|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| [clark](http://www.cs.utsa.edu/~clark/) | [cs1713](http://www.cs.utsa.edu/~clark/cs1713/cs1713.htm) | [syllabus](http://www.cs.utsa.edu/~clark/cs1713/cs1713Syllabus.htm) | [lecture notes](http://www.cs.utsa.edu/~clark/cs1713/lectureNotes.htm) | [programming assignments](http://www.cs.utsa.edu/~clark/cs1713/programmingAssignments.htm) | [homework](http://www.cs.utsa.edu/~clark/cs1713/homework.htm) | [set up](http://www.cs.utsa.edu/~clark/cs1713/setUp.htm) |
|  |  |  |  |  |  |  |

|  |  |
| --- | --- |
| **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. | 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(6) is %d\n", fib(6));  // 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.** |  |
| **Recursion and Linked Lists in LISP**  The programming language, LISP, uses a form of linked lists as its primary data structure. Instead of iterative loops (e.g., while, for), it uses recursion as its primary looping construct.  One odd fact about LISP is that the code and the data (values of variables) both are represented by linked lists. That means that code can be added at runtime by adding a linked list and it can be changed at runtime. The flexibility of having data that can easily become code has kept LISP popular for MANY years. | Example LISP code:  (DE LAST (L)  (COND ((NULL L) NIL)  ((NULL (CDR L)) (CAR L))  (T (LAST (CDR L)))  )  )  Note: (CDR L) is effectively L->NEXT  (CAR L) is effectively L->INFO  Note: you will **not** be tested on anything to do with LISP. |
| **Example Linked List** | Example #3: |
| **Linked Lists and Recursion**  Suppose we had a singly linked list that we want to traverse to print the info for each node. What would be the termination condition for printLL(Node \*p)? | void printLL(Node \*p)  {  if (p == NULL)  return;  printf("%d\n", p->iInfo);  printLL(p->pNext);  } |
| **Show a trace of printLL** |  |
| **How would you print the contents of a linked list in reverse order?**  For the linked list of example #3, we want to print  30  20  10 | void reversePrintLL(Node \*p)  {  if (p == NULL)  return;  reversePrintLL(p->pNext);  printf("%d\n", p->iInfo);  } |
| **Show a trace of reversePrintLL** |  |
| **Exercise**  Show code for the function countLL which is initially passed a pointer to the first node of a linked list. countLL should return a count of the nodes in the list.  Termination condition(s) / special cases?  1. Empty list of end of list  // initial call  iCount = countLL(pHead); | int countLL(Node \*p)  {  if (p == NULL)  return 0;  return 1 + countLL(p->pNext);  } |
| **Show a trace of countLL** |  |
| **Exercise**  Show code for sumLL(Node \*p) using recursion. It should return the sum of the iInfo values for the entire list.  Termination condition(s) / special cases?  1. **Empty** list or **End of list**  // initial call  iSum = sumLL(pHead); | int sumLL(Node \*p)  {  if (p == NULL)  return 0;  return p->iInfo + sumLL (p->pNext);  } |
| **Show a trace of sumLL** |  |
| **Exercise**  Show code for searchLL(Node \*p, int iMatch) which sorts an **ordered** singly linked list using recursion. If found return a pointer to that node. If not found, return NULL. (This does not return precedes. We will see that our new recursive insertLL doesn't need it.)  Termination condition(s) / special cases?  1. **Empty** list or **End of list:** if the parameter, p, is NULL, return NULL for not found.  2. **Match**: if our new value matches the current node's value, return the pointer to the current node.  3. **Less than:** When our matching value is less than the node's value, we know our value won't follow this node. We return NULL for not found. | Node \*searchLL(Node \*p, int iMatch)  {  if (p == NULL)  return NULL;  if (iMatch == p->iInfo)  return p;  if (iMatch < p->iInfo)  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.   We will use the **reconstruct** approach in this course. If you would like to see the other approach, please refer to my CS2123 course notes.  **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. | // This definition returns a pointer to the first node  // instead of a pointer to the existing or new node.  Node \*insertLL(Node \*p, int iInfo)  {  if (p == NULL) // case 1 or case 4  return allocateNode(iInfo);  else if (iInfo == p->iInfo) // case 5  return p;  else if (iInfo < p->iInfo) // case 2 or 3  {  Node \*pNew = allocateNode(iInfo);  pNew->pNext = p;  return pNew;  }  else  {  p->pNext = insertLL(p->pNext, iInfo);  return p;  }  } |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |