Föreläsning 12

Trie och Heap
och lite sökning

Innehåll

- Trie
- Heap
- och lite sökning

Trie

- Ytterligare en variant av träd. Vi har tidigare sett:
  - **Oordnat träd** där barnen till en nod bildar en *mängd*
  - **Ordnat träd** där barnen till en nod bildar en *lista*
- I **Trie** är barnen till en nod organiserade som tabellvärden i en *tabell* som hör till noden.
- Trie kallas också för *diskrimineringsträd*, *code-link tree* eller *radix-search tree*.

Organisation av Trie

- Man når barnen (delträden) direkt genom ”namn”, dvs argument/nycklar i barnnodens tabell.
  - När man ritar träd brukar nycklarna skrivas direkt intill motsvarande båge.
- I en trie har tabellerna en och samma nyckeltyp, till exempel tecken.
- I många tillämpningar av Trie saknar de inre noderna etiketter, träden är lövträd.
- Trie är normalt nedåtriktad.
- Binära träd kan ses som ett specialfall av Trie där nyckelvärdena är left och right.

Informell specifikation

- Två sätt:
  - Utgå från Urträdets specifikation och låt typparametern sibling ha värdet Tabell.
    - Då hanteras insättning, borttagning och värdeskoll av Tabellen själv.
    - I övrigt används de vanliga operationerna för att sätta in och ta bort barn etc.
  - Sätt in lämpliga tabelloperationer direkt i specifikationen av Trie.
    - Insert-child blir tabellens Insert, Delete-child tabellens Remove och Child tabellens Lookup.

Konstruktion av Trie

- De flesta konstruktioner av träd går bra
  - Om det går bra att byta ut de delar som hanterar barnen (till exempel som element i en lista) till att hantera dessa som tabellvärden i en tabell.
    - En länkad lista med 2-celler byts till 3-celler.
  - Implementerar man tabellen som en vektor eller som en hashtabell får man effektiva Trieimplementationer.

Tillämpningar av Trie

- Används för att konstruera Lexikon av sekvenser eller Tabeller där nycklarna är sekvenser.
- För sekvenser med element av typ A väljer vi en Trie med tabellnycklar av typ A.
  - En sekvens motsvaras då av en väg i trädet från roten till ett löv.
  - Man lägger till en slutmarkör i slutet av varje sekvens om en sekvens kan vara början på en annan sekvens.
    - En annan variant är att ha etiketter i de inre noderna också.
- Ett viktigt/vanligt specialfall är Lexikon/Tabell av textsträng. En sträng kan ju ses som en lista eller vektor av tecken.

Forts…

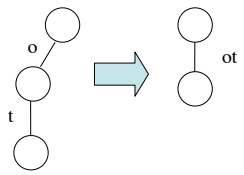
- Fördelar med att använda Trie för Lexikon/Tabeller som lagrar sekvenser som startar med samma följd av elementvärden:
  - Kompakt sätt att lagra lexikonet/tabellen på
  - Sökningens tidskomplexitet proportionell mot sekvenslängden. (En jämförelse per elementtecken)
  - Den relativa komplexiteten är oberoende av lexikonet/ tabellens storlek.
    - Det blir inte ”dyrare” att söka i ett stort lexikon jämfört med ett litet.

Tillämpningar

- Stavningskontroll
  - Skapa ett trie med alla ord som finns i språket.
- Översättningstabell
  - Löven innehåller motsvarande ord i ett annat språk.
- Filsystem på Unix/PC
- Datakomprimering
  - LZ78 algoritmen
  - Huffman kodning

Komprimerade Tries

- Alla enbarnsnoder konverteras till att innehålla hela strängen/sekvensen som är under.



Tries för strängar

- Insättning
  - I princip samma algoritm som insättning fast ”tvärtom”. Sök upp strängen som ska tas bort och radera nerifrån i trädet upp till första förgreningen.
  - När man hittar en skiljelinje, stanna och stoppa in resten av strängen som ett delträd.
  - Komprimerade tries:
    - Liknande algoritm men där är löven strängar som kan måsta delas upp i två barn
- Borttagning

LZ78 eller Lempel-Ziv kodning

- Kodning:
  - Låt frasen 0 vara strängen ””
  - Skanna igenom texten
    - Om du stöter på en ”ny” bokstav lägg till den på toppnivån på trien.
    - Om du stöter på en ”gammal” bokstav gå nedåt i trien tills du inte kan matcha fler tecken, lägg till en nod i trien som representerar den nya strängen.
    - Stoppa in paret (nodeIndex, sistaBokstaven) i den komprimerade strängen.
- Exempel:
  ”how now brown cow in town.”

LZ78 eller Lempel-Ziv kodning

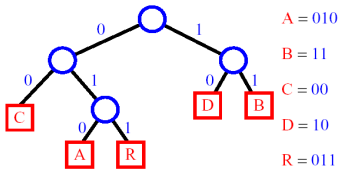
- Avkodning:
  - Varje gång du stöter på ”0” i strängen lägg nästa bokstav i strängen direkt efter den föregående i den avkodade strängen.
  - För varje index < > 0 stoppa in delsträngen som motsvaras av den noden i den avkodade strängen, följt av nästa tecken i den komprimerade strängen.
  - Notera att man inte behöver skicka med trädet.
- Exempel: 0h0o0w0_0n2w4b0r6n4c6_0i5_0t9.

Filkomprimering

- ASCII-filer är textfiler där varje bokstav representeras av en 8-bitars ascii-kod.
  - Det är alltså en *fixlängdskodning*
- Om man tittar på en textfil ser man att vissa bokstäver förekommer oftare än andra.
- Om man lagrar vanligt förekommande bokstäver med färre bitar än ovanliga så skulle vi kunna spara utrymme.

Filkomprimering

- Kodningen måste ske så att man enkelt kan avkoda strängen *entydigt* med kännedom om hur de olika tecknen översätts.
  - Exempel: Antag att de tre tecknen a, b och c kodas som 0, 1 respektive 01.
    - Om en mottagare får strängen 001 vad betyder det? aab eller ac??
- Prefix-regeln: Ingen symbol kodas med en sträng som utgör prefix till en annan symbols kodsträng.



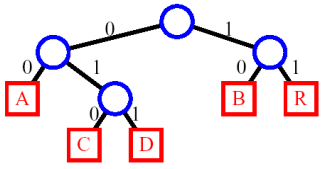
Vi använder ett trie!

- Bokstäverna lagras i löven.
- Den vänstra kanten betyder 0
- Den högra betyder 1

- Vad betyder 0101101101000010100101101101O

Vi vill ha optimal kompression!

- Så kort sträng som möjligt. Strängen 0101101101000010100101101101O = 29 bits kan kortas ned till 24 bitar (23 minst)
- 001011000100001100101100 med trädet



Huffman kodning

- Börja med en serie träd bestående av ett enda löv. Till varje löv associeras en symbol och en vikt = symbolens frekvens i texten som ska kodas.
- Välj de två träd som har minst vikt i roten. Bygg ihop dem till ett träd där de blir barn till en ny nod. Den nya noden innehåller en vikt = summan av barnens vikter.
- Upprepa förra punkten tills vi har ett enda stort träd.

Heap/Hög

- ett partiellt sorterat binärt träd
  - Etiketterna är sorterade efter en relation R så att a är förälder till b endast om a är före b i ordningen som ges av R.
  - Insättningar och borttagningar görs så att trädet hålls komplett.
  - Insert O(log n), Delete-first O(log n)

Linjär sökning

- Starta från början och sök tills elementet hittat eller sekvensen slut.
- Komplexitet
  - Elementet finns: I medel gå igenom halva listan, O(n)
  - Elementet saknas: I medel gå igenom hela listan, O(n)
- Om listan är sorterad:
  - Elementet saknas: Räcker i medel att leta genom halva listan n/2, O(n)

Binär sökning

- Om sekvensen har index (tex i en array eller numrerad lista) kan man söka binärt.
- Successiv halvering av sökintervallet.
- Vi får värsta-falls och medelkomplexitet O(log n).
- Jämför med elementet närmast mitten i intervallet.
  - Om likhet – klart!
  - Om det sökta värdet kommer före i sorteringsordningen fortsätt sökningen rekursivt i det vänstra delintervallet.
  - Om det kommer efter i sorteringsordningen fortsätt sökningen rekursivt i det högra delintervallet.

Exempel:

- **`1 2 4 4 6 7 9 13 14 19`**
- Sök efter elementet 13.
  - Linjär sökning: 8 jämförelser innan träff.
  - Binär sökning: 2 jämförelser innan träff.
- Sök efter elementet 10
  - Linjär sökning: 8 jämförelser innan man ger upp.
  - Binär sökning: 4 jämförelser innan man ger upp.