1. **Linear Data Structures – Lists and Complexity**

**An Abstract Data Type (ADT) is**

A data type together with the operations, whose properties are specified independently of any particular implementation

In computer science, a **data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.**

* **Examples of data structures:**
  + Person structure (first name + last name + age)
  + Array of integers – int[]
  + List of strings – List<string>
  + Queue of people – Queue<Person>
  1. **Arrays**

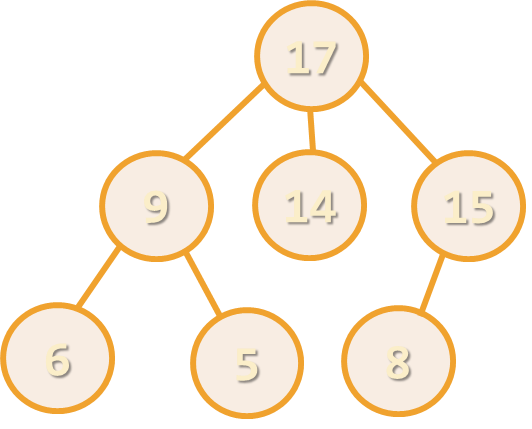
Ordered, Very lightweight, Has a fixed size, Usually built into the language

Many collections are implemented by using arrays, e.g.

* 1. **List<T>** in .NET

|  |  |  |
| --- | --- | --- |
| **Complexity** | **Notation** | **Description** |
| **constant** | **O(1)** | **n = 1 000 🡪 1-2 operations** |
| **logarithmic** | **O(log n)** | **n = 1 000 🡪 10 operations** |
| **linear** | **O(n)** | **n = 1 000 🡪 1 000 operations** |
| **linearithmic** | **O(n\*log n)** | **n = 1 000 🡪 10 000 operations** |
| **quadratic** | **O(n2)** | **n = 1 000 🡪 1 000 000 operations** |
| **cubic** | **O(n3)** | **n = 1 000 🡪 1 000 000 000 operations** |
| **exponential** | **O(nn)** | **n = 10 🡪 10 000 000 000 operations** |

1. **Linear Data Structures – Stacks and Queues**
2. **Linked List**
   * 1. *Nodes*
3. **Stacks**
4. *Static and Linked Implementation*
5. *The Stack<T> Class in .NET Framework*
6. **Queues**
7. *Circular and Linked Implementation*
8. *The Queue<T> Class in .NET Framework*
9. **Basic Tree Data Structures**
   1. **Branched recursive data structures**
      1. *Consisting of nodes*
      2. *Each node connected to other nodes*
   2. **Structure – Terminology**
      1. *Node, Edge*
      2. ***Root****, Parent, Child, Sibling*
      3. *Depth, Height*
      4. *Subtree*
      5. *Internal node, Leaf*
      6. *Ancestor, Descendant*

** **

1. **Tree Traversal Algorithms – BFS and DFS**
   1. **Depth-First Search (DFS) - first visits all descendants of given node recursively, finally visits the node itself**

**DFS algorithm pseudo code:**

*DFS (node)*

*{ for each child c of node*

*DFS(c);*

*print node;}*

* 1. **Breadth-First Search** (BFS) - first visits the neighbor nodes, then the neighbors of neighbors, etc.

**BFS algorithm pseudo code:**

*BFS (node)*

*{queue 🡨 node*

*while queue not empty*

*v 🡨 queue*

*print v*

*for each child c of v*

*queue 🡨 c}*

* 1. ***Binary Trees Traversal: Pre-order -*** *Root 🡪 Left 🡪 Right*
  2. ***Binary Trees Traversal: In-order -*** *Left 🡪 Root 🡪 Right*
  3. ***Binary Trees Traversal: Post-order -*** *Left 🡪 Right 🡪 Root*

1. **Binary Search Tree – Two Children at most**

Binary search trees are ordered binary trees

Balanced trees have roughly the same height of their left and right children

The value of the left child is smaller than the value of right child.

* What is the speed of the **Search()** and **Range(T, T)** operations on BST?
  + O(n) 

1. **Heaps and Priority Queues**
2. **Priority Queue -** *Dequeue Most Significant Element*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Insert** | **Pull** | **Peek** |
| **Unsorted Array** | **O(1)** | **O(N)** | **O(N)** |
| **Sorted Array** | **O(N)** | **O(1)** | **O(1)** |
| **Goal** | **O(logN)** | **O(logN)** | **O(logN)** |

Retains a **specific order** to the elements

**Higher priority** elements are **pushed to the beginning** of the queue

**Lower priority** elements are **pushed to the end** of the queue

* **Priority queue** abstract data type (ADT) supports:
  + **Insert(element)**
  + **Pull()** 🡪 **max/min element**
  + **Peek()** 🡪 **max/min element**
* Where **element** has a priority

1. **Heaps**
   * Tree-based data structure
   * Stored in an array

Heaps hold the **heap property** for each node:

* + **Min Heap**
    1. parent ≤ children
  + **Max Heap**
    1. parent ≥ children

1. **B-Trees and Red-Black Trees**

2-3 Tree – BST – Red-BlackTree perf., c depends on implementation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Structure** | **Worst case** | | | **Average case** | |
| **Search** | **Insert** | **Delete** | **Search Hit** | **Insert** |
| BST | N | N | N | 1.39 lg N | 1.39 lg N |
| 2-3 Tree | c lg N | c lg N | c lg N | c lg N | c lg N |
| Red-Black | 2 lg N | 2 lg N | 2 lg N | lg N | lg N |

1. **AA-Trees and AVL trees**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Structure** | **Worst case** | | | **Average case** | |
| **Search** | **Insert** | **Delete** | **Search Hit** | **Insert** |
| BST | N | N | N | 1.39 lg N | 1.39lg N |
| 2-3 Tree | *c*lg N | *c*lg N | *c*lg N | *c*lg N | *c*lg N |
| Red-Black | 2lg N | 2lg N | 2lg N | lg N | lg N |
| AVL Tree | 1.44lg N | 1.44lg N | 1.44lg N | lg N | lg N |

Въпреки, че пише 1.44lg N, AVL дървото е по-бързо за Search, но заради по-големия брой балансирания(ротации) в него Insert и Delete са малко по-бавни, отколкото в Red-Black дърво.

1. **Rope and Trie**
2. **Quad Trees, K-d Trees, Interval Trees**
3. **Hash Tables, Sets and Dictionaries**
4. **Combining Data Structures**
5. **Силни страни на конкретната структура.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Structure** | **Worst case** | | | **Average case** | |
| **Search** | **Insert** | **Delete** | **Search Hit** | **Insert** |
| BST | N | N | N | 1.39 lg N | 1.39lg N |
| 2-3 Tree | *c*lg N | *c*lg N | *c*lg N | *c*lg N | *c*lg N |
| Red-Black | 2lg N | 2lg N | 2lg N | lg N | lg N |
| AVL Tree |  | 1.44lg N | 1.44lg N | lg N | lg N |
| Ropes |  |  |  |  |  |
| Trie |  |  |  |  |  |
| Quad Tree |  |  |  |  |  |
| K-D Tree |  |  |  |  |  |
| Interval Tree |  |  |  |  |  |
| Sets |  |  |  |  |  |
| Dictionaries |  |  |  |  |  |