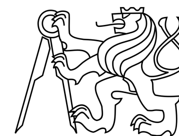
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název:	Návrh a implementace stromových datových struktur pro C++
Student:	Vladimír Vojá ek
Vedoucí:	Ing. Jan Trávní ek
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Nastudujte požadavky na stromové datové struktury algoritmu pro indexaci et zc Position Heap [1] a algoritmu vyhledávání v et zcích Aho-Corasick [2].
Navrhn te obecné rozhraní stromových abstraktních datových typ ve stylu standardní knihovny C++ zárove spl ující požadavky výše zmín ných algoritm .
Implementujte stromové abstraktní datové typy podporující Vámi navržená rozhraní.
Ov te použitelnost navržených rozhraní stromových abstraktních datových typ a jejich implementaci na implementaci algoritmu indexování et zc pomocí struktury Position Heap a algoritmu vyhledávání v et zcích Aho-Corasick.
Dále implementaci otestujte pomocí Vámi navržených jednotkových test .

Seznam odborné literatury

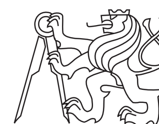
[1] Andrzej Ehrenfeucht, Ross M. McConnell, Nissa Osheim, Sung-Whan Woo, Position heaps: A simple and dynamic text indexing data structure, Journal of Discrete Algorithms, Volume 9, Issue 1, March 2011, Pages 100-121, ISSN 1570-8667, http://dx.doi.org/10.1016/j.jda.2010.12.001.
[2] Alfred V. Aho, Margaret J. Corasick, Efficient String Matching: An Aid to Bibliographic Search, Communications of the ACM, Volume 18, Issue 6, June 1975, Pages 333--340, ISSN 0001-0782, http://doi.acm.org/10.1145/360825.360855.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 7. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Fakulta informačních technologií



Bakalářská práce

Návrh a implementace stromových struktur v C++

Vladimír Vojáček

Vedoucí práce: Ing. Jan Trávníček

16. května 2017

Poděkování

Děkuji Ing. Janu Trávníčkovi za odborné vedení a konzultace, které mi poskytoval k této práci. Dále děkuji své rodině za průběžnou podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2017 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Vladimír Vojáček. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Vojáček, Vladimír. *Návrh a implementace stromových struktur v C++*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Ve své bakalářské práci jsem se zaměřil na návrh a implementaci stromových struktur v programovacím jazyce C++. Struktury jsem navrhl s ohledem na analýzu referenčních algoritmů Aho Corasick a Position heap. Na základě návrhu jsem struktury implementoval a jejich správnou funkci otestoval na zmíněných algoritmech. Čtenář se může v této práci seznámit s návrhem a připravenými strukturami, které může v této podobě použít, nebo na jejich základě vytvářet vlastní struktury se specializovanými požadavky.

Klíčová slova strom, datová struktura, stromová struktura, trie, Aho Corasick, Position heap, C++

Abstract

I focused on design and implementation of tree data structures in programming language C++ in my bachelor thesis. I designed structures according to analysis of reference algorithms Aho Corasick and Position heap. I implemented structures on the base of design and their proper function tested on mentioned algorithms. Reader can introduce with design of prepared data structures, which can be used in this form, or modified to special requests.

Keywords tree, data structure, tree structure, trie, Aho Corasick, Position heap, C++

Obsah

Úvod **1**

1 Analýza **3**
1.1 Definice stromu . . . 3
1.2 Algoritmus Aho Corasick . . . 5
1.3 Algoritmus Position heap . . . 8

2 Návrh a implementace stromových struktur **11**
2.1 Mapování . . . 11
2.2 Průchod stromovou strukturou . . . 14
2.3 Odstranění uzlu pomocí iterátoru . . . 21
2.4 Rozhraní pro průchod strukturou . . . 21
2.5 Implementované struktury . . . 22

3 Implementace algoritmů **27**
3.1 Implementace algoritmu Aho Corasick . . . 27
3.2 Implementace algoritmu Position heap . . . 28

4 Testování **31**
4.1 Testování mapování a iterátorů . . . 31
4.2 Testování algoritmů Aho Corasick a Position heap . . . 31

Závěr **33**

A Obsah přiloženého CD **35**

Seznam obrázků

1.1 Stromová struktura 5
1.2 Prefixový strom 6
1.3 Prefixový strom po vložení vzoru abc 6
1.4 Vyhledávací automat Aho Corasick 6
1.5 Position heap pro řetězec dddabcabc 9

Seznam tabulek

Úvod

Stromové struktury jsou hierarchicky organizované struktury, které široce využíváme nejen v informatice. V běžném životě je klasickým příkladem stromové struktury rodinný rodokmen, kterým reprezentujeme vztahy mezi předky a potomky, nebo organizační struktura firmy, ve které jsou někteří zaměstnanci podřízeni jiným. Díky svému charakteru je stromová struktura ideální na rekurzivní způsob řešení úloh.

Podstatou této bakalářské práce je návrh a implementace stromových struktur v programovacím jazyce C++. Požadavkem jsou různé implementace stromové struktury, například mapováním do pole, a implementace stromových struktur, které obsahují data v uzlech nebo na hranách. Současná řešení nesplňují požadavky na stromové struktury analyzovaných algoritmů. Existuje implementace n-árního stromu, knihovna `tree.hh`, která implementuje pouze jednu konkrétní stromovou strukturu a tím nesplňuje požadavky umístění dat do stromové struktury nebo efektivní vyhledání mezi potomky uzlu. Práce má přinést rozšiřitelný návrh stromových struktur, návrh jejich rozhraní a připravené stromové struktury.

Praktickou částí této práce je navrhnuté stromové struktury implementovat a ověřit funkčnost na algoritmech Aho Corasick a Position heap. Analýze zmíněných algoritmům se věnuji v první kapitole.

Ve druhé kapitole se zabývám návrhem stromových struktur v C++ a vysvětluji postup samotné implementace. Při návrhu jsem kladl důraz na návrh umožňující stromové struktury snadno upravovat ke kontrétním účelům. K dispozici jsou různé implementace mapování do stromové struktury, které implementují společné rozhraní popsané v kapitole týkající se mapování. Jednou z výhod tohoto řešení je skutečnost, že data lze do struktury stromu integrovat odlišnými způsoby. Je možné uložení do uzlů nebo na hrany, možné jsou i různé kombinace, které jsem v práci nastínil.

Poslední kapitola je zaměřena na popis testování navrhnutých struktur. Ověřuji jak správné vytvoření vztahů ve stromové struktuře přidáváním a odebíráním uzlů, tak rovněž procházení struktury pomocí iterátorů. Použitelnost

struktur je testována na referenčních algoritmech.

Výsledkem práce jsou implementované stromové struktury a implementované referenční algoritmy. Popsal jsem způsob použití a také postup pro případná rozšíření o další implementace mapovacích tříd nebo konkrétních stromových struktur.

Téma jsem si vybral, protože výsledkem je software, který bude použitelný k řešení nejrůznějších úloh, které vyžadují práci se stromovými strukturami. Také je mi blízká praktická část této bakalářské práce. Programovací jazyk C++ je můj oblíbený.

Kapitola 1

Analýza

1.1 Definice stromu

Strom T je hierarchická struktura sestávající se z hran a uzlů. Na začátek uveďme definici struktury vymezující základní pojmy hierarchie mezi uzly ve stromové struktuře.

- pokud T není prázdný, obsahuje uzel k, který nemá rodiče
- k nazýváme kořenem stromu
- každý uzel, který není k, má rodiče r
- každý uzel, který má rodiče r, je potomkem r

Písmenem r budeme v práci dále označovat uzel rodiče. Uzel r má uspořádanou n-tici potomků $P_r = (p_1, p_2, p_3, ..., p_n)$. Písmenem p budeme označovat potomka r. Pro určení i-tého potomka p_i, označíme uzel dolním indexem tak, že $i \in [1, |P_r|]$.

1.1.1 V teorii grafů

Stromovou strukturu známe z teorie grafů [?] jako souvislý acyklický graf. Nechť T_n je graf s n vrcholy. Potom jsou následující trvzení [?] ekvivalentní:

- T_n je strom
- T_n je acyklický a souvislý
- T_n je acyklický a má $n - 1$ hran
- T_n je souvislý a má $n - 1$ hran

Sled Pro dvojici uzlů a, $b \in G$ nazveme sledem střídavou posloupnost uzlů a hran, která začíná v a a končí v b.

Tah Sled, ve kterém se neopakuje žádná hrana, nazýváme tahem.

Cesta Tah, ve kterém se neopakuje žádný uzel, nazýváme cestou.

Kružnice Kružnice je cesta, která začíná a končí v témže uzlu.

Acyklický graf Acyklickým grafem nazýváme graf, který neobsahuje kružnici.

Souvislý graf Souvislý graf je takový graf, který obsahuje pro libovolnou dvojici uzlů alespoň jednu cestu.

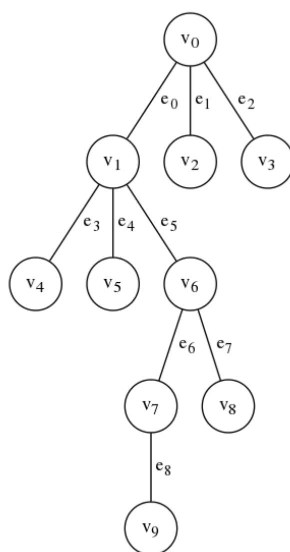
Z definice vyplývají mimo jiné tyto tvrzení, která nám blíže představují strukturu stromu.

- mezi každými dvěma vrcholy existuje právě jedna cesta
- $|V| = |E| + 1$, kde V je množina uzlů, E je množina hran G
- odebráním libovolné hrany dostáváme nesouvislý G
- přidáním hrany k libovolné dvojici uzlů vznikne v G kružnice

1.1.2 Vztahy ve struktuře stromu

Zaveďme pro uzly $p, r, k \in T$ následující pojmy označující vztahy mezi uzly ve stromové struktuře.

- následník uzlu p - každý dosažitelný uzel z p od kořene
- předchůdce uzlu p - každý uzel na cestě z p do k, pro $p = v_6$ dostaneme množinu předchůdců $\{v_1, v_0\}$
- rodič uzlu p - bezprostřední předchůdce p na cestě ke k
- potomek uzlu p - bezprostřední následník p na cestě od k
- sourozenec uzlu p - uzel, který má společného rodiče s p, pro $p = v_1$ dostáváme množinu sourozenců $\{v_1, v_2, v_3\}$ se společným rodičem v_0
- podstrom uzlu p - část stromu tvořená kořenem p a následníky p, pokud $p = v6$, tak podstromem p je množina uzlů $\{v_6, v_7, v_8, v_9\}$
- list - uzel, který nemá žádného potomka, množinou listů stromu 1.1 je $\{v_4, v_5, v_9, v_8, v_2, v_3\}$



Obrázek 1.1: Stromová struktura

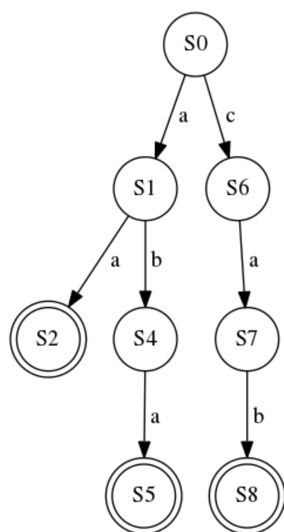
1.2 Algoritmus Aho Corasick

Aho Corasick [**?**] je algoritmus pro vyhledávání množiny vzorů $P = \{P1, ..., P_k\}$ v textu $T[1...n]$. Algoritmus konstruuje vyhledávací automat, který dokáže zpracováním vstupního textu detekovat všechny výskyty P. Vyhledávací automat je složený z dopředné funkce δ, zpětné funkce f a funkce výstupní out. Dopředná funkce mapuje stav S automatu a symbol $s \in \Sigma$ na stav S', tedy $\delta(S, s) = S'$ znamená, že existuje přechod ze stavu S do stavu S' na symbol s. Zpětná funkce f mapuje stav na jiný stav automatu a používáme jí pokud nemáme δ. Výstupní funkce out mapuje stav automatu S na množinu vzorů $P_S \subset P$, jejichž výskyt má být ve stavu S detekován.

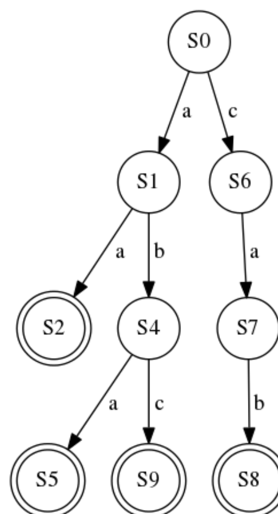
Hledat lze lineární složitostí vzhledem k délce vstupního textu, tj. algoritmus je schopný detekovat P jedním průchodem T. Výhodou je, že může být automat pro P sestaven předem, uložen a poté používán.

1.2.1 Algoritmus dopředné funkce

Pro každý vzor $P_i \in P$ vkládáme symboly $s_1, s_2, ..., s_{|P_i|} \in P_i$ do stromu. Při vkládání se snažíme využít již existující přechody, tj. přecházíme, pokud $\delta(S, s_j) \neq 0$. V případě, že přechod neexistuje, vytváříme nový stav S_n, definujeme $\delta(S, s_j) = S_n$ a přecházíme do S_n. Uzel S_k, do kterého se dostaneme posledním symbolem $s \in P_i$, označíme jako koncový pro P_i, tj. definujeme výstupní funkci $out(S_k) = i$. Přidání všech vzorů nám zabere $\sum_{n=1}^{|P|} |P_i|$, tj. $\mathcal{O}(n)$. Obrázek 1.2 představuje prefixový strom vytvořený pro množinu vzorů $P = \{aa, aba, abc, cab\}$. V uzlech $\{S_2, S_5, S_8\}$ jsou detekovány hledané vzory.



Obrázek 1.2: Prefixový strom pro množinu vzorů $P = \{aa, aba, abc, cab\}$.

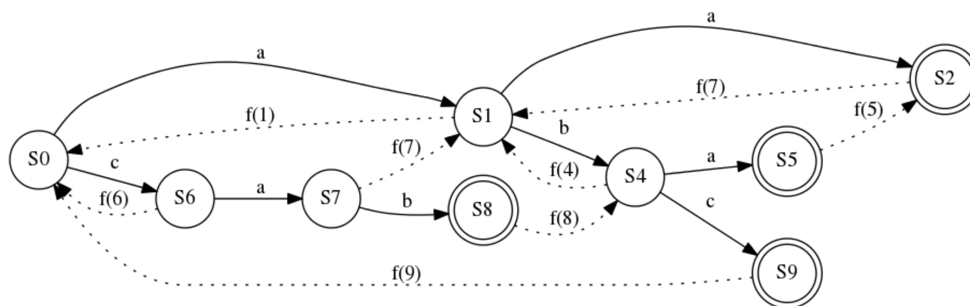


Obrázek 1.3: Prefixový strom po vložení vzoru *abc*.

Stav stromu po vložení vzoru *abc* zachycuje. 1.3. Při vkládání využijeme již existující přechody $\delta(S_0, a)$, $\delta(S_1, b)$, neboť prefix $ab \in p$ je již součástí stromu. Definujeme pouze $\delta(S_4, a) = S_5$ a označíme S_5 jako koncový pro p.

1.2.2 Konstrukce zpětné funkce

Zpětnou funkci f používáme, pokud v uzlu S na symbol s platí $\delta(S, s) = 0$, tj. nemáme definovanou dopřednou funkci. Při konstrukci zpětné funkce do-



Obrázek 1.4: Vyhledávací automat pro množinu vzorů $\{aa, aba, abc, cab\}$. Dopředná funkce δ je označena černou čarou, zpětná funkce f přerušovanou čarou. Stavy, ve kterých detekujeme vzory, jsou označeny dvojitou čarou.

končíme funkci výstupní, aby automat detekoval všechny výskyty vzorů v T, které jsou podřetězcem jiného vzoru. Algoritmus konstrukce zpětné funkce [**?**] je popsaný níže.

Strom projdeme do šířky pomocí fronty q, kterou inicializujeme vložením uzlů $\delta(S_0, s) \neq 0$ pro všechny symboly $s \in \Sigma$. Pro uzel S_A, vybraný z q, testujeme $\delta(S_A, s)$ pro všechny symboly $s \in \Sigma$. Pro definovaný přechod na symbol s do uzlu $\delta(S_A, s)$ ověřujeme $\delta(f(S_A), s) \neq 0$. Pokud neexistuje, pak $S_B = f(S_A)$ a rekurzivně používáme zpětnou funkci $S_B = f(S_B)$, dokud nezjistíme $\delta(S_B, s) = S_C$. Poté definujeme $f(\delta(S_A, s)) = S_C$ a přidáváme uzel $\delta(S_A, s)$ do q. V nejhorším případě, pokud se nám cestou ke kořeni nepodařilo přejít, používáme f až do S_0, kde $\delta(S_0, s) = S_0$ pro všechny $s \in \Sigma$. Pokud $P_i \in out(\delta(S_B, s))$, P_i bude detekován také v $\delta(S_A, s)$. Postup opakujeme, dokud q není prázdná.

Algorithm 1 Algoritmus konstrukce zpětné funkce

```
queue ← empty
for každý symbol s takový, že δ(0, s) = S ≠ 0 do
    queue ← queue ∪ {S}
    f(S) ← 0
end for
while queue ≠ empty do
    nechť je r první prvek queue
    queue ← queue − {r}
    for každý symbol s takový, že δ(r, s) = S ≠ fail do
        queue ← queue ∪ {S}
        state ← f(r)
        while δ(state, s) = fail do
            fail ← f(state)
        end while
        f(S) ← δ(state, s)
        output(S) ← output(S) ∪ output(f(S))
    end for
end while
```

1.2.3 Algoritmus hledání

Automat prochází vstupní text symbol po symbolu. Používáme dopřednou funkci δ, pokud je definována, jinak se vracíme zpětnou funkcí f směrem ke kořeni. V počátečním stavu S_0 automat setrvá dokud pro načtený symbol s $\delta(S_0, s) = 0$. Pro všechny stavy, ve kterých se automat během zpracování T nachází, detekujeme výskyt P_i pokud $P_i \in out(S)$.

Pro $T = aaaabc$ je zpracování automatem na obrázku 1.4 následující. Zpracování začneme v počátečním stavu S_0. Načteme s_1 z T, $\delta(S_0, a) = S_1$, proto

přecházíme do S_1. Načteme s_2, dopřednou funkcí $\delta(S_1, s_2)$ přecházíme do stavu S_2 a detekujeme výskyt vzorku *aa*, neboť $out(S_2) = \{1\}$. Načteme s_3, tj. symbol a. Dopředná funkce $\delta(S_2, s_3) = 0$, proto přecházíme zpětnou funkcí do stavu $f(S_2) = S_1$. Ve stavu S_1 je dopředná funkce $\delta(S_1, s_3)$ definována, přecházíme do stavu S_2 a detekujeme další výskyt *aa*. Dalším načteným symbolem je b, pro který není ve stavu S_2 definována δ, vracíme se zpětnou funkcí do stavu S_1, ve kterém $\delta(S_1, b) = S_4$. Posledním symbolem je c, přecházíme do stavu S_9 a detekujeme výskyt vzoru *abc*.

1.2.4 Výsledek analýzy

Z popsaného fungování algoritmu plyne, že pro efektivní implementaci vyhledávacího automatu Aho Corasick potřebujeme stromovou strukturu s efektivním vyhledáváním uzlu p tak, že $\delta(r, s) = p$ pro $s \in \Sigma$. Data na hranách stromové struktury reprezentují symboly abecedy. Dále potřebujeme implementovat zpětnou funkci $f(S_i) = S_j$ pro $S_i, S_j \in T$, a seznam identifikátorů vzorů pro každý uzel, které mají být v daném uzlu detekovány. Tyto informace lze považovat za data uzlu.

1.3 Algoritmus Position heap

Position heap [**?**] je datová struktura pro indexaci řetězce T, kterou využíváme pro vyhledávání vzoru v T. Position heap $H(T)$ vytvoříme postupným vkládáním přípon (T_1, T_2,... T_n) řetězce T tak, že pro každou příponu T_i vytvoříme uzel odpovídající nejkratšímu prefixu T_i, který ještě není v $H(T)$ zaindexovaný. Uzel označíme indexem i, který reprezentuje výskyt T_i v T.

1.3.1 Konstrukce

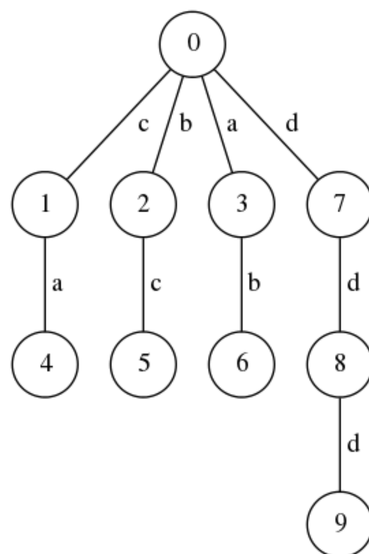
Do stromu vkládáme přípony T od nejkratší po nejdelší. Následující přípona T_i tak bude vždy delší než poslední T_{i-1} zaindexovaná přípona. Tím zabráníme, aby nám v uzlech vznikly množiny indexů. Pro všechny přípony řetězce začínáme v kořeni a snažíme se využít již existující přechody ve struktuře, poté vytvoříme index i pro T_i. Vložení indexu pro příponu je vždy $\mathcal{O}(h(T))$. Pro všechny pozice v T dostáváme $\mathcal{O}(nh(T))$.

Tabulka 1.1: Řetězec dddabcabc s indexy

10	9	8	7	6	5	4	3	2	1
d	d	d	d	a	b	c	a	b	c

Indexace řetězce *dddabcabc* proběhne následovně. První příponou je řetězec *c*. V kořeni nelze přejít na žádný symbol, vytvoříme proto nový uzel, do kterého uložíme index 1. Dále vkládáme příponu *bc*. Přechod v uzlu 0 není pro symbol *b* definován, vytváříme nový uzel 2. Poté přidáváme příponu *abc*, přechod pro

symbol a není definován, vytváříme nový uzel 3. Dále vkládáme příponu $cabc$. Pro první symbol c je přechod z kořene definován, dostáváme se do uzlu 1. Na další symbol přípony a není možné přejít, vytváříme nový uzel 4. Obdobným postupem pokračujeme pro zbývající přípony.



Obrázek 1.5: Position heap pro řetězec dddabcabc

1.3.2 Vyhledávání

Algoritmus hledání vrací množinu výskytů S hledaného vzoru ve vstupním textu T [?].

- Nechť S je množina výskytů p ve vstupním textu T
- p' je nejdelší prefix vzoru p, kterým se dostaneme do uzlu P' v H
- $S1$ buď množina pozic z P'
- $S2 \subseteq S1$, kde jsou prvky $S2$ výskyty p v T
- pokud $p' = p$, potom $S3 = \emptyset$, neboť není přidán podstrom uzlu P', jinak buď $S3$ množina následníků uzlu P'
- $S = S2 \cup S3$

Příkladem hledání pro hledaný řetězec $p = abc$. Na první symbol a přecházíme z kořene stromu do uzlu 3, poté do uzlu 6, čímž dostáváme množinu potencionálních výskytů $S1 = \{3, 6\}$. Ověřením pro tyto indexy zjistíme, že

$S1 = S2$, neboť všechny pozice z $S1$ reprezentují výskyt p v T. Množina $S3 = \emptyset$. Výsledkem je množina $S = \{3, 6\}$

Jiným příkladem je vyhledání řetězce $p = dd$. Pro první symbol můžeme přejít do uzlu 7 a poté na další symbol do uzlu 8, tj. $S1 = \{7, 8\}$. Ověřením zjistíme, že $S1 = S2$. Protože $p = p'$, potom $S3 = \{9, 10\}$, neboť následníci uzlu P' reprezentují výskyt p v T. Naopak hledáním řetězce *dda* bychom se dostali rovněž do uzlu 8, ale ověřením na pozicích 7 a 8 bychom výskyt p v T nedetekovali. Popsaný algoritmus je naivní. Jeho složitost je $\mathcal{O}(|p|^2 + k)$, kde k je počet výskytů řetězce p v T.

1.3.3 Výsledek analýzy

Obdobně jako u algoritmu Aho Corasick potřebujeme najít potomka p uzlu r podle klíče k tak, že $\delta(r, k) = p$. Analýzou jsme zjistili, že díky vlastnostem struktury lze k z řetězce T získat v konstantním čase. Dalším požadavkem na stromovou strukturu Position heap je uložení indexů do uzlů struktury.

Kapitola **2**

Návrh a implementace stromových struktur

2.1 Mapování

Mapování do stromové struktury znamená vytvoření správné hierarchální struktury mezi uzly. V práci popíšeme dva odlišné způsoby reprezentace stromové struktury. Prvním je reprezentace stromu v poli. Uzly mají odkazy na jiné uzly v podobě indexů do pole. Druhou implementací mapování je spojová reprezentace, ve které jsou uzly na různých místech v paměti a jsou propojeny pomocí ukazatelů. Mapovací třída zajišťuje vytvoření vztahů mezi uzly, tj. správné nastavení odkazů ve struktuře, a také samozřejmě správu uzlů v paměti.

V návrhu je odděleno vytváření stromové struktury do samostatného celku, který je univerzálně použitelný a nezávislý na datové části stromové struktury. Třídy realizující vytváření stromové struktury mají generický parametr reprezentující uzel. Je proto možné strukturu a rozhraní uzlu definovat podle potřeby. Požadavkem je, aby uzel implementoval rozhraní uzlu pro příslušné mapování.

1. Reprezentace stromové struktury mapováním do pole pomocí třídy `tree_array_mapper`. Každý uzel má odkaz na prvního potomka, rodiče, levého a pravého sourozence. Tato struktura je implementována v uzlu `array_node_t`.

2. Spojová reprezentace stromové struktury pomocí třídy `tree_linked_mapper`. Každý uzel `linked_node_t` má seznam ukazatelů na své potomky.

Pro definici uzlu je nutné rozhraní podědit z těchto tříd podle vybraného mapování stromové struktury. Třídu uzlu pak použijeme jako generický parametr mapovací třídy. Uzlu je nutné definovat konstruktor o dvou argumentech.

Prvním argumentem je přidělený identifikátor a druhým je ukazatel na mapovací třídu.

Listing 2.1: Ukázka definice třídy uzlu a použití uzlu s mapovací třídou.

```
class test_node_t : public array_node_t {
public:
    test_node_t(uint32_t id, tree_mapper* mapper);
        : array_node_t(id, mapper) { }
};

tree_array_mapper<tree_node_t> t;
```

Identifikace Mapovací třída přidělí každému uzlu interní celočíselným identifikátor. Kořen stromu má vždy identifikátor nastavený na 0. Identifikátor lineárně roste s rostoucím počtem uzlů, tedy standardně odpovídá pořadí, ve kterém byl uzel do stromu vložen. Toto pořadí může být porušeno, pokud byl některý uzel ze stromu vymazán. Identifikátor takového uzlu bude uvolněn, včetně všech identifikátorů uzlů jeho podstromu. Pro indexaci uzlu mají mapovací třídy instanci třídy indexer, která spravuje identifikátory. Rozhraní obsahuje metodu pro rezervaci identifikátoru, pokud je uzel přidáván do stromu, a metodu pro uvolnění identifikátoru v případě, že je uzel ze stromu odstraněn. Dále obsahuje metodu pro zjištění nejmenšího dostupného identifikátoru. Všechny třídy, které implementují vytváření stromové struktury, se na třídu indexer odkazují při volání metody `size`, která slouží pro zjištění počtu uzlů ve stromové struktuře.

Rozhraní třídy indexer

- `bool set(uint32_t index)` – nastaví bit na indexu zadaném parametrem na $true$
- `bool unset(uint32_t index)` – nastaví bit na indexu zadaném parametrem na $false$
- `bool get(uint32_t index, bool& value)` – na platném indexu vrátí ve vstupně-výstupním argumentu hodnotu bitu na zadaném indexu
- `uint32_t find_lowest_unset()` – vrátí nejnižší volný identifikátor

Důležité metody ze společného rozhraní mapovacích tříd

- `void alloc_root();` – vytvoří kořen stromové struktury

- `bool add_child(uint32_t parent, uint32_t& child_id);` – implementuje přidání uzlu do stromové struktury, prvním argumentem je identifikátor rodiče, do druhého argumentu je v případě úspěšného přidání zapsán identifikátor přidaného uzlu
- `bool remove(uint32_t id);` – implementuje odebrání uzlu ze stromové struktury
- `bool get_child_id(uint32_t parent, uint32_t child_index, uint32_t& req_child_id);` – implementuje nalezení identifikátoru potomka na zadaném indexu, prvním argumentem je identifikátor rodiče, druhý argumentem je index. V případě že potomek na zadaném indexu existuje, je do třetího argumentu zapsán jeho identifikátor

2.1.1 Mapování do pole

Mapováním do pole rozumíme reprezentaci stromové struktury, ve které má každý uzel odkaz na svého prvního potomka, rodiče a oba sourozence. K implementaci stromu stačí pouze odkaz na prvního potomka a pravého sourozence, ale pro efektivnější implementaci iteratorů je výhodnější konstruovat provázanější strukturu. Pro efektivnější implementaci vkládání uzlů do spojového seznamu mají uzly odkaz na posledního potomka.

Třída uzlu implementující rozhraní pro mapování do pole je `array_node_t`. Implementuje metody `get_first_child`, `set_first_child` pro získání a nastavení odkazu na prvního potomka, obdobně metody pro nastavení pravého a levého sourozence.

2.1.1.1 Přidávání uzlu

Při přidávání uzlu p k uzlu r mohou nastat dvě situace, které musíme vyřešit.

- uzel r je listem, nastavíme potomka p jako prvního potomka r
- jinak přidáváme p na poslední pozici spojového seznamu potomků, nastavíme odkaz přidávanému uzlu p_i na levého potomka p_{i-1}, a p_{i-1} na pravého potomka p_i

Vzhledem ke skutečnosti, že identifikátor uzlu odpovídá jeho indexu v poli, nalezneme r v $\mathcal{O}(1)$. Operace přidání p jako prvního potomka r zvládneme také v $\mathcal{O}(1)$. Rovněž přidání na konec spojového seznamu potomků r je $\mathcal{O}(1)$ operace, protože v r udržujeme odkaz na posledního potomka.

2.1.1.2 Mazání uzlu

Odstraňovaný uzel p musíme odstranit ze spojového sezanmu potomků rodiče r. Uzel r lze nalézt v konstatním čase, protože p má odkaz na r. Složitost

odstranění je p ze spojového seznamu potomků v r je $\mathcal{O}(P_r|)$, protože musíme v nejhorším případě projít celý spojový seznam potomků r. Odstraňujeme-li uzel p_i, mohou nastat následující situace.

- $i = 1$, odstraňujeme uzel p_1, který je prvním potomkem pro r, zneplatníme odkaz na prvního potomka v r
  - pokud má r více než jednoho potomka, aktualizujeme odkaz na prvního potomka na uzel p_2
  - jinak zneplatníme odkaz na prvního potomka, r se stává listem
- $i \in [2, |P_r|]$ musíme p_i odstranit ze spojového seznamu potomků, pokud $i < |P_r|$, pak musíme správně přenastavit odkaz na pravého potomka v p_{i-1} a odkaz na levého potomka v p_{i+1}, pokud je p_i posledním potomkem, stačí zneplatit odkaz na pravého sourozence v p_{i-1}

2.1.2 Mapování spojově

Spojovou reprezentací nazýváme reprezentaci stromu, ve které má každý uzel seznam ukazatelů na své potomky. Mapovací třída má ukazatel na kořen. Naivní výběr uzlu ze struktury stromu je $\mathcal{O}(n)$ operace pro n uzlů, protože je nutné projít celý strom. Pro efektivní výběr uzlu implementace obsahuje pole ukazatelů na uzly, ve kterém je identifikátor uzlu indexem. Poté dostáváme pro výběr uzlu konstatní složitost.

Třída uzlu pro spojové mapování je `linked_node_t`. Obsahuje `std::list` ukazatelů na potomky. Implementuje metody `add_child` pro přidání potomka na konec spojového seznamu a `unset_child` pro odstranění potomka, které jsou využívány mapovací třídou pro správu uzlů.

2.1.2.1 Přidávání uzlu

Přidávaný uzel p přidáme na konec spojového seznamu v r. Přidání uzlu do spojového seznamu potomků implementuje metoda uzlu `add_child`.

2.1.2.2 Mazání uzlu

Mazaný uzel je nutné odstranit z r, k čemuž slouží metoda `unset_child`. Při mazání je s mazaným uzlem odstraněn zároveň celý jeho podstrom. Je nezbytné projít podstrom mazaného uzlu, aby bylo možné uvolnit identifikátory potomků.

2.2 Průchod stromovou strukturou

Průchod stromovou strukturou je realizován pomocí iterátorů. K dispozi jsou iterátory umožňující průchod stromovou strukturou v pořadí preorder a postorder. Dále jsou k dispozici iterátory, které procházejí stromovou strukturu

přes listy, a iterátor procházející potomky zadaného uzlu. Pomocí iterátorů můžeme do stromové struktury přidávat nové uzly, nebo stávající uzly odebírat. Iterátory pracují se společným rozhraním mapovacích tříd a proto pracují nezávisle na implementaci mapování stromové struktury. Zejména to jsou metody pro zjištění ukazatele na uzel a výběr potomka na zadaném indexu.

- template <class T> T* get(uint33_t id) – metoda pro výběr uzlu S_{id} z mapovací třídy, vrací ukazatel na S_{id}, pokud $S_{id} \in T$, jinak *nullptr*
- bool get_child_id(uint32_t parent, uint32_t child_index, uint32_t& req_child_id) – metoda pro získání identifikátoru potomka na zadaném indexu, již byla popsána výše

Společné rozhraní pro iterátory

- bool next() – nastaví iterátor na následující prvek, pro iterátor na posledním uzlu v posloupnosti dojde k zneplatnění (end iterátor)
- bool prev() – nastaví iterátor na předchozí prvek
- bool has_prev() – vrací $true$, pokud má uzel předchůdce ⇔ iterátor je na prvním prvku, jinak vrací $false$
- bool has_next() – vrací $true$, pokud je má uzel následníka ⇔ iterátor je na posledním prvku, jinak vrací $false$
- node_type* get_current() – vrací ukazatel na uzel, na kterém je iterátor
- void reset() – resetuje iterátor do výchozího nastavení (na první uzel dle typu iterátoru)
- node_type* operator->() – vrací ukazatel na uzel, na kterém je iterátor
- node_type& operator*() – vrací dereferencovaný ukazatel na uzel, na kterém je iterátor
- operátory porovnání != a == – porovnávají iterátory, iterátory jsou si rovné, pokud se shodují jejich ukazatele na uzly

Pro odvozené třídy preorder_iterator, postorder_iterator, child_iterator, leaf_iterator je rozhraní rozšířeno o další metody.

- iterator& operator++() – inkrementuje iterátor
- iterator& operator-() – dekrementuje iterátor
- iterator operator+(int n) – vrací n-krát inkrementovaný iterátor
- iterator operator-(int n) – vrací n-krát dekrementovaný iterátor

2.2.1 Implementace iterátorů pro stromovou strukturu

Iterátory jsou implementované pomocí mapy M vnitřních uzlů. M obsahuje klíče $\{S_0, S_1, ..., S_k\} \in T$, pro které platí $|P_{S_i}| \neq 0$, tj. jedná se o mapu vnitřních uzlů T. Hodnota $M[S_i] \in \mathbb{N}$ reprezentuje podstrom s kořenem v $M[S_i]$-tém potomkovi S_i, ve kterém se nachází iterátor. Pro iterátor, který se nachází v V_5, je nastavení M: $M[V_0] = 1$, $M[V_1] = 2$. Čítače vnitřních uzlů jsou inkrementovány metodou *next* a dekrementovány metodou *prev*. Pokud je $M[S_i] = 0$, je klíč S_i odstraněn z M, protože S_i není vnitřní uzel.

Přidávání uzlů do T nezneplatní data iterátoru. Při odstranění uzlu z T jsou data iterátoru upravena tak, aby zůstala konzistentní se změnami ve struktuře stromu.

2.2.2 Preorder průchod

Preorder průchod je způsob průchodu stromovou strukturou, kdy je navštíven nejdříve uzel a poté jeho potomci. Posloupnost uzlů v preorder pořadí je pro strom 1.1 $(v_0, v_1, v_4, v_5, v_6, v_7, v_9, v_8, v_2, v_3)$.

2.2.2.1 Metoda next

Nechť je $p \in T$ uzlem, na kterém je iterátor, M mapa vnitřních uzlů T reprezentující pozici iterátoru v T, a P_p uspořádaná n-tice potomků uzlu p. Algoritmus 2 můžeme rozdělit do dvou částí.

1. pokud $|P_p| = 0 \vee M[p] = |P_p|$, pak hledáme následující uzel rekurzivním voláním metody *next* pro rodiče, protože p je list anebo jsme navštívili všechny potomky p, aktualizujeme p, pokud volání metody vrátí *true*

2. jinak $p \notin M \vee M[p] \neq |P_p|$, přejdeme do potomka uzlu p, inkrementujeme čítač $M[p]$, vracíme *true*

Příkladem je výpočet následujícího uzlu pro uzel V_4 z obrázku 1.1. Nejprve použijeme první pravidlo, protože V_1 je listem, použijeme metodu *next* na rodiče V_4, tj. uzel V_1. Poté použijeme druhé pravidlo, neboť $M[V_1] = 1$, $|P_{V_1}| = 3$, a tedy $M[V_1] \neq |P_{V_1}|$, přecházíme do potomka uzlu V_1 na pozici $M[V_1]$, tj. do uzlu V_5, a inkrementujeme čítač $M[V_1]$ na hodnotu 2.

Algorithm 2 Metoda next pro preorder iterátor

```
function NEXT(reference na uzel p)
    if P_p = 0 ∨ M[p] = |P_p| then
        if p není kořen T then
            u ← PARENT(p)
            if NEXT(u) then
                p ← u
                return true
            end if
        end if
        return false
    end if
    if p ∈ M then
        i ← M[p]
    else
        i ← 0
    end if
    c ← potomek p_{M[p]} uzlu u
    M[p] ← M[p] + 1
    p ← c
    return true
end function
```

2.2.2.2 Metoda prev

Nechť $p \in T$ je uzel, na kterém je iterátor, r rodič p, M mapa reprezentující pozici iterátoru v T. Algoritmus můžeme rozdělit do tří částí.

1. p je kořen stromu, zůstáváme v p
2. $M[r] = 1$, tj. p je první potomek r, přejdeme do r, dekrementujeme $M[r]$, odstraníme r z M, protože r přestává být vnitřním uzlem
3. $M[r] > 1$, přecházíme do uzlu nejvíce v pravo v podstromu s kořenem v $M[r] - 1$-tém potomkovi uzlu r, dekrementujeme $M[r]$

Pro výpočet předchůdce uzlu V_8 použijeme druhé pravidlo, voláním metody `most_right_node` přejdeme do uzlu nejvíce vpravo podstromu s kořenem v V_7, tj. V_9. Předchůdce pro V_9 je podle prvního pravidla V_7.

Iterátor je odolný proti změnám ve struktuře T. Mějme posloupnost uzlů v preorder pořadí $(V_0, V_1, V_2, ..., V_i, ..., V_n)$ a iterátor I nastavený na V_i. Pokud vložíme uzel V_j do T tak, že $j > i$, V_j bude v pořádku navštíven. Pokud $j < i$, potom se uzel V_j nachází již v navštívené části stromu a I nemá odpovídající data o struktuře T. Proto musí metoda prev implementovat mechanismus,

Algorithm 3 Metoda prev pro preorder iterátor

```
function PREV(reference na uzel p)
    if p je kořen T then
        return false
    end if
    r ← PARENT(p)
    if M[r] = 1 then
        M[r] ← M[r] − 1
        p ← r
        return true
    end if
    i ← M[r]
    p ← i − 1-tý potomek uzlu r
    M[r] ← M[r] − 1
    p ← MOST_RIGHT_NODE(p)
    return true
end function
```

Algorithm 4 Metoda most_right_node pro postorder iterátor

```
function MOVE_RIGHT_NODE(reference na uzel p)
    if |P_p| ≠ 0 then
        p ← poslední potomek uzlu p
        MOST_LEFT_NODE(p)
    end if
end function
```

který aktualizuje data I. Pro vnitřní uzel je v metodě `most_right_node` kontrolována hodnota čítače $M[S]$ proti $|P_S|$. Pokud $M[S] < |P_S|$, pak byl přidán uzel k S, čítač $M[S]$ je aktualizován na $|P_S|$.

2.2.3 Postorder průchod

Postorder iterátor je způsob průchodu stromovou strukturou, kdy jsou nejdříve navštíveni potomci uzlu a až poté samotný uzel. Pořadí pro strom z obrázku 1.1 vypadá následovně $V_4, V_5, V_9, V_7, V_8, V_6, V_1, V_2, V_3, V_0$.

2.2.3.1 Metoda next

Nechť je iterátor na uzlu p s rodičem r.

- pokud nejsme v uzlu V_0, protože V_0 je posledním uzlem posloupnosti
    - pokud $M[r] = |P_r|$, přecházíme do r

  - jinak přecházíme do uzlu nejvíce vlevo uzlu podstromu s kořenem v $i+1$ potomkovi uzlu r voláním rekurzivní funkce `most_left_node` na p_{i+1} v r, inkrementujeme čítač pro $M[r]$

Algorithm 5 Metoda next pro postorder iterátor

```
function NEXT(reference na uzel p)
    if p není kořen T then
        r ← PARENT(p)
        if M[r] = |P_r| then
            p = r
            return true
        end if
    else
        c ← potomek p_{M[r]} uzlu r
        M[r] ← M[r] + 1
        p ← c
        MOST_LEFT_NODE(p)
        return true
    end if
    return false
end function
```

Algorithm 6 Metoda most_left_node pro postorder iterátor

```
function MOST_MOST_LEFT(reference na uzel p)
    if |P_p| ≠ 0 then
        c ← první potomek uzlu p
        M[p] = 1
        p ← c
        MOST_LEFT_NODE(p)
    end if
end function
```

2.2.3.2 Metoda prev

1. pokud $|P_p| > 0 \wedge M[p] = |P_p|$, přecházíme do posledního potomka p, vracíme *true*
2. jinak voláme metodu `upward` na p
   a) pokud $M[p] = 1$, procházíme stromem nahoru směrem ke kořeni rekurzivním voláním `upward` pro rodiče p
   b) přejdeme do $p_{M[p]-1}$

Algorithm 7 Metoda prev pro postorder iterátor

```
function PREV(reference na uzel p)
    if |P_p = 0| ∧ M[p] = |P_p| then
        p ← potomek p_{M[p]} uzlu p
        return true
    end if
    if UPWARD(p) ≠ 0 then
        for každý uzel c v podstromu s kořenem p do
            M[c] ← M[c] − 1
        end for
        return true
    end if
    return false
end function
```

Algorithm 8 Metoda upward pro postorder iterátor

```
function UPWARD(reference na uzel p)
    r ← PARENT(p)
    if M[u] = 1 then
        if UPWARD(r) then
            p ← u
            return true
        end if
        return false
    end if
    p ← potomek p_{M[r]−2} uzlu r
    M[r] ← M[r] − 1
    return true
end function
```

Voláním metody *prev* na iterátoru na V_1 z 1.1 dostáváme první popsaný případ, iterátor přejde na uzel v_6. Dalším předchůdcem je uzel v_8, rovněž první případ a dále uzel v_7, kdy použijeme druhou popsanou situaci.

Podobně jako preorder iterátor musí postorder iterátor implementovat mechanismus opravy pokud došlo k vložení uzlu do části stromové struktury, která byla již navštívena. Mechanismus pracuje analogicky k preorder iterátoru.

2.2.4 Iterátor přes potomky

Iterátor prochází potomky uzlu. Průchodem pro kořen stromu V_0 získáme posloupnost (v_1, v_2, v_3). Iterátor se nachází na pozici $i \in [0, |P_r| - 1]$ pro uzel r, pro který procházíme potomky. Implementace metod pro přechod na následu-

jící a předchozí prvek je přímočará. Přejít do následujícího uzlu na pozici $i+1$ lze tehdy pokud $i < |P_r| - 1$. Do předchozího prvku $i - 1$ tehdy pokud $i > 0$.

2.2.5 Iterátor přes listy

Iterátor prochází uzly S, které jsou listy, tj. platí $|P_S| = 0$. Průchodem stromem z obrázku 1.1 dostáváme posloupnost $v_4, v_5, v_9, v_8, v_2, v_3$. Implementace je založena na preorder iterátoru. Uzly, které nejsou listy, jsou iterátorem přeskočeny.

2.3 Odstranění uzlu pomocí iterátoru

Odstranění uzlu ze stromové struktury implementuje metoda erase mapovací třídy. Metoda zajišťuje odstranění uzlu ze stromové struktury a aktualizaci dat iterátoru v souladu se změnami ve struktuře stromu. Díky tomu nemusí iterátor znovu načítat aktuální strukturu stromu, což je výhodné pro velké stromové struktury. Uzel je odstraněn a iterátor je posunut na jiný uzel podle typu iterátoru.

- preorder iterátor – po smazání prvku je iterátor posunut na předchozí prvek, protože předchozím prvekem je buď levý sourozenec nebo rodič mazaného uzlu, příkladem může být odstranění uzlu v_4 ze stromové struktury 1.1, který je prvním potomkem uzlu v_1. Stav iterátoru je $M[v_0] = 1$, $M[v_1] = 1$. Iterátor je nastaven na předchozí prvek v_1, čítač $M[v_1]$ je dekrementován a protože po dekrementaci je čítač $M[v_1] = 0$, v_1 odstraněn z M.

- postorder iterátor – iterátor je posunut na následující uzel, protože následníkem je buď pravý sourozenec, nebo rodič mazaného uzlu, pro mazaný uzel máme v iterátoru informace o struktuře jeho podstromu, vnitřní uzly tohoto podstromu musí být smazány z M, odstraníme-li z 1.1 uzel v_6, I bude nastaven na následující uzel v_1, vnitřními uzly podstromu s kořenem v uzlu v_6, tj. v_6, v_7, v_8 jsou odstraněny z M.

- child iterátor – je přesunut na předchozí uzel

2.4 Rozhraní pro průchod strukturou

Rozhraní pro průchod stromovou strukturou jsou třídy šablonované na typ iterátoru, který má být vracen metodami pro příslušný typ průchodu. Rozhraní implementuje sadu metod, které vracejí příslušné iterátory. Chování metod je pro všechna rozhraní analogické. Struktury, které mají umožňovat průchod strukturou v preorder pořadí, dědí z preorder rozhraní. Analogicky pro jiné typy průchodů.

- `begin()` – na neprázdné struktuře vrací iterátor ukazující na první uzel v preorder pořadí, jinak vrací end iterátor
- `begin(int id)` – argumentem metody je identifikátor uzlu, na který se má nastavit iterátor, pokud $S_{id} \notin T$ vrací end iterátor
- `end()` – vrací prazdný (nevalidní) iterátor, který ukazuje za poslední prvek, jeho dereference může způsobit pád programu
- `back()` – vrací iterátor ukazující na poslední prvek T v preorder pořadí
- `preorder_iterable` – definuje defaultní metody `begin`, `end`, `find`, `back`, dále implementuje metody prefixované prefixem `preorder_`, které vrací rovněž preorder iterátor
- `postorder_iterable` – definuje metody s prefixem `postorder_`, které vrací postorder iterátor
- `child_iterable` – definuje metody s prefixem `child_`, které vrací iterátor procházející pouze potomky zadaného uzlu
- `leaf_iterable` – definuje metody s prefixem `leaf_`, které vrací iterátor procházející listy

2.5 Implementované struktury

Obsahem této kapitoly je popis implementovaných struktur. Prvním podkapitola pojednává o implementaci stromové struktury, která obsahuje v uzlech data. Dále ji budeme označovat jako *tree*. Další podkapitola obsahuje popis implementace stromové struktry *trie*, která má data na hranách a umožňuje rychlý výběr potomka.

2.5.1 Tree

Tree stromová datová struktura, která obsahuje data v uzlech. Dostupné implementace jsou `array_tree` a `linked_tree` lišící se prefixem, který určuje způsob mapování do stromové struktury. Data jsou součástí uzlu `data_node_t`, který implementuje metody pro získání dat u uzlu a pro nastavení dat do uzlu. Data je možné integrovat do struktury také mimo uzly, například pomocí stl kontejneru `std::vector`, ve kterém budou data pro S_i dostupná pod indexem i.

Implementace tree

- `array_tree` je tree mapovaný do pole, používá třídu `tree_array_mapper` s uzlem `tree_array_node_t`.

Listing 2.2: Definice třídy tree uzlu mapovaného do pole.

```
template <class data_type> class tree_array_node_t
: public data_node_t<data_type>, public array_node_t {...}
```

- `linked_tree` je struktura tree se spojovým mapování, používá třídu `tree_linked_mapper` s uzlem `tree_linked_node`.

Listing 2.3: Definice třídy tree uzlu mapovaného spojově.

```
template <class data_type> class tree_linked_node_t
: public data_node_t<data_type>, public linked_node_t {...}
```

2.5.1.1 Přidávání a odebírání uzlů

Struktura má vlastní typ iterátoru, který implementuje metodu `insert` pro přidání uzlu. Jejím argumentem jsou data, která mají být umístěna do přidaného uzlu. Odebrat uzel je možné metodou erase jejíž argumentem je iterátor. Protože odstranění uzlu ze stromové struktury zajišťuje mapovací třída, implementace metod pro mazání obsahují pouze volání analogických metod pro odstranění uzlu v mapovací třídě. Jejich implementaci metody jsme již popsali výše.

- `bool erase(preorder_iterator& it);`
- `bool erase(postorder_iterator& it);`
- `bool erase(child_iterator& it);`

Tree má definované `tree_iterable` rozhraní, které dědí z `preorder_iterable`, `postorder_iterable`, `leaf_iterable`, `child_iterable`. To znamená, že je možné strukturu procházet v pořadí preorder, postorder, po listech, nebo po potomcích uzlu.

2.5.1.2 Ukázka práce s tree

Listing 2.4: C++ program, který ukazuje použití tříd implementujících tree.

```
#include "array_tree.hpp"
#include <iostream>

int main() {
    array_tree<char> tree;

    // nastavime koren stromu
    tree.alloc_root('r');

    // ziskani preorder iteratoru na koren stromu
    array_tree<char>::iterator it = tree.begin();

    // pridani potomku ke koreni
    for (int i = 0; i < 10; ++i) {
        it.insert('c');
    }

    // pruchod postorder iteratorem
    for (auto it3 = tree.postorder.begin();
            it3 != tree.postorder_end(); ++it3) {
        std::cout<<"id: "<<it->get_id()<<", data: "
            <<it->get_data()<<std::endl;
    }

    // ziskani iterator pres potomky korene
    array_tree<char>::child_iterator it4 = tree.child_begin(0);

    // smazani prvniho potomka korene pomoci child iteratoru
    tree.erase(it4);

    // vypis potomku
    while (it4) {
        std::cout<<"id: "<<it->get_id()<<", data: "
            <<it->get_data()<<std::endl;
        it4.next();
    }
}
```

2.5.2 Trie

Trie je datová struktura, která obsahuje data na hranách. Data hran v tomto případě můžeme chápat jako klíče, pomocí kterých lze vyhledávat mezi potomky. Vztah mezi rodičem r a potomkem p je definován přechodovou funkcí $\delta(r, k) = p$, kde k reprezentuje klíč, kterým se dostaneme z r do p.

Třída `trie_node` obsahuje mapu, ve které lze podle klíče vyhledávat identifikátor uzlu p. Slouží pro rychlé vyhledávání mezi potomky uzlu r podle klíče.

Implementace trie

- `array_trie` je trie mapovaný do pole, používá `tree_array_mapper` s uzlem `trie_array_node`.

  Listing 2.5: Definice třídy trie uzlu mapovaného do pole.

  ```
  template <class key_type> class trie_array_node_t
  : public trie_node<key_type>, public array_node_t {...}
  ```

- `linked_trie` je trie mapovaný do pole, používá `tree_linked_mapper` s uzlem `trie_linked_node`.

  Listing 2.6: Definice třídy trie uzlu mapovaného spojově.

  ```
  template <class key_type> class trie_linked_node_t
  : public trie_node<key_type>, public linked_node_t {...}
  ```

2.5.2.1 Přidávání a odebírání uzlů

Trie má rovněž vlastní typ iterátoru, který implementuje metodu `insert`. Argumentem je klíč, který má být umístěn na hraně spojující rodiče a přidávaný uzel. Pro tento typ iterátoru je šablonována třída `trie_iterable`. Trie lze procházet v pořadí preorder, nebo postorder. K dispozici je také iterátor přes potomky zadaného uzlu. Pro odebrání uzlu jsou implementované stejné metody jako u tree. Argumentem může být preorder, postorder nebo child iterátor. Před odstraněním uzlu z mapovací třídy je z rodiče odebrána hrana na mazaný uzel.

2.5.2.2 Ukázka práce s trie

Listing 2.7: C++ kód, který ukazuje použití trie se spojovým mapováním.

```
#include "linked_trie.hpp"

int main() {

    linked_trie<char> trie;

    // pridani potomku ke koreni stromu
    trie.begin().insert('a').insert('b').insert('c');

    linked_trie<char>::iterator it = trie.begin();

    uint32_t child_id;
    // efektivni implementace hledani
    if (it->get_child_by_key('a', child_id)) {
        // do child_id byl nastaven identifikator uzlu
        // pro hranu s klicem 'a'
    }
    else {
        // klic neexistuje
    }

    // pruchod strukturou je totozne a mazani uzlu
    // je totozne s tree
    for (auto it = trie.begin(); it != trie.end(); ++it) {
        std::cout<<"id: "<<it->get_id()<<std::endl;
    }

    // vymazani uzlu
    trie.erase(it);
}
```

Kapitola 3

Implementace algoritmů

3.1 Implementace algoritmu Aho Corasick

Algoritmy jsou implementovány pomocí připravených nebo drobně modifikovaných stromových struktur. Pro reprezentaci výsledků hledání obsahuje práce implementaci pomocných tříd `kw_matches`, která reprezentuje všechny výskyty hledaného vzoru, a `all_kw_matches`, která sdružuje `kw_matches` všech hledaných vzorů, tj. obsahuje výskyty všech hledaných vzorů. Pro reprezentaci hledaného vzoru je třída `keyword_t`. Jedná se v podstatě o šablonovaný `std::vector`, který sdružuje posloupnost klíčů. Hledané vzory můžeme přidat do třídy `dict_t`, která implementuje jednoduchý slovník.

Vyhledávací automat používá stromovou strukturu trie, ve které je možné efektivně hledat hodnotu dopředné funkce. Obsahuje vlastní definici uzlu doplněnou o množinu identifikátorů vzorů, které mají být v daném stavu detekovány, a hodnotu zpětné funkce. Automat lze postavit nad množinou hledaných vzorů P, které třídě předáme jako slovník. Rozhraní automatu implementuje dvě důležité metody. Metodu pro sestavení vyhledávacího automatu a metodu search pro vyhledávání vzorů ve vstupním textu T. Vstupním argumentem metody search je vstupní text, ve kterém má být množina vzorů $P = \{P_1, P_2, ..., P_n\}$ zadaná slovníkem detekována. Metoda vrací nalezené výskyty v třídě `all_kw_matches`, ve které jsou výskyty agregováné podle identikátoru P_i.

- `bool build()` – metoda postaví ze vzorů ve slovníku vyhledávací automat
- `all_kws_matches* search(const keyword_t<key_type>& text)` – pro zadaný vstupní text jsou detekovány všechny hledané vzory ze slovníku

Listing 3.1: Definice třídy stavu AC automatu v C++.

```
template <class key_type>
class ac_node_t : public trie_array_node_t<key_type> {
private:
    uint32_t fail_function = 0;
    std::set<uint32_t> ending_kws;
public:
    ac_node_t(uint32_t id, tree_mapper* mapper)
        : trie_array_node_t<key_type>(id, mapper) {}
};
```

Listing 3.2: Použití implementace algoritmu Aho Corasick.

```
// slovnik hledanych vzoru
dict_t<char> dict;

keyword_string_t kw("abc");

// pridani hledaneho vzoru do slovniku
dict.add_keyword(kw);

// vstupni text
keyword_string_t text(ddddabcabc");

ac_machine<char> ac(dict);
// konstrukce vyhledavaciho automatu
ac.build();

// hledani
std::unique_ptr<all_kws_matches> matches(ac.search(text));

// zobrazeni vysledku
matches->display();
```

3.2 Implementace algoritmu Position heap

Implementace využívá mapování stromové struktury do pole. K dispozici je naivní implementace konstrukce Position heap a efektivní implementace vycházející ze zdrojových kódů [**?**]. Efektivní implementace zvládne vytvořit indexy pro vstupní řetězec jedním průchodem, tj. $\mathcal{O}(n)$ pro text délky n. Časově náročný výběr rodiče s nejdelším společným prefixem je optimalizován pomocí přidané struktury dual heap $D(T)$ [**?**]

Naivní implementace konstrukce Position heap je triviální. Přípony T jsou postupně přidávany do struktury způsobem popsaným v první kapitole. Me-

todou `get_child` ověřujeme, zda je možné využít existující strukturu. Nový uzel vytvoříme metodou mapovací třídy.

Listing 3.3: Naivní algoritmus konstrukce Position heap v C++.

```
for (const auto& keyword : text->get_suffixes()) {
    int parent = 0; // pro kazdy suffix inicializovat na koren
    for (unsigned int i = 0; i < keyword.keyword.size(); ++i) {
        if (!get_child(parent, keyword.keyword[i], child, i)) {
            // vytvoreni indexu v position heap
            t->add_child(parent, child);
            break;
        }
        parent = child;
    }
}
```

Dále třída implementuje tyto metody.

- `all_kws_matches* search(const dict_t<T>& keywords)` – metoda pro hledání vzorů ve slovníku, spouští hledání pro všechny vzory ve slovníku a organizuje přidávání do výsledků hledání
- `kw_matches* search_keyword(const keyword_t<T>& keyword)` – metoda pro hledání jednoho vzoru implemetovaná způsobem popsaným v první kapitole
- `get_child(int parent_id, key_t<T>& key, int matched_child, int depth)` – slouží pro výběr potomka, do kterého se dostaneme klíčem z argumentu metody, potomci uzlu s parent_id jsou procházeni child_iterátorem a je porovnáván klíč na hraně mezi rodičem a potomkem, v případě shody je identifikátor potomka nastaven do matched_child
- `void add_subtree(int parent_id, const keyword_<T>& keyword, kw_matches* matches)` – rekurzivním voláním přidá pozice všech následníků v podstromu s kořenem v parent_id do výsledků hledání
- `void add_uptree(stack<int>& predecessors, keyword_t<T>* keyword, kw_matches* matches)` – testuje předchůdce uzlu, které jsou uloženy v zásobníku a pozice, na kterých byl ověřen výskyt přidá na výsledků hledání, efektivní implementace této metody ověřuje výskyt vzoru při sestupu strukturou a neověřuje již symboly, které byly ověřeny pomocí metody `get_child`.

Listing 3.4: C++ kód ukazující použití implementace Position heap.

```
// vstupni text
keyword_string_t text("ddddabcabc");

position_heap_t<char> heap(text);

// konstrukce position heap
heap.build();

// hledani vzoru
std::unique_ptr<all_kws_matches> matches(heap.search(dict));

// zobrazeni vysledku hledani
matches->display();
```

Kapitola 4

Testování

4.1 Testování mapování a iterátorů

Testy můžeme rozdělit do dvou skupin. První skupinou jsou testy pro mapovací třídy, které testují správné vytvoření stromové struktury, tj. vytvoření správné hierarchie ve struktuře stromu přidáváním a odebíráním uzlů. Vztahy mezi uzly se kontrolují dle způsobu mapování do stromové struktury. V případě mapování do pole musí být správně nastaven odkaz na rodiče, prvního potomka a oba sourozence, v případě spojové reprezentace na rodiče a potomky. Součástí testů jsou i testy na procházení stromových struktur. Tyto testy ověřují průchod stromovou strukturou ve správném pořadí daném typem iterátoru. Dalšími testy jsou testy na odstranění uzlu pomocí iterátoru. Poté, co je uzel ze stromové struktury smazán, je zkontrolováno správné nastavení na předchozí nebo následující uzel podle typu iterátoru.

Testy stromových struktur tree a trie testují správnou integraci dat do stromové struktury. Ve struktuře tree je kontrolováno správné uložení dat do uzlů. Pro trie se testuje správné uložení dat na hrany struktury a vyhledávání potomka podle zadaného klíče.

4.2 Testování algoritmů Aho Corasick a Position heap

Testy algoritmu Aho Corasick a Position heap testují správnou funkčnost vyhledávání vzorů v textu. V první řadě jsou implementovány testy s předpřipravenými daty a dále testovací třídy, které náhodně vybírají zadaný počet vzorů ze vstupního textu a porovnávají výstup hledání obou algoritmů. Test obsahuje měření času potřebného ke konstrukci vyhledávacího automatu Aho Corasick a ke konstrukci Position heap, měření času potřebného pro vyhledání vzorů ve vstupním textu vyhledávacím automatem a měření času vyhledávání vzorů pomocí Position heap. Vyhledávací automat Aho Corasick sestavíme

z hledaných vzorů. Hledáním rozumíme zpracování vstupního textu vyhledávacím automatem. Postup pro algoritmus Position heap je opačný, protože zpracováním rozumíme vytvoření indexů pro vstupní text. Vyhledání pomocí Position heap pracuje pouze s hledanými vzory. Pro velký počet hledaných vzorů může být sestavení vyhledávacího automatu časově náročnější než indexace kratšího textu. Naopak může být výhodnější sestavit vyhledávací automat než indexovat dlouhý text.

Závěr

Cílem bakalářské práce bylo navrhnout a implementovat stromové struktury v programovacím jazyce C++. V první řadě jsem provedl analýzu algoritmu pro vyhledávání řetězců Aho Corasick a algoritmu pro indexaci řetězce Position heap. Zmíněné algoritmy definovaly požadavky na stromové struktury, které jsou navzájem odlišné, ale zároveň dostatečně obecné pro použití v jiných algoritmech.

Na základě provedené analýzy jsem stromové struktury navrhl. Modulární uspořádání návrhu umožňuje zapojení vlastní implementace mapovacích tříd i iterátorů. Solidnost návrhu jsem prověřil implementací odlišných stromových datových struktur s různými požadavky na strukturu uzlu a na nároky na implementaci dat do struktury stromu. Pomocí implementovaných stromových struktur jsem implementoval zmíněné algoritmy, na kterých byla ověřena funkčnost navržených stromových struktur.

Správná funkce stromových struktur a algoritmů byla ověřena testy. U referenčních algoritmů jsem testoval správnou funkci vyhledávání vzorů v textu na předpřipravených datech a také jsem vytvořil testy, které porovnávají výstup hledání obou algoritmů.

Výsledkem této práce jsou implementované stromové datové struktury, které lze použít v připravené podobě, nebo je lze upravit a rozšířit podle konkrétních potřeb. Navržené struktury jsou dostatečně obecné, aby je bylo možné použít v dalších algoritmech a v programech, které vyžadují práci se stromovými strukturami.

Na úplný závěr práce uvádím, že pro mě byla teoretická i praktická část zajímavá. Získal jsem nové poznatky o referenčních algoritmech Aho Corasick a Position heap. Implementací stromových struktur jsem prohloubil své zkušenosti s jazykem C++.

Příloha A

Obsah přiloženého CD

- readme.txt .... stručný popis obsahu CD
- src
  - tree_structures .... implementace stromových struktur
  - preview .... ukázka použití stromových struktur
  - tests .... testy struktur a algoritmů
  - thesis .... zdrojová forma práce ve formátu LaTeX
  - visualization .... zdrojové kódy pro generování obrázků
- text .... text práce
  - thesis.pdf .... text práce ve formátu PDF