Фонд оценочных средств  
по дисциплине  
«Языки программирования и   
методы программирования»

Приложение 3

Лабораторная работа №3

(информатика, 3 семестр)

Варианты заданий

# Постановка задачи

Написать программу на C++ для сравнения различных алгоритмов поиска. Сравнение алгоритмов должно производиться на одной из приведенных задач, связанных с обработкой информации. Написать краткое техническое задание (ТЗ). Выполнить реализацию. Написать для нее тесты.

**Минимальные требования к программе**. В программе должно быть реализовано несколько различных методов поиска. Для каждого метода существуют различные модификации, дополняющие или улучшающие возможности или характеристики метода. Они перечислены в таблице ниже. Каждая модификация снабжена рейтинговой оценкой, пропорциональной сложности реализации. Студент самостоятельно выбирает конфигурацию для реализации исходя из условия: суммарный рейтинг должен быть не меньше 55. Суммарный рейтинг выбранной студентом задания непосредственно влияет на оценку, см. раздел 5.

Основные реализованные алгоритмы необходимо покрыть тестами. Программа должна позволять выбрать любой из реализованных алгоритмов поиска и запустить его на (достаточно произвольных) исходных данных. При этом должна быть возможность как автоматической, так и ручной проверки корректности работы алгоритмов (в т.ч. должна быть возможность просмотра как исходных данных, так и результата). Программа должна обладать пользовательским интерфейсом (консольным или графическим). Пользовательский интерфейс, в особенности, графический, тестировать не требуется. Программа должна предоставлять функцию измерения времени выполнения алгоритма. Должна быть функция сравнения алгоритмов – по времени выполнения на одних и тех же входных данных[[1]](#footnote-1). Программа должна предоставлять функционал по построению графиков зависимостей, либо по выгрузке необходимых данных в открытых форматах (например, csv).

# Задача на обработку информации

|  |  |  |
| --- | --- | --- |
| **№** | **Модификация** | **Рейтинг** |
|  | Построение гистограммы – распределения объектов по группам | 10 |
|  | Реализация обработки в стиле map-reduce | 5 |
|  | Параметризация критерия (показателя), по которому строится распределение | 5 |
|  | Обработка разреженных векторов и матриц | 8 |
|  | Обработка матриц, а не векторов | 2 |
|  | Реализация обработки в стиле map-reduce | 5 |
|  | Возможность загрузки сценария обработки из файла | 4 |
|  | Приоритезация задач на основе вычисления частотных характеристик инцидентов | 5 |
|  | Построение алфавитного указателя | 12 |
|  | Для задания размера страницы использовать количество символов, а не слов | 4 |
|  | Индексирование данных | 7 |
|  | Поддержка поиска по диапазону значений | 5 |
|  | Поддержка индексирования составных атрибутов | 7 |

## Построение гистограммы

Гистограмма – это разбиение исходного множества на заданное число подмножеств (либо на подмножества фиксированного размера, но их число заранее может быть не известно). Разбиение производится по определенному признаку (например, возраст). Множество возможных значений этого признака разбивается на элементарные подмножества (например, диапазоны в 5-10 лет). Эти элементарные множества могут быть как одинакового размера (равномерное разбиение), так и различного (неравномерное разбиение). Интерес представляет количество элементов, попадающих в каждое элементарное подмножество.

Вход:

1. Sequence – входная последовательность элементов (например, объектов Person),
2. Параметры гистограммы (диапазон значений, разбиение)

Выход:

* IDictionary: ключом является диапазон (элементарное множество), значением – количество элементов, попадающих в этот диапазон

## Обработка разреженных векторов и матриц

Разреженный вектор – это вектор (как правильно, очень большой размерности, скажем, и более), в котором подавляющее большинство элементов (~80-90% и более) равны 0. В этом случае использование стандартных массивов для хранения значений является неоправданной тратой памяти: достаточно хранить лишь ненулевые значения (и их индексы, разумеется). Данные в таком случае следует хранить в формате ключ-значение, причем ключом является индекс в векторе, а значением – собственно элемент. Потенциально работоспособными могут быть любые варианты структур данных: сортированные последовательности, деревья, хеш-таблицы.

Разреженная матрица – это матрица (как правильно, очень большой размерности, скажем, и более), в которой подавляющее большинство элементов (~80-90% и более) равны 0. В этом случае справедливы те же замечания, что и в случае с разреженными векторами, только вместе с каждым элементов нужно хранить пару индексов, а не один. В случае использования хеш-таблиц полезно написать специализированную хеш-функцию для пары числовых (целочисленных) значений.

Вход:

1. Вектор или матрица (массив или подобная структура)

Выход:

* Разреженный вектор или матрица

В задаче требуется реализовать тип данных – разреженный вектор (или матрица), используя ISortedSequence, бинарное или B/B+-дерево, IDictionary. При выборе нескольких разных структур данных следует получить их сравнительную характеристику производительности.

## Приоритезация задач на основе вычисления частотных характеристик инцидентов

Вход:

1. Sequence – входная последовательность инцидентов (структур типа Incident, в которой имеется список ключевых слов)
2. Sequence (еще лучше – ISet) – множество заданий (объектов типа Task, в которых есть список ключевых слов)

Выход:

* IPriorityQueue: перечень заданий (Task), приоритизированный по частоте ключевых слов, встречающихся в инцидентах (Incident)

*Указание*. Перечень ключевых слов может задаваться изначально или же вычисляться путем прохода по обеим последовательностям – инцидентов и заданий. Теоретически, множества ключевых слов, встречающихся в каждом из этих наборов, могут не совпадать, а в крайнем случае – даже не пересекаться (в таком случае результирующая очередь будет состоять из списка заданий с 0-м приоритетом).

## Построение алфавитного указателя

Вход:

1. Строка, состоящая из слов, разделенных пробелами
2. Размер страницы (в словах или символах)

Выход:

* ISortedSequence<Pair<string,int>>, где первый элемент каждой пары – это слово, встречающееся в исходной строке, а второй элемент – номер страницы, который вычисляется по алгоритму, описанному ниже.

Алгоритм разбиения текста на страницы.

1. Исходную строку разбить на отдельные слова, используя пробелы в качестве разделителей. В результате получится список строк, в котором каждый элемент – это слово.
2. Список слов «режется» на фрагменты так, чтобы каждый фрагмент умещался на станице. Если размер страницы указан в словах, то длина каждого фрагмента должна быть не больше этой величины. Если размер страницы задан в количестве символов, то следует вычислить суммарную длину слов выделенного фрагмента (положим, что всего фрагмент содержит слов) и прибавить величину – количество пробелов, которое необходимо, что записать эти слова раздельно.
3. При делении списка слов на страницы следует применять следующее правило: первая страница может быть заполнена не более, чем на половину, а каждая десятая – не более, чем на 3/4 от заданного размера страницы.

# Структуры данных и алгоритмы поиска и их модификации

В каждой задаче требуется реализовать одну или несколько из следующих из следующих структур: множество (ISet), сортированная последовательность (ISortedSequence), очередь с приоритетами (IPriorityQueue) или ассоциативная память (IDictionary). В зависимости от того, какие из этих структур студент реализует и какими для этого пользуется «базовыми» структурами, к рейтингу формируемого задания прибавляется величина из таблицы ниже.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sequence | Бинарное дерево | B/B+ дерево | Хеш-таблица |
| ISet | 3 | 5 | 7 | 5 |
| ISortedSequence | 3 | 5 | 7 | – |
| IPriorityQueue | – | 5 | 7 | 5 |
| IDictionary | – | 5 | 7 | 5 |

«Базовые» структуры допускают определенные вариации и усовершенствования. Указанные рейтинги пропорциональны сложности реализации. В приведенном списке элементы верхнего уровня соответствуют методу или алгоритму, а элементы нижнего уровня – это модификации. При формировании индивидуального задания рейтинг метода суммируется с реализуемыми модификациями. Модификации одного и того же метода, как правило, не являются взаимоисключающими, их рейтинги также могут суммироваться.

|  |  |  |
| --- | --- | --- |
| **№** | **Модификация** | **Рейтинг** |
|  | Бинарный поиск по отсортированной последовательности (базовая реализация) | 10 |
|  | Возможность выбора пропорции деления: пополам, либо в соответствии с «золотым сечением» | 5 |
|  | Пропорции деления задаются параметром – парой натуральных чисел, либо номером последовательности Фибоначчи | 5 |
|  | Бинарное дерево | 15 |
|  | Сбалансированное | 7 |
|  | Прошитое | 7 |
|  | B/В+-дерево (без удаления) | 20 |
|  | Операция удаления | 15 |
|  | Хеш-таблица, реализация интерфейса IDictionary<TKey,TElement> (без разрешения коллизий) | 12 |
|  | Разрешение коллизий с помощью списков | 7 |
|  | Разрешение коллизий с помощью смещения адресов | 7 |
|  | Реализовать поддержку итераторов |  |
|  | В Sequence | 3 |
|  | В бинарном дереве | 4 |
|  | В B/B+-дереве | 4 |
|  | В хеш-таблице | 3 |
|  | Хранение данных в структурированном бинарном или текстовом файле | 10 |

**Пояснения**.

1. Деревья поиска следует реализовывать как шаблонные классы. Это позволит как хранить в них сами данные, так и лишь индексы, отмечающие положение данных в последовательности. У шаблона один параметра-типа: тип хранимого значения.
   1. В случае, если хранятся простые объекты, например, вещественные числа, то тип этих объектов выбирается в качестве значения этого параметра: double (например, Tree<double>).
   2. Если дерево строится «над» числовой последовательностью и требуется хранить положение элемента (имеющего тип T, например, double, в случае вещественных чисел) в исходной последовательности, то тип хранимого значения будет парой Pair<T,int>. Ключом в такой паре ключ-значение будет первый элемент, т.е. интересующий нас объект, а значением – его индекс (целое число int) в последовательности.
   3. Если последовательность состоит из сложных информационных объектов (например, класс Student), а поиск ведется по атрибуту типа T, то дерево будет инстанцировано с параметрами шаблона: Tree<Pair<T,int>> (в предположении, что хранимым является именно индекс объекта в последовательности), либо Tree<T,Student\*> (если в дереве хранятся именно указатели).
2. Для хранения элементов в узлах B/B+ деревьев следует использовать реализованный ранее Sequence.
3. Алгоритм построения дерева из последовательности может быть реализован в одном из следующих вариантов:
   1. Как статический метод класса, описывающего дерево (не желательно, т.к. создает жесткую связку между классом дерева и классом последовательности).
   2. В составе отдельного класса (например, BinaryTreeBuilder). Это позволит в дальнейшем легко расширять функциональность этого класса, добавляя методы для построения деревьев из других структур данных (например, «голых» массивов или std:vector<>).
   3. В виде отдельной функции.
4. В случае с хранением данных в бинарном файле, дерево хранит лишь индексы – положение данных в файле.
5. Краткая спецификация SortedSequence:

|  |  |  |  |
| --- | --- | --- | --- |
| **Класс** | SortedSequence<TElement> | | |
| Является модификацией интерфейса Sequence, описывающей отсортированные последовательности. Основное отличие состоит в том, что методы Append, Prepend и InsertAt заменены одним методов Add. | | | |
| **Название** | | **Сигнатура** | **Описание** |
| Атрибуты | | | |
| Length | | int GetLength() | Длина последовательности (количество элементов) |
| IsEmpty | | int GetIsEmpty() | Признак того, является ли последовательность пустой |
| Методы | | | |
| Get | | TElement Get(int index) | Получение элемента по индексу. |
| GetFirst | | TElement GetFirst() | Получить первый элемент последовательности |
| GetLast | | TElement GetLast | Получить последний элемент последовательности |
| IndexOf | | int IndexOf(TElement element) | Получить индекс элемента последовательности. Если указанного элемента в последовательности не содержится, возвращается значение -1. |
| GetSubsequence | | SortedSequence<TElement> GetSubsequence(int startIndex, int endIndex) | Получить подпоследовательность: начиная с элемента с индексом startIndex и заканчивая элементом с индексом endIndex |
| Add | | Void Add(TElement element) | Добавить элемент в последовательность. Элемент автоматически вставляется так, что итоговая последовательность остается отсортированной. |

1. Хеш-таблица должна хранить данные в структуре данных, обеспечивающих обращение по индексу (index-based addressing) – например, Sequence. Хеш-функция должна вычислять положение в этом массиве на основании ключа. Это достигается в 2 этапа: сначала для каждого используемого типа ключа выбирается некоторая функция , которая для каждого экземпляра типа возвращает некоторое положительное целое (). Эта функция должна удовлетворять следующему обязательному условию:

Если значения этой функции для некоторых двух элементов различаются, то и сами элементы различны.

Полученное таким образом целое число является промежуточным для вычисления итогового значения хеш-функции. Хеш-функция также должна удовлетворять приведенному выше обязательному условию. Значение хеш-функции трактуется как индекс элемента в внутреннем массиве хеш-таблицы.

В минимальном варианте необходимо реализовать хотя бы одну из следующих возможностей:

* 1. использовать какую-либо списковую структуру (например, реализованный ранее тип данных Sequence) для обработки коллизий;
  2. автоматическую перестройку хеш-функции и внутренней последовательности при возникновении коллизий.

Для получения максимального балла необходимо реализовать обе указанные возможности, а также автоматическую перестройку при удалении элементов.

1. Перестройку хеш-таблицы необходимо проводить не при каждом удалении, а лишь тогда, когда число элементов стало ниже некоторого порогового значения. Более точно, следует зафиксировать два (необязательно целых) положительных числа: и (причем ). При этом хещ-таблица характеризуется двумя неотрицательными целыми числами: количеством элементов () и вместимостью (). Всякий раз, когда оказывается, что , следует сжать в раз. Если же оказывается, что , то таблицу следует расширить в раз.
2. Краткая спецификация IDictionary:

|  |  |  |  |
| --- | --- | --- | --- |
| **Класс** | IDictionary<TKey,TElement> | | |
|  | | | |
| **Название** | | **Сигнатура** | **Описание** |
| Атрибуты | | | |
| Count | | int GetCount() | Количество элементов, которое фактически хранится |
| Capacity | | int GetCapacity() | Вместимость – максимальное количество элементов, которое можно поместить в таблицу без необходимости ее перестройки. |
| Методы | | | |
| Get | | TElement Get(TKey key) | Получение элемента по ключу. Выбрасывает исключение, если элемента не существует. |
| ContainsKey | | bool ContainsKey(TKey key) | Проверка, что в таблице уже есть элемент с заданным ключом. |
| Add | | void Add(TKey key, TElement element) | Добавить элемент с заданным ключом. |
| Remove | | void Remove(TKey key) | Удаляет элемент с заданным ключом. Выбрасывает исключение, если заданный ключ отсутствует в таблице. |

**Пример**. Будем искать студентов по ФИО (FullName).

int GetStringHashCode(string s) { /\*…\*/ }

Dictionary<string,Student> students(&GetStringHashCode, 25);

students.Add(student1.GetFullName(), student1);

students.Add(student2.GetFullName(), student2);

students.Add(student2.GetFullName(), student3);

// students.ContainsKey(student1.GetFullName()) => TRUE

// students.ConstainsKey(student4.GetFullName()) => FALSE

students.Get(student1.GetFullName()) // => student1

students.ConstainsKey(student4.GetFullName()) // => Exception

students.GetCount() // => 3

students.GetCapacity() // => 25, если автоматической перестройки

// нет и 25/q, если есть

1. В зависимости от того, какие структуры выбраны для реализации, появляются дополнительные опции (О-1) – реализация итератора. Краткая спецификация IIterator:

|  |  |  |  |
| --- | --- | --- | --- |
| **Класс** | IIterator<TElement> | | |
|  | | | |
| **Название** | | **Сигнатура** | **Описание** |
| Атрибуты | | | |
| CurrentItem | | int GetCurrentItem() | Получение текущего элемента по ключу. Выбрасывает исключение, если итератор находится за границами коллекции. |
| HasNext | | bool HasNext() | Проверка того, итератор не достиг конца коллекции. |
| Методы | | | |
|  | |  |  |
| Next | | bool Next() | Перейти на следующий элемент, если возможно. Если переход удалось выполнить, возвращает true, иначе – false. |
| TryGetCurrentItem | | bool Add(TElement& element) | Попытаться получить текущий элемент. Если попытка удалась, то element содержит полученное значение, а метод возвращает true; в противном случае метод возвращает false, а значение element не определено. |

**Пример** **1** использования.

Sequence\* seq = new ArraySequence();

// добавление элементов в seq

IIterator\* it = seq->GetIterator();

for (it->Next(); it->Next(); it->HasNext())

cout << it->GetCurrentItem();

**Пример** **2**.

BTree btree = new BTree();

// добавление элементов в btree

for (it->Next(); it->Next(); it->HasNext())

cout << it->GetCurrentItem();

Для реализации итератора для бинарных и B/B+-деревьев достаточно реализовать обход дерева.

# Методика формирования индивидуального задания

Формирование индивидуального задания происходит в 2 шага:

1. Выбор решаемой задачи и ее модификаций из раздела 2. Вычисляется рейтинг задачи в баллах.
2. Выбор реализуемых алгоритмов и структур данных и их вариантов из раздела 3. Вычисляется суммарный рейтинг выбранных структур данных и их модификаций.

Сумма рейтингов, полученных на шагах 1 и 2, должна быть не меньше 55.

Каждая задача имеет базовую оценку в баллах (обычно, 1 балл). Каждая предлагаемая модификация также имеет рейтинг в баллах, пропорциональную ее сложности (). Вычисляется суммарная оценка сложности выбранной задачи: .

Далее, из таблицы 3 выбираются подходящие структуры данных и алгоритмы. Каждый из них имеет свой рейтинг . Далее, выбираются модификации реализуемых структур; каждая модификация имеет рейтинг . Вычисляется суммарный рейтинг выбранных алгоритмов и структур: .

Задача и алгоритмы для ее решения должны быть выбраны так, чтобы получившийся суммарный рейтинг был ***не* меньше** 55.

Следует учитывать, что в некоторых задачах структуры данных, которые нужно использовать, частично зафиксированы условием.

# Критерии оценки

|  |  |  |  |
| --- | --- | --- | --- |
|  | Качество программного кода: | * стиль (в т.ч.: имена, отступы и проч.) (0-2) * структурированность (напр. декомпозиция сложных функций на более простые) (0-2) * качество основных и второстепенных алгоритмов (напр. обработка граничных случаев и некорректных исходных данных и т.п.) (0-3) | 0-7  баллов |
|  | Качество пользовательского интерфейса: | * предоставляемые им возможности (0-2) * наличие ручного/автоматического ввода исходных данных (0-2) * настройка параметров для автоматического режима   отображение исходных данных и промежуточных и конечных результатов и др. (0-2) | 0-6  баллов |
|  | Качество тестов | * степень покрытия * читаемость * качество проверки (граничные и некорректные значения, и др.) | 0-3  баллов |
|  | Объем выполненного задания | Оценивается объем выполненного задания, исходя из суммарного рейтинга:   |  |  |  |  | | --- | --- | --- | --- | | 00 | – | 50% | 0 баллов | | 50 | – | 60% | 1 балл | | 60 | – | 70% | 2 балла | | 70 | – | 90% | 3-4 балла | | 90 | – | 100% | 5 баллов | | 0-5  баллов |
|  | Владение теорией | знание алгоритмов, области их применимости, умение сравнивать с аналогами, оценить сложность, корректность реализации | 0-3  баллов |
|  | Оригинальность реализации | оцениваются отличительные особенности конкретной реализации – например, общность структур данных, наличие продвинутых графических средств, средств ввода-вывода, интеграции с внешними системами и др. | 0-9  баллов |
|  | Итого | | 0-33  баллов |

Для получения зачета за выполнения лабораторной работы необходимо соблюдение всех перечисленных условий:

* оценка за п. 1 должна быть не менее 3 баллов
* оценка за п. 4 должна быть не менее 3 баллов
* оценка за п. 5 должна быть больше 0
* суммарная оценка за работу без учета п. 6 должна быть не менее 15 баллов

1. Следует рассматривать три основных случая: последовательность уже отсортирована в нужном направлении; последовательность отсортирована в обратном направлении; последовательность не отсортирована. В случае деревьев, имеется ввиду сравнение времени построения дерева, а не время поиска в уже построенном дереве. [↑](#footnote-ref-1)