CS61B Lecture #30

Today:

• More balanced search structures (DS(IJ), Chapter 9

Coming Up:

• Pseudo-random Numbers (DS(IJ), Chapter 11)
Really Efficient Use of Keys: the Trie

- Have been silent about cost of comparisons.
- For strings, worst case is length of string.
- Therefore should throw extra factor of key length, $L$, into costs:
  - $\Theta(M)$ comparisons really means $\Theta(ML)$ operations.
  - So to look for key $X$, keep looking at same chars of $X$ $M$ times.
- Can we do better? Can we get search cost to be $O(L)$?

**Idea:** Make a multi-way decision tree, with one decision per character of key.
The Trie: Example

- Set of keys
  
  \{a, abase, abash, abate, abbas, axolotl, axe, fabric, facet\}

- Ticked lines show paths followed for “abash” and “fabric”

- Each internal node corresponds to a possible prefix.

- Characters in path to node = that prefix.
Adding Item to a Trie

• Result of adding **bat** and **faceplate**.

• New edges ticked.

![Trie Diagram]

A Side-Trip: Scrunching

- For speed, obvious implementation for internal nodes is array indexed by character.
- Gives $O(L)$ performance, $L$ length of search key.
- [Looks as if independent of $N$, number of keys. Is there a dependence?]
- Problem: arrays are sparsely populated by non-null values—waste of space.

Idea: Put the arrays on top of each other!

- Use null (0, empty) entries of one array to hold non-null elements of another.
- Use extra markers to tell which entries belong to which array.
Scrunching Example

Small example: (unrelated to Tries on preceding slides)

- Three leaf arrays, each indexed 0..9

\[\begin{array}{c}
A1: \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{bass} & \text{trout} & \text{pike} \\
A2: \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{ghee} & \text{milk} & \text{oil} \\
A3: \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{salt} & \text{cumin} & \text{mace} \\
\end{array}\]

- Now overlay them, but keep track of original index of each item:

\[\begin{array}{c}
A1: \\
0^* & 1 & 2 & 3 & 4 & 5^* & 6 & 7 & 8 & 9 \\
A2: \\
0 & 1^* & 2 & 3 & 4 & 5 & 6^* & 7^* & 8 & 9 \\
A3: \\
0 & 1 & 2^* & 3 & 4 & 5^* & 6 & 7^* & 8 & 9 \\
\end{array}\]
Probabilistic Balancing: Skip Lists

- A skip list can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:

![Skip List Example](image)

- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about $1/2$ as many nodes that are $\geq k$ high as there are that are $k$ high.
- Makes searches fast with high probability.
Probabilistic Balancing: Skip Lists

- A skip list can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:

  \[ \begin{array}{cccccccccccccc}
  \infty & 10 & 20 & 25 & 30 & 40 & 50 & 55 & 60 & 90 & 95 & 100 & 115 & 120 & 125 & 130 & 140 & 150 & \infty
  \end{array} \]

- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are \( k \) high as there are that are \( k \) high.
- Makes searches fast with high probability.
Probabilistic Balancing: Skip Lists

- A skip list can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:

  ![Skip List Example Diagram]

  To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.

  In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.

  Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are \( > k \) high as there are that are \( k \) high.

  Makes searches fast with high probability.
Probabilistic Balancing: Skip Lists

- A skip list can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:

```
∞       →  120  125  130  135  140  145  150
∞       →  110  115  120  125  130  135  140
∞       →  100  105  110  115  120  125  130
∞       →  90   95   100  105  110  115  120
∞       →  80   85   90   95   100  105  110
∞       →  70   75   80   85   90   95   100
∞       →  60   65   70   75   80   85   90
∞       →  50   55   60   65   70   75   80
∞       →  40   45   50   55   60   65   70
∞       →  30   35   40   45   50   55   60
∞       →  20   25   30   35   40   45   50
∞       →  10   15   20   25   30   35   40
∞       →   0   15   20   25   30   35   40
```

- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $\geq \ell$ high as there are that are $\ell$ high.
- Makes searches fast with high probability.
A skip list can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.

More often thought of as an ordered list in which one can skip large segments.

Typical example:

To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.

In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.

Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are \( > k \) high as there are that are \( k \) high.

Makes searches fast with high probability.
Probabilistic Balancing: Skip Lists

- A skip list can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.

- More often thought of as an ordered list in which one can skip large segments.

- Typical example:

  ![Diagram of a skip list]

  To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.

  In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.

  Heights of the nodes were chosen randomly so that there are about $1/2$ as many nodes that are $> k$ high as there are that are $k$ high.

  Makes searches fast with high probability.
Probabilistic Balancing: Skip Lists

- A skip list can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.

- More often thought of as an ordered list in which one can skip large segments.

- Typical example:

  ![Skip List Diagram]

  - To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
  - In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
  - Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are \( \geq k \) high as there are that are \( k \) high.
  - Makes searches fast with high probability.
Probabilistic Balancing: Skip Lists

- A skip list can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.

- More often thought of as an ordered list in which one can skip large segments.

- Typical example:

  ![Skip List Example](image)

  - To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.

  - In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.

  - Heights of the nodes were chosen randomly so that there are about $\frac{1}{2}$ as many nodes that are $> k$ high as there are that are $k$ high.

  - Makes searches fast with high probability.
Probabilistic Balancing: Skip Lists

- A **skip list** can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:

  ![Diagram](image)

  - To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
  - In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
  - Heights of the nodes were chosen randomly so that there are about $1/2$ as many nodes that are $> k$ high as there are that are $k$ high.
  - Makes searches fast with high probability.
Probabilistic Balancing: Skip Lists

- A skip list can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.

- More often thought of as an ordered list in which one can skip large segments.

- Typical example:

```
\[ 0 \rightarrow 10 \rightarrow 20 \rightarrow 25 \rightarrow 30 \rightarrow 40 \rightarrow 50 \rightarrow 55 \rightarrow 60 \rightarrow 90 \rightarrow 95 \rightarrow 100 \rightarrow 115 \rightarrow 120 \rightarrow 125 \rightarrow 130 \rightarrow 140 \rightarrow 150 \rightarrow \infty \]
```

- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.

- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.

- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are \( k \) high as there are that are \( k \) high.

- Makes searches fast with high probability.
Probabilistic Balancing: Skip Lists

• A skip list can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.

• More often thought of as an ordered list in which one can skip large segments.

• Typical example:

```
+----------------+-----------------+-------------------+
\|                 |                 |                   |
\|                3 |                 |                   |
\|                | 2               |                   |
\|                1 |                 |                   |
\| 0              | 10              | 20 25 30 40 50 55 |
\|                | 60 90 95 100 115|
\|                | 120 125 130 140 |
\|                | 150             |                   |
\|                |                 |                   |
```

• To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.

• In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.

• Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are \( k \) high as there are that are \( k \) high.

• Makes searches fast with high probability.
Example: Adding and deleting

• Starting from initial list:

• In any order, we add 126 and 127 (choosing random heights for them), and remove 20 and 40:

• Shaded nodes here have been modified.
Summary

• Balance in search trees allows us to realize $\Theta(\lg N)$ performance.

• B-trees, red-black trees:
  - Give $\Theta(\lg N)$ performance for searches, insertions, deletions.
  - B-trees good for external storage. Large nodes minimize # of I/O operations.

• Tries:
  - Give $\Theta(B)$ performance for searches, insertions, and deletions, where $B$ is length of key being processed.
  - But hard to manage space efficiently.

• Interesting idea: scrunched arrays share space.

• Skip lists:
  - Give probable $\Theta(\lg N)$ performance for searches, insertions, deletions.
  - Easy to implement.
  - Presented for interesting ideas: probabilistic balance, randomized data structures.
Summary of Collection Abstractions

- **Multiset**
  - contains, iterator

- **List**
  - get(n)

- **Set**
  - **Ordered Set**
    - first
  - **Unordered Set**

- **Priority Queue**

- **Map**
  - contains, iterator
  - get

- **Unordered Map**
- **Ordered Map**

Blue: Java has corresponding interface
Green: Java has no corresponding interface

Data Structures that Implement Abstractions

**Multiset**

- **List**: arrays, linked lists, circular buffers
- **Set**
  - **OrderedSet**
    - *Priority Queue*: heaps
    - *Sorted Set*: binary search trees, red-black trees, B-trees, sorted arrays or linked lists
  - **Unordered Set**: hash table

**Map**

- **Unordered Map**: hash table
- **Ordered Map**: red-black trees, B-trees, sorted arrays or linked lists
**Corresponding Classes in Java**

**Multiset** (Collection)

- **List**: ArrayList, LinkedList, Stack, ArrayBlockingQueue, ArrayDeque
- **Set**
  - **OrderedSet**
    - *Priority Queue*: PriorityQueue
    - *Sorted Set (SortedSet)*: TreeSet
  - **Unordered Set**: HashSet

**Map**

- **Unordered Map**: HashMap
- **Ordered Map (SortedMap)**: TreeMap