Бинарные деревья

Вообще, надо признаться, что тема иерархических структур весьма широка..

В качестве примера дерева возьмем двоичное дерево. Что такое дерево вообще? Это некий набор данных, которые указывают на другие данные. Динамический список. Кстати списки это частный вид дерева. Причем двоичного.

Дерево состоит из веток (узлов) и листьев (элементов). Листья – это узлы, которые не указывают ни на кого, у них нет веточек. Бинарное дерево – это дерево, в котором у ветки может быть не более двух листьев или веток. Отсюда и его название – “бинарное” значит “двоичное”, т.е. элементов два или меньше. Но никак не более двух.

У обычного дерева узлов у веток больше может быть.

Не буду вдаваться в классификации деревьев. Это тоже большой объем информации. Об этом можно почитать в Википедии, какие они бывают. Можно набрать в поиске слово “Граф” и почитать.

Основное правило формирования бинарного дерева в C++: **Если значение узла больше добавляемого – добавляется ветка справа, иначе создается ветка слева.**

Правило простое, от него и будем отталкиваться. Для этого нам понадобится структура, описываемая наподобие динамического списка: Поле данных и два указателя на правую и левую ветки. В динамических списках два указателя обычно связывают следующий и предыдущий элементы, в случае с деревом этого не понадобится, поскольку, как правило, проход по дереву идет с корня. Хотя конечно же может быть и обратная связь, если очень захочется.

```
1 struct Branch
2 {
3   char Data;
4   Branch *LeftBranch;
5   Branch *RightBranch;
6 };
```

Поле Data представляет данные, на основании которых строится дерево. Точнее один из элементов данных. Поля Branch описывают левую и правую ветки дерева, и являются указателями на такую же структуру.

Элементами дерева могут быть любые значения. Массивы, строки (строка это массив символов если что), другие деревья… Полей с данными может быть множество. На каждое поле можно строить свое дерево. Этот пример будет с массивом символов – строкой.

Сразу покажу основной код построения дерева, чтоб можно было понять что за источник данных взят в качестве примера:

```
1  int main()
2  {
3   Branch *Root = 0;
4   char s[] = "12384562789";
5
6   for (int i = 0; s[i]; i++)
```

```
 7  {
 8  Add(s[i], Root);
 9  }
10
11  print(Root);
12  FreeTree(Root);
13
14  cin.get();
15  return 0;
16 }
```

Строка. Обычная строка Си, которую будем разводить по веткам. Узлы и листы будут содержать символы, по кодам символов будем же определять вправо или влево строить дерево. Хоть я и взял в строку цифровые символы, это не важно.

Как должна работать программа? Возьмем символы “12384” из строки. Сначала программа видит, что дерево без корня. Его еще нет. Нужно посеять семя, чтоб оно выросло. Первый символ соответственно станет корнем. Второй символ – “2” будет сравниваться с корнем. По коду его значение больше. Значит у корня должна вырасти ветка вправо (RightBranch) в нашем случае. Далее идет тройка. У нас уже есть две ветки. Программа должна пройтись по ним, проверить: 3 больше 1? – Да. Пойти вправо. Там двойка. 3 больше 2? Так точно. После двойки нет веток, значит нужно создать ветку вправо для узла “2”.

Дальше идет восемь. Что с ним делать? То же самое. От единицы пройти через двойку до тройки и так же отрастить ветку право.

После восьмерки идет 4. Это значение пройдет корень. Поскольку оно больше, программа должна посмотреть, есть ли у корня ветка справа. Есть – пойти по ней. Пройти “2” и “3”, ибо 4 больше по значению. Дальше справа встретится число 8. Оно меньше. Значит “4” уже пойдет от восьмерки не вправо, а влево.

Простейший код построения дерева (с иерархической структурой) будет выглядеть так:

```
 1  void Add(int aData, Branch *&aBranch)
 2  {
 3   if (!aBranch)
 4   {
 5   aBranch = new Branch;
 6   aBranch->Data = aData;
 7   aBranch->LeftBranch = 0;
 8   aBranch->RightBranch = 0;
 9   return;
10  }
11  if (aBranch->Data>aData)
12  {
13  Add(aData, aBranch->LeftBranch);
14  }
15  else
16  {
17  Add(aData, aBranch->RightBranch);
18  };
19 }
```

Обратите внимание: В функции три основных условия:

1. Если узел-ветка не создана – создать его, наполнить данными и указать NULL (0) для его веток;

2. Если передаваемые данные больше чем данные текущего узла – попытаться построить его правую ветку, или если она уже есть просто пройти по ней;
3. То же самое для левой ветки, если данные меньше по значению.

Иерархия очень дружит с рекурсией. Рекурсивно вызывать функции, чтоб пройтись по графу, дереву или другому виду иерархической структуры – это самый простой способ. Применяется кстати в искусственном интеллекте. Представленный выше код проводит своего рода поиск, но не для выдачи результата, а чтобы найти местечко для элемента.

В данном случае чтоб понять принцип его построения не нужны тонны кода. Не обязательно рассматривать вставку между узлами, чтоб к примеру упорядочить дерево или разрабатывать код для балансировки дерева.

Код обхода дерева и вывода на экран

```
1  void print(Branch *aBranch)
2  {
3   if (!aBranch) return;
4   tabs++;
5
6   print(aBranch->LeftBranch);
7
8   for (int i = 0; i < tabs; i++) cout << "  ";
9   cout << aBranch->Data << endl;
10
11
12  print(aBranch->RightBranch);
13
14  tabs--;
15  return;
16 }
```

и освобождения дерева:

```
1 void FreeTree(Branch *aBranch)
2 {
3  if (!aBranch) return;
4  FreeTree(aBranch->LeftBranch);
5  FreeTree(aBranch->RightBranch);
6  delete aBranch;
7  return;
8 }
```

Вот здесь пожалуй виден еще один принцип работы с деревьями. В print() рекурсивно вызывается подряд переход по веткам. Причем между вызовами вставлен вывод данных узла, из которого растут ветки, чтоб правильно отобразить на экране или просто отработать скажем при поиске данных.

В освобождении так же происходит рекурсивный вызов освобождения веток. Сначала всю левую, потом всю правую с их подветками и листами, и только потом освобождается сам вызывающий узел. Это важно, поскольку если поставить delete в начале функции получим либо ошибку (что скорее всего) либо все растущие ветки из узла просто не смогут освободиться, зависнув в памяти мусором.

Красивый вывод:

```
1 for (int i = 0; i < tabs; i++) cout << "  ";
2  cout << aBranch->Data << endl;
```

Перед выводом данных текущего узла выводим некоторое количество отступов. Сделано это для красоты, чтоб на экране выглядело как настоящее дерево, а не просто набор данных. Чтоб можно было увидеть структуру воочию.

Для этого определяем глобальную переменную int tabs=0; которая будет считать количество отступов, и по факту, являться носителем значения уровня узла, т.е. номером его вложенности. Сколько у этого узла родителей-веток вплоть до корня. Это понятие тоже можно встретить в литературе про деревья. Так что оно там (в коде имеется ввиду) не только для наведения красоты на экране.

Вот полный код (при запуске дерево “растет” не сверху вниз, как на рисунках, а слева направо:

```
#include <iostream>
using namespace std;

int tabs = 0; //Для создания отступов
//Кол-во отступов высчитывается по кол-ву рекурсивного вхождения при выводе в фукцию print

//Структура ветки
struct Branch
{
 char Data; //Поле данных
 Branch *LeftBranch; //УКАЗАТЕЛИ на соседние веточки
 Branch *RightBranch;
};


//Функция внесения данных
void Add(char aData, Branch *&aBranch)
{
 //Если ветки не существует
 if (!aBranch)
 { //создадим ее и зададим в нее данные
 aBranch = new Branch;
 aBranch->Data = aData;
 aBranch->LeftBranch = 0;
 aBranch->RightBranch = 0;
 return;
 }
 else //Иначе сверим вносимое
 if (aBranch->Data>aData)
 { //Если оно меньше того, что в этой ветке - добавим влево
 Add(aData, aBranch->LeftBranch);
 }
 else
 { //Иначе в ветку справа
 Add(aData, aBranch->RightBranch);
 };
}

//Функция вывода дерева
void print(Branch *aBranch)
{
 if (!aBranch) return; //Если ветки не существует - выходим. Выводить нечего
 tabs++; //Иначе увеличим счетчик рекурсивно вызванных процедур
 //Который будет считать нам отступы для красивого вывода

 print(aBranch->LeftBranch); //Выведем ветку и ее подветки слева

 for (int i = 0; i<tabs; i++) cout << " "; //Потом отступы
 cout << aBranch->Data << endl; //Данные этой ветки


```

```
52  print(aBranch->RightBranch);//И ветки, что справа
53
54  tabs--; //После уменьшим кол-во отступов
55  return;
56 }
57
58 void FreeTree(Branch *aBranch)
59 {
60  if (!aBranch) return;
61  FreeTree(aBranch->LeftBranch);
62  FreeTree(aBranch->RightBranch);
63  delete aBranch;
64  return;
65 }
66
67
68 int main()
69 {
70  Branch *Root = 0;
71  char s[] = "18452789";
72
73  for (int i = 0; s[i]; i++)
74  {
75  Add(s[i], Root);
76  }
77
78  print(Root);
79  FreeTree(Root);
80
81  cin.get();
82  return 0;
83 }
```

На экране: