

CMSC 132: Object-Oriented Programming II



Advanced Tree Structures

Department of Computer Science
University of Maryland, College Park

1

Overview

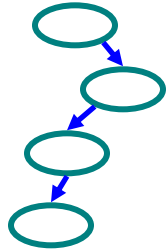
- **Binary trees**
 - Balance
 - Rotation
- **Multi-way trees**
 - Search
 - Insert
- **Indexed tries**

2

Tree Balance

■ Degenerate

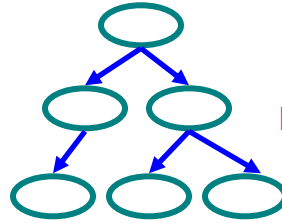
- Worst case
- Search in $O(n)$ time



Degenerate
binary tree

■ Balanced

- Average case
- Search in $O(\log(n))$ time



Balanced
binary tree

3

Tree Balance

■ Question

- Can we keep tree (mostly) balanced?

■ Self-balancing binary search trees

- AVL trees
- Red-black trees

■ Approach

- Select invariant (that keeps tree balanced)
- Fix tree after each insertion / deletion
 - Maintain invariant using **rotations**
- Provides operations with $O(\log(n))$ worst case

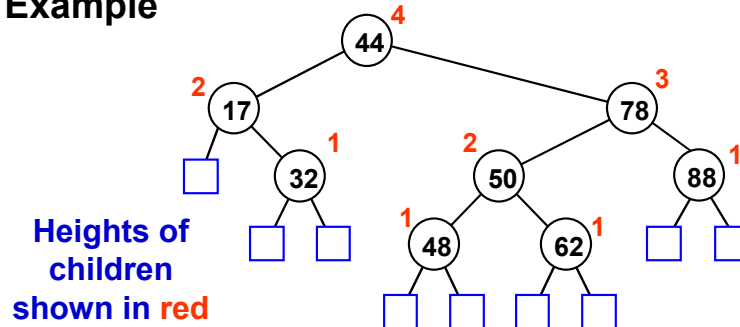
4

AVL Trees

■ Properties

- Binary search tree
- Heights of children for node differ by at most 1

■ Example



5

AVL Trees

■ History

- Discovered in 1962 by two Russian mathematicians, Adelson-Velskii & Landis

■ Algorithm

1. Find / insert / delete as a binary search tree
2. After each insertion / deletion
 - a) If height of children differ by more than 1
 - b) **Rotate** children until subtrees are balanced
 - c) Repeat check for parent (until root reached)

6

Red-black Trees

■ Properties

- Binary search tree
- Every node is red or black
- The root is black
- Every leaf is black
- All children of red nodes are black
- For each leaf, same # of black nodes on path to root

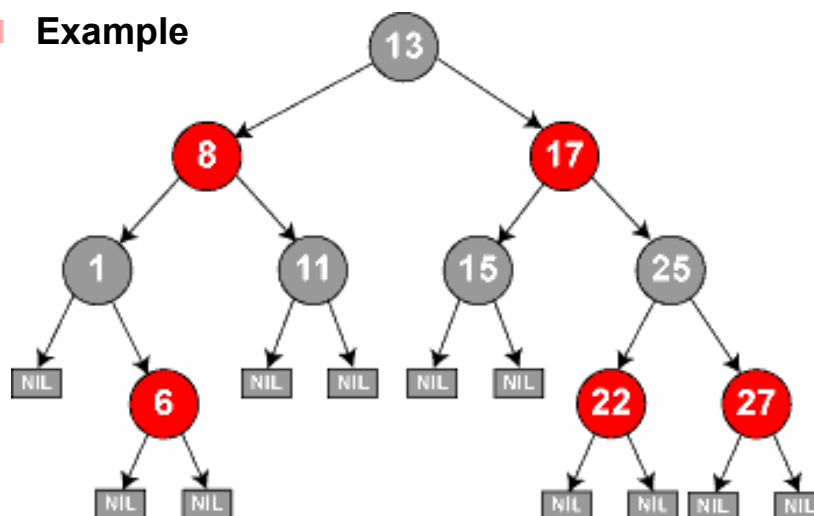
■ Characteristics

- Properties ensures no leaf is twice as far from root as another leaf

7

Red-black Trees

■ Example



8

Red-black Trees

- **History**
 - Discovered in 1972 by Rudolf Bayer
- **Algorithm**
 - Insert / delete may require complicated bookkeeping & rotations
- **Java collections**
 - TreeMap, TreeSet use red-black trees

9

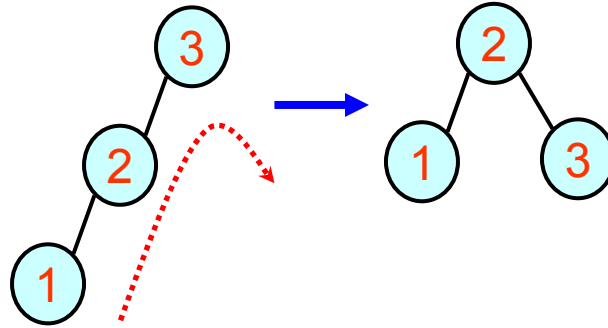
Tree Rotations

- **Changes shape of tree**
 - Move nodes
 - Change edges
- **Types**
 - **Single rotation**
 - Left
 - Right
 - **Double rotation**
 - Left-right
 - Right-left

10

Tree Rotation Example

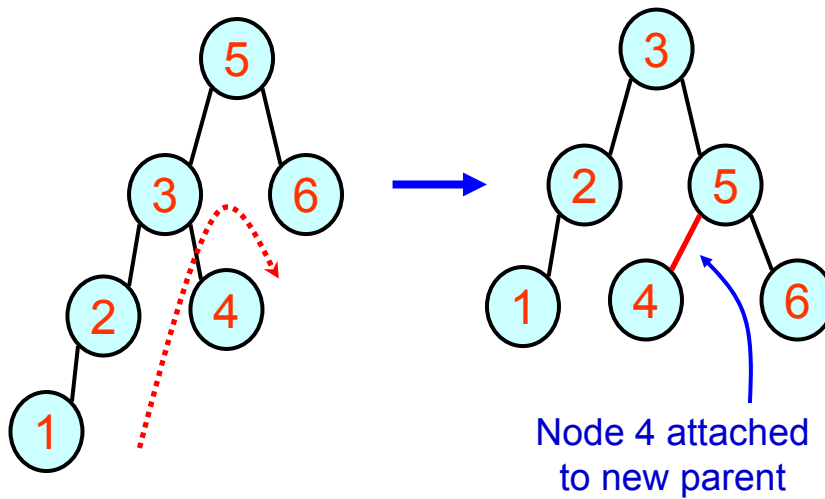
■ Single right rotation



11

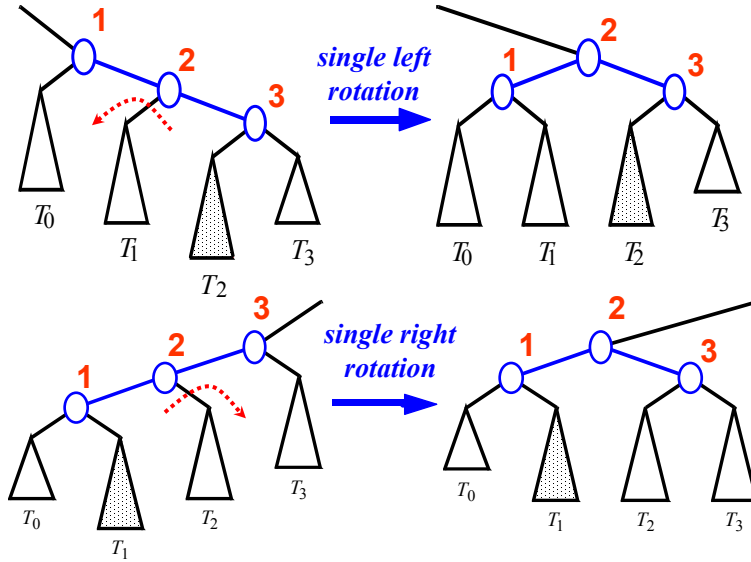
Tree Rotation Example

■ Single right rotation



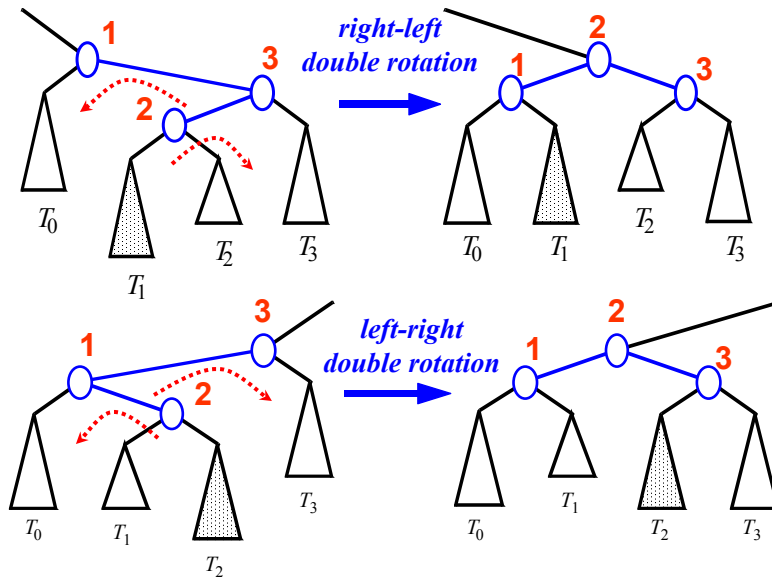
12

Example – Single Rotations



13

Example – Double Rotations



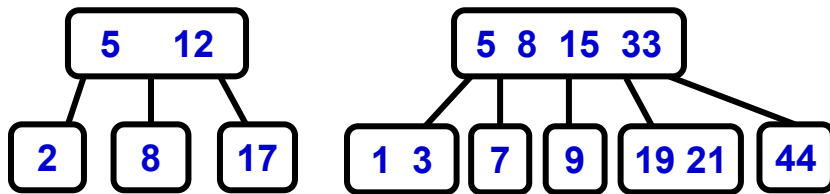
14

Multi-way Search Trees

■ Properties

- Generalization of binary search tree
- Node contains 1...k keys (in sorted order)
- Node contains 2...k+1 children
- Keys in j^{th} child < j^{th} key < keys in $(j+1)^{\text{th}}$ child

■ Examples



15

Types of Multi-way Search Trees

■ 2-3 tree

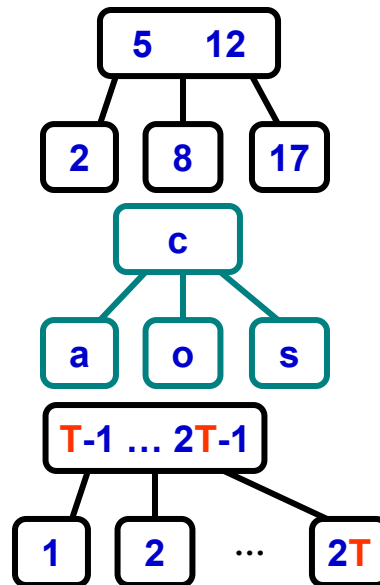
- Internal nodes have 2 or 3 children

■ Index search trie

- Internal nodes have up to 26 children (for strings)

■ B-tree

- T = minimum degree
- Non-root internal nodes have $T-1$ to $2T-1$ children
- All leaves have same depth



16

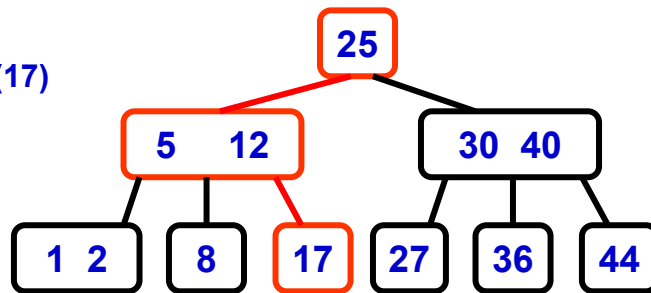
Multi-way Search Trees

■ Search algorithm

1. Compare key x to $1 \dots k$ keys in node
2. If $x = \text{some key}$ then return node
3. Else if ($x < \text{key } j$) search child j
4. Else if ($x > \text{all keys}$) search child $k+1$

■ Example

- Search(17)



17

Multi-way Search Trees

■ Insert algorithm

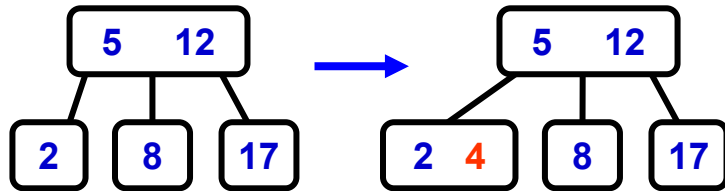
1. Search key x to find node n
2. If (n not full) insert x in n
3. Else if (n is full)
 - a) Split n into two nodes
 - b) Move middle key from n to n 's parent
 - c) Insert x in n
 - d) Recursively split n 's parent(s) if necessary

18

Multi-way Search Trees

■ Insert Example (for 2-3 tree)

■ Insert(4)

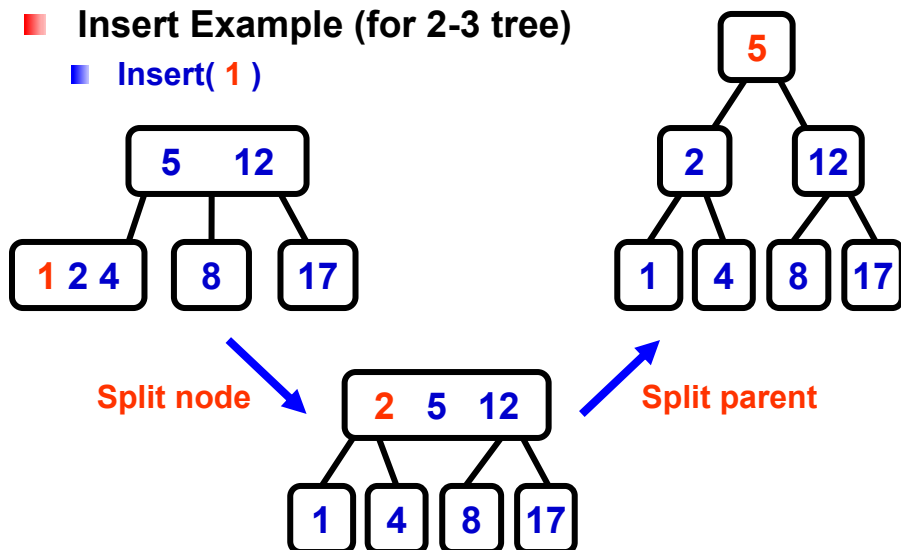


19

Multi-way Search Trees

■ Insert Example (for 2-3 tree)

■ Insert(1)



20

B-Trees

■ Characteristics

- Height of tree is $O(\log_T(n))$
- Reduces number of nodes accessed
- Wasted space for non-full nodes

■ Popular for large databases

- 1 node = 1 disk block
- Reduces number of disk blocks read

21

Indexed Search Tree (Trie)

■ Special case of tree

■ Applicable when

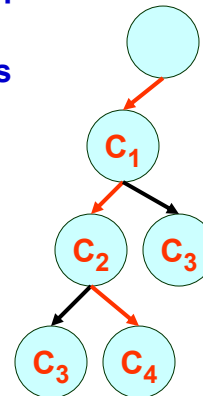
- Key **C** can be decomposed into a sequence of subkeys C_1, C_2, \dots, C_n
- Redundancy exists between subkeys

■ Approach

- Store subkey at each node
- Path through trie yields full key

■ Example

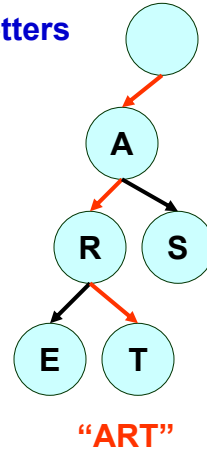
- Huffman tree



22

Tries

- Useful for searching strings
 - String decomposes into sequence of letters
 - Example
 - “ART” ⇒ “A” “R” “T”
- Can be very fast
 - Less overhead than hashing
- May reduce memory
 - Exploiting redundancy
- May require more memory
 - Explicitly storing substrings



23

Types of Tries

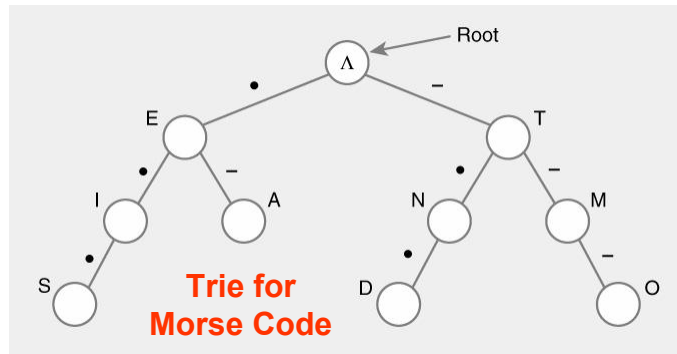
- Standard
 - Single character per node
- Compressed
 - Eliminating chains of nodes
- Compact
 - Stores indices into original string(s)
- Suffix
 - Stores all suffixes of string

24

Standard Tries

■ Approach

- Each node (except root) is labeled with a character
- Children of node are ordered (alphabetically)
- Paths from root to leaves yield all input strings

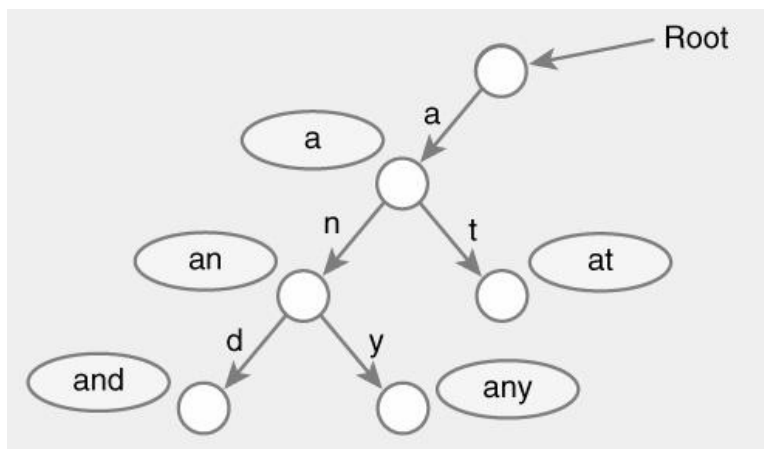


25

Standard Trie Example

■ For strings

- { a, an, and, any, at }

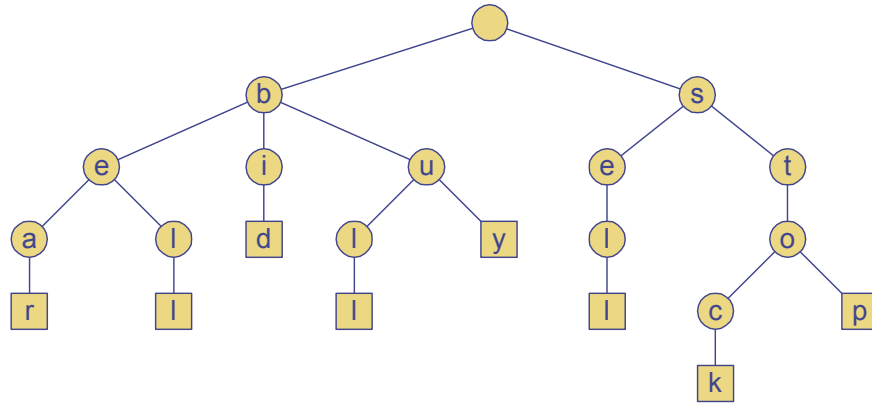


26

Standard Trie Example

■ For strings

■ { bear, bell, bid, bull, buy, sell, stock, stop }



27

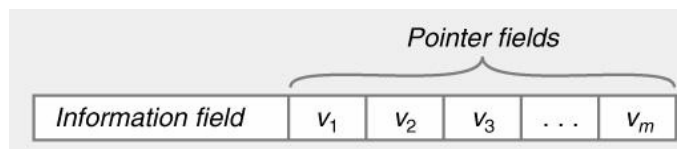
Standard Tries

■ Node structure

- Value between 1...m
- Reference to m children
 - Array or linked list

■ Example

```
Class Node {
    Letter value;           // Letter V = { V1, V2, ... Vm }
    Node child[ m ];
}
```

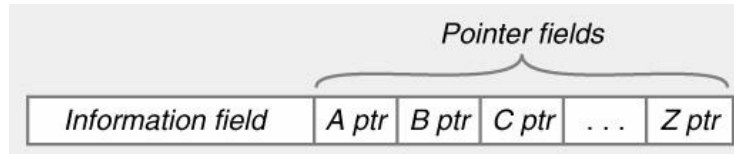


28

Standard Tries

Efficiency

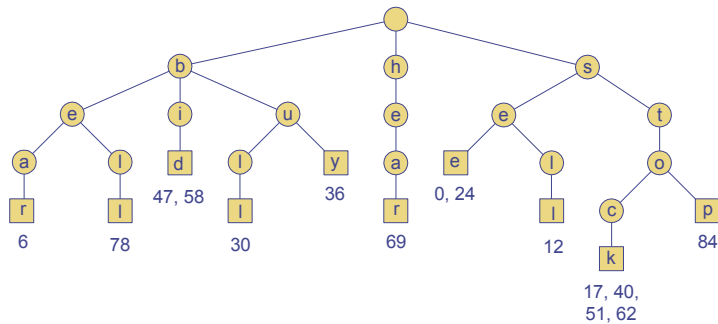
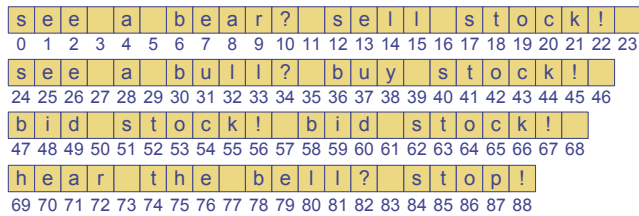
- Uses $O(n)$ space
- Supports search / insert / delete in $O(d \times m)$ time
- For
 - n total size of strings indexed by trie
 - d length of the parameter string
 - m size of the alphabet



29

Word Matching Trie

- Insert words into trie
- Each leaf stores occurrences of word in the text



30

Compressed Trie

■ Observation

- Internal node v of T is redundant if v has one child and is not the root

■ Approach

- A chain of redundant nodes can be compressed
 - Replace chain with single node
 - Include concatenation of labels from chain

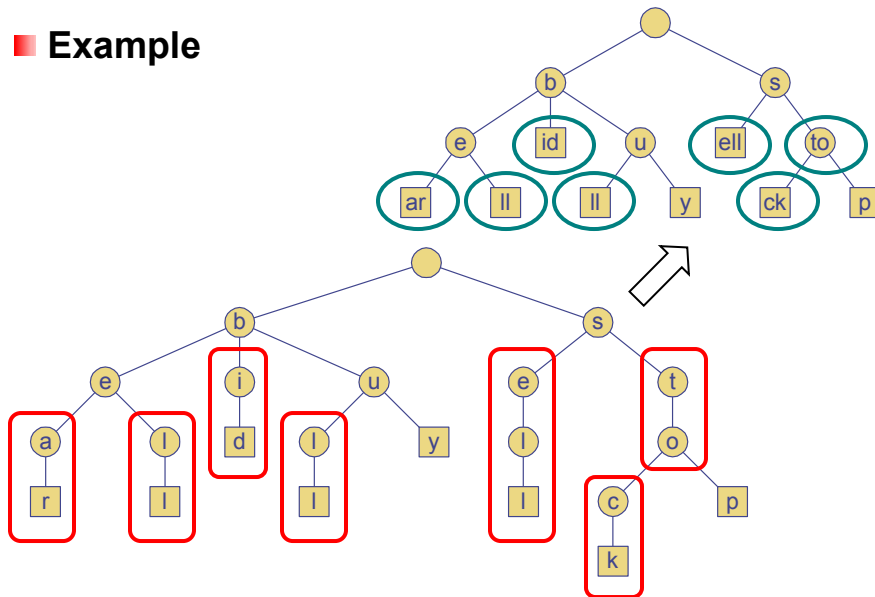
■ Result

- Internal nodes have at least 2 children
- Some nodes have multiple characters

31

Compressed Trie

■ Example



32

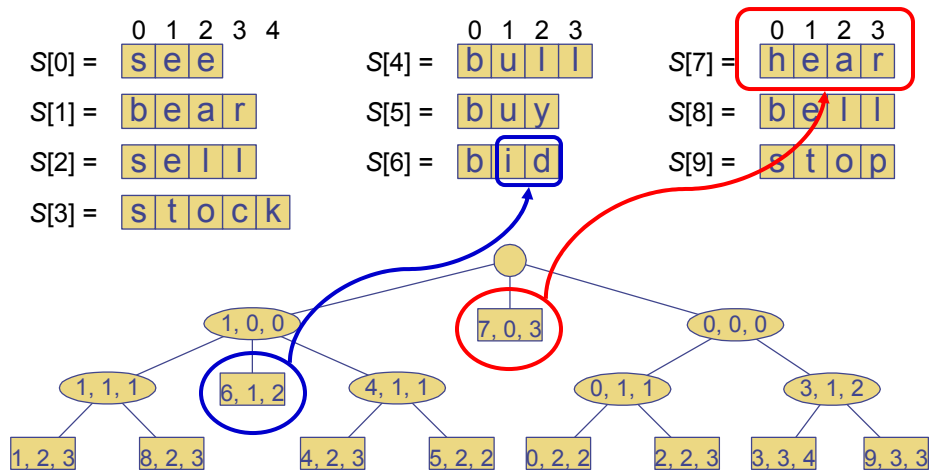
Compact Tries

- Compact representation of a compressed trie
- Approach
 - For an array of strings $S = S[0], \dots, S[s-1]$
 - Store ranges of indices at each node
 - Instead of substring
 - Represent as a triplet of integers (i, j, k)
 - Such that $X = s[i][j..k]$
 - Example: $S[0] = \text{"abcd"}$, $(0,1,2) = \text{"bc"}$
- Properties
 - Uses $O(s)$ space, where $s = \#$ of strings in the array
 - Serves as an auxiliary index structure

33

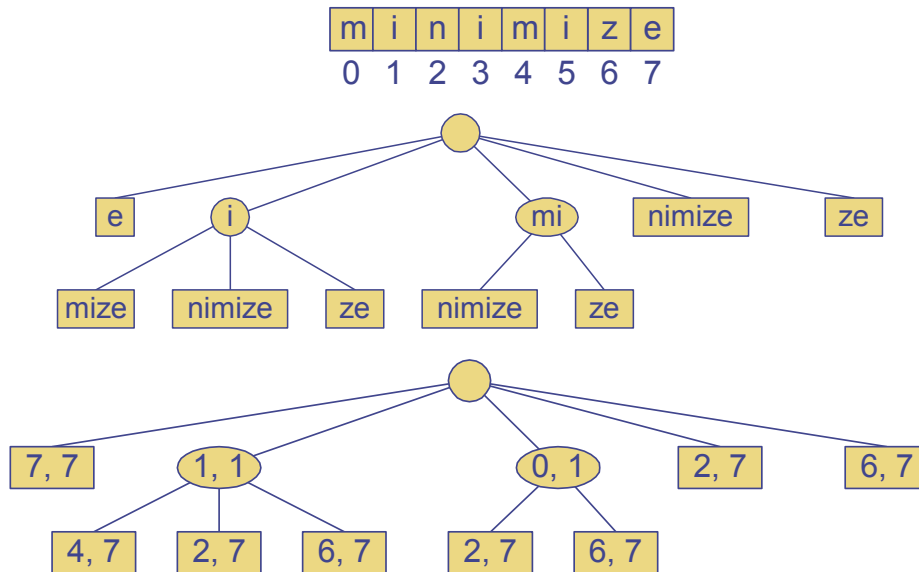
Compact Representation

■ Example



34

Suffix Trie Example



Tries and Web Search Engines

- Search engine index
 - Collection of all searchable words
 - Stored in compressed trie
- Each leaf of trie
 - Associated with a word
 - List of pages (URLs) containing that word
 - Called occurrence list
- Trie is kept in memory (fast)
- Occurrence lists kept in external memory
 - Ranked by relevance

38

Computational Biology

■ DNA

- Sequence of 4 different nucleotides (ATCG)
- Portions of DNA sequence produce proteins (genes)

■ Genome

- Master DNA sequence for organism
- For Human
 - 46 chromosomes
 - 3 billion nucleotides

39

DNA the molecule of life

Trillions of cells

Each cell:

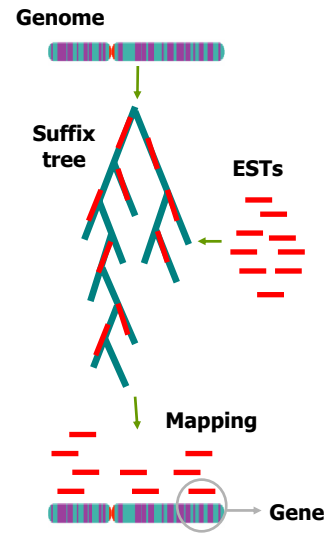
- 46 human chromosomes
- 2 meters of DNA
- 3 billion DNA subunits (the bases: A, T, C, G)
- Approximately 30,000 genes code for proteins that perform most life functions

The diagram illustrates the flow of genetic information: a cell contains chromosomes, which are made of DNA. A specific segment of DNA is a gene, which codes for a protein. The DNA double helix is shown with the bases A, T, C, and G. The protein is shown as a blue ball-and-stick model.

Y-GG 01-0085

Tries and Computational Biology

- **ESTs**
 - Fragments of expressed DNA
 - Indicator for genes (& location)
 - 5.5 million sequences at NIH
- **ESTmapper**
 - Build suffix trie of genome
 - 8 hours, 60 Gbytes
 - Search for ESTs in suffix trie
 - 11 hours w/ 8 processor Sun
- **Search genome w/ BLAST**
 - 5+ years (predicted)



41