UNIT - V

Pattern Matching and Tries:

Pattern matching algorithms-Brute force, the Boyer –Moore algorithm, the Knuth-Morris-Pratt algorithm, Standard Tries, Compressed Tries, Suffix tries.

String Searching

- The previous slide is not a great example of what ismeant by “String Searching.” Nor is it meant to ridicule people without eyes....
- The object of string searching is to find the locationof a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.).
- As with most algorithms, the main considerationsfor string searching are speed and efficiency.
- There are a number of string searching algorithms inexistence today, but the two we shall review are Brute Force and Rabin-Karp.

Brute Force

- The Brute Force algorithm compares the pattern tothe text, one character at a time, until unmatching characters are found:

*T*WO ROADS DIVERGED IN A YELLOW WOOD
*R*OADS

T*W*O ROADS DIVERGED IN A YELLOW WOOD
 *R*OADS

TW*O* ROADS DIVERGED IN A YELLOW WOOD
  *R*OADS

TWO ROADS DIVERGED IN A YELLOW WOOD
   *R*OADS

TWO ***ROADS*** DIVERGED IN A YELLOW WOOD
    ROADS

- Compared characters are italicized.
- Correct matches are in boldface type.

- The algorithm can be designed to stop on either thefirst occurrence of the pattern, or upon reaching the end of the text.

Brute Force Pseudo-Code

- Here's the pseudo-code

```
do
    if (text letter == pattern letter) compare next letter of pattern to next
        letter of text
    else
        move pattern down text by one letter
while (entire pattern found or end of text)
```

```
tetththeheehthtehtheththehehtht the
tetththeheehthtehtheththehehtht
 the tetththeheehthtehtheththehehtht
  the tetththeheehthtehtheththehehtht

   the

tetththeheehthtehtheththehehtht
    the tetththeheehthtehtheththehehtht

     the
```

Brute Force-Complexity

- Given a pattern M characters in length, and a text Ncharacters in length...

- **Worst case**: compares pattern to each substring oftext of length M.For example, M=5.

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAAH*AAAAH* **5 comparisons made**

2) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAAH*AAAAH* **5 comparisons made**

3) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAAH*AAAAH* **5 comparisons made**

4) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAAH*AAAAH* **5 comparisons made**

5) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAAH*AAAAH* **5 comparisons made**

....

N) AAAAAAAAAAAAAAAAAAAAAAA*AAAAH*

5 comparisons made *AAAAH*

- Total number of comparisons: M (N-M+1)
- Worst case time complexity: O(MN)

Brute Force-Complexity(cont.)

- Given a pattern M characters in length, and a text Ncharacters in length...
- **Best case if pattern found**: Finds pattern in first Mpositions of text. For example, M=5.

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAAH*AAAAA* **5 comparisons made**

- Total number of comparisons: M
- Best case time complexity: O(M)

Brute Force-Complexity(cont.)

- Given a pattern M characters in length, and a text Ncharacters in length...
- **Best case if pattern not found**: Always mismatchon first character. For example, M=5.

1) *A*AAAAAAAAAAAAAAAAAAAAAAAAAAAAH*O*OOOH **1 comparison made**

2) A*A*AAAAAAAAAAAAAAAAAAAAAAAAAAAH

*O*OOOH **1 comparison made**

3) AA*A*AAAAAAAAAAAAAAAAAAAAAAAAAAH

*O*OOOH **1 comparison made**

4) AAA*A*AAAAAAAAAAAAAAAAAAAAAAAAAH

*O*OOOH **1 comparison made**

5) AAAA*A*AAAAAAAAAAAAAAAAAAAAAAAAH

*O*OOOH **1 comparison made**

...

N) AAAAAAAAAAAAAAAAAAAAAAAAAAAH

1 comparison made OOOOH

- Total number of comparisons: N
- Best case time complexity: O(N)
- algorithm will do aBrute Force comparison between the pattern and theM-character sequence.
- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.
- Perhaps a figure will clarify some things... The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for eachM-character subsequence of text to be compared.
- If the hash values are unequal, the algorithm willcalculate the hash value for next M-character sequence.
- If the hash values are equal, the

  100=100

- shing is small.

The Knuth-Morris-PrattAlgorithm

- The Knuth-Morris-Pratt (KMP) string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previouscomparisons.
- A failure function (*f*) is computed that indicates howmuch of the last comparison can be reused if it fais.
- Specifically, *f* is defined to be the longest prefix ofthe pattern P[0,..,j] that is also a suffix of P[1,..,j]
  - Note: **not** a suffix of P[0,..,j]
- Example:
  - value of the KMP failure function:

j	0	1	2	3	4	5
$P[j]$	a	b	a	b	a	c
$f(j)$	0	0	1	2	3	0

- This shows how much of the beginning of the stringmatches up to the portion immediately preceding a failed comparison.
  - if the comparison fails at (4), we know the a,b inpositions 2,3 is identical to positions 0,1

The KMP Algorithm (contd.)

- Time Complexity Analysis
- define $k = i - j$
- In every iteration through the while loop, one ofthree things happens.
  - 1) if $T[i] = P[j]$, then i increases by 1, as does jk remains the same.
  - 2) if $T[i] != P[j]$ and $j > 0$, then i does not change and k increases by at least 1, since k changesfrom $i - j$ to $i - f(j\text{-}1)$
  - 3) if $T[i] != P[j]$ and $j = 0$, then i increases by 1 and

    k increases by 1 since j remains the same.
- Thus, each time through the loop, either i or $k$increases by at least 1, so the greatest possible number of loops is $2n$
- This of course assumes that f has already beencomputed.
- However, f is computed in much the same manner asKMPMatch so the time complexity argument is analogous. KMPFailureFunction is $\boldsymbol{O}(m)$
- Total Time Complexity: $\boldsymbol{O}(n + m)$

The KMP Algorithm (contd.)

- the KMP string matching algorithm: Pseudo-Code

Algorithm KMPMatch(T,P)
Input: Strings T (text) with n characters and P (pattern) with m characters.
Output: Starting index of the first substring of Tmatching P, or an indication that P is not a

substring of T.

$f \leftarrow$ KMPFailureFunction(P) {build failure function}

$i \leftarrow 0$

$j \leftarrow 0$

while $i < n$ do

if $P[j] = T[i]$ thenif $j = m - 1$ then

return $i - m - 1$ {a match}

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else if $j > 0$ then {no match, but we have advanced}

$j \leftarrow f(j\text{-}1)$ {j indexes just after matching prefix in P}

else

$i \leftarrow i + 1$

return "There is no substring of T matching P"

The KMP Algorithm (contd.)

- The KMP failure function: Pseudo-Code

Algorithm KMPFailureFunction(P);

Input: String P (pattern) with m characters

Ouput: The faliure function f for P, which maps j tothe length of the longest prefix of P that is a suffixof $P[1,..,j]$

$i \leftarrow 1$

$j \leftarrow 0$

while $i \leq m\text{-}1$ do

if $P[j] = T[j]$ then

{we have matched $j + 1$ characters}

$f(i) \leftarrow j + 1$

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else if $j > 0$ then

{j indexes just after a prefix of P that matches}

$j \leftarrow f(j\text{-}1)$else

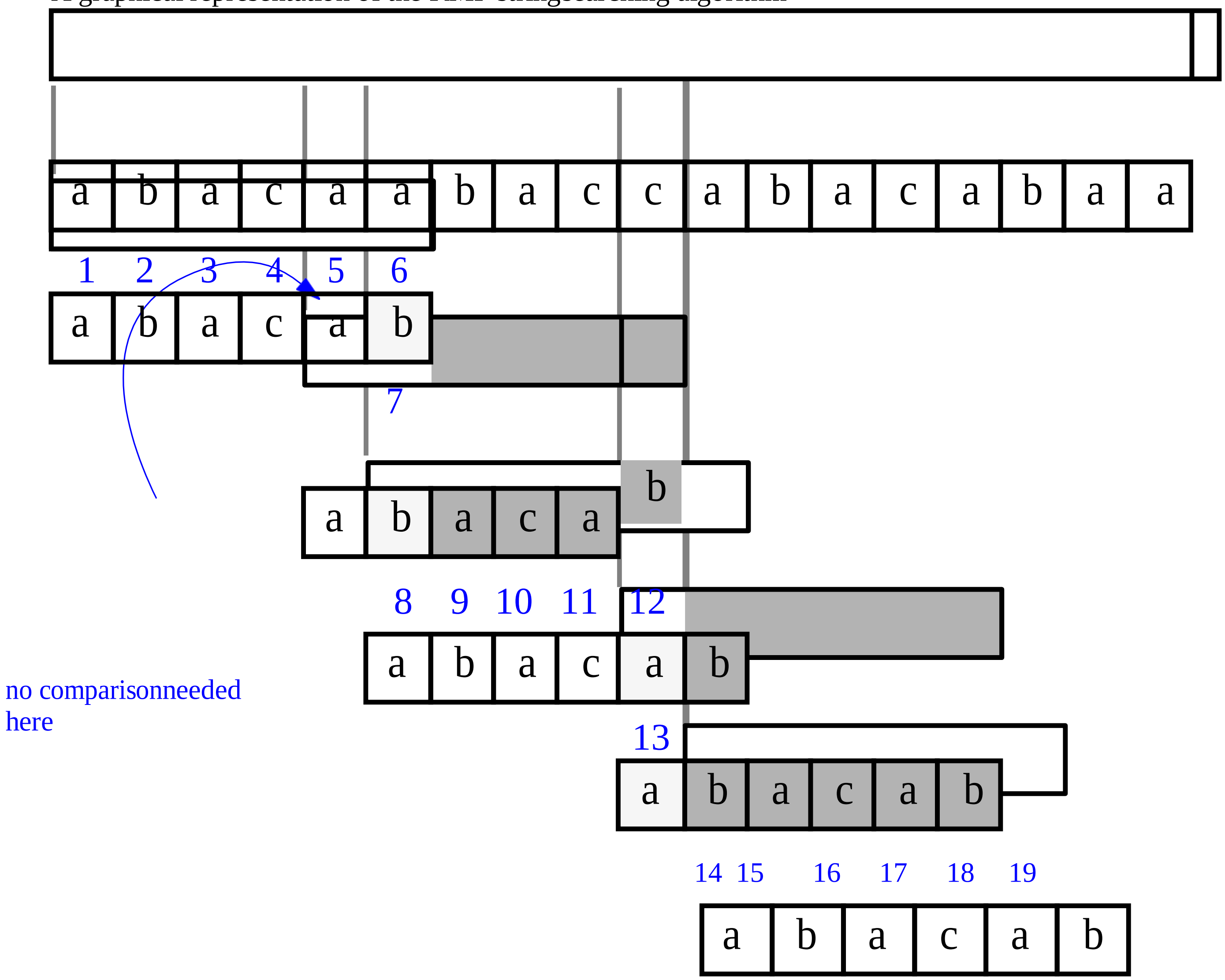
{there is no match}

$f(i) \leftarrow 0$

$i \leftarrow i + 1$

The KMP Algorithm (contd.)

- A graphical representation of the KMP stringsearching algorithm



Tries

- A trie is a tree-based date structure for storing strings in order to make pattern matching faster.

- Tries can be used to perform **prefix queries** for information retrieval. Prefix queries search for the longest prefix of a given string X that matches a prefix of some string in the trie.

- A trie supports the following operations on a set S ofstrings:

insert(X): Insert the string X into S

Input: String **Ouput**: None

remove(X): Remove string X from S

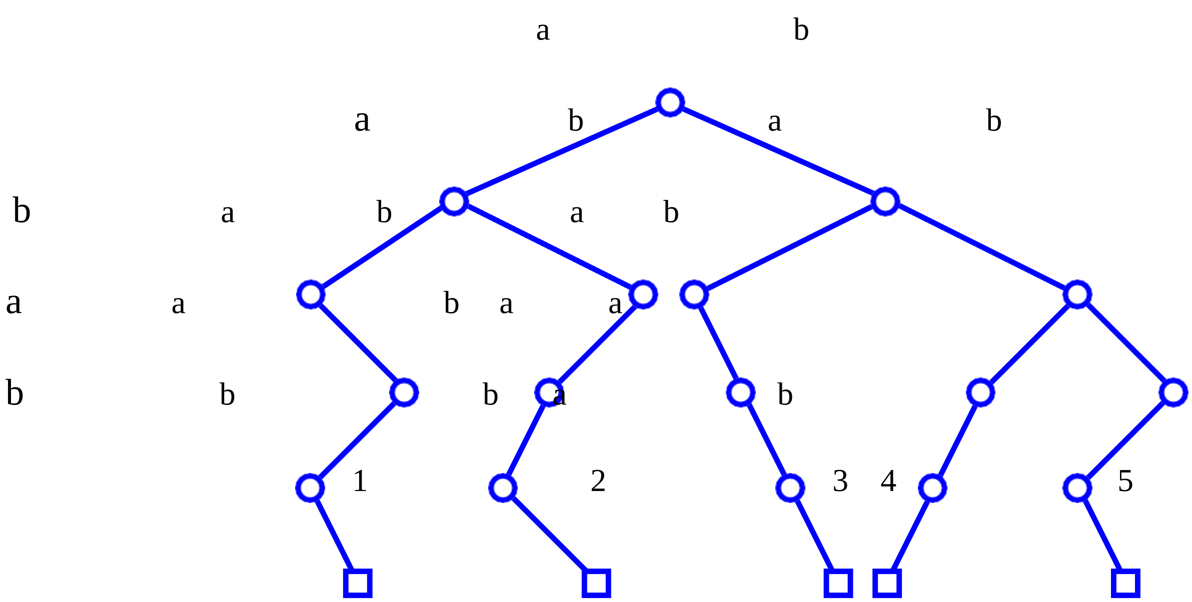
Input: String **Output**: None

prefixes(X): Return all the strings in S that have alongest prefix of X
Input: String **Output**: Enumeration ofstrings

Tries (cont.)

- Let *S* be a set of strings from the alphabet Σ suchthat no string in *S* is a prefix to another string. A standard trie for *S* is an ordered tree *T* that:
  - Each edge of *T* is labeled with a character from Σ
  - The ordering of edges out of an internal node isdetermined by the alphabet Σ
  - The path from the root of *T* to any node representsa prefix in Σ that is equal to the concantenation of the characters encountered while traversing the path.

- For example, the standard trie over the alphabet Σ = {a, b} for the set {aabab, abaab, babbb, bbaaa,bbab}



Tries (cont.)

- An internal node can have 1 to *d* children when d isthe size of the alphabet. Our example is essentially a binary tree.

- A path from the root of *T* to an internal node *v* atdepth *i* corresponds to an *i*-character prefix of a string of *S*.

- We can implement a trie with an ordered tree by storing the character associated with an edge at the child node below it.

Compressed Tries

- A compressed trie is like a standard trie but makessure that each trie had a degree of at least 2. Single child nodes are compressed into an single edge.

- A **critical node** is a node v such that v is labeled witha string from S, v has at least 2 children, or v is the root.

- To convert a standard trie to a compressed trie wereplace an edge (v_0, v_1) each chain on nodes (v_0, v_1...v_k) for k 2 such that

  - v_0 and v_1 are critical but v_1 is critical for $0<i<k$
  - each v_1 has only one child

- Each internal node in a compressed tire has at least two children and each external is associated with a string. The compression reduces the total space for the trie from O(*m*) where *m* is the sum of the the lengths of strings in *S* to O(*n*) where *n* is the numberof strings in *S*.

Compressed Tries (cont.)

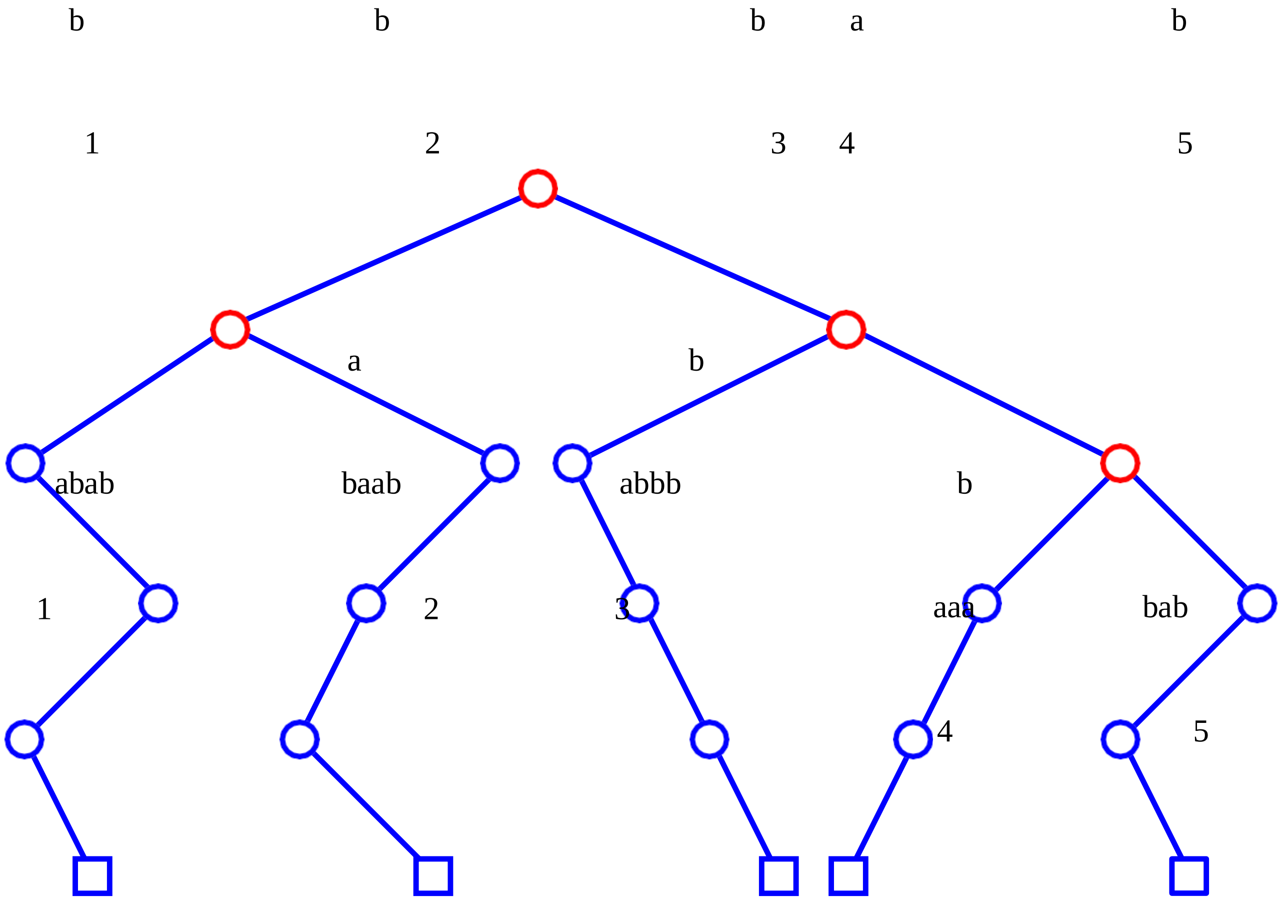
- An example:

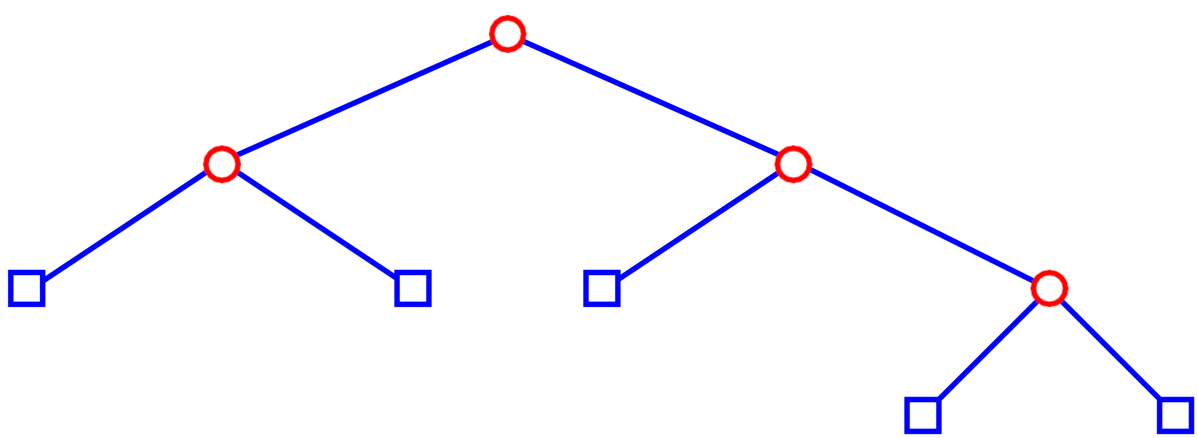
a b

a b a b

b a b a b

a a b a a


b
b
b
a
b
1
2
3
4
5
a
b
abab
baab
abbb
b
1
2
3
aaa
bab
4
5



Prefix Queries on a Trie

Algorithm prefixQuery(T, X):

Input: Trie T for a set S of strings and a query string X

Output: The node v of T such that the labeled nodes ofthe subtree of T rooted at v store the strings of S with a longest prefix in common with X

$v \leftarrow T$.root()

$i \leftarrow 0$ {i is an index into the string X}

repeat

for each child w of v **do**

let e be the edge (v,w)

$Y \leftarrow$string(e) {Y is the substring associated with e}$l \leftarrow Y$.length() {l=1 if T is a standard trie}

Z¨X.substring(i, i+l-1) {Z holds the next l characters of X}

if Z = Y **then**

$v \leftarrow$w

$i \leftarrow i$+1{move to W, incrementing i past Z}

break out of the **for** loop

else if a proper prefix of Z matched a proper prefixof Y **then**

$v \leftarrow$w

break out ot the **repeat** loop

until v is external **or** $v \neq$w

return v

Insertion and Deletion

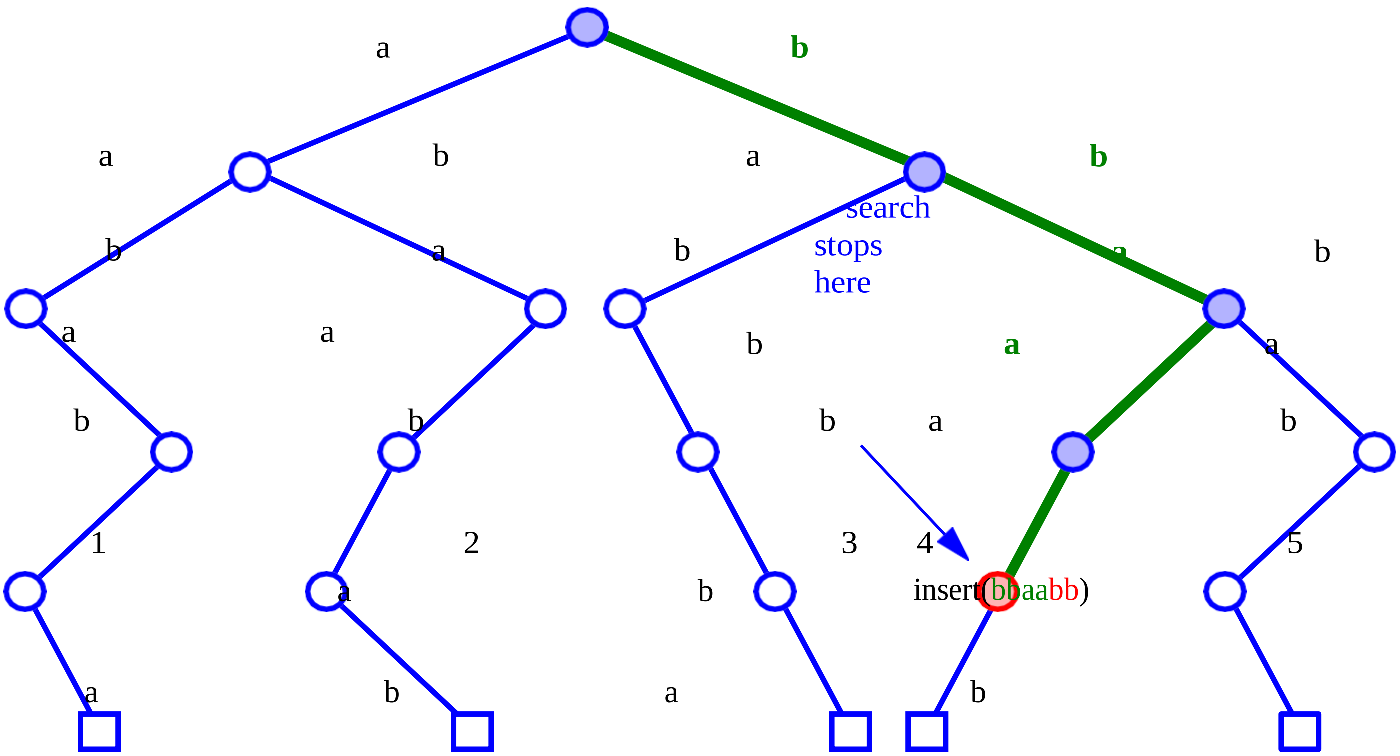
- Insertion: We first perform a prefix query for string X. Let us examine the ways a prefix query may endin terms of insertion.
  - The query terminates at node v. Let X_1 be the prefix of X that matched in the trie up to node v and X_2 be the rest of X. If X_2 is an empt string we

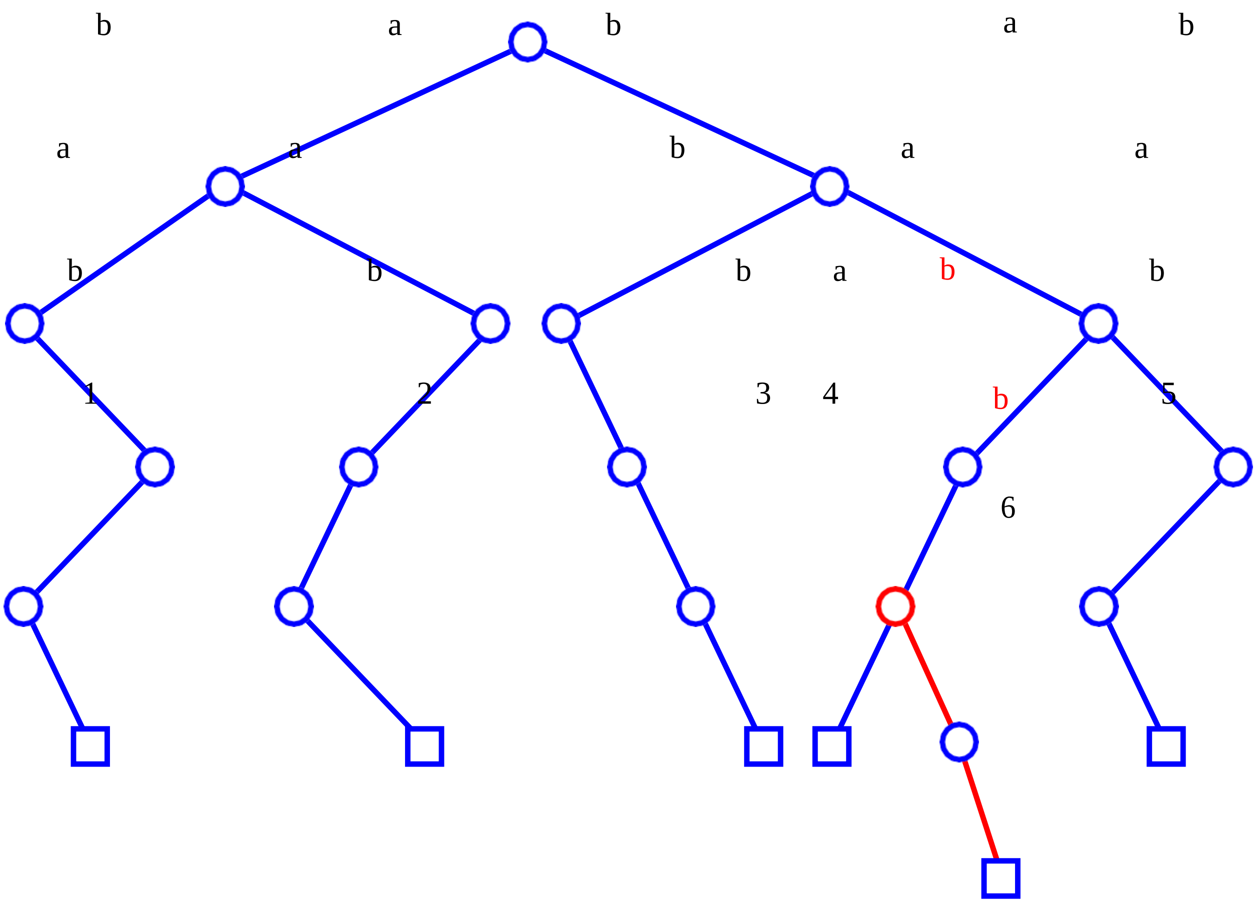
    label v with X and the end. Otherwise we creat a

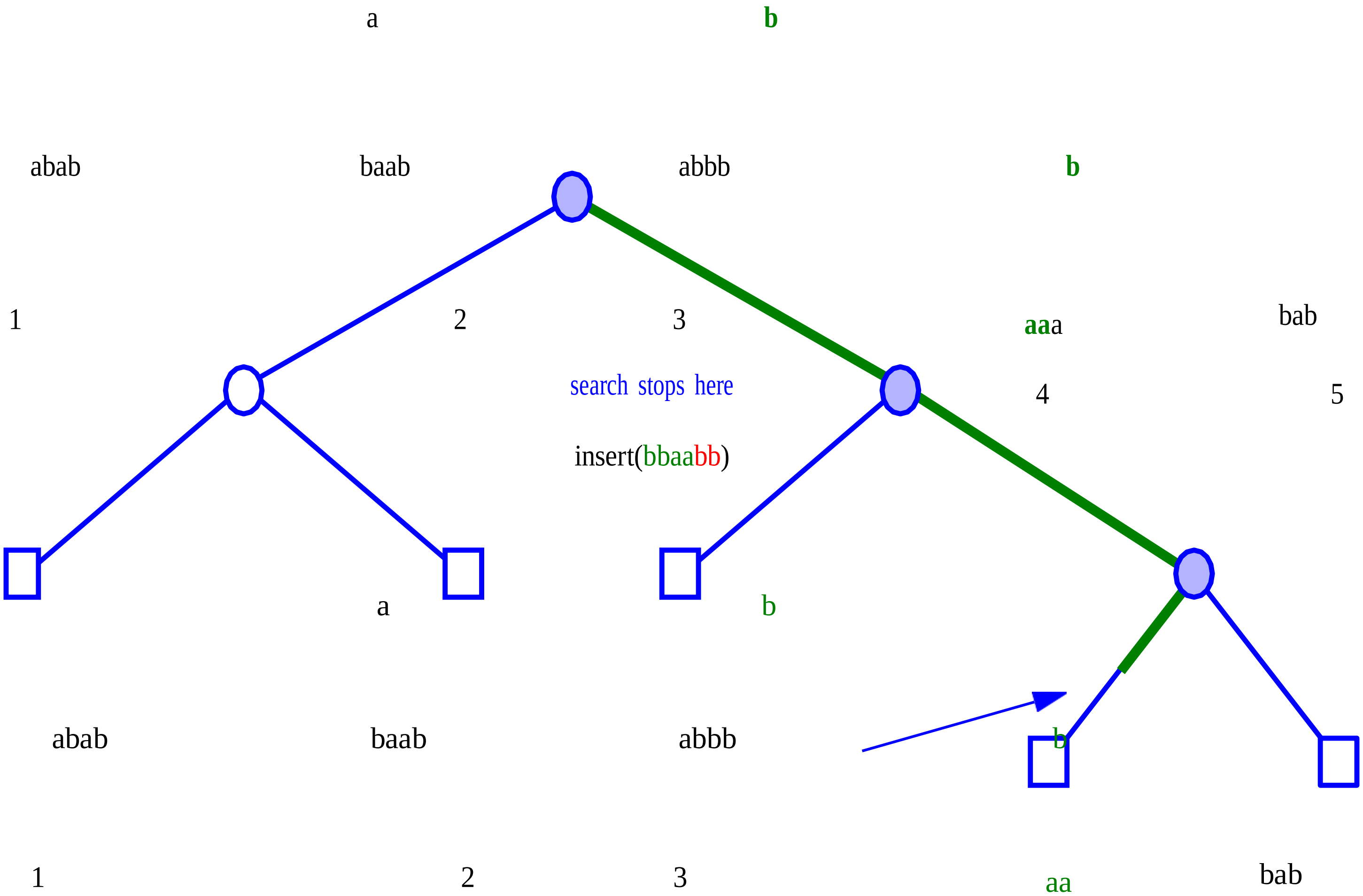
    new external node w and label it with X.
  - The query terminates at an edge e=(v, w) because a prefix of X match prefix(v) and a proper prefix of string Y associated with e. Let Y_1 be the part of Ythat X mathed to and Y_2 the rest of Y. Likewise for X_1 and X_2. Then $X=X_1+X_2$ = prefix(v) $+Y_1+X_2$.

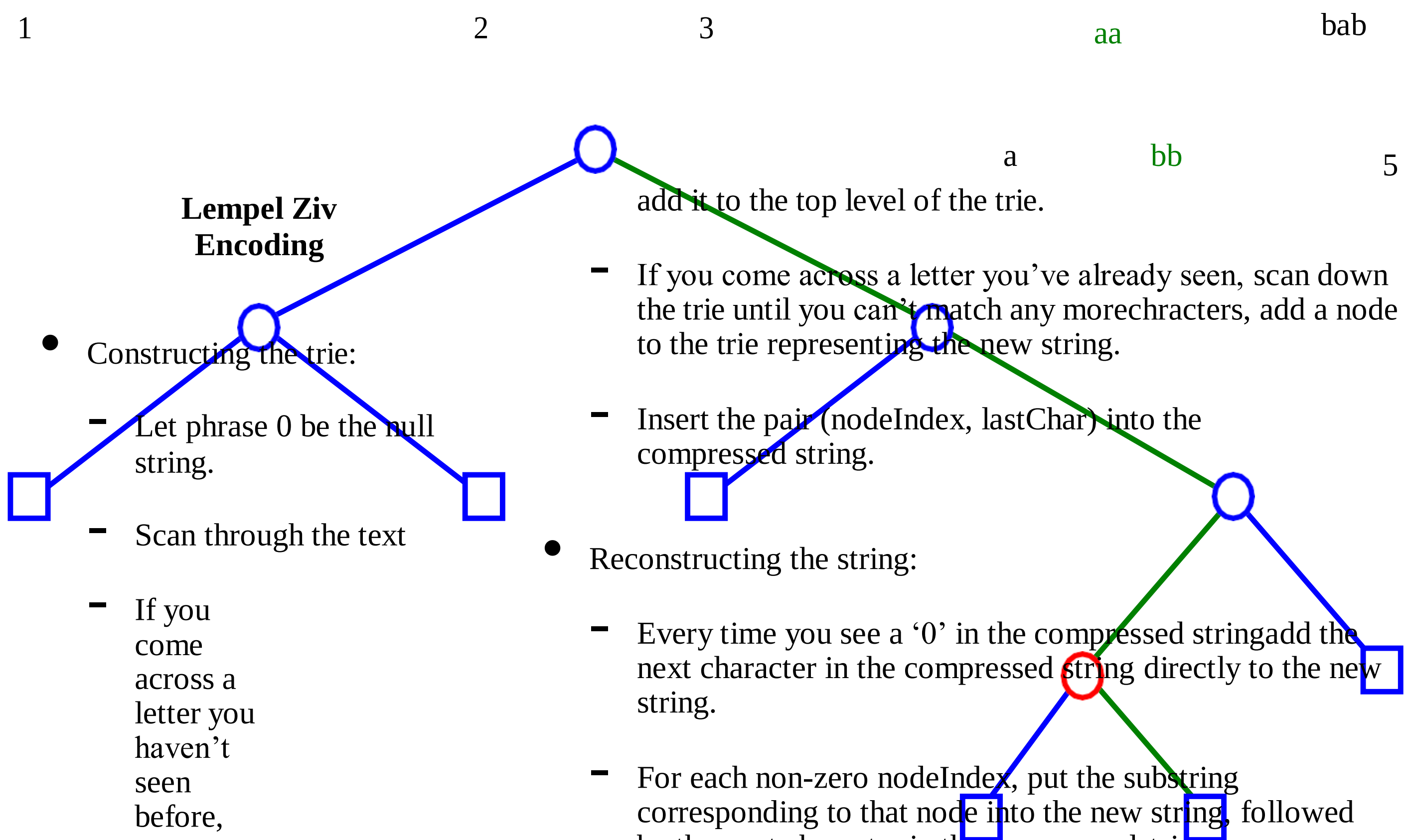
    We create a new node u and split the edges(v, u) and (u, w). If X2 is empty then w label u with X. Otherwise we creat a node z which is external andlabel it X.

- Insertion is O(dn) when d is the size of the alphabetand n is the length of the string t insert.Insertion and Deletion (cont.)









Lempel Ziv Encoding

- Constructing the trie:
  - Let phrase 0 be the null string.
  - Scan through the text
  - If you come across a letter you haven't seen before, add it to the top level of the trie.
  - If you come across a letter you've already seen, scan down the trie until you can't match any morechracters, add a node to the trie representing the new string.
  - Insert the pair (nodeIndex, lastChar) into the compressed string.
- Reconstructing the string:
  - Every time you see a '0' in the compressed stringadd the next character in the compressed string directly to the new string.
  - For each non-zero nodeIndex, put the substring corresponding to that node into the new string, followed by the next character in the compressedstring.

File Compression

- text files are usually stored by representing each character with an 8-bit ASCII code (type man ascii ina Unix shell to see the ASCII encoding)

- the ASCII encoding is an example of **fixed-length encoding**, where each character is represented withthe same number of bits

- in order to reduce the space required to store a text file, we can exploit the fact that some characters are more likely to occur than others

- **variable-length encoding** uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters,and more bits to rarely used characters.

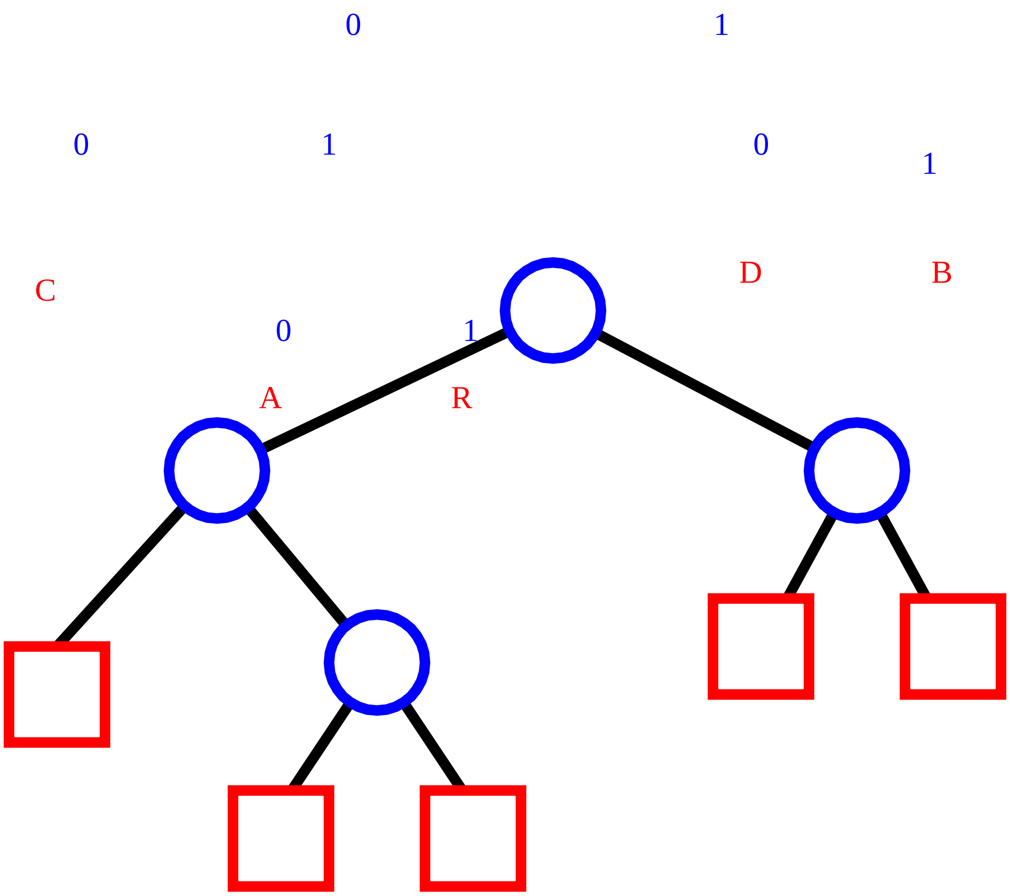
- Example:
  - text: java
  - encoding: a = “0”, j = “11”, v = “10”
  - encoded text: 110100 (6 bits)

- How to decode?
  - a = “0”, j = “01”, v = “00”
  - encoded text: 010000 (6 bits)
  - is this java, jvv, jaaaa ...

Encoding Trie

- to prevent ambiguities in decoding, we require that the encoding satisfies the **prefix rule**, that is, no codeis a prefix of another code
  - a = “0”, j = “11”, v = “10” satisfies the prefix rule
  - a = “0”, j = “01”, v= “00” does **not** satisfy the prefix rule (the code of a is a prefix of the codes of j and v)

- we use an **encoding trie** to define an encoding thatsatisfies the prefix rule
  - the characters stored at the external nodes
  - a left edge means 0
  - a right edge means 1



A = 010

B = 11

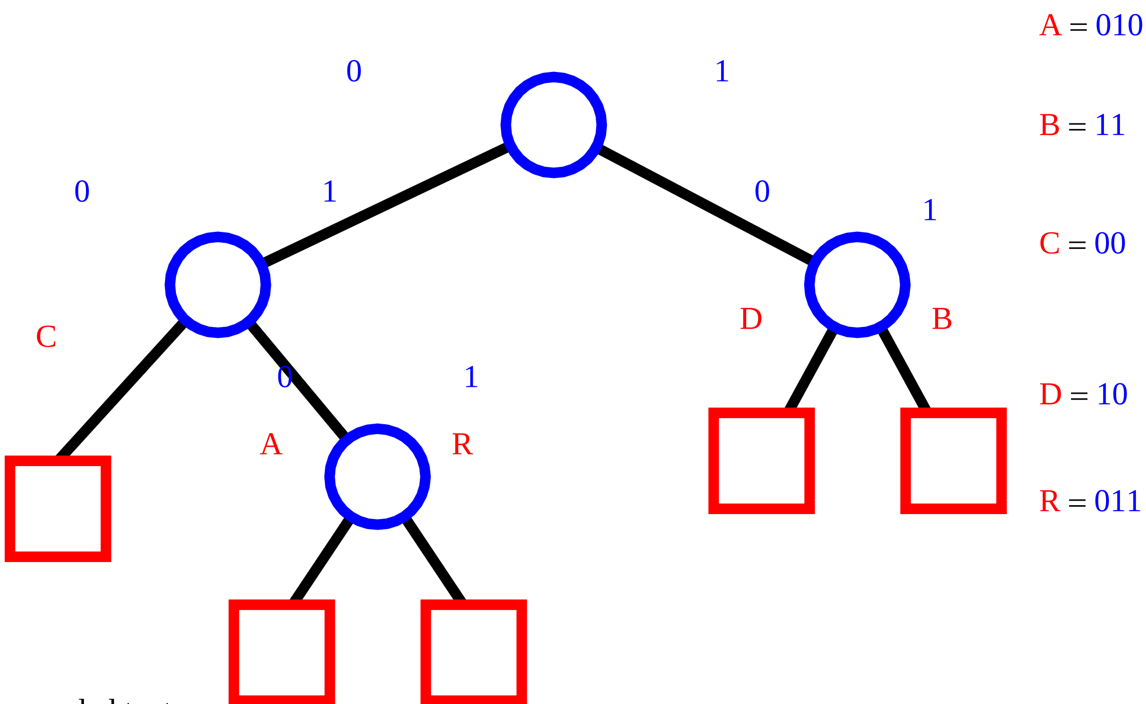
C = 00

D = 10

R = 011

Example of Decoding

- trie:



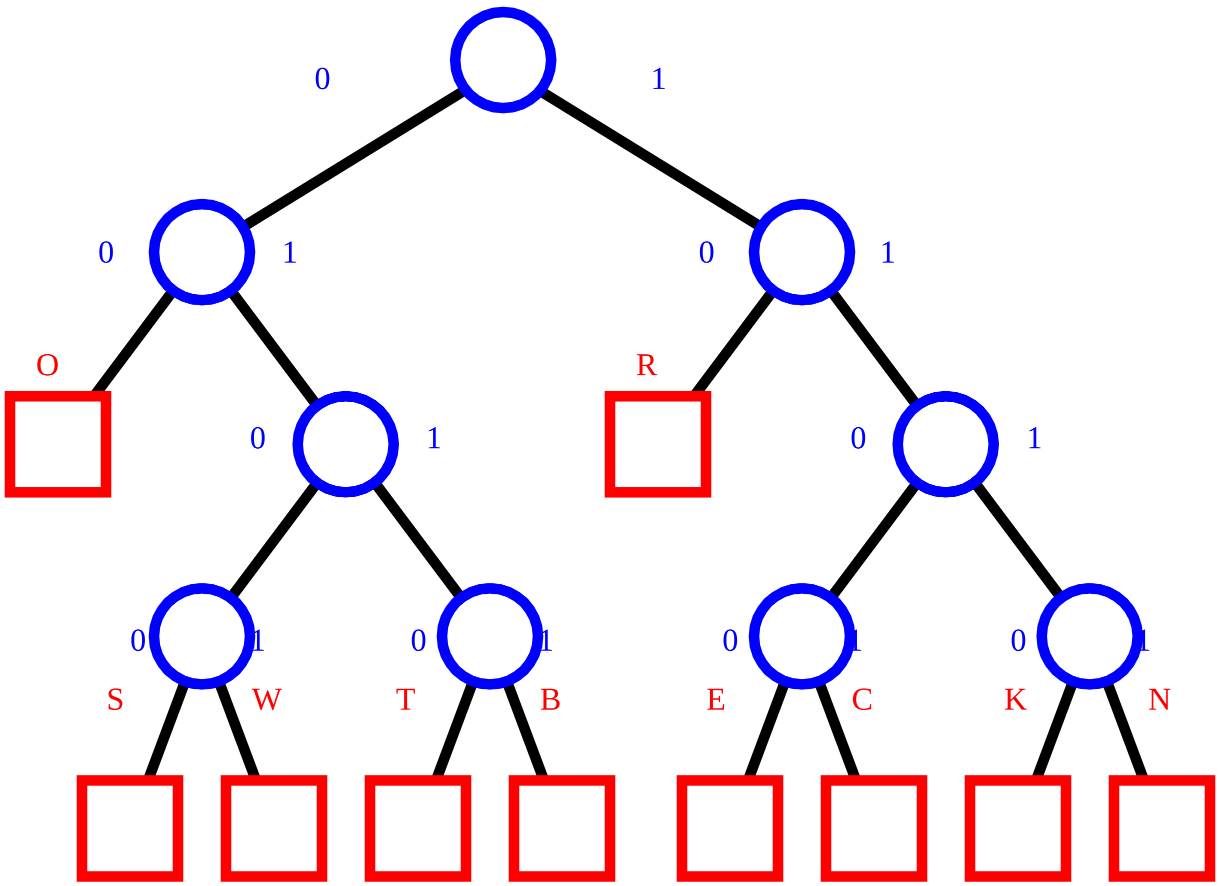
- encoded text:

010110110100001010010110110 10

- text:

A B R A C A D A B R A

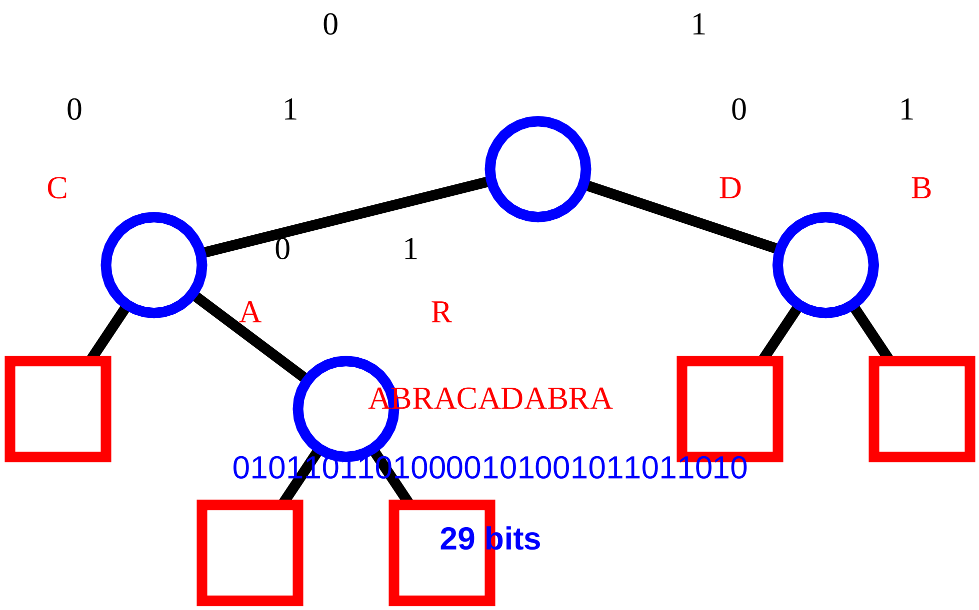
Trie this!

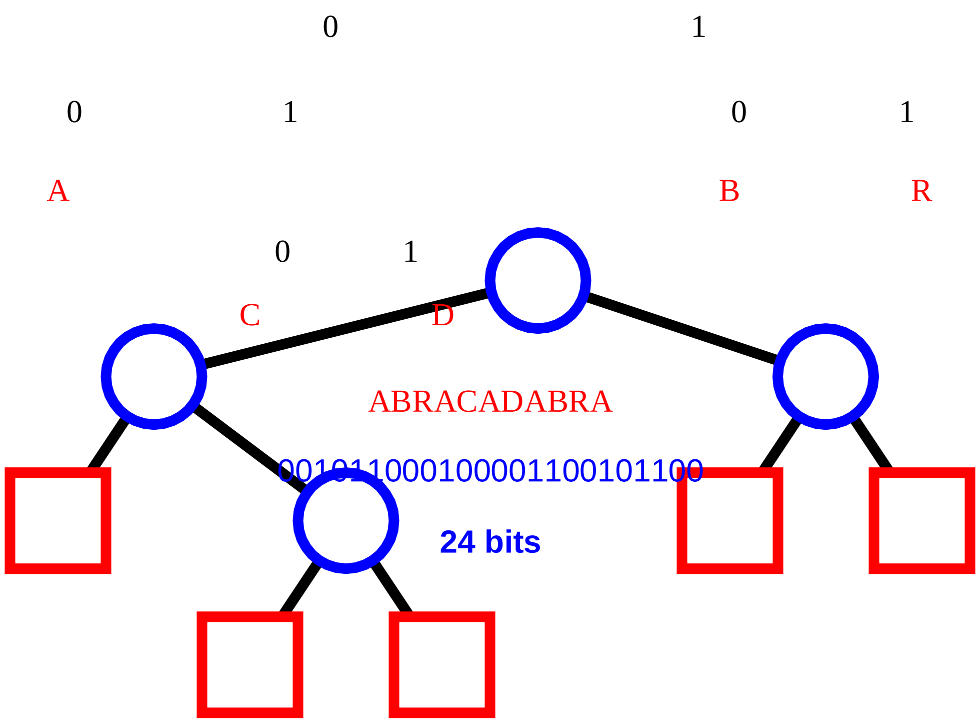


1000011111001001100011101111000101010011010100

Optimal Compression

- An issue with encoding tries is to insure that theencoded text is as short as possible:

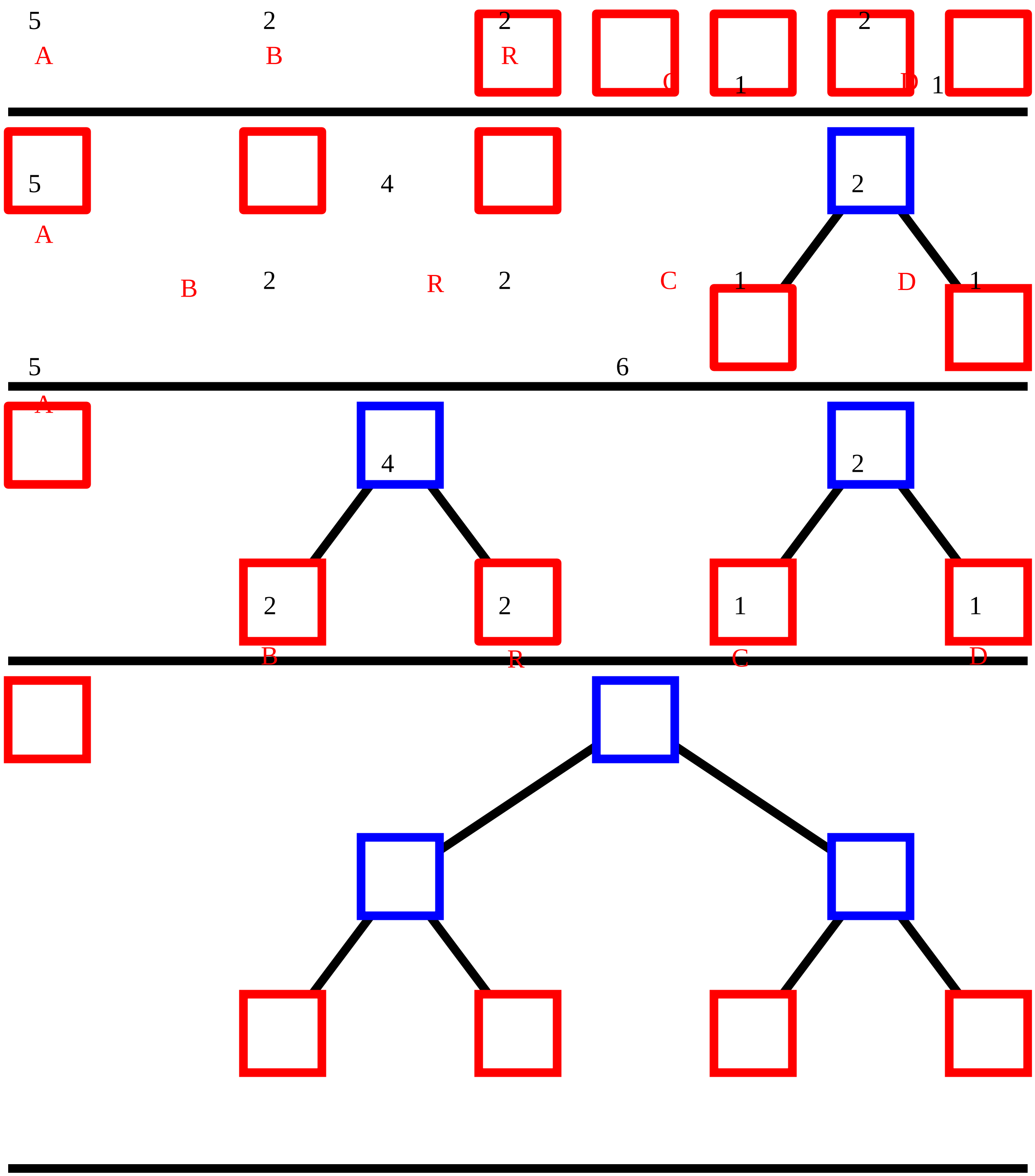




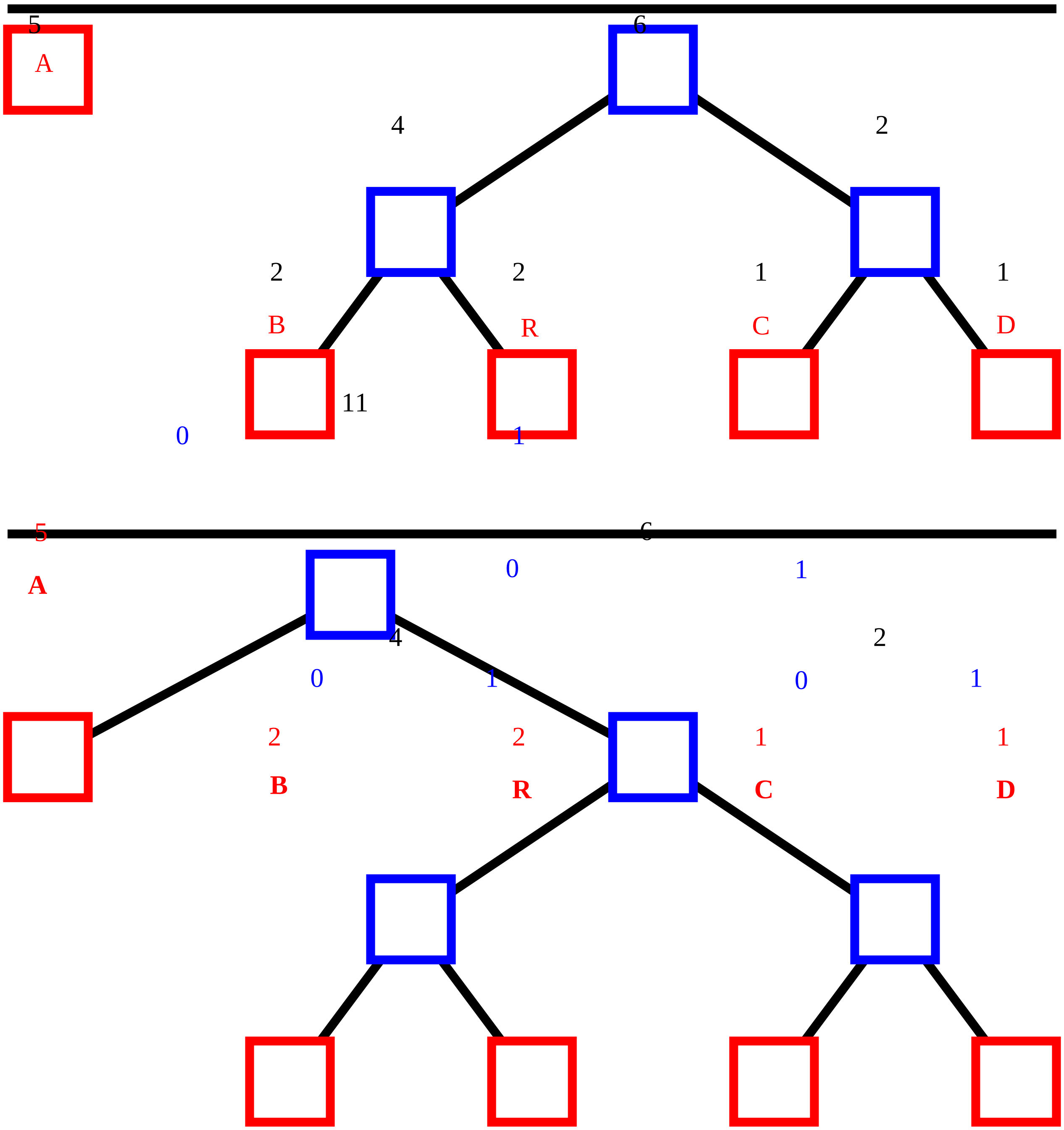
Huffman Encoding Trie

ABRACADABRA

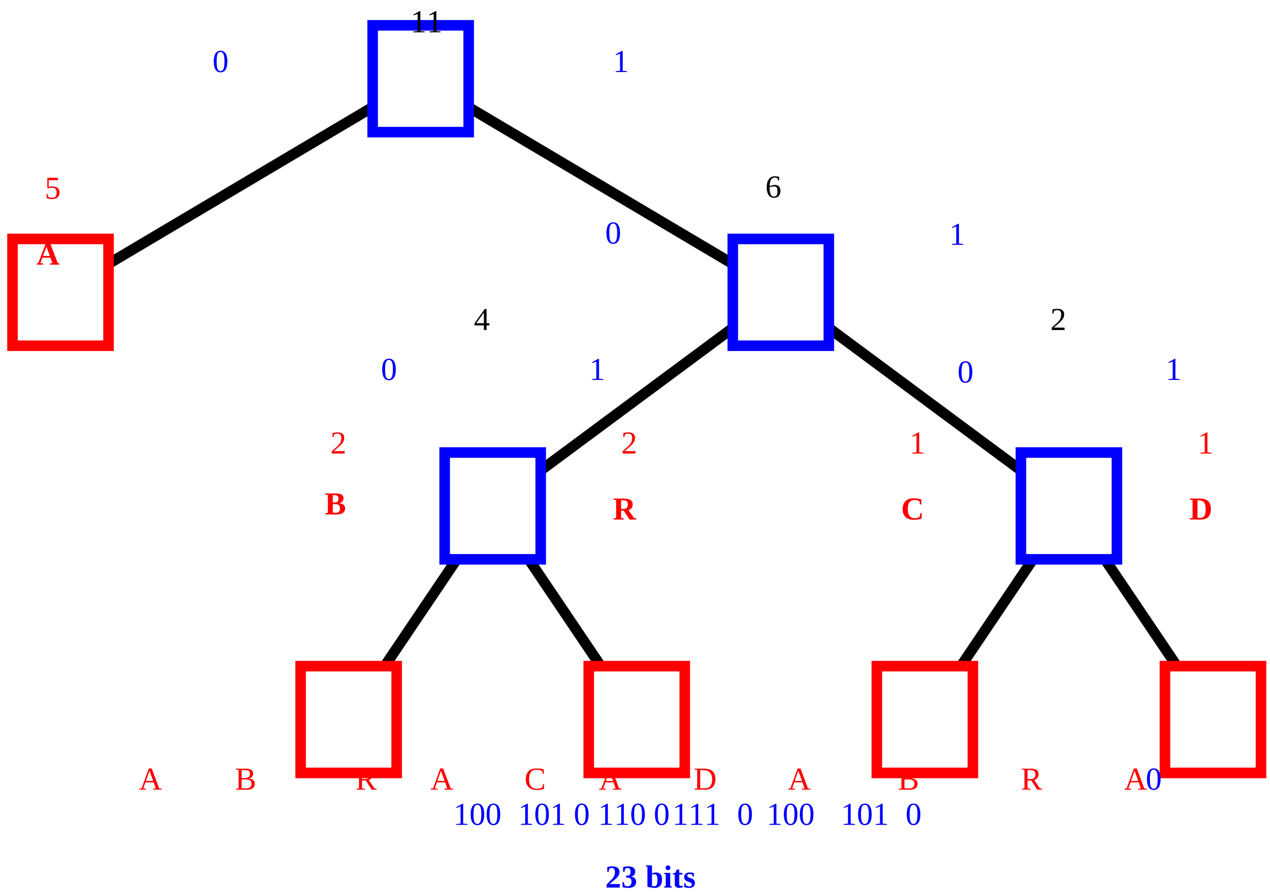
character	A	B	R	C	D
frequency	5	2	2	1	1



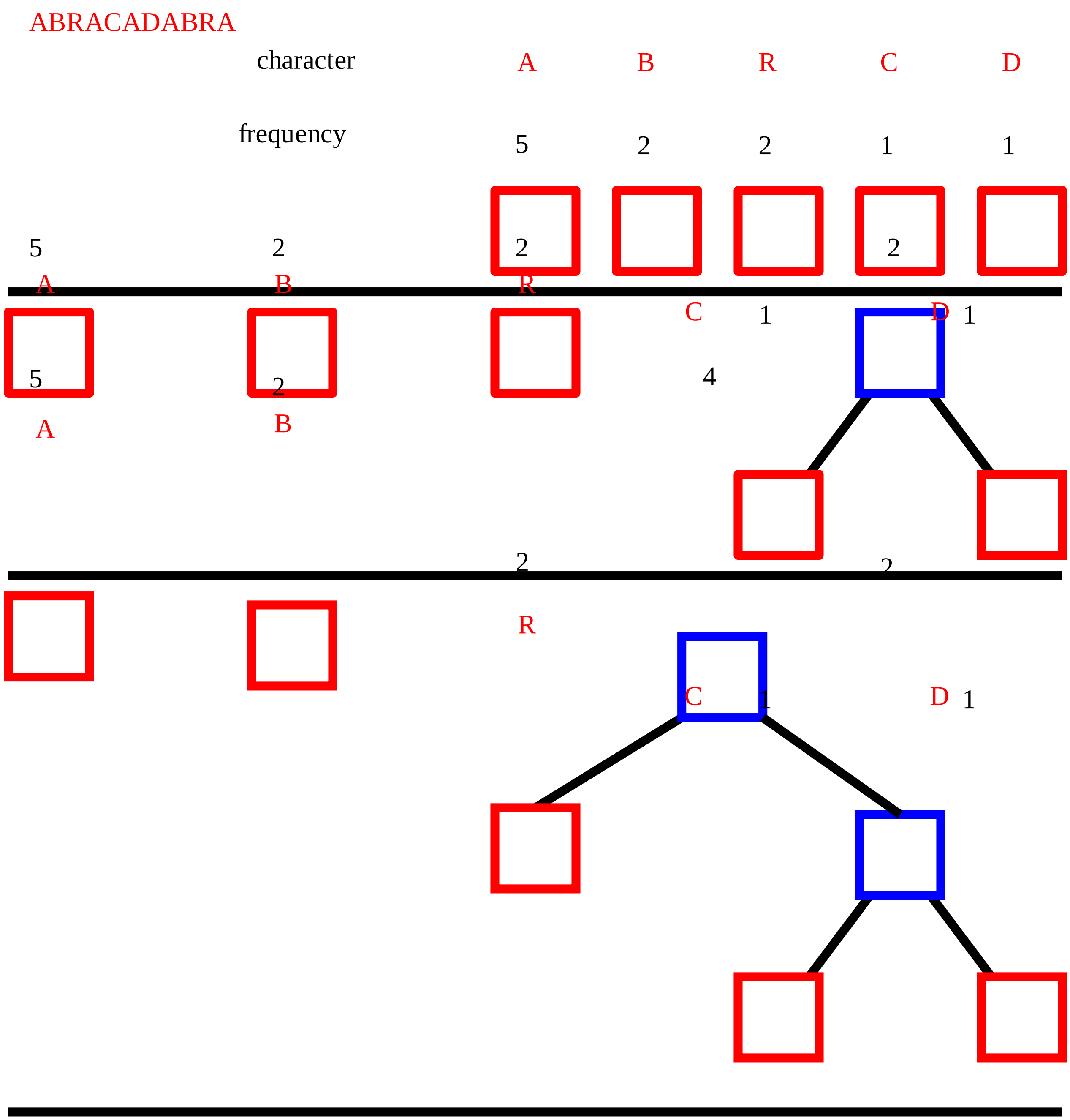
Huffman Encoding Trie (contd.)



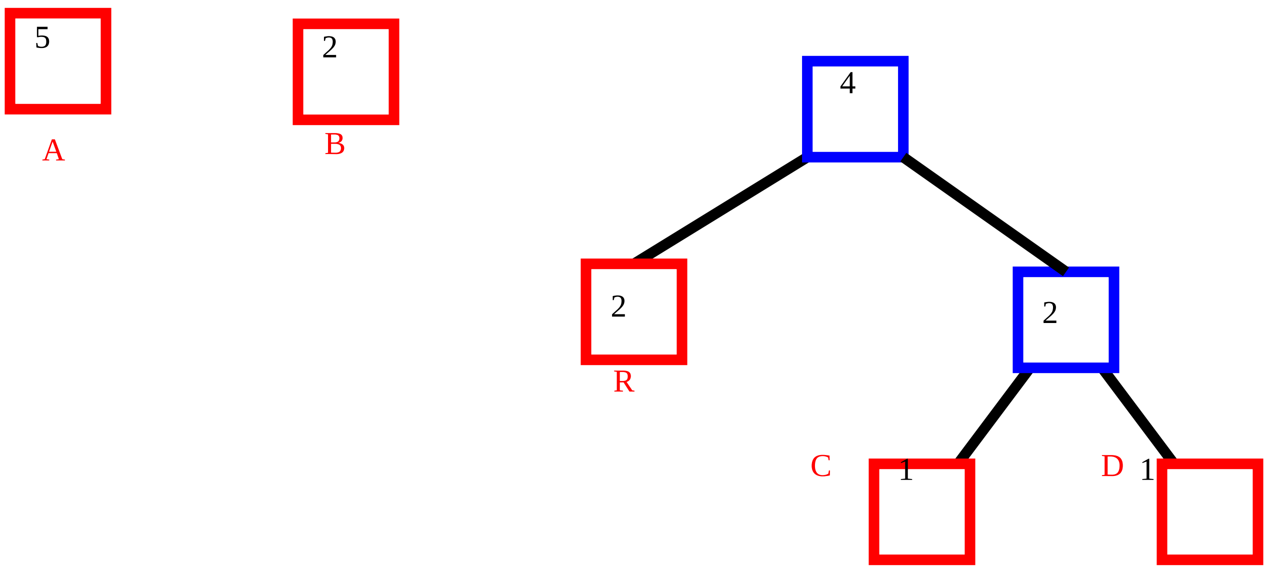
Final Huffman Encoding Trie

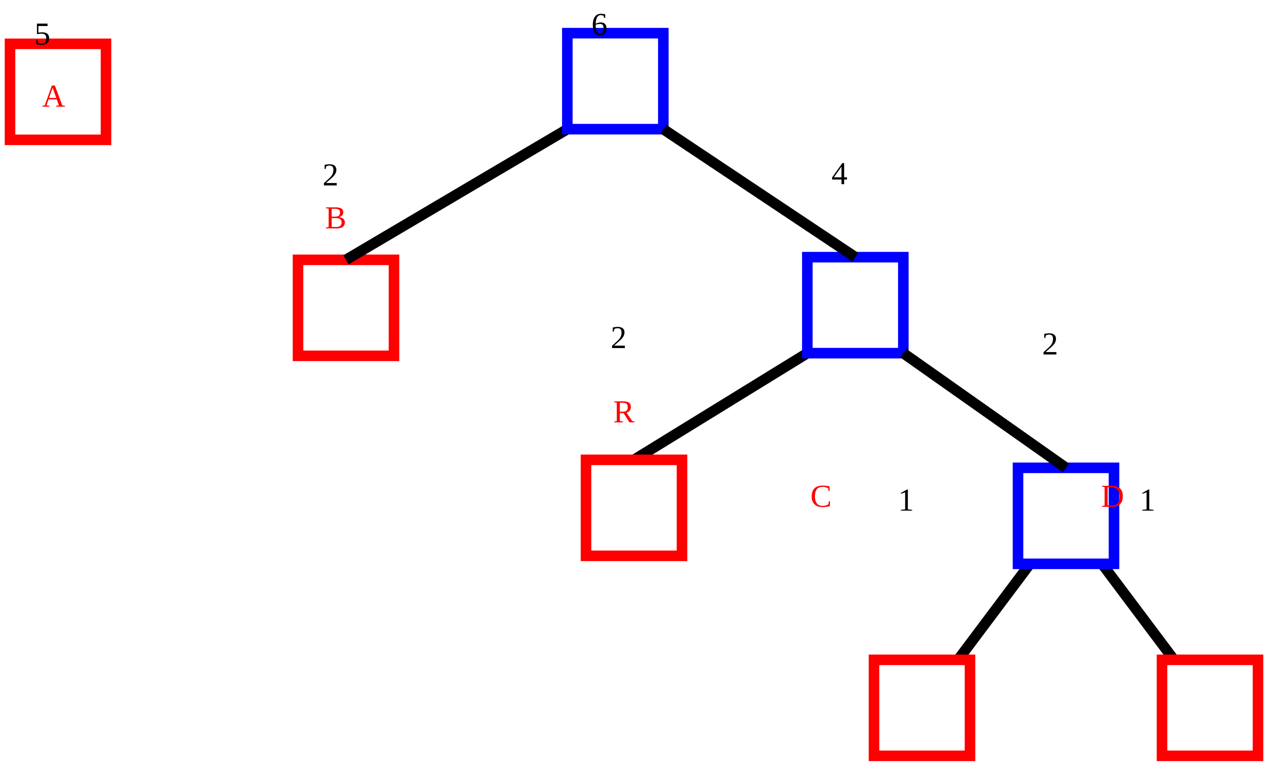


Another Huffman Encoding Trie

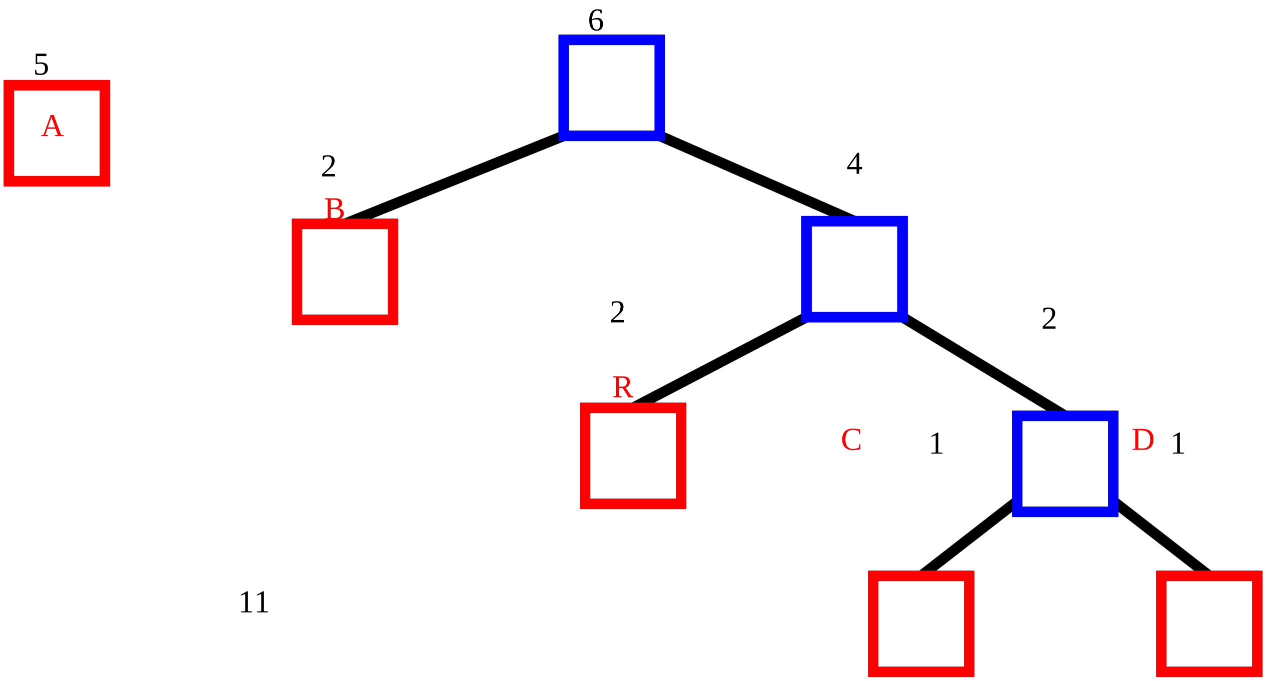


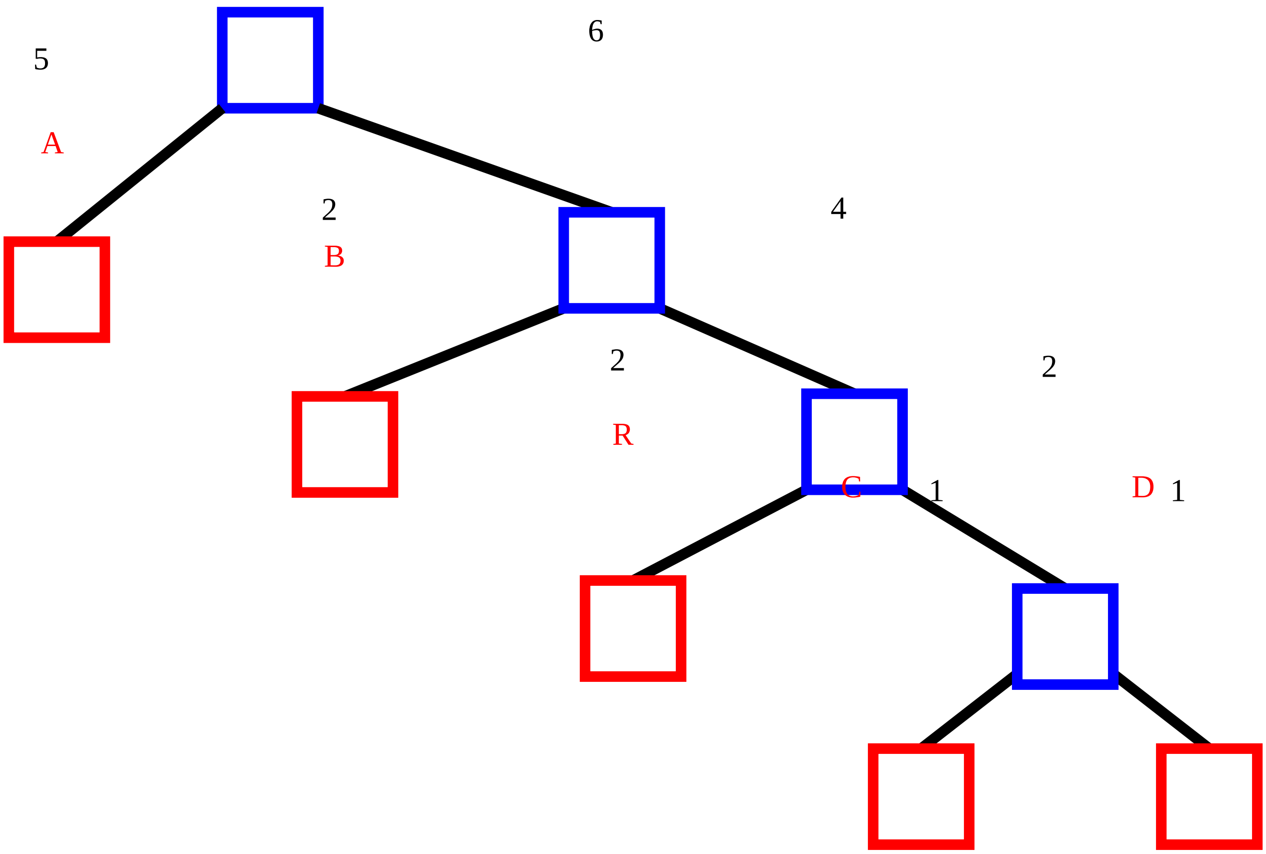
Another Huffman Encoding Trie



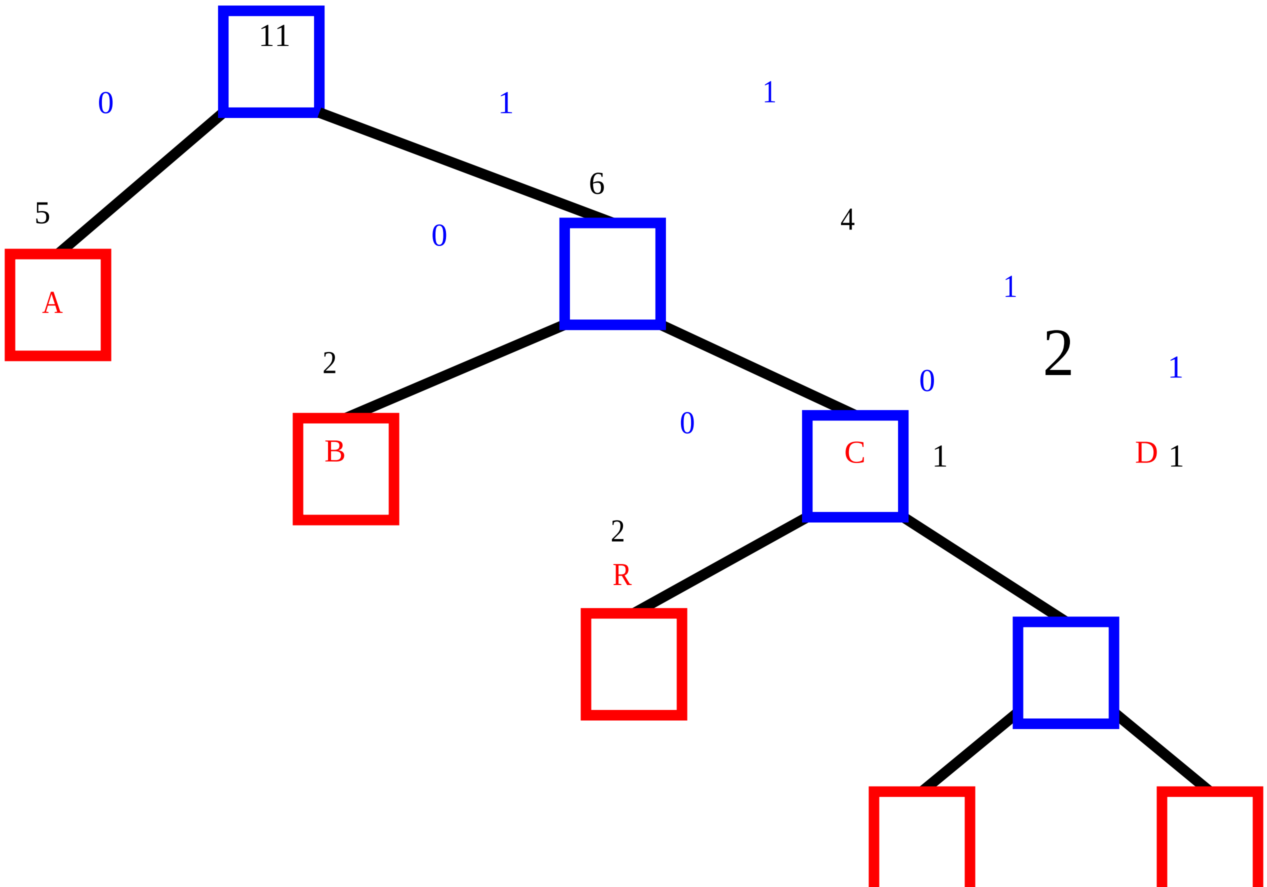


Another Huffman Encoding Trie





Another Huffman Encoding Trie



A B R A C A D A B R A0

10 110 0 1100 0 1111 0 10 110 0

23 bits

Construction Algorithm

- with a Huffman encoding trie, the encoded text hasminimal length

Algorithm Huffman(X): **Input**: String X of length n

Output: Encoding trie for X

Compute the frequency $f(c)$ of each character c of X.Initialize a priority queue Q.

for each character c in X **do**

Create a single-node tree T storing $c$$Q$.insertItem($f(c)$, T)

while Q.size() > 1 **do**

$f_1 \leftarrow Q$.minKey()

$T_1 \leftarrow Q$.removeMinElement()$f_2 \leftarrow Q$.minKey()

$T_2 \leftarrow Q$.removeMinElement()

Create a new tree T with left subtree T_1 and rightsubtree T_2.

Q.insertItem($f_1 + f_2$)

return tree Q.removeMinElement()

- runing time for a text of length n with k distinctcharacters: O(n + k log k)

- we can use Huffman encoding also for binary files(bitmaps, executables, etc.)
- common groups of bits are stored at the leaves
- Example of an encoding suitable for b/w bitmaps