Test 1: CPS 100

Owen Astrachan       Jeffrey Forbes

February 11, 2004

Name: ____________________________________________

Login: __________

Honor code acknowledgment (signature) ________________________________

<table>
<thead>
<tr>
<th></th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>14 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>18 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>28 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>84 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 16 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper #include header files. Assume that all the header files we’ve discussed are included in any code you write.
The declaration for linked list nodes on this test is:

```cpp
struct Node
{
    string info;
    Node * next;

    Node(const string& s, Node * ptr)
        : info(s), next(ptr)
    { }
};
```

Some common recurrences and their solutions.

- \( T(n) = T(n/2) + O(1) \quad O(\log n) \)
- \( T(n) = T(n/2) + O(n) \quad O(n) \)
- \( T(n) = 2T(n/2) + O(1) \quad O(n) \)
- \( T(n) = 2T(n/2) + O(n) \quad O(n \log n) \)
- \( T(n) = T(n-1) + O(1) \quad O(n) \)
- \( T(n) = T(n-1) + O(n) \quad O(n^2) \)

\[
1 + 2 + \cdots + N = \frac{N \times (N + 1)}{2}
\]

\[
2^{10} = 1024
\]
PROBLEM 1: (Ground Hog Day Dreams (14 points))

Part A (5 points)
The function hasDuplicates returns true if and only if its list parameter contains duplicate elements. It returns false for the list

("apple", "pear", "orange", "banana")

and true for the list

("apple", "pear", "orange", "banana", "apple")

Write a recurrence relation and its big-Oh solution for hasDuplicates where T(n) is the time for hasDuplicates to execute on an n-node list. Justify your answer briefly.

bool find(Node * list, const string& s)
{
    if (list == 0) return false;
    return list->info == s || find(list->next,s);
}

bool hasDuplicates(Node * list)
{
    if (list == 0) return false;
    if (find(list->next,list->info)) return true;
    return hasDuplicates(list->next);
}
Part B (4 points)
The call $\text{power}(x,n)$ returns the value $x^n$ where the function $\text{power}$ is shown below. Write a recurrence for this function where $T(n)$ is the time for $\text{power}(x,n)$ to execute. Explain why your recurrence is correct and what its solution is (in $O$-notation).

double power(double base, int expo)
// precondition: expo >= 0
// postcondition: returns base^expo (base to the power expo)
{
    if (0 == expo) return 1.0 // correct for zeroth power
    else
    {
        double semi = power(base,expo/2); // build answer from this
        if (expo % 2 == 0) { // even exponent
            return semi*semi;
        }
        else { // odd exponent
            return base*semi*semi;
        }
    }
}

Part C (5 points)
Write a recurrence for the function $\text{arrange}$ where $T(n)$ is the time for $\text{arrange}$ to execute on an n-element vector. Do not solve the recurrence, just write it.

void arrange(tvector<int>& vec)
{
    if (vec.size() != 0){
        sort(vec.begin(), vec.end());
        int mid = vec.size()/2;
        tvector<int> other;
        for(int k=0; k < mid; k++){
            other.push_back(vec[k]);
        }
        vec = other;
        arrange(vec);
    }
}
PROBLEM 2:  
(Grade Inflation (12 points))

In this problem you’ll use the standard Node declaration at the front of the test and the declarations and functions below for student records.

```cpp
struct Student
{
    string name;
    double gpa;
    Student(const string& s, double av)
       : name(s), gpa(av)
    {}
    Student()  // need default for tvector
       : name(""), gpa(0.0)
    {}
};

bool operator < (const Student& a, const Student& b)
{
    return a.gpa < b.gpa;
}
```

Part A (3 points)
The code below sorts a vector of Student objects by gpa.

```cpp
tvector<Student> list;
list.push_back(Student("ann",3.7));
list.push_back(Student("ben",3.8));
list.push_back(Student("dan",3.9));
list.push_back(Student("ken",3.7));

sort(list.begin(), list.end());
```

Change the comparison operator < so that students with the same gpa are sorted alphabetically (currently this is not the case).
Part B (9 points)

Write the function `honorRoll` that returns a linked-list of the names of students whose gpa is greater than or equal to parameter `honor`. For example, if `list` is the vector with four students initialized above (via calls to `push_back`) the call

```cpp
Node * summa = honorRoll(list,3.8);
```

should make `summa` point to the list ("ben","dan") since these students have a gpa greater than or equal to 3.8.

The order of the names stored in the returned linked list should be the same as the order in which the names occur in the vector.

```cpp
Node * honorRoll(const tvector<Student>& list, double honor)
{
```

```cpp
}
```
PROBLEM 3: (New Lists (18 points))

These question asks you to reason about the function `dostuff` below. A function that prints a list is shown. As an example, if list is "ant", the call `print(dostuff(list))` will print, on one line, `ant, ant`.

```cpp
void print(Node * list)
{
    while (list != 0){
        cout << list->info << " ";
        list = list->next;
    }
    cout << endl;
}

Node * dostuff(Node * list)
{
    if (list == 0) return 0;
    return new Node(list->info,
                    new Node(list->info,
                             dostuff(list->next)));
}
```

Part A (3 points)
What is printed by the call below if list represents the list ("ant", "bat", "dog").

```
print(dostuff(list));
```

Part B (3 points)
What is printed by the call below if list represents the list ("ant", "bat", "dog").

```
print(dostuff(dostuff(list)));
```

Part C (3 points)
What is the big-Oh complexity of `dostuff` when called with an n-element list. Justify your answer.
Part D (5 points)
The function bigstuff calls dostuff as shown below.

Node * bigstuff(Node * list, int n)
{
    if (n != 0){
        return bigstuff(dostuff(list),n-1);
    }
    return list;
}

What is printed by the call below if list represents the list ("ant", "bat", "dog").

    print(bigstuff(list,0));

What is printed by the call below if list represents the list ("ant", "bat", "dog").

    print(bigstuff(list,1));

What is printed by the call below if list represents the list ("ant", "bat", "dog").

    print(bigstuff(list,3));

Part E (4 points)
Write a big-Oh expression representing the number of nodes in the list returned by bigstuff(list,n) and the list passed to bigstuff has n nodes.
Justify your answer briefly.
PROBLEM 4: (Deque the Halls (28 points))

A Deque (pronounced deck, but an abbreviation for double-ended queue) is a data structure that permits insertion and deletion at both the front and the back in $O(1)$ time.

**Part A (3 points)**

Explain why one vector is not suitable for storing elements in a deque — that is, explain why insertion and deletion at both the front and back cannot be done in $O(1)$ time.

**Part B (3 points)**

Explain why one singly-linked list is not suitable for storing elements in a deque — that is, explain why insertion and deletion at both the front and the back cannot be done in $O(1)$ time.
Parts C,D
You’re given a declaration for a class `Deque` implemented using a doubly-linked list. The doubly-linked list does **not** have a header node.

class Deque
{
  public:
    Deque(); // initially empty deque
    int size() const; // number of elements in deque
    void push_front(const string& s); // add new element to front
    void push_back(const string& s); // add new element back
    string pop_front(); // remove and return first element
    string pop_back(); // remove and return last element

  private:
    struct Node
    {
      string info;
      Node * next;
      Node * prev;
      Node(const string& s, Node * after, Node * before)
        : info(s),
          next(after),
          prev(before)
        {
        }
    };
    Node * myFirst; // points to first node (NULL/0 if none)
    Node * myLast; // points to last node (NULL/0 if none)
    int mySize; // maintained as number of elements in deque
};

Here are the constructors and one of the methods for Deque.

Deque::Deque()
  : myFirst(0),
    myLast(0),
    mySize(0)
  // post: size of deque is 0
{
}

void Deque::push_front(const string& s)
  // post: s added to front of this deck, size incremented by 1
{
  if (myFirst == 0){
    myFirst = myLast = new Node(s,0,0);
  }
  else {
    myFirst->prev = new Node(s,myFirst,0);
    myFirst = myFirst->prev;
  }
  mySize++;
Part C (4 points)
Write the methods push_back and pop_back below. In writing pop_back assume that it is called only when it's precondition is satisfied.

```cpp
void Deque::push_back(const string& s) {
}
```

Part D (4 points)

```cpp
string Deque::pop_back() 
// pre: size() > 0 
// post: last element removed and returned, internal state updated 
//       appropriately 
{
```
Part E (6 points)
You’ve been asked to explain some observations and timings of the Deque class and to improve the runtime. Here’s the code whose performance you will examine and improve.

```cpp
while (deque.size() != 0) {
    string s = deque.pop_front();
}
```

Here are timings observed for the loop when executed on deques of different sizes. Since the pop_front method is supposed to execute in $O(1)$ time, these timings need to be explained.

<table>
<thead>
<tr>
<th>size of deque</th>
<th>runtime in microsec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>64</td>
</tr>
<tr>
<td>2000</td>
<td>256</td>
</tr>
<tr>
<td>4000</td>
<td>1,024</td>
</tr>
<tr>
<td>10000</td>
<td>6,400</td>
</tr>
</tbody>
</table>

You trace the performance and timings to this implementation of the Deque::size method.

```cpp
int Deque::size() const
{
    int count = 0;
    Node * temp = myFirst;
    while (temp != 0) {
        count++;
        temp = temp->next;
    }
    return count;
}
```

Explain the observations/timings and explain how to rewrite the Deque::size method so that the timings of the loop meet the expectations of an $O(1)$ pop_front function.
Part F (8 points) EXTRA CREDIT

This is extra credit, it is not required

Write reverse, a new method for Deque that reverses the order of elements in a Deque. After calling Deque::reverse, the first element becomes the last element (and vice versa) and a series of pop_front operations returns elements in the reverse order than before the reverse method is called.

Your code must run in $O(n)$ time for an $n$-element deque and use $O(1)$ additional storage/variables.

```cpp
void Deque::reverse()
{
}
```
Many colleges are arranged into athletic conferences. Consider the following declarations used to represent a doubly-linked list of schools (with school name), and a singly-linked list of conferences (the conference struct holds the conference name and a list of schools in that conference).

```c
struct School
{
    string name;
    int wins;
    int losses;
    School * prev;
    School * next;
    School(string newname, School *before, School * after)
        : name(newname), prev(before), next(after)
    {
    }
};

struct Conference
{
    string name;
    School * list;
    Conference * next;
    Conference(string newname, School * newlist, Conference * link)
        : name(newname), list(newlist), next(link)
    {
    }
};
```

(not all schools in each conference are shown below)
Part A (4 points)
Write the function `findSchool` whose header is given below. `findSchool` returns a pointer to the node containing `name` in a list of schools or NULL/0 if there is no such school. For example, using the diagram on the previous page `findSchool(getSchools(ncaa,"ACC"),"UNC")` returns a pointer to the node/school UNC and `findSchool(getSchools(ncaa,"ACC"), "Stanford")` returns 0 since Stanford is not in the ACC conference.

The function `getSchools` shown below should not be called in writing `findSchool`.

```c
School * getSchools(Conference * confList, const string& name)
// post: returns a pointer to the list Schools in a specific conference
{
    while (confList != 0){
        if (confList->name == name) return confList->list;
        confList = confList->next;
    }
    return 0;
}
```

```c
School * findSchool(School * list, const string& name)
// post: returns a pointer to the node containing name, or 0 if none
{
```
Part B (8 points) Write function changeConf, to move a team from one conference to another. For example:

    changeConf(ncaa, "VaTech", "Big East", "ACC");

would remove VaTech from the Big East conference and move it to the ACC conference. In writing changeConf you may call functions findSchool and getSchools specified above. Assume they work as specified. You can also call removeNode shown below which removes a node from a doubly-linked list.

```c
void removeNode(Node * dead)
// pre: dead != 0
// post: dead removed from doubly-linked list
{
    if (dead->prev != 0)
    {
        dead->prev->next = dead->next;
    }
    if (dead->next != 0)
    {
        dead->next->prev = dead->prev;
    }
}
```

```c
void changeConf(Conference* ncaa,
            const string& team,
            const string& fromConf, const string& toConf)
{
}
```