КАТЕДРА: КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ
ДИСЦИПЛИНА: АЛГОРИТМИ И СТРУКТУРИ ОТ ДАННИ

ЛАБОРАТОРНО УПРАЖНЕНИЕ № 10

ТЕМА: Двоични дървета за търсене.

ЦЕЛ:

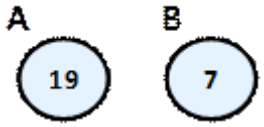
Целта на упражнението е студентите да се запознаят с наредени двоични дървета. След упражнението студентите би следвало да могат да използват тези дървета като структури от данни.

I. ТЕОРЕТИЧНА ЧАСТ

В това упражнение се разглежда един по-специфичен клас двоични дървета – наредените. Те използват едно често срещано при двоичните дървета свойство на върховете, а именно съществуването на **уникален идентификационен ключ** във всеки един от тях. Този ключ не се среща никъде другаде в рамките на даденото дърво. Друго основно свойство на тези ключове е, че са сравними. Наредените двоични дървета позволяват бързо (в общия случай с приблизително *log(n)* на брой стъпки) търсене, добавяне и изтриване на елемент, тъй като поддържат елементите си индиректно в сортиран вид.

Върховете на едно дърво могат да съдържат най-различни полета. В по-нататъшното разсъждение ще се интересуваме само от техните уникални ключове, които трябва да са сравними.

Пример. Нека са дадени два конкретни върха A и B:



Фиг.1. Два върха в дървета за търсене

В примера ключът на A и B са съответно целите числа 19 и 7. Както е известно от математиката, целите числа (за разлика от комплексните например) са сравними, което позволява те да се използват като ключове. Затова за върховете A и B може да се каже, че "A е по-голямо от B" тъй като "19 е по-голямо от 7".

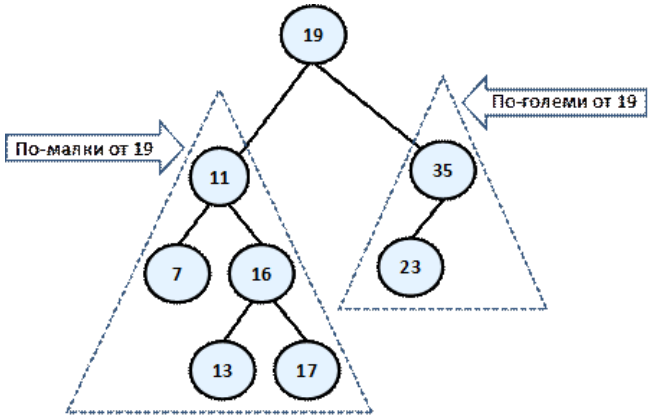
Забележка! Този път числата изобразени във върховете са техни уникални идентификационни ключове, а не както досега произволни числа.

Наредено двоично дърво (дърво за търсене, binary search tree) е двоично дърво, в което всеки връх има уникален ключ, всеки два от ключовете са сравними и което е организирано така, че за всеки връх да е изпълнено:

- Всички ключове в лявото му поддърво са по-малки от неговия ключ.
- Всички ключове в дясното му поддърво са по-големи от неговия ключ.

Тези дървета представляват специфичен тип структури, които складират „елементи“ (като числа, текст и т.н.) в паметта. Те позволяват бързо търсене, добавяне и премахване на елементи и могат да бъдат използвани за имплементирането на динамични множества от елементи или речници, които позволяват за бързото търсене по ключ (например

намиране на телефонния номер на човек по името му). На фиг. 2. е показано едно наредено двоично дърво за търсене:



Фиг.2. Наредено двоично дърво за търсене

Двоичните дървета за търсене пазят ключовете си в подреден ред, за да могат приложените операции да използват принципа на бинарното търсене: когато се търси ключ в дърво (или място за поставяне на нов ключ), те обхождат дървото от корен до листо, като по пътя проверяват търсената стойност със стойностите във възлите на дървото и спрямо проверката избират да продължат по лявото или по дясното поддърво. Средностатистически това значи, че всяка проверка позволява пропускането на около половината от дървото, така че всяко търсене, добавяне или изтриване отнема време пропорционално на логаритъм от броя на елементи пазени в дървото. Това е много по добре от линейното време нужно за да се намери елемент по ключ в (неподреден) масив.

Процедурата за търсене на елемент в такова дърво може лесно да се обобщи в няколко стъпки:

1. *Започваме от корена.*
2. *Ако дървото е null т.е. не съществува, стойността не е намерена и търсенето е приключило. В противен случай преминаваме към стъпка 3.*
3. *Сравняваме търсената стойност със стойността на текущия възел.*
4. *Ако двете стойности съвпадат, значи търсенето е приключило. В противен случай, продължаваме към стъпка 5.*
5. *Ако търсената стойност е по-малка от стойността на текущия възел, тогава следваме лявата връзка на текущия възел и се връщаме на стъпка 2.*
6. *Ако търсената стойност е по-голяма, тогава продължаваме търсенето следвайки дясната връзка на възела и се връщаме на стъпка 2.*

Алгоритъмът за търсене на стойност, може да се изрази рекурсивно или итеративно, но и в двата случая, в повечето имплементации на двоично дърво, като резултат получаваме или възела, който съдържа търсената стойност или null, ако стойността не е намерена.

След добавяне на нов елемент, дървото трябва да запази своята нареденост. Алгоритъмът е следният: ако дървото е празно, то добавяме новия елемент като корен. В противен случай:

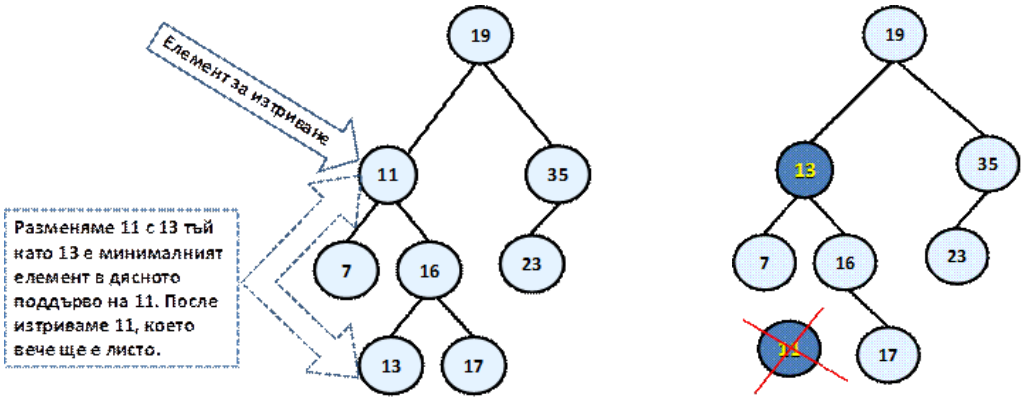
- Ако елементът е по-малък от корена, то се обръщаме рекурсивно към същия метод, за да включим елемента в лявото поддърво.
- Ако елементът е по-голям от корена, то се обръщаме рекурсивно към същия метод, за да включим елемента в дясното поддърво.
- Ако елементът е равен на корена, то не правим нищо и излизаме от рекурсията.

Ясно се вижда как алгоритъмът за включване на връх изрично се съобразява с правилото елементите в лявото поддърво да са по-малки от корена на дървото и елементите от дясното поддърво да са по-големи от корена на дървото.

Изтриването е най-сложната операция от трите основни. След нея дървото трябва да запази своята наредеността. Първата стъпка е да се намери елемента за изтриване от дървото, като се използва алгоритъма за търсене. След това се прави следното:

- Ако върхът е листо – насочваме референцията на родителя му към null. Ако елементът няма родител следва, че той е корен и просто го изтриваме.
- Ако върхът има само едно поддърво – ляво или дясно, то той се замества с корена на това поддърво.
- Ако върхът има две поддървета. Тогава намираме най-малкият връх в дясното му поддърво и го разменяме с него. След тази размяна върхът ще има вече най-много едно поддърво и го изтриваме по някое от горните две правила.

Разгледайте следния пример за изтриване: Нека се изтрие елемент с ключ 11 от дървото на фиг.2.



Фиг.3. Изтриване на елемент с ключ 11 от дървото

Наредените двоични дървета представляват една много удобна структура за търсене. Те обаче имат един основен недостатък възможно е да съществува наредено двоично дърво в което едно поддърво да има прекалено много върхове а в другото да са доста малко. Така в зависимост от това в кое под дърво се намира търсения елемент то в единия случай може да отнеме доста операции а за другия прекалено малко.

За да се избегне този недостатък се използват балансирани двоични дървета. При тях никое листо не е на „много по – голяма“ дълбочина от всяко друго листо. Дефиницията на „много по – голяма“ зависи от конкретната балансираща схема.

Съществуват и идеално балансирани двоични дървета. Това са тези дървета при които, разликата в броя на върховете на лявото и дясното поддърво на всеки от върховете е най – много единица. Трябва да се има в предвид че такива дървета рядко се постигат и често се използват приблизително балансирани дървета.

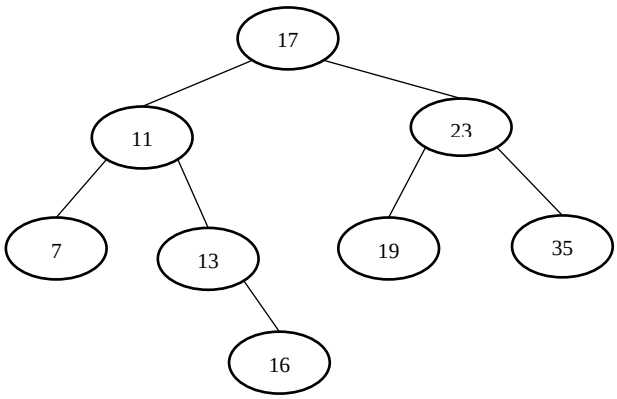
Често при прилагане на операциите добавяне и изтриване на елемент върху тях се получава дисбаланс на дървото затова е необходимо при всяка така операция да се извърша и операция за ре – балансиране. Има най различни такива ротации които извършват ре – балансирането на дърветата. Те се уточняват допълнително и зависи от конкретната структура от данни. Като примери за такива структури, може да се даде червено-черно дърво(Red – Black Tree), AVL-дърво, AA-дърво, и др. AVL-дърветата са идеално балансирани дървета, но те работят най – бавно. И са сложни за имплементиране. Red-Black – дърветата са приблизително балансирани дървета. Разликата на върховете при двете поддървета е до 2-пъти. При тях имплементацията

също е сложна. АА – дърветата също приблизително балансирани дървета, но те са по лесни за имплементация. При всички тези дървета имат свойството да изпълняват основните операции (добавяне, търсене и изтриване на елемент).

II. ПРАКТИЧЕСКА ЧАСТ

ЗАДАЧА1: *Дървото на фиг.2. не е идеално балансирано. Направете го идеално балансирано.*

РЕШЕНИЕ:



Фиг.4. Решение на задачата

ПОЯСНЕНИЕ

Броя на върховете за лявото и дясното поддърво на всеки връх е с разлика максимум единица. Поради тази причина дървото отговаря на свойството за идеално балансирано дърво.

ЗАДАЧА2: *Да се създаде програма която построява двоичното дърво на фигура 4 като се добавят постепенно елементите в него започвайки от корена.*

РЕШЕНИЕ:

```
……..
void add1(int n, elem* &t)
{
    if (t == NULL)
    {
        t = new elem;  t->key = n;
        t->left = t->right = NULL;
    }
    else
    {
        if (t->key < n)
            add1(n, t->right);
        else
            add1(n, t->left);
    }
}
……..
int main()
{
    add(17);
    add(11);
    add(23);
    add(7);
…….}
```

ПОЯСНЕНИЕ

Функцията add1 добавя елемент към дървото като търси подходящото място където да го добави. Ако дървото е празно добавя елемента като корен на дървото. Ако не е празно проверява добавящия елемент дали е по – голям от корена. Ако да се вика дясното под дърво, ако не се вика лявото под дърво. Под дърветата се викат рекурсивно и се изпълнява същата процедура върху тях. В main функцията се вика функция add към която се добавя новия елемент. Тя от своя страна би следвало да вика функцията add1.

III. Задача за самостоятелна работа.

1. Тествайте задача 2 от практическата част, като довършите липсващата част. След като подадете всички върхове пробвайте да добавите връх 17 отново. Какво ще се случи с дървото? Къде ще се добави този елемент?
2. Модифицирайте програмата така че да обходи дървото префиксно, инфиксно и постфиксно. Анализирайте резултата.
3. Добавете функция която да изключва елемент от дървото. Изключете елемент 17 два пъти. След като изключите елемента обходете дървото за да видите резултата. Какво ще се случи ако опитате да изключите елемент който не наличен в дървото?