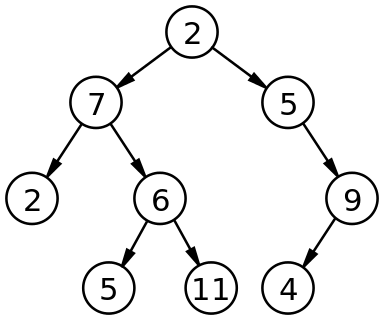
Бинарное дерево. Хм. Ну, деревьев я знаю много: дуб,липа,сосна,береза,клен и тд. А что значит бинарное? Наверное это дерево которое в компьютере!



Готово! Бинарное дерево посажено. Если честно. Это было самым простым заданием по проге.

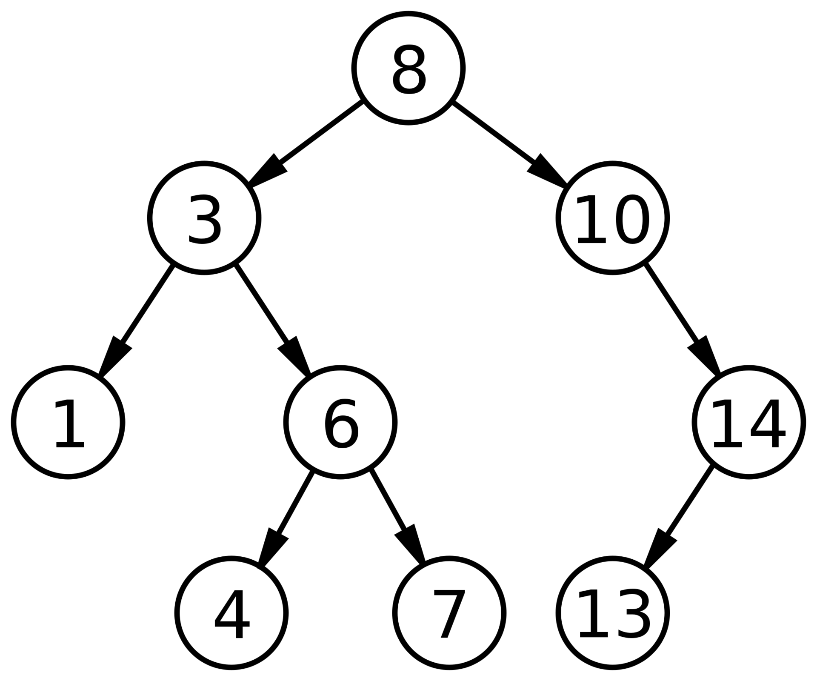
Ну а теперь серьёзно. Бинарное дерево считается сложным для понимания не подготовленных студентов и из всех рассмотренных ранее динамических структур деревья действительно будут сложнее, но мы не трусишки – сцикунишки! Сейчас мы разберем это бинарное дерево по веточкам!

Для начала разберемся с терминологией, а именно что такое Бинарное дерево и Двоичное дерево.

Двоичное дерево - иерархическая структура данных, в которой каждый узел имеет не более двух потомков. Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками. Двоичное дерево не является упорядоченным ориентированным деревом.  
  
Если коротко , то каждый элемент двоичного дерева указывает не более чем на 2 других элемента(на левый и правый), а так же элементы добавляться без какой либо системы в отличии от того же бинарного дерева. 

Как ты видишь правый и левый потомок двойки больше её самой. Чтобы двоичное дерево стало Бинарным деревом нужно ввести одно правило: Все элементы что больше идут вправо, элементы что меньше в лево.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде.



Как мы видим левее 8 нет ни одного элемента больше 8 , а правее нет ни одного элемента меньше. Это правило действует для каждого элемента Бинарного дерева.

Ну что? Не терпится увидеть код? Ну давай взглянем на него мельком.

#include "pch.h"

#include <iostream>

using namespace std;

struct Node

{

int x;

Node \*l, \*r;

};

void show(Node \*Tree)

{

if (Tree != NULL)

{

show(Tree->l);

cout << Tree->x << '\t';

show(Tree->r);

}

}

Node\* del(Node \*Tree) {

if (Tree != NULL)

{

del(Tree->l);

del(Tree->r);

delete Tree;

return NULL;

}

}

Node\* add\_node(int x, Node \*MyTree)

{

if (NULL == MyTree)

{

MyTree = new Node;

MyTree->x = x;

MyTree->l = MyTree->r = NULL;

return MyTree;

}

Node \*t = MyTree;

while (true)

{

if (t->x < x && t->r != NULL)

{

t = t->r;

}

else if(t->x > x && t->l != NULL)

{

t = t->l;

}

else

{

break;

}

}

if (t->x < x)

{

t->r = new Node;

t->r->x = x;

t->r->r = NULL;

t->r->l = NULL;

}

else

{

t->l = new Node;

t->l->x = x;

t->l->r = NULL;

t->l->l = NULL;

}

return MyTree;

}

int main()

{

Node \*Tree = NULL;

int a;

for (int i = 5; i > 0; i--)

{

cin >> a;

Tree = add\_node(i, Tree);

}

show(Tree);

cout << '\n';

Tree = del(Tree);

for (int i = 20; i > 5; i--)

{

cin >> a;

Tree = add\_node(i, Tree);

}

show(Tree);

Tree = del(Tree);

}

Воу, неплохо. Как-то резко перехотелось учить деревья. Рекурсии какие-то, l,r – СЛОЖНО!

Ну,не спеши сдаваться! Сейчас построчно все разберем и тогда все станет понятно(Надеюсь).

И так. Первым делом нас встречает описание нашего элемента дерева:

struct Node

{

int x;

Node \*l, \*r;

};

Угу,ну это мы видели.Стек,списки.Но там было по 1 указателю , а тут целых 2!

Верно, если изображать элемент дерева графически то он будет отличаться от элемента стека или списка:

Стэк или Список:

Вершина

Дерево: Корень

Угу, значит в отличие от Стека и Списка элемент дерева указывает на 2 элемента вместо 1-ого, а вершина(Первый элемент дерева) называется корнем. И как понять какой элемент нужно в право отправлять , а какой в левый.

Какой элемент куда отправлять зависит от типа дерева! В двоичном дереве мы сами решаем куда кокой элемент помещать , а вот в бинарном есть свое правило! В право идет все что больше, в лево все что меньше.

Давайте введем некую последовательность чисел и сформируем бинарное дерево на основе этих данных.

Вводим: 3,4,1,7,6,2

Первый элемент называется – корень.

3

Далее нам нужно записать 4. 4>3 , значит 4 записывается вправо.

3

4

Далее добавляем 1. 1<3, значит 1 записываем влево.

3

4

1

Далее добавляем 7. 7>3, значит добавляем вправо, но справа записана 4.Если справа что-то записано , то мы проводим точно такое же сравнение для записанного числа. 7>4 , значит мы записываем 7 вправо для элемента 4!

3

4

1

7

6>3,значит идем вправо.Там стоит 4.6>4 , значит идем вправо,там стоит 7.7>6, значит 6 идет влево для элемента с 7.

6

3

4

1

7

Добавим 2. 2<3, значит идем влево.1<2 , значит 2 идет вправо для элемента с 1.

6

7

3

3

6

7

1

4

2

Вух, вроде справились. Мы можем собой гордится! Мы правильно заполнили Бинарное дерево поиска. Вот только…

А нафига? Зачем его вообще нужно так заполнять? Что нам это дает?

Ну , давайте попробуем найти какой нибудь элемент в этом дереве.К примеру 6.  
  
Для поиска элемента мы можем использовать следующий алгоритм. Если число которое мы ищем больше того на котором мы находимся идем вправо, если меньше , то идем влево.

Мы на 3-ке. 3<6, значит идем вправо и оказываемся на 4-ке.4<6 ,значит идем вправо и оказываемся на 7-ке.7>6 , значит идем влево и оказываемся на 6. О Чудо! Мы нашли 6! И потратили на это всего 3 операции.

Бинарное дерево поиска хорошо тем , что при помощи сравнений < > мы можем сразу сказать что если мы стоим на 3 и ищем 6 , то слева 6 точно не будет. Тем самым мы ускоряем процесс поиска нужно числа так как не смотрим туда , где этого числа точно нет.

В простом массиве 3,4,1,7,6,2 нам потребовалось бы на 1 операцию больше. А если бы до 6 было бы 100 элементов которые меньше его по значению? Потребовалось бы весь массив пройти. А в бинарном дереве эти 100 элементов < 6 сразу бы отбросились бы. Ну не зря же его называют Бинарным деревом поиска.

С теорией добавления разобрались. Теперь разберемся с кодом:

Node\* add\_node(int x, Node \*MyTree)

{

if (NULL == MyTree)//Если указатель корня ни на что не указывает,то создаем корень и возвращаем его

{

MyTree = new Node; //Выделяем память

MyTree->x = x; //Заполняем информационную часть

MyTree->l = MyTree->r = NULL; //Новый элемент никуда не указывает

return MyTree; //Возвращаем указатель на корень

}

Node \*t = MyTree; //Создаем указатель при помощи которого будем идти по дереву

while (true) //Идем по узлам пока не найдем нужный

{

if (t->x < x && t->r != NULL) //Если число которое мы хотим записать больше того на котором мы находимся и справа есть какой-то элемент , то идем вправо

{

t = t->r;

}

else if(t->x > x && t->l != NULL) //Если число которое мы хотим записать меньше того на котором мы находимся и слева есть какой-то элемент , то идем влево

{

t = t->l;

}

else//Мы пришли на последний элемент, завершаем цикл

{

break;

}

}

if (t->x < x) //Если число которое мы хотим записать больше, то записываем его вправо

{

t->r = new Node;

t->r->x = x;

t->r->r = NULL;

t->r->l = NULL;

}

else //Если число которое мы хотим записать больше, то записываем его влево

{

t->l = new Node;

t->l->x = x;

t->l->r = NULL;

t->l->l = NULL;

}

return MyTree; // возвращаем вершину

}

Ух,как много кода. Значит нужно идти до самого низкого элемента и каждый элемент проверять чтобы знать куда идти: вправо или влево. Если не очень понятно,то представьте что вместо x вы вставляете числа из предыдущего примера 3,4,1,7,6,2 и посмотрите как графически это выглядит.

Так. Начало положено и начало довольно бодрое. Давайте устроим небольшой перерыв на минут 10 ,а после продолжим. И пока ваш чаек заваривается можете почитать пару шуток:

1. Самое главное отличие C от C++: на Си вы можете делать ошибки, а в C++ — еще и наследовать их.
2. Если вы хотите получать пользу от программирования. никогда не программируйте!
3. Секрет написания хорошего кода очень прост. Но секрет есть секрет.
4. -А у меня вчера друг сервер за 5 минут сломал.

-Он хакер?

-Нет. Он мудак!



Ух. Вот это шутки. Ладно, вылезайте из под стола. Продолжаем.

Следующая функция – это функция для просмотра дерева. Если вы дойдя до этого момента не знаете что такое рекурсия , то срочно заполняйте этот пробел ,ибо для просмотра дерева используется рекурсия(Просмотр без рекурсии это вообще извращенство с танцами под бубен).

void show(Node \*Tree)

{

if (Tree != NULL)

{

show(Tree->l);

cout << Tree->x << '\t';

show(Tree->r);

}

}

Функция маленькая , но для её понимания мне придётся много порисовать ☹

Возьмем наше старое дерево из примера:

2

3

6

7

1

4

Запускаем нашу функция   
  
show(Tree); //Tree = указатель на элемент с 3-ой

Three

4

1

7

6

3

2

Three указывает на что-то ,значит срабатывает условие:

if (Tree != NULL)

{

show(Tree->l);

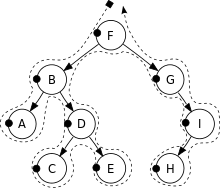
cout << Tree->x << '\t';

show(Tree->r);

}

И что же происходит внутри if? А внутри if происходит рекурсия. И что делает эта рекурсия? Она помещает элементы левой ветки элемента с номером 3 в аппаратный стек(Подробнее можно почитать про аппаратный стек в теме «Рекурсия») , а после этого выводит значения левой ветки. После вывода левой ветки начинается загрузка в аппаратный стек правой ветки и выгрузка её левой , а потом правой части каждого элемента.

Сложно? Согласен. Для лучшего понимания нужно хорошо выучить рекурсию. Задержитесь на этом моменте и попробуйте создать пару деревьев и запустите этот алгоритм просмотра. Посмотрите какие результаты вам выведет и пройдитесь по коду пошагово(f11) для лучшего понимания. Путь по дереву будет выглядеть примерно вот так:



Ну что? Надеюсь ты позапускал код и посмотрел как выводятся значения при просмотре дерева. Осталось уже немного , только удаление дерева. А удаление – это тот же самый просмотр дерева только с удалением элемента на который пришли. Если просмотр дерева вам понятен, то удаление не вызовет никаких сложностей в понимании.

Node\* del(Node \*Tree) {

if (Tree != NULL)

{

del(Tree->l);

del(Tree->r);

delete Tree;

return NULL;

}

}

Тут без комментариев. Добавляем все элементы в аппаратный стек , а после удаляем по очереди.

Ну вот. ПО теории у меня все. Если вы дошли до уровня деревьев – значит у вас уже должен быть опыт и вы должны понимать, что практика лучше любой теории! Деревья – это тот случай когда нужно запустить код и поиграться с ним в различных вариантах.