

## F12 - Sökträd, Trie

5DV149 Datastrukturer och algoritmer  
Kapitel 14.1–14.4

Niclas Börlin  
niclas.borlin@cs.umu.se

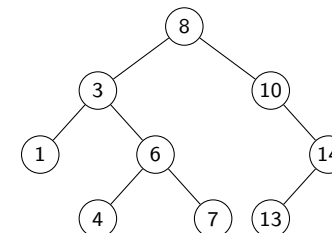
2024-02-19 Mån

- ▶ Binära sökträd
  - ▶ Specifikation
  - ▶ Exempel
  - ▶ Tillämpningar
- ▶ Trie
  - ▶ Organisation
  - ▶ Specifikation
  - ▶ Konstruktion
  - ▶ Tillämpningar
    - ▶ Komprimering
      - ▶ Huffman-kodning
      - ▶ LZ78-algoritmen

## Binära sökträd

### Binärt sökträd

- ▶ Används för **sökning** i linjära samlingar av dataobjekt, specifikt för att konstruera **tabeller** och **lexikon**
- ▶ För ett binärt träd, sorterat enligt en **sorteringsordning**  $R$  av etikett-typen, så gäller att för varje nod  $n$ :
  1.  $n$  har en **definierad etikett**,
  2. alla noder  $i$  i **vänster** delträd kommer **före**  $n$ , dvs.
    - ▶  $i.\text{label} R n.\text{label}$  är **sant**
  3.  $n$  kommer **före** alla noder  $j$  i **höger** delträd, dvs.
    - ▶  $j.\text{label} R n.\text{label}$  är **falskt**
- ▶ Exempel: Ett binärt sökträd för heltal med  $R = "<"$ :



- ▶ Skiljer sig från ett vanligt binärt träd:
  - ▶ Alla noder måste ha **etiketter**
  - ▶ Är **nedåtriktad**
  - ▶ Trädet är **sorterat**
    - ▶ Insättningar får inte förstöra **sorteringsordningen**
  - ▶ Man ska kunna ta bort **inre noder** också, inte bara löv:
    - ▶ När man tar bort en inre nod slits trädet **sönder**
    - ▶ Hur **lagar** man det?

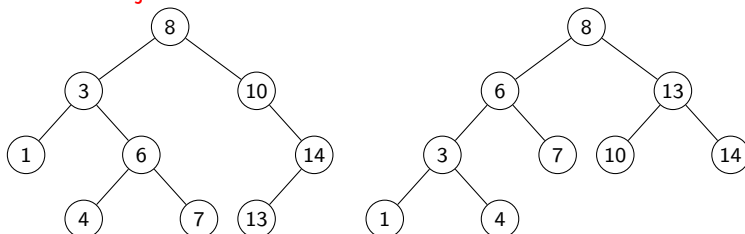
- ▶ Det går snabbt att **söka** i strukturen!
- ▶ Sökning efter värdet  $x$  i binärt sökträd:
  1. **Jämför**  $x$  med etiketten hos den aktuella noden  $n$ 
    - 1.1 Om lika har vi **hittat** det vi söker, **avsluta**
  2. annars om  $x < n.label$  är sant
    - 2.1 Sök rekursivt nedåt i **vänster** delträd
    - 2.2 Om vänster delträd **tomt** så finns det vi söker inte, **avsluta**
  3. annars ( $x > n.label$  är falskt)
    - 3.1 Sök rekursivt nedåt i **höger** delträd
    - 3.2 Om höger delträd **tomt** så finns det vi söker inte, **avsluta**

## Sökning

- ▶ Nedanstående träd har  $R = '<'$
- ▶ Sök efter 4!
- ▶ Sök efter 9!
- ▶ Sök efter 2!

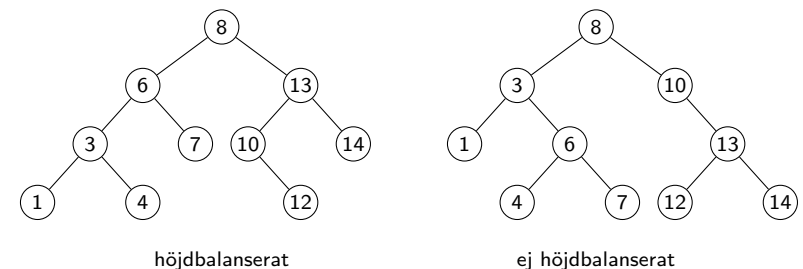


- ▶ Värsta fallskomplexitet  $O(\log n)$  om det binära trädet har **minimal höjd**



## Höjdbalanserat binärt sökträd

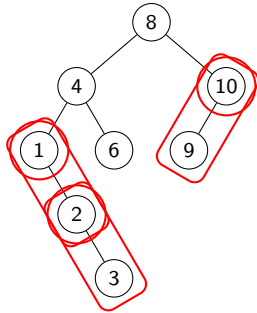
- ▶ Ett **höjdbalanserat** binärt sökträd (också kallat **AVL-träd**)<sup>1</sup>
  - ▶ **Skillnaden** mellan **höjden** av vänster och höger delträd är  $\leq 1$
  - ▶ Nästan **perfekt balans**
  - ▶ **Minimal** höjd



<sup>1</sup>[https://en.wikipedia.org/wiki/AVL\\_tree](https://en.wikipedia.org/wiki/AVL_tree)

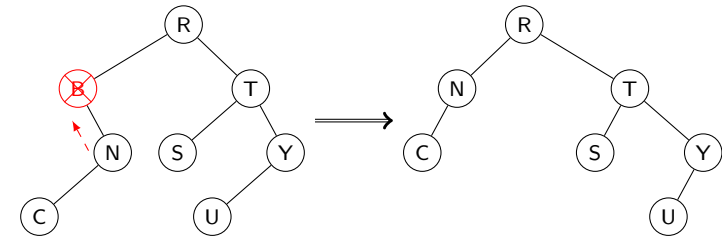
## Insättning i binärt sökträd

- ▶ Som sökning, men sätt in på platsen där vi skulle **fortsatt söka**
- ▶ Trädet kan bli **obalanserat**
- ▶ Går att konstruera  $O(\log n)$  algoritmer för insättning och borttagning som behåller höjdbalansen (ej denna kurs)
- ▶ Sätt in 2
- ▶ Sätt in 9
- ▶ Sätt in 3



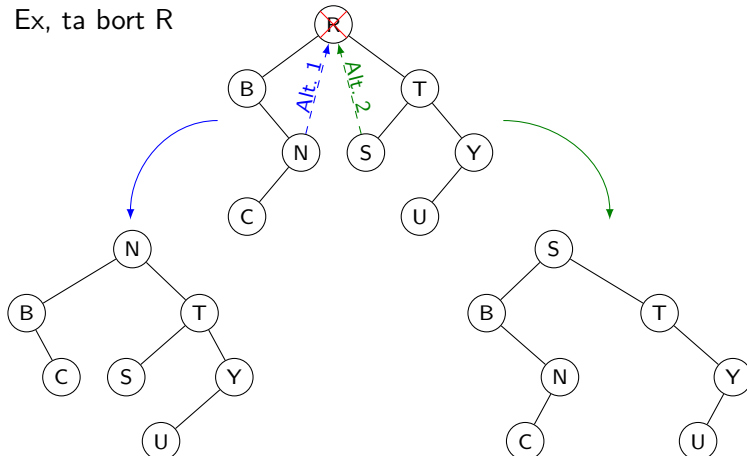
## Borttagning av nod i binärt sökträd (1)

- ▶ Borttagning av **löv** är triviale
- ▶ Borttagning av **inre nod** mer komplicerat
- ▶ Om den borttagna noden bara hade **ett delträd**:
  - ▶ **Lyft upp** det en nivå
- ▶ Ex, ta bort B



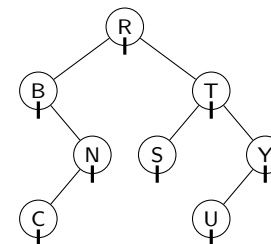
## Borttagning av nod i binärt sökträd (2)

- ▶ Om den borttagna noden hade **två** delträd:
  - ▶ Välj noden med **minsta** värdet i **höger** delträd som ersättning (alt. **största** värdet i **vänster** delträd)
- ▶ Ex, ta bort R



## 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



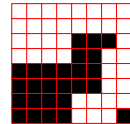
B	C	N	R	S	T	U	Y
---	---	---	---	---	---	---	---

- ▶ Ett binärt sökträd underlättar sökning i en **en-dimensionell** datamängd
- ▶ Lätt att **generalisera** till sökning i en 2-dimensionell datamängd (*quadtree*), 3-dimensionell (*octree*) eller högre

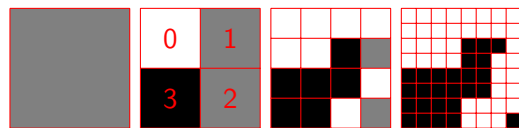
- ▶ Organiserat som ett "binärt" träd med **förgreningsfaktor 4**
- ▶ Tolkning (vanligast):
  - ▶ Rotnoden delar in den givna ytan (oftast kvadrat) i **fyra** lika stora kvadrater
  - ▶ **Vart och ett** av de fyra barnen delar i sin tur sin kvadrat i fyra osv.
  - ▶ Inga koordinater behöver lagras i inre noder
- ▶ Man kan använda det för att representera kurvor och ytor
  - ▶ Svarta kvadrater: fylls helt av objektet
  - ▶ Grå kvadrater: fylls delvis av objektet
  - ▶ Vita kvadrater: innehåller inte objektet

## Quadtree, exempel

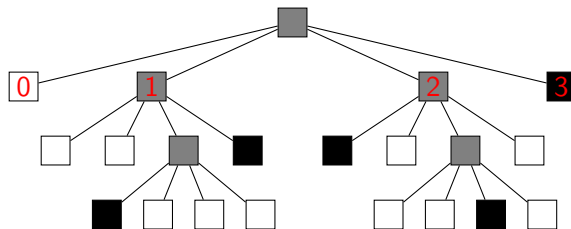
- ▶ Vi vill kunna söka om ett objekt täcker koordinat  $(i, j)$  i denna bild:



- ▶ Bygg upp ett quad-tree av bilden:

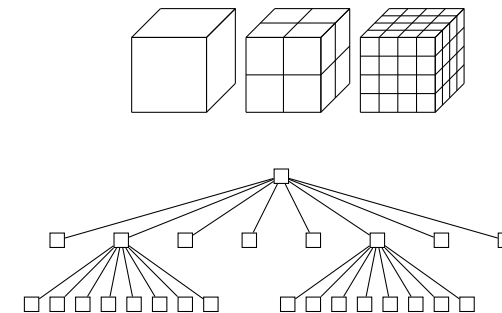


- ▶ Sök i trädet



## Octree

- ▶ Samma, fast med en **förgreningsfaktor på 8**



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

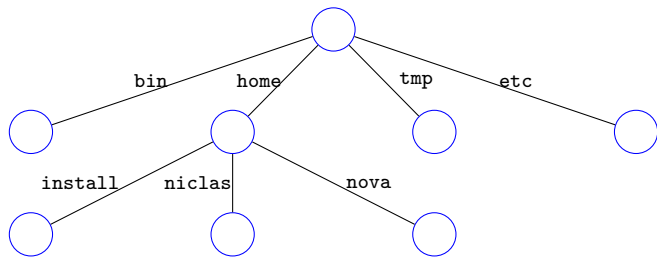
## Trie

## Trie

- ▶ Från *retrieve*, uttalas Traj
- ▶ Namngavs av Fredkin (1960)
- ▶ 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 ett Trie är **barnen** till en nod organiserade som **tabellvärden** i en tabell som hör till noden
- ▶ Trie kallas också för **diskrimineringssträd**, **code-link tree**, **radix-search tree**

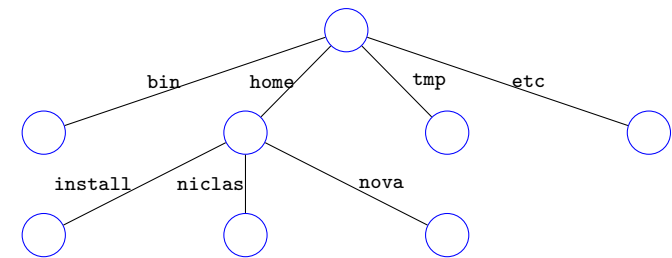
## Organisation av Trie (1)

- ▶ Man når barnen (delträden) genom "namn", dvs **argument/nycklar i nodens barntabell**
  - ▶ Ex. `Get-child(t: Tree, p: pos, name: key)`
- ▶ När man **ritar** Trie brukar nycklarna skrivas direkt intill **motsvarande bäge**



## Organisation av Trie (2)

- ▶ I en Trie har alla tabellerna **samma nyckeltyp**, till exempel tecken eller strängar
- ▶ I många tillämpningar av Trie saknar de inre noderna etiketter
  - ▶ Träden är **lövträd**
- ▶ Trie är normalt **nedåtriktade**



## Informell specifikation, två sätt

1. Utgå från **Urträdets** specifikation och låt typparametern **sibling** ha värdet **Tabell**
  - ▶ Insättning, borttagning och uppslagning hanteras av **Tabellen**
  - ▶ Navigering sker **utanför** Triet, t.ex.
    - ▶ `children = Trie-get-child-table(t, pos)`
    - ▶ `child = Table-lookup(children, name)`
  - ▶ I övrigt används de **vanliga operationerna** för att hantera etiketter, etc.
2. Sätt in lämpliga tabelloperationer direkt i **specifikationen** av Trie
  - ▶ Tabellen göms **inuti** Triet
  - ▶ `Insert-child` använder `Table-insert`
  - ▶ `Delete-child` använder `Table-remove`
  - ▶ `Child` använder `Table-lookup`

## Konstruktion av Trie

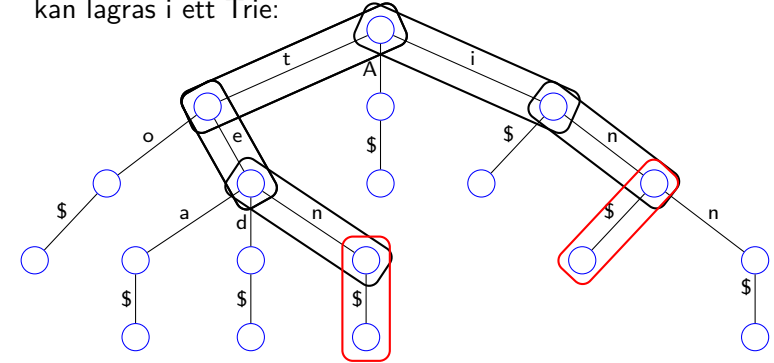
- ▶ De flesta Träd-konstruktioner går bra att utgå från
- ▶ Man måste byta ut de delar som hanterar barnen till att hantera dessa som **tabellvärden** i en Tabell
- ▶ Implementerar man tabellen som en **vektor** eller som en **hashtabell** får man **effektiva** Trie-implementationer (sökning blir  **$O(1)$** )

## Tillämpningar av Trie (1)

- ▶ Används för att konstruera Lexikon 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
  - ▶ Ska vi lagra **textsträngar** (sekvenser av tecken) så blir tabellnycklarna i Triet av typen **tecken**
  - ▶ En **sekvens** motsvaras 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
    - ▶ Ofta används dollar-tecknet (\$) som slutmarkör

## Tillämpningar av Trie: Lexikon

- ▶ Exempel: Ett **Lexikon** som innehåller följande strängar:
  - ▶ A, i, in, inn, tea, ted, ten, to
 kan lagras i ett Trie:



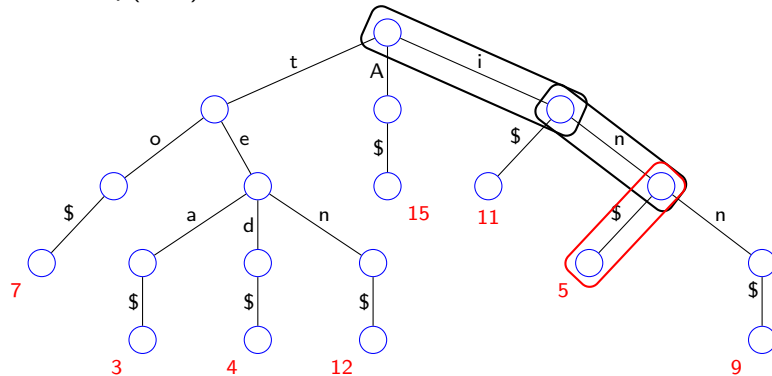
- ▶ Varje godkänd sekvens i **Lexikonet** motsvaras av att sekvensen ingår i Triet och slutar i ett **löv**
- ▶ Exempelvis så ingår sekvenserna **ten** och **in**
- ▶ Sekvensen **te** ingår inte

## Tillämpningar av Trie: Tabell (1)

- ▶ Säg att vi vill koppla **värden** till sekvenserna, dvs. skapa följande **Tabell**

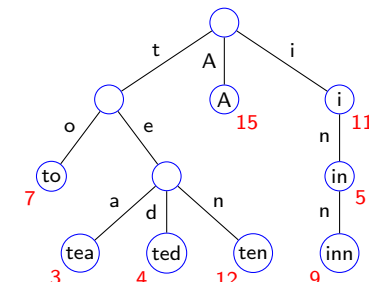
Nyckel	A	i	in	inn	tea	ted	ten	to
Värde	15	11	5	9	3	4	12	7

- ▶ Då associerar vi tabellvärdena med **lövnoderna**
- ▶ Lookup("in") skulle returnera värdet 5



## Tillämpningar av Trie: Tabell (2)

- ▶ En annan variant är att ha definierade **etiketter** för alla godkända noder (löv **och** inre noder)



- ▶ 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**?
    - ▶  $O(n)$
  - ▶ Hur ökar tidsåtgången med **antalet ord**  $n$  i lexikonet?
    - ▶  $O(n)$
- ▶ Om vi konstruerar lexikonet som ett **Trie**:
  - ▶ Vad är tidskomplexiteten för **en sökning**?
    - ▶  $O(s)$ , där  $s$  är **längden** på sekvensen
  - ▶ Hur ökar tidsåtgången med **antalet ord**  $n$  i lexikonet?
    - ▶ Det gör den inte!

- ▶ Om vi vill lagra **sekvenser** som startar med samma följd av elementvärden i ett **Lexikon/Tabell** så finns det flera **fördelar** med att använda ett **Trie**:
  - ▶ **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**
    - ▶ Inte "dyrare" att söka i ett **stort** Lexikon jämfört med ett **litet**!

## 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**

## Tillämpningar av Trie

- ▶ Stavningskontroll:
  - ▶ Skapa ett Trie med **alla ord** som finns i språket
- ▶ Autocomplete
  - ▶ Expandera till nästa förgrening eller löv
- ▶ Översättningstabell:
  - ▶ Löven innehåller **motsvarande ord** i ett annat språk
- ▶ Internet routing
- ▶ Datakomprimering:
  - ▶ Huffman-kodning (Huffman, 1952) <sup>2</sup>
  - ▶ LZ78-algoritmen (Lempel, Zip, 1978) <sup>3</sup>

<sup>2</sup>[https://en.wikipedia.org/wiki/Huffman\\_coding](https://en.wikipedia.org/wiki/Huffman_coding)

<sup>3</sup>[https://en.wikipedia.org/wiki/LZ77\\_and\\_LZ78](https://en.wikipedia.org/wiki/LZ77_and_LZ78)



## Kompressionsalgoritmer

## Fixlängdskodning

### Filkomprimering, fixlängdskodning

- ▶ En vanlig textfil i ASCII-format lagrar en sekvens av tecken 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

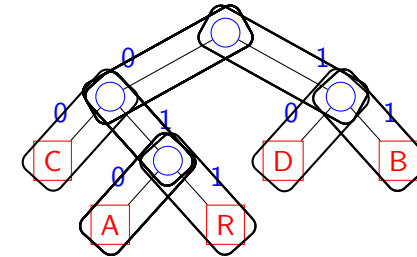
- ▶ I en typisk textfil förekommer vissa bokstäver **oftare** än andra
- ▶ Om man lagrar vanligt förekommande bokstäver med **färre bitar** än ovanliga så skulle man kunna spara utrymme
  - ▶ Morse-alfabetet ett tidigt exempel:
    - ▶ A = .-
    - ▶ E = .
    - ▶ I = ..
    - ▶ O = ----
    - ▶ Q =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

## Filkomprimering, prefixregeln

- ▶ Kodningen måste ske så att man enkelt kan avkoda strängen **entydigt**
- ▶ Motexempel:
  - ▶ 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 får kodas med en sträng som utgör ett **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

- ▶ Vi kan använda ett Trie!
  - ▶ Bokstäverna lagras i löven
  - ▶ Den vänstra kanten betyder 0
  - ▶ Den högra kanten betyder 1

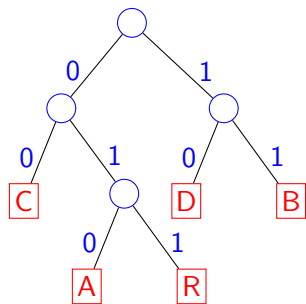


- ▶ A = 010
- ▶ B = 11
- ▶ C = 00
- ▶ D = 10
- ▶ R = 011

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

## Optimal kompression

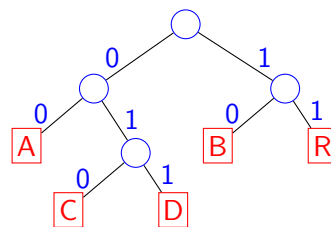
- ▶ Vilken tabell/träd vi har bestämmer kompressionens **effektivitet**
- ▶ Med tabellen/trädet nedan får vi  
01011011010000101001011011010 = 29 bitar



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

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

- ▶ Med tabellen/trädet till höger får vi  
001011000100001100101100 = 24 bitar
- ▶ Varför?
- ▶ ABRACADABRA = AAAAA BB RR C D

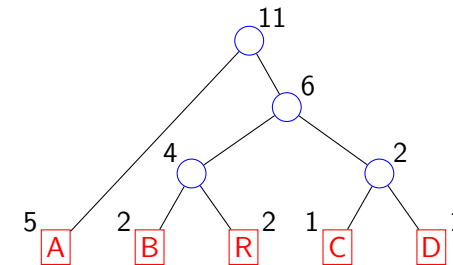


## Huffman-kodning

## Huffman-kodning

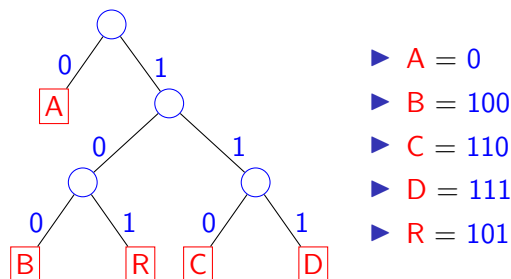
- ▶ Bygger upp **optimalt** träd från en **frekvenstabell**
- ▶ Algoritm:
  - ▶ Börja med en **mängd träd**, där varje träd består av **ett enda löv**
  - ▶ Till varje löv associeras en **symbol** och en **vikt**
    - ▶ Vikten är symbolens **frekvens** i texten som ska kodas
  - ▶ Upprepa tills vi har **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 rotnod
    - ▶ Den nya rotens vikt = **summan** av barnens vikter

## Huffman-kodning, exempel



## Huffman-kodning, optimalt exempel

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



## Huffman-kompression

- ▶ **Kompression** (kodning) med Huffman-algoritmen tar en **sträng** som indata
  - ▶ Algoritmen beräknar en frekvenstabell och därefter ett optimalt Trie/tabell
  - ▶ Utdata är **kodtabellen** följt av en **kodsträng** bestående av en sekvens av nollor och ettor
- ▶ **Dekompression** (dekodning) med Huffman-algoritmen tar kodtabellen och kodsträngen som indata
  - ▶ Dekompressionsalgoritmen bygger upp ett Trie från kodtabellen och använder Trie:t till att avkoda kodsträngen
  - ▶ Utdata är den ursprungliga **strängen**

## LZ-kodning (Lempel-Ziv-kodning)

zip, gzip, png, ...

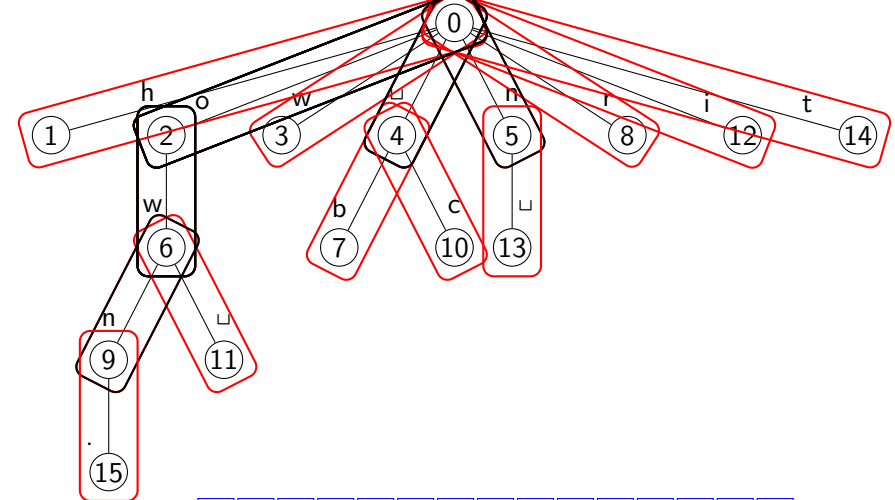
- ▶ Algoritmen för **LZ-78-kodning** tar en **sträng** som indata och levererar en **sekvens av par** som utdata
  - ▶ Varje par består av ett **index** och ett **tecken**

## LZ78 eller Lempel-Ziv-kodning, kodning

- ▶ Låt frasen 0 vara den **tomma strängen**
- ▶ Starta med ett Trie med en rotnod med nummer 0
- ▶ Skanna igenom texten **ett tecken i taget**
  1. Om du stöter på en ny, **okänd**, bokstav **c**:
    - ▶ Lägg till **c** på **toppnivån** på Triet
    - ▶ Lägg paret (0, c) sist i den **kodade strängen**
  2. Om du stöter på en gammal, **känd**, bokstav:
    - ▶ **Gå nedåt** i Triet så länge du kan matcha **nästa tecken**
    - ▶ Till slut har du nått en nod **n** med ett tecken **c** som **inte går att matcha**
    - ▶ **Lägg till** en ny nod till Triet som representerar den nya strängen
    - ▶ Lägg paret (n, c) sist i den **kodade strängen**
- ▶ Den kodade strängen är oftast mycket **kortare** än originalet
- ▶ Vi talar om att vi **komprimerat** strängen (eller filen)
- ▶ Exempelsträng: "how now brown cow in town."

## LZ78 kodningsexempel

Startsträng: h o w \_ n o w \_ b r o w n \_ c o w \_ i n \_ t o w n .  
Fraser: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



Kodad sträng: 0h 0o 0w 0\_ 0n 2w 4b 0r 6n 4c 6\_ 0i 5\_ 0t 9.

## LZ78 avkodning

- ▶ Algoritmen för **LZ-78-avkodning** tar en sekvens av (index, tecken)-par som indata och levererar en sträng som utdata
  - ▶ Exemplet **skriver ut** strängen

## LZ78 avkodning:

```
Algorithm LZ78-decode(s: Codestring)

t ← Table-empty()
n ← 0
while not Iseempty(s) do
  (ix, ch, s) ← Get-and-remove-first-index-and-char(s)

  if ix = 0 then // Sequence has empty prefix
    prefix ← ""
  else // Lookup prefix in the decoding table
    prefix ← Table-lookup(ix, t)

  // The new sequence is the prefix followed by the new char
  seq ← concat(prefix, ch)

  // Output the new sequence
  print(seq)

  // Insert new sequence into the table
  t ← Table-insert(n + 1, seq, t)
  n ← n + 1
```

## LZ78 avkodning, exempel

Input: 0h 0o 0w 0\_ 0n 2w 4b 0r 6n 4c 6\_ 0i 5\_ 0t 9.

Output: h o w \_ n o w \_ b r o w n \_ c o w \_ i n \_ t o w n .

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.

## LZW (Lempel-Ziv-Welch)

- ▶ LZW-algoritmen (Welch, 1984) är en populär förbättring av LZ78-algoritmen
  - ▶ Initierar tabellen till att innehålla alla möjliga sekvenser av längd ett

- ▶ Mediaformat
  - ▶ GIF (LZW)
  - ▶ PNG (LZ78 + Huffman)
  - ▶ JPEG (Huffman)
  - ▶ MP3 (Huffman)
- ▶ Filkompressionsalgoritmer
  - ▶ DEFLATE-algoritmen (LZ78 + Huffman) i ZIP