Terence Henriod

Test 3 Corrections

CS 302: Data Structures

December 4, 2013

|  |  |
| --- | --- |
| The Question: | Define the **scope** of a variable. |
|  |  |
| Correct Answer: | Put simply, it is the frame of reference where the variable can be referred to by name. |
|  |  |
| Supporting Quote from Text: | “The scope of a variable is the part of the program where the variable may be used… The first rule of scope you should learn is that a variable cannot be used in any part of the program before the definition.”  “**NOTE:** When a program is running and it enters the section of code that constitutes a variable’s scope, it is said that the variable *comes into scope*. This simply means the variable is now visible and the program may reference it. Likewise, when a variable *leaves scope*, it may no longer be used.”  “When a value is stored in a static member variable, it is not stored in an instance of the class… You can think of static member variables and static member functions as belonging to the class instead of to an instance of the class.” |
|  |  |
| Text and Page Reference: | Starting Out With C++: From Control Structures through Objects, by Gaddis; pages 59, 213, and 800, respectively. (The text for this class does not seem to have a suitable discussion about scope) I was also told this in the office of Frederick C Harris, Jr. |

|  |  |
| --- | --- |
| The Question: | What is a perfect hash function and when is it possible to create one? |
|  |  |
| Correct Answer: | A perfect hash is a function that will map all objects to unique values (indices), that is, without collision.  It is possible to create one when much is known about the data to be hashed, such as the size and makeup (values) of the data set. |
|  |  |
| Supporting Quote from Text: | “Ideally, you want the hash function to map each search key x into a unique integer i. The hash function in the ideal situation is called a **perfect hash function**. In fact, it is possible to construct perfect hash functions if you know all of the possible search keys that *actually* occur in the dictionary.” |
|  |  |
| Text and Page Reference: | Data Abstraction and Problem Solving with C++: Walls and Mirrors, by Carrano and Henry, page 546 |

|  |  |
| --- | --- |
| The Question: | Integers 10, 23, 235, 430, 21, 17, 54, and 51 are inserted into a hash table whose indices are 0..9. The hash function is h(key)=key%10 and collisions are resolved with linear probing. Show the configuration of the hash table after insertion. |
|  |  |
| Correct Answer: | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 10 | 430 | 21 | 23 | 54 | 235 | **51** | 17 |  |  | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
|  |  |
| Supporting Quote from Text: | No quote from the text needed, I simply forgot to insert the 51, so in marking my solution incorrect, the instructor circled the 51 in the problem statement and then drew an arrow directing the 51 to the appropriate slot. |
|  |  |
| Text and Page Reference: | CS302 Test 3, page 4 |

|  |  |
| --- | --- |
| The Question: | Write the Insert function for this maxheap of integers. |
|  |  |
| Correct Answer: | void Heap::insert( const int newItem ) throw( logic\_error )  {  int child = size; // size will always conceivable be >= 0  int parent = ( ( size – 1 ) / 2 ); // truncates to 0 when size == 0  int temp;  if( size >= maxSize )  {  throw logic\_error( “Can’t insert into a full heap.” );  }  else  {  table[ size ] = newItem;  size++;  while( data[ parent ] < data[ child ] )  {  Temp = data[ parent ];  Data[ parent ] = data[ child ];  Data[ child ] = temp;  child = parent;  int parent = ( ( child – 1 ) / 2 ); //truncates to 0 when child==0  }  }  } |
|  |  |
| Supporting Quote from Text: | A snippet of psuedocode from the book. My code inverts their logic to produce the same result:  “  // insert newData into the bottom of the tree  items[itemCount] = newData  // Trickle new item up to the appropriate spot in the tree  newDataIndex = itemCount  inPlace = false  while ( (newDataIndex >= 0) and !inPlace)  {  parentIndex = ( newDataIndex – 1) / 2  if( items[newDataIndex] < items[parentIndex]) // my code uses the opposite of this check to  // indicate if percolation should continue  {  inPlace = true;  }  else  {  Swap items[newDataIndex] and items[parentIndex]  newDataIndex = parentIndex  }  }  itemCount++  “ |
|  |  |
| Text and Page Reference: | Data Abstraction and Problem Solving with C++: Walls and Mirrors, by Carrano and Henry, page 511 |

|  |  |
| --- | --- |
| The Question: | Write the Remove function for a Binary Search Tree (BST) of integers. |
|  |  |
| Correct Answer: | bool BSTree::remove( const int key ) throw( logic\_error )  {  bool result = false;  if( root != NULL )  {  Result = remove\_sub( root, key );  }  else  {  throw logic\_error( “Can’t remove from empty tree.” );  }  return result;  }  bool BSTree::remove\_sub( Node\*& current, const int key )  {  Node\* temp = NULL;  bool result = false;  if( current != NULL )  {  if( current->data < key )  {  result = remove\_sub( current->left, key ):  }  else if( current->data > key )  {  result = remove\_sub( current->right, key ):  }  else  {  // the current node’s data is the same as the key if we end up here  result = true; // a removal will be made in one of the following . // ways  if( ( current->left == NULL ) && ( current->right == NULL ))  {  delete current;  current = NULL;  }  else if( ( current->left == NULL ) || ( current->right == NULL ))  {  temp == current;  if( current->left == NULL )  {  Current = current->right;  }  else  {  current = current->left;  }  delete temp;  temp = NULL;  }  else  {  temp = current->left;  while( temp->right != NULL )  {  temp = temp->right;  }  current->data = temp->data;  remove\_sub( current->left, key ); // no need to store this result  // we know removal will occur  }  }  }  return result;  } |
|  |  |
| Supporting Quote from Text: | A snippet of psuedocode from the book. My code uses similar logic to produce the same result, in the case that we are removing, set a flag to say the node was found and will be removed without regard to the removal case:  “  else if (subTreePtr->getItem() == target)  {  // Item is in the root of some subtree  subTreePtr = removeNode(subTreePtr) // remove the item  success = true // does not depend on which version of removal occurs  return subTreePtr  }  “ |
|  |  |
| Text and Page Reference: | Data Abstraction and Problem Solving with C++: Walls and Mirrors, by Carrano and Henry, page 478 |