Дисциплина: «Теория алгоритмов»

**Лабораторная работа № 2**

# Нелинейные древовидные структуры данных

### Цель работы

Изучение и исследование методов реализации и балансировки нелинейных древовидных структур с элементами сортировки и упорядоченности: двоичные деревья поиска на примере «Двоичного дерева поиска» , «Двоичной кучи», «Префиксного дерева», «Дерева выражений».

### Краткие теоретические сведения

**1. Структура двоичного дерева поиска (Binary search tree – BST)**

Двоичное дерево поиска – это древовидная структура, разработанная для эффективной вставки, удаления и поиска.

BST – это во многом наиболее важная, оптимизированная для поиска структура данных.

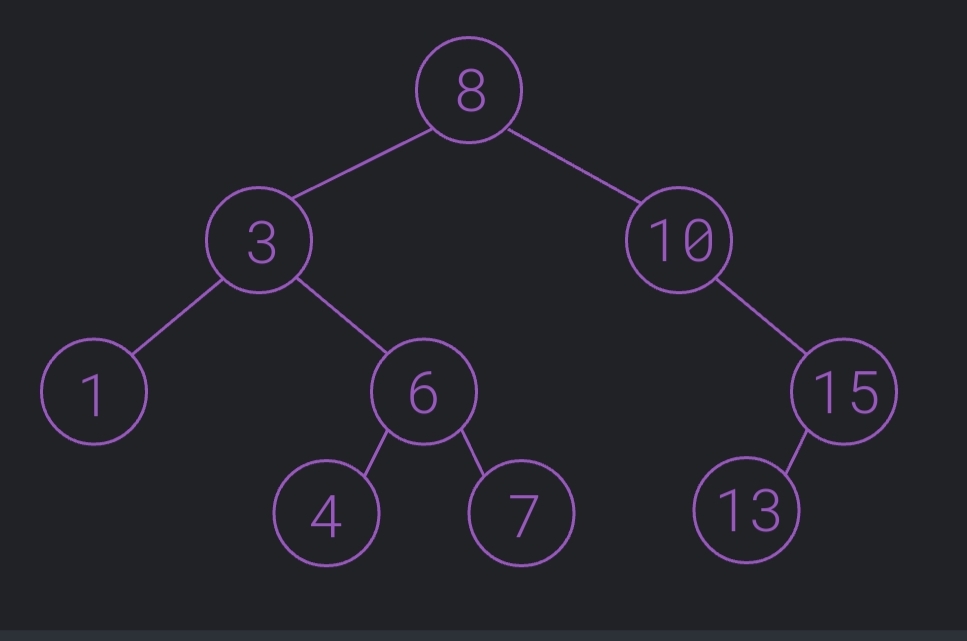
Двоичное дерево – это дерево, в котором каждый узел может иметь не более двух потомков, значения в потомках часто называют ключами (keys).

**Свойства BST:**

1) Оба поддерева— левое и правое— являются двоичными деревьями поиска.

2) У всех узлов *левого* поддерева произвольного узла X значения ключей данных *меньше* значения ключа данных самого узла X.

3) У всех узлов *правого* поддерева произвольного узла X значения ключей данных *больше либо равны* значениям ключа данных самого узла X.

Рисунок 1 — пример BST-дерева

Для примеров отображения BST часто используются числа, хотя такие структуры могут хранить любые данные, которые можно упорядочить (например строки в алфавитном порядке).

Существует одно свойство, которое делает эффективным поиск по дереву BST:

**Для каждого ключа, значение в левом потомке меньше, а в правом – больше его самого.**

Иногда возникает **задача проверки** – является ли двоичное дерево – двоичным деревом поиска?

Алгоритм проверки:

1. Значение текущего узла должно быть максимальным, по сравнению со значениями других узлов его левого поддерева.

2. Значение текущего узла должно быть минимальным, по сравнению со значениями других узлов его правого поддерева.

Особенность: для данного алгоритма изначально мы должны передать в нее значения minKey (минимально-возможное значение) и maxKey (максимально-возможно значение)

Эти ключи зависят от типа данных, которые хранит дерево, а также от особенностей языка программирования, с помощью которого реализуется BST-дерево.

Если мы реализуем дерево на С и оно хранит целые числа, то можно воспользоваться соответственно константами INT\_MIN и INT\_MAX

**Для вставки нового ключа** в двоичное дерево поиска мы должны обойти по левым и правым потомкам, в зависимости от значений вставляемого ключа в уже существующих в узлах, и найти первый пустой лист, для которого подходит ключ.

*1. Новый ключ сравнивается в корнем*

*2. Сравниваем ключ и значение в корне:*

*2А. Если ключ меньше, то далее ключ сравнивается с левым потомком.*

*2Б. Если ключ больше – то в правым.*

*3. Процесс продолжается до тех пор, пока ключ не будет сравниваться с листом.*

*4. Ключ вставляется как потомок найденного листа (слева или справа, в зависимости от значения листа).*

Особенности:

Форма дерева BST зависит от последовательности, в которой вставляются элементы. Например, если данные вставляются в возрастающем или убывающем порядке, то дерево вырождается в отсортированный связанный список. В этом случае высота дерева (расстояние от корня до самого дальнего листа) равна количеству элементов дерева.

В этом случае трудоемкость обхода из O(log n) вырождается в O(n)

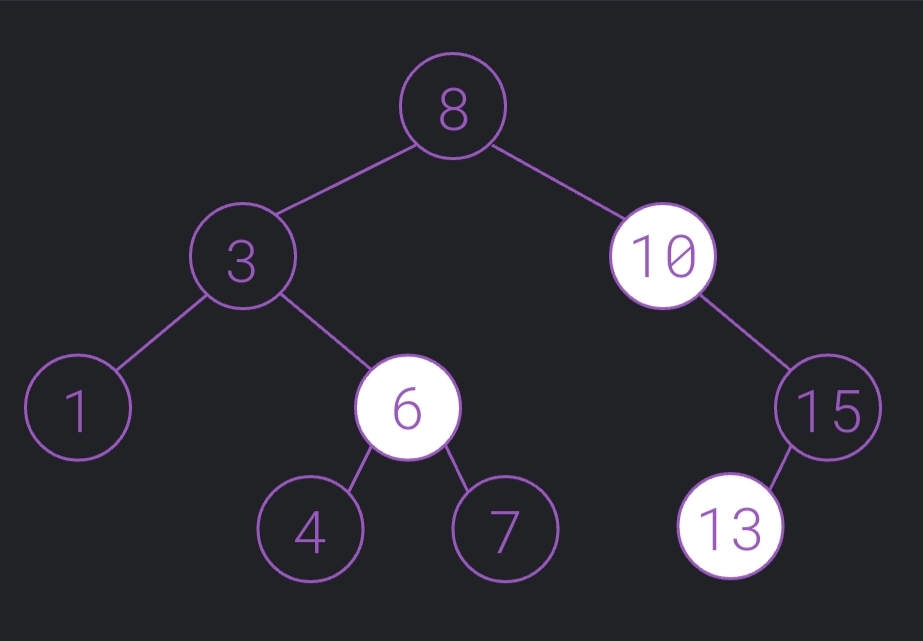
**Удаление элемента из двоичного дерева поиска (BST)**

Удаление элемента из BST сложнее, чем вставка, поскольку необходимо предусмотреть **3** возможных случая:

1. У удаляемого элемента нет потомков

2. У удаляемого элемента один потомок

3. У удаляемого элемента два потомка

Рисунок 3 — пример вариантов удаления

Первые два случая простые:

1. Если у удаляемого узла нет потомков, то удаляем данный узел (и ссылку на него у родителя)

2. Если у удаляемого узла только один потомок, то удаляем узел и заменяем его единственным потомком.

Если у удаляемого узла два потомка, то **необходимо соблюсти правило упорядочивания BST:** каждый ключ в левом поддереве меньше удаляемого узла, и каждый ключ в правом поддереве больше удаляемого узла.

Алгоритм для третьего случая:

1. Если у узла два потомка, заменить его узлом с максимальным ключом из левого поддерева.

2. Рекурсивно удалить максимальный ключ из левого поддерева.

Обратите внимание:

Узел с максимальным значением в любом поддереве дерева BST никогда не будет иметь двух потомков, т.к. у него не будет правого потомка (иначе он был бы максимальным). Это гарантирует, что наша операция удаления рекурсивно будет вызвана не более одного раза.

Аналогично, можно искать минимальный элемент правого поддерева, вместо максимального левого поддерева.

**2. Двоичная куча. Куча (Heap Data Structure)**

Куча – это древовидная структура данных, предназначенная для осуществления операций поиска максимума (find-max) и минимума (find-min) за постоянное время, в дополнение к эффективной вставки новых элементов и эффективному удалению максимального (минимального) элемента.

Кучи часто используются для реализации очередей приоритетов, где элементы расположены в порядке приоритета (или какого-то другого фактора), а не по порядку вставки.

Куча, поддерживающая операцию find-max называется max-heap.

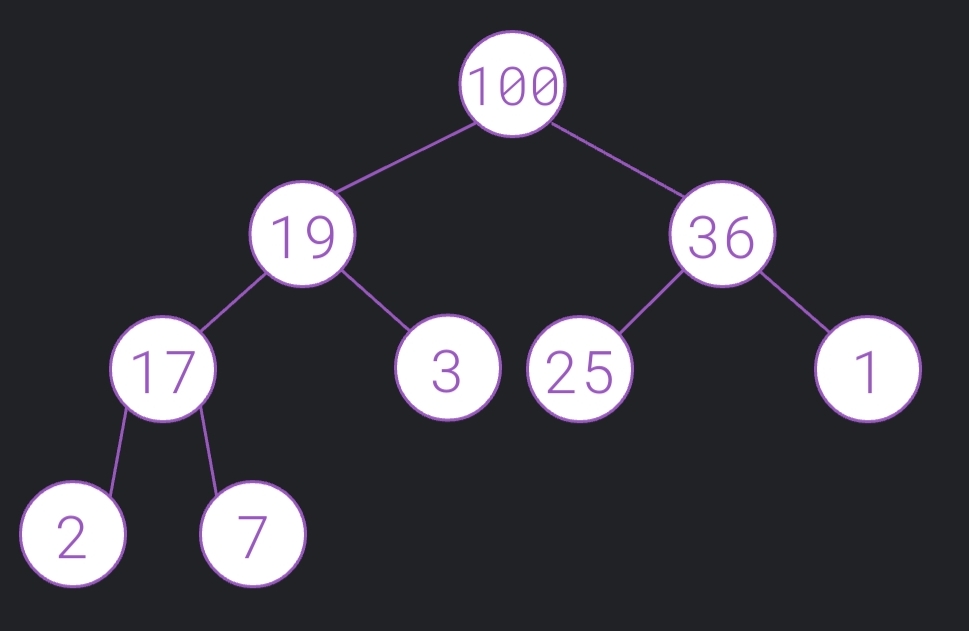
Куча, поддерживающая операцию find-min называется min-heap.

Значения в куче называются ключами (keys) и должны удовлетворять следующему свойству кучи:

Для max-heap ключи родительских узлов всегда больше или равны, ключам потомков, и самый большой ключ находится в корне кучи.

Для min-heap все наоборот (меняем максимальный на минимальный).

Самый распространенный вид кучи- двоичная куча (binary heap), построена, соответственно, на двоичном дереве.

Рисунок 4 — пример двоичной кучи

Поиск максимального элемента очень прост – это корень кучи.

Но при этом сохранение информации в дереве со свойствами кучи потребует трудоемкости O(log(n)) на вставку и удаление.

Если бы мы сохраняли ту же информацию в сортированном списке, то у нас было бы постоянное время при поиске find-max и постоянное время удаления, но трудоемкость вставки была бы O(n)

**Каким образом эффективнее всего вставить элемент у кучу?** (дерево двоичное, max-heap) (здесь не работает алгоритм с BST)

Необходимо соблюсти основные правила кучи:

1) Дерево является полным бинарным деревом (т.е. каждый нижний уровень заполнен, исключения возможны только для последнего уровня)

2) Ключ родительского узла всегда больше или равен ключу вставляемого элемента.

Для того, чтобы удовлетворить первому условию, необходимо вставлять данные в нужное место.

Допустим мы хотим вставить ключ со значением 120 в дерево на рисунке 5.

Если мы начнем со вставки его в первое доступное пустое место, то получится так:

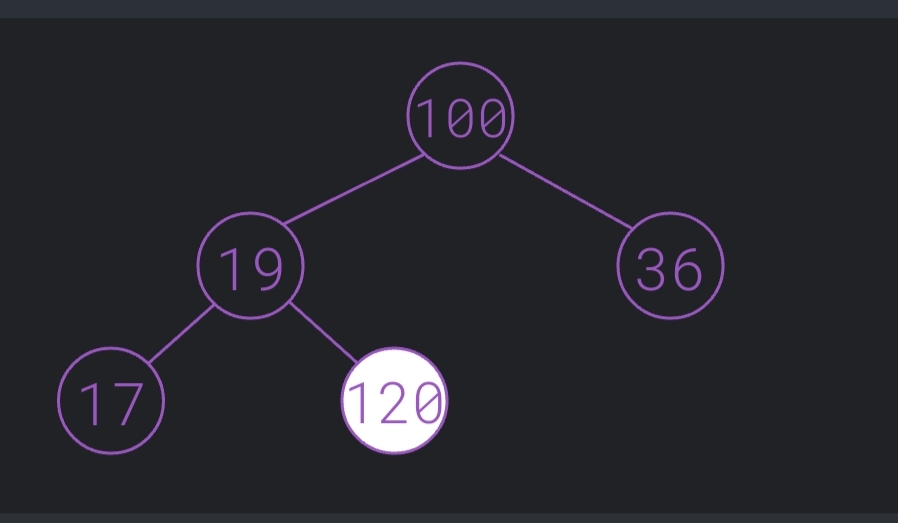


Рисунок 5 — этап вставки элемента в кучу

Но теперь дерево не удовлетворяет условию (2), поскольку 120 больше 19.

Для того, чтобы удовлетворить условию, необходимо сравнить ключ элемента в ключом родителя и поменять их местами, в случае, если условие кучи нарушается.

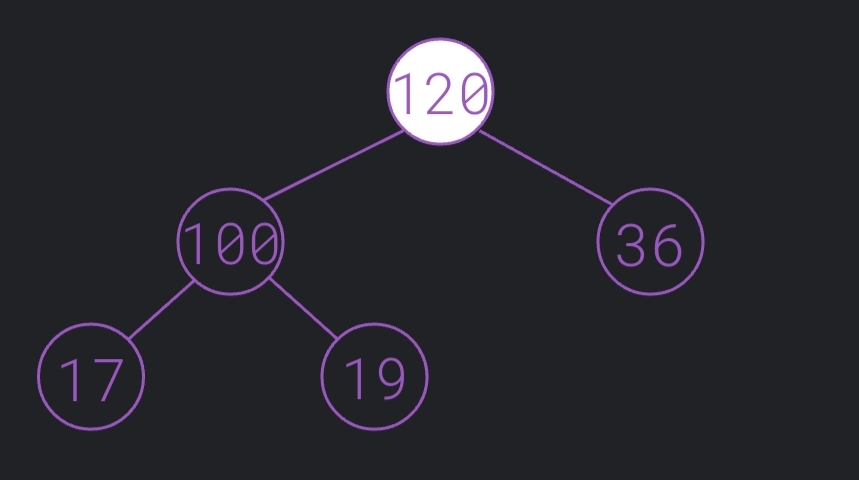
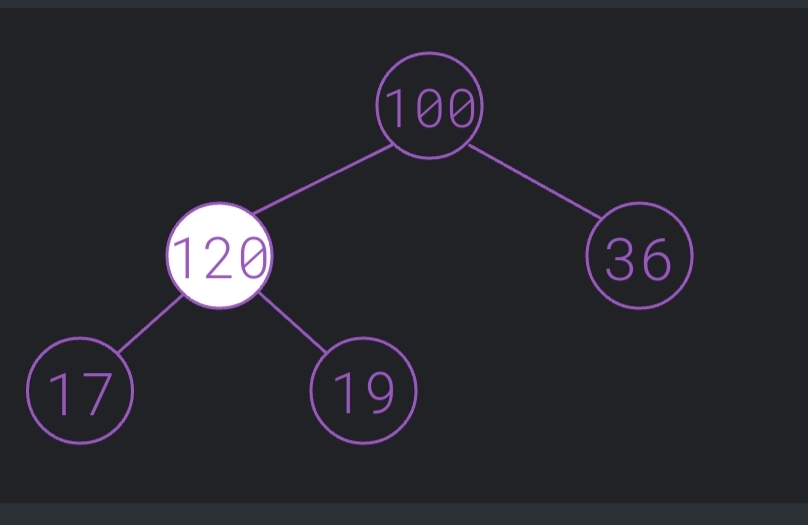
Сравнения продолжаются до тех пор пока элемент не достигнет корня или узла с бОльшим значением.

Такая операция иногда называется shift-up или upheap (поднятие кучи).

В случае рассматриваемого примера произойдут следующие действия:

Поскольку 120>19, то элементы необходимо поменять местами.

Затем 120>100 и ключи снова меняются.

Рисунок 6 — этапы вставки элемента в кучу

**Удаление элемента из кучи с операцией downheap**

Поиск максимального узла (при max-heap) или минимального (при min-heap) прост.

Но что, если необходимо удалить максимум? Например, из данной двоичной кучи (max-heap)

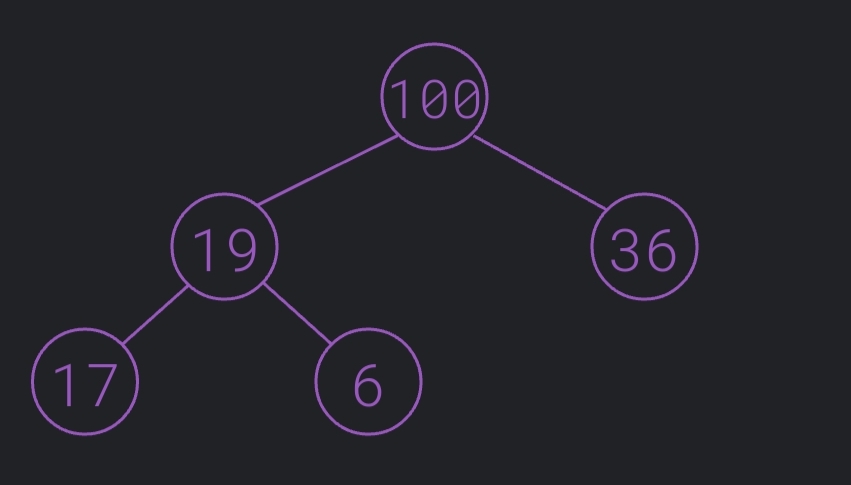


Рисунок 7 — пример кучи, удаление элемента «100»

Также как со вставкой необходимо соблюсти основные правила кучи:

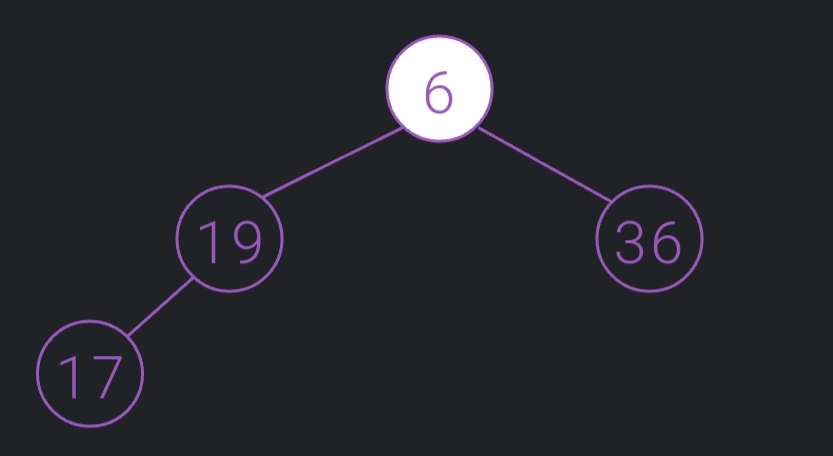
1) Дерево является полным бинарным деревом

2) Ключ родительского узла всегда больше или равен ключу вставляемого элемента.

В соответствии со свойством (1) при удалении корня мы перемещаем на его место последний элемент. В случае примера, это значение 6, после удаления 100.

Для соблюдения (2) свойства, как в случае вставки мы будем просматривать узлы дерева до тех пор, пока перестроенные данные не будут правильно упорядочены.

Так что после удаления 100 и поднятия 6, куча будет выглядеть так:

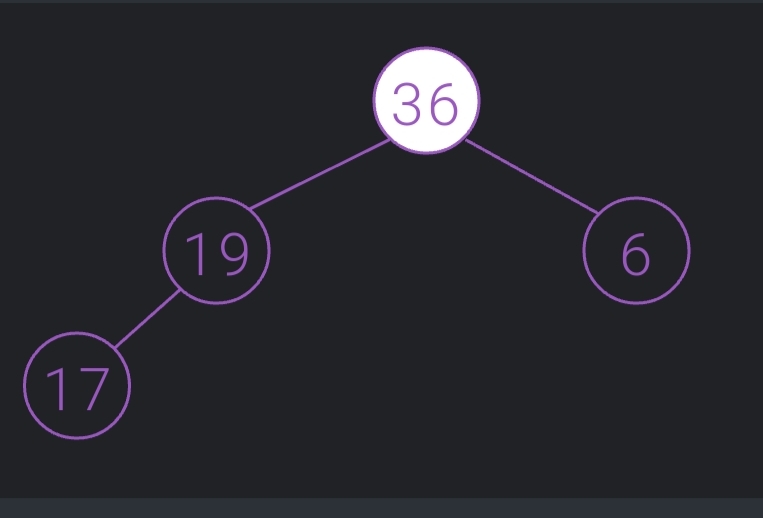
Рисунок 8 — этапы удаления элемента из кучи

Дерево вновь не соответствует свойству (2), поскольку 6 меньше обоих потомков.

Для проверки на соответствие свойству кучи мы должны сравнить 6 со значениями в узлах-потомках. И поменять 6 с максимальным из потомков.

Обмены производятся до тех пор, пока значение 6 не станет больше значений потомков.

Такая операция называется shift-heap или downheap. В случае рассматриваемого примера 6 <19 и 6<36, поэтому 6 меняем местами с 36. В данном случае, этого достаточно.

Рисунок 9 — окончание удаление элемента из кучи

Почему вставка и удаление элементов двоичной кучи составляет трудоемкость O(log(n))?

1. Двоичное дерево полное

2. Соответственно, для дерева с n узлами, высота дерева будет составлять log n

3. Операции upheap и downheap производят один обмен для каждого уровня, поэтому, если у дерева высота H, то необходимо H обменов.

Вместе эти свойства, после упрощения дают трудоемкость log(n)

**3. Двоичное дерево выражений**

Двоичное дерево выражений – это особый вид двоичных деревьев, которые хранят представление алгебраических (2+3\*5) или булевых выражений (f>v|u).

Чаще всего такие деревья используются к контексте, баз данных или любого другого потребителя, которому нужна строка, которая, возможно, будет пересылаться по сети.

Например, из бинарного дерева выражений строится SQL-запрос и отправляется серверу.

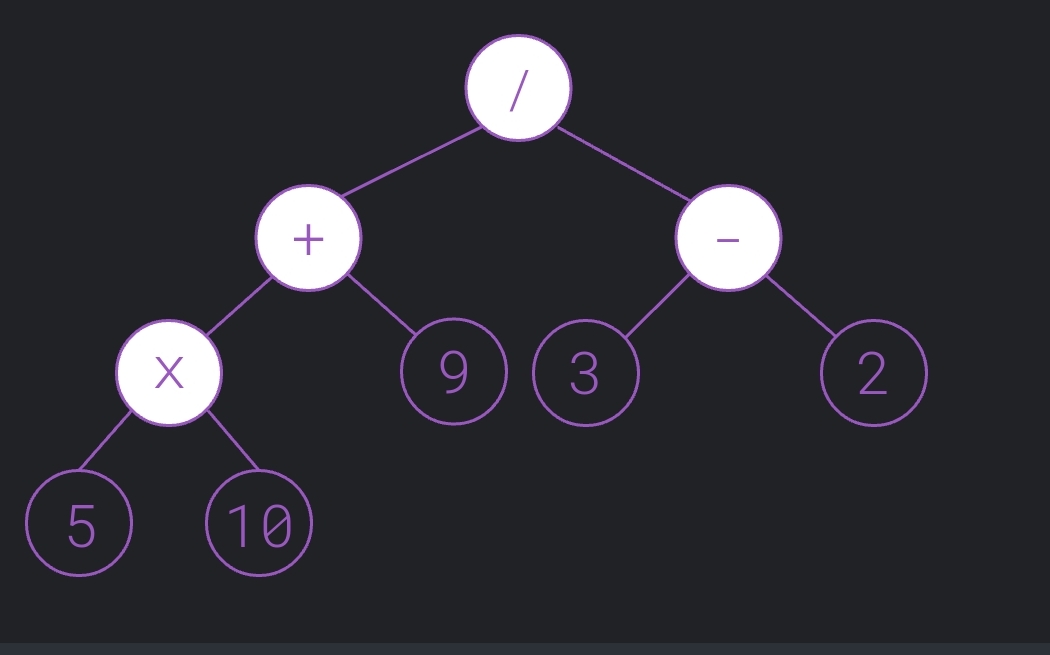
Каждый узел может иметь ноль, один или два потомка.

Листьями такого дерева являются *операнды*, а внутренними узлами – *операторы.*

Если у узла есть один потомок, то такой оператор воспринимается как унарный, например унарный минус или логическое отрицание.

Бинарное дерево выражений на выражения:

(5\*10+9)/(3-2)

Рисунок 10 — пример дерева выражений

**Симметричный обход** дерева вернет инфиксную нотацию (при этом каждое поддерево должно быть заключено в скобки):

(5\*10+9)/(3-2)

**Обход с предзаходом** вернет префиксную нотацию (Polish notation)

/+\* 5 10 9 – 3 2

**Обход с пост-заходом** вернет постфиксную нотацию (Revers Polish notation).

5 10 \* 9 +3 2 - /

Постфиксная нотация используется в языках на основе стека и системах конвейеров, например в Unix конвейерах (pipeline).

Для калькуляторов основным преимуществом данной нотации является то, что вычисление потребует меньше операций для анализа выражений, поскольку нет необходимости анализировать скобки.

*Алгоритм вычисления выражения по RPN:*

*1. Начиная с самого левого токена, до тех пор пока есть токены слева:*

*1.1 Прочитать следующий токен*

*1.2 Если токеном является значение – поместить его в стек (push)*

*1.3. Если токеном является оператор, извлечь (pop) n значений (в зависимости от оператора) из стека, подсчитать выражение и вернуть результат обратно в стек. Вернуть ошибку, если в стеке нет достаточного количества значений.*

*2. Если остается более одного значения слева в стеке после чтения всего входа, то вернуть ошибку.*

*3. иначе вернуть значение.*

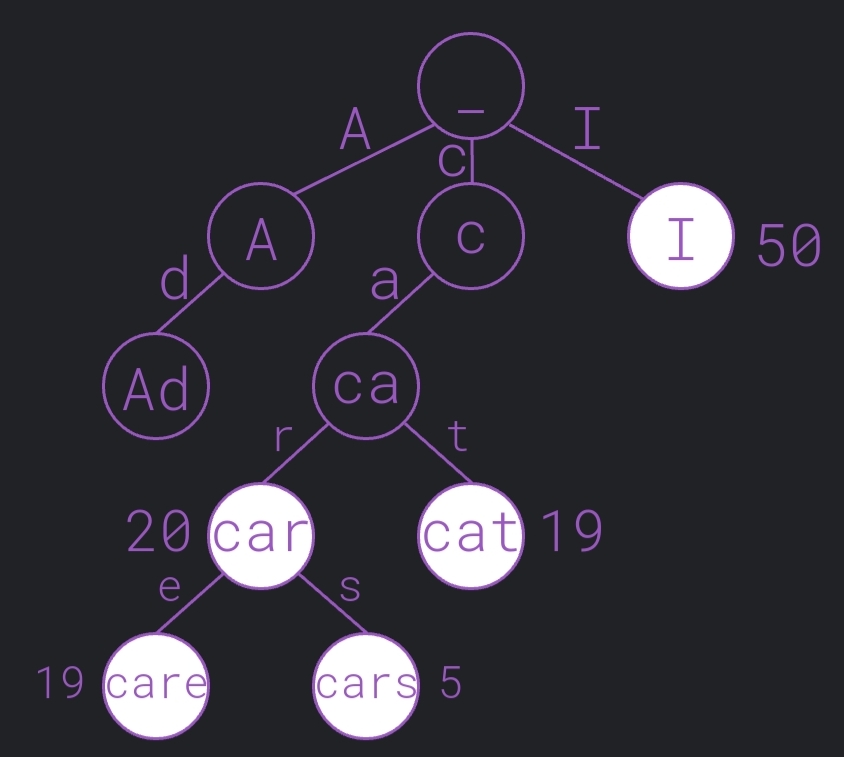
**4. Trie (Префиксное дерево)**

Префиксное дерево (нагруженное дерево, trie,prefix tree, radix tree) — упорядоченная древовидная структура, позволяющая хранить ассоциативный массив, ключами которого являются строки.

Trie хранит строки. Максимальное количество потомков узла равно размеру алфавита. Структура поддерживает поиск, вставку и удаление за время O(L), где L — длина ключа.

В отличие от двоичного дерева поиска, элементы trie не являются независимыми сущностями, а скорее последовательностями состояний, следующими друг за другом.

Лучшим примером использования дерева trie является автоподстановка текста (например, в смартфонах): узлом входа является пустой символ (пробел), затем алгоритм дает предложения слов (представленных как листья), которые можно получить, расширяя текущую последовательность символов (пусть между текущим узлом и листом).

Рисунок 12 — пример Trie

Узел имеющий связное значение, представляет собой слово с более высокой или меньшей вероятностью, которое может быть выбрано функцией автозаполнения.

Поиск данных в trie осуществляется очень быстро: худший случай - O(m), где m – это длина строки, также нет необходимости беспокоиться о двух разных словах, заканчивающихся в одном месте (коллизия - столкновение).

Недостатком trie является то, что попытки подобрать слово могут занимать много места, потому что они сохраняют каждый промежуточный шаг.

Также трудно предотвратить добавление длинных бессмысленных последовательностей символов.

Trie также называются префиксными деревьями, поскольку при обходе дерева с предзаходом, узлы посещаются в лексографическом порядке.

Почему используют Trie?

1. С использованием Trie, мы можем находить и вставлять строки за время  O(L) , где L длина слова. Это значительно быстрее алгоритма BST.
2. Другим достоинством Trie является простота печати слов в алфавитном порядке.
3. Также с помощью Trie можно реализовать эффективный поиск и автоподстановку

Недостатком деревьев Trie является требование большого количества памяти для хранения строк. Для каждого узла необходимо хранить указатели, равные числу букв алфавита.

**Порядок выполнения лабораторной работы:**

1. Выполните задания в соответствии со своим вариантом.
2. Оформите отчет по лабораторной работе.
3. Ответьте на контрольные вопросы и защитите работу.

**Требования к оформлению лабораторной работы и отчета:**

1. Работа выполняется в форме консольного приложения
2. **Модуль программы** должен предваряться комментарием с текстом исходной задачи, имена переменных и компонентов должны быть понятны стороннему пользователю или сопровождаться комментарием, объясняющим их смысл. Имена проекта и модуля должны иметь осмысленные названия.
3. **Отчет по лабораторной работе** должен содержать: титульный лист, текст задания, описание формата разработанного типа данных, таблицы или графики с полученными оценками трудоёмкости для худшего и среднего случаев построения дерева, сравнительный анализ теоретических и экспериментальных оценок трудоёмкости основных операций дерева, листинг самой программы с комментариями.

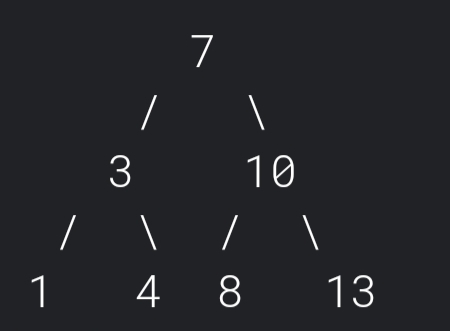
**Контрольные вопросы:**

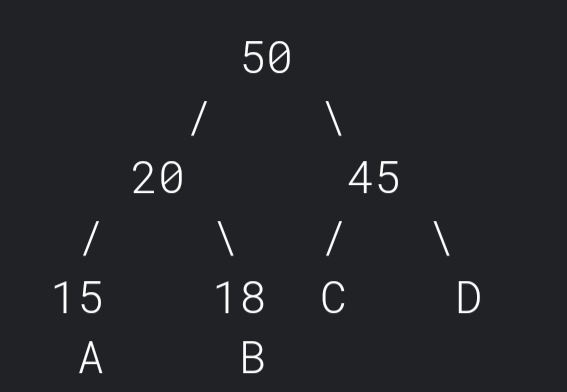
1. Каким методом вы бы воспользовались, чтобы найти слово в словаре?
2. Определите долю накладных расходов (отношение объема памяти, занимаемого данными, к общему объему памяти, отведенному под структуру) для каждой из следующих реализаций двоичного дерева с n узлами:

1. все узлы содержат данные, два указателя на дочерние узлы и указатель на родительский узел. Как поле данных, так и каждый указатель занимает 4 байта.

2. только листья содержат данные, внутренние узлы содержат два указателя на дочерние узлы. Поле данных занимает 4 байта, а каждый указатель 2 байта.

1. сколько проверок потребуется, чтобы вставить 9 в дерево:

4) В какое место кучи будет вставлен следующий узел?



**Задание**

1. Спроектировать, реализовать и провести тестовые испытания типа дерева в соответствии с вариантом задания. Элементов в дереве должно быть не менее 10ти.

2.Выполнить сравнительное тестирование трудоёмкости операций вставки, удаления и поиска для средней (случайной) и вырожденной (худший случай) структуры дерева.

|  |  |
| --- | --- |
| **Вариант 1** | |
| **Тип дерева** | BST |
| **Исходные данные** | Целое число |
| **Операции интерфейса**  **общие** | конструктор,  конструктор с параметром,  проверка, является ли дерево BST  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  включение данных с заданным ключом,  удаление данных с заданным ключом |
|  | **Конструктор дерева в прямом порядке**  Например, для набора узлов {10, 5, 1, 7, 40, 50}, должно быть построено следующее дерево.  10  / \  5 40  / \ \  1 7 50  **вывод структуры дерева на экран (обход в прямом порядке)** |

|  |  |
| --- | --- |
| **Вариант 2** | |
| **Тип дерева** | BST |
| **Исходные данные** | Символ |
| **Операции интерфейса**  **общие** | конструктор,  конструктор с параметром,  проверка, является ли дерево BST  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  включение данных с заданным ключом,  удаление данных с заданным ключом |
|  | **Проверить, содержат ли два BST-дерева одинаковые элементы**  пример функции для тестирования (для целых чисел):  **int** main()  {      // первое BST      Node\* root1 = newNode(15);      root1->left = newNode(10);      root1->right = newNode(20);      root1->left->left = newNode(5);      root1->left->right = newNode(12);      root1->right->right = newNode(25);        // второе BST      Node\* root2 = newNode(15);      root2->left = newNode(12);      root2->right = newNode(20);      root2->left->left = newNode(5);      root2->left->left->right = newNode(10);      root2->right->right = newNode(25);        // проверка на одинаковые узлы  **if** (checkBSTs(root1, root2))          cout << "YES";  **else**          cout << "NO";  **return** 0;  }  **вывод структуры дерева на экран (обход в обратном порядке)** |

|  |  |
| --- | --- |
| **Вариант 3** | |
| **Тип дерева** | BST |
| **Исходные данные** | Строка из двух символов |
| **Операции интерфейса**  **общие** | конструктор,  конструктор с параметром,  проверка, является ли дерево BST  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  включение данных с заданным ключом,  удаление данных с заданным ключом |
|  | **Поиск в BST числа, больше заданного. Вводится дерево и число N. Задача найти число в BST дереве большее либо равное N. Если число не найдено, то вернуть -1.**  **вывод структуры дерева на экран (обход в симметричном порядке)** |

|  |  |
| --- | --- |
| **Вариант 4** | |
| **Тип дерева** | BST |
| **Исходные данные** | Целое число |
| **Операции интерфейса**  **общие** | конструктор,  конструктор с параметром,  проверка, является ли дерево BST  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  включение данных с заданным ключом,  удаление данных с заданным ключом |
|  | На вход подается отсортированный массив. Написать функцию, которая создает BST -дерево.  ПРимеры:  Вход: {1, 2, 3}  Выход:  2  / \  1 3  Вход: {1, 2, 3, 4}  Выход:  3  / \  2 4  /  1    **вывод структуры дерева на экран**  **проверка сбалансированности дерева** |

|  |  |
| --- | --- |
| **Вариант 5** | |
| **Тип дерева** | BST |
| **Исходные данные** | Символ |
| **Операции интерфейса**  **общие** | конструктор,  конструктор с параметром,  проверка, является ли дерево BST  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  включение данных с заданным ключом,  удаление данных с заданным ключом |
|  | **вывод структуры дерева на экран (прямой порядок)**  **проверка сбалансированности дерева** |

|  |  |
| --- | --- |
| **Вариант 6** | |
| **Тип дерева** | Дерево выражений |
| **Исходные данные** | Выражение в виде строки инфиксной нотации |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли узел оператором  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  включение данных с заданным ключом, |
|  | **вывод структуры дерева на экран (постфиксная нотация)**  **вычисление выражения** |

|  |  |
| --- | --- |
| **Вариант 7** | |
| **Тип дерева** | Дерево выражений |
| **Исходные данные** | Выражение в виде последовательности символов префиксной нотации |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли узел оператором  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  включение данных с заданным ключом, |
|  | **вывод структуры дерева на экран (инфиксная нотация)**  **вычисление выражения** |

|  |  |
| --- | --- |
| **Вариант 8** | |
| **Тип дерева** | Дерево выражений |
| **Исходные данные** | Выражение в виде строки префиксной нотации |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли узел оператором  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  включение данных с заданным ключом, |
|  | **вывод структуры дерева на экран (постфиксная нотация)**  **проверка, является ли дерево, деревом выражений** |

|  |  |
| --- | --- |
| **Вариант 9** | |
| **Тип дерева** | Дерево выражений |
| **Исходные данные** | Выражение в виде последовательности символов в инфиксной нотации |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли узел оператором  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  включение данных с заданным ключом,  удаление данных с заданным ключом |
|  | **вывод структуры дерева на экран (постфиксная нотация)**  **вычисление выражения** |

|  |  |
| --- | --- |
| **Вариант 10** | |
| **Тип дерева** | Двоичная куча Max-heap |
| **Исходные данные** | Целое число |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли дерево кучей  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  добавить элемент в кучу,  исключить максимальный элемент из кучи  изменить значение элемента кучи по индексу |
|  | **Превратить неупорядоченный массив элементов в кучу**  **вывод структуры дерева на экран (обход в симметричном порядке)** |

|  |  |
| --- | --- |
| **Вариант 11** | |
| **Тип дерева** | Двоичная куча Max-heap |
| **Исходные данные** | Целое число |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли дерево кучей  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  добавить элемент в кучу,  исключить максимальный элемент из кучи  изменить значение элемента кучи по индексу |
|  | **Отсортировать массив путём превращения его в кучу, а кучу в отсортированный массив.**  **вывод структуры дерева на экран (обход в прямом порядке)** |

|  |  |
| --- | --- |
| **Вариант 12** | |
| **Тип дерева** | Двоичная куча Min-heap |
| **Исходные данные** | Целое число |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли дерево кучей  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  добавить элемент в кучу,  исключить максимальный элемент из кучи  изменить значение элемента кучи по индексу |
|  | **Отсортировать массив путём превращения его в кучу, а кучу в отсортированный массив.**  **вывод структуры дерева на экран (обход в обратном порядке)** |

|  |  |
| --- | --- |
| **Вариант 13** | |
| **Тип дерева** | Двоичная куча Min-heap |
| **Исходные данные** | Целое число |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли дерево кучей  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  добавить элемент в кучу,  исключить максимальный элемент из кучи  изменить значение элемента кучи по индексу |
|  | **Превратить неупорядоченный массив элементов в кучу**  **вывод структуры дерева на экран (обход в симметричном порядке)** |

|  |  |
| --- | --- |
| **Вариант 14** | |
| **Тип дерева** | Префиксное дерево |
| **Исходные данные** | Набор строк |
| **Интерфейс** | конструктор,  конструктор с параметром,  вставка элемента  поиск строки |
|  | Вывод дерева  Вход: root  / \ \  t a b  | | |  h n y  | | \ |  e s y e  / | |  i r w  | | |  r e e  |  r  Выход: Содержимое Trie:  answer  any  bye  their  there |

|  |  |
| --- | --- |
| **Вариант 15** | |
| **Тип дерева** | Префиксное дерево |
| **Исходные данные** | Набор строк |
| **Интерфейс** | конструктор,  конструктор с параметром,  вставка элемента  поиск строки |
|  | На вход подается массив строк, требуется их распечатать в алфавитном порядке, используя Trie, если обнаружатся дубликаты, то их распечатать один раз  Пример:  Вход : "abc", "xy", "bcd"  Выход : abc bcd xy |

|  |  |
| --- | --- |
| **Вариант 16** | |
| **Тип дерева** | Двоичная куча Max-heap |
| **Исходные данные** | Целое число |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли дерево кучей  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  добавить элемент в кучу,  исключить максимальный элемент из кучи  изменить значение элемента кучи по индексу |
|  | **Превратить неупорядоченный массив элементов в кучу**  **вывод структуры дерева на экран (обход в обратном порядке)** |

|  |  |
| --- | --- |
| **Вариант 17** | |
| **Тип дерева** | Двоичная куча Min-heap |
| **Исходные данные** | Целое число |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли дерево кучей  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  добавить элемент в кучу,  исключить максимальный элемент из кучи  изменить значение элемента кучи по индексу |
|  | **Превратить неупорядоченный массив элементов в кучу**  **вывод структуры дерева на экран (обход в прямом порядке)** |

|  |  |
| --- | --- |
| **Вариант 18** | |
| **Тип дерева** | Префиксное дерево |
| **Исходные данные** | Набор строк |
| **Интерфейс** | конструктор,  конструктор с параметром,  вставка элемента  поиск строки |
|  | Реализовать функцию автоподбора слов, основываясь на Префиксном дереве.  Пример:  Если в дереве Trie хранятся строки {“abc”, “abcd”, “aa”, “abbbaba”} и пользователь пишет “ab” , то функция должна ему показать {“abc”, “abcd”, “abbbaba”}. |

|  |  |
| --- | --- |
| **Вариант 19** | |
| **Тип дерева** | Префиксное дерево |
| **Исходные данные** | Набор строк |
| **Интерфейс** | конструктор,  конструктор с параметром,  вставка элемента  поиск строки |
|  | Функция удаления узлов (строки) из trie.  Пример:  root  / \ \  t a b  | | |  h n y  | | \ |  e s y e  / | |  i r w  | | |  r e e  |  r  search(root, "bye") ? cout << "Yes\n" : cout << "No\n";  **remove**(root, "bye");  search(root, "bye") ? cout << "Yes\n" : cout << "No\n";  Выход:  Yes  No |

|  |  |
| --- | --- |
| **Вариант 20** | |
| **Тип дерева** | Префиксное дерево |
| **Исходные данные** | Набор строк |
| **Интерфейс** | конструктор,  конструктор с параметром,  вставка элемента  поиск строки |
|  | Загрузить в префиксное дерево текст из файла  Реализовать функцию автоподбора слов, основываясь на Префиксном дереве. |

|  |  |
| --- | --- |
| **Вариант 21** | |
| **Тип дерева** | BST |
| **Исходные данные** | Строки |
| **Операции интерфейса**  **общие** | конструктор,  конструктор с параметром,  проверка дерева на пустоту  опрос размера дерева  включение данных с заданным ключом,  удаление данных с заданным ключом |
|  | **Конструктор дерева в прямом порядке**  **вывод структуры дерева на экран (обход в симметричном порядке порядке)** |

|  |  |
| --- | --- |
| **Вариант 22** | |
| **Тип дерева** | BST |
| **Исходные данные** | Целое число |
| **Интерфейс** | конструктор,  конструктор с параметром,  проверка, является ли дерево BST  вывод структуры дерева на экран  проверка дерева на пустоту  опрос размера дерева  очистка дерева,  включение данных с заданным ключом,  удаление данных с заданным ключом |
|  | Для полученного дерева выведите список всех листьев (вершин, не имеющих потомков) в порядке возрастания. |

**Примеры определения древовидных структур**

**1) BST для типа данных int**

**class** BST

{

**int** data;

    BST \*left, \*right;

**public**:

    // конструктор по-умолчанию

    BST();

    // параметрический конструктор

    BST(**int**);

    // Функция вставки элемента

    BST\* Insert(BST\*, **int**);

    // Inorder traversal.

**void** Inorder(BST\*);

//…..

};

Вариант main:

**int** main()

{

    BST b, \*root = NULL;

    root = b.Insert(root, 50);

    b.Insert(root, 30);

    b.Insert(root, 20);

    b.Insert(root, 40);

    b.Insert(root, 70);

    b.Insert(root, 60);

    b.Insert(root, 80);

    b.Inorder(root);

**return** 0;

}

Вывод:

20

30

40

50

60

70

80

**2) Дерево выражений**

**struct** et

{

**char** value;

    et\* left, \*right;

};

 ИЛИ

**class** node

{

**public**:

    string info;

    node \*left = NULL, \*right = NULL;

    node(string x)

    {

        info = x;

    }

};

**пример вызова функций:**

**int** main()

{

**char** postfix[] = "ab+ef\*g\*-"; //ввода строки в постфиксной нотации

    et\* r = constructTree(postfix);

**printf**("infix expression is \n");

    inorder(r);

**return** 0;

}

**int** main()

{

    // создание дерева выражений посимвольно

    node \*root = **new** node("+");

    root->left = **new** node("\*");

    root->left->left = **new** node("5");

    root->left->right = **new** node("-4");

    root->right = **new** node("-");

    root->right->left = **new** node("100");

    root->right->right = **new** node("20");

    cout << eval(root) << endl;

    cout << eval(root);

**return** 0;

}

**3) Двоичная куча**

// пример класса Min Heap

**class** MinHeap

{

**int** \*harr; // указатель на массив элементов в куче

**int** capacity; // максимальная вместимость кучи

**int** heap\_size; // текущее количество элементов в куче

**public**:

    // конструктор

    MinHeap(**int** capacity);

    // вернуть родителя

**int** parent(**int** i) { **return** (i-1)/2; }

  // вернуть левого потомка

**int** left(**int** i) { **return** (2\*i + 1); }

  // вернуть правого потомка

**int** right(**int** i) { **return** (2\*i + 2); }

    // найти и удалить минимум

**int** extractMin();

    // вернуть минимум

**int** getMin() { **return** harr[0]; }

    // удалить узел по индексу i

**void** deleteKey(**int** i);

    // вставить узел по значению

**void** insertKey(**int** k);

};

**4) Пример определения структуры для Trie**

**const** **int** ALPHABET\_SIZE = 26;

// trie

**struct** TrieNode

{

**struct** TrieNode \*children[ALPHABET\_SIZE];

    // isEndOfWord если true, то конец слова

**bool** isEndOfWord;

};