DEPARTMENT OF COMPUTER SCIENCE

CMSC 132

Object-Oriented Programming II

Spring 2025

Week 14 Lecture 1 **Advanced Trees**

UNIVERSITY OF MARYLAND FEARLESSLY FORWARD

Announcements

Project 9 Graphs has a *REQUIRED early submission TODAY Wednesday May 7th*, final submission is Sunday May 11th.

Quiz 6 Today Wednesday May 7th on Threads and Graphs

Exam 3 Regrade requests must be entered in Gradescope by *today* Wednesday May 7th at 11:00 PM

Final Exam Review Guide

CMSC132 Course Evaluation

- Please take a few minutes to complete the course evaluation.
- All feedback is read and appreciated

Log in to Student Feedback on Course Experiences: https://CourseExp.umd.edu

Advanced Tree Structures

Advanced Trees Topics

Binary Trees - Balancing and rotations

- KVL Trees
- Red-Black Trees

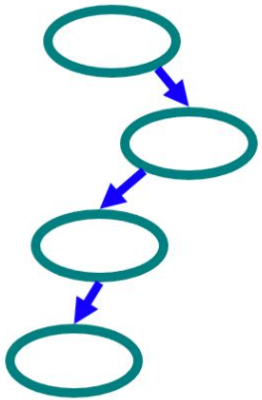
Multi-way Trees

Indexed Tries

Balanced BST

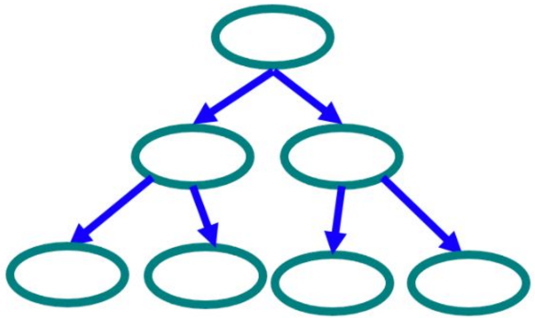
Tree Balance

Degenerate
Worst case
Search in O(n) time



Degenerate binary tree

Balanced
Mostly 2 child/node
Search in O(log(n)) time



Balanced binary tree

Tree Balance

How can we keep a tree (mostly) balanced?

Self-balancing binary search trees

- AVL trees
- Red-Black trees

The approach to keeping trees balanced:

- Select an **invariant** (that keeps tree balanced)
- Adjust (fix) the tree after each insertion/deletion (for example, maintain invariant using rotations)

These provide operations with O(log(n)) worst case

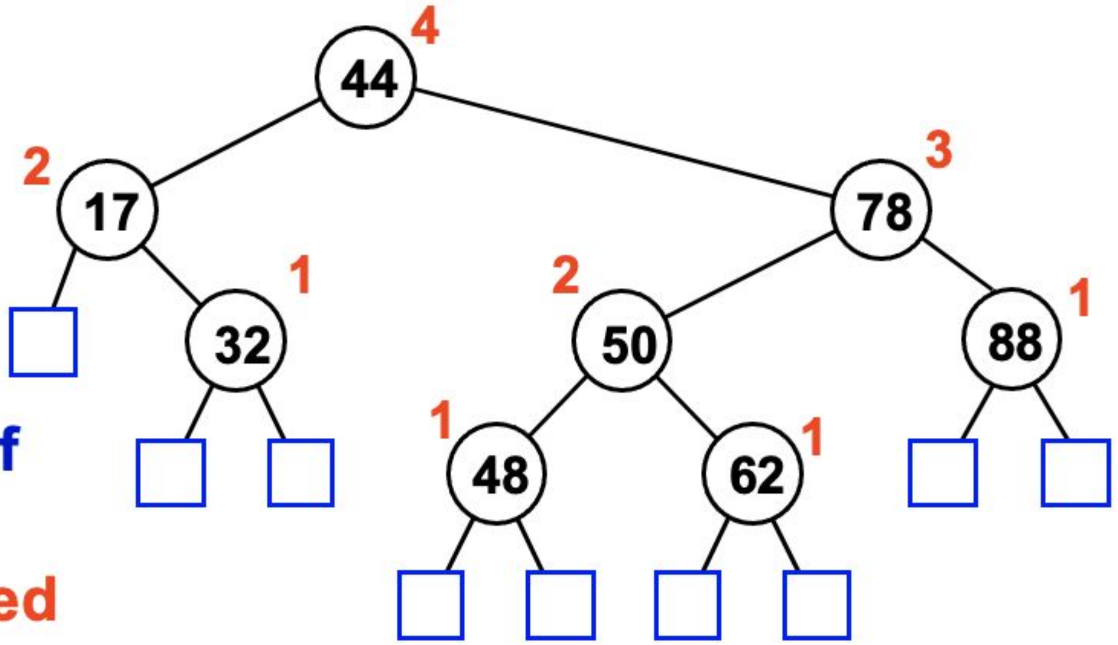
AVL Trees

Properties of an AVL Tree

- Self-balancing Binary Search Tree
- Difference between **heights** of left and right subtrees cannot be more than one for all nodes (***the invariant***)
- After each insertion or deletion the tree is checked for balance and rotated if necessary to maintain the AVL property

AVL Trees

Example AVL Tree



Heights of children shown in red

Maintaining AVL Trees

Insertion

- Insert the node the same way we insert into a standard binary search tree
- After insertion, update the heights of all of the nodes starting at the inserted node up to the root
- Calculate the **balance factor** for each node (balance factor = height of left subtree - height of right subtree)
- If any node has a balance factor of greater than 1 or less than -1 it is unbalanced.
- Restore balance with **rotations**

Maintaining AVL Trees

Insertion - Rotations

- Left rotation - when a node has a balance factor less than -1
  - Node A is unbalanced with a right child of Node B
  - A becomes the left child of B
  - The left subtree of B becomes the right subtree of A
- Right rotation - when a node has a balance factor greater than 1
  - Node A is unbalanced with a left child of Node B
  - A becomes the right child of B
  - The right subtree of B becomes the left subtree of A

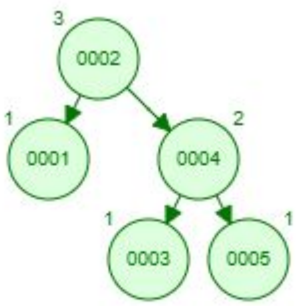
Use left-right and right-left rotation combinations when a node's child has the imbalance.

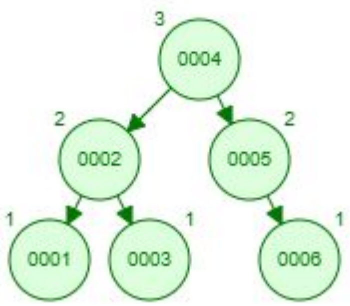
AVL Trees - Insertion Example

Insert the values 1, 2, 3, 4, 5, 6 into an AVL Tree

AVL Trees Visualization

[AVL Tree Visualizer]





Red-Black Trees

Properties of a Red-Black Tree

- Self-balancing Binary Search Tree
- Maintain Balance with invariants:
  - Each node is red or black
  - The root is always black
  - There are no two adjacent red nodes (a red node cannot have a red parent or red child)
  - Every path from a node (including the root) to any descendant NULL leaves must have the same number of black nodes
  - Every NULL leaf must be colored black
- After each insertion or deletion the tree is checked and color changes are performed along with rotations to maintain balance

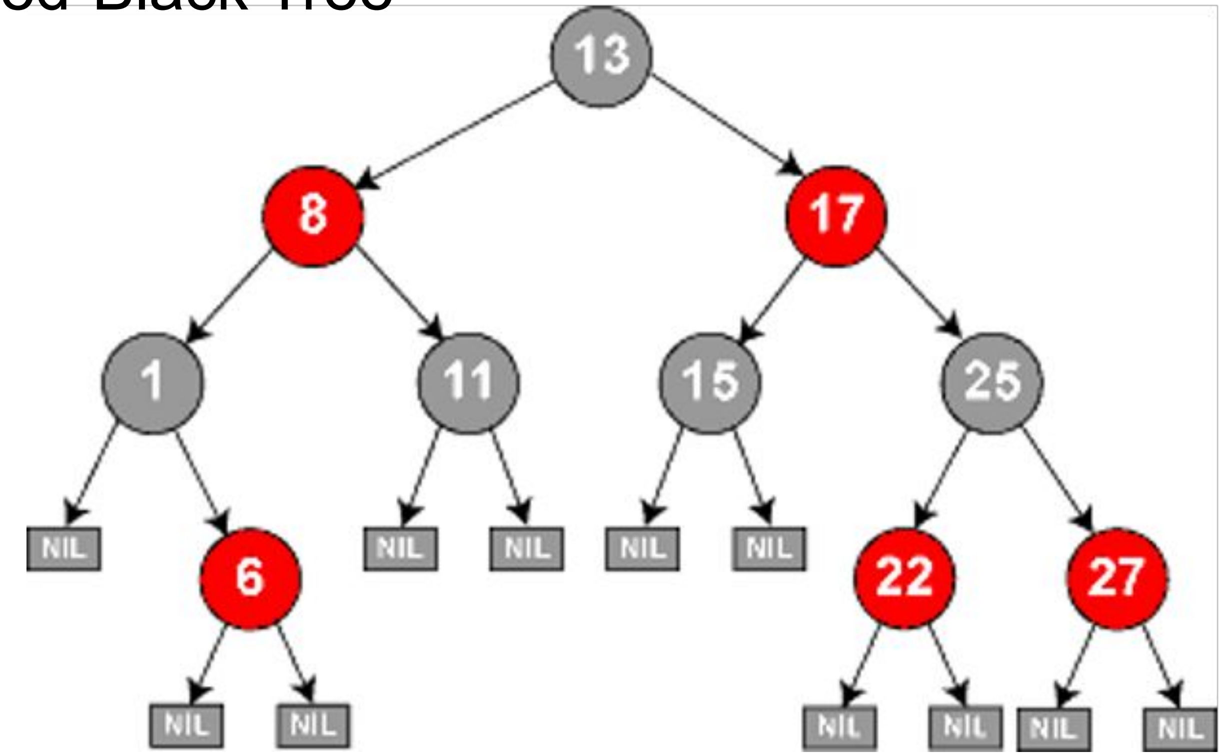
Red-Black Trees

Insertion

- Insert the node the same way we insert into a standard binary search tree, assign the new node the color red
- Check for violations of the red-black tree properties
  - If parent of the new node is black, no changes
  - If parent and the aunt/uncle node are red
    - Recolor parent and aunt/uncle to black, recolor grandparent to red
    - Continue up the tree and repeat
  - If the parent is red and the aunt/uncle are black
    - Perform rotations and recolor to fix
  - Ensure that the root node is black

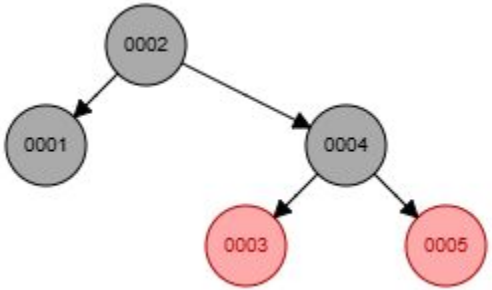
Red-Black Trees

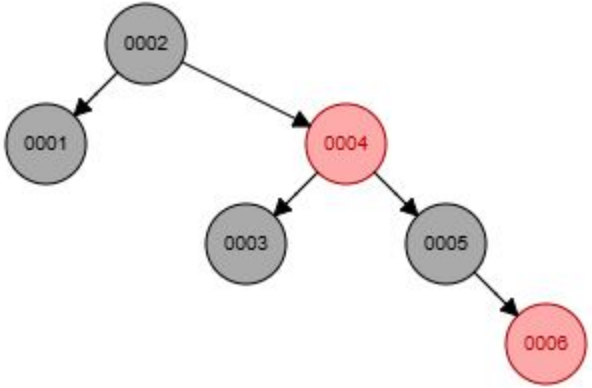
Example Red-Black Tree



Red-Black Trees Visualization

Red-Black Tree Visualizer



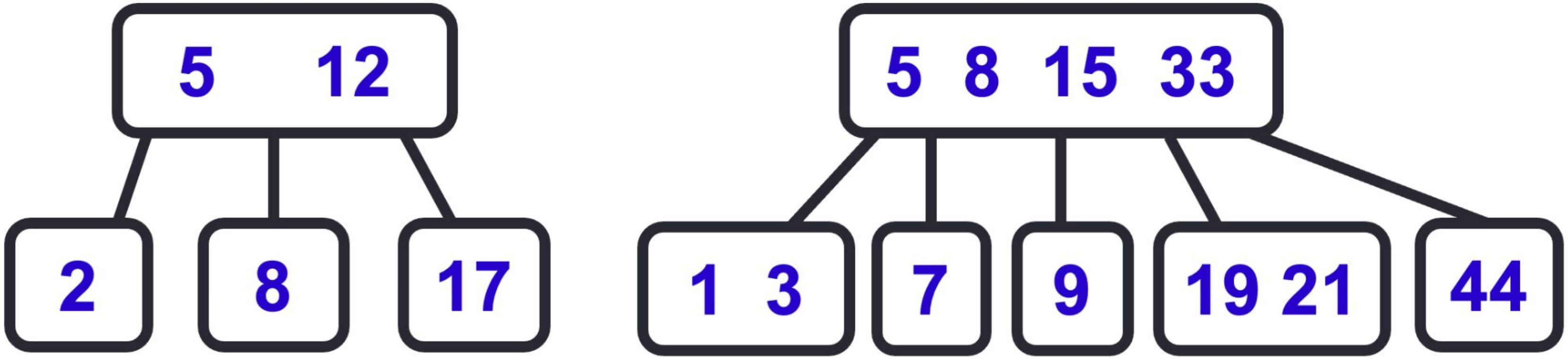


Multi-Way Search Trees

Multi-way Search Trees

Properties

- Generalization of a 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 $(j+1)^{th}$ child



Types of Multi-way Search Trees

2-3 Tree

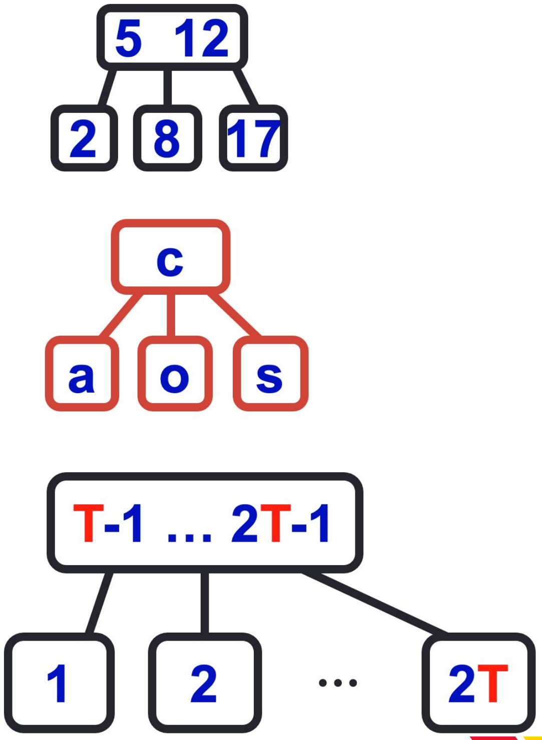
- Internal nodes have 2 or 3 children

Indexed Search Tree (Trie)

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

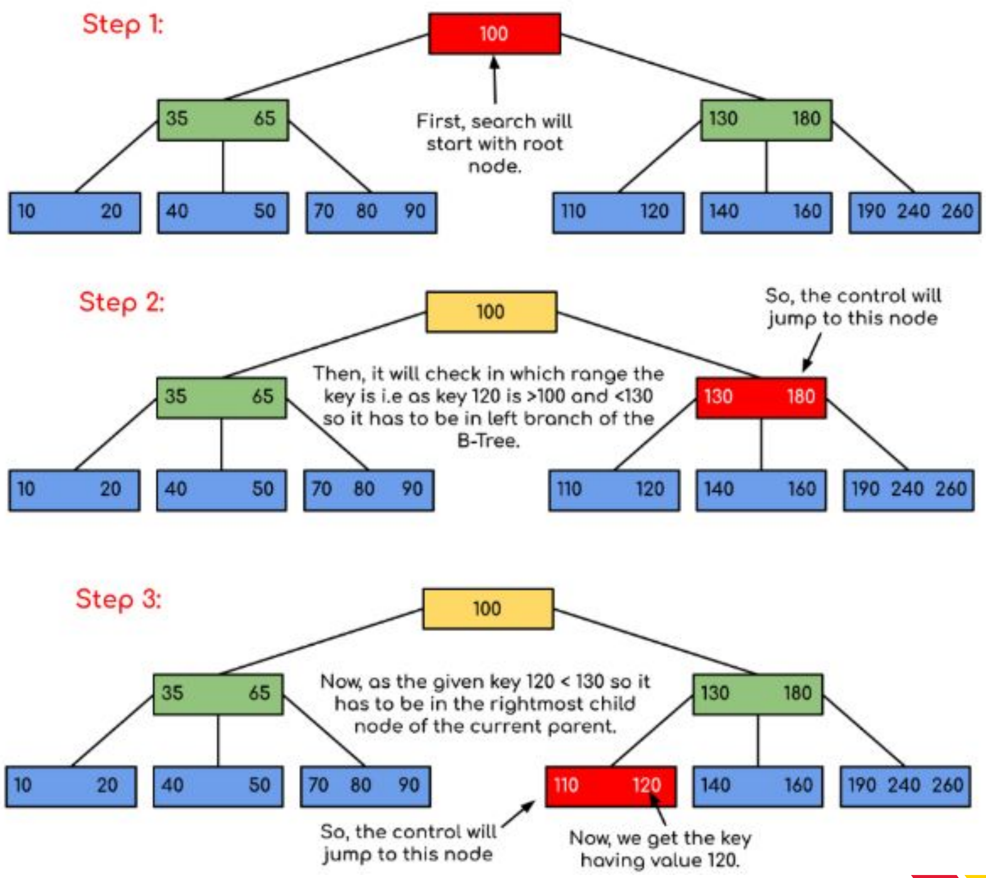
B-Tree

- T = minimum degree
- Height of tree is $O(\log_T(n))$
- All leaves have the same depth
- Popular for large database indices
  - 1 node = 1 disk block



Example B-Tree

A B Tree of order m has m or fewer children at each node.
All nodes except for the root and leaf node have at least m/2 children.
The level of each leaf node must be the same.



Indexed Search Trees - Tries

Indexed Search Tree (Trie)

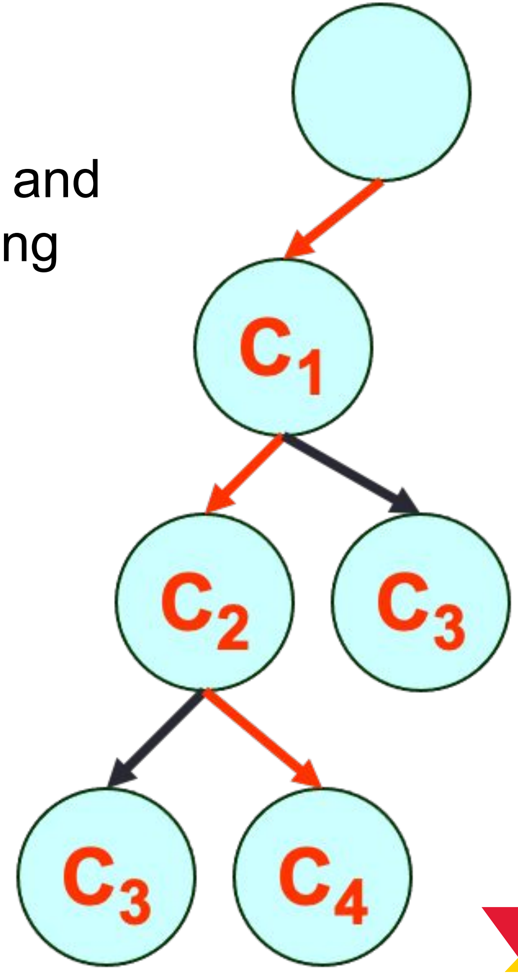
A Trie is suited for managing strings and sequences and is used for tasks such as autocomplete, spell checking and IP routing.

Special case tree, applicable when:

- Key C can be decomposed into a sequence of subkeys C_1, C_2, … C_n
- Redundancy exists between subkeys
- Common prefixes of keys are only stored once

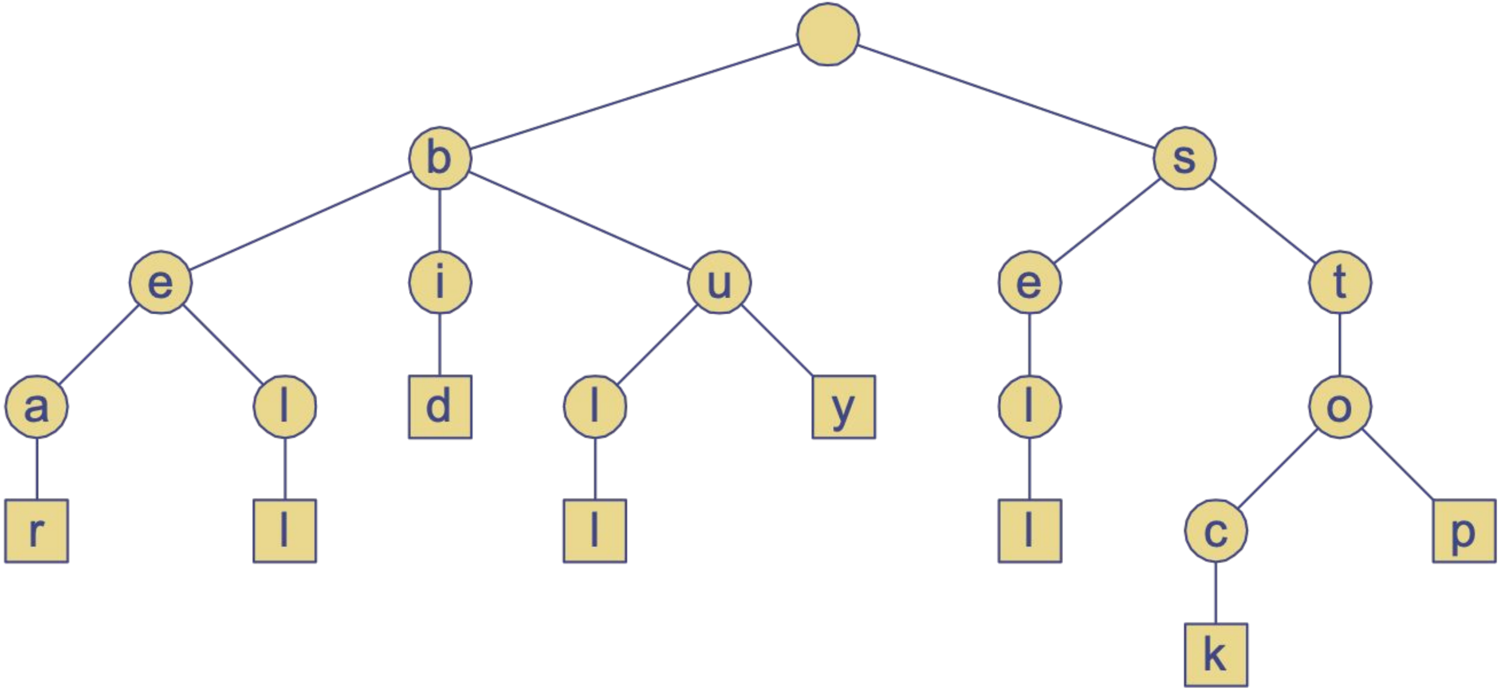
Approach

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



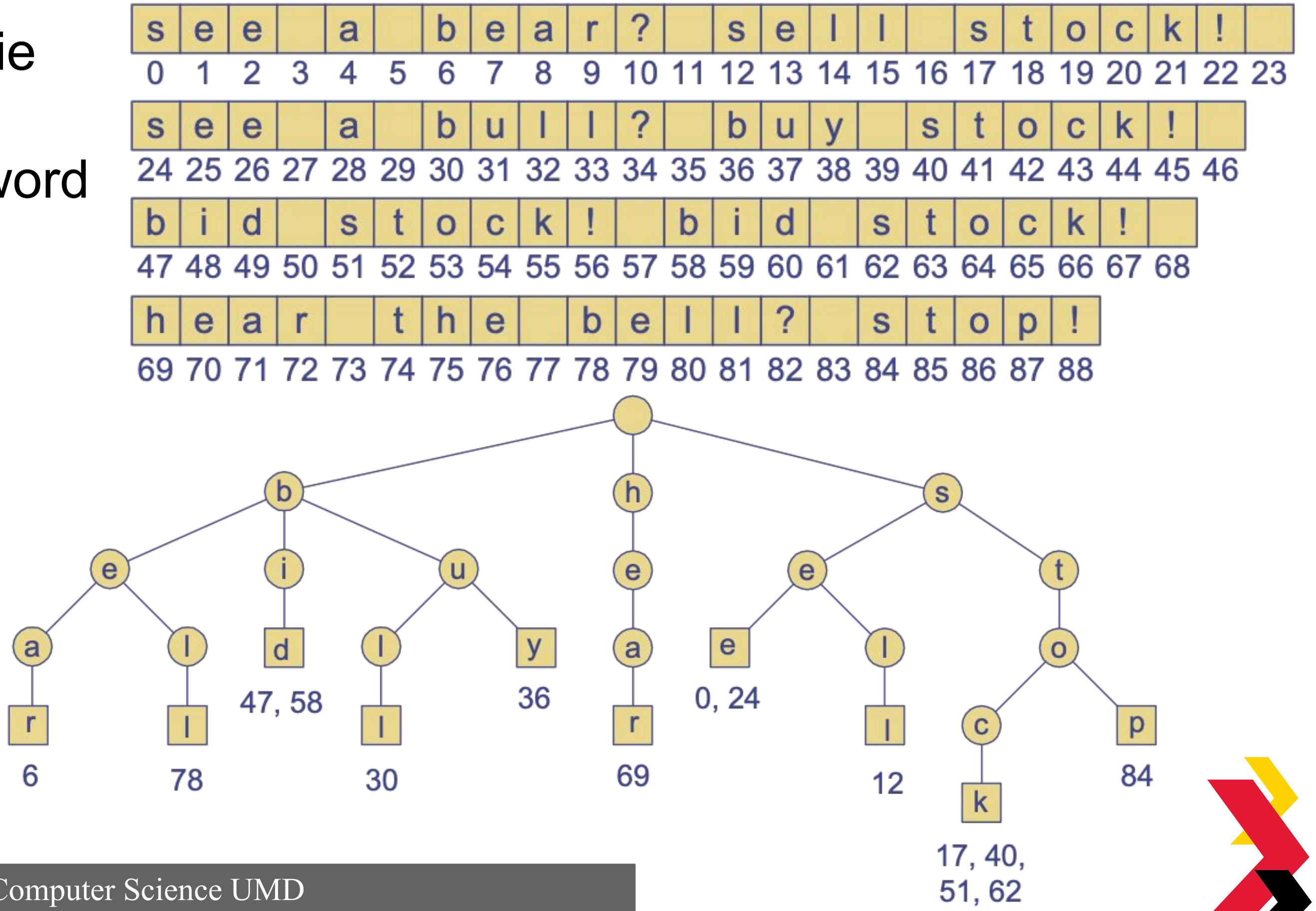
Standard Trie Example

For strings { bear, bell, bid, bull, buy, sell, stock, stop }



Word Location Trie

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



Compressed Trie

Observation

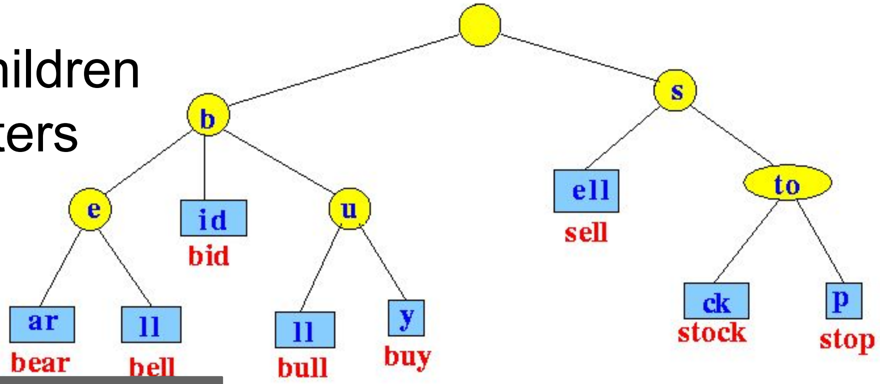
- Internal node v of T is redundant if v has one child and is not the root

Approach

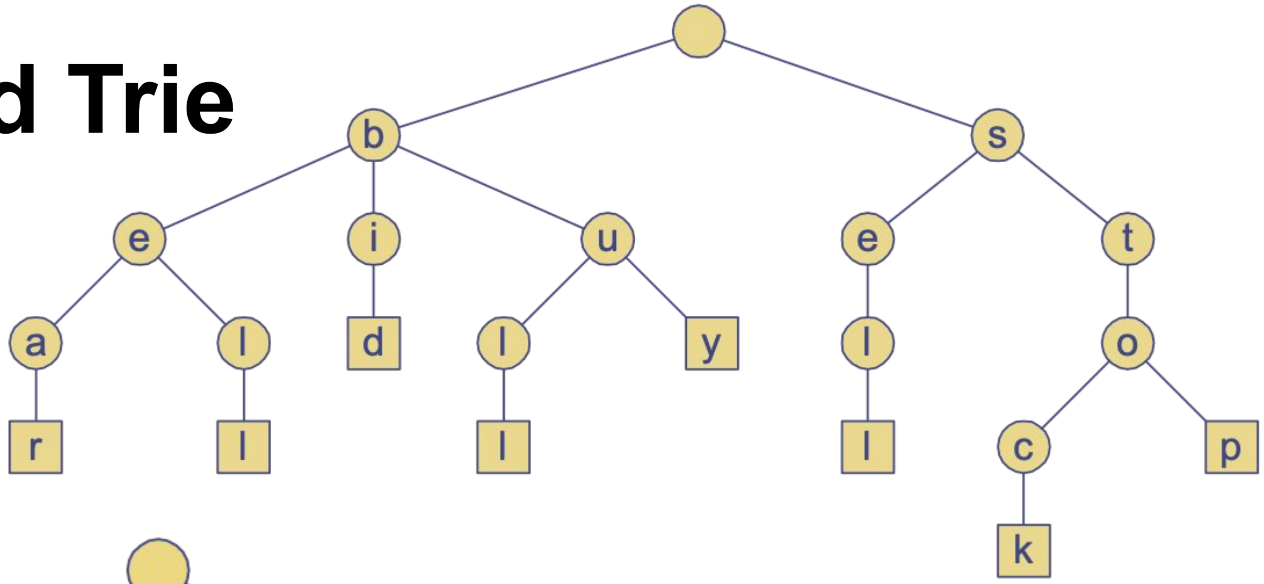
- A chain of redundant nodes can be compressed
  - Replace the chain with a single node
  - Include concatenation of labels from chain

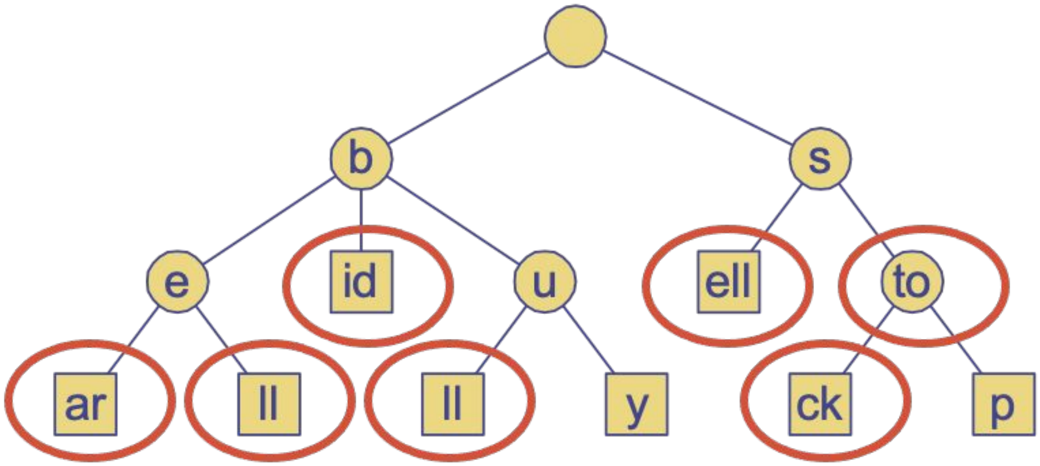
Result

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



Compressed Trie





Trie and Web Search Engines

Search Engine Index

- Collection of all searchable words
- Stored in compressed Trie

Each leaf of a Trie

- Associated with a word
- List of pages (URLs) containing that word (called an occurrence list)

Trie is kept in memory (fast access)

Occurrence lists kept in external memory

- Ranked by relevance