Advanced Algorithmics (6EAP)

MTAT.03.238

Trees

Jaak Vilo
2017 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

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

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

- The nested tags define sub-trees

```
<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

```
<some tags>
  "This is a "
  "underlined" text."
</some tags>
```

Terminology

- List is also a tree:

```
A
B
C
D
```

Terminology

- Descendants (of B) = B,C,D,E,F,G
  - a subtree with a root B
- Ancestors of I = I,H,A
  - Every node is connected via a path to root
Terminology

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

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

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

Binary Trees

Definition

- A full binary tree is where each node is:
  - a full node, or
  - a leaf node

- This has applications in
  - expression trees and Huffman encoding

Q: size of such a tree? N leaves -> Total size?

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

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 \( 2^k - 1 \) nodes
- Therefore the total number of nodes is

\[ (2^{k+1} - 1) + 1 + (2^{k+1} - 1) \]

\[ = 2^{k+1} - 1 \]
**Complete Binary Trees: Definition**

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

```
   1
  / \
 2   3
/ \ / \   \ 
4  5 6  7  8
```

**Complete Binary Trees: Array Storage**

- Fill the array following a breadth-first traversal:

```
left(i) = i*2  
right(i) = i*2+1  
parent(i) = i/2
```

**Traversal of a binary tree**

Function: `Tree-Walk(x)`

```plaintext
if x ≠ NULL then
    Tree-Walk(left(x))
    Tree-Walk(right(x))
```
Traversal of a binary tree

Tree-Walk( x )
  if x ≠ NULL then
    // pre-order operations
    Tree-Walk( left(x) )
    // in-order operations
    Tree-Walk( right(x) )
    // post-order operations

Traversal of a binary tree - size

int Tree-Size( x )
  if x == NULL then return 0
  return
    Tree-Size( left(x) ) +
    Tree-Size( right(x) ) + 1

Traversal of a binary tree – parenthesisation

Tree-Walk( x )
  if x ≠ NULL then
    print “(”, x.value ;
    Tree-Walk( left(x) ) ;
    Tree-Walk( right(x) ) ;
    print “)” ;

Parenthesisation

( 3 + 9 ( 14 ( 17 ) ( 10 ( 13 22 ) ) ) ) ...

Binary Trees
Application: Expression Trees

• Expression trees
  3(2a + c + a) + b/3 + (a – 2)

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)  (a – b) = a + (-b)
Binary Trees

Application: Expression Trees

- Performing appropriate tree traversals allows you to convert the representation
- Post-order results in reverse Polish:

\[
3 \ 2 \ a \ + \ c \ + \ + \ b \ 3 \ a \ 2 \ - \ +
\]

Evaluate the expression

```
int Eval-Tree( x )
int val1, val2;
if x->op == '+' 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

Traversals of a general tree

```
Tree-Walk( x )
if x != NULL then
    foreach c in children(x)
        Tree-Walk( c )
```

// pre-order operations

```
foreach c in children(x)
    Tree-Walk( c )
```
// post-order operations

Traversals of a general tree

```
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
    print "("; // pre-order operations
    foreach c in children(x)
        Tree-Walk( c )
    print ")"; // 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(B(C(D))(E(F(G)))(H(I)(J)(K)(L))(M)))`

Breadth-First Traversal

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

Printing Directories

- Given the directory structure

/  
usr/  
bin/  
local/  
var/  
adm/  
cron/  
log/

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 tree where values of the keys have a special order:

  values(left subtree) < value(root) <= values(right subtree)

<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. Quick sort</td>
<td>1.</td>
<td>1.</td>
<td>1.</td>
</tr>
<tr>
<td>2. Merge sort</td>
<td>2.</td>
<td>2.</td>
<td>2.</td>
</tr>
<tr>
<td>3. Heap sort</td>
<td>3.</td>
<td>3.</td>
<td>3.</td>
</tr>
<tr>
<td>4. Radix sort</td>
<td>4.</td>
<td>4.</td>
<td>4.</td>
</tr>
<tr>
<td>5. Theoretically best comparison based sort</td>
<td>5.</td>
<td>5.</td>
<td>5.</td>
</tr>
<tr>
<td>6. Theoretically best sorting (any)</td>
<td>6.</td>
<td>6.</td>
<td>6.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Draw a binary search tree (BST) after inserting:</th>
<th>2, 5, 6, 1, 3 into an initially empty tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 (of 60) did not draw that binary search tree</td>
<td>3 did AVL-like, 3 some other variant.</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

Operations on dynamic sets

SEARCHES, I)

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(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(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(X)

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

MAXIMUM(X)

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

SUCCESSOR(X)

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

PREDECESSOR(X)

A query that, given an element x whose key is in 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).

Remove

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

Remove

- 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-Velsky and E.M. Landis, who published it in their 1962 paper “An algorithm for the organization of information.”

Height of an AVL Tree

• If \( n = 88 \), the worst- and best-case scenarios differ in height by only 2:

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:

\[
\log_\varphi(\sqrt[3]{(n+2)} - 2) = \frac{\log(\sqrt[3]{n+2})}{\log(\varphi)} - 2 = \frac{\log(2)\log(\sqrt[3]{n+2})}{\log(\varphi)} - 2 \approx 1.44\log(n+2) - 6.328
\]

where \( \varphi \) 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

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

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 = \( \text{black-height}(x) \).

Example of a red-black tree

4. All simple paths from any node \( x \) to a descendant leaf have the same number of black nodes = \( \text{black-height}(x) \).

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

**Theorem.** A red-black tree with $n$ keys has height

$$h \leq 2 \log(n + 1).$$

**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 $n + 1$
  $$\implies n + 1 \geq 2^h
  \implies \log(n + 1) \geq h' \geq h/2
  \implies h \leq 2 \log(n + 1).$$

**Query operations**

**Corollary.** The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\log n)$ 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:
- $a \in A, b \in B, c \in \gamma \implies a \leq A \leq b \leq B \leq c$.

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(16).**

---

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.**

---

Pseudocode

```
RB-INSERT(T, x)

TREE-INSERT(T, x)

color[x] <- RED  // only RB property 3 can be violated

while $x \neq root(T)$ and color[parent[x]] = RED

  do if $x = left[parent[x]]$

    then $y <- right[parent[x]]$

    if color[y] = RED

      then (Case 1)

      else if $x = right[parent[x]]$

        then (Case 2)  // Case 2 falls into Case 3 (Case 3)

      else ("then" clause with "/left" and "right" swapped)

        color[root(T)] <- BLACK

  do if $y = aunt/uncle of $x$

    then # orphan

    do if $y = right[parent[x]]$

      then # right twin

      else # left twin

    do if $y = left[parent[x]]$

      then # right twin

      else # left twin

    do if $y = aunt/uncle of $x$

      then # orphan

    do if $y = right[parent[x]]$

      then # right twin

      else # left twin

    do if $y = left[parent[x]]$

      then # right twin

      else # left twin

    do if $y = aunt/uncle of $x$

      then # orphan

    do if $y = right[parent[x]]$

      then # right twin

      else # left twin

    do if $y = left[parent[x]]$

      then # right twin

      else # left twin

    do if $y = aunt/uncle of $x$

      then # orphan
**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:** $O(\lg n)$ 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)

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 $\lceil m/2 \rceil$ 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
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.

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, ..., xd]
- Does a point belong to a set?
- What is the closest point? (other data structures)
- ...

2-d data (xy, gps, coordinates)

(7,2) (5,4) (2,3) (4,7) (9,6) (8,1) ...

2-D tree (x,y coordinates)

kd tree

kd-Trees

- Suppose we wish to partition the following points in a 2-dimensional kd-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, 0.97), (0.68, 0.42), (0.97, 0.18)]
**kd-Trees**

- The first step is to order the points based on the 1\textsuperscript{st} coordinate and find the median:

\begin{itemize}
  \item (0.01, 0.48), (0.02, 0.91), (0.02, 0.97), (0.03, 0.90),
  \item (0.04, 0.06), (0.04, 0.41), (0.04, 0.61), (0.05, 0.07),
  \item (0.05, 0.88), (0.06, 0.97), (0.06, 0.44), (0.06, 0.78),
  \item (0.08, 0.89), (0.09, 0.55), (0.23, 0.23), (0.29, 0.15),
  \item (0.33, 0.07), (0.37, 0.04), (0.41, 0.48), (0.46, 0.58),
  \item (0.44, 0.21), (0.46, 0.97), (0.46, 0.78), (0.47, 0.31),
  \item (0.54, 0.65), (0.55, 0.02), (0.55, 0.54), (0.56, 0.78),
  \item (0.68, 0.42), (0.73, 0.28), (0.74, 0.37), (0.84, 0.07),
  \item (0.84, 0.44), (0.86, 0.78), (0.86, 0.44)
\end{itemize}

- The median point, (0.29, 0.15), forms the root of our kd-tree

**kd-Trees**

- This partitions the remaining points into two sets:

\begin{itemize}
  \item \{(0.01, 0.48), (0.02, 0.91), (0.02, 0.97), (0.03, 0.90),
  \item (0.04, 0.06), (0.04, 0.41), (0.04, 0.61), (0.05, 0.07),
  \item (0.05, 0.88), (0.06, 0.97), (0.06, 0.44), (0.06, 0.78),
  \item (0.08, 0.89), (0.09, 0.55), (0.23, 0.23), (0.29, 0.15),
  \item (0.33, 0.07), (0.37, 0.04), (0.41, 0.48), (0.46, 0.58),
  \item (0.44, 0.21), (0.46, 0.97), (0.46, 0.78), (0.47, 0.31),
  \item (0.54, 0.65), (0.55, 0.02), (0.55, 0.54), (0.56, 0.78),
  \item (0.68, 0.42), (0.73, 0.28), (0.74, 0.37), (0.84, 0.07),
  \item (0.84, 0.44), (0.86, 0.78), (0.86, 0.44)
\end{itemize}

- Starting with the first partition, we order these according to the 2\textsuperscript{nd} coordinate:

\begin{itemize}
  \item (0.04, 0.03), (0.04, 0.62), (0.05, 0.87), (0.23, 0.24),
  \item (0.04, 0.72), (0.04, 0.44), (0.05, 0.44), (0.05, 0.86),
  \item (0.04, 0.44), (0.05, 0.96), (0.02, 0.91), (0.02, 0.97),
  \item (0.04, 0.86), (0.06, 0.86), (0.05, 0.87)
\end{itemize}

- Starting with the second partition, we also order these according to the 2\textsuperscript{nd} coordinate:

\begin{itemize}
  \item (0.54, 0.53), (0.44, 0.52), (0.27, 0.64), (0.22, 0.47),
  \item (0.44, 0.73), (0.47, 0.20), (0.47, 0.20), (0.46, 0.84),
  \item (0.54, 0.42), (0.46, 0.46), (0.54, 0.44), (0.73, 0.64),
  \item (0.04, 0.76), (0.04, 0.86), (0.36, 0.87)
\end{itemize}
**kd-Trees**

- This point creates the right child of the root

**kd-Trees**

- Next, ordering the partitioned elements by the 1st coordinate, we choose the medians to find the children of the left child (0.09, 0.55):

  - (0.01, 0.48), (0.04, 0.41), (0.04, 0.41), (0.05, 0.47),
  - (0.04, 0.45), (0.06, 0.28), (0.23, 0.28),
  - (0.02, 0.91), (0.02, 0.97), (0.03, 0.90),
  - (0.04, 0.61), (0.04, 0.80), (0.08, 0.96)

**kd-Trees**

- Doing the same with the two right partitions, we get the children of the right child of the root:

  - (0.14, 0.67), (0.16, 0.63), (0.14, 0.62), (0.23, 0.56),
  - (0.06, 0.27), (0.07, 0.35), (0.07, 0.38)
  - (0.04, 0.49), (0.04, 0.45), (0.04, 0.41), (0.14, 0.56),
  - (0.07, 0.78), (0.07, 0.78), (0.74, 0.87)

**kd-Trees**

- At the next level, we order the points again based on the 2nd coordinate and choose the medians:

  - (0.04, 0.96), (0.06, 0.94), (0.01, 0.48)
  - (0.04, 0.95), (0.05, 0.93), (0.05, 0.92)
  - (0.04, 0.98), (0.05, 0.95), (0.05, 0.97)
  - (0.05, 0.96), (0.05, 0.98), (0.05, 0.97)
  - (0.05, 0.97), (0.06, 0.96), (0.06, 0.97)
  - (0.06, 0.89), (0.07, 0.89), (0.07, 0.90)
  - (0.07, 0.90), (0.07, 0.91), (0.07, 0.92)
  - (0.55, 0.52), (0.57, 0.54), (0.55, 0.57)
  - (0.55, 0.57), (0.56, 0.58), (0.56, 0.58)

**kd-Trees**

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

**kd-Trees**

- The result is a 2-dimensional kd-tree of the given 31 points
**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.

- 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

- 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**

- *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**, 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

K-dimensional tree

- Partitions space into multidimensional “buckets”
- If done in classical way ends up with a lot of empty buckets to search through (in fact exponentially many)
- Needs to keep track of number of points per bucket

Fast approximate max-correlation queries
K-dimensional tree

Random projection tree

Random projection tree

Random projection tree
Random projection tree

$y - y_0 = m(x - x_0)$

$m = -1/2$

$(-1/2)x + 5 = y$

$A(2, 4)$

$B(6, 2)$

$C(4, 3)$
Random projection tree

\[ \frac{1}{2}x + 1 = y \]

Y (4,4)

\[ 2 \times 4 - 5 - 4 = -1 < 0 \]
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

Methods average complexity

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

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

http://nns.tume-maatm.ee/

Octree – 3D, 8 children

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

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)

- 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
  1. remove last (x), decrease size,
  2. copy largest to new free place
  3. insert x to L[1], bubble down

Build heap
- n times insert to heap = O( n log n )

“Sort”
- n times repeat remove largest = O( n log n )

Total: O(n log n) method

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)
- ...
  \[ \sum_{i=1}^{\log_{2}n} \frac{in}{2^{i+1}} \leq \frac{n}{2} \sum_{i=1}^{\infty} \frac{in}{2^{i}} \]

1/2 + 1/4 + 1/8 + 1/16 + ... = 1
+ 1/4 + 1/8 + 1/16 + ... = 1/2
1/8 + 1/16 + ... = 1/4

Sum = 2
Introduction to Algorithms
6.046J/18.401J

LECTURE 11
Augmenting Data Structures
• Dynamic order statistics
• Methodology
• Interval trees

Prof. 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

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

Selection

Implementation trick: Use a sentinel
(dummy record) for NIL such that size[NIL] = 0.

OS-SELECT(x, i) ➔ i-th smallest element in the
subtree rooted at x

k ← size[left[x]] + 1 ➔ k = rank(x)
if i = k then return x
if i < k
then return OS-SELECT(left[x], i)
else return OS-SELECT(right[x], i - k)

(OS-RANK is in the textbook.)

Example

OS-SELECT(root, 5)

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.
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.
New operations

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

\textbf{INTERVAL-SEARCH}(i)

\begin{itemize}
  \item \(x \leftarrow \text{root}\)
  \item \text{while } x = \text{NIL} \text{ and } (\text{low}(i) > \text{high}(\text{inf}(x)) \text{ or } \text{low}(\text{inf}(x)) > \text{high}(i))
  \item \text{do } i \leftarrow i \text{ and inf}(x) \text{ don't overlap}
  \item \text{if } \text{left}(x) = \text{NIL} \text{ and } \text{low}(i) \leq m[\text{left}(x)]
  \item \text{then } x \leftarrow \text{left}(x)
  \item \text{else } x \leftarrow \text{right}(x)
  \item \text{return } x
\end{itemize}
Example 1: \textsc{interval-search}([14,16])

- \[14,16\] and \[15,18\] overlap
- \textbf{return} \[15,18\]

Example 2: \textsc{interval-search}([12,14])

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

Example 2: \textsc{interval-search}([12,14])

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

Example 2: \textsc{interval-search}([12,14])

- \[12,14\] and \[15,18\] don't overlap
- \(12 > 10 \Rightarrow x \leftarrow \text{right}[x]\)

Example 2: \textsc{interval-search}([12,14])

- \[12,14\]
- \(x = \text{NIL} \Rightarrow \) no interval that overlaps \[12,14\] exists

Analysis

Time = \(O(h) = O(\lg n)\), since \textsc{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 \textit{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.

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 $\text{low}[i] > m[\text{left}[x]]$. 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 $\text{high}[j]$.

\[ \ldots j \quad \text{high}[j] = m[\text{left}[x]] \quad \text{low}(i) \]

- Therefore, $\{ i' \in L : i' \text{ overlaps } i \} = \emptyset$.

Proof (continued)

Suppose that the search goes left, and assume that$
\{ i' \in L : i' \text{ overlaps } i \} = \emptyset.$

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

**Combining info: insert (18,20)**

- insert 18, priority 20

**Delete max**

- delete max priority...

**Combining info**

- delete max priority...

**Combining info**

- delete max priority...
Combining info

• delete max priority...

Treap

• Can be used to make a random-like tree: priorities can be assigned by random, unique values...
  • http://en.wikipedia.org/wiki/Treap
  • 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 /ˈpɔːtˌmæn.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, high_priority)
  – Left is smaller, right subtree larger than k
  
  • https://en.wikipedia.org/wiki/Treap

Union-find

• Domain X = \{ x_1, ... , x_n \}
• x_i belongs to a set S_i
  • Non-interesecting sets.
  • Union of sets: S'_i = S_i \cup S_j
  • Find: Which set S_i does an element x_j belong to?

• Sets = \{ \{1\}, \{2\}, ... \{n\} \}
• Non-overlapping, each value belongs to a set
• Merge sets i, j (give new set id i, remove j)
  – Union
  • Which set does x belong to?
  – Find
Union-Find

A data structure for maintaining a collection of disjoint sets

Course: Data Structures
Lecturer: Uri Zwick
March 2008

• 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
Union Find

- `make(x)` takes `O(1)` time.
- `union(a, b)` takes `O(\alpha(n))` time for any `a`, `b`.
- `find(x)` takes `O(\alpha(n))` time for any `x`.
- Amortized complexity:
  - `make(x)` takes `O(1)` time.
  - `union(a, b)` takes `O(\alpha(n))` time for any `a`, `b`.
  - `find(x)` takes `O(\alpha(n))` time for any `x`.

Fun applications: Generating mazes

Choose edges in random order and remove them if they connect two different regions.

Generating mazes – a larger example

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

Fun applications: Generating mazes

1 | 2 | 3 | 4
---|---|---|---
5 | 6 | 7 | 8
9 | 10 | 11 | 12
13 | 14 | 15 | 16

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 \( v \) is denoted by \( p(v) \)

Find() traces the path from \( v \) to the root

Union by rank

A tree of rank \( r \) contains at least \( 2^r \) elements

If \( v \) is not a root, then \( \text{rank}(v) = \text{rank}(p(v)) \)

Path Compression

Union Find - pseudocode

<table>
<thead>
<tr>
<th>Function</th>
<th>\text{make-set}(x)</th>
<th>\text{union}(x, y)</th>
<th>\text{find}(x)</th>
<th>\text{link}(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{p}[x]</td>
<td>( x )</td>
<td>\text{link}(\text{find}(x), \text{find}(y))</td>
<td>\text{find}(x)</td>
<td>\text{link}(x, y)</td>
</tr>
<tr>
<td>\text{rank}[x]</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Function \text{find}(x)

If \( p[x] \neq x \) then
\[ p[x] = \text{find}(p[x]) \]

Return \( p[x] \)

Function \text{link}(x, y)

If \( \text{rank}[x] > \text{rank}[y] \) then
\[ p[y] = x \]

Else
\[ p[x] = y \]

If \( \text{rank}[x] = \text{rank}[y] \) then
\[ \text{rank}[x] = \text{rank}[x] + 1 \]

Worst case

<table>
<thead>
<tr>
<th>make</th>
<th>link</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>

Amortized

<table>
<thead>
<tr>
<th>make</th>
<th>link</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>( O(\alpha(n)) )</td>
<td>( O(\alpha(n)) )</td>
</tr>
</tbody>
</table>
Nesting / Repeated application

\[ f^{(i)}(n) = f(f(\ldots(f(n))\ldots)) \]
\[ f^{(i)}(n) = f(f^{(i-1)}(n)), \text{for } i > 0 \]
\[ f(n) = n + 1 \quad f^{(5)}(n) = n + 5 \]
\[ f(n) = 2n \quad f^{(7)}(n) = 2^7n \]
\[ f(n) = 2^n \quad f^{(5)}(n) = 2^{2^n} \]
\[ f(n) = \log{n} \quad f^{(2)}(n) = \log\log{n} \]

Ackermann’s function

\[ A_k(n) = \begin{cases} 
\begin{align*}
\frac{n+1}{A_{k-1}(n)} & \text{if } k = 1, \\
A_{k+1}(n) & \text{if } k > 1.
\end{align*}
\end{cases} \]
\[ A_1(n) = n + 1 \]
\[ A_2(n) = 2n \]
\[ A_3(n) = 2^{n+1}(n+1) - 1 \]
\[ A_4(n) = ? \]

Ackermann’s function (modified)

\[ \tilde{A}_k(n) = \begin{cases} 
\begin{align*}
2n & \text{if } k = 2, \\
\tilde{A}_{k-1}(n) & \text{if } k > 2.
\end{align*}
\end{cases} \]
\[ \tilde{A}_2(n) = 2n \]
\[ \tilde{A}_3(n) = 2^n \]
\[ \tilde{A}_4(n) = \text{tower}(n) = 2^{2\ldots2} n \]

Inverse functions

\[ F(n) \implies f(n) = \min\{k \geq 1 \mid F(k) \geq n\} \]
\[ F(n) = n + 1 \quad f(n) = n - 1 \]
\[ F(n) = 2n \quad f(n) = \left\lceil \frac{n}{2} \right\rceil \]
\[ F(n) = 2^n \quad f(n) = \left\lfloor \log_2 n \right\rfloor \]
\[ F(n) = \text{tower}(n) \quad f(n) = \log^* n \]

Inverse Ackermann function

\[ \alpha_r(n) = \min\{k \geq 1 \mid A_k(r) \geq n\} \]
\[ \alpha(n) = \alpha_1(n) = \min\{k \geq 1 \mid A_k(1) \geq n\} \]
\[ \alpha(n) \text{ is the inverse of the function } A_n(1) = A_{n-1}(1) = A_{n-1}(A_{n-1}(1)) > A_{n-1}(1) \]

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

Suppose that:

\[ 0 < i < j < l \]
\[ \text{level}(x_i) = \text{level}(x_j) \]

\[ \Delta \Phi \leq \left( \text{rank}[x_i] \right) \]
\[ \text{rank}[p[x_i]] \]
\[ \text{rank}[x_j] \]

The only nodes that can retain their potential are:
the first, the last and the last node of each level

Amortized cost: \( \alpha(n) + 1 \)