**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра АПУ**

отчет

**по лабораторной работе № 8**

**по дисциплине «Программирование»**

**Тема: Бинарное дерева поиска**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3371 |  | Манешов Д. В. |
| Преподаватель |  | Писарев А.С. |

Санкт-Петербург

2024

## Постановка задачи.

Необходимо дополнить структуру данных Tree из задания 7 следующим функционалом:

* В перечисление Order добавить значения: “LevelsUpLeft”, “LevelsUpRight”, “LevelsDownLeft”, “LevelsDownRight”
* Расширить возможности функции ToArray добавив обработку новых значений Order
  + LevelsUpLeft - вывод по уровням сверху вниз, слева направо
  + LevelsUpRight - вывод по уровням сверху вниз, справа налево
  + LevelsDownLeft - вывод по уровням снизу вверх, слева направо
  + LevelsDownRight - вывод по уровням снизу вверх, справа налево

## Ход решения.

Данная программа реализует структуру данных бинарного дерева поиска (БДП) для работы с целыми числами. В структуре Tree предусмотрен ряд основных операций: добавление, удаление и проверка наличия значения в дереве, очистка дерева, подсчет числа элементов, а также балансировка дерева и преобразование его в массив. Программа поддерживает различные порядки обхода дерева, включая префиксный, инфиксный, постфиксный, обход уровней слева направо и справа налево, как снизу вверх, так и сверху вниз. Также реализованы методы поворотов узлов дерева влево и вправо, что позволяет эффективно поддерживать его балансировку.

Этот фрагмент кода (рис. 1.1) включает библиотеку <iostream>, которая предоставляет функциональность для ввода и вывода данных в C++. Благодаря этой библиотеке доступны такие объекты, как std::cin и std::cout, которые позволяют читать входные данные с консоли и выводить данные на экран. Также подключается заголовочный файл "List.h", который содержит описание и реализацию шаблонного класса List, предназначенного для создания и управления двусвязным списком.

Использование выражения using namespace std; упрощает доступ к элементам из пространства имен std, избавляя от необходимости указывать префикс std:: перед каждым элементом, таким как cout и cin.



Рисунок 1.1 – Подключение библиотек и пространства имен для работы с вводом-выводом и списками

В данном фрагменте кода (рис. 1.2) определяется структура Node, которая используется для представления узла в бинарном дереве поиска. Каждый узел содержит целочисленное значение data, а также два указателя: left и right, которые указывают на левое и правое поддеревья соответственно. Конструктор Node принимает значение value и присваивает его полю data. Указатели left и right изначально устанавливаются в nullptr, что означает отсутствие дочерних узлов.

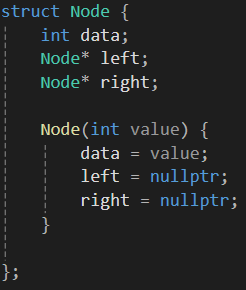


Рисунок 1.2 – Определение структуры Node

Данный фрагмент кода (рис. 1.3) представляет начало объявления класса Tree, который реализует структуру данных бинарного дерева поиска. В приватной области класса определены два члена: целочисленная переменная size и указатель root. Переменная size отслеживает общее количество узлов, присутствующих в дереве, что помогает быстро получать информацию о размере дерева. Указатель root указывает на корневой узел дерева, с которого начинаются все операции добавления, удаления и поиска. Изначально, при создании нового экземпляра класса, корень будет установлен в значение nullptr, что сигнализирует о пустом дереве.

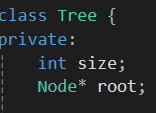


Рисунок 1.3 – Объявление класса Tree

Метод levelsUpLeftTraversal (рис. 1.4) осуществляет обход дерева в ширину, начиная с заданного узла и сохраняя значения узлов в массив. Метод принимает два параметра: указатель на корневой узел дерева node и указатель на массив array, в который будут сохраняться значения узлов. Если узел node равен nullptr, метод сразу завершается. Внутри метода создается очередь queue, реализованная на основе шаблонного класса List, и в нее добавляется корневой узел. Для отслеживания позиции в массиве устанавливается индекс index в 0.

Пока очередь не пуста, из нее извлекается первый элемент current, который затем удаляется из очереди. Значение текущего узла current->data сохраняется в массиве array на текущей позиции index, после чего индекс увеличивается. Если левый потомок текущего узла не равен nullptr, он добавляется в очередь, и аналогично, если правый потомок не равен nullptr, он также добавляется в очередь. Таким образом, метод выполняет обход дерева по уровням, начиная с корневого узла и продвигаясь по каждому уровню слева направо.

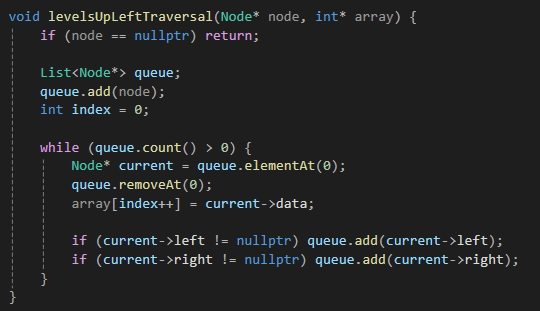


Рисунок 1.4 – Метод levelsUpLeftTraversal для обхода уровней вверх-влево

Метод levelsUpRightTraversal (рис. 1.5) осуществляет обход дерева в ширину, начиная с заданного узла и сохраняя значения узлов в массив. Метод принимает два параметра: указатель на корневой узел дерева node и указатель на массив array, в который будут сохраняться значения узлов. Если узел node равен nullptr, метод сразу завершается. Внутри метода создается очередь queue, реализованная на основе шаблонного класса List, и в нее добавляется корневой узел. Для отслеживания позиции в массиве устанавливается индекс index в 0.

Пока очередь не пуста, из нее извлекается первый элемент current, который затем удаляется из очереди. Значение текущего узла current->data сохраняется в массиве array на текущей позиции index, после чего индекс увеличивается. Если правый потомок текущего узла не равен nullptr, он добавляется в очередь, и аналогично, если левый потомок не равен nullptr, он также добавляется в очередь. Таким образом, метод выполняет обход дерева по уровням, начиная с корневого узла и продвигаясь по каждому уровню справа налево.

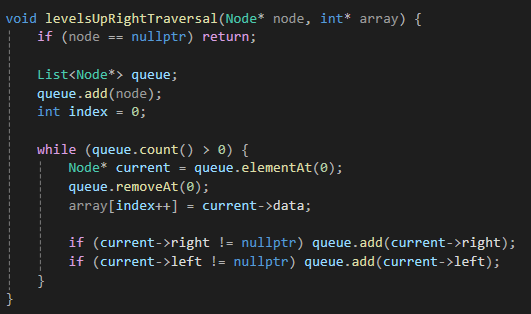


Рисунок 1.5 – Метод levelsUpRightTraversal для обхода уровней вверх-вправо

Метод levelsDownRightTraversal (рис. 1.6) осуществляет обход дерева в ширину, начиная с заданного узла и сохраняя значения узлов в массив в обратном порядке. Метод принимает два параметра: указатель на корневой узел дерева node и указатель на массив array, в который будут сохраняться значения узлов. Если узел node равен nullptr, метод сразу завершается. Внутри метода создается очередь queue, реализованная на основе шаблонного класса List, и в нее добавляется корневой узел. Для отслеживания позиции в массиве устанавливается индекс index в size - 1, где size — это размер массива.

Пока очередь не пуста, вычисляется количество узлов на текущем уровне levelSize, после чего происходит обход всех узлов этого уровня. Из очереди извлекается первый элемент current, который затем удаляется из очереди. Значение текущего узла current->data сохраняется в массиве array на текущей позиции index, после чего индекс уменьшается. Если левый потомок текущего узла не равен nullptr, он добавляется в очередь, и аналогично, если правый потомок не равен nullptr, он также добавляется в очередь. Таким образом, метод выполняет обход дерева по уровням, начиная с корневого узла и продвигаясь по каждому уровню слева направо, но сохраняя значения узлов в массиве в обратном порядке.

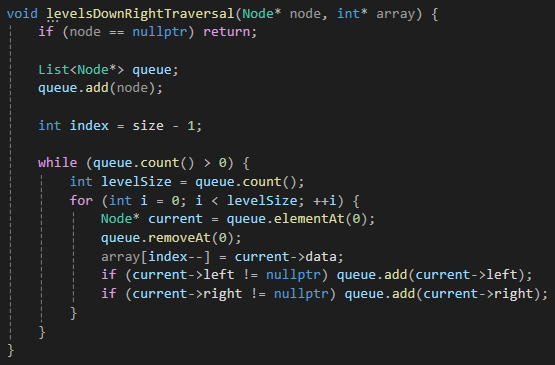


Рисунок 1.6 – Метод levelsDownRightTraversal для обхода уровней вниз-вправо

Метод levelsDownLeftTraversal (рис. 1.7) осуществляет обход дерева в ширину, начиная с заданного узла и сохраняя значения узлов в массив в обратном порядке. Метод принимает два параметра: указатель на корневой узел дерева node и указатель на массив array, в который будут сохраняться значения узлов. Если узел node равен nullptr, метод сразу завершается. Внутри метода создается очередь queue, реализованная на основе шаблонного класса List, и в нее добавляется корневой узел. Для отслеживания позиции в массиве устанавливается индекс index в size - 1, где size — это размер массива.

Пока очередь не пуста, вычисляется количество узлов на текущем уровне levelSize, после чего происходит обход всех узлов этого уровня. Из очереди извлекается первый элемент current, который затем удаляется из очереди. Значение текущего узла current->data сохраняется в массиве array на текущей позиции index, после чего индекс уменьшается. Если правый потомок текущего узла не равен nullptr, он добавляется в очередь, и аналогично, если левый потомок не равен nullptr, он также добавляется в очередь. Таким образом, метод выполняет обход дерева по уровням, начиная с корневого узла и продвигаясь по каждому уровню справа налево, но сохраняя значения узлов в массиве в обратном порядке.

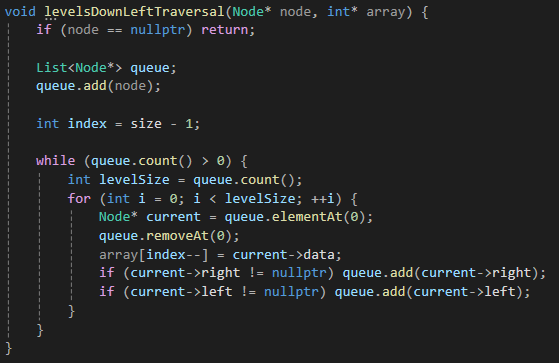


Рисунок 1.7 – Метод levelsDownLeftTraversal для обхода уровней вниз-влево

Метод rotateLeft (рис. 1.8) выполняет левый поворот узла бинарного дерева, что часто используется в алгоритмах балансировки деревьев, таких как AVL-деревья или красно-черные деревья. Повороты помогают поддерживать сбалансированность дерева, что, в свою очередь, обеспечивает эффективные операции вставки, удаления и поиска.

Метод начинается с назначения нового корня: новый корень (newRoot) устанавливается как правый дочерний узел текущего узла (node). Это будет новый корневой узел после выполнения поворота. Затем происходит перемещение поддерева: правый дочерний узел текущего узла теперь ссылается на левый дочерний узел нового корня. Это перемещает левое поддерево нового корня на место правого поддерева текущего узла.

Следующим шагом является перестановка связей: левый дочерний узел нового корня теперь ссылается на текущий узел. Таким образом, текущий узел становится левым дочерним узлом нового корня. В заключение, метод возвращает новый корень, который теперь является корнем поддерева, на котором был выполнен поворот.

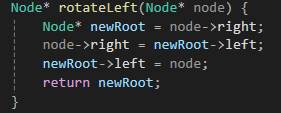


Рисунок 1.8 – Метод rotateLeft для поворота узла дерева влево

Метод rotateRight (рис. 1.9) выполняет правый поворот узла бинарного дерева, что также часто используется в алгоритмах балансировки деревьев, таких как AVL-деревья или красно-черные деревья. Повороты помогают поддерживать сбалансированность дерева, что, в свою очередь, обеспечивает эффективные операции вставки, удаления и поиска.

Метод начинается с назначения нового корня: новый корень (newRoot) устанавливается как левый дочерний узел текущего узла (node). Это будет новый корневой узел после выполнения поворота. Затем происходит перемещение поддерева: левый дочерний узел текущего узла теперь ссылается на правый дочерний узел нового корня. Это перемещает правое поддерево нового корня на место левого поддерева текущего узла.

Следующим шагом является перестановка связей: правый дочерний узел нового корня теперь ссылается на текущий узел. Таким образом, текущий узел становится правым дочерним узлом нового корня. В заключение, метод возвращает новый корень, который теперь является корнем поддерева, на котором был выполнен поворот.

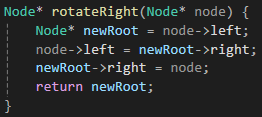


Рисунок 1.9 – Метод rotateRight для поворота узла дерева вправо

Метод balanceHelper (рис. 1.10) используется для балансировки узла бинарного дерева, что необходимо для поддержания равномерной высоты дерева и обеспечения эффективности операций вставки, удаления и поиска. Этот метод рекурсивно балансирует каждое поддерево, корректируя дисбаланс с помощью поворотов.

Метод начинается с проверки, является ли текущий узел нулевым. Если узел нулевой, метод возвращает nullptr, завершая текущий рекурсивный вызов. В противном случае, метод рекурсивно вызывает balanceHelper для левого и правого дочерних узлов текущего узла, тем самым балансируя поддеревья

Затем вычисляется глубина левого и правого поддеревьев. Если глубина левого поддерева больше глубины правого поддерева на более чем один, это указывает на дисбаланс. В таком случае, если правое поддерево левого узла глубже левого поддерева левого узла, выполняется левый поворот на левом дочернем узле. Затем выполняется правый поворот на текущем узле для устранения дисбаланса.

Аналогично, если глубина правого поддерева больше глубины левого поддерева на более чем один, метод проверяет, является ли левое поддерево правого узла глубже правого поддерева правого узла. Если это так, выполняется правый поворот на правом дочернем узле. Затем выполняется левый поворот на текущем узле для устранения дисбаланса.

После выполнения необходимых поворотов метод возвращает текущий узел, который теперь сбалансирован относительно его поддеревьев.



Рисунок 1.10 – Метод balanceHelper для балансировки узла дерева

Метод depth (рис. 1.11) используется для вычисления глубины (высоты) узла бинарного дерева. Глубина узла определяется как количество узлов на самом длинном пути от данного узла до листа (узла без дочерних узлов). Этот метод рекурсивно проходит через дерево, вычисляя глубину каждого поддерева и возвращая максимальную глубину.

Метод начинается с проверки, является ли текущий узел нулевым (nullptr). Если узел нулевой, метод возвращает 0, что указывает на отсутствие узла. Это является базовым случаем рекурсивной функции, предотвращающим дальнейшие вызовы и определяющим глубину пустого поддерева как 0.

Если узел не нулевой, метод рекурсивно вызывает себя для левого и правого дочерних узлов текущего узла, тем самым вычисляя глубину левого и правого поддеревьев. Значения глубины сохраняются в переменные leftDepth и rightDepth соответственно.

После вычисления глубины поддеревьев метод возвращает максимальное значение между leftDepth и rightDepth, увеличенное на 1. Увеличение на 1 учитывает текущий узел в общей глубине. Таким образом, глубина текущего узла равна глубине самого глубокого поддерева плюс один уровень для самого узла.

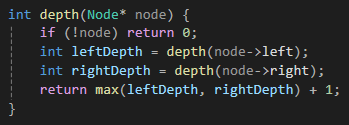


Рисунок 1.11 – Метод depth для вычисления глубины узла дерева

Метод findNode (рис. 1.12) используется для поиска узла в бинарном дереве поиска (BST) по заданному значению. Этот метод рекурсивно проходит через дерево, сравнивая каждое значение узла с искомым значением и переходя к соответствующему дочернему узлу до тех пор, пока не найдет узел с совпадающим значением или не достигнет конца дерева.

Метод начинается с проверки, является ли текущий узел нулевым (nullptr) или содержит ли он значение, равное искомому. Если узел нулевой или его значение совпадает с искомым, метод возвращает текущий узел. Это завершает поиск, если узел найден или если достигнут конец дерева (узел с искомым значением отсутствует).

Если значение искомого элемента меньше значения текущего узла, метод рекурсивно вызывает себя для левого дочернего узла. Это основано на свойстве бинарного дерева поиска, где все значения в левом поддереве меньше значения родительского узла.

Если значение искомого элемента больше значения текущего узла, метод рекурсивно вызывает себя для правого дочернего узла. Это также основано на свойстве бинарного дерева поиска, где все значения в правом поддереве больше значения родительского узла.

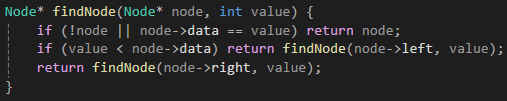


Рисунок 1.12 – Метод findNode для поиска узла в бинарном дереве поиска (BST) по заданному значению

Этот фрагмент кода (рис. 1.13) представляет собой часть объявления класса Tree, который управляет бинарным деревом. Он включает в себя конструктор, деструктор и перечисление, используемое для определения порядка обхода дерева.

Перечисление Order определяет три типа обхода дерева: префиксный (Prefix), инфиксный (Infix), постфиксный (Postfix), обход уровней вверх-влево (LevelsUpLeft), обход уровней вверх-вправо (LevelsUpRight), обход уровней вниз-влево (LevelsDownLeft) и обход уровней вниз-вправо (LevelsDownRight). Эти типы используются для различных методов обхода узлов дерева, указывая порядок, в котором посещаются узлы.

Конструктор инициализирует экземпляр класса Tree. При создании нового объекта дерева его размер устанавливается в 0, что указывает на пустое дерево, а корневой узел (root) инициализируется значением nullptr, что также указывает на отсутствие узлов в дереве. Это начальное состояние дерева, подготовленное для последующих операций добавления узлов.

Деструктор вызывается при уничтожении объекта класса Tree. Он очищает дерево, вызывая метод clear(), который должен освободить все выделенные ресурсы и удалить все узлы дерева. Это важно для управления памятью и предотвращения утечек памяти.

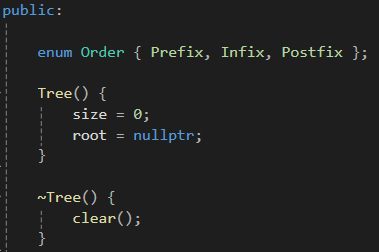


Рисунок 1.13 – Класс Tree с методами и перечислением

Метод add (рис. 1.14) добавляет новый узел в бинарное дерево поиска. При добавлении нового значения value создается новый узел с этим значением. Если дерево пустое (корень root равен nullptr), новый узел становится корнем дерева. В противном случае, метод перемещается от корня к листьям, чтобы найти место для нового узла, сравнивая значение value с данными в узлах. Если value меньше, чем данные текущего узла, метод двигается влево по дереву. Если value больше, чем данные текущего узла, метод двигается вправо. Этот процесс продолжается до тех пор, пока не будет найдено место для вставки нового узла.

Если значение value уже присутствует в дереве, новый узел не добавляется, и память, выделенная для него, освобождается с помощью оператора delete.

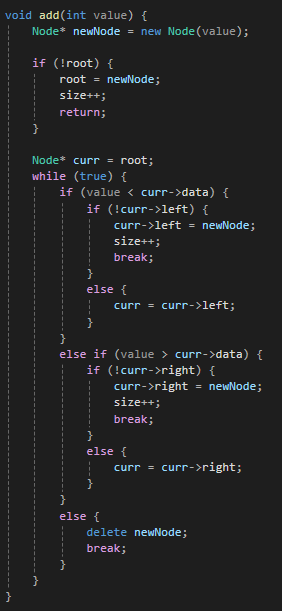


Рисунок 1.14 – Метод add для добавления элемента в дерево

Метод contains (рис. 1.15) в классе Tree проверяет наличие заданного значения value в бинарном дереве поиска. Метод начинает поиск с корневого узла дерева, хранящегося в переменной curr. Цикл продолжается, пока не будет найден узел с искомым значением или пока не будут проверены все узлы в возможном пути (т.е., пока curr не станет nullptr). Если значение меньше данных текущего узла (curr->data), поиск продолжается в левом поддереве, если больше – в правом. Когда значение равно данным в узле, метод возвращает true, подтверждая наличие значения в дереве. Если после обхода всех узлов нужное значение так и не было найдено, метод возвращает false, указывая на отсутствие значения в дереве.

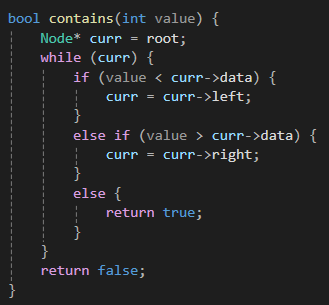


Рисунок 1.15 – Метод для проверки наличия значения value в классе Tree

Метод removeHelper (рис. 1.16) в классе Tree рекурсивно удаляет узел с заданным значением value из бинарного дерева поиска. Если текущий узел node равен nullptr, это означает, что узел с заданным значением отсутствует в дереве, и метод завершает выполнение. Если искомое значение меньше значения в текущем узле, метод вызывается рекурсивно для левого поддерева. Если значение больше — для правого поддерева. Если значение найдено (значение в узле равно искомому), происходит удаление: если у узла нет детей, он удаляется напрямую; если есть только левый или только правый ребенок, узел удаляется, а его ребенок занимает его место; если у узла два ребенка, находится минимальный узел в правом поддереве. Этот минимальный узел заменяет удаляемый, после чего удаляется рекурсивно, чтобы сохранить свойства бинарного дерева поиска. После удаления каждого узла переменная size уменьшается, что отражает общее количество узлов в дереве.

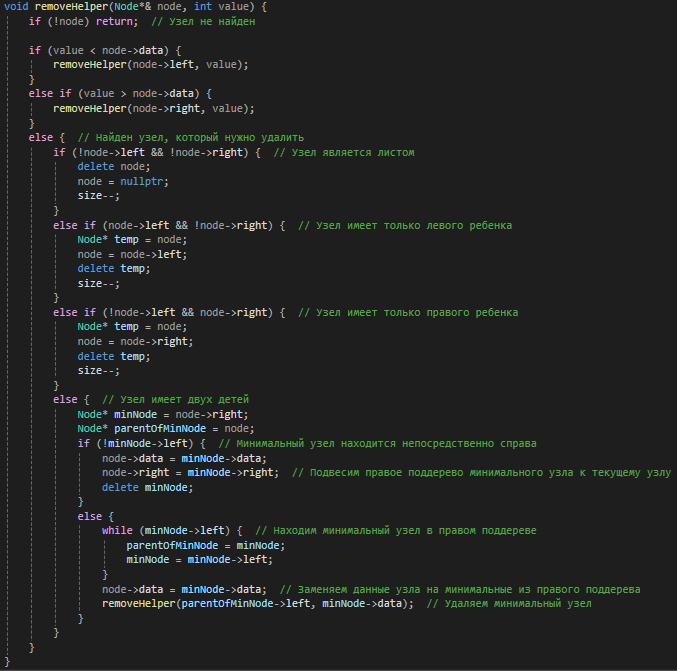


Рисунок 1.16 – Метод removeHelper для рекурсивного удаления узла с заданным значением value

Метод remove (рис 1.17) в классе Tree используется для удаления узла с заданным значением value из бинарного дерева поиска. Он делегирует выполнение этой задачи методу removeHelper, передавая ему корневой узел root и значение value.

Метод count возвращает текущее количество узлов в дереве, хранящееся в переменной size. Это позволяет легко получить общее число узлов в дереве без необходимости пересчитывать их.

Метод clear полностью очищает дерево, удаляя все его узлы и устанавливая значение root в nullptr, а переменную size обнуляет. Для удаления всех узлов вызывается вспомогательный метод clearNode, который рекурсивно обходит дерево, начиная с корневого узла. Метод clearNode вызывает сам себя для левого и правого поддеревьев каждого узла, а затем удаляет сам узел. Этот метод гарантирует, что все узлы дерева будут корректно удалены, предотвращая утечки памяти.

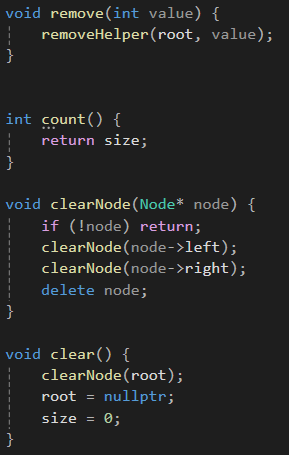


Рисунок 1.17 – Методы remove, count и clear в классе Tree

Метод toArrayHelper (рис. 1.18) используется для рекурсивного обхода бинарного дерева и записи его узлов в массив в определенном порядке. Этот метод является вспомогательным и вызывается из другого метода, который управляет созданием и инициализацией массива. Порядок обхода дерева определяется параметром order, который может принимать значения Prefix, Infix или Postfix из перечисления Order.

Метод начинается с проверки, является ли текущий узел нулевым. Если узел нулевой, метод завершает выполнение и возвращает управление вызвавшему методу, что является базовым случаем рекурсивной функции, предотвращающим дальнейшие вызовы и обрабатывающим пустые поддеревья. В зависимости от значения параметра order, метод записывает данные текущего узла в массив на разных этапах обхода. При префиксном обходе (Prefix) данные текущего узла записываются в массив до обхода левого и правого поддеревьев. При инфиксном обходе (Infix) данные текущего узла записываются в массив между обходом левого и правого поддеревьев. При постфиксном обходе (Postfix) данные текущего узла записываются в массив после обхода левого и правого поддеревьев.

Рекурсивные вызовы метода выполняются для левого и правого поддеревьев текущего узла, что обеспечивает полный обход дерева. Метод сначала обходит левое поддерево, затем записывает данные текущего узла (если это предусмотрено порядком обхода), и, наконец, обходит правое поддерево. В процессе выполнения метод обновляет индекс текущей позиции в массиве, чтобы данные каждого узла записывались в правильную позицию.

Таким образом, метод toArrayHelper обходит дерево в зависимости от указанного порядка и записывает данные узлов в массив. Это позволяет преобразовать структуру данных дерева в линейный массив, который может быть использован для дальнейших операций или анализа.

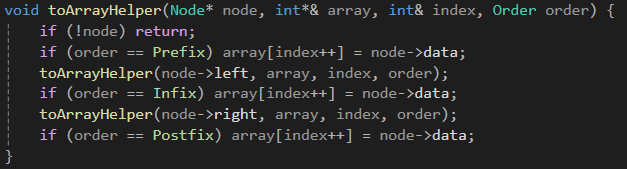


Рисунок 1.18 – Метод toArrayHelper для преобразования дерева в массив

Методы ToArray (рис. 1.19) в классе Tree предназначены для преобразования дерева в массив, упрощая доступ к элементам дерева в линейной структуре данных. Один метод предоставляет возможность указать порядок обхода дерева, в то время как другой использует порядок обхода по умолчанию.

Первый метод ToArray(Order order) преобразует дерево в массив, используя указанный порядок обхода. Метод принимает параметр order, который определяет тип обхода, указанный в перечислении Order. В начале метода создается массив array размером, равным количеству узлов в дереве (size), и переменная index, инициализируемая значением 0.

В зависимости от значения order, метод вызывает соответствующую функцию для выполнения обхода дерева. Для префиксного (Prefix), инфиксного (Infix) и постфиксного (Postfix) обходов используется вспомогательный метод toArrayHelper. Для обходов уровней вызываются соответствующие методы: levelsUpLeftTraversal, levelsUpRightTraversal, levelsDownLeftTraversal и levelsDownRightTraversal. В конце метода возвращается указатель на массив array, содержащий значения узлов дерева в порядке, заданном параметром order.

Второй метод ToArray() является перегруженной версией первого метода и вызывает его с параметром по умолчанию Infix. Это означает, что дерево будет преобразовано в массив с использованием инфиксного обхода, при котором элементы дерева будут записаны в массив в порядке возрастания, если дерево является бинарным деревом поиска. Метод ToArray() упрощает вызов, предоставляя удобный способ получения массива без необходимости явно указывать порядок обхода.

В совокупности, эти методы обеспечивают удобное преобразование дерева в массив, позволяя выбирать как порядок обхода, так и использовать порядок по умолчанию.

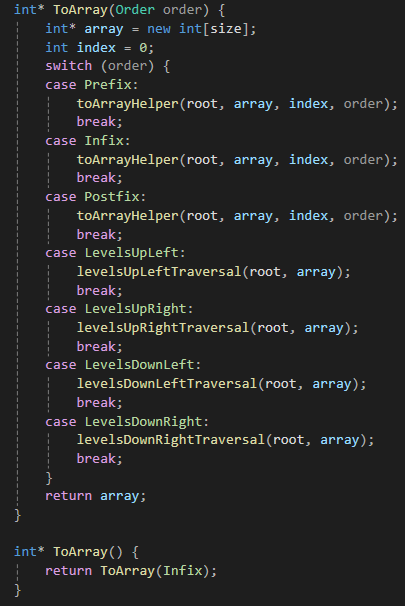


Рисунок 1.19 – Методы ToArray для преобразования дерева в массив

Метод ToLeft (рис. 1.20) предназначен для выполнения левого поворота узла бинарного дерева, если узел с указанным значением найден. Левый поворот полезен для балансировки дерева и может улучшить его структуру для более эффективного выполнения операций.

Метод начинается с инициализации двух указателей: parent и node. parent используется для отслеживания родительского узла текущего узла, а node начинает с корневого узла дерева. Затем метод выполняет цикл поиска, который проходит по дереву, сравнивая значение каждого узла с заданным значением value. Если текущее значение узла меньше искомого значения, метод переходит к левому дочернему узлу; если больше — к правому дочернему узлу. В процессе поиска parent обновляется для отслеживания родительского узла.

Если после поиска узел с указанным значением не найден или у найденного узла нет правого дочернего узла, метод завершает выполнение, так как невозможно выполнить левый поворот без правого дочернего узла. В противном случае метод проверяет, является ли найденный узел корневым узлом. Если это так, корень дерева обновляется на результат выполнения метода rotateLeft, который выполняет левый поворот. Если найденный узел не является корневым, метод проверяет, является ли он левым или правым дочерним узлом своего родительского узла, и соответственно обновляет ссылку родительского узла на результат выполнения метода rotateLeft.

Таким образом, метод ToLeft выполняет левый поворот узла, если узел с указанным значением и его правый дочерний узел существуют.

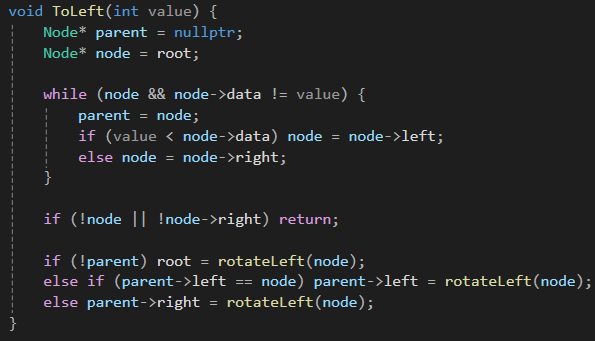


Рисунок 1.20 – Метод ToLeft для выполнения левого поворота узла дерева

Метод ToRight (рис. 1.21) предназначен для выполнения правого поворота узла бинарного дерева, если узел с указанным значением найден. Правый поворот полезен для балансировки дерева и может улучшить его структуру для более эффективного выполнения операций.

Метод начинается с инициализации двух указателей: parent и node. parent используется для отслеживания родительского узла текущего узла, а node начинает с корневого узла дерева. Затем метод выполняет цикл поиска, который проходит по дереву, сравнивая значение каждого узла с заданным значением value. Если текущее значение узла меньше искомого значения, метод переходит к левому дочернему узлу; если больше — к правому дочернему узлу. В процессе поиска parent обновляется для отслеживания родительского узла.

Если после поиска узел с указанным значением не найден или у найденного узла нет левого дочернего узла, метод завершает выполнение, так как невозможно выполнить правый поворот без левого дочернего узла. В противном случае метод проверяет, является ли найденный узел корневым узлом. Если это так, корень дерева обновляется на результат выполнения метода rotateRight, который выполняет правый поворот. Если найденный узел не является корневым, метод проверяет, является ли он левым или правым дочерним узлом своего родительского узла, и соответственно обновляет ссылку родительского узла на результат выполнения метода rotateRight.

Таким образом, метод ToRight выполняет правый поворот узла, если узел с указанным значением и его левый дочерний узел существуют.

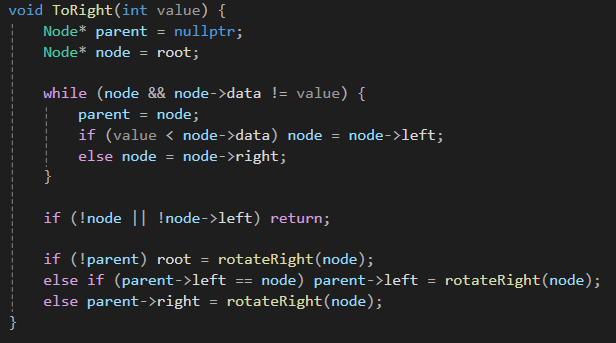


Рисунок 1.21 – Метод ToRight для выполнения правого поворота узла дерева

Метод Balance (рис. 1.22) предназначен для балансировки бинарного дерева, обеспечивая его оптимальную структуру для эффективного выполнения операций. Этот метод использует вспомогательный метод balanceHelper для рекурсивного балансирования узлов дерева.

Метод Balance вызывает метод balanceHelper с корневым узлом дерева в качестве аргумента. Результат выполнения balanceHelper назначается корневому узлу дерева, что обновляет корень и его поддеревья, обеспечивая сбалансированное состояние всего дерева.

Метод balanceHelper рекурсивно обходит дерево и выполняет необходимые повороты (левые или правые) для устранения дисбаланса, который может возникнуть в результате операций вставки или удаления узлов. В процессе выполнения balanceHelper проверяет глубину левого и правого поддеревьев каждого узла и, если обнаруживает дисбаланс, выполняет соответствующие повороты для восстановления равновесия.

Таким образом, метод Balance обеспечивает вызов balanceHelper для всего дерева, что позволяет сбалансировать структуру дерева и поддерживать его оптимальное состояние для последующих операций.

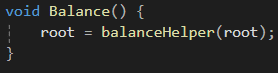


Рисунок 1.22 – Метод Balance для балансировки дерева

Функция main (рис. 1.23) является точкой входа в программу и управляет взаимодействием с деревом Tree. В функции создается объект дерева t, а затем начинается цикл для обработки пользовательского ввода. Ввод считывается в строковую переменную choice, которая определяет выполняемое действие. В зависимости от значения choice выполняются различные операции над деревом.

Если choice равно "add", то считывается значение data и добавляется в дерево с помощью функции add. Если choice равно "remove", считывается значение data и удаляется из дерева с помощью функции remove. При выборе "clear" дерево очищается функцией clear, а при выборе "count" выводится количество узлов в дереве.

При выборе "toarray" считывается строка orderStr, определяющая порядок обхода. Переменная order устанавливается в соответствующее значение перечисления Order. Затем вызывается функция ToArray, которая преобразует дерево в массив в заданном порядке, и выводится содержимое массива. После этого массив удаляется из памяти.

Если choice равно "toleft" или "toright", считывается значение data и выполняется соответствующая операция (ToLeft или ToRight). Выбор "balance" вызывает функцию Balance для балансировки дерева. Если choice равно "stop", цикл завершается, и программа выходит. Если введена неизвестная команда, программа также завершает работу.



Рисунок 1.23 – Функция main для управления операциями с деревом

**Выводы.**

В ходе выполнения лабораторной работы была разработана программа, реализующая структуру данных бинарного дерева поиска `Tree` с функциями `add(int)`, `remove(int)`, `clear()`, `contains(int)` и `count()`. Структура была дополнена перечислением `Order` для задания порядка обхода дерева и методами `ToArray(Order)`, `ToArray()`, `ToLeft(int)`, `ToRight(int)` и `Balance()`, обеспечивающими преобразование дерева в массив, выполнение поворотов узлов и балансировку дерева. Дополнительно реализованы функции обхода дерева по уровням в различных направлениях: вверх-влево, вверх-вправо, вниз-влево и вниз-вправо. В основной функции `main` реализован интерфейс командной строки для взаимодействия с деревом, предоставляющий возможность добавлять, удалять и считывать узлы, очищать дерево, преобразовывать его в массив и балансировать дерево. Эти дополнения позволяют поддерживать дерево в сбалансированном состоянии и обеспечивают эффективное выполнение операций с его элементами.

# Список использованной литературы

1. Павловская Т.А., Чаевников В.В., Юрков Н.К. Программирование на языке С++. Электронное методическое пособие. СПб.: Издательство СПбГЭТУ «ЛЭТИ». 2014.
2. Практикум по процедурному программированию на языке C++ С. А. Ивановский, А. А. Лисс, В. П. Самойленко, О. М. Шолохова. Практикум по процедурному программированию на языке C++: учеб. пособие. СПб.: Издательство СПбГЭТУ«ЛЭТИ», 2016.
3. Справочник по стандартной библиотеке C++ (STL) // Справочный сайт learn.microsoft.com URL: https://learn.microsoft.com/ru-ru/cpp/standard-library/cpp-standard-library-reference?view=msvc-170 (дата обращения 09.03.2024).

Документация по языку C++ // Справочный сайт learn.microsoft.com URL: https://learn.microsoft.com/ru-ru/cpp/cpp/?view=msvc-170 (дата обращения 10.03.2024).

# Приложение

## Код написанной программы

Файл «Lab7.cpp»:

#include <iostream>

#include "List.h"

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = nullptr;

right = nullptr;

}

};

class Tree {

private:

int size;

Node\* root;

void levelsUpLeftTraversal(Node\* node, int\* array) {

if (node == nullptr) return;

List<Node\*> queue;

queue.add(node);

int index = 0;

while (queue.count() > 0) {

Node\* current = queue.elementAt(0);

queue.removeAt(0);

array[index++] = current->data;

if (current->left != nullptr) queue.add(current->left);

if (current->right != nullptr) queue.add(current->right);

}

}

void levelsUpRightTraversal(Node\* node, int\* array) {

if (node == nullptr) return;

List<Node\*> queue;

queue.add(node);

int index = 0;

while (queue.count() > 0) {

Node\* current = queue.elementAt(0);

queue.removeAt(0);

array[index++] = current->data;

if (current->right != nullptr) queue.add(current->right);

if (current->left != nullptr) queue.add(current->left);

}

}

void levelsDownRightTraversal(Node\* node, int\* array) {

if (node == nullptr) return;

List<Node\*> queue;

queue.add(node);

int index = size - 1;

while (queue.count() > 0) {

int levelSize = queue.count();

for (int i = 0; i < levelSize; ++i) {

Node\* current = queue.elementAt(0);

queue.removeAt(0);

array[index--] = current->data;

if (current->left != nullptr) queue.add(current->left);

if (current->right != nullptr) queue.add(current->right);

}

}

}

void levelsDownLeftTraversal(Node\* node, int\* array) {

if (node == nullptr) return;

List<Node\*> queue;

queue.add(node);

int index = size - 1;

while (queue.count() > 0) {

int levelSize = queue.count();

for (int i = 0; i < levelSize; ++i) {

Node\* current = queue.elementAt(0);

queue.removeAt(0);

array[index--] = current->data;

if (current->right != nullptr) queue.add(current->right);

if (current->left != nullptr) queue.add(current->left);

}

}

}

Node\* rotateLeft(Node\* node) {

Node\* newRoot = node->right;

node->right = newRoot->left;

newRoot->left = node;

return newRoot;

}

Node\* rotateRight(Node\* node) {

Node\* newRoot = node->left;

node->left = newRoot->right;

newRoot->right = node;

return newRoot;

}

Node\* balanceHelper(Node\* node) {

if (!node) return nullptr;

node->left = balanceHelper(node->left);

node->right = balanceHelper(node->right);

int leftDepth = depth(node->left);

int rightDepth = depth(node->right);

if (leftDepth > rightDepth + 1) {

if (depth(node->left->left) < depth(node->left->right)) {

node->left = rotateLeft(node->left);

}

node = rotateRight(node);

}

else if (rightDepth > leftDepth + 1) {

if (depth(node->right->right) < depth(node->right->left)) {

node->right = rotateRight(node->right);

}

node = rotateLeft(node);

}

return node;

}

int depth(Node\* node) {

if (!node) return 0;

int leftDepth = depth(node->left);

int rightDepth = depth(node->right);

return max(leftDepth, rightDepth) + 1;

}

Node\* findNode(Node\* node, int value) {

if (!node || node->data == value) return node;

if (value < node->data) return findNode(node->left, value);

return findNode(node->right, value);

}

public:

enum Order { Prefix, Infix, Postfix, LevelsUpLeft, LevelsUpRight, LevelsDownLeft, LevelsDownRight };

Tree() {

size = 0;

root = nullptr;

}

~Tree() {

clear();

}

void add(int value) {

Node\* newNode = new Node(value);

if (!root) {

root = newNode;

size++;

return;

}

Node\* curr = root;

while (true) {

if (value < curr->data) {

if (!curr->left) {

curr->left = newNode;

size++;

break;

}

else {

curr = curr->left;

}

}

else if (value > curr->data) {

if (!curr->right) {

curr->right = newNode;

size++;

break;

}

else {

curr = curr->right;

}

}

else {

delete newNode;

break;

}

}

}

bool contains(int value) {

Node\* curr = root;

while (curr) {

if (value < curr->data) {

curr = curr->left;

}

else if (value > curr->data) {

curr = curr->right;

}

else {

return true;

}

}

return false;

}

void removeHelper(Node\*& node, int value) {

if (!node) return; // Узел не найден

if (value < node->data) {

removeHelper(node->left, value);

}

else if (value > node->data) {

removeHelper(node->right, value);

}

else { // Найден узел, который нужно удалить

if (!node->left && !node->right) { // Узел является листом

delete node;

node = nullptr;

size--;

}

else if (node->left && !node->right) { // Узел имеет только левого ребенка

Node\* temp = node;

node = node->left;

delete temp;

size--;

}

else if (!node->left && node->right) { // Узел имеет только правого ребенка

Node\* temp = node;

node = node->right;

delete temp;

size--;

}

else { // Узел имеет двух детей

Node\* minNode = node->right;

Node\* parentOfMinNode = node;

if (!minNode->left) { // Минимальный узел находится непосредственно справа

node->data = minNode->data;

node->right = minNode->right; // Подвесим правое поддерево минимального узла к текущему узлу

delete minNode;

}

else {

while (minNode->left) { // Находим минимальный узел в правом поддереве

parentOfMinNode = minNode;

minNode = minNode->left;

}

node->data = minNode->data; // Заменяем данные узла на минимальные из правого поддерева

removeHelper(parentOfMinNode->left, minNode->data); // Удаляем минимальный узел

}

}

}

}

void remove(int value) {

removeHelper(root, value);

}

int count() {

return size;

}

void clearNode(Node\* node) {

if (!node) return;

clearNode(node->left);

clearNode(node->right);

delete node;

}

void clear() {

clearNode(root);

root = nullptr;

size = 0;

}

void toArrayHelper(Node\* node, int\*& array, int& index, Tree::Order order) {

if (!node) return;

if (order == Prefix) array[index++] = node->data;

toArrayHelper(node->left, array, index, order);

if (order == Infix) array[index++] = node->data;

toArrayHelper(node->right, array, index, order);

if (order == Postfix) array[index++] = node->data;

}

int\* ToArray(Order order) {

int\* array = new int[size];

int index = 0;

switch (order) {

case Prefix:

toArrayHelper(root, array, index, order);

break;

case Infix:

toArrayHelper(root, array, index, order);

break;

case Postfix:

toArrayHelper(root, array, index, order);

break;

case LevelsUpLeft:

levelsUpLeftTraversal(root, array);

break;

case LevelsUpRight:

levelsUpRightTraversal(root, array);

break;

case LevelsDownLeft:

levelsDownLeftTraversal(root, array);

break;

case LevelsDownRight:

levelsDownRightTraversal(root, array);

break;

}

return array;

}

int\* ToArray() {

return ToArray(Infix);

}

void ToLeft(int value) {

Node\* parent = nullptr;

Node\* node = root;

while (node && node->data != value) {

parent = node;

if (value < node->data) node = node->left;

else node = node->right;

}

if (!node || !node->right) return;

if (!parent) root = rotateLeft(node);

else if (parent->left == node) parent->left = rotateLeft(node);

else parent->right = rotateLeft(node);

}

void ToRight(int value) {

Node\* parent = nullptr;

Node\* node = root;

while (node && node->data != value) {

parent = node;

if (value < node->data) node = node->left;

else node = node->right;

}

if (!node || !node->left) return;

if (!parent) root = rotateRight(node);

else if (parent->left == node) parent->left = rotateRight(node);

else parent->right = rotateRight(node);

}

void Balance() {

root = balanceHelper(root);

}

};

int main() {

Tree t;

string choice;

int data;

while (true) {

cin >> choice;

if (choice == "add") {

cin >> data;

t.add(data);

}

else if (choice == "remove") {

cin >> data;

t.remove(data);

}

else if (choice == "clear") {

t.clear();

}

else if (choice == "count") {

cout << t.count() << endl;

}

else if (choice == "toarray") {

string orderStr;

cin >> orderStr;

Tree::Order order = Tree::Infix;

if (orderStr == "prefix") order = Tree::Prefix;

else if (orderStr == "postfix") order = Tree::Postfix;

else if (orderStr == "levelsupleft") order = Tree::LevelsUpLeft;

else if (orderStr == "levelsupright") order = Tree::LevelsUpRight;

else if (orderStr == "levelsdownleft") order = Tree::LevelsDownLeft;

else if (orderStr == "levelsdownright") order = Tree::LevelsDownRight;

int\* arr = t.ToArray(order);

for (int i = 0; i < t.count(); i++) {

cout << arr[i] << " ";

}

cout << endl;

delete[] arr;

}

else if (choice == "toleft") {

cin >> data;

t.ToLeft(data);

}

else if (choice == "toright") {

cin >> data;

t.ToRight(data);

}

else if (choice == "balance") {

t.Balance();

}

else if (choice == "stop") {

break;

}

else {

break;

}

}

}

Файл «List.h»:

#pragma once

#include <string>

#include <iostream>

#include <windows.h>

#include <stdexcept>

template <typename T>

class List {

private:

struct Node {

T data;

Node\* prev;

Node\* next;

Node(T value) {

data = value;

prev = next = nullptr;

}

};

Node\* head;

Node\* tail;

int size;

Node\* currentNode;

int currentIndex;

Node\* getNode(int index) {

// Оптимизация: если запрашиваемый индекс близок к последнему

// использованному, идём от currentNode

if (currentNode != nullptr && abs(index - currentIndex) < size / 2) {

Node\* current = currentNode;

if (index > currentIndex) {

for (int i = currentIndex; i < index; i++) {

current = current->next;

}

}

else {

for (int i = currentIndex; i > index; i--) {

current = current->prev;

}

}

currentNode = current;

currentIndex = index;

return current;

}

// В остальных случаях идём от начала или конца списка

Node\* current;

if (index < size / 2) {

current = head;

for (int i = 0; i < index; i++) {

current = current->next;

}

}

else {

current = tail;

for (int i = size - 1; i > index; i--) {

current = current->prev;

}

}

currentNode = current;

currentIndex = index;

return current;

}

public:

List() {

head = tail = currentNode = nullptr;

size = currentIndex = 0;

}

~List() {

clear();

}

void add(T data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

}

else {

tail->next = newNode;

newNode->prev = tail;

}

tail = newNode;

size++;

}

void insert(int index, T data) {

if (index < 0 || index > size) {

return;

}

if (index == size) {

add(data);

return;

}

Node\* newNode = new Node(data);

if (index == 0) {

newNode->next = head;

head->prev = newNode;

head = newNode;

}

else {

Node\* curr = getNode(index);

newNode->prev = curr->prev;

newNode->next = curr;

curr->prev = newNode;

newNode->prev->next = newNode;

}

currentNode = newNode;

currentIndex = index;

size++;

}

void removeAt(int index) {

if (index < 0 || index >= size) {

return;

}

Node\* temp = nullptr;

if (index == 0) {

temp = head;

head = head->next;

if (head) {

head->prev = nullptr;

}

else {

tail = nullptr;

}

}

else if (index == size - 1) {

temp = tail;

tail = tail->prev;

tail->next = nullptr;

}

else {

temp = getNode(index);

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

}

if (temp == currentNode) {

currentNode = temp->prev;

currentIndex = max(0, index - 1);

}

else if (index < currentIndex) {

currentIndex--;

}

delete temp;

size--;

if (size == 0) {

head = tail = currentNode = nullptr;

currentIndex = 0;

}

}

T elementAt(int index) {

if (index < 0 || index >= size) {

throw ("");

}

if (index == currentIndex && currentNode != nullptr) {

return currentNode->data;

}

return getNode(index)->data;

}

int count() {

return size;

}

void clear() {

Node\* current = head;

while (current) {

Node\* next = current->next;

delete current;

current = next;

}

head = tail = nullptr;

size = 0;

}

};

Файл data1.txt:

add 10

add 5

add 15

add 3

add 7

add 13

add 17

toarray infix

toarray prefix

toarray postfix

toarray levelsupleft

toarray levelsupright

toarray levelsdownleft

toarray levelsdownright

count

toleft 10

toright 5

balance

toarray infix

clear

count

stop

Файл report1.txt:

3 5 7 10 13 15 17

10 5 3 7 15 13 17

3 7 5 13 17 15 10

10 5 15 3 7 13 17

10 15 5 17 13 7 3

3 7 13 17 5 15 10

17 13 7 3 15 5 10

7

3 5 7 10 13 15 17

0