

Efficient Use of Keys: the Trie

much about cost of comparisons.

worst case is length of string.

ould throw extra factor of key length, L , into costs:

comparisons really means $\Theta(ML)$ operations.

for key X , keep looking at same chars of X M times.

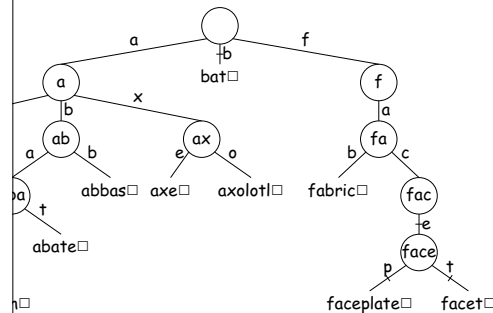
etter? Can we get search cost to be $O(L)$?

multi-way decision tree, with one decision per character

Adding Item to a Trie

Adding bat and faceplate.

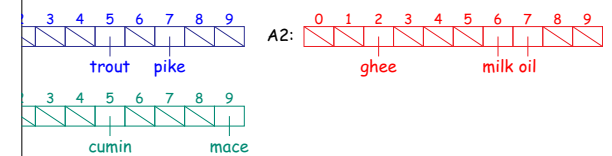
icked.



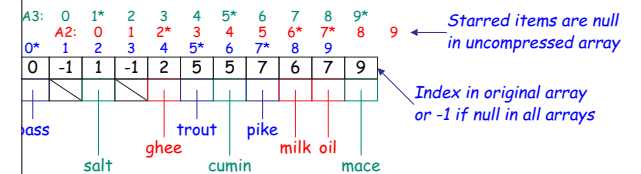
Scrunching Example

(unrelated to Tries on preceding slides)

arrays, each indexed 0..9



them, but keep track of the original index of each item:



CS61B Lecture #31

ed search structures (*DS(IJ)*, Chapter 9

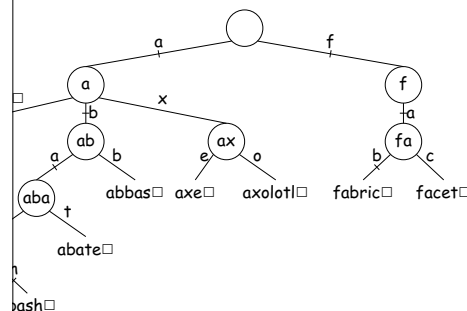
The Trie: Example

abash, abate, abbas, axolotl, axe, fabric, facet}

show paths followed for "abash" and "fabric"

Each node corresponds to a possible prefix.

Each path to node = that prefix.



A Side-Trip: Scrunching

obvious implementation for internal nodes is array in character.

performance, L length of search key.

independent of N , number of keys. Is there a depen-

arrays are *sparsely populated* by non-null values—waste of

arrays on top of each other!

(empty) entries of one array to hold non-null elements of

markers to tell which entries belong to which array.

Practicum

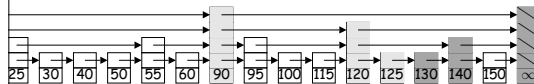
ing idea is cute, but
 od if we want to expand our trie.
 plicated.
 ore useful for representing large, sparse, fixed tables
 rows and columns.
 y, number of children in trie tends to drop drastically
 s a few levels down from the root.
 ce, might as well use linked lists to represent set of
 en...
 rrays for the first few levels, which are likely to have
 n.

7:16 2021

CS61B: Lecture #31 8

Probabilistic Balancing: Skip Lists

an be thought of as a kind of n-ary search tree in which
 put the keys at "random" heights.
 thought of as an ordered list in which one can skip large
 ple:



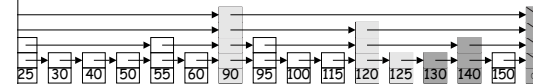
start at top layer on left, search until next step would
 hen go down one layer and repeat.
 , we search for 125 and 127. Gray nodes are looked at;
 nodes are overshoots.
 he nodes were chosen randomly so that there are about
 nodes that are $> k$ high as there are that are k high.
 hes fast *with high probability*.

7:16 2021

CS61B: Lecture #31 10

Probabilistic Balancing: Skip Lists

an be thought of as a kind of n-ary search tree in which
 put the keys at "random" heights.
 thought of as an ordered list in which one can skip large
 ple:



start at top layer on left, search until next step would
 hen go down one layer and repeat.
 , we search for 125 and 127. Gray nodes are looked at;
 nodes are overshoots.
 he nodes were chosen randomly so that there are about
 nodes that are $> k$ high as there are that are k high.
 hes fast *with high probability*.

7:16 2021

CS61B: Lecture #31 12

Scrunching Example (contd.)

Diagram illustrating the scrunching example. It shows an array A2 with indices 0-9 and values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The array is partitioned into segments: trout (indices 1-2), pike (indices 3-4), ghee (indices 5-6), and milk oil (indices 7-8). Below this, another array is shown with values: -1, 1, -1, 2, 5, 5, 7, 6, 7, 9. This array is also partitioned into segments: salt (indices 1-2), ghee (indices 3-4), cumin (indices 5-6), pike (indices 7-8), milk oil (indices 9-10), and mace (indices 11-12). A note indicates: "Starred items are null in uncompressed array".

```

    0 1* 2 3 4 5* 6 7 8 9*
    A2: 0 1 2* 3 4 5 6* 7* 8 9
    * 1 2 3 4 5* 6 7* 8 9
    ) -1 1 -1 2 5 5 7 6 7 9
    ss | | | | | | | | | |
       salt | ghee | cumin | pike | milk oil | mace
    */ (Check[i] == i) ? A123[i] : null;
    */ (Check[i + 2] == i) ? A123[i + 2] : null;
    */ (Check[i + 1] == i) ? A123[i + 1] : null;
  
```

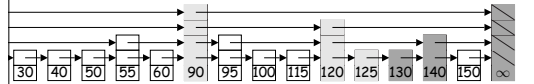
Index in original array or -1 if null in all arrays

7:16 2021

CS61B: Lecture #31 7

Probabilistic Balancing: Skip Lists

an be thought of as a kind of n-ary search tree in which
 put the keys at "random" heights.
 thought of as an ordered list in which one can skip large
 ple:



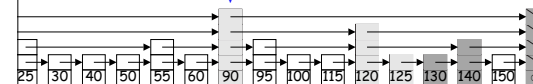
start at top layer on left, search until next step would
 hen go down one layer and repeat.
 , we search for 125 and 127. Gray nodes are looked at;
 nodes are overshoots.
 he nodes were chosen randomly so that there are about
 nodes that are $> k$ high as there are that are k high.
 hes fast *with high probability*.

7:16 2021

CS61B: Lecture #31 9

Probabilistic Balancing: Skip Lists

an be thought of as a kind of n-ary search tree in which
 put the keys at "random" heights.
 thought of as an ordered list in which one can skip large
 ple:



start at top layer on left, search until next step would
 hen go down one layer and repeat.
 , we search for 125 and 127. Gray nodes are looked at;
 nodes are overshoots.
 he nodes were chosen randomly so that there are about
 nodes that are $> k$ high as there are that are k high.
 hes fast *with high probability*.

7:16 2021

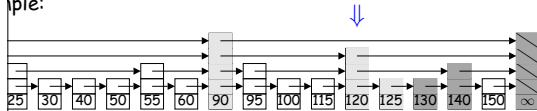
CS61B: Lecture #31 11

Probabilistic Balancing: Skip Lists

can be thought of as a kind of n -ary search tree in which we put the keys at "random" heights.

can also be thought of as an ordered list in which one can skip large

example:



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

for example, we search for 125 and 127. Gray nodes are looked at; nodes that are overshoots.

the nodes were chosen randomly so that there are about k nodes that are $> k$ high as there are that are k high.

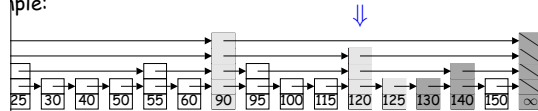
searches fast with high probability.

Probabilistic Balancing: Skip Lists

can be thought of as a kind of n -ary search tree in which we put the keys at "random" heights.

can also be thought of as an ordered list in which one can skip large

example:



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

for example, we search for 125 and 127. Gray nodes are looked at; nodes that are overshoots.

the nodes were chosen randomly so that there are about k nodes that are $> k$ high as there are that are k high.

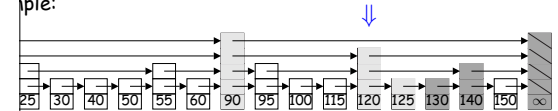
searches fast with high probability.

Probabilistic Balancing: Skip Lists

can be thought of as a kind of n -ary search tree in which we put the keys at "random" heights.

can also be thought of as an ordered list in which one can skip large

example:



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

for example, we search for 125 and 127. Gray nodes are looked at; nodes that are overshoots.

the nodes were chosen randomly so that there are about k nodes that are $> k$ high as there are that are k high.

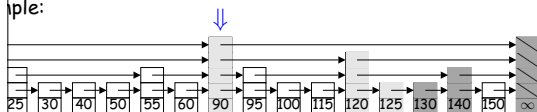
searches fast with high probability.

Probabilistic Balancing: Skip Lists

can be thought of as a kind of n -ary search tree in which we put the keys at "random" heights.

can also be thought of as an ordered list in which one can skip large

example:



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

for example, we search for 125 and 127. Gray nodes are looked at; nodes that are overshoots.

the nodes were chosen randomly so that there are about k nodes that are $> k$ high as there are that are k high.

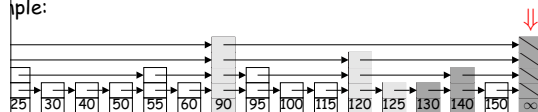
searches fast with high probability.

Probabilistic Balancing: Skip Lists

can be thought of as a kind of n -ary search tree in which we put the keys at "random" heights.

can also be thought of as an ordered list in which one can skip large

example:



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

for example, we search for 125 and 127. Gray nodes are looked at; nodes that are overshoots.

the nodes were chosen randomly so that there are about k nodes that are $> k$ high as there are that are k high.

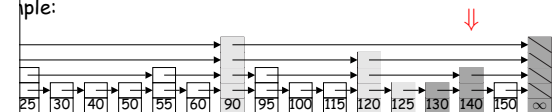
searches fast with high probability.

Probabilistic Balancing: Skip Lists

can be thought of as a kind of n -ary search tree in which we put the keys at "random" heights.

can also be thought of as an ordered list in which one can skip large

example:



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

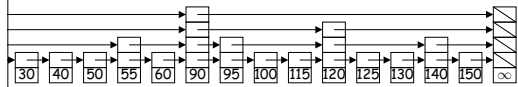
for example, we search for 125 and 127. Gray nodes are looked at; nodes that are overshoots.

the nodes were chosen randomly so that there are about k nodes that are $> k$ high as there are that are k high.

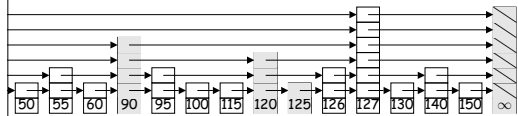
searches fast with high probability.

Example: Adding and deleting

Initial list:

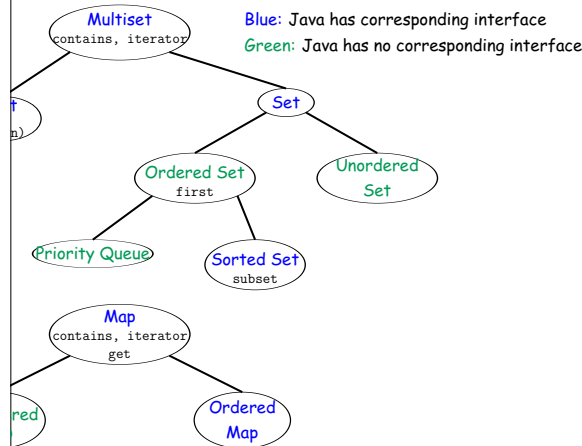


Now, we add 126 and 127 (choosing random heights for remove 20 and 40):



Nodes here have been modified.

Summary of Collection Abstractions



Corresponding Classes in Java

Queue: PriorityQueue, LinkedList, Stack, ArrayBlockingQueue, ...

Set (SortedSet): TreeSet
Unordered Set: HashSet

Map: HashMap
Ordered Map (SortedMap): TreeMap

Probabilistic Balancing: Skip Lists

Can be thought of as a kind of n -ary search tree in which we put the keys at "random" heights.

Can also be thought of as an ordered list in which one can skip large sections.

Example:



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

For example, we search for 125 and 127. Gray nodes are looked at; white nodes are overshoots.

Because the nodes were chosen randomly so that there are about k nodes that are $> k$ high as there are that are k high.

This makes it very fast with high probability.

Summary

B-trees allow us to realize $\Theta(\lg N)$ performance.

Red-black trees:

$\Theta(\lg N)$ performance for searches, insertions, deletions. Good for external storage. Large nodes minimize # of nodes.

Hash tables: $\Theta(1)$ performance for searches, insertions, and deletions, independent of length of key being processed.

Useful to manage space efficiently.

Idea: scrunched arrays share space.

Hash tables: $\Theta(1)$ performance for searches, insertions, deletions.

Implement.

Look for **interesting ideas**: probabilistic balance, random structures.

Structures that Implement Abstractions

Queue: linked lists, circular buffers

Set (SortedSet): binary search trees, red-black trees, B-trees, arrays or linked lists
Unordered Set: hash table

Map: hash table

Ordered Map (SortedMap): red-black trees, B-trees, sorted arrays or linked lists