CSC 1052 – Algorithms & Data Structures II: Linked Lists Revisited

Professor Henry Carter
Spring 2017

Recap

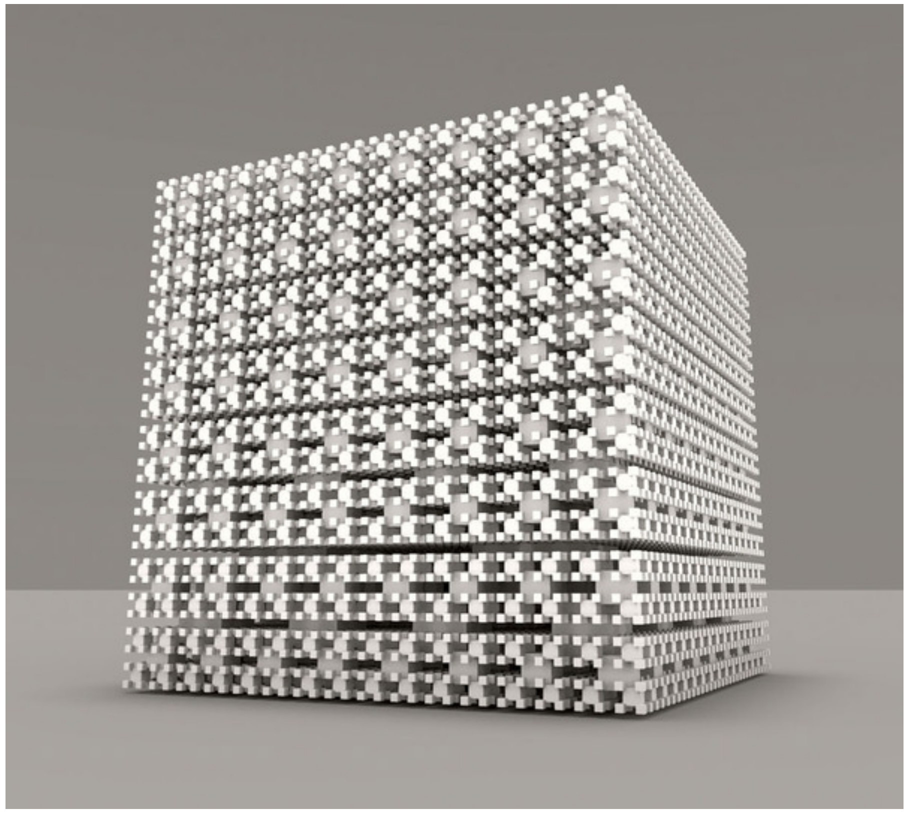
- Recursion involves defining a solution based on smaller versions of the same solution
- Three components:
  - Base case
  - Check
  - Recursive case
- Three questions are needed to verify the correctness of your algorithm
- Binary search is a very efficient search algorithm with a simple recursive definition

Exam Review

- Class average: 85
- Range: 71-100
- Bonus points: +2

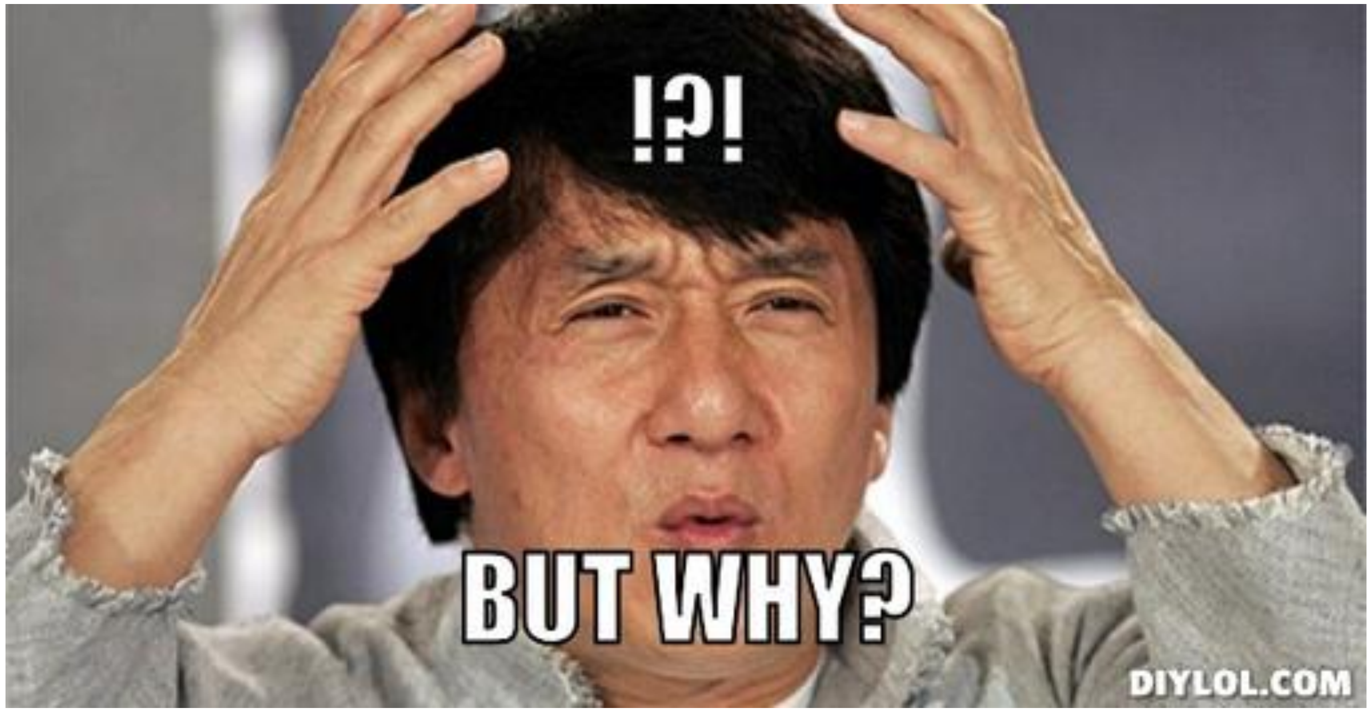
Recursive Data Structures

- Some data structures we encounter have a recursive structure
- Processing the contents of these data structures is often simplified by using recursion
- Examples:
  - Linked lists
  - Trees



Linked Lists

- Linked lists possess a recursive structure
- Evident in self-referential nature of LLNodes
- Why don't we think of arrays as recursive?



Printing a Linked List v2

- Case check?

- Base case?

- Recursive case?

Print Recursive Code

Checking our work

- Base case behavior
- Smaller-caller?
- General case behavior

Iterative vs Recursive

```
void recPrintList(LLNode<String> listRef)
{
    if (listRef != null)
    {
        System.out.println(listRef.getInfo());
        recPrintList(listRef.getLink());
    }
}

void iterPrintList(LLNode<String> listRef)
{
    while (listRef != null)
    {
        System.out.println(listRef.getInfo());
        listRef = listRef.getLink();
    }
}
```

Reverse! Reverse!

- Given a linked list, print the reversed version
- Iterative version?
- Why is this difficult?

Recurse Reverse

```
void recPrintList(LLNode<String> listRef)
{
    if (listRef != null)
    {
        recPrintList(listRef.getLink());
        System.out.println(listRef.getInfo());
    }
}
```

Modifying a Linked List

- Recall iterative versions
  - Maintain current pointer
  - Modify the node pointed to by current
- Does this map to a recursive solution?
  - Recall: how does java pass method parameters

Two approaches

- Void method that takes the list and item as parameters

- Return a pointer to the modified version of the list

Void Return Recursion

```
void recInsertEnd(String newInfo, LLNode<String> listRef)
// Adds newInfo to the end of the listRef linked list
{
  if (listRef.getLink() != null)
    recInsertEnd(newInfo, listRef.getLink());
  else
    listRef.setLink(new LLNode<String>(newInfo));
}
```

Return Pointer Code

```
LLNode<String> recInsertEnd(String newInfo, LLNode<String> listRef)
// Adds newInfo to the end of the listRef linked list
{
   if (listRef != null)
      listRef.setLink(recInsertEnd(newInfo, listRef.getLink()));
   else
      listRef = new LLNode<String>(newInfo);
   return listRef;
}
```

Return Pointer Example

Why Return Pointers?

- Simplifies recursive code
- More flexible if large changes are being made
- Uses the call stack to maintain the needed pointer state

Practice

- Implement a recursive method that counts the number of nodes in a linked list
- Implement a recursive method that deletes every occurrence of the number 5

Recap

- Data structures may have recursive structure
- Recursively processing linked lists allows for simplified code that takes advantage of this structure
- Recursion makes some tasks easier but may lead to unexpected pitfalls
    - Remember, Java is call by VALUE

Next Time...

- Dale, Joyce, Weems Chapter 3.5-3.6
  - Remember, you need to read it BEFORE you come to class!
- Check the course webpage for practice problems
- Peer Tutors
  - http://www.csc.villanova.edu/help/