SELF BALANCING BINARY SEARCH TREE

"`bintree`" module
for Python 3.x
Version 1

by Diego E. Vélez
Cuenca, Ecuador
divelez69@gmail.com
May, 17th 2014

PREFACE

Thank you for purchasing this documentation. You'll be glad you did, because you'll discover that this API is more than a simple data container. With your purchase you're helping me to come out with improvements and new Python objects soon.

The `bintree` module is open source and its (currently only) object, the `bTree`, is very simple to use. I took special care to make it look authentically native to Python. It has all "magic" methods data containers have and is embedded into the culture and philosophy of the language. In fact, the `bTree`, on the surface, looks very similar to a `dict` object but with its items' keys arranged in order. Therefore, any programmer with some Python experience could start using right away after a brief look at the USAGE post in http://www.BintreePython.net or by going directly to the Appendix B.

Though the free reference document is good enough to get started, it omits crucial details that takes the most out of this API. Some applications, dealing with large amounts of data will certainly benefit from specialized methods and techniques (Appendix D), while others may find useful the great variety of operations over slices found at the binary tree (Appendix A).

A very special and unique feature of this API is the **sp**ecial **iter**ator (aka **spiter**) that allows modifications in the binary tree while iterating its items. Usually data containers (like lists or dictionaries) shouldn't be modified while iterating their items. This is not always the case with `bTree` and I highly encourage to take advantage of the **spiter**.

The iterators supplied by this object, whether is a **spiter** or the default dfs iterator have a `goto()` method that allows leaps to the iterations.

How this document is organized?

The TUTORIAL section is a step by step guide to learn the API usage. It's not too long if the scattered greenish boxes are omitted. They are there only to provide specific and detailed information on certain topics. You can check them on later if necessary.

The OPERATIONS section is an alphabetical organized glossary of all methods and operations of the `bTree` object and its subsequent objects: views and iterators. They are concisely but fully explained... with no details left behind. I highly recommend to check this section once you've studied the tutorial. It worth the deal a lot!

Finally there are the appendices. The Appendix A is very important as it's an entire section dedicated solely to slices and all the variety of methods and forms to declare them. The Appendix B is the quick tutorial reference for programmers in a hurry. The Appendix C is a table of the methods and their respectively big-O performances; and finally the Appendix D is an important enhancement material that provides examples and tips.

Is it working fine?

I personally tested the code intensively in all imaginable and possible scenarios, so I strongly believe it's free of bugs. However, if you stumble upon one, please report to me ASAP.

A personal note, please read:

This API is free and I hope you find it very useful in your projects. Whatever your project is, I will like to hear about it and how you are using it. Please, write me a short note. If you have suggestions, I will listen.

Unfortunately, I'm currently only offering a *Self Balancing Binary Tree* for Python 3. This is the first of a string of APIs I want to deliver. I expect to provide versions for Python 2 and 3 of other binary trees (e.g. red-black trees). Your purchase is just the right support I need to keep going. Thanks!

Need advice, support or specific code for your needs?

Don't hesitate to contact me by writing a note in my blog: www.BintreePython.net

Diego E. Velez
divelez69@gmail.com
May 17th, 2014
Cuenca, Ecuador

Table of Contents:

1 INTRODUCTION 4
2 THEORY 4
3 IMPORTANT NOTES AND FEATURES 6
4 TUTORIAL 7
4.1 The "bintree" module and the "bTree" object 7
4.2 Creating a Tree (insert, retrieve and delete an item) 7
4.3 Importing items from other data structures 9
4.4 Rejected items 12
4.5 The key function 13
4.6 Clearing and copying 16
4.7 Clearing, copying and pruning slices of a tree 16
4.8 Iteration 17
4.9 Views 17
4.10 Iteration over views 18
4.11 Modifying a tree while iterating- the SPITER 18
5 OPERATIONS 19
5.1 Tree object operations 19
5.2 Tree view objects' operations 27
5.3 Iterators and the goto() method 29
5.4 The special iterator (aka SPITER) 30
5.5 Set-like operations in tree views 31
Appendix A: Slicing definitions with optional arguments 32
Appendix B: Quick tutorial refference 35
Appendix C: Operations performances 38
Appendix D: TIPS and examples 39

1 INTRODUCTION

A *Self Balancing Binary Search Tree* is a very powerful mapping data structure. Items stored in this kind of data structure must have two components: a unique key and a value. For example, a Social Security ID number (the key) and name, address, salary, etc. (all together: the value). Unlike a hash table (which also requires the same two components), the self-balancing binary search tree maintains the keys in sorted order all the time. However, there are some trade-offs as some operations in balanced binary trees require O(Log n) time instead of O(1) of a hash table. The loss in performance is highly rewarded in some applications that require keys in sorted order, especially when iterations over sorted items are necessary.

2 THEORY

(you may skip this part and go to next section if you prefer)

A binary-search-tree, or BST, is a tree data structure in which each node has at most two child nodes (denoted as the left child and the right child). Nodes with children are referred as parent nodes, and all child nodes contain a reference to its parent node, except the root node. The nodes with no children are called leafs. Following this convention, ancestral relationships can be defined in a tree, therefore one node can be an ancestor (parent, grand-parent, etc) of another node, and at the same time a descendant (a child, grand-child, great-grand-child, etc.) of another node. The root node is the only node that doesn't have ancestors and it is the ancestor of all nodes in tree. Any node in a tree can be reached from its root-node.

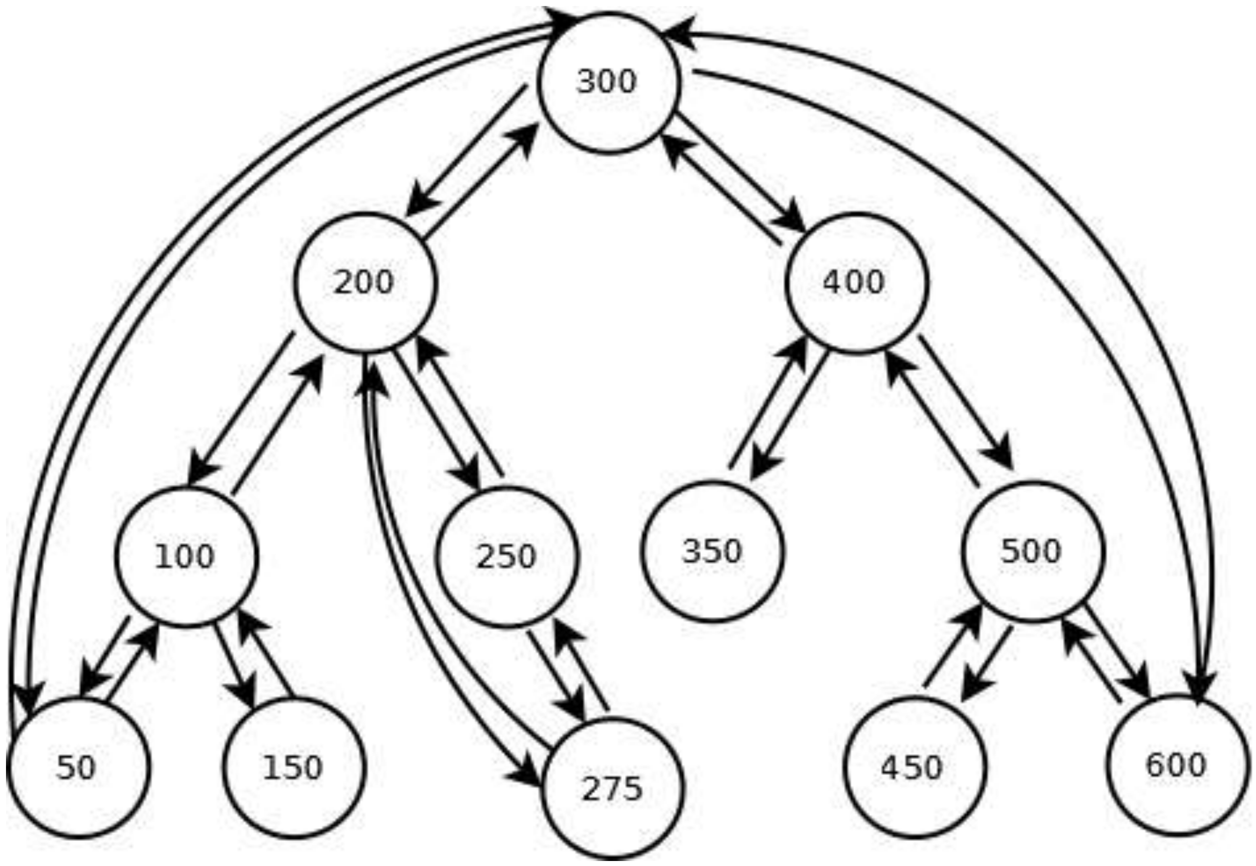
Nodes are data containers that store items from the data set. Each item must have two components: the key and the value. By convention, a node's key must be greater than its left child node's key; and lower than its right child node's key. Thus it's quite simple to reach a particular node from the root node given its key.

It is said that a child node is one level deeper from its parent node, a grandchild is two levels deeper from its grandparent, and so on. The depth of a tree (or a branch) is the maximum number of levels that has in either side or leg. An unbalanced BST with the simplest algorithms for insertion may yield a tree with a depth equal to n in rather common situations. For example, when items are inserted in sorted key order, the tree degenerates into a linked list with n nodes. Thus the search for the last inserted node from the root will take O(n) time. A balanced BST solves this problem.

A balanced node is when the depth differences of its two branches or legs (left and right) are not greater than one. A balanced BST has all its nodes balanced. This property ensures a O(Log n) search time always. For example, a tree with n = 1.000.000 items the minimum depth possible is 19 (Base 2 logarithm of one million is 19.93). This performance is highly more efficient than a linear time of one million searches!

To ensure this property the BST algorithms must always keep the balancing property every time an insertion or a deletion occurs. Tree transformations (such as tree rotations) keep depths proportional at all levels. Although certain overhead is involved, it is justified in the long run by ensuring fast execution at later operations.

There are plentiful of good resources to learn more about Self Balanced BST in the web. (e.g. http://en.wikipedia.org/wiki/AVL_tree or http://en.wikipedia.org/wiki/Binary_search_tree)



This image of a balanced BST is taken from Samuel Jacob's Weblog (http://samueldotj.com/blog/), which happens to have an extraordinary good explanation about *Self Balancing Tree as Heap*.

3 IMPORTANT NOTES AND FEATURES

- The main purpose of this API is to expand the supply of powerful data structures already found in Python. Basically, this is a mapping type, like dictionaries, but with its keys in sorted order.
- By design this is a *binary search tree* object but an inner hash table has been added in order to have the best of the two worlds. Therefore, many operations are as efficient as a dictionary. For example: search and replace of an item, given its key, take O(1) time. However, other operations like insertions and deletions take at most O(Log n) time which is normal in BSTs. In Appendix C table are all tree operations and their performances.
- Items in a tree are always arranged in sorted order (according to their keys values). They can be iterated totally or partially, in ascendant or descendant order. This feature also allows operations upon slices of the tree; therefore there's a wide range of methods to work with slices, whether using key values or items' positions in tree (indices). Appendix A is solely dedicated about slices .
- Usually it's very difficult (sometimes impossible) to perform changes and iterate a data container at the same time (e.g. lists and dictionaries). Therefore, this API offers a **sp**ecial **iter**ator (aka **spiter**) to do just that: iterate and perform changes at the same time. This is not the tree's default iterator because it's a bit slower than the default one (aka dfs iterator) which uses a fast *depth first search* technique, but tree modifications are forbidden.
- Other features offered by this API are: special searches using `min()` and `max()` methods; an optional *key function* that produces a *comparison-key* to arrange items in tree; and methods with fast and efficient algorithms to import data from other sources or to create sub trees using filters. Appendix D offers tips and examples.

4 TUTORIAL

4.1 The "bintree" module and the "bTree" object

To use this powerful data structure only the `bTree` class object has to be imported from `bintree` module at the beginning of a program:

```
from bintree import bTree
```

This is the only object needed. Other objects and functions inside the module are reserved for private use.

The self-balancing-BST (called just tree in this documentation) is a mapping object and the same rules that apply for keys in dictionaries apply for trees (please, refer to Mapping Types in Python documentation). An additional restriction has been added: keys must be of the same type so they can be orderable and therefore arranged in nodes of a tree. For example, a tree that has been accepting string type keys, can't accept an item with a key of integer type. Any attempt to do so with the methods described later will raise a **`KeyError`**. However, numeric key types can coexist in the same tree because they can be compared and therefore arranged in hierarchy according to their values. A tree fixed to accept integer keys' types accepts float keys' types. Since computers store floating-point numbers as approximations it is usually not recommended to use floats as keys, unless the programmer knows very well what he/she is doing.

In order to avoid mingling keys of different types, the `bTree` object has the **`keys_value_type`** attribute; which establishes the class type all keys in tree must match. For example, if this attribute has a value equal to **`str`** then all keys in tree must be strings; or if it has a value equal to **`int`** then all keys in tree must be integers or floats. **Notice** that this attribute value is a class-type; not a value of a certain class type.

4.2 Creating a Tree (insert, retrieve and delete an item)

The simplest way to create a tree is:

```
t = bTree()
```

Now `t` is an empty tree and `keys_value_type` attribute isn't defined yet. This situation changes with the insertion of the first item:

```
t[key] = value
```

Now `t` has one item (*key* and *value*) and `keys_value_type` attribute is automatically set equal to key's type `(keys_value_type = type(key)`... to be more precise). From now and on, `t` won't accept items with different keys' types. Any attempt to do so raises a **`KeyError`**.

To retrieve the value of an item or to delete an item use the following methods respectively:

```
t[key], del t[key]
```

In both cases, raises a **`KeyError`** if key isn't found in `t`.

The `keys_value_type` attribute can also be set at the creation of the tree instance:

```
t = bTree(typ=class_type)
```

So allowed key types are defined in advance before the insertion of any item. Argument *typ* if given must be a class type, not a variable of a class type. E.g. `t = bTree(typ=str)` initializes a tree that can only contain string type keys, while `t = bTree(typ='mice')` raises a **`TypeError`**.

Notes about keys and their class types:

- All keys in a tree must be of the same class type. The `keys_value_type` attribute prevents mingling different type of keys.
- Only immutable and hashable objects (e.g. integers, strings, tuples, floats, etc.) are allowed to be used as keys. Mutable objects like dicts or lists cannot be used as valid keys in a dictionary. Any attempt to use them raise a **`TypeError`**.
- Keys must be comparable among them, and therefore orderable. The use of objects that do not comply this requirement will eventually raise a **`TypeError`**.
- Keys in a tree are unique. A tree won't accept two or more items with the same key value. The insertion of an item with a key that already exists in another item in tree, replaces the existing item.
- Tuples can be used as keys of a tree as long as all of their inner items are also immutable and hashable. Otherwise raises a **`TypeError`**.
- The use of tuples as keys in a tree, even though all of their items are immutable and hashable, may cause subtle errors. They have to be perfectly compared among them in order to avoid undesired issues. The following two different operations and their outputs exemplify this possible situation:
  - `(34, 'cat') > (24, 85.36)` ==> Outputs a **`True`** value
  - `(34, 'cat') > (34, 85.36)` ==> Raises a **`TypeError`**.

  The programmer should be alert and avoid these possible conflicts.
- A bad idea: It's perfectly possible to set in advance the `keys_value_type` attribute to be of a mutable class type. For example:
  - `t = bTree(typ=dict)`

  Though this line of code doesn't raise any error, `t` is doomed to be an empty tree.

4.3 Importing items from other data structures

The `bTree` object offers the possibility to import a bunch of items with their own methods instead of using a *for-loop*(*) like the following:

```
for key, value in items:
     t[key] = value
```

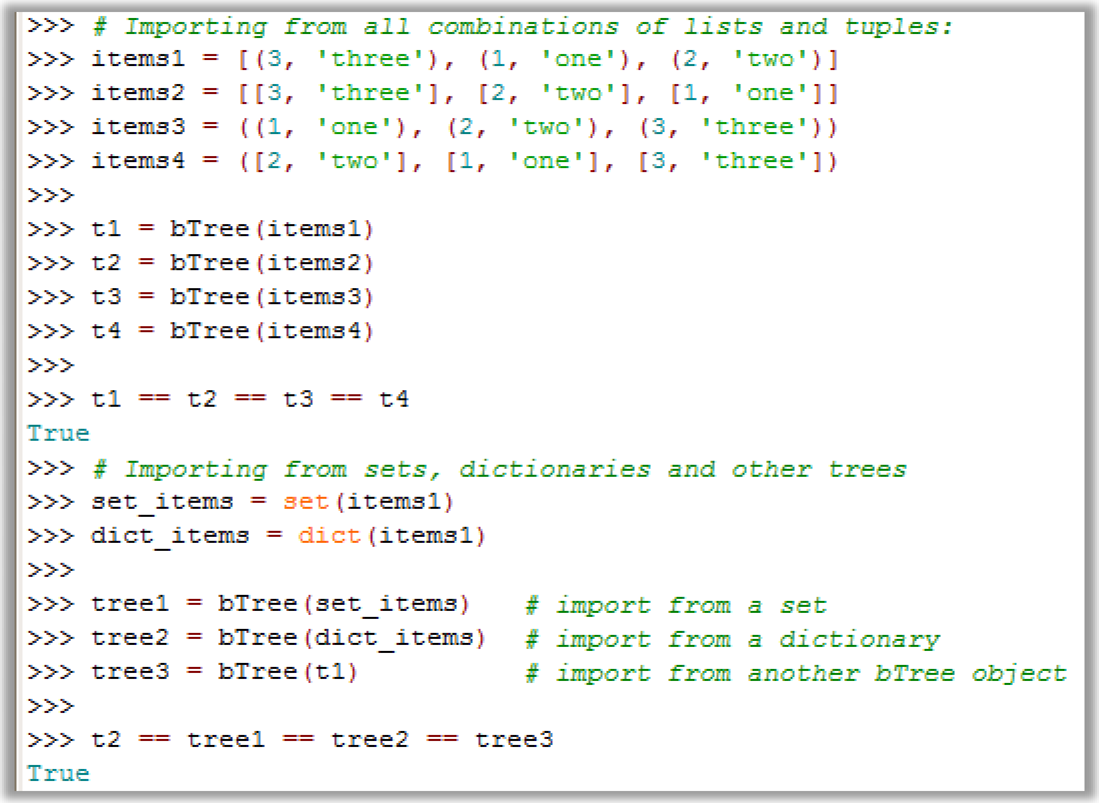
(*) This kind of for-loops should be avoided. Please read Appendix D.

The operations:

`t = bTree(items)`	==>	*Creates a new tree `t` from items*
`t.update(items)`	==>	*Imports items to `t` existing tree*

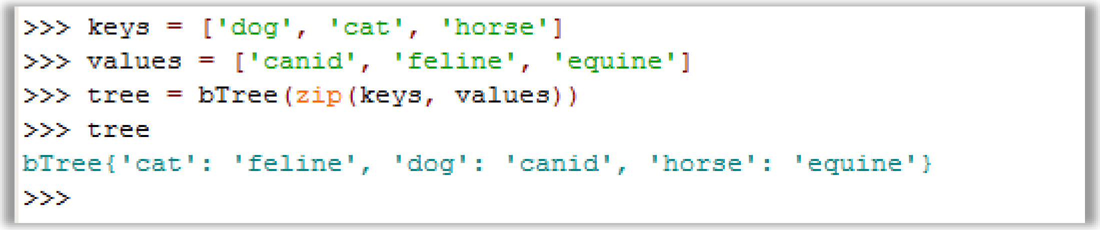
Items can be any iterable such as lists, tuples, dictionaries, trees (`bTree` objects), and so on. What is important is that each item must be convertible into a valid tree key/value pair. Naturally, sequences of two item tuples (or lists) and mapping structures (e.g. dictionaries or `bTree` objects) yield key/value pairs; and they are imported straightforward.

To illustrate, the following examples all return a tree equal to: `bTree{1: 'one', 2: 'two', 3: 'three'}`:

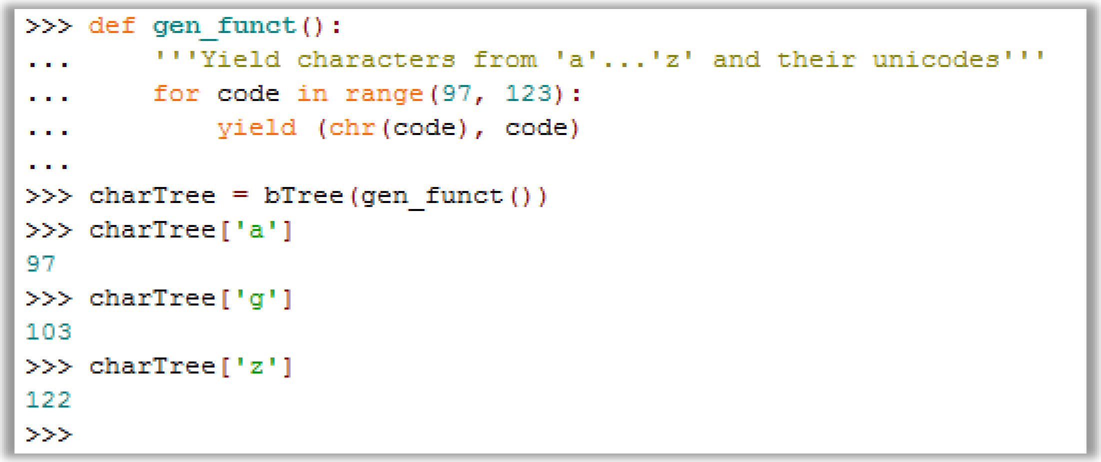


```
>>> # Importing from all combinations of lists and tuples:
>>> items1 = [(3, 'three'), (1, 'one'), (2, 'two')]
>>> items2 = [[3, 'three'], [2, 'two'], [1, 'one']]
>>> items3 = ((1, 'one'), (2, 'two'), (3, 'three'))
>>> items4 = ([2, 'two'], [1, 'one'], [3, 'three'])
>>>
>>> t1 = bTree(items1)
>>> t2 = bTree(items2)
>>> t3 = bTree(items3)
>>> t4 = bTree(items4)
>>>
>>> t1 == t2 == t3 == t4
True
>>> # Importing from sets, dictionaries and other trees
>>> set_items = set(items1)
>>> dict_items = dict(items1)
>>>
>>> tree1 = bTree(set_items)    # import from a set
>>> tree2 = bTree(dict_items)   # import from a dictionary
>>> tree3 = bTree(t1)           # import from another bTree object
>>>
>>> t2 == tree1 == tree2 == tree3
True
```

Items can also be imported from iterator objects and generator functions too:



```
>>> keys = ['dog', 'cat', 'horse']
>>> values = ['canid', 'feline', 'equine']
>>> tree = bTree(zip(keys, values))
>>> tree
bTree{'cat': 'feline', 'dog': 'canid', 'horse': 'equine'}
>>>
```



```
>>> def gen_funct():
...     '''Yield characters from 'a'...'z' and their unicodes'''
...     for code in range(97, 123):
...         yield (chr(code), code)
...
>>> charTree = bTree(gen_funct())
>>> charTree['a']
97
>>> charTree['g']
103
>>> charTree['z']
122
>>>
```

Warning: It's very important that generators halt yielding items at some point by raising a `StopIteration`, otherwise the `bTree`'s method gets stuck in an endless loop.

When initializing a new tree (`t = bTree(items)`), if items in *items* have keys of different types, the tree is formed only with those that are valid and most abundant. The rest are rejected. However, the `keys_value_type` attribute can be set in advance by explicitly giving the *typ* argument:

```
t = bTree(items=items, typ=class_type)
```

In this case, tree isn't necessarily formed with the most abundant keys' type, but with those that match the class_type. The rest are rejected. This is also the case when importing items to an existing tree:

```
t.update(items=items)
```

Only items whose keys' types match the tree's `keys_value_type` attribute are imported. However, when the importing tree doesn't have such attribute yet defined (so it's an empty tree) it's like the initialization method: imports only the valid and most abundant keys' type and rejects the rest.

When duplicate keys are found, new items replace old ones. **Note**: key and value are replaced.

Important advise: *Whenever items are known in advance, importing them with these two built-in methods are better choice than the for-loops. They are faster, more elegant, and more efficient. Appendix D has a greenish box (suitable for geeks) about this subject.*

Items importing versatility:

Items can be imported from a wider spectrum of data structures.

Given *items* must be any iterable (except strings, bytearrays and bytes); and each yielded item, while iterating *items*, must be one of the following data containers: list, tuple or dictionary. If so, inner algorithms tries to convert each one of them into a valid key/value pair according to the following rules:

If yielded item is a:

- List or a tuple: Picks the first item and becomes the *key* (as long it complies the rules for a valid key), otherwise rejects the yielded item. If not rejected, packs the rest of the items in a list and becomes the *value*. The value can't be a zero length list, if so the value is set to **`None`**. Also, if only one more item is found, it isn't packed into a list and instead the *value* is that item itself. The following table clarifies the rule:

`if len(item)`	**Process:**
`== 0:`	`item is rejected`
`== 1:`	`key = item[0];   value = None`
`== 2:`	`key = item[0];   value = item[1]`
`> 2:`	`Key = item[0];   value = list( item[1:] )`

- Dictionary: Picks the value of the item with key equal to `'key'` and becomes the *key* (as long it complies the rules for a valid key). If it fails to find such item, the yielded item is rejected. Then after removing the key's item, the yielded item dictionary becomes the *value*. For example, if following yielded item is found:

  ```
  {'address': '419 W. Willow', 'age': 41, 'key': '1211-963-7', 'name': 'mary poppins'}
  ```

  ...then the new conversed key/value pair item is:

  key = `'1211-963-7'`
  value = `{'address': '419 W. Willow', 'age': 41, 'name': 'mary poppins'}`

  Any form of `'key'` string is valid, e.g. `'KEY'`, `'Key'`, `'kEy'`, etc; but If more than one of them are found in the same yielded item dictionary, it chooses one but in the following hierarchy order from left to right: `['key', 'KEY', 'Key', 'keY', 'kEy', 'kEY', 'KeY', 'KEy']`

Please, refer to Appendix D for more importing items examples and tips.

4.4 Rejected items

Some items, while being imported, may be rejected for various reasons. They are irremediable lost, unless the *keep_reject* argument is giving `True`:

```
t = bTree(items=items, keep_reject=True)
t.update(items=items, keep_reject=True)
```

In both cases, rejected items are temporally stored inside the `bTree` object until they're retrieved some time later with `get_rejected()` method:

```
rej_itms = t.get_rejected()
```

Once retrieved, all temporally stored items so far are erased from tree and no longer available.

Retrieved Rejected items:

Retrieved rejected items are organized in a dictionary that may have up to three categories (keys): **`'init'`**, **`'update'`** and **`'setnewkey'`**. Items that may have been rejected at tree's initialization are stored inside the **`'init'`** category. Meanwhile items that `update()` and `setnewkey()` methods may reject are stored inside **`'update'`** and **`'setnewkey'`** categories respectively.

Since initialization happens only once, the group of rejected items that occurred during this process is the content of **`'init'`** category itself. On the other hand, since `update()` and `setnewkey()` methods can be called many times in a tree's lifetime, their respective rejected items' groups (if arise) are placed in stacks; which in turn are placed inside **`'update'`** and **`'setnewkey'`** categories respectively.

A rejected items' group may look like:

```
rej_items =  {'init':{...},
              'update':[{...},{...},{...}, ...{...}],
              'setnewkey':[{...}, {...}, {...}, ...{...}]
             }
```

Rejected items' groups are dictionaries. This is so, because rejected items need to be classified. Recalling **Items Importing**, there are three reasons why an item can be rejected: 1) It fails to be transformed into a key/value pair, 2) It's successfully transformed into a key/value pair, but key doesn't comply the rules of a valid key; and 3) It's a valid key/value pair, but key's type doesn't match tree's predominant `keys_value_type` attribute. Items that are rejected because of the first and second reasons are stored in a general `'no valid'` category. The others are stored in different categories according to their keys' types. E.g. a rejected items' group may look like:

```
rej_group =  {'no valid': [...],
              <class 'int'>: [...],
              <class 'str'>: [...],
              ...}
```

Each rejected item is stored in its raw form, even though it passed the key/value transformation.

4.5 The key function

```
t = bTree(keyfunc=key_function)
```

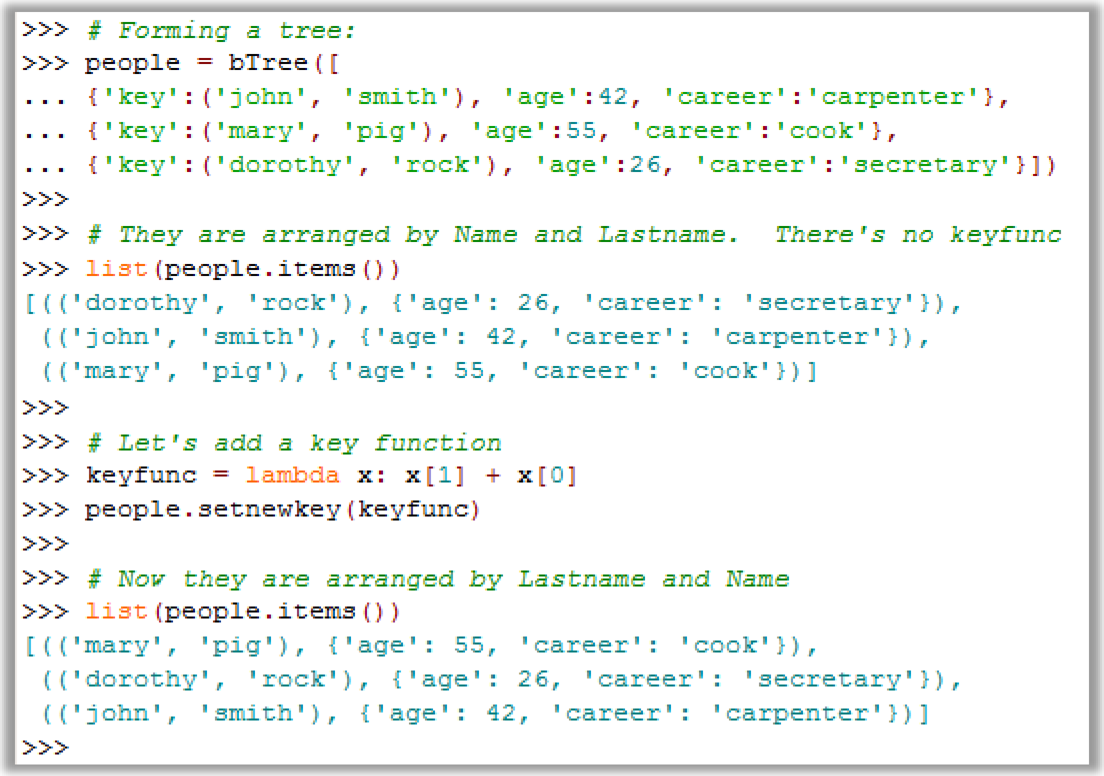
The `keyfunc` argument is equivalent to `key` in `sorted()` build in function. It specifies a function of one argument that is used to extract a *comparison-key* from each item's key (for example, `keyfunc=str.lower`). Thus nodes are arranged in tree according to their items' *comparison-keys*' values.

The following method is for changing (or setting) the *key function* in a non-empty tree:

```
t.setnewkey(new_key_function)
```

When calling this method the entire tree is re-arranged according to the new *comparison-keys* produced by the new *key function* in O(n Log n) time. This may lead to a change in the `keys_value_type` attribute, and may reject some items.

Example:



```
>>> # Forming a tree:
>>> people = bTree([
... {'key':('john', 'smith'), 'age':42, 'career':'carpenter'},
... {'key':('mary', 'pig'), 'age':55, 'career':'cook'},
... {'key':('dorothy', 'rock'), 'age':26, 'career':'secretary'}])
>>>
>>> # They are arranged by Name and Lastname.  There's no keyfunc
>>> list(people.items())
[(('dorothy', 'rock'), {'age': 26, 'career': 'secretary'}),
 (('john', 'smith'), {'age': 42, 'career': 'carpenter'}),
 (('mary', 'pig'), {'age': 55, 'career': 'cook'})]
>>>
>>> # Let's add a key function
>>> keyfunc = lambda x: x[1] + x[0]
>>> people.setnewkey(keyfunc)
>>>
>>> # Now they are arranged by Lastname and Name
>>> list(people.items())
[(('mary', 'pig'), {'age': 55, 'career': 'cook'}),
 (('dorothy', 'rock'), {'age': 26, 'career': 'secretary'}),
 (('john', 'smith'), {'age': 42, 'career': 'carpenter'})]
>>>
```

To remove a *key function* the *keyfunc* argument must be given **None**:

```
t.setnewkey(None)
```

Nodes are re-arranged and the new *comparison-keys* are the keys themselves.

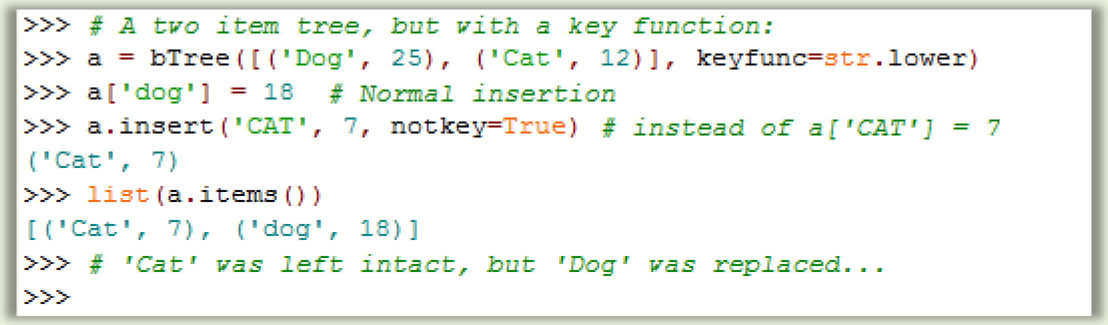
Consequences of having a *key function*:

Different keys may produce the same *comparison-key*. Two or more items with the same *comparison-key* can't coexist in the same tree, even though their keys are different. Therefore the programmer should be aware of the following situations:

1. When inserting an item (`t[key] = value`) with a *comparison-key* that already exists in `t`, the whole new item (key and value) replaces the old item. In order to leave old key intact, the `insert()` and `replace()` methods should be used instead with `notkey` argument given **`True`**:

```
t.insert(key, value, notkey=True)
t.replace(key, value, notkey=True)
```

   Example:



```
>>> # A two item tree, but with a key function:
>>> a = bTree([('Dog', 25), ('Cat', 12)], keyfunc=str.lower)
>>> a['dog'] = 18   # Normal insertion
>>> a.insert('CAT', 7, notkey=True) # instead of a['CAT'] = 7
('Cat', 7)
>>> list(a.items())
[('Cat', 7), ('dog', 18)]
>>> # 'Cat' was left intact, but 'Dog' was replaced...
>>>
```

2. When importing new items (`t.update(items)`), items with duplicate *comparison-keys* found in *items* also replace key and value of old ones in `t`. In order to leave old keys intact, the `update()` method should be used with `notkey` argument given **`True`**:

```
t.update(items, notkey=True)
```

3. When setting or changing a *key function* of a tree (`t.setnewkey(keyfunc)`) and if two or more items end up having the same *comparison-key*, tree only keeps the first one found by algorithms. The rest are expelled and temporally stored in the `bTree` object as rejected items.

4. When comparing two trees (`t1 == t2`) that share the same *key function*, algorithms test if respective *comparison-keys* and values match. Therefore, the operation may return **`True`** even though keys don't match. On the other hand, the operation return **`False`** if trees' *key functions* are different objects... even though they do the same and `t1` and `t2` match in everything else. If required an exact comparison of trees' items (keys and values) regardless of their respective *key functions*, trees' item views comparison should be used instead:

```
t1.items() == t2.items()
```

These situations are irrelevant when trees don't have key functions; but observe the persistent subtle difference: Trees usually replace keys when inserting duplicates, while dictionaries never.

Rejected items by **`setnewkey()`**:

When changing (or setting) a *key function* in a non-empty tree, some items may be expelled from tree because:

1) the new *key function* fails to return a *comparison-key* or produce one of an invalid type (e.g. mutables, unhashables, etc.);
2) *key function* produce more than one *comparison-keys*' types; therefore only those that belong to the predominant type stays in tree; and
3) Two or more keys produce the same *comparison-keys*.

Contrary to other methods, rejected items by `setnewkey()` are temporally stored in `bTree` object by default. In order to not to store them the *keep_reject* argument must explicitly be given **`False`**:

```
t.setnewkey(keyfunc, keep_reject=False)
```

As duplicates are rejected, not replaced, they are temporally stored in the subcategory `'duplicates'` that are only found in the `'setnewkey'` category of the retrieved rejected items.

Why rejected items are recoverable?

When importing items with `bTree(items)` or `t.update(items)`:

> Many times this tutorial encourages the use of these methods when *items* are know in advance. They are faster, efficient and more elegant than simple *for-loop* methods. In the process items are classified according to their keys' types even if some of them are rejected. These results may be worth to some programmers who are depurating and classifying raw data; thus in order to save time and code, rejected items are recoverable. For instance, other trees with different `keys_value_type` attributes can be formed upon the retrieved rejected items.

When changing or setting a *key function* (`t.setnewkey(keyfunc)`):

> It's not a good idea to just lose rejected items resulting from a *key function* change. They were once valid items in tree; and by temporally storing them programmer has at least a chance to check them and decide what to do with them. Remember that in this case, items with duplicate *comparison-keys* that arise from changing a *key function*, don't replace old items. Instead they are rejected and temporally stored in the bTree object as well.

There may be many other reasons why programmers would wish to have access to rejected items. However, rejected items temporally storing is an optional feature.

4.6 Clearing and copying

To clear and copy a tree:

`t.clear()`	or	`del t[:]`	==>	*Remove all items of `t` in O(1) time.*[1]
`t.copy()`	or	`t[:]`	==>	*Return a shallow copy of `t` and its items in O(n) time.*[2]

1. When removing items in a tree, `keys_value_type` attribute and *key function* are kept intact. The `keys_value_type` attribute, if defined, can't be removed or changed. On the other hand, *key function* can only be removed by changing it to a null value: `t.setnewkey(None)`.
2. Shallow copy of `t` and its items are new and different objects.

4.7 Clearing, copying and pruning slices of a tree

If given with arguments, the methods `clear()`, `copy()` and `prune()` respectively delete, copy and prune slices of a tree. For example:

`t.clear(floor='g', ceil='s')`	==>	Remove from `t` all items with keys from `'g'` to `'s'` including both ends.
`t.copy(slice=(100))`	==>	Return a new `bTree` object with a copy of the first 100 items of `t`.
`t.prune(slice=(500, None))`	==>	Prune the last 500 items of `t` and return a new `bTree` object with those items.
`t.clear(keys=keys)`	==>	Clear from `t` all keys found in *keys*. Skip those that aren't found.

Slices can also be expressed with the extended slice notation. For example:

`del t['r':'s']`	==>	Remove from `t` all items that start with the letter `'r'`.
`t[10.2:]`	==>	Return a new `bTree` object with a copy of all items with keys equal or greater to `10.2`.
`t.prune[::10]`	==>	Prune from `t` one out of 10 items starting from the first item. Return a new `bTree` object with the pruned items.

- The `start` and `stop` arguments of the standard notation (`[start:stop:step]`) refer to keys, not indices. To slice with indices, use instead the corresponding method with a given `slice` argument, e.g. `t.copy(slice=(100, 200, 3))`.

All three methods have the same optional arguments for slicing. Please, refer to Appendix A to learn how to declare the slices.

4.8 Iteration

The most basic forms of iteration over the keys of a tree (t) are:

In ascendant order:
```
for key in t:
     [...suit code...]
```

In descendant order:
```
for key in reversed(t):
     [...suit code...]
```

The bTree object supports iter() and reversed() built-in functions, which respectively return ascendant and descendant keys' iterator objects. Please refer to iter(), reversed() and next() built-in functions in Python documentation.

Iteration can also be done over views of a tree; which offer more options and features.

4.9 Views

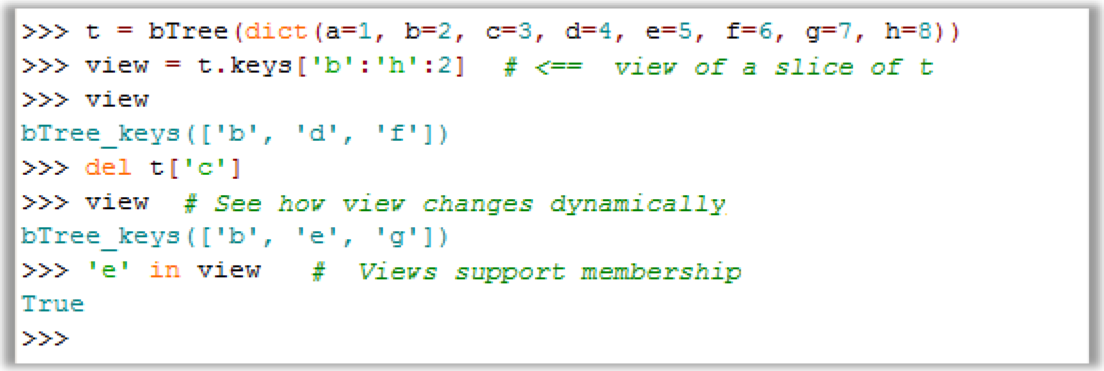
Views are objects that provide a dynamic view on the tree's entries. They are not truly data containers, but instead lazy sequences that will see changes in the underlying tree object. By design, trees are always in ascendant order, so their views' sequences too. Tree view objects behave like *dictionary view objects*.

Tree view objects, without given arguments, are bTree.keys(), bTree.items() and bTree.values() refer to all entries in tree. When given arguments, they refer to a slice of a tree. There are two ways to create a view of a slice of a tree:

With:		***Example:***
1. Giving arguments	==>	`t.keys(slice=(100, 200))`
2. Slice standard notation	==>	`t.items['b':'c']`

Please refer to Appendix A, because it explains slicing in more detail.

Views support membership and can be iterated over.



```
>>> t = bTree(dict(a=1, b=2, c=3, d=4, e=5, f=6, g=7, h=8))
>>> view = t.keys['b':'h':2]  # <==  view of a slice of t
>>> view
bTree_keys(['b', 'd', 'f'])
>>> del t['c']
>>> view  # See how view changes dynamically
bTree_keys(['b', 'e', 'g'])
>>> 'e' in view   #  Views support membership
True
>>>
```

Please, refer to Tree view objects at Tree Operations section of this tutorial.

4.10 Iteration over views

Views can be iterated over like any other sequence. For example:

```
for item in t.items():     ==>    Iterates over the items (as tuples
     [...suite code...]           of (key, value)) of t.
```

The `iter()` and `reversed()` built-in functions return iterator objects when given tree views:

Operation		Return an iterator object over...
`iter(t.keys())`	...	the **keys** of `t` in ascendant order.
`reversed(t.items(slice=(10))`	...	the first 10 **items** of `t` in descendant order.
`iter(t.values['c':'d'])`	...	the **values** of the items of `t` whose keys start with the letter 'c'.

Shortcuts:
- `iter(t)` **<==>** `iter(t.keys())`
- `reversed(t)` **<==>** `reversed(t.keys())`

Passing the iterator to repeated calls of `next()` built-in function return successive elements (keys, items or values) of the sequence. Please refer to Python documentation about iterators.

Iterations can jump to other positions in a sequence by calling iterator's `goto()` method. For the purpose this method has two optional arguments: `key` and `index`.

```
i.goto(key='apple')    ==>   Iterator i goes to item with key 'apple'.
i.goto(index=20)       ==>   Iterator i goes to item in the positional index 20.
```

After a leap, iteration continues normally from the new position.

Iterators use a *depth first search* (aka dfs) technique for iteration. A dfs iterator is a valid object while tree isn't modified. Usually, a dfs iterator detects changes in underlying tree and becomes invalid, so farther attempts to yield elements raises a **`RuntimeError`**. A dfs iterator that fails to detect changes outputs no trusted data.

4.11 Modifying a tree while iterating- the SPITER

The `bTree` object has a **special iterator** (aka **spiter**), which uses a different iteration technique that allows tree modifications while iterating a sequence. The **spiter** is obtained with the same `iter()` and `reversed()` built-in functions; but given view must have the ***spiter*** option turned *on* before. It has a `goto()` method too like the dfs iterator, and its usage is identical. Please refer to **spiter** documentation at Tree view objects section at the end of this tutorial.

It's also recommended to read about iterators and the goto method at Tree Operations at the end of this tutorial.

5 OPERATIONS

Tree supports all operations of custom mapping types, plus some new and useful ones:

5.1 Tree object operations

`len(tree)`

Return the number of items in *tree*.

`tree[key]`

Return the item of *tree* with key equal to *key*. Raises a **`KeyError`** if *key* is not in tree.

`tree[``start:stop:step`**`]`**

Return another `bTree` object containing the copy of slice of *tree* from *start* to *stop* with a step *step*. All three arguments are optional and the same extended slicing rules apply. Arguments *start* and *stop* refer to keys not indices(*). *Step* must be an integer and its default value is **`1`**.

(*) Given *start* and *stop* arguments not necessarily have to match existing keys in tree. When failed to match, the slicing takes place in all items within the range. However, *start* and *stop* value types must match `keys_value_type` attribute, otherwise raises a **`TypeError`**.

`tree[key] = value`

If *key* is not in tree, insert a new item (*key* and *value*) and automatically keeps the balancing property (performance: O(Log n) time). If *key* is already in tree, replaces the existing item with the new one (performance: O(1) time). **Note:** when replacing, it replaces both values: *key* and *value*(*).

If this is the first item in tree and *tree*'s `keys_value_type` attribute hasn't been defined yet, *key*'s value type set the attribute; else *key*'s value type must match the attribute. A **`TypeError`** raises if it fails to match.

This operation doesn't support slicing arguments. To insert a bunch of new items in *tree*, use `update()` method instead. And to remove items, use **`del`** statement, or `clear()` and `prune()` methods.

(*) The methods: `insert()`, `replace()` and `setdefault()` have an option to keep the item's *key* intact.

`del tree[key]`

Remove `tree[key]` from *tree*. Raises a **`KeyError`** if *key* is not in *tree*.

`del tree[``start:stop:step`**`]`**

Remove from *tree* `tree[start:stop:step]` slice items.

`key in tree`

Return **`True`** if *tree* has a key *key*, else **`False`**.

`key not in tree`

Equivalent to `not key in tree`.

`iter(tree)`

Return an iterator over the keys of the *tree* in ascendant order(*). This is a shortcut for `iter(tree.keys())`.

`reversed(tree)`

Return an iterator over the keys of the *tree* in descendant order(*). This is a shortcut for `reversed(tree.keys())`.

(*) Items in tree are arranged by their keys' values or their *comparison-keys* if tree has a *key function*.

`tree1 == tree2`

Compares *tree1* and *tree2* and return **`True`** if their respective items are equal (key and value) and in number. Not necessary both trees have to be the same object.

When using *key functions* and both trees share the same one, the operation compares the *comparison-keys* instead of the *keys* themselves; else return **`False`**. When *keys* need to match regardless their *key functions*, use the operation with items' view instead:

```
tree1.items() == tree2.items()
```

Please, check refer to Consequences of having a *key function* in documentation.

`clear([floor, ceil, step, slice, keys])`

Remove all items from *tree* in O(1) time if no arguments are given. The optional arguments if given, remove a slice of the tree's items.

The Appendix A explains the arguments usage for slicing.

`copy([floor, ceil, step, slice, keys])`

Return a shallow copy of *tree* if no arguments are given. The optional arguments if given, return a new `bTree` object with a copy of slice items of *tree*.

The Appendix A explains the arguments usage for slicing.

`get(key[, default])`

Return the value for *key* if *key* is in *tree*, else *default*. If *default* is not given, it defaults to **`None`**, so that this method never raises a **`KeyError`**.

`get_rejected()`

Return a list of all items rejected by methods `update()`, `setnewkey()` and `bTree()` at initialization.

Once the rejected items have been retrieved, they are erased in *tree* object, and therefore not longer available.

`haskey()`

Return **`True`** if tree has a *key function*, else return **`False`**.

`index(key)`

Return the *i-th* positional index of *key* in *tree* if found, else raises a **`KeyError`**.

`insert(key, value[, notkey])`

It is equivalent to `t[key] = value`. If *notkey* isn't given: inserts a new item or replace an existing one.

If *notkey* is given **`True`** and *key* is in tree, only replaces *value* and *key* is left intact. Else, if *key* isn't found in tree, it inserts the new item as `t[key] = value` operation does.

Return tree's item's `(key, value)` after insertion.

item(*index***)**

Return the item located at the positional *index-th* in *tree* if *index* is given positive; else return the item in the *index-th* position starting from the last*.

Examples:

`tree.item(i)` ==> Return the item located at the *i-th* position.
`tree.item(0)` ==> Return the first item in *tree* (the one with lowest *key*).
`tree.item(-1)` ==> Return the last item in *tree* (the one with highest *key*).

* Remember: items are arranged in *keys*' ascendant order.

items([*floor, ceil, step, slice, spiter***])**

Return a new view of the tree's items `(key, value)` pairs. Please refer to tree view objects operations.

keys([*floor, ceil, step, slice, spiter***])**

Return a new view of tree's keys. Please refer to tree view objects operations.

keystype()

Return `keys_value_type` attribute that all keys' types in tree must match. Return **`None`** if `keys_value_type` attribute isn't defined yet. The class types `float` and `int` are always set as `int`.

max([*ceil, getindex***])**

Return item with the highest *key* in tree if not *ceil* is given, else return item with highest *key*, but lower or equal to *ceil*. If *getindex* isn't given return a tuple, else return a dictionary with items' *index* and *item*.

Note: the `max(tree)` build in function also return the item with highest *key*, but it's encourage not to use it because it takes O(n) time. The method here described (e.g. `t.max()`) takes O(Log n) time.

`min([floor, getindex])`

Return item with the lowest *key* in *tree* if not *floor* is given, else return item with lowest *key*, but higher or equal to *floor*. If *getindex* isn't given return a tuple, else return a dictionary with items *index* and *item*.

Note: the `min(tree)` build in function also return the item with lowest *key*, but it's encourage not to use it because it takes O(n) time. The method here described (e.g. `t.min()`) takes O(Log n) time.

`pop(key[, default])`

If *key* is in tree, remove it and return its value, else return *default*. If *default* is not given and *key* is not in tree, a **`KeyError`** is raised. Performance: O(Log n) time if a deletion takes place, else O(1).

`popitem([index])`

If *index* is given remove and return the item in positional *index-th* in *tree*, else remove and return an arbitrary item from tree. Item is returned as a `(key, value)` pair tuple.

If an arbitrary item is selected, it is not a truly random decision. Instead chooses the cheapest one in terms of operations' costs. Indeed, some random decisions may take place in the process.

`popitem()`, with no *index* given, is useful to destructively iterate over a tree, as often used in set algorithms. If the tree is empty, calling **`popitem()`** raises a **`KeyError`**.

Selects item at positional *index-th* position if given *index* is positive, else selects item at *index-th* position from the last item*.

Examples:

`tree.popitem(i)`	==>	Remove and return item at the *i*-th position in *tree*
`tree.popitem(0)`	==>	Remove and return first item in *tree*
`tree.popitem(-1)`	==>	Remove and return last item in *tree*
`tree.popitem()`	==>	Remove and return an arbitrary item in *tree*

* Remember: items are arranged in keys' ascendant order.

`prune[key]`

Prunes the item of *tree* with key equal to *key* and return a new `bTree` object with that item in it. Raises a **`KeyError`** if *key* is not in tree.

Example: `new_tree = t.prune[key]`

`prune[start:stop:step]`

Prune the slice of *tree* from *start* to *stop* with a step *step*; and return a new `bTree` object with the slice items in it. The *start*, *stop* and *step* arguments are optional and the same extended slice notation rules apply. Arguments *start* and *stop* refer to keys not indices(*). *Step* must be an integer and its default value is `1`.

Example: `new_tree = t.prune[start:stop:step]`

(*) Given *start* and *stop* arguments not necessarily have to match existing keys in *tree*. When failed to match, the slicing takes place in all items within the range. However, *start* and *stop* value types must match `keys_value_type` attribute, otherwise raises a **`TypeError`**.

`prune([floor, ceil, step, slice, keys])`

Prunes a slice of *tree* and return a new `bTree` object with pruned items in it. The slice items are selected according the given optional arguments. Optional arguments usage and slice formation is explained in Appendix A.

If no arguments are given, prunes all items from tree, leaving it empty; and place them all into the new returned `bTree` object.

`replace(key[, value, notkey])`

Replace item's key and value for *key* and *value* and return *value* if *key* is in tree; else do nothing.

If *notkey* is given **`True`** and if an item's replacing takes place, only replaces item's value for *value* leaving item's *key* intact.

search(*key*[, *getall*])

Return `tree[key]` if *getall* is not given, else if *getall* is given **`True`** return item `(key, value)` pair and its positional index packed in a dictionary, whereas strings `'item'`, and `'index'` are their corresponding keys. Raises a **`KeyError`** if *key* isn't found.

setdefault(*key*[, *value*])

If *key* is in tree, return its value. If not, insert *key* with *value* and return *value*. If *value* is not given it defaults to **`None`**.

setnewkey(*keyfunc*[*keep_reject*])

Change or set a tree's *key function* attribute with *keyfunc*. The *key function* attribute is a function of one argument (equivalent to *key* in `sorted()` built-in function) that extracts a *comparison-key* from each tree's items' key (for example, `keyfunc=str.lower`). If *keyfunc* is not a real function raises a **`TypeError`**.

`setnewkey(`**`None`**`)` removes the *key function* attribute if tree has one, and the *comparison-keys* become the keys themselves.

A real change takes place when: 1) a *key function* attribute is set on a tree without one; 2) a new *key function* replaces an existing one; and 3) an existing *key function* is removed from a tree. When a real change occurs, items' nodes are re-arranged according to the new comparison keys in O(n Log n); and tree's `keys_value_type` attribute changes to match its new types.

Some items may be rejected in the process(1) and they are temporally stored by default unless *keep_reject* argument is explicitly given **`False`**(2).

Notes:

(1) Refer to the **key function** section in this tutorial.

(2) Refer to **Rejected Items** and **Rejected items by setnewkey** in this tutorial.

`update(`*`items`*`[ `*`keep_reject, notkey`*`])`

Update the tree with the items from *items*, overwriting[1] existing keys. Return **`None`**.

items can be any iterable object such like list, dictionary, tuple, set, tree, etc.[2] It can also be an iterator or generator[3] object. Whatever the nature of *items* is, it has to be a sequence of key/value list or tuple pairs; if they aren't, internal algorithms try to convert them into key/value pairs of data. Data items that failed conversion[4] are rejected.

The use of the `update()` method is by far more efficient than to importing items in a naive *for-loop* method[4].

If `keep_reject` is given **`True`**, rejected items are temporally stored in tree's object until retrieved with `get_rejected()` method. Items are rejected because internal algorithms failed to transformed them into key/value pairs or don't comply the rules for a valid tree's item[5].

If *notkey* is giving **`True`**, only replaces values when duplicates are found. *notkey*'s default value is **`False`**.

Performance: O(n Log n)

Notes:

(1) Overwrite item's key and value, unless *notkey* is given **`True`**.

(2) `strings`, `bytearrays` and `bytes` Iterable objects are not accepted.

(3) Warning: when using a generator this must stop generating at some point by raising a **`StopIteration`** error. Otherwise `update()` method will be stuck in an endless loop.

(4) Refer to **Importing Items From Other Data Structures** section to learn more about its efficacy and data items transformation processes.

(5) Refer to **Rejected Items** in documentation to learn more about them.

`values([`*`floor, ceil, step, slice, spiter`*`])`

Return a new view of the tree's values. Please refer to tree view objects operations.

5.2 Tree view objects' operations

These are the methods that return the supported tree view objects:

keys(**[***floor, ceil, step, slice, spiter***]**)

Return a new view of all *tree*'s keys if arguments `floor`, `ceil`, `step` and `slice` aren't given. If one or more of these arguments are given, return a view of items' keys of *tree*'s slice. Appendix A explains the arguments usage for slicing.

The optional *spiter* argument if given **`True`** turns *on* the launch of a **spiter**(*) next time a keys' iterator is requested. The ***spiter*** launching option is *off* by default.

items(**[***floor, ceil, step, slice, spiter***]**)

Return a new view of all *tree*'s items as `(key, value)` pair tuples, if arguments `floor`, `ceil`, `step` and `slice` aren't given. If one or more of these arguments are given, return a view of *tree*'s slice items. Appendix A explains the arguments usage for slicing.

The optional *spiter* argument if given **`True`** turns *on* the launch of a **spiter**(*) next time an items' iterator is requested. The **spiter** launching option is *off* by default.

values(**[***floor, ceil, step, slice, spiter***]**)

Return a new view of all *tree*'s values, if arguments `floor`, `ceil`, `step` and `slice` aren't given. If one or more of these arguments are given, return a view of items' values of *tree*'s slice. Appendix A explains the arguments usage for slicing.

The optional *spiter* argument if given **`True`** turns *on* the launch of a **spiter**(*) next time a values' iterator is requested. The ***spiter*** launching option is *off* by default.

(*) **Spiter** is the acronym of **spe**cial **iter**ator. **Spiter** is explained at the end of this section.

keys[start:stop:step]
items[start:stop:step]
values[start:stop:step]

Return their corresponding view of slice of tree from *start* to *stop* with a step *step*. Slice arguments *start* and *stop* refer to keys, not indices; and the same extended slice assignment notation rules apply; They have to match `keys_value_type` attribute, otherwise raises a **`TypeError`**. *Step* must be an integer different to `0`, and its default value is `1`.

Whether made with "parenthesis" or "brackets", these objects provide a dynamic view on the *tree*'s entries or its slice. Thus, they reflect changes if *tree* is modified.

Tree and slice views can be iterated over to yield their respective data, and support membership tests:

`len(view)`

Return the number of entries in the slice view of tree. If view wasn't created upon a slice, return the total number of entries in tree.

`x in view`

Return `True` if *x* is in the slice view of the underlying tree's keys, values or items (in the latter case, x should be a `(key, value)` tuple). If *view* wasn't created upon a slice, the underlying elements imply the whole tree.

`iter(view)`

Return an iterator (in items' keys ascendant order)(*) over the keys, values or items (represented as tuples of `(key, value)`) in the slice view of tree. If *view* wasn't created upon a slice, the iterator is over all tree's entries.

A **sp**ecial **iter**ator (aka **spiter**) return if previously the **spiter** option was set to *on*. If that's the case, the option is set back to *off* as soon as the **spiter** has been released. The **spiter** is explain later in this chapter.

`reversed(view)`

Return an iterator (in items' keys descendant order)(*) over the keys, values or items (represented as tuples of `(key, value)`) in the slice view of tree. If *view* wasn't created upon a slice, the iterator is over all tree's entries.

A **sp**ecial **iter**ator (aka **spiter**) return if previously the **spiter** option was set to *on*. If that's the case, the option is set back to *off* as soon as the **spiter** has been released. The **spiter** is explain later in this chapter.

(*) Keys, items and values are iterated in the order items are arranged in the tree. Items are arranged according to their *keys*, or *comparison-keys* if tree has a *key function* attribute.

By default, `iter()` and `reversed()` built-in functions return a dfs iterator object that uses a *depth first search* (aka dfs) technique to iterate over the tree's inner nodes.

Once a dfs iterator object has been created, tree should not be modified by adding or deleting entries. If so, it may raise a `RuntimeError` or give no trusted output. If there's a need of tree modifications while iterating, a **spiter** should be used instead.

Tree views have only one method:

`spiter([`*spiter*`])`

If *spiter* argument is given `True` or not given at all, turns *on* the launching of a **spiter** next time a tree view's iterator is requested; else turns *off* the **spiter** option. Examples:

`treeview.spiter()` or `treeview.spiter(True)`	==>	Set the **spiter** option *on*
`treeview.spiter(False)`	==>	Set the **spiter** option *off*

The **spiter** option can only be set to *on* if tree view is built upon a slice with a *step* parameter equal to `1` or `-1`; otherwise raises an `Exception` error.

5.3 Iterators and the goto() method

Iteration trend is given by the built-in function that issued the iterator, not the *step* parameter of the slice of the view:

Iterators issued by:	**always iterate in ... order over tree's entries**
`iter()`	***ascending***
`reversed()`	***descending***

E.g. the following iterator:

```
iter(tree.keys['shoe':'glove':-1])
```

...regardless the negative *step* slice parameter, iterates in ascending order from `'glove'` (not included) to `'shoe'` included.

Iterators are provided with a method that allow leaps while iterating tree's entries:

`goto([`*key*, *index*`])`

If *key* argument is given, the iterator **goes to** item with key equal to *key* and continues the iteration from this new position. If *key* isn't found in tree, iterator **goes to** item with the closest key value, but higher if iterator is in ascending mode (`iter()`) or lower if iterator is in descending mode (`reversed()`). Given *key*'s value type must match tree's `keys_value_type` attribute, otherwise raises a `KeyError`.

If *index* argument is given, the iterator **goes to** item in index position equal to *index* and continues iteration from this new position. **Note:** It goes to item in index position in tree, not in view. They may not coincide if view is created upon a slice of tree. *Index* sign and value follows the same convention of given index in other methods. If given positive, counts from the start in tree, being `0` the first one, if given negative, counts from the end, being `-1` the last one. *Index* if given must be an integer, otherwise raises a `TypeError`.

If both arguments are given, *key* is processed and *index* ignored.

The method return `True` if accomplishes a successful leap, whether given *key* or *index* argument. Else, return `False` (failed to accomplish a leap).

TIP: In order to use the `goto()` method inside a *for-loop*, the iterator should be created first.	`tk = t.keys()`
	`itk = iter(tk)`
	`for key in itk:`
	`    [...suit code...]`
	`    itk.goto(key=jump_key)`

5.4 The special iterator (aka SPITER)

Though a bit slower than dfs iterator, the **spiter** is an iterator object that allows tree modifications while iterating over its entries. Both iterators usage are the same (including leaps with `goto()` method), and all view objects support the **spiter** whether they are built over the entire's tree entries or upon a slice of it. In the latter case, the **spiter** is restricted to slices with a *step* parameter equal to `1` or `-1`.

All operations and methods that modify a tree (e.g. `insert()`, `del`, `update()`, `prune()`, etc.) are allowed when iterating with **spiter**; but the use of `setnewkey()` method is forbidden, because it changes the *comparison_keys*, which the **spiter** relies on.

Spiter is issued by the same `iter()` and `reversed()` built-in functions used for dfs iterators, but view's ***spiter*** option must be set to *on* right before these functions' callings. There are two ways to turn the **spiter** option *on*: 1) by explicitly given `True` the *spiter* argument at the view's creation, or 2) by calling the view's `spiter()` method without argument.

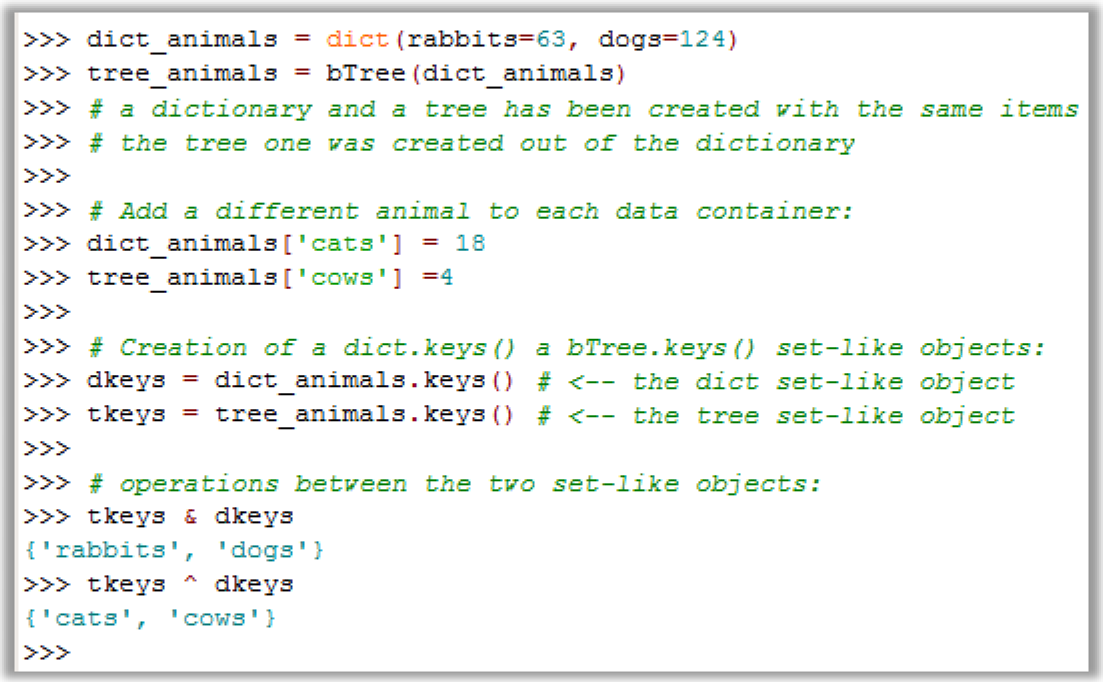
Examples:	

`treeview = tree.keys(spiter=True)`
`special_iter = iter(treeview)`
	`treeview = tree.keys['g':'h']`
	`treeview.spiter()`
	`for key in treeview:`
	`    [...suite code...]`

Once a **spiter** has been issued, view's ***spiter*** option turns *off* automatically. Thus, in order to re-launch again another **spiter** it must be turn *on* again right before its launching.

5.5 Set-like operations in tree views

Keys views are set-like since their entries are unique and hashable. If all values are hashable, so that `(key, value)` pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since their entries are generally not unique.) Set-like views inherit from abstract base class **`collections.abc.Set`**(*) all of its operations (for example, `==`, `<`, or `^`).



```
>>> dict_animals = dict(rabbits=63, dogs=124)
>>> tree_animals = bTree(dict_animals)
>>> # a dictionary and a tree has been created with the same items
>>> # the tree one was created out of the dictionary
>>>
>>> # Add a different animal to each data container:
>>> dict_animals['cats'] = 18
>>> tree_animals['cows'] =4
>>>
>>> # Creation of a dict.keys() a bTree.keys() set-like objects:
>>> dkeys = dict_animals.keys() # <-- the dict set-like object
>>> tkeys = tree_animals.keys() # <-- the tree set-like object
>>>
>>> # operations between the two set-like objects:
>>> tkeys & dkeys
{'rabbits', 'dogs'}
>>> tkeys ^ dkeys
{'cats', 'cows'}
>>>
```

(*) In Python versions 3.2 and before the SET module is `collections.Set`. Programmers should not worry about it as `bintree` module selects automatically the right module.

Operations with real set objects.

Set operations done with other set-like objects (e.g. dictionary views) perform perfectly, but some issues have been found with real set objects. Most of them have been solved by placing the real set object at the right of the operation:

`treeview & {'dogs', 'cats'}`	==> Perform the operation
`{'dogs', 'cats'} & treeview`	==> Set object raises a **`TypeError`**.

However, some operations, like `^` for example, doesn't perform even though real set object is placed at the right side. To address this issue equivalent real set object's built-in methods should be used. E. g.:

use => `set_obj.intersection(treeview)` *instead of* => `set_obj & treeview`

In future versions of this API, tree views will support set operations.

Appendix A: Slicing definitions with optional arguments

Methods: `clear()`, `copy()` and `prune()` and views: `keys()`, `items()` and `values()` when given the following optional arguments: **[** *`floor, ceil, step, slice, keys`* [*] **]** perform an operation or launch a view respectively over a slice of *tree*. Slices are the same and they are formed when one or more arguments are given.

[*] **Note:** Views don't support *`keys`* optional argument.

Arguments:

- **`[floor, ceil, step]`**: These three optional arguments define a slice alike (not equal to) the extended `[start:stop:step]` notation does. *`Floor`* and *`ceil`* refer to keys not indices, therefore their values must match the `keys_value_type` attribute, otherwise raises a **`KeyError`**. When given, the defined slice selects all items with key *k* such *`floor`* `<=` *k* `<=` *`ceil`*. Note that both ends (*`floor`* and *`ceil`*) are included if they are found in *tree*. The *`step`* argument, if given must be an integer different to `0`; otherwise raises a **`TypeError`** or a **`ValueError`** respectively. When *`step`* isn't given its default value is `1`, which means that slice includes all items within the range. When a different *`step`* value *s* is given, it means that selects an item every *s* items. If *s* > 0 starts selecting the *`floor`* (if found in *tree* or the immediately higher item) and goes upwards until reaching or surpassing *`ceil`*. If *s* < 0 starts selecting the *`ceil`* (if found in *tree* or the immediately lower item) and goes downwards until reaching or surpassing *`floor`*. Note that regardless *`step`* sign *`floor`* always must be lower than *`ceil`*, otherwise the resulting slice is empty.

  Examples:

  There's a tree *t* with 12 items with the following *keys*:
  `['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']`

Operation	**Result:**
`t.keys(floor='c', ceil='f')`	A keys' view with : `'c'`, `'d'`, `'e'` and `'f'`. Note that `'c'` and `'d'` are included, because given `floor` and `ceil` are found in *t*.
`t.keys(floor='cf', ceil='ft')`	A keys view with : `'d'`, `'e'` and `'f'`. `Floor` and `ceil` aren't found in *t*, thus the view shows only those keys within their range.
`t.clear(floor='c', ceil='j', step=3)`	Deletes from *t* the items whose keys are: `'c'`, `'f'` and `'i'`. The operation started in `'c'` and went upwards because `step` > 0.
`t.copy(floor='c', ceil='j', step=-3)`	Return a new tree with copies of items whose keys are: `'j'`, `'g'` and `'d'`. The operation started in `'j'` and went downwards because `step` < 0

- `[slice]`: This optional argument gives the possibility to define a slice using indices instead of keys. Items in a `bTree` object are always arranged in ascending order according to their keys' values. Therefore, each item occupy a unique *i-th* positional index inside a *tree*, unless a change occurs in *tree*. In order for this argument to take effect, the previous arguments (*`floor`*, *`ceil`* and *`step`*) must not be given. If any of them are given, the *`slice`* argument is ignored. The *`slice`* argument can be given in the form of a slice, tuple or an integer type. All inner values (whether it's a slice or a tuple) must be integers and no more than three: *start*, *stop* and *step*. The same rules of Extended Slices notation apply.

  Examples:

Operation	**Result:**

`t.items(slice=slice(18))`
`t.items(slice=(18))`
	Return an items view with the 18 first items of `t`.

`t.prune(slice=slice(-100, None))`
`t.prune(slice=(-100, None))`
	Cuts from t the last 100 items and return a new `bTree` object with those items.

`t.copy(slice=slice(100, 500, 5))`
`t.copy(slice=(100, 500, 5))`
	Copies one of every five items, from the 100th one to the 500th; and return a new `bTree` object with the copies in it. This is nice a sample taking example.

  Observation: Two equivalent forms have been presented in each example. The first ones take a real slice object as given value for the *`slice`* argument. However, the alternatives take either an integer or a tuple. It's intended in the examples above to show that alternatives are valid shortcuts against the use of slice objects; and by design they textually "look" similar but with one "**`slice`**" word less.

- `[keys]`: This last argument only supported by `clear()`, `copy()` and `prune()` methods, is just a list of items' keys where to apply the methods' operations. In order for this argument to take effect, all previous arguments (*`floor`*, *`ceil`*, *`step`* and *`slice`*) must not be given. If any of them are given, the *`keys`* argument is ignored. Example:

`t.copy(keys=['b', 'f', 'c'])`	Copies items whose keys are `'b'`, `'c'` and `'f'`.

  Keys in *keys* not necessarily must be in ascending order; and keys that aren't found in *tree* are just simply ignored.

Standard Slice Assignment:

Slicing methods here presented are complimentary to the standard slice assignment for mutable sequence types found in Python. Usually in notation `[start:stop:step]`, the optional arguments *start* and *stop* in sequences refer to indices, but in the case of `bTree` methods and operations they refer to keys. Thus if *step* value is positive, the slice consists of all items with keys equal and greater to *start* and lower (but not equal) to *stop*; else the other way around. The *step* argument, if given, must be an integer different to zero.

Examples of slicing with keys and indices:

`t.items['c':'d']`	Return a view of all items with keys that start with the letter 'c'
`t.items(slice=(100, 200))`	Return a view of all items located between the 100th positional index (included) and the 200th positional index (not included)

Some operations have more than one expression forms. For example:

Operation:	**Is equivalent to:**
`t.copy()`	`t[:]`
`t.prune(step=-10)`	`t.prune(slice=(None, None, -10))`
	`t.prune[::-10]`

A final note:

All operations can be performed with naive *for-loop* methods, which demand a lot more code. The following are two examples to perform the same pruning operation:

Using *for-loops*:	**Using the correct method:**

```
tp = bTree()
for key in keys:
      try:
            value = t.pop(key)
      except KeyError:
            continue
      else:
            tp[key] = value
```
	`tp = t.prune(keys=keys)`
	
	This one line of **elegant** code is **easier** to use and has a **faster** execution. Please use the correct methods here provided!

Appendix B: Quick tutorial reference: *...for programmers in a hurry!*

Construction:

`from bintree import bTree`	`bTree` object is the only object needed. This line should be at the beginning of the code.
`t = bTree()`	Creates a new tree `t`. The key of the first item to insert in `t` set the type all keys in `t` must match. (i)
`t = bTree(mapping)`	Creates a new tree `t` from *mapping*, which must have an `items()` method that return a sequence of key/value pairs.(ii)
`t = bTree(sequence)`	Creates a new tree `t` from *sequence*: [(k1, v1), (k2, v2), (k3, v3), ...(kn, vn)].(ii) & (iii)
`t = bTree(typ=class_type)`	Creates a new tree `t`. `t` is set to accept only items with keys of type *class_type*.(i)
`t = bTree(keyfunc=keyf)`	Creates a new tree `t`, with a key function *keyf*.(iv)

i. This is the `keys_value_type` attribute that all keys in tree must match.
ii. If *items* have more than one type of key, `bTree` object select only those of the most predominant type.
iii. Items with duplicate keys, if arise, replace old ones.
iv. The *key function* is a function of one argument that is used to extract a *comparison-key* from each item's key (for example, `keyfunc=str.lower`).

Operations and Methods:

`t[key] = value`	Insert an item (*key* and *value*) into tree `t`. If *key* is already found in `t`, the new item replaces the old.
`t[key]`	Return the *value* of the item with key equal to *key*.
`del t[key]`	Removes the item with key *key* from `t`.
`len(t)`	Return the number of entries of `t`.
`t.copy()`	Return a shallow copy of `t`.
`t.clear()`	Remove all items from `t`.
`t.setnewkey(keyf)`	Changes or set a new key function *keyf*(iv) in `t`. Items then are re-arranged according to the new *comparison-keys*.
`t.min([floor])`	If *floor* is given, return the item with lowest key, but equal or higher than *floor*; else the item with lowest key of `t`. O(Log n)
`t.max([ceil])`	If *ceil* is given, return the item with highest key, but equal or lower than *ceil*; else the item with highest key of `t`. O(Log n)
`t.index(key)`	If *key* is found in `t`, return the *i-th* positional index ok *key* in `t`; else raise a **`KeyError`**.

`t.item(index)`	Return the item, as a `(key, value)` tuple, located at *i-th* positional *index* in `t`.
`t.pop(key[, default])`	If *key* is in `t` remove its item and return its value; else return *default* if given, else raise a **`KeyError`**.
`t.popitem([index])`	If *index* is given, remove and return the item in positional index *index-th*; else remove and return an arbitrary item from `t`.
`t.update(mapping)`	Import items from *mapping* which must have an `items()` method that return a sequence of key/value pairs.
`t.update(sequence)`	Import items from *sequence* [(k1, v1), (k2, v2), (k3, v3),... (kn, vn)].

i. This is the `keys_value_type` attribute that all keys in tree must match.
ii. If *items* have more than one type of key, `bTree` object select only those of the most predominant type.
iii. Items with duplicate keys, if arise, replace old ones.
iv. The *key function* is a function of one argument that is used to extract a *comparison-key* from each item's key (for example, `keyfunc=str.lower`).

(*) There are more methods and special features. Please check the Tutorial and Operations section.

Operations and Methods with slices:

`t[start:stop]`
`del t[start:stop]`
`t.prune[start:stop]`
	Respectively copy, remove and prune items of `t` from `start` to `stop`.(i)

`t.copy([...arguments...])`
`t.clear([...arguments...])`
`t.prune([...arguments...])`
	Respectively copy, remove and prune a slice of `t` defined by optional given arguments.(ii)

i. Standard slice assignment notation: All items from `start` (included) to `stop` (not included). Slice arguments `start` and `stop` refer to keys, not indices.
ii. Slice of `t` is defined by the optional given arguments. Refer to Appendix A about slicing.

Tree views:

`t.keys()`
`t.items()`
`t.values()`
	Respectively return a view object of all *keys*, *items* and *values* of `t`.

`t.keys[start:stop]`
`t.items[start:stop]`
`t.values[start:stop]`
	Respectively return a view object of the *keys*, *items* and *values* from `start` to `stop`.(i)

`t.keys([...arguments...])`
`t.items([...arguments...])`
`t.values([...arguments...])`
	Respectively return a view object of the *keys*, *items* and *values* of a slice of `t` defined by given arguments.(ii)

i. Standard slice assignment notation: All items from `start` (included) to `stop` (not included). Slice arguments `start` and `stop` refer to keys, not indices.
ii. Slice of `t` is defined by the optional given arguments. Refer to Appendix A about slicing.

Operations with views:

View objects are not containers, but lazy sequences of all tree's entries or a slice of it. Views are dynamic so they reflect changes in the slice of the underlying tree. They can be iterated over, support membership tests and set-like operations like `==`, `<` or `^` are available.

`len(view)`	Return the number of tree entries shown by the view.
`x in view`	Return `True` if *x* is in the tree view sequence.
`iter(view)`	Return an iterator in **ascending** order over the tree view sequence.(i)
`reversed(view)`	Return an iterator in **descending** order over the tree view sequence.(i)
`view.spiter()`	Turns on view's **spiter** option.(ii)

i. Depending on the `bTree` method used to issue a view, elements of the tree view sequence can be *keys*, *values* or *items* (represented as tuples `(key, value)` in tree.

ii. While iterating a view, tree shouldn't be modified. However, there's a **sp**ecial **iter**ator (aka spiter) design to be used when tree modifications are needed while iterating its entries. More about the spiter at **Tree view objects** section.

Operations with iterators:

An iterator is the object that `iter(view)` and `reversed(view)` return. Both type of iterators -the default one (dfs) and the **spiter** - have one method:

`iterator.goto(key=key)`	Leaps the iterator to the item with key *key* and continues the iteration from the new position.(i)
`iterator.goto(index=index)`	Leaps the iterator to the item in index positional index and continues the iteration from the new position.

i. If an item with a key *key* isn't found in the view of the slice of tree, the new position is the one with the closest key. It looks for a higher key if iterator was returned by `iter()` built-in function, and lower if it was returned by `reversed()`.

Appendix C: Operations performances:

Operation	O(1)	O(Log n)	O(n)	O(n Log n)
`bTree()`	X			
`bTree(items)`				X
`len(t)`	X			
`tree[k]`	X			
`tree[k] = v`	if replacement	if new item insertion		
`del t[k]`		X		
`k in t`	X			
`k not in t`	X			
`iter(t)`	X			
`reversed(t)`	X			
`tree1 == tree2`			At most if equal	
`clear()`	X			
`clear([with args])`			X	
`copy()`			X	
`prune()`			X	
`get(k)`	X			
`get_rejected()`	X			
`haskey()`	X			
`index(k)`		X		
`insert(k)`	if replacement	if new item insertion		
`item(index)`		X		
`items()`	X			
`keys()`	X			
`values()`	X			
`keystype()`	X			
`max(ceil)`	if *ceil* in t	X		
`min(floor)`	if *floor* in t	X		
`pop(k)`		X		
`popitem(i)`		X		
`replace(k)`	X			
`search(k, getall)`	x	If *getall* given True		
`setdefault(k, v])`	If not item insertion	If new item insertion		
`setnewkey(keyfunc)`				X
`update(items)`				X
`len(treeview)`	X			
`x in t.keys()`	X			
`x in t.items()`	X			
`x in t.values()`			X	

Appendix D: TIPS and examples

Throughout this document, emphasis has been placed to use the proper methods (`update()` and `bTree()`) to import items from other data structures. *For-loops* work fine, but not as fast. In case an external source must have some process before passing the items, generators or iterators should be used. Let's see some examples:

1. With given *items* make a BST, but first depurate them. Some items may not go:

```
# depurate is an already existing function that depurates an item.
# It will return a new and depurated item if succesful, else
# return nothing (None)

def depur_items(items, depurate):
    for item in items:
        new_item = depurate(item)
        if new_item == None: continue
        yield new_item

t = bTree(depur_items(items, depurate), typ=classType)
```

   There are of course many ways to solve the problem, but what is important is that *items* in `bTree(items)` is replaced by a generator function that depurates and filter out *items*. When knowing in advance the `keys_value_type` attribute of tree, the process can be accelerated by explicitly giving the optional *typ* argument.

2. There's a csv external file containing the items:

   data file example: items.csv

```
dog,canid
cat,feline
horse,equine
```

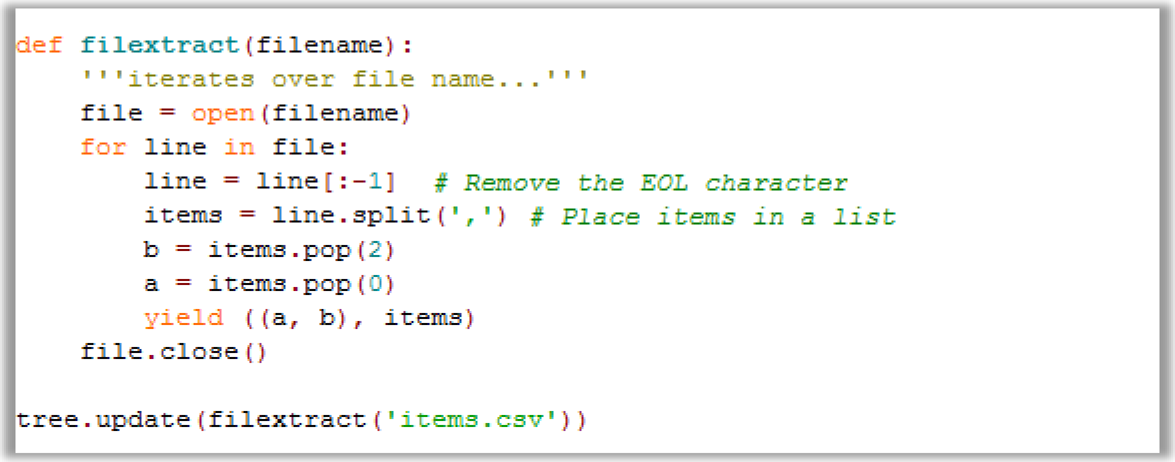
   The code to build a tree upon this data should be:

```
from bintree import bTree

def filextract(filename):
    '''iterates over filename and return a list of its items.'''
    file = open(filename)
    for line in file:
        line = line[:-1]        # Removes the EOL character
        yield line.split(',')  # Yield a list of comma separated items
    file.close()

tree = bTree(filextract('items.csv'))
```

3. Let's import data to an existing tree from an external file containing thousands of lines with several columns. Items in the first and third column should be packed in a tuple to be used as keys:



```python
def filextract(filename):
    '''iterates over file name...'''
    file = open(filename)
    for line in file:
        line = line[:-1]  # Remove the EOL character
        items = line.split(',') # Place items in a list
        b = items.pop(2)
        a = items.pop(0)
        yield ((a, b), items)
    file.close()

tree.update(filextract('items.csv'))
```

Only for GEEKS. Why *for-loops* are slower than proper methods?

Every time an insertion or a deletion occurs, all ancestor nodes up to the root has to be checked. Unbalanced nodes, if found, are automatically balanced through rotations. The process time for inserting or deleting a few nodes is insignificant, but repeating the same process over a large amount of data, individual processing times add up. Consider the following example: a string of items are being inserted in a naive *for-loop*. The first items are inserted over a small tree and the self balancing process is quite fast, but as the tree grows, so the self balancing time process too for each new inserted node. As an alternative, what if all items are known in advanced and placed in an array in sorted order. A balanced binary search tree is easily formed with an algorithm called quicktree (which is designed originally from an idea borrowed from quicksort). This method is used by `bTree(items)` and in experiments done over 100 thousand items, took only 16% of the time compared to the naive *for-loop* insertion method.

There's a situation quite similar when deleting a bunch of items. The naive *for-loop* repeats the same process over and over giving as a result a poor performance. Therefore, the `t.clear(keys=keys)` single line of code customizes the process and avoid unnecessary repetitions. In an experiment done in a one million items' tree, 200 thousand items was chosen at random to delete. The proper method took only 24% of the time of the naive *for-loop*'s choice.

Other proper methods that replace naive *for-loops* are: importing items to a tree (`t.update(items)`) and deleting, copying and pruning slices of a tree. All of them have highly customized algorithms that improve hugely their performances.

Other TIPS

- ### Membership in a tree with a *Key Function*

  A key request membership (`key in tree`) in a tree with a *key function* may not be accurate, because different keys may yield the same comparison key. The precise way is by using a view:

```
key in tree.keys()
```

  However, the line has an inherent view's creation and that involves a process. Therefore, if requests are going to happen many times, it's better to build once the view and use it over and over as it dynamically reflects tree's entries.

```
keys = tree.keys()
[...suite code...]
key in keys
```

- ### The use of `min()` and `max()` methods for special searches

  Sometimes there may be a need to search for the closest item to a certain key value in tree. Visualizing the tree as an array of items, this can be divided in two blocks. The dividing point (a *pivot*) can be a key value that may or may not exist in tree; but all items at the left are lower to and all items at the right, higher. Given optional *floor* and *ceil* arguments to `min()` and `max()` methods respectively serve as such described *pivot* point. Both methods respond equivalent to `t.search(pivot, True)` if *pivot* is found in tree. Otherwise, returns the closest item to it as the following table explains:

		In other words:
`t.min(floor=pivot)`	Returns item with key closest to *pivot*, but higher.	Returns the item with the **lowest** key value of the block at the **right** of the pivot.
`t.max(ceil=pivot)`	Returns item with key closest to *pivot*, but lower.	Returns the item with the **highest** key value of the block at the **left** of the pivot.