



**POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I
INFORMATYKI**

**KIERUNEK STUDIÓW
INFORMATYKA**

***MATERIAŁY DO ZAJĘĆ
LABORATORYJNYCH/ĆWICZENIOWYCH***

Podstawy algorytmiki

Autor:
mgr inż. Monika Kaczorowska

Lublin 2020

INFORMACJA O PRZEDMIOCIE

Cele przedmiotu:

- Cel 1. Zapoznanie studentów z technikami i metodami tworzenia algorytmów i oceną ich złożoności oraz algorytmami sortowania, wyszukiwania oraz strukturami danych tj. listy, drzewa, kopce, grafy.
- Cel 2. Nabycie przez studentów umiejętności oprogramowania i użycia poznanych algorytmów i struktur danych.

Efekty kształcenia w zakresie umiejętności:

- Efekt 1. Umie przeanalizować i zastosować w praktyce algorytmy sortowania i wyszukiwania
- Efekt 2. Potrafi zaimplementować działania na strukturach danych i grafach.

Literatura do zajęć:

- Literatura podstawowa
 - 1) Algorytmy i struktury danych, L. Banachowski, K. Diks, W. Rytter, Wydawnictwo Naukowe PWN, 2020.
 - 2) Wprowadzenie do algorytmów, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Wydawnictwo Naukowe PWN, 2018.
- Literatura uzupełniająca
 - 1) Algorytmy + struktury danych = programy, N. Wirth, Wydawnictwa Naukowo - Techniczne, 2006.

Metody i kryteria oceny:

- Oceny częściowe:
 - Przygotowanie merytoryczne do zajęć laboratoryjnych na podstawie: wykładów, literatury, pytań kontrolnych do zajęć.
 - Dwa kolokwia: próg zaliczeniowy 51% z każdego z kolokwiów.
 - Realizacja zadań praktycznych w trakcie zajęć laboratoryjnych.
- Ocena końcowa - zaliczenie przedmiotu:
 - Pozytywne oceny częściowe.
 - Ewentualne dodatkowe wymagania prowadzącego zajęcia.



Plan zajęć laboratoryjnych:

Lab1.	Proste algorytmy realizujące: min, max, średnia
Lab2.	Algorytmy sortowania elementów w tablicy: BubbleSort, InsertSort, SelectSort
Lab3.	Algorytm sortowania QuickSort
Lab4.	Algorytmy podziału zbioru na dwie albo trzy części
Lab5.	Algorytmy tekstowe
Lab6.	Algorytmy tekstowe z użyciem funkcji haszujących
Lab7.	Wyszukiwanie informacji w zbiorze
Lab8.	Kolokwium 1
Lab9.	Implementacja stosu
Lab10.	Implementacja kolejki
Lab11.	Implementacja list
Lab12.	Drzewa BST
Lab13.	Sortowanie przez kopcowanie
Lab14.	Algorytmy grafowe
Lab15.	Kolokwium 2

Algorytmy realizowane podczas laboratorium będą implementowane w języku C++ w środowisku Code::Blocks. W pierwszej części zajęć przedstawione zostaną najpopularniejsze algorytmy sortowania zbiorów, podziału zbioru oraz wyszukiwania elementu w zbiorze. Druga część laboratoriów dotyczy struktur danych. Po każdej z części przewidziane jest kolokwium.

LABORATORIUM 1. PROSTE ALGORYTMY REALIZUJĄCE: MIN, MAX, ŚREDNIA

Cel laboratorium:

- wprowadzenie do algorytmiki,
- omówienie prostych algorytmów wyszukiwania elementów w zbiorze: minimalnego oraz maksymalnego, obliczania średniej arytmetycznej,
- przypomnienie podstawowych sposobów zapisu algorytmu oraz wiadomości o programowaniu.

Zakres tematyczny zajęć:

- powtórzenie wiadomości o typach wskaźnikowych,
- powtórzenie wiadomości o tablicach jedno i dwuwymiarowych,
- powtórzenie wiadomości o tworzeniu funkcji,
- wprowadzenie do algorytmiki,
- przedstawienie sposobów zapisu algorytmu,
- generowanie liczb pseudolosowych.

Pytania kontrolne:

- 1) Czym różni się zmienna statyczna od zmiennej wskaźnikowej?
- 2) Do czego służy operator *?
- 3) Do czego służy operator &?
- 4) Jak zdefiniować wskaźnik na tablicę jednowymiarową, a jak na dwuwymiarową?
- 5) W jaki sposób przekazać zmienną statyczną do funkcji tak, aby po wyjściu z funkcji wyniki dokonanych modyfikacji zostały zachowane?
- 6) Jak jest zbudowana funkcja?
- 7) Jak wygenerować liczby naturalne pseudolosowe z danego przedziału?
- 8) W jaki sposób znaleźć element maksymalny w tablicy?

Powtórzenie wiadomości o wskaźnikach:

Wskaźnik to zmienna przechowująca adres zmiennej. Na listingu 1.1 zadeklarowane zostały dwie zmienne: *a* – zmienna statyczna oraz *b* – zmienna wskaźnikowa. Listing 1.1 zawiera:

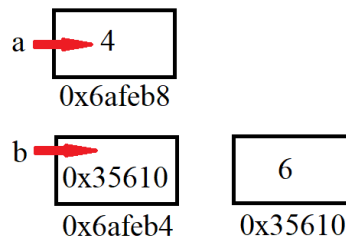
- deklarację zmiennej *a* (linijka 1) oraz przypisanie jej wartości (linijka 2),
- sposoby odwołania się do zmiennej *a*:
 - linijka 3 – wyświetlana jest wartość zmiennej *a*,
 - linijka 4 – wyświetlany jest adres pod którym, zapisana jest zmienna *a*
 - linijka 5 – wyświetlana jest wartość zmiennej *a*,
- deklarację zmiennej wskaźnikowej *b* wraz z przydzieleniem pamięci (linijka 6) oraz przypisanie wartości do komórki, której adres przechowywany jest w zmiennej *b* (linijka 7),
- sposoby odwołania się do zmiennej *b*:
 - linijka 8 – wyświetlana jest wartość, która zapisana jest w komórce, której adres przechowywany jest w zmiennej *b*,
 - linijka 9 – wyświetlany jest adres przechowywany w zmiennej *b*,

- o linijka 10 – wyświetlany jest adres zmiennej *b*,
- o linijka 11,12 - wyświetlany jest adres przechowywany w zmiennej *b*,
- o linijka 13 – usunięcie przydzielonej pamięci zmiennej *b*,

```
1  int a;  
2  a=4;  
3  cout<<"a: "<<a<<endl;  
4  cout<<"&a: "<<&a<<endl;  
5  cout<<"*&a: "<<*&a<<endl;  
  
6  int *b=newint;  
7  *b=6;  
8  cout<<"*b: "<<*b<<endl;  
9  cout<<"b: "<< b<<endl;  
10 cout<<"&b: "<<&b<<endl;  
11 cout<<"*&b: "<<*&b<<endl;  
12 cout<<"&*b: "<<&*b<<endl;  
13 delete b;
```

Listing. 1.1. Deklaracja oraz sposoby odwoływania się do zmiennych

Na rys. 1.1 przedstawiony został schematyczny rysunek ilustrujący komórki pamięci, w których przechowywane są zmienne *a* oraz *b*. Na rys. 1.2 przedstawiony został wynik działania programu z listingu 1.1.



Rys. 1.1. Schematyczny układ komórek pamięci zmiennych a oraz b

Wyjście programu przedstawia następujące dane:

```
a: 4  
&a: 0x6afeb8  
*&a: 4  
  
*b: 6  
b: 0x35610  
&b: 0x6afeb4  
*&b: 0x35610  
&*b: 0x35610
```

Rys. 1.2. Wynik działania programu z listingu 1.1

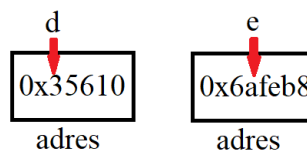
W celu pobrania adresu komórki należy użyć operatora `&`. Natomiast operator `*` służy do pobrania wartości przechowywanej w komórce, której adres przechowywany jest w zmiennej wskaźnikowej. Listing 1.2 jest kontynuacją listingu 1.1 i przedstawia operację podstawienia pod wskaźniki. Listing 1.2 zawiera:

- linijka 1,2 – deklarację zmiennych d oraz e ,
- linijka 3 – podstawienie pod zmienną d zmiennej b ; zmienna d przechowuje ten sam adres, który jest przechowywany w zmiennej b ,
- linijka 4 – wyświetlany jest adres przechowywany w zmiennej b oraz d ,
- linijka 5 – wyświetlana jest wartość, która zapisana jest w komórce, której adres przechowywany jest w zmiennej b oraz d ,
- linijka 6 – podstawienie pod zmienną e adresu zmiennej a ; zmienna e przechowuje adres zmiennej a ,
- linijka 7 - wyświetlany jest adres przechowywany w zmiennej c oraz adres zmiennej a ,
- linijka 8 - wyświetlana jest wartość, która zapisana jest w komórce, której adres przechowywany jest w zmiennej e oraz wartość zmiennej a .

```
1  int *d;  
2  int *e;  
3  d=b;  
4  cout<<"b,d: "<<b<<" "<<d<<endl;  
5  cout<<"*b,*d: "<<*b<<" "<<*d<<endl;  
6  e=&a;  
7  cout<<"e,&a: "<<e<<" "<<&a<<endl;  
8  cout<<"*e,a: "<<*e<<" "<<a<<endl;
```

Listing. 1.2. Deklaracja oraz sposoby odwoływania się do zmiennych

Na rys. 1.3 przedstawiony został schematyczny rysunek reprezentujący zmienne d oraz e . Na rys. 1.4 przedstawiony został wynik działania programu z listingu 1.3. Warto zauważyć, że zmienne wskaźnikowe b oraz d wskazują na tę samą komórkę pamięci. W związku z tym zmiana wartości przechowywanej pod adresem 0x35610 spowoduje, że obie wartości $*b$ oraz $*d$ ulegną zmianie. Podobna sytuacja zachodzi dla zmiennej wskaźnikowej e oraz zmiennej statycznej a .



Rys. 1.3. Schematyczny układ zmiennych d oraz e

```
b,d: 0x35610,0x35610  
*b,*d: 6,6  
e,&a: 0x6afeb8,0x6afeb8  
*e,a: 4,4
```

Rys. 1.4. Wynik działania programu z listingu 1.3

Powtórzenie wiadomości o tablicach jedno i dwuwymiarowych:

Tablica jest uporządkowanym zbiorem zmiennych tego samego typu. Pierwszy element tablicy znajduje się pod indeksem 0.

Wyróżniamy dwa sposoby dostępu do elementów tablicy:

- zapis indeksowy: `tab[0]`, `tab[1]`, `tab[0][0]`, `tab[2][1]`,
- zapis wskaźnikowy: `*tab`, `*(tab+1)`, `**tab`, `*(*(tab+2)+1)`.

Nazwa tablicy jest wskazaniem na pierwszy element w tablicy.

Istnieją dwa typy tablic:

- statyczne,
- dynamiczne.

Zarówno podczas używania tablic statycznych, jak i dynamicznych, można używać dowolnego sposobu dostępu do elementów tablicy.

Listing 1.3 przedstawia deklarację tablicy statycznej jedno oraz dwuwymiarowej. Listing 1.4 przedstawia deklarację tablicy dynamicznej jedno oraz dwuwymiarowej. Listing 1.5 zawiera kod umożliwiający wyświetlenie zawartości tablic.

Listing 1.3 zawiera:

- linijka 2 – deklaracja tablicy *tab1*,
- linijki 3,4,5 – inicjalizacja tablicy *tab1*,
- linijka 7 – deklaracja tablicy *tab2*,
- linijki 8,9,10 – inicjalizacja tablicy *tab2*.

```
1  cout<<"TABLICA JEDNOWYMIAROWA"<<endl;  
2  int tab1[3];  
3  tab1[0]=1;  
4  tab1[1]=2;  
5  tab1[2]=3;  
5  
6  cout<<"TABLICA DWUWYMIAROWA"<<endl;  
7  int tab[2][3];  
8  for(int i=0;i<2;i++){  
9      for(int j=0;j<3;j++){  
10         tab[i][j]=i+j;}}
```

Listing 1.3. Tablica statyczna jedno i dwuwymiarowa

Listing 1.4 zawiera:

- linijka 2 – deklaracja wskaźnika *tab1* oraz przydzielenie pamięci – 3 komórki,
- linijki 3,4,5 – inicjalizacja tablicy *tab1*,
- linijki 8,9,10 – deklaracja wskaźnika *tab2* oraz przydział pamięci – 6 komórek (2 wiersze, 3 kolumny),
- linijki 12,13,14 – inicjalizacja tablicy *tab2*,
- linijki 16,17,18,19 – zwolnienie pamięci – usunięcie wskaźników *tab1* oraz *tab2*.

```
1  cout<<"TABLICA JEDNOWYMIAROWA"<<endl;
2  int *tab1=new int[3];
3  tab1[0]=1;
4  tab1[1]=2;
5  tab1[2]=3;
6
7  cout<<"TABLICA DWUWYMIAROWA"<<endl;
8  tab2=new int*[2];
9  for(int i=0;i<2;i++){
10     tab2[i]=new int[3];}
11
12 for(int i=0;i<2;i++){
13     for(int j=0;j<3;j++){
14         tab2[i][j]=i+j;} }
15
16 for(int i=0;i<2;i++){
17     delete [] tab2[i];}
18 delete []tab2;
19 delete []tab1;
```

Listing 1.4. Tablica dynamiczna jedno i dwuwymiarowa

```
1  cout<<"TABLICA JEDNOWYMIAROWA"<<endl;
2  for(int j=0;j<3;j++){
3      cout<<tab1[j]<<" ";}
4
5  cout<<"TABLICA DWUWYMIAROWA"<<endl;
6  for(int i=0;i<2;i++){
7      for(int j=0;j<3;j++){
8          cout<<tab2[i][j]<<" ";}
9      cout<<endl;}
```

Listing 1.5. Wyświetlenie wszystkich elementów tablicy jedno i dwuwymiarowej

Powtórzenie wiadomości o tworzeniu funkcji:

Nagłówek funkcji składa się z następujących elementów:

typf funkcja(typ arg1, typ arg2, ..., typ argn), gdzie:

- typf – określa, jaki typ danych zostanie zwrócony przez funkcję, np. int, float, double, int*,
- **funkcja** – jest nazwą własną definiowanej funkcji,
- typ arg1, typ arg2, ..., typ argn – są to argumenty przekazywane do funkcji, należy określić typ każdego z argumentów.

Wyróżniamy dwa sposoby definiowania funkcji:

- stworzenie prototypu funkcji przed funkcją **main** oraz napisanie definicji funkcji po **main**,
- napisanie definicji funkcji przed **main**.

Poniższy listing przedstawia zarówno pierwsze, jak i drugie, podejście do tworzenia funkcji.

Listing 1.6 zawiera:

- linijka 3 – prototyp funkcji **dodawanie**,
- linijki 4,5 – deklaracja funkcji **odejmowanie**,
- linijka 8 – wywołanie funkcji **dodawanie**,
- linijka 11 – wywołanie funkcji **odejmowanie**,
- linijka 14 – deklaracja funkcji **dodawanie**.

```
1  #include <iostream>
2  using namespace std;
3  int dodawanie(int a, int b);
4  int odejmowanie ( int a, int b){
5      return a-b; }
6  int main(){
7      int wynik;
8      wynik=dodawanie(1,3);
9      cout<<"Wynik dodawania to: "<<wynik<<endl;
10
11     wynik=odejmowanie(1,3);
12     cout<<"Wynik odejmowania to: "<<wynik<<endl;
13     return 0; }
14 int dodawanie(int a, int b){
15     return a+b; }
```

Listing 1.6. Tworzenie własnych funkcji

Można także zadeklarować funkcję, która nie zwraca żadnej wartości i jest wtedy typu void. Listing 1.7 przedstawia tworzenie oraz wywołanie takiej funkcji.

```
1  #include <iostream>
2  using namespace std;
3  void wyswietlenie_napisu(string napis){
4      cout<<napis<<endl; }
5  int main(){
6      wyswietlenie_napisu("Ala ma kota");
7      return 0; }
```

Listing 1.7. Tworzenie funkcji typu void

Listing 1.8 przedstawia różne sposoby modyfikacji argumentów przekazanych do funkcji. Na rys. 1.5 przedstawiony został zrzut ekranu kodu wywołanego z listingu 1.8. Należy pamiętać, że jeśli chce się trwale zmienić wartość zmiennej w funkcji to jako argument należy przekazać albo wskaźnik na tą zmienną albo referencję do zmiennej.

```
1  #include <iostream>
2  using namespace std;
3  void funkcja1(int *c){
4      *c=6; }
```



```
5 void funkcja0(int f){
6     f=8;}
7 void funkcja2(int &f){
8     f=7;}
9 int main(){
10     int f;
11     f=5;
12     cout<<"f - przed wywołaniem funkcja0: "<<f<<endl;
13     funkcja0(f);
14     cout<<"f - po wywołaniu funkcja0: "<<f<<endl<<endl;
15
16     int *c=new int;
17     *c=9;
18     cout<<"c - przed wywołaniem funkcja1: "<<*c<<endl;
19     funkcja1(c);
20     cout<<"c - po wywołaniu funkcja1: "<<*c<<endl<<endl;
21
22     cout<<"f - przed wywołaniem funkcja2: "<<f<<endl;
23     funkcja2(f);
24     cout<<"f - po wywołaniu funkcja2: "<<f<<endl<<endl; }
```

Listing 1.8. Przekazywanie parametrów oraz ich modyfikacja w funkcji typu void

```
f -przed wywołaniem funkcja0: 5
f - po wywołaniu funkcja0: 5

c - przed wywołaniem funkcja1: 9
c - po wywołaniu funkcja1: 6

f - przed wywołaniem funkcja2: 5
f - po wywołaniu funkcja2: 7
```

Rys. 1.5. Wynik działania programu z listingu 1.8

Listing 1.9 zawiera kod umożliwiający przydzielenie pamięci oraz wypełnienie tablicy jedno oraz dwuwymiarowej w funkcji. Natomiast listing 1.10 zawiera częsty BŁĄD – tak NIE NALEŻY postępować. Usunięcie & z nagłówka funkcji spowoduje błąd pamięci w programie.

```
1 #include <iostream>
2 using namespace std;
3 void lokowaniePamieci2(int *&tab){
4     tab=new int[3];
5     tab[0]=1;
6     tab[1]=2;
7     tab[2]=3; }
8 void lokowaniePamieci3(int **&tab){
9     tab=new int*[2];
10    for(int i=0;i<2;i++){
11        tab[i]=new int[3]; }
```



```
12 for(int i=0;i<2;i++){
13     for(int j=0;j<3;j++){
14         tab[i][j]=i+j;
15         cout<<tab[i][j]<<" "; }
16     cout<<endl; } }
17 int main(){
18     int* tab1=nullptr;
19     cout<<"TABLICA JEDNOWYMIAROWA"<<endl;
20     lokowaniePamieci2(tab1);
21     delete []tab1;
22
23     cout<<"TABLICA DWUWYMIAROWA"<<endl;
24     int **tab3=nullptr;
25     lokowaniePamieci3(tab3);
26     for(int i=0;i<2;i++){
27         delete [] tab3[i]; }
28     delete []tab3;
29     return 0; }
```

Listing 1.9. Poprawne alokowanie pamięci w tablicy w funkcji

```
1 void alokowaniePamieci2(int *tab){
2     tab=new int[3];
3     tab[0]=1;
4     tab[1]=2;
5     tab[2]=3; }
```

Listing 1.10. Niepoprawne alokowanie pamięci w tablicy w funkcji

Wprowadzenie do algorytmiki:

Algorytm jest skończonym ciągiem zdefiniowanych czynności koniecznych do wykonania zadania lub sposobem postępowania prowadzącym do rozwiązania problemu.

Sposoby przedstawienia algorytmu:

- opis słowny,
- lista kroków (pseudokod),
- postać graficzna (schemat blokowy, schematy zwarte Nassi-Schneidermana NS),
- program.

Przykłady algorytmów:

- sortowanie bąbelkowe, sortowanie przez wstawianie,
- wyszukiwanie liczby w tablicy,
- podział zbioru na dwa rozdzielne podzbiory,
- algorytm Euklidesa,
- znajdowanie elementu maksymalnego lub minimalnego w zbiorze.

Generowanie liczb pseudolosowych:

Na listingu 1.11 przedstawione zostały niezbędne biblioteki oraz fragment kodu, który umożliwia wygenerowanie pseudolosowych liczb całkowitych o rozkładzie jednostajnym z danego przedziału $\langle a, b \rangle$

```
1 //Biblioteki:
2 #include <cstdlib>
3 #include <time.h>
4 //Ustawienie wartości początkowej generatora
5 srand(time(NULL));
6 //Generacja liczb całkowitych z zakresu od 0 do n-1:
7 int liczba1=rand()%n;
8 //Generacja liczb całkowitych z zakresu od a do b:
9 int liczba2=rand()%(b-a+1)+a;
```

Listing 1.11. Generowanie liczb pseudolosowych

Zadania do wykonania:

Zadanie 1.1. Funkcje typu void

Stwórz funkcje o poniższym prototypie:

- **void przydzielPamiec1D(int *&tab, int n)** – przydzielenie pamięci w tablicy jednowymiarowej, n – rozmiar tablicy,
- **void przydzielPamiec2D(int **&tab, int w, int k)** – przydzielenie pamięci w tablicy dwuwymiarowej, w, k - wymiary tablicy,
- **void wypelnijTablice1D(int *tab, int n, int a, int b)** –wypełnienie tablicy jednowymiarowej wygenerowanymi liczbami z zakresu $\langle a, b \rangle$,
- **void wypelnijTablice2D(int **tab, int w, int k, int a, int b)** - wypełnienie tablicy dwuwymiarowej wygenerowanymi liczbami z zakresu $\langle a, b \rangle$,
- **void usunTablice1D(int *&tab)** – zwolnienie pamięci w tablicy jednowymiarowej,
- **void usunTablice2D(int **&tab, int w)** – zwolnienie pamięci w tablicy dwuwymiarowej,
- **void wyswietl1D(int* tab, int n)** – wyświetlenie zawartości tablicy jednowymiarowej,
- **void wyswietl2D(int** tab, int w, int k)** – wyświetlenie zawartości tablicy dwuwymiarowej,

Na podstawie powyższych funkcji wykonaj poniższe zadania. Każde z zadań powinno zostać zrealizowane w oddzielnej funkcji, w której powinny zostać wywołane inne potrzebne funkcje.

Należy stworzyć jeden program, który będzie zawierał menu wielokrotnego wyboru:

- 1) zadanie 1.2
- 2) zadanie 1.3
- 3) zadanie 1.4
- 4) zadanie 1.5
- 5) wyjście z programu

Zadanie 1.2. Minimalny element

Pobierz od użytkownika rozmiar tablicy jednowymiarowej oraz przedział $\langle a, b \rangle$, z którego wylosowana zostanie zawartość tablicy. Wypełnij tablicę liczbami. Znajdź najmniejszy element w tablicy i sprawdź, czy jest on liczbą pierwszą. Na konsoli powinien zostać wyświetlony najmniejszy element oraz informacja czy jest to liczba pierwsza.

Zadanie 1.3. Zliczanie elementów w tablicy

Pobierz od użytkownika rozmiar tablicy jednowymiarowej. Przyjmij, że przedział, z którego będą losowane liczby w celu wypełnienia tablicy to $\langle 0, 9 \rangle$. Wypełnij tablicę liczbami. Należy zliczyć występowanie każdej z cyfr w tablicy i wyświetlić użytkownikowi informację, ile razy dana cyfra pojawiła się w tablicy, np. Cyfra 1 – 10 razy, itd.

Zadanie 1.4. Maksymalny element

Pobierz od użytkownika rozmiary tablicy dwuwymiarowej oraz przedział $\langle a, b \rangle$, z którego wylosowana zostanie zawartość tablicy. Wypełnij tablicę liczbami. Znajdź element maksymalny w tablicy i policz sumę cyfr znalezionej liczby. Na konsoli powinien zostać wyświetlony największy element oraz suma cyfr znalezionej liczby.

Zadanie 1.5. Średnia

Pobierz od użytkownika rozmiar kwadratowej tablicy dwuwymiarowej. Przyjmij, że przedział, z którego będą losowane liczby w celu wypełnienia tablicy to $\langle 7, 122 \rangle$. Wypełnij tablicę liczbami. Policz średnią arytmetyczną elementów znajdujących się nad główną przekątną oraz średnią arytmetyczną elementów znajdujących się pod główną przekątną. Wyświetl policzone średnie na konsoli.

LABORATORIUM 2. ALGORYTMY SORTOWANIA ELEMENTÓW W TABLICY: BUBLESORT, INSERTSORT, SELECTSORT

Cel laboratorium:

Omówienie podstawowych algorytmów sortowania elementów w tablicy.

Zakres tematyczny zajęć:

- sortowanie BubbleSort – sortowanie bąbelkowe,
- sortowanie InsertSort – sortowanie przez wstawianie,
- sortowanie SelectSort – sortowanie przez wybór.

Pytania kontrolne:

- 1) Na czym polega sortowanie bąbelkowe?
- 2) Jaka jest złożoność czasowa algorytmu sortowania bąbelkowego?
- 3) Na czym polega sortowanie przez wstawianie?
- 4) Jaka jest złożoność czasowa algorytmu sortowania przez wstawianie ?
- 5) Na czym polega sortowanie przez wybór?
- 6) Jaka jest złożoność czasowa algorytmu sortowania przez wybór?

Wprowadzenie:

Sortowanie polega na uporządkowaniu zbioru danych względem pewnych cech charakterystycznych każdego elementu tego zbioru. Złożoność jest to ilość zasobów potrzebnych na wykonanie algorytmu. Ze względu na rodzaj zasobu wyróżniamy złożoność czasową oraz pamięciową. Ze względu na sposób oszacowania można podzielić złożoność na pesymistyczną, optymistyczną lub średnią.

Podczas laboratorium przedstawione zostaną trzy podstawowe sortowania:

- BubbleSort – sortowanie bąbelkowe,
- InsertSort – sortowanie przez wstawianie,
- SelectSort – sortowanie przez wybór.

BubbleSort:

Algorytm sortowania bąbelkowego występuje w kilku wersjach. Na potrzeby laboratorium zaprezentowane zostaną dwie wersje tego algorytmu.

Poniżej znajdują się dwie wersje sortowania rosnąco ciągu liczb: 8 4 2 5 1. W pierwszej kolumnie na czerwono zostały zaznaczone aktualnie porównywane liczby. Druga kolumna przedstawia porównywane liczby po ewentualnej zamianie miejscami. Na zielono zostały zaznaczone elementy już posortowanej części tablicy. Linia przerywaną zostały oddzielone kolejne kroki. W kolejnym kroku wykonywana jest jedna operacja mniej w stosunku do kroku poprzedniego. Ideą algorytmu jest porównywanie dwóch sąsiednich liczb między sobą.

Wersja I

Złożoność czasowa:

- Optymistyczna: $O(n^2)$
- Średnia: $O(n^2)$
- Pesymistyczna: $O(n^2)$

W każdym kroku jedna liczba jest ustawiona w odpowiedniej kolejności.

Przykład:

8 4 2 5 1 -> 4 8 2 5 1

4 8 2 5 1 -> 4 2 8 5 1

4 2 8 5 1 -> 4 2 5 8 1

4 2 5 8 1 -> 4 2 5 1 8

4 2 5 1 8 -> 2 4 5 1 8

2 4 5 1 8 -> 2 4 5 1 8

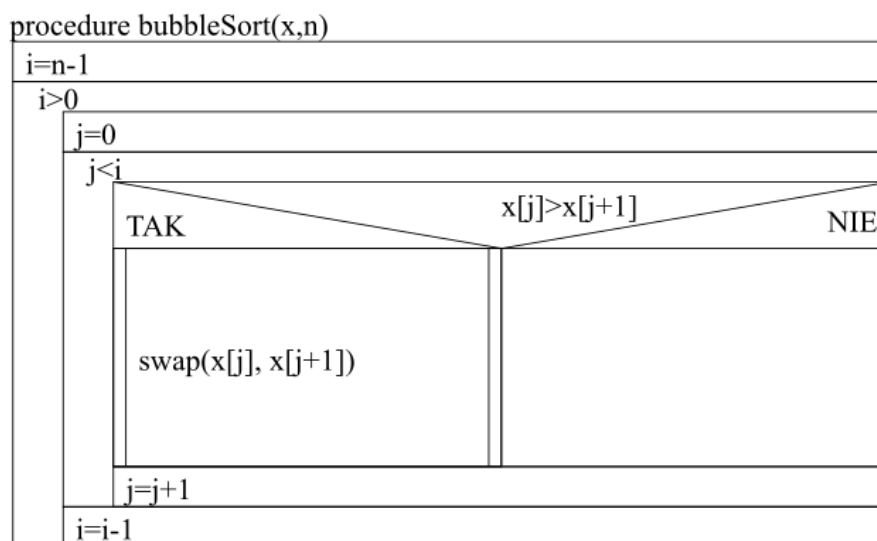
2 4 5 1 8 -> 2 4 1 5 8

2 4 1 5 8 -> 2 4 1 5 8

2 4 1 5 8 -> 2 1 4 5 8

2 1 4 5 8 -> 1 2 4 5 8 ciąg posortowany

Rysunek 2.1 przedstawia schemat zwarty NS omawianego algorytmu:



Rys. 2.1. Schemat zwarty NS – sortowanie bąbelkowe rosnąco wersja I

Wersja II

Złożoność czasowa:

- Optymistyczna: $O(n^2)$
- Średnia: $O(n^2)$
- Pesymistyczna: $O(n^2)$



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



W każdym kroku minimum jedna liczba jest ustawiona w odpowiedniej kolejności. Jeśli w danym kroku nie wystąpiła żadna zamiana elementów, oznacza to, że zbiór jest już posortowany i nie ma potrzeby wykonywać kolejnego kroku.

Przykład:

4 2 1 5 8 -> 2 4 1 5 8

2 4 1 5 8 -> 2 1 4 5 8

2 1 4 5 8 -> 2 1 4 5 8

2 1 4 5 8 -> 2 1 4 5 8

2 1 4 5 8 -> 1 2 4 5 8

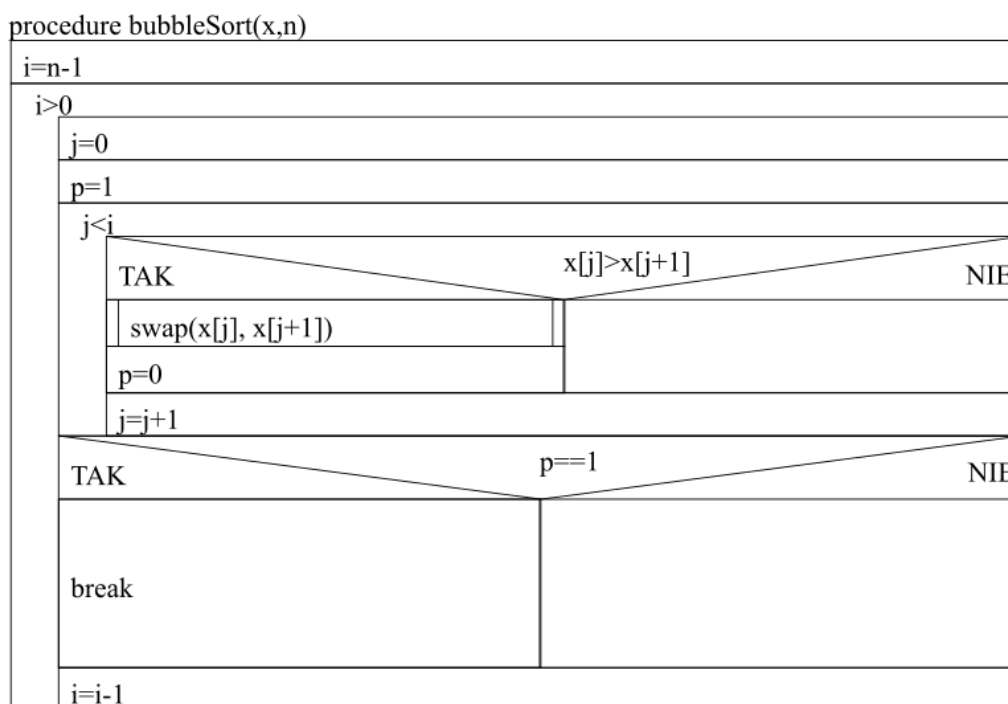
1 2 4 5 8 -> 1 2 4 5 8

1 2 4 5 8 -> 1 2 4 5 8

1 2 4 5 8 -> 1 2 4 5 8

1 2 4 5 8 -> 1 2 4 5 8 ciąg posortowany, brak zmiany

Rys. 2.2 przedstawia schemat zwarty NS zmodyfikowanego algorytmu sortowania bąbelkowego:



Rys. 2.2. Schemat zwarty NS – sortowanie bąbelkowe rosnąco wersja II

SelectionSort:

Złożoność czasowa:

- Optymistyczna: $O(n^2)$
- Średnia: $O(n^2)$
- Pesymistyczna: $O(n^2)$



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Poniżej znajduje się przykład sortowania liczb rosnąco: 3 4 2 5 1. W pierwszej kolumnie na czerwono została zaznaczona liczba, którą należy wstawić w odpowiednie miejsce. W drugiej kolumnie na zielono zaznaczone są liczby które zostały już posortowane, natomiast na czerwono została zaznaczona liczba, która została zamieniona z liczbą zaznaczoną na czerwono z pierwszej kolumny.

Linia przerywaną zostały oddzielone kolejne kroki od siebie. W kolejnym kroku wykonywana jest jedna operacja mniej w stosunku do kroku poprzedniego. W każdym kroku jedna liczba jest ustawiona w odpowiedniej kolejności. Ideą algorytmu jest wyszukiwanie elementu najmniejszego lub największego w dostępnej części tablicy.

Przykład:

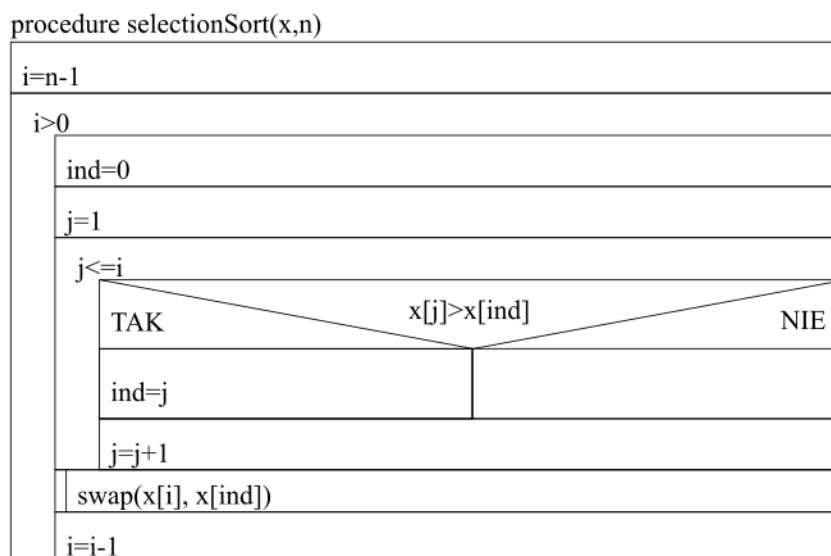
3 4 2 5 1 -> 1 4 2 5 3

1 4 2 5 3 -> 1 2 4 5 3

1 2 4 5 3 -> 1 2 3 5 4

1 2 3 5 4 -> 1 2 3 4 5 ciąg posortowany

Rys 2.3 przedstawia schemat zwarty NS algorytmu sortowania przez wybór:



Rys. 2.3. Schemat zwarty NS – sortowanie przez wybór

InsertSort:

Złożoność czasowa:

- Optymistyczna: $O(n)$
- Średnia: $O(n^2)$
- Pesymistyczna: $O(n^2)$

Poniżej przedstawiony jest przykład sortowania liczb rosnąco: 8 4 2 5 1. Linia przerywaną zostały oddzielone kolejne kroki od siebie. W kolejnym kroku wykonywana jest jedna operacja porównania więcej w stosunku do poprzedniego kroku. Na czerwono zostały

zaznaczone aktualnie porównywane liczb. Ukośnikiem / zostały oddzielone kolejne porównania.

Przykład:

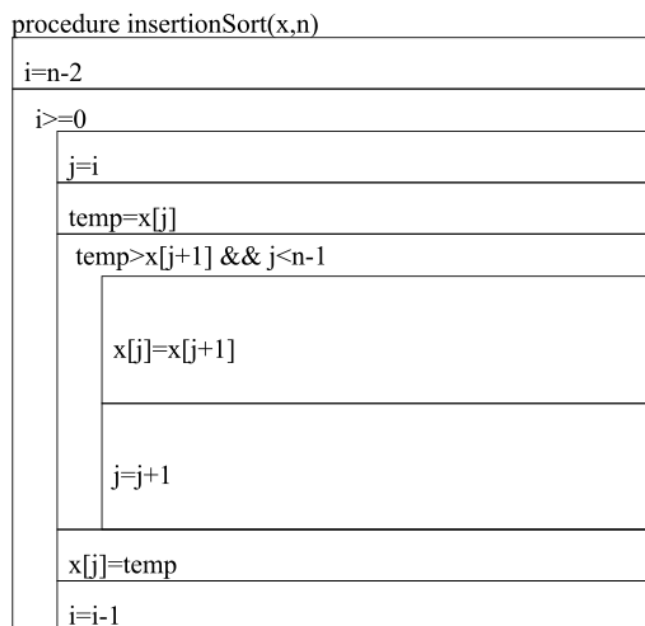
8 4 2 **5** 1 -> 8 4 2 **1** 5

8 4 **2** 1 5 -> 8 4 **1** 2 5 / 8 4 1 **2** 5 -> 8 4 1 **2** 5

8 **4** 1 2 5 -> 8 **1** 4 2 5 / 8 1 **4** 2 5 -> 8 1 **2** 4 5 / 8 1 2 **4** 5 -> 8 1 2 4 5

8 1 2 4 5 -> **1** 8 2 4 5 / **1** **8** 2 4 5 -> 1 **2** 8 4 5 / 1 2 **8** 4 5 -> 1 2 **4** 8 5 / 1 2 4 **8** 5 -> 1 2 4 **5** 8

Rys 2.3 przedstawia schemat zwarty NS algorytmu sortowania przez wstawianie:



Rys. 2.4. Schemat zwarty NS – sortowanie przez wstawianie

Zadania do wykonania:

Zadanie 2.1. Funkcje

Stwórz funkcje o poniższym prototypie:

- **void sortowanieBabelkowe(int* tab, int n, int tryb)** – sortowanie bąbelkowe tablicy jednowymiarowej malejąco/rosnąco w zależności od wartości zmiennej *tryb*,
- **void sortowaniePrzezWybor(int* tab, int n, int tryb)** – sortowanie przez wybór tablicy jednowymiarowej malejąco/rosnąco w zależności od wartości zmiennej *tryb*,
- **void sortowaniePrzezWstawianie(int* tab, int n, int tryb)** – sortowanie przez wstawianie tablicy jednowymiarowej malejąco/rosnąco w zależności od wartości zmiennej *tryb*,
- **void sortowanieBabelkowe2D(int** tab, int w, int k, int tryb, int nrKol)** - sortowanie bąbelkowe tablicy dwuwymiarowej malejąco/rosnąco w zależności od wartości zmiennej *tryb* względem kolumny *nrKol*.



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Na podstawie powyższych funkcji, wykonaj poniższe zadania. Do wykonania zadań należy skorzystać z funkcji zaimplementowanych podczas laboratoriów nr 1. Każde z zadań powinno zostać zrealizowane w oddzielnej funkcji, w której powinny zostać wywołane inne potrzebne funkcje.

Należy stworzyć jeden program, który będzie zawierał menu wielokrotnego wyboru:

- 1) zadanie 2.2
- 2) zadanie 2.3
- 3) zadanie 2.4
- 4) zadanie 2.5
- 5) wyjście z programu

Zadanie 2.2. Sortowanie bąbelkowe

Napisz funkcję, która:

- pobierze od użytkownika rozmiar tablicy, przedział $\langle a, b \rangle$, z którego będą losowane liczby w celu wypełnienia tablicy oraz tryb sortowania (rosnąco/malejąco);
- przydzieli pamięć,
- wypełni tablicę losowymi liczbami,
- wyświetli zawartość tablicy przed sortowaniem,
- posortuje algorytmem sortowania bąbelkowego dane rosnąco lub malejąco w zależności od wartości zmiennej tryb,
- wyświetli zawartość tablicy po sortowaniu,
- zwolni pamięć.

Zadanie 2.3. Sortowanie przez wybór

Napisz funkcję, która:

- pobierze od użytkownika rozmiar tablicy, przedział $\langle a, b \rangle$, z którego będą losowane liczby w celu wypełnienia tablicy oraz tryb sortowania (rosnąco/malejąco),
- przydzieli pamięć,
- wypełni tablicę losowymi liczbami,
- wyświetli zawartość tablicy przed sortowaniem,
- posortuje algorytmem sortowania przez wybór dane rosnąco lub malejąco w zależności od wartości zmiennej tryb,
- wyświetli zawartość tablicy po sortowaniu,
- zwolni pamięć.

Zadanie 2.4. Sortowanie przez wstawianie

Napisz funkcję, która:

- pobierze od użytkownika rozmiar tablicy, przedział $\langle a, b \rangle$, z którego będą losowane liczby w celu wypełnienia tablicy oraz tryb sortowania (rosnąco/malejąco),
- przydzieli pamięć,
- wypełni tablicę losowymi liczbami,
- wyświetli zawartość tablicy przed sortowaniem,
- posortuje algorytmem sortowania przez wstawianie dane rosnąco lub malejąco w zależności od wartości zmiennej tryb,



- wyświetli zawartość tablicy po sortowaniu,
- zwolni pamięć.

Zadanie 2.5. Sortowanie tablicy dwuwymiarowej

Napisz funkcję, która:

- pobierze od użytkownika rozmiary tablicy dwuwymiarowej, przedział $\langle a, b \rangle$, z którego będą losowane liczby w celu wypełnienia tablicy, tryb sortowania (rosnąco/malejąco) oraz numer kolumny względem której odbędzie się sortowanie,
- przydzieli pamięć,
- wypełni tablicę losowymi liczbami,
- wyświetli zawartość tablicy przed sortowaniem,
- posortuje algorytmem sortowania przez wstawianie dane rosnąco lub malejąco w zależności od wartości zmiennej tryb względem podanej kolumny,
- wyświetli zawartość tablicy po sortowaniu,
- zwolni pamięć.

Przykładowe dane do zadania

-1	3	9	1
10	19	-3	0
20	34	-7	4
5	7	2	9
-66	4	88	5

Sortowanie danych względem pierwszej kolumny malejąco (numeracja od 0)

20	34	-7	4
10	19	-3	0
5	7	2	9
-66	4	88	5
-1	3	9	1



LABORATORIUM 3. ALGORYTM SORTOWANIA QUICKSORT

Cel laboratorium:

Omówienie algorytmu sortowania QuickSort.

Zakres tematyczny zajęć:

- algorytm sortowania Quicksort,
- powtórzenie wiadomości o strukturach,
- wczytanie danych z pliku tekstowego.

Pytania kontrolne:

- 1) W jaki sposób działa algorytm sortowania QuickSort?
- 2) Jaka jest złożoność czasowa algorytmu sortowania QuickSort?
- 3) W jaki sposób stworzyć strukturę?
- 4) W jaki sposób odwoływać się do elementów struktury?
- 5) W jaki sposób odczytać dane z pliku?

QuickSort:

Jest to algorytm działający na zasadzie algorytmu „dziel i zwyciężaj”. Z tablicy wybierany jest element rozdzielający, następnie tablica jest dzielona na dwa fragmenty. Pierwsza część tablicy składa się z elementów mniejszych niż element rozdzielający. Natomiast druga część tablicy zawiera elementy większe bądź równe od elementu rozdzielającego dwie części tablicy. Algorytm kończy się, gdy fragment (tablica) uzyskany z podziału zawiera pojedynczy element, jako że jednoelementowa tablica nie wymaga sortowania.

Złożoność czasowa:

- Optymistyczna: $O(n \log(n))$
- Średnia: $O(n \log(n))$
- Pesymistyczna: $O(n^2)$

Poniżej przedstawiony jest przykład sortowania liczb rosnąco: 5 1 4 2 8 3. Wprowadzone zostały poniższe oznaczenia:

- *sr* – indeks, pod którym znajduje się piwot,
- *piwot* – element podziałowy,
- *p* – indeks pierwszego elementu do podziału,
- *l* – indeks ostatniego elementu do podziału,
- *g* – indeks pierwszego elementu drugiego podzbioru (do momentu ustawienia *piwota* na odpowiednie miejsce).

Obszar tablicy od lewej strony do pionowej czerwonej kreski to obszar, z którego należy porównać liczby z *piwotem*.

Na czerwono zostały zaznaczone zmiany. Jeśli *piwot* jest większy od porównywanej liczby to element, który znajduje się pod indeksem *g*, jest zamieniany ze sprawdzanym elementem i wartość *g* jest zwiększana o 1.

Przykład:

indeks	0	1	2	3	4	5
wartość	5	1	4	2	8	3

$sr=(0+5)/2=2, l=0, p=5, g=0, \text{piwot}=\text{tab}[2]=4, \text{tab}[2]=\text{tab}[p]=3$

indeks	0	1	2	3	4	5
wartość	5	1	3	2	8	3

$5 < \text{piwot}$, nie

$1 < \text{piwot}$, tak, $g=1$

indeks	0	1	2	3	4	5
wartość	1	5	3	2	8	3

$3 < \text{piwot}$, tak, $g=2$

indeks	0	1	2	3	4	5
wartość	1	3	5	2	8	3

$2 < \text{piwot}$, tak, $g=3$

indeks	0	1	2	3	4	5
wartość	1	3	2	5	8	3

$8 < \text{piwot}$, nie

Ostatnia liczba została porównana z *piwotem*. Należy pod indeksem *g* zapisać *piwota*, a pod indeksem *p* liczbę, która była zapisana pod indeksem *g*.

$g=3, \text{piwot}=4, p=5$

indeks	0	1	2	3	4	5
wartość	1	3	2	4	8	5

Po wykonaniu pierwszego kroku, *piwot* równy 4 podzielił tablicę na dwie tablice. Pierwsza tablica zawiera elementy mniejsze bądź równe wartości *piwota*, natomiast druga tablica zawiera elementy większe od wartości *piwota*. Każdą z tablic należy oddzielnie posortować.

Sortowanie pierwszej tablicy:

indeks	0	1	2
wartość	1	3	2

$sr=1, l=0, p=2, g=0, \text{piwot}=\text{tab}[1]=3, \text{tab}[sr] = \text{tab}[p] = 2$

indeks	0	1	2
wartość	1	2	2

$1 < \text{piwot}$, tak, $g=1$

Indeks	0	1	2
Wartość	1	2	2

$2 < \text{piwot}$, tak, $g=2$

indeks	0	1	2
wartość	1	2	2



Ostatnia liczba została porównana z piwotem. Należy pod indeksem g zapisać *piwota*, a pod indeksem p liczbę, która była zapisana pod indeksem g .

$g=2$

indeks	0	1	2
wartość	1	2	3

Tablica została podzielona na elementy mniejsze bądź równe 3 oraz większe od 3. Tablicę z elementami mniejszymi bądź równymi 3 należy posortować.

indeks	0	1
wartość	1	2

$sr=0, l=0, p=1, g=0, \text{piwot}=\text{tab}[0]=1, \text{tab}[sr] = \text{tab}[p] = 2$

indeks	0	1
wartość	2	2

$2 < \text{piwot}$, nie

Ostatnia liczba została porównana z *piwotem*. Należy pod indeksem g zapisać *piwota*, a pod indeksem p liczbę, która była zapisana pod indeksem g .

$g=0$

indeks	0	1
wartość	1	2

Druga tablica:

indeks	4	5
wartość	8	5

$sr=4, l=4, p=5, g=4, \text{piwot}=\text{tab}[4]=8, \text{tab}[sr] = \text{tab}[p] = 5$

indeks	4	5
wartość	5	5

$5 < \text{piwot}$, tak, $g=5$

indeks	4	5
wartość	5	5

Ostatnia liczba została porównana z *piwotem*. Należy pod indeksem g zapisać *piwota*, a pod indeksem p liczbę, która była zapisana pod indeksem g .

$g=5$

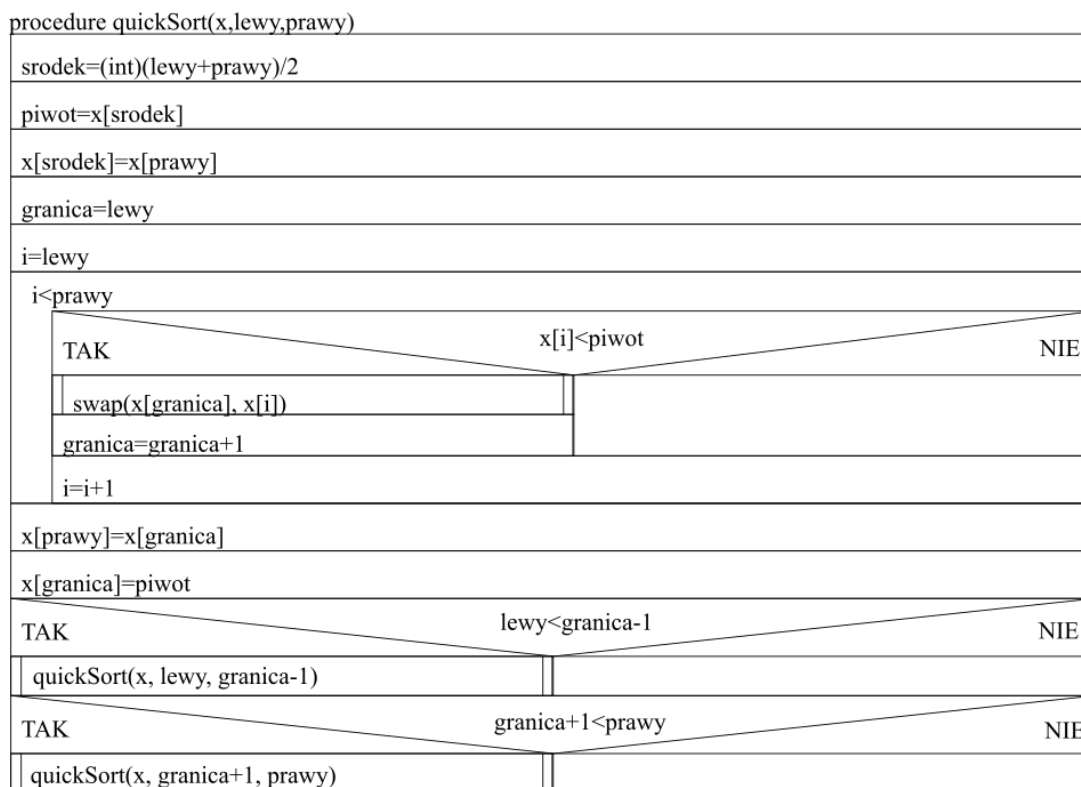
indeks	4	5
wartość	5	8

Posortowana tablica

indeks	0	1	2	3	4	5
wartość	1	2	3	4	5	8



Rys 3.1 przedstawia schemat zwarty NS algorytmu sortowania szybkiego:



Rys. 3.1. Schemat zwarty NS – szybkie sortowanie

Powtórzenie o strukturach:

Poniżej znajduje się przypomnienie wiadomości dotyczących struktur. Listing 3.1 przedstawia definicję struktury. Strukturę należy stworzyć przed funkcją **main**. Struktura posiada nazwę własną, w tym przypadku *punkt*. W strukturze przechowywane będą współrzędne punktu płaszczyzny: *x* i *y*.

```
1 struct punkt{
2   int x;
3   int y; };
```

Listing 3.1. Definicja struktury

Na listingach 3.2, 3.3, 3.4, 3.5 oraz 3.6 przedstawione zostały różne sposoby definiowania zmiennej strukturalnej wraz z odpowiadającym jej odwołaniem do elementów struktury. Listing 3.2 zawiera definicję zmiennej statycznej. Odwołanie do elementów struktury odbywa się poprzez użycie kropki.

```
1 punkt tab1[3];
2 tab1[0].x=1;
3 tab1[0].y=2;
4 cout<<"x: "<<tab1[0].x<<"y: "<<tab1[0].y<<endl;
```

Listing 3.2. Definicja i sposób odwołania się do zmiennej strukturalnej statycznej



Listing 3.3 zawiera definicję tablicy statycznej struktur. Odwołanie do elementów struktury odbywa się poprzez użycie kropki.

```
1 punkt* p2=new punkt;  
2 p2->x=2;  
3 p2->y=3;  
4 cout<<"x: "<<p2->x<<"y: "<<p2->y<<endl;
```

Listing 3.3. Definicja i sposób odwołania się tablicy statycznej struktur

Listing 3.4 zawiera definicję zmiennej wskaźnikowej. Odwołanie do elementów struktury odbywa się poprzez użycie strzałki.

```
1 punkt* p2=new punkt;  
2 p2->x=2;  
3 p2->y=3;  
4 cout<<"x: "<<p2->x<<"y: "<<p2->y<<endl;
```

Listing 3.4. Definicja i sposób odwołania się do zmiennej strukturalnej wskaźnikowej

Listing 3.5 zawiera definicję wskaźnika na tablicę, której elementami są struktury statyczne. Odwołanie do elementów struktury odbywa się poprzez użycie kropki.

```
1 punkt* tab2;  
2 tab2=new punkt[3];  
3 tab2[0].x=1;  
4 tab2[0].y=2;  
5 cout<<"x: "<<tab2[0].x<<"y: "<<tab2[0].y<<endl;
```

Listing 3.5. Definicja i sposób odwołania się do wskaźnika na tablicę, której elementami są struktury statyczne

Listing 3.6 zawiera definicję wskaźnika na tablicę, której elementami są wskaźniki na struktury. Odwołanie do elementów struktury odbywa się poprzez użycie strzałki.

```
1 punkt** tab3;  
2 tab3=new punkt*[3];  
3 for(int i=0;i<3;i++){  
4 tab3[i]=new punkt; }  
5 tab3[0]->x=1;  
6 tab3[0]->y=2;  
7 cout<<"x: "<<tab3[0]->x<<"y: "<<tab3[0]->y<<endl;
```

Listing 3.6. Definicja i sposób odwołania się do wskaźnika na tablicę, której elementami są wskaźniki na struktury



Zadania do wykonania:

Zadanie 3.1. Szybkie sortowanie

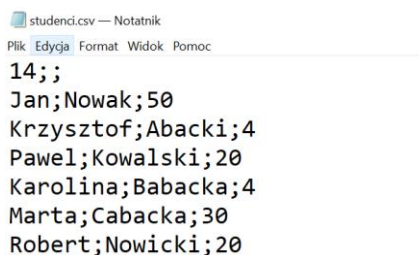
Napisz program, który:

- wczyta rozmiar tablicy z dołączonego pliku – „studenci.csv”,
- przydzieli pamięć,
- wczyta dane o studentach z dołączonego pliku,
- pobierze od użytkownika informację, w jakim trybie mają zostać posortowane punkty (rosnąco/malejąco),
- wyświetli zawartość tablicy przed sortowaniem,
- posortuje algorytmem sortowania szybkiego dane studentów względem uzyskanych, punktów rosnąco lub malejąco w zależności od wartości zmiennej *tryb*,
- wyświetli zawartość tablicy po sortowaniu,
- zwolni pamięć.

Zdefiniuj strukturę student, a następnie utwórz wskaźnik na tablicę studentów. W celu zaimplementowania zadania, należy stworzyć funkcje o poniższych prototypach:

- **void sortowanieQuickSort(student* tab, int n, int tryb)** – sortowanie tablicy jednowymiarowej rosnąco lub malejąco w zależności od wartości zmiennej *tryb*. Należy posortować dane studentów względem liczby punktów,
- **void wczytajStudentow(student*&tab, int n)** - przydzielenie pamięci na podstawie liczby studentów pobranej z pliku oraz wczytanie danych o studentach z pliku,
- **void usunTabliceStudentow(student *&tab)** – usunięcie tablicy studentów z pamięci,
- **void wyswietlStudentow(student* tab, int n)** – wyświetlenie wszystkich studentów.

Plik z danymi należy umieścić w katalogu projektu a nazwę pliku umieścić w kodzie programu. Na rys. 3.2 znajduje się zrzut ekranu pliku „studenci.csv”, który zawiera dane o studentach.



```
studenci.csv — Notatnik
Plik Edycja Format Widok Pomoc
14;;
Jan;Nowak;50
Krzysztof;Abacki;4
Pawel;Kowalski;20
Karolina;Babacka;4
Marta;Cabacka;30
Robert;Nowicki;20
```

Rys. 3.2. Plik *studenci.csv*

Pierwsza linijka pliku zawiera informację o liczbie studentów, których dane są przechowywane w pliku. W każdej kolejnej linijce znajduje się informacja o jednym studentie. Pierwsza kolumna zawiera imię studenta, druga kolumna zawiera nazwisko studenta, a w trzeciej kolumnie znajdują się punkty uzyskane przez studenta. Kolumny oddzielone są średnikami.

Poniżej, na listingu 3.7, został przedstawiony fragment kodu, który umożliwi odczytanie danych z pliku.

Listing 3.7. zawiera:

- linijka 3 – obiekt, który umożliwia odczyt z pliku,
- linijka 7 – otwarcie pliku w trybie do odczytu,
- linijka 8 – odczyt z pliku liczby studentów,
- linijki 11,12 – odczytanie średników znajdujących się w pierwszej linijce,
- linijki 14-29 – odczytywanie kolejnych linii wraz z podziałem na pola pomiędzy separatorem – średnikiem,
- linijka 30 – zamknięcie pliku.

```
1  string sciezka, linia;
2  int liczbaStudentow;
3  ifstream plik;
4  char sredniki;
5  student* tab;
6  sciezka="studenci.csv";

7  plik.open(sciezka);
8  plik >> liczbaStudentow;
9  //alokowanie pamieci w tab o dlugosci liczbaStudentow
10 //elementem tablicy jest struktura(imie, nazwisko, punkty)
11 for(int i = 0; i < 2; i++)
12     plik >> sredniki;
13
14 for(int i=0; i<liczbaStudentow; i++){
15     plik>>linia;
16     istream ss(linia);
17     getline(ss, tab[i].imie, ';');
18     getline(ss, tab[i].nazwisko, ';');
19     getline(ss, tab[i].punkty);}
20 plik.close();
```

Listing 3.7. Odczyt danych z pliku do tablicy



LABORATORIUM 4. ALGORYTMY PODZIAŁU ZBIORU NA DWIE ALBO TRZY CZĘŚCI

Cel laboratorium:

Omówienie algorytmu podziału zbioru na dwie lub trzy części.

Zakres tematyczny zajęć:

- algorytm podziału zbioru na dwie części,
- algorytm podziału zbioru na trzy części.

Pytania kontrolne:

- 1) W jaki sposób działa algorytm podziału zbioru na dwie części?
- 2) Jaka jest złożoność czasowa algorytmu podziału zbioru na dwie części?
- 3) W jaki sposób działa algorytm podziału zbioru na trzy części?
- 4) Jaka jest złożoność czasowa algorytmu podziału zbioru na trzy części?

Algorytm podziału zbioru na dwie części:

Algorytm polega na podziale zbioru na dwa rozłączne podzbiory. Pierwszy podzbiór zawiera elementy spełniające warunek podziału, drugi – niespełniające. Algorytm zwraca również indeks pierwszego elementu drugiego podzbioru. W celu wyjaśnienia algorytmu, wprowadźmy następujące oznaczenia:

- A – zbiór pierwotny,
- B – zbiór elementów spełniających warunek 1,
- C – zbiór elementów spełniających warunek 2, tj. $\text{warunek1} = \sim \text{warunek2}$
- $A = B \cup C$ – suma zbiorów daje zbiór pierwotny
- $B \cap C = \emptyset$ - iloczyn zbiorów jest zbiorem pustym stąd wiadomo, że zbiory są rozłączne.

Algorytm polega na poszukiwaniu elementu, który nie spełnia warunku1 z lewej strony tablicy oraz elementu, który nie spełnia warunku2 z prawej strony tablicy. Jeśli takie elementy zostaną znalezione, to następuje zamiana miejscami tych elementów, ponieważ, element, który nie spełniał warunku1 spełnia warunek2, i odwrotnie.

Przykład poprawnego sformułowania warunków - podział grupy studentów na podgrupy:

- pierwszą grupę stanowią osoby, które dostały zaliczenie ($\text{ocena} \geq 3$),
- drugą – osoby, które nie dostały zaliczenia ($\text{ocena} < 3$).

Zadanie zostało sformułowane poprawnie, gdyż podzbiory są wyraźnie rozłączne (albo zdał, albo nie zdał), natomiast połączenie tych podzbiorów daje cały zbiór.

Przykład niepoprawnego sformułowania warunków: Niech A jest zbiorem liczb całkowitych z przedziału $[1, 30]$. Podziel zbiór A na podzbiory B oraz C tak, że B będzie zawierał liczby podzielne przez 2, a C – podzielne przez 3. Oczywiście jest, że $B \cap C \neq \emptyset$ oraz $A \neq B \cup C$, ponieważ liczby podzielne przez 6 znajdują się zarówno w podzbiorze B jak i C .

Idea algorytmu podziału na dwa podzbiory jest bardzo podobna do wykonania jednej iteracji algorytmu QuickSort. Różnica polega na tym, że w przypadku sortowania szybkiego jako *piwot* (element, na podstawie którego zbiór jest dzielony) jest wybierany element

o indeksie $(lewy+prawy)/2$, a w przypadku algorytmu podziału zbioru na dwa podzbiory warunek podziału jest ustalany przez użytkownika/programistę.

Złożoność czasowa: $O(n)$

Poniżej przedstawiony jest przykład podziału zbioru na dwie rozłączne części - podział zbioru na liczby parzyste i nieparzyste. W pierwszej części tablicy będą znajdować się liczby parzyste a w drugiej części tablicy liczby nieparzyste. Wprowadźmy następujące oznaczenia:

- i – indeks odpowiadający za poszukiwanie elementów nieparzystych,
- j – indeks odpowiadający za poszukiwanie elementów parzystych.

Przykład:

numer indeksu	0	1	2	3	4	5	6	7
wartość	10	1	8	11	2	6	13	9

$i \rightarrow$

$\leftarrow j$

Poszukiwanie elementu nieparzystego:

$i=0, j=7$

$tab[0]\%2=0 \ \&\& \ 0<7, i=1, j=7$

$tab[1]\%2!=0 \ \&\& \ 1<7, i=1, j=7$

Poszukiwanie elementu parzystego:

$i=1, j=7$

$tab[7]\%2!=0 \ \&\& \ 1<7, i=1, j=6$

$tab[6]\%2!=0 \ \&\& \ 1<6, i=1, j=5$

$tab[5]\%2=0 \ \&\& \ 1<5, i=1, j=5$

Element nieparzysty został znaleziony po lewej stronie tablicy oraz element parzysty został znaleziony po prawej stronie tablicy więc następuje zamiana:

$1<5, swap(tab[1], tab[5]), i=2, j=4$

numer indeksu	0	1	2	3	4	5	6	7
wartość	10	6	8	11	2	1	13	9

Poszukiwanie elementu nieparzystego:

$i=2, j=4$

$tab[2]\%2=0 \ \&\& \ 2<4, i=3, j=4$

$tab[3]\%2!=0 \ \&\& \ 3<4, i=3, j=4$

Poszukiwanie elementu parzystego:

$i=3, j=4$

$tab[4]\%2!=0 \ \&\& \ 3<4, i=3, j=4$

Element nieparzysty został znaleziony po lewej stronie tablicy oraz element parzysty został znaleziony po prawej stronie tablicy więc następuje zamiana:



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



$3 < 4$, $\text{swap}(\text{tab}[3], \text{tab}[4])$, $i=4, j=3$

numer indeksu	0	1	2	3	4	5	6	7
wartość	10	6	8	2	11	1	13	9

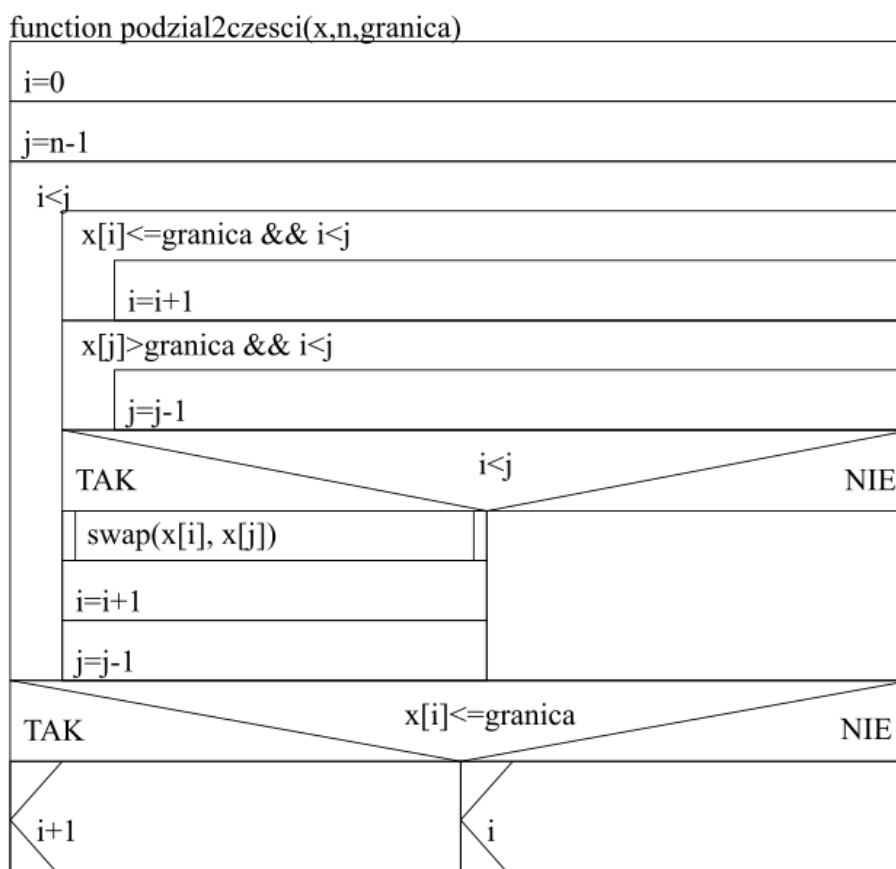
Zbiór został podzielony na dwa podzbiory. Należy ustalić, od którego indeksu zaczyna się drugi podzbiór. Jeśli element pod indeksem i znajduje się w pierwszym podzbiorze to pierwszym indeksem drugiego podzbioru będzie $i+1$.

W tym przypadku drugi podzbiór zaczyna się od indeksu 4. Na zielono zostały zaznaczone elementy należące do pierwszego podzbioru – liczby parzyste, a na pomarańczowo zostały zaznaczone elementy należące do drugiego podzbioru – liczby nieparzyste.

numer indeksu	0	1	2	3	4	5	6	7
wartość	10	6	8	2	11	1	13	9

Na rys. 4.1 przedstawiono Schemat zwarty NS algorytmu podziału zbioru na dwa podzbiory

Schemat zwarty NS zawiera warunek w postaci nierówności tj. element mniejszy bądź równy od granicy i element większy od granicy. Warunkiem niekoniecznie musi być nierówność może być to na przykład sprawdzenie podzielności liczb albo sprawdzenie czy jest to liczba pierwsza.



Rys. 4.1. Schemat zwarty NS algorytmu podziału zbioru na dwie części

Algorytm podziału zbioru na trzy części:

Algorytm polega na podziale zbioru na trzy podzbiory:

- 1 – podzbiór, którego elementy spełniają warunek1,
- 2 – podzbiór, którego elementy spełniają warunek2,
- 3 – podzbiór, którego elementy nie spełniają ani pierwszego, ani drugiego z warunków (spełniają warunek3)

W celu wyjaśnienia algorytmu, wprowadźmy następujące oznaczenia:

- A – zbiór wszystkich elementów,
- B – zbiór elementów spełniających warunek 1,
- C – zbiór elementów spełniających warunek 2,
- D – zbiór elementów nie spełniających warunków 1 oraz 2 (spełniające warunek 3),
- $A = B \cup C \cup D$
- $B \cap C = \emptyset$
- $A \cap C = \emptyset$
- $A \cap B = \emptyset$

Algorytm polega na poszukiwaniu elementów należących do skrajnych podzbiorów – pierwszego i trzeciego. Zbiór środkowy – drugi utworzy się automatycznie.

Poprawne sformułowanie zadania:

Podziel zbiór A zawierający liczby całkowite z przedziału [1,30] na trzy podzbiory:

- B - liczby podzielne przez 3,
- C - liczby podzielne przez 3 z resztą równą 1,
- D - liczby podzielne przez 3 z resztą równą 2

Jak widać, założenia są spełnione, gdyż podzbiory są parami rozłączne (reszta nie może być równocześnie równa dwóm liczbom – 1 i 2) oraz ich połączenie daje cały zbiór.

Niepoprawne sformułowanie zadania:

Podziel grupę studentów A na trzy podgrupy:

- B - studentów, którzy nie uzyskali zaliczenia,
- C - studentów którzy uzyskali zaliczenie,
- D - studentów, którzy uzyskali ocenę „Bardzo dobrą”.

Oczywiste jest, że podgrupy C oraz D mają część wspólną, gdyż osoby, które uzyskały ocenę „Bardzo dobrą”, należą również do podgrupy studentów, którzy uzyskali zaliczenie.

Złożoność czasowa: $O(n)$

Poniżej przedstawiony jest przykład podziału zbioru na trzy rozłączne części - podział zbioru na liczby podzielne przez 3, podzielne przez 3 z resztą równą 1 oraz podzielne przez 3 z resztą równą 2. W pierwszej części tablicy będą znajdować się liczby podzielne przez 3, następnie liczby podzielne przez 3 z resztą równą 1. W ostatniej części będą znajdować się liczby podzielne przez 3 z resztą równą 2.

Wprowadźmy następujące oznaczenia:

- i - indeks ostatniego elementu pierwszego podzbioru,
- k - indeks pierwszego elementu trzeciego podzbioru.



Przykład:

numer indeksu	0	1	2	3	4	5	6	7
wartość	10	1	8	11	2	6	13	9

$i=-1, j=0, k=8$

Poszukiwanie elementu podzielonego przez 3 lub podzielonego przez 3 z resztą 2.

$0 < 8$

$tab[0] \% 3 \neq 0, tab[0] \% 3 = 2, j=1, i=-1, k=8$

$1 < 8$

$tab[1] \% 3 \neq 0, tab[1] \% 3 = 2, j=2, i=-1, k=8$

$2 < 8$

$tab[2] \% 3 \neq 0, tab[2] \% 3 = 2, j=2, i=-1, k=7$

Znaleziony został element podzielony przez 3 z resztą 2, w związku z tym musi się on znaleźć w prawej części tablicy bo należy do trzeciego podzbioru. Znaleziony element (8) zamieniany jest z elementem, który znajduje się pod indeksem k (9). Nie wiadomo, do którego zbioru należy 9, w związku z tym, należy to sprawdzić w kolejnym kroku.

$swap(tab[7], tab[2])$

numer indeksu	0	1	2	3	4	5	6	7
wartość	10	1	9	11	2	6	13	8

Poszukiwanie elementu podzielonego przez 3 lub podzielonego przez 3 z resztą 2.

$2 < 7$

$tab[2] \% 3 = 0, j=2, i=0, k=7$

Znaleziony został element podzielony przez 3, w związku z tym musi się on znaleźć w lewej części tablicy bo należy do pierwszego podzbioru. Zamieniany jest z elementem, który znajduje się pod indeksem i . Nie ma potrzeby sprawdzać elementu, który został zamieniony (10) z wyszukany elementem (9), ponieważ ten element, został już wcześniej sprawdzony.

$swap(tab[0], tab[2])$

numer indeksu	0	1	2	3	4	5	6	7
wartość	9	1	10	11	2	6	13	8

$j=3$

Poszukiwanie elementu podzielonego przez 3 lub podzielonego przez 3 z resztą 2.

$3 < 7$

$tab[3] \% 3 \neq 0, tab[3] \% 3 = 2, j=3, i=0, k=6$

Znaleziony został element podzielony przez 3 z resztą 2, w związku z tym musi się on znaleźć w prawej części tablicy bo należy do trzeciego podzbioru. Znaleziony element (11)

zamieniany jest z elementem, który znajduje się pod indeksem k (13). Nie wiadomo, do którego zbioru należy 13, w związku z tym, należy to sprawdzić w kolejnym kroku.

swap(*tab*[3], *tab*[6])

numer indeksu	0	1	2	3	4	5	6	7
wartość	9	1	10	13	2	6	11	8

Poszukiwanie elementu podzielonego przez 3 lub podzielonego przez 3 z resztą 2.

$3 < 6$

$tab[3] \% 3 \neq 0$, $tab[3] \% 3 = 2$, $j=4$, $i=0$, $k=6$

$4 < 6$

$tab[4] \% 3 \neq 0$, $tab[4] \% 3 = 2$, $j=4$, $i=0$, $k=5$

Znaleziony został element podzielony przez 3 z resztą 2, w związku z tym musi się on znaleźć w prawej części tablicy bo należy do trzeciego podzbioru. Znaleziony element (2) zamieniany jest z elementem, który znajduje się pod indeksem k (6). Nie wiadomo, do którego zbioru należy 6, w związku z tym, należy to sprawdzić w kolejnym kroku.

swap(*tab*[4], *tab*[5])

numer indeksu	0	1	2	3	4	5	6	7
wartość	9	1	10	13	6	2	11	8

Poszukiwanie elementu podzielonego przez 3 lub podzielonego przez 3 z resztą 2.

$4 < 5$

$tab[4] \% 3 = 0$, $j=4$, $i=1$, $k=5$

Znaleziony został element podzielony przez 3 w związku z tym musi się on znaleźć w lewej części tablicy bo należy do pierwszego podzbioru. Zamieniany jest z elementem, który znajduje się pod indeksem i . Nie ma potrzeby sprawdzać elementu, który został zamieniony (6) z wyszukany elementem (2), ponieważ ten element, został już wcześniej sprawdzony.

swap(*tab*[1], *tab*[4])

numer indeksu	0	1	2	3	4	5	6	7
wartość	9	6	10	13	1	2	11	8

$j=5$

Zbiór został podzielony na trzy podzbiory. Drugi podzbiór zaczyna się od indeksu $i+1$ czyli od indeksu 2. Trzeci zbiór zaczyna się od indeksu k , czyli od indeksu 5.

numer indeksu	0	1	2	3	4	5	6	7
wartość	9	6	10	13	1	2	11	8

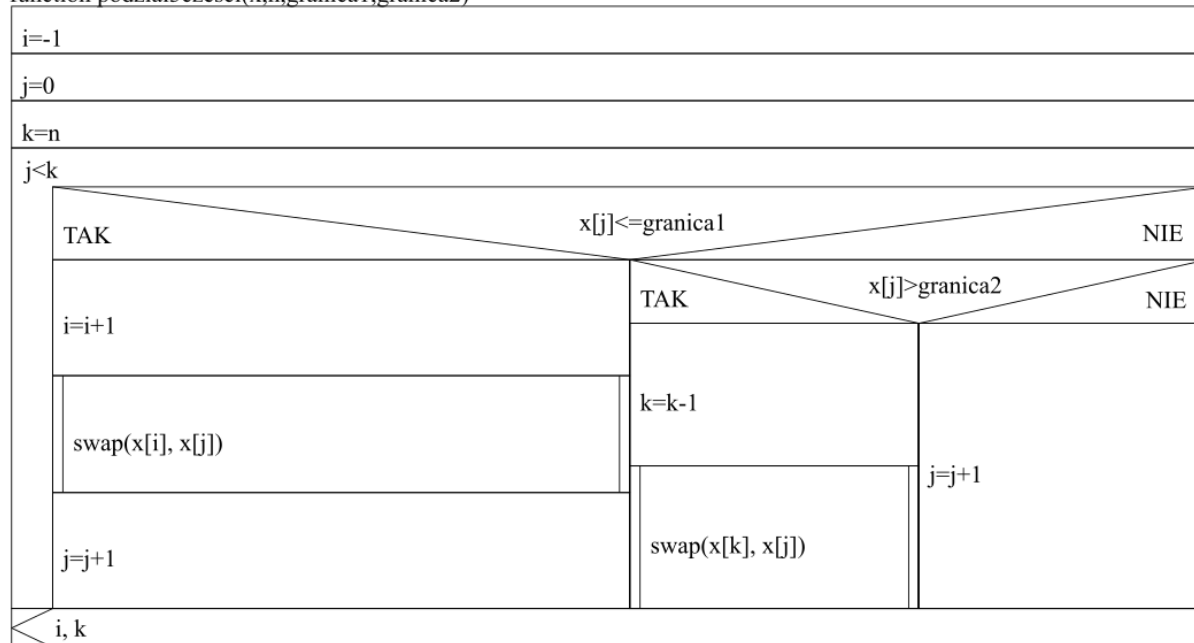


Na niebiesko zostały zaznaczone elementy należące do pierwszego podzbioru – liczby podzielne przez 3, na zielono zostały zaznaczone elementy należące do drugiego podzbioru – liczby podzielne przez 3 z resztą 1 oraz na pomarańczowo zostały zaznaczone elementy podzielne przez 3 z resztą 2.

Schemat zwarty NS:

Schemat zwarty NS, pokazany na rys 4.2, zawiera warunek w postaci nierówności tj. element mniejszy bądź równy od granicy1 i element większy od granicy2. Warunkiem niekoniecznie musi być nierówność może być to na przykład sprawdzenia podzielności liczb.

function podzial3czesci(x,n,granica1,granica2)



Rys. 4.2. Schemat zwarty NS algorytmu podziału zbioru na trzy części

Zadania do wykonania:

Do wykonania zadań należy zmodyfikować funkcje z poprzednich laboratoriów oraz wykorzystać kod wczytujący dane z pliku „studenci.csv”. Plik z danymi należy umieścić w katalogu projektu a nazwę pliku umieścić w kodzie programu.

Należy stworzyć jeden program zawierający menu wielokrotnego wyboru:

- 1) zadanie 4.1
- 2) zadanie 4.2
- 3) zadanie 4.3

Zadanie 4.1. Wczytanie danych

Należy stworzyć odpowiednią funkcję umożliwiającą wczytanie danych do programu z pliku.

Schemat wykonania zadania jest następujący:

- wczytanie rozmiaru tablicy z pliku,
- przydzielenie pamięci,

- wypełnienie tablicy danymi z pliku.

Zadanie 4.2. Podział zbioru na dwie części

Należy stworzyć funkcję, która w oparciu o wczytane z pliku dane o studentach podzieli tablicę w taki sposób, aby

- w pierwszej części znaleźli się studenci, którzy otrzymali mniej niż 10 bądź 10 punktów
- w drugiej studenci, którzy otrzymali więcej niż 10 punktów.

Dodatkowo należy zaimplementować funkcję do wyświetlania zawartości tablicy po podziale. Zawartość tablicy należy wyświetlić w następującym formacie:

Studenci, którzy otrzymali ≤ 10 punktów:

.....

Studenci, którzy otrzymali > 10 punktów:

.....

Schemat wykonania zadania jest następujący:

- wyświetlenie zawartości tablicy przed podziałem,
- podział,
- wyświetlenie zawartości tablicy po podziale.

Zadanie 4.3. Podział zbioru na trzy części

Należy stworzyć funkcję, która w oparciu o wczytane z pliku dane o studentach podzieli tablicę na trzy części. Należy podzielić tablicę w taki sposób, aby:

- w pierwszej części znaleźli się studenci, którzy uzyskali liczbę punktów podzielną przez 3,
- następnie studenci, których liczba punktów przy dzieleniu przez 3 daje resztę 1,
- a na końcu powinni znaleźć się Ci studenci, których liczba punktów przy dzieleniu przez 3 daje resztę 2.

Dodatkowo należy zaimplementować funkcję do wyświetlania zawartości tablicy po podziale. Zawartość tablicy należy wyświetlić w następującym formacie:

Studenci, którzy otrzymali liczbę punktów podzielnych przez 3:

.....

Studenci, którzy otrzymali liczbę punktów podzielnych przez 3 z resztą 1:

.....

Studenci, którzy otrzymali liczbę punktów podzielnych przez 3 z resztą 2:

.....

Schemat wykonania zadania jest następujący:

- wyświetlenie zawartości tablicy przed podziałem
- podział
- wyświetlenie zawartości tablicy po podziale



LABORATORIUM 5. ALGORYTMY TEKSTOWE

Cel laboratorium:

Omówienie algorytmów tekstowych: algorytm naiwny, algorytm Knutha-Morrisa-Pratta, algorytm Boyer'a-Moore'a.

Zakres tematyczny zajęć:

- algorytm naiwny wyszukiwania wzorca w tekście,
- algorytm Knutha- Morrisa-Pratta,
- algorytm Boyer'a-Moore'a.

Pytania kontrolne:

- 1) Jakie są algorytmy wyszukiwania wzorca w tekście?
- 2) Na czym polega algorytm naiwny?
- 3) W jaki sposób należy stworzyć tablicę dopasowań dla algorytmu Knutha-Morrisa-Pratta?
- 4) W jaki sposób dla znaku wyświetlić jego kod z tabeli ASCII?
- 5) Na czym polega algorytm Boyer'a-Moore'a?
- 6) Do czego służy tablica przesunięć dla algorytmu Boyer'a-Moore'a?
- 7) Jaka jest wada algorytmu naiwnego?

Wprowadzenie:

Z algorytmami tekstowymi związane są następujące pojęcia, które będą się pojawiały podczas pracy nad wyszukiwaniem fraz w *tekście*:

- *tekst* – ciąg znaków, w którym szukana jest dana fraza/słowo
- *wzorzec* – ciąg znaków szukany w tekście (fraz/słowo)
- wyszukiwanie wzorca – algorytm, wynikiem którego jest zbiór indeksów, od których zaczynają się słowa/frazy *tekstu*, równe *wzorcowi*

Poniżej zostaną zaprezentowane trzy algorytmy tekstowe, które służą do wyszukiwania wzorców w tekście

Algorytm naiwny:

Jest najprostszym algorytmem wyszukiwania wzorca w tekście. Jego zasada opiera się na porównywaniu odpowiednich znaków w tekście i we wzorcu. Porównywany jest znak po znaku. Jeśli porównywane znaki się zgadzają to następny znak z tekstu i wzorca są porównywane. Jeśli zostaną porównane wszystkie znaki ze wzorca, oznacza to, że wzorzec został znaleziony, algorytm rozpoczyna przeszukiwanie od następnego znaku. Jeśli podczas porównania znak z tekstu i ze wzorca nie zgadzają się, należy rozpocząć algorytm porównania od początku wzorca zaczynając od kolejnego znaku w tekście.

Przykład:

Tekst:

indeks	0	1	2	3	4	5	6
znak	A	B	B	B	A	B	B

Wzorzec:

indeks	0	1
znak	A	B

Wyszukiwanie:

Tekst[0] czy równy? *wzorzec*[0], TAK

Tekst[1] czy równy? *wzorzec*[1], TAK, wzorzec został znaleziony,

Tekst[1] czy równy? *wzorzec*[0], NIE

Tekst [2] czy równy? *wzorzec*[0], NIE

Tekst [3] czy równy? *wzorzec*[0], NIE

Tekst [4] czy równy? *wzorzec*[0], TAK

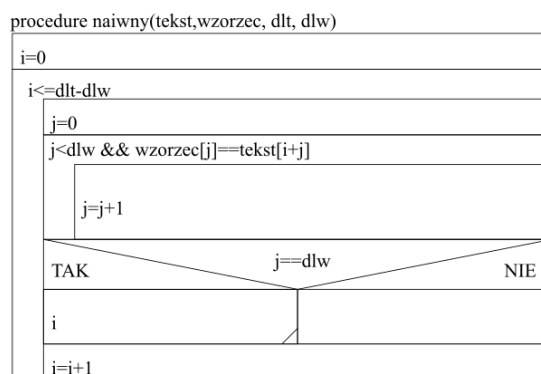
Tekst [5] czy równy? *wzorzec*[1], TAK, wzorzec został znaleziony

Tekst [5] czy równy? *wzorzec*[0], NIE

Algorytm kończy się, ponieważ liczba znaków do przeszukania jest mniejsza niż długość *wzorca*. Algorytm wyszukał, że *wzorzec* zaczyna się w *tekście* od indeksu 0 oraz 4.

Rys. 5.1 przedstawia schemat zwarty NS naiwnego algorytmu wyszukiwania wzorca w tekście

Indeks *j* odpowiada za przemieszczanie się po *wzorcu*, natomiast indeks *i* odpowiada za przesunięcie względem początku tablicy, gdzie przechowywany jest *tekst*.



Rys. 5.1. Schemat zwarty NS algorytmu naiwnego

Algorytm Knutha-Morrisa-Pratta:

Modyfikuje algorytm naiwny, który przesuwają się zawsze o jeden znak. W algorytmie KMP wykorzystywana jest informacja o tym o ile może przesunąć się algorytm, aby był sens

poszukiwań wzorca w tekście. Algorytm korzysta z tablicy częściowych dopasowań, którą należy stworzyć przed rozpoczęciem właściwego wyszukiwania. Wzorzec zawiera w sobie informację pozwalającą określić, gdzie powinna się zacząć kolejna próba dopasowania, pomijając ponowne porównywanie już dopasowanych znaków.

Tablica dopasowań

Jest wyznaczana na podstawie wzorca. Wartość każdej komórki tablicy dopasowań jest długością najdłuższego właściwego prefiksu tej części, będącego jednocześnie jej najdłuższym właściwym sufiksem. Właściwy prefiks danego słowa, który jest równy jego właściwemu sufiksowi, nazywamy prefikso-sufiksem. Poniżej znajduje się przykładowa tablica dopasowań. Właściwy prefiks/sufiks nie obejmuje całego słowa, tylko n pierwszych/ostatnich znaków.

indeks	0	1	2	3	4	5	6	7	8
wzorzec		a	b	a	c	a	b	a	b
dopasowanie	0	0	0	1	0	1	2	3	2

$dopasowanie[6]$ oznacza, że dla części całego słowa kończącej się literą znajdującą się pod indeksem 6 najdłuższy właściwy sufikso-prefiks ma długość 2:

abacab, część, która została już zanalizowana

$dopasowanie[7]$ najdłuższy sufikso-prefiks ma długość 3:

abacaba, część, która została już zanalizowana.

W oparciu o następujące prawo budowana jest tabela częściowych dopasowań:

Jeżeli $wzorzec[i] = wzorzec[p[i-1]+1]$, to $p[i] = p[i-1] + 1$, gdzie p to tablica dopasowań

Wyszukiwanie wzorca

Wyszukiwanie polega na porównywaniu kolejnych znaków z tekstu oraz ze wzorca. Jeśli zostaną porównane wszystkie znaki ze wzorca, oznacza to, że wzorzec został znaleziony lub jeśli znaki się nie zgadzają, za pomocą tablicy dopasowań wyznaczane jest przesunięcie, na podstawie którego algorytm po raz kolejny zaczyna szukać wzorca. W najgorszym przypadku wyszukiwanie zacznie się ponownie od kolejnego elementu w tekście.

Przykład:

wzorzec: abacabab wraz z tablicą dopasowań, zwaną dalej p

indeks	0	1	2	3	4	5	6	7	8
wzorzec		a	b	a	c	a	b	a	b
dopasowanie	0	0	0	1	0	1	2	3	2

tekst: abacabacabacababababab

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
tekst	a	b	a	c	a	b	a	c	a	b	a	c	a	b	a	b

i – indeks, który wskazuje skąd zaczyna się dopasowanie, przesunięcie okna o i



j – indeks, który przemieszcza się po kolejnych elementach tablicy *tekst*, sprawdzanie zaczyna się od j elementu okna

I poszukiwanie

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
tekst	a	b	a	c	a	b	a	c	a	b	a	c	a	b	a	b

$i=0$

$j=0$

$i+j=0$

dopóki $wzorzec[j]$ oraz $tekst[i+j]$ są sobie równe, $i+j=7$, $i=0$, $j=7$

$i=i+\max(1, j-p[j])=0+\max(1, 7-p[7])=\max(1, 4)=4$

$j=p[7]=3$

II poszukiwanie:

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
znak	a	b	a	c	a	b	a	c	a	b	a	c	a	b	a	b

$i=4$

$j=3$

$i+j=7$

dopóki $wzorzec[j]$ oraz $tekst[i+j]$ są sobie równe, $i+j=11$, $i=4$, $j=7$

$i=i+\max(1, j-p[j])=4+\max(1, 7-p[7])=4+\max(1, 4)=8$

$j=p[7]=3$

III poszukiwanie:

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
znak	a	b	a	c	a	b	a	c	a	b	a	c	a	b	a	b

$i=8$

$j=3$

$i+j=11$

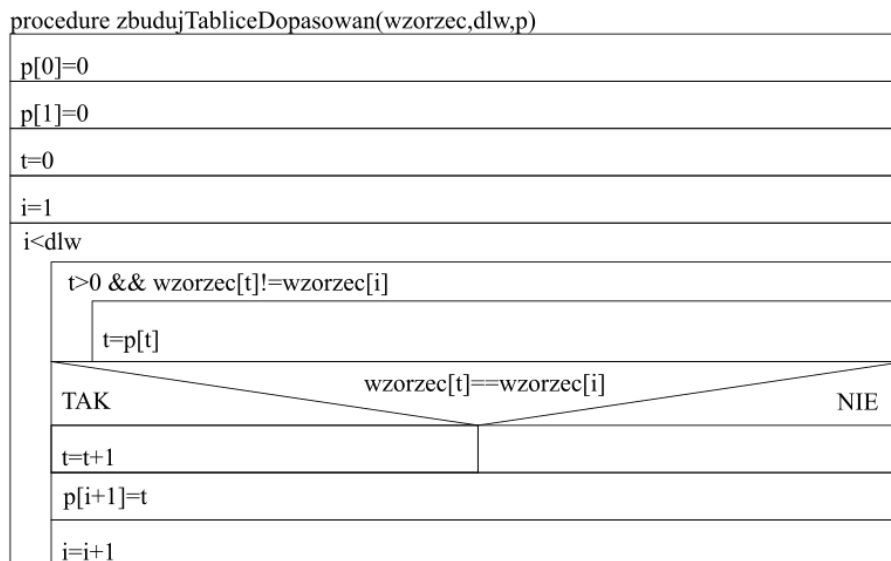
dopóki $wzorzec[j]$ oraz $tekst[i+j]$ są sobie równe, $i+j=15$, $i=8$, $j=7$

koniec algorytmu, wzorzec został znaleziony

Wzorzec zaczyna się od indeksu 8

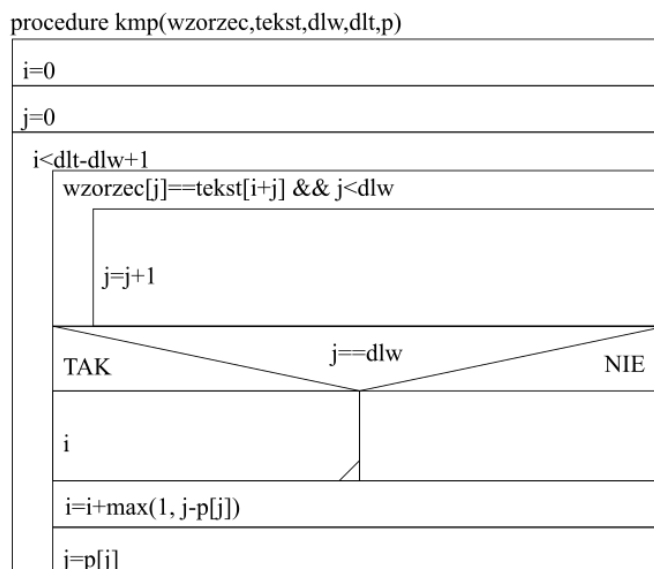
indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
znak	a	b	a	c	a	b	a	c	a	b	a	c	a	b	a	b

Na rys. 5.2 przedstawiono schemat zwarty NS algorytmu wyznaczania tablicy dopasowań dla algorytmu KMP.



Rys. 5.2. Schemat zwarty NS wyznaczania tablicy dopasowań

Na rys. 5.3 przedstawiono schemat zwarty NS algorytmu wyszukiwania wzorca tekście za pomocą metody KMP.



Rys. 5.3. Schemat zwarty NS algorytmu Knutha-Morrisa-Pratta

Tablica ASCII:

Kod znaku można uzyskać poprzez wykonanie operacji rzutowania zmiennej typu znakowego na typ całkowity: (int)znak.

Znak, odpowiadający liczbie całkowitej można uzyskać, używając operacji rzutowania zmiennej typu całkowitego na typ znakowy: (char)liczba.



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Dla przykładu kody ASCII małych liter alfabetu łacińskiego znajdują się w przedziale [97, 122], czyli literze ‘a’ odpowiada liczba 97, a literze ‘z’ – 122 itd. Poniżej na listingu 5.1. znajduje się kod umożliwiający wyświetlenie znaków i odpowiadających im kodów w rozszerzonej tablicy ASCII (podstawowa tablica ASCII zawiera 128 znaków).

```
1 for (int i = 0; i < 256; i++)
2     cout << (char)i << " " << i << endl;
```

Listing 5.1. Wyświetlenie znaków i ich kodów ASCII

Algorytm Boyer’a-Moore’a:

Porównywanie tekstu ze wzorcem zaczyna się od porównania ostatniego znaku wzorca. Dzięki temu jeśli ostatni znak wzorca nie zgadza się ze znakiem w tekście i znak z tekstu nie występuje we wzorcu, to okno wzorca można od razu przesunąć o tyle pozycji ile znaków ma wzorec. Jeśli znak w tekście występuje we wzorcu, należy tak przesunąć okno wzorca, aby znak z tekstu i ostatnie wystąpienie tego znaku ze wzorca zrównały się pozycjami. W celu wyznaczenia odpowiedniego przesunięcia algorytm korzysta z tablicy przesunięć.

Tablica przesunięć:

Tablica przyjmuje rozmiar równy liczbie elementów z którego składa się alfabet, czyli liczba wszystkich możliwych znaków, które mogą wystąpić w tekście, jak i we wzorcu. Tworząc tablicę przesunięć należy pamiętać o trzech głównych zasadach:

Jeśli znak alfabetu nie występuje we wzorcu, pod indeks odpowiadający znakowi można wpisać -1 lub długość wzorca zamiast liczby. Informacja ta służy jedynie rozróżnieniu w późniejszym etapie, czy dany znak występuje we wzorcu.

Jeśli dany znak występuje kilka razy we wzorcu, należy wpisać do tablicy największą odpowiadającą mu wartość indeksu. Można również zastosować podejście, polegające na tym, aby wpisywać najmniejszy indeks, pod którym znajduje się znak licząc od prawej strony. W przykładzie przedstawiony poniżej zostało przedstawione podejście ze wpisywaniem największej wartości indeksu, który odpowiada danemu znakowi. Jeśli znak nie występuje we wzorcu, wpisano -1.

Przykład:

Alfabet składa się z następujących liter: a, b, c, d, e.

Wzorec:

indeks	0	1	2	3
wzorec	b	a	a	c

Tablica dopasowań ma rozmiar 5, ponieważ alfabet składa się z 5 znaków.

W celu przyporządkowania literze numeru indeksu należy posłużyć się na przykład tablicą ASCII. Mała litera ‘a’ ma kod 97, ‘b’ – 98, ‘c’ – 99, ‘d’ – 100, ‘e’ – 101. Widać, że są to kolejne litery alfabetu stąd ich kody są kolejnymi liczbami. Jeśli od każdego kodu odejmiemy 97, wtedy otrzymamy, że literce ‘a’ przyporządkowane zostało 0 a literce ‘e’ – 4.

indeks	0	1	2	3	4
znak	a	b	c	d	e
wartość	2	0	3	-1	-1



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Wyszukiwanie wzorca:

Porównywanie *tekstu* ze *wzorcem* zaczyna się od porównania ostatniego znaku *wzorca*. W przypadku niezgodności wyróżniane są dwa przypadki. Niech i oznacza pozycję początku okna w tekście, j oznacza pozycję we wzorcu, na której występuje niezgodność, a p oznacza tablicę przesunięć:

- Znak z *tekstu* nie występuje we *wzorcu* – okno można przesunąć o tyle pozycji jakiej długości jest *wzorzec*, czyli $i = i + j - p[\text{znak}]$ (lub $i = i + \max(1, j - p[\text{znak}])$).
- Znak z *tekstu* występuje we *wzorcu* – okno *wzorca* należy przesunąć tak aby sprawdzany znak z *tekstu* zgadzał się ze znakiem ze *wzorca* jeżeli jest to możliwe, czyli $i = i + \max(1, j - p[\text{znak}])$. Okno zawsze porusza się w prawo stąd jest zabezpieczenie przed przesunięciem się w lewo okna: $\max(1, j - p[\text{znak}])$, jeśli $j - p[\text{znak}]$ będzie miało wartość ujemną wtedy okno przesunie się o 1 w prawo.

Przykład:

Alfabet: a,b,c,d,e

Wzorzec:

indeks	0	1	2	3
wzorzec	b	a	a	c

Tablica przesunięć(p):

indeks	0	1	2	3	4
znak	a	b	c	d	e
wartość	2	0	3	-1	-1

Tekst:

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
tekst	b	a	b	a	a	c	b	d	b	a	a	c	e	c

Na czerwono zostały zaznaczone elementy, które znajdują się w oknie, którego elementy są aktualnie porównywane ze *wzorcem*.

$$i=0$$

$$j=3$$

$$i+j=3$$

$\text{Tekst}[i+j]=a$, $\text{wzorzec}[j]=c$, znaki nie są równe, znak 'a' występuje we *wzorcu* więc:
 $i=0 + \max(1, 3 - p[0])=0+1$

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
tekst	b	a	b	a	a	c	b	d	b	a	a	c	e	c

$$i=1$$

$$j=3$$

$$i+j=4$$



$Tekst[i+j]=a$, $wzorzec[j]=c$, znaki nie są równe, znak 'a' występuje we wzorcu więc:
 $i=1+ \max (1, 3 - p[0])=1+1$

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
tekst	b	a	b	a	a	c	b	d	b	a	a	c	e	c

$i=2$

$j=3$

$i+j=5$

$Tekst[i+j]=c$, $wzorzec[j]=c$, znaki są równe, sprawdzane są wszystkie znaki zawierające się w tym oknie. Wszystkie znaki były zgodne ze wzorcem, więc wzorec został znaleziony, zaczyna się od indeksu 2; $i=3$

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
tekst	b	a	b	a	a	c	b	d	b	a	a	c	e	c

$i=3$

$j=3$

$i+j=6$

$Tekst[i+j]=b$, $wzorzec[j]=c$, znaki nie są równe, znak 'b' występuje we wzorcu więc:
 $i=3+ \max (1, 3 - p[1])=3+3=6$

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
tekst	b	a	b	a	a	c	b	d	b	a	a	c	e	c

$i=6$

$j=3$

$i+j=9$

$Tekst[i+j]=a$, $wzorzec[j]=c$, znaki nie są równe, znak 'a' występuje we wzorcu więc:
 $i=6+ \max (1, 3 - p[0])=6+1=7$

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
tekst	b	a	b	a	a	c	b	d	b	a	a	c	e	c

$i=7$

$j=3$

$i+j=10$

$Tekst[i+j]=a$, $wzorzec[j]=c$, znaki nie są równe, znak 'a' występuje we wzorcu więc:
 $i=7+ \max (1, 3 - p[0])=7+1=8$

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
tekst	b	a	b	a	a	c	b	d	b	a	a	c	e	c

$i=8$

$j=3$



$i+j=11$

$Tekst[i+j]=c$, $wzorzec[j]=c$, znaki są równe, sprawdzane są wszystkie znaki zawierające się w tym oknie. Wszystkie znaki były zgodne ze *wzorcem*, więc *wzorzec* został znaleziony, zaczyna się od indeksu 8; $i=9$

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
tekst	b	a	b	a	a	c	b	d	b	a	a	c	e	c

$i=9$

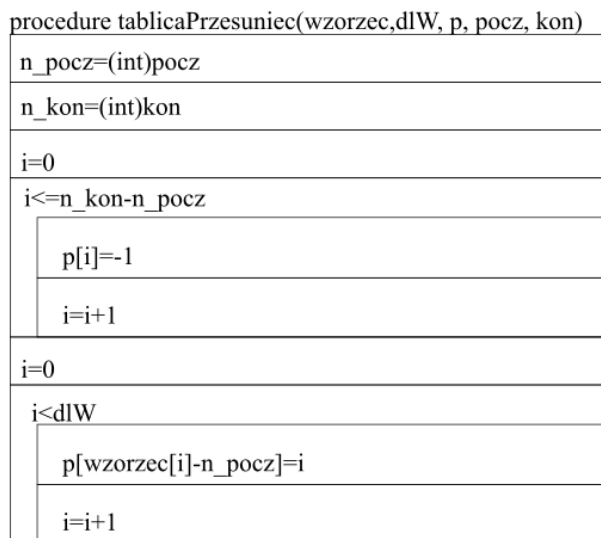
$j=3$

$i+j=12$

$Tekst[i+j]=e$, $wzorzec[j]=c$, znaki nie są równe, znak 'e'; nie występuje we *wzorcu* więc: $i=9 + \max(1, 3 - p[4])=9+4=13$, co kończy algorytm, ponieważ $j=3$ więc $i+j=16$, a ostatni indeks w tekście to 13.

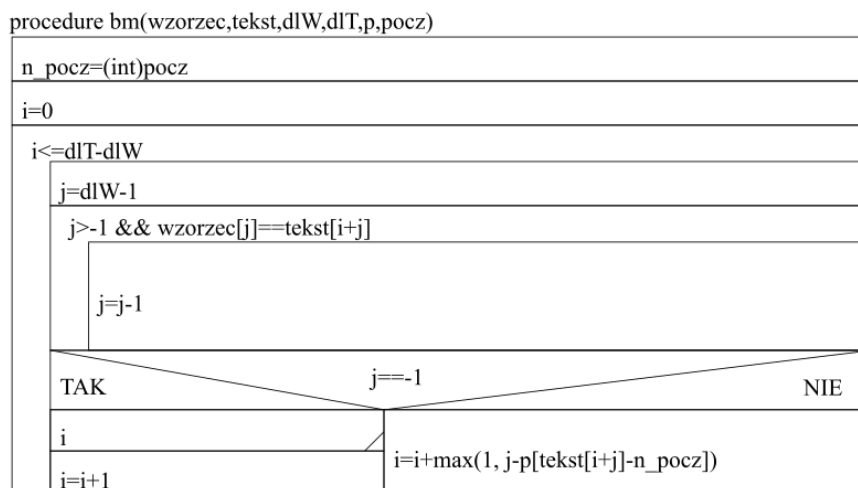
W tekście wzorec został znaleziony dwukrotnie. Pierwsze wystąpienie wzorca zaczyna się od indeksu 2 a drugie od indeksu 8.

Na rys. 5.4 podano schemat zwarty NS algorytmu zbudowania tablicy przesunięć dla algorytmu Boyer'a-Moore'a:



Rys. 5.4. Schemat zwarty NS tablicy przesunięć

Na rys. 5.5 podano schemat zwarty NS algorytmu wyszukiwania wzorca w tekście za pomocą algorytmu Boyer'a-Moore'a.



Rys. 5.5. Schemat zwarty NS algorytmu Boyer'a-Moore'a

Zadania do wykonania:

Należy stworzyć jeden program zawierający menu wielokrotnego wyboru:

- 1) zadanie 5.1
- 2) zadanie 5.2
- 3) zadanie 5.3
- 4) zadanie 5.4

Zadanie 5.1 Wczytanie danych

Należy wczytać od użytkownika wzorzec oraz tekst, w którym będzie szukany podany wzorzec. Należy zwrócić uwagę, że użytkownik może podać tekst/wzorzec zawierający spacje.

Zadanie 5.2 Algorytm naiwny

W funkcji powinny zostać wyświetlone indeksy wszystkich elementów, od których zaczyna się wzorzec w podanym przez użytkownika tekście. Znalezienie wzorca w tekście powinno zostać zrealizowane w oparciu o algorytm naiwny.

Zadanie 5.3 Algorytm Knutha-Morrisa-Pratta

W funkcji powinny zostać wyświetlone indeksy wszystkich elementów, od których zaczyna się wzorzec w podanym przez użytkownika tekście. Znalezienie wzorca w tekście powinno zostać zrealizowane w oparciu o algorytm Knutha-Morrisa-Pratta.

Zadanie 5.4 Algorytm Boyer'a-Moore'a

W funkcji powinny zostać wyświetlone indeksy wszystkich elementów, od których zaczyna się wzorzec w podanym przez użytkownika tekście. Znalezienie wzorca w tekście powinno zostać zrealizowane w oparciu o algorytm Boyer'a-Moore'a.



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



LABORATORIUM 6. ALGORYTMY TEKSTOWE Z UŻYCIEM FUNKCJI HASZUJĄCYCH

Cel laboratorium:

Omówienie algorytmów tekstowych z użyciem funkcji haszujących.

Zakres tematyczny zajęć:

- funkcja haszująca,
- algorytm Karpa-Rabina.

Pytania kontrolne:

- 1) Co to jest funkcja haszująca i do czego służy?
- 2) Na czym polega algorytm Karpa-Rabina?
- 3) W jaki sposób wyliczyć hasz na podstawie poprzednio wyliczonego haszu?
- 4) Jakie zastosowanie ma użycie arytmetyki modulo przy implementacji algorytmu Karpa-Rabina?

Funkcja haszująca w algorytmach tekstowych:

Funkcja haszująca traktuje przetwarzany łańcuch tekstu jak liczbę, zapisaną przy wybranej podstawie (podstawa może być równa rozmiarowi alfabetu lub być dowolną inną liczbą). Kolejne znaki tekstu są traktowane jak cyfry tej liczby.

Algorytm Karpa-Rabina:

Algorytm opiera się na wyliczeniu funkcji haszującej. Hasz liczony jest dla wzorca oraz dla znaków w danym oknie. Jeśli hasze są takie same, należy wykonać porównanie wzorca z bieżącym oknem tekstu znak po znaku za pomocą naiwnego algorytmu wyszukiwania wzorca. Możliwe są kolizje, czyli może wystąpić taka sama wartość haszu dla dwóch różnych ciągów znaków. Poniżej zostanie zaprezentowany algorytm wyznaczania haszu.

W celu uniknięcia zbyt dużych wartości haszu często stosowana jest arytmetyka modularna – wyniki wszystkich działań są sprowadzane do reszty z dzielenia przez wybrany moduł, który zwykle jest liczbą pierwszą. Zaletą stosowania takiej arytmetyki jest to, że można szukać dłuższych wzorców w tekście

Jeśli za podstawę w funkcji haszującej chcemy przyjąć liczebność alfabetu należy wiedzieć z czego składa się alfabet, czyli jakie znaki mogą się pojawić we wzorcu i w tekście, na przykład: duże litery alfabetu albo małe litery alfabetu. Jeśli alfabet nie jest znany należy założyć, że alfabetem jest cała tablica ASCII (256 znaków).



Wyznaczenie haszu

W celu wyznaczenia haszu należy posłużyć się poniższym wzorem:

$$hash(s) = \sum_{i=0}^{n-1} T_i p^{n-i-1}$$

gdzie:

T – liczba, odpowiadająca i -tej literze tekstu T w używanym systemie kodowania,

p – podstawa systemu kodowania,

n – liczba znaków w tekście,

i – numer litery w tekście, licząc od prawej do lewej.

Przykład:

Zakładamy, że alfabet składa się z 3 liter: A, B, C, a podstawą w funkcji haszującej będzie liczba 3.

W tabeli ASCII:

A – kod 65

B – kod 66

C – kod 67

Od każdego kodu odejmujemy 65, w celu przyporządkowania znakom odpowiadającym im liczbom możliwie najmniejszej podstawy

Otrzymujemy:

A – 0

B – 1

C – 2

Zakładamy, że alfabet składa się tylko ze znaków A, B, C oraz że znakowi A została przyporządkowane jest 0, B – 1 oraz C – 2.

indeks	0	1	2	3
tekst	A	B	A	B
wykładnik potęgi	3	2	1	0

$$Hasz = 1 \cdot 3^0 + 0 \cdot 3^1 + 1 \cdot 3^2 + 0 \cdot 3^3$$

Przykład:

Zakładamy, że alfabet składa się z trzech znaków: A, B, C oraz że znakowi A przyporządkowane jest 0, znakowi B – 1, C – 2.

Wzorzec:

indeks	0	1	2	3
wartość	A	B	A	B

Tekst:

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
wartość	C	A	B	A	B	A	B	A	B	A	B	C	A	B

Wartość *haszu* dla *wzorca* jest liczona tylko raz, ponieważ się nie zmienia:

$$HaszW = 1 \cdot 3^0 + 0 \cdot 3^1 + 1 \cdot 3^2 + 0 \cdot 3^3$$



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



W związku z tym, że wzorzec składa się z 4 znaków, to okno w *tekście* będzie miało długość 4. Algorytm zacznie wyliczać *hasz* dla znaków znajdujących się pod indeksami 0 – 3, następnie 1 – 4 itd. Nowe okno w stosunku do poprzedniego będzie pozbywało się pierwszej litery (stojącej po lewej stronie) i zawierało kolejną literę z prawej strony. Nie należy liczyć *haszu* od nowa, tylko zmodyfikować wartość poprzedniego *haszu* w celu uzyskania nowego *haszu*. Należy wykonać trzy poniższe kroki:

- odjąć pierwszą literę, która już nie bierze udziału w funkcji *hasz*,
- przesunąć cyfry o jeden rząd w lewo,
- dodać nową literę.

Wartości poszczególnych *haszów* dla okna o długości 4 (długość *wzorca*)

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
wartość	C	A	B	A	B	A	B	A	B	A	B	C	A	B

$$HaszT = 0 \cdot 3^0 + 1 \cdot 3^1 + 0 \cdot 3^2 + 2 \cdot 3^3$$

HaszW czy równy? *HaszT*, nie

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
wartość	C	A	B	A	B	A	B	A	B	A	B	C	A	B

$HaszT = HaszT - 2 \cdot 3^3$ - wartość *haszu* bez pierwszego znaku z lewej strony (C)

$HaszT = HaszT \cdot 3$ - przesunięcie cyfr o jedna pozycję w lewo

$HaszT = HaszT + 1 \cdot 3^0$ - dodanie jednej cyfry w prawej strony (B)

HaszT czy równy? *HaszW*, tak, *wzorzec* zaczyna się od indeksu 1 w *tekście*

indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
wartość	C	A	B	A	B	A	B	A	B	A	B	C	A	B

$HaszT = HaszT - 0 \cdot 3^3$ - wartość *haszu* bez pierwszego znaku z lewej strony (A)

$HaszT = HaszT \cdot 3$ - przesunięcie cyfr o jedna pozycję w lewo

$HaszT = HaszT + 0 \cdot 3^0$ - dodanie jednej cyfry w prawej strony (A)

HaszT czy równy? *HaszW*, nie

W analogiczny sposób należy wykonać kolejne kroki. Algorytm znalazł indeksy od których zaczyna się *wzorzec*: 1, 3, 5, 7

Na rys. 6.1 przedstawiono schemat zwarty NS wyznaczania wartości *hasza* w zależności od znaku(s), pozycji znaku(*exp*), podstawy(*p*) oraz przesunięcia początku stosowanego alfabetu względem początku układu ASCII(*off*). Na przykład, jeżeli stosowany alfabet zaczyna się od litery 'a', wartość zmiennej *off* wyniesie 97.

```
function make_hash(s, off, p, exp)
```

```
    hash = hash + ((int)s - off) * pow(p, exp)
```

```
    hash
```

Rys. 6.1. Schemat zwarty NS funkcji *haszującej*



Fundusze Europejskie
Wiedza Edukacja Rozwój

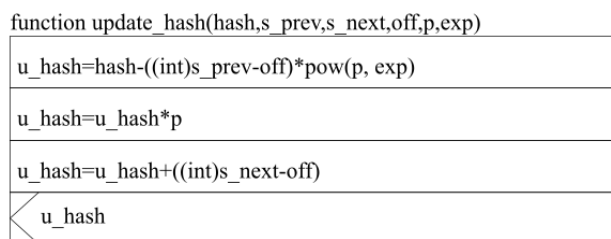


Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny

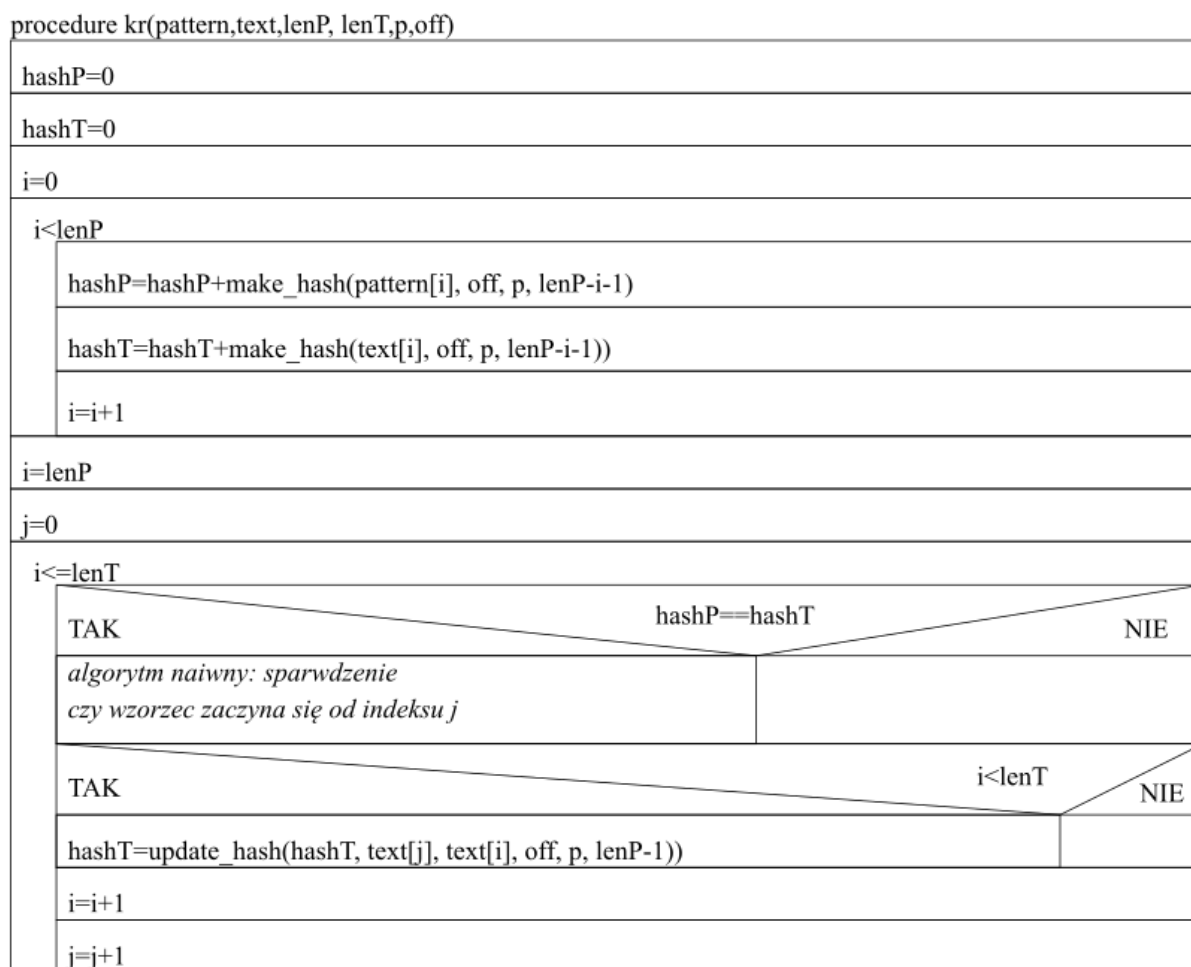


Na rys. 6.1 przedstawiono schemat zwarty NS aktualizacji wartości *hasha*.



Rys. 6.2. Schemat zwarty NS funkcji uaktualniającej hasz

Na rys. 6.1 przedstawiono schemat zwarty NS algorytmu Karpa-Rabina.



Rys. 6.3. Schemat zwarty NS algorytmu Karpa – Rabina

Zadania do wykonania:

Zadanie 6.1 Karp - Rabin

Napisz program, który znajdzie w każdej linijce w pliku tekstowym „tekst.txt” pozycje, od których w danej linijce zaczyna się wzorzec. Wzorzec znajduje się w pierwszej linijce

w pliku. Jeśli w danej linii nie występuje wzorec należy wyświetlić indeks -1. Zakładamy, że każda linijka jest analizowana oddzielnie i nie ma sytuacji w której część wzorca znajdzie się w jednej a pozostała część w kolejnej linijce. Dodatkowo zakładamy, że w pliku znajdują się tylko małe i duże litery alfabetu.

Na konsoli wyniki powinny zostać wyświetlone w następującym formacie:

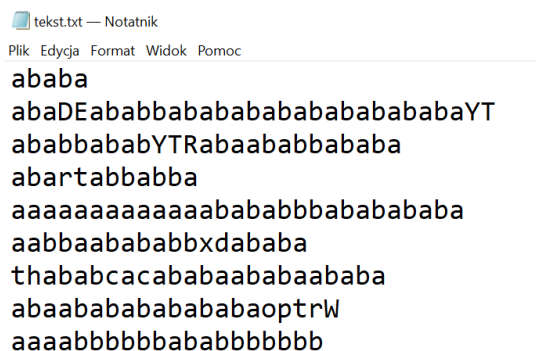
Linijka 2: numery indeksów od których zaczyna się wzorec

Linijka 3:

.....

Linijka n:

Wyszukiwanie wzorca w tekście należy zrealizować w oparciu o algorytm Karpa-Rabina. Na rys 6.4 przedstawiono przykładowy widok pliku „tekst.txt”.



tekst.txt — Notatnik
Plik Edycja Format Widok Pomoc

ababa
abaDEababbabababababababababaYT
ababbababYTRabaababbababa
abartabbabba
aaaaaaaaaaaaabababbbbababababa
aabbaabababbxdababa
thababcacababaababaababa
abaababababababaoptRW
aaaabbbbbbababbbbbbb

Rys. 6.4. Plik „tekst.txt”

LABORATORIUM 7. WYSZUKIWANIE INFORMACJI W ZBIORZE

Cel laboratorium:

Omówienie algorytmów wyszukiwania elementów w zbiorze.

Zakres tematyczny zajęć:

- algorytm wyszukiwania liniowego,
- algorytm wyszukiwania bisekcyjnego,
- zapis danych do pliku tekstowego.

Pytania kontrolne:

- 1) Na czym polega algorytm wyszukiwania liniowego?
- 2) Jaka jest złożoność czasowa algorytmu wyszukiwania liniowego?
- 3) Na czym polega algorytm wyszukiwania bisekcyjnego?
- 4) Jaka jest złożoność czasowa algorytmu wyszukiwania bisekcyjnego?
- 5) W jaki sposób zapisać dane do pliku tekstowego?

Wyszukiwanie liniowe:

Wyszukiwanie liniowe jest to rodzaj wyszukiwania sekwencyjnego polegający na przeglądaniu kolejnych elementów zbioru. Jeśli przeglądany element jest elementem który jest szukany, to zwracana jest jego pozycja(indeks) w zbiorze i algorytm kończy się. W przeciwnym razie przeglądane są kolejne elementy zbioru aż do momentu przejrzenia wszystkich elementów zbioru.

Wynikiem wyszukiwania liniowego jest indeks, pod którym znajduje się szukany element. Rozszerzona wersja algorytmu zwraca indeksy wszystkich elementów zbioru, których wartości są równe szukanej wartości.

Złożoność czasowa:

- optymistyczna: $O(1)$
- średnia: $O(n)$
- pesymistyczna: $O(n)$

Poniżej zostanie zaprezentowany przykład: znalezienie wartości 2 w tablicy.

Przykład:

numer indeksu	0	1	2	3	4
wartość	10	1	8	2	11

Szukana wartość: 2

Należy porównać każdy element tablicy z szukanym elementem. Kiedy elementy będą równe należy zwrócić indeks, pod którym znajduje się znaleziony element.

2 czy równe? 10, NIE

2 czy równe? 1, NIE

2 czy równe? 8, NIE

2 czy równe? 2, TAK

Wartość 2 znajduje się pod indeksem 3.



Fundusze Europejskie
Wiedza Edukacja Rozwój

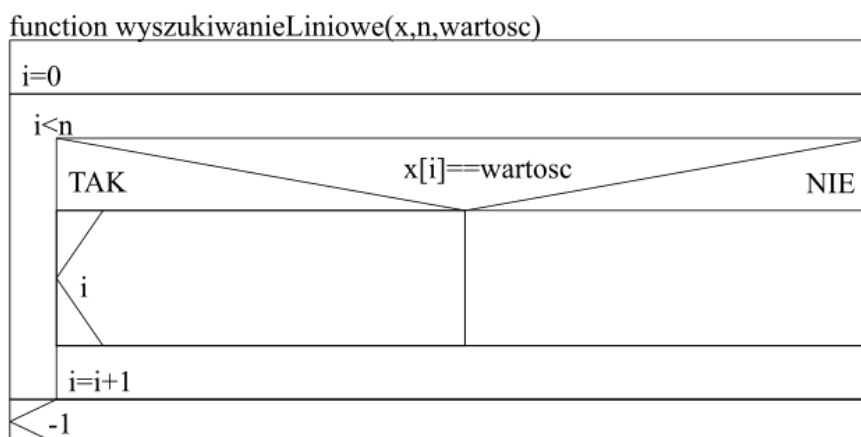


Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Na rys 7.1 przedstawiono schemat zwarty NS algorytmu wyszukiwania liniowego.



Rys. 7.1. Schemat zwarty NS – wyszukiwanie liniowe

Wyszukiwanie bisekcyjne:

Algorytm wyszukiwania bisekcyjnego nazywany jest również wyszukiwaniem binarnym, ponieważ polega na dwukrotnym zawężeniu podzbioru, w którym szukany jest element, w kolejnym kroku. Należy pamiętać, że zbiór musi być posortowany aby zastosować algorytm wyszukiwania bisekcyjnego. Algorytm polega na wyznaczeniu indeksu środkowego zbioru/podzbioru i sprawdzeniu czy element znajdujący się pod tym indeksem jest równy szukanemu. Jeśli tak, algorytm kończy się. W przeciwnym wypadku poszukiwany element jest albo mniejszy od elementu środkowego, albo większy. Elementy mniejsze niż element środkowy będą znajdowały się w lewej części podzbioru, a elementy większe będą znajdowały się w prawej części. W kolejnym kroku zawężamy obszar przeszukiwań do jednego podzbioru.

Złożoność czasowa:

- Optymistyczna: $O(1)$
- Średnia: $O(\log(n))$
- Pesymistyczna: $O(\log(n))$

Poniżej został przedstawiony przykład szukania wartości równej -1.

Wprowadźmy następujące oznaczenia:

- sr – środkowy indeks,
- k – lewy indeks,
- p – prawy indeks.

Przykład:

numer indeksu	0	1	2	3	4	5	6	7	8	9
wartość	-4	-1	2	6	11	18	56	78	81	100

$$sr = (\text{int}) (p + k) / 2,$$

$$p=0, k=9$$

$$sr = (0+9) / 2 = 4$$

$tab[4]=11$, 11 jest równe? -1, $p=0$, $k=sr-1=3$

Zbiór poszukiwań został zawężony

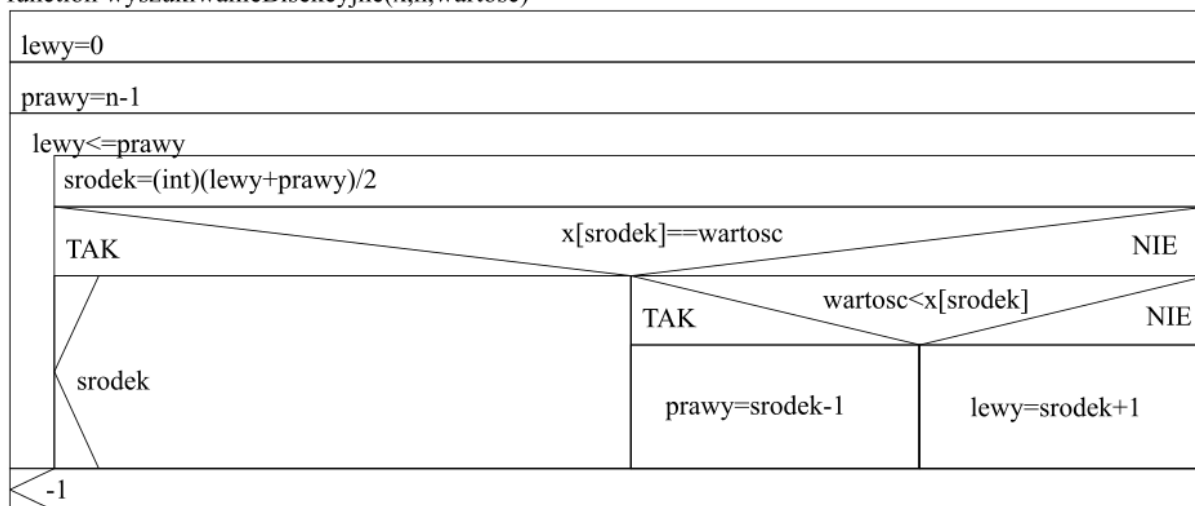
$p=0$, $k=3$

$sr=(0+3)/2 = 1$

$tab[1]=-1$, -1 jest równe? -1, wartość została znaleziona , numer indeksu: 1

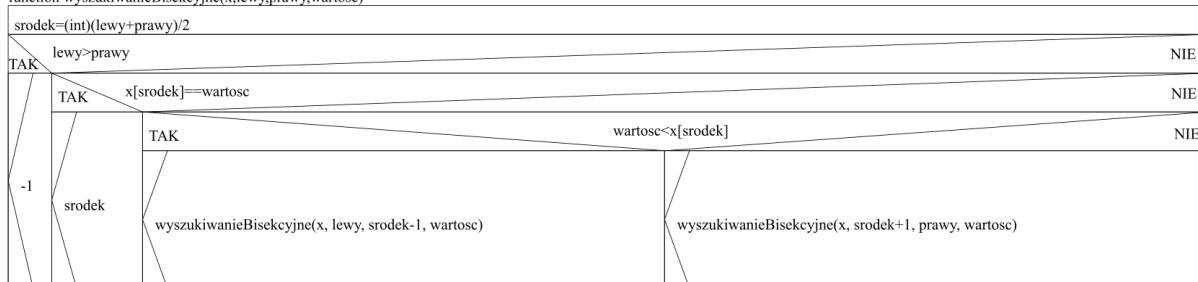
Implementacja algorytmu wyszukiwania bisekcyjnego w sposób iteracyjny została przedstawiona na rysunku 7.2. Natomiast implementacja za pomocą rekurencji została przedstawiona na rysunku 7.3.

function wyszukiwanieBisekcyjne(x,n,wartosc)



Rys. 7.2. Schemat zwarty NS – wyszukiwanie bisekcyjne - sposób iteracyjny

function wyszukiwanieBisekcyjne(x,lewy,prawy,wartosc)



Rys. 7.3. Schemat zwarty NS – wyszukiwanie bisekcyjne – rekurencja



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Zadania do wykonania:

Do wykonania zadań należy zmodyfikować funkcje z poprzednich laboratoriów oraz wykorzystać kod wczytujący dane z pliku „studenci.csv”. Plik z danymi należy umieścić w katalogu projektu a nazwę pliku umieścić w kodzie programu.

Należy stworzyć jeden program zawierający menu wielokrotnego wyboru:

- 1) zadanie 7.1
- 2) zadanie 7.2
- 3) zadanie 7.3

Zadanie 7.1. Wczytanie z pliku

Należy stworzyć odpowiednią funkcję umożliwiającą wczytanie danych do programu z pliku.

Schemat wykonania zadania:

- wczytanie rozmiaru tablicy z pliku,
- przydzielenie pamięci,
- wypełnienie tablicy danymi z pliku.

Zadanie 7.2. Wyszukiwanie liniowe

Należy stworzyć funkcję, która w oparciu o wczytane dane z pliku o studentach wyszuka i wyświetli wszystkie imiona i nazwiska studentów, którzy dostali podaną przez użytkownika liczbę punktów. W przypadku, jeśli żaden ze studentów nie posiada takiej liczby punktów, należy wyświetlić stosowny komunikat.

Zadanie 7.3. Wyszukiwanie bisekcyjne

Należy stworzyć funkcję, która w oparciu o wczytane dane z pliku o studentach wyszuka i wyświetli wszystkie imiona i nazwiska studentów, którzy dostali podaną przez użytkownika liczbę punktów. W przypadku, jeśli żaden ze studentów nie posiada takiej liczby punktów, należy wyświetlić stosowny komunikat. Dodatkowo imiona i nazwiska studentów, którzy uzyskali podaną liczbę punktów, powinny zostać zapisane do pliku o nazwie „wyniki.csv”. W każdej linijce powinny zostać zapisane dane jednego studenta. Imię i nazwisko należy oddzielić między sobą średnikiem. Plik powinien zostać zapisany w katalogu projektu.

Należy pamiętać, że przed wykonaniem algorytmu wyszukiwania bisekcyjnego należy posortować dane dowolnym algorytmem poznanym podczas poprzednich zajęć.

Poniżej został przedstawiony fragment kodu, który umożliwi zapisywanie danych do pliku: Listing 7.1 zawiera:

- linijka 1 – obiekt umożliwiający zapis do pliku,
- linijka 2 – zdefiniowanie ścieżki do zapisu,
- linijka 3 – otwarcie pliku w trybie do zapisu,
- linijka 5 – zapis liczby studentów do pliku wraz z enterem,
- linijki 6,7,8 – zapis kolejnych studentów do pliku,
- linijka 10 – zamknięcie pliku.



```
1  ofstream zapis;  
2  string sciezkaDoZapisu="wyniki.csv";  
3  zapis.open(sciezkaDoZapisu);  
4  
5  zapis<<liczbaStudentow<<endl;  
6  for(int i=0; i<liczbaStudentow; i++){  
7      zapis<<tab[i].imie<<" "<<tab[i].nazwisko<<" ";  
8      zapis<<tab[i].ocena<<endl; }  
9  
10 zapis.close();
```

Listing 7.1. Zapis do pliku tablicy zawierającej informacje o studentach

LABORATORIUM 8. KOŁOKWIUM NR 1

Cel laboratorium:

Weryfikacja nabytych umiejętności dotyczących algorytmów sortowania, podziału zbioru oraz wyszukiwania.

Wytyczne do kolokwium 2:

- zakres laboratoriów 1-7,
- próg zaliczeniowy 51%,
- pozostałe wytyczne i sposób zaliczenia kolokwium ustala prowadzący zajęcia.

LABORATORIUM 9. IMPLEMENTACJA STOSU

Cel laboratorium:

Omówienie struktury danych – stos.

Zakres tematyczny zajęć:

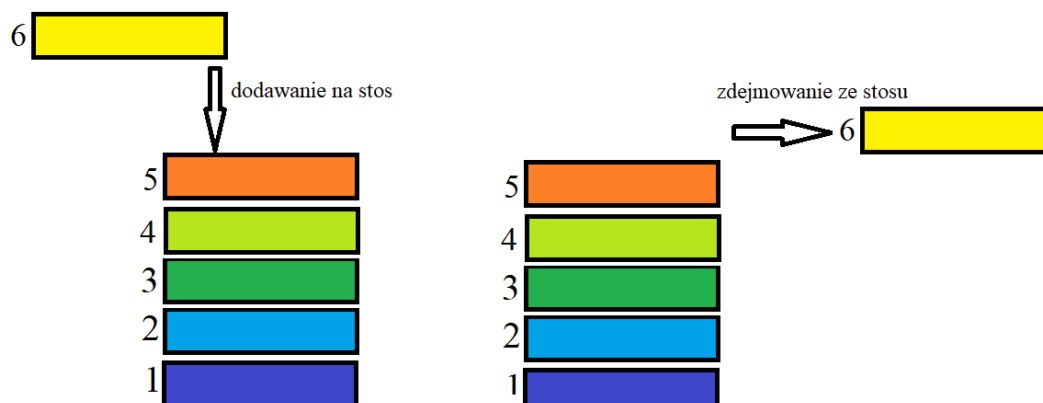
- operacje na stosie,
- implementacja stosu.

Pytania kontrolne:

- 1) Co to jest stos?
- 2) Jakie operacje można wykonać na strukturze stos?
- 3) W jakiej kolejności zdejmowane(usuwane) są elementy ze stosu?
- 4) W jaki sposób elementy są dodawane do stosu?
- 5) Co to jest skrót LIFO?
- 6) Na czym polega implementacja tablicowa stosu?
- 7) Na czym polega implementacja stosu za pomocą listy?

Stos:

Jest liniową strukturą danych, z której odczytujemy elementy w kolejności odwrotnej do ich dodawania. Struktura ta nosi nazwę LIFO (ang. Last In – First Out – wszedł ostatni, a wyszedł pierwszy). Dane dokładane są na wierzch stosu i z wierzchołka stosu są pobierane. Zdejmowanie elementów ze stosu odbywa się w odwrotnej kolejności do ich umieszczania. Ideę stosu można zilustrować jako stos książek na biurku. Jeśli chcemy położyć kolejną książkę, to kładziemy ją na szczycie (wierzchołku). Najpierw zostanie zdjęta książka, która została dodana na stos jako ostatnia. Książka, która została dodana jako pierwsza zostanie zdjęta ze stosu jako ostatnia. Na rysunku 9.1. przedstawiona została idea działania stosu.



Rys. 9.1. Idea działania stosu

Na stosie można wykonać następujące operacje:

- dodawanie elementu na stos(push) – umieszczenie elementu na szczycie stosu,



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



- usunięcie elementu ze stosu (pop) – zdejmowanie elementu znajdującego się na szczycie stosu,
- sprawdzenie, czy stos jest pusty (isEmpty),
- pobranie ostatniej wartości na stosie(top).

Implementacja stosu:

Rozróżniane są dwie implementacje stosu:

- Tablica:
 - zalety: prosta implementacja, szybkość,
 - wady: liczba elementów jest stała,
 - składa się z dwóch elementów: tablicy i indeksu wierzchołka stosu. Tablica - przechowuje elementy na stosie. Indeks wierzchołka stosu służy do zapamiętywania pozycji elementu dodanego do stosu jako ostatni. Po utworzeniu tablicy zmienna przechowująca pozycję wierzchołka stosu jest równa 0. Po dodaniu pierwszego elementu jej wartość jest zwiększana o jeden, przy kolejnych dodawanych elementach jest analogicznie. Podczas zdejmowania elementu ze stosu, wartość zmiennej przechowującej aktualną pozycję wierzchołka jest zmniejszana o jeden.
- Lista:
 - zalety: nieograniczona liczba elementów (liczba elementów ograniczona jest tylko rozmiarem pamięci operacyjnej),
 - wady: wolniejsze działanie, większe zużycie pamięci, spowodowane koniecznością przechowywania adresu następnego elementu,
 - poniżej zostanie przedstawiona implementacja stosu za pomocą listy.

Implementacja stosu za pomocą listy:

Pojedynczy element stosu w języku C++ jest implementowany w postaci struktury, składającej się z następujących pól:

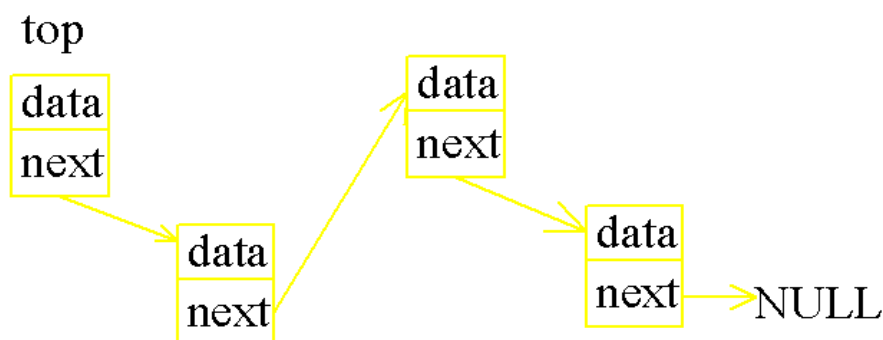
- pole/pola z danymi,
- adres elementu, znajdującego się na kolejnej pozycji.

Na listingu 9.1 dla przypomnienia wiadomości, przedstawiona została schematyczna implementacja struktury.

```
1 struct nazwa_struktury{
2     typDanych pole1;
3     typDanych pole2;
4     ...
5     typDanych polen;};
```

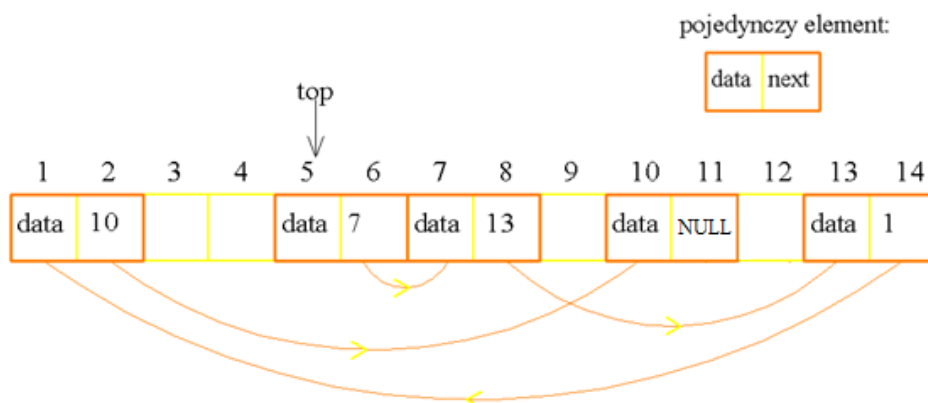
Rys. 9.1. Implementacja struktury

Poniżej na rysunku 9.2. przedstawiony został schemat ideowy struktury stos. Struktura składa się pola *data*, które przechowuje daną oraz z pola *next*, które przechowuje adres następnego elementu znajdującego się w stosie. Element, który znajduje się na samym dole stosu w polu *next* ma NULL. Zmienna *Top* przechowuje adres wierzchołka stosu.



Rys. 9.2. Ideowy schemat stosu

Poniższy schemat nie odzwierciedla struktury pamięci komputera sposób dokładny, służy jedynie wyjaśnieniu zasady funkcjonowania stosu w C++. Rysunek 9.3. przedstawia strukturę pamięci. Pojedynczy element został zaznaczony na pomarańczowo. Na żółto zaznaczony został fragment pamięci, w którym znajdują się poszczególne elementy stosu. Kolejnymi liczbami naturalnym zostały oznaczone adresy w pamięci. W lewym polu elementu przechowywana jest wartość całkowita a w prawym - adres kolejnego elementu. Wierzchołek stosu znajduje się pod adresem 5, element znajdujący się pod adresem 5 w polu *next* ma adres kolejnego elementu, który znajduje się pod adresem 7, następnie ten element w polu *next* ma adres kolejnego elementu znajdującego się pod adresem 13. Ostatni element stosu znajduje się pod adresem 10 i w polu *next* ma NULL.



Rys. 9.3. Ideowy schemat pamięci

Poniżej przedstawione zostaną schematy zwarte NS związane z wykonywaniem operacji na stosie. W celu lepszego zrozumienia działania poniższych funkcji zdefiniowana została następująca struktura – zawiera ona dwa pola – pierwsze pole przechowuje zmienną typu całkowitego a drugie pole zawiera wskaźnik na kolejny element – adres, pod którym znajduje się kolejny element – listing 9.2. W programie głównym należy utworzyć zmienną, która będzie przechowywała adres wierzchołka stosu – listing 9.3.

```
1 struct element{
2     int number;
3     element* next; };

```

Listing 9.2. Implementacja struktury przechowującej element stosu

```
1 element* stack=NULLptr;

```

Listing 9.3. Utworzenie zmiennej przechowującej adres wierzchołka stosu.

Na listingu 9.4 zaprezentowane zostało w jaki sposób NIE NALEŻY tworzyć zmiennej przechowującej adres wierzchołka stosu. Operator **new** używany jest tylko wtedy kiedy element jest dodawany do stosu. Podczas usuwania elementu ze stosu używany jest operator **delete**, który usuwa element z pamięci.

```
1 element* stack=new element;

```

Listing 9.4. Niewłaściwe utworzenie zmiennej przechowującej adres wierzchołka stosu

Na rys. 9.4, 9.5, 9.6 oraz 9.7 przedstawiono są schematy zwarte NS algorytmów operacji związanych ze stosem.

- operacja **push** – dodanie do stosu

procedure push(element* &stack, int value)

el=new element

el->number=value

el->next=stack

stack=el

Rys. 9.4. Operacja push – dodanie do stosu

- operacja **pop** – usunięcie elementu ze stosu

procedure pop(element* &stack)

temp=stack

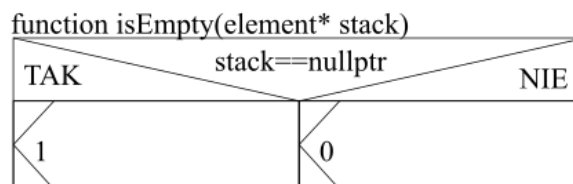
stack=stack->next

delete temp

Rys. 9.5. Operacja pop – usunięcie elementu ze stosu

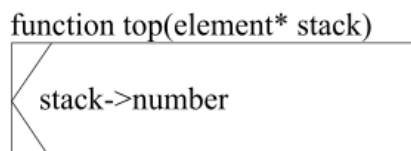


- sprawdzenie czy stos jest pusty – *isEmpty*



Rys. 9.6. Sprawdzenie czy stos jest pusty

- pobranie wartości elementu znajdującego się na wierzchołku stosu – *top*



Rys. 9.7. Operacja top – pobranie wartości elementu znajdującego się na wierzchołku stosu

Zadanie do wykonania:

Zadanie 9.1 Stos

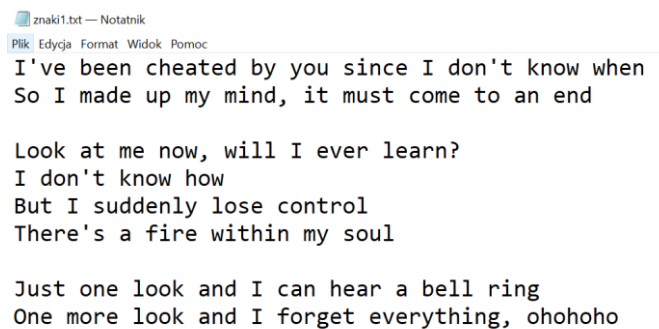
Stwórz strukturę zawierającą wskaźnik na kolejny element stosu oraz liczbę całkowitą w celu zaimplementowania stosu.

W programie głównym należy stworzyć wskaźnik na strukturę, menu wielokrotnego wyboru oraz zaimplementować funkcje:

- 1) sprawdzenie czy stos jest pusty, powinien zostać wyświetlony odpowiedni komunikat.
- 2) dodanie elementu na stos, liczbę, która powinna być dodana do stosu, należy wylosować z zakresu 1 - 10.
- 3) usunięcie elementu ze stosu, jeśli na stosie nie ma żadnego elementu należy wyświetlić stosowny komunikat, na przykład: "Stos jest pusty".
- 4) pobranie elementu ze stosu, należy wyświetlić ostatni element, który znajduje się na stosie.
- 5) usunięcie wszystkich elementów ze stosu, należy usunąć wszystkie elementy na stosie poprzez zdejmowanie kolejnych
- 6) elementów ze stosu.
- 7) wyjście z programu.

Zadanie 9.2 Odwrócenie kolejności

Napisz program, który odwróci kolejność znaków w pliku tekstowym „znaki1.txt” za pomocą struktury stos wykorzystując funkcje zaimplementowane w zadaniu 9.1. Odwrócony tekst należy zapisać do pliku „wynik.txt”.



znaki1.txt — Notatnik

Plik Edycja Format Widok Pomoc

I've been cheated by you since I don't know when
So I made up my mind, it must come to an end

Look at me now, will I ever learn?
I don't know how
But I suddenly lose control
There's a fire within my soul

Just one look and I can hear a bell ring
One more look and I forget everything, ohohoho

Rys. 9.8. Plik znaki1.txt

LABORATORIUM 10. IMPLEMENTACJA KOLEJKI

Cel laboratorium:

Omówienie struktury danych – kolejka.

Zakres tematyczny zajęć:

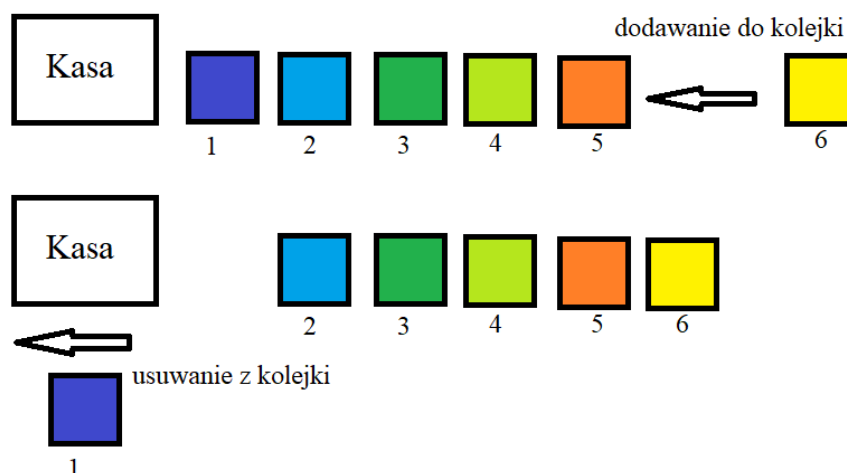
- operacje na kolejce,
- implementacja kolejki.

Pytania kontrolne:

- 1) Co to jest kolejka?
- 2) Jakie operacje można wykonać na kolejce?
- 3) Co to jest skrót FIFO?
- 4) Który element w kolejce jest usuwany jako pierwszy?
- 5) Gdzie dodawany jest element do kolejki?
- 6) Na czym polega implementacja tablicowa kolejki?
- 7) Na czym polega implementacja kolejki za pomocą listy?

Kolejka:

Kolejka jest strukturą danych, w której nowe dane dopisywane są na końcu, a z początku kolejki usuwane są elementy. Jest to struktura danych typu FIFO (First In, First Out; pierwszy na wejściu, pierwszy na wyjściu). Sposób działania struktury kolejki można porównać do kolejki w sklepie. Najpierw obsługiwani są klienci stojący z przodu obok kasy. Kiedy klient zostanie obsłużony, odchodzi i przychodzi kolejny klient. Na rysunku 10.1 przedstawiona została idea działania kolejki na przykładzie kolejki w sklepie.



Rys. 10.1. Idea działania kolejki

Operacje, które można wykonać na kolejce:

- dodanie elementu na koniec kolejki (*push*),
- usunięcie elementu z początku kolejki (*pop*),



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



- sprawdzenie czy kolejka jest pusta (*isEmpty*),
- odczytanie elementu z początku kolejki (*first*).

Implementacje kolejki:

- Tablica – rozmiar kolejki jest z góry określony – jest to rozmiar tablicy. W celu przechowywania danych w tablicy należy wprowadzić dwie zmienne: jedna ze zmiennych będzie przechowywać indeks gdzie znajduje się pierwszy element w kolejce a druga zmienna - indeks gdzie znajduje się ostatni element w kolejce.
- Lista.

Implementacja kolejki za pomocą listy:

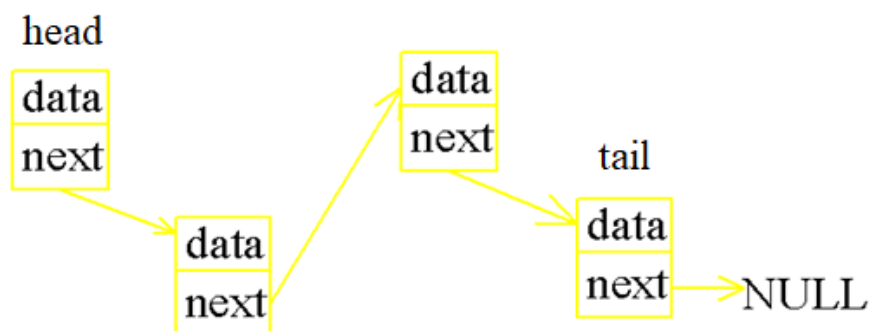
Pojedynczy element kolejki w języku C++ jest implementowany w postaci struktury, składającej się z następujących pól:

- pole/pola z danymi;
- adres elementu, znajdującego się na kolejnej pozycji.

W celu utworzenia struktury kolejki potrzebne są dwa wskaźniki:

- head – wskaźnik na pierwszy element znajdujący się w kolejce – przechowuje adres pierwszego elementu z kolejki;
- tail – wskaźnik na ostatni element znajdujący się w kolejce – przechowuje adres ostatniego elementu w kolejce.

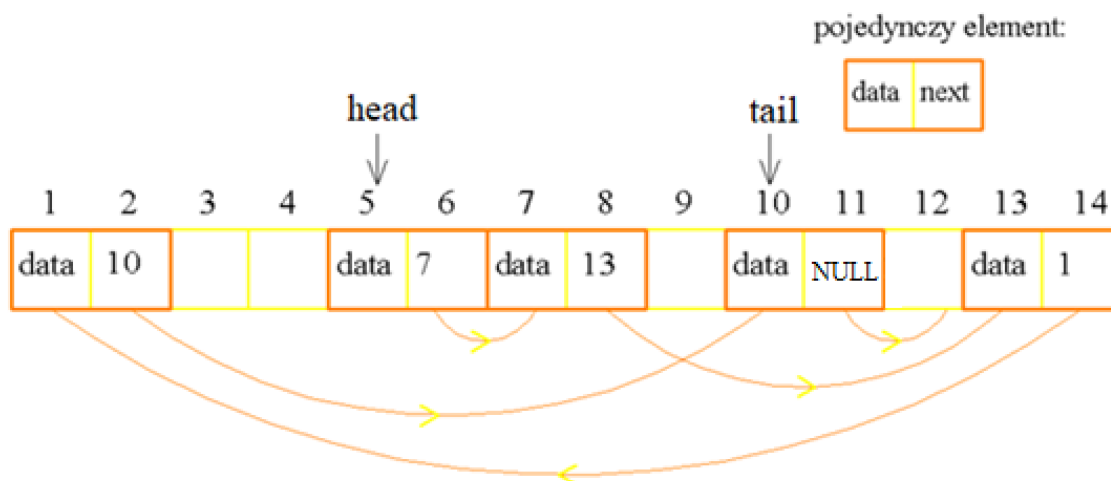
Poniżej na rysunku 10.2. przedstawiony został schemat ideowy struktury kolejki. Struktura składa się z pola *data*, które przechowuje daną, oraz z pola *next*, które przechowuje adres kolejnego elementu znajdującego się w kolejce. Ostatni element, który znajduje się w kolejce w polu *next* ma wartość NULL. Zmienna *head* przechowuje początek kolejki. Zmienna *tail* przechowuje koniec kolejki.



Rys. 10.2. Schemat działania kolejki

Poniższy schemat nie odzwierciedla struktury pamięci komputera sposób dokładny, służy jedynie wyjaśnieniu zasady funkcjonowania kolejki w C++. Rysunek 10.3 przedstawia strukturę pamięci. Pojedynczy element został zaznaczony na pomarańczowo. Na żółto zaznaczony został fragment pamięci, w którym znajdują się poszczególne elementy kolejki. Kolejnymi liczbami naturalnym zostały oznaczone adresy w pamięci. W lewym polu

elementu przechowywana jest wartość całkowita a w prawym - adres kolejnego elementu. Początek kolejki(*head*) znajduje się pod adresem 5. Element znajdujący się pod adresem 5 w polu *next* ma adres kolejnego elementu, który znajduje się pod adresem 7, następnie ten element w polu *next* ma adres kolejnego elementu znajdującego się pod adresem 13. Ostatni element kolejki(*tail*) znajduje się pod adresem 10 i w polu *next* ma wartość NULL.



Rys. 10.3. Ideowy schemat pamięci

Poniżej przedstawione zostaną schematy zwarte NS związane z wykonywaniem operacji na kolejce. W celu lepszego zrozumienia działania poniższych funkcji zdefiniowana została następująca struktura – zawiera ona dwa pola – pierwsze pole przechowuje zmienną typu znakowego a drugie pole zawiera wskaźnik na kolejny element – adres, pod którym znajduje się kolejny element – listing 10.2. Na listingu 10.3 przedstawiona została struktura zawierająca początek i koniec kolejki (opakowanie wskaźników *head* and *tail*). W programie głównym należy utworzyć zmienną, która będzie przechowywała kolejkę – listing 10.4.

```
1 struct element{
2     char character;
3     element* next; };
```

Listing 10.2. Implementacja struktury przechowującej element w kolejce

```
1 struct queue{
2     element* head;
3     element* tail; };
```

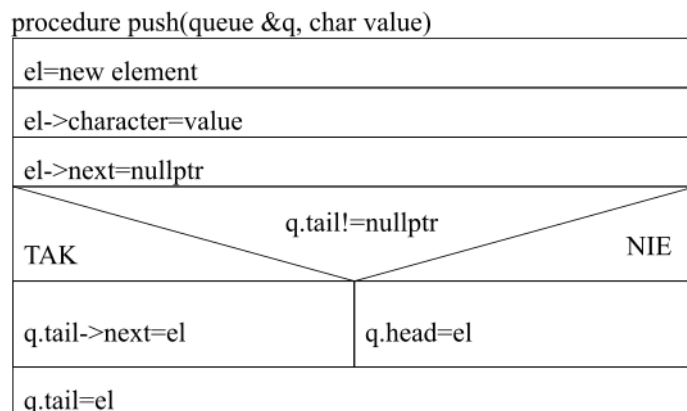
Listing 10.3. Implementacja struktury przechowującej początek i koniec kolejki

```
1 queue q;
2 q.head=nullptr;
3 q.tail=nullptr;
```

Listing 10.4. Utworzenie zmiennej przechowującej kolejkę

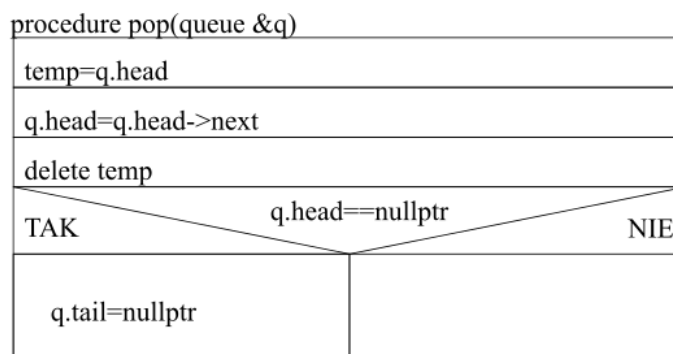
Schematy zwarte NS:

- na rys. 10.4 przedstawiono schemat zwarte NS operacji **push** – dodania elementu do kolejki:



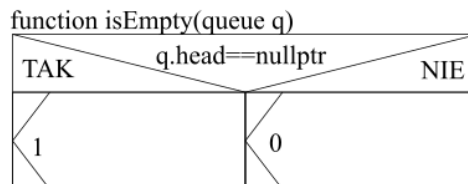
Rys. 10.4. Operacja push – dodanie elementu do kolejki

- na rys. 10.5 przedstawiono schemat zwarte NS operacji **pop** – usunięcia elementu z kolejki:



Rys. 10.5. Operacja pop – usunięcie elementu z kolejki

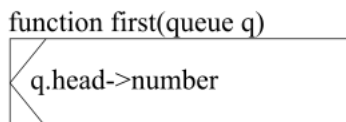
- na rys. 10.6 przedstawiono schemat zwarte NS operacji sprawdzenia czy kolejka jest pusta – **isEmpty**:



Rys. 10.6. Sprawdzenie czy kolejka jest pusta

- na rys. 10.7 przedstawiono schemat zwarte NS operacji **first** – pobranie pierwszego elementu z kolejki:





Rys. 10.7. Operacja *first* – pobranie pierwszego elementu z kolejki

Zadanie do wykonania:

Zadanie 10.1 Kolejka

Stwórz strukturę zawierającą wskaźnik na kolejny element kolejki oraz znak w celu zaimplementowania kolejki. Następnie stwórz strukturę kolejka, która będzie zawierała dwa wskaźniki: *head* oraz *tail*.

W programie głównym należy stworzyć menu wielokrotnego wyboru oraz zaimplementować funkcje:

- 1) sprawdzenie czy kolejka jest pusta,
- 2) powinien zostać wyświetlony odpowiedni komunikat.
- 3) dodanie elementu do kolejki,
należy wylosować znak, który zostanie dodany do kolejki.
- 4) usunięcie elementu z kolejki,
jeśli w kolejce nie ma żadnego elementu należy wyświetlić stosowny komunikat, na przykład "Kolejka jest pusta, nie da się usunąć elementu".
- 5) pobranie elementu z kolejki,
należy wyświetlić pierwszy znak, który znajduje się w kolejce.
- 6) usunięcie wszystkich elementów z kolejki,
podczas usuwania kolejnych elementów należy wyświetlić element, który jest usuwany.
- 7) wczytanie do kolejki dużych liter alfabetu łacińskiego z pliku tekstowego, plik powinien znajdować się w katalogu projektu. Nazwa pliku – „znaki2.txt” powinna zostać przekazana jako argument. Struktura pliku została przedstawiona na rysunku 10.8. Do kolejki powinny zostać zapisane tylko te znaki, które są dużymi literami alfabetu łacińskiego.
- 8) wyjście z programu

```
znaki2.txt — Notatnik
Plik Edycja Format Widok Pomoc
abD )! GFD - ===+ABCDE
Ke g H D T Y ### 876 - R
M A ** B ! @
alGorYtmy i strUktury dAnyh!
```

Rys. 10.8. Plik *znaki2.txt*

Podpowiedź do podpunktu 6:

Można czytać znak po znaku. Należy sprawdzić czy wczytany znak jest dużą literą alfabetu. Korzystając z kodów tabeli ASCII, wiemy, że duże litery alfabetu łacińskiego mają kody od 65 do 90 (A-Z). Za pomocą rzutowania zmiennej przechowującej znak na liczbę całkowitą otrzymamy kod znaku w tabeli ASCII. Następnie należy sprawdzić, czy uzyskana liczba zawiera się w wyżej wspomniany przedziale.

LABORATORIUM 11. IMPLEMENTACJA LIST

Cel laboratorium:

Omówienie struktur danych – list.

Zakres tematyczny zajęć:

- operacje na listach,
- implementacja list,
- rodzaje list.

Pytania kontrolne:

- 1) Co to jest lista?
- 2) Jakie operacje można wykonać na liście?
- 3) Jakie są rodzaje list?
- 4) Czym różni się lista cykliczna dwukierunkowa od listy dwukierunkowej?
- 5) Czym różni się lista cykliczna jednokierunkowa od listy jednokierunkowej?

Lista:

Jest to struktura danych, która umożliwia sekwencyjny dostęp do elementów. Z jednego elementu można przejść do drugiego. Listy stosowane są do reprezentacji w pamięci komputera danych sekwencyjnych na przykład: grafów, kolejek, stosów. Rozróżniane są trzy rodzaje list: lista jednokierunkowa, lista dwukierunkowa oraz lista cykliczna: jednokierunkowa oraz dwukierunkowa.

Operacje wykonywane na listach:

- dodawanie elementu do listy:
 - na początek listy(*add_head*),
 - na koniec listy(*add_tail*),
 - na określoną pozycję(*add_position*).
- usuwanie elementu z listy:
 - z początku listy(*delete_head*),
 - z końca listy(*delete_last*),
 - z określonej pozycji(*delete_position*).
- sprawdzanie czy lista jest pusta (*isEmpty*)
- wyświetlenie wszystkich elementów w liście (*show*),
- odczytanie elementu z początku listy(*first*),
- odczytanie elementu z końca listy(*last*).

Lista jednokierunkowa:

Pojedynczy element listy jednokierunkowej w języku C++ jest implementowany w postaci struktury, składającej się z następujących pól:

- pole/pola z danymi,

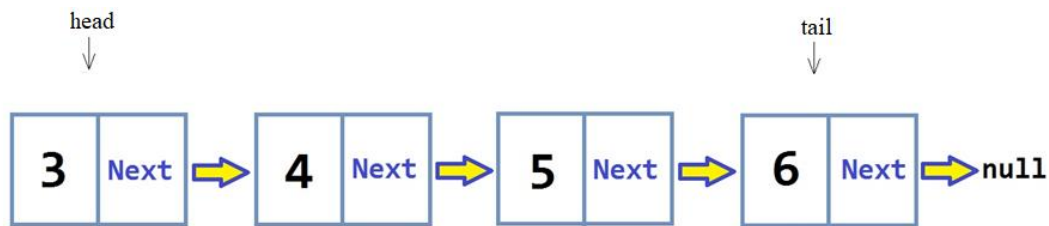


- adres elementu, znajdującego się na kolejnej pozycji.

W celu utworzenia struktury listy jednokierunkowej potrzebne są dwa wskaźniki:

- *head* – wskaźnik na pierwszy element znajdujący się w liście, przechowuje adres pierwszego elementu z listy,
- *tail* – wskaźnik na ostatni element znajdujący się w liście, przechowuje adres ostatniego elementu w liście.

Poniżej na rysunku 11.1 przedstawiony został schemat listy jednokierunkowej. Struktura składa się pola przechowującego liczbę całkowitą oraz z pola *next*, które przechowuje adres kolejnego elementu znajdującego się w liście. Ostatni element, który znajduje się w liście w polu *next* ma NULL. Zmienna *head* przechowuje początek listy. Zmienna *tail* przechowuje koniec listy.



Rys. 11.1. Schemat listy jednokierunkowej

Poniżej przedstawione zostaną schematy zwarte NS związane z wykonywaniem operacji na liście jednokierunkowej. W celu lepszego zrozumienia działania poniższych funkcji zdefiniowana została następująca struktura – zawiera ona dwa pola – pierwsze pole przechowuje zmienną typu całkowitego a drugie pole zawiera wskaźnik na kolejny element – adres, pod którym znajduje się kolejny element – listing 11.2. Na listingu 11.3. przedstawiona została struktura zawierająca początek i koniec listy (opakowanie wskaźników *head* i *tail*). Dodatkową często praktyką jest umieszczanie w strukturze zmiennej przechowującej liczbę elementów aktualnie znajdujących się w liście w tym przypadku zmienna *counter*. Nie jest to konieczne ale ułatwi pracę z listą. W programie głównym należy utworzyć zmienną, która będzie przechowywała pola: *head* oraz *tail* – listing 11.4.

```

1 struct element{
2     int number;
3     element* next; };
    
```

Listing 11.2. Implementacja struktury przechowującej jeden element w liście jednokierunkowej

```

1 struct single_list{
2     element* head;
3     element* tail;
4     int counter};
    
```

Listing 11.3. Implementacja struktury przechowującej początek i koniec listy jednokierunkowej



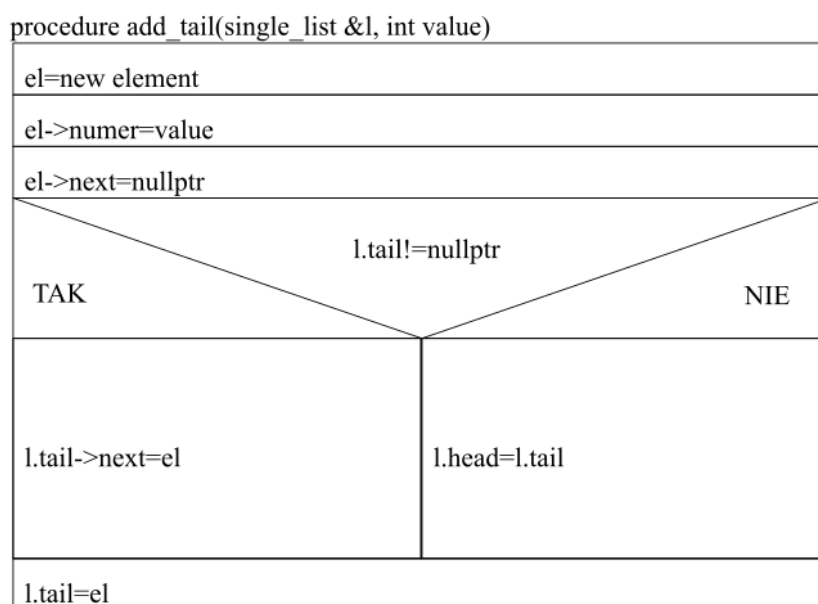
```
1 single_list l;
```

Listing 11.4. Utworzenie zmiennej przechowującej listę jednokierunkową.

Schematy zwarte NS dla listy jednokierunkowej :

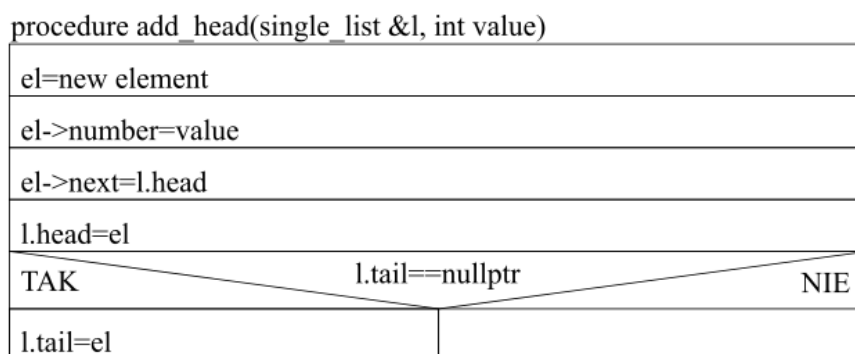
Schematy należy uzupełnić o inkrementację lub dekrementację pola *counter* w strukturze.

- Na rys. 11.2 przedstawiono schemat zwarty NS operacji dodawania elementu na koniec listy(*add_tail*).



Rys. 11.2. Schemat zwarty NS – dodawanie elementu na koniec listy jednokierunkowej

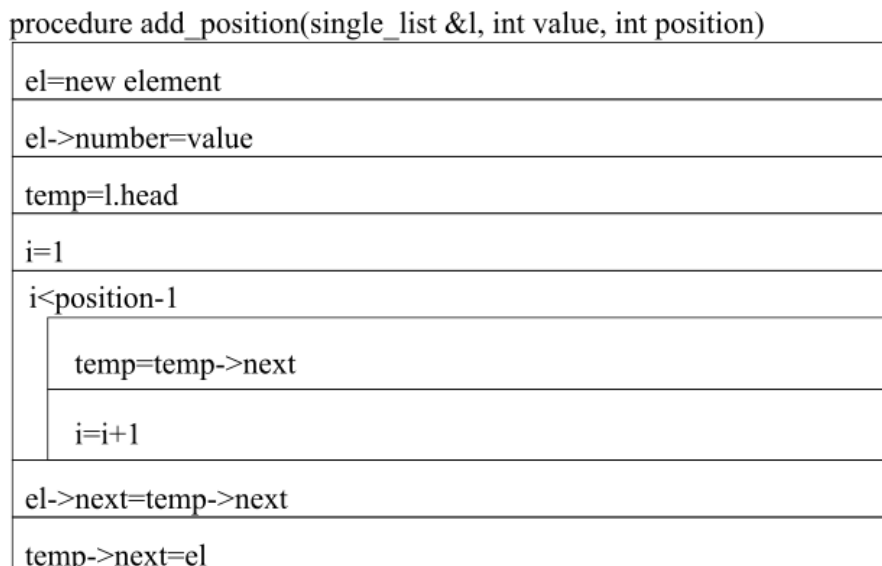
- Na rys. 11.3 przedstawiono schemat zwarty NS operacji dodawania elementu na początek listy(*add_head*).



Rys. 11.3. Schemat zwarty NS – dodawanie elementu na początek listy jednokierunkowej

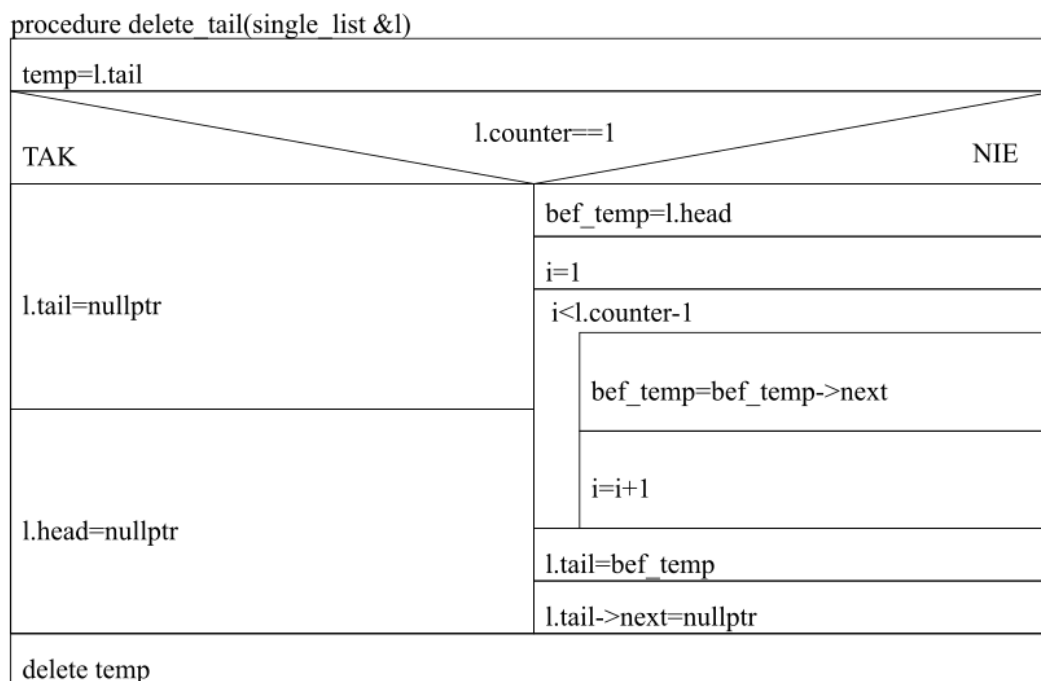
- Na rys. 11.4 przedstawiono schemat zwarty NS operacji dodawania elementu na określoną pozycję, która nie jest ani pierwszą ani ostatnią(*add_position*).





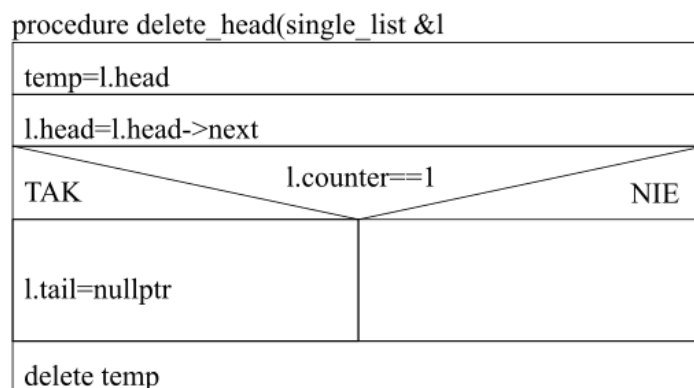
Rys. 11.4. Schemat zwarty NS – dodawanie elementu na określoną pozycję w liście jednokierunkowej

- Na rys. 11.5 przedstawiono schemat zwarty NS operacji usuwania elementu z końca listy (*delete_tail*).



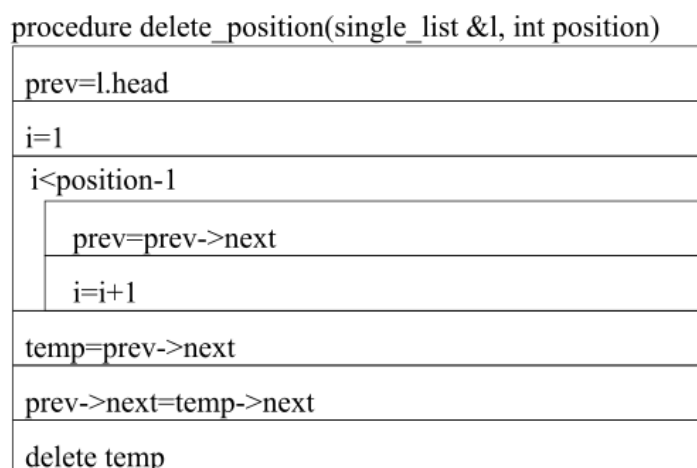
Rys. 11.5. Schemat zwarty NS – usuwanie elementu z końca listy jednokierunkowej

- Na rys. 11.6 przedstawiono schemat zwarty NS operacji usuwania elementu z początku listy(*delete_head*).



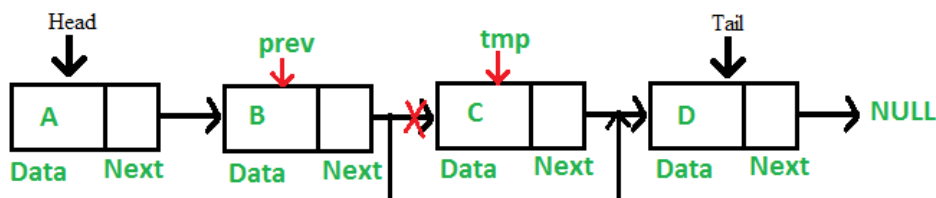
Rys. 11.6. Schemat zwarty NS – usuwanie elementu z początku listy jednokierunkowej

- Na rys. 11.2 przedstawiono schemat zwarty NS operacji usuwania elementu z pozycji, która nie jest ani pierwszą ani ostatnią(*delete_position*)



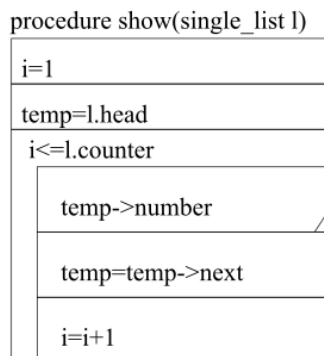
Rys. 11.7. Schemat zwarty NS – usuwanie elementu z określonej pozycji w liście jednokierunkowej

Na rys. 11.8 zaprezentowany został rysunek przedstawiający usuwanie elementu z określonej pozycji.



Rys. 11.8. Schemat usuwania elementu z określonej pozycji listy jednokierunkowej

- Na rys. 11.2 przedstawiono schemat zwarty NS operacji wyświetlenia wszystkich elementów listy(*show*).



Rys. 11.9. Schemat wyświetlania wszystkich elementów listy jednokierunkowej

Lista dwukierunkowa:

Jedyną różnicą pomiędzy listą jednokierunkową a dwukierunkową jest możliwość wyświetlenia elementów zarówno od początku do końca, jak i od końca do początku (przejście w dwóch kierunkach).

Pojedynczy element listy dwukierunkowej w języku C++ jest implementowany w postaci struktury, składającej się z następujących pól:

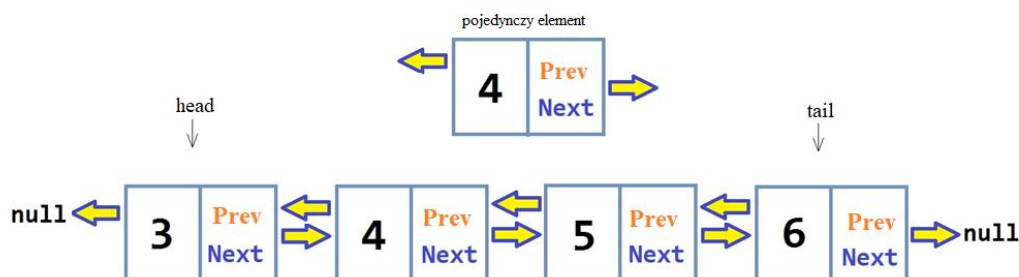
- pole/pola z danymi,
- adres elementu, znajdującego się na kolejnej pozycji,
- adres elementu, znajdującego się na poprzedniej pozycji.

W celu utworzenia struktury listy dwukierunkowej potrzebne są dwa wskaźniki:

- *head* – wskaźnik na pierwszy element znajdujący się w liście, przechowuje adres pierwszego elementu z liście,
- *tail* – wskaźnik na ostatni element znajdujący się w liście, przechowuje adres ostatniego elementu w liście.

Poniżej na rysunku 11.3. przedstawiony został schemat listy dwukierunkowej. Struktura składa się z pola przechowującego liczbę całkowitą oraz z pola *next*, które przechowuje adres następnego elementu znajdującego się w liście oraz *prev*, które przechowuje adres poprzedniego elementu znajdującego się w liście. Ostatni element, który znajduje się na liście w polu *next* ma NULL. Pierwszy element, który znajduje się w liście w polu *prev* ma NULL.

Zmienna *head* przechowuje początek listy. Zmienna *tail* przechowuje koniec listy.



Rys. 11.10. Schemat listy dwukierunkowej

Poniżej przedstawione zostaną schematy zwarte NS związane z wykonywaniem operacji na liście dwukierunkowej. W celu lepszego zrozumienia działania poniższych funkcji zdefiniowana została następująca struktura – zawiera ona trzy pola – pierwsze pole przechowuje zmienną typu całkowitego a drugie pole zawiera wskaźnik na kolejny element – adres, pod którym znajduje się następny element a trzecie pole zawiera wskaźnik – adres poprzedniego elementu – listing 11.5 Na listingu 11.6 przedstawiona została struktura zawierająca początek i koniec listy (opakowanie wskaźników *head* and *tail*). Tak jak w przypadku listy jednokierunkowej, do struktury „opakowującej” wskaźniki *head* oraz *tail* dodana została zmienna *counter*. W programie głównym należy utworzyć zmienną, która będzie przechowywała pola: *head*, *tail* oraz *counter* – listing 11.7.

```
1 struct element{
2     int number;
3     element* next;
4     element* prev; };
```

Listing 11.5. Implementacja struktury przechowującej jeden element w liście dwukierunkowej

```
1 struct double_list{
2     element* head;
3     element* tail;
4     int counter; };
```

Listing 11.6. Implementacja struktury przechowującej początek i koniec listy dwukierunkowej

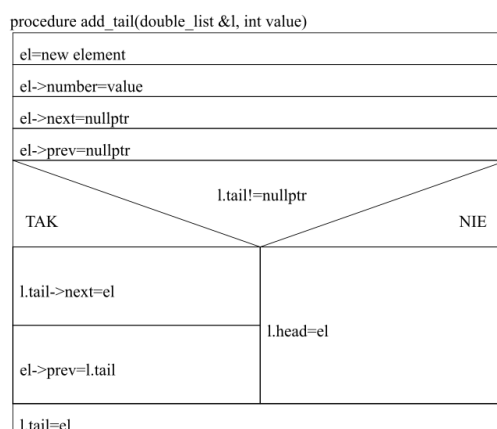
```
1 double_list l;
```

Listing 11.7. Utworzenie zmiennej przechowującej listę dwukierunkową

Na rysunkach 11.11 – 11.13 przedstawione są schematy zwarte NS operacji dodawania elementu na różne pozycje listy dwukierunkowej.

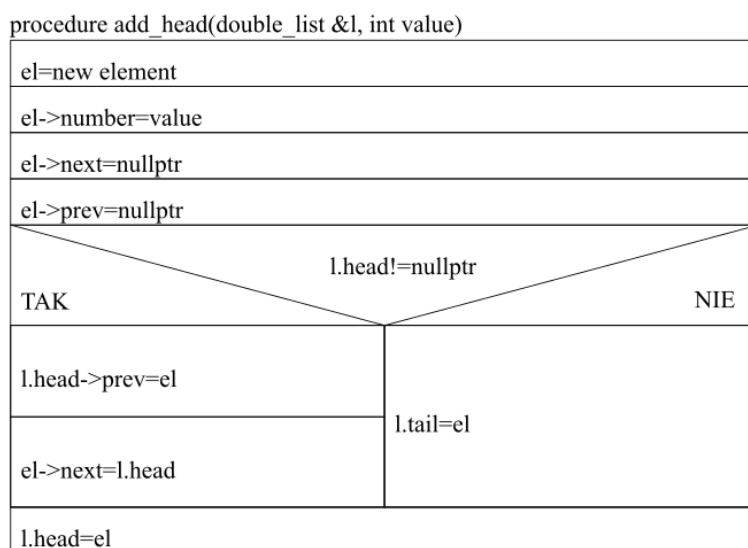
Schematy należy uzupełnić o inkrementację lub dekrementację pola *counter* w strukturze.

- dodawanie elementu na koniec listy(*add_tail*),



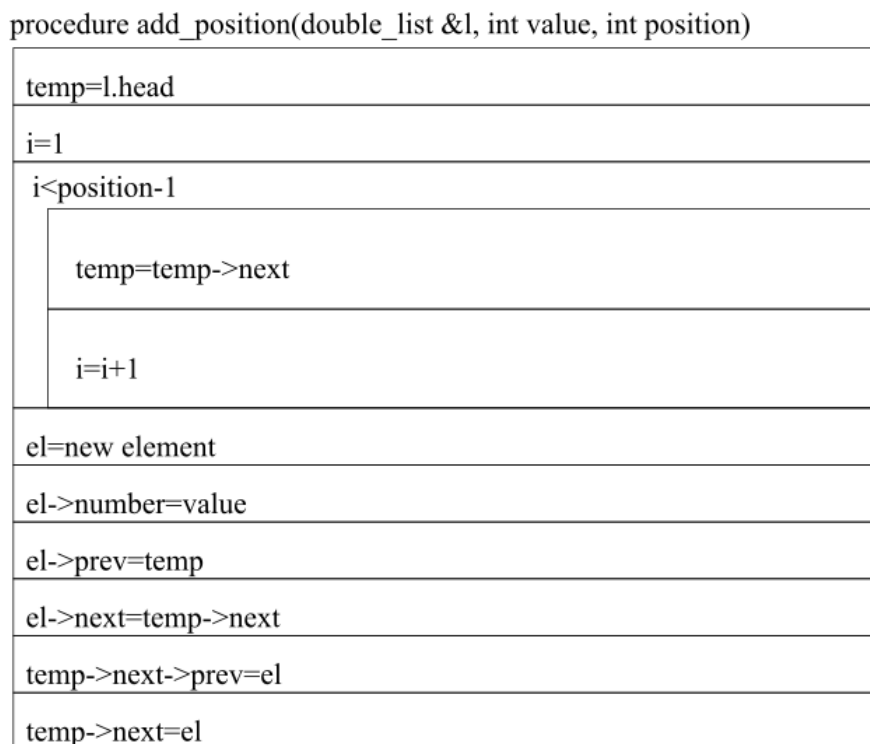
Rys. 11.11. Schemat zwarty NS – dodawanie elementu na koniec listy dwukierunkowej

- dodawanie elementu na początek listy(***add_head***),



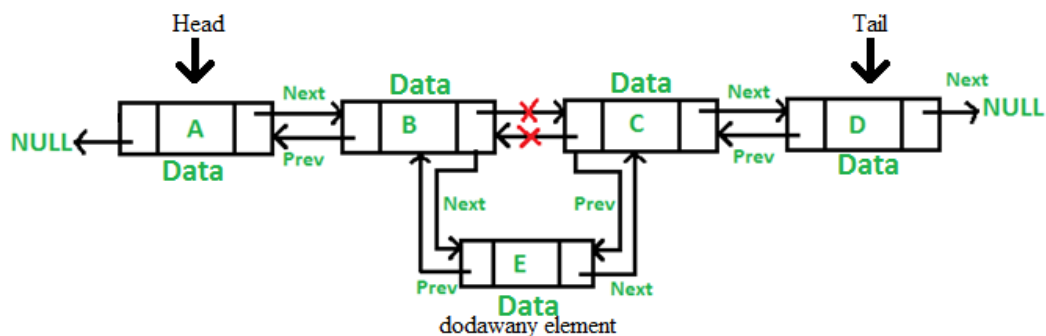
Rys. 11.12. Schemat zwarty NS – dodawanie elementu na początek listy dwukierunkowej

- dodawanie elementu na określoną pozycją, która nie jest ani pierwszą ani ostatnią(***add_position***),



Rys. 11.13. Schemat zwarty NS – dodawanie elementu na określoną pozycję w liście dwukierunkowej

Na rysunku 11.14 zaprezentowany został rysunek przedstawiający dodawanie elementu na określoną pozycję.



Rys. 11.14. Schemat dodawania elementu na określoną pozycję w liście dwukierunkowej

Na rysunkach 11.11 – 11.13 przedstawione są schematy zwarte NS operacji usuwania elementu z różnych pozycji listy dwukierunkowej.

- usuwanie elementu z końca listy(*delete_tail*)

procedure delete_tail(double_list &l)	
temp=l.tail	
TAK NIE $l.counter == 1$	
$l.tail = \text{nullptr}$	$l.tail = l.tail \rightarrow prev$
$l.head = \text{nullptr}$	$l.tail \rightarrow next = \text{nullptr}$
delete temp	

Rys. 11.15. Schemat zwarty NS – usuwanie elementu z końca listy dwukierunkowej

- usuwanie elementu z początku listy(*delete_head*)

procedure delete_head(double_list &l)	
temp=l.head	
TAK NIE $l.counter == 1$	
$l.tail = \text{nullptr}$	$l.head = l.head \rightarrow next$
$l.head = \text{nullptr}$	$l.head \rightarrow prev = \text{nullptr}$
delete temp	

Rys. 11.16. Schemat zwarty NS – usuwanie elementu z początku listy dwukierunkowej

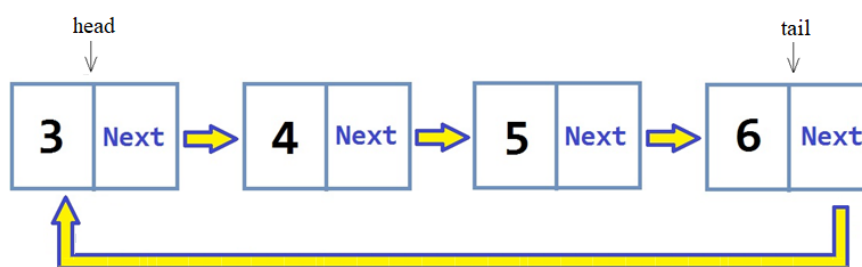
- usuwanie elementu z pozycji, która nie jest ani pierwszą ani ostatnią(*delete_position*)

procedure delete_position(double_list &l, int position)
temp=l.head
i=1
i<position-1
temp=temp->next
i=i+1
temp_us=temp->next
temp->next=temp_us->next
temp->next->prev=temp
delete temp_us

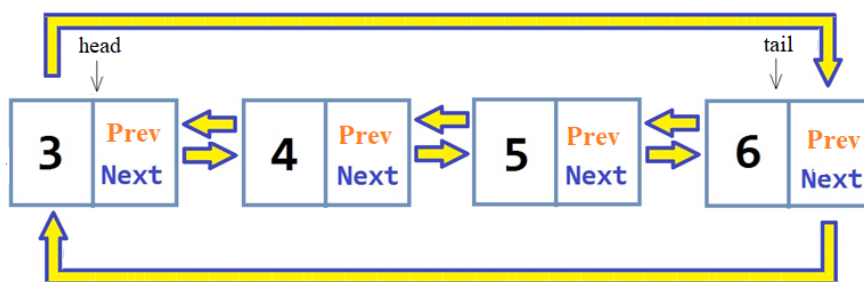
Rys. 11.17. Schemat zwarty NS – usuwanie elementu z określonej pozycji w liście jednokierunkowej

Lista cykliczna:

Lista cykliczna może być albo listą jednokierunkową albo dwukierunkową. Lista cykliczna jednokierunkowa różni się od listy jednokierunkowej tym, że w liście jednokierunkowej ostatni element przechowuje adres NULL, a w liście jednokierunkowej cyklicznej ostatni element przechowuje adres pierwszego elementu w liście. Natomiast lista cykliczna dwukierunkowa różni się od listy dwukierunkowej tym, że pierwszy element w polu *prev* przechowuje adres NULL i ostatni element w polu *next* również przechowuje NULL, a w liście dwukierunkowej cyklicznej, pierwszy element w polu *prev* przechowuje adres ostatniego elementu, a ostatni element w polu *next* przechowuje adres pierwszego elementu w liście. Na rysunku 11.5. przedstawiony został schemat listy cyklicznej jednokierunkowej, a na rysunku 11.6. przedstawiony został schemat listy cyklicznej dwukierunkowej.



Rys. 11.5. Schemat listy jednokierunkowej cyklicznej



Rys. 11.6. Schemat listy dwukierunkowej cyklicznej.

Zadania do wykonania:

Podczas wykonywania zadań należy pamiętać, aby uzupełnić schematy zwarte dla poszczególnych funkcji o inkrementację lub dekrementację pola *counter* w strukturze. Dodatkowo należy pamiętać, aby rozpatrzyć przypadki szczególne takie, jak: dodanie na pozycję pierwszą, dodanie na pozycję ostatnią, usunięcie z pozycji pierwszej, usunięcie z pozycji ostatniej. W przypadku zaistnienia, którejś z sytuacji: usunięcie z pustej listy, dodanie na pozycję, która nie istnieje, usunięcie z pozycji która nie istnieje, należy wyświetlić stosowny komunikat. Przed i po wykonaniu każdej operacji dodania lub usunięcia elementu należy wyświetlić całą listę.

Zadanie 11.1 Lista jednokierunkowa

Stwórz strukturę *Element* zawierającą wskaźnik na kolejny element kolejki oraz liczbę całkowitą.

Następnie stwórz strukturę *Single_list* zawierającą wskaźniki na początek i koniec listy: *head* oraz *tail*. Dodatkowo dodaj do struktury pole typu całkowitego przechowujące liczbę elementów w liście: *counter*. Liczbę, która powinna być dodana do listy, należy wylosować z zakresu 1 - 50.

W programie głównym należy stworzyć wskaźnik na listę oraz menu wielokrotnego wyboru oraz zaimplementować funkcje:

- 1) sprawdzenie czy lista jest pusta,
- 2) należy wyświetlić stosowny komunikat.
- 3) dodanie elementu na koniec listy,
- 4) dodanie elementu na początek listy,
- 5) dodanie elementu na określoną pozycję,
- 6) usunięcie elementu z końca listy,
- 7) usunięcie elementu z początku listy,
- 8) usunięcie elementu znajdującego się na określonej pozycji ,
- 9) pobranie pierwszego elementu z listy.
należy wyświetlić pierwszy element, który znajduje się w liście.
- 10) pobranie ostatniego elementu z listy.
- 11) należy wyświetlić ostatni element, który znajduje się w liście.
- 12) policzenie średniej arytmetycznej elementów w liście,
należy wyświetlić średnią arytmetyczną elementów w liście.
- 13) znalezienie elementu maksymalnego w liście,
należy wyświetlić element maksymalny wraz z podaniem pozycji na której się znajduje.

- 14) wyświetlenie całej listy,
- 15) usunięcie całej listy wraz ze zwolnieniem pamięci,
- 16) wyjście z programu.

Zadanie 11.2 Lista dwukierunkowa

Stwórz strukturę *Element* zawierającą wskaźniki na kolejny, poprzedni element listy oraz liczbę całkowitą. Następnie stwórz strukturę *Double_List* zawierającą wskaźniki na początek i koniec listy: *head* oraz *tail*. Dodatkowo dodaj do struktury pole typu całkowitego przechowujące liczbę elementów w liście: *counter*. Liczbę, która powinna być dodana do listy, należy wylosować z zakresu 1 - 50.

W programie głównym należy stworzyć wskaźnik na listę oraz menu wielokrotnego wyboru oraz zaimplementować funkcje:

- 1) sprawdzenie czy lista jest pusta. Należy wyświetlić stosowny komunikat.
- 2) dodanie elementu na koniec listy,
- 3) dodanie elementu na początek listy,
- 4) dodanie elementu na pozycję o podanym numerze,
- 5) usunięcie elementu z końca listy,
- 6) usunięcie elementu z początku listy,
- 7) usunięcie elementu znajdującego się na określonej pozycji,
- 8) wyświetlenie elementów od początku do końca,
- 9) wyświetlenie elementów od końca do początku,
- 10) znalezienie elementu minimalnego w liście,
należy wyświetlić element minimalny wraz z podaniem pozycji na której się znajduje.
- 11) usunięcie całej listy wraz ze zwolnieniem pamięci,
- 12) wyjście z programu.

Zadanie 11.3 Lista jednokierunkowa cykliczna

Należy wykonać zadanie 11.1. dla listy cyklicznej jednokierunkowej.

Zadanie 11.4 Lista dwukierunkowa cykliczna

Należy wykonać zadanie 11.2. dla listy cyklicznej dwukierunkowej.

LABORATORIUM 12. DRZEWA BST

Cel laboratorium:

Omówienie struktury danych – drzewa BST.

Zakres tematyczny zajęć:

- operacje na drzewie,
- struktura drzewa.

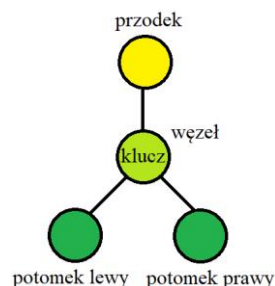
Pytania kontrolne:

- 1) Co to jest drzewo BST?
- 2) Z jakich elementów składa się jeden węzeł drzewa BST?
- 3) Jak nazywają się ostatnie węzły w drzewie?
- 4) Jak nazywa się element nie mający rodzica?
- 5) Jakie są najczęstsze operacje na drzewie?
- 6) Jakie są sposoby przejścia po drzewie?
- 7) Czy różni się prawe od lewego poddrzewa?
- 8) Co to jest następnik danego węzła?
- 9) Jakie przypadki są wyróżniane podczas usuwania węzła?

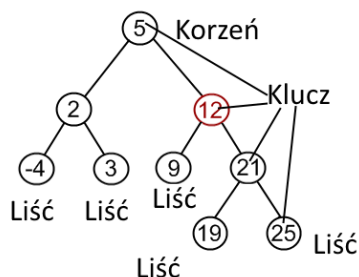
Drzewo BST:

Drzewo poszukiwań binarnych jest strukturą danych, która zbudowana jest z węzłów, w której każdy element może mieć dwóch potomków: lewego i prawego, nazywanych dziećmi tego elementu. Ten element z kolei jest dla nich przodkiem/rodzicem.

Na rys. 12.1. przedstawiony został schemat połączeń między węzłami. Kolorowe kółka to węzły, wartości wpisane w kółka to klucze. Każdy węzeł posiada przodka oraz maksymalnie dwóch potomków (prawego i lewego). Wyjątkiem jest pierwszy węzeł, który nie posiada przodka nazywany jest korzeniem, oraz ostatnie węzły, które nie posiadają potomków nazywane są liśćmi. Taka sytuacja została przedstawiona na rysunku 12.2.



Rys. 12.1. Schemat połączeń między węzłami

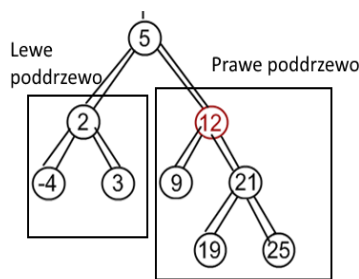


Rys. 12.2. Schemat drzewa BST

Dla każdego węzła w drzewie BST muszą być spełnione następujące warunki:

- wartości kluczy wszystkich elementów prawego poddrzewa są większe bądź równe wartości klucza danego elementu;
- wartości kluczy wszystkich elementów jego lewego poddrzewa są mniejsze od wartości klucza danego elementu.

Na rys. 12.3 pokazany został podział na prawe i lewe poddrzewo w drzewie BST.

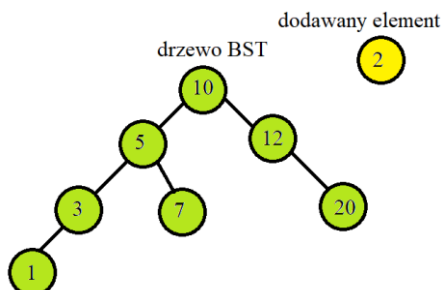


Rys. 12.3. Schemat prawego i lewego poddrzewa BST

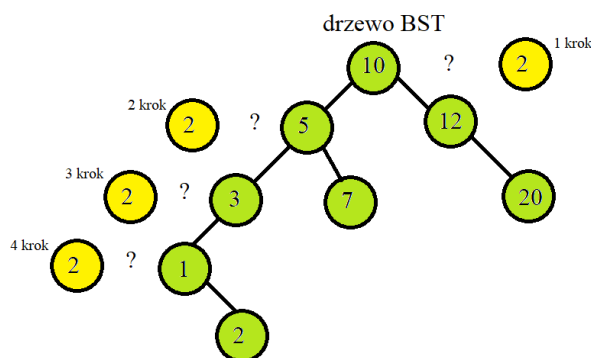
Poniżej wypisane zostały najpopularniejsze operacje na drzewie BST:

- dodawanie węzła do drzewa (add_node),
- wyszukiwanie elementu w drzewie (find_node),
- usuwanie węzła (delete_node),
- przechodzenie przez drzewo (inorder, postorder, preorder).

Rys. 12.4 oraz 12.5 przedstawiają dodawanie elementu do drzewa. Na żółto zaznaczony został dodawany element. Znak zapytania obrazuje sprawdzenie, czy dodawany element jest większy bądź równy czy mniejszy od porównywanego elementu.



Rys. 12.4. Stan drzewa BST przed dodaniem elementu



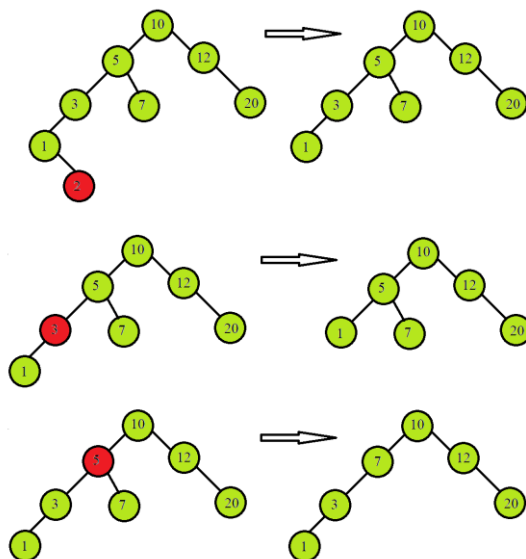
Rys. 12.5. proces dodawania elementu do drzewa BST

Poniżej przedstawione są trzy sposoby przejścia po drzewie przedstawionym na rys. 12.4.

- Preorder – algorytm zaczyna przechodzić przez węzeł, lewą gałąź drzewa BST, a następnie prawą gałąź drzewa BST: 10 5 3 1 2 7 12 20.
- Inorder – algorytm zaczyna przechodzić przez lewą gałąź drzewa BST, węzeł a następnie prawą gałąź drzewa BST. Klucze odwiedzanych węzłów tworzą ciąg niemalejący: 1 2 3 5 7 10 12 20.
- Postorder – algorytm zaczyna przechodzić przez lewą gałąź drzewa BST, prawą gałąź drzewa BST, a następnie węzeł: 2 1 3 7 5 20 12 10.

Rys. 12.5 przedstawia usuwanie wybranego węzła z drzewa. Podczas usuwania węzła z drzewa wyróżnia się trzy przypadki.

- usuwany element może być liściem – nie trzeba przebudowywać drzewa,
- jeżeli usuwany węzeł ma jednego syna to dany węzeł jest usuwany a jego syna podstawiamy w miejsce usuniętego węzła,
- jeżeli usuwany węzeł ma dwóch synów to po jego usunięciu w jego miejsce wstawiany jest węzeł, który znajduje się na skrajnie lewej pozycji w prawym poddrzewie – węzeł z najmniejszą wartością klucza.



Rys. 12.5. Usuwanie węzła z drzewa

Pojedynczy element drzewa BST w języku C++ jest implementowany w postaci struktury, składającej się z następujących pól:

- pole/pola z danymi,
- adresu przodka,
- adresu prawego potomka,
- adresu lewego potomka.

Poniżej przedstawione zostaną schematy zwarte NS związane z wykonywaniem operacji na drzewie BST. W celu lepszego zrozumienia działania poniższych funkcji zdefiniowana została następująca struktura – zawiera ona cztery pola – pierwsze pole przechowuje zmienną typu całkowitego, drugie pole zawiera wskaźnik na przodka, trzecie pole zawiera wskaźnik na lewego potomka i czwarte pole zawiera wskaźnik na prawego potomka – listing 12.1. W programie głównym należy utworzyć zmienną, która będzie przechowywała drzewo – listing 12.2.

```
1 struct node{
2     int key;
3     node* parent;
4     node* left;
5     node* right; };
```

Listing 12.1. Implementacja struktury przechowującej jeden węzeł w drzewie BST

```
1 node* tree;
```

Listing 12.2. Utworzenie zmiennej przechowującej drzewo

Na rysunkach 12.6 – 12.13 przedstawione zostały schematy zwarte NS podstawowych operacji związanych z drzewem BST:

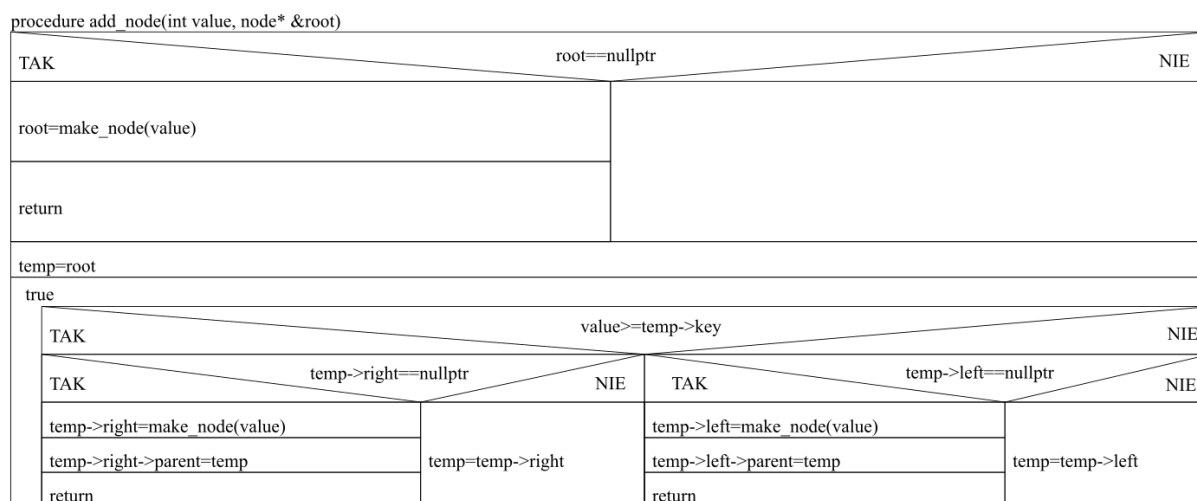
- tworzenie węzła (*make_node*)

```
function make_node(int value)
```

n=new node
n->key=value
n->left=nullptr
n->right=nullptr
n->parent=nullptr
<n

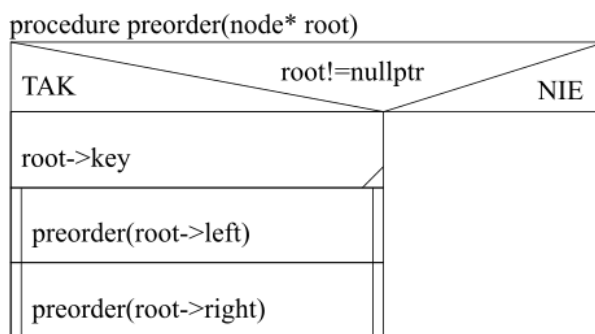
Rys. 12.6. Tworzenie węzła

- dodawanie węzła (*add_node*)



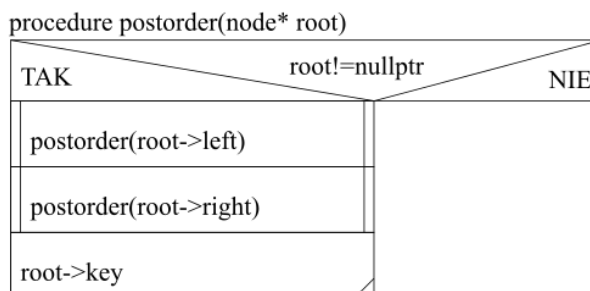
Rys. 12.7. Dodawanie węzła do drzewa BST

- przejście po drzewie *preorder*



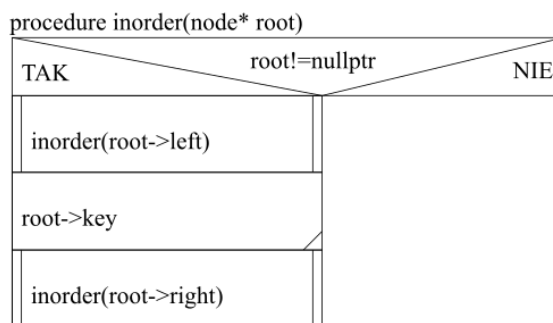
Rys. 12.8. Przejście po drzewie preorder

- przejście po drzewie *postorder*



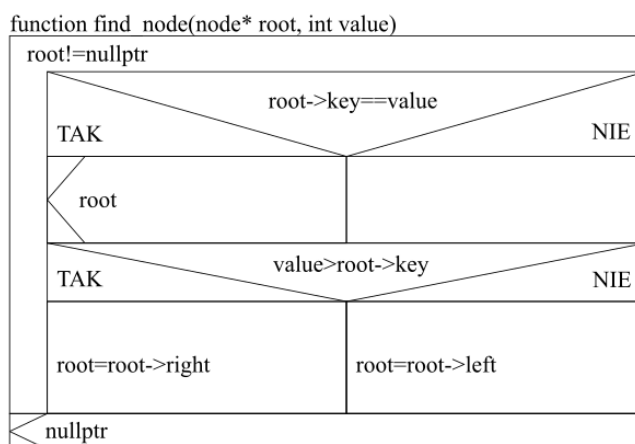
Rys. 12.9. Przejście po drzewie postorder

- przejście po drzewie **inorder**



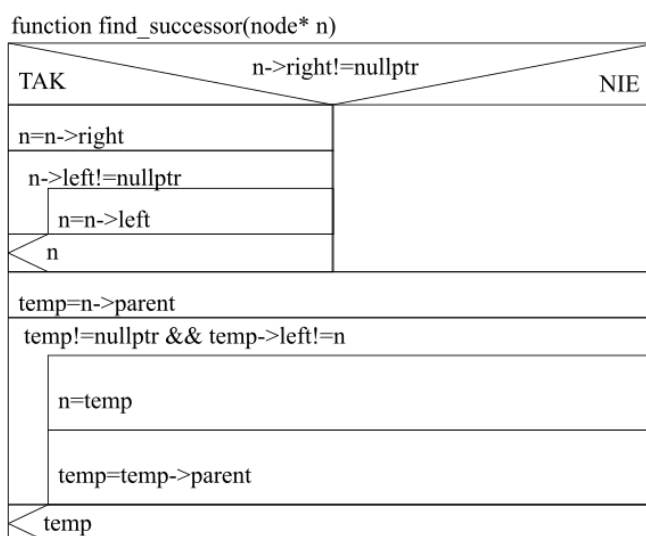
Rys. 12.10. Przejście po drzewie *inorder*

- wyszukiwanie elementu



Rys. 12.11. Wyszukiwanie zadanej wartości klucza w drzewie

- znajdowanie następnika w drzewie (**find_successor**)– następnik danego węzła jest to węzeł, który zostanie odwiedzony jako następny podczas przejścia **inorder**.



Rys. 12.12. Znajdowanie następnika w drzewie

- usuwanie węzła

procedure delete_node(node* &root, node* delete_node)

delete_node->left==nullptr delete_node->right==nullptr		TAK	NIE
temp1=delete_node	temp1=find_successor(delete_node)		
temp1->left!=nullptr		TAK	NIE
temp2=temp1->left	temp2=temp1->right		
temp2!=nullptr		TAK	NIE
temp2->parent=temp1->parent			
temp1->parent==nullptr		TAK	NIE
root=temp2	temp1==temp1->parent->left	TAK	NIE
	temp1->parent->left=temp2		
	temp1->parent->right=temp2		
temp1!=delete_node		TAK	NIE
delete_node->key=temp1->key			
delete temp1			

Rys. 12.13. Usuwanie węzła w drzewie

Zadania do wykonania:

Zadanie 12.1

Stwórz strukturę *Node* zawierającą wskaźniki: *right*, *left*, *parent* oraz *key* typu całkowitego
W programie głównym należy stworzyć wskaźnik *root* na strukturę *Node* oraz menu wielokrotnego wyboru:

- 1) sprawdzenie czy drzewo jest puste.
Powinien zostać wyświetlony odpowiedni komunikat.
- 2) dodanie nowego węzła do drzewa.
Wartość klucza powinna zostać podana przez użytkownika. Należy dodać nowy węzeł jeśli węzeł o podanej wartości klucza nie istnieje w drzewie, w przeciwnym wypadku należy wyświetlić stosowny komunikat i nie dodawać węzła do drzewa. Należy wyświetlić drzewo (*inorder*) przed dodaniem, oraz po dodaniu węzła.
- 3) sprawdzenie czy klucz o podanej wartości węzła przez użytkownika znajduje się w drzewie. Powinien zostać wyświetlony stosowny komunikat.
- 4) wyświetlenie liczby wystąpień kluczy w drzewie,
- 5) wyświetlenie drzewa – *preorder*,
- 6) wyświetlenie drzewa – *inorder*,
- 7) wyświetlenie drzewa – *postorder*,
- 8) usunięcie węzła o podanej wartości klucza przez użytkownika
Jeśli podanego klucza nie ma w drzewie powinien zostać wyświetlony stosowny komunikat, w przeciwnym razie, węzeł powinien zostać usunięty. Należy wyświetlić drzewo (*inorder*) przed usunięciem, oraz po usunięciu węzła.
- 9) usunięcie całego drzewa,
- 10) wyjście z programu.

LABORATORIUM 13. SORTOWANIE PRZEZ KOPCOWANIE

Cel laboratorium:

Omówienie sortowania przez kopcowanie.

Zakres tematyczny zajęć:

- kopiec,
- sortowanie przez kopcowanie.

Pytania kontrolne:

- 1) Co to jest kopiec?
- 2) Jaka zasada musi być spełniona, żeby drzewo binarne było kopcem?
- 3) Jaka jest złożoność czasowa sortowania przez kopcowanie?
- 4) Jaka jest zasada rozbierania kopca?

Sortowanie przez kopcowanie:

Sortowanie polega na zbudowaniu kopca oraz rozebraniu kopca. W wyniku rozebrania kopca dane zostają posortowane (rosnąco lub malejąco). Sortowanie przez kopcowanie można zrealizować w oparciu o tablicę. W tablicy budowany jest kopiec, a następnie elementy są zdejmowane z kopca – proces rozbierania kopca można uznać za właściwe sortowanie. Poniżej przedstawiony zostanie algorytm tworzenia kopca oraz rozbierania kopca. Złożoność czasowa algorytmu sortowania wynosi $O(n\log_2 n)$.

Kopiec:

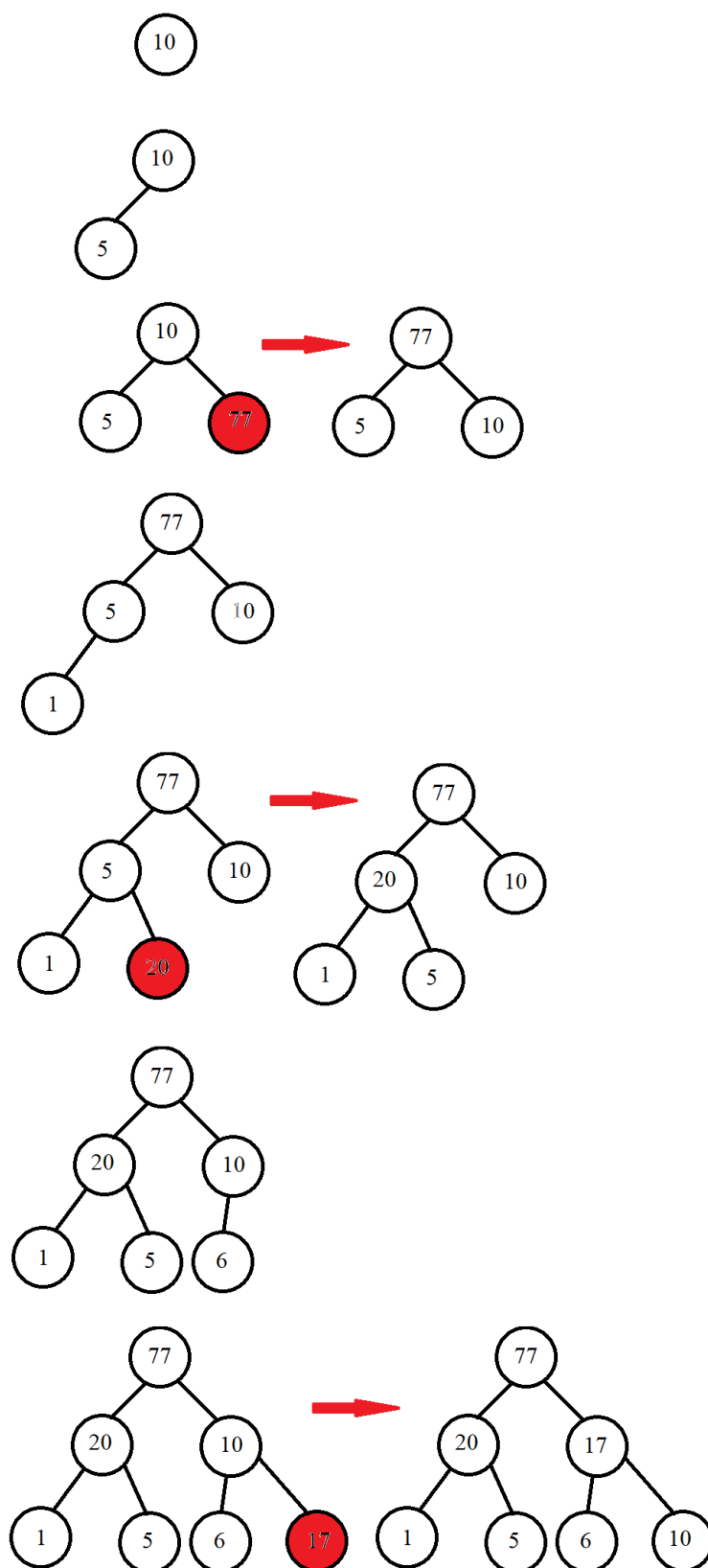
Jest to drzewo binarne, w którym wszystkie węzły spełniają następujący warunek: klucz węzła nadrzędnego jest większy lub równy (lub mniejszy bądź równy w zależności od kierunku sortowania) kluczom węzłów potomnych. Po dołączeniu do drzewa należy za każdym razem sprawdzać czy warunek kopca jest zachowany. Jeśli warunek nie jest spełniony należy dokonać zmian na ścieżce wiodącej od danego węzła do korzenia. Korzeń jest zawsze elementem największym (lub najmniejszym) w całym kopcu. Cechą charakterystyczną kopca jest to, że jest to drzewo zupełne poza liśćmi.

Rozbior kopca polega na tym, że należy zamienić wartości przechowywane w korzeniu z ostatnim liściem i usunąć liść. Po wykonaniu tego kroku należy sprawdzić czy zachowana została struktura kopca. Jeśli nie, strukturę kopca należy przywrócić idąc od jego korzenia w dół. Kroki należy powtarzać, aż kopiec zostanie pusty - wtedy dane zostały posortowane.

Tworzenie kopca:

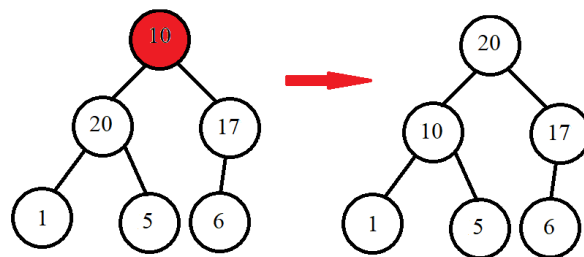
Na rys. 13.1 - 13.2 zostanie zaprezentowane tworzenie kopca dla następujących danych: 10,5,77,1,20,6,17. Na czerwono został zaznaczony element, który po dodaniu psuje strukturę kopca.



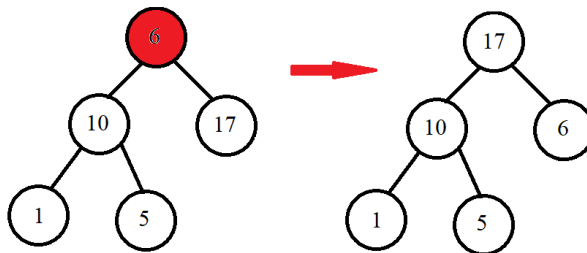


Rys. 13.1. Tworzenie kopca

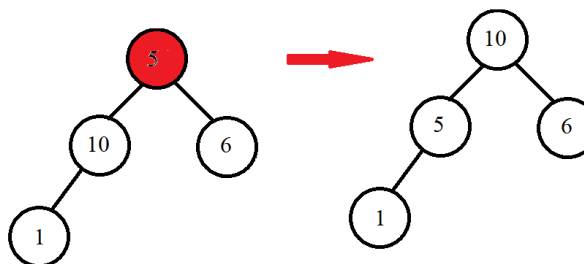
Rozbieranie kopca:



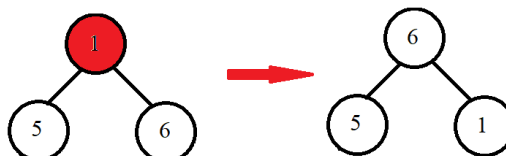
Posortowane elementy: 77



Posortowane elementy: 77,20



Posortowane elementy: 77,20,17



Posortowane elementy: 77,20,17,10



Posortowane elementy: 77,20,17,10,6



Posortowane elementy: 77,20,17,20,6,5

Posortowane elementy: 77,20,17,20,6,5,1

Rys. 13.2 Rozbieranie kopca



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

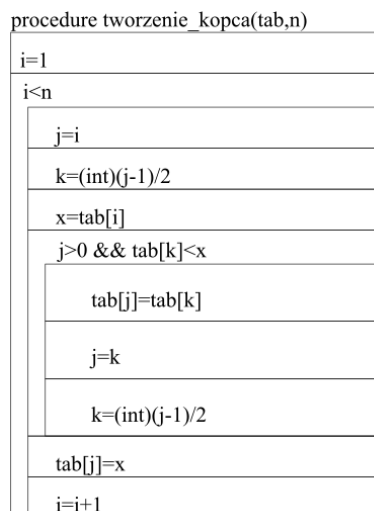
Unia Europejska
Europejski Fundusz Społeczny



Na rys. 13.3 przedstawiono schemat zwarty NS algorytmu tworzenia kopca.

Indeks j oraz k są indeksami elementów leżących na ścieżce od wstawianego elementu, w celu ustawienia go na odpowiednie miejsce, aby zachować strukturę kopca.

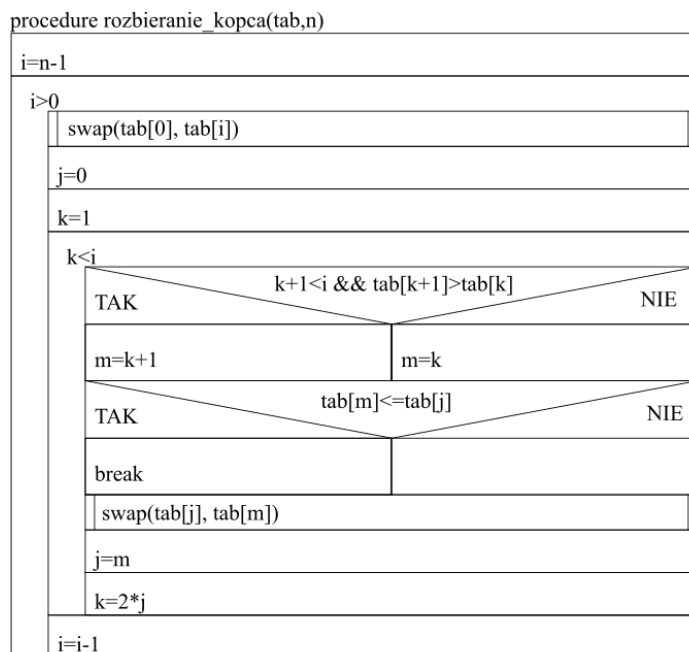
W zmiennej x tymczasowo przechowywany jest wstawiany do kopca element, który musi trafić w odpowiednie miejsce.



Rys. 13.3. Schemat zwarty tworzenia kopca

Indeksy j oraz k są indeksami elementów leżących na ścieżce od korzenia w dół. Indeks m jest indeksem większego elementu z dwóch potomnych.

Na rys. 13.4 przedstawiono schemat zwarty NS algorytmu rozbierania kopca.



Rys. 13.4. Schemat zwarty rozbierania kopca



Zadania do wykonania:

Zadanie 13.1 Sortowanie przez kopcowanie

Napisz program, który:

- wczyta rozmiar tablicy z dołączonego pliku – „studenci.csv”,
- przydzieli pamięć,
- wczyta dane o studentach z dołączonego pliku,
- pobierze od użytkownika informację w jakim trybie mają zostać posortowane dane studentów według punktów (rosnąco/malejąco),
- wyświetli zawartość tablicy przed sortowaniem,
- posortuje algorytmem przez kopcowanie dane studentów względem uzyskanych punktów rosnąco lub malejąco w zależności od wartości zmiennej tryb,
- wyświetli zawartość tablicy po sortowaniu,
- zwolni pamięć.

Stwórz strukturę student a następnie utwórz wskaźnik na tablicę studentów.

W celu napisania programu, należy stworzyć funkcje do: wczytywania danych, wyświetlania danych, sortowania, tworzenia obca oraz rozbierania kopca.

Plik z danymi należy umieścić w katalogu projektu, a nazwę pliku umieścić w kodzie programu.

LABORATORIUM 14. ALGORYTMY GRAFOWE

Cel laboratorium:

Omówienie grafów i algorytmów grafowych.

Zakres tematyczny zajęć:

- graf,
- algorytmy grafowe.

Pytania kontrolne:

- 1) Co to jest graf?
- 2) Jakie są rodzaje grafów?
- 3) Jaka jest różnica między grafem skierowanym i nieskierowanym?
- 4) W jaki sposób można przedstawić graf?
- 5) W jaki sposób konstruuje się macierz sąsiedztwa?
- 6) Jakie są algorytmy przeszukiwania grafu?
- 7) W jaki sposób można znaleźć najkrótszą ścieżkę w grafie nieważonym?

Graf:

Jest strukturą danych, która składa się z dwóch zbiorów: zbioru wierzchołków oraz zbioru krawędzi. Wierzchołki mogą być połączone krawędziami w taki sposób, że każda z krawędzi zaczyna się w jednym wierzchołku a kończy w drugim. Wyróżnia się kilka rodzajów grafów, do których należą między innymi:

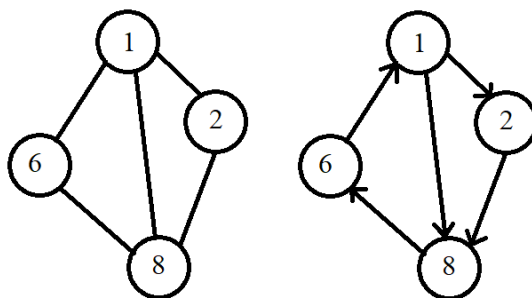
- graf skierowany – graf posiada krawędzie skierowane, czyli kolejność wierzchołków wyznaczana jest przez kierunek krawędzi;
- graf nieskierowany – graf nie posiada krawędzi skierowanych;
- graf mieszany – graf zawiera zarówno krawędzie skierowane jak i nieskierowane;
- graf ważony – graf, w którym krawędzi przypisana jest waga.

Istnieje kilka sposobów reprezentacji grafów, do najbardziej znanych należą:

- rysunek grafu – wierzchołek oznaczany jest jako koło, a krawędzią jest prosta lub krzywa;
- macierz sąsiedztwa – macierz, która zawiera informacje o połączeniach między wierzchołkami.

Na rys. 14.1. przedstawione zostały dwa rodzaje grafów: graf skierowany oraz graf nieskierowany, natomiast poniżej zostały przedstawione macierze sąsiedztwa dla grafu skierowanego i nieskierowanego.





Rys. 14.2. Przykład grafu skierowanego i nieskierowanego

Dla ułatwienia przyjmijmy, że w pierwszym wierzchołku przechowywana jest liczba 1, w drugim wierzchołku liczba 2, w trzecim wierzchołku liczba 6 oraz w czwartym wierzchołku liczba 8. Mamy 4 wierzchołki, w związku z tym macierz sąsiedztwa będzie miała wymiary 4x4. Każdy wiersz i każda kolumna reprezentuje jeden wierzchołek. Macierz sąsiedztwa uzupełniana jest w następujący sposób: jeśli istnieje droga z wierzchołka i do wierzchołka j to wstawiamy liczbę możliwych ścieżek pomiędzy tymi dwoma wierzchołkami pod element a_{ij} . Jeśli ścieżka nie istnieje wstawiamy 0. Wstawienie liczby 1 oznacza, że jest jedna ścieżka pomiędzy wybranymi wierzchołkami, jeśli wstawimy liczbę większą niż 1 to oznaczać to będzie, że występuje kilka ścieżek pomiędzy wierzchołkami tzn. ścieżki wielokrotne. Dla grafu nieskierowanego macierz przejść jest symetryczna względem głównej przekątnej, ponieważ możemy przejść z wierzchołka i do wierzchołka j i z wierzchołka j do wierzchołka i .

Macierz sąsiedztwa dla grafu nieskierowanego (rys.14.2.):

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Macierz sąsiedztwa dla grafu skierowanego (rys.14.2.):

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Poniżej przedstawione zostały niektóre pojęcia związane z teorią grafów:

- rząd grafu – liczba wierzchołków w grafie;
- rozmiar grafu – liczba krawędzi w grafie;
- ścieżka – ciąg krawędzi (lub wierzchołków), po których należy przejść aby dojść z wierzchołka startowego do wierzchołka końcowego. Wyróżniamy:
 - ścieżka Eulera – ścieżka, która przechodzi przez wszystkie krawędzie grafu;
 - ścieżka Hamiltona – ścieżka, która przechodzi przez wszystkie wierzchołki;
- cykl – ścieżka, która rozpoczyna się i kończy w tym samym wierzchołku.

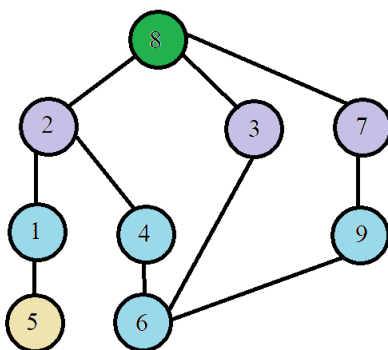
Algorytmy grafowe:

Jednymi z podstawowych algorytmów grafowych jest:

- przeszukiwaniu grafu w głąb (ang. Depth First Search – DFS),
- przeszukiwanie grafu w szerz (ang. Breadth First Search – BFS),
- znajdowanie najkrótszej ścieżki w grafie.

Przeszukiwanie grafu wszerz:

Zasada przejścia polega na oznaczeniu wierzchołka startowego jako odwiedzonego. W następnej kolejności odwiedzane są wszystkie wierzchołki, które są sąsiadami zaznaczonego elementu. Potem odwiedzani są sąsiedzi odwiedzanych elementów. Algorytm kończy się w momencie kiedy wszystkie wierzchołki zostaną odwiedzone. Należy stworzyć tablicę pomocniczą przechowującą informacje, czy dany wierzchołek został już odwiedzony. Na rys. 14.4. zaprezentowany został przykład przeszukania grafu wszerz. Przeszukiwanie zaczęło się od wierzchołka zaznaczonego na zielono, następnie zostali odwiedzeni wszyscy bezpośredni sąsiedzi wierzchołka zaznaczonego na zielono – zostali oznaczeni na fioletowo. Na niebiesko zostali zaznaczeni bezpośredni sąsiedzi wierzchołków zaznaczonych na fioletowo. Jako ostatni zostanie odwiedzony wierzchołek pomalowany na beżowo.

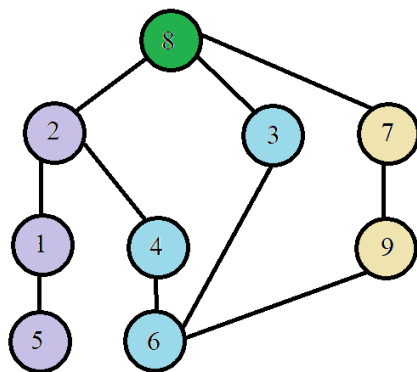


Rys. 14.4. Przeszukiwania grafu wszerz na przykładzie grafu nieskierowanego

Przeszukiwanie grafu w głąb:

Zasada przejścia polega na tym, że wierzchołek, który został już odwiedzony oznaczany jest jako odwiedzony. Następnie należy przejść do sąsiada oznaczonego wierzchołka, oznaczyć go jako odwiedzony i przejść do jego sąsiadów. Algorytm kończy się, kiedy w opisany powyżej sposób zostaną odwiedzone wszystkie wierzchołki. Należy stworzyć tablicę pomocniczą przechowującą informacje czy dany wierzchołek został już odwiedzony. Na rys. 14.5. przedstawiony został przykład przeszukiwania grafu w głąb. Przeszukiwanie zaczyna się od wierzchołka zaznaczonego na zielono. Następnie odwiedzane są wierzchołki zaznaczone na fioletowo. Potem odwiedzane są wierzchołki zaznaczone na niebiesko i na końcu zostaną odwiedzone wierzchołki zaznaczone na beżowo.





Rys. 14.4. Przeszukiwania grafu w głąb na przykładzie grafu nieskierowanego

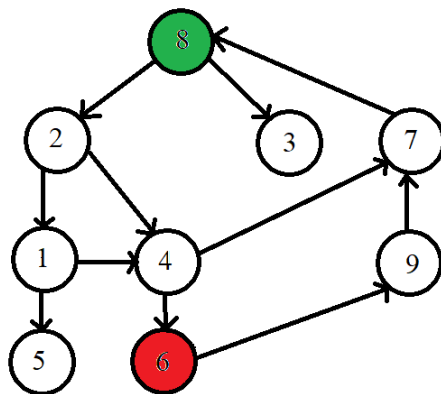
Znajdowanie najkrótszej ścieżki w grafie:

W celu znalezienia najkrótszej ścieżki zostanie wykorzystany algorytm przeszukiwania grafu wszerz BST. W tym celu wykorzystana zostanie tablica, która składa się z tylu elementów ile jest wierzchołków w grafie. W algorytmie BST najpierw odwiedzani są sąsiedzi wierzchołka startowego, następnie sąsiedzi sąsiadów itd. Indeksy w tablicy odnoszą się do wierzchołka, który jest aktualnie odwiedzany, natomiast wartość, która jest przechowywana pod indeksem to numer wierzchołka sąsiada, który został odwiedzony przed aktualnie odwiedzanym wierzchołkiem. Algorytm znajduje najkrótszą ścieżkę, o ile ona istnieje w przypadku grafu nieważonego. Algorytm kończy się w momencie znalezienia szukanego wierzchołka końcowego. Idąc po komórkach od tab[znaleziony] według ich zawartości, otrzymamy wierzchołki w kolejności odwrotnej, które należy odwiedzić, aby z wierzchołka startowego dojść do wierzchołka końcowego wybierając najkrótszą drogę. W przypadku grafu ważonego należy użyć algorytmu Dijkstry.

Na rys. 14.5. przedstawiony został graf skierowany. Na jego podstawie została wyznaczona najkrótsza ścieżka pomiędzy wierzchołkiem początkowym – 8 i wierzchołkiem końcowym 9. W celu wyznaczenia najkrótszej ścieżki należy stworzyć następującą tablicę. Na początku tablica jest pusta.

indeks	1	2	3	4	5	6	7	8	9
wartość	2	8	8	2	1	4	4	-1	

Pod indeksem w tablicy zapisywany jest wierzchołek, z którego można dojść do wierzchołka o aktualnym numerze indeksu. Algorytm zakończył wypełnianie tablicy pod indeksem 6 ponieważ, ścieżka została znaleziona. W celu znalezienia wierzchołków, po których należy przejść aby z wierzchołka początkowego dojść do wierzchołka końcowego należy iść po tablicy od tab[6] według jej zawartości, wierzchołki: 4,2,8. Wierzchołki zostały podane w odwrotnej kolejności, więc, aby przejść z wierzchołka 8 do wierzchołka 6 należy przejść przez wierzchołki: 8,2,4.

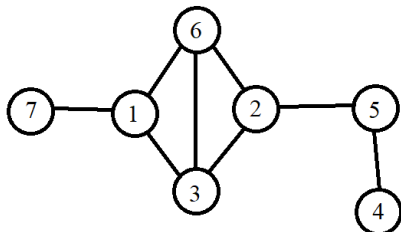


Rys. 14.5. Znajdowanie najkrótszej ścieżki na przykładzie grafu skierowanego

Zadania do wykonania:

Zadanie 14.1 Macierz przejść

Na podstawie poniższego rysunku grafu (rys. 14.6.), należy stworzyć macierz sąsiedztwa



Rys. 14.6. Graf nieskierowany

Zadanie 14.2 Przeszukanie grafu wszerz

Na podstawie grafu z zadania 14.1 należy wypisać numery wierzchołków w jakiej kolejności zostaną odwiedzone za pomocą algorytmu przeszukiwania wszerz. Przeszukiwanie należy zacząć od wierzchołka 6.

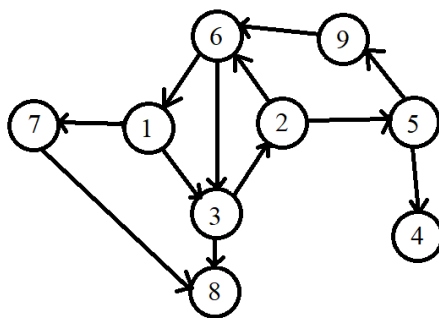
Zadanie 14.3 Przeszukanie grafu w głąb

Na podstawie grafu z zadania 14.1 należy wypisać numery wierzchołków w jakiej kolejności zostaną odwiedzone za pomocą algorytmu przeszukiwania w głąb. Przeszukiwanie należy zacząć od wierzchołka 2.

Zadanie 14.4 Najkrótsza ścieżka

Na podstawie poniższego grafu (rys. 14.7), należy znaleźć najkrótszą ścieżkę pomiędzy podanymi wierzchołkami za pomocą algorytmu przeszukiwania drzewa wszerz: wierzchołek początkowy: 6, wierzchołek końcowy: 5





Rys. 14.7. Graf skierowany

LABORATORIUM 15. KOŁOKWIUM NR 2

Cel laboratorium:

Weryfikacja nabytych umiejętności dotyczących struktur danych: stos, kolejka, lista, drzewo BST, grafy oraz algorytmu sortowania przez kopcowanie.

Wytyczne do kolokwium 2:

- zakres laboratoriów 8-14,
- próg zaliczeniowy 51%,
- pozostałe wytyczne i sposób zaliczenia kolokwium ustala prowadzący zajęcia.



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny





Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego