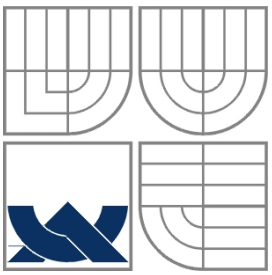
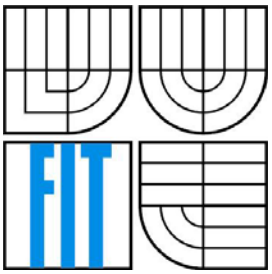


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ALGORITMY VYHLEDÁVÁNÍ V JAZYCE C

SEARCHING ALGORITHMS IN C LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

Ivan Nejezchleb

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. Ing. Jan Maxmilián Honzík, CSc.

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Nejezchleb Ivan**
Obor: Informační technologie
Téma: **Algoritmy vyhledávání v jazyce C**
Kategorie: Alg. a datové struktury

Pokyny:

1. Seznamte se detailně s textem kapitoly 4 o vyhledávání ve studijní opoře pro předmět IAL
2. Modifikujte text kapitoly zapsané pro pascalovský jazyk tak, aby zachoval co nejvíce (i v detailech) původní obsah, ale aby se vztahoval k zápisu příkladů a algoritmů v jazyce C
3. Převeďte všechny příklady a algoritmy do jazyka C a odladťe je v prostředí vytrvořeném pro tento účel
4. Vytvořte program pro animovanou demonstraci vybraných operací s využitím nástrojů poro grafickou reprezentaci
5. Navrhněte další vhodné kontrolní otázky a příklady.
6. Navrhněte příklady vhodné pro formulářově orientovanou písemnou zkoušku ze znalostí tohoto okruhu.

Literatura:

- Honzík, J.M. Algoritmy. Studijní opora pro předmět Algoritmy. Elektronický text. FIT VUT v Brně

Při obhajobě semestrální části projektu je požadováno:

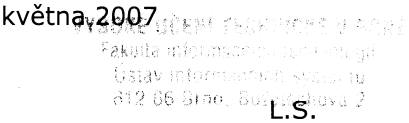
1. Přepsané příklady a algoritmy do jazyka C, odladěné v prostředí pro tento účel vytvořeném. Program animované demonstrace operací a chování vybraných metod vyhledávání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

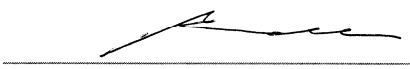
Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Honzík Jan M., prof. Ing., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007



L.S.



doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Ivan Nejezchleb**
Id studenta: 84229
Bytem: Pekařská 1914/2, 678 01 Blansko
Narozen: 28. 04. 1985, Boskovice
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Algoritmy vyhledávání v jazyce C
Vedoucí/školitel VŠKP: Honzík Jan M., prof. Ing., CSc.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☒ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: 16. 7. 2007

....

Nabyvatel Autor

Abstrakt

Vyhledávání ve všech možných formách je v dnešní době hojně používanou operací nejen v oblasti informačních technologií. Proto je pochopení a ovládnutí základních vyhledávacích algoritmů nezbytné pro každého, kdo se chce vyvíjet služby obsahujících i mechanismus vyhledávání.
Ve své práci se zabývám vyhledáváním především z pohledu programátora jazyka C. Představím zde ty nejzákladnější vyhledávací algoritmy a aplikace demonstrující jejich činnost. Cílem celé práce je vytvořit učební pomůcky pro snazší pochopení problematiky vyhledávání.

Klíčová slova

Vyhledávací algoritmy, sekvenční vyhledávání, binární vyhledávání, Dijkstrova varianta binárního vyhledávání, Fibonacciho vyhledávání, binární vyhledávací strom, tabulky s rozptýlenými položkami, demonstrační aplikace

Abstract

Searching in all possible forms is at the present time widely used operation not only in the subject of information technology. So the understanding and the grasp of the basic searching algorithms is necessary for everyone who wants to develop services containing searching mechanism.
In my work I deal with the searching from the view of C language programmer. I will introduce basic searching algorithms and demo applications of their principles. Goal of whole work is to create study aid for easier understanding of the search subject.

Keywords

Searching algorithms, sequential search, binary search, Dijkstra's modification of binary search, Fibonacci search, binary search tree, hashing tables, demo application

Citace

Nejezchleb Ivan: Vyhledávací algoritmy v jazyce C. Brno, 2007, bakalářská práce, FIT VUT v Brně.

Vyhledávací algoritmy v jazyce C

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Prof. Ing. Jana Maxmiliána Honzíka, CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Ivan Nejezchleb
2.7.2007

Poděkování

Rád bych touto formou poděkoval svému vedoucímu práce, panu Prof. Ing. Janu Maxmiliánu Honzíkovi, CSc., za jeho odborné vedení, podnětné rady a připomínky.

Obsah

Úvod

Vyhledávání je jednou z nejvýznamnějších kapitol téměř všech textů zabývajících se algoritmy. Nejde však o izolované téma, neboť zasahuje do celé řady dalších oblastí nejen informačních technologií. Vzhledem k této návaznosti je pochopení problematiky vyhledávání důležité pro každého programátora. Nejdůležitější je pochopení těch nejjednodušších principů, které jsou pak dále rozvíjeny a zdokonalovány pro vyhledávání ve specifických situacích.

Mechanismy základních vyhledávací algoritmů jsou známy již řadu let a jejich schéma je v průběhu času téměř neměnné. Jediné, co se mění, je konkrétní programovací jazyk, ve kterém tyto algoritmy zapisujeme. S jednou takovou výměnou, která proběhla v roce 2004 na Fakultě informačních technologií VUT v Brně je spojeno i téma této práce. Po přechodu z jazyka Pascal [1] na jazyk C bylo nutné přepracovat i většinu učebních textů. Tato práce je spjata s přepisem studijní opory pro předmět Algoritmy, který je na Fakultě informačních technologií vyučován v druhém ročníku. Z této opory je moje práce zaměřena na kapitolu týkající se vyhledávacích algoritmů.

Obecný popis vyhledávání, definice algoritmu a specifikace vyhledávacích algoritmů jsou součástí první kapitoly toho textu. Jednotlivým vyhledávacím algoritmům, jejich rozdělení a popisu je věnován obsah druhé kapitoly. Třetí kapitola pojednává o významu studijní opory předmětu Algoritmy, o myšlence s jakou byla psána a o její náplni. K podpoře studia tématu týkajícího se vyhledávacích algoritmů byly jako součást této bakalářské práce vytvořeny demonstrační aplikace znázorňující činnost vyhledávacích algoritmů. O návrhu a implementaci těchto programů se čtenář dočte v kapitole 4. V poslední kapitole tohoto textu je obsaženo shrnutí výsledků práce, její přínos a návrhy na případné navazující projekty.

1 Vyhledávání

Před popisem samotných vyhledávacích algoritmů by bylo vhodné sjednotit vnímání pojmu vyhledávání. Jde o činnost známou téměř každému, a proto nepůjde o nikterak obtížný úkol. "Vyhledávání je základní operací získávání dílčích informací z velkého objemu uložených dat" [2]. Tato slova mohou vyvolat dojem, že vyhledávání je činností specifickou pro počítače, ale tento dojem by byl mylný. I v běžném životě narážíme na situace, kdy něco hledáme. Dobrým příkladem by mohlo být pouhé vyhledání kontaktu v telefonním seznamu. V takových situacích sice nevyužíváme vyhledávací algoritmy, i když bychom v mnoha případech samozřejmě mohli, ale vzhledem k malému množství dat, která prohledáváme, často postupujeme zcela intuitivně. V dnešní době internetu, rozsáhlých databází, velkých společností shromažďujících informace o svých zákaznících apod. však stále častěji vyhledáváme v ohromném množství dat. A v těchto případech už jsme nuceni použít vyspělejších vyhledávacích algoritmů.

Volba vyhledávacího algoritmu může být ovlivněna mnoha hledisky. Jedním z nich je jistě množina dat, nad kterou vyhledávání probíhá. Tato data nazýváme tabulka symbolů (nebo také vyhledávací tabulka):

"Tabulka symbolů je datová struktura položek s klíči, která podporuje dvě základní operace: vložení nové položky a vrácení položky s daným klíčem." [2]

Právě vrácení položky s daným klíčem je operací vyhledání. Někdy se setkáváme i s operací rušení položky, která bude zmíněna dále.

Vyhledání většinou nespočívá v pouhém konstatování, zda se hledaná položka v tabulce symbolů nachází či nikoli, ale umožňuje i přístup k této položce. Cílem vyhledávání je totiž většinou získání informace obsažené v položce (nikoliv pouze v klíči) pro zpracování. [2] Struktura položky je odlišná případ od případu a je úzce spjata s objektem reálného světa, o kterém v sobě uchovává informace. Položka představující záznam z telefonního seznamu z výše uvedeného příkladu by pak zřejmě obsahovala informace o jméně vlastníka telefonní linky, jeho adresu a telefonní číslo. A tato položka by jistě měla odlišnou podobu oproti položce představující např. kus nábytku v inventáři muzea. Společnou složkou všech položek je však již zmíněný klíč, neboť ten je pro vyhledávání nezbytný.

Klíčem se nazývá množina hodnot, která jednoznačně identifikuje záznam. Pro množinu hodnot klíče, podle kterého se bude vyhledávat, je významné rozlišit případy, kdy jsou nad typem klíče definována pouze pravidla pro ekvivalenci dvou prvků, nebo kdy je navíc definována i relace uspořádání. [3] Definování ekvivalence je pro vyhledávání nezbytné a jen pak můžeme rozhodnout, zda jsou si vyhledávaný klíč a klíč právě porovnávané položky rovny

či nikoliv. Pokud máme o vyhledávací tabulce a množině klíčů tyto bližší informace jako např. zda jsou data v ní uspořádána či nikoli, můžeme na základě tohoto faktu zvolit efektivnější algoritmus a docílit tak lepších výsledků. Navíc i samotný klíč může mít různou strukturu. Klíč může být jednoduchý nebo strukturovaný. Priorita složek strukturovaného klíče určuje váhu (význam) jednotlivých položek. Pro ekvivalenci dvou strukturovaných klíčů musí být ekvivalentní všechny odpovídající si složky klíče. Pro relaci uspořádání dvou strukturovaných klíčů se porovnávají postupně odpovídající si složky klíče se snižující se prioritou (vahou). [3] Dalším tématem, které se týká klíče, je možnost položek s duplicitními klíči, tedy dvou a více různých položek se stejným vyhledávacím klíčem. [2] Některé vyhledávací algoritmy neumožňují duplicitní klíče vůbec. Tyto algoritmy budou tvořit většinu vyhledávacích algoritmů představených v dalších kapitolách. V jiných případech je možnost duplicitních klíčů povolena, ale je nutné vyhledávací algoritmy modifikovat. Důležitou otázkou je také to, který z prvků se shodnými klíči má být vyhledáním vrácen. Zde existuje množství variant. Nejběžnějším požadavkem je vrácení některého z krajních prvků (pravý, či levý). V některých případech však můžeme požadovat vrácení všech prvků s hledaným klíčem k jejich dalšímu zpracování aplikací. Jednou z variant je také vrácení libovolného z odpovídajících prvků. Řešení tohoto problému je závislé od konkrétních požadavků aplikace. V dalším textu zmíním jen některé možnosti, jak tento problém řešit.

1.1 Algoritmus

V předešlém odstavci je mnohokrát zmíněn pojem algoritmus. Bylo by tedy vhodné předložit definici tohoto pojmu.

Obecně lze algoritmus chápat jako přesný návod či postup, kterým lze vyřešit daný typ úlohy [4]. Tato definice je poněkud obecná a dovoluje pod pojmem algoritmus chápat i množství běžných úkonů a předpisů jako např. recept na uvaření jistého pokrmu. Přesnější definice říká, že algoritmus je:

> "konečná posloupnost/uspořádání postupů aplikovaných na konečný počet dat, jež dovoluje řešit přibližně stejné třídy problémů.“ [5]

V užším významu, používaném především v počítačové terminologii, musí algoritmus navíc splňovat jistá kriteria [4], z nichž konečnost je zmíněna již v definici výše.

- Konečnost
- Determinovanost
- Vstup
- Výstup

1.2 Vyhledávací algoritmus

Vyhledávací algoritmy, tak jak budou popisovány v tomto textu, jsou úzkou podskupinou obecných vyhledávacích algoritmů (anglicky *search algorithm*). Ty jsou v oblasti počítačových věd obecně definovány jako:

"Algoritmy, které jako vstup přijímají problém a jako svůj výstup vrací jeho řešení, obvykle po vyhodnocení množství možných řešení." [6]

Tato definice zahrnuje celou řadu skupin algoritmů jako např. neinformované hledání, informované hledání, prohledávání seznamů, prohledávání stromů, hledání řetězců a mnoho dalších.

My budeme pod pojmem vyhledávací algoritmy nadále chápat algoritmy úzce spjaté s abstraktním datovým typem (ADT) [2] vyhledávací tabulka. Takové algoritmy umožňující vyhledání požadované položky v této tabulce. ADT tabulka symbolů vedle vyhledávání implementuje také další operace jako vkládání, mazání, výběr, třídění [2]. Poslední zmíněná operace souvisí s relací uspořádání množiny hodnot klíče a s faktem, že některé vyhledávací algoritmy (často mnohem efektivnější) pracují s tabulkou symbolů setříděnou podle hodnot klíče. O této problematice se dále zmiňují kapitoly popisující jednotlivé algoritmy.

V odstavci výše byl zmíněn pojem efektivnost. U vyhledávacích algoritmů, stejně jako u algoritmů obecně, je pod pojmem efektivnost chápána složitost algoritmu. Složitost je charakteristika algoritmu umožňující jeho srovnání s jiným algoritmem. Nejvýznamnější faktory v tomto porovnání jsou především čas, za který je algoritmus proveden, a také množství paměťového prostoru, který při svém běhu algoritmus obsadí. Oba tyto aspekty jsou závislé na velikosti vstupních dat, se kterými algoritmus pracuje, a proto má složitost nejčastěji podobu funkce velikosti dat, udávané počtem položek N. Nejvýznamnější a nejčastěji používaná je asymptotická časová složitost a to především složitost O (Omikron) vyjadřující „nejhorší případ“ činnosti algoritmu [3]. S pojmem časové složitosti je spojeno i další hodnotící kriterium vyhledávacích algoritmů a to přístupová doba. Přístupová doba je doba potřebná k zajištění přístupu (vyhledání) položky s hledaným klíčem. Pro hodnocení se používá několik časových vlastností vyhledávání. Zvlášť se označují doby pro úspěšné a pro neúspěšné vyhledání, které se u některých metod liší:

- minimální doba úspěšného a neúspěšného vyhledání
- maximální doba úspěšného a neúspěšného vyhledání
- průměrná doba úspěšného a neúspěšného vyhledání

Průměrná doba úspěšného vyhledání je teoretický parametr, který je dán podílem součtu dob úspěšného vyhledání všech položek tabulky a počtu položek. [3]

2 Vyhledávací algoritmy

V této kapitole si postupně představíme jednotlivé vyhledávací algoritmy v pořadí, v jakém jsou probírány ve studijní opoře pro předmět Algoritmy. Toto pořadí však také reflektuje i strukturu vyhledávací tabulky, kterou tyto algoritmy prohledávají a jejich přístup k prvkům této tabulky.

Vyhledávací tabulka může být implementována jako lineární homogenní struktura, nejčastěji ve formě seznamu nebo jednorozměrného pole prvků. V tomto případě rozlišujeme algoritmy i z pohledu přístupu k této lineární tabulce a to na algoritmy se sekvenčním přístupem a na algoritmy s nesekvenčním přístupem.

Další možnou implementací vyhledávací tabulky je stromová struktura, nejčastěji v podobě binárního vyhledávacího stromu. V tomto případě již nerozlišujeme způsob přístupu k prvkům, neboť sekvenční přístup zde není používán. Pro stromové struktury jsou naopak podstatné vlastnosti stromu, jeho n-nárnost (maximální počet podstromů jednoho uzlu), váhová popř. výšková vyváženost.

Třetí skupina vyhledávacích metod, které budou popisovány v následujících kapitolách, jsou tabulky s rozptýlenými položkami. Tyto algoritmy pracují s vyhledávací tabulkou implementovanou dle principu pole s asociativním přístupem (můžeme také použít označení tabulky s přímým přístupem [3]). To znamená, že poloha prvku v poli je dána jeho obsahem. Jinými slovy je množina hodnot klíčů namapována do paměťového prostoru, kde se nachází vyhledávací tabulka. Toto mapování je umožněno tzv. hashovací funkcí, která na základě hodnoty klíče určí přesnou polohu prvku v tabulce symbolů.

2.1 Sekvenční vyhledávání v lineární datové struktuře

Algoritmy spadající do této kategorie patří k nejjednodušším vyhledávacím algoritmům, ale jsou také nejpomalejší. V nejzákladnější verzi však sekvenční vyhledávací algoritmus nevyžaduje žádnou znalost bližších informací o množině hodnot klíče. Pokud jsou však tyto informace přeci jen známy, nevedou k nikterak ohromnému nárůstu výkonu. Nezávislost na struktuře dat však umožňuje použít tyto algoritmy ve většině případů a jejich jednoduchá implementace je předurčuje k nasazení v aplikacích, kde na rychlost vyhledání nejsou kladeny vysoké nároky (vyhledávání v malém počtu prvků, operace vyhledání je používána jen zřídkakdy). Vyhledávání se uskutečňuje v lineární datové struktuře - pole, seznam, soubor. Pro zjednodušení se dále budeme zabývat jen případy vyhledávání v poli.

2.1.1 Sekvenční vyhledávání v poli

Algoritmus sekvenčního vyhledávání pracuje s tabulkou symbolů ve formě pole prvků. Každý prvek obsahuje klíčovou složku a složku obsahující „užitečná data“. Základní vyhledávací schéma je popsáno jako [3]:

```
Nasel= false;
while ( ! Nasel && <množina prvků není vyčerpána>) {
< prozkoumej další prvek. Je-li to hledaný, nastav Nasel na true >
} // while
return Nasel;
```

kde `!` v logickém výrazu představuje operátor negace, `&&` vyjadřuje logický operátor konjunkce a `return Nasel` zajistí vrácení výsledku hledání jako výsledku celé funkce (tento zápis, stejně jako všechny ostatní následující, je odvozen od syntaxe a sémantiky jazyka C).

Tento algoritmus vrací pouze výsledek vyhledání a nikoli index nalezeného prvku v případě nalezení. Takovéto funkce dosáhneme malou modifikací uvedeného algoritmu. Buď jako jeden ze vstupních parametrů předáme odkazem proměnnou, v níž bude po skončení úspěšného vyhledávání obsažen index nalezeného prvku, jinak bude její hodnota nedefinovaná. Druhou možností je vytvořit funkci, která jako svou návratovou hodnotu nevrací výsledek hledání ale index nalezeného prvku. V tomto případě jsme o úspěšném nalezení informováni konkrétním indexem v této návratové hodnotě, jinak má tato návratová hodnota předem smluvenou hodnotu vyjadřující, že prvek nalezen nebyl (v případě indexů pole by mohlo jít např. o hodnotu -1).

Jak bylo dříve řečeno je zvykem, že vyhledávací tabulka neimplementuje jen operaci vyhledávání ale i další. Pokud budeme předpokládat aktualizační sémantiku operace vkládání [3], můžeme ji snadno implementovat s použitím modifikované vyhledávací funkce vracející index nalezeného prvku. V případě nalezení jsou data prvku přepsána daty novými, pokud prvek nalezen nebyl, je vytvořen nový na konci obsazené části vyhledávací tabulky a je inkrementováno počítadlo prvků.

Další často nabízenou operací je operace smazání prvku. Pokud vyhledávací tabulka umožňuje i tuto operaci, můžeme hovořit o plně dynamické tabulce [3], která je schopna svůj obsah měnit jak přidáváním tak odstraňováním. Operace smazání prvku se v případě lineární struktury tabulky symbolů provádí nejčastěji přepsáním mazaného prvku prvkem posledním a následným snížením počítadla prvků. Tento způsob je ovšem možný jen pokud se nejedná o seřazenou tabulku symbolů. Takový případ je popsán v kapitole níže.

Složitost popisovaného algoritmu patří do třídy O(N), jedná se tedy o lineární složitost. Minimální doba úspěšného vyhledání je rovna 1, neboť pokud se hledaný prvek nachází na první pozici pole, je nalezen během jednoho porovnání. Maximální doba úspěšného vyhledání

se rovná N, protože leží-li daný prvek až na poslední pozici v tabulce, musíme projít celou její délku. Průměrná doba úspěšného vyhledání je pak $\frac{N}{2}$. Doba neúspěšného vyhledání je vždy rovna N.

Algoritmus vyhledávání se sekvenčním přístupem se dá mírně zrychlit použitím tzv. zarážky. Tato technika spočívá ve vložení vyhledávaného prvku za poslední prvek tabulky symbolů. Vyhledání tak vždy skončí nalezením a o tom, zda jsme našli skutečný prvek nebo právě zarážku, se ujistíme porovnáním indexu nalezeného. Urychlení algoritmu spočívá ve zjednodušení podmínky cyklu while, kdy je možné vynechat dotaz na konec množiny prvků. Jedinou nevýhodou tohoto algoritmu je fakt, že se použitelná kapacita tabulky sníží o jeden prvek, vyhrazený pro vložení zarážky.

2.1.2 Sekvenční vyhledávání v poli s uspořádáním prvků dle četnosti vyhledání

Z rozboru přístupových dob pro sekvenční vyhledávání vyplývá fakt, že prvky na počátku tabulky jsou vyhledány s menším počtem kroků a tedy i rychleji než prvky nacházející se na konci tabulky. Této vlastnosti se dá úspěšně použít v případech, kdy víme, že některé položky tabulky jsou vyhledávány častěji než jiné. Tyto položky by tedy bylo nejvhodnější umístit na počátek vyhledávací tabulky a dobu nutnou pro jejich vyhledání tak co nejvíce zkrátit. Tohoto stavu lze dosáhnout drobnou modifikací algoritmu sekvenčního vyhledávání. Postačí pokud v okamžiku nalezení daný prvek zaměníme s jeho předchůdcem (prvek o jednu pozici předcházející). Tuto výměnu samozřejmě neprovádíme pokud už je nalezený prvek na prvním místě tabulky.

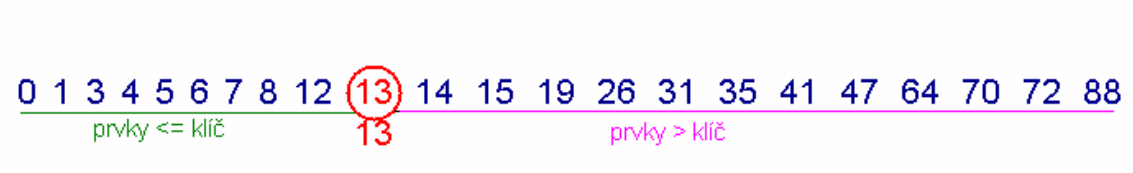
Tato metoda je první z mnoha v tomto textu, která ve snaze urychlit vyhledávání věnuje část času úpravě struktury prohledávaných dat. Tato myšlenka je používána i u jiných algoritmů. Vždy je však nutné zvážit, zda investice do údržby struktury vyhledávací tabulky nepřesáhne úspory, kterých se tím docílí při vykonávání ostatních operací.

2.1.3 Sekvenční vyhledávání v seřazeném poli

Pokud je nad množinou hodnot klíče definována relace uspořádání, lze jí použít k urychlení vyhledávacího algoritmu. Vyhledávací tabulku lze totiž setřídit podle hodnot klíčů jednotlivých položek (převážně vzestupně) a použít algoritmus využívající tohoto stavu. Opět je však nutná jistá investice do údržby struktury tabulky, neboť operace vkládání a rušení prvku musí toto uspořádání akceptovat a po jejich provedení musí zůstat tabulka symbolů seřazená. Při vkládání nového prvku do vyhledávací tabulky musí být nalezena správná pozice, na kterou se má prvek vložit, a musí pro něj být vytvořen prostor, kam jej bude možné vložit. U seznamu je tato

operace poměrně snadná, avšak v tabulce využívající pole je nutné posunout segment pole napravo od místa vkládání o jednu pozici doprava a tato operace již vyžaduje nezanedbatelný čas pro svoje provedení. Stejně tak operace mazání prvku nemůže pouze přepsat rušený prvek posledním, protože by došlo k porušení uspořádání. Pro provedení operace rušení je nutné posunout segment napravo od rušeného prvku o jednu pozici doleva a tím dojde k přepsání rušeného jeho pravým sousedem. V případech, kdy je posun pole příliš zdlouhavou operací, je možné použít ke zrušení položky tzv. zaslepení [12]. Zrušení položky zaslepením se provede tak, že se klíč rušeného prvku přepíše takovou hodnotou klíče, o níž je jisté, že nebude nikdy vyhledávána.

Během vykonávání algoritmu je v každém kroku provedeno porovnání na shodu prvku s hledaným klíčem a také porovnání zda je hledaný klíč větší než klíč právě porovnávaného prvku. Pokud je tento prvek větší než hledaný klíč, je vyhledávání ukončeno, neboť vzhledem k uspořádání prvků tabulky již nemůže být hledaný prvek v další části tabulky obsažen. Situaci znázorňuje obraz 2.1.3.1.



Obr. 2.1.3.1 – Uspořádání prvků v seřazené posloupnosti

Udržování seřazené posloupnosti a vyhledávání v ní urychluje pouze případy neúspěšného vyhledání. Pro ostatní operace je režie spojená se zachováním uspořádání komplikací.

Časová složitost algoritmu je opět O(N). Doby jednotlivých případů vyhledání jsou shodné se sekvenčním vyhledáváním v neseřazeném poli. Jen neúspěšné hledání se díky uspořádanosti urychlí a jeho průměrná doba bude rovna $\frac{N}{2}$.

Stejně jako u algoritmu sekvenčního přístupu k neseřazenému poli můžeme i tento algoritmus mírně vylepšit použitím zarážky umístěné na konci obsazené části tabulky. Tato úprava opět povede ke zjednodušení Booleovské podmínky vyhledávacího cyklu a vynechání dotazu na dosažení konce pole.

Všechny vyhledávací algoritmy se sekvenčním přístupem mají stejný řád složitosti. Jsou velmi jednoduché na implementaci. Sekvenční vyhledávání v neseřazené posloupnosti může být použito bez jakékoliv předcházející úpravy dat vyhledávací tabulky. Díky tomu je tento algoritmus použitelný za všech okolností.

2.2 Nesekvenční vyhledávání v lineární datové struktuře

Algoritmy sekvenčního přístupu dosahují rychlostí vyhledání, které jsou při práci s rozsáhlými tabulkami symbolů příliš nízké. Proto se mnohem častěji setkáme s implementacemi vyhledávacích algoritmů, které k prohledávané oblasti nepřistupují sekvenčně. U nesekvenčních algoritmů o volbě prvků pro porovnání s klíčem nerozhoduje jejich pořadí ve vyhledávací tabulce, ale jsou k porovnávání s hledaným klíčem vybírány na základě jiných pravidel, která jsou pro každý algoritmus specifická. Společným znakem těchto algoritmů je také fakt, že pracují se setříděnou tabulkou symbolů.

2.2.1 Binární vyhledávání

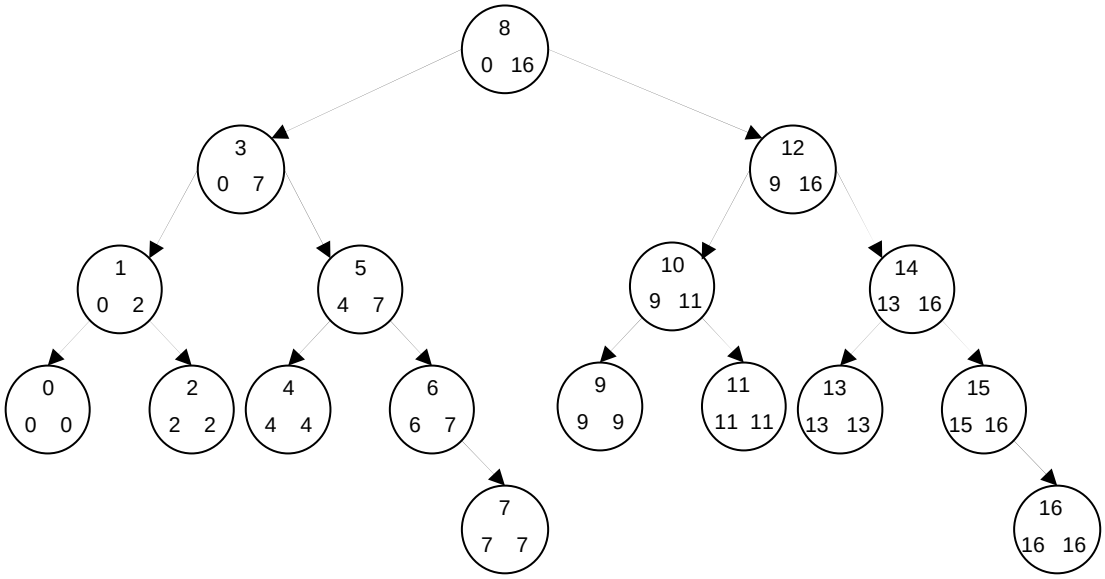
Nejrozšířenějším algoritmem této skupiny je binární vyhledávání. Jeho použitím se výrazně omezuje celkový čas hledání oproti sekvenčnímu vyhledávání. Binární vyhledávání je založeno na standardním paradigmatu „rozděl a panuj“ [2]: rozděl daný problém na několik menších (snadněji řešitelných) podproblémů stejné kategorie, vyřeš jednotlivě podproblémy a kombinací jejich řešení získej řešení prvotního problému. Typickým znakem tohoto přístupu je použití rekurze. U binárního vyhledávání je toto paradigma použito pro zmenšení prohledávané oblasti. Soubor položek se rozdělí na dvě části, určí se ke které části vyhledávaný klíč náleží, a pak se na tuto část soustředíme [2]. Rozhodnutí o tom, ve které části tabulky se hledaný klíč nachází, je učiněno na základě faktu, že binární vyhledávání pracuje s uspořádanou posloupností, a lze tedy porovnáním zjistit, zda se nachází hledaný klíč mezi prvky vpravo či vlevo od porovnávaného. Činnost algoritmu znázorňuje následující schéma:

```
if <je co dělit>
       <rozděl prohledávanou tabulku na dvě poloviny>
else
       <ukonči vyhledávání jako neúspěšné>
if <dělící prvek je shodný s hledaným>
       <ukonči vyhledávání a vrať polohu dělícího prvku>
if <dělící prvek je menší než hledaný>
       <proveď binární vyhledávání nad pravou polovinou tabulky>
else
       <proveď binární vyhledávání nad levou polovinou tabulky>
```

Volba dělícího (a porovnávaného) prvku je poměrně jednoduchá. Volí se prvek, který se nachází uprostřed prohledávané oblasti tabulky. Výraz, kterým se získá jeho poloha, je:

```
iStred = (iLevy + iPravy) / 2
```

kde operátor / představuje celočíselné dělení. Na počátku algoritmu mají indexy levého a pravého okraje hodnotu indexů prvního a posledního prvku v poli. Při následných rekurzivních voláních binárního vyhledávání pak obsahují hodnoty indexů krajních prvků právě prohledávané části tabulky. Použití rekurze však není nezbytné a často se používá i zápis algoritmu binárního vyhledávání ve formě cyklu. Mechanismus postupného rozdělování intervalu a určování středu pro tabulku s 17 prvky zobrazuje rozhodovací strom binárního vyhledávání na obrázku 2.2.1.1



Obr. 2.2.1.1 – Rozhodovací strom binárního vyhledávání

Binární vyhledávání má složitost řádu O(log $_2$ N). Doba přístupu pro případ úspěšného vyhledávání může být v některých případech o něco kratší, než je doba neúspěšného vyhledání. Pro použití binárního vyhledávání je opět zapotřebí udržovat tabulku symbolů setříděnou i po provedení operace vkládání a rušení prvku, což vyžaduje dodatečnou režii.

Algoritmus binárního vyhledávání pracuje primárně s tabulkou symbolů bez duplicitních klíčů. Pokud by byla duplicita povolena, končí vyhledávání na některém z prvků ze souvislé skupiny shodných prvků. Není však určeno na kterém. Binární vyhledávání může být rozšířeno různými způsoby, aby se s problematikou duplicitních klíčů vypořádalo. Je možné přidat čítač prvků se shodným klíčem popř. zajistit obousměrné procházení skupiny prvků se shodným klíčem od místa, kde bylo vyhledávání ukončeno a vrátit dva indexy vymezující položky s klíči shodnými s vyhledávaným klíčem. V tomto případě bude doba pro vyhledání úměrná log $_2$ N + počet nalezených položek [2]. Lze také požadovat, aby vyhledání vrátilo polohu krajního prvku ze skupiny shodných (nejlevějšího nebo nejpravějšího). Pro případ nejpravějšího by platilo:

```
prvek[i].klic == hledanyKlic && prvek[i].klic < prvek[i+1].klic
```

Taková modifikace binárního vyhledávání se nazývá Dijkstrova varianta binárního vyhledávání. Úprava algoritmu spočívá v pozměnění podmínky rozhodující o pokračování algoritmu vpravo či vlevo.

Dijkstrova varianta hledající nejpravější prvek ze skupiny shodných

```
while <je co půlit>{
 <zvol dělící prvek>
 if <dělící prvek je menší nebo roven hledaný>
       <posuň hranice vyhledávání nad pravou polovinou tabulky>
 else
       <posuň hranice vyhledávání nad levou polovinou tabulky>
}
return <prvek na levém okraji tabulky == klíč>
```

Pro variantu vyhledávající nejpravější ze shodných mohou být klíče všech prvků tabulky shodné vyjma nejpravějšího. Varianta vyhledávající nejlevějšího je popsána ve studijní opoře předmětu Algoritmy [3].

Oproti binárnímu vyhledávání je doba přístupu Dijkstrovy varianty shodná pro úspěšné i pro neúspěšné vyhledávání.

2.2.2 Fibonacciho vyhledávání

Fibonacciho vyhledávání je vyhledávací metoda používající k určení polohy právě porovnávaného prvku čísla z Fibonacciho řady [7]. Tato metoda pro určení polohy dělícího bodu používá operaci sčítání, což je výhodné v případech, kdy operace dělení je příliš náročná. Obrazem půlení intervalu u binárního vyhledávání je binární strom, dělení intervalu u Fibonacciho vyhledávání je odvozeno z Fibonacciho stromu.

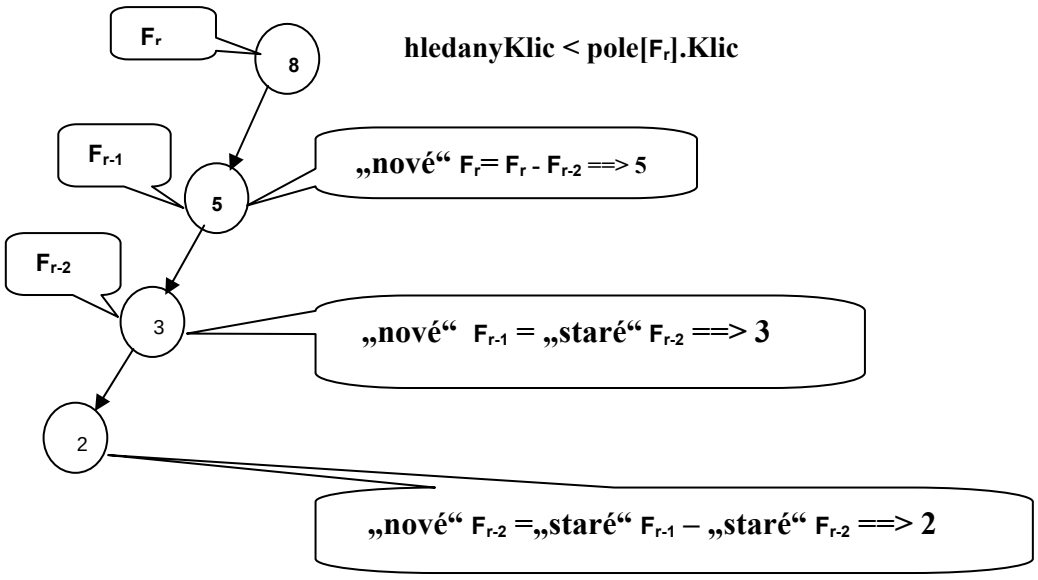
Fibonacciho strom (*F-tree*) je definován pravidly:

- F-tree i-tého řádu sestává z F_{i+1}-1 non-terminálních uzlů a z F_{i+1} terminálních uzlů.
- Je-li i=0 nebo i=1 je strom reprezentován pouze kořenem a současně terminálním uzlem [0].
- Je-li i>=1, pak je kořen stromu reprezentován hodnotou F_i, jeho levý podstrom je F-tree řádu i-1 (F_{i-1}) a pravý podstrom je F-tree řádu i-2 (F_{i-2}), v němž všechny hodnoty uzlů jsou zvýšeny o hodnotu kořene F_i. [12]

Na počátku algoritmu je nutné nalézt Fibonacciho číslo rovnající se velikosti prohledávané tabulky (indexu jejího nejvyššího prvku) a zjistit jeho řád (r). Dále lze popsat algoritmus následujícím postupem [3]:

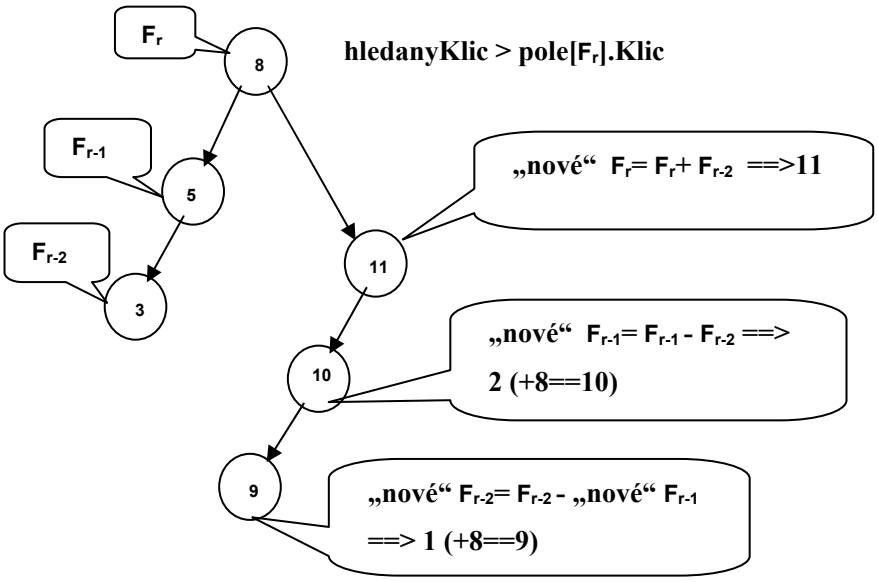
```
<urči Fibonacciho čísla řádu r, r-1 a r-2 (Fr, Fr-1, Fr-2)>
while <prvek na indexu Fr je odlišný od hledaného>&&
      <neprohledali jsme celý strom> {
            if  <hledaný < prvek na indexu Fr>
                  <posun po levé diagonále Fibonacciho stromu>
            else
                  <posun po pravé diagonále Fibonacciho stromu>
         }
         return ! <prohledali jsme celý strom>
```

Posuny po diagonálách Fibonacciho stromu jsou znázorněny na následujících obrázcích. Vyhledávání pokračuje na levé diagonále stromu:



Obr. 2.2.2.1 – Posun po levé diagonále Fibonacciho stromu [3]

Vyhledávání pokračuje na pravé diagonále stromu:



Obr. 2.2.2.2 – Posun po pravé diagonále Fibonacciho stromu [3]

Popisovaný algoritmus předpokládá, že velikost prohledávané tabulky vyhovuje přesně hodnotě některého z čísel Fibonacciho řady. Taková situace však nemusí nastat pro každou vyhledávací tabulku. Pro velikosti tabulek neodpovídající přesně Fibonacciho číslu se používá Sharova metoda.

Sharova metoda postupuje ve dvou krocích:

- V prvním kroku provede rozdělení na největším indexu, který vyhovuje metodě a který je menší než daná velikost.
- Ve druhém kroku zjišťuje, zda je hledaný klíč nalevo nebo napravo od dělicí hodnoty. Když je nalevo, postupuje jako by tabulka měla počet prvků daný rozdělovací (a pro metodu vhodnou) hodnotou. Když je napravo, provede transformaci tabulky posunem začátku pole doprava tak, aby prohledávaná část tabulky měla opět vyhovující počet prvků. [12]

Složitost Fibonacciho vyhledávání patří do třídy $O(\log_2 N)$. Pro velmi rozsáhlé tabulky symbolů dosahuje Fibonacciho vyhledávání o něco lepších výsledků než binární vyhledávání.

2.2.3 Uniformní vyhledávání

Binární vyhledávání je použitelné na všech běžných počítačových systémech používajících pro reprezentaci čísel dvojkové soustavy. V takovém prostředí, kde je operace dělení náročná, je operace půlení intervalu používaná binárním vyhledáváním příliš zdlouhavá, a proto se pro

nesekvenční přístup k prvkům tabulky symbolů používají jiné algoritmy. Jednou z možností je použití uniformního vyhledávání. Následující krok algoritmu je u této metody určen přičtením či odečtením dané odchylky od polohy aktuálního prvku. Výhodné je použít tuto metodu u statických tabulek, kde je možné spočítat hodnoty odchylek (použitím operace dělení) pro danou tabulku předem a odstranit tak jejich výpočet v těle vyhledávacího algoritmu, což vede k jeho urychlení [8]. Tato metoda je používána jen ve specifických případech a zájemce o tuto problematiku se může více dozvědět v [3] nebo [12].

2.3 Vyhledávání ve stromové struktuře

Stromy jsou v počítačovém světě široce používané datové struktury. Jsou však používány hojně i v ostatních oborech pro vyjádření struktury informací, a proto je jejich princip poměrně známý. V problematice vyhledávání je nejrozšířeněji používanou stromovou strukturou binární strom.

Binární strom je orientovaný kořenový strom, pro jehož každý uzel platí, že může mít nejvýše dva syny. Pojmy jako orientovaný graf, kořenový strom, uzel jsou pojmy z teorie grafů a pojednává o nich každá publikace zabývající se touto tematikou [9].

Základní operace implementované stromovými tabulkami jsou vkládání, rušení a hledání.

2.3.1 Binární vyhledávací strom

Binární vyhledávací strom (BVS) je specializací binárního stromu. Jeho uzly splňují následující vlastnosti [10]:

Každý uzel obsahuje hodnotu klíče.

Uzly stromu jsou podle hodnoty klíče uspořádány a to tak, že pro každý uzel platí:

Všechny klíče uzlů v jeho levém podstromu jsou menší než hodnota klíče v uzlu.

Všechny klíče uzlů v jeho pravém podstromu jsou větší než hodnota klíče v uzlu.

Binární vyhledávací strom může být efektivně převeden na seřazenou lineární strukturu pouhým průchodem inorder [11]. Díky uspořádání stromu je vyhledávání v BVS velmi rychlé.

Vyhledávání v binárním vyhledávacím stromě v podobě rekurze:

Je-li vyhledávaný klíč roven kořeni, vyhledávání končí úspěšným vyhledáním.

Je-li klíč menší než klíč kořene, pokračuje vyhledávání rekurzivně v levém podstromu.

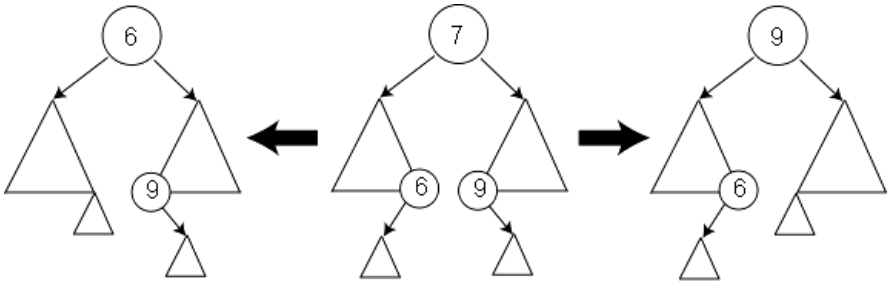
Je-li klíč větší než klíč kořene, pokračuje vyhledávání rekurzivně v pravém podstromu.

Vyhledávání končí neúspěšně, pokud je prohledávaný (pod)strom prázdný. [3]

Mechanismus vyhledávání v binárním vyhledávacím stromě je velmi podobný binárnímu vyhledávání. Jediný rozdíl je ve struktuře vyhledávací tabulky, se kterou tyto algoritmy pracují.

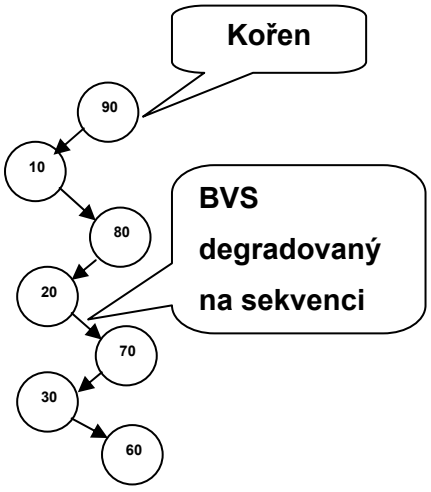
Operace vkládání do tabulky symbolů v podobě binárního vyhledávacího stromu je velmi podobná operaci vyhledání. Algoritmus prochází stromem, jako by hledal položku s klíčem, který má být vložen. Pokud strom již takový uzel obsahuje, bude nalezen a jeho data budou aktualizována novou hodnotou. Pokud však takový uzel ještě ve stromě obsažen není, skončí vyhledávání na uzlu, ke kterému vkládaný uzel připojíme tak, aby bylo dodrženo uspořádání stromu.

Rušení uzlu binárního vyhledávacího stromu musí rozlišovat tři případy rušených uzlů. Pokud je rušen listový uzel, jeho odebrání je jednoduché. Stačí jeho pouhé zrušení. Pokud rušíme uzel, který má pouze jeden podstrom, tento uzel zrušíme a kořen jeho jediného podstromu se připojí na nadřazený uzel rušeného uzlu. Nejsložitějším případem je rušení uzlu, který má oba syny. Z hlediska udržení co nejnižší výšky stromu je vhodný následující postup. Rušený uzel přepíšeme obsahem nejpravějšího uzlu levého podstromu a tento uzel pak zrušíme. Možná je i symetrická varianta s přepsáním rušeného hodnotou nejlevějšího uzlu z pravého podstromu. Situaci znázorňuje obrázek 2.3.1.1



Obr. 2.3.1.1 – Rušení uzlu s oběma syny [10]

Operace vyhledání má časovou složitost $O(\log_2 N)$ v průměrném případě. Doba vyhledání je dána hloubkou, ve které je hledaný uzel. Nevhodnou posloupností operací vkládání a rušení uzlů však může být stromová struktura degradována na lineární a v takovém případě je složitost vyhledání řádu O(N). S tímto problémem se dokáží vypořádat stromové struktury implementující operace vkládání a rušení uzlu, které zachovávají vyváženost stromu. Tyto vyhledávací stromy popisuje následující kapitola.



Obr. 2.3.1.2 – BVS degradovaný na lineární seznam [3]

2.3.2 Modifikace binárního vyhledávacího stromu

Délka vyhledávání ve stromové struktuře záleží na uspořádání stromu. Nejhorší případ neúspěšného vyhledávání je dán nejdelší cestou od kořene k listu stromu [12]. Proto je nejvhodnějším případem strom, u něhož jsou délky všech cest od kořene k listům stejně dlouhé. Pokud chceme zachovat složitost vyhledávání pro všechny případy rovnou $\log_2 N$, je nutné zabránit degradaci binárního stromu na lineární strukturu. V této souvislosti se hovoří o vyvážených binárních stromech.

> ”Dokonale vyvážený binární strom je strom, pro jehož každý uzel platí, že počet uzlů v jeho pravém a levém podstromě se liší maximálně o 1.” [12]

Dokonale vyvážený strom se nazývá také váhově vyvážený. Jeho udržování pro dynamické tabulky je však velmi pracné. Proto je v praxi používán jiný druh vyváženosti:

> “Výškově vyvážený binární vyhledávací strom je strom, pro jehož každý uzel platí, že výška jeho dvou podstromů je stejná nebo se liší o 1. “ [3]

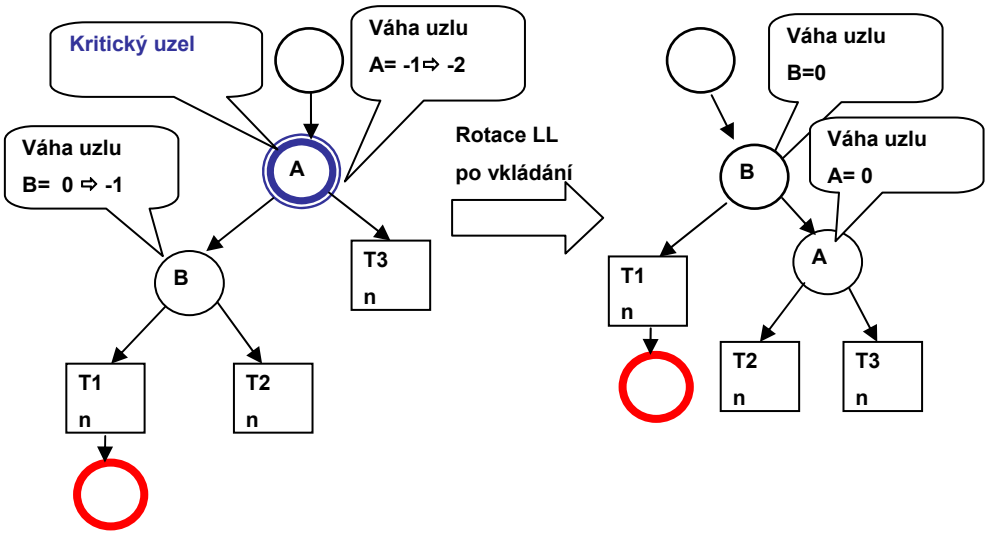
2.3.2.1 AVL strom

AVL strom je výškově vyvážený binární vyhledávací strom se samovyvažující se schopností. Mechanismus obnovení výškové vyváženosti používá dodatečné informace uložené v každém uzlu AVL stromu. Tato informace se nazývá váha a vyjadřuje vztah mezi výškou levého a pravého podstromu uzlu. Váha je dána jako rozdíl těchto výšek a uzel je označován jako vyvážený, pokud platí, že absolutní hodnota jeho váhy je ≤ 1. Při porušení rovnováhy je nutné nalézt tzv. kritický uzel a vyváženost obnovit jednoduchou operací rotace.

> “Kritický uzel je nejvzdálenější uzel od kořene, v němž je v důsledku vkládání nebo rušení porušená rovnováha.“ [3]

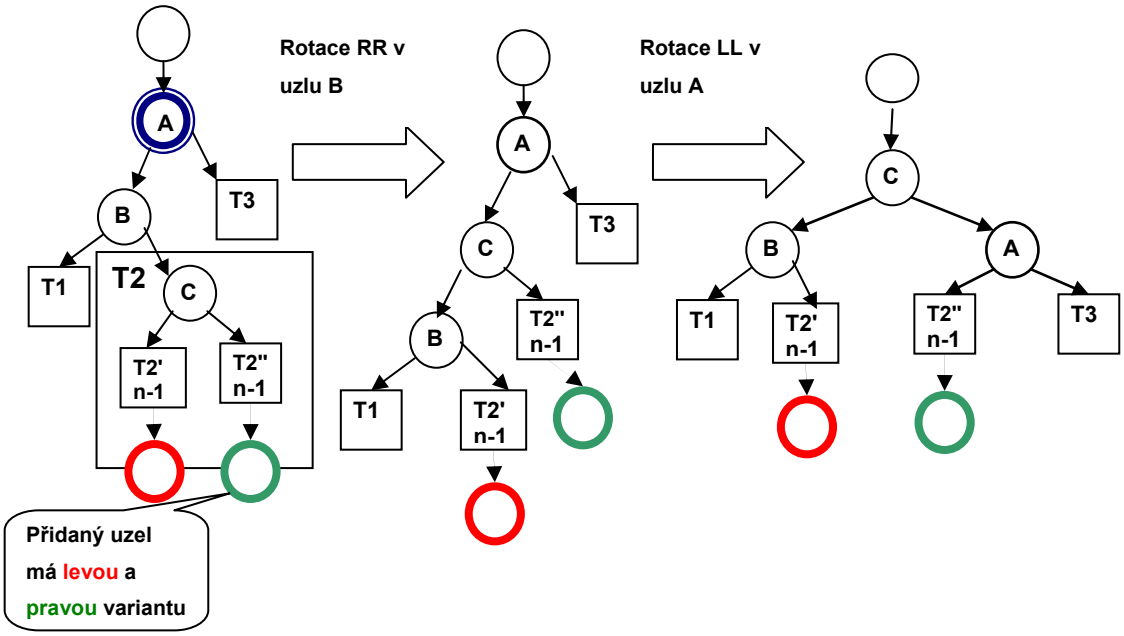
Rotace jsou používány jak po operaci vkládání tak po operaci rušení. Můžeme provádět jednoduché RR-rotace, LL-rotace nebo dvojité LR-rotace, RL-rotace. Při rotaci je nutné aktualizovat koeficient vyváženosti každého rotovaného uzlu [13].

Jednoduchá rotace LL koriguje porušení vyváženosti vložením nového (v obr. červený) uzlu. Vznikne kritický uzel (v obr. modrý). Kritický uzel i jemu podřízený levý uzel se posunou zleva doprava. Obdélníčky T_i představují podstromy s výškou označenou symbolem n [3]. Znázornění operace je na obrázku 2.3.2.1.1



Obr. 2.3.2.1.1 – Rotace LL po operaci vkládání

Rotace RR je zrcadlovým obrazem rotace LL. Dvojité rotace DLR a DRL jsou vytvořeny jako kombinace jednoduchých rotací a jsou opět navzájem zrcadlově souměrné. Všechny rotace jsou prováděny v konstantním čase vzhledem k počtu uzlů stromu.



Obr. 2.3.2.1.2 – Dvojitá rotace DLR jako kombinace obou jednoduchých rotací

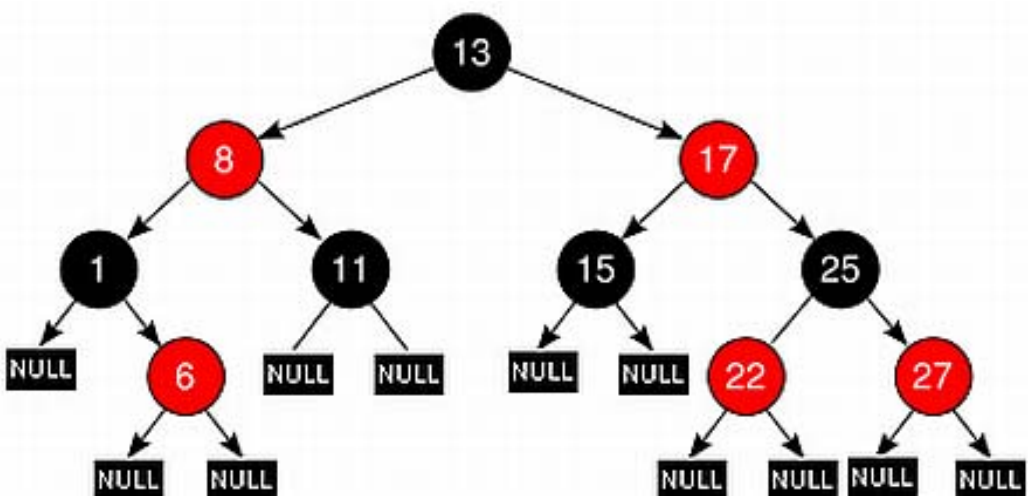
Objevitelé AVL stromu, G. M. Adelson-Velsky a E. M. Landis, dokázali, že AVL strom bude maximálně o 45 % vyšší než dokonale vyvážený strom, složený ze stejných vrcholů [13]. Protože AVL strom nemůže degradovat je doba vyhledání pro průměrný i nejhorší případ řádu $O(\log_2 N)$.

2.3.2.2 Červeno-černý strom

Červeno-černý strom je stejně jako AVL strom binární vyhledávací strom, který sám udržuje svoji výškovou vyváženost. Oproti AVL stromu nevyužívá informace o vyváženosti uzlu v podobě hodnoty jeho váhy, ale uzly stromu jsou „obarvovány“. Pro červeno-černé stromy platí:

- Každý uzel je buď červený, nebo černý.
- Kořen je černý.
- Dva červené uzly se nesmí vyskytovat „nad sebou“, tj. červený uzel má jedině černé potomky.
- Na cestě od kořene do libovolného uzlu je stejný počet černých uzlů. (Vnější uzly, tzv. Null, pokládáme za černé.) [14]

Jeden výskyt červeno-černého stromu je znázorněn na obr. 2.3.2.2.1



Obr. 2.3.2.2.1 – Ukázka červeno-černého stromu [14]

Omezení kladená na červeno-černé stromy zaručují, že nejdelší cesta od kořene k listu bude maximálně dvakrát delší než nejkratší cesta z kořene k listu. Tato pravidla navíc nejsou natolik přísná jako u AVL stromu, a proto není vyžadována tak častá úprava struktury stromu. Navíc pokud je úprava nutná, nejde jen o složité rotace, ale převažuje přebarvování uzlů, které je snadné. Díky těmto vlastnostem jsou červeno-černé stromy často používanou vyhledávací strukturou v aplikacích, které vyžadují rychlé vykonávání všech operací nad vyhledávacím stromem. Konkrétní postupy ustanovení vyváženosti jsou dostupné na [14].

Časová složitost řádu $O(\log_2 N)$ a přesně definovaná nejdelší doba vyhledání $2*\log_2 N$ jsou parametry dané pravidly platnými pro červeno-černé stromy.

2.4 Tabulky s rozptýlenými položkami

Tabulky s rozptýlenými položkami (TRP) jsou vyhledávací metoda, která se používá v situacích, kdy lze předem odhadnout maximální velikost vyhledávací tabulky, ale množina všech možných klíčů, které by se v ní mohli vyskytnout, je příliš velká.

TRP jsou založeny na principu tabulek s přímým přístupem nebo také na vyhledávání s indexovanými klíči [2]. Pokud lze vytvořit vyhledávací tabulku tak velkou, aby mohla pojmout všechny možné hodnoty klíče, lze využít mapovací funkce f, která na základě hodnoty klíče určí polohu prvku s tímto klíčem.

$f(K_i) = i$ pro $i = 0 .. n$, kde n je počet prvků množiny klíčů [12].

Ve vyhledávací tabulce tak bude prvek s klíčem K_i uložen na indexu i. Pokud prvek v tabulce není, je jeho pozice označena parametrem „volno“. Pro tabulky s přímým přístupem by pak platilo, že vyhledání je provedeno v konstantním čase 1, díky jedinému porovnání, které by jen zjistilo, zda na indexu daném mapovací funkcí prvek je či není.

Vytvoření tabulky, která by pojala všechny možné klíče, však nebývá uskutečnitelné. Velikost tabulky se omezí na maximální předpokládaný počet prvků. Hodnoty klíčů jsou pak do této tabulky namapovány rozptylovací (hashovací) funkcí. Ta transformuje klíč na index do vyhledávací tabulky. Hodnota indexu musí být v daném intervalu rozsahu pole [3]. Jelikož je počet možných klíčů větší než velikost tabulky, dochází k situaci, kdy jsou dva nestejné klíče namapovány na stejné místo tabulky. Takovému jevu se říká kolize [5]. Prvkům s klíči mapovanými do stejného místa tabulky se pak říká synonyma.

Mapovací funkce musí mít následující vlastnosti:

- Nemůže být příliš složitá, aby se tíha prohledávání množiny údajů zbytečně nepřenášela na časově náročný výpočet $f(K)$.
- Aby se zamezilo kolizím, musí nestejným hodnotám klíče odpovídat odlišné indexy buněk v tabulce (prvky v tabulce by měly být rozmístěny rovnoměrně).
- Použití funkce f by mělo zajistit rovnoměrné a náhodné rozmístění prvků v tabulce. [5]

Její nalezení však bývá obtížné především pro nenumerické klíče. V prvním kroku se nenumerický klíč převede na numerickou hodnotu (např. použitím kódování znaků). Ve druhém kroku pak převedeme tuto hodnotu na číslo spadající do intervalu indexů pole [3]. Funkce pro získání numerické hodnoty nejvíce ovlivňuje počet kolizí. Její volba je závislá na vlastnostech množiny hodnot klíčů [12]. Převod na hodnotu z intervalu indexů se nejčastěji provádí operací modulo (zbytek po celočíselném dělení).

Problém kolizí se v TRP řeší použitím indexsekvenčního přístupu [12]. Při vkládání do tabulky se pomocí rozptylovací funkce zjistí index, na který má být prvek vložen. Pokud je tato pozice volná dojde k vložení prvku. Při stavu obsazeno je vkládaný prvek vložen do seznamu synonym. Při vyhledávání je situace obdobná. Z klíče je určena poloha v poli a následně je prohledán seznam synonym na této pozici. Pokud je seznam prázdný nebo v něm hledaný prvek není obsažen, končí vyhledávání neúspěchem.

Seznam synonym lze řešit dvěma způsoby, které se liší v přístupu k pravidlům řetězení jednotlivých synonymních položek:

Explicitní zřetězení synonym - každý prvek obsahuje adresu následníka

Implicitní zřetězení synonym - adresa následníka se určí z adresy prvku [12]

2.4.1 TRP s explicitním zřetězením synonym

Explicitní zřetězení synonym je v praxi implementováno lineárně vázaným seznamem. Vyhledávací tabulka pak tvoří pole hlaviček seznamů (i prázdných).

Vkládání do takové vyhledávací tabulky probíhá ve dvou krocích. Nejprve je rozptylovací funkcí určen jeden ze seznamů, do kterého má být prvek vložen, a následně je prvek vložen do tohoto seznamu (na počátek či konec).

Vyhledání probíhá obdobně jako vkládání. Rozptylovací funkce určí seznam, kde by se měl hledaný prvek nacházet, a tento je následně prohledán sekvenčně.

Maximální doba vyhledání je rovna době prohledání nejdelšího ze seznamů synonym. Proto je nejvhodnější tabulka s rozptylovací funkcí, která tvoří co největší počet co nejkratších seznamů synonym. Pokud bychom chtěli urychlit neúspěšná vyhledávání, mohli bychom seznamy udržovat seřazené.

2.4.2 TRP s implicitním zřetězením synonym

TRP s implicitně zřetězenými synonymy ukládá všechny prvky (první prvky i synonyma) tabulky do jednoho pole. Velikost celého pole musí být navržena tak, aby nedošlo během používání takovéto vyhledávací tabulky k jejímu úplnému zaplnění a tím např. k pádu aplikace.

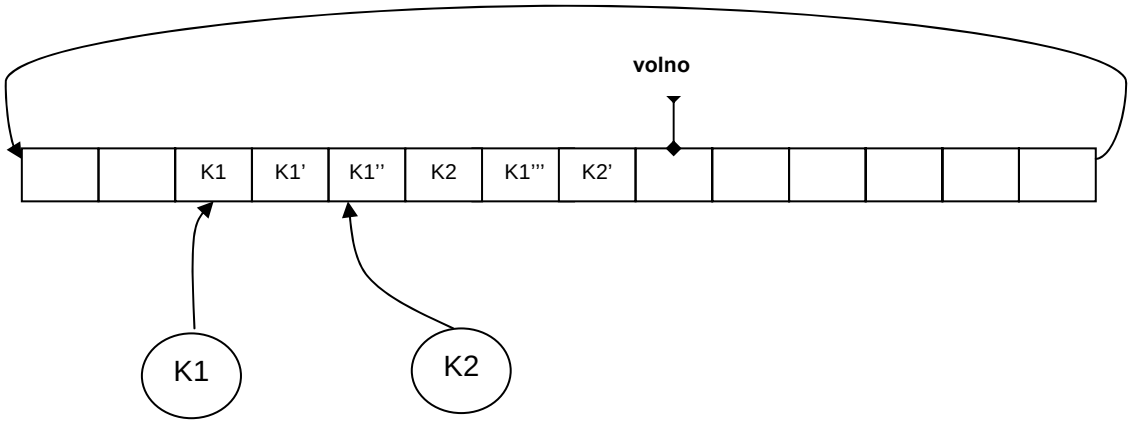
Operace vkládání do tabulky v případě zjištění kolize vloží prvek na nejbližší volnou pozici. To je v ideálním případě pozice vedlejší. Tato metoda tedy používá krok mezi synonymy roven 1. Seznam synonym tvoří po sobě jdoucí prvky a je ukončen první prázdnou pozicí. Proto je nutné přistupovat k poli jako ke kruhovému seznamu a při vytváření tabulky je navíc nutné nastavit všechny její pozice jako volné.

Vyhledávání probíhá stejně jako vkládání. Je však nutné ošetřit případ neúspěšného vyhledávání v zcela zaplněné tabulce, kdy prohledávání projde celým kruhovým seznamem a

nenalezne žádnou volnou pozici, na které by ukončilo svůj běh. Toho lze dosáhnout např. kontrolou dosažení výchozího indexu.

Operace rušení prvku se pro TRP s implicitním řetězením většinou neimplementuje nebo se používá metoda zaslepení.

Při manipulaci s tabulkou symbolů může dojít i k situaci, kdy se dva seznamy synonym promíchají, jak to znázorňuje obrázek 2.4.2.1.



Obr. 2.4.2.1 – Dva promíchané seznamy synonym

Při kroku 1 mají synonyma tendenci vytvářet v poli shluky. To znesnadňuje hledání volných míst pro vkládání a také to prodlužuje dobu neúspěšného hledání. Je výhodnější použít krok větší než 1. Krok však nesmí být dělitelem velikosti pole, neboť pak by neměl možnost projít všechny prvky pole. Toho se nejsnadněji dosáhne tak, že se velikost pole pro TRP položí rovna prvočíslu. Pak libovolný zvolený krok bude mít přístup na všechny pozice [12].

Krok nemusí být pouze konstantní pro všechny seznamy synonym. Jeho hodnota se může získat jako výsledek jiné rozptylovací funkce. Taková metoda se nazývá TRP s dvojí rozptylovací funkcí.

Index, na kterém vyhledávání (i vkládání) začíná, je dán první rozptylovací funkcí, stejně jako tomu bylo v předchozích případech. Velikost kroku je pak dána jako hodnota druhé rozptylové funkce pro hodnotu klíče.

Průměrný počet pokusů nutných k nalezení klíče K je pro případ úspěšného vyhledání roven $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$ a pro neúspěšné vyhledání roven $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$,

kde α označuje faktor naplnění tabulky. Tyto vzorce platí jen teoreticky za předpokladu, že rozptylovací funkce rozmisťuje hodnoty rovnoměrně [5]. Z uvedených vzorců je patrné, že nesmí dojít k úplnému zaplnění tabulky (α=1). Optimální je hodnota 0,6 – 0,7 [3].

3 Studijní opora

Algoritmy by mohly být popisovány slovně, ale lidský jazyk je plný nepřesností, slov s mnoha významy i spoustou synonym. Proto se k popisu a vysvětlení algoritmů nejčastěji používají schémata a především jejich zápis v nějakém programovacím jazyce. Programovací nástroje se však v průběhu doby mění velmi rychle. Vzhledem k této nestálosti je proto mnohem cennější pochopení principu činnosti algoritmu než jeho přesné zapamatování si v takové formě, v jaké je prezentován.

Tato myšlenka je jistě pravdivá, avšak ne vždy zcela vítaná. Mnoho studentů má s tímto přístupem k studiu vyhledávacích algoritmů potíže a pokud algoritmus nevidí zapsán ve svém oblíbeném programovacím jazyce nejsou schopni jeho činnost pochopit.

Vzhledem k faktu, že celé studium na Fakultě informačních technologií bylo po dlouhou dobu založeno na programovacím jazyku Pascal, byly i další předměty, včetně Algoritmů, vyučovány s použitím tohoto jazyka. V roce 2004 však systém studia přešel k výuce využívající jazyk C, ale předmět Algoritmy stále využívá výukové materiály demonstrující činnost vyhledávacích metod na programech napsaných v jazyce Pascal. To vedlo k nelibosti některých studentů. Ale také ke zrodu této bakalářské práce.

Součástí zadání je i požadavek na úpravu studijní opory pro předmět Algoritmy. Tato úprava spočívá především v úpravě zdrojových kódů, které jsou její součástí a jejich přepsání do jazyka C. Dále je také mírně modifikován samotný text, tak aby reflektoval mírné odlišnosti mezi Pascalem a jazykem C.

Rozdílů mezi těmito dvěma programovacími jazyky, které se promítly do kódů popisujících vyhledávací algoritmy, nebylo mnoho. Drobné odlišnosti jsou v syntaxi obou jazyků, logických operátorech a absenci datového typu boolean v jazyce C. Největším problémem byl různý způsob indexování prvků v poli a předávání parametrů funkcí hodnotou vs. odkazem.

Indexování prvků pole se v Pascalu děje zpravidla od 1 (jiné číslo je možné, avšak nepoužívá se). První prvek má tedy index 1 a index posledního je roven N, kde N je velikost pole (počet prvků). Naopak v jazyce C se indexování začíná 0. První prvek má tedy index 0 a poslední prvek leží na indexu N-1. Počet prvků je stále roven N. S ohledem na tento fakt byly upraveny algoritmy většiny vyhledávacích algoritmů, které pracují s vyhledávací tabulkou reprezentovanou polem.

Druhým problémem bylo předávání struktur, které se v jazyce C děje hodnotou. To zabraňuje provedení změny obsahu struktury uvnitř funkce tak, aby se tato změna projevila i mimo ni. To je problém operací vkládání a rušení, které obsah vyhledávací tabulky pozměňují. Aby se změna provedená uvnitř funkce projevila i mimo její rámec, je nutné modifikovanou

strukturu předávat odkazem (předávat její adresu) a uvnitř funkce je nutné provést dereferenci – zpřístupnění obsahu pomocí adresy. Více k tomuto tématu nalezneme v [15].

Algoritmy byly odladěny a jejich funkčnost ověřena v aplikacích k tomu určených. Pro algoritmy pracující s tabulkou symbolů v podobě lineární struktury byla vytvořena okenní aplikace v jazyce C++, která se oproti demonstračním programům nezaměřovala na průběh algoritmů, a proto nebylo nutné do jejich struktury zasahovat a mohl být odladěn přesně ten kód, který je obsažen ve studijní opoře. Navíc oproti animačnímu programu je zde implementováno také vkládání a rušení prvků. Kód těchto operací je ve studijní opoře také uveden, a tak bylo nezbytné odladit i jej. Pro testování algoritmů nad binárním vyhledávacím stromem byl použit testovací program, který je používán v předmětu Algoritmy ke kontrolování domácích úkolů na toto téma. Jeho autorem je Martin Tuček z Fakulty informačních technologií VUT v Brně. Jde o konzolovou aplikaci pracující s jednou reprezentací binárního vyhledávacího stromu, nad kterým jsou prováděny jednotlivé operace vkládání, vyhledávání a rušení uzlů. Tyto operace jsou pevně dány kódem aplikace. Není umožněn žádný zásah uživatele. Pro testovací účely je však tato aplikace dostačující.

Text studijní opory byl zbaven chyb z kategorie překlepů a několika drobných nedostatků v obsahové části. Byly také dodány poznámky upozorňující na odlišnosti mezi jazykem Pascal a C, které jsou zmíněny výše.

Nově vzniklý text je téměř rovnocenný původnímu a je jen volbou studenta, kterému z nich dá přednost.

4 Demonstrační aplikace

Cílem demonstrační aplikace je jasné a přehledné znázornění činnosti jednotlivých vyhledávacích algoritmů a to nejen vizualizací vyhledávací tabulky, ale i doplněním informací pro konkrétní algoritmus významných. K tomuto účelu jsem použil jednak barevné odlišování významných prvků a také výpis hodnot některých proměnných ovlivňujících vykonávání algoritmu.

Po prvních pokusech v rámci semestrálního projektu, ve kterém jsem se snažil vytvořit jednotné prostředí pro všechny vyhledávací algoritmy, jsem se rozhodl vyčlenit algoritmy pracující se stromovou strukturou do samostatné aplikace. K tomuto kroku jsem se odhodlal z důvodu odlišného vykreslování jednotlivých prvků a také s ohledem na jiný způsob tvorby celé tabulky symbolů.

Pro algoritmy nad sekvenční tabulkou symbolů je významný popis jejich přístupu k prvkům vyhledávací tabulky během operace vyhledávání. Operace vkládání a rušení prvků jsou méně podstatné a jejich animace se téměř neliší od operace hledání. Ani struktura tabulky se jejich aplikací výrazně nemění, neboť se pouze mění její délka.

U vyhledávací tabulky v podobě binárního vyhledávacího stromu jsou významné všechny tři operace i jejich postupná aplikace a způsob jakým ovlivňují strukturu stromu.

4.1 Volba obsahu

V první fázi vývoje programů bylo nutné určit, které algoritmy budou v aplikacích obsaženy.

Pro aplikaci demonstrující činnost vyhledávacích algoritmů pracujících s vyhledávací tabulkou v podobě lineární struktury jsem se rozhodl implementovat všechny výše popsané metody kromě uniformního vyhledávání. To patří do okrajové skupiny vyhledávacích algoritmů a ani při výuce není jeho přesná znalost vyžadována.

U vyhledávací tabulky se stromovou strukturou byla volba jednoduchá. AVL stromy a ostatní modifikace binárního vyhledávacího stromu jsou samostatným zadáním jiné bakalářské práce, a proto jsem implementoval pouze binární vyhledávací strom a všechny tři základní operace nad ním prováděné.

Jako množinu hodnot klíčů jsem pro obě aplikace zvolil celá čísla v rozsahu od 0 do 99. Pro jednoduchou ukázku je možnost minimálně sto prvků dostačující, číselné hodnoty se snadno porovnávají a usnadní se i jejich vykreslování.

4.2 Volba vývojového prostředí

Volba vývojového prostředí a nástrojů použitých k vývoji aplikace mi přišla velmi podstatná. Již několikrát jsem se setkal s problémem, že víme co chceme vytvořit, ale nemáme k tomu potřebné nástroje. Dle mého názoru není nic horšího než, když se práce na projektu zastaví z důvodu nedostatku technologií. Proto jsem se nad volbou vývojového prostředí pozastavil a nejprve jsem si vytvořil co nejkomplexnější návrh samotných aplikací, abych se ujistil, že vybraný nástroj je schopen splnit všechny mé požadavky.

Během svého studia jsem se seznámil s programovacími jazyky C a C++. Z důvodu nedostatku času na studium nového programovacího jazyka jsem se zaměřil na tyto dva.

Z mého návrhu a povahy aplikací bylo zřejmé, že bude nutný mechanismus vykreslování grafických objektů v okně aplikace. Dvourozměrná animace mi připadla jako dostačující. Bylo také jisté, že aplikace bude tvořena oknem, ve kterém bude uspořádáno uživatelské rozhraní a oblast pro animaci.

Multiplatformní knihovna Glut, se kterou jsem se seznámil v předmětu Základy počítačové grafiky, by dokonale splňovala nároky na vykreslovací schopnosti. Avšak způsob, jak ji spojit s příjemným uživatelským rozhraním, by nebyl příliš programátorsky pohodlný.

Jako vhodné se z pohledu tvorby uživatelských rozhraní jevily tzv. toolkity jako Fltk, Gtk+ popř. jiný. S těmi jsem získal několik zkušeností během práce na projektu grafického editoru v předmětu vyučujícím jazyk C++. Věděl jsem tedy, že i vykreslování grafiky by v nich bylo realizovatelné.

Obdobné zkušenosti jsem měl i s WinApi, které je rozšířením jazyka C pro vytváření okenních aplikací. I zde jsem měl ověřeno, že tento nástroj podporuje funkce pro snadné vykreslování. Výhodou by navíc bylo, že by byla aplikace napsána přímo v jazyce C, a proto by se dalo dosáhnout vysoké použitelnosti již odladěných zdrojových kódů ze studijní opory.

U všech doposud zmíněných nástrojů mě však odrazoval přístup k návrhu uživatelského rozhraní, jehož každá změna se provádí ruční úpravou zdrojových kódů aplikace. Z tohoto pohledu mi nejvíce vyhovovalo vytváření okenní aplikace v prostředí platformy .NET. Vizuální návrhář uživatelského rozhraní obsažený ve vývojovém nástroji Visual Studio poskytuje dokonalou kontrolu nad výsledným vzhledem aplikace a nastavování mnoha vlastností jednotlivých ovládacích prvků přímo z nabídek. Navíc jsem měl s tímto přístupem k tvorbě okenních aplikací nejvíce zkušeností. I tato varianta nabízela široké množství funkcí pro zobrazování grafiky a to ještě lépe a snadněji použitelných než u WinApi. Další výhodou je také bohatá dokumentace k této platformě dostupná [16].

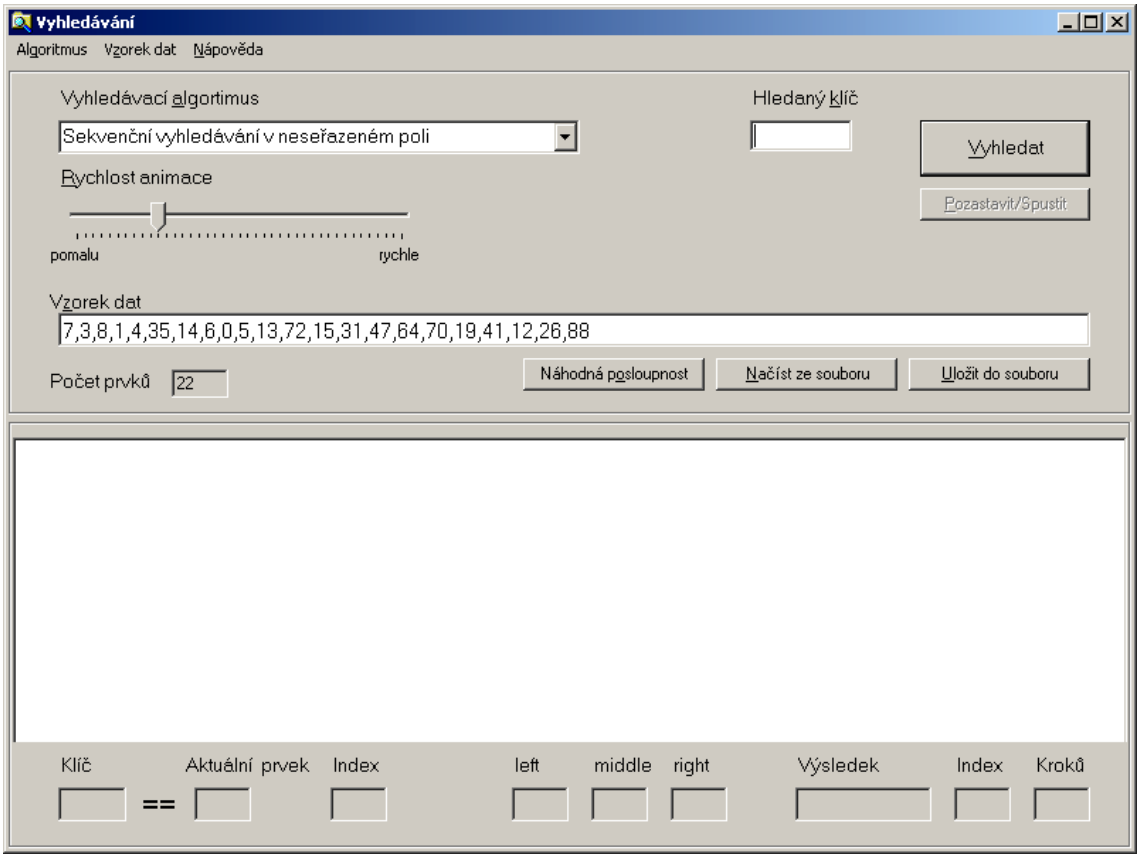
Pro tuto práci jsem si tedy zvolil platformu .NET 2.0 a vývojové prostředí Visual Studio 2003 i za cenu omezení použitelnosti demonstračních aplikací pouze na operačních systémech rodiny Windows s nainstalovaným frameworkem .NET 2.0.

4.3 Animace vyhledávání v sekvenční struktuře

Tuto aplikaci jsem nazval Sekvenční vyhledávání. Program navazoval na můj semestrální projekt. Během jeho vývoje jsem došel k závěru, že původní koncepce není přímo vhodná pro animační účel. Veškerý popis činnosti algoritmu byl založen na výpisu hodnot několika proměnných. To však nebylo dostačující k přehledné ukázce běhu algoritmu, neboť uživatel by musel věnovat pozornost a neustále vyhodnocovat množství jemu předkládaných číselných dat. Proto jsem tuto cestu částečně opustil a zaměřil se více na vykreslování a grafickou stránku popisu algoritmů.

4.3.1 Návrh uživatelského rozhraní

Se změnou přístupu jsem také přepracoval uživatelské rozhraní, které jsem se pokusil zpřehlednit a jednotlivé ovládací prvky umístit tak, aby jejich poloha více odrážela pořadí, v jakém se k nim upírá uživatelova pozornost při práci s aplikací. Další text bude popisovat uživatelské rozhraní zachycené na obrázku 4.3.1.1.



Obr. 4.3.1.1 – UI aplikace Vyhledávání

Hlavní menu zprostředkovává stejné funkce jako jednotlivá tlačítka rozmístěná v okně. Plocha okna je opticky rozdělena na dvě části což by mělo zpřehlednit ovládání.

V horní časti jsou ovládací prvky sloužící k nastavení aplikace před spuštěním vyhledávání, nabídka k výběru vyhledávacího algoritmu, posuvník umožňující nastavení rychlosti animace, textové pole pro ruční zadání položek vyhledávací tabulky. Data není nutné zadávat ručně, ale je možné nechat si vygenerovat náhodnou posloupnost čísel nebo načíst data ze souboru. Pokud uživatel nechce svoji oblíbenou množinu dat zadávat neustále ručně, může si ji uložit do souboru pro pozdější opětovné použití. Podstatným prvkem je textové pole pro zadání vyhledávaného klíče a také tlačítko pro spuštění vyhledávání. Za běhu programu je možné animaci pozastavit tlačítkem „Pozastavit / Spustit“. Pokud je animace pozastavena je umožněno krokování vpřed i vzad celým průběhem pomocí dalších dvou tlačítek. Tato funkce je popsána níže.

V dolní části je umístěna plocha sloužící pro animaci běhu algoritmu a pod ní textová pole, ve kterých se objevují hodnoty podstatné pro vykonávání algoritmu a na závěr i výsledek vyhledávání.

Všechny ovládací prvky jsem se snažil umístit tak, aby jejich poloha odrážela pořadí v jakém jsou použity. Data vyhledávací tabulky jsou vygenerována automaticky po startu programu, proto pokud uživatel nevyžaduje žádnou změnu, nemusí do nich vůbec zasahovat. Nejpodstatnější je výběr vyhledávací metody, zvolení hledaného klíče a spuštění animace.

Během průběhu animace jsou všechny ovládací prvky, které v tento okamžik ztratily svůj význam, nastaveny jako vypnuté.

4.3.2 Implementované algoritmy

Algoritmy implementované programem jsou následující:

- Sekvenční vyhledávání v neseřazeném poli
- Sekvenční vyhledávání v neseřazeném poli se zarážkou
- Sekvenční vyhledávání v seřazeném poli
- Sekvenční vyhledávání v seřazeném poli se zarážkou
- Vyhledávání s rekonfigurací pole podle četnosti vyhledání
- Binární vyhledávání
- Dijkstrova varianta - nejpravější
- Dijkstrova varianta - nejlevější
- Fibonacciho vyhledávání

4.3.3 Změna velikosti okna aplikace

Nejmenší rozměry okna jsou nastaveny na 800*600 obrazových bodů. Toto rozlišení je také minimální, kterým musí disponovat počítač, na kterém má být tato aplikace spuštěna. Jelikož se jedná o aplikaci, jejíž podstata spočívá v přehledném znázornění grafického výstupu, rozhodl jsem se vytvořit program podporující libovolné zvětšení okna aplikace.

Povolení změny velikosti formuláře, který tvoří hlavní okno aplikace je otázkou změny jednoho jeho parametru. Avšak po změně velikosti zůstanou všechny ostatní ovládací prvky v nezměněné poloze vůči levému hornímu rohu a taktéž jejich rozměry se nezmění. Tento nedostatek jsem musel odstranit. Po propátrání několika odkazů na internetu, které vedly na produkty třetích stran, které problém při změně velikosti okna řeší, jsem našel článek popisující použití dvou vlastností společných pro všechny prvky formuláře, jejichž přenastavení efektivně a jednoduše můj problém řešilo [17].

Jde o použití vlastností Dock a Anchor. Přesnější informace a ukázku demonstrující chování objektů po změně těchto atributů lze nalézt v citovaném článku.

4.3.4 Krokování, vykreslování animace

Mezi funkcemi nabízenými oběma programy je také možnost pozastavení samovolně běžící animace a její krokování. Automatické krokování je umožněno použitím komponenty timer, která po uplynutí zadaného časového intervalu spustí nadefinovanou akci a poté opět začne odpočítávat čas. V mých programech je touto akcí funkce pro vykreslení dalšího kroku algoritmu. K dosažení této funkčnosti jsem v původním návrhu používal vlákna a možnosti jejich uspání na jistý časový interval. Při implementaci krokování jsem však narazil na problém kroků zpět.

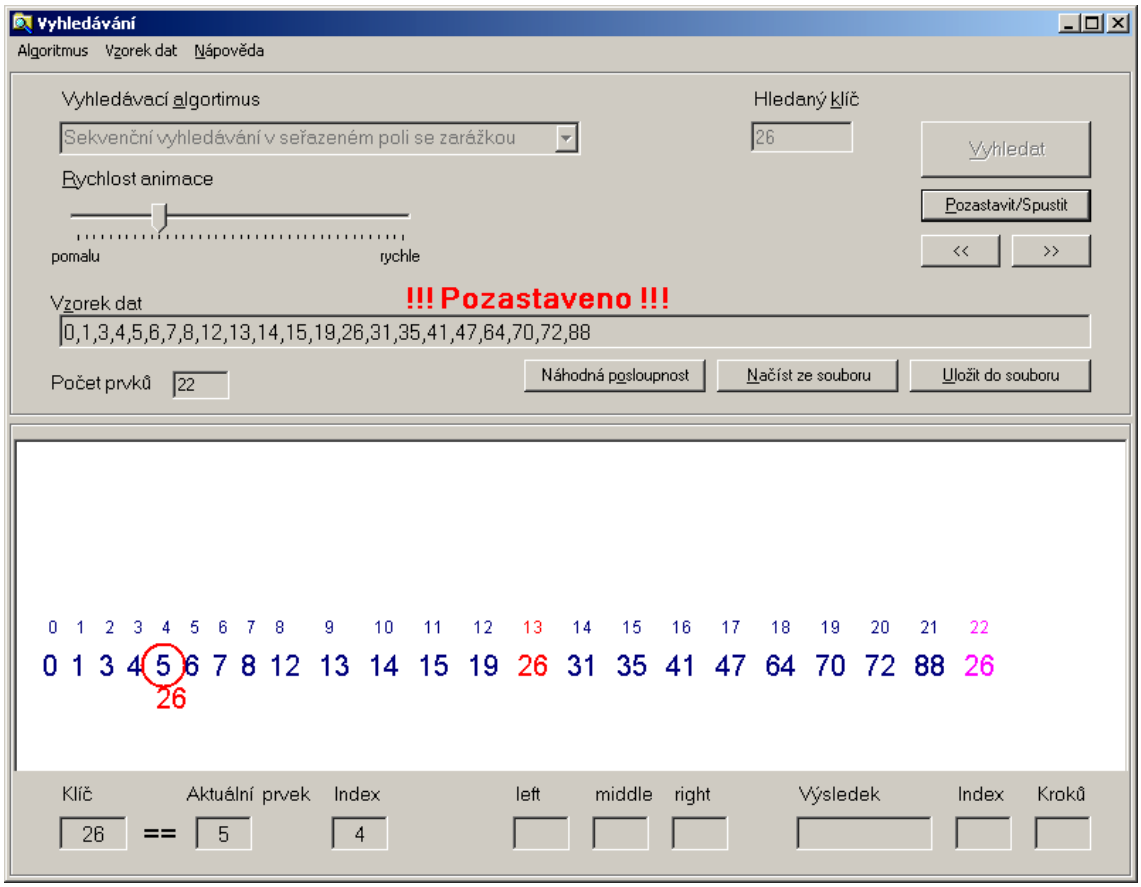
Krok kupředu je pouhým znázorněním následujícího kroku algoritmu (iterace cyklu). Při kroku zpět však nastává problém, jak z aktuálního stavu zjistit stav předchozí a vykreslit jej. Tento problém se dá řešit ukládáním předchozího kroku algoritmu, což by ovšem bylo řešením pouze pro jedno stisknutí tlačítka zpět. Krokování však mělo být umožněno v celém rozsahu průběhu algoritmu. Z tohoto důvodu jsem se rozhodl pro ukládání všech kroků algoritmu a informací nutných k jejich vykreslení do pomocného pole. Po spuštění vyhledání se provede celý algoritmus vyhledání a vloží všechny své kroky do pomocného pole. Vzhledem k malému počtu prvků se vyhledání provede velmi rychle, bez povšimnutí uživatele. Až poté je spuštěna animace, která vykresluje již jen kroky uložené v poli. V tomto poli kroků je možné postupovat vpřed i vzad ať už použitím funkce krokování nebo automatického vykreslování timerem.

Funkce vykreslování jednoho kroku zvolí podle aktuálního rozměru okna a počtu vykreslovaný prvků vyhovující velikost fontu pro vykreslení hodnot prvků. Kromě hodnot klíčů se pro každý prvek vykresluje i jeho index. Pro názornost je použito barevné zvýraznění

významných prvků. Těmi je především hledaný prvek, pokud se v tabulce vyskytuje, tzv. zarážka u algoritmů, které ji používají, a prvek podílející se na záměně polohy u algoritmu s rekonfigurací pole. Dále je zvýrazněn pomocí orámování aktuální prvek. Spolu s animací se při každém kroku vypisují do oblasti pod ní i hodnoty aktuálního prvku a jeho indexu, hledaného klíče a na závěr vyhledávání také výsledek.

Vykreslování průběhu algoritmu probíhá do prvku pictureBoxu. Tato komponenta je pro tento úkon navržena, i když vykreslování grafických primitiv je možné i do jiných prvků formuláře (např. tlačítko). Ve svých aplikacích nekreslím přímo do této oblasti, ale do bitmapy, která je z rozměrů pictureBoxu vytvořená.

Všechny změny jsou nejprve vykresleny do bitmapy a teprve poté je obsah bitmapy vykreslen do pictureBoxu. Tento přístup zabrání případnému problikávání obrazovky. Obrazovka je překreslena také při události změny velikosti okna. Při vykreslení každého kroku je vykresleno celé pole tabulky. Kdyby tomu tak nebylo, zůstávala by za aktuálním prvkem zvýrazněna i stopa jeho předchůdců.



Obr. 4.3.4.1 – Ukázka animace průběhu vyhledávacího algoritmu

4.4 Animace binárního vyhledávacího stromu

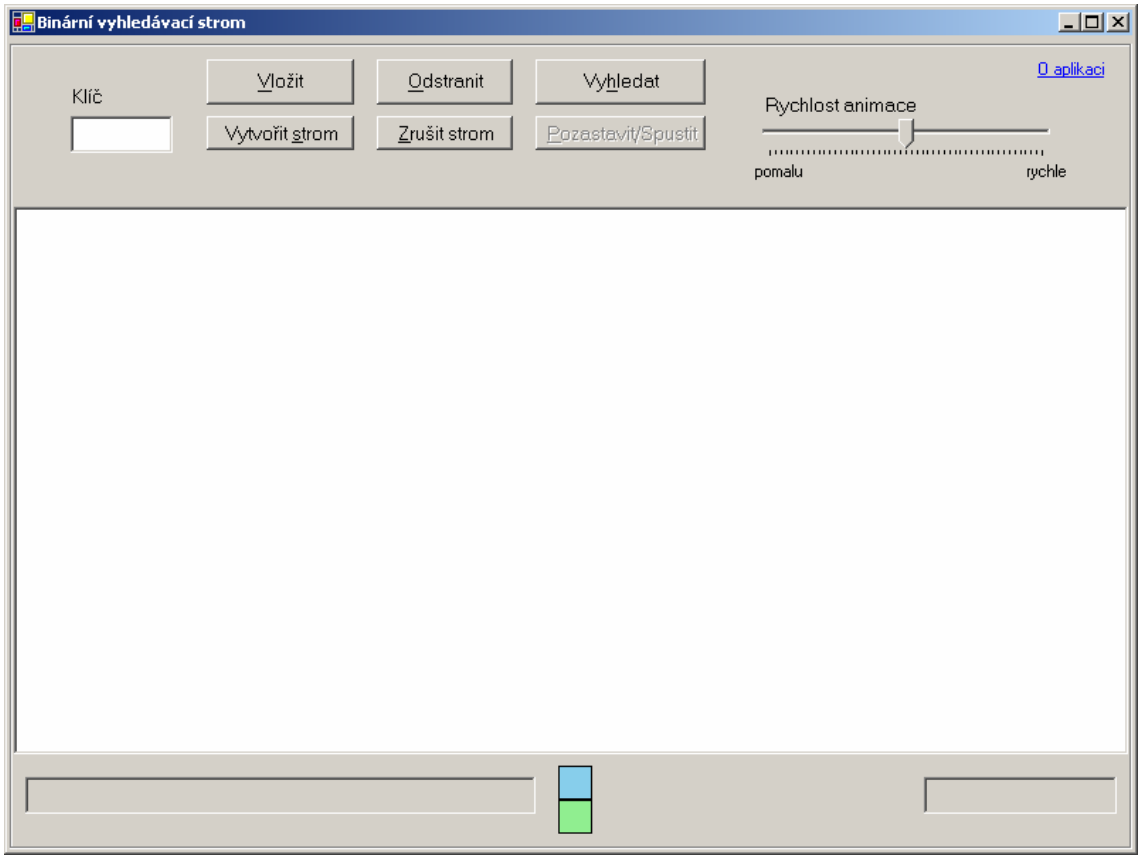
K návrhu animace BVS jsem přistupoval poněkud odlišně. Nenavazoval jsem na žádný předchozí projekt, takže jsem provedl kompletní návrh od samých počátků. Navíc jsem byl bohatší o zkušenosti z implementace předchozí aplikace, a tak jsem se s některými problémy vypořádal snadněji.

Pokusil jsem se o jednodušší uživatelské rozhraní, neboť operace nad stromem nepotřebují tolik ovládacích prvků. Také jsem musel navrhnout způsob vykreslování, který měl v tomto případě používat více zvýrazňování pomocí barev.

4.4.1 Návrh uživatelského rozhraní

Okno programu je opět vizuálně rozděleno na část sloužící k provádění operací nad stromem a část, která je užita k animaci průběhu algoritmů a výpisu dodatečných informací.

Jelikož nad BVS provádíme operace vkládání a rušení prvku, aby se dala znázornit změna struktury po jejich provedení, bylo nutné přidat dvě tlačítka pro tyto operace. Další změnou je absence ovládacích prvků pro jednorázové zadání obsahu celé vyhledávací tabulky uživatelem. To je nahrazeno právě postupnou manipulací s jednotlivými uzly stromu pomocí vkládání a rušení a funkcí vygenerování náhodného stromu.



Obr. 4.4.1.1 – UI aplikace Binární vyhledávací strom

Textové pole pro klíč slouží pro všechny tři operace. Každá z operací vkládání, vyhledání a rušení uzlu se spouští samostatným tlačítkem. Navíc jsou přidány funkce pro vytvoření náhodného stromu a zrušení celého stromu. Opět je použito tlačítko pro pozastavení běhu animace a posuvník pro její urychlování či zpomalení. V režimu pozastavení animace je umožněno její krokování.

Vykreslovací oblast animace je v tomto programu zvětšena, neboť stromová struktura může růst jak do šíře tak do výšky. Informační oblast pod vykreslovací plochou je zachována a doplněna o vysvětlení významu barevných označení uzlů, pro konkrétní algoritmy.

4.4.2 Algoritmus určování polohy uzlu

Problematika vykreslování stromové struktury je poněkud odlišná od vykreslování prvků pole. Je nutné určit obě souřadnice polohy každého uzlu. U pole zůstává jedna souřadnice konstantní pro všechny prvky.

U pole je snadné určit proměnlivou horizontální souřadnici odvozením od indexu prvku. Pro stromovou strukturu toto možné není. Nebo jsem se alespoň domníval, že tomu tak není. Po prostudování několika různých zdrojů, které se problematikou vykreslování stromů zabývají [18], [19], [20] jsem však jednoduchý a přitom dostačující algoritmus nalezl. Komplexní vykreslovací algoritmy mají sice kvalitní výstup, splňující všechna kriteria vykreslování stromů včetně minimální plochy, ale pro moji aplikaci mi jejich implementace přišla zbytečně náročná a jejich činnost složitá. Navíc je u těchto algoritmů složité určit rozměry výsledné kresby ještě před jejich započetím. Proto jsem se rozhodl pro nejjednodušší vykreslovací algoritmus, který je popsán v [20].

Vertikální poloha uzlu je vypočtena z hloubky daného uzlu ve stromu. Horizontální souřadnice je pak odvozena od indexu, který příslušnému uzlu náleží v poli vzniklém průchodem inorder stromem. Z pouhé znalosti počtu uzlů stromu je také možné zjistit celkovou šířku kresby. Tento algoritmus je velmi jednoduchý, takže jeho vykonání je rychlé. Ze základních estetických kriterií jsou splněna všechna. Nedostatkem je, že otec není umístěn v přesném středu mezi svými potomky, a také fakt, že prostor potřebný pro vykreslení není minimální. Domnívám se však, že ani jeden z nedostatků není pro aplikaci překážkou.

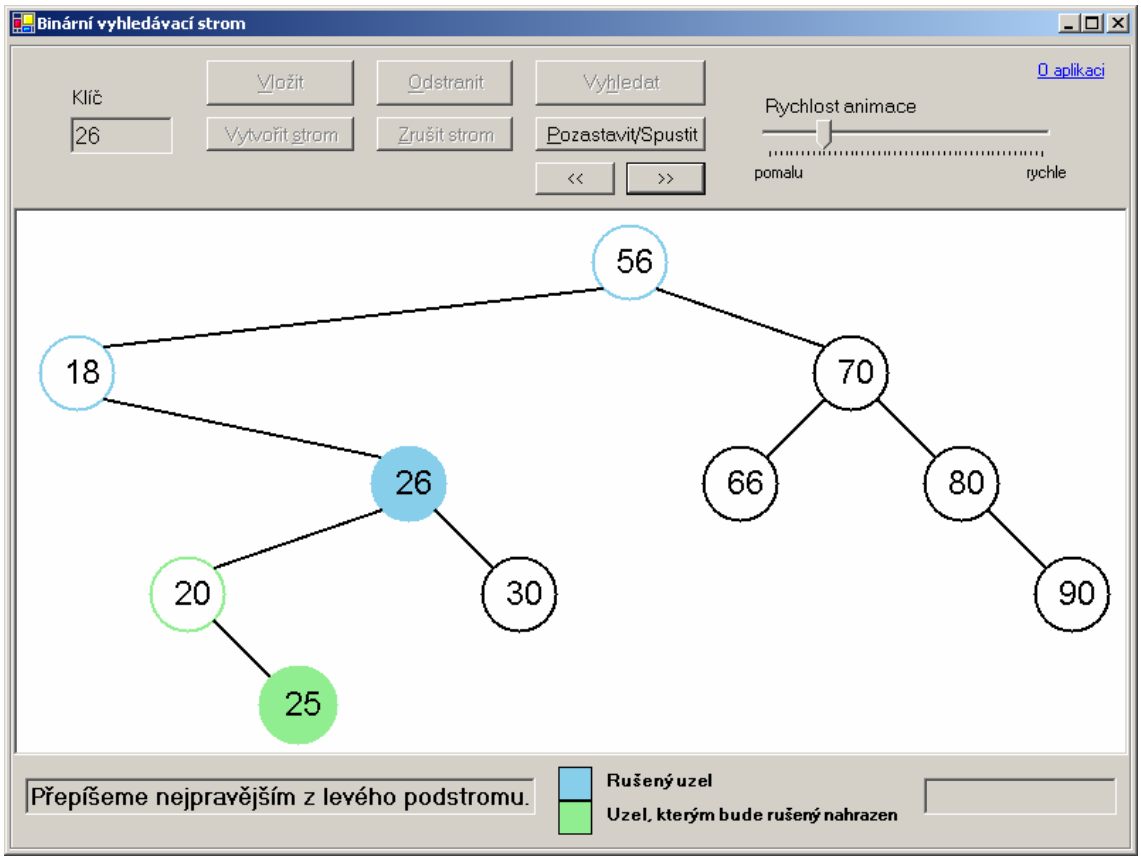
4.4.3 Animace

Animace operací nad BVS je téměř totožná s první demonstrační aplikací. Běh animace je buď automatický s nastavitelnou rychlostí provádění, nebo lze v režimu pozastavení použít mechanismus krokování. Odlišná je datová struktura, do které se ukládají informace o jednotlivých krocích algoritmu. Ta obsahuje více informací pro vykreslování odlišnými barvami.

Jistý problém se vyskytl u animací operací vykládání a rušení uzlu. Neboť zde nemohla být použita technika spuštění algoritmu, sběr jednotlivých kroků a teprve následné vykreslení. Tímto způsobem by se totiž vykreslil až strom po provedení operace. Nebyla by tak patrná změna struktury. Tuto situaci jsem vyřešil použitím funkce hledání. Ta předchází každému vkládání a rušení a demonstruje vyhledávání uzlu, ke kterému je připojen nový uzel operací vkládání, nebo který má být zrušen. Tento způsob animace sice vyžaduje dva průchody stromem (jeden pro vyhledávání, druhý pro příslušnou operaci), ale animován je jen první z nich. Po provedení příslušné operace je vykreslen jen její důsledek na strukturu stromu. Na obrázku 4.4.3.1 je zachycena operace rušení uzlu s klíčem 26, který bude přepsán uzlem s hodnotou 25.

Vykreslování je opět prováděno do objektu pictureBoxu s pomocným mezikrokem vykreslení do bitmapy. Důraz je kladen na použití barev. Pomocí nich jsou zvýrazněny nejen vkládaný, vyhledávaný a rušený uzel, ale změnou barvy orámování je také vykreslena celá cesta průchodu algoritmu stromem. Další barevné odlišení je použito při operaci rušení, pro znázornění hledání uzlu, který nahradí rušený uzel na jeho pozici. Opět je zvýrazněn právě aktuální uzel stromu.

Průchod stromem je doplněn textovým znázorněním operací porovnání aktuálního prvku s hledaným a rozhodování o směru pokračování průchodu.



Obr. 4.4.3.1 – Animace rušení uzlu

5 Závěr

Tato bakalářská práce měla několik cílů. Všechny však měly společnou myšlenku a záměr a to vytvořit doplňkové učební pomůcky, které by usnadnily studentům pochopení problematiky vyhledávacích algoritmů.

Bod zadání týkající se nastudování příslušné části studijní opory byl téměř zbytečný, neboť bez dokonalého seznámení se s problematikou vyhledávání by nebyla další práce na projektu možná.

Existující studijní opora byla zbavena drobných nedostatků v podobě překlepů a některých chyb vzniklých vytvářením metodou „copy/paste“ a algoritmy v ní obsažené byly přepsány do jazyka C. Pro tento účel vzniklo několik drobných aplikací, pomocí nichž byla odzkoušena funkčnost přepsaných algoritmů. Tyto programy jsou považovány spíše za vedlejší produkt celého procesu přepisu kódu do jazyka C. Textová část byla pouze mírně modifikována, aby reflektovala odlišnosti mezi Pascalem a jazykem C. V obsahové části nebyly provedeny žádné změny, které by odlišovaly původní studijní oporu od modifikované verze. Původní text se zdrojovými kódy napsanými v pseudojazyce Pascalovského typu je dále zachován a udržován. Nově vzniklý text pouze nabízí studentům možnost volby jim bližšího programovacího jazyka a tím snad i možnost snadnějšího porozumění předkládaným algoritmům. Vzhledem k tomu, že text přepracované opory je součástí této práce, omezil jsem množství kódu prezentovaného v samotné bakalářské práci. Případný zájemce o přesný zápis vyhledávacích algoritmů v jazyce C je odkázán na tuto modifikovanou studijní oporu.

Vytvoření demonstrační aplikace k podpoře výuky bylo hlavním cílem bakalářské práce. Vzhledem k faktu, že až dosud pro výuku žádné podobné animace využívány nebyly, bylo nutné navrhnout kompletní aplikaci. Výsledkem práce jsou dva programy, které pokrývají rozdílné skupiny vyhledávacích algoritmů. Jejich rozhraní bylo vytvořeno s ohledem na přehlednost a jednoduchost používání a s důrazem především na samotnou animaci činnosti algoritmů. Rozhraní programů byla vytvořena s využitím C++ a platformy .NET 2.0. To umožnilo návrh klasické okenní aplikace s uživatelským rozhraním obsahujícím všechny v dnešní době obvyklé ovládací prvky. Volba jazyka C++ také umožnila využití kódu obsaženého v upravené studijní opoře jen s minimálními změnami. Tyto změny byly spojeny výhradně s nutností získávání informací o průběhu vykonávání algoritmu a nemění nijak jeho vyhledávací funkci.

Součástí řešení bakalářské práce je i seznam doplňkových otázek ke kapitole vyhledávání a seznam testově zaměřených otázek určených k prověření znalostí z této kapitoly formou testu.

Co se časové náročnosti týče bylo pro mě nejnáročnější vytvoření demonstračních aplikací. Vzhledem k odlišnosti vyhledávání v sekvenční a stromové struktuře jsem vytvářel dvě

různá uživatelská rozhraní, což bylo časově náročnější než práce na jedné komplexnější aplikaci. Značnou část času také vyžadovalo seznámení se se zvolenou platformou a studium materiálů týkajících se tvorby okenních aplikací. Snažil jsem se o maximální využití kódu již napsaného pro studijní oporu, a proto jsem měl komplikovanější pozici pro začlenění tohoto kódu do samotné aplikace a grafické znázornění jeho činnosti. Animace průběhu algoritmu nebyla příliš složitá, i když např. volba algoritmu pro určení polohy uzlů stromu vyžadovala jistý čas k nastudování dostupných možností.

Po dokončení projektu jsem si vědom některých nedostatků a slabin své práce. Především demonstrační aplikace bych se znalostmi, které mám nyní, tvořil poněkud odlišně s větším důrazem na jejich komplexnost a pozdější modifikovatelnost. Jako navazující činnost na svoji bakalářskou práci bych si dokázal představit semestrální projekt některého studenta bakalářského studia, který by měl za úkol vytvořit podobnou demonstrační aplikaci zaměřenou na vyhledávaní s využitím tabulek s rozptýlenými položkami.

Práce na bakalářském projektu mě obohatila o mnoho zkušeností. Tento projekt byl prvním větším prověřením mých schopností. Prošel jsem celým procesem vývoje od shromažďování materiálů a jejich studia, přes návrh kompletní aplikace, její implementace, testování a ladění až po tvorbu této technické zprávy a uživatelské dokumentace programů. Zda byla celá snaha úspěšná ukáže až čas a (ne)spokojenost studentů, kteří budou výsledky této práce používat ve svém vzdělávacím procesu.

Literatura

[1] Mickel, B. A. Pascal user manual and report-ISO pascal standard. Berlín, Springer Verlag, 1991.

[2] Sedgewick, Robert. Algoritmy v C, Části 1-4: Základy, datové struktury, třídění, vyhledávání. Praha, SoftPress, 2003.

[3] Honzík, Jan Maxmilián. Algoritmy: Studijní opora předmětu Algoritmy. Dostupný na URL: https://www.fit.vutbr.cz/study/courses/IAL/private/Opora/Opora-IAL-2007-02-RP-verze-3.pdf (červenec 2007).

[4] Algoritmus - Wikipedie, otevřená encyklopedie [online]. Dokument dostupný na URL: http://cs.wikipedia.org/wiki/Algoritmus (červenec 2007).

[5] Piotr Wróblewski. Algoritmy – Datové struktury a programovací techniky. Brno, Computer Press 2004.

[6] Search algorithm – Wikipedia, the free encyclopedia [online]. Dokument dostupný na URL: http://en.wikipedia.org/wiki/Search_algorithm (červenec 2007).

[7] Fibonacci number – Wikipedia, the free encyclopedia [online]. Dokument dostupný na URL: http://en.wikipedia.org/wiki/Fibonacci_number (červenec 2007).

[8] Programujte.cz – Vyhledávání II – Nesekvenční vyhledávání [online]. Dokument dostupný na URL: http://programujte.com/view.php?cisloclanku=2006021602 (červenec 2007).

[9] Gross, J. L., Yellen J. Handbook of graph theory. Boca raton : CRC Press, 2004.

[10] Binary search tree – Wikipedia, the free encyclopedia [online]. Dokument dostupný na URL: http://en.wikipedia.org/wiki/Binary_search_tree (červenec 2007).

[11] Tree traversal – Wikipedia, the free encyclopedia [online]. Dokument dostupný na URL http://en.wikipedia.org/wiki/In-order_traversal (červenec 2007).

[12] Honzík, J. M., Hruška, T., Máčel, M. Vybrané kapitoly z programovacích technik. Brno, Nakladatelství Vysokého učení technického v Brně, 1991. Dokument dostupný na URL: https://www.fit.vutbr.cz/study/courses/IAL/public/materials/scripta.php (červenec 2007)

[13] AVL-strom - Wikipedie, otevřená encyklopedie [online]. Dokument dostupný na URL: http://cs.wikipedia.org/wiki/AVL-strom (červenec 2007).

[14] Red-black tree - Wikipedia, the free encyclopedia [online]. Dokument dostupný na URL: http://en.wikipedia.org/wiki/Red-black_trees (červenec 2007).

[15] Herout Pavel. Učebnice jazyka C, 4. přepracované vydání. České Budějovice, Kopp, 2004.

[16] MSDN Home page [online]. Dokument dostupný na URL: http://msdn2.microsoft.com/en-us/default.aspx (červenec 2007).

[17] Mathew B. K. Designing Resizable Windows Forms in Visual Studio .NET [online]. Dokument dostupný na URL: http://www.devx.com/dotnet/Article/6964/0/page/5 (červenec 2007).
[17] Mathew B. K. Designing Resizable Windows Forms in Visual Studio .NET [online]. Dokument dostupný na URL: http://www.devx.com/dotnet/Article/6964/0/page/5 (červenec 2007).
[18] Kennedy, A. J. Drawing Trees. Dokument dostupný na URL: http://research.microsoft.com/~akenn/fun/DrawingTrees.pdf (červenec 2007)

[19] Tong Luo. TreeDraw A Tree Drawing System. [Diplomová práce]. University of Waterloo. Dokument dostupný na URL: ftp://ftp.cs.ust.hk/pub/dwood/tong/thesis.ps.gz (červenec 2007)

[20] Cruz, I. F., Tamassia R. Graph drawing tutorial. Dokument dostupný na URL: http://www.cs.brown.edu/~rt/papers/gd-tutorial/gd-constraints.pdf (červenec 2007)

Seznam příloh

Příloha 1. Kontrolní otázky k tématu vyhledávání

1) Zapište algoritmus sekvenčního vyhledávání v podobě funkce vracející index nalezeného prvku v případě úspěchu. Při neúspěšném hledání funkce vrací -1. V těle funkce použijte cyklus *for*.

```
int Search(TPol Tab[N], TKlic K);
```

2) Porovnejte pro sekvenční vyhledávání a sekvenční vyhledávání v seřazeném poli doby vyhledání pro všechny možné případy. Ve kterých z nich je vyhledávání v seřazené posloupnosti rychlejší?

3) Zapište algoritmus binárního vyhledávání v iterační i rekurzivní podobě.

4) Nakreslete rozhodovací strom binárního vyhledávání pro pole s 13 prvky.

5) Kolikrát je BVS rychlejší než sekvenční vyhledání v seřazeném poli v tom nejhorším případě, který může nastat?

6) Jak jinak by se u TRP s implicitním řetězením synonym dala ošetřit kruhovost pole než zabráním jedné položky, která musí zůstat neobsazená?

Příloha 2. Příklady pro formulářově orientovanou písemnou zkoušku z okruhu vyhledávacích algoritmů

1) Co je myšleno pod pojmem **zaslepení**?
   a) Jednoduchý algoritmus vyhledávání, který nezohledňuje dodatečné informace o struktuře a uspořádání prvků vyhledávací tabulky, a provádí vyhledávání slepým průchodem tabulkou a porovnává aktuální prvek s hledaným.
   b) Princip implementace operace Delete, při kterém se rušený prvek neruší fyzicky, ale je jeho klíč přepsán hodnotou, která se nebude nikdy vyhledávat.
   c) Vyhledávací algoritmy s nesekvenčním přístupem se zaměřují jen na tu část tabulky, kde se předpokládá výskyt hledaného prvku, a druhou část tabulky neberou pro svoji činnost v úvahu – zaslepí ji.
   d) Situace, kdy je u TRP použita nevhodná hashovaní funkce, jejíž obor hodnot nepokryje celý rozsah tabulky, a tak některé místa tabulky nemohou být nikdy obsazena.

2) Který index vrátí následující funkce při volání s hodnotou K == 3 a tabulkou

1	3	3	3	4	5	6	6	6	6	6	7

```
int Search(TTab T, TKlic K){
      int left=0;
      int right=T.N-1;
      int middle;
      do {
            middle=(left+right)/2;
                  if (K < T.Tab[middle].Klic)
                        right=middle-1;
                  else
                        left= middle+1;
      } while ((K!=T.Tab[middle].Klic) && (right >= left));
      if( K==T.Tab[middle].Klic)
            return middle;
      else
            return -1;
}
```

a) -1

b) 2

c) 3

d) 6

e) 8

f) 10

3) Kterou operaci používá Fibonacciho vyhledávání k určení polohy dělícího bodu?

a) +

b) % (modulo)

c) /

d) *

4) Maximální doba neúspěšného vyhledání je u binárního vyhledávacího stromu v nejhorším případě rovna?

a) 1

b) $\log_2 N$

c) $N * \log_2 N$

d) N

5) Do následujícího úseku kódu doplňte dva řádky tak, aby se jednalo o Dijkstrovu variantu binárního vyhledávání vyhledávající nejpravější ze shodných prvků.

```
....
int left=0;
int right=T.N-1;
int middle;
while (right != (left+1)){
  middle=(left+right)/2;
  <sem doplňte první část>
    left=middle;
  else
    right=middle;
}
return <sem doplňte druhou část>;
```

a) `if (T.Tab[middle].Klic < K) , K==T.Tab[middle].Klic`

b) `if (T.Tab[middle].Klic >= K) , K==T.Tab[middle].Klic`

c) `if (T.Tab[middle].Klic <= K) , K==T.Tab[left].Klic`

d) `if (T.Tab[left].Klic <= K) , K==T.Tab[left].Klic`

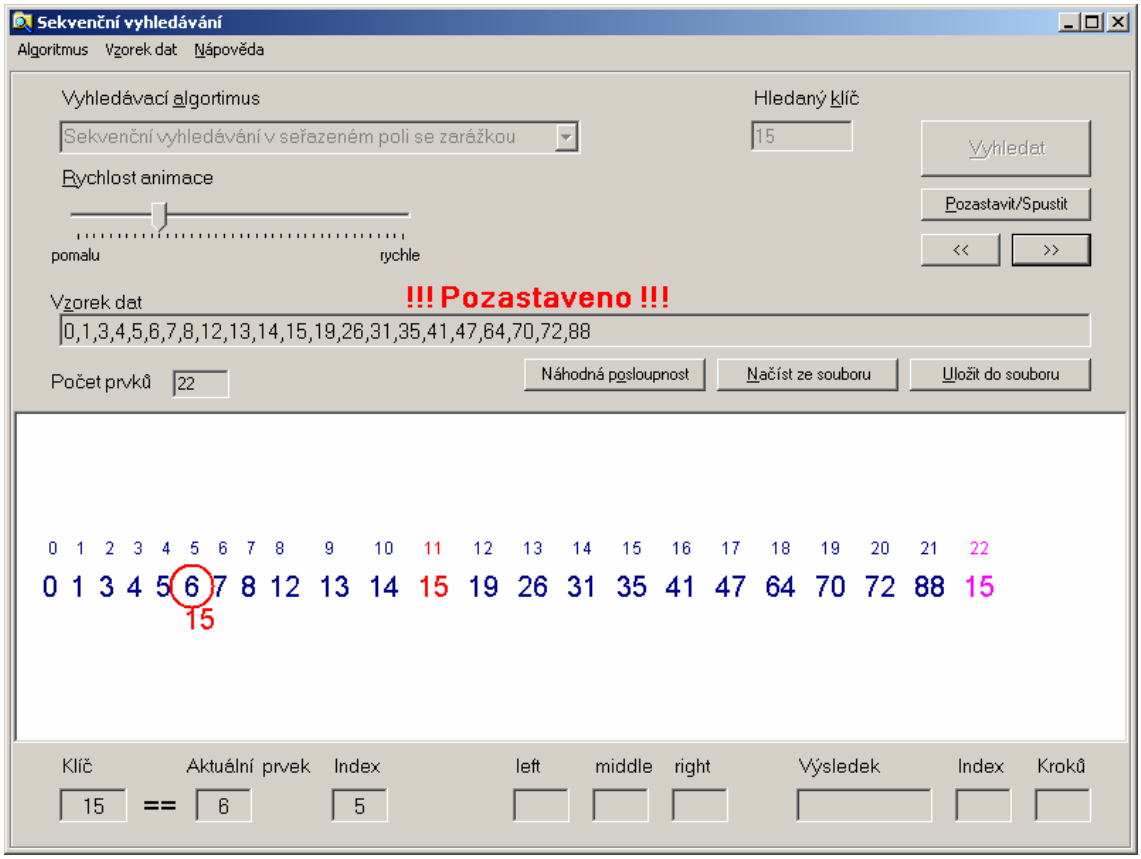
Příloha 3. Manuál demonstračních aplikací

Uživatelský manuál k aplikaci Sekvenční vyhledávání

Autor: Ivan Nejezchleb, xnejez06@stud.fit.vutbr.cz

Fakulta informačních technologií VUT v Brně, 2007

Aplikace slouží k demonstraci činnosti vyhledávacích algoritmů, které pracují s lineární vyhledávací tabulkou. Cílem je usnadnění pochopení problematiky vyhledávacích algoritmů.



Požadavky na systém

Minimální rozlišení obrazovky 800*600 obrazových bodů.

Nainstalovaný framework .NET 2.0 nebo vyšší.

Obsažené vyhledávací algoritmy

- Sekvenční vyhledávání v neseřazeném poli
- Sekvenční vyhledávání v neseřazeném poli se zarážkou
- Sekvenční vyhledávání v seřazeném poli
- Sekvenční vyhledávání v seřazeném poli se zarážkou
- Vyhledávání s rekonfigurací prvků podle četnosti vyhledání
- Binární vyhledávání
- Dijkstrova varianta binárního vyhledávání - nejpravější
- Dijkstrova varianta binárního vyhledávání - nejlevější
- Fibonacciho vyhledávání

Používání programu

Po spuštění programu zvolte vyhledávací algoritmus, jehož činnost chcete demonstrovat. Zadejte klíč, který má být vyhledáván. Hodnoty klíče jsou omezeny na celá čísla z intervalu 0 až 99. Hodnoty prvků tabulky, mezi kterými se bude vyhledávat, jsou dány obsahem textboxu. Pokud automaticky vygenerovaná posloupnost nevyhovuje, můžete hodnoty změnit ruční editací textu, vygenerováním nové náhodné posloupnosti, nebo načtením dat z textového souboru viz. Formát textového souboru.

Vyhledávání se spouští pomocí tlačítka „Vyhledat“. Animace běží samovolně s rychlostí určenou polohou posuvníku „Rychlost animace“. Rychlost animace se dá změnit kdykoli během jejího provádění. Animaci lze také kdykoli pozastavit a poté opět spustit tlačítkem „Pozastavit / Spustit“. Při pozastavení je možné využít krokovací mechanismu, viz Režim krokování.

Hlavní menu programu

Pomocí menu je možné zvolit prováděný algoritmus a dále manipulovat s prvky vyhledávací tabulky prostřednictvím voleb pro vygenerování náhodné posloupnosti, načtení ze souboru a uložení do souboru.

Poslední položkou menu je „O aplikaci“ vyvolávající okno se stručnými informacemi o programu.

Formát textového souboru

Hodnoty prvků z vyhledávací tabulky mohou být uloženy do souboru a poté z něj opětovně načteny do aplikace. Jde o běžný textový soubor. Hodnoty musí splňovat následující podmínky, které platí i při přímé editaci obsahu textového pole v aplikaci:

- Hodnoty prvků musí být celá čísla z intervalu od 0 do 99.
- Jednotlivá čísla jsou navzájem oddělena čárkou (bez mezer). Žádný jiný oddělovač akceptován není.
- Maximální délka načtených dat včetně oddělovačů je 150 znaků. Ostatní znaky jsou ignorovány.

Splnění těchto podmínek je kontrolováno před spuštěním animace.

Režim krokování

Krokování je dostupné pouze při pozastavení automatického běhu animace. Pomocí tlačítek pro krokování („<<“, „>>“) můžete procházet jednotlivými kroky algoritmu v celém průběhu. Po dosažení posledního kroku animace je tato ukončena a vypsán výsledek vyhledávání. V tomto okamžiku už není možné krokovat zpět.

Animace, význam barev

Během animace jsou vykresleny všechny prvky vyhledávací tabulky spolu se svými indexy.

- Aktuální prvek je zvýrazněn červeným orámováním. Pod tímto orámováním je vykreslena hodnota hledaného klíče.
- Červenou barvou je odlišen hledaný prvek pokud se v tabulce nachází.
- Při animaci algoritmů používajících zarážku je tento prvek na konci pole zvýrazněn fialovou barvou.
- U vyhledávání s rekonfigurací prvků podle četnosti vyhledání je zvýrazněn fialovou barvou prvek, se kterým si nalezené promění pozici.

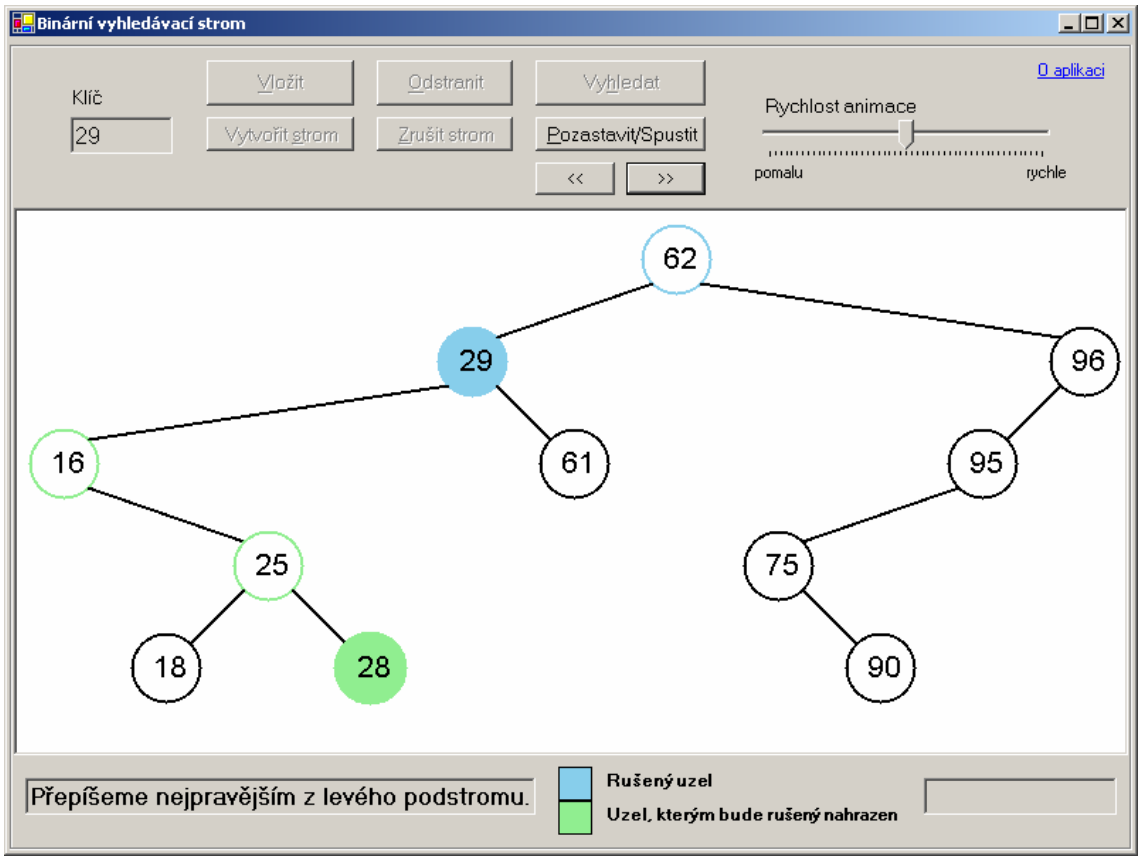
Při každém kroku algoritmu je v informační oblasti pod animací vypisována hodnota hledaného klíče, klíč aktuálního prvku a index aktuálního prvku. Pro algoritmy nesekvenčního přístupu jsou vypisovány navíc hodnoty okrajů prohledávaných oblastí a index dělícího bodu. Po skončení animace je v levém dolním rohu vypsán výsledek vyhledávání a počet kroků vykonávání algoritmu. V případě nalezení je vypsán i index nalezeného prvku.

Uživatelský manuál k aplikaci Binární vyhledávací strom

Autor: Ivan Nejezchleb, xnejez06@stud.fit.vutbr.cz

Fakulta informačních technologií VUT v Brně, 2007

Aplikace slouží k demonstraci činnosti algoritmů, které pracují s binárním vyhledávacím stromem. Implementovány jsou operace vkládání, vyhledávání a rušení uzlu stromu a vygenerování náhodného stromu a zrušení celého stromu. Cílem je usnadnění pochopení problematiky vyhledávacích algoritmů.



Požadavky na systém

Minimální rozlišení obrazovky 800*600 obrazových bodů.

Nainstalovaný framework .NET 2.0 nebo vyšší.

Používání programu

Po spuštění programu neexistuje žádný strom. Je nutné nejprve nějaký vytvořit postupným užitím funkce vkládání nebo vygenerovat náhodný strom (počet jeho uzlů je pevně nastaven na 10).

Pro použití operace vkládání zadejte klíč, s jehož hodnotou má být vytvořen nový uzel. Hodnoty klíče jsou omezeny na celá čísla z intervalu 0 až 99. Vkládání se spouští tlačítkem „Vložit“. Vyhledávání se spouští pomocí tlačítka „Vyhledat“, operace rušení uzlu se zadaným klíčem se spouští tlačítkem „Odstranit“. Obě tyto operace pro svoji činnost potřebují zadat klíč stejně jako vkládání. Po stisknutí některého z těchto tlačítek se spustí animace příslušné operace. Animace běží samovolně s rychlostí určenou polohou posuvníku „Rychlost animace“. Rychlost animace se dá změnit kdykoli během jejího provádění. Animaci lze také kdykoli pozastavit a poté opět spustit tlačítkem „Pozastavit / Spustit“. Při pozastavení je možné využít krokovací mechanismu, viz Režim krokování.

Stručnou informaci o aplikaci si zobrazíte kliknutím na odkaz „O aplikaci“ v pravém horním rohu.

Režim krokování

Krokování je dostupné pouze při pozastavení automatického běhu animace. Pomocí tlačítek pro krokování („<<“, „>>“) můžete procházet jednotlivými kroky operace v celém průběhu. Po dosažení posledního kroku animace je tato ukončena a vypsán výsledek vyhledávání. V případě vkládání či rušení prvku je provedena operace s daným uzlem a vykreslen strom v nové podobě. V tomto okamžiku už není možné krokovat zpět.

Animace, význam barev

Během animace jsou vykresleny všechny uzly stromu v patřičné struktuře binárního vyhledávacího stromu. Při provádění operace je aktuální prvek zvýrazněn tučným orámováním jiné barvy než ostatní uzly. Navíc jsou stejnou barvou ale jinou tloušťkou čáry zvýrazněny všechny uzly na cestě od kořene k aktuálnímu prvku.

Při všech operacích je to barva modrá. Pouze u operace rušení uzlu je v druhé části algoritmu, kdy je vyhledáván uzel, který přepíše rušeného, použita barva zelená.

Kromě použití barev pro orámování uzlu jsou použity barvy i pro vybarvení významných uzlů:

- Při operaci vkládání je zvýrazněn modrou barvou vložený prvek po jeho přidání do stromu. V případě, že tento prvek se už ve stromě vyskytuje, je tento uzel zvýrazněn již během průchodu algoritmu.
- Při operaci vyhledávání v binárním vyhledávacím stromě je modrou barvou pozadí zvýrazněn hledaný prvek, pokud se ve stromě vyskytuje.
- Při operaci rušení uzlu je modrou barvou zvýrazněn rušený prvek a zelenou barvou prvek, kterým bude rušený přepsán. To vše samozřejmě jen v případě, že se rušený prvek ve stromě nalézá.

Význam použití jednotlivých barev v závislosti na prováděné operaci je zobrazen i v informační oblasti pod prostorem animace.

Při každém kroku algoritmu je v informační oblasti pod animací znázorněno porovnání hledaného uzlu s aktuálním a rozhodnutí, v jakém podstromě se bude pokračovat. To je společné i pro vkládání, kdy musí být nalezen uzel, ke kterému bude vkládaný připojen. U operace rušení uzlu je ve fázi hledání rušeného vypisována stejná informace a v okamžiku jeho nalezení se přidá informace o tom, o jaký případ rušení uzlu se jedná.

Po skončení operace je vypsán její výsledek v levém dolním rohu okna aplikace:

- Pro operaci vkládání je to buď potvrzení vložení uzlu, nebo oznámení, že strom už obsahuje uzel s daným klíčem.
- Pro operaci vyhledávání je to výsledek hledání
- Pro operaci rušení uzlu je to potvrzení zrušení uzlu, nebo informace, že rušený prvek ve stromě neexistuje.
- Po operacích vygenerování celého stromu nebo jeho zrušení je vypsáno potvrzení o provedení operace.

Příloha 4. Studijní opora předmětu Algoritmy

ALGORITMY VYHLEDÁVÁNÍ V JAZYCE C

Studijní opora

Autor: Ivan Nejezchleb

Verze: 1.5 - 17.7.2007

Předmluva

PŘEDMLUVA

Tento učební text vznikl jako modifikace původní opory pro předmět Algoritmy, který je vyučován na FIT VUT. Modifikovaná opora vznikla se souhlasem a s pomocí autora původního díla pana Prof. Ing. Jana Maxmiliána Honzíka, CSc.

Konstanty

V ukázkách kódu obsažených v textu je použito booleovských konstant true a false. Vzhledem k tomu, že jazyk C implicitně tyto hodnoty nepodporuje, je v celém textu předpokládáno jejich nadefinování

```
#define true 1
#define false 0
```

Indexování pole

Narozdíl od jazyka Pascal je pole v jazyce C indexováno od 0. Poslední prvek má tedy index `delkaPole-1` a nikoli `delkaPole`.
Tento fakt může v programátorských začátcích způsobovat nemalé potíže.

	Pascal	jazyk C
první prvek pole	`pole[1]`	`pole[0]`
poslední prvek pole	`pole[delkaPole]`	`pole[delkaPole-1]`

4 VYHLEDÁVÁNÍ

Cíl kapitoly

Cílem kapitoly je popis metod implementace ADT "vyhledávací tabulka". Bude zaměřena na vyhledávání ve vnitřní (operační) paměti s přímým přístupem a s využitím dynamických struktur a DPP.

V této kapitole uvedeme metody, které se liší způsobem přístupu k paměťovému prostoru, v němž je vyhledávací tabulka umístěna, a to:

- sekvenční vyhledávání
- nesekvenční vyhledávání v poli
- vyhledávání v binárním vyhledávacím stromu (*binary search tree*)
- vyhledávání v tabulkách s rozptýlenými položkami (*hash-table*)

Nejvýznamnějším kriteriem pro hodnocení vyhledávácích tabulek je "přístupová doba" (*access time*). Je to doba potřebná k zajištění přístupu (k vyhledání) položky s hledaným klíčem. Pro hodnocení se používá několik časových vlastností vyhledávání. Zvlášť se označují doby pro úspěšné a pro neúspěšné vyhledání, které se u některých metod liší:

- minimální doba úspěšného a neúspěšného vyhledání
- maximální doba úspěšného a neúspěšného vyhledání
- průměrná doba úspěšného a neúspěšného vyhledání

Průměrná doba úspěšného vyhledání je teoretický parametr, který je dán podílem součtu dob úspěšného vyhledávání klíčů všech položek tabulky a počtu položek.

Vlastnosti klíče

Pro implementaci tabulky v konkrétním prostředí je nutné znát vlastnosti množiny hodnot klíče, podle kterého se bude vyhledávat. Je významné rozlišit případy, kdy nad typem klíč je definována pouze relace ekvivalence, nebo kdy je navíc definována i relace uspořádání.

Klíč může být jednoduchý nebo strukturovaný. Priority složek strukturovaného klíče určují váhu (význam) jednotlivých položek. Pro ekvivalenci dvou strukturovaných klíčů musí být ekvivalentní všechny odpovídající si složky klíče. Pro relaci uspořádání dvou strukturovaných klíčů se porovnávají postupně odpovídající si složky klíče se snižující se prioritou (vahou).

Příklad: Položka "TOsoba" má jako klíč "datum narození", které sestává ze tří složek: Rok, Mesic a Den. Priorita složek pro relaci uspořádání je:

1. Rok, 2. Mesic, 3. Den.

Pro relaci uspořádání v seznamu osob uspořádaných podle pořadí narozenin v roce (s tím, že starší bude v den stejných narozenin uveden dříve) je:

1. Mesic, 2. Den, 3. Rok.

Schéma vyhledávání

Základní schéma vyhledávacího algoritmu má tvar:

```
Nasel = false;
while (!Nasel && <množina prvků není vyčerpána>){
    < prozkoumej další prvek. Je-li to hledaný, nastav Nasel
na true >
} // while
return Nasel;
```

Nevhodný zápis dvou podmínek v cyklu while může způsobit chybu při pokusu o vyhledání neexistujícího prvku. V případě práce s polem by zápis:

```
i = 0;
while (K != Pole[i] && i < MAX){
   i++;
}
return K == Pole[i];
```

vedl k referenci na neexistující prvek `Pole[MAX]`.

Je-li vyhledávací tabulka v poli o `MAX` prvcích zcela zaplněná a přitom prvek s hledaným klíčem neexistuje, dojde k pokusu o referenci neexistujícího prvku `Pole[MAX]`!

Vhodným řešením je použití booleovské proměnné:


```
Nasel = false;
i = 0;
while (!Nasel && i < MAX){
    if (K == Pole[i])
       Nasel = true;
    else
       i++;
}
return Nasel;
```

Lze také použít zápisu se "zkratovým vyhodnocením booleovského výrazu" (*short-cut evaluation of Boolean expression*). Pokud vzroste i nad horní hodnotu, booleovský výraz se vyhodnotí jako false dříve, než dojde k nekorektní referenci neexistujícího prvku `Pole[i + 1]`:


```
i = 0;
// zkratové vyhodnocení Booleovského výrazu
while (i < MAX && K != Pole[i]){
   i++;
}
Search = i < MAX;
```

Pozn. V algoritmech IAL se "zkratově vyhodnocované booleovské výrazy" používají spíše výjimečně. V případě použití, jsou vždy uvedeny komentářem.

Podobně chybná situace může nastat v seznamové nebo souborové struktuře:

```
Uk = UkZac;
while (K != Uk->Klic && Uk != NULL){
// chybná reference
  Uk = Uk->Puk;
}
```

Řešení zkratovým vyhodnocením

```
.....while (Uk != NULL && K != Uk->Klic)...
```

nebo raději:

```
Nasel = false;
while (!Nasel && Uk != NULL){
  if (K == Uk->Klic)
     Nasel = true;
  else
     Uk = Uk->Puk;
}
```

Dynamika tabulky

Vyhledávácí tabulka se z pohledu dynamiky může chovat zcela staticky (seznam obcí v republice), staticky po etapách (papírový telefonní seznam aktualizovaný po roce), dynamicky s ohledem na operaci insert (změna jen přidáváním nových položek - tabulka identifikátorů v překladači), nebo plně dynamicky (dovoluje rušení položek). Některé implementace tabulek nedovolují nejvyšší stupeň dynamiky, nebo je jeho provedení těžkopádné a neefektivní. Rušení položky v každé tabulce je možné tzv. "**zaslepením**".

Zaslepení

Zrušení položky **zaslepením** se provede tak, že se klíč rušené hodnoty přepíše takovou hodnotou klíče, o níž je jisté, že nebude nikdy vyhledávána.

V rámci této kapitoly budou uvedeny následující metody. Některé z nich budou uvedeny jen pro ilustraci zajímavých přístupů a nejsou předmětem zkoušení,

- Sekvenční vyhledávání v neseřazeném poli
- Sekvenční vyhledávání v neseřazeném poli se zarážkou
- Sekvenční vyhledávání v seřazeném poli
- Sekvenční vyhledávání v seřazeném poli se zarážkou
- Sekvenční vyhledávání v poli seřazeném podle pravděpodobnosti vyhledání klíče
- Sekvenční vyhledávání v poli s adaptivním uspořádáním podle četnosti vyhledání
- Binární vyhledávání v seřazeném poli
    - normální binární vyhledávání
    - Dijkstrova varianta binárního vyhledávání
- Uniformní binární vyhledávání
- Fibonacciho vyhledávání
- Binární vyhledávací stromy (BVS)
- AVL stromy
- Tabulky s rozptýlenými položkami (Hashing tables) - TRP

4.1 Sekvenční vyhledávání v poli

4.1 Sekvenční vyhledávání v poli bude používat následující datové typy:

```
#define MAX ...  // maximální kapacita tabulky

typedef struct { // typ položky tabulky
   TKlic Klic;
   TData Data;
}TPol;

typedef struct { // typ tabulka implementovaná polem
   TPol Tab[MAX]; // pole tabulky
   int N; // aktuální počet prvků v tabulce
}TTab;
```

Pozn. Pro typ klíče budeme používat nejčastěji identifikátor TKlic, pro název klíčové složky položky tabulky identifikátor Klic a pro hodnotu vyhledávaného klíče identifikátor K.

- Inicializace tabulky – nulování aktuálního počtu prvků N
- Operace vyhledávání "Search" :

```
int Search(TTab T, TKlic K){
// funkce vrací hodnotu "true" v případě nalezení prvku s
// hledaným klíčem
   int i = 0;
   int Nasel = false;
   while (!Nasel && i<T.N){
      if (K == T.Tab[i].Klic)
         Nasel = true;
      else
         i++;
   } // while
   return Nasel;
} // Search
```

- Varianta operace Search v podobě funkce vracející výsledek hledání a polohu (index) nalezeného prvku v parametru předávaném odkazem:

```
int SearchIns(TTab T, TKlic K, int* Kde){
   int i = 0;
   int Nasel = false;
   while (!Nasel && i<T.N){
      if (K == T.Tab[i].Klic)
         Nasel = true;
      else
         i++;
    } // while
    *Kde = i; // pro Nasel==false je i nedefinováno
    return Nasel;
} // SearchIns
```

- Operace Insert používá funkce SearchIns pro účely vrácení polohy nalezeného prvku:

x+y

```
int Insert(TTab* T, TPol Pol){
   int Overflow = false; // počáteční nastavení příznaku
                         // plné tabulky
   int Kde;
   int Nasel = SearchIns(*T,Pol.Klic,&Kde); // vyhledání
   //za účelem vkládání
   if (Nasel){
      T->Tab[Kde]= Pol; // přepsání staré položky
   }
   else { // má se vložit nový prvek
      Kde=T->N;
      if (Kde < MAX){  // je v  tabulce místo ?
      // lze  vložit
      T->Tab[Kde]=Pol;  // vkládání
      T->N++;     // aktualizace  počitadla
      }
      else {
         Overflow = true; // nelze vložit - přetečení
      } // if (Kde < MAX)
   } // if (Nasel)
   return Overflow;
} //Insert
```

- Operace Delete se provede výměnou rušeného prvku s posledním a zrušením posledního:

x+y

```
void Delete(TTab* T, TKlic K){
   int Kde;
   int Nasel;
   Nasel = SearchIns(*T, K, &Kde);
   if (Nasel){
      T->Tab[Kde] = T->Tab[T->N-1]; // rušený je přepsán
                                    // posledním
      T->N--;
   } // if
} // Delete
```

Pozn. Operaci Delete lze také implementovat zaslepením: Klíč rušené položky se přepíše hodnotou, která se nikdy nebude vyhledávat. Tento způsob ale snižuje efektivní kapacitu tabulky!

Sekvenční vyhledávání lze snadno realizovat i v typických sekvenčních strukturách, jako je seznam nebo soubor. Operací insert (s aktualizační sémantikou) přepíšeme prvek v případě úspěšného nalezení nebo nový vložíme na kterékoli místo (začátek u seznamu, konec u souboru). Operace Delete přináší nutnost rušit v seznamu (snadnou u dvousměrného, méně snadnou u jednosměrného seznamu). U sekvenčního souboru je rušení prvku problematické. Řešení s pracovním souborem, do kterého se přepíše vše kromě rušeného, a přepis zpět může mít nepřijatelnou časovou náročnost.

Hodnocení metody

Hodnocení metody sekvenčního vyhledávání

- minimální čas úspěšného vyhledání = 1
- maximální čas úspěšného vyhledání = N
- průměrný čas úspěšného vyhledání = N/2
- čas neúspěšného vyhledání = N
- nejrychleji jsou vyhledány položky, které jsou na počátku tabulky

4.2 Rychlé sekvenční vyhledávání

4.2 Vyhledávání se zarážkou – tzv. **rychlé sekvenční vyhledávání** používá zarážku, což je přidaná položka na konec obsazené části tabulky, do níž se vloží hledaný klíč. Vyhledání tak vždy skončí nalezením a úspěšnost se pozná podle indexu, na němž vyhledání skončilo. Rychlost spočívá ve zkrácení booleovského výrazu.

Zarážka (*sentinel, guard, stop-point*) je často používaná technika, která umožní vynechat test na konec pole (sekvence, průchodu). Nutnost rezervovat místo pro zarážku snižuje efektivní kapacitu tabulky o 1 položku.

```
int SearchG(TTab T, TKlic K){
   int i = 0;
   T.Tab[T.N].Klic = K; // vložení zarážky
   while (K != T.Tab[i].Klic)
      i++;
   return i!=(T.N);// když našel až zarážku, tak vlastně
                   // nenašel
} // SearchG
```

4.3 Sekvenční vyhledávání v seřazeném poli

4.3 **Sekvenční vyhledávání v seřazeném poli** (*ordered array*)

- Podmínkou vyhledávání v seřazeném poli je definovaná relace uspořádání (dříve stačila relace rovnosti) nad typem klíč.
- Pole je seřazeno podle velikosti klíče.
- Operace Search skončí neúspěšně, jakmile narazí na položku s klíčem, který je větší, než je hledaný klíč!
- Operace Search se urychlí jen pro případ neúspěšného vyhledávání. **<u>To je jediný význam vyhledávání v seřazeném poli!!</u>** Jinak se věci jen komplikují...!
- Operace Insert musí najít správné místo, kam vloží nový prvek, aby zachovala seřazenost pole. Segment pole od nalezeného místa se musí posunout o jedničku. (N se zvýší o jedničku).
- Operace Delete posune segmentem pole napravo od vyřazovaného doleva o jednu pozici a tím se vyřazovaný přepíše. (Počet prvků N se sníží o jedničku.)

Pozn. Pozor: Posun segmentu pole doprava se dělá cyklem zprava a naopak! Posun segmentu [Dolni..(Horni-1)] o jednu pozici doprava:

```
for(int i = Horni; i < (Dolni + 1); i--)
  Pole[i] = Pole[i-1];
```

```
int Search(TTab T, TKlic K){
// funkce vrací hodnotu "true" v případě nalezení prvku s
// hledaným klíčem
  int Konec=false;
  int i=0;
  while (! Konec && (i < T.N)){
      if (K <= T.Tab[i].Klic)
        Konec=true;
      else
        i++;
  } // while
  if (i < T.N)
    return T.Tab[i].Klic == K;
  else
    return false;
} //Search
```

Upravte algoritmus sekvenčního vyhledávání v seřazeném poli tak, aby pracoval se zarážkou.

4.4 Vyhledávání s rekonfigurací pole

4.4 Sekvenční vyhledávání s adaptivní rekonfigurací pole podle četnosti přístupů.

Nejvýhodnější by bylo uspořádání pole podle četnosti vyhledání tak, aby nejčastěji vyhledávané položky byly na počátku pole. To lze realizovat občasným seřazením položek podle počitadla, které se aktualizuje po každém přístupu k položce.

Výhodnější variantou je vyhledávání s adaptivním rekonfigurací pole podle četnosti vyhledání. Po každém přístupu k položce se položka vymění se svým levým sousedem, pokud sama již není na první pozici:

Součástí vyhledávacího cyklu této metody je příkaz:

```
if (Kde>0){ // vymena polozek
  TPol tmp=T.Tab[Kde];
  T.Tab[Kde]=T.Tab[Kde-1];
  T.Tab[Kde-1]=tmp;
}
```

Tato metoda je velmi elegantní a účinná všude tam, kde se spokojíme se sekvenčním vyhledáváním a lineární složitostí.

4.5 Binární vyhledávání

4.5 Binární vyhledávání v seřazeném poli se provádí nad seřazenou množinou klíčů s náhodným přístupem (v poli). Metoda připomíná metodu půlení intervalu pro hledání jediného kořene funkce v daném intervalu. Hlavní vlastností binárního vyhledávání je jeho složitost, která je v nejhorším případě **logaritmická $\log_2(n)$.**

K samostatné úvaze. Porovnejte zaokrouhlenou hodnotu pro nejhorší případ binárního vyhledávání v poli o 1000 prvcích a pro nejhorší případ sekvenčního vyhledávání v tomto poli. (Jinými slovy, kolikrát se vám podaří rozpůlit interval o 1000 položkách, aby již nebylo co půlit?)

Pro pole tabulky implementované binárním vyhledáváním platí:

```
  Tab[0].Klic < Tab[1].Klic < ...< Tab[N-1].Klic
```

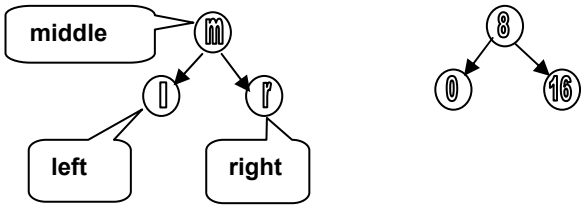
a pro vyhledávaný klíč musí platit:

```
  (K >=Tab[0].Klic ) && (K<=Tab[N-1].Klic)
```

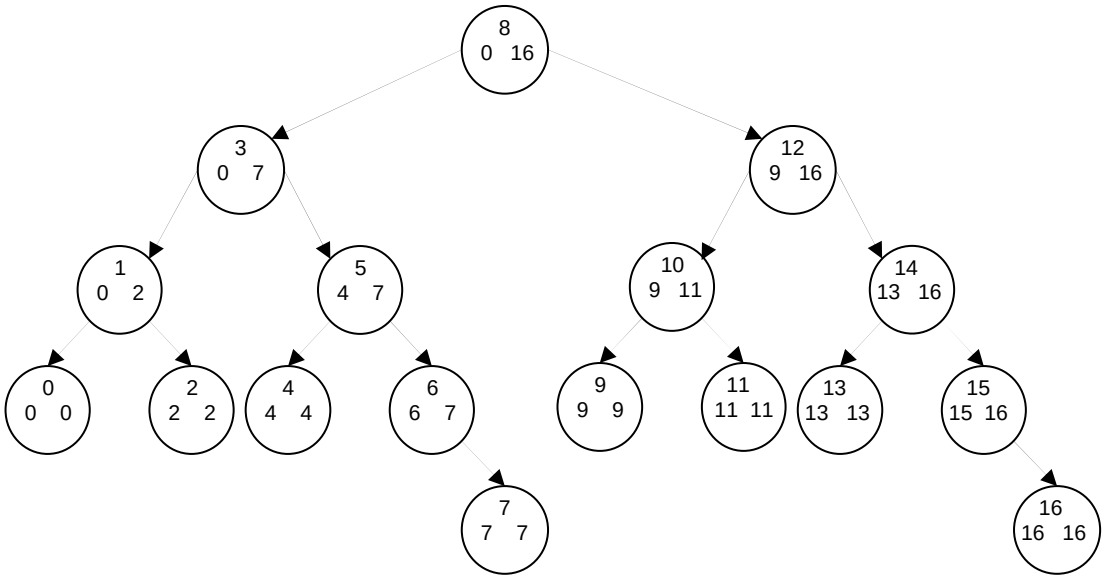
pak algoritmus vyhledávání tvoří sekvence příkazů:

```
int BinarySearch(TTab T, TKlic K){
  int left=0;  // levý index
  int right=T.N-1;  // pravý index
  int middle;
  do {
    middle=(left+right)/2;
    if (K < T.Tab[middle].Klic)
      right=middle-1;    // hledaná položka je v levé
                         // polovině
    else
      left= middle+1; // hledaná položka je v pravé
                      // polovině
  } while ((K!=T.Tab[middle].Klic) && (right >= left));
  return K==T.Tab[middle].Klic;
} // BinarySearch
```

Mechanismus výpočtu středu je middle=(left+right)/2



Mechanismus postupného rozdělování intervalu a určování středu zobrazuje rozhodovací strom binárního vyhledávání.



4.6 Dijkstrova varianta

4.6 Dijkstrova varianta binárního vyhledávání

- E.W.Dijkstra – významný teoretik programování druhé poloviny minulého století.
- Dijkstrova varianta binárního vyhledávání vychází z předpokladu, že v poli může být více položek se shodným klíčem. (To se neočekává u vyhledávací tabulky. Dijkstrova varianta se používá pro účely řazení.)

Polohu kterého z několika položek se shodným klíčem má vrátit mechanismus Search? Obvyklým požadavkem je některý z krajních (nejčastěji poslední ze shodných). Tomuto požadavku odpovídá algoritmus, který nekončí tím, že najde shodu s klíčem, ale tím, že se dalším dělením dostane až na dvojici sousedních prvků, o nichž platí:

`Tab[i].Klic == K` a současně `Tab[i].Klic  < Tab[i+1].Klic`

To zaručuje, že i-tá položka je nejpravější ze shodných klíčů rovnajících se hledanému klíči K.

Pro tuto možnost (hledání nejpravějšího) pak platí:

```
Tab[0].Klic<=Tab[1].Klic<=...<=Tab[N-2].Klic<Tab[N-1].Klic
```

// všechny klíče pole mohou být shodné, kromě nejpravějšího

a také

```
    (K >= Tab[0].Klic) && (K < Tab[N-1].Klic)
```

// hodnota klíče musí být uvnitř intervalu mezi levým a pravým okrajem

Pak Dijkstrova varianta má podobu sekvence příkazů:

```
....
int left=0;
int right=T.N-1;
int middle;
while (right != (left+1)){
  middle=(left+right)/2;
  if (T.Tab[middle].Klic <= K)
    left=middle;
  else
    right=middle;
}
return K==T.Tab[left].Klic;
```

Příklad:

V poli: 1,2,3,4,5,5,6,6,6,8,9,13 najde algoritmus Dijkstrovy varianty klíč K=6 na indexu 8.

V poli: 1,1,1,1,1,1,1,1,1,1,2 najde algoritmus Dijkstrovy varianty klíč K=1 na indexu 9.

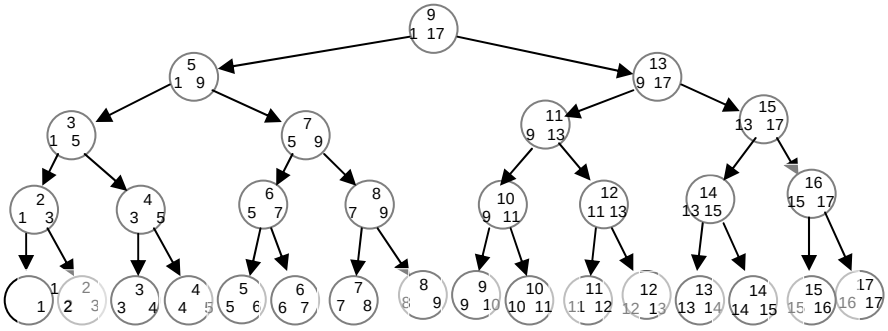
Rozdíl mezi vyhledáváním nejlevější a nejpravější položky mezi položkami se stejným klíčem se liší jen ostrostí nerovnosti v relaci

```
Tab[middle].Klic <= K // hledá nejpravější

Tab[middle].Klic < K   // hledá nejlevější
```

a okrajovým prvkem, který nesmí být vyhledáván a musí být větší/menší než ostatní pro verzi nejpravější/nejlevější.

Rozhodovací strom Dijkstrovy varianty pro pole [1..17] má tvar:



Hodnocení binárního vyhledávání

a) Vyhledávání má logaritmickou složitost
b) Je zvlášť výhodné pro statické tabulky, kde není nutný potenciálně časově náročný posun segmentu pole
c) Operace Insert a Delete mají stejný charakter jako u sekvenčního vyhledávání v seřazeném poli.
d) U Dijkstrovy varianty má doba pro úspěšné i neúspěšné vyhledání stejnou velikost $\log_2 N$.

4.7 Uniformní a Fibonacciho vyhledávání

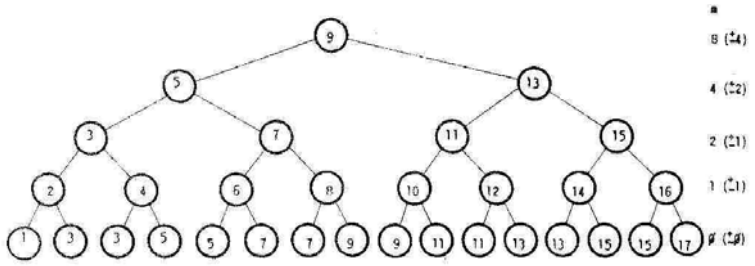
4.7 Uniformní a Fibonacciho vyhledávání nabízejí řešení pro případ, že operace půlení intervalu je operací časově náročnou. K takovému případu může dojít, pokud např. prostředí programového systému nepoužívá k zobrazení čísel dvojkového kódu.

Pozn. Počítače specializované na masové použití vstup/výstupních operací číselných hodnot (orientované např. na hromadné zpracování dat při ekonomických výpočtech) mohou používat specializovanou aritmetiku, založenou na binárně kódované desítkové soustavě (*binary coded decimal*) - BCD aritmetice. V prostředí binární aritmetiky je půlení realizováno jedním posunem bitu doprava a je velmi rychlou operací. V prostředí aritmetiky BCD může být operace půlení mnohem pomalejší a v rozsáhlém problému může výrazně snížit efektivnost metody založené na půlení. Metody uniformního binárního a Fibonacciho vyhledávání se snaží tuto nevýhodu odstranit.

Uniformní binární vyhledávání je založeno na principu určení hranic intervalu odchylkou od středu. Pro danou tabulku se spočítá pole odchylek a kraje intervalu se určí z hodnoty odchylky levého a pravého okraje od středu. V následujícím rozhodovacím stromu je vidět, že na dané úrovni jsou odchylky krajů intervalu od středu vždy stejné, proto se metoda jmenuje "uniformní".

Na nejnižší úrovni jsou kraje intervalu 1 a 3 vzdáleny od středu 0 o hodnotu 1 (v rozhodovacím stromu se např. k uzlu 5 mohu dostat doleva od středu 6 nebo doprava od středu 4). Pro danou tabulku lze odchylky stanovit jednorázově dopředu a v průběhu vyhledávání je již opakovaně nevyčíslovat. Tím se zabrání zdlouhavému půlení. Tabulky, které mají počet prvků o hodnotě 2^n-1, mají pole odchylek se společným základem. Pro takové tabulky lze připravit univerzální pole odchylek. Použití tabulky s libovolnou velikostí řeší **Sharova metoda**, která bude dále uvedena.

Rozhodovací binární strom pro uniformní binární vyhledávání.



Obr. 6.3 Rozhodovací binární strom pro uniformní binární vyhledávání

Fibonacciho strom

Tak, jako je binární vyhledávání v poli úzce spjato s binárním rozhodovacím stromem, který znázorňuje postup dělení intervalu v poli, je Fibonacciho vyhledávání úzce spjato s tzv. **Fibonacciho stromem.**

Fibonacciho posloupnost

DEF

Základem Fibonacciho stromu je **Fibonacciho posloupnost** 1. řádu, definovaná vztahy:

$A_0=0$, $A_1=1$, pro $i>1$ $A_i= A_{i-1}+A_{i-2}$

Následující prvek posloupnosti je dán součtem aktuálního prvku a jednoho jeho předchůdce. Pro počáteční hodnoty 0, a 1 má posloupnost tvar:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 atd.

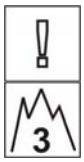
Definice Fibonacciho stromu

DEF

Fibonacciho strom (*F-tree*) je definován pravidly:

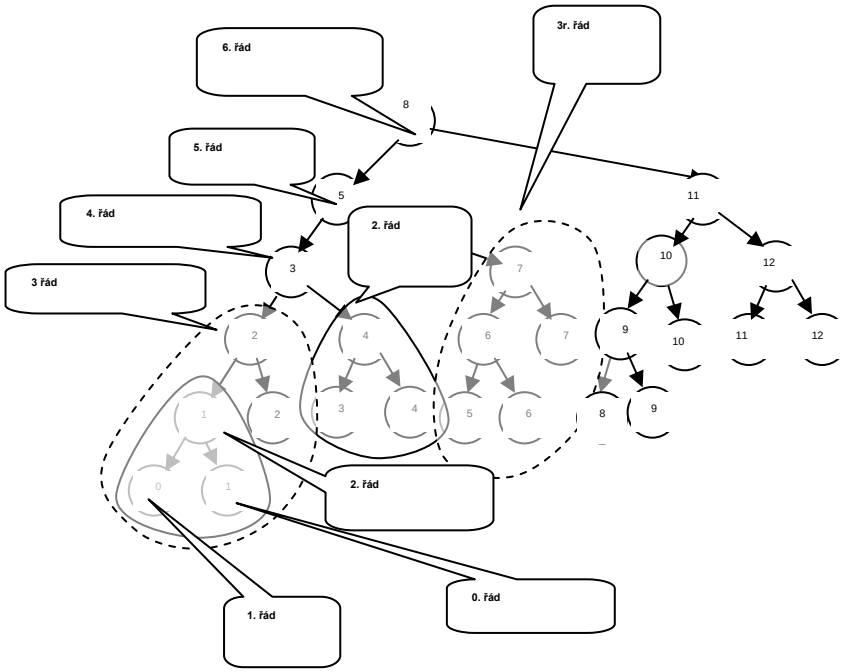
- F-tree i-tého řádu sestává z F_{i+1}-1 non-terminálních uzlů a z F_{i+1} terminálních uzlů.
- Je-li i=0 nebo i=1 je strom reprezentován pouze kořenem a současně terminálním uzlem [0].
- Je-li i>=1, pak je kořen stromu reprezentován hodnotou F_i, jeho levý podstrom je F-tree řádu i-1 (F_{i-1}) a pravý podstrom je F-tree řádu i-2 (F_{i-2}) , v němž všechny hodnoty uzlů jsou zvýšeny o hodnotu kořene F_i.
- Celkový počet uzlů Fibonacciho stromu i-tého řádu PocFtree(i) je dán vztahy:
  PocFtree(0)=PocFtree(1)=1
  PocFtree(i)=PocFtree(i-1)+PocFtree(i-2)+1.

Zatímco binární vyhledávání spočívá na půlení, jehož obrazem je binární strom, Fibonacciho vyhledávání spočívá na rozdělení intervalu na dva nestejné podintervaly, jehož obrazem je Fibonacciho strom. Na obrázku je prvotní interval 0..12, daný nejlevějším a nejpravějším uzlem, rozdělen kořenem a představuje index pole 8. Pokud je hledaný klíč v levém podintervalu, jde o posun doleva a další vyhledávání bude v levém podstromu. V opačném případě se bude vyhledávat v pravém podstromu. Situaci znázorňují další obrázky.



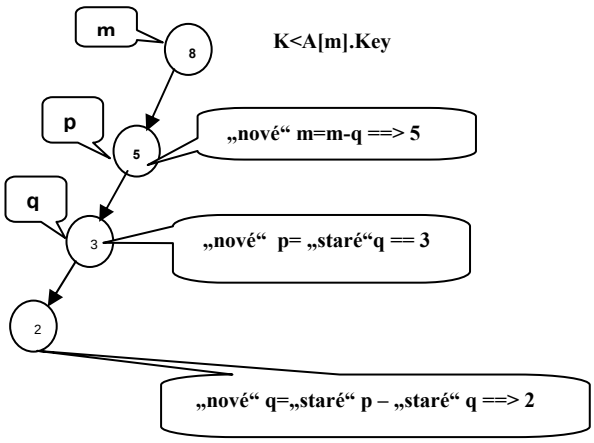
Neúspěšné vyhledání končí vždy na levém nebo na pravém terminálu nejnižšího stromu, který je vždy 2. řádu. To poznáme podle toho, že levý uzel ("oproštěný" od přidané hodnoty kořene) - algoritmu q - má hodnotu nula, nebo že pravý uzel ("oproštěný" od hodnoty kořene) - v algoritmu p - má hodnotu 1.

Fibonacciho strom



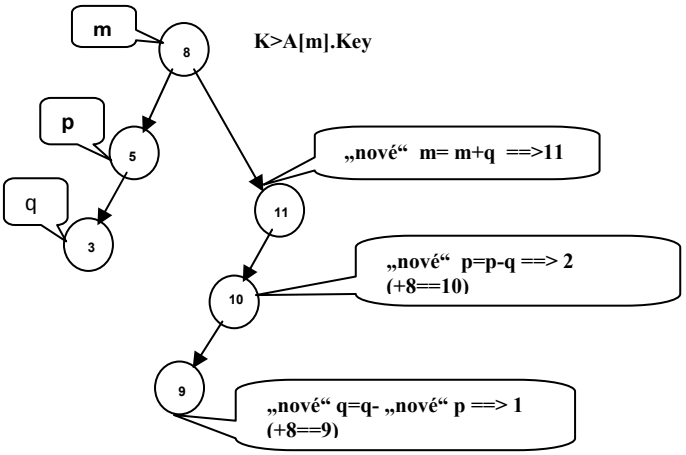
Posun doleva

Další vyhledávání bude v levém podstromu. Pomocné ukazatele m, p a q při tomto "posunu" po diagonále dolů dostanou nové hodnoty, jak je naznačeno v obrázku.



Posun doprava

Při vyhledávání v pravém podstromu dojde k následující změně indexových ukazatelů:



Vyhledávací algoritmus má tvar:

```
m=F(l);
p=F(l-1);
q=F(l-2);
TERM=false;
while ((K!=A[m].Key) && ! TERM){
  if (K<A[m].Key){// hledání pokračuje v levém podstromu
    if (q==0) // mohli bychom i otestovat zda nejde o klic
      TERM=true; // search končí na nulovém terminálu
    else{  // posun levého syna po diagonále
      m=m-q;  p1=q;
      q1=p-q;  p=p1;
      q=q1;
    } if (q==0)
  }
  else{ // search pokračuje v pravém podstromu
    if (p==1)// mohli bychom i otestovat zda nejde o klic
      TERM=true;//search končí na pravém terminálu 1. řádu
    else{ //  nové hodnoty m, p a q v pravém podstromu
      m=m+q;
      p=p-q;
      q=q-p;
    } // if (p==1)
  } // if (K<A[m].Key)
} // while
return !TERM;
```

Sharova metoda

Sharova metoda řeší případ, kdy skutečná velikost (počet prvků) tabulky je jiný, než je hodnota vhodná pro Uniformní binární nebo Fibonacciho vyhledávání. Metoda postupuje ve dvou krocích:

- V prvním kroku provede rozdělení na největším indexu, který vyhovuje metodě a který je menší než daná velikost.
- Ve druhém kroku zjišťuje, zda je hledaný klíč nalevo nebo napravo od dělicí hodnoty. Když je nalevo, postupuje jako by tabulka měla počet prvků

daný rozdělovací (a pro metodu vhodnou) hodnotou. Když je napravo, provede transformaci tabulky posunem začátku pole doprava tak, aby prohledávaná část tabulky měla opět vyhovující počet prvků.

Uniformní binární i Fibonacciho vyhledávání je zvláštním a okrajovým příkladem v oblasti vyhledávání. Ilustruje způsoby hledání efektivních algoritmů ve speciálních případech.

Fibonacciho vyhledávání je ukázkovým příkladem toho, že krátký a přehledný algoritmus je bez znalosti jednoduchého, ale málo známého principu Fibonacciho stromu, zcela nesrozumitelný i pro jinak zkušeného programátora. Je to podnět k tomu, že správná dokumentace musí kromě základní navigace v programu obsahovat i popis méně známých principů.

Kontrolní otázky a příklady

- Co je to strukturální ekvivalence?
- Co je to vyhledávání?
- Co je to přístupová doba?
- Má smysl použít cyklus FOR pro vyhledávání.?
- Co je to zkratové vyhodnocování?
- Vysvětlete pojem zaslepení.
- Co je to zarážka?
- Jaká je výhoda vyhledávání v seřazené posloupnosti?
- Jaká je nevýhoda operací insert a delete v tabulce seřazených položek oproti neseřazeným položkám?
- Jaká je maximální doba přístupu k prvku tabulky s binárním vyhledáváním?
- Jak se liší v době přístupu normální a Dijkstrova varianta binárního vyhledávání?
- Kdy je výhodné použití Uniformního nebo Fibonacciho vyhledávání?
- Čím se liší Fibonacciho vyhledávání od metod binárního vyhledávání?
- Nakreslete Fibonacciho strom 5. řádu.
- Jak se pozná konec neúspěšného vyhledávání ve Fibonacciho vyhledávání.
- V čem spočívá princip vyhledávání s adaptivní rekonfigurací položek podle četnosti přístupů a jaké má přednosti?

4.8 Binární vyhledávací stromy

4.8 Binární vyhledávací strom - BVS (*binary search tree*) je jednou z nejpoužívanějších implementací pro plně dynamické vyhledávací tabulky.

DEF

Uspořádaný strom je kořenový strom, pro jehož každý uzel platí, že n-tice kořenů podstromů uzlu je uspořádaná.

Binární vyhledávací strom je uspořádaný binární strom pro jehož každý uzel platí, že klíče všech uzlů levého podstromu jsou menší než klíč v uzlu a klíče všech uzlů pravého podstromu jsou větší než klíč v uzlu.

Rekurzívní definice:

Binární vyhledávací strom je buď prázdný nebo sestává z kořene, hodnota jehož klíče je větší než hodnota všech klíčů levého binárního vyhledávacího podstromu a je menší než hodnota všech klíčů pravého binárního podstromu.

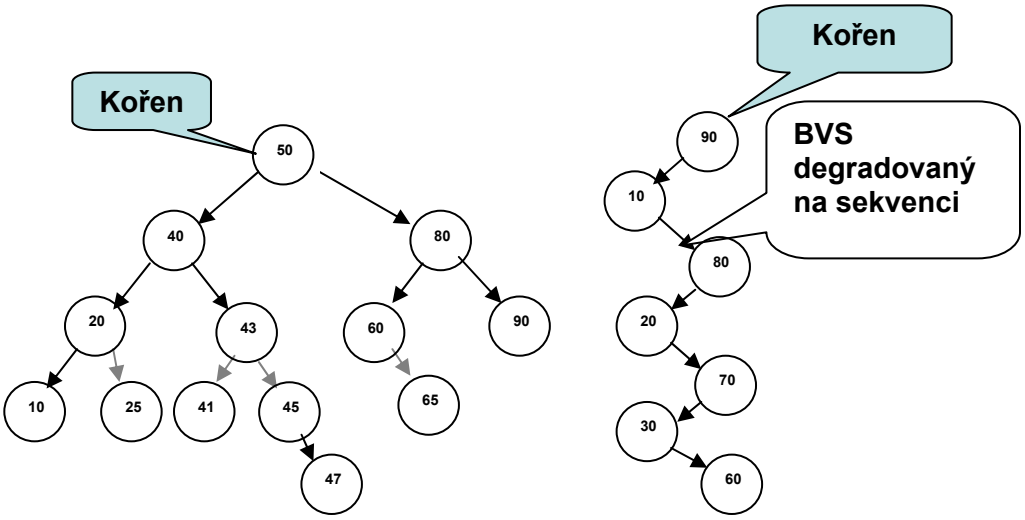
Pozn. Rekurzívní definice BVS vede k rekurzívním zápisům řady algoritmů nad BVS

Průchod InOrder binárním vyhledávacím stromem dává posloupnost prvků seřazenou podle velikosti klíče.

Cesta k terminálnímu uzlu váhově vyváženého BVS s N uzly prochází lg_2N uzly.

x+y

Příklady BVS



Průchod InOrder BVS stromem dává seřazenou posloupnost:

1 strom: 10,20,25,40,41,43,45,47,50,60,65,80,90
2 strom: 10,20,30,60,70,80,90

Vyhledávání v BVS

Vyhledávání v BVS je podobné binárnímu vyhledávání v seřazeném poli.

Je-li vyhledávaný klíč roven kořeni, vyhledávání končí úspěšným vyhledáním. Je-li klíč menší než klíč kořene, pokračuje vyhledávání v levém podstromu, je-li větší, pokračuje v pravém podstromu. Vyhledávání končí neúspěšně, pokud je prohledávaný (pod)strom prázdný.

Datové typy

Datové typy používané pro BVS jsou obdobné jako pro dvojsměrný seznam resp. binární strom:

```
struct Uzel;
typedef struct Uzel* TUk;

typedef struct Uzel
{
   TKlic Klic;
   TData Data;
   TUk LUk;
   TUk PUk;
}TUzel;
```

Rekurzívní zápis Search

Rekurzívní zápis funkce vyhledávání v BVS

x+y

```
int Search(TUk UkKor, TKlic K){// UkKor je ukazatel
                               // kořene BVS
  if (UkKor!=NULL){ // Strom je neprázdný
    if (UkKor->Klic==K){ // Našel hledaný klíč
      return true;
    } else //nenašel
        if (UkKor->Klic>K) // hledání pokračuje v levém
                           // podstromu
          return Search(UkKor->LUk,K);
        else // hledání pokračuje v pravém podstromu
          return Search(UkKor->PUk,K);
  }else   // cesta končí na termin. uzlu - nenalezl
     return false;
} // Search
```

Search vracející nalezeného

Varianta vyhledávání v BVS jako funkce, která vrací odkaz na nalezenou položku v parametru Kde předávaném odkazem. V případě nenalezení je `Kde==NULL`.

x+y

```
void SearchTree(TUk UkKor, TKlic K, TUk* Kde){
  if (UkKor == NULL)
    *Kde=NULL;
  else
    if (UkKor->Klic != K){
      if (UkKor->Klic > K)
        SearchTree(UkKor->LUk, K, Kde);
      else
        SearchTree(UkKor->PUk, K, Kde);
    }
    else *Kde=UkKor;  // našel a nastavuje výstupní
parametr Kde
} // SearchTree
```

Nerekurzívní Search

x+y

Nerekurzívní zápis funkce Search.

```
int Search (TUk UkKor,TKlic K){
   // Nerekurzívní zápis vyhledávání v BVS
   int Fin; // řídicí proměnná cyklu
   int Search=false;
   Fin=(UkKor==NULL);
   while (! Fin){
     if (UkKor->Klic==K) {
       Fin=true;
       Search=true;
     } else
         if (UkKor->Klic > K)
            UkKor=UkKor->LUk; // Pokračuj vlevo
         else
            UkKor=UkKor->PUk; // Pokračuj vpravo
     if (UkKor==NULL)
       Fin=true;
  } // while
  return Search;
} // Search
```

Insert v BVS

Operace Insert aplikuje „aktualizační sémantiku“, tzn., že v případě, že uzel s daným klíčem existuje, přepíše operace stará data aktuálními. V případě, že uzel s daným klíčem neexistuje, vloží operace nový uzel jako terminální uzel tak, aby se zachovala pravidla BVS.

Pro zkrácení zápisu použijeme tuto pomocnou procedura pro vytvoření uzlu:

```
void VytvorUzel(TUk* UkUzel, TKlic K, TData D){
  (*UkUzel)=(TUk)malloc(sizeof(TUzel));
  (*UkUzel)->Klic=K;
  (*UkUzel)->Data=D;
  (*UkUzel)->LUk=NULL;
  (*UkUzel)->PUk=NULL;
}
```

Rekurzívní Insert

x+y

Rekurzívní zápis operace Insert.

```
void Insert (TUk* UkKor, TKlic K, TData D){
  if ((*UkKor)==NULL)
    VytvorUzel(UkKor, K, D); // vytvoření kořene či term.
                             // uzlu
  else if (K<(*UkKor)->Klic)
      Insert(&((*UkKor)->LUk),K,D); //pokračuj vlevo
    else if (K>(*UkKor)->Klic)
         Insert(&((*UkKor)->PUk),K,D); // pokračuj vpravo
       else (*UkKor)->Data=D;  // přepiš stará data novými
} // Insert
```

Nerekurzívní Insert

x+y

Nerekurzívní zápis operace Insert využívá operace Search upravené tak, aby vracela polohu nalezeného uzlu pro přepis starých dat novými, nebo polohu uzlu, ke kterému se připojí nový vkládaný terminální uzel.

```
int SearchIns(TUk UkKor, TKlic K, TUk* Kde){
// Vyhledání za účelem nerekurzívního vkládání
  int Found=false;
  if (UkKor==NULL){
    Found=false;
    *Kde=NULL; // prázdný BVS
  } else
    do{
      *Kde=UkKor;  // uchování výstupní hodnoty Kde
      if (UkKor->Klic > K)
        UkKor=UkKor->LUk; // posun doleva
      else if (UkKor->Klic < K)
          UkKor=UkKor->PUk;  // posun doprava
        else
          Found=true; // našel, vrací hodnotu Kde
    }while (!Found && (UkKor!=NULL));
    return Found;
} //SearchIns
```

x+y

Procedura Insert v sobě vyvolá pomocnou proceduru SearchIns a přepíše nebo vloží data.

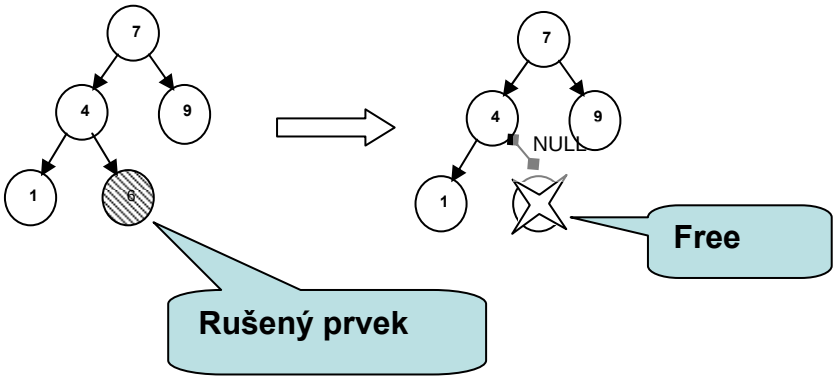
```
void Insert (TUk* UkKor, TKlic K, TData D){
  TUk PomUk, Kde;
  int Found =SearchIns(*UkKor,K,&Kde);
  if (Found)
     Kde->Data=D; // přepsání starých dat novými
  else{
     VytvorUzel(&PomUk,K,D);
     if (Kde ==NULL)
        *UkKor=PomUk; // prázdný strom, nový uzel je kořen
     else if (Kde->Klic>K) // neprázdný strom, nový uzel
                           // se připojí jako terminál
           Kde->LUk=PomUk; // nový uzel se připojí vlevo
        else // uzel se připojí vpravo
           Kde->PUk=PomUk;
   }
} // Insert
```

Operace Delete

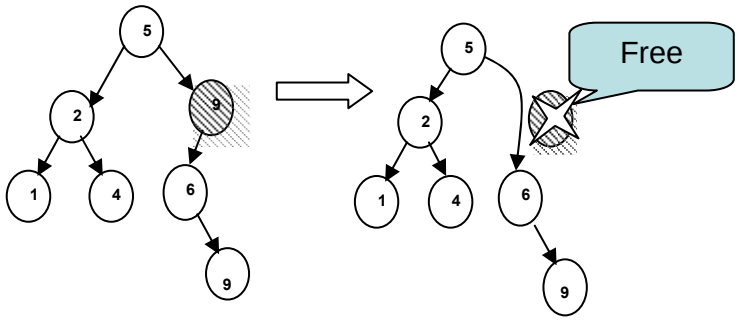
Rušení uzlu – operace Delete, je vždy složitější, než operace Insert. Rušení terminálního uzlu je snadné.

x+y

Příklad rušení terminálního uzlu

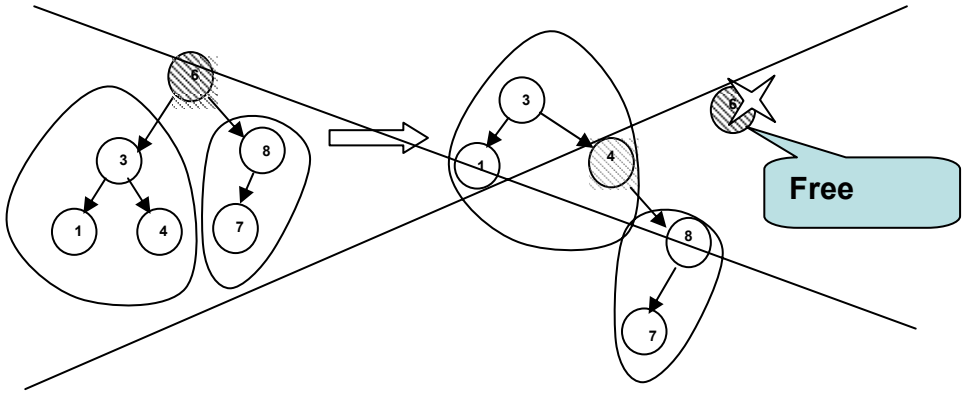


Zrušení uzlu, který má jen jednoho syna, je také snadné. Synovský uzel se připojí na nadřazený uzel rušeného uzlu.



Zrušit uzel, který má dva podstromy lze také tak, že levý podstrom připojíme na nejlevější uzel pravého podstromu nebo tak, že pravý podstrom rušeného uzlu připojíme na nejpravější uzel levého podstromu, jak je to uvedeno na následujícím obrázku. Toto řešení je však pro vyhledávací stromy nepřijatelné, protože zbytečně zvyšuje výšku stromu a tím i maximální dobu vyhledávání.

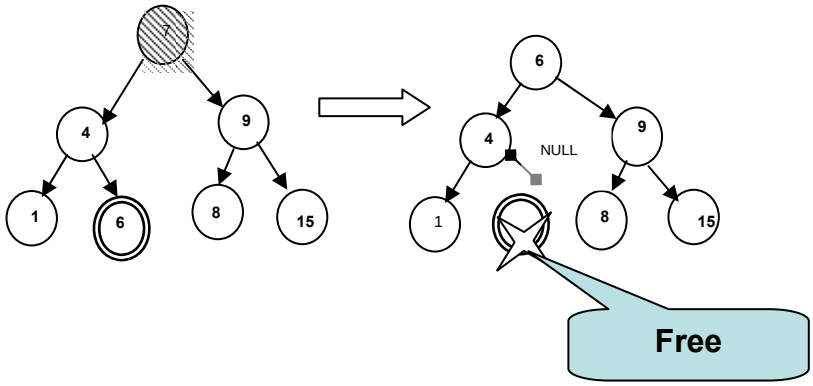
Nevhodný způsob rušení uzlu se dvěma synovskými uzly.



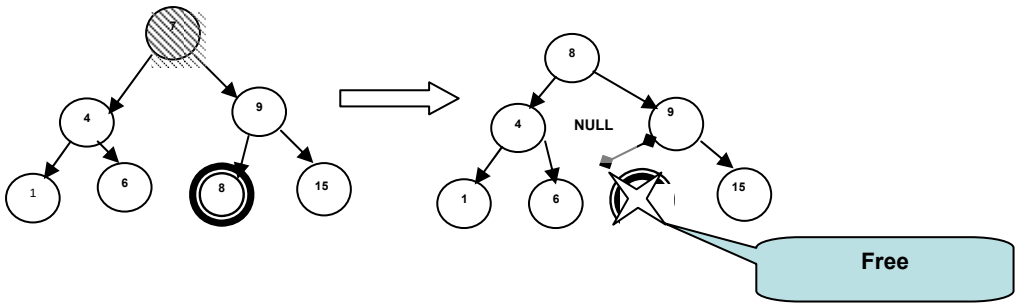
Rušení uzlu se dvěma synovskými uzly:

Rušený uzel se dvěma syny se nezruší „fyzicky“, ale jeho hodnota se přepíše hodnotou takového uzlu, který lze zrušit snadno a přitom při přepisu nesmí dojít k porušení uspořádání (pravidel) BVS. Takovým uzlem je nejpravější uzel levého podstromu rušeného uzlu nebo symetricky nejlevější uzel pravého podstromu rušeného uzlu.

Příklad rušení uzlu se dvěma podstromy



Symetrické řešení



Rekurzívní Delete

Rekurzívní zápis procedury Delete obsahuje pomocnou rekurzívní proceduru Del, která prochází po pravé diagonále levého podstromu a hledá prvek, jehož hodnotou se přepíše rušený uzel a který je následně rušen.

Algoritmus rekurzívní procedury je netriviální a u písemných zkoušek se neočekává jeho „rekonstrukce“. U závěrečné zkoušky lze žádat vysvětlení předloženého algoritmu.

```
/* Pomocná procedura Del se pohybuje po pravé diagonále
levého podstromu rušeného uzlu a hledá nejpravější uzel.
Když ho najde je obsah PomUk přepsán uzlem Uk a Uk je
připraven pro následnou operaci free.*/

void Del(TUk PomUk, TUk* Uk ){
  if ((*Uk)->PUk !=NULL)
    Del(PomUk,&((*Uk)->PUk)); // pokračuj v pravém
podstromu
  else { // nejpravější uzel je nalezen, přepsání a
uvolnění uzlu
    PomUk->Data=(*Uk)->Data;
    PomUk->Klic=(*Uk)->Klic;
    PomUk=(*Uk);
    (*Uk)=(*Uk)->LUk; // Uvolnění uzlu Uk! Pozor!
    // V proceduře Del je Uk ukazatelová složka uzlu
nadřazeného k uzlu Uk
  }
} // konec pomocné Del
```

```
void Delete(TUk* UkKor, TKlic K){
  TUk PomUk=NULL;
  if ((*UkKor) != NULL){// vyhledávání neskončilo;
                        //hledaný uzel může stále být v BVS
    if (K < (*UkKor)->Klic)
      Delete (&((*UkKor)->LUk),K); // pokračuj vlevo
    else if (K > (*UkKor)->Klic)
         Delete(&((*UkKor)->PUk),K);  // pokračuj vpravo
      else { // Klíč nalezen. Uzel  UkKor se bude rušit
         PomUk=(*UkKor);
         if (PomUk->PUk==NULL)// uzel nemá pravý podstrom
           (*UkKor)=PomUk->LUk;
          else // uzel má pravý podstrom;
            if (PomUk->LUk==NULL)//má jen pravý
               (*UkKor)=PomUk->PUk; // připojení pravého
```

```
                                    // podstromu
          else //uzel ma oba uzly
            Del(&PomUk, &(PomUk->LUk));
            // bude přepsán nejpravějším uzlem levého
            // podstromu procedurou Del
          free (PomUk); // uvolnění uzlu
      }
  }
  else ; // zde může být reakce na skutečnost, že uzel
         //nebyl nalezen; normálně se neděje nic
} // Delete
```

Nerekurzívní zápis procedury Delete obsahuje několik pomocných procedur. Procedura DeleteSearch vyhledává rušený prvek. Vrací polohu rušeného prvku, polohu uzlu jemu nadřazenému a booleovskou hodnotu o straně, ke které je rušený uzel k nadřazenému uzlu připojen. Procedura rušení musí v počáteční fázi ošetřit případ rušení kořene a rušení jediného uzlu.

```
/*Procedura hledá klíč K v BVS zadaném ukazatelem UkKor.
Parametry Found, OtecZleva, UkPraotce a UkOtce jsou
výstupní parametry. Je-li strom prázdný, pak Found==false
a UkOtce a UkPraOtce nejsou definovány. Je-li Found==true
a nalezený uzel je kořen, pak UkPraOtce==NULL,
UkOtce==UkKor a OtecZleva je nedefinovaný. Když nalezený
uzel je vnitřní uzel stromu, tak Found==true, UkPraOtce je
uzel nadřazený otci, OtecZleva==true je v případě, že otec
je levým synem praotce. Když je uzel nenalezen, je
Found==false a ostatní parametry jsou nedefinované.*/

void DeleteSearch(TUk UkKor, TKlic K,int* Found, int*
OtecZleva, TUk* UkOtce,TUk *UkPraOtce){
/* Procedura hledá rušený prvek. Kdyz ho najde, pak není-
li to kořen, vrací také ukazatel na nadřazený uzel a
informaci o tom, je-li k nadřazenému uzlu připojen zleva
nebo zprava */
  int Fin; // řídicí proměnná cyklu
  *Found=false;
  *UkOtce=UkKor;
  if (UkKor!=NULL){
    if (UkKor->Klic==K){
      *Found=true; // končí se, nalezený je kořen
      *UkPraOtce=NULL; // kořen nemá nadřazený uzel
    } else {
      Fin=false;
      while (! Fin){
        if ((*UkOtce)->Klic==K){
          *Found=true;
          Fin=true;
        } else {
          *UkPraOtce=*UkOtce; // Uchování ukazatele na
                              // nadřazený uzel
          if ((*UkOtce)->Klic > K){ // uchování směru
                    //připojení k nadřazenému uzlu
```

```
            *OtecZleva=true;
            *UkOtce=(*UkOtce)->LUk;
          } else {
            *OtecZleva=false;
            *UkOtce=(*UkOtce)->PUk;
          }
        } //  if
        if (*UkOtce==NULL)
            Fin=true;
      } // while
    }// if
  }//if (UkKor!=NULL)
}// DeleteSearch

/*Procedura najde Nejpravější uzel levého podstromu,
přepíše data otce nalezeným uzlem, uvolní nejpravější uzel
a vrátí ho v parametru pro pozdější free */
```

```
void Nejprav(TUk* UkOtce){
  TUk Nejpr,OtecNejpr;
  Nejpr=(*UkOtce)->LUk;
  if (Nejpr->PUk!=NULL){ // hledej nejpravější uzel
      do{
         OtecNejpr=Nejpr;
         Nejpr=Nejpr->PUk;
      }while (Nejpr->PUk!=NULL);
      OtecNejpr->PUk=Nejpr->LUk;
  } else { // Nejpr je  Nejprav sám
    (*UkOtce)->LUk=Nejpr->LUk;
  }
  (*UkOtce)->Klic=Nejpr->Klic; // přepis klíče
  (*UkOtce)->Data=Nejpr->Data; // přepis  dat
  (*UkOtce)=Nejpr;
/*Otec, ve skutečnosti nejpravější, bude uvolněn operací
free */
}
```

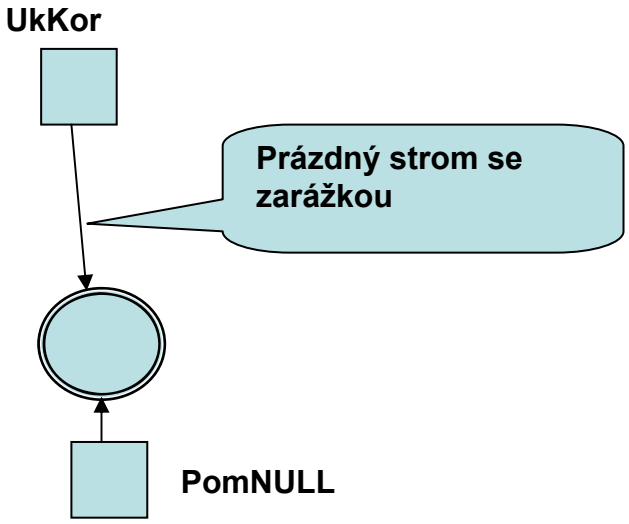
```
void Delete(TUk* UkKor, TKlic K){
  TUk UkOtce,UkPraOtce;
  int Found, OtecZleva;
  DeleteSearch(*UkKor, K, &Found, &OtecZleva, &UkOtce,
  &UkPraOtce);
  if (Found){
    if (UkOtce->PUk==NULL){ // Nalezený nemá pravého syna,
                            // jde zrušit snadno
      if (UkPraOtce==NULL)
        *UkKor =UkOtce->LUk; // Rušený je kořen s jediným
        //levým synem; syn (může být i prázdný) se stane
        //kořenem, je-li levý prázdný, zůstane jen kořen
      else // Připoj levého syna (může být i prázdný) na
           // praotce
        if (OtecZleva)
           UkPraOtce->LUk=UkOtce->LUk; // připoj syna
                       //(NULL) k  praotci zleva
        else UkPraOtce->PUk=UkOtce->LUk; // připoj syna
                       //(NULL)  zprava
```

```
    }
    else // nalezený má pravého syna
      if (UkOtce->LUk==NULL){ // nalezený nemá levého syna
        if (UkPraOtce==NULL) //rušený je kořen s jediným
//pravým synem; syn (může být i prázdný) se stane kořenem
           *UkKor=UkOtce->PUk;
        else if (OtecZleva) // připoj pravého syna (může
                            // být i prázdný) k praotci
               // připoj syna (NULL)  zleva
               UkPraOtce->LUk=UkOtce->PUk;
             else // připoj syna (NULL)  zprava
               UkPraOtce->PUk=UkOtce->PUk;
        }
      else Nejprav(&UkOtce);
      free(UkOtce); // fyzické rušení uzlu
  } // if (Found)
} // Delete
```

4.9 BVS se zarážkou

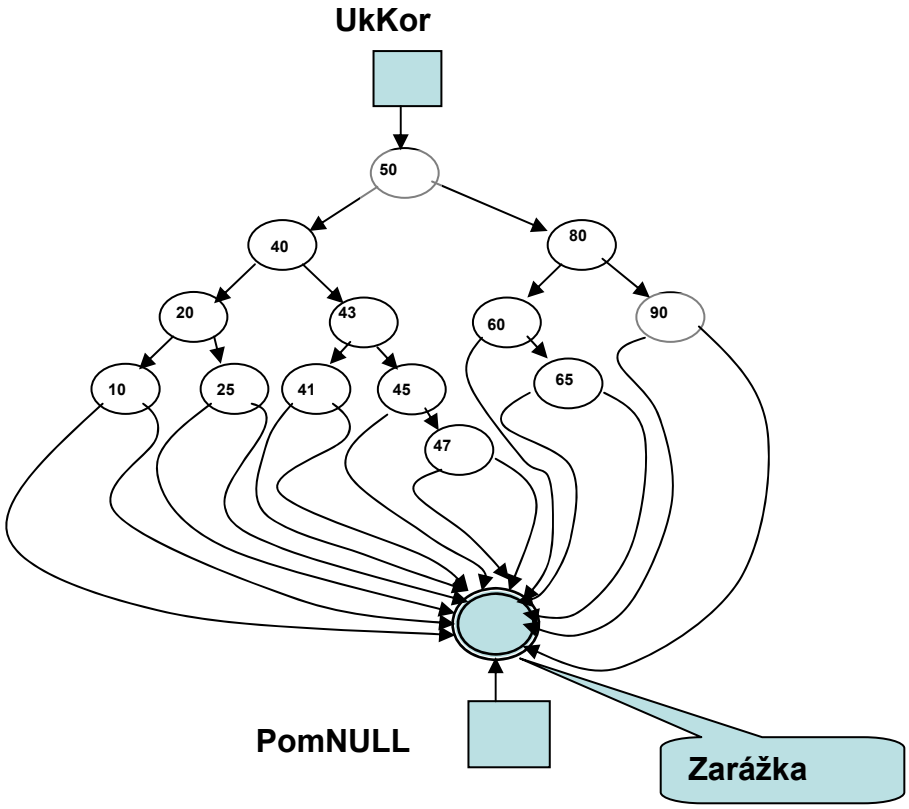
4.9 Binární vyhledávací strom se zarážkou používá pomocný uzel jako zarážku, do které vloží hledaný klíč. Tento uzel vždy najde, ale podle ukazatele pozná, zda jde o skutečný uzel nebo o zarážku. Ukazatel na tento uzel slouží jako "pomocný NULL".

Prázdný BVS se zarážkou



BVS se zarážkou

x+y



Inicializace tabulky implementované BVS se zarážkou.

x+y

```
void TInit(TUk* UkKor) {// proměnná PomNULL je globální
   PomNULL=malloc(sizeof(TUk));
   *UkKor=PomNULL;
}
```

Fukce Search

x+y

```
int SearchTree(TUk Uk, TKlic K) {
// proměnná PomNULL je globální
   PomNULL->Klic=K;
   while (Uk->Klic != K){
      if (Uk->Klic>K)
         Uk=Uk->LUk;
      else
         Uk=Uk->PUk;
    } // while
   return Uk!=PomNULL;
} //SearchTree
```

BVS se zarážkou je příklad použití zarážky v nelineární struktuře. Zarážka umožňuje vynechat opakující se test na konec a urychluje algoritmus.

4.10 BVS se zpětnými ukazateli

4.10 Binární vyhledávací strom se zpětnými ukazateli má význam pouze tehdy, chceme-li se při průchodu InOrder vyhnout rekurzi nebo použití zásobníku.

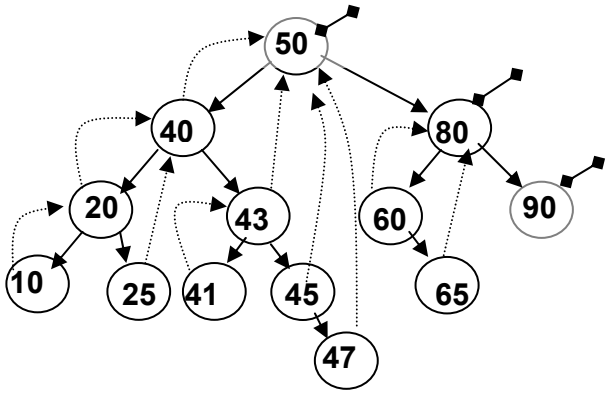
Operace **Search** je stejná, jako u normálního BVS.

Operace **Insert** musí při vkládání nového terminálního uzlu respektovat následující pravidla:

- Zpětný ukazatel kořene ukazuje na NULL (všechny uzly vedlejší diagonály ukazují na NULL...)
- Zpětný ukazatel levého syna ukazuje na svého otce
- Zpětný ukazatel pravého syna dědí ukazatele od otce (ukazuje tam kam otec).

Operace **Delete** se provádí stejně jako u normálních BVS, ale v případě, že se ruší prvek s jedním a to s pravým podstromem, musí se udělat korekce zpětných ukazatelů na vedlejší diagonále pravého podstromu.

Ukázka BVS se zpětnými ukazateli.



Definujme typy:

```
struct Uzel;
typedef struct Uzel* TUk;

typedef struct Uzel
{
   TKlic Klic;
   TData Data;
   TUk LUk, PUk, ZpetUk;
}TUzel;

TUk Nejlev(TUk UkKor) {
// funkce vrátí ukazatel na nejlevější uzel stromu
  TUk UkNaNejlev =UkKor;
  while (UkKorr != NULL){
    UkNaNejlev =UkKor;
    UkKor=UkKor->LUk;
  } // while
return UkNaNejlev;
}
```

InOrder

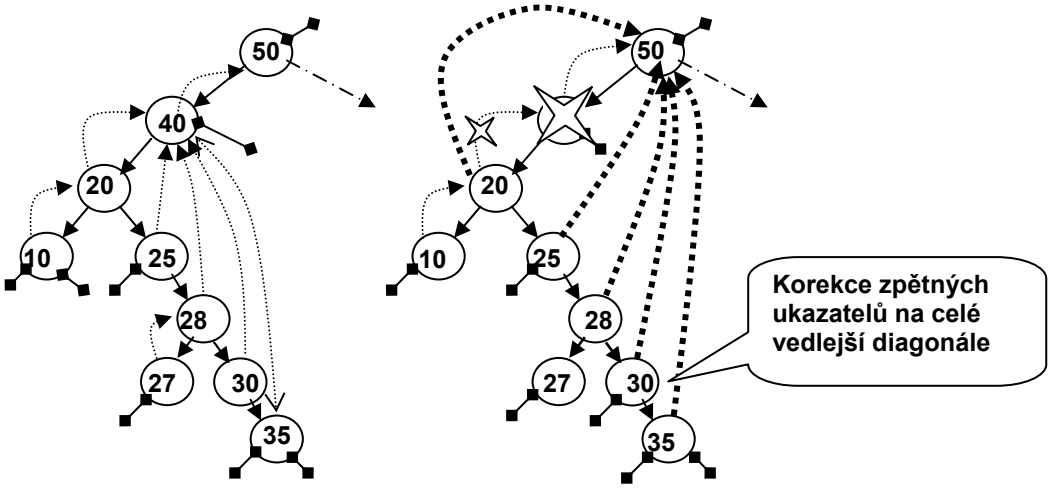
x+y

Pak procedura `InOrder` bude mít tvar:

```
void Inorder(TUk UkKor, TDlist* DL){
// Průchod Inorder vkládá data do dvojsměrného seznamu
  int Fin;
  TUk UkNaNejlev;
  DListInit (DL); // Inicializace dvojsměrného seznamu
  UkNaNejlev =Nejlev(UkKor);
  Fin=(UkNaNejlev==NULL);  // řídicí proměnná cyklu
  while (! Fin) {
    DInsertLast(DL, UkNaNejlev->Data); //vkládání do sezn.
    if (UkNaNejlev->PUk!=NULL)
        UkNaNejlev=Nejlev(UkNaNejlev->PUk);
    else if (UkNaNejlev->ZpetUk==NULL)
             Fin=true;
         else UkNaNejlev=UkNaNejlev->ZpetUk; // posun zpět
  } // while
} // Inorder
```

Delete

Operace Delete se provádí stejně, jako u normálního BVS, ale v případě rušení uzlu s jediným a to levým podstromem se musí provést korekce zpětných ukazatelů na celé levé diagonále rušeného prvku.
Situaci znázorňuje následující obrázek.



4.11 Kontrolní otázky a úlohy

- Vytvořte následující nerekurzívní varianty zápisu procedury pro vyhledávání v BVS:
    - funkce vrátí „Booleovský“parametr Search a ukazatel `UkKde` pro nalezený uzel (v případě `Search==false` je `UkKde` nedefinováno)
    - `int InsertSearch1(TUk UkKor, TKlic K, TUk* UkKde);`
    - procedura vrátí v `UkKde` ukazatel na nalezený uzel nebo NULL v případě neúspěšného vyhledávání.
    - `void InsertSearch2(TUk UkKor, TKlic K,TUk* UkKde);`
- Je dán BVS (nevyvážený). Je zadán (maximální) počet jeho uzlů. Vytvořte jeho váhově vyváženou verzi. Zapište řešení ve formě rekurzívní i nerekurzívní procedury. Zvolte vhodnou datovou strukturu uzlů a definujte potřebné typy.

Pozn. Řešte s použitím pomocného pole, do kterého vložíte všechny hodnoty nevyváženého stromu. Z pole pak vytvořte nový, váhově vyvážený strom.

- Proveďte diskuzi, za kterých okolností je výhodnější použít binární vyhledávání než vyhledávání v BVS.
- Čím je dán nejhorší případ přístupové doby v (nevyváženém) BVS?
- V čem spočívá princip a jakou má výhodu BVS se zarážkou?
- K čemu se používá BVS se zpětnými ukazateli?
- Čím se liší BVS se zpětnými ukazateli od normálního BVS?

4.13 TRP

4.13 Tabulky s rozptýlenými položkami - TRP (*hash table, hashing table*) jsou založeny na použití pole, které je primárním prostorem pro práci s tímto typem tabulek.

Tabulka s přímým přístupem

Základem TRP je princip **tabulky s přímým přístupem.**

Definice:

Nechť existuje množina klíčů **K** dané tabulky a množina sousedních míst (adres) v paměti **H,** které realizují vyhledávací tabulku. Existuje-li jedno-jednoznačná funkce mapující každý prvek první množiny do druhé množiny a naopak, pak hovoříme o tabulce s přímým přístupem. Této funkci se říká **mapovací funkce**.

Příklad: Kdyby množina klíčů byla dána intervalem 0..99 a tabulku reprezentovalo pole T[0..99], pak mapovací funkce pro klíč K je T[K]. Každý prvek pole musí mít navíc Booleovskou složku „obsazeno/volno“ , která určuje, zda daný prvek na své adrese je, či není. Při inicializaci se nastaví všechny prvky tabulky na „volno“.

Pak vyhledání spočívá v přímém zjištění, zda na pozici klíče (indexu) dané tabulky je obsazeno a pak tam prvek je nebo je volno a pak prvek v tabulce není.

Jinými slovy - každý prvek má své místo, to místo je přímo určitelné z klíče a na tomto místě prvek je, či není.

Časová složitost přístupu v přímé tabulce je 1.

Skutečnost, že se tabulky s přímým přístupem nepoužívají všeobecně spočívá v tom, že nalezení vhodné mapovací funkce je obtížné.

Mapovací funkce

Hledání vhodné **mapovací funkce** je obtížné. Uvádí se následující příklad: Pro 31 prvků, které se mají zobrazit do 41 prvkové množiny existuje 41^{31} tj. cca 10^{50} různých možných mapovacích funkcí. Přitom jen (41!/10!) z nich dává odlišné hodnoty pro různé klíče (jsou jedno-jednoznačné). Poměr „vhodných“ funkcí ku všem možným je tedy asi 1:10 000 000.

Situaci ilustruje příklad nazvaný **"Paradox společných narozenin“**.

Je dobrá naděje, že mezi 23 osobami, které se sejdou ve společnosti, se najdou dvě osoby, které mají narozeniny ve stejný den“.

Jinými slovy: Najdeme-li náhodně funkci, která mapuje 23 klíčů do tabulky o 365 prvcích, je pravděpodobnost, že se žádné dva klíče nenamapují do stejného místa rovna 0.4927. Naděje, že se do stejného místa namapují právě překročila hodnotu 0.5.

Jevu, kdy se dva různé klíče mapují do stejného místa říkáme **kolize.**
Dvěma nebo více klíčům, které se namapují do téhož místa říkáme **synonyma.**

Dobrá mapovací funkce musí splňovat dva požadavky:

- rychlost mapování
- vytváření co nejméně kolizí

Mapovací funkce transformuje klíč na index do primárního (rozptylového, hashovacího pole). Hodnota indexu musí být v daném intervalu rozsahu pole. Nejčastěji se mapovací funkce dělí do dvou etap:

- převod klíče na přirozené číslo (N>0)
- převod přirozeného čísla na hodnotu spadající do intervalu indexu pole (nejčastěji s použitím operace modulo).

Nechť K je celé číslo větší než nula. Pak funkce

```
h(K) = K mod MAX
```

získá hodnoty z intervalu 0..MAX-1 a funkce

```
h(K) = K mod MAX + 1
```

získá hodnoty z intervalu 1..MAX

Princip TRP

Princip tabulky s rozptýlenými položkami je obdobný principu **index-sekvenčního přístupu.**

Operace Insert:
Při vkládání nové položky se položka vloží na mapovací funkcí stanovený index primárního pole (nebo na místo stanovené položkou na tomto indexu), pokud je toto místo volné. Pokud je místo obsazené, jde o vkládání synonymního klíče. Synonyma se vkládají do sekvenčně uspořádané struktury (seznamu), jejímž prvním prvkem je první z vložených synonym.

Operace Search:
Při vyhledávání začíná mechanismus na položce pole, ke které se dostane indexem určeným mapovací funkcí. Je-li položka volná, znamená to "neúspěšné vyhledání" - položka neobsazená. Je-li položka obsazená, pak mechanismus začne prohledávat seznam synonym, začínající prvním prvkem. Pokud je hledaný prvek v seznamu synonym nalezen, je vyhledávání úspěšné, v opačném případě je neúspěšné.

Pozn. Mechanismus je "index-sekvenční", protože v prvním kroku "index" se dostává na začátek sekvence a ve druhém kroku "sekvenční" - se zahajuje sekvenční vyhledávání v posloupnosti synonym.

Explicitní a implicitní zřetězení synonym

Seznam synonym může být zřetězen explicitně nebo implicitně.

- Při explicitním zřetězení obsahuje každý prvek posloupnosti adresu následníka.
- Při implicitním zřetězení se adresa následníka získá hodnotou funkce adresy předchůdce

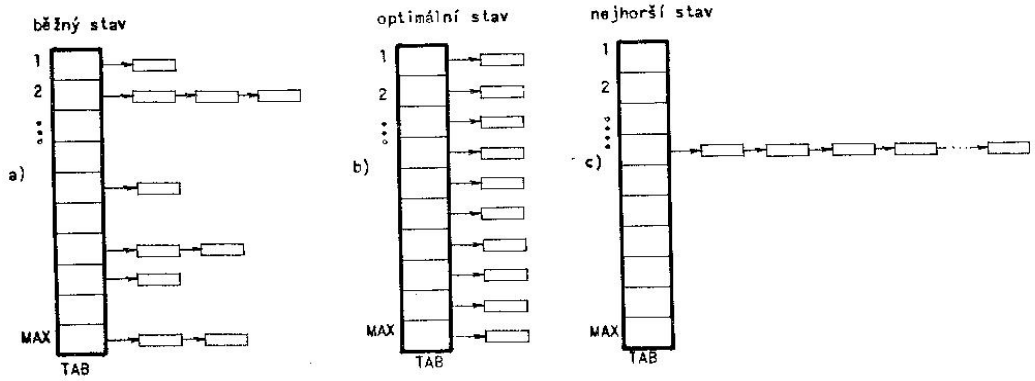
Tabulka s rozptýlenými položkami sestává z mapovacího prostoru (pole) a ze seznamů synonym. Každý seznam začíná na jednom prvku mapovacího pole.

TRP s explicitním zřetězením synonym

TRP s explicitním zřetězením synonym má podobu pole, v němž jsou uloženy začátky (ukazatele) na seznamy synonym.

Pozn. I první prvek vytváří jednoprvkový seznam synonym
Maximální doba vyhledávání je dána délkou nejdelšího seznamu synonym. Rušení prvků je stejné jako ve zřetězeném seznamu. Seznamy prvků mohou být seřazeny podle klíče. To urychlí neúspěšné vyhledávání v seřazeném seznamu.

Situaci znázorňuje následující obrázek, zobrazující běžný stav, optimální stav a nejhorší stav, v němž je TRP degradována na lineární seznam.



Vytvořte operace Insert, Search a Delete pro vyhledávací tabulku implementovanou TRP s explicitním zřetězením. Klíč je typu integer. Rozptylové pole má 100 prvků.

- **Rozptylové pole je pole ukazatelů na jednosměrné seznamy synonym.**
- **Rozptylové pole je pole ADT dvojsměrný seznam. Pro práci se seznamem použijte výhradně abstraktních operací nad ADT TDList.**

Klasifikace TRP

TRP lze klasifikovat na základě způsobu, jakým jsou vytvořeny seznamy synonym:

- TRP s explicitně zřetězenými synonymy
  - zřetězený seznam s využitím ukazatelů a DPP
  - zřetězený seznam v poli společném pro prvotní rozptýlení i pro prvky synonymních prvků (tzv. Knuthova metoda).
- TRP s implicitně zřetězenými synonymy v poli
  - s konstantním krokem
    - krok určuje programátor
    - krok určuje program
  - s proměnným krokem (kvadratická metoda)

Uvedený příklad a obrázek reprezentoval zřetězený seznam s využitím ukazatelů a DPP. Knuthova metoda je uvedena ve skriptech [1] na webové adrese: https://www.fit.vutbr.cz/study/courses/IAL/private/

TRP s pevným krokem

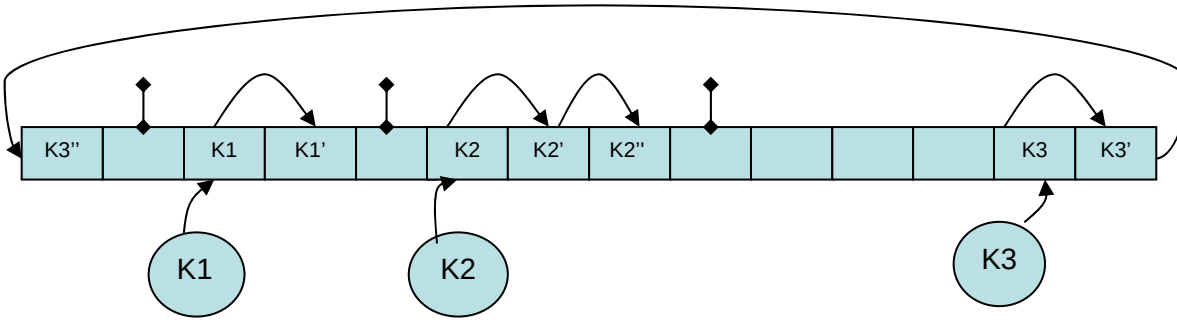
TRP s implicitním zřetězením s pevným krokem je implementována v poli**,** ve kterém jsou jak první prvky seznamu synonym, tak jejich další položky. Na obrázku je popsána tabulka, kde dalším prvkem seznamu je adresa o 1 větší

(pevný krok je 1), a kde

$$A_{i+1} = A_i + 1$$

"Seznam" synonym je tvořen po sobě jdoucími prvky a je ukončen první volnou položkou pole. Z toho vyplývají dvě zásady:

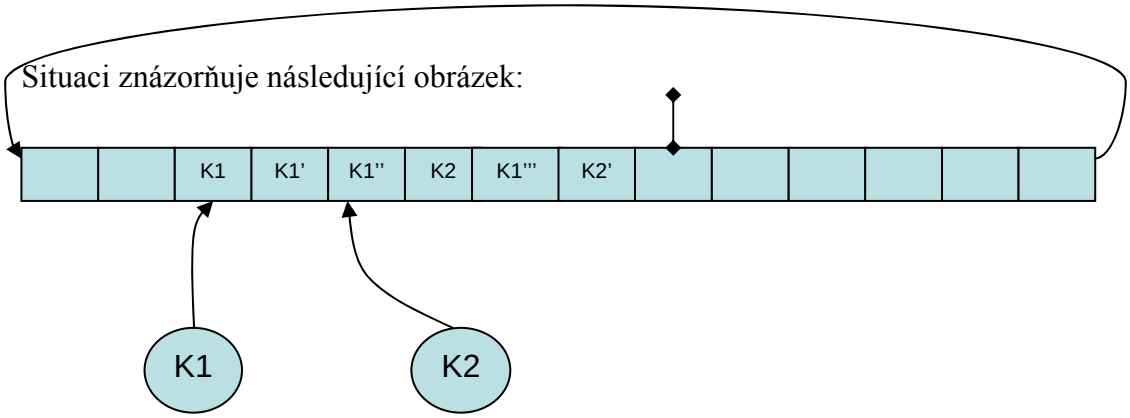
- S polem se pracuje jako s kruhovým seznamem!
- Položky pole musí být inicializovány tím, že se jejich indikátor obsazenosti nastaví na "volný"



Konec seznamu synonym je dán prvním volným prvkem, který se najde se zadaným krokem. Nové synonymum se uloží na první volné místo (na konec seznamu). Tabulka (pole) musí obsahovat alespoň jeden volný prvek. Efektivní kapacita je o 1 menší než je počet položek. Tabulka je implementovaná kruhovým polem.

Příklad:
Nechť jsou již do tabulky vloženy klíče K1, K1’ a K1’’.
Následně se klíč K2 namapoval do položky, která je obsazena (je tam klíč K1’’). Klíč byl uložen na první volné místo. Další klíč K1’’’ se namapoval do položky K1. První volné místo pro klíč K1’’’ bylo nalezeno za K2. Klíč K2’ se namapoval do položky K2. První volné místo se nalezlo za klíčem K1’’’.
V této tabulce se dva seznamy synonym překrývají. Prvek K1’’ je vstupním bodem seznamu synonym K2. Nelze je zrušit ani zaslepením.

Situaci znázorňuje následující obrázek:



Velikost pole Krok s hodnotou jedna má tendenci vytvářet “shluky“ (*cluster*). Výhodnější je krok větší než 1. Takový krok by ale měl mít možnost „navštívit“ všechny položky pole. Kdyby pole mělo sudý počet prvků, pak sudý krok (např. krok = 2) by z východiska lichého indexu mapovaného prvku mohl navštívit pouze liché indexy. Např. pole 0..9, krok 2, začátek na indexu 5 projde indexy: 5,7,9,1,3. Krok 4 a začátek na indexy 7 projde indexy: 7,1,5,9,3.

Kdyby měl krok hodnotu prvočísla, které je nesoudělné s jakoukoli velikostí pole, pak by mohl postupně projít všemi prvky pole. Výhodnější ale je, aby hodnotu prvočísla měla velikost mapovacího pole. Pak jakýkoli krok dovolí projít všemi prvky mapovacího pole.

Je vhodné dimenzovat velikost mapovacího pole TRP tak, aby bylo rovno **prvočíslu.**

Kvadratická metoda

Mezi metody s automatickou změnou kroku patří tzv. „kvadratická metoda". Hodnota kroku se v ní zvětšuje s každým krokem o 1. Při rozvinutí kruhového pole vytvářejí „navštívené" adresy kvadratickou funkci. Kvadratická metoda má předpoklady nevytvářet shluky.

Ve skriptech [1] je podrobnější popis dvou variant kvadratické metody. Z hlediska použití jsou tyto metody bezvýznamné. Za pozornost stojí nápad, kterým v historické době zlepšovali programátoři nevyhovující vlastnosti TRP.

TRP s dvojí rozptylovací funkcí

Metoda s dvojí rozptylovací (hashovací) funkcí patří mezi metody, v níž je krok v rozptylovacím poli určen programem – jeho druhou rozptylovací funkcí.

Nechť má rozptylovací pole rozsah 0..MAX, (kde hodnota MAX+1 je prvočíslo) a KInt je klíč transformovaný na celou hodnotu KInt>0, pak:

- První rozptylovací funkce vytváří hodnotu z intervalu 0..MAX-1 **ind= Kint mod MAX**. (pro určení indexu v poli)
- Druhá rozptylovací funkce vytváří krok s hodnotou z intervalu 1..MAX **krok = Kint mod MAX+1**.

Search

Vyhledávací cyklus operace **Search** začíná na indexu získaném první rozptylovací funkcí. Končí úspěšně při nalezení prvku s hledaným klíčem, neúspěšně při dosažení konce seznamu synonym (prázdným prvkem). Index následujícího prvku je dán přičtením kroku k aktuálnímu prvku při respektování kruhovosti pole:

$$ind_{i+1} = (ind_i + krok) \bmod MAX$$

Insert

Operace **Insert** vloží nový prvek na místo prvního prázdného prvku.

Delete

TRP s implicitním zřetězením se používají v aplikacích, v nichž se nepoužívá operace **Delete.** Pozn. Pokud je přeci jen zapotřebí prvek rušit, je možné použít mechanismu "zaslepení".

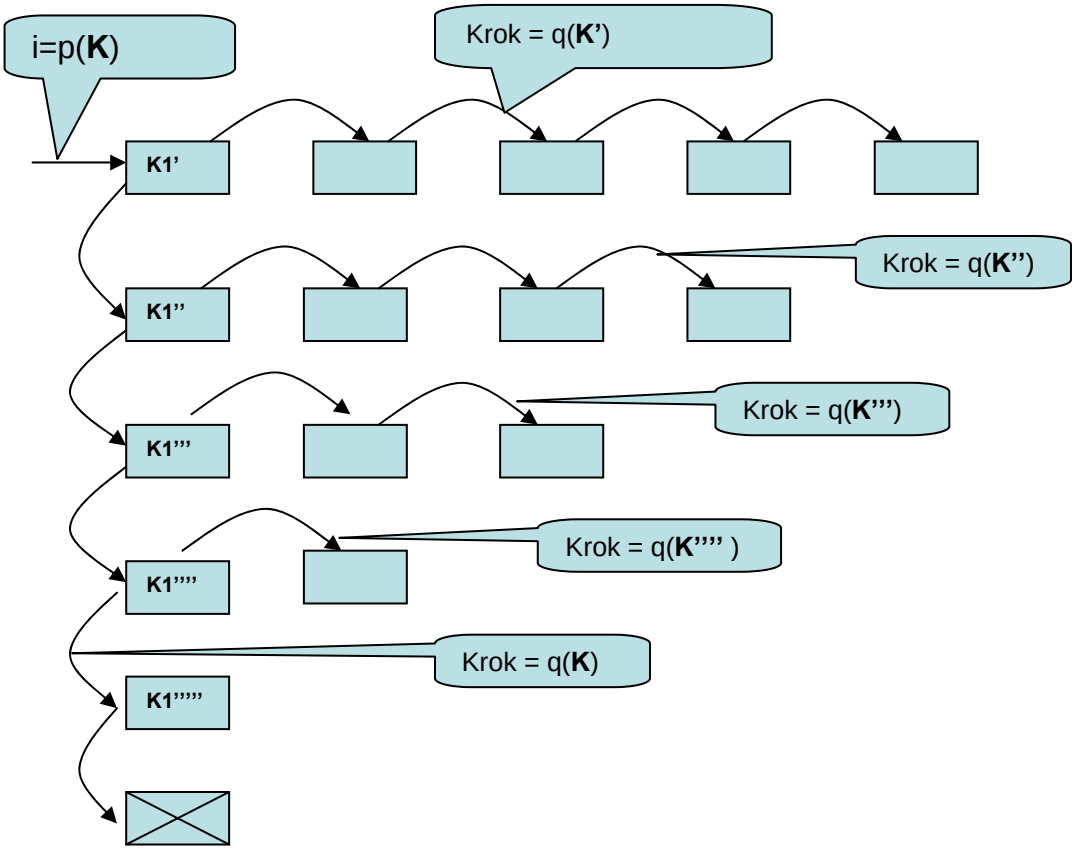
- Maximální kapacita TRP pro rozsah pole s indexy 0..MAX-1 je **MAX-1** (o 1 menší než počet prvků). Pozn. Alespoň jeden prvek musí zůstat jako „zarážka" vyhledávání.

- Pro efektivní použití TRP s implicitním zřetězením se pole tabulky dimenzuje tak, aby její maximální zaplnění, dané poměrem N_{akt}/MAX, nebylo větší než cca 0.6-0.7.

Brentova varianta

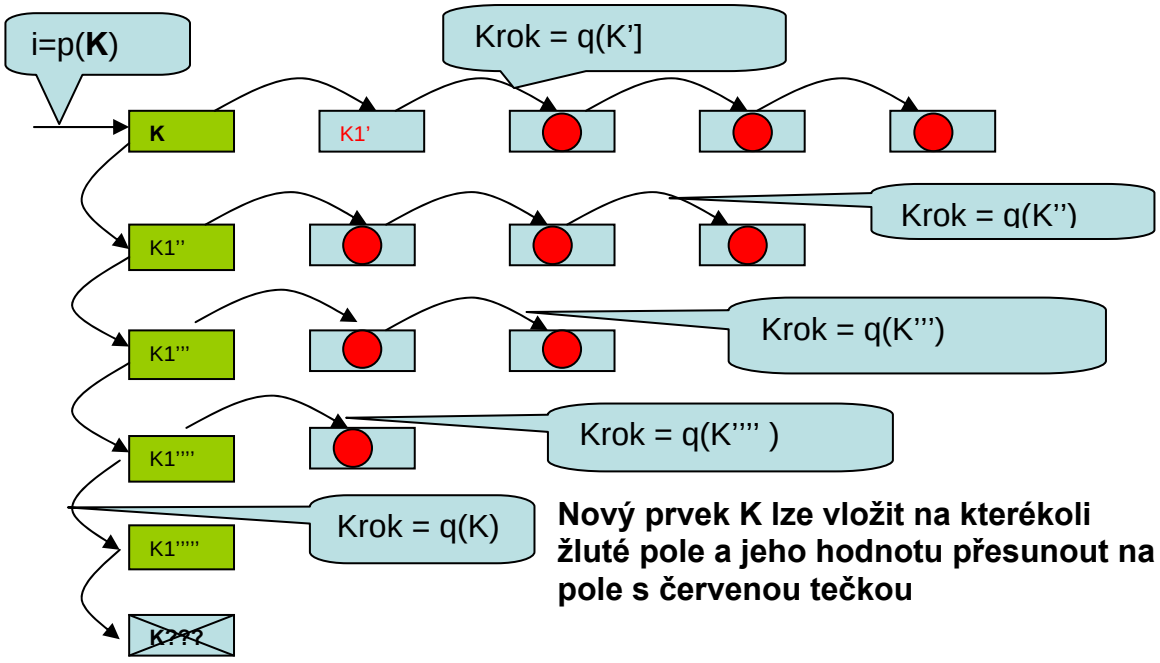
Brentova varianta je varianta metody TRP se dvěma rozptylovacími funkcemi. Princip vyhledávání (Search) je shodný s TRP se dvěma rozpt. funkcemi.

Brentova varianta je vhodná za podmínky, že počet případů úspěšného vyhledávání je častější, než počet neúspěšného vyhledání s následným vkládáním. Brentova varianta provádí při vkládání (operace Insert) dodatečnou rekonfiguraci prvků pole s cílem investovat do vkládání a získat tím lepší průměrnou dobu vyhledání.



Prvek K1' se přesune na první volné místo s krokem q(K1') a na jeho místo se vloží K.

Normální metoda dvojího hashování by nový klíč vložila na první volné místo s krokem q(k) – zde po 6 krocích.
Brentova varianta hledá první volné místo mezi červeně zakroužkovanými poli s krokem q(K1'), resp. q(K1'') atd. Na toto místo vloží prvek K1', resp. K1'' atd. a na uvolněné místo vloží prvek K.
Protože posun čelního prvku je menší než zde 5, je celková průměrná hodnota délky vyhledávání menší, než kdyby byl prvek K vložen na 6 pozici shora.



Nový prvek K lze vložit na kterékoli žluté pole a jeho hodnotu přesunout na pole s červenou tečkou

Hodnocení TRP

- Neexistuje obecné pravidlo jak nalézt nejvhodnější rozptylovací funkci. Dobrá rozptylovací funkce se může stanovit jen na základě znalosti vlastností množiny klíčů.
- Operaci Delete lze řešit pomocí „zaslepení“ – vložením klíče, který nebude nikdy vyhledáván.

Závěr **Závěr**

Čtvrtá kapitola představuje druhou stěžejní oblast předmětu. Prezentuje a vysvětluje nejvýznamnější algoritmy vyhledávání včetně jejich složitosti a stability. Znalosti této kapitoly patří k prerekvizitám řady následujících předmětů. Studenti se seznámí s rekurzívním i nerekurzívním způsobem zápisu vyhledávacích algoritmů tam, kde je to významné.

Příloha 5. CD