МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Факультет Информационных технологий и программирования

Направление: Прикладная математика и информатика
Специализация: Математическое и программное обеспечение вычислительных машин
Академическая степень: магистр математики

Кафедра Компьютерных технологий Группа 6538

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Применение деревьев для представления строковой информации

Автор магистерской диссертации Абдрашитов Д. С.

Научный руководитель Корнеев Г. А.

Санкт-Петербург, 2008 г.

Содержание

Введение ... 4
Глава 1. Методы представления строковой информации ... 6
1.1. Изменения в структурах данных ... 6
1.2. Существующие методы представления строк ... 8
1.2.1. Операции над строками ... 8
1.2.2. Изменяемые строки ... 10
1.2.3. Изменяемые строки на циклических очередях ... 11
1.2.4. Неизменяемые строки ... 12
1.2.5. Строки на основе изменяемых списков ... 13
1.2.6. Строки на основе неизменяемых списков ... 15
1.2.7. Анализ существующих методов представления строк ... 17
1.3. Примеры хронологических структур данных ... 18
1.3.1. Хронологический стек ... 19
1.3.2. Хронологическое декартово дерево ... 21
1.3.3. Хронологическое 2-3-дерево ... 24
1.3.4. Невозможность объединяемо хронологической структуры за O(1) 25
1.4. Постановка задачи ... 25
Глава 2. Строки на основе хронологических деревьев ... 28
2.1. Общая идея реализации ... 28
2.1.1. Описание операций ... 31
2.1.2. Оценка времени операций ... 33
2.1.3. Оценка требуемого количества памяти ... 35
2.2. Конкретные реализации ... 37
2.2.1. Реализация на основе декартовых деревьев ... 37
2.2.2. Реализация на основе 2-3-деревьев ... 43
Глава 3. Улучшенный метод представления строк ... 46
3.1. Сведение операций к базовым операциям ... 46
3.1.1. Набор базовых операций ... 46

3.1.2. Оценка времени через базовые операции ... 46
3.2. Амортизированная реализация ... 49
3.2.1. Описание ... 49
3.2.2. Оценка времени базовых операций ... 52
3.2.3. Оценка количества памяти ... 54
3.3. Реализация реального времени ... 55
3.3.1. Описание ... 55
3.3.2. Оценка времени базовых операций ... 59
3.3.3. Оценка количества памяти ... 60
3.4. Сравнение реализаций представления строковых данных ... 61
3.4.1. Окончательные оценки времени всех операций ... 61
3.4.2. Сравнительная таблица ... 62
Глава 4. Практическое внедрение ... 64
4.1. Замена строк на Java ... 64
4.1.1. Идея замены ... 64
4.1.2. Реализация строк ... 65
4.1.3. Реализация строковых буферов ... 65
4.2. Оценка результатов замены ... 65
4.2.1. Приложение «ant» ... 66
4.2.2. Приложение «Poseidon» ... 66
4.2.3. Сравнение результатов ... 67
4.3. Рекомендации по применению. ... 68
Заключение ... 69
Список литературы ... 71

Введение

Большое значение в программировании имеет абстракция «строка». В большинстве типизированных языков программирования для строк выделен специальный тип данных. Однако распространенные методы представления строк в памяти компьютера довольно примитивны. Они обеспечивают быстрое выполнение только некоторых строковых операций, остальные же операции требуют количество ресурсов пропорциональное длине обрабатываемой строки.

Цель данной работы – исследовать возможность эффективного представления строковых данных. Рассматриваемым критерием эффективности являются асимптотические оценки временной сложности стандартных операций со строками. Также следует учитывать затраты используемой для хранения строк памяти.

В работе рассмотрены существующие методы представления строковой информации, а также некоторые структуры данных, которые могут быть использованы для хранения строк. В результате анализа предметной области была поставлена и обоснована задача разработки метода представления строковой информации, обеспечивающего для каждой из стандартных строковых операций время выполнения, асимптотически не превышающего время выполнения этой операции для любого другого метода более чем в логарифм раз.

Основой для решения поставленной задачи выбрано представление строк в виде деревьев. Для обеспечения желаемых логарифмических оценок необходима сбалансированность деревьев. Применение хронологических деревьев может обеспечить строкам свойство неизменяемости. Одним из возможных решений является адаптация алгоритмов сбалансированных деревьев поиска для балансировки строк на деревьях. Такой подход уже применялся в работе Аткинсона, Боема и Пласса [1]. Подход обеспечивает амортизированные логарифмические оценки всех операций.

После получения логарифмического по всем операциям метода следующей решенной задачей было усовершенствование его таким образом, чтобы часть операций выполнялась еще быстрее с точки зрения асимптотических оценок. Усовершенствование разработанного метода привело к получению структуры данных, которая может использоваться в качестве хронологического дека с операцией конкатенации. Полученная структура данных по своим характеристикам сравнима с близким результатом Каплана и Тарьяна [2].

Завершает работу экспериментальная проверка эффективности разработанного метода. Реализация метода выполнена на языке Java. Время выполнения операций разработанного метода сравнивается со стандартной реализацией строкового типа данных языка Java. Для того чтобы оценить практическую ценность разработанного метода, сравнение производится в условиях реальных приложений.

Глава 1. Методы представления строковой информации

Первая глава является обзором используемых в работе методов, алгоритмов и понятий. Сначала рассмотрено понятие хронологических структур данных. Далее приведено определение строк и перечислен ряд существующих методов представления строковых данных. Затем в качестве примера хронологических структур данных приведены структуры, используемые в работе для построения более сложных составных структур. В завершение главы сформулирована задача, решению которой посвящена работа.

1.1. Изменения в структурах данных

При совершении операций над обычными структурами данных происходит их изменение. Доступным остается только новое состояние структуры, содержащее внесенные изменения. Состояние же, в котором структура находилась до внесения изменений, безвозвратно теряется. Такие структуры данных называются изменяемыми (эфемерными [3]). Однако существует ряд задач, в которых требуется доступ ко всей истории состояний структуры. Наиболее известным примером такой задачи является задача о нахождении расположения точки на плоскости. В задаче дано разбиение плоскости на области отрезками, требуется по точке определить область, которой эта точка принадлежит. Одномерный вариант этой задачи решается с помощью двоичного дерева поиска. Идея, предложенная Слитором и Тарьяном [4], сводит решение двумерной задачи к решению одномерной, при этом в качестве второго измерения выступает номер версии дерева поиска, используемого для решения одномерной задачи.

Структура данных позволяющая обеспечить доступ ко всем состояниям, в которых она когда-либо находилась, называется хронологической. Дрискол, Сарнак, Слитор и Тарьян в работе [3] систематизировано рассмотрели понятие хронологичности и предложили методы создания

хронологических аналогов изменяемых структур данных, основанных на ссылках. В их работе приводится следующая классификация хронологичности.

В зависимости от возможности модификации предыдущих версий разделяют понятия абсолютно и частично хронологических структур. Частично хронологической называется структура данных, в которой все версии структуры доступны для просмотра, а самая последняя версия доступна для модификации. Абсолютно хронологической называется структура данных, в которой все версии доступны как для просмотра, так и для модификации.

Иногда операции совершаются не над одной структурой, а над несколькими. В качестве примера такой операции можно привести объединение двух списков. Если структура данных поддерживает операции, комбинирующие несколько ее версий, и является абсолютно хронологической, то такая структура называется объединяемо хронологической. Хронологические структуры данных используются в различных областях. В качестве примера из вычислительной геометрии уже была приведена задача о поиске расположения точки на плоскости [4]. Также хронологичность применяется в области редактирования текста [5] и при реализации языков высокого уровня [6].

При программировании практическое значение хронологичности состоит в решении проблемы совмещения имен – ситуации в программировании, когда две разных переменных в программе указывают на одну область памяти. Проблема состоит в том, что внесение изменений в одной переменной отражается сразу на всех совмещенных переменных, но это может быть не предусмотрено программистом и вызывать непредсказуемые и трудно находимые ошибки. Для хронологических структур данных такой проблемы не возникает, поскольку в результате выполнения операции создается новая структура, а старая полностью сохраняется.

В области параллельных вычислений хронологичность является единственной альтернативой синхронизации [7] для обеспечения безопасного доступа к данным. Попытка параллельного доступа к памяти зачастую требует синхронизации на аппаратном уровне, особенно это касается доступа на запись. Однако в случае хронологических структур параллельной записи не происходит, и это может обеспечить выигрыш в производительности. Даже в случае однопоточного программирования из соображений хронологичности может быть извлечена дополнительная информация, обеспечивающая оптимизацию программы на этапе компиляции.

1.2. Существующие методы представления строк

Основным объектом, изучаемым в данной работе, является строка. Дадим определение понятия строки и описание операций совершаемых со строками. Далее перечислим ряд распространенных методов представления строковых данных.

1.2.1. Операции над строками

Конечные последовательности однотипных элементов являются распространенным объектом в различных областях математики и информатики. И. В. Романовский [8] приводит в качестве примера таких объектов последовательность коэффициентов многочлена в математическом анализе, последовательность точек обхода многоугольника в геометрии, набор ненулевых элементов разряженной матрицы в линейной алгебре, абстракция списка в программировании, текст в информатике.

Для описания таких объектов может быть использован термин «строка». Согласно определению, приведенному в работе [9], строка – это конечная последовательность элементов, называемых символами, некоторого конечного множества, называемого алфавитом.

Рассмотрим некоторые операции, совершаемые со строками.

Нахождение длины строки. Длина строки – это количество символов в последовательности. В зависимости от алгоритма представления строки длина может либо храниться явно, либо вычисляться.

Сравнение строк. Сравнение – операция, используемая для задания отношения порядка на строках (обычно это лексикографический порядок), либо для задания классов эквивалентности строк (сравнение на равенство).

Получение подстроки. Подстрока строки S – это строка, состоящая из произвольной последовательности идущих подряд символов строки S. В современных языках программирования встречается как операция получения подстроки заданной длины, начинающейся с заданного места, так и выделенные операции получения подстроки от начала и с конца строки.

Конкатенация строк. Конкатенация – это операция соединения двух строк в одну. Ее результатом является строка, в которой последовательно выписаны сначала все символы первой строки, а затем все символы второй.

Поиск. Поиск вхождения некоторого образца в качестве подстроки является распространенной задачей в различных областях (например, в текстовых редакторах [9]). Задача состоит в нахождении области в строке, которая соответствует определенному критерию. В качестве критерия может выступать эквивалентность заданному образцу, а в более сложном варианте критерий задается регулярным выражением. Иногда поиск совмещают с заменой найденного участка на заданную строку, в этом случае операцию называют подстановка.

Получение символа. В некоторых случаях требуется получить символ строки по его номеру в последовательности, назовем эту операцию индексацией. В программировании часто индексацию используют для последовательного получения всех символов строки, например, при выводе строки в файл или на экран. Однако в некоторых программных реализациях строк стоимости произвольного и последовательного получения символов могут отличаться, поэтому будем рассматривать последовательное

получение очередного символа отдельно и назовем эту операцию итерированием.

1.2.2. Изменяемые строки

Поддержка изменяемых строк на уровне языковых конструкций или стандартных библиотек существует в различных языках программирования, например, в Паскале, Си, Руби. В статье [10] описаны особенности реализации строкового типа данных языка Паскаль для среды разработки Delphi. Рассмотрим изменяемые строки в этой реализации.

Сама последовательность символов хранится в памяти непрерывным блоком и заканчивается нулевым символом. Нулевой символ в конце строки – это прием, использующийся для обозначения длины строки в null-terminated строках, другом распространенном методе хранения строк. В рассматриваемом методе, в отличие от null-terminated строк, нулевой символ смысловой нагрузки не несет и введен только из соображений совместимости с другими методами. Дополнительная информация, хранящаяся вместе с последовательностью, состоит из длины строки и счетчика ссылок.

Длина храниться для обозначения того, где в выделенной под строку памяти заканчивается последовательность символов. Поскольку длина хранится в явном виде, нет необходимости ее вычислять. Операция получения длины строки занимает константное время.

При конкатенации в переменную результата выписываются последовательности символов конкатенируемых строк, а сумма их длин записывается в поле для длины результата. Если выделенной под результат памяти не хватает, выделяется новый блок памяти большего размера и результат копируется туда. Всего конкатенация требует времени пропорционально количеству символов в результирующей строке.

Аналогичную оценку времени имеет операция получения подстроки. Для получения подстроки в переменную результата копируется требуемая часть исходной строки и выставляется соответствующая длина.

Операции вставки и удаления требуют перемещение данных в памяти для сохранения целостности последовательности символов так, чтобы не возникало разрывов и наложений. Стоимость перемещения пропорциональна объему перемещаемых данных.

Расположение последовательности символов подряд позволяет получить доступ к определенному символу напрямую через его адрес в памяти. Таким образом, операции индексации, итерирования и замены символа на символ выполняются за константное время.

1.2.3. Изменяемые строки на циклических очередях

Если обратить внимание на набор совершаемых со строками операций, то можно обнаружить, что существуют классические структуры данных, которые могут быть использованы для реализации строк. Рассмотрим структуру, которая упомянута в труде Д. Кнута [11] под названием циклическая очередь.

Структура состоит из массива и двух индексов. Первый индекс указывает позицию, где в массиве начинается последовательность данных. Второй – позицию, где последовательность заканчивается. Таким образом, между указателями располагается непрерывная последовательность данных. При этом массив рассматривается как кольцо, в котором за последним элементом следует первый, а первому элементу предшествует последний.

Циклические очереди по своей сути очень похожи на изменяемые строки, рассмотренные в прошлом разделе. Главным отличием является возможность перемещения начала строки, которая позволяет эффективно добавлять и удалять символы с обоих концов. Поскольку структура симметрична, рассмотрим добавление и удаление символов на примере начала строки. При добавлении символ записывается в ячейку массива, на которую указывает индекс начала строки, а сам индекс уменьшается на единицу. Если индекс после уменьшения стал меньше чем позиция начала массива, он перемещается на конец массива ввиду цикличности. Если он

достиг индекса конца, это означает, что данные целиком заполнили массив и требуется его расширить. Для удаления символа просто увеличивается индекс начала, при этом первый символ строки выпадает из области между началом и концом. Учитывая цикличность, если при увеличении индекс превысил конец массива, требуется установить его в начало. Если индекс начала при увеличении достиг индекса конца, то строка считается пустой.

Длина строки на циклической очереди вычисляется как разность индекса конца и индекса начала по модулю размера массива.

1.2.4. Неизменяемые строки

Одним из вариантов реализации хронологических структур данных является концепция неизменяемости. Неизменяемый объект не имеет механизмов для изменения своего внутреннего состояния и, будучи однажды созданным, больше никогда не меняется. Неизменяемость автоматически означает абсолютную хронологичность. Эта концепция широко используется в стандартной библиотеке языка Java. В книге [12] описаны преимущества неизменяемых объектов в контексте языка Java. Среди преимуществ есть безопасность с точки зрения многопоточности, возможность безопасного совместного использования таких объектов, удобство использования при проектировании более сложных объектов.

Строки являются одним из неизменяемых объектов языка Java. Компания Sun Microsystems предоставляет свободный доступ к исходным кодам своей библиотеки классов языка Java. Исходные коды позволяют разобраться во внутреннем устройстве строк.

Объект неизменяемой строки содержит ссылку на массив символов, смещение в этом массиве и длину данной строки. Последовательностью символов строки является содержимое массива с заданного смещения и до достижения требуемой длины. Поскольку длина строки хранится в явном виде, операция ее получения требует константного времени.

Операции итерирования и индексации являются простым обращением к определенному элементу массива, выполняемым за константное время. Индекс в массиве получается сложением индекса в строке и смещения строки относительно начала массива.

Отличием от изменяемых строк является неизменяемость, выражающаяся в частности в том, что запрещены изменения содержимого массива. Поэтому для замены символа, вставки и удаления требуется создание нового массива. Эти операции требуют времени и памяти пропорционально длине результата. Такую же оценку времени и памяти имеет операция конкатенации, поскольку также требует создание нового массива для копирования в него конкатенируемых последовательностей.

Из сказанного может сложиться впечатление, что для достижения хронологичности строк языка Java в жертву принесена эффективность. Это впечатление не верно. Гарантии неизменности внутреннего массива символов обеспечивают преимущество при получении подстроки. Объект результата операции получения подстроки может ссылаться на тот же массив, что и исходная строка. В результате операция сводится к копированию ссылки на массив и установки правильного смещения и длины, а это требует константного времени и памяти. Таким образом, неизменность позволила существенно повысить эффективность одной из основных операций со строками.

1.2.5. Строки на основе изменяемых списков

У Д. Кнута [11] описан ряд структур, появившихся так давно и имеющих такое широкое распространение, что считаются уже компьютерным фольклором. Одной из таких структур является двусвязный список. Рассмотрим возможность реализации строк на основе двусвязных списков.

Напомним, что двусвязный список состоит из последовательности элементов. Каждый элемент помимо полезной информации хранит в себе

ссылки на два соседних элемента. Сам список задается ссылками на первый и последний элементы. Обычно списки являются изменяемой структурой данных.

Полезной информацией, хранящейся в элементах списка, в случае строк являются отдельные символы. А вся последовательность элементов списка образует последовательность символов строки. Итерирование является операцией свойственной спискам. Наличие ссылок на следующий и предыдущий элементы позволяет просто переходить между ними за константное время. В отличие от массивов для списков не существует механизма адресации, поскольку отдельные элементы списка могут неупорядоченно располагаться в различных местах памяти. Единственным способом добраться до нужного элемента является итерирование от начала или конца списка. Худшим случаем для операции индексации является обращение в середину строки. Такое обращение вызовет необходимость прохода по половине элементов списка и займет времени пропорционально длине строки.

Для получения подстроки для начала требуется найти то место в списке, где начинается запрашиваемая подстрока. Как уже было сказано, поиск элемента в списке имеет линейную трудоемкость. После нахождения нужного элемента необходимо посетить все элементы, соответствующие подстроке, и выполнить их копирование в новый список. Этот этап требует времени пропорционального длине результата.

Обычно в явном виде длина списка не хранится. Поэтому для ее получения требуется обход и подсчет всех элементов, количество которых равно длине строки. Вычисление длины строки получается довольно дорогой операцией. Эту ситуацию можно обойти, если хранить длину строки вместе со списком. Поддержание корректного значения длины при выполнении различных операций потребует всего лишь константного количества лишних действий и не повлечет за собой изменения асимптотики оценок времени.

Вставка и удаление являются простыми операциями для двусвязных списков. Для удаления элемента правая ссылка левого соседа данного элемента устанавливается на правого соседа, а левая ссылка правого соседа – на левого. Таким образом, элемент выпадает из списка. При вставке элемента его просто связывают ссылками с его новыми соседями. Все эти манипуляции производятся с константным числом ссылок и поэтому требуют константного количества времени. Однако для вставки и удаления конкретного элемента его необходимо сначала найти, а поиск обладает линейной от длины списка сложностью. Только вставки и удаления по краям строки, где не требуется поиск, обладают константной оценкой сложности.

Также как и в случае неизменяемых строк, структура строк на двусвязных списках позволяет получить лучшую оценку времени для одной из операций. Этой операцией является конкатенация. При наличии двух списков для объединения их в один достаточно связать ссылками конец одного и начало другого. Началом нового списка становится начало первого, а концом – конец второго. Эта простая операция состоит из фиксированного количества действий со ссылками. Таким образом, достигается константная оценка времени операции конкатенации для строк основанных на двусвязных списках. Следует отметить, что в данной работе рассматривается всего две структуры, обладающие такой эффективностью выполнения конкатенации.

1.2.6. Строки на основе неизменяемых списков

В функциональных языках программирования [13] выполнение программы всецело состоит из вычисления выражений. Чисто функциональные языки вообще не содержат механизмов изменения объектов, поэтому все объекты автоматически являются неизменяемыми, а значит хронологическими. Как и во многих языках программирования, в функциональных языках существуют стандартные конструкции для строк. Рассмотрим функциональный вариант представления строковых данных на примере языка Хаскель, принадлежащего классу функциональных языков.

Язык Хаскель [14] содержит базовый тип для представления строковой информации, языковые конструкции для задания строковых литералов, функции для работы с объектами строкового типа. Фактически же строковый тип является не совсем базовым. В его основе лежит другой тип – список. В частности строки принимаются функциями, требующими в качестве аргумента список. Элементами списка, лежащего в основе строки, являются символы.

Список в языке Хаскеле описывается рекурсивно. Он состоит из своего первого элемента и списка, содержащего оставшуюся часть. База рекурсии – это пустой список, который ни из чего не состоит. Введенная для удобства нотация с записью элементов списка через запятую в квадратных скобках является лишь сокращением для рекурсивного определения списка. По сути, списки в Хаскеле являются обычными односвязными списками, с той лишь разницей, что они являются неизменяемыми.

Для изменения элемента неизменяемого списка сначала требуется создать измененную копию этого элемента, поскольку сам элемент изменен быть не может. Далее требуется создать копию элемента, предшествовавшего ему в списке, чтобы он связан с измененным элементом. И так далее до начала списка. В результате будет создана копия начала списка, а хвост останется прежним. Таким образом, неизменяемые списки образуют деревья, в листьях которых находятся начала списков, а корнем является элемент общий для всех списков, входящих в дерево.

С точки зрения строковых операций неизменяемые списки обладают теме же набором проблем, что и изменяемые двусвязные списки: необходимость вычислять длину, затраты на копирование при получении подстроки, линейная сложность индексации и других операций, связанных с нахождением конкретного места в строке. Дополнительную проблему создает несимметричность структуры, вызванная односвязностью списка. Эффективность работы с началом строки не изменяется. Но для оперирования с концом строки необходимо как минимум пройти через

список от начала до конца, а в случае внесения изменений придется выполнить копирование всего списка целиком.

Требование неизменяемости вносит корректировку и в алгоритм конкатенации. В случае изменяемых списков для конкатенации достаточно было изменить последний элемент первого списка и первый элемент второго. В неизменяемых списках ссылки направлены только в одном направлении, поэтому достаточно изменить только последний элемент первого списка так, чтобы он указывал на первый элемент второго. Однако такое изменение повлечет за собой необходимость копирования всего первого списка. Таким образом, конкатенация строк на неизменяемых списках требует количество действий пропорциональное длине первой из конкатенируемых строк.

1.2.7. Анализ существующих методов представления строк

Сведем оценки времени работы операций для различных методов представления строковой информации в единую таблицу (табл. 1).

Таблица 1. Оценки времени работы строковых операций

Операция	Часть строки	Изменяемые строки	Неизменяемые строки	Строки на циклических очередях	Строки на двусвязных списках	Функциональные строки
Получение длины		O(1)	O(1)	O(1)	O(1) *	O(1) *
Получение подстроки длины m	Начало	O(m)	O(1)	O(m)	O(m)	O(m)
	середина	O(m)	O(1)	O(m)	O(n + m)	O(n + m)
	Конец	O(m)	O(1)	O(m)	O(m)	O(n)
Конкатенация со строкой длины m		O(n + m)	O(n + m)	O(m)	O(1)	O(n)
Итерирование		O(1)	O(1)	O(1)	O(1)	O(1)
Индексация	Концы	O(1)	O(1)	O(1)	O(1)	O(1) **
	середина	O(1)	O(1)	O(1)	O(n)	O(n)
Вставка символа	Начало	O(n)	O(n)	O(1)	O(1)	O(1)
	середина	O(n)	O(n)	O(n)	O(n)	O(n)
	Конец	O(1)	O(n)	O(1)	O(1)	O(n)
Удаление символа	Начало	O(n)	O(1)	O(1)	O(1)	O(1)
	середина	O(n)	O(n)	O(n)	O(n)	O(n)
	Конец	O(1)	O(1)	O(1)	O(1)	O(n) ***
Замена символа на символ	Начало	O(1)	O(n)	O(1)	O(1)	O(1)
	середина	O(1)	O(n)	O(1)	O(n)	O(n)
	Конец	O(1)	O(n)	O(1)	O(1)	O(n)
Вставка строки длины m	Начало	O(n + m)	O(n + m)	O(m)	O(m)	O(m)
	середина	O(n + m)	O(n + m)	O(n + m)	O(n + m)	O(n + m)
	Конец	O(m)	O(n + m)	O(m)	O(m)	O(n)

* Для достижения такой оценки требуется хранение в явном виде избыточной информации о длине строки.

** Индексация в начале строки имеет константную оценку времени. Однако в конце строки для достижения такой оценки требуется дополнительное хранение последнего символа в явном виде.

*** В разделе о хронологическом стеке будет предложен способ улучшения этого времени до O(1).

Светлой заливкой выделены ячейки таблицы соответствующие оптимальной среди рассмотренных методов оценке времени для данной операции, но только если существуют методы, обладающие более плохим временем работы. Таблица наглядно демонстрирует различия в производительности методов представления строковой информации.

1.3. Примеры хронологических структур данных

В результате исследований в области хронологических структур данных за последние двадцать лет было разработано большое число различных структур данных, обладающих тем или иным видом хронологичности, предложен ряд универсальных методов получения хронологических аналогов изменяемых структур. Этот раздел посвящен краткому обзору

хронологических структур данных, которые будут использованы в данной работе.

1.3.1. Хронологический стек

Согласно работе [11] стек – это линейный список, в котором все операции вставки, удаления и доступа к значениям выполняются только на одном из концов списка. Дейкстрой была предложена аналогия с железнодорожным разъездом для наглядного объяснения механизма работы стека. С одной стороны такого железнодорожного разъезда находится тупик, а с другой производится въезд и выезд вагонов. Таким образом, реализуется принцип «последним пришел – первым вышел».

Существуют довольно сложные реализации хронологического стека. Например, абсолютно хронологическая структура, предложенная в работе [15], помимо операций, свойственных стеку, поддерживает также доступ к произвольным элементам. Операции удаления из конца и вставки в конец выполняются за константное время, а доступ к произвольному элементу выполняется в худшем случае за $O(\log(i))$, где i – индекс запрашиваемого элемента.

Однако в данной работе требуется только стек с обычным набором операций, поэтому использование сложных структур неоправданно. Каплан и Тарьян упоминают в работе [2] более простой подход для получения хронологического стека. Стек может быть представлен в виде ссылки на первый элемент односвязного списка. Похожая структура уже была рассмотрена в разделе о строках, основанных на неизменяемых списках. Опишем стековые операции с такой структурой.

Верхним элементом стека считается тот, на который указывает ссылка стека (начало списка). Таким образом, для удаления верхнего элемента достаточно передвинуть ссылку стека на следующий элемент списка. При этом старый стек целиком сохраняется, поскольку список не был изменен, а новый стек будет иметь на один элемент меньше.

Для добавления значения на верхушку стека требуется создать новый элемент, который будет хранить в себе новое значение. Ссылка этого элемента, которая должна указывать на следующий элемент в списке, устанавливается на старую вершину стека. После этого указатель стека устанавливается на вновь созданный элемент.

Далее в работе дополнительно к стандартным операциям от хронологического стека потребуется операция удаления определенного количества нижних элементов стека. При обычном подходе эту операцию можно выполнить в два этапа. Сначала со стека по одному снимаются элементы и складываются во временный стек. Когда в исходном стеке остаются только удаляемые элементы, временный стек перекладывается в новый стек, который и становится результатом операции. Временный стек необходим потому, что при перекладывании элементов из одного стека в другой последовательность элементов переворачивается, а при перекладывании через временный стек последовательность переворачивается дважды, и элементы остаются в правильном порядке. Такой алгоритм имеет линейную от количества элементов в стеке оценку сложности.

Можно модифицировать структуру стека так, чтобы операция удаления нижних элементов выполнялась более эффективно. Для этого будем хранить количество элементов в стеке в явном виде. Поддержание актуального значения этой величины вызовет лишь фиксированное количество лишних действий на каждую операцию со стеком. Когда в варианте без хранения длины стек опустошался, его ссылка начинала указывать на элемент, следующий за последним элементом стека. За последним элементом других элементов не было, поэтому стек считался пустым тогда, когда его ссылка указывала никуда. В варианте же с хранением длины критерием опустошения стека должно являться нулевое значение длины. Тогда для удаления x нижних символов стека достаточно уменьшить значение хранимой длины на x.

1.3.2. Хронологическое декартово дерево

Арагоном и Зейделем в 1989 году была изобретена структура данных трип [16]. Название структуры полностью отражает ее суть. Оно образовано от слов `tree` и `heap`, которые в переводе с английского языка означают дерево и куча соответственно.

Структура состоит из набора вершин, образующих двоичное дерево. Каждая вершина обладает двумя ключами. По первому ключу дерево является двоичным деревом поиска. По второму ключу, назовем его приоритетом, дерево должно удовлетворять условию кучи, которое требует, чтобы приоритет каждого из детей вершины был не больше приоритета самой вершины. Для вершины v первый ключ обозначим $k[v]$, а второй – $p[v]$. Тогда для вершины v, ее левого ребенка l и ее правого ребенка r можно записать условия в виде неравенств:

- $k[l] \le k[v] \le k[r]$;
- $p[l] \le p[v]$;
- $p[r] \le p[v]$.

Доказано, что если в качестве значений приоритета выбираются случайные числа, то математическое ожидание глубины пути от корня до вершин равно логарифму количества вершин в дереве. Первый ключ выбирается произвольно, поэтому по нему дерево может рассматриваться как двоичное дерево поиска. За счет логарифмической оценки ожидаемой глубины операции с таким деревом в среднем требуют времени пропорционально логарифму размера дерева. Главным отличием трипа от сбалансированных деревьев поиска является то, что сбалансированные деревья имеют логарифмические оценки времени выполнения операций в худшем случае, а для трипа оценки вероятностные.

Те же авторы в 1996 году опубликовали работу [1], в которой идея трипа получила свое дальнейшее развитие. В работе было показано, что можно отказаться от явного хранения ключа, по которому дерево является кучей. Вместо хранения приоритеты генерируются заново по мере необходимости.

При каждом сравнении двух вершин вместо приоритетов сравниваются два новых случайных числа. Если правильно выбрать распределения этих случайных чисел, то сохраняется логарифмическая оценка на глубину дерева в среднем. Получившаяся структура была названа рандомизированным двоичным деревом поиска.

Операции дерева поиска для этой структуры реализуются через две базовых операции – разделение и конкатенация. Разделение по заданному ключу делит дерево на два так, что в первом из них находятся все элементы, у которых значение ключа меньше заданного, а во втором – те, у которых больше. Конкатенация объединяет два дерева в одно при условии, что максимальный ключ элементов первого дерева меньше минимального ключа второго.

В терминах разделения и конкатенации удаление элемента – это разделение дерева по ключу удаляемого элемента с последующей конкатенацией двух получившихся деревьев. Для добавления элемента дерево также сначала разделяется по ключу добавляемого элемента, а затем конкатенируется первое получившееся дерево с деревом, состоящим из одного добавляемого элемента, и результат конкатенируется со вторым деревом.

Предложенный в работе [3] метод копирования узлов позволяет получить хронологическую структуру из эфемерной структуры, основанной на ссылках. Суть метода состоит в том, что вместо модифицирования узла структуры создается его модифицированная копия. Если на узел в структуре ссылались другие узлы, они также должны быть скопированы, чтобы их ссылки вели в новую версию модифицированного узла. Рассмотрим метод на примере базовых операций рандомизированного двоичного дерева поиска.

Входными данными алгоритма разделения является ссылка на корень дерева, которое требуется разделить, и ключ, по которому разделение будет проводиться. На выходе алгоритма получается две ссылки. Одна указывает на корень дерева, содержащего все вершины с ключом меньшим заданного,

вторая – дерева с ключами большими заданного. Опишем алгоритм. Если дерево пусто, то возвращается два пустых дерева. Если ключ дерева равен ключу разделения, то в качестве результата возвращаются левое и правое поддеревья корня. Иначе возможны два варианта: ключ корня меньше ключа разделения и больше ключа разделения. Ввиду симметричности случаев рассмотрим первый. Разделим рекурсивным вызовом правое поддерево с тем же ключом разделения. Правый результат такого разделения вернем в качестве правого дерева ответа. Левое дерево ответа создается из нового корня со значением и левым поддеревом совпадающими со значениями старого корня всего дерева, а правым поддеревом устанавливается левый результат рекурсивного разделения. Поскольку все изменения проводились со вновь созданными вершинами, то старое дерево осталось неизменным, следовательно, операция обеспечила хронологичность дерева.

Конкатенация выполняется над двумя деревьями, обозначим L и R. Результатом конкатенации является дерево, объединяющее все элементы входных деревьев. Для соблюдения условия кучи корневым элементом результата станет тот из двух корней, который обладает большим приоритетом. Работа [1] предлагает выбирать вместо ключей случайные числа из диапазона от нуля до размера поддерева соответствующей вершины. Ввиду симметричности случаев рассмотрим случай, когда корень дерева L оказался больше при сравнении приоритетов. Создадим копию этого корня, она станет результатом операции. Ссылка на левое поддерево у результата сохраняется, а правым поддеревом устанавливается результат рекурсивной конкатенации старого правого поддерева и дерева R.

Каждый рекурсивный вызов этих алгоритмов требует константного времени и сокращает высоту дерева на один. Всего высота в среднем пропорциональна логарифму числа вершин в дереве, следовательно, сложность операций с рандомизированным двоичным деревом поиска можно оценить логарифмом числа вершин. Доказательство ожидания высоты можно посмотреть в работах [1, 16].

1.3.3. Хронологическое 2-3-дерево

Основное преимущество рандомизированных деревьев состоит в простоте реализации. Однако они неприменимы в случаях, когда требуется гарантированная оценка сверху времени работы алгоритма, поскольку обладают логарифмической оценкой только в среднем, в худшем же случае высота рандомизированного дерева может составлять величину порядка количества вершин. Там, где требуется гарантированная оценка, больше подходят сбалансированные деревья.

Одним из видов сбалансированных деревьев являются 2-3-деревья, изобретенные Хопкрофтом в 1970 году [17]. 2-3-дерево – это B-дерево [18], внутренние вершины которого имеют по два или три ребенка. Поскольку все листья находятся на одной глубине, дерево является сбалансированным. Высота дерева ограничена логарифмом по основанию два от количества вершин.

Помимо стандартных для деревьев поиска операций добавления, поиска и удаления элементов 2-3-деревья поддерживают операции разделения и конкатенации. Основная идея сохранения корректности структуры 2-3-дерева при совершении операций состоит в передаче детей между соседними вершинами. Если у внутренней вершины остался только один ребенок, он вместе с поддеревом перевешивается на соседнюю вершину, а сама вершина удаляется. Если у внутренней вершины стало четыре ребенка, она расщепляется на две, а дети делятся между новыми вершинами пополам.

Хронологичность 2-3-дерева обеспечивается методом копирования вершин, уже рассмотренным на примере рандомизированных двоичных деревьев поиска. Хопкрофтом показано, что при совершении операций разделения и конкатенации над 2-3-деревом совершается порядка логарифма от количества вершин действий. Следовательно, количество вершин

подлежащих копированию асимптотически может быть оценено сверху логарифмом количества вершин.

1.3.4. Невозможность объединяемо хронологической структуры за O(1)

Фиат и Каплан в работе [19] доказали, основываясь на простых аргументах из теории информации, что для любого метода приведения к объединяемо хронологическому виду существует ациклический граф версий, который не может быть представлен с использованием константного количества памяти на каждое присваивание. Ими было определено понятие эффективной глубины ациклического графа версий, как логарифм количества различных путей в этом графе, начинающихся в корне. В их работе было показано, что для объединяемо хронологических структур данных, полученных каким либо общим методом, количество битов информации, необходимых для представления одного присваивания, не меньше эффективной глубины ациклического графа версий. Для использования памяти требуется время как минимум пропорциональное использованной памяти.

1.4. Постановка задачи

Цель данной работы – исследовать возможность эффективного представления строковых данных. Рассматриваемым критерием эффективности является временная сложность стандартных операций со строками. Также должны быть учтены затраты используемой памяти.

Анализ существующих методов представления строковой информации показал, что в зависимости от условий использования эффективность того или иного метода может существенно отличаться. Многие методы, обладая константной сложностью одних операций, для остальных операций требуют времени пропорционального длине строки. В данной работе будет предпринята попытка разработать структуру данных, позволяющую избежать

таких перепадов в производительности и за счет этого получить выигрыш в производительности для условий реальных приложений.

Учитывая рассуждения о хронологических структурах, поставим в качестве одного из условий для разрабатываемой структуры объединяемую хронологичность. Хронологические структуры данных обладают рядом преимуществ перед изменяемыми аналогами. Кроме того, хронологичность может позволить эффективно эксплуатировать схожесть аргументов и результатов таких строковых операций, как конкатенация и получение подстроки.

Для получения оценок производительности, которых можно ожидать от разрабатываемой структуры, проведем рассуждения похожие на рассуждения Фиата и Каплана при доказательстве невозможности хронологической структуры за `O(1)`. Возьмем строку из одного символа. Конкатенируем эту строку саму с собой. Результат опять конкатенируем с собой. Продолжение процесса приведет к получению строки экспоненциальной длины от числа шагов. При эффективной операции конкатенации может получиться строка, длина которой превысит любой заранее заданный размер памяти, поэтому схемы с прямой адресацией в такой ситуации не применимы, а индексация, основанная на последовательном выборе из нескольких ссылок, потребует порядка логарифма действий.

Таким образом, задача состоит в получении структуры, позволяющей представлять строковую информацию и выполнять над ней строковые операции за время порядка логарифма. Для выполнения каждой операции допустимо использование памяти не более чем порядка логарифма от длины.

В практике программирования автора довольно часто встречаются ситуации, когда строки конструируются путем последовательного добавления небольшого количества символов в начало или в конец строки. Рассуждения, обосновывающие логарифмическую оценку времени операций, справедливы только для всей строки целиком. Если рассматривать концы строки отдельно, есть надежда, что путем дополнительных усилий удастся

добиться выполнения операций с концами быстрее, чем за логарифм. Попробуем достичь оценки `O(1)` для операций добавления и удаления символов на концах строки.

Глава 2. Строки на основе хронологических деревьев

В этой главе предложено развитие идеи представления строк с использованием деревьев. Для достижения желаемых логарифмических оценок времени применяется балансировка деревьев. Помимо общего метода, позволяющего использовать любые хронологические сбалансированные деревья поиска, предложены также два конкретных алгоритма балансировки.

2.1. Общая идея реализации

Одной из первых реализаций идеи представления строковой информации в виде древовидной структуры была организация строковых данных в языке программирования Кедр [20]. Идея состояла в том, чтобы максимально ускорить скорость конкатенации. Для представления текста в Кедре есть два типа данных. Один из них хранит обычный текст, как последовательность символов, располагающихся подряд в памяти. Второй тип возникает при попытке конкатенации двух строк и содержит лишь ссылки на конкатенируемые строки, которые могут быть как обычным текстом, так и результатом других конкатенаций. Таким образом, строка в Кедре является деревом, узлы которого означают операцию конкатенации, а в листьях располагается текстовая информация. Пример такого дерева изображен на рисунке 1.

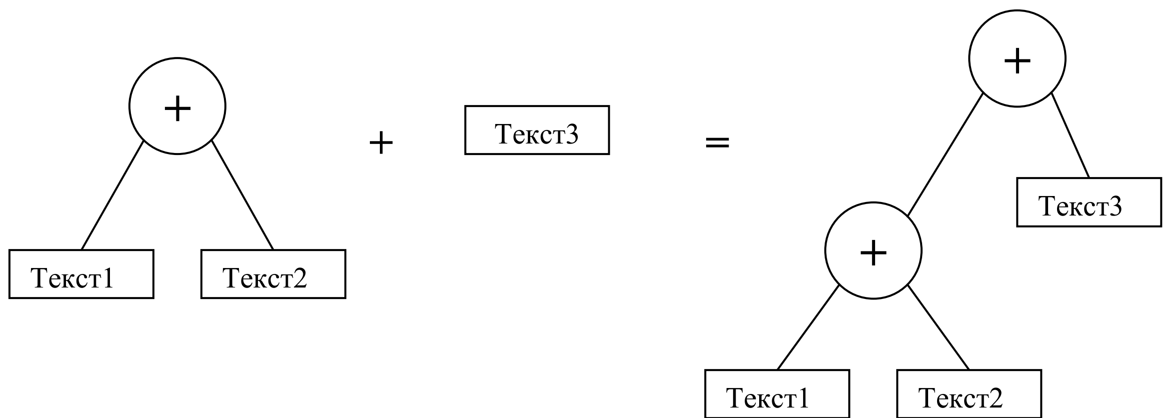

Рис. 1. Представление конкатенации узлами дерева

Тип данных, который представляет в Кедре конкатенацию, называется rope. Русскоязычного варианта этого термина найти не удалось, поэтому в данной работе будет использоваться термин связка, как перевод, наиболее точно отражающий суть структуры. Связки появились в языке Кедр с момента создания, однако не были широко известны. В публикациях, посвященных языку (например, в работах [21, 22]), обычно лишь упоминались связки, но приводилось их подробного описания.

Связки позволяют выполнять конкатенацию за O(1). Однако чаще всего в результате многих конкатенаций получается вытянутое дерево (рис. 2), глубина которого пропорциональна количеству вершин. Попытка доступа к данным в нижней части дерева приводит к необходимости пройти по всему пути от корня и занимает время линейное от числа вершин. В статье [23] предлагается балансировать связки так, как это делается в сбалансированных деревьях поиска. Для достижения амортизированной оценки времени выполнения операций O(log n) авторами статьи предлагается проводить полную балансировку связок при наступлении определенных условий.

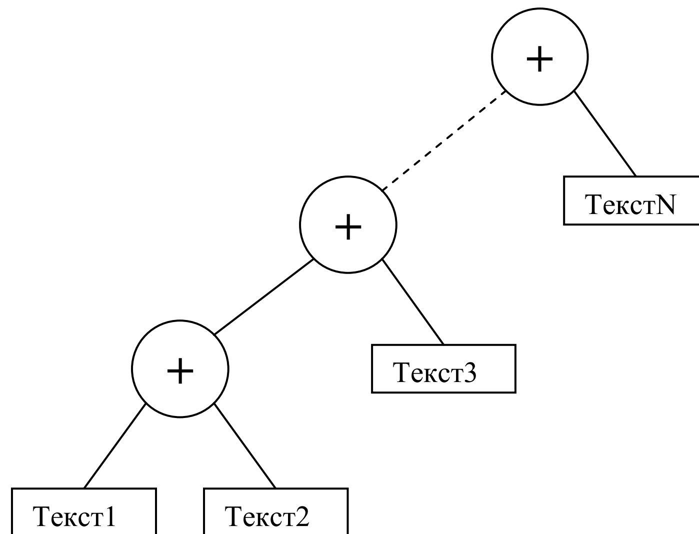

Рис. 2. Пример вытянутого дерева

Минус подхода с полной перебалансировкой связки состоит в том, что связки являются неизменяемыми. Представим связку, в которой почти выполнено условие необходимости балансировки. Поскольку при совершении операции с этой связкой, она остается неизменной, операции с ней могут производиться несколько раз, каждый раз вызывая полную балансировку. Логарифмическая оценка в таком случае оказывается неверной.

Рассмотрим возможность постоянного поддержания связки в сбалансированном состоянии. Если представить строку, как набор индексов от 0 до $n - 1$, где n – длина строки, каждому из которых поставлен в соответствие определенный символ, то такой набор данных может храниться в сбалансированном дереве поиска. Индекс в таком случае является ключом, а символ – значением. Посмотрим, как строковые операции могут выполняться со строкой, хранящейся в таком виде.

2.1.1. Описание операций

Обычно в вершинах дерева хранится размер поддерева в явном виде. Если это не так, то требуется дополнить алгоритм механизмами, поддерживающими информацию о размере поддерева в вершинах. Это позволит получать размер строки, представленной деревом, обращением к корню. Поскольку поддеревом корня является все дерево, размер этого поддерева совпадает с размером строки.

Для получения конкретного символа строки необходимо найти вершину дерева, в которой хранится требуемый символ. Поиск значения по индексу является одной из основных задач, решаемых с помощью деревьев поиска. Поэтому операция индексации напрямую представляется через операцию поиска по дереву. То же самое касается операций удаления символа, реализующейся через удаление элемента из дерева по его ключу, и вставки символа, соответствующей добавлению нового элемента в дерево. Операция замены одного символа на другой может быть реализована через последовательное удаление старого символа и вставки нового.

Деревья поиска организованы так, что если выписать все ключи в порядке обхода по правилу левый-корень-правый, получается возрастающая или убывающая последовательность. Предположим, что последовательность получится возрастающая, иначе следует ее развернуть. Если в дереве рассматриваемым способом хранится строка, то это будет последовательность индексов от 0 до $n - 1$. Следовательно, если выписывать не ключи, а соответствующие символы, результатом обхода будет последовательность символов строки в правильном порядке. Таким образом, процесс итерирования по строке может быть получен в ходе обхода дерева.

Некоторые деревья поиска поддерживают операции слияния и разделения. К таким деревьям относятся, уже упоминавшиеся в первой главе, рандомизированные деревья поиска и 2-3-деревья. Выясним, как на основе этих операций могут быть реализованы конкатенация и получение подстроки.

Слияние обеспечивает объединение множеств элементов двух деревьев, но только в случае, если интервалы, в которых лежат значения ключей каждого из деревьев, не пересекаются. Для деревьев представляющих строки такое условие на ключи не выполняется. Допустим, одно из деревьев содержит строку длины n, а другое – длины m. Напрямую слить эти деревья нельзя, поскольку интервалы $(0, n - 1)$ и $(0, m - 1)$ пересекаются. Если произвести конкатенацию этих строк, то в строке-результате индексы символов первой строки останутся теми же, какими они были до конкатенации. А индексы символов второй строки будут лежать в интервале $(n, n + m - 1)$. Если выполнить увеличение на n всех ключей в дереве, представляющим вторую строку, то интервалы станут непересекающимися и появится возможность выполнения слияния деревьев. В объединенном дереве символы из первого дерева будут иметь индексы от 0 до $n - 1$, а символы из второго дерева – индексы от n до $m + n - 1$. Такое расположение в точности совпадает с расположением символов в результате конкатенации двух строк. Остается только вопрос о том, как эффективно увеличить все ключи в дереве. Для этого можно воспользоваться следующим приемом. Поставим в корень метку о том, что в его поддереве все ключи необходимо увеличить на n. Дальше при любой попытке доступа к вершине, обладающей такой меткой, необходимо сначала протолкнуть метку вниз. Для этого ключ вершины увеличивается на значение, записанное в метке, с самой вершины метка снимается, а детям вершины ставятся точно такие же метки. Если в какой-то вершине уже есть метка, то числа, записанные на метках, суммируются.

Рассмотрим операцию получения подстроки с *i-го* по *j-ый* символ. Эту операцию можно представить как последовательность из двух разделений дерева по ключу. Сначала дерево делится по ключу $j + 1$. В результате получается два дерева u_1 и u_2. Дерево u_1 содержит символы с 0-го по *j-ый*. А дерево u_2 оставшиеся символы строки, которые не представляют интереса. Дальше дерево u_1 требуется разделить по ключу $i - 1$. Результатом будут

являться два дерева v_1 и v_2. Первое из них содержит символы с 0-го по (i - 1)-ый, они больше не потребуются. А второе – с *i-го* по *j-ый*, что совпадает с требуемыми по условию операции получения подстроки границами. Однако дерево v_2 не может считаться результатом операции получения подстроки, поскольку не соответствует используемой структуре строк. Индексы в v_2 начинаются с i, а не с нуля. Для завершения операции получения подстроки требуется уменьшить все индексы в дереве на i, а для этого можно воспользоваться приемом аналогичным применявшемуся при конкатенации.

2.1.2. Оценка времени операций

Для оценки времени, необходимого на выполнение операций со строками, следует отметить связь количества вершин в дереве, представляющем строку, и длиной этой строки. В случае, когда каждый символ представлен парой ключ-значение, отдельные символы представлены разными вершинами, а каждая вершина хранит ровно по одному символу. Следовательно, число вершин в дереве равно количеству символов в строке, а значит ее длине. Теперь оценки времени выполнения операций со строками от длины напрямую связаны с оценками времени выполнения соответствующих операций с деревьями от размера.

Глубина сбалансированных деревьев составляет величину порядка логарифма от числа вершин. Поэтому многие операции со сбалансированными деревьями выполняются за время пропорциональное логарифму вершин. К таким операциям относятся добавление, поиск и удаление элементов. С деревьями, поддерживающими операции слияния и разделения, эти операции также выполняются за логарифмическое время.

Операции индексации, удаления символа, добавления символа, замены символа на символ напрямую отображаются в операции вставки, поиска и удаления элементов из дерева. Поэтому выполняются за логарифм от длины строки. С получением подстроки и конкатенацией ситуация чуть сложнее. Помимо операций слияния и разделения, выполняемых за логарифмическое

время, требуется увеличение и уменьшение ключей в дереве. Операция изменения всех ключей в дереве состоит в присвоении корню специальной метки. Такое действие может быть выполнено за константное время. Заметим, что все последующие перемещения этих меток выполняются во время других операций. Оценим вклад этих дополнительных действий в трудоемкость остальных операций. Перемещение метки вниз требует $O(1)$ времени. На каждое перемещение приходится одна попытка доступа к вершине, которая даже без перемещения требует не менее чем константного времени. Таким образом, если операция без перемещения меток требовала $O(f(n))$ времени, то все перемещения также можно оценить $O(f(n))$. Следовательно, вклада в асимптотическую оценку времени работы остальных операций перемещение меток не вносит.

Получение следующего символа при итерировании заключается в переходе к следующей вершине по порядку обхода. Заметим, что расстояние по ребрам дерева между двумя последовательными вершинами в обходе может оказаться довольно большим. Например, в сбалансированном двоичном дереве расстояние от самой правой вершины левого поддерева корня до корня пропорционально высоте дерева, хотя эти две вершины являются соседними в смысле обхода. Несмотря на то, что отдельные переходы между соседними вершинами могут требовать более чем $O(1)$ времени, суммарно итерирование по всем символам строки займет $O(n)$ времени. Докажем это утверждение. Алгоритм обхода дерева использует каждое ребро дважды – один раз при переходе по направлению от корня к листьям, второй раз при возвращении обратно. Известный факт из теории графов утверждает, что ребер в дереве на одно меньше вершин. Следовательно, всего обход потребует $2 * (n - 1)$ переходов. Получается, что итерирование по всей строке занимает $O(n)$ времени. Следовательно, несмотря на долгие переходы между некоторыми парами символов, амортизировано получение следующего символа при итерировании имеет оценку времени $O(1)$.

2.1.3. Оценка требуемого количества памяти

Неизменяемые строки языка Java для обеспечения неизменяемости требуют копирования всей строки почти при всех операциях, связанных с модификацией строки (исключение составляет получение подстроки). Предложенные же сбалансированные связки используют память более эффективно. Это получается за счет того, что при представлении строк, получаемых в результате операций, используется их схожесть со строками-аргументами этих операций. Если изменяется только часть строки, то большая часть оставшейся неизменной информации совместно используется в обеих версиях строки.

Для совершения операций со строками сначала должны быть созданы какие-то стартовые строки, которые будут являться аргументами этих операций. В случае сбалансированных связок несложно построить сбалансированное дерево из заранее известных и отсортированных по ключам элементов. Дерево будет состоять из n вершин и $n - 1$ ребра, где n – длина строки. Каждая вершина может представлять собой запись, содержащую фиксированное количество полей данных. Такая запись для своего хранения требует $O(1)$ памяти. Ребра же вообще могут не храниться в явном виде, а входить в состав записей для вершин в виде ссылок на другие вершины. Всего для хранения дерева, представляющего строку длины n, потребуется $O(n)$ памяти. Столько будет занимать строка в результате создания с нуля, по этому показателю асимптотика метода не отличается от традиционных методов представления строк.

Отличия возникают, когда строка создана в результате какой-либо операции с другими строками. Операции, модифицирующие строку, на самом деле модифицируют лишь часть вершин дерева. Поскольку структура является хронологической, то прямая модификация запрещена. Вместо модифицирования некоторой вершины создается ее копия, а изменения

вносятся в эту скопированную вершину. Это означает, что модификации строки приводят к созданию определенного количества новых вершин. Создание каждой из вершин требует $O(1)$ памяти. Для оценки количества вершин, созданных в результате модифицирования дерева, воспользуемся тем принципом, что за определенное время невозможно использовать более чем линейное от этого времени количество памяти. Вспоминая, что операции удаления символа, вставки символа, замены символа на символ, конкатенации и получения подстроки требуют $O(\log n)$ времени, приходим к выводу, что память необходимая для представления результата этих операций может быть оценена $O(\log n)$.

Операции, не модифицирующие строку, не требуют памяти для представления нового состояния строки. Результат таких операций, как получение длины, индексация и итерирование, может быть представлен одним числом или символом. Можно предположить, что операции требуют $O(1)$ памяти. Для получения длины это верно, а для индексации и итерирования почти верно. Действительно после завершения итерирования или индексации результат может быть размещен в константном количестве памяти. Однако в процессе выполнения может потребоваться дополнительная память. Например, итерирование выполняется эмуляцией обхода в глубину. Для представления состояния обхода в глубину требуется сохранение стека рекурсивных вызовов обхода или аналогичной структуры заменяющей обход в глубину. Глубина дерева составляет $O(\log n)$, следовательно, в промежуточные моменты времени процесс итерирования может потребовать памяти порядка логарифма от длины строки. Можно было бы избежать необходимости хранения стека рекурсии, если бы в вершинах помимо ссылок на детей хранились ссылки вверх по дереву на родительскую вершину. Но хранение ссылки вверх по дереву выводит структуру из разряда ациклических структур данных, основанных на ссылках. Используемые методы получения хронологичности требуют при изменении одного узла полного копирования всех узлов, достижимых из

него. При наличии ссылок вверх достижимыми оказываются все вершины дерева. Это приведет к ухудшению эффективности по памяти и по времени всех строковых операций, а это неприемлемо. Следует отметить, что после завершения итерирования вся дополнительная память может быть освобождена.

2.2. Конкретные реализации

Рассмотрим особенности реализации метода представления строк на основе сбалансированных деревьев для конкретных видов деревьев. Для рассмотрения были выбраны хронологические версии рандомизированных двоичных деревьев поиска и 2-3-деревьев.

2.2.1. Реализация на основе декартовых деревьев

Будем для краткости называть декартовыми деревьями рандомизированные двоичные деревья поиска. Формально эти понятия не совпадают, однако вторые являются эволюционным развитием первых, поэтому такое смешение понятий допустимо. Хронологическая версия декартовых деревьев была выбрана для реализации сбалансированных связок по нескольким причинам. Идея связок легко переносится на структуру декартовых деревьев, поэтому связки на декартовых деревьях просты для понимания. Операции с декартовыми деревьями могут быть реализованы всего несколькими строками программы. Это позволяет обеспечить простоту реализации сбалансированных связок. Также одной из причин является сравнительно небольшая константа, скрывающаяся под оценкой $O(\log n)$, на практике.

Остановимся на реализации более подробно. Обычно какие-то строки, использующиеся в программе, заданны в исходном коде в виде литералов или просто возникают в виде готовой последовательности символов. Воспользуемся идеей связок языка Кедр. Не будем строить для таких строк дерево. Используем для их хранения метод неизменяемых строк, рассмотренный на примере строк языка Java. Однако в отличие от связок

Кедра, где текстовые элементы хранились только в листьях дерева, в наших связках текст будет храниться во всех вершинах. Порядок символов внутри каждой вершины задается строкой, хранимой в этой вершине, а взаимный порядок символов из разных вершин задается по правилу обхода левый-корень-правый. Задание порядка структурой дерева позволяет отказаться от явного хранения ключей. Пример представления строки «СПбГУИТМО» приведен на рис. 3. Для того чтобы дерево было двоичным, отсутствие ребенка у какой-либо из вершин заменяется вершиной типа *null*. Вершина *null* соответствует пустой строке и не хранит в себе никаких данных.

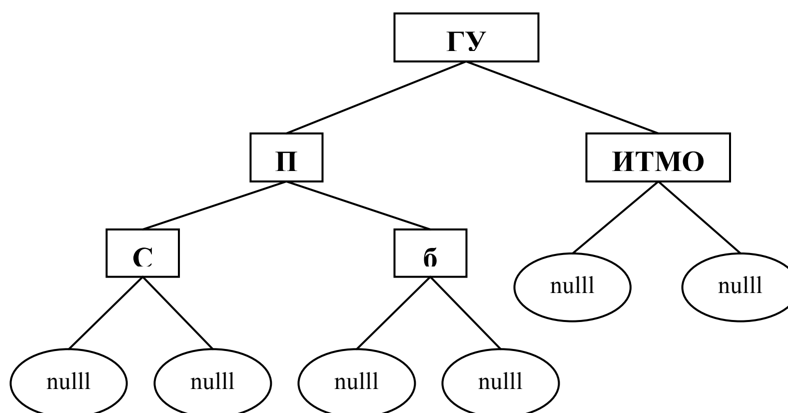

Рис. 3. Представление строки «СПбГУИТМО» на декартовом дереве

Авторы статьи [1] избавились от явного хранения приоритетов в декартовых деревьях. Неявное задание порядка символов связки позволило избавиться и от явного хранения ключей. Возникает вопрос, что же осталось от декартовых деревьев. От декартовых деревьев в данном методе представления строковой информации остается алгоритмы слияния и разделения.

Конкатенация связок выполняется через слияние. Возможны четыре варианта ситуаций при выполнении конкатенации, они проиллюстрированы на рис. 4. Прямоугольниками на рисунке обозначены вершины. Треугольники соответствуют целым деревьям. Вершины *null* изображены овалами. Рассмотрим возможные варианты. Конкатенация любой связки с вершиной *null* не изменяет содержимое связки (рис. 4.а и 4.б). Варианты 4.в и

4.г отражают ситуацию, когда конкатенируются два непустых дерева. Различие состоит в результате сравнения приоритетов вершин *a* и *b*. Ситуации являются симметричными, поэтому остановимся на случае, когда *a* имеет больший приоритет, чем *b* (рис 4.в). Вершина *a* имеет максимальный приоритет в своем поддереве, поскольку она оказалась корнем. Также ее приоритет больше, чем у любой вершины из поддерева *b*, поскольку он больше приоритета *b*, а *b* является корнем своего дерева и имеет там максимальный приоритет. Следовательно, *a* должна стать корнем нового дерева, являющегося результатом конкатенации. Из соображений сохранения порядка символов левым поддеревом *a* становится A_1, а правым поддеревом становится результат рекурсивной конкатенации A_2 и всего дерева *b*.

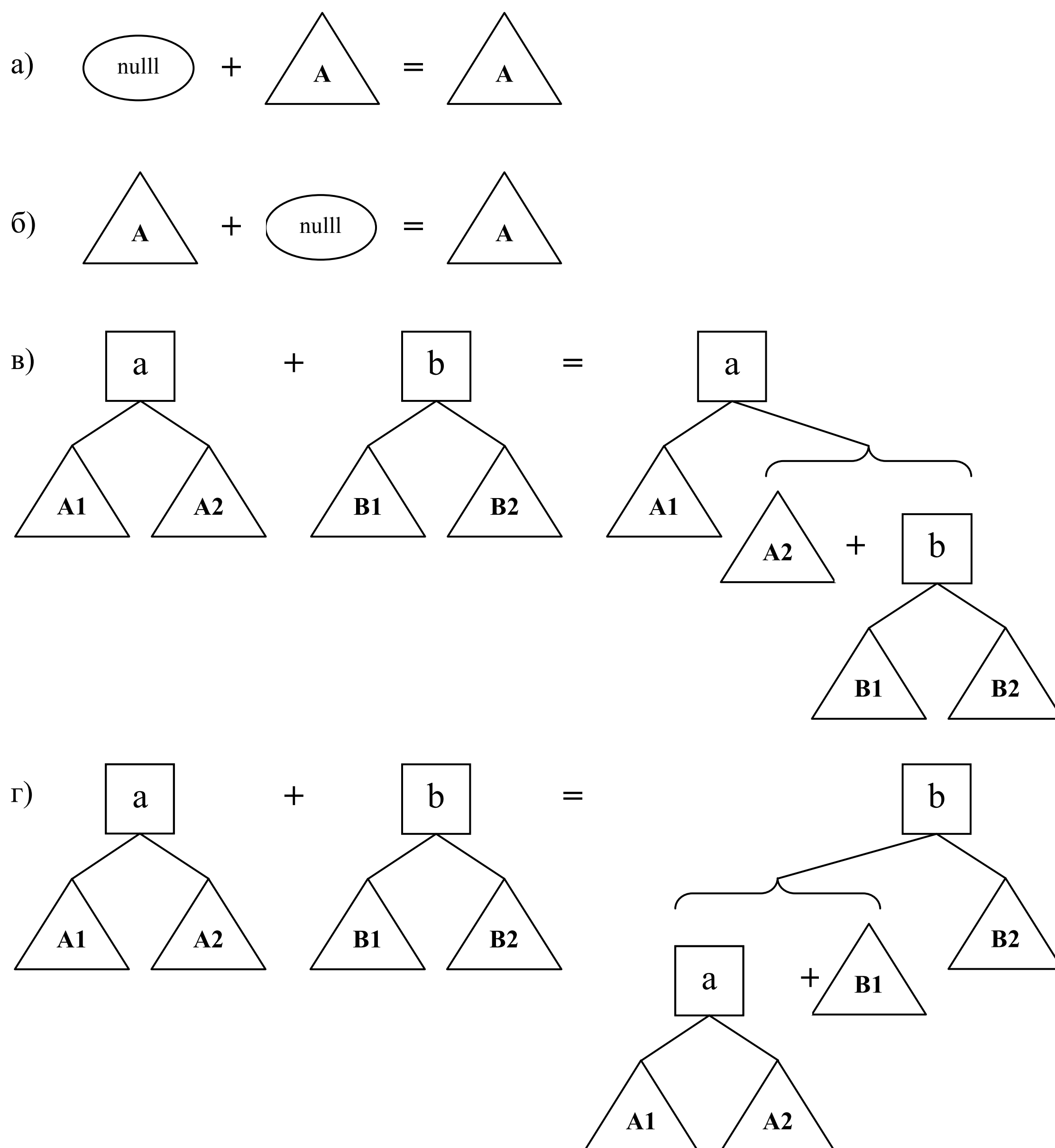

Рис. 4. Конкатенация: а) *null* и *A*; б) *A* и *null*; в) $a > b$; г) $a < b$

Опишем, каким образом производится сравнение приоритетов вершин, если в явном виде приоритеты не хранятся. Рассмотрим множество вершин обоих деревьев. Представим, что каждой из них в качестве приоритета присвоено случайное вещественное число от нуля до единицы. Вероятность p_1 того, что вершина окажется максимальной среди n вершин своего дерева, равна $1/n$. Вероятность p_2 того, что вершина окажется максимальной одновременно среди n вершин своего дерева, а также среди m вершин второго дерева, равна $1/(n + m)$. Согласно формуле условной вероятности

вероятность того, что вершина окажется максимальной среди всех при условии, что она является максимальной среди n из них, равна $p_2 / p_1 = n / (n + m)$. Вернемся к сравнению вершин дерева в ситуации, когда в явном виде приоритеты не заданы. Возьмем случайное число от нуля до единицы. Если оно окажется меньше $n/(n + m)$, где n – длина строки, соответствующей первому дереву, а m – длина строки второго дерева, то будем считать, что корень первого дерева имеет больший приоритет, чем корень второго. В противном случае будем считать приоритет корня второго дерева больше. Несложно заметить, что вероятности исходов в таком случае соответствуют вероятностям исходов, когда для каждого символа приоритет хранился в явном виде.

Как уже отмечалось, операция получения подстроки может быть выполнена с помощью двух разделений. Рассмотрим алгоритм разделения. Он проиллюстрирован на рис. 5. Решение о том, на какую вершину дерева попадает граница раздела, может быть принято на основании информации, хранящейся в каждой вершине, о длине строки, соответствующей поддереву вершины. Если граница раздела попадает в корень дерева (рис. 5.а), то корень раздваивается, а строка, хранящаяся в корне, делиться согласно границе раздела. Левое поддерево корня подвешивается в качестве левого поддерева на левый результат, а правое – в качестве правого на правый результат. Ситуации, когда граница раздела попадает в одно из двух поддеревьев корня (рис 5.б и 5.в), симметричны, поэтому рассмотрим случай, когда граница попадает в левое поддерево. Рекурсивным вызовом левое поддерево корня делится на два дерева A_1 и A_2. Дерево A_1 само по себе является левым результатом операции разделения. Правым результатом операции является исходное дерево, в котором левое поддерево A изменено на A_1.

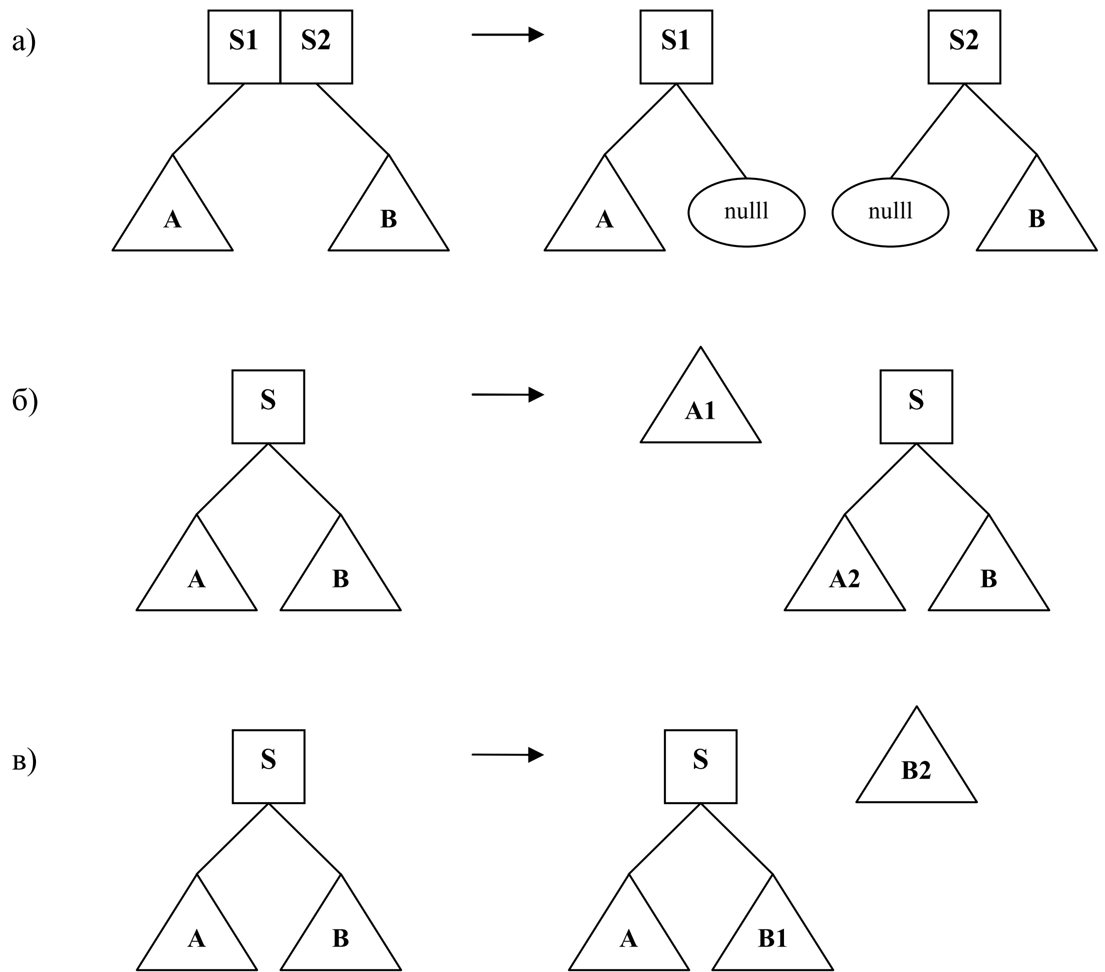

Рис. 5. Разделение: а) в корне; б) в левом поддереве; в) в правом поддереве

Длиной вершины назовем длину строки, соответствующей поддереву этой вершины. Длина может поддерживаться следующим образом. При создании новой вершины ее длина устанавливается, как сумма длин левого ребенка, правого ребенка и длины строки хранящейся в данной вершине. Индексация выполняется аналогично тому, как делается поиск для двоичных деревьев поиска. Если индекс требуемого элемента i меньше длины левого ребенка, то требуется рекурсивно вызвать поиск с ним. Если i больше величины s равной сумме длины левого поддерева и длины строки, хранящейся в корне, то искомый символ в правом поддереве и требуется рекурсивно выполнить поиск там, уменьшив i на s. Иначе требуемый символ

содержится в строке самого корня и может быть получен вызовом к ней за `O(1)`.

Вставку и удаление удобнее всего представить через разделение и конкатенацию. При удалении какой-либо части конкатенируется некоторый префикс и некоторый суффикс аргумента. При вставке строка разделяется в нужном месте, левый результат разделения конкатенируется с деревом, представляющим вставляемую строку, а результат конкатенации конкатенируется с правым результатом разделения. Можно таким же образом реализовать замену символа на символ, но для этой операции существует более простое решение. Для замены сначала находится заменяемый символ алгоритмом, совпадающим с алгоритмом поиска. Затем создается копия вершины, в которой заменяется символ. В завершении алгоритма производится копирование всего пути от этой вершины до корня для получения хронологичности.

2.2.2. Реализация на основе 2-3-деревьев

Реализация связок, основанная на 2-3 деревьях, может потребоваться в условиях, когда необходима гарантированная оценка времени выполнения, в так называемых системах реального времени, поскольку декартовые деревья не обладают логарифмической оценкой в худшем случае.

Откажемся от ключей, представляющих индексы символов в явном виде. Однако сделано это будет не так, как для декартовых деревьев. Символы будут располагаться только в листьях дерева. Внутренние же вершины имеют смысл конкатенации двух или трех детей. Порядок символов строки, представляемой деревом, определяется отношением предшествования в порядке обхода дерева по правилу левый-корень-правый. Пример такого представления строки «СПбГУИТМО» изображен на рисунке 6.

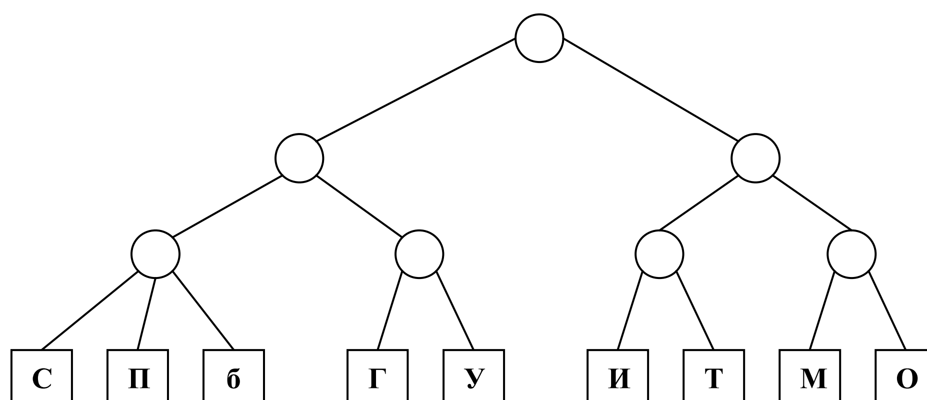

Рис. 6. Представление строки «СПбГУИТМО» на декартовом дереве

Процесс слияния двух строк (конкатенация) производится снизу вверх. Предположим, что дерево, представляющее первую строку, выше дерева второй строки. Случай, когда высоты соотносятся в другом порядке, обрабатывается симметрично. Сначала находится самая правая внутренняя вершина, находящаяся на высоте корня второго дерева. Она является потенциальным «братом» корня второй строки – требуется прикрепить добавляемый корень к ее родительской вершине. Дальнейшие действия зависят от количества детей у родительской вершины. Возможные варианты изображены на рис. 7. Когда родительская вершина A имеет двух детей B и C (рис. 7.а), корень добавляемого дерева D делается ребенком вершины A, у которой после этого становится три ребенка, а процесс на этом останавливается. Если родительская вершина A имеет три ребенка B, C и D (рис 7.б), то она раздваивается на A_1 и A_2. Вершина A_1 наследует от A принадлежность к дереву, однако из детей у нее остаются только B и C. Детьми вершины A_2 становятся вершины D и E. После этого требуется соединить вершины A_1 и A_2, что делается аналогично только на уровень выше. Процесс останавливается, когда достигает корня первого дерева.

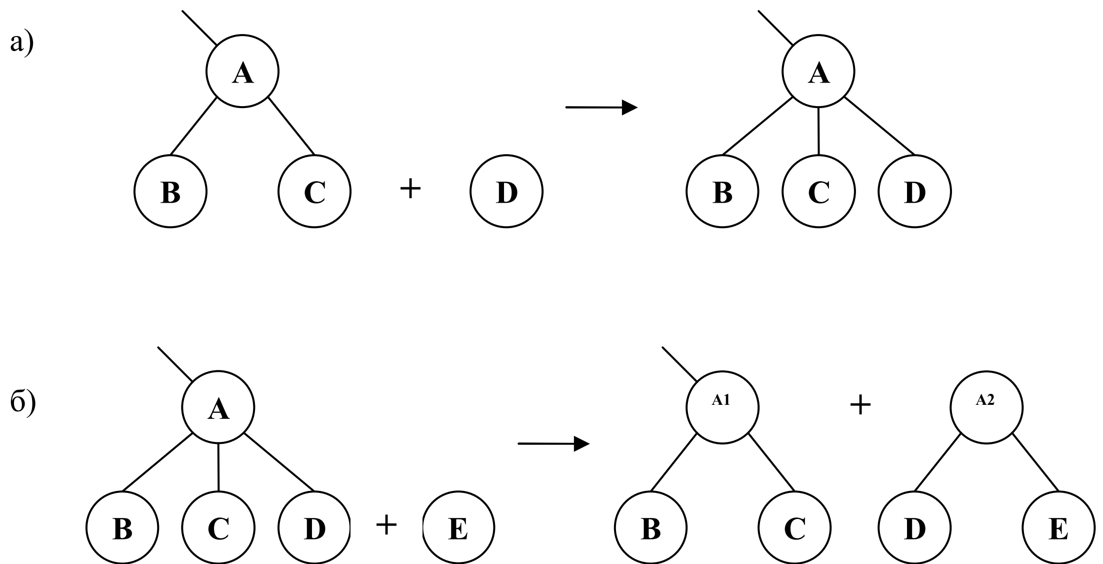

Рис. 7. Разделение связки на 2-3 дереве: а) 2 ребенка; б) 3 ребенка

Процедура разделения дерева по позиции в дереве начинается с поиска требуемой позиции. Поиск выполняется, как обычно, спуском от корня к нужному листу путем приема решений о направлении движения в узлах на основе информации о длине строк поддеревьев. Когда место разделения найдено, перерубаются все ребра, пересекающие границу раздела. Если в результате удаления ребер у каких-то внутренних вершин осталось только по одному ребенку, такие вершины удаляются вместе с ведущими в них ребрами. Таким образом, дерево рассыплется две последовательности деревьев с одной и с другой стороны от раздела. Каждую из последовательностей необходимо собрать в единое дерево при помощи операции конкатенации. Результатом операции разделения станут получившиеся два дерева. Более подробно с процессом разделения хронологических 2-3 деревьев, послужившим основой для приведенного алгоритма, можно ознакомиться в работе [24].

Глава 3. Улучшенный метод представления строк

В этой главе будет предложен улучшенный вариант представления строк с помощью деревьев. Основу метода составляют принципы, изложенные во второй главе. Однако ряд усовершенствований позволил дополнительно достичь асимптотического ускорения ряда строковых операций.

3.1. Сведение операций к базовым операциям

Множество распространенных операций со строками является довольно широким. Не имеет смысла пытаться отдельно описывать алгоритм каждой из операций для предлагаемого метода. Вместо этого может быть выделено и описано некоторое подмножество операций, через которые выражаются все остальные. Назовем такие операции базовыми. Остальные операции, рассматриваемые в данной работе, будут сведены к выполнению базовых операций.

3.1.1. Набор базовых операций

К базовым операциям относятся следующие четыре операции:

- разделение строки на две по определенной позиции;
- конкатенацию двух строк;
- добавление символа в один из концов строки;
- удаление символа с одного из концов строки.

3.1.2. Оценка времени через базовые операции

Длина строки во всех вариантах реализации может храниться отдельно и поддерживаться в актуальном состоянии с константной дополнительной нагрузкой на остальные операции. Поэтому в случаях, когда методом хранения эффективный способ получения длины строки не предусмотрен, эта операция все равно может быть оценена `O(1)`.

Расширение алгоритма коснется повышения эффективности операций на концах строки. В связи с этим требуются отдельные оценки для различных

частей строки. Операцией на конце строки назовем операцию, затрагивающую первый, либо последний символ строки. Все остальные случаи будем называть операциями с серединой.

Допустим, удаление символа в начале строки выполняется за $O(del(n))$. Процесс итерирования может быть выражен через последовательные удаления первого символа строки. Первый символ удаляется из строки и возвращается в качестве результата операции, а оставшаяся часть строки сохраняется для последующего итерирования. Каждый отдельный переход к следующему символу осуществляется за $O(del(n))$, а проход по всей строке занимает $O(n * del(n))$ времени.

Операция получения подстроки в середине представляется двумя разделениями. Первым разделением от строки отделяется лишний суффикс. Вторым – префикс. Если разделение выполняется за $O(split(n))$, то получение подстроки можно оценить $O(2 * split(n))$, что эквивалентно $O(split(n))$. С концов строки подстрока – это суффикс или префикс. Во-первых, для их получения достаточно только одного разделения. Во-вторых, если требуется подстрока достаточно небольшого размера, то может оказаться выгоднее с точки зрения использования времени получить подстроку путем итерирования. Итерированием по строке получаются требуемая последовательность символов, из которой последовательным добавлением конструируется строка-результат. Таким образом, получение подстроки с концов можно оценить $O(\min(split(n), m * (del(n) + add(n))))$, где $add(n)$ функция в O-оценке времени выполнения добавления символа в конец строки, а min – функция, возвращающая минимум из своих аргументов.

Вставка символов в концы является базовой операцией. Для вставки символов в середину строка сначала разрезается в месте вставки. Далее вставляемый символ добавляется в конец левой части разрезанной строки. Получившаяся в результате добавления строка конкатенируется с правой частью, образуя результат операции. Вся последовательность действий

оценивается $O(split(n) + add(n) + merge(n))$, если $O(merge(n))$ – оценка времени конкатенации.

Для удаления символа в середине может применяться следующая последовательность действий. Строка разделяется по позиции, предшествующей удаляемому символу. В правом результате разделения первый символ – это символ, который требуется удалить. Поскольку он находится в начале строки, то может быть удален операцией удаления с концов за $O(del(n))$. Далее левая часть строки конкатенируется с правой частью без первого символа. В результате получается исходная строка без того символа, который требовалось удалить. Операция требует $O(split(n) + del(n) + merge(n))$ времени.

Замена символа сводится даже не к базовым операциям, а уже рассмотренным удалению символа и вставке символа. Если замена требуется на концах строки, то оценка получается $O(del(n) + add(n))$. Замена в середине оценивается $O(split(n) + del(n) + merge(n) + split(n) + add(n) + merge(n))$, а эта оценка эквивалентна $O(split(n) + del(n) + add(n) + merge(n))$.

Рассмотрим ситуацию, когда вставляется не отдельный символ, а целая строка. Если строка вставляется с концов, то это конкатенация. Однако для небольшой строки может оказаться более эффективным добавить каждый ее символ поддельности. Тогда оценка вставки строки с концов принимает вид $O(\min(merge(n), m * (del(m) + add(n))))$. Когда вставка производится в середину, дополнительно требуется сначала разделить строку в месте вставки, а затем произвести конкатенацию. Оценка в таком случае принимает вид $O(split(n) + \min(merge(n), m * (del(m) + add(n))) + merge(n))$.

Для выполнения индексации разделим строку по позиции, заданной индексом. После разделения искомый символ стал последним символом строки и может быть получен за $O(del(n))$ операцией удаления. Всего оценка индексации получается $O(split(n) + del(n))$. На практике индексацию можно

выполнить более эффективно, но на асимптотической оценке это не сказывается.

3.2. Амортизированная реализация

Изложение усовершенствованного метода начнем с амортизированной версии алгоритма. В основе этой версии метода лежит метод со строками на декартовых деревьях. Поскольку оценки декартовых деревьев верны только в среднем, а в худшем случае оценки медленнее, то метод, построенный на них, все равно неприменим для системы реального времени. Следовательно, требовать от улучшенного алгоритма оценок в худшем случае не имеет смысла. В ряде случаев построить алгоритм, обладающий амортизированной оценкой, проще, чем алгоритм реального времени. Предлагаемые улучшения – это один из таких случаев.

3.2.1. Описание

Идея улучшения состоит в добавлении двух символьных буферов в начало и в конец строки. Середина строки хранится в виде сбалансированной связки на декартовых деревьях. В качестве символьных буферов используются хронологические стеки, рассмотренные в первой главе. Порядок символов строки задается структурой следующим образом. Сначала следуют символы буфера начала строки в порядке от головы стека к его дну. Далее в середине строки порядок задается порядком в связке. Завершают последовательность символы буфера конца строки в порядке от дна стека к его голове. Пример представления строки «СПбГУИТМО» с использованием буферов приведен на рис. 8.

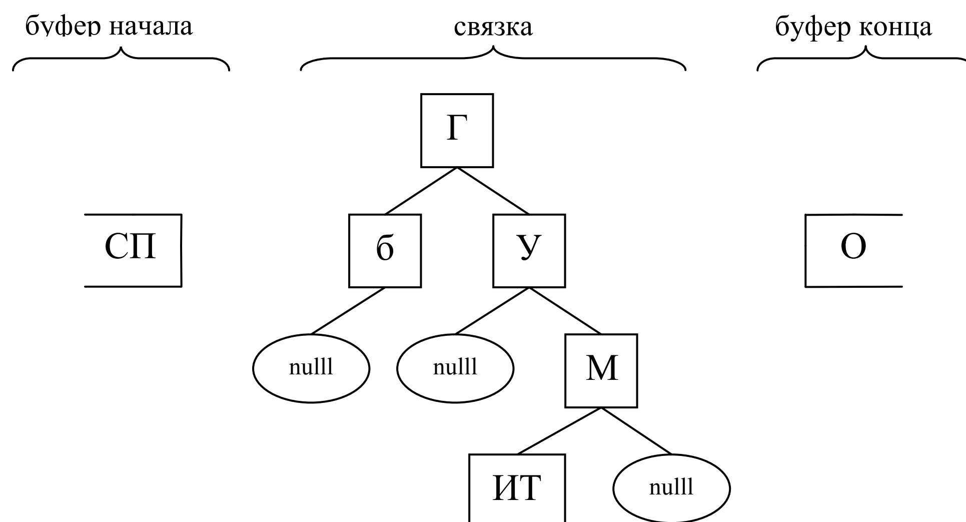

Рис. 8. Представление строки «СПбГУИТМО» в амортизированной реализации

Все строковые операции можно было бы выполнять напрямую со связкой, но наличие символов в буферах может этому помешать. Например, если берется подстрока, часть символов которой находятся в связке, а другая часть в одном из буферов. Однако буферы можно опустошить, не изменяя при этом последовательность символов строки. Для этого из символов, содержащихся в буферах, строятся две неизменяемые строки, одна – для префикса, а другая – для суффикса. Каждая из строк оборачивается вершинами дерева. После этого их можно считать связками, состоящими из одной вершины. Эти связки конкатенируются с центральной связкой, образуя единую связку, содержащую все символы исходной строки в правильном порядке. В качестве новых буферов строки создаются пустые стеки. Процесс продемонстрирован на рис. 9.

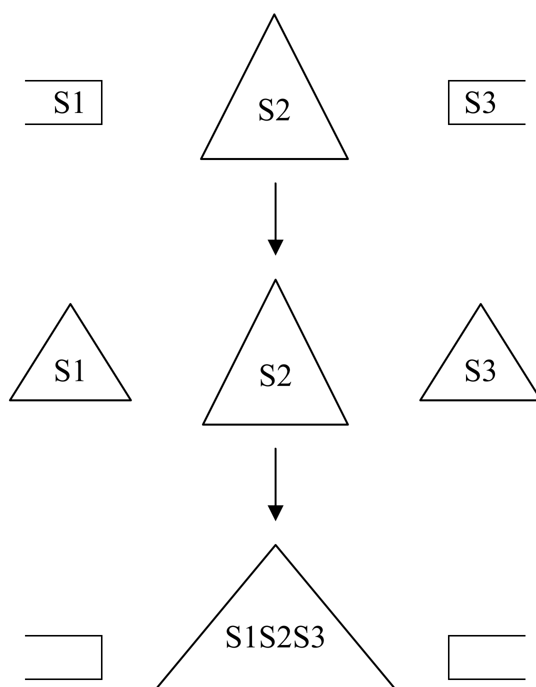

Рис. 9. Опустошение буферов

Перед конкатенацией двух строк первая освобождается от правого буфера, а вторая – от левого. Затем их центральные связки конкатенируются. В качестве буферов строки-результата берутся левый буфер первой строки и правый буфер второй.

Для разделения производится опустошение обоих буферов строки. Затем разделяется центральная связка. Результатам разделения справа и слева присоединяются новые пустые буферы.

Буферы используются для ускорения операций добавления и удаления символов. При добавлении в один из концов символ помещается в соответствующий буфер. Остальная структура при этом остается неизменной. Удаление символов тоже производится из буферов. Проблема заключается в том, что буфер может опустеть, тогда становится невозможно удалить из него символ. В этом случае сначала производится очистка буферов так, чтобы второй буфер также опустел. Далее проводится операция противоположная опустошению буферов – их наполнение за счет центральной связки. От центральной связки отделяется связка-префикс и связка-суффикс длины $\log n$, где n – длина строки. Итерированием по этим

связкам получаются их последовательности символов, которые складываются во вновь созданные буферы. Следует заметить, что для получения корректного левого буфера итерирование по префиксу должно производиться в обратном порядке от конца к началу. Процесс наполнения связок проиллюстрирован на рис. 10. В результате получается строка с непустыми буферами, из которых может быть произведено удаление символа.

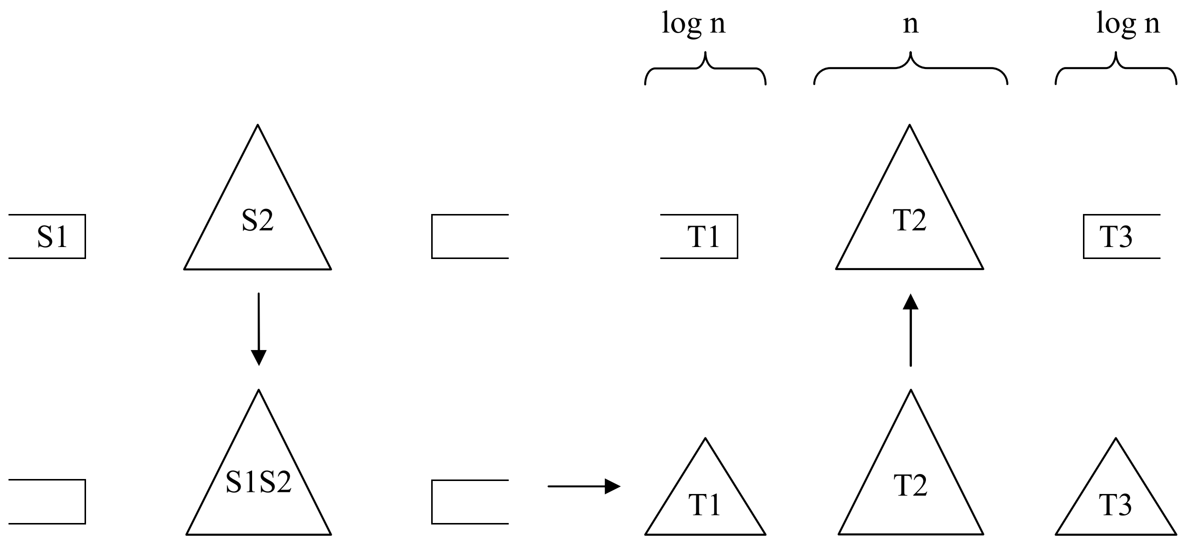

Рис. 10. Наполнение буферов

3.2.2. Оценка времени базовых операций

Для оценки временной сложности конкатенации сначала предположим, что размер буферов никогда не превышает величины порядка логарифма количества вершин в строке. Этого можно добиться интеграцией в центральную связку половины буфера каждый раз, когда его размер достигает двух логарифмов длины строки. Оценим сложность очистки буферов. Новая связка по содержимому буфера строится за время пропорциональное длине, а в предположении о логарифмическом размере буфера, эта величина составляет $O(\log n)$. Далее полученные связки конкатенируются с центральной связкой согласно оценкам из второй главы также за $O(\log n)$. Таким образом, вся очистка требует $O(\log n)$ времени. Возвращаясь к конкатенации, делаем вывод, что ее сложность

складывается из сложности очистки и сложности конкатенации двух центральных связок, которая в свою очередь тоже логарифмическая. Суммарно получается $O(\log n)$. Распространить эту оценку на случай, когда размер буфера может превышать логарифм, позволяют амортизированные рассуждения. Каждый раз, когда производится добавление символа в буфер, будем резервировать константное время для ближайшей очистки буфера. Когда потребуется очистить буфер, содержащий m символов, в резерве будет накоплено времени порядка $O(m)$. Этот резерв используем для построения связки из буфера. Таким образом, в оценке очистки остается только конкатенация, которая выполняется за $O(\log n)$.

Описанная амортизированная оценка очистки буфера перед конкатенацией не является специфичной только для конкатенации. Тот же созданный резерв времени может быть использован при очистке буфера для операции разделения. Следовательно, очистка буфера при разделении оценивается $O(\log n)$, столько же требует разделение центральной связки. Суммарная оценка операции разделения получается $O(\log n)$.

Добавление символа в один из концов строки выполняется за чистое неамортизированное $O(1)$, поскольку состоит только из добавления одного нового элемента в хронологический стек.

При оценке стоимости удаления символа на концах строки придется опять воспользоваться амортизированным подходом. Большинство удалений являются только извлечениями верхнего элемента из хронологического стека и выполняются за $O(1)$. Однако попытка удаления символа из пустого стека приводит к необходимости сначала наполнить буфер, что делается за $O(\log n)$. Возьмем этот логарифм времени в долг, и будем считать, что удаление произошло за $O(1)$. Возврат долга будет осуществляться резервированием константного времени при каждом последующем удалении из вновь созданного буфера. Для того чтобы удалить все символы в этом буфере,

потребуется не менее $O(\log n)$ удалений, которые как раз и обеспечат резерв $O(\log n)$ времени, покрывающий позаимствованное время.

3.2.3. Оценка количества памяти

Всего в представлении строки задействуется три структуры: два хронологических стека и одна сбалансированная связка на декартовых деревьях. Суммарный размер этих структур пропорционален длине строки. Поэтому если оценивать количество памяти, занимаемое строкой, без учета памяти, используемой совместно с другими строками, получается $O(n)$, где n – длина строки.

Однако строки, получаемые в результате операций, совместно используют часть своего содержимого со строками, являвшимися аргументами операции. Добавление символа в конец строки вызывает создание ровно одного нового элемента стека, это требует всего $O(1)$ памяти.

При удалении символа с конца строки память скорее освобождается, чем требуется, однако $O(1)$ памяти все же будет задействовано для представления новой версии строки, поскольку она состоит из ссылок на новые версии входящих в нее структур. В худшем случае удаление может потребовать даже выделения порядка логарифма новой памяти. Это происходит, когда из-за пустого буфера приходится отрезать часть центральной связки.

Разделение и конкатенация требуют порядка логарифма памяти на создание связок, представляющих буферы, порядка логарифма памяти для выполнения конкатенации этих связок с центральной связкой, и еще порядка логарифма на проведение соответственно разделения или конкатенации центральной связки. Всего эти операции задействуют $O(\log n)$ памяти.

3.3. Реализация реального времени

Дальнейшее развитие предлагаемого метода представления строковых данных привело к разработке структуры, обладающей гарантированными оценками. Однако получившаяся в результате структура данных настолько сложна, что автор пока не отважился ее реализовать на практике, поэтому пока структура имеет чисто теоретический характер. Опишем, каким образом можно адаптировать амортизированную версию к условиям реального времени.

3.3.1. Описание

Использование сбалансированных связок на декартовых деревьях для построения строк реального времени невозможно по причине отсутствия гарантированных оценок. Вместо связок на декартовых деревьях более уместно использование связок на 2-3 деревьях. Связки на 2-3 деревьях сложнее устроены и непросты в реализации, но они обладают необходимыми гарантированными оценками времени выполнения операций.

Как и в случае амортизированного метода связки занимают центральную позицию в представлении строки. Большая часть символов хранится именно в связке. Только некоторые префикс и суффикс находятся в соответствующих буферах начала и конца строки. Буферы в отличие от буферов амортизированной версии состоят не только из хронологического стека. Буфер разделен на две части. Часть, содержащая символы, расположенные ближе к соответствующему концу строки, хранится в хронологическом стеке. А та часть, которая расположена ближе к центру строки, представлена связкой. Схема устройства такой строки изображена на рисунке 11. Порядок символов строки задается следующей последовательностью: сначала символы стека B_1, затем символы связки S_1, связки S, связки S_2, в конце перевернутая (от дна к голове) последовательность символов стека B_2.

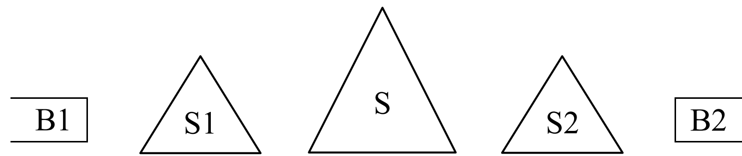

Рис. 11. Схема строки в версии реального времени

Сначала рассмотрим операции добавления и удаления символов на концах строки. В зависимости от количества символов в буферах структура всегда находится в одном из пяти состояний (рис. 12). Нормальным считается состояние строки (рис. 12.а), когда размеры стеков B_1 и B_2 лежат в пределах от `log(n)` до `2 * log(n)`, размеры связок S_1 и S_2 равны `log(n)`, а все остальные из n символов строки находятся в связке S. В этом состоянии вставка и удаление символов на одном из концов строки производится путем вставки и удаления элементов в соответствующем стеке.

Когда в результате вставок и удалений ограничения нормального состояния на размеры составных частей строки перестают выполняться, строка переходит в одно из остальных состояний. В этих состояниях символы по-прежнему вставляются и удаляются из стеков, но при каждой вставке или удалении дополнительно производится фиксированное число действий, направленных на возвращение в нормальное состояние. Если размер стека B_1 стал больше `2 * log(n)`, строка переходит в состояние очистки левого буфера. Задача этого состояния состоит в переносе части символов из центральной связки S в левый буфер. Возможность совершения дополнительных действий используется для следующей последовательности операций. Связки S_1 и S конкатенируются в одну связку S`, которая после завершения всех трансформаций займет место центральной связки $S$. Из `log(n)` последних символов стека B_1 формируется связка S_3, которая займет место связки левого буфера S_1. После окончания создания связки S_3 из буфера удаляются использованные при этом символы. Последним этапом перехода строки обратно в нормальное состояние является замена S_1 на S_3, а S

на $S`$. Учитывая симметрию, аналогичным образом обрабатывается состояние очистки правого буфера (рис. 12.в).

На рис. 12.г изображена схема состояния наполнения левого буфера. В это состояние строка переходит, когда размер левого стека становится меньше `log(n)`. Дополнительные действия в этом случае направлены на создание нового стека из связки левого буфера. Итерированием по связке S_1 формируется стек B_3. Для того чтобы B_3 содержал правильную последовательность символов, итерирование производится в обратном порядке (от конца в начало). Далее связка S делится на две S_3 и $S`$ так, чтобы в $S_3$ попало `log(n)` символов, а в $S`$ остались все остальные символы связки S. После завершения этих действий строка готова переходу в нормальное состояние. Однако переход осуществляется не сразу, а только после опустошения стека B_1. Если опустошения таки и не происходит, а размер стека возвращается в ограничения, устанавливаемые нормальным состоянием, то все структурные изменения отменяются, а строка просто переходит в нормальное состояние. Если стек все-таки становится пустым, возврат в нормальное состояние сопровождается заменой стека левого буфера на B_3, связки левого буфера на S_3, а центральной связки на $S`$. Аналогично обрабатывается состояние наполнения правого буфера (рис. 12.д).

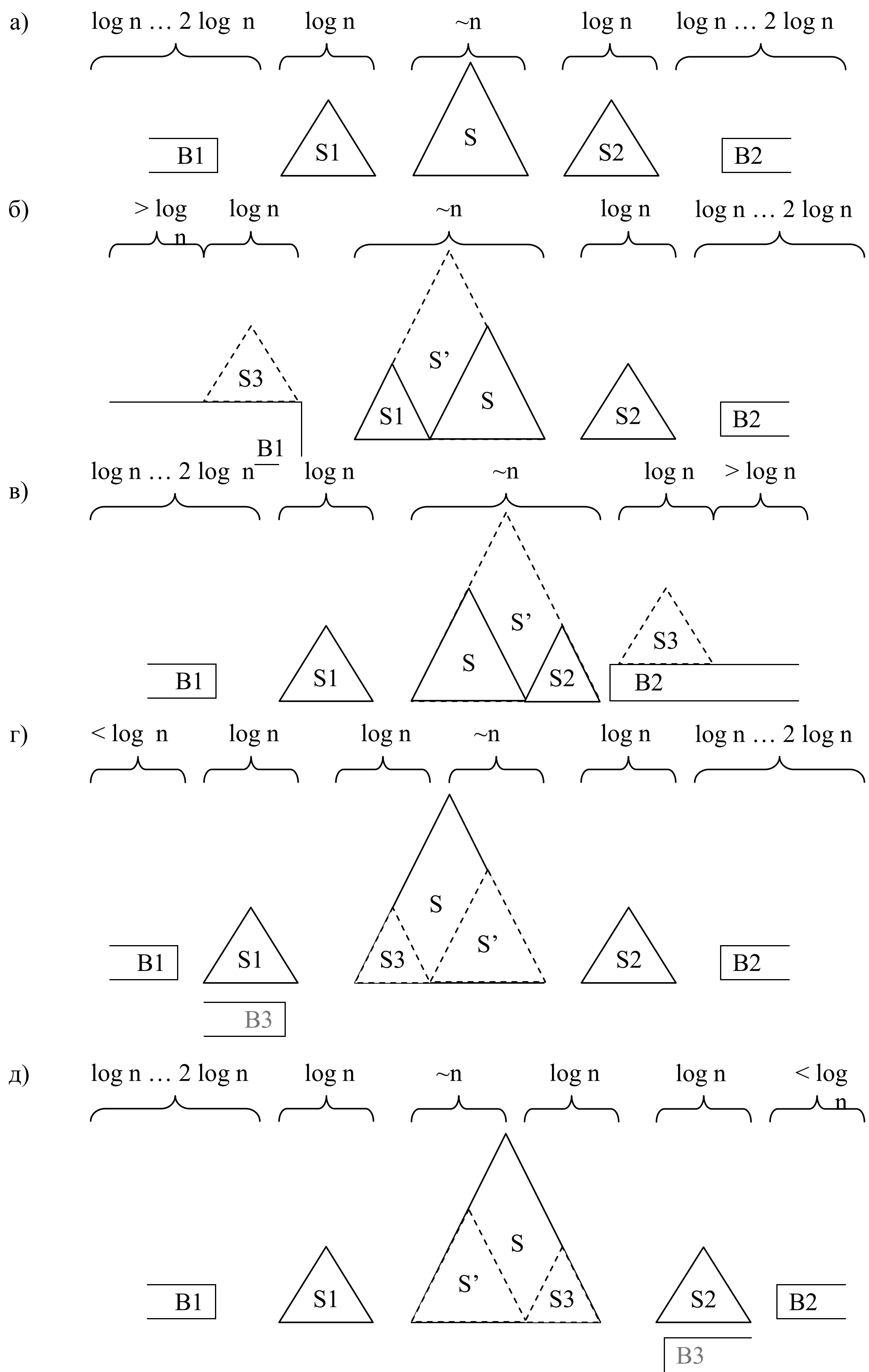

Рис. 12. Состояния строки: а) нормальное; б) очистка левого буфера; в) очистка правого буфера; г) наполнение левого буфера; д) наполнение правого буфера

Для совершения со строкой операций разделения и конкатенации сначала оба буфера интегрируются в центральную связку. Связки буферов просто конкатенируются с центральной связкой. Из стеков буферов формируются новые связки, которые в свою очередь конкатенируются с результатом конкатенации центральной и буферных связок. После интеграции все символы строки содержатся в центральной связке, с которой могут быть выполнены операции разделения или конкатенации. После завершения операций для возвращения строки в нормальное состояние производится дезинтеграция. От центральной связки отрезаются с каждой стороны по $\log(n)$ символов для буферных стеков и $\log(n)$ символов для буферных связок. Отрезанные символы представляют собой связки. Из символов, отрезанных для буферных стеков, требуется сформировать стеки путем обхода связок. Заметим, что для формирования правильной последовательности в стеке правого буфера, соответствующую связку следует обходить в обратном направлении.

3.3.2. Оценка времени базовых операций

Все дополнительные действия, проводимые параллельно добавлениям и удалениям символов вне нормального состояния, требуют $O(\log n)$ времени. Допустим, константа в этой оценке равна некоторому числу C. Тогда, если при каждом добавлении или удалении выполнять дополнительные действия в течение времени $3C$, размеры стеков не успеют опустеть или наоборот заполниться настолько, чтобы сделать невозможным возвращение в нормальное состояние после завершения предусмотренных состоянием модификаций. Следовательно, добавления и удаления символов выполняются за гарантированное время $O(1)$, при этом поддерживая в одном из пяти предусмотренных состояний структуру строки. Другим следствием является то, что размер буферных стеков никогда, даже в состоянии очистки буферов, не превысит ограничения в $8 * \log(n) / 3$.

Поскольку размер буферов в любом состоянии можно оценить $O(\log n)$, создание связок из стеков и построение стеков из связок требует времени $O(\log n)$. Помимо трансформации связок и стеков операции интеграции и дезинтеграции состоят из фиксированного количества операций конкатенации и разделения, каждая из которых требует $O(\log n)$ времени. Следовательно, интеграция и дезинтеграция состоят из константного количества логарифмических операций, и их время выполнения можно оценить $O(\log n)$. Это ведет к логарифмической оценке операций разделения и конкатенации строки. Действительно, конкатенация и разделение отличаются от соответствующих операций над центральной связкой только предварительной дезинтеграцией и последующей интеграцией.

3.3.3. Оценка количества памяти

Входящие в состав строки структуры имеют линейную оценку размера требующейся памяти от количества символов, хранящихся в них. Поскольку все символы строки разделены между этими структурами на непересекающиеся части, то всего памяти для их хранения требуется $O(n)$, где n – длина строки. Эта оценка проведена без учета совместно используемой между различными строками памяти.

Если же учитывать, что строки, получающиеся в результате операций, разделяют большую часть своего содержимого со строками, являющимися аргументами этих операций, то оценка может быть меньше. Разбирать каждую операцию с целью выяснения количества памяти, требующейся для выполнения, довольно непростое занятие, а главное это вовсе необязательно. Можно просто вспомнить, что за время A, невозможно использовать больше $O(A)$ памяти. Учитывая оценки времени выполнения операций, получаем следующие оценки. Удаление и вставка символа используют $O(1)$ памяти. Разделение и конкатенация – $O(\log n)$ памяти.

3.4. Сравнение реализаций представления строковых данных

Сравним оценки времени выполнения операций для существующих и предложенных в работе методов представления строковых данных. При подстановке оценок времени выполнения базовых операций в оценки остальных операций получаются окончательные оценки всех операций. После получения всех оценок, предложенные методы могут быть включены в сравнительную таблицу оценок времени для различных методов.

3.4.1. Окончательные оценки времени всех операций

Оба предложенных метода представления строковой информации помимо логарифмических оценок таких операций, как индексация, конкатенация, получение подстроки, удаление, вставка и замена в середине строки, обладает константной оценкой операций добавления и удаления символов на концах строки.

Если абстрагироваться от понимания символа, как части текстовой информации, а представить, что символом может являться любой объект, то можно рассматривать предложенную организацию хранения строки, как структуру данных. Учитывая то, что операции вставки и удаления на обоих концах выполняются за $O(1)$, получается, что эта структура выполняет функции дека. Кроме того, она поддерживает конкатенацию и является объединяемо хронологическим.

Наиболее близкой структурой данных, сочетающей в себе все эти свойства, является полностью функциональный дек реального времени с поддержкой конкатенации, предложенный в работе Каплана и Тарьяна [2]. Их дек также является объединяемо хронологическим. Все операции выполняются за гарантированное время. Также помимо операций дека поддерживается конкатенация. Различие заключается в том, что дек, предложенный в данной работе, конкатенируется за $O(\log n)$, а дек

Каплана и Тарьяна – за $O(1)$. Преимуществом же дека из данной работы является то, что он поддерживает индексацию и разделение за $O(\log n)$, а в работе Каплана и Тарьяна этих операций не предусмотрено.

3.4.2. Сравнительная таблица

Теперь после получения всех оценок может быть проведено сравнение различных методов представления строковой информации. Оценки собраны в таблицу (табл. 2). Темно-серым цветом выделены ячейки, соответствующие оптимальным оценкам. Оптимальными считаются те оценки, которые нельзя асимптотически улучшить путем выбора другого метода. Если все методы по какой-либо операции обладают оптимальными оценками, то в таблице цветом они не выделяются. Также в таблице отмечены оценки, отличающиеся от оптимальных оценок не более чем в $\log(n)$ раз. Они выделены светло-серым цветом. Выделяются такие оценки потому, что отличие в $\log(n)$ раз по сравнению с отличием в n раз не является очень существенным, можно считать такие оценки близкими к оптимальным оценкам. Для сохранения обозримости в таблице из методов, приведенных в табл. 1, оставлены только неизменяемые строки и строки на циклических очередях. Для деков Каплана и Тарьяна некоторые операции не поддерживаются, оценка для таких операций недоступна. При обозначении части строки концы и середины сокращены «кон.» и «сер.» соответственно для экономии ширины таблицы.

Таблица 2. Сравнительная таблица методов представления строк

Операция	Часть строки	Неизменяемые строки	Строки на циклических очередях	Связки на декартовых деревьях	Связки на 2-3 деревьях	Деки Каплана и Тарьяна
Получение длины		O(1)	O(1)	O(1)	O(1)	O(1)
Получение подстроки длины m	кон.	O(1)	O(m)	O(min(m, log n))*	O(min(m, log n))	O(m)
	сер.	O(1)	O(m)	O(log n)*	O(log n)	н/д
Конкатенация со строкой длины m		O(n + m)	O(m)	O(min(m, log n))*	O(min(m, log n))	O(1)
Итерирование		O(1)	O(1)	O(1)*	O(1)	O(1)
Индексация	кон.	O(1)	O(1)	O(1)	O(1)	O(1)
	сер.	O(1)	O(1)	O(log n)*	O(log n)	н/д
Вставка символа	кон.	O(n)	O(1)	O(1)	O(1)	O(1)
	сер.	O(n)	O(n)	O(log n)*	O(log n)	н/д
Удаление символа	кон.	O(1)	O(1)	O(1)*	O(1)	O(1)
	сер.	O(n)	O(n)	O(log n)*	O(log n)	н/д
Замена символа на символ	кон.	O(n)	O(1)	O(1)*	O(1)	O(1)
	сер.	O(n)	O(1)	O(log n)*	O(log n)	н/д
Вставка строки длины m	кон.	O(n + m)	O(m)	O(min(m, log n))*	O(min(m, log n))	O(1)
	сер.	O(n + m)	O(n + m)	O(log n)*	O(log n)	н/д

* Звездочкой отмечены оценки не пригодные для условий реального времени. Такие оценки являются либо амортизированными, либо ожидаемыми оценками в среднем.

Глава 4. Практическое внедрение

В ходе работы была выполнена практическая реализация одного из предлагаемых методов и проведена экспериментальная проверка его эффективности. В этой главе описан проведенный эксперимент и приведены его результаты. В конце главы подведены итоги экспериментального исследования и приведены рекомендации по применению разработанного метода, учитывающие результаты экспериментальной проверки.

4.1. Замена строк на Java

Суть эксперимента состоит в сравнении времени требующегося для работы стандартных строковых классов языка Java и строковых классов, реализующих предлагаемый в работе метод. Строки языка Java в стандартной реализации являются неизменяемыми. Основой их внутренней структуры является массив. В проверяемом методе основой структуры строки является сбалансированная связка.

4.1.1. Идея замены

Для проверки практической применимости разработанных в работе алгоритмов было принято решение реализовать на языке Java строковые классы, полностью повторяющие по своей сигнатуре соответствующие классы стандартной библиотеки, но реализованные с использованием разработанного метода.

Предполагалось, что это позволит подменить стандартные классы новыми классами. На практике подмена вызвала сбой виртуальной машины. Исследование причин сбоя привело к выводу, что стандартная реализация представления строковых данных глубоко интегрирована в среду исполнения, ее внутренняя структура не может быть изменена без существенной модификации среды исполнения.

Тогда было принято решение изменить стандартную реализацию строковых классов так, чтобы велась история всех вызовов строковых

методов вместе с аргументами. Запись истории осуществляется в файл. Это позволило по полученным историям полностью повторить набор вызовов, вызывая при этом методы классов в модифицированной реализации строк и сравнить производительность со стандартной реализацией.

4.1.2. Реализация строк

Для замены класса *String* были выбраны связки на декартовых деревьях. Все методы класса были переписаны в соответствии с изменением метода представления строковых данных. Сигнатура класса была сохранена.

4.1.3. Реализация строковых буферов

Для ситуаций, когда строка строится последовательным добавлением отдельных символов или строк, в стандартной библиотеке Java предусмотрен класс *StringBuilder*. *StringBuilder* является изменяемым строковым буфером. Время добавления в него пропорционально длине добавляемой части. Внутренней структурой для хранения символов в *StringBuilder* является массив.

При модификации внутренняя структура была изменена на модифицированную версию класса *String*. Добавление при этом было реализовано как конкатенация со строкой внутреннего представления.

4.2. Оценка результатов замены

Для сравнения производительности стандартной и альтернативной реализации строковых классов была произведена запись истории вызовов строковых методов, происходивших при работе реальных приложений, написанных на языке Java. При сравнении для удобства замера времени каждая запись истории вызовов повторялась несколько раз. Для тестирования были выбраны два известных приложения.

4.2.1. Приложение *ant*

Приложение *ant* – это система автоматической сборки, написанная на языке Java. Система является приложением с открытыми исходными кодами. Больше всего ant подходит для сборки проектов на Java. Правила сборки задаются в специальных файлах. Приложение имеет консольный интерфейс.

Запись истории вызовов проводилась во время сборки небольшого тестового проекта. В табл. 3 приведены результаты запуска на полученной истории стандартных и альтернативных строк.

Таблица 3. Результаты тестирования на приложении `ant`

Класс и метод	Число вызовов (x10000)	Суммарное время для неизменяемых строк (мс)	Суммарное время для связок (мс)
`String.replace`	8462	13940	17191
`String.concat`	2201	4111	1545
`StringBuilder.append(String)`	9660	31881	14310
`String.compareTo`	183	124	205
`String.charAt`	481001	22188	76741
`StringBuilder.append(Character)`	4967	6878	4213
`String.equals`	159589	36818	51933
`StringBuilder.append(Integer)`	62	328	16
`String.substring`	18789	4792	45993

4.2.2. Приложение *Poseidon*

Приложение *Poseidon* – это программа, предназначенная для создания моделей с использованием унифицированного языка моделирования UML. Приложение написано на языке Java, имеет графический интерфейс. *Poseidon* распространяется на коммерческой основе, однако существует возможность использования пробной версии. Тестирование производилось на пробной версии.

Запись истории производилась во время старта приложения, поскольку начало работы чаще всего сопряжено с наиболее заметными для пользователя задержками. Оптимизация старта приложения может повысить удобство использования программного продукта. Результаты тестирования

на истории вызовов строковых методов при старте приложения *Poseidon* приведены в табл. 4.

Таблица 4. Результаты тестирования на приложении *Poseidon*

Класс и метод	Число вызовов	Суммарное время для неизменяемых строк (мс)	Суммарное время для связок (мс)
`String.replace`	3100	9209	11224
`String.concat`	1593	2936	1245
`StringBuilder.append(String)`	16929	64713	16666
`String.compareTo`	564	94	610
`String.charAt`	1744581	81430	284508
`StringBuilder.append(Character)`	69728	102312	53386
`String.equals`	30909	16970	7321
`StringBuilder.append(Integer)`	215	340	155
`String.substring`	22537	6037	60608

4.2.3. Сравнение результатов

Если просуммировать все время, затраченное каждой из реализаций на выполнение историй вызовов, то получается, что реализация предложенного в работе метода медленнее стандартной реализации на 37%. Суммарное время работы стандартного алгоритма равно 405101 мс, альтернативного – 647870 мс.

Структура распределения времени по различным операциям показывает, что наибольшее отставание возникает из-за операции индексации (метод `charAt` класса `String`). Это объяснимо, поскольку в стандартной реализации эта операция выполняется за *O(1)*, а в альтернативной за *O(log n)*. Количество вызовов этой операции свидетельствует о том, что она часто используется для итерирования по строке. Асимптотика же итерирования у обоих методов одинакова, поэтому можно надеяться на улучшение результатов, если код использующий строки будет учитывать различие между итерированием и индексацией.

Сильной стороной предложенного метода, как и предполагалось, является конкатенация. Логарифмическая оценка позволяет добиться преимущества в операциях связанных с конкатенацией. Если

просуммировать времена запуска без учета индексации, то результат получается в пользу альтернативной реализации строк (301483 мс против 286621 мс).

4.3. Рекомендации по применению

Получение выигрыша во времени предложенный метод позволяет достичь только при использовании, учитывающем различия между операциями итерирования и индексации. Какая-либо замена метода представления строковой информации на предложенный, когда код приложения уже написан, скорее всего, не приведет к ускорению работы. Для эффективного использования предложенного метода требуется разрабатывать приложения, используя наиболее подходящие операции. Однако продуманное использование стандартных структур, включая строковые буферы, также может привести к повышению эффективности.

Заключение

В работе выполнен обзор ряда существующих методов представления строк, а также структур данных, которые могут быть использованы для хранения строковых данных. На основании исследования предметной области поставлена задача создания метода представления строк, обеспечивающего не более чем в логарифм раз большую асимптотику времени выполнения каждой из строковых операций по сравнению с любым другим методом.

Решением поставленной задачи стали два метода хранения строк, основанные на хранении строк в виде деревьев и использовании алгоритмов балансировки аналогичных используемым для сбалансированных деревьев поиска. Первый метод базируется на алгоритмах рандомизированных двоичных деревьев поиска. Главное преимущество использования рандомизированных двоичных деревьев поиска состоит в простоте получившегося метода. Однако асимптотические оценки времен работы операций со строками, представленными этим методом, являются лишь ожидаемыми в среднем и амортизированными. Во втором методе за основу взяты 2-3 деревья. Этот метод несколько сложнее, но предоставляет гарантированные оценки времен работы и может применяться в условиях реального времени.

Дальнейшее совершенствование методов позволило добиться оценки $O(1)$ времени работы ряда строковых операций. К таким операциям относятся добавление и удаление символов на концах строки, а также все операции, которые через них выражаются.

Предложенные усовершенствования привели к получению новой структуры данных для представления хронологических деков. Полученные деки поддерживают операцию конкатенации и являются объединяемо хронологическими. Близкий результат Тарьяна и Каплана [2] – структура, которая обладает более высокой скоростью конкатенации, но в ней не

предусмотрены операции индексирования, получения подпоследовательности и ряд других операций.

Экспериментальная проверка эффективности разработанного метода показала, что его применение не дает выигрыша в скорости выполнения операций для существующих приложений. Причиной основных потерь производительности является то, что в существующих приложениях итерирование реализовано через индексацию. В разработанном же методе итерирование выполняется за O(1), а индексация – за O(log n). Эксперимент показал потенциальную возможность получения преимущества перед стандартными методами представления строк при условии использования индексации только там, где это необходимо.

Список литературы

1. *Seidel R. G., Aragon C. R.* Randomized Search Trees /Foundations of Computer Science, 30th Annual Symposium, 1989.
2. *Kaplan H., Tarjan R. E.* Purely functional, real-time deques with catenation //Journal of the ACM. 1999. 46(5).
3. *Driscoll J. R., Sarnak N., Sleator D. D., Tarjan R. E.* Making Data Structures Persistent // Journal of Computer and System Sciences. 1989. Vol. 38. № 1.
4. *Sarnak N., Tarjan R. E.* Planar point location using persistent search trees // Communications of the ACM. 1986. 29(7).
5. *Reps T., Teitelbaum T., Demers A.* Incremental context-dependent analysis for language-based editors /ACM TOPLAS 5, 1983.
6. *Krijnen T., Meertens L.* Making B-Trees Work for B. IW 219/83. The Mathematical Centre. Amsterdam, 1983.
7. *Lea D.* Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley, 2000.
8. *Романовский И. В.* Дискретный Анализ. СПб.: Невский Диалект, 2000.
9. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ, МЦНМО, 2002.
10. *Сысоев А. П.* Строковые типы в Delphi. Особенности реализации и использования // Sources.RU Magazine. 2004. № 1.
11. *Кнут Д. Э.* Искусство программирования. Том 1. Основные алгоритмы. М.: Вильямс, 2007.
12. *Bloch J.* Effective Java: programming language guide. Addison-Wesley, 2001.
13. *Hudak P.* Conception, evolution, and application of functional programming languages // CSUR. 1989. 21(3).
14. *Hutton G.* Programming in Haskell. Cambridge University Press, 2007.
15. *Okasaki C.* Purely functional random-access lists. Functional Programming Languages and Computer Architecture, 1995.
16. *Seidel R. G., Aragon C. R.* Randomized Search Trees // Algorithmica. 1996. 16.

17. *Aho A. V., Hopcroft J. E., Ullman J. D.* The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.
18. *Bayer R., McCreight E.* Organization and Maintenance of Large Ordered Indexes // Acta Informatica. 1972. 1.
19. *Fiat A., Kaplan H.* Making data structures confluently persistent / Symposium on Discrete Algorithms, 2001.
20. *Lampson B.* A Description of the Cedar Language. Xerox PARC Technical Report. CSL-83-15. 1983.
21. *Swinehart D., Zellweger P., Hagmann R.* The structure of Cedar / ACM SIGPLAN, 1985.
22. *Swinehart D., Zellweger P., Beach R., Hagmann R.* A structural view of the Cedar programming environment / TOPLAS. 1986. 8(4).
23. *Atkinson R., Boehm H. J., Plass M.* Ropes: an Alternative to Strings // Software Practice and Experience. 1995. 12.
24. *Bent S. W., Sleator D. D., Tarjan R. E.* Biased 2-3 trees / 21st Annual IEEE Symposium on Foundations of Computer Science, 1980.