UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

3dc8ac68-0ab2-4817-b352-48b4038e9b46

Knižnica štandardných algoritmov pre kompilátor FreePascal

2011

Vladimír Boža

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Knižnica štandardných algoritmov pre kompilátor FreePascal

(bakalárska práca)

Študijný program:	Informatika
Študijný odbor:	9.2.1 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Michal Forišek PhD.

Bratislava, 2011

Vladimír Boža

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Vladimír Boža
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

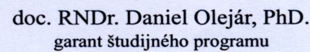
Názov : Knižnica štandardných algoritmov pre kompilátor FreePascal

Cieľ : Cieľom práce je pomocou FreePascal Generics implementovať knižnicu analogickú ku knižnici STL v jazyku C++. Táto knižnica musí používateľovi poskytovať sadu všeobecných algoritmov a dátových štruktúr, nezávisiacich na type ukladaných údajov. Perspektívne by bolo vhodné, keby sa táto knižnica následne stala súčasťou štandardnej distribúcie FreePascalu.

Vedúci : RNDr. Michal Forišek, PhD.

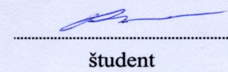
Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 27.10.2010

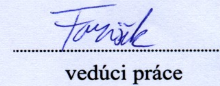


Dátum schválenia: 28.10.2010

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

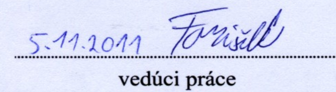


študent



vedúci práce

Dátum potvrdenia finálnej verzie práce, súhlas s jej odovzdaním (vrátane spôsobu sprístupnenia)



vedúci práce

Abstrakt

Sada štandardných algoritmov a kontajnerov je dôležitosťou súčasťou skoro každého programovacieho jazyka. V tejto práci implementujeme najdôležitejšie z nich pre kompilátor FreePascal, v ktorom sa doteraz nenachádzali.

Výsledkom práce je jedna ucelená knižnica priložená k práci a ktorá bude tiež v dohľadnej dobe dostupná spolu s FreePascalom.

Kľúčové slová: FreePascal, generics, algoritmy a dátové štruktúry

Abstract

Library implementing standard algorithms and containers is important part of almost every programming language. We implemented most common algorithms and container for FreePascal compiler, which did not have them yet.

Result of this work is library which is distributed with this work and should be part of FreePascal compiler soon.

Key words: FreePascal, generics, algorithms and data structures

Obsah

Úvod **1**

1 Rozhranie knižnice **2**
- 1.1 Kontajnery pre sekvencie 3
  - 1.1.1 TVector 3
  - 1.1.2 TStack 5
  - 1.1.3 TDeque 6
  - 1.1.4 TQueue 8
- 1.2 Usporiadané kontajnery 9
  - 1.2.1 TPriorityQueue 9
  - 1.2.2 TSet 10
  - 1.2.3 TMap 13
- 1.3 Neusporiadané kontajnery 16
  - 1.3.1 THashSet 16
  - 1.3.2 THashMap 18
- 1.4 Algoritmy pre polia 20
  - 1.4.1 Algoritmy bez znalosti poradia prvkov 20
  - 1.4.2 Algoritmy so znalosťou poradia prvkov 20

2 Implementácia **22**
- 2.1 Kontajnery pre sekvencie 22
  - 2.1.1 Pole s premenným počtom prvkov 22
  - 2.1.2 Obojsmerná fronta 22
- 2.2 Usporiadané kontajnery 22
  - 2.2.1 Prioritná fronta 22
  - 2.2.2 Usporiadaná množina 23
- 2.3 Neusporiadané kontajnery 25
- 2.4 Triedenie 25
  - 2.4.1 Quick sort 25
  - 2.4.2 Heap sort 26
  - 2.4.3 Intro sort 26
- 2.5 Testovanie implemetácie 26

3 Praktické testy **27**
3.1 Triedenie 27
3.2 Obedy 27
3.3 Asfalt 28
3.4 Básničky II. 29
3.5 Zhodnotenie testovania 30

Záver **31**

Úvod

Väčšina programovacích jazykov okrem základných príkazov obsahuje aj sadu knižníc, ktorá implementuje ďalšie užitočné funkcionality. Pre nás sú zaujímavé hlavne knižnice, ktoré implementujú základné algoritmy a kontajnery v danom jazyku (príkladom je Standard Template Library - [STL] z C++). Dôležitou vlastnosťou týchto algoritmov je ich schopnosť fungovať nezávisle na type spracovávaných dát.

V kompilátoroch pre jazyk Pascal sa doteraz takéto knižnice nevyskytovali. Jedným z hlavných dôvodov bola slabá alebo takmer žiadna podpora pre generics. V roku 2008 vyšla verzia 2.2 kompilátora FreePascal, ktorá zaviedla chabú podporu pre generics. My sme sa rozhodli niektoré základné kontejnery pre FreePascal pomocou generics implementovať a pretlačiť ich medzi knižnice distribuované spolu s FreePascalom.

Členenie tejto práce je nasledovné. V prvej kapitole prezentujeme rozhranie našej knižnice, spolu s niekoľkými jednoduchými príkladmi použitia. Táto kapitola tvorí akýsi základný manuál. V druhej kapitole prezentujeme implementačné detaily jednotlivých komponentov. V poslednej kapitole ukážeme niekoľko experimentálnych porovnaní efektívnosti našej implementácie s implementáciou STL v C++.

Kapitola 1

Rozhranie knižnice

Najprv sa pokúsime v stručnosti vysvetliť koncept generics. Niekedy je vhodné, aby trieda pri práci s dátami nepoznala dopredu typ spracovaných dát. Inými slovami miesto bežných typov (ako integer, ...) sa v triede vyskytnú vopred nešpecifikované typy. Vďaka tomu sa vieme prispôsobiť typu dát a pracovať s takmer ľubovoľným typom. Samozrejme spracovávaný typ musí spĺňať nejaké syntaktické predpoklady (napr. mal by s ním napríklad fungovať operátor priradenia, v niektorých prípadoch požadujeme aj ďalšie vlastnosti).

Na to, aby sme generics triedu vedeli použiť, ju najprv treba špecializovať, teda špecifikovať kompilátoru všetky typy, ktoré trieda potrebuje. Potom kompilátor vyrobí už použiteľnú triedu (v podstate akoby nahradil všetky výskyty nešpecifikovaných typov konkrétnymi), ktorá predstavuje nový dátový typ, s ktorým môžeme následne pracovať. Tento nový typ sa samozrejme dá použiť aj pri špecializácii ďalších typov. Takto vieme dosiahnúť napr. pole polí celých čísel.

Príklad syntaxe špecifikovania možno nájsť v príkladoch k jednotlivým kontajnerom.

Poznámka: Keďže vo FreePascale je tento koncept pomerne nový, tak je možné špecializovať len triedy. Niekedy by sme ale chceli špecializovať len funkciu (napríklad funkciu pre triedenie). Toto nie je možné priamo vykonať a preto naša knižnica obsahuje v podstate „umelé“ triedy, ktoré obsahujú tieto funkcie.

1.1 Kontajnery pre sekvencie

Všetky nasledujúce kontajnery potrebujú na špecializáciu poznať iba typ prvku, ktorý skladujú.

1.1.1 TVector

Základný kontajner, ktorý sa správa ako pole. Oproti poľu má navyše správu pamäte a schopnosť v amortizovanom konštantnom čase vkladať prvok na koniec.

Prehľad metód:

Metóda	Garancia časovej zložitosti
Popis	
`Create`	$O(1)$
Konštruktor, vytvorí prázdne pole.	
`function Size(): SizeUInt`	$O(1)$
Vráti veľkosť pola.	
`procedure PushBack(value: T)`	Amortizovane $O(1)$, niektoré operácie môžu trvať $O(N)$ času.
Vloží prvok na koniec poľa (a zväčší veľkosť poľa o 1)	
`procedure PopBack()`	$O(1)$
Vyhodí posledný prvok z poľa (a zmenší veľkosť poľa o 1). Keď je pole prázdne, nič sa nestane.	
`function IsEmpty(): boolean`	$O(1)$
Vráti true, ak je pole prázdne.	
`procedure Insert(position: SizeUInt; value: T)`	$O(N)$
Vloží prvok na miesto udané premennou position. Prvky, ktorých pozícia bola väčšia alebo rovná ako táto pozícia, sa posunú o jedno miesto dozadu.	
`procedure Erase(positine: SizeUInt; value: T)`	$O(N)$
Vymaže element z pozície position. Prvky, ktoré boli za touto pozíciou, sa posunú o 1 miesto dopredu.	
`procedure Clear`	$O(1)$
Vymaže obsah poľa (nastaví veľkosť na 0). Ale neuvoľní pamäť používanú poľom.	
`function Front: T`	$O(1)$
Vráti prvý prvok poľa. Pokiaľ je pole prázdne, tak negarantuje nič o vrátenej hodnote.	
`function Back: T`	$O(1)$
Vráti posledný prvok poľa.	
`procedure Resize(num: SizeUInt)`	$O(N)$
Zmení veľkosť poľa na num. Negarantuje nič o hodnote novo vzniknutých prvkov.	

`procedure Reserve(num: SizeUInt)`	$O(N)$
Alokuje pre pole pamäť veľkosť num. Vhodné pokiaľ chceme veľa krát volať PushBack a ušetriť čas na realokáciu poľa.	

`property item[i: SizeUInt]: T; default;`	$O(1)$
Property na prístup k i-tému prvku poľa (indexujeme o 0). Umožňuje používanie len pomocou hranatých zátvoriek (keďže je default property).	

`property mutable[i: SizeUInt]: PT;`	$O(1)$
Vráti pointer na i-ty prvok. Užitočné, keď skladujeme zložené typy, ktoré sa pri priradení kopírujú (record, object) a chceme meniť len časť ich vlastností.	

Príklad použitia:

```
uses gvector;

type TVectorlli = specialize TVector<longint>;

var Buffer:TVectorlli; i:longint;

begin
  Buffer := TVectorlli.Create;
  {Push 5 elements at the end of array}
  for i:=1 to 5 do
    Buffer.PushBack(i);
  {change 3rd element to 47}
  Buffer[2] := 47;
  {pop last element}
  Buffer.PopBack;
  {print all elements}
  for i:=0 to Buffer.Size-1 do
    writeln(Buffer[i]);

  Buffer.Destroy;
end.
```

1.1.2 TStack

Zásobník, dovoľuje prezerať, pridávať a odoberať prvok z vrchu zásobníka.

Prehľad metód:

Metóda	Garancia časovej zložitosti
Popis	
`Create`	$O(1)$
Konštruktor. Vyrobí prázdny zásobník.	
`function Size(): SizeUInt`	$O(1)$
Vráti počet prvkov v zásobníku.	
`procedure Push(value: T)`	Amortizovane $O(1)$, niektoré operácie môžu trvať $O(N)$ času.
Vloží prvok na vrch zásobníka.	
`procedure Pop()`	$O(1)$
Zmaže prvok z vrchu zásobníka. Ak je zásobník prázdny, nerobí nič.	
`function IsEmpty(): boolean`	$O(1)$
Vráti true, ak je zásobník prázdny.	
`function Top: T`	$O(1)$
Vráti vrchný prvok zo zásobníka.	

Príklad použitia:

```
uses gstack;

type stacklli = specialize TStack<longint>;

var data:stacklli; i:longint;

begin
  data:=stacklli.Create;
  for i:=1 to 10 do
    data.Push(10*i);
  while not data.IsEmpty do begin
    writeln(data.Top);
    data.Pop;
  end;

  data.Destroy;
end.
```

1.1.3 TDeque

Obojsmerná fronta. Alebo aj pole, do ktorého je možné vkladať a odoberať prvky z oboch strán v amortizovanom konštantnom čase.

Metóda	Garancia časovej zložitosti
Popis	
`Create`	$O(1)$
Konštruktor, vytvorí prázdne pole.	
`function Size(): SizeUInt`	$O(1)$
Vráti veľkosť pola.	
`procedure PushBack(value: T)`	Amortizovane $O(1)$, niektoré operácie môžu trvať $O(N)$ času.
Vloží prvok na koniec poľa (a zväčší veľkosť poľa o 1)	
`procedure PopBack()`	$O(1)$
Vyhodí posledný prvok z poľa (a zmenší veľkosť poľa o 1). Keď je pole prázdne, nič sa nestane.	
`procedure PushFront(value: T)`	Rovnako ako PushBack.
Vloží prvok na začiatok poľa (a zväčší veľkosť poľa o 1)	
`procedure PopFront()`	$O(1)$
Vyhodí prvý prvok z poľa (a zmenší veľkosť poľa o 1). Keď je pole prázdne, nič sa nestane.	
`function IsEmpty(): boolean`	$O(1)$
Vráti true, ak je pole prázdne.	
`procedure Insert(position: SizeUInt; value: T)`	$O(N)$
Vloží prvok na miesto udané premennou position. Prvky, ktorých pozícia bola väčšia alebo rovná ako táto pozícia, sa posunú o jedno miesto dozadu.	
`procedure Erase(positine: SizeUInt; value: T)`	$O(N)$
Vymaže element z pozície position. Prvky, ktoré boli za touto pozíciou, sa posunú o 1 miesto dopredu.	
`procedure Clear`	$O(1)$
Vymaže obsah poľa (nastaví veľkosť na 0). Ale neuvoľní pamäť používanú poľom.	
`function Front: T`	$O(1)$
Vráti prvý prvok poľa. Pokiaľ je pole prázdne, tak negarantuje nič o vrátenej hodnote.	
`function Back: T`	$O(1)$
Vráti posledný prvok poľa.	
`procedure Resize(num: SizeUInt)`	$O(N)$
Zmení veľkosť poľa na num. Negarantuje nič o hodnote novo vzniknutých prvkov.	

`procedure Reserve(num: SizeUInt)`	$O(N)$
Alokuje pre pole pamäť veľkosť num. Vhodné pokiaľ chceme veľa krát volať PushBack a ušetriť čas na realokáciu poľa.	

`property item[i: SizeUInt]: T; default;`	$O(1)$
Property na prístup k i-tému prvku poľa (indexujeme o 0). Umožňuje používanie len pomocou hranatých zátvoriek (keďže je default property).	

`property mutable[i: SizeUInt]: T;`	$O(1)$
Vráti pointer na i-ty prvok. Užitočné, keď skladujeme zložené typy, ktoré sa pri priradení kopírujú (record, object) a chceme meniť len časť ich vlastností.	

Príklad použitia:

```
uses gdeque;

type TDequelli = specialize TDeque<longint>;

var Buffer:TDequelli; i:longint;

begin
  Buffer := TDequelli.Create;
  {Push 5 elements at the end of array}
  for i:=1 to 5 do
    Buffer.PushBack(i);
  {change 3rd element to 47}
  Buffer[2] := 47;
  {pop last element}
  Buffer.PopBack;
  {push 3 element to front}
  for i:=1 to 3 do
    Buffer.PushFront(i*10);
  {print all elements}
  for i:=0 to Buffer.Size-1 do
    writeln(Buffer[i]);

  Buffer.Destroy;
end.
```

1.1.4 TQueue

Fronta. Dá sa pridávať prvok na koniec, pozerať a odoberať prvok na začiatku.

Prehľad metód:

Metóda	Garancia časovej zložitosti
Popis	
`Create`	$O(1)$
Konštruktor. Vyrobí prázdnu frontu.	
`function Size(): SizeUInt`	$O(1)$
Vráti počet prvkov vo fronte.	
`procedure Push(value: T)`	Amortizovane $O(1)$, niektoré operácie môžu trvať $O(N)$ času.
Vloží prvok na koniec fronty.	
`procedure Pop()`	$O(1)$
Zmaže prvok zo začiatku fronty. Ak je fronta prázdna nerobí nič.	
`function IsEmpty(): boolean`	$O(1)$
Vráti true, ak je fronta prázdna.	
`function Front: T`	$O(1)$
Vráti prvý prvok fronty.	

Príklad použitia:

```
uses gqueue;

type queuelli = specialize TQueue<longint>;

var data:queuelli; i:longint;

begin
  data:=queuelli.Create;
  for i:=1 to 10 do
    data.Push(10*i);
  while not data.IsEmpty do begin
    writeln(data.Front);
    data.Pop;
  end;

  data.Destroy;
end.
```

1.2 Usporiadané kontajnery

Nasledovné kontajnery potrebujú okrem prvku, ktorý uskladňujú, poznať aj usporiadanie týchto prvkov. Na usporiadanie treba definovať triedu, ktorá obsahuje class metódu (v C++ sa to nazýva statická metóda, teda metóda, ktorú je možné volať aj bez inštanciovania triedy) c, ktorá akceptuje 2 parametre a, b - porovnávané prvky a vracia boolean. A vracia true, keď prvok a je v usporiadaní pred prvkom b (obdoba operátora $<$).

Príklad takejto porovnávacej triedy je možné vidieť v príklade pri nasledovnom kontajneri.

1.2.1 TPriorityQueue

Prioritná fronta. Umožňuje vkladanie prvkov a pozeranie a výber najväčšieho prvku z fronty.

Prehľad metód:

Metóda	Garancia časovej zložitosti
Popis	
`Create`	$O(1)$
Konštruktor. Vyrobí prázdnu frontu.	
`function Size(): SizeUInt`	$O(1)$
Vráti počet prvkov vo fronte.	
`procedure Push(value: T)`	Amortizovane $O(\log_2 N)$, niektoré operácie môžu trvať $O(N)$ času.
Vloží prvok do fronty.	
`procedure Pop()`	$O(\log_2 N)$
Zmaže najväčší prvok z fronty. Ak je fronta prázdna, nerobí nič.	
`function IsEmpty(): boolean`	$O(1)$
Vráti true, ak je fronta prázdna.	
`function Top: T`	$O(1)$
Vráti najväčší prvok z fronty.	

Príklad použitia:

```
{$mode objfpc}

uses gpriorityqueue;

type
  lesslli = class
    public
    class function c(a,b: longint):boolean;inline;
```

```
  end;

class function lesslli.c(a,b: longint):boolean;inline;
begin
  c:=a<b;
end;

type priorityqueuelli = specialize TPriorityQueue<longint, lesslli>;

var data:priorityqueuelli; i:longint;

begin
  data:=priorityqueuelli.Create;
  for i:=1 to 10 do
    data.Push(random(1000));
  while not data.IsEmpty do begin
    writeln(data.Top);
    data.Pop;
  end;

  data.Destroy;
end.
```

1.2.2 TSet

Usporiadaná množina. Umožňuje vkladanie, zmazávanie a hľadanie prvkov. Každý prvok sa v množine môže vyskytovať maximálne raz.

Prehľad metód:

Metóda	Garancia časovej zložitosti
Popis	
`Create`	$O(1)$
Konštruktor. Vyrobí prázdnu množinu.	
`function Size(): SizeUInt`	$O(1)$
Vráti počet prvkov v množine.	
`procedure Insert(value: T)`	$O(\log_2 N)$
Vloží prvok do množiny, ak sa tam vkladaný prvok už nachádza, nestane sa nič.	
`function InsertAndGetIterator (value: T):TIterator`	$O(\log_2 N)$
Vloží prvok do množiny, ak sa tam vkladaný prvok už nachádza, nestane sa nič. Zároveň vráti iterátor, ktorý ukazuje na miesto, kde sa prvok po vložení nachádza. Podrobný popis metód iterátora sa nachádza nižšie.	

`procedure Delete(value: T)`	$O(\log_2 N)$
Vymaže prvok z množiny, ak sa tam nenachádza, nič sa nestane.	
`function Find(value: T):TIterator`	$O(\log_2 N)$
Hľadá prvok v množine. Ak sa tam nenachádza, vracia nil. Ináč vráti iterátor, ktorý ukazuje na prvok.	
`function FindLess(value: T):TIterator`	$O(\log_2 N)$
Hľadá najväčší prvok v množine, ktorý je menší ako value. Ak sa tam nenachádza, vracia nil. Ináč vráti iterátor, ktorý ukazuje na prvok.	
`function FindLessEqual(value: T):TIterator`	$O(\log_2 N)$
Hľadá najväčší prvok v množine, ktorý je menší alebo rovný ako value. Ak sa tam nenachádza, vracia nil. Ináč vráti iterátor, ktorý ukazuje na prvok.	
`function FindGreater(value: T):TIterator`	$O(\log_2 N)$
Hľadá najmenší prvok v množine, ktorý je väčší ako value. Ak sa tam nenachádza, vracia nil. Ináč vráti iterátor, ktorý ukazuje na prvok.	
`function FindGreaterEqual(value: T):TIterator`	$O(\log_2 N)$
Hľadá najmenší prvok v množine, ktorý je väčší alebo rovný ako value. Ak sa tam nenachádza, vracia nil. Ináč vráti iterátor, ktorý ukazuje na prvok.	
`function Min:TIterator`	$O(\log_2 N)$
Vráti iterátor ukazujúci na najmenší prvok z množiny. Ak je množina prázdna, vracia nil.	
`function Max:TIterator`	$O(\log_2 N)$
Vráti iterátor ukazujúci na najväčší prvok z množiny. Ak je množina prázdna, vracia nil.	
`function IsEmpty(): boolean`	$O(1)$
Vráti true, ak je množina prázdna.	

V zozname metód sa vyskytla zmienka o type TIterator. Je to vnorený typ v TSet. Slúži hlavne na prechádzanie cez prvky v množine. Iterátor zostáva platným pri akýchkoľvek operáciách nad množinou (okrem vymazania prvku, na ktorý ukazuje iterátor). Obsahuje nasledujúce metódy:

Metóda	Garancia časovej zložitosti
Popis	
`function Next:boolean`	$O(\log_2 N)$ v najhoršom prípade, ale prechod celej množiny trvá čas $O(N)$.
Posunie iterátor na najbližší väčší prvok v množine. Ak sa to podarilo vráti true, ak iterátor už ukazuje na najväčší prvok, vráti false.	

function Prev:boolean	$O(\log_2 N)$ v najhoršom prípade, ale prechod celej množiny trvá čas $O(N)$.
Posunie iterátor na najbližší menší prvok v množine. Ak sa to podarilo vráti true, ak iterátor už ukazauje na najmenší prvok, vráti false.	

property Data:T	$O(1)$
Property, pomocou ktorej sa dá čítať prvok, na ktorý iterátor ukazuje.	

Príklad použitia:

```
uses gset, gutil;

type lesslli=specialize TLess<longint>;
     setlli=specialize TSet<longint, lesslli>;

var data:setlli; i:longint; iterator:setlli.TIterator;

begin
  data:=setlli.Create;

  for i:=0 to 10 do
    data.insert(i);

  {Iteration through elements}
  iterator:=data.Min;
  repeat
    writeln(iterator.Data);
  until not iterator.next;
  {Don't forget to destroy iterator}
  iterator.Destroy;

  iterator := data.FindLess(7);
  writeln(iterator.Data);
  iterator.Destroy;

  data.Destroy;
end.
```

1.2.3 TMap

Asociatívne pole. Uchováva usporiadané dvojice (kľúč, hodnota) usporiadané podľa hodnoty kľúča. Zároveň platí, že nemôžu existovať dva prvky s rovnakým kľúčom. Na špecializáciu potrebuje tri argumenty - typ kľúča, typ ukladanej hodnoty a porovnávaciu triedu pre kľúče.

Prehľad metód:

Metóda	Garancia časovej zložitosti
Popis	
`Create`	$O(1)$
Konštruktor. Vyrobí prázdne pole.	
`function Size(): SizeUInt`	$O(1)$
Vráti počet prvkov v poli.	
`procedure Insert(key:TKey; value: TValue)`	$O(\log_2 N)$
Vloží danú dvojicu do poľa. Ak sa už prvok s daným kľúčom v poli nachádza, hodnota sa prepíše.	
`function InsertAndGetIterator (key:TKey; value: TValue):TIterator`	$O(\log_2 N)$
To isté ako Insert, ale vracia aj iterátor ukazujúci na vložený prvok.	
`procedure Delete(key: TKey)`	$O(\log_2 N)$
Vymaže prvok z poľa, ak sa tam nenachádza nič sa nestane.	
`function Find(key:TKey):TIterator`	$O(\log_2 N)$
Hľadá prvok v poli s daným kľúčom. Ak sa tam nenachádza, vracia nil. Ináč vráti iterátor, ktorý ukazuje na prvok.	
`function FindLess(key:TKey):TIterator`	$O(\log_2 N)$
Hľadá najväčší prvok v poli, ktorého kľúč je menší ako key. Ak sa tam nenachádza, vracia nil. Ináč vráti iterátor, ktorý ukazuje na prvok.	
`function FindLessEqual(key:TKey):TIterator`	$O(\log_2 N)$
Hľadá najväčší prvok v poli, ktorého kľúč je menší alebo rovný ako key. Ak sa tam nenachádza, vracia nil. Ináč vráti iterátor, ktorý ukazuje na prvok.	
`function FindGreater(key:TKey):TIterator`	$O(\log_2 N)$
Hľadá najväčší prvok v poli, ktorého kľúč je väčší ako key. Ak sa tam nenachádza, vracia nil. Ináč vráti iterátor, ktorý ukazuje na prvok.	
`function FindGreaterEqual(key:TKey):TIterator`	$O(\log_2 N)$
Hľadá najväčší prvok v poli, ktorého kľúč je väčší alebo rovný ako key. Ak sa tam nenachádza, vracia nil. Ináč vráti iterátor, ktorý ukazuje na prvok.	
`function Min:TIterator`	$O(\log_2 N)$
Vráti iterátor ukazujúci na prvok s najmenším kľúčom. Ak je pole prázdne, vracia nil.	
`function Max:TIterator`	$O(\log_2 N)$

Vráti iterátor ukazujúci na prvok s najväčším kľúčom. Ak je pole prázdne, vracia nil.	
`function IsEmpty(): boolean`	$O(1)$
Vráti true, ak je pole prázdne.	
`property item[i: Key]: TValue; default;`	O(ln N)
Property na prístup k prvkov pola. Pri čítaní vráti hodnotu asociovanú s daným kľúčom. Pri zapisovaní prepíše danú hodnotu pri kľúči a ak kľúč neexistuje vytvorí novú hodnotu. Pri pokuse o čítanie neexistujúcej hodnoty spadne. Je to default property, takže na použitie stačia hranaté zátvorky.	

V zozname metód sa vyskytla zmienka o type TIterator. Je to vnorený typ v TMap. Slúži hlavne na prechádzanie cez prvky v poli a zmenu hodnoty priradenej ku kľúču. Iterátor zostáva platným pri akýchkoľvek operáciách nad poľom (okrem vymazania prvku, na ktorý ukazuje iterátor). Obsahuje nasledujúce metódy:

Metóda	Garancia časovej zložitosti
Popis	
`function Next:boolean`	$O(\log_2 N)$ v najhoršom prípade, ale prechod celého poľa trvá čas $O(N)$.
Posunie iterátor na najbližší prvok s väčším kľúčom. Ak sa to podarilo vráti true, ak iterátor už ukazuje na najväčší kľúč, vráti false.	
`function Prev:boolean`	$O(\log_2 N)$ v najhoršom prípade, ale prechod celého poľa trvá čas $O(N)$.
Posunie iterátor na najbližší prvok s väčším kľúčom. Ak sa to podarilo vráti true, ak iterátor už ukazuje na najmenší kľúč, vráti false.	
`property Key:TKey`	$O(1)$
Property, pomocou ktorej sa dá čítať kľúč, na ktorý iterátor ukazuje.	
`property Value:TValue`	$O(1)$
Property, pomocou ktorej sa dá čítať a prepisovať hodnota, na ktorú iterátor ukazuje.	
`property MutableValue:PValue`	$O(1)$
Vráti pointer na uloženú hodnotu. Užitočné pri prepisovaní hodnoť v uložených recordoch a objektoch.	

Príklad použitia:

```
uses gmap, gutil;

type lesslli=specialize TLess<longint>;
     maplli=specialize TMap<longint, longint, lesslli>;
```

```
var data:maplli; i:longint; iterator:maplli.TIterator;

begin
  data:=maplli.Create;

  for i:=0 to 10 do
    data[i]:=10*i;

  writeln(data[7]);
  data[7] := 42;

  {Iteration through elements}
  iterator:=data.Min;
  repeat
    writeln(iterator.Key, ' ', iterator.Value);
    iterator.Value := 47;
  until not iterator.next;
  iterator.Destroy;

  iterator := data.FindLess(7);
  writeln(iterator.Value);
  iterator.Destroy;

  data.Destroy;
end.
```

1.3 Neusporiadané kontajnery

Tieto kontajnery sú vnútorne implementované ako hašovacie tabuľky. Preto okrem znalosti typu prvku, potrebujú hašovaciu funkciu. Hašovaciu funkciu definujeme ako class funkciu hash, ktorá akceptuje dva argumenty, prvým je daný prvok a druhým je číslo N. Táto funkcia má následne vrátiť číslo z rozsahu $0, 1, \ldots, N-1$. Môže očakávať, že N bude mocnina 2. Vrátená hodnota má závisieť iba od hodnôť parametroch, teda pri dvoch volaniach s rovnakými parametrami musí vrátiť vždy rovnakú hodnotu. Cieľom je túto funkciu napísať tak, aby mala čo najmenej kolízii, od toho závisí potom rýchlosť týchto kontajnerov.

1.3.1 THashSet

Neusporiadaná množina. Umožňuje vkladať, mazať, hľadať prvok a prechádzať prvkami v nejakom vopred nedefinovanom poradí. Každý prvok sa v množine môže vyskytovať maximálne raz.

Metóda	Garancia časovej zložitosti
Popis	
`Create`	$O(1)$
Konštruktor. Vyrobí prázdnu množinu.	
`function Size(): SizeUInt`	$O(1)$
Vráti počet prvkov v množine.	
`procedure Insert(value: T)`	V priemere $O(1)$. V najhoršom prípade $O(N)$.
Vloží prvok do množiny. Ak sa tam už nachádza, tak sa nič nestane.	
`procedure Delete(value: T)`	Rovnako ako insert
Zmaže prvok z množiny. Ak sa tam nenachádza, nerobí nič.	
`function Contains(value: T):boolean`	Rovnako ako insert
Zistí, či sa prvok nachádza v množine.	
`function IsEmpty(): boolean`	$O(1)$
Vráti true, ak je množina prázdna.	
`function Iterator():TIterator`	$O(1)$
Vráti iterátor, ktorým môžeme prechádzať cez množinu. Ak je množina prázdna, vracia nil.	

Pozor tento iterátor nie je imúnny na zmeny množiny. Môže sa veľmi ľahko stať, že napr. po vložení prvku do množiny ukazuje na iný prvok ako predtým. Má nasledovné funkcie:

Metóda	Garancia časovej zložitosti
Popis	

function Next:boolean	$O(N)$ v najhoršom prípade, ale prechod celej množiny trvá čas $O(N)$.
Posunie iterátor na ďalší prvok v množine. Ak sa to podarilo vráti true, ak sme prešli celou množinou, vráti false.	

property Data:T	$O(1)$
Property, pomocou ktorej sa dá čítať prvok, na ktorý iterátor ukazuje.	

Príklad použitia (hašovacia funkcia v príklade je len na ukážku, určite sa nedá považovať za dobrú funkciu):

```
{$mode objfpc}

uses ghashset;

type hashlli=class
       public
       class function hash(a:longint; b:SizeUInt):SizeUInt;
     end;
     setlli=specialize THashSet<longint, hashlli>;

class function hashlli.hash(a:longint; b:SizeUInt):SizeUInt;
begin
  hash:= a mod b;
end;

var data:setlli; i:longint; iterator:setlli.TIterator;

begin
  data:=setlli.Create;

  for i:=0 to 10 do
    data.insert(i);

  {Iteration through elements}
  iterator:=data.Iterator;
  repeat
    writeln(iterator.Data);
  until not iterator.Next;
  {Don't forget to destroy iterator}
  iterator.Destroy;
```

```
  data.Destroy;
end.
```

1.3.2 THashMap

Neusporiadané asociatívne pole. Uchováva dvojice (kľúč, hodnota). Platí, že nemôžu existovať dva prvky s rovnakým kľúčom. Na špecializáciu potrebujeme tri argumenty - typ kľúča, typ ukladanej hodnoty a hašovaciu funkciu pre kľúče.

Metóda	Garancia časovej zložitosti
Popis	
`Create`	$O(1)$
Konštruktor. Vyrobí prázdne pole.	
`function Size(): SizeUInt`	$O(1)$
Vráti počet prvkov v poli.	
`procedure Insert(key:TKey; value:TValue)`	V priemere $O(1)$. V najhoršom prípade $O(N)$.
Vloží danú dvojicu do poľa. Ak sa už prvok s daným kľúčom v poli nachádza hodnota sa prepíše.	
`procedure Delete(key: TKey)`	Rovnako ako insert
Vymaže prvok z poľa, ak sa tam nenachádza nič sa nestane.	
`function Contains(key: TKey):boolean`	Rovnako ako insert
Zistí, či sa prvok s daným kľúčom nachádza v poli.	
`function IsEmpty(): boolean`	$O(1)$
Vráti true, ak je pole prázdne.	
`function Iterator():TIterator`	$O(1)$
Vráti iterátor, ktorým môžeme prechádzať cez pole. Ak je pole prázdne vracia nil.	
`property item[i: Key]: TValue; default;`	O(ln N)
Property na prístup k prvkov pola. Pri čítaní vráti hodnotu asociovanú s daným kľúčom. Pri zapisovaní prepíše danú hodnotu pri kľúči a ak kľúč neexistuje vytvorí novú hodnotu. Pri pokuse o čítanie neexistujúcej hodnoty vráti niečo náhodné. Je to default property, takže na použitie stačia hranaté zátvorky.	

Pozor tento iterátor nie je imúnny na zmeny množiny. Môže sa veľmi ľahko stať, že napr. po vložení prvku do množiny ukazuje na iný prvok ako predtým. Má nasledovné funkcie:

Metóda	Garancia časovej zložitosti
Popis	

function Next:boolean	$O(N)$ v najhoršom prípade, ale prechod celej množiny trvá čas $O(N)$.
Posunie iterátor na ďalší prvok v množine. Ak sa to podarilo vráti true, ak sme prešli celou množinou, vráti false.	

property Key:TKey	$O(1)$
Property, pomocou ktorej sa dá čítať kľúč, na ktorý iterátor ukazuje.	
property Value:TValue	$O(1)$
Property, pomocou ktorej sa dá čítať a prepisovať hodnota, na ktorú iterátor ukazuje.	

property MutableValue:PValue	$O(1)$
Vráti pointer na uloženú hodnotu. Užitočné pri prepisovaní hodnôt v uložených recordoch a objektoch.	

Príklad použitia (hašovacia funkcia v príklade je len na ukážku, určite sa nedá považovať za dobrú funkciu):

```
{$mode objfpc}

uses ghashmap;

type hashlli=class
         public
         class function hash(a:longint; b:SizeUInt):SizeUInt;
      end;
      maplli=specialize THashMap<longint, longint, hashlli>;

class function hashlli.hash(a:longint; b:SizeUInt):SizeUInt;
begin
  hash:= a mod b;
end;

var data:maplli; i:longint; iterator:maplli.TIterator;

begin
  data:=maplli.Create;

  for i:=0 to 10 do
    data[i] := 17*i;

  data.delete(5);

  {Iteration through elements}
```

```
  iterator:=data.Iterator;
  repeat
    writeln(iterator.Key, ' ', iterator.Value);
  until not iterator.Next;
  {Don't forget to destroy iterator}
  iterator.Destroy;

  data.Destroy;
end.
```

1.4 Algoritmy pre polia

Tieto funkcie pracujú nad poľami zadanej veľkosti. Pre špecializáciu potrebujú poznať typ poľa a typ prvku, ktorý je uchovávaný v poli. Pole nemusí byť nutne typ Array, môže to byť akýkoľvek kontajner, ku ktorého prvkom vieme pristupovať pomocou hranatých zátvoriek.

1.4.1 Algoritmy bez znalosti poradia prvkov

Všetky tieto algoritmy sú class metódy triedy TArrayUtils. Je tu len jedna metóda:

Metóda	Garancia časovej zložitosti
Popis	
`class procedure RandomShuffle(arr: TArr, size:SizeUint)`	$O(N)$
Preusporiada prvky na indexoch $0, 1, \ldots, size - 1$ do náhodného poradia. Každé poradie má rovnakú pravdepodobnosť.	

1.4.2 Algoritmy so znalosťou poradia prvkov

Tieto algoritmy sú class metódy triedy TOrderingArrayUtils. Táto trieda potrebuje na špecializáciu okrem typu prvku a typu poľa aj porovnávaciu triedu pre prvky.

Obsahuje nasledovné metódy:

Metóda	Garancia časovej zložitosti
Popis	
`procedure Sort(arr: TArr, size:SizeUint)`	$O(N \log N)$ v najhoršom prípade.
Usporiada prvky na indexoch $0, 1, \ldots, size - 1$ od najmenšieho po najväčší.	
`function NextPermutation (arr: TArr, size:SizeUint):boolean`	Jedno volanie $O(N)$. Ale prejdenie všetkými permutáciami trvá $O(N!)$ času.

Usporiada prvky na indexoch $0, 1, \ldots, size - 1$ do najbližšej lexikograficky väčšej permutácie a vráti true. Pokiaľ je už daná permutácia lexikograficky najväčšia, vráti false.

Príklad použitia:

```
uses garrayutils, gutil, gvector;

type vectorlli = specialize TVector<longint>;
     lesslli = specialize TLess<longint>;
     sortlli = specialize
         TOrderingArrayUtils<vectorlli, longint, lesslli>;

var data:vectorlli; n,i:longint;

begin
  randomize;
  data:=vectorlli.Create;
  read(n);
  for i:=1 to n do
    data.pushback(random(1000000000));
  sortlli.sort(data, data.size());
  for i:=1 to n do
    writeln(data[i-1]);

  data.Destroy;
end.
```

Kapitola 2

Implementácia

2.1 Kontajnery pre sekvencie

2.1.1 Pole s premenným počtom prvkov

Táto štruktúra si na pozadí udržuje bežné pole (ktoré nie je nutne rovnako veľké ako pole pri pohľade zvonku), pamätá si jeho veľkosť a pamätá si počet použitých prvkov (veľkosť poľa pri pohľade zvonku). Vkladanie prvkov na koniec poľa je potom vo väčšine prípadov jednoduché – vložíme prvok na prvé nepoužité miesto. Pokiaľ nám dôjde miesto v poli, tak jeho veľkosť zväčšíme k-krát (bežne $k = 2$) a obsah pôvodného poľa presunieme. Jednoduchým myšlienkovým postupom vieme ukázať, že amortizovaná zložitosť pridávania prvkov na koniec je $O(1)$.

Nech sa práve zväčšilo pole veľkosti N na veľkosť kN. Toto nás stálo $O(N)$ času. Od predchádzajúceho zväčšenia sme ale $N - N/k = O(N)$ krát vložili prvok na koniec. To znamená, že amortizovaná zložitosť jedného pridania prvku na koniec je $O(1)$.

Ostatné operácia s touto štruktúrou vieme implementovať pomerne priamočiaro.

Implementácia zásobníka je len jedna vrstva nad touto štruktúrou, ktorá pri volaní push/pop pridáva/odoberá prvky z konca poľa.

2.1.2 Obojsmerná fronta

Implementácia je podobná ako pri pole s premennou veľkosťou. Navyše si ale udržuje index na ktorom je začiatok fronty. Fronta je navyše v poli uložená cyklicky.

Implementácia obyčajnej fronty je len vrstva nad obojsmernou frontou.

2.2 Usporiadané kontajnery

2.2.1 Prioritná fronta

Prioritnú frontu implementujeme pomocou haldy.

Definícia 2.2.1 *Halda je binárny strom, v ktorom platia nasledujúce vlastnosti:*

1. *Každý prvok je väčší ako jeho synovia.*
2. *Strom je rovnomerne vyvážený, tzn. hĺbka listov sa líši maximálne o 1 a a listy s väčšou hĺbkou sú naľavo od ostatných.*

Takýto strom si vieme pohodlne pamätať v poli nasledovne: Koreň bude mať index 1 a synovia vrcholu X budú mať indexy $2X$ a $2X+1$. Všimnite si, že toto nám priamo ukazuje, že hĺbka haldy bude maximálne $O(\log_2 N)$. Keďže veľkosť haldy sa môže ľubovoľne meniť ako pole použijeme už implementované pole s premennou veľkosťou.

Operácie pridania prvku a vymazania maxima implementujeme nasledovne:

Pridanie prvku
Na pozíciu $P+1$ pridáme nový prvok. A zároveň ak je väčší ako jeho otec, tak ho vymeníme s jeho otcom a takto rekurzívne postupujeme smerom nahor. Takto vykonáme maximálne $O(\log_2 N)$ operácií.

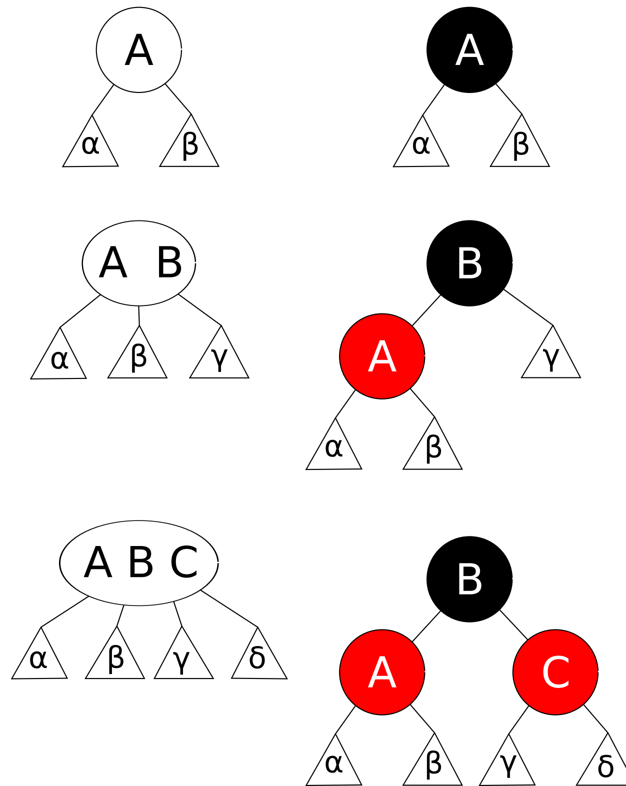
Vymazanie maxima
Vymeníme prvky na pozícii 1 a N. Posledný prvok (N) môžeme vyhodiť. Teraz ale máme pokazenú haldu na pozícii 1. Zavoláme procedúru `uprav` na index 1. Procedúra `uprav` pre prvok X funguje nasledovne: Ak je niektorý syn väčší, tak prvok X s vymení s väčším synom a zavolá `uprav` na tohoto syna. Takýmto spôsobom „dotlačíme“ tento prvok tam kam patrí a urobíme maximálne h krokov, kde h je hĺbka stromu, takže máme zložitosť $O(\log_2 N)$.

2.2.2 Usporiadaná množina

Pri implementácii usporiadanej množiny sú prirodzenou voľbou binárne vyhľadávacie stromy. Na to, aby sme garantovali dobrú časovú zložitosť, treba stromy vyvažovať. Vyvažovaných stromov je viacero druhov. Pre nás sú nevhodné tie stromy, pri ktorých je garantovaná časová zložitosť amortizovaná alebo závisí od generátora náhodných čísel (napriek tomu, že tieto stromy sa väčšinou implementujú ďaleko jednoduchšie). Jedným zo stromov, ktorý spĺňa tieto požiadavky, je red-black strom.

Definícia 2.2.2 *Red-black strom je binárny vyvažovaný strom, ktorý spĺňa nasledovné požiadavky:*

- *Každý vrchol stromu je buď červený alebo čierny*
- *Koreň stromu je vždy čierny*
- *Červený vrchol môže mať len čiernych potomkov*
- *Každá cesta z vrcholu k listu obsahuje rovnako veľa čiernych vrcholov*



Obr. 2.1: Izomorfizmus medzi 2-3-4 stromom a left-leaning red-black stromom

Vlastnosti stromu zabezpečujú, že jeho hĺbka neprekročí $2\log_2 N$. Na druhej strane je o ňom všeobecne známe, že jeho implementácia je pomerne dlhá a nepríjemná. Podrobný popis implementácie možno nájsť v [CSRL01].

V roku 2008 R. Sedgewick vymyslel left leaning red-black stromy [Sed], ktoré pridávajú k red-black stromom jednu podmienku navyše, ale majú oproti nim jednoduchšiu implementáciu.

Definícia 2.2.3 *Left-leaning red-black strom je red-black strom s nasledovnou vlastnosťou navyše:*

- *Pokiaľ má vrchol iba jedného červeného potomka, tak to môže byť len jeho ľavý syn.*

Vďaka tejto vlastnosti existuje jednoduchý izomorfizmus medzi týmto stromom a 2-3-4 stromom, ktorý je zobrazený na obrázku 2.1.

Navyše sa nám zredukuje aj počet možných prípadov, ktoré musíme pri vyvažovaní stromu ošetriť, vďaka čomu sa nám kód značne zjednoduší.

Asociatívne pole implementujeme tiež pomocou binárnych vyhľadávacích stromov, ale miesto jednej hodnoty si musíme pamätať usporiadanú dvojicu (kľúč, hodnota).

2.3 Neusporiadané kontajnery

Neusporiadané kontajnery sme implementovali pomocou hašovacích tabuliek.

Základný popis funkcie možno nájsť v [CSRL01]. V našej implementácii sme zvolili v prípade kolízii jednoduché riešenie pomocou zoraďovania kľúčov za seba do poľa s premennou veľkosťou. Taktiež keďže zo začiatku nie je veľkosť dát dopredu známa, museli sme implementovať „rozťahovanie“ tabuľky, čiže keď počet kľúčov uložených v tabuľke presiahne k-násobok veľkosti tabuľky, tak veľkosť tabuľky zväčšíme na n-násobok a hodnoty popresúvame na nové miesta. V našom prípade sme zvolili $k = n = 2$.

Pokiaľ užívateľ zvolí dobrú hašovaciu funkciu, tak potom je očakávaný čas vyhľadávania, vkladania a mazania kľúča $O(1)$ (výnimka je len, keď sa zrovna po vložení celá tabuľka „roztiahne“).

2.4 Triedenie

Bežné známe algoritmy, ktoré triedia porovnávaním v čase $O(N \log_2 N)$, sú merge sort, heap sort a quick sort (posledný menovaný len v priemernom prípade). Merge sort je v našom prípade nepoužiteľný, keďže v bežnej implementácii potrebuje pomocnú pamäť rovnako veľkú ako triedené pole.

2.4.1 Quick sort

Tento algoritmus vymyslel v roku 1962, C. A. R. Hoare [Hoa62]. Algoritmus pracuje nasledovne:

1. Vyberie z postupnosti prvok X, tzv. pivot
2. Rozdelí postupnosť na prvky menšie ako X a väčšie alebo rovné ako X
3. Opakuje tento postup rekurzívne na rozdelených postupnostiach

Veta 2.4.1 *Algoritmus quick sort má v najhoršom prípade zložitosť $O(N^2)$, v priemernom prípade $O(N \log_2 N)$.*

Dôkaz. Dôkaz sa dá nájsť už v spomínanom článku od Hoareho ([Hoa62]).

Užitočnou vlastnosťou quick sortu je, že je tzv. in-place triedením teda triedenie vykonáva priamo v zadanej postupnosti.

Hlavnou otázkou pri implementácii je výber pivota. Pri výbere pivota ako prvého-/posledného prvku z postupnosti dostávame zlú časovú zložitosť pre (takmer) utriedené postupnosti.

Inou možnosťou je tzv. median-of-3. Vyberieme median z prvého, posledného a stredného prvku postupnosti. Táto možnosť má celkom dobré výsledky v praxi, ale existujú tzv. median-of-3 killer postupnosti, ktoré sú utriedené pomaly. Ako sa dajú generovať sa dá nájsť napríklad v [Mus97].

2.4.2 Heap sort

Tento algoritmus využíva už popísanú prioritnú frontu (haldu). Najprv vloží všetky prvky do haldy. A následne z nej postupne vyberá najväčšie prvky a tie umiestňuje na koniec poľa. Obidve časti nie je problém implementovať priamo v danom poli (prvá časť poľa je halda a zbytok je ešte nevložená časť pri vkladaní, resp. už vybraná časť pri vyberaní). Navyše existuje in-place Floydov algoritmus, ktorý v $O(N)$ čase vybuduje haldu (popis napr. v[CSRL01]).

Priamo z popisu algoritmu vyplýva, že jeho časová zložitosť je aj v najhoršom prípade $O(N \log_2 N)$ (urobíme $2N$ operácii s haldou). To je jeho výhodnou oproti quick sortu. Existuje niekoľko vylepšení, ktoré zlepšujú hlavne konštantné faktory, zaujímavý je napr. bottom up heapsort ([Weg93]).

Napriek všetkým týmto vylepšeniam heap sort na súčasných počítačoch nemá šancu byť rýchlejší ako quick sort. Dôvodom je spôsob prístupu ku prvkom postupnosti. Kým quick sort k prvkom pristupuje v podstate postupne, tak heap sort v dvoch nasledovných krokoch berie prvky, ktoré sú v postupnosti celkom ďaleko od seba. A toto sa dosť prejaví v počte tzv. cache missov (kedy sa dáta z pamäte musia natiahnuť do L1/L2 cache procesora), keďže pri načítaní údajov do cache dosť často načítame nie len jeden prvok z poľa, ale niekoľko susedných.

2.4.3 Intro sort

V roku 1997 navrhol D. Musser intro sort ([Mus97]). Jeho fungovanie by sa dalo zhrnúť nasledovne: Najprv triedi ako quick sort, ale pokiaľ je už hĺbka rekurzie príliš veľká utriedi danú časť postupnosť pomocou heap sortu. Vďaka tomu v priemernom prípade je rýchly ako quick sort a zároveň garantuje najhoršiu časovú zložitosť $O(N \log_2 N)$.

Posledným vylepšením, ktoré má je navyše to, že veľmi krátke postupnosti už netriedi. Ale na konci ešte celú postupnosť utriedi insert sortom, ktorý má síce kvadratickú časovú zložitosť, ale keď sa každý prvok od svojej finálnej pozície nachádza blízko, tak je rýchly.

V našej implementácii sme použili práve intro sort.

2.5 Testovanie implemetácie

FreePascal obsahuje vo svojej štandardnej distribúcii modul FPCUnit, čo je obdoba JUnitu. Vďaka nemu vieme pre každú funkcionalitu napísať sadu automatizovaných testov, ktoré otestujú funkčnosť a uistia nás, že v implementácii sa nenachádzajú skryté chyby.

Kapitola 3

Praktické testy

Cieľom tejto kapitoly je otestovať použiteľnosť a efektívnosť našej implementácie na niekoľkých praktických príkladoch. Väčšina príkladov sa vyskytla na programátorských súťažiach. Naša metodika bude pomerne jednoduchá. Napíšeme ten istý program v C++ s použitím STL a v Pascale s použitím našej knižnice a porovnáme čas behu.

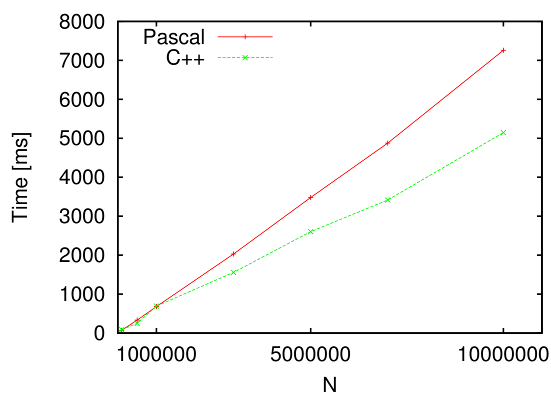
3.1 Triedenie

Prvý test je pomerne jednoduchý. Program dostane na vstupe N celých čísel a má ich utriediť. Obidve implementácie najprv načítajú vstup do poľa s premennou veľkosťou a následne ho utriedia. Výsledky možno vidieť v tabuľke a grafe.

3.2 Obedy

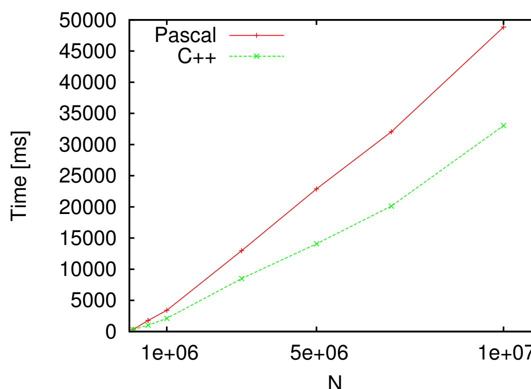
Táto úloha je 8. úloha z 2. série 28. ročníka Korešpondenčného seminára z programovania. Zadanie sa dá zhrnúť nasledovne: Máme zadané 2 postupnosti čísel $a_1, \ldots, a_n$, $b_1, \ldots, b_m$,

	Čas behu [ms]	
N	C++	Pascal
10^5	69	86
$5 \cdot 10^5$	249	333
10^6	693	679
$3 \cdot 10^6$	1542	2122
$5 \cdot 10^6$	2603	3479
$7 \cdot 10^6$	3213	4912
10^7	5146	7259



Obr. 3.1: Porovnanie rýchlosti triedenia

	Čas behu [ms]	
N	C++	Pascal
10^5	313	326
$5 \cdot 10^5$	1026	1779
10^6	2136	3419
$3 \cdot 10^6$	8512	12972
$5 \cdot 10^6$	14099	22888
$7 \cdot 10^6$	20135	32057
10^7	33051	48860



Obr. 3.2: Porovnanie rýchlosti hľadania najdlhšej rastúcej podpostupnosti

pričom v prvej z nich sa každé číslo vyskytuje maximálne raz (teda čísla $a_1, \ldots, a_n$ sú navzájom rôzne). Úlohou je nájsť najdlhšiu spoločnú podpostupnosť týchto postupností.

Prvý krok riešenia je nasledovný: Číslo b_i prepíšeme na číslo c_i pričom platí $a_{c_i} = b_i$, ak také číslo c_i neexistuje, tak číslo b_i z postupnosti vyhodíme. Inými slovami čísla z druhej postupnosti nahradíme ich indexmi v postupnosti prvej. Tento krok vieme veľmi ľahko spraviť pomocou asociatívneho poľa. A teraz ideme hľadať v tejto upravenej postupnosti najdlhšiu rastúcu podpostupnosť.

Hľadanie najdlhšej rastúcej postupnosti môžeme vykonať napr. nasledovne: Postupnosť spracúvame postupne po prvkoch. Osobitne si pamätáme nasledovnú dvojicu údajov (v, d), ktoré hovoria, že v je najmenšie číslo, v ktorom môže zatiaľ končiť rastúca podpostupnosť dĺžky d. Tento zoznam vieme efektívne skladovať v usporiadanom asociatívnom poli (kľúč je v). Celková časová zložitosť riešenia je $O(N \log_2 N)$, kde N je celková dĺžka oboch postupností.

Časový výsledok porovnania rýchlosti vidno v priloženej tabuľke a grafe.

3.3 Asfalt

Úloha je z celoštátneho kola 26. ročníka olympiády v informatike.

Máme zadanú mriežku rozmerov $R \times S$ (obidve veľkosti boli najviac 1000), ktorá určuje výšky jednotlivých políčok. Chceme postaviť najlacnejšiu diaľnicu z políčka $(0, 0)$ do políčka $(R-1, S-1)$. Vieme cenu za jedno políčko diaľnice. A tiež vieme, že diaľnica môže viesť medzi políčkami, ktorých výška sa líši maximálne o 1. Tiež môžeme stavať tunely (resp. mosty), ktoré môžu viesť medzi políčkami, ktoré majú rovnakú výšku a aspoň jednu súradnicu rovnakú, a všetky políčka medzi nimi majú väčšiu (resp. menšiu) výšku. Tiež vieme cenu za jedno políčko mostu, resp. tunela.

Riešenie je pomerne priamočiare. Vybudujeme graf medzi políčkami. Mosty a tunely vieme nájsť v lineárnom čase prechodom každého stĺpca a riadku (pri prechode si pomáhame zásobníkom, v ktorom máme uložené políčka, ku ktorým sa ešte dá zapojiť most, resp.

	Čas behu [ms]	
Číslo vstupu	C++	Pascal
6	179	286
7	86	139
8	899	1249
9	909	1776
10	946	1099
11	1226	2603
12	1493	2253
13	1643	2383
14	1326	3443
15	1353	3436

Obr. 3.3: Porovnanie rýchlosti hľadania najkratšej cesty

tunel). Následne v tomto grafe nájdeme najkratšiu cestu pomocou Dijkstroveho algoritmu s prioritnou frontou.

Porovnanie časov je zhrnuté v tabuľke (keďže vstupy sa nedajú jednoducho charakterizovať veľkosťou). Niektoré vstupy sme vynechali, keďže sú príliš malé na to, aby programy na nich vykazovali merateľný rozdiel.

3.4 Básničky II.

Úloha je opäť z celoštátneho kola 26. ročníka olympiády v informatike.

Chceme zostaviť báseň, ktorá má N slôh. Každá sloha obsahuje presne dva verše. Navyše musí platiť, že druhý verš i-tej slohy a rýmu s prvým veršom $(i + 1)$-vej slohy (a druhý verš N-tej slohy sa rýmuje s prvým veršom prvej slohy). Pre každú slohu máme zadaných niekoľko variantov (kódy zakončení jednotlivých veršov, rýmujú sa iba tie, ktoré majú rovnaký kód). Úlohou je zostaviť básničku, prípadne podať správu, že neexistuje.

Riešenie využíva v podstate hrubú silu. Pre každý možný začiatočný verš postupne všetky možné zakončenia $1, 2, \dots, N$-tej slohy. Ak je medzi možnými zakončeniami N-tej slohy aj začiatočný verš, tak sme zostrojili dobrú báseň. Keďže kód zakončení veršov môžu byť aj veľké celé čísla, tak si ich uchovávame v neusporiadanom asociatívnom poli (keďže si potrebujeme aj pamätať ako daný verš vznikol, aby sme vedeli rekonštruovať báseň).

Porovnanie časov je zhrnuté v tabuľke (keďže vstupy sa nedajú jednoducho charakterizovať veľkosťou). Niektoré vstupy sme vynechali, keďže sú príliš malé na to, aby programy na nich vykazovali merateľný rozdiel.

	Čas behu [ms]	
Číslo vstupu	C++	Pascal
7	213	139
11	109	69
12	1069	569
13	1743	716
14	7886	3186
15	433	639

Obr. 3.4: Porovnanie rýchlosti skladania básne

3.5 Zhodnotenie testovania

Z výsledkov môžeme vidieť niekoľko vecí. Vidíme, že naša implementácia je o niečo pomalšia ako implementácia ekvivalentných štruktúr v C++, ale na druhej nie je oveľa pomalšia.

Pokiaľ by sme chceli hľadať príčiny spomalenia, tak sú hlavne dve. Jednak C++ kompilátor vie o niečo lepšie optimalizovať kód. A dvak Pascal má ďaleko pomalšiu dynamickú alokáciu.

Záver

V tejto práci sme úspešne implementovali a prezentovali rozhranie knižnice štandartných algoritmov pre FreePascal. Navyše sa nám podarilo našu knižnicu dostať medzi knižnice, ktoré sú distribuované spolu s FreePascalom.

Literatúra

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[Hoa62] C. A. R. Hoare. Quicksort. *Comput. J.*, 5(1):10–15, 1962.

[Mus97] David Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27:983–993, 1997.

[Sed] Robert Sedgewick. Left-leaning red-black trees. `http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf`.

[STL] Standard template library programmer's guide. `http://www.sgi.com/tech/stl/`.

[Weg93] Ingo Wegener. Bottom-up-heapsort, a new variant of heapsort beating, on an average, quicksort (if n is not very small). *Theor. Comput. Sci.*, 118:81–98, September 1993.