Report for Assignment 9

Deep Kiran Shroti
Roll Number- 12EC35013

March 27, 2016

Fibonacci Heaps

Assignment Statement
We consider a practical generalization, where we have k lists of blocks of m items so that the items (integers) are sorted within and across the blocks in each list. Note that with a higher value of k there would have been more lists. There will be up to kmkm elements in the heap. It is necessary to merge the integers of these lists so that the resulting is sorted and also grouped in blocks of m items so that the items (integers) are sorted within and across the blocks. At a time all integers are to be transferred out a block in the input list. This generalization is considered to model reading and writing to disks where i/o is done in block of some given size, say m. The merging is to be done using Fibonacci heaps and associated operations. Integers from a block should be used to first create a Fibonacci heap and then that should be merged into the main Fibonacci heap used for creating the output list. After integers have been read from a block and introduced to the heap, the last (and largest) integer read from that block should be reduced to one less than the minimum element in the heap if that is larger than the last value written to the merged list, if any. This is an artificial thing to do. It has been introduced so that the DecKey operation of heaps is also applied.

1 Introduction

1.1 Fibonacci Heaps

The Fibonacci heap data structure serves a dual purpose.
First, it supports a set of operations that constitutes what is known as a "merge-able heap."
Second, several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently.

Mergeable heaps A mergeable heap is any data structure that supports the following five operations, in which each element has a key:

- MAKE-HEAP() creates and returns a new heap containing no elements.
- INSERT(H,x) inserts element x, whose key has already been filled in, into heap H .
- MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.
- EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.
- UNION(H1,H2) creates and returns a new heap that contains all the elements of heaps H 1 and H 2 . Heaps H 1 and H 2 are "destroyed" by this operation.

In addition to the mergeable-heap operations above, Fibonacci heaps also support the following two operations:

- DECREASE-KEY(H,x,k) assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.
- DELETE(H,x) deletes element x from heap H.

2 Implementation Details

FibHeapInsert()

```
FIB-HEAP-INSERT(H,x)
  x.degree = 0
  x.p = NIL
  x.child = NIL
  x.mark = FALSE
   if H.min == NIL
    create a root list for H containing just x
     H.min = x
  else insert x into H root list
    if x.key < H.min.key
      H.min = x
  H.n = H.n + 1
```

FibHeapUnion()

```
FIB-HEAP-UNION(H1,H2)
  H = M AKE-FIB-HEAP()
  H.min = H1.min
  concatenate the root list of H 2 with the root list of H
  if (H1.min == NIL)or (H2.min != NIL and H2.min.key <
     H1.min.key)
    H.min = H2.min
  H.n = H1.n + H2.n
return H
```

FibHeapMin()

```
FIB-HEAP-EXTRACT-MIN(H)
  z = H.min
  if != NIL
    for each child x of
      add x to the root list of H
        x.p = NIL
    remove z from the root list of H
    if z == z.right
      H.min = NIL
    else H.min = z.right
      CONSOLIDATE (H)
    H.n = H.n-1
return z

CONSOLIDATE(H)
  let A[0 ... D(H.n)]be a new array
  for i = 0 to D(H.n)
    A[i] = NIL
  for each node w in the root list of H
     x = w
     d = x.degree
     while A[d] != NIL
        y = A[d] //another node with same degree
        if x.key > y.key
          exchange x with y
        FIB-HEAP-LINK (H,y,x)
        A[d] = NIL
        d = d + 1
    A[d] = x
  H.min = NIL
  for i = 0 to D(H.n)
    if A[i] != NIL
      if H.min == NIL
        create a root list for H containing just A[i]
        H.min = A[i]
```

```
          else insert A[i] into the H root list
            if A[i].key < H.min.key
              H.min = A[i]

FIB-HEAP-LINK(H,y,x)
  remove y from the root list of H
  make y a child of x, incrementing x.degree
  y.mark = FALSE
```

fibHeapDecKey()

```
FIB-HEAP-DECREASE-KEY(H,x,k)
 if k > x.key
  error (new key is greater than current key)
 x.key = k
 y = x.p
 if y != NIL and x.key < y.key
  CUT(H,x,y)
  CASCADING-CUT(H,y)
 if x.key < H.min.key
  H.min = x

CUT(H,x,y)
  remove x from the child list of y, decrementing y.degree
  add x to the root list of H
  x.p = NIL
  x.mark = FALSE

CASCADING-CUT(H,y)
  z = y.p
  if z != NIL
      if y.mark == FALSE
      y.mark = TRUE
  else CUT(H,y,z)
      CASCADING-CUT(H,z)
```

fibHeapDelete()

```
FIB-HEAP-DELETE(H,x)
  FIB-HEAP-DECREASE-KEY(H,x,INT_MIN)
  FIB-HEAP-EXTRACT-MIN(H)
```

3 Complexity Analysis:

Potential function
As mentioned, we shall use the potential method to analyze the performance of Fibonacci heap operations. For a given Fibonacci heap H , we indicate by t(H) the number of trees in the root list of H and by m(H) the number of marked nodes in H. We then define the potential φ(H) of Fibonacci heap H by

$$\varphi(H) = t(H) + 2m(H)$$

Creating a new Fibonacci heap
To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H, where H.n = 0 and H.min = NIL; there are no trees in H. Because t(H) = 0 and m(H) = 0, the potential of the empty Fibonacci heap is φ(H) = 0. The amortized cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost.

Inserting a node
To determine the amortized cost of FIB-HEAP-INSERT, let H be the input Fibonacci heap and H 0 be the resulting Fibonacci heap. Then, $t(H') = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is
$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1$
Since the actual cost is $O(1)$, the amortized cost is $O(1) + 1 = O(1)$.

Finding the minimum node
The minimum node of a Fibonacci heap H is given by the pointer H.min, so we can find the minimum node in $O(1)$ actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its O(1) actual cost.

Uniting two Fibonacci heaps
The change in potential is
$\varphi(H) - (\varphi(H_1) + \varphi(H_2))$
$= ((t(H) + 2m(H)) - (t(H_1) + 2m(H_1)) + ((t(H_2) + 2m(H_2)))$
$= 0$
because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortized cost of FIB-HEAP-UNION is therefore equal to its $O(1)$ actual cost.

Extracting the minimum node
The potential before extracting the minimum node is $t(H) + 2m(H)$, and the potential afterward is at most $(D(n) + 1) + 2m(H)$, since at most $D(n) + 1$ roots remain and no nodes become marked during the operation. The amortized cost is thus at most $O(D(n)) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H))$
$= O(D(n)) + O(t(H)) - t(H)$
$= O(D(n))$

Since we can scale up the units of potential to dominate the constant hidden in $O(t(H))$. Intuitively, the cost of performing each link is paid for by the reduction in potential due to the link's reducing the number of roots by one. We shall see in that $D(n) = O(log(n))$, so that the amortized cost of extracting the minimum node is $O(log(n))$.