1. **ArrayList ->** типичен лист, с основа масив – добавят се елементи и ако се достигне лимита, размера на масива отдолу се удвоява, а при махане ако масива е с брой равен или по-малък от капацитета на масива / 4, капацитета се разделя на 2.
2. **ReversedList ->** същото, само елементите се достъпват в обратен от реда на добавяне.
3. **LinkedList<T>** -> Серия от Node-ове, началния е Head, последния Tail, свързани са в едната посока като всеки елемент има Node.Next.
4. **Circular Queue** -> Кръгова опашка, на която началния и крайния индекс могат да бъдат всяка клетка от масива, който стои отдолу. Enqueue и Dequeue си запазват реда – първия влязъл излиза първи. За да се запази кръговото обикаляне се ползва формулата „***sourceIndex = (sourceIndex + 1) % this.elements.Length;***“
5. **DoublyLinkedList<T>** -> Същото като LinkedList<T>, само че елементите са свързани и в двете посоки – имат и Next, и Previous елементи закачени за тях.
6. **Tree<T> ->** Всеки Tree<T> елемент има T Value и списък от Children. Принтираме рекурсивно с identation – Print(int indent = 0) Console.Write(new string(' ', 2 \* indent));

Console.WriteLine(this.Value); и всяко дете с child.Print(indent + 1);

Имплементираме Each(Action<T> action), за да имаме Traverse на дървото с достъп до всеки елемент и възможност за действие върху даден елемент. action(this.Value) -> child.Each(action);

DFS -> Прави се private DFS(this*(което е Tree<T>tree)*, resultList), където рекурсивно имаме foreach ->{DFS(child, resultList)} resultList.Add(tree.Value)

BFS -> прави се Queue, добавяме root-а и започваме да пълним resultList отгоре – надолу. While(queue.Count > 0)

{var current = queue.Dequeue();

resultList.Add(current.Value); - > foreach child in children

queue.Enqueue(child);}

1. **BinaryTree<T>**

Изграждаме структурата на нашето BinaryTree<T> чрез съставните му елементи – BinaryTree<T>, които имат стойност, ляво дете и дясно дете. Отново принтираме рекурсивно с начален indent = 0 и ако например имаме PreOrder първо принтираме корена, после ляво дете и накрая дясно. С всяко рекурсивно „навлизане“ по-дълбоко в структурата на дървото indent-а се увеличава ->

this.LeftChild(или RightChild).PrintIndentedPreOrder(indent + 1);

Когато Each е InOrder реда на извикване е this.LeftChild.EachInOrder(action), после настоящия елемент action(this.Value) и накрая дясното дете, отново рекурсивно.

1. **BinarySearchTree<T>**

Изграждаме си **Node** структура с **T Value, Node Left и Node Right.** В **EachInOrder(Action<T> action)** извикваме нов private EachInOrder, който приема Node и action. Тъй като трябва да бъде InOrder, реда отново е ляво - this.EachInOrder(node.Left, action), корен - action(node.Value) и дясно - this.EachInOrder(node.Right, action).

При **bool Contains** вземаме Node current = root, ако има такъв започваме да вървим в създадения While(current != null) наляво или надясно според стойността на търсения елемент и CompareTo() и в крайна сметка или ще стигнем до елемента, или до null, затова последния ред е return current != null;

В **Insert** отново първо проверяваме root-а и ако е null го сетваме да е равен на New Node(новия елемент), в противен случай си правим parent = null и current = this.root. Докато current != null сетваме parent = current и използваме CompareTo(), за да намерим място на нашия нов елемент, ако имаме вече такъв елемент, в конкретния пример игнорираме нововъведения елемент с директен return от Insert метода.

След като сме намерили място(parent-а) на новия ни елемент остава само да сравним и ако е по-голям от parent-a – става десен Child, ако е по-малък – ляв.

В **Search** започваме търсенето от Node current = root-а и според резултата от CompareTo() се движим наляво или надясно, тъй като трябва да бъде върнат резултат от тип BinarySearchTree<T> връщаме New BinarySeachTree<T>(current), в който нов конструктор ползваме рекурсивен Copy метод, в който ако current не е null копираме дървото в същия вид - > this.Insert(node.Value);

this.Copy(node.LeftNode);

this.Copy(node.RightNode);

В DeleteMin намираме най-малкия елемент

В **Range** правим Queue, вземаме границите, правим Traverse на дървото и ако някой елемент е в границите го добавяме в Queue-то

1. **BinarySearchTree – Exercise**
2. **Max Binary Heap**
3. **Heap Sort**
4. **DecreaseKey(Heap Exercise)**
5. **A\*(Heap Exercise)**
6. **RedBlackTree Insert**
7. **Hierarchy**
8. **AVL Insertion**
9. **AVL First Last List**
10. **AVL Delete AVL Tree**
11. **Ropes and Tries – Trie**
12. **Ropes and Tries – Text Editor**
13. **Interval Tree**
14. **K-d Tree**