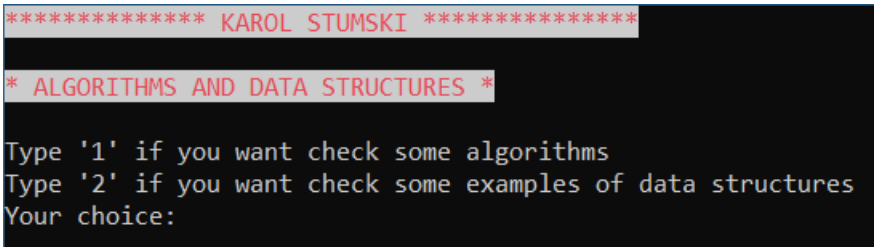
Spis treści

Algorithms and data structures

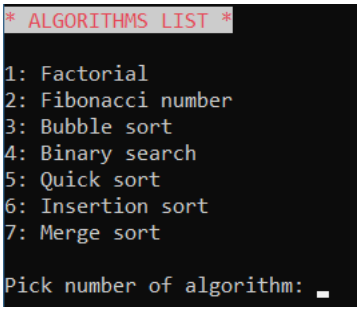
KAROL STUMSKI – I rok informatyka (dzienne)

Instrukcja poruszania się po programie

Aby przejść do poszczególnych komponentów aplikacji wpisujemy odpowiednią cyfrę związaną z naszym wyborem i naciskamy klawisz Enter.



Po wybraniu interesującej nas listy algorytmów/struktur danych postępujemy tak samo jak poprzednio – wybieramy cyfrę, a następnie naciskamy Enter.



Po wykonaniu każdego algorytmu/struktury danych wyświetla się menu poruszania się po programie.

```
If you want try again use this algorithm, type 'R'.
If you want back to algorithms list, type 'L'.
If you want back to main menu, type 'Q'
Type:
```

Jeśli chcemy ponownie użyć dany algorytm/strukturę danych wpisujemy literkę R oraz naciskamy Enter. Podobnie w przypadku powrotu do listy algorytmów lub struktur danych (literka L) oraz w przypadku powrotu do początku programu (literka Q).

Algorytmy

```
* ALGORITHMS LIST *

1: Factorial
2: Fibonacci number
3: Bubble sort
4: Binary search
5: Quick sort
6: Insertion sort
7: Merge sort

Pick number of algorithm:
```

Sortowanie bąbelkowe (Bubble sort)

Jest to prosta metoda sortowania o złożoności czasowej $O(n^2)$ i pamięciowej $O(1)$. Polega na porównywaniu dwóch kolejnych elementów i zamianie ich kolejności, jeżeli zaburza ona porządek, w jakim się sortuje tablicę. Sortowanie kończy się, gdy podczas kolejnego przejścia nie dokonano żadnej zmiany.

```
for (int i = 0; i < bubblesize.Length; i++) //i dete
{
    for (int j = bubblesize.Length - 1; j > i; j--)
    {
        if (bubblesize[j - 1] > bubblesize[j]) //if
        {

            var temp = bubblesize[j];
            bubblesize[j] = bubblesize[j - 1]; //swa
            bubblesize[j - 1] = temp;
        }
    }
}
```

W głównej pętli zmienna „i” służy jako n-liczba kroków sortowania. Następnie zmienna „j” w tzw. „pętli w pętli” oznacza (n-1) liczbę kroków porównań oraz indeks dla każdego kroku, które należy zbadać. Następnie sprawdzamy warunek – jeśli pierwszy element jest większy od drugiego to następuje zamiana.

Sortowanie szybkie (Quick sort)

Jeden z popularnych algorytmów sortowania działających na zasadzie „dziel i zwyciężaj”. Sortowanie szybkie (z ang. quick sort) zostało wynalezione w 1962 przez C.A.R. Hoare’a. Algorytm sortowania szybkiego jest wydajny: jego średnia złożoność obliczeniowa jest rzędu O(n log n). Ze względu na szybkość i prostotę implementacji jest powszechnie używany. Jego implementacje znajdują się w bibliotekach standardowych wielu środowisk programowania.

```
private static int Sorting(int[] quicksize, int left, int right)
{
    int pivot = quicksize[left]; //pivot is chosen element in sort
    while (true)
    {
        while (quicksize[left] < pivot) //until pivot is bigger th
        {
            left++;
        }
        while (quicksize[right] > pivot) //until right side is big
        {
            right--;
        }
        if (left < right)
        {
            int temp = quicksize[left];
            quicksize[left] = quicksize[right]; //swap
            quicksize[right] = temp;

            if (quicksize[left] == quicksize[right]) //if same num
                left++;
        }
        else
        {
            return right; //if conditions are not met then return
        }
    }
}
```

Najpierw omówimy metodę sortującą. Wejściem jest rozmiar tablicy do posortowania, lewa strona oraz prawa. Inicjujemy pivot – czyli element rozdzielający. Następnie ustawiamy elementy nie większe na lewo tej wartości, natomiast nie mniejsze na prawo za pomocą dwóch pętli while. Następnie każda z tych „podtablic” dzieli się tak samo i tak powstaje tablica posortowana.

```
private static void Quick_Sort(int[] quicksize, int left, int right) //pick pivot
{
    if (left < right)
    {
        int pivot = Sorting(quicksize, left, right);

        if (pivot > 1)
        {
            Quick_Sort(quicksize, left, right: pivot - 1);
        }

        if (pivot + 1 < right)
        {
            Quick_Sort(quicksize, left: pivot + 1, right);
        }
    }
}
```

Tutaj wybieramy za pomocą warunków co ma być naszym pivotem.

Sortowanie przez wstawianie (Insertion sort)

Jeden z najprostszych algorytmów sortowania, którego zasada działania odzwierciedla sposób w jaki ludzie ustawiają karty – kolejne elementy wejściowe są ustawiane na odpowiednie miejsca docelowe. Jest efektywny dla niewielkiej liczby elementów, jego złożoność wynosi $O(n^2)$. Pomimo tego, że jest znacznie mniej wydajny od algorytmów takich jak quicksort czy heapsort.

```
int val, flag;
for (int i = 1; i < insertionsortamount; i++)
{
    val = array[i];
    flag = 0;
    for (int j = i - 1; j >= 0 && flag != 1;)
    {
        if (val < array[j])
        {
            array[j + 1] = array[j];
            j--;
            array[j + 1] = val;
        }
        else flag = 1; //it's go until all is
    }
}
```

Pętla działa dopóki „i” jest mniejsze od rozmiaru tablicy, którą wprowadziliśmy. Inicjujemy dodatkową zmienną „flag”, która będzie służyła jako do kończenia algorytmu. Na początek bierzemy drugi element i porównujemy go z elementem poprzedzającym. Jeśli napotykamy liczbę większą, to przesuwamy ją o jeden w prawo – pętla działa do czasu napotkania liczby niemniejszej (lub gdy skończą się liczby, czyli flag = 1).

Sortowanie przez scalanie (Merge sort)

Jest to rekurencyjny algorytm sortowania danych, stosujący metodę dziel i zwyciężaj. Odkrycie algorytmu przypisuje się Johnowi von Neumannowi. Złożoność algorytmu wynosi O(n log n), więc jest wydajniejszy niż sortowanie bąbelkowe i przez wstawianie.

```
private static readonly int X = ArraySizeMethod(); //size of the array to sort
private static readonly int[] ArraySize = new int[X]; //array to sort
private static readonly int[] ArraySizeSecondary = new int[X]; //secondary array (for help)
```

Inicjujemy dwie tablice o rozmiarze podanym przez użytkownika (zmienna „X").

```
static void Merging(int beg, int end) //beg - beginning of
{
    //copy values to secondary array
    for (var i = beg; i <= end; i++)
    {
        ArraySizeSecondary[i] = ArraySize[i];
    }
    //merge array
    int b = beg;
    int e = (beg + end) / 2 + 1; //second array has index f
    int x = beg;
    while (b <= (beg + end) / 2 && e <= end) // first array
    {
        if (ArraySizeSecondary[b] < ArraySizeSecondary[e])
        {
            ArraySize[x] = ArraySizeSecondary[b];
            x++;
            b++;
        }
        else
        {
            ArraySize[x] = ArraySizeSecondary[e];
            x++;
            e++;
        }
    }
    //rewrite ending
    while (b <= (beg + end) / 2)
    {
        ArraySize[x] = ArraySizeSecondary[b];
        x++;
        b++;
    }
}
```

Najpierw kopiujemy wszystkie wartości do tablicy zastępczej. Następnie zgodnie z zasadą działania algorytmu dzielimy na dwie części i czynność w pętli powtarza się do momentu otrzymania tablic jednoelementowych.

```
static void Sorting(int beg, int end) //beg
{
    if (beg < end)
    {
        //split the sorted array in half
        Sorting(beg, (beg + end) / 2);
        Sorting((beg + end) / 2 + 1, end);
        //merge sorted arrays
        Merging(beg, end);
    }
}
```

Wyżej jest samo użycie algorytmu. Najpierw rozdzielamy tablice na pół, następuje sortowanie i finalnie scalenie gotowej posortowanej tablicy.

Wyszukiwanie binarne (Binary search)

Algorytm opierający 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.

```
Type amount of your numbers: 8
Type your 1 number (index 0): 5
Type your 2 number (index 1): 9
Type your 3 number (index 2): 4
Type your 4 number (index 3): 69
Type your 5 number (index 4): 3
Type your 6 number (index 5): 66
Type your 7 number (index 6): 44
Type your 8 number (index 7): 1
What number would you like to search? Type: 20
Element not present
Do you want search another number? Type 'Y' (yes) or 'N' (no): Y
What number would you like to search? Type: 69
Element found at index 3
```

Np. jeśli tablica zawiera milion elementów, wyszukiwanie binarne musi sprawdzić maksymalnie 20 elementów ($\log_2 1\,000\,000 \approx 20$) w celu znalezienia żądanej wartości. Dla porównania wyszukiwanie liniowe wymaga w najgorszym przypadku przejrzenia wszystkich elementów tablicy.

```
private static int Search(int[] array, int x)
{
    int left = 0, right = array.Length - 1;
    while (left <= right)
    {
        int mid = left + (right - left) / 2;
        if (array[mid] == x) //check if x is at mid
            return mid;
        if (array[mid] < x) //if x is bigger, ignore left half
            left = mid + 1;
        else //if x is smaller, ignore right half
            right = mid - 1;
    }
    return -1; //element not found
}
```

Definiujemy dwie strony tablicy – lewą (indeks = 0) oraz prawa (długość tablicy – 1). Następnie tworzymy pętlę, która działa dopóki prawa strona jest większa lub równa od lewej. W pętli definiujemy środek tablicy [left + (right – left), prawą odejmujemy od lewej by uniknąć overflow]. Sprawdzamy, czy dana liczba jest na środku – jeśli tak, to zwracamy wynik. Jeśli dana liczba jest większa od indeksu na środku, to ignorujemy lewą część, a jeśli x jest mniejsze to ignorujemy prawą część. Jeśli liczba nie zostanie znaleziona, to zwracamy -1 (nie znaleziono elementu).

Struktury danych

```
* DATA STRUCTURES LIST *

1: Stack
2: Array
3: List
4: LinkedList
5: Queue
6: Binary Tree
Pick number of data structure:
```

Stos (Stack)

To liniowa struktura danych, w której dane dokładane są na wierzch stosu i z wierzchołka stosu są pobierane (bufor typu LIFO, Last In, First Out; ostatni na wejściu, pierwszy na wyjściu). Ideę stosu danych można zilustrować jako stos położonych jedna na drugiej książek – nowy egzemplarz kładzie się na wierzch stosu i z wierzchu stosu zdejmuje się kolejne egzemplarze. Elementy stosu poniżej

wierzchołka można wyłącznie obejrzeć, aby je ściągnąć, trzeba najpierw po kolei ściągnąć to, co jest nad nimi.

```
private static int _top = -1;
```

```
private static void Push(int value) //push - to add item
{
    //check it's full or not
    if (_top == X-1) //here StackSize() - 1
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("\nStack is FULL (or overflow)");
        Console.ResetColor();
    }
    else
    {
        _top++;
        TabStack[_top] = value;
    }
}
```

Pierwsza metoda służy do umieszczenia wartości na szczycie stosu. Po prostu zwiększamy indeks o 1 i przypisujemy nową wartość. Dodatkowo sprawdzamy warunek, czy stos nie jest już pełny bądź przepełniony (na podstawie wielkości tablicy, którą ustala użytkownik na początku działania programu).

```
private static void Pop() //pop - to remove item
{
    //check it's empty or not
    if (_top == -1) //because array is counted from 0
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("\nStack is EMPTY (or underflow)");
        Console.ResetColor();
    }
    else
        _top--; //remove from top
}
```

Powyższa metoda służy do zdjęcia wartości ze stosu. Odpowiednio wcześniej mamy też przypisaną wartość -1 do zmiennej „_top” (tablica jest liczona od 0, więc w przypadku użycia Pop() gdy w tablicy jest 0 elementów wystąpi wynik -1, dlatego ustaliśmy tak zmienną w celu uniknięcia underflow).

```
private static void Peak() //peak - to display item on top
{
    //check it's empty or not
    if (_top == -1) //because array is counted from 0
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("\nStack is EMPTY (or underflow)");
        Console.ResetColor();
    }
    else
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("On top is: " + TabStack[_top]);
        Console.ResetColor();
    }
}
```

Peak() działa podobnie do Pop(), jednak zamiast usuwać wartość na samej górze to ją wyświetla.

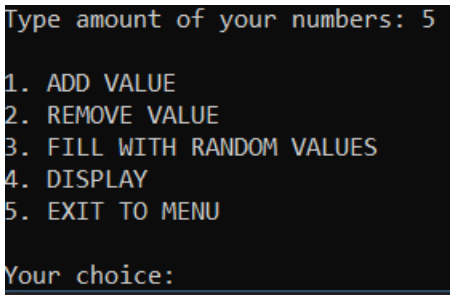
```
private static void Display() //display - to view out
{
    if(_top == -1) //because array is counted from 0
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("\nNothing to display!");
        Console.ResetColor();
    }
    else
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("\nThis is your stack: ");
        for (int i = _top; i >= 0; i--)
        {
            Console.WriteLine(TabStack[i]);
        }
        Console.ResetColor();
    }
}
```

Tutaj wyświetlamy nasz stos. Tak jak w przypadku Pop(), wykorzystujemy to, że wcześniej przypisaliśmy wartość -1 do zmiennej „_top” w celu stwierdzenia, czy aby na pewno mamy co wyświetlić (jeśli _top == -1 to stos jest pusty).

Tablica (Array)

To najprostsza struktura (kontener) danych. Poszczególne elementy dostępne są za pomocą kluczy (indeksu). Indeks najczęściej przyjmuje wartości numeryczne. Rozmiar tablicy jest albo ustalony z góry (tablice statyczne), albo może się zmieniać w trakcie wykonywania programu (tablice

dynamiczne). Tablice jednowymiarowe mogą przechowywać inne tablice, dzięki czemu uzyskuje się tablice wielowymiarowe. W tablicach wielowymiarowych poszczególne elementy są adresowane przez ciąg indeksów.



Praktycznie wszystkie języki programowania obsługują tablice – jedynie w niektórych językach funkcyjnych zamiast tablic używane są listy, także przedstawione w tym programie (choć tablice zwykle też są dostępne). W matematyce odpowiednikiem tablicy jednowymiarowej jest ciąg, a tablicy dwuwymiarowej - macierz.

```
if (choice == 1) //add value
{
    int i, j;
    Console.Write("Type number of index: ");
    while (!int.TryParse(Console.ReadLine(), out i) || !(i >= 0 && i < array.Length))
        Console.Write("Something wrong! Type again: ");
    Console.Write("Type new value: ");
    while (!int.TryParse(Console.ReadLine(), out j) || !(j > 0))
        Console.Write("Something wrong! Type again: ");
    array[i] = j;
}
```

Tutaj po prostu dodajemy wartość do tablicy – podajemy numer indeksu oraz wartość, którą chcemy mu przypisać.

```
if (choice == 2) //remove value
{
    int indexToRemove;
    Console.Write("Type number of index to remove: ");
    while (!int.TryParse(Console.ReadLine(), out indexToRemove) || !(indexToRemove >= 0))
        Console.Write("Something wrong! Type again: ");
    array[indexToRemove] = 0;
}
```

W powyższym wybieramy numer indeksu, którego wartość zostanie usunięta (w tym przypadku zastąpiona 0, uznajemy to jako brak wartości).

```
if (choice == 3) //fill with random numbers
{
    Random random = new Random();
    for (int i = 0; i < array.Length; i++)
    {
        array[i] = random.Next(2000);
    }
}
```

Tutaj po prostu wypełniamy tablicę losowymi wartościami z limitem 2000 (można ten limit dowolnie zmienić w kodzie).

```
if (choice == 4) //display all
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("Your array: ");
    int i=0;
    foreach (var index in array)
    {
            Console.WriteLine($"Index {i}: " + index);
            i++;
    }
    Console.ResetColor();
    Console.WriteLine();
}
```

Używając pętli foreach (dla każdego elementu) wyświetlamy indeksy oraz ich zawartość.

Listy wiązane (Linked list)

Zacznijmy od pojęcia listy (którą także zaimplementowałem w programie używając klasy generycznej). Jest to struktura danych służąca do reprezentacji zbiorów dynamicznych, w której elementy ułożone są w liniowym porządku. Rozróżniane są dwa podstawowe rodzaje list: lista jednokierunkowa w której z każdego elementu możliwe jest przejście do jego następnika oraz lista dwukierunkowa w której z każdego elementu możliwe jest przejście do jego poprzednika i następnika.

```
1. INSERT FRONT
2. INSERT LAST
3. DELETE BY KEY
4. DELETE BY POSITION
5. DISPLAY
6. FIND
7. REVERSE
8. EXIT TO MENU
Your choice:
```

Elementy na liście wiązanej są łączone za pomocą wskaźników. Dzięki temu można łatwo przejść od jednego elementu do następnego.

```
internal class Node //c
{
    internal int data;
    internal Node Next;
    2 references | Zenderable, 5 d
    public Node(int d)
    {
        data = d;
        Next = null;
    }
}
```

Na początek definiujemy Node – z ang. węzeł. Inicjujemy konstruktor, dzięki któremu tworzymy nowy węzeł oraz zmienną/właściwość Next (następny), dzięki któremu mamy dostęp do następnej wartości.

```
private class SingleLinkedList
{
    internal Node Head;
}
```

Powyżej inicjujemy Head – z ang. głowa. Jest to pierwszy element (aka początek) naszej jednokierunkowej listy wiązanej.

```
private static void InsertFront(SingleLinkedList singlyList, int newData)
{
    Node newNode = new Node(newData); //create a new node. The next of th
    newNode.Next = singlyList.Head;
    singlyList.Head = newNode; //the previous Head node is now the second
}
```

Przechodzimy do pierwszej metody, służy ona do dodawania elementu na początek listy (=AddFirst). Tworzymy nowy węzeł, który będzie od strony „głowy”. Zastępuje on poprzednią „głowę/początek”, która staje się teraz drugim elementem (następnym).

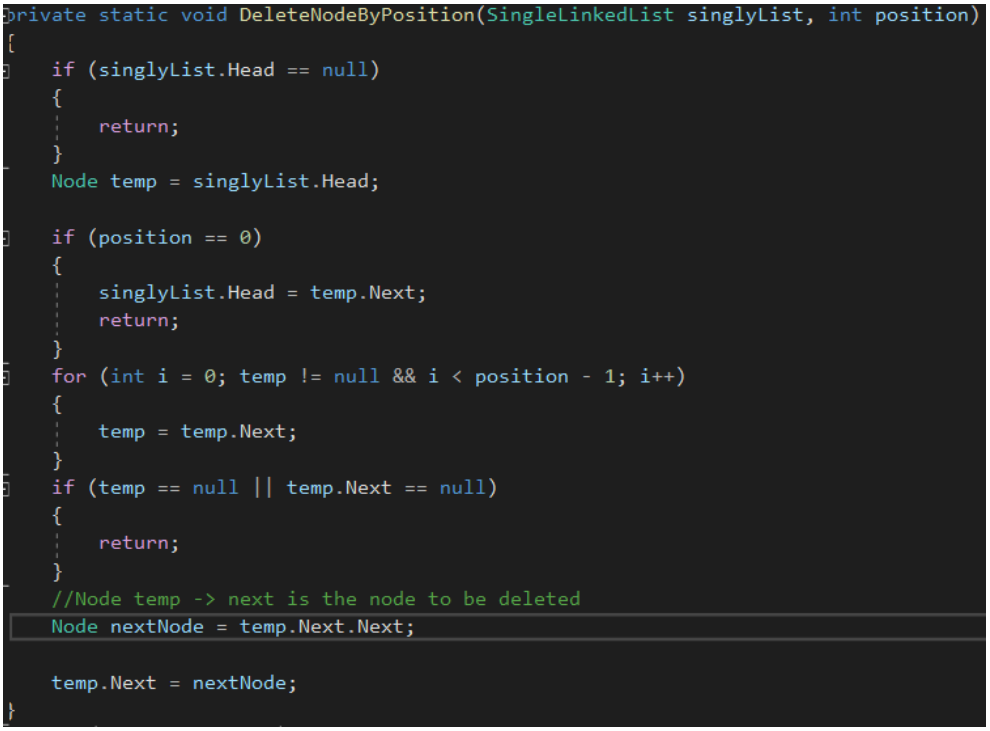
```
private static void InsertLast(SingleLinkedList singlyList, int newData)
{
    Node newNode = new Node(newData);

    if (singlyList.Head == null) //if the linked list is empty, then we s
    {
        singlyList.Head = newNode;
        return;
    }
    Node lastNode = GetLastNode(singlyList); //if the linked list is not
    lastNode.Next = newNode;
}
1 reference | Zenderable, 6 days ago | 1 author, 1 change
private static Node GetLastNode(SingleLinkedList singlyList)
{
    Node temp = singlyList.Head;
    while (temp.Next != null)
    {
        temp = temp.Next;
    }
    return temp;

}
```

Owa metoda służy do dodawania elementu na koniec listy (=AddLast). Tworzymy nowy węzeł, jednak by dodać element na koniec listy, musimy najpierw sprawdzić czy owa lista nie jest pusta, ponieważ w takim przypadku dodana wartość stanie się tzw. głową. Jeśli lista nie jest pusta, to używamy oddzielnej metody, która służy do wyszukania ostatniego węzła, który dzięki temu staje się przedostatnim węzłem, zaś nowy podany przez nas węzeł staje się tym ostatnim.

```
private static void DeleteNodeByKey(SingleLinkedList singlyList, int key)
{
    Node temp = singlyList.Head;
    Node prev = null;
    if (temp != null && temp.data == key)
    {
        singlyList.Head = temp.Next;
        return;
    }
    while (temp != null && temp.data != key)
    {
        prev = temp;
        temp = temp.Next;
    }
    if (temp == null)
    {
        return;
    }
    if (prev != null) prev.Next = temp.Next;
}
```

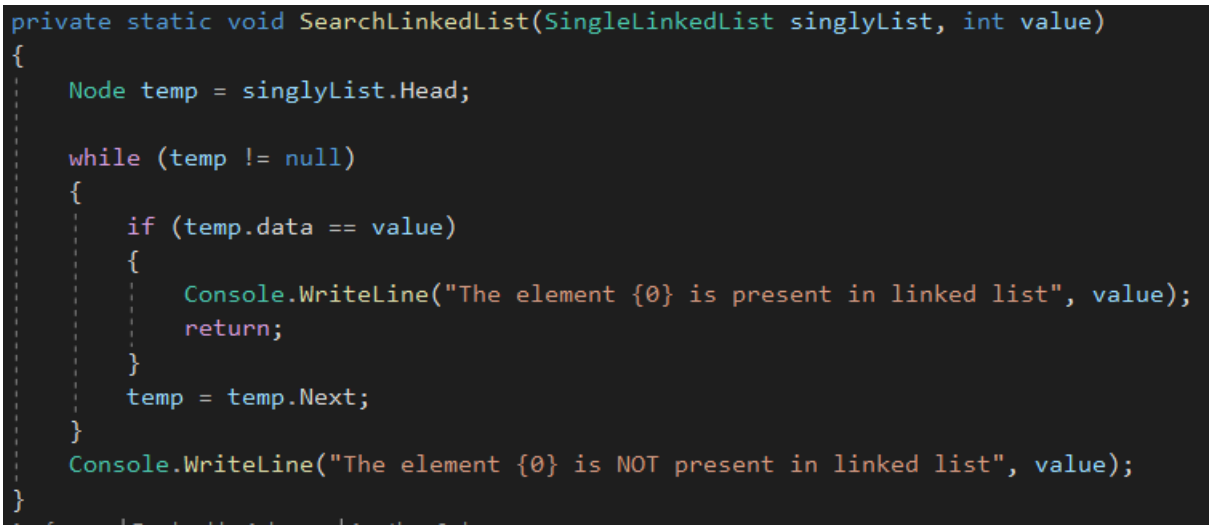
Dzięki tej metodzie mamy możliwość usunięcia pierwszego wystąpienia podanej przez nas wartości. Sprawdzamy, czy lista nie jest pusta i zawiera daną wartość – jeśli tak, to do głowy zostaje

przywiązana następna wartość (ta, która nie będzie usuwana). Następnie w pętli program dopóki nie znajdzie danej wartości, będzie przeszukiwać listę. Usunięta wartość staje się null (z ang. zero/unieważniona/nieważna).



```
private static void DeleteNodeByPosition(SingleLinkedList singlyList, int position)
{
    if (singlyList.Head == null)
    {
        return;
    }
    Node temp = singlyList.Head;

    if (position == 0)
    {
        singlyList.Head = temp.Next;
        return;
    }
    for (int i = 0; temp != null && i < position - 1; i++)
    {
        temp = temp.Next;
    }
    if (temp == null || temp.Next == null)
    {
        return;
    }
    //Node temp -> next is the node to be deleted
    Node nextNode = temp.Next.Next;

    temp.Next = nextNode;
}
```

Tutaj mamy podobne działanie, jednak zamiast usuwać wartość, to usuwana jest wartość na podanej pozycji (aka indeksu) przez użytkownika. Sprawdzamy, czy lista nie jest pusta i tworzymy węzeł tymczasowy. Jeśli wybrana pozycja jest równa 0, to usuwamy głowę i ustawiamy następną wartość nową głową. W pętli szukamy wybranej pozycji. Następnie wartość w węźle tymczasowym zostaje usunięta.



```
private static void SearchLinkedList(SingleLinkedList singlyList, int value)
{
    Node temp = singlyList.Head;

    while (temp != null)
    {
        if (temp.data == value)
        {
            Console.WriteLine("The element {0} is present in linked list", value);
            return;
        }
        temp = temp.Next;
    }
    Console.WriteLine("The element {0} is NOT present in linked list", value);
}
```

Powyższa metoda służy do wyszukania wartości w liście wiązanej. Tworzymy węzeł tymczasowy i za pomocą pętli szukamy danej wartości. Jeśli nie zostanie znaleziona, to wyświetlamy odpowiedni komunikat.

```
private static void ReverseLinkedList(SingleLinkedList singlyList)
{
    Node prev = null;
    Node current = singlyList.Head;

    while (current != null) //traverse linked list using two pointer
    {
        var temp = current.Next; //Move one pointer by one and other
        current.Next = prev;
        prev = current;
        current = temp;
    }
    singlyList.Head = prev; //when the fast pointer reaches end slow
}
```

Tutaj odwracamy naszą listę wiązaną. Używając dwóch punktów (prev i current) w pętli odwracamy listę. Jeden punkt przesuwamy o jeden, drugi o dwa. Pierwszy (szybszy) punkt osiągnie koniec, zaś drugi wolniejszy osiągnie środek – wtedy lista jest w pełni odwrócona.

```
private static void PrintList(SingleLinkedList singlyList)
{
    Node n = singlyList.Head;
    while (n != null)
    {
        Console.Write(n.data + " ");
        n = n.Next;
    }
}
```

Ta funkcje wyświetla całą listę za pomocą pętli while (dopóki n nie równa się null, czyli koniec listy).

Kolejka (Queue)

To liniowa struktura danych, w której nowe dane dopisywane są na końcu kolejki, a z początku kolejki pobierane są dane do dalszego przetwarzania (bufor typu FIFO, First In, First Out; pierwszy na wejściu, pierwszy na wyjściu).

```
Type size of queue: 5

1. INSERT
2. REMOVE
3. PEEK
4. DISPLAY
5. EXIT TO MENU
Your choice:
```

Operacje związane z kolejką zwyczajowo nazywa się enqueue („zakolejkuj”) oraz dequeue („odkolejkuj”). Kolejka działa tak jak kolejka w sklepie i ma podobną strukturę. Składa się z początku (Front, Head) oraz końca (Rear). Nowy element zawsze dodawany jest na końcu, analogicznie do klienta, który zawsze staje na końcu kolejki. Usunięcie z kolejki odpowiada obsłużeniu klienta w sklepie, czyli osoby, która aktualnie czekała najdłużej (tj. znajduje się z przodu kolejki).

```
private static int _front, _rear, _capacity;
private static int[] _queue;
1 reference | Zenderable, 1 day ago | 1 author, 2 changes
private Queue(int c)
{
    _front = _rear = 0;
    _capacity = c;
    _queue = new int[_capacity];
}
```

Tworzymy konstruktor kolejki, rozmiar (zdefiniowany przez użytkownika) oraz dwie zmienne początek „_front” i koniec „_rear” równe 0, co oznacza, że kolejka jest pusta. Koniec jest indeksem, do którego elementy są przechowywane w tablicy, zaś początek jest indeksem pierwszego elementu tablicy.

```
private void Enqueue(int data) //function t
{
    if (_capacity == _rear) //check queue i
    {
        Console.Write("\nQueue is full\n");
    }
    else //insert element at the rear
    {
        _queue[_rear] = data;
        _rear++;
    }
}
```

W tym przypadku mamy metodę zakolejkowania. Funkcja ta dopisuje nową wartość na końcu kolejki. Sprawdzamy, czy kolejka jest pełna, jeśli nie jest to zwiększamy indeks o 1 na końcu kolejki i dodajemy nową wartość.

```
private void Dequeue() //function to delete
{
    if (_front == _rear) //if queue is empty
    {
        Console.Write("\nQueue is empty\n");
    }
    else //shift all the elements from index
    {
        for (int i = 0; i < _rear - 1; i++)
        {
            _queue[i] = _queue[i + 1];
        }
        if (_rear < _capacity) //store 0 at
            _queue[_rear] = 0;
        _rear--; //decrement rear
    }
}
```

Powyżej mamy metodę odkolejkowania. Usuwa wartość z końca kolejki. Oczywiście sprawdzamy, czy kolejka nie jest pusta (_rear > 0). Wartość może zostać usunięta, jeśli wszystkie pozostałe wartości przesuniemy o jedno „w lewo”.

```
private void Display() //print queue elements
{
    int i;
    if (_front == _rear)
    {
        Console.Write("\nQueue is empty\n");
        return;
    }
    for (i = _front; i < _rear; i++) //traverse front to rear
    {
        Console.Write(" {0} <-- ", _queue[i]);
    }
}
1 reference | Zenderable, 1 day ago | 1 author, 2 changes
private void Peek()  //print front of queue
{
    if (_front == _rear)
    {
        Console.Write("\nQueue is empty\n");
        return;
    }
    Console.Write("\nFront element is: {0}", _queue[_front]);
}
```

Powyżej mamy metodę wyświetlania kolejki oraz wyświetlania co jest na jej początku. W pierwszym przypadku sprawdzamy oczywiście, czy nie jest pusta – jeśli nie jest, to w pętli wyświetlamy zawartość kolejki. W przypadku Peek() także sprawdzamy czy kolejka nie jest pusta i po prostu wyświetlamy jej początek za pomocą zmiennej „_front” w tablicy.

Binarne drzewo przeszukiwań (Binary search tree)

W informatyce drzewo binarne to jeden z rodzajów drzewa (struktury danych), w którym liczba synów każdego wierzchołka wynosi nie więcej niż dwa. Wyróżnia się wtedy lewego syna i prawego syna danego wierzchołka.

```
This is a binary tree data structure. Type root value (the best will be > 100): 150
Type left value: 60
Type right value: 180

1. ADD
2. REMOVE
3. TRAVERSE
4. DISPLAY
5. EXIT TO MENU
Your choice: 4
3
 ----->
      150
  ┌----+----┐
 60         180
```

Binarne drzewo przeszukiwań to dynamiczna struktura danych będąca drzewem binarnym, w którym lewe poddrzewo każdego węzła zawiera wyłącznie elementy o kluczach mniejszych niż klucz węzła a prawe poddrzewo zawiera wyłącznie elementy o kluczach nie mniejszych niż klucz węzła. Węzły, oprócz klucza, przechowują wskaźniki na swojego lewego i prawego syna oraz na swojego rodzica.

```
public class BinaryTreeNode<T> : TreeNode<T>
{
    4 references | Zenderable, 4 hours ago | 1 author, 1 change
    public BinaryTreeNode() => Children = new List<TreeNode<T>>() { null, null };
    11 references | Zenderable, 4 hours ago | 1 author, 1 change
    public BinaryTreeNode<T> Parent { get; set; }
    17 references | Zenderable, 4 hours ago | 1 author, 1 change
    public BinaryTreeNode<T> Left
    {
        get { return (BinaryTreeNode<T>)Children[0]; }
        set { Children[0] = value; }
    }
    15 references | Zenderable, 4 hours ago | 1 author, 1 change
    public BinaryTreeNode<T> Right
    {
        get { return (BinaryTreeNode<T>)Children[1]; }
        set { Children[1] = value; }
    }
    2 references | Zenderable, 4 hours ago | 1 author, 1 change
    public int GetHeight()
    {
        int height = 1;
        BinaryTreeNode<T> current = this;
        while (current.Parent != null)
        {
            height++;
            current = current.Parent;
        }
        return height;
    }
}
```

Węzeł drzewa reprezentowany jest przez wystąpienie klasy BinaryTreeNode, która dziedziczy po klasie generycznej TreeNode. W powyższej klasie deklarujemy dwie wartości – Left oraz Right, które reprezentują dwoje dzieci. Kolekcja dzieci powinna zawierać dwa elementy o początkowej wartości null. Konstruktor nadaje właściwości Children wartość domyślną. Aby dodać dziecko należy umieścić jego referencję na początku listy (czyli w Children). Dzięki temu ma dokładnie zawsze dwa elementu i zapewnia dostęp do każdego z nich. Jeśli do elementu przypisano węzeł – zwracana jest referencja dla niego, jeśli nie – wartość null.

```
public BinaryTreeNode<T> Root { get; set; }
6 references | Zenderable, 1 day ago | 1 author, 1 change
public int Count { get; set; }
```

Powyżej wykorzystujemy klasę generyczną. Root wskazuje korzeń, zaś Count zawiera całkowitą liczbę węzłów drzewa.

```
public class BinarySearchTree<T> : BinaryTree<T>
    where T : IComparable
```

Całe drzewo prezentowane jest przez wystąpienie powyższej klasy, dziedziczącej po klasie generycznej BinaryTree. Dane umieszczone w węzłach powinny dawać się porównywać, z tego powodu implementowany jest interfejs IComparable (algorytm musi znać relacje między wartościami).

```
public void Add(T data)
{
    BinaryTreeNode<T> parent = GetParentForNewNode(data);
    BinaryTreeNode<T> node = new BinaryTreeNode<T>() { Data = data, Parent = parent };

    if (parent == null)
    {
        Root = node;
    }
    else if (data.CompareTo(parent.Data) < 0)
    {
        parent.Left = node;
    }
    else
    {
        parent.Right = node;
    }

    Count++;
}
```

Powyższa metoda przyjmuje jeden parametr – wartość, którą trzeba dodać do drzewa. W obrębie metody znajdowany jest element (za pomocą GetParentForNewNode), który będzie rodzicem nowego węzła. Następnie tworzy się wystąpienia klasy BinaryTreeNode oraz nadaje wartości dla

Data i Parent. Następnie sprawdzamy czy znaleziony rodzic ma wartość null (oznacza to, że drzewo nie ma węzłów, a nowy węzeł będzie korzeniem). Następnie sprawdzamy czy dodawana wartość jest mniejsza niż wartość rodzica (w tym przypadku należy dodać nowy węzeł jako lewe dziecko), w przeciwnym razie prawe dziecko. Na koniec zwiększamy wartość licznika elementów przechowywanych w drzewie.

```
private BinaryTreeNode<T> GetParentForNewNode(T data)
{
    BinaryTreeNode<T> current = Root;
    BinaryTreeNode<T> parent = null;
    while (current != null)
    {
        parent = current;
        int result = data.CompareTo(current.Data);
        if (result == 0)
        {
            throw new ArgumentException($"Node {data} exist.");
        }
        else if (result < 0)
        {
            current = current.Left;
        }
        else
        {
            current = current.Right;
        }
    }
    return parent;
}
```

Powyższa metoda pomocnicza zawiera dwie zmienne reprezentujące obecnie analizowany węzeł (current) oraz jego rodzica (parent). W pętli while argument jest modyfikowany dopóki nie znajdzie właściwego miejsca na nowy węzeł. W pętli sprawdzamy czy wartość dodana jest równa wartości dodawanego węzła, jeśli tak to zgłaszamy wyjątek, ponieważ nie możemy dodać kolejnego elementu o tej samej wartości. Jeśli dodawana wartość jest mniejsza niż wartość bieżącego węzła, algorytm szuka dalej miejsca na nowy węzeł w lewym poddrzewie. W przeciwnym razie przeszukiwane jest prawe poddrzewo obecnego węzła. Na końcu zwracamy wartość zmiennej parent wskazującej na nowy węzeł.

```
public void Remove(T data)
{
    Remove(Root, data);
}
```

Powyższa metoda wywołuje kolejną metodę:

```
private void Remove(BinaryTreeNode<T> node, T data)
{
    if (data.CompareTo(node.Data) < 0)
    {
        Remove(node.Left, data);
    }
    else if (data.CompareTo(node.Data) > 0)
    {
        Remove(node.Right, data);
    }
    else
    {
        if (node.Left == null && node.Right == null)
        {
            ReplaceInParent(node, null);
            Count--;
        }
        else if (node.Right == null)
        {
            ReplaceInParent(node, node.Left);
            Count--;
        }
        else if (node.Left == null)
        {
            ReplaceInParent(node, node.Right);
            Count--;
        }
        else
        {
            BinaryTreeNode<T> successor = FindMinimumInSubtree(node.Right);
            node.Data = successor.Data;
            Remove(successor, successor.Data);
        }
    }
}
```

W celu znalezienia wartości do usunięcia porównywana jest bieżąca wartość węzła z wartością do usunięcia i wywołuje rekurencyjnie metodę Remove w lewym albo prawym poddrzewie bieżącego węzła. W warunkach sprawdzane są różne scenariusze (prawy węzeł, lewy węzeł itd.). Najbardziej skomplikowane jest usunięcie węzła mającego dwoje dzieci. W tym przypadku w prawym poddrzewie znajdowany jest węzeł o minimalnej wartości. Następnie usuwaną wartość zamienia się na wartość znalezionego węzła. Na koniec wystarczy dla znalezionego węzła wywołać rekurencyjnie metodę Remove.

```
private void ReplaceInParent(BinaryTreeNode<T> node, BinaryTreeNode<T> newNode)
{
    if (node.Parent != null)
    {
        if (node.Parent.Left == node)
        {
            node.Parent.Left = newNode;
        }
        else
        {
            node.Parent.Right = newNode;
        }
    }
    else
    {
        Root = newNode;
    }
    if (newNode != null)
    {
        newNode.Parent = node.Parent;
    }
}
```

Pierwsza z metod pomocniczych przyjmuje dwa parametry: usuwany węzeł (node) i węzeł, który ma go zastąpić w rodzicu (newNode). Jeśli usuwany korzeń nie jest korzeniem, sprawdzamy czy jest lewym dzieckiem rodzica. Jeśli tak, odpowiednia referencja jest uaktualniana, to znaczy lewym dzieckiem rodzica usuwanego węzła staje się nowy węzeł. W podobny sposób robimy z prawym dzieckiem. Jeśli usuwany węzeł jest korzeniem, wtedy nowym korzeniem staje się węzeł, który ma go zastąpić. Na końcu sprawdzamy czy nowy węzeł nie ma wartości null, co oznacza, że usuwany węzeł nie jest liściem. W tym przypadku właściwości Parent nowego węzła nadaje się wartość wskazującą, że będzie mieć tego samego rodzica co usuwany węzeł.

```
private BinaryTreeNode<T> FindMinimumInSubtree(BinaryTreeNode<T> node)
{
    while (node.Left != null)
    {
        node = node.Left;
    }
    return node;
}
```

Ostatnia pomocnicza metoda przyjmuje jeden parametr – korzeń poddrzewa, w którym szuka wartości minimalnej. Pętla while pobiera element położony najbardziej na lewo, jeśli nie istnieje to zwracana jest aktualna wartość zmiennej node.

```
public bool Contains(T data)
{
    BinaryTreeNode<T> node = Root;
    while (node != null)
    {
        int result = data.CompareTo(node.Data);
        if (result == 0)
        {
            return true;
        }
        else if (result < 0)
        {
            node = node.Left;
        }
        else
        {
            node = node.Right;
        }
    }
    return false;
}
```

Metoda Contains, sprawdza czy drzewo zawiera węzeł o danej wartości. Wyszukiwana wartość w pętli while jest porównywana z wartością bieżącego węzła. Jeśli są jednakowe (porównanie zwraca wynik 0), czyli wartość została znaleziona, zwracana jest wartość logiczna true (sukces). Jeżeli jest mniejsza, to algorytm kontynuuje poszukiwanie w poddrzewie, którego korzeniem jest lewe dziecko bieżącego węzła. W przeciwnym przypadku – prawe poddrzewo. Pętla wykonuje się do momentu, gdy zostanie znaleziony węzeł albo zabraknie dzieci, do których można przejść.

```
public List<BinaryTreeNode<T>> Traverse(TraversalEnum mode)
{
    List<BinaryTreeNode<T>> nodes = new List<BinaryTreeNode<T>>();
    switch (mode)
    {
        case TraversalEnum.PREORDER:
            TraversePreOrder(Root, nodes);
            break;
        case TraversalEnum.INORDER:
            TraverseInOrder(Root, nodes);
            break;
        case TraversalEnum.POSTORDER:
            TraversePostOrder(Root, nodes);
            break;
    }
    return nodes;
}
```

Ostatnią metodą, jest Traverse – czyli przechodzenie przez wszystkie węzły drzewa. Istnieją trzy metody przechodzenia: przejście wzdłużne (preOrder), poprzeczne (inOrder) oraz wsteczne (postOrder). Aby przejść wzdłużnie, należy rozpocząć od korzenia. Potem odwiedza się lewe

dziecko, a na końcu prawe (zasada dla każdego węzła). Za pomocą powyższej metody (oraz wyliczenia TraversalEnum) wybieramy interesujące nas przejście.

```
private void TraversePreOrder(BinaryTreeNode<T> node, List<BinaryTreeNode<T>> result)
{
    if (node != null)
    {
        result.Add(node);
        TraversePreOrder(node.Left, result);
        TraversePreOrder(node.Right, result);
    }
}
```

Metoda ta porównuje dwa parametry: bieżący węzeł (node) i listę odwiedzonych już węzłów (result). Najpierw sprawdzamy czy węzeł istnieje (nie może mieć wartości null), następnie bieżący węzeł dodaje do kolekcji odwiedzonych i tą samą metodą przechodzi przez lewe dziecko, a na końcu prawe.

```
private void TraverseInOrder(BinaryTreeNode<T> node, List<BinaryTreeNode<T>> result)
{
    if (node != null)
    {
        TraverseInOrder(node.Left, result);
        result.Add(node);
        TraverseInOrder(node.Right, result);
    }
}
```

InOrder wywołuje się rekurencyjnie na lewym dziecku, dodaje bieżący węzeł i zaczyna się przejście poprzeczne po prawym dziecku.

```
private void TraversePostOrder(BinaryTreeNode<T> node, List<BinaryTreeNode<T>> result)
{
    if (node != null)
    {
        TraversePostOrder(node.Left, result);
        TraversePostOrder(node.Right, result);
        result.Add(node);
    }
}
```

Metoda przejścia wstecznego wygląda tak ja powyżej, robimy to samo tylko w innej kolejności – wywołujemy rekurencyjnie na lewym dziecku, następnie na prawym i dopiero przechodzimy wstecznie.

```
public int GetHeight()
{
    int height = 0;
    foreach (BinaryTreeNode<T> node in Traverse(TraversalEnum.PREORDER))
    {
        height = Math.Max(height, node.GetHeight());
    }
    return height;
}
```

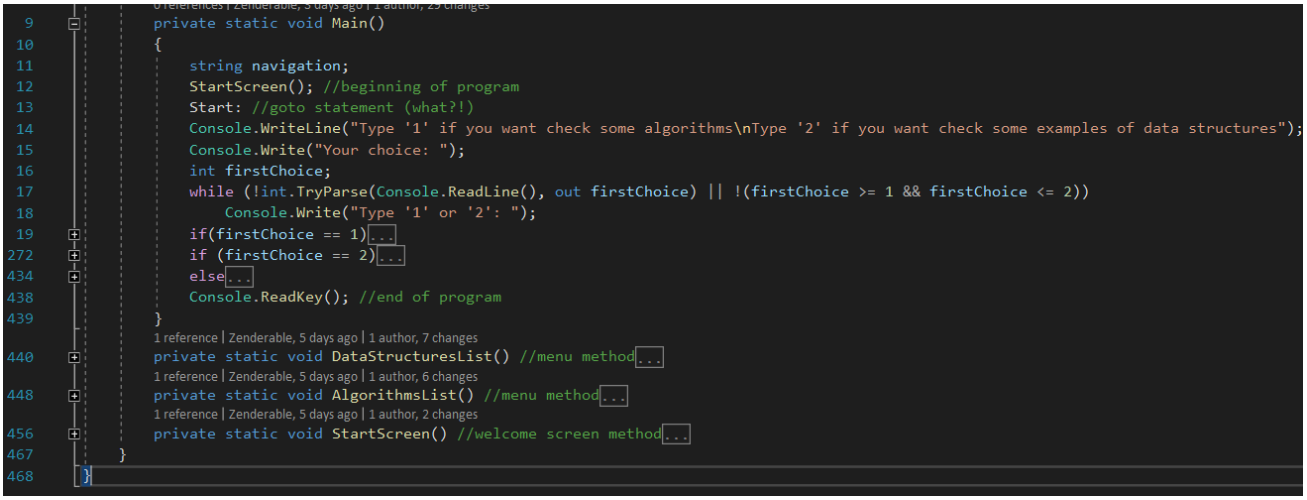
Ostatnia metoda zwraca wysokość drzewa (maksymalna liczba kroków potrzebnych do przejścia z jednego liścia do korzenia). Kod przechodzi przez wszystkie węzły drzewa metodą wzdłużną – jeśli jest większa niż dotychczasowa, to zapisuje wysokość jako nową. Na końcu zwracana jest obliczona wartość.

Inne

Oprócz algorytmów, w programie napisane są menu do poruszania się po danym algorytmie/strukturze danych. Wszystkie są w podobnym stylu jak poniżej:

```
public static void StackMenu()
{
    int choice;
    do
    {
        Console.Write("\n1. PUSH\n2. POP\n3. PEAK\n4. DISPLAY\n5. EXIT TO MENU\nYour choice: ");
        while (!int.TryParse(Console.ReadLine(), out choice) || !(choice >= 1 && choice <= 5))
            Console.Write("Type '1' to '5': "); //if wrong, type correct integer
        if (choice == 1) //push
        {
            Console.Write("Enter value: ");
            int value;
            while (!int.TryParse(Console.ReadLine(), out value))
                Console.Write("Type value: ");
            Push(value); //add item
        }
        if (choice == 2) //pop
        {
            Pop(); //remove item
        }
        if (choice == 3) //peak
        {
            Peak(); //display top item
        }
        if (choice == 4) //display
        {

            Display(); //view
        }
    } while (choice != 5); //exit, end of loop
}
```

Za pomocą pętli do… while wywołujemy daną metodę bądź kończymy algorytm.



AlgorithmsAndDataStructures.cs służy jako Main programu. Wszystkie funkcje są tam wywoływane oraz zbudowane jest menu poruszania się za pomocą goto.

Źródła

- Wszystkie opisy algorytmów/struktur danych (co to za algorytm) pochodzą z *https://pl.wikipedia.org/wiki/*
- Opis implementacji listy wiązanej *https://www.c-sharpcorner.com/article/linked-list-implementation-in-c-sharp/*
- Wyszukiwanie binarne *https://www.geeksforgeeks.org/binary-search/*
- Kolejka *https://www.geeksforgeeks.org/queue-data-structure/#implementation*
- Struktury danych i algorytmy w języku C#, Marcin Jamro (Drzewo binarne)
- Pozostałe: *https://stackoverflow.com/*