

САА – Упражнение 12

Алгоритми за търсене на елементи в масив

1. Търсене на елемент в несортиран масив

- Алгоритъм с последователно обхождане на всички елементи в масива. Елементите на масива се обхождат последователно, докато се достигне края на масива или се открие търсения елемент. В най-тежкия случай този алгоритъм ще използва $2n$ сравнения, където n е броят на елементите в масива. На всяка стъпка ще се правят 2 проверки:
 1. за достигане на край на масива;
 2. за откриване на търсения елемент.
- Алгоритъм с добавяне на нов елемент. В края на масива се добавя нов елемент със стойност, равна на търсената. По този начин се избягва проверка 1 от предишния алгоритъм и така броя на сравненията намалява до $n+1$.

2. Търсене на елемент в сортиран масив

- Алгоритъм с последователно обхождане. Елементите на сортирания масив се обхождат последователно, докато се достигне края на масива или елемент, който е по-голям от търсения. В най-тежкия случай този алгоритъм ще използва $2n$ сравнения.
- Двоично търсене. Нека разглеждания масив има начален индекс l и краен индекс r . От масива се избира елемента със среден индекс – т. е. $(l + r)/2$ и се сравнява с търсения елемент. Ако съвпадат – търсеният елемент е намерен. В случай, че не съвпадат са възможни два варианта:
 1. Търсеният елемент е по-малък от елемента със среден индекс – тогава търсенето продължава в лявата част с двойно по-малък размер: начало l и край $(l + r)/2 - 1$.
 2. Търсеният елемент е по-голям от елемента със среден индекс – тогава търсенето продължава в дясната част с двойно по-малък размер: начало $(l + r)/2 + 1$ и край r .

Сложността на алгоритъма за двоично търсене е $O(\log_2 n)$. Това е най-бързият алгоритъм за търсене в сортиран масив.

Програмна реализация на функцията за двоично търсене:

```
int BinSearch(int l, int r, float x, float A[])
{
    int m;
    while(l <= r)
    {
```

```

        m=(l+r)/2;
        if (x<A[m])
            r=m-1;
        else
            if (x>A[m])
                l=m+1;
            else
                return m;
    }
    return -1;
}

```

Двоични търсещи дървета

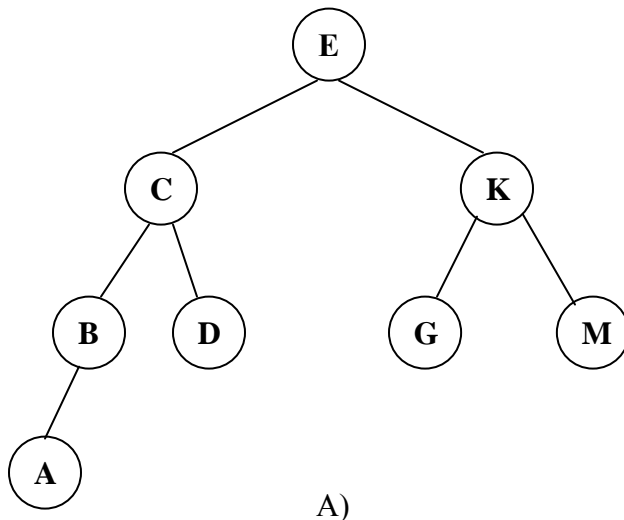
Двоичното търсещо дърво е двоично дърво със следните характеристики:

- 1) всички върхове в лявото поддърво на даден връх са с ключове, чиито стойности са по-малки от ключа на този връх;
- 2) всички върхове в дясното поддърво на даден връх са с ключове, чиито стойности са по-големи или равни на ключа на този връх;
- 3) левите и десните поддървета за даден връх също трябва да са двоични търсещи дървета.

Пример за двоично търсещо дърво е показан на следващата фигура. Всеки връх в това дърво може да бъде представен чрез структура с три полета: ключ от символен тип и два указателя съответно към ляво и дясно поддърво. Според определението за двоично търсещо дърво, за ключовете на връх *v* и неговите преки наследници са в сила следните условия:

$$(v \rightarrow \text{key} > v \rightarrow \text{left} \rightarrow \text{key})$$

и

$$(v \rightarrow \text{right} \rightarrow \text{key} \geq v \rightarrow \text{key})$$


```

struct tree
{
    char key;
    struct tree *left;
    struct tree *right;
};

```

Б)

Търсене на елемент в двоично търсещо дърво

Търсенето на елемент по ключ в двоично търсещо дърво започва от корена. За текущия елемент, зададен чрез указател, проверяваме:

- 1) Ако указателят има стойност *NULL*, приключваме с търсенето, а резултатът е, че ключът, който търсим, не се намира в дървото;
- 2) Ако ключът, който търсим, е по-малък от ключа на текущия елемент, продължаваме рекурсивно търсенето в лявото поддърво;
- 3) Ако ключът, който търсим, е по-голям от ключа на текущия елемент, продължаваме рекурсивно търсенето в дясното поддърво;
- 4) Ако има съвпадение между ключът, който търсим и ключа на текущия елемент, търсенето приключва с намерен елемент.

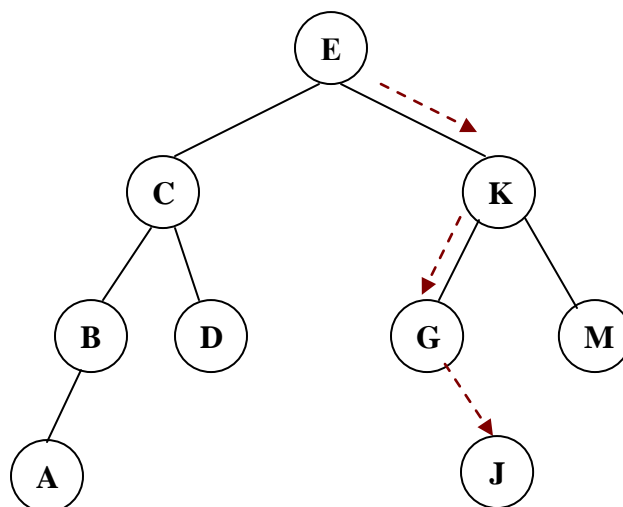
Добавяне на елемент към двоично търсещо дърво

Добавянето на елемент в двоично търсещо дърво се извършва аналогично на търсенето, като целта в този случай е при обхождането на върховете да се стигне до указател със стойност *NULL*. Този указател ще насочим към елемента, който ще добавим в дървото. За всеки текущ елемент от дървото, зададен чрез указател проверяваме:

- 1) Ако ключът на елемента, който ще добавяме е по-малък от ключа на текущия елемент, продължаваме рекурсивно в лявото поддърво;
- 2) Ако ключът на елемента, който ще добавяме е по-голям от ключа на текущия елемент, продължаваме рекурсивно в дясното поддърво;
- 3) Ако има съвпадение между ключът на елемента, който ще добавяме и ключа на текущия елемент, следва че елементът вече е бил добавен в дървото;
- 4) Ако сме достигнали до указател със стойност *NULL*, следва че сме намерили позицията на новия елемент. В този случай ще заделим необходимата памет, ще инициализираме всички полета и ще пренасочим указателите.

Нека като пример разгледаме как ще добавим елемент със стойност 'J' към двоичното търсещо дърво от следващата фигура. Започваме от корена.

- 1) сравняваме ключа на новия елемент J с ключа на корена. Тъй като $J > E$, продължаваме в дясното поддърво;
- 2) сравняваме J и K. Тъй като $J < K$, продължаваме в лявото поддърво;
- 3) сравняваме J и G. Тъй като $J > G$, преминаваме към дясно поддърво;
- 4) текущият указател има стойност *NULL*, от което следва, че сме намерили позицията на новия елемент и го добавяме като десен наследник на G.



Изтриване на елемент от двоично търсещо дърво

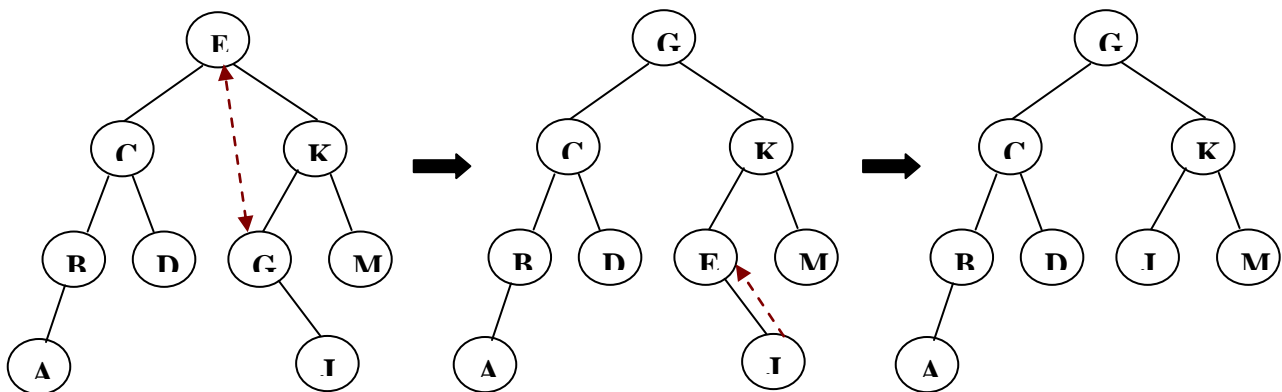
За да изтрием елемент от двоично търсещо дърво, първо трябва да намерим неговата позиция в дървото и след това да го изключим от структурата. При изтриване на елемент от дървото са възможни три случая:

1) елементът, който трябва да бъде изтрит е листо. Действията, които са необходими в тази ситуация, са: освобождаване на заетата от елемента памет и задаване на стойност *NULL* на указателя, който е сочел към този елемент.

2) елементът, който трябва да бъде изтрит има само ляво или само дясно поддърво. В този случай премахваме елемента от дървото като на негово място поставяме неговия пряк наследник.

3) елементът, който трябва да бъде изтрит има две (ляво и дясно) поддървета – това е най-тежкият случай. Нека означим с *d* елементът, който трябва да изтрием. Трябва да разменим *d* с неговия инфиксен предшественик (най-десния връх в лявото му поддърво) или инфиксен наследник (най-левия връх в дясното му поддърво). След размяната *d* ще има най-много един пряк наследник и ще може да бъде изтрит, като се приложи някое от горните две решения.

Нека разгледаме как ще изтрием корена на дървото от следващата фигура. Коренът има два преки наследника: ляв и десен. За да го изтрием, ще трябва да го разменим с неговия инфиксен предшественик – връх D или с инфиксния му наследник – връх G. Избираме да използваме връх G за размяната. Така първо разменяме E и G. След размяната върхът с ключ E има единствен наследник – връх J. Този път заменяме E с J и накрая изтриваме върха с ключ E.



Задачи:

1. Реализирайте програмно като отделни функции всеки от алгоритмите за търсене (нерекурсивни).
2. Подредете алгоритмите по възходящ ред според тяхната сложност.
3. Докажете сложността на алгоритъма за двоично търсене.
4. Напишете функция, която реализира програмно алгоритъм за търсене в несортиран списък.

! Домашна работа

Да се реализират програмно функциите за добавяне, търсене и изтриване на елемент в двоично търсещо дърво.