Теми за Държавен Изпит

4 август 2024 г.

Процедурно програмиране – основни конструкции.

1.Основни принципи

Основните принципи на процедурното програмиране са принципа за модулност и принципа за абстракция на данните. Съгласно принципа за модулност, програмата се разделя на подходящи взаимосвързани части, всяка от които се реализира чрез определени средства. Съгласно принципа за абстракция на данните, методите за използване на данните се отделят от методите за тяхното коректно представяне. Програмите се конструират така, че да работят с абстрактни данни - данни с неуточнено представяне.

2.Управление на изчислителния процес

Условни оператори Чрез тези оператори се реализират разклоняващи се изчислителни процеси. Оператор, който дава възможност да се изпълни (или не) един или друг оператор в зависимост от някакво условие, се нарича условен. Примери за условни оператори са: $if, if/else$ и $switch$.

If оператор
Синтаксис
if (<условие>) <оператор>, където if е запазена дума, <условие> е булев израз и <оператор> е произволен оператор.
Семантика
Пресмята се стойността на булевия израз, представящ условието. Ако резултатът е $true$, изпълнява се <оператор>, в противен случай не се изпълнява.

If/else оператор
Синтаксис
if (<условие>) <оператор1> $else$ <оператор2>, където if и $else$ са запазени думи, <условие> е булев израз и <оператор1> и <оператор2> са произволни оператори.
Семантика
Пресмята се стойността на булевия израз, представящ условието. Ако резултатът е $true$, изпълнява се <оператор1>, в противен случай се изпълнява

<оператор2>.

Switch оператор
Синтаксис
switch (<израз>)
case < израз$_1$ > : < редица_от_оператори$_1$ >
...
case < израз$_{n-1}$ > : <редица_от_оператори$_{n-1}$ >
[default : < редица_от_оператори$_n$ >], където *switch*, *case* и $default$ са запазени думи,
Семантика
Намира се стойността на *switch* израза, получената константа се сравнява последователно със стойностите на етикетите израз$_1$израз$_2$ и при съвпадение, се изпълняват операторите на съответния вариант и операторите на всички варианти, разположени след него, до срещане на оператор *break*. В противен случай, ако участва $default$-вариант, се изпълнява редицата от оператори, която му съответства и в случай, че не участва такъв - не следват никакви действия от оператора *switch*.

Оператори за цикъл
Операторите за цикъл се използват за реализиране на циклични изчислителни процеси. Изчислителен процес, при който оператор или група от оператори се изпълняват многократно за различни стойности на техни параметри, се нарича цикличен. Съществуват два вида изчислителни процеси - индуктивни, при които броят на повторенията е известен предварително и итеративни, при които броят на повторенията не е известен предварително.

For оператор
Синтаксис
for (<инициализация>; <условие>; <корекция>) <оператор>, където for е запазена дума, <условие> е булев израз, <инициализация> е или точно една дефиниция с инициализация на една или повече променливи, или няколко оператора за присвояване или въвеждане, отделени със , и не завършващи с ;, <корекция> е един или няколко оператора, незавършващи с ; (в случай че са няколко се отделят със ,) и <оператор> е точно един произволен оператор, който се нарича тяло на цикъла.
Семантика
Изпълнението започва с изпълнение на частта <инициализация>. След това се намира стойността на <условие>. Ако в резултат се е получило $false$, изпълнението на оператора for завършва, без тялото да се е изпълнило нито веднъж. В противен случай последователно

се повтарят следните действия: изпълнение на тялото на цикъла, изпълнение на операторите от частта <корекция>, пресмятане на стойността на <условие> докато стойността на <условие> е *true*.

While оператор
Синтаксис
while (<условие>) <оператор>, където *while* е запазена дума, <условие> е булев израз и <оператор> е произволен оператор, който описва действията, които се повтарят и се нарича тяло на цикъла.
Семантика
Пресмята се стойността на <условие>. Ако тя е $false$, изпълнението на оператора *while* завършва без да се е изпълнило тялото му нито веднъж. В противен случай, изпълнението на <оператор> и пресмятане на стойността на <условие> се повтарят докато <условие> е *true*. В първия момент, когато <условие> стане $false$, изпълнението на *while* завършва.

Do/while оператор
Синтаксис
do <оператор>
while (<условие>), където *do* и *while* е запазени думи, <условие> е булев израз и <оператор> е произволен оператор, който описва действията, които се повтарят и се нарича тяло на цикъла.
Семантика
Изпълнява се тялото на цикъла, след което се пресмята стойността на <условие>. Ако то е $false$, изпълнението на оператора *do/while* завършва. В противен случай, се повтарят действията: изпълнение на тялото и пресмятане на стойността на <условие>, докато стойността на <условие> е *true*. Веднага, след като стойността му стане $false$, изпълнението на оператора завършва.

3.Променливи

Променливата е място за съхранение на данни, което може да съдържа различни стойности по време на изпълнение на програмата. Означава се чрез редица от букви, цифри и долна черта, започваща с буква или долна черта. Променливите имат три характеристики: тип, име и стойност. Преди да бъдат използвани, те трябва да бъдат дефинирани. C++ е строго типизиран език за програмиране. Всяка променлива има тип, който явно се указва при дефинирането й.

Дефиниране на променливи

Синтаксис: <име_на_тип> <променлива> [= <израз>] , <променлива> [= <израз>] ; където <име_на_тип> е дума, означаваща име на тип като int, double и др., а <израз> е правило за получаване на стойност - цяла, реална, знакова и др. тип, съвместим с <име_на_тип>.

Семантика: Дефиницията свързва променливата с множеството от допустимите стойности на типа, от който е променливата или с конкретна стойност от това множество. За целта се отделя определено количество оперативна

памет (толкова, колкото да се запише най-голямата константа от множеството от допустимите стойности на съответния тип) и се именува с името на променливата. Тази памет е с неопределено съдържание или съдържа стойността на указания израз, ако е направена инициализация. Не се допуска една и съща променлива да има няколко дефиниции в рамките на една и съща функция.

Видове променливи

- Локални - променливи, които имат локална област на действие. Те са дефинирани във функции и не са достъпни за кода в другите функции на модула. Областта им се определя според общо правило – започва от мястото на дефинирането и завършва в края на оператора (блока), в който идентификаторът е дефиниран. Формалните параметри на функциите също имат локална видимост. Областта им е тялото на функцията. ??????
- Глобални - променливи, които се дефинират извън функциите и които са "видими"за всички функции, дефинирани след тях. Декларират се както се дефинират другите (локалните) променливи. Използването на много глобални променливи е лош стил за програмиране и не се препоръчва. Всяка глобална променлива трябва да е съпроводена с коментар, обясняващ предназначението ѝ.

Инициализация на променлива ?????

Оператор за присвояване

Синтаксис: <променлива> = <израз>;
където <променлива> и <израз> са от един и същ тип.

Семантика: Пресмята стойността на <израз> и я записва в паметта, именувана с променливата от лявата страна на знака за присвояване =.

Пример:

```
a = 1;
b = 2;
c = 2 * (a + b);
```

Чрез операторите за присвояване, променливите a, b и c получават текущи стойности.

4.Функции, процедури и параметри
Функциите са основни структурни единици, които изграждат програмите на езика. Сред функциите задължително трябва да има точно една с име *main*, наречена главна функция. Тя е първата функция, която се изпълнява при стартиране на програмата. Използването на функции има следните

предимства:

- програмите стават ясни и лесни за тестване и модифициране
- избягва се многократно повторение на едни и същи програмни фрагменти, които се дефинират еднократно и могат да се изпълняват произволен брой пъти
- постига се икономия на памет, тъй като кодът на функцията се съхранява само на едно място в паметта, независимо от броя на нейните изпълнения

Разпределение на оперативната памет за изпълнима програма

Разпределението на оперативната памет най-общо се състои от: програмен код, област на статичните данни, област на динамичните данни и програмен стек. В програмният код е записан изпълнимият код на всички функции, изграждащи потребителската програма. В областта на статичните данни са записани глобалните обекти на програмата. За реализиране на динамични структури от данни се използват средства за динамично разпределение на паметта. Чрез тях се заделя и освобождава памет в процеса на изпълнение на програмата, която е областта на динамичните данни. Програмният стек е вид памет, която съхранява данните на функциите на програмата. Стекът е динамична структура, организирана по правилото *LIFO*, представляваща редица от елементи с пряк достъп до елементите от единия си край, наречен връх. Достъпът се реализира чрез указател. Операцията включване се осъществява само пред елемента от върха, а операцията изключване - само за елементита от върха. Елементите на програмния стек са "блокове"от памет, съхраняващи данни, дефинирани в някоя функция, които се наричат стекови рамки.

Пример за функция:

```
int gcd(int a, int b) {
    while (a != b) {
        (a > b) ? a = a - b : b = b - a;
    }

    return a;
}
```

Двуаргументната функция с име *gcd*, е от тип *int*, а параметрите *a* и *b* се наричат формални параметри на функцията. Тялото и е блок, реализиращ алгоритъма на Евклид за намиране на НОД на две числа, който завършва с оператора *return*. Чрез него се прекратява изпълнението на функцията като стойността на израза след *return* се връща като стойност на *gcd*.

5.Символни низове

Логическо описание
Редица от краен брой символи, заградени в кавички, се нарича символен низ или само низ. Броят на символите в редицата се нарича дължина на низ. Низ, който се съдържа в друг низ, се нарича негов подниз. Конкатенация на два низа е низ, получен като в края на първия низ се запише вторият. Два символни низа се сравняват лексикографски - сравнява се всеки символ от първия низ със символа на другия низ на същата позиция, докато не се намерят два различни символа или до края на поне един от символните низове. Ако кодът на символ от първия низ е по-малък от кода на съответния символ от втория низ се приема, че първият низ е по-малък от втория.

Физическо представяне
В оперативната памет, низовете се представят последователно. Съществуват 2 начина за разглеждане на низовете в езика $C++$:

- масиви от символи
- указатели към тип *char*

Ще разгледаме символните низове като масиви от символи.

```
char str[100]; // дефинира променливата str като масив от 100
// символа
char str2[5] = {'a', 'b', 'c'} // дефинира масив от символи
// str2 и го инициализира, тук тъй като при инициализацията са
// указани по-малко от 5 символа, останалите се допълват с '0'
```

Низовете разглеждаме като редица от символи, завършващи с нулевия символ \0, наречен още терминираща нула. Предимството на това е, че не е необходимо за всеки низ да се пази в променлива дължината му, тъй като знакът за края на низ позволява да се определи краят му. ????

Процедурно програмиране – указатели, масиви и рекурсия.

1.Указатели и указателна аритметика

Тип указател Променливите са свързани със стойности - неопределени или константи от същия тип, които се наричат *rvalue*. Мястото в паметта, в което е записана *rvalue* се нарича адрес на променливата или още *lvalue*. Ако имаме дефиницията $inti = 1024;$, тогава стойността на променливата $i(rvalue)$ е 1024 и тя именува място от паметта ($lvalue$) с размери 4 байта, като *lvalue* е адреса на първия байт. Намирането на адреса на дефинирана променлива става чрез ундарния префиксен дясноасоциативен оператор &:

Синтаксис: &<променлива>, където <променлива> е дефинирана променлива

Семантика: Намира адреса на <променлива>.

2.Масиви - основни операции и алгоритми

Логическо описание
Масивът е крайна редица от фиксиран брой елементи от един и същ тип. Към всеки елемент от редицата е възможен пряк достъп, който се осъществява чрез индекс. Операциите включване и изключване на елемент в/от масива са недопустими (статична структура от данни).

Физическо представяне
Елементите на масива се записват последователно в паметта на компютъра, като за всеки елемент на редицата се отделя определено количество памет.

3.Рекурсия

Функция, която се обръща пряко или косвено към себе си, се нарича рекурсивна. Програма, съдържаща рекурсивна функция е рекурсивна. Пример:

```
int fact(int n)  {
    if (n <= 1) return 1;
    return n * fact(n - 1);
}
```

Рекурсивната функция $fact$, приложена към естествено число, връща факториела на това число. Стойността на функцията се определя посредством обръщение към самата функция в оператора $return\ \ n * fact(n-1)$. Нека проследим хода на рекурсивната функция при примерно извикване:

```
cout << n << "!=" << fact(n);
```

Обектно-ориентирано програмиране

ООП. Основни принципи. Класове и обекти. Наследяване и капсулация.

1.Абстракция със структури от данни. Класове и обекти. Видове конструктори. Управление на динамичната памет и ресурсите (RAII). Методи

Абстракция с данни е подход, при който методите за използване на данните са разделени от методите за тяхното представяне. Програмите се конструират така, че да работят с "абстрактни данни данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени конструктори, селектори и предикати, които реализират "абстрактните данни"по конкретен начин.

Класовете са основни програмни единици в обектно-ориентираните езици, представляващи съвкупност от променливи и функции, които са обвързани в логическа структура и работят заедно. Те служат като модели за представяне на реални обекти, описвайки атрибути (свойства) и методи (поведение) на обектите.
Декларацията на клас се състои от заглавие и тяло. Заглавието започва с ключовата дума *class*, следвана от идентификатор, задаващ името на класа. Тялото съдържа декларации на компонентите на класа (член-данни и член-функции). Декларациите са заградени във фигурни скоби, следвани от знака ; или списък от обекти следван от ;.

Пример:

```
class Date {
    //Тяло на класа
};
```

Обектите са екземпляри на даден клас. Връзката между клас и обект е подобна на връзката между тип данни и променлива величина, но за разлика от обикновените променливи, обектите се състоят от множество компоненти (член-данни и член-функции). Ако в клас явно не е дефиниран конструктор, **дефиницията на обект** от този клас не трябва да съдържа инициализация с явно обръщение към конструктор, т.е. трябва да има вида: <име_на_клас><обект>;.

Пример:

```
Date d;
```

В този случай, автоматично в класа се създава подразбиращ се конструктор и инициализацията на обекта се осъществява чрез него. Този конструктор изпълнява редица действия, като заделяне на памет за обектите, инициализиране на системни променливи и т.н.

Създаването на обекти е свързано със заделяне на памет, запомняне на текущо състояние, задаване на начални стойности и други дейности, които се наричат инициализация на обектите. Тя се изпълнява от специален вид член-функции на класовете - **конструкторите**. Конструкторът е член-функция, която притежава повечето характеристики на другите член-функции и има редица особености като:

- името на конструктора съвпада с името на класа
- типът на резултата не се указва явно
- изпълнява се автоматично при създаване на обекти
- не може да се извиква явно

В клас може явно да не е дефиниран конструктор, но може да са дефинирани и няколко конструктора. Всички те имат едно и също име - името на

класа, но трябва да се различават по броя и/или типа на параметрите си. Такива конструктори се наричат предефинирани. При създаването на обект от клас се изпълнява само един от тях. **Основните видове конструктори** са:

- Конструктор по подразбиране
- Конструктор с параметри
- Конструктор за копиране

Ако в даден клас не е дефиниран конструктор, автоматично се създава т.н. конструктор по подразбиране. Той реализира множество от действия като заделяне на памет за член-данните на обект, инициализиране на системни променливи и др. Този конструктор може да бъде предефиниран, като за целта е необходимо в класа да бъде дефиниран конструктор без параметри.

Пример:

```
date::date() {
    month = 9;
    day = 13;
    year = 2024;
}
```

Инициализацията на новосъздаден обект на даден клас може да зависи от друг обект на същия клас. За да се укаже такава зависимост, се използват операторът за присвояване или кръгли скоби. Конструкторът за присвояване е специален конструктор, който поддържа формален параметър от тип *const*<име_на_клас>&. Ако в един клас явно не е дефиниран конструктор за присвояване, компилаторът автоматично създава такъв, в момента когато новосъздаден обект се инициализира с обект, намиращ се от дясната страна на оператора за присвояване или кръгли скоби. Този конструктор за присвояване се нарича конструктор за копиране. Пример:

```
date::date(const date& other) {
    month = other.month;
    day = other.day;
    year = other.year;
}
```

2.Наследяване. Производни и вложени класове. Капсулация. Статични полета и методи.

Наследяването е начин за създаване на нови класове чрез използване на компоненти и поведение на вече съществуващи класове. Класовете, които са вече дефинирани се наричат базови, а новосъздадените - **производни**. Производният клас може да наследи компонентите на един или няколко базови класа. В първия случай наследяването се нарича единично, а във втория - множествено. Ако множество от класове имат общи член-данни и

методи, тези общи части могат да се обособят като основни класове, а всяка от останалите части да се дефинира като производен клас на съответните основни класове. По този начин се спестява памет, тъй като се избягва многократното описание на едни и същи програмни фрагменти. При конструирането на производни класове е достатъчно да се разполага само с обектните модули на основните класове, а не с техния програмен код. Това позволява да бъдат създавани библиотеки от класове, които да бъдат използвани при създаването на производни класове. Тези предимства, а също и възможността за реализиране на полиморфизъм, мотивират въвеждането на наследяване и на производни класове. Подобно на обикновените, производните класове се дефинират, като се декларира класът и се дефинират неговите методи.

Множеството от компонентите на производен клас се състои от компонентите на неговите базови класове и компонентите, декларирани в самия производен клас. Механизъмът, чрез който производния клас получава компонентите на базовия, се нарича наследяване. Процесът на наследяване се изразява в следното: ...???? Защо пак обясняваме какво е наследяване

Пример:

```
#include <iostream>
#include <cstring>
using namespace std;

// Базов клас
class Person {

    public:
        void readPerson(const char*, const char*);
        void printPerson() const;

    private:
        char* name;
        char* ucn;
};

void Person::readPerson(const char* na, const char* uc) {
    name = new char[strlen(na) + 1];
    strcpy(name, na);

    ucn = new char[strlen(uc) + 1];
    strcpy(ucn, uc);
}

```

```
25 void Person::printPerson() const {
26     cout << "Name: " << name << endl;
27     cout << "UCN: " << ucn << endl;
28 }
29
30 // Производен клас
31 class Student : public Person {
32
33     public:
34         void readStudent(const char*, const char*, long,
35         double);
36         void printStudent() const;
37     private:
38         long FN;
39         double gpa;
40 };
41
42 void Student::readStudent(const char* na, const char* uc,
43 long fn, double g) {
44     readPerson(na, uc);
45     FN = fn;
46     gpa = g;
47 }
48
49 void Student::printStudent() const {
50     cout << "Faculty number: " << FN << endl;
51     cout << "GPA: " << gpa << endl;
52 }
53
54 int main() {
55
56     Student s;
57     s.readStudent("Ivan Ivanov", "1234567890", 81992, 4.5);
58     s.printStudent();
59
60     return 0;
61 }
```

Капсулация Пред всяко име на базов клас в заглавието на декларацията на производен клас, може да се постави ключовата дума *public*, *private* или *protected*. Нарича се атрибут за област, тъй като определя областта на наследените членове.

Капсулация
Механизъм за ограничаване на директния достъп до някои от компонентите на обектите.

ООП. Подтипов и параметричен полиморфизъм. Множествено наследяване.

1.Виртуални функции и подтипов полиморфизъм. Абстрактни методи и класове.

В езика за програмиране $C++$, при извикване на функции с едно и също име, е имплементиран механизъм за разпознаване коя функция да се изпълни. Той се изразява в следното: по време на компилация се сравняват формалните с фактическите параметри в обръщението и по правилото за най-добро съвпадане се избира необходимата функция. След заместване на формалните с фактическите параметри, се изпълнява тялото на функцията. При член-функциите на йерархия от класове конфликтът между имената на наследените и собствените методи с един и същ тип на резултата и с един и същи типове на явно указаните формални параметри се разрешава също по време на компилация чрез правилото на локалния приоритет и чрез явно посочване на класа, към който принадлежи методът. В тези два случая, тъй като разрешаването на конфликтите приключва по време на компилация и не може да бъде променяно по време на изпълнение на програмата, се казва че се осъществява статично свързване. Езикът поддържа още един механизъм за разрешаване на връзките, наречен динамично свързване. При него изборът на функция, която трябва да се изпълни, се осъществява по време на изпълнение на програмата. Динамичното свързване капсулира детайлите в реализацията на йерархията. При него не се налага преобразуване на типове, текстовете на програмите се опростяват, а промени се налагат много по-рядко. Прилагането на механизма на динамичното свързване се осъществява над специални член-функции на класове, наречени **виртуални функции** (методи). Те се декларират чрез поставяне на ключовата дума *virtual* пред декларацията им:

```
virtual void print();
```

Чисто виртуални функции наричаме функции, които не са дефинирани, а само декларирани. Синтаксисът за тях е следния:

```
virtual void print() = 0;
```

Полиморфизмът е важна характеристика на обектно-ориентираното програмиране, която се изразява в това, че едни и същи действия се реализират по различен начин в зависимост от обектите, над които се прилагат, т.е. действията са полиморфни (с много форми). Полиморфизмът е свойство на член-функциите на обектите и се реализира чрез виртуални функции. За да се реализира полиморфно действие, класовете над които то ще се прилага, трябва да имат общ родител, т.е. да бъдат производни на един и същ клас. В този клас трябва да бъде дефиниран виртуален метод, съответстващ на полиморфното действие. Във всеки от производните класове този метод може да бъде предефиниран съобразно особеностите на класа. Активирането на полиморфното действие се осъществява чрез указател към

базовия клас или чрез псевдоним на обект на базовия клас. В зависимост от типа на обекта, към който сочи указателят, ще бъде изпълняван един или друг виртуален метод. Ако класовете, над които ще се реализира полиморфното действие, нямат общ родител, такъв може да бъде създаден изкуствено чрез дефиниране на т.н. абстрактен клас.

Пример:

```
#include <iostream>
using namespace std;

class Animal {
    public:
        void print() const {
            cout << "Animal\n";
            cout << "None\n";
        }
};

class Dog : public Animal {
    public:
        void print() const {
            cout << "Animal\n";
            cout << "Dog\n";
        }
};

class Cat : public Animal {
    public:
        void print() const {
            cout << "Animal\n";
            cout << "Cat\n";
        }
};

int main() {

    Animal a;
    a.print();
    Dog d;
    d.print();
    Cat c;
    c.print();

    return 0;
}
```

Член-функцията $voidprint()const$ на всеки от класовете извежда общата информация *Animal* и специфичната за всеки клас информация - определяща вида на животното. ??? не е добър пример

Клас, в който е декларирана поне една чисто виртуална функция, се нарича **абстрактен**. Абстрактните класове имат следните свойства:

- не могат да се създават обекти от абстрактен клас, но могат да се дефинират указатели и псевдоними от такъв клас
- чисто виртуалните член-функции задължително трябва да бъдат предефинирани в класовете, производни на абстрактния клас или да бъдат обявени за чисто виртуални в тях

Абстрактните класове са предназначени да са базови на други класове. Полиморфизмът, с помощта на абстрактните класове, позволява създаването на хетерогенни структури от данни, т.е. елементите на които са от различен тип (това могат да бъдат стек, опашка, дърво, граф и др.).

Пример:

```
class File {
    public:
        // Чисто виртуална функция
        virtual void open() const = 0;

        // Чисто виртуална функция
        virtual void close() const = 0;
};
```

2.Параметричен полиморфизъм. Шаблони на функция и клас.

3.Множествено наследяване.

Наследяване наричаме множествено, когато производният клас наследява директно повече от един базов клас. Чрез това наследяване могат да се изграждат програми с мрежова структура.

Структури от данни и програмиране

Структури от данни. Стек, опашка, списък, дърво. Основни операции върху тях. Реализация.

1.Структури от данни.

Структура от данни наричаме организирана информация, която може

да бъде описана, създадена и обработена с помощта на програма. За да се определи една структура от данни, е необходимо да се опише:

- логическо описание на структурата - описание на базата на декомпозицията ѝ на по-прости структури и декомпозицията на операциите над структурата с по-прости операции
- физическо представяне на структурата - описание на методи за представяне на структурата в паметта на компютъра

2.Списък.

Логическо описание
Нека T е даден тип данни. Линеен списък с елементи от тип T е линейна динамична структура от данни. Определя се като крайна редица от елементи от тип T. Един от елементите, например най-левият, се нарича първи, глава или начален елемент. Останалите елементи на редицата също образуват линеен списък, който се нарича подсписък или опашка на списъка. Над линеен списък могат да се изпълняват следните операции:

- създаване на празен линеен списък
- определяне дали линеен списък е празен
- добавяне на елемент към линеен списък
- премахване на елемент от линеен списък
- достъп до елемента на линеен списък

Операциите добавяне, отстраняване и достъп се извършват в произволно място на списъка. Операциите премахване и достъп са възможни само ако списъкът не е празен. Достъпът е пряк само до първия елемент и е последователен до елементите на опашката на списъка. С цел улесняване на изпълнението на операциите повечето реализации допускат пряк достъп до последния елемент на списъка, както и до произволен елемент, ако е известен адресът му.

Физическо представяне
Използват се главно 2 начина за физическо представяне на линеен списък в паметта на компютъра - последователно и свързано. Последователното представяне не се използва заради добре развитите средства за динамично разпределение на паметта. Свързаното представяне може да реализираме върху списък с една или списък с две връзки. ???????

3.Стек.

Логическо описание
Нека T е даден тип данни. Стек с елементи от тип T е линейна динамична структура от данни. Определя се като крайна редица от елементи от тип

T. Единият край на редицата, в случай че тя не е празна, се нарича връх или начало на стека. Стекът се определя като структура "последен влязъл - пръв излязъл"(Last In First Out - $LIFO$). Над стек могат да се изпълняват следните операции:

- създаване на празен стек
- определяне дали стек е празен
- добавяне на елемент във върха на стек
- премахване на елемент от върха на стек, ако той не е празен
- достъп до елемента от върха на стек, ако той не е празен

Физическо представяне
Последователно представяне При това представяне се запазва блок от паметта, в който стекът да се променя: нараства при добавяне на елементи или намалява при премахване на елементи. При добавяне на елементи, те се поместват в последователни адреси в неизползваната част веднага след върха на стека. Тази операция е възможна до пълното изчерпване на неизползваната част. При отстраняване на елементи се променя указателят към върха на стека. Блокът от паметта, в който се съхранява стекът, се реализира чрез едномерен масив, а указателят към върха на стека - чрез индекс. Свързано представяне При това представяне елементите на стека не се разполагат в последователни адреси в паметта, а са разпръснати на свободни места в паметта. Реализира се като всеки елемент на стека се представя чрез блок с две полета: информационно (съдържа стойността на елемента) и адресно (съдържа адреса на предишния елемент на стека). Адресното поле на последния елемент е $NULL$. Добавянето на елемент се осъществява чрез създаване на нов блок. Премахването на елемент се реализира като в помощна променлива се записва информационното поле на блока, представящ върха на стека, указателят *start* се свързва с адреса от адресното поле, след което се разрушава блока от върха на стека.

4.Опашка.

Логическо описание
Нека T е даден тип данни. Опашка с елементи от тип T е линейна динамична структура от данни. Определя се като крайна редица от елементи от тип T. Единият край на редицата, в случай че тя не е празна, се нарича глава или начало на опашката, а другият край се нарича край или задна част на опашката. Опашката се определя като структура "първи влязъл - първи излязъл"(First In First Out - $FIFO$). Над опашка могат да се изпълняват следните операции:

- създаване на празна опашка
- определяне дали опашка е празна

- добавяне на елемент в само в края на опашката
- премахване на елемент само в началото на опашката, ако тя не е празна
- достъп до елемента в началото на опашката, ако тя не е празна

Физическо представяне
Последователно представяне При това представяне се запазва блок от паметта, в който опашката да се променя: нараства при добавяне на елементи или намалява при премахване на елементи. Включването на елемент в опашката се осъществява чрез поместването му в последователни адреси в неизползваната част веднага след края на опашката. Съществуват разновидности на това представяне. Една от тях, която най-често се използва е цикличното представяне. При него, при изчерпване на масива, ако има свободна памет в началото му, включването се извършва там. При отстраняване на елементи се променя указателят към началото на опашката и се освобождава паметта, заета от отстранения елемент. Блокът от паметта, в който се съхранява опашката, се реализира чрез едномерен масив, а указателите към началото и след края на опашката - чрез индекси. Свързано представяне Това представяне е аналогично на свързаното представяне представяне на стек. Елементите на опашката не се разполагат в последователни адреси в паметта, а са разпръснати на свободни места в паметта. Всеки елемент на опашката се представя чрез блок с две полета: информационно (съдържащо стойността на елемента) и адресно (съдържащо адреса на следващия елемент от опашката). Адресното поле на последния елемент е означено чрез константата $NULL$, указателят към началото на опашката е означен с $front$, а указателят към края й - с $rear$.

5.Дърво.

Дърво от тип T е структура от данни, която е или празна, или е образувана от:

- данна от тип T, наречена корен на двоичното дърво от тип T
- крайно, възможно празно множество от променлив брой несвързани помежду си елементи $T_1, ..., T_n$, които са дървета от тип T, наречени поддървета на дървото от тип T

Линейният списък се нарича изродено дърво, защото може да се разглежда като дърво, всеки възел на което има най-много едно поддърво. Над структурата от данни дърво от тип T, могат да се изпълняват следните операции:

- достъп до връх
- включване и изключване на поддърво
- включване и изключване на връх от тип T

- проверка дали дърво е празно
- обхождане на дърво от тип T (в практиката се използват предхождащо и последващо обхождане)

?????????????? Физическо представяне
Най-често се използва свързано представяне на дърво от тип T. За целта се използва линеен списък, представен с една връзка. При това физическо представяне в един свързан списък се реализират коренът и указателите към поддърветата на дървото от тип T.

6.Двоично дърво.

Логическо описание
Двоично дърво от тип T е структура от данни, която е или празна, или образувана от:

- данна от тип T, наречена корен на дървото от тип T
- двоично дърво от тип T, наречено ляво поддърво на двоичното дърво от тип T
- двоично дъро от тип T, наречено дясно поддърво на двоичното дърво от тип T

Над двоично дърво могат да се изпълняват следните операции:

- достъп до връх
- включване и изключване на връх
- проверка дали двоично дърво е празно
- обхождане на двоично дърво (осъществява се, че изпълнение на три действия - обработка на корена, обхождане на ляво поддърво и обхождане на дясно поддърво)

Съществуват шест начина за обхождане на двоично дърво: ляво-корен-дясно, корен-ляво-дясно, ляво-дясно-корен, дясно-корен-ляво, корен-дясно-ляво и дясно-ляво-корен. Двоично дърво, в което всеки възел, който не е листо, има точно два наследника, се нарича строго. Ако всички листа на строго двоично дърво са на едно и също ниво, двоичното дърво се нарича пълно.

Физическо представяне
Използват се главно 2 начина за физическо представяне на двоично дърво от тип T - свързано и верижно. Свързаното представяне се реализира чрез указател към кутия с три полета - информационно (стойността на корена) и две адресни (представянията на ляво поддърво и дясно поддърво). Верижното (последователно) представяне се реализира чрез три масива - за представяне на върховете на дървото, за адресите на лявото и адресите на

дясното поддърво. Ролята на адреси се изпълнява от индекси. i-ят елемент на масива "върхове съдържа стойността на връх на двоичното дърво, i-ят елемент на масива "ляво поддърво"съдържа адреса на лявото поддърво на поддървото с корен i-тия елемент, а i-ят елемент на масива "дясно поддърво адреса на дясното му поддърво. Поддържа се указател, който съдържа адреса на корена.

Функционално програмиране

Функционално програмиране. Обща характеристика на функционалния стил на програмиране. Дефиниране и използване на функции. Модели на оценяване. Функции от по-висок ред.

1.Характеристика на функционалния стил. Основни компоненти. Примитивни изрази. Дефиниране на функции. Оценяване.

Функционалното програмиране е начин за съставяне на програми, при който единственото действие е обръщението към функции, единственият начин за разделяне на програмата на части е въвеждането на име на функция и задаването за това име на израз, който пресмята стойността на функцията, а единственото правило за композиция е суперпозицията на функции. Тук под функция се разбира програмна част, която "връща"резултат (по-точно, във функционалното програмиране се работи с т. нар. строги функции, които не предизвикват никакви странични ефекти, а само връщат стойности). Най-съществени елементи на функционалния стил на програмиране (по-точно, на програмирането във функционален стил) са дефинирането и използването на функции. Не се използват никакви клетки от паметта и оператори за присвояване и за цикъл, не се описват действия като предаване на управлението и т.н. Програмирането във функционален стил се състои от:

- дефиниране на функции, които пресмятат (връщат) стойности. При това тези стойности еднозначно се определят от стойностите на съответните аргументи (фактически параметри);
- прилагане (апликация) на тези функции върху подходящи аргументи, които също могат да бъдат обръщения към функции, тъй като всяка функция връща стойност. Затова езиците за ФП се наричат още апликативни езици.

Предимства на функционалното програмиране:

- може да се извършва лесна проверка и поправка на съответните програми поради липсата на странични ефекти
- подходящ при проектирането на езици за програмиране, предназначе-

ни за многопроцесорни компютри, в които много от пресмятанията се извършват паралелно (поради липсата на странични ефекти не се налага програмистът да се грижи за евентуални грешки при синхронизацията, причинени от промени на стойности, извършени в неправилен ред);

- свойства на функционалните програми могат да бъдат доказвани точно (с математически средства)

Недостатъци на функционалното програмиране:

- строгата функционалност понякога изисква многократно пресмятане на едни и същи изрази – потенциален проблем, който има добри теоретични и практически решения;
- използването му при решаване на задачи от процедурен (алгоритмичен) характер е неестествено и често неефективно

2.Функции от по-висок ред. Анонимни (ламбда) функции.

Функция от по-висок ред се нарича всяка функция, която получава поне една функция като параметър или връща функция като резултат. Пример

Ламбда нотацията се използва, когато имаме за цел да използваме директно дадена функция, вместо да я именуваме и дефинираме. Пример $\backslash m \rightarrow n + m$. В Haskell този вид изрази се наричат **ламбда изрази**, а функциите дефинирани чрез тях се наричат **анонимни функции**.

Функционално програмиране. Списъци. Потоци и отложено оценяване.

Най-често използванта структура във функционалните езици е **списъкът**. Той представя редица от (променлив брой) елементи, принадлежащи на един и същ тип. Записът на списъците в Haskell е следният:

- [ ] обозначава празен списък (списък без елементи)
- $[e_1...e_n]$ обозначава списък с елементи $e_1...e_n$

Друга форма на запис на списъци от числа, знакове и елементи на изброими типове е чрез аритметична последователност:

- $[n...m]$ е списъкът $[n, n+1, ..., m]$
- $[n, p, ...m]$ е списъкът, чийто първи два елемента са n и p, последният му елемент е m и стъпката на нарастване на елементите му е $p - n$

Конструирането на списъци чрез определяне на техния обхват (List Comprehension) ни дава по-голяма свобода. Синтаксисът е следният: $[expr \mid q_1, ..., q_k]$, където $expr$ е израз, а q_i може да бъде генератор от вида $p \leftarrow lExp$ или тест (филтър), който е булев израз.

Логическо програмиране

Синтаксис и семантика на термовете и формулите на предикатното смятане от първи ред. Унификация.

Дефиниция **Терм** в $\mathcal{L}$ дефинираме индуктивно:

- индивидните променливи са термове от $\mathcal{L}$
- индивидните константи са термове от $\mathcal{L}$
- ако $f \in \mathcal{F}, I(f) = n$, $\tau_1 ... \tau_n$ са термове, то думата $f(\tau_1 ... \tau_n)$ е терм от $\mathcal{L}$

Дефиниция **Предикатна формула** в $\mathcal{L}$ дефинираме индуктивно:

- ако $p \in \mathcal{P}, I(p) = n, \tau_1 ... \tau_n$ са термове, то $p(\tau_1 ... \tau_n)$ е предикатна формула от $\mathcal{L}$
- ако f_1 и f_2 са формули, то $\neg f_1, f_1 \& f_2, f_1 \vee f_2, f_1 \implies f_2, f_1 \iff f_2$
- ако f е предикатна формула и x е индивидна променлива, то $\forall x f$ и $\exists x f$ са предикатни формули

Дефиниция **Субституция** наричаме крайно множество от вида $\{x_1/\tau ... x_n/\tau\}$ за $x_1 ... x_n \in Var$ различни помежду си, $\tau_1, ..., \tau_n \in \tau_{\mathcal{L}}$ и $x_i \neq \tau_i, i \in \{1, ..., n\}$

Дефиниция Нека $E \neq \varnothing$ е множество от термове, а σ е субституция. Казваме, че σ е **унификатор** за E, ако всеки път, когато $\tau_1, \tau_2 \in E$, то $\tau_1\sigma = \tau_2\sigma$, т.е. $E = \{\tau\sigma | \tau \in E\}$ е синглетон.

Дефиниция Нека $E \neq \varnothing$ е множество от термове. Казваме, че субституцията σ е **най-общ унификатор** за E, ако:

- σ е унификатор за E
- всеки път когато ξ е унификатор за E, то $\exists \eta$ - субституция, такава че $\xi = \sigma\eta$

Алгоритъм **(намиране на НОУ)**

Нека $\mathcal{L}$ е език на предикатното смятане от първи ред.

Дефиниция **Структура** за $\mathcal{L}$ наричаме наредената двойка $\mathcal{A} =< A, I >$, където:

- $A \neq \varnothing$ е универсум
- I е интерпретация, при която:
  - $I(c) = c^{\mathcal{A}} \in A$
  - $I(p) = p^{\mathcal{A}} \subseteq A \times ... \times A = A^{\#p}, p = Pred_{\mathcal{L}}, p^{\mathcal{A}} : A^{\#p} \to \{T, F\}$

– $I(f) = f^{\mathcal{A}} : A^{\#f} \to A, f \in Func_{\mathcal{L}}$

Дефиниция Нека $\mathcal{A}$ е структура за $\mathcal{L}$. **Оценка** на v в $\mathcal{A}$ наричаме изображението $v : Var \to A$, т.е. $v(x) \in A$

Дефиниция Нека $\mathcal{A}$ е структура за $\mathcal{L}$ и v е оценка от $\mathcal{A}$. За всеки терм стойността му в $\mathcal{A}$ при оценката v бележим така: $||\tau||^{\mathcal{A}}[v]$ или $\tau^{\mathcal{A}}[v]$

Дефиниция Нека $\varphi \in F_\alpha$. Казваме, че φ е **изпълнима**, ако съществува структура $\mathcal{A}$ и оценка v такава, че $\mathcal{A} \models_v \varphi$. Ако Γ е множество от формули над $\mathcal{L}$, то Γ е **изпълнимо**, ако съществува структура $\mathcal{A}$ и оценка v такава, че $(\forall \psi \in \Gamma)[\mathcal{A} \models_v \psi]$

Дефиниция Казваме, че формулата φ има **модел**, ако има структура $\mathcal{A}$ такава, че $\mathcal{A} \models \varphi$ (т.е. при всяка оценка в $\mathcal{A}$). Казваме, че множество от формули Γ има **модел**, ако има структура $\mathcal{A}$ такава, че $\mathcal{A} \models \Gamma \iff (\forall \psi \in \Gamma)[\mathcal{A} \models \psi]$

Дефиниция Казваме, че формулата φ е **затворена**, ако $Var^{free}[\varphi] = \varnothing$

Дефиниция Казваме, че формулата φ е в **ПНФ**, ако $\varphi = \underbrace{Q_1 x_1 ... Q_n x_n}_{\text{кванторен префикс}} \Theta$, където $Q_1 ... Q_n$ са квантори, а $x_1 ... x_n$ са две по две различни променливи.

Дефиниция Казваме, че формулата φ е **универсална**, ако φ е в ПНФ и кванторния префикс има само квантори за всеобщност.

Дефиниция Казваме, че съждителна формула е **вярна при булева интерпретация** I_0, ако $I(\varphi) = \tau$, където I е единственото продължение на I_0 от $PVAR$ до множеството от всички съждителни формули ($I_0 \models^\delta \varphi \iff I(\varphi) = \tau$)

Дефиниция Нека Δ е множество от формули. Казваме, че Δ е **булево изпълнимо**, ако има интерпретация I_0, която е булев модел за Δ, т.е. за всяка формула φ от Δ е в сила, че $I_0 \models^\delta \varphi$ (бележим с $I_0 \models^\delta \Delta$)

Дефиниция Нека φ е затворена формула в ПНФ, $\varphi = \forall x_1 ... \forall x_n \Theta$, т.е. е универсална и безкванторна за Θ. Нека $\tau_1, ..., \tau_n \in \tau_{\mathcal{L}}$. Формулата $\Theta[x_1/\tau_1, ..., x_n/\tau_n]$ се нарича **затворен частен случай** на φ.
$S_i(\Gamma) = \bigcup_{\varphi \in \Gamma} S_i(\varphi) = \bigcup_{\varphi \in \Gamma} \{\Theta[x_1/\tau_1, ..., x_n/\tau_n] \mid \varphi = \forall x_1 ... \forall x_n \Theta, \Theta$ е безкв. формула$\}$

Теорема
Нека Γ е множество от затворени универсални формули над $\mathcal{L}$ - език от първи ред. Тогава Γ има модел $\iff S_i(\Gamma)$ е булево изпълнимо

Метод на резолюцията в съждителното и в предикатното смятане от първи ред. Хорнови клаузи.

Дефиниция **Съждителен литерал** наричаме съждителна променлива p или съждителна формула $\neg p, p \in PVAR$

Дефиниция **Съждителен дизюнкт** наричаме крайно множество от съждителни литерали. Празният дизюнкт означаваме с $\blacksquare = \{\}$

Дефиниция Дизюнктът D е **резолвента** на D_1 и D_2, ако $\exists L$ - литерал, такъв че $D = Res_L(D_1, D_2)$.

Дефиниция Нека S е множество от съждителни дизюнкти. **Резолютивен извод** от S наричаме крайна редица $D_0 ... D_r$ от дизюнкти, така че $D_0, D_1 \in S$ и $1 < i \leqslant n : D_i$

- $\in S$
- $\exists j, k : 0 \leqslant j < k < i$ и има $h \in D_j$ и $h^{\partial} \in D_k$ и $D_i = R_L(D_j, D_k)$

Теорема **(Коректност на резолютивна изводимост)**
Нека S е множество от дизюнкти. Ако $S \overset{r}{\vdash} \blacksquare$, то S е неизпълнимо.

Доказателство. $S \overset{r}{\vdash} \blacksquare \iff \blacksquare \in S^* \iff S$ е неизпълнимо $\iff S$ е неизпълнимо $\square$

Лема
Нека S^* е множество от съждителни дизюнкти, което е затворено относно правилото за резолюцията. Ако $\blacksquare \notin S^*$, то S е изпълнимо.

Теорема **(Пълнота на резолютивна изводимост)**
Нека S е множество от дизюнкти. Ако S е неизпълнимо, то $S \overset{r}{\vdash} \blacksquare$.

Доказателство. Допускаме, че S е неизпълнимо, но $S \overset{r}{\vdash} \blacksquare$. Тогава имаме, че $S \overset{r}{\nvdash} \blacksquare \iff \blacksquare \notin S^*$ и от лемата следва, че S е изпълнимо. Абсурд. $\square$

Метод на резолюцията
1. Привеждаме формулата в отрицателна нормална форма
2. Привеждаме формулата в пренексна нормална форма
3. Привеждаме формулата в Скулемова нормална форма
3. Привеждаме формулата в конюктивна нормална форма

Дефиниция Един дизюнкт наричаме **Хорнов**, ако съдържа не повече от един положителен литерал

- $\{p\}$ - факт

- $\{p, \neg q_1, ..., \neg q_n\}$ - правило, $n > 0$
- $\{\neg q_1, ..., \neg q_n\}$ - цел, $n > 0$

Теорема
Изпълнимите множества от Хорнови дизюнкти имат най-малък модел

Доказателство. Нека $P_0 P_i$ е множество от променливи. □