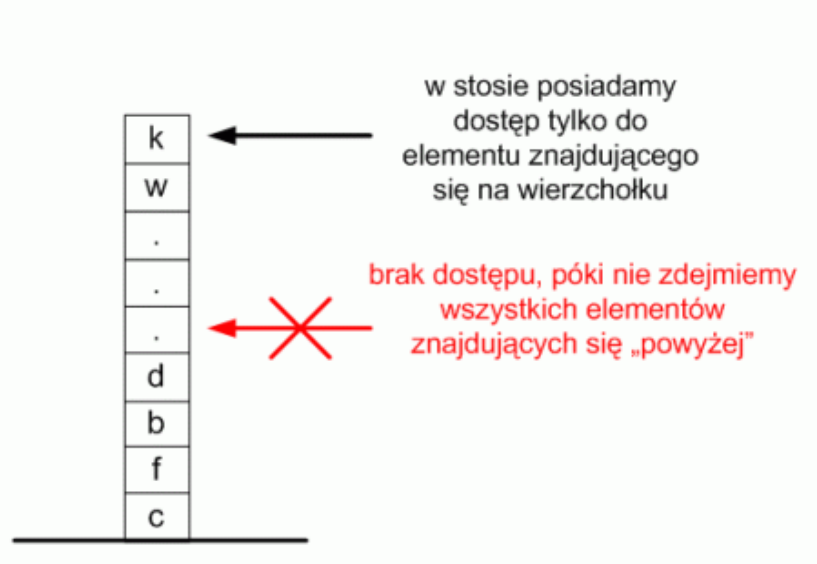
W lekcji 6. podręcznika dowiecie się o pewnym szczególnym typie kolejek - o kolejkach priorytetowych. Odwołując się do "życiowych" zagadnień, można to porównać do kolejki składającej się ze zwykłych klientów oraz "ważniejszych" osób, które, mimo że przyszły po nas, to zostaną obsłużone przed nami - oznacza to tyle, że te osoby mają wyższy priorytet od zwykłych ludzi. Tak właśnie w przybliżeniu funkcjonują kolejki priorytetowe.

Stos, czyli struktura LIFO

Stos (ang. *stack*) jest jedną z najważniejszych struktur danych używanych w algorytmach komputerowych. Podstawowe cechy można zawrzeć w stwierdzeniu "ostatni wchodzi - pierwszy wychodzi", czyli w skrócie **LIFO** (Last In – First Out). Hasło podobne brzmiące jak w przypadku struktury kolejki, jednak zasada przetwarzania danych jest zgoła przeciwna. Tam dane były przetwarzane w kolejności pojawienia się na wejściu, a w przypadku stosu są one „obsługiwane” dokładnie w odwrotnej kolejności – rekordy, które pojawiły się jako pierwsze, będą wykorzystywane jako ostatnie, z kolei dane, które „dotarły” do wejścia jako ostatnie, w pierwszej kolejności zostaną przetworzone.

Obsługa stosu wygląda następująco: dostęp do danych ułożonych na stosie mamy umożliwiony tylko od góry – możemy sobie wyobrazić, że mamy przykładowo stos kart do gry. Kładziemy karty jedna na drugiej, potem kolejną, i jeszcze następną. Gdy przyjdzie nam ochota zobaczyć pierwszą z położonych kart, musimy najpierw zdjąć jedną kartę położoną jako ostatnia, potem jeszcze kilka kolejnych aż w końcu „dobrniemy” do interesującej nas karty. Musieliśmy się dostać aż na sam spód tego stosu. Niebawem wyjaśnimy, na czym polegają zalety takiego ułożenia danych.

Poniższy rysunek przedstawia ogólną strukturę stosu:

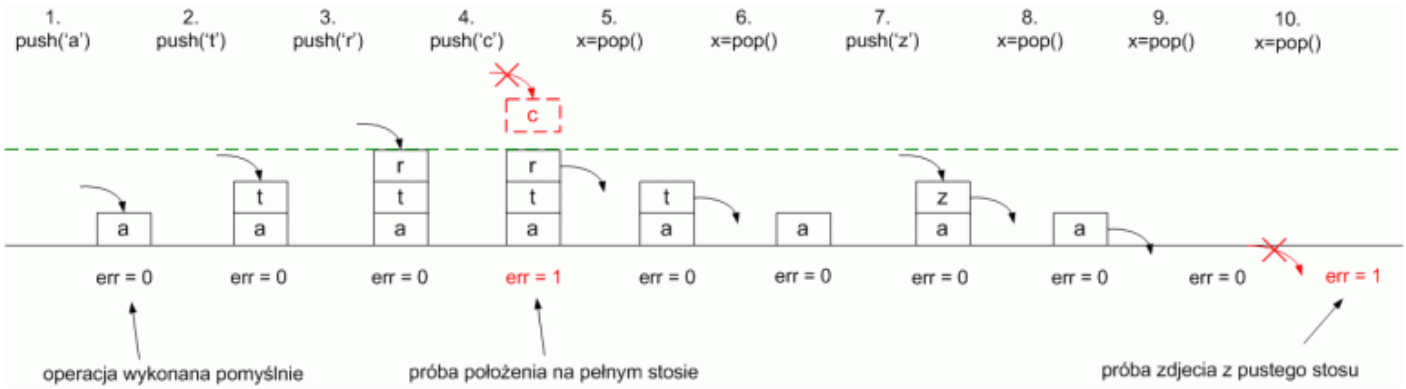


Zazwyczaj przy implementacji stosu definiuje się dwie funkcje operujące na nim - najczęściej są nazywane w ten sposób:

- **push**(x)
- **pop**()

Pierwsza z nich ma za zadanie odłożenie wartości zmiennej x na stos, natomiast funkcja pop pobiera argument leżący na wierzchu stosu i zwraca jego wartość. Podczas implementacji obu tych funkcji należy pamiętać o właściwej obsłudze błędów. Przed położeniem elementu na stosie należy sprawdzić, czy nie jest on pełny - jeśli tak, to program powinien sygnalizować błąd przepełnienia stosu. Podobnie jest w przypadku wywołania funkcji pop - na samym początku ciała funkcji powinno się stwierdzić, czy dany stos jest pusty - w razie odpowiedzi twierdzącej wypadałoby poinformować o błędzie niedomiaru.

W celu pokazania zasad działania funkcji push i pop zamieszczamy poniżej schemat:



Pojemność przedstawionego stosu wynosi 3 miejsca (zaznaczona zieloną przerywaną prostą). Jako argument funkcji push przekazujemy w tym przykładzie stałą znakową – oczywiście można przekazywać zmienną – wtedy na stosie „odkładana" jest jej wartość. Natomiast funkcja pop zwraca wartość (element zdjęty ze stosu), którą zapisujemy pod zmienną x.

> Implementacja struktury stosu musi być opracowana w taki sposób, by dostęp do elementów na stosie był jedynie poprzez jego wierzchołek – w przypadku programowania obiektowego jest to szczególnie łatwe: tworzy się klasę stos zawierającą prywatną zmienną tablicową przechowującą elementy na stosie. Jedynie metody klasy push i pop powinny mieć możliwość zmiany zawartości stosu.

Stos znakowy - tworzenie odwrotnej notacji polskiej (ONP)

Jak już wcześniej wspominaliśmy, stos jest taką strukturą danych, która skutecznie pomaga zwiększyć efektywność niektórych algorytmów. Pokażemy to na przykładzie obliczeń związanych z Odwrotną Notacją Polską (**ONP**), znaną w światowej literaturze pod skrótem **RPN** – *Reverse Polish Notation*.

Warto na wstępie wyjaśnić podstawy tej notacji. Obecnie najczęściej używaną notacją znaną nam wszystkim jest notacja **infiksowa** (inaczej: wrostkowa), która jest konwencjonalnym zapisem algebraicznym. Operator działania arytmetycznego znajduje się pomiędzy argumentami, np. wyrażenia **3+5*4** i **(3+5)*4** są zapisane właśnie w notacji infiksowej. Nawiasy pozwalają zmienić kolejność wykonywania operacji wynikającą z priorytetów operatorów.

Zarówno ONP, jak i NP (Notacja Polska, opracowana przez polskiego naukowca Jana Łukasiewicza), z której ONP się wywodzi, są notacjami beznawiasowymi. NP, w której operator występuje przed argumentami operacji (czyli jest to zapis przedrostkowy, **prefiksowy**), w sposób jednoznaczny określa kolejność wykonywania działań i nie ma potrzeby wprowadzania nawiasów. To samo dotyczy ONP, która jest zapisem przyrostkowym (**postfiksowym**) – tutaj operator działania występuje po argumentach. Obecnie NP i ONP są wykorzystywane praktycznie już tylko w informatyce. I właśnie w informatyce strukturą danych najczęściej wykorzystywaną przy algorytmach konwersji z notacji konwencjonalnej do ONP oraz obliczania wartości wyrażenia zapisanego w ONP jest właśnie stos.

Spróbujmy przyjrzeć się tym algorytmom.

Najpierw warto prześledzić istotę działania metody konwersji wyrażenia zapisanego w notacji wrostkowej na zapis ONP. Zajmiemy się jej wersją opracowaną przez E. Dijkstrę, która jest określana potocznie jako „stacja rozrządowa". Otóż w tym algorytmie będziemy potrzebowali wejścia i wyjścia – dwóch zmiennych będących łańcuchem znaków, a także zmiennej tablicowej reprezentującej stos, na którym będziemy odkładali znaki operatorów oczekujące na późniejsze „przełożenie" ich do ciągu wyjściowego.

Słowny opis algorytmu przetwarzania jednego znaku pobranego z wejścia można w skrócie przedstawić tak:

1. jeżeli znak odczytany z wejścia jest cyfrą, dodaj go do łańcuchu wyjściowego
2. jeżeli znak jest operatorem, to:
   - jeśli na szczycie stosu znajduje się operator o wyższym lub równym priorytecie od odczytanego z wejścia – zdejmij go ze stosu i dodaj do kolejki wyjściowej, aż do momentu, gdy na (nowym) szczycie stosu znajdzie się operator o niższym priorytecie, wtedy:
   - odłóż odczytany operator na stos – tak samo postąp, gdy na stos będzie pusty (od razu bądź też po zdjęciu ze stosu innych operatorów)
3. jeżeli znak jest lewym nawiasem, to odłóż go na stos, natomiast
4. jeżeli znak jest prawym nawiasem, to zdejmuj po kolei operatory ze stosu i przekazuj na wejście aż do momentu, gdy na szczycie stosu nie znajdzie się lewy nawias – wtedy zdejmij nawias z wierzchołka i żadnego z nawiasów nie kładź na stosie

Algorytm przetwarzania zapisu infiksowego na ONP kończy się, jeżeli wszystkie znaki z wejścia zostały odczytane i przetworzone. W sytuacji, gdy stos jest niepusty, należy przekazać pozostałe znaki operatorów na wyjście.

Zapiszemy jeszcze raz to samo w pseudokodzie - jeden krok przetworzenia znaku pobranego z wejścia:

Procedura **Przetworz**(znak **aktualny_znak**)

```
1.  // x - znak
2.  // znak_na_szczycie - znak
3.  if(aktualny_znak jest cyfra)
4.       zapisz_na_wyjscie(aktualny_znak)
5.  else if(aktualny_znak jest znakiem operatora) {
6.     while (znak_na_szczycie jest operatorem i priorytet(aktualny_znak) <= priorytet(znak_na_szczyc
7.          x=pop()              // zdejmij znak ze szczytu stosu
8.          zapisz_na_wyjscie(x)
9.       }
10.      push(aktualny_znak)
11. }
12. else if(aktualny_znak jest lewym nawiasem)
13.      push(aktualny_znak)
14. else if(aktualny_znak jest prawym nawiasem) {
15.      x=pop()
16.      while(x nie jest lewym nawiasem)
17.          zapisz_na_wyjscie(x)
18.      x=pop()
19. }
```

Powyższy algorytm wykonuje się poprawnie dla liczb całkowitych, zarówno jedno-, jak i wielocyfrowych. W sytuacji odczytywania z wejścia kolejnych cyfr liczby, znaki te są w poprawnej kolejności przenoszone do łańcucha wyjściowego.

Tabela priorytetów dla czterech podstawowych operatorów arytmetycznych wygląda następująco:

Operator	**Priorytet**
+ -	1
* /	2

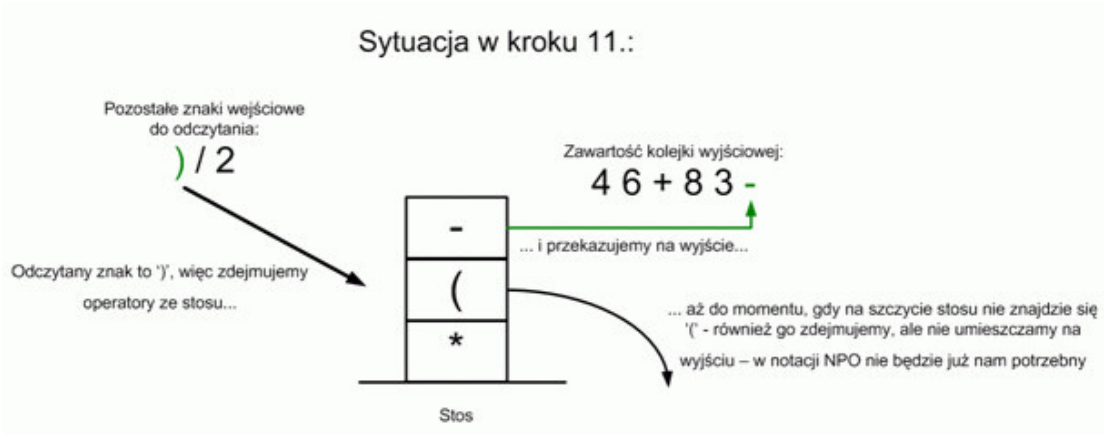
Warto teraz prześledzić poniższy przykład.

Ciąg znaków wejściowych: **(4+6)*(8-3)/2**

Nr kroku	**Znak wejściowy**	**Zawartość stosu***	**Znak przekazywany na wyjście**
1	(	(	
2	4	(	4
3	+	+ (	
4	6	+ (	6
5	)		+
6	*	*	
7	(	(*	
8	8	(*	8
9	-	- (*	
10	3	- (*	3
11	)	*	-
12	/	/	*
13	2	/	2
14. Koniec znaków			/

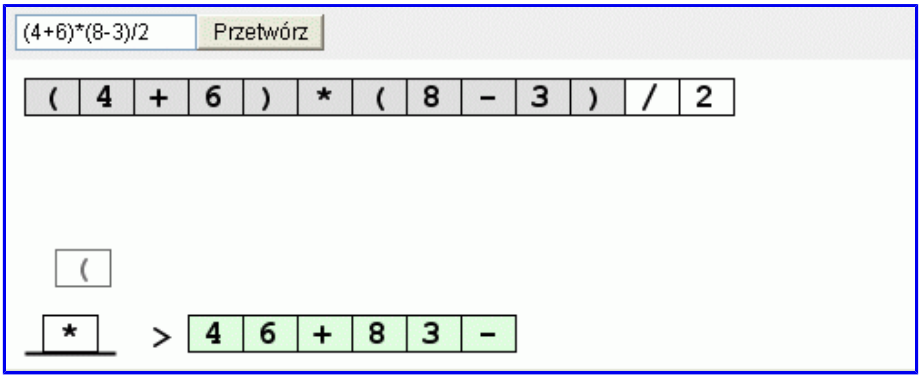
** skrajnie lewy symbol oznacza szczyt stosu, oprócz tego zakładamy, że stos jest typu znakowego*

Ciąg znaków wyjściowych: **4 6 + 8 3 - * 2 /**



Podczas implementacji należy pamiętać o tym, by po dopisaniu do łańcuchu wyjściowego operatora bądź też ostatniej cyfry liczby umieścić znak biały ' ', dzięki któremu unikniemy „złączenia" się dwóch liczb w jedną (w naszym przykładzie musimy zapisać „4 6 + 8 3 - * 2 /", a nie „46+83-*2/").

Na zakończenie zapraszamy do zabawy poniższym apletem (wejście przez kliknięcie w obrazek)- wpiszcie dowolne wyrażenie typu opisanego powyżej i zobaczcie, jak przy użyciu stosu znakowego powstaje z niego ONP:



Stos liczbowy - obliczanie wartości wyrażeń zapisanych w ONP

Wiemy już, jak przetworzyć zapis w notacji konwencjonalnej do postaci ONP. Za chwilę dowiemy się, jak za pomocą stosu - tym razem liczbowego - w szybki sposób obliczać wartość wyrażenia przedstawionego właśnie w zapisie postfiksowym.

Okazuje się, że taki algorytm może być naprawdę prosty w zapisie, a przy tym efektywny. Tym razem jest nam potrzebny ciąg wejściowy będący zapisem wyrażenia w notacji ONP oraz stos pozwalający na przechowywanie liczb. Wynik wyrażenia jest zdejmowany ze stosu na samym końcu.

Kroki algorytmu są następujące:

1. jeżeli znak odczytany z wejścia jest cyfrą, to pobieraj kolejne znaki budując z nich liczbę, aż natrafisz na znak nie będący cyfrą - wtedy otrzymaną liczbę odłóż na stos
2. jeżeli znak jest operatorem, to
   - zdejmij ze stosu dwie liczby (pierwszą zdjętą niech będzie x, drugą y)
   - oblicz wartość wyrażenia y operator x
   - odłóż obliczoną wartość na stos
3. gdy już skończą się znaki w ciągu wejściowym, zdejmij ze stosu końcowy wynik

Ponownie spróbujmy zapisać w postaci pseudokodu jeden krok algorytmu odpowiadający przetworzeniu pobranego z łańcucha ONP ciągu cyfr lub znaku operatora.

Procedura **Oblicz**()

```
1.  // aktualny znak - znak
2.  // cyfra, liczba, x, y - całkowite
3.  wczytaj_z_wejścia(aktualny_znak)
4.  if (aktualny_znak jest cyfrą) {
5.     liczba = 0
6.      while (aktualny_znak jest cyfrą) {
7.          zamień ten znak  na jego wartość liczbową (zmienna cyfra)
8.          liczba =10*liczba+cyfra          // budujemy liczbę z kolejnych cyfr
9.          wczytaj_z_wejścia(aktualny_znak)
10.     }
11.     push(liczba)
12. }
13. else if(aktualny_znak jest znakiem operatora) {
14.     x=pop()
15.     y=pop()
16.     z=oblicz(y operator_odpowiadający_aktualnemu_znakowi x)
17.     push(z)
18. }
```

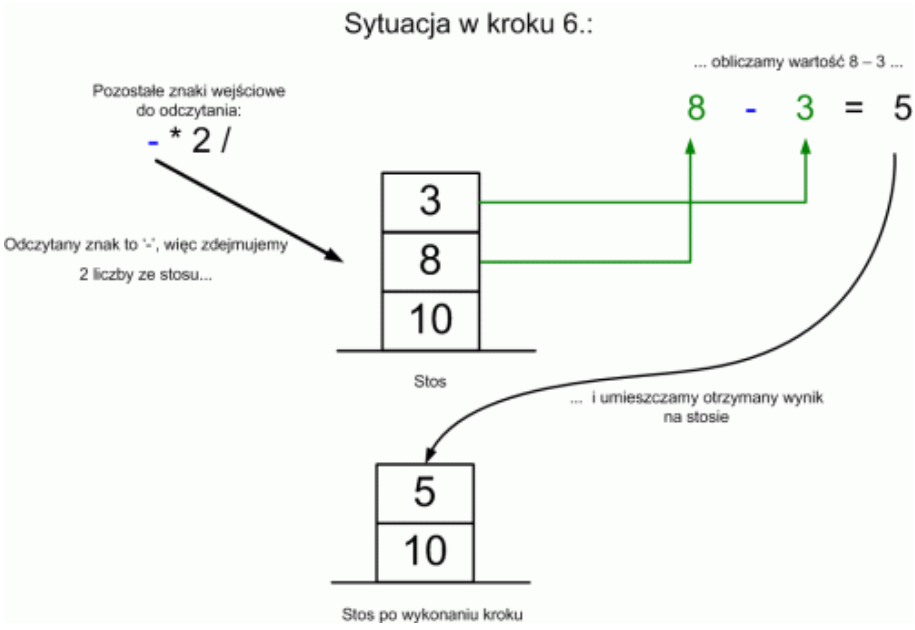
Kontynuując nasz przykład, weźmiemy ciąg znaków otrzymany na wyjściu poprzedniego algorytmu, który teraz stanie się argumentem funkcji obliczającej wyrażenia arytmetyczne zapisane w notacji ONP.

Ciąg znaków wejściowych: **4 6 + 8 3 - * 2 /**

Nr kroku*	Odczytany operator bądź liczba	Zawartość stosu**	WYNIK
1	4	4	
2	6	6 4	
3	+	10	
4	8	8 10	
5	3	3 8 10	
6	-	5 10	
7	*	50	
8	2	2 50	
9	/	25	
10. Koniec znaków			**25**

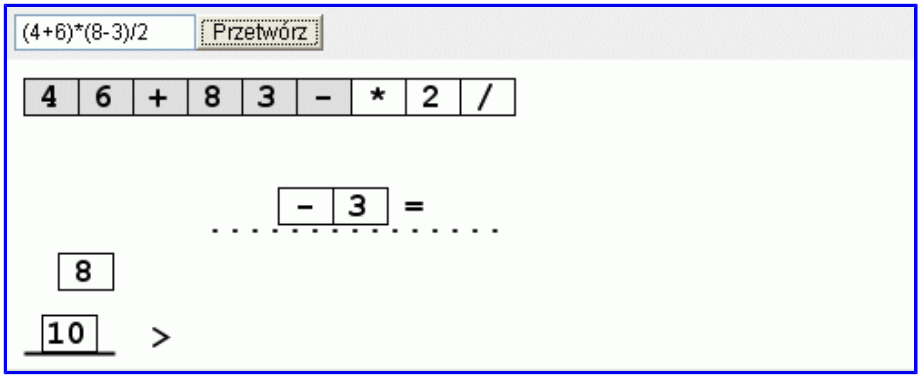
** krokiem określamy moment, gdy odczytaliśmy nowy znak (operator bądź liczbę jednocyfrową) lub ciąg cyfr, po której następuje znak biały(' ') – wtedy wiemy, że odczytaliśmy liczbę*

*** zakładamy, że stos musi zawierać elementy typu liczbowego*



Doszliśmy ostatecznie (po wywołaniu dwóch funkcji) do momentu, gdy wartość wyrażenia została obliczona. Możemy stwierdzić, że taki sposób tworzenia zapisu ONP jest przejrzysty, a użycie stosu jest jak najbardziej właściwe.

Po tych szczegółowych wyjaśnieniach proponujemy ćwiczenia z wykorzystaniem kolejnego apletu - po wpisaniu dowolnego wyrażenia w notacji zwykłej (infiksowej) zobaczycie od razu jego odpowiednik w ONP i będziecie mogli obserwować kolejne kroki, które prowadzą do obliczenia wartości tego wyrażenia.



Na zakończenie warto podkreślić, że pokazany wyżej sposób przetwarzania prostych wyrażeń arytmetycznych można znacznie uogólnić - wprowadzając wiecej operatorów (wraz z ich priorytetami) i umożliwiając przetwarzanie zmiennych i liczb rzeczywistych. Opisany proces można też uogólnić na translację wyrażeń warunkowych zawierających słowa kluczowe czy translację instrukcji przypisania. Używając tak prostej struktury danych jak stos możecie się pokusić o translację rozmaitych wyrażeń w wymyślonych przez siebie językach skryptowych.