# ****[DSA] Zadanie 3 – Popolvár****

**Autor: Ema Richnáková**

# ****Implementácia****

# ****Mapy****

Pri spustení programu sa program pýta na výber mapy. Mapu môžeme získať buď zo súboru *mapa.txt*, kde sú údaje napísané vo formáte:

*[výška mapy] [šírka mapy] [čas]  
[výpis mapy]*

Príklad:

4 6 15  
HCPNHC  
CCHPNC  
CCCCCH  
PNHHCD

Ak súbor nie je dostupný, dá sa vybrať z predefinovaných máp. Ku každej je v programe popis, čo by mala mapa predstavovať.

Alebo je v programe aj možnosť generovania náhodnej mapy. Parametre potrebné k mape sú definované v konštantách na začiatku kódu.

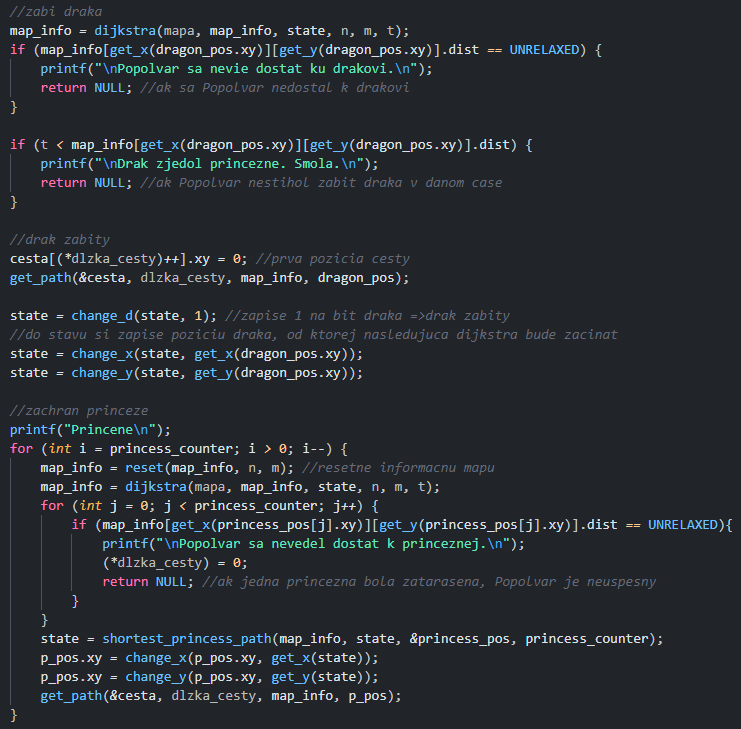
* **Záchrana princezien**

Pozície jednotlivých polí si ukladá do štruktúry *POSITION*, ktorá má v sebe jeden *unsigned int (32 bit)*, v ktorom si ukladá 2 hodnoty zároveň a to sú *x* a *y* poľa v mape. Ukladá si ich tam pomocou bitových operácií, ktoré budú objasnené neskôr v dokumente. Pre spomenuté čísla je vyhradených 13 bitov pre každé v 32-bitovom čísle. Takáto implementácia spôsobuje efektívnejšie využitie pamäte programom, keďže dané čísla nezaberajú 32 bitov v pamäti, ale 13 každé. Keďže mapa nebude väčšia ako 100x100, tak dá sa predpokladať, že čísla nebudú zaberať viac ako 13-bitov.

Dôležitá premenná je *state*, do ktorej sa ukladajú údaje o začiatočnej polohe, z ktorej sa bude začínať dijkstrov algoritmus a aj údaje o tom, či bol drak zabitý alebo konkrétna princezná zachránená. Údaj o polohe funguje na rovnakom princípe ako ukladanie do premennej v štruktúre *POSITION*, kedy prvých 26 bitov je zabratých polohou, a zvyšných 6 bitov je určených pre draka a princezné, keďže ich stavy stačí naznačiť 1 bitom a teda 0 alebo 1 (resp. nezabitý/zabitý, nezachránená/zachránená).  
Bitová vizualizácia premennej *state*:  


V programe si pred samotným hľadaním nájde pozície draka a princezien pomocou funkcie *find\_positions*, kedy taktiež aj spočíta, koľko princezien sa nachádza na mape.

Popolvár začína na poly [0, 0] a pomocou funkcie *dijkstra* nájde Popolvár najkratšiu cestu ku drakovi. Ak existuje, uloží si ju a môže pokračovať v hľadaní princezien.  
Spustí dijkstrov algoritmus od draka a hľadá. Ak hľadanie skončí, následne si pozrie, ktoré princezné ešte neboli zachránené a ku ktorej viedla najkratšia cesta. Cestu si uloží, a dijkstrov algoritmus bude spúšťať od zachránenej princeznej a bude postupovať rovnako, ako od draka.   
Hľadanie cesty nie je robené cez permutácie, kedy by sa postupovalo tak, že by sa najprv našli všetky kombinácie, ako sa dajú navštíviť princezné a potom by sa cez dijkstov algoritmus takou postupnosťou navštevovali.   
Implementované hľadanie je teda o čosi rýchlejšie, najmä pri väčšom počte princezien, ale nie je zaručené, že bude nájdená tá najrýchlejšia cesta.



Program je ošetrený pre prípady, ak Popolvár začína na prekážke, ak sa nevedel dostať k drakovi, ak nebol dostatočne rýchly a drak zjedol princezné alebo ak sa nevedel dostať k jednej z princezien. Ak nastane prípad z vyššie uvedených, tak funkcia *zachran\_princezne*, vráti namiesto cesty NULL s dĺžkou cesty 0.

* **Dijkstrov algoritmus**

 Vo funkcii *dijkstra*, je tento algoritmus implementovaný. Funguje na takom princípe, ak sa nachádza Popolvár na nejakom poli, obzrie sa okolo všetkých svojich susedných polí, a ak je pole validné, resp. ak nevychádza z mapy, nie je to pole, z ktorého sa dostal na terajšie pole, nie je to ani prekážka a vie sa naňho dostať s menšou vzdialenosťou, akou má momentálne pole určené (ak nebolo pole ešte navštívené, má nastavenú vzdialenosť na ,,nekonečno‘‘), vtedy dané pole pridáva do štruktúry min heap-u. Ak Popolvár navštívi všetkých validných susedov, bude sa posúvať po mape. Vždy pri posúvaní vyberá pole s najmenšou vzdialenosťou. Vyberá sa pole z min heap-u, kde pole s najmenšou vzdialenosťou bude vždy na vrchu min heap-u a teda si vyberie prvý prvok min heap-u. Takto pokračuje, pokiaľ nenavštívil všetky polia mapy a teda keď je min heap prázdny.

* **Min heap**

Min heap (ďalej len heap) je štruktúra podobná binárnemu stromu, kedy koreň stromu je najmenší prvok a má vždy pár detí, ktoré sú väčšie. Prvok do heap-u sa vkladá pomocou funkcie *add\_min\_heap*, ktorá funguje tak, že vkladaný prvok vloží na koniec heap-u a porovnáva daný prvok z jeho rodičom, ak je daný prvok menší ako rodič, tak sa vymenia.  
Pri vyberaní prvku z heap-u sa vyberá iba prvý prvok a teda ten najmenší. No ak sa prvý prvok odstráni, na jeho miesto sa uloží posledný prvok z heap-u a musí sa celá štruktúra usporiadať pomocou funkcie *min\_heapify*, aby sa zachovala myšlienka min heap-u.  
V *min\_heapify* sa porovnáva rodič s menším potomkom ako je samotný rodič. Ak taký existuje, tak si vymenia miesta v heap-e. Táto funkcia pokračuje rekurzívne, za rodiča si zvolí predošlého najmenšieho potomka a porovnáva ho s jeho potomkami.

* **Bitové operácie**

Vysvetlené po funkciách:

*uint32\_t change\_state(uint32\_t state, unsigned new\_val, unsigned offset, unsigned bit\_size);*

Slúži na zmenu premennej *state*, kam sa bude ukladať nová hodnota z *new\_val*, táto hodnota bude uložená na mieste, ku ktorému je offset v premennej *offset* a bude sa tam ukladať toľko bitov, koľko je zapísané v *bit\_size.* Najprv sa vytvorí akoby resetovacia maska, ktorá zresetuje určený počet bitov na 0 na mieste v *state*, kde sa bude ukladať nová hodnota a potom sa bity zo *state* spoja *OR* operáciou s bitmi z *new\_val*, ktoré sú posunuté presne o toľko bitov, kde sa bude nachádzať v *state,* určené offsetom*.*

unsigned get\_x(uint32\_t state);

Číslo x-ovej súradnice polohy sa nachádza v prvých 13-stich bitoch z ľava v premennej *state,* takže ak posunieme *state* o 19 bitov doľava, dostanem hodnotu x.

uint32\_t change\_x(uint32\_t state, unsigned x);

Číslo x-ovej súradnice polohy sa mení pomocou funkcie *change\_state,* len sa dosadia príslušné hodnoty.

unsigned get\_y(uint32\_t state);

Číslo y-ovej súradnice polohy sa nachádza v druhých 13-stich bitoch z ľava v premennej *state,* takže najprv sa musí odstrániť prvých 13 bitov, kde je uložená x-ová súradnica, a to takým spôsobom, že jednotku posunieme o 13 bitov, to nám vytvorí 14-bitové číslo, od ktorého keď odčítame 1, tak dostaneme 13-bitovú postupnosť jednotiek, ktorú posunieme o offset k x-ku a ak to obrátime operáciou *NOT,* tak na mieste, kde by malo byť x sú nuly, inak sú jednotky a ak to operáciou *AND* spojíme so *state,* tak premažeme x nulami, ostatné zostane a to celé posunieme o 6 bitov, kedy sa zbavíme ostatných bitov ku drakovi a princeznám a získame čisté y.

uint32\_t change\_y(uint32\_t state, unsigned y);

Číslo y-ovej súradnice polohy sa mení pomocou funkcie *change\_state,* len sa dosadia príslušné hodnoty.

unsigned get\_d(uint32\_t state);

Získava sa podobne ako hodnota y, ale premazáva sa celé x a y a bity princezien a dostane sa bit draka.

uint32\_t change\_d(uint32\_t state, unsigned D);

Bit draka sa mení pomocou funkcie *change\_state,* len sa dosadia príslušné hodnoty.

unsigned get\_p(uint32\_t state, unsigned num);

Získava sa podobne ako hodnota bitu draka, ale premazáva sa celé x a y a bit draka a podľa toho, k akej princeznej chceme pristupovať, tak sa *posúvame o* rôzny počet bitov.

uint32\_t change\_p(uint32\_t state, unsigned num, unsigned P);

Bit princezny sa mení pomocou funkcie *change\_state,* len sa dosadia príslušné hodnoty.

unsigned print\_bin32(uint32\_t num, int bit);

Slúžilo na kontrolu posúvania bitov.

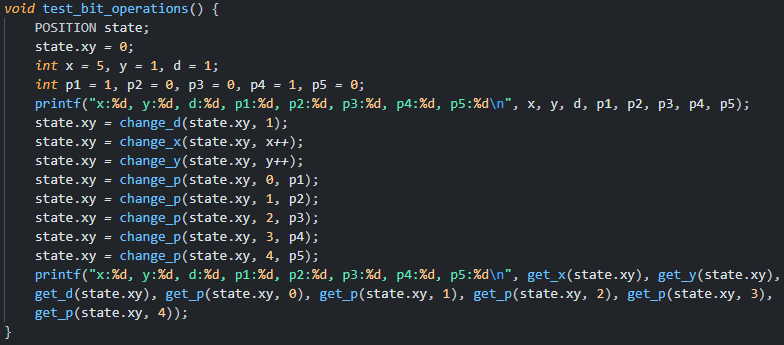
# ****Testovanie****

2 fázy testovania:

* Manuálne
* Automatické

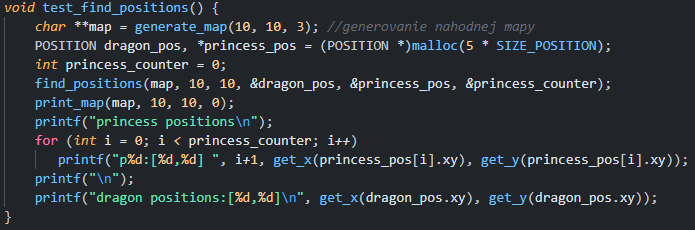
**Manuálne testovanie**  
Manuálne testovanie prebiehalo počas tvorenia nového bloku kódu, kedy kontrola prebiehala pomocou rôznych výpisov a debugovania.  
Pri fungujúcom programe vizuálna kontrola prebiehala pomocou nástroja <https://popolvar.surge.sh/>.

**Automatické testovanie**Jednotkové testy boli napísané na kontrolu správnosti bitových operácií:



Kontrolovalo sa menenie a získavanie jednotlivých hodnôt z premennej *state*.

Ďalší jednotkový test bol napísaný pre kontrolu hľadania polohy draka a princezien.



Typy máp, ktoré majú vykresľovať hraničné situácie sú vypísané vo funkcii *main\_zo\_zadania\_case\_2*. Hraničné situácie: Popolvár sa nevie dostať k zatarasenému drakovi, Popolvár nestihol zabiť draka v čas, Popolvár sa nevie dostať k zatarasenej princeznej, mapa sa začína prekážkou, časovo kratšia cesta je dlhšia, čo sa týka políčok a ak sú 2 princezné rovnako ďaleko od draka, ako Popolvár vyhodnotí najkratšiu cestu.  
Všetky typy máp mi moja implementácia zbehla s dobrými výsledkami, až na mapu  
  
, keďže nie je implementované prechádzanie princezien podľa ich permutácii, tak od draka išiel Popolvár ku strednej princeznej a nie k spodnej.

# ****Zhodnotenie****

Zložitosti:

* Priestorová zložitosť dijkstrovho algoritmu, kde je využitý min heap a pole s najkratšou cestou, sa zložitosť rovná O(x) + O(x) = O(2\*x) = O(x) , kde x predstavuje veľkosť min heap-u aj veľkosť poľa (čo v mojom programe predstavuje výška\*šírka).
* Časová zložitosť záleží od vkladania do min heap-u, čo je O(log x), kedy x je veľkosť min heap-u.