Instructions for MA2

This document contains the material for the second module (MA2) in the course Computer Programming II. This module is an introduction to data structures using Python. We introduce some details about classes and other topics like *operator overloading*, *iterators* and *generators*.

The exercises for the module can be found in the yellow frames below. There are **14** exercises in this module. Each group of exercises follows a related theory and a set of examples.

The following files are associated with the module:

1. `linked_list.py` in which you shall write the solutions to linked list exercises,
2. `bst.py` in which you shall write the solutions to binary tree exercises, and
3. a number of test files with the name `test_LinkedList_`*`name`* or `test_BST_`*`name`* where *`name`* is the name of the function in the class `LinkedList` and `BST` respectively that is tested.
4. There is tests.sh file to run all tests from terminal.

Follow the instruction below when preparing your solutions for presenting:

1. The tasks must be solved in the order they come and with the tools (classes, functions, methods, ...) that are introduced before the exercise.
2. Unless otherwise stated, you may not use packages other than those already included in the downloaded files.
3. The functions must be named and return exactly what is stated in the task.
4. Your code may not use any global variables.
5. Tasks to be answered with text should be written as comments or text strings at the end of the code document. The answers can be kept short.
6. Present your solutions to a teacher or assistant in the class or a Zoom room. Be prepared to answer questions about the complexity of the methods.
7. When your orally presented solutions are approved:
   (a) Fill in the name, email, assistants / teachers who reviewed the assignment and the date of the review the documents.
   (b) Make a zip file containing `linked_list.py` and `bst.py` as well as test files and upload into Studium.

Note: You may collaborate with other students, but you must write and be able to explain your own code. You may not copy code, neither from other students nor from the Internet except from the places explicitly pointed out in this lesson. Changing variable names and similar modifications does not count as writing your own code. Since the assignments are part of the examination, we are obliged to report failures to follow these rules.

1 Data structure

A *data structure* is a way of organizing a collection of values. Examples of such collections include a car register, a family tree, a table with distances between different places etc. We call the individual objects (i.e. cars, people, places, ...) *records* or, sometimes, `elements`.

We will discuss different operations on the structures and analyse how efficient they are Examples of such operations:

- Enter new records.
- Remove records.
- Search records for specific contents.
- Visit all records.

Some common data structures:

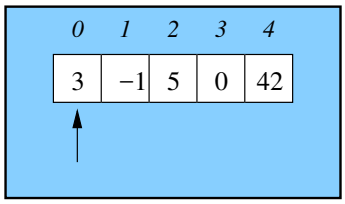
- Arrays.
- Linear linked lists.
- Trees.
- General linked structures (graphs).
- Hash tables.

Arrays

This is the most fundamental data structure due to the fact that it is closely related to the memory organization in computers.

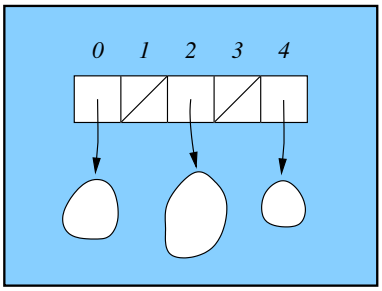
Typical features of arrays:

- all elements are of the same data type (e.g. integer) which means that they take up the same amount of space and
- all elements are stored in a contiguous memory area.



These properties mean that the address of any element can be easily calculated if you know the address of the first element (in place 0) and the element size.

If the elements are of different sizes, we can store pointers to the elements (which gives us an array of pointers)

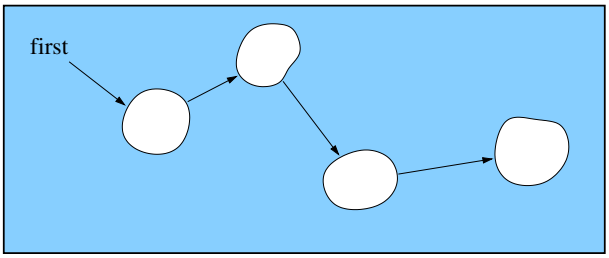


The main advantage of arrays is the fast index operation ($\Theta(1)$)

A disadvantage is that it is more difficult to add a new element at a given index because all subsequent elements must be moved ($\Theta(n)$).

Linear linked lists

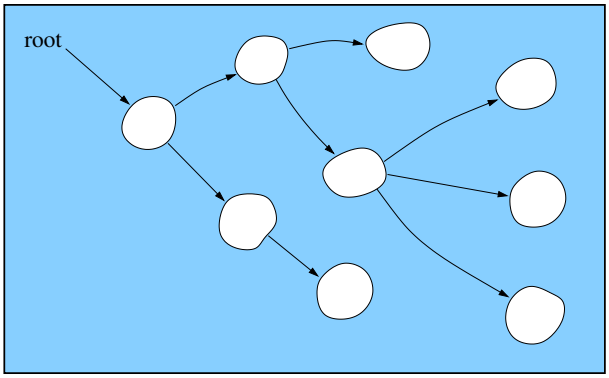
Instead of storing the records in a contiguous memory, we can let them point to each other i.e. a record can contain *the address of* the next record.



The advantage of this is that it is possible to link in new elements anywhere without moving already stored elements. One drawback is that the index operation requires $\Theta(n)$ operations because we have to start from the beginning and follow pointers until we come to the elements with the sought index.

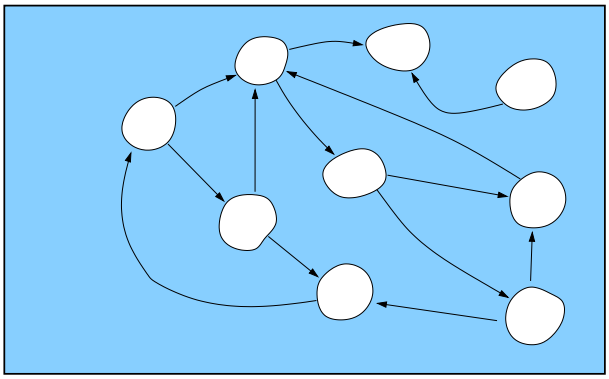
Trees

In linked lists, each record has (at most) one successor. If you allow a record to have multiple followers, we get *tree structure*:



Graphs

If a record can have several relations both backwards and forwards and also circularly, we call the structure a graph:



Hash table

A hash table is a way of implementing a mapping from a set of keys to a set of values such that each element in the key set relates to one specific element in the value set.

Data structures in Python

The basic data structure in Python is the *list*. A list is implemented as an array of references to the record thus allowing fast indexing ($\Theta(1)$). Since lists can be expanded, the array must be *dynamic* i.e. it can grow when needed.

The *tuple* data type is similar to list, but since it is unchangeable (cannot be expanded), the operations are more efficient.

The *dict* data type represents pairs of keys and values. It is implemented with hash technique.

Many packages and modules define other data structures. For example, the NumPy package defines *multidimensional arrays* and the Panda package defines *dataframes*.

2 Linked lists

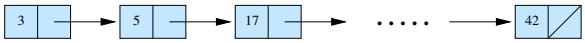
Python's built-in, array-based, lists are extremely powerful and often the best option for general lists. There are still reasons to study how to make linked lists. It's an easy way to illustrate techniques that handle all sorts of dynamic structures — not just linear lists.

In addition, it is a good exercise in recursive thinking which can be applied to more complicated structures such as trees and graphs.

We will exemplify the technique with lists of integers as data in the nodes. We keep the lists sorted so that the data parts come in order of size.

Using integers as data imposes no limitation. The only thing that is important is that the data parts are comparable value-wise.

A linked list with integers can be illustrated as follows:



We represent the individual elements in the list as objects from the class `Node`:

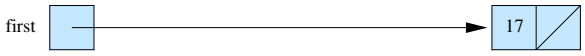
```
1 class Node:
2     def __init__(self, data, succ):
3         self.data = data
4         self.succ = succ
```

(In this situation we don't care about providing the instance variables "protection" i.e. to start the names with `__` for the variables in the class.)

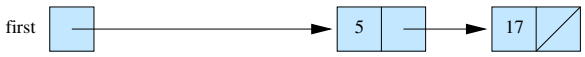
To keep track of the list, you need a reference to the first element.

Here is an illustration of the effect of three lines of code:

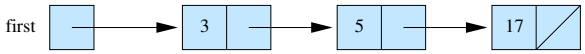
```
first = Node(17, None)
```




```
first = Node(5, first)
```




```
first = Node(3, first)
```

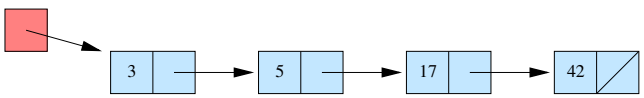


Now we can iterate over the list and, for example, sum the stored numbers:

```
1 summa = 0
2 f = first
3 while f:
4     summa += f.data
5     f = f.succ
6 print('Summa:', summa)
```

The code will produce the line `Summa: 25`

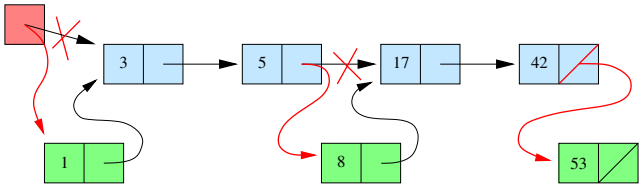
Now suppose we want to write code that adds a new value to a list keeping it sorted.



There are four different cases:

1. In an empty list.
2. As the first element. For example inserting 1
3. As the last element. For example inserting 53
4. In the inner. For example inserting 8

We have to find the pointer to be changed, i.e. either the pointer to the first node or the pointer in the node that will be *immediately in front* the new node:



Assume that the value we want to insert is in the variable `x`. The code can then be written:

```
if first == None or x <= first.data:    # case 1 and 2
    first = Node(x, first)
else:
    f = first
    while f.succ != None and x > f.succ.data:
        f = f.succ
    f.succ = Node(x, f.succ)            # case 3 and 4

```

In case the value already exists, we store the new value in front of the old value. If we want to store the new value after the old value, the code becomes a little more complicated. (If only integers are stored, it is unimportant how equal numbers are stored *but* if it is an object with several properties then the insertion order can be important.)

It is a special case if the new value is to be placed first in the list, either depending on the list being empty or the new value being less than the first element. In those cases, the variable `first` must be changed.

If the new value is not to be the new first element, we use a `while`-loop to find the node that should come *immediately in front* of the new node.

Remember that the order is important — you must first know that there is a node present before you can look at its contents!

A class for the list

It is convenient to put the linked list stuff in its own class. Sketch:

```
1  class LinkedList:
2      class Node:
3          def __init__(self, data, succ):
4              self.data = data
5              self.succ = succ
6
7      def __init__(self):
8          self.first = None
9
10     def print(self):    # Unnecessary but serves as a simple example
11         pass
12
13     def insert(self, data):
14         pass
15
```

The class `Node` is the same as before, but it is now placed inside the class — it has no use outside `LinkedList`.

The class `LinkedList` has only one instance variable: the first node (or, more precisely, a reference to the first node). From the beginning it is set to `None` i.e. the list is empty.

Insertion in the list

We place the code for insertion, which was previously described, in a method:

```
1  def insert(self, x):
2      if self.first is None or x <= self.first.data:
3          self.first = LinkedList.Node(x, self.first)
4      else:
5          f = self.first
6          while f.succ and x > f.succ.data:
7              f = f.succ
8          f.succ = LinkedList.Node(x, f.succ)
```

Since the class `Node` is an attribute in the class, we must refer to it with `LinkedList.Node`. Here we write `f.succ` as the first condition in the loop on line 6. In a condition, `None` is interpreted as `False` so this is just a shorter way to write the same thing.

A simple print method

As an example of a simpler method than the `insert` method, we show a method for printing lists. It is actually unnecessary because prints could be made using the method `__str__` but it serves as a simple example.

First attempt:

```
def print(self):
    f = self.first
    while f:
        print(f.data)
        f = f.succ
```

This gives one value per line, which is probably not the way you want to see the list.

With a little more effort, we can get all values on the same row, separated by comma and the whole list enclosed by parentheses:

```
def print(self):
    print('(', end='')
    f = self.first
    while f:
        print(f.data, end='')
        f = f.succ
        if f:
            print(', ', end='')
    print(')')

```

The code that we have discussed so far is in the downloaded liinked_list.py

Exercise 1: Method `length`

Write a method `length(self)` which, using a `while`-loop, calculates and returns the number of nodes in the list. Original list must not be destroyed.

Exercise 2: Method `remove_last`

Write a method which removes the last node from the list. The method should return the value in the deleted node.
If the list is empty, a `ValueError` should be raised. (`ValueError` is a standard exception class in Python.)

Exercise 3: Method `remove`

Write a method `remove(self, x)` which deletes the first node containing `x` as data. If the method finds a node with this content, it should return `True` else `False`.
Use iteration here. We will discuss how to use recursion here soon.

3 Recursive list methods

Linked structures are well-suited for recursive methods. This is because a list can be defined recursively in this way:

A list is
either
empty (consists of 0 elements)
or
consists of one element followed by a list.

We will now show the method `length` recursively instead of iteratively as in the exercise above. This can be done in two ways.

Method 1: With a help method in the `Node` class

We write a method in the *node class* which tells you how long the list is that starts with that node:

```
1  class Node:
2      ...
3      def length(self):
4          if self.succ is None:
5              return 1
6          else:
7              return 1 + self.succ.length()
8      ...
```

Thus, if this node has no successor, the length is 1. Otherwise it is 1 *plus* the length of the list beginning with the successor.

Then, in the class `LinkedList` we write the required method:

```
1   def length(self):
2       if self.first is None:
3           return 0
4       else:
5           return self.first.length()
```

We have to start by checking if there are any nodes in the list at all.

Method 2: Using an auxiliary method in the `LinkedList` class

In the second way we do nothing in the `Node` class but we write an internal method `_length(self, f)` in the `LinkedList` class that returns the number of nodes in the list that begins with `f` and use it in the externally callable function:

```
1  def length(self):
2      return self._length(self.first)
3
4  def _length(self, f):
5      if f is None:
6          return 0
7      else:
8          return 1 + self._length(f.succ)
```

The auxillary method can also be placed as a local function in `length`:

```
1  def length(self):
2
3      def _length(f):
4          if f is None:
5              return 0
6          else:
7              return 1 + _length(f.succ)
8
9      return _length(self.first)
```

Since it is now not a method in the class, it has no `self` as a parameter or `self.` before the call.

The choice between method 1 and method 2 is a matter of taste. Since method 2 often gives fewer special cases, the code becomes shorter, especially when it comes to trees. However, these methods may be more difficult to understand.

Recursive `insert` using method 1

Here we will write an `insert` in the `Node` class. When in a node, it is not possible to change the pointer *to* the node but only the pointer *in* the node. Just as in the iterative variant, we must therefore stop at the node which is to be immediately in front the new node. This means that we should only proceed in the recursion if the following node for sure will be in front of the new node. Thus, we get the following code in the `Node` object:

```
1  def insert(self, x):
2      if self.succ is None or x <= self.succ.data:
3          self.succ = LinkedList.Node(x, self.succ)
4      else:
5          self.succ.insert(x)
```

The new node is created on line 3.

The method in the main class must first check that the new node is not in the first place,

i.e. the code must ensure that there is at least one element in the list and that this element should come before the new node.

It returns the following code `LinkedList`:

```
1 def insert(self, x):
2     if self.first is None or x <= self.first.data:
3         self.first = LinkedList.Node(x, self.first)
4     else:
5         self.first.insert(x)
```

A bit ugly to have to repeat basically the same code ...

Recursive `insert` using method 2

In method 2, we place an auxiliary method in the main class. The help method gets a reference to the first node and returns the first node in the modified list.

```
1  def insert(self, x):
2      self.first = self._insert(x, self.first)
3
4  def _insert(self, x, f):
5      if f is None or x <= f.data:
6          return LinkedList.Node(x, f)    # Return the new node
7      else:
8          f.succ = self._insert(x, f.succ)
9          return f                 # Return the same node we got in
```

Note how the help method `_insert` is called on line 2! The new node can become the first in the list.

The help method works as follows:

It gets a reference to the first node in a sublist and returns the first node in the modified sublist. If the new node has become the first in the sublist, that is the one which is returned, but if it ended up further away then the "old" first node is returned (line 9).

The nice thing about this method is that we do not have to handle a new first node or an empty list as special cases.

The next exercise should be solved with recursion instead of iteration. Go on working with the code you downloaded earlier.

Exercise 4: The method `to_list`.
Write a recursive method `to_list (self)` which returns a standard Python list with the values from the linked list in the same order.

3.1 Iterators and generators for the class `LinkedList`

To process all elements in many of the common structures in Python (lists, tuples, strings, files ...) we can use a `for` statement. It is said that these structures are *iterable*. For a structure to be iterable, there must be a method `__iter__` which returns an *iterator*.

An iterator is an object which keeps track of a *current element*. The class must include a constructor and two methods.

An iterator for objects of class `LinkedList` could look like this:

```
class LinkedListIterator:
    def __init__(self, lst):
        self.current = lst.first

    def __iter__(self):
        return self

    def __next__(self):
        if self.current:
            res = self.current.data
            self.current = self.current.succ
            return res
        else:
            raise StopIteration
```

The constructor (the method `__init__`) receives (a reference to) a `LinkedList` object and saves a reference to its first node in the instance variable `current`, the method `__next__` returns the value in the current element and advances `current` to the next item in the list.

If the last element has been passed, the exception `StopIteration` is raised.

The method `__iter__` just returns itself.

If this class exists, we can for example write:

```
it = LinkedListIterator(lst)
for x in it:
    print(x)
```

So the `for`-statement in Python uses exactly these methods.

You can also iterate over the list with a `while` statement which uses the methods in `LinkedListIterator`:

```
it = LinkedListIterator(lst)
while True:
    try:
        print(next(it))
    except StopIteration:
        break
```

which shows how the `for`-statement is implemented.

Note that `next(it)` is a nicer way to write `it.__next__()`. If you want to try this iterator you can download `linked_list_iterator.py` (optional).

Instead of making a separate iterator class, you can provide the class `LinkedList` with the

methods `__iter__` and `__next__`.

The method `__iter__` is then given the task of creating the `current` variable:

```
def __iter__(self):
    self.current = self.first
    return self
```

Note that this method thus adds an instance variable to the class!

The method `__next__` is completely unchanged.

Now we can iterate over linked lists in the same way as we iterate over ordinary lists:

```
for x in lst:
    print(x)
```

This can be achieved even easier by making `__iter__` as a *generator*:

```
def __iter__(self):
    current = self.first
    while current:
        yield current.data
        current = current.succ
```

The Python command `yield` will act as a `return` statement *but* next time the generator is called, it will continue where it left off. We do not need to write any `__next__` method or throw any `StopIteration` — these are created automatically.

This generator is included in the code you downloaded from the beginning.

More about iterators and generators can be found on YouTube, for example in iterators and generators by Corey Schafer.

3.2 Operator overloading

For standard lists we can use the operator `in` and, for example, write expressions like `if x in list:`. We can also introduce this in our own list class by defining the method `__in__`:

```
def __in__(self, x):
    for d in self:
        if d == x:
            return True
        elif x < d:
            return False # No point in searching more
    return False
```

This code is included in the class you downloaded in the beginning.

Note how we can use our own class generator in the `for` statement.

In a similar way, we can define other operators such as `<`, `<=`, `==`, ... by defining the methods `__lt__`, `__le__`, `__eq__`, ...

More about special methods and operator overloading can be found, for example, in Corey Schaefer's YouTube-lesson.

Exercise 5: The method `__str__`.

Use the iterator or generator to write the method `__str__(self)` which returns the list as a string of comma-separated values with parentheses surrounding the list.
Examples of resulting strings: `()`, `(1)`, `(1, 2)`. Note that the result must have *exactly* that appearance, i.e. with commas followed by spaces *between* every value.

Exercise 6: The method `copy`.

The `copy`-method in the downloaded code looks like this:
It creates and returns a copy of the `LinkedList` object.

```
1 def copy(self):
2     result = LinkedList()
3     for x in self:
4         result.insert(x)
5     return result
```

What is the complexity of this method?
Rewrite the code so it gets a better complexity! Note that the copy may not share any list nodes with the original!
What is the complexity of your method?

Exercise 7: A Person class in `LinkedList`.

Suppose you have the class:

```
1 class Person:
2     def __init__(self, name, pnr):
3         self.name = name
4         self.pnr = nr
```

1. Create an empty list `plist` in `main` and then a person object `p`.
2. Insert the person object into the list using the list `insert` method. Print `plist`. This should work well.
3. Create another person object `q` and insert it with `insert`. The program will be aborted in the `insert` method on line
   `if f is None or x <= f.data:`
   The diagnosis is
   `TypeError: '<=' not supported between instances of 'Person' and 'Person'`

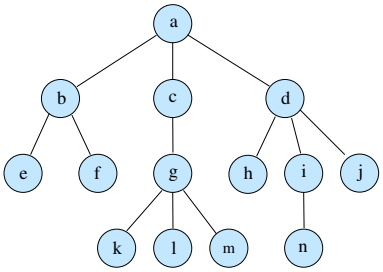
The problem is that Python doesn't know how to compare two person objects in terms of size — `insert` is supposed to make sure that the list is sorted in order of size.
To be able to use relational operators such as `<`, `<=`, `==` ... between two objects the methods `__lt__`, `__le__`, `__eq__`, ... must exist.
Write the methods in the `Person` class that are needed to be able to insert such objects into the `LinkedList`.
Note that you *not* should change anything in the `LinkedList` class but define comparison operators in the class `Person`.
Choose whether `Person` objects should be sorted by name or number.

4 Tree structures

Definitions and terminology

In a list, each node has exactly one successor (except the last) and one predecessor (except the first).

If we allow a node to have multiple followers, we get a *tree structure*:



The terminology is taken from both the forest and the family:

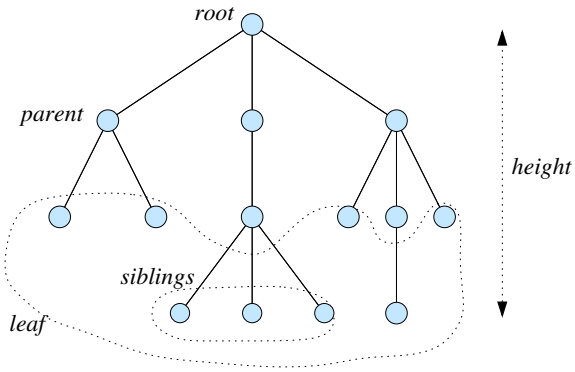
A *parent* is a node that has one or more successors (*children*).

A *child* is a node that has a predecessor (*parent*).

Nodes with the same parent are called *siblings*. The siblings come in order of age with the oldest furthest to the left.

A leaf is a node without children.

All nodes except the *root* have a parent.



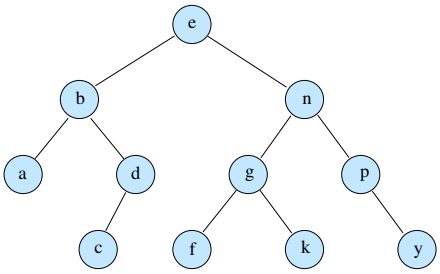
The tree *height* is the largest number of nodes on the way between the root and a leaf. The height of the above tree is thus 4.

Sometimes the height is defined as the number of 'arcs' between the root and a leaf. With this definition, the height in the tree above is 3. It does not matter much which definition you use, as long as you are consistent.

A *binary tree* is a set of nodes that is either empty or consists of three subsets:

1. one with just one node – *root*,
2. one subset called the *left subtree* and
3. one subset called the *right subtree*.

The two sub-trees are in turn binary trees.



Note the difference between a binary tree and an ordered tree with a maximum of two children per node:

1. An ordered tree has at least one node while a binary tree can be empty.
2. In a binary tree, there is a difference between having an empty right subtree and an empty left subtree — no such difference exists for ordered trees.

Tree operations

There are a number of operations you may want to perform on trees depending on the application:

- *Traverse* i.e. visit all the nodes (in some order).
- *Search* for a node with a certain content or in a certain position.
- *Insert* and *delete* nodes.
- *Merge* trees.
- Compute various measurements (height, width, number of nodes, ...).

Here we only discuss algorithms for traversals.

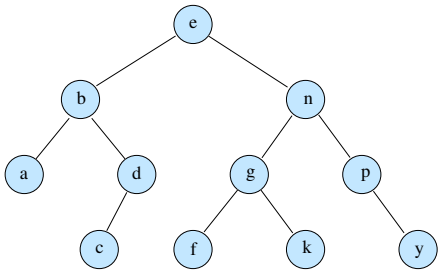
Other operations are discussed in the coding examples and the assignments.

Tree traversals

Traversing a structure means visiting all the elements in some particular order. For trees, there are mainly four possible schemes: *preorder*, *inorder*, *postorder* and *level order*.

Preorder:

1. Visit the root
2. Visit the children in *preorder* (with the oldest first)



On the example tree, the order will be: e b a d c n g f k p y

Postorder:

1. Visit the children in *postorder* (with the oldest first)
2. Visit the root

On the example tree, the order will be: a c d b f k g y p n e

Inorder (for binary trees):

1. Visit the left subtree in *inorder*
2. Visit the root
3. Visit the right subtree in *inorder*

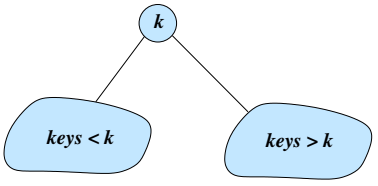
On the example tree, the order will be: a b c d e f g k n p y

Binary search trees

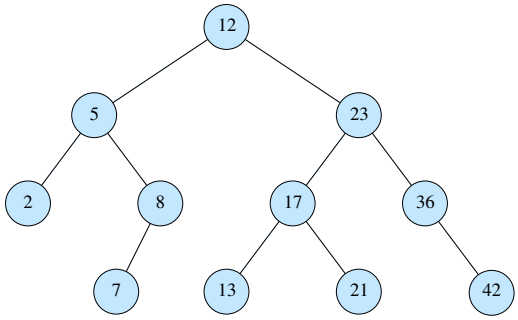
Binary *search trees* are an important application of binary trees. In such a tree, each node has a *key* which uniquely identifies the node. A key can be, for example, a social security number, a vehicle registration number or a word. Typically, then, there are no two nodes with the same key.

The keys must have an order relationship defined, i.e. one should be able to compare the keys in size.

The key in a node must be larger than all keys in the node's left subtree and less than all keys in the node's right subtree:

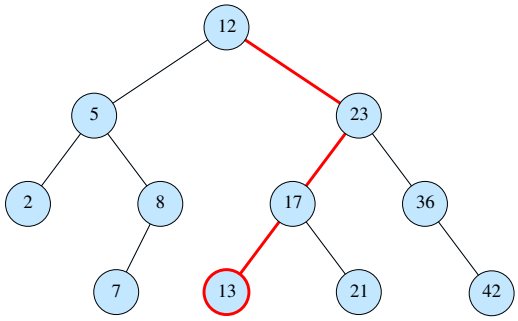


Example: A binary search tree with integers as keys:

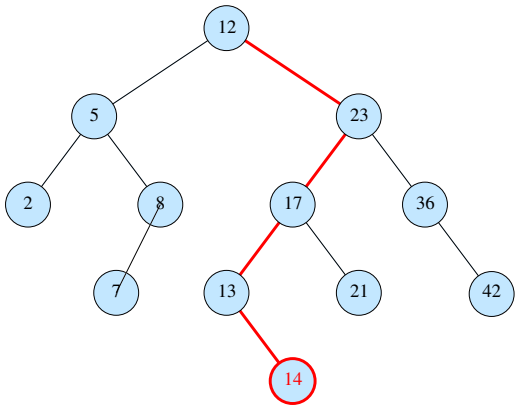


To search for a node with a given key, we start from the root. The order relation can then guide the search down to the requested node.

For example, to find `13`, go to the right at `12`, to the left at `23` and `17` since `13` is greater than `12` but less than `23` and `17`:



The same is done when inserting new keys: start from the root and let the nodes steer in which direction one should go. There is always exactly one place that a new key should go into. The figure shows insertion of the key `14`:



The work of both searching and inserting keys in a binary search tree is limited upwards by *the height* of the tree.

Representation and algorithms for binary search trees

Here we will build a structure for storing nodes in binary search trees. In the code, we name the stored records `keys` and use integers as an example, but all types of data that have the order relations defined works well. We do not allow the same key to occur multiple times.

We want to be able to create trees, add new records, search for certain records and print the contents of the tree.

```
class BST:

    class Node:
        def __init__(self, key):
            self.key = key
            self.left = None
            self.right = None

    def __init__(self):
    """ Create an empty tree"""
        self.root = None

    def insert(self, key):
    """ Insert a new key in the tree"""
        pass

    def contains(self, k):
            """Check i k is in the tree"""
        pass

    def print(self):
    """ Print the contents of the tree"""
        pass

```

The method `print` is included from the beginning to be able to easily print trees but it will later be replaced by the more general `__str__` method.

The tree thus consists of node objects where each node contains a key as well as references the left and right subtrees.

The tree object has only the instance variable `root` which is initialized to `None` i.e. the tree is empty from the beginning.

We start by looking at the method `contains`:

Note that the search is controlled by the relationship between the searched value and the contents of the node.
Iteration is ongoing as long as `n` is not `None` and the key is not in `n`.
If `n` is not `None` when the loop ends, the value has been found.

```
    def contains(self, k):
        n = self.root
        while n and n.key != k:
            if k < n.key:
                n = n.left
            else:
                n = n.right
        return n is not None
```

We have written this method iteratively. In many cases, however, it is much easier to write tree methods recursively, which the `print` method is an example of.

The method prints the tree on one line with the nodes sorted by size.

```
    def print(self):
        self._print(self.root)

    def _print(self, r):
        if r:
            self._print(r.left)
            print(r.key, end=' ')
            self._print(r.right)

```

This pattern with an auxiliary method (which gets a reference to a node to work with) and one main method (which only starts the help method by submitting the root node) will be used in many tree methods.

Unlike the `print` method, the `insert` method is easy to write iteratively, but we do it recursively anyway.

```
    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, r, key):
        if r is None:
            return self.Node(key)
        elif key < r.key:
            r.left = self._insert(r.left, key)
        elif key > r.key:
            r.right = self._insert(r.right, key)
        else:
            pass # Already there
    return r
```

We also showed this technique as 'method 2' in the section on linked lists, i.e. the help method *returns* a reference to the root node in the modified the subtree. The nice thing about this technique is that we do not have to differentiate between the cases of a completely empty tree, the new node being a left child and the new node being a right child.

We show another method which calculates the number of nodes. It uses the same recursive technique with a help method:

```
    def size(self):
        return self._size(self.root)

    def _size(self, r):
        if r is None:
            return 0
        else:
            return 1 + self._size(r.left) + self._size(r.right)
```

A generator

In the `LinkedList` class we presented a generator to be able to iterate over the nodes. The keyword there was `yield`. We will now look at a generator for our BST class which goes

over the nodes in *inorder*, meaning order of magnitude.

We start by writing a generator for the `Node` class:

```
class Node:
    ... # Same as before
    def __iter__(self):
        if self.left:              # If there is a node to the left
            for n in self.left:    # use its generator
                yield n.key        # to return node references
        yield self                 # Then return ourself
        if self.right:             # Then the nodes to the right
            for n in self.right:
                yield n

```

It is recursive in the sense that a generator in a node uses a generator in other nodes.

Note that we need to check that there is a node before we try using its generator.

This can be written a little more compactly using the construction `yield form`:

```
class Node:
    ...
    def __iter__(self):
        if self.left:
            yield from self.left
        yield self.key
        if self.right:
            yield from self.right

```

Now we can easily make a generator for the tree:

```
def __iter__(self):
    if self.root:
        yield from self.root
```

Deletion of nodes in binary search trees

Removing a key from a search tree is a little more difficult. If the key is in a node with no or just one child, there is no problem. It is basically the same thing as removing nodes from a linked list. The problem is removing keys that are in nodes with two children because we then do not know how to keep track of the node's children.

The usual technique is to keep the physical node but replace the content (key) with another key that is to be kept in the tree. Since we have to maintain the search tree property (small on the left, large on the right), there are only two possible keys to move up to the node where the key to be removed is: the *largest* among the small ones or the *smallest* among the large. Both of these have the property of at most one child — otherwise they are not the biggest or the smallest. Which of them you choose does not matter.

We give here a pseudocode for the operation:

```
1  def remove(self, key):
2      self.root = self._remove(self.root, key)
3
4  def _remove(self, r, key):
5      if r is None:
6          return None
7      elif k < r.key:
8          r.left = # left subtree with k removed
9      elif k > r.key:
10         r.right = # right subtree with k removed
11     else: # This is the key to be removed
12         if r.left is None:      # Easy case
13             return r.right
14         elif r.right is None:  # Also easy case
15             return r.left
16         else: # This is the tricky case.
17                         # Find the smallest key in the right subtree
18                         # Put that key in this node
19                         # Remove that key from the right subtree
20     return r # Remember this! It applies to some of the cases above
```

Note that there is recursion on lines 8, 10 and 19.

Exercise 8: A method `contains`

Write the `contains` method with recursion instead of iteration.
Note that `contains` is checked in the `main`.
There is no test file for this function, but you may first write it for the existing function `contains` and then test on your own function.

Exercise 9: Compute the height.

Write a method `height` which returns the height of the tree. The empty tree has height 0, the tree with 1 node has height 1, the tree with 2 nodes has height 2 while the tree with 3 nodes have a height of 2 or 3 depending on the order of insertion.

Exercise 10: A `__str__`-method.

The generator code is included in the downloaded file. Use it to write a `__str__` method! The elements in the string should be *separated* by commas followed by a space and the string should be surrounded by `'<'` and `'>'`.
Examples of results: `'<>'`, `'<1>'` and `'<1, 2>'`.

Exercise 11: A `to_list`-method.

Write the method `to_list` which creates and returns a standard Python list with the values from the tree. What is the complexity?

Exercise 12: A `to_LinkedList`-method.

Write the method `to_LinkedList` that creates and returns a `LinkedList` with the values from the tree. Complexity of your algorithm should be linear.

Exercise 13: Remove a key.

Complete the method `remove` according to the given pseudocode.

Exercise 14: For which operations is the generator suitable?

You have used the tree generator (i.e. the `__iter__` method) for implementing the `__str__` method. For which of the following methods could the tree generator be used in the implementation?

1. `size`
2. `height`
3. `contains`
4. `insert`
5. `remove`