Practice test for Midterm 1

February 29, 2024

1 C++ review

▶ Write a function which takes a vector of `int` and returns true if all the elements of the vector are positive (≥ 0) and false otherwise:

```
bool all_pos(vector<int> data)
{
```

▶ From assignment 1, the `ordered_array` class, we want to add a function to insert multiple copies of a given value:

```
void ordered_array::insert_multiple(int x, int n)
{
    // Code here
}
```

which inserts n copies of x. Since the array is ordered, all copies will be inserted at the same position. Write the code for `insert_multiple`.

- What does the following function do:

```
bool f(vector<int> x)
{
  bool g = true;
  for(int i = 0; i < x.size() - 1; ++i)
    if(x[i] > x[i+1])
      g = false;

  return g;
}
```

Give examples of non-empty vectors, one which will cause the function to return true, and one which will cause it to return false.

- Given the following ordered array class

```
class ordered_array {
  public:
    ...
    void remove_multiple(int x);
  private:
    int* data;
    int sz;
    int cap;
};
```

Implement the `remove_multiple` method, which should remove all the copies of x in the array (the normal `remove` only removes one copy).

```
void ordered_array::remove(int x)
{
```

2 Big-O analysis; Vectors, lists, stacks, and queues

► This function searches for all occurences of a pattern p in a string s:

```
void find_all(string s, string p)
{
  for(int i = 0; i < s.length() - p.length(); ++i) {
    bool found = false;
    for(int j = 0; j < p.length(); ++j)
      if(s[i + j] ≠ p[j])
        found = false;

    if(found)
      cout << "Found at " << i << endl;
  }
}
```

It p has length m and s has length n, and we assume that n is much larger than m, what is the big-O complexity of this function? Does it have different best- and worst-case complexities?

► Here is a function that checks a vector to see if it is sorted:

```
bool is_sorted(vector<int>& v) {
  for(int i = 0; i < v.size() - 1; ++i)
    if(v[i] > v[i+1])
      return false;
  return true;
}
```

Analyze the cost of this function, in terms of the number of comparisons `v[i] > v[i+1]` C, the number of increments `++i` I, and the number of vector lookups `v[...]` L. What is the best case cost? What is the worst case cost? When (for what inputs) do the best/worst cases occur?

► Given the following implementation of `vector::push_back` trace through the cost of the first 10 pushbacks, if a "cheap" pushback (i.e., a single copy) has a cost of 1, and the initial size and capacity are 0.

```
void vector::push_back(int x)
{
  if(size == capacity) {
    // Full, reallocate to make room
    int* old_data = data;
```

```
    int new_cap = 1 + capacity + capacity / 2
    data = new int[new_cap];

    // Copy everything to the new array
    for(int i = 0; i < capacity; ++i)
      data[i] = old_data[i];

    capacity = new_cap;
    delete[] old_data;
  }

  // Add new element
  data[size++] = x;
}
```

- Given the following `node` definition

```
class list {
  public:

    struct node {
      int value;
      node* next;
    };

    node* find(int x);

  private:
    node* head;
}
```

Implement the `find` method searches through the list for the node with value x, returning a pointer to it if it exists, or `nullptr` if it does not exist.

```
list::node* list::find(int x)
{
```

► While it's not possible *in general* to make the linked-list `.at(i)` operation run in better than $O(n)$ time, it is possible to make `.at(i)` run in $O(1)$ time when used in this specific way:

```
for(int i = 0; i < sz; ++i)
    ... l.at(i) ...
```

When `.at(`i_1`)` is called, it saves both the index and `node*` into class member variables; when `.at(`i_2`)` is called later, we check to see if $i_2 = i_1 + 1$; if so, we use the saved `node*` to get to the next node.

Using the following `ordered_array` class, make the necessary changes to `at` so that it runs in $O(1)$ time when used as above.

```
class list {
  public:
    ...
    node* at(int i);
    ...
  private:
    node* hd;
    int sz;

    // at() has not been called yet, so we don't have a saved node*
    node* last_at_ptr = nullptr;
    int   last_at_ind = -1;
};

list::node* list::at(int i)
{
    // Your code here
```