Advanced Algorithmics (4AP)
Trees
Jaak Vilo
2009 Spring

Tree

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

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
• ...

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
• ...

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)

Example: XHTML and CSS

• The nested tags define sub-trees

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

• This defines a single tree

Example: XHTML and CSS

• This may be rendered by a web browser

Terminology

- **Root A**
- **Leaf (D,F,G,J,K,L,M)**
- **Degree** B = 2, degree I = 3
- **Every node has 1 parent except root has 0 parents**
- **Depth** = 3
- **Level** C,E,I,M = 2
- **Width** = 6 (at level 3)
- **Successor** = children siblings (F,G), (J,K,L)
- **Path to K** = A,H,I,K

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**
- Depends on application if order is important or not
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

Binary Trees

• This peach tree is not a binary tree...

Basic node structure

• 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
Perfect Binary Trees
Definition
- Standard 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 $h-1$.
- By assumption, each sub-tree has $2^{h-1} - 1$ nodes.
- Therefore the total number of nodes is
  \[
  (2^{h-1} - 1) + 1 + (2^{h-1} - 1) = 2^{h+1} - 1
  \]

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:

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

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

Binary Trees
Application: Expression Trees

• 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 complete, represent subtraction
    and division as unary operators
    \((a/b) = a \times b^{-1}\) \((a - b) = a + (-b)\)

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 - + +   
```

Binary Trees
Application: Expression Trees

• Expression trees

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

Traversals 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
Evaluate the expression

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

Traversal of a general tree

```c
TreeWalk(x)
if x != NULL then
    // pre-order operations
    foreach c in children(x)
        TreeWalk(c)
    // post-order operations
```

Trie for P={he, she, his, hers}

General Trees: Design

- Children – in a linked list

Implementation

```c
template <class Object>
class GeneralTree {
private:
    Object element;  // the stored in the node
    SingleList< GeneralTree<Object> > children;  // a linked list of general trees
public:
    Object retrieve() {
        return element;
    }
    // ...
};
```
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.

Traversal of a binary tree

```java
Traverse( x )
if x != NULL then
    print("(", x=value ;
    Traverse( left(x) )
    Traverse( right(x) )
    print ")" ;
```

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

Depth-First Traversals

- Passing such a visitor results in the output:
  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:

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 to MIT course material]

• Binary tree where values of the keys have a special order:
  values(left subtree ) < value(root) <= values (right subtree)

Breadth-First Traversal

• Performing such a traversal would visit the nodes in the order:

Examples

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

Breadth-First Traversal

```plaintext
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 )
```

Examples

• There are many different representations of the same ordered data:
Operations on dynamic sets

SEARCH(S, 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.

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

Operations - search

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

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...

Iterative search

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

(Tail) Recursion “unrolling” – should be more efficient
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 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

Remove

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

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

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

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

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

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?

Traversal of a binary tree

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

Balance

• If elements are added in random, tree is “automatically balanced” on average

• Otherwise: we must re-balance it ourselves...

Reading

• CLRS: Binary Search Trees

• Visualisations:
Balanced Binary Search Trees
- MIT [link]
- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- Red-black trees

Height of an AVL Tree
- If \( n = 88 \), the worst- and best-case scenarios differ in height by only 2:

AVL-trees
- Adelson-Velskii and Landis
  - [link]
- 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 \( n = 10^6 \), the bounds on \( h \) are:
  - The minimum height: \( \log_2(10^6) - 1 \approx 19 \)
  - The maximum height: \( \log_2(10^6 / 1.8944) < 28 \)

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

• 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 \( \lg n \)
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

Height of a red-black tree

Theorem. A red-black tree with $n$ keys has height $h \leq 2 \log_2(n+1)$.

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 \( h \leq 2 \log_2(n + 1) \).

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

**INTUITION:**
- Merge red nodes into their black parents.
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth \( h' \) of leaves.

---

**Rotations**

*Right-Rotate (B)*

*Left-Rotate (A)*

Rotations maintain the inorder ordering of keys:
- \( a < b < c \) \( \Rightarrow a < b < c \)
- A rotation can be performed in \( O(1) \) time.

---

**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.

---

**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:**

---
**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.

---

**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).**

---

**Pseudocode**

```plaintext
RB-INSERT(T, x)
TREE-INSERT(T, x)
color[x] ← RED  // only RB property 3 can be violated
while x ≠ root[T] and color[parent[x]] = RED
  do if y = left[parent[x]]
      then y ← right[parent[x]]
          if color[y] = RED
            then (Case 1)
          else if y = 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
```

---

**Graphical notation**

Let $\triangle$ denote a subtree with a black root.

All $\triangle$'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:** $O(\log n)$ with $O(1)$ rotations.

**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

Figure 3: The basic case for insertion of a red-black node.
Bottom-Up Insertions

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

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

Complexity

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

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!
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 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

B-tree (m-way)

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: Root Insertion

- Parent is full, so split it into two

Example: Two-level Insertion

- The root node must be updated

Example: Root Insertion

- Root is full, so split it into two

Example: Root Insertion

- Insert 67
  - Leaf 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.  

Analogy between R-B and B-trees

k-d tree

- 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

- Suppose we wish to partition the following points in a 2-dimensional kd-tree:
**kd-Trees**

- 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, 0.64), (0.04, 0.06), (0.05, 0.97), (0.06, 0.41), (0.06, 0.61), (0.06, 0.28), (0.08, 0.88), (0.08, 0.15), (0.21, 0.26), (0.29, 0.15), (0.13, 0.67), (0.17, 0.91), (0.18, 0.98), (0.19, 0.21), (0.23, 0.15), (0.34, 0.63), (0.36, 0.04), (0.37, 0.04), (0.37, 0.18), (0.32, 0.78), (0.35, 0.42), (0.37, 0.91), (0.74, 0.97), (0.74, 0.04), (0.42, 0.07), (0.37, 0.97), (0.37, 0.14)]

**kd-Trees**

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

\[(0.06, 0.36), (0.04, 0.06), (0.05, 0.07), (0.23, 0.26), (0.06, 0.28), (0.08, 0.41), (0.01, 0.48), (0.01, 0.88), (0.02, 0.88), (0.02, 0.87), (0.05, 0.86), (0.08, 0.86), (0.1, 0.57)]

**kd-Trees**

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

**kd-Trees**

- 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, 0.62), (0.28, 0.02), (0.37, 0.04), (0.33, 0.27), (0.26, 0.27), (0.27, 0.07), (0.37, 0.24), (0.31, 0.42), (0.35, 0.54), (0.46, 0.06), (0.46, 0.87), (0.42, 0.78), (0.54, 0.78), (0.41, 0.86), (0.74, 0.57)]
kd-Trees

- This point creates the right child of the root:

kd-Trees

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

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):

kd-Trees

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

kd-Trees

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

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

• 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

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

• 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 \textsc{NP-hard} [13], and approximate nearest-neighbour methods are used instead.
See also (Wikipedia)

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

Use Array based implementation

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

Priority queue

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

Binary heap - Insert

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

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 – “Bubble up”
Insert

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

• Remove top value (make free space)
• Remove last element
• Insert to top value & bubble down to rightful place

Bubble up

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

Heap-sort

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

Binary heap – Delete – “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 \( \frac{n}{2} \) – ignore
- \( \frac{n}{4} \) - bubble down (at most by 1 level)
- \( \frac{n}{8} \) – bubble down (at most by 2 levels)
- \[ \sum_{i=1}^{\log_2 n} \frac{in}{2^i} \leq \frac{n}{2} \sum_{i=4}^{\infty} \frac{in}{2^i} \]

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 \( x \in S \) 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:

\[ \frac{\text{key}}{\text{size}} \]

Example of an OS-tree

\[ \text{size}[x] = \text{size}[\text{left}[x]] + \text{size}[\text{right}[x]] + 1 \]

Selection

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

- \( \text{OS-SELECT}(x, i) \triangleright i \)th smallest element in the subtree rooted at \( x \)
  - \( k \leftarrow \text{size}[\text{left}[x]] + 1 \triangleright k = \text{rank}(x) \)
  - if \( i = k \) then return \( x \)
  - if \( i < k \) then return \( \text{OS-SELECT}(\text{left}[x], i) \)
  - else return \( \text{OS-SELECT}(\text{right}[x], i - k) \)

(OS-RANK is in the textbook.)
**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.

---

**Example of insertion**

**INSERT(“K”)**

---

**Example**

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

Running time = $O(h) = O(lg n)$ for red-black trees.

---

**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:**

---

**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.

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

---
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 $O(n)$ 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.

### 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 $\text{int}[x]$ corresponding to the key.

### New operations

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

   **INTERVAL-SEARCH($i$)**
   
   $x \leftarrow \text{root}$
   
   while $x \neq \text{NIL}$ and $(\text{low}[x] > \text{high}[\text{int}[x]])$
   or $(\text{low}[\text{int}[x]] > \text{high}[i])$
   do $x \leftarrow \text{left}[x]$
   if $\text{left}[x] \neq \text{NIL}$ and $\text{low}[i] \leq m[\text{left}[x]]$
   then $x \leftarrow \text{left}[x]$
   else $x \leftarrow \text{right}[x]$
   return $x$

### Example interval tree

- $m[x] = \max \{\text{high}[\text{int}[x]], m[\text{left}[x]], m[\text{right}[x]]\}$
- **Example 1:** **INTERVAL-SEARCH(14,16)**

- $x \leftarrow \text{root}$
- [14,16] and [17,19] don’t overlap
- $14 \leq 18 \Rightarrow x \leftarrow \text{left}[x]$
**Example 1:** \textsc{Interval-Search}([14,16])

1. \[ x \leftarrow \text{root} \]
2. \([14,16] \text{ and } [5,11] \text{ don’t overlap} \]
3. \[ 14 \geq 8 \Rightarrow x \leftarrow \text{right}[x] \]
4. \([14,16] \text{ and } [15,18] \text{ overlap} \]
5. \text{return } [15,18]

**Example 2:** \textsc{Interval-Search}([12,14])

1. \[ x \leftarrow \text{root} \]
2. \([12,14] \text{ and } [5,11] \text{ don’t overlap} \]
3. \[ 12 \geq 8 \Rightarrow x \leftarrow \text{right}[x] \]
4. \([12,14] \text{ and } [15,18] \text{ don’t overlap} \]
5. \[ 12 \leq 10 \Rightarrow x \leftarrow \text{left}[x] \]
6. \[ x = \text{NIL} \Rightarrow \text{no interval that overlaps } [12,14] \text{ exists} \]
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 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: \(O(k + \lg n)\).

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 \(\{i' \in R : i' \text{ overlaps } i\} = \emptyset\).

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 \(\text{high}[j]\).

\[
\begin{align*}
\text{high}[j] - m[\text{left}[x]] & \leq \text{low}[i] \\
\text{low}[i] & > m[\text{left}[x]] \\
\text{Therefore, } \{i' \in L : i' \text{ overlaps } i\} & = \emptyset.
\end{align*}
\]
Combining info
• delete max priority...

Combining info
• delete max priority...

Combining info
• delete max priority...

Combining info
• delete max priority...

Union-find
• Domain $X = \{ x_1, \ldots, x_n \}$
• $x_i$ belongs to a set $S_i$
• Non-intersecting sets.

Union of sets: $S_i = S_i \cup S_j$
• Which set $S_i$ does an element $x_i$ belong to?

Combining info
• 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
• 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

• **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`: $O(1)$
- `union`: $O(\alpha(n))$
- `find`: $O(\alpha(n))$

Amortized

Generating mazes – a larger example

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

Fun applications: Generating mazes

<table>
<thead>
<tr>
<th>make(1)</th>
<th>make(2)</th>
<th>make(16)</th>
<th>find(6)=find(7)</th>
<th>union(6,7)</th>
<th>find(7)=find(11)</th>
<th>union(7,11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

More serious applications:

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

Fun applications: Generating mazes

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

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

Path Compression

Union-Find

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

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

Nesting / Repeated application
\[
f^{(i)}(n) = f(f(\ldots(f(n))\ldots))
\]
\[
i \text{ times}
\]
\[
f^{(0)}(n) = n
\]
\[
f^{(i)}(n) = f(f^{(i-1)}(n)) \text{, for } i > 0
\]
\[
f(n) = n + 1 \quad f^{(5)}(n) = n + 5
\]
\[
f(n) = 2n \quad f^{(7)}(n) = 2^7 n
\]
\[
f(n) = 2^n \quad f^{(3)}(n) = 2^{2^n}
\]
\[
f(n) = \log n \quad f^{(2)}(n) = \log \log n
\]

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

Union Find - pseudocode

Function make-set($x$)
  \[
  p[x] \leftarrow x \\
  \text{rank}[x] \leftarrow 0
  \]

Function union($r$, $y$)
  \[
  \text{link}(\text{find}([x]), \text{find}([y]))
  \]

Function link($x$, $y$)
If $\text{rank}[x] > \text{rank}[y]$
  \[
  p[y] \leftarrow x
  \]
Else
  \[
  p[x] \leftarrow y
  \text{if } \text{rank}[y] = \text{rank}[x] \text{ then }
  \text{rank}[y] \leftarrow \text{rank}[y] + 1
  \]

Function find($x$)
If $p[x] \neq x$ then
  \[
  p[x] \leftarrow \text{find}(p[x])
  \]
return $p[x]$
Ackermann’s function (modified)

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

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

\[
\bar{A}_2(n) = 2n \\
\bar{A}_3(n) = 2^n \\
\bar{A}_4(n) = \text{tower}(n) = 2^{2^{\cdots^{2}}}
\]

Inverse functions

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

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

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

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

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

Inverse Ackermann function

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

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

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

\[
A_1(1) = A_{n-1}(1) = A_{n-2}(A_{n-1}(1)) > A_{n-1}(n)
\]

Level and Index

Back to union-find...

\[
\begin{array}{c}
\text{If } p[x] \neq x \text{ and } \text{rank}[x] > 0: \\
\text{level}(x) = \max\{k \geq 1 \mid A_k(\text{rank}[x]) \leq \text{rank}[p[x]]\}, \\
\text{index}(x) = \max\{i \geq 1 \mid A_i(\text{level}(x)) \leq \text{rank}[p[x]]\}.
\end{array}
\]

\[
\text{If } p[x] = x \text{ or } \text{rank}[x] = 0: \\
\text{level}(x) = \text{index}(x) = 0
\]

Potentials

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

\[
\Phi = \sum_x \phi(x)
\]

\[
\text{If } p[x] \neq x \text{ and } \text{rank}[x] > 0: \\
1 \leq \text{level}(x) < \alpha(n), \\
1 \leq \text{index}(x) \leq \text{rank}[x].
\]

\[
\phi(x) \geq 0
\]

On level

If \( p[x] \neq x \) and \( \text{rank}[x] > 0 \):

\[
\text{level}(x) = \max\{k \geq 1 \mid A_k(\text{rank}[x]) \leq \text{rank}[p[x]]\}
\]

If \( p[x] \neq x \) and \( \text{rank}[x] > 0 \):

\[
1 \leq \text{level}(x) < \alpha(n)
\]

\[
1 \leq \text{rank}[x] < \text{rank}[p[x]] < n
\]

\[
A_1(\text{rank}[x]) = \text{rank}[x] + 1 \leq \text{rank}[p[x]]
\]

\[
A_{\alpha(n)}(\text{rank}[x]) \geq A_{\alpha(n)}(1) \geq n > \text{rank}[p[x]]
\]
Bounds on index

If \( p[x] \neq x \) and \( \text{rank}[x] > 0 \):
\[
\text{index}(x) = \max\{ i \geq 1 \mid A_{\text{level}(x)}(i) \leq \text{rank}[p[x]] \}
\]

If \( p[x] \neq x \) and \( \text{rank}[x] > 0 \):
\[
1 \leq \text{index}(x) \leq \text{rank}[x]
\]

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 \( \alpha(n) \)
\[ \Delta \Phi \leq \alpha(n) \]
Actual cost: \( O(\alpha(n)) \)
Amortized cost of find

\[
\phi(x) = (\alpha(n) - \text{level}(x)) \cdot \text{rank}[x] - \text{index}(x)
\]
is either unchanged or is decreased

Amortized cost of find

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