Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ «МОСКОВСКИЙ ИНСТИТУТ ЭЛЕКТРОННОЙ ТЕХНИКИ»

Институт системной и программной инженерии

и информационных технологий (СПИНТех)

**Отчёт**

по дисциплине «Алгоритмы и структуры данных»

**Лабораторная работа №2**

**Вариант-10**

Руководитель

\_\_\_\_\_\_\_\_\_\_\_ Волков A. C.

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_2023 г.

Студент группы ПИН-23

\_\_\_\_\_\_\_\_\_\_\_ Исламов Р. Р.

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_2023 г.

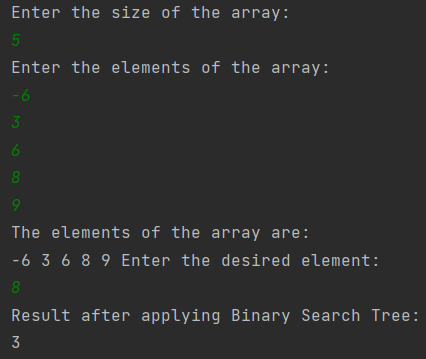
*Москва*

*2023*

# 1.1 Листинг. Поиск по Бору

import java.util.Scanner;  
  
class BinarySearchTree {  
 public static int binarySearchRecursively(int[] array, int key) {  
 return *binarySearchRecursively*(array, 0, array.length, key);  
 }  
  
 public static int binarySearchRecursively(  
 int[] array, int fromIndex, int toIndex, int key) {  
 if (toIndex <= fromIndex) return -1;  
  
 int mid = (fromIndex + toIndex) >>> 1;  
 int midVal = array[mid];  
  
 if (key == midVal) {  
 return mid;  
 } else if (key < midVal) {  
 return *binarySearchRecursively*(array, fromIndex, mid, key);  
 } else {  
 return *binarySearchRecursively*(array, mid + 1, toIndex, key);  
 }  
 }  
  
 public static void main(String[] args) {  
  
  
 Scanner sc = new Scanner(System.*in*);  
  
 System.*out*.println("Enter the size of the array: ");  
 int arr\_size = 0;  
 if (sc.hasNextInt()) {  
 arr\_size = sc.nextInt();  
 }  
 int[] arr = new int[arr\_size];  
  
 System.*out*.println(  
 "Enter the elements of the array: ");  
 for (int i = 0; i < arr\_size; i++) {  
 if (sc.hasNextInt()) {  
 arr[i] = sc.nextInt();  
 }  
 }  
 System.*out*.println(  
 "The elements of the array are: ");  
 for (int i = 0; i < arr\_size; i++) {  
 System.*out*.print(arr[i] + " ");  
 }  
 System.*out*.println("Enter the desired element: ");  
 int key = sc.nextInt();  
 System.*out*.println("Result after applying Binary Search Tree: ");  
 System.*out*.println(*binarySearchRecursively*(arr, key));  
 }  
}

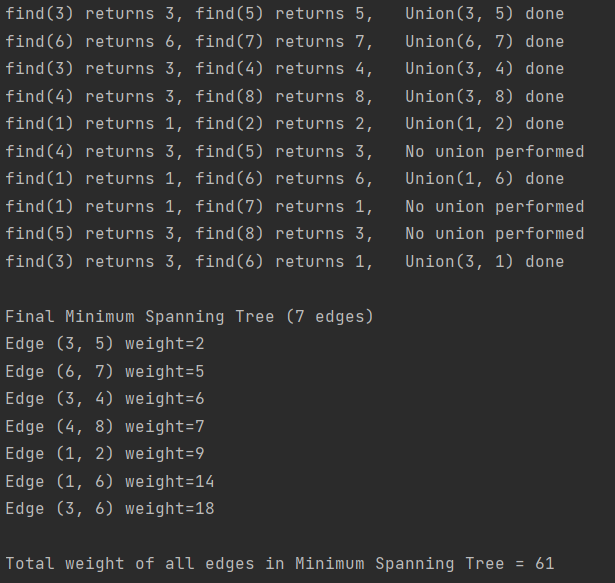
# 1.2 Результат работы



# 2.1 Листинг. Метод Крускала

package Lab2;  
import java.util.\*;  
import java.io.\*;  
  
public class Kruskal {  
 public static void main(String[] args) {  
 ArrayList<Edge> graphEdges = new ArrayList<Edge>(); //edge list, not adjacency list  
 graphEdges.add(new Edge(3, 5, 2));  
 graphEdges.add(new Edge(6, 7, 5));  
 graphEdges.add(new Edge(3, 4, 6));  
 graphEdges.add(new Edge(4, 8, 7));  
 graphEdges.add(new Edge(1, 2, 9));  
 graphEdges.add(new Edge(4, 5, 11));  
 graphEdges.add(new Edge(1, 6, 14));  
 graphEdges.add(new Edge(1, 7, 15));  
 graphEdges.add(new Edge(5, 8, 16));  
 graphEdges.add(new Edge(3, 6, 18));  
 graphEdges.add(new Edge(3, 8, 19));  
 graphEdges.add(new Edge(7, 5, 20));  
 graphEdges.add(new Edge(2, 3, 24));  
 graphEdges.add(new Edge(7, 8, 44));  
 graphEdges.add(new Edge(6, 5, 30));  
  
 int nodeCount = 8;  
 Kruskal graph = new Kruskal();  
 graph.kruskalMST(graphEdges, nodeCount);  
 }  
  
 public void kruskalMST(ArrayList<Edge> graphEdges, int nodeCount){  
 String outputMessage="";  
  
 Collections.*sort*(graphEdges);  
 ArrayList<Edge> mstEdges = new ArrayList<Edge>();  
 DisjointSet nodeSet = new DisjointSet(nodeCount+1);  
 for(int i=0; i<graphEdges.size() && mstEdges.size()<(nodeCount-1); i++){  
 Edge currentEdge = graphEdges.get(i);  
 int root1 = nodeSet.find(currentEdge.getVertex1());  
 int root2 = nodeSet.find(currentEdge.getVertex2());  
 outputMessage+="find("+currentEdge.getVertex1()+") returns "+root1+", find("+currentEdge.getVertex2()+") returns "+root2; //just print, keep on same line for union message  
 String unionMessage=",\tNo union performed\n";  
 if(root1 != root2){  
 mstEdges.add(currentEdge);  
 nodeSet.union(root1, root2);  
 unionMessage=",\tUnion("+root1+", "+root2+") done\n";  
 }  
 outputMessage+=unionMessage;  
 }  
  
 outputMessage+="\nFinal Minimum Spanning Tree ("+mstEdges.size()+" edges)\n";  
 int mstTotalEdgeWeight=0;  
 for(Edge edge: mstEdges){  
 outputMessage+=edge +"\n";   
 mstTotalEdgeWeight += edge.getWeight();  
 }  
 outputMessage+="\nTotal weight of all edges in Minimum Spanning Tree = "+mstTotalEdgeWeight;  
  
 System.*out*.println(outputMessage);  
  
 }  
}  
  
  
class Edge implements Comparable<Edge>{  
 private int vertex1;  
 private int vertex2;  
 private int weight;  
  
 public Edge(int vertex1, int vertex2, int weight){  
 this.vertex1=vertex1;  
 this.vertex2=vertex2;  
 this.weight=weight;  
 }  
  
 public int getVertex1(){  
 return vertex1;  
 }  
  
 public int getVertex2(){  
 return vertex2;  
 }  
  
 public int getWeight(){  
 return weight;  
 }  
  
 @Override  
 public int compareTo(Edge otherEdge) { //Compare based on edge weight (for sorting)  
 return this.getWeight() - otherEdge.getWeight();  
 }  
  
 @Override  
 public String toString() {  
 return "Edge ("+getVertex1()+", "+getVertex2()+") weight="+getWeight();  
 }  
}  
  
class DisjointSet{  
 private int[] set; //the disjoint set as an array  
  
 public int[] getSet(){ //mostly debugging method to print array  
 return set;  
 }  
  
 public DisjointSet(int numElements) { //constructor creates singleton sets  
 set = new int [numElements];  
 for(int i = 0; i < set.length; i++){ //initialize to -1 so the trees have nothing in them  
 set[i] = -1;  
 }  
 }  
  
  
 public void union(int root1, int root2) {  
 if(set[root2] < set[root1]){ // root2 is deeper  
 set[root1] = root2; // Make root2 new root  
 }  
 else {  
 if(set[root1] == set[root2]){  
 set[root1]--; // Update height if same  
 }  
 set[root2] = root1; // Make root1 new root  
 }  
 }  
  
  
 public int find(int x) {  
 if(set[x] < 0){ //If tree is a root, return its index  
 return x;  
 }  
 int next = x;  
 while(set[next] > 0){ //Loop until we find a root  
 next=set[next];  
 }  
 return next;  
 }  
  
}

# 2.2 Результат работы



# 3 Контрольные вопросы

Контрольные вопросы по тексту лабораторной 3:

1. Каковы особенности итеративного и рекурсивного алгоритма? 2. В каких случаях целесообразно использовать рекурсивный или итеративный алгоритм? Приведите примеры итерации и рекурсии. 3. Все ли языки программирования дают возможность рекурсивного вызова процедур? 4. Приведите пример рекурсивной структуры данных. 5. Что такое указатели и динамические переменные в языке Турбо Паскаль? 6. Укажите виды обхода бинарных деревьев. 7. Приведите пример рекурсивной структуры данных. 8. Что такое указатели и динамические переменные в алгоритмических языках?

1. Особенности итеративного и рекурсивного алгоритма:

* Итеративный алгоритм основан на использовании циклов и повторяющихся операций. Он выполняет последовательность шагов для достижения результата. При итеративном подходе каждый шаг может быть явно контролируемым и последовательным.
* Рекурсивный алгоритм опирается на вызов самого себя для решения задачи. Он разбивает задачу на более маленькие подзадачи и решает их рекурсивно до достижения базового случая. Рекурсия требует базового случая и правильного условия остановки, чтобы избежать бесконечной рекурсии.

1. Целесообразность использования рекурсивного или итеративного алгоритма зависит от природы задачи и требований. Вот несколько примеров:

* Рекурсивный алгоритм может быть полезен, когда задача может быть разбита на подзадачи с тем же самым характером, и когда каждая подзадача решается таким же способом, что и исходная задача. Примеры рекурсии: вычисление факториала числа, обход деревьев, сортировка слиянием.
* Итеративный алгоритм хорошо подходит для простых задач, которые могут быть решены последовательностью действий без необходимости разбиения на подзадачи. Примеры итерации: поиск максимального элемента в массиве, сортировка пузырьком.

1. Не все языки программирования предоставляют встроенную поддержку рекурсии. Однако большинство современных языков программирования позволяют выполнять рекурсивные вызовы процедур. Некоторые из таких языков включают C, C++, Java, Python, JavaScript и другие.
2. Пример рекурсивной структуры данных - связанный список. Связанный список состоит из узлов, каждый из которых содержит значение и ссылку на следующий узел. Рекурсия возникает при переборе элементов связанного списка, когда каждый узел ссылается на следующий узел, пока не достигнут конечный узел (базовый случай).
3. В языке Turbo Pascal указатели и динамические переменные используются для работы с памятью напрямую:

* Указатели - это переменные, которые содержат адреса в памяти. Они позволяют обращаться к данным по этому адресу и выполнять операции над этими данными. Указатели в Turbo Pascal объявляются с использованием символа "^". Например, "var p: ^Integer;" объявляет указатель p на переменную типа Integer.
* Динамические переменные - это переменные, которые создаются и освобождаются во время выполнения программы. Они позволяют динамически выделять память для хранения данных. В Turbo Pascal динамические переменные создаются с помощью оператора "New" и освобождаются с помощью оператора "Dispose".

1. Виды обхода бинарных деревьев:

* Прямой (pre-order) обход: сначала посещается корень дерева, затем рекурсивно обходятся левое поддерево и затем правое поддерево.
* Симметричный (in-order) обход: сначала рекурсивно обходится левое поддерево, затем посещается корень и, наконец, рекурсивно обходится правое поддерево.
* Обратный (post-order) обход: сначала рекурсивно обходится левое поддерево, затем правое поддерево и, наконец, посещается корень.

1. Пример рекурсивной структуры данных - дерево. Дерево состоит из узлов, каждый из которых может иметь ноль или более дочерних узлов. Каждый дочерний узел также является деревом. Рекурсия возникает при обходе и обработке узлов дерева, где каждый узел рекурсивно рассматривается как дерево.
2. В алгоритмических языках указатели и динамические переменные используются для работы с памятью и динамическим выделением ресурсов. Указатели позволяют обращаться к данным по адресу в памяти, а динамические переменные позволяют выделять и освобождать память во время выполнения программы. Они могут быть полезны при работе с большими объемами данных, динамическом создании структур данных или при написании оптимизированных алгоритмов. Однако использование указателей и динамических переменных требует аккуратности и правильного управления памятью, чтобы избежать проблем, таких как утечки памяти или неправильное обращение к данным.

Контрольные вопросы по тексту лабораторной 4:

1. Что понимается под остовным деревом? 2. Каковы особенности методов Крускала и Прима? 3. В чем состоит методика анализа сложности алгоритмов построения остовного дерева графа? 4. Определить, является ли связным заданный граф. 5. Найти все вершины графа, к которым существует путь заданной длины от выделенной вершины графа. 6. Найти все вершины графа, достижимые из заданной. 7. Подсчитать количество компонент связности заданного графа. 8. Найти диаметр графа, т.е. максимум расстояний между всевозможными парами его вершин. 9. Найти такую нумерацию вершин орграфа, при которой всякая дуга ведет от вершины с меньшим номером к вершине с большим номером.

1. Остовное дерево - это подграф связного неориентированного графа, который включает все вершины и является деревом (не содержит циклов), сохраняя при этом связность графа. Остовное дерево содержит все вершины исходного графа и наименьшее возможное количество ребер.
2. Метод Крускала и метод Прима являются алгоритмами построения остовного дерева во взвешенном неориентированном графе:

* Метод Крускала: строит остовное дерево, добавляя на каждом шаге ребро с наименьшим весом, при условии что добавление этого ребра не создаст цикл.
* Метод Прима: начиная с некоторой начальной вершины, на каждом шаге добавляет ребро с минимальным весом, присоединяющее текущее дерево к новой вершине, и продолжает этот процесс, пока все вершины не будут включены в остовное дерево.

1. Методика анализа сложности алгоритмов построения остовного дерева графа включает оценку времени выполнения и объема затрачиваемой памяти. Она основывается на анализе количества операций, которые алгоритм выполняет в зависимости от размера входных данных (число вершин и ребер графа). Обычно сложность алгоритмов измеряется величиной O(f(n)), где n - размер входных данных, а f(n) - функция, описывающая зависимость количества операций от n.
2. Для определения, является ли заданный граф связным, можно использовать алгоритм обхода графа, такой как обход в глубину (DFS) или обход в ширину (BFS). Если в результате обхода каждая вершина достигается из любой другой вершины, то граф является связным. Если существуют недостижимые вершины, то граф не является связным.
3. Для поиска всех вершин графа, к которым существует путь заданной длины от выделенной вершины, можно использовать модифицированный обход графа (например, DFS или BFS) с отслеживанием длины пути. При достижении вершины с заданной длиной пути, эта вершина добавляется в результат.
4. Для поиска всех вершин графа, достижимых из заданной вершины, можно использовать обход графа, такой как обход в глубину (DFS) или обход в ширину (BFS). При обходе графа от заданной вершины, все достигнутые вершины могут быть добавлены в результат.
5. Количество компонент связности графа может быть подсчитано с использованием алгоритма обхода графа, такого как обход в глубину (DFS) или обход в ширину (BFS). Каждый раз, когда обход стартует из новой непосещенной вершины, это означает новую компоненту связности.
6. Диаметр графа - это наибольшее расстояние между всевозможными парами вершин. Для нахождения диаметра графа можно применить алгоритм Флойда-Уоршелла или алгоритм Дейкстры для нахождения кратчайших путей между всеми парами вершин. Затем находится максимальное значение среди всех найденных расстояний.
7. Для нахождения такой нумерации вершин орграфа, при которой каждая дуга ведет от вершины с меньшим номером к вершине с большим номером, можно использовать топологическую сортировку. Топологическая сортировка упорядочивает вершины графа таким образом, что для каждой дуги из вершины A в вершину B, вершина A имеет меньший номер.