Selection Problem · Week III

Data Structures: Theory and Practice

Sunday 29th November, 2009

Johan Sejr Brinch Nielsen

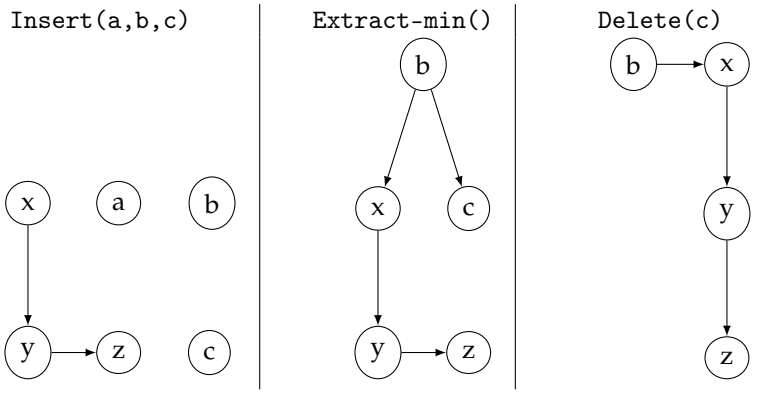
Email: zerrez@diku.dk
Cpr.: 260886-2547

Dept. of Computer Science,
University of Copenhagen

1) Exercise 19.4-1

For any natural number n it is possible to generate a fibonacci heap of size n, for which its internal tree structure is linear.

- Base case $n = 0$: Insert an element. The heap is now of size 1 and linear.
- Induction step $n \to n + 1$: Suppose we have a linear fibonacci heap of size n with x as minimum. Insert three nodes a, b and c where $a < b < c$ and all three are lower than the minimum of the heap (a is the new minimum). Call extract-min to extract a. The `Consolidate` operation will now make a tre with b as root and x and c as children. Delete c. We now have a linear fibonacci heap of size $n + 1$.



2) Problem 19-4

A mergeable heap backed by a 2-3-4-tree. The operations could be implemented by:

- `Minimum`: Maintain a pointer to the smallest node to achieve $O(1)$.
- `Decrease-key`: Decrease the key of the leaf and update all the affected `small` values of its parent etc. Update the minimum-pointer if needed. This operation is $O(\log n)$.
- `Insert`: First, find the leaf where the node is to be inserted by choosing the appropriate child (largest `small` value lower than the value inserted). Now, create a new leaf for the value and add it to the parent of the found leaf. If the parent is full do splitting as in 2-3-4-trees. Update the small values and the minimum-pointer if the inserted value is a new minimum. Time: $O(\log n)$
- `Delete`: This is done as in an 2-3-4-tree. First delete the node. If its parent becomes empty perform the appropriate merging. Update the appropriate `small` values. If the deleted key was the minimum, scan the tree for the new minimum. This operation is $O(\log n)$ time.

- `Extract-min`: Call `Delete` on the minimum (given by the pointer). This is $O(\log n)$.
- `Union`: How to make a new heap H from two existing heaps H_0 and H_1. If the two H_0 and H_1 have equal heights, make a new root that these as children. Set its `small` value to the minimum of the two heaps. This case takes $O(1)$.

  If H_0 is tallest, find some node in H_0 whose subtrees are the same height as H_1 and add H_1 as a child.

  Finding H_1's new parent in H_0 is $O(\log(\max(n, m))) = O(\log(n + m))$.

Stubbed C++ code

File: heap.h++

```
using namespace std;

template <class T>
class HeapNode {
public:
        // only internal nodes:
        T small;
        HeapNode<T> *children[3];

        // only leafs:
        int key_count;
        T keys[3];
};



template <class T>
class Heap {
public:
        HeapNode<T>* search(T key);
        void insert(T key);
        void delete_key(HeapNode<T> *node);
        HeapNode<T>* minimum();
        void decrease_key(HeapNode<T> *node, T new_key);
        HeapNode<T>* extract_min();
        Heap<T>* merge(Heap<T> *heap);
        void print(HeapNode<T> *node);
private:
        HeapNode<T>* root;
        HeadNode<T>* minimum;
};
```

File: heap.c++

```
#include "heap.h++"
#include<stdlib.h>
```

```
#include<stdio.h>

using namespace std;

template <class T>
void Heap<T>::insert(T key) {}

template <class T>
HeapNode<T>* Heap<T>::search(T key) {}

template <class T>
void delete_key(HeapNode<T> *node) {}

template <class T>
HeapNode<T>* minimum() {}

template <class T>
HeapNode<T>* extract_min() {}

template <class T>
Heap<T>* merge(Heap<T> *heap) {}

template <class T>
void print(HeapNode<T> *node) {}
```

3) Binomial Heaps

I have investigated some running times in binomial heaps:

- n inserts: $O(n)$
- m deletes: $O(m \log m)$
- n inserts then m deletes: $O(n + m \log m)$

I convinced that the running time of a mixed sequence of inserts and deletes are worst case $O(N \log N)$ where N is the total number of operations.

4) Pennants

A pennant is a heap ordered binary tree with a size of 2^r where r is the rank. The root node has no left child and its right subtree is a perfect binary tree. The minimum value of the tree is stored in the root node. Consider now an operation that merges to trees of same rank:

```
Merge(s, t)
```

Two pennants can be merged by:

1. Initialize a new pennant p.
2. If s is the tree with the smallest root and t the other tree then extract the root of s and let it be the new root of p.

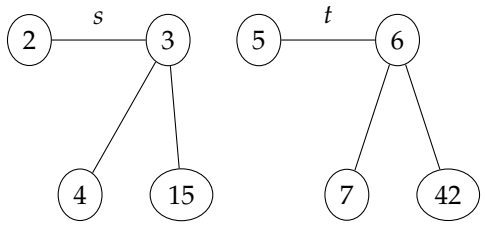


Figure 0.1: Example of two pennants s and t

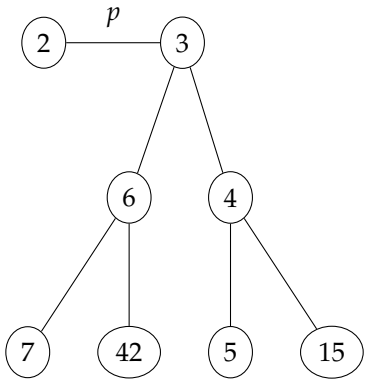


Figure 0.2: The two pennants s and t merged

3. If now Root(t) $<$ Root(s) then set Right(Root(p)) = t and Left(t) = s. p is now the merged tree. This case requires $O(1)$ time.

   If instead Root(t) $>$ Root(s) then swap the two and set Right(Root(p)) = t. Set Left(t) = s. The problem now is that Left(t) may now be a heap-ordered tree. It's root could be larger than its children. We therefore need to push the down until it reaches a smaller node or the end. This case requires $O(r)$ time (This is the case illustrated in the example).

The worst case running time of the `Merge` operation is thereby $O(r)$ where r is the rank of the two trees (each of size 2^r).

The worst case situation when performing a `Union` operation on two pennant queues of size S is when both queues has a pennant for each rank r. In this case S `Merge` operations are performed. The worst case running time of this operation is thereby:

$$O\Big(\sum_{r=0}^{S} r\Big) = O(\frac{S^2+S}{2}) = O(S^2) = O(\log^2(N))$$

Where N is the number of elements in the united queues.

5) Fibonacci Heaps

A normal fibonacci heap has the following worst case running times for the priority queue operations:

- `Insert`: $O(1)$
- `Find-min`: $O(1)$
- `Extract-min`: $O(n)$

The worst case of `Extract-min` is bounded by the maximum of children the minimum-node has. If we give an upper bound of $O(\log n)$ on this number the operation would run in worst case $O(\log n)$. My idea is to move some of the work from the `Extract-min` onto the `Insert` operation which currently runs in $O(1)$. It goes as follows:

- Set an upper bound of $2 \log n$ on the children any node can have.
- Keep track of the count of a node's children when inserting and extracting using a counter.
- If a node has $2 \log n$ children when inserting a new child, cut the $\log n$ of them and make them roots. Run consolidate.

If one could actually maintain an invariant that guaranteed that no node has more than $O(\log n)$ children, then `Extract-min` would have $O(\log n)$ worst case.