Homework 3

Fundamental Algorithms, Spring 2003, Professor Yap

Due: Wed March 5, in class

Solutions prepared by: T.A. Igor Chikanian.

INSTRUCTIONS:

- REMINDER: Our Midterm Exam will be held in the Week of March 10th. The midterm will be held either on Monday or Wednesday.
- Please read the questions carefully. When in doubt, please ask.

---

1. (20 Points)

Let us see the details for implementing insertion in (a, b)-trees. Assume each node u stores an array of alternating "key/references" elements:

$$u = (r_0, k_1, r_1, k_2, r_2, \ldots, r_{m-1}, k_m, r_m)$$

where r_i's are references (or pointers) to nodes, and k_j's are keys, with

$$k_1 < k_2 < \cdots < k_m.$$

So this node has m keys, but $m + 1$ references. We have the general constraint $a \leq m + 1 \leq b$ with two exceptions: in case u is the root, then we only have $2 \leq m + 1 \leq b$. And in case u is the child of the root, we only have

$$\lfloor (b + 1)/2 \rfloor \leq m + 1 \leq b.$$

Recall the method described in the Lecture Notes, where a, b satisfy the inequality

$$a \leq \frac{2b + 1}{3}.$$

In this question, we develop 3 subroutines that are useful for implementing insertion.

(a) Suppose we insert a new key k (with an associated reference) into u. The resulting node has the same alternating structure as before; if the node now has $\leq b - 1$ keys, we are done. So assume that u now has b keys. Suppose u is the root. Describe how to split u and form a new root.

(b) Continuing from (a), assume that u is not the root. Then we look at the siblings of u (note that u has either one or two siblings). If any sibling has less than $b - 1$ keys, then we can move one of our keys to this sibling. Describe this "move" in detail. HINT: keys and references in the parent of u is affected as well.

(c) Suppose that (b) fails. Then u has b keys and it also has a sibling v with $b - 1$ keys. Describe how to split u and v into three nodes, and modify the (a, b)-tree. HINT: Be sure to describe how this affects the common parent w of u and v. Furthermore, you must clearly describe the recursive nature of this process.

NOTE on writeup: we prefer that you describe (a) and (b) without using a specific programming language, but in precise (mathematical) terms. If you must using a programming language, use it ONLY to supplement your English prose. Drawing pictures is very useful, but pictures are silent – you must describe explicitly how to interpret your pictures.

Solution:

(a) Root u temporarily has $b + 1$ references (see Figure 1).

Choose $s = \lfloor (b - 1)/2 \rfloor$. Split the node into 2 new nodes as shown in the figure, and make them the children of a new root. The only key of the new root is k_{s+1} (see Figure 2).

To prove that this split is correct, we must show that each child of the root has at least $\lfloor (b + 1)/2 \rfloor$. We suggest you verify this for yourself.

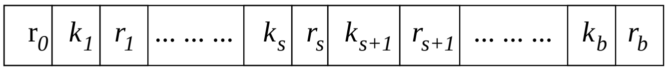

Figure 1: Q1(a) Temporary root u

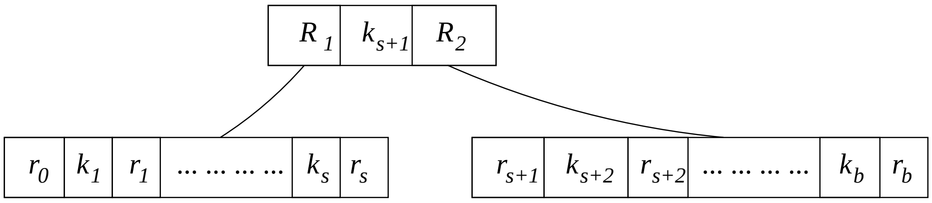

Figure 2: Q1(a) Final Configuration

(b) u is not a root but it has a sibling v with less than b references (see Figure 3).

We move the r_b of u into v, but k_b moves into the parent, and K_{i+1} (from the parent) moves into v as in Figure 4.

(c) u temporarily has b+1 children, and a sibling v with b children (Figure Figure 5)

We temporarily merge u and v into w, thus key K_{i+1} of parent gets inserted into w (see Figure 6.) Now choose $s = \lfloor \frac{2b+1}{3} \rfloor - 1, s' = 2b - \lceil \frac{2b+1}{3} \rceil$. Then we split w into 3 parts as shown on Figure 7.

After this entire procedure, the parent of former u and v might have an overflow of children, and if so, we must recursively call the whole procedure of Question 1 on the parent.

We must verify that the choice of s, s' is correct. It is easily checked that the first child has $s + 1 = \lfloor (2b+1)/3 \rfloor$ children, and the last child has $2b - s' = \lceil (2b+1)/3 \rceil$ children. It follows that the middle child has $\lfloor (2b+1)/3 \rceil$ children.

2. (20 Points)
Describe an efficient algorithm which, given two closed paths $p = (v_0, v_1, \ldots, v_k)$ and $q = (u_0, u_1, \ldots, u_\ell)$, determine whether they represent the same cycle, i.e., they are equivalent. What is the complexity of your algorithm? Make explicit any assumptions you need about representation of paths and vertices.

Solution:

Let us make the following assumptions about the representations of closed path: assume the vertices are distinct integers, and a closed path will be an array of integers $P[0..L]$, where $P[0] = P[L]$. Further, let us introduce the following procedures:

Procedure *Find* returns the position of vertex v in path p if found, -2 otherwise.

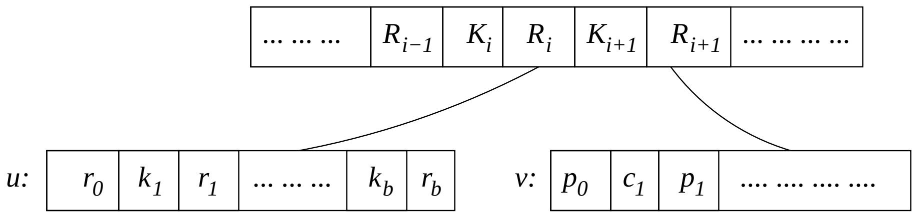

Figure 3: Q1(b) Initial configuration

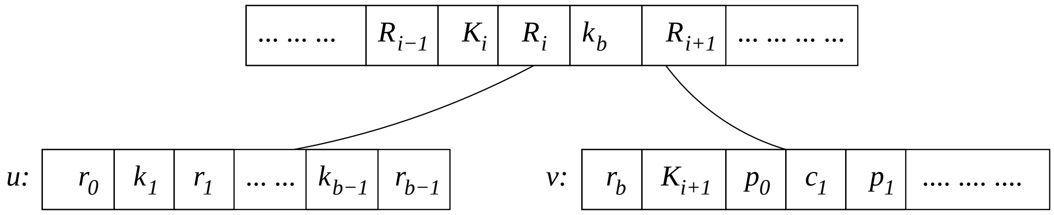

Figure 4: Q1(b) Final Configuration

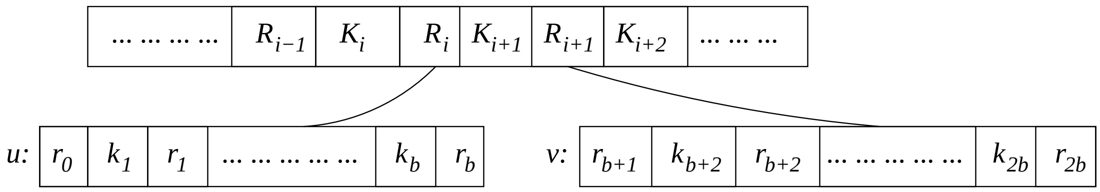

Figure 5: Q1(c) Initial Configuration

```
int   Find(int v, path p, int z (starting position) )
      i := z;
      while i ≤ length(p)
            if v==p[i] return i;
            i := i + 1;
      return −2;
```

Procedure $Match$ returns True/False depending on whether path p matches path q starting at $q[j]$ modulo length L.

```
Bool Match(path p, path q, int j, int L)
      for   i = 0 to L
            if p[i] ≠ q[j + i mod L] return False;
      return True;
```

Procedure $Equal$ will determine if two given cycles are equivalent:

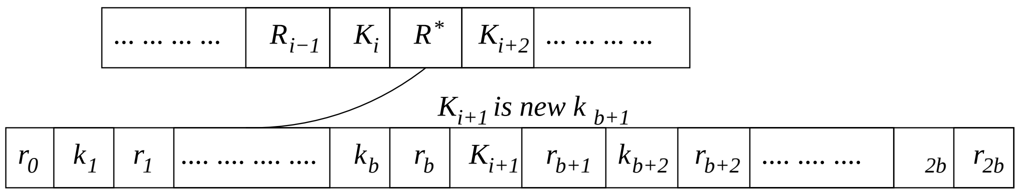

Figure 6: Q1(c) Intermediate Configuration

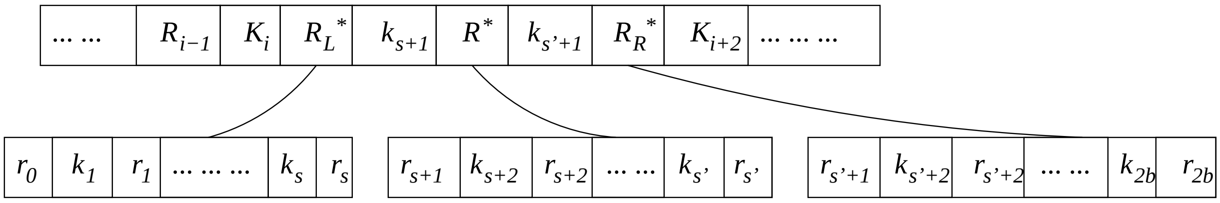

Figure 7: Q1(c) Final Configuration

Bool *Equal*(path p, path q)
    if $length(p) \neq length(q)$ return False;
    $L := length(p)$;
    $j := 0$;
    while $j \geq 0$
        $j := Find(p[0], q, j)$;
        if $j < 0$ return False;
        if $Match(p, q, j, L)$ return True;
        $j := j + 1$;

The above solution will determine equivalence even of self-intersecting cycles. Let L be the length of the cycles. Then the running time on simple cycles is $\mathcal{O}(L)$, since *Find* and *Match* will be called only once. The running time on cycles with self intersections can be as high as $O(L^2)$ in the worst case.

To see the $O(L^2)$ behavior is possible, consider the closed paths $p = a^n ba$ and $q = a^{n+2}$. We suggest you convince yourself that the behaviour is $O(L)$ when the paths are simple.

3. (20 Points)
(a) Describe an efficient algorithm for computing the reduced graph G^c for any input digraph G. Assume the digraph has an adjacency list representation, and that its vertex set is $V = \{1, 2, \ldots, n\}$. State other assumptions you need. If the reduced graph G^c has m vertices, assume that the vertex set of G^c are indexed by $\{1, 2, \ldots, m\}$.
(b) Analyze the running time of your algorithm in (a). This can be fairly brief.

Solution:

(a) In class Lecture Notes (page 16 of Lecture IV) it is shown how to obtain a permutation of graph's vertices, such that the DFS trees, searched in that order by the Driver program will be exactly the strong components. So let us assume that we have this permutation P.

In the following procedures, labeling a vertex u by $[m]$ means that the u belongs to the mth strong component. (So two vertices are in the same strong component if and only if they have the same label.) The following algorithm is a variant of the STRONG_COMPONENT_SUBROUTINE in the Lecture Notes. It constructs the reduced graph G^c, given a graph G:

```
Procedure Construct − G^c
Input: G = (V, E), |V| = n, Permutation P[1..n] from above.
Output: G^c = (V^c, E^c).
Initialize G^c = (V^c, E^c) to be empty.
Let m = 1;
for i = 1 to n
    if unseen(P[i])
        Add vertex m to V^c;
        DFS'(P[i], m);
        m := m + 1;
```

```
Procedure DFS'(int v, int m)
mark v as seen;
label v by [m];
for each w in adjacency list of v:
    if seen(w)
        if (w is labeled by [k]
        and m ≠ k) then
            Add edge (m, k) to E^c;
    else
        DFS'(w, m);
```

(b) The permutation P is constructed in time $\mathcal{O}(|V| + |E|)$ and the above DFS' traverses each edge once, with the driver cycling trough each vertex once, so the total time is $\mathcal{O}(|V| + |E|)$.

NOTE: this $O(|V| + |E|)$ bound comes from standard arguments for DFS and BFS driver programs. If you are not familiar with this, please check the notes.

4. (20 Points)
In Lecture IV, §7, we defined what it means for a permutation array $per[1..n]$ to be a "topological ranking" of a DAG G.
(a) Modify the Simple DFS Algorithm in the Lecture Notes to compute such a permutation. The idea is this: *when you reach a leaf in the DFS Tree, you know that you can safely put this leaf at the end of the permutation.* HINT: you will need a Driver Program for the Simple DFS Algorithm. You need only to specify the "Application Specific" subroutines such as INIT, VISIT and POSTVISIT used in the Simple DFS Algorithm.
(b) Briefly argue why your algorithm is correct.

Solution:

(a)

```
Procedure Driver
Input: DAG G = (V, E), n = |V|
Output: modified permutation P[1..n] containing topological ranking of G.
rank := n;
for  i=1 to n
    if unseen(i) DFS^t(i);
```

```
Procedure DFS^t(v)
mark v as seen;
for each w adjacent to v:
      if unseen(w) DFS^t(w);
P[rank] := v;
rank := rank − 1;
```

(b) We start with $rank = n$. Then we perform DFS on the dag, and only after making sure that all descendants of a vertex v are topologically ranked via DFS^t (i.e. their priorities are set as later with respect to v), do we rank v and then decrease the global $rank$ variable. The driver ensures that all of the DAG will be covered.