CS 240 – Data Structures and Data Management

Module 6E: Dictionaries for special keys - Enriched

T. Biedl   E. Kondratovsky   M. Petrick   O. Veksler
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2022

version 2022-02-08 13:51

Outline

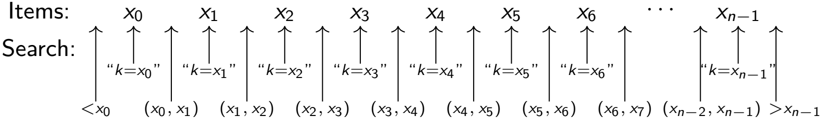
- A tighter lower bound
- Improving binary search
- More on interpolation search
- More on pruned tries

Outline

- A tighter lower bound
- Improving binary search
- More on interpolation search
- More on pruned tries

A tighter lower bound

- Create $2n + 1$ instances:



- **Claim:** These instances must lead to distinct leaves (assuming no equality-comparison).

- So we require at least $\lceil \log(2n + 1) \rceil$ comparisons.

Outline

- A tighter lower bound
- Improving binary search
- More on interpolation search
- More on pruned tries

Improving binary search

- *binary-search* uses $\approx 2\log n$ comparisons.
- **Goal:** Improve it to use $\lceil \log(2n+1) \rceil \approx \log n + 1$ comparisons.
- Main ingredient: Do only *one* comparison per round.

binary-search-optimized(A, n, k)
A: Sorted array of size n, k: key
$\ell \leftarrow 0$, $r \leftarrow n-1$, $\chi \leftarrow 0$
while $(\ell < r)$
  $m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
  if $(A[m] < k)$ **then** $\ell \leftarrow m+1$
  else $r \leftarrow m$, $\chi \leftarrow 1$ // this is different!
if $(k < A[\ell])$ **then return** "not found, between $A[\ell-1]$ and $A[\ell]$"
else if $\chi = 1$ or $(k \leq A[\ell])$ **then return** "found at $A[\ell]$"
else "not found, between $A[\ell]$ and $A[\ell+1]$"

(χ needed for optimum # of comparisons, but not normally used)

Improving binary search

- **Claim 1:** This terminates.
  Right sub-array is clearly smaller.
  If $\ell < r$, then $m \leq \frac{\ell+r}{2} < \frac{r+r}{2} < r$ so left sub-array is smaller.
- **Claim 2:** This returns correctly.
  Loop-invariant suprisingly tricky: $A[\ell-1]<k\leq A[r+1]$, plus others.
  (See textbook).
- **Claim 3:** This uses at most $\lceil \log n \rceil + 2$ comparisons.
  Sub-array has size $\leq \lceil n/2 \rceil$, so $\lceil \log n \rceil$ rounds.
  One comparison per round. At most 2 comparisons at the end.
- **Claim 4:** If χ is used, then $\#$ comparisons $\leq \lceil \log(2n+1) \rceil$.
  (Straightforward but tedious cases. See textbook for details.)

- This uses the *optimum* number of comparisons and also in practice performs better than *binary-search*.
  - But normally omit χ (only needed in Claim 4)
  - Can replace two comparisons in lines 6-7 by equality-comparison.

Outline

- A tighter lower bound
- Improving binary search
- More on interpolation search
- More on pruned tries

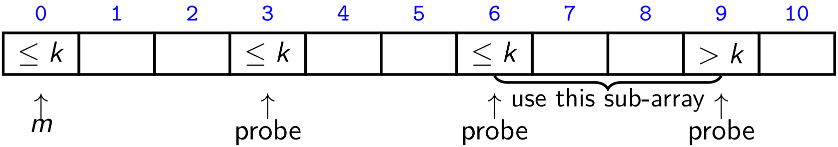
Improving Interpolation Search

- Had: Average-case run-time of *interpolation-search* is $O(\log \log n)$.
- This is very complicated to prove!

  (
  - Study error, i.e., distance between index of k and where we probed.
  - Argue that error is in $O(\sqrt{n})$ in first round.
  - Argue that error is in $O(\frac{1}{2^i} n)$ after i rounds.
  - Study the martingale formed by the errors in the rounds.
  - Argue that its expected length is $O(\log \log n)$.

)
- Instead: Define a variant of *interpolatation-search*
  - Better worst-case run-time.
  - Easier to analyze.
- Idea: *Force* the sub-array to have size $\sqrt{n}$
- To do so, search for suitable sub-array with probes.
- Crucial question: how many probes are needed?

Improving Interpolation Search



- First compare ("probe") at m as before.
- If $A[m] \leq k$, probe rightward.
- Probes always go $\lceil \sqrt{N} \rceil$ indices rightward (where $N = r - \ell - 1 \approx$ size of currently studied sub-array)
- Continue probing until $> k$ or out-of-bounds
- Recurse in the only sub-array where k can be; it has size $O(\sqrt{N})$.
- Observe: $\#$ probes $\in O(\sqrt{N})$

Improving Interpolation Search

Interpolation-search-modified(A, n, k)
A: sorted array of size n, k: key

if $(k < A[0]$ or $k > A[n-1])$ **return** "not found"
if $(k = A[n-1])$ **return** "found at index $n-1$"
$\ell \leftarrow 0$, $r \leftarrow n-1$ // have $A[\ell] \leq k < A[r]$
while $(N \leftarrow (r - \ell - 1) \geq 1)$
  $m \leftarrow \ell + \lceil \frac{k - A[\ell])}{A[r] - A[\ell]} \cdot (r - \ell - 1) \rceil$
  if $(A[m] \leq k)$ // probe rightward
    for $h = 1, 2, \ldots$
      $\ell \leftarrow m + (h-1)\lceil \sqrt{N} \rceil$, $r' \leftarrow \min\{r, m + h\lceil \sqrt{N} \rceil\}$
      if $(r' = r$ or $A[r'] > k)$ **then** $r \leftarrow r'$ and **break**
  else ... // symmetrically probe leftward
if $(k = A[\ell])$ **return** "found at index ℓ"
else return "not found"

Analysis of *interpolation-search-improved*

- $T(n) \leq T(\text{size of sub-array}) + O(\#\text{probes})$
- size of sub-array $\leq \sqrt{n} + O(1)$, $\#$ probes $\leq \sqrt{n} + O(1)$
- Use a sloppy recursion:

$$T^{\text{worst}}(n) \leq \begin{cases} c & n \leq 15 \\ T^{\text{worst}}(\sqrt{n}) + c \cdot \sqrt{n} & \text{otherwise} \end{cases}$$

- Easy induction proof: $T^{\text{worst}}(n) \leq 2c\sqrt{n}$.
- Therefore worst-case run-time is $O(\sqrt{n})$.

Analysis of *interpolation-search-improved*

- What is the number of probes on average?
- Rephrase: If numbers are chosen uniformly at random, what is the expected number of probes?
- **Claim:** Expected number of probes is $c \leq 2.5$.

Analysis of *interpolation-search-improved*

- Sloppy recursion: $T^{\text{avg}}(n) \leq \begin{cases} T^{\text{avg}}(\sqrt{n}) + c & n \geq 4 \\ c & \text{otherwise} \end{cases}$
- **Claim:** This resolves to $T^{\text{avg}}(n) \leq c\lceil \log\log n \rceil$.

Key ingredient: $\log\log\sqrt{n} \leq \lceil \log\log n \rceil - 1$.

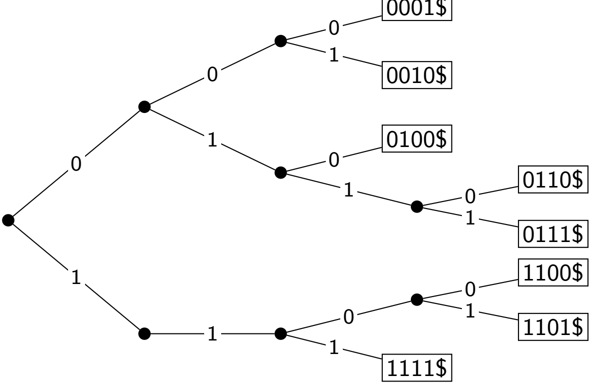
- Therefore the average-case # comparisons is $\leq 2.5\lceil \log\log n \rceil$.
- Fewer than *binary-search-optimized*'s $\lceil \log n \rceil + 1$ for $n \geq 16$.

Outline

- A tighter lower bound
- Improving binary search
- More on interpolation search
- More on pruned tries

Pruned tries and MSD-radix sort

For bitstrings: Pruned trie equals recursion tree of MSD radix-sort.



Pruned tries can store real numbers

If we have a generator for each bit of a real number, then we can store them in a pruned trie.