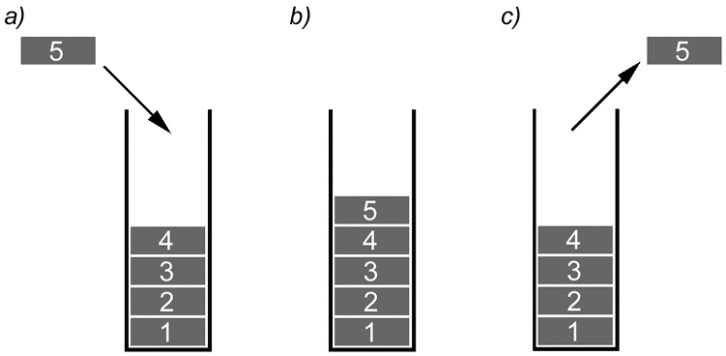
Podstawowe struktury danych: stos, kolejka, listy, drzewo binarne

Stos (ang. *stack*) to obszar wewnętrznej pamięci komputerowej przeznaczonej do czasowego przechowywania informacji związanych z wykonywanym programem. Dla rekurencji istotne jest, by stos posiadał strukturę **LIFO** (ang. ***L**ast **I**n **F**irst **O**ut* – ostatni na wejściu, pierwszy na wyjściu). Działanie stosu zobrazowano na rysunku jako swoistego rodzaju studnię informacyjną albo stos książek (***a*** – dodanie informacji, ***b*** – przechowanie informacji, ***c*** – pobranie informacji ze stosu).



Komputer odzyskuje potrzebne do wykonania programu informacje, pobierając je z wierzchołka stosu. Żądany element lokalizowany jest dzięki rejestrowi zwanemu **wskaźnikiem stosu** (ang. *stack pointer*), który jest inkrementowany o jeden każdorazowo przed umieszczeniem kolejnego elementu na stosie i dekrementowany o jeden po zdjęciu elementu ze stosu. Łatwo zauważyć, że gdy wskaźnik ma wartość zero, to stos jest pusty. Stos jest obszarem pamięci o ograniczonej pojemności, dlatego łatwo może dojść do jego przepełnienia. Podczas rekurencji zdarza się to nader często i wywołuje błąd, który sygnalizowany jest komunikatem o nazwie **„przepełnienie stosu”** (ang. *stack overflow*).

Mamy następujące funkcje (metody) obsługujące stos:

a) **`empty()`** – sprawdza, czy stos jest pusty,

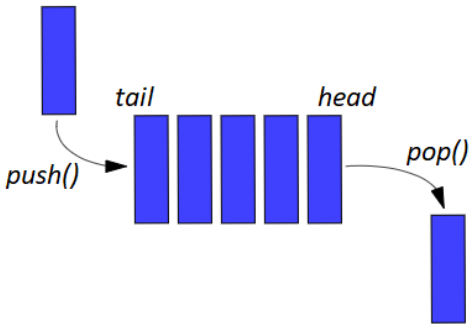
b) **`size()`** – zwraca rozmiar stosu,

c) **`top()`** – pozwala wykonać operacje na dostępnym (ostatnim) elemencie stosu,

d) **`push()`** – dodaje element na szczyt stosu (o ile to możliwe),

e) **`pop()`** – usuwa element ze szczytu stosu (o ile to możliwe).

Kolejka (ang. *queue*) jest abstrakcyjną strukturą danych typu **FIFO** (ang. *First In First Out* – pierwszy na wejściu, pierwszy na wyjściu) i składa się z **czoła** (ang. *head*) oraz **ogona** (ang. *tail*). Struktura ta przypomina obsługę kolejki w sklepie, oczywiście przy założeniu, że nie ma w niej klientów uprzywilejowanych (priorytetowych). Stos jest więc niejako jej przeciwieństwem.



Podobnie jak stos, kolejka FIFO jest strukturą o ograniczonym dostępie do danych i zakłada dwie podstawowe operacje:

1. **Push** – wprowadzenie danych do ogona kolejki,
2. **Pop** – usunięcie (poprzez obsłużenie) danych z czoła kolejki.

W związku z tym mamy cztery podstawowe funkcje (metody) obsługujące kolejkę:

a) **`empty()`** – sprawdza, czy kolejka jest pusta,

b) **`pop()`** – usuwa element na początku kolejki,

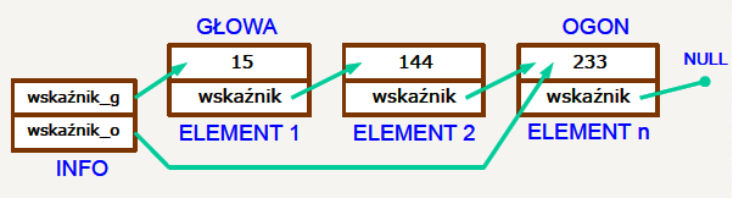
c) **`push()`** – dodaje element na końcu kolejki,

d) **`size()`** – zwraca rozmiar kolejki.

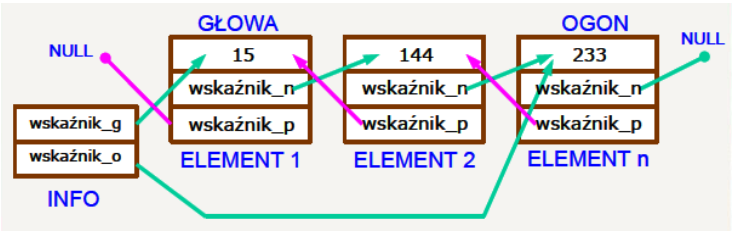
Istnieje również abstrakcyjny typ danych zwany **kolejką priorytetową** (ang. *priority queue*), w której każda ze znajdujących się w niej danych ma przypisany priorytet (inaczej wartość, klucz), decydujący o kolejności wykonania danych w kolejce. Ten typ kolejki wykorzystywany jest np. w systemach operacyjnych, w których zasoby sprzętowe przydzielane są uruchomionym procesom z wykorzystaniem priorytetów.

Lista jednokierunkowa jest oszczędną pamięciowo liniową strukturą danych, pozwalająca grupować dowolną (ograniczoną tylko przez dostępną pamięć) liczbę elementów: liczb, znaków, rekordów itd. Jest to duża zaleta w porównaniu ze zwykłymi tablicami, a nawet z tablicami dynamicznymi.

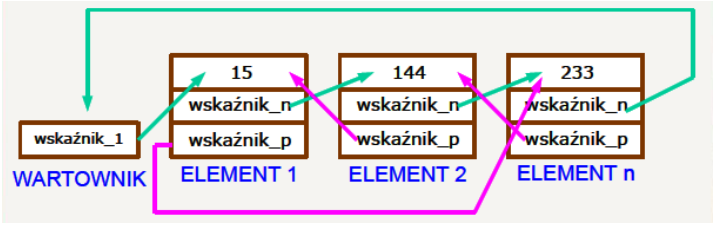
W liście jednokierunkowej występuje blok INFO (informacyjny) zawierający głowę (wskaźnik do pierwszego elementu) oraz ogon (wskaźnik do ostatniego elementu), przy czym wskaźnik oznacza adres komórki pamięci komputera przechowywany w zmiennej typu wskaźnikowego.



Lista dwukierunkowa różni się tym od jednokierunkowej, że każdy element zawiera wskaźnik zarówno do następnego, jak i do poprzedniego elementu.



Oprócz dwóch omówionych rodzajów list istnieje jeszcze **lista cykliczna**, zwana również **listą z wartownikiem**. Pierwszy element to następnik wartownika, ostatni element to poprzednik wartownika, a lista pusta składa się wyłącznie wartownika. Zatem po liście można poruszać w sposób cykliczny. Oba typy list mogą być cykliczne – na rysunku przedstawiono dwukierunkową listę cykliczną z wartownikiem.



Funkcje (metody) obsługujące listy:

a) **`push_front()`** – dodaje element na początku listy,

b) **`push_back()`** – dodaje element na końcu listy,

c) **`insert()`** – wstawia element we wskazanym miejscu listy,

d) **`pop_front()`** – usuwa element z początku listy,

e) **`pop_back()`** – usuwa element z końca listy,

f) **`size()`** – zwraca liczbę elementów listy,

g) **`max_size()`** – zwraca maksymalną liczbę elementów, jakie może zmieścić lista,

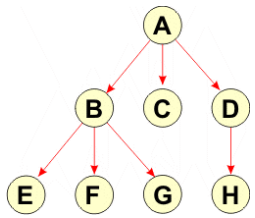
h) **`empty()`** – sprawdza, czy lista jest pusta,

i) **`remove()`** – usuwa z listy wszystkie elementy o danej wartości,

j) **`sort()`** – układa rosnąco elementy listy,

k) **`reverse()`** – odwraca kolejność elementów na liście.

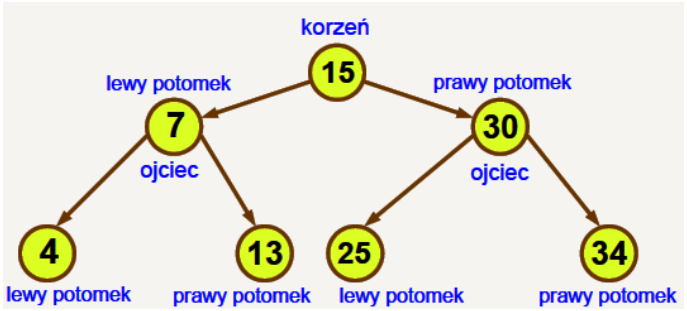
Ogólnie **drzewo** w informatyce oznacza strukturę danych, która implementuje drzewo matematyczne. Drzewa ułatwiają i przyspieszają wyszukiwanie, lecz także pozwalają na łatwe operowanie na danych posortowanych. Zastosowanie drzew jest olbrzymie (bazy danych, grafika komputerowa, teleinformatyka, łączenie serwerów itd.). Ogólną strukturę drzewa przedstawiono na rysunku.



A...H – węzły, przy czym **A** reprezentuje także korzeń drzewa (ang. *root*)
Strzałki – krawędzie. Grot strzałki oznacza kierunek rodzic → dziecko (ang. *parent* → *child*)
B, **C**, **D** – bracia i jednocześnie dzieci węzła **A** (rodzica)
E, **F**, **G** – bracia i jednocześnie dzieci węzła **B** (rodzica)
H – dziecko węzła **D** (rodzica)
C, **E**, **F**, **G**, **H** – liście

Długość ścieżki prostej od korzenia do danego węzła nazywa się **poziomem węzła** (ang. *node level*). **Wysokością drzewa** (ang. *tree height*) nazywa się najdłuższą ścieżkę rozpoczynającą się w korzeniu, czyli na przedstawionym rysunku wynosi 2, przy czym korzeń drzewa (zawsze) ma poziom 0, a węzły najniższe (tutaj) poziom 2.

Natomiast **drzewem binarnym** (ang. *binary tree, B-tree*) nazywamy drzewo, w którym węzły rodzice (ojcowie) mogą posiadać co najwyżej dwoje dzieci (synów) – ze swojej prawej i lewej strony (ang. *right child node, left child node*). Wszystkie elementy znajdujące się w lewym poddrzewie muszą być mniejsze od swojego ojca, a w prawym większe. Przypuśćmy, że mamy drzewo o wysokości 2, które należy wypełnić liczbami: 15 (korzeń) oraz 7, 13, 30, 25, 4, 34. Wypełniając po kolei, uzyskamy następujący efekt:



Mamy do dyspozycji następujące funkcje (metody):

a) **`add()`** – dodanie elementu do drzewa,

b) **`find()`** – znajdowanie i zwracanie elementu w drzewie,

c) **`remove()`** – usuwanie elementu z drzewa,

d) **`print()`** – drukowanie zawartości drzewa.