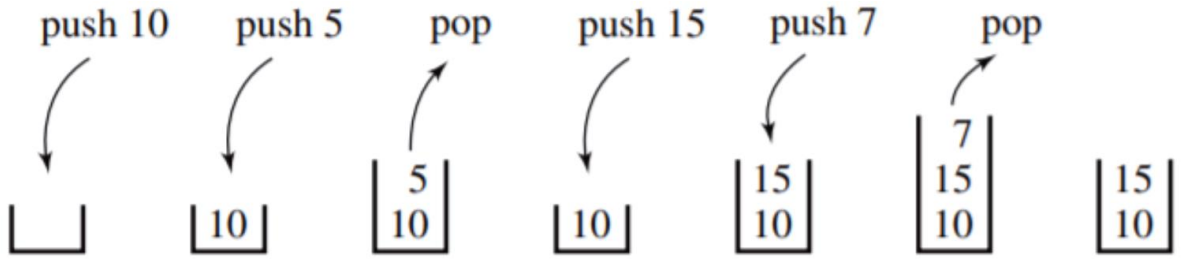
STRUKTURY DANYCH – STOS, KOLEJKA

Abstrakcyjnym typem danych nazywamy zbiór operacji określających, co jest robione, lecz nie jak.

STOS

Stos jest abstrakcyjnym typem danych, w którym przepływ elementów przebiega na zasadzie LIFO (ang. Last In – First Out). Łatwo zobrazować go na podstawie stosu książek – mamy dostęp jedynie do ostatniej książki, odłożonej na wierzch.



Stos (std::stack<T>) jest to adapter („wrapper") do kontenera typu deque (Double-Ended Queue), ale można wykorzystać również inny kontener (np. vector lub list).

- top() – zwraca wartość elementu na szczycie stosu;
- empty() – zwraca true, jeśli stos jest pusty; false – w przeciwnym przypadku, złożoność O(1).
- size() – zwraca liczbę elementów umieszczonych na stosie;
- push(e) – umieszcza element e na szczycie stosu, złożoność O(1)
- pop() – zdjęcie(usuniecie) istniejącego elementu ze szczytu stosu; funkcja nic nie zwraca, złożoność O(1).

Lista w Pythonie a stos

Lista to jeden z wbudowanych typów danych w Pythonie, używany do przechowywania uporządkowanych sekwencji elementów. Warto wspomnieć, że chociaż nazwy metod listy różnią się częściowo od tych zdefiniowanych powyżej, to listy oferują funkcjonalność stosu.

Zadanie 1. Zaimplementuj strukturę stosu w Python.

Wskazówka: Pythonie, ze względu na jego obiektowość, najwygodniejszym sposobem będzie zdefiniowanie klasy Stack z operacjami zaimplementowanymi w formie metod. Ze względu na podobieństwo między stosem a pythonową listą, można wykorzystać tą ostatnią jako punkt wyjścia do naszej implementacji.

Zadanie 2. Zapoznaj się z poniższym algorytmem sprawdzania poprawności nawiasów i zaproponuj jego rozwiązanie (implementacja w Python oraz schemat blokowy).

W większości zastosowań liczba nawiasów musi być zbilansowana, tzn. powinno być tyle samo prawych co lewych nawiasów. Ponadto, nawiasy muszą być poprawnie zagnieżdżone:

(()()()())

((((()))

(()((())()))

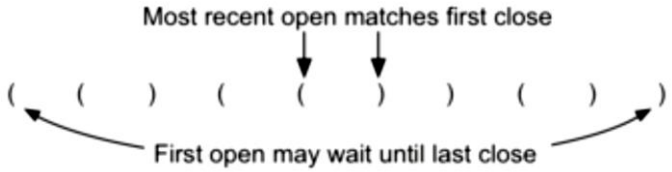
Dla porównania, poniżej znajduje się kilka przykładów niepoprawnych nawiasów:

(((((())

()))

(()()(()

Sprawdzanie poprawności nawiasów w różnych wyrażeniach to jedno z częstych i ważnych zadań, z którym muszą uporać się programiści. Naszym celem będzie teraz uporanie się z tym problemem przy użyciu stosu. Zauważmy przede wszystkim że w sekwencji nawiasów pierwszy z nawiasów zamykających musi pasować do ostatniego nawiasu otwierającego. Podobnie, pierwszy z nawiasów otwierających musi "czekać" do ostatniego nawiasu zamykającego. Innymi słowy, nawiasy zamykające powinny pasować do otwierających w odwrotnym porządku.



Z tego właśnie powodu stos wydaje się być strukturą odpowiednią do rozwiązania tego zagadnienia. Po wybraniu struktury reszta algorytmu jest prosta:

1. Rozpocznij z pustym stosem.
2. Wczytaj ciąg znaków z nawiasami.
3. Przetwarzaj ciąg idąc od lewej do prawej.
4. Jeśli aktualny znak jest nawiasem otwierającym, wstaw go na stos.
5. Jeśli aktualny znak jest nawiasem zamykającym, ściągnij nawias otwierający z wierzchołka stosu.
6. Jeśli po dojściu do końca ciągu stos jest pusty i nie wystąpiły żadne błędy, nawiasy są poprawne.

Zadanie 3. Zaproponuj algorytm sprawdzania poprawności różnych symboli.

Zadanie 4. Napisz program czytający wyrażenie arytmetyczne w notacji postfiksowej (Odwrotna Notacja Polska) i obliczający jego wartość z wykorzystaniem stosu.

ONP (ang. RPN - Reverse Polish Notation) jest sposobem zapisu wyrażeń arytmetycznych bez stosowania nawiasów, w którym symbol operacji występuje po argumentach. Poniżej podajemy przykłady wyrażeń w ONP:

Notacja normalna	**ONP**
2 + 2 =	2 2 + =
3 + 2 * 5 =	3 2 5 * + =
2 * (5 + 2) =	2 5 2 + * =
(7 + 3) * (5 - 2) ^ 2 =	7 3 + 5 2 - 2 ^ * =
4 / (3 - 1) ^ (2 * 3) =	4 3 1 - 2 3 * ^ / =

Algorytm obliczania wartości wyrażenia ONP wykorzystuje stos do składowania wyników pośrednich. Zasada pracy tego algorytmu jest bardzo prosta:

Z wejścia odczytujemy kolejne elementy wyrażenia. Jeśli element jest liczbą, zapisujemy ją na stosie. Jeśli element jest operatorem, ze stosu zdejmujemy dwie liczby, wykonujemy nad nimi operację określoną przez odczytany element, wynik operacji umieszczamy z powrotem na stosie. Jeśli element jest końcowym znakiem '=', to na wyjście przesyłamy liczbę ze szczytu stosu i kończymy. Inaczej kontynuujemy odczyt i przetwarzanie kolejnych elementów.

Przykładowo obliczmy tą metodą wartość wyrażenia **7 3 + 5 2 - 2 ^ * =**

we	stos	wy
7	7	
3	7 3	
+	10	
5	10 5	
2	10 5 2	
-	10 3	
2	10 3 2	
^	10 9	
*	90	
=		90

KOLEJKA

Kolejka, jest taką strukturą danych w której przepływ elementów przebiega na zasadzie FIFO (ang. First In – First Out). Łatwo zobrazować ją sobie na podstawie zwykłej kolejki w sklepie: kto pierwszy zajął miejsce do kasy, ten pierwszy zapłaci za zakupy. W kolejce mamy dostęp jedynie do pierwszego elementu.

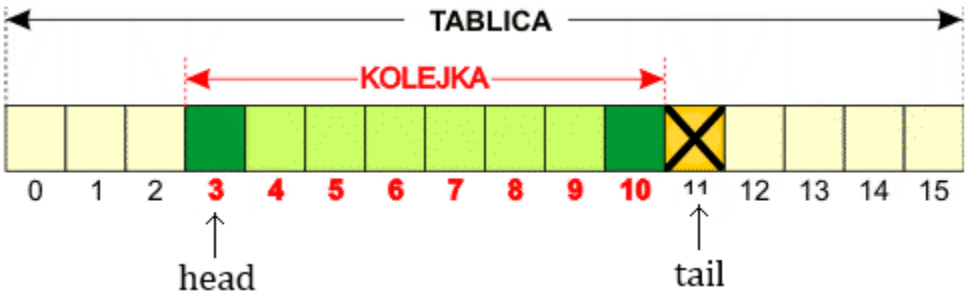
- Queue() tworzy nową kolejkę,
- enqueue(item) dodaje element na końcu kolejki (nic nie zwraca),
- dequeue() usuwa z kolejki element znajdujący się na jej początku i zwraca go,
- isEmpty() testuje, czy kolejka jest pusta,
- size() zwraca liczbę elementów w kolejce.

Operacja	Zawartość kolejki	Zwracana wartość
q.isEmpty()	[]	True
q.enqueue(4)	[4]	
q.enqueue('dog')	['dog',4]	
q.enqueue(True)	[True,'dog',4]	
q.size()	[True,'dog',4]	3
q.isEmpty()	[True,'dog',4]	False
q.enqueue(8.4)	[8.4,True,'dog',4]	
q.dequeue()	[8.4,True,'dog']	4
q.dequeue()	[8.4,True]	'dog'
q.size()	[8.4,True]	2

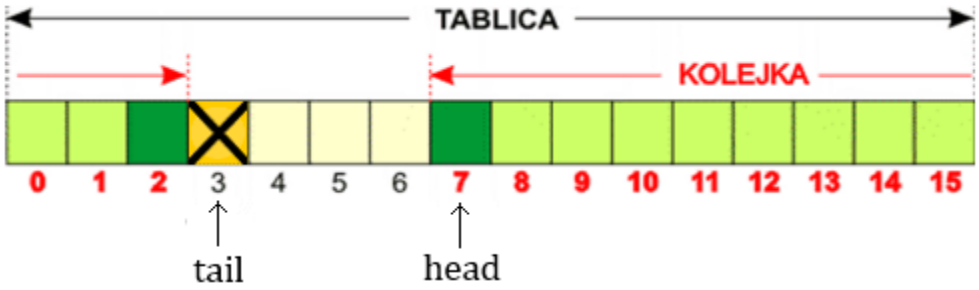
Przykład

Kolejkę możemy traktować jak bufor, w którym są przechowywane wprowadzane do niej dane. Naturalną strukturą danych do realizacji takiej kolejki jest lista, jednakże list jeszcze nie omawialiśmy, zatem zrealizujemy kolejkę w znanej nam strukturze danych – tablicy. Oprócz samej tablicy będziemy potrzebowali dodatkowo dwóch zmiennych:

head – wskaźnik początku kolejki zawiera indeks elementu, który jest początkiem kolejki; tail – wskaźnik końca kolejki zawiera indeks elementu, który leży tuż za ostatnim elementem kolejki.



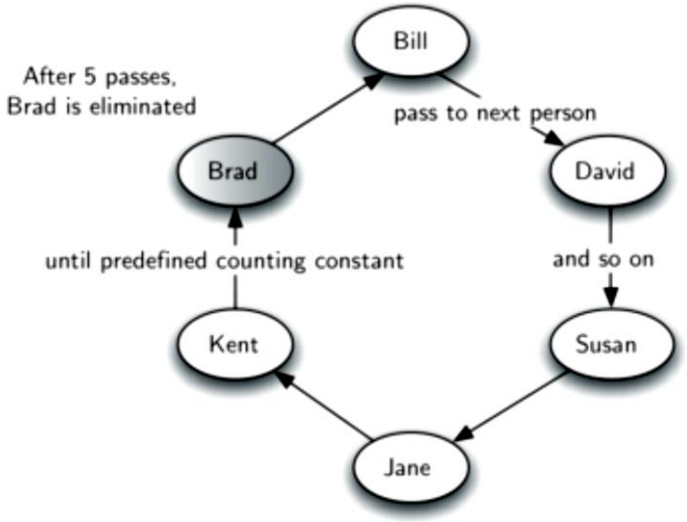
Dane zawsze wpisujemy do elementu tablicy, którego indeks zawiera wskaźnik końca kolejki tail. Po tej operacji tail zwiększamy o 1. Jeśli tail wyjdzie poza ostatni element tablicy, to jest on zerowany. Dane pobieramy (obsługujemy) z komórki tablicy o indeksie head. Po odczycie, head zwiększamy o 1. Jeśli wskaźnik head wyjdzie poza ostatni element tablicy, to podobnie jak tail, jest zerowany.



Zadanie 1. Zaimplementuj strukturę kolejki w Python.

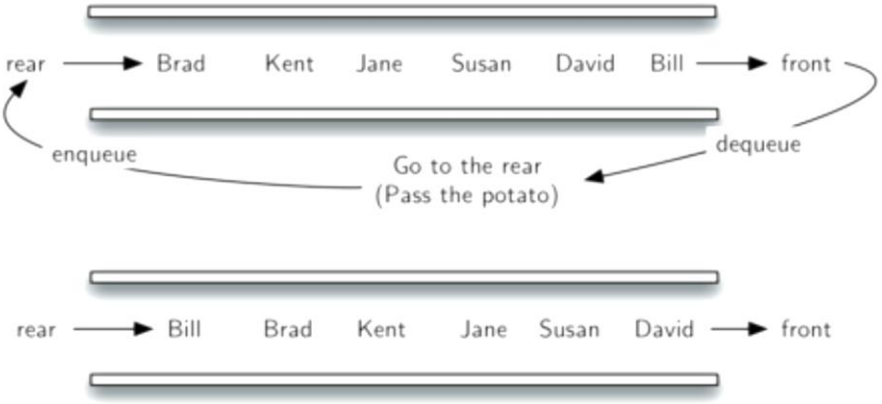
Zadanie 2. Gra w gorącego ziemniaka.

Typowe zastosowania kolejek to symulacje rzeczywistych sytuacji, w których dane muszą być przetwarzane na sposób FIFO. Przykładem może być towarzyska gra w gorącego ziemniaka. Ustawieni w kręgu uczestnicy przekazują sobie nawzajem (do najbliższego sąsiada) pewien element (ziemniak). W pewnym momencie akcja zostaje przerwana. Uczestnik, który jest akurat w posiadaniu elementu, odpada z gry. Akcja jest kontynuowana aż do pozostania jednego uczestnika.



Zabawa ta to w zasadzie nowoczesna wersja problemu Józefa Flawiusza: https://pl.wikipedia.org/wiki/Problem_J%C3%B3zefa_Flawiusza

Celem jest stworzenie symulacji tej gry. Wykorzystamy przy tym kolejkę, aby zaimplementować ustawienie uczestników w kręgu. Załóżymy mianowicie, że ziemniaka ma zawsze uczestnik znajdujący się na początku kolejki. Przekazanie ziemniaka sąsiadowi będzie odpowiadało usunięciu tego uczestnika z początku kolejki i wstawienie go natychmiast na koniec. Co pewien czas (po określonej liczbie operacji usuwania/wstawiania) będziemy usuwać uczestnika znajdującego się na początku kolejki na stałe. Symulacja toczy się do momentu, aż w kolejce zostanie tylko jeden uczestnik.


rear
Brad
Kent
Jane
Susan
David
Bill
front
enqueue
dequeue
Go to the rear
(Pass the potato)

rear
Bill
Brad
Kent
Jane
Susan
David
front