Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
Balanced Search Trees

- 2-3 search trees
- Red-black BSTs
- B-trees
- Geometric applications of BSTs
<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></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</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>lg 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>goal</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

- **Challenge.** Guarantee performance.
Balanced Search Trees

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

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.
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.
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.
2-3 tree demo

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

search for H

H is less than M
(go left)
2-3 tree demo

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

search for H

H is between E and J
(go middle)
Search.

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

search for H

found H (search hit)
Search.

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

**search for B**

B is less than M (go left)
2-3 tree demo

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

search for B

M
  / | \\
E J B
  | |
AC H L
  |
P SX
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)
Search.

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

**search for B**

.. image:: 2-3_tree_demo.png
   :width: 100%
   :alt: 2-3 tree demo

- B link is null (search miss)
2-3 tree demo

Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

```
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.

\[\text{insert } K\]

K is greater than J
(go right)
Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

insert K
Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

**insert K**
Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

**2-3 tree demo**
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 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 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

search ends here
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

replace 3-node with temporary 4-node containing Z
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 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.
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
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.

class 2-3 tree demo

insert L

class convert 3-node into 4-node
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.
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.

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.

**insert L**

![Diagram of 2-3 tree demo](image-url)
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.

height of tree increases by 1

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.

insert L
Search in a 2-3 tree

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

successful search for H

unsuccessful search for B

H is less than M so look to the left

H is between E and L so look in the middle

found H so return value (search hit)

B is less than M so look to the left

B is less than E so look to the left

B is between A and C so look in the middle
link is null so B is not in the tree (search miss)
Case 1. Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.
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.

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.
- If you reach the root and it's a 4-node, split it into three 2-nodes.
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.

- **root**
  - a b c
- **parent is a 2-node**
  - left
    - a b c
     - d
  - right
    - a
     - b c d
- **parent is a 3-node**
  - left
    - a b c
     - d e
  - middle
    - a e
     - b c d
e
  - right
    - a b
     - c d e
Perfect balance. Every path from root to null link has same length.

Tree height.
- Worst case:
- Best case:
2-3 tree: performance

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

Tree height.

• Worst case: $\lg N$. [all 2-nodes]
• Best case: $\log_3 N \approx 0.631 \lg N$. [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></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</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>lg 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>
</tbody>
</table>

Constants depend upon implementation.
**2-3 tree: implementation?**

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

**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.

"perfect black balance"
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).

![Search implementation diagram](image)

```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
}

private boolean isRed(Node x)
{
    if (x == null) return false;
    return x.color == RED;
}
```

null links are black
Elementary red-black BST operations

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

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

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

rotate E left (after)

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.

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.

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

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

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

private void flipColors(Node h) {
    assert !isRed(h);
    assert isRed(h.left);
    assert 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.
Warmup 1. Insert into a tree with exactly 1 node.

**Insertion in a LLRB tree**

**Left**
- Search ends at this null link
- Red link to new node containing a converts 2-node to 3-node
- Red link to new node containing a
- Root

**Right**
- Search ends at this null link
- Attached new node with red link
- Rotated left to make a legal 3-node
- Root
**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.

![Diagram showing insertion process]

- **Add new node here.**
- **Right link red, so rotate left.**
Warmup 2. Insert into a tree with exactly 2 nodes.

**Insertion in a LLRB tree**

- **Larger**
  - search ends at this null link
  - attached new node with red link
  - colors flipped to black

- **Smaller**
  - search ends at this null link
  - attached new node with red link
  - rotated right
  - colors flipped to black

- **Between**
  - search ends at this null link
  - attached new node with red link
  - rotated left
  - rotated right
  - colors flipped to black
**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).

---

**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).
- Repeat case 1 or case 2 up the tree (if needed).
insert S
Red-black BST insertion

insert E
Red-black BST insertion

insert A
Red-black BST insertion

insert A

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

![Red-black BST](image-url)
Red-black BST insertion

red–black BST
Red-black BST insertion

insert R
red–black BST
Red-black BST insertion

red–black BST
Red-black BST insertion

insert C
Red-black BST insertion

right link red
(rotate A left)
Red-black BST insertion

red–black BST
Red-black BST insertion

red-black BST

Diagram of a red-black BST with nodes E, C, S, A, and R.
Red-black BST insertion

red–black BST
Red-black BST insertion

insert H
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

right link red
(rotate E left)
Red-black BST insertion

red–black BST

A

C

E

H

R

S
Red-black BST insertion

red–black BST
Red-black BST insertion

red-black BST

![Red-black BST insertion diagram](image-url)
Red-black BST insertion

insert X
Red-black BST insertion

insert X

right link red (rotate S left)
Red-black BST insertion

red-black BST
Red-black BST insertion

red–black BST

![Red-black BST diagram]
Red-black BST insertion

red–black BST
Red-black BST insertion

insert M
Red-black BST insertion

insert M

right link red (rotate H left)
Red-black BST insertion

red–black BST
Red-black BST insertion

insert P
Red-black BST insertion

insert P

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

insert P

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

right link red
(rotate E left)
Red-black BST insertion

two left reds in a row
(rotate R right)
Red-black BST insertion

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

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

red–black BST
Red-black BST insertion

red–black BST

```
  M
 / \
 E  R
 /  /\  
C  H  P  X
 / \   /   /
A   S
```
Red-black BST insertion

red-black BST

![Red-black BST diagram]
Red-black BST insertion

insert L
Red-black BST insertion

insert L

right link red
(rotate H left)
Red-black BST insertion

red-black BST

\[\text{M} \quad \text{E} \quad \text{R} \quad \text{P} \quad \text{X} \quad \text{C} \quad \text{L} \quad \text{A} \quad \text{H} \quad \text{S}\]
LLRB tree insertion trace

Standard indexing client.

- **insert S**
  - **red-black BST**
  - **corresponding 2–3 tree**
Standard indexing client (continued).

LLRB tree insertion trace

red–black BST

corresponding 2–3 tree
Insertion in a LLRB tree: Java implementation

Same 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;
}
```

only a few extra lines of code
provides near-perfect balance
Insertion in a LLRB tree: visualization

N = 255
max = 8
avg = 7.0
opt = 7.0

255 insertions in ascending order
Insertion in a LLRB tree: visualization

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

N = 255
max = 8
avg = 7.0
opt = 7.0

255 insertions in descending order
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 inserts)</th>
<th>average case (after N random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</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>lg 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.
• ...

A Dichromatic Framework For Balanced Trees

Leo J. Guibas
Xerox Palo Alto Research Center,
Palo Alto, California, and
Carnegie-Mellon University

Robert Sedgewick*
Program in Computer Science
Brown University
Providence, R. I.

Abstract

In this paper we present a uniform framework for the implementation and study of balanced tree algorithms. We show how to imbed in this

the way down towards a leaf. As we will see, this has a number of significant advantages over the older methods. We shall examine a

number of variations on a common theme and exhibit full implementations which are notable for their brevity. One

implementation is examined carefully, and some properties about its
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).
Probes. 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 $\log_{M-1} N$ and $\log_{M/2} N$ probes.

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

**In practice.** Number of probes is at most 4. \[ M = 1024; N = 62 \text{ billion} \]

**Optimization.** Always keep root page in memory.
Building a large B tree

Each line shows the result of inserting one key in some page.
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: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.
Balanced Search Trees

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

- 1D range search
- Line segment intersection
- kd trees
- Rectangle intersection
This lecture. Intersections among geometric objects.

Applications. CAD, games, movies, virtual reality, VLSI design, databases, ...

Efficient solutions. Binary search trees (and extensions).
Geometric applications of BSTs

- 1D range search
- Line segment intersection
- kd trees
Id range search

Extension of ordered symbol table.
• Insert key-value pair.
• Search for key $k$.
• Delete key $k$.
• Range search: find all keys between $k_1$ and $k_2$.
• Range count: number of keys between $k_1$ and $k_2$.

Application. Database queries.

Geometric interpretation.
• Keys are point on a line.
• Find/count points in a given 1d interval.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert B</td>
<td>B</td>
</tr>
<tr>
<td>insert D</td>
<td>B D</td>
</tr>
<tr>
<td>insert A</td>
<td>A B D</td>
</tr>
<tr>
<td>insert I</td>
<td>A B D I</td>
</tr>
<tr>
<td>insert H</td>
<td>A B D H I</td>
</tr>
<tr>
<td>insert F</td>
<td>A B D F H I</td>
</tr>
<tr>
<td>insert P</td>
<td>A B D F H I P</td>
</tr>
<tr>
<td>count G to K</td>
<td>2</td>
</tr>
<tr>
<td>search G to K</td>
<td>H I</td>
</tr>
</tbody>
</table>
Unordered array. Fast insert, slow range search.

Ordered array. Slow insert, binary search for $k_1$ and $k_2$ to do range search.

### Order of Growth of Running Time for 1d Range Search

<table>
<thead>
<tr>
<th>data structure</th>
<th>insert</th>
<th>range count</th>
<th>range search</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>$1$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>ordered array</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$R + \log N$</td>
</tr>
<tr>
<td>goal</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$R + \log N$</td>
</tr>
</tbody>
</table>

### Parameters.

- $N = \text{number of keys}$.
- $R = \text{number of keys that match}$.
Id range count: BST implementation

Id range count. How many keys between \( l_0 \) and \( h_i \) ?

Proposition. Running time is proportional to \( \log N \) (assuming BST is balanced).

Pf. Nodes examined = search path to \( l_0 \) + search path to \( h_i \).
Id range search. Find all keys between $lo$ and $hi$.

- Recursively find all keys in left subtree (if any could fall in range).
- Check key in current node.
- Recursively find all keys in right subtree (if any could fall in range).

Proposition. Running time is proportional to $R + \log N$ (assuming BST is balanced).

Pf. Nodes examined = search path to $lo$ + search path to $hi$ + matching keys.
Geometric applications of BSTs

- 1D range search
- Line segment intersection
- kd trees
Orthogonal line segment intersection search

Given $N$ horizontal and vertical line segments, find all intersections.

Quadratic algorithm. Check all pairs of line segments for intersection.

Nondegeneracy assumption. All $x$- and $y$-coordinates are distinct.
Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- $x$-coordinates define events.
- $h$-segment (left endpoint): insert $y$-coordinate into BST.
Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- $x$-coordinates define events.
- $h$-segment (left endpoint): insert $y$-coordinate into BST.
- $h$-segment (right endpoint): remove $y$-coordinate from BST.
Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- $x$-coordinates define events.
- $h$-segment (left endpoint): insert $y$-coordinate into BST.
- $h$-segment (right endpoint): remove $y$-coordinate from BST.
- $v$-segment: range search for interval of $y$-endpoints.
Orthogonal line segment intersection search: sweep-line algorithm analysis

Proposition. The sweep-line algorithm takes time proportional to $N \log N + R$ to find all $R$ intersections among $N$ orthogonal line segments.

Pf.

- Put $x$-coordinates on a PQ (or sort). \(N \log N\)
- Insert $y$-coordinates into BST. \(N \log N\)
- Delete $y$-coordinates from BST. \(N \log N\)
- Range searches in BST. \(N \log N + R\)

Bottom line. Sweep line reduces 2d orthogonal line segment intersection search to 1d range search.
Geometric applications of BSTs

- 1D range search
- Line segment intersection
- kd trees
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 \( h-v \) rectangle.

rectangle is axis-aligned

Applications. Networking, circuit design, databases,...
Grid implementation.

- Divide space into $M$-by-$M$ grid of squares.
- Create list of points contained in each square.
- Use 2d array to directly index relevant square.
- Insert: add $(x, y)$ to list for corresponding square.
- Range search: examine only those squares that intersect 2d range query.
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.
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.
Grid implementation. Fast and simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.
Ex. USA map data.

13,000 points, 1000 grid squares
half the squares are empty
half the points are in 10% of the squares
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.
Recursively partition plane into two halfplanes.

Insertion in a 2d tree
Data structure. BST, but alternate using $x$- and $y$-coordinates as key.

- Search gives rectangle containing point.
- Insert further subdivides the plane.
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

find all points in 2d tree that are contained in green query rectangle
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

search root node
check if query rectangle contains point 1
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

search left subtree
check if query rectangle contains point 3
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

query rectangle intersects splitting line
search bottom and top subtrees
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

search left subtree
check if query rectangle contains point 4
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

search left subtree
check if query rectangle contains point 5
(search hit)
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

search bottom subtree
stop since empty
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

return from function call
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

search top subtree
check if query rectangle contains point 6
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

query rectangle to left of splitting line
search only in left subtree
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

return from function call
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

Range search. Find all points in a query axis-aligned rectangle.
• Check if point in node lies in given rectangle.
• Recursively search left/bottom subdivision (if any could fall in rectangle).
• Recursively search right/top subdivision (if any could fall in rectangle).

Typical case. $R + \log N$.

Worst case (assuming tree is balanced). $R + \sqrt{N}$. 
Nearest neighbor search in a 2d tree

query point

find closest points in 2d tree
to green query point
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

search root node
compute distance from query point to 1
(update champion nearest neighbor)
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

query point is to the left of splitting line
search left subtree first
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

search left subtree
compute distance from query point to 3
(update champion)
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

query point is above splitting line
search top subtree first
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

search top subtree
compute distance from query point to 6
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

query point is to left of splitting line
search left subtree first
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

query point is to left of splitting line
search left subtree first
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

query point is above splitting line
search top subtree first
• Check distance from point in node to query point.
• Recursively search left/bottom subdivision (if it could contain a closer point).
• Recursively search right/top subdivision (if it could contain a closer point).
• Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

```
return from function call
search right subtree next
```
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

search right subtree
prune since nearest neighbor can't be in subdivision
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

return from function call
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Nearest neighbor = 5
Nearest neighbor search. Given a query point, find the closest point.
- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Typical case. $\log N$.

Worst case (even if tree is balanced). $N$.
**Kd tree**

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

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

![Diagram of Kd tree]

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!

Jon Bentley
**N-body simulation**

**Goal.** Simulate the motion of \( N \) particles, mutually affected by gravity.

Brute force. For each pair of particles, compute force.

\[ F = \frac{G m_1 m_2}{r^2} \]

http://www.youtube.com/watch?v=ua7YN4eL_w
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 \( N^2 \) ⇒ enables new research.