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

- This defines a single tree

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 may be rendered by a web browser

Terminology

- List is also a tree:

```
    A
   / \  
  B   C
 / \  /  
D   F E   G
```

Terminology

- Descendants (of B) = B,C,D,E,F,G

- 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

- 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

- 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 $\rightarrow$ 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 $2^{k+1} - 1$ nodes
- Therefore the total number of nodes is

$$2^h + 1 + (2^{h+1} - 1)$$

$$= 2^{h+2} - 1$$

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:

\[
\begin{array}{ccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15
\end{array}
\]

left(i) = \(i \times 2\)  right(i) = \(i \times 2 + 1\)

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

Traversal of a binary tree

Tree-Walk( x )

\[
\text{if } x \neq \text{NULL then}
\]

// pre-order operations

Tree-Walk( left(x) )

// in-order operations

Tree-Walk( right(x) )

// post-order operations

Traversal of a binary tree - size

```c
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 \neq \text{NULL then}

    print "(" , x.value ;
    Tree-Walk( left(x) ) ;
    Tree-Walk( right(x) ) ;
    print ")" ;
```
Parenthesisation
( 3 ( 9 ( 14 (17) (15) ) (10 (13) (23) ) ) ) ... 

Binary Trees
Application: Expression Trees

• Expression trees
\[3(2a + c + a) + b/3 + (a - 2)\]

• Observations:
  – internal nodes store operators
  – leaf nodes store operands
  – no nodes have just one sub tree
  – the order is not relevant for
    • addition and multiplication (commutative)
    • subtraction and division (non-commutative)
  – to ignore order completely, represent subtraction and division as unary operators
    \[(a/b) - a * b^{-1} \quad (a - b) - a + (-b)\]

Evaluate the expression

```c
int Eval-Tree( x )
{
  int val1, val2;
  if( x->op == 'i' ) 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
  // 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(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: [link](http://ocw.mit.edu/ocw/index.php/EECS/6-046J/VideoLectures/detail/embed09.htm)

• Binary tree where values of the keys have a special order:

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

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(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(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

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
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
• MIT
• 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-Velski and Landis
- 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."

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: \( \log_2(10^6) - 1 = 19 \)
  - the maximum height : \( \log_2(10^6 / 1.8944) < 28 \)

Re-balancing

- Can be done during each insertion-deletion
- AVL, Red-Black, ...
- or each lookup (e.g. Splay trees)
- Special re-balancing processes when computer otherwise idle
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

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

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

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.

Height 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 of 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 \( \text{SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR} \)
all run in \( O(\lg n) \) time on a red-black tree with \( n \) nodes.

**Modifying operations**

The operations \( \text{INSERT} \) and \( \text{DELETE} \) 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:
- \( a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c \).

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(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.
- **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**

```
RB-INSERT(T, x)
    TREE-INSERT(T, x)
    color[x] ← RED  \( \rightarrow \) only RB property 3 can be violated
    while \( x \neq root[T] \) and color[par[x]] = RED
        do if \( p[x] = left[par[x]] \)
            then \( y ← right[par[x]] \)  \( \rightarrow \) \( y \) = aunt/uncle of \( x \)
                if \( color[y] = RED \)
                    then (Case 1)
                        else if \( x = right[par[x]] \)
                            then (Case 2)  \( \rightarrow \) Case 2 falls into Case 3 (Case 3)
                                else (**then** clause with “left” and “right” swapped)
                                    color[par[x]] ← BLACK
```

---
Graphical notation

Let Δ denote a subtree with a black root.
All Δ’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).

Bottom-Up Insertions

- Suppose that A and D, respectively were swapped
- In both these cases, we perform similar rotations as before, and we are finished
Bottom-Up Insertions

• In the other case, where both children of the grandparent are red, we simply swap colours, and recurs back to the root

![Diagram showing red-black tree](image1.png)

• If, at the end, the root is red, it can be coloured black

![Diagram showing red-black tree](image2.png)

Next we color the new node. If we color it red, we risk violating the first color invariant. If we color it black, we risk violating the second color invariant. However, note that the first invariant is a local property and the second invariant is a global property. In the hope that a local property will be easier to fix than a global property, we color the new node red.

If the parent of the new node is also red, then we have violated the first color invariant and need to rearrange and recolor the tree to restore the invariant. This is where rotations are used in ordinary red-black trees. Instead of rotations we use the following balancing transformation:

Take the red child, the red parent, and the (black) grandparent and locally balance these three nodes by making the smallest and largest nodes children of the middle node. Then color the middle node red and the other two nodes black. The middle node is linked back into the tree in place of the former black grandparent.

This balancing transformation is illustrated in Figure 1.

![Figure 1: The four cases for balancing a red-black tree](image3.png)

Complexity

• Rebalancing may need to rebalance the whole path up to the root ⇒ $O(\log n)$

![Diagram showing red-black tree](image4.png)
### 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:

#### 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

#### Red-black representation of a 2-3-4 tree

### 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 $\lceil m/2 \rceil$ 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. [1]
*Multi-dimensional data*
- 2-dim \((x,y)\)
- 3D \((x,y,z)\)
- \(d\)-dim \((x_1, \ldots, x_d)\)
*Does a point belong to a set?*
*What is the closest point? (other data structures)*

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

\[
\begin{align*}
(0.01, 0.48), & (0.02, 0.91), (0.02, 0.97), (0.03, 0.90), \\
(0.04, 0.06), & (0.04, 0.41), (0.04, 0.89), (0.05, 0.07), \\
(0.06, 0.08), & (0.06, 0.97), (0.06, 0.01), (0.06, 0.12), \\
(0.08, 0.08), & (0.09, 0.13), (0.10, 0.22), (0.10, 0.11), \\
(0.12, 0.07), & (0.13, 0.07), (0.14, 0.86), (0.14, 0.56), \\
(0.14, 0.61), & (0.15, 0.82), (0.15, 0.16), (0.14, 0.78), \\
(0.16, 0.42), & (0.17, 0.49), (0.17, 0.97), (0.04, 0.01), \\
(0.06, 0.07), & (0.07, 0.08), (0.07, 0.18)
\end{align*}
\]

**kd-Tree**
- The median point, \((0.29, 0.15)\), forms the root of our \(kd\)-tree.
kd-Trees

- This partitions the remaining points into two sets:

$$\{(0.01, 0.48), (0.02, 0.01), (0.03, 0.97), (0.04, 0.00),
(0.06, 0.43), (0.08, 0.43), (0.04, 0.44), (0.05, 0.07),
(0.05, 0.88), (0.05, 0.97), (0.06, 0.16), (0.06, 0.28),
(0.08, 0.91), (0.06, 0.51), (0.23, 0.12)\}$$

$$\{(0.33, 0.07), (0.37, 0.04), (0.41, 0.89), (0.46, 0.58),
(0.54, 0.43), (0.55, 0.23), (0.54, 0.54), (0.56, 0.78),
(0.68, 0.42), (0.73, 0.45), (0.74, 0.97), (0.94, 0.23),
(0.85, 0.73), (0.87, 0.66), (0.97, 0.16)\}$$

kd-Trees

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

$$\{(0.04, 0.02), (0.04, 0.06), (0.05, 0.07), (0.23, 0.23),
(0.04, 0.28), (0.04, 0.61), (0.05, 0.48), (0.01, 0.68),
(0.04, 0.44), (0.05, 0.06), (0.02, 0.08), (0.05, 0.87),
(0.01, 0.08), (0.08, 0.89), (0.05, 0.07)\}$$

kd-Trees

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

$$\{(0.55, 0.02), (0.06, 0.02), (0.07, 0.04), (0.24, 0.27),
(0.94, 0.27), (0.67, 0.06), (0.87, 0.18), (0.64, 0.44),
(0.55, 0.54), (0.49, 0.54), (0.44, 0.03), (0.73, 0.49),
(0.54, 0.78), (0.43, 0.86), (0.74, 0.07)\}$$

kd-Trees

- This point creates the left child of the root

$$\{(0.06, 0.05), (0.04, 0.41), (0.04, 0.07),
(0.05, 0.43), (0.04, 0.06), (0.05, 0.28), (0.23, 0.23),
(0.02, 0.91), (0.02, 0.97), (0.03, 0.95), (0.04, 0.41),
(0.05, 0.88), (0.08, 0.89), (0.08, 0.95)\}$$

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):

$$\{(0.04, 0.48), (0.06, 0.06), (0.04, 0.41), (0.04, 0.07),
(0.04, 0.85), (0.06, 0.28), (0.92, 0.91),
(0.05, 0.01), (0.05, 0.07), (0.08, 0.89)\}$$
**kd-Trees**

- Doing the same with the two right partitions, we get the children of the right child of the root:
  - (0.33, 0.07), (0.37, 0.04), (0.55, 0.02), (0.94, 0.02), (0.95, 0.07), (0.97, 0.18), (0.41, 0.89), (0.46, 0.58), (0.54, 0.65), (0.55, 0.54), (0.56, 0.78), (0.73, 0.69), (0.74, 0.97)

  - (0.04, 0.06), (0.04, 0.41), (0.01, 0.48), (0.06, 0.05), (0.23, 0.23), (0.06, 0.28), (0.03, 0.90), (0.02, 0.91), (0.02, 0.97), (0.05, 0.88), (0.06, 0.89), (0.05, 0.57), (0.04, 0.58), (0.04, 0.86), (0.41, 0.89), (0.74, 0.57)

**kd-Trees**

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

**kd-Trees**

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

**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

**Nearest neighbour search**

- 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

**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
Random projection tree: partitioning

- 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**

- 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

---

**Quadtree** – divide area recursively to quadrants

**Octree** – 3D, 8 children

**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

Insert

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

Bubble up

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

Cost
- 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)
- ...
  \[ \sum_{i=1}^{\log n} \frac{in}{2^i} \leq \frac{n}{2} \sum_{i=1}^{\infty} \frac{in}{2^i} \]
**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:**

```
key
size
```

**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 viewpoint, 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 Θ(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.

Example of insertion

Handling rebalancing

Don’t forget that RB-INSERT and RB-DELETE may also need to modify the red-black tree in order to maintain balance.

• **Recolorings**: no effect on subtree sizes.
• **Rotations**: fix up subtree sizes in O(1) time.

Example:

 Interval trees

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

\[ i = [7, 10] \]

\[ \text{low}(i) = 7 \quad \text{high}(i) = 10 \]

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 \( im[x] \) corresponding to the key.

Interval, key = interval.left
Example interval tree

\[ m[x] = \max \begin{cases} \text{high}[m[x]] \\ \text{m[left][x]} \\ \text{m[right][x]} \end{cases} \]

Modifying operations

3. Verify that this information can be maintained for modifying operations.
   - INSERT: Fix m’s on the way down.
   - Rotations — Fixup = \(O(1)\) time per rotation:

   \[ \text{Total Insert time} = O(\log n); \text{ DELETE similar.} \]

New operations

4. Develop new dynamic-set operations that use the information.
   \[
   \text{INT\_SEARCH(i)} \\
   x <- \text{root} \\
   \text{while } x \neq \text{NIL} \text{ and (low}[i] \geq \text{high}[m[x]]) \cup (\text{low}[m[x]] > \text{high}[i]) \\
   \text{do } \text{if } \text{left}[x] \neq \text{NIL} \text{ and low}[i] \leq \text{m[left][x]} \\
   \text{then } x <- \text{left}[x] \\
   \text{else } x <- \text{right}[x] \\
   \text{return } x
   \]

Example 1: \(\text{INT\_SEARCH}([14,16])\)

\[ \text{[14,16] and [5,11] don't overlap} \]

\[ 14 > 8 \Rightarrow x <- \text{right}[x] \]

Example 1: \(\text{INT\_SEARCH}([14,16])\)

\[ \text{[14,16] and [15,18] overlap} \]

\[ \text{return } [15,18] \]
Example 2: INTERVAL-SEARCH([12,14])

\[ x \leftarrow \text{root} \]

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

\[ x \leftarrow \text{right}(x) \]

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

Example 2: INTERVAL-SEARCH([12,14])

\[ x \leftarrow \text{root} \]

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

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

Analysis

Time = \( O(h) = O(\lg n) \), since 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
  \[ \{ i' \in L : i' \text{ overlaps } / \} = \emptyset. \]

- If the search goes left, then
  \[ \{ i' \in R : i' \text{ overlaps } / \} = \emptyset \]
  \[ \Rightarrow \{ i' \in R : i' \text{ overlaps } i \} = \emptyset. \]

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.
**Treat**

- Tree + Heap = Treap

- BST + Heap

  of (key,priority) pair at the same time.

---

**Correctness proof**

**Proof.** Suppose first that the search goes right.
- If left[x] = Nil, then we’re done, since L = ∅.
- Otherwise, the code dictates that we must have low[i] > m[left[x]]. The value m[left[x]] corresponds to the high endpoint of some interval j ∈ L, and no other interval in L can have a larger high endpoint than high[j].

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

- Therefore, \{i’ ∈ L : i’ overlaps i \} = ∅.

**Proof (continued)**

Suppose that the search goes left, and assume that \{i’ ∈ L : i’ overlaps i \} = ∅.
- Then, the code dictates that \text{low}[i] ≤ m[\text{left}[x]] = \text{high}[j] for some j ∈ L.
- Since j ∈ 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’ ∈ R, we have \text{low}[j] ≤ \text{low}[i’].
- But then \{i’ ∈ R : i’ overlaps i \} = ∅.

---

**Combining info: Treap**

- **Heap and binary search tree properties together**

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

• insert 18, priority 20

Delete max

• 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 /pɔːtˈmæntəʊ/ (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.
Union-find

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

Sets = $\{ \{1\}, \{2\}, \ldots, \{n\} \}$
- 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

Find $= O(1)$
Union $= O(n)$

At every find — “flatten the tree”
Union-Find

A data structure for maintaining a collection of disjoint sets

Course: Data Structures
Lecturer: Uri Zwick
March 2008

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

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

Generating mazes – a larger example

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

- 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 $\text{rank}(y) = \text{rank}(p[x])$

Path Compression
### Union Find - Pseudocode

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

Function union(x, p):
    link(find(x), find(y))

Function find(x):
    if p[x] ≠ x then
        p[x] ← find(p[x])
    return p[x]
```

### Union-Find

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

### Nesting / Repeated application

\[ f^{(i)}(n) = f(f(\ldots f(n)\ldots)) \]

- \( f^{(0)}(n) = n \)
- \( f^{(i)}(n) = f(f^{(i-1)}(n)) \), for \( i > 0 \)

\[
\begin{align*}
  f(n) &= n + 1 \\
  f(n) &= 2n \\
  f(n) &= \log n \\
  f^{(1)}(n) &= n + 5 \\
  f^{(2)}(n) &= 2^n \\
  f^{(3)}(n) &= 2^{2^n} \\
  f^{(2)}(n) &= \log \log n
\end{align*}
\]

### Ackermann’s function

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

- \( A_1(n) = n + 1 \)
- \( A_2(n) = 2n + 1 \)
- \( A_3(n) = 2^{n+1}(n+1) - 1 \)
- \( A_4(n) = ? \)

### Ackermann’s function (modified)

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

\[ A_k(n) = \begin{cases} 
  2n & \text{if } k = 2, \\
  A_{k-1}(1) & \text{if } k > 2.
\end{cases} \]

- \( A_2(n) = 2n \)
- \( A_3(n) = 2^n \)
- \( A_4(n) = \text{tower}(n) = 2^{2^\ldots^2} \)

### Inverse functions

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

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

The first "column" "diagonal"

---

**Amortized cost of make**

**Actual cost:** \( O(1) \)

\[ \Delta \Phi = 0 \]

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

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

Amortized cost: \( O(\alpha(n)) \)

---

**Amortized cost of find**

Suppose that:

\[ 0 < i < j < l \]

\[ \text{level}(x_i) = \text{level}(x_j) \]

\[ A^{(\text{index}(x_i)+1)}(\text{rank}[x_i]) = A_{\text{level}(x_i)}(A^{\text{index}(x_i)}(\text{rank}[x_i])), \text{rank}[x_i]) \leq A_{\text{level}(x_i)}(\text{rank}[p[x_j]]) \leq \text{rank}[p[x_j]] \leq \text{rank}[x_j] \]

\( \alpha(n) \) is decreased \( \uparrow \)

---

Amortized cost of find

The only nodes that can retain their potential are:

- the first
- the last
- and the last node of each level

\[ \text{Actual cost: } l + 1 \]

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

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