2. Znajdowanie drogi wyjścia z labiryntu

Zapewne nie raz spotkaliście się z grą, w której trzeba przeprowadzić bohatera przez labirynt korytarzy. Być może w szkole podstawowej pisaliście programy umożliwiające sterowanie postacią na planszy. W tym temacie poznasz algorytmy sprawdzające, czy istnieje droga prowadząca do wyjścia z labiryntu, oraz znajdziesz tę drogę.

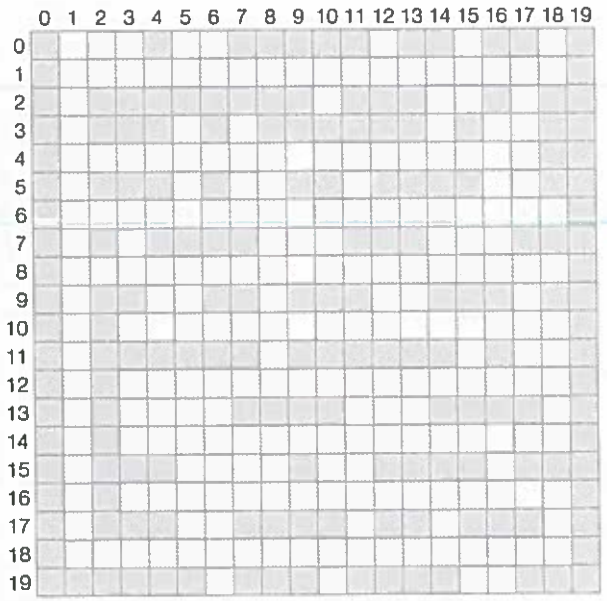
Cele lekcji

- Poznasz algorytmy znajdowania wyjścia z labiryntu.
- Znajdziesz najkrótszą drogę prowadzącą do wyjścia z labiryntu.
- Zastosujesz dynamiczną strukturę danych o nazwie kolejka.

2.1. Przygotowanie planszy z labiryntem

Tablica dwuwymiarowa, podręcznik *Informatyka na czasie 2. Zakres rozszerzony*, s. 219

Planszę z labiryntem będziemy reprezentować za pomocą **tablicy dwuwymiarowej**, w której liczba kolumn i liczba wierszy są równe. Przykładowy labirynt przedstawia rysunek 2.1. Pola szare oznaczają ściany, pola białe tworzą korytarze. Przyjmiemy, że przy krawędzi może się znajdować tylko jedno białe pole – będzie to wyjście. Polem startowym będzie dowolne białe pole. W szczególnym przypadku wyjście i pole startowe mogą być w tym samym miejscu.



Rys. 2.1. Przykładowa plansza z labiryntem

Opis planszy wygodnie jest przygotować w pliku tekstowym. Pola można oznaczyć konkretnymi znakami, np. pole szare wielką literą X, a pole białe – wielką literą W. Plansza na rysunku 2.1 ma wymiary 20 × 20, więc plik tekstowy ją opisujący będzie się składał z 20 wierszy, a każdy wiersz z 20 znaków (wielkich liter X lub W). Trzy pierwsze wiersze pliku dla planszy z rysunku 2.1 są następujące:

```
XWXXXXXXXXXXXXXXXXXX
XWWWWWWWWWWWWWWWWWWX
XWXXXXXXXXWXXXWXXXXX
```

Warto wiedzieć

Istnieją algorytmy pozwalające wygenerować labirynt, w którym z dowolnego miejsca można wyznaczyć drogę prowadzącą do wyjścia.

Algorytmy poszukujące drogi wyjścia z labiryntu będą oznaczały pola już rozpatrzone. Wykorzystają do tego liczby całkowite. Dlatego w tablicy reprezentującej planszę przechowamy takie właśnie liczby. Pola tworzące ściany oznaczymy liczbą –1. Liczbę 0 przypiszemy polom korytarzy, które nie były jeszcze rozpatrywane. Liczby dodatnie wykorzystamy przy poszukiwaniu drogi, a liczbę –2 do oznaczenia znalezionej drogi. Wartości tablicy reprezentującej planszę z rysunku 2.1 przedstawia rysunek 2.2.

Warto wiedzieć

Informacja o tym, czy pole jest oznaczone kolorem szarym czy białym, to informacja zero-jedynkowa (logiczna).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
2	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	0	-1	-1	-1	-1	-1
3	-1	0	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1
4	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-1
5	-1	0	-1	-1	-1	0	-1	-1	0	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1
6	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
7	-1	0	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	0	-1	-1	-1	-1
8	-1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
9	-1	0	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1
10	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
11	-1	0	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1
12	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
13	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1	-1
14	-1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
15	1	0	-1	-1	-1	-1	-1	0	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
17	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1
18	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
19	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Rys. 2.2. Wartości tablicy reprezentującej planszę z rysunku 2.1

Funkcja wczytująca do programu opis labiryntu z pliku tekstowego powinna zamieniać odczytane znaki (X lub W) na odpowiednie wartości liczbowe. Parametrem funkcji będzie tablica dwuwymiarowa Lab. Liczbę wierszy i kolumn określimy za pomocą zdefiniowanej w programie stałej N. Należy pamiętać o dodaniu w kodzie programu dyrektywy `#include <fstream>` – powoduje ona dołączenie biblioteki służącej do **obsługi plików**. Kod źródłowy funkcji wczytującej opis labiryntu z pliku tekstowego *labirynt.txt* może być następujący.

Dobra rada

Pamiętaj, że definicję stałej rozpoczyna się od słowa kluczowego `const`. Po tym słowie podaj typ stałej i jej nazwę, a po znaku równości – wartość stałej.

Obsługa plików tekstowych w języku C++, s. 417–419

Kod źródłowy funkcji wczytującej do programu opis planszy z labiryntem z pliku tekstowego

```
1.  void WczytajLabirynt(int Lab[][N])
2.  {
3.      string s;
4.      ifstream we("labirynt.txt");
5.      for (int i=0;i<N;i++)
6.      {
7.          we>>s;
8.          for (int j=0;j<N;j++)
9.              if (s[j]=='X') Lab[i][j]=-1;
10.             else Lab[i][j]=0;
11.     }
12.     we.close();
13. }
```

Dobra rada

Gdy parametrem funkcji jest tablica dwuwymiarowa, nie możesz pominąć w nawiasach jej drugiego wymiaru (liczby kolumn).

Dobra rada

Pamiętaj, że tablice są przekazywane do funkcji przez wskaźnik. Oznacza to, że operacje są wykonywane na oryginalnej tablicy (nie jest tworzona kopia takiego parametru), a dokonane w niej zmiany zostają zachowane po zakończeniu działania funkcji.

W linii 4 deklarujemy zmienną plikową we typu `ifstream` i otwieramy do odczytu skojarzony z nią plik zawierający opis planszy. W linii 7 program wczytuje do zmiennej s wiersz pliku reprezentowanego przez zmienną we. Pętla w liniach 8–10 przegląda znaki wczytanego napisu i jeśli dany znak jest literą X, to odpowiedniemu elementowi tablicy przypisuje wartość –1, w przeciwnym przypadku – wartość 0. W linii 12 przy użyciu metody close zamykamy plik skojarzony ze zmienną we.

Napiszemy kod źródłowy funkcji, która na podstawie wartości zapisanych w tablicy Lab wypisze planszę z labiryntem na ekranie. Pole korytarza wyświetli się jako puste, a dla pola oznaczającego ścianę wypiszemy znak X. Oprócz labiryntu wypiszemy numery wierszy oraz kolumn. Na każde pole przeznaczymy trzy znaki. Oto kod źródłowy funkcji:

Kod źródłowy funkcji wypisującej planszę z labiryntem na podstawie wartości elementów tablicy

```
1.  void WypiszLabirynt(int Lab[][N])
2.  {
3.      cout<<"   ";
4.      for (int j=0;j<N;j++)
5.          cout<<setw(3)<<j;
6.      cout<<endl;
7.      for (int i=0;i<N;i++)
8.      {
9.          cout<<setw(3)<<i;
10.         for (int j=0;j<N;j++)
11.             if (Lab[i][j]==-1) cout<<"  X";
12.             else cout<<"   ";
13.         cout<<endl;
14.     }
15. }
```

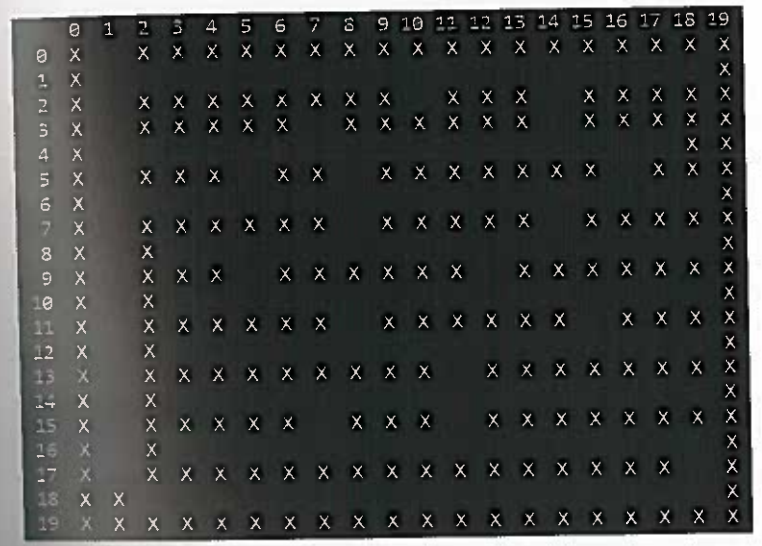
Na ekranie najpierw zostaną wypisane trzy spacje (linia 3), a następnie numery kolumn (linie 4–5). Przy wypisywaniu numeru kolumny wykorzystujemy **funkcję setw** z biblioteki iomanip, formatującą wypisywane wartości. Jej parametrem jest liczba całkowita, która określa, ile znaków

Funkcja setw

będzie przeznaczonych na wyświetlenie napisu. Dla parametru równego 3 liczba jednocyfrowa zostanie poprzedzona dwiema spacjami, a liczba dwucyfrowa – jedną spacją. Jeśli wyświetlany napis liczyłby więcej znaków, niż określa to parametr funkcji, to szerokość pola zostałaby odpowiednio zwiększona. Funkcja **setw** jest przykładem **manipulatora strumienia**. Manipulatory strumieni pozwalają m.in. określić sposób interpretowania wczytywanych danych i wyświetlania informacji na ekranie.

Manipulator strumienia

Pętla w liniach 7–14 odpowiada za wypisanie znaków w kolejnych wierszach. Najpierw wypisywany jest numer wiersza (linia 9) – tu także wykorzystana jest funkcja **setw**. Pętla w liniach 10–12 wypełnia wiersz na podstawie tablicy Lab: dla wartości elementu –1 (ściana) wypisuje znak X poprzedzony dwiema spacjami, w przeciwnym przypadku – trzy spacje (wolne pole). Rysunek 2.3 przedstawia efekt działania funkcji WypiszLabirynt dla wartości tablicy takich jak na rysunku 2.2 ze s. 29.



Rys. 2.3. Efekt działania funkcji WypiszLabirynt dla wartości tablicy z rysunku 2.2

Dobra rada

Aby korzystać w programie z funkcji zdefiniowanych w bibliotece iomanip, pamiętaj o dodaniu dyrektywy `#include <iomanip>`.

Warto wiedzieć

W bibliotece iomanip dostępnych jest wiele manipulatorów strumieni. Manipulatory zdefiniowano także w bibliotece iostream. Standard języka C++ nie precyzuje, jak manipulatory zachowują się w przypadku nieprawidłowego użycia.

Warto wiedzieć

Dzięki funkcji **setw** można też określić sposób wyświetlania wartości innego typu niż liczby całkowite, np. znaków. Poniższa instrukcja wyświetli znak X poprzedzony dwiema spacjami:
`cout<<setw(3)<<'X';`

Ćwiczenie 1

Napisz program, który wczyta opis labiryntu z pliku otrzymanego od nauczyciela (np. *labirynt.txt*) i wypisze planszę z labiryntem na ekranie. Wykorzystaj funkcje WczytajLabirynt oraz WypiszLabirynt.

2.2. Rekurencyjny algorytm znajdowania wyjścia z labiryntu

Sprawdzimy najpierw, czy istnieje droga prowadząca z pola startowego do wyjścia. Następnie wyznaczymy taką drogę.

Rekurencyjny algorytm sprawdzający, czy istnieje droga wyjścia z labiryntu

Specyfikacja problemu jest następująca:

Dobra rada

Staraj się nadawać zmiennym takie nazwy, które podpowiadają, co będzie w nich pamiętane. Dzięki temu kod źródłowy programu jest czytelniejszy i łatwiej go poprawiać. Na przykład w naszym programie nazwa w jest pierwszą literą słowa „wiersz", a k – pierwszą literą słowa „kolumna".

Specyfikacja

Dane: n – liczba całkowita dodatnia określająca liczbę wierszy i liczbę kolumn tablicy Lab,

Lab[0..n–1][0..n–1] – tablica o n wierszach i n kolumnach, której elementami są liczby całkowite opisujące planszę z labiryntem, Lab[i][j]=–1, gdy pole (i,j) jest ścianą, Lab[i][j]=0 – w przeciwnym przypadku, $0 \leqslant i < n$, $0 \leqslant j < n$,

(w,k) – pole startowe, Lab[w][k]=0, $0 \leqslant w < n$, $0 \leqslant k < n$.

Wynik: wartość logiczna **prawda**, gdy istnieje droga od pola (w,k) do wyjścia, czyli pola (w1,k1), gdzie w1 = 0 lub w1 = n – 1 lub k1 = 0 lub k1 = n – 1, **fałsz** – w przeciwnym przypadku.

Jeśli pole startowe leży w skrajnym wierszu lub skrajnej kolumnie, to znaleźliśmy wyjście z labiryntu. Algorytm kończy wówczas działanie. W przeciwnym przypadku należy sprawdzić pola sąsiadujące z tym, na którym się aktualnie znajdujemy. Jeśli wolno stanąć na sąsiednim polu, to potraktujemy je jako nowe pole startowe i wywołamy rekurencyjnie funkcję dla tego pola.

Odwiedzone pola należy oznaczać, inaczej wielokrotnie moglibyśmy poszukiwać drogi z tego samego pola (wywoływać funkcję rekurencyjnie dla tych samych parametrów, a więc mielibyśmy do czynienia z rekurencją nieskończoną). Do oznaczenia odwiedzonych pól wykorzystamy wartość 1. Jeśli po rekurencyjnym sprawdzeniu, czy istnieje droga z sąsiednich pól, nie dotrzemy do wyjścia, to nie istnieje droga z pola startowego do wyjścia. Funkcja zwróci wówczas wartość **fałsz**.

Oto zapis w pseudokodzie funkcji sprawdzającej, czy istnieje droga prowadząca do wyjścia z labiryntu:

```
funkcja Droga(Lab[][],w,k)
    Lab[w][k] ← 1
    jeśli w = 0 lub w = n - 1 lub k = 0 lub k = n - 1 to
        zwróć prawda i zakończ
    jeśli Lab[w-1][k] = 0 oraz Droga(Lab,w-1,k) to
        zwróć prawda i zakończ
    jeśli Lab[w+1][k] = 0 oraz Droga(Lab,w+1,k) to
        zwróć prawda i zakończ
    jeśli Lab[w][k-1] = 0 oraz Droga(Lab,w,k-1) to
        zwróć prawda i zakończ
    jeśli Lab[w][k+1] = 0 oraz Droga(Lab,w,k+1) to
        zwróć prawda i zakończ
    zwróć fałsz i zakończ
```


Dla labiryntu z rysunku 2.2 ze s. 29 oraz pola startowego (1,14) podany algorytm oznaczy liczbą 1 pola labiryntu wyróżnione na rysunku 2.4 kolorem zielonym. Jeśli w algorytmie zmienimy kolejność wywołań rekurencyjnych, liczbą 1 mogą zostać oznaczone inne pola.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	-1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	-1
2	-1	1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	1	-1	-1	-1	-1	-1
3	-1	1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1
4	-1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	-1	-1
5	-1	0	-1	-1	-1	1	-1	-1	1	-1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1
6	-1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
7	-1	0	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1
8	-1	0	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
9	-1	0	-1	-1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1	-1	-1
10	-1	0	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
11	-1	0	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1
12	-1	0	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
13	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1	-1	-1	-1
14	-1	0	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
15	-1	0	-1	-1	-1	-1	-1	1	-1	-1	-1	1	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	0	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
17	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	-1
18	-1	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
19	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Rys. 2.4. Labirynt z polami oznaczonymi przez algorytm rekurencyjny liczbą 1, gdy polem startowym jest pole (1,14)

Warto wiedzieć

Algorytm sprawdzający, czy istnieje droga wyjścia z labiryntu, nie musi rozpatrzeć wszystkich wolnych pól planszy. Na przykład na rysunku 2.4 widać, że pola (1,13) i (1,15), sąsiadujące z polem startowym (1,14), nie zostały odwiedzone. Stało się tak dlatego, że przed nimi było rozpatrywane pole (2,14), a po przejściu przez nie można znaleźć drogę do wyjścia z labiryntu.

Ćwiczenie 2

Napisz program, który wczyta opis labiryntu z pliku otrzymanego od nauczyciela (np. *labirynt.txt*) oraz wypisze planszę z labiryntem na ekranie. Następnie wczyta współrzędne pola startowego i sprawdzi, czy istnieje droga prowadząca z tego pola do wyjścia. Jeśli droga istnieje, program wypisze „TAK", w przeciwnym przypadku – „NIE".

Rekurencyjny algorytm wyznaczający drogę do wyjścia z labiryntu

Oznaczenie odwiedzonych pól taką samą liczbą (w naszym przypadku liczbą 1) nie wystarczy, aby wskazać drogę prowadzącą do wyjścia. W algorytmie wyznaczającym drogę odwiedzone pola będziemy oznaczać kolejnymi liczbami całkowitymi dodatnimi, zaczynając od 1. Pole startowe otrzyma zatem wartość 1, następne odwiedzone pole – wartość 2 itd. W ten sposób droga przejścia będzie oznaczona kolejnymi liczbami całkowitymi dodatnimi. W funkcji Droga dodamy parametr x określający wartość, którą należy oznaczyć aktualnie rozpatrywane pole. Oto zmodyfikowana funkcja zapisana w pseudokodzie.

```
funkcja Droga(Lab[][],w,k,x)
    Lab[w][k] ← x
    jeśli w = 0 lub w = n - 1 lub k = 0 lub k = n - 1 to
        zwróć prawda i zakończ
    jeśli Lab[w-1][k] = 0 oraz Droga(Lab,w-1,k,x+1) to
        zwróć prawda i zakończ
    jeśli Lab[w+1][k] = 0 oraz Droga(Lab,w+1,k,x+1) to
        zwróć prawda i zakończ
    jeśli Lab[w][k-1] = 0 oraz Droga(Lab,w,k-1,x+1) to
        zwróć prawda i zakończ
    jeśli Lab[w][k+1] = 0 oraz Droga(Lab,w,k+1,x+1) to
        zwróć prawda i zakończ
    zwróć fałsz i zakończ
```

Rysunek 2.5 przedstawia labirynt z polami, które algorytm oznaczy kolejnymi liczbami całkowitymi dodatnimi, gdy startujemy z pola (1,14).

Warto wiedzieć

Plansza może wyglądać inaczej, jeśli w algorytmie zmienimy kolejność wywołań rekurencyjnych.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	-1	25	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	24	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	-1	
2	-1	23	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	2	-1	-1	-1	-1	-1
3	-1	22	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	3	-1	-1	-1	-1	-1
4	-1	21	20	19	18	17	0	0	10	9	8	7	6	5	4	47	46	47	-1	-1
5	-1	0	-1	-1	-1	16	-1	-1	11	-1	-1	-1	-1	-1	-1	-1	45	-1	-1	-1
6	-1	0	0	0	0	15	14	13	12	47	46	45	44	43	42	43	44	45	46	-1
7	-1	0	-1	-1	-1	-1	-1	-1	13	-1	-1	-1	-1	-1	41	-1	-1	-1	-1	-1
8	-1	0	-1	19	18	17	16	15	14	41	40	39	38	39	40	41	42	43	44	-1
9	-1	0	-1	-1	-1	18	-1	-1	-1	-1	-1	-1	37	-1	-1	-1	-1	-1	-1	-1
10	-1	0	-1	21	20	19	20	21	22	39	38	37	36	35	34	33	34	35	36	-1
11	-1	0	-1	-1	-1	-1	-1	-1	23	-1	-1	-1	-1	-1	-1	32	-1	-1	-1	-1
12	-1	0	-1	29	28	27	26	25	24	25	26	27	28	29	30	31	32	33	34	-1
13	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	29	-1	-1	-1	-1	-1	-1	-1
14	-1	0	-1	41	40	39	38	37	38	39	40	29	30	31	32	33	34	35	36	-1
15	-1	0	-1	-1	-1	-1	-1	36	-1	-1	-1	30	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	0	-1	39	38	37	36	35	34	33	32	31	32	33	34	35	36	37	38	-1
17	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	39	-1
18	-1	-1	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	-1
19	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Rys. 2.5. Labirynt z polami oznaczonymi przez algorytm rekurencyjny kolejnymi liczbami całkowitymi dodatnimi, gdy polem startowym jest (1,14)

Zwróć uwagę, że na planszy powtarzają się niektóre liczby. Spójrz np. na pole (6,8) oznaczone wartością 12. Dwa pola obok niego – pola (6,7) oraz (7,8) – są oznaczone wartością 13. Zgodnie z algorytmem po polu (6,8) najpierw rozpatrywane jest pole (7,8). Ponieważ po przejściu przez to pole nie uda się dotrzeć do wyjścia (są ślepe korytarze lub algorytm napotka pole wcześniej odwiedzone), algorytm wróci do pola (6,8) i poszuka drogi przechodzącej przez pole po lewej – pole (6,7).

Żeby wskazać drogę, należy ją ponownie przejrzeć w odwrotnym kierunku – od wyjścia do pola startowego. Gdybyśmy zaczęli od pola startowego, po dotarciu do pola, dla którego sąsiednie pola mają te same

wartości, np. pola (6,8), nie byłoby wiadomo, na które z sąsiednich pól przejść. Drogę można oznaczyć konkretną wartością, np. liczbą –2. Oto zmodyfikowana specyfikacja problemu:

Specyfikacja

Dane: n – liczba całkowita dodatnia określająca liczbę wierszy i liczbę kolumn tablicy Lab,

Lab[0..n–1][0..n–1] – tablica o n wierszach i n kolumnach, której elementami są liczby całkowite opisujące planszę z labiryntem, Lab[i][j]=–1, gdy pole (i,j) jest ścianą, Lab[i][j]=0 – w przeciwnym przypadku, $0 \leqslant i < n$, $0 \leqslant j < n$,

(w, k) – pole startowe, Lab[w][k]=0, $0 \leqslant w < n$, $0 \leqslant k < n$.

Wynik: wartość logiczna **prawda**, gdy istnieje droga od pola (w,k) do wyjścia, czyli pola (w1,k1), gdzie w1 = 0 lub w1 = n – 1 lub k1 = 0 lub k1 = n – 1, **fałsz** – w przeciwnym przypadku,

Lab – tablica z drogą oznaczoną liczbą –2: od pola (w,k) do pola (w1,k1), jeśli taka droga istnieje.

Zapis funkcji oznaczającej drogę może być taki jak poniżej. Parametry w i k oznaczają tym razem nie pole startowe, ale znalezione wyjście.

Warto wiedzieć

Funkcję OznaczDroge można również zapisać rekurencyjnie.

```
funkcja OznaczDroge(Lab[][],w,k)
    x ← Lab[w][k]
    Lab[w][k] ← -2
    dopóki x > 1 wykonuj
        x ← x - 1
        jeśli w > 0 oraz Lab[w-1][k] = x to
            w ← w - 1
        w przeciwnym przypadku
            jeśli w < n - 1 oraz Lab[w+1][k] = x to
                w ← w + 1
            w przeciwnym przypadku
                jeśli k > 0 oraz Lab[w][k-1] = x to
                    k ← k - 1
                w przeciwnym przypadku
                    k ← k + 1
    Lab[w][k] ← -2
```

Zmienna x przyjmuje wartość pamiętaną dla pola, które jest znalezionym wyjściem. Polu temu następnie przypisujemy wartość –2. Dopóki nie dojdziemy do pola startowego (wartość zmiennej x jest większa od 1), poszukujemy sąsiedniego pola o wartości o 1 mniejszej względem aktualnie rozpatrywanego pola, które następnie oznaczamy jako należące do drogi. Jeśli dwa sąsiednie pola są oznaczone taką samą wartością, to nie ma znaczenia, które z nich wybierzemy – przez obydwa prowadzą drogi do wyjścia z labiryntu.

Kody źródłowe funkcji Droga oraz OznaczDroge w programie wyznaczającym drogę wyjścia z labiryntu mogą być następujące:

Fragment kodu źródłowego programu wyznaczającego rekurencyjnie drogę do wyjścia z labiryntu – funkcje Droga i OznaczDroge

```
1.  bool Droga(int Lab[][N], int w, int k, int x,
2.             int &w1, int &k1)
3.  {
4.      Lab[w][k]=x;
5.      if (w==0 || w==N-1 || k==0 || k==N-1)
6.      {
7.          w1=w; k1=k;
8.          return true;
9.      }
10.     if (Lab[w-1][k]==0 && Droga(Lab,w-1,k,x+1,w1,k1))
11.         return true;
12.     if (Lab[w+1][k]==0 && Droga(Lab,w+1,k,x+1,w1,k1))
13.         return true;
14.     if (Lab[w][k-1]==0 && Droga(Lab,w,k-1,x+1,w1,k1))
15.         return true;
16.     if (Lab[w][k+1]==0 && Droga(Lab,w,k+1,x+1,w1,k1))
17.         return true;
18.     return false;
19. }
20.
21. void OznaczDroge(int Lab[][N], int w, int k)
22. {
23.     int x=Lab[w][k];
24.     Lab[w][k]=-2;
25.     while (x>1)
26.     {
27.         x--;
28.         if (w>0 && Lab[w-1][k]==x) w--;
29.         else if (w<N-1 && Lab[w+1][k]==x) w++;
30.              else if (k>0 && Lab[w][k-1]==x) k--;
31.                   else k++;
32.         Lab[w][k]=-2;
33.     }
34. }
```

Dobra rada

Pamiętaj, że przy przekazaniu parametru przez referencję zmiany nanoszone są bezpośrednio na parametrze, nie na jego kopii. Dzięki temu zmieniona wartość parametru jest pamiętana po zakończeniu działania funkcji.

Zwróć uwagę na nagłówek funkcji Droga. Ma ona dwa dodatkowe parametry w1 i k1, przekazywane przez referencję. Za ich pomocą funkcja zwróci współrzędne znalezionego wyjścia. Są one potrzebne do oznaczenia drogi – ich wartości będą parametrami aktualnymi funkcji OznaczDroge.

Do wypisania labiryntu ze znalezioną drogą wykorzystamy zdefiniowaną wcześniej **funkcję** WypiszLabirynt. Zmodyfikujemy ją tak, aby dla wartości elementu tablicy Lab równej –2 była wypisywana litera D. Oto instrukcja warunkowa odpowiedzialna za wyświetlenie pola labiryntu w zależności od wartości elementu tablicy Lab.

Kod źródłowy funkcji wypisującej planszę z labiryntem na podstawie wartości elementów w tablicy, s. 30

Zmodyfikowany fragment kodu źródłowego funkcji wypisującej planszę z labiryntem na podstawie wartości elementów tablicy

```
1.  if (Lab[i][j]==-1) cout<<"  X";
2.  else if (Lab[i][j]==-2) cout<<"  D";
3.       else cout<<"   ";
```

Kod źródłowy funkcji main programu znajdującego i wypisującego drogę wyjścia z labiryntu może być następujący:

Fragment kodu źródłowego programu wyznaczającego rekurencyjnie drogę do wyjścia z labiryntu – funkcja main

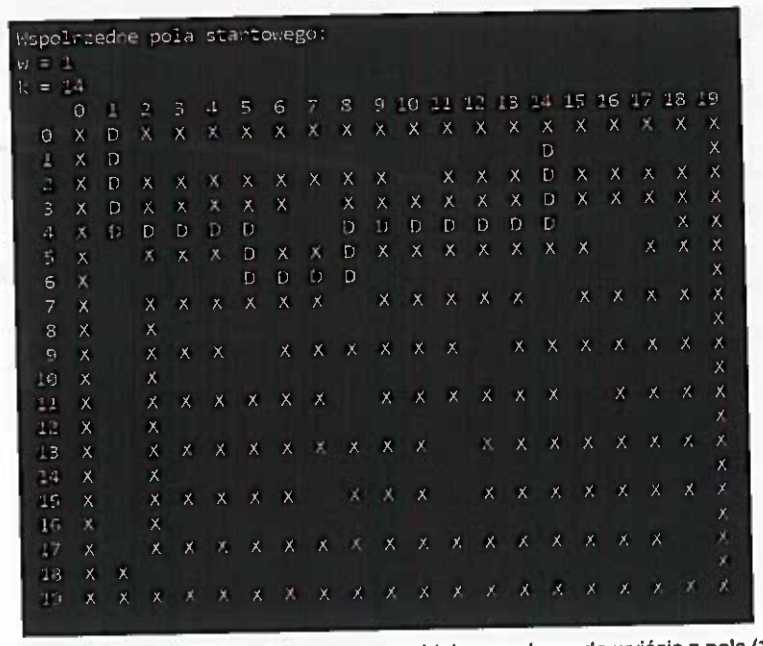
```
1.  int main()
2.  {
3.      int w, k, w1, k1;
4.      int Lab[N][N];
5.      WczytajLabirynt(Lab);
6.      WypiszLabirynt(Lab);
7.      cout<<"Wspolrzedne pola startowego: "<<endl;
8.      cout<<"w = "; cin>>w;
9.      cout<<"k = "; cin>>k;
10.     if (Droga(Lab,w,k,1,w1,k1))
11.     {
12.         OznaczDroge(Lab,w1,k1);
13.         WypiszLabirynt(Lab);
14.     }
15.     else
16.         cout<<"Brak drogi";
17.     return 0;
18. }
```

W liniach 7–9 wczytujemy współrzędne pola startowego. Jeśli droga do wyjścia zostanie odnaleziona (funkcja Droga przyjmie wartość **true** – linia 10), to program oznacza drogę (linia 12) oraz wypisuje labirynt z zaznaczoną drogą (linia 13). Gdy nie ma drogi prowadzącej do wyjścia, wyświetli się odpowiedni komunikat (linia 16). Przykład działania programu przedstawia rysunek 2.6 na s. 38.

A to ciekawe

W sieci korytarzy

Motyw labiryntu jest znany od tysiącleci i ma bogatą symbolikę. Pojawił się np. w micie o Tezeuszu, w którym główny bohater wydostaje się z tej pułapki dzięki nici Ariadny. Niektórzy uważają, że mityczny labirynt znajdował się w pałacu w Knossos na Krecie. Labirynty budowano m.in. po to, aby utrudnić dotarcie do grobowców i skarbców. Pełniły także funkcję dekoracyjną – do dziś zdobią wnętrza wielu świątyń (np. katedr w Amiens i Chartres we Francji), były także bardzo popularne w renesansowych ogrodach.



Rys. 2.6. Przykład działania programu znajdującego drogę do wyjścia z pola (1,14)

Ćwiczenie 3

Napisz program, który z pliku przekazanego ci przez nauczyciela (np. *labirynt.txt*) wczyta informacje o planszy z labiryntem, a następnie wypisze tę planszę na ekranie. Po wczytaniu współrzędnych pola startowego program wypisze planszę z oznaczoną drogą do wyjścia, jeśli taka istnieje, lub komunikat informujący, że drogi do wyjścia nie ma. Znaleziona droga nie musi być najkrótszą możliwą. Przetestuj działanie programu dla różnych pól startowych.

Algorytm przeszukiwania z nawrotami

Opisany rekurencyjny algorytm poszukiwania drogi wyjścia z labiryntu jest przykładem **algorytmu przeszukiwania z nawrotami**. Jest to ogólna technika rozwiązywania problemów polegająca na usystematyzowanym (nielosowym) przeglądaniu możliwych rozwiązań (w naszym przypadku dróg w labiryncie). Gdy okazuje się, że kandydat na rozwiązanie nie jest właściwy, algorytm powraca do punktu, w którym może modyfikować rozwiązanie. W naszym programie, gdy okazuje się, że wytyczana droga nie prowadzi do wyjścia, program wraca do pola, w którym może modyfikować drogę, i sprawdza kolejnego kandydata.

Warto wiedzieć

Innym przykładem problemu, który można rozwiązać metodą z nawrotami, jest problem ustawienia na szachownicy 8 hetmanów tak, aby się nawzajem nie szachowały.

Zwróć uwagę, że jeśli istnieje droga prowadząca do wyjścia z labiryntu, to omówiony algorytm ją znajdzie, ale nie zawsze będzie to droga najkrótsza. Na przykład dla planszy i pola startowego przedstawionych na rysunku 2.6 można wskazać krótszą drogę.

Zapamiętaj

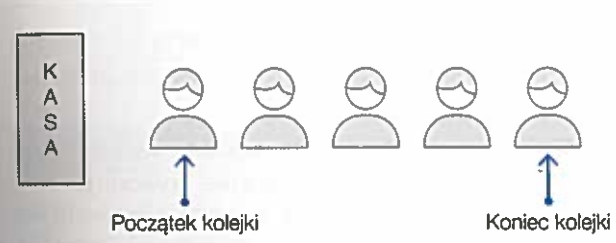
Rekurencyjny algorytm znajdowania drogi do wyjścia z labiryntu jest przykładem algorytmu z nawrotami. Jeśli kandydat na rozwiązanie nie jest właściwy, algorytm powraca do punktu, w którym może modyfikować rozwiązanie. Podany algorytm zawsze znajdzie drogę, o ile ona istnieje, ale niekoniecznie będzie to droga najkrótsza.

2.3. Czym jest kolejka?

Drogę do wyjścia z labiryntu można znaleźć za pomocą innego algorytmu, który wykorzystuje strukturę danych nazywaną **kolejką**. Kolejka (ang. *queue*) to **dynamiczna struktura danych**, w której nowy element można dodać tylko na końcu, a usunąć można tylko element znajdujący się z przodu. W kolejce przechowuje się dane tego samego typu. Zasada przy dodawaniu i usuwaniu elementów jest taka jak w przypadku kolejki klientów do kasy. Ten klient, który pierwszy stanął w kolejce, jako pierwszy będzie obsłużony. Nowy klient może stanąć jedynie na końcu kolejki (rys. 2.7).

Kolejka

Dynamiczna struktura danych, s. 12



Rys. 2.7. Kolejka do kasy

Kolejka jest strukturą danych, która realizuje strategię **FIFO** (od ang. *first in, first out* – pierwszy wchodzi, pierwszy wychodzi). Możemy wyróżnić następujące **operacje na kolejce**:

FIFO

Operacje na kolejce

- *push* – umieszczenie elementu na końcu kolejki,
- *pop* – usunięcie elementu z początku kolejki,
- *front* – pobranie wartości elementu z początku kolejki,
- *empty* – sprawdzenie, czy kolejka jest pusta.

Zapamiętaj

Kolejka jest dynamiczną strukturą danych, a więc jej rozmiar może się zmieniać w trakcie działania programu. Operacje na kolejce wykonuje się na jej początku i końcu. Do operacji tych należą: dodanie elementu na końcu kolejki (*push*), usunięcie elementu z początku kolejki (*pop*), pobranie wartości elementu z początku kolejki (*front*) oraz sprawdzenie, czy kolejka jest pusta (*empty*).

2.4. Iteracyjny algorytm znajdowania wyjścia z labiryntu

Warto wiedzieć

Kolejkę często wykorzystuje się w systemach operacyjnych. Ustawia się w niej m.in. zadania oczekujące na wydrukowanie lub przetworzenie przez procesor.

Wadą rekurencyjnego algorytmu znajdującego drogę prowadzącą do wyjścia z labiryntu jest nierówne traktowanie poszczególnych pól. Uprzywilejowane jest to sąsiednie pole, dla którego najpierw następuje wywołanie rekurencyjne. Jeśli przy tym wywołaniu algorytm znajdzie drogę do wyjścia, to ta droga jest wynikiem działania algorytmu. Nie musi być to jednak droga najkrótsza. Być może krótszą drogę udałoby się znaleźć, jeśli przeszlibyśmy na inne sąsiednie pole. Wszystkie sąsiednie pola powinny być więc traktowane tak samo.

Dla danego pola istnieją maksymalnie cztery sąsiednie pola, na które można przejść. Wszystkich ich równocześnie nie rozpatrzymy, więc współrzędne tych, na które można przejść, umieścimy w kolejce. Odwiedzone pola tak jak poprzednio będziemy oznaczać kolejnymi liczbami całkowitymi dodatnimi. Na początku do kolejki wstawiamy współrzędne pola startowego i oznaczamy to pole wartością 1. Następnie, dopóki nie znajdziemy wyjścia i są jakieś pola w kolejce do rozpatrzenia (kolejka nie jest pusta), pobieramy pole z kolejki i sprawdzamy, czy jest wyjściem. Jeśli nie jest, to rozpatrujemy sąsiednie pola. Jeśli nie zostały jeszcze odwiedzone, to oznaczamy je odpowiednią liczbą i umieszczamy w kolejce.

Oto zapis algorytmu w pseudokodzie:

```
wyjście ← fałsz
wstaw pole (w,k) do kolejki
Lab[w][k] ← 1
dopóki nie wyjście oraz nie pusta kolejka wykonuj
    przypisz w i k współrzędne pola z początku kolejki
    usuń pole z kolejki
    jeśli w = 0 lub w = n - 1 lub k = 0 lub k = n - 1 to
        wyjście ← prawda
    w przeciwnym przypadku
        jeśli Lab[w-1][k] = 0 to
            Lab[w-1][k] ← Lab[w][k] + 1
            wstaw pole (w-1,k) do kolejki
        jeśli Lab[w+1][k] = 0 to
            Lab[w+1][k] ← Lab[w][k] + 1
            wstaw pole (w+1,k) do kolejki
        jeśli Lab[w][k-1] = 0 to
            Lab[w][k-1] ← Lab[w][k] + 1
            wstaw pole (w,k-1) do kolejki
        jeśli Lab[w][k+1] = 0 to
            Lab[w][k+1] ← Lab[w][k] + 1
            wstaw pole (w,k+1) do kolejki
```

Rysunek 2.8 przedstawia labirynt z polami, które algorytm wykorzystujący kolejkę oznaczy liczbami całkowitymi dodatnimi, startując z pola (1,14).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	-1	15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	14	13	12	11	10	9	8	7	6	5	4	3	2	1	2	3	4	5	-1
2	-1	15	-1	-1	-1	-1	-1	-1	-1	-1	6	-1	-1	-1	2	-1	-1	-1	-1	-1
3	-1	0	-1	-1	-1	-1	-1	12	-1	-1	-1	-1	-1	-1	3	-1	-1	-1	-1	-1
4	-1	0	16	15	14	13	12	11	10	9	8	7	6	5	4	5	6	7	-1	-1
5	-1	0	-1	-1	-1	14	-1	-1	11	-1	-1	-1	-1	-1	-1	-1	7	-1	-1	-1
6	-1	0	0	0	16	15	14	13	12	13	14	13	12	11	10	9	8	9	10	-1
7	-1	0	-1	-1	-1	-1	-1	13	-1	-1	-1	-1	-1	-1	11	-1	-1	-1	-1	-1
8	-1	0	-1	0	0	0	16	15	14	15	16	15	14	13	12	13	14	15	16	-1
9	-1	0	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	15	-1	-1	-1	-1	-1	-1	-1
10	-1	0	-1	0	0	0	0	0	0	0	0	0	16	0	0	0	0	0	0	-1
11	-1	0	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1
12	-1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
13	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1	-1
14	-1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
15	-1	0	-1	-1	-1	-1	-1	0	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
17	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1
18	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
19	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Rys. 2.8. Labirynt z polami, które algorytm wykorzystujący kolejkę oznaczy liczbami całkowitymi dodatnimi, gdy polem startowym jest (1,14)

Warto wiedzieć

Algorytm znajdujący najkrótszą drogę do wyjścia z labiryntu z wykorzystaniem kolejki rozpatruje tylko pola znajdujące się w odległości od pola startowego nie większej niż długość najkrótszej drogi. W przykładzie z rysunku 2.8 pola z liczbą 16 zostały oznaczone tą liczbą i dołączone do kolejki, ale nie były rozpatrywane.

Tabela 2.1 przedstawia elementy w kolejce dla 14 początkowych iteracji pętli, gdy polem startowym jest pole (1,14).

Iteracja pętli	Rozpatrywane pole	Stan kolejki
–	–	(1,14)
1	(1,14)	(2,14); (1,13); (1,15)
2	(2,14)	(1,13); (1,15); (3,14)
3	(1,13)	(1,15); (3,14); (1,12)
4	(1,15)	(3,14); (1,12); (1,16)
5	(3,14)	(1,12); (1,16); (4,14)
6	(1,12)	(1,16); (4,14); (1,11)
7	(1,16)	(4,14); (1,11); (1,17)
8	(4,14)	(1,11); (1,17); (4,13); (4,15)
9	(1,11)	(1,17); (4,13); (4,15); (1,10)
10	(1,17)	(4,13); (4,15); (1,10); (1,18)
11	(4,13)	(4,15); (1,10); (1,18); (4,12)
12	(4,15)	(1,10); (1,18); (4,12); (4,16)
13	(1,10)	(1,18); (4,12); (4,16); (2,10); (1,9)
14	(1,18)	(4,12); (4,16); (2,10); (1,9)

Tabela 2.1. Elementy w kolejce przy pierwszych 14 iteracjach pętli dla przykładu z rysunku 2.8

Pierwszy wiersz tabeli 2.1 ze s. 41 to stan przed rozpoczęciem wykonywania pętli. Rozpatrzenie pola startowego powoduje dodanie trzech elementów do kolejki. Zbadanie pól (4,14) oraz (1,10) sprawia, że do kolejki dodajemy za każdym razem dwa elementy. Po rozpatrzeniu pola (1,18) nie dopiszemy do kolejki żadnego pola. Pozostałe pola z tabeli 2.1 powodują, że do kolejki dodajemy po jednym polu.

Aby zaimplementować omówiony algorytm wyznaczania najkrótszej drogi do wyjścia z labiryntu, wykorzystamy szablon kolejki z biblioteki STL. Należy wtedy dodać do programu bibliotekę `queue` za pomocą dyrektywy `#include <queue>`. Ogólna postać **deklaracji kolejki** jest następująca:

Deklaracja kolejki

```
queue<typ elementów kolejki> nazwa_kolejki;
```

Typ `queue` jest typem obiektowym, a więc udostępnia metody do przetwarzania danych. Należą do nich m.in. **metody** `push`, `pop`, `front`, `empty`. Działanie tych metod odpowiada wymienionym wcześniej **operacjom na kolejce**. Parametrem metody `push` jest element, który chcemy umieścić na końcu kolejki. Metody `pop`, `front`, `empty` są bezparametrowe.

Metody `push`, `pop`, `front`, `empty` dla klasy `queue`

Operacje na kolejce, s. 39

Elementami pamiętanymi w kolejce są współrzędne pól, a więc dwie liczby całkowite. Zdefiniujemy w związku z tym typ reprezentujący pojedyncze pole jako **strukturę**:

Struktura, podręcznik *Informatyka na czasie 2. Zakres rozszerzony*, s. 109

```
struct pole
{
    int w, k;
};
```

Oto kod źródłowy funkcji `Droga`, realizującej omawiany algorytm:

Fragment kodu źródłowego programu znajdującego najkrótszą drogę do wyjścia z labiryntu (z wykorzystaniem kolejki) – funkcja `Droga`

```
1.  bool Droga(int Lab[][N], pole p1, pole &p2)
2.  {
3.      int w, k;
4.      bool wyjscie=false;
5.      queue<pole> Q;
6.      Q.push(p1);
7.      Lab[p1.w][p1.k]=1;
8.      while (!wyjscie && !Q.empty())
9.      {
10.         p2=Q.front(); Q.pop(); w=p2.w; k=p2.k;
11.         if (w==0 || w==N-1 || k==0 || k==N-1)
12.             wyjscie=true;
13.         else
14.         {
15.             if (Lab[w-1][k]==0)
16.             {
17.                 Lab[w-1][k]=Lab[w][k]+1;
18.                 p2.w=w-1; p2.k=k; Q.push(p2);
19.             }
```



```
20.             if (Lab[w+1][k]==0)
21.             {
22.                 Lab[w+1][k]=Lab[w][k]+1;
23.                 p2.w=w+1; p2.k=k;
24.                 Q.push(p2);
25.             }
26.             if (Lab[w][k-1]==0)
27.             {
28.                 Lab[w][k-1]=Lab[w][k]+1;
29.                 p2.w=w; p2.k=k-1;
30.                 Q.push(p2);
31.             }
32.             if (Lab[w][k+1]==0)
33.             {
34.                 Lab[w][k+1]=Lab[w][k]+1;
35.                 p2.w=w; p2.k=k+1;
36.                 Q.push(p2);
37.             }
38.         }
39.     }
40.     return wyjscie;
41. }
```

Warto wiedzieć

W przedstawionym programie w każdej z kolejnych instrukcji `if` trzeba aktualizować obie współrzędne pola p2, ponieważ mogły zostać zmienione w poprzedniej instrukcji `if`.

Zwróć uwagę na parametry funkcji `Droga` (linia 1). Parametr p1 (przekazany przez wartość) jest polem startowym, a parametr p2 (przekazany przez referencję) to znalezione współrzędne pola będącego wyjściem z labiryntu, pod warunkiem że droga istnieje, czyli wartością funkcji **`Droga`** jest **`true`**.

W linii 5 jest zadeklarowana kolejka o nazwie Q, służąca do pamiętania elementów typu `pole`. W linii 6 parametr p1 (przechowujący współrzędne pola startowego) jest wstawiany do kolejki, a w linii 7 pole startowe jest oznaczane liczbą 1. Pętla w liniach 8–39 pobiera najpierw z kolejki pierwszy element i w pomocniczych zmiennych w i k zapisuje współrzędne aktualnie rozpatrywanego pola (linia 10). Jeśli jest to pole będące wyjściem z labiryntu, pętla kończy działanie (zmienna logiczna `wyjscie` przyjmuje wartość **`true`** – linia 12). W przeciwnym przypadku badane są cztery sąsiednie pola (linie 13–38). Jeśli dane pole nie zostało jeszcze odwiedzone (jest oznaczone liczbą 0), to jest mu przypisywana wartość o jeden większa od wartości zapisanej dla bieżącego pola, a następnie pole jest dopisywane na końcu kolejki.

Kod źródłowy funkcji `main` wykorzystującej funkcję `Droga` znajduje się na s. 44. Funkcje `WczytajLabirynt`, `WypiszLabirynt` i `OznaczDroge` są takie same jak w programie rekurencyjnym szukającym drogi w labiryncie.

Rysunek 2.9 na s. 44 przedstawia przykład wywołania programu znajdującego najkrótszą drogę prowadzącą do wyjścia z labiryntu, jeśli polem startowym jest pole (16,10).

Fragment kodu źródłowego programu znajdującego najkrótszą drogę do wyjścia z labiryntu (z wykorzystaniem kolejki) – funkcja `main`

```
1.  int main()
2.  {
3.      int Lab[N][N];
4.      pole p1, p2;
5.      WczytajLabirynt(Lab);
6.      WypiszLabirynt(Lab);
7.      cout<<"Wspolrzedne pola startowego: "<<endl;
8.      cout<<"w= "; cin>>p1.w;
9.      cout<<"k= "; cin>>p1.k;
10.     if (Droga(Lab,p1,p2))
11.     {
12.         OznaczDroge(Lab,p2.w,p2.k);
13.         WypiszLabirynt(Lab);
14.     }
15.     else cout<<"Brak drogi";
16.     return 0;
17. }
```

Warto wiedzieć

Może istnieć kilka najkrótszych dróg, jednak algorytm znajdzie tylko jedną z nich. Na przykład na rysunku 2.9 z pola (6,8) można przejść na pole (5,8), potem na pole (4,8), a następnie w lewo do pola (4,1) i dalej do wyjścia.

```
Wspolrzedne pola startowego:
w = 16
k = 10
   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
 0 X D X X X X X X X X X  X  X  X  X  X  X  X  X  X
 1 X D                                             X
 2 X D X X X X X X X X    X  X  X     X  X  X  X  X
 3 X D X X X X X    X X X  X  X  X     X  X  X  X  X
 4 X D                                          X  X
 5 X D X X X    X X    X X  X  X  X  X  X  X     X  X
 6 X D D D D D D D D                               X
 7 X    X X X  X X X D X X  X  X  X  X  X     X  X  X
 8 X    X       D D D D                            X
 9 X    X X X D X X X X X X    X  X  X  X  X  X  X  X
10 X    X       D D D D                            X
11 X    X X X X X X D X X X  X  X  X     X  X  X  X
12 X    X             D D D D                      X
13 X    X X X X X X X X X D  X  X  X  X  X  X  X  X
14 X    X                 D                        X
15 X    X X X X X    X X X D  X  X  X  X  X  X  X  X
16 X    X                 D D                      X
17 X    X X X X X X X X X X  X  X  X  X  X  X     X
18 X X                                            X
19 X X X X X X X X X X X X  X  X  X  X  X  X  X  X
```

Rys. 2.9. Labirynt z zaznaczoną najkrótszą drogą do wyjścia z pola (16,10)

Ćwiczenie 4

Napisz program, który znajdzie najkrótszą drogę prowadzącą od wczytanego pola do wyjścia z labiryntu oraz wypisze planszę ze znalezioną drogą. Jeśli droga prowadząca do wyjścia nie istnieje, program wyświetli odpowiedni komunikat. Opis labiryntu wczytaj z pliku otrzymanego od nauczyciela (np. *labirynt.txt*).

Podsumowanie

- Rekurencyjny algorytm znajdowania drogi wyjścia z labiryntu polega na rekurencyjnym wywoływaniu funkcji poszukującej drogi dla tych sąsiednich pól, na które można stanąć i które nie były jeszcze rozpatrywane.
- Algorytm rekurencyjny znajdowania drogi wyjścia z labiryntu nie musi znaleźć najkrótszej możliwej drogi.
- Rekurencyjny algorytm poszukujący drogi wyjścia z labiryntu jest przykładem algorytmu przeszukiwania z nawrotami. Algorytm taki przegląda możliwe rozwiązania, a gdy okazuje się, że kandydat na rozwiązanie nie jest właściwy, powraca do punktu, w którym może modyfikować rozwiązanie.
- Kolejka to dynamiczna struktura danych typu FIFO (ang. *first in, first out*), w której operacje są wykonywane na jej początku i końcu. Z kolejki można pobrać jedynie element znajdujący się na początku. Nowy element można dopisać tylko na końcu kolejki.
- Przykładowe operacje na kolejce to: umieszczenie elementu na końcu kolejki (*push*), usunięcie elementu z początku kolejki (*pop*), pobranie wartości pierwszego elementu (*front*) i sprawdzenie, czy kolejka jest pusta (*empty*).
- Najkrótszą drogę do wyjścia z labiryntu można znaleźć za pomocą algorytmu iteracyjnego z wykorzystaniem kolejki. W kolejce przechowywane są współrzędne pól, które należy jeszcze rozpatrzeć.

Zadania

1. Napisz program symulujący kolejkę do kasy. Wprowadzenie z klawiatury liczby 1 oznacza, że na końcu kolejki staje klient, a wpisanie liczby –1 oznacza, że klient z początku kolejki został obsłużony. Podanie liczby 0 kończy wczytywanie. Program powinien wypisać największą liczbę klientów, którzy jednocześnie stali w kolejce. Na przykład dla liczb 1 1 1 –1 1 1 –1 0 program zwróci liczbę 4. Załóż, że wprowadzane dane są poprawne, a więc nie zostanie podana liczba –1, gdy w kolejce nikt nie stoi.
2. Przedstawiony w tym temacie algorytm znajdujący najkrótszą drogę do wyjścia z labiryntu zmodyfikuj tak, aby do przechowania pól labiryntu, które należy rozpatrzeć, zamiast kolejki zastosować stos. Czy zmodyfikowany algorytm zawsze znajdzie drogę do wyjścia w labiryncie, jeśli ona istnieje? Czy będzie to najkrótsza droga?
3. Zaprojektuj planszę z labiryntem o n wierszach i m kolumnach, a następnie utwórz plik tekstowy opisujący tę planszę. Napisz program, który znajdzie najkrótszą drogę w labiryncie o n wierszach i m kolumnach, a następnie sprawdź jego działanie, korzystając z przygotowanego pliku.

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

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

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

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

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

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

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

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

3. Wykorzystanie list w rozwiązywaniu problemów

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

Cele lekcji

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

3.1. Czym jest lista?

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

Lista

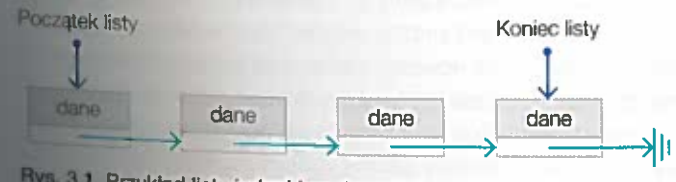
Dynamiczna struktura danych, s. 12

Stos, s. 11
Kolejka, s. 39

Rodzaje list

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

Lista jednokierunkowa



Rys. 3.1. Przykład listy jednokierunkowej złożonej z czterech elementów

Warto wiedzieć

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