

# Algorytmy i struktury danych – laboratorium 3.

Zliczanie, wyszukiwanie, sortowanie, drzewa, drzewa BST

1. Napisz program, który policzy, ile razy występuje każdy znak w pliku tekstowym podanym jako argument wywołania. W zaproponowanym programie nie można użyć w części algorytmicznej wyrażenia warunkowego if. Do analizy zawartości pliku możesz użyć zaadaptowanego fragmentu następującego kodu:

```
var sciezka = @"C:\Users\Student\Desktop\plik.txt";

string zawartosc = File.ReadAllText(sciezka, Encoding.UTF8);
Console.WriteLine(zawartosc); // wyświetlenie wczytanej zawartości

foreach(char c in zawartosc)
{
    Console.WriteLine(c); // wyświetlenie znaków w oddzielnych
liniach
}
```

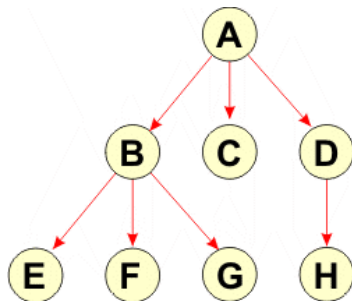
Zauważ, że powyższy kod nie zawiera elementów sprawdzających, czy dany plik istnieje.

2. Używając napisanej przez siebie funkcji min() i max() znajdź znaki najczęściej pojawiające się we wczytanym wcześniej pliku – wyświetl te znaki.
3. Jako rozwinięcie powyższego programu posortuj uzyskaną tablicę dwoma napisanymi przez siebie algorytmami sortującymi - jednym algorytmem prostym (wstawianie, wybieranie, bąbelkowe) i algorytmem quicksort. W tym celu napisz funkcje sortujące tablicę. Zwróć uwagę, że celem zadania jest uzyskanie tablicy zawierającej informacje o tym jakie znaki występują najczęściej w pliku. Informacja ta powinna zostać wyświetlona następnie w sposób następujący: np. znak a -122, znak e-100, znak x-40, ..... Dostosuj algorytm sortujący tak, aby zachowywał informację o tym jakiej literze dotyczy określona liczba wystąpień.
4. Wyszukiwanie liniowe (ang. linear search), zwane również sekwencyjnym (ang. sequential search) polega na przeglądaniu kolejnych elementów tablicy TAB. Jeśli przeglądany element posiada odpowiednie własności (np. jest liczbą o poszukiwanej wartości), to zwracamy jego pozycję w zbiorze i kończymy. W przeciwnym razie kontynuujemy poszukiwania aż do przejścia wszystkich pozostałych elementów zbioru Z. W przypadku pesymistycznym, gdy poszukiwanego elementu nie ma w zbiorze lub też znajduje się on na samym końcu zbioru, algorytm musi wykonać przynajmniej n obrotów pętli sprawdzającej poszczególne elementy. Wynika z tego, iż pesymistyczna klasa złożoności obliczeniowej jest równa  $O(n)$ , czyli jest liniowa - stąd pochodzi nazwa metody wyszukiwania. Często chcemy znaleźć wszystkie wystąpienia w zbiorze poszukiwanej wartości elementu. W takim przypadku algorytm powinien stworzyć tablicę indeksów, do której będzie zapisywał znalezione wartości indeksów. Napisz algorytm pozwalający odnaleźć w utworzonej tablicy znak występujący w pliku tekstowym określoną liczbę razy. W przypadku gdy kilka znaków pojawia się również tyle samo razy powinny one również zostać wyświetlone. Stworzoną funkcję nazwij `LinearSearch`
5. Wyszukiwanie binarne jest algorytmem opierającym się na metodzie dziel i zwyciężaj, który w czasie logarytmicznym stwierdza, czy szukany element znajduje się w uporządkowanej tablicy i jeśli się znajduje, podaje jego indeks. Przeszukiwanie binarne, zwane też połówkowym, jest algorytmem sprawdzającym czy uporządkowany rosnąco ciąg liczbowy zawiera podaną liczbę x. Algorytm przeszukiwania binarnego korzysta z uporządkowania ciągu liczbowego dzieląc go każdorazowo na połowy. Poszukiwaną liczbę x porównuje się z liczbą środkową ciągu i w zależności od wyniku tego porównania przeszukiwana jest albo lewa, albo prawa część tablicy. Obszar poszukiwania wyznaczają

dwa indeksy: left i right oznaczające odpowiednio: początek i koniec przeszukiwanego fragmentu tablicy -rozpoczynając przeszukiwanie zmiennym tym przypisujemy wartości left = 0 i right = n-1. Sprawdzamy, czy left<=right. Jeśli tak, to: obliczamy indeks elementu środkowego mid = (left + right)/2.. Wówczas testowane jest, czy element zapisany pod indeksem mid jest tym poszukiwanym — jeśli tak, to algorytm w tym miejscu kończy działanie. W przeciwnym razie przedział jest zawężany - dzięki uporządkowaniu danych wiadomo, że albo poszukiwany element może znajdować się gdzieś przed indeksem mid albo za nim. Innymi słowy wybór ogranicza się do przedziału [left , mid - 1], gdy poszukiwany element jest mniejszy od zapisanego pod indeksem mid, albo [mid + 1 , right] w przeciwnym razie. Algorytm kończy się niepowodzeniem, jeśli przedział będzie pusty, tzn. left > right (lewy koniec przedziału "znajdzie się" za prawym końcem). Zostanie wtedy zwrócona wartość -1. Wykonaj zadanie z poprzedniego przykładu przy użyciu wyszukiwania binarnego. Funkcję nazwij BinarySearch.

6. Wykonując 10000 razy sortowanie generowanej każdorazowo tablicy złożonej z 10000 elementów będących liczbami całkowitymi z przedziału 1-100000 porównaj czasy wykonywania napisanych przez Ciebie algorytmów.
7. **Drzewo** (ang. tree) jest strukturą danych zbudowaną z elementów, które nazywamy **węzłami** (ang. node ). Dane przechowuje się w węzłach drzewa. Węzły są ze sobą powiązane w sposób hierarchiczny za pomocą **krawędzi** (ang. edge ), które zwykle przedstawia się za pomocą strzałki określającej hierarchię. Pierwszy węzeł drzewa nazywa się **korzeniem** (ang. root node ). Od niego "wyrastają" pozostałe węzły, które będziemy nazywać **synami** (ang. child node ). Synowie są węzłami podrzędnymi w strukturze hierarchicznej. Synowie tego samego ojca są nazywani **braćmi** (ang. sibling node ). Węzeł nadrzędny w stosunku do syna nazwiemy **ojcem** (ang. parent node ). Ojcowie są węzłami nadrzędnymi w strukturze hierarchicznej. Jeśli węzeł nie posiada synów, to nazywa się **liściem** ( ang. leaf node ), w przeciwnym razie nazywa się **węzłem wewnętrznym** (ang. internal node, inner node, branch node ).

#### Struktura drzewa

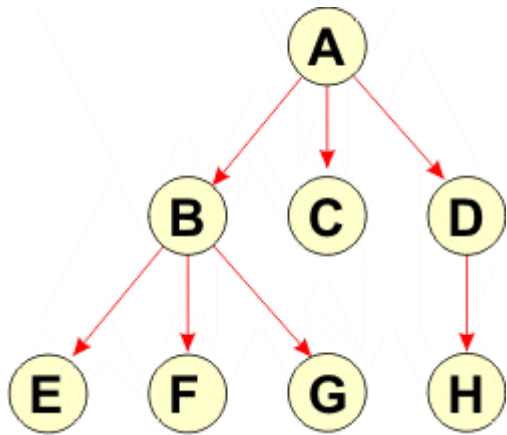


A...H – węzły drzewa  
 strzałki – krawędzie. Zwrot określa kierunek hierarchii rodzic → dziecko.  
 A – korzeń drzewa  
 B, C, D – bracia, synowie węzła A, który jest dla nich ojcem  
 E, F, G – bracia, synowie węzła B, który jest dla nich ojcem  
 H – syn węzła D, który jest dla niego ojcem  
 A, B, D – węzły wewnętrzne  
 C, E, F, G, H – liście

Za wyjątkiem korzenia wszystkie pozostałe węzły w drzewie posiadają swojego ojca. W normalnym drzewie liczba synów dla dowolnego węzła nie jest ograniczona. Istnieje jednakże bardzo ważna klasa drzew, w których dany węzeł może posiadać co najwyżej dwóch synów. Noszą one nazwę **drzew binarnych** ( ang. binary tree ).

Ciąg węzłów połączonych krawędziami nazwiemy **ścieżką** (ang. path ). Od korzenia do określonego węzła w drzewie istnieje zawsze dokładnie jedna ścieżka prosta, tzn. taka, iż zawarte w niej węzły pojawiają się tylko jeden raz. **Długością ścieżki** (ang. path length) nazwiemy liczbę krawędzi łączących węzły w ścieżce. Dla naszego drzewa mamy następujące ścieżki proste od korzenia do kolejnych węzłów:

### Struktura drzewa

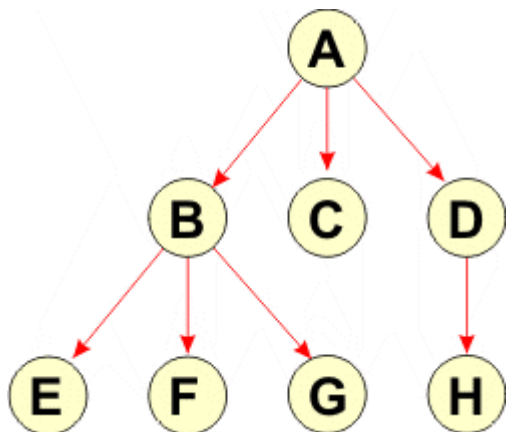


### Ścieżki

ścieżka od A do A: długość 0: A  
ścieżka od A do B: długość 1: A→B  
ścieżka od A do C: długość 1: A→C  
ścieżka od A do D: długość 1: A→D  
ścieżka od A do E: długość 2: A→B→E  
ścieżka od A do F: długość 2: A→B→F  
ścieżka od A do G: długość 2: A→B→G  
ścieżka od A do H: długość 2: A→D→H

Długość ścieżki prostej od korzenia do danego węzła nazywa się **poziomem węzła** (ang. node level ). Korzeń drzewa ma zawsze poziom 0. W naszym drzewie węzły B, C i D mają poziom 1, a E, F, G i H mają poziom 2. **Wysokość drzewa** (ang. tree height) jest równa największemu poziomowi węzłów (lub najdłuższej ścieżce rozpoczynającej się w korzeniu ). Dla naszego drzewa wysokość jest równa 2. **Wysokość węzła** (ang. node height ), to długość najdłuższej ścieżki od tego węzła do liścia. Dla korzenia wysokość węzła jest równa wysokości drzewa:

### Struktura drzewa

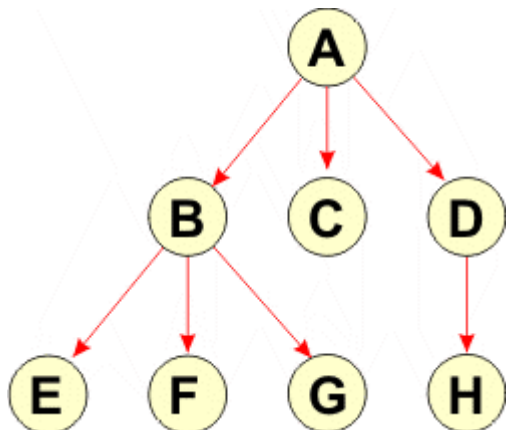


### Wysokości węzłów

węzeł A: wysokość = 2  
węzeł B: wysokość = 1  
węzeł C: wysokość = 0  
węzeł D: wysokość = 1  
węzeł E: wysokość = 0  
węzeł F: wysokość = 0  
węzeł G: wysokość = 0  
węzeł H: wysokość = 0

**Poziom drzewa** (ang. tree level, the level of a tree) dla danego węzła to długość ścieżki prostej od korzenia do danego węzła.

### Struktura drzewa



### Poziomy

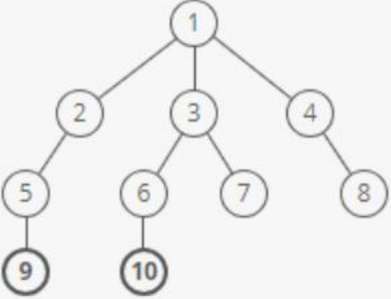
Poziom 0: A  
Poziom 1: B, C, D  
Poziom 2: E, F, G, H

Liczba krawędzi powiązanych z danym węzłem nosi nazwę **stopnia węzła** (ang. node degree ). Krawędzie drzewa są **krawędziami skierowanymi** (ang. directed edge) i oznaczamy je za pomocą strzałek. Kierunek strzałki jednoznacznie określa pozycję w hierarchii – strzałka wychodzi od ojca i kończy się na synu. Z tego powodu stopień węzła rozбивa się na dwa stopnie:

**stopień wejściowy** ( ang. node in-degree) – liczba krawędzi wchodzących do węzła, dla drzewa nigdy nie przekracza 1, a jest równy 0 tylko dla korzenia,

**stopień wyjściowy** (ang. node out-degree) – liczba krawędzi wychodzących z węzła, określa liczbę synów.

Poniżej znajduje się rysunek pewnego drzewa i jego implementacja w języku C# wraz z implementacją algorytmu BFS – przechodzenia drzewa wszerz (breadth-first search). Algorytm ten pozwala znaleźć najkrótszą ścieżkę pomiędzy węzłem startowym a dowolnym innym węzłem.

 <pre> graph TD     1((1)) --&gt; 2((2))     1 --&gt; 3((3))     2 --&gt; 5((5))     3 --&gt; 6((6))     3 --&gt; 7((7))     4((4)) --&gt; 8((8))     5 --&gt; 9((9))     6 --&gt; 10((10)) </pre>	<pre> BinaryTree binaryTree = new();  binaryTree.Add(1); binaryTree.Add(2); binaryTree.Add(7); binaryTree.Add(3); binaryTree.Add(10); binaryTree.Add(5); binaryTree.Add(8);  Node node = binaryTree.Find(5); int depth = binaryTree.GetTreeDepth();  Console.WriteLine("Przechodzenie PreOrder:"); binaryTree.TraversePreOrder(binaryTree.Root); Console.WriteLine();  Console.WriteLine("Przechodzenie InOrder:"); binaryTree.TraverseInOrder(binaryTree.Root); Console.WriteLine();  Console.WriteLine("Przechodzenie PostOrder:"); binaryTree.TraversePostOrder(binaryTree.Root); ; Console.WriteLine();  binaryTree.Remove(7); binaryTree.Remove(8);  Console.WriteLine("PreOrder Traversal After Removing Operation:"); binaryTree.TraversePreOrder(binaryTree.Root); Console.WriteLine();  Console.ReadLine();  public class Node {     public Node LeftNode { get; set; }     public Node RightNode { get; set; }     public int Data { get; set; } }  public class BinaryTree { </pre>
--	--

```

public Node Root { get; set; }

public bool Add(int value)
{
    Node before = null, after =
this.Root;

    while (after != null)
    {
        before = after;
        if (value < after.Data) //Is new
node in left tree?
            after = after.LeftNode;
        else if (value > after.Data) //Is
new node in right tree?
            after = after.RightNode;
        else
        {
            //Exist same value
            return false;
        }
    }

    Node newNode = new Node();
    newNode.Data = value;

    if (this.Root == null)//Tree ise
empty
        this.Root = newNode;
    else
    {
        if (value < before.Data)
            before.LeftNode = newNode;
        else
            before.RightNode = newNode;
    }

    return true;
}

public Node Find(int value)
{
    return this.Find(value, this.Root);
}

public void Remove(int value)
{
    this.Root = Remove(this.Root, value);
}

private Node Remove(Node parent, int key)
{
    if (parent == null) return parent;

    if (key < parent.Data)
parent.LeftNode = Remove(parent.LeftNode,
key);
    else if (key > parent.Data)
        parent.RightNode =
Remove(parent.RightNode, key);

    // if value is same as parent's
value, then this is the node to be deleted

```

```

else
{
    // node with only one child or no
child
    if (parent.LeftNode == null)
        return parent.RightNode;
    else if (parent.RightNode ==
null)
        return parent.LeftNode;

    // node with two children: Get
the inorder successor (smallest in the right
subtree)
    parent.Data =
MinValue(parent.RightNode);

    // Delete the inorder successor
    parent.RightNode =
Remove(parent.RightNode, parent.Data);
}

    return parent;
}

private int MinValue(Node node)
{
    int minv = node.Data;

    while (node.LeftNode != null)
    {
        minv = node.LeftNode.Data;
        node = node.LeftNode;
    }

    return minv;
}

private Node Find(int value, Node parent)
{
    if (parent != null)
    {
        if (value == parent.Data) return
parent;
        if (value < parent.Data)
            return Find(value,
parent.LeftNode);
        else
            return Find(value,
parent.RightNode);
    }

    return null;
}

public int GetTreeDepth()
{
    return this.GetTreeDepth(this.Root);
}

private int GetTreeDepth(Node parent)
{

```

	<pre>         return parent == null ? 0 :         Math.Max(GetTreeDepth(parent.LeftNode),         GetTreeDepth(parent.RightNode)) + 1;     }      public void TraversePreOrder(Node parent)     {         if (parent != null)         {             Console.Write(parent.Data + " ");             TraversePreOrder(parent.LeftNode);             TraversePreOrder(parent.RightNode);         }     }      public void TraverseInOrder(Node parent)     {         if (parent != null)         {             TraverseInOrder(parent.LeftNode);             Console.Write(parent.Data + " ");             TraverseInOrder(parent.RightNode);         }     }      public void TraversePostOrder(Node parent)     {         if (parent != null)         {             TraversePostOrder(parent.LeftNode);             TraversePostOrder(parent.RightNode);             Console.Write(parent.Data + " ");         }     } } </pre>
--	---

### Zrealizuj funkcje pozwalające:

Zliczyć wszystkie liście

Określić wysokość drzewa

Stwórz funkcję, która znajdzie ścieżkę w drzewie binarnym od korzenia do liścia, mającą największą sumę wartości.

Stwórz algorytm, który znajdzie wszystkie ścieżki w drzewie binarnym od korzenia do liścia