Homework 10

15-210: Parallel and Sequential Data Structures and Algorithms (Fall'12)

`tag` — Version 1.0

***due:** Fri, Dec 7th @ 11:59pm*

1 Introduction

In this assignment you will explore another dynamic programming application, consider another way to handle deletions when using linear probing for hash tables, and practice with leftist heaps. This assignment is strictly a written homework; there are no programs that you need to hand in. It should be good practice for the final.

1.1 Submission

Submit your solutions by placing your solution files in your handin directory, located at

```
/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn10/
```

Name the files exactly as specified below. You can run the check script located at

```
/afs/andrew.cmu.edu/course/15/210/bin/check/10/check.pl
```

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw10.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

Writeups should be short (excessively long or unclear answers will lose points even if they are correct).

2 Independent Sets in Trees

Suppose you have a rooted tree on n vertices, where each vertex has an integer weight. Your goal is to find an independent set A of vertices such that the sum of the weights of the vertices in your set is maximized. Recall that A is an independent set if no two vertices in A are connected by an edge.

You will develop an $O(n)$ work dynamic programming solution to this problem. Let `root` be the root of weighted tree, and let W_v and C_v represent weight and the children of a vertex v, respectively. Note that a tree only has $n-1$ edges, so $\sum_v |C_v|$ is $O(n)$.

Assumptions:

- Vertex weights may be negative.
- There may be duplicate weights.
- You may *not* assume anything about the structure of the tree.
- The empty set has a sum of 0.

Even though you will need to find an actual independent set of vertices of maximum weight, you will first find the maximum achievable weight of independent sets of vertices. Later you will reconstruct an actual set of vertices

Task 2.1 (5%). Clearly define the subproblems you will solve. Which problem provides the overall maximum weight?

Task 2.2 (15%). Use a recursive definition to describe your dynamic programming solution, including all base cases (you can use either mathematical notation or code). You do not need to show memoization.

Task 2.3 (5%). Give a bound on the number of vertices (distinct subproblems) in your DAG and the longest path in the DAG, in terms of the number of vertices n and the depth d of the tree. What is the work and span of your algorithm. Use Θ notation and explain why?

Task 2.4 (10%). Describe how you can find an actual independent set of vertices that has the maximum weight returned by your algorithm above. You need find just one such independent set.

3 Delete with linear probing

As described in lecture, *open addressing* is a hashing technique that does not need any linked lists but instead stores every key directly in the array. Every cell in the array is either empty or contains a key. A probe sequence is used to find an empty slot to insert new elements. Recall that with *linear probing*, to insert a key k, it first checks the position at $h(k)$, and then checks each following location in succession, wrapping to position 0 as necessary, until it finds an empty location. The advantage of linear probing over separate chaining or other probing sequences is that it causes fewer cache misses since typically all probe locations are on the same cache line.

One issue with open addressing, however, is that you cannot simply delete elements from the table, as you then may not be able to find elements that follow the deleted elements in their probe sequence. The solution suggested in lecture is to use a *lazy delete* where deleted elements are replaced with a special `HOLD` value. The problem with lazy deletes is that they persist to fill the table, and result in slower insert and find operations, and more frequent rehashing of the complete table to clear the `HOLD` locations. Lazy deletes are particularly problematic with linear probing, as the keys tend to cluster. Any key that hashes to any position in a cluster (not just collisions), must probe beyond the cluster and adds to the cluster size. Worse yet, this clustering also makes it more likely that the cluster will be lengthen even further. Once the table fills beyond two-thirds full (either with keys and `HOLD` values), linear probing's performance rapidly degrades.

With linear probing, though, it is possible to take another approach for handling deletes: fill a hole created by deleting a value x with another value y in the table. When filling the hole with y, we need to ensure that the value y, and all other values in the table, can still be found with a standard search. Also note that filling the whole with y could create a new hole that might need to be filled. Your task is to write delete using this approach when using linear probing.

Task 3.1 (2%). Beyond needing to fill the new hole, give one other reason why you might not be able to use the value at the position immediately after the hole when using the above method.

Task 3.2 (1%). Which value(s) could safely fill the deleted value so that it is still possible to find it?

Task 3.3 (2%). How would you fill the new hole made by moving y? When can you stop finding new values to fill a hole?

Task 3.4 (15%). Write a function `delete`(T, k) (in pseudocode or SML) that replaces the key k with another value in the table T.

4 Leftist Heaps

As covered in lecture, a *leftist heap* is a data structure for priority queues that holds the keys in a binary tree that maintains the heap property. Unlike binary heaps, however, it not does maintain the complete binary tree property. The goal of the leftist heap is to make the `meld` operation run in $O(\log n)$ work.

Task 4.1 (5%). Given a sequence $S = \langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$, what is the resulting leftist heap after applying the following code?

```
fun singleton v = Node(1, v, Leaf, Leaf)
Seq.reduce meld Leaf (Seq.map singleton S)
```

where `meld` for leftist heaps is as given in the lecture 27 notes.