Theorie Algoritmen II - 2013/2014

Andreas De Lille

Augustus 2014

Inhoudsopgave

Deel I

Gegevens Structuren II

Deel II

Grafen II

Deel III

Strings

Hoofdstuk 1

Gegevensstructuren voor strings

1.1 Inleiding

- Sleutels nu niet rangschikken op de afzonderlijke sleutelelementen (in hun geheel), maar op de elementen van de sleutel een voor een. Dit wordt Radix Search genoemd.
- toegangstijd bij radix search is competief met die van hashing en binaire zoekbomen.
- in het slechte geval blijft deze toegangstijd meestal goed, zonder de complexiteit van evenwichtige zoekbomen te gebruiken.
- Ze kunnen wel veel geheugen innemen
- de individuele elementen van de sleutel moeten vlot toegankelijk zijn.

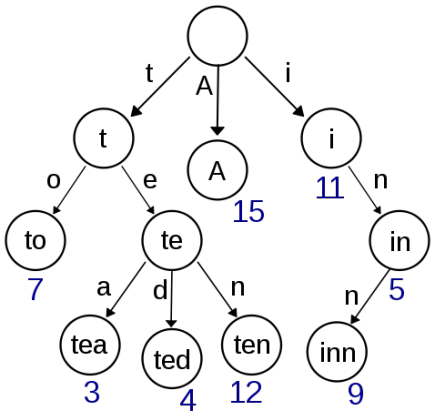
Hieronder worden een aantal mogelijk implementaties gegeven waarbij steeds verondersteld wordt dat geen enkele sleutel een prefix is van een andere.

1.2 Digitale zoekbomen

- Vergelijkbaar met gewone zoekbomen, de sleutels worden opgeslagen in de knopen, toevoegen & verwijderen gebeurd op dezelfde manier
- ER is een verschil : de juiste deelboom wordt niet bepaald door de zoeksleutel te vergelijken met de sleutel in de knoop, maar enkel door het volgende element van de zoeksleutel. Vergelijken gebeurd van links naar rechts .
- Meestal worden hiervoor de bits gebruikt van de sleutel, al hoeft dat niet zo te zijn.
- In order overlopen van de boom, zal de sleutels NIET altijd in volgorde teruggeven. De sleutels links zijn welliswaar kleiner dan de sleutels rechts en ze hebben dezelfde beginbits. Toch kan de wortel eender waar vallen tussen zijn 2 kinderen.
- Elke sleutel in een digitale zoekboom bevindt zich ergens op de weg vanuit de wortel bepaald door de opeenvolgende bits van deze sleutel. De langste weg in de boom is hierdoor beperkt door het aantal bits van de langste opgeslagen sleutel (en dus ook de hoogte).
- Deze bomen zeer performant voor een groot aantal sleutels met een kleine bitlengte
- Als we veronderstellen dat de kans op een nul of een bit steeds 50% is waardoor de boom mooi verdeeld zal zijn, dan is de gemiddelde weg O(lg(n)). Het aantal vergelijking is trouwens nooit meer dan het aantal bits van de zoeksleutel.

Digitale zoekbomen: net als gewone zoekbomen, maar we kijken naar de opeenvolgende bits ipv de hele sleutel. De performantie is vergelijkbaar met de performantie van rood zwarte bomen, terwijl hun implementatie stukken eenvoudiger is. Het enige beperkende minpunt is dat we efficiënte toegang moeten hebben tot de bits van de sleutels en dat er enkel woordenboekoperaties mogelijk zijn.

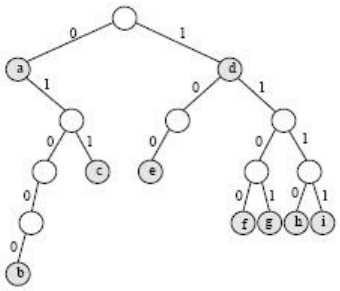
1.3 Tries



Figuur 1.1

Een tie is een digitale zoekstructuur die wel de volgorde van de opgeslagen sleutels behoudt. Er zijn verschillende soorten:

1.3.1 Binaire tries



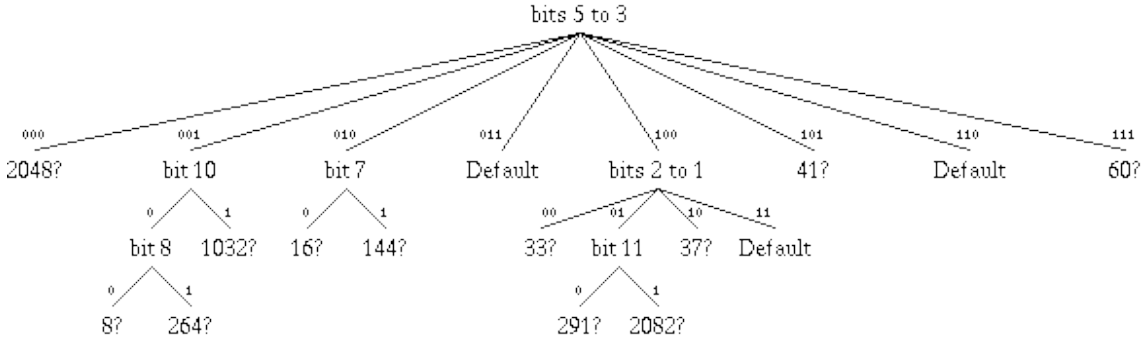
Figuur 1.2

- zoekweg wordt volledig bepaald door de opeenvolgende bits van de zoeksleutel.
- de sleutels worden hier enkel in de bladeren opgeslagen, waardoor een inorder doorloop wel de sleutels in de juiste volgorde teruggeeft.

- de zoeksleutel moet niet meer vergeleken worden met een sleutel in elke knoop op e zoekweg. Zoeken en toevoegen gebruiken dus enkel de opeenvolgende bits van de zoeksleutel om de juiste weg te volgen.
  - Eindigen we in een ledige boom → sleutel niet in de boom, eventueel toevoegen
  - Eindigen we in een blad → Dit blad is de enige mogelijk kandidaat om de sleutel te zijn. Een sleutelvergelijking geeft uitsluitsel. Indien het de sleutel niet is en we hem willen toevoegen dan zijn er verschillende gevallen:
    1. indien het eerst volgende bit verschilt, kunnen we het blad eenvoudig weg vervangen door een knoop en vervolgens het originele blad en de nieuwe sleutel kinderen maken van die knoop.
    2. Indien er nog bits gelijk zijn, moet het blad vervangen worden door een reeks van opeenvolgende knopen. Het vervangen moet dus herhaald worden zolang de bits gelijk zijn. Bij de eerste verschillende bit krijgt de laatste knoop steeds 2 kinderen.
- We zien de nadelen van deze gegevensstructuur: indien de sleutels veel gelijke beginbits hebben, zal de boom slechter verdeeld zijn, we krijgen dus veel knopen met maar 1 kind.
- Een trie met n gelijkmatig verdeelde sleutels heeft gemiddeld $n/\ln 2 \approx 1.44n$ inwendige knopen.
- Verder moeten we ook opmerken dat bits van een sleutel in zijn geheel geen prefix mogen vormen van een langere sleutel; Als dit gebeurd dan zijn er geen bits meer over om een onderscheid te vormen.
- De structuur van een trie is onafhankelijk van de volgorde waarin de sleutels toegevoegd worden. Voor elke verzameling sleutels is er dus slechts een trie mogelijk.
- een trie maakt geen gebruik van sleutelvergelijkingen tijdens het afdalen, enkel de opeenvolgende bits worden getest. Eenmaal afgedaald volgt er eventueel een sleutelvergelijking. Dit is interessant voor als de sleutels bijvoorbeeld lange strings zijn.
- Een trie opgebouwd uit n gelijkmatig verdeelde sleutels zal nooit meer dan $O(\ln(n))$ bitoperaties nodig hebben om zoeken of toevoegen uit te voeren. Dit aantal is nog eens beperkt door de bitlengte van de langste sleutel. De bovengrens van de hoogte wordt dan ook gevormd door deze lengte.

Binaire tries: Zelfde als digitale zoekbomen, maar slaan nu alle sleutels op in de bladeren, waardoor de inorder doorloop wel klopt. Werken door het vergelijken van opeenvolgende bits → een sleutel mag nooit een prefix zijn van een andere.

1.3.2 Meerwegs tries



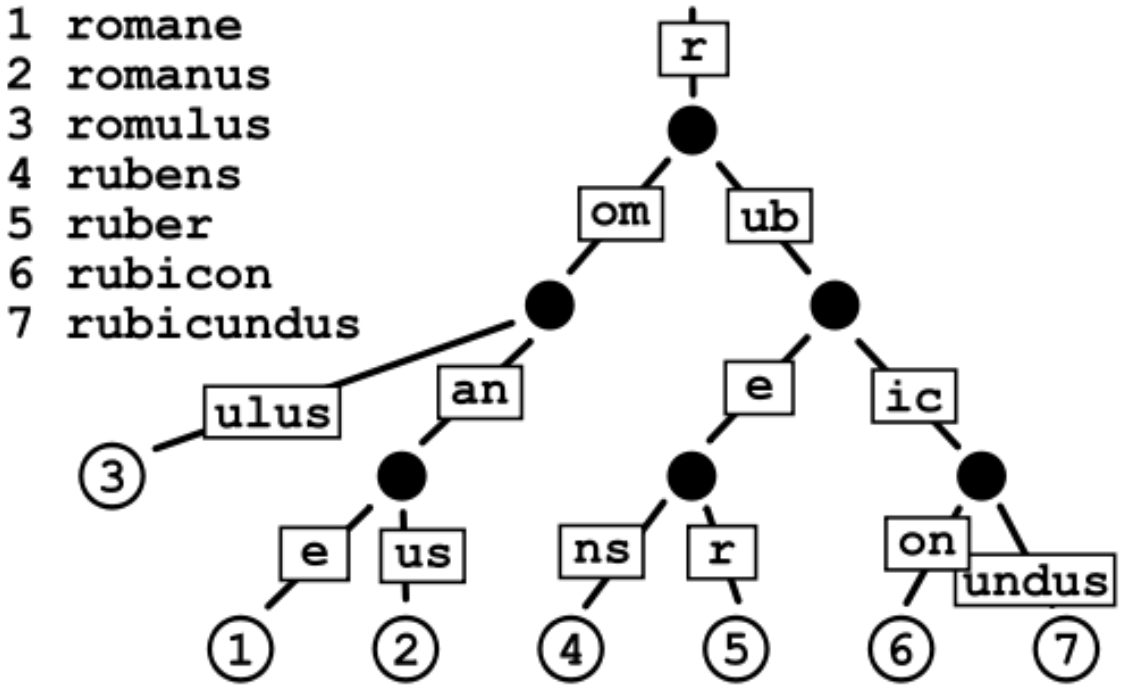
Figuur 1.3

- meer dan 1 bit tegelijk vergelijken → meer dan 2 kinderen per boom.
- zoeken en toevoegen werkt analoog aan een binaire trie, in elke knoop op de zoekweg moet nu een m-wegbeslissing genomen worden, op grond van het volgende sleutelelement. Om deze operatie O(1) te houden, kan men de wijzers naar de kinderen opslaan in de tabel en deze indexeren met het sleutel element.
- toevoegen van een sleutel kan ook resulteren in de aanmaak van een opeenvolgende reeks van inwendige knopen met maar 1 kind.
- Aangezien de kinderen van de knoop geordend zijn volgens de sleutelelementen, kunnen sleutels opnieuw in algabetische volgorde uit de boom opgehaald worden.
- performantie is analoog aan binaire tries. Zoeken of toevoegen vereis gemiddeld O(logm n) testen. Het aantal testen is nooit groter dan de lengte van de langst opgeslagen sleutel.
- Mag de sleutel een prefix zijn van een andere?
  - Is mogelijk als we de informatie in inwendige knopen opslaan, een inwendige knoop vormt immers de prefix van de sleutel.
  - Vervolgens gebruiken we een afsluitkarakter, dat nergens anders kan voorkomen, om een onderscheid te maken tussen de prefix en de inwendige knoop. De inwendige knoop krijgt met andere woorden een extra kind dat overeenkomt met dit afsluit element.

- een nadeel is dat deze trees veel geheugen gebruiken. De knopen dicht bij de wortel hebben meestal veel kinderen, maar dieper in de boom hebben we vaak weinig kinderen. (dit toch plaats vrijhouden voor eventueel toekomstige kinderen). Dit kan opgelost worden:
  - geen tabel te gebruiken, maar een gelinkte lijst om de kindwijzers bij te houden. Dit is iets trager, maar bespaard wel geheugen. Elke knoop in deze lijst heeft een sleutelelement. De elementen zijn gesorteerd.
  - Gebruik maken van verschillende mechanismen: zo kan men bovenaan in de boom wel tabellen gebruiken, in het midden hashmaps en beneden gelinkte lijsten.
  - een andere mogelijkheid is de volledige trie enkel voor de eerste niveaus te gebruiken en dan in de lagere lagen over te schakelen op een andere gegevensstructuur.

Meerwegs tries: Zelfde als binaire tries, maar nu met meerdere bits tegelijk, wat zorgt voor meerdere kinderen. Doordat het aantal kinderen nu meer kan zijn dan 2, kunnen we eventueel ook prefixen van sleutels toevoegen.

1.3.3 Patricia tries



Figuur 1.4

- Veel trie knopen hebben maar 1 kind → geheugen verlies
- 2 soorten knopen
  1. Inwendige knopen met kindwijzers
  2. bladeren met sleutel, zonder kinderen.
- Practical Algorithm To Retrieve Information Coded In alphanummeric lost deze problemen op door:
  - enkel knopen met meer dan een kind toegelaten. In deze knoop slaat men de index van het daar te testen sleutel element op.
  - Gebruik maken van een soort knoop. Sleutels worden wel opgeslagen in inwendige knopen. Bladeren worden nu vervangen door wijzers naar deze knopen; deze wijzen dus terug omhoog in de boom.

- We behandelen enkel binaire tries, de knopen bevatten dan 1 sleutel en 2 wijzers. Of de wijzers inwendige knopen of kinderen bevatten wordt afgeleid uit de indices bij de betrokken knopen. Al kan hiervoor ook een extra logische waarde voorzien worden. Voor n sleutels zijn er dus n knopen en bijgevolg 2n wijzers waarvan er n naar bladeren wijzen en n-1 naar inwendige knopen. (een wijzer wordt dus niet gebruikt)

- patricia tries negeren sleutels in de knopen op de zoekweg en test enkel de aangeduide zoeksleutelbit. Op het einde volgt een sleutelvergelijking wanneer een wijzer naar een hogere gelegen knoop refereert (en we dus in een blad komen).

- elke sleutel ligt opgeslagen op zijn eigen zoekweg en zowel als interne knoop en als blad gevonden wordt.

- toevoegen begint met zoeken. Eenmaal we de plaats gevonden hebben zoeken we de meest linkse bit waarin beide sleutels verschillen en zoeken de meest linkse bit waarin beide sleutels verschillen, en zoeken de sleutel een tweede maal in de boom, waarbij die bitpositie vergeleken wordt met de bitposities (indices) bij de knopen op de zoekweg: (we zoeken of de sleutel niet ergens anders hoger in de boom kan staan)

    1. Komen we nu uit in een hogere met een hogere bitpositie dan de onderscheidende bit, dna moeten we een nieuwe knoop tussenvoegen waar die bit getest wordt. Want bij die knoop zal de trie een knoop met een nulwijzer hebben.
    2. Vinden we geen enkele knoop met een hogere bitpositie dan zou zoeken in de onderlggende trie geïndigd zijn met een blad. We kunnen nu de laatste sleutelwijzer vervangen door een wijzer naar een nieuwe knoop met de nieuwe sleutel, de index van de onderscheidende bit en wijzers naar de originele knoop & de nieuwe knoop

- bemerk dat hierboven slechts een knoop toegevoegd wordt, die de meeste linkse onderscheidende bit tussen de zoeksleutel en de gevonden sleutel aanduidt. Bij een gewone trie was dit niet het geval en werden er soms meerdere knopen tussen gevoegd.

- De structuur hangt hier wel af van de volgorde waarin de sleutels toegevoegd worden

- In een binaire patricia tree met n gelijkmatig verdeelde sleutels vereist zoeken of toevoegen van een willekeurig gemiddeld O(lg(n)) bitvergelijkingen, en nooit meer dan de bitlengte van de zoeksleutel. De hoogte van de boom en dus het maximaal aantal bitvergelijkingen worden opnieuw beperkt door de lengte van de langst opgeslagen sleutel.

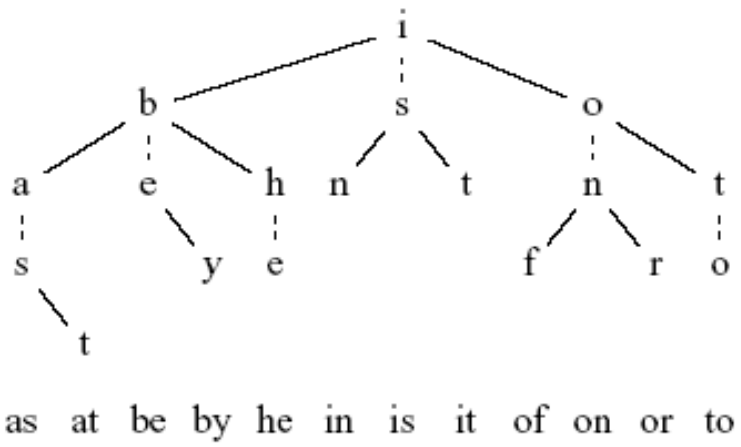
- zoektijd in een gewone trie hangt af van de sleutellengte, patricia tries testen echter meteen de belangrijkste bits, zodat de zoektijd niet toeneemt met de sleutellengte. Ze zijn dus beter geschikte voor lange sleutels.

- Een patricia trie is een alternatieve voorstelling van een gewone trie → dus ook de volgorde van de sleutels blijft bewaard.

Patricia Tries: Combineren de voordelen van digitale zoekbomen en tries;

- ze gebruiken niet meer geheugen dan nodig door de sleutels intern op te slaan (n sleutels = n knopen).
- Ze zijn even efficiënt als tries (gemiddeld O(lg(n)) bittesten + sleutel vergelijking.
- Ze respecteren de volgorde van de sleutels zodat bijkomende operatie mogelijk zijn.

1.4 Ternaire zoekbomen



Figuur 1.5

- een alternatieve voorstelling van een meerwegstrie
- m kindwijzers in een knoop → geheugen verspilling.
- opgelost door een ternaire boom te gebruiken waarvan elke knoop een sleutel element bevat.
- zoeken zal altijd een zoeksleutel met het element in de huidige knoop vergelijken. Indien het element kleiner is gaan we naar links, groter naar rechts. Indien het element gelijk is zoeken we verder in de middenste boom.

- Het element dat we vergelijken wordt niet meer bepaald door de diepte van de boom.
- We gebruiken terug een geschikt afsluitelement.
- een sleutel wort gevonden wanneer we met zijn afsluitelement bij een knoop met datzelfde element terrecht komen. De boom bevat de zoeksleutel niet wanneer we bij een nulwijzer terechtkomen, of wanneer we het einde van de zoeksleutel bereiken vooraleer een afsluitelement in de boom gevonden werd.
- Toevoegen begint met zoeken, gevolgd door het aanmaken van een reeks opeenvolgende knopen voor alle volgende elementen van de zoeksleutel, net zoals bij oorspronkelijke tries.
- tijd nodig is evenredig met de sleutellengte
- het aantal knopen is onafhankelijk van de toevoegvolgorde.
- de volgorde van de opgeslagen sleutels blijft behouden.
- opvragen van een voorloper of opvolger is ook mogelijk.
- Ternaire bomen hebben een aantal voordelen:
  - Ze passen zich goed aan , aan onregelmatig verdeelde zoeksleutels, als blijkt dat de sleutels in de praktijk een gestructureerd formaat hebben bv 26 vrs letters, zal de boom zich hier automatisch naar gedragen.
  - zoeken naar afwezige sleutels is meestal zeer efficiënt, doordat er maar enkele sleutelelementen getest worden.
  - ze laten meer complexere zoekoperaties toe zoals zoeken naar sleutels waarvan bepaalde elementen niet gespecificeerd zijn of het opzoeken van alle sleutels die niet meer dan 1 element verschillen van de zoeksleutel.

- Er zijn ook enkele verbeteringen mogelijk, deze zorgen ervoor de ternaire bomen behoren tot de efficiënste woordenboekstructuren voor strings en ze laten bovendien nog extra operaties toe:
  - Het aantal knopen kan beperkt worden door sleutels in bladeren op te slaan zodra men ze kan onderscheiden en door Inwendige knopen met een kind te elimineren via een sleutelelementindex. Net zoals patricia tries wordt de zoektijd dan onafhankelijk van de zoeklengte.
  - Een andere eenvoudige, maar effectieve verbetering vervangt de wortel door een meerwegstrieknoop, zodat we een tabel van ternaire zoekbomen bekomen. Als het aantal mogelijke sleutelelementen m niet te groot is , kunnen we een tabel van m^2 ternaire zoekbomen gebruiken zodat er een zoekboom overeenkomt met elke eerste paar sleutelelementen.

Ternaire zoekbomen: hebben 3 kinderen: links, rechts en midden. Indien we midden volgen bekomen we verschillende letters die op dezelfde plaats staan! Ternaire bomen met bovenstaande optimalisatie behoren tot de snelste woordenboekstructuren.

Hoofdstuk 2

Zoeken in strings

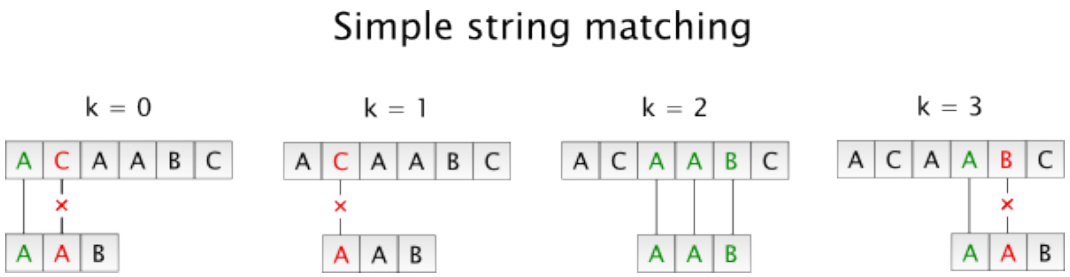
Dit hoofdstuk geeft uitleg over verschillende methodes om strings op te zoeken in een tekst. Hierbij veronderstellen we dat we alle locatie willen vinden waar de string voorkomt. Merk op dat een string met zichzelf kan overlappen.

Daarvoor maken we gebruik van volgende terminologie:

- het patroon P dat we zoeken lengte = m
- tekst T met lengte = n
- alfabet $\sum$ met d karaters

2.1 Variabele tekst

2.1.1 Eenvoudige methode



Figuur 2.1

Deze methode is de eenvoudigste en zoekt gewoon of P vanaf een positie j voorkomt in T door de overeenkomstige karakters (op positie i) te vergelijken (P[i] == T[j+i-1] → i++;). Indien er een verschillend karakter gevonden wordt zullen we de start positie eentje opschuiven (j++;) en opnieuw de karakters van P vergelijken.

Als de strings regelijk random zijn en we een groot alfabet hebben zal de eerste letter vaak verschillen waardoor we meestal na 1 vergelijking al kunnen opschuiven. Het gemiddeld geval is dan ook O(n).

het slechtste geval is O(nm)dit doet zich voor als de eerste O(m) karakters van P op O(n) posities overeenkomen.(bv aaaab zoeken in aaaa....aaaaab) Dit lijkt zeer onwaarschijnlijk, maar als we de technieke gebruiken voor binaire strings of DNA strings met een klein alfabet is dit wel problematisch.

2.1.2 Gebruik van de Prefixfunctie

Prefixfunctie

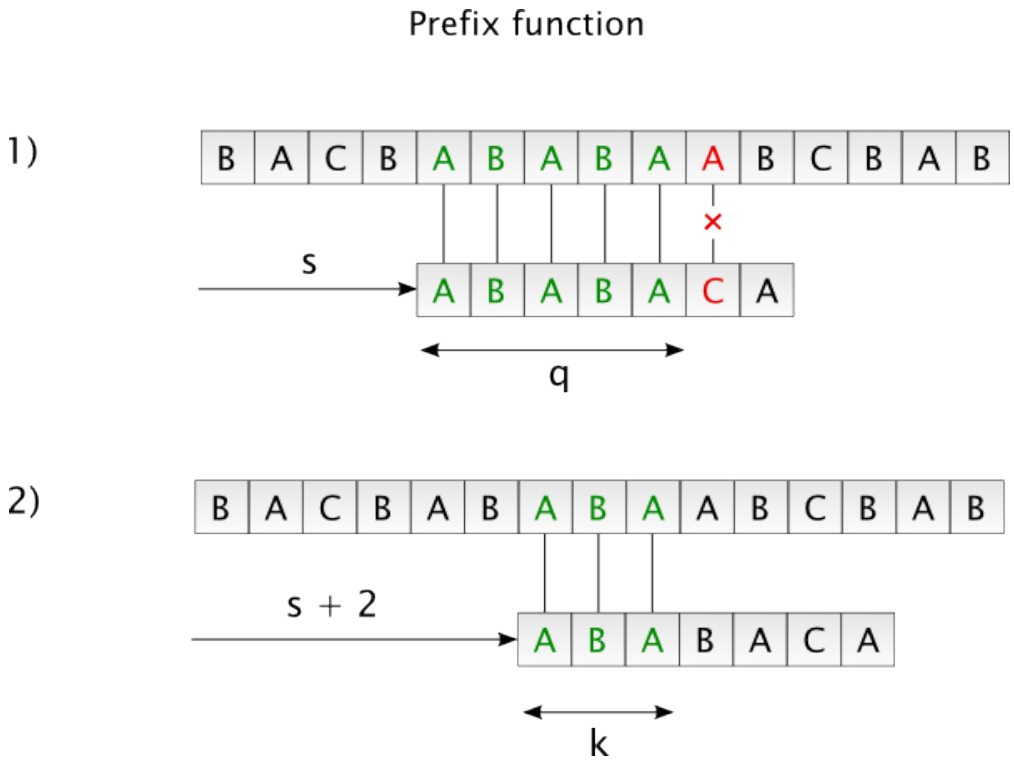
index	1	2	3	4	5	6	7	8	9
str	a	a	b	a	a	b	a	b	b
p()	0	1	0	1	2	3	4	0	0

De prefix functie p() van een string P bepaalt voor elke stringpositie i de lengte van het langste prefix van P, die overeenkomst met een deelstring van P, eindigend bij de positie i (1 ¡= i ¡= m). p(i) is dus steeds kleiner dan i en p(1) = nul.

```
vector<int> PrefixFunction(string S) {
    vector<int> p(S.size());
    int j = 0;
    for (int i = 1; i < (int)S.size(); i++) {
        while (j > 0 && S[j] != S[i])
            j = p[j-1];

        if (S[j] == S[i])
            j++;
        p[i] = j;
    }
    return p;
}
```

Een eenvoudige lineaire methode



Figuur 2.2

Zoals je kan zien op 2.2 zien we dat we het gevonden stuk string kunnen hergebruiken. We hebben namelijk ABABA al gevonden, echter de C klopt niet. Door nu echter de AB door te schuiven vermijden we extra vergelijkingen die toch negatief zouden zijn.

We berekenen m prefix waarden wat $\theta(m)$ tijd inneemt (gemiddeld). Dit betekent dat de buitenste herhaling gemiddeld (geatmortiseerd) O(1) is. Om dat aan te tonen moeten we gebruik maken van een potentiaal functie, de potentiaal komt overeen met de prefixfunctie zelf. Deze vormt zeker een bovengrens voor de uitvoeringstijd, want ze is initieel nul, en wordt nooit negatief. In de buitenste herhaling wordt ze minstens een kleiner (want we schuiven een op (zie 2 in de figuur)).

De afname van de potentiaal in de binnenste herhaling is dus minstens zo groot als het werk in die herhaling. De geatmortiseerde tijd van een buitenste herhaling is haar echte tijd plus de toename in potentiaal en dus 0(1). Aangezien er m-1 buitenste herhalingen zijn, is de totale performantie inderdaad $\theta(n)$.

Knurth-Morris-Pratt

We doen hetzelfde als hierboven maar met wat gepruts kunnen we het aantal binnenste iteraties verder beperken.

1. bepaal de prefix functie

char:	a	b	a	b	a	b	c	a
index:	0	1	2	3	4	5	6	7
p():	0	0	1	2	3	4	0	1

2. We overlopen de string tot we een fout vinden op positie i (in P).
3. i == 1 ? → p eentje naar rechts
4. i > 1 ? → een prefix vinden in P met lengte i-1 die eindigt op positie j-1 in T. Nu moeten we verschuiven volgende de prefix functie
5. als P[p(i-1) +1] == T[j] : na verschuiving komen deze letters overeen en kunnen we dus vanaf hier verder doen
6. als dat niet overeenkomst moeten we P nog wat verder opschuiven tot P.firstChar terug gelijk is aan T[i]
7. tot hier passen we eigenlijk gewoon de lineaire methode toe
8. **als P(i) != T[j] hebben we direct terug een fout;** we hadden dus beter de 2e langste prefix genomen ipv de langste.
    - we hebben dus een bijkomende vereiste voor de prefix functie, namelijk dat het karakter rechts van de prefix verschilt van P[i]. Hierdoor zal de verschuiving groter zijn dan die van de originele methode. Dat het karakter verschilt kan niet garanderen dat het overkomt met T[j] (maar als het hetzelfde is dan is het zowieso fout dus), daarom moeten we eventueel nog een kleinere prefix nemen.
    - aangezien deze vereiste enkel afhankelijk is van P kunnen we ze op voorhand volledig bepalen, dit noemen we p2() of p'().

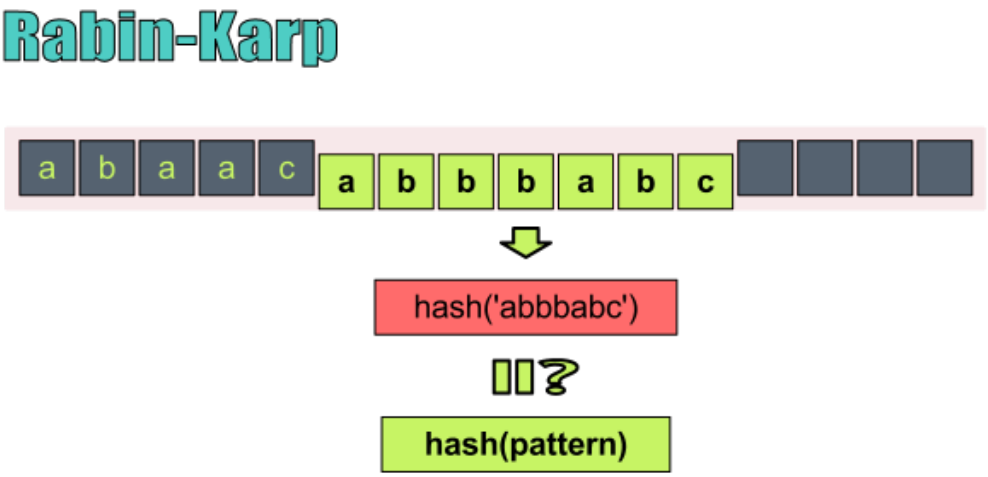
9. **Uiteindelijk gebruiken we nog een foutfunctie f(i)**

   - (i) = p'(i-1) + 1
   - Deze functie geeft voor elke patroonpositie i groter dan een meteen de patroon positie waarmee T[j] moet vergeleken worden.
   - als we ook f(m+1) op die manier definiëren dan geeft die de nieuwe te testen patroon positie na het vinden van P. (Dus als we P vinden hoeveel moeten we dan schuiven om een volgende te vinden (overlap)).
   - Als i == 1 moet P een positie opgeschoven worden en wordt P[1] vergeleken met T[j+1] daarom moet f(1) = 1 zijn en p'(0) = 0 ; p'(1) = 0

het aantal karakter vergelijkingen van dit algoritme is $\theta(n)$. Want na elke verschuiving van P wordt hoogstens een karakter van T getest dat vroeger reeds getest werd. (Als het opnieuw niet overeenkomt wordt er doorgeschoven). Het totaal aantal karaktervergelijking is dus hoogstens gelijk aan de lengte T, plus het aantal verschuivingen. Elke verschuiving gebeurd over minstens 1 positie, zodat het aantal verschuivingen O(n) is. Dit levert dus idd $\theta(n)$.

Aangezien de voorbereiding van de prefix functie, de nieuwe prefixfunctie en de foutfunctie $\theta(m)$ is, wordt de totale performantie $\theta(m + n)$

2.1.3 Karp-Rabin



Figuur 2.3

- Indien getallen vergelijken sneller gaat dan strings, de stings omzetten naar getallen (met hashing) en die vergelijken.
- als we dit doen krijgen we d^m verschillende strings met lengte m, en dus evenveel getallen → te groot → delen door priemgetal → conflicten (zoals bij hashing).
- De getallen moeten in een processor woord passen!
- gelijke strings → gelijke getallen maar gelijke getallen != gelijke strings
- Op lossen door meerdere hashes te gebruiken of een volledige string vergelijking uit te voeren bij gelijke getallen
- Hoe bekomen we deze getallen?
  - waarde deelstring op j+1 halen uit de waarde op j en dat in O(1)
  - Hiervoor beschouwen we een d tallig stelsel, waarbij elk stringelement voorgesteld wordt door een cijfer tussen 0 en d-1
  - H(p) = $\sum_{i=1}^{n}$ dit vereist slechts m optellingen en evenveel vermenigvuldigingen.
  - om beperkte hashwaarde te bekomen, nemen we de rest.
  - doordat de rest van een som gelijk is aan de som van de resten van de termen, mogen we verder rekenen met de resten.
- bewijs p103

- perfomantie: $\theta(n + m)$
- hoe p kiezen?
  - p vast : hierbij nemen we een zo groot mogelijk priemgetal zodat $pd <= 2^w$. Zo groot mogelijk zodat we minder valse positieven hebben, niet groter dan processor woord voor de efficiëntie.
  - p random: een of meerdere random p waarden gebruiken uit een interval.
- Valse positieven?
  - groter priemgetal nemen, maar let op processorwoord lengte
  - Omvormen naar een Las-Vegas algoritme door nog een string vergelijking uit te voeren bij een positieven getallen vergelijking
  - Herbeginnen met een nieuwe random p als dezelfde test fouten genereert.
  - meerdere fingerprints/hashes tegelijk gebruiken
- tweedimensionale patroonherkenning of random vormpjes mogelijk.
- tegelijk naar meerdere strings zoeken door verschillende hashwaarden te zoeken

2.1.4 Shift-And / Shift-Or

- eenvoudig, bitgeoriënteerde methode die zeer efficiënt werkt voor kleine patronen.
- maakt gebruik van dynamisch programmeren
- voor elke positie j in de tekst T worden de prefixen van P bijgehouden die eindigen op positie j. Dit zit in een eendimensionale tabel van m logische waarden.
- het i^{de} element komt overeen met de prefix van lengte i.
- R_j is de waarde die hoort die positie j, dan is R_j[i] waar als de eerste i karakters van p overeenkomen met de i tekstkarakters eindigt in j. Als P eindigt op j en R_j[m] waar is , dan hebben we P gevonden op plaats j-m+1
- T[J+1] kan sommige prefixen verlengen tot aan j+1; R_{j+1} kan dus uit R_j gehaald worden als volgt:
  - R_{j+1}[1] = (P[1] == T[j+1])
  - R_{j+1}[i] = (R_j[i-1] && P[i] == T[j+1]) voor 1 < i <= m
- Performantie: $\theta(nm)$ (wordt hieronder nog verder verbeterd)

- bewerkingen zijn voor elk tabelelement analoog en we kunnen op meerdere bits tegelijk werken in het processorwoord.
- R_{j+1} is afhankelijk van T[j+1] == P[i]
- P is vast dus op voorhand de posities bepalen waarvoor T[j+1] == [i] ; Dit doen we door een tabel S op te stellen met d woorden (met d = grootte van het alfabet).

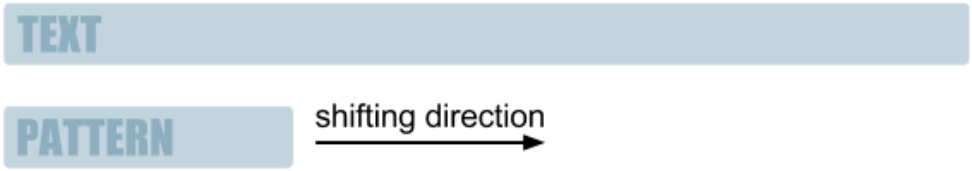
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
		G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
0	G	0	1	1	1	1	0	1	1	0	1	0	1	0	1	1	1	1	1	1	0	1	1	1	0
1	C	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	A	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	G	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	A	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	G	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
6	A	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
7	G	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1

Figuur 2.4

- eenmaal we S hebben , moeten we niet meer vergelijken, maar kunnen we rechtstreeks in de tabel opzoeken of het waar of false is.
- $R_{j+1} = Schuif(R_j)ENS[T[j+1]]$ Hierbij veronderstellen we dat er vooraan een waar bit (1) wordt ingeschoven. De waarde R_{j+1} wordt dan volledig bepaald door de eerste bit van S[T[j+1]], maw ze is dus : (T[j+1] == P[1]).
- Deze operaties kunnen elk met 1 processor instructie gebeuren : O(1)
- R_1 heeft geen voorloper $\rightarrow$ beginnen met een tabel die volledig onwaar is.
- Shift-or : nul inschuiven; schift-and: 1 inschuiven
- totale performantie is $\theta(n+m)$; zeer snelle methode voor kleine strings die bovendien weinig geheugen gebruikt.

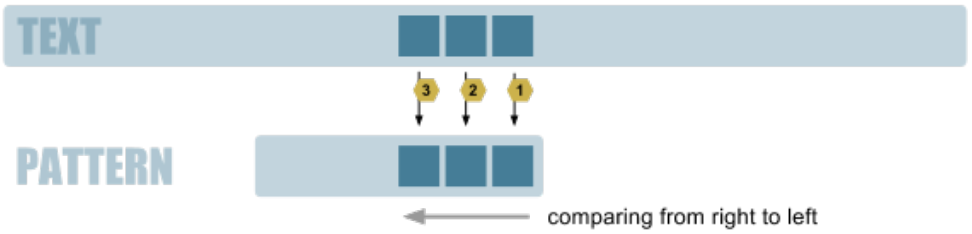
2.1.5 Boyer-Moore

Boyer-Moore is nog performanter dan bovenstaande methodes. Het zoekt naar een patroon door het patroon van links naar rechts te schuiven over de tekst.



Figuur 2.5

Een belangrijk verschil met de voorgaande methodes is dat we het patroon zelf van rechts naar links aflopen.

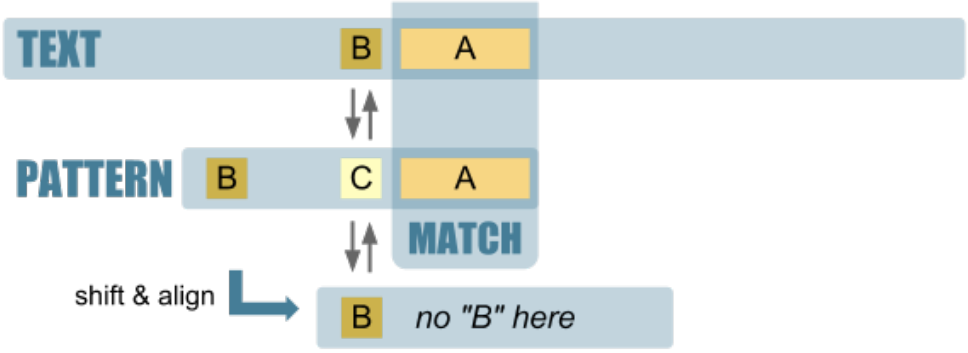


Figuur 2.6

boyer-moore maakt gebruik van 2 heuristieken om de performantie te garanderen.

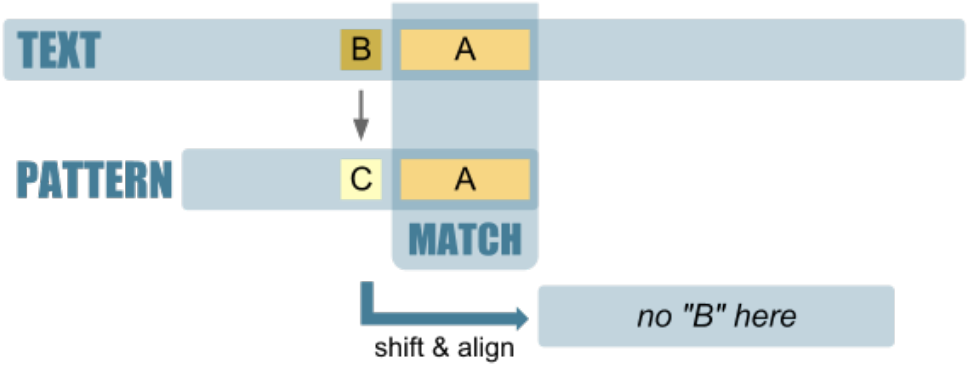
Verkeerde karakter

Als we gestopt zijn bij een mismatch, dan kunnen we P naar rechts schuiven. Dit heeft echter slechts zin als het laatste character van P overeenkomt met het bovenliggende char in T (anders direct terug een fout). Je kan zien in de figuur dat we zullen doorschuiven tot waar B overeenkomt. Immers het stuk van C tot aan B in het pattern bevat geen eerdere B, en kan dus ook geen matches veroorzaken.



Figuur 2.7

Een ander geval is dat er geen B in het Pattern zit, dan kunnen we direct het hele patroon doorschuiven.



Figuur 2.8

Er bestaan echter verschillende varianten/implementaties van deze heuristiek. Merk op dat als we de heuristiek fout implementeren we foutieve verschuivingen kunnen krijgen in de omgekeerde richten. Als we bij 2.7 bijvoorbeeld een B hadden rechts in het patroon zou men het patroon terug schuiven, wat uiteraard niet de bedoeling is.

1. **Uitgebreide heuristiek:** Hierbij vermeden we negatieve verschuivingen, door de meest rechtste P te nemen, links van de patroonpositie. Hierdoor is de verschuiving steeds positief

2. We houden een 2-dim tabel bij om de verschuiving voor elke mogelijk letter op elke mogelijk positie te bepalen. Bepalen door P eenmaal van links naar rechts te overlopen

3. Compromis tussen plaats en tijd:

   - geen tabel maar een gelinkte lijst bijhouden die dalend gesorteerd is.
   - iets trager, maar gebruikt minder geheugen.

4. **Variant van Horspool:** betere oplossing die dezelfde tabel als de originele methode gebruikt. Voor elk karakter in het alfabet bevat de tabel de meest rechtse positie j van dat karakter in P , links van positie m. Als het karakter daar niet voorkomt dan is j = 0. Wanneer er bij het vergelijken een fout optreedt bij patroonpositie i en tekstpositie k dan moet P opgeschoven worden. Hierbij moet het nieuwe karakter terug gelijk zijn, dit kunnen we vinden met index T[k+m-i]. Vervolgens verschuiven we m-j positief, wat steeds positief is.

   De verschuiving is bovendien onafhankelijk van de foutieve patroon positie i , en meestal groter dan de oorspronkelijke versie.

5. **Variant van Sunday:** gebruikt dezelfde tabel als de originele methode, maar als er een fout optreed tussen patroon positie i en tekstpositie k, dan zorgt men ervoor dat bij de verschuiving het gepaste patroonkarakter tegenover tekstkarakter T[k+m-i+1] terecht komt (net voorbij P[m] dus).

   Hierbij is ook de verschuiving onafhankelijk van de foutieve patroonpositie i. Het gevolg hiervan is dat de karakters van P die met T vergeleken moeten worden, geen rol meer speelt. We kunnen ze dus overlopen op de kans dat ze voorkomen.

Deel IV

P en NP

Deel V

Artikels