Lesson: More on Data Structures Pillar: Data Structures

So far, you have been introduced to various elementary and high level data structures: arrays, linked lists, stacks, queues, and binary trees. Specific to Python, you have used lists (similar to arrays) in your programs. In this lesson, we will discuss some powerful functions that work with lists and introduce several new data structures.

## Useful list functions

As you have seen, Python lists are extremely useful data structures. In the first Python lesson, you were introduced to several list functions that, for example, reverse a list, sort a list, etc. In this lesson, we will cover several more powerful built-in functions that are quite useful when used with lists.

The `filter` function returns an iterator that contains a new list consisting of only the items within an existing list for which some user-defined function is true. The user-defined function can be anything that evaluates an input in the existing list and returns true or false. The format for the `filter` function is as follows:

```
filter(function, mylist)
```

The parameter `function` represents the name of the function that will evaluate each item in the existing list. The parameter `mylist` is, of course, the existing list of items to evaluate.

An "iterator" is an object that contains a list. We can use iterators to operate on a list of values. To convert this iterator into a list object, that would expose the list it contains, we should wrap the call to filter with a list constructor:

```
list(filter(function, mylist))
```

This converts the returned iterator into a list that we can print directly.

Suppose, for example, that you want to find all of the multiples of three or five that are less than or equal to 30 and make a list of them. Here's one way to do this:

```
multiples = []

for i in range(3, 31):
    if (i % 3 == 0 or i % 5 == 0):
        multiples.append(i)
```

However, here's how it could be done with the `filter` function:

```
def f(x):
    return (x % 3 == 0 or x % 5 == 0)

multiples = list(filter(f, range(3, 31)))
```

Both of these methods generate the following list:

```
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24, 25, 27, 30]
```

The `map` function returns an iterator that contains a new list consisting of the return values generated by a user-defined function on each item in an existing list. The user-defined function is called for each item in the existing list; the return values form the new list. The format for the `map` function is as follows:

```
map(function, mylist)
```

Like the filter function, we need to wrap the call to map inside of a list constructor call:

```
list(map(function, mylist))
```

The parameters are the same as specified for the `filter` function. Suppose, for example, that you want to square each item in a list. Here's one way to do this:

```
squares = list(range(1, 10))

for i in range(len(squares)):
    squares[i] *= squares[i]
```

Although the snippet of code above does modify the existing list, it could be easily changed if needed. Here's how it could be done with the `map` function:

```
def f(x):
    return x * x

squares = list(map(f, range(1, 10)))
```

Both methods produce the following list:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Lastly, the `reduce` function processes the elements in a list through a user-defined function and returns a *single* value. The function works by first processing the first two items of the list. The result of this is processed by the function, along with the next item in the list. This continues for all of the remaining items in the list. The format for the `reduce` function is as follows:

```
reduce(function, list)
```

Again, the parameters are the same as specified for the previous functions. Suppose, for example, that you want to compute the factorial of 10. Here's one way to do this:

```
fact = 1

for i in range(1, 11):
    fact *= i
```

Finally, here's how to do it with the `reduce` function. Note that the `reduce` function comes from the `functools` module that is part of Python. This means we need to import the function from that module before we can use it:

```
from functools import reduce

def f(x, y):
    return x * y
    fact = reduce(f, range(1, 11))
```

Both methods calculate the factorial of 10 (which is 3628800).
The following table summarizes the list functions discussed above:

| **Function** | **Purpose** | **Syntax** | **Returns** |
|---|---|---|---|
| `filter` | Select list elements using a function | `filter(function, list)` | iterator |
| `map` | Apply a function to every list element | `map(function, list)` | iterator |
| `reduce` | Reduce a list to a single value using a function | `reduce(function, list)` | value |

**List comprehensions**
Consider the simple problem of creating a list of the cubes of the integers 0, 1, 2, etc, up to 9 (i.e., 0, 1, 8, 27, 64, …, 729). Try to write a snippet of Python code to accomplish this in the space below:

Another way uses a concept known as list comprehensions. A **list comprehension** provides a simple, concise way of creating lists (even complex ones). The most common use of this concept creates a list where each element is the result of some operation or expression applied to each element of another list. Here's an example that does the same thing as the snippet of code above:

```
    cubes = [x * x * x for x in range(10)]
```

Yes, it's a single statement! A list comprehension uses the for loop to generate or to iterate through the items of a sequence and applies some operation or expression to each of those items. In the statement above, the generated sequence is the range of values from 0 through 9. The expression that is applied to each of the elements in the generated sequence is `x * x * x` (i.e., it cubes each element). The result is a new list of the cubes of the elements in the generated sequence (0 through 9):

```
    [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

The list comprehension can be read in English as, "the variable cubes is to be a list that contains some *x* cubed for each *x* in the range 0 through 9." In fact, we can map the English version to the Python statement:

| | | | |
|---|---|---|---|
| the variable cubes is to be | a list that contains | some value *x* cubed | for each *x* in the range 0 through 9 |
| `cubes =` | `[` | `x * x * x` | `for x in range(10)]` |

Minimally, a list comprehension consists of brackets containing an expression (e.g., `x * x * x`) followed by a for-loop. Additional for-loops or even if-statements can be chained after the first for-loop. The resulting list is an evaluation of the expression in the context of the for-loops and if-statements that follow it. Here's a seemingly convoluted example:

```
    sums = [x + y for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
```

The output of this statement is:

```
[4, 5, 5, 3, 6, 4, 7]
```

This statement generates a list of the sums of the pairs that can be formed by combining a single element from the first list (`[1, 2, 3]`) with a single element from the second list (`[3, 1, 4]`), **so long as the elements differ** (i.e., `if x != y`). The elements of each list are processed from left-to-right. The first sum is calculated by adding the first element of the first list to the first element of the second list (1 + 3 = 4). The second sum is calculated by adding the first element of the first list to the third element of the second list (1 + 4 = 5). Why the third element and not the second? Because this would mean that both elements have the same value (1). The if statement ensures that sums will only be produced if the list elements differ.

Here's another (similar) example of list comprehension:

```
pairs = [[x, y] for x in [1, 2, 3] for y in [3, 1, 4] if x < y]
```

The output of this statement is:

```
[[1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
```

This statement generates a list of the pairs that can be formed by combining a single element from the first list (`[1, 2, 3]`) with a single element from the second list (`[3, 1, 4]`), **so long as the element from the first list is less than the element from the second list**. Again, the list elements are processed from left-to-right. It is similar to the previous statement; however, instead of generating a list of sums, a list of pairs of elements is generated. In fact, it's a list of lists! The enclosed sublists are composed of only two elements each.

Really cool things can be done with list comprehension. For example, here's a neat way to compute pi to various precisions from two through eight digits:

```
from math import pi

pi_digits = [round(pi, n) for n in range(2, 9)]
```

And here's the output:

```
[3.14, 3.142, 3.1416, 3.14159, 3.141593, 3.1415927, 3.14159265]
```

The function `round` does what you expect it to do: it *rounds* the value specified in the first parameter to a precision specified in the second parameter. For example, the expression `round(pi, 3)` rounds pi to three digits to the right of the decimal point (i.e., 3.142).

**Sets**

In Python, a set is just another type of sequence. The mathematical definition of a set is an unordered collection of unique elements. That is, a set is basically just a list with no duplicates.

**Definition**: *A* ***set*** *is a type of Python sequence that contains an unordered collection of unique values.*

Since sets are mathematically defined, they support mathematical operations such as union, intersection, and difference. Defining a set can be done in several ways. The first is to formally define one, very much like you would define a list. However, instead of using square brackets, we use curly braces:

```
a = { 1, 2, 3, 4, 5 }
b = { 3, 4, 5, 6, 7 }
```

These Python statements declare and initialize two sets, *a* and *b*. Another way to create a set is to do so from some other sequence (such as a list):

```
c = [ 3, 1, 4, 1, 5, 9 ]
d = set(c)
```

The list *c* is used as input to create the set *d*. The set *d* will only have one instance of any duplicated element in the list *c*; therefore, the value of the set *d* is as follows:

```
{1, 3, 4, 5, 9}
```

Note that the element 1 appears only once in the set *d*. In addition, the set is unordered; that is, its elements don't necessarily have to be in the same order as those in the list.

A set can even be created from a string (since a string is a sequence):

```
e = set("sweet")
```

The value of the set *e* is as follows:

```
{'s', 't', 'w', 'e'}
```

Again, note that the elements of this set are unique. To illustrate the set operations (union, intersection, and difference), we will use the sets *a* and *b* defined above. The **union** of the sets *a* and *b* represents the elements **in either *a* or *b***. The Python expression for this is written as `a | b`. Its output is the following set:

```
{1, 2, 3, 4, 5, 6, 7}
```

These elements are all of the unique elements in *a* or *b*. Similarly, the **intersection** of the sets *a* and *b* represents the elements **in both *a* and *b***. The Python expression for this is written as `a & b`. Its output is the following set:

```
{3, 4, 5}
```

These elements are the only unique elements in both *a* and *b*. Lastly, the **difference** of the sets *a* and *b* represents the elements **in *a* but not in *b***. The Python expression for this is written as `a - b`. Its output is the following set:

```
{1, 2}
```

The difference operation can be thought of as a subtraction of the set *b* from the set *a*. All elements in both *a* and *b* are removed. The remaining elements in a make up the resulting set. Any additional elements in *b* that are not in *a* are ignored.

## Dictionaries

A dictionary is perhaps one of the most powerful data structures at our disposal in Python. As you have seen, sequences (like lists) are indexed by a range of numbers (i.e., the first element is placed at index 0, the second element is placed at index 1, and so on).

**Definition**: *A **dictionary** is a data structure whose elements are indexed by unique keys. A **key** is just an unchangeable value. The elements are known as **values**, and are associated with the keys. That is, a single key maps to a single value. This is why we often say that dictionaries contain key-value pairs.*

Technically, a list pairs an index (which could be called a key) with a value (the element at that index). The difference is that dictionaries permit keys to be of almost any type, so long as a key is not susceptible to change. That is, it must be **immutable**. For example, the integer 5, the floating point number 3.14, the string "Jones", and the character '%' are all valid keys. Note that all keys in a single dictionary must be unique (i.e., there can be no duplicate keys).

Consider a dictionary that you are familiar with: the kind that you lookup the definitions of words in. Using such a dictionary typically involves searching for some word in order to obtain its definition. In such a dictionary, the word is the key, and its definition is the value associated with that key. You should have noticed that, to search a dictionary, the key is required. The unknown is the value that is associated with the key (a definition). In some programming languages, this type of data structure is known as an **associative array**.

Another dictionary data structure that you are probably familiar with is a phone book (although you've probably only used some online version and not an actual book). What are the keys in a phone book? What about the values? Clearly, a name is the key (e.g., Bob Jones). The values associated with the keys are records that contain an address and a phone number. Certainly, such records can be represented as long strings (perhaps even with newlines). But we may also wish to represent the records as objects of some **PersonInfo** class!

Dictionaries are created similarly to sets (using braces). The difference is that key-value pairs are specified in the format `key: value`. Here's an example of a dictionary with strings representing last names as the keys and integers representing office numbers as the values:

```
    offices = { "Jones": 247, "Smith": 121, "Kennedy": 108 }
```

This creates a dictionary with the following key-value pairs (in no particular order):

| Last name | Office number |
|---|---|
| Jones | 247 |
| Smith | 121 |
| Kennedy | 108 |

The main operations associated with a dictionary are to store some key-value pair and to retrieve a value associated with a key. Adding the new key-value pair `"Wilkerson": 355`, for example, can be added to the dictionary above as follows:

```
    offices["Wilkerson"] = 355
```

The dictionary now has the following key-value pairs (in no particular order):

| Last name | Office number |
|---|---|
| Jones | 247 |
| Smith | 121 |
| Kennedy | 108 |
| Wilkerson | 355 |

Retrieving a value matching the key "Smith", for example, can be done as follows:

```
loc = offices["Smith"]
```

The value of the variable `loc` is therefore 121. Note that attempting to retrieve a value using a key that is not in the dictionary results in an error.

An existing key-value pair in the dictionary may be overwritten by simply inserting a new value with the same key. For example, suppose that Kennedy changed offices (to, say, 111). The dictionary can be updated as follows:

```
offices["Kennedy"] = 111
```

The dictionary now has the following key-value pairs (in no particular order):

| Last name | Office number |
|---|---|
| Jones | 247 |
| Smith | 121 |
| Kennedy | 111 |
| Wilkerson | 355 |

A key-value pair can be removed from the dictionary using the `del` keyword as follows:

```
del offices["Smith"]
```

The dictionary now has the following key-value pairs (in no particular order):

| Last name | Office number |
|---|---|
| Jones | 247 |
| Kennedy | 111 |
| Wilkerson | 355 |

Determining if a key is in the dictionary without actually returning the value associated with the key can be done by using the keyword `in` as follows:

`"Kennedy" in offices` → true
`"Smith" in offices` → false (since it was just removed)

```
if ("Smith" in offices):
    ...
```

The keys and values in a dictionary don't have to be homogeneous; that is, they can each be of different types. For instance, the following key-value pair could be added to the dictionary:

```
offices[12345] = "abracadabra"
```

Although it doesn't necessarily make sense, the dictionary now has the following key-value pairs (in no particular order):

| **Last name** | **Office number** |
|---|---|
| Jones | 247 |
| Kennedy | 111 |
| Wilkerson | 355 |
| 12345 | abracadabra |

A neat way to obtain a list of all of the keys in a dictionary is to use the `keys` function as follows:

```
office_keys = list(offices.keys())
```

The `keys` function returns a "view" object that can be converted into a list by using the list constructor (similar to how we used it for the `filter` and `map` functions). The variable `office_keys` above then has the following value:

```
['Jones', 12345, 'Wilkerson', 'Kennedy']
```

There are several ways of iterating through the values of a dictionary. One uses the `keys` function just described. This can be accomplished as follows:

```
for k in offices.keys():
    print(offices[k])
```

Note that we don't need to convert the keys into a list if we are simply iterating through them. Of course, we can convert it to a list if we wanted to, but in the context above it isn't necessary.
The output of this snippet of Python code is:

```
247
abracadabra
355
111
```

Of course, to produce a key-value pair mapping, only a small modification is required:

```
for k in offices.keys():
    print(k, "->", offices[k])
```

The output of this now includes both the keys and values:

```
Jones -> 247
12345 -> abracadabra
Wilkerson -> 355
Kennedy -> 111
```

Another way that Python provides to do the same thing and which produces the same output is to use the dictionary method `items` as follows:

```
for k, v in offices.items():
    print(k, "->", v)
```

The `items` function returns a view object that contains pairs of values, each of which is a key-value pair in the dictionary.

Note that `k` and `v` are used to stand for `key` and `value`. Any variable name will do (as long as it is descriptive) but these are normally used by convention with dictionaries. Using `key` and `value` (or `val`) as opposed to `k` and `v` would also be perfectly fine.

**Dictionary comprehensions**
Just as with lists, comprehensions can be used to create dictionaries. Of course, these are known as dictionary comprehensions. Here's one that creates a dictionary with the key-value pairs such that the values are cubes of the keys, and the keys range from 1 through 5:

```
dict = {x: x ** 3 for x in range(1, 6)}
```

The created dictionary `dict` is therefore `{1: 1, 2: 8, 3: 27, 4: 64, 5: 125}`. The key-value pairs are specified in the dictionary comprehension as `x: x**3` (i.e., a key is some value *x*, and its associated value is *x* cubed). The range for the values (1 through 5) taken on by the variable *x* is specified as `for x in range(1, 6)`.

In the space below, try to modify the dictionary comprehension above so that the values are stored as strings instead of integers:

**Activity 1: Room Adventure...Reloaded**

In this activity, we will update the Room Adventure game that was designed in a previous RPi activity. The goal will be to replace the parallel arrays in the game with dictionaries. Such a substitution makes sense because parallel arrays associate (or map) the elements of two or more arrays by index value.

That is, the first element of one array is paired with the first element of another, and so on. Dictionaries are perfectly suited for this because they associate one value with another!

Recall that parallel arrays were used to represent the following relationships:

(1) Exits with exit locations (through the lists `exits` and `exitLocations`); and
(2) Items with item descriptions (through the lists `items` and `itemDescriptions`).

Exits were strings like “north” and “west”, and exit locations were rooms (instances of the class **Room**). Items were strings like “table” and “fireplace”, and item descriptions were strings like “It is made of oak. A golden key rests on it.” and “It is full of ashes.”

Very quickly, we see that we can replace the lists `exits` and `exitLocations` with a single dictionary (perhaps just called `exits`). Suppose, for example, that an exit to the east led to some instance of a room represented by the variable `r2`, and an exit to the north led to some instance of a room represented by the variable `r3`. A dictionary that represents this could be created as follows:

```
exits = {"east": r2, "north": r3}
```

Of course, this supposes that the variables `r2` and `r3` exist.

**Step 1: Replace the parallel lists with dictionaries**
The first thing to do to modify our game is to remove the parallel arrays for both exits and items, and replace them with dictionaries. This must be done in the constructor of the **Room** class; specifically, in lines 16 through 19:

```
14:  def __init__(self, name):
15:      self.name = name
16:      self.exits = []
17:      self.exitLocations = []
18:      self.items = []
19:      self.itemDescriptions = []
20:      self.grabbables = []
```

A few notes: (1) line numbers specified in this activity are valid only within the existing source code (i.e., not the one that is being modified because changes may invalidate the line numbers); and (2) some comments in the source code have been removed in this activity for brevity. The statements on lines 16 through 19 are the parallel arrays that will need to be replaced with dictionaries. The lists `exits` and `exitLocations` are paired; so are the lists `items` and `itemDescriptions`. Let's replace them with two dictionaries instead:

```
def __init__(self, name):
    self.name = name
    self.exits ={}
    self.items = {}
    self.grabbables = []
```

The highlighted statements are the two new dictionaries. There is no longer a need for matching lists since the dictionaries intrinsically match keys to values! Pay attention to the braces (as opposed to brackets).

**Step 2: Remove the accessors and mutators for the deleted parallel lists**

Recall that accessors and mutators were implemented for each of a **Room**'s instance variables. Since the lists `exitLocations` and `itemDescriptions` were removed, their respective accessors and mutators must also be removed. The existing accessors and mutators for the instance variables `exits` and `items` remain unchanged. In fact, they will work seamlessly with the new dictionaries.

The accessor and mutator for the old instance variable `exitLocations` are located on lines 39 through 45, and must be removed from the source code:

```
39:  @property
40:  def exitLocations(self):
41:      return self._exitLocations
42:
43:  @exitLocations.setter
44:  def exitLocations(self, value):
45:      self._exitLocations = value
```

The accessor and mutator for the old instance variable `itemDescriptions` are located on lines 55 through 61, and must also be removed from the source code:

```
55:  @property
56:  def itemDescriptions(self):
57:      return self._itemDescriptions
58:
59:  @itemDescriptions.setter
60:  def itemDescriptions(self, value):
61:      self._itemDescriptions = value
```

**Step 3: Modify the `addExit` and `addItem` functions**

The next step is to change the `addExit` and `addItem` functions in the **Room** class so that they appropriately insert new exits and items into dictionaries instead of parallel lists as is currently done. First, let's change the `addExit` function, which begins on line 74:

```
74:  def addExit(self, exit, room):
75:      # append the exit and room to the appropriate lists
76:      self._exits.append(exit)
77:      self._exitLocations.append(room)
```

Note how the function currently appends the exit to one list and the room to another. Let's change this so that the exit and room are added as a key-value pair to the appropriate dictionary:

```
def addExit(self, exit, room):
    # append the exit and room to the appropriate dictionary
    self._exits[exit] = room
```

Now, let's change the `addItem` function, which begins on line 82:

```
82:  def addItem(self, item, desc):
83:      # append the item and description to the appropriate lists
84:      self._items.append(item)
85:      self._itemDescriptions.append(desc)
```

This function is similar to the old `addExit` function. It appends the item to one list and the description to another. Let's change this so that the item and description are added as a key-value pair to the appropriate dictionary:

```
def addItem(self, item, desc):
    # append the item and description to the appropriate dictionary
    self._items[item] = desc
```

**Step 4: Modify the `__str__` function**
In the game, the player is continually presented with the status: location, items, exits, and inventory. This is specified in the **Room** class and specifically involves the bit of source code that generates the string representation of a **Room**. This is done in the `__str__` function, which begins on line 100:

```
100: def __str__(self):
101:     # first, the room name
102:     s = "You are in {}.\n".format(self.name)
103:
104:     # next, the items in the room
105:     s += "You see: "
106:     for item in self.items:
107:         s += item + " "
108:     s += "\n"
109:
110:     # next, the exits from the room
111:     s += "Exits: "
112:     for exit in self.exits:
113:         s += exit + " "
114:
115:     return s
```

The statements that need to be changed are on lines 106 and 112. In their current form, they iterate through the two lists. The list `items` used to contain the items (e.g., table). Since `items` is now a dictionary (with the items as keys and item descriptions as values), then we must iterate through its keys! This can be done by replacing line 106 with the following statement:

```
for item in self.items.keys():
```

Similarly, The list `exits` used to contain the exits (e.g., south). Since `exits` is now a dictionary (with the exits as keys and room objects as values), then we must also iterate through its keys. This can be done by replacing line 112 with the following statement:

```
for exit in self.exits.keys():
```

**Step 5: Modify the main part of the program**
There are two places that need modification in the main part of the program. Both are required because the current source code refers to the old lists that have been replaced by dictionaries. The first modification occurs in the part of the source code that is executed if the verb in the action specified by the player is *go* (e.g., "go south"). This part of the source code begins on line 247. Note that some of the next part of the if statement is provided for clarity:

```
247: if (verb == "go"):
248:     # set a default response
249:     response = "Invalid exit."
250:
251:     # check for valid exits in the current room
252:     for i in range(len(currentRoom.exits)):
253:         # a valid exit is found
254:         if (noun == currentRoom.exits[i]):
255:             # change the current room to the one ...
256:             currentRoom = currentRoom.exitLocations[i]
257:             # set the response (success)
258:             response = "Room changed."
259:             # no need to check any more exits
260:             break
261: # the verb is: look
262: elif (verb == "look"):
263:     ...
```

Currently, the algorithm iterates through the list of exits. If one matches the noun specified by the player, then the current room is changed to the matching exit location (in the parallel list). If this occurs, the break statement exits the loop (i.e., we don't need to check for more exits since a valid one has already been found). The changes required affect the highlighted lines in the snippet of source code above. Ultimately, the part of the if statement that is executed if the verb is *go* should be changed to the following source code. Again, some of the next part of the if statement is provided for clarity:

```
# the verb is: go
if (verb == "go"):
    # set a default response
    response = "Invalid exit."

    # check for valid exits in the current room
    if (noun in currentRoom.exits):
        # if one is found, change the current room ...
        currentRoom = currentRoom.exits[noun]
        # set the response (success)
        response = "Room changed."
# the verb is: look
elif (verb == "look"):
    ...
```

Note that no break statement is required because there is no enclosing loop! The second modification occurs in the part of the source code that is executed if the verb in the action specified by the player is *look* (e.g., "look table"). This part of the source code begins on line 261. Again, some of the next part of the if statement is provided for clarity:

```
# the verb is: look
elif (verb == "look"):
    # set a default response
    response = "I don't see that item."

    # check for valid items in the current room
    for i in range(len(currentRoom.items)):
        # a valid item is found
        if (noun == currentRoom.items[i]):
            # set the response to the item's description
            response = currentRoom.itemDescriptions[i]
            # no need to check any more items
            break
# the verb is: take
elif (verb == "take"):
```

As before, the algorithm iterates through a list (of items in this case). If one matches the noun specified by the player, then the response is changed to the matching item description (in the parallel list). If this occurs, the break statement exits the loop (i.e., we don't need to check for more items since a valid one has already been found). The changes required affect the highlighted lines in the snippet of source code above. Ultimately, the part of the if statement that is executed if the verb is *look* should be changed to the following source code. Again, some of the next part of the if statement is provided for clarity:

```
# the verb is: look
elif (verb == "look"):
    # set a default response
    response = "I don't see that item."

    # check for valid items in the current room
    if (noun in currentRoom.items):
        # if one is found, set the response to the ...
        response = currentRoom.items[noun]
# the verb is: take
elif (verb == "take"):
```

And that's it!

**Wrapping up data structures**

By now, you should have a good idea of the need for data structures. Generally, programs that solve problems store and manipulate data in some way. Data structures make this possible. In this curriculum, various useful data structures were introduced. Here is a table that summarizes them:

| Data structure | Description |
|---|---|
| Array | Similar pieces of data store in contiguous memory locations. Data values (elements) are stored at index locations. The size of an array must be known before using one. |

| | |
|---|---|
| Linked list | Similar pieces of data stored in nodes located in various memory locations. Nodes store data values and a link to the next node in memory. The list has a head (the first element) and a tail (the last element). The size of a linked list can grow or shrink as needed. |
| Stack | A list-like structure where inserting and deleting is performed at one end (usually called the top of the stack). Inserting is called a push operation; deleting is called a pop operation. The last item inserted in a stack is the first item out of the stack; therefore, a stack is a LIFO (last-in, first-out) data structure. |
| Queue | A list-like structure where inserting is performed at one end (usually called the back of the queue) and deleting is performed at the other end (usually called the front of the queue). Inserting is called an enqueue operation; deleting is called a dequeue operation. The first item inserted in a queue is the first item out of the queue; therefore, a queue is FIFO (first-in, first-out) data structure. |
| Binary tree | A tree-like data structure made up of nodes that store similar pieces of data. Each node has links to (up to) two children. The binary tree has a root (the top node) and leaves (nodes at the bottom with no children). From any node, a subtree can be described such that the node is the root of that subtree. |
| Ordered binary tree | A binary tree such that the values of all children in the left subtree of each node are less than the value of the node, and the values of all children in the right subtree of each node are greater than or equal to the value of the node. |
| Dictionary | Also known as an associative array, a data structure that maps keys to values. Keys are unchangeable pieces of data; values are associated with keys (one value per key). |