Homework 4

Matt Forbes

November 16, 2010

Problem 1

a) A linked list provides a very good representation of the mechanics of the game. The list is the 'circle' that the kids are lined up in, and simulating the game is as easy as iterating through the list and removing every w^{th} kid.

To start with, we would need to build the list. Each kid gets his own node in the list, and they are inserted in order (where order is determined by the circle, original caller is 1, his right is 2, etc.). This would take $O(p)$ time as there are p kids.

Simulating the selection requires $p-1$ deletions from the list. One for each kid that is eliminated. Starting from the head of the list, follow $w-1$ links in the list. Then delete the next link. Now, follow $w-1$ links again, and then delete the next. Whenever you reach the end of the list, wrap around to the first element (counting that as one visit). Continue doing this until there is only one link left in the list. That last element remaining is the kid who is "it." In this step, we followed $w-1$ pointers exactly $p-1$ times, which is $O(pw)$.

Adding the running times of both the creation of the list and elimination phase gives $O(p+pw)$ which is more simply 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 *)

(* remove p greatest elements in heap *)
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)
```