Advanced Algorithmics (6EAP)  
MTAT.03.238  
Trees  
Jaak Vilo  
2018 Fall

Contents
• Tree as a data model  
• Data structures  
• Search trees  
  – binary trees and balancing  
  – (2,4)-trees, B-trees  
  – k-d trees  
• Heaps  
• Union-find problem  
• ...

Trees
• Some of the very basic essence of computer science and programming  
• Chapter 5 – “The Tree Data Model” [pp 223-285] in  
  Foundations of Computer Science: C Edition  
  Alfred V. Aho, Jeffrey D. Ullman  
  W. H. Freeman (October 15, 1994)

Tree
• Acyclic graph  
  – root of a tree  
  – children, parents, siblings, internal nodes, leaves  
• Binary tree – node has 0, 1, or 2 children

Data model
• Abstraction
• File directory system  
• Hierarchical organisation structure  
  – divide and conquer  
• Hierarchical controlled vocabulary (simple ontology)  
• syntactic structure of a (sentence in a) language  
• syntax – e.g. paired parentheses  
• ...

Example: XHTML and CSS
• The nested tags define sub-trees

 Douglas Wilhelm Harder – Univ. Waterloo
Example: XHTML and CSS

- The nested tags define sub-trees

```html
<html>
<head>
<title>Hello World!</title>
</head>
<body>
<h1>This is a <u>Heading</u></h1>
<p>This is a paragraph with some <u>underlined</u> text.</p>
</body>
</html>
```

Example: XHTML and CSS

- This defines a single tree

```
<html>
<head>
<title>Hello World!</title>
</head>
<body>
<h1>This is a <u>Heading</u></h1>
<p>This is a paragraph with "test."</p>
</body>
</html>
```

Example: XHTML and CSS

- This may be rendered by a web browser

Terminology

- List is also a tree:

```
<html>
<head>
<title>Hello World!</title>
</head>
<body>
<h1>This is a <u>Heading</u></h1>
<p>This is a paragraph with "test."</p>
</body>
</html>
```

Terminology

- List is also a tree:

Root node: A
Leaf nodes: B, C, D, E, F, G
Degree of B: 2
Degree of I: 3
Every node has 1 parent except root node A has 0 parents
Depth of A: 3
Level of I: 3
Width at level 3: 6
Successors: children (F, G, J, K, L)
Path to K: A, H, I, K

Descendants of B: B, C, D, E, F, G
Ancestors of I: I, H, A

Every node is connected via a path to root A.
Terminology

Topologically equal to previous slide
Depends on application if order is important or not

Trie for \( P = \{ \text{he, she, his, hers} \} \)

Implementation of branching

Binary trees

Binary tree

• This peach tree is not a binary tree...
Binary Trees
Definition: Any node can have 0, 1 or 2 children
• A **full** node is a node where both the left and right sub-trees are non-empty trees

• Legend:
  - full nodes
  - neither
  - leaf nodes

Basic node structure

Binary Trees
• An **empty node** or a **null sub-tree** is any location where a new leaf node could be inserted

Perfect Binary Trees: Definition
• A perfect binary tree of height \( h \) is a binary tree where
  - All leaves have the same depth \( h \)
  - All other nodes are full

Perfect Binary Trees: Examples
• Perfect binary trees of height \( h = 0, 1, 2, 3 \) and 4

Q: size of such a tree? N leaves \( \rightarrow \) Total size?
Perfect Binary Trees

2^h + 1 – 1 Nodes

- Using the recursive definition, both sub-trees are perfect trees of height h = k – 1
- By assumption, each sub-tree has 2^k + 1 – 1 nodes
- Therefore the total number of nodes is
  \[(2^k + 1) + 1 + (2^k + 1) = 2^{k+1} - 1\]

Complete Binary Trees: Definition

- A complete binary tree filled at each depth from left to right:

Complete Binary Trees: Array Storage

- Fill the array following a breadth-first traversal:

left(i) = i\times2 
right(i) = i\times2 + 1 
parent(i) = \lfloor i/2 \rfloor

Complete Binary Trees: Array Storage

- To insert another node while maintaining the complete-binary-tree structure, we must insert into the next array location:

Traversal of a binary tree

function: Tree-Walk(x)
Traversals of a binary tree

- **Pre-order traversal:**
  ```
  Tree-Walk( x )
  if x ≠ NULL then
    Tree-Walk( left(x) )
    Tree-Walk( right(x) )
  ``

- **In-order traversal:**
  ```
  Tree-Walk(x)
  if x ≠ NULL then
    // pre-order operations
    Tree-Walk( left(x) )
    // in-order operations
    Tree-Walk( right(x) )
    // post-order operations
  ``

- **Post-order traversal:**
  ```
  Tree-Walk(x)
  if x ≠ NULL then
    Tree-Walk( left(x) )
    Tree-Walk( right(x) )
    // post-order operations
  ``

**Application: Expression Trees**

- **Expression trees**
  ```
  3(2a + c + a) + b/3 + (a – 2)
  ```

**Parenthesization**

```
{ 3 ( 9 ( 14 (17) (15) ) (10 (13) (23) ) ) ... }
```
Binary Trees
Application: Expression Trees
– internal nodes store operators
– leaves store operands
– no node has just one sub tree
– the order is not relevant for addition and multiplication (commutative)
– the order is relevant for subtraction and division (non-commutative)
– to ignore order completely, represent subtraction and division as unary operators

(a/b) = a * b\(^{-1}\)  \(\frac{a}{b}\) = a + (-b)

Evaluate the expression

```c
int Eval-Tree( x )
int val1, val2;
if x->op == 'l' return x->value; // x is a leaf, integer value
else
val1 = Eval-Tree( x->left )
val2 = Eval-Tree( x->right )
switch ( n->op ) {
  case '+': return val1 + val2;
  case '-': return val1 - val2;
  case '*': return val1 * val2;
  case '/': return val1 / val2;
}
```

General Trees: Design
• Children – in a linked list

Traversal of a general tree

```c
Tree-Walk( x )
if x ≠ NULL then
foreach c in children(x)
  Tree-Walk( c )
```
Traversal of a general tree

Tree-Walk( x )
if x ≠ NULL then
  // pre-order operations
  foreach c in children(x)
    Tree-Walk( c )
  // post-order operations

Depth-first Traversal

• We note that each node could be visited twice in such a scheme
  – the first time the node is approached, and
  – the last time it is approached.

Pre-order Depth-first Traversal

• Visiting each node first results in the sequence
  A, B, C, D, E, F, G, H, I, J, K, L, M

Post-order Depth-first Traversal

• Visiting the nodes with their last visit:

Parenthesised tree serialisation

• Passing such a visitor results in the output:
  (A(BC(D)(EF)(G))(HIJKLU)(M))
Breadth-First Traversal

• Breadth-first traversal would visit the nodes in the order:

Breadth-First Traversal

Breadth-First (x)

Breadth-First Traversal

Breadth-First (x)

1 enqueue(Q, x)
2 while not empty(Q)
3   x = dequeue(Q)
4   print x->name // process node x
5   foreach c in next-child(x)
6     enqueue(Q, c)

Breadth-First Traversal

Printing Directories

• Given the directory structure

Printing Directories

Printing Directories

Exercise

• Print the following statistics for a given (e.g. current working) directory:
  – subdirectory size (# of all subdirectories and files)
  – depth (maximal height)
  – width at all levels of depth...
  – maximal depth
  – largest directory in nr of subdirs and files in that directory
  – ...

Binary Search Tree (BST)

• MIT

Binary Search Tree (BST)

Binary Search Tree (BST)

Binary Search Tree (BST)

Binary Search Tree (BST)

Binary Search Tree (BST)

Binary Search Tree (BST)
35(of 60) did not draw that binary search tree ... 3 did AVL-like, 3 some other variant.

<table>
<thead>
<tr>
<th>What is the time complexity of:</th>
<th>Worst case</th>
<th>Average case</th>
<th>Could you implement it without consulting literature?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Quicksort</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2. Merge Sort</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3. Heap Sort</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4. Radix sort</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5. Theoretically best comparison based sort</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6. Theoretically best sorting (any)</td>
<td>6</td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

Examples

• Here we see a complete binary search tree, and a binary search tree which is close to being complete -- balanced

Examples

• There are many different representations of the same ordered data:

Operations on dynamic sets

SEARCH(x, k)
A query that, given a set S and a key value k, returns a pointer x to an element in S such that key[x] = k, or NIL if no such element belongs to S.

INSERT(S, x)
A modifying operation that augments the set S with the element pointed to by x. We usually assume that any fields in element x needed by the set implementation have already been initialized.

DELETE(S, x)
A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation uses a pointer to an element x, not a key value.)

MINIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.

MAXIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the largest key.

SUCCESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

PREDECESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

Operations - search

TREE-SEARCH(x, k)
1 if x = NIL or k = x.key
2 then return x
3 if k < x.key
4 then return TREE-SEARCH(x.left, k)
5 else return TREE-SEARCH(x.right, k)

Iterative search

ITERATIVE-TREE-SEARCH(x, k)
1 while x ≠ NIL and k ≠ x.key
2 if k < x.key
3 then x = x.left
4 else x = x.right
5 return x

(Tail) Recursion "unrolling" -- should be more efficient
### Min and Max

**Tree-Minimum (x)**
1. while left[x] ≠ NIL
2. x = left[x]
3. return x

**Tree-Maximum (x)**
1. while right[x] ≠ NIL
2. x = right[x]
3. return x

### Successor

**Tree-Successor (x)**
1. if right[x] ≠ NIL
2. then return Tree-Minimum(right[x])
3. y = parent[x]
4. while y ≠ NIL and x = right[y]
5. x = y; y = parent[y]
6. return y

### Insert a node

- Find such a node where “next” position is missing...

### Remove

- Suppose we wish to remove a node
- There are three situations: the node being removed
  - is a leaf node,
  - has exactly one child, or
  - is a full node (two children).

- If it is a leaf node, we can remove it:

- If the node has only one child, we can promote that child (with all the subtree underneath):
Remove

• If it is a full node, we copy the minimum element from the right sub-tree
• Recursively delete the value we copied

Example

• Consider the following tree
• We will twice remove the root

Example

• First, to remove 15, it is a full node
• We find the minimum element in the right sub-tree

Example

• We promote 42 to the root
• Proceed to remove 42 from the right sub-tree

Example

• This has one child, so we promote the entire sub-tree to replace 42

Example

• The root has been deleted, and the result is still a binary search tree
Example

• Next, let us remove 42
• Once again, it is a full node, so get the minimum element in the right sub-tree

Example

• We promote 45 to the root and proceed to delete 45 from the right sub-tree

Example

• The node 45 is a leaf node, so we may simply remove it

Example

• Thus, the final tree, having removed 15 and then 42 is

Reading

• CLRS: Binary Search Trees
• Visualisations:

Complexity...

• (Almost) all operations depend on the depth of the tree (or node affected)
• Binary search tree can get unbalanced, depth $O(n)$
• How to ensure this does not happen?
Balanced Binary Search Trees

• AVL trees
• 2-3 trees
• 2-3-4 trees
• B-trees
• Red-black trees

Balance

• If elements are added in random, tree is “automatically balanced” on average
• Otherwise: we must re-balance it ourselves...

AVL-trees

• Adelson-Velskii and Landis
  • http://en.wikipedia.org/wiki/AVL_tree
  • In an AVL tree, the heights of the two child subtrees of any node differ by at most one;
  • The AVL tree is named after its two inventors, G.M. Adelson-Velskky and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information."

Height of an AVL Tree

• In an AVL tree, the heights of the two child subtrees of any node differ by at most one;
  • Difference: -1, 0, 1
  • Re-balance using rotations when getting out of balance...
  • $O(\lg n)$ normal operations
  • Up to $O(\lg n)$ re-balancing operations of $O(1)$
  • an AVL tree's height is limited to $1.44 \lg n$
### Height of an AVL Tree

- If \( n = 10^6 \), the bounds on \( h \) are:
  - The minimum height: \( \log_2(10^6) - 1 \approx 19 \)
  - The maximum height: \( \log_2(10^6 / 1.8944) < 28 \)

An AVL tree’s height is strictly less than:

\[
\log_\phi (\sqrt{\phi + 1}) - 2 = \log_2(1) - 2 = \log_2(1) - 2 = 1.44\log_2(e) - 0.28
\]

where \( \phi \) is the golden ratio.

### Red-Black Trees

1. A node is either red or black.
2. The root is black. (This rule is used in some definitions and not others. Since the root can always be changed from red to black but not necessarily vice-versa this rule has little effect on analysis.)
3. All leaves are black.
4. **Both children of every red node are black.**
5. Every simple path from a node to a descendant leaf contains the same number of black nodes.

![Red-Black Trees](image)

**Figure 1:** The four cases for balancing a red-black tree.

### Red-black trees

This data structure requires an extra one-bit color field in each node.

**Red-black properties:**

1. Every node is either red or black.
2. The root and leaves (NIL’s) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node \( x \) to a descendant leaf have the same number of black nodes = black-height(\( x \)).

![Example of a red-black tree](image)

**Example of a red-black tree**
Height of a red-black tree

**Theorem.** A red-black tree with \( n \) keys has height

\[ h \leq 2 \lg(n + 1). \]

**Proof.** (The book uses induction. Read carefully.)

**Intuition:**
- Merge red nodes into their black parents.

---

**Proof (continued)**

- We have \( h' \geq h/2 \), since at most half the leaves on any path are red.
- The number of leaves in each tree is \( n + 1 \)
  \[ \Rightarrow n + 1 \geq 2^{h'} \]
  \[ \Rightarrow \lg(n + 1) \geq h' \geq h/2 \]
  \[ \Rightarrow h \leq 2 \lg(n + 1). \]

---

**Query operations**

**Corollary.** The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in \( O(\lg n) \) time on a red-black tree with \( n \) nodes.

---

**Modifying operations**

The operations INSERT and DELETE cause modifications to the red-black tree:
- the operation itself,
- color changes,
- restructuring the links of the tree via "rotations".
**Rotations**

Rotations maintain the inorder ordering of keys:
\[ a < b < c \Rightarrow \alpha \leq \beta \leq \gamma \]

A rotation can be performed in \( O(1) \) time.

**Insertion into a red-black tree**

**IDEA:** Insert \( x \) in tree. Color \( x \) red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**

- Insert \( x = 15 \).
- Recolor, moving the violation up the tree.
- **RIGHT-ROTATE(18).**

**Insertion into a red-black tree**

**IDEA:** Insert \( x \) in tree. Color \( x \) red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**

- Insert \( x = 15 \).
- Recolor, moving the violation up the tree.
- **LEFT-ROTATE(7) and recolor.**

**Insertion into a red-black tree**

**IDEA:** Insert \( x \) in tree. Color \( x \) red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**

- Insert \( x = 15 \).
- Recolor, moving the violation up the tree.
- **RIGHT-ROTATE(18).**
- **LEFT-ROTATE(7) and recolor.**

**Insertion into a red-black tree**

**IDEA:** Insert \( x \) in tree. Color \( x \) red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**

- Insert \( x = 15 \).
- Recolor, moving the violation up the tree.
- **RIGHT-ROTATE(18).**
- **LEFT-ROTATE(7) and recolor.**
**Pseudocode**

```plaintext
RB-INSERT(T, x)
TREE-INSERT(T, x)
    color[x] ← RED
    while x ≠ root(T) and color[par[x]] = RED
        do if par[x] = left[par[par[x]]]
            then y ← right[par[x]]
                if color[y] = RED
                    then (Case 1)
                        then (Case 2)
                            Case 2 falls into Case 3
                            Case 3
                    else ("then" clause with "left" and "right" swapped)
                        color[root[T]] ← BLACK
            if y ≠ right[par[x]]
                then (Case 2)
                    Case 2 falls into Case 3
                    Case 3
            else ("then" clause with "left" and "right" swapped)
                color[root[T]] ← BLACK
        done
    done
```

**Graphical notation**

Let $A$ denote a subtree with a black root.

All $A$'s have the same black-height.

---

**Case 1**

Recolor

(Or, children of $A$ are swapped.)

Push $C$'s black onto $A$ and $D$, and recurse, since $C$'s parent may be red.

---

**Case 2**

LEFT-ROTATE($A$)

Transform to Case 3.

---

**Case 3**

RIGHT-ROTATE($C$)

Done! No more violations of RB property 3 are possible.

---

**Analysis**

- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

**Running time:** $O(g(n))$ with $O(1)$ rotations, RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT (see textbook).
Other ideas

- Balancing can be an independent process — at night?
- Many search & insert & delete processes, and few rebalancing processes
- Local locking. Must ensure no deadlocks occur!

Properties of Red-Black trees

- No overhead for searching — efficient
- 100-200 lines of code, many symmetric cases
- Left-Leaning Red-Black trees (LLRB)
  - Robert Sedgewick:

Splay trees (Sleator, Tarjan 1985)

- Self-adjusting BST
- Recently accessed elements are brought to the top of the tree
- Repeated accesses will be executed faster!
- No extra bookkeeping
- Average log(n) but worst case O(n)

2-3, 2-3-4, B-trees

- Binary trees are useful for memory-based data structures
- Large databases and disk based systems would benefit of fewer reads of larger block sizes
- Organise data in a search tree that minimizes disk accesses
B-tree (m-way)

\[ h = O(\log_m n) \]

In practice: 3-5 accesses to disk...

B-tree properties

A B-tree of order \( m \) (the maximum number of children for each node) is a tree which satisfies the following properties:

- Every node has at most \( m \) children.
- Every node (except root and leaves) has at least \( \frac{m}{2} \) children.
- The root has at least two children if it is not a leaf node.
- All leaves appear in the same level, and carry information.
- A non-leaf node with \( k \) children contains \( k-1 \) keys.

Half-full property ensures that ...

- two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there is room to push one element up into the parent).

Example: Two-level Insertion

- Inserting 29
  - Leaf node is full, so we split it into two

Example: Two-level Insertion

- Parent node is full, so we must split it
Example: Two-level Insertion
• The root node must be updated

Example: Root Insertion
• Insert 67
• Leaf is full, so split it into two

Example: Root Insertion
• Parent is full, so split it into two

Example: Root Insertion
• Root is full, so split it into two

Example: Root Insertion
• Create a new root node

The creators of the B-tree structure, Rudolf Bayer and Ed McCreight, have not explained what, if anything, the B stands for. Douglas Comer suggests a number of possibilities:
• "Balanced," "Broad," or "Bushy" might apply [since all leaves are at the same level]. Others suggest that the "B" stands for Boeing [since the authors worked at Boeing Scientific Research Labs in 1972]. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.[1]
Variants of B-trees

• Keys and data in leaves or internal nodes
• Order statistics
• ...

Analogy between R-B and B-trees

k-d tree

• Multi-dimensional data
  – 2-dim (x,y)
  – 3D (x,y,z)
  – d-dim (x₁,...,x_d)
• Does a point belong to a set?
• What is the closest point? (other data structures)
• ...

2-d data (xy, gps, coordinates)

2-D tree (x,y coordinates)

kd tree
Suppose we wish to partition the following points in a 2-dimensional \textit{k}d\textit{-}tree:

- (0.03, 0.90), (0.37, 0.04), (0.56, 0.78),
- (0.01, 0.48), (0.41, 0.89), (0.95, 0.07),
- (0.97, 0.09), (0.54, 0.65), (0.04, 0.61),
- (0.73, 0.69), (0.46, 0.58), (0.08, 0.89),
- (0.04, 0.41), (0.94, 0.02), (0.33, 0.07),
- (0.55, 0.54), (0.06, 0.05), (0.04, 0.06),
- (0.74, 0.97), (0.29, 0.15), (0.05, 0.88),
- (0.23, 0.23), (0.55, 0.02), (0.02, 0.97),
- (0.05, 0.07), (0.06, 0.28), (0.09, 0.55),
- (0.02, 0.91), (0.05, 0.97), (0.68, 0.42),
- (0.97, 0.18)

The first step is to order the points based on the 1st coordinate and find the median:

- (0.01, 0.48), (0.02, 0.91), (0.02, 0.97), (0.03, 0.90),
- (0.04, 0.06), (0.04, 0.41), (0.04, 0.61), (0.05, 0.07),
- (0.05, 0.88), (0.05, 0.97), (0.06, 0.05), (0.06, 0.28),
- (0.08, 0.89), (0.09, 0.55), (0.23, 0.23), (0.29, 0.15),
- (0.33, 0.07), (0.37, 0.04), (0.41, 0.89), (0.46, 0.58),
- (0.54, 0.65), (0.55, 0.02), (0.55, 0.54), (0.56, 0.78),
- (0.68, 0.42), (0.73, 0.69), (0.74, 0.97), (0.89, 0.02),
- (0.93, 0.97), (0.97, 0.18), (0.97, 0.18)

The median point, (0.29, 0.15), forms the root of our \textit{k}d\textit{-}tree.

Starting with the first partition, we order these according to the 2nd coordinate:

- (0.06, 0.05), (0.04, 0.06), (0.07, 0.05), (0.23, 0.23),
- (0.09, 0.55), (0.06, 0.28), (0.04, 0.41), (0.03, 0.90),
- (0.01, 0.48), (0.02, 0.91), (0.02, 0.97), (0.04, 0.61),
- (0.06, 0.05), (0.04, 0.06), (0.03, 0.91), (0.03, 0.97),
- (0.02, 0.91), (0.05, 0.97), (0.07, 0.18)

This point creates the left child of the root.
 kd-Trees

- Starting with the second partition, we also order these according to the 2nd coordinate:
  - (0.55, 0.02), (0.94, 0.02), (0.37, 0.04), (0.33, 0.07),
  - (0.95, 0.07), (0.97, 0.09), (0.97, 0.18),
  - (0.68, 0.42), (0.55, 0.54), (0.46, 0.58), (0.54, 0.65), (0.73, 0.69),
  - (0.56, 0.78), (0.41, 0.89), (0.74, 0.97)
kd-Trees

• Finally, the last point, a leaf node, falls within the given box

kd-Trees

• A useful application of a kd-tree provides an efficient data structure for counting the number of points which fall within a given k-dimensional rectangle

kd-Trees

• This is used in image processing: locating objects within a scene, ray tracing, etc.
• Find the points which lie in the quadrant \([0.5, 1] \times [0, 0.5]\)

kd-Trees

• The traversal rules we will follow are:
  – we always match the coordinate corresponding to the level we are current at
  – if that coordinate is less than the corresponding interval of the box, we only need to visit the right sub-tree
  – if that coordinate is greater than the corresponding interval, we need only visit the left sub-tree
  – otherwise, we check if the root is in the box and we visit both sub-trees

kd-Trees

• Starting with the left sub-tree:
  \(0.94 \in [0.5, 1]\)
• We note that
  \((0.94, 0.02) \in [0.5, 1] \times [0, 0.5]\)
and we visit both sub-trees

Nearest neighbour search

http://upload.wikimedia.org/wikipedia/commons/9/9c/KDTree-animation.gif
• *kd*-trees are not suitable for efficiently finding the nearest neighbour in high dimensional spaces.
• As a general rule, if the dimensionality is $D$, then number of points in the data, $N$, should be $N \gg 2^D$.
• Otherwise, when *kd*-trees are used with high-dimensional data, most of the points in the tree will be evaluated and the efficiency is no better than exhaustive search.
• The problem of finding NN in high-dimensional data is thought to be *NP-hard* \[^{[2]}\], and approximate nearest-neighbour methods are used instead.

---

**High dimensionality**

• Data often comes in high-dimensional form
• Curse of dimensionality
  – *K*-d tree nodes become empty after a few levels already...
• Everything is “far” from everything else
  – Difference along even one dimension makes them far from each other

---

Random Projection (RP) trees

Dmytro Fishman

---

Random projection tree (construction)

Choose a random vector

---

Random projection tree (construction)
Choose a split point (median of projected points in this case)

This creates a partition of the original data into two subsets

We must store a random vector and the median at each node of the tree

Again choose a random vector

Project all points on this new vector
Define a split

Random projection tree (construction)

Repeat recursively until limit of points in each leaf is reached (3 in this case)

Random projection tree (construction)

We need to find a query point

Random projection tree (search)
We need to find a query point. At each level of the tree we will project query point on a random vector and figure from which side of the median this projection lies.
By constructing several trees you can improve the accuracy of the algorithm.
### Methods average complexity

<table>
<thead>
<tr>
<th>Method</th>
<th>Building Time</th>
<th>Search Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-d Tree</td>
<td>(O(n \log(n^2)))</td>
<td>(O(n \log(n) + nd))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>RP Tree</td>
<td>(O(nd \log(n)))</td>
<td>(O(nd \log(n)))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

Random Projection trees and low dimensional manifolds by Dasgupta et al. ([link](http://cseweb.ucsd.edu/~dasgupta/papers/rptree-stoc.pdf))
Random projection tree

- Places hyper planes between two arbitrary points each time dividing hyperspace into two equal parts
- Also keeps track of number of points in the node

See also (Wikipedia)

- implicit kd-tree
- min/max kd-tree
- Quadtree
- Octree
- Bounding Interval Hierarchy
- Nearest neighbor search
- Klee's measure problem
- kd-trie

Quadtree – divide area recursively to quadrants
Quadtree

- 2-dimensional
- 4 quadrants
- Either point or area based

Octree – 3D, 8 children

R-Tree

Overlapping
Minimal bounding boxes
B-Tree analog on k-D

R-Tree

Variants

- \( R, R^+, R^* \)

**Difference between \( R^+ \) trees and \( R \) trees**

\( R^+ \) trees are a compromise between \( R \) trees and \( kd \)-trees; they avoid overlapping of internal nodes by inserting an object into multiple leaves if necessary.

\( R^+ \) trees differ from \( R \) trees in that:

- Nodes are not guaranteed to be at least half filled
- The entries of any internal node do not overlap
- An object ID may be stored in more than one leaf node

**Advantages**

Because nodes are not overlapped with each other, point query performance benefits since all spatial regions are covered by at most one node. A single path is followed and fewer nodes are visited than with the \( R \)-tree.
Priority queue

- Insert Q, x
- Retrieve next x from Q s.t. x.value is largest
- Sorted list implementation:
  - O(n) to insert x into right place
  - O(1) access, O(1) delete

Binary heap

Complete – missing nodes only at the lowest level
Heap property – on any path parent has higher priority
Typically: min-heaps
Priority queue
  insert ( Q, x )
  pop Q

Binary heap - Insert

Insert into a next allowed place
Make sure heap property is restored

Binary heap – Insert – “Bubble up”
Use Array based implementation

![](image)

left = i*2;
right = i*2 + 1;
parent = i/2;

Insert

```c
insert( int A[], int x, int *last) {
    (*last)++ ;
    A[*last] = x;
    bubbleUp( A, *last ) ;
}
```

Bubble up

```c
BubbleUp( int A[], int i) {
    while ( (i>1) && A[i] > A[i/2] ){
        swap( A, i, i/2 ) ;
        i=i/2;
    }
}
```

Delete (max)

```
    25
   /  
  18   5
 /    /  
3  2   7
```

• Remove top value (make free space)
• Remove last element
• Insert to top value location, then bubble down to the correct place

Binary heap – Delete – “Bubble down”
Cost

- Insert – \(O(\log n)\)
- Delete – \(O(\log n)\)

Heap-sort

- Heapify the array
- while not empty
  - pop_largest
  - copy to next free place

Heapify... in linear time

- last \(n/2\) – ignore
- \(n/4\) - bubble down (at most by 1 level)
- \(n/8\) – bubble down (at most by 2 levels)
- ... 
  \[ \sum_{i=1}^{\log n} \frac{i}{2^i} \leq \frac{n}{2} \sum_{i=1}^{\infty} \frac{i}{2^i} \]

1/2 + 1/4 + 1/8 + 1/16 + ... = 1
+ 1/4 + 1/8 + 1/16 + ... = 1/2
1/8 + 1/16 + ... = 1/4
... 
1/8 nodes, 3 steps down...
Sum = 2
**Introduction to Algorithms**

**LECTURE 11**

Augmenting Data Structures

- Dynamic order statistics
- Methodology
- Interval trees

**Professor Charles E. Leiserson**


**Dynamic order statistics**

**OS-SELECT**(i, S): returns the i\(^{th}\) smallest element in the dynamic set S.

**OS-RANK**(x, S): returns the rank of x ∈ S in the sorted order of S’s elements.

**IDEA:** Use a red-black tree for the set S, but keep subtree sizes in the nodes.

**Notation for nodes:**

**Example of an OS-tree**

```
M
  /
 /   
C     D
  /   /   /
A     F    H
  /       /   
1     3    1
```

```
size[x] = size[left[x]] + size[right[x]] + 1
```

**Example**

**OS-SELECT**(root, 5)

```
S
  /
 /   
C     D
  /   /   /
A     F    H
  /       /   
1     3    1
```

```
i = 5
k = 6
```

```
M
  /
 /   
C     D
  /   /   /
A     F    H
  /       /   
1     3    1
```

```
i = 3
k = 2
```

```
N
  /
 /   
H     Q
  /   /   
1     1
```

```
i = 1
k = 1
```

Running time = \(O(h) = O(\log n)\) for red-black trees.

**Data structure maintenance**

**Q.** Why not keep the ranks themselves in the nodes instead of subtree sizes?

**A.** They are hard to maintain when the red-black tree is modified.

**Modifying operations:** INSERT and DELETE.

**Strategy:** Update subtree sizes when inserting or deleting.

27.09.18
In computer science, an interval tree, also called a segment tree or segtree, is an ordered tree data structure to hold intervals. Specifically, it allows one to efficiently find all intervals that overlap with any given interval or point. It is often used for windowing queries, for example, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene.

The trivial solution is to visit each interval and test whether it intersects the given point or interval, which requires $\Theta(n)$ time, where $n$ is the number of intervals in the collection. Since a query may return all intervals, for example if the query is a large interval intersecting all intervals in the collection, this is asymptotically optimal; however, we can do better by considering output-sensitive algorithms, where the runtime is expressed in terms of $m$, the number of intervals produced by the query.

**Data-structure augmentation**

**Methodology:** (e.g., order-statistics trees)

1. Choose an underlying data structure (red-black trees).
2. Determine additional information to be stored in the data structure (subtree sizes).
3. Verify that this information can be maintained for modifying operations (RB-INSERT, RB-DELETE — don’t forget rotations).
4. Develop new dynamic-set operations that use the information (OS-SELECT and OS-RANK).

These steps are guidelines, not rigid rules.

**Interval trees**

**Goal:** To maintain a dynamic set of intervals, such as time intervals.

\[ low[i] = 7 \quad 10 = high[i] \]

\[ 4 \pseite 8 \pseite 11 \pseite 15 \pseite 17 \pseite 19 \]

**Query:** For a given query interval $i$, find an interval in the set that overlaps $i$.

**Following the methodology**

1. Choose an underlying data structure.
   - Red-black tree keyed on low (left) endpoint.
2. Determine additional information to be stored in the data structure.
   - Store in each node $x$ the largest value $m[x]$ in the subtree rooted at $x$, as well as the interval $int[x]$ corresponding to the key.
**New operations**

4. Develop new dynamic-set operations that use the information.

   **INTERVAL-SEARCH(i)**
   
   ```
   x ← root
   while x ≠ NIL and (low[i] > high[int(x)])
     or low[int(x)] > high[i])
     do i and int(x) don’t overlap
       if left(x) ≠ NIL and low[i] ≤ m[left(x)]
         then x ← left(x)
       else x ← right(x)
   return x
   ```

---

**Cases for overlap:**

- Queries i
- Int(x)
- `max(tree)`

**Example 1:** `INTERVAL-SEARCH([14,16])`

- `x ← root`
- `[14,16]` and `[17,19]` don’t overlap
  - `14 ≤ 18 ⇒ x ← left(x)`

**Example 1:** `INTERVAL-SEARCH([14,16])`

- `x ← root`
- `[14,16]` and `[5,11]` don’t overlap
  - `14 > 8 ⇒ x ← right(x)`
Example 1: \(\text{INTERVAL-SEARCH}([14,16])\)

- [14,16] and [15,18] overlap
- return [15,18]

Example 2: \(\text{INTERVAL-SEARCH}([12,14])\)

- \(x \leftarrow \text{root}\)
- [12,14] and [17,19] don’t overlap
- \(12 \leq 18 \Rightarrow x \leftarrow \text{left}[x]\)

Example 2: \(\text{INTERVAL-SEARCH}([12,14])\)

- [12,14] and [5,11] don’t overlap
- \(12 > 8 \Rightarrow x \leftarrow \text{right}[x]\)

Analysis

- Time = \(O(h) = O(\lg n)\), since \(\text{INTERVAL-SEARCH}\) does constant work at each level as it follows a simple path down the tree.
- List all overlapping intervals:
  - Search, list, delete, repeat.
  - Insert them all again at the end.
- Time = \(O(k \lg n)\), where \(k\) is the total number of overlapping intervals.
- This is an output-sensitive bound.
- Best algorithm to date: \(O(k + \lg n)\).
Correctness

Theorem. Let \( L \) be the set of intervals in the left subtree of node \( x \), and let \( R \) be the set of intervals in \( x \)'s right subtree.
- If the search goes right, then
  \[ \{ i' \in L : i' \text{ overlaps } i \} = \emptyset. \]
- If the search goes left, then
  \[ \{ i' \in L : i' \text{ overlaps } i \} = \emptyset \]
  \[ \Rightarrow \{ i' \in R : i' \text{ overlaps } i \} = \emptyset. \]

In other words, it's always safe to take only 1 of the 2 children: we'll either find something, or nothing was to be found.

Go right:

Correctness proof

Proof. Suppose first that the search goes right.
- If \( \text{left}[x] = \text{NIL} \), then we're done, since \( L = \emptyset \).
- Otherwise, the code dictates that we must have \( \text{low}[j] > m[\text{left}[x]] \). The value \( m[\text{left}[x]] \) corresponds to the high endpoint of some interval \( j \in L \), and no other interval in \( L \) can have a larger high endpoint than \( \text{high}[j] \).

\[ \text{high}[j] = m[\text{left}[x]] \]

\[ \text{low}(i) \]

\[ \therefore \{ i' \in L : i' \text{ overlaps } i \} = \emptyset. \]

Left:

Proof (continued)

Suppose that the search goes left, and assume that
\[ \{ i' \in L : i' \text{ overlaps } i \} = \emptyset. \]
- Then, the code dictates that \( \text{low}[j] \leq m[\text{left}[x]] = \text{high}[j] \) for some \( j \in L \).
- Since \( j \in L \), it does not overlap \( i \), and hence
  \[ \text{high}[j] < \text{low}[j]. \]
- But, the binary-search-tree property implies that for all \( i' \in R \), we have \( \text{low}[j] \leq \text{low}[i'] \).
- But then \( \{ i' \in R : i' \text{ overlaps } i \} = \emptyset. \]

Treap

- Tree + Heap = Treap
- BST + Heap
  of (key, priority) pair at the same time.

Combining info: Treap

- Heap and binary search tree properties together
- Treap
  - Red – heap
  - Blue – BST

A treap. Letters are search keys; numbers are priorities.

Use rotations to “push up”

- Can be used to make a random-like tree: priorities can be assigned by random, unique values...
- In computer science, a treap is a binary search tree that orders the nodes by adding a random priority attribute to a node, as well as a key. The nodes are ordered so that the keys form a binary search tree and the priorities obey the max heap order property. The name treap is a portmanteau of tree and heap.
- A portmanteau word (pronounced /pɔːtˈmæn.təʊ/ (help info)) is used broadly to mean a blend of two (or more) words and narrowly in linguistics fields to mean only a blend of two or more function words.
### Bulk operations

- Union of two Treaps
- Intersection
- Set Difference

These rely on two helper functions – `split` and `merge`

### Split on k

- Insert \((k, \text{high\_priority})\)
  - Left is smaller, right subtree larger than \(k\)


### Union-find

- Domain \(X = \{x_1, \ldots, x_n\}\)
- \(x_i\) belongs to a set \(S_i\)
- **Non-intersecting sets.**

- **Union** of sets: \(S_i' = S_i \cup S_j\)
- **Find**: Which set \(S_i\) does an element \(x_j\) belong to?

### Link elements until “0”

<table>
<thead>
<tr>
<th>Set</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>n</td>
</tr>
</tbody>
</table>

Find = \(O(1)\)
Union = \(O(\text{n})\)

<table>
<thead>
<tr>
<th>Set</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>n</td>
</tr>
</tbody>
</table>

\(s_1 = \{3, 4\}\)
\(s_2 = \{6, 8\}\)

Find = \(O(\text{n})\)
Union = \(O(\text{n})\)
Union-Find
A data structure for maintaining a collection of disjoint sets

Course: Data Structures
Lecturer: Uri Zwick
March 2008

- At every find – “flatten the tree”

Union-Find
Make(x): Create a set containing x
Union(x,y): Unite the sets containing x and y
Find(x): Return a representative of the set containing x

Fun applications: Generating mazes

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>make(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>make(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>make(16)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>find(6)=find(7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>union(6,7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>find(7)=find(11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>union(7,11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Choose edges in random order and remove them if they connect two different regions
Fun applications: Generating mazes

Generating mazes – a larger example

Construction time -- $O(n^2 \alpha(n^2))$

More serious applications:

• Maintaining an equivalence relation
• Incremental connectivity in graphs
• Computing minimum spanning trees
• ...

Union Find
Represent each set as a rooted tree
Union by rank   Path compression

The parent of a vertex $x$ is denoted by $p(x)$
Find($x$) traces the path from $x$ to the root
Union by rank on its own gives $O(\log n)$ find time.

A tree of rank $r$ contains at least $2^r$ elements.

If $x$ is not a root, then $\text{rank}(x) = \text{rank}(p[x])$.

Nesting / Repeated application:

\[
\begin{align*}
f^{(0)}(n) &= n \\
f^{(i)}(n) &= f(f^{(i-1)}(n)), \text{ for } i > 0 \\
f(n) &= \log n
\end{align*}
\]

Path Compression

Union by rank - pseudocode

```python
Function make-set(x):
    p[x] = x
    rank[x] = 0

Function union(x, y):
    if rank[x] > rank[y] then
        p[y] = x
    else if rank[x] < rank[y] then
        p[x] = y
    else
        p[x] = y
        rank[y] += 1

Function find(x):
    if p[x] != x then
        p[x] = find(p[x])
    return p[x]
```

Union-Find

\[
\begin{array}{|c|c|c|}
\hline
& \text{make} & \text{link} & \text{find} \\
\hline
\text{Worst case} & O(1) & O(1) & O(\log n) \\
\hline
\text{Amortized} & O(1) & O(\alpha(n)) & O(\alpha(n)) \\
\hline
\end{array}
\]

Ackermann’s function:

\[
A_k(n) = \begin{cases} 
  n+1 & \text{if } k = 1, \\
  A_{k-1}(A_{k-1}(n)) & \text{if } k > 1.
\end{cases}
\]

$A_1(n) = n + 1$

$A_2(n) = 2n + 1$

$A_3(n) = 2^{n+1}(n + 1) - 1$

$A_4(n) = ?$
**Ackermann’s function (modified)**

\[ A_k(n) = \begin{cases} 
  n + 1 & \text{if } k = 1, \\
  A_{k-1}^{n+1}(n) & \text{if } k > 1.
\end{cases} \]

\[ \bar{A}_k(n) = \begin{cases} 
  2n & \text{if } k = 2, \\
  \bar{A}_{k-1}^n(1) & \text{if } k > 2.
\end{cases} \]

\[ \bar{A}_2(n) = 2n \]

\[ \bar{A}_3(n) = 2^n \]

\[ \bar{A}_4(n) = \text{tower}(n) = 2^{2^{\cdots^n}} \]

---

**Inverse functions**

\[ F(n) \implies f(n) = \min\{k \geq 1 \mid F(k) \geq n\} \]

\[ F(n) = n + 1 \quad f(n) = n - 1 \]

\[ F(n) = 2n \quad f(n) = \lfloor \frac{\sqrt{n}}{2} \rfloor \]

\[ F(n) = 2^n \quad f(n) = \lfloor \log_2 n \rfloor \]

\[ F(n) = \text{tower}(n) \quad f(n) = \log^* n \]

---

**Inverse Ackermann function**

\[ \alpha_r(n) = \min\{k \geq 1 \mid A_k(r) \geq n\} \]

\[ \alpha(n) = \alpha_1(n) = \min\{k \geq 1 \mid A_k(1) \geq n\} \]

\[ \alpha(n) \] is the inverse of the function \[ A_n(1) \]

\[ A_n(1) = A_n^{(2)}(1) = A_{n-1}(A_{n-1}(1)) > A_{n-1}(n) \]

---

**Amortized cost of make**

Actual cost: \( O(1) \)

\[ \Delta \Phi: \quad 0 \]

Amortized cost: \( O(1) \)

---

**Amortized cost of link**

Actual cost: \( O(1) \)

The potentials of \( y \) and \( z_1, \ldots, z_k \) can only decrease

The potentials of \( x \) is increased by at most \( \omega(n) \)

\[ \Delta \Phi \leq \alpha(n) \]

Actual cost: \( O(\omega(n)) \)

---

**Amortized cost of find**

\[ \phi(x) = (\alpha(n) - \text{level}(x)) \cdot \text{rank}[x] - \text{index}(x) \]

is either unchanged or is decreased
Suppose that:

\[ 0 < i < j < l \]

level(x_i) = level(x_j)

Amortized cost of \textit{find}

\[ \Delta \Phi \leq (\alpha(n) + 1) - (l + 1) \]

Actual cost:

\[ l + 1 \]

Amortized cost:

\[ \alpha(n) + 1 \]