Dynamic Data Structures

John Ross Wallrabenstein
Recursion Review

• In the previous lecture, I asked you to find a recursive definition for the following problem:
  • Given: A base10 number $X$
  • Assume $X$ has $N$ digits
  • Output: $X$ printed vertically
Print Vertically

- Did anyone come up with a solution?
Print Vertically
Print Vertically

- What is our base case?
Print Vertically

- What is our base case?
- When X is a single digit: print X
Print Vertically

- What is our base case?
- When $X$ is a single digit: print $X$
- What is our recursive call?
Print Vertically

- What is our base case?
  - When X is a single digit: print X

- What is our recursive call?
  - When X has more than one digit:
Print Vertically

- What is our base case?
  - When X is a single digit: print X

- What is our recursive call?
  - When X has more than one digit:
    - Print the N-1 most significant digits
Print Vertically

- What is our base case?
  - When X is a single digit: print X

- What is our recursive call?
  - When X has more than one digit:
    - Print the N-1 most significant digits
    - Print the least significant digit
public class PrintVertically {

    public static void main(String[] args) {
        int x = 1234;
        System.out.println("writeVertical( \" + x + \"):");
        writeVertical(x);
    }

    public static void writeVertical(int x){

        //If X is a single digit
        if(x < 10){
            System.out.println(x);
        }

        //If X has more than a single digit
        else{
            //Print the most significant digits vertically
            writeVertical(x/10);

            //Print the least significant digit vertically
            System.out.println(x%10);
        }
    }
}
Asymptotic Notation

• When analyzing an algorithm, it is convenient to have a simple way to express the efficiency of the algorithm

• This allows us to compare the efficiency of two different algorithms
“Big-Oh” Notation

• Assumption: Our algorithm is defined in terms of functions whose domains are the set of natural numbers

• $\mathbb{N} = \{0, 1, 2, \ldots\}$
“Big-Oh” Notation

• We would like to *bound* the running time of our algorithm by a measure of the number of atomic (constant time) operations it performs.
“Big-Oh” Notation

- We say that a function $f(n)$ is $O(g(n))$ iff:
  - there exists some constant $c$ and $n_0$ such that:

$$0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0$$
“Big-Oh” Notation

Formally:

\[ f(n) = O(g(n)) \ \text{iff} \ \exists c, n_0 \in \mathbb{R}^+ \ \text{s.t.} \]

\[ 0 \leq f(n) \leq cg(n) \ \forall n \geq n_0 \]
“Big-Oh” Notation
“Big-Oh” Notation

• Example:

• This program runs in $O(n)$ time

```java
for(int i = 0; i < n; ++i){
    //Atomic Operations
}
```
“Big-Oh” Notation

- Example:
- This program runs in $O(n^2)$ time
Asymptotic Analysis

• The smaller the asymptotic bound, the faster the algorithm

• e.g. $O(n)$ is preferable to $O(n^2)$
Data Structures
Data Structures

• What are they?
Data Structures

• What are they?

• A *data structure* is an abstraction used to model a set of related data
Data Structures

• What are they?
• A data structure is an abstraction used to model a set of related data
• What data structures are you familiar with?
Data Structures

• What are they?

• A data structure is an abstraction used to model a set of related data

• What data structures are you familiar with?

• Most Common Example: Arrays
Data Structures

- What are they?
- A *data structure* is an abstraction used to model a set of related data
- What data structures are you familiar with?
  - Most Common Example: Arrays
  - Food for Thought: File Systems
Static Data Structures

• Static Data Structures

• Data Structures that do not change* over the lifetime of their use

*This usually refers to the size of the structure
Static Data Structures

• Benefits:
  • Fast Access to Individual Elements

• Drawbacks:
  • Expensive to “update”
    • e.g. Insertion, Deletion
  • Finite size known at compile time
    • e.g. Arrays
Dynamic Data Structures

• Dynamic Data Structures

• Data Structures that change over their lifetime, allowing for efficient updates
Dynamic Data Structures

• Benefits:
  • Fast Updates (Insertion, Deletion)
  • Flexible Size

• Drawbacks:
  • Slow access to individual elements
Arrays

- Recall that the size of an array must be declared at compile time
- How does this affect insertion/deletion?
Array Deletion

• When we *delete* an element from an array, we usually waste memory

• If we want to reclaim this memory, we have to create a *new* array (of size \( n-1 \)) and copy all \( n-1 \) elements

• This is inefficient!
Array Insertion

• Case: Insert an element between existing elements
  • We have to copy the entire array to a new location, leaving space for the new element
  • This is inefficient!
Array Insertion

• Case: Insert (push) an element onto the end of a full array

• We have to:
  • Allocate space for an array of size $n+1$
  • Copy $n$ elements into the new array
  • Add new element to the end
  • Deallocate memory for old array

• (For C Programmers)
Array Insertion

• Can we do better than $O(n)$ time?

• How?
Array Insertion

- Improving Insertion Efficiency

- Rule: When the array is full and a new element is to be added, proceed by:
  - Allocating space for an array of size $2n$
  - Copy original $n$ elements
  - Add new element
Array Insertion

• How does this help?

• In the *amortized* case, we now have that insertion takes $O(1)$!

• *Amortized analysis* involves taking the cost of an operation over the number of times the operation is invoked

• In this case:

• A single *push* is effectively $O(1)$
You have probably seen ArrayLists.

An ArrayList is a hybrid between a static data structure (arrays) and a fully dynamic structure.

ArrayLists use the previous optimization when inserting elements into a full array.
Linked Lists

- A linked list is a dynamic data structure composed of nodes
- Each node contains:
  - Data
  - Pointer to next node in list*

*In the case of a singly linked list
Linked Lists

- How is the order of the elements in an array determined?
  - By the index of the element

- How is the order of the elements in a linked list determined?
  - By the next pointer in each node
Linked Lists

• What are the benefits of this structure?

• Insertion:
  • Point the current node’s next to predecessor’s next
  • Point predecessor’s next to the current node

• No resizing!
Linked Lists

• What are the benefits of this structure?

• Deletion:
  • Point the previous node’s (predecessor) next to the current node’s next (successor)

• No inefficient use of memory!
Linked Lists

- What are the drawbacks of this structure?
  - Representation on disk may be fragmented
  - Why?
  - Linked Lists do not need contiguous blocks of memory
  - Thus, nodes can be stored anywhere
Dynamic Sets

• When building algorithms, sets of data must be able to change over time*

• We must be able to:
  • Insert
  • Delete
  • Test Membership

*Have we seen this phrase before? ...
Dynamic Sets

• What dynamic sets have you seen before?
  • Stacks
  • Queues
  • Linked Lists
  • Etc...

• Let’s look at a particular application of sets
Disjoint Sets

• Two sets $A$ and $B$ are said to be disjoint if their intersection is null.

• Many times, it is useful to partition a larger set into smaller, disjoint sets.

• How can we keep track (Find) of the elements and combine (Union) the smaller sets into larger sets efficiently?

• Dynamic Data Structures to the rescue!
Disjoint Sets

• A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets

• Each set is identified by a representative
Disjoint Set Operations

• We require the following operations to be supported:
  • Make-Set(x) - create a new set whose only member is x
  • Union(x,y) - unites the dynamic sets that contain elements x and y
  • Find-Set(x) - return a pointer to the representative of the set containing x
Why We Care

• Before proceeding, why should we worry about how to use dynamic data structures to handle dynamic disjoint sets?

• Common Application:
  • Determining the Connected Components of a Graph

• See Board
Union / Find

- We will focus on constructing a dynamic data structure that computes *Union* and *Find* efficiently.
Union / Find

- To support the operations Union and Find on disjoint sets, we will use a disjoint set forest of trees

- Setup:
  - Each member points only to its parent
  - The root is the representative of the set
  - (and also its own parent)
Union / Find

(a)  
(b)
To implement \( \text{Find}(x) \) and return its representative element:

- Follow parent pointers until the root is reached
- Return the root (representative)
Union

- To implement $\text{Union}(x,y)$
- Set the parent of one of the set’s representatives to the root of the other set
Consider the following scenario:

- We have $x_1, x_2, ..., x_n$ objects
- We make each $x_i$ a set with $\text{Make-Set}(x)$
- We $\text{Union}$ each $x_i$ into one large set

See Board

Does order matter?
Union / Find

- Consider the running time of \( \text{Find} \) if we only use the tools we have seen so far:
  - To \( \text{Find} \) element \( X \) in a set, if we have a linear chain of \( n \) nodes takes \( O(n) \) time
  - If we have \( n \) calls to \( \text{Find} \) our running time is then:
    \[
    \sum_{i=1}^{n} i = O(n^2)
    \]
- Can we do better?
Union / Find

- What is the source of inefficiency?
  - We are appending a larger set onto a smaller set ($x_i$)

- How can we fix this?
  - Maintain the size of each set (easy)
  - Always append the smaller set onto the larger set
  - Use path compression to reduce tree height
Union By Rank

- To always append the *smaller* set to the *larger* set, we maintain a *rank* for each set.
- The rank is an upper bound on the height of the set tree.
- The root with the smaller rank points to the root of the larger set.
Union By Rank

- See Board for Example
- Performance Analysis: $\Theta(m \log n)$
  - Where we perform $m$ operations on $n$ elements
- The proof of this bound is beyond the scope of this course
Path Compression

- To lower the number of parent nodes we have to pass through to reach the root in a $\text{Find}(x)$ operation:
  - Run $\text{Find}(x)$ once to retrieve the root
  - Run $\text{Find}(x)$ again, pointing each node along the way to root

- See Board

- Running time: $\Theta(n + f \cdot (1 + \log \frac{f}{2+\frac{f}{n}} n))$
Combining Both

- When both Union By Rank and Path Compression are used:
  - Running Time: $\theta(m\alpha(n))$
  - Here, \textit{alpha} represents the Ackermann Function, which is effectively constant
Questions