zda potřebujeme dvakrát nebo třikrát víc paměti nebo instrukcí procesoru, než je množství vstupních dat. Absolutní konstanty a koeficienty úměrnosti prostě zanedbáváme. Ba ještě víc: má-li výpočtová složitost tvar polynomu (např. $5 \cdot N^3 + 4 \cdot N^2 + 7$), zajímá nás jenom nejrychleji rostoucí člen (tj. N^3), ostatní členy zanedbáme. To má ovšem za následek, že takto vyjádřená výpočtová složitost platí až pro dostatečně velké hodnoty N.

Na obr. 1.2 vidíme, že např. lineární funkce $10 \cdot N$ roste zpočátku rychleji než kvadratická funkce N^2, teprve až pro velká N roste rychleji funkce kvadratická. Proto se takto vyjádřená výpočtová složitost někdy také nazývá složitostí asymptotickou.[26]

V praxi nás zajímají zejména tyto výpočtové složitosti:

konstantní	C, 1 apod.	např. přístup k prvku pole
logaritmická	$\log N$	např. hledání prvku ve vyhledávacím stromě s N vrcholy
lineární	N	např. prohledání spojového seznamu s N členy
lineárně logaritmická	$N \cdot \log N$	např. seřazení pole heapsortem
kvadratická	N^2	např. cyklus v cyklu
exponenciální	k^N	např. hledání cesty dlouhé N hran ve stromě
superexponenciální	$\gg k^N$	např. Ackermannova funkce

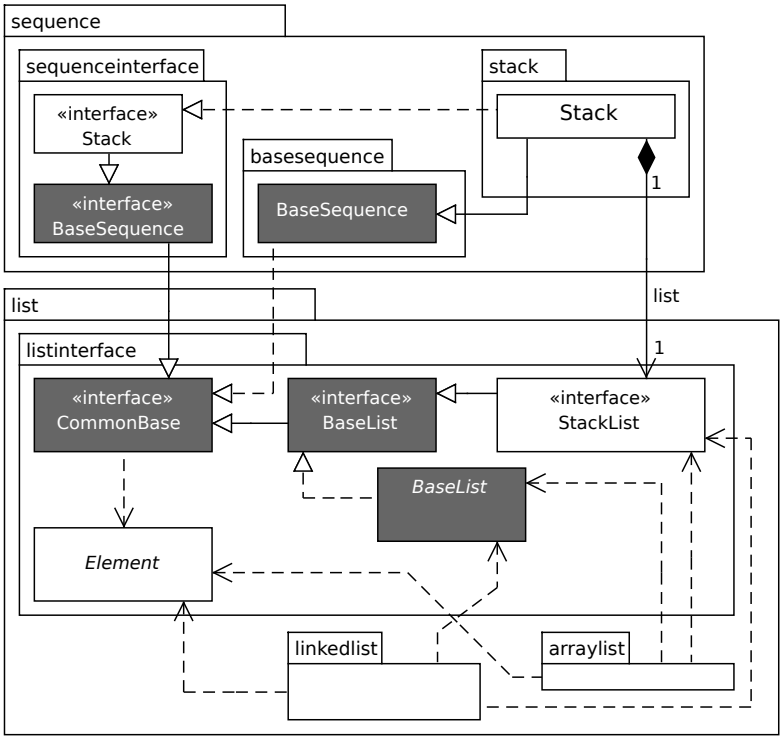
Zhruba můžeme odhadnout, že algoritmy s menší složitostí než exponenciální můžeme běžně používat, exponenciální algoritmy používáme jen po dobré úvaze v omezeném rozsahu N, kdežto superexponenciální algoritmy bývají úplně nepoužitelné. Příklad takového nepoužitelně složitého algoritmu je tzv. *Ackermannova funkce,* kterou

[26] To znamená, že jsme vyjádřili hranici výpočtové složitosti, ke které se skutečná složitost přibližuje tím blíž, čím větší je N.

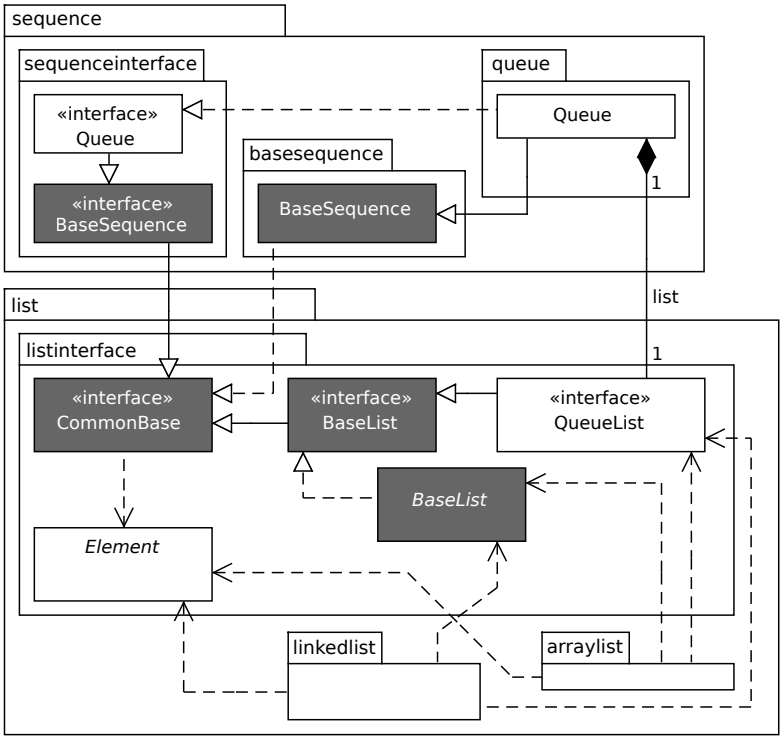
takhle definovat nedá, protože musí vyhovovat matematické definici funkce. A funkce může mít pro každou hodnotu argumentu právě jednu (v případě parciální funkce nejvýše jednu) funkční hodnotu, a to je v případě operace *Pop* nová hodnota změněného zásobníku. Při programování se však nemusíme přísně řídit matematickou definicí funkce a můžeme si dovolit praktičtější implementaci operace *Pop*.

3.2.2 Implementace obecného zásobníku

Rozhraní `Stack` je v knihovně implementováno třídou `Stack` (viz obr. 3.11, 3.12 a 3.13). Třída `Stack` vznikla specializací (tj. děděním) z třídy `BaseSequence` – to je společný základ všech posloupností:



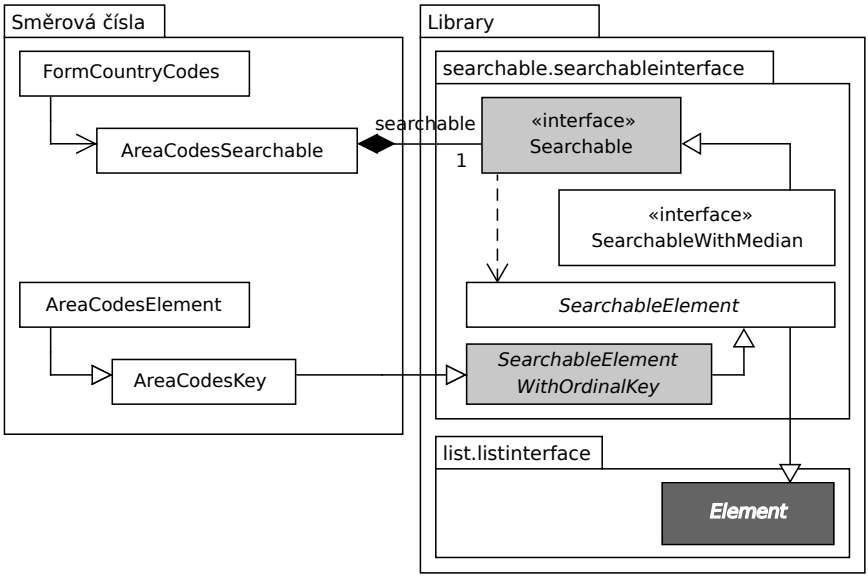
Obr. 3.11: Design obecného zásobníku `Stack`



Obr. 3.28: Design obecné fronty `Queue`

vit novou nerozkolísanou hodnotu otáček vždycky, když nám přijde nový pulz ze snímače. Kdybychom počítali aritmetický průměr třeba ze šedesáti hodnot, ty pak zahodili a čekali, až naměříme dalších šedesát hodnot, měřili bychom aktuální hodnoty otáček šedesátkrát méně často – a považte, co vám motor někdy dokáže udělat za jedinou sekundu, když pořádně šlápnete na plyn (nebo taky když prudce sundáte nohu z plynu)!

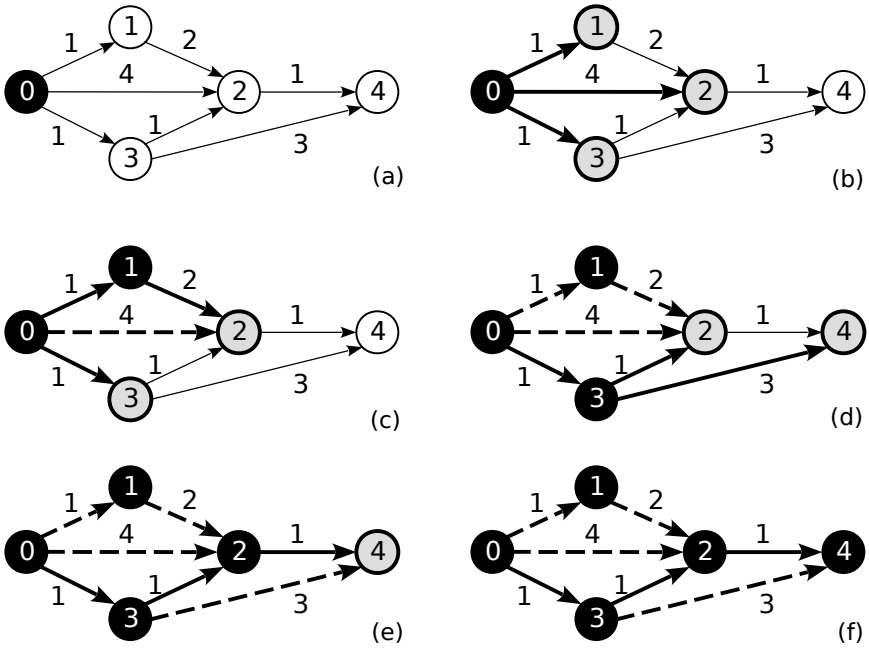
Pro naše potřeby si úlohu zjednodušíme: místo snímače zubů umístíme do formuláře tlačítko a tím budeme simulovat „cvakání“ zubů procházejících okolo snímače. Frekvenci otáček budeme vypisovat do textových políček – jednak frekvenci okamžitou (nezprůměrovanou), jednak klouzavý průměr z několika posledních hodnot. Tento formulář vidíte na obrázku 3.25.



Obr. 5.2: Design aplikace *„Směrová čísla“*

nosti dokážeme hledat rychleji než v posloupnosti neuspořádané). To si podrobně ukážeme v následujících podkapitolách.

Uživatelské rozhraní aplikace tvoří formulář typu `FormCountryCodes`. Aplikace je oddělená od knihovny opakovaně použitelných tříd pomocí rozhraní `Searchable` a hierarchie abstraktních tříd `Element` – `SearchableElement` – `SearchableElementWithOrdinalKey`. Aplikace dědí abstraktní třídu `SearchableElementWithOrdinalKey`, která slouží jako obecný prvek posloupnosti, a specializuje ji na prvek obsahující celočíselný klíč `AreaCodesKey`. Z této třídy dále odvozuje prvek s klíčem i hodnotou typu `AreaCodesElement`. Ve formuláři se pak využívá speciální vyhledávací datová struktura typu `AreaCodesSearchable`, která je navržena jako obal okolo obecné vyhledávací struktury představované rozhraním `Searchable`. Rozhraní `Searchable` je pak implementováno v knihovně různým způsobem: jako různé druhy vyhledávacích posloupností, jako strom nebo jako tabulka.



Obr. 10.19: Ukázka Dijkstrova algoritmu

3. Obr. 10.19c: Vrchol 1 se nastaví do stavu dokončený, do vrcholu 2 byla nalezena kratší cesta.

4. Obr. 10.19d: Vrchol 3 se nastaví do stavu dokončený, do vrcholu 2 byla nalezena ještě kratší cesta, bylo dosaženo cílového vrcholu 4, ale ještě není dokončený.

5. Obr. 10.19e: Vrchol 2 se nastaví do stavu dokončený, do vrcholu 4 byla nalezena kratší cesta.

6. Obr. 10.19f: Cílový vrchol se nastaví do stavu dokončený.

Tento příklad Dijkstrova algoritmu v 6 krocích je stručně shrnut do tabulky 10.1. Kurzívou je v každém kroku vyznačen navštívený vrchol nejblíž od vrcholu výchozího.

Při odhadu časové složitosti vyjdeme z toho, že v krajně nepříznivém případě musíme navštívit všechny vrcholy grafu, některé vrcholy

Tab. 10.1: Příklad Dijkstrova algoritmu v 6 krocích

Krok \ vrchol	0	1	2	3	4
1	0, Cpl.	Unr.	Unr.	Unr.	Unr.
2		*1, Vis.*	4, Vis.	1, Vis.	
3		1, Cpl.	3, Vis.	*1, Vis.*	
4			*2, Vis.*	1, Cpl.	4, Vis.
5			2, Cpl.		*3, Vis.*
6					3, Cpl.

pak navštívíme dokonce víckrát. Každý vrchol grafu, který leží na nejkratší cestě (a může se tedy stát součástí řešení), musíme dokončit. V krajním případě musíme dokončit všech V vrcholů. Dokončený vrchol vybíráme z prioritní fronty – na to potřebujeme $\log V$ kroků výpočtu (v případě, že případě, že prioritní frontu implementujeme haldou). Z každého z nejvýše V dokončených vrcholů pak musíme prohlédnout i všechny jeho následovníky, kterých může být také až V. V tom případě vyjde odhad časové složitosti úměrný $V^2 \cdot \log_2 V$ vzhledem k počtu vrcholů. Vezmeme-li však v úvahu i počet hran E, můžeme využít i toho, že každou hranou projdeme nejvýše jednou. Pak vyjde časová složitost nanejvýš přímo úměrná $(V + E) \cdot \log_2 V$, kde V je počet vrcholů a E je počet hran.

Jednoduchou úpravou můžeme změnit Dijkstrův algoritmus tak, aby počítal nejkratší cestu i v grafech se záporným ohodnocením hran, ale bez cyklů. Nejkratší cestu v grafu bez ohodnocení můžeme hledat Dijkstrovým algoritmem tak, že všechny hrany fiktivně ohodnotíme jedničkou. Doc. Töpfer v knize [60] popisuje i další zajímavé modifikace Dijkstrova algoritmu, např. hledání nejkratší cesty v grafu, kde hrany jsou ohodnoceny několika hodnotami (např. doba jízdy a cena). Dejme tomu, že hledáme nejrychlejší spojení a z nich nás pak zajímá to nejlacinější. Jde vlastně jen o to, abychom uměli správně porovnat dvě dvojice hodnot: doba jízdy má přednost, a když je stejná, potom rozhoduje cena.