CSCI-1200 Data Structures — Fall 2025
Lecture 10 — Simple Recursion & Linked Lists

Review from Lecture 9

- Unfortunately, erasing items from the front or middle of vectors is inefficient.
- Introduction to iterators: for access, increment, decrement, erase, & insert
- Differences between indices and iterators
- Introduction to STL's `list` class

Today's Class

- Visualization of Recursion,
- Iteration vs. Recursion
- "Rules" for Writing Recursive Functions
- *Implementing our own* linked list data structure from scratch:
  - Stepping through a list, searching for an element
  - Push front, push back, insert in the middle
- Singly-linked vs. Doubly-linked lists!
- Next Lecture: Building a `ds_list` class implementation (mimicking STL `list`)

10.1 Recursive Definitions of Factorials and Integer Exponentiation

- Factorial is defined for non-negative integers as:

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

- Computing integer powers is defined as:

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

- 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:

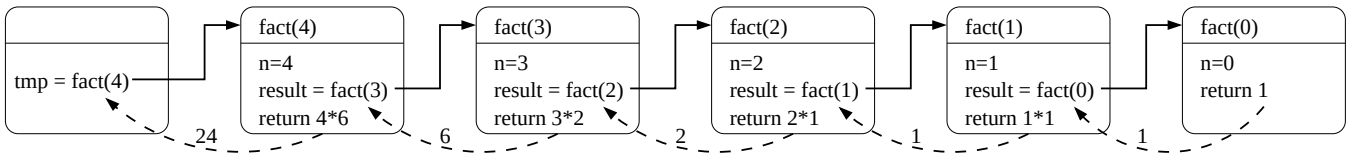
```
int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    int result = fact(n-1);
    return n * result;
  }
}
```

```
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*. The stack facilitates storing multiple separate memory locations for each instance of the arguments and local variables for the recursive function.
- We've already been drawing pictures of the stack and the local variables that functions place on the stack. Once we leave a function (or any scope delimited with `{ ... }`), the local variables on the stack are cleaned up and we cannot / should not try to access them again.

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 some more complex recursive functions require much more effort to rewrite in iterative form, and it may be preferable for readability to leave them in recursive form.
- Note: Big O 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 a little bit 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 / stack frames for the function call:

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

2. Write an iterative version of `intpow`.
3. What is the big O 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 sets up some data, initializes some variables, etc. for 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 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);
}
```

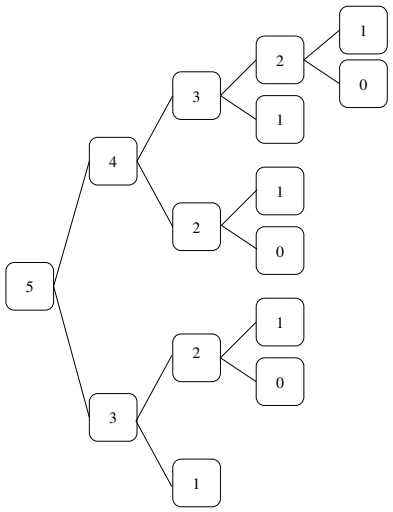
10.9 Exercises

1. Draw a picture of the activation records / stack frames generated by the binary search algorithm.
2. What is the Big O Notation for the binary search algorithm?
3. Write a non-recursive version of binary search.

10.10 Fibonacci: Big O Notation of Time vs. Space

- Mathematical definition:

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



- The *simple* recursive version of Fibonacci that is a direct translation of the mathematical definition of the Fibonacci sequence:

```
int fib ( int n ) {
  assert ( n >= 0 );
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fib (n-1) + fib (n-2);
  }
}
```

- A revised algorithm that is *iterative*, and uses an STL vector:
  NOTE: Often when re-writing a recursive function to be iterative, we have to use extra memory to explicitly store intermediate values.

```
int ifib_vec ( int n ) {
  assert ( n >= 0 );
  std::vector<int> v;
  v.push_back(1);
  v.push_back(1);
  for (int i = 2; i <= n; i++) {
    v.push_back(v[i-1] + v[i-2]);
  }
  return v[n];
}
```

  NOTE: Usually when re-writing a recursive function to be iterative, they both have the same Big O Notation!

- A further revision of the *iterative* algorithm that doesn't use an STL vector:

```
int ifib_novec ( int n ) {
  assert ( n >= 0 );
  if (n < 2) return 1;
  int prevprev = 1;
  int prev = 1;
  for (int i = 2; i < n; i++) {
    int tmp = prevprev + prev;
    prevprev = prev;
    prev = tmp;
  }
  return prevprev + prev;
}
```

  NOTE: Not every algorithm can be revised to remove all non-constant memory usage!

- What is the Big O Notation for the *running time* for each version?
- What is the Big O Notation for the *memory usage* for each version *ignoring* the memory used to store the recursive function calls on the stack?
- What is the Big O Notation for the *memory usage* for each version *including* the memory used to store the recursion function calls on the stack?

10.11 Definition of a Linked List

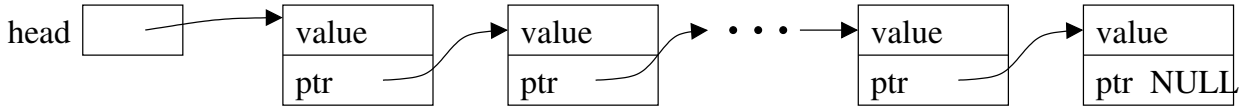
- A linked list is either:
  - Empty, or
  - Contains a node storing a value and a pointer to a linked list.
- Yes, the definition is recursive.
  And similarly, our implementation of many linked list functions can be recursive as well!

10.12 Visualizing & Implementing Linked Lists

- A linked list is made of `Node` objects. The `Node`s are be templated to allow linked lists of different types:

```
template <class T>
class Node {
public:
  T value;
  Node* ptr;
};
```

- The first node in the linked list is called the *head* node. In the example below, we store a pointer to the head node in a variable on the stack named `head`. The `Node`s of the linked list are dynamically allocated on the heap.



- It is important to have a separate variable pointer to the first node, since the "first" node may change as we edit the data stored in the list.
- Note that the diagram above is a conceptual view only. The memory locations could be anywhere – they aren't necessarily arranged in in this order, in adjacent memory locations. The actual values of the memory addresses aren't usually meaningful.
- The last node MUST have NULL for its pointer value — you will have all sorts of trouble if you don't ensure this!
- You should make a habit of drawing pictures of linked lists to figure out how to do the operations.

10.13 Stepping Through the List, Searching for a Value

- We'd like to write a function to determine if a particular value, stored in `x`, is 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.

10.14 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 Big O Notation of our `is_there` function?
- If you wrote `is_there` iteratively, re-write it recursively. If you wrote it recursively, then re-write it iteratively.

10.15 Overview: Adding an Element at the Front of the List

- We must create a *new* node.
- We must permanently update the `head` pointer variable's value.
  Therefore, we must pass the pointer variable ***by reference*** .

10.16 Exercise: Write `push_front`

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

- If the input linked list chain contains n elements, what is the Big O Notation of our `push_front` function?

10.17 Overview: Adding an Element at the Back of the List

- We must step to the end of the linked list, remembering the *pointer to the last node.*
- 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.

10.18 Exercise: Write `push_back`

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

- If the input linked list chain contains n elements, what is the Big O Notation of our `push_back` function?

10.19 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. *NOTE: This is not how STL list's insert function works!*
- Let's say that `p` is a pointer to this node, and `x` holds the value to be inserted. First, draw a picture to illustrate what is happening.

- Then, write a *fragment of code* that will do the insertion.

- 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?

10.20 Limitations of Singly-Linked Lists

- We can only move through it in one direction
- We need a pointer to the node *before* the spot where we want to insert or erase.
  This requirement does not match the STL list specification!
- Appending a value at the end requires that we step through the entire list to reach the end.
  The Big O Notation does not match the STL list specification!

10.21 Generalizations of Singly-Linked Lists

- Three common generalizations (can be used separately or in combination):
  - Doubly-linked: allows forward and backward movement through the nodes
  - Circularly linked: simplifies access to the tail, when doubly-linked
  - Dummy header node: simplifies special-case checks

10.22 Transition to a Doubly-Linked List Structure

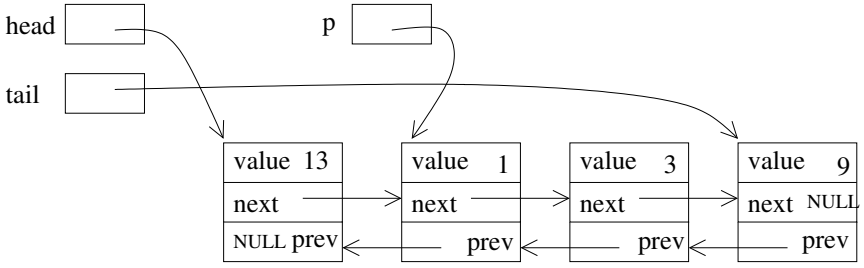
- The revised `Node` class has two pointers, one going "forward" to the successor in the linked list and one going "backward" to the predecessor in the linked list. We will have a `head` pointer to the beginning *and* a `tail` pointer to the end of the list.

```
template <class T> class Node {
public:
    Node() : next_(NULL), prev_(NULL) {}
    Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}
    T value_;
    Node<T>* next_;
    Node<T>* prev_;
};
```

- The tail pointer is not strictly necessary to access the data, but it facilitates efficient push-back operations.
- **Question:** If we have the tail pointer, do we still need the list to be doubly-linked?

10.23 Inserting a Node into the Middle of a Doubly-Linked List

- Suppose we want to insert a new node containing the value 15 following the node containing the value 1. We have a temporary pointer variable, `p`, that stores the address of the node containing the value 1.

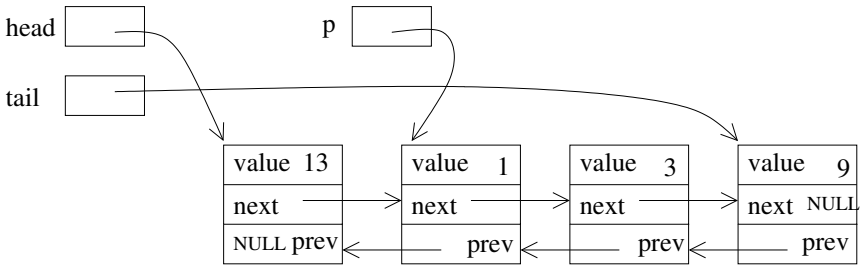


- What must happen? Editing the diagram above...
  - The new node must be created, using another temporary pointer variable to hold its address.
  - Its two pointers must be assigned.
  - Two pointers in the current linked list must be adjusted. Which ones?

  Assigning pointers for the new node MUST occur before changing pointers of the existing linked list nodes!
- **Exercise:** Write the code as just described. Focus first on the general case: Inserting a new into the middle of a list that already contains at least 2 nodes.

10.24 Removing a Node from the Middle of a Doubly-Linked List

- Now instead of inserting a value, suppose we want to remove the node pointed to by `p` (the node whose address is stored in the pointer variable `p`)
- Two pointers need to change before the node is deleted! All of them can be accessed through the pointer variable `p`.
- **Exercise:** Edit the diagram below, and then write this code.



10.25 Special Cases of Remove

- If `p==head` and `p==tail`, the single node in the list must be removed and both the `head` and `tail` pointer variables must be assigned the value `NULL`.
- If `p==head` or `p==tail`, then the pointer adjustment code we just wrote needs to be specialized to removing the first or last node.