University of Illinois at Urbana-Champaign
Department of Computer Science

Second Examination

CS 225 Data Structures and Software Principles
Summer 2005
3:00pm – 4:15pm Monday, July 18

Name: SOLUTIONS
NetID:
Lab Section (Day/Time):

- This is a **closed book** and **closed notes** exam. No electronic aids are allowed, either.
- You should have 6 sheets total (the cover sheet, plus numbered pages 1-11). The last sheet is scratch paper; you may detach it while taking the exam, but must turn it in with the exam when you leave. can use this sheet as reference while taking the exam.
- Unless otherwise stated in a problem, assume the best possible design of a particular implementation is being used.
- Unless the problem specifically says otherwise, (1) assume the code compiles, and thus any compiler error is an exam typo (though hopefully there are not any typos), and (2) assume you are NOT allowed to write any helper methods to help solve the problem, nor are you allowed to use additional arrays, lists, or other collection data structures unless we have said you can.

Problem	Points	Score	Grader
1	15		
2	15		
3	15		
4	15		
5	15		
6	15		
Total	90		

1. **[Simple List Code – 15 points].**

   You are given the `ListNode` class shown on page 8 of the exam. You want to write a function `insertAsSecondToLast` that has two parameters, and returns nothing. The first parameter, `head`, is of type reference-to-pointer-to-`ListNode`, and will point to a linked list made up of objects of type `ListNode`. If the list is empty, then `head` will point to `NULL`; otherwise, the `prev` of the first node and the `next` of the last node will both point to `NULL`. The second parameter, `insElem`, is of type `int`. You want to insert this integer as the *second-to-last* value of the list – i.e. right before the last node. If the list is empty, then insert this value as the first value instead.

   Whatever linked list this results in, the `head` parameter should be pointing to the first node of that list when you are done. You must physically move nodes – you cannot reassign the `element` variables in the nodes.

```
void insertAsSecondToLast(ListNode * & head, int insElem) {
   // your code goes here

   if (head == NULL)
      head = new ListNode(insElem);
   else
   {
      ListNode* temp = head;
      while (temp->next != NULL)
         temp = temp->next;
      ListNode* insertee = new ListNode(insElem);
      insertee->next = temp;
      insertee->prev = temp->prev;
      if (temp->prev != NULL)
         temp->prev->next = insertee;
      else
         head = insertee;
      temp->prev = insertee;
   }
}
```

2. **[Skiplists – 15 points].**

   You have a skiplist class as shown on page 9 of the exam. You want to write a member function `levelsDouble` that will have no parameters and will return a `bool` value. If there is exactly one node on the top level (`maxLevel`), and each level below that level, has exactly double the number of nodes that were on the previous level, this function should return `true`. Otherwise, this function should return `false`.

```
bool SkipList::levelsDouble() {
   // your code goes here

   if ((head[maxLevel] == NULL) || ((head[maxLevel]->ptrArray)[maxLevel] != NULL))
      return false;
   else {

      int level = maxLevel - 1;
      int numOnPrevLevel = 1;

      while (level >= 0) {

         SkipNode* temp = head[level];
         int numNodes = 0;

         while (temp != NULL) {
            numNodes++;
            temp = (temp->ptrArray)[level];
         }

         if (numNodes != 2 * numOnPrevLevel)
            return false;
         else {
            level--;
            numOnPrevLevel = numNodes;
         }
      }
      return true;
   }
}
```

3. **[Sum Of Removal – 15 points].**

   You have the `ListNode` class shown on page 8 of the exam. You want to write a member function `sumOfRemoval` that will have one parameter and will return nothing. The parameter is of type reference-to-pointer-to-`ListNode`, and will point to the first node of a doubly-linked list. If the list is empty, the pointer points to `NULL`; otherwise, the first node's `prev` and the last node's `next` are both `NULL`.

   Your task is to remove all nodes whose values are *greater than* `100`, and then to add a new node at the end of the list, whose value is equal to the sum of all the values in all the nodes you removed. For example, if the parameter list had been
   `4->502->10->12->7->33->5->821->11->103->7->NULL`, then your resultant list should be
   `4->10->12->7->33->5->11->7->1426->NULL`

   Whatever linked list this results in, the `head` parameter should be pointing to the first node of that list when you are done. You must physically move nodes – you cannot reassign the `element` variables in the nodes.

```
void SumOfRemoval(ListNode * & head) {
   // your code goes here

   if (head == NULL)
      head = new ListNode(0);
   else
   {
      int total = 0;
      ListNode* trav = head;

      while (trav != NULL) {
         if (trav->element > 100) {

            total = total + trav->element;
            ListNode* toDelete = trav;
            trav = trav->next;

            if (toDelete->prev != NULL)
               toDelete->prev->next = toDelete->next;
            else
               head = head->next;

            if (toDelete->next != NULL)
               toDelete->next->prev = toDelete->prev;

            delete toDelete;
         }
         else
            trav = trav->next;
      }

      // ...continued on next page...
```

(continued)

```
      // you could also have kept careful track of the tail above
      ListNode* tail = head;
      if (tail == NULL)
         head = new ListNode(total);
      else {
         while (tail->next != NULL)
            tail = tail->next;
         tail->next = new ListNode(total);
         tail->next->prev = tail;
      }
   }
}
```

4. **[Ordered Trees – 15 points].**

   You are given the `OrderedNode` class on page 8 of the exam. You want to write a function `insertAsFirst` that has three parameters. The first two parameters are integers. The third parameter is of type pointer-to-`OrderedNode`, and points to a binary tree representation of an ordered tree (i.e. the "first child/next sibling" representation). (If the tree is empty, the pointer would point to `NULL`.) The function should insert the second integer into the tree, as the first child of whatever node the first integer is stored in. (That node's existing first child then becomes the second child, the second child becomes the third child, and so on.) If the first integer is NOT in the tree, then do not alter the tree at all.

```
void insertAsFirst(int parent, int insChild, OrderedNode* ptr) {
   // your code goes here

   if (ptr != NULL) {

      if (ptr->element == parent) {

         OrderedNode* temp = ptr->firstChild;
         ptr->firstChild = new OrderedNode(insChild);
         ptr->firstChild->nextSibling = temp;
      }
      else {

         insertAsFirst(parent, insChild, ptr->firstChild);
         insertAsFirst(parent, insChild, ptr->nextSibling);
      }
   }
}
```

5. **[Unordered Trees – 15 points].**

   You are given the `TreeNode` class on page 8 of the exam. You want to write a function `equivalent` that has two parameters, both of type pointer-to-`TreeNode`. Each of these pointers, points to a binary tree. You want to find out of these two trees are equivalent unordered trees. That is, you want to find out if ignoring the order of the child subtrees makes these two trees equivalent. If switching the order of the left and right *subtrees* at various nodes of the first tree, could lead to the second tree, you want to return `true`; otherwise, you want to return `false`.

```
bool equivalent(TreeNode* first, TreeNode* second) {
   // your code goes here

   if ((first == NULL) && (second == NULL))
      return true;
   else if ((first == NULL) || (second == NULL))
      return false;
   else if ((equivalent(first->left, second->left) &&
               equivalent(first->right, second->right))  ||
            (equivalent(first->left, second->right) &&
               equivalent(first->right, second->left)))
      return true;
   else
      return false;
}
```

6. **[Finding Depth – 15 points].**

   You are given the `TreeNode` class on page 8 of the exam. You want to write a function `findDepth` that has two parameters. The first parameter is of type pointer-to-`TreeNode`, and points to a binary search tree (which could be empty). The second parameter will be an integer. You want to return the depth of the node that contains that parameter integer in the parameter binary search tree. (Remember that the depth of the root node of a tree is `0`, not `1`.) If the parameter integer is not in the parameter binary search tree at all, return `-1`.

```
int findDepth(TreeNode* ptr, int value) {
   // your code goes here

   if (ptr == NULL)
      return -1;
   else if (ptr->element == value)
      return 0;
   else if (value < ptr->element)
   {
      int result = findDepth(ptr->left, value);
      if (result == -1)
         return -1;
      else
         return 1 + result;
   }
   else  // value > ptr->element
   {
      int result = findDepth(ptr->right, value);
      if (result == -1)
         return -1;
      else
         return 1 + result;
   }
}
```

```
class ListNode {  // needed for problems 1 and 3
public:
   int element;
   ListNode* next;
   ListNode* prev;

   ListNode(int value) {element = value; next = NULL; prev = NULL;}
};

class OrderedNode {  // needed for problem 4
public:
   int element;
   OrderedNode* firstChild;
   OrderedNode* nextSibling;

   OrderedNode(int value) {element = value; firstChild = NULL; nextSibling = NULL;}
};

class TreeNode {   // needed for problems 5 and 6
public:
   int element;
   TreeNode* left;
   TreeNode* right;

   TreeNode(int value) {element = value; left = NULL; right = NULL; }
};
```

```
template <typename Etype>
class Array  {    // needed for the SkipList class below

   // Here are the member function declarations for the Array class;
   // we've left the declarations for the variables, iterators and iterator
   // support functions (begin(), end(), etc.) out; you don't need them.
   Array();   // size 0 array, indiced 0 through -1
   Array(int low, int high);   // indices low through high
   Array(Array<Etype> const & origVal);  // copy constructor
   ~Array();                             // destructor
   Array<Etype> const & operator=(Array<Etype> const & origVal);//assignment op
   Etype const & operator[](int index) const;   // accesses cell at param index

   Etype & operator[](int index);   // accesses cell at param index
   void initialize(Etype const & initElement); // inits all cells to param
   void setBounds(int theLow, int theHigh);  // changes bounds of array,
   int size() const;   // returns number of cells in array
   int lower() const;  // returns lowest index
   int upper() const;  // returns upper index
};
```

```
class SkipList {        // needed for problem 2
public:
   // various member functions here
private:
   class SkipNode {
      int element;              // value stored in this node

      // array of "next pointers" for all levels this node is a part of;
      // ptrArray.lower() is 0 and ptrArray.upper() is the node's max level
      Array < SkipNode* > ptrArray;
   };

   // array of pointers to first node on each level; head.lower() is 0
   // and head.upper() is the skiplists's max level
   Array < SkipNode* > head;
   int maxLevel;          // largest level any node is allowed to have
   int numElements;       // number of nodes in list
};
```

(scratch paper, page 1)

(scratch paper, page 2)