Data Structures and Objects
CSIS 3700

Spring Semester 2018 — CRN 21212

First Midterm

Part I. Short answer. 7 points each.

1. What five things should be placed in a header file?

   Answer:

   (a) Other `#include` files
   (b) Constants, `#define`s and macros
   (c) Data structures and classes
   (d) Global variable definitions
   (e) Function prototypes

2. Consider SentinelSearch and ForgetfulBinarySearch. Which is better, and why? What limitations, if any, are there on these methods?

   Answer:
   ForgetfulBinarySearch is the better algorithm, since its run time is in $O(\lg n)$. For ForgetfulBinarySearch to work, the list must be sorted. For SentinelSearch to work, there must be space available at the end of the list for the key to be placed. For SentinelSearch to work well, you must be able to get to the end of the list quickly — in less than $\Theta(n)$ operations.

3. When should you pass data by reference?

   Answer:

   - If the called function has a need to modify the calling function's data *e.g.*, `cin` and its methods
   - If the called function must return more than one value
   - If the data being passed is large

4. I claimed in class that `insert`, `remove` and `search` operations for hash tables require $O(1)$ time on average, if two conditions are met. What are the two conditions?

   Answer:

   (a) The hash function must evenly distribute the keys throughout the table
   (b) The table cannot become overly full; a threshold is established beyond which no new items can be added.

5. In collision resolution, one of the problems we examined involved all of the keys that hash to one slot colliding with each other. How might you overcome this?

   Answer:
   The issue is that linear and quadratic probes as well as pseudorandom number generators follow a fixed *collision path* through the table.

   The solution is to take colliding keys and rehash with a second hash function. Ideally, such a function sends keys along different collision paths.

6. Name three distinct uses for stacks.

   Answer:
   The four key roles are:

   - Data reversal
   - Parsing
   - Postponement of operations
   - Backtracking and recursion

7. What is the purpose of big-O notation? What part of a program do we examine to determine a big-O value?

   Answer:
   Big-O values allow us to estimate how much time (work) a program or algorithm requires to process a dataset of a given size. It also allows us to compare two comparable programs or algorithms to see if either requires significantly more time than the other.

   We focus on the loop structures within the program or algorithm to determine the big-O value.

Part II. Slinging hash. 15 points each.

1. Given the keys `248576`, `659482`, `013546`, `859419`, `150139` and `485437`, show how these keys would be hashed using either mod division or pseudorandom number generation, and by one of digit extraction, folding, rotation or midsquare. Your table must have more than `100` slots.

   Answer:
   There are infinitely many correct answers for this. My solution assumes a table size of `101` positions and uses mod division for one method and digit extraction for the other, extracting the first, last and middle two digits in that order.

Key	*k* mod 101	**Digit Extraction**	**After** mod 101
248576	15	2685	59
659482	53	6294	32
013546	12	0635	29
859419	10	8994	5
150139	53	1901	83
485437	31	4754	7

2. Using mod `101` division, the keys `111`, `212`, `313`, `414` and `515` all collide. Use your favorite collision resolution method to show where the keys would end up.
   Answer:
   The keys all collide in position 10. Using a simple linear probe, the keys would end up in the following positions:

Key	**Final Position**
111	10
212	11
313	12
414	13
515	14

Part III. Code trace. 20 points. Show all output.

```
1  Stack s;
2
3  bool bSearch(int data[],int nItems,int key) {
4    int low=0,mid,high=nItems-1;
5
6    while (low < high) {
7      mid = (low + high) / 2;
8      s.push(mid);
9
10     if (key > data[mid])
11       low = mid + 1;
12     else
13       high = mid;
14   }
15
16   if (key == data[low])
17     return true;
18   else
19     return false;
20 }
21
22 int main(void) {
23   int num,data[] = {3,5,10,15,16,19,23,37};
24
25   if (bSearch(data,8,19))
26     cout << "found" << endl;
27   else
28     cout << "not_found" << endl;
29
30   while (!s.isEmpty()) {
31     num = s.pop();
32     cout << num << endl;
33   }
34
35   return 0;
36 }
```

Answer:
The `main()` function begins by calling `bSearch()`. Throughout the call, `data[]` is the array of eight integers, `nItems` is 8 and `key` is 19. The following table shows the values of key variables during the execution of `bSearch()`:

Line	low	mid	high	data[mid]	s
6	0	–	7	–	⟨⟩
9	0	3	7	15	⟨3⟩
6	4	3	7	15	⟨3⟩
9	4	5	7	19	⟨3, 5⟩
6	4	5	5	19	⟨3, 5⟩
9	4	4	5	16	⟨3, 5, 4⟩
6	5	4	5	16	⟨3, 5, 4⟩

The fourth time line 6 is executed, the loop condition fails and control goes to line 16. Since `low` is 5, `data[low]` is 19, which is the key. Thus, the condition is true and `true` is returned.

Returning to line 25, since the returned value is `true`, `found` is output at this point. Then, the stack is repeatedly popped until empty, outputting each value popped. Thus, the output is:

```
found
4
5
3
```

Part IV. Extra credit. 7 points.

Consider ForgetfulBinarySearch and the regular, quit-early version BinarySearch. I claimed in class that ForgetfulBinarySearch was usually faster than the regular version, even though it may keep looking after finding the key. Why is it usually faster?

Answer:
ForgetfulBinarySearch only performs one comparison per loop iteration, while BinarySearch performs two per iteration. BinarySearch is faster only if it can do at most half as many loop iterations. However, there is a 50% chance that the key is not found until the last possible iteration, a 75% chance of finding it in the last two possible iterations, 87.5% chance of finding it within the last 3 possible iterations, and so on. In general, for a list of n items, the chance of finding the key in half the possible iterations or less is less than $3/\sqrt{n}$.