

- 4 W omówionym w tym temacie programie znajdującym najkrótszą drogę do wyjścia z labiryntu zastąp kolejkę stosem. Wkładaj na stos pola w takiej kolejności, aby uzyskać taką samą drogę jak za pomocą programu rekurencyjnego przedstawionego w tym temacie (dla takiej samej planszy). Plik tekstowy z opisem planszy prześle ci nauczyciel (np. *labirynt.txt*).

- 5 Przyjmij następującą reprezentację planszy z labiryntem: tablica dwuwymiarowa składająca się z elementów typu pole.

```
struct pole
{
    int x;
    bool D, G, L, P;
};
```

Na każdym polu można stanąć. O tym, czy można przejść na sąsiednie pole, decydują wartości logiczne określające cztery kierunki: D – dół, G – góra, L – lewo, P – prawo. Napisz program znajdujący najkrótszą drogę wyjścia z labiryntu. Dane planszy odczytaj z pliku tekstowego, który otrzymasz od nauczyciela (np. *labirynt\_logiczny.txt*). W pliku znajduje się opis planszy o wymiarach  $10 \times 10$ . Jeden wiersz pliku opisuje jeden wiersz planszy i składa się z 40 znaków. Jedno pole opisane jest czterema cyframi 0 lub 1 określającymi kolejno wartości D, G, L, P.

- 6 Poszukaj w dostępnych źródłach informacji na temat algorytmów automatycznego generowania plansz labiryntów, a następnie napisz program generujący taką planszę. Przyjmij następującą reprezentację planszy: tablica dwuwymiarowa składająca się z elementów typu pole.

```
struct pole
{
    int x;
    bool D, G, L, P;
};
```

Na każdym polu można stanąć. O tym, czy można przejść na sąsiednie pole, decydują wartości logiczne określające cztery kierunki: D – dół, G – góra, L – lewo, P – prawo.

- 7 Napisz program rozwiązujący problem ustawienia hetmanów na szachownicy o wymiarach  $n \times n$  ( $n$  – dana liczba całkowita dodatnia). Należy na niej ustawić  $n$  figur hetmanów tak, aby się nie szachowały. Program powinien wypisać współrzędne  $n$  pól, na których należy ustawić hetmanów.

## 3. Wykorzystanie list w rozwiązywaniu problemów

Na pewno z dzieciństwa znacie wiele wyliczanek. Być może podczas zabawy używaliście ich do wytypowania osoby, która ma wykonać jakieś zadanie. Pewna znana z historii wyliczanka stała się inspiracją do sformułowania problemu matematycznego. Zajmiemy się nim w tym temacie. Wykorzystamy przy tym dynamiczną strukturę danych o nazwie lista. Użyjemy jej także do sortowania słów według porządku, jaki stosuje się m.in. w słownikach.

### Cele lekcji

- Dowiesz się, czym jest lista, i poznasz różne rodzaje list.
- Zrozumiesz, na czym polega problem Flawiusza.
- Wykorzystasz listę do symulacji problemu Flawiusza oraz porządkowania słów leksykograficznie.

### 3.1. Czym jest lista?

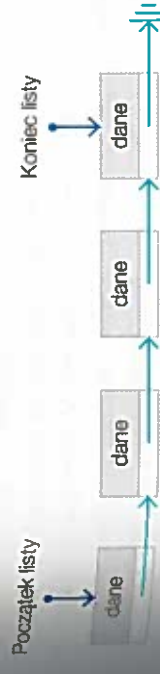
**Lista** (ang. *list*) to dynamiczna struktura danych, w której dostęp do **Lista** elementów jest sekwencyjny. Oznacza to, że od danego elementu listy można przejść bezpośrednio do elementu sąsiedniego. Aby dostać się do elementu znajdującego się na  $i$ -tym miejscu listy, trzeba przejść przez wszystkie elementy, które nas od niego oddzielają. Nowy element można wstawić w dowolne miejsce listy, można też usunąć dowolny element z listy. W liście przechowuje się dane tego samego typu. Struktury danych **stos** i **kolejka** są szczególnymi przypadkami listy.

**Stos,**  
s. 11 [↗](#)  
**Kolejka,**  
s. 39 [↗](#)

**Dynamiczna struktura danych,**  
s. 12 [↗](#)

### Rodzaje list

Rysunek 3.1 przedstawia przykład **listy jednokierunkowej**. W takiej liście każdy element przechowuje oprócz danych informację o tym, który element jest następny. Listę jednokierunkową można przeglądać tylko w jedną stronę: od początku do końca. Zatem z danego elementu listy możemy uzyskać dostęp tylko do następnego elementu, poprzednie nie są dostępne.



Rys. 3.1. Przykład listy jednokierunkowej złożonej z czterech elementów

### Warto wiedzieć

W graficznym przedstawieniu listy do zaznaczenia końca danych czasami używa się znaku np. z elektroniki symbolu uziemienia.



Jeśli ostatni element listy jednokierunkowej zawiera informację o pierwszym elemencie, to taka lista jest **listą jednokierunkową cykliczną** (rys. 3.2).



Rys. 3.2. Przykład listy jednokierunkowej cyklicznej złożonej z czterech elementów

Innym przykładem listy jest **lista dwukierunkowa**. Można ją przeglądać w dwóch kierunkach: od początku do końca i od końca do początku. Każdy element takiej listy zawiera informację zarówno o tym, który element jest następny, jak i o tym, który jest poprzedni (rys. 3.3).

#### Warto wiedzieć

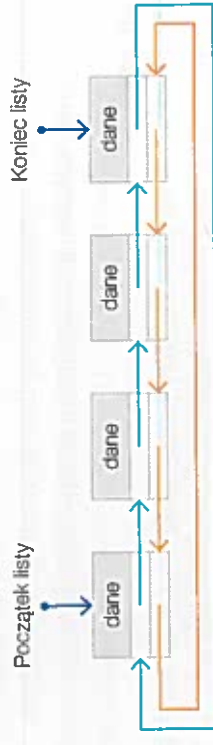
Lista dwukierunkowa zajmuje więcej pamięci komputera od listy jednokierunkowej, ponieważ w każdym elemencie oprócz informacji o elemencie następnym trzeba przechowywać informację o poprzednim.



Rys. 3.3. Przykład listy dwukierunkowej złożonej z czterech elementów

Jeśli w liście dwukierunkowej pierwszy element zawiera informację o tym, który element jest ostatni, a ostatni o tym, który jest pierwszy, to mamy do czynienia z **listą dwukierunkową cykliczną** (rys. 3.4).

**Lista dwukierunkowa cykliczna**



Rys. 3.4. Przykład listy dwukierunkowej cyklicznej złożonej z czterech elementów

#### Warto wiedzieć

Lista nie wymaga ciągłego obszaru pamięci komputera, ponieważ jej elementy nie muszą być przechowywane obok siebie. Nawet jeśli są tak przechowywane, to kolejność elementów na liście nie musi być taka sama jak w pamięci. Inaczej jest w przypadku tablicy – jej kolejne elementy muszą być zapisane w sąsiednich miejscach pamięci komputera.

#### Zapamiętaj

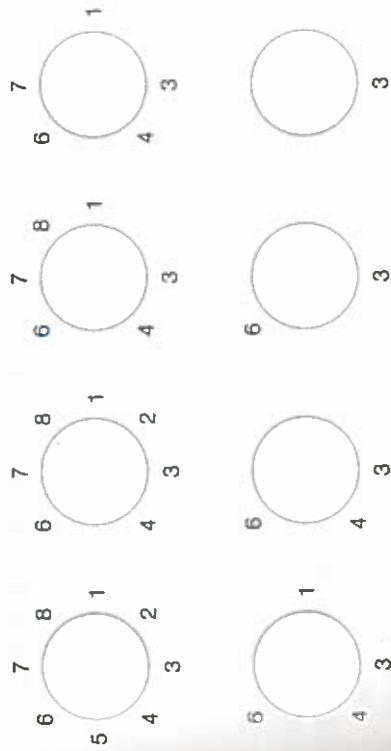
Lista jest dynamiczną strukturą danych o sekwencyjnym dostępie do elementów. Nowy element można wstawić w dowolnym miejscu listy, można też usunąć z niej dowolny element. W zależności od tego, jak możemy przeglądać elementy listy, wyróżniamy listy jednokierunkowe i dwukierunkowe. Dodatkowo oba rodzaje list mogą być listami cyklicznymi.

## 3.2. Symulacja problemu Flawiusza

Wyobraź sobie, że przy okrągłym stole siedzi  $n$  osób. Krzesła, na których siedzą, są ponumerowane od 1 do  $n$ . Cyklicznie co  $k$ -ta osoba wstaje, odstawia swoje krzesło i wychodzi. Odliczanie rozpoczynamy od pierwszej osoby, a więc jako pierwsza wyjdzie osoba siedząca na krześle o numerze  $k$ . Dalej odliczamy w tym samym kierunku. Należy wskazać osobę (numer jej krzesła), która jako ostatnia wstanie od stołu.

To zadanie jest jedną z wersji **problemu Józefa Flawiusza**. Ogólnie

polega on na tym, że ze zbioru  $n$  elementów usuwa się co  $k$ -ty i trzeba określić, który element zostanie usunięty jako ostatni. Rysunek 3.5 przedstawia symulację problemu dla 8 osób, gdy od stołu wstaje co 5. osoba.



Rys. 3.5. Przykład symulacji problemu dla 8 osób, gdy co 5. wstaje od stołu

### Ćwiczenie 1

Rozważ opisany problem dla 7 osób, gdy co 3. wstaje od stołu.

#### A to ciekawe

## Wylicznika na śmierć i życie

Nazwa problemu Józefa Flawiusza pochodzi od imienia i nazwiska żydowskiego historyka z I w. n.e. W powstaniu Żydów przeciwko Rzymianom dowodził on obroną Jotapaty. Według niektórych źródeł, gdy miasto się poddało, ukrył się wraz z grupą powstańców. Ponieważ nie chcieli trafić do niewoli, a religia zabraniała im samobójstwa, zdecydowali, że będą losować, kto kogo ma zabić. Odliczali co określoną liczbę i każda wyznaczona osoba miała zginąć z rąk następnej wyznaczonej osoby. Ten, kto zostanie, miał sam odebrać sobie życie. Gdy jednak ostatnimi dwoma okazali się Flawiusz z jednym z towarzyszy, obaj wybrali niewolę.



**Warto wiedzieć**

Według niektórych źródeł pierwszą osobą, która przekształciła historię Józefa Flawiusza w problem matematyczny, był szesnastowieczny francuski matematyk Claude-Gaspard Bachet de Méziriac.

Oto specyfikacja problemu Flawiusza:

**Specyfikacja**

**Dane:**  $n$  – liczba całkowita dodatnia określająca liczbę elementów położonych na okręgu,

$k$  – liczba całkowita dodatnia określająca, co który element jest usuwany.

**Wynik:**  $m$  – liczba całkowita dodatnia określająca numer elementu, który zostanie usunięty jako ostatni.

Założmy, że mamy cykliczną listę jednokierunkową o nazwie *lista* złożoną z  $n$  elementów pamiętających kolejne liczby całkowite dodatnie od 1 do  $n$ . Rysunek 3.6 przedstawia przykład takiej listy dla  $n = 8$ .



Rys. 3.6. Cykliczna lista jednokierunkowa złożona z liczb od 1 do 8

Ponieważ chcemy wyznaczyć element, który będzie usunięty jako ostatni, trzeba usunąć z listy  $n - 1$  elementów, czyli doprowadzić do sytuacji, że lista będzie się składać z jednego elementu. Dlatego w algorytmie symulującym problem Flawiusza liczba powtórzeń zewnętrznej pętli będzie równa  $n - 1$ . Żeby wyznaczyć kolejny element do usunięcia, trzeba przesunąć się na liście o  $k - 1$  elementów. Na przykład jeśli  $k = 2$ , to za pierwszym razem usuwamy drugi element, czyli należy się ustawić na następnym elemencie za pierwszym (przesunąć o jedną pozycję). Zapis algorytmu w pseudokodzie może być następujący:

```
dla i ← 1, 2, ..., n - 1 wykonuj
    dla j ← 1, 2, ..., k - 1 wykonuj
        przejdź do następnego elementu listy
    usuń bieżący element z listy
```

**3.3. Program symulujący problem Flawiusza**

W programie symulującym problem Flawiusza użyjemy szablonu **list** z biblioteki STL. Ogólna deklaracja **listy** jest następująca:

```
list<typ elementów listy> nazwa_listy;
```

Korzystanie z typu **list** jest możliwe po dołączeniu do programu biblioteki **list** (dyrektywa `#include <list>`). Typ **list** reprezentuje listę dwukierunkową. My będziemy przeglądać listę tylko w jednym kierunku. Ponieważ typ **list** nie reprezentuje listy cyklicznej, po rozpoznaniu końca listy zaczniemy ponownie jej przeglądanie od początku.

Lista dwukierunkowa.  
s. 48

Dostęp do elementów listy jest możliwy dzięki iteratorom. **Iterator** to iterator jest zmienną pozwalającą na sekwencyjne przeglądanie zawartości struktury danych z biblioteki STL. Najczęściej zawiera wskaźnik (adres w pamięci) danego elementu struktury danych.

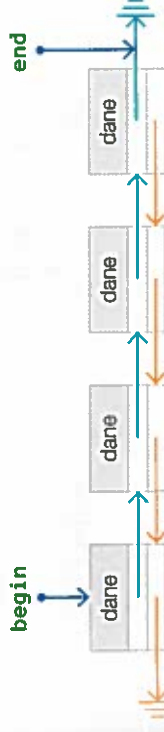
Ogólna postać deklaracji iteratora dla typu **list** jest następująca:  **Deklaracja iteratora dla typu list**

```
list<typ elementów listy>::iterator nazwa_iteratora;
```

Dostępne są różne metody, które zwracają iteratory. Dwie z nich, obie bezparametrowe, to:

► **metoda begin** – zwraca iterator wskazujący na pierwszy element listy; jeśli lista jest pusta, metoda zwróci ten sam iterator co metoda **end**;

► **metoda end** – zwraca iterator wskazujący miejsce za ostatnim elementem na liście; ponieważ iterator nie wskazuje żadnego elementu listy, nie należy wykonywać operacji na wyniku metody **end**.

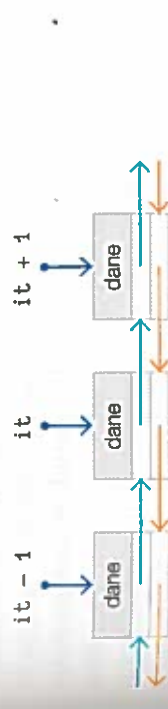


Rys. 3.7. Wskazania iteratorów **begin** i **end** dla listy dwukierunkowej

Na przykład przejście elementów listy o nazwie *lista* i wypisanie wartości wszystkich tych elementów zrealizują następujące instrukcje:

```
list<int> lista;
list<int>::iterator it;
...
for (it=lista.begin(); it!=lista.end(); it++)
    cout<<*it;
```

Żeby odwołać się do wartości elementu wskazywanego przez iterator, należy przed jego nazwą umieścić znak `*`. Oznacza to pobranie wartości wskazywanej przez iterator (pamiętanej pod danym adresem). Inkrementacja iteratora (czyli powiększenie jego wartości o 1: `it++`) w pętli **for** oznacza przejście do następnego elementu listy.



Rys. 3.8. Fragment listy z wartościami iteratorów *it*, *it - 1* oraz *it + 1*

Kod źródłowy funkcji **main** programu symulującego problem Flawiusza, zgodnie ze specyfikacją podaną na s. 50, może być następujący.

**Warto wiedzieć**

W przypadku listy dwukierunkowej (typu **list**) można wykonywać także dekrementację iteratora (pomniejszanie go o 1), co oznacza przejście do poprzedniego elementu listy.



**Fragment kodu**  **źródłowego programu symulującego problem Flawiusza – funkcja main**

```
1. int main()
2. {
3.     int n, k;
4.     list<int> lista;
5.     list<int>::iterator it;
6.     cout<<"n = "; cin>>n;
7.     cout<<"k = "; cin>>k;
8.     for (int i=1; i<=n; i++) lista.push_back(i);
9.     it=lista.begin();
10.    for (int i=1; i<=n; i++)
11.    {
12.        for (int j=1; j<=k; j++)
13.        {
14.            it++;
15.            if (it==lista.end()) it=lista.begin();
16.        }
17.        cout<<"Krok "<<i<<" : usunięty element ";
18.        cout<<"it<<"endl;
19.        it=lista.erase(it);
20.        if (it==lista.end()) it=lista.begin();
21.    }
22.    cout<<"Ostatni element: "<<*it;
23.    return 0;
24. }
```

#### Warto wiedzieć

Aby wypisać wartość elementu będącego rozwiązaniem problemu Flawiusza, można także skorzystać z iteratora wskazującego pierwszy element listy:

`*lista.begin()` lub z bezparametrowej metody `front`, której wartością jest pierwszy element listy: `lista.front()`

#### **Metoda `push_back` dla klasy `list`**

#### Dobra rada

Pamiętaj, że każda klasa udostępnia metody umożliwiające przetwarzanie danych. Nazwy metod mogą się powiarać dla różnych klas, ale sposób korzystania z nich może być różny.

#### **Metoda `erase` dla klasy `list`**

#### Dobra rada

Więcej metod dostępnych dla klasy `list` znajdziesz w dodatku 5 na s. 426.

## Ćwiczenie 2

Napisz program symulujący problem Flawiusza zgodnie ze specyfikacją podaną na s. 50. Przetestuj działanie programu.

### 3.4. Sortowanie leksykograficzne

W słownikach i indeksach, z których korzystacie, hasła są ułożone według **porządku leksykograficznego**, czyli alfabetycznie. Aby uporządkować hasła w ten sposób, porównuje się znaki, z których są złożone, z pominięciem znaków spozu alfabetu (np. spacji, jeśli hasło jest dwuwyrzowe). Dla słów *a*, *b* słowo *a* wystąpi przed słowem *b*, gdy zachodzi jeden z dwóch przypadków.

1. Słowo *a* jest początkowym fragmentem słowa *b*. Na przykład słowo *kot* jest słowem wcześniejszym od słowa *kotwica*.

2. Na co najmniej jednej pozycji występuje znak różniący słowa *a* i *b*. Wówczas słowo *a* znajduje się przed słowem *b*, jeśli na pierwszej różniącej je pozycji (od lewej) ma znak wcześniejszy, np. *kotlina* zapiszemy przed *kotwica* – decydują o tym litery na czwartej pozycji.

W ogólnym przypadku **sortowanie leksykograficzne** polega na porządkowaniu ciągów złożonych z określonych elementów. O tym, który z dwóch ciągów trzeba zapisać wcześniej, decyduje pierwsza pozycja od lewej różniąca te dwa ciągi. Jeśli taka pozycja nie istnieje, to jeden ciąg jest początkowym fragmentem drugiego.

Rozwiążemy problem sortowania leksykograficznego słów złożonych z liter pełnego polskiego alfabetu. W praktyce alfabet możemy zdefiniować dowolnie – wystarczy określić zbiór znaków go tworzących i kolejność ich występowania.

#### Sortowanie leksykograficzne słów

Do uporządkowania leksykograficznego słów wykorzystamy **sortowanie kubełkowe**. Polega ono na grupowaniu elementów mających wspólne cechy w tzw. kubełkach, a następnie przepisywaniu ich we właściwej kolejności. Wspólną cechą słów będzie taka sama litera na tej samej pozycji. Utworzymy po jednym kubełku dla każdej litery występującej w wyrazach. Kubełki będą ustawione w kolejności alfabetycznej. Załóżmy, że słowa są pamiętane w liście. Usuamy wyrazy z listy i wkładamy do odpowiednich kubełków. Następnie usuwamy słowa z kolejnych kubełków i dopisujemy je na końcu listy. Jeśli w kubełku znajduje się kilka wyrazów, to dopisujemy je do listy w kolejności, w jakiej były wkładane do kubełka.

Litery słów będziemy przeglądać od prawej do lewej. Najpierw rozpatrzmy najdłuższe wyrazy – uporządkujemy je według liter na ostatniej pozycji. Następnie posortujemy słowa mające literę na pozycji o jeden mniejszej itd. – aż do uporządkowania słów względem pierwszej pozycji.



# Algorytm porządkujący słowa leksykograficznie

Uporządkujemy leksykograficznie listę słów: *wir, kra, kwiat, wiatr, świat, świt, kraj*. W tych słowach występuje osiem różnych liter: *a, i, j, k, r, ś, t, w*. Użyjemy zatem ośmiu kubeków reprezentujących te litery. Kubeczki muszą być ustawione w kolejności alfabetycznej. Najdłuższe wyrazy są pięcioliterowe, więc sortowanie rozpoczniemy, uwzględniając litery na piątej pozycji. Sposób postępowania ilustruje krok 1. W kolejnych krokach podobnie porządkujemy słowa według liter z wcześniejszych pozycji.

Lista słów na początku algorytmu:

- wir kra kwiat wiatr świat świt kraj

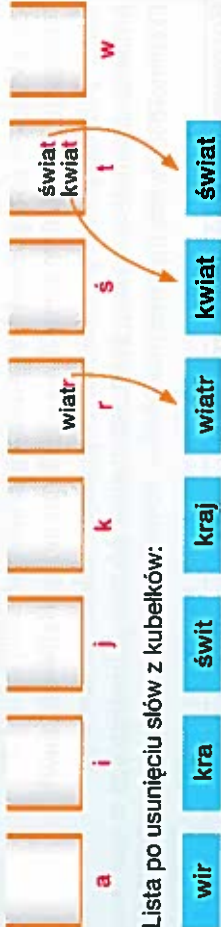
## Krok 1

Sortujemy słowa według liter na piątej pozycji. Są trzy wyrazy pięcioliterowe: *kwiat, wiatr i świat* – usuwamy je kolejno z listy i każdy umieszczamy w kubeczku z literą odpowiadającą ostatniej literze wyrazu. Następnie usuwamy wyrazy z kolejnych kubeków i dopisujemy je na końcu listy. Z kubeczka oznaczonego literą *t* usuwamy najpierw słowo, które znalazło się w nim jako pierwsze. Listę słów i kubeczki przeglądamy od lewej do prawej.

Lista po usunięciu słów mających literę na piątej pozycji:

- wir kra świt kraj

Usunięte słowa umieszczone w kubekach:



Lista po usunięciu słów z kubeków:

- wir kra świt kraj wiatr świat

## Krok 2

Porządkujemy słowa według liter na czwartej pozycji. Usuwanie z listy słowa mające literę na czwartej pozycji i umieszczamy je we właściwych kubekach.

Lista po usunięciu słów mających literę na czwartej pozycji:

- wir kra

Usunięte słowa umieszczone w kubekach:



Lista po usunięciu słów z kubeków:

- wir kra kwiat świat kraj świt wiatr

## Krok 3

Porządkujemy słowa według liter na trzeciej pozycji. Wszystkie wyrazy mają literę na trzeciej pozycji, więc po przeniesieniu słów do kubeków lista jest pusta. Słowa przyporządkowujemy do odpowiednich kubeków, a potem umieszczamy je na liście we właściwej kolejności.

Słowa umieszczone w kubekach:



Lista po usunięciu słów z kubeków:

- kra kraj wiatr kwiat świat świt wir

## Krok 4

Porządkujemy słowa, uwzględniając litery na drugiej pozycji. Podobnie jak w poprzednim kroku, wszystkie wyrazy mają literę na drugiej pozycji, więc po przeniesieniu słów do kubeków lista jest pusta.

Słowa umieszczone w kubekach:



Lista po usunięciu słów z kubeków:

- wiatr wir kra kraj kwiat świat świt

## Krok 5

Ustalamy kolejność słów na liście według liter na pierwszej pozycji. Jest to ostatni krok algorytmu – nie ma już więcej pozycji do rozpatrzenia.

Słowa umieszczone w kubekach:



Lista po usunięciu słów z kubeków:

- kra kraj kwiat świat świt wiatr wir

Wynikiem działania algorytmu jest lista:

- kra kraj kwiat świat świt wiatr wir



### Ćwiczenie 3

Korzystając z algorytmu sortowania kubełkowego, uporządkuj słowa: *krzesło, krzak, krem, kres, kraina, kret, kraksa, kredyt*.

#### Zapamiętaj

Sortowanie leksykograficzne polega na porządkowaniu ciągów złożonych z określonych elementów. W przypadku porządkowania słów elementami tymi są litery. O tym, który ciąg powinien być zapisany wcześniej, decyduje pierwsza pozycja (od lewej), na której w tych ciągach występują różne elementy. Jeśli taka pozycja nie istnieje, to krótszy ciąg zapisujemy przed dłuższym.

### Implementacja algorytmu sortowania leksykograficznego

Sformułujemy najpierw specyfikację problemu leksykograficznego sortowania słów oraz zapiszemy omówiony algorytm w pseudokodzie. Słowa przechowamy w liście, a zawartość kubełków – w kolejkach.

#### Specyfikacja

**Dane:** alfabet – napis złożony z uporządkowanych znaków tworzących alfabet,

mdl – liczba całkowita określająca maksymalną długość słowa,

lista – lista słów o maksymalnej długości mdl, złożonych ze znaków napisu alfabet.

**Wynik:** lista – lista danych słów uporządkowanych leksykograficznie.

```
dla i ← mdl - 1, mdl - 1, ..., 0 wykonuj
  dopóki nie koniec listy wykonuj
    słowo ← aktualny element listy
    jeśli długość słowa > i to
      usuń aktualny element listy
      dodaj słowo do kolejki dla znaku słowo[i]
dla j ← 0, 1, ..., długość alfabetu - 1 wykonuj
  dopóki nie pusta kolejka
    dla znaku alfabet[j] wykonuj
      dopisz na końcu listy element
      z początku kolejki
      usuń element z początku kolejki
```

Do implementacji algorytmu wykorzystamy typ **list** oraz tablicę kolejek typu **queue**. Słowa do posortowania program wczyta z pliku tekstowego – w każdym wierszu pliku znajdzie się jedno słowo. Wynik sortowania program również zapisze w pliku tekstowym.

#### Dobra rada

Pamiętaj, że program uwzględniający polskie znaki diakrytyczne zadziała poprawnie, jeśli plik tekstowy wykorzystywany przez program oraz plik **cpp** z kodem źródłowym będą używały tej samej strony kodowej. W środowisku Code::Blocks pracującym w polskiej wersji systemu Windows użyj strony kodowej CP1250.

56

W programie zdefiniujemy dwie stałe. Jedna z nich będzie określała zestaw znaków tworzących alfabet, druga – liczbę znaków w alfabecie. Wykorzystamy pełny polski alfabet. Definicja tych stałych może wyglądać następująco:

```
const string alfabet="abcdefghijklmnopqrstuvwxyz";
const int N=35;
```

Kody źródłowe funkcji odczytującej słowa z pliku *słowa.txt* i budującej listę oraz funkcji zapisującej słowa do pliku *słownik.txt* mogą wyglądać tak jak poniżej. Podczas odczytywania słów wyliczamy też maksymalną długość słowa.

```
1. void BudujListe(list<string> &lista, int &mdl)
2. {
3.     ifstream we("słowa.txt");
4.     string s;
5.     mdl=0;
6.     while (we>>s)
7.     {
8.         lista.push_back(s);
9.         if (s.size()>mdl) mdl=s.size();
10.    }
11.    we.close();
12. }
13.
14. void ZapiszListe(list<string> lista)
15. {
16.     ofstream wy("słownik.txt");
17.     list<string>::iterator it;
18.     for (it=lista.begin(); it!=lista.end(); it++)
19.         wy<<*it<<endl;
20.     wy.close();
21. }
```


Fragment kodu źródłowego programu porządkującego słowa leksykograficznie – definicja funkcji wczytującej słowa z pliku i budującej listę oraz funkcji zapisującej słowa do pliku

Zwróć uwagę na parametry funkcji *BudujListe*. Obydwa są przekazywane przez referencję, ponieważ funkcja ma zwrócić zbudowaną listę oraz znaleźć maksymalną długość słowa.

W linii 6 w warunku pętli **while** do zmiennej *s* wczytywane jest słowo z pliku tekstowego. Jeśli nie uda się go wczytać (dojdziemy do końca pliku), instrukcja zwróci wartość **false**. W linii 8 dodajemy wczytane słowo na koniec listy, w następnej linii aktualizujemy dotychczas znalezioną maksymalną długość słowa, jeśli wczytane słowo było dłuższe. Funkcja *ZapiszListe* przegląda całą listę za pomocą iteratora *it* i zapisuje wartość aktualnego elementu do pliku łącznie ze znakiem końca wiersza.

Oto kod źródłowy funkcji *SortujSłowa*, realizującej algorytm sortowania leksykograficznego zapisany wcześniej w pseudokodzie.

57

**Fragment kodu**  **źródłowego programu**  **porządkującego słowa**  **leksykograficznie –**  **definicja funkcji**  **sortującej słowa** 

```

1. void SortujSlova(list<string> &lista, int mdl)
2. {
3.     queue<string> Kubelki[N];
4.     list<string>::iterator it;
5.     int i, j;
6.     string s;
7.     for (i=mdl-1; i>=0; i--)
8.     {
9.         it=lista.begin();
10.        while (it!=lista.end())
11.        {
12.            s=*it;
13.            if (s.size()>i)
14.            {
15.                it=lista.erase(it);
16.                j=alfabet.find(s[i]);
17.                Kubelki[j].push(s);
18.            }
19.            else it++;
20.        }
21.        for (j=0; j<N; j++)
22.        while (!Kubelki[j].empty())
23.        {
24.            lista.push_back(Kubelki[j].front());
25.            Kubelki[j].pop();
26.        }
27.    }
28. }

```

#### **Dobra rada**

Pamiętaj, że wartością metody `erase` z klasy `list` jest iterator wskazujący na element listy znajdujący się za elementem usuwanym.

W linii 3 deklarowana jest tablica `N`-elementowa, której elementami są kolejki reprezentujące kubelki. Wartość zmiennej sterującej główną pętlą `for` (linia 7–27) jest zmniejszana od wartości równej największemu indeksowi liter w słowie do zera. Pierwsza wewnętrzna pętla (linia 10–20) przegląda listę słów. W zmiennej pomocniczej `s` pamiętana jest aktualna wartość elementu listy (linia 12). Jeśli długość słowa (linia 13) jest dostatecznie duża (istnieje litera o indeksie `i`), słowo jest usuwane z listy (linia 15, iterator `it` wskazuje na następny element za usuniętym) i dodawane do odpowiedniej kolejki (linia 16–17). W przeciwnym przypadku (linia 19) słowo pozostaje na liście i przechodzi się do następnego słowa. Pętla w liniach 21–26 przegląda kolejno wszystkie kubelki. Wewnętrzna pętla (linia 22–26) przegląda pojedynczy kubelek, dopisuje jego elementy na końcu listy (linia 24) i usuwa je z kubelka (linia 25).

#### **Ćwiczenie 4**

Napisz program, który wczyta słowa z pliku otrzymanego od nauczyciela (np. `slova.txt`), następnie posortuje je w porządku leksykograficznym i zapisze w pliku tekstowym o nazwie `sloownik.txt`.

Zwróć uwagę, że jeśli w pliku z danymi będą występowały takie same słowa, to w pliku wynikowym pojawią się w tej samej kolejności. Sortowanie, w którym elementy równie zachowują swoją kolejność po uporządkowaniu, nazywamy **sortowaniem stabilnym**.

 **Sortowanie stabilne**

### 3.5. Ocena złożoności obliczeniowej algorytmu rozwiązyującego problem Flawiusza i algorytmu sortowania leksykograficznego

Czasowa złożoność obliczeniowa przedstawionej w temacie symulacji problemu Flawiusza wynosi  $O(n \cdot k)$ , gdzie  $n$  – liczba wszystkich elementów,  $k$  – liczba okręślająca, co który element jest usuwany.

Ostatni usuwany element można znaleźć sprawniej, w złożoności  $O(n)$ , bez użycia listy, korzystając z następującego wzoru rekurencyjnego:

$$f(n, k) = \begin{cases} 1 & \text{dla } n = 1 \\ ((f(n-1, k) + k - 1) \bmod n) + 1 & \text{dla } n > 1 \end{cases}$$

Gdybyśmy numerowali elementy od 0, zależność rekurencyjna byłaby następująca:

$$f(n, k) = \begin{cases} 0 & \text{dla } n = 0 \\ (f(n-1, k) + k) \bmod n & \text{dla } n > 0 \end{cases}$$

Po pierwszym usunięciu ( $k$ -tego elementu, który znajduje się na pozycji  $k-1$ ) problem redukuje się do rozwiązania problemu dla  $n-1$  elementów, z tym że kolejne odliczanie rozpoczynamy od elementu o numerze  $k$  (pierwszy element za usuniętym, dlatego dodajemy wartość  $k$ ). Otrzymany wynik sprowadzamy do zakresu od 0 do  $n-1$  (operacja **mod**).

Ponieważ w rzeczywistości numerujemy elementy od 1, najpierw odejmujemy 1, a po wykonaniu operacji **mod** dodajemy 1.

Dla przypadku, gdy  $k=2$ , można obliczyć wynik w złożoności  $O(\log n)$ , korzystając ze wzoru:

$$f(n) = \begin{cases} 1 & \text{dla } n = 1 \\ 2 \cdot f(n \div 2) - 1 & \text{dla } n > 1, n - \text{liczba parzysta} \\ 2 \cdot f(n \div 2) + 1 & \text{dla } n > 1, n - \text{liczba nieparzysta} \end{cases}$$


Czasowa złożoność obliczeniowa omówionego algorytmu sortowania leksykograficznego słów zależy od długości listy, czyli liczby słów oraz ich długości. Oznaczmy długość listy przez  $n$ , a  $k$  niech będzie maksymalną długością słowa. Operacje wstawiania oraz usuwania elementów z listy i kolejek są wykonywane w **złożoności stałej**  $O(1)$ . Złożoność czasowa tego algorytmu wynosi więc  $O(n \cdot k)$ .

#### **Warto wiedzieć**

Dla  $k=2$  istnieje także wzór jawny:

$$f(n) = 2 \cdot (n - 2^{\lfloor \log_2 n \rfloor}) + 1.$$

Znaki `l` i `u` oznaczają zaokrąglenie w dół do liczby całkowitej, nazywane podłogą.

**Złożoność stała.** Podręcznik Informatyka na czasie 2. Zakres rozszerzony, s. 167 





## Podsumowanie

- Lista to dynamiczna struktura danych, w której dostęp do elementów jest sekwencyjny. Nowy element można wstawić w dowolnym miejscu, można też usunąć dowolny element z listy.
- Wyróżniamy listy jednokierunkowe i dwukierunkowe. Dodatkowo mogą one być listami cyklicznymi.
- Stos i kolejka są szczególnymi przypadkami listy.
- Problem Flawiusza polega na cyklicznym eliminowaniu elementu położonego co określonej liczbie pozycji, aż pozostanie jeden element.
- Do symulacji problemu Flawiusza można użyć listy cyklicznej.
- Sortowanie leksykograficzne polega na porządkowaniu ciągów złożonych z określonych elementów. O tym, który ciąg wystąpi wcześniej, decyduje pierwsza pozycja różniąca ciągi. Jeśli taka pozycja nie istnieje, to wcześniej znajdzie się krótszy ciąg.
- Przykładem sortowania leksykograficznego jest porządkowanie haseł w słownikach i skrowidzach (indeksach).
- Do sortowania leksykograficznego słów możemy wykorzystać listę oraz algorytm sortowania kubełkowego.
- Sortowanie kubełkowe polega na grupowaniu elementów o takich samych cechach w tzw. kubełkach, a następnie przepisywaniu elementów z kolejnych kubełków. Kolejność kubełków jest ściśle określona – zgodna z alfabetem. Wspólną cechą słów jest taka sama litera na danej pozycji. Litery słów przegląda się od prawej strony do lewej.



## Zadania

- 1 Napisz funkcję rekurencyjną rozwiązującą problem Flawiusza dla przypadku, gdy co drugi element jest usuwany, w złożoności czasowej  $O(\log n)$ , gdzie  $n$  – liczba elementów zbioru.
- 2 Napisz funkcję rekurencyjną rozwiązującą problem Flawiusza dla ogólnego przypadku, gdy usuwany jest co  $k$ -ty element, w złożoności czasowej  $O(n)$ , gdzie  $n$  – liczba elementów zbioru.
- 3 Napisz program losujący tablicę  $n$  liczb całkowitych z zakresu od  $-9$  do  $9$  i sortującą niemalejąco według wartości bezwzględnej liczb. Sortowanie powinno być stabilne, tzn. elementy równe co do wartości bezwzględnej powinny zachować kolejność występowania w posortowanym ciągu. Na przykład dla danych:  
-6 5 -1 2 1 -1 -5 6 -2  
wynikiem będzie ciąg: -1 1 -1 2 -2 5 -5 -6 6
- 4 Program sortujący wyrazy leksykograficznie, omówiony w tym temacie, zmodyfikuj tak, aby podczas odczytywania słów z pliku wyznaczał nie tylko maksymalną długość słowa, lecz także alfabet złożony wyłącznie z liter, które występują w co najmniej jednym słowie. Plik ze słowami do sortowania otrzymasz od nauczyciela (np. *slova\_alfabet.txt*).

- 5 Napisz program losujący tablicę  $n$  liczb całkowitych nieujemnych i sortującą ją niemalejąco według cyfry jedności, tzn. najpierw w posortowanym ciągu wystąpią liczby zakończone cyfrą 0, potem 1 itd. Sortowanie powinno być stabilne, tzn. elementy z taką samą ostatnią cyfrą powinny zachować kolejność występowania w posortowanym ciągu. Na przykład dla danych:  
62 59 12 24 19 17 52 60 22  
wynikiem będzie ciąg:  
60 62 12 52 22 24 17 59 19



- 6 W pliku tekstowym otrzymanym od nauczyciela (np. *liczby2.txt*) znajduje się 10 ciągów liczbowych. Każdy z ciągów zapisany jest w oddzielnym wierszu, składa się z maksymalnie 10 liczb całkowitych oddzielonych spacjami i jest zakończony liczbą 0, która nie należy do ciągu. Napisz program sortujący ciągi leksykograficznie i zapisujący uporządkowane ciągi do pliku tekstowego *liczby2\_wynik.txt*. Na przykład ciąg:  
62 59 12 20 19 17 52 60 22  
62 59 12 24 80  
62 59 12

powinny być uporządkowane następująco:

```
62 59 12
62 59 12 20 19 17 52 60 22
62 59 12 24 80
```



- 7 W pliku tekstowym, który otrzymasz od nauczyciela (np. *hasla.txt*), znajduje się 10 haseł, każde w oddzielnym wierszu. Każde hasło składa się z jednego lub kilku słów (maksymalnie 5, złożonych z małych liter alfabetu łacińskiego) oddzielonych spacjami. Napisz program, który uporządkuje hasła leksykograficznie i zapisze wyniki do pliku *indeks.txt*. Jeśli dwa hasła mają to samo słowo na tej samej pozycji, to w kolejnych wystąpieniach powinno ono być zastąpione myślnikiem. Na przykład dla haseł:

```
algorytm rekurencyjny
algorytm iteracyjny
algorytm naiwny
algorytm obliczania silni iteracyjny
algorytm obliczania silni rekurencyjny
algorytm
algorytm obliczania silni
plik wynikowy powinien mieć postać:
algorytm
- iteracyjny
- naiwny
- obliczania silni
- - - iteracyjny
- - - rekurencyjny
- rekurencyjny
```