Задачи за задължителна самоподготовка
по
Структури от данни в програмирането

Калин Георгиев
kalin@fmi.uni-sofia.bg

8 ноември 2018 г.

Съдържание

1 Линеен едносвързан и двусвързан списък

1.1 Представяне на двусвързан списък

Възел на линеен двусвързан списък представяме със следния шаблон на структура:

```
template <class T>
struct dllnode
{
  T data;
  dllnode<T> *next, *previous;
};
```

Освен ако не е указано друго, задачите по-долу да се решат като се реализират методи на клас `DLList` със следния скелет:

```
template <class T>
class DLList
{
  //...
  private:
  dllnode<T> *first, *last;
};
```

Преди да пристъпите към задачите, реализирайте подходящи контруктори, деструктор и оператор за присвояване на класа.

Следните задачи да се решат като упражнение за директно боравене с възлите на линеен двусвързан списък. Функциите (методите) да се тестват с подходящи тестове.

1.1. Да се дефинира функция `int count(dllnode<T>* l,int x)`, която преброява колко пъти елементът `x` се среща в списъка с първи елемент `l`.

1.2. Фунцкция `dllnode<int>* range (int x, int y)` която създава и връща първия елемент на списък с елементи $x, x+1, ..., y$, при положение, че $x \leq y$.

1.3. Да се дефинира функция `removeAll (dllnode<T>*& l,const T& x)`, която изтрива всички срещания на елемента `x` от списъка `l`.

1.4. Да се дефинира функция `void append(dllnode*<T>& l1, dllnode<T>* l2)`, която добавя към края на списъка l_1 всички елементи на списъка l_2. Да се реализира съответен оператор `+=` в класа на списъка.

1.5. Да се дефинира функция `dllnode* concat(dllnode<T>* l1, dllnode<T>* l2)`, който съединява два списъка в нов, трети списък. Т.е. `concat`(l_1, l_2) създава и връща нов списък от елементите на l_1, следвани от елементите на l_2. Да се реализира съответен оператор `+` в класа на списъка.

1.6. Да се дефинира функция `reverse`, която обръща реда на елементите на списък. Например, списъкът с елементи $1, 2, 3$ ще се преобразува до списъка с елементи $3, 2, 1$.

1.7. Да се напише функция `void removeduplicates (dllnode *&l)`, която изтрива всички дублиращи се елементи от списъка l.

1.2 Списъци и сложности

Функцията `std::clock()` от `<ctime>` връща в абстрактни единици времето, което е изминало от началото на изпълнение на програмата. Обикновено тази единица за време, наречена “`tick`”, е фиксиран интервал “реално” време, който зависи от хардуера на системата и конфирграцията ѝ. Константата `CLOCKS_PER_SEC` дава броя `tick`-ове, които се съдържат в една секунда реално време.

Чрез следния примерен код може да се измери в милисекунди времето за изпълнение на програмния блок, обозначен с “...”.

```
clock_t start = std::clock();
//...
clock_t end = std::clock();

long milliseconds = (double)(end-start)/
                    (CLOCKS_PER_SEC/1000.0);
```

1.8. За шаблона DLList да се дефинира метод `bool find(const T& x)`, който проверява дали дали `x` е елемент на списъка или не. Да се напише подходящ тест и да се изследва времевата сложност на метода емпирично.

1.9. За шаблона `DLList` да се реалзиира изтриване на елемент по индекс.

1.10. Да се изпробват поне две различни стратегии за разширяване на динамичен масив (например, увеличаване на размера с 1 и с коефициент). Да напишат подходящи тестове и да се сравнят производителностите на двата подхода емпирично.

1.3 Итератори за линейни СД

Следните задачи да се решат като упражнение за работа с итератори. Задачите изискват реализация на клас динамичен масив и линеен двусвързан списък и `forward` итератори за тях. Всяка функциия да се тества с подходящи тестове върху двата вида контейнери. Има ли разлика в проиводителността за някои от тях в зависимост от избора на контейнер?

1.11. Да се разшири интераторът на динамичен масив така, че да поддържа оператора за стъпка назад `--`.

1.12. Да се дефинира функция `map`, която прилага едноаргументна функция $f : int \to int$ към всеки от елементите на произволен контейнер. Да се дефнира и шаблон на функцията за списък с произолен тип на елементите.

1.13. Да се напише функция `bool duplicates (...)`, която проверява дали в контейнер има дублиращи се елементи.

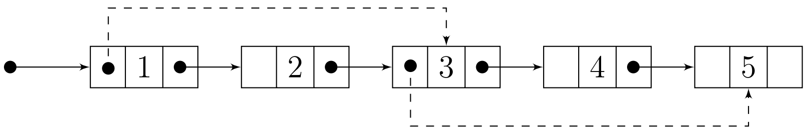
1.14. Да се напише фунцкия `bool issorted (...)`, която проверява дали елементите на даден контейнер са подредени в нарастващ или в намаляващ ред.

1.15. Да се напише фунцкия `bool palindrom (...)`, която проверява дали редицата от елементите на даден контейнер обрзува палиндром (т.е. дали се чете еднакво както отляво надясно така и отдяно наляво).

1.4 Skip List

Разглеждаме *опростена* реализация на структурата от данни Skip List ("Списък с прескачене, СП"). Възелът на линейния едносвързан списък разширяваме с още един указател към следващ елемент:

```
template <class T>
struct lnode
{
```



Фигура 1: Списък с прескачане

```
  T data;
  lnode<T> *next[2];
};
```

Както и при стандартния едносвързан списък, всеки от елментите на СП съдържа в указателя `next[0]` адреса на непосредствения си съсед. Някои от елементите могат да съдържат в указателя `next[1]` дреса на друг елемент, намиращ се по-напред в редицата от елементи (вж. Фигура 1). Например, нека имаме СП с n елемента в нарастващ ред. Ако списъкът е построен така, че всеки $\sqrt{n}$-ти елемент има указател към следващия $\sqrt{n}$-ти елемент, то търсенето на елемент ще бъде със сложност $O(\sqrt{n})$ на цената на линейно нарастване на необходимата памет. Идеята може да се продължи така, че всеки елемент да може да има и по-голям брой указатели към елементи все по-напред в СП, но за нашите цели ще се ограничим до описания прост СП.

Следващите задачи изискват реализация на клас `SkipList` с основните му канонични методи и метод за построяване на "бързите връзки". Реализирайте обиновен метод за вмъкване на елементи `insert`, който вмъква елементи грижейки се само за непосредствените връзки (`next[0]`), и метод `optimize`, който построява бързите връзки в списъка след като в него са вмъкнати определен брой елементи.

1.16. Да се реализира итератор на `SkipList` така, че да се възползва от "бързите връзки" в списъка. *Упътване: при обхождането извършвайте "прескачане" в случаите, в които има бърза връзка и в които няма да отидете твърде далеч напред в списъка. Класът на итератора трябва да се промени, за да позволява конструиране на итератор към конкретен елемент на списъка.*

1.17. Да се извърши времево измерване на проблема за търсене на елемент в подреден `SkipList`, както е обяснено в Секция 1.2, и да се изобрази чрез графика. Да се извършат емпирични сравнения на производителността на търсенето със и без оптимизацията.

2 Двоични дървета

2.1 Прости обхождания

Възел на двоично дърво представяме със следния шаблон на структура:

```
template <class T>
struct btreenode
{
  T data;
  btreenode<T> *left, *right;
};
```

Освен ако не е указано друго, задачите по-долу да се решат като се реализират методи на клас `BTree` със следния скелет:

```
template <class T>
class BTree
{
  //...
  private:
  btreenode<T> *root;
};
```

Преди да пристъпите към задачите, реализирайте подходящи контруктори, деструктор и оператор за присвояване на класа.

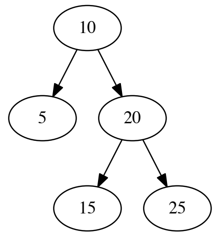
2.1. Да се деифнира метод `count` на клас `BTree`, който намира броя на елементите на дървото.

2.2. Да се деифнира метод `countEvens` на клас `BTree`, който намира броя на елементите на дърво от числа, които са четни.

2.3. Да се дефинира метод `int BTree<T>::searchCount (bool (*pred)(const T&))` към клас `BTree`, който намира броя на елементите на дървото, които удовлетворяват предиката `pred`.

   Да се приложи `searchCount` за решаване на горните две задачи.

2.4. Да се дефинира метод `bool BTree<T>::height ()`, намиращ височината на дърво. *Височина на дърво наричаме дължината (в брой върхове) на най-дългия път от корена до кое да е листо на дървото. Пример. Височината на дървото на Фигура 2 е **3**.*



Фигура 2: Двоично наредено дърво

2.5. Да се деифнира метод `countLeaves` на клас `BTree`, който намира броя на листата в дървото.

2.6. Да се деифнира метод `maxLeaf` на клас `BTree`, който намира най-голямото по стойност листо на непразно дърво. Да се приеме, че за типа `T` на шаблона `BTree` е дефиниран операторът `<`.

2.7. Нека е дадено дървото `t` и низът `s`, съставен само от символите 'L' и 'R' ($s \in \{L, R\}^*$). Нека дефинираме "съответен елемент" на низа `s` в дървото `t` по следния начин:

- Ако дървото `t` е празно, низът `s` няма съответен елемент
- Ако низът `s` е празен, а дървото `t` - не, то коренът на дървото `t` е съответният елемент на низа `s`
- Ако първият символ на низа `s` е 'L' и дървото `t` не е празно, то съответният елемент на низа `s` в дървото `t` е съответният елемент на низа $s + 1$ в **лявото** поддърво на `t`
- Ако първият символ на низа `s` е 'R' и дървото `t` не е празно, то съответният елемент на низа `s` в дървото `t` е съответният елемент на низа $s + 1$ в **дясното** поддърво на `t`

Пример. За дървото от Фигура 2, съответният елемент на празния низ е 10, на низа "RL" е 15, а "RLR" няма съответен елемент.

Да се дефинира метод `T& BTree<T>::getElement (const char *s)`, който намира съответния елемент на низа `s`. Какво връща методът в случаите на липса на съответен елемент е без значение.

2.2 Отпечатване и сериализация

2.8. Да се дефинира метод `void BTree<T>::prettyPrint ()`, отпечатващ дървото на конзолата по следния начин: (1) всеки наследник е вдясно от родителя си, (2) елементите на еднакво ниво в дървото се отпечатват на еднаква колона от екрана, (3) десните наследници са на предишен ред от родителя си и (4) левите наследни са следващ ред спрямо родителя си.

Например, дървото от Фигура 2 би изглеждало по следния начин (включени са номерата на редовете на конзолата):

```
1:          25
2:      20
3:          15
4: 10
5:      5
```

2.9. Да се дефинират методи за сериализация и де-сериализация на двоично дърво, като се използва "`Scheme` формат".

Представяне на двоично дърво в "Scheme формат" наричаме еднозначно текстово представяне на структурата от данни, образувано по следните правила:

- Празното дърво се прдставя с низа "()"
- Нека е дадено дървото t с корен x, ляво поддърво t_l и дясно поддърво t_r. Ако s_l е представянето в "Scheme формат" формат на t_l, а s_r е представянето в "Scheme формат" на t_r, то низът "(x s_l s_r)" е представянето на дървото t, където "x", "s_l" и "s_r" са съответните низове.

Например, дървото от Фигура 2 се представя по следния начин:

```
(10 (20 (25 () ()) (15 () ())) (5 () ()))
```

2.3 Извличане на нескаларни свойства

2.10. Да се реализира метод `std::vector<T> BTree<T>::listLeaves ()` намиращ списък със стойностите на листата на дървото.

2.11. Да се дефинира метод std::string BTree<T>::findTrace (const T& x). Ако x е елемент на дървото, функцията да връща следата на x (според дефиницията на “следа”, обсъдена на лекции). Ако x не е елемент на дървото, функцията да връща низ “_”.

Пример: За дървото от Фигура 2, следата на елемента със стойност 25 е “RR”.

2.4 Итериране на елементите на двоично дърво

2.12. Да се дефинира оператор T& BTree<T>::operator[](int i), който намира i-тият пореден елемент на дървото при обхождане корен-ляво-дясно.

Пример: За дървото от Фигура 2, елементът с пореден номер 0 е 10, с номер 1 е 5, с номер 2 е 20 и т.н.

2.5 Представяне на аритметичен израз чрез дърво

2.13. Нека е даден израз, построен по правилата на следната граматика:

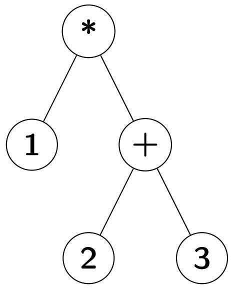
```
<expression> ::= <digit> | (<expression><operator><expresson>)
<digit> ::= 1..9
<operator> ::= + | - | * | /
```

Да се реалзира метод на клас BTree<char>, void parseExpression (std::string s), който по правилно построен израз, записан в низа s, създава двоично дърво от символи, представящо израза по следното правило:

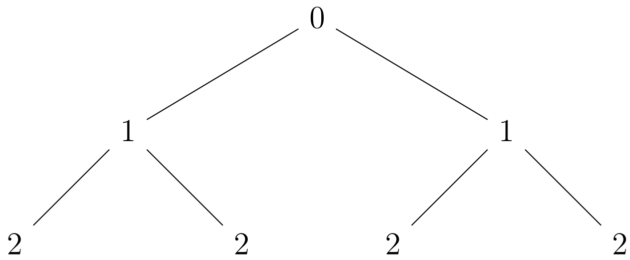
- Ако изразът е от типа “x”, където x е цифра, то съответното му дърво е листо със стойност символа x.
- Ако изразът е от типа “(<израз 1><op><израз 2>)”, то съответното му дърво има като стойност на корена символа на съответния оператор, ляво поддърво, съответно на <израз 1> и дясно поддърво, съответно на <израз 2>.

Дървото на Фигура 7 съответства на изарза (1*(2+3)).

2.14. Да се реализира метод double BTree<char>::calculateExpresisonTree (), който намира стойността на израз, построен от решението на предишната задача.



Фигура 3: Дърво на израза `(1*(2+3))`.



Фигура 4: Идеално балансирано двоично дърво с височина 3

2.6 Построяване и модификации на дърво

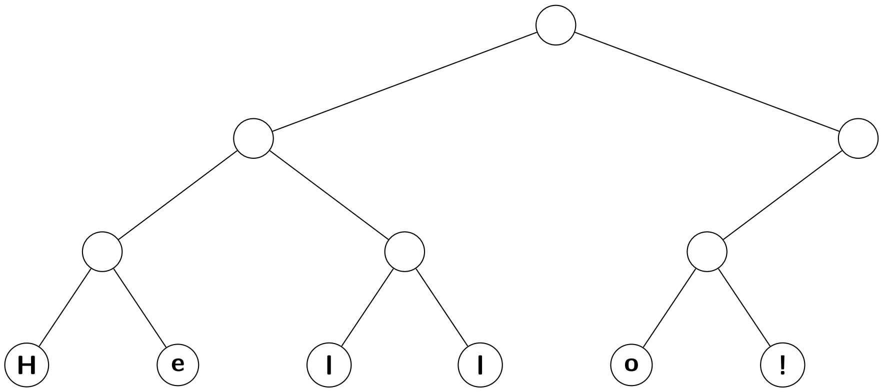
2.15. По дадено число h да се построи идеално балансирано двоично дърво с височина h. Стойността на всеки от елементите на дървото да е равна на нивото, в което се намира елемента (вж. Фигура 4).

2.16. Нека е даден символен низ s с дължина n. Нека h е такова, че $2^h \geq n > 2^{h-1}$ (минималната височина на двоично дърво, което има поне n листа в последното си ниво). Да се дифинира функция, която по даден низ s построява двоично дърво от символи с височина h, такова, че низът s е разположен в листата на дървото, четени от ляво надясно. Възлите на дървото, които не са листа, да съдържат символа интервал. Вж. Фигура 5

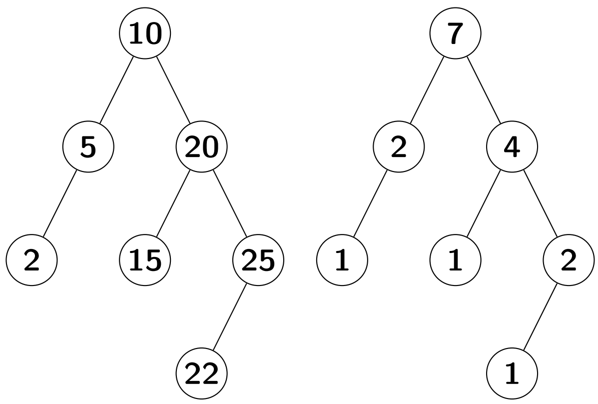
2.17. Стойността на всеки възел `V` в дадено двоично дърво от числа да се замени с броя на всички елементи на поддървото, на което `V` е корен. Вж. Фигура 6. *При операцията всеки от възлите да бъде посетен най-много веднъж!*

2.18. Дадено е дърво с низове по върховете. Дървото да се балансира по следния начин:

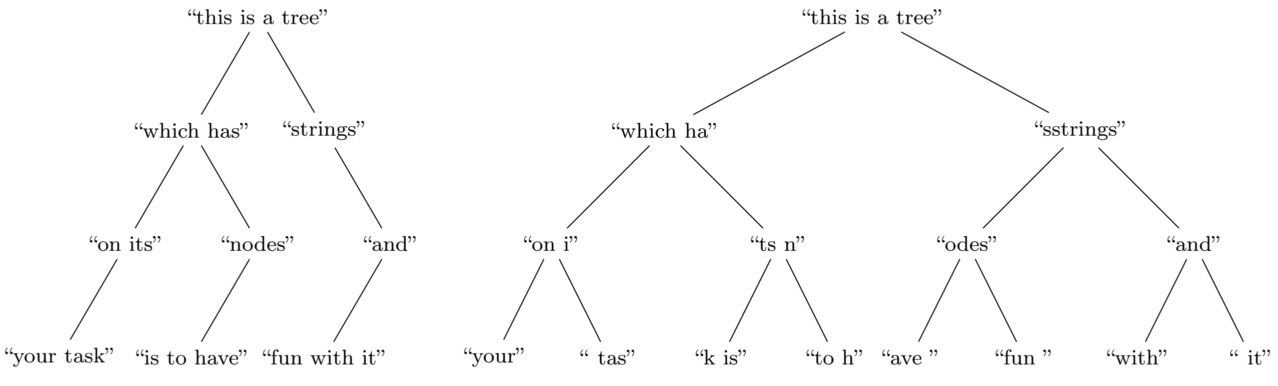
a) Резултатното дърво има същия брой нива като изходното.



Фигура 5: Дърво, в чиито листа е разположен низът “Hello!”



Фигура 6: Примерно дърво и същото дърво, стойностите на чиито възли са заместени с размера на съответното им поддърво



Фигура 7: Примерно дърво от низове преди и след балансирането

б) Всяко k-то ниво на резултатното дърво да съдържа точно 2^k елемента (считаме, че коренът е на ниво 0).

в) Нека s_k е низът, получен при конкатенацията на всички низове на ниво k на изходното дърво, обхождани от ляво надясно. Нека дължината на низа s_k е n_k символа. i-тият пореден елемент на нивото k в резултатното дърво да съдържа i-тата поредна последователност от $\lceil n_k/2^k \rceil$ на брой символи на s_k, освен най-десния, който съдържа последните "останали" символи от s_k. Т.е. s_k да се "раздели" поравно между елементите в резултатното дърво.

На Фигура 7 са илюстрирани примерно изходно дърво и резултатът от балансирането му по горното правило. Всички елементи на ниво 1, освен последния, съдържат по $8 = \lceil 16/2 \rceil$ символа. Всички елементи на ниво 2, освен последния, съдържат по $4 = \lceil 14/4 \rceil$ символа и т.н.

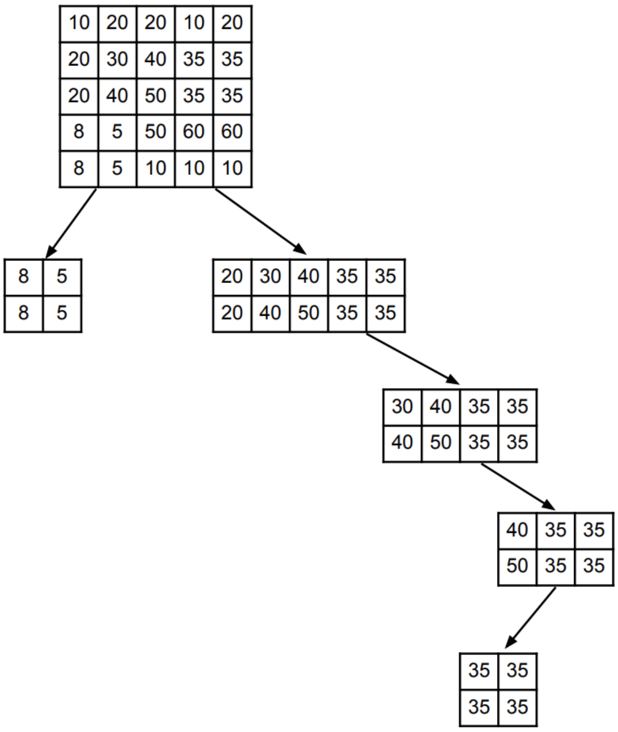
Упътване: предварително намерете вектора $(s_0, s_1, ..., s_{h-1})$ *и го използвайте за балансирането.*

2.19. Нека е дадена матрица от цели числа $A_{M \times N}$ с елементи $(a_{i,j})$. "Лява" подматрица на A наричаме такава подматрица $A'_{M' \times N'}$ на A, всеки елемент на която е по-малък от $a_{0,0}$. "Дясна" подматрица на A наричаме такава подматрица $A'_{M' \times N'}$ на A, всеки елемент на която е по-голям от $a_{0,0}$. По дадена матрица A да се построи двоично дърво T със следните свойства:

- Коренът на T съдържа матрицата A.
- Нека v е произволен възел от дървото T, съдържащ матрица X. Ако X има поне една лява подматрица с размер поне 2×2, то левият наследник на X съдържа най-голямата (по брой елементи) лява подматрица на X. Ако има повече от една ляввa подматрциа с максимален брой елементи, то левият наследник на v е произволна една от тях. Ако X няма лява подматрица с размер поне 2×2, то v няма ляв наследник.
- Аналлогичното свойство за десния наследник на v и най-голямата дясна подматрица (подматрици) на X.

На Фигура 8 е изобразено едно такова дърво.

а) Да се избере подходящо представяне на матрици и на двоично дърво с матрици по върховете.



Фигура 8: Наредено дърво от матрици

б) Да се дефинира функция за построяване на дърво по горното правило по дадена матрица за корена му.

в) Да се отпечата дървото чрез Graphviz. Повече информация за отпечатване на матрици като елемент на дървото може да се намери в документацията на Graphviz.

Литература