Overview

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

Tree Balance

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

Degenerate binary tree

Balanced binary tree

AVL Trees

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

Heights of children shown in red

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)
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

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

Tree Rotations

- Changes shape of tree
  - Move nodes
  - Change edges

- Types
  - Single rotation
    - Left
    - Right
  - Double rotation
    - Left-right
    - Right-left

Tree Rotation Example

- Single right rotation

Tree Rotation Example

- Single right rotation
  - Node 4 attached to new parent
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 jth child < jth key < keys in (j+1)th child
- **Examples**

```
   5     12
   /     /   \
  2     8    17
```

```
   5     8     15     33
   /     /  \     /   \   \
  1     3     9     19    21
```

Multi-way Search Trees

- **Search algorithm**
  1. Compare key x to 1...k keys in node
  2. If x = some key then return node
  3. Else if (x < key j) search child j
  4. Else if (x > all keys) search child k+1
- **Example**
  - Search(17)

```
   5     12
   /     /   \
  2     8    17
```

```
   5     12
   /     /   \
  2     8    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

```
   5     12
   /     /   \
  2     8    17
```

```
   5     12
   /     /   \
  2     8    17
```

Multi-way Search Trees

- **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

```
   5     12
   /     /   \
  2     8    17
```

```
   5     12
   /     /   \
  2     8    17
```

```
   5     12
   /     /   \
  2     8    17
```

```
   5     12
   /     /   \
  2     8    17
```

```
   5     12
   /     /   \
  2     8    17
```

Multi-way Search Trees

```
   5     12
   /     /   \
  2     8    17
```

```
   5     12
   /     /   \
  2     8    17
```

```
   5     12
   /     /   \
  2     8    17
```

```
   5     12
   /     /   \
  2     8    17
```
Multi-way Search Trees

- Insert Example (for 2-3 tree)
  - Insert( 4 )

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

Indexed Search Tree (Trie)

- Special case of tree
- Applicable when
  - Key $C$ can be decomposed into a sequence of subkeys $C_1, C_2, \ldots, C_n$
  - Redundancy exists between subkeys

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

- Example
  - Huffman tree

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

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
**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

**Standard Trie Example**

**For strings**
- \{ a, an, and, any, at \}

**Trie for Morse Code**

**Standard Tries**

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

**Standard Tries**

**Node structure**
- Value between 1...m
- Reference to \( m \) children
- Array or linked list

**Example**

```java
Class Node {
    Letter value; // Letter V = \{ V_1, V_2, ... V_m \}
    Node child[ m ];
}
```

**Word Matching Trie**

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

**Pointer fields**

```
Information field V_1 V_2 V_3 ... V_m
```
**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

**Compact Tries**

- **Compact representation of a compressed trie**

- **Approach**
  - For an array of strings $S = S[0], \ldots, S[s-1]$
  - 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

**Suffix Trie**

- **Compressed trie of all suffixes of text**

- **Example: “IPDPS”**
  - Suffixes
    - IPDPS
    - PDPS
    - DPS
    - PS
    - S

- **Useful for finding pattern in any part of text**
  - Occurrence $\Rightarrow$ prefix of some suffix
  - Example: find PDP in IPDPS

**Suffix Trie**

- **Properties**
  - For
    - String $X$ with length $n$
    - Alphabet of size $m$
    - Pattern $P$ with length $d$
  - Uses $O(n)$ space
  - Can be constructed in $O(n)$ time
  - Find pattern $P$ in $X$ in $O(d \times m)$ time
  - Proportional to length of pattern, not text
### 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

### 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

### 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)