CMSC 420: Spring 2023

Homework 4: Hashing, B-Trees, and Tries

Handed out Tue, May 2. Due **Thu, May 11, 9:30am** (that is, by the start of class).

Note: Solutions will be discussed in class, so **no late submissions will be accepted**. You may drop your lowest homework score.

Problem 1. (15 points) In this problem, you will show the result of inserting a sequence of three keys into a hash table, using linear and quadratic probing and double hashing. For each operation (9 in total) show contents of the hash table after the insertion. If successful, indicate the number of *probes*, that is, the number of array elements accessed. (Note that the initial access counts as a probe, so if there is no collision, the number of probes is 1.) If the sequence fails, i.e., loops infinitely, indicate that the insertion *fails*.

See Figs. 3 and 4 from Lecture 15 for an example.

(a) (5 points) Show the results of inserting the keys "X" then "Y" then "Z" into the hash table shown in Fig. 1(a), assuming *linear probing*. (Insert the keys in *sequence*, so each inserted key remains for the later insertions.)

(a) Linear probing

```
insert("X")  h("X") = 2
insert("Y")  h("Y") = 13
insert("Z")  h("Z") = 11
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	M	L		H		C	I			J	F	B

(b) Quadratic probing

```
insert("X")  h("X") = 3
insert("Y")  h("Y") = 12
insert("Z")  h("Z") = 4
```

0	1	2	3	4	5	6	7	8	9	10	11	12
F	W		P	L	J		Q			N		

Figure 1: Hashing with linear and quadratic probing.

(b) (5 points) Repeat (a) using the hash table shown in Fig. 1(b) assuming *quadratic probing*.

(c) (5 points) Repeat (a) using the hash table shown in Fig. 2 assuming *double hashing*, where the second hash function g is shown in the figure.

(c) Double hashing

```
insert("X")  h("X") = 4; g("X") = 2
insert("Y")  h("Y") = 9; g("Y") = 3
insert("Z")  h("Z") = 3; g("Z") = 5
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
E				M		K			F			B	Q	

Figure 2: Hashing with double hashing.

Problem 2. (12 points) Consider the B-trees of order 4 shown in Fig. 3 below. Let us assume two conventions. First, key rotation (when possible) has precedence over splitting/merging.

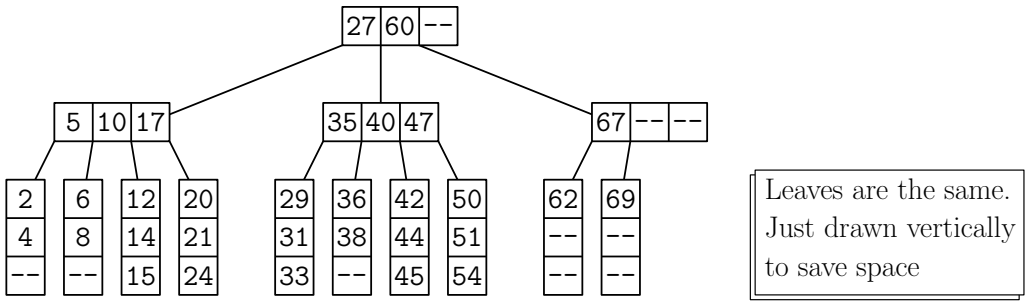


Figure 3: B-tree operations.

Second, when splitting a node, use the convention given in the lecture, that the left node has $\lceil m/2 \rceil$ children and the right node has the remaining children.

(a) (4 points) Show the final tree that results after inserting the key `23` into the tree of Fig. 3.

(b) (4 points) Show the final tree that results after inserting the key `55` into the (original) tree of Fig. 3.

(c) (4 points) Show the final tree that results after deleting the key `62` from the (original) tree of Fig. 3.

(Intermediate results are not required, but may be given to help assigning partial credit.)

Problem 3. (12 points) In this problem we will build a suffix tree for $S =$ `"abbabbaba$"`.

(a) (4 points) List the 10 substring identifiers of S as shown in Fig. 7 of Lecture 18.

(b) (8 points) Draw the suffix tree for S. Use the same form as shown in Fig. 7 of Lecture 18. When drawing your tree, order the children of each node from left to right in the order `a` $<$ `b` $<$ `$`. (**Hint:** Be sure to compress paths wherever possible. Note that there are online suffix tree generators, but we strongly encourage you to use these only to verify your final answer. They generally do not match the format we are looking for, and you won't be able to use them on the final exam.)

Problem 4. (11 points) In class, we discussed the de la Briandais representation of a trie, in which the trie is stored using the first-child/next-sibling tree representation (see Fig. 4). We assume that the strings stored in the tree are *prefix free*, meaning that no string is a prefix of any other.

Let us assume our tree is an extended tree. Each internal node `p` store a single character as the `p.key` and its first-child and next-sibling pointers, `p.firstChild` and `p.nextSibling`, respectively. External nodes store the final string as `p.key`. To distinguish between node types, each node stores a boolean, `p.isExternal`, which is `true` for external nodes and false otherwise. Finally, each node contains a value `p.weight` which indicates the number of external nodes in the subtree rooted at `p`. Let `root` denote the root of the tree.

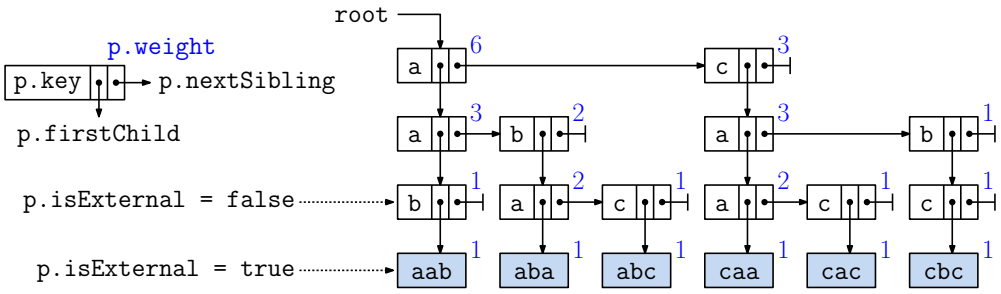


Figure 4: Pattern searching in a de la Briandais trie.

(a) (5 points) Present pseudocode for an efficient function `int prefixCount(String pattern)`, which returns a count of the number of strings having `pattern` as a prefix. If no string has this prefix, return 0.

For example, in Fig. 4, the call `prefixCount("ca")` would return 2, for the strings `"caa"` and `"cac"`. The call `prefixCount("abacus")` would return 0, since no string has this as a prefix.

Your algorithm should run in time proportional to the product of the length of the pattern string and the alphabet size. (**Hint:** Your algorithm may be recursive or iterative. You may use any standard string processing utilities you like.

(b) (6 points) Present pseudocode for an efficient function `int wildCount(String pattern)`, which is given a pattern string containing *wildcard symbols*. If the pattern string contains the symbol `"*"`, this will match any single character at this position. In order to be counted, an entire string of the dictionary must match the entire pattern.

For example, in Fig. 4:

`wildCount("*a*")`: Matches all 3-character strings having an `"a"` as the middle character, and so would return 3 (for the strings `"aab"`, `"caa"`, and `"cac"`).

`wildCount("*bb")`: Matches all 3-character strings ending in `"bb"`, and so returns 0.

`wildCount("c")`: Matches the string `"c"`, and so would return 0.

`wildCount("c**")`: Matches all 3-character strings that start with `"c"`, and so would return 3 (for the strings `"caa"`, `"cac"`, and `"cbc"`).

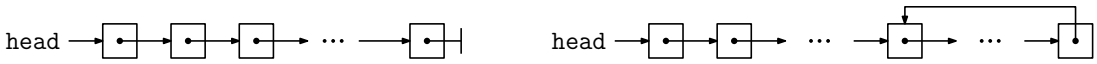
Challenge Problem. You are given a very, very long linked list, where each node contains a single member, `next`, which points to the next element in the linked list. The variable `head` points to the head of the linked list. There are two possible forms that the list might take:

Open: The linked list eventually ends with a `null` pointer (see Fig. 5(a)).

Closed: The linked list loops back on itself, with one `next` pointer that points to an earlier node in the list (see Fig. 5(b)).

Describe an efficient function that determines whether the list is open or closed. Here is the catch:

- You do not know how many elements are in the list



(a): Open (b): Closed

Figure 5: Open or closed linked list?

- You cannot modify the nodes of the list, you do not know how many nodes there are in the list
- You cannot use more than a constant amount of additional working storage

This means that you cannot create an array, you cannot allocate any linked structures, and you cannot make recursive calls (since recursive calls implicitly use the system's recursion stack, which is an array).

Your function should run in $O(n)$ time, where n is the number of elements in the linked list.