Politechnika Wrocławska

Wydział Elektroniki, Fotoniki i Mikrosystemów

Laboratorium informatyki

Ćwiczenie nr 10. Algorytmy i struktury danych

Zagadnienia do opracowania:

- metody opisu algorytmów
- paradygmat *dziel i zwyciężaj*
- wybrane algorytmy sortowania i ich implementacje
- lista, stos i kolejka jako przykłady struktur danych

Spis treści

1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z podstawowymi algorytmami i strukturami danych oraz ich implementacją w językach *C* i *C++*.

2. Wprowadzenie

2.1. Algorytmy

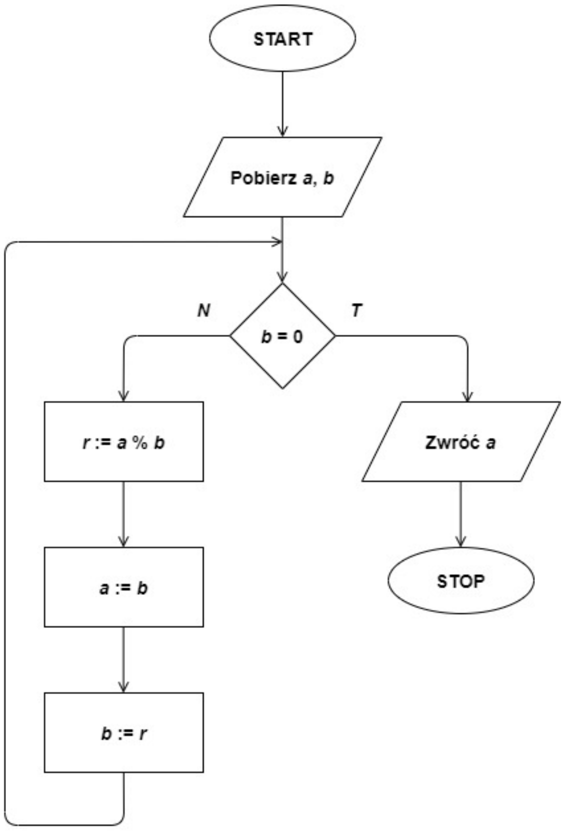
Algorytmem nazywamy **skończony** ciąg czynności prowadzący do rozwiązania określonego problemu. Algorytmy mogą być opisywane z wykorzystaniem różnych (równoważnych) metod takich, jak:

- opis słowny;
- lista kroków;
- schemat blokowy;
- pseudokod;
- język programowania.

Za przykład może posłużyć popularny algorytm wyznaczania największego wspólnego dzielnika dwóch liczb $\boldsymbol{a}$ i $\boldsymbol{b}$, zwany ***algorytmem Euklidesa***. Można go zapisać w następujący sposób:

1. jeżeli $\boldsymbol{b}$ jest równe $\mathbf{0}$, zwróć $\boldsymbol{a}$, w przeciwnym razie:
2. oblicz $\boldsymbol{r}$ jako resztę z dzielenia $\boldsymbol{a}$ przez $\boldsymbol{b}$
3. podstaw $\boldsymbol{a} \longleftarrow \boldsymbol{b}$ oraz $\boldsymbol{b} \longleftarrow \boldsymbol{r}$
4. przejdź do punktu 1.

Można go również przedstawić w postaci schematu blokowego (rys. 2.1).



Rys. 2.1. Reprezentacja algorytmu Euklidesa w postaci schematu blokowego

Algorytm Euklidesa może zostać zaimplementowany w języku C w postaci funkcji rekurencyjnej ***gcd()*** (*ang. greatest common divisor*):

```
int gcd(int a, int b) {
    return (b == 0) ? a : gcd(b, a % b);
}
```

Rzecz jasna, jest to jedna z możliwych implementacji (porównaj: implementację iteracyjną i rekurencyjną algorytmu Euklidesa, Ćw. 5). Odmienne sposoby realizacji algorytmu będą się różnić między sobą czasem wykonania programu, ilością wykorzystywanych zasobów, czytelnością kodu, itp. Popularnym sposobem opisu ***złożoności obliczeniowej*** algorytmu jest ***notacja dużego O*** (patrz: Ćw. 5).

Przykładem algorytmu o liniowej złożoności obliczeniowej *O(n)* może być algorytm wyszukiwania wartości minimalnej (lub maksymalnej) w zbiorze. Przykład implementacji przedstawiono na listingu 1. Funkcja ***min()*** przyjmuje jako argumenty stały wskaźnik na tablicę liczb całkowitych ***tab*** (algorytm nie modyfikuje tablicy), rozmiar tablicy ***n*** oraz wskaźnik na bufor, do którego ma zostać zapisana wyszukana wartość minimalna zbioru (tablicy) ***buffer***. Funkcja zwraca wartość *true* w przypadku powodzenia albo wartość *false* w przeciwnym razie, np. w razie pustej tablicy. Takie rozwiązanie zastosowano ze względu na brak możliwości rozróżnienia wyszukanej wartości minimalnej od kodu błędu (np. -1) – obie wartości to liczby całkowite. Stąd, rozwiązanie, w którym funkcja ***min()*** zwracałaby wartość minimalną zamiast ***flagi powodzenia operacji*** prowadziłaby do niejednoznaczności w kodzie. [*Uwaga: Język C++ wprowadził doskonalsze techniki obsługi błędów, zwane wyjątkami. Patrz: Dodatek, Ćw. 8*] Algorytm na początku zapisuje do bufora wyjściowego wartość pierwszego elementu tablicy, a następnie, iterując po wszystkich pozostałych elementach, dokonuje porównania wartości przechowywanej w tablicy pod kolejnym indeksem z wartością aktualnie przechowywaną w buforze. Jeżeli wartość w buforze jest większa niż sprawdzany element tablicy, następuje nadpisanie zawartości bufora mniejszą wartością. Aby algorytm wyszukiwał wartość maksymalną zbioru wystarczy zmienić operator porównania.

```c
#include <stdio.h>
#include <stdbool.h>

bool min(const int * tab, unsigned int n, int *
    buffer) {
    bool result = false;
    if (tab != NULL && buffer != NULL) {
```

```
        // Zapisz do bufora pierwszy element
        *buffer = tab[0];
        // Iteruj po elementach 1 ... n - 1
        for (unsigned int i = 1; i < n; ++i)
            // Jezeli aktualna wartosc bufora jest wieksza niz tab[i]
            if (*buffer > tab[i])
                // Zapisz do bufora tab[i]
                *buffer = tab[i];
        // Sukces
        result = true;
    }
    return result;
}

int main() {
    int tab[] = {3, -1, 2, 5, -7, 0, 2};
    int value;
    bool result;
    result = min(tab, sizeof(tab) / sizeof(int), &value);
    if (result) printf("Min value of array: %d\n", value);
    else printf("Invalid array\n");
  return 0;
}
```

Listing 1. Implementacja algorytmu wyszukiwania wartości minimalnej

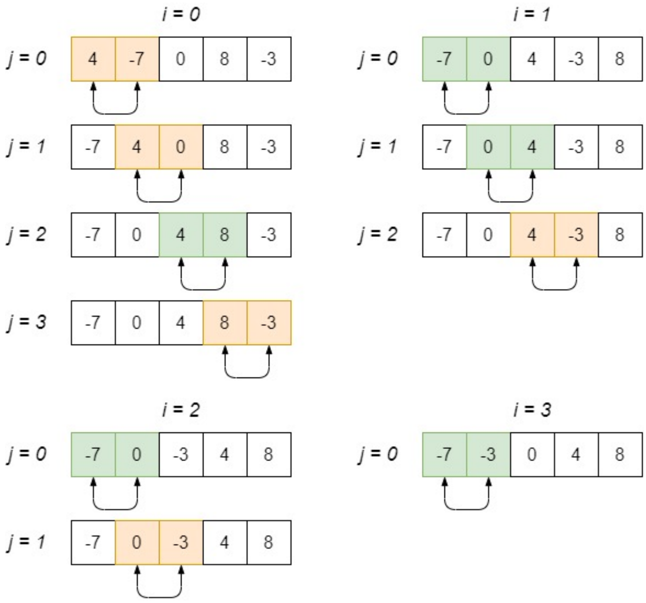
2.2. Algorytmy sortowania

Ponieważ temat algorytmów jest obszerny i mógłby sam w sobie stanowić materiał na osobny kurs, w ramach tego ćwiczenia zostaną omówione jedynie wybrane algorytmy, w szczególności algorytmy sortowania. Reprezentantem tej grupy jest ***algorytm sortowania bąbelkowego*** (*ang. bubble sort*). Niech dana będzie tablica liczb ***tab*** o rozmiarze ***n***. Funkcja realizująca algorytm ***sortowania bąbelkowego*** wykonuje $n-1$ przejść przez elementy tablicy, dokonując porównania dwóch sąsiednich elementów i ewentualnej zamiany ich kolejności. W związku z tym w każdym przejściu funkcja wykonuje $n-i$ porównań, gdzie $i = 1, 2, 3, ..., n-1$ to numer kolejnego przejścia przez elementy tablicy. Czasowa złożoność obliczeniowa dla tego algorytmu, w notacji dużego O, wynosi $O(n^2)$, co w praktyce ogranicza jego zastosowanie do celów dydaktycznych (prosta implementacja). Realizację ***algorytmu sortowania bąbelkowego*** (w kolejności rosnącej) przedstawiono na listingu 2. Kolejne etapy sortowania tablicy przez funkcję ***bubbleSort()*** przedstawiono na rys. 2.2.

```c
#include <stdio.h>

void bubbleSort(int * tab, unsigned int n)
{
    if (tab != NULL)
        // n - 1 przejsc
        for (unsigned int i = 0; i < n - 1; ++i)
            // n - i porownan (-1 dla i = 0, 1, 2, ...)
            for (unsigned int j = 0; j < n - i - 1; ++j)
                if (tab[j] > tab[j + 1]) {
                    // zmiana kolejnosci elementow
```

```
                    int tmp = tab[j];
                    tab[j] = tab[j + 1];
                    tab[j + 1] = tmp;
                }
}

int main() {
    const unsigned int tabSize = 5;
    int tab[tabSize] = {4, -7, 0, 8, -3};
    bubbleSort(tab, tabSize);
    for (unsigned int i = 0; i < tabSize; ++i)
        printf("%d", tab[i]);
    return 0;
}
```

Listing 2. Implementacja algorytmu sortowania bąbelkowego



Rys. 2.2. Kolejne kroki procesu sortowania za pomocą algorytmu *bubble sort*

Kod z listingu 2. można zoptymalizować kończąc algorytm, jeżeli w poprzednim przejściu nie została przeprowadzona ani jedna zamiana kolejności elementów tablicy (implementacja z listingu 2. **zawsze** wykonuje $n - 1$ przejść):

```
void bubbleSort(int * tab, unsigned int n)
{
    if (tab != NULL) {
        for (unsigned int i = 0; i < n - 1; ++i) {
            // Flaga przerwania algorytmu
            bool wasSwapped = false;
            for (unsigned int j = 0; j < n - i - 1; ++j)
                if (tab[j] > tab[j + 1]) {
                    int tmp = tab[j];
                    tab[j] = tab[j + 1];
                    tab[j + 1] = tmp;
                    // Zmieniono kolejnosc elementow
                    wasSwapped = true;
                }
            // Jesli nie zmieniono kolejnosci, zakoncz algorytm
            if (!wasSwapped)
                break;
        }
    }
}
```

Inne, bardziej zaawansowane (zoptymalizowane czasowo) algorytmy sortowania takie, jak ***algorytm szybkiego sortowania*** (*ang. quick sort*) czy ***algorytm sortowania przez scalanie*** (*ang. merge sort*), bazują na paradygmacie ***dziel i zwyciężaj*** (*ang. divide and conquer*). Jest to metoda projektowania algorytmów zakładająca rekurencyjny podział złożonego problemu na mniejsze problemy tak długo, aż powstałe zadania będą trywialne do rozwiązania. Pomniejsze wyniki łączy się, otrzymując ostateczne rozwiązanie. Złożoność czasowa ***algorytmów szybkiego sortowania*** i ***sortowania przez scalanie*** w notacji dużego O wynosi $O(nlog(n))$.

Algorytm szybkiego sortowania może zostać opisany w następujący sposób:

1. wybierz (losowo) punkt podziału tablicy (***piwot***)
2. w jednej tablicy umieść wszystkie elementy mniejsze lub równe, a w drugiej większe od elementu rozdzielającego
3. jeżeli powstała po podziale tablica zawiera co najwyżej jeden element – zakończ sortowanie, w przeciwnym razie przejdź do punktu 1.

Jest wiele technik doboru punktu podziału tablicy (***piwota***):

- pierwszy element tablicy;
- ostatni element tablicy;
- środkowy element tablicy;
- losowy element tablicy.

Na listingu 3. przedstawiono przykładową implementację ***algorytmu szybkiego sortowania*** w języku *C*. Jako ***piwot*** obierany jest ostatni element tablicy (*algorytm partycjonowania Nico Lomuto*). Kolejne etapy sortowania tablicy przez funkcję ***quickSort()*** przedstawiono na rys. 2.3.

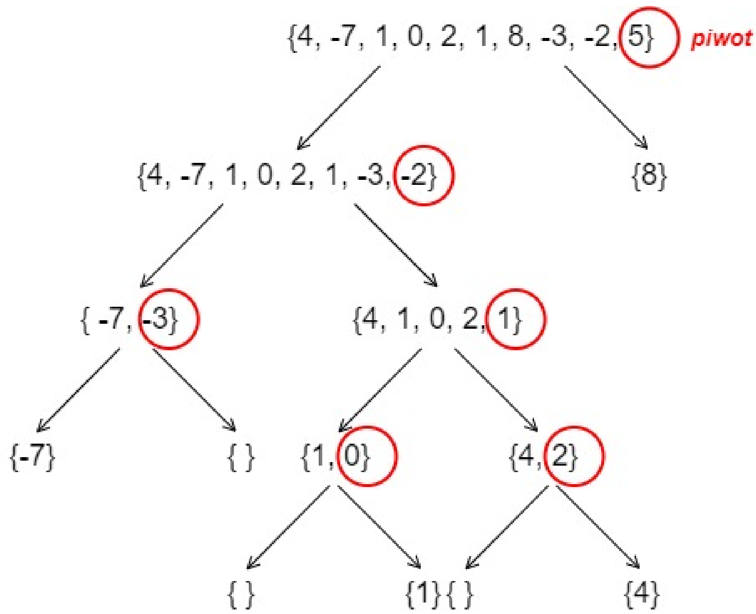
```c
#include <stdio.h>

// Funkcja wyswietlajaca zawartosc tablicy
void printTab(const int * tab, unsigned int size) {
    if (tab != NULL)
        for (unsigned int i = 0; i < size; ++i)
            printf("%d", tab[i]);
}

// Funkcja zmieniajaca kolejnosc elementow tablicy
void swap(int * a, int * b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

// Funkcja partycjonowania tablicy
int partition(int * tab, int lo, int hi) {
    // Ostatni element tablicy to piwot
    int pivot = tab[hi];
    // Ustaw punkt podzialu tablicy na indeks poczatkowy
    int pivotIdx = lo;
    // Iteruj po zakresie indeksow [lo; hi)
    for (unsigned int i = lo; i < hi; ++i) {
        // Jezeli element tablicy jest niewiekszy niz piwot
        if (tab[i] <= pivot) {
            // Umiesc element przed punktem podzialu tablicy
            swap(tab + i, tab + pivotIdx);
```



```
            // Zaktualizuj indeks podzialu
            pivotIdx++;
        }
    }
    // Umiesc piwot na wlasciwej pozycji
    swap(tab + hi, tab + pivotIdx);
    return pivotIdx;
}

/*
    tab - tablica do posortowania
    lo - indeks poczatkowy
    hi - indeks koncowy
*/
void quickSort(int * tab, int lo, int hi) {
    if (tab != NULL && hi > lo) {
        // Wyznacz indeks podzialu tablicy
        int pivotIdx = partition(tab, lo, hi);
        // Osobno posortuj tablice powstale po podziale
        quickSort(tab, lo, pivotIdx - 1);
        quickSort(tab, pivotIdx + 1, hi);
    }
}

int main() {
    const unsigned int tabSize = 10;
    int tab[tabSize] = {4, -7, 1, 0, 2, 1, 8, -3, -2, 5};
    quickSort(tab, 0, tabSize - 1);
    printTab(tab, tabSize);
```

```
    return 0;
}
```

Listing 3. Implementacja algorytmu szybkiego sortowania



Rys. 2.3. Kolejne kroki procesu sortowania za pomocą algorytmu *quick sort*

2.3. Struktury danych

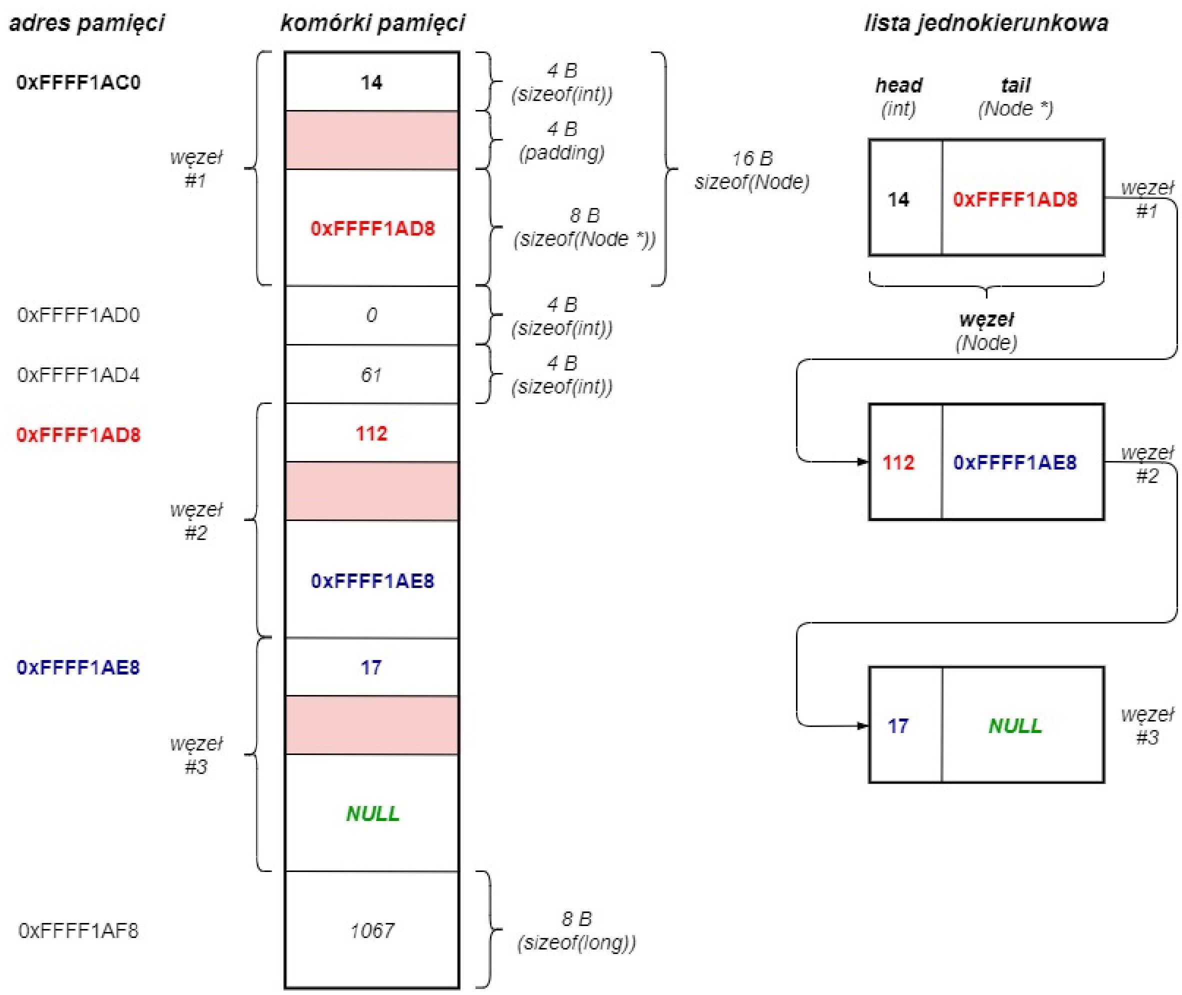
Strukturą danych nazywa się skończony zbiór obiektów (*węzłów*) z funkcją wyznaczania następnika w zbiorze. Przykładem ***struktur danych*** są m.in. ***tablica***, ***lista***, ***stos***, ***kolejka***, ***drzewo*** czy ***graf***. ***Struktury danych*** znajdują szerokie zastosowanie w informatyce i współistnieją razem z ***algorytmami***, przeprowadzającymi na nich określone operacje (sortowanie, przeszukiwanie, dodawanie czy usuwanie elementów). W **paradygmacie programowania obiektowego** często ***algorytmy*** zawierają się wewnątrz ***struktur danych*** w postaci ***metod klas***. Począwszy od standardu ***C++11*** wiele gotowych i zoptymalizowanych rozwiązań ***struktur danych*** oraz operujących na nich ***algorytmów*** jest dostępnych w ramach ***Standardowej Biblioteki Szablonów*** (*ang. Standard Template Library, STL*).

2.3.1. Lista

Lista (*ang. linked list*) to przykład ***struktury danych***, której elementy ułożone są w porządku liniowym, ale **nie muszą znajdować się w ciągłym obszarze pamięci**. Każdy element, zwany ***węzłem*** (*ang. node*), zawiera swoją **wartość** (*ang. head*) oraz ***wskaźnik do następnika*** (*ang. tail*). W przypadku ***list dwukierunkowych*** każdy węzeł posiada również **wskaźnik do poprzednika**. Ostatni ***węzeł listy*** wskazuje na adres ***NULL***. Analogicznie w przypadku ***list dwukierunkowych*** pierwszy ***węzeł*** wskazuje na ***NULL***, jako element poprzedzający. Dzięki takiej implementacji możliwe jest iterowanie po ***węzłach listy*** podobnie, jak po elementach tablicy, z tą różnicą, że poszczególne ***węzły listy*** mogą być położone w dowolnych lokalizacjach w pamięci komputera. Dodawanie lub usuwanie kolejnych ***węzłów listy*** jest przeprowadzane przez zmianę wartości wskaźnika (*tail*) ***następnika*** (i ***poprzednika*** w przypadku ***list dwukierunkowych***). W ten sposób rozmiar ***listy*** może być dynamicznie zmieniany w trakcie działania programu (w przeciwieństwie do tablicy). Na rys. 2.4. przedstawiono schematyczną strukturę trzyelementowej ***listy jednokierunkowej***. ***Węzeł listy***, przechowujący liczbę całkowitą, można zaimplementować wykorzystując strukturę *Node*, przedstawioną na listingu 4.

```c
typedef struct Node {
    int head;
    struct Node * tail;
} Node_t;
```

Listing 4. Struktura węzła listy jednokierunkowej



Rys. 2.4. Struktura listy jednokierunkowej

Każdy ***węzeł*** jest osobno tworzony na ***stercie***, wykorzystując funkcję ***createNode()***:

```
Node_t * createNode(int head, Node_t * tail) {
    // Dynamicznie utworz wezel
    // Rownowaznie: (Node_t *) malloc(sizeof(Node_t))
    Node_t * node = (Node_t *) malloc(sizeof(*node));
```

```
    // Zainicjalizuj wezel
    node->head = head;
    node->tail = tail;
    // Zwroc utworzony wezel
    return node;
}
```

Kolejne ***węzły*** mogą być dodawane na koniec (***pushBack()***) lub na początek ***listy*** (***pushFront()***). Przejście do kolejnego węzła listy odbywa się przez odwołanie do ***wskaźnika tail***. Funkcja ***pushFront()*** przyjmuje *wskaźnik na wskaźnik* na początek listy, ponieważ **konieczna jest modyfikacja samego wskaźnika**, a nie zmiennej przez niego wskazywanej – pod wartość wskaźnika na początek listy przypisywany jest adres nowo utworzonego elementu ($*root = node$):

```
void pushBack(Node_t * root, int value) {
    Node_t * currentNode = root;
    if (currentNode != NULL) {
        // Przejdz na koniec listy
        while (currentNode->tail != NULL)
            currentNode = currentNode->tail;
        // Dodaj nowy element na koniec listy
        currentNode->tail = createNode(value, NULL);
    }
}

void pushFront(Node_t ** root, int value) {
    if (root != NULL) {
        // Utworz nowy wezel na poczatku listy
        // *root to adres poczatku listy
       Node_t * node = createNode(value, *root);
        *root = node;
```

```
    }
}
```

Zdejmowanie elementów z ***listy*** można zaimplementować w sposób analogiczny. Funkcje ***popFront()*** i ***popBack()*** przyjmują jako argumenty *wskaźnik na wskaźnik* na początek listy oraz wskaźnik na bufor, do którego ma zostać zapisana wartość zdejmowanego ***węzła***. W przypadku niepowodzenia funkcje zwracają wartość ***false***.

```
bool popBack(Node_t ** root, int * buffer) {
    bool result = false;
    if (root != NULL && *root !=NULL && buffer !=
  NULL) {
        Node_t * currentNode = *root;
        // Jezeli lista ma jeden element, usun go
        if (currentNode->tail == NULL) {
            // Odczytaj wartosc
            *buffer = currentNode->head;
            // Usun element
            free(currentNode);
            *root = NULL;
        }
        // W przeciwnym wypadku przejdz do
  przedostatniego elementu
        else {
            while (currentNode->tail->tail != NULL)
                currentNode = currentNode->tail;
            // Odczytaj wartosc ostatniego elementu
            *buffer = currentNode->tail->head;
            // Usun ostatni element
            free(currentNode->tail);
```

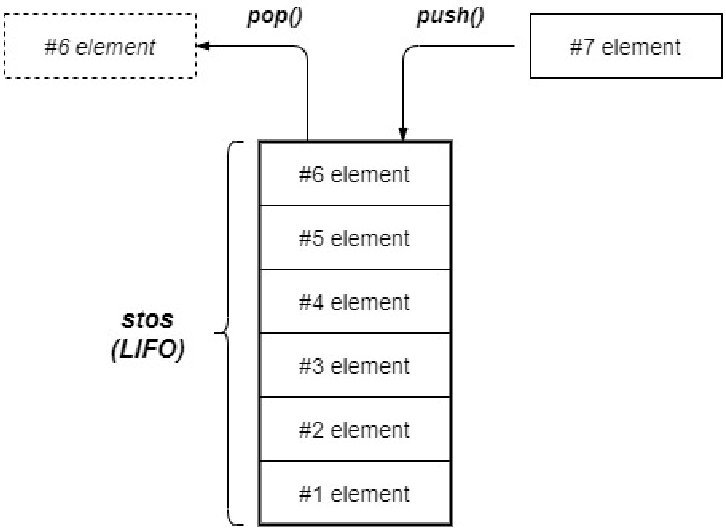
```
            // Ustaw koniec listy na przedostatnim elemencie
            currentNode->tail = NULL;
        }
        // Sukces
        result = true;
    }
    return result;
}

bool popFront(Node_t ** root, int * buffer) {
    bool result = false;
    if (root != NULL && *root != NULL && buffer != NULL) {
        // Odczytaj wartosc pierwszego elementu
        *buffer = (*root)->head;
        // Wez drugi element z listy
        Node_t * next = (*root)->tail;
        // Usun pierwszy element
        free(*root);
        // Ustaw poczatek listy na drugi element
        *root = next;
        // Sukces
        result = true;
    }
    return result;
}
```

Korzystając z tak zaimplementowanej ***listy*** można odtworzyć strukturę przedstawioną na rys. 2.4:

```c
Node_t * root = createNode(112, NULL);
pushBack(root, 17);
pushFront(&root, 14);

int value;
bool result;
result = popFront(&root, &value);
if (result) printf("Popped value: %d\n", value);
result = popBack(&root, &value);
if (result) printf("Popped value: %d\n", value);
result = popFront(&root, &value);
if (result) printf("Popped value: %d\n", value);
```

2.3.2. Stos

Stos (*ang. stack*) to struktura liniowo uporządkowanych danych, realizująca założenia bufora ***LIFO*** (*ang. Last in, First out*) (rys. 2.5). Często przytaczaną analogią z życia codziennego jest stos książek. Książka, która została położona na samą górę (jako ostatnia), zostanie zdjęta ze stosu w pierwszej kolejności. W przeciwnym wypadku stos może się rozpaść. ***Stos*** można zaimplementować wykorzystując ***listę jednokierunkową***, ograniczając możliwość dodawania i usuwania elementów do jednej ze stron ***listy***. Przykład implementacji ***stosu*** w języku *C* przedstawiono na listingu 5.



Rys. 2.5. Struktura stosu

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

typedef struct Node {
    int head;
    struct Node * tail;
} Node_t;

Node_t * createNode(int value) {
    Node_t * node = (Node_t *) malloc(sizeof(*node));
    node->head = value;
    node->tail = NULL;
    return node;
}

bool isEmpty(Node_t * root) {
    // Rownowaznie root == NULL
```

```
    return !root;
}

void push(Node_t ** root, int value) {
    if (root) {
        // Utworz nowy wezel
        Node_t * node = createNode(value);
        // Dowiaz element do stosu
        node->tail = *root;
        // Ustaw element na wierzcholek stosu
        *root = node;
    }
}

bool pop(Node_t ** root, int * buffer) {
    bool result = false;
    // Rownowaznie if (root != NULL && *root != NULL
  )
    if (root && *root) {
        Node_t * node = *root;
        // Przesun wskaznik wierzcholka stosu
        *root = (*root)->tail;
        *buffer = node->head;
        free(node);
        // Sukces
        result = true;
    }
    return result;
}

int main() {
```

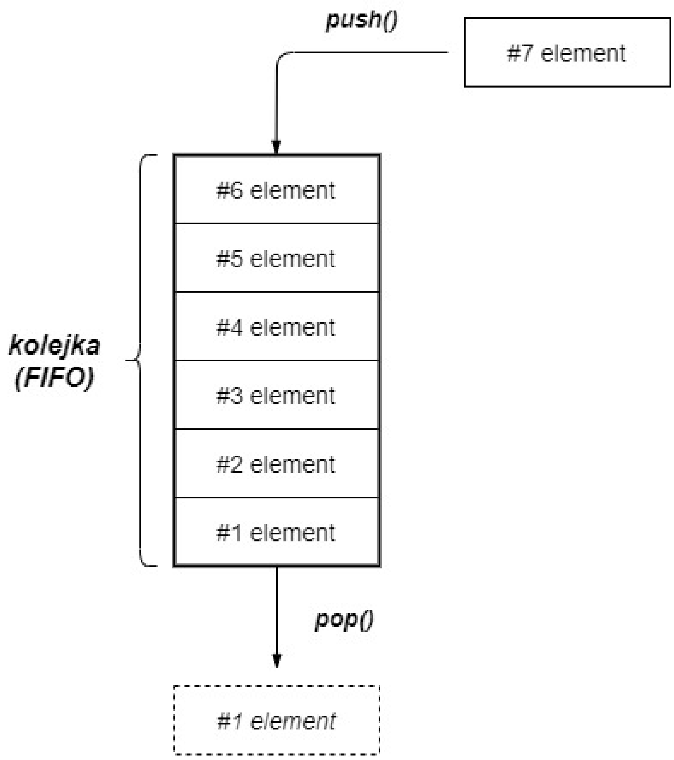
```c
    Node_t * root = NULL;
    push(&root, 14);
    push(&root, 112);
    push(&root, 17);

    int value;
    bool result;
    result = pop(&root, &value);
    if (result) printf("Popped value: %d\n", value);
    result = pop(&root, &value);
    if (result) printf("Popped value: %d\n", value);
    result = pop(&root, &value);
    if (result) printf("Popped value: %d\n", value);
    return EXIT_SUCCESS;
}
```

Listing 5. Implementacja struktury danych stosu

2.3.3. Kolejka

Kolejka (*ang. queue*) to struktura liniowo uporządkowanych danych, realizująca założenia bufora ***FIFO*** (*ang. First in, First out*). Oznacza to, że pierwszy element, który został dodany do ***kolejki*** zostanie z niej zdjęty w pierwszej kolejności (rys. 2.6), w przeciwieństwie do ***stosu***, w którym ostatnio dodany element jest równocześnie elementem zdejmowanym jako pierwszy. ***Kolejka*** może być zaimplementowana za pomocą funkcji:

- ***push()*** – dodającej element na koniec kolejki;
- ***pop()*** – zdejmującej element z początku kolejki;
- ***isEmpty()*** – sprawdzającej czy kolejka jest pusta.



Rys. 2.6. Struktura kolejki

3. Program ćwiczenia

Zadanie 1. Zaimplementuj funkcję ***bool median(const int * tab, unsigned int n, float * buffer)*** wyznaczającą medianę zbioru liczb całkowitych zapisanych w tablicy ***tab*** o rozmiarze ***n***. Funkcja zapisuje wyznaczoną wartość do bufora ***buffer***. Funkcja zwraca wartość ***true*** w przypadku powodzenia albo wartość ***false*** w przeciwnym wypadku. Przetestuj działanie zaimplementowanej funkcji. [*Uwaga: funkcja nie może modyfikować oryginału tablicy (const int * tab)*]

Zadanie 2. Wykorzystując implementację ***listy jednokierunkowej*** przedstawioną w rozdziale **2.3.1**, napisz funkcję ***void removeByIndex(Node_t ** root, unsigned int index)*** usuwającą ***węzeł listy jednokierunkowej*** o indeksie ***index***. Jeżeli lista nie posiada elementu o zadanym indeksie, to funkcja nie modyfikuje listy. Przetestuj działanie zaimplementowanej funkcji, rozpatrz następujące przypadki:

- pusta lista – usuwanie węzła o indeksie 0;
- pusta lista – usuwanie węzła o indeksie większym niż 0;
- lista jednoelementowa - usuwanie węzła o indeksie 0;
- lista jednoelementowa - usuwanie węzła o indeksie większym niż 0;
- lista n-elementowa – usuwanie węzła o indeksie 0;
- lista n-elementowa – usuwanie węzła o indeksie 1;
- lista n-elementowa – usuwanie węzła o indeksie n + 1.

Zadanie 3. Wykorzystując strukturę ***Node_t*** (listing 4.), zaimplementuj ***kolejkę*** definiując funkcje ***void push(Node_t ** root, int value)***, ***bool pop(Node_t ** root, int * buffer)*** oraz ***bool isEmpty(Node_t * root)***. Przetestuj działanie zaimplementowanej struktury danych.

4. Dodatek

4.1. Algorytm sortowania przez scalanie

Algorytm sortowania przez scalanie (*ang. merge sort*) wpisuje się, obok ***algorytmu szybkiego sortowania***, w paradygmat ***dziel i zwyciężaj*** (*ang. divide and conquer*). Można go opisać w następujący sposób:

1. dziel tablicę liczb na pół tak długo, aż każda podtablica będzie zawierać tylko jeden element

2. scalaj podzielone tablice porządkując łączone elementy do otrzymania posortowanej tablicy

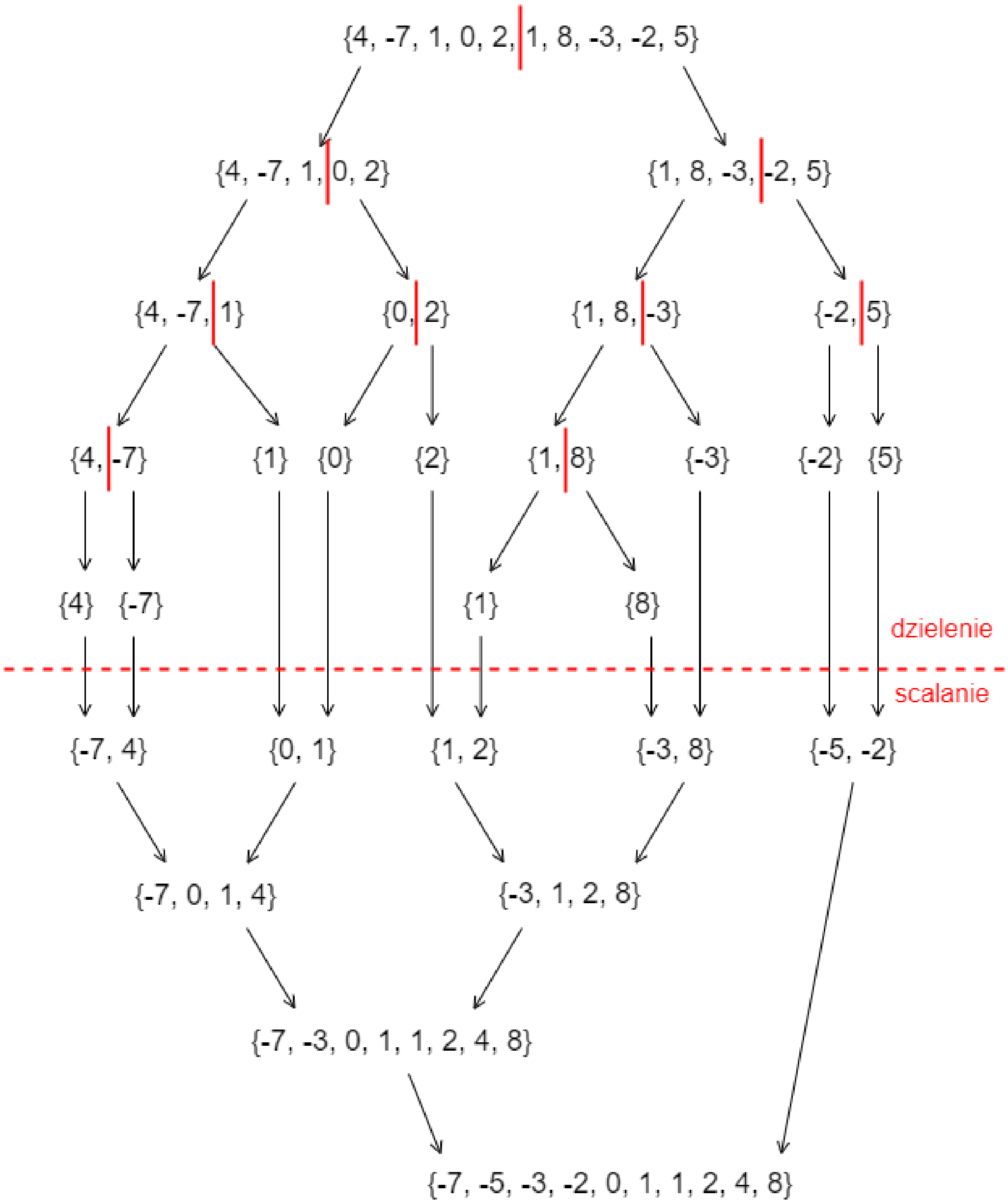
Na listingu 6. przedstawiono przykładową implementację ***algorytmu sortowania przez scalanie*** w języku *C*. Kolejne etapy sortowania tablicy przez funkcję ***mergeSort()*** przedstawiono na rys. 4.1.

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

// Funkcja wyswietlajaca zawartosc tablicy
void printTab(const int * tab, unsigned int size) {
    if (tab)
        for (unsigned int i = 0; i < size; ++i)
            printf("%d", tab[i]);
}

// Funkcja scalania tablic
void merge(int * array, unsigned int lo, unsigned
   int mi, unsigned int hi) {
    // Wykonaj kopie przekazanej tablicy
  const unsigned int tabSize = hi + 1;
```

```c
  int arrayCopy[tabSize];
  memcpy(arrayCopy, array, sizeof(*arrayCopy) * tabSize);

  unsigned int iterator = lo;
  unsigned int loIterator = lo;
  unsigned int hiIterator = mi + 1;
    // Scalaj tablice do tablicy wynikowej az do wyczerpania elementow jednej z podtablic
  while (loIterator <= mi && hiIterator <= hi)
    if (arrayCopy[loIterator] <= arrayCopy[hiIterator])
      array[iterator++] = arrayCopy[loIterator++];
    else
      array[iterator++] = arrayCopy[hiIterator++];

    // Dopisz pozostale elementy pierwszej podtablicy do tablicy wynikowej, jesli istnieja
  while (loIterator <= mi)
    array[iterator++] = arrayCopy[loIterator++];

    // Dopisz pozostale elementy drugiej podtablicy do tablicy wynikowej, jesli istnieja
  while (hiIterator <= hi)
    array[iterator++] = arrayCopy[hiIterator++];
}

/*
    array - tablica do posortowania
    lo - indeks poczatkowy
    hi - indeks koncowy
```

```
*/
void mergeSort(int * array, unsigned int lo,
    unsigned int hi) {
  if (array && hi > lo) {
        // Znajdz punkt podzialu tablicy
        unsigned int mi = (lo + hi) / 2;
        // Wywolania rekurencyjne na kazdej z
    powstalych podtablic
        mergeSort(array, lo, mi);
        mergeSort(array, mi + 1, hi);
        // Scal tablice
        merge(array, lo, mi, hi);
    }
}

int main() {
    int tab[] = {4, -7, 1, 0, 2, 1, 8, -3, -2, 5};
    const unsigned int tabSize = sizeof(tab) /
    sizeof(*tab);
    mergeSort(tab, 0, tabSize - 1);
    printTab(tab, tabSize);
    return 0;
  return 0;
}
```

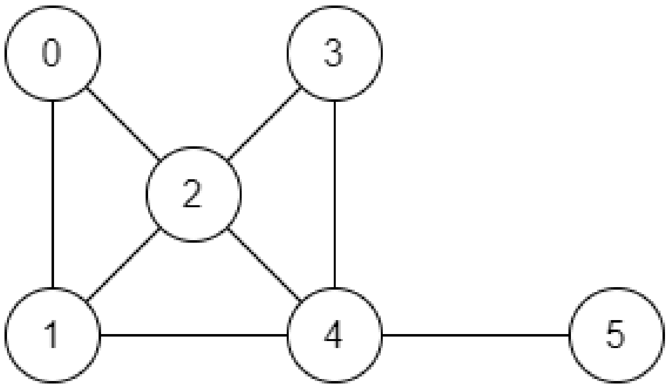
Listing 6. Implementacja algorytmu sortowania przez scalanie



Rys. 4.1. Kolejne kroki procesu sortowania za pomocą algorytmu *merge sort*

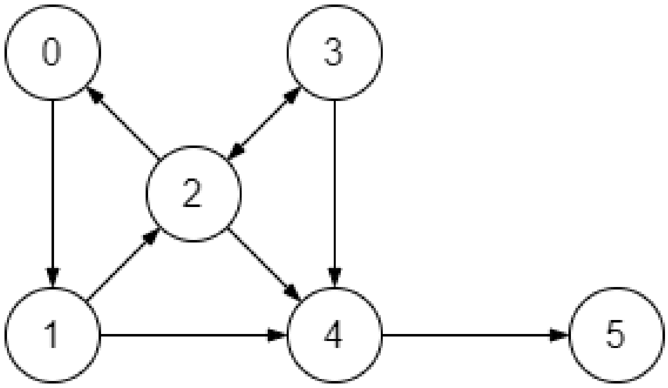
4.2. Grafy

Grafem nazywamy zbiór ***wierzchołków*** (*ang. vertices, nodes*) oraz połączeń między nimi – ***krawędzi*** (*edges*). Każda ***krawędź*** ma początek i koniec w jednym z ***wierzchołków***. Każdy ***wierzchołek*** może mieć od **0** do **n - 1** połączeń z innymi ***wierzchołkami***, gdzie **n** to liczba wszystkich ***wierzchołków*** w ***grafie***. Jeżeli każdy ***wierzchołek*** jest połączony z wszystkimi pozostałymi ***wierzchołkami grafu*** (istnieją wszystkie możliwe do utworzenia ***krawędzie***), to taki ***graf*** nosi nazwę ***pełnego*** (*ang. complete*). Przykład ***grafu*** o pięciu ***wierzchołkach*** i ośmiu ***krawędziach*** przedstawiono na rys. 4.2. Jeżeli kolejność ***wierzchołków*** tworzących ***krawędź*** nie ma znaczenia, to taki ***graf*** nazywa się ***prostym*** lub ***nieskierowanym***.



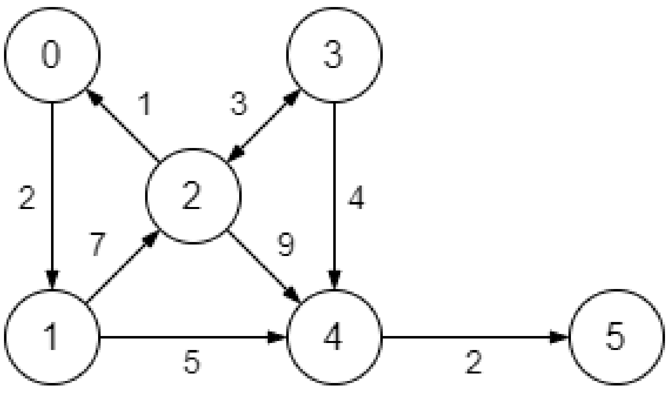
Rys. 4.2. Graf prosty (nieskierowany)

Jeżeli ***krawędzie grafu*** stanowią uporządkowane pary ***wierzchołków***, tj. ***krawędź*** (x, y) nie jest tożsama z ***krawędzią*** (y, x), to taki ***graf*** nazywa się ***skierowanym*** (rys. 4.3).



Rys. 4.3. Graf skierowany

Każda z ***krawędzi*** może dodatkowo mieć przypisaną pewną wartość zwaną ***wagą*** (*ang. weight*) lub ***kosztem*** (*ang. cost*). Umożliwia to zawarcie dodatkowych informacji na ***grafie***, gdy połączenia między poszczególnymi ***wierzchołkami*** mają różne znaczenie. Przykładowo, jeśli ***wierzchołki grafu*** stanowią pewne miasta, to ***krawędzie*** reprezentują połączenia drogowe między tymi miastami. Wówczas ***wagi*** przy ***krawędziach*** mogą określać długości tych połączeń (odległości między miastami). Przykład ***grafu skierowanego z wagami*** przedstawiono na rys. 4.4.



Rys. 4.4. Graf skierowany z wagami

Poza **rysunkiem**, typowymi sposobami reprezentacji ***grafu*** są **macierz** (*ang. adjacency matrix*) i **lista sąsiedztwa** (*ang. adjacency list*). **Macierz**

sąsiedztwa to macierz kwadratowa o rozmiarze **n x n** elementów, gdzie **n** oznacza liczbę wszystkich ***wierzchołków grafu***. Wartość na przecięciu *i-tego* wiersza i *j-tej* kolumny macierzy określa czy między ***wierzchołkami*** w_i oraz w_j jest (**1**) czy nie ma (**0**) połączenia. Innymi słowy, czy istnieje ***krawędź*** (i, j). ***Macierz sąsiedztwa*** dla ***grafu prostego*** z rys. 4.2 wynosi:

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

Główna przekątna macierzy zawiera same zera, ponieważ ***wierzchołki*** nie mają połączeń do samego siebie (brak *pętli własnych*). Warto zauważyć, że dla ***grafu prostego*** macierz jest symetryczna względem głównej przekątnej. Zasada ta nie jest spełniona dla ***grafu skierowanego*** (rys. 4.3):

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

Aby opisać ***graf z wagami***, na przecięciu *i-tego* wiersza i *j-tej* kolumny macierzy należy wpisać **0**, jeśli nie istnieje ***krawędź*** (i, j), albo wartość stanowiącą ***wagę*** dla tej ***krawędzi*** (rys. 4.4):

$$\begin{bmatrix} 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 5 & 0 \\ 1 & 0 & 0 & 3 & 9 & 0 \\ 0 & 0 & 3 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Macierz sąsiedztwa wymaga $\mathbf{n^2}$ jednostek pamięci do zapisania ***grafu***, w związku z czym jej zastosowanie sprowadza się do ***grafów*** o niewielkiej liczbie ***wierzchołków***. Zaletą takiego rozwiązania jest stały czas dodawania i usuwania ***krawędzi***.

Alternatywną formą reprezentacji ***grafu*** jest **lista sąsiedztwa**, gdzie dla każdego ***wierzchołka*** zapisywana jest lista połączonych z nim ***wierzchołków***. Takie rozwiązanie wymaga ilości pamięci proporcjonalnej do liczby ***krawędzi***, kosztem zwiększonej złożoności obliczeniowej przy dodawaniu i usuwaniu ***krawędzi grafu***. W związku z tym **lista sąsiedztwa** stosowana jest głównie do implementacji ***grafów*** o niewielkiej liczbie ***krawędzi***. **Lista sąsiedztwa** dla ***grafu prostego*** z rys. 4.2 wygląda następująco:

$$\begin{aligned} 0 &\longrightarrow 1, 2 \\ 1 &\longrightarrow 0, 2, 4 \\ 2 &\longrightarrow 0, 1, 3, 4 \\ 3 &\longrightarrow 2, 4 \\ 4 &\longrightarrow 1, 2, 3, 5 \\ 5 &\longrightarrow 4 \end{aligned}$$

W przypadku ***grafu skierowanego*** (rys. 4.3) lista musi zostać ograniczona:

$$0 \longrightarrow 1$$
$$1 \longrightarrow 2, 4$$
$$2 \longrightarrow 0, 3, 4$$
$$3 \longrightarrow 2, 4$$
$$4 \longrightarrow 5$$
$$5 \longrightarrow \emptyset$$

Możliwe jest również uwzględnienie ***wag*** dla poszczególnych ***krawędzi*** (rys. 4.4):

$$0 \longrightarrow 1(2)$$
$$1 \longrightarrow 2(7), 4(5)$$
$$2 \longrightarrow 0(1), 3(3), 4(9)$$
$$3 \longrightarrow 2(3), 4(4)$$
$$4 \longrightarrow 5(2)$$
$$5 \longrightarrow \emptyset$$

Do implementacji ***wierzchołka grafu*** można wykorzystać strukturę *Node*, przedstawioną na listingu 4. Dodatkowo należy zdefiniować struktury *Edge* oraz *Graph* realizujące odpowiednio ***krawędź*** między ***wierzchołkami*** o wartościach ***start*** i ***end*** oraz ***graf***, jako zbiór ***wierzchołków*** (***list***) o rozmiarze ***size***:

```
typedef struct Edge {
    int start, end;
} Edge_t;

typedef struct Graph {
    // Nodes count
    unsigned int size;
```

```
    // Nodes list
    Node_t ** list;
} Graph_t;
```

Poszczególne ***wierzchołki*** tworzone są z wykorzystaniem funkcji ***createNode()***, a usuwane przy użyciu funkcji ***deleteNode()*** (listing 7). Warto zwrócić uwagę, że dla zadanego ***wierzchołka*** usuwane są wszystkie ***wierzchołki*** należące do jego **listy sąsiedztwa**.

```
Node_t * createNode(int head, Node_t * tail) {
    Node_t * node = (Node_t *) malloc(sizeof(*node));
    node->head = head;
    node->tail = tail;
    return node;
}

void deleteNode(Node_t * node) {
    while (node) {
        Node_t * tmp = node->tail;
        free(node);
        node = tmp;
    }
}
```

Listing 7. Tworzenie i usuwanie wierzchołków grafu

Do tworzenia i usuwania struktury *Graph* służą funkcje ***createGraph()*** oraz ***deleteGraph()*** przedstawione na listingu 8. Funkcja ***createGraph()*** przyjmuje jako argumenty listę ***krawędzi edges***, rozmiar listy ***krawędzi edgesCount*** oraz liczbę ***wierzchołków grafu size***. W pierwszym kroku alokowana jest pamięć dla struktury *Graph* oraz wstępnie wyzerowana pamięć dla listy ***wierzchołków list***. Następnie, dla każdej ***krawędzi*** z listy, tworzony jest ***wierzchołek node***, do którego przypisywana jest wartość ***end krawędzi***. Do utworzonego ***wierzchołka*** dowiązywany jest adres ***wierzchołka*** aktualnie przechowywanego na liście pod indeksem ***start*** – tworzona jest **lista sąsiedztwa** dla konkretnego ***wierzchołka***. ***Wierzchołek node*** jest podstawiany na liście ***list*** pod indeksem ***start***. Funkcja ***deleteGraph()*** zwalnia pamięć zaalokowaną dynamicznie przez każde z wywołań funkcji *malloc()* i *calloc()* wewnątrz funkcji ***createGraph()***.

```
Graph_t * createGraph(Edge_t * edges, unsigned int
   edgesCount, unsigned int size) {
    Graph_t * graph = (Graph_t *) malloc(sizeof(*
   graph));
    graph->list = (Node_t **) calloc(size, sizeof(
   Node_t *));
    graph->size = size;

    if (edges) {
        for (unsigned int i = 0; i < edgesCount; ++i
   ) {
            int start = edges[i].start;
            int end = edges[i].end;

            Node_t * node = createNode(end, graph->
   list[start]);
            graph->list[start] = node;
        }
    }

    return graph;
}

```

```
void deleteGraph(Graph_t * graph) {
    if (graph) {
        for (unsigned int i = 0; i < graph->size; ++
   i) {
            deleteNode(graph->list[i]);
        }
        free(graph->list);
        free(graph);
    }
}
```

Listing 8. Tworzenie i usuwanie struktury grafu

Aby wypisać na ekranie **listę sąsiedztwa** można posłużyć się funkcją ***printAdjacencyList()***:

```
void printAdjacencyList(Graph_t * graph) {
    if (graph) {
        for (unsigned int i = 0; i < graph->size; ++
   i) {
            Node_t * node = graph->list[i];
            printf("%d -> ", i);
            while (node != NULL) {
                printf("%d, ", node->head);
                node = node->tail;
            }
            printf("\n");
        }
    }
}
```


Wykorzystując przedstawioną implementację ***grafu*** można odtworzyć strukturę z rys. 4.3:

```c
int main() {
    // Krawedzie (i, j) miedzy wierzcholkami o wartosciach i oraz j
    Edge_t edges[] = {
        {0, 1}, {1, 2}, {1, 4}, {2, 0}, {2, 3}, {2, 4}, {3, 2}, {3, 4}, {4, 5}
    };
    const int size = 6;
    const int edgesCount = sizeof(edges) / sizeof(Edge_t);
    Graph_t * graph = createGraph(edges, edgesCount, size);
    printAdjacencyList(graph);
    deleteGraph(graph);
    return 0;
}
```

Funkcja ***createGraph()*** z listingu 8. realizuje ***graf skierowany***. Aby utworzyć ***graf prosty*** należy odpowiednio zmodyfikować funkcję ***createGraph()***, dodając symetryczne **listy sąsiedztwa** dla ***węzłów*** z drugiego końca ***krawędzi***:

```c
Graph_t * createGraph(Edge_t * edges, unsigned int edgesCount, unsigned int size) {
    Graph_t * graph = (Graph_t *) malloc(sizeof(*graph));
    graph->list = (Node_t **) calloc(size, sizeof(Node_t *));
```

```
    graph->size = size;

    if (edges) {
        for (unsigned int i = 0; i < edgesCount; ++i) {
            int start = edges[i].start;
            int end = edges[i].end;

            Node_t * node = createNode(end, graph->list[start]);
            graph->list[start] = node;

            // Undirected graph
            node = createNode(start, graph->list[end]);
            graph->list[end] = node;
        }
    }
    return graph;
}
```

Na listingu 9. przedstawiono zmodyfikowany kod umożliwiający utworzenie ***skierowanego grafu z wagami***. ***Wagi*** poszczególnych krawędzi przechowywane są pod zmienną ***weight***.

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

typedef struct Node {
    int head, weight;
    struct Node * tail;
} Node_t;
```

```c

typedef struct Edge {
    int start, end, weight;
} Edge_t;

typedef struct Graph {
    // Nodes count
    unsigned int size;
    // Adjacency list
    Node_t ** list;
} Graph_t;

Node_t * createNode(int head, int weight, Node_t * tail) {
    Node_t * node = (Node_t *) malloc(sizeof(*node));
    node->head = head;
    node->weight = weight;
    node->tail = tail;
    return node;
}

Graph_t * createGraph(Edge_t * edges, unsigned int edgesCount, unsigned int size) {
    Graph_t * graph = (Graph_t *) malloc(sizeof(*graph));
    graph->list = (Node_t **) calloc(size, sizeof(Node_t *));
    graph->size = size;

    if (edges) {
```



```
        for (unsigned int i = 0; i < edgesCount; ++i) {
            int start = edges[i].start;
            int end = edges[i].end;
            int weight = edges[i].weight;

            Node_t * node = createNode(end, weight, graph->list[start]);
            graph->list[start] = node;
        }
    }

    return graph;
}

void deleteNode(Node_t * node) {
    while (node) {
        Node_t * tmp = node->tail;
        free(node);
        node = tmp;
    }
}

void deleteGraph(Graph_t * graph) {
    if (graph) {
        for (unsigned int i = 0; i < graph->size; ++i) {
            deleteNode(graph->list[i]);
        }
        free(graph->list);
        free(graph);
```

```c
    }
}

void printAdjacencyList(Graph_t * graph) {
    if (graph) {
        for (unsigned int i = 0; i < graph->size; ++i) {
            Node_t * node = graph->list[i];
            printf("%d -> ", i);
            while (node != NULL) {
                printf("%d (%d), ", node->head, node->weight);
                node = node->tail;
            }
            printf("\n");
        }
    }
}

int main() {
    // Krawedzie (i, j) miedzy wierzcholkami o wartosciach i oraz j, wraz z wagami
    Edge_t edges[] = {
        {0, 1, 2}, {1, 2, 7}, {1, 4, 5}, {2, 0, 1}, {2, 3, 3}, {2, 4, 9}, {3, 2, 3}, {3, 4, 4}, {4, 5, 2}
    };
    const int size = 6;
    const int edgesCount = sizeof(edges) / sizeof(Edge_t);
```



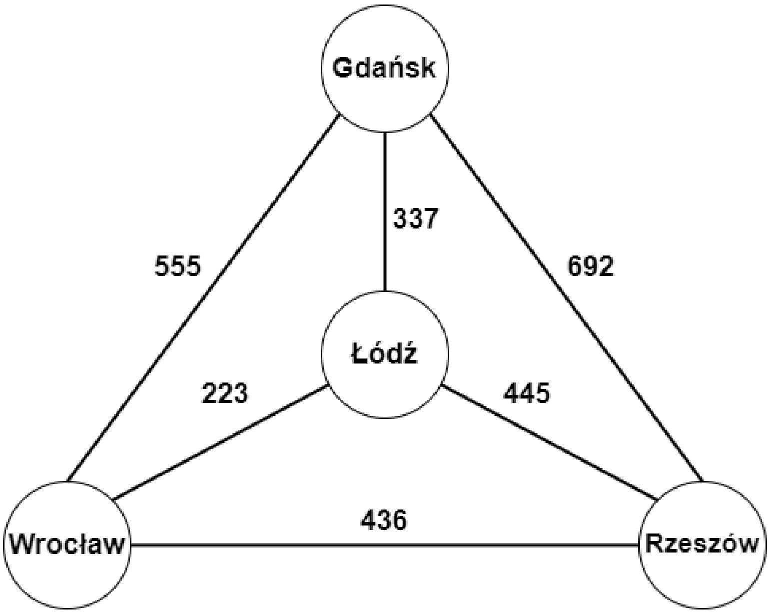
```c
    Graph_t * graph = createGraph(edges, edgesCount,
    size);
    printAdjacencyList(graph);
    deleteGraph(graph);
    return 0;
}
```

Listing 9. Implementacja grafu skierowanego z wagami

Graf może być zastosowany do opisu tzw. ***problemu komiwojażera*** (*ang. traveling salesman problem, TSP*). Zagadnienie to można streścić następująco: dla danych n miast znaleźć najkrótszą drogę łączącą wszystkie miasta tak, aby każde miasto odwiedzić dokładnie jeden raz i zakończyć podróż w mieście początkowym. Liczba możliwych kombinacji wynosi $\frac{(n-1)!}{2}$. Na rys. 4.5. przedstawiono przykładowy problem dla czterech miast. Każde z miast znajduje się w jednym z wierzchołków ***grafu***, natomiast ***wagi*** poszczególnych ***krawędzi*** reprezentują odległości między miastami mierzone w kilometrach. Jest to przykład ***grafu pełnego nieskierowanego***. Ponieważ ***graf*** jest pełny, a punkt początkowy jest jednocześnie punktem końcowym trasy, to wybór miasta początkowego jest dowolny. Niech punktem początkowym będzie Wrocław, wówczas możliwe kombinacje (z pominięciem tras symetrycznych) to:

- Wrocław → Gdańsk → Łódź → Rzeszów → Wrocław (1773 km)
- Wrocław → Łódź → Rzeszów → Gdańsk → Wrocław (1915 km)
- Wrocław → Rzeszów → Gdańsk → Łódź → Wrocław (1688 km)

Problem znacznie komplikuje się wraz ze zwiększeniem liczby miast, które musi odwiedzić komiwojażer.



Rys. 4.5. Graf jako reprezentacja problemu komiwojażera

Problem komiwojażera można rozwiązać wykorzystując algorytm przedstawiony na listingu 10.

```
// Pomocnicza funkcja do zmiany kolejnosci elementow
void swap(int * a, int * b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

// Funkcja obliczajaca dlugosc trasy
int calculateRoute(Graph_t * graph, const int *
   nodesOrder) {
    int route = 0;
    // Przygotuj tablice odleglosci miedzy miastami
    int * distances = (int *) calloc(graph->size,
   sizeof(int));
```

```c
    printf("Route: %d", nodesOrder[0]);

    // Iteruj po wszystkich miastach
    for (unsigned int i = 0; i < graph->size; ++i) {
        // Pobierz liste sasiedztwa dla i-tego miasta
        Node_t * node = graph->list[nodesOrder[i]];
        // Zapisz liste sasiedztwa do tablicy odleglosci
        while (node) {
            distances[node->head] = node->weight;
            node = node->tail;
        }
        // Pobierz identyfikator nastepnego miasta w trasie
        int nodeIndex = nodesOrder[i + 1];
        // Odczytaj odleglosc miedzy kolejnymi miastami
        int distance = distances[nodeIndex];
        printf(" -> %d ", nodeIndex);
        // Dodaj odczytana odleglosc do wynikowej dlugosci trasy
        route += distance;
    }

    printf(" has length equal to %d km\n", route);
    // Zwolnij pamiec zaalokowana dynamicznie
    free(distances);
    return route;
}

```

```
// Funkcja wykonujaca permutacje kolejno odwiedzanych miast
int permute(Graph_t * graph, int * nodesOrder, int lo, int hi)
{
    // Zainicjalizuj tylko raz
    static int shortestRoute = INT_MAX;
    if (lo == hi) {
        // Skopiuj miasto poczatkowe na ostatnia pozycje w trasie
        nodesOrder[graph->size] = nodesOrder[0];
        // Oblicz dlugosc trasy
        int route = calculateRoute(graph, nodesOrder);
        // Zapisz wartosc, jesli obliczona trasa jest krotsza niz poprzednia
        if (route < shortestRoute)
            shortestRoute = route;
    } else {
        for (int i = lo; i <= hi; ++i) {
            // Zamien miejscami wierzcholki
            swap(nodesOrder + lo, nodesOrder + i);
            // Wywolanie rekurencyjne
            permute(graph, nodesOrder, lo + 1, hi);
            // Przywroc stan poczatkowy
            swap(nodesOrder + lo, nodesOrder + i);
        }
    }
    return shortestRoute;
}

```

```
// Funkcja wyznaczajaca najkrotsza trase komiwojazera
int findShortestRoute(Graph_t * graph) {
    // Przygotuj tablice zawierajaca identyfikatory kolejno odwiedzanych miast (+1 na powrot do miasta poczatkowego)
    int * nodesOrder = (int *) calloc(graph->size + 1, sizeof(int));
    for (unsigned int i = 0; i < graph->size; ++i)
        nodesOrder[i] = i;
    // Oblicz najkrotsza trase sposrod wyznaczonych permutacji
    // Miasto o identyfikatorze 0 zostalo arbitralnie wybrane jako punkt poczatkowy
    int shortestRoute = permute(graph, nodesOrder, 1, graph->size - 1);
    // Zwolnij pamiec zaalokowana dynamicznie
    free(nodesOrder);
    return shortestRoute;
}

int main() {
    /**
     * 0 - Wroclaw
     * 1 - Gdansk
     * 2 - Rzeszow
     * 3 - Lodz
     */
    Edge_t edges[] = {
        {0, 1, 555}, {0, 2, 436}, {0, 3, 223}, {1, 2, 692}, {1, 3, 337}, {2, 3, 445}
```

```
    };
    const int size = 4;
    const int edgesCount = sizeof(edges) / sizeof(
  Edge_t);
    Graph_t * graph = createGraph(edges, edgesCount,
   size);
    printf("Shortest route equals %d km\n",
  findShortestRoute(graph));
    deleteGraph(graph);
    return 0;
}
```

Listing 10. Rozwiązanie problemu komiwojażera z wykorzystaniem implementacji grafu