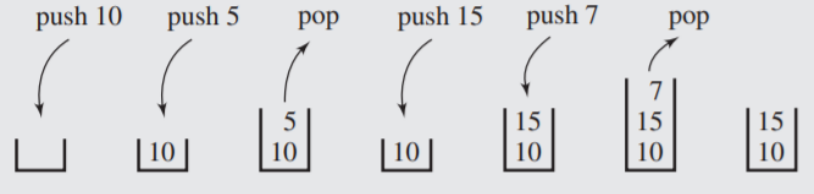
Abstrakcyjnym typem danych nazywamy zbiór operacji określających, *co* jest robione, lecz nie *jak*.

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;
size() – zwraca liczbę elementów umieszczonych na stosie;
push(e) – umieszcza element *e* na szczycie stosu;
pop() – zdjęcie(usuniecie) *istniejącego* elementu ze szczytu stosu; funkcja nic nie zwraca.

Struktura danych	Operacja	Złożoność
Stos	`push()`	O(1)
	`pop()`	O(1)
	`empty()`	O(1)

Przykład. Implementacja stosu w STL, zobacz projekt *przyklad_stos_kolejka*

Zadanie 1. Napisz program wykorzystujący stos, który sprawdza czy w danym wyrażeniu nawiasy są prawidłowo zagnieżdżone. Np. wyrażenie "(()(()))" jest prawidłowe, a ")(" oraz "(()" są nieprawidłowe.

Zadanie 2. *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.

Kolejka (std::queue<T>) jest to adapter do kontenera typu *deque* (domyslnie), ale można wykorzystać również inny kontener (np. *list*).

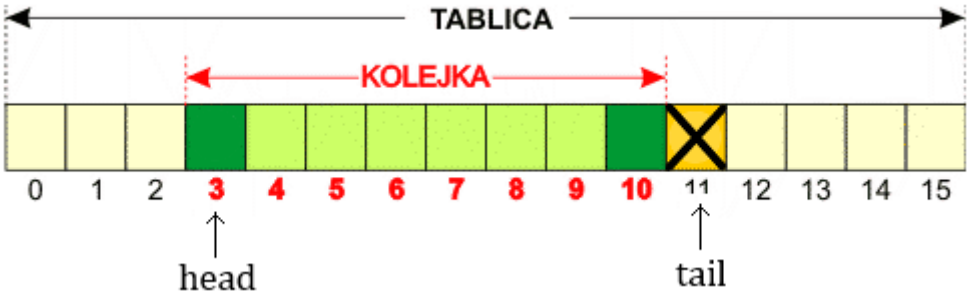
front() – zwraca wartość pierwszego elementu w kolejce;
back() – zwraca wartość ostatniego elementu w kolejce;
empty() – zwraca *true*, jeśli kolejka jest pusta; *false* – w przeciwnym przypadku;
size() – zwraca ilość elementów umieszczonych w kolejce;
push(e) – umieszczenie nowego elementu *e* na końcu kolejki;
pop() – usunięcie istniejącego elementu z początku kolejki.

Struktura danych	Operacja	Złożoność
Kolejka	`dequeue()`	O(1)
	`enqueue()`	O(1)
	`empty()`	O(1)

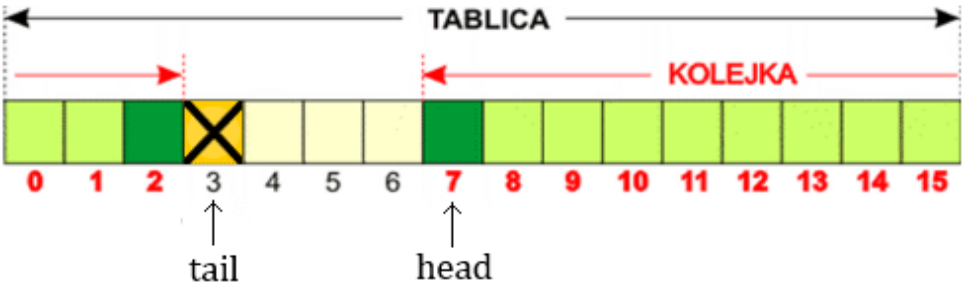
Przykład. Implementacja kolejki w STL, zobacz projekt *przyklad_stos_kolejka*

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

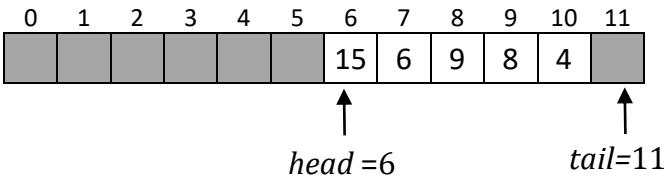


Dane zawsze wpisujemy do elementu tablicy, którego indeks zawiera wskaźnik końca kolejki *tail*. Po tej operacji *tail* zwiększamy o 1. Jeśli *tail* wyjdzie poza ostatni element tablicy, to jest on zerowany. Dane pobieramy (obsługujemy) z komórki tablicy o indeksie *head*. Po odczycie, *head* zwiększamy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

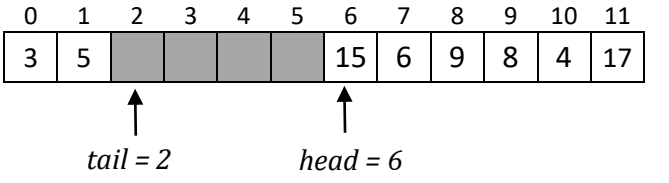


Przykład

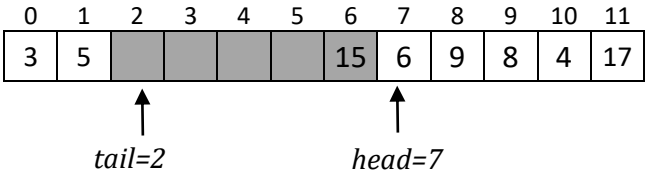
a) Kolejka zawiera 5 elementów, które znajdują się w tablicy od indeksu 6 do 10:



b) Stan kolejki po dodaniu do niej trzech wartości – *enqueue*(17), *enqueue*(3), *enqueue*(5):

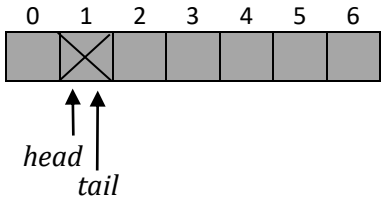


c) Stan kolejki po wywołaniu funkcji *dequeue*() – usunięciu/obsłudze elementu:



Zadanie 3. Zaimplementuj kolejkę dwustronną (o dwóch końcach), która pozwala wstawiać i usuwać elementy na obu jej końcach.
(*Pliki do wykorzystania*: projekt VS *zadanie3_deque*)

empty()



filled()

enqueue_head(char element)

- jeśli kolejka jest pełna, to wypisujemy „*Błąd przepełnienia*”;
- w pozostałych przypadkach, przesuń *head* i wstaw element;

enqueue_tail(char element)

- jeśli kolejka jest pełna, to wypisujemy „*Błąd przepełnienia*”;
- w pozostałych przypadkach: dodaj element w miejscu *tail* i przesuń *tail*;

char **dequeue_head**()

- jeśli kolejka jest pusta, to wypisujemy „*Błąd niedomiaru*”, zwracamy ‘\0’;
- w pozostałych przypadkach: zapamiętaj znak, na który wskazuje *head*, następnie przesuń *head* i zwróć zapamiętany znak;

`char` **`dequeue_tail`**`()`

- jeśli kolejka jest pusta, to wypisujemy „*Błąd niedomiaru*”, zwracamy ‘\0’;
- w pozostałych przypadkach: przesuń *tail* i zwróć znak, na który teraz wskazuje *tail*;