Задачи за задължителна самоподготовка по Увод в програмирането, Обектно-ориентирано програмиране и Структури от данни

Калин Георгиев
kalin@fmi.uni-sofia.bg

12 януари 2020 г.

Съдържание

Някои от задачите по-долу са решени в сборника [1]*Магдалина Тодорова, Петър Армянов, Дафина Петкова, Калин Георгиев, "Сборник от задачи по програмиране на C++. Първа част. Увод в програмирането"*. За задачите от сборника е посочена номерацията им от сборника.

1 Увод, основи и примери

1.1 Основни примери

1.1. Превърнете рожденната си дата шестнадесетична, в осмична и в двоична бройни системи.

1.2. Как бихте кодирали вашето име само с числа? Измислете собствено представяне на символни константи чрез редици от числа и запишете името си в това представяне.

Разгледайте стандартната ASCii таблица (`http://www.asciitable.com/`) и запишете името си чрез серия от ASCii кодове.

1.2 Променливи, вход и изход, логически и аритметични операции, условен оператор

1.3. Задача 1.6.[1] Да се напише програма, която по зададени навършени години намира приблизително броя на дните, часовете, минутите и секундите, които е живял човек до навършване на зададените години.

1.4. Задача 1.7.[1] Да се напише програма, която намира лицето на триъгълник по дадени: а) дължини на страна и височина към нея; б) три страни.

1.5. Задача 2.7.[1] Да се напише програма, която въвежда координатите на точка от равнина и извежда на кой квадрант принадлежи тя. Да се разгледат случаите, когато точката принадлежи на някоя от координатните оси или съвпада с центъра на координатната система.

1.6. Задача 1.14.[1] Да се запише булев израз, който да има стойност истина, ако посоченото условие е вярно и стойност - лъжа, в противен случай:

а) цялото число p се дели на 4 или на 7;

б) уравнението $ax^2 + bx + c = 0 (a \neq 0)$ няма реални корени;

в) точка с координати (a, b) лежи във вътрешността на кръг с радиус 5 и център (0, 1); г) точка с координати (a, b) лежи извън кръга с център (c, d) и радиус f;

г) точка принадлежи на частта от кръга с център (0, 0) и радиус 5 в трети квадрант;

д) точка принадлежи на венеца с център (0, 0) и радиуси 5 и 10;

е) x принадлежи на отсечката [0, 1];

ж) x е равно на max {a, b, c};

з) x е различно от max { a, b, c};

и) поне една от булевите променливи x и y има стойност true;

к) и двете булеви променливи x и y имат стойност true;

л) нито едно от числата a, b и c не е положително;

м) цифрата 7 влиза в записа на положителното трицифрено число p;

н) цифрите на трицифреното число m са различни;

о) поне две от цифрите на трицифреното число m са равни помежду си;

п) цифрите на трицифреното естествено число x образуват строго растяща или строго намаляваща редица;

р) десетичните записи на трицифрените естествени числа x и y са симетрични;

с) естественото число x, за което се знае, че е по-малко от 23, е просто.

1.7. Задача 2.12.[1] Да се напише програма, която проверява дали дадена година е високосна.

1.3 Цикли

1.8. Задача 1.20.[1] Да се напише програма, която по въведени от клавиатурата цели числа x и k ($k \geq 1$) намира и извежда на екрана k-тата цифра на x. Броенето да е отдясно наляво.

1.9. Задача 2.40.[1] Да се напише програма, която (чрез цикъл for) намира сумата на всяко трето цяло число, започвайки от 2 и ненадминавайки n (т.е. сумата 2 + 5 + 8 + 11 + ...).

1.10. Задача 2.44.[1] Дадено е естествено число n ($n \geq 1$). Да се напише програма, която намира броя на тези елементи от серията числа $i^3+13\times i\times n+n^3$, $i = 1, 2, ..., n$, които са кратни на 5 или на 9.

1.11. За въведени от клавиатурата естествени числа n и k, да се провери и изпише на екрана дали n е точна степен на числото k.

Упътване: Разделете променливата n на променливата k "колкото пъти е възможно" и проверете дали n достига единица или някое друго число след края на процеса. Използвайте добре подбрано условие за for цикъл, оператора % за намиране на остатък при целочислено деление, и оператора за целочислено деление /.

1.4 Машини с неограничени регистри

Дефиницията на Машина с неогрничени регистри по-долу е взаимствана от учебника [2]*А. Дичев, И. Сосков, "Теория на програмите", Издателство на СУ, София, 1998.*

"Машина с неограничени регистри" (или МНР) наричаме абстрактна машина, разполагаща с неограничена памет. Паметта на машината се представя с безкрайна редица от естествени числа $m[0], m[1], ...$, където $m[i] \in \mathcal{N}$. Елементите $m[i]$ на редицата наричаме "клетки" на паметта на машината, а числото i наричаме "адрес" на клетката $m[i]$.

МНР разполага с набор от инструцкии за работа с паметта. Всяка инструкция получава един или повече параметри (операнди) и може да предизвика промяна в стойността на някоя от клетките на паметта. Инструкциите на МНР за работа с паметта са:

1) `ZERO n`: Записва стойността 0 в клетката с адрес n
2) `INC n`: Увеличава с единица стойността, записана в клетката с адрес n
3) `MOVE x y`: Присвоява на клетката с адрес y стойността на клетката с адрес x

"Програма" за МНР наричаме всяка последователност от инструкции на МНР и съответните им операнди. Всяка инструкция от програмата индексираме с поредния ѝ номер. Изпълнението на програмата започва от първата инструкция и преминава през всички инструкции последователно, освен в някои случаи, описани по-долу. Изпълнението на програмата се прекратвя след изпълнението на последната ѝ инструкция. Например, след изпълнението на следната програма:

```
0: ZERO 0
1: ZERO 1
2: ZERO 2
3: INC 1
4: INC 2
5: INC 2
```

Първите три клетки на машината ще имат стойност 0, 1, 2, независимо от началните им стойности.

Освен инструкциите за работа с паметта, МНР притежават и една инструкция за промяна на последователноста на изпълнение на програмата:

4) `JUMP x`: Изпълнението на програмата "прескача" и продължава от инструкцията с пореден номер x. Ако програмата има по-малко от $x+1$ инструкции, изпълнението ѝ се прекратява

5) `JUMP x y z`: Ако съдържанията на клетките x и y съвпадат, изпълнението на програмата "прескача" и продължава от инструкцията с пореден номер z. В противен случай, програмата продължава със следващата инструкция. Ако програмата има по-малко от $z+1$ инструкции, изпълнението ѝ се прекратява

Например, нека изпълнето на следната програма започва при стойности на клиетките на паметта 10,0,0,...:

```
0: JUMP 0 1 5
1: INC 1
2: INC 2
3: INC 2
4: JUMP 0
```

След приключване на програмата, първите три клетки на машината ще имат стойности 10, 10, 20.

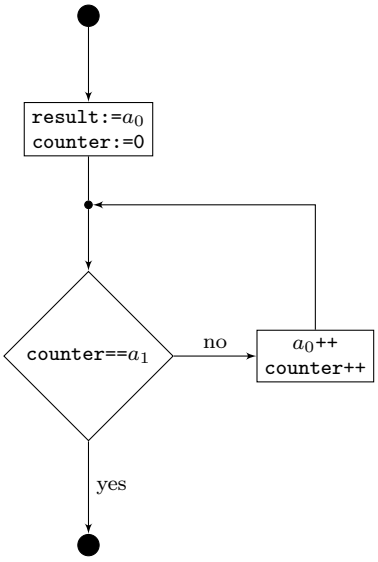
1.12. Нека паметта на МНР е инициалирана с редицата $m, n, 0, 0, \ldots$. Да се напише програма на МНР, след изпълнението на която клетката с адрес 2 съдържа числото $m + n$.

1.13. Нека паметта на МНР е инициалирана с редицата $m, n, 0, 0, \ldots$. Да се напише програма на МНР, след изпълнението на която клетката с адрес 2 съдържа числото $m \times n$.

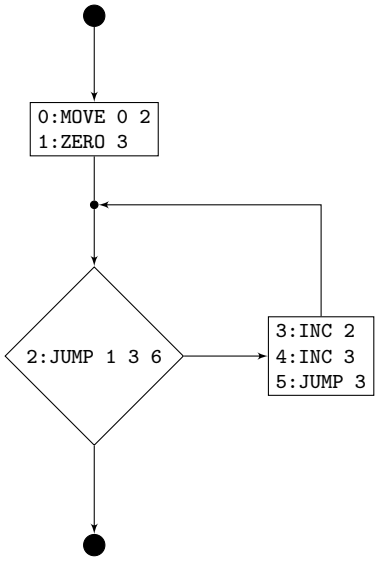
1.14. Нека паметта на МНР е инициалирана с редицата $m, n, 0, 0, \ldots$. Да се напише програма на МНР, след изпълнението на която клетката с адрес 2 съдържа числото 1 тогава и само тогава, когато $m > n$ и числото 0 във всички останали случаи.

Упътване: На Фигура 1 (a) е показана блок схема на програма, изпозлваща само операторите `=`, `==`, `++` и `if`, която намира в променливата `result` сумата на променливите a_0 и a_1. a_0 и a_1 считаме за дадени. Променливата `count` се инициилиаира с 0, а `result` - с a_0. В цикъл се добавя по една единица към `count` и `result` дотогава, докато `count` достигне стойността на a_1. По този начин, към `result` се добавят a_1 на брой единици, т.е. стойността ѝ се увеличава с a_1 спрямо налчалната ѝ стойност a_0.

На Фигура 1 (b) е показана същата програма, като операторите от първата са заменени със сътответните им инструцкии на МНР. Резултатът от



(a)Програма за сумиране на числата a_0 и a_1 с използване само на операторите `=`, `==`, `++` и `if`.



(b)Програма за сумиране на клетките $m[0]$ и $m[1]$ с инструкции на МНР.

Фигура 1: Блок схеми на програма за сумиране на числа

програмата се получава в клетката $m[2]$, а за брояч се ползва клетката $m[3]$. На блок схемата са дадени поредните номера на инструкциите в окончателната програмата на МНР:

```
0:MOVE 0 2
1:ZERO 3
2:JUMP 1 3 6
3:INC 2
4:INC 3
5:JUMP 3
```

2 Типове и функции

2.1 Прости примери за функции

2.1. Задача 4.12.[1] Да се напише булева функция, която проверява дали дата, зададена в следния формат: dd.mm.yyyy е коректна дата от грегорианския календар.

2.2. Задача 4.25.[1] Да се дефинира процедура, която получава целочислен параметър n и база на бройна система $k \leq 16$. Процедурата да отпечатва на екрана представянето на числото n в системата с база k.

2.3. Задача 2.57.[1] Да се напише булева функция, която проверява дали сумата от цифрите на дадено като параметър положително цяло число е кратна на 3.

2.4. Задача 2.81.[1] Едно положително цяло число е съвършено, ако е равно на сумата от своите делители (без самото число). Например, 6 е съвършено, защото 6 = 1+2+3; числото 1 не е съвършено. Да се напише процедура, която намира и отпечатва на екрана всички съвършени числа, ненадминаващи дадено положително цяло число в параметър n.

2.2 Елементарна растерна графика

Следните задачи да се решат с показаните на лекции графични примитиви, базирани на платформата за компютърни игри SDL2. За целта е необходимо да инсталирате SDL2 на компютъра си и да настроите средата си за програмиране така, че да свърже SLD2 с вашия проект. Информация за това можете да намерите на сайта на платформата. Задачите можете да решите с помощта на всяка друга билиотека, поддържаща примитивите за рисуване на точки и отсечки.

Примерната програма от лекции използва файла `mygraphics.h`, който можете да намерите в хранилището на курса:

```
#include "mygraphics.h"
```

`Mygraphics` "обвива" библиотеката SLD2 и дефинира следните лесни за използване макроси:

- `setColor (r,g,b)`: Дефинира цвят на рисуване с компоненти $r, g, b \in [0, 255]$. Например, белият цвят се задава с $(255, 255, 255)$, червеният с $(255, 0, 0)$ и т.н.
- `drawPixel(x,y)`: Поставя една точка на екранни коррдинати (x, y).

- `drawLine (x1,y1,x2,y2)`: Рисува отсечка, свързваща точките с екранни координати (x_1, y_1) и (x_2, y_2).
- `updateGraphics()`: Извиква се веднъж в края на програмата, за да се изобрази нарисуваното с горните примитиви.

2.5. Да се нарисуват програмно следните фигури чрез дефиниране на подходящи функции:

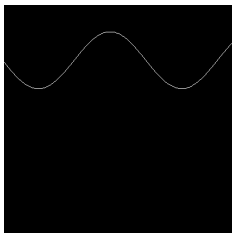
- Равностранен триъгълник по дадена дължина на страната
- Равностранен шестоъгълник по дадена дължина на страната
- Графиката на функциите $y = sin(x)$, $y = x^2$, и $y = x^3 + 5x^2$ поотделно и заедно
- Равностранен многоъгълник по дадени координати на пресечната точка на симетралите му (център), брой страни n и разстояние от центъра до върховете r. При какви стойности на n фигурата наподобява окръжност?
- Логаритмична крива
- Елипса с център дадени (x, y) и радиуси дадени r_1 и r_2

2.6. По дадени екранни координати (x, y) на горния ляв ъгъл на квадрат, дължина на страната a на квадрата и число n:

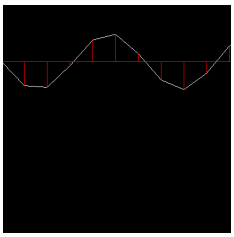
- Да се нарисува квадратна матрица от $n\times$n квадрата със страна a/n, изпълваща дадения квдрат.
- Квадратите от горното условие да се заменят с тригъгълниците, образувани от пресичането на диагоналите им.

Упътване към задачата за чертане на графика на функция. Да разглевдаме функцията $y = sin(x)$: Рисуват се отсечки между последователни точки от графиката на функцията, като всяка следваща точка се получава като увеличаваме стойността на аргумента x с числото `stepX`. Добре е `stepX` да е достатъчно малко, за да не се "накъса" графиката на функцията и да е достатъчно голямо, за да е видимо изменението ѝ. За този пример сме избрали `stepX = 0.05`.

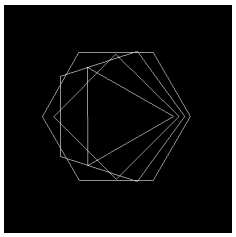
Тъй като $sin(x) \in [-1, 1]$, ако директно визуализираме точките на получените по този начин координати $(x, sin(x))$, те ще са "сгъстени" около правата $y = 0$ и резултатът няма да е добър. Поради това, координатите на получените точки от кривата се умножават по `scaleX` и `scaleY` съответно, $(x * scaleX, sin(x) * scaleY)$, за да се "разпъне" графиката по двете оси. (Вж. Фигура 2)



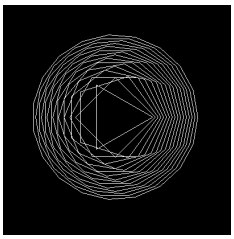
(a)Графика на $y = sin(x)$, нарисувана с 300 отсечки



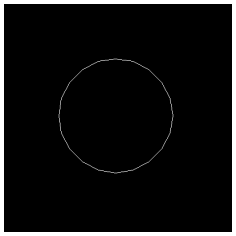
(b)Графика на $y = sin(x)$ с 10 отсечки. Нарисувани са също отсечки между точките от графиката на функцията и абсцисата



(c)Четири многоъгълника, нарисувани един върху друг с нарастващи радиус и брой върхове



(d)Шестнадесет многоъгълника, нарисувани един върху друг с нарастващи радиус и брой върхове



(e)Многоъгълник с 20 върха, приближаващ окръжност

Фигура 2: Примерни резултати от решенията на някои задачи

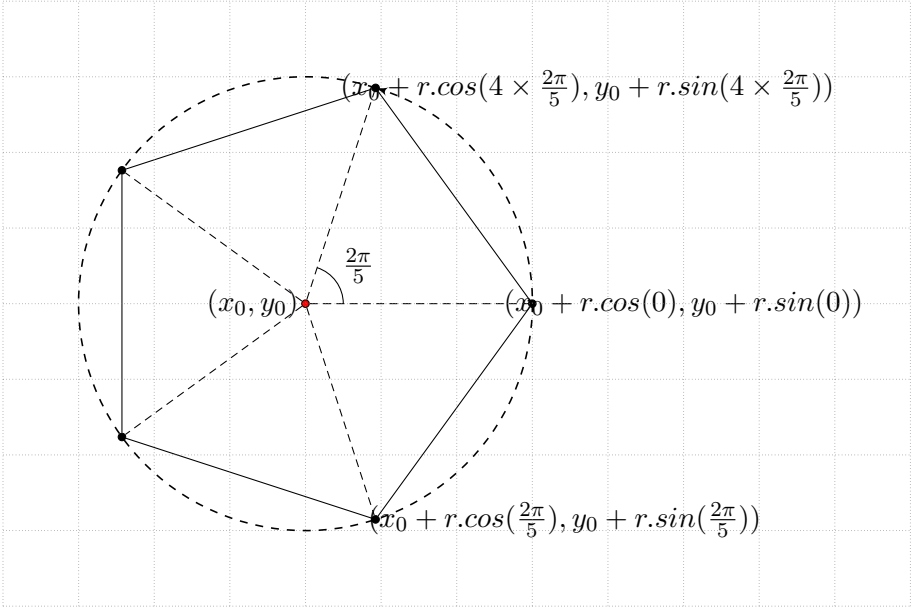
```
const double //scaleX: коефициент на скалиране по X
             scaleX = 40.0,
             //y0: ординатa на началната точка
             y0 = 100,
             //scaleY: коефициент на скалиране по Y
             scaleY = 50.0,
             //stepX: стъпка за нарастване на аргумента
             stepX = 0.05;
             //nsegments: брой сегменти от кривата
const int nsegments = 300;

for (int i = 0; i < nsegments; i++)
{
      double x     = scaleX*i*stepX,
            xnext = scaleX*(i+1)*stepX,
            y     = y0+scaleY*sin(stepX*i),
            ynext = y0+scaleY*sin(stepX*(i+1));
      drawLine (x,y,xnext,ynext);
}
```

Упътване към задачата за чертане на многоъгълник: Върховете на мнгоъгълника получаваме, като започнем с точката с координати $(x + radius, y)$ и получаваме всяка следваща точка като "завъртим" предишната около (x, y) с $2\pi/n$ радиана, където n е броят на върховете на многоъгълника. Получените по този начин точки се съединяват с отсечки. (Вж. Фигура 2 и Фигура 3.)

```
/*
Функция polygon (n,x,y,radius): Рисуване на многоъгълник
параметър n: брой върхове на многоъгълника
параметри x,y: координати на центъра на многоъгълника
параметър radius: разстояние от центъра до върховете
*/
void polygon (int n, double x, double y, double radius)
{
  for (int side = 0; side < n; side++)
  {
    drawLine (radius*cos(side*2.0*M_PI/n)+x,
              radius*sin(side*2.0*M_PI/n)+y,
              radius*cos((side+1)*2.0*M_PI/n)+x,
              radius*sin((side+1)*2.0*M_PI/n)+y);
  }
}
```

2.7. Да се нарисува програмно котката Pusheen на Фигура 4 (вж. [3]).

2.8. (*) *Следната задача илюстрира метода на трапеците (Trapezoidal rule) за приближено изчисление на определени интеграли:*



Фигура 3: Рисуване на петоъгълник чрез намиране на 5 равноотдалечени точки по окръжността с радиус r и център (x_0, y_0)

Фигура 4: Pusheen the cat. Фигурата е от [3]

Да се нарисуват програмно координатни оси на евклидова координнатна система с даден център в екранните координати (x,y). Да приемем, че в програмата е дефинирана функция `double f (double x)`, за която знаем, че е дефинирана за всяка стойност на x.

- Да се изобрази графиката на фукцията спрямо нарисувана координатна система
- Да се приближи чрез трапеци с дадена дължина на основата δ фигурата, заключена между видимата графика на фигурата и абсцисата
- Да се визуализират така получените трапеци
- Да се изчисли сумата от лицата на така получените трапеци
- Да се експериментра с различни дефиниции на функцията f

3 Цикли, масиви и низове

3.1 Цикли II

Където не е посочено изрично, под "редица от числа $a_0, a_1, ..., a_{n-1}$" по-долу се разбира последователност от n числа, въведени от стандартния вход. Задачите да се решат *без* използването на масиви.

3.1. Задача 3.1. [1] Да се напише програма, която въвежда редица от n цели числа ($1 \leq n \leq 50$) и намира и извежда минималното от тях.

3.2. Задача 3.2. [1] Да се напише програма, която въвежда редицата от n ($1 \leq n \leq 50$) цели числа $a_0, a_1, ..., a_{n-1}$ и намира и извежда сумата на тези елементи на редицата, които се явяват удвоени нечетни числа.

3.3. Задача 3.3. [1] Да се напише програма, която намира и извежда сумата от положителните и произведението на отрицателните елементи на редицата от реални числа $a_0, a_1, ..., a_{n-1}$ ($1 \leq n \leq 20$).

3.4. Задача 3.7. [1] Да се напише програма, която изяснява има ли в редицата от цели числа $a_0, a_1, ..., a_{n-1}$ ($1 \leq n \leq 100$) поне два последователни елемента с равни стойности.

3.5. Задача 3.8. [1] Да се напише програма, която проверява дали редицата от реални числа $a_0, a_1, ..., a_{n-1}$ ($1 \leq n \leq 100$) е монотонно растяща.

3.6. Задача 3.15. [1] Да се напише програма, която въвежда реланите вектори $a_0, a_1, ..., a_{n-1}$ и $b_0, b_1, ..., b_{n-1}$ ($1 \leq n \leq 100$), намира скаларното им произведение и го извежда на екрана.

3.7. Задача 3.10. [1] Да се напише програма, която за дадена числова редица $a_0, a_1, ..., a_{n-1}$ ($1 \leq n \leq 100$) намира дължината на най-дългата ú ненамаляваща подредица $a_i, a_{i+1}, ..., a_{i+k}$ ($a_i \leq a_{i+1} \leq ... \leq a_{i+k}$).

3.2 Цикли и низове

Където не е посочено изрично, под "редица от символи $s_0, s_1, ..., s_{n-1}$ ($1 \leq n \leq 100$) $a_0, a_1, ..., a_{n-1}$" по-долу се разбира символен низ с дължина n, въведен от клавиатурата в масив от тип `char[255]`.

3.8. Задача 3.11. [1] Дадена е редицата от символи $s_0, s_1, ..., s_{n-1}$ ($1 \leq n \leq 100$). Да се напише програма, която извежда отначало всички символи, които са цифри, след това всички символи, които са малки латински букви и накрая всички останали символи от редицата, запазвайки реда им в редицата.

3.9. Задача 3.13. [1] Задача 3.13. Да се напише програма, която определя дали редицата от символи $s_0, s_1, ..., s_{n-1}$ $(1 \leq n \leq 100)$ е симетрична, т.е. четена отляво надясно и отдясно наляво е една и съща.

3.10. Да се напише функция, която по два низа намира дължината на най-дългия им общ префикс. *Префикс на низ наричаме подниз със същото начало като дадения. Пример: празният низ и низовете "a", "ab", и "abc" са всички възможни префикси на низа "abc". Дължината на най-дългия общ префикс на низовете "abcde" и "abcuwz" е 3.*

3.11. Да се напише функция, която в даден низ замества всички малки латински букви със съответните им големи латински букви.

3.12. Да се напише функция `reverse(s)`, която превръща даден низ в огледалния му образ. *Например, низът "abc" ще се пребразува до "cba".*

3.13. Да се напише функция, която по даден низ s, всички букви в който са латински, извършва следната манипулация над него: Ако s съдържа повече малки, отколкото големи букви, замества всички големи букви в s с малки. В останалите случаи, всички малки букви се заместват с големи.

3.14. Задача 3.26. "Хистограма на символите"[1] Символен низ е съставен единствено от малки латински букви. Да се напише програма, която намира и извежда на екрана броя на срещанията на всяка от буквите на низа.

3.15. Да се напише булева функция, която по дадени низове s_1 и s_2 проверява дали s_2 е подниз на s_1 (*Например, низът "uv" е подниз на низовете "abuvc", "uvz", "zuv" и "uv", но не е подниз на низа "uwv".*). Функцията да *не* използва вложени цикли.

3.16. Задача 3.28. "Търсене на функция"[1] Дадени са два символни низа с еднаква дължина s_1 и s_2, съставени от малки латински букви. Да се напише програма, която проверява дали съществува функция $f : char \to char$, изобразяваща s_1 в s_2, така че $f(s_1[i]) = f(s_2[i])$ и $i = 1..$дължината на s_1 и s_2. *Упътване: За да е възможна такава функция, не трябва в s_1 да има символ, на който съответстват два или повече различни символи в s_2. Например, низът "aba" може да бъде изобразен в низа "zwz", но не и в низа "zwu".*

3.3 Матрици и вложени цикли

3.17. Задача 3.18. [1] Дадени са числовите редици $a_0, a_1, ..., a_{n-1}$ и $b_0, b_1, ..., b_{n-1}$ ($1 \leq n \leq 50$). Да се напише програма, която въвежда от клавиатурата двете редици и намира броя на равенствата от вида $a_i = b_j$ ($i = 0, ..., n-1, j = 0, ..., n-1$).

3.18. Задача 3.21. [1] Две числови редици си приличат, ако съвпадат множествата от числата, които ги съставят. Да се напише програма, която въвежда числовите редици $a_0, a_1, ..., a_{n-1}$ и $b_0, b_1, ..., b_{n-1}$ ($1 \leq n \leq 50$) и установява дали си приличат.

3.19. Задача 3.29. [1] Дадена е квадратна целочислена матрица A от n-ти ред ($1 \leq n \leq 50$). Да се напише програма, която намира сумата от нечетните числа под главния диагонал на A (без него).

3.20. Задача 3.45. [1] Матрицата A има седлова точка в $a_{i,j}$, ако $a_{i,j}$ е минимален елемент в i-тия ред и максимален елемент в j-тия стълб на A. Да се напише програма, която извежда всички седлови точки на дадена матрица A с размерност $n \times m (1 \leq n \leq 20, 1 \leq m \leq 30)$.

3.21. Задача 3.113. (периодичност на масив). [1] Да се напише програма, която проверява дали в едномерен масив от цели числа съществува период. Например, ако масивът е с елементи 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, периодът е 3. Ако период съществува, да се изведе.

3.22. Да се напише програма, която въвежда от клавиатурата матриците от цели числа $A_{N \times M}$ и $B_{M \times N}$ и извежда на екрана резултатът от умножението на двете матрици.

3.4 Елементарно сортиране на масиви

3.23. Да се реализира функция, сортираща масив по метода на мехурчето. При този метод масивът $a[0], .., a[n-1]$ се обхожда от началото към края, като на всяка стъпка се сравняват двойката съседни елементи $a[i]$ и $a[i+1]$. Ако $a[i+1] < a[i]$, местата им се разменят. Този процес се извършва n пъти.

3.24. Да се напише функция `unsortedness`, която оценява доколко един масив е несортиран като преброява колко от елементите му "не са на местата си". Т.е. функцията намира броя на тези елементи a_i, които не са i-ти по големина в масива. Например, за масива $\{0, 2, 1\}$ това число е 2.

3.25. Да се напише функция `swappable`, която за масива $a[0], .., a[n-1]$ проверява дали има такова число $i (0 < i < n-1)$, че масива $a_i, .., a_n, a_0, .., a_{i-1}$ е стортиран в нарастващ ред. Т.е. може ли масивът да се раздели на две

части (незадължително с равна дължина) така, че ако частите се разменят, да се получи нареден масив. Пример за такъв масив е $\{3, 4, 5, 1, 2\}$.

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);
```

Функцията `rand()` от `<cstdlib>` генерира редица от псевдо-случайни числа. Всяко последователно изпълнение на функцията генерира следващото число от редицата. За да се осигури, че при всяко изпълнение на програмата ще се генерира различна редица от псевдо-случайни числа, е необходимо да се изпълни функцията `srand()` с параметър, който е различен за всяко изпълнение на програмата. Една лесна възможност е да се позлва резултата на функцията `time(0)`, която дава текущото време на системния часовник в стандарт `epoch time`. Достатъчно е `srand()` да се изпълни веднъж за цялото изпълнение на програмата.

Чрез следния примерен код може да се генерира редица от 10 (практически) случайни числа, които са различни при всяко изпълнение на програмата.

```
srand (time(0));
for (int i = 0; i < 10; i++)
{
  std::cout << rand () << std::endl;
}
```

Стойностите на `rand()` са в интервала $[0..INT\_MAX]$. Ако е нужно да генерирате стойности в друг интервал, например $[0..N]$, това може да стане

по формулата $\frac{rand()}{INT\_MAX} \times N$ (трябва да избегнете целочисленото делене)!

3.26. Да се измери емпирично времето за изпълнение на алгоритъма за сортиране по метода на мехурчето. Да се начертае графика на зависимостта на времето за изпълнение от големината на масива. Всеки тест да е с наново генериран масив от случайни числа.

3.27. Да се въведе матрица от числа $A_{N \times M}$.

- Да се сортира всеки от редовете на матрицата
- Да се сортира всяка от колоните на матрицата

Така получените матрици да се отпечатат на стандартния изход.

`Bozosort` е случайностен алгоритъм за сортиране на масиви. При този алгоритъм, на всяка стъпка се разменят две случайни числа от масива, след което се проверява дали масивът се е сортирал. Процесът продължава до сортиране на масива.

3.28. Да се реализира алгоритъма `Bozosort`. Да се измери емпирично времето му за изпълнение. *Внимание: тествайте с достатъчно малки масиви, тъй като този алгоритъм е изключително бавен.*

3.5 Низове II

- Задача 3.55. [1] Дадена е квадратна таблица $A_{n \times n}$ $(1 \leq n \leq 30)$ от низове, съдържащи думи с максимална дължина 6. Да се напише програма, която проверява дали изречението, получено след конкатенацията на думите от главния диагонал (започващо от горния ляв ъгъл) съвпада с изречението, получено след конкатенацията на думите от вторичния главен диагонал на A (започващо от долния ляв ъгъл).

- Задача 3.56. [1] Дадена е квадратна таблица A от n-ти ред $(1 \leq n \leq 20)$ от низове, съдържащи думи с максимална дължина 9. Да се напише програма, която намира и извежда на екрана изречението, получено след обхождане на A по спирала в посока на движението на часовниковата стрелка, започвайки от горния ляв ъгъл. Например ако матрицата A има вида:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

изречението след обхождането по спирала е: “abcfihgde”.

- Задача 3.57 (Inner Join). [1] Нека са дадени два масива от низове – students и grades с най-много 20 низа във всеки. Низовете в масива students имат вида “XXXXXX YYYY...”, където “XXXXXX” е шестцифрен факултетен номер, а “YYYY...” е име с произволна дължина. Низовете в grades имат вида “XXXXXX YYYY”, където “XXXXXX” е шестцифрен факултетен номер, а “YYYY” е оценка под формата на число с плаваща запетая. И двата масива са сортирани във възходящ ред по факултетен номер. Възможно е в някой от двата масива да има данни за факултетни номера, за които няма данни в другия. И в двата списъка даден факултетен номер се среща най-много един път. Да се напише програма, която извежда на екрана имената и оценките на тези студенти, за които има информация и в двата списъка, като оценките са увеличени с 1 единица, но са максимум 6.00.

3.6 Инвариант на цикъл и верификация на програми

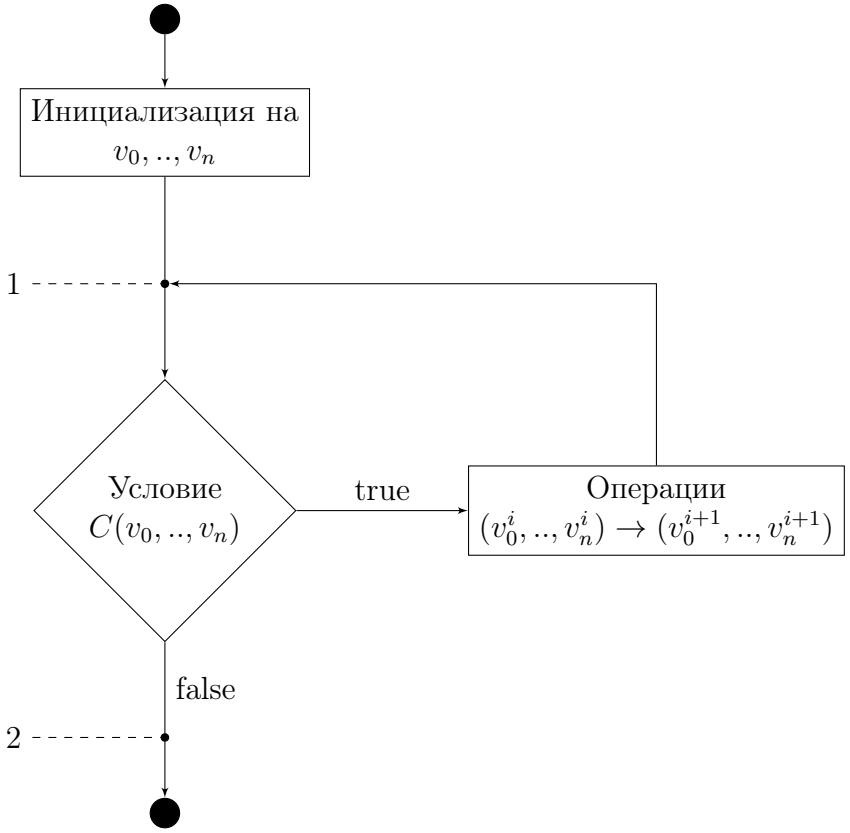
Изложението в тази секция е силно опростена версия на метода на Флойд (Floyd) за верификация на блок схеми, описан в монументалната му статия [4]. Редица идеи и задачи са взаимствани от книгата [5] на А. Соскова и С. Николова “Теория на програмите в задачи”. За по-подробно изложение и допълнителни примери и задачи препоръчваме тази книга. Изложението по-долу е нарочно опростено, като на места е жертвана прецизността му.

Инвариант на цикъл

Нека е дадена програма или част от програма, която се се състои от единствен `while` цикъл. Нека имаме някакъв набор от работни променливи $v_1, .., v_n$, които се инициализират непосредствено преди `while` цикъла. Условието на цикъла зависи изцяло от работните променливи, а тялото на цикъла използва или променя само тях. Приемаме, че програмата не произвежда странични ефекти и не зависи от такива.

Тоест, за програмата знаем, че: (1) не извършва входни и изходни операции, (2) поведението ѝ зависи изцяло от началните стойности на работните променливи $v_1, .., v_n$ и (3) не модифицира никакви други променливи, освен работните. Такава програма можем да изобразим чрез блок схемата на Фигура 5.

На фигурата $C(v_1, .., v_n)$ е някакъв логически израз, зависещ само от $v_1, .., v_n$. Да допуснем, че сме “замразили” изпълнението на програмата в точката, обозначена с “1”, точно преди да започне i-тото поред ($i \geq 0$) изпълнение на тялото на цикъла, т.е. непосредствено преди да бъдат изпълнени операторите в него за i-ти път. В този момент n-торката работни про-



Фигура 5: Блок схема на част от програма с `while` цикъл

менливи $(v_1, .., v_n)$ имат някакви конкретни стойности. Да ги обозначим с $(v_0^i, .., v_n^i)$. След изпълнение на всички оператори в тялото на цикъла работните променливи получават някакви нови стойности. Това са точно стойностите $(v_0^{i+1}, .., v_n^{i+1})$, които работните променливи ще имат в началото на $i+1$-вата итерация. Това е изобразено на фигурата чрез прехода $(v_0^i, .., v_n^i) \to (v_0^{i+1}, .., v_n^{i+1})$.

Нека приемем, че при някакво конкретно изпълнение на програмата, при конкретни дадени начални стойности на работните променливи $(v_0^0, .., v_n^0)$, тялото на цикъла ще бъде изпълнено точно $k \geq 0$ пъти, след което условието $C(v_1, .., v_n)$ на цикъла ще се наруши и той ще приключи. Изпълнението на програмата достига точката, обозначена с "2". По този начин се получава редицата от n-торки $(v_0^0, .., v_n^0), .., (v_0^k, .., v_n^k)$, като за всички нейни членове $0 \leq i < k$, освен за последния, знаем че е вярно $C(v_0^i, .., v_n^i)$, а за последния, k-ти член, е вярно обратното — $\neg C(v_0^k, .., v_n^k)$.

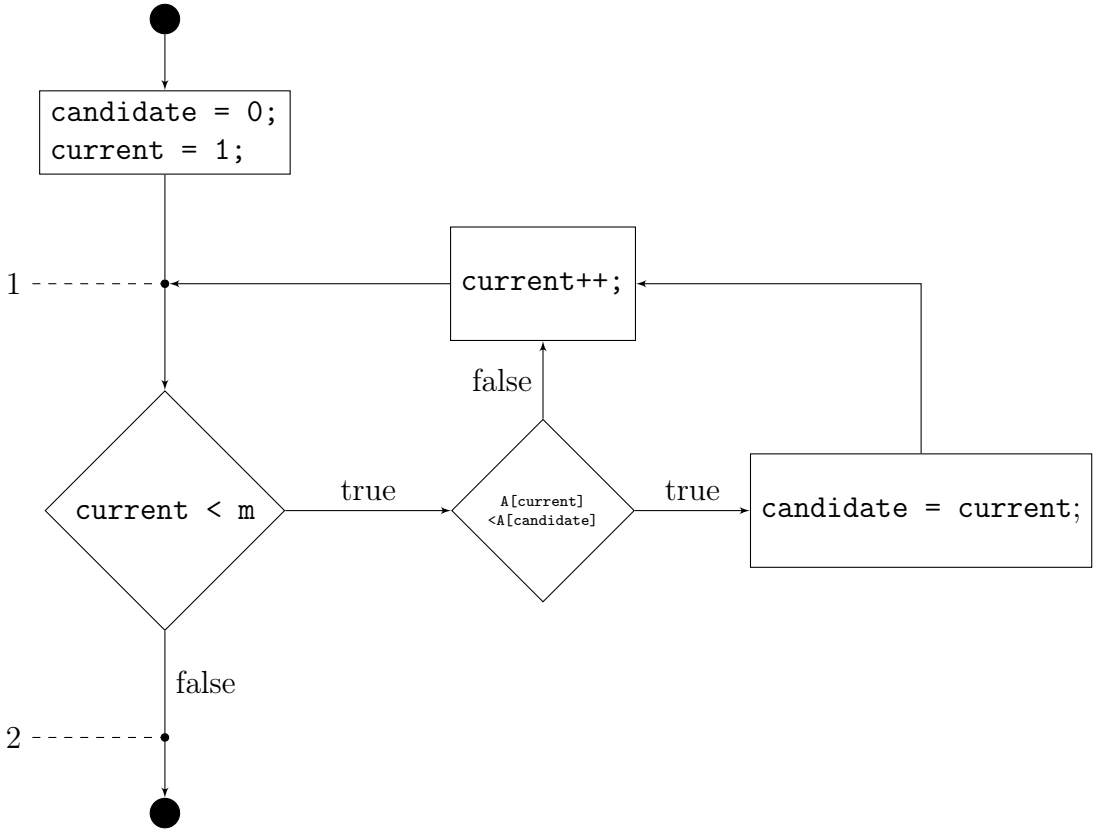
Освен свойството $C(v_0^i, .., v_n^i)$ (или неговото отрицание), което знаем за всички последователни стойности на работните променливи, можем да въведем още едно свойство $I(v_0^i, .., v_n^i)$, наречено "инвариант на цикъла". За свойството I искаме да е вярно за всички възможни стойности на работните променливи, дори и за последния член на редицата им. От там идва името "инвариант", т.е. факт, което е винаги верен.

Това свойство може да е произволно, например тъждествената истина `true` изпълнява условието за инвариант, тъй като е вярна за всички членове на редицата на работните променливи. Инвариантът обаче е най-полезен, когато представя "смисъла" на работните променливи, и ако от верността на $\neg C(v_0^k, .., v_n^k)$ & $I(v_0^k, .., v_n^k)$ можем да изведем нещо полезно за "крайния резултат" от изпълнението на цикъла, съдържащ се в стойностите $(v_0^k, .., v_n^k)$.

Верификация на програми

Понятието "инвариант" ще илюстрираме чрез едно негово приложение: "Верификация на програма". Верификацията на програма е доказателство, че при необходимите начални условия дадена програма изчислява стойност, удовлетворяваща някакво желано свойство. Като пример да разгледаме следната програма, за която ще се уверим, че намира най-малък елемент на едномерен масив A с $m > 0$ елемента $A[0], .., A[m-1]$.

```
size_t candidate = 0, current = 1;
while (current < m)
{
  if (A[candidate] < a[current])
```



Фигура 6: Блок схема на програмата за намиране на най-малък елемент в масив

```
  {
    candidate = current;
  }
  current++;
}
```

Можем да приложим метода на инварианта, за да докажем строго, че когато цикълът приключи, то елементът $A[candidate]$ е гарантирано по-малък или равен на всички останали елементи на масива. Като първа стъпка, за улеснение изобразяваме програмата като блок схемата на Фигура 6.

Работните променливи на програмата, освен масива A, са целочислените променливи `candidate` и `current`. Какъв е смисълът на тези променливи? Веднага се вижда, че `current` е просто брояч — служи за обхождане на елементите на масива последователно от $A[0]$ до $A[m-1]$, като на всяка итерация от цикъла се разглежда стойността на $A[current]$.

Интуитивно се вижда, че `candidate` е намереният най-малък елемент на масива до текущия момент от обхождането. Тоест, можем да се надяваме, че ако сме разгледали първите i елемента на масива, то правилно сме определили, че най-малкият от тях е $A[current]$.

Тези размишления може да запишем чрез инварианта:

$I ::= A[candidate] = min(A[0], .., A[current - 1])$.

Този инвариант очевидно е верен в началото на изпълнението на програмата, когато $current = 1$, а $candidate = 0$. Как да се уверим, че инвариантът е валиден за всички стойности, през които преминават работните променливи?

Нека допуснем, че инвариантът е верен за някаква стъпка i от изпълнението на цикъла. Тоест, допускаме $I(candidate^i, current^i)$, или $A[candidate^i] = min(A[0], .., A[current^i - 1])$.

На итерация $i + 1$ имаме и $current^{i+1} = current^i + 1$, и следователно трябва да покажем, че $A[candidate^{i+1}] = min(A[0], .., A[current^{i+1} - 1])$.

След навлизане в тялото на цикъла, имаме две възможности:

1) $A[current^i] \geq A[candidate^i]$. От допускането следва, че добавянето на $A[current^i]$ към $min(A[0], .., A[current^i - 1])$ не променя стойността на минимума, или $min(A[0], .., A[current^i - 1]) = min(A[0], .., A[current^i - 1], A[current^i])$. От тук директно се вижда, че инвариантът е верен за итерация $i + 1$.

2) $A[current^i] < A[candidate^i]$. Тъй като по допускане $A[candidate^i] = min(A[0], .., A[current^i - 1])$, от тук следва, че $A[current^i] < min(A[0], .., A[current^i - 1])$, следователно

   $A[current^i] = min(A[0], .., A[current^i - 1], A[current^i])$.

Но след присвояването имаме $candidate^{i+1} = current^i$, от където $A[candidate^{i+1}] = min(A[0], .., A[current^i])$. Но $current^i = current^{i+1} - 1$, от където получаваме верността на I за итерация $i+1$: $A[candidate^{i+1}] = min(A[0], .., A[current^{i+1} - 1])$.

Доказателството протича по индукция. Уверяваме се, че инвариантът е верен за цялата редица от стойности на `candidate` и `current`.

Какво се случва в края на цикъла, когато имаме $\neg(current < m)$? Тъй като числата са цели, а `current` се увеличава само с единица, можем да заключим, че $current = m$. Замествайки това равенство в инварианта, получаваме твърдението

$A[candidate] = min(A[0], .., A[m-1])$.

Тоест, намерили сме минималния елемент на масива.

3.29. Да се докаже строго, че за следната функция е вярно $pow(x, y) = x^y$:

а)
```
unsigned int pow (unsigned int x, usnigned int y)
{
  unsigned int p = 1, i = 0;
  while (i < y)
  {
    p *= x;
    i++;
  }
  return p;
}
```

б)
```
unsigned int pow (unsigned int x, usnigned int y)
{
  unsigned int p = 1;
  while (y > 0)
  {
    p *= x;
    y--;
  }
  return p;
}
```

в)
```
unsigned int pow (unsigned int x, usnigned int y)
{
  unsigned int z = x, t = y, p = 1;
  while (t > 0)
  {
    if (t%2 == 0)
    {
      z *= z;
```

```
        t /= 2;
      }else{
        t = t - 1;
        p *= z;
      }
    }
    return p;
  }
```

3.30. Да се докаже строго, че за следната функция е вярно $sqrt(n) = [\sqrt{n}]$.

```
unsigned int sqrt (unsigned int n)
{
  unsigned int x = 0, y = 1, s = 1;
  while (s <= n)
  {
    x++;
    y += 2;
    s += y;
  }
  return x;
}
```

3.31. Да се напише програма, която проверява дали дадени два масива A и B с еднакъв брой елементи m са еднакви, т.е. съдържат същите елементи в същия ред. Да се докаже строго, че програмата работи правилно.

3.32. Да се докаже, че следната функция връща истина тогава и само тогава, когато масивът `A` с `n` елемента съдържа елемента `x`.

```
bool member (int A[], size_t n, int x)
{
  size_t i = 0;
  while (i < n && A[i] != x)
  {
    i++;
  }
  return i < n;
}
```

3.33. Да се напише програма, която проверява дали даден масив A с m елемента е сортиран, т.е. дали елементите му са наредени в нарастващ ред. Да се докаже строго, че програмата работи правилно.

Съществуват редица съвременни методи за верификация на програми. Пълната версия на метода, използван по-горе, се нарича "логика на Флойд-Хоар" (Floyd–Hoare logic) и е само един представител на този клас методи.

Също така, в компютърните науки има направление, наречено "Синтез на програми" (Program refinement). При синтеза на програми се решава обратната задача: по спецификация на входа и изхода да се генерира програма, която удовлетворява спецификацията.

Препоръчваме на любознателния читател да се запознае с логиката на Floyd–Hoare и методите за синтез на програми.

4 Указатели и програмен стек

4.1 Предаване на масиви и указатели като параметри на функции

4.1. Да се дефинира функция, която получава като параметри два масива с еднакъв брой елементи. Функцията да разменя съответните елементи на масивите ($a[i] \leftrightarrow b[i]$).

4.2. Да се дефинира функция `swap([подходящ тип] a,[подходящ тип] b)`, която разменя стойностите на две целочислени променливи, предадени на функцията чрез a и b.

4.2 Масиви от указатели

4.3. Да се напише булева функция `bool duplicates (long *ponters[])`, която получава като параметър масив `pointers` от указатели към целочислени променливи. Функцията да проверява дали поне две от съответните променливи имат еднакви стойности.

4.4. Да се дефинира функцията `bool commonel (int *arrays[],int npointers, int arrlengths[])`. Масивът `arrays` съдържа `npointers` на брой указатели към масиви от цели числа. `i`-тият масив има големина `arrlengths[i]`. Функцията да връща истина, ако има поне едно число x, което е елемент на всички масиви.

4.5. Да се дефинира функцията `bool subarrays (int *arrays[],int npointers, int arrlengths[])`. Масивът `arrays` съдържа `npointers` на брой указатели към масиви от цели числа. `i`-тият масив има големина `arrlengths[i]`. Функцията да връща истина, ако поне един от масивите е подмасив на друг масив. Масивът a наричаме подмасив на b, ако заетата от a памет е част от заетата от b памет. Да се напишат подходящи тестове за функцията.

4.3 Програмен стек

4.6. Да се дефинира рекурсивна функция `double sum(size_t n)`, която въвежда n числа от стандартния вход връща сумата им. *Да не се използват оператори за цикъл!*

4.7. Да се дефинира рекурсивна функция `reverse(n)`, която въвежда n числа от стандартния вход и ги извежда в обратен ред. *Да не се използват масиви. Да се използва програмния стек чрез рекурсия.*

4.8. Да се дефинира функция `void getmax (long *pmax, size_t n)`, която въвежда n числа от стандартния вход и записва максималното от тях в променливата, сочена от указателя `pmax`.

Пример: Следната програма ще изведе най-малкото от 5 въведени от стандартния вход числа.

```
int main ()
{
  long max = -1;
  getmax (&max,5);
  std::cout << max;

  return 0;
}
```

Функцията да се реализира по два начина: чрез цикъл и чрез използване на рекурсия без оператори за цикъл.

5 Рекурсия

5.1 Прости рекурсивни функции

5.1. Задача 5.2.[1] Да се дефинира рекурсивна функция за намиране на стойността на полинома на Ермит $Hn(x)$ (x е реална променлива, а n неотрицателна цяла променлива), дефиниран по следния начин:

$H_0(x) = 1$

$H_1(x) = 2x$

$H_n(x) = 2xH_{n-1}(x) + 2(n-1)H_{n-2}(x), n > 1$

5.2. Задача 5.3.[1] Произведението на две положителни цели числа може да се дефинира по следния начин:

$mult(m, n) = m$, ако n = 1

$mult(m, n) = m + mult(m, n-1)$, иначе.

Да се дефинира рекурсивна функция, която намира произведението на две положителни цели числа по описания по-горе начин.

5.3. Задача 5.5.[1] Да се дефинира функция, която намира най-големия общ делител на две неотрицателни цели числа, поне едното от които е различно от 0.

5.4. Задача 5.7.[1] Дадени са естествените числа n и k ($n \geq 1, k > 1$). Да се дефинира рекурсивна функция, която намира произведението на естествените числа от 1 до n със стъпка k.

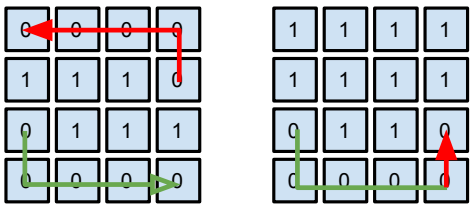
5.5. Задача 5.10.[1] Дадено е неотрицателно цяло число n в десетична бройна система. Да се дефинира рекурсивна функция, която намира сумата от цифрите на n в бройна система с основа k ($k > 1$).

5.6. Задача 5.11.[1] Да се дефинира рекурсивна функция, която установява дали в записа на неотрицателното цяло число n, записано в десетична бройна система, се съдържа цифрата k.

5.7. Задача 5.19.[1] Да се дефинира рекурсивна функция, която проверява дали дадено положително цяло число е елемент на редицата на Фибоначи.

5.8. Задача 5.28.[1] Да се дефинира рекурсивна функция, която намира максималния елемент на редицата от цели числа $a_0, a_1, a_2, ..., a_{n-1}$, където $n \geq 1$.

Забележка: Редицата е представена като масив.



Фигура 7: Примерени лабиринти

5.9. Задача 5.31.[1] Да се напише функция

`void insertSorted (long x, long arr[], long n)`,

която включва цялото число `x` число в сортирана във възходящ ред редица от цели числа `arr`, в която има записани `n` елемента. Вмъкването да запазва наредбата на елементите. Предлолага се, че за редицата е заделена достатъчно памер за допълване с още едно число.

5.10. Задача 5.34.[1] Да се дефинира рекурсивна функция, която сравнява лексикографски два символни низа.

5.2 Търсене с пълно изчерпване

5.11. Нека е дадена квадратна матрица от цели числа $N \times N$, представяща "лабиринт". Елементи на матрицата със стойност 0 смятаме за "проходими", а всички останали - за "непроходими". Път в лабиринта наричаме всяка последователност от проходими елементи на матрицата, които са съседни вертикално или хоризонтално, такава че (1) никой елемент от последователноста не е последван директно от предшественика си (забранено е "връщането назад") и (2) най-много един елемент на последователноста се среща в нея повече от веднъж (има най-много един "цикъл").

Да се дефинира функция `bool downstairs (int sx, int sy, int tx, int ty)`, която проверява дали съществува път от елемента (sx, sy) до елемента (tx, ty), такъв, че всеки следващ елемент от пътя е или вдясно, или под предишния. Такъв път да наричаме "низходящ".

Пример: На Фигура 7(а) такъв път съществува от елемента $(0, 2)$ до елемента $(3, 3)$, но не и от $(3, 1)$ до $(0, 0)$.

5.12. При условията на дефинициите от предишната задача, да се дефинира функция `bool connected()`, която проверява дали от всеки елемент на

матрицата (sx, sy) до всеки елемент на матрицата (tx, ty), такива, че $sx \leq tx$ и $sy \leq ty$, съществува низходящ път.

Пример: За лабиринта от Фигура 7(а) условието е изпълнено, но не и за лабиринта отФигура 7(б).

5.13. Да се напише програма, която по въведени от клавиатурата $4 \leq n \leq 8$ и $0 \leq k \leq n$ намира извежда на екрана всички възможни конфигурации на абстрактна шахматна дъска с размери $n \times n$ с разположени на нея k коня така, че никоя фигура не е поставена на поле, което се "бие" от друга фигура според съответните шахматни правила.

Пример за отпечатана конфигурация с $n = 5, k = 2$:

```
_ _ _ _ _
_ _ H _ _
_ _ _ _ _
_ _ _ _ H
_ _ _ _ _
```

5.14. При условията на задача 5.11. да се напише функция

`int minDistance (int sx, int sy, int tx, int ty)`,

която по въведени от клавиатурата координати на елементи $s = (sx, sy)$ и $t = (tx, ty)$ намира *дължината* на най-краткия път между s и t. Обърнете внимание, че се иска *най-къс*, а не просто низходящ път.

5.15. При условията на задача 5.11. да се напише функция, която по въведени от клавиатурата координати на елементи $s = (sx, sy)$ и $t = (tx, ty)$ намира и отпечатва на екрана елементите, от които се състои най-краткия път между s и t. Обърнете внимание, че се иска *най-къс*, а не просто низходящ път.

5.16. Пъзел на Синди[6].

Дадена е игрова дъска като на фигура 2, която се състои от n черни и n бели фигури. Фигурите могат да бъдат разположени на $2n + 1$ различни позиции. Играта започва с разполагане на всички черни фигури вляво, а всички бели - вдясно на дъската.

Черните фигури могат да се местят само надясно, а белите - само наляво. На всеки ход важат следните правила:

- всяка фигура се мести само с по една позиция, ако съответната позиция не е заета;

- ако позицията е заета, фигурата X може да прескочи точно една фигура Y от противоположния цвят, ако позицията след Y е свободна.

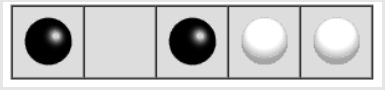
Да се напише програма, която по въведено число n отпечатва на екрана инструкции за игра така, че в края на играта всички бели фигури да са вляво на дъската, а всички черни - вдясно. Инструкциите да са от следния вид:

```
...
Преместете черна фигура фигура от позиция 1 на позиция 2.
Преместете бяла фигура от позиция 5 на позиция 3.
...
```

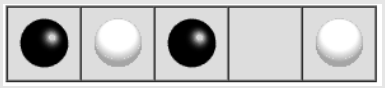
Допустимо е инструкциите да бъдат отпечатани в обратен ред.

На следните фигури е даден пример за игра:

1. Начална конфигурация.

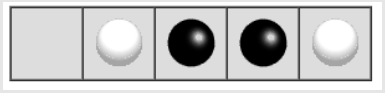


2. Преместване на черна фигура с един ход надясно.



3. Преместване на бяла фигура с прескачане.

4. Преместване на черна фигура с един ход надясно.



5. Преместване на черна фигура чрез прескачане

След ход 5 конфигурацията на играта е безперспективна.

Някои от задачите по обектно-ориентирано програмиране са решени в сборника [7]*Магдалина Тодорова, Петър Армянов, Калин Николов, "Сборник от задачи по програмиране на C++. Част втора. Обектно-ориентирано програмиране"*. За тези задачи е запазена номерацията в сборника.

6 Структури

Задачите за полиноми са на базата на разработените на лекции примери за полиноми, представени чрез структурата:

```
const size_t maxPower = 50;
struct Poly
{
  double coefs[maxPower];
  size_t power;
};
```

6.1. Да се реализира функция `diff` за за намиране на първата производна на полином относно променливата `x`.

Задачата да се реши в два варианта: като функция изображние Poly → Poly и като "деструктивна" функция с тип на резултата `void`*, която изменя стойността на аргумента си.*

И двете функции да се тестват с подходящи примери!

6.2. Да се реализира функция `prod` за умножение на два полинома.

6.3. Задача 1.1. [7] Нека е дефинирана структурата Product:

```
struct Product
{
  char description[32];
  //описание на изделие
  int partNum;
  //номер на изделие
  double cost;
  //цена на изделие
};
```

а) Да се създадат две изделия и се инициализират чрез следните данни:

description	partNum	cost
screw driver	456	5.50
hammer	324	8.2-0

б) Да се изведат на екрана компонентите на двете изделия;

в) Да се дефинира масив от 5 структури Product. Елементите на масива да не се инициализират;

г) Да се реализира цикъл, който инициализира масива чрез нулевите за съответния тип на полетата стойности;

д) Да се променят елементите на масива така, че да съдържат следните стойности:

description	partNum	cost
screw driver	456	5.50
hammer	324	8.20
socket	777	6.80
plier	123	10.80
hand-saw	555	12.80

е) Да се изведат елементите на масива на конзолата с подходящо форматиране;

6.4. Задача 1.4. [7] Да се дефинират структурите polar и rect, задаващи вектор с полярни и с правоъгълни координати съответно. Да се дефинират функции, които преобразуват вектор, зададен чрез правоъгълни координати, в полярни координати и обратно, както и функции, които извеждат вектор, зададен чрез полярните си и чрез правоъгълните си координати.

В главната функция да се дава възможност за избор на режим на въвеждане: r – за въвеждане в правоъгълни и p – в полярни координати. За всеки избран режим да се въведат произволен брой вектори, да се преобразуват в другия режим и да се изведат.

6.5. Задача 1.8.[7] Да се дефинира функция, която сортира лексикографски във възходящ ред редица от точки в равнината. За целта да се дефинира структура Point, описваща точка от равнината с декартови координати.

6.6. Задача 1.Б.5.[7] Да се дефинира структура Planet, определяща планета по име (символен низ), разстояние от слънцето, диаметър и маса (реални числа). Да се дефинират функции, изпълняващи следните действия:

а) въвежда данни за планета от клавиатурата;

б) извежда данните за планета;

в) връща като резултат броя секунди, които са необходими на светлината да достигне от слънцето до планетата (да се приеме, че светлината има скорост 299792 km/s и че разстоянието на планетата до слънцето е зададено в километри).

г) създава едномерен масив от планети с фиксиран размер и въвежда данните за тях от стандартния вход;

д) извежда данните за планетите от масив, подаден на функцията като параметър;

е) отпечатва данните за планетата с най-голям диаметър от масив, подаден на функцията като параметър;

7 Динамична памет

7.1 Заделяне на динамична памет

7.1. Задача 1.4.24. [1] Да се дефинира функция `strduplicate`, която създава копие на символен низ. Функцията да се грижи за заделянето на памет за новия низ.

7.2. Задача 1.4.25. [1] Да се дефинира функция, която преобразува положително цяло число в съответния му символен низ и връща така построения символен низ.

7.3. Задача 1.4.30. [1] Обединение на два символни низа s_1 и s_2 наричаме всеки символен низ, който съдържа без повторение всички символи на s_1 и s_2. Да се дефинира функция, която намира и връща обединението на два символни низа.

7.4. Задача 1.4.36. За [1] работа със символни низове могат да бъдат използвани следните основни функции:

- `char car(const char* x)`, която връща първия символ (елемент) на низа x;
- `char* cdr(char* x)`, която връща останалата част от низа x след отделянето на първия елемент на низа x;
- `char* cons(char x, const char* y)`, която връща указател към символен низ, разположен в динамичната памет и съдържащ конкатенацията на символа x със символния низ y;
- `bool eq(const char* x, const char* y)`, която връща true тогава и само тогава, когато низовете съвпадат.

Да се дефинират описаните функции. Като се използват тези функции, да се дефинират следните функции:

- `char* reverse(char* x)`, която връща указател към символен низ, разположен в динамичната памет и съдържащ символите на x, записани в обратен ред;
- `char* copy(char* x)`, която връща указател към символен низ, разположен в динамичната памет и съдържащ копие на символния низ x;
- `char* car_n( char* x, int n)`, която връща указател към символен низ, разположен в динамичната памет и съдържащ първите n символа на символния низ x;

- `char* cdr_n(char* x, int n)`, която връща останалата част от низа x след отделянето на първите n символа. Предварително е известно, че x притежава поне n символа;
- `int number_of_char( char* x, char ch)`, която намира колко пъти символът ch се среща в символния низ x;
- `int number_of_substr( char* x, char* y)`, която намира колко пъти символният низ y се среща в символния низ x;
- `char* delete_substr(char* x, char* y)`, която връща указател към символен низ, разположен в динамичната памет и съдържащ символите на низа x, от който са изтрити всички срещания на символния низ y.

7.2 Масиви от структури в динамичната памет и текстови файлове

7.5. Решението на задача 6.3. да се разшири така, че масив от структури да може да се протича от и записва във текстов файл. Броят на записаните във файла структури да може да е произволен и да се използва динамична памет за инициализация на масива с необходимия брой елементи.

7.6. Решението на задача 6.5. да се разшири като се добави диалогов режим (т.нар. "меню"), чрез който може да се модифицира съдържанието на глобална редица (масив) от точки в равнината:

а) Да може към редицата от точки да се добавят $n \geq 2$ точки от отсечка с начало точката (x_1, y_1) и край точката (x_2, y_2). Координатите на началото и края, както и числото n, се задават от потребителя. Точките да са на равно разстояние помежду си.

б) Да могат да се изпишат на стандартния изход всички точки от редицата, които са в посочен от потребителя кавадрант.

в) Да може да се премахнат всички точки, лежащи на дадена права. Правата се въвежда чрез коефициентите на уравнението на правата $ax + by + c = 0$.

г) Редицата от точки да може да се зпише в текстов файл "points.dat".

д) Редицата от точки да може да се прочете от текстов файл "points.dat". Ако текущата работна редицата не е празна, съществуващите точки се изтриват.

8 Шаблони и указатели към функции

8.1 Шаблони на функции

8.1. Да се реализира шаблон на функция `void input ([подходящ тип] array, int n)`, която въвежда от клавиатурата стойностите на елементите на масива `array` от произволен тип `T` с големина `n`.

Какви са допустимите типове `T` за този шаблон? Защо функцията е от тип `void`?

Да се реализира и изпълни подходящ тест за функцията.

8.2. Да се реализира шаблон на фукнция `bool ordered ([подходящ тип] array, int n)`, която проверява дали елементите на масива `array` от произволен тип `T` с големина `n` образуват монотонно-растяща редица спрямо релацията `<`.

Какви са допустимите типове `T` за този шаблон?

Да се реализира и изпълни подходящ тест за функцията.

8.3. Да се реализира шаблон на фукнция `bool member ([подходящ тип] array, int n, [подходящ тип]x)`, която проверява дали `x` е елемент на масива `array` от произволен тип `T` с големина `n`.

Има ли в C++ тип `T`, който не е съвместим с този шаблон?

Да се реализира и изпълни подходящ тест за функцията.

8.2 Функции от високо ниво

8.4. Да се дефинира масив `functions` с 5 елемента от тип фунцкия *double* → *double*. Да се дефинират 5 произволни функции от този тип и адресите им да се се присвоят на елементите на масива.

При въведено от клавиатурата число x : *double*, да се намери и отпечата индексът на тази функция в масива `functions`, чиято стойност е най-голяма в точката `x` спрямо стойностите на всички функции в масива. Ако има няколко такива функции, да се отпечата индекса на коя да е от тях.

8.5. Да се дефинира функция `double fmax([подходящ тип]f, [подходящ тип]g, double x)`, където `f` и `g` са две произволни функции от тип *double* → *double*, за които приемаме, че са дефинирани в `x`. Функцията да връща по-голямата измежду стойностите на `f` и `g` в точката `x`.

Да се реализира и изпълни подходящ тест за функцията.

8.6. Да се дефинира фунцкия `double maxarray ([подходящ тип] array, int n, double x)`, където `array` е масив от финкции от тип *double* → *double* с големина `n`.

Фунцкцията `maxarray` да връща най-голямата измежду стойностите на всички функции от масива в точката `x` като приемаме, че всички те са дефинирани в тази точка.

Задачата да се реши със и без използването на функцията `fmax` от предишната задача.

Да се реализира и изпълни подходящ тест за функцията.

8.7. Нека е дадена следната структура: `struct S {int a; int b; int c;};`. Да се дефинира функция `void sort([подходящ тип]array, int n, [подходящ тип]compare)`, където `array` е масив от `n` структури от тип `S`.

Типът на функцията `compare` да се подбере така, че чрез нея да може да се реализира произволна наредба за типа `S`, т.е. функцията да може да сравнява "по големина" две структури от `S` по произволен критерий.

Да се създаде и инициализира масив с 5 структури от тип `S`. Като се използва функцията `sort` да се сортира масива по веднъж по всеки от следните начини:

а) по полето `a`

б) по полето `b`

в) лексикографски по тройката (a, b, c)

8.3 Map и Reduce

8.8. Да се изведат всички елементи на масив чрез функцията `map`.

8.9. Нека е дадена следната структура `struct S {int a; int b; int c;}`. Да се дефинира и попълни примерен масив `A` с елементи от `S`.

а) Чрез подходяща помощна функция и използване на `map`, да се отпечата сумата на полетата `a`, `b` и `c` на всеки от елементите на `A`.

б) Чрез подходяща помощна функция и използване на `map`, да се въведат елементите на `A`.

в) Чрез подходяща помощна функция и използване на `map`, да се увеличи с единица всяко поле `a` на елементите на `A`.

г) Чрез подходяща помощна функция и използване на `map`, да се разменят стойностите на полетата `a` и `b` на елементите на `A`.

д) Да се тестват решенията на горните задачи.

9 Класове I: Прости класове, методи и оператори

9.1. Задача 2.39.[7] Да се дефинира клас `Time`, който определя момент от денонощието по зададени час и минути. Класът да съдържа подходящи методи за:

- достъп и промяна на часа и минутите с проверки за коректност;
- добавящ към времето цяло число минути;
- достъп до боря минути, изминали от началото на денонощието;
- оператор за сравнение (казваме, че $t_1 < t_2$, ако t_2 е по-късно в денонощието от t_1).

Да се предефинират операторите +, - и *, така че да могат да се събират и изваждат две времена, както и да се умножават време с цяло число и цяло число с време. Да се включи дефинираният клас в програма и направят обръщения към член-функциите му и предефинираните оператори.

9.2. Да се дефинира структура `Point`, описваща точка в евклидовата равнина и клас `Line`, описващ права в евклидовата равнина, зададена чрез две нейни точки.

Класът `Line` да съдържа методи, чрез които може да се извършват следните операции:

- Проверка дали две прави са успоредни;
- Проверка дали дадена точка лежи на дадена права;
- Намиране на пресечната точка на две прави. Приемаме, че правите не са успоредни. Стойността на резултата може да е произволна в противен случай.
- Създаване на права, която е ъглополовяща на по-големия ъгъл, образуван от две прави. Стойността на резултата може да е произволна в противен случай.

Където е подходящо да се дефинират оператори вместо методи.

9.3. Задача 2.44. [7] (асоциативен масив) Да се дефинира клас `Dictionary`, който създава тълковен речник. Тълковният речник се състои от не повече от 500 двойки дума–тълкуване, като думата е символен низ с не повече от 100 символа, а тълкуването е символен низ с не повече от 500 символа.

- Да се дефинира подходяща структура, описваща една двойка дума-тълкувание;
- Да се дефинират подходящи член-данни на клас `Dictionary`;

Клас `Dictionary` да съдържа методи, с които може да се извършват следните операции над речника:

- Инициализация на празен речник;
- извеждане на всички думи в речника и техните тълкуания;
- включване на нова двойка дума–тълкуване в речника;
- изключване на двойка дума–тълкуване от речника (по дадена дума);
- търсене на значението на дадена дума в речник.
- извеждане на всички думи в речника и техните тълкуания по азбучен ред на думите;

Да се дефинира оператор +, обединяващ два речника, такъв че:

- Ако някои думи имат значение и в двата речника, значенията да се конкатенират в резултатния сумарен речник;
- Ако общият брой на думите в двата речника надхвърля 500, да се използват само първите 500 думи (при произволна наредба).

10 Жизнен цикъл на обектите

10.1 Конструктори

10.1. Да се дефинира клас Word, описващ дума, съставена от не повече от 20 символа от тип char. Класът да съдържа следните операции:

- оператор `[]` за намиране на `i`-тия пореден символ в думата
- оператори `+` и `+=` за добавяне на един символ в края на думата. Ако думата вече има 20 символа, операторите да нямат ефект
- оператори `<` и `==` за сравнение на думи спрямо лексикографската наредба
- подходящи конструктори

Да се реализира и изпълни подходящ тест за класа и неговите методи.

10.2. Да се реализира клас `NumbersSummator`, който поддържа сума на цели числа. При създаване на обект от класа, съответната му сума да се инициализира с число, което се подава като аргумент на конструктора. За класа да се реализират следните методи:

- sum, който връща текущата стойност на сумата
- add, увеличаващ сумата с дадено число
- sub, намаляващ сумата с дадено число
- num, връща колко пъти сумата е била променяна
- average, връщащ средното аритметично на всички числа, с които сумата е била променяна.

*Забележка:*Функционалност извън тези 4 метода, като например съхраняване на отделните числа от поредицата, не е необходима. Пример:

```
NumbersSummator seq1 (10);
seq1.add (10);
seq1.add (5);
seq1.sub (15);
cout << seq1.sum() ; //->10 (10+10+5-15)
cout << seq1.average(); //->0 (10+5-15)/3
```

10.3. Да се дефинира клас `BrowserHistory`, който съдържа информация за историята на посещението до най-много `N` Web сайта. `N` е параметър на конструктора на класа. За целта да се реализира структура `HistoryEntry`, описваща едно посещение на сайт чрез:

а) Месец от годината, през който е посетен сайтът;

б) Неговото URL.

Класът codeBrowserHistory да поддържа следните операции:

- Метод за добавяне на нов сайт към историята. Информацията за всеки сайт се въвежда от клавиатурата
- Оператор += с параметър `HistoryEntry`, добавящ сайт към историята
- Метод за отпечатване на информацията за всички сайтове в историята
- Метод, който по даден месец от годината намира броя на сайтовете, посетени през този месец
- Намиране на този месец от годината, в който има най-много посетени сайтове
- Премахване на най-скоро добавеният сайт в историята
- Оператор +, който обединява двете истории

Да се реализира и изпълни подходящ тест за класа и неговите методи.

10.2 Копиране

10.4. За клас `BrowserHistory` от задача 10.3. да се реализират конструктор за копиране, оператор за присвояване, оператори за събиране `+` и `+=`, обединяващи две истории и деструктор.

Да се реализира подходящ тест на класа.

10.5. Задача 2.2.44. (асоциативен масив) [7] Да се дефинира клас `Dictionary`, който създава тълковен речник. Тълковният речник се състои от не повече от 500 двойки дума–тълкуване, като думата е символен низ с не повече от 100 символа, а тълкуването е символен низ с не повече от 500 символа.

- Да се дефинира подходяща структура, описваща една двойка дума-тълкувание;
- Да се дефинират подходящи член-данни на клас `Dictionary`;

Клас `Dictionary` да съдържа методи, с които може да се извършват следните операции над речника:

- Инициализация на празен речник;
- извеждане на всички думи в речника и техните тълкуания;
- включване на нова двойка дума–тълкуване в речника;
- изключване на двойка дума–тълкуване от речника (по дадена дума);
- търсене на значението на дадена дума в речник.
- извеждане на всички думи в речника и техните тълкуания по азбучен ред на думите;

Да се дефинира оператор $+$, обединяващ два речника, такъв че:

- Ако някои думи имат значение и в двата речника, значенията да се конкатенират в резултатния сумарен речник;
- Ако общият брой на думите в двата речника надхвърля 500, да се използват само първите 500 думи (при произволна наредба).

10.6. Клас `Dictionary` от задача 10.5. да се реализира така, че максималният брой `N` на двойки ключ-стойност, които могат да бъдат добавени към речника, да се задава като параметър на конструктора на класа. За класа да се реализират конструктор за копиране, оператор за присвояване

и деструктор.

Да се реализират оператори за събиране `+` и `+=`, обединяващи два речника. Ако в речниците `a` и `b` има еднакви думи с различни значения, то за тези думи в речника `a+b` да се използва значението им от речника `a`.

Да се реализира подходящ тест на класа.

Някои от следващите задачи са върху примерния шаблон `Vector`, разработен на лекции. Шаблонът реализира прост контейтер чрез масив в динамичната памет:

```
template <typename T>
class Vector
{
public:
    Vector();
    Vector(const Vector<T> &);
    Vector<T>& operator=(const Vector<T> &);
    ~Vector ();
    Vector<T> operator + (const Vector<T> &) const;
    Vector<T>& operator+=(const Vector<T> &);
    T& operator[] (size_t i);
    T operator[](size_t i) const;
    void push (const T& x);
    void print () const;
    size_t size() const;
  private:
    T* elements;
    size_t nCapacity;
};
```

10.7. За шаблона `Vector` от лекции да се дефинира метод `Vector::resize`, с който да може динамично да се променя капацитета на масива. При намаляване на капацитета да отпадат най-левите елементи на масива. При увеличаване на капацитета на масива, новите елементи да остават неинициализирани.

Да се реализират подходящи тестове.

10.8. Като се използва шаблона `Vector` да се създаде масив `M` то 3 елемента, чиито елементи са масиви от по 3 числа от тип `double`. Да се въведат елементите на `M` от клавиатурата.

10.9. За шаблона на клас `Vector` от лекции да се дефинира метод:

`Vector<Vector<T>> Vector<T>::slice(size_t n)`.

Ако приемем, че изходният масив е с елементи от тип T, то методът `slice` създава и връща масив от масиви, т.е. резултатният масив се състои от масиви с елементи от тип T.

Методът да "разделя" изходният масив на равни по големина части с по `n` последователни елемента. `i`-тият поред масив от резултата съдържа i-тата поредна n-торка от последователни членове на изходния масив. Последният масив в резултата може да съдържа по-малко от `n` елемента, ако броят на елементите на изходният масив не е кратен на `n`.

Пример: Нека масивът `a` има елементите `[1,2,3,4,5,6,7,8,10,11]`. При тези условия, `a.slice(3)` създава и връща масива от масиви `[[1,2,3],[4,5,6],[7,8,9],[10,11]]`.

Да се напишат подходящи тестове.

11 Ламбда функции

Някои от следните задачи са взаимствани от библиотеката на `JavaScript` за функцоонално програмиране "lodash". За примери и повече информация: документация на библиотеката.

11.1. Да се дефинира функция `negate(p)`, където $p : A \to bool$ е едноместен предикат. `negate` да връща предиката $\neg p$.

11.2. Да се дефинира функция `repeated(k,f)`, където $k \geq 0$ е естествено число, а $f : A \to A$ е едноместна функция. Ако $h = repeated(k, f)$, то $h : A \to A$ е такава, че $h = f^k(x) = \underbrace{f(f...}_{k}(x))$.

11.3. Да се дефинира фунцкия `createfn(args,values)`, където `args` е вектор с елементи от тип `U`, а values е вектор с елементи от тип `V`. Двата вектора са с еднакъв брой елементи. Ако $h = createfn(args, values)$, то $h : U \to V$. По дефиниция $h(u) = v$ т.с.т.к. `u` е елемент на `args` с индекс `i`, а $v = values[i]$ (при повече от едно срещания на `u` приемаме най-малкия индекс). Ако `u` не е елемент на `values`, функцията h е недефинирана.

11.4. Да се дефинира функция `switch(n,f,g)`, където $n \geq 1$ е естествено число, а $f, g : A \to B$ са едноместни функции. `switch` да връща функция $h : A \to B$, която при първите си `n` извиквания да дава същите стойности като `f`, а след това - като `g`.

11.5. Да се дефинира функция `before(n,f)`, където $n \geq 1$ е естествено число, а $f : A \to B$ е едноместна функция от произволен тип. `before` да връща функция $h : A \to B$, която при първите си `n` извиквания да дава същите стойности като `f`, а след това - последната върната от `f` стойност.

12 Линейни едносвързани списъци

Следващите задачи са върху примерния шаблон `LList`, разработен на лекции. Шаблонът реализира прост контейтер чрез линеен едносвързан списък:

```
template <typename T>
struct box
{
  box(const T, box<T>*);
  box();
  T data;
  box<T>* next;
};
template <typename T>
class LList
{
public:
  LList ();
  LList (const LList<T>&);
  LList<T>& operator = (const LList<T>&);
  void push (const T&);
  void pop ();
  void insertAt (size_t, const T&);
  void deleteAt (size_t);
  void print () const;
  ~LList ();
private:
  box<T> *first;
};
```

Следните задачи да се решат като упражнение за директно боравене с указателите и двойните кутии, вместо да се свеждат до използването на вече готови методи от реализацията на класа. Т.е. решенията на задачите да не ползват други методи, освен ако не са помощни функции, специално написани за тях.

12.1. Да се реализира метод `int LList<T>::count(int x)`, който преброява колко пъти елементът `x` се среща в списъка.

12.2. Да се реализира конструктор с два аргумента x и y от тип *int*. Конструкторът създава списък с елементи $x, x+1, ..., y$, при положение, че $x \leq y$.

12.3. Да се реализира метод `LList<T>::push_back` за дабавяне на елемент от тип `T` към *края* на списъка.

12.4. Да се реализира метод оператор `LList<T>::operator +=` за дабавяне на елемент от тип `T` към *края* на списъка.

12.5. Да се реализира метод `LList<T>::get_ith(int n)` за намиране на `n`-тия поред елемент на списъка.

12.6. Да се реализира метод `LList<T>::push_back` за дабавяне на елемент от тип `T` към *края* на списъка.

12.7. Да се реализира метод `LList::removeAll (x)`, който изтрива всички срещания на елемента `x` от списъка.

12.8. Да се реализира метод l_1`.append(`l_2`)`, която добавя към края на списъка l_1 всички елементи на списъка l_2.

12.9. Да се реализира метод `LList<T>::concat`, който съединява два списъка в нов, трети списък. Т.е. l_1`.concat(`l_2`)` създава и връща нов списък от елементите на l_1, следвани от елементите на l_2.

12.10. Да се дефинират оператори `LList<T>::operator+=` и `LList<T>::operator+`, съответни на методите `append` и `concat`.

12.11. Да се дефинира оператор за индексиране, позволяващ четене и писане на елемент на даден индекс в списъка.

12.12. Да се дефинира метод `LList::reverse`, който обръща реда на елементите на списъка. Например, списъкът с елементи $1, 2, 3$ ще се преобразува до списъка с елементи $3, 2, 1$.

12.13. Да се дефинира функция `map` за списъци във функционален и деструктивен вариант.

12.14. Да се дефинира функция `reduce` за списъци.

13 Клас символен низ

Следващите задачи са върху примерния клас, реализиращ символен низ, разработен на лекции. Шаблонът реализира символен низ в динамичната памет:

```
class String
{
public:
  String ();
  String (const String<T>&);
  String (const char*);
  String& operator = (const String&);
  ~String();

  char operator [] (size_t) const;
  bool operator != (const String&) const;

  friend std::ostream &operator<<(std::ostream &, const String &);
  ~String ();
private:
  char *str;
};
```

13.1. Да се резлизира метод `substring (startIndex, endIndex)` на клас `String`, намиращ подниз с дадено начло и край.

13.2. Да се резлизира метод `substring (s)` на клас `String`, който намира индекса на първото срещане на подниза `s` в дадения низ, или `-1`, ако `s` не е подниз на дадения низ.

13.3. Да се релизира метод `split(char separator)` на клас `String`. Методът да връща `std::vector` от символни низове (обекти от клас `String`), които се получават като се раздели на части дадения низ според дадения разделител. Например, низът "*Hello wold, have a nice day!*" с разделител символа ',' ще генерира масива {"Hello world", " have a nice day!"}.

13.4. Да се добавят оператори `+=` към класа `String`, позволяващи добавяне към края на низа на: символ (`char`), цяло число (`int`), булева стойност (`bool`).

13.5. Да се добави оператор `=` към класа `String`, позволяващ на низ да се присвои произволен вектор с елементи от произволен тип `T`. Резултатният низ s от присвояването $s = v$ да има формата "$\{v_1, .., v_k\}$", където $v_1, ..v_k$

са представянията на елементите на v като низове. Приемаме, че типа `T` е някой от типовете, поддържани от оператора `+=` на клас `String`.

13.6. (*)Да се реализира оператор за вход от поток. *Внимание: Да се съобрази, че броят на въведените символи от оператора `>>` за `char[]` не е предварително известен!*

14 Наследяване и полиморфизъм

14.1 Наследване, виртуални методи и абстрактни класове

14.1. Шахматни фигури.

а) Да се дефинира структура `ChessPosition`, описваща коректна позиция на фигура върху шахматна дъска ('A'-'H',1-8). Да се дефинира абстрактен клас `ChessPiece`, описващ шахматна фигура със следните операции:

- `ChessPosition getPosition ()`: Дава позицията на фигурата на дъската
- `[подходящ тип] allowedMoves ()`: Дава списък с всички възможни позиции, до които дадена фигура може да достигне с един ход
- `bool wins (ChessPosition)`: Проверява дали фигурата "владее" дадена позиция, т.е. дали позицията е в списъка с възможни ходове на фигурата

б) Да се дефинират класовете `Rook` и `Knight`, наследници на `ChessPiece`, описващи съответно шахматните фигури топ и кон.

в) "Стабилна конфигурация" наричаме такава подредба на фигурите по дъската, при която никоя фигура да не е върху позволен ход на друга фигура (т.е. никои две фигури не се "бият"). Да се дефинира функция `allMoves ([подходящ тип] pieces[, ...])`, която по списъка `pieces`, съдържащ произволен брой разнородни шахматни фигури, отпечатва на конзолата всеки възможен ход на фигура от `pieces` такъв, че след изпълнението му списъка с фигури представлява стабилна конфигурация. Информацията за ходовете да съдържа типа на фигурата, старата позиция и новата позиция, например:

```
Queen A1 -> B2
Knight B3 -> A5
```

Забележка: Реализирайте всички конструктори и други операции, които смятате, че са необходими на съответните класове.
Забележка: Под "списък" се има предвид обект от класовете за линеен едносвързан списъл или динамичен масив, разработени на лекции, или който е да е друг тип, който познавате и който представлява контейнер за обекти.

14.2. Походова игра.

Нека GameBoard е дадена квадратна матрица `NxN` от цели числа, всеки елемент на която има стойност 0, 1 или 2. Елементите на матрицата със стойност 0 наричаме "земя", тези със стойност 1 наричаме "огън", а тези със стойност 2 - "вода". `GameBoard` ще наричаме "игрова дъска". Под "съседна позиция" на позицията (i, j) ще разбираме тези елементи на матрицата (i', j'), такива че $|i - i'| \leq 1$ и $|j - j'| \leq 1$.

В условието по-долу `N` и `GameBoard` да се приемат за предварително дефинирани глобални променливи.

а) Да се дефинира структура `Position`, описваща позиция на игровата дъска, задаща реда и колоната на нейн елемент. Да се дефинира абстрактен клас `GamePlayer`, който описва играч на игровата дъска със следните операции:

- `Position getPosition ()`: Дава позицията на играча на дъската
- `[подходящ тип] allowedMoves ()`: Дава списък с всички възможни позиции, до които даден играч може да достигне с един ход
- `bool wins (Position)`: Проверява дали играча "владее" дадена позиция, т.е. дали позицията е в списъка с възможни ходове на играча

б) Да се дефинира клас `Knight`, наследник на `GamePlayer`, описващ "сухопътен рицар". Позволените ходове на сухопътния рицар се задават със следното правило: Рицарят може да се премести в позициите, които са съседни на текущата му и които:

- са земя
- нямат огън в съседство

в) Да се дефинира клас `SeaMonster`, наследник на `GamePlayer`, описващ "морско чудовище". Позволените ходове на морското чудовище се задават със следното правило: Морското чудовище може да се премести във всички позиции, които са достижими от неговата при придвижване по хоризонтала или вертикала, което преминава само през вода. Например, ако вдясно от играча има три поредни позиции с вода, следвани от една позиция земя, то и трите водни позиции са достижими, но земната и всички вдясно от нея - не.

г) "Стабилна конфигурация" наричаме такава подредба на играчите по дъската, при която никой играч да не е върху позволен ход на друг играч (т.е. никои два играча не се "бият"). Да се дефинира функция `allMoves ([подходящ тип] players[, ...])`, която по списъка `players`, съдържащ произволен брой разнородни играчи, отпечатва на конзолата всеки възможен ход на играч от `players`

такъв, че след изпълнението му списъка с играчи представлява стабилна конфигурация. Информацията за ходовете да съдържа типа на играча, старата позиция и новата позиция, например:

```
Knight (0,0) -> (1,1)
SeaMonster (2,2) -> (5,2)
```

Забележка: Реализирайте всички конструктори и други операции, които смятате, че са необходими на съответните класове.
Забележка: Под "списък" се има предвид обект от класовете за линеен едносвързан списъл или динамичен масив, разработени на лекции, или който е да е друг тип, който познавате и който представлява контейнер за обекти.

14.3. Задача 2.4.25[7] В софтуерна фирма има два вида служители – програмисти и мениджъри. Отдел "личен състав" поддържа следната информация за всеки от програмистите:

- име;
- стаж (в месеци);
- дали знае C++;
- дали знае C#

и следната информация за всеки от мениджърите:

- име;
- стаж (в месеци);
- колко човека управлява.

Да се напише програма, която позволява на отдел "личен състав" да поддържа списък с всички програмисти и мениджъри във фирмата. Програмата да може да извършва следните операции:

- постъпване на нов служител;
- напускане на служител;
- извеждане на списък с данни за всички служители.

Да се въведат примерни данни за служители в софтуерна фирма и над тях да се изпълнят следните операции:

- изтриване на всички служители, които имат стаж по-малко от 3 месеца;
- записване на данните в текстов файл;
- прочитане на данните от текстов файл;

14.2 Сериализация и десериализация на контейнери. Шаблон "Фабрика"

14.4. Да се сериализира и десериализира масив от масиви от числа:

`vector<vector<int> >`

Да дефинират необходимите за целта оператори. Да се тества програмата!

14.5. Да се сериализира и десериализира масив от масиви от символни низове (`char*`):

`vector<vector<char*> >`

Да дефинират необходимите за целта оператори. Да се тества програмата!

14.6. Да се реализира абстрактен клас `NetworkDevice`, който дефинира следните (абстрактни, чисто втируални) операции:

- `bool attachTo(NetworkDevice* device)`: свързва устройството с друго устройство `device`.

  Реализаициите на метода в наследниците на `NetworkDevice` ще връщат `true`, ако свързването е възможно и `false`, ако свързването не е възможно. Правилата, по които се определя дали може или не може да се свърже устройството, зависят от конкретния наследник на `NetwrokDevice`.

  При дефиниране на метода в наследените класове осигурете, че създадената връзка е двупосочна. Т.е. счита се, че ако устройството `A` е свързано с устройството `B`, то и устройството `B` е свързано с устройството `A`. Не допускайте дадено устройство да може да се свърже повече от веднъж с едно и също устройство или пък да се свърже със себе си.

- `[попълнете правилния тип] getAttachedDevice(int i)`. Връща (*Упътване: указател към*) i-тото поред устройство, към което даденото устройство е свързано и NULL, ако индексът i не е валиден.

Класът `NetworkDevice` също да съдържа и уникален идентификатор от тип `int` за всяко устройство. Идентификаторът може да се задава чрез конструкторите на наследниците.

Да се реализират производните класове `EndDevice` и `Switch`.

- За `EndDevice` е характерно, че устройството може да е свързано най-много с още едно устройство. Т.е. във всеки момент от времето `EndDevice` или не е свързан с нищо, или е свързан с точно едно

устройство. Веднъж свързано, `EndDevice` не може да бъде свързано отново с друго устройство.

- За Switch е характерно, че устройството може да е свързано с максимум 8 други устройства. При достигане на броя на свързаните устройства до 8, устройството `Swith` не може да бъде свързвано с повече устройства.

Промяна на вече създадена връзка не е възможна и при двата вида устройства.

За така дефинираните класове да се решат следните задачи:

а) Да се реализира функкция `void printConnections(NetworkDevice* devices[],int n)`, която отпечатва на екрана информация за връзките на всяко устройство в масив от устройства. *Упътване: добавете нова виртуална функция printConncection в базовия клас. Връзките можете да печатате във формата* `<идентификатор 1> -- <идентификатор 2>`.

б) Да се създаде примерна програма, в която се инициализират няколко устройства, добавят се в масив, създават се връзки между тях и се използва функцията `printConnections` за отпечатване на връзките.

в) Да се реализира функция `bool connected([попълнете правилния тип] d1, [попълнете правилния тип] d2)`, която проверява дали има връзка (пряка или косвена) между устройствата d1 и d2. За улеснение приемете, че няма циклични връзки между устройствата. *Упътване: използвайте рекурсивна функция по подобие на функциите за търсене на път.*

г) *Допълнителна задача:*Решете горната задача и при случая на възможни циклични връзки.

д) *Допълнителна задача:*Напишете функция, която по масив от устройства търси дали има циклична връзка между някои две от тях.

е) *Допълнителна задача с повишена трудност:* Сериализирайте и десериализирайте масив от свързани устройства.

14.3 Наследяване и агрегация. Йерархичи дървета от обекти, съдържащи други обекти. Шаблон за дизайн "Посетител."

Проблемите около агрегацията на обекти от същата йерархия и получаващите се йерархичи дървета от обекти, съдържащи други обекти, са демонстрирани чрез йерархия от геометрични фигури, наследници на абстрактен базов клас `Shape`.

```
class Shape
{
public:
    Shape ();
    Shape (int x,int y,const char *title);
    Shape (const Shape&);
    virtual Shape* clone () = 0;
    void set_x (int);
    void set_y (int);
    void set_text (const char*);
    void operator = (const Shape& s);
    int get_x();
    int get_y();
    char* get_text();
    virtual ~Shape();
  protected:
    int x;
    int y;
    char *text;
};
```

Към йерархията е добавен клас `Group`, който освен че е наследник на `Shape`, съдържа и контейнер от наследници на `Shape`:

```
  class Group: public Shape
  {
    public:
      //...
      void addShape (Shape*);
      size_t get_nChildren () const;
      Shape* getChild (size_t i) const;
      //...
```

```
    private:
    std::vector<Shape*> children;
};
```

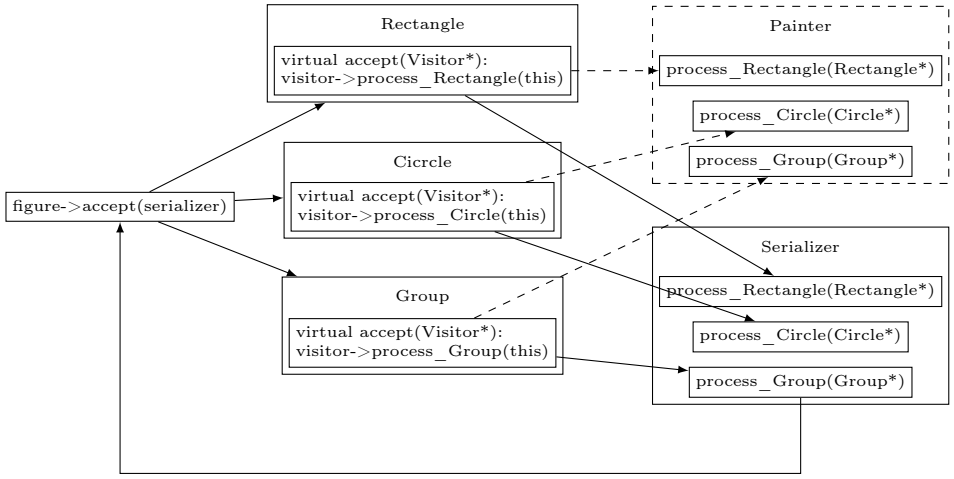
За изобразяване на фигурите и сериализацията им във файл се ползва шаблонът посетител (Visitor Pattern). Всеки конкретен посетител е наследник на клас:

```
class Visitor
{
public:
    virtual void visitReactangle (Rect*) = 0;
    virtual void visitCircle (Circle*) = 0;
    virtual void visitGroup (Group*) = 0;
};
```

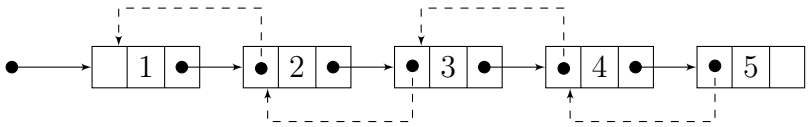
Йерархията от фигури реализира виртуалния метод
`void accept (Visitor*)`,
като всяка конкретна фигура изпълнява съответния на нея метод на посетителя. По този начин се реализира т.нар. “double dispatch”. Т.е. в следната ситуация:
`figure->accept(visitor)`,
кой точно метод ще се изпълни се определя динамично не само според типа на обекта, сочен от `figure`, но и според типа на конктретния посетител, сочен от `visitor`. Това е изобразено на фигура 8.

14.7. Да се реализира десериализация на фигури чрез шаблона “Фабрика”.

14.8. Да се дефинира посетител `SurfaceCalculator`, който обхожда йерархия от фигури и изчисля сумата на лицата на всички фигури в нея. Приемеме, че лицето на група от фигури е равно на сумата на лицата на фигурите в групата.



Фигура 8: Double Dispatch при йерархии от фигури и посетители



Фигура 9: Двусвързан списък

15 Линеен едносвързан и двусвързан списък

15.1 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.

15.1. 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`.

15.2. 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 $x \leq y$.

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

15.4. 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.

15.5. 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.

15.6. 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 $3, 2, 1$.

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

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

15.2 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);
```

15.9. За шаблона DLList да се дефинира метод `size_t find(const T& x)`, който намира поредния номер на елемента `x` в списъка (или връща размера на списъка, ако такъв елемент няма). Да се напише подходящ тест и да се изследва времевата сложност на метода емпирично.

15.10. 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.

15.3 SList (опростен вариант на 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:

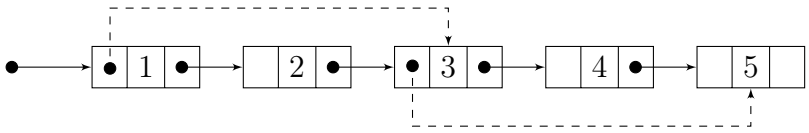
```
template <class T>
struct lnode
{
  T data;
  lnode<T> *next, *skip;
};
```

Както и при стандартния едносвързан списък, всеки от елментите на СП съдържа в указателя `next` адреса на непосредствения си съсед. Елементите на списъка са под дадена наредба. Някои от елементите могат да съдържат в указателя `skip` дреса на друг елемент, намиращ се по-напред в редицата от елементи (вж. Фигура 10). 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 подсписъка, съдържащ даден елемент, ще бъде със сложност $O(\sqrt{n})$. Търсенето в рамките на подсписъка с $\sqrt{n}$ елемента остава линейно. 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 `SList` с основните му канонични методи и метод за построяване на "бързите връзки". Реализирайте обиновен метод за вмъкване на елементи `insertSorted`, който вмъква елементи грижейки се само за непосредствените връзки (`next`), и метод `speedup`, който построява бързите връзки в списъка, след като в него са вмъкнати определен брой елементи.

15.11. Да се реализират конструктор за копиране и оператор на присвояване на `SList`, които репликират и бързите връзки.

15.12. Да се подобри методът за вмъкване на елемент `insertSorted`. Ако при вмъкване на нов елемент някой подсписък (обособен от дадена бърза



Фигура 10: Списък с прескачане

връзка) получи повече от $2\sqrt{(n)}$ елемента, да се добави нова бърза връзка така, че подсписъка да се раздели на два подсписъка.

15.13. Да се извърши времево измерване на проблема за търсене на елемент в подреден `SList`, както е обяснено в Секция 15.2, и да се изобрази чрез графика. Да се извършат емпирични сравнения на производителността на търсенете със и без оптимизацията.

15.4 Итератори за линейни СД

Следните задачи да се решат като упражнение за работа с итератори. Задачите изискват реализация на клас линеен двусвързан списък и `forward` итератор за него. Всяка функциия да се тества с подходящи тестове.

15.14. Да се разшири интераторът на двусвързан списък така, че да поддържа оператора за стъпка назад `--`.

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

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

15.17. 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).

15.18. Да се реализира константен итератор за двусвързани списъци и горните функции да се преработят така, че да могат да работят с константни итератори.

15.5 Функции от високо ниво и оператори над итератори

Нека "поредица" да наричаме последователни елементи на някакъв контейнер, зададени чрез итератори към началото и края на последователността. С помощта на следния шаблон:

```
template <class Iterator>
class Sequence{
    public:
        Sequence (const Iterator &b,
                  const Iterator &e):_begin(b), _end (e){}
        Iterator begin()
        {return _begin;}
        Iterator end()
        {return _end;}
    private:
        Iterator _begin;
        Iterator _end;
};
```

става възможно не само да оформим двойката начало-край в общ обект, но и да итерираме съответната поредица с цикъл `for`. Ако `begin` и `end` са съответните итератори към елементи от тип `E` и имаме обекта `Sequence<SomeIterator>(begin,end` то можем да направим обхождането:

```
for (E& x : seq){...}
```

На лекции разглеждаме деструктивен вариант на функцията от високо ниво `map`, извършваща еднотипна трансформация на елементите на поредица, като се абстрахира напълно от контейнера:

```
template <class Iterator, class Element>
void map(Sequence<Iterator> seq,
          Element (*f)(const Element &))
{
    for (Element& e: seq)
    {e = f (e);}
}
```

Може да разгледаме и фунцкията за агрегации `reduce`, позната от функционалното програмиране:

```
template <class Iterator, class Element>
Element reduce(Sequence<Iterator> seq,
                Element (*op)(const Element &, const Element &),
                Element null_val)
{
    Element accum = null_val;
    for (Element e : seq)
    { accum = op(accum, e); }
    return accum;
}
```

Възможно е също да реализираме операция с поредица, която изменя броя или реда на елементите ѝ, отново без знание за подлежащия контейнер. Такава е например операцията `filter`. Това налага да се конструира нова поредица, като по някакъв начин се измени поведението на итераторите на оригиналната поредица. Например, реализирания на лекции `FilterIterator` "изменя" даден друг итератор като "прескача" елементите, които не удовлетворяват даден предикат. По този начин можем да дефинираме:

```
template <class Iterator, class Element>
Sequence<FilterIterator<Iterator,Element>>
    filter(Sequence<Iterator> &seq,
           bool (*pred)(const Element &))
{
  FilterIterator<Iterator,Element> begin (seq.begin(),seq.end(),pred);
  FilterIterator<Iterator,Element> end (seq.end(),seq.end(),pred);

  return Sequence<FilterIterator<Iterator, Element>> (begin, end);
}
```

Използването на `Sequence` както за параметър, така и за резултат на тези функции, ни позволява да ги използваме за обхождане на новополучените подредици. Също така, можем да композираме функциите:

```
for (E x : filter (seq, even)) {...обработка само на четните елементи на seq...}
reduce(filter(seq,even),plus,0);//сума на четните елементи на seq
```

Ако по подобие на `FilterIterator` реализираме `MapIterator`, можем да направим функционален вариант на `map`, `mapf`. Това ще ни позволи на намерим сумата на четните елементи на поредица, след като ги увеличим с единица, по следния начин:

```
reduce(mapf(filter(seq,even), inc), plus, 0)
```

15.19. По аналогия на дадените примери да се дефинира функция `append(a,b)`, която създава нова поредица чрез слепване на поредиците `a` и `b`.

15.20. По аналогия на дадените примери да се дефинира функция `combine(a,b,f)`, която създава нова поредица от стойностите на двуместната функция `f`, приложена върху съответните елементи на `a` и `b`. Функцията да се демомстрира като се съберат съответните елементи на две редици.

15.21. По аналогия на дадените примери да се дефинира функция `merge(a,b)`. Ако $a = a_1, ..., a_k$, $b = b_1, ..., b_l$, то $merge(a, b) = a_1, b_1, a_2, b_2, ...$

15.22. По аналогия на дадените примери да се дефинира функция `zip(a,b)`, която създава нова поредица от двойките (`std::pair`) от стойностите на съответните елементи на `a` и `b`.

16 Приложения на структурата от данни стек

16.1 Общи задачи за стекове

16.1. Нека е даден масив с n елемента. За всеки от елементите на масива да се изведе следващия в масива по-голям елемент. Т.е. за всеки елемент $a[i]$ да се отпечата $a[j]$ такъв, че $n > j > i$ и $\forall k \in (i, j) : a[k] \leq a[i]$. Ако такъв елемент няма, да се изведе числото -1. Пример, за масива `{4, 5, 2, 25}`, да се изведат двойките `(4,5), (5,25), (2,25), (25,-1)`. Алгоритъмът да работи с линейна сложност спрямо n. *Упътване: Задачата е известна под името "Next Greater Element(NGE)".*

16.2 Изрази и стекове

16.2. Даден е израз, който може да съдържа отварящи и затварящи скоби. Да се напише функция, която проверява дали скобите на израза са правилно балансирани. Например, изразът `(x+(y+(1+2)))` считаме за правилно балансиран, но не и израза `(x+y)*3)+(x+(1+2)`.

16.3. Да се реши горната задача при положение, че изразът може да има едновременно кръгли, фигурни и квадратни скоби.

По време лекции разглеждаме реализация на *Shunting Yard* алгоритъма за пребразуване на инфиксен аритметичен израз в постфиксен вид (Обратен полски запис). Реализацията работи само с двуместни оператори и скоби, като отчита приоритет на операциите.

16.4. При изчисляването на стойността на постфиксен израз, да се разгледа случая на ляво-асоциативни оператори (като деление и изваждане).

16.5. При изчисляването на стойността на постфиксен израз, да се позволи използването на едноместни функции без влагане. Да се поддържа ограничен набор от предефинирани функции (напр. $sin(x)$ и $\sqrt{x}$.). Функциите да са само едноместни и да не се позволява влагане на извиквания.

16.6. Да се преработи алгоритъма за изчисляването на стойността на постфиксен израз така, че да се генерира прав полски запис (префиксна нотация).

16.3 Цикъл със стек вместо рекурсия

Упътване:Решете задачите с рекурсия и след това преобразувайте решението в решение със стек.

16.7. (*)Да се дефинира функция за намиране на стойността на полинома на Ермит $Hn(x)$ (x е реална променлива, а n неотрицателна цяла променлива), дефиниран по следния начин:

$H_0(x) = 1$

$H_1(x) = 2x$

$H_n(x) = 2xH_{n-1}(x) + 2(n-1)H_{n-2}(x), n > 1,$

за дадени n и x с използване на стек.

16.8. Нека е дадена абстрактна шахматна дъска с размери $n \times n$, $4 \leq n \leq 8$ и число k, $0 \leq k \leq n$. Казваме, че разположени на дъската k коня образуват "валидна конфигурация", ако никоя фигура не е поставена на поле, което се "бие" от друга фигура според съответните шахматни правила.

Да се дефинира клас `KnightConfig`, представящ "конфигуратор" на шахматни коне. Конструкторът на класа инициализира конфигуратора с числата n и k. Класът позволява "обхождането" една по една на всички валидни конфигурации за дадените параметри, по подобие на `forward` итератор на структура от данни. Класът да притежава следните методи:

- `void KnightConfig::printCurrentConfig()`: Отпечатва текущо намерената конфигурация. Пример за отпечатана конфигурация с $n = 5, k = 2$:

```
_ _ _ _ _
_ _ H _ _
_ _ _ _ _
_ _ _ _ H
_ _ _ _ _
```

- `void KnightConfig::findNextConfig()`: Намира следваща конфигурация.
- `bool KnightConfig::noMoreConfigs()`: Показва дали всички възможни конфигурации са вече изчерпани.

16.9. Да се реши задачата за Ханойските кули с използване на стек.

Да се дефинира клас `HanoyPlayer` със следните методи:

- Конструктор с параметър, указващ броя дискове върху лявото колче за началното състояние на играта.
- Метод `bool final()`, който показва дали играта е достигнала финанлно състояние (т.е. всички дискове са на дясното колче).

- Метод `makeMove()`, който извършва един ход от играта.
- Метод `printBoard()`, който отпечатва текущото състояние на игровата дъска, например по следния начин:

```
2
3     1
5  *  4
```

  На примера е изобразено състояние на играта, при което на лявото колче има три диска - с размери 5, 3 и 2, на средното колче няма дискове, а на дясното има два диска - с размери 4 и 1.

17 Двоични дървета

17.1 Прости обхождания

Възел на двоично дърво представяме със следния шаблон на структура:

```
template <class T>
struct BinTreeNode
{
  T data;
  BinTreeNode *left, *right;
  //... помощни конструктори
};
```

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

```
template <class T>
class BinTree
{
  //...
  private:
  BinTreeNode<T> *root;
};
```

17.1. Да се деифнира метод `count` на клас `BinTree`, който намира броя на елементите на дървото.

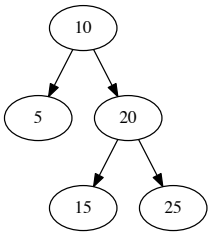
17.2. Да се деифнира метод `countEvens` на клас `BinTree`, който намира броя на елементите на дърво от числа, които са четни.

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

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

17.4. Да се дефинира метод `bool BinTree<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 11 е 3.*

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



Фигура 11: Двоично наредено дърво

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

17.7. 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 11, съответният елемент на празния низ е 10, на низа "RL" е 15, а "RLR" няма съответен елемент.

Да се дефинира метод `T& BinTree<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.

17.2 Отпечатване и сериализация

В следващите задачи се очаква дефинирането на външни за класа на дървото функции, вместо методи на класа, както в уводните задачи. За целта се използва клас `Position`, заместващ указател към възел в дадено дърво, по подобие на итератор, и с чиято помощ може да се извърши рекурсивно

обхождане на дървото.

```
template <class T>
class Position
{
public:
    Position<T> left () const;
    Position<T> right () const;
    T operator * () const;
    bool empty () const;
    //...
};
```

В условията по-долу с `[подходящ тип]` е указано, че част от задачата е да се изберат правилните типове на параметрите на фунцкиите така, че рекурсия да може да се осъществи.

17.8. Да се дефинира функция `prettyPrint ([подходящ тип]t)`, отпечатваща двоично дърво на стандарния изход 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 11 би изглеждало по следния начин (да се отпечатат и номерата на редовете на стандартния изход):

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

17.9. Да се дефинират функции `std::string serializeTree ([подходящ тип]t)` и `BinTree<int> parseTree (std::string)` за сериализация и де-сериализация на двоично дърво, съдържащо числа, като се използва "`Scheme` формат".

Може ли да се сериализира дърво, съдържащо други дървета?

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 11 се представя по следния начин:

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

17.3 Обхождания

17.10. Да се реализира функция `std::vector<T> listLeaves ([подходящ тип]t)` намираща списък със стойностите на листата на дървото.

17.11. Да се дефинира функция `std::string findTrace ([подходящ тип]t, 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 низа "_".

Пример: За дървото от Фигура 11, следата на елемента със стойност 25 е "RR".

17.12. Да се дефинира функция `T getAt([подходящ тип]t, size_t i)`, която намира i-тият пореден елемент на дървото при обхождане корен-ляво-дясно.

Пример: За дървото от Фигура 11, елементът с пореден номер 0 е 10, с номер 1 е 5, с номер 2 е 20 и т.н.

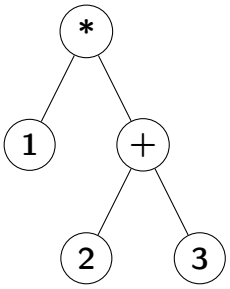
17.4 Представяне на аритметичен израз чрез дърво

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

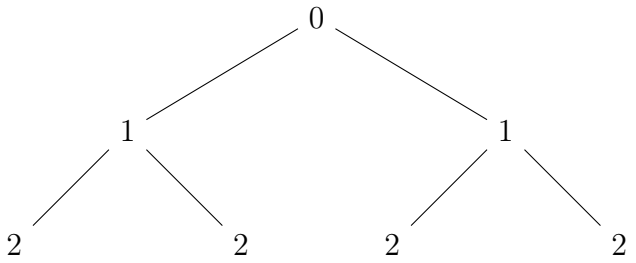
```
<expression> ::= <digit> | (<expression><operator><expresson>)
<digit> ::= 1..9
<operator> ::= + | - | * | /
```

Да се реализира функция на клас `BinTree<char> parseExpression (std::string s)`, която 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`.



Фигура 12: Дърво на израза `(1*(2+3))`.



Фигура 13: Идеално балансирано двоично дърво с височина 3

- 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>`.

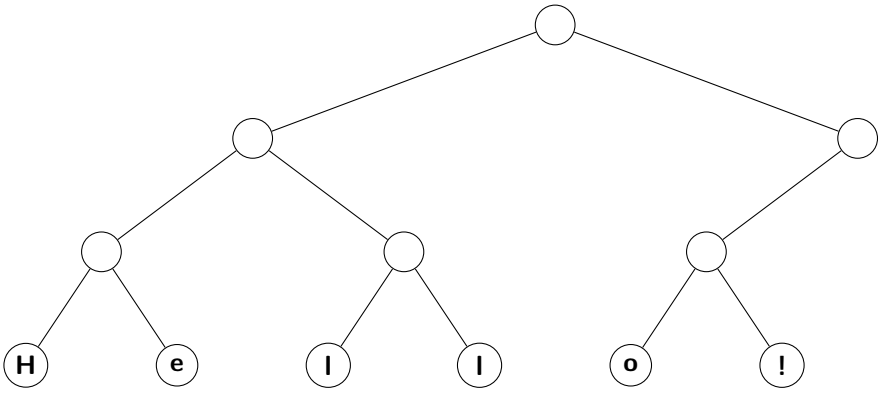
Дървото на Фигура 16 съответства на изарза `(1*(2+3))`.

17.14. Да се реализира функция `double calculateExpresisonTree ([подходящ тип]t)`, която намира стойността на израз по дадено дърво, построено от решението на предишната задача.

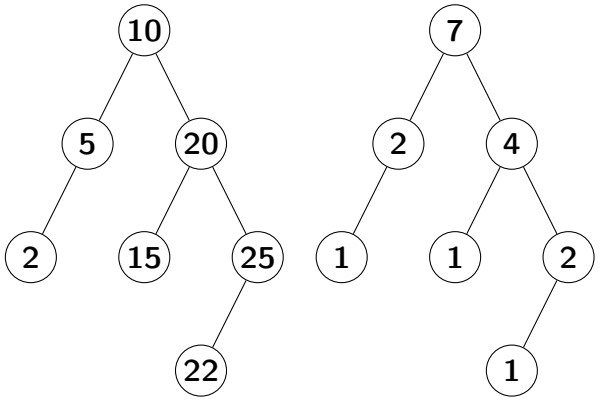
17.5 Построяване и модификации на дърво

17.15. 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 13).

17.16. 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 14



Фигура 14: Дърво, в чиито листа е разположен низът “Hello!”



Фигура 15: Примерно дърво и същото дърво, стойностите на чиито възли са заместени с размера на съответното им поддърво

17.17. 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 15.

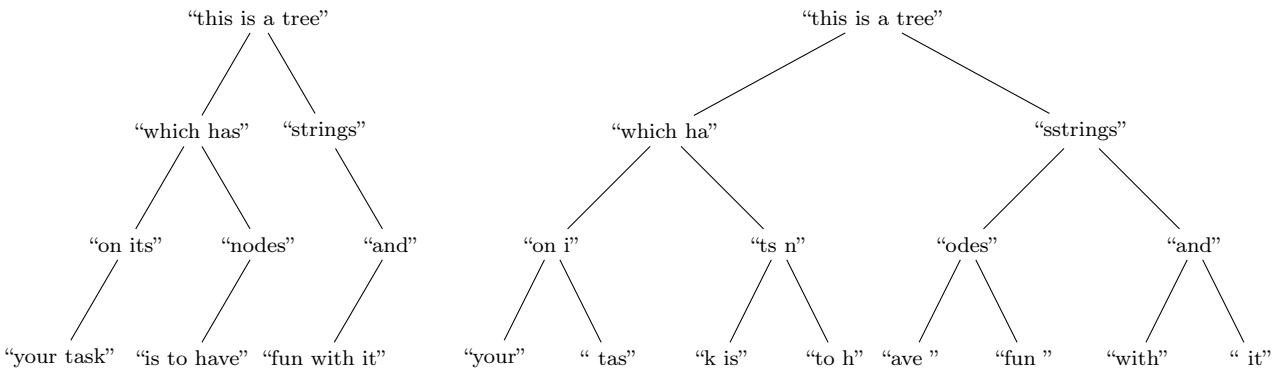
При операцията всеки от възлите да бъде посетен най-много веднъж!

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

а) Резултатното дърво има същия брой нива като изходното.

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



Фигура 16: Примерно дърво от низове преди и след балансирането

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.

Ако $n_k < 2^k$, да се използва низ, получен от минималния брой копия на s_k, така, че да се получи низ с дължина $\geq 2^k$.

На Фигура 16 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.*

18 Префиксно дърво (Trie)

Префиксното дърво (Trie) може да се използва за представяне на множество от думи (вж. Фигура 17). Възлите в дървото може да имат произволен брой наследници, като всяка дъга е анотирана със символ. Всеки възел е анотиран с флаг, който разделя възлите на финални и нефинални (на схемата финалните възли са отбелязани с "X"). Символите по дъгите на всеки път от корена на дървото до финален възел задават една дума от множеството. Дадена дума е елемент на множеството т.с.т.к. съществува такъв път. Обърнете внимание, че даден възел може да е както финален, така и част от по-дълга дума. Това се случва, когато в можеството се съдържа някоя дума заедно с някой нейн префикс. Използваме просто представяне на възел в дървото, като сме ограничили множеството на допустимите символи до малки латински букви:

```
struct TrieNode
{
    bool eow;
    TrieNode *children[26];
    TrieNode ();
};
```

На лекции дефинираме функция за добавяне на дума към множеството, представено с дадено дъро и за извеждане на дърво в `dot` формат:

```
void trieInsert (TrieNode *&root, const char *s);
void dottyPrint (std::ostream& out, TrieNode *root)
```

18.1. Да се напише функция за намиране на височина на Trie.

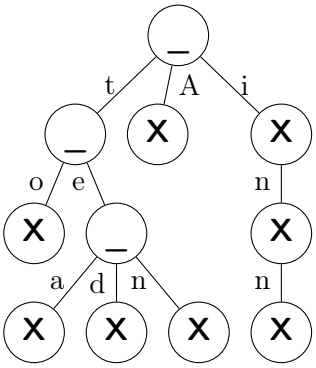
18.2. Да се напише функция за намиране на дължината на най-дългата дума, записана в Trie.

18.3. Да се напише функция за намиране на броя на записаните думи в Ttie.

18.4. Да се напише функция за намиране на броя на буквите на латинската азбука, които *не участват* в никоя дума в дадено Trie.

18.5. Да се обвие реализацията на Trie в клас `StringSet`. За класа да се реализират:

- Канонични методи
- Оператор == за сравнение на две множества



Фигура 17: Trie, съдържащо думите “to”, “tea”, “ted”, “ten”, “A”, “i”, “in” и “inn”.

- Оператори `+` и `+=` за добавяне на дума към множеството
- Булев оператор `[]`, прверяващ дали дадена дума се съдържа в множеството
- Оператори `-` и `-=` за премахване на дума от множеството
- Оператори `<` и `<=`, реализиращи проверка за подмножество

18.6. За клас `StringSet` да се реализира итератор, повлоляващ обхождане на думите в множеството в реда на лексикографската наредба.

19 Изображения (Maps)

19.1 HashMap

Следнитте задачи са базирани на проста реализация на хеш таблица с отворено адресиране:

```
template <class KeyType, class ValueType>
class HashMap;
```

Редовете в хеш таблицата са елементи от структурата:

```
template <class KeyType, class ValueType>
struct TableElement
{
  KeyType key;
  ValueType value;
  TableElement<KeyType,ValueType> *next;
}
```

Самата хеш таблица е предтавена с член данната на класа `HashMap` `TableElement<KeyType, ValueType> **table`.

Реализиран е и итератор, позволяващ обхобдане на ключовете, записани в хеш таблицата.

19.1. Да се дефинира метод `HashMap::efficiency()`, който изчислява ефективността на хеш таблицата като отношението $\frac{all-coliding}{all}$, където *coliding* е броят на ключовете, записани при колизия, а *all* е броят на всички записани ключове.

19.2. Да се дефинира оператор `<<` за клас `HashMap`, който отпечатва в поток всички двойки ключ-стойност в Хеш таблицата.

19.3. Да се напише програма, която въвежда от клавиатурата две текста с произволна големина t_1 и t_2. Програмата да извежда броя на всички срещания на думи в t_2, които се срещат и в t_1.

Пример: за следните текстове

In computing, a hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.

и

Ideally, the hash function will assign each key to a unique bucket, but this situation is rarely achievable in practice (usually some keys will hash to the same bucket)

Този брой е 10, съставен от думите *the* (2 срещания във втория текст), *a* (1 срещане), *hash* (2), *function* (1), *to* (2), *is* (1), *keys* (1).

19.4. Да се напише програма, която въвежда от клавиатурата две текста с произволна големина t_1 и t_2. Програмата да извежда броя на уникалните думи в t_2, които се срещат и в t_1.

Пример: за двата текста от предишната задача, този брой е 7, съставен от думите *the*, *a*, *hash*, *function*, *to*, *is*, *keys*.

19.5. Да се напише програма, която прочита от входа даден текст с произволна големина и намира такава дума с дължина повече от 3 букви, която се среща най-често в текста. Пример: за текста

In computing, a hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.

Най-често срещаната дума е *hash*.

19.6. От клавиатурата да се въведе цялото положително число n, следвано от $2 \times n$ цели положителни числа $a_1, b_1, a_2, b_2, ..., a_n, b_n$. Програмата да печата на екрана `"Yes"`, ако изображението, дефинирано като $h(a_i) = b_i, i = 1, ..., n$ е добре дефинирана функция. Т.е. програмата да проверява дали има два различни индекса i и j, за които е изпълнено $a_i = a_j$, но $b_i \neq b_j$.

19.7. Да се дефинира метод

```
void map ([подходящ тип] f)
```

на хеш-таблицата, който прилага функцията `f` над всички стойности в хеш-таблицата.

19.8. Да се добави възможност за итериране само на тези ключове от хеш таблицата, чиито стойности удовлетворяват някакъв предикат, зададен чрез (шаблон на) указател към функция. *Упътване: разширете итератора така, че да може да се инициализира с предикат. Предикатът може да се задава, например, чрез метода begin. Последният трябва да има стойност по подразбиране на параметъра си, за да може да се ползва цикъл* `for (<key> : <hash table>)`.

Литература

[1] Магдалина Тодорова, Петър Армянов, Дафина Петкова, Калин Георгиев, “Сборник от задачи по програмиране на C++. Първа част. Увод в програмирането”

[2] А. Дичев, И. Сосков, “Теория на програмите”, Издателство на СУ, София, 1998

[3] Wikihow, How to Draw Pusheen the Cat, https://www.wikihow.com/Draw-Pusheen-the-Cat

[4] R. W. Floyd. “Assigning meanings to programs.” Proceedings of the American Mathematical Society, Symposia on Applied Mathematics. Vol. 19, pp. 19–31. 1967.

[5] Александра Соскова, Стела Николова, “Теория на програмите в задачи”, Софтех, 2003

[6] David Matuszek, “Backtracking”, https://www.cis.upenn.edu/ matuszek/cit594-2012/Pages/backtracking.html.

[7] Магдалина Тодорова, Петър Армянов, Дафина Петкова, Калин Николов, “Сборник от задачи по програмиране на C++. Втора част. Обектно ориентирано програмиране”