Абстрактные типы данных в Java

В Java существует множество контейнерных типов. Контейнерные классы в Java реализуют интерфейс Collection и далее делятся на Queue, Set и List;

Списки:
LinkedList - реализует интерфейс List и представляет собой обычный двунаправленный список.

```
static LinkedList<Integer> list = new LinkedList<>();
list.add(1);
list.add(2);
list.addFirst(0);
list.removeLast();
list.forEach(System.out::println);
```

ArrayList - списочный массив. Обобщённый класс, реализующий интерфейс абстрактного класса List.
Как известно, в Java массивы имеют фиксированную длину, и после того как массив создан, он не может расти или уменьшаться. ArrayList может менять свой размер во время исполнения программы, при этом не обязательно указывать размерность при создании объекта.

```
static ArrayList<Integer> array = new ArrayList<>();
array.add(1);
array.add(3);
array.add(5);
array.add(4);
array.add(2);
array.remove(1);
System.out.println(String.format("Third element is %d", array.get(0)));
array.sort(Comparator.comparingInt(o -> o));
array.forEach(System.out::println);
```

Деревья:
TreeMap - обобщённое красно-чёрное дерево. Реализует интерфейс Map. Объекты можно сортировать, передав свой объект класса Comparator.

TreeSet - обобщённое дерево, для проверки наличия элементов во множестве.

```
static TreeMap<Integer, String> tree = new TreeMap<>();
tree.put(1, "one");
tree.put(2, "two");
if (tree.containsKey(1)) {
   System.out.println(tree.get(1));
```

Отображения:

HashMap - основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени. Разрешение коллизий осуществляется с помощью метода цепочек.

```
static HashMap<Integer, String> hashMap = new HashMap<>();
hashMap.put(1, "one");
hashMap.put(2, "two");
hashMap.put(3, "3");
hashMap.replace(3, "three");
System.out.println(hashMap.get(3));
```

	Временная сложность							
	Среднее				*Худшее*			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hashtable	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
HashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeMap	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))

Абстрактные типы данных в C++

Упрощённая реализация дерева

```
#include<iostream>

using namespace std;

class BST
{
   struct node
   {
      int data;
      node* left;
      node* right;
   };

   node* root;

   node* makeEmpty(node* t)
   {
      if(t == NULL)
         return NULL;
      {
         makeEmpty(t->left);
         makeEmpty(t->right);
         delete t;
      }
      return NULL;
   }

   node* insert(int x, node* t)
   {
      if(t == NULL)
      {
         t = new node;
         t->data = x;
         t->left = t->right = NULL;
      }
      else if(x < t->data)
         t->left = insert(x, t->left);
      else if(x > t->data)
```

```
        t->right = insert(x, t->right);
    return t;
}

node* findMin(node* t)
{
    if(t == NULL)
        return NULL;
    else if(t->left == NULL)
        return t;
    else
        return findMin(t->left);
}

node* findMax(node* t)
{
    if(t == NULL)
        return NULL;
    else if(t->right == NULL)
        return t;
    else
        return findMax(t->right);
}

node* remove(int x, node* t)
{
    node* temp;
    if(t == NULL)
        return NULL;
    else if(x < t->data)
        t->left = remove(x, t->left);
    else if(x > t->data)
        t->right = remove(x, t->right);
    else if(t->left && t->right)
    {
        temp = findMin(t->right);
        t->data = temp->data;
        t->right = remove(t->data, t->right);
    }
    else
    {
        temp = t;
        if(t->left == NULL)
```

```
            t = t->right;
        else if(t->right == NULL)
            t = t->left;
        delete temp;
    }

    return t;
}

void inorder(node* t)
{
    if(t == NULL)
        return;
    inorder(t->left);
    cout << t->data << " ";
    inorder(t->right);
}

node* find(node* t, int x)
{
    if(t == NULL)
        return NULL;
    else if(x < t->data)
        return find(t->left, x);
    else if(x > t->data)
        return find(t->right, x);
    else
        return t;
}

public:
BST()
{
    root = NULL;
}

~BST()
{
    root = makeEmpty(root);
}

void insert(int x)
{
```

```
        root = insert(x, root);
    }

    void remove(int x)
    {
        root = remove(x, root);
    }

    void display()
    {
        inorder(root);
        cout << endl;
    }

    void search(int x)
    {
        root = find(root, x);
    }
};
```

Пример использования

```
 int main()
{
    BST t;
    t.insert(20);
    t.insert(25);
    t.insert(15);
    t.insert(10);
    t.insert(30);
    t.display();
    t.remove(20);
    t.display();
    t.remove(25);
    t.display();
    t.remove(30);
    t.display();
}
```

Упрощённая реализация список:

```
#include <iostream>
#include <cstdlib>


class Node
{
public:
    Node* next;
    int data;
};

using namespace std;

class LinkedList
{
public:
    int length;
    Node* head;

    LinkedList();
    ~LinkedList();
    void add(int data);
    void print();
};

LinkedList::LinkedList(){
    this->length = 0;
    this->head = NULL;
}

LinkedList::~LinkedList(){
    std::cout << "LIST DELETED";
}

void LinkedList::add(int data){
    Node* node = new Node();
    node->data = data;
    node->next = this->head;
    this->head = node;
    this->length++;
}
```

```
void LinkedList::print(){
    Node* head = this->head;
    int i = 1;
    while(head){
        std::cout << i << ": " << head->data << std::endl;
        head = head->next;
        i++;
    }
}

int main(int argc, char const *argv[])
{
    LinkedList* list = new LinkedList();
    for (int i = 0; i < 100; ++i)
    {
        list->add(rand() % 100);
    }
    list->print();
    std::cout << "List Length: " << list->length << std::endl;
    delete list;
    return 0;
}
```

Отображения

Реализуются в стандартной библиотеке STL.

Example:

```
void mapExample() {
  // Красно-чёрное дерево
  std::map<int, string> myMap;
  myMap.insert(std::make_pair(1, "one"));
  myMap.insert(std::make_pair(2, "two"));
  myMap.insert(std::make_pair(3, "three"));
  // ...
  cout << myMap[1] << endl;  // one
  cout << myMap[-1] << endl; //
}
```

Абстрактные типы данных в Python

Списки в Python - упорядоченные изменяемые коллекции объектов произвольных типов (почти как массив, но типы могут отличаться).

Чтобы использовать списки, их нужно создать. Создать список можно несколькими способами. Например, можно обработать любой итерируемый объект (например, строку) встроенной функцией **list**:

```
>>>
```

```
>>> list('список')
['с', 'п', 'и', 'с', 'о', 'к']
```

```
>>> s = []  # Пустой список
>>> l = ['s', 'p', ['isok'], 2]
>>> s
[]
>>>
['s', 'p', ['isok'], 2]
```

Как видно из примера, список может содержать любое количество любых объектов (в том числе и вложенные списки), или не содержать ничего.

И еще один способ создать список - это **генераторы списков**. Генератор списков - способ построить новый список, применяя выражение к каждому элементу последовательности. Генераторы списков очень похожи на цикл for.

```
>>> c = [c * 3 for c in 'list']
>>> c
['lll', 'iii', 'sss', 'ttt']
```

Возможна и более сложная конструкция генератора списков:

```
>>> c = [c * 3 for c in 'list' if c != 'i']
>>> c
['lll', 'sss', 'ttt']
>>> c = [c + d for c in 'list' if c != 'i' for d in 'spam' if d != 'a']
>>> c
['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']
```

Но в сложных случаях лучше пользоваться обычным циклом for для генерации списков.

Таблица "методы списков"

Метод	Что делает
list.append(x)	Добавляет элемент в конец списка
list.extend(L)	Расширяет список list, добавляя в конец все элементы списка L
list.insert(i, x)	Вставляет на i-ый элемент значение x
list.remove(x)	Удаляет первый элемент в списке, имеющий значение x. ValueError, если такого элемента не существует
list.pop([i])	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
list.index(x, [start [, end]])	Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end)
list.count(x)	Возвращает количество элементов со значением x
list.sort([key=функция])	Сортирует список на основе функции

list.reverse()	Разворачивает список
list.copy()	Поверхностная копия списка
list.clear()	Очищает список

Реализация дерева поиска

Код для конструктора класса `BinarySearchTree` и нескольких других различных функций показан в *листинге 1*.

Листинг 1

```
class BinarySearchTree:

  def __init__(self):
    self.root = None
    self.size = 0

  def length(self):
    return self.size

  def __len__(self):
    return self.size

  def __iter__(self):
    return self.root.__iter__()
```

Класс TreeNode предоставляет множество вспомогательных функций, которые значительно облегчают работу BinarySearchTree. Конструктор TreeNode вместе с этими

функциями показан в *листинге 2*. Из него видно, что большинство дополнительных функций помогают классифицировать узел в соответствии с его положением как потомка (левого или правого) и типом его детей.

Так же класс TreeNode явно отслеживает родителя, как атрибут каждого узла. Вы поймёте важность этого, когда мы будем обсуждать реализацию оператора del.

Другой любопытный аспект TreeNode в *листинге 2* заключается в использовании опциональных параметров Python. Они позволяют нам упростить его создание в соответствии с различными обстоятельствами. Иногда мы захотим сконструировать новый объект TreeNode, имеющий и parent, и child. С уже существующими потомком и предком их можно просто передать, как параметры. В следующий раз мы создадим TreeNode с парой ключ-значение, не передавая при этом ни parent, ни child. В этом случае для параметров будут использоваться значения по умолчанию.

Листинг 2

```
class TreeNode:
   def __init__(self,key,val,left=None,right=None,
         parent=None):
  self.key = key
  self.payload = val
  self.leftChild = left
  self.rightChild = right
  self.parent = parent

def hasLeftChild(self):
    return self.leftChild

def hasRightChild(self):
    return self.rightChild

def isLeftChild(self):
    return self.parent and self.parent.leftChild == self

def isRightChild(self):
    return self.parent and self.parent.rightChild == self

def isRoot(self):
    return not self.parent

def isLeaf(self):
    return not (self.rightChild or self.leftChild)
```

```
def hasAnyChildren(self):
   return self.rightChild or self.leftChild

def hasBothChildren(self):
   return self.rightChild and self.leftChild

def replaceNodeData(self,key,value,lc,rc):
   self.key = key
   self.payload = value
   self.leftChild = lc
   self.rightChild = rc
   if self.hasLeftChild():
      self.leftChild.parent = self
   if self.hasRightChild():
      self.rightChild.parent = self
```

Теперь, когда у нас есть обёртка BinarySearchTree и класс TreeNode, пришло время написать метод put, который позволит строить двоичные деревья поиска. Он будет принадлежать классу BinarySearchTree. Метод будет выполнять проверку на наличие корня дерева, а в случае отсутствия последнего создавать объект TreeNode и устанавливать его, как корневой узел. В противном случае put вызовет приватную рекурсивную вспомогательную функцию _put для поиска места в дереве по следующему алгоритму:

1) Начиная от корня, проходим по двоичному дереву, сравнивая новый ключ с ключом текущего узла. Если первый меньше второго, то идём в левое поддерево. Наоборот - в правое.
2) Когда не осталось левых или правых потомков для поиска - мы нашли позицию для установки нового узла.
3) Чтобы добавить узел в дерево, создаём новый объект TreeNode и помещаем его на найденное за предыдущие шаги место.

Листинг 3 показывает код Python для вставки нового узла в дерево. Функция _put написана рекурсивно и следует описанным выше пунктам. Отметьте, что когда в дерево вставляется новый потомок, currentNode передаётся как родитель нового поддерева. Одной из серьёзных проблем нашей реализации является то, что дубликаты ключей не обрабатываются правильным образом. У нас дубль создаст новый узел с точно таким же значением ключа и поместит его в правое поддерево узла с оригинальным ключом. В результате новый узел никогда не будет обнаружен в процессе поиска. Для управления вставкой дубликатов ключей есть способ лучше: сделать так, чтобы значение, ассоциированное с новым ключом, заменяло старое. Мы оставляем вам исправление этого недочёта в качестве упражнения.

Листинг 3

```
def put(self,key,val):
  if self.root:
     self._put(key,val,self.root)
  else:
     self.root = TreeNode(key,val)
  self.size = self.size + 1

def _put(self,key,val,currentNode):
  if key < currentNode.key:
     if currentNode.hasLeftChild():
      self._put(key,val,currentNode.leftChild)
     else:
      currentNode.leftChild = TreeNode(key,val,parent=currentNode)
  else:
     if currentNode.hasRightChild():
      self._put(key,val,currentNode.rightChild)
     else:
      currentNode.rightChild = TreeNode(key,val,parent=currentNode)
```

Словари
Словарь — изменяемый объект-отображение.

dict([obj,][**kwargs])

obj : Первым необязательным позиционным аргументом может являться отображение или итерирующийся объект (при этом каждый его элемент должен быть тоже итерирующемся и содержать ровно два объекта).
****kwargs** : Поддерживаются также необязательные именованные аргументы. При использовании вкупе с позиционными аргументами и совпадении ключей значениями из именованных пользуются приоритетом
Словари реализованы при помощи динамических хеш-таблиц. По сравнению с двоичными деревьями, это, в большинстве случаев, даёт выигрыш при получении значений (наиболее часто используемая операция); кроме этого упрощается реализация.

Для каждого ключа при помощи функции hash() вычисляется хеш-код. Код этот широко варьируется в зависимости от ключа и данных процесса (например, хеш для «Python» может быть -539294296, в то время как для «python» — отличается от первого одним битом — он будет 1142331976). Хеш используется для определения места (во внутреннем массиве), где хранится значение. В случае если все используемые вами ключи будут иметь различные хеши, для получения значения по ключу будет затрачено постоянное время — O(1).

Абстрактные типы данных в Lisp

Атомы ≡ символы ∪ числа.
Список - структура, состоящая из элементов, которыми могут быть атомы или другие списки.
(t (t nil t) t),
(1 (2 (3 4) 5))

— это списки. Список может не содержать элементов вовсе, такой список называется пустым и обозначается () или nil.
Список - это фундамент лиспа, ибо в зависимости от интерпретации список может представлять как данные, так и лисповый код. (symbol-value t) - тоже список.
Помимо вышеперечисленного, в лиспе есть еще тонна различных типов данных. Но у тебя, уважаемый читатель, уже есть достаточно знаний, чтобы перейти к изучению функций, что и советую сделать, прежде чем переходить к знакомству с другими типами данных. А у меня пока будет время, чтобы дописать этот раздел :-)
Точечная пара (она же "точечный список") - структура, состоящая из двух элементов, разделенных точкой. Выглядит это примерно так:
1. (1 . 2),
2. (('a 'b) . 3),
3. ('a . (4 . 9)),
4. ('a . nil) и т.д.

Первый элемент точечной пары называется головой, второй элемент хвостом. Конструировать точечные пары можно при помощи функции cons, у которой всего два аргумента - голова и хвост будущей точечной пары. Например, вызов (cons 1 2) создаст точечную пару (1 . 2). Функция cons не создает список, путем присоединения головы списка к ее хвосту, как пишут во многих учебниках и интернетах.