Finger search trees

Motivating example:
Binary search on a line.

\[ x_1 < x_2 < x_3 \ldots \]

\( X_{m+1} < X_{1000} \)

(Guess a number between 1, 1000)

- What if \( x \) likely closer to left side?
- What if list is unbounded or unknown size?

\[ x_1 \quad x_2 \quad \ldots \quad ? \]

(Guess a number \( \geq 1 \))

\[ 1 \quad 2 \quad 4 \quad 8 \quad 16 \quad \ldots \quad 2^k \]

- Chaining search

1. \( k = \lceil \log_2 d \rceil \) \( \text{search}(x) \) (how many smaller items in the list?)
2. \( 2^{k-1} < d \leq 2^k \) \( \lceil \log_2 d \rceil \) comparisons

Total: \( 2 \cdot \log_2 d \) comparisons

End of search.

\[ 1 \quad k \]

2. \( \log_2 k < \log_2 d \)
Exercise: improve \( \leq d \)

Summary (doubling search)

- Binary search in time \( O(\log d) \) instead of \( O(\log n) \)
- \( n \) items in set
- Smaller than target \( x \)
- \( \leq d \)

Doubling search viewed as Binary Search Tree

- Right spine with complete trees hanging to the left of increasing size

- Depth \( \approx 2 \log_2 n \)
- Capture “doubling search”
- How to do insert/delete?
  - Should maintain subtree-sizes as powers of 2
  - Maintain approximately (doubling/halving)
  - When node too large/small, split/merge
  (doubling, but tricky)

So far we assumed that search starts at minimum key.

Suppose we search around \( x \).

\( \Rightarrow \) Doubling search from \( x \).
Goal: Given a points $X$ (query)

- Search ($y$)

Should be efficient if "$x$ is close to $y$"

Search should take $O(\log d)$

$d$: # elements in the tree that are in $[x, y]$

Dynamic?

$O(\log d)$ not worse than $O(\log n)$

First idea

- go up from $x$ to $\text{lca}(x, y)$
- go down from $\text{lca}(x, y)$ to $y$

Two issues

- how to detect $\text{lca}(x, y)$
- even if we detect $\text{lca}(x, y)$, the path may be too long

E.g.

- query search from $x$ to $y$

$\text{lca}(x, y) = r$

$d = O(1)$: only element between $x, y$ is $r$
General approach

- use level links (contract circular doubly linked lists over nodes at each level)
- two extra pointers per node
- helps to navigate tree
- makes detection of LCA \((x, y)\) possible

Issue: how to maintain level links under insert/delete/... ?

(dible, but tricky — see it later)

- search, insert, delete take \(O(\log d)\) time
  - \# items between \(x, y\)
  - layer recorded/rooted/deleted

- additionally: split/join

Applications

1. An adaptive heap

- \(\text{insert}(x)\)
- \(\text{extract-min}\)

\(O(\log d)\) time \{ amortized \}
\(O(1)\) time

- \(d \leq m\)
  - \(d\) : rank of inserted element \(x\)
  - \(#\) elements in heap will keep smaller than \(x\)

- \(d \leq m\)
  - most very efficient for elements close to \(\text{min}\)

Implementation

- fager search tree, \(\text{min}\) pointer (finger) at the minimum.
• Extract-min
  • Search for successor/min of min
    \( \text{ offline search takes } O(1) \text{ time} \)
  • Delete/return min \( O(1) \)
  • Set node points to new minimum min
  • Insert \( x \)
    • Insert from
      \( \text{ merge } \) \( \text{ node } \)
      \( O(lg d) \)
    
    \( \text{ rand } (i) = \# \text{ elements in tree between min and } x \)

2. Merge two sorted lists

\[
\begin{array}{c}
x \quad \overline{x_1 \ldots x_m} \quad \text{size } m \\
\uparrow \\
y \quad \overline{y_1 \ldots y_n} \quad \text{size } n \\
\end{array}
\]

\( \text{Standard } \text{"tape merging" takes } O(m+n) \text{ comparisons} \)

\( \text{Size } m+n. \)

What if \( m \gg n \)?

• Assume \( X \) stored as BST \( T \)

\[
\text{merge } (x_i, y_1): \\
\text{ for } i = 1 \ldots m \\
\text{ insert } y_1 \text{ into } T
\]

• Cost of insert
\( O \left( \log (m+n) \right) = O \left( \log m \right) \)

• Total cost of merge
\( O \left( m \cdot \log m \right) \)

\[
\begin{aligned}
Y_1 < Y_2 < Y_3 < \ldots < Y_n \\
Y_1 \quad \overline{y_1 \ldots y_n} \quad \text{size } n
\end{aligned}
\]

\( \text{(Again, assume } X \text{ is given as } \text{finger search tree. If not, we can build it from a sorted list with no comparisons.)} \)

Improvement

Idea: insert \( y_i \) using linear search

\( \text{Cost of insert } y_i \) is
\( \log(d(Y_{i-1}, Y_i)) \)

\( \# \text{ of elements of } X \text{ between } Y_{i-1} \text{ and } Y_i \)
Let \( d_i = \lfloor X \cap [y_{i-1}, y_i] \rfloor \). 
\( d_i \) is the number of elements of \( X \) in \( [y_{i-1}, y_i] \).

\[
\sum_{i=1}^{m} d_i \leq m \\
d_{j} = \# \text{ elements of } X \text{ in } (-\infty, y_j]
\]

Total cost of merging:

\[
\sum_{i=1}^{m} \log(d_i) \leq m \log \left( \frac{\sum d_i}{m} \right) \leq m \cdot \log \frac{m}{n}
\]

* \( \log \) function is concave

\[
\frac{1}{m} \sum_{i=1}^{m} \log(d_i) \leq \log \left( \frac{\sum d_i}{m} \right) \quad \text{( Jensen's inequality)}
\]

Illustration

Using binary search we improve \( O(m \cdot \log m) \) to \( O\left( m \log \frac{m}{n} \right) \).

\( m > n \)

**Theorem:** Merging two sorted lists of sizes \( m \) and \( n \) requires \( \Omega \left( m \log \frac{m}{n} \right) \) comparisons.

![Illustration of merging](attachment:image)

\( x \)

\( \{0, 1, 2, 4, 5, 7, 9, 10, 11, 12\} \)

\( y \)

\( \{0, 1, 2, 4, 5, 7, 9, 10, 11, 12\} \)

\( m+n \) positions in result (we just need to fix positions that come from \( y \))

How many possible 'merge orders' are there: \( \binom{m+n}{n} \)

Possible merge orders

\( \binom{m+n}{n} \) leaves, depth of decision tree

\( \geq \log_2 \left( \binom{m+n}{n} \right) \)

Any merging algorithm can be seen as a decision tree; hence one comparison eliminates one possible merge order.

For some input, merging will take \( \log_2 \left( \frac{m+n}{n} \right) \) comparisons.
\[
\log_2 \left( \frac{m+n}{n} \right) \geq \log_2 \left( \left( \frac{m+n}{n} \right)^n \right) = \Omega \left( n \cdot \log \frac{m}{n} \right)
\]

\[
\begin{align*}
(a) &= \frac{a!}{b! \cdot (a-b)!} = \frac{(a-b+1) \cdot (a-b+2) \cdots a}{1 \cdot 2 \cdots b} = \frac{a-b+1}{b-1} \cdot \frac{\binom{a}{b}}{b!} \\
\end{align*}
\]

3) Collection of splittable lists.

- Maintain a set of sorted lists.
- Initially a single list of size \(n\).
- Option: split \((L, x)\)

- Continue until all lists are singleton (of size 1).
- Assume all splits are proper.

\[
L = [4, 1, 2, 5, 7] \rightarrow L_1 = [4, 1, 2] \rightarrow L_2 = [5, 7] \rightarrow L_3 = [5, 7, 8, 9, 10]
\]

- Output: total time \(T(n) = \log n + T(n-1)\) which is \(O(m \log n)\) where \(m\) is the total time overall.

\[
T(n) = \log n + T(n-1)
\]
look more carefully at a single split
Split $(L, x)$

\[
\text{cont: } O(\log \min \{k, m-k\}) \quad \text{(instead of } O(\log n))
\]

\[
\text{O} \left( \log \frac{k}{m} \right)
\]

\[
\text{O} \left( \log (m-k) \right)
\]

Best of both by searching simultaneously from both ends

\[
\text{cont: } O \left( \log \min \{k, m-k\} \right)
\]

\[\text{Analysis} \]

\[\text{Amortized analysis.}\]

Define potential of a list of given

\[m - \log_2 n\]

Total potential \( \phi \) is sum of list potentials:

\[\phi = m - \log_2 n\]

\[\Delta \phi = m - \log_2 n\]

\[\Delta \phi > 0\]

\[\Delta \phi > 0\]

\[\Delta \phi = \text{actual cont} + \Delta \phi\]

\[= \log \min \{k, m-k\} + k - \log k + (m-k) - \log (m-k) - \frac{m - \log m}{n}
\]

\[= \log \min \{k, m-k\} + \log m - \log k - \log (m-k)
\]

\[= x\]
\[
- \log_k (\log (m-k)) = - \log (\min \{k, m-k\}) - \log (\max \{k, m-k\}) \\
= \log m - \log (\max \{k, m-k\}) \leq \\
\leq \log m - \log \frac{m}{2} \leq 1
\]

amortized cost of split \( \leq 1 \)

total cost of \( n-1 \) splits: \( O(n) \).

---

Implementing finger search trees

**Idea:**
- balanced search tree
- augmented with level-links
  
  (AVL/red-black
  not most suitable)

**Alternative search tree:** \((a,b)\)-tree

Recap: What is an \((a,b)\)-tree:

- keys stored only in leaves
- every leaf has same depth
- keys sorted left-to-right
- internal nodes are used only for minimax
- internal nodes have degree \(za\), \(z \leq b\)

\[
\text{Insert/Deletion/Searching:}\quad O(\log z)\text{ time}
\]

"distance" between shortest node \(x\) and target \(y\)

\# items between \(x,y\)

split/merge

used in databases, file systems, etc.
**Search in (2,4)-tree takes \( O(\log n) \) time**

- Degree \( \geq 2 \) \( \Rightarrow \) depth \( \leq \log_2 n + 1 \)

**Insert:**
- Find where new key should go
- Attach as extra key, update parent keys
- If parent degree \( = 5 \) split by splitting

- If parent node had degree 2 or 3 before, O.K.
- If parent already had degree 4, need to split

Split nodes of degree 5, could propagate up towards root

- If root itself splits, free height incr. by 1

**Cost of insert:**
- Cost of search: \( O(\log n) \)
- Cost of insert: \( O(1) + \# \text{ splits} \)

**Exercise:**
- A upper bound \# of splits is \( O(1) \)
Delete
- Find leaf to be deleted
- Remove leaf
- Fix parent if needed

- If degree of parent was 3,4 then OK.
- If degree of parent was 2:
  - Look at neighbor of parent
    - If any of them has degree $\geq 3$,
      - Borrow a child
    - If neighbor has degree = 2, merge
      (update keys accordingly)

Borrow:

Merge:

Cost of delete:
- $O(\log n)$ for search
- For actual delete $O(1) + \frac{\# \text{merges}}{\log n} \in O(\log n)$

# merges is amortized $O(1)$

Merge:
$\text{dep} 2 + \text{dep} 2 \rightarrow \text{dep} 3$

Summary:
- Search: $O(\log n)$ time
- Insert: $O(1)$ amortized time if we have pointer to location
  and need not search
Finger search trees

Extended (2,4)-trees with level-hulls (between nodes at same level)

Finger search for y, starting from x

Search should cost $O(\log d)$

Suppose $x < y$ (other case symmetric)

Go up from x until one of the following happens:

1. We reach right path of tree
2. Right leaf-neighbor has a key $> y$
3. We reach left path of tree, left neighbor (rightmost) has a key $\leq y$

In all cases, search down for y in one or two subtrees

Case when y could be in one of two subtrees
Analysis of search

Look at step before last, when (1)-(6) did not hold.

We can find a subtree with keys in [x,y]

Subtree has \( \leq d \) leaves
Subtree has \( \geq 2^{h-2} \) leaves

\[ 2^{h-2} \leq d \quad \text{or} \quad 2^{h-2} \leq \log d \]

\[ h-2 \leq \log d \Rightarrow \text{cost} = O(\log d) \]

Finer analysis:
We can also find a subtree with keys outside of [x,y]

Case 1. h stop
Case 2.

In both cases we have a subtree with root at level \( h-1 \), with keys outside [x,y]

Subtree has \( \geq 2^{h-2} \) leaves
Subtree has \( \leq m-d \) leaves

\[ 2^{h-2} \leq m-d \Rightarrow h = O(\log (m-d)) \]

Cost of search in finger search tree:

\[ n = O(\log \min\{d, m-d\}) \]

Insert/Delete

must maintain level-labels under merge/split (easy)
Split/Join in a Finger Search Tree

Split \((T, x)\)

- Go up from \(x\) until reaching leftmost/riestmost
- Split all internal nodes along path
  
  \[ \text{Cost: } O(\log n) \]

Join

- Assume keys in \(T_1\) are \(< \) keys in \(T_2\)
- Combine \(T_1, T_2\) into a single tree

\[ \text{Cost: } O(\log n) \]
Fixed split (2-pivot split)

Split \((T, x, y)\)

Cost: \(O(\log \min \{d, n-d\})\)

Idea: walk up at the same time from both \(x\) and \(y\), until we reach the same internal node or two neighboring nodes

**Case 1:**

**Case 2**

**Case 3**

\(e\) or \(f\) may be on leaves.
4. Detach subtree rooted at e (call it \( T_L \))

- Fix remaining \( T \) (\( \ldots \) )

2. Split \( T_L \) at \( x \)

\[ \Rightarrow T_L^0 \ (\leq x) \]
\[ \Rightarrow T_L^1 \ (\geq x) \]

3. Join \( T_L^1 \) and remaining \( T \) \( \Rightarrow T' \)

4. Split \( T' \) at \( y \)

\[ \Rightarrow T_R^1 \ (\leq y) \]
\[ \Rightarrow T_R^2 \ (\geq y) \]

5. Return \( T_R^1 \) (within \( E[X,Y] \))

- Join (\( T_L^0, T_R^2 \)) (outside \( E[X,Y] \))

Cost \( O(h) = O(\log \text{ size of input}) \)

Analysis similar to search.

(other cases omitted)