Лабораторная работа №14
Тема: «Бинарное дерево»

Цель работы: сформировать знания и умения по работе с подпрограммами, приобрести навыки написания программ с использованием бинарных деревьев.

Время выполнения: 4 часа.

Теоретические сведения

Двоичное дерево в первую очередь дерево. В программировании – структура данных, которая имеет корень и дочерние узлы, без циклических связей. Если рассмотреть отдельно любой узел с дочерними элементами, то получится тоже дерево. Узел называется внутренним, если имеет хотя бы одно поддерево. Самые нижние элементы, которые не имеют дочерних элементов, называются *листами или листовыми узлами*.

В узлах может храниться любая информация, от примитивных типов до объектов. Кроме такого подхода, возможен альтернативный подход для двоичного дерева — хранение в массиве.

Первая особенность двоичного дерева, что любой узел не может иметь более двух детей. Их называют просто — левый и правый потомок, или левое и правое поддерево.

Вторая особенность двоичного дерева, и основное правило его построения, заключается в том, что левый потомок меньше текущего узла, а правый потомок больше. Отношение больше/меньше имеет смысл для сравниваемых объектов, например, числа, строки, если в дереве содержатся сложные объекты, то для них берётся какая-нибудь процедура сравнения, и она будет отрабатывать при всех операциях работы с деревом.

Напишем класс для создания двоичного дерева:

```
// дополнительный класс для хранения информации узла
class BinaryTreeItem {
  constructor(itemValue) {
    this.value = itemValue;
    this.left = null;
    this.right = null;
  }
}

const elementExistMessage =
  "The element has already in the tree";
```

```
class BinaryTree {
  // в начале работы дерево пустое, root отсутствует
  constructor() {
    this.root = null;
  }

  insertItem(newItem) {
    // создание нового узла дерева
    const newNode = new BinaryTreeItem(newItem);

    // проверка на пустой root, если пустой, то заполняем
    // и завершаем работу
    if (this.root === null) {
      this.root = newNode;
      return;
    }

    // вызов рекурсивного добавления узла
    this._insertItem(this.root, newNode);
  }

  _insertItem(currentNode, newNode) {
    // если значение в добавляемом узле
    // меньше текущего рассматриваемого узла
    if (newNode.value < currentNode.value) {
      // если меньше и левое поддерево отсутствует
      // то добавляем
      if (currentNode.left === null) {
        currentNode.left = newNode;
      } else {
        // если левое поддерево существует,
        // то вызываем для этого поддерева
        // процедуру добавления нового узла
        this._insertItem(currentNode.left, newNode);
      }
    }

    // для правого поддерева алгоритм аналогичен
    // работе с левым поддеревом, кроме операции сравнения
```

```
    if (newNode.value > currentNode.value) {
      if (currentNode.right === null) {
        currentNode.right = newNode;
      } else {
        this._insertItem(currentNode.right, newNode);
      }
    }

    // если элемент равен текущему элементу,
    // то можно реагировать по разному, например просто
    // вывести предупреждение
    // возможно стоит добавить проверку на NaN,
    // зависит от потребностей пользователей класса
    if (newNode.value === currentNode.value) {
      console.warn(elementExistMessage);
    }
  }
}

const binaryTree = new BinaryTree();

binaryTree.insertItem(3);
binaryTree.insertItem(1);
binaryTree.insertItem(6);
binaryTree.insertItem(4);
binaryTree.insertItem(8);
binaryTree.insertItem(-1);
binaryTree.insertItem(3.5);
console.log(binaryTree);
```

Обход

Рассмотрим несколько алгоритмов обхода/поиска элементов в двоичном дереве.

Мы можем спускаться по дереву, в каждом из узлов есть выбор куда можем пойти в первую очередь и какой из элементов обработать сначала: левое поддерево, корень или право поддерево. Такие варианты обхода называются обходы в глубину (depth first).

Какие возможны варианты обхода (слово поддерево опустим):

корень, левое, правое (preorder, прямой);
корень, правое, левое;
левое, корень, правое (inorder, симметричный, центрированный);
левое, правое, корень (postorder, обратный);
правое, корень, левое;
правое, левое, корень.

Также используется вариант для обхода деревьев по уровням. Уровень в дереве — его удалённость от корня. Сначала обходится корень, после этого узлы первого уровня и так далее. Называется обход в ширину, по уровням, breadth first, BFS — breadth first search или level order traversal.

Выбирается один из этих вариантов, и делается обход, в каждом из узлов применяя выбранную стратегию.

Обычно для обходов в глубину применяется рекурсия. Реализуем один из вариантов, например симметричный: левое поддерево, корень, правое поддерево.

При этом мы обработаем первым самый левый узел, где левое поддерево окажется пустым, но правое может присутствовать. То есть в каждом из узлов будем спускаться ниже и ниже, пока левое поддерево не окажется пустым.

```
class BinaryTreeItem {
  constructor(itemValue) {
    this.value = itemValue;
    this.left = null;
    this.right = null;
  }
}

const elementExistMessage =
  "The element has already in the tree";

class BinaryTree {
  constructor() {
    this.root = null;
  }

  insertItem(newItem) {
    // .....
```

```
  }

  inorder(handlerFunction) {
    // просто вызываем функцию с другими параметрами,
    // добавляя текущий обрабатываемый узел
    // в рекурсивные вызов
    this._inorderInternal(this.root, handlerFunction);
  }

  _insertItem(currentNode, newNode) {
    // .....
  }

  _inorderInternal(currentNode, handlerFunction) {
    // если узла нет, то его обрабатывать не нужно
    if (currentNode === null) return;

    // порядок обхода, для каждого из поддеревьев:
    // 1. проваливаемся в левое поддерево
    // 2. вызываем обрабатывающую функцию
    // 3. проваливаемся в правое поддерево
    this._inorderInternal(currentNode.left,
      handlerFunction);
    handlerFunction(currentNode.value);
    this._inorderInternal(currentNode.right,
      handlerFunction);
  }
}

const binaryTree = new BinaryTree();
binaryTree.insertItem(3);
binaryTree.insertItem(1);
binaryTree.insertItem(6);
binaryTree.insertItem(4);
binaryTree.insertItem(8);
binaryTree.insertItem(-1);
binaryTree.insertItem(3.5);

binaryTree.inorder(console.log);
```

```
// вызов inorder(console.log) выведет
// -1
// 1
// 3
// 3.5
// 4
// 6
// 8
```

Для реализации других вариантов обхода просто меняем порядок вызова функций в функции _inorderInternal. И нужно не забыть переименовать функцию, чтобы название соответствовало содержимому.

Рассмотрим inorder алгоритм обхода на примере дерева, созданного в предыдущем блоке кода.

```
// 1
this._inorderInternal(currentNode.left, handlerFunction);
// 2
handlerFunction(currentNode.value);
// 3
this._inorderInternal(currentNode.right, handlerFunction);
```

Сначала мы спустимся в самое левое поддерево — узел -1. Зайдем в его левое поддерево, которого нет, первая конструкция выполнится, ничего не сделав внутри функции. Вызовется обработчик handlerFunction, на узле -1. После этого произойдёт попытка войти в правое поддерево, которого нет. Работа функции для узла -1 завершится.

В вызов для узла -1 мы пришли через вызов функции _inorderInternal для левого поддерева узла 1. Вызов для левого поддерева -1 завершился, вызовется обработчик для значения узла 1, после этого — для правого поддерева. Правого поддерева нет, функция для узла 1 заканчивает работу. Выходим в обработчик для корня дерева.

Для корня дерева левое поддерево полностью отработало, происходит переход ко второй строке процедуры обхода — вызов обработчика значения узла. После чего вызов функции для обработчика правого поддерева.

Аналогично продолжая рассуждения, и запоминая на какой строке для определенного узла мы вошли в рекурсивный вызов, можем пройти алгоритм «руками», лучше понимая его работу.

Для обходов в ширину используется дополнительный массив.

```
class BinaryTreeItem {
  constructor(itemValue) {
    this.value = itemValue;
    this.left = null;
    this.right = null;
  }
}

const elementExistMessage =
  "The element has already in the tree";

class BinaryTree {
  constructor() {
    this.root = null;
  }

  insertItem(newItem) {
    // .....
  }

  breadthFirstHandler(handlerFunction) {
    if (this.root === null) return;

    // массив, в который будем добавлять элементы,
    // по мере спускания по дереву
    const queue = [this.root];
    // используем позицию в массиве для текущего
    // обрабатываемого элемента
    let queuePosition = 0;

    // можем убирать обработанные элементы из очереди
    // например функцией shift
    // для обработки всегда брать нулевой элемент
    // и завершать работу, когда массив пуст
```

```
    // но shift работает за линейное время, что увеличивает
    // скорость работы алгоритма
    // while (queue.length > 0) {
    //   const currentNode = queue.shift();

    while (queuePosition < queue.length) {
      // текущий обрабатываемый элемент в queuePosition
      const currentNode = queue[queuePosition];
      handlerFunction(currentNode.value);

      // добавляем в список для обработки дочерние узлы
      if (currentNode.left !== null) {
        queue.push(currentNode.left);
      }
      if (currentNode.right !== null) {
        queue.push(currentNode.right);
      }

      queuePosition++;
    }
  }

  _insertItem(currentNode, newNode) {
    // ......
  }
}

const binaryTree = new BinaryTree();
binaryTree.insertItem(3);
binaryTree.insertItem(1);
binaryTree.insertItem(6);
binaryTree.insertItem(4);
binaryTree.insertItem(8);
binaryTree.insertItem(-1);
binaryTree.insertItem(3.5);

binaryTree.breadthFirstHandler(console.log);
// вызов breadthFirstHandler(console.log) выведет
// 3 корень
// 1 узлы первого уровня
```

```
// 6
// -1 узлы второго уровня
// 4
// 8
// 3.5 узел третьего уровня
```

Поиск

Операция поиска — вернуть true или false, в зависимости от того, содержится элемент в дереве или нет. Может быть реализована на основе поиска в глубину или ширину, посмотрим на реализацию на основе алгоритма обхода в глубину.

```
search(value) {
  return this._search(this.root, value);
}

_search(currentNode, value) {
  // дополнительные проверки,
  // обрабатывающие завершение поиска
  // либо проваливание в несуществующий узел
  // либо найденной значение
  if (currentNode === null) return false;
  if (currentNode.value === value) return true;

  // this._search проваливаются в дерево
  // когда поиск завершен
  // то по цепочке рекурсивных вызовов
  // будет возвращен результат
  if (value < currentNode.value) {
   return this._search(currentNode.left, value);
  }
  if (value > currentNode.value) {
   return this._search(currentNode.right, value);
  }
}
```

Функция сравнения или получение ключа

До этого мы рассматривали простые данные, для которых определена операция сравнения между ключами. Не всегда возможно реализовать сравнение таким простым образом.

Можно сделать функцию, которая будет получать ключ из данных, которые хранятся в узле.

```
class BinaryTreeItem {
  constructor(itemValue) {
    this.value = itemValue;
    this.left = null;
    this.right = null;
  }
}

const elementExistMessage =
  "The element has already in the tree";

class BinaryTree {
  // параметр при создании дерева -
  // функция получения ключа
  // ключи должны быть сравнимы
  constructor(getKey) {
    this.root = null;
    this.getKey = getKey;
  }

  insertItem(newItem) {
    const newNode = new BinaryTreeItem(newItem);

    if (this.root === null) {
      this.root = newNode;
      return;
    }

    this._insertItem(this.root, newNode);
  }

  breadthFirstHandler(handlerFunction) {
    // .....
  }

  _insertItem(currentNode, newNode) {
```

```
    // отличие во всех процедурах сравнения
    // вместо просто сравнивания value
    // перед этим применяем функцию получения ключа
    if (this.getKey(newNode.value) <
      this.getKey(currentNode.value)) {
      if (currentNode.left === null) {
        currentNode.left = newNode;
      } else {
        this._insertItem(currentNode.left, newNode);
      }
    }

    if (this.getKey(newNode.value) >
      this.getKey(currentNode.value)) {
      if (currentNode.right === null) {
        currentNode.right = newNode;
      } else {
        this._insertItem(currentNode.right, newNode);
      }
    }

    if (this.getKey(newNode.value) ===
      this.getKey(currentNode.value)) {
      console.warn(elementExistMessage);
    }
  }
}

const getKey = (element) => element.key;

const binaryTree = new BinaryTree(getKey);
binaryTree.insertItem({ key: 3 });
binaryTree.insertItem({ key: 1 });
binaryTree.insertItem({ key: 6 });
binaryTree.insertItem({ key: 4 });
binaryTree.insertItem({ key: 8 });
binaryTree.insertItem({ key: -1 });
binaryTree.insertItem({ key: 3.5 });

binaryTree.breadthFirstHandler(console.log);
```

Можно передать в конструктор специальную функцию сравнения. Эту функцию можно сделать как обычно делают функции сравнения в программировании, возвращать 0, если ключи равны. Значение больше нуля, если первый переданный объект больше второго, и меньше нуля если меньше. Важно не перепутать когда что возвращается и правильно передать параметры. Например, текущий узел, уже существующий в дереве, первым параметром, а тот, с которым производится текущая операция — вторым.

Для реализации такой возможности потребуется во всех местах сравнения использовать эту функцию

```
class BinaryTreeItem {
  constructor(itemValue) {
    this.value = itemValue;
    this.left = null;
    this.right = null;
  }
}

const elementExistMessage =
  "The element has already in the tree";

class BinaryTree {
  // в конструкторе передаем функцию сравнения
  constructor(compareFunction) {
    this.root = null;
    this.compareFunction = compareFunction;
  }

  insertItem(newItem) {
    const newNode = new BinaryTreeItem(newItem);

    if (this.root === null) {
      this.root = newNode;
      return;
    }

    this._insertItem(this.root, newNode);
  }
```

```
  breadthFirstHandler(handlerFunction) {
    // .....
  }

  _insertItem(currentNode, newNode) {
    // вместо сравнения value
    // вызываем функцию сравнения
    // и проверяем больше или меньше нуля
    // получился результат сравнения
    if (this.compareFunction(currentNode.value,
      newNode.value) > 0) {
      if (currentNode.left === null) {
        currentNode.left = newNode;
      } else {
        this._insertItem(currentNode.left, newNode);
      }
    }

    // текущий узел меньше нового,
    // значит новый узел должен быть отправлен
    // в правое поддерево
    if (this.compareFunction(currentNode.value,
      newNode.value) < 0) {
      if (currentNode.right === null) {
        currentNode.right = newNode;
      } else {
        this._insertItem(currentNode.right, newNode);
      }
    }

    if (this.compareFunction(currentNode.value,
      newNode.value) === 0) {
      console.warn(elementExistMessage);
    }
  }
}

const compare = (object1, object2) => {
  return object1.key - object2.key;
```

```
};
const binaryTree = new BinaryTree(compare);
binaryTree.insertItem({ key: 3 });
binaryTree.insertItem({ key: 1 });
binaryTree.insertItem({ key: 6 });
binaryTree.insertItem({ key: 4 });
binaryTree.insertItem({ key: 8 });
binaryTree.insertItem({ key: -1 });
binaryTree.insertItem({ key: 3.5 });

binaryTree.breadthFirstHandler(console.log);
```

Индивидуальные задания к лабораторной работе №14

Опишите класс — дерево, необходимое для решения задачи, указанной в вашем варианте задания, и реализуйте его методы.

Продемонстрируйте работу основных методов работы с деревом: построение, вывод, обход.

Составьте программу решения задачи, указанной в вашем варианте задания.

Варианты заданий

1. Найти дубликаты в списке чисел с использованием дерева поиска.

2. На основе операции обхода сверху вниз реализовать операцию, определяющую, подобны ли два бинарных дерева (два бинарных дерева подобны, если они оба пусты, либо их левые и правые поддеревья подобны).

3. По бесскобочной постфиксной записи арифметического выражения с операндами, записанными в виде строк символов, построить дерево выражения и получить полноскобочное инфиксное выражение.

4. Реализовать операцию определения уровня узла с заданным указателем в бинарном дереве, построить дерево с узлами — символами и определить минимальный и максимальный уровни листа.

5. На основе функции обхода сверху вниз реализовать операцию поиска узла с заданным значением в дереве, не являющемся деревом поиска. Построить дерево минимальной высоты с элементами-символами. Используя операции Brother и Value, найти адрес брата узла с введенным с терминала значением и вывести его значение.

6. Построить дерево поиска с элементами — строками. Используя операции Addr, Father и Value, найти узел, являющийся самым «молодым» общим предком двух заданных узлов, и вывести его значение.

7. По постфиксной записи арифметического выражения с операндами-строками построить дерево выражения и получить инфиксную запись выражения, содержащую только необходимые скобки.

8. Реализовать дерево поиска, содержащее элементы с одинаковыми ключами. Операция Addr возвращает адрес первого элемента с заданным ключом. Операция Delete исключает первый элемент с заданным ключом.

9. Построить дерево поиска с элементами — вещественными числами. Определить количество элементов дерева на каждом уровне. Удалить элементы с заданными значениями.

10. Построить дерево поиска с элементами — числами. С использованием операций Ac1c1г и Oe^eЬей найти узел с заданным значением и исключить его левое поддерево. Вывести число узлов в дереве до и после исключения.

11. На основе просмотра текста построить дерево поиска, элементами которого являются символы, а ключами — количества вхождений этих символов в текст. Исключить из дерева символы с заданным числом повторений.

12. Построить дерево поиска с элементами — строками. Исключить из дерева все узлы с заданными словами. При каждом исключении выводить текущее состояние дерева и сообщение о том, каким потомком является исключаемый узел: левым или правым.

13. На основе процедуры обхода дерева «снизу-вверх» реализовать операцию поиска узла с заданным значением в дереве, не являющемся деревом поиска. Из двух последовательностей символов построить два бинарных дерева минимальной высоты. В первом дереве найти элемент с заданным значением и подключить второе дерево в качестве его левого поддерева, если оно пусто, или левого поддерева первого из его крайних левых потомков, имеющих пустое левое поддерево.

14. На основе операции обхода сверху вниз реализовать операцию, определяющую, являются ли два бинарных дерева зеркально подобными (два бинарных дерева зеркально подобны, если они оба либо пусты, либо левое поддерево каждого дерева подобно правому дереву другого).

15. По бесскобочной постфиксной записи арифметического выражения с операндами — строками построить дерево выражения и получить полноскобочное инфиксное выражение.

16. По бесскобочной префиксной записи арифметического выражения с операндами — строками построить дерево выражения и получить выражение в инфиксной записи, содержащее только необходимые скобки.

17. Построить бинарное дерево с элементами — символами. Вывести элементы дерева по уровням.

18. Реализовать дерево выражения с операндами — числами и написать метод вычисления значения арифметического выражения. По бесскобочной префиксной записи построить дерево выражения и вычислить его значение.

19. Реализовать операцию определения уровня узла с заданным указателем в дереве поиска, построить дерево с заданными числовыми значениями узлов и определить минимальный и максимальный уровни листа.

20. На основе процедуры обхода дерева слева направо реализовать операцию поиска узла с заданным значением в дереве, не являющемся деревом поиска. Построить дерево минимальной высоты с числовыми значениями узлов. Используя необходимые операции, удалить из дерева поддерево, корнем которого является отец символа с заданным значением.

21. Построить два дерева поиска для студентов двух групп с полученными на экзаменах оценками. Сформировать дерево для студентов, получивших отличные оценки, причем расположить их в дереве по алфавиту.

22. Реализовать дерево поиска, содержащее элементы с одинаковыми ключами. Операция AddrLast возвращает адрес последнего элемента с заданным ключом. Операция DeleteFirst исключает первый элемент с заданным ключом.

23. С использованием дерева поиска удалить из заданного текста дубликаты слов.

24. Построить бинарное дерево с элементами — словами. Сформировать предложение из слов дерева, получающееся при обходе слева направо. Удалить поддерево, начинающееся с заданного слова.

25. Построить дерево поиска с элементами — целыми числами. Удалить из дерева элементы с заданным значением, а все отрицательные элементы заменить нулями.

26. Построить дерево поиска с заданными числовыми значениями. С использованием операций Addr и Father найти узел с заданным значением и удалить из дерева поиска отца этого узла.

27. Построить дерево поиска с элементами — символами. Определить число повторяющихся символов в дереве и удалить дубликаты.

28. Реализовать дерево поиска, содержащее элементы с одинаковыми ключами. Операция AddrLast возвращает адрес последнего элемента с заданным ключом. Операция DeleteAll исключает все элементы с заданным ключом.

29. На основе просмотра текста построить дерево поиска, элементами которого являются символы, а ключами — количества вхождений этих символов в текст. Исключить из дерева символы с одинаковым числом повторений, используя строку для хранения последовательности узлов дерева (результатов обхода/

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание по вариантам, представленные в теоретических сведениях, сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.

2. Индивидуальное задание из раздела «Теоретические сведения» с кодом, комментариями и скриншотами работающих программ.

3. Построение блок-схем.