МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ОБНИНСКИЙ ИНСТИТУТ АТОМНОЙ ЭНЕРГЕТИКИ — филиал
федерального государственного автономного образовательного учреждения
высшего профессионального образования
«Национальный исследовательский ядерный университет «МИФИ»
(ИАТЭ НИЯУ МИФИ)

Факультет кибернетики
Кафедра компьютерных систем, сетей и технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1
Бинарное дерево

По курсу «Объектно-ориентированное программирование»

Студент
группы ВТ-С-Б12 ..... Луценко Г.А.

Руководитель
доцент
кафедры КССТ ..... Тельнов В.П.

Обнинск 2014

1 Постановка задачи

Разработайте в MS Visual Studio программное решение на языке Си, которое реализует динамическую структуру данных (контейнер) типа «Двоичное дерево». Каждый элемент контейнера содержит строки символов произвольной длины.

В программном решении следует реализовать следующие операции над контейнером:

- создание и уничтожение контейнера;
- добавление и извлечение элементов контейнера;
- обход всех элементов контейнера (итератор);
- удаление из контейнера дублирующих элементов;
- вычисление количества элементов в контейнере;
- реверс контейнера (первый элемент контейнера становится последним, второй элемент становится предпоследним и т.д.);
- объединение, пересечение и вычитание контейнеров;
- сохранение контейнера в дисковом файле и восстановление контейнера из файла.

Ограничения. Реализуйте простейший проект типа «приложение командной строки» (т.е. без оконного интерфейса). Средства C++ (объекты, классы, шаблоны классов) использовать не следует. Готовые контейнерные классы из библиотеки STL также использовать не следует. Разработайте контейнер самостоятельно на языке Си.

Рекомендации. Начните работу с изучения wiki:
https://github.com/djbelyak/OOPLab-Tree/wiki

Найдите и изучите в рекомендованной литературе и в документации MS Visual Studio описания и примеры реализаций данной структуры данных. Обдумайте и обсудите с преподавателем алгоритмы, состав функций, интерфейс и общую структуру программы. Возникающие затруднения пытайтесь преодолеть самостоятельно, потом обращайтесь за помощью.

Письменный отчет по работе должен содержать следующие разделы:

1. Постановку задачи.
2. Описание контейнера как динамической структуры данных, в том числе:
   - рисунки, на которых изображена структура данных и поясняются основные алгоритмы;
   - описание алгоритмов, которые используются при работе с контейнером;
   - область применения данной структуры данных, её преимущества и недостатки.

3. Листинг разработанного авторского кода на языке Си. Код должен быть надлежащим образом структурирован и снабжен комментариями.

Для успешной сдачи лабораторной работы необходимо представить письменный отчет, продемонстрировать на практике работоспособность программного решения и ответить на вопросы преподавателя.

2 Описание контейнера

Бинарное (двоичное) дерево (binary tree) - это упорядоченное дерево, каждая вершина которого имеет не более двух поддеревьев, причем для каждого узла выполняется правило: в левом поддереве содержатся только ключи, имеющие значения, меньшие, чем значение данного узла, а в правом поддереве содержатся только ключи, имеющие значения, большие, чем значение данного узла.

Бинарное дерево является рекурсивной структурой, поскольку каждое его поддерево само является бинарным деревом и, следовательно, каждый его узел в свою очередь является корнем дерева.

Узел дерева, не имеющий потомков, называется листом.

Схематичное изображение бинарного дерева:

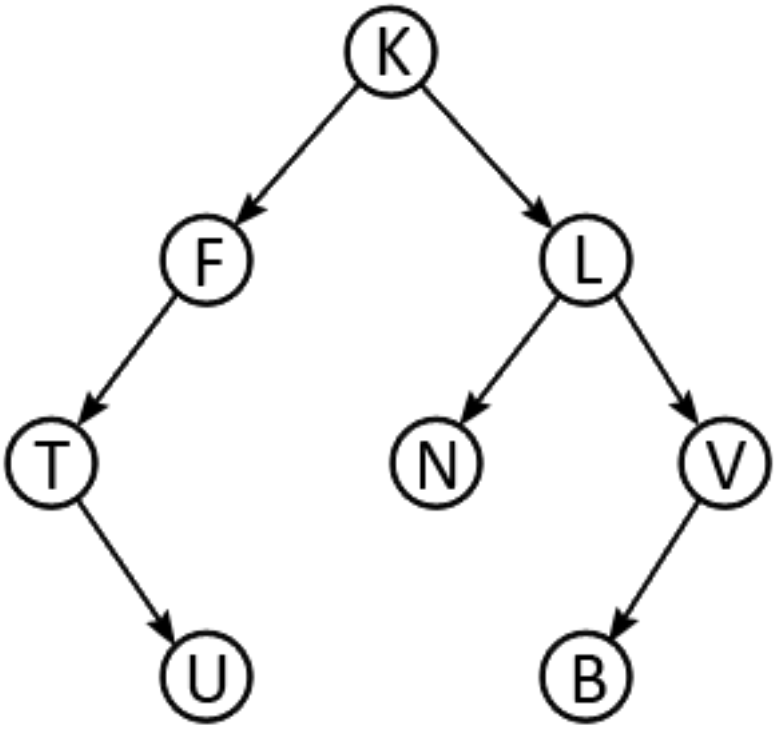

Рис. 1. Пример двоичного дерева

Организация данных с помощью бинарных деревьев часто позволяет значительно сократить время поиска нужного элемента. Поиск элемента в линейных структурах данных обычно

осуществляется путем последовательного перебора всех элементов, присутствующих в данной структуре. Поиск по дереву не требует перебора всех элементов, поэтому занимает значительно меньше времени. Максимальное число шагов при поиске по дереву равно высоте данного дерева, т.е. количеству уровней в иерархической структуре дерева.

Бинарное дерево является рекурсивной структурой, поскольку каждое его поддерево само является бинарным деревом и, следовательно, каждый его узел в свою очередь является корнем дерева.

При работе с деревьями обычно используются рекурсивные алгоритмы. Использование рекурсивных функций менее эффективно, поскольку многократный вызов функции расходует системные ресурсы. Тем не менее, использование рекурсивных функций является оправданным, поскольку нерекурсивные функции для работы с деревьями гораздо сложнее и для написания, и для восприятия кода программы.

Основные операции в бинарном дереве:

- обход дерева;
- добавление элемента;
- удаление элемента;
- поиск элемента;

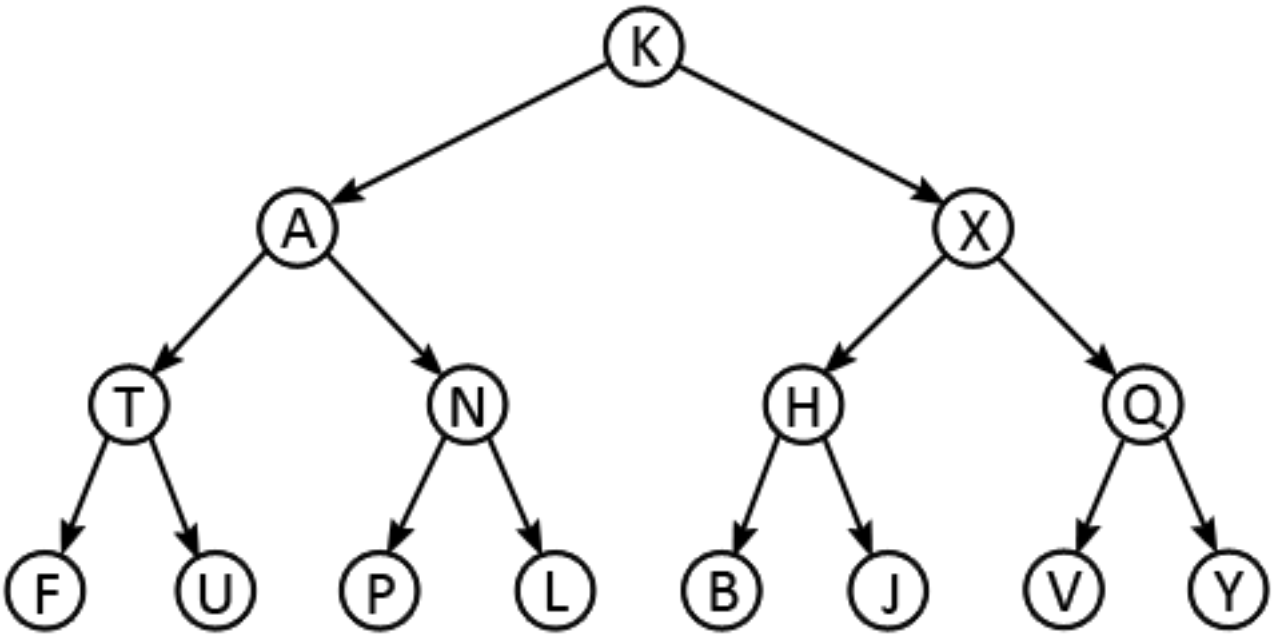

Рис. 2. Пример полного двоичного дерева

Операция, при которой вершины дерева поочередно просматриваются и каждая только один раз называется **обходом дерева**. Выделяют четыре основных метода обхода:

- обход в ширину;
- прямой обход;
- обратный обход;
- симметричный обход;

Обход в ширину – это поуровневая обработка узлов слева на право. Работа этого метода заключается в просмотре всех вершин, начиная с n-ого уровня и некоторой вершины.

Возьмем нулевой уровень за начальный (рис. 2), и, начиная с вершины K, будем методом обхода в ширину поочередно двигаться вниз, просматривая при этом вершины в следующем порядке: K A X T N H Q F U P L B J V Y.

Обход в прямом порядке вначале предполагает обработку предков, а потом их потомков, то есть сначала посещается вершина дерева, далее левое и правое поддеревья, именно в описанном порядке. Для нашего дерева последовательность прямого обхода такая: K A T F U N P L X H B J Q V Y.

Обход в обратном порядке противоположен прямому обходу. Первыми осматриваются потомки, а уже затем предки, иначе говоря, первоначально обращение идет к элементам нижних уровней левого поддерева, потом то же самое с элементами правого, и в конце осматривается корень. Обратный обход дерева с рисунка 2: F U T P L N A B J H V Y Q X K.

Обход в симметричном порядке заключается в посещении левого узла, перехода в корень, и оттуда в правый узел. Все для того же дерева узлы будут осмотрены в следующем порядке: F T U A P N L K B H J X V Q Y.

Узел бинарного дерева можно описать следующим образом:

Listing 1 – "Описание узла бинарного дерева на C++"

```
struct BTree
{
        T data;
        BTree* left;
        BTree* right;
};
```

Где T - тип данных хранимых внутри узла, a left и right - указатели на левое и правое дочерное поддерево узла.

Как в случае со списком, программа должна хранить указатель на первый элемент дерева, его вершину, который ещё называют **корнем дерева** (Root). В начале работы программы дерево пусто и корневой элемент может быть определён как *BTree* Root = NULL;*.

Добавление элемента в дерево.

Процедура добавления имеет следующий алгоритм работы:

1. если проверяемый узел пуст, то:
   (a) создаём новый узел;
   (b) левый и правый указатель указывают на пусто;

2. иначе выбираем ветку, по которой продолжим просмотр дерева. Если значение текущего элемента больше искомого, просматриваем левую ветку поддерева, в противном случае правую.

Удаление элемента из дерева.
Процедура удаления имеет следующий алгоритм работы:

1. Если текущий узел пуст, то остановиться
2. Иначе сравнить ключ искомого узла (K) с ключом текущего узла (T).
   (a) Если K>T, рекурсивно удалить искомый узел из правого поддерева.
   (b) Если K<T, рекурсивно удалить K из левого поддерева T.
   (c) Если K=T, то необходимо рассмотреть два случая:
      i. Если одного из детей нет, то значения полей второго ребёнка m ставим вместо соответствующих значений текущего узла, затирая его старые значения и освобождаем память, занимаемую узлом.
      ii. Если оба потомка присутствуют, то:
         A. найдём узел m, являющийся самым левым узлом правого поддерева;
         B. скопируем значения полей (ключ, значение) узла m в соответствующие поля узла n.
         C. у предка узла m заменим ссылку на узел m ссылкой на правого потомка узла m (который, в принципе, может быть пустым).
         D. освободим память, занимаемую узлом m (на него теперь никто не указывает, а его данные были перенесены в узел n).

Поиск элемента в дереве.
Процедура поиска имеет следующий алгоритм работы:

1. Если дерево пусто, то сообщить, что узел не найден, и остановиться.
2. Иначе сравнить искомый ключ (K) со значением ключа текущего узла (T).
   (a) Если K=T, выдать ссылку на этот узел и остановиться.
   (b) Если K>T, рекурсивно искать ключ K в правом поддереве T.
   (c) Если K<T, рекурсивно искать ключ K в левом поддереве T.

Может возникнуть вопрос, зачем нужны такие сложности, если можно просто хранить данные в виде списка или массива. Ответ прост — операции с деревом работают быстрее. При реализации списком все функции требуют O(n) действий, где n — размер структуры. Операции с деревом же работают за O(h), где h — максимальная глубина дерева (глубина — расстояние от корня до вершины). В оптимальном случае, когда глубина всех листьев одинакова, в дереве будет $n=2^h$ вершин. Значит, сложность операций в деревьях, близких к оптимуму будет

$O(\log_2 n)$. К сожалению, в худшем случае дерево может выродится и сложность операций будет как у списка, например, если вставлять значения в порядке их возрастания или убывания.

Бинарные деревья применяются не только для хранения и поиска информации, но и используются в различных алгоритмах сортировки и сжатия данных. Кроме того, существуют более продвинутые варианты бинарных деревьев, которые используются в компьютерной графике, в играх, при создание алгоритмов принятия решений и для множества других целей.

3 Листинг исходного кода

Listing 2 – container.h

```
#include <iostream>

struct BTree // описание узла дерева
{
        char* data; // данные узла (строка)
        BTree* left; // левое поддерево
        BTree* right; // правое поддерево
};

void FreeBTree(BTree*&); // уничтожение дерева
void InitBTree(BTree*&); // инициализация дерева
void AddBTreeNode(BTree*&, char*); // добавление узла в дерево
void InitBTreeNode(BTree*&, char*); // инициализация узла дерева
BTree* SearchBTree(BTree*, char*); // поиск элемента
```

Listing 3 – container.cpp

```
#include "container.h"

void FreeBTree(BTree* &Node) // уничтожение дерева
{
        if(Node != NULL)
        {
                // уничтожение поддеревьев
                FreeBTree(Node->left);
                FreeBTree(Node->right);
                // уничтожение данных и узла
                delete Node->data;
                delete Node;

                Node = NULL;
        }
}

void InitBTree(BTree* &Root) // инициализация дерева
{
        FreeBTree(Root); // уничтожаем и обнуляем корень
}

void InitBTreeNode(BTree* &Node, char* data) // инициализация узла дерева
{
        Node = new BTree;
        // инициализация данных
        Node->data = NULL;
        if(data != NULL)
```

```
        {
                Node->data = new char[strlen(data)];
                strcpy(Node->data, data);
        }
        // инициализация поддеревьев
        Node->left = NULL;
        Node->right = NULL;
}

void AddBTreeNode(BTree* &Node, char *data) // добавление узла в дерево
{
        if(Node == NULL) // если узел пуст
                InitBTreeNode(Node, data); // создаём новый узел на месте пустого
        else
        {
                if(strcmp(data, Node->data) < 0)  // если новый ключ "меньше" ключа текущего узла
                        AddBTreeNode(Node->left, data); // добавляем в левое поддерево
                else
                        AddBTreeNode(Node->right, data); // добавляем в правое поддерево
        }
}

BTree* SearchBTree(BTree* Root, char* data) // поиск элемента
{
        if(Root != NULL) // если узел не пуст
        {
                int compare = strcmp(data, Root->data); // сравнить ключи

                if(compare == 0)
                        return Root; // элемент найден

                if(compare < 0) // если искомый ключ меньше ключа текущего узла
                        return SearchBTree(Root->left, data); // продолжаем поиск в левом поддереве

                if(compare > 0)// если искомый ключ больше ключа текущего узла
                        return SearchBTree(Root->left, data); // продолжаем поиск в правом поддереве
        }

        return NULL; // элемент не найден
}
```

Listing 4 – stdafx.h

```
// stdafx.h: включаемый файл для стандартных системных включаемых файлов
// или включаемых файлов для конкретного проекта, которые часто используются, но
// не часто изменяются
//

#pragma once

#include <tchar.h>
#include <stdio.h>
#include <locale.h>
#include <iostream>
#include <conio.h>
// TODO: Установите здесь ссылки на дополнительные заголовки, требующиеся для программы
```

Listing 5 – main.cpp

```
#include "stdafx.h"
#include "container.h"
```

```
int main(int argc, char** argv)
{
        BTree* Root = NULL; // инициализация корня дерева
        InitBTree(Root); // инициализация дерева

        FreeBTree(Root); // уничтожение дерева
        return 0;
}
```