Ćwiczenia z ASD

Michał Knapik

Ostatnio kompilowane: 16 maja 2021

Ćwiczenia 1

Poprawność częściowa i pełna algorytmów

1.1 Rozgrzewka

Zadanie 1.1.1. *Rozważmy następujące funkcje:*

```
FUN1(n):
1.     k = 1
2.     while (k < n):
3.           k := 2*k
4.     return k

FUN2(n):
1.      k = 1
2.      while (k*k < n):
3.           k := k + 1
4.      return k
```

Operacje dominujące w `FUN1` *i* `FUN2` *to działania arytmetyczne. Oblicz dokładną złożoność obliczeniową obu funkcji.*

Zadanie 1.1.2. *Uszereguj niemalejąco względem rzędu następujące funkcje:*

1. $n!$
2. $0.001n^5 + n^3$
3. $\log(n!)$
4. $n^{1000} - n^{999}$
5. $10000n^4 + 999n^3$,
6. $(1.000001)^n$
7. $\log(\log(n))$
8. $n\log(n)$
9. n^n

Zadanie 1.1.3. *W dalszej części zajęć bedziemy zajmowali się m.in. sortowaniem. Dobre algorytmy sortowania pozwalają na posortowanie tablicy o rozmiarze n wykonując około nlogn operacji elementarnych. Słabsze algorytmy mogą wymagać n^2 operacji elementarnych.*

Skrót MIPS oznacza liczbę milionów operacji na sekundę. Procesor Zilog Z80 (opracowany w 1976) może wykonać 0.5 MIPS. Procesor AMD Ryzen Threadripper 3990X (2020) może wykonać 2300000 MIPS. Przyjrzymy się, jak brutalna siła obliczeniowa ma się do dobrego wyboru algorytmów.

1. *Dobry programista opracował algorytm sortujący w czasie nlogn i uruchomił go na maszynie z procesorem Z80. Słabszy programista napisał program sortujący w czasie n^2 i uruchomił go na maszynie z procesorem AMD Ryzen Threadripper 3990X.* **Znajdź wielkość tablicy którą Z80 posortuje szybciej niż AMD Ryzen.**

2. *Z poprzedniego punktu wynika, iż tablica taka będzie bardzo duża. Czy rzeczywiście jest to sukces? Ile czasu potrwa sortowanie tej tablicy przy pomocy maszyny AMD? A ile czasu potrwałoby, gdyby zastosowano na tej maszynie algorytm sortujący o złożoności ok. nlogn?*

Przykładowe rozwiązanie:

```
def compare_exec_time(mips1, mips2, n):
    """True iff the worse algorithm is faster or data of size n."""
    return n**2/mips1 < n*math.log(n)/mips2

n = 10
while compare_exec_time(2300000, 0.5, n):
      n += 100
print(f'Z80 starts winning at n={n}')

print(f'It would take AMD {(n**2/2300000)/(60*60*24*365)}'
       'years using slower algorithm...')
print(f'and {(n*math.log(n,2)/2300000)/(60)} minutes with faster?!')
```

Zadanie 1.1.4. *Porównaj następujące funkcje:*

```
FUN3(n):
1.     m = 1
2.     for i = 1 to 3
3.           m := m*m
4.           for j = 1 to n
5.               m := m+j
4.     return m

FUN4(n):
1.     m = 1
2.     for i = 1 to n
3.           m := m*m
4.           for j = 1 to 3
5.               m := m+j
4.     return m
```

Która ma większą złożoność obliczeniową? Oblicz szybko oszacowanie dolne na wartość (nie złożoność) $FUN3(n)$ i $FUN4(n)$, używając notacji Ω. Jakiego rodzaju gwarancje daje to oszacowanie? Przetestuj eksperymentalnie implementację w Pythonie.

1.2 Podstawowe definicje

Algorytm ma własność pełnej poprawności, gdy dla wszystkich danych wejściowych:

1. jeśli dane wejściowe są poprawne (zgodne ze specyfikacją wejścia), to algorytm po zatrzymaniu się zwróci poprawny wynik (zgodny ze specyfikacją),

2. jeśli dane wejściowe są poprawne, to algorytm w końcu zatrzyma się.

Algorytm jest częściowo poprawny, gdy spełniony jest pierwszy z powyższych warunków. Drugi warunek jest zazwyczaj nazywany warunkiem stopu i jest, swoją drogą, nierozstrzygalny - tzn. nie istnieje algorytm U, który biorąc pseudokod dowolnego algorytmu A i dowolne dane wejściowe x powie nam, czy A zatrzyma się po wykonaniu na danych x.

Zadanie 1.2.1. *Napisać jednolinijkowy algorytm, trywialnie częściowo poprawny dla dowolnych danych.*

Przykładowe rozwiązanie:

```
Alg(x):
    while (true) PASS
```

Powyższy algorytm jest częściowo poprawny, niezależnie od specyfikacji - bowiem nigdy się nie zatrzyma.

Specyfikacja danych wejściowych składa się zazwyczaj z opisu typu, wraz z ew. dodatkowym jego zawężeniem. Poprawne dane to dane zgodne z tą specyfikacją, często wyrażoną jako formuła pewnej logiki. Poprawność zwracanego wyniku weryfikowana jest w podobny sposób - powinna być zgodna z pewną specyfikacją. Istnieją języki takie jak Ada-Spark, w których można formułować w/w warunki poprawności explicite i próbować statycznie weryfikować programy. Języki te są używane dość sporadycznie, głównie w tzw. zastosowaniach krytycznych.

1.3 Metoda niezmienników

Metoda niezmienników służy do dowodzenia poprawności programów. Niezmiennik pętli to warunek (specyfikowany formalnie lub w języku bardziej naturalnym), który jest:

- prawdziwy przed pierwszą iteracją pętli (*inicjalizowanie*);
- oraz jeśli był prawdziwy przed daną iteracją pętli, to będzie prawdziwy i po niej (*utrzymanie*).

Jeśli więc udowodnimy, że dana własność jest niezmiennikiem pętli oraz pętla zatrzyma się po skończonej liczbie iteracji, to wykażemy, że niezmiennik będzie prawdziwy po zakończeniu pętli. Oczywiście musi być on zbudowany tak zręcznie, by wynikało z tego, że badany program jest poprawny.

Zadanie 1.3.1 (Schemat Hornera)**.** *Celem jest efektywne obliczenie wartości wielomianu* $f(x) = \sum_{i=0}^{n} a_i x^i$. *Jako dane wejściowe podana jest tablica współczynników wielomianu* $A[0, \ldots, n]$. *Algorytm powinien realizować obliczenia zgodnie ze wzorcem:*

$$f(x) = a_0 + x(a_1 + x(a_2 + \ldots x(a_{n-1} + xa_n))).$$

Zapisz pseudokod algorytmu i zbadaj pełną poprawność algorytmu.

Przykładowe rozwiązanie:

```
HORNER(A,x):
1.    res := 0
2.    i := len(A) - 1
3.
4.    while (i >= 0):
5.        res += A[i] + x*res
6.        i--
7.
8.    return res
```

Warunek stopu jest spełniony w oczywisty sposób - pętla while musi się zatrzymać dla poprawnych danych wejściowych (a więc domyślnie dla $i = n$, gdzie $n \in \mathbb{N}$), i to w $n + 1$ krokach.

Niezmiennik pętli. Zacznijmy od przykładu, który pozwoli na zbudowanie pewnej intuicji. Załóżmy, że $f(x) = 9x^3 - 3x^2 + 7x - 4$ i przyjrzyjmy się zawartości zmiennej `res` podczas kolejnych iteracji:

- Gdy `i` $=$ `0` to `res` $=$ `0` przed wykonaniem iteracji i `res` $=$ `9` po wykonaniu.

- Gdy $\texttt{i} = 1$ to $\texttt{res} = 9$ przed i $\texttt{res} = 9\texttt{x} - 3$ po wykonaniu iteracji.
- Gdy $\texttt{i} = 2$ to $\texttt{res} = 9\texttt{x} - 3$ przed i $\texttt{res} = 9\texttt{x}^2 - 3\texttt{x} + 7$ po.
- Gdy $\texttt{i} = 3$ to $\texttt{res} = 9\texttt{x}^2 - 3\texttt{x} + 7$ przed i $\texttt{res} = 9\texttt{x}^3 - 3\texttt{x}^2 + 7\texttt{x} - 4$ po.

Możemy więc sformułować niezmiennik pętli w następujący sposób:

- Gdy $\texttt{i} = \texttt{n}$, to $\texttt{res} = 0$.
- W przeciwnym przypadku $\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A[j]x}^{\texttt{j}-(\texttt{i}+1)}$.

Prawdziwość tego warunku dla $\texttt{i} = \texttt{n}$ jest oczywista. Dla porządku zauważmy, że po pierwszej iteracji pętli mamy $\texttt{i} = \texttt{n} - 1$ i $\texttt{res} = \texttt{A[n]} = \sum_{\texttt{j}=\texttt{n}-1+1}^{\texttt{n}} \texttt{A[j]x}^{\texttt{j}-((\texttt{n}-1)+1)}$. Zajmijmy się utrzymaniem warunku. Jeśli przed rozpoczęciem i–tej iteracji (tzn. znajdując się w linijce 4) mamy:

$$\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A[j]x}^{\texttt{j}-(\texttt{i}+1)},$$

to wykonanie linijki 5 zmodyfikuje nam zawartość zmiennej `res` następująco:

$$\texttt{res} = \texttt{A[i]} + \texttt{x} * (\sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A[j]x}^{\texttt{j}-(\texttt{i}+1)}),$$

a po dalszych transformacjach otrzymamy wartość zmiennej `res` po i–tej iteracji:

$$\texttt{res} = \texttt{A[i]} + \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A[j]x}^{\texttt{j}-\texttt{i}} = \sum_{\texttt{j}=\texttt{i}}^{\texttt{n}} \texttt{A[j]x}^{\texttt{j}-\texttt{i}}.$$

W kolejnej linijce wartość zmiennej `i` jest zmniejszona o jeden, więc powyższa zależność zapisana przy użyciu nowej wartości tej zmiennej przyjmie postać:

$$\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A[j]x}^{\texttt{j}-(\texttt{i}+1)}.$$

Kończymy, przyglądając się zawartości zmiennej `res` po wykonaniu ostatniego kroku pętli. Krok ten zaczyna się z $\texttt{i} = 0$ a kończy z $\texttt{i} = -1$.

$$\texttt{res} = \sum_{\texttt{j}=-1+1}^{\texttt{n}} \texttt{A[j]x}^{\texttt{j}-(-1+1)} = \sum_{\texttt{j}=0}^{\texttt{n}} \texttt{A[j]x}^{\texttt{j}}.$$

A to jest dokładnie to o co chodziło, czyli wartość $f(x)$.

1.4 W stronę analizy złożoności

Schemat Hornera pozwala na obliczenie wartości wielomianu w czasie liniowym. Naiwne podejście do tego problemu daje algorytm działający w czasie kwadratowym. Źródłem złożoności jest tu potęgowanie, realizowane przez szereg mnożeń. Spróbujemy teraz zaprojektować algorytm pozwalający na szybkie potęgowanie.

Zadanie 1.4.1. *Wykorzystaj przedstawienie wykładnika w postaci binarnej do zaimplementowania efektywnego potęgowania i oszacuj złożoność algorytmu. Zapisz pseudokod funkcji* $\texttt{FASTPOW}(\texttt{n}, \texttt{k}) = \texttt{n}^{\texttt{k}}$. *Podpowiedź: zauważ, że* $n^{11_{dec}} = n^{1011_{bin}} = n^{1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0} = n^{2^3} \cdot n^{2^1} \cdot n^{2^0}$. *Zaobserwuj zgodnie z jaką regułą jest tworzony następujący ciąg:* $n^1, (n^2)^2 = n^{2^2}, ((n^2)^2)^2 = n^{2^3}, \ldots$

Przykładowe rozwiązanie (z przeliczaniem decymalne-binarne w locie):

```
FASTPOW(n,k):
1.    if k == 0 return 1

2.    if (k mod 2 == 1) res := n
3.    else res := 1

4.    k = k/2
8.    mult = n*n
5.    while (k > 0)
6.         if (k mod 2 == 1) res *= mult
7.         k = k/2
8.         mult = mult*mult

9.    return res
```

Zadanie 1.4.2. *Wykorzystaj zależność:*

$$n^k = \begin{cases} n(n^{\frac{k-1}{2}})^2 & \textit{dla } k \textit{ nieparzystych,} \\ (n^{\frac{k}{2}})^2 & \textit{dla } k \textit{ parzystych} \end{cases}$$

do zaprojektowania algorytmu efektywnego potęgowania. Użyj rekursji (metoda dziel i rządź). Przygotuj równanie rekurencyjne na pesymistyczną złożoność czasową rozwiązania i oszacuj ją.

Przykładowe rozwiązanie:

```
RPOW(n,k):
1.   if (k == 0) return 1
2.   if (k mod 2 == 1) return n * (rpow(n, (k-1)/2)* rpow(n, (k-1)/2))
3.   else return (rpow(n, k/2)* rpow(n, k/2))
```

Równanie rekurencyjne dla złożoności pesymistycznej:

$$T(k) = \begin{cases} 1 + T(\frac{k-1}{2}) & \text{dla } k \text{ nieparzystych,} \\ 1 + T(\frac{k}{2}) & \text{dla } k \text{ parzystych} \end{cases}$$

Ćwiczenia 2

Wyszukiwanie i sortowanie - początki

2.1 Algorytm wyszukiwania binarnego

Idea algorytmu binarnego wyszukiwania elementu **e** w ciągu uporządkowanym **arr**:

1. Inicjalizacja $l := 0, p := len(\mathbf{arr}) - 1$.
2. Element **e**, o ile jest obecny w ciągu, znajduje się w przedziale indeksowanym $[l, p]$. Jeśli $p < l$, zwróć -1 (**e** nie występuje w **arr**).
3. Obejrzyj wartość mediany: $\mathbf{med} = \mathbf{arr}[\lfloor \frac{l+p}{2} \rfloor]$. Jeśli $\mathbf{med} = \mathbf{e}$, zwróć $\lfloor \frac{l+p}{2} \rfloor$.
4. Jeżeli $\mathbf{med} < \mathbf{e}$, niech $l = \lfloor \frac{l+p}{2} \rfloor + 1$ i wróć do 2.
5. Jeżeli $\mathbf{med} > \mathbf{e}$, niech $p = \lfloor \frac{l+p}{2} \rfloor - 1$ i wróć do 2.

Zadanie 2.1.1. *Wypisz elementy z którym zostanie porównany klucz* **e** *przy wyszukiwaniu go w ciągu* **arr***:*

- $\mathbf{e} = 251, \mathbf{arr} = [0, 1, 3, 7, 11, 210, 113, 251, 325, 412, 1213, 2351]$.
- $\mathbf{e} = 2, \mathbf{arr} = [0, 1, 3, 7, 11, 210, 113, 251, 325, 412, 1213, 2351]$.

Zadanie 2.1.2. *Implementacja z wykładu wyszukiwania binarnego nie korzysta z rekurencji. Zaprojektuj rekurencyjną wersję wyszukiwania binarnego. Jakie są zalety a jakie wady rekurencji?*

Przykładowe rozwiązanie:

```
Rec-BinSearch(s,l,p,e):
    if l > p return -1
    m = floor((l+p)/2)
    if s[m] = e return e
    if s[m] < e return Rec-BinSearch(s, m+1, p)
    if s[m] > e return Rec-BinSearch(s, l, m-1)
```

Zadanie 2.1.3. *UVA: 10611 - The Playboy Chimp. Podpowiedź: zastosuj algorytm wyszukiwania binarnego zwracający ostatnio sprawdzany indeks zamiast -1 i obejrzyj elementy sąsiadujące ze zwróconym indeksem.*

Zadanie 2.1.4. *Metoda bisekcji służy do znajdowania pierwiastków równania* $f(x) = 0$. *Zakładamy, że* f *jest funkcją rzeczywistą, ciągłą, określoną na przedziale* $[a, b]$ *i taką, że* $f(a) \cdot f(b) < 0$ *(dlaczego to wystarczy, by istniał pierwiastek w w/w przedziale?). Zaprojektować algorytm, który dla zadanego* $\epsilon > 0$ *znajduje przedział* $[c, d] \subseteq [a, b]$ *zawierający pierwiastek* f *i taki, że* $(d - c) \leq \epsilon$.

2.2 Algorytm turniejowy

Cel algorytmu turniejowego: znaleźć **drugi najmniejszy** element **e** w ciągu nieuporządkowanym **arr** zawierającym elementy bez powtórzeń. Idea działania:

1. Zbuduj najniższe piętro drzewa binarnego, złożone z elementów **arr**. Będziemy budowali drzewko od tej podstawy, ku zwieńczeniu.
2. Budujemy nowe piętro. Iteruj (np. od lewej) po **parach** sąsiednich węzłów poprzedniego piętra (żadne dwie pary nie mają wspólnych elementów). Z każdej takiej pary $(node(\mathbf{arr}[i]), node(\mathbf{arr}[i+1]))$ wybierz mniejszy element i umieść go w nowym węźle. Lewa i prawa gałąź nowego węzła powinny prowadzić do, odpowiednio, węzłów zawierających $\mathbf{arr}[i]$ i $\mathbf{arr}[i+1]$. Jeśli poprzednie piętro miało nieparzystą liczbę elementów, przepisz element, który nie wziął udziału w porównaniach do nowego piętra.
3. Jeżeli najnowsze piętro ma więcej niż jeden element, wróć do punktu 2.
4. Jeśli ma tylko jeden element, jest to element najmniejszy **mine**. Zejdź od korzenia w dół drzewa po węzłach zawierających **mine**, oglądając elementy porównywane z **mine**. Wybieramy najmniejszy z tych elementów.

Zadanie 2.2.1. *Narysuj drzewko porównywań w algorytmie turniejowym dla ciągu* $\mathbf{arr} = [7, 4, 9, 1, 8, 15]$.

Zadanie 2.2.2. *Dlaczego element znaleziony w czasie schodzenia w dół drzewka jest drugim najmniejszym w ciągu?*

2.3 Statystyki pozycyjne

Niech **arr** będzie ciągiem (dla uproszczenia - różnych) nieuporządkowanych liczb. i–tą statystyką pozycyjną nazwiemy i–ty najmniejszy element **arr**, gdzie $0 \leq i < |\mathbf{arr}|$ (tradycyjnie, liczymy od 0). Na wykładzie przedstawiono chyba już zarys algorytmu wykorzystującego podziały Hoare'a, obliczającego statystyki pozycyjne w średnim czasie liniowym (przy założeniu zrandomizowanego wyboru elementu dzielącego w podziale Hoare'a). Załóżmy, że mamy procedurę `Hoare-Partition(arr, p, r)`, która:

1. zwraca indeks q taki, że $p \leq q \leq r$, oraz
2. permutuje tablicę $\mathbf{arr}[p \ldots r]$ w taki sposób, że elementy $\mathbf{arr}[p \ldots (q-1)]$ są mniejsze od elementów $\mathbf{arr}[(q+1) \ldots r]$, zaś element $\mathbf{arr}[q]$ jest na właściwym miejscu (tzn. tam, gdzie znalazłby się po posortowaniu **arr**).

Ilustracja: jeśli $\mathbf{arr} = [5, 1, 4, 9, 3, 8]$, to przykładowe wykonanie procedury `Hoare-Partition(arr, 0, 6)` może zwrócić $q = 3$ i zmodyfikować tablicę $\mathbf{arr} = [3, 1, 4, 5, 9, 8]$. Pomimo, iż nie jest posortowana, to wartość 5 znalazła się na swoim miejscu.

Procedura `Select(arr, p, r, i)` wykorzystuje podziały Hoare'a w celu rekurencyjnego wyszukania i–tej statystyki tablicy **arr** [p. . . r].

```
Select(arr, p, r, i):
    if p = r then return arr[p]

    q = Hoare-Partition(arr, p, r)
    k = q - p + 1
    if i = k then return arr[q]
    if i < k then return Select(arr, p, q-1, i)
    else return Select(arr, q+1, r, i-k)
```

Szczegóły procedury `Hoare-Partition(arr, p, r)` zostaną przedstawione przy okazji algorytmu QuickSort.

Zadanie 2.3.1. *Znaleźć najgorszy przypadek dla procedury* `Select` *i określić jego złożoność.*

2.4 Sortowanie - początki

Algorytm sortowania przez wstawianie (InsertionSort) polega na wstawianiu w już posortowaną początkową część tablicy kolejnego elementu **e**. Polega to na przesuwaniu elementów w prawo, dopóty, dopóki nie znajdziemy właściwego miejsca do wstawienia **e**. Algorytm ma dość elegancką i prostą implementację (patrz wykład), ale jego średnia złożoność jest kwadratowa.

Zadanie 2.4.1. *Zasymuluj działanie InsertionSort na tablicy* $\mathbf{arr} = [4, 1, 5, 1, 6, 9, 10]$. *Określ pesymistyczny (największa liczba porównań) i optymistyczny (najmniejsza) przypadek dla tego algorytmu.*

Zadanie 2.4.2. *InsertionSort ma swoją nieco bardziej praktyczną wersję - sortowanie Shella (ShellSort). Zapoznaj się z opisem tego algorytmu i przetestuj kilka wybranych sekwencji wyboru kroku.*

Algorytm sortowania jest *stabilny*, jeśli posortowana tablica zachowuje pierwotną kolejność elementów o tych samych kluczach. Rozważmy tablicę $\mathbf{arr} = [3, 1, 4, 3, 5]$. Zakładamy tutaj, że elementy tablicy posiadają jakąś strukturę wewnętrzną, odróżniające czerwoną trójkę od niebieskiej. Stabilny algorytm sortowania zastosowany do **arr** zwróci $[1, 3, 3, 4, 5]$. Niestabilny może zwrócić $[1, 3, 3, 4, 5]$. Stabilność jest wartościową cechą - rozważmy sytuację, gdy sortujemy książkę telefoniczną, najpierw po imionach, potem po nazwiskach.

Zadanie 2.4.3. *Rozważmy dwie wersje algorytmu InsertionSort (tym razem zapisane w pseudoJavie):*

```
insertionSortA(arr, len){

    for(next = 1; next < len; next++){

    curr = next;
    temp = arr[next];

    while((curr > 0) && (temp < arr[curr - 1])){
        arr[curr] = arr[curr - 1];
        curr--;
    }

    arr[curr] = temp;
  }
}

insertionSortB(arr, len){

    for(next = 1; next < len; next++){

    curr = next;
    temp = arr[next];

    while((curr > 0) && (temp <= arr[curr - 1])){
        arr[curr] = arr[curr - 1];
        curr--;
    }

    arr[curr] = temp;
  }
}
```

Który z nich jest stabilny i dlaczego?

Algorytm sortowania przez scalanie (MergeSort) polega na podziale wejściowej tablicy $\mathbf{arr}[p \dots r]$ na dwie części (mniej więcej równej wielkości) i rekurencyjnym wywołaniu na każdej z nich. Przy dobrej implementacji jest to algorytm stabilny, o złożoności pesymistycznej rzędu $nlogn$. Idea działania:

1. Znajdź indeks środkowy $\mathbf{med} = \lfloor \frac{p+r}{2} \rfloor$. Wywołaj rekurencyjnie procedurę dla podtablic $\mathbf{arr}[p \dots \mathbf{med}]$ i $\mathbf{arr}[\mathbf{med} \dots r]$.
2. *Scal* posortowane podtablice $L = \mathbf{arr}[p \dots \mathbf{med}]$ i $R = \mathbf{arr}[\mathbf{med} \dots r]$:
   (a) Zadeklaruj nową tablicę pomocniczą **arrh** o wielkości $r - p$.
   (b) Ustaw wskaźniki i, j na początkach tablic L i R.
   (c) Wybierz mniejszy ze wskazywanych elementów (lub lewy - w przypadku równości, by utrzymać stabilność) i wstaw go do tablicy **arrh**. Zwiększ wykorzystany wskaźnik.
   (d) Jeśli któryś ze wskaźników dotarł do krańca tablicy, wstaw pozostałe elementy drugiej tablicy do **arrh**.
   (e) Jeśli zostało coś do przepisania, wróć do (c).
   (f) Przepisz **arrh** do $\mathbf{arr}[p \dots r]$.

Zadanie 2.4.4. *Zasymuluj na tablicy* $[10, 5, 1, 6, 1, 7, 8, 12, 3, 5, 13]$ *działanie algorytmu MergeSort: narysuj drzewo wywołań i drzewo złączania.*

Zadanie 2.4.5. *Dodatkowa złożoność pamięciowa jest główną wadą MergeSort. Można jej uniknąć poprzez wykorzystanie list. Pojawia się tu kilka niuansów - jednym z nich jest problem znalezienia środka listy jednokierunkowej. Zaprojektować algorytm rozwiązujący ten problem wykorzystujący co najwyżej* dwa *wskaźniki i dokonujący tylko* jednego przebiegu.

Ćwiczenia 3

Wyszukiwanie i sortowanie - QuickSort i inne

3.1 Hoare-Partition

Przyjrzyjmy się teraz dwóm wersjom procedury Hoare-Partition. Pierwsza, zgodna z pseudokodem z wykładu, pracuje przesuwając wskaźniki lewy i prawy od dwóch odp. skrajnych końców tablicy ku sobie. Utrzymywany jest warunek, iż na lewo od lewego wskaźnika znajdują się elementy niewiększe od dzielącego a na prawo od prawego niemniejsze.

```
Hoare-Partition(A,p,r):

x = A[p]
i = p + 1
j = r

do
    while (i < r and A[i] <= x) i++
    while (j > i and A[i] >= x) j--
    if i < j then swap(A,i,j)
while (i < j)

if A[i] > x then
    swap(A,p,i-1)
    return i - 1
else
    swap(p,i)
    return i
```

Działanie drugiej z procedur jest odmienne. Wskaźniki lewy i prawy wędrują od początku tablicy do jej końca. Utrzymywany jest pomiędzy nimi "mur" wartości większych od dzielącej. Gdy prawy wskaźnik napotyka element niewiększy od wartości dzielącej, "podaje" go lewemu, zamieniając go na pobrany od lewego element większy od wartości dzielącej. Następnie lewy wskaźnik robi jeden krok, zaś prawy rusza w dalszą drogę, aż do dotarcia na koniec tablicy lub ponownego trafienia na element niewiększy od dzielącego.

```
Hoare-Partition-Modern(A,p,r):

x = A[p]
i = p - 1

for j = p to r - 1
    if A[j] <= x then
        i++
        swap(A,i,j)
```

```
swap(A,i,p)

return i
```

Zadanie 3.1.1. *Zasymuluj wykonanie obu algorytmów na tablicy* **arr** $= [7, 2, 4, 6, 1, 9, 8, 11, 5]$.

3.2 Quick Sort

Znając procedurę Hoare-Partition (w dowolnej wersji), możemy napisać kod algorytmu szybkiego sortowania.

```
Quicksort(a, l, r):

if(l >= r) return
k = partition(a, l, r)
quicksort(a, l, k - 1);
quicksort(a, k + 1, r);
```

Jak wiadomo, algorytm ten jest szybki w praktyce (jego przeciętna złożoność czasowa jest liniowo-logarytmiczna), pomimo pesymistycznej złożoności kwadratowej. Jak się okazuje $\Theta(n \log n)$ jest dolnym ograniczeniem złożoności czasowej dla problemu sortowania przez porównywanie.

3.3 Sortowanie w czasie liniowym

Teoria: znakomicie wytłumaczone na wykładzie!

Zadanie 3.3.1. *Przedstaw kolejne etapy sortowania tablicy* **arr** $=$ *[4,5,1,3,2,5,1,5] metodą* `CountingSort` *(w wersji stabilnej). Ściślej:*

- *Przedstaw wartości tablicy* `counts` *(1) po zliczaniu elementów, (2) po podsumowaniu przyrostowym, a następnie...*
- *przedstaw wartości tablic* `counts` *i* `results` *podczas wypełniania tej ostatniej elementami.*

Zadanie 3.3.2. *Pytanie: dlaczego w sortowaniu przez zliczanie wędrujemy od prawej strony tablicy wejściowej w lewo? Czy algorytm byłby poprawny, gdybyśmy wędrowali od lewej? A stabilny?*

Ćwiczenia 4

Struktury danych - początki

4.1 Tablice i tablice dynamiczne

Tablice typowo implementowane są jako ciągły obszar pamięci o swobodnym dostępie przy użyciu indeksu. Tablice mają zadaną, niezmienną wielkość i z założenia nie posiadają bezużytecznych (pustych) elementów. Można więc powiedzieć, że zwykłe tablice nie marnują miejsca.

Tablice dynamiczne są wersją zwykłych tablic zdolną do pomieszczenia zmiennej liczby elementów. Indeksowanie w nich przebiega w taki sam sposób, jak w zwykłych tablicach. Dodatkowe względem tablic dynamicznych operacje to *pushBack* (wstawienie elementu na koniec) i *popBack* (usunięcie elementu z końca tablicy). Ich efektem ubocznym może być zmiana wielkości tablicy. Można więc powiedzieć, że tablice dynamiczne mają *rozmiar logiczny* n (liczba elementów) i *rozmiar fizyczny* w (maksymalna liczba elementów). Różnica pomiędzy tymi rozmiarami to bezużyteczna przestrzeń (co może być mylącym określeniem, bo jest to w rzeczywistości bufor bezpieczeństwa).

Przykładowa strategia działania tablic dynamicznych może być taka:

- Jeśli po wykonaniu operacji *pushBack(e)* tablica jest pełna, alokujemy w pamięci miejsce na $A \times n$ elementów (typowo $A = 2$ - zwiększamy rozmiar dwukrotnie) i przepisujemy tablicę na nowe miejsce.
- Jeśli po wykonaniu operacji *popBack(e)* tablica ma wielkość logiczną równą $B\%$ wielkości fizycznej (typowo 25% - czyli $\frac{1}{4}$ tablicy zawiera elementy), również alokujemy w pamięci miejsce na $C\% \times w$ (typowo $C = 50\%$) elementów i przepisujemy tablicę na nowe miejsce.

Strategia w której powiększamy dwukrotnie tablicę w momencie jej wypełnienia i zmniejszamy o połowę w momencie gdy wypełniona jest $\frac{1}{4}$ miejsc daje *stały koszt zamortyzowany* operacji *pushBack* i *popBack*. Intuicja jest taka, że koszt wykonania m operacji tego typu będzie miał złożoność czasową rzędu $\Theta(m)$, więc pojedyńcza operacja ma złożoność rzędu $\Theta(1)$.

Zadanie 4.1.1. *Zasymuluj następującą kolejkę operacji, zaczynając od tablicy o* $w = 1$*:* $[pushBack(1), pushBack(2), popBack(), pushBack(3), pushBack(4), popBack()]$.

Zadanie 4.1.2. *Przyjmijmy* $A = 2$ *i* $B = C = 50\%$*. Innymi słowy, powiększamy tablicę dwukrotnie, gdy się wypełni i zmniejszamy o połowę gdy połowa miejsc jest wypełniona. To jest niedobra strategia i można zbudować sekwencję* m *operacji pushBack i popBack, której czas wykonania jest rzędu* $\Theta(m^2)$*. Zaproponuj taką sekwencję. Podpowiedź: rozważ* $m = 2^k$ *operacji i spróbuj coś "popsuć" po wypełnieniu połowy tablicy.*

4.2 Listy

Listy są prostymi strukturami danych złożonymi z połączonych węzłów. Każdy z węzłów przechowuje element i wskaźnik do następnego węzła. Ten najprostszy typ

listy nazywa się listą jednokierunkową (SList w poniższej implementacji). Umieszczenie wskaźnika do poprzedniego węzła daje listy dwukierunkowe (DList poniżej). Zwykle przechowujemy początek listy (głowę). Wskaźnik next ostatniego węzła oraz wskaźnik prev pierwszego (w przypadku list dwukierunkowych) wskazują na NULL, chyba, że mamy do czynienia z listami cyklicznymi. W tym ostatnim przypadku wskaźnik next ostatniego węzła pokazuje pierwszy, zaś wskaźnik prev pierwszego wskazuje ostatni węzeł.

```
SList {
      Type element;
      SList* next;
      }

DList {
      Type element;
      SList* next;
      SList* prev;
      }
```

Dla list, koszt czasowy dostępu do n-tego elementu jest rzędu $\Theta(n)$. Natomiast operacje *pushFront* (wstawienie elementu na początek, tj. przed głowę listy) i *popFront* (usunięcie głowy) działają w czasie $\Theta(1)$. W przypadku list jednokierunkowych operacje *pushBack* i *popBack* (wstawienie/usunięcie elementu na końcu listy) działają w czasie liniowym. Dwukierunkowe listy cykliczne pozwalają na wykonanie *pushBack* i *popBack* w czasie $\Theta(1)$.

Zadanie 4.2.1. *Zaproponuj implementację (pseudokod) czterech wymienionych wyżej operacji dla list jednokierunkowych i dwukierunkowych list cyklicznych.*

Zadanie 4.2.2. *Rozszerz implementację dla list jednokierunkowych o wskaźnik do ostatniego elementu i licznik liczby elementów.*

Zadanie 4.2.3. *Zaprojektuj algorytm, działający w czasie liniowym, usuwający z listy jednokierunkowej wszystkie pojawiające się pod rząd duplikaty. Np. lista* $1 \to 2 \to 2 \to 3 \to 4 \to 4 \to 4 \to 5$ *zostanie przekształcona na* $1 \to 2 \to 3 \to 4 \to 5$.

Zadanie 4.2.4. *Zaprojektuj algorytm, który odwraca kierunek listy. Dozwolone jest tylko jedno przejście od początku do końca i* $\Theta(1)$ *pamięci dodatkowej. Np. lista* $1 \to 2 \to 3 \to 4 \to 5$ *zostanie przebudowana na* $5 \to 4 \to 3 \to 2 \to 1$.

4.3 Kolejki

4.3.1 Stosy

Stos, zwany też kolejką LIFO (last-in, first-out) jest bardzo prostą strukturą danych o następującym interfejsie:

```
Stack {
      push(element); //wstaw element na wierzcholek stosu
      pop(); //zdejmij element z wierzcholka stosu i zwroc go
      top(); //zwroc element z wierzcholka stosu bez usuwania, opcjonalne
      empty(); //true wtw gdy stos jest pusty, opcjonalne
};
```

Stos spełnia zależność $e \equiv (push(e); pop())$. Intuicja jest taka, jak ze stosem talerzy: nowy odkładamy na górę, zdejmujemy również tylko ten z góry.

Zadanie 4.3.1. *Zaproponuj implementację stosu przy użyciu list z dowiązaniami lub tablic.*

Zadanie 4.3.2. *Masz do dyspozycji* tylko i wyłącznie *dwa stosy i ew. jedną zmienną pomocniczą typu int. Na jednym znajduje się łańcuch znakowy* **s**, *a drugi jest pusty. Sprawdź, czy* **s** *jest palindromem.*

Zadanie 4.3.3 (Domowe)**.** *Zapoznaj się z definicją Odwrotnej Notacji Polskiej (RPN). Napisz program, przyjmujący na wejściu łańcuch w RPN obsługujący podstawowe operacje arytmetyczne (40%) oraz unarne operacje takie jak silnia, funkcje trygonometryczne, logarytm naturalny (60%). Podpowiedź: wystarczy użyć jednego stosu i przeglądać wejściowy łańcuch od lewej.*

Zadanie 4.3.4 (Dość trudne)**.** *Zaproponuj rekurencyjne rozwiązanie problemu wież Hanoi z dyskami umieszczanymi na stosach.*

4.3.2 Kolejki FIFO

Kolejki FIFO (first-in, first-out) są bardzo prostymi strukturami danych o następującym interfejsie:

```
Queue {
      inject(element); //wstaw element na koniec kolejki
      out(); //zdejmij element z początku kolejki
      front(); //zwroc element z początku kolejki bez usuwania, opcjonalne
      empty(); //true wtw gdy kolejka jest pusta, opcjonalne
};
```

Kolejka FIFO spełnia zależność $(e \equiv (inject(e); out())) => empty()$. Intuicja jest taka, jak z kolejką przy kasie: obsługiwana jest osoba na początku, kolejka wydłuża się na końcu.

Zadanie 4.3.5. *Masz do dyspozycji tylko i wyłącznie dwa stosy. Zaimplementuj przy ich pomocy trzy główne operacje kolejki FIFO. Podaj złożoność czasową algorytmu.*

Zadanie 4.3.6. *Porównaj, krok po kroku, efekty wykonania następujących operacji przy wykorzystaniu stosu i kolejki FIFO:* wstaw(1), zdejmij(), wstaw(2), wstaw(3), wstaw(4), zdejmij(), wstaw(5), zdejmij(), zdejmij().

Zadanie 4.3.7. *Zaproponuj implementacje procedury* search(e) *wyszukujące element* **e** *w (1) stosie (2) kolejce. Rozważ sytuacje, gdy jako struktury pomocniczej używasz tylko stosu lub tylko kolejki (wszystkie cztery możliwości).*

Zadanie 4.3.8. *Zaproponuj implementację tablicową kolejek (taka kolejka ma pewną wielkość maksymalną). Czy łatwo byłoby zmodyfikować ją tak, by uzyskać wersję dwukierunkową kolejki (tzn. uzupełnić o możliwość dodawania elementów na początku kolejki i zdejmowania elementów z końca). Wykonaj to samo zadanie dla implementacji listowej.*

Ćwiczenia 5

Kopce

5.1 Przypomnienie - drzewa binarne

- Definicja drzewa binarnego.
- **Pytanie:** ile węzłow znajduje się na i-tym poziomie pełnego drzewa binarnego?
- **Pytanie:** ile węzłów znajduje się w pełnym drzewie o wysokości h?
- **Pytanie:** jakiej minimalnej wysokości musi być drzewo binarne, by pomieścić k elementów? (Podać dokładną wartość i asymptotyczny rząd wielkości.)

5.2 Kopce - podstawy

- Definicja i warunek kopca (w wersji min). Przypomnienie: prawie pełne drzewo binarne, którego ścieżki tworzą niemalejące ciągi elementów.
- **Pytanie:** ile jest kopców o elementach 1,2,3,4,5?
- **Pytanie:** gdzie można się spodziewać drugiego najmniejszego elementu w kopcu? Gdzie może znaleźć się trzeci? A do jakiej głębokości można spodziewać się n-tego elementu?
- Implementacja tablicowa kopca, przypomnienie:
  - elementy złożone są do tablicy poziomami, idąc od korzenia i od lewej w prawo;
  - indeksujemy elementy tablicy od 1 (ew. ignorując indeks 0);
  - $parent(i) = \lfloor i/2 \rfloor$, $left(i) = 2 \cdot i$, $right(i) = 2 \cdot i + 1$.
- **Zadanie:** Zapisz w tablicy zadany w postaci graficznej kopiec.

5.3 Downheap i zastosowania

- Przypomnienie działania downheap: odszukujemy indeks min wskazujący na najmniejszą wartość z trójki $i, left(i), right(i)$ i dokonujemy ew. zamiany wartości pod min i i, po czym wywołujemy downheap rekurencyjnie na min. Zilustrować to na przykładzie.
- **Pytanie:** dlaczego dokonujemy zamiany z mniejszym z dwójki dzieci?
- Przypomnienie operacji construct: wywołujemy downheap na indeksach od $n/2$ do 1; trzy słowa o tym, dlaczego to ma złożoność liniową.
- **Zadanie:** Wykonaj operację construct na przykładowej tablicy elementów (np. [3,5,7,4,3,6,3,2,4]). Zapisz pośrednie wartości tablicy.
- **Pytanie:** Czy posortowana rosnąco tablica jest kopcem?

- Przypomnienie i ilustracja operacji delmin: usuwamy korzeń i wstawiamy w jego miejsce element kończący najniższy poziom a następnie wykonujemy na korzeniu downheap.
- **Zadanie:** Wykonaj na kopcu z poprzedniego zadania operację delmin aż do pełnego jego opróżnienia.
- Uwaga: w/w schemat: construct + delmin until empty jest realizacją sortowania przez kopcowanie (HeapSort). Jaka jest jego złożoność pesymistyczna?
- **Zadanie:** Jak usunąć dowolny (zadany poprzez indeks) element z kopca (operacja delete)? Jaka jest złożoność tej operacji? Zaprojektuj algorytm i uzasadnij jego poprawność. Podpowiedź: jak w przypadku usuwania korzenia, ale korzeniem będzie usunięty element.
- **Zadanie:** Zaprojektuj algorytm IncreaseKey(i), zwiększający wartość w zadanym węźle. Podpowiedź: po prostu użyj downheap.

5.4 Upheap i zastosowania

- Przypomnienie działania upheap: porównujemy wartość węzła z rodzicem; jeśli zaburzony jest porządek, to zamieniamy wartości miejscami i wywołujemy upheap rekurencyjnie na rodzicu. Zilustrować na przykładzie.
- **Pytanie:** Często unikamy rekurencji ze względów pamięciowych (rosnący stos wywołań). Czy w przypadku kopca jest to duży problem? (Java bez problemu zniesie 5000 wywołań rekurencyjnych; Python powinien sobie poradzić do przynajmniej 900).
- Przypomnienie operacji insert: wstawiamy nowy element na pierwsze wolne miejsce na najniższym poziomie (pamiętamy o warunku prawie pełnego drzewa binarnego) i wywołujemy na nim procedurę upheap.a
- **Zadanie:** Wstaw do początkowo pustego kopca elementy 3,5,7,4,3,6,3,2,4. Porównaj wynik z kopcem poprzednio zbudowanym przy użyciu operacji construct.

5.5 Zadania różne

- Otrzymujesz min-kopiec H zawierający liczby dodatnie. Korzystając tylko z operacji kopców wypisz elementy H w kolejności malejącej. Możesz modyfikować H. Podpowiedź: wystarczy zdejmować najmniejszy element z i wstawiać z powrotem jego odwrotność.
- Wylosuj w sposób całkowicie jednorodny (tzn. każdy z węzłów powinien mieć taką samą szansę) węzeł z pełnego drzewa binarnego o wysokości h. Drzewo zadane jest w zapisie dowiązaniowym. Podpowiedź: wystarczy wylosować liczbę z przedziału $[0, 2^{h+1})$ i potraktować jej binarny zapis jako opis ścieżki do węzła. (dopracować)
- (Trochę trudniejsze) Wypisz wszystkie elementy kopca bez modyfikacji kopca i wykorzystując O(1) pamięci dodatkowej. Podpowiedź: jedno z rozwiązań może opierać się na sukcesywnym wypisywaniu 1-ego, 2-ego,... najmniejszego elementu w kopcu (np. przy pomocy funkcji rekurencyjnej). W takim przypadku należy zoptymalizować głębokość wyszukiwania (patrz pytania na początku zajęć).
- ...

Ćwiczenia 6

Słowniki

Przypomnienie: **słowniki** pozwalają na przechowywanie par (k, v) złożonych z klucza k i wartości v. W ogólności, z danym kluczem może być powiązanych kilka wartości. Obsługiwane operacje to:

- *insert*(k, v): wstawienie elementu v pod kluczem k;
- *delete*(k, v): usunięcie elementu v związanego z kluczem k, jeśli taki zestaw istnieje. (W innej wersji ta operacja przyjmuje wskaźnik do obiektu do usunięcia jako argument.)
- *search*(k): zwraca zbiór elementów o kluczu k (zwykle zakładamy, że klucze są unikalne).

6.1 Rozgrzewka: adresowanie bezpośrednie

W adresowaniu bezpośrednim z każdym kluczem ze zbioru $0, \ldots, m$ związana jest dokładnie jedena wartość. Wartości przechowywane są w tablicy $tab[0 \ldots m]$. Wszystkie operacje działają w czasie stałym, ale struktura taka jest nieefektywna pamięciowo, bo zwykle większość kluczy nie jest używana (potrzebna pamięć to $|\mathcal{U}|$).

Zadanie 6.1.1. *W tablicy $tab[0 \ldots m]$ znajduje się m różnych elementów. Napisz algorytm wypisujący wszystkie ich podzbiory. Zilustruj jego działanie, pokazując dla tablicy $tab = [a, b, c, d]$, jakie zbiory będą po kolei wypisane. Podpowiedź: skorzystaj z zapisu binarnego liczb od 0 do $2^{m+1} - 1$.*

6.2 Funkcje mieszające

Idea: naszą intencją jest wstawianie do tablicy $tab[0, \ldots, m]$ elementów identyfikowanych kluczami z (zazwyczaj dużego) uniwersum $\mathcal{U}$. Aby uniknąć narzutu pamięciowego adresowania bezpośredniego korzystamy z funkcji $h \colon \mathcal{U} \to [0, \ldots, m]$ przyporządkowującej kluczom pozycje w tablicy. Z racji wielkości uniwersum kolizje są nieuniknione - tzn. należy oczekiwać, że może zdarzyć się, iż dwa różne klucze zostaną przyporządkowane tej samej pozycji. Radzimy sobie z tym na kilka sposobów.

6.2.1 Metoda łańcuchowa

W metodzie łańcuchowej tablica *tab* zawiera dynamiczne struktury danych (np. kolejki), mogące pomieścić kolidujące elementy. Czasem nazywamy te struktury wiaderkami.

Zadanie 6.2.1. *Rozważmy f. mieszającą $h_5(x) = x \mod 7$ dla tablicy $tab[0, \ldots, 6]$. Załóżmy, że dla każdego i typ $tab[i]$ to (a) stos, (b) lista dwukierunkowa. Zasymuluj wstawienie do tablicy elementów $(1, a), (8, b), (3, c), (6, d), (4, e), (10, f), (11, g)$. Następnie zasymuluj wyszukanie elementów $(8, b)$ i $(13, f)$.*

Dobra praktyka - wartość m w funkcji $h_m(x) = x \mod m$ powinna być liczbą pierwszą (odległą od najbliższej potęgi dwójki). Powód - niech $m = 5 \cdot 7$ i wstawiajmy strumień elementów o kluczach $1 \cdot 5, 2 \cdot 5, 3 \cdot 5, \ldots$ do tablicy. Nawet po zredukowaniu modulo 35 docelowe pozycje będą dzieliły się przez 5, więc tylko $\frac{1}{5}$ tablicy będzie wykorzystana.

Przy założeniu, że dane wejściowe rozmieszczane są przez funkcję mieszającą w taki sposób, że n–ty element może pojawić się z takim samym prawdopodobieństwem na dowolnej z pozycji, średnia długość listy w tablicy $tab[0, \ldots, m]$ wynosi $\alpha = \frac{n}{m}$. Zatem wszystkie operacje słownika wymagają $\Theta(1)$ czasu na obliczenie $h_m(k)$ plus $\Theta(\frac{n}{m})$ na przeszukanie odpowiedniej listy.

Zadanie 6.2.2. *Dobrą praktyką przy założeniu, że wstawiane dane są losowe jest utrzymywanie współczynnika obciążenia* $\alpha = \frac{\textit{liczba wstawionych elementów}}{\textit{pojemność tablicy}}$ *poniżej* 0.75. *Zaprojektuj zarys implementacji metody insert, który będzie spełniał w/w wymagania. Podpowiedź: rehashing.*

6.2.2 Metoda mieszania wielokrotnego

W tej metodzie tablica $tab[0, \ldots, m]$ bezpośrednio przechowuje pary (k, v). Indeks do wstawienia elementu (k, v) obliczany jest przy użyciu *funkcji próbkującej* $h(k, i)$. Idea jest taka, że wstawiając element (k, v) do tablicy badamy po kolei jej zawartość pod indeksami $h(k, 0), h(k, 1), h(k, 2), \ldots$ i wstawiamy (k, v) w pierwsze ze znalezionych wolnych miejsc:

```
OPENHASH-INSERT(tab, (k,v)):

    for i = 0 to len(tab):
       index = h(k,i)
       if (tab[index] is NIL) then:
          tab[index] = (k,v)
          return index

    return -1 //table full
```

Zadanie 6.2.3. *Jest wielu kandydatów na funkcje* $h(k, \cdot)$ *(patrz książka Cormena). Rozważ* $h(k, i) = (k + 3 \cdot i + 5 \cdot i^2) \mod 7$ *w powyższej funkcji i wstaw do tablicy* $tab[0, \ldots 7]$ *elementy o kluczach* $5, 6, 11, 9, 2, 7$.

Nie przedstawiamy kodu operacji usuwania, ponieważ jest to jeden ze słabszych punktów mieszania wielokrotnego. Pytanie: dlaczego?

6.3 Binarne drzewa wyszukiwań

Binarne drzewa wyszukiwań (BST) można postrzegać jako połączenie funkcjonalności słowników i kolejek priorytetowych. Są to binarne drzewa, których węzły zawierają klucze oraz powiązane z nimi wartości. (Zazwyczaj w praktyce pomijamy dla czytelności wartość v i skupiamy się na operacjach związanych z kluczem.) Klucze rozmieszczone są w węzłach w szczególny sposób, tak by spełnić warunek taki, że dla każdego węzła n:

- węzły w poddrzewie o korzeniu $n.left$ zawierają klucze nie większe od $n.key$;
- węzły w poddrzewie o korzeniu $n.right$ zawierają klucze nie mniejsze od $n.key$.

BST oferują implementacje następujących operacji:

- *insert*(n), *delete*(n), *search*(k): normalne operacje słowników. Dla uproszczenia zakładamy, że n to struktura danych zawierająca pola $n.key$, $n.val$.
- *min*(), *max*(): zwrócenie najmniejszej i największej wartości w drzewie;

- *succ*(n): zwraca najmniejszy klucz spośród obecnych w drzewie i większych od $n.key$;
- *pred*(n): zwraca największy klucz spośród obecnych w drzewie i mniejszych od $n.key$.

Zadanie 6.3.1. *Załóżmy, że mamy drzewo BST. W jakiej kolejności wypisze jego węzły następujący program?*

```
INORDER-BST-WALK(n):

   if x != NIL then:
      INORDER-BST-WALK(n.left)
      print(n.key)
      INORDER-BST-WALK(n.right)
```

Zmodyfikuj powyższy program tak, aby przepisał wartości z drzewa do tablicy w sposób posortowany niemalejąco. Założenie: drzewo nie może zostać zmodyfikowane. Podpowiedź: można użyć globalnej tablicy i bieżącego wskaźnika pozycji.

Przyjrzyjmy się zarysowi implementacji każdej z operacji BST:

- *min*(): przesuwaj wskaźnik c po drzewie, startując od korzenia i przestawiając go sukcesywnie na jego lewego syna - dopóki ten syn istnieje.
- *max*(): przesuwaj wskaźnik c po drzewie, startując od korzenia i przestawiając go sukcesywnie na jego prawego syna - dopóki ten syn istnieje.
- *succ*(n): jeśli $n.right$ istnieje, zwróć $min(n.right)$. Jeśli nie - sukcesywnie przesuwamy wskaźnik c po drzewie, startując od n i ustawiając go na jego własnego ojca. Powtarzamy przesuwanie, gdy poprzedni ruch był w lewo, kończymy gdy doszliśmy do NIL (brak sukcesora) lub wykonaliśmy ruch w prawo (zwracamy $c.key$). Pytanie: dlaczego to działa? Pseudokod dla jasności:

  ```
  BST-SUCCESSOR(n):

      if x.right != NIL then:
         return min(x.right)

      y = y.parent
      while y != NIL and x == y.right:
         x := y
         y := y.parent

      return y
  ```

- *search*(k): jak w przypadku wyszukiwania binarnego - jeśli bieżący węzeł n (zaczynamy od korzenia) zawiera klucz k, to zwracamy go. Jeśli $n.key < k$, to kontynuujemy poszukiwanie w poddrzewie $n.right$. W przeciwnym przypadku poszukujemy w poddrzewie $n.left$. Jeśli w/w wybór poddrzewa jest niemożliwy, to klucz nie jest obecny w drzewie.
- *insert*(n): wykonujemy wyszukiwanie binarne (ignorując sytuację, gdy badany wierzchołek ma wartość równą $n.key$), dopóki nie znajdziemy węzła NIL. Wstawiamy n w miejsce tego węzła.

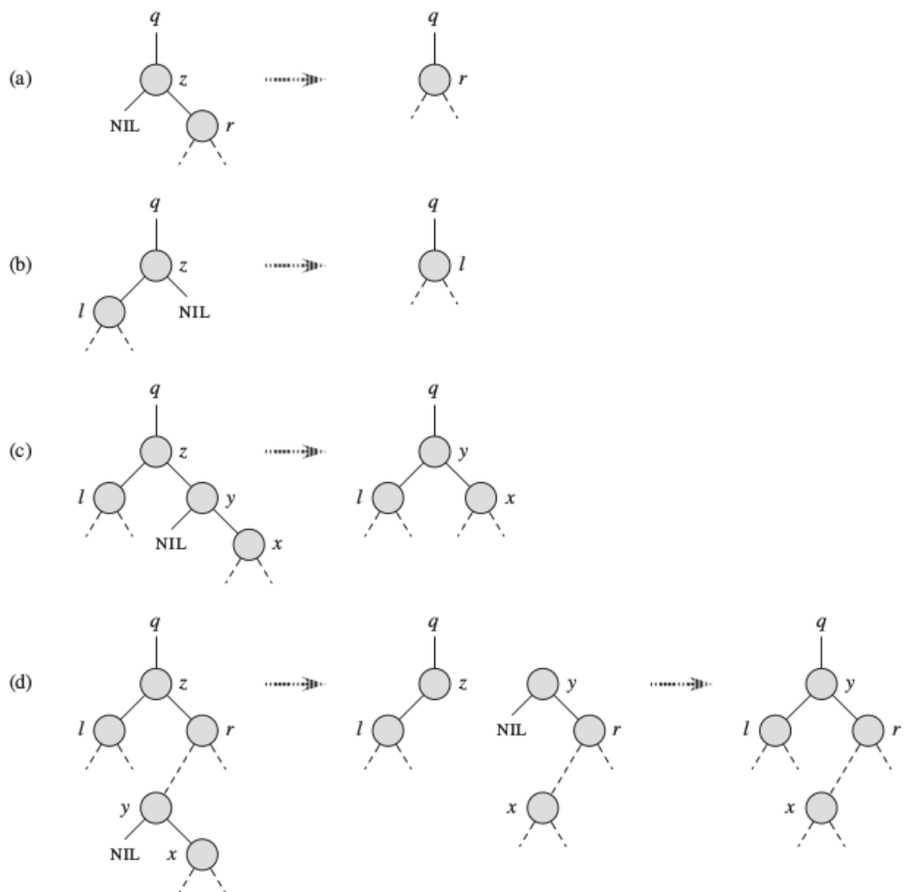
  ```
  BST-INSERT(z)
      y = NIL
      x = root
  ```

```
while x != NIL:
    y = x
    if z.key < x.key then:
       x = x.left
    else x = x.right

z.p = y

if y == NIL then:
   root = z
elseif z.key < y.key then:
   y.left = z
else y.right = z
```

- *delete*(z): jest to najbardziej złożony przypadek (TODO - na razie wklejam stronę z Cormena)



Ilustracja: TODO (zilustrować każdą z operacji.)

Zadanie 6.3.2. *Wyjaśnij, dlaczego* `BST-SUCCESSOR(n)` *działa poprawnie. Zaproponuj pseudokod operacji* `BST-PREDECESSOR(n)`*.*

(Uwaga - na wykładzie podano wersje w których niektóre operacje przyjmują jako argument wartość klucza.)

Zadanie 6.3.3. *Zaproponuj pseudokod procedury* `BST-CUT(T,a,b)`*, która przycina drzewo T w taki sposób, że pozostają w nim tylko elementy o kluczach $a \leq k \leq b$. Podpowiedź: zacznij od przypadku, gdy korzeń $a \leq root.key \leq b$.*