CMSC 420: Spring 2023

Solutions to Homework 4: Hashing, B-Trees, and Tries

Solution 1:

(a) See Fig. 1 (left).

```
insert("X")
0  1  2  3  4  5  6  7  8  9 10 11 12 13
A  M  L  X  H  C  I  J  F  B
            3 probes

insert("Y")
0  1  2  3  4  5  6  7  8  9 10 11 12 13
Y  A  M  L  X  H  C  I  J  F  B
            2 probes

insert("Z")
0  1  2  3  4  5  6  7  8  9 10 11 12 13
Y  A  M  L  X  H  Z  C  I  J  F  B
            10 probes
```

(b) See Fig. 1 (right). Note that the quadratic offsets are taken with respect to the initial hash position. (For example, for "X", we access locations \( h("X") + \{0, 1, 4, 9, \ldots\} = 3 + \{3, 4, 7, 12, \ldots\} \) modulo 13.)

In the case of "Z" we go into an infinite loop. To see why, a quick Internet search reveals that the quadratic residues of 13 are \( \{0, 1, 3, 4, 9, 10, 12\} \). We start at \( h("Z") = 4 \). So (taking indices mod 13), we will probe the locations \( \{4, 5, 7, 8, 0, 1, 3\} \). But all these positions are already occupied, so we will never find an empty spot.

(c) See Fig. 2. In the case of "Z" we fail even though there is room in the table. This is because all of the positions we will probe \( h("Z") + i \cdot g(Z) = (3 + 5i) \mod 15 = \{3, 8, 13\} \), but all are already occupied. This illustrates why \( m \) and \( g(x) \) should be relatively prime.

```
insert("X")
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
E  F  W  P  L  J  Q  N  X
    X

insert("Y")
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
E  F  W  P  L  J  Q  Y  N  X
    Y

insert("Z")
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
E  F  W  P  L  J  Q  Y  N  X
    Z
```

Figure 1: Hashing with linear and quadratic probing.

```
Figure 2: Hashing with double hashing.
```
**Solution 2:** The solutions are shown in Figs. 3–5. Since \( m = 4 \), each node can have from 2 to 4 children and from 1 to 3 keys. Inserting 23 causes its leaf node to split. (Key rotation is not possible.) By convention, the left node gets one key and the right gets two, and the remaining middle key is promoted to the parent. This propagates to its parent, which also splits. Inserting 55 also causes its leaf node to split, but in this case a key rotation (adoption) with the right sibling is possible. The key 51 rotates into the parent, key 60 rotates to the right child, and the subtree \([54, 55]\) moves to the right child. The deletion of 62 causes a merge with the sibling leaf. The parent also underflows, but it can adopt from the sibling to its left. The key 47 rotates into the parent, key 60 rotates to the right child, and the subtree \([50, 51, 55]\) moves to the right child.

![Figure 3: Solution to Problem 2(a): insert(23).](image3.png)

![Figure 4: Solution to Problem 2(b): insert(55).](image4.png)

![Figure 5: Solution to Problem 2(c): delete(62).](image5.png)
Solution 3: See Fig. 6. The substring identifiers are shown (in suffix order) in the upper left. They are sorted lexicographically in the lower left. The final suffix tree is shown on the right.

![Figure 6: Suffix tree.](image)

Solution 4:

(a) We present the algorithm in recursive form. The initial call is `pcHelper(root, pattern)`. When we arrive at a node p, if p is null, we must have fallen off a next-sibling chain, and so we fail. If we run out of symbols to match (`pattern.size == 0`) we have successfully found a matching node, and so we return its weight. If p is an external node then our pattern must be longer than any possible matching string, and again we fail. Otherwise, we search for a matching symbol on the current level. While the current key is smaller than the current pattern symbol, we keep moving to the right. If we find a match, then we strip off the current character from the pattern (`pattern.substring(1)`), and recurse downwards. Otherwise, we have gone beyond the pattern symbol, and again we fail.

```java
int pcHelper(Node p, String pattern) {
    if (p == null) // fell out of the tree?
        return 0
    else if (pattern.size == 0) // run out of pattern symbols?
        return p.weight // everything in this subtree matches
    else if (p.isExternal) // hit external too soon?
        return 0
    else if (p.key < pattern[0]) // scan for next pattern symbol
        return pcHelper(p.nextSibling, pattern)
    else if (p.key == pattern[0]) // matched this symbol?
        return pcHelper(p.firstChild, pattern.substring(1)) // recurse
    else return 0 // gone too far - no match
}
```
(b) We present the algorithm in recursive form. The initial call is `wcHelper(root, pattern)`. When we arrive at a node `p`, if `p` is `null`, we must have fallen off a next-sibling chain, and so we fail. Otherwise, we consider two cases, depending on the node type. If it is an external node and we have exhausted all the pattern symbols, we have a match, and return 1. Otherwise, the pattern is longer than the string at this node, and we fail.

If this is an internal node, we check first whether we have run out of pattern symbols, and if so we fail. Otherwise, if we run out of symbols to match (`pattern.size == 0`) we have successfully found a matching node, and so we return its weight. Next, if the pattern symbol is "*", we automatically match and we make a recursive call to the child, and strip this symbol off the current symbol from the pattern (`pattern.substring(1)`). We also invoke the function on the sibling and combine the two counts. Otherwise, we scan along the list of siblings, until we either find a match or we go too far. If we find a match, we recurse on the child and strip off the first symbol of the pattern.

An example showing a trace of the recursive calls is given in Fig. 7.

```java
int wcHelper(Node p, String pattern) {
    if (p == null) // fell out of the tree?
        return 0
    else if (p.isExternal) { // external node
        if (pattern.size == 0) // matched last pattern symbol?
            return 1
        else // more pattern symbols remain?
            return 0
    }
    else { // internal node
        if (pattern.size == 0) // no more pattern symbols?
            return 0
        else if (pattern[0] == "*") // wildcard symbol?
            return wcHelper(p.firstChild, pattern.substring(1)) + wcHelper(p.nextSibling, pattern)
        else if (p.key < pattern[0]) // keep looking for next symbol
            return wcHelper(p.nextSibling, pattern)
        else if (p.key == pattern[0]) // matched this symbol?
            return wcHelper(p.firstChild, pattern.substring(1)) // next symbol
        else return 0 // no match among siblings
    }
}
```

**Solution to the Challenge Problem.:** Our algorithm follows Aesop’s fabled race between the tortoise and the hare. The lead pointer (the hare) advances two steps with every iteration, and the trailing pointer (the tortoise) advances once with every iteration. If the list ends, the hare will be the first to hit `null`, and reports that the list is “open.” On the other hand, if there is a cycle, the hare will eventually enter it and will loop forever. However, in this case, the tortoise will also arrive to the loop (after about twice as much time) and will also start looping as well. Because the hare moves faster, it will eventually pass the tortoise within the loop. When we detect this, we report that the list is “closed.”
openOrClosed(Node head) {
    Node tortoise = hare = head; // start the race
    while (hare != null && hare.next != null) { // hare hasn’t hit end
        hare = hare.next.next; // advance hare by 2
        if (tortoise == hare || tortoise.next == hare) // hare passes tortoise?
            return "closed" // ...must be looped
        tortoise = tortoise.next; // advance tortoise by 1
    }
    return "open"
}

You might notice that we advance the hare before the if-statement and the tortoise after. This is to avoid triggering the loop detection on the very first iteration.

We claim that the running time is $O(n)$, where $n$ is the number of nodes. If the list is open, then we discover when the hare hits the end, after $n/2 = O(n)$ iterations.. On the other hand, if the list has a cycle, let $m$ denote the number of nodes in the cycle. After $n - m$ iterations, both the tortoise and the hare have made it into the cycle. Because the hare moves twice as fast, after an additional $m$ iterations, the tortoise will make one full lap around the cycle, but the hare will make two full laps. So, no matter where they started in the cycle, the hare must have passed the tortoise at some point. The total number of iterations before detecting this is at most $n - m$ (to get both into the cycle) plus $2m$ (chasing each other within the cycle), which yields a total running time of

$$(n - m) + 2m = n + m \leq 2n = O(n).$$

Figure 7: Trace of recursive calls by \texttt{wcHelper("*ab")} (blue) and return values (red).