Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Vizualizace standartních algoritmů nestandartním způsobem

Plzeň 2012 Libor Váchal

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 10. května 2012

Libor Váchal

Abstract

Visualization of standard algorithms in non-standard way.

The objective of this work is to create a set of visualizations of fundamental data structures and algorithms. These visualizations should give the reader a clear view of the functionality and implementation of its solution. This work also familiarize the reader with concepts such as complexity or recursion.

Poděkování

Rád bych poděkoval vedoucí práce Ing. Janě Hájkové, Ph.D. za cenné rady při návrhu appletů a čas strávený při konzultacích. Dále bych rád poděkoval Michalovi Konkolovi za vytvoření šablony pro systém TEX, která byla použita při psaní této práce.

Obsah

1 Úvod **1**

2 Algoritmy a datové struktury **2**

2.1 Řadící algoritmy 2

2.1.1 Insert Sort 3

2.1.2 Bubble Sort 5

2.1.3 Select Sort 7

2.1.4 Shell Sort 8

2.1.5 Quick Sort 11

2.1.6 Merge Sort 14

2.1.7 Heap Sort 18

2.2 Datové struktury 23

2.2.1 Spojový seznam 23

2.2.2 Zásobník 25

2.2.3 Fronta 26

2.2.4 Binární vyhledávací strom 27

2.2.5 Halda 28

3 Vizualizace **30**

3.1 Výběr prostředku pro vizualizaci 30

3.1.1 Adobe Flash 31

3.1.2 JavaFX 31

3.1.3 Microsoft Silverlight 32

3.1.4 Srovnání platforem 32

3.2 Adobe Flex 32

4 Aplikace **34**

4.1 Požadavky na aplikaci 34

4.2 Návrh 34

4.3 Realizace 35

4.3.1 Časová osa 38

4.3.2 Problémy 39
4.4 Nástroje a testování 40
4.5 Spuštění 40

5 Závěr **42**

1 Úvod

Cílem této práce je vytvořit množinu vizualizací základních datových struktur a algoritmů. Vizualizace, by měly přiblížit fungování a implementaci jednotlivých řešení. Celé řešení, by mělo být zpracováno tak, aby bylo možné jeho použití v rámci vyúky.

Prvním úkolem, je vybrat vhodné struktury a algoritmy. Při výběru je třeba se zaměřit na složitost a praktickou použitelnost jednotlivých konstrukcí. Čtenář bude v této práci seznámen s funkcí, parametry a možnou implementací vybraných struktur a algoritmů. Dále mu bude poskytnuta vizualizace daného řešení, která umožní lepší pochopení problému. Druhým úkolem je zvolit vhodný nástroj k implementaci vizualizace. V této části vybereme vhodnou technologii, pomocí které zpracujeme vizuální část práce. Nakonec ověříme funkčnost a stabilitu výsledného řešení.

2 Algoritmy a datové struktury

Algoritmy a datové struktury zpracované v této práci byly vybrány na základě materiálů určených pro výuku předmětů KIV/PPA1 a KIV/PPA2 vyučovaných na FAV ZČU v Plzni. Jedná se o základní a široce používané konstrukce.

Z řadících algoritmů byly vybrány následující: Bubble Sort, Heap Sort, Insert Sort, Merge Sort, Quick Sort, Select Sort, Shell Sort.

Z datových struktur tyto: Spojový seznam, Zásobník, Fronta, Binární vyhledávací strom, Halda.

2.1 Řadící algoritmy

Řazení dat je jeden ze základních problémů, s kterým se dříve nebo později setká každý programátor. Zároveň, díky nenáročné implementaci většiny řešení, je toto téma vhodné k výuce algoritmizace. Jedná se o proces, při kterém řadíme prvky z množiny, kde každý prvek obsahuje data a klíč. Prvky jsou porovnávány podle klíče a definovaného kritéria. V této práci jsou pro názornost prvky celá čísla, takže data i klíč jsou jedno a to samé, kritériem pak operace, zda je klíč jednoho prvku větší nebo menší než klíč jiného prvku. V obecnějším případě, by mohl být klíč řetězec znaků a kritériem funkce, porovnávající např. délku řetězců apod.

Cílem řazení je uspořádat množinu tak, abychom s daty mohli lépe a rychleji pracovat. Pokud seřadíme množinu celých čísel od nejmenšího k největšímu, bude nalezení maxima i minima triviální záležitostí, vybráním prvního resp. posledního prvku. Při hledání extrémů v neseřazené množině s n prvky, by bylo potřeba v každém novém hledání provést n operací porovnání.

V následujícím popisu metod a jejich fungování je řazeno sestupně, od největšího prvku k nejmenšímu. Pro řazení vzestupně, je jediným zásahem do fungování metod, změna operátorů ($>$, $<$), princip zůstává stejný.

2.1.1 Insert Sort

Metoda řazení vkládáním funguje na tomto principu. Rozdělíme pole na část seřazenou a neseřazenou. V každém kroku algoritmu vezmeme první prvek z neseřazené části a vložíme ho na správné místo v seřazené části. Při porovnávání postupujeme od konce již seřazené části, pokud je vkládaný prvek větší než prvek v seřazené části, posuneme prvek ze seřazené části o jedno místo doprava. Pokud narazíme na začátek nebo porovnání nesplní podmínku, vložíme prvek do vzniklé mezery[Wro04].

Složitost algoritmu je $O(n^2)$.

Následuje ukázka řazení viz. Obrázek 2.1.

1. Na začátku je v seřazené části prvek 39. Vkládáme do seřazené části prvek 49. Vložíme prvek 49 do dočasné proměnné a porovnáváme s prvky v seřazené části tedy s 39. Platí $39 < 49$, posuneme 39 o jedno místo doprava. Tím na místě, kde se nacházelo 39 vznikne mezera. Dostali jsme se na začátek, vložíme prvek z dočasné proměnné do mezery.

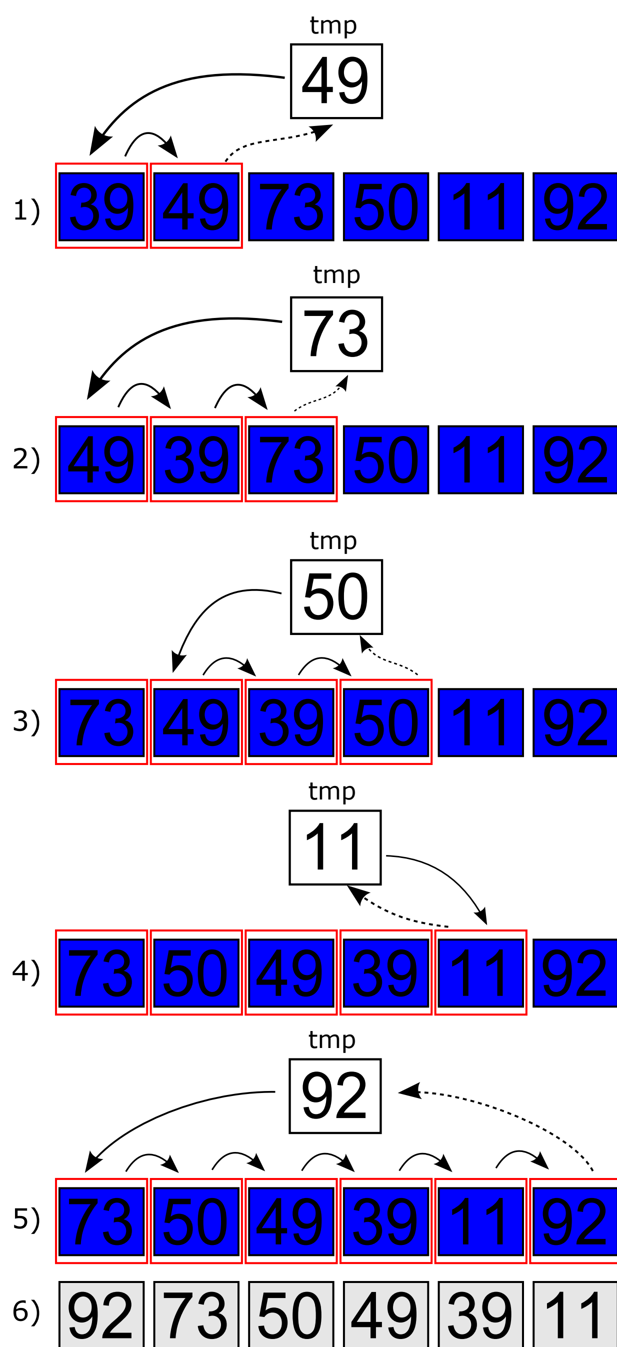
2. Vložíme do dočasné proměnné první prvek z neseřazené části 73. Postupně porovnáme se všemi prvky v seřazené části. Platí $73 > 39$, posuneme 39 o místo doprava. Platí $73 > 49$, posuneme 49 o místo doprava. Dostali jsme se na začátek, vložíme 73 z dočasné proměnné do mezery vzniklé posunutím 49.

3. Do dočasné proměnné vložíme 50 a porovnáváme se seřazenou částí. Platí $50 > 39$, posuneme 39 doprava, platí $50 > 49$, posuneme 49 doprava. Platí $50 < 73$, narazili jsme na větší prvek, vložíme 50 z dočasné proměnné do mezery.

4. Do dočasné proměnné vložíme 11. Platí $11 < 39$, 11 je menší, vložíme 11 zpět.

5. Do dočasné proměnné vložíme 92. $92 > 11$, posuneme 11 doprava, $92 > 39$, posuneme 39 doprava, $92 > 49$, posuneme 49 doprava, $92 > 50$, posuneme 50 doprava, $92 > 73$ posuneme 73 doprava, nacházíme se na začátku pole, vložíme 92 do vzniklé mezery.

6. V neseřazené části se již nenachází žádný prvek, pole je tedy seřazeno.



Obrázek 2.1: Insert sort.

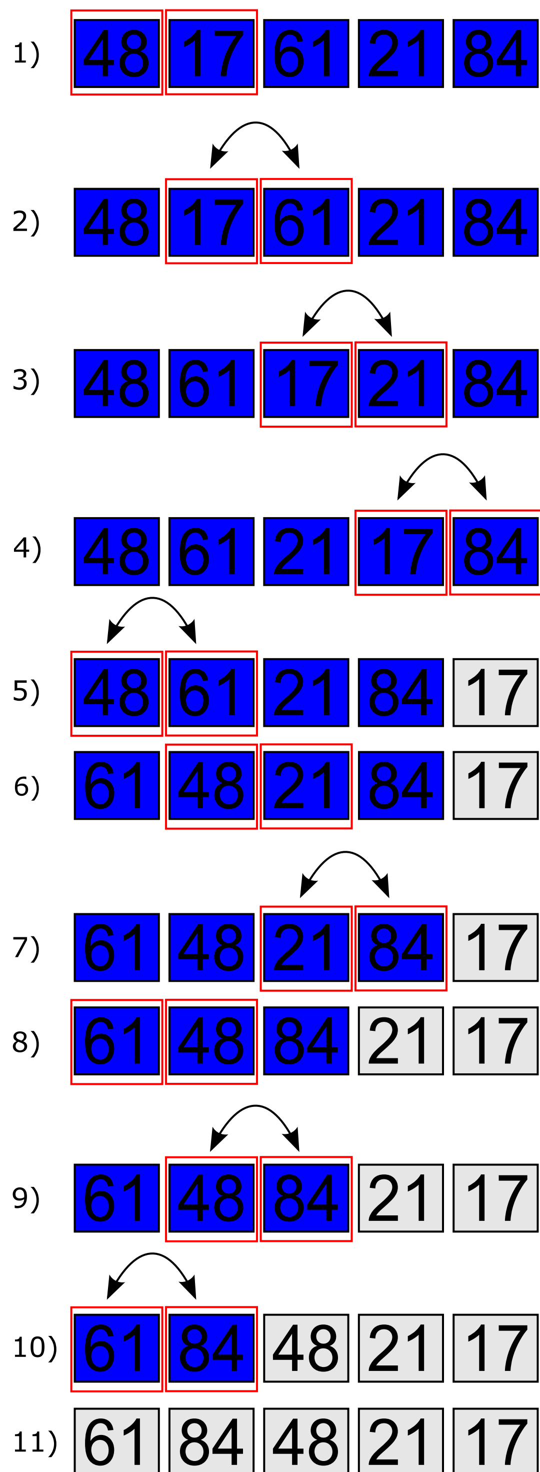
2.1.2 Bubble Sort

Princip bublinkového řazení spočívá v porovnávání dvou sousedních hodnot, pokud je prvek na indexu n menší než prvek na $n+1$, tak se prvky na daných indexech prohodí, pokud ne, nic se neprohazuje. Dál postupujeme na porovnávání prvků na indexech $n+1$ a $n+2$ a opět, pokud je $n+1$ menší než $n+2$ prohazujeme, jinak neděláme nic. Tímto postupem je zajištěno, že se při každém průchodu polem, dostane („probublá“) nejmenší prvek na jeho konečnou pozici. Po každém průchodu se zmenší počet procházených prvků o 1, protože poslední prvek zpracovávané části je na správné pozici[Wro04].

Složitost algoritmu je $O(n^2)$.

Následuje ukázka řazení viz. Obrázek 2.2.

1. Porovnáváme hodnoty na prvních dvou indexech. Platí, že hodnota na prvním indexu (48) > hodnota na druhém indexu (17), neprohazujeme nic.
2. Posuneme se o jeden prvek dál. Platí $17 < 61$, prohodíme prvky na porovnávaných indexech.
3. Opět se posuneme o prvek dál. $17 < 21$, prohodíme prvky.
4. Posun o prvek dál. $17 < 84$, prohodíme prvky. Nacházíme se na konci pole, na který se dostal nejmenší prvek. Předchozí kroky budeme opakovat od začátku, ale do pole zkráceného o jedna.
5. $48 < 61$, prohodíme prvky.
6. Posuneme se o prvek dál. $48 < 21$, neprohazujeme nic.
7. Posuneme se o prvek dál. $21 < 84$, prohodíme prvky. Nacházíme se na konci pole, zmenšíme pole o jedna a budeme pokračovat od začátku.
8. $61 > 48$, neprohazujeme nic.
9. Posuneme se o prvek dál. $48 < 84$, prohodíme, nacházíme se na konci, zmenšíme pole o jedna, pokračujeme od začátku.
10. $61 < 84$, prohodíme, nacházíme se na konci, zmenšíme pole o jedna, pokračujeme od začátku.
11. V poli zbyl poslední prvek, pole je seřazeno.



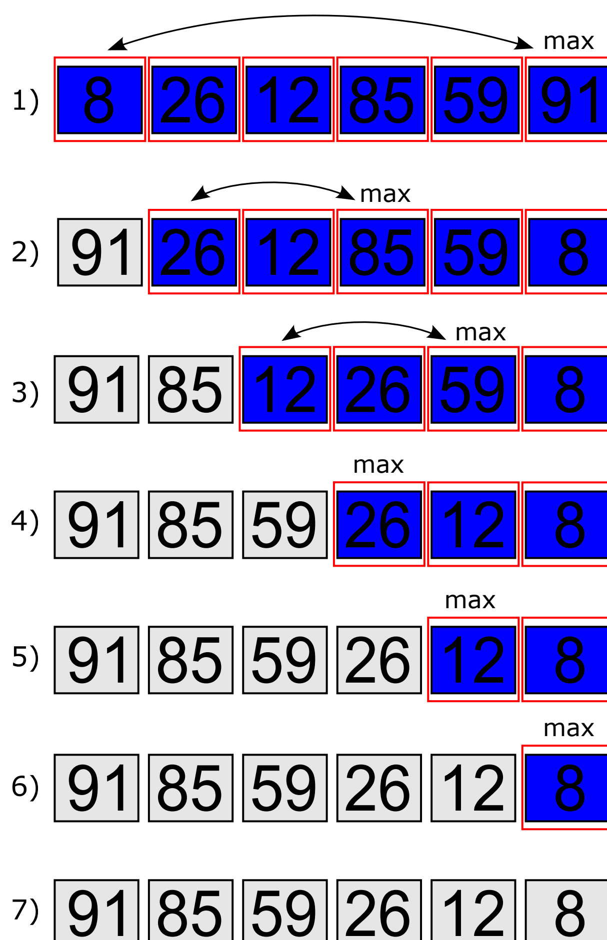
Obrázek 2.2: Bubble sort.

2.1.3 Select Sort

Algoritmus řazení výběrem funguje následovně. V každé iteraci najdeme největší prvek z dané posloupnosti a vyměníme ho s prvním prvkem. V dalším kroku zmenšíme prohledávané pole o 1. V prohledávání pokračujeme dokud je velikost pole vetší než 1. Na začátku tedy prohledáváme celé pole, najdeme největší prvek a vyměníme ho s prvkem na indexu 0. Na indexu 0 je maximum. Dál prohledáváme od indexu 1, najdeme maximum a vyměníme ho s prvkem na indexu 1. Pokračujeme v prohledávání od indexu 2 atd.

Složitost algoritmu je $O(n^2)$.

1. V právě zpracovávané části nalezneme maximum. Nalezený prvek 91 prohodíme s prvním prvkem ve zpracovávané části, což je 8. Prvek na prvním místě ve zpracovávané části se nachází na jeho konečné pozici, zmenšíme zpracovávanou část o jedna.
2. Ve zpracovávané části nalezneme maximum 85 a prohodíme ho s prvním prvkem zpracovávané části 26. Zmenšíme zpracovávanou část o jedna.
3. Nalezneme maximum 59 a prohodíme s prvním prvkem 12. Zmenšíme zpracovávanou část o jedna.
4. Nalezneme maximum 26 a prohodíme s prvním prvkem což je také 26. Zmenšíme zpracovávanou část o jedna.
5. Nalezneme maximum 12 a prohodíme s prvním prvkem což je také 12. Zmenšíme zpracovávanou část o jedna.
6. Zpracovávaná část má délku jedna, prvek je na místě.
7. Pole je seřazeno.



Obrázek 2.3: Select sort.

2.1.4 Shell Sort

Shellovo řazení je modifikace řazení vkládáním (viz. 2.1.1), kdy se porovnávají prvky, mezi kterými je vzdálenost h. Tato vzdálenost se v průběhu algoritmu zkracuje až do dosažení $h = 1$, kdy se jedná o Insert Sort, kde porovnáváme sousední prvky. Myšlenka použití vychází z toho, že pokud budou u konce řazené posloupnosti největší prvky, tak při použití obyčejného Insert Sortu musí prvek u konce projít celou posloupnost a porovnávat se s každým prvkem v cestě. Pokud zvolíme $h = 4$, bude se porovnávat s každým 4. prvkem a počet porovnání se 4-násobně sníží.

Volba vhodné posloupnosti h je pro účinnost algoritmu zásadní. Tvůrce al-

goritmu Donald Shell navrhl posloupnost $\frac{n}{2}, \frac{n}{4}, ..., 1$ kde n je počet prvků pole viz. [Laf03]. Nevýhoda při použití této posloupnosti je, že porovnání prvků na sudých a lichých pozicích je provedeno až v posledním kroku kdy $h = 1$. Alternativou je posloupnost, kterou vymyslel Donald Knut, kdy vydělíme délku pole 3, a poté vezmeme nejvyšší číslo z posloupnosti $h = 3i + 1, i = 0, 1, 2, ..$ pro které platí $h <= \frac{n}{3}$ viz. [Laf03]. Prvky této posloupnosti jsou tedy 1, 4, 13, 40, 121, 364, ... Optimální metoda zatím objevena nebyla, dobré výsledky ukazuje posloupnost Marcina Ciura 1, 4, 10, 23, 57, 132, 301, 701. Více viz. [Ciu01].

V appletu byla použita Knutova posloupnost díky výkonnosti a jednoduchosti implementace. Složitost algoritmu je $O(n^2)$.

1. Řazená posloupnost má 6 prvků platí tedy $n = 6$ a nejvyšší $h = 4$. V prvním kroku tedy začínáme od indexu $i = 4$. Hodnotu na indexu 4 přesuneme do dočasné proměnné a porovnáme s prvky, které jsou ve vzdálenosti ob 4 prvky. Platí $79 > 13$, posuneme 13 o h-prvků doprava, takže o 4 prvky. Nacházíme se na začátku pole, vložíme tedy prvek z dočasné proměnné do vzniklé mezery a posuneme se o jedno místo doprava.

2. Do dočasné proměnné vložíme 62, porovnáme s prvek 39, $62 > 39$, přesuneme 39 o 4 prvky doprava. Index o 4 menší než aktuální neexistuje, vložíme tedy prvek z dočasné proměnné do vzniklé mezery. Nacházíme se na konci pole, dopočteme nové h, $h = 1$ a budeme pokračovat od začátku s novou vzdáleností.

3. Nyní se $h = 1$, budeme postupovat stejně jako při řazení insert sortem. Do dočasné proměnné vložíme 62 a porovnáme s již seřazenou částí. $62 < 79$, vložíme 62 zpět.

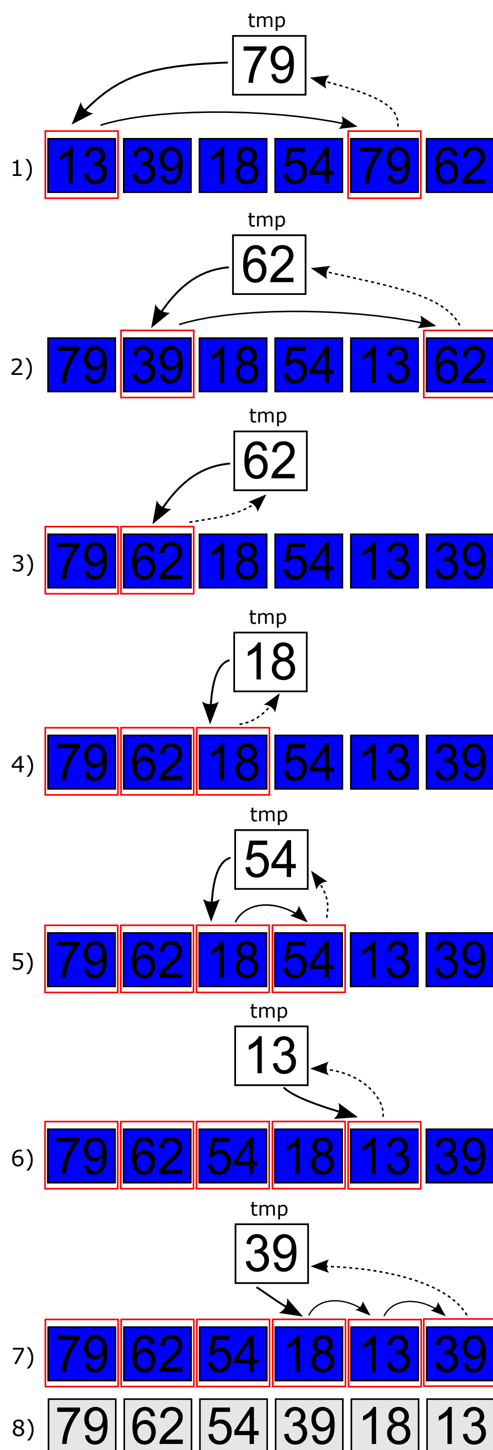
4. Do dočasné proměnné vložíme 18, $18 < 62$, vložíme 18 zpět.

5. Do dočasné proměnné vložíme 54, $54 > 18$, posuneme 18 o jeden prvek doprava, $54 < 62$, vložíme 54 do vzniklé mezery.

6. Do dočasné proměnné vložíme 13, $13 < 18$, vložíme 13 zpět.

7. Do dočasné proměnné vložíme 39, $39 > 13$, posuneme 13 doprava, $39 > 18$, posuneme 18 doprava, $39 < 54$, vložíme 39 do vzniklé mezery.

8. V neseřazené části již není žádný prvek, pole je seřazeno.



Obrázek 2.4: Shell sort.

2.1.5 Quick Sort

Metoda „rychlého řazení“ je jeden z příkladů algoritmů typu rozděl a panuj, kdy je hlavní problém rozdělen na několik podproblémů a ty jsou pak řešeny. Princip je následující. Zvolíme jeden prvek, kterému budeme říkat pivot. Rozdělíme pole tak, aby nalevo od pivota byly prvky s větší hodnotou a napravo od pivota prvky s menší hodnotou. V tuto chvíli je pivot na jeho konečné pozici. Stejnou proceduru provedeme nad oběma rozdělenými částmi. Takto pokračujeme dokud délka zpracovávané posloupnosti je větší než 1.

Důležitou částí je volba pivota. Pokud bychom ho volili tak, aby se vždy po přeuspořádání větších a menších hodnot nacházel uprostřed, došlo by k ideálnímu rozdělení pole na polovinu a bylo by tedy potřeba $log_2 n$ volání. V každé podposloupnosti, by docházelo k nejvýše n přeházení prvků. Celková složitost by byla O($nlog_2 n$). Pokud bychom volili pivota nejhorším možným způsobem, brali bychom v každé podposloupnosti největší nebo nejmenší prvek, pole by se nedělilo na poloviny, ale zmenšilo pouze o tento prvek. Docházelo by k n volání procedury dělení. V každém poli by opět docházelo k nejvýše n přeházení prvků. Celková složitost by byla O(n^2). Jako pivota často volíme levý krajní prvek řazené posloupnosti, náhodný prvek z řazené posloupnosti nebo medián prvního, prostředního a posledního prvku řazené posloupnosti [Mic12].

V závislosti na volbě pivota se složitost pohybuje v rozmezí O($log_2 n$) až O(n^2).

1. Na začátku pracujeme s celou posloupností. Jako pivot je volen první prvek zpracovávané části (42). Nyní začneme procházet pole od prvního prvku napravo od pivota. Za pivotem postupně seřadíme všechny větší prvky a za ně všechny menší prvky. Pokud narazíme na větší prvek než je pivot, přesuneme ho za poslední prvek, který je větší než pivot. $78 > 42$, žádný větší prvek za pivotem není, 78 zůstane na místě. Posuneme se na další prvek. $33 < 42$, neprohazujeme nic a posunem se na další prvek. $54 > 42$, prohodíme 54 s 33 a posuneme se na další prvek.

2. Pokračujeme dalším prvkem 21, $21 < 42$, neděláme nic a posuneme se na další prvek. $67 > 54$, prohodíme 67 s 33 a posuneme se na další prvek. $14 < 42$, neděláme nic.

3. V předchozím kroku jsme se dostali na konec zpracovávané oblasti, vy-

měníme pivota (54) s posledním prvkem, který je větší než pivot (67). V tuto chvíli jsou nalevo od pivota všechny větší a napravo všechny menší prvky. Pivot je na konečné pozici. Předchozí postup nyní aplikujeme na levou část pole.

4. Pivotem je první prvek zpracovávané posloupnosti 67. $78 > 67$, žádný větší prvek za pivotem není, 78 zůstane na místě a posuneme se na další prvek. $54 < 67$, neprohazujeme nic. Dostali jsme se na konec zpracovávané oblasti, vyměníme pivota (67) s posledním prvkem, který je větší než pivot (78). 67 je nyní na konečné pozici. Předchozí postup provedeme na levou část pole.

5. V této části se nachází pouze jeden prvek, je tedy seřazena a vracíme se o úroveň výš. V aktuální úrovni je již levá část seřazena, seřadíme pravou část.

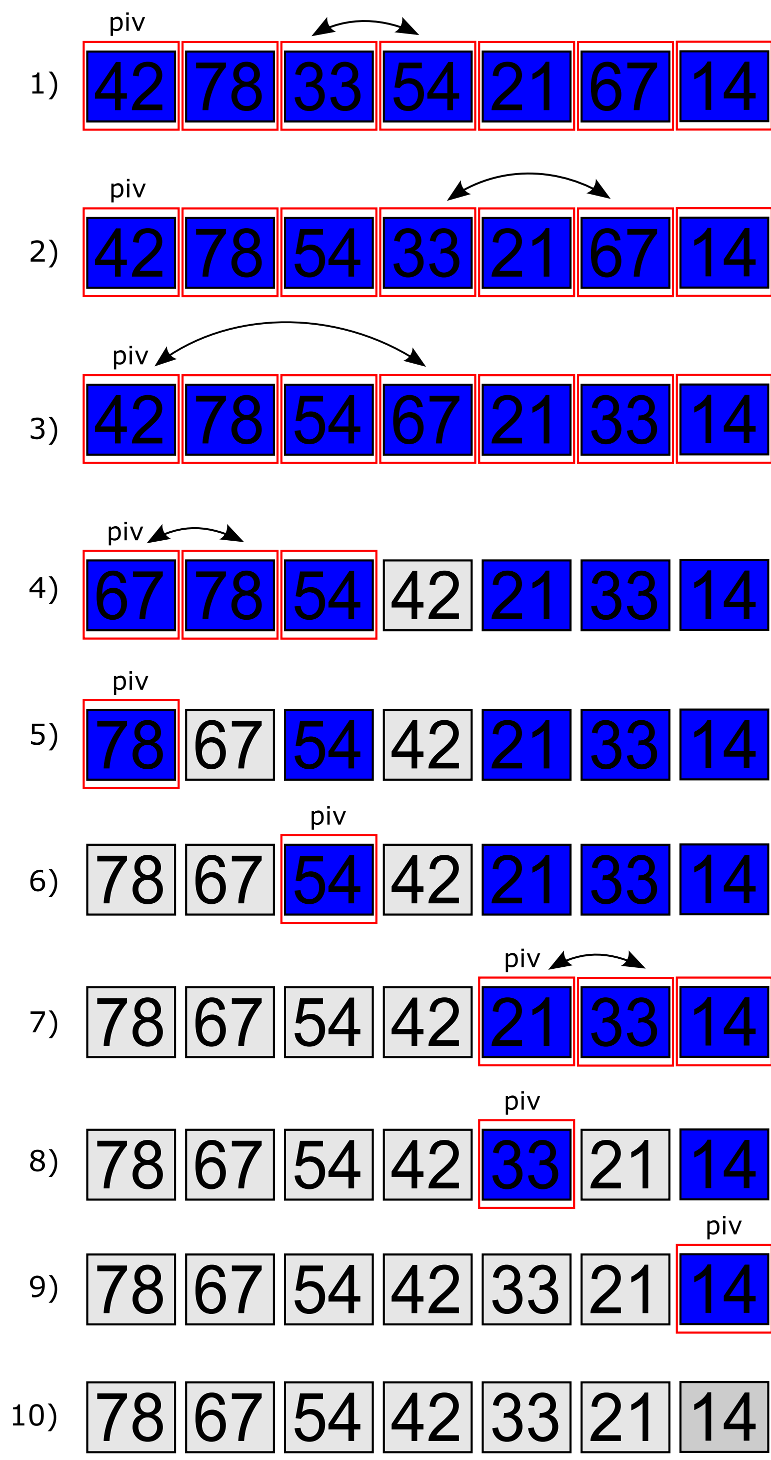
6. Opět jsme narazili na jeden prvek, vracíme se o úroveň výš. V aktuální úrovni jsou již obě části seřazeny, vracíme se o úroveň výš.

7. V aktuální úrovni je levá strana již seřazena, seřadíme pravou. Pivot je první prvek ze zpracovávané posloupnosti 21. $33 > 21$, žádný větší prvek za pivotem není, 33 zůstane na místě a posuneme se na další prvek. $14 < 21$, neprohazujeme nic. Nacházíme se na konci zpracovávané části, prohodíme 21 s 33 a seřadíme levou část.

8. Narazili jsme na jeden prvek, vracíme se o úroveň výš. V aktuální úrovni je již levá část seřazena, seřadíme pravou část.

9. Narazili jsme na jeden prvek, vracíme se o úroveň výš. V aktuální úrovni jsou již obě části seřazeny, vracíme se o úroveň výš. Takto pokračujeme až do nulté úrovně, v tuto chvíli jsou totiž seřazeny všechny části.

10. Nacházíme se v nulté úrovni, pole je seřazeno.



Obrázek 2.5: Quick sort.

2.1.6 Merge Sort

Metoda řazení slučováním je další příklad algoritmu typu rozděl a panuj. Počáteční posloupnost je rozdělena na dvě stejně velké části (pokud je lichý počet prvků je jedna část větší, to ale nehraje roli). Na každou podposloupnost aplikuje opět toto dělení až do doby, kdy je počet prvků posloupnosti rovný jedné. Poté se algoritmus začne rekurzivně vracet zpět a slučovat hodnoty ze sousedních posloupností. Slučování probíhá takto. Na vstupu máme dvě sestupně seřazené posloupnosti (při návratu z rekurze máme totiž zaručeno, že slučujeme již sestupně seřazené posloupnosti nebo posloupnosti o jednom prvku). Do pomocného pole, kopírujeme prvky z obou polí a to tak, že vybíráme vždy z té větve, kde je větší prvek. Pokud v obou polích již není žádný prvek, sloučíme obě pole a nakopírujeme do něj hodnoty z pomocného pole. Tím je procedura skončena a vracíme se o úroveň výš.

Protože je posloupnost dělena vždy na polovinu, je počet dělení roven $log_2 n$. V každé této posloupnosti provádíme až n porovnání. Celková složitost je O($nlog_2 n$).

1. Počáteční posloupnost rozdělíme na dvě části a vstoupíme do levé části. Toto budeme opakovat dokud nenarazíme na posloupnost o jednom prvku.
2. Rozdělíme posloupnost na dvě části a vstoupíme do levé části.
3. Rozdělíme posloupnost na dvě části a vstoupíme do levé části.
4. Narazili jsme na posloupnost s jedním prvkem, vracíme se o úroveň výš a vstupujeme do pravé větve.
5. Narazili jsme na posloupnost s jedním prvkem, vracíme se o úroveň výš.
6. V levé (červená) a pravé (zelená) části máme dvě seřazené posloupnosti, aplikujeme metodu slučováním kdy do pomocného pole kopírujeme vždy ten větší z prvků z obou posloupností. Posloupnost z pomocného pole překopírujeme zpět. Vracíme se o úroveň výš.
7. Levá část posloupnosti je již seřazena, vstoupíme do pravé části.
8. Narazili jsme na posloupnost s jedním prvkem, vracíme se o úroveň výš.

9. V levé a pravé části máme dvě seřazené posloupnosti, aplikujeme metodu slučování. Posloupnost z pomocného pole překopírujeme zpět. Vracíme se o úroveň výš.

10. Levá část posloupnosti je již seřazena, vstoupíme do pravé části.

11. Rozdělíme posloupnost na dvě části a vstoupíme do levé části.

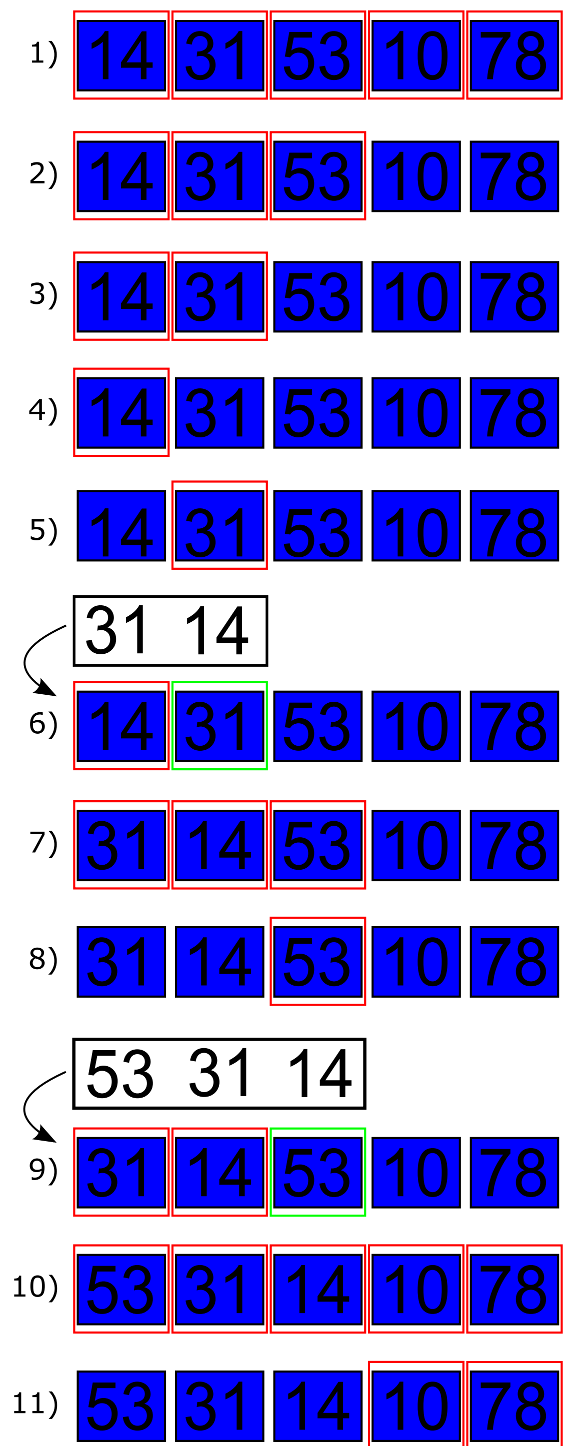
12. Narazili jsme na posloupnost s jedním prvkem, vracíme se o úroveň výš a vstupujeme do pravé větve.

13. Narazili jsme na posloupnost s jedním prvkem, vracíme se o úroveň výš.

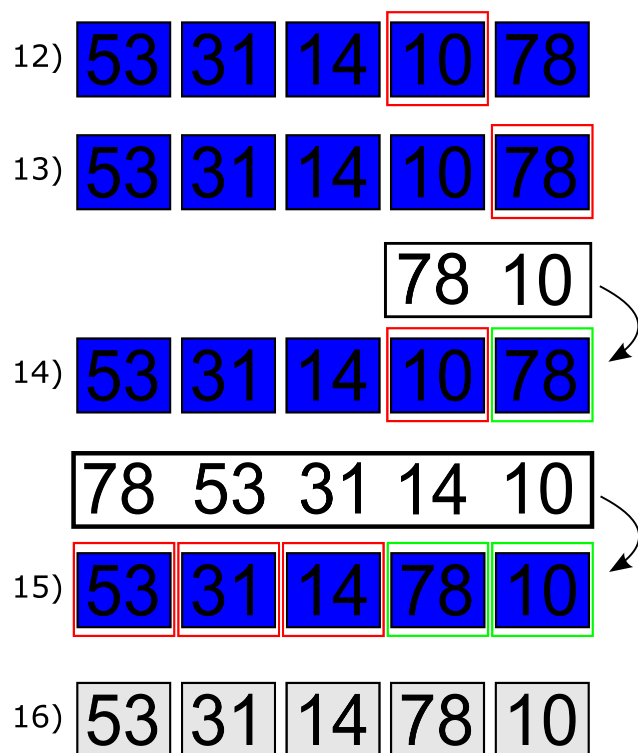
14. V levé a pravé části máme dvě seřazené posloupnosti, aplikujeme metodu slučováním . Posloupnost z pomocného pole překopírujeme zpět. Vracíme se o úroveň výš.

15. V levé a pravé části máme dvě seřazené posloupnosti, aplikujeme metodu slučováním . Posloupnost z pomocného pole překopírujeme zpět. Nacházíme se v nulté úrovni, algoritmus končí

16. Pole je seřazeno.



Obrázek 2.6: Merge sort 1. část.



Obrázek 2.7: Merge sort 2. část.

2.1.7 Heap Sort

Heap sort využívá k řazení datovou strukturu halda. Více o struktuře viz. 2.2.5. Nad daným polem hodnot k seřazení vytvoříme haldu. Vezmeme nejnižší prvek (vrchol haldy) a prohodíme ho s posledním prvkem haldy. Zmenšíme haldu o 1, na konec pole jsme přemístili nejmenší prvek, který se nyní nachází na své konečné pozici a již do haldy nepatří. Na vrcholu haldy se v tuto chvíli nachází jiný prvek a je porušena vlastnost haldy, spustíme opravnou proceduru, která obnoví vlastnost haldy. Po provedení této procedury se bude na vrcholu haldy opět nacházet nejmenší prvek. Opět ho prohodíme s posledním prvkem haldy, na předposlední místo v poli se tedy dostane druhý nejmenší prvek, a pokračujeme jako v předešlém případě. Takto pokračujeme, dokud jsou v haldě prvky.

Složitost vytvoření haldy ze zadaného pole je O($nlog_2n$), protože vkládáme n prvků a vložení jednoho prvku je se složitostí log_2n. V řadící části provádíme n vyjmutí prvků z haldy, kde složitost vyjmutí je opět log_2n. Celková složitost je tedy O($nlog_2n$).

1. Předpokládejme, že je pole indexováno od 1. Protože řadíme sestupně, budeme vytvářet min-haldu. V první fázi vytvoříme ze zadaného pole haldu. Do haldy vložíme 66 a posuneme se na další prvek.

2. Vložíme do haldy 11. Procedurou `up()` se zajistí správné umístění vkládaného prvku. 11 je na indexu 2, porovnáme s rodičem na indexu $i/2$ což je 66, $11 < 66$, prohodíme prvky.

3. Vložíme do haldy 79. Porovnáme s rodičem $79 < 11$ nic neprohazujeme.

4. Vložíme do haldy 50. $50 < 66$, prohodíme prvky, $50 > 11$, neprohazujeme.

5. Vložíme do haldy 31. $31 > 50$, prohodíme prvky, $31 > 11$, neprohazujeme.

6. Vložíme do haldy 58. $58 < 79$, prohodíme prvky, $58 > 11$, neprohazujeme.

7. Nyní máme vytvořenou haldu. Následuje řadící část, ve které vždy odebereme vrchní prvek haldy, vyměníme ho s posledním prvkem haldy, zmenšíme haldu o jedna a obnovíme její vlastnost. Tak budeme pokračovat dokud v haldě bude nějaký prvek.

8. Prohodíme vrchol (11) s posledním prvkem (79) a zmenšíme haldu o jedna. Na konci pole se nyní nachází nejmenší prvek.

9. Obnovíme vlastnost haldy procedurou `down()`, abychom dostali na vrchol nejmenší prvek ze zbylé posloupnosti. $79 > 31$, prohodíme prvky, $79 > 50$, prohodíme prvky. Žádný další potomek neexistuje, vlastnost haldy je obnovena.

10. Prohodíme vrchol (31) s posledním prvkem (79) a zmenšíme haldu o jedna.

11. Obnovíme vlastnost haldy. $79 > 50$, prohodíme prvky, $79 > 66$, prohodíme prvky. Žádný další potomek neexistuje, vlastnost haldy je obnovena.

12. Prohodíme vrchol (50) s posledním prvkem (79) a zmenšíme haldu o jedna.

13. Obnovíme vlastnost haldy. $79 > 58$, prohodíme prvky. Žádný další potomek neexistuje, vlastnost haldy je obnovena.

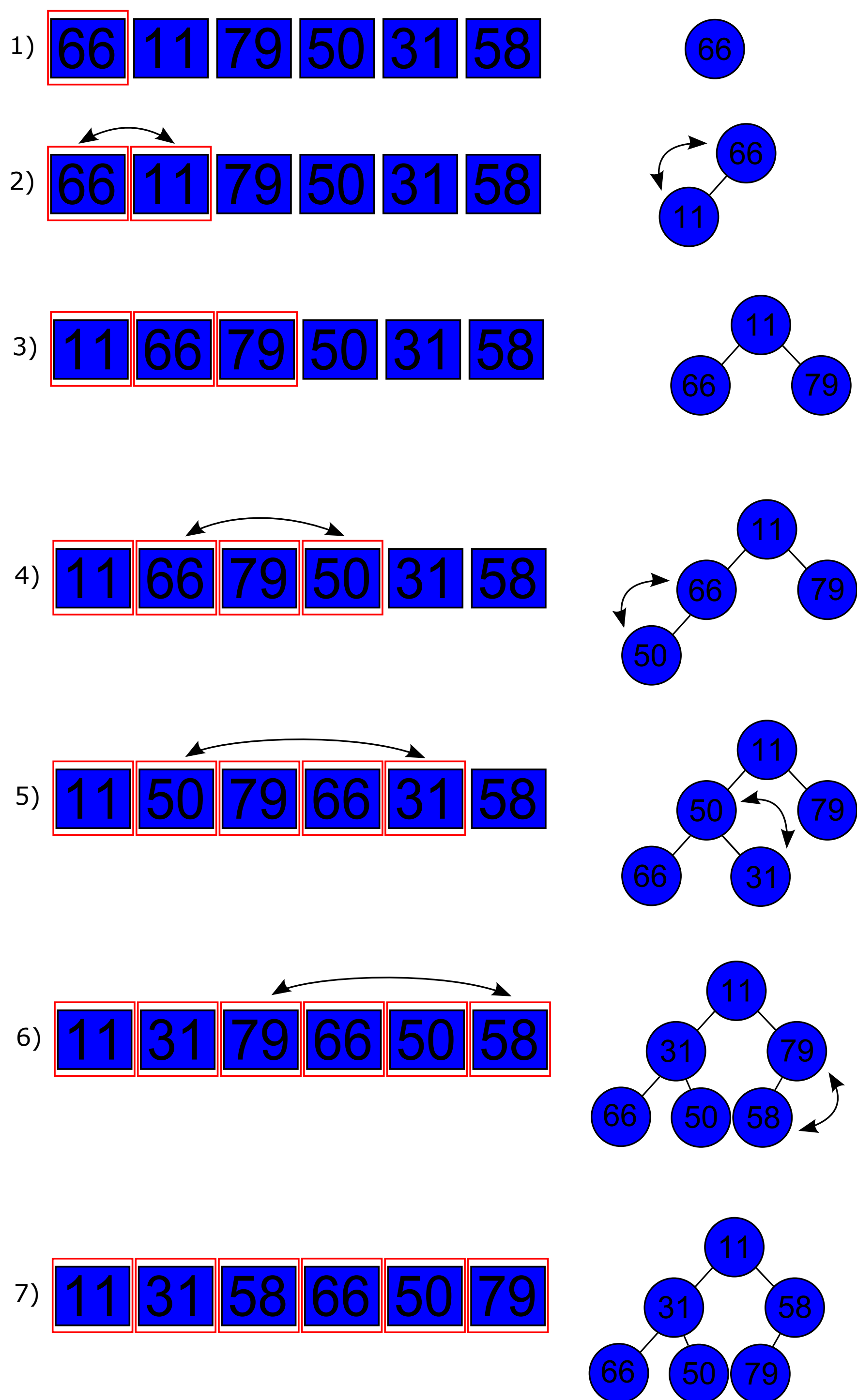
14. Prohodíme vrchol (58) s posledním prvkem (79) a zmenšíme haldu o jedna.

15. Obnovíme vlastnost haldy. $79 > 66$, prohodíme prvky. Žádný další potomek neexistuje, vlastnost haldy je obnovena.

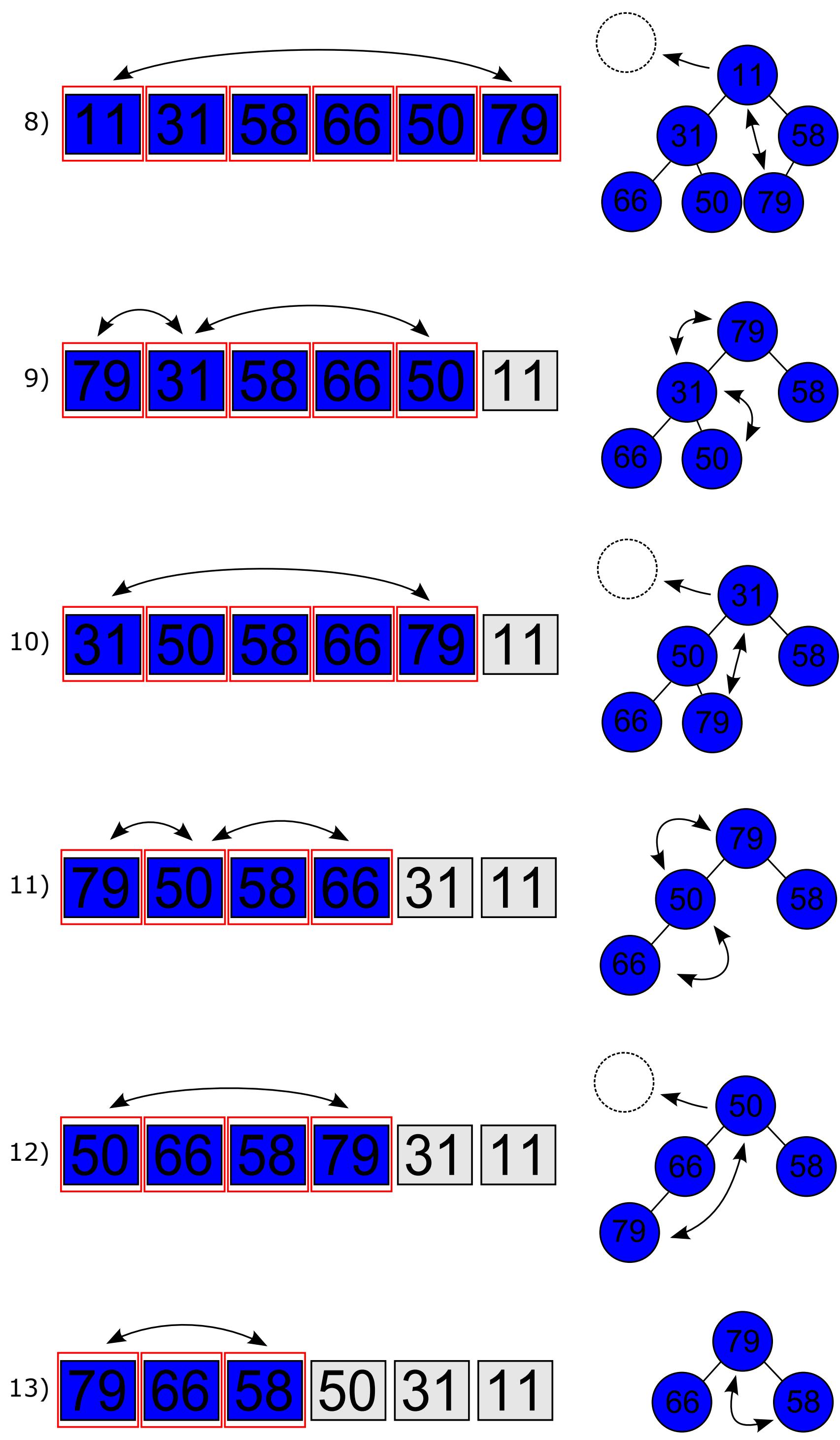
16. Prohodíme vrchol (66) s posledním prvkem (79) a zmenšíme haldu o jedna.

17. V haldě se nachází pouze jeden prvek, odebereme ho z haldy.

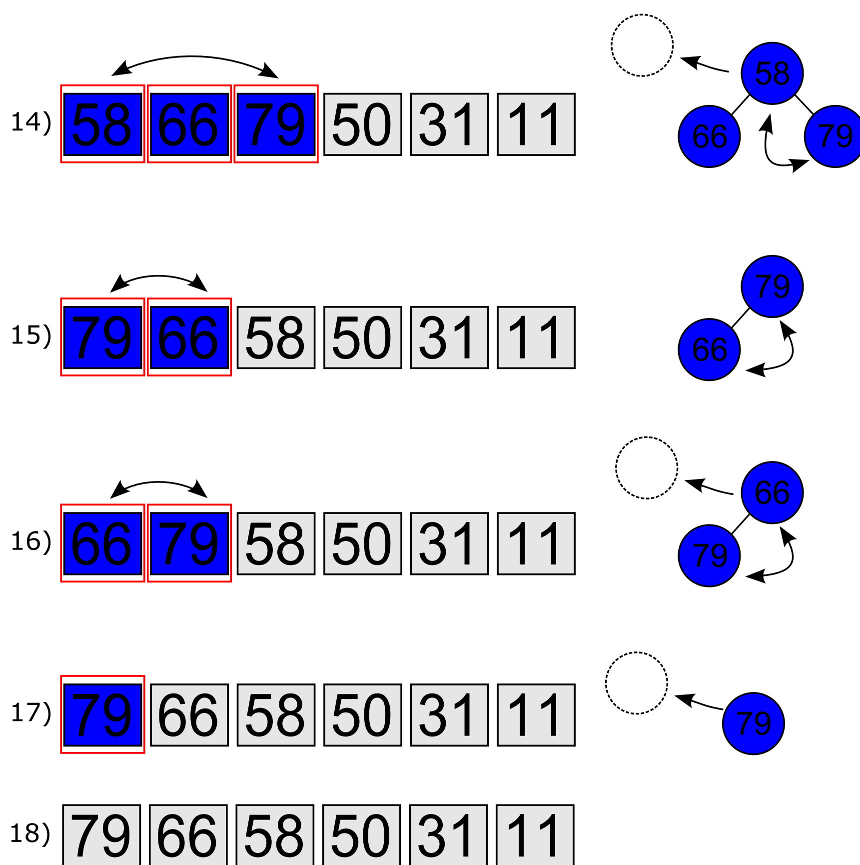
18. V tuto chvíli nejsou v haldě žádné prvky a pole je seřazeno.



Obrázek 2.8: Heap sort - vytvoření haldy.



Obrázek 2.9: Heap sort - řazení 1.

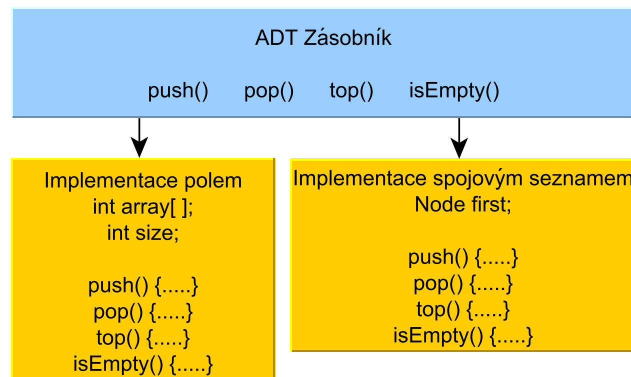


Obrázek 2.10: Heap sort - řazení 2.

2.2 Datové struktury

Datová struktura definuje způsob uložení dat a množinu operací nad daty. Používáme je tam, kde je potřeba nějakým způsobem organizovat a pracovat s daty. Ideálně tak, aby jejich organizace byla srozumitelná pro člověka a efektivní pro jejich další zpracování.

S pojmem datová struktura souvisí pojem Abstraktní datový typ (ADT). ADT definuje množinu operací, které nad danou strukturou můžeme provádět, přičemž neuvádí způsob jak daná operace funguje a jak jsou data uložena. Příkladem ADT je zásobník, kdy nad zásobníkem definujeme operace `pop()`, `push()`, `top()`, `isEmpty()`. Neuvádíme žádnou definici způsobu uložení dat, ani nezabíháme do implementační složitosti operací [Laf03]. Obrázek 4.1 ukazuje jak je definováno ADT a zároveň odděleno od implementace. Je vidět, jak je možno implementovat ADT různými způsoby, zde je naznačena implementace zásobníku polem a spojovým seznamem.



Obrázek 2.11: ADT

Informace o ADT, datových strukturách a jejich vlastnostech byly čerpány z [Laf03].

2.2.1 Spojový seznam

Spojový seznam je dynamická struktura, která se používá v případech, kdy předem neznáme počet záznamů, které budeme vkládat. Skládá se ze zá-

znamů obsahujících datovou část a část s ukazatelem na další záznam. Datovou částí může být jakýkoli objekt, nejen celé číslo.

V základním provedení mluvíme o **jednosměrném seznamu**, kde si v proměnné *first* uchováváme ukazatel na první záznam. Nevýhodou tohoto typu je možnost vstupovat do seznamu pouze přes první záznam. Pokud chceme vložit záznam na poslední místo, musíme projít od začátku přes všechny záznamy až k poslednímu. Poslední záznam poznáme podle toho, že jeho ukazatel na další záznam obsahuje *null*. Tento typ seznamu se hodí např. k implementaci zásobníku, kde vkládáme i vybíráme prvky pouze na začátku.

Dalším typem je **obousměrný seznam**. Tento typ modifikuje záznam tak, aby kromě ukazatele na další záznam obsahoval i ukazatel na předchozí záznam. Dále přidáme do definice seznamu parametr *last*, který bude ukazovat na poslední záznam. Usnadňuje se nám tím pohyb po prvcích v seznamu, kdy nemusíme do seznamu vstupovat přes parametr *first* a prohledávat jen jedním směrem, ale můžeme vstoupit přes *last* a prohledávat odzadu. Dále nám umožní vkládat a vybírat prvky na začátek i na konec se složitostí O(1), protože jsou okamžitě přístupné. Tento seznam použijeme při implementaci fronty, kdy na konec vkládáme nový záznam a ze začátku vybíráme nejstarší záznam.

K procházení seznamu se často používá konstrukce zvaná **Iterátor**. Tento objekt obsahuje proměnnou s odkazem na aktuální prvek seznamu. Můžeme si ho představit jako kurzor v textovém editoru, místo přeskakování po písmenech, ale přeskakuje po jednotlivých záznamech seznamu. Výhodou tohoto přístupu je např. možnost pokračovat v hledání daného prvku od místa, kde se nachází kurzor. Vytvoření instance iterátoru umožňuje seznam operací `getIterator()`. Typickými operacemi iterátoru jsou `next()` přesun iterátoru na další záznam, `getCurrent()` vrácení aktuálního záznamu, `reset()` nastavení iterátoru na začátek seznamu, `insertAfter()` vložení za aktuální záznam, `insertBefore()` vložení před aktuální záznam, `deleteCurrent()` smazání aktuálního záznamu.

Operace

`insertFirst(Record r)` - Vloží na první pozici nový záznam *r*.

`delete(int pos)` - Odstraní ze seznamu prvek na pozici *pos*.

`find(Record r)` - Prohledá seznam, pokud se v něm záznam *r* nachází, vrátí ho.

`isEmpty()` - Zjistí zda je seznam prázdný.

2.2.2 Zásobník

Zásobník je dynamická datová struktura. Hlavní vlastností, je možnost přistupovat k záznamům pouze „shora“. Záznamy jsou uložené tak, že nejstarší záznam je vždy vespod a nejnovější vždy na vrchu. Pokud tedy budeme chtít vybrat záznam, který je úplně vespod, musíme nejdříve vybrat všechny co jsou nad ním. Díky této vlastnosti je zásobník označován jako LIFO (last in, first out). Zásobník si udržuje proměnnou *sp* (stack pointer), ve které je uložena adresa vrcholu zásobníku (ukazatel na prvek, který je na vrcholu). Tato proměnná se aktualizuje vždy při vložení a výběru. Záznam může být jakýkoli objekt. Složitost všech operací nad zásobníkem je O(1).

Zásobník se implementuje polem nebo spojovým seznamem. Při implementaci seznamem stačí použít základní jednosměrný seznam. Ukazatel na vrchol zásobníku *sp* bude reprezentovat parametr seznamu *first*, který ukazuje na první záznam seznamu. Operace zásobníku implementujeme pomocí `insertFirst(Record r)`, `deleteFirst()`, `getFirst()` a `isEmpty()`.

Při implementaci polem budeme jednotlivé záznamy ukládat do pole `array[]`. V proměnné *sp* si budeme udržovat hodnotu indexu o jedna větší, než kde se nachází vrchol. Vkládat nový záznam budeme na pozici `array[sp]`, a vybírat `array[sp-1]`. Nevýhoda této implementace je, že při překročení kapacity pole je nutné pole zvětšit.

Operace

`push(Record r)` - Vloží do zásobníku nový záznam *r*.

`pop()` - Vybere ze zásobníku záznam na vrcholu a vrátí ho.

`top()` - Vrátí záznam na vrcholu a ponechá ho v zásobníku.

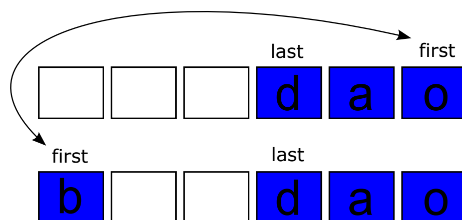
`isEmpty()` - Zjistí zda je zásobník prázdný.

2.2.3 Fronta

Jde o další dynamickou strukturu. Již podle názvu je jasné, že je zde určitá podobnost s frontou, kterou známe z každodenního života. Automobily přijíždějící na křižovatku, kde svítí červená, se postupně řadí za sebe. Jakmile padne zelená, opustí křižovatku auto, které přijelo jako první po něm druhé atd. U datové struktury se vkládané záznamy postupně řadí za sebe, tak jak přicházejí a odebírá se vždy první záznam ve frontě. Označuje se FIFO (first in, first out).

Jedna z možností implementace je spojovým seznamem, kde ze začátku seznamu vybíráme záznam a na konec seznamu přidáváme nový záznam. Je nutné použít obousměrný spojový seznam, abychom získaly přístup k poslednímu prvku a operace výběru i vkládání byly O(1). Operace fronty implementujeme pomocí `insertLast(Record r)`, `deleteFirst()`, a `isEmpty()`.

Jiný přístup je implementace polem. Záznamy se ukládají do pole `array[]`. Udržujeme si hodnoty indexů *first*, který ukazuje na začátek fronty a *last* ukazující na konec. Při přidávání záznamu je záznam uložen na `array[first]`, při odebírání je vybrán z `array[last]`. Pokud některý z indexů dosáhne konce pole, je přemístěn začátek (pokud není fronta plná) viz Obrázek 2.12.



Obrázek 2.12: Fronta - implementace polem

Operace

`insert(Record r)` - Vloží na konec fronty nový záznam *r*.

`remove()` - Vybere z fronty prvním záznam.

`isEmpty()` - Zjistí zda je fronta prázdná.

2.2.4 Binární vyhledávací strom

Jedná se strukturu připomínající obrácený strom, kořen je nahoře a strom se rozvětvuje směrem dolů. V binárním vyhledávacím stromu (dále BVS) ukládáme záznamy, které se skládají z celočíselného klíče, datové části (obsahující jakýkoli objekt) a ukazatelů na levého a pravého potomka. Záznamy uspořádáváme dle hodnoty klíče. Na vrcholu stromu se nachází záznam, který označujeme jako kořen. Vrcholy (záznamy), které nemají žádného potomka nazýváme listy.

BVS má tyto vlastnosti:

- Každý vrchol má maximálně 2 potomky, levého a pravého.
- Všechny vrcholy z levého podstromu daného vrcholu, mají menší hodnotu než tento vrchol.
- Všechny vrcholy z pravého podstromu daného vrcholu, mají větší hodnotu než tento vrchol.
- Každý vrchol kromě kořene má právě jednoho předchůdce

Při implementaci je třeba vytvořit třídu *Node*. Tato třída reprezentuje vrchol a obsahuje datovou část, klíč a ukazatel na levého a pravého potomka a ukazatel na rodiče. Ve třídě stromu poté uchováváme proměnou *root*, do které je uložen kořen stromu.

Operace

Operace předpokládají, že zpracovávaný vrchol *node*, obsahuje klíč, který je nutný k porovnání vrcholů a správnému průchodu stromem.

`insert(node)` - Vloží vrchol na správné místo ve stromu. Porovnává hodnotu klíče vrcholu *node* s hodnotou aktuálního vrcholu, pokud je menší vstoupí do jeho levé větve, pokud větší tak do pravé. Vrchol umístí, když narazí na prázdný podstrom.

`remove(node)` - Odebere ze stromu daný vrchol. Prochází strom stejným způsobem, jako v případě vkládání, dokud nenarazí na prvek *node*. Při odebírání může dojít k porušení vlastností BVS a je tedy nutné vlastnost obnovit dodatečným přemístěním příslušných vrcholů.

`search(node)` - Vyhledá zda se ve stromu nachází prvek *node*. Postupuje stejně jako při operaci vkládání, pokud narazí na prázdný podstrom, prvek ve stromu není.

Pokud potřebujeme projít všechny vrcholy, máme několik možností jak to udělat. Průchody jsou definovány rekurzivně.

`inorder(node)` - Procházíme v pořadí levý podstrom, vrchol a pravý podstrom.

`preorder(node)` - Procházíme v pořadí vrchol, potom levý a pravý podstrom.

`postorder(node)` - Procházíme v pořadí levý a pravý podstrom a potom vrchol.

2.2.5 Halda

Halda je speciální druh vyváženého binárního stromu. Vyvážený znamená, že výška obou podstromů všech vrcholů se liší maximálně o jedna. Dále v haldě platí, že každý potomek má nižší nebo stejnou hodnotu jako otec (platí v max-haldě kde je na vrcholu největší prvek, v min-haldě, kde je na vrcholu nejmenší prvek, by to bylo naopak). Díky této vlastnosti máme jistotu, že na vrcholu haldy se vždy nachází prvek s největší hodnotou. Halda se často používá pro implementaci prioritní fronty nebo k řazení haldou viz. 2.1.7

Haldu můžeme implementovat podobně jako tomu bylo u BVS, častěji se ale setkáme s implementací polem. Vrchol haldy uložíme do pole na index 1, jeho levý potomek na index $2i$ a pravý na $2i + 1$. Pokud toto aplikujeme na všechny vrcholy, zaručíme tím, že v poli nebudou vynechány žádné mezery mezi prvky. Nultý prvek pole sice zůstává prázdný.

Operace

`insert(int n)` - Vložení nového prvku do haldy. Prvek je připojen jako poslední list a je spuštěna procedura `up()` pro jeho správné zařazení.

`returnTop()` - Vybere největší prvek a vrátí ho. Na vrchol haldy dá poslední prvek a provede proceduru `down()` pro obnovení haldy.

Procedury obnovení haldy

`down()` - Obnoví vlastnost haldy, víme-li, že vrchol haldy nerespektuje uspořádání. Vrchol postupně porovnává s jeho potomky, pokud je některý z potomků větší než vrchol, vymění si pozici s tím větším. Takto pokračuje, dokud nedojde k listu nebo dokud nejsou oba potomci menší.

`up()` - Pokud je otec nového prvku menší, vymění si s otcem pozici. Tento postup opakujeme dokud se nedostaneme k vrcholu nebo dokud nenarazíme na otce, který je větší nebo roven novému prvku.

3 Vizualizace

Vizualizace je proces, kdy vytváříme obraz nebo představu něčeho, co zrovna není vidět [Viz06]. Při studiu řadících algoritmů, které byly zmíněny výše, byl v ukázkách jejich funkce znázorněn přesun prvků v poli při běhu algoritmu pomocí šipek. Již zde se jednalo o formu vizualizace. Ve skutečnosti totiž prvky nejsou uloženy v modrých čtvercích, ale v paměti v binární formě a při jejich přesunech se mění binární hodnoty na příslušných adresách. Vizualizace tedy slouží k lepšímu pochopení problému, který modeluje.

3.1 Výběr prostředku pro vizualizaci

Prvním úkolem je vybrat platformu, která umožňuje vytvořit interaktivní animace. Na webu existuje spousta videí, na kterých jsou vysvětlovány principy fungování řadících algoritmů zajímavým způsobem. V jednom skupinka lidí seřazených do řady představuje prvky pole, vepředu stojící člověk (řadící algoritmus) ukazuje, kdo si s kým má vyměnit místo. V jiném videu figurují tři lego panáčci a několik různě vysokých věží postavených z lega. Jeden z panáčků ukazuje dalším dvěma, kam mají přemístit danou věž a tuto činnost opakují dokud nejsou věže seřazeny. Video je vytvořeno ze série snímků a to tak, aby po spojení snímků vytvářelo iluzi plynulého pohybu věží a panáčků. V dalším autor provádí názornou ukázku řazení s žolíkovými kartami. Příklady viz. [You12].

Tato videa jsou sice snadno pochopitelná, ale problém nastává ve chvíli, kdy by uživatel chtěl ovlivnit vstupní data. Těmto vizualizacím chybí **interaktivita**. Pokud má být vizualizace interaktivní, musí mít uživatel možnost ovlivnit vstup. Řešením je vytvořit aplikaci, která toto umožní.

Při hledání vhodné platformy jsem se zaměřil na ty, které jsou určeny k tvorbě RIA[1], a to z těchto důvodů:

- Aplikace po nainstalování příslušného pluginu, běží přímo v internetovém prohlížeči, zároveň mohou běžet, ale i jako desktopové aplikace.

[1]Rich Internet Aplications, směr vývoje především internetových aplikací, kde je kladen důraz na uživatelskou přívětivost

- Platformy poskytují mnoho hotových komponent pro snadnou tvorbu GUI [2].
- Jsou přímo určeny k tvorbě efektivních animací a prezentací.

Jako vhodné kandidáty jsem po rešerši označil následující, **JavaFX**, **Microsoft Silverlight** a **Adobe Flash**.

3.1.1 Adobe Flash

Adobe Flash je z uváděných technologií tou nejstarší a nejrozšířenější. Většina přehrávačů hudby a videa, které nalezneme na webu, je vytvořena právě na této platformě. K vytvoření programové logiky používá skriptovací jazyk ActionScript, který je momentálně ve verzi 3.0. Tato verze již plně podporuje koncept OOP[3]. Pro spuštění aplikací je nutné jedno z běhových prostředí, Flash Player - pro běh v prohlížeči nebo Adobe Air - pro běh jako nainstalovaná dektopová aplikace. Součástí této platformy je framework Adobe Flex, který zavádí značkovací jazyk MXML pro zjednodušení definice uživatelských rozhraní a rozšiřuje soubor komponent použitelných v aplikaci. Vývojovým prostředím pro flex aplikace je Adobe Flex Builder postavený na IDE[4] Eclipse.

3.1.2 JavaFX

JavaFX je postavená na platformě Java. Java applety vytvořené pomocí Swingu[5] se poměrně často potýkají s problémy pomalého načítání, padání, „nehezkým“ GUI apod. Odstranění těchto nedostatků byl jeden z důvodů vzniku JavaFX. K definování uživatelských rozhraní se používá jazyk FXML, logika aplikace je programována v jazyce Java. Pro běh je nutné běhové prostředí Java Runtime Enviroment. Pro vývoj je možno použít, kterékoli IDE podporující jazyk Java.

[2] Graphics user interface - možnost ovládat aplikací pomocí grafického rozhraní
[3] OOP - Object oriented programing
[4] IDE - Integrated Development Environment
[5] Java Swing - Knihovna pro tvorbu GUI

3.1.3 Microsoft Silverlight

Microsoft Silverlight je technologie spadající pod platformu .NET. Způsob programování aplikací je podobný jako v předcházejících dvou případech. K tvorbě GUI se používá jazyk XAML, k vytvoření programové logiky si vývojář může vybrat, kterýkoli z jazyků podporovaný platformou .NET (C#, Visual Basic, J#, atd.). Na koncovém zařízení je opět potřeba mít nainstalován příslušný plugin. Ten je již součástí nových distribucí OS Windows, majitelům toho OS tedy tento krok odpadá.

3.1.4 Srovnání platforem

Všechny technologie poskytují velmi podobnou funkcionalitu. To co se dá vytvořit ve Flashi je možné vytvořit jak v JaveFX tak v Silverlightu. V [Ern11] je několik testů, které srovnávají výkonnostní parametry uvedených technologií. Z uvedených testů, zaměříme-li se na výsledky v testech 2D acceleration (v této práci 2D akceleraci grafiky určitě využijeme), vychází Adobe Flash jako nejlepší volba. V době tvorby této práce, jsou již k dispozici novější verze pluginů i webových prohlížečů, které v [Ern11] testovány být nemohly, předpokládejme ale, že výkonnostní trend zůstal stejný.

Rozhodnutí, které vychází z předešlých pozorování, je použít platformu Adobe Flash. Microsoft Silverlight je neprověřené technologie, která nepřináší nic převratného. Výhoda JavyFX je v možnosti používání spousty komponent přímo z Javy, v této práci, ale využijeme jen několik základních. JavaFX applety trpí neduhem, kterým je dlouhá doba spouštění. Adobe Flash je, oproti předchozím, léty prověřená a rozšířená platforma. V odkazu na [Ern11] můžeme říct, že pro tuto práci nejlepší volba i ohledně výkonu.

3.2 Adobe Flex

Pro vývoj RIA aplikací, firma Adobe nabízí Flex SDK. Tento balík obsahuje kompilátory jazyků MXML a ActionScript, knihovnu komponent a několik dalších nástrojů jako je debugger nebo ASDoc (nástroj pro automatické generování dokumentace) [Ber11]. Standardní vývojové prostředí je Adobe Flash Builder (aktuálně verze 4.5). Jedná se o komerční nástroj, který je pro aka-

demické použití zdarma. Toto prostředí obsahuje WYSIWYG[6] editor, což zpříjemňuje a zrychluje návrh GUI.

MXML jazyk vychází z jazyka XML a je určen pro tvorbu GUI. Při překladu je MXML kód nejprve převeden na ActionScript a teprve po té zkompilován[Ber11]. Kompilátor vytvoří binární soubor s příponou *.**swf**, který je spustitelný Flash Playerem (plugin zmíněný dříve).

Následující příklad ukazuje zápis vytvoření tlačítka s popisem „Start“, přidání funkce, která obsluhuje událost kliknutí a vložení tlačítka do kontejneru aplikace.

```
<s:Application>
  <s:Button label="Start" click="onClick()"/>
</s:Application>
```

To samé zapsáno v ActionScriptu.

```
var myButton:Button = new Button();
myButton.label = "Start";
myButton.addEventListener("click", onClick);
this.addElement(myButton);
```

V prvním příkladu je vidět, jak jednoduše vytvoříme instanci objektu, nastavíme jí vlastnosti a umístíme jí do hierarchie aplikace.

[6]What You See Is What You Get - možnost vytvářet GUI jednoduše přetáhnutím a umístěním komponenty z nabídky přímo na pracovní plochu aplikace

4 Aplikace

Protože se jedná o velké množství rozdílných problémů, bylo nutné a rozumné, vytvořit pro každý řadící algoritmus a datovou strukturu vlastní aplikaci (applet).

4.1 Požadavky na aplikaci

Požadavky jsou uzpůsobeny tak, aby výsledná aplikace co nejnázorněji demonstrovala funkci daného řešení.

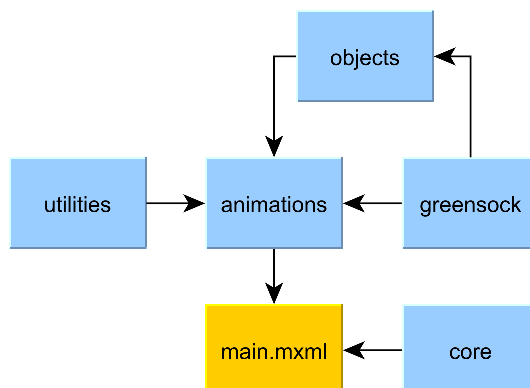
1. Možnost interaktivně pracovat s daty
2. Zobrazení abstrakce i implementace zároveň

4.2 Návrh

Struktura appletů je velmi podobná. Obrázek 4.1 znázorňuje strukturu appletů datových struktur. Obsahují jeden hlavní soubor (označený žlutě), který obsahuje definici uživatelského rozhraní, obsluhuje události vyvolané uživatelem, incializuje třídy provádějící animaci a jádro příslušné struktury.

Modré obdelníky představují soubor tříd (package), kde každý má na starost určitou činnost.

- `core` - Obsahuje implementaci datové struktury. Například u spojového seznamu se zde nachází třída, která definuje záznam a třída definující seznam.
- `animations` - Obsahuje třídy, které se starají o vykreslení, pozicování a animování jednotlivých prvků animace.
- `utilities` - Obsahuje třídu *Scroller*, která přidává možnost scrollovat se zobrazeným obsahem, pokud přesahuje viditelné meze.



Obrázek 4.1: Struktura appletů datových struktur

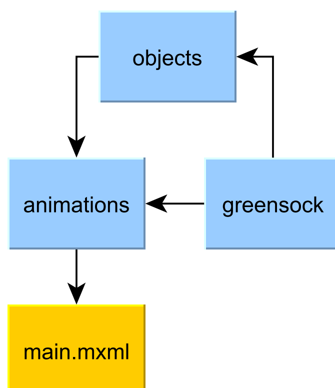
- `greensock` - Komplexní knihovna, určená k jednoduché a efektivní tvorbě animací. Autorem je Jack Doyle viz. [Gre12]
- `objects` - Definice objektů používaných v animacích. Např. položka seznamu, spojovací šipka mezi záznamy, vrchol BVS apod. Jsou zde definovány i různé animace, spojovací šipka se vytahuje z počátečního bodu, položky seznamu blikají apod.

Toto rozdělení poskytuje mnoho výhod. Chceme-li změnit, přidat nebo upravit parametr některého z objektů, stačí zasáhnout do příslušné třídy v balíku *objects*. Pokud bychom chtěli k animaci odtržení vrcholu z BVS přidat efekt, kdy odtržený vrchol odletí náhodným směrem, stačí nám zasáhnout do třídy BVS v balíku *animations*. Dodržení tohoto návrhu, nám přinese jednoduchou editaci a snadnou rozšiřitelnost appletů.

Na obrázku 4.2 je znázorněna struktura appletů řadících algoritmů. Chybí zde balík *core*, protože jádro řadícího algoritmu je vždy zakomponováno v animační části. Stejně tak chybí balík *utilities*, kde se nachází scroller. Ten u animací řadících algoritmů nenašel využití.

4.3 Realizace

Applety mají podobné GUI, kdy ve spodní části se nachází prvky k ovládání aplikace a v horní části je zobrazována animace, případně informace o stavu



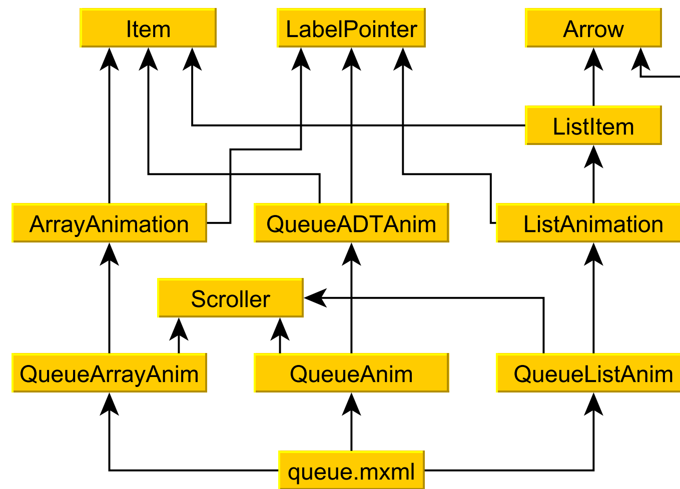
Obrázek 4.2: Struktura appletů řadících algoritmů

důležitých proměnných. GUI je vytvořeno tak, aby bylo dostatečně intuitivní. Protože je struktura appletů datových struktur velmi podobná, vezmeme jako ukázkový příklad například frontu.

V souboru `queue.mxml` je definováno rozmístění jednotlivých komponent a obsloužení událostí jako je kliknutí na tlačítko apod. Při volání jakékoli operace nad frontou, se volají příslušné metody tříd `QueueAnim, QueueArrayAnim, QueueListAnim`. V těchto třídách se definuje doba dílčích animací, přidávají a nastavují parametry důležité pro scrollování, řeší se posuny celé animace pokud přesahuje viditelný rozsah. Třídy `ArrayAnimation, QueueADTAnim, ListAnimation` obstarávají vlastní animaci. Udržují si informace o parametrech jednotlivých položek jako je pozice, velikost, barva. Jsou zde definovány animace operací (vložení, výběr, dotaz na první prvek). Třídy `Item, ListItem, Arrow, LabelPointer`, představují jednotlivé objekty, z kterých se sestavují animace. U každého jsou definovány různé vlastnosti v závislosti na daném objektu. Struktura appletu je znázorněna na obrázku 4.3.

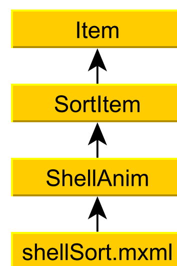
Tato struktura je společná všem appletům, některé třídy nejsou použité, koncept je, ale pro všechny stejný.

U appletů řešících problémy řazení je struktura značně jednodušší viz obrázek 4.4. Jako ukázkový příklad vezmeme Shell sort. V souboru `shellSort.mxml` je opět definováno rozmístění jednotlivých komponent, obsloužení událostí a validace dat, kdy se kontroluje správný zápis vkládaných čísel k seřazení. Třída `ShellAnim` obsahuje algoritmus Shelova řazení a řídí jeho animaci. Jedná se o hlavní třídu, v které je prováděn celý algoritmus. Třída `sortItem`



Obrázek 4.3: Použití jednotlivých tříd u appletu fronty

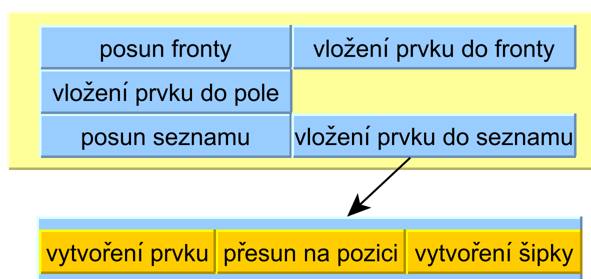
definuje jeden prvek posloupnosti, jeho parametry a animace jako je červené bliknutí, zmizení apod. Třída `Item` reprezentuje základní prvek, od kterého je odděděna `sortItem`.



Obrázek 4.4: Použití jednotlivých tříd u řadících appletů

4.3.1 Časová osa

V každém appletu se nachází hlavní časová osa, na kterou jsou přidávány jednotlivé animace. U datových appletů se nachází ve třídách na 2. úrovni odspoda viz. obrázek 4.3, u řadících appletů vždy ve třídě, kde je definována animace algoritmu (`ShellAnim`). Na obrázku 4.5 jsou na hlavní časovou osu vloženy animace, které se provádí při přidávání nového prvku do fronty. Tento případ nastane, pokud by se přidávaný prvek v animaci fronty a animaci seznamu dostal mimo zobrazovanou oblast. Zároveň se posouvají celá fronta i seznam a je přidáván prvek do animace pole, které není třeba nikam posouvat, protože po přidání zůstane v zobrazované oblasti. Po posunutí jsou přidány prvky do fronty a seznamu. Animace vložení se skládá z dalších částí jak je vyznačeno na spodní části obrázku.



Obrázek 4.5: Časová osa

U řadících appletů, se po spuštění řadícího algoritmu na časovou osu přidávají příslušné animace tak, jak algoritmus probíhá. Mimo animaci přemisťování prvků, se animuje kurzor v části, kde je zapsána implementace algoritmu. Na časovou osu jsou dále umisťována volání, která zajišťují aktualizaci důležitých proměnných v závislosti na tom, kde se animace nachází.

Jako základ pro jádro algoritmů jednotlivých datových struktur a řadících algoritmů, byly po úpravách použita řešení od Pavla Micky viz. [Mic12].

4.3.2 Problémy

Každý ze zobrazovaných objektů, je ve flashi potomek třídy `displayObject`. Mezi důležité parametry této třídy, patří vlastnosti x a y, které určují pozici objektu na scéně vzhledem k levému hornímu rohu scény [0,0]. Důležitá je také šířka a výška objektu, *width* a *height*. Při animování pohybu a změně rozměrů objektů pomocí animačních tříd z balíku `greensock`, se pozice i velikost objektů viditelně mění, ovšem jejich parametry x, y, *width* a *height* zůstávají stejné. Ke každému objektu, který se tímto způsobem animuje, bylo potřeba zavést další čtyři parametry, které se aktualizují při příslušných posunech nebo změnách velikosti.

U řadících appletů se vyskytl problém „zamrzání“ uživatelského rozhraní. Nastává ve chvíli spuštění řadící metody v které se vytváří příslušné animace a jsou přidávány na časovou osu. Je to způsobeno tím, že renderování scény je prováděno až po dokončení actionscript kódu. Pokud se tedy změní velikost nějaké komponenty, není tato změna vidět okamžitě, ale až ve chvíli kdy dostane renderer prostor a překreslí scénu. V zásadě to aplikaci nijak neubírá funkčnost, před spuštěním animace je zobrazen text „Loading“ do doby, než se animace načte a po té aplikace pracuje dál. Pokud bychom chtěli použít komponentu preloaderu, kde se ukazuje průběh načítání, komponenta nebude reagovat, protože přerendrování scény a zároveň i komponenty, se provede opět až po dokončení veškerého kódu. Jedním z řešení je použití konstrukcí popsaných na [Sen12]. Toto řešení, by vyžadovalo obtížnou dekompozici řadících algoritmů, obzvlášť těch rekurzivních. Řešením by bylo nechat provádět výpočet separátní vlákno, protože je Flash player jednovláknový není to možné. V druhé polovině letošního roku (2012), vychází nová verze Flash playeru, která by měla obsahovat podporu multithreadingu viz. [Mul12],
čímž by se problém vyřešil. Do té doby, zůstává tento problém otevřený.

Původní návrh appletu BVS, implementoval strom tak, že se při vytvoření nové úrovně stromu, automaticky zvětšili mezery mezi prvky tak, aby prvky v poslední větvi byly těsně u sebe. Docházelo ale k tomu, že při již relativně malé hloubce stromu (5 - 6) se strom roztáhl mimo viditelné meze. Problém byl vyřešen tak, že při každém přidávání prvku se přeuspořádají všechny vrcholy a hrany tak, aby bylo vytvořeno místo právě pro nový prvek. Rozvržení stromu tedy není úplně symetrické, vypadá ale o mnoho lépe než původní řešení.

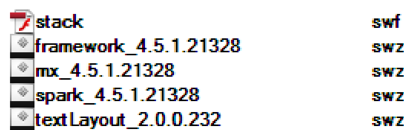
4.4 Nástroje a testování

Jako vývojové prostředí byl použit Flash Builder 4.5 postavený nad prostředím IDE Eclipse, kompilace probíhala přímo z vývojového prostředí. Vývoj probíhal na Intel Core Duo 1,66GHz, 1 GB ram s operačním systémem Windows XP SP3.

Během vývoje byl k odladění chyb používán zabudovaný debugger, společně s konstrukcí `trace()`, určenou k ladícím výpisům. U appletů datových struktur jsou reakce programu okamžité a neměl by nastat problém. U řazení závisí doba nahrávání animace na velikosti vstupní posloupnosti. Délka posloupnosti je kvůli čitelnost a výkonnosti omezena na 30 prvků. Nejhorší výsledky ukazuje animace bubble sortu, kdy při 30 prvcích trvá nahrávání kolem 22 s. U select sortu, se doba pohybuje kolem 10 s. U ostatních řazení je to do 5,5 s. Jedna z příčin může souviset s počtem vytvořených instancí v průběhu animace. Při bubble sortu je jich vytvářeno přibližně 3 krát víc než u shell sortu nebo insert sortu. Toto bylo zjištěno nástrojem pro kontrolu paměti Flash Profile, který je součástí prostředí Flash Builder. Tato doba by se tedy dala zkrátit snížením počtu instancí, ale na úkor vizuálního zpracování animace. Proto k tomuto kroku nebylo přistoupeno.

4.5 Spuštění

Pro běh jednotlivých appletů, je nutné mít nainstalován Adobe Flash Player, jak bylo zmíněno dříve. Applety jsou uloženy na přiloženém CD v adresáři applets. V každém adresáři se nachází spustitelný soubor s příponou *swf*. Název souboru reprezentuje název datové struktury nebo řadícího algoritmu. Dále je v adresáři několik podpůrných souborů s příponou *swz*, v kterých je zkompilován flex framework. Struktura adresáře je na obrázku 4.6.



Obrázek 4.6: Adresář obsahující applet.

Applety je možné zvětšit přes celou obrazovku kombinací Ctrl + F. Ovládací prvky appletu jsou vždy u spodního okraje obrazovky. Jsou řádně popsány a ovládání by mělo být intuitivní. Ukázky viz. obrázky 4.7 a 4.8.

Obrázek 4.7: Applet zásobníku.

Obrázek 4.8: Applet shell sortu.

5 Závěr

Podařilo se implementovat celou množinu datových struktur a algoritmů, která byla na začátku vybrána. Výběr technologie Flash pro vizualizaci splnil očekávání, kdy výsledné applety a animace vypadají poměrně dobře. Problém s touto technologií se vyskytl v již zmíněném problému s odezvou GUI u řadících algoritmů.

V první části práce, jsme se seznámili se základními řadícími algoritmy. V rámci některých z nich jsme zjistili, jak je možné použít rekurzi k dekompozici problému. Byl zde zmíněn důležitý pojem složitosti algoritmu. V druhé části, došlo na datové struktury. Bylo vysvětleno k čemu slouží a proč se používají, a po té následovalo seznámení s realizací několika z nich. V části vizualizace, jsme se seznámili s technologiemi, které umožňují efektivně vytvářet a pracovat s grafickými objekty. Dle požadavků jsme pro tvorbu vizualizace vybrali jednu z nich. V závislosti na technologii, byl vytvořen návrh struktury aplikace, který byl při tvorbě appletů dodržen. Realizace a implementace jednotlivých řešení se odvíjela od konkrétního problému. V závěru, byly applety otestovány a zjištěny jejich výkonnostní parametry.

Budoucí vývoj počítá s rozšířením množiny o několik dalších appletů struktur, jako jsou AVL stromy, Red-Black stromy, Hash tabulky apod. Dále by bylo vhodné pokusit se s příchodem vícevláknového Flash playeru o vyřešení problému s odezvou GUI.

Literatura

[Wro04] Piotr Wróblewski: *Algoritmy datové struktury a programovací techniky*. Computer Press, a.s., Brno, 2004. ISBN 80-251-0343-9.

[Laf03] Robert Lafore: *Data Structures and Algorithms in Java, Second Edition*. Sams Publishing, Indianapolis, 2003. ISBN 0-672-32453-9.

[Ber11] Borek Bernard: *Adobe Flex kompletní průvodce tvorbou interaktivních aplikací*. Computer Press, a.s., Brno, 2011. ISBN 978-80-251-2765-0.

[Ern11] Timo Ernst: *Performance Analysis and Acceleration for Rich Internet Application Technologies*, diploma thesis, May 2011. [online], 8.5.2012.
http://www.timo-ernst.net/wp-content/uploads/2010/09/ria-timo-ernst.pdf

[Ciu01] Marcin Ciura: *Best Increments for the Average Case of Shellsort*, August 2001. [online], 8.5.2012.
http://sun.aei.polsl.pl/ mciura/publikacje/shellsort.pdf

[Viz06] Ondřej Skutka: *Vizualizace geometrických algoritmů pro potřeby výuky*, Brno 2006. [online], 9.5.2012.
http://is.muni.cz/th/39249/fi_m/diplomka.pdf

[Gre12] Jack Doyle: *Greensock* [online], 8.5.2012. http://www.greensock.com

[Sen12] *senocular.com - Asynchronous ActionScript Execution* [online], 8.5.2012. http://www.senocular.com/flash/tutorials/asyncoperations/

[Mul12] *Adobe roadmap for the Flash runtimes* [online], 8.5.2012. http://www.adobe.com/devnet/flashplatform/whitepapers/roadmap.html

[Mic12] *www.algoritmy.net* [online], 8.5.2012. http://www.algoritmy.net/

[You12] *Vizualizace řazení* [online], 9.5.2012.
http://www.youtube.com/watch?v=INHF_5RIxTE
http://www.youtube.com/watch?v=MtcrEhrt_K0
http://www.youtube.com/watch?v=Fr0SmtN0IJM