Data Structures 3:
Sets, Maps and Hashing
Review of Binary Search Trees
BST Class (1/4)

- What do BSTs know how to do?
  - much the same as sorted linked lists: insert, remove, size, empty
  - BSTs also have their own search method – a bit more complicated than simply iterating through its nodes

- What would an implementation of a BST class look like…
  - in addition to data, left, and right child pointers, we’ll add a parent “back” pointer for ease of implementation (for the remove method – analogous to the previous pointer in doubly-linked lists!)
  - you’ll learn more about implementing data structures in CS200!
Nodes, data items, and keys

- **item** is a composite that can contain many properties,
- one of which is a **key** that **Nodes** are sorted by (here, ISBN #)

**BinarySearchTree**

- **root**

**Node<Book>**

- **item**
- **parent**
- **right**
- **left**

**null**

**Book**

- isbn = 9783245206
- pubDate = ...
- title = ...
- author = ...

**Node<Book>**

- ...

**Node<Book>**

- ...

Andries van Dam □ 2023 11/14/23
Comparable Book Class

- `compareBooks` is defined so we can easily compare the ISBN number of 2 books
  - Returns number that is <0, ==0 or >0, depending on the ISBN numbers
    • <0 if stored `this.isbn < toCompare`
    • ==0 if `this.isbn == toCompare`
    • >0 if `this.isbn > toCompare`

A more general Comparable interface with a compare method to compare any two objects is in the appendix of last lecture

```java
public class Book {
    // variable declarations, e.g., isbn, elided
    public Book(String author, String title, int isbn) {
        // variable initializations elided
    }

    public int getISBN() {
        return this.isbn;
    }

    // other methods elided

    // compare isbn of book passed in to stored one
    public int compareBooks(Book toCompare) {
        return (this.isbn - toCompare.getISBN());
    }
}
```
BST Class (2/4)

- Our **BinarySearchTree** stores objects of type **Book**, meaning we will be able to use all methods **Book** has within our BST

  ```java
  public class BinarySearchTree<Book> {
      private Node<Book> root;
      public BinarySearchTree(Book item) {
          //Root of the tree
          this.root = new Node(item, null);
      }
      // other methods shown next slide
  }
  ```

*In our example, we use Book as Type*

*If you’d like to see an example of a BST using a generic type that works for more than just books, check out last lecture slide 88*

*We’ll go over what Node is in a few slides ☺*
public class BinarySearchTree<Book> {

    private Node<Book> root;

    public BinarySearchTree(Book item) {
        // Root of the tree
        this.root = new Node(item, null);
    }

    public void insert(Book newItem) {
        // ...
    }

    public void remove(Book itemToRemove) {
        // ...
    }

    public Node<Book> search(Book itemToFind) {
        // ...
    }

    public int size() {
        // ...
    }

} // end of class

// class continued
public void remove(Book itemToRemove) {
    // ...
}

public Node<Book> search(Book itemToFind) {
    // ...
}

public int size() {
    // ...
}

} // end of class
Our implementations of **LinkedLists**, **Stacks**, and **Queues** are “smart” data structures that chain “dumb” nodes together.

- the lists did all the work by maintaining *previous* and *current* pointers and did the operations to search for, insert, and remove information – thus, nodes were essentially data containers

Now we will use a “dumb” tree with “smart” nodes that will delegate using **recursion**.

- tree will delegate action (such as searching, inserting, etc.) to its root, which will then delegate to its appropriate child, and so on
- creates specialized **Node** class that stores its data item, parent, and children, and can perform operations such as *insert* and *remove*
“Smart” Node includes the following methods:

```java
// pass in entire data item, containing key and returns that item if present
public Node<Book> search(Book itemToFind);

// pass in entire data item, containing key and inserts into the tree
public Node<Book> insert(Book newItem);

/* deletes Node pointing to itemToRemove, which contains key; removing Node
also will remove the matched data item instance (here, a Book) unless there’s
another reference to it */
public Node<Type> remove(Book itemToRemove);
```

Plus setters and getters of instance variables, defined in the next slides...
**BST: Node Class (2/3)**

- **Nodes** have a maximum of two non-null children that hold data
  - four instance variables: `item`, `parent`, `left`, and `right`, with each having a `get` and `set` method
  - `item` represents the data item that **Node** stores. It also contains the key attribute that **Nodes** are sorted by – we’ll make a **Tree** that stores **Books**
  - `parent` represents the direct parent (another **Node**) of **Node**—only used in `remove` method
  - `left` represents **Node**’s left child and contains a subtree, all of whose data items are **less** than **Node**’s data item
  - `right` represents **Node**’s right child and contains a subtree, all of whose data items are **greater** than **Node**’s data item
  - arbitrarily select which child should contain item **equal** to **Node**’s data item
public class Node<Book> {
    private Book item;
    private Book parent;
    private Node<Book> left;
    private Node<Book> right;
    public Node(Book myItem, Node<Book> parent) {
        this.item = myItem;
        this.parent = parent;
        // child ptrs null for leaf nodes; set for internal nodes when child is created
        this.left = null;
        this.right = null;
    }
    // will define other methods in next slides...
}

Smart Node Approach

- **BinarySearchTree** is “dumb,” so it delegates to root, which in turn will delegate recursively to its left or right child, as appropriate

  ```java
  // search method for entire BinarySearchTree:
  public Node<Book> search(itemToFind) {
    return this.root.search(itemToFind);
  }
  ```

- Smart node approach makes our code clean, simple and elegant
  - non-recursive method is much messier, involving explicit bookkeeping of which node in the tree we are currently processing
    - we used the non-recursive method for sorted linked lists, but trees are more complicated, and recursion is easier – a tree is composed of subtrees!
Let’s Search a BST

For a step-by-step walkthrough of this algorithm, see slide 100
TopHat Question

What's the runtime of (recursive) search in a BST and why?

A. $O(n)$ – because you only iterate once
B. $O(2n)$ – because you go visit both the left and right subtrees
C. $O(n/2)$ – because you incorporate the idea of “bisection” to eliminate half the number of nodes to search at each recursion
D. $O(\log_2 n)$ - because you incorporate the idea of “bisection” to eliminate half the number of nodes to search at each recursion
E. $O(n^2)$ – because recursion makes your runtime quadratic
Searching a BST Recursively Is $O(\log_2 N)$

- Search path: start with root $M$ and choose path to $I$ (for a reasonably balanced tree, $M$ will be more or less “in the middle,” and left and right subtrees will be roughly the same size)
  - structurally, the height of a reasonably balanced tree with $n$ nodes is about $\log_2 n$
  - at most, we visit each level of the tree once
  - so, runtime performance of searching is $O(\log_2 N)$ as long as tree is reasonably balanced, which will be true if entry order is reasonably random
  - $O(\log_2 N)$ is much less than $N$, this is thus much more efficient!

Andries van Dam 2023 11/14/23
Searching a BST Recursively

```java
public Node<Book> search(Book itemToFind) {
    //if itemToFind is the thing we’re searching for
    if(this.item.compareBooks(itemToFind) == 0) {
        return this.item;
    //if item > itemToFind, can only be in left tree
} else if(this.item.compareBooks(itemToFind) > 0) {
    if(this.left != null) {
        return this.left.search(itemToFind);
    }
    //if item < itemToFind, can only be in right tree
} else if (this.right != null) {
    return this.right.search(itemToFind);
}
//Only get here if itemToFind isn’t in tree, otherwise would’ve returned sooner
return null;
}
```
Let’s Add to a BST (1/3)

For a step-by-step walkthrough of this algorithm, see slide 109 of last lecture
Let’s Add to a BST (2/3)

For a step-by-step walkthrough of this algorithm, see slide 112
Let’s Add to a BST (3/3)

For a step-by-step walkthrough of this algorithm, see slide 112
Insertion into a BST

- Search BST starting at root until we find where the data to insert belongs
  - insert data when we reach a Node whose appropriate L or R child is null
- That Node makes a new Node, sets the new Node’s data to the data to insert, and sets child reference to this new Node
- Runtime is $O(\log_2 N)$, yay!
  - $O(\log_2 N)$ to search the nearly balanced tree to find the place to insert
  - constant time operations to make new Node and link it in
Insertion Code in BST

- Again, we use a “Smart Node” approach and delegate

```java
// Tree’s insert delegates to root
public Node<Book> insert(Book newItem) {
    // if tree is empty, make first node. No traversal necessary!
    if (this.root == null) {
        this.root = new Node(newItem, null); // root’s parent is null
        return this.root;
    }
    else {
        // delegate to Node’s insert() method
        return this.root.insert(newItem);
    }
}
```
Insertion Code in Node

```java
public Node<Book> insert(Book newItem) {
    if (this.item.compareBooks(newItem) > 0) { //newItem should be in left subtree
        if(this.left == null) { //left child is null – we’ve found the place to insert!
            this.left = new Node(newItem, this);
            return this.left;
        } else { //keep traversing down tree
            return this.left.insert(newItem);
        }
    } else { //newItem should be in right subtree
        if(this.right == null) { //right child is null—we’ve found the place to insert!
            this.right = new Node(newItem, this);
            return this.right;
        } else { //keep traversing down tree
            return this.right.insert(newItem);
        }
    }
}
```

Reference to the new Node is passed up the tree so it can be returned by the tree
Notes on Trees (1/2)

- Different insertion order of nodes results in different trees
  - if you insert a node referencing data value of 18 into empty tree, that node will become root
  - if you then insert a node referencing data value of 12, it will become left child of root
  - however, if you insert node referencing 12 into an empty tree, it will become root
  - then, if you insert one referencing 18, that node will become right child of root
  - even with same nodes, different insertion order makes different trees!
  - on average, for reasonably random (unsorted) arrival order, trees will look similar in depth so order doesn’t play a major role in runtime
Notes on Trees (2/2)

- When searching for a value, reaching another value that is greater than the one being searched for does not mean that the value being searched for is not present in tree (whereas it does in linked lists!)
  - it may well still be contained in left subtree of node of greater value that has just been encountered
  - thus, where you might have given up in linked lists, you can’t give up here until you reach a leaf (but depth is roughly \( \log_2 N \) for a nearly balanced tree, which is much smaller than \( N/2! \))
Preorder Traversal of BST

- **Preorder traversal**
  - “pre-order” because self is visited before (“pre-”) visiting children
  - again, use recursion!

```java
public void preOrder() {
    //Check for null children elided
    System.out.println(curr.item);
    this.left.preOrder();
    this.right.preOrder();
}
```

MHBAJILPOYS

Preorder traversal!
Postorder Traversal of BST

- **Postorder** traversal
  - “post-order” because self is visited after (“post-”) visiting children
  - again, use recursion!

```java
public void postOrder() {
    // Check for null children elided
    this.left.postOrder();
    this.right.postOrder();
    System.out.println(curr.item);
}
```

---

**ABILJHOSYPM**

Postorder traversal!
Inorder Traversal of BST

- **Inorder** traversal
  - “in-order” because self is visited between ("in-“) visiting children
  - again, use recursion!

```java
public void inOrder() {
    //Check for null children elided
    this.left.inOrder();
    System.out.println(curr.item);
    this.right.inOrder();
}
```

To learn more about the exciting world of trees, take CS200 (CSCI0200): **Program Design with Data Structures and Algorithms**!
Tree Runtime

- Binary Search Tree has a search of $O(\log_2 n)$ runtime, can we make it faster?
- Could make a ternary tree! (each node has at least 3 children)
  - $O(\log_3 n)$ runtime
- Or a 10-way tree with $O(\log_{10} n)$ runtime
- 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
- Furthermore, binary tree makes it easy to produce an ordered list
Prefix, Infix, Postfix Notation for Arithmetic Expressions (1/2)

- 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 parentheses 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
Prefix, Infix, Postfix Notation for Arithmetic Expressions (2/2)

- 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:
    
    \[- + * 1 2 * 3 4 + - 5 6 / 7 8\]
  - Postfix:
    
    \[1 2 * 3 4 * + 5 6 - 7 8 / + -\]

- Graphical representation for equation:
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.
  - 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 it’s matching "("
Dijkstra’s Infix-to-Postfix Algorithm (2/2)

\[(a + (b \times (c \wedge d))) \Rightarrow a \ b \ c \ d \ ^\wedge \times \ +\]

Operand Stack

Operator Stack

Precedence Checker

Incoming Operator Stack

\[(\quad ^\wedge \quad */ \quad +-)\]

\[
\begin{array}{cccccc}
^\wedge & A & B & B & B & C \\
*/ & A & A & B & B & C \\
\end{array}
\]
Outline

• Trees, Continued
• Sets
• Maps
• Extra Material: Preview of CS200
Sets and Maps
Introducing... Sets

• A set is a collection of unique, unordered elements
  o no duplicates
  o A == {2,3,5} == {5,3,2}
  o A, B can be single elements or sets of multiple elements

• Basic operations of the Set data structure:
  o add element to set
  o remove element from set
  o check if element is in set
  o Union: merge two sets together
    ▪ ex: Union set contains students who are CS15 students or graduate students (or both – inclusive or)
  o Intersection: Intersection set contains only elements in two sets that are in both
    ▪ ex: students who are both CS15 students and graduate students
Set Data Structure (1/2)

- Sets can be implemented using arrays, lists, hashing (coming up), 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 ECON0100).
    - Since there is no explicit intersection operator in Java, we must loop through the elements of the smaller set, and check membership in the larger set
  - preventing array from storing duplicates by checking an element to be inserted against a set of previously encountered names:
    - if already in the set, it is a duplicate, if not, enter it into array and set.
    - efficiency of checking if an element is in a set \(O(1)\) vs. efficiency of searching the array \(O(N)\)
Set Data Structure (2/2)

• Because there is no order/index, Sets can be implemented differently (and far simpler) than Lists and other data structures we have shown so far.

• Java has a class `java.util.HashSet<Type>` specialized for set operations. This class implements the `Set` interface and is backed by a Hash Table (later).
HashSet Methods (1/2)

/*Constructor returns new HashSet capable of holding elements of type Type. *Java will let us create non-homogeneous sets, but we rarely want this, so *specify use the generic Type to enforce homogeneity */
public HashSet<Type>()

/*adds element e to 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 since the elements of *your set could be any object, but you should supply one of type Type *for good programming practices */
public boolean contains(Object o)
HashSet Methods (2/2)

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

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

/*removes specified element from this set if present
 *note on parameter type: Java accepts any Object since the elements of
 *your set could be any object, but you should supply one of type Type*/
public boolean remove(Object o)

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

//see JavaDocs for more methods, including set union and intersection
Iteration over a HashSet

• You can also iterate over elements stored in a HashSet by using a for-each loop.
  o as it is a set, there is no guaranteed order of processing elements

```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("CS0200"){
    System.out.println("I can take cs200 next semester!");
}
//elided checking for other classes

As we will see, each check for set membership takes just \(O(1)!\) i.e., no actual searching!
Outline

• Trees, Continued
• Sets
• Maps
• Extra Material: Preview of CS200
Introducing... Maps (1/2)

• Maps are used to store (key, value) pairs.
  o 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
• Think of a map as discrete function that maps from domain to co-domain

<table>
<thead>
<tr>
<th>Domain(HTAs)</th>
<th>Codomain(ID#)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anastasio</td>
<td>B23456835</td>
</tr>
<tr>
<td>Allie</td>
<td>B24562457</td>
</tr>
<tr>
<td>Lexi</td>
<td>B48568458</td>
</tr>
<tr>
<td>Cannon</td>
<td>B34502346</td>
</tr>
<tr>
<td>Sarah</td>
<td>B79234600</td>
</tr>
</tbody>
</table>

Example of a simple map using just an array.
Introducing… Maps (2/2)

• Java provides `java.util.HashMap<K,V>` class
• Often called a “hash table”
• Other structures that provide maps include `TreeMap`, `HashTable`, `LinkedHashMap`, and more
  o each has its own advantages and drawbacks
  o we will focus on `HashMap`
• `HashMaps` have constant time insert, removal, and search!—explained shortly
HashMap Syntax (1/2)

• Like other data structures, 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>();
```

• Only one entry for a given key - no duplicates
HashMap Syntax (2/2)

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

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

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

- Note: Can’t use `int` or `boolean` as a type because they are primitives, not classes
  - so use a built-in class that is equivalent to that primitive, `Integer` or `Boolean` respectively
java.util.HashMap Methods (1/2)

/*K refers to type of Key, V to type of value.
 *Adds specified key, value pair to the table, returns value.
 *If there already was an entry for this key, it is replaced*/
public V put(K key, V value)

/*returns value to which the specified key is mapped, or null
 *if 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 the key. *Predicate tests if specified object is a key in this hash table*/

```java
public boolean containsKey(Object key)
```

//returns true if hash table maps at least one key to this value
```java
public boolean containsValue(Object Value)
```

/*removes key and its corresponding value from hash table,
 *returns value which the key mapped to or null if key had no mapping */
```java
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
  o `String[] friends`, an array of your friends’ names
  o `CSStudent[] students`, an array of students with each entry consisting of a “csLogin” and a “real name”, accessed via accessor methods
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())) {
            //getName() elided
            String login = students[j].getLogin(); //getLogin() elided
            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)

• Better solution: use a HashMap to store students instead of an array:
  
  - **key** → name
  
  - **value** → cslogin
  
  - use name to look up cslogin!
HashMap<String, String> myTable = new HashMap<>();

for (CSStudent student : students){ //same array of students
    //getName() and getLogin() code elided
    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){ //if friend does not have login
        System.out.println("No login found for "+ friendName);
        continue;
    }

    System.out.println(friendName + "’s login is " + login + "!");
}

• Each insert and search in HashMap is only O(1)!
Map Implementation (1/4)

• How do we implement a Map with constant-time insertion, removal, and search?

• In essence, we are searching through a data structure for value associated with key
  
  o similar to searching problem we have been trying to optimize

• Searching in an array:
  
  o unsorted array is $O(n)$
  
  o sorted array is $O(\log n)$, as is tree
    ▪ remember binary partitioning of array in merge sort where tree depicting passes had depth of $\log_2 n$?; same for binary search tree?
      
  o can we do even better than $\log_2 n$?!? That would be $O(1)$!!!
    ▪ yes: with hashing, but has limitations
Map Implementation (2/4)

- Try a radically different approach, using an array
- What if we could directly use the key as an index to access appropriate spot in the array?
- Remember: digits, alphanumerics, symbols, even control characters are all stored as 8-bit bytes—"it's bits all the way down…"
  - see ASCII table
  - numbers in binary can be used to index into an array to get oct or hex equivalent
  - O(1) to find the hex or oct string in the array at a given index!!!
Map Implementation (3/4)

- But creating an array to look up CS15 students (value) based on Banner ID # (key) would be a *tremendous* waste of space
  - if ID number is the letter B followed by eight digits (e.g., B00011111), there are $10^8$ combinations!
  - do not want to allocate 100,000,000 words for no more than 400 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 400 students! And think about arbitrary names <30 chars: need $26^{30}$ !!
Map Implementation (4/4)

- 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 instances of the class HTA containing all relevant data

Hash('Allie') = 0
Hash('Cannon') = 4
Hash('Sarah') = 8
Hash('Lexi') = 10
Hash('Anastasio') = 95
Hash Functions (1/4)

• An example of a hash function for alphanumerical keys
  o ASCII is a bit representation that lets us represent all alphanumerical symbols as integers
  o take each character in key, convert to integer, sum integers—sum is index
  o but what if index is greater than array size?
  o use mod, i.e. (index % arrayLength) to ensure final index is in bounds
    ▪ think as if index is being “wrapped around”
  o Note: hash functions are non-reversible, meaning can’t get original data from output of hash function
Hash Functions (2/4)

- 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!
- 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 (exclusive OR) them together, then % (mod) that result by table size
- Will cover this more in CS200!
Hash Functions (3/4)

• We want to turn “allie masthay” into an integer index for an array of size 101
  o Group into 4 character substrings
    ▪ “alli”, “emas”, “thay”
  o Turn each character into ASCII
    ▪ 97 108 108 105 | 101 109 97 115 | 116 104 97 121
    ▪ Turn each ASCII character into binary
      o 01100001 01101100 01101100 01101001 | 01100101 01101101 01100001 01110011 | 01110100 01101000 01100001 01111001
Hash Functions (4/4)

• We want to turn “allie masthay” into an integer index for an array of size 101
  o Turn each group into one value by mashing bits together
    ▪ “alli” 01100001011011000110110001101001
    ▪ “emas” 01100101011011010110000101110011
    ▪ “thay” 01110100011010000110000101111001
    ▪ XOR the 3 groups together
      ▪ 01100001011011000110110001101001
         ^
      ▪ 01100101011011010110000101110011
         ^
      ▪ 01110100011010000110000101111001
         = 01110000011010010110110001100011
      ▪ 01110000011010010110110001100011 (binary) 1885957219 (decimal)
  o Mod by size of list to ensure it’s within the array
    ▪ Index = 1885957219 % 101 = 76
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 – $k$ will be small, so brute force search is fine.

- The best hash functions minimize collisions.

- Java has its own efficient hash function, covered in CS200.

- 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!

- Built-in Java `HashMap` deals with collisions on its own.
Under The Hood: Java HashMaps

Keys
- key1
- key2
- key3
- ...
- key5929
- key5930

Values
- value1
- value2
- value3
- ...
- value5929
- value5930

Java

Andries van Dam 2023 11/14/23 65/100
HashMap Pseudocode

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

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

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

public V get(K key):
    int index = h(key)
    /*search through (key, val) pairs
     *in bucket at table[index] */
    for each (k, v) in table[index]:
      if k == key:
        return v
    return null //if not found, return null

Indexing with hash is O(1), and buckets are usually well under 100, so linear search time is trivial, O(1)

Note: LinkedLists only hold one element per node, so in actual code, use instance of a class that holds key and value
HashMap Example (1/2)

• We have seen some powerful data structures
• Can we combine them to create even more powerful data structures?!
• YES! … but how?
• We saw how Java HashMaps leverage other data structures for the collision buckets - we can combine even more!

• Example:
  o We want to create a massive survival multiplayer game with 10,000 users.
    ▪ every player needs to be linked to their in-game properties
    ▪ knowing a player’s username, how can we get information about their properties?
    ▪ previously, we saw HashMaps that mapped Strings to Strings
    ▪ what if instead we used a simple data structure as our compound value?
    ▪ and stored a player’s properties into an ArrayList...
HashMap Example (2/2) ft. ArrayLists

• Every player has a unique username (a String), this will be our **key**
• Let’s store each player’s properties into an **ArrayList**, this will be our **value**
• So we will map **Strings** to **ArrayLists** as such: `HashMap<String,ArrayList<>>`
• Allows for storing of dynamically updated data!
HashMaps… efficiency for free?

• Not quite
• While `put()` and `get()` methods on average run in $O(1)$ time, each takes more time than inserting at the end of an `ArrayList`, 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
  o (key, value) pairs are stored in random order based on hash. The best hash is random to minimize collisions.
  o trees can answer certain types of questions far more efficiently than a random hashmap (e.g., what is the value closest to a given value)
Hash Tables vs. Trees

- Hash Tables and Trees are different data structures used for different kinds of problems

- For just searching, insert/remove, a Hash Table will be faster
  - you know the exact key to search for
  - find a student’s Banner ID given their name, key is name and value is Banner ID

- If you’re trying to solve a nearest neighbors problem, a BST will be faster
  - find 4 people closest to a 95 in the class, key is grade and value is student name
  - if you’re trying to find the min and max in an array of numbers, a BST will be faster

- Can produce an already sorted list of data items by traversing the tree
Cryptographic Hashing (1/2)

- new techniques being developed even now that build on old ways of hashing
- normal hashing used for table look-up (key → value mapping)
  - takes large string of data and compresses it into small number of bytes to index into a relatively small table of buckets (collisions expected)
- Cryptographic hashing conversely is used to map arbitrarily large text strings into unique small strings, e.g., to map entire files into a 256 byte strings, for security and authentication. The hash, also called a digest, is essentially a data fingerprint
- cryptographic hashing (SHA-1, MD4, SHA-2, MD5, SHA-256, …)
  - encoded strings are unique and nearly impossible to replicate (effectively no collisions!)
    - SHA-256 has $2^{256}$ combinations, meaning a $1/2^{256}$ chance of a collision!
  - these hash functions are extremely computationally efficient
    - somewhat more complicated, involving many more steps (but steps, e.g. xor’s, are very fast)
    - output of the hash function is ‘random’, i.e. even the smallest change produces and entirely different hash
  - hashes are nearly impossible to reverse, can’t get valuable information just by having the hash
Cryptographic Hashing (2/2)

- In SHA-256, messages up to $2^{64}$ bit (or 2.3 billion gigabytes) are transformed into hashes of 256 bits (32 bytes)
  - this means that an object 7 times the size of Facebook’s data warehouse in 2014 (300 million GB) passed to SHA-256 would produce a 32-letter string of ASCII characters


- blockchain/cryptocurrency, digital signatures, anti-virus software, version control software (Git)
  - can check if files have changed since last time we saved/opened them
  - can check if a file has been corrupted

- “Andy Van Dam” □ 0203360322c39ddbcd8175f455d6c9a0cd94e7bfcd955ad164519d5c3e9cd441
  - using SHA-256
Outline

- Trees, Continued
- Sets
- Maps
- Extra Material: Preview of CS200
Announcements

• Tetris deadlines
  o late handin: Wednesday 11/15

• Final Projects Introduced at Thursday’s Lecture!!!

• HTA Hours Friday 3-4pm (as always!) in CIT 210
  o come talk to us about which FP to do!
The Last Lecture: SRC After 15

November 16 2023
SRC @ Brown

- Student & alum-founded initiatives

- **Better World by Design**: Student-led initiative at Brown University and Rhode Island School of Design that celebrates the interdisciplinary collaboration between designers, educators, innovators, and learners.
- **AI Robotics Ethics Society (AIRES)**: Club at Brown which talks about AI Ethics
- **Design for America**: Organization which connects students to local design projects tackling health, economic, educational, or environmental challenges (chapter at Brown)
- **Impact Labs**: Connects students to careers at the intersection of technology and social good (founded by Adi Melamed and Aaron Mayer)
- **Independent Studies**: Get course credit for working on interesting self-directed research projects with faculty (Brandon’s GPTA is an example! 😊)
SRC Courses @ Brown

• SRC-adjacent courses exist in lots of departments!
• Below are some (not all) courses offered next semester!
  • **Spring 2024**
    • MCM 0230: Digital Media
    • PHIL 0403: Ethics and Politics of Data
    • HMAN 2300: Introduction to Digital Humanities
    • SRC-specific courses in CS
      • CSCI1300: User Interfaces and User Experience
      • CSCI 1800: Cybersecurity and International Relations
      • CSCI 1952B: Responsible Computer Science in Practice
      • CSCI 1952X: Contemporary Digital Policy and Politics
      • Most courses in CS @ Brown have a SRC component (format & content as appropriate)

• SRC topics may be covered in some of the following departments:
  • e.g., **Science and Technology Studies (STS), CS, Public Policy, Psychology/Cognitive Science (CLPS), History, Literary Arts, Urban Studies, Environmental Studies, Philosophy, Modern Culture and Media (MCM), independent concentrations, and more!!**
Alumni and SRC Beyond Brown (past & current projects)

- Alums working in Social Impact/Ethics & Tech: lots!
  - danah boyd: Assistant Prof @ Georgetown and Partner Researcher at Microsoft Research
    - Lectures in AVD’s + Norm Meyorwitz’s Hypermedia course
  - Daniel Kahn Gillmor: Senior Staff Technologist @ ACLU
  - Solon Barocas: Cornell Information Science & Microsoft Research (AI Ethics)
  - Sharon Lo: Product Manager (PM), Copilot + Repsonsible AI @ Github, Former PM, Ethics @ Microsoft + founded Hack@Brown!
  - Vandhana Ravi: Beeck Center for Social Impact @ Georgetown U
  - Aaron Mayer: Impact Labs
  - Tiffany Chen: Formerly Inclusive Design @ Microsoft
  - Merrie Ringel Morris (CS15 TA): Director for Human-AI Interaction Research @ Google DeepMind
  - Tanay Padhi (CS15 TA): phone-based contact tracing (Google + Apple + …)
  - Matt Lerner: "Two Screens for Every Teacher"
  - Brandon Diaz (Ex CS15 HTA, CS15 UTA): Investigated impact of generative AI on intro CS education and created GPTA app.
  - …
Thank you!!

• Thanks for listening and engaging with SRC material!
• “You have agency!” - Andy
• If you have any questions/want to learn more/want to learn about being an STA/just want to chat about STAing/UTAing/Brown please reach out!
  • Adam: adam_mroueh@brown.edu
  • Faizah: faizah_naqvi@brown.edu
  • Katie: katie_y_li@brown.edu
Extra Optional Material

A Sneak-Peak Toward CS200!
Building a Node List
How To Build A Node List

• Now that we have a building block, there are a number of methods we can implement to make a higher-level NodeList that implements Java’s List interface (like ArrayList does)
  o note: List interface is very general...

• Main addition List mandates is to support indexing into the NodeList. Let’s write one of the simpler ones:
  o get(int i) method that returns element (Type) at that index
**search** Private Helper Method

- First, define a `search` helper method to return node at a particular index

- Want to use this helper method in the class, but don’t want to expose found nodes publicly; that would violate encapsulation - make helper `private`

- If a provided index is out of bounds, return `null` (print line is an optional error message)

- Otherwise, iterate through list until node at desired index is reached and return that node

```java
public class NodeList<Type> {
    //constructor elided
    private Node<Type> search(int i) {
        if(i < 0 || i >= this.size) {
            System.out.println("Invalid index");
            return null;
        }
        Node<Type> curr = this.head;
        //for loop stops at i; pointer-chase to i
        for (int counter = 0; counter < i; counter++) {
            curr = curr.getNext();
        }
        return curr;
    }
}
```
search Private Helper Method Runtime

private Node<Type> search(int i) {
    if (i >= this.size || i < 0) { // 1 op
        System.out.println("Invalid index"); // 1 op
        return null; // 1 op
    }
    Node<Type> curr = this.head; // 1 op
    for (int counter = 0; counter < i; counter++) { // n ops
        curr = curr.getNext(); // 1 op
    }
    return curr; // 1 op
}

→ search(int i) is O(n)
Public Wrapper Method

- Write the publicly accessible wrapper code for the NodeList's `get` method
  - this shows a very common pattern of “thin wrappers” over private code

```java
//inside NodeList
public Type get(int i) {
    return this.search(i).getElement();
}
```
Reversing a Linked List
An Exercise
(common job interview question)

- Write a method that reverses the order of a MyLinkedList

```java
Node<Type> head
  Node<Type> next
    Type element

Node<Type> 2
  Node<Type> next
    Type element

Node<Type> 3
  Node<Type> next
    Type element

Node<Type> tail
  Node<Type> next
    Type element
```

Andries van Dam  2023 11/14/23
Solution A

• If list is empty or has 1 node, return list

• Otherwise, create a new list of same type as input list

• Iterate through input list, removing first element each time and adding it as first element of new list

```java
public MyLinkedList<Type> reverse(MyLinkedList<Type> toReverse) {
    if (toReverse.size() < 2) {
        return toReverse;
    }
    MyLinkedList<Type> newList = new MyLinkedList<Type>();
    int origSize = toReverse.size();
    while (newList.size() < origSize) {
        newList.addFirst(toReverse.removeFirst());
    }
    return newList;
}
```
Solution B (1/2)

• Is there a better way?

• First algorithm reversed in O(n) time
  o but it wasn’t “in-place” – (had to create a new list)
  o memory use is also O(n)

• Can write a method within MyLinkedList that reverses itself without creating new nodes
  o still O(n) but in-place and therefore more efficient
Solution B (2/2)

• Keep track of previous, current, and next node
• While current node isn’t null, iterate through nodes, resetting node pointers in reverse
• In doing so, must be careful not to delete any references further on in the list
• Finally, set the head pointer to what had been the last node (held in the prev variable)

• If the list is empty curr will be null, so the loop will never begin and head will continue to point to null

```java
public void reverse() {
    Node<Type> prev = null;
    Node<Type> curr = this.head;
    Node<Type> next = null;
    //set tail to head
    this.tail = this.head;
    this.tail = this.head;

    while (curr != null) {
        next = curr.getNext();
        curr.setNext(prev);
        prev = curr;
        curr = next;
    }
    this.head = prev;
}
```
next = null
prev = null
Solution B Walkthrough (2/15)

Andries van Dam  © 2023 11/14/23
Solution B Walkthrough (3/15)

```java
prev = null
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```

prev = null

![Diagram of linked list with nodes A, B, and C connected from left to right.](image-url)
Solution B Walkthrough (4/15)

```
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```

Andries van Dam 2023 11/14/23
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (7/15)

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (9/15)

```java
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (11/15)

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

head

A

B

C

tail

prev

curr

null

next = null

Andries van Dam  2023 11/14/23

101
Solution B Walkthrough (12/15)

```java
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```
Solution B Walkthrough (13/15)

```java
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```
Solution B Walkthrough (14/15)

```java
while (curr != null) {
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```
Solution B Walkthrough (15/15)

```
this.head = this.prev;
```
Using a Stack to Reverse a Linked List
Reverse a Linked List with a Stack (1/4)

● How can we use a Stack to reverse a Linked List?

head

● Linked List: Caden, Chloe, Gaby, Orlando

● Note: user wouldn’t see head and tail – implementation detail
Reverse a Linked List with a Stack (2/4)

- Solution:
  - while Linked List is not empty, remove from Linked List and push elements onto Stack
  - then, while Stack is not empty, pop elements from Stack and add to Linked List
Reverse a Linked List with a Stack (3/4)

while(!this.list.isEmpty()) {
    stack.push(this.list.removeFirst());
}

Stack

head
Caroline  Patrick  Charlie  Allie
tail

Andries van Dam □ 2023 11/14/23
Reverse a Linked List with a Stack (4/4)

```
while(!this.list.isEmpty()) {
    stack.push(this.list.removeFirst());
}
```

[Diagram of a stack with names: Orlando, Gaby, Chloé, Caden]
Check for Balanced Parentheses
Check for Balanced Parentheses (1/2)

- Check for balanced parentheses in a given string
- Balanced: [()()]{[()]} 
- Not balanced: [(])
Check for Balanced Parentheses (2/2)

- Go through every character, if it is a starting bracket, push it onto the stack
- If it is a closing bracket, pop from the stack
  - if stack is empty, return false
- The bracket you pop should be the opening bracket that corresponds to the closing bracket you are looking at
  - if it is not, return false
- If you get through every character and you haven’t returned false, check if stack is empty
- If it is, the brackets are balanced!
Check for Balanced Parentheses Pseudocode

for each bracket in string:
    if it is a starting bracket:
        push it onto stack
    if it is a closing bracket:
        pop from the stack
        if the popped character is not the matching opening bracket:
            return false
    if stack is empty:
        return true
Check for Balanced Parentheses Walkthrough (1/6)

for each bracket in string:

- if it is a starting bracket: push it onto stack
- if it is a closing bracket: pop from the stack
  - if the popped character is not the matching opening bracket: return false

if stack is empty return true
for each bracket in string:
    if it is a starting bracket:
        push it onto stack
    if it is a closing bracket:
        pop from the stack
        if the popped character is not the matching opening bracket:
            return false
    if stack is empty
        return true
Check for Balanced Parentheses Walkthrough (3/6)

for each bracket in string:
  if it is a starting bracket:
    push it onto stack
  if it is a closing bracket:
    pop from the stack
    if the popped character is not the matching opening bracket:
      return false
if stack is empty
  return true

Match! Keep going
for each bracket in string:
  if it is a starting bracket:
    push it onto stack
  if it is a closing bracket:
    pop from the stack
    if the popped character is not the matching opening bracket:
      return false
if stack is empty
  return true

[ ( ) ]

Match! Keep going
for each bracket in string:
   if it is a starting bracket:
       push it onto stack
   if it is a closing bracket:
       pop from the stack
          if the popped character is not the matching opening bracket:
            return false
   if stack is empty
     return true
for each bracket in string:
  if it is a starting bracket:
    push it onto stack
  if it is a closing bracket:
    pop from the stack
    if the popped character is not the matching opening bracket:
      return false
  if stack is empty
    return true

[ ( ) ]

Stack
Exercise 2 Actual Code

for (int i=0; i<parenthesesArray.length; i++) {
    // If the element at this index is either starting bracket, push onto stack
    if (parenthesesArray[i].equals("[") || parenthesesArray[i].equals("(") ) {
        myStack.push(parenthesesArray[i]);
    }
    // If the element at this index is either closing bracket, pop off of stack
    // Note use of built-in equals() method to compare Strings- returns a boolean
    if (parenthesesArray[i].equals("]") || parenthesesArray.equals(”)”) ) {
        String popped = myStack.pop();
        if (parenthesesArray[i].equals(“)”) && !popped.equals("(" ) {
            return false;
        }
        else if (parenthesesArray[i].equals("]”) && !popped.equals("[”) ) {
            return false;
        }
    }
}
if (myStack.isEmpty()) {
    return true;
}
Model TA Hours Line
TA Hours Line (1/2)

• Let’s model the TA hours line
• Because it is FIFO, we need to use a queue!
• What functionality do we need?
  o a method for students to be added to the line
  o a method for TAs to help the line until it is empty
TA Hours Line (2/2)

• Start by initializing `queue` and `ta`

• Define a method for adding to hours line
  o this can be used before hours or during hours to sign up

• Define a method for seeing a student –uses `CS15TA's help()`

• Define a method for emptying the queue
  o useful after the cutoff is set

```java
public class TAHoursLine{
    private Queue<Student> queue;
    private CS15TA ta;

    public TAHoursLine(CS15TA ta){
        this.queue = new Queue<Student>();
        this.ta = ta;
    }

    public Student addToLine(Student s){
        return this.queue.enqueue(s);
    }

    public void seeStudent(){
        this.ta.help(this.queue.dequeue());
    }

    public void holdHoursUntilCutoff(){
        while(!this.queue.isEmpty()){
            this.seeStudent();
        }
    }
}
```
Counting Frequency in an Array
Counting frequency in an Array (1/4)

• How many times does a given word show up in a given string?
• Consider a book as one long String. That’s too hard to search, so let’s chop the string into individual words using punctuation as a separator and put each word in an array

• Givens
  o String[] _book, an array of Strings, each an individual word
  o String searchTerm, the word you’re looking for
Counting frequency in an Array (2/4)

```java
public void frequency(String searchTerm) {
    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 term in the book?
  o should we iterate through 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>();
/* book is an array of words.  
 * If currWord in _book matches a search term, 
 * put currWord back with updated count. By using  
 * put(), we replace current entry in hashMap. 
 * Note use of Integer rather than int because you  
 * can’t use base types as generics */
for (String currWord : _book){
    if (countMap.containsKey(currWord)){
        Integer count = countMap.get(currWord);
        count++;
        countMap.put(currWord, count);
    } else{
        //First time seeing word
        countMap.put(currWord, 1);
    }
}

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

Despite increase in search terms, still O(n)