Text

<table>
<thead>
<tr>
<th>Implementation</th>
<th>worst-case cost (after N inserts)</th>
<th>average case (after N random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>delete</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
</tr>
<tr>
<td>goal</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

- **Challenge.** Guarantee performance.
Allow 1 or 2 keys per node.
- 2-node: one key, two children.
- 3-node: two keys, three children.

**Perfect balance.** Every path from root to null link has same length.

**Symmetric order.** Inorder traversal yields keys in ascending order.

**Search.**
- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).
2-3 tree demo

Search.
- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

search for H

M
H is between E and J (go middle)
H
E J
L
P
S X

search for H

M
E J
H
L
P
S X

found H (search hit)

search for B

B is less than M (go left)
B
M
E J
R
A C
H
L
P
S X

search for B

B is less than E (go left)
B
E J
R
A C
H
L
P
S X
**2-3 tree demo**

**Search.**
- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

**search for B**

B is between A and C (go middle).

**2-3 tree demo**

**Insert into a 2-node at bottom.**
- Search for key, as usual.
- Replace 2-node with 3-node.

**Insert K**

K is less than M (go left).

**2-3 tree demo**

**Insert into a 2-node at bottom.**
- Search for key, as usual.
- Replace 2-node with 3-node.

**Insert K**

K is greater than J (go right).
2-3 tree demo

Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.

2-3 tree demo

Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.

2-3 tree demo

Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.

2-3 tree demo

Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

2-3 tree demo

Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
### 2-3 tree demo

- Insert into a 3-node at bottom.
  - Add new key to 3-node to create temporary 4-node.
  - Move middle key in 4-node into parent.

```
2-3 tree demo

insert Z
```

```
2-3 tree demo

insert Z
```

```
2-3 tree demo

insert Z
```

```
2-3 tree demo

insert Z
```
2-3 tree demo

Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

Insert Z

split 4-node into two 2-nodes
(pass middle key to parent)

2-3 tree demo

Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

Insert Z

2-3 tree demo

Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

Insert Z

2-3 tree demo

Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

Insert L

convert 3-node into 4-node
2-3 tree demo

Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it’s a 4-node, split it into three 2-nodes.

insert L
2-3 tree demo

Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

Search in a 2-3 tree

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

Insertion in a 2-3 tree

Case 1. Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.
**Insertion in a 2-3 tree**

**Case 2.** Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.

**Local transformations in a 2-3 tree**

Splitting a 4-node is a local transformation: constant number of operations.

**Global properties in a 2-3 tree**

**Invariants.** Maintains symmetric order and perfect balance.

**Pf.** Each transformation maintains symmetric order and perfect balance.
2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

Tree height.
• Worst case:
• Best case:

Tree height.
• Worst case: \[ \lg N \] [all 2-nodes]
• Best case: \[ \log_3 N = .631 \lg N \] [all 3-nodes]
• Between 12 and 20 for a million nodes.
• Between 18 and 30 for a billion nodes.

Guaranteed logarithmic performance for search and insert.

ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>worst-case cost (after N inserts)</th>
<th>average case (after N random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td>( N/2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>( N )</td>
<td>( 1.39 \lg N )</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( 1.39 \lg N )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-3 tree</td>
<td>( c \lg N )</td>
<td>( c \lg N )</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( c \lg N )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( c \lg N )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( c \lg N )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bottom line. Could do it, but there’s a better way.
**Balanced Search Trees**

- 2-3 search trees
- Red-black BSTs
- B-trees
- Geometric applications of BSTs

---

**Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)**

1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3–nodes.

---

**An equivalent definition**

A BST such that:
- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

---

**Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees**

Key property. 1–1 correspondence between 2–3 and LLRB.
Search implementation for red-black BSTs

**Observation.** Search is the same as for elementary BST (ignore color).

```java
public Val get(Key key) {
    Node x = root;
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```

**Remark.** Most other ops (e.g., ceiling, selection, iteration) are also identical.

Red-black BST representation

Each node is pointed to by precisely one link (from its parent) ⇒ can encode color of links in nodes.

```java
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node {
    Key key;
    Value val;
    Node left, right;
    boolean color; // color of parent link
    Node(Key key, Value val) {
        this.key   = key;
        this.val   = val;
        this.N     = 1;
        this.color = RED;
    }
}
```

Elementary red-black BST operations

**Left rotation.** Orient a (temporarily) right-leaning red link to lean left.

```java
private Node rotateLeft(Node h) {
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

**Invariants.** Maintains symmetric order and perfect black balance.
Elementary red-black BST operations

**Right rotation.** Orient a left-leaning red link to (temporarily) lean right.

```java
private Node rotateRight(Node h) {
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

**Invariants.** Maintains symmetric order and perfect black balance.

---

Elementary red-black BST operations

**Color flip.** Recolor to split a (temporary) 4-node.

```java
private void flipColors(Node h) {
    assert !isRed(h);
    assert isRed(h.left);
    asset isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

**Invariants.** Maintains symmetric order and perfect black balance.
**Insertion in a LLRB tree: overview**

**Basic strategy.** Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black BST operations.

**Insertion in a LLRB tree**

**Case 1.** Insert into a 2-node at the bottom.
- Do standard BST insert; color new link red.
- If new red link is a right link, rotate left.

**Warmup 1.** Insert into a tree with exactly 1 node.

**Warmup 2.** Insert into a tree with exactly 2 nodes.
**Insertion in a LLRB tree**

**Case 2.** Insert into a 3-node at the bottom.
- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).

**Red-black BST insertion**

- Insertion in a LLRB tree: passing red links up the tree
  - Do standard BST insert; color new link red.
  - Rotate to balance the 4-node (if needed).
  - Flip colors to pass red link up one level.
  - Rotate to make lean left (if needed).
  - Repeat case 1 or case 2 up the tree (if needed).
Red-black BST insertion

Insert A

Red-black BST insertion

two left reds in a row
(rotate S right)

Red-black BST insertion

both children red
(flip colors)

Red-black BST insertion

both children red
(flip colors)
Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion
Red-black BST insertion

ded-black BST

![Diagram of a red-black BST with nodes A, E, S, and R.]

Red-black BST insertion

insert C

![Diagram showing the insertion of node C into the red-black BST.]

Red-black BST insertion

right link red
(rotate A left)

![Diagram showing the rotation of node A to the left.]
Red-black BST insertion

red-black BST

E
C
S
A
R

Red-black BST insertion

red-black BST

E
C
S
A
R

Red-black BST insertion

insert H

E
C
S
A
R
H

Red-black BST insertion

two left reds in a row
(rotate S right)

E
C
S
A
R
H
Red-black BST insertion

E
C
A

both children red
(flip colors)

Red-black BST insertion

E
C
A

both children red
(flip colors)

Red-black BST insertion

H
S
R
C
A

right link red
(rotate E left)

Red-black BST insertion

H
S
E
C
A

red-black BST
Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion
Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

red-black BST

\[ \text{insert } M \]
Red-black BST insertion

insert M

\begin{center}
\begin{tikzpicture}
  \node (E) at (0,0) {E};
  \node (R) at (1,1) {R};
  \node (X) at (2,0) {X};
  \node (C) at (-1,1) {C};
  \node (H) at (-1,0) {H};
  \node (M) at (0,-1) {M};
  \node (S) at (1,-1) {S};
  \node (A) at (-2,0) {A};

  \draw (E) -- (R);
  \draw (E) -- (C);
  \draw (E) -- (H);
  \draw (E) -- (M);
  \draw (E) -- (S);
  \draw (C) -- (A);
  \draw (H) -- (M);
  \draw (M) -- (S);

  \node [red] at (1.5,1) {right link red (rotate H left)};
\end{tikzpicture}
\end{center}

Red-black BST insertion

red-black BST

\begin{center}
\begin{tikzpicture}
  \node (E) at (0,0) {E};
  \node (R) at (1,1) {R};
  \node (X) at (2,0) {X};
  \node (C) at (-1,1) {C};
  \node (M) at (0,-1) {M};
  \node (S) at (1,-1) {S};
  \node (A) at (-2,0) {A};

  \draw (E) -- (R);
  \draw (E) -- (C);
  \draw (E) -- (M);
  \draw (E) -- (S);
  \draw (C) -- (A);
  \draw (M) -- (S);

\end{tikzpicture}
\end{center}

Red-black BST insertion

insert P

\begin{center}
\begin{tikzpicture}
  \node (E) at (0,0) {E};
  \node (R) at (1,1) {R};
  \node (X) at (2,0) {X};
  \node (C) at (-1,1) {C};
  \node (M) at (0,-1) {M};
  \node (S) at (1,-1) {S};
  \node (A) at (-2,0) {A};
  \node (H) at (-1,0) {H};
  \node (P) at (0,-2) {P};

  \draw (E) -- (R);
  \draw (E) -- (C);
  \draw (E) -- (M);
  \draw (E) -- (S);
  \draw (C) -- (A);
  \draw (M) -- (S);
  \draw (C) -- (H);
  \draw (H) -- (P);

\end{tikzpicture}
\end{center}

Red-black BST insertion

\begin{center}
\begin{tikzpicture}
  \node (E) at (0,0) {E};
  \node (R) at (1,1) {R};
  \node (X) at (2,0) {X};
  \node (C) at (-1,1) {C};
  \node (M) at (0,-1) {M};
  \node (S) at (1,-1) {S};
  \node (A) at (-2,0) {A};
  \node (H) at (-1,0) {H};
  \node (P) at (0,-2) {P};

  \draw (E) -- (R);
  \draw (E) -- (C);
  \draw (E) -- (M);
  \draw (E) -- (S);
  \draw (C) -- (A);
  \draw (M) -- (S);
  \draw (C) -- (H);
  \draw (H) -- (P);

  \node [red] at (1.5,1) {two red children (flip colors)};
\end{tikzpicture}
\end{center}
Red-black BST insertion

Insert P

Red-black BST insertion

Right link red (rotate E left)

two red children (flip colors)

Red-black BST insertion

two red children (flip colors)

two left reds in a row (rotate R right)

Red-black BST insertion

two red children (flip colors)
Red-black BST insertion

![Diagram 1]

- Two red children (flip colors)

![Diagram 2]

- Red-black BST

![Diagram 3]

- Red-black BST

![Diagram 4]

- Red-black BST
Red-black BST insertion

insert L

Red-black BST insertion

right link red
(rotate H left)

LLRB tree insertion trace

Standard indexing client.

red-black BST
corresponding 2-3 tree
Standard indexing client (continued).

### LLRB tree insertion trace

#### Red-black BST

- **Left child red, left child black:** rotate left.
- **Left child, left-left grandchild red:** rotate right.
- **Both children red:** flip colors.

#### Corresponding 2-3 tree

#### Insertion in a LLRB tree: Java implementation

#### Code for both cases.

- Right child red, left child black: rotate left.
- Left child, left-left grandchild red: rotate right.
- Both children red: flip colors.

```java
private Node put(Node h, Key key, Value val) {
    if (h == null) return new Node(key, val, RED);
    int cmp = key.compareTo(h.key);
    if      (cmp  < 0) h.left  = put(h.left,  key, val);
    else if (cmp  > 0) h.right = put(h.right, key, val);
    else
        if (cmp == 0) h.val = val;
        if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
        if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
        if (isRed(h.left)  && isRed(h.right)) flipColors(h);
    return h;
}
```

#### Insertion in a LLRB tree: visualization

#### 255 insertions in ascending order

#### Insertion in a LLRB tree: visualization

#### 255 insertions in descending order

#### Remark.

Only a few extra lines of code to standard BST insert.
**Insertion in a LLRB tree: visualization**

**Remark.** Only a few extra lines of code to standard BST insert.

N = 255  
max = 10  
avg = 7.3  
opt = 7.0

255 random insertions

---

**Balance in LLRB trees**

**Proposition.** Height of tree is \( \leq 2 \lg N \) in the worst case.

**Pf.**
- Every path from root to null link has same number of black links.
- Never two red links in a row.

**Property.** Height of tree is \( \sim 1.00 \lg N \) in typical applications.

---

**ST implementations: summary**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>worst-case cost (after N insert)</th>
<th>average cost (after N random inserts)</th>
<th>ordered revision?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>( \lg N )</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.39 ( \lg N )</td>
</tr>
<tr>
<td>2-3 tree</td>
<td>c ( \lg N )</td>
<td>c ( \lg N )</td>
<td>c ( \lg N )</td>
<td>c ( \lg N )</td>
</tr>
<tr>
<td>red-black BST</td>
<td>2 ( \lg N )</td>
<td>2 ( \lg N )</td>
<td>2 ( \lg N )</td>
<td>1.00 ( \lg N )</td>
</tr>
</tbody>
</table>

^ exact value of coefficient unknown but extremely close to 1

---

**War story: why red-black?**

**Xerox PARC innovations.** [1970s]
- Alto.
- GUI.
- Ethernet.
- Smalltalk.
- InterPress.
- Laser printing.
- Bitmapped display.
- WYSIWYG text editor.
- ...

---

**Proposition.** Height of tree is \( \leq 2 \lg N \) in the worst case.

**Pf.**
- Every path from root to null link has same number of black links.
- Never two red links in a row.

**Property.** Height of tree is \( \sim 1.00 \lg N \) in typical applications.
Balanced Search Trees

- 2-3 search trees
- Red-black BSTs
- B-trees
- Geometric applications of BSTs

File system model

Page. Contiguous block of data (e.g., a file or 4,096-byte chunk).
Probe. First access to a page (e.g., from disk to memory).

Property. Time required for a probe is much larger than time to access data within a page.

Cost model. Number of probes.

Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to $M-1$ key-link pairs per node.
- At least 2 key-link pairs at root.
- At least $M/2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

Choose $M$ as large as possible so that $M$ links fit in a page, e.g., $M = 1024$.

Anatomy of a B-tree set ($M = 6$)

Searching in a B-tree

- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.

Searching in a B-tree set ($M = 6$)
**Insertion in a B-tree**

- Search for new key.
- Insert at bottom.
- Split nodes with $M$ key-link pairs on the way up the tree.

**Balance in B-tree**

**Proposition.** A search or an insertion in a B-tree of order $M$ with $N$ keys requires between $\log_{M/2} N$ and $\log_{M} N$ probes.

**Pf.** All internal nodes (besides root) have between $M/2$ and $M - 1$ links.

**In practice.** Number of probes is at most 4.

**Optimization.** Always keep root page in memory.

---

**Building a large B tree**

Each line shows the result of inserting one key into a page.

- Red: occupied portion of page
- Black: unoccupied portion of page
- Full page: about to split

---

**Balanced trees in the wild**

Red-black trees are widely used as system symbol tables.

- Java: java.util.TreeMap, java.util.TreeSet.
- C++ STL: map, multimap, multiset.
- Linux kernel: completely fair scheduler, linux/rbtree.h.

**B-tree variants.** B+ tree, B*tree, B# tree, ...

**B-trees (and variants) are widely used for file systems and databases.**

- Windows: HFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.
2-d orthogonal range search

Extension of ordered symbol-table to 2d keys.
- Insert a 2d key.
- Delete a 2d key.
- Search for a 2d key.
- Range search: find all keys that lie in a 2d range.
- Range count: number of keys that lie in a 2d range.

Geometric interpretation.
- Keys are point in the plane.
- Find/count points in a given $h \times v$ rectangle.

Applications. Networking, circuit design, databases,...
2d orthogonal range search: grid implementation costs

Space-time tradeoff.
- Space: $M^2 + N$.
- Time: $1 + N/M^2$ per square examined, on average.

Choose grid square size to tune performance.
- Too small: wastes space.
- Too large: too many points per square.
- Rule of thumb: $\sqrt{N}$-by-$\sqrt{N}$ grid.

Running time. [if points are evenly distributed]
- Initialize data structure: $N$.
- Insert point: 1.
- Range search: 1 per point in range.

Clustering

Grid implementation. Fast and simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.
- Lists are too long, even though average length is short.
- Need data structure that gracefully adapts to data.

Space-partitioning trees

Use a tree to represent a recursive subdivision of 2d space.

Grid. Divide space uniformly into squares.
2d tree. Recursively divide space into two halfplanes.
Quadtree. Recursively divide space into four quadrants.
BSP tree. Recursively divide space into two regions.
Space-partitioning trees: applications

Applications.
- Ray tracing.
- 2d range search.
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- Nearest neighbor search.
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.

Kd tree

Kd tree. Recursively partition $k$-dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2d trees.

Efficient, simple data structure for processing $k$-dimensional data.
- Widely used.
- Adapts well to high-dimensional and clustered data.
- Discovered by an undergrad in an algorithms class!

N-body simulation

Goal. Simulate the motion of $N$ particles, mutually affected by gravity.

Brute force. For each pair of particles, compute force. $F = \frac{G m_1 m_2}{r^2}$

Appel algorithm for N-body simulation

Key idea. Suppose particle is far, far away from cluster of particles.
- Treat cluster of particles as a single aggregate particle.
- Compute force between particle and center of mass of aggregate particle.
Appel algorithm for N-body simulation

- Build 3d-tree with $N$ particles as nodes.
- Store center-of-mass of subtree in each node.
- To compute total force acting on a particle, traverse tree, but stop as soon as
distance from particle to subdivision is sufficiently large.

Impact. Running time per step is $N \log N$ instead of $N^2 \Rightarrow$ enables new research.