CS 581 Homework 5 Solution

Due on 02/15/2018

Problem 1.

a) Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

b) Show how to implement a stack using one queue. Analyze the running time of the stack operations.

c) What do you think is the essential difference between a stack and a queue? In other words, what's the reason behind the fact that a queue can simulate a stack but a stack alone cannot simulate a queue?

Answer

a) Keeping two stacks, *in* and *out*, we can implement a queue structure. The queuing method would be implemented in constant time: Queuing elements would be pushing them into the *in* stack. Switching from queuing to dequeuing would run in linear time: All elements from *in* are popped out and pushed into *out*, then the elements can be dequeued in constant time with pops. So enqueue is $O(1)$ and dequeue is $O(n)$. The same can be said about switching from dequeuing to queuing.

b) To use a single queue for implementation of a stack, the push operation would run in linear time. This operation would require every a re-order of the queue every time a new element is pushed. Specifically, after queuing the new element, every element in front of the new element (the rest of the queue) must be dequeued and queued back so that the next pop will result in a simple dequeue. Since this is the case, the pop operation runs in constant time. Similarly, if push is $O(1)$ time then pop must be $O(n)$.

c) You essentially access element in any position in a queue, which is basically a RAM, but with more time complexity. While with a stack, you can only access the top element. With two stacks you can again access any element by popping and pushing.

Problem 2.

Linux kernel extensively uses linked list. However, Linux implementation of the linked list is different from the many linked list implementations you might have seen. Usually a linked list contains the items that are to be linked. For example:

```
struct my_list{
  void *myitem;
  struct my_list *next;
  struct my_list *prev;
};
```

The kernel's implementation of linked list gives the illusion that the list is contained in the items it links! For example, if you were to create a linked list of struct `my_klist` you would do the following:

```
struct list_head{
  struct list_head *next;
  struct list_head *prev;
}

struct my_klist{
  struct list_head list; /* kernel's list structure */
  int my_data;
};
```

Based the linked lists above, answer the following questions and justify the answers.

a) There must be good reasons why Linux implements linked lists that way. Name at least one advantage of the design of `my_klist` over `my_list`?

b) Does the design of `my_klist` change any complexity of adding, deleting and traversing the linked list? If so, does it increase or reduce the complexities?

c) For a given `list_head` pointer, how do you access the actual data item associated with it, such as `my_data`?

Answer

a) Type oblivious (linking nodes of different types together); Can have multiple lists in one node, etc.

b) No. Same complexity.

c) This is performed by a macro in the kernel:

```
#define list_entry(ptr, type, member) \
((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

ptr is the pointer to the *list_head* node. *type* is the type of the encapsulating data structure, and *member* is the name of the *list_head* member variable of the encapsulating data structure.

Problem 3.
Let H be a class of hash functions in which each $h \in H$ maps the universe U of keys to {0, 1, ... , m-1}. We say that H is **2-universal** if, for every fixed pair $\langle x, y\rangle$ of keys where $x \neq y$, and for any h chosen uniformly at random from H, the pair $\langle h(x), h(y)\rangle$ is equally likely to be any of the m^2, pairs of elements from {0, 1, ... , m-1}. (The probability is taken only over the random choice of the hash function).

a) Show that, if H is 2-universal, then it is universal.

b) Construct a specific family H that is universal, but not 2-universal, and justify your answer. Write down the family as a table, with one column per key, and one row per function. Try to make m, $|U|$ and $|H|$ as small as possible. (Hint: There is an example with m, $|H|$ and $|U|$ all less than 4).

c) Suppose that an adversary knows the hash family H and controls the keys we hash, and the adversary wants to force a collision. In this problem part, suppose that H is universal. The following scenario takes place: we choose a hash function h randomly from H, keeping it secret from the adversary, and then the adversary chooses a key x and learns the value $h(x)$. Can the adversary now force a collision? In other words, can it find a $y = x$ such that $h(x) = h(y)$ with probability greater than $1/m$? If so, write down a particular universal hash family in the same format as in part (b), and describe how an adversary can force a collision in this scenario. If not, prove that the adversary cannot force a collision with probability greater than $1/m$. (Hint: it's possible to extend the example of the previous problem to get the answer).

d) Answer the question from part (c), but supposing that H is 2-universal, not just universal.

Answer

a) if H is 2-universal, then for every distinct x, y, and $i \in \{0, 1, ...m-1\}$ the probability of $\langle h(x), h(y)\rangle = \langle 1, 1\rangle$ is m^{-2}. There are exactly m possible ways for x and y to collide. Therefore, the probability of $h(x) = h(y)$ is the sum of the probability of $\langle h(x), h(y)\rangle = \langle 1, 1\rangle$, which is $\Sigma_{i=0}^{m-1}(m^{-2}) = \frac{1}{m}$. So by definition, H is universal.

b)

	x	y
h_1	1	0
h_2	0	0

This is a universal hash family, because the probability of 2 keys x and y mapping to the same location is the probability of h_2, which is $\frac{1}{2} = \frac{1}{m}$.
However, since $\langle h(x), h(y)\rangle$ never equals $\langle 0, 1\rangle$ or $\langle 1, 1\rangle$, H is not 2-universal.

c)

	x	y	z
h_1	1	0	1
h_2	0	0	1

Yes. The adversary can determine which hash function is used by supplying x. If $h(x) = 1$ then the adversary knows that we have chosen h_1. Otherwise, the adversary knows that we have chosen h_2. If we have chosen h_1, the adversary will supply z, which will cause a collision. Otherwise the adversary will supply y, which will cause a collision. This is still a universal hash collection, because x and y will collide with probability $\frac{1}{2}$, x and z will collide with probability $\frac{1}{2}$, and y and z will collide with probability $0 < \frac{1}{2}$.

d) No. By the definition of 2-universal hash collections, every fixed pair $\langle x, y\rangle$ have to appear and are equally possible. Knowing $h(x)$ does not give any information about $h(y)$. This is because the two keys are independent.

Problem 4.
Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

Answer For any node, if the right subtree exists, the successor is the left most child in the right subtree, so it has no left child; if the left subtree exists, the predecessor is the right most child in the left subtree, so it has no right child.

Problem 5.
A binary search tree can have poor worst-case performance if it is unbalanced. Name two common self-balancing binary search trees and use concise language to explain how it achieves self-balancing during *insertion* (No need to explain other operations and an insertion may have multiple cases if the tree has multiple types of nodes) and its time complexity. (No need to compare the two techniques).

Answer AVL tree, B-tree, red black tree, 2-3 tree etc. Insertion time complexity is usually $O(\log n)$. Usually every insertion needs to satisfy some properties to make it balanced. For example, red-black tree has the following properties:

- Each node is either red or black.
- The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
- All leaves (NIL) are black.
- If a node is red, then both its children are black.

- Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes. Some definitions: the number of black nodes from the root to a node is the node's black depth; the uniform number of black nodes in all paths from root to the leaves is called the black-height of the redblack tree.

Problem 6.
A group of the local mafia is considering the establishment of a gaming house. The businessmen have already purchased a building with a floor space limitation of 25 square yards. These men are considering using four types of gambling. They include blackjack, poker, craps, and roulette. In the table below, S_i denotes the space required per table and P_i denotes the profit added per table when added the ith table. For example, one blackjack table has a profit of 10 while two blackjack tables have a profit of 17. The estimated value per table and space required is given as follows (estimated by the mafia's financial wizards):

Game	S_i	P_1	P_2	P_3	P_4
Blackjack (i = 1)	4	10	7	4	1
Poker (i = 2)	5	9	9	8	8
Craps (i = 3)	6	11	10	9	8
Roulette (i = 4)	3	8	6	4	2

It is important to note that the investors have realized that their marginal returns decrease for each game as more tables are added. This indicates that the clientele may fill one roulette table all the time but a second table might be idle for part of the time. Thus we have the varying values for each item (game). Assume that at most 4 of each game can be used. How many tables should be installed for each game in order to maximize profits? Solve the problem with a dynamic programming formulation where stage 1 corresponds to the roulette decision, stage 2 to the craps decision, stage 3 to the poker decision, and stage 4 to the blackjack decision. You will need the variables of the DP framework given in the class, such as states, decisions, best return, best decision etc. Provide the following for your solution:

- The 4 tables (one for each game) that are constructed by the dynamic programming approach.
- The maximum profit that the gaming house can provide.
- The optimal combination of games that provides the maximum profit.

Answer See here