**Linked Lists Part4 Recursion**

|  |  |
| --- | --- |
| **Recursion**  A recursive algorithm uses recursion (a routine calling itself) to solve the problem. Some problems can be solved easily using recursion or iteration.  It is extremely important to determine the termination conditions for a recursive algorithm. For fib(n), the termination conditions are when n = 0 or n = 1. | **Example #1: Fibbonacci Number using recursion**  Fibonacci Number definition  F(0) = 0  F(1) = 1  F(n) = F(n-1)+F(n-2)  The sequence is  0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597, ...  printf("fib(3) is %d\n", fib(3));  // recursive function in C  int fib(int n)  {  switch (n)  {  case 0:  case 1:  return n;  default:  return fib(n-1) + fib(n-2);  }  } |
| **Show how to trace recursion.** |  |
| **Linked Lists and Recursion**  Suppose we had a singly linked list that we want to traverse to print the key for each node. What would be the termination condition for printLL(NodeLL \*p)?  // Invoking:  LinkedList list;  // Assume it contains:    printLL(list->pHead); | **Example #2: printing the contents of a singly LL using recursion**  void printLL(NodeLL \*p)  {  if (p == NULL)  return;  printf("%d ", p->element.key);  printLL(p->pNext);  }  That wasn't an improvement over the iterative solution. However, the next example is much easier using recursion. |
| **How would you print the contents of a linked list in reverse order?**  For the linked list from above, we want to print  30  20  10  // Invoking:  reversePrintLL(list->pHead); | **Example #3: printing a LL in reverse order**  void reversePrintLL(NodeLL \*p)  {  if (p == NULL)  return;  reversePrintLL(p->pNext);  printf("%d\n", p->iInfo);  } |
| **Show a trace of reversePrintLL** |  |
| **Exercise#1**  Show code for sumLL(NodeLL\*p) using recursion. It should return the sum of the element.data values for the entire list.  What is the termination condition? What gets returned? | **Exercise#1: sumLL recursively totals the iInfo values in each node.**  int sumLL(NodeLL \*p)  {  if (p == NULL)  return 0;  return p->element.data + sumLL (p->pNext);  } |
| **Show a trace.** |  |
| **Exercise#2**  Show code for searchLL(NodeLL\*p, int iMatch) using recursion return NULL if not found. (This does not return precedes. We will see that it is not needed with a recursive insertion and by address parameter passing.)  Termination condition(s)?  1. Empty list.  2. Matched.  3. Less than a node's element key.  4. End of list reached. | **Exercise#2: recursive searchLL which doesn't return preceding**  NodeLL \*searchLL(NodeLL \*p, int iMatch)  {  if (p == NULL)  return NULL;  if (iMatch == p->element.key)  return p;  if (iMatch < p->element.key)  return NULL;  return searchLL(p->pNext, iMatch);  } |
|  |  |
| **Recursive insertLL (modified definition)**  There are two major approaches to implement a recursive function for insertion into an ordered singly linked list:  **Reconstruct** - reconstruct the linked list recursively   * This approach adds a new node, but it reassigns the **next** pointer for many nodes. * We have to change our definition. **insertLL** won't always return the new node (or an existing matching node); instead, the return for the initial call will be the head (which could be the existing head or the new node if it should be inserted at the beginning of the list) * Every call returns a pointer to the remainder of the list.   **Utilize By Address** - take advantage of C's by address parameter passing   * Only change the pointers that are needed. * Java cannot use this approach since it doesn't support by address parameter passing * In C, this approach looks confusing due to how C passes parameters.   The reconstruct approach is shown on the right. **In CS2123, we are using the by address approach.**  **Five cases to consider:**   1. **Empty List:** if the parameter, p, is NULL, return the new node. 2. **First in List:** if our new value < current node's value, set the new node's next to the current node. Return the new node. 3. **Intermediate**: if our new value < current node's value, set the new node's next to the current node. Return the new node. 4. **End of List:** if the parameter, p, is NULL, return the new node. 5. **Match**: if our new value matches the current node's value, return the pointer to the current node. | **Example #4: recursive insertLL using reconstruct approach from cs1713**  // This definition returns a pointer to the first node  // instead of a pointer to the existing or new node.  NodeLL \*insertLL(NodeLL \*p, int iInfo)  {  if (p == NULL) // case 1 or case 4  return allocNodeLL(iInfo);  else if (iInfo == p->iInfo) // case 5  return p;  else if (iInfo < p->iInfo) // case 2 or 3  {  NodeLL \*pNew = allocNodeLL(iInfo);  pNew->pNext = p;  return pNew;  }  else  {  p->pNext = insertLL(p->pNext, iInfo);  return p;  }  }  // Example invoking code:  list->pHead = insertLL(list->pHead, 25); |
| **Before we do insertLL utilizing by address , we need to understand parameter passing.** | Refer to the parameter passing ppt. |
| **Advantage of By Reference and By Address Parameter Passing**  Languages that support **by reference** (e.g., PL/I) or by address (C) parameter passing can take advantage of having the address of the calling function's argument. We can change its value. (Note that Java doesn't support by address parameter passing.)  Initially, we pass &(list->pHead). Let's call the pointer receiving the address of list->pHead, **pp.**  **For case 1**:   * What is the value pp? address of pHead * What is the value of **\*pp**? NULL * What do we change the value of if we assign a value to **pp**?   pNew = allocNodeLL(value);  pp = pNew;   * What do we change the value of if we assign a value to \*pp?   pNew = allocNodeLL(value);  \*pp = pNew;  **For case 2**:   * What is the value of pp? address of pHead * What is the value of \*pp? address of node containing 40 * What is the value of (\*pp)->element.key? 40 * what is the value of (\*pp)->pNext? address of node containing 50 * what is the value of list->pHead? address of node containing 40 * what is the value of &( list->pHead )? address of pHead * what is the value of &((\*pp)->pNext)?address of the pNext attribute in the node containing 40. | **Example #5: parameter passing discussion prior to By Reference approach**  Case 1: Empty List    Case 2: First in List (insert 20)    Case 3: Intermediate between nodes (insert 45)    Case 4: End of List (insert 70) |
| **Recursive insertLL**  This can take advantage of passing the address of what is pointing to the node we are to process.   * Initially, we pass in &pHead. That is an address to a pointer variable. * Subsequently, we will pass the address of pNext. * **If we change the dereferenced parameter,** we will change the value of pHead or the value of that previous pNext.   **Our four cases:**   1. **Empty List:** if the dereferenced parameter is NULL, assign the parameter the new node. 2. **First in List:** if our new value < the pHead's (i.e., (\*pp)-> element.key), assign the parameter the new node**.** 3. **Intermediate**: if our new value < current node's key, assign the parameter the new node. 4. **End of List:** if the dereferenced parameter is NULL, assign the parameter the new node. | NodeLL\*insertLL(NodeLL \*\*pp, Element element)  {  // check for null pointer (i.e., end of list or  // empty list)  if (\*pp == NULL)  { // Case 1 or Case 4  \*pp = allocNodeLL(element);  return \*pp;  }  else if (element.key < (\*pp)->element.key)  { // Case 2 or Case 3  NodeLL \*pNew;  pNew = allocNodeLL(element);  pNew->pNext = \*pp;  \*pp = pNew;  return pNew;  }  else if (element.key == (\*pp)->element.key)  return \*pp;  else  return insertLL( &((\*pp)->pNext), element);  } |
|  |  |
|  |  |

**©2019 Larry W. Clark,** UTSA CS students may make copies for their personal use