Министерство Образования, Культуры и Исследований

**Молдавский Государственный Университет**

**Факультет Математики и Информатики**

**Департамент Информатики**

Algoritmi, Structuri de Date și Complexitate

Отчёт по лабораторной работе №3

Выполнил: Maryia Rotar IA2102

Проверил преподаватель: Mihail Croitor

Кишинев, 2023

**Задание:**

**Для выполнения данной работы воспользуйтесь результатом индивидуальной работы №0.**

Из лабораторной работы №0 возьмите текстовый файл с данными и определения структур данных.

Используя данный текстовый файл реализуйте следующие операции:

* обход и вывод,
* вставку,
* поиск,
* удаление,

для следующих динамических структур данных (на выбор):

* односвязные списки,
* двусвязные списки,
* кольцевые списки,
* стеки,
* очереди.

Используя текстовый файл подготовленный для предыдущих индивидуальных работ реализовать бинарное дерево поиска и следующие операции:

* обход в прямом порядке,
* обход в обратном порядке,
* центрированный/симметричный обход,
* вставку элемента,
* удаление элемента,
* поиск элемента.

**Бинарное дерево поиска**

Двоичное дерево представляет конечное множество элементов (узлов),

которое либо пусто, либо состоит из одного специального узла, называемого

корнем, и двух непересекающихся между собой двоичных деревьев,

называемых соответственно левым поддеревом и правым поддеревом

данного двоичного дерева.

Под обходом двоичного дерева понимают последовательный переход от одного узла к другому через все узлы в целях обработки или поиска. В основном используются три способа обхода (engl. order traversal):

• прямой (engl. preorder traversal, rom. preordine, parcurgere în lăţime sau directă) – прямой порядок / обход в ширину;

• обратный (engl. symmetric order traversal / inorder traversal, rom. inordine, parcurgere simetrică) – обратный порядок / симметричный обход ;

• концевой (engl. postorder traversal / endorder traversal, rom. postordine, parcurgere în adîncime) – концевой порядок / обход в глубину.

**Прямой обход** состоит из следующих шагов:

a) обработка корня;

b) обход левого поддерева в прямом порядке, если оно не пусто;

c) обход правого поддерева в прямом порядке, если оно не пусто

public static void preorderTraversal(BinaryTree node) {  
 if (node == null) {  
 return;  
 }  
  
 System.*out*.print(node + " ");  
 *preorderTraversal*(node.getLeftChild());  
 *preorderTraversal*(node.getRightChild());  
}

**Обратный (симметричный) обход** состоит из следующих шагов:

a) обход левого поддерева в обратном порядке, если оно не пусто;

b) обработка корня;

c) обход правого поддерева в обратном порядке, если оно не пусто

public static void inorderTraversal(BinaryTree node) {  
 if (node == null) {  
 return;  
 }  
  
 *inorderTraversal*(node.getLeftChild());  
 System.*out*.print(node + " ");  
 *inorderTraversal*(node.getRightChild());  
}

**Концевой обход** (в глубину) состоит из следующих шагов:

a) обход левого поддерева в концевом порядке, если оно не пусто;

b) обход правого поддерева в концевом порядке, если оно не пусто;

c) обработка корня.

public static void postorderTraversal(BinaryTree node) {  
 if (node == null) {  
 return;  
 }  
  
 *postorderTraversal*(node.getLeftChild());  
 *postorderTraversal*(node.getRightChild());  
 System.*out*.print(node + " ");  
}

**Удаление элемента**

Идея удаления элемента делится на несколько случаев:

- у узла нет дочерних узлов;

- у узла есть левый дочерних узлов;

- у узла есть правый дочерних узлов;

- у узла есть оба ребёнка.

В случае 1 просто удаляем узел, дополнительная работа не требуется.

В случае 2 и 3 заменяем удаляемый узел на его потомка, на этом удаление заканчивается. Суть 4-го случая — найти в правом поддереве минимальный элемент и переместить его на место удаляемого узла.

public void delete(int key) {  
 root = deleteNode(root, key);  
}  
  
private BinaryTree deleteNode(BinaryTree root, int key) {  
 if (root == null) {  
 return root;  
 }  
  
 if (key < root.getNode().getId()) {  
 root.setLeftChild(deleteNode(root.getLeftChild(), key));  
 } else if (key > root.getNode().getId()) {  
 root.setRightChild(deleteNode(root.getRightChild(), key));  
 } else {  
 // Узел для удаления найден  
  
 // Узел без потомков или с одним потомком  
 if (root.getLeftChild() == null) {  
 return root.getRightChild();  
 } else if (root.getRightChild() == null) {  
 return root.getLeftChild();  
 }  
  
 // Узел с двумя потомками  
 root.setNode(minValue(root.getRightChild()));  
  
 // Удаление наименьшего значения из правого поддерева  
 root.setRightChild(deleteNode(root.getRightChild(), root.getNode().getId()));  
 }  
  
 return root;  
}  
  
  
  
private Product minValue(BinaryTree root) {  
 Product minValue = root.getNode();  
 while (root.getLeftChild() != null) {  
 minValue = root.getLeftChild().getNode();  
 root = root.getLeftChild();  
 }  
 return minValue;  
}

**Добавление элемента**

Добавление нового узла в дерево: если у узла отсутствует левая ветвь, то узел добавляется слева, иначе, если отсутствует правая ветвь, то узел добавляется справа, иначе узел не добавляется. Функция должна возвращать указатель на добавленный узел или 0;

public void insert(Product key) {  
 root = insertNode(root, key);  
}  
  
private BinaryTree insertNode(BinaryTree root, Product key) {  
 if (root == null) {  
 root = new BinaryTree(key);  
 return root;  
 }  
  
 if (key.getId() < root.getNode().getId()) {  
 root.setLeftChild(insertNode(root.getLeftChild(), key));  
 } else if (key.getId() > root.getNode().getId()) {  
 root.setRightChild(insertNode(root.getRightChild(), key));  
 }  
  
 return root;  
}

**Очередь (Queue)**

Очередь (engl. queue, FIFO list, pushup stack, pushup list) - это динамическая структура данных, состоящая из конечного упорядоченного набора элементов одного и того же типа, организованная таким образом, что добавление новых элементов осуществляется только в конец очереди, а доступ к ее элементам (преимущественно извлечение) осуществляется только с начала очереди, согласно принципу ―первым вошел – первым вышел ‖ (engl. FIFO – First In, First Out).

К очередям применяются следующие операции:

 добавление нового элемента к очереди (put), предыдущий последний становится

предпоследним;

 извлечение очередного элемента из очереди (get), предыдущий следующий элемент

становится первым;

 проверка если очередь пуста (isempty), т. е. нельзя применять операцию get;

 проверка если очередь полностью заполнена (isfull), т. е. нельзя применять операцию

put.

**Put**

public void put(T item) {  
 if (isFull()) {  
 throw new IllegalStateException("Queue is full");  
 }  
 rear = (rear + 1) % *BUFFER\_SIZE*;  
 buffer[rear] = item;  
 size++;  
}

**Get**

public T get() {  
 if (isEmpty()) {  
 throw new NoSuchElementException("Queue is empty");  
 }  
 T item = buffer[front];  
 front = (front + 1) % *BUFFER\_SIZE*;  
 size--;  
 return item;  
}

**IsEmpty**

public boolean isEmpty() {  
 return size == 0;  
}

**Isfull**

public boolean isFull() {  
 return size == *BUFFER\_SIZE*;  
}

Пример использования очереди:

System.*out*.println("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_QUEUE\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");  
  
MyQueue<Integer> queue = new MyQueueImpl<>();  
  
queue.put(1);  
queue.put(2);  
queue.put(3);  
  
while (!queue.isEmpty()) {  
 System.*out*.println(queue.get());  
}  
int i = 0;  
while (!queue.isFull()) {  
 queue.put(i++);  
}  
System.*out*.println("Queue is full -> " + queue.isFull() + ", size -> " + queue.getSize());

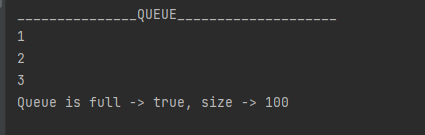


Рис.1 В результате работы очереди видно, что она работает по принципу FIFO и элементы, положенные первыми в буфер, первыми же из него и достаются.

**Выводы**

В данной лабораторной работе были разобраны динамические структуры данных Очередь и Бинарное дерево. Они имеют следующие свойства: возможность обхода и вывода элементов, вставку, поиск элемента в структуре и удаление элемента.