Why Learn About Data Structures? (1/2)

• They are inherently interesting

• They are a core component to the field of computer science
  o fundamental to any form of programming - computation, after all, is about handling data
  o huge emphasis on data today in all fields: Data Science, Big Data,…

• They’re a big part of CS research (even first-years can start doing research with some faculty)
Why Learn About Data Structures? (2/2)

• They are central to CS16: Algorithms and Data Structures

• They will be useful in your CS15 final project (but not necessarily in Tetris)

• They are essential knowledge for tech job interviews (if that interests you)
Some Reminders

• Start early
  o These projects are *definitely not* designed to be completed in a single day
  o We can’t prevent hours lines from getting (very) long close to the deadline, so come to
    hours early

• You should always try to turn something in (even if it doesn’t compile)
  o We give lots of partial credit!
  o If your project is deemed *non-functional*, you must re-submit a working version by the
    end of the semester to get credit for the course

• You have the tools you need to do the projects on your own
  o Don’t rely on the TAs for help – use the debugging skills you learned in lab
  o Hand simulation can solve many bugs – try and follow the flow of control in your
    head/on paper
  o Eclipse is an amazing power tool – we require that you use it if you come to hours
Continuation From Last Time
Prefix, Infix, Postfix Notation for Arithmetic Expressions

- Infix, Prefix, and Postfix refer to where the operator goes relative to its operands
  - Infix: (fully parenthesized)
    \[((1 \times 2) + (3 \times 4)) - ((5 - 6) + (7 / 8))\]
  - Prefix:
    \[-\times 1 \times 2 \times 3 \times 4 \times + 5 \times 6 \times / 7 \times 8\]
  - Postfix:
    \[1 \times 2 \times 3 \times 4 \times + 5 \times 6 \times - 7 \times 8 \times / + -\]

- Graphical representation for equation:
Using Prefix, Infix, Postfix Notation

● When you type an equation into a spreadsheet, you use Infix; when you type an equation into many Hewlett-Packard calculators, you use Postfix, also known as “Reverse Polish Notation,” or “RPN,” after its inventor Polish Logician Jan Lukasiewicz (1924)

● Easier to evaluate Postfix because it has no parenthesis and evaluates in a single left-to-right pass

● Use Dijkstra’s 2-stack shunting yard algorithm to convert from user-entered Infix to easy-to-handle Postfix – compile or interpret it on the fly
Dijkstra’s infix-to-postfix Algorithm (1/2)

- 2 stack algorithm for single-pass Infix to Postfix conversion, using operator precedence
- \((a + (b * (c ^ d))) \Rightarrow a \ b \ c \ d \ ^ \ * \ +\)
- Use rule matrix to implement strategy
  - A) Push operands onto operand stack; push operators in precedence order onto the operator stack
  - B) When precedence order would be disturbed, pop operator stack until order is restored, evaluating each pair of operands popped from the operand stack and pushing the result back onto the operand stack.
  - C) “(“ starts a new substack; “)” pops until its matching “(“.

<table>
<thead>
<tr>
<th>Top of Stack</th>
<th>Incoming Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>^</td>
</tr>
<tr>
<td>^</td>
<td>*/</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>e</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
</tr>
</tbody>
</table>

Note: our Stack implementation doesn’t allow accessing the top-element without popping it; Java’s implementation has a peek method.
Dijkstra's infix-to-postfix Algorithm (2/3)

\[ a \cdot b - (c + d^e) - f \]

A) Push operands onto operand stack; push operators in precedence order onto the operator stack.

B) When precedence order would be disturbed, pop operator stack until order is restored, evaluating each pair of operands popped from the operand stack and pushing the result back onto the operand stack. Note that equal precedence displaces. At the end of the statement (marked by ; or CR) all operators are popped.

C) “(“ starts a new substack; “)” pops until its matching “(“.

<table>
<thead>
<tr>
<th>Incoming Operator</th>
<th>Top of Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>A A A A A C</td>
</tr>
<tr>
<td>^</td>
<td>A B B B B C</td>
</tr>
<tr>
<td>*/</td>
<td>A A B B B C</td>
</tr>
<tr>
<td>+-</td>
<td>A A A B B C</td>
</tr>
<tr>
<td>e</td>
<td>A A A A A E</td>
</tr>
</tbody>
</table>
Dijkstra’s infix-to-postfix Algorithm (2/2)

(a + (b * (c ^ d))) \Rightarrow a \ b \ c \ d \ ^ \ * \ +

<table>
<thead>
<tr>
<th>Operator Stack</th>
<th>Top of Stack</th>
<th>Incoming Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a + (b * (c ^ d)))</td>
<td>(</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>)</td>
<td>C</td>
</tr>
</tbody>
</table>

Stack diagram:

- Operand Stack:
  - a
  - b
  - c
  - d
- Operator Stack:
  - +
  - *
  - ^
- Top of Stack:
  - (, ^, *, +)

Precedence Checker:

- 8 - 4 * 16
- -64

Andries van Dam © 2015 11/12/15
Challenge Questions Solutions

• Q: How would you print the elements of a Binary Search Tree in increasing order?

• A: You would traverse the BST in-order

• Q: How would you find the ‘successor’ (i.e., next greatest number) of a node in a Binary Search Tree?

• A: The pseudo-code for the solution is as follows:

```python
if node.hasRight():
    node = node.right()
while(node.hasLeft()):
    node = node.left()
return node
```
Hashing

Sets and Maps
So Far …

- Covered a variety of Abstract Data Types (ADTs) which store a collection of objects (Stacks, Queues, Lists, Trees), and a variety of ways to implement them (arrays, linked lists and trees of nodes)

- Now cover another ADT which stores a collection, called a Set
Introducing… Sets

- A set is a collection of unique, **unordered** elements
  - no duplicates
  - A == {2,3,5} == {5,3,2}
  - A, B can be elements or sets

- Basic Set Operations:
  - Add element to set
  - Remove element from set
  - Merge two sets together (Union)
    - Ex: all CS15 students and CS17 students
  - Get elements in two sets which overlap (intersection)
    - Ex: all CS15 students who are freshmen
Set Abstract Data Type (1/2)

- Sets can be implemented using arrays, lists, hashing (slide 28), etc.
- No indices, no random access
- Useful for:
  - checking if elements of one collection are also a part of another collection (e.g., finding all students in CS15 who are also taking APMA1650)
  - keeping track of objects which meet some criteria (e.g., use set to check an unordered array for duplicates by iterating over array, first checking if current element exists in set)
    - if YES, add to another array which holds the duplicates for further processing
    - if NO: add it to the set
Set Abstract Data Type (2/2)

- Because there is no order/index, Sets can be implemented more efficiently than Lists and the other ADTs we have shown so far.
- Java has an implementation specialized for set operations, `java.util.HashSet<Type>`.
HashSet Methods (1/2)

//The constructor, returns a new HashSet capable of holding elements of
//type Type
public HashSet<Type> HashSet<Type>(){

//adds element e to the HashSet, if not already present (returns false if
//element is already present)
public boolean add(Type e)

//returns true if this set contains the specified element
//Note on parameter type: Java accepts any Object, but you
//should supply the same type as you are adding
public boolean contains(Object o)
HashSet Methods (2/2)

//removes all of the elements from this set
public void clear()

//returns true if this set contains no elements
public boolean isEmpty()

//removes the specified element from this set if it is present
//Note on parameter type: Java accepts any Object, but you should supply the same type as you are adding
public boolean remove(Object o)

//returns the number of elements in this set
public int size()

//see JavaDocs for more methods
Iteration over a HashSet

- You can also iterate over elements stored in a `HashSet` by using an enhanced `for` loop.
  - as it is a set, there is no guaranteed order of elements over the iteration

```java
HashSet<String> strings = new HashSet<String>();

//elided adding elements to the set

for (String s:strings) {
    System.out.println(s);  //prints all Strings in HashSet
}
```
HashSet Example

//somewhere in your app
HashSet<String> springCourses = new HashSet<String>();
springCourses.add("BIOL0200");
springCourses.add("ECON0110");
//elided adding rest of Banner

//in another part of your program
if (springCourses.contains("CS0160"){
  System.out.println("I can take cs16 next semester!");
}
//elided checking for other classes

As we will see, each such check for set membership takes just $O(1)$! i.e., no actual searching
Introducing… Maps (1/3)

- Maps are used to store (key, value) pairs, so a key is used to lookup its corresponding value
- (Word, Definition) in a dictionary
- (Brown ID, Person), in banner
- (Name, Phone #) in a contacts list
- (Identifier, Memory address) in compiler – called symbol table
Introducing… Maps (2/3)

- Java provides `java.util.HashMap<K,V>` class
- Often called a hash table
- Other structures that provide maps include `TreeMap`, `Hashtable`, `LinkedHashMap`, and more
  - each has its own advantages and drawbacks
  - we will focus on `HashMap`
- `HashMaps` have **constant-time** insert, removal, and search! – explained shortly
HashMap Syntax

- Like other ADTs, need to specify type of elements it holds
- This time need to specify type of both key AND value
- The Key and Value can be instances of any class

```java
new HashMap<KeyClass, ValueClass>();
```
HashMap Syntax

- If we wanted to map an Integer to its String representation
  ```java
  HashMap<Integer, String> intTable = new HashMap<Integer, String>();
  ```

- If we wanted to map a TA to his/her Birthday
  ```java
  HashMap<CS15TA, Date> birthdayTable = new HashMap<CS15TA, Date>();
  ```

- In all cases, both key and value types must resolve to a type(e.g. class, interface)

- Note: Can’t use `<int, boolean>` because both `int` and `boolean` are primitives, not classes
  - cannot use a primitive type as a generic, so use a built-in class that is equivalent to that primitive (wrapper)

- Instead use `<Integer, Boolean>`
java.util.HashMap Methods (1/2)

//K refers to the type of Key, V to type of value.
//adds the specified key, value pair to the table
public V put(K key, V value)

//returns the value to which the specified key is mapped, or null
//if the map contains no mapping for the key
//Note on parameter type: Java accepts any Object, but you should
//supply the same type as the key
public V get(Object key)

//returns the number of keys in this hashtable
public int size()
java.util.HashMap Methods (2/2)

//Note on parameter type: Java accepts any Object, but you should supply the same type as either the key or the value

//tests if the specified object is a key in this hashtable
public boolean containsKey(Object key)

//returns true if the hashtable maps at least one key to this value
public boolean containsValue(Object Value)

//removes the key and its corresponding value from the hashtable
//returns value which the key mapped to or null if key had no mapping
public V remove(Object key)

//More methods in JavaDocs
Finding out your friends’ logins (1/4)

- Given an array of CS students who have the properties “csLogin” and “real name”, how might you efficiently find out your friends’ logins?

- Givens
  - String[] _friends, an array of your 30 friends’ names
  - CSStudent[] _students, an array of students with a “csLogin” and a “realname”
Finding out your friends’ logins (2/4)

- Old Approach:
  ```java
  for (int i=0; i < _friends.length; i++) { //for all friends
    for (int j=0; j < _students.length; j++) { //for all students
      if (_friends[i].equals(_students[j].getName())){
        String login = _students[j].getLogin();
        System.out.println(_friends[i] + "'s login is " + login + "!");
      }
    }
  }
  ```

- Note: Use `String` class’ `equals()` method because `==` checks for equality of reference, not of content
- This is $O(n^2)$ – far from optimal
Finding out your friends’ logins (3/4)

- An approach using a HashMap:
  - Key is name
  - Value is login
  - Use name to look up login!
Finding out your friends’ logins (4/4)

- Using a **HashMap**

```java
HashMap<String, String> myTable = new HashMap<String, String>();
for (CSStudent student : _students){ //same array of students
    myTable.put(student.getName(), student.getLogin()); //build HashMap
}
for (String friendName : _friends){ //same array of friends
    String login = myTable.get(friendName); //look up friend’s login
    if (login == null){
        System.out.println("No login found for “ + friendName);
        continue;
    }
    System.out.println(friendName + “’s login is “ + login + “!");
}

- What’s the runtime now?
- **O(n)** – because each insert and search is O(1); much better!
```
Counting frequency in an Array (1/4)

● How many times does a given word show up in a given string?

● Givens
  o String[] _book, an array of Strings containing many words
  o String _searchTerm, the String you’re looking for
Counting frequency in an Array (2/4)

```java
int wordCounter = 0; // frequency of single term
for (String word : _book){
    if (word.equals(_searchTerm)){
        wordCounter++;
    }
}
System.out.println(_searchTerm + " appears " + wordCounter + " times");
```
Counting frequency in an Array (3/4)

- When tracking one word, code is simple
- But what if we wanted to keep track of 5 words? 100?
- Should we make instance variables to count the frequency of each word? For each of the terms in the book?
- If for each term, should we iterate through the _book for each of the search terms? Sounds like $O(n^2)$...
Counting frequency in an Array (4/4)

HashMap<String, Integer> countMap = new HashMap<String, Integer>();

//removes currWord, increments count, then puts currWord back with updated count
for (String currWord : _book){
    if (countMap.containsKey(currWord)){
        Integer count = countMap.get(currWord);
        countMap.remove(currWord);
        count++;
        countMap.put(currWord, count);
    } else{
        //First time seeing word
        countMap.put(currWord, 1);
    }
}

//separate method: _searchTerms is an array of Strings we’re counting
for (String word : _searchTerms){
    Integer freq = countMap.get(word);
    if (freq == null){
        freq = 0;
    }
    System.out.println(word + " shows up " + freq + " times!");
}

Despite increase in search terms, still O(n)
Map Implementation (1/5)

- How do we implement a Map with constant-time insertion, removal, and search?
- In essence, we are searching through a data structure for the value associated with the key
  - similar to the searching problem we have been trying to optimize
- Data structures we have so far:
  - Runtime to search in an unsorted array is $O(n)$
  - To search in a sorted array using binary search is $O(\log n)$
  - Using a binary search tree, search is also $O(\log n)$, but we have faster insertion and removal
  - Can we do better than a binary search tree?
Map Implementation (2/5)

- How about a ternary search tree (each node has at most 3 children)?
  - \(O(\log_3 N)\)
- Or a 10-way tree with \(O(\log_{10} N)\)
- Let’s try the runtime for a search with 1,000,000 nodes
  - \(\log_{10} 1,000,000 = 6\)
  - \(\log_2 1,000,000 < 20\), so shallower but broader tree
- Analysis: the logs are not sufficiently different and the comparison (basically an n-way nested if-else-if) is far more time consuming, hence not worth it
Map Implementation (3/5)

- Try a radically different approach, using an array
- What if we could directly use the key as an index to access the appropriate spot in the array?
- Remember: digits, alphanumerics, symbols, even control characters are all stored as bit strings—“it’s bits all the way down…”
  - see ASCII table
  - bit strings can be interpreted as numbers in binary that can be used to index into an array
Map Implementation (4/5)

- But creating an array to look up CS15 students (value) based on some ID number (key) would be a *tremendous* waste of space
  - If ID number is one letter followed by five digits (e.g., D00011), there are $26 \times 10^5$ combinations!
  - do not want to allocate 2,600,000 words for no more than 350 students
  - (1 word = 4 bytes)
  - array would be terribly sparse…

- What about using social security number?
  - would need to allocate $10^9$ words, about 4 gigabytes, for no more than 350 students! And think about arbitrary names <30 chars, need $26^{30}$!!
Map Implementation (5/5)

- Thus, two major problems:
  - How can we deal with arbitrarily long keys, both numeric and alphanumeric?
  - How can we build a small, dense (i.e., space-efficient) array that we can index into to find keys and values?
- Impossible?
- No, we approximate
Hashing

- How do we approximate?
  - We use Hashing
  - Hashing refers to computing an array index from an arbitrarily large key using a hash function
  - Hash function takes in key and returns index in array

- Index leads to a simple value or an entire object
- Therefore, a two-step process:
  - hash to create index, use index to get value
Hashing

- Array used in hashing typically holds several hundred to several thousand entries; size typically a prime (e.g., 1051)
  - array of links to instances of the class TA

Hash('Miranda')=0
Hash('Marley')=1
Hash('Hans')=2
Hash('Nick')=4
Hash Functions (1/2)

- An example of a hash function for alphanumeric keys
  - ASCII is a bit representation that lets us represent all alphanumeric symbols as integers
  - Take each character in key, convert to integer, sum integers - sum is index
  - But what if index is greater than array size?
  - Use mod, i.e. (index % arrayLength) to ensure final index is in bounds
Hash Functions (2/2)

- A better hash function
  - take a string, chop it into sections of 4 letters each, then take value of 32 bits that make up each 4-letter section and XOR them together, then % that result by table size

- Almost any reasonable function that uses all bits will do, so choose a fast one, and one that distributes more or less uniformly (randomly) in the array to minimize holes!
Collisions (1/2)

- If we have 6,000 Brown student names that we are mapping to Banner IDs using an array of size 1051, clearly, we are going to get “collisions” where different keys will hash to the same index.

- Does that kill the idea? No!

- Instead of having an array of type Value, we instead have each entry in the array be a _head pointer to an overflow “bucket” for all keys that hash to that index. The bucket can be, e.g., our perennial favorite, the unsorted singly linked list, or an array, whatever…

- So, if we get a collision, the linked list will hold all values with keys associated to that bucket.
Collisions (2/2)

- Since collisions are frequent, for methods like `get(key)` and `remove(key)`, HashMap will have to iterate through all items in the hashed bucket to `get` or `remove` the right object.

- This is $O(k)$, where $k$ is the length of a bucket – it will be small, so brute force search is fine.

- The best hash functions minimize collisions.

- Java has its own efficient hash function, covered in CS16.

- A way to think about hashing: a fast, large initial division (e.g., 1051-way), followed by a brute force search over a small bucket – even bucket size 100 is fast!
HashMap Pseudocode

table = array of lists of some size
h = some hash function

public put(K key, V val):
    int index = hash(key)
    table[index].addFirst(key, val)

O(1), if h() runs in O(1) time

public V get(K key):
    index = hash(key)
    for (k, v) in table[index]:
        if k == key:
            return v
    return null  //key not found

Runs in O(k) time, where k is size of bucket, usually small

Note: LinkedLists only hold one element per node, so in actual code, you would need to make a class to hold the key and the value
HashMaps… efficiency for free?

- Not quite
- While `put()` and `get()` methods run in $O(1)$ time, each takes more time than inserting at the end of a queue, for example
- A bit more memory expensive (array + buckets)
- Inefficient when many collisions occur (array too small)
- But it is likely the best solution overall, if you don’t need order
- No support for ordering
  - (key, value) pairs are not stored in any logical order
Announcements

• Tetris help session **today 6-8pm in Metcalf AUD**

• Reminder: Homework 3 is due 2pm on Sunday
  o Use the hand-in script (as always, no *emailed* hand-ins will be accepted on Sunday)

• We will be running an optional help session on hand-simulation on Saturday
  o Time and location TBA