EECS402 Lecture 20

Andrew M. Morgan

Savitch, Ch. 13
Recursion

Printing a Singly-Linked List


```
class ListNodeClass
{
  public:
    int val;
    ListNodeClass *next;
};
class LinkedListClass
{
  public:
    ListNodeClass *head;     //data member

    LinkedListClass():head(0) {} //ctor
    void insertHead(int inVal);
    void printFwd()
    {
      ListNodeClass *tmp = head;
      cout << "Fwd List!" << endl;
      while (tmp != 0)
      {
        cout << "Val: " << tmp->val << endl;
        tmp = tmp->next;
      }
    }
};
```

From main():

```
LinkedListClass ll;
ll.insertHead(1);
ll.insertHead(5);
ll.insertHead(2);
ll.insertHead(9);
ll.printFwd();
```

Output:

```
Fwd List!
Val: 9
Val: 2
Val: 5
Val: 1
```

Printing a Singly-Linked List Backwards

- How would you write a printBwd() member function?

```
void LinkedListClass::printBwd()
{
  int i, j;
  int count = 0;
  ListNodeClass *temp = head;
  cout << "Bwd print!" << endl;
  while (temp != 0)
  {
    count++;
    temp = temp->next;
  }
  for (i = 0; i < count; i++)
  {
    temp = head;
    for (j = 0; j < (count - 1) - i; j++)
      temp = temp->next;
    cout << "Val: " << temp->val << endl;
  }
}
```

This works, but it is very inefficient, and very error-prone.

Leaving the "-1" out of the loop results in a seg fault, but it is not intuitive that it belongs there, etc...

Intro To Recursion

- A recursive function is a function that calls itself either:
  - Directly, in the function body
  - Indirectly, by calling another function, which, in turn calls the recursive function
- Many iterative algorithms can be done using recursion
- Note: Just because something can be done recursively does not mean it is the best way to do it
- *Recursion is best used when a problem can be continually broken down into problems that are easier to solve than the previous*
- A recursive function MUST have a terminal condition, or else infinite recursion will occur
  - Always provide a condition that will end the recursive calls
  - Condition should be checked BEFORE making the recursive call
- Think of the call stack that is created when thinking of recursion

Common Example, Factorial

- First, a simple, incredibly common example - factorial
  - n! = n * (n - 1) * (n - 2) * ... * (n - (n-1))
  - 4! = 4 * (4 - 1) * (4 - 2) * (4 - 3) = 4 * 3 * 2 * 1 = 24
- Notice that 4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1!
- This is a pattern that suggest recursion as a good solution

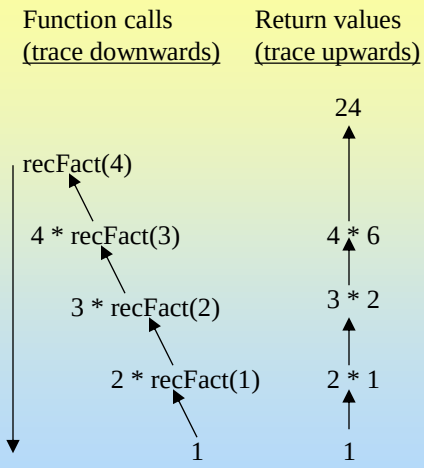
Iterative solution (NOT recursive):

```
int factorial(int n)
{
  int i;
  int fact = 1;
  for (i = n; i >= 1; i--)
  {
    fact *= i;
  }
  return fact;
}
```


Recursive Factorial Implementation

- Here is a recursive solution for the factorial problem:

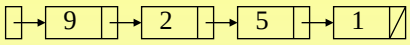
```
int recFact(int n)
{
  if (n <= 1)
  {
    //this is the base case
    //(or terminal case)
    return 1;
  }
  else
  {
    //Otherwise, return the
    //current val times the
    //factorial of the next
    //lowest integer
    return n * recFact(n - 1);
  }
}
```



Back To the Backwards List

- As you may have guessed, we can use recursion to print out a list backwards
  - While you haven't reached the end of the list, call the function with the "next" node
  - When you are done with the if statement, print the value
- Need to pass in a ListNodeClass pointer
- Start with "head" but we don't want the user to worry about implementation
  - Implement two functions
    - 1. Public function available in the interface, with no params This function will just call function #2 with head as the param
    - 2. Private function to do recursive call, taking in a ListNodeClass *

Recursively Printing a List Backwards




```
void recPrintBwd(ListNodeClass *n)
{
  bool endOfList = false;

  if (n != 0)
    recPrintBwd(n->next); //recursion
  else
    endOfList = true;  //base case

  if (!endOfList)
    cout << "Val: " << n->val << endl;
}

void printBwd2()
{
  cout << "Bwd list!" << endl;
  recPrintBwd(head);
}
```

On the way down

For node 9:
endOfList=false
recPrintBwd(node 2)

For node 2:
endOfList=false
recPrintBwd(node 5)

For node 5:
endOfList=false
recPrintBwd(node 1)

For node 1:
endOfList=false
recPrintBwd(node NULL)

For node NULL:
endOfList=true

On the way up

Node NULL:
no print - returns to node 1

Node 1:
prints 1 - returns to node 5

Node 5:
prints 5 - returns to node 2

Node 2:
prints 2 - returns to node 9

Node 9:
prints 9 - returns to calling function

Comments About Recursion

- Recursion is never "necessary"
  - Anything that can be done recursively, can be done iteratively
  - Recursive solution may seem more logical
    - For example, printing the list - the iterative solution given is very awkward, and does not model the human way of doing the problem, if given a list
- The recursive solution did not use any nested loops, while the iterative solution did
- However, the recursive solution made many more function calls, which adds a lot of overhead
- Recursion is NOT an efficiency tool - use it only when it helps the logical flow of your program

Problems With Recursion (Example 1)

- Avoid the following pitfalls demonstrated below:

```
int badRecursion1(int num)
{
  return num + badRecursion(num - 2);
}
```

Problems With Recursion (Example 2)

- Avoid the following pitfalls demonstrated below:

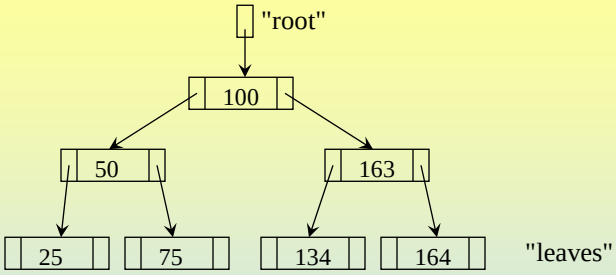
```
int badRecursion2(int num)
{
  if (num == 0)
  {
    return 1;     //base case
  }
  else
  {
    return num + badRecursion2(num - 2);
  }
}
```


A Binary Tree Data Structure

- A binary tree is yet another data structure
- Nodes of a binary tree consist of a value and 2 pointers

```
class BinaryTreeNodeClass
{
  int value;
  BinaryTreeNodeClass *left;
  BinaryTreeNodeClass *right;
};

BinaryTreeNodeClass *root;
```

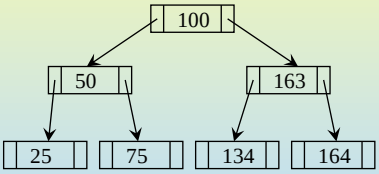
Graphical View of Binary Tree



- This specific tree has a couple special properties
  - It is a complete binary tree
    - All levels of the tree are completely filled
  - It is a binary search tree
    - For <u>every</u> node:
      - Any node to the left has a lower value
      - Any node to the right has a higher value

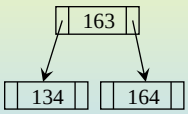
Binary Search Tree Information

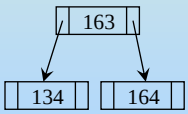
- ***Every*** node of a binary search tree is the root of a subtree, which is also a binary search tree
- Unlike linked lists, binary search trees can be searched in an efficient way
  - Possibly equivalent to a binary search of a sorted array
  - Example: Search for 134 in the shown tree



Root node contains 100
Since 134 > 100, it will be in the right subtree. No need to search left subtree, which disregards half of the tree elements!

Remaining Subtree:





Root node contains 163
Since 134 < 163, it will be in the left subtree. Able to disregard half of the nodes that remain

Remaining Subtree:

Value found!

Searching A BST, Iteratively


```
bool inBST(BSTNodeClass *root, int val)
{
  bool found = false;
  BSTNodeClass *tmp = root;

  while (tmp != 0 && !found)
  {
    if (tmp->value > val)
    {
      tmp = tmp->left;
    }
    else if (tmp->value < val)
    {
      tmp = tmp->right;
    }
    else
    {
      found = true;
    }
  }
  return found;
}
```


Searching A BST, Recursively

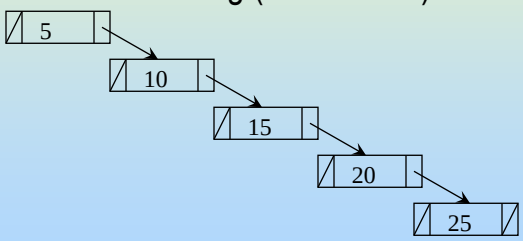

```
bool inBSTRec(BSTNodeClass *root, int val)
{
  bool found;

  if (root == 0)
  {
    found = false; //A base case
  }
  else if (root->value < val)
  {
    found = inBSTRec(root->right, val); //Recursive call
  }
  else if (root->value > val)
  {
    found = inBSTRec(root->left, val);  //Recursive call
  }
  else if (root->value == val)
  {
    found = true; //Another base case
  }
  return found;
}
```

Potential BST Problems

- Search and insert are efficient for a BST.
  - True with assumption that tree is "balanced"
- Consider the following operations

```
insertIntoBST(root, 5);
insertIntoBST(root, 10);
insertIntoBST(root, 15);
insertIntoBST(root, 20);
insertIntoBST(root, 25);
```

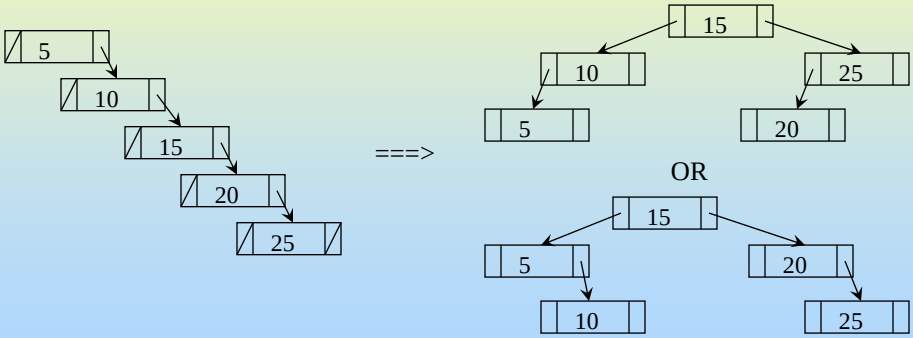
- Results in the following (unbalanced) tree



Note: Searching this BST results in a ***linear search*** instead of the binary search that was the primary advantage of a binary search tree!

Unbalanced BST Problems

- An unbalanced BST, as previously shown, has a search time of O(n).
- A balanced BST has a search time of O(log n).
- How can you be sure you keep your BST balanced?
- Write a balanceTree() function!



BST Efficiency Discussions

- Discuss the efficiency of the balanceTree() function
  - When should the function be called?
- Discuss the efficiency of deleting a node from a BST
  - Are there way to improve the efficiency of the delete operation?