Homework 4

Matt Forbes

November 16, 2010

Problem 1

a) Using a linked list is probably the most straight-forward way to simulate eeny meeny miny mo. To build the list, we would start with the first kid as the head of the list. Then we append each kid to the list in a row, going around the circle to the right.

Once, we have the list built, simulating the game would just be moving forward in the list for each word in the song, and removing the element we hit when the song is finished. So, starting with the head of the list, visit the next node w times (wrapping around to the head when we hit the end of the list). After w visits, we remove the current element we visiting from the list.

We do this while the number of elements in the list is greater than 1. When there is only one kid left in the list, he is 'it.'

This way, we are doing w visits for each iteration of the song (definition of the game). And we have to remove p-1 kids from the list (p-1 iterations), so in total, we are visiting about pw nodes in the list.

So in total, we did p insertions to the list, and pw visits on the list. Therefore, the running time of the algorithm is $O(p + pw)$ or just $O(pw)$.

b) **Assumptions**

This code assumes that we have the augmented data structure OS_Tree that we described and used in class, with methods OS_Insert(T, obj) and OS_Select(T, r) which selects the node within tree T with rank r. This OS_Tree holds kid datatypes that hold a pointer to the kid and also their original index in the circle, labeled kid and index respectively. The tree sorts on index.

Pseduo Code

```
(* emmm_select performs eeny-meeny-miny-mo selection on the
array of p kids (kid_arr), using a song with w words *)

def emmm_select(kid_arr, p, w):
    OS_Tree T = new OS_Tree

    (* insert kids in original order *)
    for i in 1...p:
        OS_Insert(T, { kid_arr[i], i } )

    (* simulate p-1 iterations of the song *)
    for i in 1...(p-1):
        rank = w % (p-i+1) (* rank of kid to be removed *)
        kid = OS_Select(T, rank)
        remove(T, kid)

    (* return last kid left in T *)
    return OS_Select(T, 1).kid
```

Justification

This algorithm works by first creating an order-statistic balanced binary tree (the one from class), and filling it with the kids playing the game. The ranks of the kids are determined by their distance from the current song-singer (whose rank is 1). So the singer's rank is 1, the kid to his right is rank 2, all the way around the circle. The kid who is going to be removed from the circle is the one who is w kids to the right of the singer (with wrapping). This rank is just w % number of kids left in the circle. Determing this rank and retrieving the kid from the tree is $O(\log_2(p))$ using OS_Select. We need to remove $p - 1$ kids, so the algorithm's running time is $O(p \log_2(p))$.

Problem 2

Insert Pseudocode

```
(* assumed definitions: *)
(* add_to_end - create p-heap node at end of H with given items and return it *)
(* get_parent - return parent of given node *)
(* get_elem - get ith element out of a node *)
(* swap(node,i,node2,j)  - swap ith element of node with jth element of node2 *)


(* insert elems (array length p) in to p-heap H *)
def insert(H, p, elems):
    node = add_to_end(H, elems)
    for i in 1...p:
        bubble_up(node, i, p)

def bubble_up(node, i, p):
    parent = get_parent(node)
    elem = get_elem(node, i)
    best = infinity
    best_ix = nil
    for j in 1...p:
        parent_elem = get_elem(parent, j)
        if elem > parent_elem and parent_elem < best:
            best = parent_elem
            best_ix = j
    if best_ix is not nil:
        swap(node, i, parent, best_ix)
        bubble_up(parent, best_ix, p)
```

Extract Max Pseudocode

```
(* assumed definitions: *)
(* get_elem - get ith element out of a node *)
(* swap(node,i,node2,j)  - swap ith element of node with jth element of node2 *)
(* left/right - get left or right subtree of a node *)
(* detach_root - remove root node from heap and return it *)
(* attach_root - set root of the tree to given node *)
(* detach_last - remove last node from heap and return it *)
(* items - return items on a node *)

def extract_max(H, p):
    top = detach_root(H)
    last = detach_last(H)
    attach_root(H, last)
    for i in 1...p:
        bubble_down(last, i)
    return items(top)

def bubble_down(node, i, p):
    best = get_element(node, i)
    best_ix = nil
    best_node = nil
    for j in 1...p:
        check = get_element(left(node), j)
        if check > best:
            best = check
            best_ix = j
            best_node = left(node)
        check = get_element(right(node), j)
        if check > best:
            best = check
            best_ix = j
            best_node = right(node)
    if best_ix is not nil:
        swap(node, i, best_node, best_ix)
        bubble_down(best_node, best_ix, p)
```