

CSCI-1200 Data Structures — Spring 2020

Lecture 10 — Linked Lists & Recursion

Review from Lecture 9

- Explored a program to maintain a class enrollment list and an associated waiting list.
- Unfortunately, erasing items from the front or middle of vectors is inefficient.
- Iterators can be used to access elements of a vector
- Iterators and iterator operations (increment, decrement, erase, & insert)
- STL's `list` class
- Differences between indices and iterators, differences between STL `list` and STL `vector`.

Today's Class

- Simple recursion, Visualization of recursion, Iteration vs. Recursion
- “Rules” for writing recursive functions, Lots of examples!
- Building *our own* basic linked lists:
 - Stepping through a list
 - Push back
 - ... & even more in the next couple lectures!

10.1 Recursive Definitions of Factorials and Integer Exponentiation

- Factorial is defined for non-negative integers as:
- Computing integer powers is defined as:

$$n! = \begin{cases} n \cdot (n-1)! & n > 0 \\ 1 & n == 0 \end{cases} \qquad n^p = \begin{cases} n \cdot n^{p-1} & p > 0 \\ 1 & p == 0 \end{cases}$$

- These are both examples of *recursive definitions*.

10.2 Recursive C++ Functions

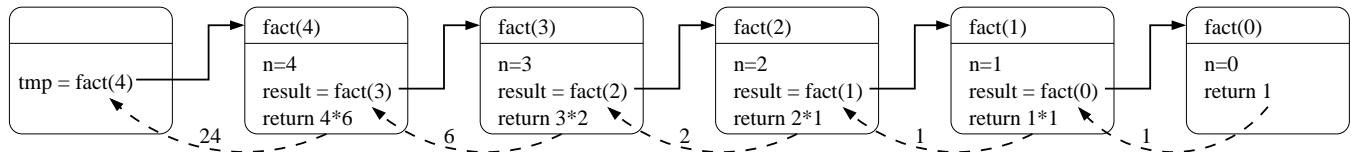
C++, like other modern programming languages, allows functions to call themselves. This gives a direct method of implementing recursive functions. Here are the recursive implementations of factorial and integer power:

```
int fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        int result = fact(n-1);
        return n * result;
    }
}

int intpow(int n, int p) {
    if (p == 0) {
        return 1;
    } else {
        return n * intpow( n, p-1 );
    }
}
```

10.3 The Mechanism of Recursive Function Calls

- For each recursive call (or any function call), a program creates an *activation record* to keep track of:
 - **Completely separate instances** of the parameters and local variables for the newly-called function.
 - The location in the calling function code to return to when the newly-called function is complete. (Who asked for this function to be called? Who wants the answer?)
 - Which activation record to return to when the function is done. For recursive functions this can be confusing since there are multiple activation records waiting for an answer from the same function.
- This is illustrated in the following diagram of the call `fact(4)`. Each box is an activation record, the solid lines indicate the function calls, and the dashed lines indicate the returns. Inside of each box we list the parameters and local variables and make notes about the computation.



- This chain of activation records is stored in a special part of program memory called *the stack*.

10.4 Iteration vs. Recursion

- Each of the above functions could also have been written using a `for` or `while` loop, i.e. *iteratively*. For example, here is an iterative version of factorial:

```
int ifact(int n) {  
    int result = 1;  
    for (int i=1; i<=n; ++i)  
        result = result * i;  
    return result;  
}
```

- Often writing recursive functions is more natural than writing iterative functions, especially for a first draft of a problem implementation.
- You should learn how to recognize whether an implementation is recursive or iterative, and practice rewriting one version as the other. Note: We'll see that not all recursive functions can be *easily* rewritten in iterative form!
- Note: The order notation for the number of operations for the recursive and iterative versions of an algorithm is usually the same. However in C, C++, Java, and some other languages, *iterative functions are generally faster than their corresponding recursive functions*. This is due to the overhead of the function call mechanism. Compiler optimizations will sometimes (but not always!) reduce the performance hit by automatically eliminating the recursive function calls. This is called *tail call optimization*.

10.5 Exercises

1. Draw a picture to illustrate the activation records for the function call

```
cout << intpow(4, 4) << endl;
```

2. Write an iterative version of `intpow`.

3. What is the order notation for the two versions of `intpow`?

10.6 Rules for Writing Recursive Functions

Here is an outline of five steps that are useful in writing and debugging recursive functions. Note: You don't have to do them in exactly this order...

1. Handle the base case(s).
2. Define the problem solution in terms of smaller instances of the problem. Use *wishful thinking*, i.e., if someone else solves the problem of `fact(4)` I can extend that solution to solve `fact(5)`. This defines the necessary recursive calls. It is also the hardest part!
3. Figure out what work needs to be done before making the recursive call(s).
4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation. (What are you going to do with the result of the recursive call?)
5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!

10.7 Location of the Recursive Call — Example: Printing the Contents of a Vector

- Here is a function to print the contents of a vector. Actually, it's two functions: a *driver function*, and a true recursive function. It is common to have a driver function that just initializes the first recursive function call.

```
void print_vec(std::vector<int>& v, unsigned int i) {
    if (i < v.size()) {
        cout << i << ": " << v[i] << endl;
        print_vec(v, i+1);
    }
}

void print_vec(std::vector<int>& v) {
    print_vec(v, 0);
}
```

- What will this print when called in the following code?

```
int main() {
    std::vector<int> a;
    a.push_back(3); a.push_back(5); a.push_back(11); a.push_back(17);
    print_vec(a);
}
```

- How can you change the second `print_vec` function as little as possible so that this code prints the contents of the vector in reverse order?

10.8 Fibonacci Optimization: Order Notation of Time vs. Space

The Fibonacci sequence is defined:

$$\text{Fib}_n = \begin{cases} 1 & n == 0 \\ 1 & n == 1 \\ \text{Fib}_{n-1} + \text{Fib}_{n-2} & n > 1 \end{cases}$$

1. Write a *simple* recursive version of Fibonacci that is a direct translation of the mathematical definition of the Fibonacci sequence.
2. Write an *iterative* version of Fibonacci that uses a vector to improve the running time of the function.
3. What is the order notation of the *running time* for each version?
4. What is the order notation of the *memory usage* for each version?

10.9 Binary Search

- Suppose you have a `std::vector<T> v` (for a placeholder type T), sorted so that:

`v[0] <= v[1] <= v[2] <= ...`

- Now suppose that you want to find if a particular value x is in the vector somewhere. How can you do this without looking at every value in the vector?
- The solution is a recursive algorithm called *binary search*, based on the idea of checking the middle item of the search interval within the vector and then looking either in the lower half or the upper half of the vector, depending on the result of the comparison.

```
template <class T>
bool binsearch(const std::vector<T> &v, int low, int high, const T &x) {
    if (high == low) return x == v[low];
    int mid = (low+high) / 2;
    if (x <= v[mid])
        return binsearch(v, low, mid, x);
    else
        return binsearch(v, mid+1, high, x);
}
template <class T>
bool binsearch(const std::vector<T> &v, const T &x) {
    return binsearch(v, 0, v.size()-1, x);
}
```

10.10 Exercises

1. Write a non-recursive version of binary search.
2. If we replaced the if-else structure inside the recursive binsearch function (above) with

```
if ( x < v[mid] )
    return binsearch( v, low, mid-1, x );
else
    return binsearch( v, mid, high, x );
```

would the function still work correctly?

10.11 Basic Mechanisms: Stepping Through the List

- We'd like to write a function to determine if a particular value, stored in `x`, is also in the list.
- We can access the entire contents of the list, one step at a time, by starting just from the `head` pointer.
 - We will need a separate, local pointer variable to point to nodes in the list as we access them.
 - We will need a loop to step through the linked list (using the pointer variable) and a check on each value.

10.12 Exercise: Write `is_there`

```
template <class T> bool is_there(Node<T>* head, const T& x) {
```

- If the input linked list chain contains n elements, what is the order notation of `is_there`?

10.13 Basic Mechanisms: Pushing on the Back

- Goal: place a new node at the end of the list.
- We must step to the end of the linked list, remembering the pointer to the last node.
 - This is an $O(n)$ operation and is a major drawback to the ordinary linked-list data structure we are discussing now. We will correct this drawback by creating a slightly more complicated linking structure in our next lecture.
- We must create a new node and attach it to the end.
- We must remember to update the `head` pointer variable's value if the linked list is initially empty.
 - Hence, in writing the function, we must pass the pointer variable **by reference**.

10.14 Exercise: Write `push_front`

```
template <class T> void push_front( Node<T>* & head, T const& value ) {
```

- If the input linked list chain contains n elements, what is the order notation of the implementation of `push_front`?

10.15 Exercise: Write `push_back`

```
template <class T> void push_back( Node<T>* & head, T const& value ) {
```

- If the input linked list chain contains n elements, what is the order notation of this implementation of `push_back`?

10.16 Inserting a Node into a Singly-Linked List

- With a singly-linked list, we'll need a pointer to the node *before* the spot where we wish to insert the new item.
- If `p` is a pointer to this node, and `x` holds the value to be inserted, then the following code will do the insertion. Draw a picture to illustrate what is happening.

```
Node<T> * q = new Node<T>; // create a new node
q -> value = x;           // store x in this node
q -> next = p -> next;    // make its successor be the current successor of p
p -> next = q;           // make p's successor be this new node
```

- Note: This code will not work if you want to insert `x` in a new node at the *front* of the linked list. Why not?

10.17 Removing a Node from a Singly-Linked List

- The remove operation itself requires a pointer to the node *before* the node to be removed.
- Suppose `p` points to a node that should be removed from a linked list, `q` points to the node before `p`, and `head` points to the first node in the linked list. Note: Removing the first node is an important special case.
- Write code to remove `p`, making sure that if `p` points to the first node that `head` points to what was the second node and now is the first after `p` is removed. Draw a picture of each scenario.

10.18 Exercise: Singly-Linked List Copy

Write a *recursive* function to copy all nodes in a linked list to form an new linked list of nodes with identical structure and values. Here's the function prototype:

```
template <class T> void CopyAll(Node<T>* old_head, Node<T>*& new_head) {
```

10.19 Exercise: Singly-Linked List Remove All

Write a *recursive* function to delete all nodes in a linked list. Here's the function prototype:

```
template <class T> void RemoveAll(Node<T>*& head) {
```

10.20 Next time... Can we get better performance out of linked lists? Yes!