**SECTION 1. ABSTRACT DATA TYPES (ADT)**

* 1. **ADT concept**

Before describing data structures in the programming language Golang should start with the concepts of 'data', 'data type' and 'data structure'. In the most general sense, data is any unstructured set (collection) of characters that are collected and processed for any purpose, primarily to extract information for subsequent acquisition of knowledge and meanings. Data structures manifold was predetermined the appearance of a data type category . The data type should be understood as an atomic, indivisible data unity, defined either by the programming language standard or by the user, and representing a set of real-world objects at the level of their most essential parameters. Generalization of experience of application of different types of data in different programming languages led to the emergence of the «abstract data type» category.

The data types were first described by D. Knuth in his classical work “The Art of Programming” [Knut], in which the author described data structures defined as ways of organizing data within a program. Together with the description of the data structures themselves, Knuth provides the “processing algorithms” of these structures in a language of special terms, reflecting actions with elements of this structure, such as structure size, adding a new element, push and pop items, etc.

In most literary sources, the abstract data type is understood as a mathematical model that defines a set of data values (carrier set) and a set of methods for handling that data (method set). Here are some examples of ATD:

*Boolean -* the logical ATD is a set of values {true, false}. The set of methods includes comparison operators (as well as logical operations &&, II, ==, !=).

*Integer -* integer ATD is a set of {..., -2, -1, 0, 1, 2, ...}, and the set of methods includes addition, subtraction, multiplication, division, remainder of division, change of sign, etc.

*String -* аn ATD string set is a set of all finite sequences of characters in a certain alphabet, including an empty sequence (an empty string). The set of methods includes concatenation, definition of string length, substring, character index, etc.

*Bit string – the* carrier set of bit string is a set of all end-sequences of bits, including empty bits strings.The set of ATD method of bit string includes a complement (which flips all bits), shifts (which rotate a string of bits left or right), connection and disjunction (which merge bits at appropriate locations in strings), concatenation and truncation.

The author of the outstanding work «Perfect Code» S. McConnell significantly expanded the concept of ATD and drew attention to the fact that abstract data types «allow to work with entities of the real world, and not with low-level entities of implementation» [Mac, C. 122]. In particular, these entities are data structures for organizing the computing process. The simplified classification of abstract data structures can be presented as follows (Figure 1.1.)

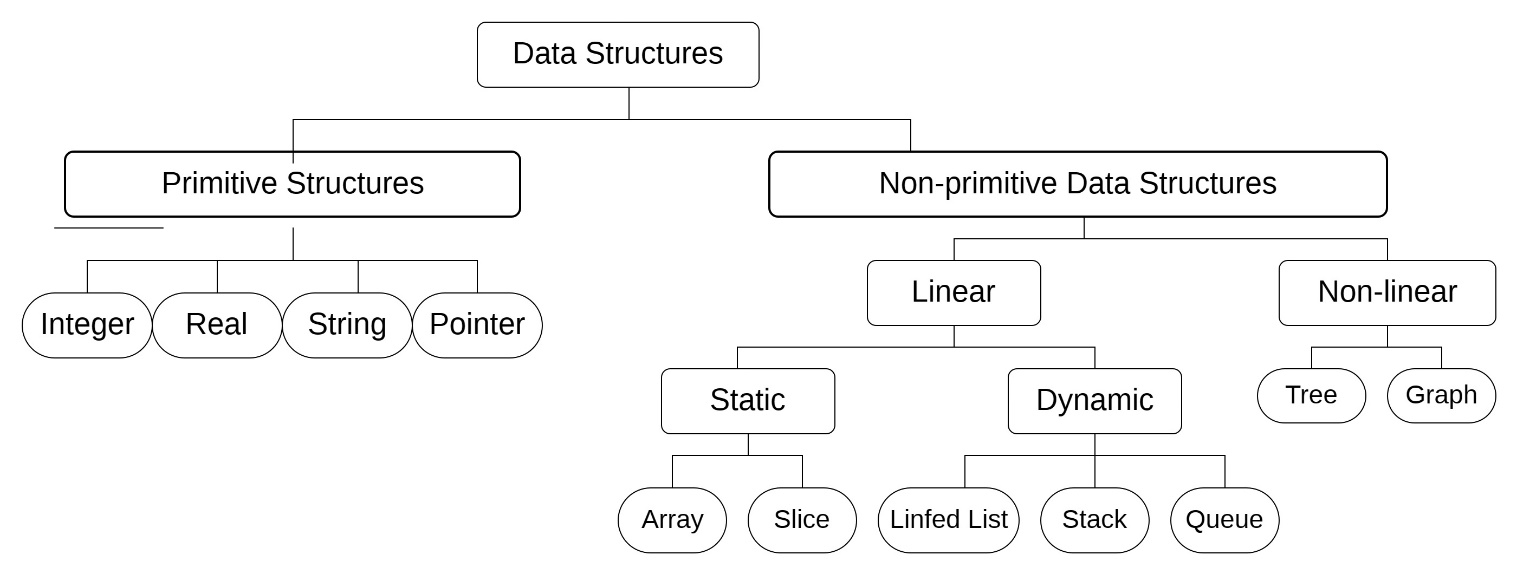


Figure 1.1. Data structures classification

Let's take a closer look at the classification. Note that each of these structures has its own advantages and disadvantages, which predetermines the need for their thorough analysis in terms of computer memory costs and access time to their elements. In this work, the main attention is focused on data structures presented in RAM. First of all, data structures are divided into linear and non-linear. In a linear data structure, its elements are contiguous, that is, the data is ordered sequentially. Such designs are quite simple to implement. At the same time, linear data structures do not provide efficient use of memory.

In non-linear data structures, the organization of elements is not sequential. Data elements in a non-linear structure may be connected to several other data elements to reflect special relationships between them. In addition, in non-linear structures, it is impossible to go through the elements in one pass. Nonlinear data structures include maps, dictionaries, trees, and graphs. Homogeneous data structures include most of these data structures, non-homogeneous - structures that consist of data of a different nature. For example, lists are based on nodes, represented by structures that include two fields: a numerical value and the address of the next node. Typical examples of heterogeneous structure are *dictionaries, maps, and hash tables*. Let's consider the main abstract data types according to the principle of linearity.

* 1. **Linear Abstract Data Types**

**1.2.1 Array and Slice**

Let's start with the fundamental abstract data type, the array. The fundamental nature of an array, as a data structure, lies in their direct correspondence to memory systems on all computers. To retrieve the contents of a word from memory, machine language requires an address. Thus, the entire memory of the computer can be considered as an array, where memory addresses correspond to indices. Most machine language processors translate programs that use arrays into efficient machine language programs that access memory directly.

An array is a fixed set of same-type data that is stored as a continuous sequence, the indexing of which elements can start with 0 or 1. When created and initialized, the array is declared via an identifier or an address pointer at the initial address (0 or 1) element (Figure 1.2).

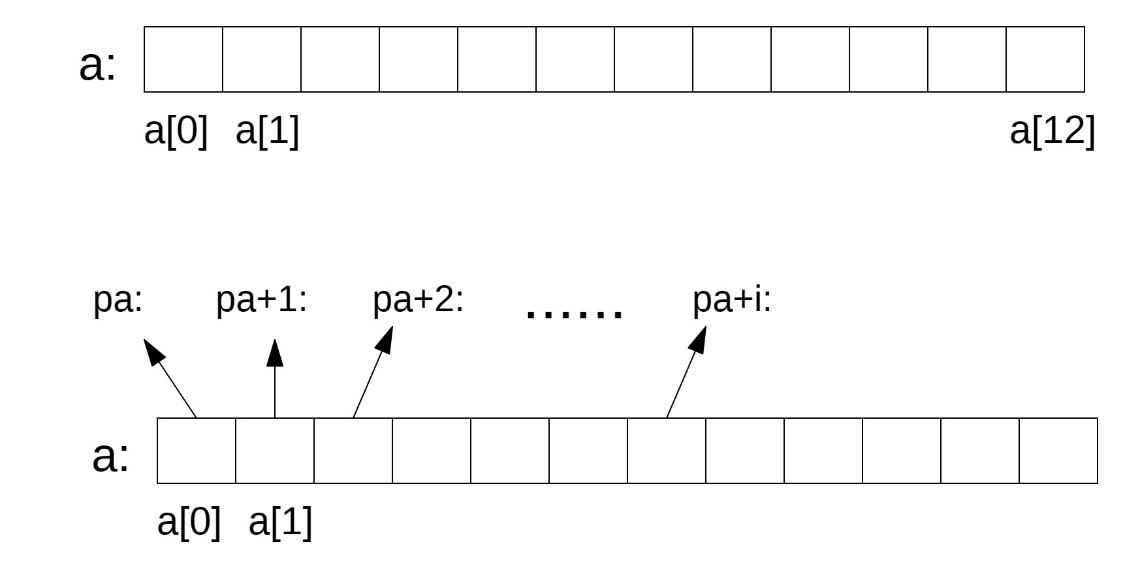


Figure 1.2. Array declaration method (*pa*  - cell address *a[0]*)

The choice of declaration type determines how individual elements are accessed. The entry a[i], where a is an identifier, refers us to the i-th element of the array. At the same time, an array can be declared using a pointer, which refers the program code to the address of the initial element of the array. In most languages ​​that use the concept of pointers, an array is specified by the expression: \*pa, and the real address in a hexadecimal expression is specified by the character &: pa = &a[0]. If pa points to some element of the array, then pa+1 by definition points to the next element, pa+i to the i-th element after pa, and pa-i to the i-th element before pa. So if pa points to a[0], then \*(pa+1) is the content of a[1], a+i is the address of a[i], a \*(pa+i) is the content of a[i].

There is one difference between an array name and a pointer acting as an array name. A pointer is a variable, so pa = a can be written, but an array name is not a variable, and entries like a = pa are not allowed. Consider the most common properties of an array as an abstract data type:

• All array elements belong to the same type (homogeneity);

• The size of the array is set once and does not change during the operation (persistence);

• All array elements have the same access (equity);

• All elements are arranged sequentially in RAM cells (location sequence);

• Array elements are uniquely identified by their indices (indexing);

• Indexes must be a simple ordinal data type.

It should be noted that any abstract data type consists of a set of values and a set of methods. For an array, the main methods are:

• Get element with number N;

• Record element with number N;

• Get the size of an array.

The advantages of ATD «array» include:

* Arrays store multiple data types with the same name.
* It provides arbitrary access to elements.
* Since the array is fixed size and stored in adjacent memory regions, there is no memory shortage or overflow.
* It is useful to store any type of data with a fixed size.
* Because elements in an array are stored in adjacent areas of memory, it is easy to iterate in this data structure, and access to the element, if known, takes a unit of time.

Limitations of array data structure include:

* The size of the array must be known in advance.
* An array is a static data structure with a fixed size, so the array size cannot be changed additionally, and therefore no changes can be made during execution.
* Insertion and deletion operations are expensive in arrays because elements are stored in continuous memory.
* If the size of the advertised array exceeds the required size, this can result in memory loss.

This disadvantage can be overcome by introducing an abstract data type such like *slice*. S*lice* is a flexible and variable data structure consisting of several elements of the same type. Like arrays, the slice is indexed and has a size (length) that can be programmatically regulated (increasing and decreasing) by having such a property like *capacity*. The slice capacity is the number of elements in the base array, counting from the first element in a slice. If a slice has sufficient capacity, then its length can be increased by it re-slicing. The slice may consist of three elements: *a pointer* indicating the first slice element, a length (slice element number) and a capacity - the maximum size to which it can be extended (Figure 1.3.):

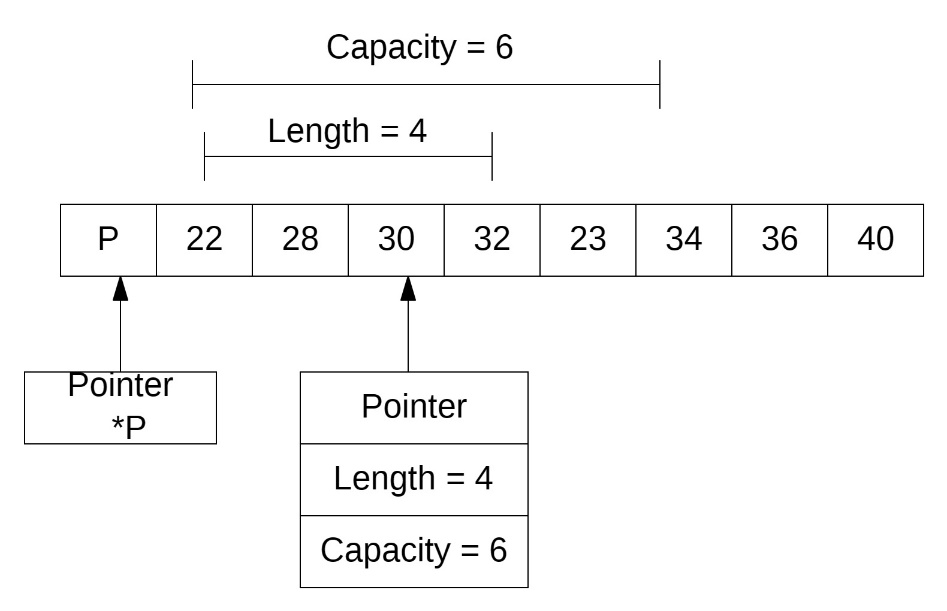


Figure 1.3. Slice and it components

The main methods for slice are the operations of adding, removing, determining a slice length, changing a capacitance. The program implementation of Golang slice (declaration, initialization, basic methods of working with this type of data) is described in more detail in the next section.

**1.2.2. Linked List**

A list is an ordered set consisting of a variable number of elements to which inclusion and deletion operations apply. A list reflecting the neighbourhood between elements is called linear. The list length is equal to the number of elements in the list. If the component is not related to any other, the index field contains a value that does not indicate any element. This link is indicated by a special name - null.

There are three common types of connected types:

* Singly linked list;
* Doubly Linked list;
* Circular Linked List.

**a) Singly linked list**

Figure 1.4. shows the structure of the simply connected list, where the field *item* is containing *data*, the next one - *pointer* to the next element of the list. Each list must have a special element called the top of the list or head of the list, which is usually different in format from the rest of the items. In the index field of the last element of the list there is special feature null indicating the end of the list (Figure 1.4):

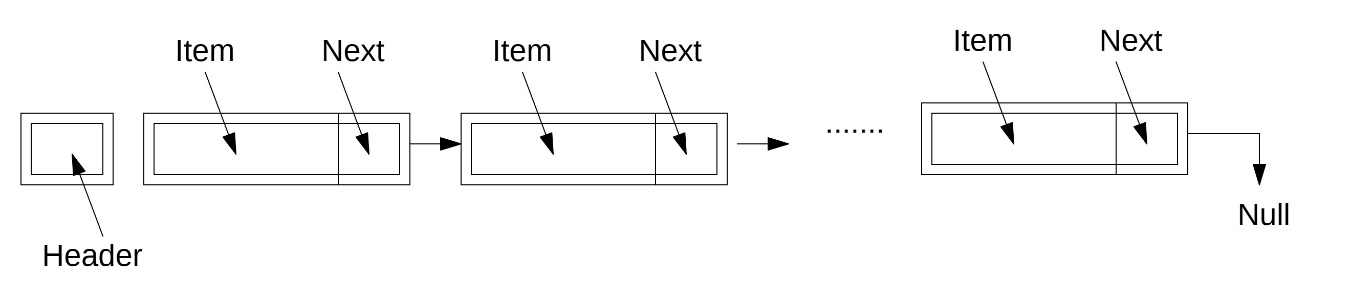


Figure 1.4. Single-linked list (item – data; next - *pointer* to the next element)

Single-linked list, thus defined, has a number of properties:

* The size of a list is the number of elements in it, excluding the last "zero" element, which is by definition an empty list.
* The element type is the type on which the list is built; all items in the list must be of this type.
* Sorting - the list can be sorted according to some sort criteria (for example, by increasing integer values if the list consists of integers).
* Access capabilities - some lists, depending on the implementation, can provide the programmer with selectors to access directly to a given number.
* Comparability - lists can be compared against each other for consistency, and depending on the implementation, the list comparison operation can use different technologies.

Below is some set of functions that can be performed over the ATD list:

• Initialize the list;

• Insert a new element;

• Remove any element;

• Find the k-th element;

• Read the next element;

• Printing elements;

**b) Doubly linked list**

In some cases, it is necessary to include links on both sides. In this case, both direct and reverse access is possible. It is defined as a sequence of nodes, where each node defines a region of memory. Each node is divided into three fields: *item* is the data field, *next* is the address of the next field, or forward field, and the previous, or reverse field. The data field is used to store data values. The *prev* field is used to store the address of the previous element, *next* is used to store the address of the next element. The first node of the previous field and the last node of the next field are always zero (Figure 1.5.).

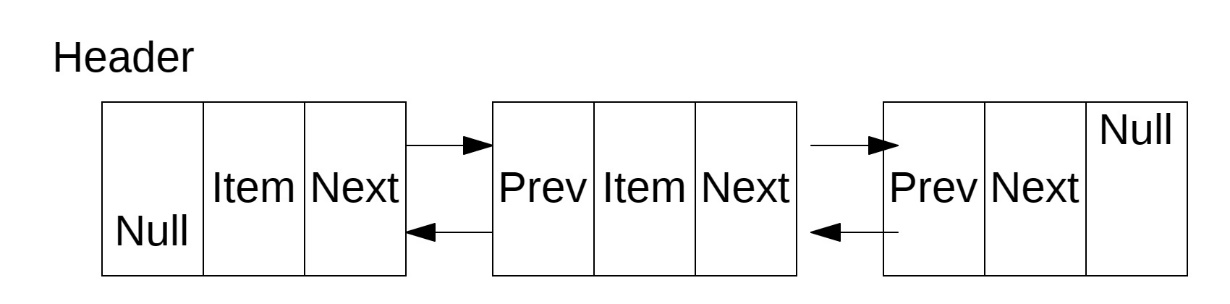


Figure 1.5. Doubly linked list

The properties and functions for the biconnected list are almost the same as for the simply connected list. The main advantage of the two-linked list is that it simplifies many operations; the main drawback is the extra memory required for address pointers.

**c) Circular linked list**

A variation of the types of linear lists considered is the circular list, which can be organized on the basis of both simple and two-linked lists. In this case, in the simply connected list, the pointer of the last element should point to the first element; in the two-linked list in the first and last element, the corresponding pointers are overridden, as shown in Figure 1.6.

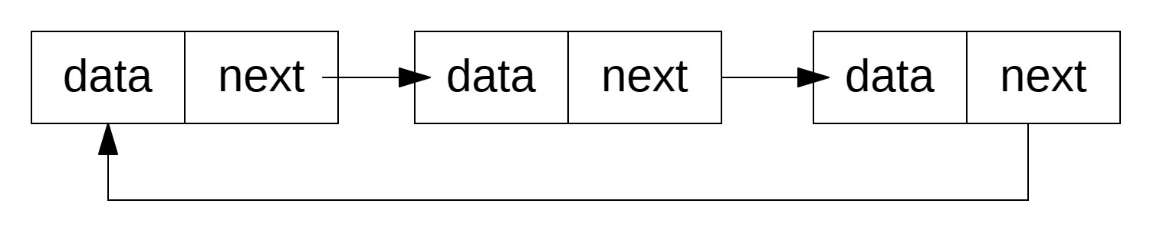


Figure 1.6. Circular list structure

When working with ring lists, some procedures are also simplified; however, when viewing such a list, you should take some precautions not to get into the infinite loop [www.algolistmanual.list].

Some procedures are simplified when dealing with such lists. However, when viewing such a list, you should take certain precautions not to get into an endless cycle.

* + 1. **Stack and Queue**

The stack is a dynamic, ever-changing set of data, the addition and deletion of which are performed solely on the principle of "last logged in, first went out" (Last-In-First-Out or LIFO) (Figure 1.7). The stack works like manipulating a stack of books, which are shaped by actions to delete and add books.

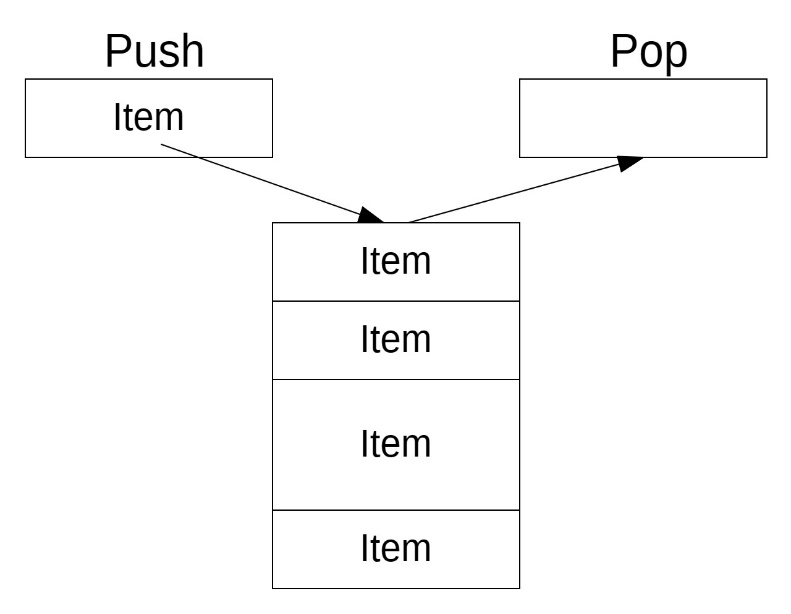


Figure 1.7. Stack structure

Stacks are widely used in various data processing scenarios, such as implementing "undo" in text editors; this operation is performed by storing all text changes to the stack. The stack is often used to organize methods calls and recursions.

The main operations that can be performed on the stack are:

• Add new data;

• Remove data from the stack;

• Return (without removing) the top value;

• Check stack full status;

In many programming languages, the stack is created either from an array or from a single-linked list. The Golang language can apply the concept of interfaces, which will be discussed further.

Unlike the ATD stack, which has much in common with the stack, elements are added (enqueue) and removed (dequeue) from different ends. This method is called "first entered, first released" (First-In-First-Out or FIFO). That is, elements are taken from the queue in the same order in which they are placed (Figure 1.8).

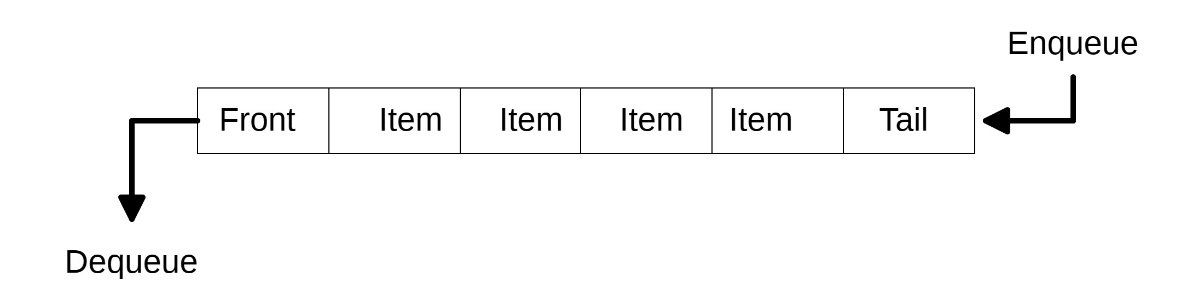


Figure 1.8. Data structure “Queue”

The following functions are used to manipulate queue components.

• Create a new empty queue;

• Add a new element to the end of the queue;

• Remove the front element from the queue;

• Check the queue for emptiness;

• Check the queue for filling;

• Determine the length (number of elements) in the queue.

The data type of the queue can have several varieties:

A *circular queue* is an extended version of a regular queue in which the last element is connected to the first element, resulting in a circular structure. Circular queue solves the main constraint of regular queue - inefficient memory usage.

*A priority queue* is a special type of queue in which each element is associated with a priority value. And the elements are serviced on the basis of their priority. That is, the higher priority elements are served first. However, if items with the same priority meet, they are served according to their order in the queue. In a normal queue, the "first in first out" rule is implemented, while in a priority queue, the values are removed based on priority. The highest priority element is removed first [Queue as data structure, types, implementation, application,... (intellect.icu)].

Queued data storage is widely used to plan CPU, printer, and other output tasks. The FIFO method is used when processing interrupts.

* 1. **Nonlinear abstract data types**

This subsection considers nonlinear data structures, which include the most common structures, namely *trees and graphs*. Recall that nonlinear data structures allow the expression of more complex relationships between elements than linear neighbourhood relationships, as in linear structures.

**1.3.2.** **Trees as ATD.** In the abstract representation, trees are understood as non-linear data structures consisting of nodes representing a hierarchy of relationships (parent-children). Each vertex of the tree, if there are descendants, is connected to them by direct edges (Figure 1.9).

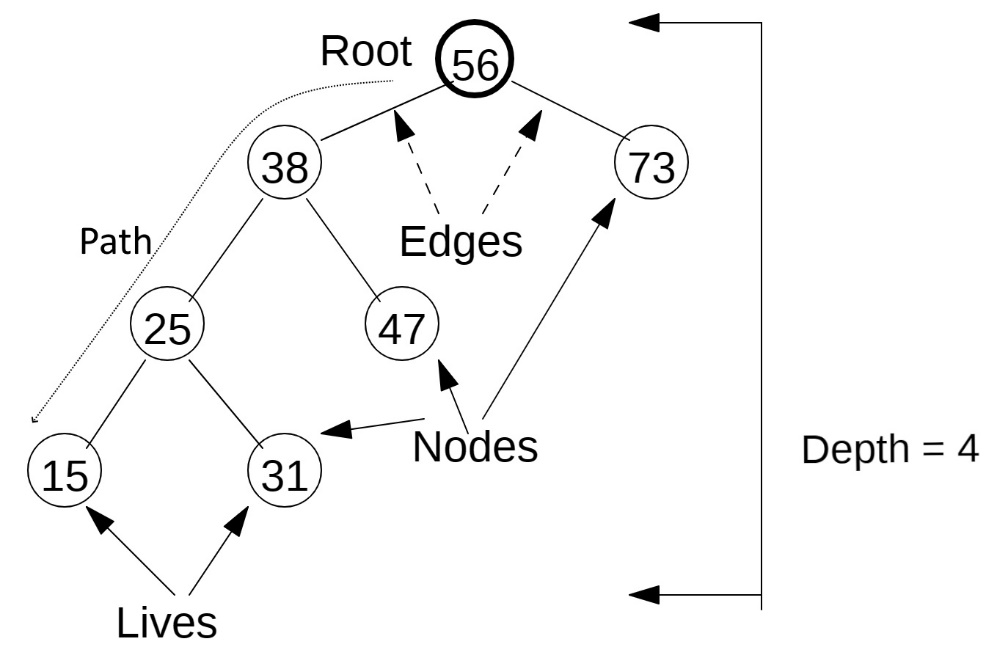
****

Figure 1.9. Basic tree terminology

Let’s give a definition of the basic concepts of the theory of trees.

*Root*: The root of the tree is the only node without any incoming edges. This is the top node of the tree;

*Node*: This is the fundamental element of the tree. Each node has data and two links that can point to null or its descendants;

*Edge:* This is also the fundamental part of the tree that is used to connect two nodes.

*Path:* A path is an ordered list of nodes connected by edges.

*Leaf:* A leaf knot is a knot that has no descendants.

*Tree height:* Tree height is the number of edges on the longest path between the root and the leaf.

*Node level:* Node level - number of edges on the path from the root node of this node*.*

Several computational tasks require the organization of data in different types of trees, but binary trees (binary tree) [DSA] are particularly well studied and common. Within the concept of abstract data types, a binary tree is a set of connected nodes in which each node contains a value (the data element itself) and has no more than two children. This means that the power of the binary tree is zero or one or two. From the ATD perspective, binary trees contain values of elements of type T. The load-bearing set of this type is the set of all binary trees whose vertices are of type T. The carrying set thus includes an empty tree, trees only with root of type T, trees with root and left child, trees with root and right daughters, and so on.

To the degree of the vertices, binary trees are: strict - the vertices of the tree have a degree of zero (in the leaves) or two (in the nodes); weaker - the vertices of the tree have a degree of zero (in the leaves), one or two (in the nodes). In general, a k-level binary tree can have up to 2k-1 vertices. A binary tree containing only fully populated levels (that is, 2k-1 vertices on each k level) is called complete. Binary tree species are shown in Figure 1.10.

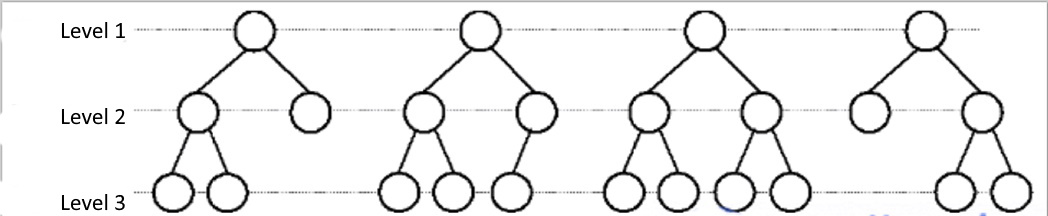


Figure1.10. Binary tree species

Common operations for binary trees as abstract data types are:

* Insert *k*-th element;
* Remove *k*-th element;
* Search for an element with the specified value *k*;
* Find the maximum value stored in the tree;
* Find the minimum value stored in the tree.
* Find the number of tree levels.

The implementation of basic algorithms for binary search trees is presented in Section 8.

**1.3.2. Graphs like ATD.** In the most general definition, a graph G is given by a set of vertices {V} and a set of edges {E} connecting all or part of these vertices. In other words, a graph G is completely defined by a pair of{V, E}. If edges are oriented, which is usually shown by an arrow, then they are called arcs, and the graph with such edges is called a directed (directed) graph (Figure 1.11 a). If edges have no orientation, then the graph is called non-directed (undirected) (Figure 1.11 b):

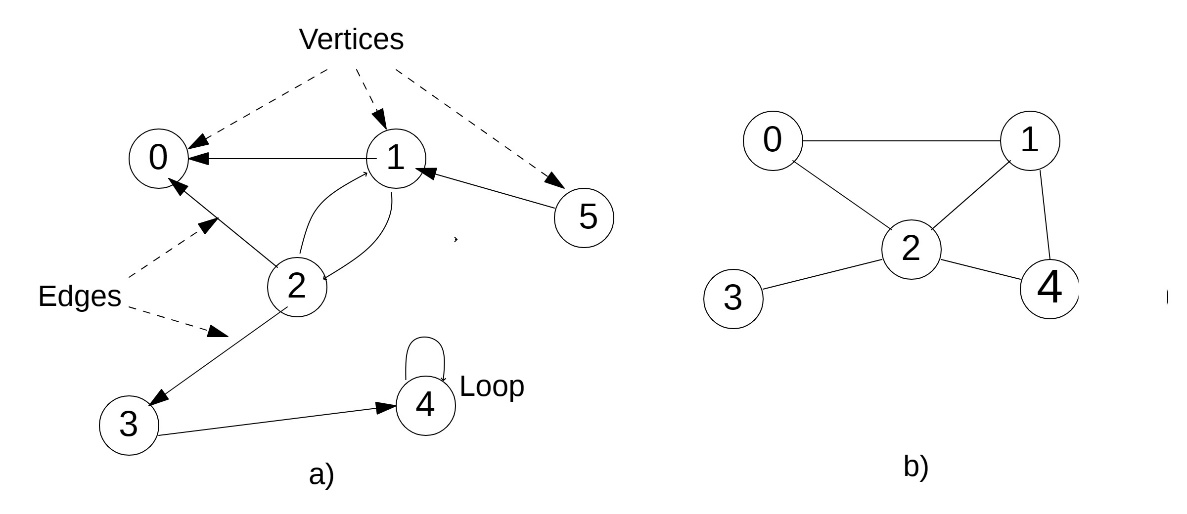
****

Figure 1.11. View of graph a) directed; b) non-directed

Vertices and edges are called graph elements, the number of vertices in the graph is an order, the number of edges is the size of the graph. The vertices (u,v) are called the endpoints of the e = {u,v}, and the two endpoints of the same edge are called adjacent. Two edges are called adjacent if they have a common endpoint. Two edges are called multiples if the sets of their endpoints are the same. An edge is called a loop if its ends coincide, that is, e = {u,u}. If the vertex vi is the beginning or end of the edge ek, then they (vertex and edge) are incident. The number of edges incident to a vertex is called the vertex degree (Figure 1.12).

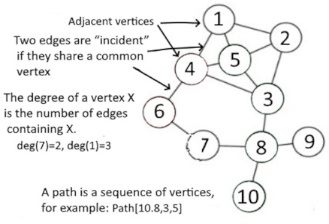


Figure 1.12. Basic graph parameters

There are many types of graphs, among which the most significant are:

* a zero graph in which there are no edges;
* a trivial graph with only one vertex;
* a non-directed graph in which the edges have no direction;
* a directional graph in which the edges have directions;
* a complete graph in which each node has an edge to another node;
* The weighted graph in which the edges are specified with the weight.

Accordingly, there are many operations with graph elements, in particular:

▪ add an edge between two vertices;

▪ operation to remove an edge while maintaining all vertices of the graph;

▪ operation of adding a vertex to a set of vertices;

▪ operation of adding this rib to the set of ribs;

▪ Dijkstra operation, which determines the minimum distance between two given nodes.

More detailed graph algorithms are discussed in SECTION 9.

...