**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**

**імені ІГОРЯ СІКОРСЬКОГО”**

**ЗВІТ**

**з лабораторної роботи №7**

**з навчальної дисципліни “****Рекурсивні алгоритми”**

**Тема:**

**Поняття бінарного дерева. Обхід бінарного дерева. Створення, відображення дерева. Вставлення, видалення елементів у бінарному дереві.**

**Виконав студент групи ТР–12**

Каркушевський Владислав

Лабораторну роботу захищено

з оцінкою \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Перевірив доцент кафедри**

\_\_\_\_\_\_\_\_\_\_\_ Андрій ОНИСЬКО

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2022 року

**Київ 2022**

**Мета роботи**: набути навичок створення та обробки бінарних дерев.

**Теоретична частина:**

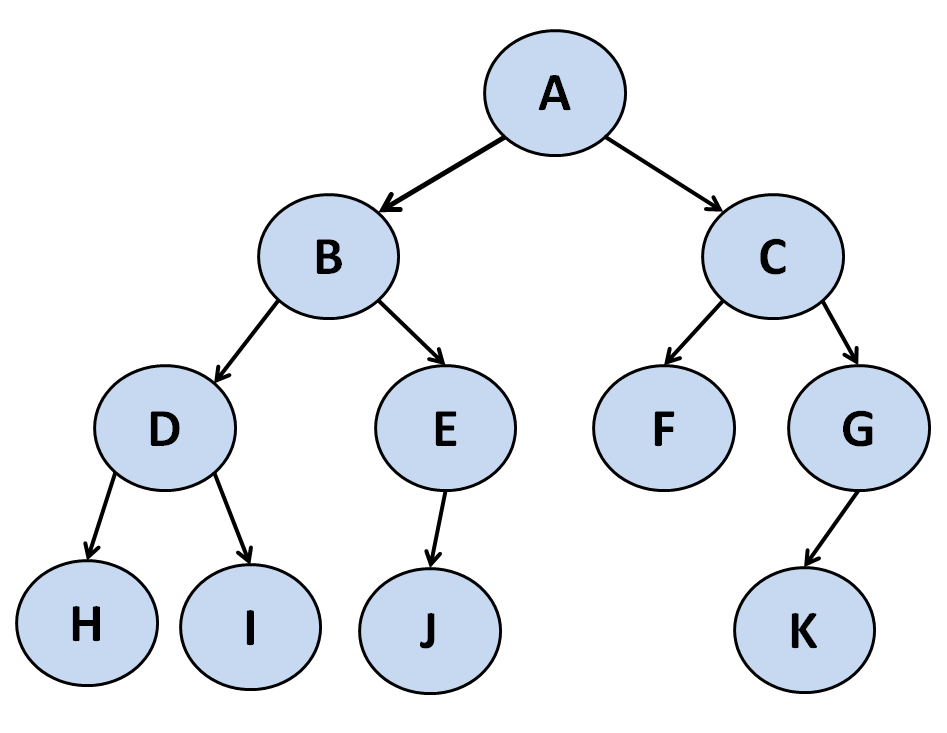
Дерево - це ієрархічна структура деякої сукупності елементів. Термін дерево вибраний не випадково , адже ця структура даних має багато спільного з деревами рослинного світу – дерево має корінь, гілки та листя. Єдиною суттєвою відмінністю є те , що дерева у програмуванні завжди схематично

зображують так , що корінь розташовується вгорі, а гілки йдуть донизу. Фактично дерево – це різновид динамічного списку

Дерево – це сукупність вузлів і відношень, що утворює ієрархічну структуру.

Вузол (або вершина) дерева – це математична абстракція, яка моделює абстрактні об’єкти чи об’єкти реального світу. Вузол, як правило, має ім’я, яке називається його ключем або ідентифікатором.

Відношення визначають зв’язки між вершинами типу батько-дитина. Ці зв’язки називаються гілками (ребрами) дерева.



Малюнок 1.1 Дерево

**Упорядкування вузлів дерева**

Дерево називається впорядкованим, якщо набір дітей кожного його вузла є впорядкований. Діти вузла впорядкованого дерева, як правило впорядковуються зліва на право, якщо інакший порядок окремо не зазначений.

Впорядкування дерев використовується для співставлення вузлів, що не пов’язані співвідношенням типу батько-син.

Основними операціями при роботі з деревами є:

- додавання елемента до дерева;

- пошук елемента дерева, що відповідає заданому критерію пошуку;

- сортування елементів дерева;

- вилучення

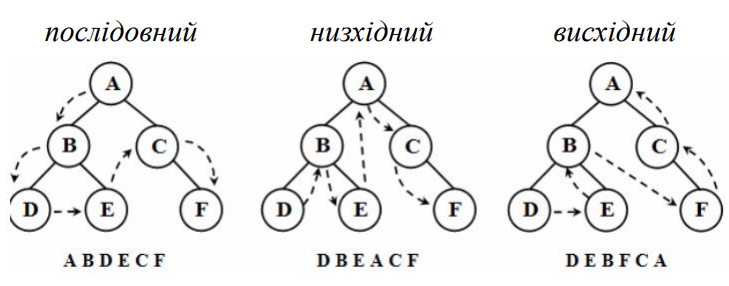
Процес доступу до елементів дерева називається проходженням дерева.

Існує три способи проходження дерев: послідовний, низхідний та висхідний.

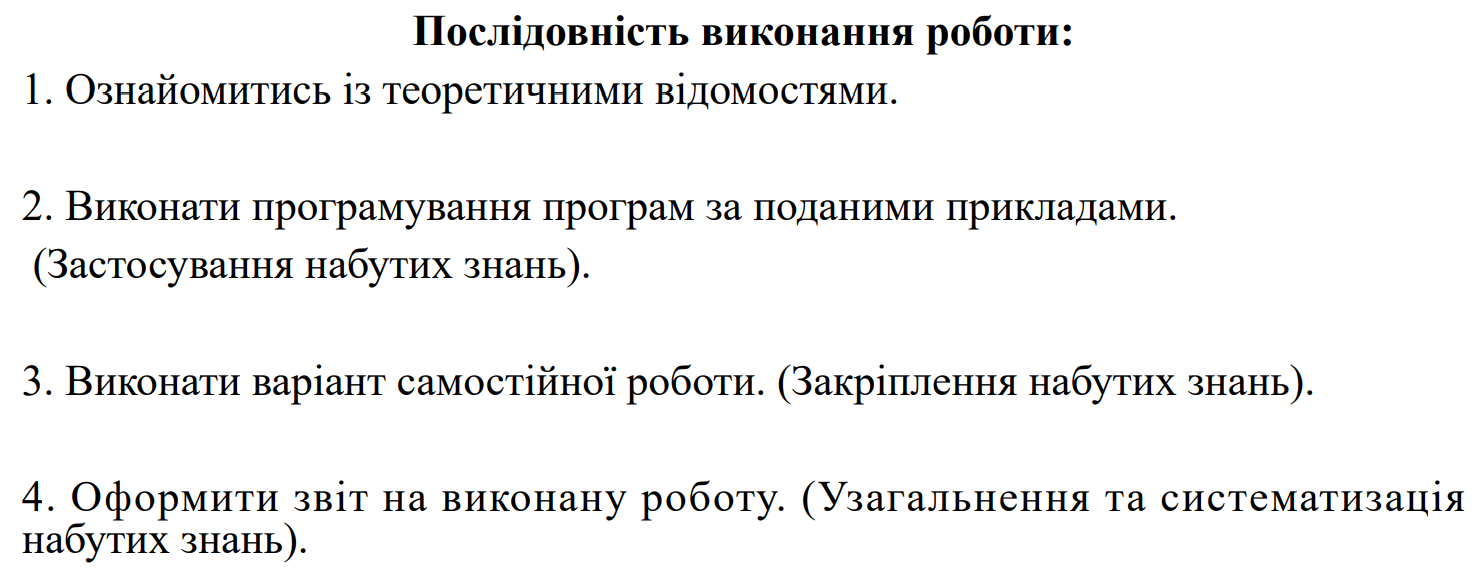
При послідовному методі: спочатку розглядається самий лівий відносно кореня елемент, потім його батьківський вузол, потім правий елемент даної ланки, потім переходять до елемента попереднього рівня, що є батьківським по відношенню до батьківського вузла даної ланки і т. д. Вверх до кореня, а потім від кореня вниз до самого правого елемента

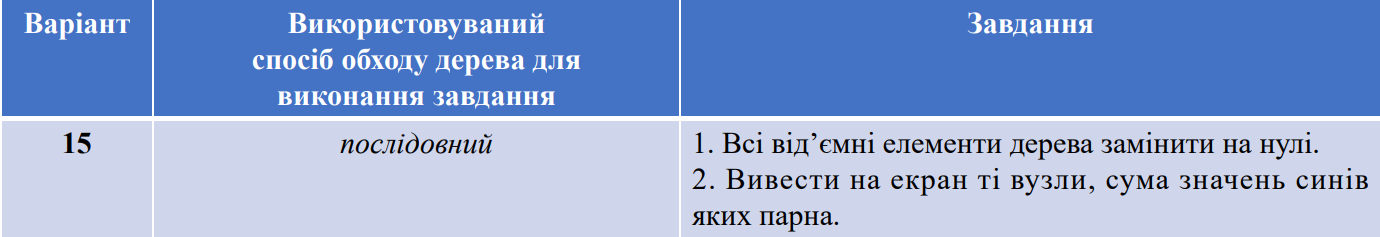
При низхідному проходженні дерева порядок обходу елементів зверху-вниз та зліва-направо. Тобто спочатку проходиться корінь, потім його лівий елемент, а потім правий і т.д.

При висхідному проходженні порядок проходження зліва-направо та знизу-вверх. Тобто спочатку проходиться лівий вузол найнижчої ланки, потім правий вузол цієї ланки, а потім їх батьківський вузол і т.д.



**Завдання**





У коді будуть використовувати Stack і DoublyLinkedList які були створенні у минулих лабораторних.

**Код програми:**

**Stack**

*import* java.util.*Iterator*;  
*import* java.util.NoSuchElementException;  
  
*public class* Stack<T> *implements Iterable*<T> {  
  
 *private* Node<T> firstNode;  
 *private int* size = 0;  
  
 *public* Stack() {*//створимо конструктор* firstNode = *new* Node<T>(*null*, *null*);  
 }  
  
 *public int* size() {*//повертає розмір стека  
 return* size;  
 }  
  
 *public* T pop() {*//Повертає елемент, що знаходиться у верхній частині стека, видаляючи його у процесі  
 if* (isEmpty()) *throw new* NoSuchElementException("Stack is empty");  
 T removeElem = firstNode.getNext().getElem();  
 firstNode.setElem(*null*);  
 firstNode = getNext(firstNode);  
 size--;  
 *return* removeElem;  
 }  
  
 *public* T peek() {  
 *return* firstNode.getNext().getElem();*//Повертає елемент, що знаходиться у верхній частині стека, але не видаляє його.* }  
  
 *public* T push(T elem) {*//додає елемент у стек* Node<T> next = firstNode;  
 next.setElem(elem);  
 firstNode = *new* Node<>(*null*, next);  
 size++;  
 *return* elem;  
 }  
  
 *public* T get(*int* index) {*//повертає елемент за індексом* Node<T> temp = firstNode.getNext();  
 *for* (*int* i = 0; i < index; i++) {  
 temp = getNext(temp);  
 }  
  
 *return* temp.getElem();  
 }  
  
 *public int* search(T element) {*//повертає індекс елемента у стеку  
 int* index = -1;  
 Node<T> temp = firstNode.getNext();  
 *for* (*int* i = 0; i < size(); i++) {  
 *if* (temp.getElem() == element) {  
 index = i;  
 *break*;  
 }  
 temp = getNext(temp);  
 }  
 *return* index;  
 }  
  
 *public boolean* isEmpty() {*//перевіряє чи стек пустий  
 return* size == 0;  
 }  
  
 @Override  
 *public* String toString() {*// Перевизначимо метод toString  
  
 if* (size == 0) *return* "Stack is clear";  
 StringBuilder sb = *new* StringBuilder();  
 sb.append("[");  
  
 Node<T> current = firstNode.getNext();  
 *while* (current != *null*) {  
  
 *if* (current.getElem() == *null*) {  
 current = getNext(current);  
 *continue*;  
 }  
  
 sb.append(current.getElem());  
 *if* (current.next == *null*) {  
 *break*;  
 }  
 current = current.next;  
  
 *if* (current != *null*) sb.append(", ");  
 }  
 sb.append("]");  
 *return* sb.toString();  
 }  
  
 *private* Node<T> getNext(Node<T> current) {  
 *return* current.getNext();  
 }  
  
 *private class* Node<T> {  
  
 *private* T elem;  
 *private* Node<T> next;  
  
 *private* Node(T elem, Node<T> next) {*//створимо конструктор  
 this*.elem = elem;  
 *this*.next = next;  
 }  
  
 *public* T getElem() {*//створимо сетери і гетери  
 return* elem;  
 }  
  
 *public void* setElem(T elem) {  
 *this*.elem = elem;  
 }  
  
 *public* Node<T> getNext() {  
 *return* next;  
 }  
  
 *public void* setNext(Node<T> next) {  
 *this*.next = next;  
 }  
  
 }  
  
 @Override  
 *public Iterator*<T> iterator() {*//створимо ітератор  
 return new* Iterator() {  
 *int* counter = 0;  
  
 @Override  
 *public boolean* hasNext() {  
 *return* counter < size;  
 }  
  
 @Override  
 *public* Object next() {  
 *return* get(counter++);  
 }  
 };  
 }  
}

**DoublyLinkedList**

*import* java.util.*Iterator*;  
*import* java.util.NoSuchElementException;  
  
  
*public class* DoublyLinkedList<E> *implements Iterable*<E> {  
  
 *private* Node<E> firstNode;  
 *private* Node<E> lastNode;  
 *private int* size = 0;  
  
 *public* DoublyLinkedList() {  
  
 lastNode = *new* Node<E>(*null*, firstNode, *null*);  
 firstNode = *new* Node<E>(*null*, *null*, lastNode);  
 }  
  
 *public int* size() {  
 *return* size;  
 }  
  
 *public boolean* isEmpty() {  
 *return* size == 0;  
 }  
  
 *public void* addFirst(E e) {  
 Node<E> next = firstNode;  
 next.setElem(e);  
 firstNode = *new* Node<>(*null*, *null*, next);  
 next.setPrev(firstNode);  
 size++;  
 }  
  
 *public void* addLast(E e) {  
 Node<E> prev = lastNode;  
 prev.setElem(e);  
 lastNode = *new* Node<E>(*null*, prev, *null*);  
 prev.setNext(lastNode);  
 size++;  
 }  
  
 *public void* removeFirst() {  
 *if* (isEmpty()) *throw new* NoSuchElementException("List is clear");  
 firstNode = firstNode.getNext();  
 size--;  
  
 }  
  
 *public void* removeLast() {  
 *if* (isEmpty()) *throw new* NoSuchElementException("List is clear");  
 lastNode = lastNode.getPrev();  
 size--;  
 }  
  
 *public void* remove(*int* index) {  
 *if* (index < 0 || index >= size) *throw new* NoSuchElementException("No element index");  
 *if* (index == 0) {  
 removeFirst();  
  
 } *else if* (index == size() - 1) {  
 removeLast();  
  
 } *else* {  
  
 Node<E> temp = firstNode.getNext();  
 *for* (*int* i = 0; i < index - 1; i++) {  
 temp = temp.next;  
 }  
 Node<E> temp2 = temp.next;  
 temp.setNext(temp2.next);  
 temp2.next.setPrev(temp);  
 temp2.setElem(*null*);  
 size--;  
  
 }  
  
 }  
  
 *public void* clear() {  
 size = 0;  
 lastNode = *new* Node<E>(*null*, firstNode, *null*);  
 firstNode = *new* Node<E>(*null*, *null*, lastNode);  
  
 }  
  
 *public* E get(*int* index) {  
  
 *if* (index < 0 || index >= size) *throw new* NoSuchElementException("No element index");  
 Node<E> temp = firstNode.getNext();  
 *for* (*int* i = 0; i < index; i++) {  
 temp = getNext(temp);  
 }  
 *return* temp.getElem();  
 }  
  
 *public void* set(*int* index, E value) {  
 *if* (index < 0 || index >= size) *throw new* NoSuchElementException("No element index");  
 Node<E> temp = firstNode.getNext();  
 *for* (*int* i = 0; i < index; i++) {  
 temp = getNext(temp);  
 }  
 temp.setElem(value);  
 }  
  
 *public int* indexOf(E elem) {  
 Node<E> temp = firstNode.getNext();  
 *int* index = -1;  
 *for* (*int* i = 0; i < size; i++) {  
 *if* (elem.toString().equals(temp.getElem().toString())) {  
 index = i;  
 *break*;  
 }  
 temp = getNext(temp);  
 }  
  
 *if* (index == -1) *throw new* NoSuchElementException("No element index");  
 *return* index;  
 }  
  
 @Override  
 *public* String toString() {*//Перевизначимо метод toString  
  
 if* (size == 0) *return* "List is clear";  
 StringBuilder sb = *new* StringBuilder();  
 sb.append("[");  
  
 Node<E> current = firstNode.getNext();  
  
 *while* (current != *null*) {  
  
 sb.append(current.getElem());  
 *if* (current.next == lastNode) {  
 *break*;  
 }  
 current = current.next;  
 *if* (current != *null*) sb.append(", ");  
 }  
  
 sb.append("]");  
 *return* sb.toString();  
 }  
  
 @Override  
 *public Iterator*<E> iterator() {  
 *return new* Iterator<E>() {  
 *int* counter = 0;  
  
 @Override  
 *public boolean* hasNext() {  
 *return* counter < size;  
 }  
  
 @Override  
 *public* E next() {  
 *return* get(counter++);  
 }  
 };  
 }  
  
 *private* Node<E> getNext(Node<E> current) {  
 *return* current.getNext();  
 }  
  
 *private* Node<E> getPrev(Node<E> current) {  
 *return* current.getPrev();  
 }  
  
 *private class* Node<E> {  
  
 *private* E elem;  
 *private* Node<E> prev;  
 *private* Node<E> next;  
  
 *private* Node(E elem, Node<E> prev, Node<E> next) {  
 *this*.elem = elem;  
 *this*.prev = prev;  
 *this*.next = next;  
 }  
  
 *public* E getElem() {  
 *return* elem;  
 }  
  
 *public void* setElem(E elem) {  
 *this*.elem = elem;  
 }  
  
 *public* Node<E> getNext() {  
 *return* next;  
 }  
  
 *public void* setNext(Node<E> next) {  
 *this*.next = next;  
 }  
  
 *public* Node<E> getPrev() {  
 *return* prev;  
 }  
  
 *public void* setPrev(Node<E> prev) {  
 *this*.prev = prev;  
 }  
  
  
 }  
}

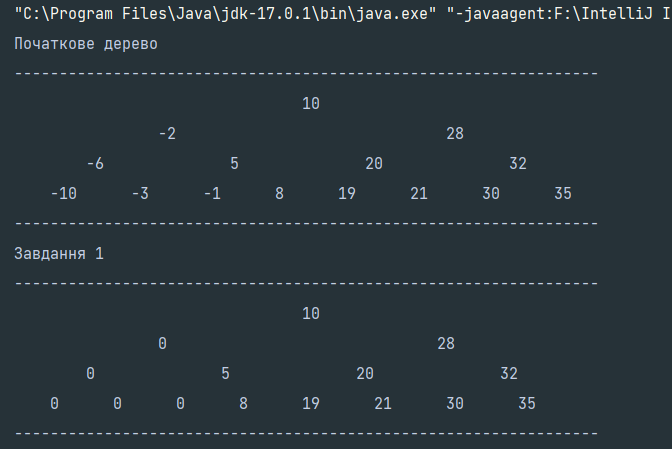
**Tree**

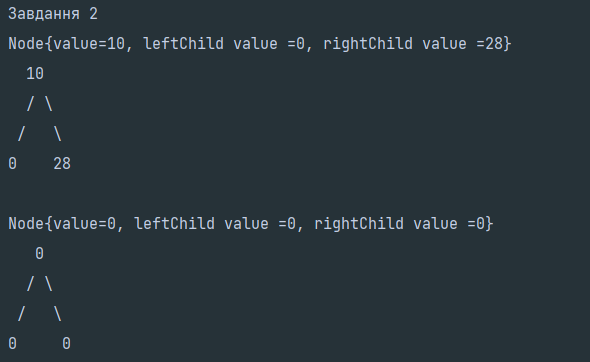
*import* java.util.NoSuchElementException;  
  
*public class* Tree {  
  
 *private* Node root;  
  
 *public boolean* isEmpty() {  
 *return* size() == 0;  
 }  
  
 *public int* size() {*//метод для виведення розміру дерева  
 return* size(root);  
  
 }  
  
 *private int* size(Node temp) {  
 *if* (temp == *null*) *return* 0;  
 *else return* temp.size;  
 }  
  
 *public void* put(*int* value) {*//метод для добавлення ключа* root = put(root, value);  
 }  
  
 *private* Node put(Node temp, *int* value) {  
 *if* (temp == *null*) *return new* Node(value, 1);  
 *int* cmp = value - temp.getValue();  
 *if* (cmp < 0) temp.left = put(temp.left, value);  
 *else if* (cmp > 0) temp.right = put(temp.right, value);  
 temp.size = 1 + size(temp.left) + size(temp.right);  
 *return* temp;  
 }  
  
 *public* Node findNodeByValue(*int* value) { *// поиск узла по значению* Node currentNode = root; *// начинаем поиск с корневого узла  
 while* (currentNode.getValue() != value) { *// поиск покуда не будет найден элемент или не будут перебраны все  
 if* (value < currentNode.getValue()) { *// движение влево?* currentNode = currentNode.getLeft();  
 } *else* { *//движение вправо* currentNode = currentNode.getRight();  
 }  
 *if* (currentNode == *null*) { *// если потомка нет,  
 return null*; *// возвращаем null* }  
 }  
 *return* currentNode; *// возвращаем найденный элемент* }  
  
 *public int* min() {*//метод для звертання до мінімального ключа  
 if* (isEmpty()) *throw new* NoSuchElementException("called min() with empty symbol table");  
 *return* min(root).getValue();  
 }  
  
 *private* Node min(Node temp) {  
 *if* (temp.left == *null*) *return* temp;  
 *else return* min(temp.left);  
 }  
  
  
 *public void* deleteMin() {*//видалення мінімального елементу  
 if* (isEmpty()) *throw new* NoSuchElementException("MyStructures.BST underflow");  
 root = deleteMin(root);  
 }  
  
 *private* Node deleteMin(Node temp) {  
 *if* (temp.left == *null*) *return* temp.right;  
 temp.left = deleteMin(temp.left);  
 temp.size = size(temp.left) + size(temp.right) + 1;  
 *return* temp;  
 }  
  
  
 *public void* delete(*int* value) {*//метод для виведення ключа* root = delete(root, value);  
 }  
  
 *private* Node delete(Node temp, *int* value) {  
 *if* (temp == *null*) *return null*;  
  
 *int* cmp = value - temp.getValue();  
 *if* (cmp < 0) temp.left = delete(temp.left, value);  
 *else if* (cmp > 0) temp.right = delete(temp.right, value);  
 *else* {  
 *if* (temp.right == *null*) *return* temp.left;  
 *if* (temp.left == *null*) *return* temp.right;  
 Node t = temp;  
 temp = min(t.right);  
 temp.right = deleteMin(t.right);  
 temp.left = t.left;  
 }  
 temp.size = size(temp.left) + size(temp.right) + 1;  
 *return* temp;  
 }  
  
  
 *public void* task1() {*//завдання №1* preorderForTask1(root);  
 }  
  
 *public void* preorderForTask1(Node temp) {  
  
 *if* (temp == *null*)  
 *return*;  
  
 *if* (temp.getValue() < 0) {  
 temp.setKey(0);  
 }  
 preorderForTask1(temp.getLeft());  
 preorderForTask1(temp.getRight());  
  
 }  
  
 *public void* task2() {*//завдання №2* preorderForTask2(root);  
 }  
  
 *public void* preorderForTask2(Node temp) {  
  
 *if* (temp == *null*)  
 *return*;  
  
 *if* (temp.getRight() != *null* && temp.getLeft() != *null*) {  
 *if* ((temp.getRight().getValue() + temp.getLeft().getValue()) % 2 == 0) {  
 System.***out***.println(temp);  
 System.***out***.printf(" %2d \n", temp.getValue());  
 System.***out***.println(" /\t\\ \n / \t \\ ");  
 System.***out***.printf("%d %2d\n\n", temp.getLeft().getValue(), temp.getRight().getValue());  
 }  
 }  
 preorderForTask2(temp.getLeft());  
 preorderForTask2(temp.getRight());  
  
 }  
  
  
 *public void* printTree() { *// метод для виведення дерева* Stack<Node> globalStack = *new* Stack<>();  
 globalStack.push(root);  
 *int* gaps = 32;  
 *boolean* isRowEmpty = *false*;  
 String separator = "-----------------------------------------------------------------";  
 System.***out***.println(separator);  
 *while* (!isRowEmpty) {  
 Stack<Node> localStack = *new* Stack<>();  
 isRowEmpty = *true*;  
  
 *for* (*int* j = 0; j < gaps; j++)  
 System.***out***.print(' ');  
 *while* (!globalStack.isEmpty()) {  
 Node temp = globalStack.pop();  
 *if* (temp != *null*) {  
 System.***out***.print(temp.getValue());  
 localStack.push(temp.getLeft());  
 localStack.push(temp.getRight());  
 *if* (temp.getLeft() != *null* ||  
 temp.getRight() != *null*)  
 isRowEmpty = *false*;  
 } *else* {  
 System.***out***.print("\_\_");  
 localStack.push(*null*);  
 localStack.push(*null*);  
 }  
 *for* (*int* j = 0; j < gaps \* 2 - 2; j++)  
 System.***out***.print(' ');  
 }  
 System.***out***.println();  
 gaps /= 2;  
 *while* (!localStack.isEmpty())  
 globalStack.push(localStack.pop());  
 }  
 System.***out***.println(separator);  
 }  
  
  
 *private static class* Node {  
  
 *private int* value;  
 *private* Node left, right;  
 *private int* size;  
  
 *public* Node(*int* value, *int* size) {  
 *this*.value = value;  
 *this*.size = size;  
 }  
  
 *public int* getValue() {  
 *return* value;  
 }  
  
 *public* Node getLeft() {  
 *return* left;  
 }  
  
 *public* Node getRight() {  
 *return* right;  
 }  
  
 *public void* setKey(*int* value) {  
 *this*.value = value;  
 }  
  
 @Override  
 *public* String toString() {  
 *return* "Node{" +  
 "value=" + value +  
 ", leftChild value =" + getLeft().getValue() +  
 ", rightChild value =" + getRight().getValue() +  
 '}';  
 }  
  
 }  
  
}

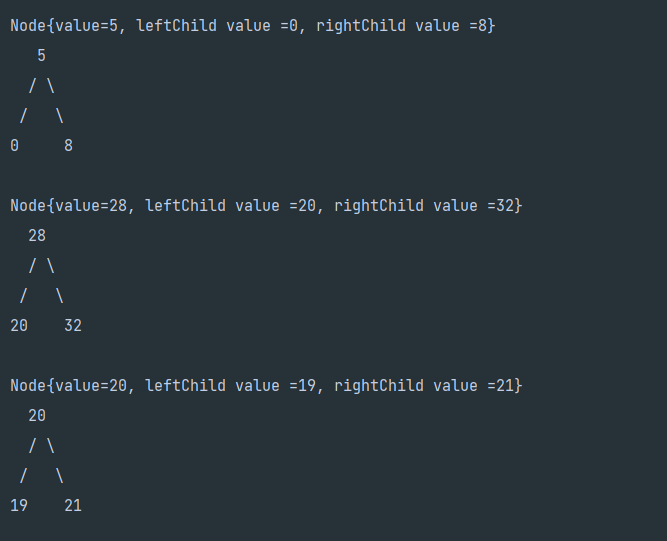
**Main**

*public class* Main {  
 *public static void* main(String[] args) {  
 Tree tree = *new* Tree();  
 tree.put(10);  
 tree.put(-2);  
 tree.put(28);  
 tree.put(-6);  
 tree.put(5);  
 tree.put(20);  
 tree.put(32);  
 tree.put(-10);  
 tree.put(-3);  
 tree.put(-1);  
 tree.put(8);  
 tree.put(19);  
 tree.put(21);  
 tree.put(30);  
 tree.put(35);  
 System.***out***.println("Початкове дерево");  
 tree.printTree();  
 System.***out***.println("Завдання 1 ");  
 tree.task1();  
 tree.printTree();  
 System.***out***.println("Завдання 2 ");  
 tree.task2();  
   
 }  
  
}

**Результат програми:**

****

****

****

**Код до 1 завдання**

*public void* task1() {*//завдання №1* preorderForTask1(root);  
}  
  
*public void* preorderForTask1(Node temp) {  
  
 *if* (temp == *null*) *return*;  
   
 *if* (temp.getValue() < 0) {  
 temp.setKey(0);  
 }  
 preorderForTask1(temp.getLeft());  
 preorderForTask1(temp.getRight());  
}

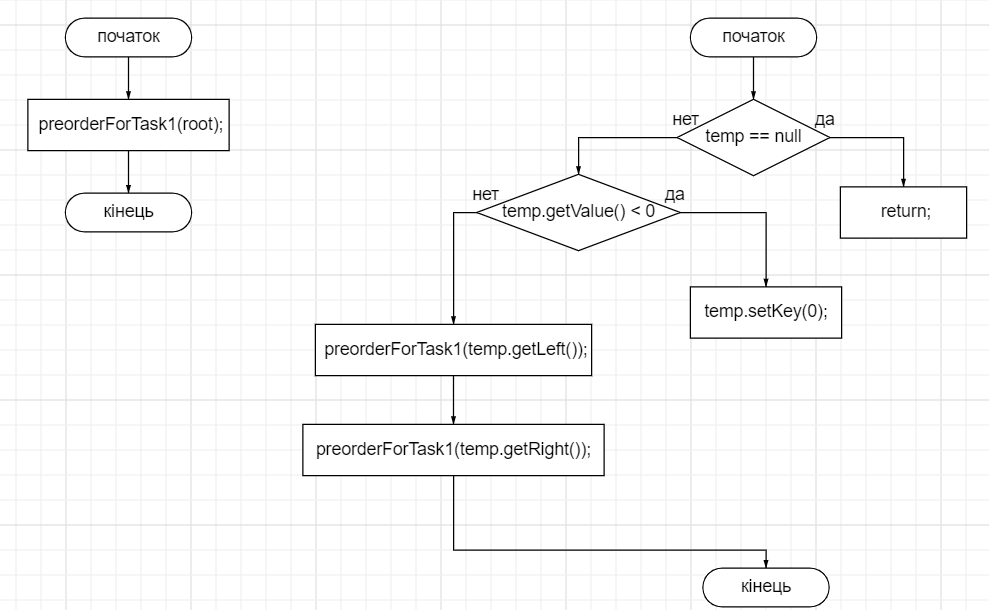
**Код до 2 завдання**

*public void* task2() {*//завдання №2* preorderForTask2(root);  
}  
  
*public void* preorderForTask2(Node temp) {  
  
 *if* (temp == *null*) *return*;  
  
 *if* (temp.getRight() != *null* && temp.getLeft() != *null*) {  
 *if* ((temp.getRight().getValue() + temp.getLeft().getValue()) % 2 == 0) {  
 System.***out***.println(temp);  
 System.***out***.printf(" %2d \n", temp.getValue());  
 System.***out***.println(" /\t\\ \n / \t \\ ");  
 System.***out***.printf("%d %2d\n\n", temp.getLeft().getValue(), temp.getRight().getValue());  
 }  
 }  
 preorderForTask2(temp.getLeft());  
 preorderForTask2(temp.getRight());  
}

**Блок схема до 1 завдання**

Task1

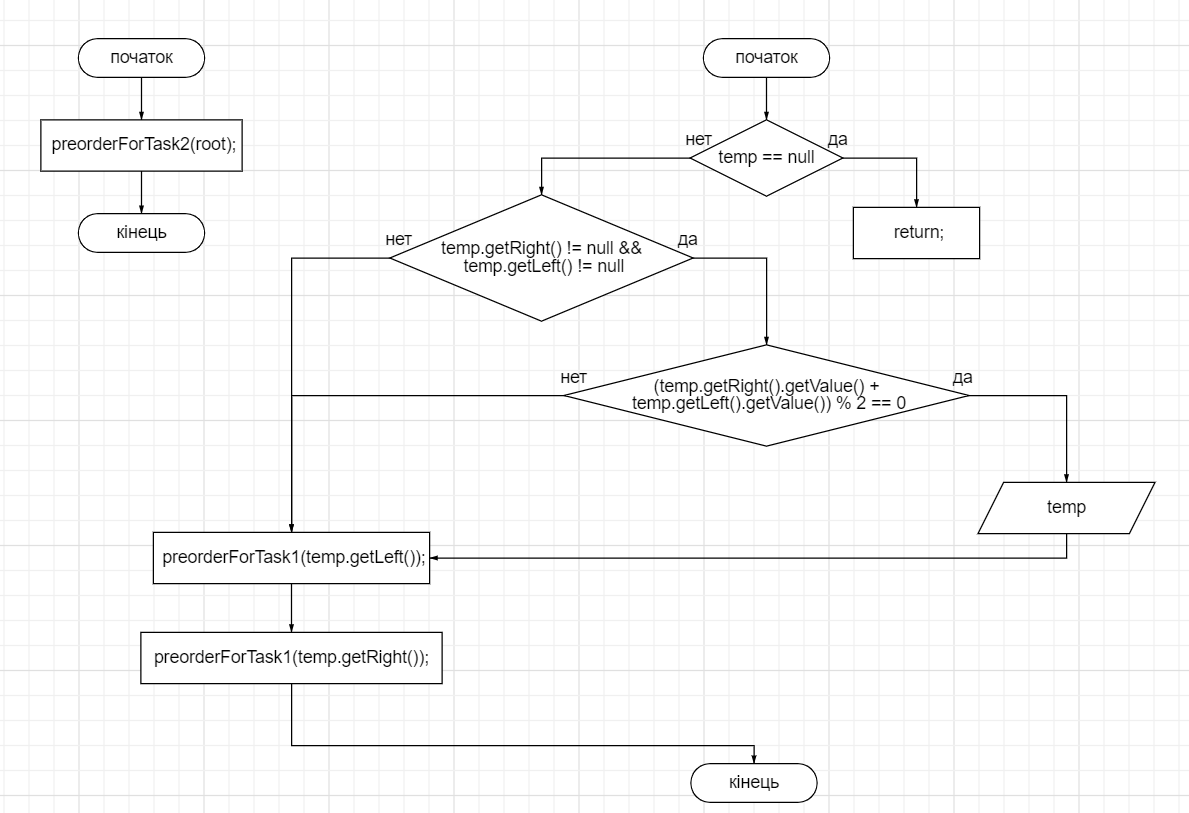
preorderForTask1

****

**Блок схема до 2 завдання**

Task2

preorderForTask2

****

**Висновок:** На цій лабораторній роботі було набуто навички створення та обробки бінарного дерева. Ознайомлено із теоретичними відомостями. Виконано завдання згідно варіанту. Перевірено результати на правильність. Розроблено блок схеми до кожного завдання.