CS430

LECTURE NOTE NO.6

Data Structure for Efficiency

In this lecture we will describe the design of data structures for various set operations. Given a set of items, we need to design data structures for the classical heap operations. While heaps can be maintained via balanced binary tree structures, we will describe two structures, binomial heaps and fibonacci heaps which have particularly interesting properties which are useful in the design of algorithms.

1 Union-Find

In this problem we have a collection of sets on which the folowing operations are to be performed

- $UNION(S_1, S_2)$: Constructs the union of two sets.
- $FIND(x)$: Finds the set containing an item x.

The Union-Find data structure represents each set, S, as a rooted tree, $T(S)$.

And a collection of sets as a forest. Initially, each set is assumed to be singleton, represented by a single root node.

A union operation is implemented by a merge of the trees. We consider the height of the tree to implement the union operation. let $h(T)$ be the height of tree T. Consider the two sets S_1 and S_2 which are to be merged. $T(S_1)$ is made the child of the root of $T(S_2)$ if $h(T(S_1)) \leq h(T(S_2))$ else $T(S_2)$ is made the child of the root of $T(S_1)$.

Note that the FIND operation requires $O(h(T)$ where T is the tree containing the element whose set is to be found.

With the above UNION rule we can show the folllowing:

Lemma 1.1. *The height of any tree in the forest is at most* $O(\log m)$

The proof is by induction on the height of the tree. It is required to show that $N(T) \geq 2^{h(T)}$.

In order to analyze the data structure with path compression, consider resequencing the operations so that all the UNIONs occur first followed by the FINDs. The result of the UNIONs are tree(s) of height at most $O(\log m)$, when there are m union operations. However the FINDs need to be accurately representing the original FIND and thus they are partial FINDs in that the compression goes up to the node which was the root of the set when the FIND operation took place.

We define a charging operation for each link which is changed during compression. Note that the last link can be charged to the FIND operation. Consider the tree T containing the node u at which the FIND takes place. Define $rank(u)$ to be the height of the subtree with u as the root.

Note that the number of nodes of rank k, $N(r)$ satisfy

$$N(r = k) \leq n/2^k$$

When each such node is the origin of a FIND operation, links which are changed can be charged to this node. However not all links can be charged. We only charge links as long as the compression raises the rank of this node within a band of ranks.

We define $Band(0) = \{u | 0 \leq rank(u) < 2\}$. And in general $Band(i) = \{u | B(i-1) \leq rank(u) < B(i)\}$ where $B(i) = 2^{B(i-1)}$ with $B(0) = 0$.

Note that the number of bands is bounded by $\log^* n$.

To bound the charging, we first consider the charges generated for the nodes in a band.

The charges on nodes in $Band(i)$ are as follows:

$$\Sigma_{i=B(i)}^{i=B(i+1)}(n/2^i)(B(i+1)-B(i))$$

since a node may be charged on $B(i+1)-B(i)$ times. Its rank increases by at least 1 with each link change.

The above is bounded by $2(B(i+1)-B(i))n/2^{B(i)} \leq 2n$.

Since the number of bands is $O(\log^* n)$ the total charges generated within the bands is $O(n \log^* n)$. Links which cross bands are charged once, when these links are changed. Thus the total number of charges is $(m \log^* n)$ since there can be at most $O(\log^* n)$ rank changes within the bands when there are m finds.

Lemma 1.2. *The time required for m unions and n finds, where $n \leq m$ is $O(m \log^*(n))$ operations*

2 Binomial Heaps

BINOMIAL HEAPS: A data structure over a set of values that allows for insertions, deletions and extracting the minimum

A binomial heap is a forest of trees. We will let $Parent(u)$ denote the parent of node u. Each node u is associated with a set item with the value of the item denoted as $key(u)$. The trees are heap-ordered, i.e. $key(u) \geq key(parent(u)$. We define the rank of a tree in the forest as the number of children of the root of the tree. The trees will be rank ordered.

Structure of Binomial Forests

The Binomial Forest comprises trees, with the following rule:

One Rank Rule: The forest has at most one tree of a given rank.

A tree of rank k, T_k is composed of two trees of rank $k-1$. Let A and B be two trees of rank $k-1$. A tree of rank k is created by comparing the keys at the roots $root(A)$ and $root(B)$ of A and B, respectively. Supopose $key(root(A) \leq key(root(B))$. Then the root of B is made a child of $root(A)$. The reverse operation is performed when $key(root(A) > key(root(B))$. This is called a *merge* operation.

A tree of rank k thus has as children of the root node subtrees of rank $0, 1 \ldots k-1$.

Operations

We analyse insertions and deletions:

Insertions: A key is inserted into a binomial heap as a tree of rank 0. This may violate the *One Rank Rule*. In which case a merge operation is perfomed. This merge operation is repeated while the *One Rank rule* invariant is violated.

Deletions: To delete a key at the root node, the root node is deleted and the subtrees become trees in the forest. A merge operation is performed repeatedly while there is any violation of the *One Rank Rule*. To delete an arbitrary node, u, let P_u be the path from the node u to the root of the tree. The node to be deleted is given a key value smaller than the minimum and promoted to the root along the path P_u. The root is then deleted.

Time Complexity

To bound the time complexity of a seqence of n operations we first show a bound on the height of binomial tree.

From the structure of the binomial tree, it should be obvious to the reader that

Lemma 2.1. *The height of a tree of rank of k is at most k.*

Further,

Lemma 2.2. *A tree of rank k has at most 2^k nodes*

This is easy to show, given that a tree of rank k is composed of two trees of rank $k-1$.

Consuequently there are at most $O(\log n)$ trees in a forest storing n key values and the maximum rank is $O(\log n)$.

We next introduce an amortized bound on the complexity of each of the heap operations. We need the following potential function:

$PF_i = -$ number of trees in the forest at the ith operation.

We define Amortized Work to be Actual work - change in potential function, i.e. $AM_i = Actual_i - \Delta PF_i$ where $\Delta PF_i = PF_i - PF_{i-1}$

Note that consequently for n operations,

$$\sum_i AM_i = \sum_i Actual_i - \sum \Delta PF_i$$

or by telescoping

$$\sum_i AM_i = \sum_i Actual_i - (PF_n - PF_0)$$

or $\sum_i Actual_i = \sum_i AM_i + (PF_n - PF_0)$

Note that PF_n and PF_0 are $O(\log n)$ since there are at most $O(\log n)$ trees when the heap contains n nodes.

So the goal is to bound the amortized complexity of each operation.

To bound the amortized complexity of an insertion first that the potential function changes. One tree is added into the forest and a number of merges, say m, may be performed. Thus ΔPF is $m-1$. Since the actual work is $1+m$ where m is the number of merges the amortized complexity of an insertion at the ith step

$$AM_i(Insertion) = 2$$

For the deletion operation, the actual work performed is $1+s+m$ where s is the number of subtrees deleted and added to the forest and m is the number of merges performed. The change in potential is $m-s$ since the trees grow in number by s and reduce in number by m, the number of merges. Thus

$$AM_i(deletion) = 1 + 2s$$

Since the number of subtrees affected is $O(\log n)$

$$AM_i(deletion) = O(\log n)$$

The complexity of extracting the minimum can be expressed in terms of determining the minimum and deleting it. The minimum can be determined from the keys at the root of the trees in the forest $O(\log n)$ steps since there are at most $O \log n)$ trees in the forest. The complexity of deleting a node has already been bounded.

3 Fibonacci Heaps

We next consider another important operation on heaps. Suppose we decrease the value of a key. How fast can this operation be implemented? Note that the obvious way of deleting and re-inserting this item would simply be $O(\log n)$ in binomial heaps. The reader is invited to try various strategies while keeping intact the structure of binomial heaps.

In order to achieve an $O(1)$ complexity of decreasing the minimum, consider simply deleting the node from the tree and hanging the subtree rooted at the node v as a tree in the forest. A potential complication arises in that this destroys the structure of the binomial forest and may threaten to create unbalanced or lopsided trees. To ensure that a balanced structure is maintained, the rule employed is to not delete more than one child of a node. An analysis shows that the lopsided nature of the trees is not too serious.

We first briefly describe the operations:

The implementation utilizes the notion of marking a node as dirty when it looses a chiled. A node cannot be double marked. The following invariants are maintained.

Invariant 1: There is exactly one tree of a given rank in the forest.
Invariant 2: Any node is marked dirty at-most once.

Insertions: This is exactly the same as in the case of binomial heaps. A node is added into the heap as a tree of rank 0.

Deletions: In this case the node is deleted from its parent. The node is removed and all its subtrees are hung in the forest as trees. Further the parent of the node is marked dirty. If the parent has already been marked dirty, cascading cuts are performed.

Cascading Cuts: This operation deletes a doubly dirty node and hangs it into the forest. The parent is marked dirty and the cascading cut is then recursively repeatedly on the parent.

Decrease Key: The key value at a node v in a tree is decreased. If this value is lower than that at its parent the node is deleted and the subtree rooted at the node v is hung into the forest. The parent is marked dirty and the cascading cut operation is performed if required as above.

Complexity

The analysis uses the following potential function:

$PF = -$ No. of tree -2 No. of marked nodes.

We now analyze each of the operations, except insertion whose analysis is similar to the one above.

Deletion: Let u be the node deleted with number of children $r(u)$.
$Actual - work = 1 + r(u) + \#cc + \#m$
where $\#cc$ is the number of nodes deleted (all these have two children removed and are doubly dirty) and $\#m$ is the number of merge operations.
$\Delta PF = -(r(u) + \#cc - \#m) - (-2\#cc)$
Thus the amortized work is $Am = 1 + 2r(u)$

Decrease key: The analysis is similar except that the children of the node whis key is decreased are not considered individually. This is crucial because this saves the term$r(u)$ resulting in an amortized complexity of $O(1)$.

We thus simply need to bound the maximum number of children any node can have in the heap structure.

This is done by considering the structure of the tree in the forests. Every node has at most one child deleted. So the minimum number of nodes N_r in a subtree of rank r is defined by the recurrence: $N_r = N_{r-2} + N_{r-3} \ldots N_0$ with $N_0 = 1$.

Solving the recurrence gives $N_r = \phi^r$ where $\phi = (1 + \sqrt(5))/2$, the golden ratio.

Thus the rank of every node is bounded by $O(\log n)$.