Binary search trees

Binary search trees

- Most common application of binary trees.
- A data structure for representing finite subsets of elements of some type *T*.
- The type *T* is arbitrary but is assumed to have a < operator defined.
- More efficient than vectors or lists for many operations.

Binary search tree

- A finite subset represented by a labeled binary tree.
- The labels of the nodes are the elements in the subset, with nodes having distinct labels.
- For every node, all nodes in the left subtree have labels < label of the node.
- For every node, label of the node < label of any node in the right subtree.

Basic operations

- *The basic operations on a set are*
  - *insert(S,x)* : insert an element x if not present in subset *S.*
  - *find(S,x) :* find if element x belongs to the subset *S.*
  - *erase(S,x) :* remove element *x* from *S* if present*.*
- All three can be implemented in time proportional to the height of the binary search tree representation of the subset.

Insert

- $insert(empty, x) := root(empty, empty, x)$
- $insert(root(T_l, T_r, y), x)) :=$

  $root(insert(T_l, x), T_r, y)$ *if* $x < y$

  $root(T_l, insert(T_r, x), y)$ *if* $y < x$

  $root(T_l, T_r, y)$ *otherwise.*

Find

- $find(empty, x) := false$
- $find(root(T_l, T_r, y), x) :=$

$$
\begin{array}{ll}
find(T_l, x) & \text{if } x < y \\
find(T_r, x) & \text{if } y < x \\
true & \text{otherwise.}
\end{array}
$$

Erase

- $erase(empty, x) := empty$
- $erase(root(T_l , T_r , y), x) :=$

$$root(erase(T_l , x), T_r , y) \qquad \text{if } x < y$$

$$root(T_l , erase(T_r , x), y) \qquad \text{if } y < x$$

$$erase\_root(root(T_l ,T_r ,y)) \qquad \text{otherwise}$$

Erase_root

- $erase\_root(empty) := empty$
- $erase\_root(root(empty, T_r , y)) := T_r$
- $erase\_root(root(T_l , empty, y)) := T_l$
- $erase\_root(root(T_l , T_r , y)) :=$

  $root(T_l , erase\_min(T_r), min(T_r))$

  if both T_l and T_r are not *empty*.

Erase_min

- $erase\_min(empty) := empty$
- $erase\_min(root(empty, T_r , y)) := T_r$
- $erase\_min(root(T_l , T_r , y)) :=$

  $root(erase\_min(T_l) , T_r , y)$

  if T_l is not *empty*.

Min

- $min(empty) := undefined$
- $min(root(empty, T_r, y)) := y$
- $min(root(T_l, T_r, y)) := min(T_l)$

  if T_l is not *empty*.

Time

- Each operation on a tree involves a recursive call on a tree of smaller height.
- Time for all operations is of the order of the height of the tree.
- In the worst case, height can be $\Omega(n)$, where n is the number of nodes.
- Average height is much smaller, many more \`good’ trees than \`bad’.

Average insertion time

- *n* distinct values are inserted in an empty set.
- Random order of insertion, each permutation of the values is equally likely.
- First value could be any one of the *n*, each equally likely.
- This will be the root of the tree and will be compared with all remaining values.
- Values less than it inserted in left subtree and others in the right, their orders are also equally likely.

Average insertion time

- Let T(n) be the average number of comparisons.

$$T(n)=\frac{1}{n}\sum_{i=0}^{n-1}\left(T(i)+T(n-1-i)\right) \quad + \quad (n-1)$$

$$(n+1)T(n+1)-nT(n)=2T(n)+2n$$

$$\frac{T(n+1)}{n+2}=\frac{T(n)}{n+1}+\frac{2n}{(n+1)(n+2)} \qquad T(n)=(n+1)\sum_{i=1}^{n-1}\frac{2i}{(i+1)(i+2)}$$

$$T(n)=O(n\ln(n))$$

Balanced Trees

- Modify the operations to maintain small heights.
- A binary tree with n nodes has height $\Omega(\log n)$.
- Try to ensure it is $O(\log n)$.
- Ensure sufficient number of nodes in both subtrees of any node.
- Each subtree should contain some fixed fraction of total number of nodes.

AVL trees

- The first balanced trees-Adelson Velskii Landis.
- Balance condition – For any node, height of left subtree differs from that of right subtree by at most 1.
- Number of nodes in an AVL tree of height *h* is at least the Fibonacci number $F_h - 1$.
- Height is $O(\log n)$.

Rotation

- Rotation operation to maintain balance.
- $left\_rotate(empty) = empty$
- $left\_rotate(root(empty, T_r, x)) := root(empty, T_r, x)$
- $left\_rotate(root(root(T_{l1}, T_{r1}, y), T_r, x) :=$

  $root(T_{l1}, root(T_{r1}, T_r, x), y).$
- Right rotate defined symmetrically.

Rotation

- A rotation can be implemented in *O(1)* time with pointer manipulations.
- After a normal operation on a binary search tree, rotations used to restore balance property.
- Many kinds of balanced trees.
- STL uses red-black trees.
- Details are not important, important point is height is *O(log n)* in the **worst** case.

C++ Implementation

- STL provides two classes *set* and *map* that use balanced trees (red-black trees).
- Template types with elements of any type.
- $set<T>$ is a set with elements of type T.
- $map<T_1, T_2>$ is a set of ordered pairs, with first element of type T_1 and second of T_2.
- Comparison in maps based only on first element.

C++ Implementation

- set and map use < operator for type *T*.
- < should be a strict weak order - asymmetric, transitive and incomparability is an equivalence relation.
- Possible to define different comparison function.
- *set<T,comp>* - set of type *T* using the comparison object comp.

Sets and Maps

- insert, find and erase operations defined for sets and maps.
- Take $O(\log n)$ worst case time.
- Subscript operator for maps- $M[x] = y$ is the same as replacing any pair (x,z), if present in M, by (x,y).
- Many other useful functions defined.
- Multisets and multimaps also available.