More on Linked Lists
Introduction to Recursion

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, February 13, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ The structure of a package
- ✓ Parameter passing
- ✓ Operator overloading
- ✓ Silently written & called functions
- ✓ Pointers & dynamic allocation
- ✓ Managing resources in a class
- ✓ Templates
- ✓ Containers & iterators
- ✓ Error handling
- ✓ Introduction to exceptions
- Introduction to Linked Lists

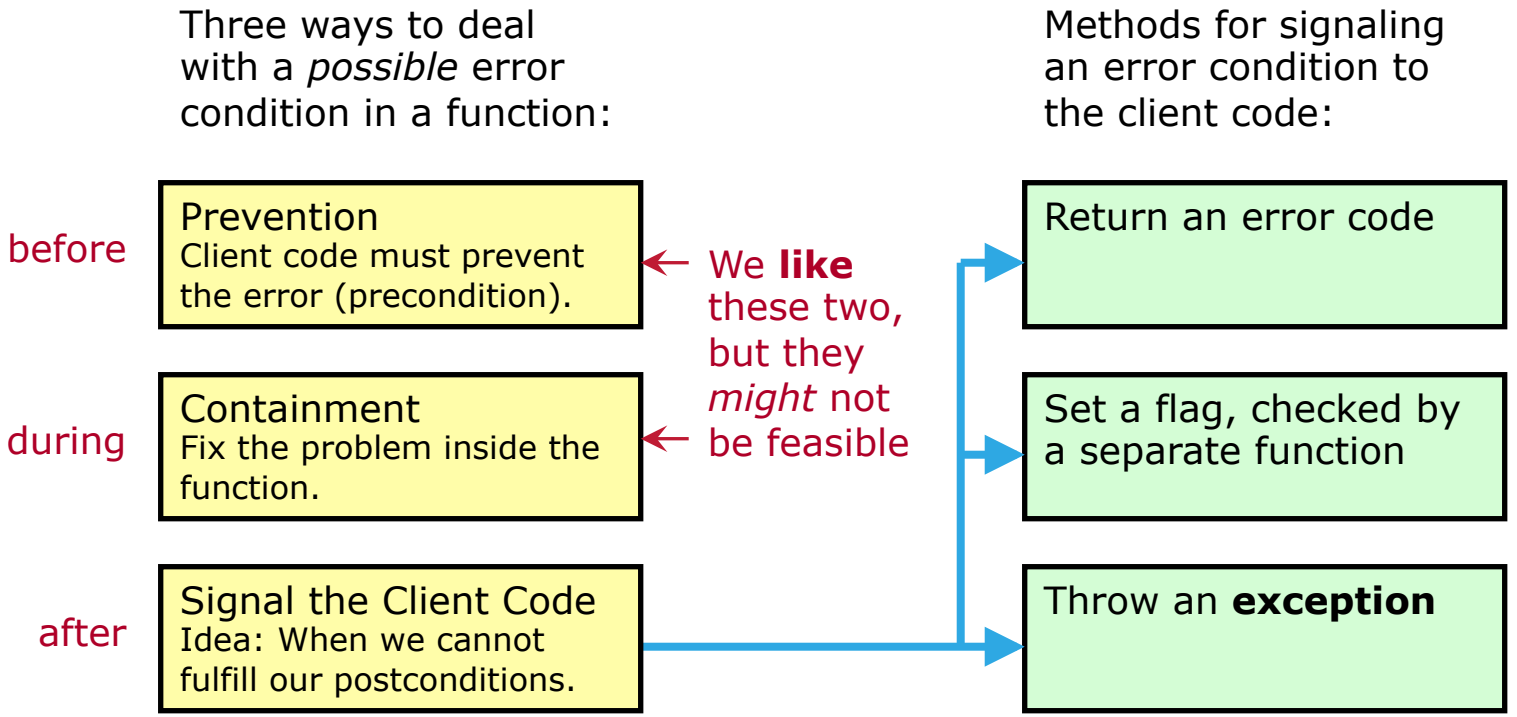
Major Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Invariants
- ✓ Testing
- ✓ Some principles

Review
Error Handling

An **error condition** (or "error") is a condition occurring during runtime that cannot be handled by the normal flow of execution.

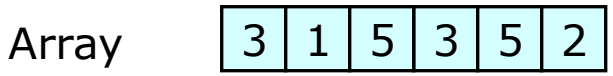
- Not necessarily a bug or a user mistake.
- Example: Could not read file.



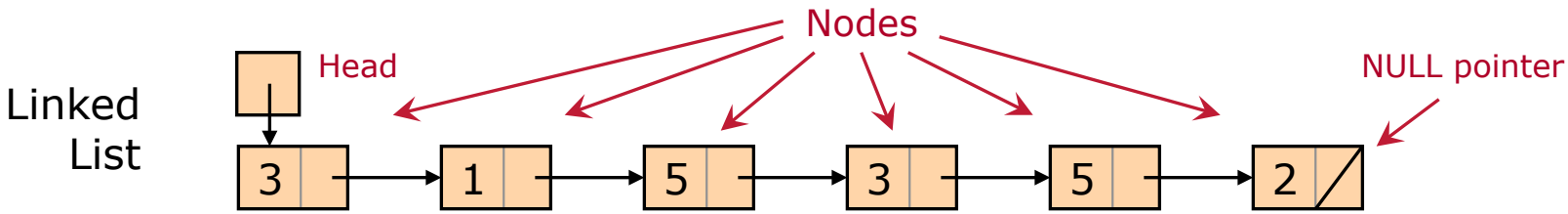
Introduction to Linked Lists
Review: Basics

We discuss Linked Lists in detail later in the semester. For now:

- Like an array, a Linked List is a structure for storing a sequence of items.



- A Linked List is composed of **nodes**. Each has a single data item and a pointer to the next node.



- These pointers are the **only** way to find the next data item. Thus, unlike an array, we cannot quickly skip to (say) the 100th item in a Linked List. Nor can we quickly find the previous item.
- A Linked List is a one-way sequential-access data structure. Thus, its natural iterator is a **forward iterator**, which has only the ++ operator.

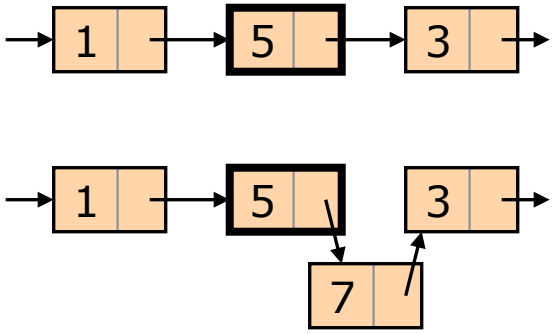
Introduction to Linked Lists
Review: Advantages

Why not always use (smart) arrays?

- One important reason: we can often insert and remove much faster with a Linked List.

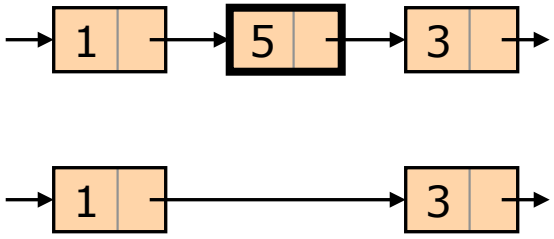
Inserting

- Inserting an item at a given position in an array is slow-ish.
- Inserting an item at a given position (think "iterator") in a Linked List is very fast.
- Example: insert a "7" after the bold node.



Removing

- Removing the item at a given position from an array *is also slow-ish.*
- Removing the item at a given position from a Linked List is very fast.
  - We need an iterator to the *previous* item.
- Example: Remove the item in the bold node.



Introduction to Linked Lists
Review: Implementation

A Linked List node might be implemented like this.

```
template <typename ValueType>
struct LLNode {
    ValueType data_;  // Data for this node
    LLNode * next_;   // Ptr to next node, or NULL if none

    // The following simplify creation & destruction
    LLNode(const ValueType & theData, LLNode * theNext = 0)
        :data_(theData), next_(theNext)
    {}
    ~LLNode()
    { delete next_; }
};
```

Then the head of our list would keep an **(LLNode<…> *)**.

Unit Overview
Recursion & Searching

We now begin a unit on recursion & searching.

Major Topics

- Introduction to Recursion
- Search Algorithms
- Recursion vs. Iteration
- Eliminating Recursion
- Recursive Search with Backtracking

After this, we will look at Algorithmic Efficiency & Sorting.

Introduction to Recursion
Basics — Definition

A **recursive** algorithm is one that makes use of itself.

- An algorithm solves a problem. If we can write the solution of a problem in terms of the solutions to **simpler** problems of the same kind, then recursion may be called for.
- At some point, there needs to be a simplest problem, which we solve directly. This is the **base case**.

Introduction to Recursion
Basics — Four Questions

When designing a recursive algorithm or function, consider the following questions:

- How can you define the problem in terms of a smaller problem of the same type?
- How does each recursive call diminish the size of the problem?
- What instance of the problem can serve as the base case?
- As the problem size diminishes, will you reach this base case?

Introduction to Recursion
Sum Example — The Goal

We start with a (somewhat silly) example: Write a recursive function to find the sum of the first *n* integers, given *n*.

- So, given 3, we return 1 + 2 + 3 = 6.
- We look at this as practice in thinking about recursion. Then we will try a more serious example.

Introduction to Recursion
Sum Example — Four Questions

How can you define the problem in terms of a smaller problem of the same type?

- $1 + 2 + \ldots + n = [1 + 2 + \ldots + (n-1)] + n$.
- Say $f(n) = 1 + 2 + \ldots + n$. Then, for $n > 0$, $f(n) = f(n-1) + n$.
  - This is called a **recurrence relation**.

How does each recursive call diminish the size of the problem?

- It reduces by 1 the number of numbers to be summed.

What instance of the problem can serve as the base case?

- $n = 0$.
  - $n = 1$ would also have worked.

As the problem size diminishes, will you reach this base case?

- Yes, as long as n is nonnegative.
- Therefore the statement "$n >= 0$" needs to be a **precondition**.

Sum Example — Specifications & Algorithm

Next we write **specifications**.

- Let's write a recursive function **`sumUpTo`** that takes a single **`int`** parameter and returns an **`int`**.
  - The parameter will be "*n*", and the return value will be the sum.
- Preconditions
  - *n* >= 0.
- Postconditions
  - Return == 1 + … + *n*.

The recurrence relation turns into an **algorithm**.

- Are we in the base case? If so handle it.
  - If *n* is 0, then return 0.
- Otherwise, recurse.
  - Recursive call, parameter: *n* – 1.
  - Add *n* to the result.
  - Return this.

For $n > 0$, $f(n) = f(n-1) + n$.

Introduction to Recursion
Sum Example — Coding

Now we write the actual code:

```
// sumUpTo
// Given n, return sum of integers 1 to n.
// Recursive.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n)
{
    if (n == 0)
        return 0;
    else
        return sumUpTo(n-1) + n;
}
```

How do we know we can make the recursive call?

- Hint: When we call a function, we must satisfy its preconditions.

Sum Example — Invariants

We know we can make the recursive call because we have an **invariant** that makes the preconditions for the call true:

```
// sumUpTo
// Given n, return sum of integers 1 to n.
// Recursive.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n)
{
    if (n == 0)
        return 0;
    else  // Invariant: n >= 1. (Therefore n-1 >= 0.)
        return sumUpTo(n-1) + n;
}
```

Here we have an invariant that says $n \geq 0$ (the precondition).

Here we leave if n is exactly 0.
Result: If we stay, then $n \geq 1$.

This is the precondition for **this** function call.

Sum Example — Iterative Version

Often we do not really need recursion:

```
// sumUpTo
// Given n, return sum of integers 1 to n.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n)
{
    int sum = 0;
    for (int i = 1; i <= n; ++i)
        sum += i;
    return sum;
}
```

This uses **iteration** (a loop) instead.

Sum Example — Formula Version

And sometimes there is a not-so-obvious way to do things *much* faster:

```
// sumUpTo
// Given n, return sum of integers 1 to n.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n)
{
    return n * (n+1) / 2;
}
```

Search Algorithms
Binary Search Example — What is It?

The sum example from “Introduction to Recursion” was a little silly. Here is one that is not.

Binary Search

- How does it work?
- How would you implement it recursively?

Binary Search Example — Method

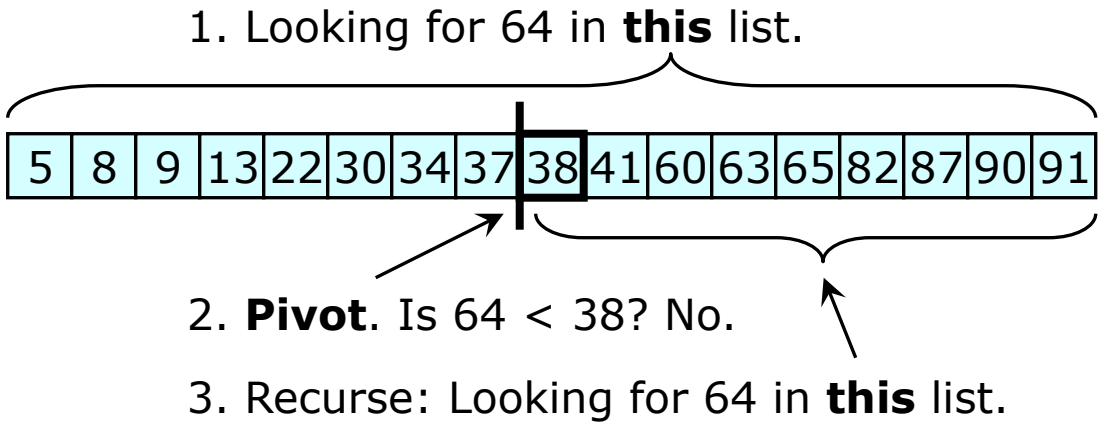
Binary Search is an algorithm to find a given **key** in a **sorted list**.

- Here, *key* = thing to search for. Often there is associated data.
- In computing, *sorted* = in (some) order.

Procedure

- Pick an item in the middle: the **pivot**.
- Use this to narrow search to top or bottom half of list. Recurse.

Example: Binary Search for 64 in the following list.



Binary Search Example — Four Questions

How can you define the problem in terms of a smaller problem of the same type?

- Look at the middle of the list. Then recursively search the top or bottom half, as appropriate.

How does each recursive call diminish the size of the problem?

- It cuts the size of the list in half (roughly).

What instance of the problem can serve as the base case?

- List size is 0 or 1.

As the problem size diminishes, will you reach this base case?

- Yes.
  - A list cannot have negative size.

Binary Search Example — Getting Practical

Suppose we wish to write a function to do Binary Search.

- How should the list to search in be given?
- Should there be any other parameters?
- What should the function return?

Suppose we wish to write a function to do Binary Search.

- How should the list to search in be given?
  - Parameters: Two iterators, one pointing to first item and one pointing just past last item.
- Should there be any other parameters?
  - Yes, the value to search for.
- What should the function return?
  - There are several options: just **`true`** or **`false`** (found or not), an iterator to the value found, an iterator to the first equal value in the list, two iterators specifying the range of equal values, etc.
  - Our function will return a **`bool`** indicating whether the value was found.

Binary Search Example — Do It

TO DO

- Examine a function **`binSearch`** to do Binary Search, as discussed.

Binary Search Example — Comments

Function **`binSearch`** only determines **whether** an item is in a list. It does not tell **where** it is in the list.

All less-than/greater-than comparisons on objects of the value type are performed with **`operator<`**.

Document "requirements on types", to indicate which types the function can be used with.

- Iterators given (**`RAIter`**) must be random-access iterators.
  - Recall: "Random-access iterator" is a standard *iterator category*.
- Dereferencing an **`RAIter`** must give a **`ValueType`**.
- Type **`ValueType`** needs copy ctor, **`operator<`**, **`operator==`**, dctor.
  - To think about: Can we make do with fewer than this?

We also need pre- & postconditions.

- Useful idea: A **valid range** is one in which, if we start at the beginning and increment repeatedly, we will eventually reach the end, and all the data we pass through in the interim, are valid.

Search Algorithms
Binary vs. Sequential Search [1/3]

We have discussed **Binary Search**. Another algorithm is **Sequential Search**.

- Also called "Linear Search".
- In *Sequential Search*, we search a list from beginning to end, looking at each item until we either find what we are searching for, or we reach the end of the list.

Sequential Search is applicable to more situations than Binary Search. To do a Binary Search efficiently, the list should be:

- Sorted (required for Binary Search)
- Random-Access (for efficiency)

So, why do we like Binary Search?

Search Algorithms
Binary vs. Sequential Search [2/3]

We like Binary Search better than Sequential Search, because it is **much faster** …

Number of Items in List	Look-Ups: Binary Search* (worst case)	Look-Ups: Sequential Search (worst case)
1	1	1
2	2	2
4	3	4
100	8	100
10,000	15	10,000
1,000,000	21	1,000,000
10,000,000,000	35	10,000,000,000
n	*Roughly* $\log_2 n$	n

*Using our version of the algorithm. Other variations may differ slightly, but the main point of this table remains valid.

Search Algorithms
Binary vs. Sequential Search [3/3]

… and, therefore, the amount of data it can process in the **same** amount of time is **much greater**.

Number of Look-Ups We Have Time to Perform	Maximum Allowable List Size: Binary Search	Maximum Allowable List Size: Sequential Search
1	1	1
2	2	2
3	4	3
4	8	4
10	512	10
20	524,288	20
40	549,755,813,888	40
k	*Roughly* 2^k	*k*

"The fundamental law of computer science: As machines become more powerful, the efficiency of algorithms grows more important, not less."

— Nick Trefethen

Better Binary Search — Equality vs. Equivalence

Let's improve our Binary Search function.

First, some ideas.

If we use **`operator<`** to search for an item, then we probably should also use **`operator<`** to make sure we have the correct item.

- This allows us to handle types that:
  - Do not have **`operator==`**.
  - Have **`operator==`**, but do not define it in a way that is consistent with **`operator<`**.
- "**`a == b`**" is **equality**.
- "**`!(a < b) && !(b < a)`**" is **equivalence**.

Random-access iterators can do pointer-style arithmetic:

- Adding Integers
  - `iter1 = iter2 + 3;`
  - `iter1 += 3;`
- Difference
  - `n = iter1 - iter2;`

In general, iterators cannot do all this.

However, we can get the same results for more general **forward** iterators, using **`std::advance`** and **`std::distance`**.

- These are defined in **`<iterator>`**.
- "**`std::advance(iter, n)`**" is like "**`iter += n`**".
- "**`std::distance(iter1, iter2)`**" is like "**`iter2 - iter1`**".
- These two functions are fast for random-access iterators; they may be slower for other iterators.
- See the STL doc's.

TO DO

- Improve function **`binSearch`**:
  - Avoid redundant computations in finding size of list, base case.
  - Do not require **`ValueType`** to have a copy ctor, dctor, or **`operator==`**.
    - Use **only** **`operator<`**.
  - Allow the iterators to be forward iterators.
    - Is this really an improvement? Maybe …