F13 - Trie, sökträd

5DV149 Datastrukturer och algoritmer
Kapitel 14.1–14.4

Niclas Börlin
niclas.borlin@cs.umu.se
Anna Jonsson
aj@cs.umu.se

2020-03-02 Mon

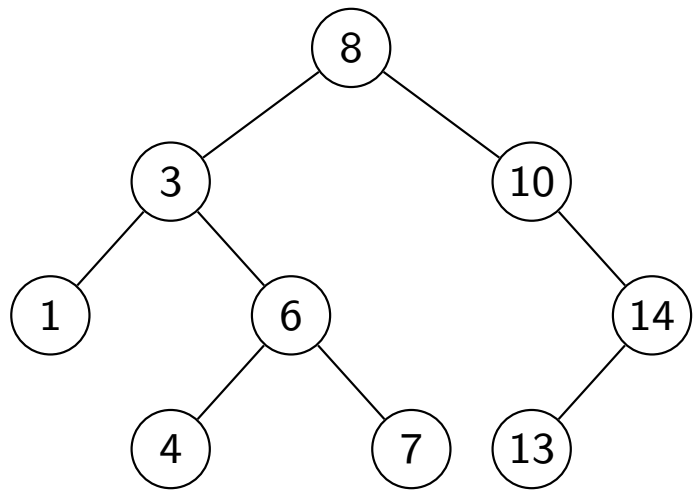
Innehåll

- Binära sökträd
- Trie
  - Filkomprimering

Binärt sökträd

- Används för sökning i linjära samlingar av dataobjekt, specifikt för att konstruera tabeller och lexikon.
- För ett binärt träd som är sorterat med avseende på en *sorteringsordning* R av etikett-typen så gäller att för varje nod *n*:
  - *n* har en definierad etikett,
  - alla noder *i* i *vänster* delträd kommer *före* *n*:
    - *i*.label R *n*.label är sann,
  - *n* kommer *före* alla noder *j* i *höger* delträd
    - *j*.label R *n*.label är falskt.

- Exempel: Ett binärt sökträd för heltal med R = ”<”:

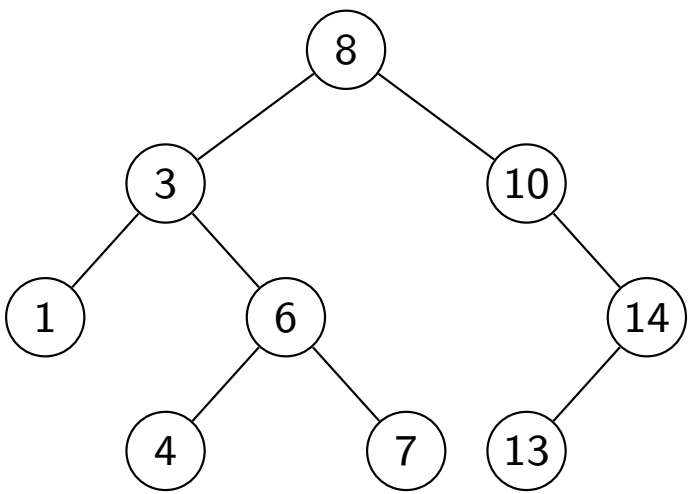


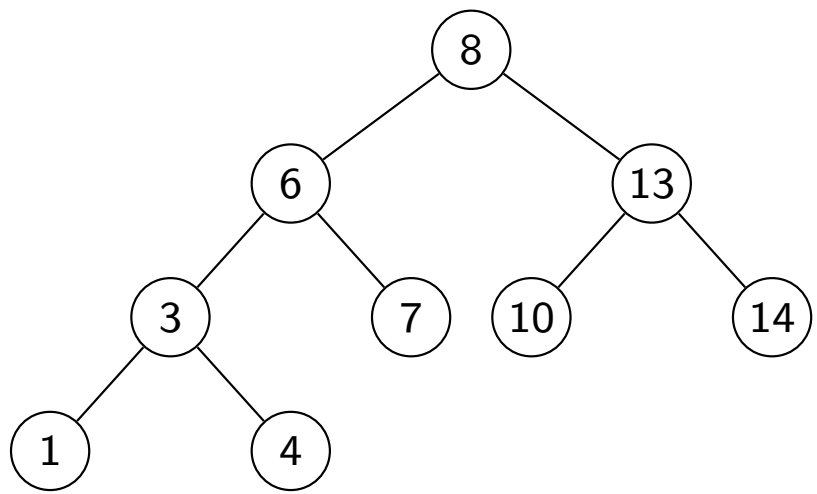
Binärt sökträd, informell specifikation

- Skiljer sig från ett vanligt binärt träd:
  - Alla noder måste ha etiketter:
    - Ta bort `Create`, `Has-Label` och `Set-Label` och inför `Make` som skapar rotnod med värde.
    - `Insert`-operationerna utökas med ett etikettvärde.
  - Man ska kunna ta bort inre noder också, inte bara löv:
    - Positionsparametern i `Delete-node` behöver inte peka på ett löv.
    - När man rycker bort en inre nod slits trädet sönder.
    - Hur lagar man det?
  - Är nedåtriktat:
    - Parent kan utelämnas.
  - Kan inte få stoppa in ett nytt element var som helst:
    - Måste uppfylla sorteringsordningen.

Varför sorterat träd?

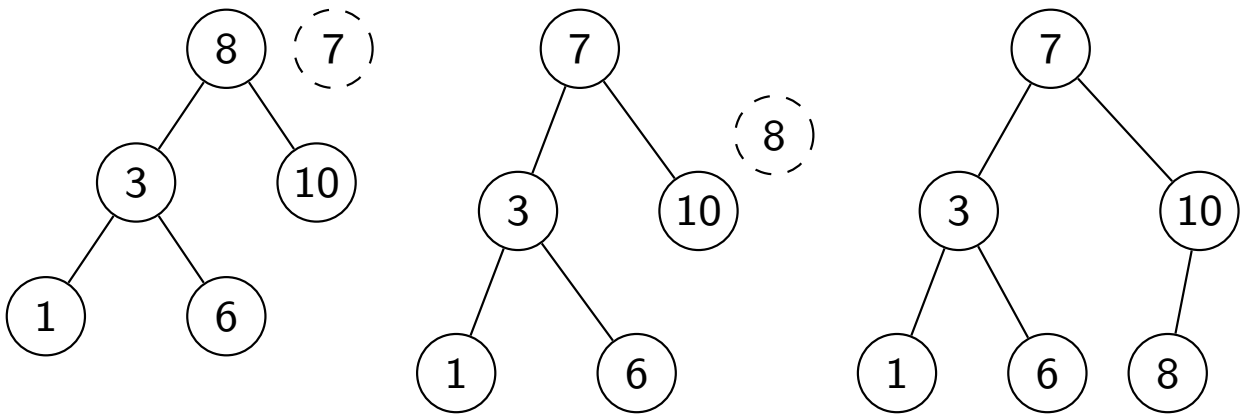
- Det går snabbare att söka i strukturen!
- För binärt sökträd:
  - Kolla om det sökta värdet finns i den aktuella noden.
  - Om inte, sök rekursivt nedåt i vänster eller höger delträd beroende på om det sökta elementet kommer före eller efter nodens värde i sorteringsordningen.
- Värstafallskomplexitet $O(\log n)$ om det binära trädet har minimal höjd, t.ex. är komplett.





Insättning i komplett träd

- Hur ser man till att trädet förblir komplett vid insättning?
- Om vi vid en jämförelse upptäcker att vänstra delträdet redan är fullt, men inte det högra:
  - Sätt in nya värdet i aktuell nod.
  - Gör insättning av gamla nodvärdet i höger delträd.

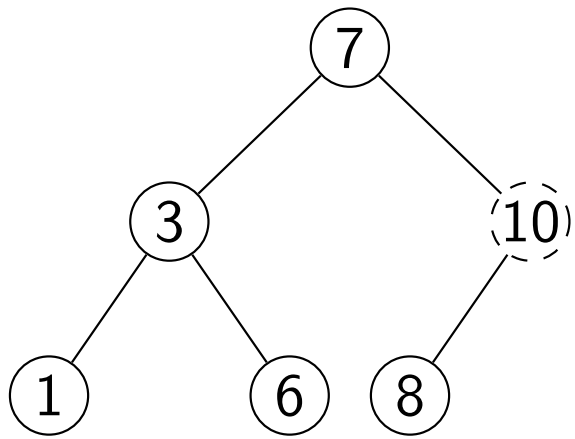


Borttagning av nod i binärt sökträd

- Hur lagar man ett träd när man tar bort en nod mitt i?

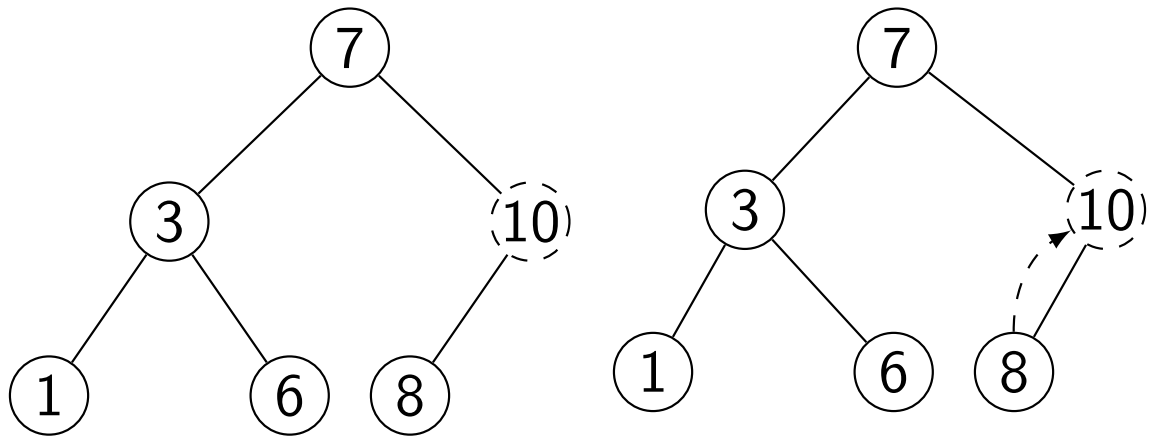
Borttagning av nod i binärt sökträd

- Hur lagar man ett träd när man tar bort en nod mitt i?
  - Om den borttagna noden bara har ett delträd:



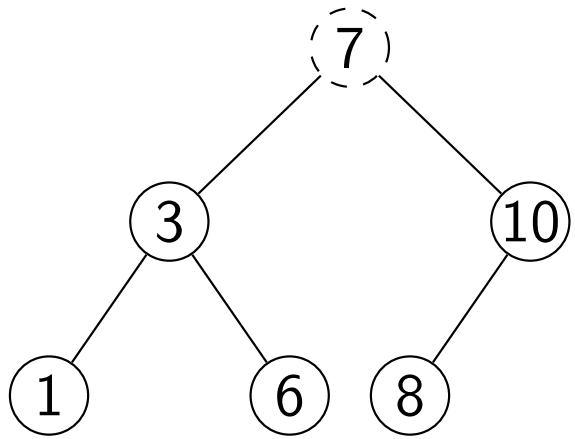
Borttagning av nod i binärt sökträd

- Hur lagar man ett träd när man tar bort en nod mitt i?
  - Om den borttagna noden bara har ett delträd:
    - Lyft upp värdet en nivå.



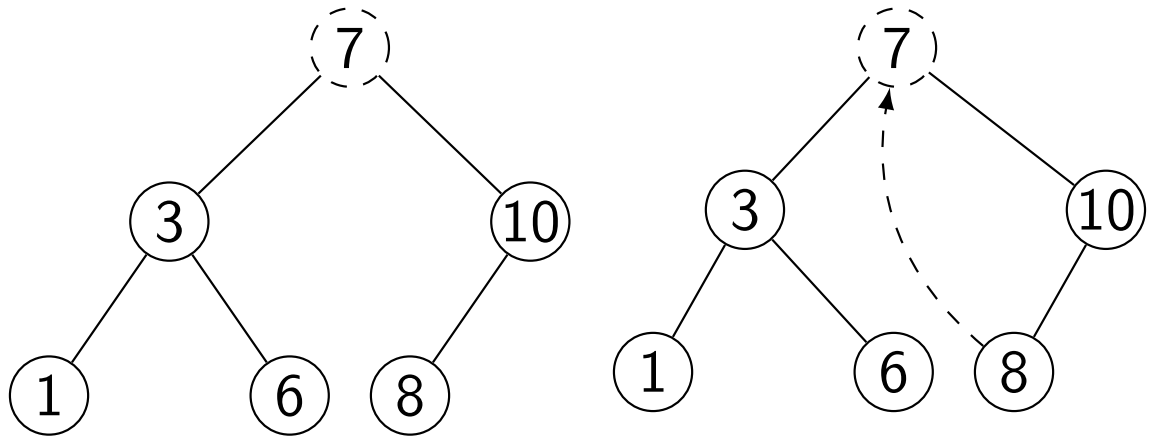
Borttagning av nod i binärt sökträd

- Hur lagar man ett träd när man tar bort en nod mitt i?
  - Om den borttagna noden bara har ett delträd:
    - Lyft upp värdet en nivå.
  - Om den borttagna noden har två delträd:



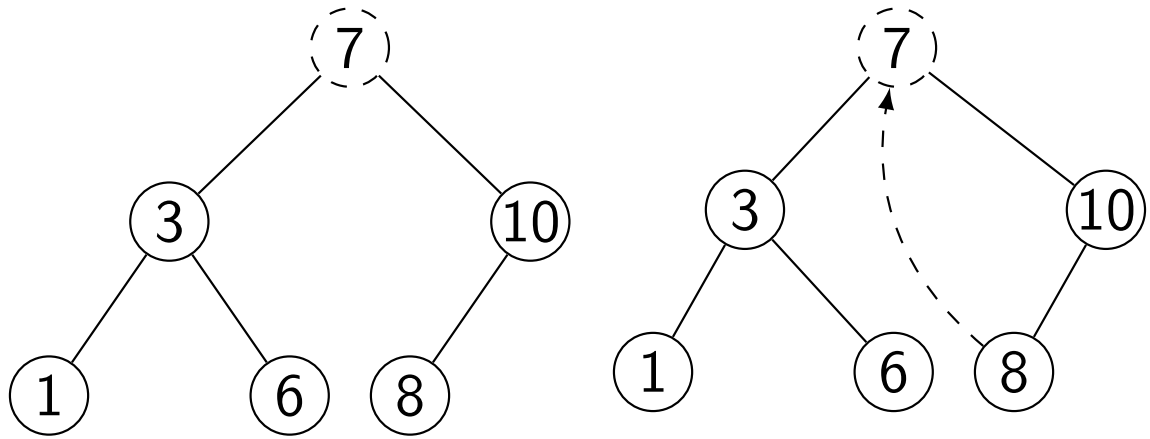
Borttagning av nod i binärt sökträd

- Hur lagar man ett träd när man tar bort en nod mitt i?
  - Om den borttagna noden bara har ett delträd:
    - Lyft upp värdet en nivå.
  - Om den borttagna noden har två delträd:
    - Välj noden med det lägsta värdet i höger delträd.



Borttagning av nod i binärt sökträd

- Hur lagar man ett träd när man tar bort en nod mitt i?
  - Om den borttagna noden bara har ett delträd:
    - Lyft upp värdet en nivå.
  - Om den borttagna noden har två delträd:
    - Välj noden med det lägsta värdet i höger delträd.
- Detta är standardkonstruktionen, är upp till den som implementerar trädet.
  - De vanligaste tillämpningarna är inte beroende av denna detalj.
  - Viktigt att visa sitt beslut i specifikation och dokumentation.



Tillämpningar av Binärt sökträd

- Framför allt till konstruktioner av Lexikon och Tabell.
- Inorder-traversering av binärt sökträd ger en *sorterad sekvens* av de ingående elementen.
  - Sorteringsalgoritm:
    1. Stoppa in elementen ett och ett i ett tomt Binärt sökträd.
    2. Inorder-traversera trädet.

Generaliseringar

- Ett binärt sökträd underlättar sökning i en en-dimensionell datamängd.
- `C~~N~~R~~S~~T~~U~~Y`
- Lätt att generalisera detta till sökning i en 2-dimensionell datamängd (*quadtree*), 3-dimensionell (*octree*) eller högre.

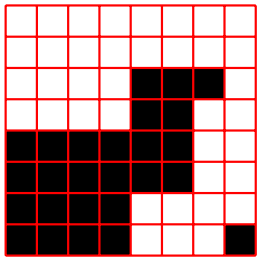
Quadtree (Fyrträd)

- Organiserat som ett ”binärt” träd med förgreningsfaktor 4.
- Tolkning (vanligast):
  - Rotnoden delar in den givna ytan (oftast kvadrat) i fyra lika stora kvadrater.
  - Vart och ett av de fyra barnen delar i sin tur sin kvadrat i fyra osv.
  - Inga koordinater behöver lagras i inre noder.
- Man kan använda det för att representera kurvor och ytor.

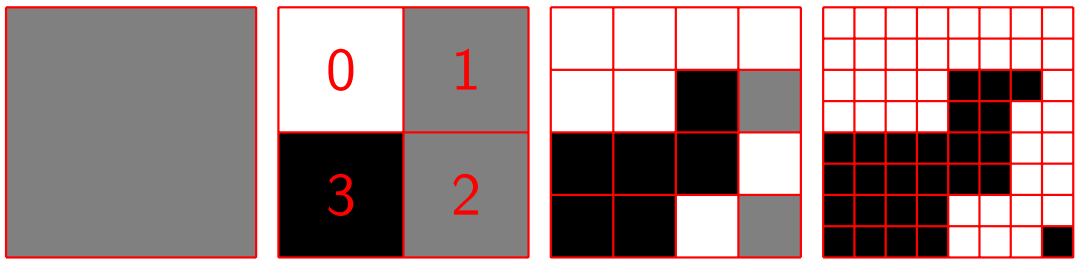
  Svarta kvadranter fylls helt av objektet.
  Grå kvadranter fylls delvis av objektet.
  Vita kvadranter innehåller inte objektet.

Quadtree, exempel

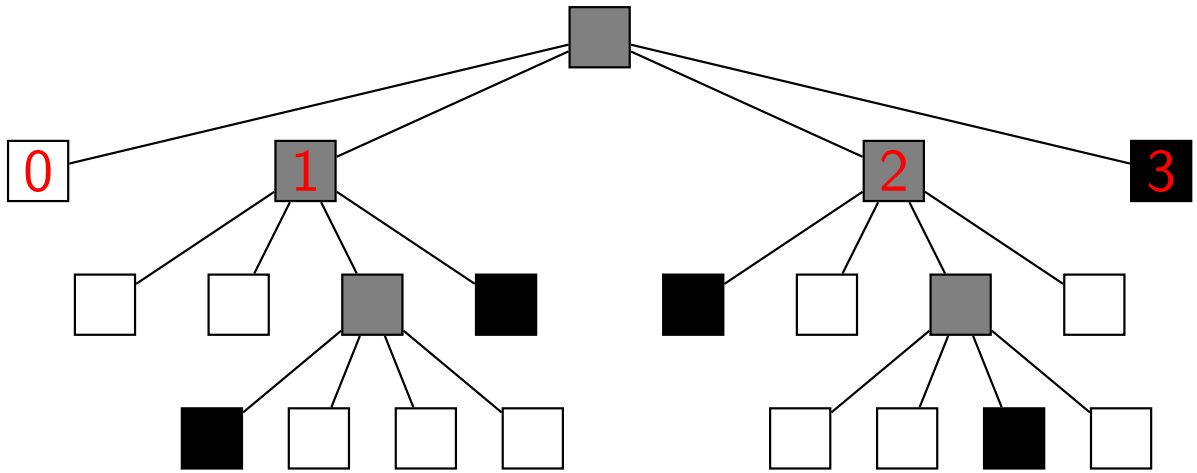
- Vi vill avgöra om ett objekt täcker koordinat (i, j) i denna bild:



- Bygg upp ett quad-tree av bilden:

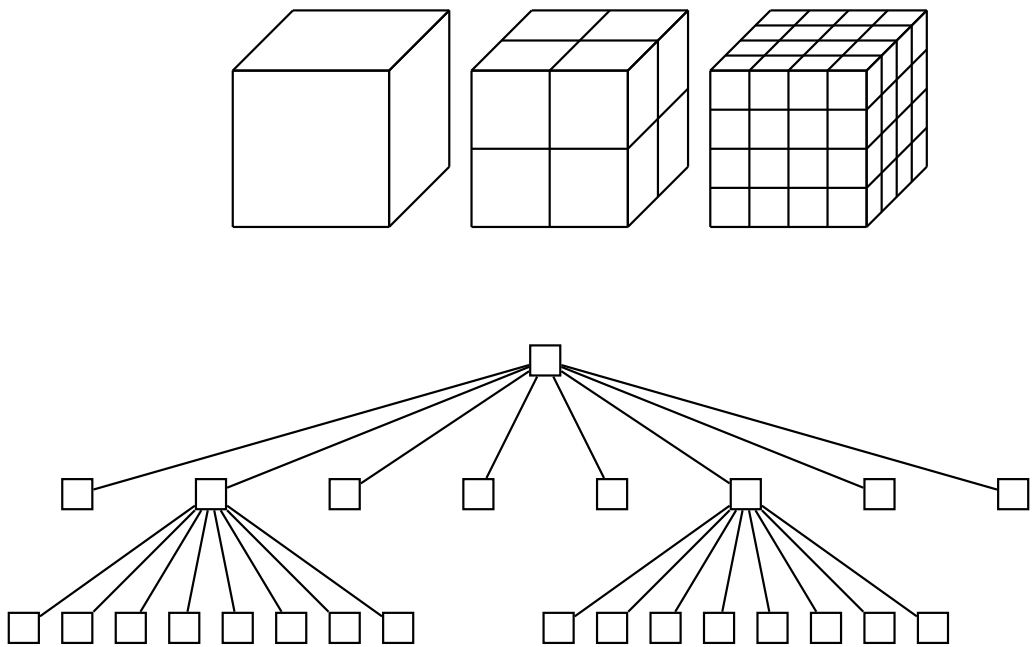


- Sök i det trädet.



Octree

- Samma, fast med en förgreningsfaktor på 8.



Quadtree++, tillämpningar

- 2D: Geografiska informationssystem (GIS).
- 3D: Kollisionsdetektion vid 3D-simuleringar.

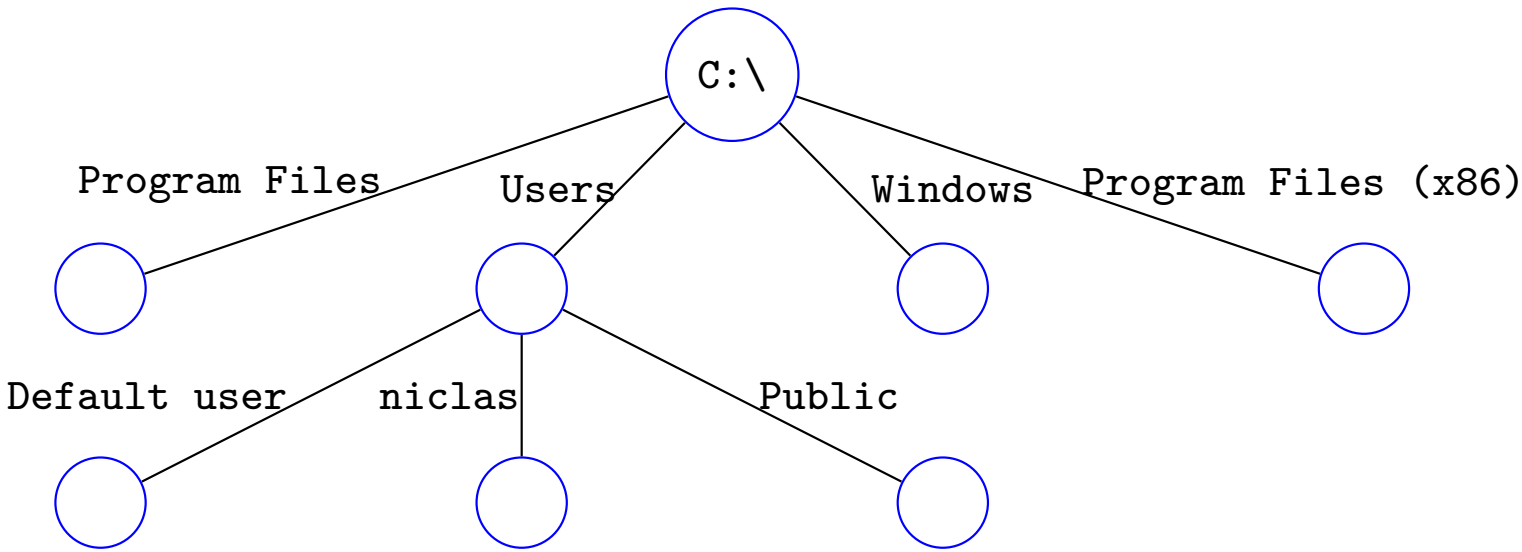
Trie

- Från *retrieve*, uttalas Traj.
- Ytterligare en variant av träd. Vi har tidigare sett:

  Oordnat träd Barnen till en nod bildar en *mängd*.
  Ordnat träd 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*, *radix-search tree* eller *prefix-träd*.

Organisation av Trie

- Man når barnen (delträden) direkt genom "namn", dvs argument/nycklar i nodens barntabell.
  - När man ritar träd brukar nycklarna skrivas direkt intill motsvarande båge.



- I en Trie har tabellerna en och samma nyckeltyp, till exempel tecken eller strängar.

Organisation av Trie

- I många tillämpningar av Trie saknar de inre noderna etiketter, träden är *lövträd*.
- Trie är normalt nedåtriktad.

Informell specifikation

- Två sätt:
  - Utgå från Urträdets specifikation och låt typparametern `sibling` ha värdet Tabell.
    - Då hanteras insättning, borttagning och uppslagning av Tabellen.
    - 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`.

Implementation av Trie

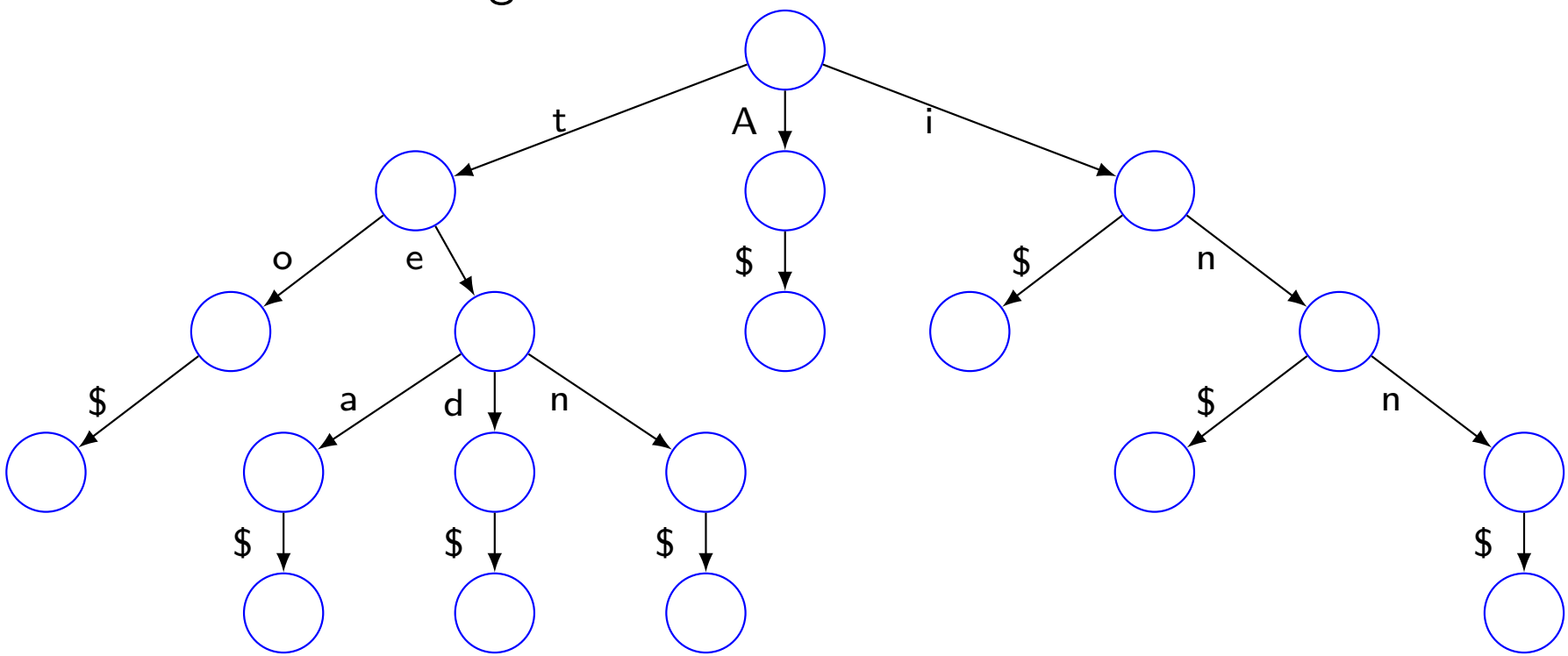
- De flesta Träd-implementationer går bra att utgå från.
- Man måste byta ut de delar som hanterar barnen till att hantera dessa som tabellvärden i en Tabell:
  - En länkad lista med 2-celler byts till 3-celler.
- Implementerar man tabellen som en vektor eller som en hashtabell får man effektiva Trie-implementationer.

Tillämpningar av Trie (1)

- Används för att konstruera Lexikon av sekvenser eller Tabeller där nycklarna är *sekvenser*.
- Ett viktigt exempel är Lexikon/Tabell av textsträng.
- 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.
  - Om sekvenserna kan vara av variabel längd:
    - Lägg till en slutmarkör i slutet av varje godkänd sekvens.

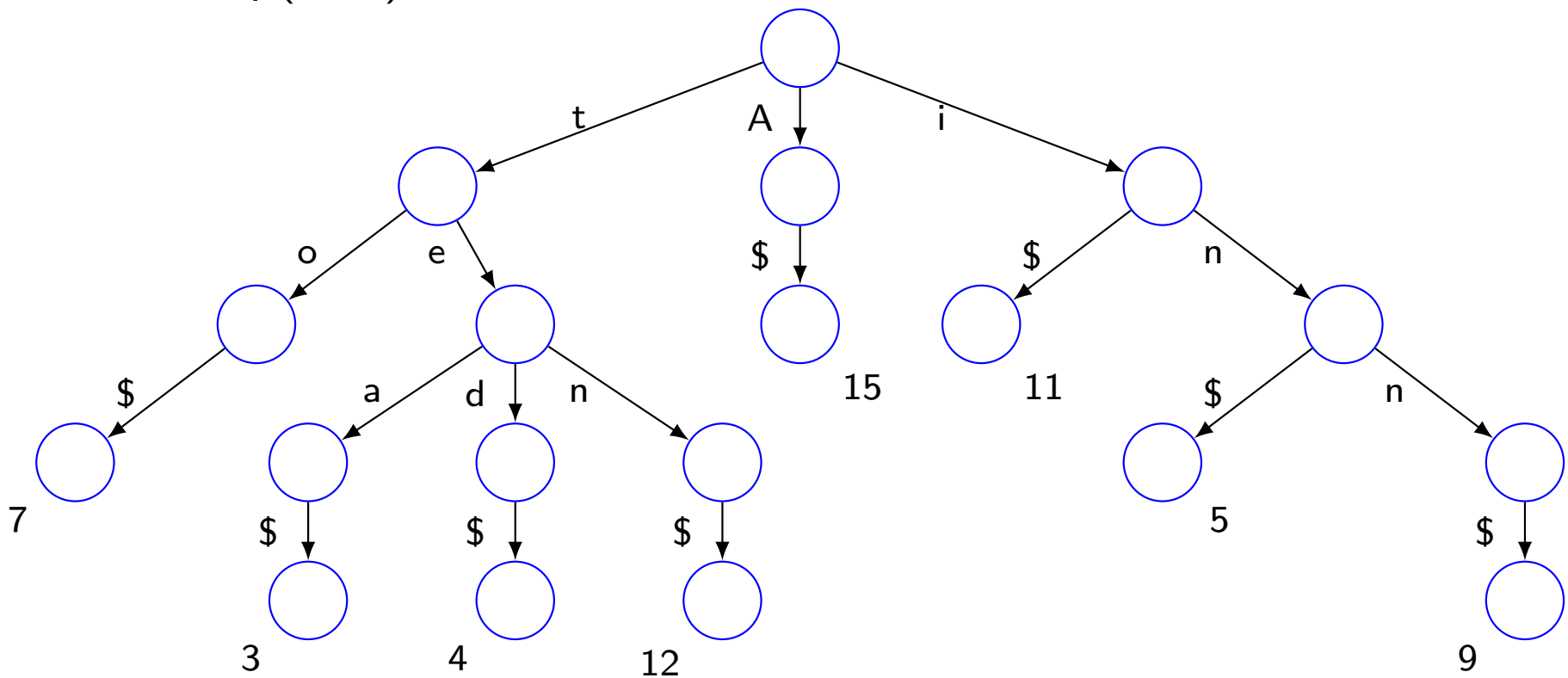
Tillämpningar av Trie (2)

- För lexikon så motsvarar varje godkänd sekvens att sekvensen ingår i lexikonet.
- Exempelvis så ingår sekvensen *ten* och *in* i nedanstående lexikon.
- Sekvensen *te* ingår inte.



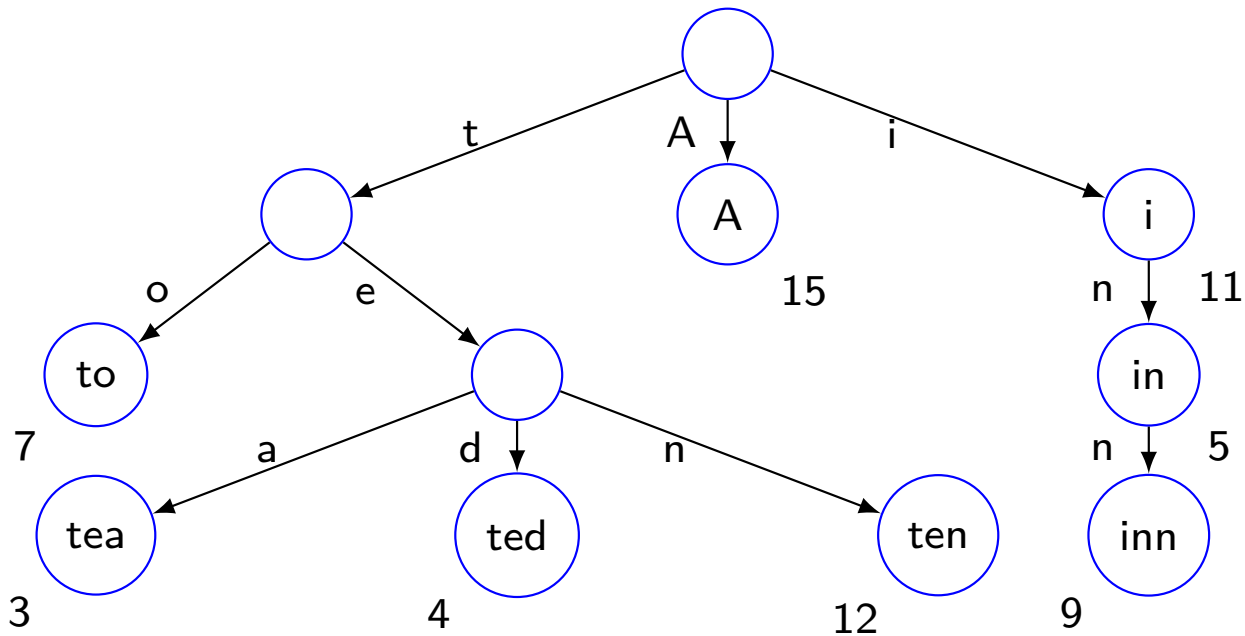
Tillämpningar av Trie (3)

- För tabeller associeras tabellvärden med lövnoderna.
- Lookup("in") skulle returnera värdet 5.



Tillämpningar av Trie (4)

- En annan variant för variabla sekvenslängder är att ha definierade etiketter för alla godkända noder (löv och inre noder).



Frågor

- Antag vi vill skapa ett lexikon av ord (sekvenser av tecken).
- Om vi konstruerar lexikonet som en Tabell (utan tabellvärden):
  - Vad är tidskomplexiteten för en sökning?
  - Hur förändras tidskomplexiteten för en sökning med antalet ord n?
- Om vi konstruerar lexikonet some ett Trie:
  - Vad är tidskomplexiteten för en sökning?
  - Hur förändras tidskomplexiteten för en sökning med antalet ord n?

Fördelar med Trie

- 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 av Trie (5)

- Stavningskontroll:
  - Skapa ett Trie med alla ord som finns i språket.
- Översättningstabell:
  - Löven innehåller motsvarande ord i ett annat språk.
- Filsystem.
- Datakomprimering:
  - Huffman-kodning
    https://en.wikipedia.org/wiki/Huffman_coding.
  - LZ78-algoritmen
    https://en.wikipedia.org/wiki/LZ77_and_LZ78.

Filkomprimering, fixlängdskodning

- ASCII-filer är textfiler där varje bokstav representeras av en 8-bitars ASCII-kod.
  - `A = 65 = 01000001`
  - `B = 66 = 01000010`
  - `C = 67 = 01000011`
  - `D = 68 = 01000100`
  - `R = 82 = 01010010`
- Varje symbol har en fix längd — *fixlängdskodning*.

Filkomprimering, variabel kodlängd

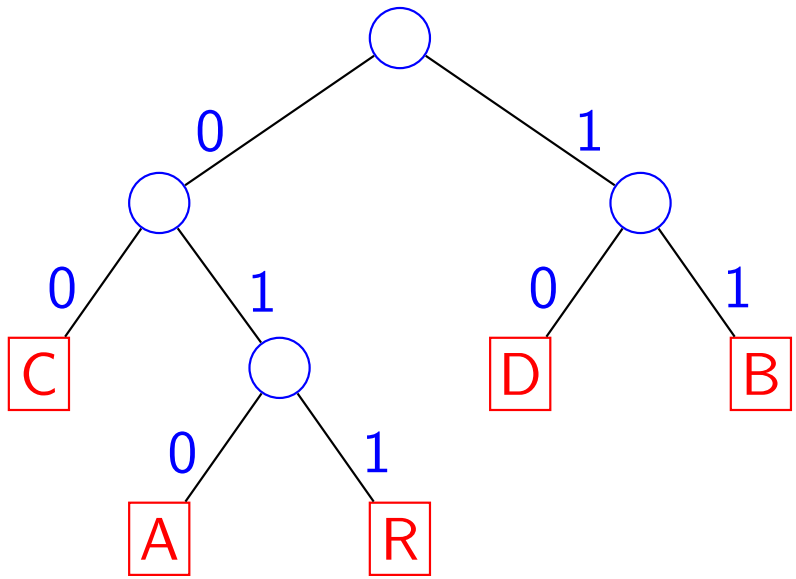
- Om man tittar på en textfil ser man att vissa bokstäver förekommer oftare än andra.
- Om man lagrar vanligt förekommande bokstäver med *färre bitar* än ovanliga så skulle man kunna spara utrymme.
  - Morse-alfabetet ett tidigt exempel:
    - A = .-
    - E = .
    - I = ..
    - O = --
    - Q = --.-
    - S = ...

Filkomprimering, prefixregeln

- Kodningen måste ske så att man enkelt kan avkoda strängen **entydigt**.
- Dåligt exempel:
  - Antag att tecknen `a`, `b` och `c` kodas som `0`, `1` respektive `01`.
  - Om en mottagare får strängen `001`, betyder det `aab` eller `ac`?
- Prefix-regeln: *Ingen symbol kodas med en sträng som utgör prefix till en annan symbols kodsträng.*
- Vi vill alltså:
  1. Använda sekvenser av variabel längd.
  2. Ingen sekvens som motsvarar en symbol får vara prefix till någon annan sekvens.

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

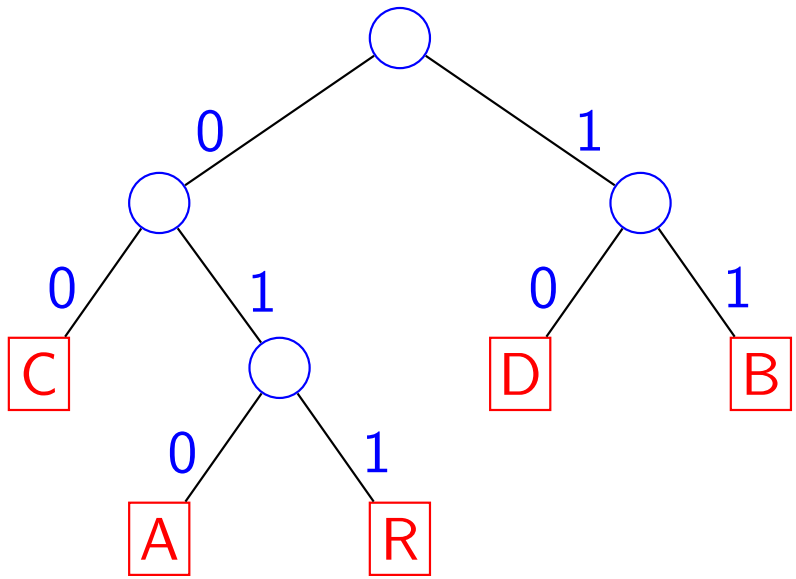


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
-

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

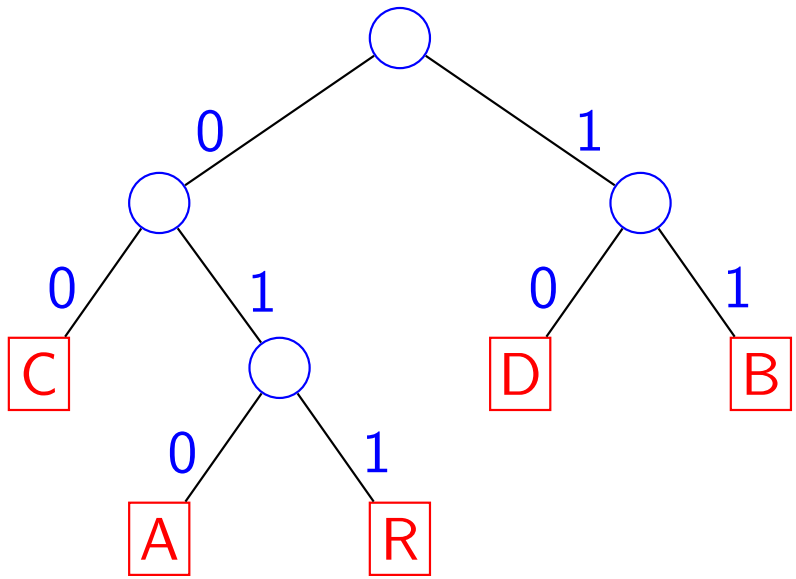


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A`

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

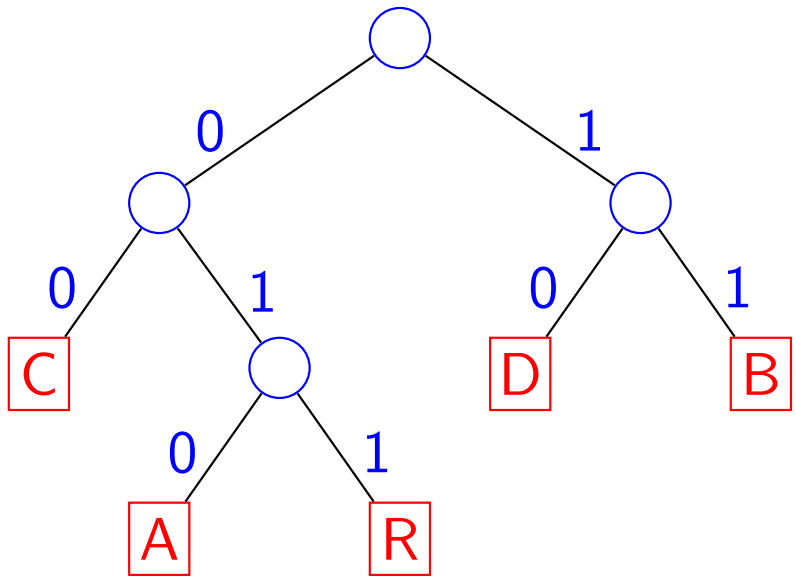


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A B`

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

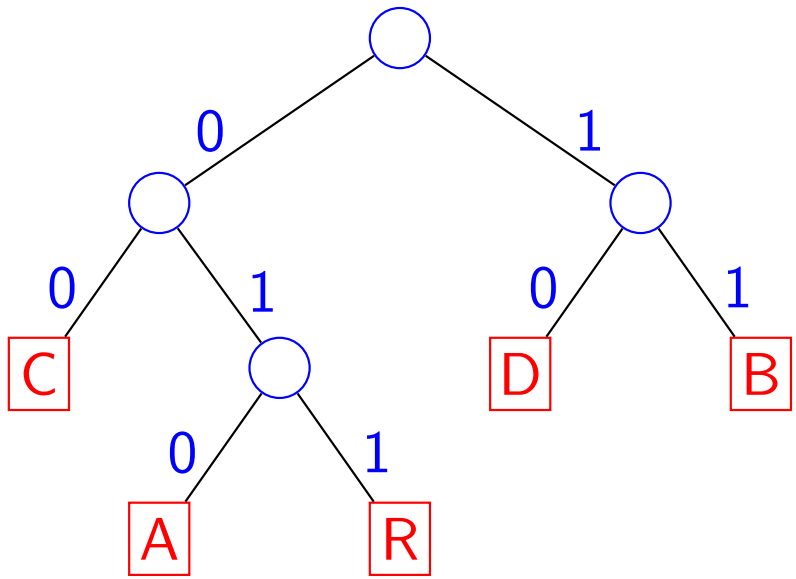


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A B R`

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

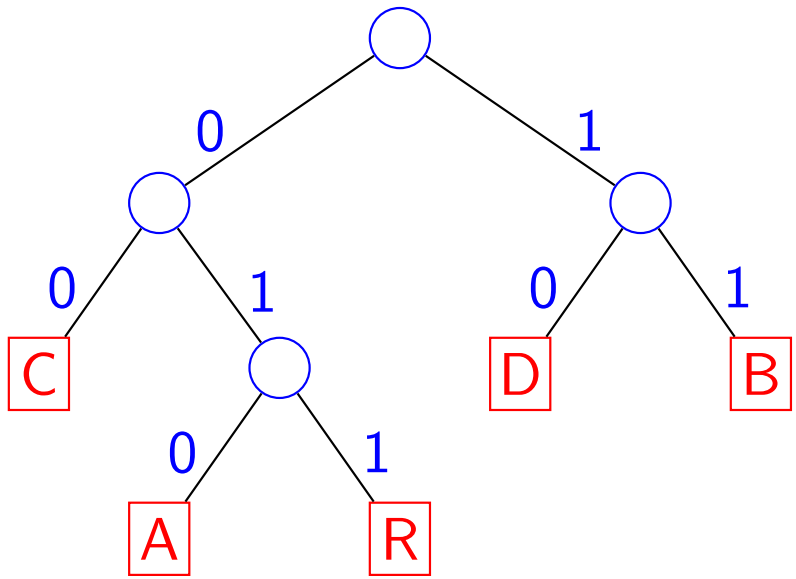


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A B R A`

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

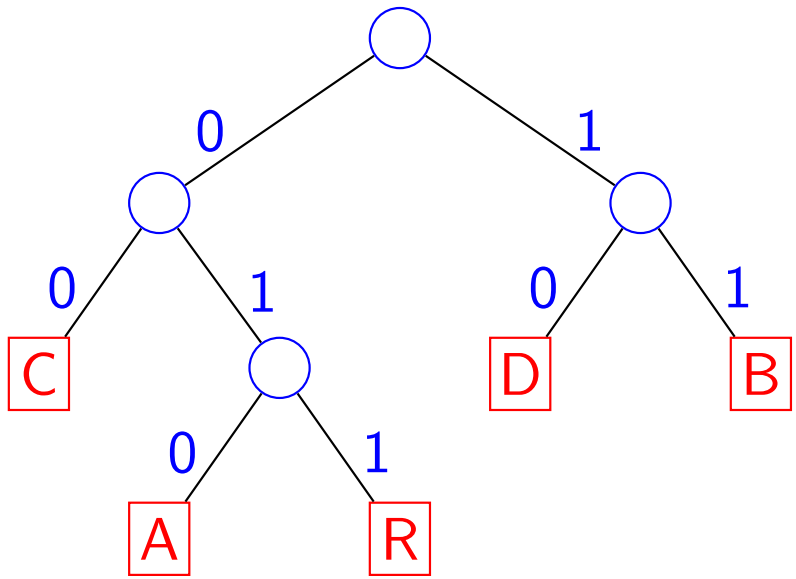


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A B R A C`

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

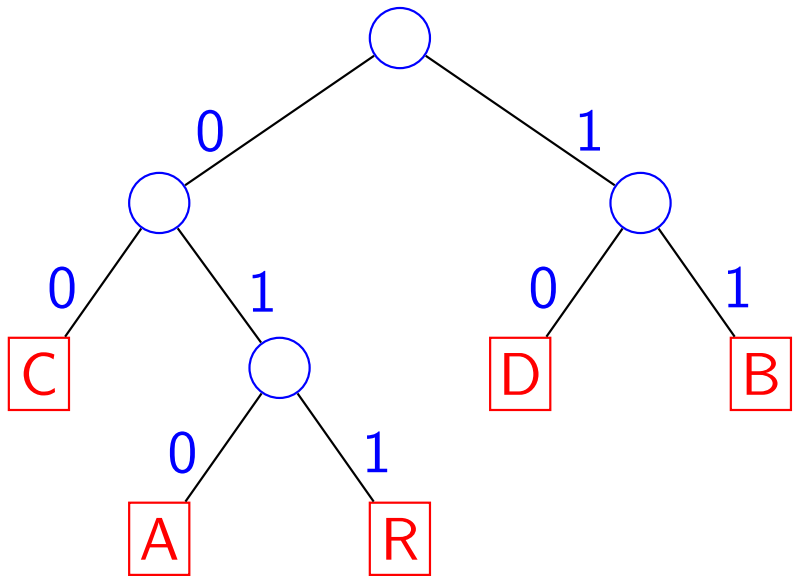


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A B R A C A`

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

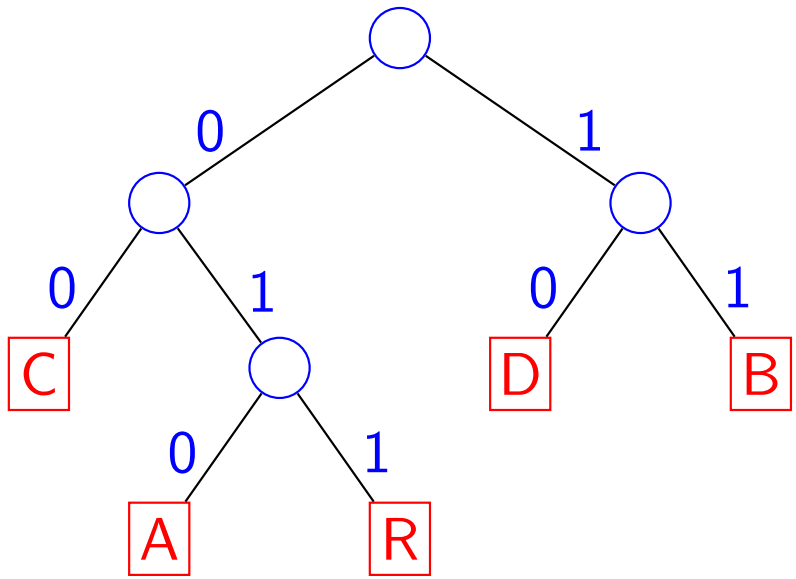


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A B R A C A D`

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

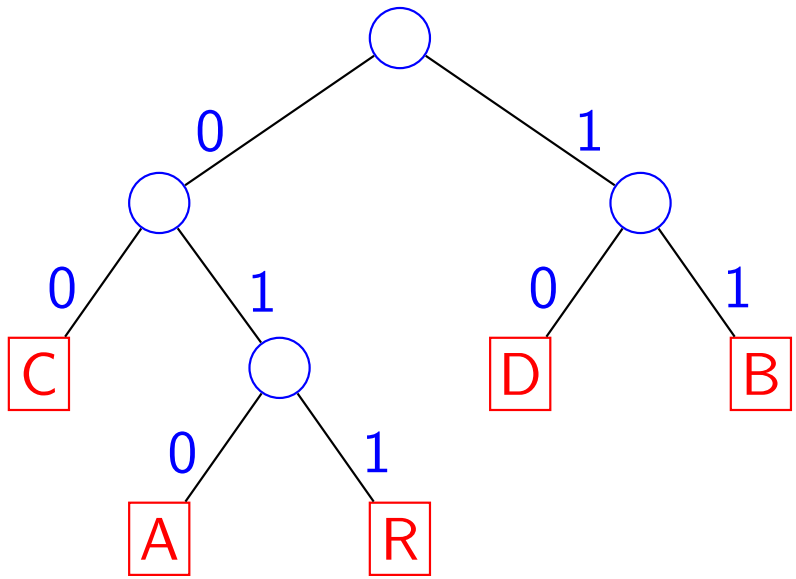


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A B R A C A D A`

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

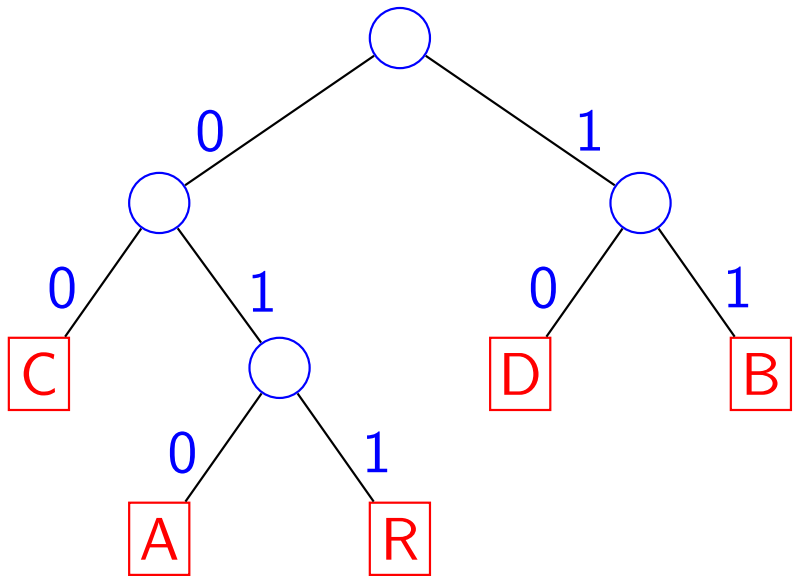


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A B R A C A D A B`

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.

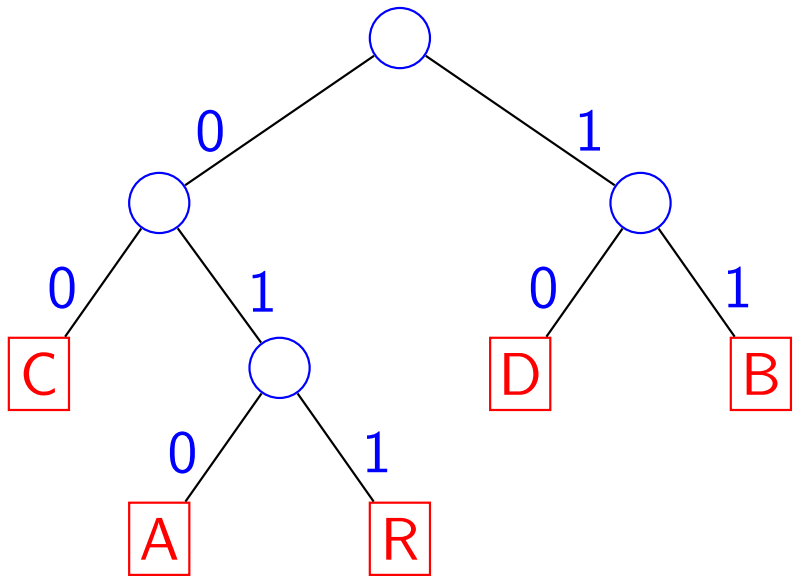


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A B R A C A D A B R`

Prefixkodning

- Använd ett Trie:
  - Bokstäverna lagras i löven.
  - Den vänstra kanten betyder 0.
  - Den högra kanten betyder 1.



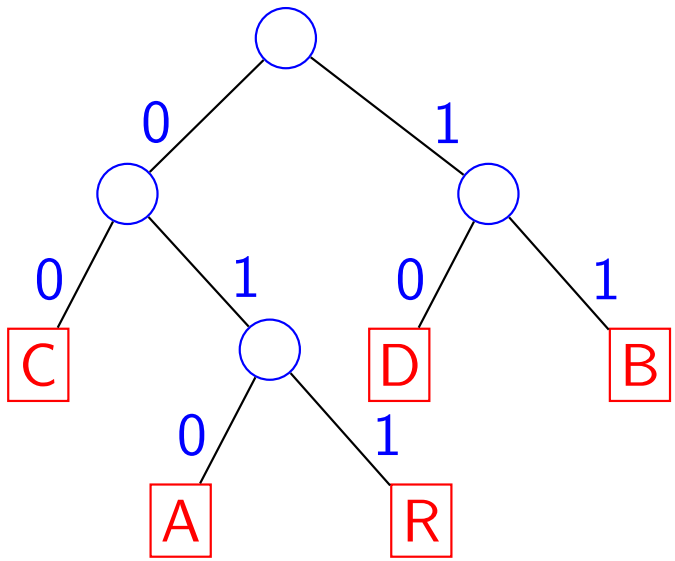
- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

- Vad betyder 01011011010000101001011011010?
- `A B R A C A D A B R A`

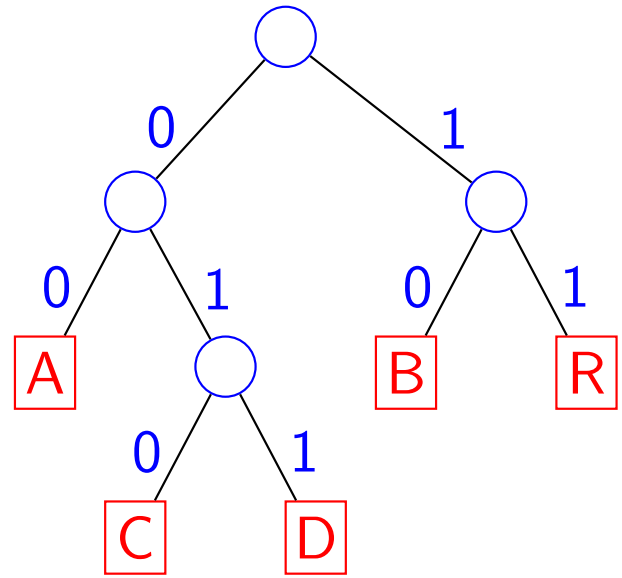
Optimal kompression

- Vilken tabell/träd vi har bestämmer kompressionens effektivitet.
- Med tabellen/trädet nedan får vi $01011011010000101001011011010 = 29$ bitar.
- Med tabellen/trädet till höger får vi $00101100010000110010110 0 = 24$ bitar.
- Varför?

- A = 00
- B = 10
- C = 010
- D = 011
- R = 11

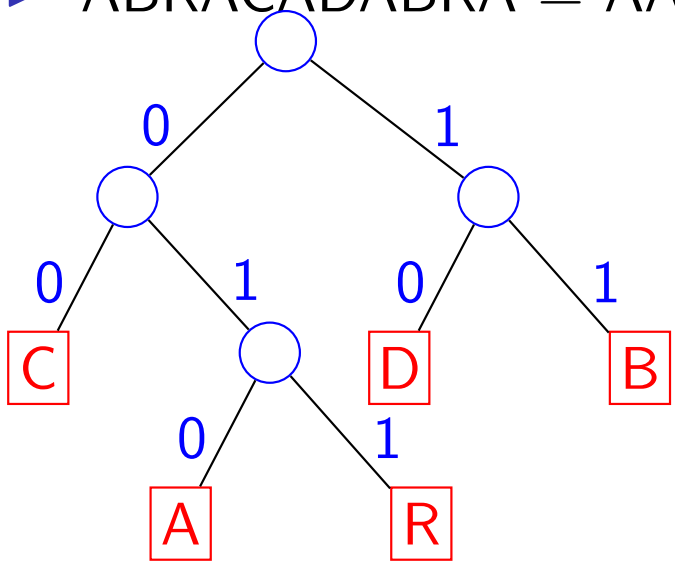


- A = 010
- B = 11
- C = 00
- D = 10
- R = 011

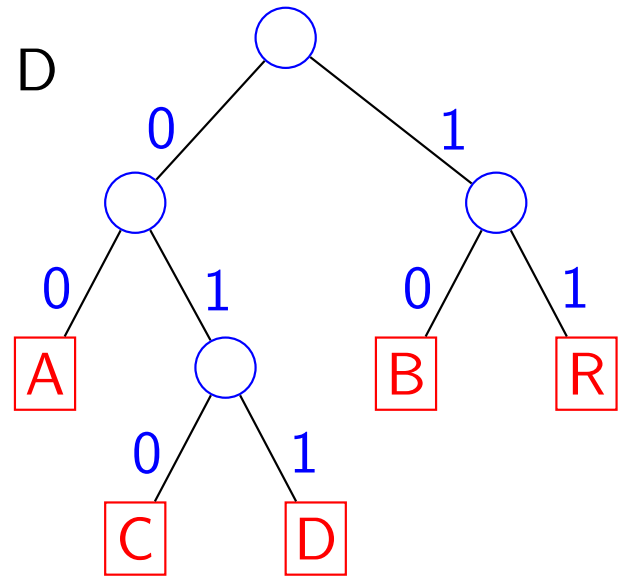


Optimal kompression

- Vilken tabell/träd vi har bestämmer kompressionens effektivitet.
- Med tabellen/trädet nedan får vi $01011011010000101001011011010 = 29$ bitar.
- Med tabellen/trädet till höger får vi $00101100010000110010110 0 = 24$ bitar.
- Varför?
- ABRACADABRA = AAAAA BB RR C D



- A = 010
- B = 11
- C = 00
- D = 10
- R = 011



- A = 00
- B = 10
- C = 010
- D = 011
- R = 11

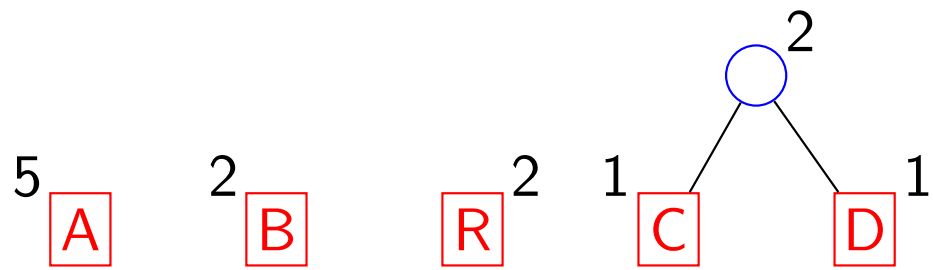
Huffman-kodning

Huffman-kodning

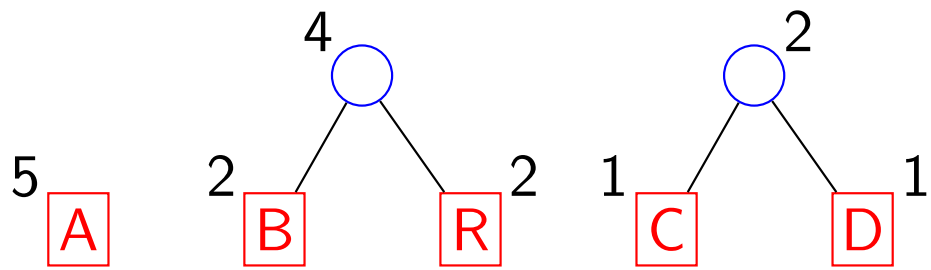
- Bygger upp optimalt träd från en *frekvenstabell*.
- Algoritm:
  - Börja med en mängd träd bestående av ett enda löv.
  - Till varje löv associeras en symbol och en vikt = symbolens frekvens i texten som ska kodas.
  - Upprepa tills vi har ett enda stort träd:
    - Välj de två träd som har minst vikt 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.
- Den genererade kodtabellen måste skickas först i meddelandet.

Huffman-kodning, exempel

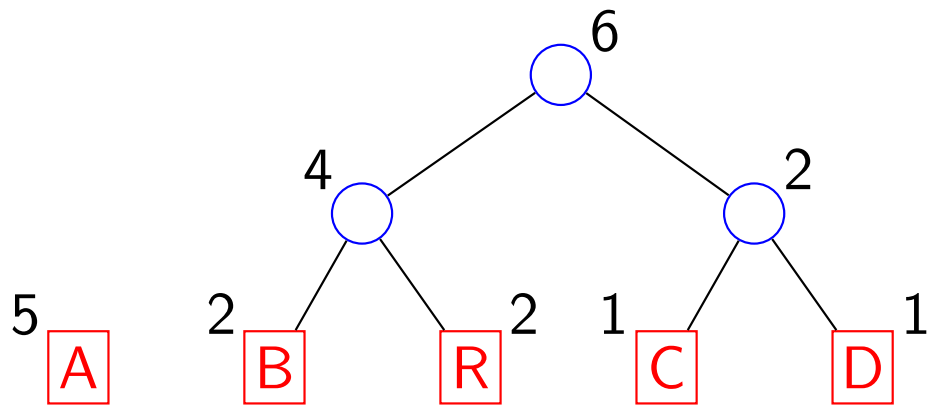
Huffman-kodning, exempel



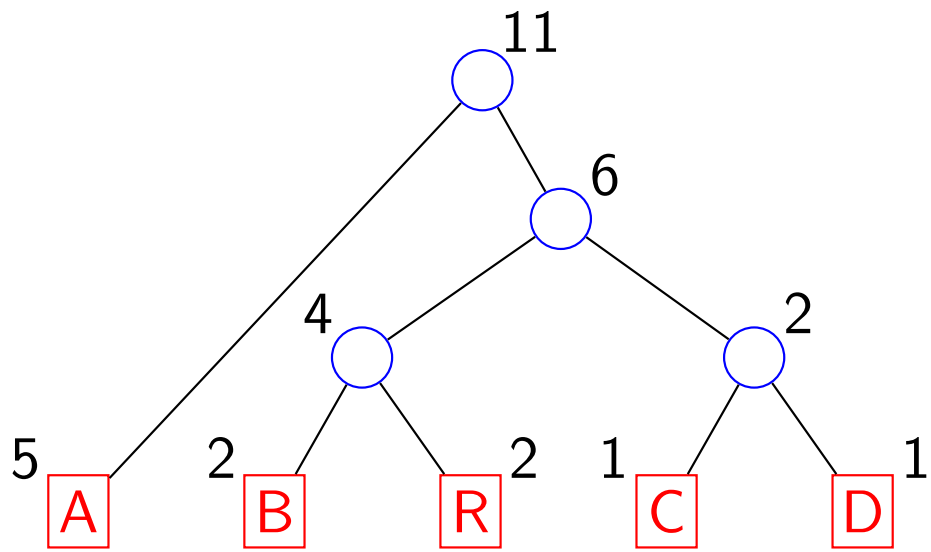
Huffman-kodning, exempel



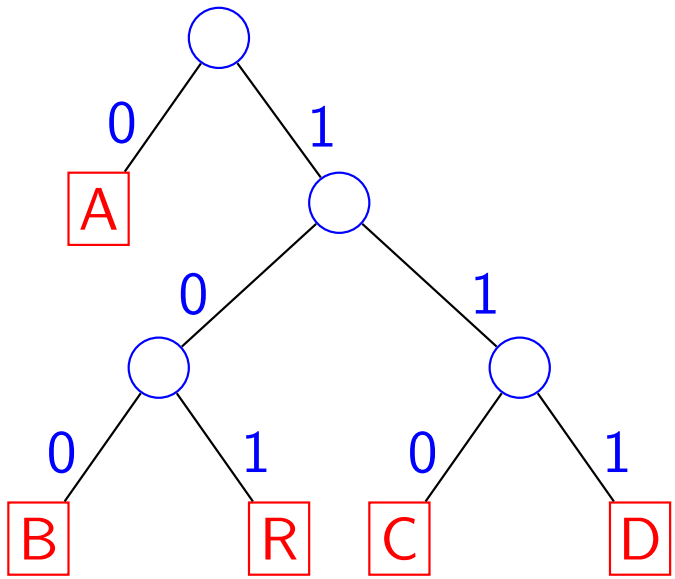
Huffman-kodning, exempel



Huffman-kodning, exempel



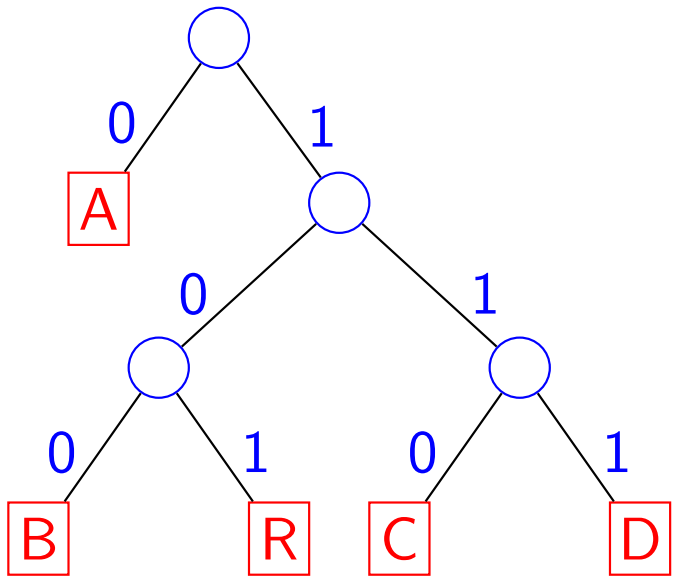
Huffman-kodning



- A = 0
- B = 100
- C = 110
- D = 111
- R = 101

Huffman-kodning

- Optimal tabell:
  - 01001010110011101001010 = 23 bitar
  - `AB R AC AD AB R A`



- A = 0
- B = 100
- C = 110
- D = 111
- R = 101

LZ-kodning (Lempel-Ziv-kodning)

LZ78 eller Lempel-Ziv-kodning, kodning

- Låt frasen 0 vara den tomma strängen.
- Skanna igenom texten:
  - Om du stöter på en ny, okänd, bokstav lägg till den på toppnivån på Triet.
  - Om du stöter på en gammal, känd, bokstav:
    - Gå nedåt i Triet tills du inte kan matcha fler tecken.
    - Lägg till en nod i Triet som representerar den nya strängen.
    - Stoppa in paret `(nodeIndex, lastLetter)` i den komprimerade strängen.
- Exempel: “how now brown cow in town”.

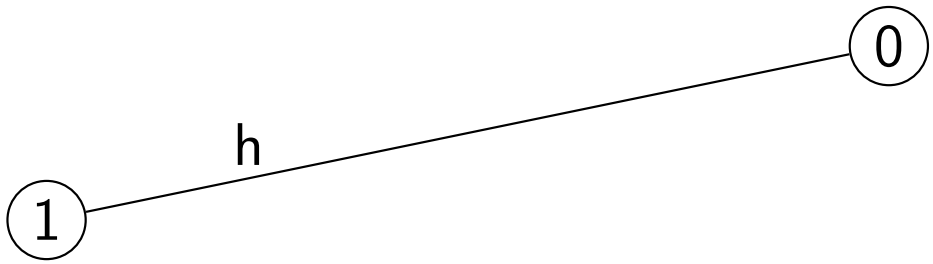
LZ78 kodningsexempel

- Input: `how␣now␣brown␣cow␣in␣town.`

- Output:

LZ78 kodningsexempel

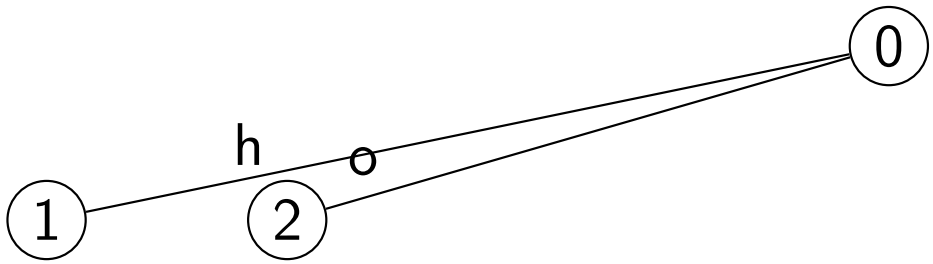
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h

LZ78 kodningsexempel

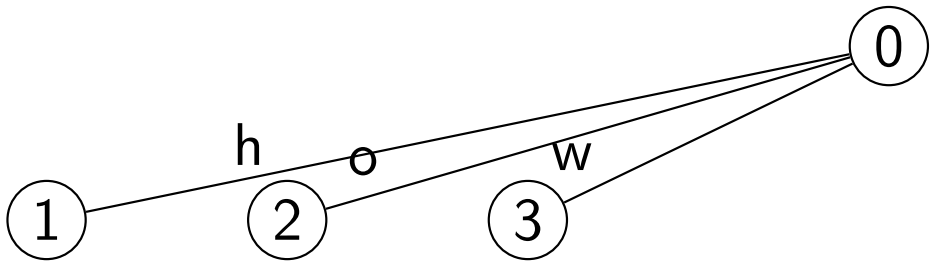
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o

LZ78 kodningsexempel

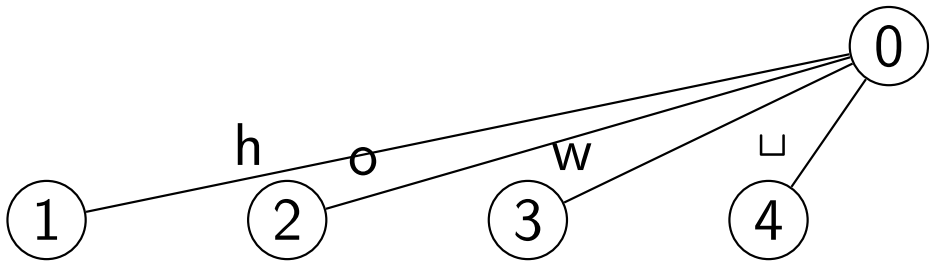
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w

LZ78 kodningsexempel

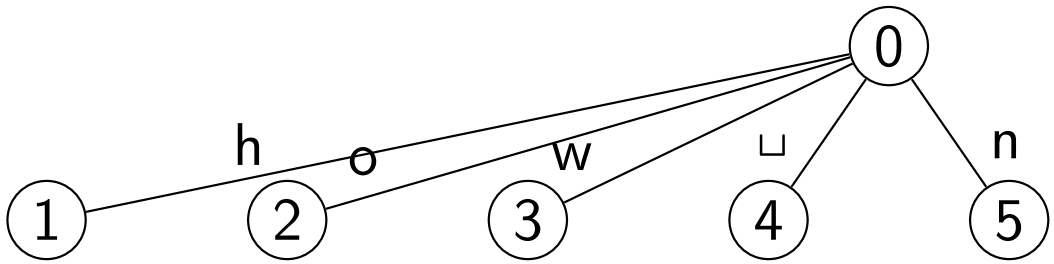
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣

LZ78 kodningsexempel

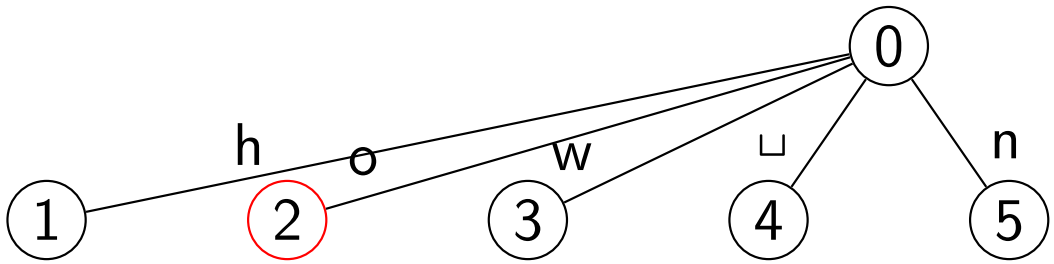
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n

LZ78 kodningsexempel

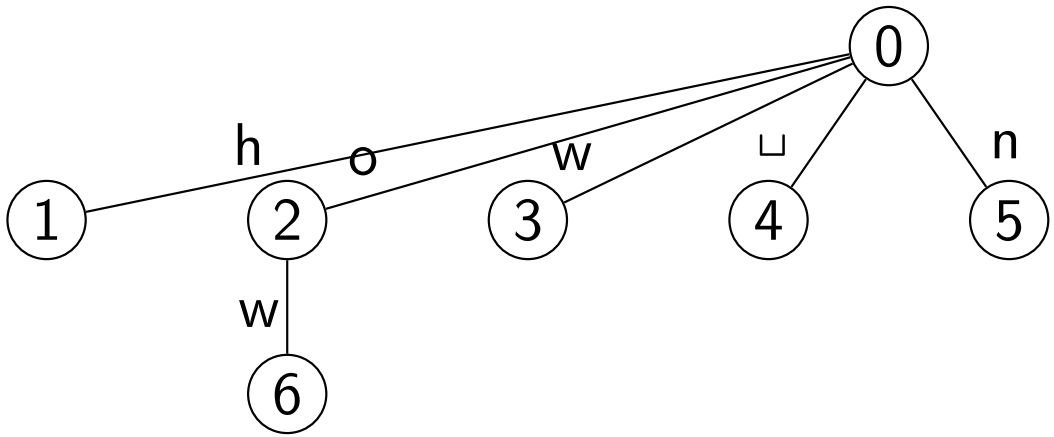
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n

LZ78 kodningsexempel

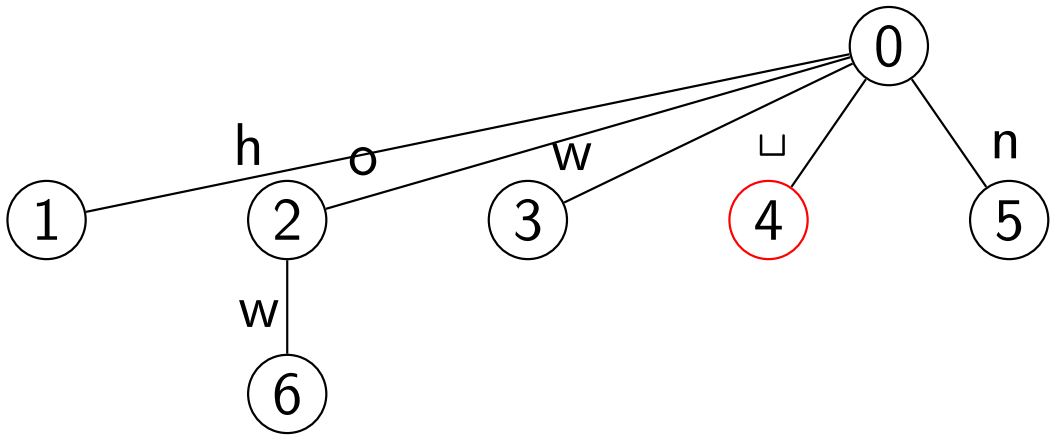
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w

LZ78 kodningsexempel

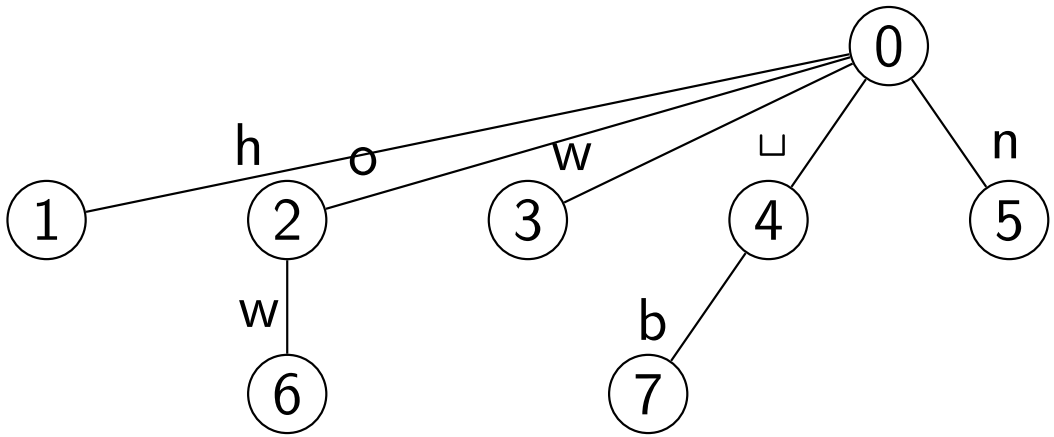
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w

LZ78 kodningsexempel

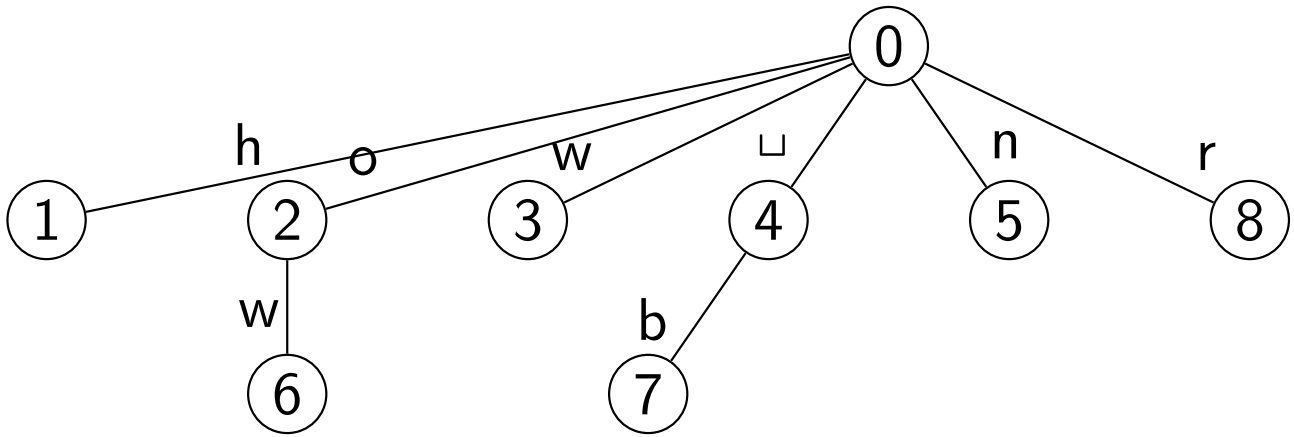
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b

LZ78 kodningsexempel

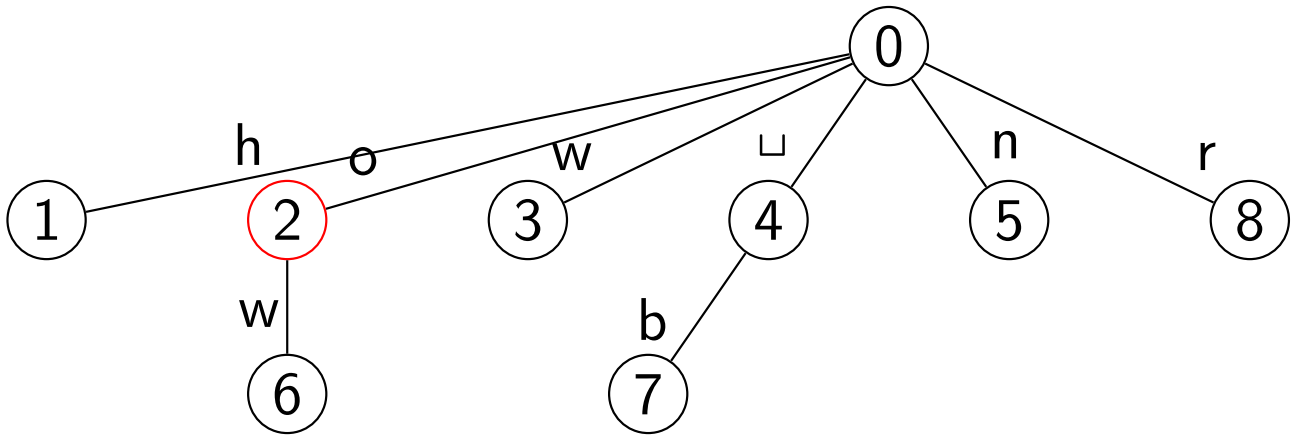
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r

LZ78 kodningsexempel

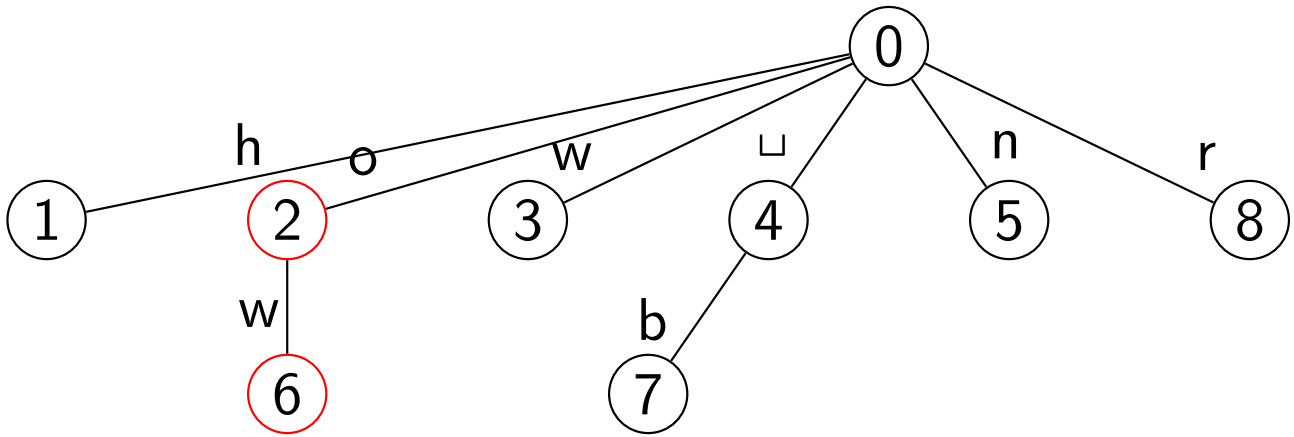
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r

LZ78 kodningsexempel

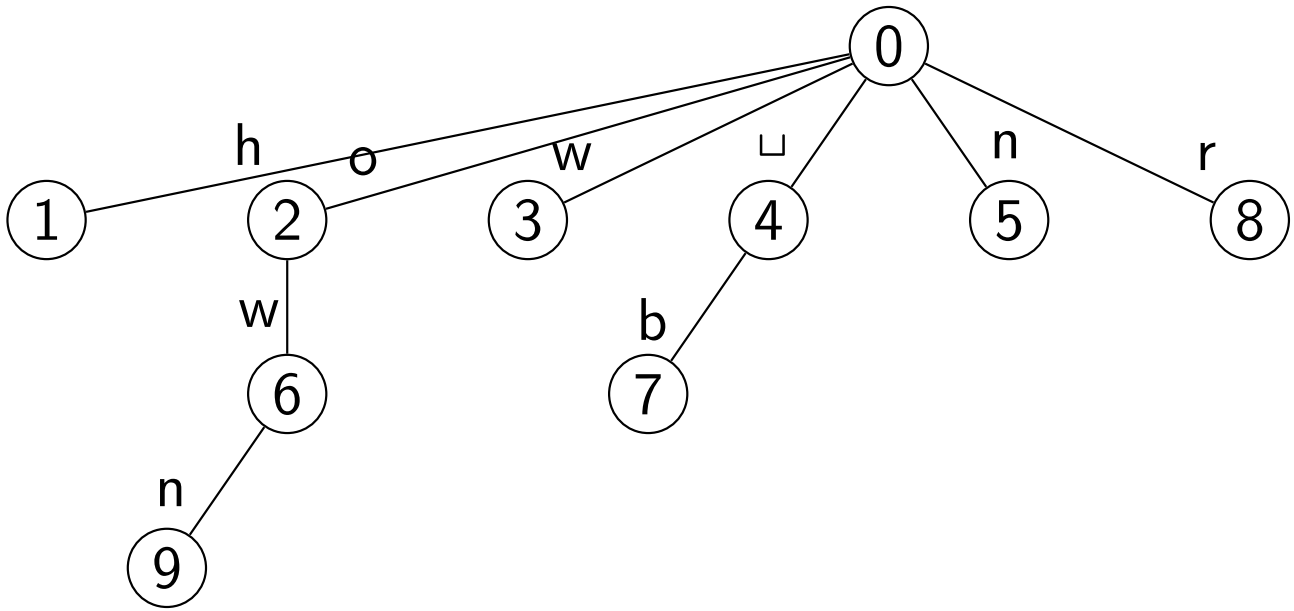
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r

LZ78 kodningsexempel

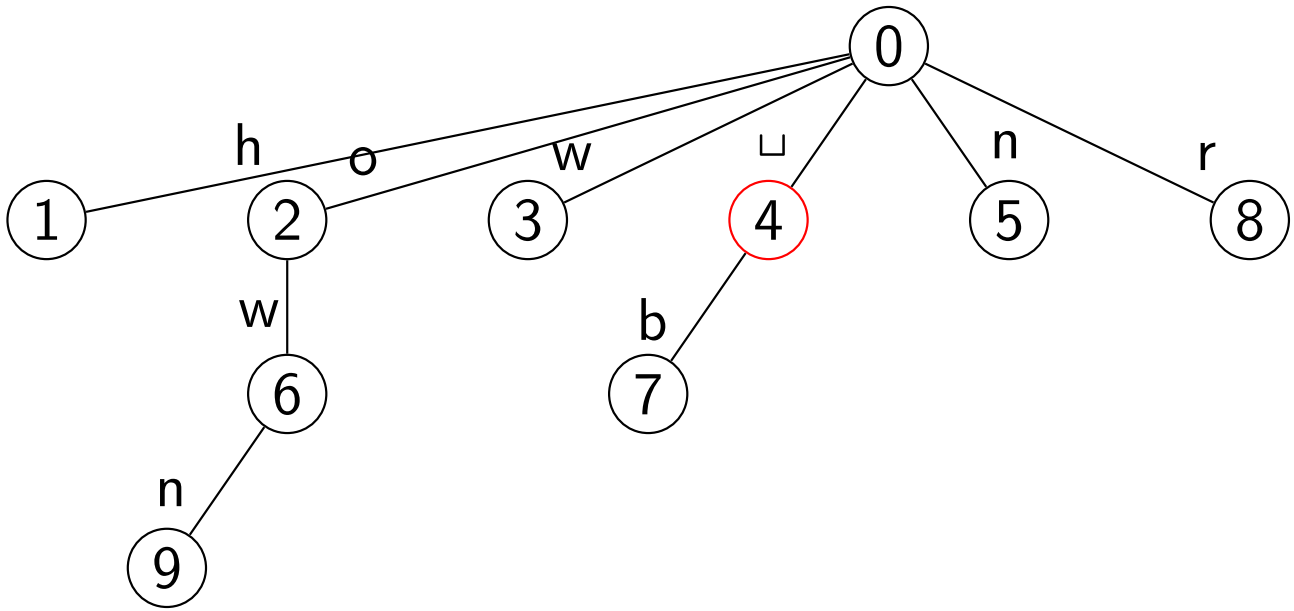
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n

LZ78 kodningsexempel

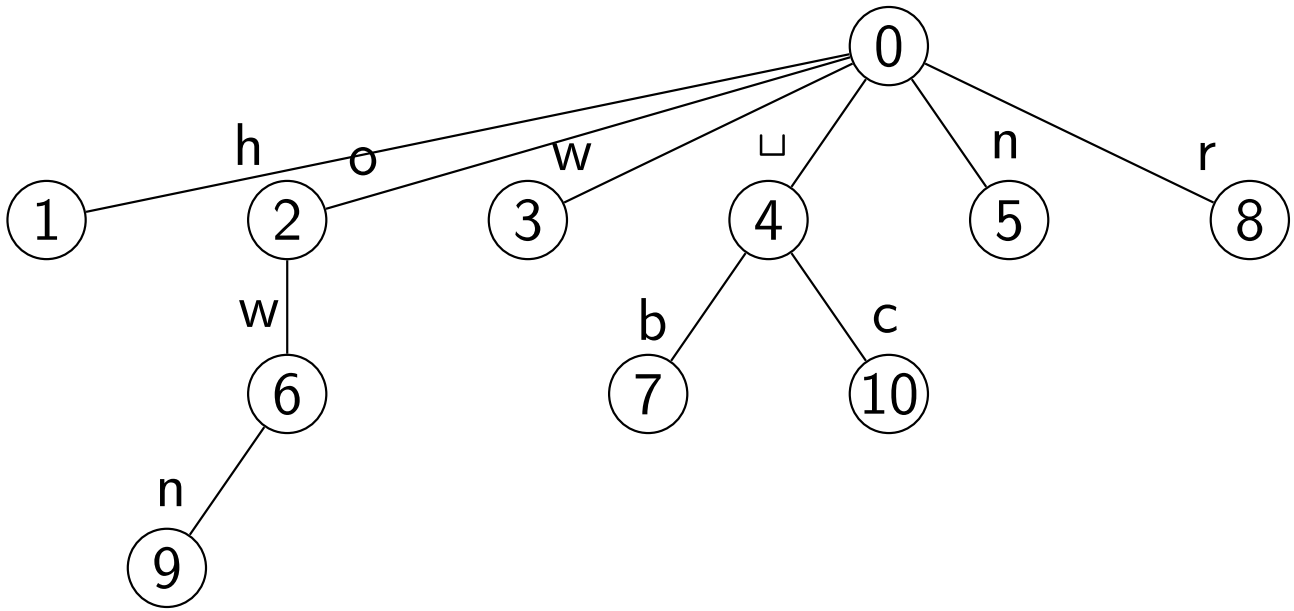
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n

LZ78 kodningsexempel

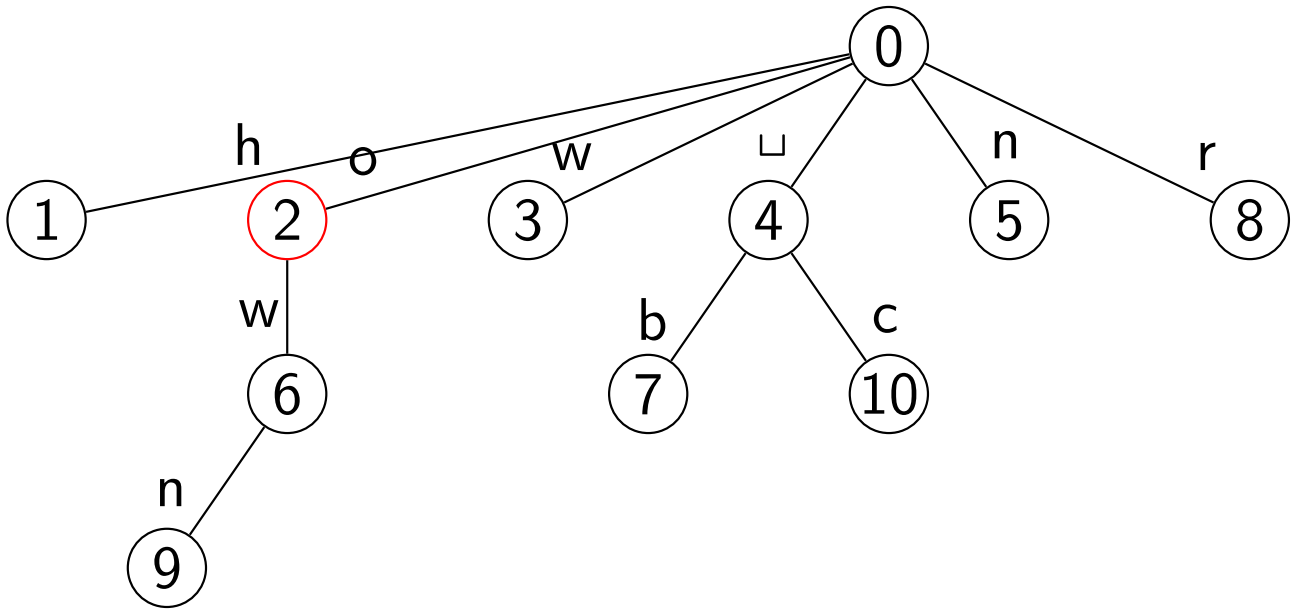
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c

LZ78 kodningsexempel

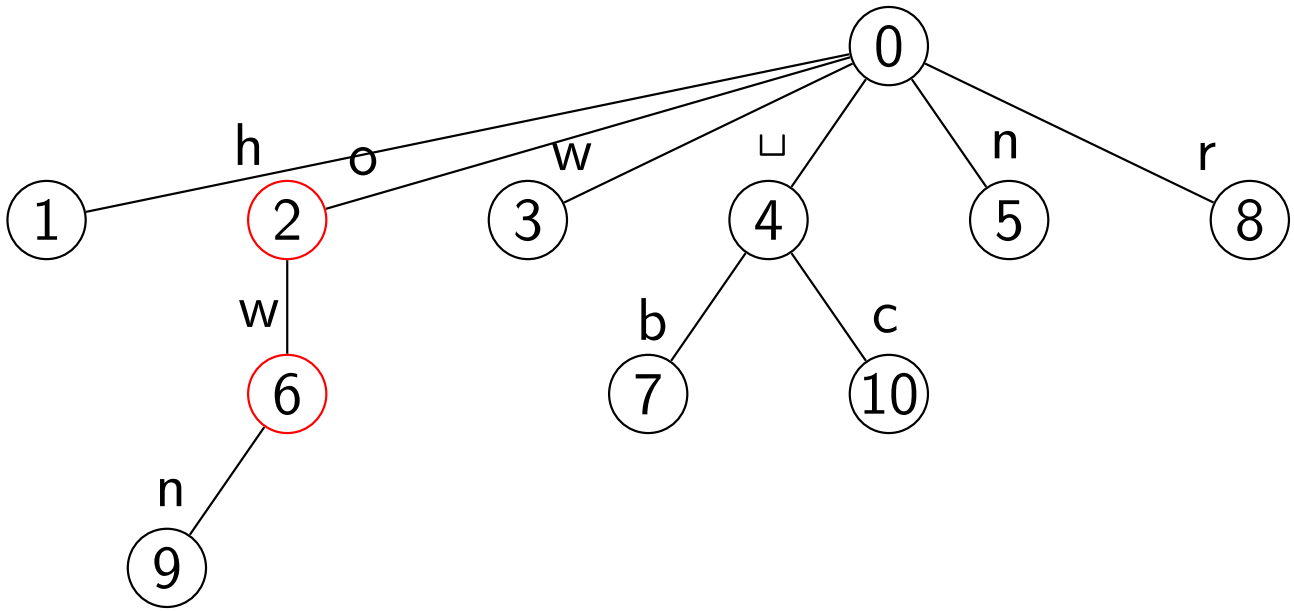
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c

LZ78 kodningsexempel

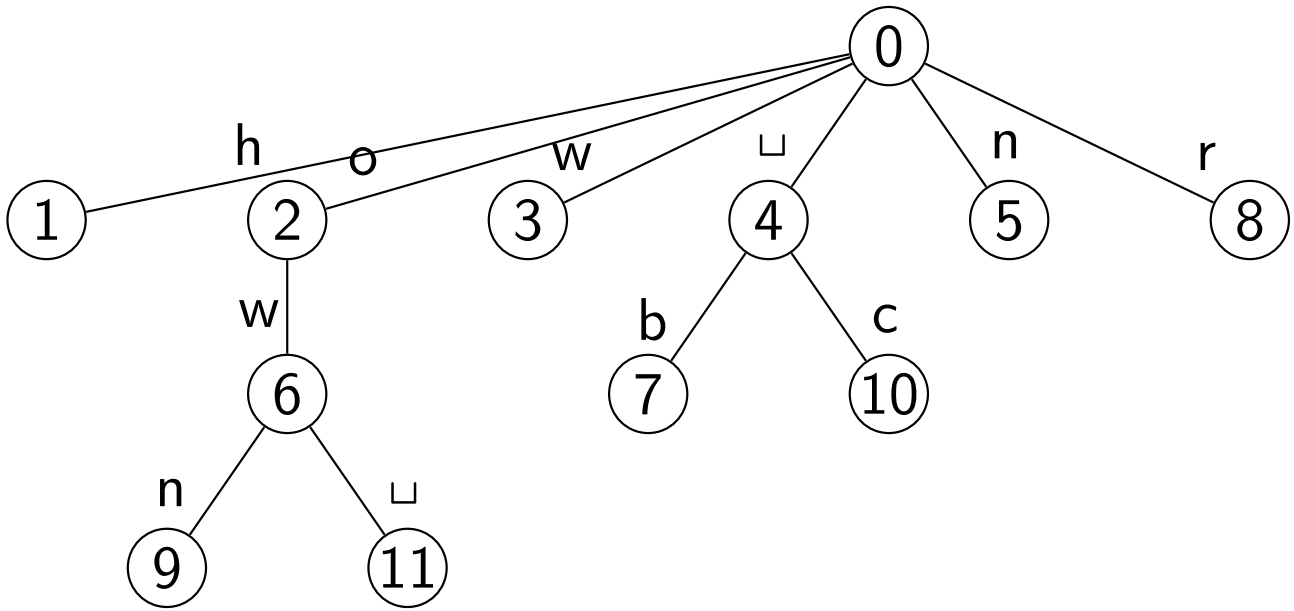
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c

LZ78 kodningsexempel

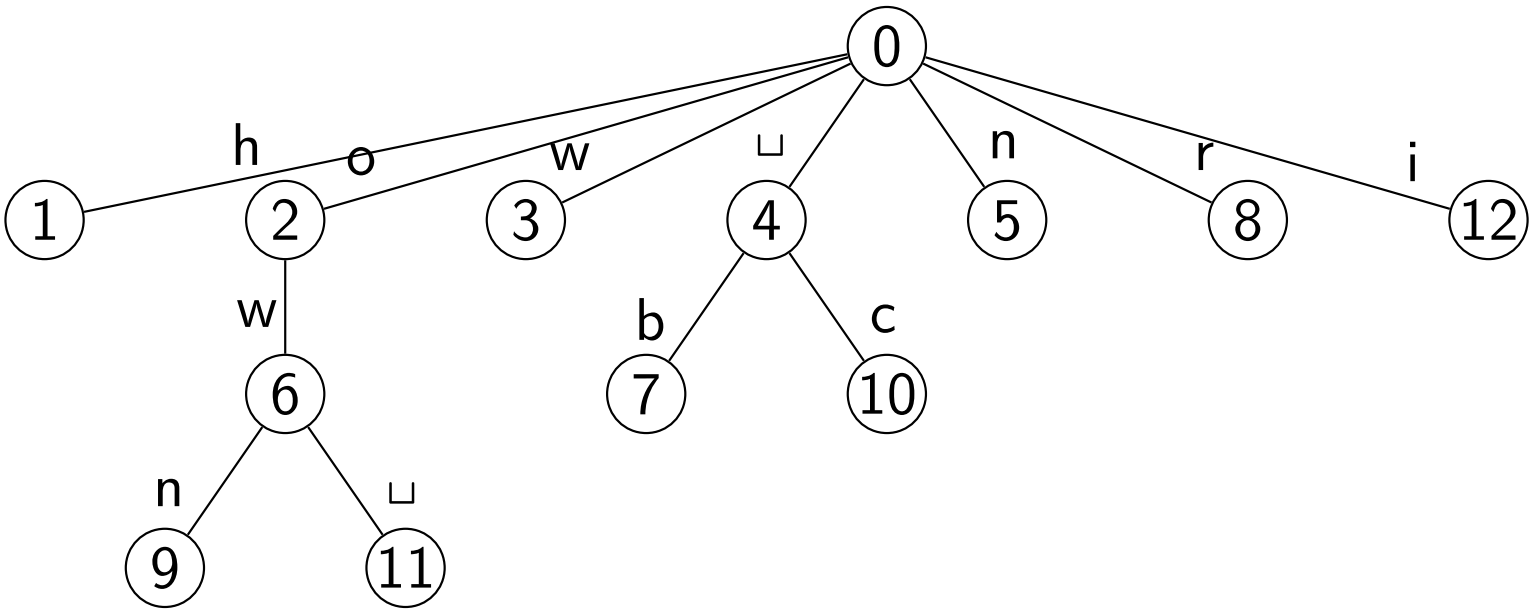
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c6␣

LZ78 kodningsexempel

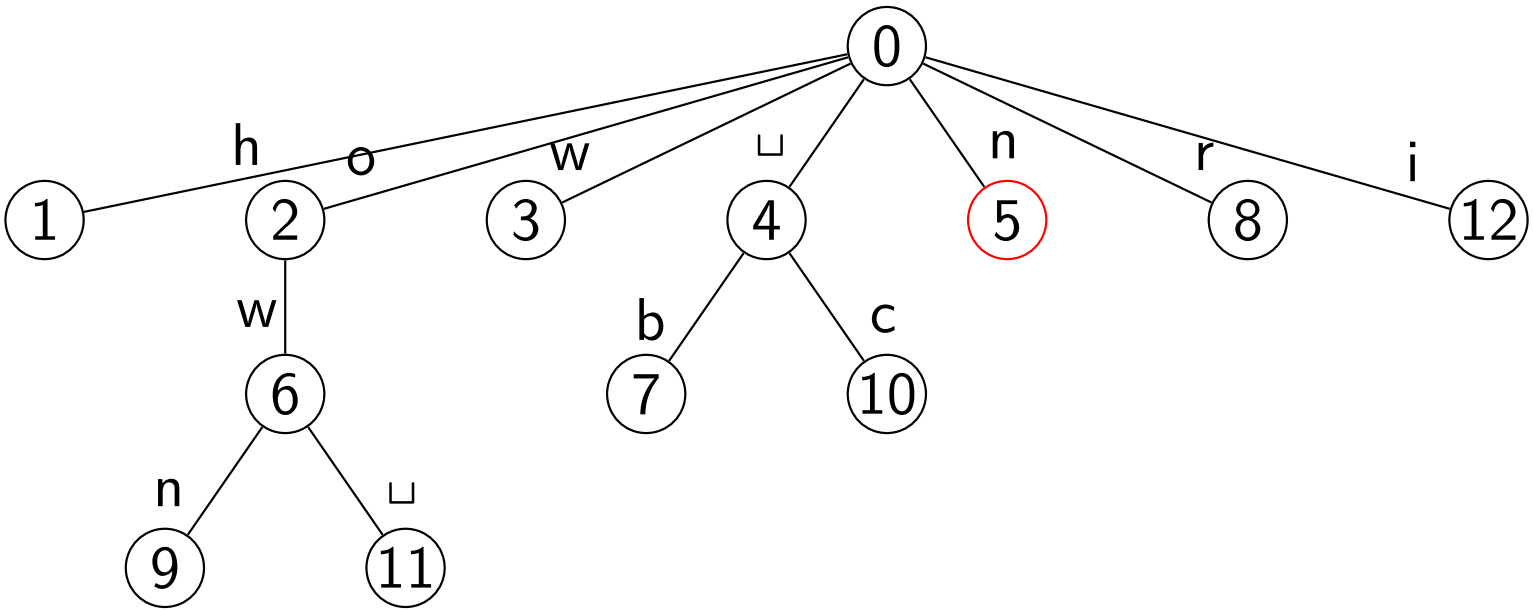
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c6␣0i

LZ78 kodningsexempel

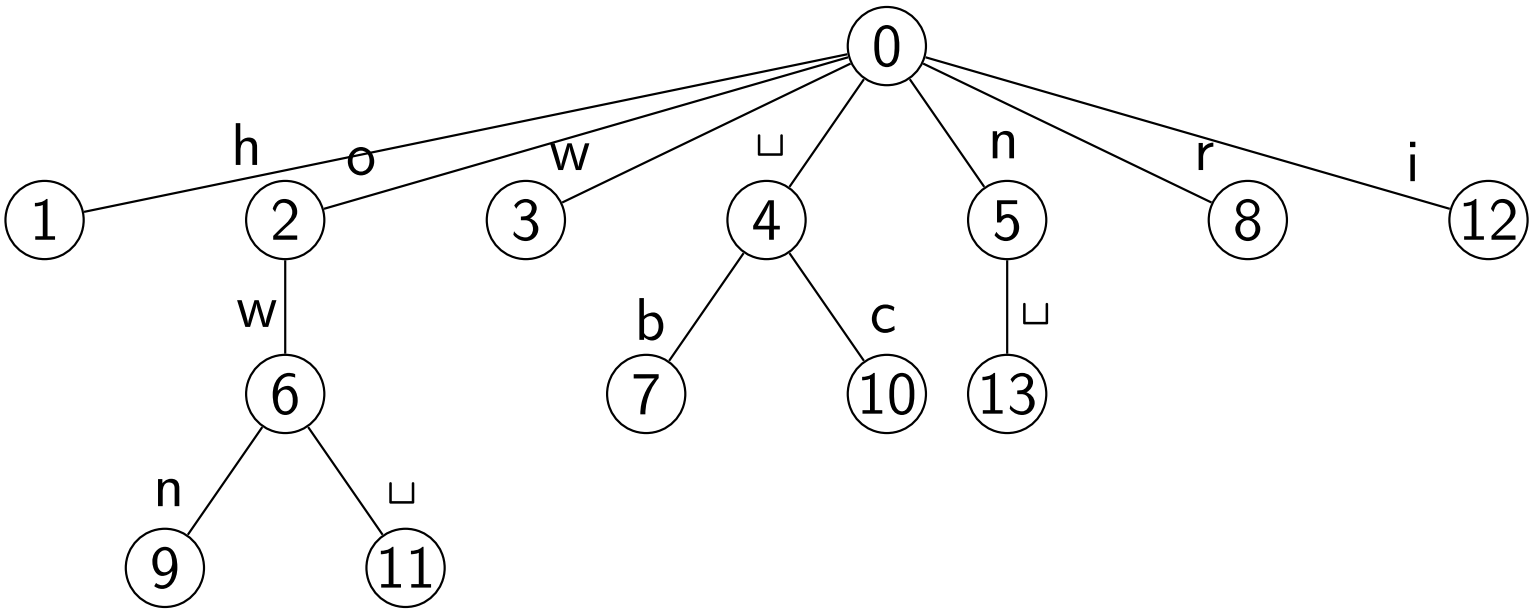
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c6␣0i

LZ78 kodningsexempel

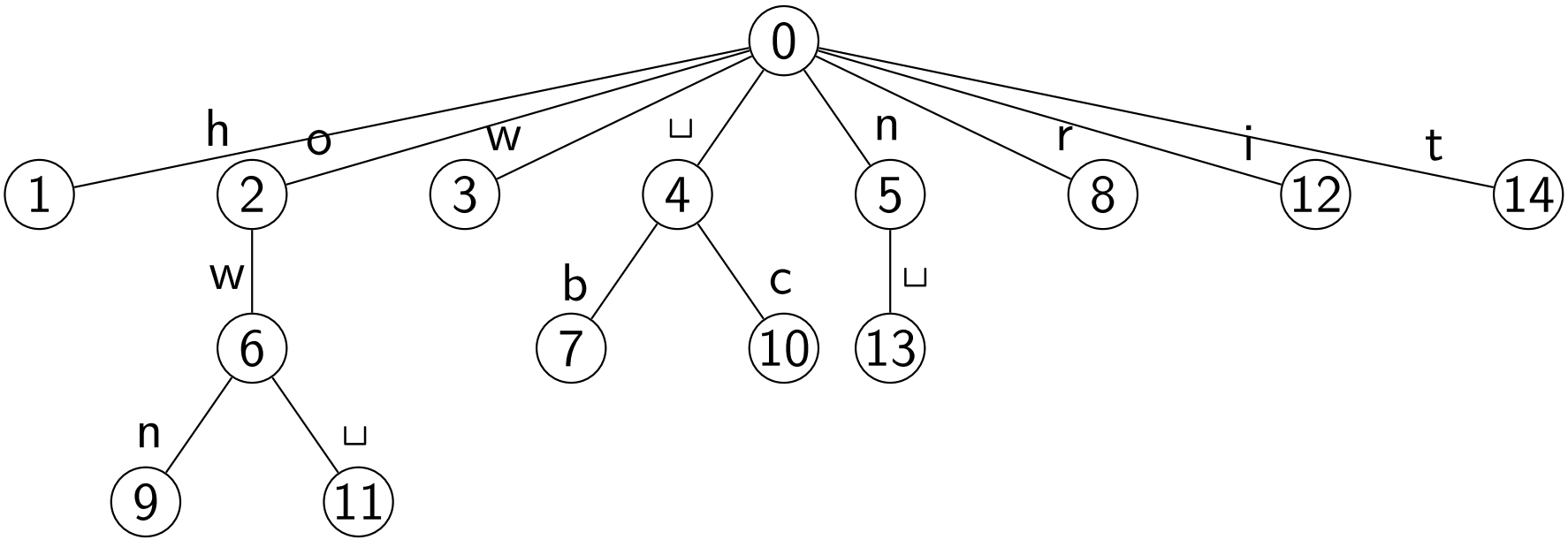
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c6␣0i5␣

LZ78 kodningsexempel

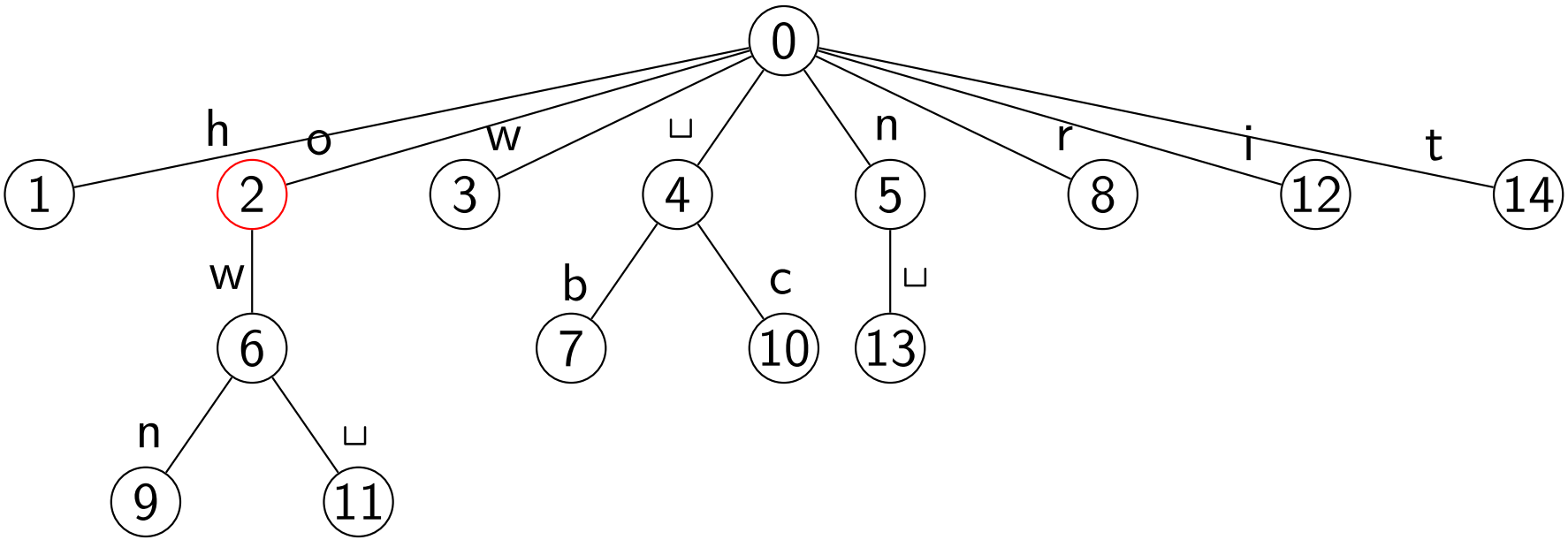
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t

LZ78 kodningsexempel

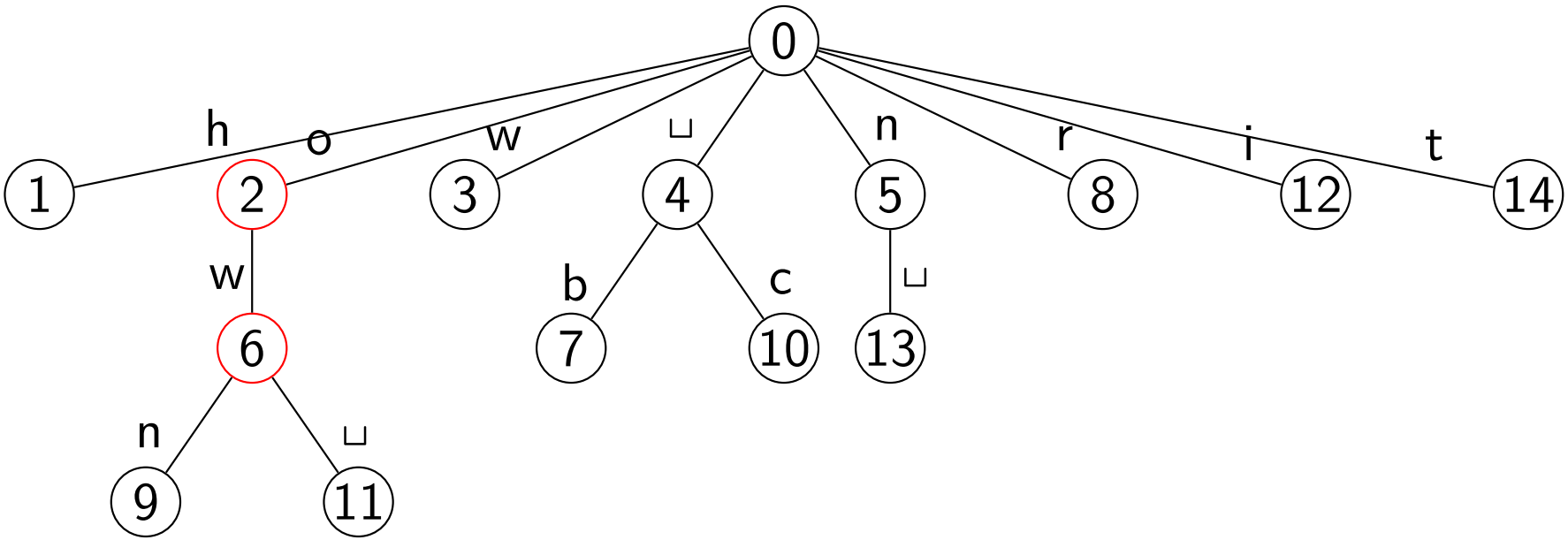
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t

LZ78 kodningsexempel

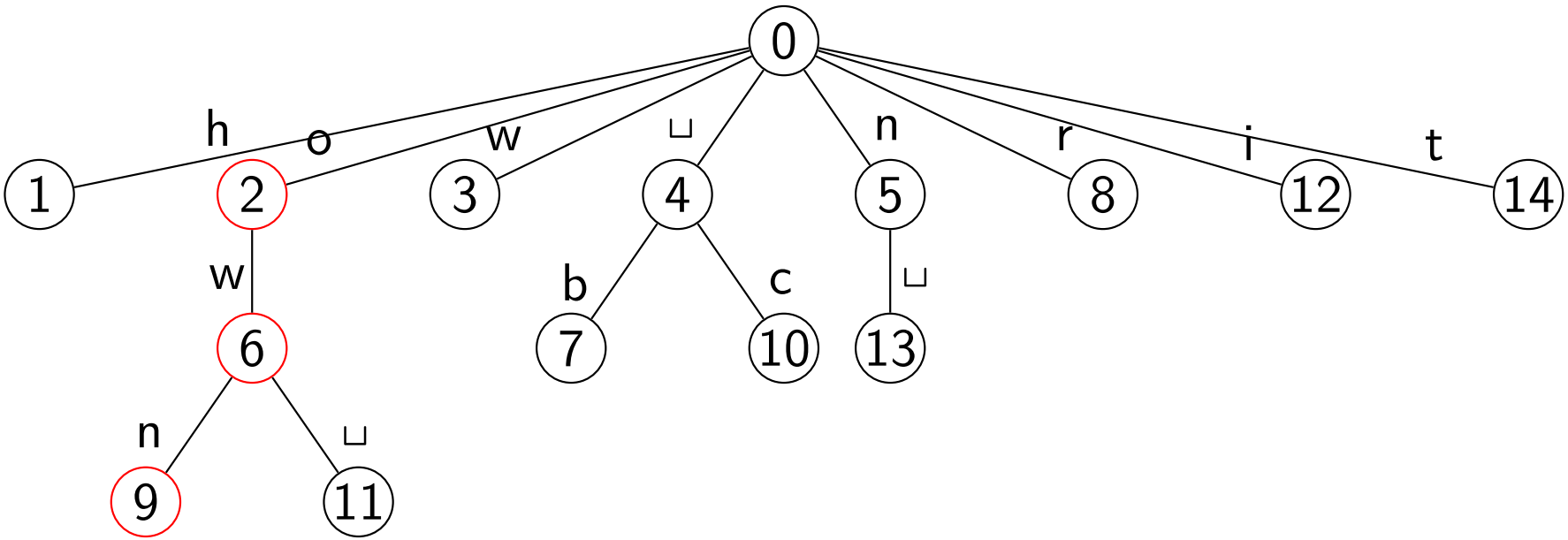
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t

LZ78 kodningsexempel

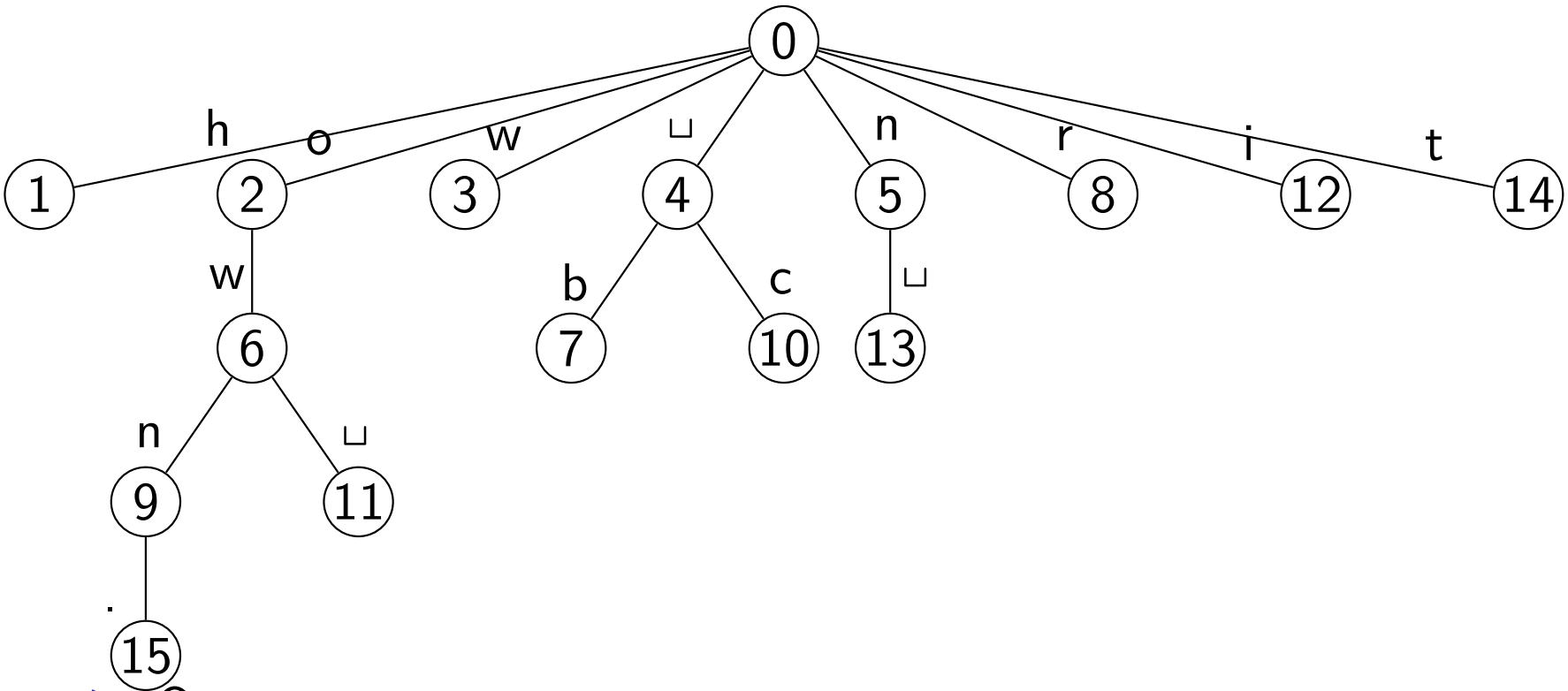
- Input: `how␣now␣brown␣cow␣in␣town.`



- Output:
  0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t

LZ78 kodningsexempel

- Input: `how␣now␣brown␣cow␣in␣town.`



- Output: 0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.

LZ78 avkodning:

- Table t=Empty(), $n = 0$.
- Läs in index i och tecken q från indata.
- Om $i = 0$,
  - print q
  - t=Insert($n + 1$,q, t)
  - $n = n + 1$

  annars
  - print Lookup(i, t)
  - print q
  - t=Insert($n + 1$, concat(Lookup(i, t), q), t)
  - $n = n + 1$

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

- Output:

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`
  - t(1)=h

- Output: h

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`
  - t(1)=h
  - t(2)=o

- Output: ho

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

  - t(1)=h
  - t(2)=o
  - t(3)=w

- Output: how

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`
  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣

- Output: how␣

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n

- Output: how␣n

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n
  - t(6)=ow

- Output: how␣now

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n
  - t(6)=ow
  - t(7)=␣b

- Output: how␣now␣b

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n
  - t(6)=ow
  - t(7)=␣b
  - t(8)=r

- Output: how␣now␣br

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`
  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n
  - t(6)=ow
  - t(7)=␣b
  - t(8)=r
  - t(9)=own
- Output: how␣now␣brown

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n
  - t(6)=ow
  - t(7)=␣b
  - t(8)=r
  - t(9)=own
  - t(10)=␣c
- Output: how␣now␣brown␣c

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n
  - t(6)=ow
  - t(7)=␣b
  - t(8)=r
  - t(9)=own
  - t(10)=␣c
  - t(11)=ow␣
- Output: how␣now␣brown␣cow␣

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n
  - t(6)=ow
  - t(7)=␣b
  - t(8)=r
  - t(9)=own
  - t(10)=␣c
  - t(11)=ow␣
  - t(12)=i
- Output: how␣now␣brown␣cow␣i

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n
  - t(6)=ow
  - t(7)=␣b
  - t(8)=r
  - t(9)=own
  - t(10)=␣c
  - t(11)=ow␣
  - t(12)=i
  - t(13)=n␣

- Output: how␣now␣brown␣cow␣in␣

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`

  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n
  - t(6)=ow
  - t(7)=␣b
  - t(8)=r
  - t(9)=own
  - t(10)=␣c
  - t(11)=ow␣
  - t(12)=i
  - t(13)=n␣
  - t(14)=t

- Output: how␣now␣brown␣cow␣in␣t

LZ78 avkodning, exempel

- Input: `0h0o0w0␣0n2w4b0r6n4c6␣0i5␣0t9.`
  - t(1)=h
  - t(2)=o
  - t(3)=w
  - t(4)=␣
  - t(5)=n
  - t(6)=ow
  - t(7)=␣b
  - t(8)=r
  - t(9)=own
  - t(10)=␣c
  - t(11)=ow␣
  - t(12)=i
  - t(13)=n␣
  - t(14)=t
  - t(15)=own.
- Output: how␣now␣brown␣cow␣in␣town.

Popularitet

- Lempel-Ziv-Welch (LZW) i GIF-bildformatet, unix-kommandot `compress`.
- DEFLATE-kompression (LZ77+Huffman) i `gzip` och PNG-formatet.
- Huffman-kodning i JPEG-bildformatet.

Tries för strängar, implementation

- Insättning
  - Starta i roten och gå nedåt i trädet så länge det finns en matchande väg.
  - När man hittar en skiljelinje, stanna och stoppa in resten av strängen som ett delträd.
- Borttagning
  - I princip samma algoritm som insättning fast “tvärtom”.
  - Sök upp strängen som ska tas bort och radera nerifrån i trädet upp till första förgreningen.