Tree 2

Yu-Tai Ching
Department of Computer Science
National Chiao Tung University

Binary Tree Traversal and Tree Iterators

- “Iterator”: A term (programming style) in C++, you have a container, design an iterator to visit every one in the container.
- Now a tree is a container, there are different ways that systematically visit every node in the tree- tree traversal.
- A traversal produces a linear order.

Binary Tree Traversal

- Let L, V, and R stand for: at a node, we move left (L), visit the node (V), moving right (R),
- then there are 6 possible combination: LVR, LRV, VLR, VRL, RVL, RLV.
- Suppose that we always “moving left” before “moving right”, there are LVR, LRV, and VLR.
- To these, we assign *inorder*, *postorder*, and *preorder*.
- The position of V is “in between”, “after”, and “before”, “moving left” and “moving right”.

Traversal and Expession

- Given the binary tree as in FIgure 5.16
- It contains binary operators and operands
- for an operator, left subtree stands for left operand, right subtree stands for right operand.
- inorder traversal produces infix expression, postorder traversal produces postfix expression.

Inorder Traversal

template <class T>
void $Tree < T >:: Inorder()$
{
    $Inorder(root)$;
}
template <Class T>
void $Tree < T >:: Inorder(TreeNode < T > *currentNode)$
{
    if ($currentNode$) {
        $Inorder(currentNode- > leftChild)$;
        $Visit(currentNode)$;
        $Inorder(CurrentNode- > rightChild)$;
    }
} run though Figure 5.16 $A/B * C * D + E$

Preorder Traversal

template <class T>
void $Tree < T >:: Preorder()$
{
   $Preorder(root)$;
}
template <Class T>
void $Tree < T >:: Preorder(TreeNode < T > *currentNode)$
{
   if ($currentNode$) {
      $Visit(currentNode)$;
      $Preorder(currentNode- > leftChild)$;
      $Preorder(CurrentNode- > rightChild)$;
   }
} run though Figure 5.16 $+ * */ABCDE$

Postorder Traversal

template <class T>
void $Tree < T >:: Postorder()$
{
    $Postorder(root)$;
}
template <Class T>
void $Tree < T >:: Postorder(TreeNode < T > *currentNode)$
{
    if ($currentNode$) {
        $Postorder(currentNode- > leftChild)$;
        $Postorder(CurrentNode- > rightChild)$;
        $Visit(currentNode)$;
    }
} run though Figure 5.16 $AB/C * D * E+$

Iterative Inorder Traversal

template <class T>
void $Tree < T >:: NonrecInorder()$
{
  $Stack < TreeNode < T > * > s$; declare and initialize stack
  $TreeNode < T >$ * $currentNode = root$;
  while(1) {
    while ($currentNode$) {
      $s.Push(currentNode)$;
      $currentNode = currentNode- > leftChild$;
    }

```
    if (s.IsEmpty()) return;
    currentNode = s.Top();
    s.Pop();
    Visit(currentNode);
    currentNode = currentNode-> rightChild;
  }
}
```

- Need a stack
- Push a node into stack then move left until left subtree is traversed,
- then get one from top of the stack,
- traverse right subtree.

- Can we traverse without a stack?
- A possible approach is to add a parent field to each node.
- Another approach is to use thread: Threaded binary tree.

Level-Order Traversal

template <class T>
void $Tree < T >:: LevelOrder()$
{
  $Queue < TreeNode < T > * > q$;
  $TreeNode < T > * currentNode = root$;
  while $(currentNode)$ {
    $Visit(currentNode)$;
    if ($currentNode- > leftChild$) $q.Push(currentNode- > leftChild)$;
    if ($currentNode- > rightChild$) $q.Push(currentNode- > rightChild)$;
    if $(q.IsEmpty())$ return;
    $currentNode = q.Front()$;
    $q.Pop()$
  }
}

The Satisfiability Problem

- A set of formulas constructed by taking variables x_1, x_2, $\ldots$, and operators, $\wedge$ **and**, $\vee$ **or** and $\neg$ **not**.
- Variable holds two possible values, $true$ or $false$.
- The set of formulas using these variables and operators is defined by the rules,
  1. a variable is an expression,
  2. if x and y are expressions then $x \wedge y$, $x \vee y$, and $\neg x$ are expressions.
  3. parentheses can be used to alter the normal order of evaluation, which is **not** before **and** before **or**.
- Propositional calculus.

$$x_1 \vee (x_2 \wedge \neg x_3)$$

is a formula.

- if x_1 and x_3 are $false$ and x_2 is $true$,
- $false \vee (true \wedge \neg false)$
- = $false \vee true$
- = $true$
- “Satisfiability Problem”: for formulas or propositional calculus asks if there is an assignment of values to the variables that causes the value of the expression to be true.

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

- Assume the formula is in the tree Figure 5.18.
- Inorder traversal of the tree gives $x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_3 \vee \neg x_3$,
- infix form of the expression
- to determine the satisfiability, to let (x_1, x_2, x_3) take all possible combinations of $true$ and $false$.
- For n variables, 2^n possible combinations for $true$ and $false$.
- To evaluate the expression, postorder traverse the tree,
- when we visit an operator node, left subtree and right subtree are visited, we can then get the result after the operator applied.

Threaded Binary Trees

- In linked representation of any binary tree, there are more 0-links than actual pointers.
- Suppose there are n nodes ($2n$ links), $n + 1$ 0-links and $n - 1$ actual pointers.
- to replace the 0-links by pointers, called threads, to other nodes in the tree.

- The threads are constructed using the rules,
  1. A 0 $rightChild$ field in node p is replaced by a pointer to the node that would be visited after p when traversing the tree in inorder. It is replaced by the inorder successor of p.
  2. A 0 $leftChild$ link at node p is replaced by a pointer to the node that immediately precedes node p in inorder. Replaced by the inorder predecessor of p.
- Figure 5.20 shows the threaded binary tree of Figure 5.10 b. And FIgure 5.21, a node in the threaded binary tree. To prevent special case of an empty tree, a head node (Figure 5.22).
- Inorder traversal, $x->rightThread == \mathbf{true}$, $x->rightChild$ poitns to the inorder successor of x.

...

Insertion a Node into a Threaded Binary Tree

- Only look at the example of inserting r as the right child of a node s.
- Two cases,
  1. If s has an empty right subtree, then the insertion is simply and shown in Figure 5.23.
  2. If the right child of s is not empty, then the right subtree of s is made the right subtree of r after insertion. Figure 5.23.
- Time complexity?

Heaps

- Priority Queue, a kind of queue that
  - delete the largest (or the smallest)
  - At any time, insert arbitrary priority into the queue.
- Implemented by using an unsorted array.
- Implemented by using a sorted array.
- A heap can do these operations in $O(\log n)$ time.

Definition: A max (min) $tree$ is a tree in which the key value in each node is no smaller (larger) that the key values in its children.

Definition: A $max\ heap$ is a complete binary tree that is also a max tree.

Definition: A $min\ heap$ is a complete binary tree that is also a min tree.

- Some examples are shown in Figure 5.24, 5.25.
- Note that a complete tree can be stored in an array.
- Figure 5.26, insertion, Figure 5.27 deletion both can be done in $O(\log n)$ time.

Binary Search Trees

- A “dictionary” is a collection of pairs, each pair has a key and an associated element. (Assume that all pairs are distinct.)

```
template <class K, class E>
class Dictionary {
public:
   virtual bool IsEmpty() const = 0;
   virtual pair<K,E> *Get(const K&) const =0;
   virtual void Insert(const pair<K,E> &)=0);
   virtual void Delete (const K& )=0;
};
```

Implement a Dictionary

- Unsorted array
- Sorted array
- Linked list,
- Make a linked list to support binary search.

Definition: A “binary search tree” is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:

1. Every element has a key (assuming distinct at present time).
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

Figure 5.28.

Given a binary search tree, and a key k, a search is done by

- suppose we are calling $BinarySearch(p)$, p the node we are at.
- if $p == 0$ return 0;
- if $(k < p- > data)$ return $BinarySearch(p- > leftChild)$,
- if $(k > p- > data)$ return $BinarySearch(p- > rightChild)$,
- if none of the above, this must be the case, we have a succesful search. $p- > data == key$, return $p- > data$,

Time required proportinal to the number of node visited, $O(h)$ h is the depth of the tree.

What if we are looking for the kth largest element in the tree.

- Define the “rank” of a node: its position in inorder.
- Each node should have another additional field, $leftSize$, the number of nodes in the left subtree.
- So the search will be $k == (p- > leftSize + 1)$ we are done,
- Otherwise if $(k \leq p- > leftSize)$, we search on the left subtree, and
- if $(k \geq p- > leftSize)$, we search on the right subtree.
- Again, time depends on the number of node visited, $O(h)$.

Insertion into a Binary Search Tree

- Given a key k to be inserted.
- Search for k in the tree,
- an unsuccessful search reaches an external node,
- insert the node at the external node.
- Cost for the insertion, search $O(h)$, insertion $\Theta(1)$.

Deletion from a Binary Search Tree

- Given a key k to be deleted.
- Search for k in the Binary search tree,
- if an external node is reached, an unsuccessful search
- otherwise we should stop at an internal node p.
- if p has no subtrees or p has one subtree, delete p (similar to that what we have done in processing the linked list).
- if p has two nonempty subtree,

Joining and Splitting Binary Tree

Other than Search, Insertion, and Deletion, there are some other useful operations

- $ThreeWayJoin(small, mid, big)$: there are two binary search tree $small$ and big, and a node $middle$, make a new tree consists of $small$, $middle$, and big.
- $TwoWayJoin(small, big)$, Make $small$ and big a single binary search tree.
- $Split(k, small, mid, big)$, Given a binary search tree T, T will be splitted into $small$, big, and if k is in T, k will be mid.

Three-way Join

- Get a new node, make it data field to be mid.
- Make $small$ to be its left subtree,
- make big to be its right subtree.
- Time required is $\Theta(1)$.
- Height of the tree is $\max(height(small), height(big)) + 1$.

Two-way Join

- Suppose that one of the $small$ and big is empty, the result is the other.
- If both are non-empty, delete the maximum from the $small$ and let it be mid,
- perform a Three-way join.
- time required
  - Delete maximum from the $small$, $O((h(small))$,
  - Three-way Join, $\Theta(1)$

Split at k

- Givne a tree T
- If k is less than the root of T, root and the right subtree of root are in the big.
- We may split at the left sub tree of root, the splitted left subtree is T and root and right subtree is big.
- if k greater that the root of T, root and the left subtree must be in $small$,
- split the right subtree to be T, and the root and the left subtree is $small$.
- it k is less than the root of T, again, root and the right subtree is in big,

- we split at the left subtree, left subtree is T, append the splitted root and right subtree to the left subtree of the big,
- Note that we can do this without violiting the binary search tree definition
- repeat this process, each time a comparison and than split and append.
- Compare need $O(h)$ time, each append takes $\Theta(1)$ time, total need $O(h)$ time.

Height of a Binary Search Tree with n nodes

- Height determines the time required for searching, insertion, deletion, joint, and splitting.
- In the worst case, a skew binary search tree, height is $O(n)$.
- The height is at leat $O(\log n)$.
- Some trees are designed so that the height is always $(\log n)$, such as AVL-tree, 2-3 Tree, we call that height balance tree.

Selection Trees

Problem definition:

- k ordered sequences, called “runs”
- merged into a sequence
- Each runs consists of records, each record has a “key”
- To merge, each time output the record with the smallest key,
- brute force approach, choose the smallest from k runs, need $k-1$ comparison.
- selection tree can do this in $\log k$ comparisons.
- winner trees and loser trees.

Winner Trees

- A complete binary tree,
- each node represents the smallest the smallest node of its children.
- The root has the smallest. Figure 5.31
- output the smallest can be found at the root,
- suppose the smallest is the head of ith run, take the smallest and put it into the selection tree, and update the tree.
- height of the selection tree, $\Theta(\log k)$.
- if there are n records in k runs, merging can be done in $\Theta(n \log k)$.

Loser Tree

- Do the same thing
- make operations a little bit simpler.
- Each node has a pointer pointing to the head of the loser,
- There a node on the top of the tree which is the overall winner. Figure 5.33
- Remove (output) the overall winner, take the second largest in the run into the selection tree,
- it is simple because the previous loser was there, we can make comparison directly.

Forest

Definition: A forest is a set of $n \geq 0$ disjoint trees.

- Figure 5.34, a 3 trees forest.
- transform a forest into a binary tree:
  - each tree is transformed into a binary tree
  - these tree are linked throug the $rightChild$ field.
  - Figure 5.34, the binary tree of previous 3 trees forest.

Representation of Disjoint Sets

- Use of trees in the representation of sets.
- Assume that the sets are numbers 0, 1, 2, . . . , $n - 1$. (They can be actually pointers to the name of the sets.)
- Assume that the sets are pairwise disjoint, i.e., S_i and S_j, if $i \neq j$, then $S_i \cap S_j$ is $\emptyset$.
- An example, in “Equivalence classes”, if the relation is “in the same set”, there are 3 sets.
- Another example, if $n = 10$ (0, 1, . . . , 9), the elements can be partitioned into three disjoin sets, $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, and $S_3 = \{2, 3, 5\}$.
- Figure 5.36 shows a possible representation, the forest representation. Root of the tree, the representative of the set.

The Operations

- $Disjoint\ Set\ Union$. If S_i and S_j are two disjoint sets, then the union $S_i \cup S_j$ = $\{0, 6, 7, 8, 1, 4, 9\}$. The two old sets are replaced by the new set which is the union of the previous two.
- $Find(i)$. Find the set containing element i. $Find(3)$ returns S_3.
- We call this the $Union$ and $Find$ operations.
- $Union(S_i, S_j)$, make one of the roots has the parent pointer points the the root of the other tree. (Figure 5.37).
- $Find(k)$, trace the parent links to the root of the tree.
- Implemented using an array (Figure 5.39).

Performance of $Union$ and $Find$

- A single $Union$, it takes $\Theta(1)$ time.
- A single $Find(i)$, depends on the path length from i to the root.
- Suppose that $Union(i, j)$ makes root of i points to root of j,
- And suppose we have the sequence of operations $Union(0, 1)$, $Union(1, 2)$, $Union(2, 3)$, $Union(3, 4)$, $\ldots$, $Union(n-2, n-1)$ followed by $Find(0)$, $Find(1)$, $\ldots$, $Find(n-1)$.
- The total cost will be $\sum_{i=1}^{n} i$ = $O(n^2)$.
- Can this be improved? Try to prevent the “bad” union.

Definition *Weighting rule for* $Union(i,j)$: If the number of nodes in the tree rooted i is less than the number of nodes in the tree rooted j, then make j the parent of i, otherwise make i the parent of j.

- To implement the weighting rule, we need a $count$ field other than the $parent$ field.
- if $count[i] == count[j]$, tie broken arbitrarily.
- It still take $\Theta(1)$ time for a single $Union$,
- but with the weighting rule, we prevent the bad case,
- but how much can we improve?

Lemma 5.5: Assume that we start with a forest of trees, each having one node. Let T be a tree with m nodes created as a result of a sequence of unions each performed using the weighting rule. The height of T is no greater than $\lfloor \log_2 m + 1 \rfloor$.

Proof: By induction on the number of nodes, m, in T. Clear true for $m = 1$.
Assume it is true for all trees with $i \leq m - 1$. We show that it is also true for $i = m$.
Let T be a tree with m nodes. And T was created by applying the weighting rule $Union(k, j)$. And we let the two trees be denoted T_k and T_j. Let a be the number of nodes in T_j (thus tree T_k has $m - a$ nodes).

We may assume without loss of geneality that $1 \leq a \leq m/2$. Thus root of T_k is the root of the tree after $Union$. Let $|T|$ denote the height of T. There are two cases,

1. $|T_k| > |T_j|$, the resulted tree T has the same height as T_k. Height of T_k is $\leq \lfloor \log_2 (m-a) \rfloor + 1 \leq \lfloor \log_2 m \rfloor + 1$.
2. $|T_k| < |T_j|$, height of T is $|T_j| + 1$, i.e., $|T|$ is $\leq \lfloor \log_2 a \rfloor + 1 + 1 \leq \lfloor \log_2 m/2 \rfloor + 2 \leq \lfloor \log_2 m \rfloor + 1$.

In both cases, we have $|T| \leq \lfloor \log_2 m \rfloor + 1$.

- Note that this bound is tight (it is achievable).
- Each single $Find$ is bounded by $O(\log n)$.
- Can be further improved.

Definition [Collapsing Rule]: If j is a node on the path from i to its root and $parent[i] \neq root(i)$, the set $parent[j]$ to $root(i)$.

- Apply collapsing rule in $Find(i)$.
- There are two passes.
- in the first pass, $Find(i)$ traces the parent links to the root.
- in the 2nd pass, change the $parent[j]$, j are nodes on the path from i to the root.
- Pay some more efforts this time, but the $Find()$ operations in the future could save time.

Effects of the Weighting Rule and Collapsing Rule

- Each $Union$ still takes $\Theta(1)$ time. And there are at most $n-1$ $Union$s.
- Each $Find$ takes no more than $O(\log n)$ time by the wieghting rule.
- When collapsing rule is applied, each $Find$ takes $O(\alpha(p,q))$ time.
- $\alpha(p,q)$ is a function which is the inverse of the the Ackermann's function $A(i,j)$.
- Ackermann's function is a function grows very fast.
- $\alpha(p,q)$ monotonically increasing as p or q increase, but very very slowly increases.

- A function $n = 2^k$, its inverse $\log_2 n$ = k. As n increases, k increases. n double but k just increment by 1.

- Another function $n = 2^{2^{2^{\cdots^{2}}}}$, $k = \log^* n$ is the height of the tower. $\log^* n$ inverse of the the function $n = 2^{2^{2^{\cdots^{2}}}}$. Again, k increases as n increases. But n increases a lot so that k increment by 1.

- Ackermann's function grows much faster than $n = 2^{2^{2^{\cdots^{2}}}}$, its inverse is $\alpha(p, q)$ which gorws even lower than $\log^* n$.

Ackermann's Function

$$
\begin{aligned}
A(1,j) &= 2^j, \text{ for } j \geq 1, &(1)\\
A(i,1) &= A(i-1,2) \text{ for } j \geq 2, &(2)\\
A(i,j) &= A(i-1, A(i,j-1)) \text{ for } i,j \geq 2 &(3)
\end{aligned}
$$

Inverse of Ackermann's Function

$$\alpha(p,q) = \min\{z \geq 1 | A(z, \lfloor p/q \rfloor) > \log_2 q\}, p \geq q \geq 1.$$

Discussions on $A(i,j)$ and $\alpha(p,q)$

- $A(i,j)$ very rapidly growing function, thus α is a very slowly growing function.
- $A(3,1) = 16$, $\alpha(p,q) \leq 3$ for $q < 2^{16} = 65536$ and $p > q$.
- Since $A(4,1)$ is a very large number,
- and in Union/Find application, q will be the number, n, of set elements and p will be $n + f$ (f number of finds),
- $\alpha(p,q) \leq 4$ for all practical purposes.

Lemma 5.6 [Tarjan and Van Leeuwen]: Assume that we start with a forest of trees, each having one node. Let $T(f,u)$ be the maximum time required to process many intermixed sequence of f finds and u unions. Assume that $u \geq n/2$. Then

$$k_1(n + f\alpha(f+n,n)) \leq T(,u) \leq k_2(n + f\alpha(f+n,n))$$

for some positive constants k_1 and k_2.

- An application, solve the Equivalence relationship problem.