CS61B (Clancy)
Spring 1996
Exam 3, solutions, and grading standards.

April 12, 1996
Exam 3

Read and fill in this page now.
Do NOT turn the page until you are told to do so.

Your name:
Your login name:
Your lab section day and time:
Your lab t.a.:
Name of the person sitting to your left:
Name of the person sitting to your right:

Problem 0 __
Problem 1 __ __ __
Problem 2 __
Problem 3 __
Problem 4 __
Problem 5 __ __

Total: __/20

This is an open-book test. You have approximately fifty minutes to complete it. You may consult any books, notes, or other paper-based inanimate objects available to you. To avoid confusion, read the problems carefully. If you find it hard to understand a problem, ask us to explain it. If you have a question during the test, please come to the front or the side of the room to ask it.

This exam comprises 10% of the points on which your final grade will be based. Partial credit will be given for wrong answers. Your exam should contain six problems (numbered O through 5) on nine pages. Please write your answers in the spaces provided in the test.

Anywhere you are directed to write a function, your solution may include auxiliary functions. You need not rewrite a function that appears in Deitel and Deitel, in Carrano, or in any of the CS 61B handouts; merely cite the page in the textbook or the handout in which the function appears.
A few students are taking this exam next week. Do not discuss this exam with them, or post any news about the exam until next Wednesday.

Relax--this exam is not worth having heart failure about.

Problem 0 (1 point, 1 minute)

Put your login name on each page. Also make sure you have provided the information requested on the first page.

Problem 1 (3 points, 8 minutes)

A debugged version of the code for heap verification and insertion used in lab assignment 11 appears below. Buestions in parts a and b refer to this code.

heaps.h

```
class Heap (
public:
 Heap (ifstream&);
 bool IsHeap ();
 void Insert (int);
private:
 int size;
 Vector<int> values;
};
```

heaps.cc

```
Heap::Heap (ifstream& valuesIn):
 size(0), values(some positive integer) {
   int value;
   while (valuesIn >> value) {
     values[size] = value;
     size++;
   };
```

```
   assert (IsHeap ());
}

bool Heap::IsHeap () {
 for (int k=O; k<size/2; k++) {
  if (values[k] < values[2*k+ll) {
      return false;
  }
  if (k < (size-1)/2 && values[k] < values[2*k+21] {
     return false;
  }
 }
 return true;
}

void Heap::Insert (int newValue) {
  values[size] = newValue;
  size++;
  for (int k=size-l; k>0; k = (k-1)/2) {
    if (values[k] < values[(k-1)/2]) {
        break;
    }
    int temp = values[k];
    values[k] = values[(k-1)/2];
    values[(k-1)/2] = temp;
  }
}
```

Part a

Suppose that a heap is declared with the statements

```
     ifstream fileIn ("data");
     Heap h (fileIn);
```

and the data file contains the values

6
3
5
1
2
4

If the values as input form a max heap--that is, the call to IsHeap at the end of the Heap constructor returns true--draw the resulting heap. If the values as input do not form a max heap, display the value of k at which IsHeap returns false.

Part b

Draw the heap that results from starting with an empty heap and using the Insert function to insert the values in the opposite order, that is,
4
2
1
5
3
6
Show your work for full credit.

Part c

In the original undebugged version of IsHeap, the loop body was as follows:

```
if (values[k] < values[2*k+l] || values[k] < values[2*k+2]) {
  return false;
}
```

This code correctly checks that some arrangements of values in the values array form a max heap. Fill in the blank in the statement below to describe, as completely as possible, all value arrangements for which the undebugged IsHeap function correctly determines whether or not the values form a heap.

IsHeap correctly determines whether the elements of values fom a max heap when value.size ( ) is _____.

Problem 2 (6 points, 15 minutes)

Describe the implementation of a generalized vector class called GenVector. Its constructor takes as argument a list of the integers to be used

as legal subscripts for the GenVector object. For example, passing the constructor a list containing the integers -5, 42, and 363 should set up a three-element GenVector.

Your implementation should allow fast execution of the following operations for large subscript sets:

- retrieving the value associated with a given index (the [ ] operation for regular Vectors);

- replacing the value associated with a given index by a new value (the operation analogous to assignment with elements of regular Vectors)

- detection of whether a given index value is a legal subscript for the GenVector object.

It should also not use an extravagant amount of memory when the subscript set is small.

Your description should specify what data structure is used to implement the generalized vector, what it contains and, if your data structure includes an array, how big the array is. You should also explain how the expression a[k] would be evaluated for a given generalized vector a and integer k. Your description should be in terms suitable for another CS 61B student to understand how it would be translated into code.

Problem 3 (5 points, 14 minutes)

Suppose that the environment for homework assignment 9+10 is represented as a List of objects of class Swimmer*. Swimmers have the following properties:

- each Swimmer object represents either a fish or a shark;

- Swimmer member functions include a function IsFish that returns true if the Swimmer object represents a fish and fals otherwise.

Given below is a class definition for Lists of Swimmer* objects.

```
struct ListNode (
  Swimmer* info;
  ListNode* next;
};
class List (
```

```
public:
  // (public member functions are defined here)
  // Initialize an iterator for fish or sharks.
  void InitIterator (bool);
  / / See if more of the requested type of swimmer remain.
  bool MoreRemain ( );
  // Get the next of the requested type of swimmer.
  Swimmer* Next ( );
  private:
     int size;
     ListNode* head;          // pointer to  first thing
in the list
  };
```

You are to implement the three iterator functions for Swimmers. InitIterator takes a boolean argument; a true value specifies that the iteration should proceed through only the sharks in the environment, while a false value specifies a fish iterator. Next returns a pointer to the next of the specified type of Swimmer, and MoreRemain says whether more of the specified type of Swimmer remain to be returned. Thus step 1 of the Wa-Tor simulation algorithm would be done as follows:

```
List environment;
   ...
environment.InitIterator(false) ; // iterate through fish
while (MoreRemain ( )) {
  Swimmer* fishPtr = Next ( );
  // determine if the fish moves or breeds
}
```

and step 2 would be

```
environment.InitIterator(true); // iterate through sharks
while (MoreRemain ( )) {
  Swimmer* sharkPtr = Next ( );
  // determine if the shark eats, moves, breeds, or dies
     ...
}
```

Your solution will be penalized for unnecessary inefficiency. You are not to call any of the existing List member functions, though you may define your own private auxiliary functions. Assume that the fish and sharks are

ordered appropriately in the list, i.e. in increasing order by position (row by row and from left to right within a row).

Put your solution--the code you would add to the lists.cc file--below. We will assume that undefined variables your code includes will be definedas private data members and that any auxiliary functions you define will be declared as private member functions unless you specify otherwise.

```
void List::InitIterator (bool sharkIterating) {




}
bool List::MoreRemain ( ) {

}
Swimmer* List::Next ( ) {


}
```

Problem 4 (2 points, 5 minutes)

Recall the DuplicateElements function from the first midterm exam. The code appears below.

```
void List::DuplicateElements () {
   for (ListNode* ptr=head; ptr!=0;                    ) {
     ListNode* ptr2
       = new ListNode (ptr->info, ptr->next);
     ptr->next = ptr2;
     ptr = ptr->next->next;
     size++;
   }
}
```

Adding this function to the template List class you defined in lab assignment 10 requires that a line be added to the List class declaration--

```
template <class InfoType>
class List {
public:
      ...
   void DuplicateElements ();
      ...
};
```

and that the function definition itself be modified. Indicate clearly what changes to the function definition are necessary to add this function to the template List class from lab assignment 10.

Problem 5 (3 points, 7 minutes)

Given below is an untemplated version of code from lab assignment 11 that implements a binary search tree.

```
class BinSrchTree {
public:
   BinSrchTree ();
   void Insert (ItemType);

private:
   TreeNode* tree; // 0 for an empty tree; points to the root of
a nonempty tree
   TreeNode* InsertHelper (ItemType, TreeNode*);
};

void BinSrchTree::Insert (ItemType x) {
  tree = InsertHelper (x, tree);
}
TreeNode* BinSrchTree::InsertHelper (ItemType x, TreeNode*
treePtr) {
  if (treePtr == 0) {
      return new TreeNode (x);
   } else if (x <= treePtr->info) {
      treePtr->left = InsertHelper (x, treePtr->left);
      return treePtr;
   } else {
      treePtr->right = InsertHelper (x, treePtr->right);
```

```
            return treePtr;
        }
    }
```

Comparisons are highlighted in boldface.

Part a

Draw the tree that results from inserting the values 1, 3, 4, 2 in sequence into an initially empty tree.

Part b

How many total comparisons are made when inserting the values 1, 3, 4, 2 in sequence into an initially empty tree? Show your work for full credit.

Exam information and SOLUTIONS

225 students took the exam. Scores ranged from 1 to 20, with a median of 13 and an average of 12.8. Score totals were rounded up where appropriate; e.g. a total score of 13 1/2 was counted as 14. There were 74 scores between 16 and 20, 80 between 11 and 15, 56 between 6 and 10, and 16 between 1 and 5. (Were you to receive a grade of 16 on all your midterm exams, 48 on the final exam, plus good grades on homework and lab, you would receive an A-; similarly, a test grade of 11 may be projected to a B-.)

There were four versions of the exam, A, B, C, and D. (The version indicator appears at the bottom of the first page.) Versions A and C were identical except for the order of the problems. Versions B and D were also identical except for the order of the problems.

If you think we made a mistake in grading your exam, describe the mistake in writing and hand the description with the exam to your lab t.a. or to Mike Clancy. We will regrade the entire exam.

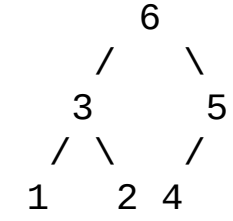
Solutions and grading standards for versions A and B
Problem 0 (1 point)

As usual, deductions for this problem were for omitting your

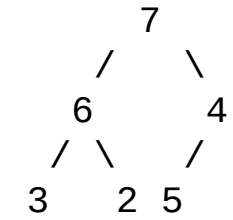
login name on pages or for omitting information on the front page.

Problem 1 (3 points)

Part a was to determine if specified values, when loaded directly into an array (i.e. not using the Insert function), formed a heap. In version A, the values 6, 3, 5, 1, 2, 4 form the heap



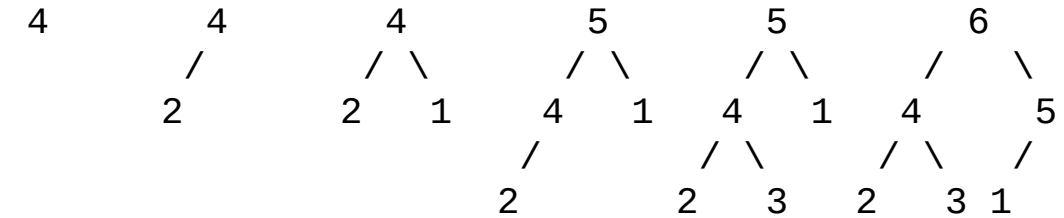
In version B, the values 7, 6, 4, 3, 2, 5 do not form a heap; the IsHeap function detects the non-heapness of the structure



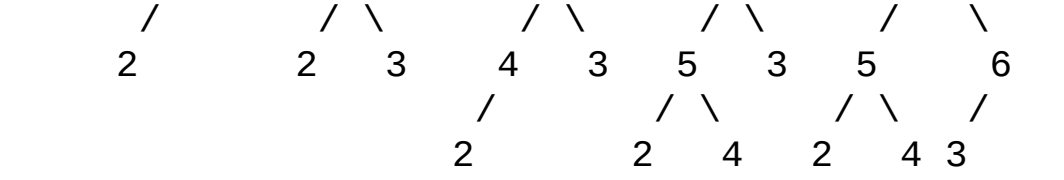
at k=2.

Part b involved repeated insertion of the values (in the opposite order from part a). The six insertions result in the following heaps:

Version A



Version B



In part c, you had to identify what arrangements of the array were correctly processed by the buggy IsHeap from lab assignment 11 (in version A) or incorrectly processed by IsHeap (in version B). The

bug, as you may recall, was that element size of the array (i.e. the element immediately following the last element of the heap) was checked in some circumstances. This happened when the rightmost heap item in the next-to-last level had only one child, which happens when there are an even number of items in the heap. Thus the answer in version A was the word "odd", and the answer in version B was "even".

Parts a, b, and c were each worth 1 point. Generally, the grading was all or none, except for clearly careless errors (which lost 1/2 point). In part b, you received no credit if you merely gave the final heap with no work shown. You received 1/2 point for the right answer and a few temporary values, but no intermediate heaps drawn.

Problem 2 (6 points)

This problem was the same on all versions. A variety of solutions received full credit; here are representative solutions.

*A hash table of size depending on the number of subscripts, in which collisions are resolved by chaining, and each element of the chain is a subscript/value pair. Steps for the evaluation of the expression a[k] would be as follows. First k is hashed and the corresponding chain searched for the node that has k as a subscript. If no such node exists, then k is an illegal subscript. Otherwise the associated value is returned. If there are N subscripts, a table with N/2 cells should be sufficient. (The Vector for the hash table need not be of constant size; it can be resized once the length of the list is determined.)

*A balanced binary search tree, in which each node contains a subscript and the corresponding value along with the two tree pointers, and in which nodes are ordered by subscript values. The expression a[k] is evaluated by searching (in the standard way) for the node with subscript value k; if no such node exists, k is an illegal subscript, otherwise the associated value is returned. One can ensure that the tree is balanced, since all the elements are known in advance (code similar to the Reorg function from lab assignment 11 could be used to create the tree).

*An array of subscript/value pairs, sorted by subscript values. As with the hash table, the array can be resized to contain exactly as many elements as subscripts in the GenVector constructor argument list. Evaluating a[k] involves searching for the pair containing k as a subscript using binary search. If such a pair

exists, the associated value is returned; if not, k is an illegal subscript.

Some solutions introduced an extra array that contained only values; the contents of the hash table, binary search tree, or array were pairs consisting of a "virtual subscript" (an element of the GenVector constructor argument list) and an actual subscript into the value array.

You needed to state clearly that your structure was storing *both subscrtipts and associated values* to receive any more than 1 point on this problem, no matter how elegant your description was otherwise. Many students seemed to be confused about the role of the elements of the GenVector constructor list (perhaps because they didn't understand that "index" and "subscript" are synonyms?), and described structures to store and retrieve only the subscripts; these solutions received O points out of 6.

Imperfect solutions that clearly described the storage both of subscripts and of associated values lost points as follows.

*(solutions using hashing) Allocating a constant size hash table lost 1 point; neglecting to say that collisions were handled via chaining, implying the need for a perfect hash function, lost 2 points.

*(solutions using a binary search tree) Neglecting to say that the tree was balanced lost 1 point.

*A solution using linear search lost 3 points.

*An array-based solution involving an array of size equal to the largest subscript value minus the smallest subscript value + 1 lost 3 points. (It would use an extravagant amount of storage to represent the GenVector with the two subscripts +/-1,000,000,000.)

Problem 3 (5 points)

This problem was essentially the same in both versions. The only difference was in the argument to the InitIterator constructor. (In version A, the argument is true when we're looking for sharks; in version B, it's true when we're looking for fish.)

Again, there were a variety of correct solutions. One version A

solution, based on iterators presented in class and devised in lab assignment 10, is as follows.

```
void List::InitIterator (bool sharkIterating) {
  savedPtr = head;
  seekingSharks = sharkIterating;
  Locate ( );
}

bool List::MoreRemain ( ) {
  return savedPtr != 0;
}

Swimmer* List::Next ( ) {
  ListNode* temp = aavedPtr;
  Locate ( );
  return temp->info;
}

void List::Locate ( ) {
  if (seekingSharks)  {
     while (aavedPtr != O && !savedPtr->info->IsFish ()) {
        savedPtr = savedPtr->next;
     }
  } else {
     while (savedPtr != O && savedPtr->info->IsFish ()) {
          savedPtr = savedPtr->next;
     }
  }
}
```

The invariant relation between iterator function calls is that savedPtr points to the node corresponding to the next Swimmer* to return. The seekingSharks variable is not really necessary, since the Locate function need merely locate a Swimmer that's the same kind as the one that was most recently found.

Some students coded recursive equivalents (or nonequivalents) for the while loops. Another approach involved having InitIterator construct the list of fish or sharks to return, then coding MoreRemain and Next exactly as in lab assignment 10.

A more complicated solution involved having MoreRemain do the

locating. The invariant for this solution is that savedPtr points somewhere after the node most recently returned, but not necessarily at the next node to return; a call to MoreRemain or Next should advance it appropriately. Note that calls to MoreRemain and Next need not alternate, the way they do in the examples; one might know for other reasons that there are at least a certain number of elements in the list, and make that many calls to Next without any calls to MoreRemain.

Points were awarded as follows: 2 points for advancing savedPtr correctly; 1 point for each of the rest of InitIterator, MoreRemain, and Next. Common bugs were missing the first element of the list or forgetting to move to the first relevant list item in InitIterator, forgetting in Next to increment savedPtr before returning, checking beyond savedPtr in MoreRemain (thus coding "two more remain"), and starting over at the beginning of the list each time.

Problem 4 (2 points)

Version A involved converting the DuplicateElements function from the first exam to be included in a template List class; version B involved doing the same for the RemoveAlternateElements function from the first exam. All references to the List and ListNode types had to be changed to List<lnfoType> and ListNode<lnfoType>, and the line "template <class InfoType>" had to be added before each function. The changes appear in boldface below.

```
template <class InfoType>
void List<InfoType>::Duplicate elements () {
  for (ListNode<InfoType>* ptr=head; ptr!=0; ) {
     ListNode<InfoTyps>* ptr2
       = new ListNodeInfoType> (ptr->info, ptr->next);
           ... // the rest is unchanged

template <class InfoType>
void List<InfoType>::RemoveAlternateElements () {
  if (head != 0 && head->next != 0) {
     for (ListNode<InfoType>* ptr=head;
         ptr!=0 && ptr->next!=0; ptr=ptr->next) {
        ListNode<InfoType>* ptr2 = ptr->next;
        ... // the rest is unchanged
```

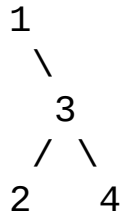
In each problem, you earned 1 point for making the changes to

ListNode and 1 more point for making the other changes. (A very common error was to omit the changes to ListNode.) You were expected to know that struct ListNode was declared in the lists.h code, and to use it rather than inventing your own type for list nodes. Some students erroneously changed ListNode to InfoType, losing 1 point.

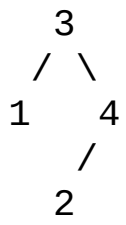
Problem 5 (3 points)

This problem involved creating a binary search tree and counting the comparisons needed to do so. The versions differed only in the order in which values were to be inserted into the binary search tree: in version A, it was 1, 3, 4, 2, and in version B, it was 3, 1, 4, 2. The resulting trees are

version A



version B



Part a, creating the correct tree, was worth 1 point. Your answer to part b was evaluated using whatever tree you provided for part a. The comparison count is shown below.

version A

1 comparison to insert 1
3 comparisons to insert 3
5 comparisons to insert 4
5 comparisons to insert 2

total 14

version B

1 comparison to insert 3
3 comparisons to insert 1
3 comparisons to insert 4
5 comparisons to insert 2

total 12

Part b was worth 2 points. You lost 1/2 point for an addition error and 1 point for each other error, e.g forgetting the last comparison for each insertion.
