Modul 3: Data structures

This lesson is an introduction to data structures using Python.

We also introduces some new details about classes and some other topics like *operator overloading*, *iterators* and *generators*.

The module contains a number of exercises, some of which are mandatory and must be solved and presented. However, we recommend that you do *all* of the exercises and discuss them with an assistant if there is something you are unsure about!

Instructions

1. (a) `linked_list.py` in which you shall write the solutions to linked list exercises,
   (b) `linked_list_test.py` with unit tests for linked lists,
   (c) `bst.py` in which you shall write the solutions to binary tree exercises, and
   (d) `bst_test.py` with unit tests for binary trees.
2. The tasks must be solved in the order they come and with the tools (classes, functions, methods, ...) that are introduced before the exercise.
3. Unless otherwise stated, you may not use other package than those already included in the downloaded files.
4. The functions must be named and return exactly what is stated in the task.
5. Write your own test code in the `main` functions. When you think that your solutions work, run the downloaded test files.

   Note that the unit tests of one task may be based on the solutions to other tasks. Thus, there is no point in trying run unit tests before you have solved all the mandatory tasks.
6. Sections and tasks marked with an asterisk (*) are optional and need not to be read or done.
7. 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.
8. Present your solutions to a teacher or assistant in the Zoom Room. Be prepared to answer questions about the complexity of the methods.
9. 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 in both documents.
   (b) Upload the two files in Studium under OU3.

Note: The course material is new! Please email errors and ambiguities to tom.smedsaas@it.uu.se.

Observera: 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.

Introduction

A *data structure* is a way of organizing a collection of values. Examples of such collections are 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 with a specified contents.
- Visit all records.

Some common data structures:

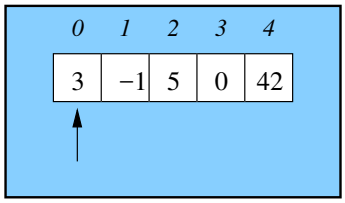
- Arrays.
- Linear linked lists.
- Trees.
- General linked structures (graphs)
- Hash tables.

Arrays

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

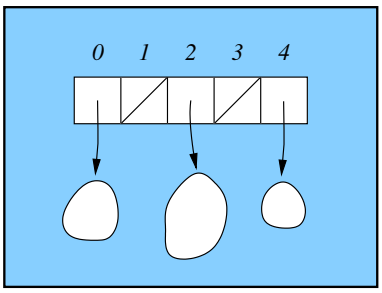
Typical features of arrays:

- 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)

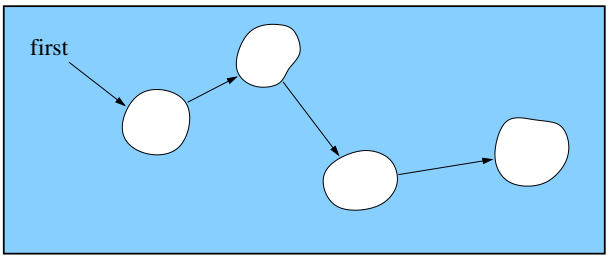


The main advantage of arrays is the fast index operation ($\Theta(1)$)

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

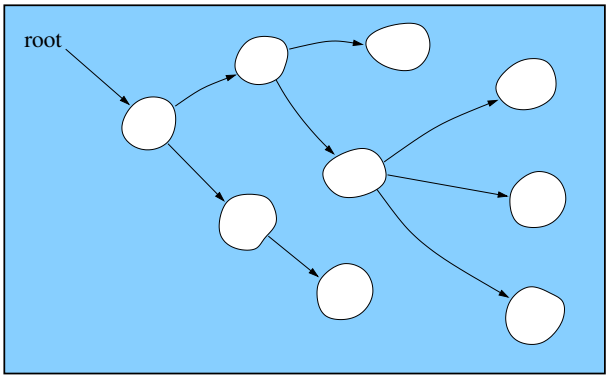
Instead of storing the records in a contiguous memory, we can let them point to each other i.e. a record can contain *the address of* the next record.



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 elements with the sought index.

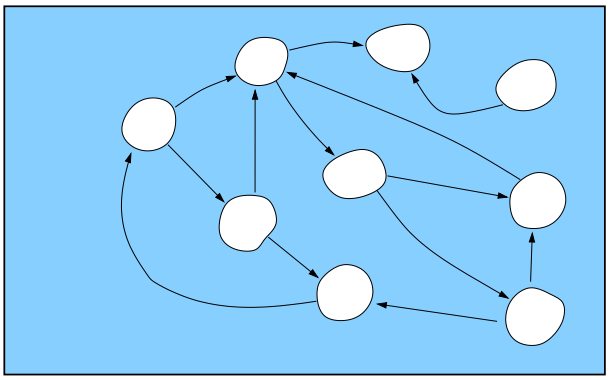
Trees

In linked lists, each record has (at most) one successor. If you allow an 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:



Hashtabell

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 a *multidimensional arrays* and the Panda package defines *dataframes*.

Linked structures in Python

Linked lists

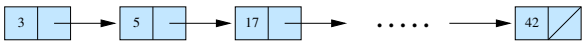
Python's built-in, array-based, lists are extremely powerful and often the best option for general lists. There are still reasons to study how to make linked lists. It's an easy way to illustrate techniques to handle all sorts of dynamic structures — not just linear lists.

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`:

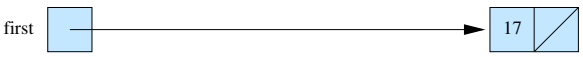
```
1 class Node:
2     def __init__(self, data, succ):
3         self.data = data
4         self.succ = succ
```

(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.)

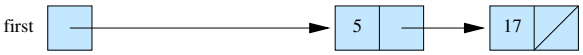
To keep track of the list, you need a reference to the first element.

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

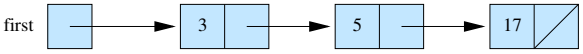
```
first = Node(17, None)
```




```
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)
```

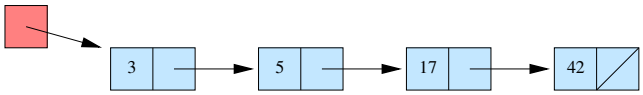
The code will produce the line `Summa: 25`

Question: What happens if we forget the line `f = f.succ`?

Question: Do we need the variable `f`? Can't you just use `first` in the `while`-loopen?

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

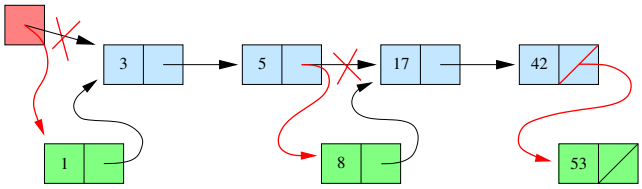
Sample list:



Different cases:

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

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

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

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

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
```

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)
```

Two details:

1. Since the class `Node` is an attribute in the class, we must refer to it with `LinkedList.Node`
2. In the first version we wrote `f.succ != None` but 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, end=' ')
        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 disussed so far is in the downloaded linked_list.py.

Exercise 1:

Write a method `length(self)` which, using a `while`-loop calculates and returns the number of nodes in the list.

Exercise 2:

Write a method which calculates and returns the mean of the numbers in the list.

Exercise 3:

Write a method which removes the last node from the list. The method should return the value in the deleted node.

What should you do if the list is empty?

Exercise 4: Metoden `remove` Mandatory!

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`.

Write suitable test cases that convince you that the method works according to the instructions!

Use iteration here! We will discuss how to use recursion in the next section.

Recursive list methods

As mentioned in the introduction, 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 that 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` (compare the example with memorization when calculating Fibonacci numbers).

Which way you prefer 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 should not be 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 following three exercises should be solved with recursion instead of iteration. Go on working with the code you downloaded earlier.

Exercise 5: The method `count`

Write a recursive method `count (x)` which counts how many times `x` is in the list. We recommend method 2, i.e. with an auxiliary method in the class `LinkedList`.

Exercise 6: The method `to_list`. Mandatory!

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

Exercise 7: The method `remove_all`. Mandatory!

Write a recursive method `remove_all(self, x)` which removes *all* nodes which contains `x` as data.

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*. To 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.

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 from 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__`, ...

Exercise 8: The method `__str__`. Mandatory!

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 9: The method `copy`. Mandatory!

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!

What is the complexity of your method?

Exercise 10: The index operator. Mandatory!

In regular lists, you can use an index to reach an element at a certain position. Implement the index operator for linked lists!

Hint: Search the Internet to see how to define the index operator!

Note that you do not want to be able to use the index operator to *change* the value because that may destroy the sorting property!

Exercise 11: A Person class in `LinkedList`. Mandatory!

Suppose you have the class:

```
class Person:
    def __init__(self, name, pnr):
        self.name = name
        self.pnr = nr
```

Write the methods in the class `Person` that are needed for it to be possible to enter such objects in `LinkedList`. Note that you should not change anything in the `LinkedList` class. Instead define comparison operators in the class `Person`. Decide for yourself if `Person` objects should be sorted by name or personal number.

Hint: You have to write methods with names like `__lt__`, `__le__`, ...

The class is located in the downloaded code immediately before the `main` function. It is a very illogical location but is chosen for management reasons. How should it really be done?

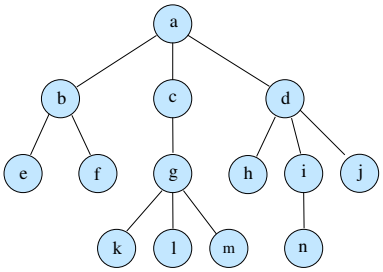
More about special methods and operator overloading can be found, for example, in Corey Schaefer's YouTube-lektion.

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:

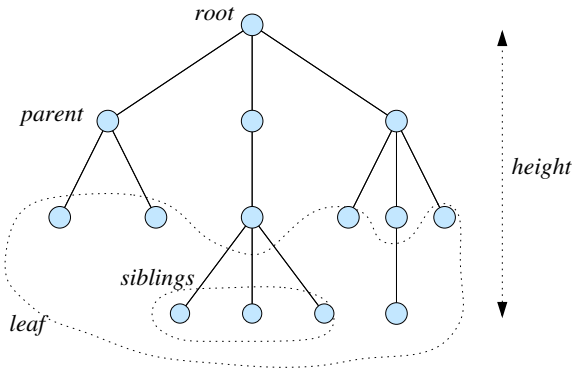
A *parent* is a node that has one or more successors (*children*).

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 nodesexcept the *root* has a parent.



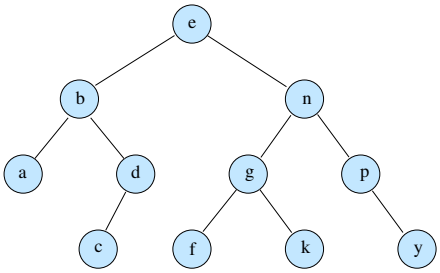
The tree *height* is the largest number of nodes on the way between the root and a leaf. The height of the above trees is thus 4.

Note: Sometimes the height is defined as the number of ärcsbetween the root and a leaf. With this definition, the height becomes 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 – textitroot,
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 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.

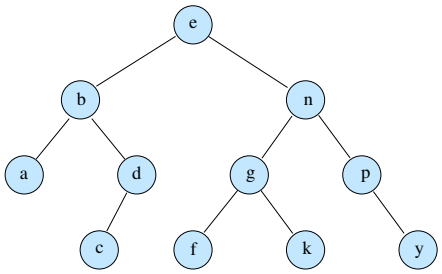
Other operations are discussed in the coding examples and the assignments.

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 subtreei in *inorder*
2. Visit the root
3. Visit the right subtree in *inorder*

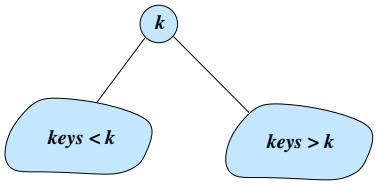
On the example tree, the order will be: a b c d e f g k n p y

Binary search trees

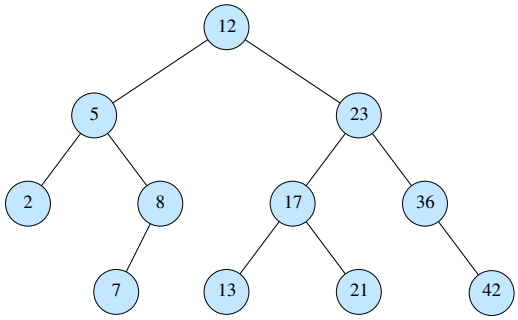
Binary *searchtrees* is an important application of binary trees. In such a tree, each node has a *key* which uniquely identifies the node. A key can be, for example, a social security number, a vehicle registration number or a word. Typically, then, there are no two nodes with the same key.

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:

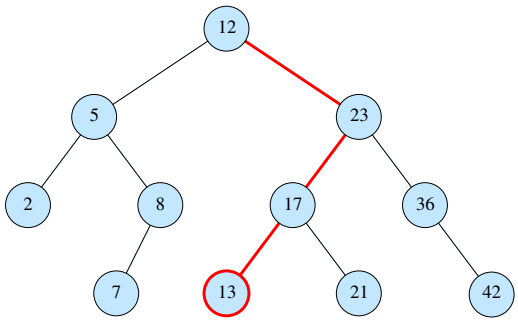


Example: A binary search tree with integers as keys:

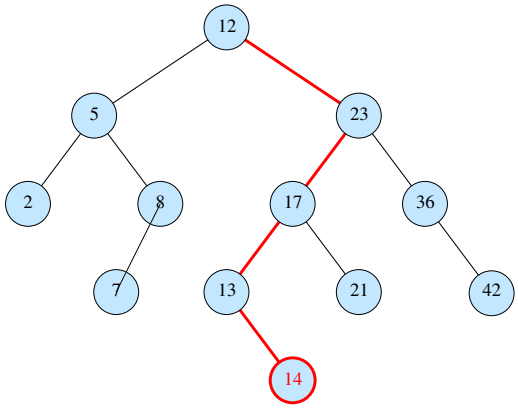


To search for a node with a given key, we start from the root. The order relation can then guide the search down to the requested node.

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`:



Observation: 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.

At a minimum, 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.

To think about: How to write this method without recursion?

Unlike the `print`, 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 completely empty tree, the new node is a left child and the new node is a right child.

Study the code and make sure you understand it!

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)
```

Do the following exercises in bst.py which you downloaded in the beginning.

Exercise 12:

Write the `insert` method without using recursion.

Exercise 13:

Write the `contains` method with recursion instead of iteration.

Exercise 14: Compute the height. Mandatory!

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.

Deletion of nodes in binary searchtrees

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's basically the same thing as removing nodes from a linked list. The problem is to remove 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 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 15: Remove a key. Mandatory!

Implement the above method according to the given pseudocode.

Also write test cases which test each branch of the code.

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
```

Exercise 16: A `__str__`-method. Mandatory!

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 17: A `to_list`-method. Mandatory!

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

Exercise 18: A `to_LinkedList`-method. Mandatory!

Write the method `to_LinkedList` that creates and returns a `LinkedList` with the values from the tree. What is the complexity?

Properties of binary trees

A binary tree with k filled levels contains n nodes there

$$n = 1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$$

dvs.

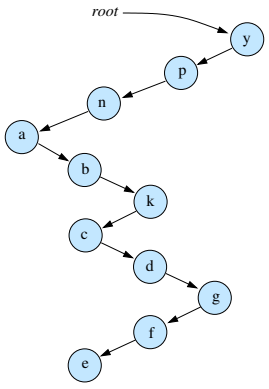
$$k = \log_2(n+1) \approx \log_2 n$$

This is the *minimum height of a binary tree with n nodes.*

In such a binary search tree, the work for

- searching is $\mathcal{O}(\log n)$
- insertion is $\mathcal{O}(\log n)$
- deletion is $\mathcal{O}(\log n)$

The largest possible height is obtained if each node has only one child:



In such a binary search tree, we have at least in the worst case

- searching is $\Theta(n)$
- insertion is $\Theta(n)$
- deletion is $\Theta(n)$

What about on average?

Above we have discussed the height (and thus the work) in the best and worst case. More interesting is however, what to expect on average.

We would like to know the average work to search (add, delete) a key in an äveragetree.

The following three measures are used to analyze how good trees are for search / insert / delete:

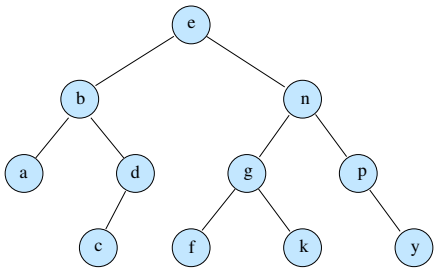
- *height*: for worst case analysis

- *internal path length*: for successful search analysis
- *external path length*: for unsuccessful search analysis

The *internal path length* (*ipl* or I) is defined as the sum of all nodes' *levels*

The root has level 1, the root's child level 2, etc.

Example:



Adjacent trees have internal path length

$$i = 1 + 2 \cdot 2 + 4 \cdot 3 + 4 \cdot 4 = 33$$

The average work to find a stored key ("a successful search") is

$$\frac{i}{n} = \frac{33}{11} = 3$$

The *external path length* (*epl* or E) is defined as the sum of the levels of the places where a new node would end up.

In the example tree above, it will be

$$E = 4 + 4 + 5 + 5 + 4 + 5 + 5 + 5 + 5 + 4 + 5 + 5 = 56.$$

Inserting a new key requires on average $E/(n+1)$ node visits (there are $n+1$ possible locations and we count the creation of the node as a visit).

The following relationship between internal and external road length applies

$$E = I + 2n + 1$$

which can easily be shown with induction (exercise!).

I can be shown (see for example Genomsnittligt sökdjup i binära sökträd * in Swedish) that the average internal path length for all trees formed by all $n!$ permutations of n keys is

$$1.39 \cdot n \log_2 n + \mathcal{O}(n)$$

Thus, to find a key in the "average tree" takes on average $1.39 \cdot \log_2 n + \mathcal{O}(1)$ operations.

For example, searching a tree with 10^6 nodes thus requires on the order of approximately $1.39 \cdot \log_2(10^6) \approx 28$ node visits. The corresponding search in a linked list would require $500,000$ node visits.

Insertions and deletions can also be made with $\mathcal{O}(\log(n))$ operations.

Thus: A binary search tree is an efficient structure for *storing*, *searching* and *deleting* nodes with keys. Since it also maintains a sorting of the keys, operations such as *find the smallest* and *find largest* are efficient.

However: There are many insertion orders that lead to so called *degenerated* trees i.e. trees where almost no nodes have two children. See e.g. the figure at the top of the previous page. The operations on the tree then have the same complexity as the corresponding operations on a list i.e. typically $\Theta(n)$. This will for example happen if you insert the keys in sorted

order. The logarithmic complexity applies to the äverage tree".

There are balancing mechanisms that ensure that the trees do not degenerate i.e. they *guarantee* $\mathcal{O}(\log n)$ for the operations. The insertion algorithms are modified so that they adjust the tree if it becomes too unbalanced. See for example AVL-trees* and Red-black trees *. In these, all the operations we have discussed are $\mathcal{O}(\log n)$ - even in the worst case.

Exercise 19: The method `ipl`. Mandatory!

Write the method `ipl (self)` which calculates and returns the internal path length as defined above.

Exercise 20: Verify the theory with experiments. Mandatory!

Experimentally verify that the internal path length (IPL) grows as $1.39n\log_2(n) + O(n)$ in trees built up with random numbers. At the same time, try to see how the height seems to depend on the number of nodes.

Hint: Write a function (not a method) `random_tree(n)` which creates and returns a tree with `n` random numbers. Use `random.random()` which returns random numbers (floats) in the range $[0, 1)$. Then write the code that examines trees of different sizes at the end of the `main` function. Print the observed height and $\frac{\text{IPL}}{n}$. How well does that agree with the theory? What can you guess about the height?

Exercise 21: For which operations is the generator suitable? Mandatory!

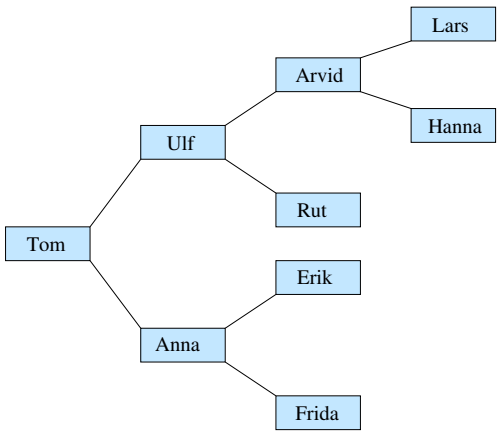
Earlier (exercise 16) you have used the tree generator (i.e. the `__iter__` method) for implementing the `__str__` method. For which of the following methods could the tree generator be used in the implementation?

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

Other examples of trees

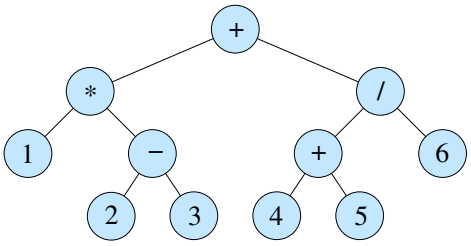
Example: Parent tree

A tree that shows a person's ancestors is a binary tree but not a search tree:



Example: Representation of arithmetic expressions

$1 * (2 - 3) + (4 + 5)/6$



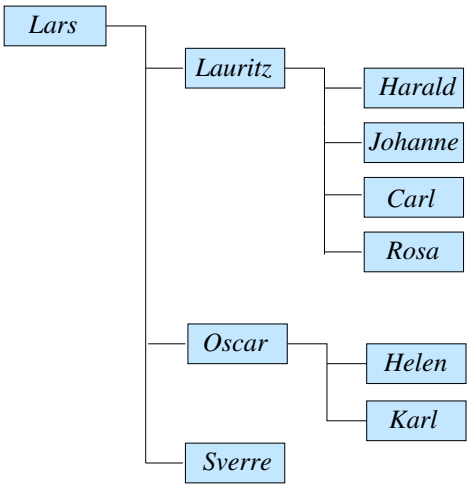
Note: The tree shows the evaluation order. No parentheses needed.

The value is computed recursively (a post order traversal)

1. Compute the value of the left subtree.
2. Compute the value of the right subtree.
3. Perform the operation on the two calculated values

Example: Family tree

A family tree that lists a person's descendants in age order is an example of one *orderded* tree:



A family tree could be based on the following node:

```
class Person:
    __init__(self):
        self._mother = None
        self._father = None
        self._children =[]  #Ordinary list with Person-object
```

and, of course, with more personal information (name, date of birth, ...)

Hash tables

This section is a brief introduction to hash tables. We only introduce so-called *linked* collision management. The aim is just to present the concept of hash tables to get some understanding of their properties. In Python, dictionaries are implemented with hash methodology and there is hardly any reason to make your own implementations.

Introduction

Hash tables are an implementation of an abstract data type which stores items with unique *keys* with efficient support for the operations *search*, *insert* and *delete*.

Structur	Complexity for	
	search	*insert*
linked list	$\Theta(n)$	$\Theta(n)$
sorted array	$\Theta(\log n)$	$\Theta(n)$
binary search tree	$\Theta(\log n)$	$\Theta(\log n)$
hash table	$\Theta(1)$	$\Theta(1)$

In hash tables, you can thus search, insert and delete records in a constant time *no matter how many records are stored in the table*!

I can be proven that, on average, $\Omega(\log n)$ operations are required to locate a given key among n keys if you only allow keys to be compared by size. With that limitation one can say that binary search in arrays and search in (balanced) binary search trees are optimal — there is simply no way to do better.

However, it is possible to achieve faster methods if one, instead of basing search on *size comparisons*, uses the keys to *calculate the location*. The ideal would be that each key should have have an *unique* place (index in a list). However, since the amount of *possible* keys usually is far too large this is seldom achievable. That is when *hash tables* can be used.

Hash methodology

The records are stored in an indexable structure (an *array* or a *Python list*) called the *hash table*. The records are uniquely identified by a *key* (social security number, registration number, variable name, word, ...). In practice, the records often contain information other than the keys (e.g .address, owner, variable value, ...) but here we only look at the keys since they decides where the record is to be stored.

The hash table must have a fixed size and must be selected according to the *expected number* keys to be stored. For example, a register of Sweden's population should therefore have approximately $10,000,000,000$ places.

We also need a so-called *hash function* which specifies where in the table a certain key should be placed. The function should map the set of keys on the set of indexes. If the table size is m then the function should have a value in the range $[0 : m - 1]$.

There are two important questions: How should the hash function be constructed and how

do we handle two different keys getting the same placement?

For now, we assume that the keys are positive integers. As hash function h we can then use the modulo function (the rest for integer division):

$$h(k) = k \bmod m$$

The modulo operation guarantees that the function value comes in the range $[0, m-1]$.

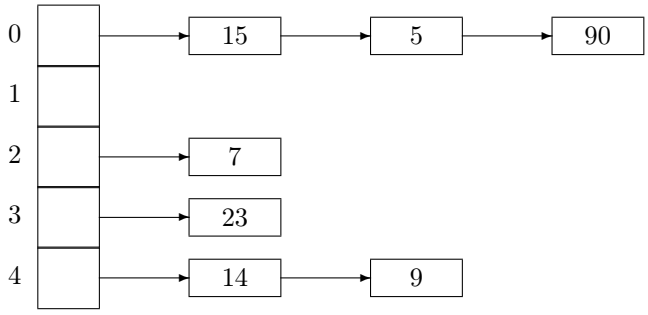
If two different keys k_1 and k_2 get the same location, this is called one *collision*. One way to handle this is to link together all elements that get the same location in a linked list. Usually the hash table T is then allowed to contain pointers to list elements. This is called *linked collision management*.

In e.g. Python you to use a regular list — it does not need to be linked.

Example

Assume that the keys 14, 15, 5, 7, 90, 23 and 9 are inserted in a hash table with 5 places ($m = 5$) and with linked collision management.

k	14	15	5	7	90	23	9
$h(k)$	4	0	0	2	0	3	4



Assuming that each key entered is equally likely to be searched, a successful search in this table requires

$$s = \frac{4 \cdot 1 + 2 \cdot 2 + 1 \cdot 3}{7} = \frac{11}{7} = 1.57 \tag{1}$$

on average.

A failed search requires on average

$$u = \frac{4 + 1 + 2 + 2 + 3}{5} = \frac{12}{5} = 2.4 \tag{2}$$

attempts if all keys are equally likely to be searched. (It takes one attempt to search an empty list.)

Analysis of linked collision management*

To analyze hash methods, we assume that a given key has the same probability of ending up in each of the m places.

If n keys are stored, the m lists are on average n/m long. To discover that a given key *does not* exist (a *failed* search) requires on the average

$$u(n,m) = 1 + n/m \tag{3}$$

attempts (searching an empty list requires, as we said, one attempt).

Searching for a key that exists (*successful* search) is a little harder to analyze because the average should be formed over the n keys – not over the number of table places.

We assume that each key is equally likely to be searched.

Observation: *It takes the same number of attempts to find an existing key as it was required to enter that key* which is the same as *the number of attempts required to search for that key before it was inserted.*

Thus, the average work of searching for an existing key is just as great as the work of building the table

$$\begin{aligned} s(n,m) &= \frac{1}{n}\left((1+\frac{0}{m})+(1+\frac{1}{m})+(1+\frac{2}{m})+\cdots+(1+\frac{n-1}{m})\right) \\ &= \frac{1}{n}\left(n+\frac{1}{m}\sum_{i=0}^{n-1} i\right) \\ &= 1+\frac{n(n-1)}{2mn} \\ &= 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{1}{2m} \approx 1+\frac{\alpha}{2} \end{aligned} \tag{4}$$

where $\alpha = n/m$.

Hash functions

The function

$$h(k) = k \bmod m$$

which we have used in the example is not at all bad provided that m is a prime number. If m is not a prime number then the function becomes more sensitive to systematics in the keys.

A more general function is

$$h(k) = ((c_1 k + c_2) \bmod p) \bmod m$$

where $p > m$ is a large prime number, $p > 2^{20}$ (e.g. $p = 1048583$, c_1 a positive integer less than p and c_2 a non-negative integer less than p. The constant c_2 is often set to 0. Different c_1 provide completely different hash functions!

If the key is a string $s_1 s_2 s_3 \cdots s_k$ we can use the characters to produce a number e.g through

$$(((s_1 \cdot 37 + s_2) \cdot 37 + s_3) + \cdots) + s_k$$

and then use any of the above hash functions.

Hash tables in programming languages

Since the hash methodology is extremely fast, it is built into most modern programming languages and you need rarely implement this yourself.

In Python, *dictionaries*) is implemented as hash tables. Python allows a dictionary to grow. If the size of the hash table changes, the hash function must change, which means that already stored elements need to be reshashed and Python has advanced methods to do this automatically and efficiently.

Java has a number of classes which implement hash tables, among others a `HashMap` and a `HashSet`. Here it is the user's responsibility to provide an estimate of how many elements there are to be stored.