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

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>N</td>
<td>N</td>
<td>N/2</td>
<td>equals()</td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td></td>
</tr>
<tr>
<td>search hit</td>
<td>N/2</td>
<td>N</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>N</td>
<td>N/2</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>lg N</td>
<td>lg N</td>
<td>compareTo()</td>
</tr>
<tr>
<td>(ordered array)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td>compareTo()</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>N</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td>compareTo()</td>
</tr>
<tr>
<td>(unordered list)</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td></td>
</tr>
<tr>
<td>goal</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>compareTo()</td>
</tr>
<tr>
<td></td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td></td>
</tr>
<tr>
<td></td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td></td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

2-3 tree

You can read it as 2 or 3 children tree
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.
- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

search for H

search for B
2-3 tree demo

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

search for B

2-3 tree demo

Insert Operation

- Problem with Binary Search Tree: when the tree grows from leaves, it is possible to always insert to same branch. (worst-case)
- Instead of growing the tree from bottom, try to grow upwards.
  - If there is space in a leaf, simply insert it
  - Otherwise push nodes from bottom to top, if done recursively the tree will be balanced as it grows (increasing the height by introducing a new root)
- If we keep on inserting to same branch:

  BST:  
  ```
  9
  / 
  7  10
  |
  6
  ```

  2 or 3 Tree:  
  ```
  9
  / 
  8, 10
  |
  6
  ```

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.

insert K

K is greater than J
(go right)

search ends here

insert K

replace 2-node with
3-node containing K
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.

Search ends here

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.

search ends here

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.

replace 3-node with temporary 4-node containing Z

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.

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.

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.
- 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

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

Search in a 2-3 tree

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

• Insert L

• Height of tree increases by 1
**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:** $\lg N$
- **Best case:** $\log_3 N \approx 0.631 \lg N$.

- 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 inserts?</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.

Multiple Node Types

- In 2-3 Trees, the algorithm automatically balances the tree
- However, we have to keep track of two different node types, complicating the source code.
  - Nodes with one key
  - Nodes with two keys
- Instead of multiple nodes:
  - Multiple edge types; red and black
  - Rotations instead of Split

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.

- Larger key is root
- Red links "glue" nodes within a 3-node
- Black links connect 2-nodes and 3-nodes
- Corresponding red-black BST
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.
  - We will only allow one red link to simulate 2 keys in node
  - A node with two red links would be the same as having 3 keys
• Red links lean left (correct ordering)

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.

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

**Key property.** 1–1 correspondence between 2–3 and LLRB.

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;
    int N;
    boolean color;
    // color of link from parent to this node
    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.

```
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.

```
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.

**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**

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

**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).
Red-black BST insertion

Insert S

Insert E

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 insertion

red-black BST
Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion
Red-black BST insertion

Insert H

E

C

S

A

R

H

two left reds in a row
(rotate S right)

E

C

S

A

R

H

both children red
(flip colors)

E

C

R

A

H

S

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

right link red (rotate E left)

red-black BST

E
R
H
S

A

C

red-black BST

E
R
H
S

A

C

red-black BST

E
R
H
S

A

C
Red-black BST insertion

insert X

A C H R E X

right link red (rotate S left)

Red-black BST insertion

A C H R E X

red-black BST

A C H S R X

Red-black BST insertion

A C H S R X

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 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

Insert P

right link red
(rotate E left)

Red-black BST insertion

Insert P

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 insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion
Red-black BST insertion

A red-black BST is an augmented balanced binary search tree (BST) where each node is augmented with an extra attribute, called the color, that is either red or black. The coloring rules for a red-black tree are:

1. Every node is either red or black.
2. The root is black.
3. All leaves (NIL nodes) are black.
4. If a node is red, then both its children are black.
5. Every path from a given node to any of its descendant leaves contains the same number of black nodes.

Insertion in a red-black tree:

1. Perform the usual BST insertion.
2. If the newly inserted node is red, no further adjustments are needed.
3. If the newly inserted node is black, then:
   a. If the parent of the newly inserted node is red, then:
      i. The grandparent of the newly inserted node must be black.
      ii. If the grandparent of the newly inserted node is red, then:
         1. If the newly inserted node has a red sibling, then:
            a. The color of the sibling is changed to red, and the parent and grandparent are changed to black.
            b. The parent is a red-black tree.
         2. If the newly inserted node has a black sibling, then:
            a. The color of the sibling is changed to red, and the parent is changed to black.
            b. The grandparent is a red-black tree.
   b. If the parent of the newly inserted node is black, then:
      i. The parent is a red-black tree.
      ii. If the newly inserted node has a red sibling, then:
         a. The color of the sibling is changed to red, and the parent and grandparent are changed to black.
         b. The parent is a red-black tree.
      iii. If the newly inserted node has a black sibling, then:
         a. The color of the sibling is changed to red, and the parent is changed to black.
         b. The grandparent is a red-black tree.

LLRB tree insertion trace

An LLRB (Left-Lean Red-Black) tree is a self-balancing binary search tree that maintains the balance of the tree by using a technique that is similar to that of red-black trees. The LLRB tree is defined as follows:

1. Every node is either red or black.
2. If a node is red, then both its children are black.
3. Every path from a given node to any of its descendant leaves contains the same number of black nodes.

Insertion in an LLRB tree:

1. Perform the usual BST insertion.
2. If the newly inserted node is red, no further adjustments are needed.
3. If the newly inserted node is black, then:
   a. If the parent of the newly inserted node is red, then:
      i. The grandparent of the newly inserted node must be black.
      ii. If the grandparent of the newly inserted node is red, then:
         1. If the newly inserted node has a red sibling, then:
            a. The color of the sibling is changed to red, and the parent and grandparent are changed to black.
            b. The parent is an LLRB tree.
         2. If the newly inserted node has a black sibling, then:
            a. The color of the sibling is changed to red, and the parent is changed to black.
            b. The grandparent is an LLRB tree.
   b. If the parent of the newly inserted node is black, then:
      i. The parent is an LLRB tree.
      ii. If the newly inserted node has a red sibling, then:
         a. The color of the sibling is changed to red, and the parent and grandparent are changed to black.
         b. The parent is an LLRB tree.
      iii. If the newly inserted node has a black sibling, then:
         a. The color of the sibling is changed to red, and the parent is changed to black.
         b. The grandparent is an LLRB tree.

Java implementation:

```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 provide near-perfect balance.
Insertion in a LLRB tree: visualization

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

Balance in LLRB trees

Proposition. Height of tree is $\leq 2 \log N$ 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 $\sim 1.00 \log N$ in typical applications.
ST implementations: frequency counter

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Worst-case cost (after N inserts)</th>
<th>Average case (after N random inserts)</th>
<th>Ordered</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>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

ST implementations: summary

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.

---

**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.

---

**Searching in a B-tree**
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.

---

**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.

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

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

- 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 $h\times v$ rectangle.

Applications. Networking, circuit design, databases,...

Gridded implementation.
- Divide space into $M\times M$ 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.

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.
Clustering

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

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

- 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.