CSE 12:

Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Some slides adapted from Paul Kube.

Lecture Ten
17 Aug 2011

Linear data structures: a brief review.

Linear data structures

- So far in this course we have learned the basic *linear* data structures:
  - Array list
  - Linked list
  - Stack
  - Queue
- These structures are *linear* because each element contained within them is *adjacent* to at most 2 other elements.

Linear data structures

- Linked lists and array lists provide a form of "permanent" storage of arbitrary data.
- Stacks and queues provide (typically) "temporary" storage to data that we expect to remove at some later point in time.
    - LIFO for stack, FIFO for queue.
- All these data structures provide convenient containers for storing *unrelated* data.
    - There needn't be any relationship among the individual data.

Linear data structures

- With Java generics, we gained the ability to restrict membership to an ADT to a particular class.
  - E.g., allow only `String` objects to be added to a `List12` container).
- But beyond the class of the objects, we didn't "care" about any relationships between the data.
- In particular, we didn't care whether the ADT stored the individual data in some "natural order":
  - E.g., alphabetical order for `String`s, integer order for `Integer`s.

Linear data structures

- Ignoring any relationships between data elements allowed for an ADT that was:
  - Simple to implement -- no need to *consider* order relations.
  - Flexible to use -- no need to *define* an order relation.
- However, this simplicity/flexibility comes at the cost that data retrieval is often *slower than it needs to be*.
  - By considering the natural order relations between objects, we can create data structures with superior asymptotic time costs for storage/retrieval operations.

Linear data structures: asymptotic time costs

- Let's review the "score card" of the ADTs we've covered so far.
- Let's consider three fundamental operations:
  - `void add (T o);`
  - `void remove (T o);`
  - `T find (T o);`
    Search for an element in the container that `equals o` and returns it; if no such object exists, then returns `null`.

Array-list and linked-list scorecard

	Array-list	Linked-list	
`add(o)`	$O(1)$	$O(1)$	Adding is fast.
`find(o)`	$O(n)$	$O(n)$	Finding is slow.
`remove(o)`	$O(n)$	$O(n)$	Removing is slow.

Array-list and linked-list scorecard

- There are many occasions where the user will *add* new data relatively *rarely*, but want to *find* data already in the data structure relatively *frequently*.
- In order to improve the asymptotic time cost of the **`find(o)`** and **`remove(o)`** operations, we will make use of order relationships between data elements.
  - Once we've *found* an element within a data structure, it is typically easy for the data structure to *remove* it.

Why `find` something?

- It may strike some as odd that an ADT would support the method `T find (T o)`, e.g.:

```
final Student student = ...
final Student student2 = _list.find(student);
```

- After all, if the user knows the object o he/she is looking for, then why call `find` at all?
- *Answer*: sometimes the user knows *part* of the information about an object `o`, but does not have the whole record.
  - This illustrates the difference between a record's *key* and its *value*.

Keys and values

- The part of the `Student` object that the user always knows is called the *key* (e.g., student ID number at Student Health).
- The rest of the `Student` record is called the *value*.

```
class Student {
  String _studentID;
  String _firstName, _lastName;
  String _address;

  Student (String studentID) {
    _studentID = studentID;
  }

  Student (String studentID, String firstName, String lastName,
           String address) {
    _studentID = studentID;
    _firstName = firstName;
    _lastName = lastName;
    _address = address;
  }
}
```

Key

Value

Keys and values

- The user may store many **`Student`** objects inside a **`List12`** container, e.g.:

```
list.add(new Student("A123", "Bill", "Carter", "123 Main St"));
list.add(new Student("A213", "Jimmy", "Clinton", "124 Main St"));
...
list.add(new Student("B092", "Hillary", "Nixon", "125 Main St"));
```

- Later, the user may wish to find a particular **`Student`** object using just the key, e.g., the student ID:

```
final Student cse12Student = list.find(new Student("A123"));
```

Student containing both the key and value.

Student initialized with just the key.

Overriding `equals(o)`

- In order for the `find(o)` method to work properly, the `Student` object must also override the `equals(o)` method so that a `Student` initialized with just a key will "equal" a `Student` initialized with both key and value:

```
class Student {
  ...
  boolean equals (Object o) {
    final Student other = (Student) o;
    return _studentID.equals(other._studentID);
  }
}
```

Downcast!

Accessible even though `_studentID` may be `private`.

Overriding `equals(o)`

- The implementation of the `find(o)` method will then implicitly call this method.
- E.g., consider the `find(o)` method in an ArrayList:

```
T find (T o) {
  for (int i = 0; i < _numElements; i++) {
    if (_underlyingStorage[i].equals(o)) {
      return _underlyingStorage[i];
    }
  }
  return null;
}
```

Will call `Student.equals(o)`.

Overriding `equals(o)`

- Note that `Student.equals(o)` will be called even if `find(o)` is implemented in terms of `Object`s. (This assumes of course that `Student` objects were actually added to the list.)
- This is due to Java's *dynamic binding* of methods -- the *runtime* type of o is used to determine which `equals(o)` method to call.

```
Object find (Object o) {
  for (int i = 0; i < _numElements; i++) {
    if (_underlyingStorage[i].equals(o)) {
      return _underlyingStorage[i];
    }
  }
  return null;
}
```

Will still call `Student.equals(o)` instead of `Object.equals(o)`.

Keys and values

- Some data structures explicitly separate the key from the value when the user adds the element to the container.
- Example:
  - A "hash map/table" (covered later in this course) allows $O(1)$-time retrieval of any *value* given its *key*.
  - To add a new entry to the table, the user calls `put(key, value)`, e.g.:

```
hashMap.put("A123",     Key
            new Student("A123", "Bill", "Carter",     Value
                        "123 Main St")
           );
```

Finding a particular key

- Given a request to find a particular key, and given that keys often have an *order relation* defined between them, it seems silly to search through the container *as if the keys were all unrelated*.
- *Example*: Suppose we are searching for the student ID "C237". Do we really need to start at the very beginning?

Search

```
A101    B972    D192
A102    C092     ...
A125    C100
A192    C200
A204    C203
B135    C237
B193    C292
```

Finding a particular key

- Given a request to find a particular key, and given that keys often have an *order relation* defined between them, it seems silly to search through the container *as if the keys were all unrelated*.
- *Example*: Suppose we are searching for the student ID "`C237`". Do we really need to start at the very beginning?

Search

```
A101    B972    D192
A102    C092     ...
A125    C100
A192    C200
A204    C203
B135    C237
B193    C292
```

No -- the *natural order* among *keys* imposes structure on the "search problem" that lets us find a particular key much more quickly.

`compareTo(o)`

- In Java, a binary ordering relation between two objects can be expressed using the `compareTo` method:
  `int compareTo (T o);`
- `o1.compareTo(o2)` is:
  - < 0 if `o1` is "less than" `o2`
  - == 0 if `o1` is "equal to" `o2`
  - > 0 if `o1` is "greater than" `o2`
- Classes that implement the `compareTo(o)` method can implement the `Comparable<T>` interface.

Comparable<T>

- Example:

Each `Student` might be "comparable to" objects of a different class, e.g., `UCSDMember` (since faculty and staff also have ID numbers).

```
class Student implements Comparable<Student> {
  ...
  int compareTo (T other) {
    // Compare this._studentID to
    // other._studentID -- return -1, 0, or 1
    // if this._studentID is "less than",
    // "equal to", or "greater than"
    // other._studentID, respectively.
    ...
  }
}
```

Comparable<T>

- Example:

```
class Student implements Comparable<Student> {
  ...
  int compareTo (T other) {
    return _studentID.compareTo(
      other._studentID
    );
  }
}
```

In this particular case, we can just delegate to the `String.compareTo(o)` method, since `String` implements `Comparable<String>`.

Searching a sorted list

- How will defining this "ordering relation" using `Comparable<T>` help us to find a key more quickly?
- Let's consider a simpler example in which we wish to find an integer within a *sorted* list of numbers.
- We will implement a method

```
int search (int[] numbers, int targetNum,
            int startIdx, int endIdx);
```

which will search through an array of **`numbers`**, starting at the **`startIdx`** and ending at the **`endIdx`**, looking for the **`targetNum`**.

Searching a sorted list

- Consider the following example:

  `search(numbers, targetNum, startIdx, endIdx):`

  where

```
int targetNum = 79;

int startIdx = 0;
int endIdx = 15;

int[] numbers = {
  16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, 79, 87, 88
};
```

- What is the optimal search strategy given that numbers is already sorted?

Binary search

- The optimal search strategy (minimum time cost) for a list of sorted elements is *binary search*.
  - The search is *binary* because we repeatedly divide the list into 2 *pieces*.
- Search algorithm:

```
Pick a guessIdx = (startIdx + endIdx) / 2;
if (numbers[guessIdx] == targetNum) {
  return guessIdx;
} else if (numbers[guessIdx] < targetNum) {
  Search the "right half" of the list for targetNum.
} else {
  Search the "left half" of the list for targetNum.
}
```

```
16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, 79, 87, 88
```

Binary search

- Let's look for `targetNum=79`.

- Search algorithm:

```
Pick a guessIdx = (startIdx + endIdx) / 2;
if (numbers[guessIdx] == targetNum) {
  return guessIdx;
} else if (numbers[guessIdx] < targetNum) {
  Search the "right half" of the list for targetNum.
} else {
  Search the "left half" of the list for targetNum.
}
```

16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, 79, 87, 88

Binary search

- Let's look for **`targetNum=79`**.

- Search algorithm:

```
Pick a guessIdx = (startIdx + endIdx) / 2;
if (numbers[guessIdx] == targetNum) {
  return guessIdx;
} else if (numbers[guessIdx] < targetNum) {
  Search the "right half" of the list for targetNum.
} else {
  Search the "left half" of the list for targetNum.
}
```

`16, 26, 31, 40, 43, 45, 51, 55,` **`58, 67, 69, 73, 79, 87, 88`**

Binary search

- Let's look for **`targetNum=79`**.

- Search algorithm:

```
Pick a guessIdx = (startIdx + endIdx) / 2;
if (numbers[guessIdx] == targetNum) {
  return guessIdx;
} else if (numbers[guessIdx] < targetNum) {
  Search the "right half" of the list for targetNum.
} else {
  Search the "left half" of the list for targetNum.
}
```

```
16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, 79, 87, 88
```

Binary search

- Let's look for **`targetNum=79`**.

- Search algorithm:

```
Pick a guessIdx = (startIdx + endIdx) / 2;
if (numbers[guessIdx] == targetNum) {
  return guessIdx;
} else if (numbers[guessIdx] < targetNum) {
  Search the "right half" of the list for targetNum.
} else {
  Search the "left half" of the list for targetNum.
}
```

`16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73,` **`79, 87, 88`**

Binary search

- Let's look for `targetNum=79`.

- Search algorithm:

```
Pick a guessIdx = (startIdx + endIdx) / 2;
if (numbers[guessIdx] == targetNum) {
  return guessIdx;
} else if (numbers[guessIdx] < targetNum) {
  Search the "right half" of the list for targetNum.
} else {
  Search the "left half" of the list for targetNum.
}
```

16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, 79, 87, 88

Binary search

- Let’s look for **`targetNum=79`**.

- Search algorithm:

```
Pick a guessIdx = (startIdx + endIdx) / 2;
if (numbers[guessIdx] == targetNum) {
  return guessIdx;
} else if (numbers[guessIdx] < targetNum) {
  Search the “right half” of the list for targetNum.
} else {
  Search the “left half” of the list for targetNum.
}
```

16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, **79**, 87, 88

Binary search

- Let’s look for **`targetNum=79`**.

- Search algorithm:

```
Pick a guessIdx = (startIdx + endIdx) / 2;
if (numbers[guessIdx] == targetNum) {
  return guessIdx;
} else if (numbers[guessIdx] < targetNum) {
  Search the “right half” of the list for targetNum.
} else {
  Search the “left half” of the list for targetNum.
}
```

Done in 4 guesses!

`16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, 79, 87, 88`

Binary search and recursion

- Binary search is a classic example of a *recursive algorithm*:
  - The algorithm makes repeated *calls to itself* to get its work done, e.g.:
    "Search algorithm:

    ```
    ...
    Search the "right half" of the list for targetNum.
    ```
    "
  - Each *recursive call* operates on a smaller problem than the original (e.g., it searches only half the list).
  - Eventually, the algorithm operates on a trivial input size (e.g., a list of 1 element) and terminates.

Sorting and recursion

- Recall, however, that binary search requires the list to have been *already* sorted.
    - How was this accomplished?
- It turns out that the fastest sorting algorithms are implemented using *recursion*:
    - For instance, the MergeSort algorithm (next week) successively divides a list of ordered elements into two halves, sorts them separately, and then combines the results.

Data structures and recursion

- Even though a sorted list of data is useful, what happens if we want to add more data into the list? How do we *keep* the data in sorted order?
  - Using a list in these cases will be inefficient.
  - More efficient is a *tree-based* data structure.
    - Trees (tomorrow, next week) are *non-linear* data structures because each element may be adjacent to more than 2 other elements.
    - Trees are *recursive data structures* -- each "branch" of a tree forms a "tree" in itself.

Data structures and recursion

- Even *computer programs themselves* are recursive data structures -- a Java class, for example, may contain multiple methods, instance variables, and *inner classes*.
  - Each inner class may contain multiple methods, instance variables, and *inner classes*.
    - Each inner class of an inner class may contain multiple methods, instance variables, and *inner classes*.
      - Each inner class of an inner class of an inner class may contain multiple methods, instance variables, and *inner classes*.
        - ...

Compilers and recursion

- In order to convert your source code into machine language, the Java compiler compiles this *recursive data structure* (your program) using *recursive algorithms* for:
  - *Lexing* -- combining the individual ASCII symbols of code into *tokens* (or *lexemes*).
  - *Parsing* -- inferring the structure among tokens that lead to meaningful statements of code.

Recursion

Recursion

- A recursive function is a function that calls itself.
- A recursive definition is a definition that defines a concept in terms of itself.
- *Important*: The recursion has to stop at some point.
  - Otherwise you have a function that never returns, or you have a completely circular definition.
- Recursion has applications in:
  - Defining mathematical concepts
  - Specifying programming language elements
  - Defining data structures
  - Designing algorithms

Fibonacci sequence

- One of the classic examples of recursion in mathematics is the Fibonacci sequence.
- The Fibonacci sequence is one of the simplest sequences of numbers one can define recursively.

- Fibonacci sequence definition:

  (a) The 1st and 2nd Fibonacci numbers are both 1.

  (b) The *n*th Fibonacci number is the sum of the *n*-1th Fibonacci number plus the *n*-2th Fibonacci number.

Fibonacci sequence

- Fibonacci sequence:
  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- Fibonacci sequence definition:

  (a) The 1st and 2nd Fibonacci numbers are both 1.

  (b) The *n*th Fibonacci number is the sum of the *n*-1th Fibonacci number plus the *n*-2th Fibonacci number.

Fibonacci sequence

Part (a) is called the *base case* -- it defines the *simplest possible* Fibonacci number, and it prevents the sequence definition from being circular.

Part (b) is the *recursive part* -- it defines the *n*th Fibonacci number in terms of other, *smaller* Fibonacci numbers.

- Fibonacci sequence definition:

Base case

(a) The 1st and 2nd Fibonacci numbers are both 1.

Recursive part

(b) The *n*th Fibonacci number is the sum of the *n*-1th Fibonacci number plus the *n*-2th Fibonacci number.

Fibonacci sequence

- Note that the recursive part must somehow bring the definition "closer" to the base case.
  - It would be useless to have the base case if the recursive part defined the *n*th number in terms of the *n*+1th and *n*+2th number.

- Fibonacci sequence definition:

  (a) The 1st and 2nd Fibonacci numbers are both 1.

  (b) The *n*th Fibonacci number is the sum of the *n*-1th Fibonacci number plus the *n*-2th Fibonacci number.

From definition to computation

- Given this recursive definition of Fibonacci numbers, it is straightforward to compute any given Fibonacci number -- *a recursive definition often lends itself readily to being translated into code*.

- Fibonacci sequence definition:

  (a) The 1st and 2nd Fibonacci numbers are both 1.

  (b) The *n*th Fibonacci number is the sum of the *n*-1th Fibonacci number plus the *n*-2th Fibonacci number.

From definition to computation

```
// Fibonacci numbers are defined only for n >= 1
int fibonacci (int n) {
  if (n == 1 || n == 2) {
    return 1;
  } else {
    return fibonacci(n-1) + fibonacci(n-2);
  }
}
```

- Fibonacci sequence definition:

  (a) The 1st and 2nd Fibonacci numbers are both 1.

  (b) The *n*th Fibonacci number is the sum of the *n*-1th Fibonacci number plus the *n*-2th Fibonacci number.

Factorial

- Another classic recursive mathematical definition is *factorial*.

- Factorial of *n* (written *n*!):

(a) If $n = 0$, then n! is 1. Base case

(b) If n > 0, then $n! = n * (n-1)!$. Recursive part

Factorial

- Translated into code, this definition becomes:

```
// Factorial is defined only for n >= 0.
int factorial (int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n-1);
  }
}
```

- Factorial of *n* (written *n*!):

(a) If *n* = 0, then n! is 1.

(b) If n > 0, then n! = *n* * (*n*-1)!.

Factorial

- Note that, *informally*, one sometimes defines factorial of *n* as "take every number between 1 and *n* and multiply them together."
- This is closer to an *iterative* definition of factorial:

```
// Factorial is defined only for n >= 0.
int factorial (int n) {
  int product = 1;
  for (int i = 1; i <= n; i++) {
    product *= i;
  }
  return product;
}
```

Iteration versus recursion

- It turns out that, in computer programming, every function that can be computed recursively can also be computed iteratively.
- However, some algorithms and structures can be more easily *conceptualized* using recursion than iteration.
- Moreover, it is often simpler to *write code* from a recursive definition than from an iterative definition.
  - Translation of recursive definitions to code can be fully automated in some cases.
    - The most prominent example is *compilers*.

Recursive binary search

- Let's return to our example of searching through an array **`numbers`** of sorted integers for a particular **`targetNum`**.
- Search algorithm:

```
// Assume targetNum is always somewhere inside numbers
int search (int[] numbers, int targetNum, int startIdx, int endIdx) {
  int guessIdx = (startIdx + endIdx) / 2;
  if (numbers[guessIdx] == targetNum) {
    return guessIdx;
  } else if (numbers[guessIdx] < targetNum) {
    Search the "right half" of the list for targetNum.
  } else {
    Search the "left half" of the list for targetNum.
  }
}
```

Base case

Recursive part

Recursive binary search

- Let's return to our example of searching through an array **`numbers`** of sorted integers for a particular **`targetNum`**.
- Search algorithm:

```
// Assume targetNum is always somewhere inside numbers
int search (int[] numbers, int targetNum, int startIdx, int endIdx) {
  int guessIdx = (startIdx + endIdx) / 2;
  if (numbers[guessIdx] == targetNum) {
    return guessIdx;
  } else if (numbers[guessIdx] < targetNum) {
    return search(numbers, targetNum, guessIdx+1, endIdx);
  } else {
    return search(numbers, targetNum, startIdx, guessIdx-1);
  }
}
```

Base case

Recursive part

Recursive binary search

- This recursive algorithm operates by dividing the list in half many times in succession.
- Eventually, the algorithm will either "get lucky" and the "middle element" it picks will equal targetNum, or the sub-list it is searching is of size 1, and the targetNum *must* be contained in that list.
- The worst-case time-cost of the binary search algorithm is computed based on the maximum number of times the search method would be called recursively.
- Since each search operates on only half the list of its "parent call", then the worst-case asymptotic time cost on an array of n elements is $\log_2(n)$.
  - I.e., the number of times n can be divided by 2 before the result is <= 1.

Recursive structures.

Source code as a "recursive structure"

- The source code of a computer program is one of the most commonly used *recursive structures* in computer science.
- Let's look at examples of recursion that arise when a compiler examines some source code...

Recursion in compilation

- The source code of a compiler is usually generated automatically from a set of recursive definitions.
- Recursive lexical definitions are used to write the portion of the compiler's source code that can separate a stream of *symbols* (ASCII characters) into meaningful *tokens*:
  - Example: **`int x = 14;`**
    `int` is a primitive type.
    `x` is an identifier (variable name).
    `=` is an assignment.
    `14` is a constant
    `;` is a separator

Recursion in compilation

- The source code of a compiler is usually generated automatically from a set of recursive definitions.
- Recursive lexical definitions are used to write the portion of the compiler's source code that can separate a stream of *symbols* (ASCII characters) into meaningful *tokens*:
  - Example: **`int x = 14;`**
    `int` is a primitive type.
    `x` is an identifier (variable name).
    `=` is an assignment.
    `14` is a constant
    `;` is a separator

When reading each symbol 1-by-1, how does the compiler "know" that 14 is a "constant" and not, say, another variable?

Integer constants: recursive definition

- An integer constant in a Java program can be defined *iterative*ly as "a sequence of 1 or more digits (0-9)."
- However, it can also be defined recursively:
  - Integer constant definition:

    (a) A digit (0-9)

    (b) An *integer constant* followed by a digit (0-9).

Backus-Naur Form

- Recursive definitions for programming language compilation are often written in a formal language called Backus-Naur Form (BNF).
- Definitions written by BNF can then be analyzed by "compiler compilers" to generate *automatically* the source code of a compiler.
- The compiler source code is then compiled, yielding the compiler itself.
- The compiler can then analyze your source code and recognize, for example, a sequence of symbols as an *integer constant*.

Backus-Naur Form

- Written in BNF, the recursive definition of a Java integer constant might be:
```
<IntConst> := <digit> | <digit><IntConst>
<digit> := 0|1|2|3|4|5|6|7|8|9
```
- `IntConst` and `digit` are both *non-terminal symbols* in BNF.
  - Non-terminals are defined *recursively*.
- `0`, `1`, `2`, ..., `9` are terminal symbols.
  - Terminals represent the *base cases* of recursive definitions.
- The `|` symbol means "or" -- e.g., an integer constant is *either* a single digit *or* a digit followed by an integer constant.

BNF Derivations

- The character-sequences "`82`", "`162354`", and "`3`" are all integer constants according to the definition above.
- "`235X1`", "`X1`", and "`1-2`" are *not* integer constants.
- This is obvious just from visual inspection, but how does the compiler know this based off just the recursive definition?

BNF Derivations

- A string of terminal symbols *S* satisfies a BNF definition of a non-terminal symbol if you can *derive* the string from the *rules* listed in the BNF definition.
- To derive a given string:
  - Start with the non-terminal symbol (e.g., IntConst).
  - At each step, try to replace *one non-terminal symbol* in the string you have so far with *one of its definitions* (e.g., `<IntConst> := <digit>`, or `<IntConst> := <digit><IntConst>`).
- If, by applying these rules, you arrive at the string *S*, then *S* satisfies the BNF definition.

BNF Derivation

- Let's apply this algorithm and try to derive the string "`163`" from the definition of the `IntConst` non-terminal symbol:

```
                Rule (b)                            Rule (b)
<IntConst>   ==>   <digit> <IntConst>   ==>
<digit> <digit> <IntConst>   ==> Rule (a)
<digit> <digit> <digit>   ==>   163
```

```
<IntConst> := <digit> |                    Rule (a)
              <digit><IntConst>            Rule (b)

<digit> := 0|1|2|3|4|5|6|7|8|9
```

Checking for IntConst in code

- Let's translate this recursive definition of IntConst into Java code:

Base cases

```
boolean isIntConstant (String s) {
  // Is it a single digit?
  if (s.length()==1 && Character.isDigit(s.charAt(0))) {
    return true;
  // Is it a digit followed by an integer literal constant?
  } else if (s.length() > 1 && Character.isDigit(s.charAt(0)) &&
             isIntConstant(s.substring(1))) {
    return true;
  // Otherwise, it doesn't fit the definition!
  } else {
    return false;
  }
}
```

Recursive part

"Compiler compilers" create this code *automatically* from the BNF rules.

Balanced parantheses

- A more interesting example of BNF-in-action is to test whether a string S contains *balanced parentheses*:
- We can define a non-terminal symbol `BalancedParen` as:

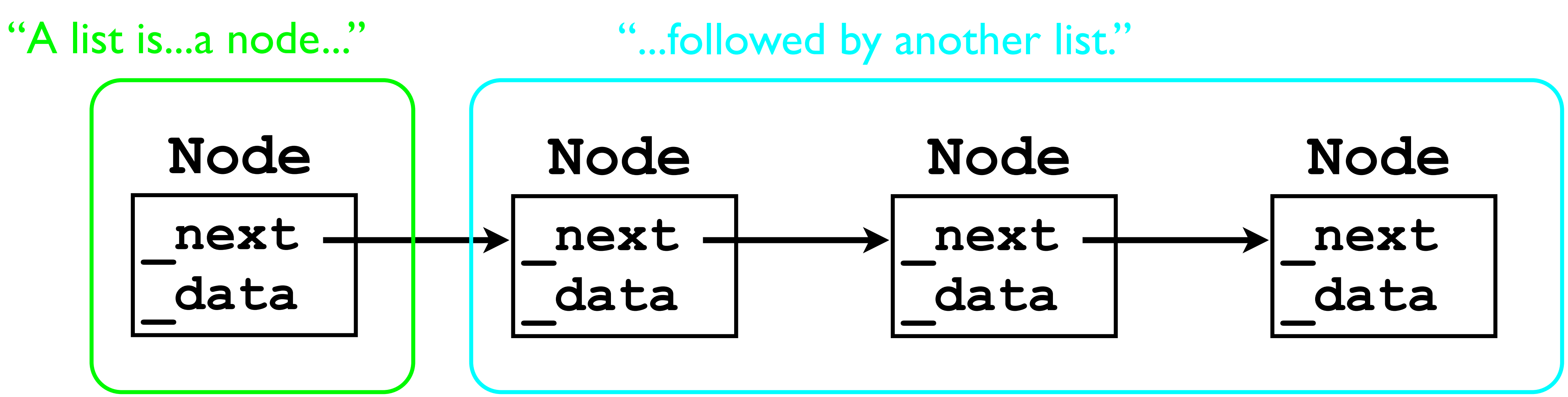
  `<BalancedParen> := ( <BalancedParen> ) | <IntConst>`

- Using this BNF definition, we can derive the strings `12`, `(53)`, `((1236))` and `((((2))))`, but we cannot derive the strings `(112` or `(((3))`.
  - I.e., there exists *no successive application* of "rules" that starts at `BalancedParen` and yields `(112` or `(((3))`.

Recursive data structures

- Let's bring this "recursive machinery" back to the world of data structures.
- The "prototypical" example of a recursive data structures is a *tree*, but in fact we can define a simple *list* recursively too:
  - A list is either empty, *or* it is a node, *or* it is a node followed by another list.

Recursive part



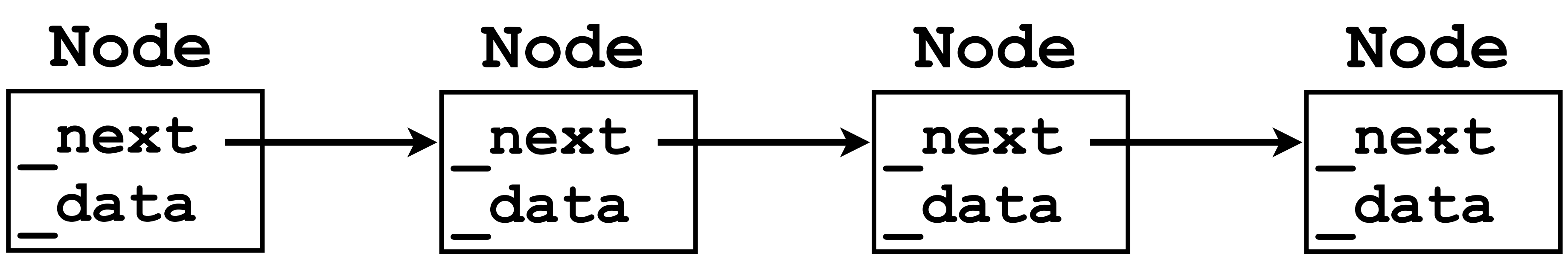
Recursive data structures

- BNF definition of *linked list* (without dummy nodes):
  -
```
<LinkedList> := nothing |
                <Node> |
                <Node> <LinkedList>
```
- By applying the rules of "what it means to be a linked list" many times in succession, we can *derive* a list of *any length* >= 0.

“Recursive” linked lists

- Derivation of linked list with 4 nodes:
  - ```
    <LinkedList>  ==>
    <Node> <LinkedList>  ==>
    <Node> <Node> <LinkedList> ==>
    <Node> <Node> <Node> <LinkedList> ==>
    <Node> <Node> <Node> <Node>
    ```

Done!


    ```

Next lecture

- Next lecture we will look at naturally recursive data structures -- trees and heaps. (A heap is a special kind of tree.)
- Tree:

  Base case

  - a tree is either a node; or
    a tree is a node with *children*, where each *child* is a *tree*. Recursive part