1 Úloha 1

Tato konstanta se v ukázce nachází ze dvou důvodů, a to kvůli tomu, že krok změny rychlosti může být takový, že můžeme dosáhnout zadané rychlosti jen přibližně, a kvůli chybám, které se vyskytují při výpočtech s čísly s plovoucí čárkou (floating-point error).

První případ může nastat tehdy, když krok změny rychlosti je 0,1 a požadovaná rychlost je 0,75. Tehdy nemůžeme dosáhnout přesně této rychlosti, proto pokud bychom tuto konstantu nepoužívali, došlo by k tomu, že by po celou dobu vracela funkce `False` a rychlost by se střídala mezi 0,7 a 0,8. Abychom tohle vyřešili, musíme nastavit `ODCHYLKA_SENZORU = 0,1`.

Druhý případ je dle mého názoru zásadnější než první. Protože na počítačích jsou většinou desetinná čísla ukládána jako čísla s plovoucí čárkou, u nichž se zapisují místa po čárce v dvojkové soustavě, nemůže mnoho čísel být vyjádřeno precizně. To je též důvod, proč většina programovacích jazyků bude vracet $0,1+0,2 = 0,300...04$. Proto kdyby konstanta `ODCHYLKA_SENZORU` nebyla použita, došlo by ke stejnému případu jako výše.

2 Problém 4

Důležité informace, které se musíme u tohoto simulátoru uvědomit je, že když jsou příkazy `e.speedUp(i)` a `e.speedDown(i)` zavolány několikrát v jednom kole, na konečnou rychlost má vliv jen poslední z těchto příkazů. Proto nemůže dojít k tomu, že v první a druhá funkce `goToFloor(...)` si navzájem vynulují nastavení rychlosti.

U mé implementace, které spoléhají jen na parametry a nevyužívají žádné globální proměnné, dojde k tomu, že se výtah bude pohybovat mezi přízemím a bodem, kam výtah dorazí nejmenší nenulovou rychlostí za jedno kolo a který je výše než přízemí.

Takhle to bude i u většiny implementací, což lze vysvětlit následovně. V první kole první funkce spustí příkaz k zvýšení rychlosti a druhá funkce nebude nic dělat a vrátí `True`, protože se nacházíme v přízemí a změny vyvolané přechodovou funkcí se projeví až v dalším kole. V druhém kole musí první funkce navýšit rychlost, ale druhá funkce to přepíše a rychlost vynuluje, protože se bude pokoušet výtah dostat zpět do přízemí. Ve třetím kole první funkce zase chce navyšovat rychlost, avšak druhá funkce to znova přepíše a obrátí směr výtahu směrem k přízemí. Ve čtvrtém kole dorazí výtah k přízemí a obě funkce mu vynulují rychlost. Dál se cyklus opakuje.

Pokud by chtěl někdo tento problém vyřešit, může vytvořit globální proměnnou typu `set`, která bude sloužit jako fronta bez duplicitních položek. Pak by to šlo implementovat tak, že by docházelo k požadovanému výsledku, a to pohybu mezi přízemím a druhým patrem.

3 Problém 6

Nejdříve rozeberu tento problém a následně pak se pokusím přijít na nějaké řešení tohoto problému.

Musíme se nejdříve zamyslet, jak by se výtah měl chovat. Protože nechceme, aby výtah začal prudce brzdit, protože místo toho, aby jel původně např. do druhého patra, změnil cíl na první patro, jakmile bude mít výtah cíl jízdy, nebude ho měnit.

Ze zadání je jasné, že budeme potřebovat ukládat stisknutá tlačítka do fronty. Zároveň je budeme muset nějakým způsobem seřazovat. Avšak musíme si uvědomit, že patra, které jsou výš než je aktuální cíl výtahu (když čeká, bude aktuální cíl roven patru, kde výtah čeká), musíme seřazovat vzestupně a patra, které jsou níž než je aktuální cíl výtahu, musíme seřazovat sestupně. Pokud je stisknuto tlačítko patra, kde se výtah zrovna nachází, neuděláme nic.

Proto budou potřeba dvě různé proměnné pro ukládání pater k navštívení. Když do nich budeme přidávat další patro, vyhodnotíme, jestli je výš nebo níž než je výtah v daný okamžik, a podle toho ji vložíme do jedné z těchto proměnných tak, aby zůstali seřazené. Pro získání dalšího patra a následné jeho odstranění budeme ukládat směr jízdy výtahu.

Musíme též uvážit, jestli má výtah měnit směr jízdy během toho, co proměnné pro ukládání pater nejsou prázdné, nebo jakmile v jednom směru navštíví všechny patra, které měl navštívit. Přikláním se spíše k druhé možnosti, přestože to někdy bude znamenat delší uraženou vzdálenost, ale nebude pak docházet k některým zvláštím případům.

První řešení, které mě napadlo, používá dvě pole a binární vyhledávání. Při přidávání dalšího patra spustíme binární vyhledávání na poli, kam se dané patro má přidat. Když patro už bude v daném poli, nebudeme dělat nic, ale když nic nenajde, zjistíme index, kam to patro přidat, a zbytek pater v poli posuneme. Binární vyhledávání má časovou složitost $\mathcal{O}(\log n)$ a posunování prvků v poli $\mathcal{O}(n)$, proto celková časová složitost přidávání prvků bude $\mathcal{O}(n)$. Pokud budeme dávat patra s nejvyšší prioritou nakonec, odstraňování a získávání patra s nejvyšší prioritou bude konstantní. Budeme ukládat jenom ty pole, proto prostorová složitost bude $\mathcal{O}(n)$.

Další řešení využívá spojové seznam. U spojového seznamu se nevyplácí používat binární vyhledávání, proto budeme prohledávat patra postupně od začátku, dokud nenajdeme dané patro nebo vyhodnotíme, že se dané patro v seznamu nenachází. Pak přidáme patro do seznamu tak, aby seznam zůstal seřazený. Vyhledávání teď bude trvat $\mathcal{O}(n)$ času, ale přidávání doprostřed je teď konstantní, proto celková časová složitost je $\mathcal{O}(n)$. Můžeme uvažovat o vytvoření "skip listu" pro rychlejší prohledávání na úkor prostorové složitosti, ale neznáme předem velikost seznamu, proto by to dělilo časovou složitost jen konstanta-krát, což se na asymptotické časové složitosti příliš neprojeví. Avšak v realitě, když nastavíme ideální konstantu, může tato implementace mít zásadní efekt. Odstraňování a získávání patra s nevyšší prioritou je konstantní, stejně jako u prvního řešení.

Poslední řešení bude nejsložitější na implementaci, ale celkově nejefektivnější. Když budeme využívat vyvažované binární vyhledávací stromy, při přidávání dalšího patra dostaneme časovou složitost $\mathcal{O}(\log n)$. Pokud spolu se stromem implementujeme iterátor, získáme konstantní průměrnou časovou složitost získání a odstranění patra s nejvyšší prioritou s nejhorším případem $\mathcal{O}(\log n)$. Toto si díky podmínce, že výtah nemůže měnit svůj cíl jízdy, můžeme dovolit, jinak by to nemohlo fungovat.