Notatki do kolokwium

Algorytmy sortowania

- Algorytm stabilny - nie zamienia równych elementów
- Algorytm niestabilny - zamienia równe elementy

- Sortowanie bąbelkowe (bubble sort)

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

- Sortowanie przez wstawianie (insertion sort)

```
void insertionSort(int *arr, int n) {
// Początkowo zakładamy, że pierwszy element jest posortowany
    for (int i = 1; i < n; i++) {
        // Wybieramy następny element do wstawienia
        int key = arr[i];

        // Wstawiamy element w odpowiednie miejsce w posortowanej części tablic
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            // Przesuwamy elementy większe od klucza o jeden indeks w prawo
            arr[j + 1] = arr[j];
            j--;
        }

        // Wstawiamy klucz w odpowiednie miejsce
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = {5, 2, 8, 3, 1, 6, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, n - 1);

    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

- Sortowanie przez scalanie (merge sort)

```
// Funkcja merge łącząca dwie posortowane tablice w jedną posortowaną tablicę
void merge(int* arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Tworzymy dwie tymczasowe tablice
    int leftArr[n1];
    int rightArr[n2];

    // Kopiujemy dane do tymczasowych tablic
    for (int i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArr[j] = arr[mid + 1 + j];
    }

    // Łączymy dwie posortowane tablice w jedną posortowaną tablicę
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    // Kopiujemy pozostałe elementy z lewej tablicy
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    // Kopiujemy pozostałe elementy z prawej tablicy
    while (j < n2) {
```

```
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

// Funkcja mergeSort rekurencyjnie dzieląca tablicę na mniejsze części i łącząca je w pos
void mergeSort(int *arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Dzielimy tablicę na dwie mniejsze części
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Łączymy dwie posortowane części w jedną posortowaną tablicę
        merge(arr, left, mid, right);
    }
}
```

- Sortowanie szybkie (quick sort)

```
// Funkcja zamieniająca dwa elementy w tablicy
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Funkcja partition dzieląca tablicę na dwie części wokół pivotu
int partition(int arr[], int left, int right) {
    int pivot = arr[right]; // Wybieramy pivot jako ostatni element tablicy
    int i = left - 1;

    for (int j = left; j < right; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[right]);
    return i + 1;
}

// Funkcja quickSort rekurencyjnie sortująca tablicę
void quickSort(int arr[], int left, int right) {
    if (left < right) {
        int pivot = partition(arr, left, right);

        // Rekurencyjnie sortujemy lewą i prawą część tablicy
        quickSort(arr, left, pivot - 1);
        quickSort(arr, pivot + 1, right);
    }
}

int main() {
    int arr[] = {5, 2, 8, 3, 1, 6, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);
```

```
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Dynamiczna tablica

```
#include <iostream>

using namespace std;

int *extendArray(int *arr, int *rozmiar){
    cout << "Rozszerzam tablice; nowy rozmiar: " << (*rozmiar)*2 << endl;

    int *new_arr = new int[*rozmiar*2];

    for(int i = 0; i < *rozmiar; i++){
        new_arr[i] = arr[i];
    }

    *rozmiar *= 2;

    delete[] arr;

    return new_arr;
}

int main() {
    int rozmiar = 10;
    int *arr = new int[rozmiar];

    for(int i=0; i<100; i++){
        if(i >= rozmiar){
            arr = extendArray(arr, &rozmiar);
        }

        arr[i] = i;
    }

    for(int i=0; i<100; i++){
        cout << arr[i] << " ";
    }
}
```

Drzewo binarne

```
#include <iostream>

using namespace std;

struct Node{
    Node *left = nullptr;
    Node *right = nullptr;

    int data = 0;

    Node(int data){
        this->data = data;
    }
};

Node *insert(Node *root, int data){
    if(root == nullptr){
        root = new Node(data);
    }else{
        if(data < root->data){
            root->left = insert(root->left, data);
        }else if(data > root->data){
            root->right = insert(root->right, data);
        }
        //trzeci warunek - jeśli równe
    }

    return root;
}

void printTree(Node *root){
    if(root == nullptr){
        return;
    }

    printTree(root->left);
    cout << root->data << " ";
    printTree(root->right);
}
```

```
int main(){
    Node *root = nullptr;

    int data[] = {10, 5, 15, 3, 7, 12, 18};
    int n = sizeof(data) / sizeof(data[0]);

    for(int i = 0; i < n; i++){
        root = insert(root, data[i]);
    }

    printTree(root);

    return 0;
}
```