# 2 Linear structures

## 2.1 linear list, list

Array

The table is a simple linear data structure in which the tables are arranged sequentially in a row. Typically, a space is reserved for the table from a memory space, i.e., the table does not consist of separate nodes such as dynamic tree structures.

The table is usually ready for basic information structure in high-level programming languages, so its use is easy for programmers.

BUT

The Python programming language has no array (array) in its original meaning. Instead, Python has a List type to simulate tables. List is an abstract data type, so its exact implementation is not exactly defined (in principle, implementation of different Python languages ​​could be implemented in different ways). This causes problems in the algorithm performance analysis because it can not be safely said in the general case whether references to the List-type table can be referred to within a standard time. This course can be used to assume that, if spoken from a table, indexing takes place in standard time even if the corresponding example is in Python. Python is available as libraries such as nympy, where the table is implemented efficiently.

**The basic features of the table are:**

All the places can be directly referred to by the table index (array index).

Depending on the programming language, the indexes in the N-seated table are numbered (see place number) fluently 0 ... N-1 (eg C and Java).

It is essential that the locations are indexed from the first to the last, and that the value range is numbered and can thus be returned to an integer (a 10-seater table could also be indexed e.g. A, B, C, D ..., J).

The order of the places in the table is fixed. It can not be changed (see the chairs' seats can not be changed in the theater), although the values ​​of the items in the place (which value is in a certain place or who is sitting in the chair) can be changed. Thus, the table is a variable data structure, although it is possible to create unchanged tables in some programming languages.

It is often thought that a set of embryos (payloads) has been stored in the table. An individual embryo (record) can consist of several different types of fields, that is, it can be an arbitrarily complicated data structure, although in the textbooks the actual payload is often abstractioned and the embryo is tagged eg in the images with the single key of the record (eg integer).

It is important to understand what the difference is in the table element's index and the value of the table element. A specific element (index) can be referred to as a constant time because the item's address in the memory can be easily calculated by knowing

1.table address in memory,

2.the size of an item (how much one item takes memory) and

3.embryo index.

**Two-dimensional table metaphor: seating in theater**

If you know the number of your own place, you can go (in the case of the table) to your own place (by place and row number). But: if you do not know where your friend is sitting, you have to go through the places one by one, so you can not just guess where he is sitting to see him. In the worst case, your friend did not come to the theater and you had to go through all the places to see this (unorganized table). How will the situation change if you know that the places are shared to the public by alphabetical order (organized table)?

There are a number of limitations in the table that can make it use as a root:

**The size of the structure is determined when it is created and the entire space is reserved before it is actually needed. Thus, the table is a static data structure.** Some programming languages ​​have specific types of information that resemble the table and where its size can be increased later, but in practice it means that the whole table must be copied to the new memory area (takes a momentarily time) or the table is implemented as a tree structure and actually consists of several tables. This is a dynamic table.

Because of the fixed order of the embryos, no new embryos can be inserted between the embryos, and no intermediate can be removed from the embryos.

**Scala Vector**

The Scala programming language has a Vector data structure that is a dynamic table. Another of its specialty is that it is immutable. Its implementation has a bit of the same features as in later B-trees (see the balanced search pages).

**linked list**

Linked list is in many respects a more flexible linear data structure than a table, but this flexibility also has its back side. The linked list consists of individual locations, each with a stored item, plus a pointer (link) to a linked list to the next location. Thus, the places can be sequenced sequentially to the extent permitted by any available memory.

Linked list metaphor

It is more difficult for a linked list to come up with a good metaphor: it is a kind of chain that is easy to break anywhere and add a new place (updating links so that the chain does not break). Places in a chain, i.e. a particular breakpoint, can only be found by leaving the first place in the chain and following links.

Linked list

In the figure, the points of the chain, ie, the list nodes, represent rectangles with payload (numeric values modeling embryos) and a separate field with the pointer (link, arrow in the image) to the next location. The lists are also handled with the help of pointers: someone has to indicate where is the first element of the list (here's the pointer p). The cursor p is therefore not a joke but just a reference to a place. In order to facilitate the processing of the lists, the pointers are usually defined more (here the indicators q and r). The last point in the list shows the "empty position" (null) to indicate that the list ends. Instead of a vacant position, a separate concrete slot (z) indicating the end of the list is often used to return later.

The following is an example of how Python could define a class (ListNode) that matches a single linked list node.

**class** **ListNode**:

**def** **\_\_init\_\_**(self, value, previous=None, next=None):

self.val = value

self.next = next

self.previous = previous

By changing the values of the pointer, a linked list can be added anywhere and can be removed. In addition, the place of embryos can be exchanged within a linked list by swapping places with fairly simple cursor swapping. Naturally, the contents of the places (payload stored in the list, ie embryos) can be changed or exchanged with each other.

通过更改指针的值，可以在任何位置添加链接列表，并可以将其删除。 此外，通过用相当简单的光标交换交换位置，可以在链表中交换胚胎的位置。 自然地，可以改变或交换地方的内容（存储在列表中的有效载荷，即胚胎）。

**Add to Linked List**

An addition can be made easily if we know which element (pointer q) the new element (r) becomes.

**def** **add**(value, after):

r = ListNode(value)

r.next = after.next

after.next = r

add(**99**, q)

The biggest limitation in the linked list is that the embryos can only be passed in succession. That is, to handle a specific item, such as an embedded list item

k

k from the beginning, it has to search it by visiting the linked list through the platform until then. Note that in the table

k

The position of the k: n element can be calculated within the standard time but not in the linked list. This is of great importance when analyzing algorithms utilizing linear data structures.

链表中最大的限制是胚胎只能连续传递。 也就是说，来处理一个特定的项目，比如一个嵌入的列表项目ķ从一开始，它必须通过平台访问链表进行搜索，直到此为止。 请注意在表中k：n元素的位置可以在标准时间内计算，但不能在链表中计算。 当使用线性数据结构分析算法时，这是非常重要的。

Removing a Linked List

Consider, for example, the removal of an element (r) from a linked list. The item you want to delete must first have to retrieve from the list, which may even refer to the whole list from the beginning. In fact, during the search you should keep a record of the predecessor (q) of the embryo to be deleted. When the predecessor of an item to be removed is found, removal from the chain is easy.

Deleted item after successor

**def** **delete\_after**(after):

r = after.next

after.next = r.next

delete\_after(q)

If the embryo also has a link to the previous element, it is a double-linked list. In this case, when searching for the item to be deleted, it is not necessary to keep a record of the predecessor separately, because the information from the predecessor is also stored in the item to be deleted.

To facilitate the handling of a linked list, links to both the first and the last item are often stored in the auxiliary variables. In this case, the last embryo is found in constant time. These auxiliary variables may be located in a special header element, which is a list element and allows simplification of the implementation (the actual first element of the list can be treated just like any other element). For example, information on the length of a linked list can be maintained in the header subdirectory.

**Head and z-nodes**

Thanks to the header element, the first embryo can also be deleted at the same time as the code snippet. For the same reason (avoidance of special cases), at times, an extra element z is referred to as a reference to itself.

**Checking the end of a list:**

**if (q == q.next) ... OR if (q == z) ...**

The linked list is used in a variety of applications for temporary storage of information, as it is very simple to implement. It is suitable for situations where the amount of data to be stored can vary (you can not reserve a suitable whole table), you need to add or remove embryos from the middle of the material, or the items are sequenced sequentially. It should be noted, however, that consecutive viewing is a very ineffective search method, so at a slightly higher level (e.g.N> 1000), you should use some more powerful search structure, such as a tree structure. In them, addition and removal of embryos can also be performed efficiently.

Data type List refers to a data structure that linked list or array.

Python, lists is a dynamic array.

The table and the list thus have different characteristics, and in particular the performance of the operations involved there are significant differences. For example, adding the Python's List structure to the beginning is a laborious operation (all items must be moved to the right). However, the embryos can be crawled on a regular basis (therefore, it can be easily explored which is the value of the n element). In a linked list, it is not possible to index an item, but the embryo should be searched from the beginning until it comes to the correct item. On the other hand, both the beginning and the end of the linked list would be easy (a record can be kept both at the beginning and the end of the list, so adding a new element is a so-called standard-time operation).

2.2 Stack and Queue

Stack

Stack is an abstract data type. It describes a linear data structure that can handle only one end of one end. The following operations are defined for the stack, all of which can be performed in a fixed time. Ts. the stack size (the number of embedded stacks) does not affect the execution time of the operations.

Push (x)

Put the item x on the stack.

Pop

Returns and removes the top element of the stack.

Top

Returns the top of the stack without removing it.

Empty

Returns the information on whether the stack is empty.

The stack therefore operates on the principle 'last in first out' (LIFO).

The stack can be implemented in a number of ways, for example, by a table or a linked list. In Table Execution, a maximum number of embryos is reserved for the stack, and a stack-pointer variable is referred to the top element of the stack. Its value increases by one when an element is added to the stack and vice versa. If all the elements in the stack have data, then the stack is full and a new item can not be added. In a linked list-based implementation, the stack pointer always points to the first element in the list and its value is changed with additions and deletions.

The stack is used in a variety of applications where data must be processed in a different order than the one where it is accessed. For example, a pin can reverse an input bin or use it to calculate the value of the expressions (the intermediate results of the calculation are stored in the stack).

Application programs do not modify the link list, for example, but it is a habit of implementing Pino-abstraction as its own data structure. This makes the writing of the application program a bit smoother and flawless. Take the next stack using the linked list you previously made. You will benefit from using your stack implementation in appendices of the next chapter, so you should be careful to test its functionality.

**Queue**

The queue is an abstract data type for which the following operations are defined:

Put (x) or Enqueue (x)

Add element x to the end of the queue.

Get or Deque

Returns and deletes the first item in the queue.

First

Returns the first item in the queue without deleting it.

Empty

Returns the information on whether the queue is empty.

The queue works on the principle 'first in first out' (FIFO). All queue operations can be implemented within a standard time.

The queue can be implemented eg by a table or a linked list. There may be two pointers in the table, one in the first and the last in the queue. In the case of additions and deletions, the value of the indicators is increased by one and when the pointer enters the table at the end of the table, it is spun at the beginning of the table. It is a so-called. circular table. As with the table-based stack, the circular table may become full when all the items have information.

The queue is useful, for example, in applications where the information to be processed is buffered into storage prior to processing. The embryos are stored in a queue as they come and are dealt with at one end of the queue. In this case, the items will be processed in the same order as they originally entered.

**Deque**

Packet is an abstract data type that can be considered a sort of combination of stack and queue. It has defined the following operations:

PutFirst (x)

Add element x to the top of the deck.

PutLast (x)

Add element x to the bottom of the pack.

GetFirst

Returns and removes the bulk element of the pack.

GetLast

Returns and removes the bottom element of the deck.

First

Returns the bulk element of the slave without removing it.

Last

Returns the lowest element of the deck without removing it.

Empty

Returns the information on whether the package is empty.

In the pack, the operations can then be directed at one of its ends. Operations can be implemented within a standard time.

Note. The names of the operations can vary in literature depending on the author, even though the functionality is the same. There may also be limited versions of the snack, which can only be added to or removed from the other end, not both.

There is therefore a wide variety of applications available to the stack. There were just a few of them above. The stack is therefore a strong abstraction based on the implementation of a linear data structure. Instead of directly using a linked list in programming, the stack is often customized as its own data structure. Such an abstraction raising is useful to make the actual application (algorithm) cleaner and avoid programming errors. Take advantage of the pile you previously carried out in the following tasks.