Exam 2

October 19, 2015

- Do not open this exam until we instruct you to do so.
- This is a closed book, closed notes, closed electronics, closed neighbor, pencil and paper exam.
- This exam is designed to be completed in 50 minutes. However, you may take 120 minutes to work on the questions. You must immediately turn in the exam when we call for it.
- Write answers in the spaces provided. Indicate your final answer clearly. Cleanly erase or cross out any work you do not want graded. Show your work and explain your reasoning where appropriate; this will help to earn partial credit.
- Backpacks must be left at the front of the room.
- Electronic devices of any kind, including cell phones and calculators, must be completely turned off and stowed in your backpack.
- The exam must be written with a pencil.

1. Define: $f(n) \in O(g(n))$.

2. Draw a picture of the data structure generated by the following function:

```
struct node {
  int data;
  node *next;
  node(int d, node *n) : data(d), next(n) {}
};

int main()
{
  const size_t K = 4;
  node *A[K];
  for (int s=0;s<=2;s++)
    for (size_t i=0;i<K;i++) {
      size_t next = (i+1) % K; //mod wrap-around...
      if (s==0)
        A[i] = NULL;
      else if (s==1)
        A[i] = new node(next,A[i]);
      else
        A[i]->next = A[next];
    }
}
```

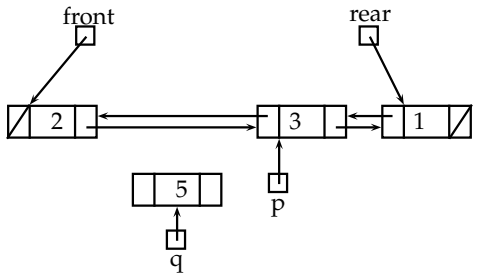
3. Write a complete *function* which takes a `list` of integers (`lst`), and removes all the integers which are negative, and returns a new `list` consisting of those numbers. For example, if `lst` is `[-3, 0, 8, -1, 2, -7, 3, -1, -2]` then after calling the function, `lst` is `[0, 8, 2, 3]` and the function returns the list `[-3, -1, -7, -1, -2]`. For your convenience, the `list` specific ation is attached to this exam.

4. Write a function that takes two nonempty singly-linked lists and performs the following side effect: The last node of the first list is linked to the first node of the second list. This need not be done recursively, though that would be acceptable.

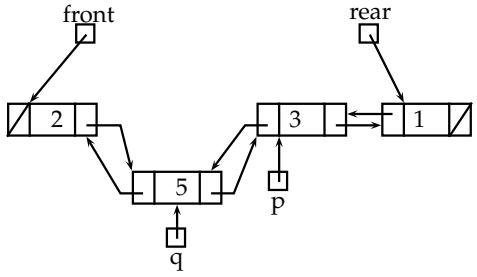
```
void connect(node * list1, node * list2);
```

5. Suppose we have a non-empty doubly-linked list **with no sentinels**, managed by `front` and `rear` pointers, as shown here. The variable `p` points to a node in the list; the variable `q` points to a single node outside the list. Write a code fragment that inserts `q` into the list before the one to which `p` points. Update *all* pointers as necessary.

   Before:



   After:



   Take very good care. `p` can point to any node in the list. **There are no sentinels.**

6. Assume the declarations:

```
struct node {
  node *prev;
  int data;
  node *next;
  node(int v) : prev(NULL), data(v), next(NULL) {}
};

node* R;
```

and the function:

```
node* build(int min, int max)
{
  if (min > max)
    return NULL;
  int mid = (min + max) / 2;
  node *answer = new node(mid);
  answer->prev = build(min,mid-1);
  answer->next = build(mid+1,max);
  return answer;
}
```

Draw a picture of the data structure generated by the expression:

```
R = build(1, 7)
```

Appendix: list.h

```
// list.h
// Alistair Campbell
// Fall 2014, Exam 2 version (private section removed)

class list
{
 public:
  list();
  // Create an empty doubly-linked list with front and rear sentinels,
  // current points to rear, and currentIndex = 0;

  ~list();
  // Delete all nodes (including the sentinels)

  list(const list & orig);
  // Make a (deep) copy of a list.
  // currentIndex and current can be set arbitrarily in the copy

  list & operator=(const List & rhs);
  // Support List assignment, with deep copy.  Similarly, currentIndex
  // and current are set arbitrarily.

  void add(const list_element & item,size_t index);
  // Insert a new element at position index
  // PRE:  List contains <v0, v1, ..., vindex, ...>
  //       0 <= index <= getSize()
  // POST: List contains <v0, v1, ..., item, vindex, ... >
  //       current points to new node containing item.
  // Guarantees constant time access to either end or vicinity of
  // currentIndex.

  void remove_at(size_t index);
  // Remove an element from the list.
  // Pre: 0 <= index < getSize()
  //      List contains <v0, v1, ..., vindex, v(index+1), ... >
  // Post: List contains <v0, v1, ..., v(index+1), ... >
  //       current points to v(index+1) (or the rear sentinel)
  // Guarantees constant time access to either end or vicinity of
  // currentIndex.

  void remove(const list_element & item);
  // Let i be the smallest integer where get(i)==item.
  // Post: same as for removeAt(i)
```

```
  // If no such i exists, no change to the list.

  size_t find(const list_element & item) const;
  // return smallest integer where get(i)==item,
  // or getSize() if no such integer

  list_element get(size_t index) const;
  // return the element at position i.
  // Guarantees constant time access to either end or vicinity of
  // currentIndex.

  size_t size() const;
  // return the number of elements of the list
  // runs in constant time

  void output(std::ostream & ostr) const;
  // output the list to ostr using format
  // <v0, v1, v2, ..., vn-1>

 private:
  // Not Relevant to Exam

};
```