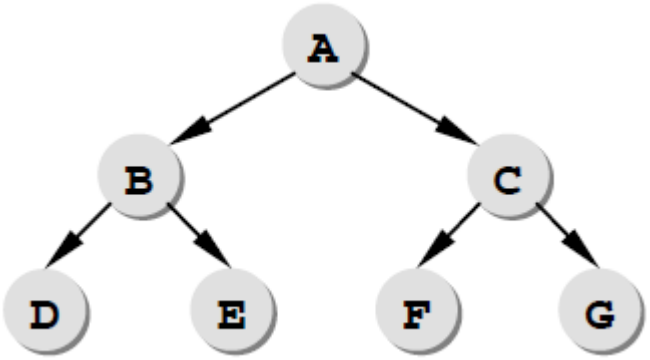
Деревья

Дерево – это структура данных, отражающая иерархию (отношения подчиненности, многоуровневые связи). Напомним некоторые основные понятия, связанные с деревьями.

Дерево состоит из узлов и связей между ними (они называются дугами). Самый первый узел, расположенный на верхнем уровне (в него не входит ни одна стрелка-дуга) – это корень дерева. Конечные узлы, из которых не выходит ни одна дуга, называются листьями. Все остальные узлы, кроме корня и листьев – это промежуточные узлы.

Из двух связанных узлов тот, который находится на более высоком уровне, называется «родителем», а другой – «сыном». Корень – это единственный узел, у которого нет «родителя»; у листьев нет «сыновей».



Используются также понятия «предок» и «потомок». «Потомок» какого-то узла – это узел, в который можно перейти по стрелкам от узла-предка. Соответственно, «предок» какого-то узла –это узел, из которого можно перейти по стрелкам в данный узел.

В дереве на рисунке справа родитель узла E – это узел B, а предки узла E – это узлы A и B, для которых узел E – потомок. Потомками узла A (корня дерева) являются все остальные узлы.

Высота дерева – это наибольшее расстояние (количество рёбер) от корня до листа. Высота дерева, приведённого на рисунке, равна 2.

Формально **дерево** можно определить следующим образом:

1) пустая структура – это дерево;

2) дерево – это корень и несколько связанных с ним отдельных (не связанных между собой) деревьев.

Здесь множество объектов (деревьев) определяется через само это множество на основе простого базового случая (пустого дерева). Таким образом дерево – это рекурсивная структура данных. Поэтому можно ожидать, что при работе с деревьями будут полезны рекурсивные алгоритмы.

Чаще всего в информатике используются двоичные (или бинарные) деревья, то есть такие, в которых каждый узел имеет не более двух сыновей. Их также можно определить рекурсивно.

Двоичное дерево:

1) пустая структура – это двоичное дерево;

2) двоичное дерево – это корень и два связанных с ним отдельных двоичных дерева («левое» и «правое» поддеревья).

Деревья широко применяются в следующих задачах:

• поиск в большом массиве данных;

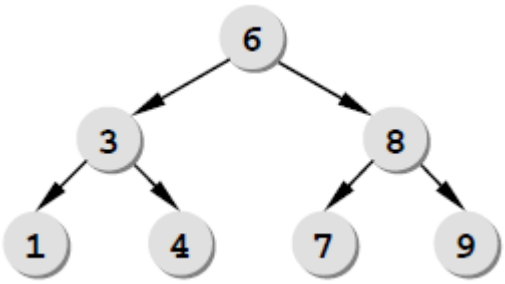
• сортировка данных;

• вычисление арифметических выражений;

• оптимальное кодирование данных (метод сжатия Хаффмана).

Деревья поиска

Известно, что для того, чтобы найти заданный элемент в неупорядоченном массиве из N элементов, может понадобиться N сравнений. Теперь предположим, что элементы массива организованы в виде специальным образом построенного дерева, например:



Значения, связанные с каждым из узлов дерева, по которым выполняется поиск, называются ключами этих узлов (кроме ключа узел может содержать множество других данных). Перечислим важные свойства показанного дерева:

• слева от каждого узла находятся узлы с меньшим ключом;

• справа от каждого узла находятся узлы, ключ которых больше или равен ключу данного узла.

Дерево, обладающее такими свойствами, называется двоичным деревом поиска.

Например, нужно найти узел, ключ которого равен 4. Начинаем поиск по дереву с корня. Ключ корня – 6 (больше заданного), поэтому дальше нужно искать только в левом поддереве, ит.д.

Скорость поиска наибольшая в том случае, если дерево сбалансировано, то есть для каждой его вершины высота левого и правого поддеревьев различается не более чем на единицу. Если при линейном поиске в массиве за одно сравнение отсекается 1 элемент, здесь – сразу примерно половина оставшихся. Количество операций сравнения в этом случае пропорционально $\log_2 N$, то есть алгоритм имеет асимптотическую сложность $O(\log N)$. Конечно, нужно учитывать, что предварительно дерево должно быть построено. Поэтому такой алгоритм выгодно применять в тех случаях, когда данные меняются редко, а поиск выполняется часто (например, в базах, данных).

Обход дерева

Обойти дерево – это значит «посетить» все узлы по одному разу. Если перечислить узлы в порядке их посещения, мы представим данные в виде списка.

Существуют несколько способов обхода двоичного дерева:

• КЛП = «корень – левый – правый» (обход в прямом порядке):

```
посетить корень
обойти левое поддерево
обойти правое поддерево
```

• ЛКП = «левый – корень – правый» (симметричный обход):

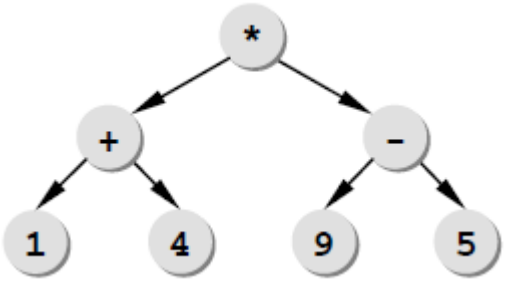
```
обойти левое поддерево
посетить корень
обойти правое поддерево
```

• ЛПК = «левый – правый – корень» (обход в обратном порядке):

```
обойти левое поддерево
обойти правое поддерево
посетить корень
```

Как видим, это рекурсивные алгоритмы. Они должны заканчиваться без повторного вызова, когда текущий корень – пустое дерево.

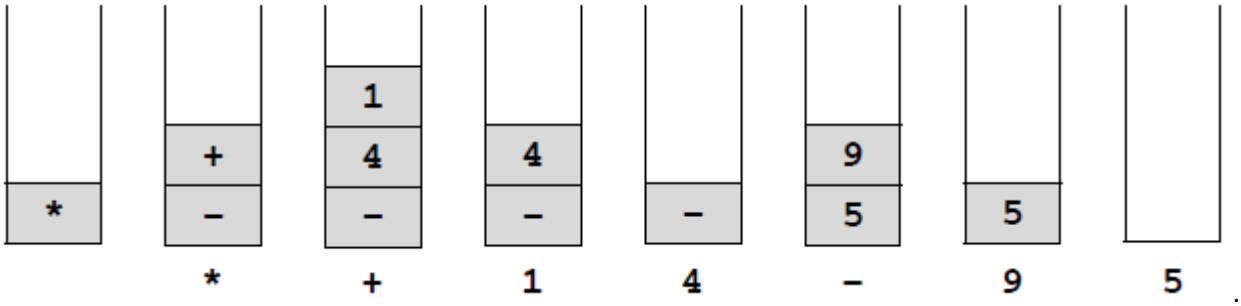
Рассмотрим дерево, которое может быть составлено для вычисления арифметического выражения **(1+4)*(9-5)**:



Выражение вычисляется по такому дереву снизу-вверх, то есть корень дерева – это последняя выполняемая операция.
Различные типы обхода дают последовательность узлов:
КЛП: *** + 1 4 – 9 5**
ЛКП: **1 + 4 * 9 - 5**
ЛПК: **1 4 + 9 5 - ***
В первом случае мы получили префиксную форму записи арифметического выражения, во втором – привычную нам инфиксную форму (только без скобок), а в третьем – постфиксную форму. Напомним, что в префиксной и в постфиксной формах скобки не нужны.
Обход КЛП называется «обходом в глубину», потому что сначала мы идём вглубь дерева полевым поддеревьям, пока не дойдём до листа. Такой обход можно выполнить с помощью стека следующим образом:

```
записать в стек корень дерева
while стек не пуст:
  выбрать узел V с вершины стека
  посетить узел V
  if у узла V есть правый сын:
    добавить в стек правого сына V
  if у узла V есть левый сын:
    добавить в стек левого сына V
```

На рисунке показано изменение состояния стека при таком обходе дерева. Под стеком записана метка узла, который посещается (например, данные из этого узла выводятся на экран).



Существует еще один способ обхода, который называют «обходом в ширину». Сначала посещают корень дерева, затем – всех его «сыновей», затем – «сыновей сыновей» («внуков») и т.д., постепенно спускаясь на один уровень вниз. Обход в ширину для приведённого выше дерева даст такую последовательность посещения узлов:
обход в ширину: *** + - 1 4 9 5**
Для того, чтобы выполнить такой обход, применяют очередь. В очередь записывают узлы, которые необходимо посетить. На псевдокоде обход в ширину можно записать так:

```
записать в очередь корень дерева
while очередь не пуста:
  выбрать первый узел V из очереди
  посетить узел V
  if у узла V есть левый сын:
    добавить в очередь левого сына V
  if у узла V есть правый сын:
    добавить в очередь правого сына V
```

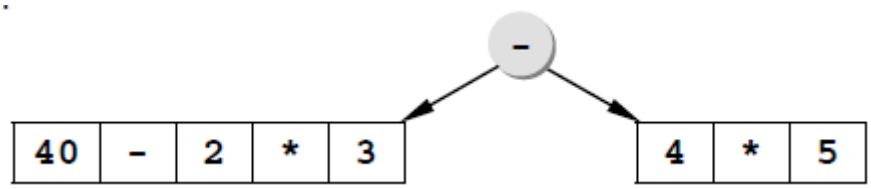
Вычисление арифметических выражений

Один из способов вычисления арифметических выражений основан на использовании дерева. Сначала выражение, записанное в линейном виде (в одну строку), нужно «разобрать» и построить соответствующее ему дерево. Затем в результате прохода по этому дереву от листьев к корню вычисляется результат.

Для простоты будем рассматривать только арифметические выражения, содержащие числа и знаки четырёх арифметических действий: **+−*/**. Построим дерево для выражения

40 − 2 * 3 − 4 * 5

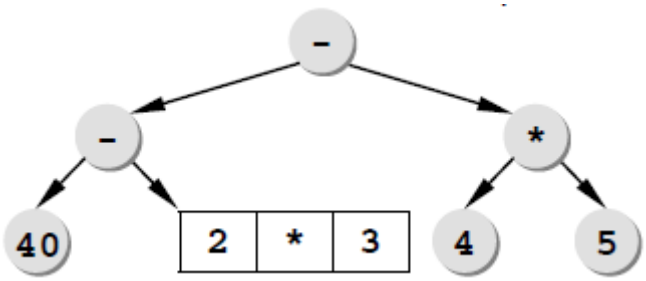
Так как корень дерева – это последняя операция, нужно сначала найти эту последнюю операцию, просматривая выражение слева направо. Здесь последнее действие – это второе вычитание, оно оказывается в корне дерева.



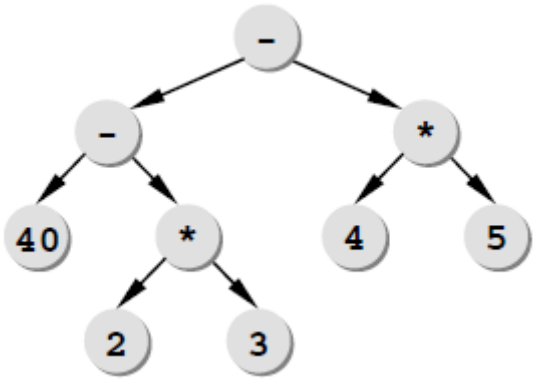
Как выполнить этот поиск в программе? Известно, что операции выполняются в порядке приоритета (старшинства): с начала операции с более высоким приоритетом (слева направо), потом – с более низким (также слева направо). Отсюда следует важный вывод:

В корень дерева нужно поместить последнюю из операций с наименьшим приоритетом.

Теперь нужно построить таким же способом левое и правое поддеревья:



Левое поддерево требует еще одного шага:



Эта процедура рекурсивная, её можно записать в виде псевдокода:

```
найти последнюю выполняемую операцию
if операций нет:
  создать узел-лист
  return
поместить найденную операцию в корень дерева
построить левое поддерево
построить правое поддерево
```

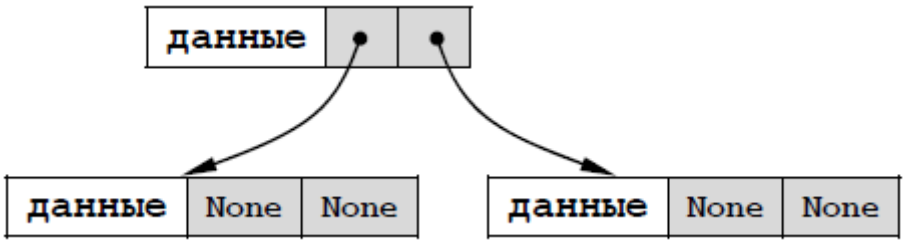
Рекурсия заканчивается, когда в оставшейся части строки нет ни одной операции, значит, там находится число (это лист дерева).
Теперь вычислим выражение по дереву. Если в корне находится знак операции, её нужно применить к результатам вычисления поддеревьев:

```
n1 = значение левого поддерева
n2 = значение правого поддерева
результат = операция ( n1, n2 )
```

Снова получился рекурсивный алгоритм.
Возможен особый случай (на нём заканчивается рекурсия), когда корень дерева содержит число (то есть это лист). Это число и будет результатом вычисления выражения.

Использование связанных структур

Поскольку двоичное дерево – это нелинейная структура данных, использовать список для размещения элементов не очень удобно (хотя возможно). Вместо этого будем использовать связанные узлы. Каждый такой узел – это структура, содержащая три области: область данных, ссылка на левое поддерево (указатель) и ссылка на правое поддерево (второй указатель). У листьев нет «сыновей», в этом случае в указатели будем записывать специальное значение **None** («пусто», «ничто»). Дерево, состоящее из трёх таких узлов, показано на рисунке:



В данном случае область данных узла будет содержать одно поле – символьную строку, в которую записывается знак операции или число в символьном виде.

Введём новый тип (класс) данных – структуру **TNode** – узел дерева

```
class TNode:
  pass
```

Определим функцию, которая создаёт новый узел, записывает в его поле **data** переданные данные и присваивает нулевые значения указателям на поддеревья (полям **left** и **right**):

```
def newNode( d ):
  node = TNode()
  node.data = d
  node.left = None
  node.right = None
  return node
```

Пусть s – символьная строка, в которой записано арифметическое выражение (будем предполагать, что это правильное выражение без скобок). Тогда вычисление выражения сводится к двум вызовам функций:

```
T = makeTree ( s )
print ( "Результат: ", calcTree(T) )
```

Здесь функция **makeTree** строит в памяти дерево по строке **s**, а функция **calcTree** – вычисляет значение выражения по готовому дереву.

При построении дерева нужно выделить в памяти новый узел и искать последнюю выполняемую операцию – это будет делать функция **lastOp**. Она вернет «–1», если ни одной операции не обнаружено, в этом случае создается лист – узел без потомков. Если операция найдена, её обозначение записывается в поле **data**, а в указатели – адреса поддеревьев, которые строятся рекурсивно для левой и правой частей выражения:

```
def makeTree ( s ):
  k = lastOp(s)
  if k < 0:                              # создать лист
    Tree = newNode ( s )
  else:                                  # создать узел-операцию
    Tree = newNode ( s[k] )
    Tree.left = makeTree ( s[:k] )
    Tree.right = makeTree ( s[k+1:] )
  return Tree
```

Функция **calcTree** (вычисление арифметического выражения по дереву) тоже будет рекурсивной:

```
def calcTree ( Tree ):
  if Tree.left == None:
    return int(Tree.data)
  else:
    n1 = calcTree ( Tree.left )
    n2 = calcTree ( Tree.right )
    if Tree.data == "+":   res = n1 + n2
    elif Tree.data == "-": res = n1 - n2
    elif Tree.data == "*": res = n1 * n2
    else: res = n1 // n2
    return res
```

Если ссылка на узел, переданная функции, указывает на лист (нет левого поддерева), то значение выражения – это результат преобразования числа из символьной формы в числовую (с помощью функции **int**). В противном случае вычисляются значения для левого и правого поддеревьев, и к ним применяется операция, указанная в корне дерева.

Осталось написать функцию **lastOp**. Нужно найти в символьной строке последнюю операцию с минимальным приоритетом. Для этого составим функцию, возвращающую приоритет операции (переданного ей символа):

```
def priority ( op ):
  if op in "+-": return 1
  if op in "*/": return 2
  return 100
```

Сложение и вычитание имеют приоритет 1, умножение и деление – более высокий приоритет 2, а все остальные символы (не операции) – приоритет 100 (условное значение).

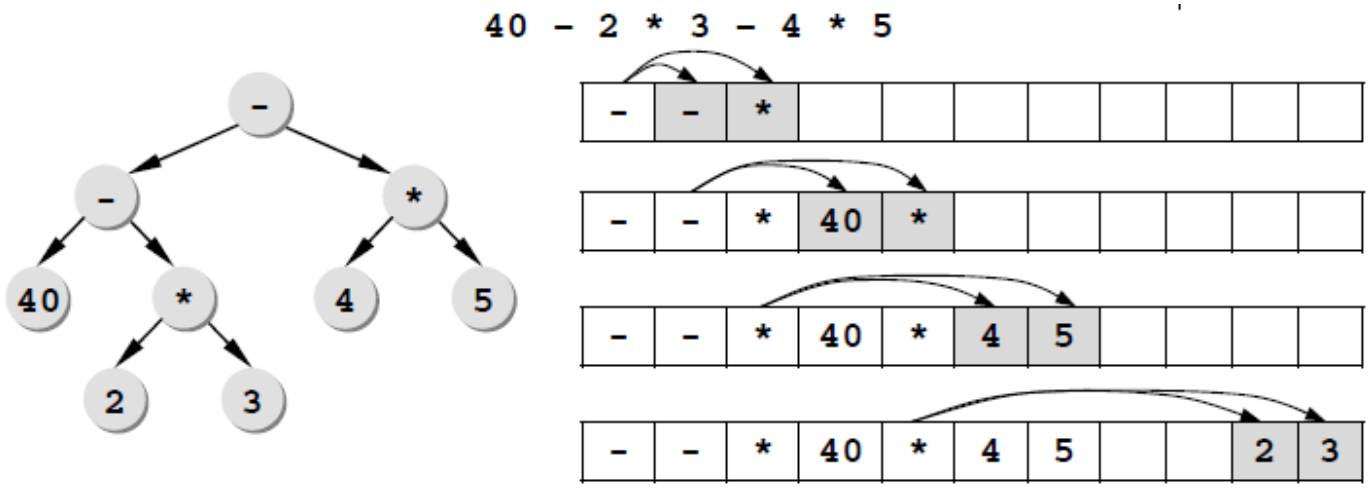
Функция **lastOp** может выглядеть так:

```
def lastOp ( s ):
  minPrt = 50    # любое между 2 и 100
  k = -1
  for i in range(len(s)):
    if priority(s[i]) <= minPrt:
      minPrt = priority(s[i])
      k = i
  return k
```

Обратите внимание, что в условном операторе указано нестрогое неравенство, чтобы найти именно последнюю операцию с наименьшим приоритетом. Начальное значение переменной **minPrt** можно выбрать любое между наибольшим приоритетом операций (2) и условным кодом не-операции (100). Тогда если найдена любая операция, условный оператор срабатывает, а если в строке нет операций, условие всегда ложно и в переменной **lastOp** остается начальное значение «–1».

Хранение двоичного дерева в массиве

Двоичные деревья можно хранить в массиве (списке). Вопрос о том, как сохранить структуру (взаимосвязь узлов) решается достаточно просто. Если нумерация элементов массива A начинается с 0, то «сыновья» элемента **A[i]** – это **A[2*i+1]** и **A[2*i+2]**. На рисунке показан порядок расположения элементов в массиве для дерева, соответствующего выражению



Алгоритм вычисления выражения остается прежним, изменяется только метод хранения данных. Обратите внимание, что некоторые элементы остались пустые, это значит, что их «родитель» –лист дерева. В программе на Python в такие (неиспользуемые) элементы массива можно записывать «пустое» значение **None**. Конечно, лучше всего так хранить сбалансированные деревья, иначе будет много пустых элементов, которые зря расходуют память.

Модульность

При разработке больших программ нужно разделить работу между программистами так, чтобы каждый делал свой независимый блок (*модуль*). Все подпрограммы, входящие в модуль, должны быть связаны друг с другом, но слабо связаны с другими процедурами и функциями.

В нашей программе в отдельный модуль можно вынести все операции с деревьями, то есть определение класса **TNode** и все подпрограммы. Назовём этот модуль **bintree** (от англ. *binary tree* – двоичное дерево) и сохраним в файле с именем **bintree.py**. Теперь в основной программе можно использовать этот модуль стандартным образом, загружая его с помощью команды **import**:

```
import bintree
…
T = bintree.makeTree ( s )
print ( "Результат: ", bintree.calcTree ( T ) )
```

или загружая только нужные нам функции:

```
from bintree import makeTree, calcTree
…
T = makeTree ( s )
print ( "Результат: ", calcTree ( T ) )
```

Обратите внимание, что нам не нужно знать, как именно устроены функции **`makeTree`** и **`calcTree`**: какие типы данных они используют, по каким алгоритмам работают и какие дополнительные функции вызывают. Для использования функции модуля достаточно знать *интерфейс* –соглашение о передаче параметров (какие параметры принимают подпрограммы и какие результаты они возвращают).

Модуль в чём-то подобен айсбергу: видна только надводная часть (интерфейс), а значительно более весомая подводная часть скрыта. За счёт этого все, кто используют модуль, могут не думать о том, как именно он выполняет свою работу. Это один из приёмов, которые позволяют справляться со сложностью больших программ. Разделение программы на модули облегчает понимание и совершенствование программы, потому что каждый модуль можно разрабатывать, изучать и оптимизировать независимо от других.