Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ

УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра комплексной информационной безопасности электронных вычислительных систем (КИБЭВС)

ОТЧЕТ

по практической работе №4

по дисциплине Технологии и методы программирования

Тема «Деревья»

Выполнила студентка гр.726-2

\_\_\_\_\_\_\_\_\_\_Кравцова А.В.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Принял к.т.н., доцент каф. БИС

Романов А. С.

\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

Томск 2018

**1 Введение**

Цель: научиться программировать деревья.

Задание:

1. Изучить теоретический материал по теме «Деревья».
2. При выполнении задания активно использоваться git
3. Реализовать программу, соответствующую Вашему варианту задания. Дерево реализовать в виде класса, необходимый функционал – в виде методов. Структура данных «дерево» должна быть реализована с помощью указателей.
4. В процессе реализации не забывать делать коммиты в github.
5. Написать отчет, залить его в MOODLE.
6. Защитить отчет у преподавателя.

**2 Ход работы**

**2.1 Краткие теоретические сведения**

Дерево – это АТД (абстрактный тип данных) для хранения информационных элементов, имеющих нелинейные отношения. За исключением элемента, который находится во главе дерева, каждый элемент имеет родителя, а также ноль или более дочерних элементов. Также говорят, что дерево – это множество элементов, среди которых есть один выделенный который называется корнем, а остальные содержатся в N непересекающихся подмножествах, которые называются поддеревьями. Очевидно, что дерево относится к классу динамических структур данных наряду с нам уже знакомыми стеком, очередями. С помощью дерева можно представить, например, следующую организационную структуру:

Наиболее известным классическим примером деревьев является генеалогическое дерево. Также в качестве примера можно привести принятую в большинстве файловых систем организацию с помощью иерархически вложенных директорий и файлов. Наиболее ярко это проявляется в О.С. linux и unix, где файловую систему представляют в виде единого дерева, которое начинается с корневой директории обозначаемой “/”.

При дальнейшем обсуждении деревьев следует условиться о терминологии. Корень дерева (root) – первый его элемент (в приведенном выше изображении это “Средняя школа # 123”). Каждый элемент данных называют узлом (node), иногда листом (leaf). Фрагмент дерева называется поддеревом (subtree) или ветвью. Узел, не имеющий отходящих от него ветвей ли поддеревьев, называются терминальным или конечным узлом (terminal node). Высота дерева (height) – определяется количеством уровней, на которых располагаются его узлы. При работе с деревьями полезно изображать их на бумаге, однако при этом всегда следует помнить, что деревья – это метод логической организации информации в памяти компьютера, не физической.

Следует помнить, что большинство операций, которые выполняются над деревьями, являются рекурсивными, т.к. деревья имеют рекурсивную структуру – каждое дерево содержит множество поддеревьев.

О деревьях можно судить как об упорядоченных или не упорядоченных. Мы говорим, что дерево упорядочено, если каждому из элементов можно сопоставить порядковый номер: 1,2,3. На рисунках упорядоченные “братья-узлы” представляются слева направо.

Пример упорядоченного дерева: книга. Книга имеет древовидную структуру с делением на разделы, каждый из которых состоит из множества тем, а тема включает отдельные вопросы. Каждый из вопросов также можно разделить на элементарные части – абзацы, картинки, таблицы. Очевидно, что эти элементы имеют четкий порядок следования, нарушение которого приводит к потере смысла книги.

В информатике наиболее часто используются именно двоичные деревья - это деревья, каждый узел которого содержит не более чем два подчиненных ему (дочерних элемента). Бинарное дерево считается правильным, если каждый узел не содержит одного или содержит два дочерних элемента. Таким образом – генеалогическое дерево – это правильно двоичное дерево. В бинарных деревьях дочерние узлы именуют “правый” и “левый”.

С помощью двоичного дерева можно легко представить арифметическое выражение с использованием бинарных операций.

Деревья широко применяются при создании экспертных систем и баз знаний. С помощью деревьев можно представлять деревья поиска решений, в которых переход от одного утверждения к другому есть спуск от корневого элемента (находящегося вверху дерева) к одному из конечных. В качестве еще одного примера можно привести изображение дерева игры в крестики-нолики, в данном дереве каждый узел дерева - это состояние на поле в определенный момент времени.

Для каждого узла дерева можно определить понятие глубины. Допустим, V – узел дерева. Глубина узла выражается количество предков V, за исключением самого V. По определению глубина корня равна нулю.

Глубина определяется рекурсивно:

* если узел – корень, то его глубина ноль;
* в противном случае глубина узла V равна 1 + глубина его родителя V`.

Способы представления деревьев: Представление дерева с помощью массива

Когда мы начинаем переходить к вопросу практического использования дерева, то следует решить, каким именно образом нам следует хранить его элементы, так чтобы не потерять ранжиры. Как ни странно, но можно решить данную задачу, даже если использовать обычный массив. Рассмотрим пример подобного представления для двоичного дерева, это не нарушает общности рассуждений, если вы хотите работать с мульти-деревом, то в этой ситуации необходимо четко определиться с максимальным количеством прямых потомков для каждого из узлов, если такого постоянного значения определить нельзя, то представление с помощью массивов не возможно, и вам следует использовать прием с динамически выделяемой памятью и даже некоторого синтеза дерева и списка. В каждом элементе списка будет храниться ссылка на каждого из потомков текущего узла V. Для представления с помощью массива мы каждому из элементов дерева дадим порядковый номер:

Очень важно, что если дерево не полное, то отсутствующим элементам назначаются номера и для них выделяется место в массиве, просто в ячейке с номером “8, 9, 13, 29” и других ничего не хранится.

Представление дерева с помощью ссылок

В случае множественного дерева (каждый узел может иметь произвольное количество потомков) возможно, воспользоваться подобным объявлением:

1. struct MultiNode {
2. DATA\_TYPE data;
3. MultiNode \* parent;
4. LIST\_OF\_NODES children;
5. };

В данном примере DATA\_TYPE – тип данных соответствующий хранимому в дереве, например, если у нас дерево математического выражения, то это может быть число или знак арифметической операции. Если дерево служит для хранения генеалогической информации – то в качестве типа каждого узла может быть строка с именем человека, или даже произвольная структура или класс CHuman, описывающий кого-то из предков. Обратите внимание на другое, в структуре присутствует ссылка на узел дерева, который является родительским, в общем случае это не требуется, однако при решении многих задач поиска и модификации дерева позволяет их упростить, благодаря подобной ссылке мы можем перемещаться по дереву не только в направлении от корня к нижележащим элементам, но и в обратном порядке.

Под LIST\_OF\_NODES подразумевается некий контейнер способный хранить внутри себя достаточное количество ссылок на поддеревья. В случае если количество не велико и относительно стабильно, то можно воспользоваться обычным массивом. В случае, когда количество элементом может варьироваться в произвольном диапазоне, то рекомендуется применять один из ранее изученных контейнеров: очередь, дек. Если вы используете средства STL, то можно объявить ссылку на множество поддеревьев так:

1. Vector <MultiNode \*> children;
2. List <MultiNode \*> children;

В случае использования частного случая – бинарного дерева, объявление узла выглядит проще:

1. struct BinNode {
2. DATA\_TYPE data;
3. BinNode \* parent;
4. BinNode \* left , \* right;
5. };

Алгоритмы формирования деревьев

До текущего момента все рассмотренные алгоритмы предполагали, что у нас есть построенное двоичное или мульти-дерево. Теперь мы рассмотрим схему формирования деревьев с упором именно на практические направления. Одним из важнейших направлений использования бинарных деревьев является оптимизация задач поиска. Когда мы рассматривали тему поиска, то из двух способов – линейных поиск (затраты порядка O(n)) и двоичного поиска (затраты порядка O(log n)) очевидным было применение именно второго подхода. Однако в этом случае нам было необходимо выполнить сортировку массива данных в противном случае алгоритм не работал. В случае использования двоичных деревьев можно создать такой алгоритм его формирования, что задача поиска в нем нужного элемента также будет составлять порядка O(log n).

Деревья, для которых выполняется вышеописанное соотношение, называются деревьями поиска. В таком дереве, для того чтобы найти нужный элемент на каждом шаге пути необходимо принимать решение о выборе направления в зависимости от того, как соотносятся искомый элемент X и значение текущего узла Vi. Если Vi >= X то в таком случае поиск должен идти именно в левом поддереве, в противном случае в правом. Следует отметить, что получившееся двоичное дерево и соответственно эффективность операций поиска сильно зависит от степени упорядоченности поступающих наборов чисел.

В идеальном случае дерево должно быть сбалансированным. Напомню, что сбалансированным деревом является то, где количество узлов в правом и левом поддеревьях отличаются не более чем на 1. для повышения скорости работы алгоритма поиска в бинарном дереве также используют прием буфера (барьера). В этом случае все ветви будут заканчиваться не ссылкой на NULL, а ссылкой на этот буферный элемент

Операции над деревьями. Проход дерева.

Методы обхода дерева будем рассматривать на примере двоичного дерева изображенного на следующем изображении (следует отметить, что выбор в качестве пример именно бинарного дерева ни коим образом не ограничивает общности наших рассуждений).

Проход дерева представляет собой систематическое посещение всех его узлов. Наиболее известна схема прямого и обратного прохода.

Прямой проход предполагает, что первое обращение осуществляется к корню дерева, а затем по очереди выполняется рекурсивный проход вдоль каждого из поддеревьев выходящих из корневого элемента

Обратный проход

Данный проход в некотором смысле является обратным к алгоритму прямого прохода. Прежде всего, идет посещение всех дочерних узлов и только потом корневого.

В приведенном фрагменте кода предполагается, что каждый узел дерева имеет ряд методов для доступа к поддеревьям.

\* hasChildren – метод возвращает логическую величину – признак того есть ли у текущего узла поддеревья.

\* getCountChildren – возвращается количество подчиненных узлов.

\* getChildAtPosition – получение поддерева по порядковому номеру.

\* getStringPresentation – метод который возвращает строку представляющую (описывающую) текущий узел.

Разумеется, что конкретный код реализации данных методов зависит от того какие именно задачи, какую предметную область представляет данное дерево. Однако обратите внимание на то, что сначала мы выполняем проход по всем поддеревьям накапливая количество всех подузлов которые есть у текущего элемента дерева.

В качестве еще одного пример обратного прохода можно привести определение в каком-либо из файловых менеджеров (far, windows commander, Norton commander) объема занимаемого некоторой директорией. Очевидно, что объем директории состоит из объема дискового пространства для хранения служебной информации непосредственно о директории, также объема занимаемого всеми файлами которые находятся в каталоге, и внимание, объема который занимают подкаталоги. Следовательно, мы снова вынуждены выполнить проход по всем подкаталогам (на всех уровнях вложенности) чтобы определить объем текущего каталога.

Иные способы прохода деревьев

Метод прямого и обратного прохода наиболее часто встречается, однако есть и иные способы, например проход по глубине.

Как вы уже знаете каждый узел дерева имеет глубину (расстояние между ним и корневым элементом). В методе прохода по глубине сначала выполняется проход по все узлам, имеющим глубину X, и только потом выполняется переход к узлам с глубиной X + 1.

Исключение элементов из дерева

В дереве T необходимо удалить узел V. К сожалению, задача удаления не столь проста как операция добавления нового узла. В зависимости от того, какое именно количество непосредственных потомков (детей) есть у узла V возможны следующие стратегии:

элемент его левого поддерева или же на самый левый элемент его правого поддерева.

1. Если узел V не имеет потомков (терминальный узел) то надо просто освободить занимаемую им память, а родительской ссылке на V присвоить значение NULL.
2. Если узел V имеет только одного потомка, (например V->left != NULL, а V->right==NULL) в таком случае мы после освобождения ресурсов памяти занимаемых V должны будем родительской ссылке на него присвоить ссылку на потомка V.
3. V имеет двух потомков. В этом случае удаляемый элемент следует заменить на либо самый правый

**2.2 Описание программы**

Программа представляет собой шаблон класса со следующими функциями:

* find – выполняет поиск нужного элемента по его идентификатору и передаёт указатель на него;
* addNode – добавляет новый узел к указанному узлу, т.е. создает наследника указанному элементу;
* del – удаляет дерево;
* list – печатает элементы из листьев дерева;
* Derevo – конструктор класса, создаёт корень дерева;
* ~Derevo – деструктор класса, удаляет дерево

**2.3 Код шаблона класса**

template <typename myTip>

class Derevo

{

struct MultiNode

{

myTip data; //тип элементов дерева

int id; //идентификатор

MultiNode \* parent; //указатель на родительскую структуру

vector <MultiNode \*> \*children; //контейнер ссылок на потомков

};

int curr;

MultiNode \* find(int id1, MultiNode \* ukaz)

{

MultiNode \* temp = NULL;

if ((ukaz->id) == id1)

return ukaz;

else

{

if ((ukaz->children) == NULL)

return NULL;

for (int i = 0; i <= (ukaz->children)->size() - 1; i++)

{

temp = find(id1, (\*(ukaz->children))[i]);

if (temp != NULL)

return temp;

}

}

}

public:

MultiNode \* koren;

int addNode(myTip x, int id)

{

MultiNode \* newNode = new MultiNode;

MultiNode \* p = find(id, koren);

if (p->children == NULL)

p->children = new vector<MultiNode\*>;

(p->children)->push\_back(newNode);

newNode->parent = p;

newNode->id = ++curr;

newNode->data = x;

newNode->children = NULL;

cout << "Идентификатор этого элемента = " << curr << endl;

return curr;

}

void del()

{

for (int i = curr; i >= 0; i--)

{

MultiNode \* p = find(i, koren);

if (p->parent != NULL)

{

((p->parent)->children)->pop\_back();

if (((p->parent)->children)->size() == 0)

(p->parent)->children = NULL;

}

delete p;

}

curr = -1;

}

void list(MultiNode \* ukaz)

{

if (ukaz == NULL)

{

cout << "Дерево пусто =(" << endl;

return;

}

if ((ukaz->children) == NULL)

{

cout << ukaz->data << ' ' << endl;

}

else

{

for (int i = 0; i <= (ukaz->children)->size() - 1; i++)

list((\*(ukaz->children))[i]);

}

}

Derevo(myTip x)

{

koren = new MultiNode;

koren->data = x;

curr = 0;

koren->id = 0;

koren->parent = NULL;

koren->children = NULL;

cout << "Идентификатор корня = 0" << endl;

}

~Derevo()

{

del();

}

};

**2.4 Код основной программы**

int main()

{

setlocale(0, "");

int x;

cout << "Введите корень дерева" << endl;

cin >> x;

Derevo<int> tree(x);

cout << "Введите наследника дерева" << endl;

cin >> x;

tree.addNode(x, 0);

cout << "Введите наследника дерева" << endl;

cin >> x;

tree.addNode(x, 0);

cout << "Введите наследника дерева" << endl;

cin >> x;

tree.addNode(x, 0);

cout << "Введите наследника для элемента 1" << endl;

cin >> x;

tree.addNode(x, 1);

cout << "Введите наследника для элемента 2" << endl;

cin >> x;

tree.addNode(x, 2);

cout << "Введите наследника для элемента 3" << endl;

cin >> x;

tree.addNode(x, 3);

tree.list(tree.koren);

return 0;

}

**3 Заключение**

Вывод: в ходе работы была создана универсальная программа для создания любого дерева и вывода элементов из его листьев.