УДК 519.163
ББК 32.972.134

ОСОБЕННОСТИ РЕАЛИЗАЦИИ ПРОГРАММНОГО СКРИПТА ДЛЯ ОПИСАНИЯ ПОИСКОВОГО АЛГОРИТМА НА БАЗЕ B-ДЕРЕВЬЕВ*

Дубровин Т.Г.[1], Добринец И.М.[2]

[1]МАОУ г. Иркутска «Лицей ИГУ»
[2]ФГБОУ ВО «Иркутский государственный университет»
[1]tihon_baikal_99@mail.ru, [2]dobr.ph@gmail.com

Исследование возможностей поисковых алгоритмов в сложных древовидных структурах информации в рамках тестовой Linux-платформы проекта является одним из приоритетных исследовательских направлений по проектированию среды BaikalIntelli [1] для решения задачи быстрого поиска информации в больших гетерогенных базах данных (знаний). При этом информация может содержаться не только в привычных и хорошо знакомых разработчикам SQL-базах данных, но и в огромном количестве файлов специальных структурированных форматов (xml, json).

B-дерево — структура данных, дерево поиска. **B** в слове означает **«Balanced»**, то есть «сбалансированное». Использование B-деревьев впервые было предложено Р. Бэйером (англ. R. Bayer) и Э. МакКрейтом (англ. E. McCreight) в 1970 году. Сбалансированность означает, что длина любых двух путей от корня до листьев различается не более, чем на единицу. Ветвистость дерева — это свойство каждого узла дерева ссылаться на большое число узлов-потомков.

B-деревья позволяют быстро находить целые блоки данных по заданному ключу. Кроме того, они решают сразу большой объем проблем: поиск, добавление, удаление. Благодаря этим достоинствам они получили широкую популярность в базах данных, в частности в ISAM и MyISAM от MySQL.

Для того чтобы лучше понять алгоритмы работы с B-деревьями, были апробированы многочисленные симуляторы, собрана библиотека примеров на разных языках программирования [2, 3]. В ходе сравнительного анализа было выявлено, что существующие поисковые симуляторы дают представление только о принципах, а не о тонкостях алгоритмизации. Примеры программ часто неполны, устарели или содержат алгоритмические ошибки.

При построении B-дерева применяется фактор t, который называется минимальной степенью. Каждый узел, кроме корневого узла, должен иметь, как минимум $t-1$, и не более $2t-1$ ключей. Обозначается $n[x]$ – количество ключей в узле x.

Ключи в узле хранятся в неубывающем порядке. Если x не является листом, то он имеет $n[x]+1$ детей. Если занумеровать ключи в узле x, как $k[i]$, а детей $c[i]$, то для любого ключа в поддереве с корнем $c[i]$ (пусть $k1$), выполняется следующее неравенство – $k[i-1] \le k1 \le k[i]$ (для $c[0]: k[i-1] = -\infty$,а для $c[n[x]]: k[i] = +\infty$). Таким образом, ключи узла задают диапазон для ключей их детей.

Все листья B-дерева должны быть расположены на одной высоте, которая и является высотой дерева. Высота B-дерева с $n \ge 1$ узлами и минимальной степенью $t \ge 2$ не превышает $\log_t(n+1)$. Это важное утверждение. $h \le \log_t((n+1)/2)$ — логарифм по основанию t.

Для реализации сбалансированного поиска дерева было описано два класса:

* Исследование выполнено при финансовой поддержке РФФИ и Правительства Иркутской области в рамках научного проекта № 20-41-385002

- класс Node, отвечающий за операции над узлами B-дерева;
- класс BPlusTree, отвечающий за операции над B-деревом.

Класс Node имеет в себе важные функции, без которых само дерево BPlusTree работать не будет, и также не будут работать вспомогательные функции, которые облегчают работу нашего BPlusTree.

Класс Node содержит следующие атрибуты и функции:

- атрибут порядок (order);
- атрибут ключи (keys);
- атрибут значения (values);
- атрибут лист (leaf);
- функция добавления ключей и значений (add);
- функция разделения узла на две части (split);
- функция проверки на полноту (is_full);
- функция вывода на экран ключей узла (show).

Класс BPlusTree содержит следующие атрибуты и функции:

- атрибут корень (Node);
- функция поиска места для вставки узла по ключу (find);
- функция извлечения свободного элемента (merge);
- функция вставки пары ключей-значений (insert);
- функция поиска значения по ключу (retrieve);
- функция вывода на экран ключей на каждом уровне (show).

Особенности реализации класса Node

Функция add дополняет узел ключами и значениями. Для этого мы спускаемся по дереву, в поисках места для вставки нового ключа. Если вставляемый ключ больше текущего, идём вправо; если меньше, то идём влево; если равен, то останавливаемся и выходим: ключ уже есть в дереве и вставлять не нужно. Затем проверяем узел на наличие ключей. Если ключей нет вообще, записываем добавляемый ключ и значение в узел и завершаем работу функции. Иначе перебираем все ключи по порядку. Если вставляемый ключ окажется равен одному из имеющихся в узле, то добавляем вставляемое значение к значениям этого индекса узла.

Если будет найден ключ, который больше вставляемого ключа (напоминаем, что ключи в узле всегда упорядочены по возрастанию), то вставляем ключ и значение в узел перед найденным индексом большего ключа, сохраняя упорядочивание.

Если список ключей кончился, то добавляем ключ и значение в конец списка узла.

```
def add(self, key, value):
   if not self.keys:
      self.keys.append(key)
      self.values.append([value])
      return None
   for i, item in enumerate(self.keys):
      if key == item:
         self.values[i].append(value)
         break
      elif key < item:
         self.keys = self.keys[:i] + [key] + self.keys[i:]
         self.values = self.values[:i] + [[value]] + self.values[i:]
         break
    #i отстаёт на одно значение, key[i] - i+1-й элемент списка->элемент №4
      elif i + 1 == len(self.keys):
         self.keys.append(key)
         self.values.append([value])
         break
```

Функция __init__ создаёт (инициализирует) новый пустой узел заданного порядка, который является листом и не содержит ни ключей, ни значений.

```
def __init__(self, order):
    self.order = order
    self.keys = []
    self.values = []
    self.leaf = True
```

Фунция split делит заданный узел на две части. Инициализируем два новых узла (левый и правый). Ищем середину исходного узла и записываем ключи и значения меньше индекса середины в левый, а больше – в правый список. Затем оставляем в исходном списке лишь один ключ, младший в правом узле (забираем его из этого узла). В качестве значений исходного узла принимаем левый и правый узлы полностью.

Из правой части забираем самый первый ключ в исходный ключ.

Объявляем, что исходный узел не лист.

```
def split(self):
     left = Node(self.order)
     right = Node(self.order)
     mid = self.order // 2 - 1
     left.keys = self.keys[:mid]
     left.values = self.values[:mid]
     right.keys = self.keys[(mid):]
     right.values = self.values[(mid):]
     self.keys = [right.keys[0]]
     right.keys = right.keys[1:]
     right.values = right.values[1:]
     self.values = [left, right]
     self.leaf = False
```

Функция is_full дает ответ, заполнен ли наш узел полностью. Для этого функция возвращает результат сравнения порядка узла и количества ключей в нём (истину в случае совпадения или ложь в случае несовпадения значений).

```
def is_full(self):
     return len(self.keys) == self.order
```

Функция show выводит все ключи заданного списка. Если список не является листом, то рекурсивно запускаем эту функцию для всех значений заданного списка.

```
def show(self, counter=0):
    print(counter, str(self.keys))
    if not self.leaf:
       for item in self.values:
       item.show(counter + 1)
```

Особенности реализации класса BPlusTree

Фунция init задаёт атрибут root в качестве нового пустого узла.

```
def __init__(self, order=8):
    self.root = Node(order)
```

Функция find возвращает индекс для заданного узла и ключа, где ключ должен быть вставлен и список значений по этому индексу. Начинает перебор всех ключей в узле. Если найден больше, чем заданный, то возвращаем значение и индекс этого ключа. Иначе возвращаем последнее значение и последний индекс узла.

```
def _find(self, node, key):
    for i, item in enumerate(node.keys):
        if key < item:
            return node.values[i], i
    return node.values[i + 1], i + 1
```

Функция merge извлекает для родительского и дочернего узла свободный элемент из дочернего и вставляет в ключи родителя. Вставляет значения от ребенка в значения родителя.

Удаляем одно из значений родительского списка по заданному индексу с конца. Сохраняет отдельно первый ключ родительского списка. Ищем место для сохранённого ключа среди ключей родительского списка. Если место найдено, то вставляем на него ключ и все значения дочернего списка. Иначе дописываем ключ и значения в самый конец родительского списка.

```
def _merge(self, parent, child, index):
    parent.values.pop(index)
    pivot = child.keys[0]
    for i, item in enumerate(parent.keys):
        if pivot < item:
            parent.keys = parent.keys[:i] + [pivot] + parent.keys[i:]
            parent.values = parent.values[:i] + child.values +
            parent.values[i:]
            break
        elif i + 1 == len(parent.keys):
            parent.keys += [pivot]
            parent.values += child.values
            break
```

Функция insert вставляет пару ключ-значений после перехода к конечному узлу. Если листовой узел заполнен, разделяет листвой узел на две части.

```
def insert(self, key, value):
    parent = None
    child = self.root
    while not child.leaf:
        parent = child
        child, index = self._find(child, key)
    child.add(key, value)
    if child.is_full():
        child.split()
    if parent and not parent.is_full():
        self._merge(parent, child, index)
```

Функция retrieve возвращает значение для данного ключа и «None», если ключ не существует. Создаём дочерниного узла равного корневому узлу нашего дерева. Затем ищем по нашему ключу, в каком из листов нашего дерева может содержаться ключ. После нахождения нужного листа, начинаем перебирать все ключи, которые в нём содержаться. При нахождении ключа возвращаем значение по его индексу. Иначе возвращаем ответ «None».

```
def retrieve(self, key):
    child = self.root
    while not child.leaf:
        child, index = self._find(child, key)
    for i, item in enumerate(child.keys):
        if key == item:
            return child.values[i]
    return None
```

Функция show перенаправляет запрос печати всех ключей к корню дерева (узлу). Затем одноимённая функция для класса Node рекурсивно печатает все дочерние узлы корня нашего дерева.

```
def show(self):
    self.root.show()
```

Вывод. Описанный алгоритм B-дерева может являться полноценной основой для проектирования программ, использующих структуры данных в виде B-деревьев, а также может помочь при разработке альтернативных скриптов на базе B-деревьев. Листинг кода рабочего скрипта опубликован на сервисе GitHub [4].

Исследование выполнено при финансовой поддержке РФФИ и Правительства Иркутской области в рамках научного проекта № 20-41-385002.

1. Arguchintsev A.V. Metatechnology o fdesigning a hierarchical modiable-intersecting database for remote control of bioecological parameters. / A.V. Arguchintsev, V.S. Kedrin, L.S. Kravtsova, I.M. Dobrinets, Glyzina M.A. (Voylo) // Limnology and Freshwater Biology. 2020. Vol. 4 P. 641-642. DOI:10.31951/2658-3518-2020-A-1-106.
2. «BTrees: Scalable persistent object containers», URL: https://pypi.org/project/BTrees/ (дата обращения: 23.04.2020)
3. «B+ tree - analogous to binary search tree but with more branching at each node, as shallow trees makes better use of caches», URL: https://github.com/savarin/sql (дата обращения: 23.03.2020)
4. Дубровин Т.Г., «Tihon99 / sql», URL: https://github.com/Tihon99/sql/blob/master/index.py/ (дата обращения: 14.05.2020)