CS61B Lecture #31

Today:
- More balanced search structures \((DS(IJ),\) Chapter 9

Coming Up:
- Pseudo-random Numbers \((DS(IJ),\) Chapter 11

Last modified: Thu Nov 1 19:39:39 2018
Really Efficient Use of Keys: the Trie

• Haven’t said much 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.
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

A1: 0 1 2 3 4 5 6 7 8 9
    bass trout pike

A2: 0 1 2 3 4 5 6 7 8 9
    ghee milk oil

A3: 0 1 2 3 4 5 6 7 8 9
    salt cumin mace

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

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

A123: 0 -1 1 -1 2 5 5 7 6 7 9

bass trout pike ghee milk oil mace
Practicum

- The scrunching idea is cute, but
  - Not so good if we want to expand our trie.
  - A bit complicated.
  - Actually more useful for representing large, sparse, fixed tables with many rows and columns.

- Furthermore, number of children in trie tends to drop drastically when one gets a few levels down from the root.

- So in practice, might as well use linked lists to represent set of node’s children...

- ...but use arrays for the first few levels, which are likely to have more children.
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](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:

![Diagram of a skip list with keys 0 to 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.*
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](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:**

- 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:

  ![Skip List 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:

  ![Skip List 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 \( \frac{1}{2} \) 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:

  ![Skip List 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:

  ![Skip List 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:

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

- 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 $> \kappa$ high as there are that are $\kappa$ 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:

  - 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 1 2 3 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:

```
−∞  10  20  25  30  40  50  55  60  90  95  100  115  120  125  130  140  150  ∞
```

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

```
−∞  10  25  30  50  55  60  90  95  100  115  120  125  130  140  150  ∞
```

• 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**
  - **Sorted Set**: `subset`
- **Map**: contains, iterator
  - **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