Podstawy Programowania 2

Stos i kolejka

Arkadiusz Chrobot

Katedra Systemów Informatycznych

10 kwietnia 2024

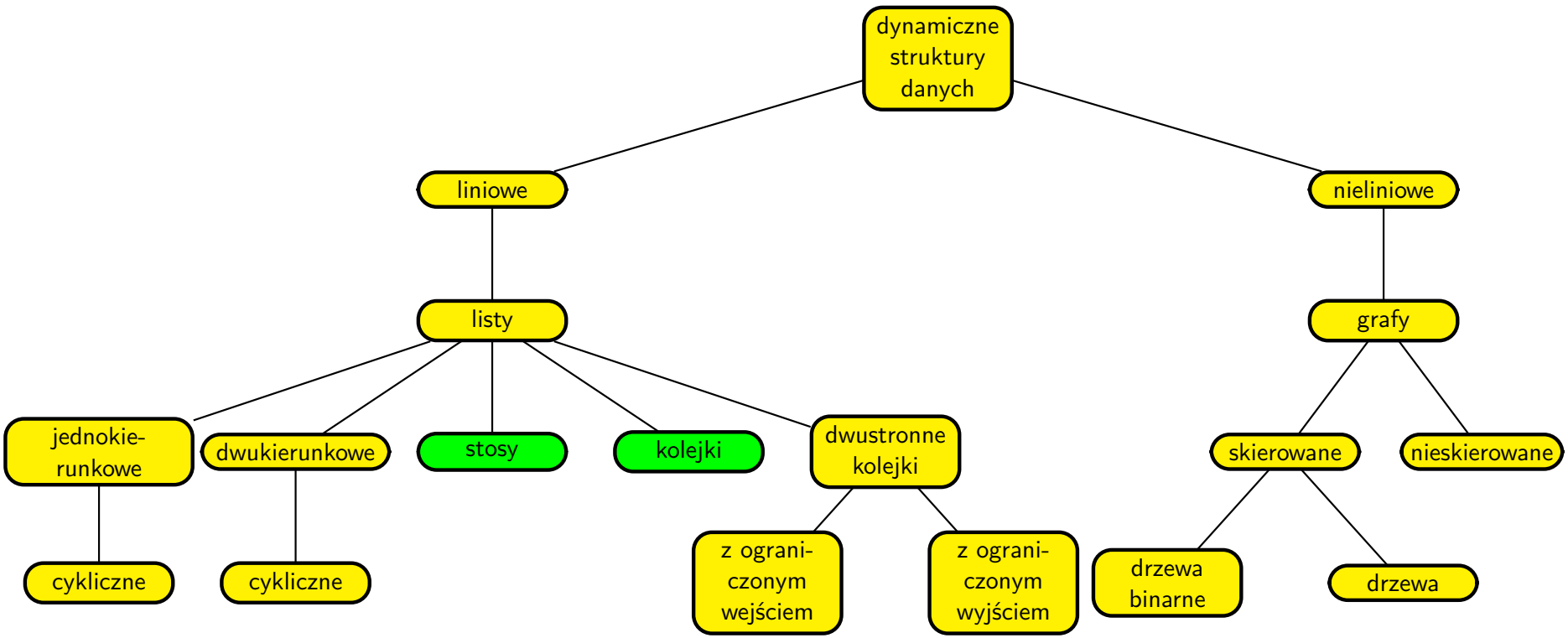
Plan

Struktury danych

Na pierwszym wykładzie omówiliśmy funkcje zarządzające obszarem pamięci programu, który nazywamy stertą. Dzięki nim programiści mogą tworzyć struktury danych, które nie są częścią standardu języka C. Większość z nich bazuje na strukturach i wskaźnikach oraz wymaga zdefiniowania specjalnych operacji, które implementowane są w postaci funkcji. Na następnym slajdzie znajduje się ilustracja przedstawiająca klasyfikację niektórych ze struktur danych, które mogą zostać utworzone w ten sposób. Ten wykład będzie dotyczył stosu i kolejki.

Stos był już opisywany, choćby na wykładzie dotyczącym rekurencji. To struktura danych, w której elementy z danymi (nazywane *węzłami*) są dodawane lub usuwane zgodnie z porządkiem *pierwszy nadszedł, ostatni wychodzi* (ang. *Last In First Out* — LIFO). Oznacza to, że dane ze stosu są odczytywane w odwrotnej kolejności, w stosunku do tej, w której zostały dodane. Kolejka jest podobną strukturą danych, ale węzły w niej są dodawane i usuwane zgodnie z porządkiem „pierwszy nadszedł, pierwszy wychodzi” (ang. *First In First Out* — FIFO).

Klasyfikacja struktur danych



Klasyfikacja struktur danych

Stos — wprowadzenie

Stos może być zrealizowany na kilka różnych sposobów. Jednym z nich jest dynamiczna struktura danych, której elementy są ze sobą powiązane za pomocą wskaźników. Stos jest także szczególnym przypadkiem innych struktur danych, które nazywamy *listami*. Aby lepiej jednak zrozumieć czym jest stos przyjmiemy teraz, że lista to ciąg połączonych ze sobą węzłów z dwoma końcami. Stos jest listą, w której operacje dodawania i usuwania elementów wykonywane są wyłącznie na jednym z tych końców. Zazwyczaj ilustruje się go jako pionową listę, której wyróżniony koniec nazywa się *szczytem* lub *wierzchołkiem* (ang. *top*).

Stos — implementacja

Implementacja stosu wymaga nie tylko zdefiniowania typu danych dla jego węzłów, ale również operacji, które będą na tej strukturze danych wykonywane. To zadanie musimy wykonać korzystając z elementów, które są dostarczane przez język C. Jego pierwszą część zrealizujemy za pomocą struktury, a drugą przy pomocy funkcji. Przyjmiemy w tym wykładzie, że stos będzie przechowywał w swoich węzłach liczby typu `double`. W ogólnym przypadku typ danych przechowywanych w węźle stosu programista musi określić zgodnie ze swoimi potrzebami.

Stos — implementacja

Typ danych dla węzła stosu

```
struct stack_node {
    double number;
    struct stack_node *next;
};
```

Stos — implementacja

Poprzedni slajd prezentuje definicję przykładowej struktury określającej typ danych pojedynczego elementy stosu. Ta struktura zawiera dwa pola. Pierwsze pole służy do przechowywania danych. Ogólnie, takich pól może być więcej i mogą one mieć bardziej złożone typy danych. W przykładzie jest tylko jedna taka składowa o nazwie `number` i typie `double`. Drugie pole jest polem wskaźnikowym. Spotykane są stosy, w których takich pól może być więcej, ale zawsze musi występować przynajmniej jedno. Proszę zwrócić uwagę na typ tego wskaźnika. Jest to wskaźnik na strukturę, w której jest on zawarty. Zatem struktura z poprzedniego slajdu jest strukturą o budowie *rekurencyjnej*. Oznacza, to że ten wskaźnik może wskazywać na inną strukturę, której typ jest taki sam, jak tej, w której jest on zwarty. Innymi słowy, dzięki temu wskaźnikowi węzły stosu mogą być ze sobą połączone.

Stos — implementacja

Kolejne elementy stosu będą powiązane ze sobą za pomocą umieszczonych w nich pól wskaźnikowych. Aby jednak móc wykonywać operacje na stosie, program zawsze musi wiedzieć, gdzie jest jego wierzchołek. Do zapamiętania adresu tego wierzchołka będzie potrzebna osobna zmienna wskaźnikowa, która może być lokalna lub globalna. W programach może mieć ona różną nazwę, ale ogólnie określa się ją mianem *wskaźnika stosu* lub, bardziej dokładnie, *wskaźnika wierzchołka stosu*. Mając typ danych węzła stosu i wskaźnik na wierzchołek stosu, musimy zdefiniować jeszcze operacje, które będą na tym stosie wykonywane. Najbardziej podstawowymi z nich są dwie: dodanie nowego węzła do stosu, określane angielską nazwą *push* i zdjęcie (usunięcie) węzła ze stosu — *pop*. Obie te operacje wykonywane są na wierzchołku stosu. Na wykładzie zdefiniujemy jeszcze trzecią operację, która jest także często spotykana, ale nieobowiązkowa — odczytanie wartości węzła na szczycie stosu, czyli po angielsku *peek*.

Stos — implementacja

Funkcja `push()`

Operacja *push* została zaimplementowana w postaci funkcji o tej samej nazwie (`push()`). Określimy dwa warunki (niezmienniki), które musi spełniać ta funkcja, abyśmy mogli uznać jej działanie za poprawne:

1. Przed wykonaniem funkcji wskaźnik stosu musi wskazywać węzeł będący na szczycie stosu, albo być wskaźnikiem pustym — w tym ostatnim przypadku będziemy mieli do czynienia z *pustym stosem*.
2. Funkcja w wyniku swojego działania zwróci wskaźnik stosu, który będzie albo taki sam, jak jej został przekazany — będzie to oznaczało, że nie udało się dodać nowego węzła do stosu, albo będzie wskazywał na nowy węzeł, który znajdzie się na szczycie stosu — to będzie oznaczało poprawne zakończenie operacji.

Stos — implementacja

Funkcja push()

```
1  struct stack_node *push(struct stack_node *top, double
   ↪  number)
2  {
3      struct stack_node *new_node = (struct stack_node
   ↪  *)malloc(sizeof(struct stack_node));
4      if(new_node!=NULL) {
5          new_node->number = number;
6          new_node->next = top;
7          top = new_node;
8      }
9      return top;
10 }
```

Uwaga! Numery wierszy nie są częścią kodu źródłowego. Są one wprowadzone aby ułatwić jego opis.

Stos — implementacja

Funkcja `push()`

Funkcja `push()` przyjmuje dwa argumenty, które są podstawiane pod widoczne w definicji funkcji parametry. Pierwszym z nich jest wskaźnik stosu, a drugim liczba, która ma być zapisana w nowym węźle. Pierwszą czynnością wykonywaną wewnątrz funkcji jest przydział pamięci na nowy węzeł (wiersz nr 3). Przebieg dalszych działań zależy od jej powodzenia. Jeśli się ona nie powiedzie, to wskaźnik `node_new` będzie miał wartość `NULL`, a funkcja zakończy działanie zwracając poprzednią wartość wskaźnika stosu (parametr `top`). W przeciwnym przypadku wskaźnik `new_node` będzie zawierał adres nowego węzła. W polu `number` tego elementu zostanie zapisana wartość liczby, którą będzie on przechowywał (wiersz nr 5). Następnie, w wierszu nr 6, w polu `next` nowego węzła zostanie zapisany adres bieżącego wierzchołka stosu. W ten sposób nowy element zostanie połączony ze stosem i stanie się jego nowym wierzchołkiem. To na niego teraz powinien wskazywać wskaźnik `top`, dlatego w wierszu nr 7 jest do tego wskaźnika zapisywany adres nowego węzła.

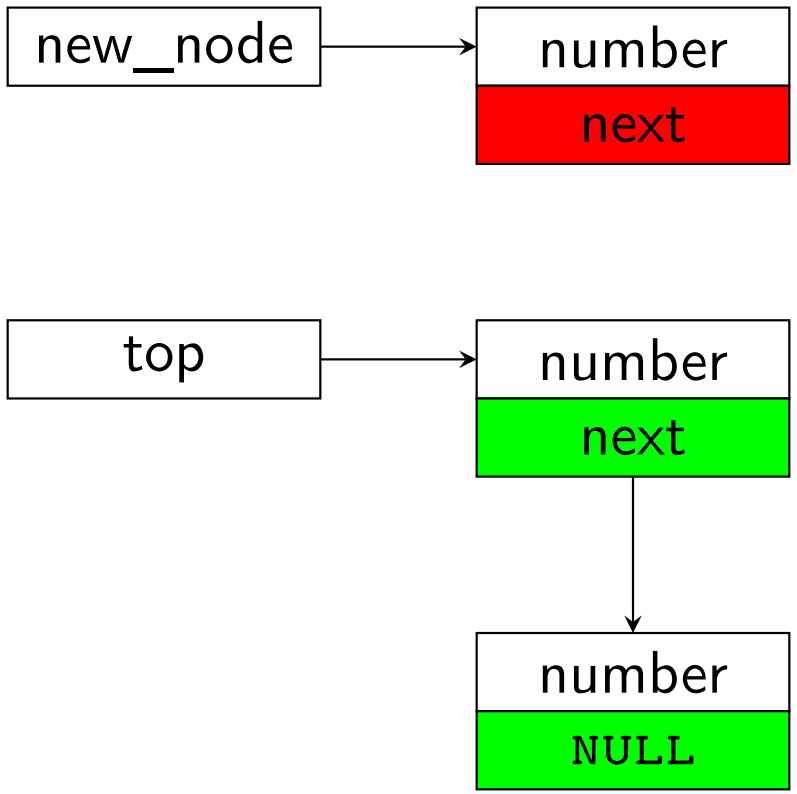
Stos — implementacja

Funkcja `push()`

Po wykonaniu tych czynności funkcja kończy działanie zwracając adres nowego wierzchołka stosu.
Funkcję `push()` można zrealizować na kilka sposobów. Na przykład wskaźnik stosu może być przekazany przez parametr będący podwójnym wskaźnikiem modyfikowanym wewnątrz funkcji. To rozwiązanie zostanie dokładniej objaśnione w opisie działania funkcji `pop()`.
Kolejne slajdy zawierają ilustrację udanego dodania nowego węzła do niepustego stosu, składającego się z dwóch elementów. Proszę zwrócić uwagę, że pole `next` nowego elementu jest oznaczone początkowo na czerwono. Oznacza to, że jest nieprawidłowym wskaźnikiem.

Stos — implementacja

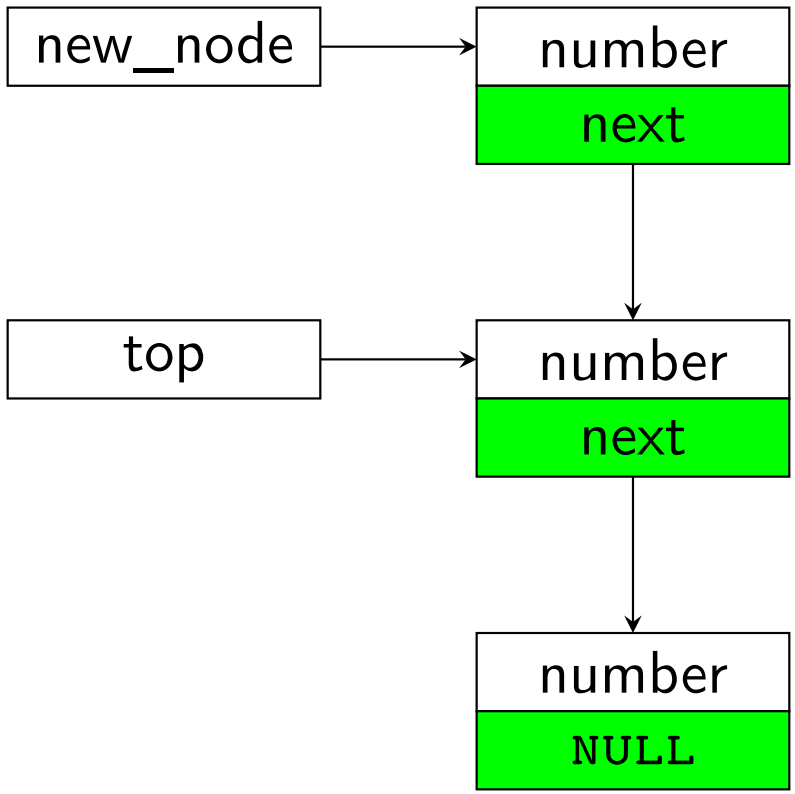
Funkcja `push()`



Przed wykonaniem wiersza nr 6 funkcji `push()`

Stos — implementacja

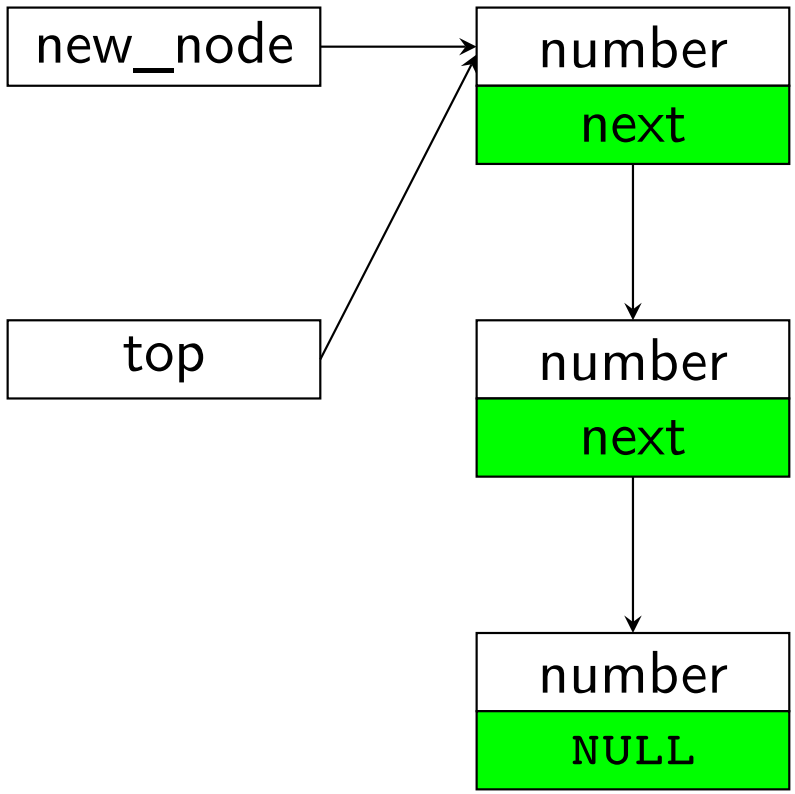
Funkcja `push()`



Przed wykonaniem wiersza nr 7 funkcji `push()`

Stos — implementacja

Funkcja `push()`



Przed wykonaniem wiersza nr 8 funkcji `push()`

Stos — implementacja

Funkcja `push()`

Proszę zwrócić uwagę, na charakterystyczną cechę pola `next` węzła znajdującego się na *dnie* stosu. Ma ono wartość `NULL`. Taka jego wartość oznacza, że jest to ostatni węzeł stosu i za nim nie ma już innych elementów tej struktury. Zastanówmy się, czy funkcja `push()` gwarantuje, że pole `next` ostatniego węzła stosu będzie zawsze miało tę wartość. Okazuje się, że dzieje się tak tylko wtedy, gdy przy pierwszym jej wywołaniu, związanym z tworzeniem pierwszego elementu danego stosu, zostanie jej przekazany wskaźnik wierzchołka o wartości `NULL`. Wtedy jego wartość zostanie zapisana do pola `next` pierwszego i bieżąco jedynego, a wiec także ostatniego węzła stosu. Jeśli jednak funkcji zostanie przekazany nieprawidłowy wskaźnik, to jego wartość także zostanie umieszczona w tym polu i jest to bardzo niebezpieczna sytuacja, bo program nie będzie miał możliwości rozpoznania końca stosu. Musimy zatem zadbać, aby przy *dodawaniu pierwszego węzła do stosu* funkcji `push()` przekazać pusty wskaźnik, szczególnie wtedy, gdy ten wskaźnik jest zmienną lokalną.

Stos — implementacja

Funkcja `pop()`

Operacja *pop* zostanie zaimplementowana w postaci funkcji `pop()`. Podobnie jak dla funkcji `push()` wyznaczymy dla niej dwa niezmienniki, które muszą być spełnione, abyśmy mieli pewność, że została ona zrealizowana prawidłowo:

1. Przed wykonaniem funkcji wskaźnik stosu powinien wskazywać na węzeł znajdujący się na szczycie stosu lub być pusty.
2. Po wykonaniu funkcji wskaźnik stosu powinien wskazywać na węzeł znajdujący się na szczycie stosu mniejszego o jeden element lub powinien być pusty.

Stos — implementacja

Funkcja pop()

```
1  double pop(struct stack_node **top)
2  {
3      double result = 0.0;
4      if(*top) {
5          result = (*top)->number;
6          struct stack_node *temporary = (*top)->next;
7          free(*top);
8          *top = temporary;
9      }
10     return result;
11 }
```

Stos — implementacja

Funkcja `pop()`

Funkcja `pop()` posiada tylko jeden parametr, który jest *podwójnym wskaźnikiem*, albo inaczej *wskaźnikiem na wskaźnik*. Przez ten parametr przekazywany jest do funkcji adres wskaźnika stosu. Został on zastosowany ponieważ wewnątrz funkcji modyfikowany jest stos poprzez usunięcie z jego szczytu węzła. Oznacza to, że trzeba będzie także zmienić wartość wskaźnika stosu. Nie można jednak zastosować tego samego rozwiązania, co w przypadku `push()`, jeżeli funkcja `pop()` ma zwracać wartość usuwanego elementu, czyli liczbę zapisaną w jego polu `number`. Wyjściem jest właśnie parametr będący wskaźnikiem na wskaźnik, za który podstawiany jest adres wskaźnika stosu.

Stos — implementacja

Funkcja `pop()`

Funkcja `pop()` ma zmienną lokalną (`result`) typu `double`, która ma wartość początkową równą `0.0`. Jeśli stos będzie pusty, to funkcja zwróci właśnie taką liczbę i zakończy działanie. To, czy stos nie jest pusty, sprawdzane jest w czwartym wierszu funkcji. Wyrażenie `*top` jest skróconym zapisem warunku `*top!=NULL`. Jeśli jest on prawdziwy, to funkcja zapisuje do zmiennej `result` wartość bieżącego elementu na szczycie stosu (wiersz 5), a następnie zapamiętuje w zmiennej wskaźnikowej `temporary` adres następnego węzła stosu, który jest przechowywany w polu `next` elementu z wierzchołka (wiersz 6). Potem element na szczycie jest usuwany (wiersz 7). Tym samym wskaźnik stosu przestaje mieć prawidłową wartość — wskazuje na nieistniejący element zamiast na węzeł z wierzchołka stosu. Aby naprawić tę sytuację, w wierszu 8 zapisywany jest do niego adres przechowywany w zmiennej `temporary`. Po tej operacji funkcja zwraca liczbę zapamiętaną w zmiennej `result` i kończy działanie.

Stos — implementacja

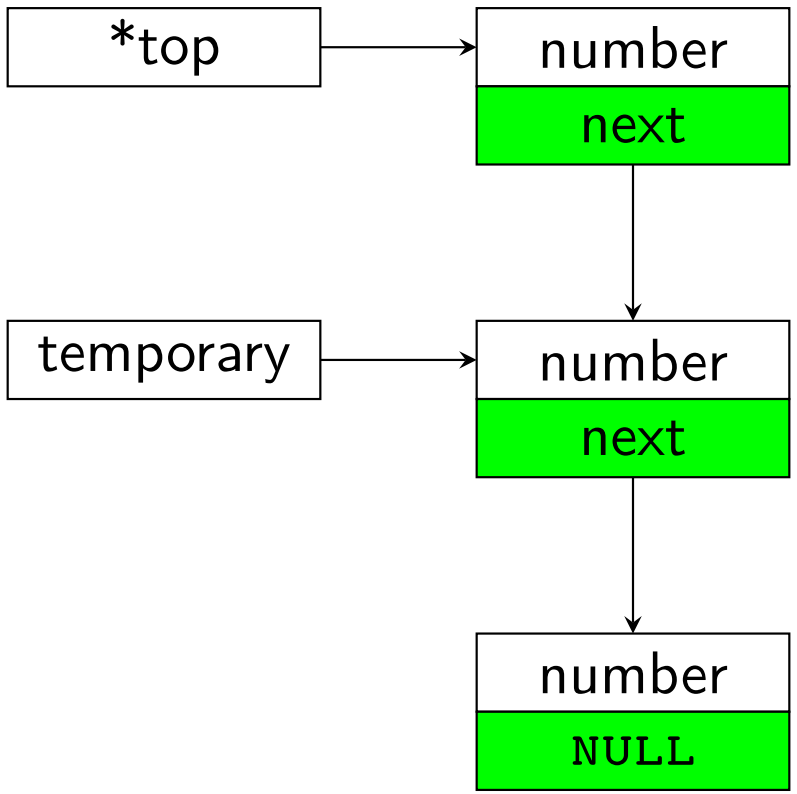
Funkcja `pop()`

Proszę zwrócić uwagę, że funkcja `pop()` usuwa poprawnie również ostatni węzeł stosu (ten z jego *dna*). Jego wskaźnik `next` ma wartość `NULL` i taką też wartość uzyska wskaźnik `temporary`, po wykonaniu dla jednoelementowego stosu wiersza nr 6 funkcji. Po wykonaniu wiersza nr 8 ta wartość będzie nadana również wskaźnikowi stosu. Jest to oczekiwany wynik, gdyż w rezultacie usunięcia ostatniego węzła ze stosu powinniśmy uzyskać pusty stos.

Kolejne slajdy ilustrują działanie wybranych wierszy funkcji `pop()` w przypadku, gdy usuwa ona węzeł ze stosu składającego się początkowo z trzech takich elementów. Proszę zwrócić uwagę, że ten rysunek jest uproszczony — po wywołaniu funkcji `free()` pamięć przeznaczona na zwolniony węzeł stosu nie znika, ani wskaźnik na na ten element wskazujący nie jest zerowany. Mimo to, nie powinniśmy się już odwoływać do tego węzła, ani nie powinniśmy posługiwać się tym wskaźnikiem do chwili ponownego przypisania mu poprawnego adresu, z powodów, które zostały wyjaśnione na pierwszym wykładzie.

Stos — implementacja

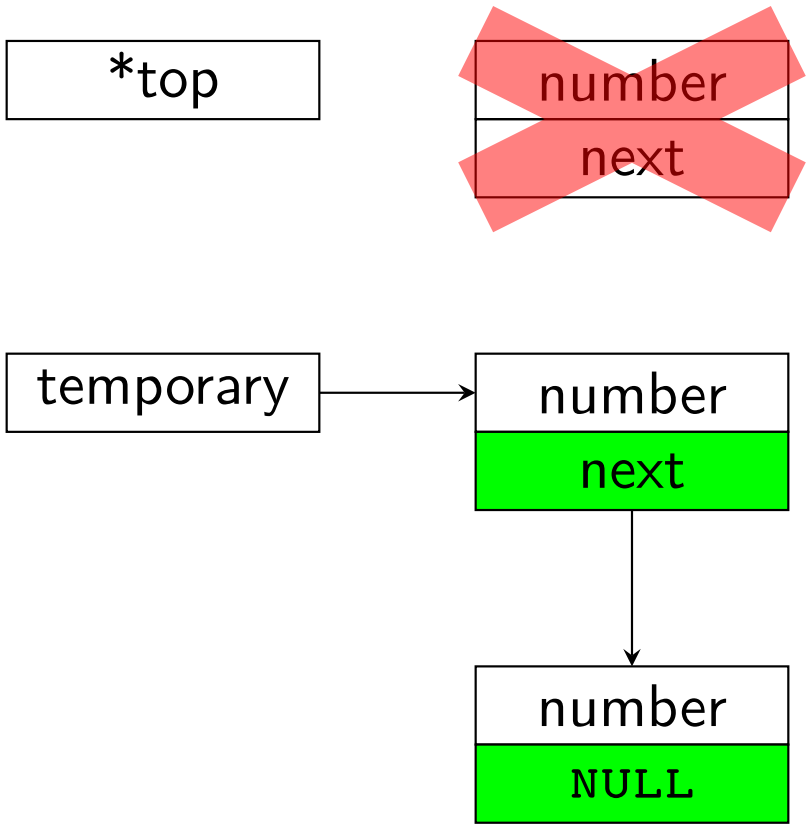
Funkcja pop()



Po wykonaniu wiersza nr 6 funkcji pop()

Stos — implementacja

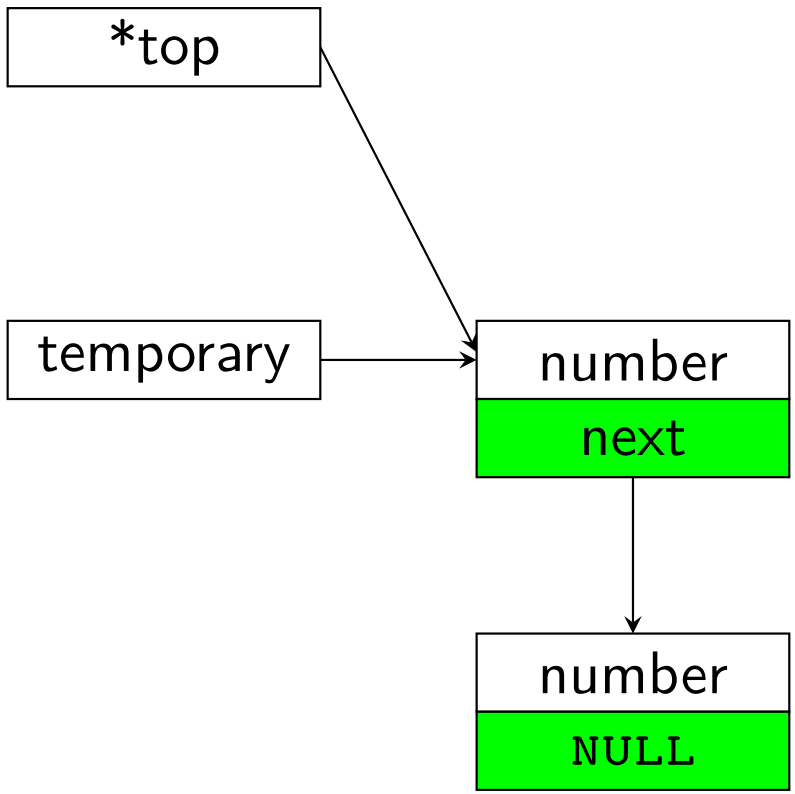
Funkcja pop()



Po wykonaniu wiersza nr 7 funkcji pop()

Stos — implementacja

Funkcja pop()



Po wykonaniu wiersza nr 8 funkcji pop()

Stos — implementacja

Funkcja peek() — opcjonalna

```
1  double peek(struct stack_node *top)
2  {
3      if(top)
4          return top->number;
5      else {
6          fprintf(stderr,"Stos jest pusty.\n");
7          return 0.0;
8      }
9  }
```

Stos — implementacja

Funkcja `peek()` — opcjonalna

Operacja *peek* jest opcjonalna — nie trzeba jej implementować w każdym stosie. Na tym wykładzie zostanie jednak przedstawiona przykładowa jej realizacja w postaci funkcji `peek()`. Jej definicja jest stosunkowo prosta. Przez parametr `top` przyjmuje ona wskaźnik na stos. Jeśli nie jest on pusty (warunek `top` jest skróconym zapisem warunku `top!=NULL`), to zwraca ona liczbę zapisaną w węźle stosu znajdującym się na jego wierzchołku, a jeśli nie, to umieszcza komunikat w standardowym strumieniu diagnostycznym (najczęściej jest to ekran monitora) i zwraca taką samą wartość umowną jak funkcja `pop()`.

Wycieki pamięci

Implementacja dynamicznej struktury danych może być obarczona wieloma poważnymi błędami. W przypadku stosu i pokrewnych struktur jednym z takich błędów jest brak spójności takiej struktury, polegający na tym, że jej elementy nie są prawidłowo ze sobą powiązane. Wyobraźmy np. sobie, że pewien nadgorliwy programista mógłby na początku funkcji `push()` zerować parametr `top`. To spowodowałoby, że każdy utworzony węzeł stosu nie byłby powiązany z pozostałymi. Co gorsza, na żaden z tych elementów, poza ostatnio utworzonym, nie wskazywałby żaden wskaźnik. Takich węzłów nie można by było już usunąć ze sterty. Obszary tej pamięci przydzielone na nie byłby bezpowrotnie stracone do końca działania programu. Taki błąd nazywa się w żargonie informatycznym *wyciekiem* lub *wyciekami pamięci* (ang. *memory leaks*). W najgorszym przypadku może on prowadzić do wyczerpania miejsca na stercie. Pierwszą obroną przed nim jest jego unikanie, czyli dokładne przeanalizowanie implementacji wszelkich operacji na strukturze danych. Istnieją też narzędzia programowe, od programów debugujących, po specjalne biblioteki, które ułatwiają wykrywanie tego typu błędów. Niestety, nie są one częścią standardu języka C, gdyż ich działanie zależy od użytego komputera i systemu operacyjnego.

Stos — zastosowania

Obliczanie wyrażeń w ONP

Stos używany jest do wyznaczania wartości wyrażeń arytmetycznych zapisanych w Odwrotnej Notacji Polskiej (ONP) (ang. *Reverse Polish Notation*), nazywanej też notacją *przyrostkową* lub *postfiks*. Notację tę opracował (wraz z innymi osobami) australijski informatyk i filozof Charles Hamblin, bazując na Notacji Polskiej (ang. *Polish Notation*), nazywanej też *przedrostkową* lub *prefiks*, wymyślonej przez polskiego logika Jana Łukasiewicza. Obie notacje są beznawiasowe, tzn. nie wymagają nawiasów, aby określić kolejność działań w wyrażeniu arytmetycznym. W Notacji Polskiej wszystkie operatory dwuargumentowe poprzedzają swoje argumenty. W ONP wszystkie operatory dwuargumentowe znajdują się za swoimi argumentami. Następny slajd zawiera kilka przykładów wyrażeń arytmetycznych zapisanych w tradycyjnej notacji wrostkowej (lub notacji *infiks*) oraz ONP.

Stos — zastosowania

Obliczanie wyrażeń w ONP

$2 + 2 \Rightarrow 2\ 2\ +$

$(5 - 2) * (4 + 1) \Rightarrow 5\ 2\ -\ 4\ 1\ +\ *$

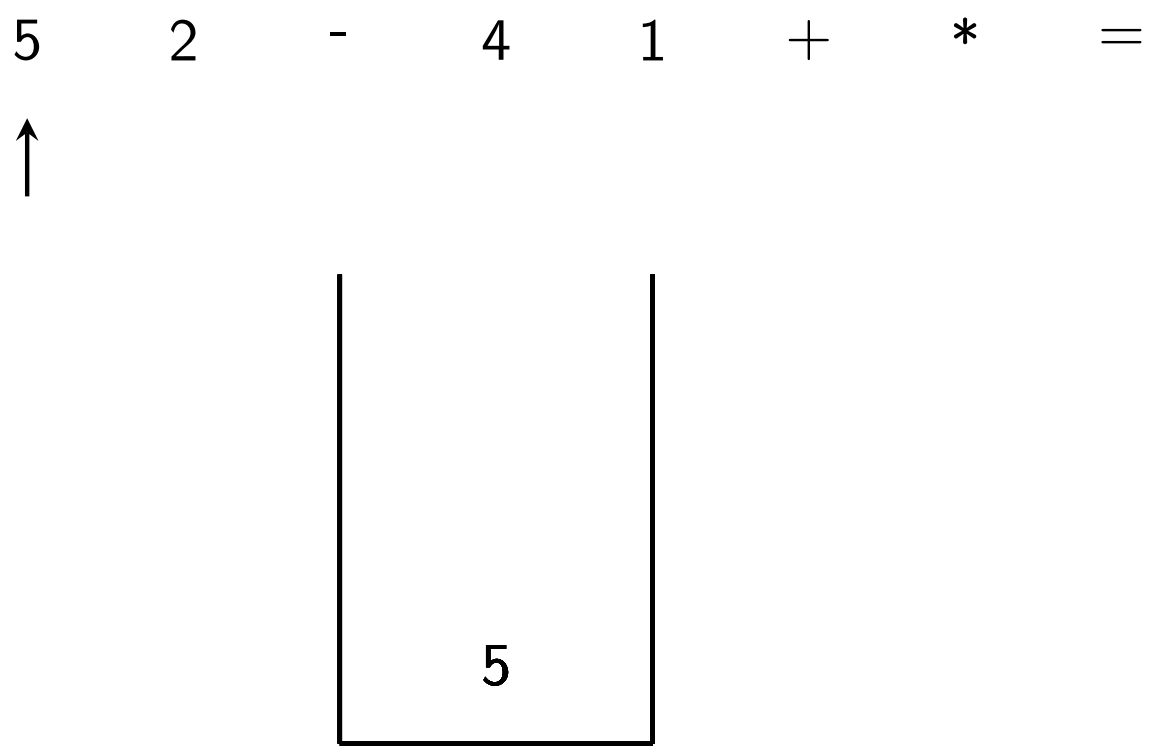
$(3 + 2) * 7 \Rightarrow 3\ 2\ +\ 7\ *$

$3 + 2 * 7 \Rightarrow 2\ 7\ *\ 3\ +$

Stos — zastosowania

Obliczanie wyrażeń w ONP

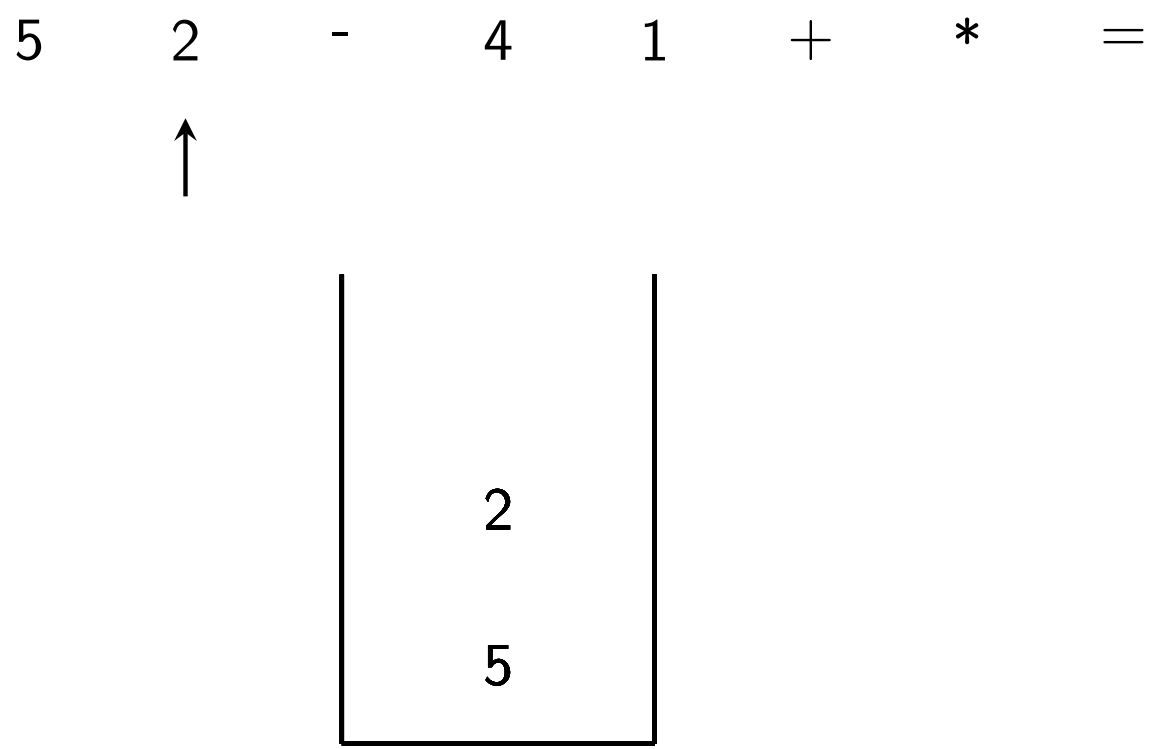
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos — zastosowania

Obliczanie wyrażeń w ONP

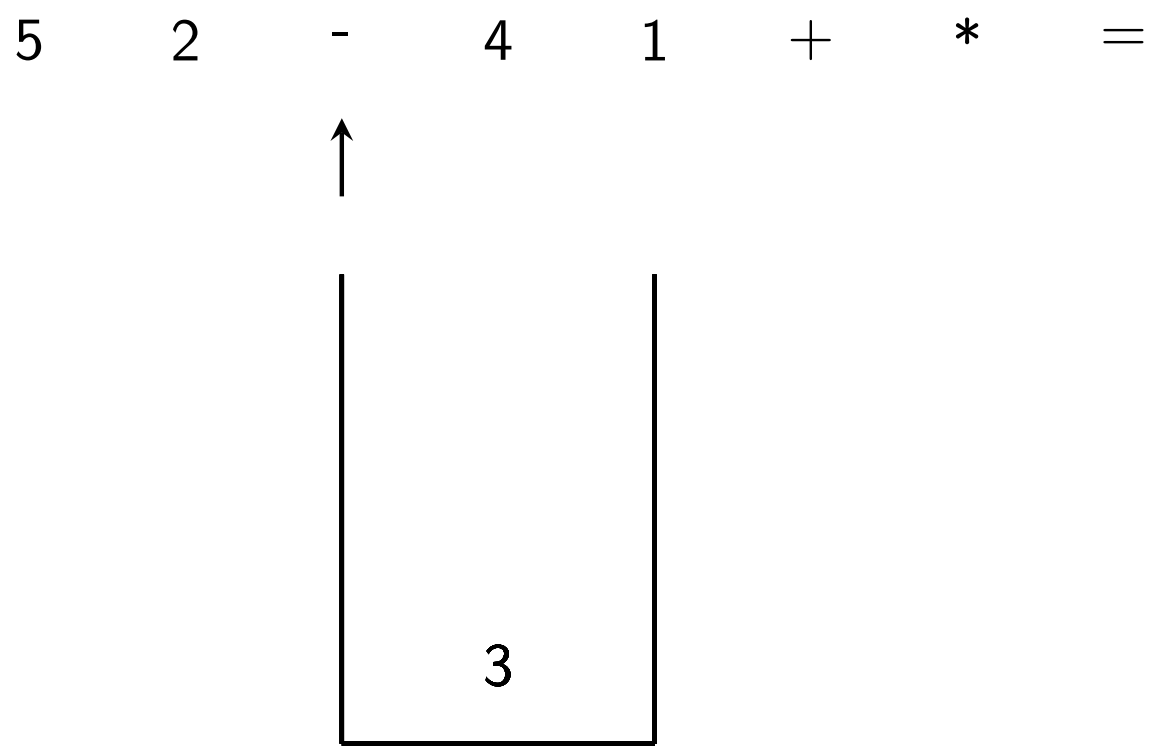
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos — zastosowania

Obliczanie wyrażeń w ONP

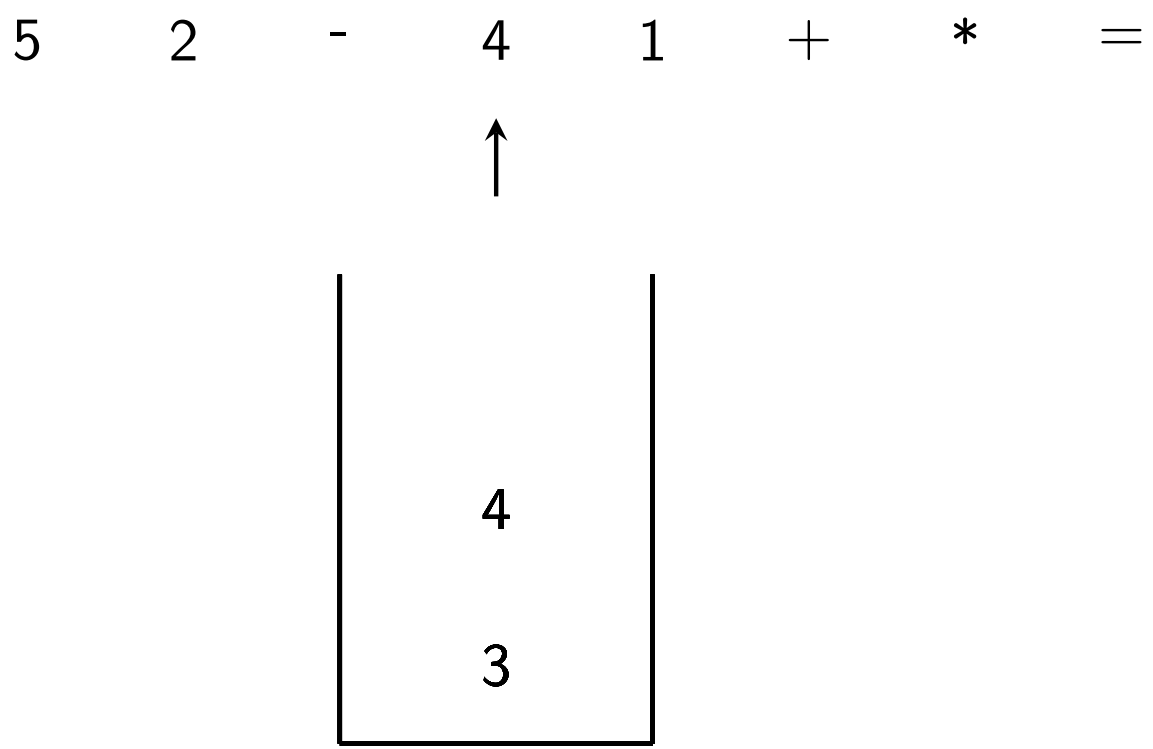
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos — zastosowania

Obliczanie wyrażeń w ONP

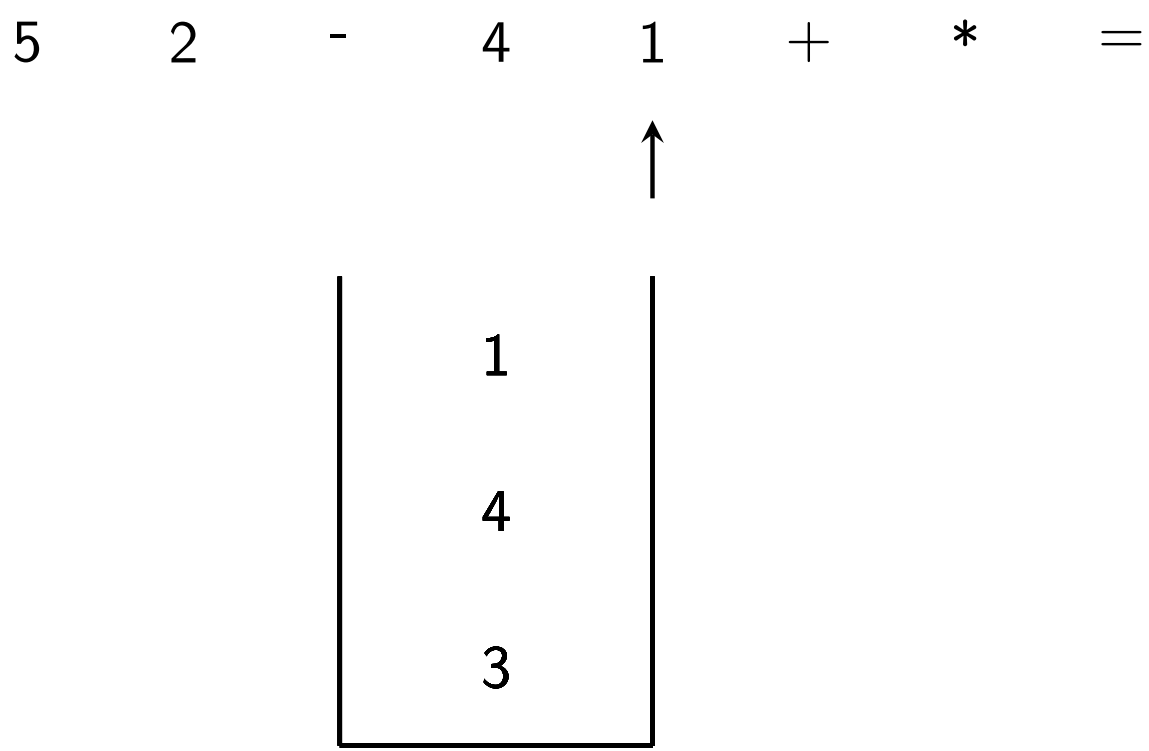
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos — zastosowania

Obliczanie wyrażeń w ONP

Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.

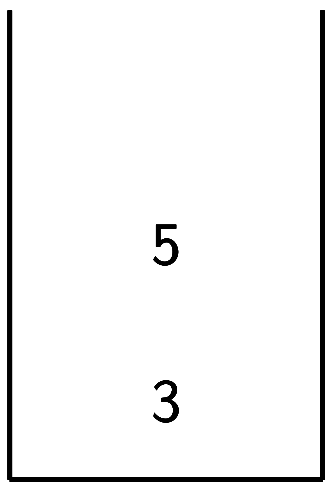


Stos — zastosowania

Obliczanie wyrażeń w ONP

Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.

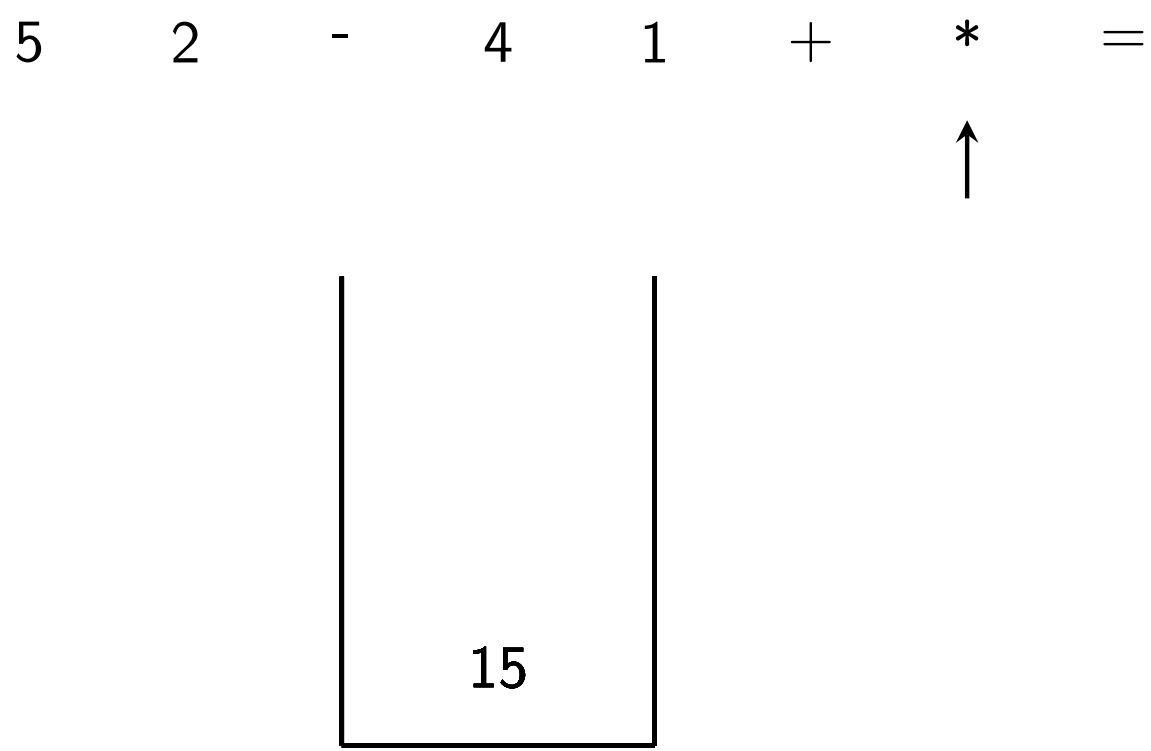
5 2 - 4 1 + * =



Stos — zastosowania

Obliczanie wyrażeń w ONP

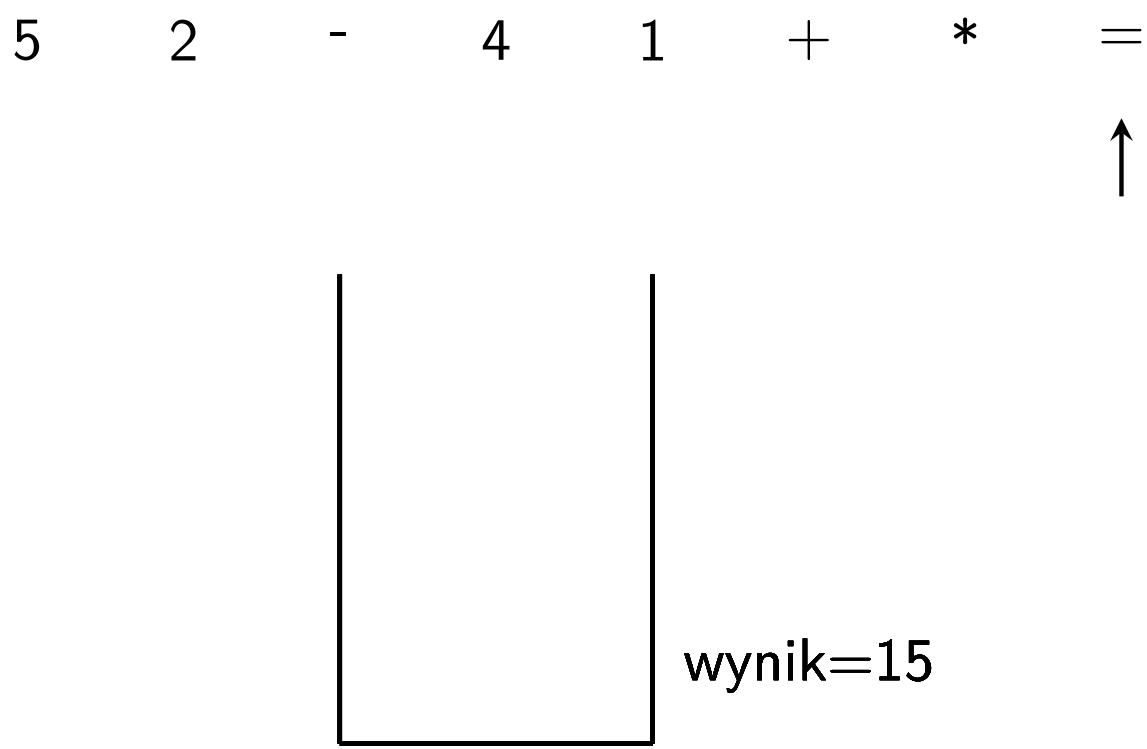
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos — zastosowania

Obliczanie wyrażeń w ONP

Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos — zastosowania

Obliczanie wyrażeń w ONP

Z zamieszczonej na poprzedniej stronie animacji wynika, że wyrażenie w ONP jest czytane od lewej strony. Jeśli napotkany jego element (token) jest liczbą, to jest ona odkładana na stos, a jeśli operatorem, to ze stosu pobierane są jego argumenty (jeśli wyrażenie jest poprawne, to będą one już na stosie), wykonywane jest działanie, a jego wynik umieszczany jest z powrotem na stosie. Pisząc program przyjmiemy następujące, upraszczające problem, założenia:

1. wyrażenie składa się wyłącznie z nieujemnych liczb zmiennoprzecinkowych oraz czterech działań: dodawania, odejmowania, mnożenia i dzielenia
2. tokeny w wyrażeniu ONP są rozdzielone pojedynczą spacją,
3. token = kończy wyrażenie i nakazuje programowi podać jego wartość,
4. program zakłada, że wyrażenie jest prawidłowo zapisane.

Stos — zastosowania

Obliczanie wyrażeń w ONP

Program obliczający wartość wyrażenia w ONP korzysta tylko z dwóch operacji na stosie — *push* i *pop*. Jego kod rozpoczyna się od trzech dyrektyw włączających pliki nagłówkowe `stdio.h`, `stdlib.h` i `string.h`. Po nich następują definicje typu danych węzła stosu oraz funkcji `push()` i `pop()`.

Stos — zastosowania

Obliczanie wyrażeń w ONP

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

struct stack_node {
    double number;
    struct stack_node *next;
};
```

Stos — zastosowania

Obliczanie wyrażeń w ONP

```
struct stack_node *push(struct stack_node *top, double
↪   number)
{
    struct stack_node *new_node = (struct stack_node *)
↪   malloc(sizeof(struct stack_node));
    if (new_node) {
        new_node->number = number;
        new_node->next = top;
        top = new_node;
    }
    return top;
}
```

Stos — zastosowania

Obliczanie wyrażeń w ONP

```
double pop(struct stack_node **top)
{
    double result = 0.0;
    if (*top) {
        result = (*top)->number;
        struct stack_node *temporary = (*top)->next;
        free(*top);
        *top = temporary;
    }
    return result;
}
```

Stos — zastosowania

Obliczanie wyrażeń w ONP

```
double evaluate(char *token)
{
    static struct stack_node *top = 0;
    double first_argument, second_argument;
    switch (token[0]) {
    case '+':
        top = push(top, pop(&top) + pop(&top));
        break;
    case '*':
        top = push(top, pop(&top) * pop(&top));
        break;
    case '-':
        second_argument = pop(&top);
        first_argument = pop(&top);
        top = push(top, first_argument - second_argument);
        break;
```

Stos — zastosowania

Obliczanie wyrażeń w ONP

```
        case '/':
            second_argument = pop(&top);
            first_argument = pop(&top);
            top = push(top, first_argument/second_argument);
            break;
        case '=':
            return pop(&top);
        default:
            top = push(top, atof(token));
        }
        return 0.0;
    }
```

Stos — zastosowania

Obliczanie wyrażeń w ONP

Funkcja `evaluate()` przyjmuje jeden argument, którym jest łańcuch znaków reprezentujący pojedynczy token wyrażenia w ONP. Jako jedyna w programie używa ona stosu, którego rolę wskaźnika pełni lokalna zmienna o nazwie `top`, zadeklarowana z użyciem słowa kluczowego `static`. To słowo zapewnia, że adresy przechowywane w tej zmiennej nie zostaną utracone między kolejnymi wywołaniami `evaluate()`. Ta funkcja sprawdza w instrukcji `switch` pierwszy znak, który rozpoczyna przekazany jej łańcuch znaków reprezentujący token. Jeśli ten symbol jest jedynym z dwuargumentowych operatorów (plus, minus, razy, przez), to pobiera przy pomocy funkcji `pop()` dwa argumenty ze stosu, wykonuje odpowiednie działanie i umieszcza na stosie wynik używając do tego funkcji `push()`. Zauważmy, że w przypadku odejmowania i dzielenia, argumenty są najpierw przypisywane do dwóch zmiennych lokalnych. To jest konieczne, bo te działania są nieprzemienne, a ich argumenty są umieszczone na stosie w odwrotnej kolejności.

Stos — zastosowania

Obliczanie wyrażeń w ONP

Jeśli tokenem jest symbol =, to `evaluate()` usuwa jedyny węzeł stosu przy użyciu funkcji `pop()` i zwraca liczbę zwróconą przez tę funkcję, ponieważ jest to wartość obliczanego wyrażenia w ONP. Jeśli token nie jest żadnym z opisanych wcześniej symboli, to oznacza, że jest łańcuchem reprezentującym liczbę zmiennoprzecinkową. W związku z tym `evaluate()` konwertuje go na liczbę typu `double` przy pomocy funkcji `atof()` i zapisuje ją na stosie.
Funkcja `evaluate()` działa prawidłowo tylko wtedy, gdy oblicza wartość prawidłowego wyrażenia w ONP. Wskaźnik stosu (zmienna `top`) ma nadaną wartość początkową 0, która jest zamiennikiem `NULL`. Zmienne lokalne zadeklarowane z użyciem słowa `static` mają właśnie taką wartość domyślną, więc ta inicjacja jest w tym przypadku zbędna, ale prawidłowa. Funkcja `evaluate()` zwraca 0 za każdym razem, gdy wywoływana jest dla tokena, który nie jest symbolem =.

Stos — zastosowania

Obliczanie wyrażeń w ONP

```
double parse(char expression[])
{
    double result = 0.0;
    char *token = strtok(expression, " ");
    while (token) {
        result = evaluate(token);
        token = strtok(0, " ");
    }
    return result;
}
```

Stos — zastosowania

Obliczanie wyrażeń w ONP

Funkcja `parse()` jest odpowiedzialna za podzielenie przekazanego jej łańcucha reprezentującego wyrażenie w ONP na tokeny. Znakiem oddzielającym tokeny jest w tym przypadku pojedyncza spacja. Każdy z otrzymanych tokenów jest przekazywany w pętli `while` do funkcji `evaluate()`, celem jego rozpoznania. Wynik tej funkcji jest zapisywany w zmiennej `result`. Po zakończeniu pętli wartość wyrażenia znajduje się w tej zmiennej i jest zwracana przez `parse()`.
W terminologii informatycznej operacja wykonywana razem przez funkcje `parse()` i `evaluate()` nazywa się *parsowaniem* (ang. *parsing*).

Stos — zastosowania

Obliczanie wyrażeń w ONP

```
int main(void)
{
    char rpn_expression[201];

    puts("Proszę wprowadzić wyrażenie w ONP:");
    scanf("%200[^\n]s", rpn_expression);
    double result = parse(rpn_expression);
    printf("Wynik: %.3lf\n", result);
    return 0;
}
```

Stos — zastosowania

Obliczanie wyrażeń w ONP

W funkcji `main()` program prosi użytkownika o wprowadzenie wyrażenia w notacji przyrostkowej i zapisuje łańcuch, które je reprezentuje w zmiennej lokalnej `rpn_expression`. Ten ciąg znaków jest następnie przekazywany do funkcji `parse()`, której wynik jest zapisywany do zmienne lokalnej `result` i wyświetlany na ekranie.

Kolejka

Podobnie jak stos, kolejka (nazywana również *kolejką* FIFO lub krótko FIFO) może być zaimplementowana w postaci dynamicznej struktury danych. W takim przypadku jest to lista powiązanych ze sobą węzłów, gdzie nowy węzeł jest dodawany na końcu (nazywanym także *ogonem* (ang. *tail*) lub *tyłem* (ang. *rear*) kolejki), a usuwanie węzła odbywa się na początku (nazywanym też *czołem* (ang. *head*) lub *przodem* (ang. *front*) kolejki).
Implementacja kolejki w postaci dynamicznej struktury danych zostanie zaprezentowana w tym wykładzie przy pomocy programu, który używa jej do przechowywania argumentów wiersza poleceń.

Kolejka

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<stdbool.h>
4  #include<string.h>
5
6  #define LENGTH 500
7
8  struct fifo_node {
9      char data[LENGTH];
10     struct fifo_node *next;
11 };
12
13 struct fifo_pointers {
14     struct fifo_node *head, *tail;
15 } fifo;
```

Kolejka

Do programu są dołączane cztery pliki nagłówkowe. Plik `stdio.h` został dodany, ponieważ program używa funkcji `printf()`. Funkcje niezbędne do zarządzania stertą są zadeklarowane w pliku nagłówkowym `stdlib.h`. Niektóre z funkcji zdefiniowanych w programie zwracają wartości typu `bool`, dlatego został dołączony plik `stdbool.h`. Program używa także funkcji, która przeprowadza operacje na ciągu znaków, stąd obecność pliku `string.h`.
Maksymalna długość ciągu znaków przechowywanych w pojedynczym węźle kolejki jest określona przez stałą `LENGTH` i wynosi 500 znaków.
Typ danych pojedynczego węzła kolejki jest zdefiniowany jako struktura z dwoma polami (wiersze 8–11). Jednym z tych pól jest tablica znaków (wiersz nr 9), a drugim wskaźnik o tym samym typie, co ta struktura (wiersz nr 10). Proszę zwrócić uwagę na podobieństwo między definicjami typów danych dla węzła stosu i kolejki. Pole wskaźnikowe ma takie samo przeznaczenie, jak w przypadku stosu — jest używane do łączenia węzłów w kolejce.

Kolejka

Aby operacje na kolejce mogły być realizowane w sposób efektywny potrzebne są dwa wskaźniki: jeden na początek, a drugi na koniec kolejki. Można je zadeklarować jako osobne zmienne, ale wygodniej jest je uczynić polami w strukturze. W programie ta struktura nazywa się `fifo` i jest zadeklarowana w wierszu nr 15. Z kolei jej typ danych jest zdefiniowany w wierszach 13–15. Ma ona dwa pola będące wskaźnikami typu `struct fifo_node *`.

Kolejka

```
1  void copy_string(char *destination, char *source)
2  {
3      strncpy(destination,source,LENGTH-1);
4      destination[LENGTH-1] = '\0';
5  }
```

Kolejka

Funkcja `copy_string()` kopiuje łańcuch znaków przekazanych jej przez parametr `source` do tablicy, której adres jest przekazany jej przez parametr `destination`, w sposób bardziej bezpieczny, niż robi to funkcja `strcpy()`. Kopiuje ona co najwyżej `LENGTH-1` znaków ze źródłowego ciągu i zakańcza ciąg w tablicy wskazywanej przez `destination`, dodając znak `'\0'`, na wypadek gdyby ciąg źródłowy nie był nim zakończony.
Funkcja może uciąć część znaków należących do oryginalnego łańcucha. W opisywanym programie nie stanowi to problemu, ale w innym oprogramowaniu powinna ona być stosowana ostrożnie.

Kolejka

Operacja *enqueue*

Zakładamy, że implementacja operacji dodania nowego węzła do kolejki (ang. *enqueue*) będzie spełniała następujące warunki:

- Jeśli kolejka ma już co najmniej jeden węzeł, to nowy element jest dodawany na jej końcu, a jeśli kolejka jest pusta, to operacja tworzy i dodaje jej pierwszy węzeł.
- Jeśli nie uda się utworzyć nowego węzła, to stan kolejki pozostaje bez zmian.
- W wyniku pomyślnego zakończenia operacji kolejka powiększa się o nowy element.

Kolejka

Operacja *enqueue*

```
1  bool enqueue(struct fifo_pointers *fifo, char *data)
2  {
3      struct fifo_node *new_node = (struct fifo_node
   ↪   *)malloc(sizeof(struct fifo_node));
4      if(new_node) {
5          copy_string(new_node->data,data);
6          new_node->next = NULL;
7          if(fifo->head == NULL && fifo->tail == NULL) {
8              fifo->head = fifo->tail = new_node;
9          } else {
10             fifo->tail->next = new_node;
11             fifo->tail = new_node;
12         }
13         return true;
14     } else
15         return false;
16 }
```

Kolejka

Operacja *enqueue*

Operacja *enqueue* została zaimplementowana w postaci funkcji o takiej samej nazwie. Przyjmuje ona dwa argumenty przekazywane jej przez parametry. Pierwszym jest adres struktury zawierającej wskaźniki kolejki, a drugim adres łańcucha, który będzie skopiowany do nowego węzła. Wartość zwracana przez funkcję jest typu `bool`.

Na początku funkcja `enqueue()` próbuje przydzielić pamięć na nowy węzeł (wiersz nr 3). To, czy ta operacja się powiodła jest sprawdzane w wierszu nr 4. Jeśli nie, to funkcja zwraca wartość `false` (wiersz nr 15) i kończy działanie. W przeciwnym przypadku wywołuje ona funkcję `copy_string()`, by skopiować łańcuch, którego adres został jej przekazany przez parametr `data`, do pola `data` nowego węzła (wiersz nr 5) i przypisuje polu `next` tego elementu wartość `NULL` (wiersz nr 6). Następnie funkcja `enqueue()` sprawdza, który z dwóch możliwych przypadków musi obsłużyć:

1. węzeł jest dodawany do pustej kolejki,
2. węzeł jest dodawany na końcu niepustej kolejki.

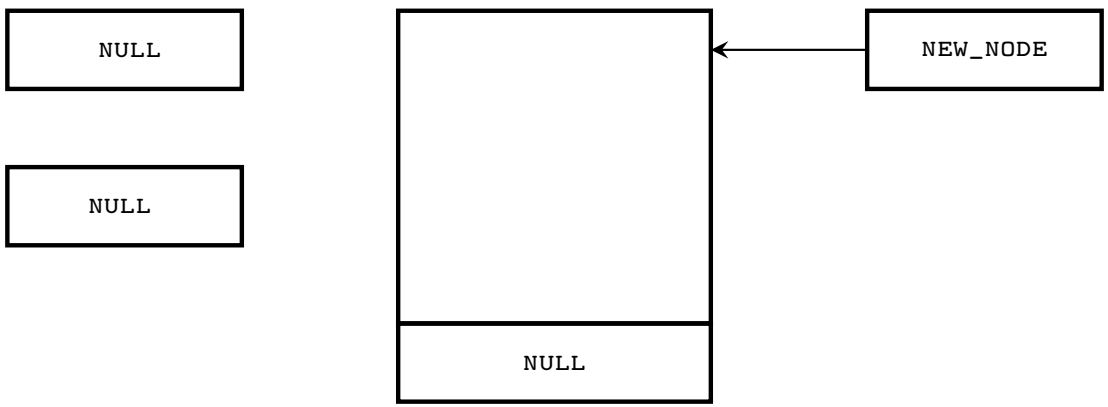
Kolejka

Operacja *enqueue*

Aby to ustalić sprawdza ona wartości wskaźników kolejki (wiersz nr 7). Jeśli oba są wskaźnikami pustymi, to pierwszy przypadek musi być obsłużony i funkcja `enqueue()` przypisuje im obu adres nowego węzła (wiersz nr 8), co jest zilustrowane na następnym slajdzie.
W drugim przypadku (wiersze 10–11) funkcja najpierw zapisuje adres nowego węzła w polu `next` węzła bieżąco znajdującego się na końcu kolejki (wiersz nr 10). To powoduje dodanie tego nowego elementu do kolejki, ale po tej operacji wskaźnik `tail` wskazuje na niewłaściwy węzeł — powinien zawsze wskazywać ostatni element w kolejce. Z tego powodu w wierszu nr 11 funkcja `enqueue()` przypisuje do tego wskaźnika adres nowego elementu. Ta operacja jest zilustrowana na kolejnym po następnym slajdzie. Zauważmy, że w obu przypadkach do pola `next` nowego węzła musi być zapisana wartość `NULL`, bo sygnalizuje ona, że jest to ostatni węzeł w kolejce. Po pomyślnym dodaniu nowego węzła do kolejki, niezależnie od tego, który przypadek wystąpił, `enqueue()` zwraca wartość `true`.

Kolejka

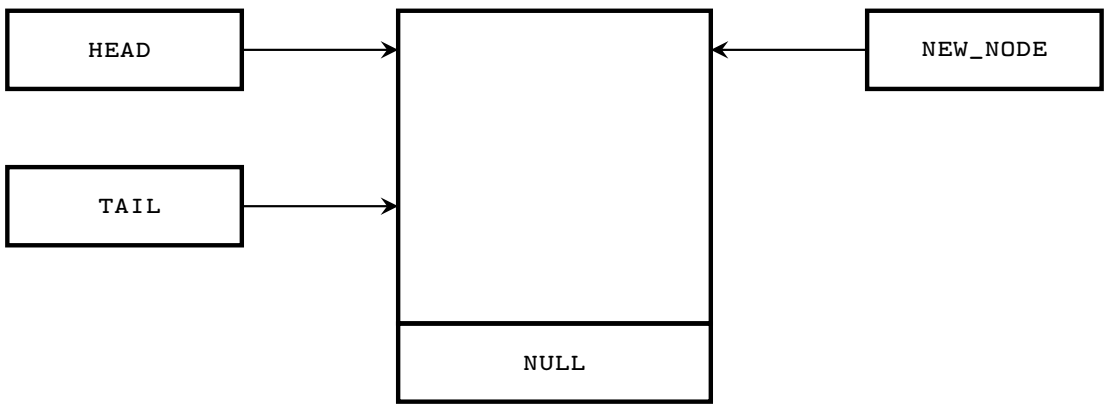
Operacja *enqueue*



Kolejka przed wykonaniem 8 wiersza funkcji `enqueue()`

Kolejka

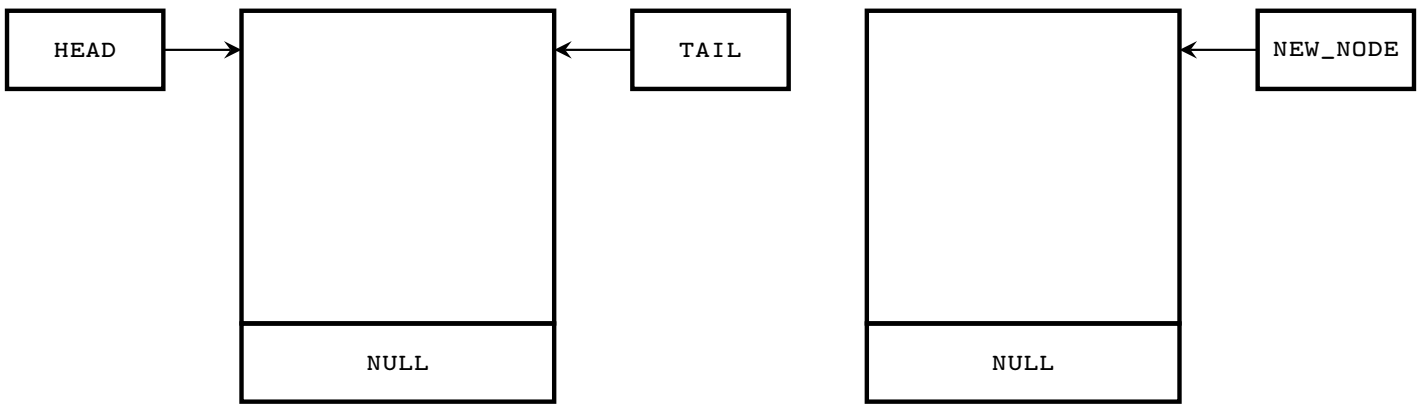
Operacja *enqueue*



Kolejka po wykonaniu 8 wiersza funkcji `enqueue()`

Kolejka

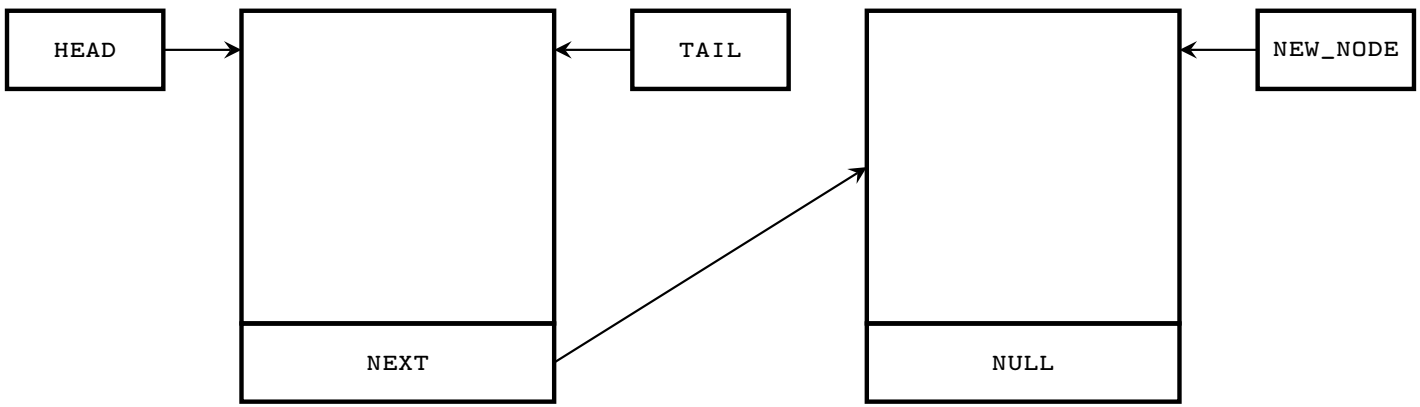
Operacja *enqueue*



Przed wykonaniem 10 wiersza funkcji `enquene()`

Kolejka

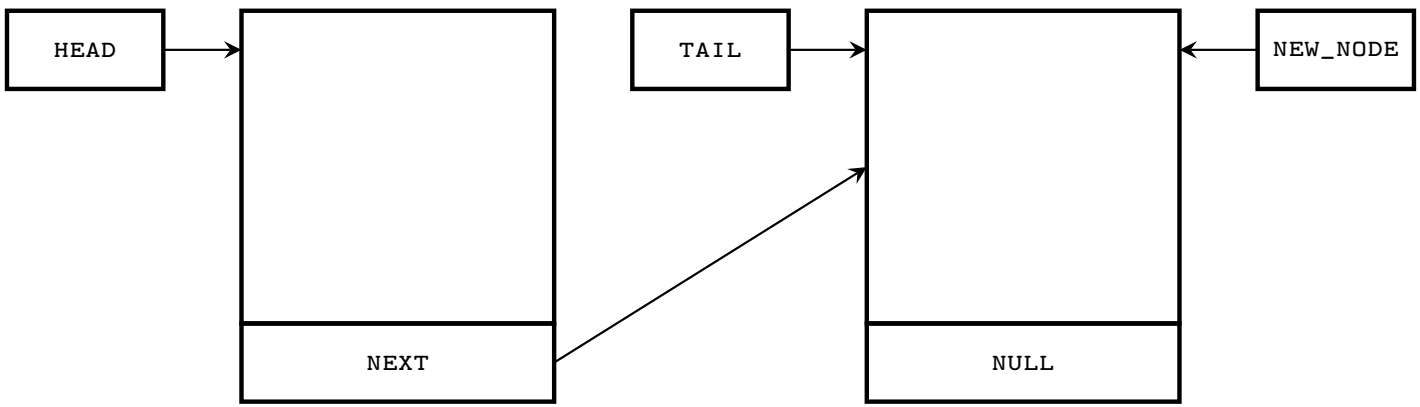
Operacja *enqueue*



Po wykonaniu 10 wiersza funkcji enquene()

Kolejka

Operacja *enqueue*



Po wykonaniu 11 wiersza funkcji enquene()

Kolejka

Operacja *dequeue*

Operacja *dequeue* usuwa element z początku (czoła) kolejki i zakładamy, że powinna spełniać następujące warunki:

- Jeśli kolejka jest pusta to stan jej wskaźników po wykonaniu operacji usuwania węzeła nie może ulec zmianie — wartości obu powinny wynosić `NULL`.
- Jeśli usuwany jest węzeł z kolejki o jedyny węźle, to po wykonaniu tej operacji oba wskaźniki kolejki powinny mieć wartość `NULL`.
- Jeśli usuwany jest element z kolejki składającej się z więcej niż jednego węzła, to jej długość zostaje zmniejszona o jeden element, a po wykonaniu tej operacji wskaźniki kolejki poprawnie wskazują jej początek i koniec.

Kolejka

Operacja *dequeue*

```
1  bool dequeue(struct fifo_pointers *fifo, char *data)
2  {
3      if(fifo->head) {
4          struct fifo_node *temporary = fifo->head->next;
5          copy_string(data,fifo->head->data);
6          free(fifo->head);
7          fifo->head = temporary;
8          if(temporary == NULL)
9              fifo->tail = NULL;
10         return true;
11     }
12     return false;
13 }
```

Kolejka

Operacja *dequeue*

Operacja *dequeue* została zaimplementowana w postaci funkcji o tej samej nazwie. Przyjmuje ona dwa argumenty. Pierwszym jest adres struktury ze wskaźnikami kolejki, który jest przekazany przez jej pierwszy parametr. Przez drugi parametr przekazywany jest adres tablicy znaków.

Najpierw funkcja sprawdza, czy wskaźnik `head`, który powinien wskazywać na pierwszy węzeł kolejki, nie jest pusty (wiersz nr 3). Jeśli tak jest, to zapisuje ona adres przechowywany w polu `next` tego węzła w lokalnym wskaźniku o nazwie `temporary` (wiersz nr 4). Jest to adres drugiego węzła w kolejce. Następnie funkcja `dequeue()` kopiuje łańcuch z pola `data` pierwszego węzła do tablicy, której adres został przekazany jej przez drugi parametr (wiersz nr 5). Potem zwalnia ona pamięć przydzieloną na ten węzeł (wiersz nr 6). Po przeprowadzeniu tej operacji wskaźnik `head` staje się nieprawidłowy, bo wskazuje na nieistniejący już węzeł.

Kolejka

Operacja *dequeue*

W związku z tym w wierszu nr 7 do wskaźnika `head` przypisywany jest adres przechowywany w zmiennej `temporary`. Jest to adres poprzednio drugiego, a obecnie pierwszego węzła kolejki. W przypadku gdyby `dequeue()` usunęła jedyny element z kolejki i tym samym kolejka stała się pusta, to po wykonaniu wiersza nr 7 oba wskaźniki, tj. `temporary` i `head` będą miały wartość `NULL`, co jest oczekiwanym stanem. Jednakże w tej sytuacji pozostaje jeszcze jeden wskaźnik, który będzie miał wartość nieprawidłową. Jest nim `tail`, który wskazuje na nieistniejący węzeł. Dlatego funkcja `dequeue()` w wierszu nr 8 sprawdza wartość wskaźnika `temporary`, by wykryć taki przypadek. Jeśli warunek w tym wierszu będzie spełniony, to przypisze ona wskaźnikowi `tail` wartość `NULL`. Po pomyślnym usunięciu węzła z kolejki `dequeue()` zwraca wartość `true`. Jeśli warunek w wierszu nr 3 nie jest spełniony, to oznacza to, że kolejka jest pusta i funkcja `dequeue()` sygnalizuje to zwracając wartość `false` (wiersz nr 12).

Kolejka

```
1  int main(int argc, char *argv[])
2  {
3      for(int i=0; i<argc; i++)
4          if(!enqueue(&fifo, argv[i]))
5              printf("Argument: %s nie został dodany do
   ↪  kolejki!",argv[i]);
6      while(fifo.head) {
7          char string[LENGTH];
8          if(dequeue(&fifo,string))
9              printf("Dane z kolejki: %s\n", string);
10     }
11     return 0;
12 }
```

Kolejka

W funkcji `main()` program dodaje w pętli `for` otrzymane argumenty wiersza poleceń do kolejki (wiersze 3–5), sprawdzając w każdej iteracji, czy ta operacja się powiodła (wiersz nr 4). Następnie, w pętli `while`, pobiera on te argumenty z kolejki i wyświetla po kolei na ekranie (wiersze 6–10). Proszę zauważyć, że ścieżka do pliku wykonywalnego programu również jest dodawana do kolejki — jest ona wskazywana przez `argv[0]`. Pętla `while` wykonywana jest do momentu aż wskaźnik `head` stanie się pusty. Dodatkowo program sprawdza w jej ciele, czy funkcja `dequeue()` zwróciła wartość `true`, zanim wypisze zawartość tablicy `string` na ekranie.

Podsumowanie

Zarówno stos, jak i kolejka mogą być zrealizowane na kilka innych sposobów. Mogą one zostać zaimplementowane zarówno sprzętowo jaki i w postaci oprogramowania. Można je również utworzyć na bazie tablicy, na którą pamięć jest przydzielana statycznie lub dynamicznie. W tym przypadku węzłami stosu lub kolejki są elementy tej tablicy, a zamiast wskaźników są zazwyczaj używane indeksy.
Obie struktury mają wiele zastosowań, np. w kompilatorach, systemach operacyjnych, jak i innego typu oprogramowaniu. Warto wspomnieć, że istnieje algorytm opracowany przez Edsgara Dijkstrę, noszący nazwę *algorytmu stacji rozrządowej* (ang. *shunting-yard algorithm*), który pozwala na konwersję wyrażeń w notacji infiksowej do postfiksowej. On również używa stosu, ale w przeciwieństwie do algorytmu, który wyznacza wartość wyrażeń w ONP, przechowuje w nim operatory i nawiasy zamiast liczb.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!